

Ice 分布式程序设计

Michi Henning
Mark Spruiell

以下人士为本文档做出了贡献

Benoit Foucher, Marc Laukien,
Matthew Newhook, Bernard Normier

马维达 译

制造商和销售商用来区分其产品的许多名称已被声明为商标。如果这些名称在本书中出现，而且 ZeroC 注意到其商标声明，则名称的首字母或所有字母会大写。

本书的作者及出版者精心制作了本书，但不提供任何类型的担保，无论是明确的还是隐含，同时也不对错误或疏漏承担任何责任。如果本书包含的信息或程序在使用时引发偶然或继起的损坏，本书作者及出版者不承担任何责任。

版权所有 © 2004 by ZeroC, Inc.
<mailto:info@zeroc.com>
<http://www.zeroc.com>

修订版 1.3.0，2004 年 3 月 1 日

本修订版文档描述的是 Ice 1.3 版。

关于中文版的意见、建议，请发送至：weida@flyingdonkey.com
<http://www.flyingdonkey.com>

Ice 源码包使用了以下第三方产品：

- Berkeley DB，由 Sleepycat Software 开发 (<http://www.sleepycat.com>)
- bzip2/libbzip2，由 Julian R. Seward 开发 (<http://sources.redhat.com/bzip2>)
- The OpenSSL Toolkit，由 OpenSSL Project 开发 (<http://www.openssl.org>)
- SSLeay，由 Eric Young 开发 (<mailto:ey@cryptsoft.com>)
- Expat，由 James Clark 开发 (<http://www.libexpat.org>)

上述各产品的授权协议，见 Ice 源码包。

附注： 文档中含有一些标注为 “XREF” 的交叉引用，它们指向的是还未写成、但将在未来加入的内容。

目录

第 1 章	引言	1
	1.1 引言	1
	1.2 The Internet Communications Engine (Ice)	3
	1.3 本书的篇章结构	4
	1.4 排字惯例	4
	1.5 源码示例	5
	1.6 联系作者	5
	1.7 Ice 支持	5
	第一部分 Ice 综述	7
第 2 章	Ice 综述	9
	2.1 本章综述	9
	2.2 Ice 架构	9
	2.3 Ice 服务	21
	2.4 Ice 架构提供的好处	23
	2.5 与 CORBA 的对比	25
第 3 章	Hello World 应用	33
	3.1 本章综述	33
	3.2 编写 Slice 定义	33
	3.3 编写使用 C++ 的 Ice 应用	34
	3.4 编写使用 Java 的 Ice 应用	41
	3.5 总结	48

第二部分 Ice 核心概念 51

第 4 章	Slice 语言	53
4.1	本章综述	53
4.2	引言	53
4.3	编译	54
4.4	源文件	57
4.5	词法规则	59
4.6	基本的 Slice 类型	62
4.7	用户定义的类型	63
4.8	接口、操作，以及异常	70
4.9	类	92
4.10	提前声明	106
4.11	模块	107
4.12	类型 ID	109
4.13	Object 上的操作	110
4.14	本地类型	111
4.15	Ice 模块	112
4.16	名字与作用域	113
4.17	元数据	117
4.18	使用 Slice 编译器	118
4.19	Slice 与 CORBA IDL 的对比	119
4.20	总结	127
第 5 章	一个简单文件系统的 Slice 定义	137
5.1	本章综述	137
5.2	文件系统应用	137
5.3	文件系统的 Slice 定义	138
5.4	完整的定义	140

第 6 章	客户端的 Slice-to-C++ 映射	143
6.1	本章综述	143
6.2	引言	143
6.3	标识符的映射	144
6.4	模块的映射	144
6.5	Ice 名字空间	145
6.6	简单内建类型的映射	146
6.7	用户定义类型的映射	146
6.8	常量的映射	150
6.9	异常的映射	151
6.10	运行时异常的映射	154
6.11	接口的映射	154
6.12	操作的映射	161
6.13	异常处理	167
6.14	类的映射	169
6.15	slice2cpp 命令行选项	183
6.16	与 CORBA C++ 映射比较	184
第 7 章	开发 C++ 文件系统客户	189
7.1	本章综述	189
7.2	C++ 客户	189
7.3	总结	194
第 8 章	客户端的 Slice-to-Java 映射	197
8.1	本章综述	197
8.2	引言	197
8.3	标识符的映射	198
8.4	模块的映射	198
8.5	Ice Package	199
8.6	简单内建类型的映射	200
8.7	用户定义类型的映射	200
8.8	常量的映射	204
8.9	异常的映射	205
8.10	运行时异常的映射	206
8.11	接口的映射	207
8.12	操作的映射	213
8.13	异常处理	219
8.14	类的映射	220
8.15	Packages	224
8.16	slice2java 命令行选项	225

第 9 章	开发 Java 文件系统客户	229
	9.1 本章综述	229
	9.2 Java 客户	229
	9.3 总结	233
第 10 章	服务器端的 Slice-to-C++ 映射	235
	10.1 本章综述	235
	10.2 引言	235
	10.3 服务器端 main 函数	236
	10.4 接口的映射	247
	10.5 参数传递	249
	10.6 引发异常	251
	10.7 对象体现	252
	10.8 总结	257
第 11 章	开发 C++ 文件系统服务器	261
	11.1 本章综述	261
	11.2 实现文件系统服务器	261
	11.3 总结	276
第 12 章	服务器端的 Slice-to-Java 映射	279
	12.1 Chapter Overview	279
	12.2 引言	279
	12.3 服务器端 main 函数	280
	12.4 接口的映射	285
	12.5 参数传递	287
	12.6 引发异常	288
	12.7 Tie 类	289
	12.8 对象体现	292
	12.9 总结	296
第 13 章	开发 Java 文件系统服务器	297
	13.1 本章综述	297
	13.2 实现文件系统服务器	297
	13.3 总结	306

第 14 章	Ice 属性与配置	307
	14.1 本章综述	307
	14.2 属性	307
	14.3 配置文件	309
	14.4 在命令行上设置属性	309
	14.5 Ice.Config 属性	310
	14.6 命令行解析与初始化	311
	14.7 Ice.ProgramName 属性	312
	14.8 在程序中使用属性	313
	14.9 总结	323
第 15 章	C++ 线程与并发	325
	15.1 本章综述	325
	15.2 引言	325
	15.3 Ice 线程模型	326
	15.4 线程库综述	326
	15.5 互斥体	327
	15.6 递归互斥体	332
	15.7 读写递归互斥体	335
	15.8 定时锁	338
	15.9 监控器	341
	15.10 线程	350
	15.11 可移植的信号处理	357
	15.12 总结	358

第三部分 高级 Ice 363

第 16 章	Ice Run Time 详解	365
	16.1 引言	365
	16.2 通信器	365
	16.3 对象适配器	369
	16.4 对象标识	375
	16.5 Ice: : Current 对象	376
	16.6 Servant 定位器	377
	16.7 服务器实现技术	391
	16.8 The Ice: : Context Parameter	414
	16.9 Invocation Timeouts	419
	16.10 Oneway Invocations	421
	16.11 Datagram Invocations	425
	16.12 Batched Invocations	427
	16.13 Testing Proxies for Dispatch Type	429
	16.14 The Ice: : Logger Interface	429
	16.15 The Ice: : Stats Interface	431
	16.16 Location Transparency	432
	16.17 A Comparison of the Ice and CORBA Run Time	434
	16.18 Summary	436
Chapter 17	Asynchronous Programming	443
	17.1 Chapter Overview	443
	17.2 Introduction	443
	17.3 Using AMI	446
	17.4 Using AMD	454
	17.5 Summary	461
Chapter 18	The Ice Protocol	463
	18.1 Chapter Overview	463
	18.2 Data Encoding	464
	18.3 Protocol Messages	487
	18.4 Compression	497
	18.5 Protocol and Encoding Versions	499
	18.6 A Comparison with IIOP	503
Chapter 19	Ice Extension for PHP	511
	19.1 Chapter Overview	511
	19.2 Introduction	511
	19.3 Configuration	513
	19.4 Client-Side Slice-to-PHP Mapping	516

Part IV Ice Services 533

Chapter 20	IcePack	535
20.1	Chapter Overview	535
20.2	Introduction	535
20.3	Concepts	536
20.4	Ice Locator Facility	538
20.5	IcePack Registry	542
20.6	IcePack Node	546
20.7	IcePack Administration Tool	549
20.8	Deployment	552
20.9	Descriptor Reference	559
20.10	Troubleshooting	568
20.11	Summary	570
Chapter 21	Freeze	571
21.1	Chapter Overview	571
21.2	Introduction	572
21.3	The Freeze Map	572
21.4	Using a Freeze Map in the File System Server	581
21.5	The Freeze Evictor	607
21.6	Using the Freeze Evictor in a File System Server	613
21.7	Summary	632
Chapter 22	FreezeScript	633
22.1	Chapter Overview	633
22.2	Introduction	633
22.3	Database Migration	634
22.4	Transformation Descriptors	640
22.5	Using transformdb	653
22.6	Database Inspection	658
22.7	Using dumpdb	669
22.8	Descriptor Expression Language	672
22.9	Summary	676
Chapter 23	IceSSL	677
23.1	Chapter Overview	677
23.2	Introduction	677
23.3	Configuring IceSSL	679
23.4	Configuring Applications	683
23.5	Configuration Reference	685
23.6	Programming with IceSSL	694
23.7	Summary	694

Chapter 24	Glacier	695
	24.1 Chapter Overview	695
	24.2 Introduction	695
	24.3 Using Glacier	699
	24.4 Callbacks	701
	24.5 The Glacier Starter	706
	24.6 Glacier Security	711
	24.7 Summary	716
Chapter 25	IceBox	717
	25.1 Chapter Overview	717
	25.2 Introduction	717
	25.3 The Service Manager	718
	25.4 Developing a Service	719
	25.5 Starting IceBox	726
	25.6 Summary	727
Chapter 26	IceStorm	729
	26.1 Chapter Overview	729
	26.2 Introduction	729
	26.3 Concepts	731
	26.4 IceStorm Interface Overview	734
	26.5 Using IceStorm	736
	26.6 IceStorm Administration	745
	26.7 Topic Federation	746
	26.8 Quality of Service	754
	26.9 Configuring IceStorm	755
	26.10 Summary	756
Appendices		757

Appendix A	Slice Keywords	759
-------------------	-----------------------	------------

Appendix B	Slice Documentation	761
B.1	Ice	761
B.2	Ice: AbortBatchRequestException	765
B.3	Ice: AdapterAlreadyActiveException	765
B.4	Ice: AdapterNotFoundException	765
B.5	Ice: AlreadyRegisteredException	765
B.6	Ice: BadMagicException	766
B.7	Ice: CloseConnectonException	766
B.8	Ice: CloseTimeoutException	767
B.9	Ice: CollocationOptimizationException	767
B.10	Ice: Communicator	767
B.11	Ice: CommunicatorDestroyedException	775
B.12	Ice: CompressionException	776
B.13	Ice: CompressionNotSupportedException	776
B.14	Ice: ConnectFailedException	776
B.15	Ice: ConnectTimeoutException	777
B.16	Ice: ConnectionLostException	777
B.17	Ice: ConnectionNotValidatedException	777
B.18	Ice: ConnectionTimeoutException	778
B.19	Ice: Current	778
B.20	Ice: DNSException	779
B.21	Ice: DatagramLimitException	780
B.22	Ice: EncapsulationException	780
B.23	Ice: EndpointParseException	780
B.24	Ice: FacetNotExistException	781
B.25	Ice: Identity	781
B.26	Ice: IdentityParseException	782
B.27	Ice: IllegalIdentityException	782
B.28	Ice: IllegalIndirectionException	783
B.29	Ice: IllegalMessageSizeException	783
B.30	Ice: Locator	783
B.31	Ice: LocatorRegistry	785
B.32	Ice: Logger	786
B.33	Ice: Marshal Exception	788
B.34	Ice: MemoryLimitException	788
B.35	Ice: NegativeSizeException	788
B.36	Ice: NoEndpointException	789
B.37	Ice: NoObjectFactoryException	789
B.38	Ice: NotRegisteredException	790
B.39	Ice: ObjectAdapter	790
B.40	Ice: ObjectAdapterDeactivatedException	799
B.41	Ice: ObjectAdapterIdleUseException	800

B.42	Ice: : ObjectFactory	800
B.43	Ice: : ObjectNotExi stExcepti on	802
B.44	Ice: : ObjectNotFoundExcepti on	802
B.45	Ice: : Operati onMode	802
B.46	Ice: : Operati onNotExi stExcepti on	803
B.47	Ice: : Pl ugi n	804
B.48	Ice: : Pl ugi nI ni ti al i zati onExcepti on	804
B.49	Ice: : Pl ugi nManager	805
B.50	Ice: : Process	806
B.51	Ice: : Properti es	806
B.52	Ice: : Protocol Excepti on	811
B.53	Ice: : ProxyParseExcepti on	811
B.54	Ice: : ProxyUnmarshal Excepti on	812
B.55	Ice: : RequestFai l edExcepti on	812
B.56	Ice: : Router	813
B.57	Ice: : ServantLocator	814
B.58	Ice: : ServerNotFoundExcepti on	816
B.59	Ice: : SocketExcepti on	816
B.60	Ice: : Stats	817
B.61	Ice: : Syscal l Excepti on	818
B.62	Ice: : Ti meoutExcepti on	818
B.63	Ice: : TwowayOnl yExcepti on	819
B.64	Ice: : UnknownExcepti on	819
B.65	Ice: : UnknownLocal Excepti on	820
B.66	Ice: : UnknownMessageExcepti on	820
B.67	Ice: : UnknownRepl yStatusExcepti on	821
B.68	Ice: : UnknownRequestI dExcepti on	821
B.69	Ice: : UnknownUserExcepti on	821
B.70	Ice: : Unmarshal OutOfBoundsExcepti on	822
B.71	Ice: : UnsupportedEncodi ngExcepti on	822
B.72	Ice: : UnsupportedProtocol Excepti on	823
B.73	Ice: : Versi onMi smatchExcepti on	824
B.74	Freeze	824
B.75	Freeze: : Connecti on	825
B.76	Freeze: : DatabaseExcepti on	826
B.77	Freeze: : Deadl ockExcepti on	826
B.78	Freeze: : EmptyFacetPathExcepti on	827
B.79	Freeze: : Evi ctor	827
B.80	Freeze: : Evi ctorDeacti vatedExcepti on	833
B.81	Freeze: : Evi ctorI terator	833
B.82	Freeze: : Evi ctorStorageKey	835
B.83	Freeze: : I nval i dPosi ti onExcepti on	835

B.84	Freeze: : NoSuchElementException	835
B.85	Freeze: : NotFoundException	836
B.86	Freeze: : ObjectRecord	836
B.87	Freeze: : ServantInitializer	836
B.88	Freeze: : Statistics	837
B.89	Freeze: : Transaction	838
B.90	Freeze: : TransactionAlreadyInProgressException	839
B.91	IceBox	839
B.92	IceBox: : FailureException	839
B.93	IceBox: : FreezeService	840
B.94	IceBox: : Service	841
B.95	IceBox: : ServiceBase	842
B.96	IceBox: : ServiceManager	842
B.97	IcePack	843
B.98	IcePack: : AdapterDeploymentException	843
B.99	IcePack: : AdapterNotExistsException	843
B.100	IcePack: : Admin	844
B.101	IcePack: : BadSignalException	855
B.102	IcePack: : DeploymentException	855
B.103	IcePack: : NodeNotExistsException	856
B.104	IcePack: : NodeUnreachableException	856
B.105	IcePack: : ObjectDeploymentException	857
B.106	IcePack: : ObjectExistsException	857
B.107	IcePack: : ObjectNotExistsException	857
B.108	IcePack: : ParserDeploymentException	857
B.109	IcePack: : Query	858
B.110	IcePack: : ServerActivation	859
B.111	IcePack: : ServerDeploymentException	860
B.112	IcePack: : ServerDescription	860
B.113	IcePack: : ServerNotExistsException	862
B.114	IcePack: : ServerState	862
B.115	IceSSL	864
B.116	IceSSL: : CertificateException	864
B.117	IceSSL: : CertificateKeyMatchException	864
B.118	IceSSL: : CertificateLoadException	865
B.119	IceSSL: : CertificateParseException	865
B.120	IceSSL: : CertificateSignatureException	865
B.121	IceSSL: : CertificateSigningException	865
B.122	IceSSL: : CertificateVerificationException	866
B.123	IceSSL: : CertificateVerifier	866
B.124	IceSSL: : CertificateVerifierTypeException	867
B.125	IceSSL: : ConfigParseException	867

B.126	IceSSL: : Configurati onLoadi ngExcepti on	868
B.127	IceSSL: : ContextExcepti on	868
B.128	IceSSL: : ContextIni tial i zati onExcepti on	868
B.129	IceSSL: : ContextNotConfi guredExcepti on	869
B.130	IceSSL: : ContextType	869
B.131	IceSSL: : Pl ugi n	870
B.132	IceSSL: : Pri vateKeyExcepti on	874
B.133	IceSSL: : Pri vateKeyLoadExcepti on	874
B.134	IceSSL: : Pri vateKeyParseExcepti on	875
B.135	IceSSL: : Protocol Excepti on	875
B.136	IceSSL: : ShutdownExcepti on	875
B.137	IceSSL: : Ssl Excepti on	876
B.138	IceSSL: : TrustedCerti fi cateAddExcepti on	876
B.139	IceSSL: : UnsupportedContextExcepti on	877
B.140	Gl acier	877
B.141	Gl acier: : CannotStartRouterExcepti on	877
B.142	Gl acier: : NoSessi onManagerExcepti on	878
B.143	Gl acier: : Permi ssi onDeni edExcepti on	878
B.144	Gl acier: : Permi ssi onsVeri fi er	878
B.145	Gl acier: : Router	879
B.146	Gl acier: : Sessi on	880
B.147	Gl acier: : Sessi onManager	880
B.148	Gl acier: : Starter	881
B.149	IceStorm	882
B.150	IceStorm: : Li nkExi sts	883
B.151	IceStorm: : Li nkI nfo	883
B.152	IceStorm: : NoSuchLi nk	884
B.153	IceStorm: : NoSuchTopi c	885
B.154	IceStorm: : Topi c	885
B.155	IceStorm: : Topi cExi sts	888
B.156	IceStorm: : Topi cManager	888
B.157	IcePatch	890
B.158	IcePatch: : BusyExcepti on	890
B.159	IcePatch: : Di rectory	891
B.160	IcePatch: : Di rectoryDesc	891
B.161	IcePatch: : Fi l e	891
B.162	IcePatch: : Fi l eAccessExcepti on	892
B.163	IcePatch: : Fi l eDesc	892
B.164	IcePatch: : Regul ar	893
B.165	IcePatch: : Regul arDesc	894

Appendix C	Properties	895
C.1	Ice Configuration Property	895
C.2	Ice Trace Properties	896
C.3	Ice Warning Properties	898
C.4	Ice Object Adapter Properties	900
C.5	Ice Plug-in Properties	902
C.6	Ice Thread Pool Properties	903
C.7	Ice Default and Override Properties	904
C.8	Ice Miscellaneous Properties	906
C.9	IceSSL Properties	912
C.10	IceBox Properties	917
C.11	IcePack Properties	919
C.12	IceStorm Properties	927
C.13	Glacier Router Properties	929
C.14	Glacier Router Starter Properties	934
C.15	Freeze Properties	940
Appendix D	Proxies and Endpoints	945
D.1	Proxies	945
D.2	Endpoints	947
Bibliography		955

第 1 章

引言

1.1 引言

自从上世纪九十年代以来，计算工业一直在使用像 DCOM[3] 和 CORBA[4] 这样的面向对象中间件平台。在使分布式计算能为应用开发者所用的进程中，面向对象中间件是十分重要的一步。开发者第一次拥有了这样的可能：不必是一个网络古鲁（guru），就可以构建分布式应用——中间件平台会照管大部分网络杂务，比如整编（marshaling）和解编（unmarshaling）（对数据进行编码与解码，以进行传送）、把逻辑对象地址映射到物理传输端点、根据客户和服务器的原生机器架构改变数据的表示，以及应需自动启动服务器。

然而，由于一些原因，无论是 DCOM 还是 CORBA，都未能成功占领大部分计算市场：

- DCOM 是 Microsoft 的独家解决方案，在异种网络中，各种机器会运行多种操作系统，无法使用 DCOM。
- DCOM 不能支持大量对象（数十万或数百万），这在很大程度上是它的分布式垃圾收集机制带来的开销造成的。
- 尽管有多家供应商提供 CORBA 产品，几乎不可能找到一家供应商，能够为异种网络中的所有环境提供实现。尽管进行了大量标准化工作，不同的 CORBA 实现之间仍缺乏互操作性，从而不断地造成各种问题；而且，由于供应商常常会自行定义扩展，而 CORBA 又缺乏针对多线程环境的规范，对于像 C 或 C++ 这样的语言，源码兼容性从未完全实现过。

- DCOM 和 CORBA 都过于复杂。要熟悉 DCOM 或 CORBA，并进行相应的设计和编程，是一项需要许多个月来掌握的艰难任务（而要达到专家水平，需要好几年）
- 在其各自的历史中，性能问题一直在折磨这两种平台。DCOM 只有一种实现可用，所以不可能购买性能更好的实现。而尽管有多家供应商提供 CORBA 产品，很难找到遵从标准、性能良好的实现——其主要原因是 CORBA 规范自身所带来的复杂性（在许多情况下，其特性都丰富得超出了需要）
- 在异种环境中，让 DCOM 和 CORBA 共存从来都不是一件容易的事情：尽管有供应商提供互操作产品，这两种平台之间的互操作从来都不是无缝的，而且难以管理，会产生互不相连的技术孤岛。

2002 年，Microsoft .NET 平台 [11] 取代了 DCOM。但尽管 .NET 提供了比 DCOM 更强大的分布式计算支持，它仍然是 Microsoft 的独家解决方案，因而不是异种环境下的选择。另一方面，CORBA 近年来已停滞不前，许多供应商离开了市场，给消费者留下了不再受到广泛支持的平台；剩下的少数供应商在进一步标准化方面的兴趣也已衰退，致使 CORBA 规范中的许多缺陷未能得到解决，或是在它们被报告多年之后才得到解决。

在 DCOM 和 CORBA 衰败的同时，分布式计算社群对 SOAP[23] 和 web services[24] 产生了浓厚的兴趣。使用无处不在的 WWW 基础设施和 HTTP 来开发中间件平台的想法十分迷人——至少在理论上。SOAP 和 web services 曾经允诺要成为 Internet 上的分布式计算通用语言。但尽管引发了很大的公众效应，发表了许多论文，web services 却没有能兑现其允诺：在本书撰写的时候，用 web services 架构开发的商业系统非常少。其原因是：

- 无论是在网络带宽方面，还是在 CPU 开销方面，SOAP 都会给应用造成严重的性能恶化，以致于该技术无法适用于许多有苛刻性能要求的系统。
- 尽管 SOAP 提供了 "on-the-wire" 规范，要开发现实的应用，那仍是不够的，因为该规范提供的抽象层次太低。应用可以把各种 SOAP 消息拼凑在一起，但这样做极其繁琐而易错。
- 缺乏更高级的抽象促使供应商提供各种应用开发平台，使遵从 SOAP 的应用开发自动化。但是，除了协议一级，这些开发平台完全没有标准化，不可避免是私有的，所以用一家供应商开发的应用无法与其他供应商的中间件产品一起使用。
- 关于 SOAP 和 web services 的架构安全性，有一些严重的担忧 [15]。特别地，许多专家都表示，他们对该平台缺乏内在的安全性感到担忧。

- **web services** 是一项还处在幼年的技术。到目前为止，已进行的标准化很少 [24]，而且看起来还需要好几年，其标准化水平才能达到源码兼容性和跨供应商互操作性所要求的完备程度。

结果，想要使用中间件平台的开发者面临着一些使人不快的选择：

- **.NET**

最严重的缺点是不支持非 **Microsoft** 平台。

- **CORBA**

最严重的缺点是老化的平台、高度的复杂性，以及仍在发生的供应商磨擦。

- **web services**

最严重的缺点是严重低效，需要使用私有开发平台，并且还存在着安全问题。

这些选择看起来很可能会让你失败：你可以选择只能在 **Microsoft** 架构上运行的平台，选择复杂的、正在被遗弃的平台，或选择低效的、由于缺乏标准化而归属私有的平台。

1.2 Internet Communications Engine (Ice)

针对这些使人不快的选择，ZeroC, Inc. 决定开发 **Internet Communications Engine**，简称 **Ice**¹。其主要设计目标是：

- 提供适用于异种环境的面向对象中间件平台。
- 提供一组完整的特性，支持广泛的领域中的实际的分布式应用的开发。
- 避免不必要的复杂性，使平台更易于学习和使用。
- 提供一种在网络带宽、内存使用和 **CPU** 开销方面都很高效的实现。
- 提供一种具有内建安全性的实现，使它适用于不安全的公共网络。

更简单地说，**Ice** 的设计目标可陈述为：“让我们构建与 **CORBA** 一样强大的中间件平台，而又不去犯 **CORBA** 所犯下的任何错误”。

1. 首字母缩写 “Ice” 的发音与 **Ice**（结冰的水）这个词一样，是单音节的。

1.3 本书的篇章结构

本书划分为四个部分，另外有几个附录：

- 第一部分，Ice 综述，对 Ice 所提供的各种特性进行综述，并解释 Ice 对象模型。在阅读了这一部分之后，你将会理解 Ice 平台的主要特性和架构，理解它的对象模型和请求分派模型（request dispatch model），并且了解用 C++ 和 Java 构建一个简单应用的基本步骤。
- 第二部分，Ice 核心概念，解释 Slice 定义语言，并介绍 C++ 和 Java 语言映射。第二部分还将涵盖 Ice 线程模型和 Ice 线程 API。在阅读了这一部分之后，你将详细地了解到怎样为分布式应用规定接口，以及怎样用 C++ 或 Java 实现该应用。
- 第三部分，高级 Ice，详细介绍 Ice 的许多特性，并涵盖服务器开发的一些高级的方面，比如对象生命周期、对象定位、持久，以及异步方法祈用与分派（asynchronous method invocation and dispatch）。在阅读了这一部分之后，你将会了解 Ice 的一些高级特性，以及怎样按照你的应用需求、有效地使用它们，在性能和资源消耗之间求得适当的平衡。
- 第四部分，Ice 服务，涵盖 Ice 所提供的一些服务，比如 IcePack（完备的部署工具）、Glacier（Ice 防火墙解决方案）、IceStorm（Ice 消息服务）、IcePatch（软件修补服务）²。
- 附录含有 Ice 的参考资料。

1.4 排字约定

本书采用了以下排字约定：

- Slice 源码所用字体是 Lucida Sans Typewriter。
- C++ 或 Java 源码所用字体是 Courier。
- 文件名所用字体是 Courier。
- 命令所用字体是 **Courier Bold**。

有时候，我们会给出终端上的交互式会话的副本。在这样的情况下，我们采用的是 Bourne shell（或它的某种派生 shell，比如 **ksh** 或 **bash**）。系

2. 如果你注意到 Ice 的各种特性的命名方式中的共同性，那只是表明，软件开发者仍然根深蒂固地爱说双关语。

统给出的输出用 Courier 字体显示，输入则用 **Courier Bold** 字体显示。例如：

```
$ echo hello  
hello
```

Slice 和 C++ 或 Java 常常会使用相同的标识符。当我们在和语言无关的一般意义上谈到某个标识符时，我们会使用 Lucida Sans Typewriter 字体。当我们在与特定语言相关的意义上谈到某个标识符时，我们会使用 Courier 字体。

1.5 源码示例

贯穿全书，我们将采用一个案例研究来阐述 Ice 的各个方面。这个案例研究将实现一个简单的分布式层次文件系统，我们将随着内容的进展逐渐对其加以改进，以利用各种更高级的特性。案例研究的各个阶段的源码随本书一同提供。我们鼓励你试验这些代码示例（以及随同 Ice 发布的许多演示程序）。

1.6 联系作者

如果你在本书中发现了问题（不管有多小），我们非常愿意收到你的来信。我们也希望听到你提出关于本书内容的意见，以及关于如何改进本书的建议。你可以通过 e-mail 联系我们：<mailto:icebook@zeroc.com>。

1.7 Ice 支持

ZeroC 在 <http://www.zeroc.com/support.html> 设有讨论和支持论坛。你可以在论坛上提出任何与 Ice 平台有关的问题或建议，也可以在那里就你遇到的特定问题寻求解答。

第一部分

Ice 综述

第 2 章

Ice 综述

2.1 本章综述

在这一章，我们将在高级层面上综述 Ice 的架构。2.2 节介绍基础性的概念和术语，并概述 Slice 定义、语言映射，以及 Ice run time 和协议是怎样协同工作、创建客户与服务器的。2.3 节简要介绍 Ice 提供的对象服务，2.4 节概述 Ice 架构带来的各种好处。最后，2.5 节简要地对比 Ice 与 CORBA 架构。

2.2 Ice 架构

2.2.1 引言

Ice 是一种面向对象的中间件平台。从根本上说，这意味着 Ice 为构建面向对象的客户—服务器应用提供了工具、API 和库支持。Ice 应用适合在异种环境中使用：客户和服务器可以用不同的编程语言编写，可以运行在不同的操作系统和机器架构上，并且可以使用多种网络技术进行通信。无论部署环境如何，这些应用的源码都是可移植的。

2.2.2 术语

每一种计算技术都会随着自身的演化创造出自己的词汇表。Ice 也不例外。但它使用的新行话（jargon）的数量很少。我们尽可能使用已有的术语，而不是发明新的。如果你曾经使用过其他中间件技术，比如 CORBA，你将会很熟悉后面使用的大多数术语（但我们建议，你至少应该浏览一下这部分内容，因为 Ice 使用的一些术语确实与对应的 CORBA 术语不同）。

客户与服务器（Clients and Servers）

客户与服务器这两个术语不是对应用的特定组成部分的严格指称，而是表示在某个请求从发生到结束期间，应用的某些部分所承担的角色：

- 客户是主动的实体。它们向服务器发出服务请求。
- 服务器是被动的实体。它们提供服务，响应客户请求。

在从不发出请求、而只是响应请求的意义上，许多服务器常常不是“纯粹的”服务器：它们常常充当某些客户的服务器，但为了完成它们的客户的请求，它们又会充当另外的服务器的客户。

与此类似，在只从某个对象那里请求服务的意义上，客户常常也不是“纯粹的”客户：它们常常是客户—服务器混合物。例如，客户可以在服务器上启动一个长时间运行的操作，在启动该操作时，客户可以向服务器提供回调对象（callback object），供服务器用于在操作完成时向客户发出通知。在这种情况下，客户在启动操作时充当客户，而在接收操作完成通知时充当服务器。

这样的角色反转在许多系统中都很常见，所以，许多客户—服务器系统常常可以被更准确地描述为对等（peer-to-peer）系统。

Ice 对象（Ice Objects）

Ice 对象是一种概念性的实体（或称抽象）。Ice 对象具有以下特征：

- Ice 对象是本地或远地的地址空间中、能响应客户请求的实体。
- 一个 Ice 对象可在单个或多个服务器中实例化（后者是冗余方式）。如果某个对象同时有多个实例，它仍是一个 Ice 对象。
- 每个 Ice 对象都有一个或多个接口。一个接口是一个对象所支持的一系列有名称的操作。客户通过祈用（invoking）操作来发出请求。
- 一个操作有零个或更多参数，以及一个返回值。参数和返回值具有明确的类型。参数是有名称的，并且有方向：in 参数由客户初始化，并传给服务器；out 参数由服务器初始化，并传给客户（返回值只是一种特殊的 out 参数）。

- 一个 Ice 对象具有一个特殊的接口，称为它的**主接口**。此外，Ice 对象还可以提供零个或更多其他接口，称为 *facets*（面）。客户可以在某个对象的各个 facets 之间进行挑选，选出它们想要使用的接口。
- 每个 Ice 对象都有一个唯一的**对象标识**（object identity）。对象标识是用于把一个对象与其他所有对象区别开来的标识值。Ice 对象模型假定对象标识是全局唯一的，也就是说，在一个 Ice 通信域中，不会有两个对象具有相同的对象标识。

在实践中，你不需要使用像 UUID[14] 这样的全局唯一的对象标识，只要你使用的标识与你感兴趣的域中的其他任何标识不相冲突，就可以了。但在架构上，使用全局唯一的标识符能带来一些好处，我们将在 XREF 中对此加以探究。

代理 (Proxies)

要想与某个 Ice 对象联系，客户必须持有这个对象的**代理**¹。代理是客户的地址空间中的一种制品（artifact）；对客户而言，代理就是 Ice 对象的代表（该对象可能在远地）。一个代理充当的是一个 Ice 对象的本地大使：当客户祈用代理上的操作时，Ice run time 会：

1. 定位 Ice 对象
2. 如果 Ice 对象的服务器没有运行，就激活它
3. 在服务器中激活 Ice 对象
4. 把所有 in 参数传送给 Ice 对象
5. 等待操作完成
6. 把所有 out 参数及返回值返回给客户（或在发生错误的情况下抛出异常）

代理封装了完成这一系列步骤所必需的全部信息。特别地，代理包含有：

- 寻址信息：用于让客户端 run time 联系正确的服务器
- 对象标识：用于确定服务器中的哪一个对象是请求的目标
- 可选的 facet 标识符：用于确定代理所引用的是对象的哪一个 facet

1. 代理是 CORBA 对象引用（object reference）的等价物。我们使用“代理”，而不是“引用”，是为了避免混淆：在各种编程语言里，“引用”已经有了太多其他含义。

串化代理 (Stringified Proxies)

代理中的信息可以用串的形式表示。例如：

```
SimplePrinter:default -p 10000
```

这个字符串表示的是一个代理，我们可以阅读这种表示方式。Ice run time 提供了一些 API 调用，允许你把代理转换成它的串化形式，或是进行相反的转变。例如，如果你要把代理存储在数据库表或文本文件中，这种功能会很有用。

倘若客户知道某个 Ice 对象的标识及其寻址信息，使用这些信息，它可以“凭空”创建代理。换句话说，代理内部的所有信息都被认为是透明的；要与某个对象联系，客户只需要知道这个对象的标识、寻址信息，以及对象的类型（为了能祈用操作），就可以了。

直接代理 (Direct Proxies)

直接代理是这样一种代理：其内部保存有某个对象的标识，以及它的服务器的运行地址。该地址由以下两项内容完全确定：

- 协议标识符（比如 TCP/IP 或 UDP）
- 针对具体协议的地址（比如主机名和端口号）

为了联系直接代理所代表的对象，Ice run time 使用代理内部的寻址信息来联系服务器；每当客户向服务器发出请求时，也会把对象的标识发送过去。

间接代理 (Indirect Proxies)

间接代理是这样一种代理：其内部保存有某个对象的标识，以及对象适配器名（object adapter name）。要注意，间接代理没有包含寻址信息。为了正确地定位服务器，客户端 run time 会使用代理内部的对象适配器名，将其传给某个定位器服务，比如 IcePack 服务。然后，定位器会把适配器名当作关键字，在含有服务器地址的表中进行查找，把当前的服务器地址返回给客户。客户端 run time 现在知道了怎样联系服务器，就会像平常一样分派（dispatch）客户请求。

这整个过程与 Domain Name Service（DNS）所进行的映射类似，DNS 会把 Internet 域名映射到 IP 地址：当我们使用域名（比如 www.zeroc.com）来查找某个网页时，主机名首先在幕后被解析成 IP 地址，一旦得到了正确的 IP 地址，这个地址就会用于连接服务器。在 Ice 中，对象适配器名会映射到协议—地址对（protocol-address pair），但其他方面则非常类似。客户端 run time 会通过配置了解怎样去和 IcePack 服务联系（就像 web 浏览器会通过配置了解要使用哪一个 DNS）。

直接绑定 vs. 间接绑定 (Direct Versus Indirect Binding)

把代理里面的信息解析为协议—地址对的过程称为 *绑定*。不奇怪，*直接绑定* 用于直接代理，而 *间接绑定* 用于间接代理

间接绑定的主要好处是，它允许我们移动服务器（也就是说，改变它们的地址），同时又不会使客户所持有的已有代理失效。换句话说，使用直接代理，你不用为了定位服务器而进行额外的查找，但如果服务器被移到其他机器上，它就不再能工作。而另一方面，即使我们移动（或 *迁移*，migrate）服务器，间接代理也能够继续工作。

Servants

我们在第 10 页提到过，Ice 对象是一种具有类型、标识，以及寻址信息的概念性实体。但客户请求最终必须到达具体的服务器端的处理实体，由这样的实体提供操作祈用（operation invocation）的行为。换句话说，客户请求最后必须到达服务器，在其内部执行代码，而这些代码用特定的编程语言编写，并在特定的处理器上执行。

在服务器端提供操作祈用的行为的制品叫作 *servant*。一个 servant 提供一个或多个 Ice 对象的实质内容（或 *体现* 这些对象，incarnate）。实际上，servant 就是服务器开发者编写的类的实例，这些类作为一个或多个 Ice 对象的 servant、向服务器端 run time 进行登记。类的方法对应于 Ice 对象的接口上的操作，并且提供这些操作的行为。

一个 servant 可以只体现一个 Ice 对象，也可以同时体现若干 Ice 对象。如果是前一种情况，servant 所体现的 Ice 对象的标识在这个 servant 中是隐含的。如果是后一种情况，在每次收到请求时，servant 也会收到 Ice 对象的标识，这样，servant 可以决定在处理该请求期间，体现哪一个对象。

反过来，一个 Ice 对象也可以拥有多个 servant。例如，我们可以为某个 Ice 对象创建一个代理，这个对象有两个不同的地址，分别在两台机器上。在这种情况下，我们将拥有两个服务器，每个服务器都有一个 servant，但两个 servant 体现的是同一个 Ice 对象。当客户调用这样的 Ice 对象上的操作时，客户端 run time 只把请求发给一个服务器。换句话说，使用体现同一个 Ice 对象的多个 servant，你可以构建冗余的系统：客户端 run time 试着把请求发给一个服务器，如果失败，就把请求发给第二个服务器。只有在第二次尝试也失败的情况下，错误才会报告给客户端应用代码。

“最多一次”语义 (At-Most-Once Semantics)

Ice 请求具有“最多一次”语义：Ice run time 尽力把请求递送给正确的目的地，同时，取决于实际情况，可以重新尝试递送失败的请求。Ice 保证：或者递送请求，或者在无法递送请求的情况下，通过适当的异常通知

客户；在任何情况下请求都不会递送两次，也就是说，只有在确知先前的尝试已经失败的情况下，才会进行重试²。

“最多一次”语义十分重要，因为这保证了非 *idempotent* 操作可以安全使用。*idempotent* 操作是这样的操作：如果执行两次，其效果与执行一次相同。例如，`x = 1;` 是 *idempotent* 操作：如果我们两次执行该操作，最终结果与执行一次是一样的。另一方面，`x++;` 不是 *idempotent* 操作：如果我们执行该操作两次，最终结果与我们执行一次不一样。

如果不采用“最多一次”语义，我们可以构建在出现网络故障时更加健壮的分式系统。但现实系统需要非 *idempotent* 操作，所以“最多一次”语义是一种必需品，即使这会使系统在遇到网络故障时的健壮性降低。Ice 允许你把个别的操作标记为 *idempotent* 的。对于这样的操作，Ice run time 使用的错误恢复机制要比用于非 *idempotent* 操作的机制更积极。

同步方法祈用 (Synchronous Method Invocation)

在缺省情况下，Ice 使用的请求分派模型是同步的远地过程调用：操作祈用的行为就像是本地过程调用，也就是说，在调用期间，客户线程被挂起，并在调用完成（及它的所有结果可用）时恢复。

异步方法祈用 (Asynchronous Method Invocation)

Ice 还支持异步方法祈用 (AMI)：客户可以异步地祈用操作，也就是说，客户像平常一样使用代理来祈用操作，但除了传递通常的参数以外，还要传递一个回调对象，而客户祈用会立即返回。一旦操作完成，客户端 run time 会祈用一开始所传递的回调对象上的方法，把操作的结果传给该对象（或在失败时传递异常信息）。

服务器无法区分异步调用和同步调用——无论是哪种情况，服务器看到的都只是客户调用了某个对象上的操作。

异步方法分派 (Asynchronous Method Dispatch)

异步方法分派 (AMD) 是 AMI 的服务器端等价物。在进行同步分派时（这是缺省方式），服务器端 run time 向上调用 (up-call) 服务器中的应用代码，对操作祈用作出响应。当操作在执行时（或在休眠时，例如，因为它在等待数据），一个执行线程会被“束缚”在服务器中；只有在操作完成后这个线程才会被释放。

2. 这条规则有一个例外：UDP 传输机制上的数据报祈用。在这样的情况下，重复的 UDP 包可能会造成对“最多一次”语义的违反。

而采用异步方法分派，当操作祈用到达时，服务器端应用代码会收到通知。但是，服务器端应用不会被迫立即处理请求，而是可以选择延缓处理，从而释放用于处理该请求的执行线程。至此，服务器端应用代码就可以随意做它想做的任何事情了。最后，当操作的结果可用时，服务器端应用代码可以发出一个 API 调用，告诉服务器端 **Ice run time**，先前分派的某个请求现在已经完成；这时，操作的结果就会返回给客户。

异步方法分派十分有用，例如，它可以用于这样的情况：服务器提供的操作要在很长一段时间里阻塞客户。又例如，服务器可能拥有这样一个对象：这个对象有一个 **get** 操作，返回的是来自外部异步数据源的数据，并且会阻塞客户，直到数据变得可用为止。采用同步分派，每个等待数据到达的客户都会在服务器中占用一个线程。显然，采用这种途径，无法扩展到支持几十个以上的客户。采用异步分派，数百或数千客户可以阻塞在同一个操作祈用中，而又不用在服务器中占用任何线程。

使用异步方法分派的另一种方式是：完成操作，把操作的结果返回给客户，但在操作祈用期过后仍保留操作的执行线程。这样，你可以在把结果返回给客户之后继续进行各种处理，例如，执行清理工作，或把更新写到持久存储器中。

对于客户而言，同步和异步分派是透明的，也就是说，客户不知道服务器会选择同步还是异步方式对请求进行处理。

单向方法祈用（Oneway Method Invocation）

客户可以把操作当作单向操作来祈用。单向祈用具有“尽力”语义。在处理单向祈用时，客户端 **run time** 会把祈用交给本地传输机制，只要本地传输机制缓冲了该祈用，客户端的祈用就算完成了。随后，实际的祈用会由操作系统异步发送。服务器不会答复单向祈用，也就是说，通信数据只会从客户流向服务器，而不会反向流动。

单向祈用是不可靠的。例如，目标对象可能不存在，在这种情况下，祈用会丢失。与此类似，即使操作分派给了服务器中的 **servant**，它也可能会失败（例如，因为参数值无效）；如果是这样，客户不会收到出错通知。

只有对没有返回值、没有输出参数、也不抛出用户异常的操作（参见第 4 章），才能进行单向祈用。

对于服务器端的应用代码而言，单向调用是透明的，也就是说，没有办法区分双向祈用和单向祈用。

只有当目标对象提供了面向流的传输机制时（比如 **TCP/IP** 或 **SSL**），才能使用单向祈用。

注意，即使单向操作是在面向流的传输机制上发送的，服务器也可能不按次序处理它们。之所以会这样，是因为每个祈用都在它自己的线程中被分派：即使祈用发起（**initiated**）的次序与祈用到达服务器的次序相同，

也不意味着它们将以那样的次序被处理——变化莫测的线程调度可能会造成某个单向调用先于其他更早收到的单向调用完成。

成批的单向方法祈用 (Batched Oneway Method Invocation)

每个单向祈用都会向服务器发送一条单独的消息。如果是一系列短小的消息，这样做的开销相当可观：客户和服务器端 `run time` 必须为了每一条消息、在用户模式与内核模式之间切换，同时，在网络层，每条消息还会带来流控制和确认开销。

*成批的单向祈用*允许你在一条消息中发送一系列单向祈用：每次你调用一个这样的单向操作，祈用都会缓冲在客户端 `run time` 中。一旦你累积了所有你想要发送的单向祈用，你就发出另外一个 API 调用，一次发送所有祈用。客户端 `run time` 随即在单条消息中发送所有已缓冲的祈用，而服务器会在单条消息中收到所有祈用。这样，无论是客户还是服务器，都避免了产生反复进入内核的开销，而且在两者之间的网络上，数据传送也更加轻松，因为传送一条大的消息比传送许多小消息要更加高效。

成批的单向消息中的各个祈用由一个单独的线程分派，其分派次序就是它们被放进这个批消息时的次序。这保证了成批的单向消息中的各个操作会按照次序在服务器中被处理。

对于各种消息传递服务（比如 `IceStorm`，参见第 26 章），以及为小属性提供了 `set` 操作的细粒度操作，成批的单向调用特别有用。

数据报祈用 (Datagram Invocations)

*数据报祈用*具有与单向祈用类似的“尽力”语义。但数据报祈用要求对象提供 `UDP` 作为传输机制（而单向祈用要求提供 `TCP/IP`）。

和单向祈用一样，只有当操作没有返回值、输出参数，或是用户异常时，才能进行数据报祈用。数据报祈用使用 `UDP` 来祈用操作。一旦本地的 `UDP` 栈接受了消息，操作就会返回；实际的操作祈用由网络栈在幕后异步发送出去。

数据报和单向祈用一样是不可靠的：目标对象在服务器上可能并不存在、服务器可能没有运行，还有可能，操作在服务器上进行了祈用，但由于客户发送的参数无效，导致祈用失败。与单向祈用一样，客户也不会收到关于这些错误的通知。

但与单向祈用不同，数据报祈用可能会发生另外一些错误：

- 个别启用可能会在网上丢失。

这是由于 `UDP` 包的递送是不可靠的。例如，如果你依次祈用三个操作，中间的祈用可能会丢失（单向祈用不会发生这样的事情——因为它们是通过面向连接的传输机制递送的，不可能发生个别祈用丢失的情况）。

- 各个祈用可能会不按次序到达。

这也是 UDP 数据报的本性所致。因为每个祈用都是用单独的数据报发送的，而且各个数据报可能会按照不同的路径通过网络，所以可能会发生这样的情况：各个祈用的到达次序与它们的发送次序不一样。

数据报祈用很适用于 LAN 上的小消息，在 LAN 上，丢失的可能性很小。它们也适用于低延迟比可靠性更重要情形，比如快速的交互式 Internet 应用。

成批的数据报祈用 (Batched Datagram Invocations)

与成批的单向祈用一样，*成批的数据报祈用*允许你在缓冲区中累积一批祈用，然后发出一个 API 调用，刷出缓冲区，在一个数据报中把整个缓冲区发送出去。成批的数据报能够减少重复的系统调用造成的开销，并且让底层网络更高效地进行操作。但只有在这样的情况下，成批的数据报祈用才有用：实际的成批消息的总尺寸没有超过网络的 PDU 限制——如果一个成批数据报的尺寸太大，UDP 分段 (fragmentation) 会使得一个或多个分段丢失的可能性增大，从而导致整个成批消息的丢失。但你可以确信，一个成批消息中的祈用或者全部被递送，或者一个也不被递送。不可能发生一个成批消息中的个别祈用丢失的情况。

成批的数据报使用服务器上的一个单独的线程来分派一个成批消息中的各个祈用。这保证了各个祈用会按照它们的排列次序执行——各个调用不可能在服务器上发生次序变化。

运行时异常 (Run-Time Exceptions)

任何操作祈用都可以引发*运行时异常*。运行时异常由 Ice run time 预先定义，涵盖了常见的出错情况，比如连接失败、连接超时，或资源分配失败。对于应用而言，运行时异常是作为适当的 C++ 或 Java 异常给出的，并且与这些语言的原生异常处理能力完好地集成在了一起。

用户异常 (User Exceptions)

用户异常用于向客户指示应用特有的出错情况。用户异常可以携带任意数量的复杂数据，并且可以按照继承层次进行组织，这样，通过捕捉处在继承层次上层的异常，客户就能够轻松地对各个错误范畴进行一般化的处理。与运行时异常一样，用户异常会映射到 C++ 和 Java 的原生异常。

属性 (Properties)

Ice run time 有大量功能都是通过*属性*来配置的。属性就是“名—值”对，比如 Ice.Default.Protocol=tcp。属性通常存储在文本文件中，

Ice run time 会对其进行解析，从而配置各种选项，比如线程池尺寸、跟踪级别，以及各种其他的配置参数。

2.2.3 Slice (Ice 规范语言)

在第 10 页提到，每个 Ice 对象都有一个接口，该接口具有一些操作。接口、操作，还有在客户及服务器间交换的数据的类型，都是用 *Slice 语言* 定义的。Slice 允许你以一种独立于特定编程语言（比如 C++ 或 Java）的方式定义客户-服务器的合约。Slice 定义由一个编译器编译成特定编程语言的 API，也就是说，与你所定义的接口和类型对应的那一部分 API，会由生成的代码组成。

2.2.4 语言映射

*语言映射*是一些规则，决定怎样把每个 Slice 成分（construct）翻译到特定语言。例如，就 C++ 映射而言（参见第 6 章），Slice 序列（sequence）会作为 STL 向量（vector）出现，而就 Java 映射而言（参见第 8 章），Slice 序列会作为 Java 数组出现。为了确定特定 Slice 成分的 API 是什么样的，你只需要知道 Slice 定义，并且了解语言映射的规则。这些规则都非常简单和规范，所以要想知道怎样使用生成的 API，你无需阅读生成的代码。

当然，你可以去阅读生成的代码。但作为一条准则，你应该知道，那样做效率很低，因为生成的代码不一定适于让人阅读。我们建议你让自己通晓语言映射的各种规则；这样，你通常就可以忽略生成的代码，只需在你要了解某个具体细节时参考这些代码就可以了。

Ice 目前提供了 C++、Java，以及 PHP（用于客户端，参见第 19 章）语言映射。

2.2.5 客户与服务器的结构

Ice 客户与服务器内部的逻辑结构如图 2.1 所示：

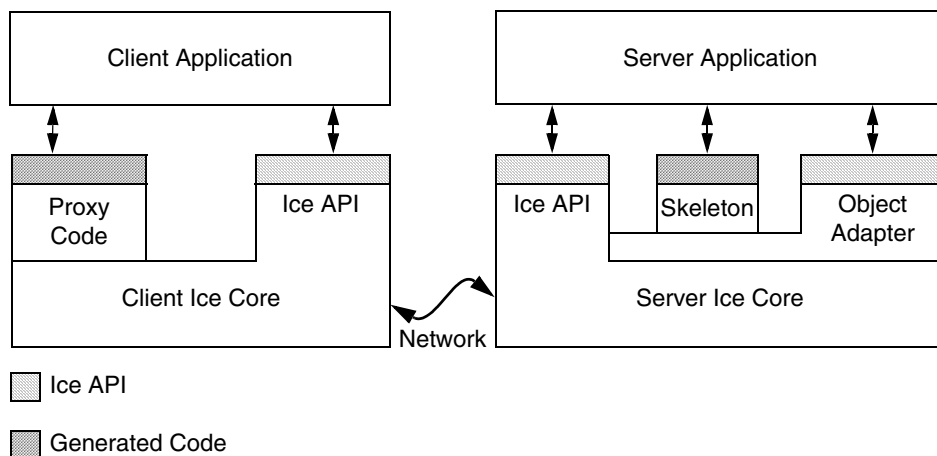


图 2.1. Ice 客户与服务器的结构

客户与服务器都由这样一些代码混合而成：应用代码、库代码、根据 Slice 定义生成的代码：

- Ice 核心为远地通信提供了客户端和服务端运行时支持。其中的大量代码所涉及的是网络通信、线程、字节序，以及其他许多与网络有关的问题，我们想要让应用代码与这些问题隔离开来。Ice 核心是作为客户和服务端可与之链接的库提供的。
- Ice 核心的通用部分（也就是说，与你用 Slice 定义的特定类型无关的部分）可通过 Ice API 访问。你用 Ice API 来照管各种管理事务，比如 Ice run time 的初始化和结束。用于客户和服务端的 Ice API 是一样的（尽管服务端使用的 API 比客户要多）。
- 代理代码是根据你的 Slice 定义生成的，因此，与你用 Slice 定义的对象和数据的类型是对应的。代理代码有两个主要功能：
 - 它为客户提供了一个向下调用（down-call）接口。如果你调用“生成的代理 API”中的某个函数，就会有一个 RPC 消息被发给服务器，祈用目标对象上的某个对应的函数。
 - 它提供了整编（marshaling）和解编（unmarshaling）代码。

整编是使复杂的数据结构（比如序列或词典）序列化、以在线路上进行传送的过程。整编代码把数据转换成适于传送的标准形式，这种形式不依赖于本地机器的 "endian-ness" 和填充（padding）规则。

解编是整编的逆过程，也就是说，使通过网络到达的数据解除序列化，并且对数据重新进行构造，用与所使用的编程语言相适应的类型来加以表示。

- 骨架（skeleton）代码也是根据你的 Slice 定义生成的，因此，与你用 Slice 定义的对象和数据的类型是对应的。骨架代码是客户端代理代码的服务器端等价物：它提供了向上调用（up-call）接口，允许 Ice runtime 把控制线程转交给你编写的应用程序代码。骨架也含有整编和解编代码，所以服务器可以接收客户发送的参数，并把参数和异常返回给客户。
- 对象适配器（object adapter）是专用于服务器端的 Ice API 的一部分：只有服务器才使用对象适配器。对象适配器有若干功能：
 - 对象适配器把来自客户的请求映射到编程语言对象上的特定方法。换句话说，对象适配器会跟踪在内存中，都有哪些 servant，其对象标识又是什么。
 - 对象适配器与一个或多个传输端点关联在一起。如果与某个适配器关联的传输端点不止一个，你可以通过多种传输机制到达在该适配器中的 servant。例如，为了提供不同的服务质量和性能，你可以把一个 TCP/IP 端点和一个 UDP 端点与一个适配器关联在一起。
 - 对象适配器要负责创建可以传给客户的代理。对象适配器知道它的每个对象的类型、标识，以及传输机制的详细情况，并且会在服务器端应用程序代码要求创建代理时在其中嵌入正确的信息。

注意，从进程的层面来看，所涉及的进程只有两个：客户与服务器。对分布式通信的所有运行时支持都是由 Ice 库以及根据 Slice 定义生成的代码提供的（在使用间接代理时，需要使用第三个进程 IcePack 来把代理解析为传输端点）。

2.2.6 Ice 协议

Ice 提供了一种 RPC 协议，既可以把 TCP/IP、也可以把 UDP 用作底层传输机制。此外，Ice 还允许你把 SSL 用作传输机制，让客户与服务器间的所有通信都进行加密。

Ice 协议定义了：

- 一些消息类型，比如请求和答复类型，

- 协议状态机，确定客户与服务器以怎样的序列交换不同的消息类型，同时还包括相关的 TCP/IP 连接建立和关闭语义，
- 编码规则，确定在线路上怎样表示数据的类型，
- 每种消息类型的头，其中含有像这样的细节：消息类型、消息尺寸、所使用的协议及编码版本。

Ice 还支持在线路上进行压缩：通过设置一个配置参数，你可以让所有的网络通信数据都被压缩，从而节省带宽。如果你的应用要在客户与服务器间交换大量数据，这种功能会很有用。

Ice 协议适用于构建高效的事件转发机制，因为要想转发消息，你不需要了解消息内部的详细信息。这意味着，消息交换机不需要对消息进行任何解编或重整编——它们可以简单地把消息当作不透明的字节缓冲区加以转发。

Ice 协议还适用于构建双向操作：如果服务器想要把一条消息发送给客户提供的某个回调对象，这个回调对象可以通过客户原来创建的连接传给服务器。如果客户在防火墙后面，连接只能外出，不能进入，这种特性就特别重要。

2.2.7 对象持久

Ice 拥有内建的对象持久服务，叫作 *Freeze*。*Freeze* 能够让我们轻松地在数据库中存储对象状态：你用 *Slice* 定义你的对象要存储的状态，*Freeze* 编译器会生成代码，用以在数据库中存储和取回对象状态。*Freeze* 使用 Berkeley DB[18] 作为它的缺省数据库。但如果你更喜欢在其他数据库中存储对象状态，你也可以那样做。我们将在第 21 章详细讨论 *Freeze*。

Ice 还提供了一些工具，能让我们更轻松的管理数据库，并在对象的类型定义发生变化时，把已有数据库的内容迁移到新的数据库中。我们将在第 22 章讨论这些工具。

2.3 Ice 服务

Ice 核心为分布式应用开发提供了一个完善的客户—服务器平台。但现实应用需要的常常不止是远地通信能力：你通常还需要拥有这样的能力：按需启动服务器、把代理分发给客户、分发异步事件、配置你的应用、分发应用补丁，等等。

在 Ice 中有一些服务，能够提供上述特性及其他一些特性。这些服务被实现成 Ice 服务器，你的应用充当的是这些服务器的客户。这些服务都没有使用 Ice 的任何向应用开发者隐藏起来的内部特性，所以在理论上，你可以

自行开发等价的服务。但让这些服务成为平台的一部分，你就可以专注于应用开发，而不必先构建许多基础设施。而且，构建这样的服务所需的工作量并非微不足道，所以你应该了解有哪些服务可用，而不要重新发明你自己的轮子。

2.3.1 IcePack

我们在第 12 页提到过，IcePack 是 Ice 的定位服务，用于在使用间接绑定定时把符号性的（symbolic）适配器名解析为协议—地址对。除了这样的定位服务，IcePack 还提供了其他特性：

- IcePack 允许你登记服务器，进行自动启动：当客户发出请求时，服务器无需在运行，IcePack 会在第一个客户请求到达时，按需启动服务器。
- IcePack 支持部署描述符（deployment descriptors），能让你轻松地配置含有若干服务器的复杂应用。
- IcePack 提供了一种简单的对象查找服务，客户可用来获取它们感兴趣的对象的代理。

2.3.2 IceBox

IceBox 是一种简单的应用服务器，可用于协调许多应用组件的启动和停止。应用组件可以作为动态库、而不是进程进行部署。例如，你可以在单个 Java 虚拟机中运行若干应用组件，而无需使用多个拥有自己的虚拟机的进程，从而减轻整个系统的负担。

2.3.3 IceStorm

IceStorm 是一种发布—订阅服务，能够解除客户与服务器的耦合。在本质上，IceStorm 充当的是事件分发交换机。发布者把事件发给这个服务，由它发给订阅者。这样，发布者发布的单个事件就可以发送给多个订阅者。事件按照主题进行分类，订阅者会指定它们感兴趣的主题。只有那些与订阅者感兴趣的主题相吻合的主题才会发给这个订阅者。这个服务允许你指定服务质量标准，让应用在可靠性和性能之间进行适当的折衷。

如果你需要把信息分发给大量应用组件，IceStorm 就会特别有用（一个典型的例子是，拥有大量订阅者的证券报价应用）。IceStorm 能解除信息的发布者与订阅者的耦合，并负责重新分发已发布的信息。此外，IceStorm 还可以作为联盟（federated）服务运行，也就是说，服务的多个实例可以在不同的机器上运行，使处理负载分摊到许多 CPU 上。

2.3.4 IcePatch

IcePatch 是一种软件修补服务。你可以用它来轻松地把软件更新分发给客户。客户可以简单地连接到 IcePatch，请求获得特定应用的更新。这个服务会自动检查客户的软件版本，并以一种压缩形式下载任何更新过的应用组件，从而节省带宽。你可以用 Glacier 服务来保护软件补丁，只让得到授权的客户下载软件补丁。

2.3.5 Glacier

Glacier 是 Ice 防火墙服务：它能让客户与服务器通过防火墙安全地进行通信，且又不牺牲安全性。客户—服务器之间的通信数据使用公钥证书进行了完全的加密，并且是双向的。Glacier 支持相互认证，以及安全的会话管理。

2.4 Ice 在架构上提供的好处

Ice 在架构上为应用开发者提供了一些好处：

- 面向对象的语义

Ice “在线路上”完全保留了面向对象范型。所有的操作都用都使用迟后绑定，所以操作的实现的选定，是根据对象在运行时的（而不是静态的）实际类型决定的。

- 支持同步和异步的消息传递

Ice 提供了同步和异步的操作调用和分派，并且通过 IceStorm 提供了发布—订阅消息传递机制。这样，你可以根据你的应用的需要来选择通信模型，而不必把你的应用硬塞进某种模型里。

- 支持多个接口

通过 facets，对象可以提供多个不相关的接口，同时又跨越这些接口、保持单一的对象标识。这提供了极大的灵活性，特别是在这样的情况下：应用在发生演化，但又需要与更老的、已经部署的客户保持兼容。

- 机器无关性

客户及服务器与底层的机器架构屏蔽开来。对于应用代码而言，像字节序和填充这样的问题都隐藏了起来。

- 语言无关性

客户和服务端可以分别部署，所用语言也可以不同（目前支持 C++、Java，以及 PHP（客户端））。客户和服务端所用的 Slice 定义建立两者之间的接口合约，这样的定义也是它们唯一需要达成一致的东西。

- 实现无关性

客户不知道服务端是怎样实现其对象的。这意味着，在客户部署之后，服务端的实现可以改变，例如，它可以使用不同的持久机制，甚至不同的程序设计语言。

- 操作系统无关性

Ice API 完全是可移植的，所以同样的源码能够在 Windows 和 UNIX 上编译和运行。

- 线程支持

Ice run time 完全是线程化的，其 API 是线程安全的。作为应用开发者，（除了在访问共享数据时进行同步）你无需为开发线程化的高性能客户和服务端付出额外努力。

- 传输机制无关性

Ice 目前采用了 TCP/IP 和 UDP 作为传输协议。客户和服务端代码都不需要了解底层的传输机制（你可以通过一个配置参数选择所需的传输机制）。

- 位置和服务端透明性

Ice run time 会负责定位对象，并管理底层的传输机制，比如打开和关闭连接。客户与服务端之间的交互显得像是无连接的。如果在客户祈用操作时，服务端没有运行，你可以通过 IcePack 让它们按需启动。服务端可以迁移到不同的物理地址，而不会使客户持有的代理失效，而客户完全不知道对象实现是怎样分布在多个服务端进程上的。

- 安全性

通过 SSL 强加密，可以使客户和服务端完全安全地进行通信，这样，应用可以使用不安全的网络安全地进行通信。你可以使用 Glacier 穿过防火墙，实现安全的请求转发，并且完全支持回调。

- 内建的持久机制

使用 Freeze，创建持久的对象实现变成了一件微不足道的事情。Ice 提供了对高性能数据库 Berkeley DB[18] 的内建支持。

- 开放源码

Ice 的源码是开放的。尽管要使用 Ice 平台，并不一定要阅读源码，通过源码你可以了解各种事情是怎样实现的，或把这些代码移植到新的操作系统上。

总而言之，Ice 提供了一流的分布式计算开发和部署环境，比我们所知道的其他任何平台都更完整。

2.5 与 CORBA 的对比

显而易见，Ice 采用的许多思想也能在 CORBA 及以前的一些分布式计算平台（比如 DCE[14]）中找到。在有些方面，Ice 与 CORBA 非常接近，而在另外一些方面，它们的差异则意义深远，并且在架构上有着广泛的影响。如果你曾经使用过 CORBA，了解这些差异十分重要。

2.5.1 对象模型的差异

尽管从表面看来，Ice 对象模型与 CORBA 对象模型是一样的，但它们在重要方面却有所不同。

类型系统

Ice 对象和 CORBA 对象一样，都只有一个派生层次最深的（most derived）主接口。但 Ice 对象可以提供其他接口作为 facets。重要的是要注意到，一个 Ice 对象的所有 facets 都具有相同的对象标识，也就是说，客户看到的是具有多个接口的单个对象，而不是看到多个对象、每个对象有不同的接口。

facets 提供了极大的架构灵活性。特别地，它们为版本管理问题提供了一种解决途径：你可以简单地给已经存在的对象增加新的 facet，轻松地扩展某个服务器的功能，而不会破坏已有的、已经部署的客户。

代理语义

Ice 代理（CORBA 对象引用的等价物）不是不透明的。客户只要知道对象的类型和标识，无需其他系统组件的支持，就可以创建出代理（在使用间接绑定时，不必了解对象的传输地址）。

允许客户按需创建代理有许多好处：

- 客户无需询问外部的查找服务，比如命名服务，就能够创建代理。实际上，对象标识和对象的名字被认为是同一事物。这样能够消除命名服务

的内容与实际情况失去同步所可能带来的问题；同时，为了让客户和服务器正常工作、必须正常运转的系统组件的数目也会减少。

- 通过创建所需的初始对象的代理，客户可以轻松地进行自引导（bootstrap）。这样就无需使用单独的引导服务了。
- 不需要对序列化代理进行不同的编码。一种统一的表示就足够了，而这种表示是人可以阅读的。这样就避免了 CORBA 的三种不同的对象引用编码（IOR、corbaloc，以及 corbaname）所带来的各种复杂问题。

开发者多年使用 CORBA 的经验表明，对象引用的不透明性很成问题：它不仅需要更加复杂的 API 和运行时支持，还会妨碍我们构建现实的系统。为此，CORBA 增加了像 corbaloc 和 corbaname 这样的机制，以及用于进行引用比较的 is_equivalent 和 hash 操作（这些操作的定义有问题）。所有这些机制都降低了对对象引用的不透明性，但 CORBA 平台的其他部分仍试图维持引用是不透明的这样一个错觉。结果，开发者在两方面所得的东西都是最糟的：引用既不是完全不透明的，也不是完全透明的——这样所带来的混乱和复杂性相当大。

对象标识

Ice 对象模型假定对象标识在任何地方都是唯一的（但并没有把这个要求强加给应用开发者）。这种对象标识的主要好处是，你可以迁移服务器，也可以把多个不同服务器中的对象合并进一个服务器，而不用考虑名字冲突的问题：如果每个 Ice 对象都具有唯一的标识，它就不可能与另外的域中的对象的标识发生冲突。

Ice 对象模型还使用了强对象标识：使用本地的客户端操作，你就能确定两个代理表示的是否是同一个对象（在 CORBA 中，要进行可靠的标识比较，你必须祈用远地对象上的操作）。本地标识比较要高效得多，而且对于有些应用领域而言（比如分布式事务服务），这样的比较也至关重要。

2.5.2 平台支持的差异

取决于你阅读的是哪种规范，CORBA 提供了 Ice 所提供的许多服务。例如，CORBA 支持异步方法祈用，并且还通过组件模型、支持某种形式的多接口。但问题是，你通常不可能在某一种实现中找到这些特性。有太多 CORBA 规范或者是可选的，或者没有被广泛实现，所以作为开发者，你通常必须决定不使用哪些特性。

下面的一些 Ice 特性没有直接的 CORBA 等价物：

- 异步方法分派（AMD）

CORBA API 没有提供任何机制来挂起服务器中的某个操作的处理、释放控制线程，并在后面恢复该操作的处理。

- 安全性

尽管在 CORBA 规范中有许多与安全性有关的内容，其中大部分特性至今没有实现。特别地，CORBA 至今没有给出实际的解决方案、让 CORBA 与防火墙共存。

- 协议特性

Ice 协议提供了双向支持。要让回调穿过防火墙，这种特性是一种基本需求（CORBA 曾经规定了一种双向协议，但在技术上有问题，并且，据我们所知，从未得以实现）。此外，Ice 还允许你使用 UDP 和 TCP，所以在可靠的网络（局域网）上，事件分发可以极其高效，并且可以很轻。CORBA 不支持把 UDP 用作传输机制。

Ice 协议的另一个重要特性是，在线路上，所有消息和数据都是完全封装起来的。这能够让 Ice 极其高效地实现像 IceStorm 这样的服务，因为，要转发数据，不需要进行解编和重整编。对于协议桥（比如 Glacier）的部署而言，封装也很重要，因为你无需用针对特定类型的信息对桥进行配置。

2.5.3 复杂性上的差异

CORBA 被公认是一个大而复杂的平台。这在很大程度上是 CORBA 的标准化方式造成的：各种决策是根据集体意见和多数表决做出的。这实际上意味着，在对某种新技术进行标准化时，达成一致的唯一途径是，把所有各方所宠爱的特性都容纳进来。结果就产生了这样的规范：大而复杂，要负担各种冗余的或无用的特性。所有这些复杂性继而又会带来大而低效的实现。规范的复杂性反映在 CORBA 的各种 API 的复杂性上：即使是有多年经验的专家也仍然需要把参考文档放在手边，而且，由于这样的复杂性，应用常常会受到许多潜在的 bug 的折磨，这些 bug 会在部署之后才暴露出来。

CORBA 的对象模型进一步加大了 CORBA 的复杂性。例如，不透明的对象引用导致开发者必须使用命名服务，因为客户必须通过某种途径访问对象引用。这继而要求开发者学习又一种 API，并且配置和部署又一种服务，而使用 Ice 对象模型，开发者从一开始就无需使用命名服务。

就复杂性而言，CORBA 最为臭名昭著的一个方面是 C++ 映射。CORBA C++ API 极其晦涩难懂；特别地，许多开发者都无法忍受这种映射带来的内存管理问题。而实现这种 C++ 映射所需的代码既不会特别少，也

不会高效；产生的二进制代码本来可以更小，并且在运行时占用较少内存。如果你曾经通过 C++ 使用过 CORBA，你将会欣赏 Ice C++ 映射的简单、高效，以及与 STL 的整洁的集成。

与 CORBA 相比，Ice 首先是一个简单的平台。Ice 的设计者为了挑选出够用的最小特性集而煞费苦心：你可以做你想做的每一件事情，你也可以用最小、最简单的 API 去做这些事情。随着你开始使用 Ice，你会欣赏这样的简单性的。我们因此能够轻松地学习和理解 Ice 平台，用更短的时间进行开发，并且使部署后的应用缺陷更少。与此同时，Ice 并没有牺牲特性：你可以用 Ice 完成你能用 CORBA 完成的每一件事情，但工作量更小，代码更少，复杂性也更小。我们将此视为 Ice 相对其他任何中间件平台而言最引人注目的优点：事情很简单，事实上，它们是如此简单，以致于只需用几天时间学习 Ice，你就可能已经在用它开发工业级的分布式应用了。

第 3 章

Hello World 应用

3.1 本章综述

在这一章，我们将考察怎样分别使用 C++ 和 Java 语言，创建一个非常简单的客户—服务器应用。这个应用提供远地打印功能：客户发送要打印的文本给服务器，再由服务器把文本发给打印机。

为简单起见（同时也因为我们不想处理各种平台的假脱机打印系统的特质），我们的打印程序只是把文本打印到终端，而不是真正的打印机。这样做的损失并不大：这个练习的目的是说明客户怎样与服务器通信；一旦控制线程到达服务器应用代码，这些代码当然可以做它想做的任何事情（包括把文本发给真正的打印机）。怎样做这些事情与 Ice 无关，所以在这里是不相干的。

注意，我们现在不会解释源码中的大量细节。我们的意图是向你说明，怎样着手使用 Ice，并让你感觉一下 Ice 开发环境；我们将在本书的整个余下部分给出所有的细节。

3.2 编写 Slice 定义

编写任何 Ice 应用的第一步都是要编写一个 Slice 定义，其中含有应用所用的各个接口。我们为的小打印应用编写了这样的 Slice 定义：

```
interface Printer
{
    void printString(string s);
};
```

我们把这段文本保存在叫作 `Printer.ice` 的文件中。

我们的 `Slice` 定义含有一个接口，叫作 `Printer`。目前，我们的接口非常简单，只提供了一个操作，叫作 `printString`。`printString` 操作接受一个串作为它唯一的输入参数；这个串的文本将会出现在（可能在远地的）打印机上。

3.3 编写使用 C++ 的 Ice 应用

这一节将说明怎样创建一个使用 C++ 的 Ice 应用。等价的 Java 版本将在第 41 页的 3.4 节给出。

编译用于 C++ 的 Slice 定义

要创建我们的 C++ 应用，第一步是要编译我们的 `Slice` 定义，生成 C++ 代理和骨架。在 UNIX 上，你可以这样编译定义：

```
$ slice2cpp Printer.ice
```

`slice2cpp` 编译器根据这个定义生成两个 C++ 源文件：`Printer.h` 和 `Printer.cpp`。

- `Printer.h`

`Printer.h` 头文件含有与我们的 `Printer` 接口的 `Slice` 定义相对应的 C++ 类型定义。在客户和服务端源码中必须包括这个头文件。

- `Printer.cpp`

`Printer.cpp` 文件含有我们的 `Printer` 接口的源码。所生成的源码同时为客户和服务端提供针对特定类型的运行时支持。例如，它包含了在客户端整编参数数据（传给 `printString` 操作的串）的代码，以及在服务端解编数据的代码。

我们必须编译 `Printer.cpp` 文件，并把它链接进客户和服务端。

编写和编译服务器

服务器的源码只有几行，下面给出了其完整代码：

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;

class PrinterI : public Printer {
public:
    virtual void printString(const string & s,
                             const Ice::Current &);
};

void
PrinterI::
printString(const string & s, const Ice::Current &)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "default -p 10000");
        Ice::ObjectPtr object = new PrinterI;
        adapter->add(object,
            Ice::stringToIdentity("SimplePrinter"));
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception & e) {
        cerr << e << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

对于一个只打印一个串的服务器而言，这里的代码似乎有很多。不要担心这一点：大多数代码都是不会变化的公式化代码。对于这个非常简单的服务器而言，其代码几乎就由这些公式化代码组成。

每个 Slice 源文件的一开始都有一条用于包括 Ice.h 的指令，在 Ice.h 中包含了 Ice run time 的各种定义。我们还包括了由 Slice 编译器生成的 Printer.h，其中含有我们的打印机接口的 C++ 定义；为了使以后的代码保持简洁，我们还导入了 std 名字空间的内容：

```
#include <Ice/Ice.h>
#include <Printer.h>
```

```
using namespace std;
```

我们的服务器实现了一个打印机 servant，其类型是 PrinterI。我们查看 Printer.h 中的生成的代码，发现了以下内容（为去掉无关细节，进行了一下整理）：

```
class Printer : virtual public Ice::Object {
public:
    virtual void printString(const std::string &,
                           const Ice::Current & = Ice::Current()
                           ) = 0;
};
```

Printer 骨架类定义是由 Slice 编译器生成的（注意，printString 是纯虚方法，所以这个骨架类不能被实例化）。我们的 servant 类继承自骨架类，提供了 printString 纯虚方法的实现（按照惯例，我们用 I 后缀表示这个类实现了一个接口）。

```
class PrinterI : public Printer {
public:
    virtual void printString(const string & s,
                           const Ice::Current &);
};
```

printString 方法的实现很简单：它会简单地把它的串参数写到 stdout：

```
void
PrinterI::
printString(const string & s, const Ice::Current &)
{
    cout << s << endl;
}
```

注意，`printString` 有第二个参数，类型是 `Ice::Current`。从 `Printer::printString` 的定义你可以看到，`Slice` 编译器会为此形参生成缺省的实参，所以在我们的实现中可以不使用它（我们将在 16.5 节考察 `Ice::Current` 的用途）。

接下来是服务器的主程序。注意这段代码的总体结构：

```
int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {

        // Server implementation here...

    } catch (const Ice::Exception & e) {
        cerr << e << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

`main` 的主体声明了两个变量 `status` 和 `ic`。`status` 变量含有程序的退出状态，而类型为 `Ice::Communicator` 的 `ic` 变量含有 `Ice run time` 的主句柄。

在声明的后面是一个 `try` 块，我们把所有的服务器代码都放在其中；然后是两个 `catch` 处理器。第一个处理器捕捉 `Ice run time` 可能抛出的所有异常，其意图是，如果代码在任何地方遇到意料之外的 `Ice` 运行时异常，栈会一直回绕到 `main`，打印出异常，然后把失败返回给操作系统。第二个处理器捕捉串常量，其意图是，如果我们在代码某处遇到致命错误，我们可以简单地抛出带有出错消息的串文本。这也会使栈回绕到 `main`，打印出出错消息，然后把失败返回给操作系统。

在 `try` 块的后面，我们看到一行清理代码调用通信器的 `destroy` 方法（前提是通信器进行过初始化）。清理调用之所以在 `try` 块的外部，原因是：不管代码是正常终止，还是由于异常而终止，我们都必须确保 `Ice run time` 得以执行结束工作（`finalized`）¹。

我们的 `try` 块的主体含有实际的服务器代码：

```
ic = Ice::initialize(argc, argv);
Ice::ObjectAdapterPtr adapter
    = ic->createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000");
Ice::ObjectPtr object = new PrinterI;
adapter->add(object,
    Ice::stringToIdentity("SimplePrinter"));
adapter->activate();
ic->waitForShutdown();
```

这段代码包含以下步骤：

1. 我们调用 `Ice::initialize`，初始化 Ice run time（我们之所以把 `argc` 和 `argv` 传给这个调用，是因为服务器可能有 run time 感兴趣的命令行参数；就这个例子而言，服务器不需要任何命令行参数）。`initialize` 调用返回的是一个智能指针，指向一个 `Ice::Communicator` 对象，这个指针是 Ice run time 的主句柄。
2. 我们调用 `Communicator` 实例上的 `createObjectAdapterWithEndpoints`，创建一个对象适配器。我们传入的参数是 `"SimplePrinterAdapter"`（适配器的名字）和 `"default -p 10000"`，后者是要适配器用缺省协议（TCP/IP）在端口 10000 处侦听到来的请求。
3. 这时，服务器端 run time 已经初始化，我们实例化一个 `PrinterI` 对象，为我们的 `Printer` 接口创建一个 `servant`。
4. 我们调用适配器的 `add`，告诉它有了一个新的 `servant`；传给 `add` 的参数是我们刚才实例化的 `servant`，再加上一个标识符。在这里，`"SimplePrinter"` 串是 `servant` 的名字（如果我们有多个打印机，每个打印机都可以有不同的名字，更正确的说法是，都有不同的对象标识）。
5. 接下来，我们调用适配器的 `activate` 方法激活适配器（适配器一开始是在扣留（holding）状态创建的；这种做法在下面这样的情况下很有用：我们多个 `servant`，它们共享同一个适配器，而在所有 `servant` 实例化之前我们不想处理请求）。一旦适配器被激活，服务器就会开始处理来自客户的请求。
6. 最后，我们调用 `waitForShutdown`。这个方法挂起发出调用的线程，直到服务器实现终止为止——或者通过发出一个调用关闭 run time，

1. 如果在程序退出之前没有调用通信器的 `destroy`，就会造成不确定的行为。

或者是对某个信号作出响应（目前，当我们不再需要服务器时，我们会简单地在命令行上中断它）。

注意，尽管这里的代码不算少，但它们对所有的服务器都是一样的。你可以把这些代码放在一个辅助类里，然后就无需再为它费心了（Ice 提供了一个这样的辅助类，叫作 `Ice::Application`，参见 10.3.1 节）。就实际的应用代码而言，服务器只有几行代码：六行代码定义 `PrinterI` 类，再加上三² 行代码实例化一个 `PrinterI` 对象，并向对象适配器登记它。

假定我们的服务器代码放在一个叫作 `Server.cpp` 的文件中，我们可以这样编译它：

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Server.cpp
```

这条命令编译我们的应用代码，以及 Slice 编译器生成的代码。我们假定 `ICE_HOME` 环境变量被设成 Ice run time 所在的顶层目录（例如，如果你把 Ice 安装在 `/opt/Ice` 中，就把 `ICE_HOME` 设成该路径）。取决于你的平台，你可能需要给编译器增加额外的包括指令或其他选项（比如增加包括 `STLport` 头的指令，或是对模板实例化进行控制）；要了解详情，请参考随 Ice 发布的演示程序。

最后，我们需要把服务器链接成可执行程序：

```
$ c++ -o server Printer.o Server.o \  
-L$ICE_HOME/lib -lIce -lIceUtil
```

再一次，取决于你的平台，你需要链接的库可能会更多。随 Ice 发布的演示程序含有所有的细节。在此，需要提及一个要点：Ice run time 是在两个库中：`libIce` 和 `libIceUtil`。

编写和编译客户

客户代码看起来与服务器非常像。下面是完整的代码：

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;

int
main(int argc, char * argv[])
{
    int status = 0;
```

2. 哦，是两行，真的：打印空间的限制迫使我们更频繁地折行，比你在实际的源文件中的折行次数更多。

```

Ice::CommunicatorPtr ic;
try {
    ic = Ice::initialize(argc, argv);
    Ice::ObjectPrx base = ic->stringToProxy(
        "SimplePrinter:default -p 10000");
    PrinterPrx printer = PrinterPrx::checkedCast(base);
    if (!printer)
        throw "Invalid proxy";

    printer->printString("Hello World!");
} catch (const Ice::Exception & ex) {
    cerr << ex << endl;
    status = 1;
} catch (const char * msg) {
    cerr << msg << endl;
    status = 1;
}
if (ic)
    ic->destroy();
return status;
}

```

注意，总体的代码布局与服务器是一样的：我们包括 **Ice run time** 的头和 **Slice** 编译器生成的头，我们用同样的 **try** 块和 **catch** 处理器处理错误。

try 块中的代码所做的事情是：

1. 和在服务器中一样，我们调用 `Ice::initialize` 初始化 **Ice run time**。
2. 下一步是获取远地打印机的代理。我们调用通信器的 `stringToProxy` 创建一个代理，所用参数是 `"SimplePrinter:default -p 10000"`。注意，这个串包含的是对象标识和服务器所用的端口号（显然，在应用中硬编码对象标识和端口号，是一种糟糕的做法，但它目前很有效；我们将在第 20 章看到在架构上更加合理的做法）。
3. `stringToProxy` 返回的代理的类型是 `Ice::ObjectPrx`，这种类型位于接口和类的继承树的根部。但要实际与我们的打印机交谈，我们需要的是 **Printer** 接口、而不是 **Object** 接口的代理。为此，我们需要调用 `PrinterPrx::checkedCast` 进行向下转换。这个方法会发送一条消息给服务器，实际询问“这是 **Printer** 接口的代理吗？”如果是，这个调用就会返回 **Printer** 的一个代理；如果代理代表的是其他类型的接口，这个调用就会返回一个空代理。

4. 我们测试向下转换是否成功，如果不成功，就抛出出错消息，终止客户。
5. 现在，我们在我们的地址空间里有了一个活的代理，可以调用 `printString` 方法，把享誉已久的 "Hello World!" 串传给它。服务器会在它的终端上打印这个串。

客户的编译和链接看起来与服务器很像：

```
$ c++ -I. -I$ICE_HOME/include -c Printer.cpp Client.cpp  
$ c++ -o client Printer.o Client.o -L$ICE_HOME/lib -lIce -lIceUtil
```

运行客户和服务

要运行客户和服务，我们首先要在一个单独的窗口中启动服务器：

```
$ ./server
```

我们在这时不会看到任何东西，因为服务器会简单地等待客户与它连接。我们在另外一个窗口中运行客户：

```
$ ./client  
$
```

客户会运行并退出，不产生任何输出；但在服务器窗口中，我们会看到打印机产生的 "Hello World!"。要终止服务器，我们目前的做法是在命令行上中断它（我们将在 10.3.1 节看到更干净的服务器终止方式）。

如果出现任何错误，客户会打印一条出错消息。例如，如果我们没有先启动服务器就运行客户，我们会得到：

```
Network.cpp:471: Ice::ConnectFailedException:  
connect failed: Connection refused
```

注意，要想成功运行客户和服务，你必须设置一些取决于平台的环境变量。例如，在 Linux 上，你需要把 Ice 库目录增加到你的 **LD_LIBRARY_PATH**。请看一看随 Ice 发布的演示应用，以了解你的平台的各种相关细节。

3.4 编写使用 Java 的 Ice 应用

这一节说明怎样编写使用 Java 的 Ice 应用。在第 34 页的 3.3 节给出了等价的 C++ 版本。

编写用于 Java 的 Slice 定义

要创建我们的 Java 应用，第一步是要编译我们的 Slice 定义，生成 C++ 代理和骨架。在 UNIX 上，你可以这样编译定义³：

```
$ mkdir generated
$ slice2java --output-dir generated Printer.ice
```

--output-dir 选项告诉编译器，要把生成的文件放进 generated 目录。这样，生成的文件就不会搅乱工作目录。**slice2java** 编译器根据这个定义生成一些 Java 源文件。我们目前无需关注这些文件的确切内容——它们包含的是编译器生成的代码，与我们在 Printer.ice 中定义的 Printer 接口相对应。

编写和编译服务器

要实现我们的 Printer 接口，我们必须创建一个 servant 类。按照惯例，servant 类的名字是它们的接口的名字加上一个 I 后缀，所以我们的 servant 类叫作 PrinterI，并放在 PrinterI.java 源文件中：

```
public class PrinterI extends _PrinterDisp {
    public void
    printString(String s, Ice.Current current)
    {
        System.out.println(s);
    }
}
```

PrinterI 类继承自叫作 _PrinterDisp 的基类。这个基类由 **slice2java** 编译器生成，是一个抽象类，其中含有一个 printString 方法，其参数是打印机要打印的串，以及类型为 Ice.Current 的对象（目前我们会忽略 Ice.Current 参数。我们将在 16.5 节详细讨论它的用途）。我们的 printString 方法的实现会简单地把它的参数写到终端。

服务器代码的其余部分在一个叫作 Server.java 的源文件中，下面给出了其完整代码：

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
```

3. 在本书中，只要我们给出 UNIX 命令，我们都假定是在使用 Bourne 或 Bash shell。用于 Windows 的命令在本质上是一样的，所以没有给出。

```

Ice.Communicator ic = null;
try {
    ic = Ice.Util.initialize(args);
    Ice.ObjectAdapter adapter
        = ic.createObjectAdapterWithEndpoints(
            "SimplePrinterAdapter", "default -p 10000");
    Ice.Object object = new PrinterI();
    adapter.add(
        object,
        Ice.Util.stringToIdentity("SimplePrinter"));
    adapter.activate();
    ic.waitForShutdown();
} catch (Ice.LocalException e) {
    e.printStackTrace();
    status = 1;
} catch (Exception e) {
    System.err.println(e.getMessage());
    status = 1;
} finally {
    if (ic != null)
        ic.destroy();
}
System.exit(status);
}
}

```

注意这段代码的总体结构:

```

public class Server {
    public static void
    main(String[] args) {
        int status = 0;
        Ice.Communicator ic = null;
        try {

            // Server implementation here...

        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        } finally {
            if (ic != null)
                ic.destroy();
        }
    }
}

```

```

    }
    System.exit(status);
}
}

```

main 的主体含有一个 try 块，我们把所有的服务器代码都放在其中；然后是两个 catch 处理器。第一个处理器捕捉 Ice run time 可能抛出的所有异常，其意图是，如果代码在任何地方遇到意料之外的 Ice 运行时异常，栈会一直退回到 main，打印出异常，然后把失败返回给操作系统。第二个处理器捕捉串常量，其意图是，如果我们在代码某处遇到致命错误，我们可以简单地抛出带有出错消息的串文本。这也会使栈退回到 main，打印出出错消息，然后把失败返回给操作系统。

这段代码会在退出之前销毁通信器（如果曾经成功创建过）。要使 Ice run time 正常结束，这样做是必需的：程序必须调用它所创建的任何通信器的 destroy；否则就会产生不确定的行为。我们把对 destroy 的调用放进 finally 块，这样，不管前面的 try 块中发生什么异常，通信器都保证会正确销毁。

我们的 try 块的主体含有实际的服务器代码：

```

ic = Ice.Util.initialize(args);
Ice.ObjectAdapter adapter
    = ic.createObjectAdapterWithEndpoints(
        "SimplePrinterAdapter", "default -p 10000");
Ice.Object object = new PrinterI();
adapter.add(
    object,
    Ice.Util.stringToIdentity("SimplePrinter"));
adapter.activate();
ic.waitForShutdown();

```

这段代码包含了以下步骤：

1. 我们调用 Ice.Util.initialize 初始化 Ice run time（我们之所以把 args 传给这个调用，是因为服务器可能有 run time 感兴趣的命令行参数；就这个例子而言，服务器不需要任何命令行参数）。对 initialize 的调用返回的是一个 Ice::Communicator 引用，这个引用是 Ice run time 的主句柄。
2. 我们调用 Communicator 实例上的 createObjectAdapterWithEndpoints，创建一个对象适配器。我们传入的参数是 "SimplePrinterAdapter"（适配器的名字）和 "default -p 10000"，后者是要适配器用缺省协议（TCP/IP）在端口 10000 处侦听到来的请求。

3. 这时，服务器端 run time 已经初始化，我们实例化一个 PrinterI 对象，为我们的 Printer 接口创建一个 servant。
4. 我们调用适配器的 add，告诉它有了一个新的 servant；传给 add 的参数是我们刚才实例化的 servant，再加上一个标识符。在这里，"SimplePrinter" 串是 servant 的名字（如果我们有多个打印机，每个打印机都会有不同的名字，更正确的说法是，都会有不同的对象标识）。
5. 接下来，我们调用适配器的 activate 方法激活适配器（适配器一开始是在扣留（holding）状态创建的；这种做法在下面这样的情况下很有用：如果我们有多个 servant，它们共享同一个适配器，而在所有 servant 实例化之前我们不想处理请求）。一旦适配器被激活，服务器就会开始处理来自客户的请求。
6. 最后，我们调用 waitForShutdown。这个方法挂起发出调用的线程，直到服务器实现终止为止——或者通过发出一个调用关闭 run time，或者是对某个信号作出响应（目前，当我们不再需要服务器时，我们会简单地在命令行上中断它）。

注意，尽管这里的代码不算少，但它们对所有的服务器都是一样的。你可以把这些代码放在一个辅助类里，然后就无需再为它费心了（Ice 提供了一个这样的辅助类，叫作 `Ice::Application`，参见 10.3.1 节）。就实际的应用代码而言，服务器只有几行代码：六行代码定义 PrinterI 类，再加上三⁴行代码实例化一个 PrinterI 对象，并向对象适配器登记它。

我们可以这样编译服务器代码：

```
$ mkdir classes
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar \
-source 1.4 Server.java PrinterI.java generated/*.java
```

这条命令编译我们的应用代码，以及 Slice 编译器生成的代码。我们假定 `ICEJ_HOME` 环境变量被设成 Ice run time 所在的顶层目录（例如，如果你把 Ice 安装在 `/opt/Ice` 中，就把 `ICEJ_HOME` 设成该路径）。注意，Ice for Java 使用了 `ant` 构建环境来控制源码的构建（`ant` 与 `make` 类似，但对 Java 应用而言要更灵活）要了解怎样使用这个工具，你可以查看随 Ice 发布的演示代码。

4. 哦，是两行，真的：打印空间的限制迫使我们更频繁地折行，比你在实际的源文件中的折行次数更多。

编写和编译客户

客户代码在 `Client.java` 中，看起来与服务器非常类似。下面是完整的代码：

```
public class Client {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectPrx base = ic.stringToProxy(
                "SimplePrinter:default -p 10000");
            PrinterPrx printer
                = PrinterPrxHelper.checkedCast(base);
            if (printer == null)
                throw new Error("Invalid proxy");

            printer.printString("Hello World!");
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        } finally {
            if (ic != null)
                ic.destroy();
        }
        System.exit(status);
    }
}
```

注意，总体的代码布局与服务器是一样的：我们包括 `Ice run time` 的头和 `Slice` 编译器生成的头，我们用同样的 `try`、`catch` 以及 `finally` 块处理错误。`try` 块中的代码所做的事情是：

1. 和在服务器中一样，我们调用 `Ice::initialize` 初始化 `Ice run time`。
2. 下一步是获取远地打印机的代理。我们调用通信器的 `stringToProxy` 创建一个代理，所用参数是 `"SimplePrinter:default -p 10000"`。注意，这个串包含的是对象标识和服务器所用的端口号

(显然, 在应用中硬编码对象标识和端口号, 是一种糟糕的做法, 但它目前很有效; 我们将在第 20 章看到在架构上更加合理的做法)。

3. `stringToProxy` 返回的代理的类型是 `Ice::ObjectPrx`, 这种类型位于接口和类的继承树的根部。但要实际与我们的打印机交谈, 我们需要的是 `Printer` 接口、而不是 `Object` 接口的代理。为此, 我们需要调用 `PrinterPrxHelper.checkedCast` 进行向下转换。这个方法会发送一条消息给服务器, 实际询问 “这是 `Printer` 接口的代理吗?” 如果是, 这个调用就会返回 `Printer` 的一个代理; 如果代理代表的是其他类型的接口, 这个调用就会返回一个空代理。
4. 我们测试向下转换是否成功, 如果不成功, 就抛出出错消息, 终止客户。
5. 现在, 我们在我们的地址空间里有了一个活的代理, 可以调用 `printString` 方法, 把享誉已久的 “Hello World!” 串传给它。服务器会在它的终端上打印这个串。

客户的编译看起来与服务器很像:

```
$ javac -d classes -classpath classes:$ICEJ_HOME/lib/Ice.jar \
-source 1.4 Client.java PrinterI.java generated/*.java
```

运行客户和服务

要运行客户和服务, 我们首先要在一个单独的窗口中启动服务器:

```
$ java Server
```

我们在这时不会看到任何东西, 因为服务器会简单地等待客户与它连接。我们在另外一个窗口中运行客户:

```
$ java Client
$
```

客户会运行并退出, 不产生任何输出; 但在服务器窗口中, 我们会看到打印机产生的 “Hello World! ”。要终止服务器, 我们目前的做法是在命令行上中断它 (我们将在第 12 章看到更干净的服务器终止方式)。

如果出现任何错误, 客户会打印一条出错消息。例如, 如果我们没有先启动服务器就运行客户, 我们会得到:

```
Ice.ConnectFailedException
    at IceInternal.Network.doConnect(Network.java:201)
    at IceInternal.TcpConnector.connect(TcpConnector.java:26)
    at
IceInternal.OutgoingConnectionFactory.create(OutgoingConnectionFactory.java:80)
    at Ice._ObjectDelM.setup(_ObjectDelM.java:251)
```

```
        at Ice.ObjectPrxHelper.__getDelegate(ObjectPrxHelper.java:
642)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:41)
        at Ice.ObjectPrxHelper.ice_isA(ObjectPrxHelper.java:30)
        at PrinterPrxHelper.checkedCast(Unknown Source)
        at Client.main(Unknown Source)
Caused by: java.net.ConnectException: Connection refused
        at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method
)
        at
sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:
518)
        at IceInternal.Network.doConnect(Network.java:173)
        ... 8 more
```

注意，要想成功运行客户和服务端，你的 **CLASSPATH** 必须包括 Ice 库目录和类目录，例如：

```
$ export CLASSPATH=$CLASSPATH:./classes:$ICEJ_HOME/lib/Ice.jar
```

请看一看随 Ice 发布的演示应用，以了解你的平台的各种相关细节。

3.5 总结

本章介绍了一个非常简单（但却完整）的客户和服务端。我们已经看到，编写 Ice 应用涉及以下步骤：

1. 编写 Slice 定义并编译它。
2. 编写服务端并编译它。
3. 编写客户并编译它。

如果有人已经编写了服务端，而你只是在编写服务端，你无需编写 Slice 定义，只需编译它就可以了（显然，在这种情况下你无需编写服务端）

现在不要担心有大量内容你还不明白。本章的目的是让你了解一下开发过程，而不是对 Ice 的各种 API 详加解释。我们将在本书的余下部分涵盖所有的细节。

第二部分

Ice 核心概念

第 4 章

Slice 语言

4.1 本章综述

在这一章我们将介绍 Slice 语言。我们首先讨论 Slice 的角色与用途，解释与语言无关的规范是怎样被编译到特定的实现语言、从而创建实际的实现的。4.8 节与 4.9 节涵盖接口、操作、异常，以及继承等核心的 Slice 概念。这些概念对分布式系统的行为有着深远的影响，你应该详细阅读。

4.2 引言

Slice¹（Specification Language for Ice）是一种用于使对象接口与其实现相分离的基础性抽象机制。Slice 在客户与服务器之间建立合约，描述应用所使用的各种类型及对象接口。这种描述与实现语言无关，所以编写客户所用的语言是否与编写服务器所用的语言相同，这没有什么关系。

Slice 定义由编译器编译到特定的实现语言。编译器把与语言无关的定义翻译成针对特定语言的类型定义和 API。开发者使用这些类型和 API 来提供应用功能，并与 Ice 交互。用于各种实现语言的翻译算法称为 *语言映射*（language mappings）。Ice 目前定义了 C++ 和 Java 的语言映射。

1. 尽管 Slice 是首字母缩写词，其发音与 "a slice of bread" 中的 "slice" 一样，是单音节的。

因为 Slice 描述的是接口和类型（不是实现），它是一种纯粹的描述性语言；你无法用 Slice 编写可执行语句。

Slice 定义关注的焦点是对象接口、这些接口所支持的操作，以及操作可能引发的异常。此外，Slice 还提供了一些用于对象持久的特性（参见第 21 章）。这需要相当多的支持机制；特别地，Slice 的相当一部分关注的是数据类型的定义。这是因为，只有在其类型用 Slice 进行了定义之后，数据才能在客户与服务器之间交换。你不能在客户与服务器之间交换任意的 C++ 数据，因为这可能会摧毁 Ice 的语言无关性。但是，你总能创建一种 Slice 类型定义，与你想要发送的 C++ 数据相对应，然后你就可以传送这种 Slice 类型了。

在此我们将介绍 Slice 的完整语法和语义。因为 Slice 的许多语法和语义都是以 C++ 和 Java 为基础的，我们将特别关注 Slice 与 C++ 或 Java 不同的部分，或是 Slice 以某种方式限制了等价的 C++ 或 Java 特性的部分。与 C++ 和 Java 特性相同的 Slice 特性通常会用例子来说明。

4.3 编译

Slice 编译器生成的源文件必须与应用代码相结合，才能产生客户和服务器的可执行程序。

开发过程的产出结果是可执行的客户程序及服务器程序。这两种可执行程序可以部署到任何地方，无论它们的目标环境使用的操作系统是否相同，也无论它们是否使用相同的语言实现。唯一的约束是宿主机器必须提供必要的运行时环境，比如任何必需的动态库；同时双方的宿主机器要能够建立连接。

4.3.1 客户与服务器使用同一种开发环境

图 4.1 说明客户和服务器都用 C++ 开发的情况。Slice 编译器根据源文件 `Printer.ice` 中的 Slice 定义生成了两个文件：一个头文件（`Printer.h`）和一个源文件（`Printer.cpp`）。

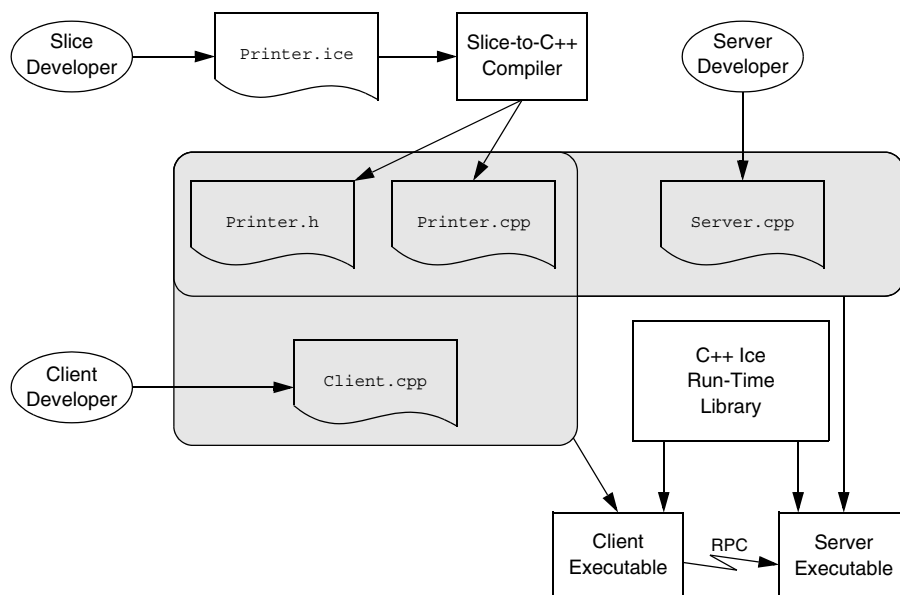


图 4.1. 在客户与服务器共享相同的开发环境时的开发过程

- `Printer.h` 头文件含有与 Slice 定义中所用的类型相对应的定义。在客户和服务器的源码中都会包括它，以确保客户和服务器就应用所用的类型和接口达成一致。
- `Printer.cpp` 源文件给客户提供一个 API，用于把消息发送给远地对象。客户源码（`Client.cpp`，由客户开发者编写）含有客户端的应用逻辑。生成的源码和客户代码都被编译并链接进可执行的客户程序中。

`Printer.cpp` 源文件包含的源码提供了一个向上调用接口，从 Ice run time 调用开发者编写的服务器代码；同时还提供了 Ice 的网络层与应用代码之间的连接。服务器实现文件（`Server.cpp`，由服务器开发者编写）含有服务器端应用逻辑（对象实现，用适当的术语说，叫

作 *servants*)。生成的源码和实现源代码都被编译并链接进可执行的服务器程序中。

为了获得必要的运行时支持，客户和服务器都会链接一个 Ice 库。

你并非只能编写一种客户和服务器实现。例如，你可以构建多个服务器，每个服务器都实现相同的接口，但使用不同的实现（例如，具有不同的性能特征）。多个这样的服务器实现可以共存于同一个系统中。这种方案提供了 Ice 中的一种基础性的伸缩机制：如果你发现，随着对象数目的增长，某个服务器进程开始陷入困境，你可以在另外的机器上运行另外一个具有相同接口的服务器。这样的联盟服务器在逻辑上是一种服务，但却分布在不同机器的许多进程上。联盟中的每个服务器都实现同样的接口，但充当的是不同对象实例的宿主（当然，各联盟服务器必须以某种方式确保它们在联盟中共享的任何数据库的一致性）。

Ice 还提供了对重复（replicated）服务器的支持。在这样的服务器中，多个服务器各自实现同一组对象实例。这能够改善性能和可伸缩性（因为可以在许多服务器上分摊客户负载），并提高冗余性（因为每个对象都在不止一个服务器中实现）。

4.3.2 客户和服务器使用不同的开发环境

如果客户和服务器是用不同的语言开发的，它们不能共享任何源文件或二进制组件。例如，用 Java 编写的客户不能包括 C++ 头文件。

图 4.2 说明客户用 Java 编写，而对应的服务器用 C++ 编写的情况。在这种情况下，客户和服务器开发者是完全独立的，分别使用自己的开发环

境和语言映射。客户和服务开发者之间的唯一链接是它们各自使用的 Slice 定义。

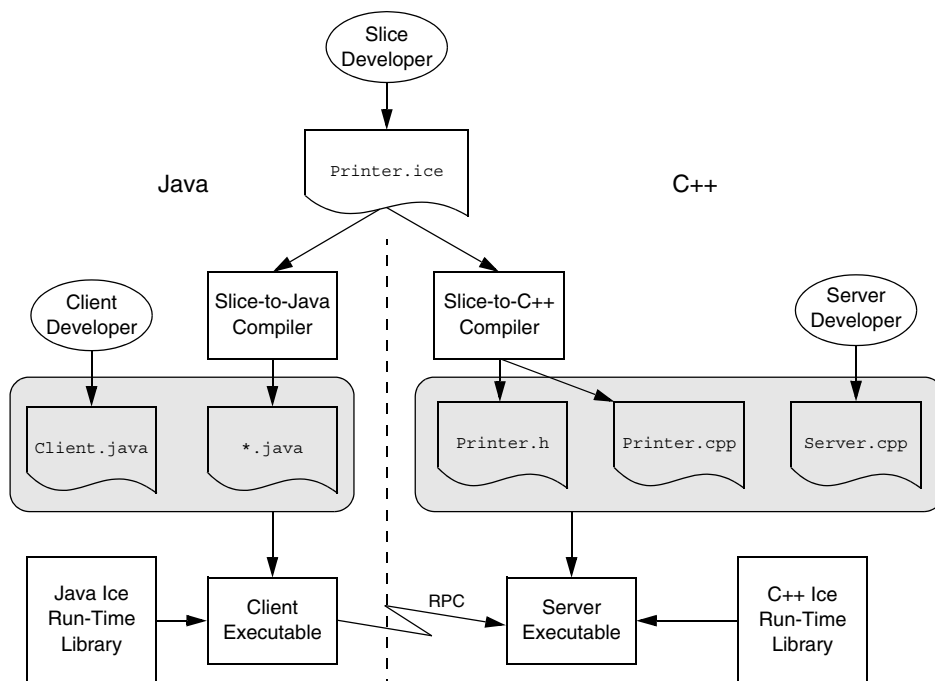


图 4.2. 使用不同的开发环境时的开发过程

在开发 Java 应用时，slice 编译器会创建一些文件，其名称取决于各种 Slice 成分的名称（在图 4.2 中这些文件总称为 *.java）。

4.4 源文件

Slice 为 Slice 源文件的命名和内容定义了一些规则。

4.4.1 文件命名

含有 Slice 定义的文件必须以 .ice 扩展名结尾，例如，Clock.ice 就是一个有效的文件名。编译器拒绝接受其他扩展名。

对于大小写不敏感的文件系统（比如 DOS），文件扩展名可以写成大写，也可以写成小写，所以 `clock.ICE` 是合法的。对于大小写敏感的文件系统（比如 UNIX），`clock.ICE` 是非法的（扩展名必须小写）。

4.4.2 文件格式

Slice 是一种形式自由的语言，所以你可以使用空格、横向和纵向制表符、换页，以及换行字符，按照你希望的任何方式安排代码的布局（空白字符是 `token` 分隔符）。Slice 不会把语义与定义的布局关联起来。你可以遵循我们在本书中的 Slice 例子中使用的风格。

4.4.3 预处理

Slice 支持 `#ifndef`、`#define`、`#endif`，以及 `#include` 预处理指令。它们的使用方式有严格的限制：

- 你只能把 `#ifndef`、`#define`，以及 `#endif` 指令用于创建双包括（double-include）块。例如：

```
#ifndef _CLOCK_ICE
#define _CLOCK_ICE

// #include directives here...

// Definitions here...

#endif _CLOCK_ICE
```

- `#include` 指令只能出现在 Slice 源文件的开头，也就是说，它们必须出现在其他所有 Slice 定义的前面。此外，在使用 `#include` 指令时，只允许使用 `<>` 语法来指定文件名，不能使用 `"`。例如：

```
#include <File1.ice>    // OK
#include "File2.ice"    // Not supported!
```

你不能把这些预处理指令用于其他目的，也不能使用其他的 C++ 预处理指令²（比如用 `\` 字符来连接行、`token` 粘贴，以及宏展开，等等）。

2. 在写下这段文字时，Slice 的预处理实际上是由 C++ 预处理器完成的。但在未来的版本中，可能会用其他机制取代预处理器；如果是那样，实际支持的就将只有这里所描述的预处理指令和用法

通过 `#include` 指令，Slice 定义可以使用其他源文件中定义的类型。Slice 编译器会解析源文件中的所有代码，包括通过 `#include` 包括的文件。但是，编译器只为在命令行上指定的顶层文件生成代码。你必须分别编译用 `#include` 包括的各个文件，为组成你的 Slice 定义的所有文件生成代码。

我们强烈建议你在所有的 Slice 定义中使用双 `include` 守卫（如第 58 页所示），防止多次包括同一文件。

4.4.4 定义次序

Slice 的成分，比如模块、接口，或类型定义，可以按照你喜欢的任何次序出现。但标识符在使用之前必须先声明。

4.5 词法规则

除了标识符的一些差异，Slice 的词法规则与 C++ 和 Java 的词法规则非常类似。

4.5.1 注释

在 Slice 定义里，既可以使用 C 的、也可以使用 C++ 的注释风格：

```
/*  
 * C-style comment.  
 */  
  
// C++-style comment extending to the end of this line.
```

4.5.2 关键字

Slice 使用了一些关键字，你必须以小写方式拼写它们。例如，`class` 和 `dictionary` 都是关键字，必须按照所示方式拼写。这个规则有两个例外：`Object` 和 `Local Object` 也是关键字，必须按照所示方式让首字母大写。在 Appendix A 中给出了完整的 Slice 关键字清单。

4.5.3 标识符

标识符以一个字母起头，后面可以跟任意数目的字母或数字。Slice 标识符被限制在 ASCII 字符范围内，不能包含非英语字母，比如 Å（如果支

持非 ASCII 标识符，要把 Slice 映射到不支持这种特性的目标语言会非常困难）。

与 C++ 标识符不同，Slice 标识符不能有以下划线。这种限制初看上去显得很苛刻，但却是必要的：保留下划线，各种语言映射就获得了一个名字空间，不会与合法的 Slice 标识符发生冲突。于是，这个名字空间可用于存放从 Slice 标识符派生的原生语言标识符，而不用担心其他合法的 Slice 标识符会碰巧与之相同，从而发生冲突。

大小写敏感性

标识符是大小写不敏感的，但大小写的拼写方式必须保持一致。例如，在一个作用域内，`TimeOfDay` 和 `TIMEOFDAY` 被认为是同一个标识符。但是，Slice 要求你保持大小写的一致性。在你引入了一个标识符之后，你必须始终一致地拼写它的大写和小写字母；否则，编译器就会将其视为非法而加以拒绝。这条规则之所以存在，是要让 Slice 既能映射到忽略标识符大小写的语言，又能映射到把大小写不同的标识符当作不同标识符的语言。

是关键字的标识符

你可以定义在一种或多种实现语言中是关键字的 Slice 标识符。例如，`switch` 是完全合法的 Slice 标识符，但也是 C++ 和 Java 的关键字。语言映射定义了一些规则来处理这样的标识符。要解决这个问题，通常要用一个前缀来使映射后的标识符不再是关键字。例如，Slice 标识符 `switch` 被映射到 C++ 的 `_cpp_switch`，以及 Java 的 `_switch`。

对关键字进行处理的规则可能会产生难以阅读的源码。像 `native`、`throw`，或 `export` 这样的标识符会与 C++ 或 Java（或两者）的关键字发生冲突。为了让你和别人生活得更轻松一点，你应该避免使用是实现语言的关键字的 Slice 标识符。要记住，以后 Ice 可能会增加除 C++ 和 Java 以外的语言映射。尽管期望你总结出所有流行的编程语言的所有关键字并不合理，你至少应该尽量避免使用常用的关键字。使用像 `self`、`import`，以及 `while` 这样的标识符肯定不是好主意。

转义的标识符

在关键字的前面加上一个反斜线，你可以把 Slice 关键字用作标识符，例如：

```
struct dictionary {      // Error!  
    // ...  
};  
  
struct \dictionary {     // OK  
    // ...  
};
```

反斜线会改变关键字通常的含义；在前面的例子中，`\dictionary` 被当作标识符 `dictionary`。转义机制之所以存在，是要让我们在以后能够在 `Slice` 中增加关键字，同时尽量减少对已有规范的影响：如果某个已经存在的规范碰巧使用了新引入的关键字，你只需在新关键字前加上反斜线，就能够修正该规范。注意，从风格上说，你应该避免用 `Slice` 关键字做标识符（即使反斜线转义允许你这么做）。

保留的标识符

`Slice` 为 `Ice` 实现保留了标识符 `Ice` 及以 `Ice`（任何大小写方式）起头的所有标识符。例如，如果你试图定义一个名为 `Icecream` 的类型，`Slice` 编译器会发出错误警告³。

以下面任何一种后缀结尾的 `Slice` 标识符也是保留的：`Helper`、`Holder`、`Prx`，以及 `Ptr`。`Java` 和 `C++` 语言映射使用了这些后缀，保留它们是为了防止在生成的代码中发生冲突。

3. 你可以用 `--ice` 编译器选项来抑制这一行为，这个选项的用途是允许你定义以 `Ice` 起头的标识符。但除非你正在编译 `Ice run time` 自身的 `Slice` 定义，否则不要使用这个选项。

4.6 基本的 Slice 类型

Slice 提供了一些内建的基本类型，如表 4.1. 所示。

Table 4.1. Slice 基本类型

类型	范围	尺寸
bool	false 或 true	未规定
byte	a	位
short	15 到 15	位
int	31 到 31	位
long	到 63	位
float	IEEE 单精度	位
double	IEEE 双精度	位
string	所有 Unicode 字符，除了所有位为零的字符	变长

a. 或 ，取决于语言映射

当在客户与服务器之间传送时，所有基本类型（除了 byte）的表示都可能发生变化。例如，在从 little-endian 机器发往 big-endian 机器时，long 值的各字节会交换。与此类似，如果串在从一种 EBCDIC 实现发往 ASCII 实现，它们的表示也会发生变化，而串的字符的尺寸可能会发生变化（不是所有架构都使用 8 位字符）。但这些改变对于程序员而言是透明的，而且会严格按照需要来实行。

4.6.1 整数类型

Slice 提供了整数类型 short、int，以及 long，这些类型的范围分别是 16 位、32 位，以及 64 位。注意，在有些架构上，这些类型都有可能映射到更宽的原生类型。另外还要注意，Slice 没有提供无符号类型（之所以做出这样的决定，是因为无符号类型难以映射到没有原生的无符号类型的语言，比如 Java。此外，无符号整数给一种语言带来的价值很少。要想更好地理解这个问题，参见 [9]）。

4.6.2 浮点类型

这些类型遵循 IEEE 的单精度和双精度浮点表示规范 [6]。如果某种实现不支持 IEEE 格式的浮点值，Ice run time 会把这样的值转换为原生的浮点表示（取决于原生浮点格式的容量，可能会损失精度，甚至是数量级）。

4.6.3 串

Slice 串使用的是 Unicode 字符集。唯一一个不能出现在串中的字符是零字符⁴。

Slice 数据模型没有 null 串的概念（在 C++ null 指针的意义上）。之所以做出这样的决定，是因为 null 串难以映射到不直接支持这一概念的语言（比如 Python）。不要设计接口、依赖于 null 串来表示“不在那里”的语义。如果你需要表示可选的串，用类（参见 4.9 节）、串的序列（参见 4.7.3 节），或空串（empty string）来表示 null 串的概念（当然，最后这种做法假定你的应用没有把空串另外用作合法的串值）。

4.6.4 布尔值

布尔值只有 false 和 true 两种值。如果有对应的原生布尔类型，语言映射就会使用该类型。

4.6.5 字节

Slice 的 byte 类型是一种（至少）8 位的类型，当在地址空间之间传送时，它保证不会发生任何改变。这一保证允许你交换二进制数据，在传送过程中这些数据不会被篡改。

4.7 用户定义的类型

除了提供内建的基本类型，Slice 还允许你定义复杂的类型：枚举（enumerations）、结构（structures）、序列（sequences），以及词典（dictionaries）。

4. 这个决定是对 C++ 的让步，在 C++ 中，使用标准库例程（比如 strlen 或 strcat）处理嵌有零字符的串是不可能的。

4.7.1 枚举

Slice 的枚举类型定义看起来就像是 C++ 的枚举类型定义：

```
enum Fruit { Apple, Pear, Orange };
```

这个定义引入了一种名为 `Fruit` 的类型，这是一种拥有自己权利的新类型。关于怎样把顺序值（ordinal values）赋给枚举符的问题，Slice 没有作出定义。例如，你不能假定，在各种实现语言中，枚举符 `Orange` 的值都是 2。Slice 保证枚举符的顺序值会从左至右递增，所以在所有实现语言中，`Apple` 都比 `Pear` 要小。

与 C++ 不同，Slice 不允许你控制枚举符的顺序值（因为许多实现语言不支持这种特性）：

```
enum Fruit { Apple = 0, Pear = 7, Orange = 2 }; // Syntax error
```

在实践中，只要你不地址空间之间传送枚举符的顺序值，你就不用管枚举符使用的值是多少。例如，发送值 0 给服务器来表示 `Apple` 可能会造成问题，因为服务器可能没有用 0 表示 `Apple`。相反，你应该就发送值 `Apple` 本身。如果在接收方的地址空间中，`Apple` 是用另外的顺序值表示的，Ice run time 会适当地翻译这个值。

与在 C++ 里一样，Slice 枚举符也会进入围绕它的名字空间，所以下面的定义是非法的：

```
enum Fruit { Apple, Pear, Orange };
enum ComputerBrands { Apple, IBM, Sun, HP }; // Apple redefined
```

Slice 不允许定义空的枚举。

4.7.2 结构

Slice 支持含有一个或多个有名称的成員的结构，这些成员可以具有任意类型，包括用户定义的复杂类型。例如：

```
struct TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};
```

与在 C++ 里一样，这个定义引入了一种叫作 `TimeOfDay` 的新类型。结构定义会形成名字空间，所以结构成员的名字只需在围绕它们的结构里是唯一的。

在结构内部，只能出现数据成员定义，这些定义必须使用有名字的类型。例如，你不可能在结构内定义结构：

```
struct TwoPoints {
    struct Point {          // Illegal!
        short x;
        short y;
    };
    Point coord1;
    Point coord2;
};
```

这个规则大体上适用于 **Slice**：类型定义不能嵌套（除了模块确实支持嵌套——参见 4.11 节）。其原因是，对于某些目标语言而言，嵌套的类型定义可能会难以实现，而且，即使能够实现，也会极大地使作用域解析规则复杂化。对于像 **Slice** 这样的规范语言而言，嵌套的类型定义并无必要——你总能以下面的方式编写上面的定义（这种方式在风格上也更加整洁）：

```
struct Point {
    short x;
    short y;
};

struct TwoPoints {          // Legal (and cleaner!)
    Point coord1;
    Point coord2;
};
```

4.7.3 序列

序列是变长的元素向量：

```
sequence<Fruit> FruitPlatter;
```

序列可以是空的——也就是说，它可以不包含元素；它也可以持有任意数量的元素，直到达到你的平台的内存限制。

序列包含的元素自身也可以是序列。这种设计使得你能够创建列表的列表：

```
sequence<FruitPlatter> FruitBanquet;
```

序列可用于构建许多种 **collection**，比如向量、列表、队列、集合、包（**bag**），或是树（次序是否重要要由应用决定；如果无视次序，序列充当的就是集合和包）。

序列的一种特别的用法已经成了惯用手法，即用序列来表示可选的值。例如，我们可能拥有一个 `Part` 结构，用于记录小汽车的零件的详细资料。这个结构可以记录这样的资料：零件名称、描述、重量、价格，以及其他详细资料。备件通常都有序列号，我们用一个 `long` 值表示。但有些零件，比如常用的螺丝钉，常常没有序列号，那么我们在螺丝钉的序列号字段里要放进什么内容？要处理这种情况，有这样一些选择：

- 用一个标记值，比如零，来指示“没有序列号”的情况。

这种方法是可行的，只要确实有标记值可用。尽管看起来不大可能有人把零用作零件的序列号，这并非是不可能的。而且，对于其他的值，比如温度值，在其类型的范围中的所有值都可能是合法的，因而没有标记值可用。

- 把序列号的类型从 `long` 变成 `string`。

串自己有内建的标记值，也就是空串，所以我们可以用空串来指示“没有序列号”的情况。这也是可行的，但却会让大多数人感到不快：我们不应该为了得到一个标记值，而把某种事物自然的数据类型变成 `string`。

- 增加一个指示符来指示序列号的内容是否有效：

```
struct Part {
    string name;
    string description;
    // ...
    bool    serialIsValid; // true if part has serial number
    long    serialNumber;
};
```

对于大多数人而言，这也让人讨厌，而且最终肯定会让你遇到麻烦：迟早会有程序员忘记在使用序列号之前检查它是否有效，从而带来灾难性的后果。

- 用序列来建立可选字段。

这种技术使用了下面的惯用手法：

```
sequence<long> SerialOpt;
```

```
struct Part {
    string    name;
```

```

    string    description;
    // ...
    SerialOpt serialNumber; // optional: zero or one element
};

```

按照惯例，`Opt` 后缀表示这个序列是用来建立可选值的。如果序列是空的，值显然就不在那里；如果它含有一个元素，这个元素就是那个值。这种方案明显的缺点是，有人可能会把不止一个元素放入序列。为可选值增加一个专用的 `Slice` 成分可以纠正这个问题。但可选值并非那么常用，不值得为它增加一种专门的语言特性（我们将在 4.9 节看到，你还可以用类层次来建立可选字段）。

4.7.4 词典

词典是从键类型到值类型的映射。例如：

```

struct Employee {
    long    number;
    string  firstName;
    string  lastName;
};

dictionary<long, Employee> EmployeeMap;

```

这个定义创建一种叫作 `EmployeeMap` 的词典，把雇员号映射到含有雇员详细资料的结构。你可以自行决定键类型（在这个例子中是 `long` 类型的雇员号）是否是值类型（在这个例子中是 `Employee` 结构）的一部分——就 `Slice` 而言，你无需让键成为值的一部分。

词典可用于实现稀疏数组，或是具有非整数键类型的任何用于查找的数据结构。尽管含有键—值对的结构序列可用于创建同样的事物，词典要更为适宜：

- 词典明确地表达了设计者的意图，也就是，提供从值的域（`domain`）到值的范围（`range`）的映射（含有键—值对的结构序列没有如此明确地表达同样的意图）。
- 在编程语言一级，序列被实现成向量（也可能是列表），也就是说，序列不大适用于内容稀疏的域，而且要定位具有特定值的元素，需要进行线性查找。而词典被实现成支持高效查找的数据结构（通常是哈希表或红黑树），其平均查找时间是 $O(\log n)$ ，或者更好。

词典的键类型无需为整型。例如，我们可以用下面的定义来翻译一周的每一天的名称：

```
dictionary<string, string> WeekdaysEnglishToGerman;
```

服务器实现可以用键—值对 Monday—Montag、Tuesday—Dienstag，等等，对这个映射表进行初始化。

词典的值类型可以是用户定义的任何类型。但词典的键类型只能是以下类型之一：

- 整型（byte、short、int、long、bool，以及枚举类型）
- string
- 元素类型为整型或 string 的序列
- 数据成员的类型只有整型或 string 的结构

复杂的嵌套类型，比如嵌套的结构或词典，以及浮点类型（float 和 double），不能用作键类型。之所以不允许使用复杂的嵌套类型，是因为这会使词典的语言映射复杂化；不允许使用浮点类型，是因为浮点值在跨越机器界线时，其表示会发生变化，有可能导致成问题的相等语义。

4.7.5 常量定义与直接量

Slice 允许你定义常量。常量定义的类型必须是以下类型中的一种：

- 整型（bool、byte、short、int、long，或枚举类型）
- float 或 double
- string

下面有一些例子：

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string     Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double     PI = 3.1416;
```

```
enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

直接量（literals）的语法与 C++ 和 Java 的一样（有一些小的例外）：

- 布尔常量只能用关键字 false 和 true 初始化（你不能用 0 和 1 来表示 false 和 true）。
- 和 C++ 一样，你可以用十进制、八进制，或十六进制方式来指定整数直接量。例如：

```
const byte TheAnswer = 42;
const byte TheAnswerInOctal = 052;
const byte TheAnswerInHex = 0x2A;           // or 0x2a
```

注意，如果你把 `byte` 解释成数字、而不是位模式，你在不同的语言里可能会得到不同的结果。例如，在 C++ 里，`byte` 映射到 `char`，取决于目标平台，`char` 可能是有符号的，也可能是无符号的。

注意，用于指示长常量和无符号常量的后缀（C++ 使用的 `l`、`L`、`u`、`U`）是非法的：

```
const long Wrong = 0u;           // Syntax error
const long WrongToo = 1000000L; // Syntax error
```

整数直接量的值必须落在其常量类型的范围内，如第 62 页的表 4.1 所示；否则编译器就会发出诊断消息。

- 浮点直接量使用的是 C++ 语法，除了你不能用 `l` 或 `L` 后缀来表示扩展的浮点常量；但是，`f` 和 `F` 是合法的（但会被忽略）。下面是一些例子：

```
const float P1 = -3.14f;           // Integer & fraction, with suffix
const float P2 = +3.1e-3;          // Integer, fraction, and exponent
const float P3 = .1;               // Fraction part only
const float P4 = 1.;               // Integer part only
const float P5 = .9E5;             // Fraction part and exponent
const float P6 = 5e2;              // Integer part and exponent
```

浮点直接量必须落在其常量类型（`float` 或 `double`）的范围内；否则编译器会发出诊断警告。

- 串直接量支持与 C++ 相同的转义序列。下面是一些例子：

```
const string AnOrdinaryString = "Hello World!";

const string DoubleQuote = "\"";
const string TwoSingleQuotes = "'\''";           // ' and \' are OK
const string NewLine = "\n";
const string CarriageReturn = "\r";
const string HorizontalTab = "\t";
const string VerticalTab = "\v";
const string FormFeed = "\f";
const string Alert = "\a";
const string Backspace = "\b";
const string QuestionMark = "\?";
const string Backslash = "\\";
```

```
const string OctalEscape =      "\007";    // Same as \a
const string HexEscape =      "\x07";    // Ditto

const string UniversalCharName = "\u03A9"; // Greek Omega
```

和在 C++ 里一样，相邻的串直接量会连接起来：

```
const string MSG1 = "Hello World!";
const string MSG2 = "Hello" " " "World!";          // Same message

/*
 * Escape sequences are processed before concatenation,
 * so the string below contains two characters,
 * '\xa' and 'c'.
 */
```

```
const string S = "\xa" "c";
```

注意，Slice 没有 null 串的概念：

```
const string nullString = 0;    // Illegal!
```

null 串在 Slice 里根本不存在，因此，在 Ice 平台的任何地方它都不能用作合法的串值。这一决定的原因是，null 串在许多编程语言里不存在⁵。

4.8 接口、操作，以及异常

Slice 的焦点是接口的定义，例如：

```
struct TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

5. 除了 C 和 C++，许多语言都把字节数组用作内部的串表示。在这样的语言里不存在 null 串（要进行映射也非常困难）。

这个定义定义了一个叫作 `Clock` 的接口。这个接口支持两个操作：`getTime` 和 `setTime`。要访问支持 `Clock` 接口的类，客户要祈用该对象的代理上的操作：要读取当前时间，客户祈用 `getTime` 操作；要设置当前时间，客户祈用 `setTime` 操作，把类型为 `TimeOfDay` 的参数传给它。

如果你祈用代理上的操作，就会让 `Ice run time` 发送一条消息给目标对象。目标对象可能是在另外的地址空间里，也可能是与调用者并置在一起的——目标对象的位置对于客户而言是透明的。如果目标对象是在另外的（可能是远地的）地址空间里，`Ice run time` 就会通过远地过程调用来祈用操作；如果目标与客户是并置的，`Ice run time` 就会使用普通的函数调用，以避免产生整编开销。

你可以把接口定义看作是 C++ 类定义的 `public` 部分的等价物，或是 Java 接口的等价物，并把操作定义看作是（虚）成员函数。注意，只有操作定义能出现在接口定义内部。特别地，你不能在接口内定义类型、异常，或数据成员。这并非意味着你的对象实现不能包含状态——它能包含，但这样的状态的实现方式对于客户而言是隐藏的，因此无需出现在对象的接口定义里。

一个 `Ice` 对象只有一个（派生层次最深的）`Slice` 接口类型（或类类型——参见 4.9 节）。当然，你可以创建多个类型相同的 `Ice` 对象；用 C++ 来做类比，`Slice` 接口对应的是 C++ 类定义，而 `Ice` 对象对应的是 C++ 类实例（但 `Ice` 对象可以在多个不同的地址空间中实现）。

通过一种叫作 *facets* 的特性，`Ice` 还提供了多重接口。我们将在 XREF 中详细讨论 *facets*。

`Slice` 接口定义了 `Ice` 中的最小分布粒度：每个 `Ice` 对象都有一个唯一的标识（封装在它的代理中），这个标识使这个对象与其他所有的 `Ice` 对象区别开来；要进行通信，你必须祈用对象的代理上的操作。在 `Ice` 中没有其他的可寻址实体的概念。例如，你无法实例化一个 `Slice` 结构，让客户从远地操纵这个结构。要让结构能被访问，你必须创建一个接口来让客户访问该结构。

因此，把应用划分成一些接口对总体的架构有着深远的影响。分布界线必须遵循接口（或类）的界线；你可以使接口的实现散布到多个地址空间中（你也可以在相同的地址空间中实现多个接口），但你不能在不同的地址空间中实现接口的各个部分。

4.8.1 参数与返回值

操作定义必须包含返回类型，以及零个或更多参数定义。例如，第 70 页的 `getTime` 的返回类型是 `TimeOfDay`，而 `setTime` 操作的返回类型是

`void`。你必须用 `void` 来表明某个操作不返回值——Slice 操作没有缺省的返回类型。

一个操作可以有一个或多个输入参数。例如，`setTime` 有一个叫作 `time` 的输入参数，类型是 `TimeOfDay`。当然，你可以使用多个输入参数，例如：

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    // ...
};
```

注意，参数名（和 Java 一样）是必需的。你不能省略参数名，所以下面的定义是错误的：

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay, TimeOfDay); // Error!
    // ...
};
```

在缺省情况下，参数会从客户发往服务器，也就是说，它们是输入参数。要把值从服务器传到客户，你可以使用输出参数，这种参数用 `out` 关键字指示。例如，你可以用另外一种方式定义第 70 页上的 `getTime` 操作：

```
void getTime(out TimeOfDay time);
```

这能够达到同样的目的，但使用的是输出参数，而不适返回值。和输入参数一样，你可以使用多个输出参数：

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    void getSleepPeriod(out TimeOfDay startTime,
                        out TimeOfDay stopTime);
    // ...
};
```

如果你的操作既有输入参数，又有输出参数，输出参数必须放在输入参数的后面：

```
void changeSleepPeriod(    TimeOfDay startTime,        // OK
                          TimeOfDay stopTime,
                          out TimeOfDay prevStartTime,
                          out TimeOfDay prevStopTime);
void changeSleepPeriod(out TimeOfDay prevStartTime,
                      out TimeOfDay prevStopTime,
                      TimeOfDay startTime,        // Error
                      TimeOfDay stopTime);
```


`Slice` 不支持既是输入、又是输出参数的参数（传引用调用）。其原因是，对于远地调用，使用引用参数并不能带来“使用编程语言中的传引用调用”所能带来的“节余”（在两个方向上数据都仍然需要复制，而在整编效率上的一点提高可以忽略不计）。而且，引用（输入—输出）参数会造成语言映射变得更复杂，同时还会带来代码尺寸的增大。

操作定义的风格

正如你可能会期望的，语言映射会遵循你在 `Slice` 中使用的操作定义的风格：`Slice` 返回类型映射到编程语言的返回类型，而 `Slice` 参数映射到编程语言的参数。

对于只返回一个值的操作，常见的做法是从操作返回这个值、而不是使用 `out` 参数。这种风格能够自然地映射到所有的编程语言。注意，如果你使用的是 `out` 参数，你就是在把一种不同的风格强加给客户：大多数编程语言都允许你忽略函数的返回值，但你通常不能忽略输出参数。

对于返回多个值的操作，常见的做法是把所有的值作为 `out` 参数返回，并用 `void` 来做返回类型。但规则并非那么确定无疑，因为对于有些具有多个输出值的操作，其中的某个特定的值可能会被认为比其他值更“重要”。一个常见的例子是下面这样的迭代器例子，它一个接一个地从 `collection` 中返回数据项：

```
bool next(out RecordType r);
```

`next` 操作返回两个值：它所取回的记录，以及一个指示是否已到 `collection` 的末尾的布尔值（如果返回值是 `false`，就已经到达了 `collection` 的末尾，参数 `r` 的值就是不确定的）。这种风格的定义也可能会有用，因为它能够自然地与程序员编写控制结构的方式相对应。例如：

```
while (next(record))
    // Process record...

if (next(record))
    // Got a valid record...
```

重载

`Slice` 不支持任何形式的操作重载。例如：

```
interface CircadianRhythm {
    void modify(TimeOfDay startTime,
                TimeOfDay endTime);
    void modify(    TimeOfDay startTime,           // Error
```

```

        Ti meOfDay endTi me,
        out ti meOfDay prevStartTi me,
        out Ti meOfDay prevEndTi me);
};

```

同一接口中的各个操作必须具有不同的名称，不管它们的参数的类型是什么，数目是多少。之所以存在这个限制，是因为重载函数无法有效地映射到没有内建的重载支持的语言⁶。

Nonmutating 操作

有些操作，比如第 70 页上的 `getTi me`，不会修改它们所操作的对象的状态。它们在概念上与 C++ `const` 成员函数是等价的。你可以这样来指示这一点：

```

interface Clock {
    nonmutating Ti meOfDay getTi me();
    void setTi me(Ti meOfDay ti me);
};

```

`nonmutating` 关键字说明，`getTi me` 操作不会改变它的对象的状态。这是有用的，原因有两个：

- 语言映射可以利用关于操作行为的这项附加知识。例如，在使用 C++ 时，`nonmutating` 操作会映射到骨架类上的 C++ `const` 成员函数。
- 知道了某个操作不会修改它的对象的状态，Ice run time 就可以尝试进行更积极的错误恢复。特别地，Ice 会保证操作祈用的“最多一次”语义。

对于普通的操作，Ice run time 在处理错误时必须保守。例如，如果客户发送一个操作祈用给服务器，然后连接断掉了，在这种情况下，客户端 run time 无法知道它所发送的请求是否已实际到达服务器。这意味着，客户端 run time 不能通过重新建立连接、并再次发送请求来进行错误恢复，因为这可能会造成操作两次祈用，从而违反“最多一次”语义；客户端 run time 别无选择，只能把错误报告给应用。

而另一方面，对于 `nonmutating` 操作，客户端 run time 可以尝试重新建立与服务器的连接，并安全地再次发送先前失败的请求。如果第二次尝试能够联系上服务器，那么万事大吉，应用不会注意到发生过（暂时的）失败。只有在第二次尝试也失败的情况下，客户端 run time

6. 在这种情况下，名称搅乱（name mangling）并非是一种可行的办法：对于编译器而言它能够很好地工作，但对于人而言却不可接受。

才需要把错误报告给应用（可以通过 Ice 的一个配置参数来增加重试次数）。

Idempotent 操作

我们可以进一步修改第 74 页上的 Clock 接口的定义，指明 setTime 操作是 *idempotent* 操作：

```
interface Clock {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};
```

如果对某个操作进行两次连续的祈用，其效果与一次祈用是一样的，这个操作就是 *idempotent* 操作。例如，`x = 1;` 是一个 *idempotent* 操作，因为它执行一次还是两次没有关系——不管怎样，`x` 的值最后都是 1。另一方面，`x += 1;` 不是一个 *idempotent* 操作，因为它执行两次和执行一次，`x` 的值不一样。

idempotent 关键字表明某个操作可以安全地多次执行。和 *nonmutating* 操作的情况一样，Ice run time 利用这一知识来更积极地进行错误恢复。

一个操作可以有 *nonmutating* 修饰符，也可以有 *idempotent* 修饰符，但不能同时有这两个修饰符（*nonmutating* 隐含了 *idempotent*）。

4.8.2 用户异常

我们查看第 70 页上的 setTime 操作，发现了一个潜在的问题：

TimeOfDay 结构使用 short 作为每个字段的类型，如果客户祈用 setTime 操作，传入的 TimeOfDay 值的字段值无意义（比如分钟字段的值是 -199，小时字段的值是 42），会发生什么事情呢？显然，向调用者指出这是没有意义的，是一种很好的做法。Slice 允许你定义用户异常，用以向客户指示错误情况。例如：

```
exception Error {}; // Empty exceptions are legal

exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

用户异常很像是含有一些数据成员的结构。但是，与结构不同，异常可以有零个数据成员，也就是说，是空的。当操作的实现出错时，异常允许

你向客户返回任意数量的出错信息。操作可以使用异常规范来说明可能会有异常返回给客户：

```
interface Clock {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
        throws RangeError, Error;
};
```

这个定义表明，`setTime` 操作可能会抛出 `RangeError` 或 `Error` 用户异常（不会抛出其他类型的异常）。如果客户收到 `RangeError` 异常，在该异常中会包含传给 `setTime`、并导致出错的 `TimeOfDay` 值（在 `errorTime` 成员中），以及允许使用的最小和最大时间值（在 `minTime` 和 `maxTime` 成员中）。如果 `setTime` 的失败不是非法的参数值造成的，它就会抛出 `Error`。显然，因为 `Error` 没有数据成员，客户不会知道到底出了什么问题——它只知道操作失败了。

操作只能抛出在它的异常规范中列出的那些用户异常。如果某个操作的实现在运行时抛出的异常没有在它的异常规范中列出，客户就会收到一个运行时异常（参见 4.8.4 节），说明这个操作做了非法的事情。要说明某个操作不会抛出任何用户异常，只需忽略异常规范就可以了（在 `Slice` 中没有空异常规范）。

异常不是第一类数据类型，各种第一类数据类型也不是异常：

- 你不能把异常当作参数值传递。
- 你不能把异常用作数据成员的类型。
- 你不能把异常用作序列的元素类型。
- 你不能把异常用作词典的键或值类型。
- 你不能抛出非异常类型的值（比如 `int` 或 `string` 类型的值）。

之所以作出这些限制，是因为有些实现语言单独为异常使用了专门的类型（就像 `Slice` 这样）。对于这样的语言，如果异常能被用作普通的数据数据类型，异常的映射将会很困难（在各种编程语言中，C++ 有点不同寻常，它允许程序员把任意的类型用作异常）。

4.8.3 异常继承

异常支持继承。例如：

```
exception ErrorBase {
    string reason;
};
```

```

enum RError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RError err;
};

enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ };

exception LogicError extends ErrorBase {
    LError err;
};

exception RangeError extends LogicError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};

```

这些定义设立了一个简单的异常层次：

- `ErrorBase` 位于树的根部，含有一个用于解释出错原因的串。
- 从 `ErrorBase` 派生的是 `RuntimeError` 和 `LogicError`。它们各自含有一个枚举值，用于进一步划分错误的范畴。
- 最后，`RangeError` 是从 `LogicError` 派生的，用于报告特定错误的详情。

设立这样的异常层次不仅有助于创建可读性更好的规范（因为错误得到了分类），而且能够在语言层面上带来好处。例如，`Slice C++` 映射会保持这样的异常层次，所以你可以用基异常一般化地捕捉异常，也可以设置异常处理器，处理特定的异常。

查看一下第 76 页上的异常层次，我们不清楚应用在运行时是会只抛出派生层次最深的异常（比如 `RangeError`），还是也会抛出基异常（`LogicError`、`RuntimeError`，以及 `ErrorBase`）。如果你想要指明某个基异常、接口，或类是抽象的（不能实例化），你可以用注释加以说明。

注意，如果某个操作的异常规范指明了具体的异常类型，该操作的实现在运行时也可能会抛出派生层次最深的异常。例如：

```

exception Base {
    // ...
};

exception Derived extends Base {
    // ...
};

```

```
};

interface Example {
    void op() throws Base;           // May throw Base or Derived
};
```

在这里例子里，`op` 可能会抛出 `Base` 或 `Derived` 异常，也就是说，在运行时，可以抛出任何与异常规范中列出的异常类型兼容的异常。

随着系统的演化，你常常要在已有的层次中加入新的、派生的异常。假定我们一开始用下面的定义构造客户和服务端：

```
exception Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

再假定在现场部署了大量客户，也就是说，你无法轻易升级所有客户。随着应用的演化，一个新的异常增加到系统中，服务端用新的定义重新进行了部署：

```
exception Error {
    // ...
};

exception FatalApplicationError extends Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

这带来了一个问题：如果服务端从 `doSomething` 中抛出一个 `FatalApplicationError`，会发生什么事情？答案取决于客户是用老的、还是更新过的定义构建的：

- 如果客户是用与服务端相同的定义构建的，它就会接收到一个 `FatalApplicationError`。
- 如果客户是用原来的定义构建的，客户就不知道 `FatalApplicationError` 的存在。在这种情况下，`Ice run time` 会自动把这个异常切成接收者能够理解的派生层次最深的类型（在这种情况下是 `Error`），并丢弃

与异常的派生部分相对应的信息（这与按值来捕捉 C++ 异常完全类似——异常被切成 `catch` 子句中所用的类型）。

异常只支持单继承（多继承难以映射到许多编程语言）。

4.8.4 Ice 运行时异常

在 2.2.2 节提到过，除了在操作的异常规范中列出的用户异常，操作还可能会抛出 *Ice 运行时异常*。运行时异常是预定义的异常，用于指示与平台有关的运行时错误。例如，如果网络错误造成了客户和服务器之间通信的中断，客户就会通过运行时异常收到通知，比如 `ConnectTimeoutException` 或 `SocketException`。

操作的异常规范不能列出任何运行时异常（所有操作都能抛出运行时异常，你不能重申这一点）。

异常的继承层次

如图 4.3 所示，所有的 Ice 运行时异常和用户异常都处在一个继承层次中。

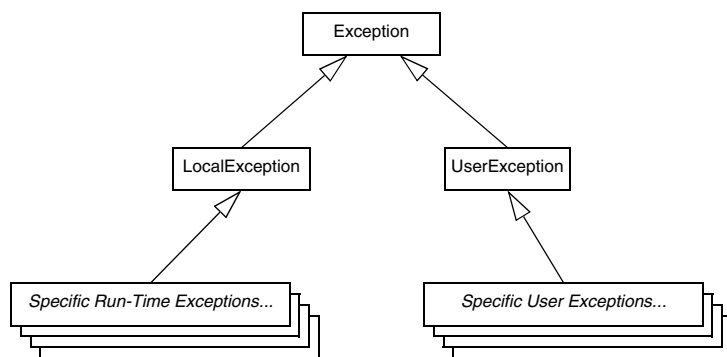


图 4.3. Ice 运行时异常的继承结构

`Ice::Exception` 位于这个继承层次的根部。从根部派生的是（抽象的）`Ice::LocalException` 和 `Ice::UserException` 类型。所有的运行时异常又派生自 `Ice::LocalException`，所有的用户异常则派生自 `Ice::UserException`。

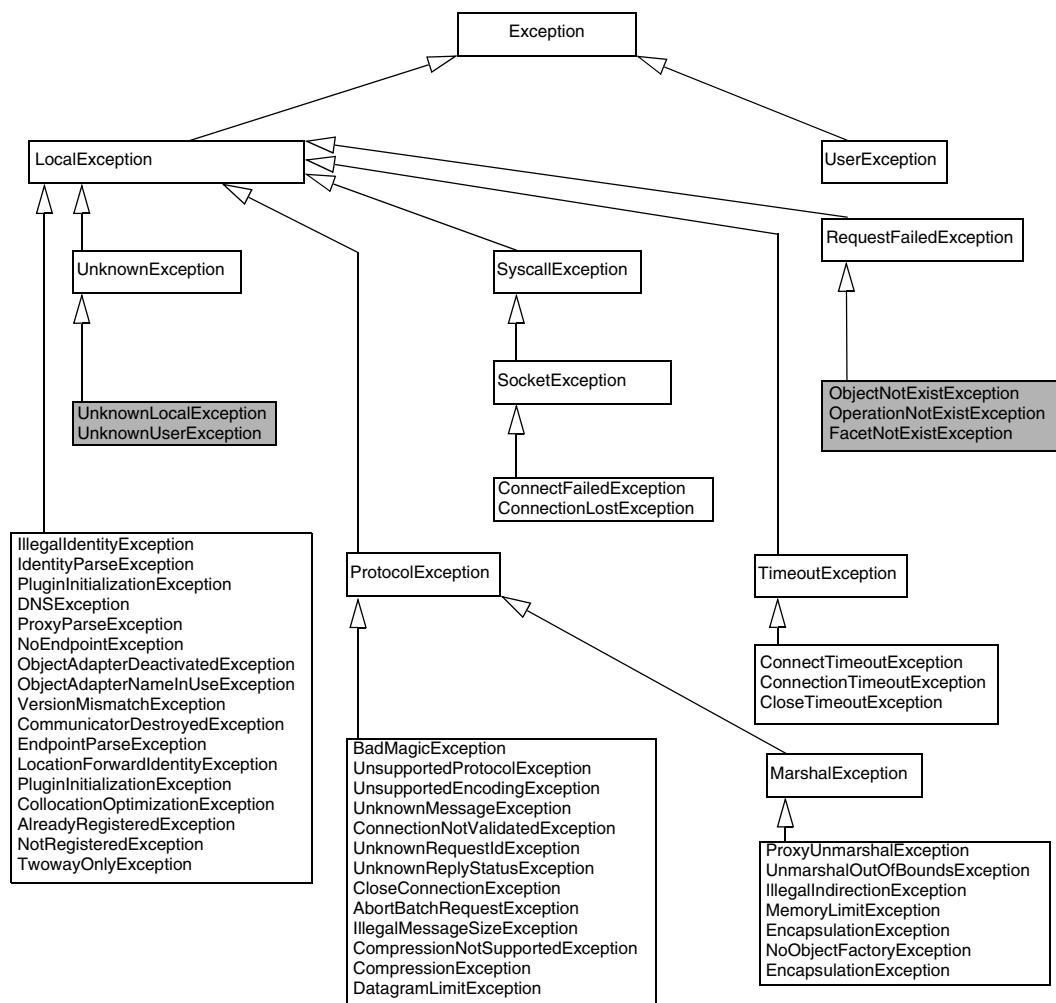
图 4.4 给出了 Ice 运行时异常的完整层次⁷。

图 4.4. Ice 运行时异常层次（服务器可能会发送有阴影的异常）

7. 在本书中，我们使用了 Unified Modeling Language（UML）来绘制对象模型图（详情参见 [1] 和 [13]）。

注意，为了节省空间，图 4.4 把若干相关的异常放进了一个方框中（严格地说，这不是正确的 UML 语法）。还要注意，有些运行时异常有数据成员，为简洁起见，我们在图 4.4 中忽略了它们。这些数据成员提供了关于出错的确切原因的额外信息。

许多运行时异常都有着自明的名称，比如 `MemoryLimitException`。另外一些则指示 Ice run time 中的问题，比如 `EncapsulationException`。还有一些只有在出现应用编程错误时才会发生，比如 `NotRegisteredException`。在实践中，你可能不会看到这里的大多数异常。但有一些异常是你会遇到的，你应该了解它们的含义。

本地异常 vs. 远地异常

大多数出错情况都会在客户端检测到。例如，如果联系服务器的尝试失败，客户端 run time 就会引发 `ConnectTimeoutException`。但有三种特定的异常情况会由服务器检测到（在图 4.4 中加了阴影），这些异常会通过 Ice 协议显式地告诉客户端 run time：

- `ObjectNotExistException`

这个异常表明，某个请求已经递送到了服务器，但服务器无法找到一个 `servant`，具有与代理中嵌入的标识相同的标识。换句话说，服务器无法找到一个对象来把请求分派给它。

`ObjectNotExistException` 是一份死亡证明书：它表明目标对象在服务器中不存在，而且，也不会存在。如果你收到这个异常，你应该清理你分配过的、与这个对象有关的所有资源。

- `FacetNotExistException`

客户试图联系某个对象的一个 `facet`，而这个 `facet` 不存在（关于 `facets` 的讨论，参见 XREF）。

- `OperationNotExistException`

如果服务器能够定位到一个具有正确标识的对象，但在尝试分派客户的操作祈用时，它发现目标对象没有这样一个操作，就会引发这个异常。只在以下两种情况下，你才会看到这个异常：

- 你对类型不正确的代理做了不进行检查的向下转换（关于不进行检查的向下转换，参见第 159 页和第 211 页）。
- 在构建客户和服务器时，使用了不一致的 `Slice` 接口定义，也就是说，客户构建时使用的对象接口定义表明某个操作存在，而服务器构建时使用了另外一种版本的接口定义，其中没有这个操作。

在服务器端出现的错误情况，如果不能用上述三种异常来描述，就会作为两种一般异常（在图 4.4 中加了阴影）之一告诉客户：

- `UnknownUserException`

这个异常表明，某个操作实现抛出的异常没有在操作的异常规范中声明。例如，如果服务器中的操作抛出了一个 C++ 异常（比如一个 `char *`），或是一个 Java 异常（比如一个 `ClassCastException`），客户就会收到 `UnknownUserException`。

- `UnknownLocalException`

如果某个操作实现引发了除 `ObjectNotExistException`、`FacetNotExistException`，以及 `OperationNotExistException` 之外的运行时异常（比如 `NotRegisteredException`），客户就会收到 `UnknownLocalException`。换句话说，Ice 协议不会传送在服务器中遇到的确切异常，而是会简单地在回复中向客户返回一个位，表明服务器遇到了运行时异常。

客户收到 `UnknownLocalException` 的常见原因是，服务器没有捕捉并处理所有异常。例如，如果某个操作的实现没有处理它遇到的某个异常，这个异常就会沿着调用栈一路传播，直到栈回绕到 Ice run time 祈用这个操作的地方。Ice run time 会捕捉从某个操作中“逃脱”的所有异常，并把它们作为 `UnknownLocalException` 返回给客户。

所有其他的运行时异常（在图 4.4 中没有加阴影）都由客户端 run time 负责检测，并且在本地引发。

操作的实现也有可能抛出 Ice 运行时异常（以及用户异常）。例如，如果客户持有某个在服务器中已经不存在的对象的代理，你的服务器应用代码就应该抛出 `ObjectNotExistException`。如果你确实要从应用代码中抛出运行时异常，你应该只在适当的情况下这么做，也就是说，不要用运行时异常来指示本该是用户异常的情况。如果你这样做了，客户可能会非常困惑：如果应用“劫持”了某些运行时异常，用于自己的目的，客户就不再能确定异常是 Ice run time 抛出的，还是服务器应用代码抛出的。这可能会使调试变得非常困难。

4.8.5 接口语义与代理

在 Clock 例子的基础上，我们可以创建一个世界时间服务器的定义：

```
exception GenericError {
    string reason;
};

struct TimeOfDay {
    short hour;           // 0 - 23
    short minute;        // 0 - 59
```

```

        short second;           // 0 - 59
    };

    exception BadTimeVal extends GenericError {};

    interface Clock {
        nonmutating TimeOfDay getTime();
        idempotent void setTime(TimeOfDay time) throws BadTimeVal;
    };

    dictionary<string, Clock*> TimeMap; // Time zone name to clock map

    exception BadZoneName extends GenericError {};

    interface WorldTime {
        idempotent void addZone(string zoneName, Clock* zoneClock);
        void removeZone(string zoneName) throws BadZoneName;
        nonmutating Clock* findZone(string zoneName)
                                throws BadZoneName;
        nonmutating TimeMap listZones();
        idempotent void setZones(TimeMap zones);
    };

```

WorldTime 接口充当的是时钟 collection 的管理器，每个时区一个时钟。换句话说，WorldTime 接口负责管理时区—时钟对的 collection。每一对时区—时钟的第一个成员是时区名称；第二个成员是负责提供该时区的时间的时钟。这个接口含有一些操作，允许你在映射表中增加或移除时钟（addZone 和 removeZone）、按照名称查找特定的时区（findZone），以及读写整个映射表（listZones 和 setZones）。

WorldTime 例子说明了 Slice 的一个重要概念：addZone 接受的参数的类型是 Clock*，而 findZone 返回的参数的类型是 Clock*。换句话说，接口是有资格的类型，可以作为参数传递。* 操作符叫作 *代理操作符*。其左手的参数必须是接口（或类，参见 4.9 节），返回类型则是代理。代理就像是能代表对象的指针。代理的语义与 C++ 类实例指针的语义非常像：

- 代理可以为 null（参见第 88 页）。
- 代理可以悬空（dangle）（指向的对象已经不存在）
- 通过代理分派的操作使用的是迟后绑定：如果与代理的类型相比，代理所代表的对象的实际运行时类型派生层次更深，祈用的就将是派生层次最深的接口的实现。

当客户把一个 Clock 代理传给 addZone 操作时，代理代表的是服务器中的一个实际的 Clock 对象。这个代理代表的 Clock Ice 对象可以在与 World-

Time 接口所在的服务器进程中实现，也可以在另外的服务器进程中实现。无论是对客户而言，还是对实现 WorldTime 接口的服务器而言，Clock 对象在物理上是在哪里实现的都无关紧要；如果它们祈用了特定时钟上的某个操作，比如 getTime，就会有一个 RPC 调用发往实现这个时钟的任何一个服务器。换句话说，代理充当的是远地对象的本地“大使”；如果你祈用代理上的某个操作，你的祈用会转发给实际的对象实现。如果对象实现是在另外的地址空间中，就会产生一个远地过程调用；如果对象实现是并置在相同的地址空间中的，Ice 就会使用普通的本地函数调用，从代理那里调用对象实现。

注意，在共享语义方面，代理的行为也和指针非常像：如果两个客户都有一个代理，指向相同的对象，一个客户造成的状态变化（比如设置时间）就会被另一个客户看到。

代理是强类型的（至少对于像 C++ 和 Java 这样的静态类型语言来说是这样）。这意味着，你不能把 Clock 代理以外的东西传给 addZone 操作；编译器会拒绝这样的企图。

4.8.6 接口继承

接口支持继承。例如，我们可以扩展我们的世界时间服务器，支持闹钟的概念：

```
interface AlarmClock extends Clock {
    nonmutating TimeOfDay getAlarmTime();
    idempotent void          setAlarmTime(TimeOfDay alarmTime)
                               throws BadTimeVal;
};
```

这个接口的语义与 C++ 或 Java 中的情况是一样的：AlarmClock 是 Clock 的子类型，只要能使用 Clock 代理的地方，都能够使用 AlarmClock 代理。显然，与 Clock 一样，AlarmClock 也支持 getTime 和 setTime 操作，但它还支持 getAlarmTime 和 setAlarmTime 操作。

你也可以进行多重接口继承。例如，我们可以这样构造一个无线电闹钟：

```
interface Radio {
    void setFrequency(long hertz) throws GenericError;
    void setVolume(long dB) throws GenericError;
};

enum AlarmMode { RadioAlarm, BeepAlarm };
```

```
interface RadioClock extends Radio, AlarmClock {
    void      setMode(AlarmMode mode);
    AlarmMode getMode();
};
```

RadioClock 既扩展了 Radio，又扩展 AlarmClock，因此，只要是需要 Radio、AlarmClock，或 Clock 的地方，都可以使用 RadioClock。这个定义的继承图看起来像这样：

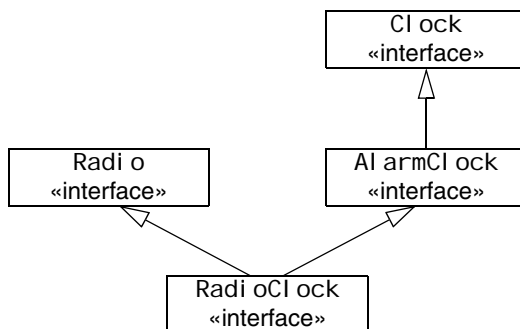


图 4.5. RadioClock 的继承图

从不止一个基接口继承的基类可能会共有相同的基类。例如，下面的定义是合法的：

```
interface B { /* ... */ };
interface I1 extends B { /* ... */ };
interface I2 extends B { /* ... */ };
interface D extends I1, I2 { /* ... */ };
```

这个定义会产生为人所熟知的菱形图案：

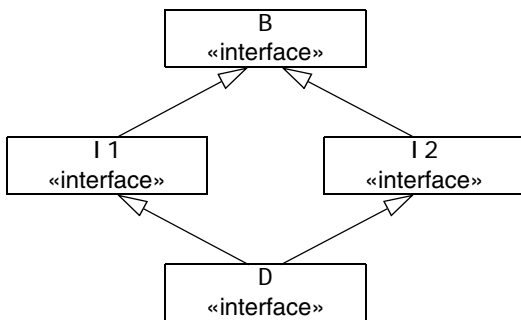


图 4.6. 菱形的继承图

接口继承的局限

如果一个接口使用多重继承，它不能从不止一个基接口那里继承相同的操作名称。例如，下面的定义是非法的：

```

interface Clock {
    void set(TimeOfDay time);           // set time
};

interface Radio {
    void set(long hertz);               // set frequency
};

interface RadioClock extends Radio, Clock { // Illegal!
    // ...
};
  
```

这个定义之所以是非法的，是因为 RadioClock 继承了两个 set 操作：Radio::set 和 Clock::set。Slice 编译器之所以认为这是非法的，是因为（与 C++ 不同）许多编程语言都没有内建的设施，可用于消除这样的操作的歧义。Slice 的简单规则就是，所有继承来的操作都必须拥有唯一的名称（在实践中，这很少会成问题，因为继承很少会在“事后”才增加到接口层次中。为了避免偶然发生冲突，我们建议你使用描述性的操作名称，比如 setTime 和 setFrequency。这能降低偶然发生名字冲突的可能性）。

隐含地继承 Object

所有 Slice 接口最终都派生自 Object. 例如，把图 4.5 中的继承层次改成图 4.7 中的继承层次，会更准确一点。

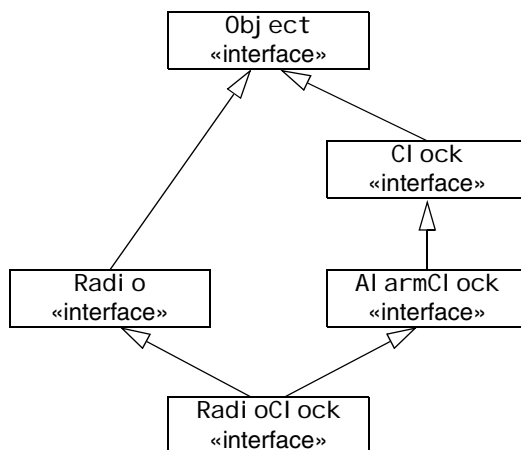


图 4.7. 隐含地继承 Object

因为所有接口都有一个相同的基接口，我们可以把任何类型的接口当成这种接口传递。例如：

```
interface ProxyStore {
    idempotent void putProxy(string name, Object* o);
    nonmutating Object* getProxy(string name);
};
```

Object 是一个 Slice 关键字（注意大小写），表示的是继承层次的根类型。ProxyStore 接口是一种通用的代理存储设施：客户可以调用 putProxy，在指定的名下增加任意类型的代理，然后在需要时调用 getProxy，给出前面使用的名字，取回该代理。有了以这种方式一般化地存储代理的能力，我们可以构建各种通用设施，比如能存储代理、并把代理递送给客户的命名服务。有了这样的服务，我们还能够避免在客户和服务器中硬性编写代理的细节（参见第 20 章）。

对 Object 类型的继承总是隐含的。例如，下面的 Slice 定义是非法的：

```
interface MyInterface extends Object { /* ... */ }; // Error!
```

所有接口都隐含地继承自 Object 类型；你不能再重申这一点。

Null 代理

我们再一次查看 `ProxyStore` 接口，注意到 `getProxy` 没有异常规范。于是就有了这样一个问题：如果客户调用 `getProxy` 时所用的名字没有对应的代理，会发生什么事情？显然，我们可以给 `getProxy` 增加一个异常，用以指示这种情况。但另外一种做法是返回 *null 代理*。Ice 有内建的 *null 代理* 概念：*null 代理* 就是“哪儿也不指向”的代理。当客户收到这样的代理时，它可以测试所返回的代理的值，检查它到底是 *null*，还是代表有效的对象。

一个更有意思的问题：哪一种做法更恰当？抛出异常，还是返回 *null 代理*？答案取决于接口可能会以什么样的方式使用。例如，如果在正常的操作中，你不希望客户用不存在的名称调用 `getProxy`，那就最好抛出异常（这很可能是我们的 `ProxyStore` 接口的情况：不存在 `list` 操作这一事实表明，客户应该知道在使用中的名称有哪些）。

另一方面，如果你预计客户偶尔会试图查找不存在的的东西，那最好是返回 *null 代理*。其原因是，抛出异常会打断客户中的正常的控制流，并且要求使用特殊的处理代码。这意味着，你只应在异常的情形下抛出异常。例如，如果数据库查找返回空结果集、就抛出异常，这是一种错误的做法；结果集偶尔会是空的，而且这也是正常的。

你应该注意这样的设计问题——能够正确处理这些细节、设计良好的接口会更容易使用和理解。这样的接口不仅能使客户开发者的生活更轻松，同时也能降低潜伏的 *bug* 在以后带来问题的可能性。

自引用的接口

代理具有指针语义，所以我们可以定义自引用的接口。例如：

```
interface Link {
    nonmutating SomeType getValue();
    nonmutating Link* next();
};
```

`Link` 接口含有一个 `next` 操作，这个操作返回的是一个指向 `Link` 接口的代理。显然，这可以用来创建接口链；接口链中最后的链接的 `next` 操作返回 *null 代理*。

空接口

下面的 `Slice` 定义是合法的：

```
interface Empty {};
```

`Slice` 编译器在编译这个定义时不会发出抱怨。一个有意思的问题是：“我为何需要使用空接口？”在大多数情况下，空接口都说明在设计上存在错误。这里有一个例子：


```
interface ThingBase {};  
  
interface Thing1 extends ThingBase {  
    // Operations here...  
};  
  
interface Thing2 extends ThingBase {  
    // Operations here...  
};
```

考察这个定义，我们可以观察到两个事实：

- Thing1 和 Thing2 有共同的基类，因此是相关的。
- 不管 Thing1 和 Thing2 有什么共同之处，都可以在 ThingBase 接口中找到。

当然，只要看一看 ThingBase，我们就会发现 Thing1 和 Thing2 根本没有共享任何操作，因为 ThingBase 是空的。假如我们是在使用面向对象范型，这种做法就显然很奇怪：在面向对象模型中，与某个对象通信的唯一途径是向它发送消息。但要发送消息，我们需要有操作。假如 ThingBase 没有操作，我们就无法向它发送消息，而 Thing1 和 Thing2 也就是不相关的，因为它们没有共同的操作。但看到 Thing1 和 Thing2 有共同的基类，我们就会得出这样的结论：它们是相关的，否则共同的基类就根本不会存在。到了这里，大多数程序员都会开始挠头，想知道到底在发生什么事情。

使用这样的设计的一个常见理由是，要多态地处理 Thing1 和 Thing2。例如，我们可以继续先前的定义：

```
interface ThingUser {  
    void putThing(ThingBase* thing);  
};
```

现在使用共同的基类的目的就清楚了：我们既想要把 Thing1 代理、也想要把 Thing2 代理传给 putThing。这能否证明使用空的基接口是正当的？要回答这个问题，我们需要思考一下在 putThing 的实现中发生的事情。显然，putThing 不可能祈用 ThingBase 上的操作，因为在那里没有操作。这意味，putThing 必须要能做以下两件事情之一：

1. putThing 能够记住事物的值。
 2. putThing 能够试着向下转换到 Thing1 或 Thing2，然后祈用操作。
- putThing 的伪码实现看起来可能像是这样：

```
void putThing(ThingBase thing)  
{  
    if (is_a(Thing1, thing)) {  
        // Do something with Thing1...    }}
```

```

    } else if (is_a(Thing2, thing)) {
        // Do something with Thing2...
    } else {
        // Might be a ThingBase?
        // ...
    }
}

```

这个实现试着依次把它的参数向下转换成每种可能的值，直到它找到参数实际的运行时类型。当然，任何一本像样的面向对象课本都会告诉你，这是在滥用继承，并且会带来维护问题。

如果你发现自己在编写像 `putThing` 这样的操作，依赖于人为的基接口，问问你自己，你是否真的需要采用这种做法。例如，这样的设计可能更加适宜：

```

interface Thing1 {
    // Operations here...
};

interface Thing2 {
    // Operations here...
};

interface ThingUser {
    void putThing1(Thing1* thing);
    void putThing2(Thing2* thing);
};

```

在这种设计中，`Thing1` 和 `Thing2` 是不相关的，而 `ThingUser` 为每种类型的代理提供了单独的操作。这些操作的实现不需要使用任何向下转换，而且在我们的面向对象世界里，一切都安然无恙。

下面是空的基接口的另一种常见用法：

```

interface PersistentObject {};

interface Thing1 extends PersistentObject {
    // Operations here...
};

interface Thing2 extends PersistentObject {
    // Operations here...
};

```

显然，这种设计把持久功能放在 `PersistentObject` 基接口中，并且要求想要拥有持久状态的对象继承 `PersistentObject`。表面上，这是合理

的：毕竟，这样使用继承是一种沿用已久的设计模式，那么，它可能有什么问题？我们发现，这种设计有这样一些问题：

- 上面的继承层次用来给 `Thing1` 和 `Thing2` 增加行为。但在严格的 OO 模型中，行为只能通过发送消息来祈用。

这引发了这样一个问题：`PersistentObject` 实际上该怎样着手完成它的工作；推测起来，它对 `Thing1` 和 `Thing2` 的实现（也就是，内部状态）有所了解，所以它可以把该状态写入数据库。但如果是这样，`PersistentObject`、`Thing1`，以及 `Thing2` 就不能再在不同的地址空间中实现了，因为如果是那样，`PersistentObject` 就不再能知道 `Thing1` 和 `Thing2` 的状态。

换一种做法，`Thing1` 和 `Thing2` 可以使用 `PersistentObject` 提供的某种功能，使它们的内部状态持久。但 `PersistentObject` 没有任何操作，那么 `Thing1` 和 `Thing2` 实际上又该怎样去完成这件事情呢？再一次，唯一可行的做法是，在同一个地址空间中实现 `PersistentObject`、`Thing1`，以及 `Thing2`，并让它们在幕后共享实现状态，也就是说，它们不能在不同的地址空间中实现。

- 上面的继承层次把世界分成两半，一个含有持久对象，另一个含有非持久对象。这种做法有着深远的影响：
- 假定你有一个应用，它已经实现了一些非持久对象。随着时间推移，需求发生变化，你发现现在你想让部分对象持久。采用上面的设计，你无法做到这一点，除非你改变你的对象的类型，因为它们现在必须继承 `PersistentObject`。这当然是一个极其糟糕的消息：你不仅要改变你的服务器中的对象的实现，还要找到并更新所有正在使用你的对象的客户，因为它们突然有了一种全新的类型。更糟糕的是，你无法让它们向后保持兼容：或者让所有客户随着服务器发生改变，或者一个客户都不改变。要想让某些客户“不升级”，这是不可能的。
- 这种设计不能扩展到支持多种特性。设想一下，我们有另外一些行为，对象可以继承它们，比如序列化、容错、持久，以及用搜索引擎进行搜索的能力。我们很快就会陷入多重继承的泥淖。更糟糕的是，每种可能的特性组合都会创建一种完全独立的类型层次。这意味着，你不再能编写出一些操作，一般化地对一些对象类型进行操作。例如，你不能把持久对象传到需要非持久对象的地方，*即使对象的接收者并不在乎对象的持久方面*。这很快会造成碎片化的、难以维护的类型系统。不久，你会发现，你不是在重写应用，就是获得了某种难以使用也难以维护的东西。

但愿前面的讨论成为一个警告：`Slice` 是一种接口定义语言，与实现没有任何关系（但空接口几乎总是表明，你的应用通过所定义的接口之外的

机制共享了实现状态)。如果你发现自己在编写空的接口定义,你至少应该后退一步,思考一下手上的问题;其他设计可能会更加适宜,更能清晰地表达你的意图。如果无论如何你都要使用空接口,那么要注意,你几乎肯定会失去这样的能力:改变对象模型在物理的服务器进程上的分布方式,因为你无法把共享了隐藏状态的接口分置在不同的地址空间中。

接口继承 vs. 实现继承

要记住, Slice 接口继承只适用于接口。特别地,如果两个接口处在继承关系中,这并不意味着这些接口的实现也要发生继承关系。你可以在实现接口时选择使用实现继承,但你也可以使这些实现相互独立(对于 C++ 程序员而言,这可能会让人惊奇,因为 C++ 缺省地使用实现继承,而要实现接口继承,需要做额外的事情)。

总而言之, Slice 继承只是在建立类型兼容性。它并没有说出任何与接口的实现方式有关的事情,因此,你可以针对你的应用的实际情况来选择实现方式。

4.9 类

除了接口, Slice 还允许你定义类。类像是接口:它们都能有操作;类也像是结构:它们都能有数据成员。这就产生了一种混合的对象,你可以把它们当作接口,通过引用进行传递;也可以把它们当作值,通过值进行传递。类提供了很大的架构灵活性。例如,类允许你在客户端实现行为,而接口只允许你在服务器端实现行为。

类支持继承,因此是多态的:在运行时,只要实际的类类型是从操作的型构的形参类型派生的,你就可以把一个类实例传给一个操作。这也使得类能够用作类型安全的联合,就像 Pascal 的 discriminated variant record。

4.9.1 简单类

Slice 的类定义与结构定义类似,但所用关键字是 `class`。例如:

```
class TimeOfDay {  
    short hour;           // 0 - 23  
    short minute;         // 0 - 59  
    short second;         // 0 - 59  
};
```

除了关键字 `class`, 这个定义与我们在第 64 页上看到的结构定义是相同的。能使用 Slice 结构的地方,你都能使用 Slice 类(但我们很快就会看

到，出于性能上的考虑，如果结构已经够用，你就不应该使用类）。与结构不同，类可以是空的：

```
class EmptyClass {};    // OK
struct EmptyStruct {};  // Error
```

在使用空接口时要考虑的大多数设计问题（参见第 88 页）也适用于空类：在决定采用空类之前，你至少应该停一停，重新思考一下你的做法。

4.9.2 类继承

与结构不同，类支持继承。例如：

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;             // 1 - 31
    short month;           // 1 - 12
    short year;            // 1753 onwards
};
```

这个例子说明了我们为什么要使用类的一个主要原因：类可以通过继承扩展，而结构不能扩展。前面的例子定义的 `DateTime` 用日期扩展了 `TimeOfDay` 类⁸。

类只支持单继承。下面的定义是非法的：

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

class Date {
    short day;
    short month;
    short year;
```

8. 如果注释中的 1753 年让你感到困惑，你可以用“1752 date change”搜索一下 Web。许多国家在这一年以前使用的错综复杂的历法会让你几个月脱不了身……

```
};

class DateTime extends TimeOfDay, Date {    // Error!
    // ...
};
```

派生类也不能重新定义其基类的数据成员：

```
class Base {
    int integer;
};

class Derived extends Base {
    int integer;                // Error, integer redefined
};
```

4.9.3 类的继承语义

类使用的传值语义和结构一样。如果你把一个类实例传给一个操作，这个类和它的所有成员都会被传递。通常的类型兼容规则在此是适用的：你可以把派生实例传给期望基实例的地方。如果关于派生实例的实际运行时类型、接收者拥有静态的类型知识，那么它接收到的就是派生实例；而如果接收者没有关于派生类型的静态类型知识，实例就会被切成基类型。例如，假如我们这样一个定义：

```
// In file Clock.ice:

class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
};

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};

// In file DateTime.ice:

#include <Clock.ice>

class DateTime extends TimeOfDay {
```

```

    short day;           // 1 - 31
    short month;         // 1 - 12
    short year;          // 1753 onwards
};

```

因为 `DateTime` 是 `TimeOfDay` 的子类，服务器可以从 `getTime` 返回 `DateTime` 实例，而客户可以把 `DateTime` 实例传给 `setTime`。在这种情况下，如果客户和服务器都链接了为 `Clock.ice` 和 `DateTime.ice` 生成的代码，它们就会分别收到实际的 `DateTime` 派生实例，也就是，实例的实际运行时类型得到了保留。

拿上面的情况与这样的情况对比一下：服务器链接了根据 `Clock.ice` 和 `DateTime.ice` 生成的代理，但客户只链接了为 `Clock.ice` 生成的代码。也就是，服务器懂得 `DateTime` 类型，能从 `getTime` 返回 `DateTime` 实例，而客户只懂 `TimeOfDay`。在这种情况下，服务器返回的 `DateTime` 派生实例会在客户中被切成它的 `TimeOfDay` 基类型（对于客户而言，这个实例的派生部分中的信息就丢了）。

如果你需要多态的*值*（而不是多态的*接口*），类继承会很有用。例如：

```

class Shape {
    // Definitions for shapes, such as size, center, etc.
};

class Circle extends Shape {
    // Definitions for circles, such as radius...
};

class Rectangle extends Shape {
    // Definitions for rectangles, such as width and length...
};

sequence<Shape> ShapeSeq;

interface ShapeProcessor {
    void processShapes(ShapeSeq ss);
};

```

注意 `ShapeSeq` 的定义，以及它怎样被用作传给 `processShapes` 操作的参数：类继承允许我们传递多态的形状序列（而不必为每种形状定义单独的操作）。

`ShapeSeq` 的接收者可以遍历序列的各个元素，并把每个元素向下转换成它实际的运行时类型（接收者还可以向每个元素要它的类型 ID，用以确定它的类型——参见 6.14.1 节和 8.11.2 节）。

4.9.4 类用作联合

Slice 没有提供专门的联合，因为那是多余的。你可以从共同的基类派生多个类，从而得到与使用联合相同的效果：

```
interface ShapeShifter {
    Shape translate(Shape s, long xDistance, long yDistance);
};
```

`translate` 操作的参数 `s` 可被视为两个成员的联合：`Circle` 和 `Rectangle`。Shape 实例的接收者可以用实例的类型 ID（参见 4.12 节）来确定它收到的是 `Circle` 还是 `Rectangle`。另外，如果你想要得到一种与传统的区分性联合（discriminated union）更相像的东西，你可以采用这种做法：

```
class UnionDiscriminator {
    int d;
};

class Member1 extends UnionDiscriminator {
    // d == 1
    string s;
    float f;
};

class Member2 extends UnionDiscriminator {
    // d == 2
    byte b;
    int i;
};
```

在这种做法里，`UnionDiscriminator` 类提供了一种区分符。联合的“成员”是从 `UnionDiscriminator` 派生的类。对于每一个派生类，区分符都有一个不同的值。这种联合的接收者可以在 `switch` 语句中使用区分符的值，以选择活动的联合成员。

4.9.5 自引用的类

类可以自引用。例如：

```
class Link {
    SomeType value;
    Link next;
};
```


这看起来与第 88 页上的自引用接口例子非常像，但其语义却非常不同。注意 `value` 和 `next` 是数据成员，而不是操作，而 `next` 的类型是 `Link`（不是 `Link*`）。如你可能会预期的一样，这个定义形成了与第 88 页上的 `Link` 接口相同的链表：`Link` 类的每个实例都含有一个 `next` 成员，指向链中的下一个链接；最后一个链接的 `next` 成员包含的是 `null` 值。所以，这看起来像是一个类在包括自身，但表达的却是指针语义：`next` 数据成员含有一个指针，指向链中的下一个链接。

这时你可能在想，那么第 88 页上的 `Link` 接口和第 96 页上的 `Link` 类有什么区别？它们的区别是，类具有 *值* 语义，而代理具有 *引用* 语义。为了说明这一点，让我们再考察一下第 88 页上的 `Link` 接口：

```
interface Link {
    nonmutating SomeType getValue();
    nonmutating Link* next();
};
```

在这个定义里，`getValue` 和 `next` 都是操作，而 `next` 的返回值是 `Link*`，也就是说，`next` 返回的是 *代理*。代理具有 *引用* 语义，也就是说，它代表在某个地方的对象。如果你调用 `Link` 代理上的 `getValue` 操作，就会有一条消息发往这个代理的 `servant`（可能在远地）。换句话说，对于代理而言，对象呆在它的服务器进程中，而我们通过远地过程调用访问这个对象的状态。把这个定义与我们 `Link` 类的定义比较一下：

```
class Link {
    SomeType value;
    Link next;
};
```

在这个定义里，`value` 和 `next` 都是数据成员，而 `next` 的类型是具有 *值* 语义的 `Link`。特别地，尽管 `next` 的“观感”（look and feel）像是指针，*它不能代表在另外的地址空间中的实例*。这意味着，如果我们有一个 `Link` 实例链，所有这些实例都在我们的本地地址空间中，而当我们读写某个值数据成员时，我们就是在进行本地的地址空间操作。这意味着，像 `getHead`

这样返回一个 `Link` 实例的操作，返回的不只是链头，而是整个链。如图 4.8 所示：

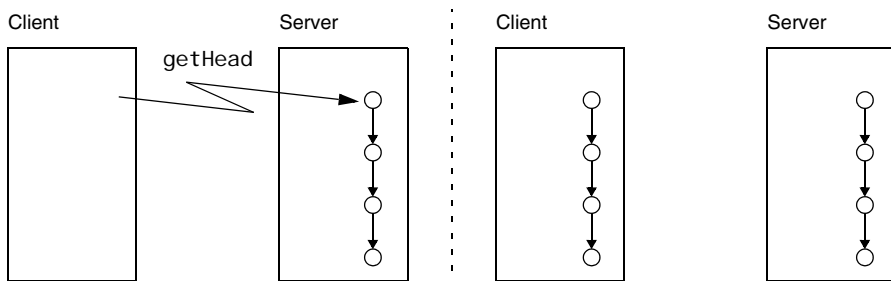


图 4.8. 类版本的 `Link` 在调用 `getHead` 之前和之后的情况

另一方面，对于接口版本的 `Link`，我们不知道所有的链接在物理上都是在哪里实现的。例如，有四个链接的链可以让每个对象实例都有自己的物理服务器进程；这些服务器进程可以分别放在不同的大陆上。如果你有一个代理，指向这个链的头，并且通过遍历每个链接上的 `next` 操作来遍历这个链，那么你就会发送四个远地过程调用，给每个对象一个。

在为图（`graphs`）建模时，自引用的类特别有用。例如，我们可以通过以下几行定义创建一个简单的表达式树：

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand extends Node {
    long val;
};
```

这个表达式树由类型为 `Operand` 的叶节点，以及类型为 `UnaryOperator` 和 `BinaryOperator` 的内部节点组成，这些节点分别有一个或两个后代。所有这三个类都派生自共同的基类 `Node`。注意 `Node` 是一个空类。在相当少的一些情况下，使用空类是正当的，这里就是其中一种情况。（参见第 88 页上的讨论：一旦我们给这个类层次增加了操作（参见 4.9.7 节），基类就不再是空的了）。

例如，如果我们编写了一个操作，其参数是一个 `Node`，那么传递该参数就会造成整个树被传送给服务器：

```
interface Evaluator {  
    long eval (Node expression); // Send entire tree for evaluation  
};
```

自引用的类不仅能用于非循环图，也能用于循环图；`Ice run time` 允许循环：它保证不会发生资源泄漏，而且无限循环会在整编过程中得以避免。

4.9.6 类 vs. 结构

一个明显的问题：既然类显然可用于创建结构，`Ice` 为何既提供结构，又提供类？答案与实现的代价有关：类提供了一些特性，是结构所没有的：

- 类支持继承。
- 类可以自引用。
- 类可以有操作（参见 4.9.7 节）。
- 类可以实现接口（参见 4.9.9 节）。

显然，类的这些额外的特性会带来实现上的代价，无论是生成的代码的尺寸，还是在运行时消耗的内存和 CPU 周期的数量。而在另一方面，结构是值的简单集合（“`plain old structs`”），所用的实现机制非常高效。这就意味着，与使用类相比，使用结构，你可以期望得到更好的性能，占用更少内存（特别是直接支持“`plain old structures`”的语言，比如 `C++` 和 `C#`）。只有当你需要类的更强大的特性时，你才应该使用类。

4.9.7 有操作的类

除了数据成员，类还可以有操作。类的操作定义的语法和接口的操作的语法是相同的。例如，我们可以这样修改 4.9.5 节的表达式树：

```

enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};

```

与 4.9.5 节的版本相比，唯一的变化是 `Node` 类现在有了 `eval` 操作。这样的操作的语义和 C++ 中的虚成员函数是一样的：每个派生类都从其基类那里继承操作，并且可以替换操作的定义。对于我们的表达式树而言，`Operand` 类提供了一种实现，简单地返回其 `val` 成员的值，而 `UnaryOperator` 和 `BinaryOperator` 类提供的实现会计算它们各自的子树的值。如果我们调用某个表达式树的根节点上的 `eval`，那么无论我们拥有的是一个复杂的表达式，还是一棵仅由一个 `Operand` 节点组成的树，这个操作返回的都是这个表达式树的值。

类上的操作总是在调用者的地址空间中执行，也就是说，类的操作是本地操作。因此，调用类上的操作并不会产生远地过程调用。当然，这马上就会引发一个有意思的问题：如果客户从服务器那里接收到一个有操作的类，但客户和服务器的实现是用不同的语言实现的，会发生什么事情？有操作的类要求接收者提供类实例工厂。**Ice run time** 只整编类的数据成员。如果类有操作，类的接收者必须提供一个类工厂，在接收者的地址空间里实例化这个类，接收者还要负责提供类的操作的实现。

因此，如果你使用了有操作的类，客户和服务器的实现应该能各自访问这个类的操作的一种实现。在线路上不会发送代码（如果一个环境由异种节点组成，使用的是不同的操作系统和语言，发送代码是不可行的）。

4.9.8 类在架构上的影响

类在架构上有一些影响，值得详细探讨一下。

没有操作的类

没有使用继承、只有数据成员的类不会带来架构问题：它们就是一些值，会像其他任何值一样整编，这样的值的例子有序列、结构，或词典。如果类使用了派生，也不会带来问题：如果派生实例的接收者知道这种派生类型，它就会接收派生的类型；否则，这个实例就会切成接收者所知道的、派生层次最深的类型。这样，当系统随时间推移而扩展时，类继承会很有用：你可以创建派生类，无需一次性升级系统的所有部分。

有操作的类

有操作的类需要我们多进行一些思考。这里有一个例子：假定你正在创建一个 Ice 应用，其中的 Slice 定义使用了相当一些有操作的类。你销售你的客户和服务端（都用 Java 编写），最后部署了几千个系统。

随着时间的推移和需求的变化，你注意到用 C++ 编写的客户会很有市场。出于商业上的考虑，你希望让用户或第三方来开发 C++ 客户，但这时你发现了一个小问题：你的应用有许多有操作的类，比如：

```
class ComplexThingForExpertsOnly {  
    // Lots of arcane data members here...  
    MysteriousThing mysteriousOperation(/* parameters */);  
    ArcaneThing arcaneOperation(/* parameters */);  
    ComplexThing complexOperation(/* parameters */);  
    // etc...  
};
```

这些操作所做的事情到底是什么并不重要（推测起来，你曾出于性能上的考虑，决定把你的应用的部分处理工作转移到客户端）。现在你想让其他开发者编写 C++ 客户，结果，这些开发者必须为所有的客户端操作提供实现，而且，这些实现必须与你的 Java 实现的语义完全吻合——只有这样，你的应用才能够工作。取决于这些操作所做的事情，用不同的语言编写语义上严格等价的实现也许并不是轻而易举的事情，所以你决定由你自己提供 C++ 实现。但现在，你发现了另一个问题：C++ 客户需要支持多种操作系统，它们使用的是许多不同的 C++ 编译器。突然间，你的任务变得相当让人畏缩：实际上，你需要为客户所用的操作系统和编译器版本的所有组合提供实现。考虑到各种编译器对 ISO C++ 标准的遵从程度不同，而不同的操作系统的状况又错综复杂，你也许会发现，你自己面临的开发任务比预期的要大得多。当然，如果你需要再用另外一种语言编写客户实现，同样的事情还会再度发生。

这个故事的寓意并非是你应该避免使用有操作的类；它们能够显著地提高性能，并不一定就很糟糕。但你要记住，一旦你使用了有操作的类，你实际上就是在使用客户端原生代码，因此，你不再能享受到接口所提供的实现透明性。这意味着，只有当你能够紧紧地控制客户的部署环境时，你才应该使用有操作的类。如果不是这样，你就最好使用接口和没有操作的类。这样，所有的处理就将在服务器上进行，客户和服务端之间的合约就会由 Slice 定义单独提供，而不会附加上客户端代码的语义——在使用有操作的类时必须提供这些代码。

持久的类

Ice 还提供了内建的持久机制，只需完成非常少的实现工作，你就能在数据库中存储类的状态。要访问这些持久特性，你必须定义一个 Slice 类，其成员存储的是类的状态。我们将在第 21 章详细讨论 Slice 的持久特性。

4.9.9 实现接口的类

Slice 类也可以用作服务器中的 servant，也就是说，类的实例可以用来提供接口的行为，例如：

```
interface Time {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

class Clock implements Time {
    TimeOfDay time;
};
```

关键字 `implements` 表明 `Clock` 类提供了 `Time` 接口的一种实现。这个类可以提供自己的数据成员和操作；在上面的例子中，`Clock` 类存储的是当前时间，可以通过 `Time` 接口访问。一个类可以实现若干接口，例如：

```
interface Time {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

interface Radio {
    idempotent void setFrequency(long hertz);
    idempotent void setVolume(long dB);
};
```

```
class Radi oClock implements Time, Radi o {
    TimeOfDay time;
    Long hertz;
};
```

Radi oCl ock 类既实现 Ti me 接口，又实现 Radi o 接口。
除了实现接口，一个类还可以扩展另一个类：

```
interface Time {
    nonmutating TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};

class Cl ock implements Time {
    TimeOfDay time;
};

interface Al armCl ock extends Time {
    nonmutating TimeOfDay getAl armTime();
    idempotent void setAl armTime(TimeOfDay al armTime);
};

interface Radi o {
    idempotent void setFrequency(Long hertz);
    idempotent void setVol ume(Long dB);
};

class Radi oAl armCl ock extends Cl ock
    implements Al armCl ock, Radi o {
    TimeOfDay al armTime;
    Long hertz;
};
```

这些定义产生了如图 4.9 所示的继承图：

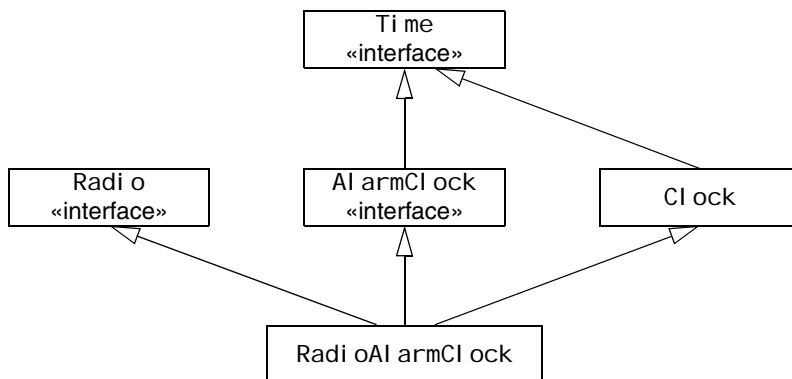


图 4.9. 一个使用实现和接口继承的类

在这个定义中，Radi o 和 Al armCl ock 是抽象的接口，而 Cl ock 和 Radi o-Al armCl ock 是具体的类。和 Java 中的情况一样，一个类可以实现多个接口，但最多只能扩展一个类。

4.9.10 类继承的局限

和接口继承一样，类不能重新定义它从基接口或基类继承来的操作或数据成员。例如：

```

interface BaseInterface {
    void op();
};

class BaseClass {
    int member;
};

class DerivedClass extends BaseClass implements BaseInterface {
    void someOperation();           // OK
    int op();                       // Error!
    int someMember;                 // OK
    long member;                    // Error!
};
  
```


4.9.11 传值 vs. 传引用

我们在 4.9.5 节已经看到，类自然就支持传值语义：如果你传递一个类，就会把这个类的数据成员传送给接收者。接收者对这些数据成员所做的变动，都只会影响接收者的类副本；接收者做出的变动不会影响发送者的类的数据成员。

除了通过值来传递类，你还可以通过引用来传递类。例如：

```
class TimeOfDay {
    short hour;
    short minute;
    short second;
    string format();
};

interface Example {
    TimeOfDay* get(); // Note: returns a proxy!
};
```

注意，`get` 操作返回的是 `TimeOfDay` 类的代理，而不是 `TimeOfDay` 实例的代理。其语义是这样的：

- 当客户从 `get` 调用那里收到一个 `TimeOfDay` 代理时，它所持有的代理与某个接口的普通代理没有区别。
- 客户可以通过这个代理祈用操作，但不能访问数据成员。这是因为代理没有数据成员的概念，它们代表的是接口：尽管 `TimeOfDay` 类有数据成员，通过代理只能访问它的操作。

实际效果就是，在前面的例子中，服务器持有 `TimeOfDay` 类的一个实例。这个实例的一个代理被传给了客户。客户只能用这个代理祈用 `format` 操作。这个操作的实现由服务器提供，当客户祈用 `format` 时，它发送一条 RPC 消息给服务器，就和它祈用接口上的操作时所做的一样。`format` 操作的实现完全由服务器决定（服务器可能会使用它所持有的 `TimeOfDay` 实例的数据成员，返回一个含有时间的串给客户）。

上面的例子看起来好像只是为类设计的，但它在类实现了接口的情况下也完全有意义：你的应用的某些部分可以通过值来交换类实例（也就是交换状态），而系统的另外一些部分可以把这些实例当作远地接口。例如：

```
interface Time {
    string format();
    // ...
};

class TimeOfDay implements Time {
```

```

    short hour;
    short minute;
    short second;
};

interface I1 {
    TimeOfDay get();           // Pass by value
    void put(TimeOfDay time); // Pass by value
};

interface I2 {
    Time* get();               // Pass by reference
};

```

在这个例子中，处理 I1 接口的客户知道 TimeOfDay 类，并通过值传递它，而处理 I2 接口的客户只与 Time 接口打交道。但是，服务器中的 Time 接口的实现使用了 TimeOfDay 实例。

这个系统混用了传值和传引用语义，在设计这样的系统时要小心。除非你清楚系统的哪些部分与接口（传引用）方面打交道，哪些部分与类（传值）方面打交道，否则你开发出的系统就会很混乱，不会对你有帮助。

在 Freeze（see 第 21 章）中，你可以找到实际使用这个特性的好例子。Freeze 允许你把类增加到已有的接口，从而实现持久能力。

4.10 提前声明

接口和类都可以进行提前声明。提前声明能够让你创建相互依赖的对象，例如：

```

interface Child;           // Forward declaration

sequence<Child*> Children; // OK

interface Parent {
    Children getChildren(); // OK
};

interface Child {           // Definition
    Parent* getMother();
    Parent* getFather();
};

```

如果不提前声明 `Child`，这个定义显然无法编译，因为 `Child` 和 `Parent` 是相互依赖的接口。你可以使用提前声明的接口和类来定义类型（比如前面这个例子中的 `Children` 序列）。提前声明的接口和类可以用作结构、异常，或类成员的类型，用作词典的值类型，用作操作的参数和返回类型。但是，在编译器见到提前声明的接口或类的定义之前，你不能继承它们：

```
interface Base;                                // Forward declaration

interface Derived1 extends Base {};            // Error!

interface Base {};                             // Definition

interface Derived2 extends Base {};            // OK, definition was seen
```

上述规则是必要的，因为如果不这样，编译器就无法保证派生的接口没有对出现在基接口中的操作进行重新定义⁹。

4.11 模块

全局名字空间的污染是大型系统中的常见问题：随着时间的推移，一些孤立的系统被集成起来，发生名字冲突的可能性会变得相当大。`Slice` 提供了 `module` 语言成分来减轻这一问题：

```
module MutableReals {
    module WishClient {
        // Definitions here...
    };
    module WishServer {
        // Definitions here...
    };
};
```

模块可以包含任何合法的 `Slice` 语言成分，包括其他的模块定义。使用模块来把相关的定义归总在一起，能够避免污染全局名字空间，并使偶然发生名字冲突的可能性降到相当低的程度（你可以用一个众所周知的名字，比如公司或产品的名称，来做最外层的模块的名字）。

模块可以重新打开：

9. 可以使用多遍编译器，但那样会增加复杂性，并不值得。

```

module MutableRealms {
    // Definitions here...
};

// Possibly in a different source file:

module MutableRealms { // OK, reopened module
    // More definitions here...
};

```

对于较大的项目，重新打开模块是有用的：它们能让你把一个模块的内容分散在若干个不同的源文件中。这样做的好处是，当开发者改动了模块的某一部分时，只有依赖于变动部分的文件需要重新编译（不必重新编译使用该模块的所有文件）。

模块分别映射到 C++ 的 namespace 和 Java 的 package。所以，你可以使用适当的 C++ using 或 Java import 声明，不让源码中出现过长的标识符。

作用域限定操作符 :: 能让你引用非局部作用域中的类型。例如：

```

module Types {
    sequence<long> LongSeq;
};

module MyApp {
    sequence<Types::LongSeq> NumberTree;
};

```

在这个定义中，受到限定的名字 Types::LongSeq 引用的是 Types 模块中定义的 LongSeq。前置的 :: 表示全局作用域，所以我们可以用 ::Types::LongSeq 来引用这个 LongSeq。

作用域限定操作符还能让你在不同的模块中定义相互依赖的接口。下面这种直接的做法行不通：

```

module Parents {
    interface Children::Child; // Syntax error!
    interface Mother {
        Children::Child* getChild();
    };
    interface Father {
        Children::Child* getChild();
    };
};

module Children {
    interface Child {

```

```

        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};

```

之所以行不通，是因为在语法上，在另外一个模块中提前声明接口是非法的。要让它工作，我们必须重新打开模块：

```

module Children {
    interface Child; // Forward declaration
};

module Parents {
    interface Mother {
        Children::Child* getChild(); // OK
    };
    interface Father {
        Children::Child* getChild(); // OK
    };
};

module Children { // Reopen module
    interface Child { // Define Child
        Parents::Mother* getMother();
        Parents::Father* getFather();
    };
};

```

尽管这种技术是可行的，其价值却有点可疑：按照定义，相互依赖的接口紧密地耦合在一起。另一方面，模块的用途就是，把相关的定义放进同一个模块，把不相关的定义放进不同的模块。当然，这带来了一个问题：如果接口如此紧密地相关联，以至于它们相互依赖，那它们又为何要在不同的模块中定义呢？为了清晰起见，你应该避免这样使用模块，即使这样做是合法的。

4.12 类型 ID

你定义的每种 Slice 类型都有一个内部的类型标识符，称为它的类型 ID。类型 ID 就是每种类型的受到了完全限定（fully-qualified）的名字。例如，前面的例子中的 Child 接口的类型 ID 是 ::Children::Child。所有的类型 ID 都以 :: 起头，所以 Parents 模块的类型 ID 是 ::Parents（不是 Parents）。一般而言，类型 ID 起头是全局作用域 (::)，然后在后面附加

包含该类型的各嵌套模块的名字，最后再以该类型自身的名字结束——这就是该类型的受到了完全限定的名字；类型 ID 的各组成部分之间用 :: 分隔。

Ice run time 在内部把类型 ID 用作每种类型的唯一标识符。例如，当某个异常被引发时，在线路上，整编过后的异常的最前面是它的类型 ID，返回给客户的就是这种形式的异常。客户端 run time 会首先读取类型 ID，然后基于这个信息，以与该异常的类型相适宜的方式解编数据的余下部分。

ice_isA 操作也使用了类型 ID（参见第 110 页）。

4.13 Object 上的操作

Object 接口有许多操作。我们不能用 Slice 定义 Object 类型，因为 Object 是一个关键字；不管怎样，如果 Object 可以有一个合法的 Slice 定义，那么它的部分看起来像是这样的：

```
sequence<string> StrSeq;
```

```
interface Object {                                // "Pseudo" Slice!
    void    ice_ping();
    bool    ice_isA(string typeId);
    string  ice_id();
    StrSeq  ice_ids();
    // ...
};
```

注意，在这个定义里，除了非法地使用了关键字 Object 来做接口名，各个操作名都含有一个下划线。这是故意的：在这些操作名里放入下划线，这些内建的操作就不可能和用户定义的操作发生冲突。这意味着，所有的 Slice 接口都可以继承 Object，而不会发生名字冲突。有三种常用的内建操作：

- ice_ping

所有接口都支持 ice_ping 操作。这个操作对调试很有用，因为它能够提供一种基本的能力，测试某个对象是否可到达：如果该对象存在，而且消息能够成功地分派给它，ice_ping 就会成功返回。如果该对象不能到达，或者不存在，ice_ping 就会抛出一个运行时异常，说明失败的原因。

- ice_isA

ice_isA 操作的参数是一个类型标识符（比如 ice_id 返回的标识符），它会测试目标对象是否支持指定的这个类型，如果是就返回

true。你可以用这个操作来检查目标对象是否支持特定的类型。例如，让我们再看一下图 4.7，假定你持有一个代理，指向的是一个 `AI armClock` 类型的对象。表 4.2 给出了用各种参数调用该代理上的 `ice_isA` 所得到的结果：

Table 4.2. 调用一个代理上的 `ice_isA`，这个代理代表的是 `AI armClock` 类型的对象

Argument	Result
<code>::Ice::Object</code>	true
<code>::Clock</code>	true
<code>::AI armClock</code>	true
<code>::Radio</code>	false
<code>::RadioClock</code>	false

和我们预期的一样，对于 `::Clock` 和 `::AI armClock`，`ice_isA` 返回真，对于 `::Ice::Object`，`ice_isA` 也返回真（因为所有的接口都支持这种类型）。显然，一个 `AI armClock` 既不支持 `Radio` 接口，也不支持 `RadioClock` 接口，所以对于这些类型，`ice_isA` 返回假。

- `ice_id`

`ice_id` 操作返回某个接口的派生层次最深的类型的 ID（参见 4.12 节）。

- `ice_ids`

`ice_ids` 操作返回一个类型 ID 序列，其中含有某个接口所支持的所有类型的 ID。例如，对于图 4.7 中的 `RadioClock` 接口，`ice_ids` 返回的序列含有这样一些类型 ID：`::Ice::Object`、`::Clock`、`::AI armClock`、`::Radio`，以及 `::RadioClock`。

4.14 本地类型

为了访问 Ice run time 的某些特性，你必须使用库所提供的一些 API。但是，Ice 并没有为各种实现语言定义一种专门的 API，而是通过 `Slice`、使用 `local` 关键字来定义它的 API。用 `Slice` 定义 API 的优点是，只需一种定义，就

足以为所有可能的实现语言定义 API。然后 Slice 编译器会为每种实现语言生成该语言专用的实际 API。Ice 库提供的类型是用 Slice `local` 关键字定义的。例如：

```
module Ice {
    local interface ObjectAdapter {
        // ...
    };
};
```

任何 Slice 定义（不止是接口）都可以有 `local` 修饰符。如果使用了 `local` 修饰符，Slice 编译器不会为相应的类型生成整编代码。这意味着，本地类型永远不能从远地访问，因为它不能在客户和服务端之间传送（Slice 编译器不允许你在非 `local` 上下文中使用 `local` 类型）。

此外，本地接口和本地类也没有继承 `Ice::Object`。相反，本地接口和类有它们自己的、完全独立的继承层次。如图 4.10 所示，这个层次的根是 `Ice::LocalObject` 类型。

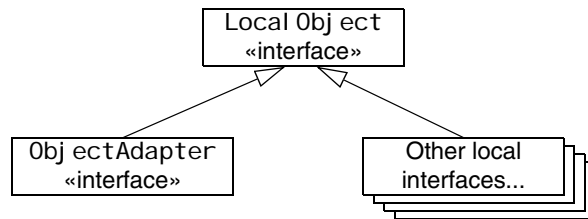


图 4.10. 以 `LocalObject` 为根的继承层次

因为本地接口形成了完全独立的继承层次，在需要非本地接口的地方，你不能使用本地接口，反之亦然。

你很少需要为你自己的应用定义本地类型——`local` 关键字之所以存在，主要是为了定义 Ice run time 的 API（因为本地对象不能从远地调用，应用定义本地对象的意义不大；你完全可以定义你所用的编程语言的普通对象）但这条规则有一个例外：`servant` 定位器必须作为本地对象实现（参见 16.6 节）。

4.15 Ice 模块

除了少数不能在 Ice 中表示的、专用于特定语言的调用，Ice run time 的 API 都是在 Ice 模块中定义的。换句话说，整个 Ice API 的大部分内容都是

作为 Slice 定义表示的。这样做的好处是，只需一种定义，就足以为 Ice 支持的所有语言定义 Ice API。然后，各种语言的映射规则会确定每种实现语言的每种 Ice API 的确切形式。

我们将在本书的余下部分渐进地探索 Ice 模块的内容。

4.16 名字与作用域

Slice 有一些关于标识符的规则。你通常无需考虑这些规则。但偶尔，了解 Slice 怎样使用名字作用域，怎样查找标识符，也会有好处。

4.16.1 名字作用域

下列 Slice 成分会建立一个名字作用域：

- 全局（文件）作用域
- 模块
- 接口
- 类
- 结构
- 异常
- 枚举
- 参数列表

在一个名字作用域内，标识符必须是唯一的，也就是说，你不能把同一个标识符用于不同的目的。例如：

```
interface Bad {  
    void op(int p, string p);    // Error!  
};
```

因为一个参数列表形成了一个名字作用域，把同一个标识符 `p` 用于不同的参数是非法的。与此类似，数据成员、操作名、接口和类名，等等，在它们的作用域内都必须是唯一的。

在一个名字作用域内，一个标识符是在第一次使用时引入的；在此之后，这个名字作用域内，这个标识符不能改变含义。例如：

```
sequence<string> Seq;

interface Bad {
    Seq op1();           // Seq introduced here
    int Seq();           // Error, Seq has changed meaning
};
```

`op1` 的声明用 `Seq` 做它的返回类型，从而把 `Seq` 引入了 `Bad` 接口的作用域。在此之后，`Seq` 只能用作表示串序列的类型名，所以编译器会认为第二个操作的声明有问题。

4.16.2 大小写敏感性

如果两个标识符只有大小写不同，将被认为是相同的，所以在一个名字作用域内，你必须使用不只是大小写不同的标识符。例如：

```
struct Bad {
    int m;
    string M; // Error!
};
```

`Slice` 编译器还要求你在使用标识符时，始终使用同样的大小写。一旦你定义了一个标识符，在此之后你必须使该标识符保持同样的大小写。例如，下面的定义是错误的：

```
sequence<string> StringSeq;

interface Bad {
    stringSeq op(); // Error!
};
```

注意，标识符不能只在大小写上与 `Slice` 的关键字不同。例如，下面的定义是错误的：

```
interface Module { // Error, "module" is a keyword
    // ...
};
```

4.16.3 嵌套作用域中的名字

在外层作用域中定义的名字可以在内层作用域中重新定义。例如，下面的定义是合法的：

```

module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;
    };
};

```

在 Inner 模块内，名字 Seq 指的是 short 类型的值的序列，它使得 Outer::Seq 的定义隐藏了起来。使用显式的作用域限定符，你仍然可以引用另外这个定义，例如：

```

module Outer {
    sequence<string> Seq;

    module Inner {
        sequence<short> Seq;

        struct Confusing {
            Seq          a;          // Sequence of short
            ::Outer::Seq b;          // Sequence of string
        };
    };
};

```

不用说，你应该避免进行这样的重定义——它们会使阅读者难以理解你的规范的含义。

名字相同的成分不能直接相互嵌套。例如，名叫 M 的模块不能包含任何也叫作 M 的成分。对于接口、类、结构、异常，以及操作而言，情况也是如此。例如，下面的例子全都有错误：

```

module M {
    interface M { /* ... */ }; // Error!
};

interface I {
    void I(); // Error!
    void op(string op); // Error!
};

struct S {
    long s; // Error, even if case differs!
};

```

之所以做出这样的限制，是因为拥有相同名字的嵌套类型难以映射到某些语言。例如，C++ 和 Java 保留了类名作为构造器的名字，所以，如果没有采用人为的规则来避免发生名字冲突，接口 I 就不能包含名叫 I 的操作。为了简单起见，Slice 禁止使用这样的成分。

4.16.4 名字查找规则

在搜索某个名字的定义时，编译器首先会反向搜索这个名字的定义的当前作用域。如果能在当前作用域中找到这个名字，它就使用这个定义。否则，编译器就会继续搜索外围的作用域，直到到达全局作用域。下面有一个对此加以说明的例子：

```
sequence<double> Seq;

module M1 {
    sequence<string> Seq;           // OK, hides ::Seq

    interface Base {
        Seq op1();                 // Returns sequence of string
    };
};

module M2 {
    interface Derived extends M1::Base {
        Seq op2();                // Returns sequence of double
    };

    sequence<bool> Seq;           // OK, hides ::Seq

    interface I {
        Seq op();                 // Returns sequence of bool
    };
};

interface I {
    Seq op();                     // Returns sequence of double
};
```

注意，M2::Derived::op2 返回的是 double 序列，而 M1::Base::op1 返回的是 string 序列。也就是说，在确定派生接口中的某个类型的含义时，基接口中的类型并不相干——编译器总是只在当前作用域及外围作用域中搜索定义，而绝不会从基接口或基类中取得某个名字的含义。

4.17 元数据

Slice 具有元数据指令的概念。例如：

```
["j java: type: java.util.LinkedList"] sequence <int> IntSeq;
```

元数据指令可以作为任何 Slice 定义的前缀出现。元数据指令出现在一对方括号中，含有一个或多个由逗号分隔的串直接量。例如，下面的元数据指令含有两个串，在语法上是有效的：

```
["a", "b"] interface Example {};
```

元数据指令在本质上不是 Slice 语言的组成部分：元数据指令的存在对客户—服务器合约没有影响，也就是说，元数据指令不会以任何方式改变 Slice 类型系统。相反，元数据指令意在用于特定的后端，比如特定语言映射的代码生成器。在上面的例子中，j java: 前缀表明这条指令意在用于 Java 代码生成器。

元数据指令可用于提供辅助性的信息，这些信息不会改变正在定义的 Slice 类型，但却会以某种方式影响编译器为这些定义生成代码的方式。例如，元数据指令 j java: type: java.util.LinkedList 指示 Java 代码生成器，把序列映射到链表，而不是数组（后者是缺省方式）。

元数据指令还用于创建支持 AMI 和 AMD 的代理及骨架，也即异步方法祈用和异步方法分派（参见第 17 章）。

除了与特定定义系在一起的元数据指令，还存在全局的元数据指令。例如：

```
["j java: package: com.acme"]]
```

注意，全局的元数据指令包围在两对方括号中，而局部的元数据指令（与特定定义系在一起的元数据指令）则在一对方括号中。全局的元数据指令用于传递能影响整个编译单元的指令。例如，上面的元数据指令指示 Java 代码生成器，把生成的源文件的内容放进 Java package com.acme 中。在文件中，全局的元数据指令必须放在所有定义之前（但可以出现在 #include 指令后面）。

我们将在适当的相关章节中讨论具体的元数据指令。

4.18 使用 Slice 编译器

Ice 为每种语言映射提供了单独的 Slice 编译器。对于 C++ 映射，编译器的可执行程序是 **slice2cpp**；对于 Java 映射，是 **slice2java**。这两种编译器的命令行语法都是：

```
<compiler-name> [options] file...
```

不管你使用的是哪一种编译器，有一些命令行选项是共通的（要了解特定语言映射专用的选项，请查阅相关的语言映射章节）。这些共通的命令行选项是：

- **-h, --help**
显示帮助信息。
- **-v, --version**
显示编译器版本。
- **-DNAME¹⁰**
定义预处理器符号 **NAME**。
- **-DNAME=DEF¹⁰**
定义预处理器符号 **NAME**，其值为 **DEF**。
- **-UNAME¹⁰**
解除预处理器符号 **NAME** 的定义。
- **-IDIR**
在 `#include` 指令的搜索路径中增加目录 **DIR**。
- **--include-dir DIR**
把 **DIR** 用作 Ice 提供的 Slice 定义的搜索目录。
- **--output-dir DIR**
把生成的文件放进目录 **DIR**。
- **--dll-export SYMBOL**
把 **SYMBOL** 用于 DLL 导出（在非 Windows 平台上会忽略这个选项）
- **-d, --debug**
打印调试信息，显示 Slice 编译器的操作。

10.注意，在 Slice 未来的版本中，可能不再支持预处理器，所以使用这些选项要慎重。

- **--ice**

启用正常情况下保留的标识符前缀 Ice。应该只在编译 Ice run time 的源码时使用这个选项。

Slice 编译器允许你编译不止一个源文件，所以你可以同时编译若干 Slice 定义：

```
slice2cpp -I. file1.ice file2.ice file3.ice
```

4.19 Slice 与 CORBA IDL 的对比

拿 Slice 和 CORBA IDL 对比一下，富有启发意义，因为两种语言不同的特性集说明了许多设计原则。在这一节，我们将简要地对比两种语言，并解释每种特性存在或不存在的缘由。

Slice 既不是 CORBA IDL 的子集，也不是它的超集。相反，Slice 取消了一些特性，也增加了一些特性。总的结果就是，我们得到了一种比 CORBA IDL 更简单、更强大的规范语言。我们将在下面的小节里看到这一点。

4.19.1 Slice 有、而 CORBA IDL 没有的特性

相对于 CORBA IDL，Slice 增加了许多特性。主要有：

- 异常继承

在 CORBA IDL 中没有异常继承，这一直在折磨 CORBA 程序员。这种缺失使得 IDL 定义不能自然地映射到有等价的原生异常支持的语言，比如 C++ 和 Java。而这又造成了结构化错误处理实现起来很困难。

结果，CORBA 应用常常会使用过多的异常处理器，跟在每个调用或调用块后面，或者，走到另一极端，只在过高的层面上进行一般化的异常处理，以致于无法获得有用的诊断信息。这样的情况迫使你进行苛刻的权衡：你或者拥有好的错误处理及诊断信息，但代码搅成一团、难以维护，或者为了保持代码整洁而牺牲错误处理。

- 词典

“我怎样把 Java 的哈希表发给服务器？”这是最常见的 CORBA 问题之一。标准答案是用结构的序列来构造哈希表，用每个结构容纳键和值，把哈希表复制进序列，发送序列，然后在另一端重新构造哈希表。

这种做法不仅会浪费 CPU 周期和内存，还会用这些数据类型污染 IDL：它们的存在只是为了弥补 CORBA 平台的局限（而不是出于应用

的需要)。Slice 词典能够让你高效地发送作为第一类概念的查找表，从而消除 CORBA 做法所带来的浪费和晦涩。

- **nonmutating 和 idempotent 操作**

知道了某个操作不会修改它的对象的状态，Ice run time 就能够透明地从暂时的错误中恢复出来；这样的错误本来必须由应用进行处理。这样，我们就得到了一个更可靠、更方便的平台。此外，nonmutating 操作可以映射到目标语言中的对应成分（如果有的话），比如 C++ const 成员函数。这改善了系统的静态类型安全性。

- **类**

Slice 提供的类既支持传值、也支持传引用语义。与此相反，CORBA 的值类型（与 Ice 的类有点类似）只支持传值语义：你无法创建一个指向值类型实例的 CORBA 引用，从远地祈用该实例。

Slice 还把类用于它的自动持久机制（参见第 21 章）。CORBA 没有提供与之等价的特性。

- **元数据**

元数据指令能够让你以一种受控的方式对语言进行扩展，而又不影响客户—服务器合约。异步方法祈用（AMI）和异步方法分派（AMD）是使用元数据指令的两个例子。

4.19.2 CORBA IDL 有、而 Slice 没有的特性

Slice 有意放弃了 CORBA IDL 的相当一些特性。它们可以宽泛地分类如下：

- **多余的特性**

CORBA IDL 的有些特性是多余的：它们提供了不止一种途径来完成一件事情。这并不是程序员所要的，原因有两个：

1. 提供不止一种途径来完成同样的事情会在使代码和数据量变大。所造成的代码膨胀还会造成性能恶化，所以应该加以避免。
2. 多余的特性既不符合工效学，又会造成混乱。与完成同一件事情的两个特性相比，一个特性既更易于学习（对于程序员），又更易于实现（对于供应商）。而且，多余的特性总会带来一种让人烦扰的不适感，特别是初学者：“我怎么能用两种不同的方式做这件事情？在什么时候一种风格比另一种风格更好？两种特性真的是等价的，还是我忽略了某种微妙的差异？”不提供不止一种方式来做同样的事情，就完全不会产生这样的问题了。

- 非特性

CORBA IDL 的许多特性没有必要存在，因为它们几乎从未被使用过。如果通过合理的方式能完成某种事情，而不用使用某种特殊用途的特性，那么这个特性就不应该存在。这样，系统就会更易学易用，更小，并且性能更高。

- 有问题的特性

CORBA IDL 的有些特性是有问题的，因为它们所做的事情如果不是完全错了，至少也是在价值上成问题的。如果一个成问题的特性造成误用的可能性，比它带来的益处还要大，那么这个特性就应该被忽略，系统就会更简单、可靠，性能也就会更高。

多余的特性

1. C++ 预处理器的使用

像 Slice 这样的规范语言无需使用预处理器。在使用 CORBA IDL 时，文件包括和双重包括块是（有判断力的）人们会使用的唯一两样东西。比 C++ 预处理器更简单、更小的机制可以取而代之。在最好的情况下，对 CORBA IDL 预处理器的其他用法是不必要的，在最坏的情况下则会使代码变得混乱。

2. IDL 属性

IDL 属性是访问器（accessor）操作（对于只读属性）或一对访问器及修改器（modifier）操作（用于可写属性）的简记法。只需直接定义访问器和修改器操作，就能实现同样的目的¹¹。

属性给 CORBA run time 和 API 引入了相当大的复杂性（例如，使用 Dynamic Invocation Interface 的程序员必须记住，要设置或获取某个属性，他们必须使用 `_get_<attribute-name>` 和 `_set_<attribute-name>`，而对于操作，则必须使用原来的名字。

3. 无符号整数

无符号整数给类型系统带来的价值非常少，但给它增加的复杂性却相当大。此外，如果目标编程语言不支持无符号整数（Java 就不支持），要处理溢出情况就会变得几乎不可能。目前，Java CORBA 程序员处理这个问题的办法是忽略它（BIBREF 非常令人信服地讨论了无符号类型的各种坏处）。

¹¹IDL 属性还是二等公民，因为在 CORBA 3.x 发布之前，不能在访问属性时抛出异常。

4. 标识符中的下划线

在很大程度上，标识符是否应该包含下划线是一个个人口味问题。但是，为语言映射保留某个范围内的标识符、以避免发生名字冲突，这是一件重要的事情。例如，如果一个 CORBA IDL 规范在同一个作用域内包含有标识符 `T` 和 `T_var`，这个规范无法映射到有效的 C++（对于 C++ 及其他一些语言，有许多这样的冲突）。

在 Slice 中不允许使用下划线，这样就保证了所有有效的 Slice 规范都能映射到有效的编程语言源码，因为在编程语言一级，下划线能够很可靠地用于避免名字冲突。

5. 数组

CORBA IDL 既提供了数组，也提供了序列（序列又进一步划分成有界序列和无界序列）。考虑到很容易用序列来构造数组，数组可以取消。数组比序列要更精确一点：你可以精确地说明需要 n 个元素，而不是最多 n 个元素。但是，表达能力上的一点提高却完全被复杂性上的代价抵消掉了：数组不仅会造成代码膨胀，还会给类型系统带来许多漏洞（数组的弱类型安全性给 CORBA C++ 映射带来的折磨比其他任何特性都多）。

6. 有界序列

在很大程度上，对数组所做的论证也适用于有界序列。有界序列在表达能力上带来的提高，并不值得让我们在语言映射中引入那么多复杂性。

7. 通过序列进行自引用的结构

CORBA IDL 允许结构拥有这样的成员：其类型就是这个结构的类型。尽管这个特性是有用的，要表达它，需要采用奇怪的、人为制造的序列成分。随着 CORBA 值类型的引入，这个特别变得多余了，因为值类型能够更干净、更优雅地支持同样的东西。

8. 通过 `#pragma version` 进行仓库 ID 版本管理

CORBA IDL 中的 `#pragma version` 指令没有任何用处。原来的意图是想要提供接口的版本管理功能。但不同的主要和次要版本号实际上并不能定义向后的兼容性（或不兼容性）的概念。相反，它们只是定义了一种新的类型，通过其他手段也能做到这一点。

非特性

1. 数组

我们先前把数组归类为冗余的特性。经验告诉我们，数组也是一个非特性：在十多年来发布的 IDL 规范里，你用一只手的手指就能数出使用了数组的地方。

2. 常量表达式

CORBA IDL 允许你用常量表达式对常量进行初始化，比如 $x * y$ 。尽管这看起来有吸引力，对于规范语言而言却并无必要。考虑到涉及到的所有值都是编译时常量，你完全可以一次性地计算出这些值，把它们直接写进规范（同样在这些年所发布的 IDL 规范里，你用一只手的手指就能数出常量表达式被使用的次数）¹²。

3. char 和 wchar 类型

在像 Slice 这样的规范语言里，根本不需要字符类型。在少数需要使用字符的地方，可以使用 string 类型¹³。

4. 定点类型

定点类型是为了支持财务应用而引入 CORBA 的，这些应用需要计算货币值（浮点类型并不适用于这种用途，因为它们不能存储足够的小数位数，而且可能会进行不合需要的舍入，或是出现表达上的错误）。

给 CORBA 增加定点类型在代码尺寸和 API 复杂性方面所造成的代价相当可观。尤其是没有原生定点类型支持的语言，必须提供支持机制来模拟这种类型。这样的代价需要一再付出，即使是通常并不会用于财务计算的语言。而且，实际上没有人用 IDL 的这些类型来进行计算——相反，IDL 只是充当了传送这些类型的手段。用串来表示定点值，并在客户和服务端中把它们转换成原生的定点类型，这样的做法完全可行（实现起来也很简单）。

5. 扩展的浮点类型

尽管有些应用确实需要使用扩展的浮点类型，如果没有底层硬件的原生支持，很难提供这个特性。结果，许多 CORBA 产品都没有实现对

12. 在 CORBA 的 2.6.x 版中，IDL 常量表达式的语义在很大程度上仍然不明确：没有类型强制转换规则，没有溢出处理规则，也没有定义二进制表示；因此，常量表达式的语义依赖于实现。

13. 我们无法回想起我们曾经看到有（非教学用的）IDL 规范使用了 char 或 wchar 类型。

扩展的浮点类型的支持。在不支持扩展的浮点类型的平台上，这种类型会被不声不响地重新映射到普通的 `double`。

有问题的特性

1. typedef

与其他任何 IDL 特性相比，IDL `typedef` 带来的复杂性、语义问题，以及应用 `bug` 都更多。`typedef` 的问题在于，它没有创建新的类型。相反，它创建的是已有类型的别名。如果明智地加以使用，类型定义可以改善规范的可读性。但是，在这样的外表之下，类型可以有别名这样一个事实会造成各种各样的问题。有许多年，整个的等价类型这种概念在 CORBA 中完全是不明确的，为了确定类型别名所造成的各种语义复杂性，在规范中有大量篇幅花在了繁琐的解释上。¹⁴

没有这个特性，复杂性就消失了（无论是在 `run time` 中，还是在 API 中）：Slice 不允许类型有别名，于是对类型的名字（因而也包括等价类型）就不可能有任何疑问了。

2. 嵌套的类型

与 C++ 类似，（举例来说）IDL 允许你在接口或异常的作用域内定义类型。这个决定造成的复杂性的数量相当惊人：名字查找的次序、确切地决定类型是何时引入某个作用域的，以及类型可以（或不可以）怎样使名字相同的其他类型隐藏起来，为了处理上述这样的问题，CORBA 规范包含了大量复杂的规则。复杂性还带进了语言映射：使用嵌套的类型定义，会产生更复杂（也更大）的源码，同时，如果语言不支持这样的嵌套定义，就会难以处理¹⁵。

Slice 只允许在全局作用域和模块作用域定义类型。经验告诉我们，这就是我们所需要的全部特性，而且它消除了所有复杂性。

3. 联合

大多数 OO 课本都会告诉你，联合并不是必需的；相反，你可以通过从基类进行派生来实现同样的事情（并且能享受由此而来的好处：可以对类的成员进行类型安全的访问）。IDL 联合是复杂性的又一来源，其复杂性与这种特性的可用性完全失去了比例。语言映射因此而受到了严重的折磨。例如，即使对于专家来说，IDL 联合的 C++ 映射也是

14.就等价类型而言，CORBA 2.6.x 仍然有一些未解决的问题。

15.在已发布的规范中，实际使用嵌套类型的次数也能用一只手的手指指数出来；而每一个 CORBA 平台都必须负担由此所带来的复杂性。

一种挑战。而且，和其他特性一样，联合也会使你在代码尺寸和运行时性能方面付出代价。

4. #pragma

IDL 允许使用 `#pragma` 指令来控制类型 ID 的内容。这是一个最不幸的选择：因为 `#pragma` 是一个预处理指令，它完全处在语言的正常作用域规则之外。由此造成的复杂性足以在规范中占据若干页篇幅（而且，即使有了规范中的解释，程序员也仍然有可能无意义地使用 `#pragma` 指令，而且还不能作为错误被诊断出来）。

Slice 不允许程序员控制类型 ID，很简单，因为那没有必要。

5. oneway 操作

IDL 允许你用 `oneway` 关键字来标记操作。增加了这个关键字之后，ORB 就会按照“尽力”语义去分派操作。在理论上，`run time` 只是发出请求，然后完全忘掉它，并不在乎请求是否会丢失。在实践中，这种做法有一些问题：

- 和普通祈用一样，`oneway` 祈用也是通过 TCP/IP 递送的。即使服务器可以不回复 `oneway` 请求，底层的 TCP/IP 实现也仍然会保证递送请求。这意味着，即使使用了 `oneway` 关键字，发送请求的 ORB 也仍然受流控制的辖制（也就是说客户可能会阻塞）。此外，在 TCP/IP 一级，即使是 `oneway` 请求，从服务器到客户也仍然会有返回的数据（以确认和流控制包的形式）。
- 在 CORBA 的早期版本中，`oneway` 祈用语义的定义很糟糕；后来为了让客户能对“递送保证”有所控制，其语义又做了精炼。遗憾的是，这又给应用 API 和 ORB 实现带来了更多的复杂性。
- 在架构上，在 IDL 中使用 `oneway` 是可疑的：IDL 是一种接口定义语言，但 `oneway` 与接口毫无关系。相反，它所控制是调用分派的一个方面，很大程度上独立于具体的接口。这就带来了一个问题：如果 `oneway` 与客户和服务器的合约毫无关系，那它又为什么是第一类的语言概念？

尽管 Slice 支持单向祈用，它没有 `oneway` 关键字。这样，我们就不至于用与类型无关的指令污染 Slice 定义了。对于异步方法祈用，Slice 使用了元数据指令。使用这样的元数据完全不会影响客户-服务器合约：如果你删除规范中的所有元数据定义，然后只重新编译客户或服务器，客户和服务器的接口合约将保持不变，并且仍然是有效的。

6. IDL 上下文

IDL 上下文是一种通用的“紧急出口”，在本质上，它允许名一串对（`name-string pairs`）的序列随每次操作祈用一起发送；服务器可以检

查这些名一串对，用它们的内容来改变自己的行为。遗憾的是，IDL 上下文完全处在类型系统之外，并且不能为客户或服务器提供任何保证：即使操作有上下文子句，也不能保证客户会发送任何指定的上下文，或发送这些上下文的正确的值。CORBA 不了解这些名一串对的含义或类型，因此，不能提供任何帮助来保证它们的内容是正确的（无论是在编译时，还是在运行时）。实际结果就是，IDL 上下文把 IDL 类型系统射穿了一个大洞，会产生难以理解、编写，以及维护的系统。

7. 宽串

CORBA 支持多代码集及字符集宽串，并且采用了一种复杂的磋商机制，允许客户和服务器就宽串的传送所用的特定代码集达成一致。这种选择带来了大量复杂性：就 CORBA 3.0 而言，许多 ORB 实现在交换宽串数据时仍然会遇到各种互操作性问题。该规范仍然含有许多未解决的问题，这使得互操作性不可能在短期内成为现实。

Slice 的宽串使用了 Unicode，也就是说，在传送宽串时，使用的是单个字符集，以及单个明确的代码集。这极大地简化了 run time 的实现，并且避免了各种互操作性问题的发生。

8. Any 类型

IDL 的 Any 类型是一种通用的容器类型，可以容纳任何 IDL 类型的值。这种特性有点类似于在 C 中，把无类型的数据当作 `void *` 来交换，但其“自省能力”（introspection）更强，所以其内容的类型可在运行时判定。遗憾的是，Any 类型增加了大量复杂性，所得却很少：

- 用于处理 Any 类型及其相关的类型描述的 API 费解而复杂。使用 Any 类型的代码极其易错（特别是在 C++ 里）。此外，语言映射需要生成大量辅助函数和操作符，从而降低编译速度，并在 run time 中占用相当多的代码和数据空间。
- 尽管 Any 类型是自描述的，同时在线路上发送的每个实例都含有对值的类型的完整（且庞大的）描述，如果不对值进行完整的解编和重整编，一个进程要想接收并重新传送 Any 类型的值，是不可能的。这会像 CORBA Event Service 这样的程序：额外的整编代价支配了总体的执行时间，并且限制了性能——对于许多应用来说，这是不可接受的。

Slice 用类取代了 IDL 联合及 Any 类型。与使用 Any 类型相比，这种途径更简单，更类型安全。此外，通过 Slice 协议，你无需解编和重整编数据，你就能接收并重新传送这些数据：与用 CORBA 构建的系统相比，通过这种方式所得到的系统更小，性能也更好。

9. 匿名类型

CORBA 的早期版本允许使用匿名的 IDL 类型（这些类型是内联定义的，没有自己的名字）。匿名类型会给语言映射带来问题，所以在 CORBA 中已成为不赞成使用的特性。但由于要保持向后兼容，匿名类型带来的复杂性仍然可以看到。Slice 确保每种类型都有名字，所以不会有匿名类型所造成的各种问题。

4.20 总结

Slice 是一种用于定义客户—服务器合约的基础性机制。你通过用 Slice 定义数据类型和接口，创建出与语言无关的 API 定义，编译器可以把这样的定义翻译成针对 C++ 或 Java 的 API。

Slice 提供了常用的内建类型，并允许你创建用户定义的、具有任意复杂度的类型，比如序列、枚举、结构、词典，以及类。多态是通过接口、类，以及异常的继承来实现的。而异常又为你提供了一些设施，可以进行高级的错误报告和处理。模块允许你把一个规范的相关部分汇总在一起，并防止污染全局名字空间；通过使用针对特定编译器后端的指令，元数据可用于对 Slice 定义进行补充说明。

第 5 章

一个简单文件系统的 Slice 定义

5.1 本章综述

本书的余下部分将使用一个文件系统应用来阐释 Ice 的各个方面。我们将在阐释过程中逐渐改进和修改这个应用，使它演化成一个现实的应用，从而说明 Ice 的架构和代码的各个方面。这样，我们就能在探索 Ice 平台的各种能力的过程中达到现实的复杂度，而又不会在早期就用各种混乱的细节淹没你。5.2 节 概述这个文件系统的功能，5.3 节开发了文件系统所需的数据类型和接口，5.4 节给出了这个应用完整的 Slice 定义。

5.2 文件系统应用

我们的文件系统应用将实现一个简单的层次结构的文件系统，就像我们在 Windows 或 UNIX 上所看到的文件系统。为了让例子代码的数量保持在可以管理的范围内，我们忽略了真实的文件系统的许多方面，比如所有权、权限、符号链接，以及其他一些特性。但我们所构建的功能足以告诉你，可以怎样实现一个功能完备的文件系统，而且我们还考虑了像性能和可伸缩性这样的问题。以这种方式，我们可以创建一个具有现实的复杂度的应用，而又不会被埋葬在大量代码中。

首先要说明的是，这个文件系统是非分布式的：尽管我们是在服务器中实现这个应用，由客户对其进行访问（所以我们可以从远地访问文件系

统)，应用的初始版本要求文件系统中的所有文件都由一个服务器提供。这就意味着，在文件系统的根之下的所有目录和文件都是在一个服务器上实现的（我们将在 XREF 中讨论怎样消除这个限制，创建真正的分布式文件系统）。

我们的文件系统由目录和文件组成。目录是可以容纳目录或文件的容器，也就是说，这个文件系统是层次结构的。在文件系统的根上有一个专用目录。每个目录和文件都有名字。其父目录相同的文件和目录必须具有不同的名字（但父目录不同的文件和目录的名字可以相同）。换句话说，目录形成了命名范围，单个目录中的各个项的名字必须唯一。你可以列出目录的内容。

目前，我们没有路径名的概念，也不能创建或销毁文件及目录。相反，服务器提供的是数目固定的目录和文件（我们将在 XREF 中处理文件的创建和销毁）。

你可以读写文件，但目前，读和写针对的总是文件的整个内容；你不可能读写文件的部分内容。

5.3 文件系统的 Slice 定义

在给出了刚才概述的非常简单的需求之后，我们可以着手设计系统的接口了。文件和目录有共同之处：它们都有名字，而且文件和目录都可以包含在目录中。这提示我们，可以基类型来提供共有的功能，用派生类型来提供目录和文件专有的功能。如图 5.1 所示：

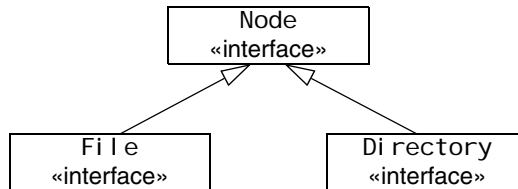


图 5.1. 文件系统的继承图

它们的 Slice 定义看起来是这样的：

```
interface Node {
    // ...
};

interface File extends Node {
    // ...
}
```

```
};

interface Directory extends Node {
    // ...
};
```

接下来，我们需要想一想每个接口应该提供什么操作。因为目录和文件都有名字，我们可以给 Node 基接口：

```
interface Node {
    nonmutating string name();
};
```

File 接口提供了用于读写文件的操作。我们暂时只处理文本文件（关于怎样处理二进制文件的讨论，参见 XREF）。为了简单起见，我们假定 read 操作永远不会失败，只有 write 操作会遇到出错的情况。于是就有了下面的定义：

```
exception GenericError {
    string reason;
};

sequence<string> Lines;

interface File extends Node {
    nonmutating Lines read();
    idempotent void write (Lines text) throws GenericError;
};
```

注意，read 被标为 nonmutating，因为这个操作不会修改文件的状态。write 操作被标为 idempotent，因为它会用 text 参数的内容取代文件的整个内容。这意味着，连续用同样的参数值两次调用这个操作是安全的：这样做的实际效果与只（成功地）调用一次是相同的。

write 操作可能引发 GenericError 类型的异常。这个异常的唯一的数据成员是 reason，其类型是 string。如果因为某种原因，write 操作失败了（比如文件系统空间耗尽），操作就会抛出 GenericError 异常，在 reason 数据成员中会提供对失败原因的解释。

目录提供了用于列出其内容的操作。因为目录既可以包含目录，也可以包含文件，我们利用了 Node 基接口所提供的多态：

```
sequence<Node*> NodeSeq;

interface Directory extends Node {
    nonmutating NodeSeq list();
};
```

NodeSeq 序列包含的是 Node* 类型的元素。因为 Node 既是 Directory、也是 File 的基接口，NodeSeq 序列可以包含任一种类型的代理（显然，为了使用派生接口提供的操作，NodeSeq 的接收者必须把每个元素向下转换成 File 或 Directory；只有 Node 基接口中的 name 操作可以不经向下转换就直接祈用。注意，因为 NodeSeq 的元素的类型是 Node*（不是 Node），我们是在使用传引用语义：list 操作返回的值是一些代理，每个代理都指向在服务器上的一个远地节点。

这些定义已足以构建一个简单的（但也是可以工作的）文件系统。显然，有一些问题仍然未得到解答，比如客户怎样获得根目录的代理。我们将在 XREF 中解答这些问题。

5.4 完整的定义

为了避免污染全局名字空间，我们把我们的各个定义放在一个模块中，从而得到了这样的最终定义：

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };

    exception GenericError {
        string reason;
    };

    sequence<string> Lines;

    interface File extends Node {
        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        nonmutating NodeSeq list();
    };

    interface Filesys {
        nonmutating Directory* getRoot();
    };
};
```

第 6 章

客户端的 Slice-to-C++ 映射

6.1 本章综述

在这一章，我们将介绍客户端的 Slice-to-C++ 映射（第 8 章将介绍客户端的 Slice-to-Java 映射）。客户端的部分 C++ 映射所处理的是，把每种 Slice 数据类型表示成对应的 C++ 类型所遵循的规则；我们将在 6.3 节到 6.10 节涵盖这些规则。映射的另一部分所处理的是，客户怎样祈用操作、传递和接收参数，以及怎样处理异常。这些话题将在 6.11 节到 6.13 节涵盖。Slice 的类既有数据类型的特征，又有接口的特征，将在 6.14 节涵盖。最后，我们将简要地比较 Slice-to-C++ 映射和 CORBA C++ 映射，以此作为这一章的结束。

6.2 引言

客户端 Slice-to-C++ 映射定义的是：怎样把 Slice 数据类型翻译成 C++ 类型，客户怎样祈用操作、传递参数、处理错误。大部分 C++ 映射都很直观。例如，Slice 序列会映射到 STL 向量，所以要在 C++ 中使用 Slice 的序列，本质上你不需要学习什么新东西。

组成 C++ 映射的规则很简单，也很有规律。特别地，C++ 映射不会掉进内存管理的各种潜在陷阱：所有类型都能自己管理自己，并在实例退出作用域时自动进行清理。这意味着，你不会因为下面这样的举动而偶然地

引入内存泄漏：忽略某个操作祈用的返回值，或者忘记释放被调用的操作所分配的内存。

C++ 映射完全是线程安全的。例如，类的引用机制（参见 6.14.5 节）针对并行访问进行了互锁，所以如果有许多线程共享一个类实例，引用计数不会被破坏。显然，在从多个线程访问数据时，你仍然必须进行同步。例如，如果你有两个线程共享一个序列，当一个线程在遍历该序列时，你无法安全地让另一个线程对序列进行插入。但你只需要考虑你自己的数据的并发访问——Ice run time 自身完全是线程安全的，没有哪个 Ice API 调用要求你为了安全地调用它而获取或释放锁。

这一章的大部分内容是参考资料。我们建议你在初次阅读时略读这些资料，然后在需要时再参考特定的部分。但我们认为你至少应该详细阅读从 6.9 节到 6.13 节的内容，因为这些内容讲述了客户应该怎样调用操作、传递参数、处理异常。

在开始之前，你应该注意：要使用 C++ 映射，你只需使用你的应用的 Slice 定义，并且了解 C++ 映射的规则。特别地，为了理解 C++ 映射的用法而查看生成的头文件，很可能会造成你的困惑，因为这些头文件并不一定是拿给人看的，有时，为了处理操作系统和编译器的特质，在这些文件中会含有各种各样的晦涩成分。当然，有时为了确认映射的某个细节，你也可以参考某个头文件，但要想了解应当怎样编写客户端代码，我们建议你还是使用这里给出的资料。

6.3 标识符的映射

Slice 标识符映射到相同的 C++ 标识符。例如，Slice 标识符 `Clock` 会变成 C++ 标识符 `clock`。这条规则有一个例外：如果一个 Slice 标识符与某个 C++ 关键字是一样的，对应的 C++ 标识符就会加上前缀 `_cpp_`。例如，Slice 标识符 `while` 会被映射成 `_cpp_while`¹。

6.4 模块的映射

Slice 模块映射到 C++ 名字空间。映射会保持 Slice 定义的嵌套层次。例如：

1. 如我们在第 60 页的 4.5.3 节所建议的，你应该尽量避免使用这样的标识符。


```
module M1 {  
    module M2 {  
        // ...  
    };  
    // ...  
};  
  
// ...  
  
module M1 {      // Reopen M1  
    // ...  
};
```

这个定义映射到对应的 C++ 定义：

```
namespace M1 {  
    namespace M2 {  
        // ...  
    }  
    // ...  
}  
  
// ...  
  
namespace M1 {  // Reopen M1  
    // ...  
}
```

如果一个 Slice 模块重新打开，对应的 C++ 名字空间也会重新打开。

6.5 Ice 名字空间

为了避免与其他库或应用的定义发生冲突，Ice run time 的所有 API 都嵌在 Ice 名字空间中。Ice 名字空间的有些内容是根据 Slice 定义生成的；其他一些部分提供的是一些专用的定义，没有对应的 Slice 定义。我们将在本书余下的部分逐渐涵盖 Ice 名字空间的内容。

6.6 简单内建类型的映射

如表 6.1 所示， Slice 内建类型映射到一些 C++ 类型。

Table 6.1. 把 Slice 内建类型映射到 C++

Slice	C++
bool	bool
byte	Ice::Byte
short	Ice::Short
int	Ice::Int
long	Ice::Long
float	Ice::Float
double	Ice::Double
string	std::string

Slice 的 bool 和 string 映射到 C++ 的 bool 和 std::string。其他的 Slice 内建类型映射到一些 C++ 类型定义，而不是 C++ 原生类型。这样，Ice run time 就能够针对每种目标架构、提供合适的定义（例如，Ice::Int 在一种架构上可以定义成 long，在另一种上可以定义成 int）。

注意，Ice::Byte 是 unsigned char 的类型定义。这将保证，字节值的范围总在 0 到 255 的范围内。

所有基本类型都肯定是一种不同的 C++ 类型，也就是说，你可以安全地对函数进行重载，这些函数的不同只在于它们使用了表 6.1 中的不同类型。

6.7 用户定义类型的映射

Slice 支持用户定义的类型：枚举、结构、序列，以及词典。

6.7.1 枚举的映射

枚举映射到对应的 C++ 枚举。例如：

```
enum Fruit { Apple, Pear, Orange };
```

不奇怪，生成的 C++ 定义是一样的：

```
enum Fruit { Apple, Pear, Orange };
```

6.7.2 结构的映射

Slice 结构映射到同名的 C++ 结构。对于每一个 Slice 数据成员，C++ 结构都会包含一个 public 数据成员。例如，下面是我们在 4.7.4 节看到过的 Employee 结构：

```
struct Employee {  
    long number;  
    string firstName;  
    string lastName;  
};
```

Slice-to-C++ 编译器为这个结构生成这样的定义：

```
struct Employee {  
    Ice::Long    number;  
    std::string  firstName;  
    std::string  lastName;  
    bool operator==(const Employee&) const;  
    bool operator!=(const Employee&) const;  
    bool operator<(const Employee&) const;  
};
```

对于 Slice 定义中的每一个数据成员，C++ 结构都含有一个对应的 public 数据成员，且名字相同。

注意，这个结构还含有一些比较操作符。这些操作符的行为如下：

- operator==
如果两个结构的所有成员都相等（递归地），它们就是相等的。
- operator!=
如果两个结构有一个或多个成员不相等（递归地），它们就不相等。
- operator<
这个比较操作符把结构的成员当作排序标准：第一个成员被当作是第一个标准，第二个成员被当作是第二个标准，等等。假定我们有两个

Employee 结构 s1 和 s2，所生成的代码就会使用下面的算法来比较它们：

```
bool Employee::operator<(const Employee &rhs) const
{
    if (this == &rhs)    // Short-cut self-comparison
        return false;

    // Compare first members
    //
    if (number < rhs.number)
        return true;
    else if (rhs.number < number)
        return false;

    // First members are equal, compare second members
    //
    if (firstName < rhs.firstName)
        return true;
    else if (rhs.firstName < firstName)
        return false;

    // Second members are equal, compare third members
    //
    if (lastName < rhs.lastName)
        return true;
    else if (rhs.lastName < lastName)
        return false;

    // All members are equal, so return false
    return false;
}
```

之所以要提供比较操作符，是为了使我们能把结构用作 Slice 词典的键类型；Slice 词典会映射到 C++ 的 `std::map`（参见 6.7.4 节）。

注意，复制构造和赋值总是具有深度复制语义。你可以随意用结构或结构成员进行相互赋值，而不必担心内存管理的问题。下面的代码片段阐释了比较和深度复制语义：

```
Employee e1, e2;
e1.firstName = "Bjarne";
e1.lastName = "Stroustrup";
e2 = e1;                                // Deep copy
```

```
assert(e1 == e2);
e2.firstName = "Andrew";           // Deep copy
e2.lastName = "Koenig";            // Deep copy
assert(e2 < e1);
```

因为串会映射到 `std::string`，在这段代码里没有内存管理问题，结构赋值和复制会像我们所期望的那样工作（C++ 编译器生成的缺省的“按成员”复制构造器和赋值操作符合做正确的事情）。

6.7.3 序列的映射

下面是我们在 4.7.3 节见过的 `FruitPlatter` 序列的定义：

```
sequence<Fruit> FruitPlatter;
```

`Slice` 编译器会为 `FruitPlatter` 生成这样的 C++ 定义：

```
typedef std::vector<Fruit> FruitPlatter;
```

你可以看到，序列会简单地映射到 STL 向量。所以，你可以像使用其他任何 STL 向量一样使用序列，例如：

```
// Make a small platter with one Apple and one Orange
//
FruitPlatter p;
p.push_back(Apple);
p.push_back(Orange);
```

正如你可能会期望的，你可以把平常所用的所有 STL 迭代器和算法用于这个向量。

6.7.4 词典的映射

下面是我们在 4.7.4 节见过的 `EmployeeMap` 的定义：

```
dictionary<long, Employee> EmployeeMap;
```

下面的代码是根据这个定义生成的：

```
typedef std::map<Ice::Long, Employee> EmployeeMap;
```

同样不奇怪：`Slice` 词典会简单地映射到 `map`。所以，你可以像使用其他任何 STL `map` 一样使用词典，例如：

```
EmployeeMap em;
Employee e;
```

```
e.number = 42;
```

```
e.firstName = "Stan";
e.lastName = "Lipmann";
em[e.number] = e;

e.number = 77;
e.firstName = "Herb";
e.lastName = "Sutter";
em[e.number] = e;
```

显然，和对待其他任何 STL 容器一样，你可以把平常所用的 STL 迭代器和算法用于这个映射表。

6.8 常量的映射

Slice 常量定义映射到对应的 C++ 常量定义。下面是我们在第 68 页的 4.7.5 节见过的常量定义：

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;
```

```
enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

下面是为这些常量生成的定义：

```
const bool      AppendByDefault = true;
const Ice::Byte  LowerNibble = 15;
const std::string Advice = "Don't Panic!";
const Ice::Short TheAnswer = 42;
const Ice::Double PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

所有的常量都直接在头文件中初始化，所以它们是编译时常量，可以在需要使用编译时常量表达式的地方，比如数据的维数，或是 switch 语句的 case 标签。

6.9 异常的映射

下面是我们在第 82 页的 4.8.5 节看到过的世界时间服务器的部分 Slice 定义：

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

这些异常定义会映射到：

```
class GenericError: public Ice::UserException {
public:
    std::string reason;

    virtual const std::string & ice_name() const;
    virtual Ice::Exception * ice_clone() const;
    virtual void ice_throw() const;
    // Other member functions here...
};

class BadTimeVal: public GenericError {
public:
    virtual const std::string & ice_name() const;
    virtual Ice::Exception * ice_clone() const;
    virtual void ice_throw() const;
    // Other member functions here...
};

class BadZoneName: public GenericError {
public:
    virtual const std::string& ice_name() const;
    virtual Ice::Exception * ice_clone() const;
    virtual void ice_throw() const;
};
```

每个 Slice 异常都会映射到一个同名的 C++ 类。对于每个异常成员，对应的类都会包含一个 **public** 数据成员（显然，因为 `BadTimeVal` 和 `BadZoneName` 没有成员，为这两个异常生成的类也没有成员）。

在生成的类中，Slice 异常的继承结构得到了保持，所以 `BadTimeVal` 和 `BadZoneName` 都是从 `GenericError` 继承的。

每个异常都有三个额外的成员函数：

- `ice_name`

顾名思义，这个成员函数返回异常的名字。例如，如果你调用 `BadZoneName` 异常的 `ice_name` 成员函数，它会（不奇怪）返回 `"BadZoneName"` 串。如果你一般化地捕捉异常，并且想要给出更有意义的诊断信息，`ice_name` 成员函数会很有用，例如：

```
try {
    // ...
} catch (const Ice::GenericError &e) {
    cerr << "Caught an exception: " << e.ice_name() << endl;
}
```

如果有异常被引发，这段代码会打印实际异常的名字（`BadTimeVal` 或 `BadZoneName`），因为异常是通过引用被捕捉到的（为了避免切断）。

- `ice_clone`

这个成员函数允许你多态地克隆异常。例如：

```
try {
    // ...
} catch (const Ice::UserException & e) {
    Ice::UserException * copy = e.clone();
}
```

如果你想要得到某个异常的副本，却又不知道它的确切的运行时类型，`ice_clone` 会很有用。这种特性允许你记住异常，并在后面调用 `ice_throw` 抛出它。

- `ice_throw`

`ice_throw` 允许你在不知道某个异常的确切运行时类型的情况下抛出它。它被实现为：

```
void
GenericError::ice_throw() const
{
    throw *this;
}
```

你可以调用 `ice_throw` 抛出你先前通过 `ice_clone` 克隆的异常。

注意，生成的异常类含有其他一些成员函数，没有在第 151 页上给出。这些类是供 C++ 映射内部使用的，应用代码不应该调用它们。

所有的用户异常最终都继承自 `Ice::UserException`。而 `Ice::UserException` 又继承 `Ice::Exception`（这是 `IceUtil::Exception` 的一个别名）：


```

namespace IceUtil {
    class Exception {
        virtual const std::string & ice_name() const;
        Exception * ice_clone() const;
        void ice_throw() const;
        virtual void ice_print(std::ostream &) const;
    };
    // ...
    std::ostream &operator<<(std::ostream &, const Exception &);
    // ...
}

namespace Ice {
    typedef IceUtil::Exception Exception;

    class UserException: public Exception {
    public:
        virtual const std::string & ice_name() const = 0;
        // ...
    };
}

```

`Ice::Exception` 是异常继承树的根。除了常用的 `ice_name`、`ice_clone`，以及 `ice_throw` 成员函数，它还含有 `ice_print` 成员函数。`ice_print` 打印异常的名字。例如，调用 `BadTimeVal` 异常的 `ice_print` 会打印：

```
BadTimeVal
```

为了让打印更方便，`Ice::Exception` 重载了 `operator<<`，所以你也可以这样编写代码：

```

try {
    // ...
} catch (const Ice::Exception & e) {
    cerr << e << endl;
}

```

这会产生同样的输出，因为 `operator<<` 会在内部调用 `ice_print`。

对于 `Ice` 运行时异常，`ice_print` 还会打印出异常被抛出处的文件名和行号。

6.10 运行时异常的映射

在遇到一些预先定义的错误情况时，Ice run time 会抛出运行时异常。所有运行时异常都直接或间接地派生自 `Ice::LocalException`（而这个异常又派生自 `Ice::Exception`）。`Ice::LocalException` 具有一些常用的成员函数（`ice_name`、`ice_clone`、`ice_throw`，以及（继承自 `Ice::Exception` 的）`ice_print`、`ice_file`，以及 `ice_line`）。

在第 80 页的图 4.4 中给出了用户和运行时异常的继承图。通过在该继承层次的适当点上捕捉异常，你可以根据这些异常所指示的错误范畴来处理异常。例如，`ConnectTimeoutException` 可以作为下面的任何一种异常类型来处理：

- `Ice::Exception`

这是整个继承树的根。捕捉 `Ice::Exception` 会既捕捉用户异常，又捕捉运行时异常。

- `Ice::UserException`

这是所有用户异常的根。捕捉 `Ice::UserException` 会捕捉所有用户异常（但不会捕捉运行时异常）。

- `Ice::LocalException`

这是所有运行时异常的根异常。捕捉 `Ice::LocalException` 会捕捉所有运行时异常（但不会捕捉用户异常）。

- `Ice::TimeoutException`

这既是操作所用超时、也是连接建立超时的基异常。

- `Ice::ConnectTimeoutException`

如果在初次尝试建立与服务器的连接时超时，就会引发这个异常。

你可能很少需要按范畴来捕捉异常；对异常层次的余下部分所提供的细粒度异常处理感兴趣的，主要是 Ice run time 实现。

6.11 接口的映射

Slice 接口的映射是围绕这样一个思想来考虑的：要调用一个远地操作，你要调用一个本地类实例的成员函数，这个实例代表的是远地的对象。这使得映射变得很容易，使用起来也很直观，因为，（除了错误语义，）就各方面而言，发出远地过程调用都与进行本地过程调用没有区别。

6.11.1 代理类和代理句柄

在客户端，接口映射到这样的类：它的成员函数与接口上的操作相对应。考虑下面的简单接口：

```
interface Simple {
    void op();
};
```

Slice 编译器生成下面的定义，供客户使用：

```
namespace IceProxy {

    class Simple : public virtual IceProxy::Ice::Object {
    public:
        void op();
        void op(const Ice::Context &);
        // ...
    };

}

typedef IceInternal::ProxyHandle<IceProxy::Simple> SimplePrx;
```

你可以看到，编译器在 `IceProxy` 名字空间中生成了一个代理类 `Simple`，在全局名字空间中生成了一个代理句柄 `SimplePrx`。一般而言，生成的名字是 `IceProxy::<interface-name>` 和 `::<interface-name>Prx`。如果某个接口嵌套在模块 `M` 中，生成的名字就是 `IceProxy::M::<interface-name>` 和 `::M::<interface-name>Prx`。

在客户的地址空间中，`IceProxy::Simple` 实例是“远地的服务器中的 `Simple` 接口的实例”的“本地大使”，叫作代理类实例。与服务器端对象有关的所有细节，比如其地址、所用协议、对象标识，都封装在该实例中。

注意，`Simple` 继承自 `IceProxy::Ice::Object`。这反映了这样一个事实：所有的 `Ice` 接口都隐含地继承自 `Ice::Object`。对于接口中的每个操作，代理类都有两个重载的、同名的成员函数。就前面的例子而言，我们会发现操作 `op` 映射到了两个成员函数 `op`。

其中一个函数的最后一个参数的类型是 `Ice::Context`。`Ice` run time 用这个参数存储关于请求的递送方式的信息；你通常并不需要为此提供一个值，可以假装这个参数不存在（我们将在第 16 章详细考察 `Ice::Context` 参数。`IceStorm` 使用了这个参数——参见第 26 章）。

客户端应用永远不会直接操纵代理类。事实上，你不允许直接实例化代理类。下面的代码无法编译，因为 `IceProxy::Simple` 含有纯虚的成员函数：

```
IceProxy::Simple s;      // Compile-time error!
```

代理实例总是由 **Ice run time** 替客户实例化，所以客户代码永远都不需要直接实例化代理。当客户从 **run time** 那里接收代理时，它会得到指向该代理的*代理句柄*，其类型是 `<interface-name>Prx`（对于前面的例子就是 `SimplePrx`）。客户通过代理的句柄来访问代理；句柄负责把操作祈用转发给其底层的代理，并且会对代理进行引用计数。这意味着，你不会遇到内存管理问题：代理的释放是自动的，会在指向代理的最后一个句柄消失时（退出作用域时）发生。

因为应用代码总是使用代理句柄，而决不会直接使用代理类，我们通常会用*代理*这个术语来表示代理句柄，也会用它来标识代理类。这反映了这样一个事实：在实际的使用中，代理的“观感”就像是底层的代理类实例。如果区分它们很重要，我们会使用术语*代理类*、*代理类实例*，以及*代理句柄*。

6.11.2 代理句柄上的方法

我们在前面的例子中已经看到，句柄实际上是类型为 `IceInternal::ProxyHandle` 的模板，其参数是代理类。这个模板有缺省构造器、复制构造器，以及赋值构造器：

- 缺省构造器

你可以通过缺省方式构造代理句柄。缺省的构造器创建的是哪里也不指向的代理（也就是说，根本不指向对象）。如果你祈用这样的 `null` 代理上的操作，你会收到 `IceUtil::NullHandleException`：

```
try {
    SimplePrx s;          // Default-constructed proxy
    s->op();               // Call via nil proxy
    assert(0);            // Can't get here
} catch (const IceUtil::NullHandleException &) {
    cout << "As expected, got a NullHandleException" << endl;
}
```

- 复制构造器

复制构造器负责确保你能根据另一个代理句柄构造出一个代理句柄。在内部，这会使代理的引用计数加一；析构器会使引用计数减一，

一旦计数降到零，就释放底层的代理类实例。这样就不会发生内存泄漏了：

```
{
    SimplePrx s1 = ...;           // Enter new scope
    SimplePrx s2(s1);             // Get a proxy from somewhere
    assert(s1 == s2);             // Copy-construct s2
                                // Assertion passes
}                                 // Leave scope; s1, s2, and the
                                // underlying proxy instance
                                // are deallocated
```

注意这个例子中的断言：代理句柄支持比较操作（参见 6.11.3 节）。

- 赋值操作符

你可以随意把一个代理句柄赋给另一个句柄。句柄的实现会保证进行适当的内存管理。自赋值（self-assignment）是安全的，你无需针对这种情况进行保护：

```
SimplePrx s1 = ...;             // Get a proxy from somewhere
SimplePrx s2;                   // s2 is nil
s2 = s1;                        // both point at the same object
s1 = 0;                         // s1 is nil
s2 = 0;                         // s2 is nil
```

宽化赋值（Widening assignments）会隐式地进行。例如，如果我们有两个接口 Base 和 Derived，我们可以隐式地把一个 DerivedPrx 变宽成一个 BasePrx：

```
BasePrx base;
DerivedPrx derived;
base = derived;                 // Fine, no problem
derived = base;                 // Compile-time error
```

隐式的窄化转换（narrowing conversions）会造成编译错误，这也正是 C++ 通常的语义：你总是可以把派生类型赋给基类型，但反过来不行。

- 检查转换（checked cast）

代理句柄提供了一个 checkedCast 方法：

```
namespace IceInternal {
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
        template<class Y>
        static ProxyHandle checkedCast(const ProxyHandle<Y> & r);
    };
}
```

```

    // ...
};
}

```

对于代理来说，检查转换的作用就像是 C++ `dynamic_cast` 相对于指针的作用：它能够让你把基代理赋给派生代理。如果基代理的运行时类型与派生代理的静态类型相容，赋值就能成功，而在赋值之后，基代理代表的对象与派生代理相同。而如果基代理的运行时类型与派生代理的静态类型不相容，派生代理就会被设成 `null`。下面用一个例子来加以说明：

```

BasePrx base = ...;      // Initialize base proxy
DerivedPrx derived;
derived = DerivedPrx::checkedCast(base);
if (derived) {
    // Base has run-time type Derived,
    // use derived...
} else {
    // Base has some other, unrelated type
}

```

表达式 `DerivedPrx::checkedCast(base)` 测试 `base` 指向的是否是 `Derived` 类型的对象。如果是，则转换成功，`derived` 指向的对象会被设成与 `base` 指向的相同。否则，转换就会失败，而 `derived` 被设成 `null` 代理。

注意，`checkedCast` 是一个静态方法，所以，要进行向下转换，你总是使用这样的语法：`<interface-name>Prx::checkedCast`。

还要注意，你可以在布尔上下文中使用代理。例如，如果代理不为 `null`，`if(proxy)` 会返回真（参见 6.11.3 节）。

在你调用 `checkedCast` 时，通常会有一条远地消息发往服务器²。这条消息会实际询问服务器：“这个引用所代表的对象的类型是不是 `Derived`？”服务器的答复会以成功（非 `null`）或失败（`null`）的形式传达给应用代码。发送远地消息是必要的，因为作为一条原则，如果没有服务器的确认，客户无法找出代理实际的运行时类型（例如，服务器可能会用一个派生层次更深的对象实现取代现有的某个代理的对象实现）。这意味着，你必须准备好处理 `checkedCast` 失败的情况。例如，如果服务器没有运行，你就会收到 `ConnectFailedException`

2. 在有些情况下，Ice run time 可以对转换进行优化，避免发送消息。但这种优化只适用于有限的情形，所以你不能假定某个 `checkedCast` 不发送消息。

；如果服务器在运行，但代理所代表的对象已经不存在，你就会收到 `ObjectNotExistException`。

- 不检查转换（`Unchecked cast`）

在有些情况下，你知道某个对象支持一个接口，其派生层次比其代理的静态类型的派生层次更深。对于这样的情况，你可以使用不进行检查的向下转换：

```
namespace IceInternal {
    template<typename T>
    class ProxyHandle : public IceUtil::HandleBase<T> {
    public:
        template<class Y>
        static ProxyHandle uncheckedCast(const ProxyHandle<Y> & r);
        // ...
    };
}
```

`uncheckedCast` 提供了向下转换，并且不会就对象实际的运行时类型去询问服务器，例如：

```
BasePrx base = ...;        // Initialize to point at a Derived
BasePrx derived;
derived = DerivedPrx::uncheckedCast(base);
// Use derived...
```

只有在你确知代理真的支持派生层次更深的类型时，你才应该使用 `uncheckedCast`：顾名思义，`uncheckedCast` 不会进行任何检查；它不会联系服务器中的对象，如果失败，它不会返回 `null`（不检查转换在内部的实现就像是 `C++ static_cast`，不会进行任何一种检查）。如果你使用的代理是通过不正确的 `uncheckedCast` 得到的，其行为将是不确定的。你很可能会收到 `ObjectNotExistException` 或 `OperationNotExistException`，但取决于具体情形，`Ice run time` 也可能报告一个异常，说解编失败，甚至还可能会不声不响地返回垃圾结果。

尽管有危险，`uncheckedCast` 仍然是有用的，因为它不用付出向服务器发消息的代价。而且，在初始化过程中（参见第 7 章），应用常常会收到静态类型是 `Ice::Object` 的代理，但你知道它的具体的运行时类型。在这样的情况下，`uncheckedCast` 可以节省发送远地消息的开销。

6.11.3 对象标识与代理比较

除了 6.11.2 节所讨论的方法，代理句柄还支持比较操作。，proxy handles also support comparison. 下面的操作符得到了明确的支持：

- ==
- !=

这两个操作符允许你比较代理是否相等和不等。为了测试代理是否为 null，你可以与直接量 0 进行比较，例如：

```
if (proxy == 0)
    // It's a nil proxy
else
    // It's a non-nil proxy
```

- <

代理支持 operator<。这使得你能把代理放入 STL 容器，比如映射表或有序列表。

- 布尔比较

代理有一个转换操作符，可以把自己转换成 bool。如果代理不是 null，这个操作符返回真，否则就返回假。所以你可以编写这样的代码：

```
BasePrx base = ...;
if (base)
    // It's a non-nil proxy
else
    // It's a nil proxy
```

注意，在通过重载的操作符 ==、!=，以及 < 进行代理比较时，会使用代理中的*所有*信息。这意味着，对象标识不仅要匹配，代理中的其他资料也必须相同，比如协议和端点信息。换句话说，如果你用 == 和 != 进行比较，测试的是*代理*的同一性，而不是*对象*的同一性。一种常见的错误是编写这样的代码：

```
Ice::ObjectPrx p1 = ...;           // Get a proxy...
Ice::ObjectPrx p2 = ...;           // Get another proxy...

if (p1 != p2) {
    // p1 and p2 denote different objects           // WRONG!
} else {
    // p1 and p2 denote the same object             // Correct
}
```


尽管 `p1` 和 `p2` 是不同的，它们代表的可能是同一个 Ice 对象。例如，如果 `p1` 和 `p2` 包含了相同的对象标识，但各自使用了不同的协议联系目标对象，就可能会发生上述情况。与此类似，协议可能是一样的，但使用的端点不同（因为单个 Ice 对象可以通过若干不同的传输端点联系）。换句话说，如果两个代理用 `==` 比较是相等的，我们就知道这两个代理代表的是同一个对象（因为它们在所有方面都相同）；但如果两个用 `==` 比较不相等，我们什么也不知道：这两个代理所代表的可能是、也可能不是同一个对象。

要比较两个代理的对象同一性，你必须使用 Ice 名字空间里的一个辅助函数：

```
namespace Ice {

    bool proxyIdentityLess(const ObjectPrx &,
                          const ObjectPrx &);
    bool proxyIdentityEqual(const ObjectPrx &,
                           const ObjectPrx &);

}
```

只要嵌在两个代理中的对象标识是一样的，`proxyIdentityEqual` 函数就会返回真，代理中的其他信息会被忽略，比如 `facet` 和传输机制信息。`proxyIdentityLess` 函数建立了代理的总序（total ordering）。其目的主要是为了让你把对象标识比较用于 STL 的有序容器。

`proxyIdentityEqual` 能让你正确地比较代理的对象标识：

```
Ice::ObjectPrx p1 = ...;           // Get a proxy...
Ice::ObjectPrx p2 = ...;           // Get another proxy...

if (!Ice::proxyIdentityEqual(p1, p2) {
    // p1 and p2 denote different objects      // Correct
} else {
    // p1 and p2 denote the same object        // Correct
}
```

6.12 操作的映射

我们在 6.11 节已经看到，对于接口上的每个操作，代理类都有一个对应的同名成员函数。要祈用某个操作，你要通过代理句柄调用它。例如，下面是 5.4 节给出的文件系统的部分定义：

```

module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};

```

Node 接口的代理类如下所示（作了整理，去掉了无关的细节）：

```

namespace IceProxy {
    namespace Filesystem {
        class Node : virtual public IceProxy::Ice::Object {
        public:
            std::string name();
            // ...
        };
        typedef IceInternal::ProxyHandle<Node> NodePrx;
        // ...
    }
    // ...
}

```

name 操作返回的是类型为 string 的值。假定我们有一个代理，指向的是 Node 类型的对象，客户可以这样祈用操作：

```

NodePrx node = ...;           // Initialize proxy
string name = node->name();    // Get name via RPC

```

代理句柄重载了 operator->，把方法调用转发给底层的代理类实例，而后者又把操作祈用发给服务器、等待操作完成，然后解编返回值，并把它返回给调用者。

因为返回值的类型是 string，忽略返回值是安全的。例如，下面的代码没有内存泄漏：

```

NodePrx node = ...;           // Initialize proxy
node->name();                  // Useless, but no leak

```

对于所有被映射的 Slice 类型而言都一样：你可以安全地忽略操作的返回值，不管它的类型是什么——返回值总是通过传值返回。如果你忽略返回值，不会发生内存泄漏，因为返回值的析构器会按照需要释放内存。

6.12.1 普通的、idempotent，以及 nonmutating 操作

你可以给 Slice 操作增加 idempotent 或 nonmutating 限定符。就对应的代理方法的型构而言，idempotent 或 nonmutating 没有任何效果。例如，考虑下面的接口：

```
interface Example {
    string op1();
    idempotent string op2();
    nonmutating string op3();
};
```

这个接口的代理类是：

```
namespace IceProxy {
    class Example : virtual public IceProxy::Ice::Object {
    public:
        std::string op1();
        std::string op2();           // idempotent
        std::string op3();           // nonmutating
        // ...
    };
}
```

因为 `idempotent` 和 `nonmutating` 影响的是调用分派（call dispatch）、而不是接口的某个方面，这三个方法看起来一样，是有道理的。

6.12.2 传递参数

in 参数

C++ 映射的参数传递规则非常简单：参数或者通过值（对于小的值）、或者通过 `const` 引用（对于大于一个机器字的值）传递。在语义上，这两种传递参数的方式是等价的：祈用肯定不会改变参数的值（第 166 页给出了一些警告）。

下面的接口有一些操作，会把各种类型的参数从客户传给服务器：

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

Slice 编译器为这个定义生成这样的代码：

```
struct NumberAndString {
    Ice::Int x;
    std::string str;
    // ...
};

typedef std::vector<std::string> StringSeq;

typedef std::map<Ice::Long, StringSeq> StringTable;

namespace IceProxy {
    class ClientToServer : virtual public IceProxy::Ice::Object {
    public:
        void op1(Ice::Int, Ice::Float, bool, const std::string &);
        void op2(const NumberAndString &,
                  const StringSeq &,
                  const StringTable &);
        void op3(const ClientToServerPrx &);
        // ...
    };
}
```

假定我们有一个代理，指向的是 `ClientToServer` 接口，客户代码可以这样传递参数：

```
ClientToServerPrx p = ...; // Get proxy...

p->op1(42, 3.14, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14;
bool b = true;
string s = "Hello world!";
p->op1(i, f, b, s); // Pass simple variables

NumberAndString ns = { 42, "The Answer" };
StringSeq ss;
ss.push_back("Hello world!");
StringTable st;
st[0] = ss;
p->op2(ns, ss, st); // Pass complex variables

p->op3(p); // Pass proxy
```

你可以把直接量或变量传给各个操作。因为所有的参数都是通过值或 `const` 引用传递的，你不需要考虑内存管理问题。

out 参数

C++ 映射通过引用传递输出参数。下面是在第 163 页上见过的 `Slice` 定义，为了让所有参数作为输出参数传递而作了修改：

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

`Slice` 编译器为这个定义生成这样的代码：

```
namespace IceProxy {
    class ServerToClient : virtual public IceProxy::Ice::Object {
    public:
        void op1(Ice::Int &, Ice::Float &, bool &, std::string &);
        void op2(NumberAndString &, StringSeq &, StringTable &);
        void op3(ServerToClientPrx &);
        // ...
    };
}
```

假定我们有一个代理，指向的是 `ServerToClient` 接口，客户代码可以这样传递参数：

```
ServerToClientPrx p = ...;           // Get proxy...

int i;
float f;
bool b;
string s;
```

```

p->op1(i, f, b, s);
// i, f, b, and s contain updated values now

NumberAndString ns;
StringSeq ss;
StringTable st;

p->op2(ns, ss, st);
// ns, ss, and st contain updated values now

p->op3(p);
// p has changed now!

```

这段代码同样没有什么让人惊讶之处：调用者简单地把变量传给操作；一旦操作完成，服务器就会设置这些变量的值。

最后一个调用值得一看：

```
p->op3(p);      // Weird, but well-defined
```

在这里，`p` 既是用来分派调用的代理，也是这个调用的输出参数，也就是说，服务器将会设置它的值。一般而言，把同一个参数同时用作输入和输出参数是安全的：`Ice run time` 会正确地处理所有加锁和内存管理活动。

下面是一个有点病态的例子：

```

sequence<int> Row;
sequence<Row> Matrix;

interface MatrixArithmetic {
    void multiply(Matrix m1,
                  Matrix m2,
                  out Matrix result);
};

```

假定我们有一个代理，指向的是 `MatrixArithmetic` 接口，客户代码可以这样做：

```

MatrixArithmeticPrx ma = ...;      // Get proxy...
Matrix m1 = ...;                   // Initialize one matrix
Matrix m2 = ...;                   // Initialize second matrix
ma->squareAndCubeRoot(m1, m2, m1); // !!!

```

在不会出现内存混乱或加锁问题的意义上，这段代码在技术上是合法的，但它的行为令人吃惊：因为 `m1` 既被用作输入参数，又被用作输出参数，`m1` 最后的值是不确定的——特别地，如果客户和服务端并置在同一地址空间中，在计算结果的过程中，操作的实现会改写输入矩阵 `m1` 的部分内容，因为结果写往的物理内存位置，也正是输入之一所在的位置。一般

而言，在把同一个变量同时用作输入和输出参数时，你应该多加小心，而且，应该只在被调用的操作肯定能良好工作时这样做。

链式祈用

考虑下面的简单接口，它有两个操作，一个设置值，一个获取值：

```
interface Name {  
    string getName();  
    void setName(string name);  
};
```

假定我们有两个代理 `p1` 和 `p2`，它们指向的是 `Name` 类型的接口。我们进行了这样的连锁祈用：

```
p2->setName(p1->getName());
```

这行代码的效果完全合乎意图：`p1` 返回的值被转交给 `p2`。不存在内存管理或异常安全性问题³。

6.13 异常处理

任何操作祈用都可以抛出运行时异常（参见第 154 页的 6.10 节），而且，如果操作有异常规范，还可以抛出用户异常（参见第 151 页的 6.9 节）。假定我们有这样一个简单的接口：

```
exception Tantrum {  
    string reason;  
};  
  
interface Child {  
    void askToCleanUp() throws Tantrum;  
};
```

`Slice` 异常是作为 C++ 异常抛出的，所以你可以把一个或更多操作祈用放在 `try-catch` 块中：

3. 这值得一提，因为在 CORBA，同样的代码会泄漏内存（就在在许多情况下忽略返回值一样）。

```
ChildPrx p = ...;          // Get proxy...
try {
    p->askToCleanUp(); // Give it a try...
} catch (const Tantrum & t) {
    cout << "The child says: " << t.reason << endl;
}
```

在典型情况下，你只需针对操作所用捕捉你感兴趣的一些异常；其他异常，比如意料之外的运行时错误，通常会由更高层次的异常处理器来处理。例如：

```
void run()
{
    ChildPrx p = ...; // Get proxy...
    try {
        p->askToCleanUp(); // Give it a try...
    } catch (const Tantrum & t) {
        cout << "The child says: " << t.reason << endl;

        p->scold(); // Recover from error...
    }
    p->praise(); // Give positive feedback...
}

int
main(int argc, char * argv[])
{
    try {
        // ...
        run();
        // ...
    } catch (const Ice::Exception & e) {
        cerr << "Unexpected run-time error: " << e << endl;
        return 1;
    }
    return 0;
}
```

出于局部的考虑，这段代码会在调用的地方处理一个具体的异常，并且一般化地处理其他异常（这也是我们在第 3 章的第一个简单应用所采用的策略）。

出于效率上的考虑，你应该总是通过 `const` 引用捕捉异常。这样，编译器就能够生成不调用异常的复制构造器的代码（当然，同时也防止异常被切成基类型）。

异常与 out 参数

当操作抛出异常时，Ice run time 不保证 输出参数的状态：参数的值可能没有变，也可能已经被目标对象中的操作实现改变了。换句话说，对于输出参数，Ice 提供的是弱异常保证 [19]，没有提供强异常保证⁴。

异常与返回值

就返回值而言，如果有异常抛出，C++ 会保证接收操作返回值的变量不会被改写（当然，只有在你没有把同一个变量同时用作 out 参数和接收返回值的参数时，这个保证才会成立（参见第 166 页））。

6.14 类的映射

Slice 类映射到同名的 C++ 类。对于每一个 Slice 数据成员，生成的类都有一个 public 数据成员与之对应，而每一个操作都有一个对应的虚成员函数。考虑下面的类定义：

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
    string format();      // Return time as hh:mm:ss
};
```

Slice 编译器为这个定义生成这样的代码：

```
class TimeOfDay : virtual public Ice::Object {
public:
    Ice::Short hour;
    Ice::Short minute;
    Ice::Short second;

    virtual std::string format() = 0;

    virtual bool ice_isA(const std::string &);
    virtual const std::string & ice_id();
    static const std::string & ice_staticId();

    static const Ice::ObjectFactoryPtr & ice_factory();
};
```

4. 这样做是出于效率上的考虑：提供强异常保证会产生更多并不值得的开销。

```

    // ...
};

typedef IceInternal::Handle<TimeOfDay> TimeOfDayPtr;

```

关于生成的代码，注意以下几点：

1. 生成的 TimeOfDay 类继承自 Ice::Object。这意味着，所有的类都隐式地从 Ice::Object 继承，Ice::Object 是所有类最终的祖先。注意，Ice::Object 和 IceProxy::Ice::Object 并不相同。换句话说，你不能在需要代理的地方传入类，反之亦然（但你可以传入类的代理——参见 6.14.5 节）。
2. 对于每个 Slice 数据成员，生成的类都有一个对应的 public 成员。
3. 对于每个 Slice 操作，生成的类都有一个对应的纯虚成员函数。
4. 生成的类含有另外一些成员函数：ice_isA、ice_id、ice_staticId，以及 ice_factory。
5. 编译器会生成一个类型定义 TimeOfDayPtr。这种类型实现了一种智能指针，把动态分配的类实例包装起来。一般而言，这种类型的名字是 <class-name>Ptr。不要把它与 <class-name>Prx 混淆起来——这种类型也存在，但却是类的代理句柄，而不是智能指针。

在此有相当一些内容要讨论，所以我们将依次考察每一项。

6.14.1 从 Ice::Object 继承

与接口一样，类也隐式地继承自一个共同的基类 Ice::Object。但正如 6.14.1 节所示，类继承自 Ice::Object，而不是 Ice::ObjectPrx（后者是代理的继承层次的根）。所以，在需要代理的地方，你不能传入类（反之亦然），因为类和代理的基类型并不相容。

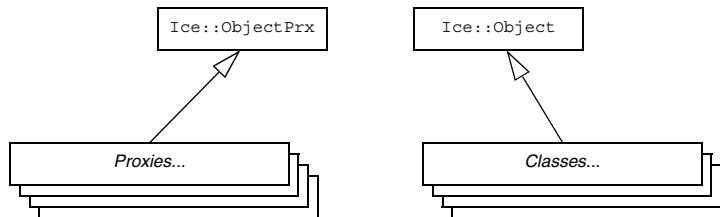


图 6.1. 从 Ice::ObjectPrx 和 Ice::Object 继承

Ice::Object 含有一些成员函数：

```
namespace Ice {
    class Object : virtual public IceUtil::Shared {
    public:
        virtual bool ice_isA(const std::string &,
                             const Current & = Current());
        virtual void ice_ping(const Current & = Current());
        virtual const std::string & ice_id(
                             const Current & = Current());
        static const std::string & ice_staticId();
        virtual Ice::Int ice_hash() const;

        virtual bool operator==(const Object &) const;
        virtual bool operator!=(const Object &) const;
        virtual bool operator<(const Object &) const;
    };
}
```

Ice::Object 的成员函数的行为如下：

- operator==
operator!=
operator<
比较操作符，允许你用类来做 STL 有序容器的元素。
- ice_hash
这个方法返回类的哈希值，这样你可以轻松地把类放入哈希表。
- ice_ping
和接口一样，ice_ping 为类提供了基本的可到达测试。
- ice_id
这个函数返回类的实际的运行时类型 ID。如果你通过指向基实例的智能指针调用 ice_id，返回的类型 ID 是实例实际的类型 ID（派生层次可能会更深）。
- ice_staticId
这个函数返回类的静态类型 ID。

6.14.2 类的数据成员

类的数据成员的映射方式与结构和异常的映射方式完全一样：对于 Slice 定义中的每一个数据成员，生成的类都有一个对应的 public 数据成员。

6.14.3 类的操作

在生成的类中，类上的操作被映射到纯虚的成员函数。这意味着，如果类含有操作（比如我们的 `TimeOfDay` 类的 `format` 操作），你必须从生成的类派生一个类，并在这个类中提供操作的实现。例如：

```
class TimeOfDayI : virtual public TimeOfDay {
public:
    virtual std::string format() {
        std::ostringstream s;
        s << setw(2) << setfill('0') << hour;
        s << setw(2) << setfill('0') << minute;
        s << setw(2) << setfill('0') << second;
        return s.c_str();
    }
};
```

6.14.4 类工厂

创建了这样的类之后，我们有了一个实现，可以实例化 `TimeOfDayI` 类，但我们不能把它作为返回值进行接收，也不能用作某个操作祈用的 `out` 参数。要想知道为什么，考虑下面的简单接口：

```
interface Time {
    TimeOfDay get();
};
```

当客户祈用 `get` 操作时，Ice run time 必须实例化 `TimeOfDay` 类，返回它的一个实例。但 `TimeOfDay` 是一个抽象类，不能实例化。除非我们告诉它，否则 Ice run time 不可能魔法般地知道我们创建了一个 `TimeOfDayI` 类，实现了 `TimeOfDay` 抽象类的 `format` 抽象操作。换句话说，我们必须向 Ice run time 提供一个工厂，这个工厂知道 `TimeOfDay` 抽象类有一个 `TimeOfDayI` 具体实现。Ice::Communicator 接口为我们提供了所需的操作：

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        void removeObjectFactory(string id);
    };
};
```

```

        ObjectFactory findObjectFactory(string id);
        // ...
    };
};

```

要把我们的 `TimeOfDayI` 类的工厂 提供给 Ice run time，我们必须实现 `ObjectFactory` 接口：

```

module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
};

```

Ice run time 会在需要实例化 `TimeOfDay` 类时调用对象工厂的 `create` 操作；并且会在工厂解除登记、或其 `Communicator` 销毁时调用工厂的 `destroy` 操作。我们的对象工厂的一种可能的实现是：

```

class ObjectFactory : public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string &) {
        assert(type == "::TimeOfDay");
        return new TimeOfDayI;
    }
    virtual void destroy() {}
};

```

`create` 方法的参数是要实例化的类的类型 ID（参见 4.12 节）。对于我们的 `TimeOfDay` 类，其类型 ID 是 `"::TimeOfDay"`。我们的 `create` 实现会检查类型 ID：如果是 `"::TimeOfDay"`，就实例化并返回一个 `TimeOfDayI` 对象。如果是其他类型 ID，断言就会失败，因为它不知道怎样实例化其他类型的对象。

假定我们有一个工厂实现，比如我们的 `ObjectFactory`，我们必须把这个工厂的存在告知 Ice run time：

```

Ice::CommunicatorPtr ic = ...;
ic->addObjectFactory(new ObjectFactory, "::TimeOfDay");

```

现在，每当 Ice run time 需要实例化类型 ID 是 `"::TimeOfDay"` 的类时，它就会调用已登记的 `ObjectFactory` 实例的 `create` 方法。

当你调用 `Communicator::removeObjectFactory` 时，或者 `Communicator` 销毁时，Ice run time 就会调用对象工厂的 `destroy` 操作。这样，你就有了清理你的工厂使用的任何资源的机会。当工厂仍然登记在 `Communicator` 上时，不要调用工厂的 `destroy`——如果你这样做了，Ice run time 不知道这件

事情的发生，取决于你的 `destroy` 实现所做的事情，当 Ice run time 下一次使用工厂时，这可能会导致不确定的行为。

注意，你不能针对同一个类型 ID 两次登记工厂：如果你这样做了，Ice run time 就会抛出 `AlreadyRegisteredException`。与此类似，如果你试图移除一个并没有登记过的工厂，Ice run time 就会抛出 `NotRegisteredException`。

最后，要记住，如果一个类只有数据成员，没有操作，你无需为了传送这样的类的实例而创建并登记对象工厂。只有当类有操作时，你才需要定义并登记对象工厂。

6.14.5 用于类的智能指针

C++ 程序员需要反复面对这样的问题：在程序中处理内存的分配和释放。其困难众所周知：异常、函数中的多条返回路径，以及被调用者分配的、必须由调用者释放的内存——在面对这些情况时，要确保程序不泄漏资源可能会极其困难。这在多线程程序中尤其重要：如果你没有严格地跟踪动态内存的所有权，某个线程仍然在使用的内存可能会被其他线程删除，从而带来灾难性的后果。

为了缓解这一问题，Ice 提供了用于类的智能指针。这些智能指针用引用计数来跟踪每个类实例，并在指向一个类实例的最后一个引用消失时，自动删除该实例。Slice 编译器会为每种类类型生成智能指针。对于 Slice 类 `<class-name>`，编译器会生成叫作 `<class-name>Ptr` 的 C++ 智能指针。我们不会在此给出生成的类的全部细节，而是给出了基本的使用模式：每当你在堆上分配类实例时，你可以简单地把 `new` 返回的指针赋给用于类的智能指针。从此以后，内存管理就是自动的，类实例会在它的最后一个智能指针退出作用域时被删除：

```
{                                     // Open scope
    TimeOfDayPtr tod = new TimeOfDayI; // Allocate instance
    tod->second = 0;                   // Initialize
    tod->minute = 0;
    tod->hour = 0;
    // Use instance...

}                                     // No memory leak here!
```

你可以看到，你使用 `operator->`，通过类的智能指针来访问类的成员。当 `tod` 智能指针出作用域时，它的析构器会运行，继而调用底层的类实例的 `delete`，所以不会出现内存泄漏。

智能指针对其底层的类实例进行引用计数：

- 类的构造器将其引用计数设成零。

- 在用动态分配的类实例初始化智能指针时，智能指针会使这个类的引用计数加一。
- 如果你以复制方式构造一个智能指针，这个类的引用计数会加一。
- 把一个智能指针赋给另一个智能指针，会使目标的引用计数加一，并使源的引用计数减一（自赋值是安全的）。
- 智能指针的析构器会使引用计数减一，如果引用计数降到零，就调用它的类实例的 `delete`。

图 6.2 说明了在以缺省方式构造了一个这样的智能指针之后的情况：

```
TimeOfDayPtr tod;
```

这条语句创建一个智能指针，其内部指针为 `null`。



Diagram illustrating the initial state of the smart pointer `tod`. It is represented by a rectangular box containing the text `tod`.

图 6.2. 新初始化的智能指针

如果你构造一个类实例，所创建的实例的引用计数是零；如果你对类指针进行赋值，智能指针就会使类的引用计数加一：

```
tod = new TimeOfDayI;    // Refcount == 1
```

在图 6.3 中说明了之后的情况。

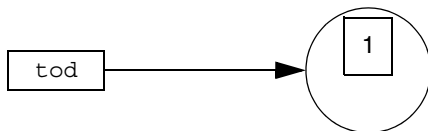


图 6.3. 已初始化的智能指针

如果你以赋值或复制方式构造智能指针，智能指针（不是底层的实例）就会以你指定的方式构造，并且使实例的引用计数增加：

```
TimeOfDayPtr tod2(tod); // Copy-construct tod2
TimeOfDayPtr tod3;
tod3 = tod;              // Assign to tod3
```

图 6.4 说明了执行这些语句后的情况：

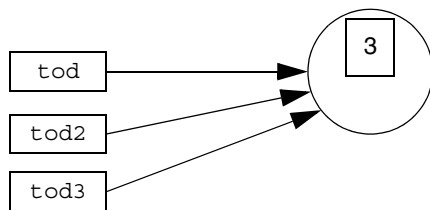


图 6.4. 指向同一类实例的三个智能指针

继续这个例子，我们可以构造第二个类实例，把它赋给原来的智能指针中的 tod2：

```
tod2 = new TimeOfDayI;
```

这会使 tod2 原来代表的类的引用计数减少，并增加赋给 tod2 的类的引用计数，从而产生图 6.5 中所示的情况：

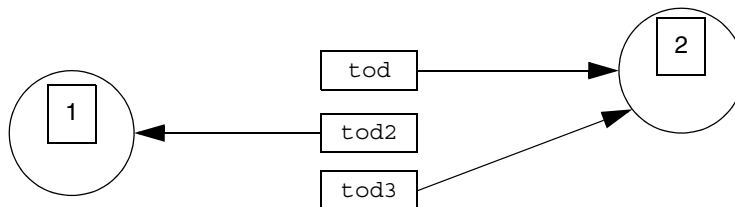


图 6.5. 三个智能指针和两个实例

你可以把零赋给智能指针，从而使它清零：

```
tod = 0;           // Clear handle
```


如你可能会预期的那样，这会减少实例的引用计数。如图 6.6 中所示：

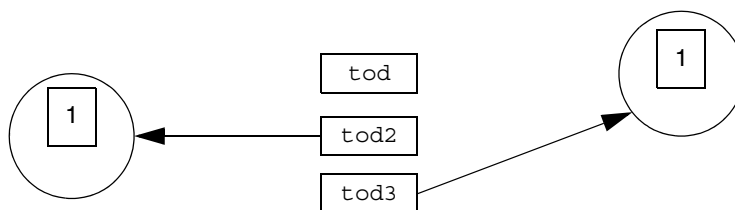


图 6.6. 在使智能指针清零后引用计数减少了

如果智能指针出了作用域、被清零，或是被赋予了新的实例，这个智能指针都会减少其实例的引用计数。如果引用计数降到零，智能指针就会调用 `delete` 释放实例。下面的代码片段把 `tod2` 赋给 `tod3`，从而释放了右边的实例：

```
tod3 = tod2;
```

这产生了图 6.7 中所示的情况。

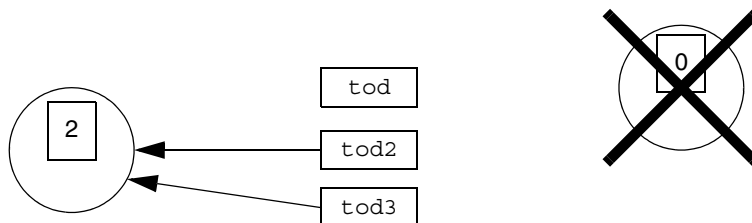


图 6.7. 释放引用计数为零的实例

通过智能指针对类进行深度复制

因为智能指针的赋值只影响智能指针，而不会影响底层的实例，你无法通过对智能指针进行赋值实现类实例的深度复制。要进行深度复制，你必须实现并祈用类的某个成员函数，比如 `clone` 方法。于是这个 `clone` 方法可以返回类实例的深度副本。例如，要创建 `TimeOfDayI` 实例的深度副本，你可以编写这样的代码：

```
class TimeOfDayI;

typedef IceInternal::Handle<TimeOfDayI> TimeOfDayIPtr;

class TimeOfDayI : virtual public TimeOfDay {
```

```

public:
    virtual string format() { /* as before... */ }

    virtual TimeOfDayIPtr clone() {
        TimeOfDayIPtr clone = new TimeOfDayI;
        clone->hour = this->hour;
        clone->minute = this->minute;
        clone->second = this->second;
        return clone;
    }
};

```

注意，这段代码把 `TimeOfDayIPtr` 定义为 `IceUtil::Handle<TimeOfDayI>` 的类型定义。这个类是一个模板，其中含有智能指针的实现。如果你想要把智能指针用于不是由 `Slice` 编译器生成的类，你必须像这个类型定义一样定义一种智能指针类型。

`clone` 成员函数会简单地制作它的各数据成员的副本，并把一个新实例作为智能指针返回。注意，要对类进行深度复制，你必须定义 `clone` 函数。这是因为，为了避免在通过基类智能指针操纵派生类实例时出现的问题，`Slice` 生成的所有类都禁用赋值操作符：试图通过赋值来对类进行深度复制，将会产生编译时错误。

在定义了 `clone` 成员函数之后，我们可以这样对 `TimeOfDayI` 实例进行深度复制：

```

TimeOfDayIPtr tod1 = new TimeOfDayI;
TimeOfDayIPtr tod2 = tod1->clone();    // Deep copy

```

这产生了图 6.8 中所示的情况。

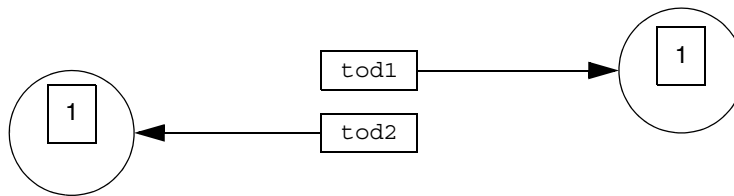


图 6.8. 通过 `clone` 对类实例进行深度复制

注意，有些比较老的编译器仍然没有实现协变的（`covariant`）返回类型。如果你拥有一些类层次，想要在每一层上都有一个 `clone` 成员函数，就会有问题。在这种情况下，唯一的选择是用指向基类的智能指针来充当 `clone` 成员函数的返回类型，同时，对于所有的派生类，都增加能够进行

向下转换的静态成员函数。例如，假定我们有两个 Slice 类 Base 和 Derived，你可以这样实现 BaseI 和 DerivedI 的 clone 方法：

```
class BaseI;
typedef IceUtil::Handle<BaseI> BaseIPtr;

class BaseI : virtual public Base {
public:
    virtual BaseIPtr clone();
    // ...
};

class DerivedI;
typedef IceUtil::Handle<DerivedI> DerivedIPtr;

class DerivedI : virtual public Derived {
public:
    virtual BaseIPtr clone();    // Note: returns BaseIPtr
    // ...
};
```

注意，DerivedI::clone 在这种情况下返回的是 BaseIPtr（因为我们假定编译器不能处理协变的返回类型）。于是，要对 DerivedI 实例进行深度复制，你可以编写：

```
DerivedIPtr p1 = new DerivedI;
DerivedIPtr p2 = DerivedIPtr::dynamicCast(p1->clone());
```

dynamicCast 静态成员函数是 IceUtil::Handle 模板的一部分，而且，顾名思义，它实现智能指针的向下转换。和 C++ dynamic_cast 一样，如果参数不是预期的类型，dynamicCast 的返回值就是 null。

Null 智能指针

在 null 智能指针里，指向其底层实例的 C++ 指针为 null。这意味着，如果你试图解除一个 null 智能指针的引用，你将引发

IceUtil::NullHandleException。

```
TimeOfDayPtr tod;                                // Construct null handle

bool gotNullHandleException = false;

try {
    tod->minute = 0;                                // Dereference null handle
} catch (const IceUtil::NullHandleException &) {
```

```

    gotNullHandleException = true;
}

assert(gotNullHandleException); // Must have seen exception

```

在栈上分配类实例

尽管类实例是通过引用计数进行管理的，你仍然可以在栈上分配类实例，这样做不会造成问题，例如：

```

{
    TimeOfDayI t;           // Stack-allocated class instance
    // ...
}
// Close scope, t is destroyed

```

当控制线程离开围绕变量 `t` 的块时，编译器生成的代码会调用 `t` 的析构器，析构器会和平常一样，清理类所使用的任何资源。这里没有代码会调用 `delete`，因为并没有智能指针牵涉进来（记住，当引用计数降到零时，调用 `delete` 的是*智能指针*的析构器，而不是类的析构器。

尽管如此，在栈上分配类实例实际上并没有用处，因为所有的 Ice APIs 都期望参数是智能指针，而不是类实例。这意味着，要用在栈上分配的类实例做任何事情，你必须为这个实例初始化一个智能指针。但是，这样做并不可行，因为它肯定会带来冲突：

```

{
    TimeOfDayI t;           // Stack-allocated class instance
    TimeOfDayPtr todp;      // Handle for a TimeOfDay instance

    todp = &t;              // Legal, but dangerous
    // ...
}
// Leave scope, looming crash!

```

这段代码有很大的错误，因为，当 `todp` 出作用域时，它把类的引用计数减到零，并针对在栈上分配的类实例调用 `delete`。

下面的代码试图修正这个问题，但也注定会失败：

```

{
    TimeOfDayI t;           // Stack-allocated class instance
    TimeOfDayPtr todp;      // Handle for a TimeOfDay instance

    todp = &t;              // Legal, but dangerous
    // ...
    todp = 0;               // Crash imminent!
}

```

这段代码试图通过显式地使智能指针清零来绕开问题。但这样做同样会使智能指针把类的引用计数减到零，所以这段代码和前面的例子一样，最后同样会针对在栈上分配的实例调用 `delete`。

所有这些论述的要点是：*永远不要在栈上分配类实例*。C++ 映射假定所有的类实例都是在堆上分配的，再多的编程花招也不能改变这一点。

智能指针和循环

你需要注意一件事情：引用计数不能处理循环依赖（cyclic dependencies）。例如，考虑下面的自引用的类：

```
class Node {  
    int val;  
    Node next;  
};
```

直观地看，这个类实现的是节点的链表。只要在节点列表中没有循环，就没有任何问题，而我们智能指针也将正确地释放类实例。但是，如果我们引用循环，我们就会遇到问题：

```
{ // Open scope...  
  
    NodePtr n1 = new Node; // N1 refcount == 1  
    NodePtr n2 = new Node; // N2 refcount == 1  
    n1->next = n2;         // N1 refcount == 2  
    n2->next = n1;         // N2 refcount == 2  
  
} // Destructors run:      // N2 refcount == 1,  
                           // N1 refcount == 1, memory leak!
```

`n1` 和 `n2` 所指向的节点没有名字，但为了行文方便，让我们假定 `n1` 的节点叫作 `N1`，`n2` 的节点叫作 `N2`。当我们分配 `N1` 实例，并把它赋给 `n1` 时时，智能指针 `n1` 会把 `N1` 的引用计数加到 1。与此类似，在分配了节点、并把它赋给 `n2` 之后，`N2` 的引用计数也是 1。接下来的两条语句使 `n1` 和 `n2` 的 `next` 互相指向，从而在它们之间设置了一个循环依赖。这样做会使 `N1` 和 `N2` 的引用计数都变成 2。当它们退出作用域时，`n2` 的析构器会先被调用，把 `N2` 的引用计数减为 1，然后 `n1` 的析构器也会被调用，把 `N1` 的引用计数减为 1。实际的结果就是，两个引用计数都没有降到零，所以 `N1` 和 `N2` 都泄漏了。

类实例的垃圾收集

前面的例子说明了使用引用计数来进行内存释放的一个一般问题：如果在一个图（graph）中的任何地方存在循环依赖（可能是通过许多中间节点），循环中的所有节点都会泄漏。

为了避免由于这样的循环而泄漏内存，Ice for C++ 包含了一个垃圾收集器。收集器会找出这样的类实例、并删除它们：这些实例是一个或多个循环的一部分，但在程序中已经不能到达它们：

- 在缺省情况下，只要你销毁通信器，垃圾就会被收集。这意味着，当你的程序退出时，不会发生内存泄漏（当然，前提是你按照 10.3 节的描述，正确销毁你的通信器。）
- 你可以调用 `Ice::collectGarbage`，显式调用垃圾收集器。例如，前面的例子所造成的泄漏可以这样来避免：

```
{                                     // Open scope...

    NodePtr n1 = new Node; // N1 refcount == 1
    NodePtr n2 = new Node; // N2 refcount == 1
    n1->next = n2;         // N1 refcount == 2
    n2->next = n1;         // N2 refcount == 2

} // Destructors run:                // N2 refcount == 1,
                                     // N1 refcount == 1

Ice::collectGarbage();               // Deletes N1 and N2
```

对 `Ice::collectGarbage` 的调用会删除不再能到达的实例 N1 和 N2（以及其他先前积累起来的、不再能到达的任何实例）。

- 通过显式调用来删除泄漏的内存可能会更方便，因为对收集器的调用会污染代码。你可以把 `Ice.GC.Interval` 属性设成非零的值⁵，要求 Ice run time 运行一个垃圾收集线程，周期性地清理泄漏的内存。例如，把 `Ice.GC.Interval` 设成 5，收集器线程就会每五秒运行一次垃圾收集器。把 `Ice.Trace.GC` 设成非零的值，你可以跟踪收集器的执行（Appendix C）。

注意，*只有在你的程序真的创建了循环的类图的情况下*，垃圾收集器才有用。在没有创建这样的循环的程序中运行垃圾收集器，是没有意义的事情（因此，在缺省情况下，收集器线程是禁用的，只有在你显式地把 `Ice.GC.Interval` 设成了非零的值之后，收集器线程才会运行）。

5. 关于属性的设置，参见第 14 章。

智能指针比较

和代理句柄一样（参见第 160 页的 6.11.3 节），类句柄也支持比较操作符 ==、!=，以及 <。所以你可以在 STL 有序容器中使用类句柄。注意，智能指针并不会用对象标识进行比较，因为类实例没有标识。相反，这些操作只是比较它们所指向的类的内存地址。这意味着，只有当两个智能指针指向的是同一个物理类实例时，operator== 才会返回真：

```
// Create a class instance and initialize
//
TimeOfDayIPtr p1 = new TimeOfDayI;
p1->hour = 23;
p1->minute = 10;
p1->second = 18;

// Create another class instance with
// the same member values
//
TimeOfDayIPtr p2 = new TimeOfDayI;
p2->hour = 23;
p2->minute = 10;
p2->second = 18;

assert(p1 != p2);           // The two do not compare equal

TimeOfDayIPtr p3 = p1;      // Point at first class again

assert(p1 == p3);           // Now they compare equal
```

6.15 slice2cpp 命令行选项

除了在 4.18 节描述的标准选项以外，Slice-to-C++ 编译编译器 **slice2cpp** 还提供了以下命令行选项：

- **--header-ext EXT**

把生成的头文件的文件扩展名从缺省的 h 变成 **EXT** 所指定的扩展名。

- **--source-ext EXT**

把生成的源文件的文件扩展名从缺省的 cpp 变成 **EXT** 所指定的扩展名。

- **--impl**

生成示范性的实现文件，这个选项不会覆盖已有文件。

- **--depend**

这个选项以适用于 **make** 实用程序的形式，把文件依赖关系打印到 stdout。

6.16 与 CORBA C++ 映射比较

要比较 Slice 和 CORBA 的 C++ 映射有点困难，因为它们是如此不同。任何 CORBA C++ 开发者都知道，CORBA C++ 映射大而复杂，而且，在有些地方还很晦涩难解。例如，开发者需要担起大量容易出错的内存管理责任，而什么由开发者释放，什么由 ORB 释放，相关的规则并不一致。

总体而言，Ice C++ 映射要容易使用得多，并与 STL 集成在一起；而且，由于生成的代码数量较少，也要高效得多。

第 7 章

开发 C++ 文件系统客户

7.1 本章综述

在这一章，我们将给出一个 C++ 客户的源码，用于访问我们在第 5 章开发的文件系统（Java 版本的客户见第 9 章）。

7.2 C++ 客户

我们现在所知道的客户端 C++ 映射，已经足以让我们开发一个完整的客户，用于访问我们的远地文件系统。为方便参考起见，这里再把文件系统的 Slice 定义给出一次：

```
module Filesystem {  
    interface Node {  
        nonmutating string name();  
    };  
  
    exception GenericError {  
        string reason;  
    };  
  
    sequence<string> Lines;  
  
    interface File extends Node {
```

```

        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
};

sequence<Node*> NodeSeq;

interface Directory extends Node {
    nonmutating NodeSeq list();
};

interface Filesys {
    nonmutating Directory* getRoot();
};
};

```

为了演练文件系统，客户会从根目录开始，递归地列出文件系统的内容。对于文件系统中的一个节点，客户都会显示节点名，以及该节点是文件还是目录。如果节点是文件，客户就获取文件的内容，并打印它们。

下面是客户代码的主体：

```

#include <Ice/Ice.h>
#include <Filesystem.h>
#include <iostream>
#include <iterator>

using namespace std;
using namespace Filesystem;

static void
listRecursive(const DirectoryPrx & dir, int depth = 0)
{
    // ...
}

int
main(int argc, char * argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        // Create a communicator
        //
        ic = Ice::initialize(argc, argv);

        // Create a proxy for the root directory
        //

```

```

Ice::ObjectPrx base
    = ic->stringToProxy("RootDir:default -p 10000");
if (!base)
    throw "Could not create proxy";

// Down-cast the proxy to a Directory proxy
//
DirectoryPrx rootDir = DirectoryPrx::checkedCast(base);
if (!rootDir)
    throw "Invalid proxy";

// Recursively list the contents of the root directory
//
cout << "Contents of root directory:" << endl;
listRecursive(rootDir);
} catch (const Ice::Exception & ex) {
    cerr << ex << endl;
    status = 1;
} catch (const char * msg) {
    cerr << msg << endl;
    status = 1;
}

// Clean up
//
if (ic)
    ic->destroy();

return status;
}

```

1. 代码包括了一些头文件:

1. Ice/Ice.h

客户和服务端都总是会包括这个文件。它提供了要访问 Ice run time 所必需的定义。

2. Filesystem.h

这是 Slice 编译器根据 Filesystem.ice 中的 Slice 定义生成的头文件。

3. iostream

客户使用了 iostream 库来产生其输出。

4. iterator

listRecursive 的实现使用了一个 STL 迭代器。

2. 这段代码增加了针对 `std` 和 `Filesystem` 名字空间的 `using` 声明。
3. `main` 的代码结构遵循了我们在第 3 章看到过的结构。在初始化 `run time` 之后，客户创建一个代理，指向文件系统的根目录。在这个例子中，我们假定服务器运行在本地主机上，并且使用缺省协议（`TCP/IP`）在 10000 端口处侦听。根目录的对象标识叫作 `RootDir`。
4. 客户把代理向下转换成 `DirectoryPrx`，并把这个代理传给 `listRecursive`，由它打印出文件系统的内容。

大多数工作都是在 `listRecursive` 中完成的：

```
// Recursively print the contents of directory "dir" in
// tree fashion. For files, show the contents of each file.
// The "depth" parameter is the current nesting level
// (for indentation).

static void
listRecursive(const DirectoryPrx & dir, int depth = 0)
{
    string indent(++depth, '\t');

    NodeSeq contents = dir->list();

    for (NodeSeq::const_iterator i = contents.begin();
         i != contents.end();
         ++i) {
        DirectoryPrx dir = DirectoryPrx::checkedCast(*i);
        FilePrx file = FilePrx::uncheckedCast(*i);
        cout << indent << (*i)->name()
              << (dir ? " (directory)：" : " (file)：") << endl;
        if (dir) {
            listRecursive(dir, depth);
        } else {
            Lines text = file->read();
            for (Lines::const_iterator j = text.begin();
                 j != text.end();
                 ++j) {
                cout << indent << "\t" << *j << endl;
            }
        }
    }
}
```

这个函数收到的参数是一个代理，指向要列出的目录；另外还有一个缩进层次参数（缩进层次随着每一次递归调用增加，这样，打印每个节点名

时的缩进层次就会与该节点的树深度对应)。listRecursive 调用目录的list 操作，并且遍历所返回的节点序列：

1. 代码调用 checkedCast，把 Node 代理窄化成 Directory 代理；并且调用 uncheckedCast，把 Node 代理窄化成 File 代理。在这两个转换中只有、而且肯定会有一个成功，所以不需要两次调用 checkedCast：如果节点是一个 Directory，代理就使用 checkedCast 返回的 DirectoryPrx；如果 checkedCast 失败，我们知道了这个节点是一个 File，因此，要获得 FilePrx，使用 uncheckedCast 就足够了。

一般而言，如果你知道向下转换到特定类型能成功，就最好使用 uncheckedCast，而不是 checkedCast，因为 uncheckedCast 不需要进行任何网络通信。

2. 代码打印文件或目录的名字，然后，取决于成功的转换是哪一个，在名字的后面打印 "(directory)" 或 "(file)"。
3. 代码检查节点的类型：
 - 如果是目录，代码就会递归，同时增加缩进层次。
 - 如果是文件，代表就调用文件的 read 操作，取回文件内容，然后遍历返回的内容行序列，打印每一行。

假定我们有一个很小的文件系统，由两个文件和一个目录组成：

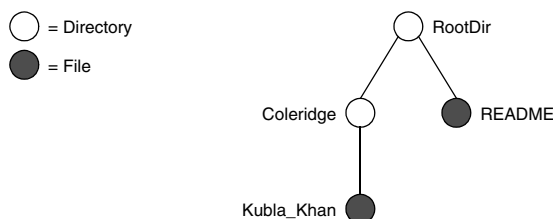


图 7.1. 一个小文件系统

这个文件的客户产生的输出是：

```

Contents of root directory:
  README (file):
    This file system contains a collection of poetry.
  Coleridge (directory):
    Kubla_Khan (file):
      In Xanadu did Kubla Khan
  
```

```
A stately pleasure-dome decree:  
Where Alph, the sacred river, ran  
Through caverns measureless to man  
Down to a sunless sea.
```

注意，迄今为止，我们的客户（以及服务器）并不很成熟：

- 协议和地址信息是硬写在代码中的。
- 客户进行了一些并非绝对必要的远地过程调用；只要稍稍重新设计一下 `Slice` 定义，就可以避免许多这样的调用。

我们将在 XREF 和 XREF 中看到怎样消除这些缺点。

7.3 总结

这一章介绍了一个非常简单的客户，用于访问我们在第 5 章开发的文件系统服务器。你可以看到，你很难把这些 C++ 代码和一个普通的 C++ 程序区分开来。这是 Ice 的最大的优点之一：访问远地对象就和访问普通的本地 C++ 对象一样容易。这样，你就可以把精力放在该放的地方，也就是说，集中精力开发你的应用逻辑，而不用去和晦涩的网络 APIs 作斗争。我们将在第 11 章看到，对服务器端来说同样也是如此，也就是说，你可以轻松而高效地开发分布式应用。

第 8 章

客户端的 Slice-to-Java 映射

8.1 本章综述

在这一章，我们将介绍基本的 Slice-to-Java 映射（基本的 Slice-to-C++ 映射见第 6 章）。客户端的部分 Java 映射所涉及的是，把每种 Slice 数据类型表示成对应的 Java 类型所遵循的规则；我们将在 8.3 节到 8.10 节涵盖这些规则。映射的另一部分所处理的是，客户怎样祈用操作、传递和接收参数，以及怎样处理异常。这些话题将在 8.11 节到 8.13 节涵盖。Slice 的类既有数据类型的特征，又有接口的特征，将在 8.14 节涵盖。最后，我们将简要地比较 Slice-to-Java 映射和 CORBA Java 映射，以此作为这一章的结束。

8.2 引言

客户端 Slice-to-Java 映射定义的是：怎样把 Slice 数据类型翻译成 Java 类型，客户怎样祈用操作、传递参数、处理错误。大部分 Java 映射都很直观。例如，Slice 序列映射到 Java 数组，所以要在 Java 中使用 Slice 序列，本质上你不需要学习什么新东西。

Ice run time 的 Java API 完全是线程安全的。显然，在从多个线程访问数据时，你仍然必须进行同步。例如，如果你有两个线程共享一个序列，当一个线程在遍历该序列时，你无法安全地让另一个线程对序列进行插入。

但你只需要考虑你自己的数据的并发访问——Ice run time 自身完全是线程安全的，没有哪个 Ice API 调用要求你为了安全地调用它而获取或释放锁。

这一章的大部分内容是参考资料。我们建议你在初次阅读时略读这些资料，然后在需要时再参考特定的部分。但我们认为你至少应该详细阅读从 8.9 节到 8.13 节的内容，因为这些内容讲述了客户应该怎样调用操作、传递参数、处理异常。

在开始之前，你应该注意：要使用 Java 映射，你只需使用你的应用的 Slice 定义，并且了解 Java 映射的规则。特别地，为了理解 Java 映射的用法而查看生成的头文件，很可能会造成你的困惑，因为这些头文件并不一定是拿给人看的，有时，为了处理操作系统和编译器的特质，在这些文件中会含有各种各样的晦涩成分。当然，有时为了确认映射的某个细节，你也可以参考某个头文件，但要想了解应当怎样编写客户端代码，我们建议你还是使用这里给出的资料。

8.3 标识符的映射

Slice 标识符映射到相同的 Java 标识符。例如，Slice 标识符 Clock 会变成 Java 标识符 clock。这条规则有一个例外：如果一个 Slice 标识符与某个 Java 关键字是一样的，对应的 Java 标识符的前面就会加上一个下划线。例如，Slice 标识符 while 会被映射成 `_while`¹

8.4 模块的映射

Slice 模块映射到 Java package，名字保持不变。位于全局作用域处的定义成为无名的 Java 全局 package 的一部分。映射会保持 Slice 定义的嵌套层次。例如：

```
// Definitions at global scope here...

module M1 {
    // Definitions for M1 here...
    module M2 {
        // Definitions for M2 here...
    };
};
```

1. 如我们在第 60 页的 4.5.3 节所建议的，你应该尽量避免使用这样的标识符。

```
};  
  
// ...  
  
module M1 {      // Reopen M1  
    // More definitions for M1 here...  
};
```

这个定义映射到对应的 Java 定义：

```
// Definitions at global scope here...  
  
package M1;  
// Definitions for M1 here...  
  
package M1.M2;  
// Definitions for M2 here...  
  
package M1;  
// Definitions for M1 here...
```

注意，这些定义会出现在适当的源文件中；为位于全局作用域处的定义所生成的源文件会放在顶层的输出目录中，为模块 M1 中的定义所生成的源文件会放在顶层目录之下的 M1 目录中，而为模块 M2 中的定义所生成的源文件会放在顶层目录之下的 M1/M2 目录中。在使用 **slice2java** 时，你可以用 **--output-dir** 选项设置顶层输出目录（参见 4.18 节）。

8.5 Ice Package

为了避免与其他库或应用的定义发生冲突，Ice run time 的所有 API 都放在 Ice package 中。Ice package 的有些内容是根据 Slice 定义生成的；其他一些部分提供的是一些专用的定义，没有对应的 Slice 定义。我们将在本书余下的部分逐渐涵盖 Ice package 的内容。

8.6 简单内建类型的映射

如表 8.1 所示， Slice 内建类型映射到 Java 的一些类型。

Table 8.1. 把 Slice 内建类型映射到 Java

Slice	Java
bool	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
string	String

8.7 用户定义类型的映射

Slice 支持用户定义的类型：枚举、结构、序列，以及词典。

8.7.1 枚举的映射

Java 没有枚举类型，所以 Slice 枚举是用 Java 类模拟的：Slice 枚举的名字会变成 Java 类的名字；对于每一个枚举符，类都有对应的 final 成员，一个的名字和枚举符一样，另一个的名字是枚举符的名字、前面加上一个下划线。例如：

```
enum Fruit { Apple, Pear, Orange };
```

下面是生成的 Java 类：


```
public final class Fruit {
    public static final int _Apple = 0;
    public static final int _Pear = 1;
    public static final int _Orange = 2;

    public static final Fruit Apple = new Fruit(_Apple);
    public static final Fruit Pear = new Fruit(_Pear);
    public static final Fruit Orange = new Fruit(_Orange);

    public int value() {
        // ...
    }

    public static Fruit
    convert(int val) {
        // ...
    }

    // ...
}
```

注意，生成的类含有另外一些成员，我们没有给出这些成员。这些成员供 **Ice run time** 内部使用，你不能在你的应用代码中使用它们（因为在各版本之间，这些成员可能会发生变化）。

有了上面的定义，我们可以这样使用枚举值：

```
Fruit favoriteFruit = Fruit.Apple;
Fruit otherFavoriteFruit = Fruit.Orange;

if (favoriteFruit == Fruit.Apple) // Compare with constant
    // ...

if (f1 == f2) // Compare two enums
    // ...

switch (f2.value()) { // Switch on enum
case Fruit._Apple:
    // ...
    break;
case Fruit._Pear
    // ...
    break;
case Fruit._Orange
    // ...
    break;
}
```

你可以看到，生成的这个类使得我们能够很自然地使用枚举值。有前置下划线的 `int` 成员是常量，是每个枚举符的编码；`Fruit` 成员是预先初始化的枚举符，你可以把它们用于初始化和比较操作。

`value` 和 `convert` 方法充当的是访问器和修改器，所以你可以把枚举变量的值当作一个 `int` 来读写。如果你在使用 `convert` 方法，你必须确保所传递的值在枚举的范围之内；如果你没有这样做，就会产生断言失败：

```
Fruit favoriteFruit = Fruit.convert(4); // Assertion failure!
```

8.7.2 结构的映射

`Slice` 结构映射到同名同名的 `Java` 类。对于每一个 `Slice` 数据成员，`Java` 类都会包含一个 `public` 数据成员。例如，下面是我们在 4.7.4 节看到过的 `Employee` 结构：

```
struct Employee {  
    long number;  
    string firstName;  
    string lastName;  
};
```

`Slice-to-Java` 编译器为这个结构生成这样的定义：

```
public final class Employee implements java.lang.Cloneable {  
    public long number;  
    public String firstName;  
    public String lastName;  
  
    public boolean equals(java.lang.Object rhs) {  
        // ...  
    }  
    public int hashCode() {  
        // ...  
    }  
  
    public java.lang.Object clone()  
        throws java.lang.CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

对于 `Slice` 定义中的每一个数据成员，`Java` 类都含有一个对应的 `public` 数据成员，且名字相同。`equals` 成员函数比较两个结构是否相等。注意，生成的类还提供了惯常的 `hashCode` 和 `clone` 方法（`clone` 的缺省行为是制作浅副本）。

8.7.3 序列的映射

Slice 序列映射到 Java 数组。这意味着，Slice-to-Java 编译器不会为 Slice 序列生成单独命名的类型。例如：

```
sequence<Fruit> FruitPlatter;
```

这个定义会简单地对应到 Java 类型 `Fruit[]`。很自然，因为 Slice 序列映射到 Java 数组，你可以利用 Java 提供的所有数组功能，比如初始化、赋值、克隆，以及 `length` 成员。例如：

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
assert(platter.length == 2);
```

其他映射方式

你可以通过元数据指令改变序列的缺省映射方式，例如：

```
["java: type: java.util.LinkedList"] sequence<Fruit> FruitPlatter;
```

这条指令告诉编译器，要在生成的代码中使用 `java.util.LinkedList` 类型，而不是使用缺省的数组。除了使用 Java 提供的某种 `collection`，你还可以把序列映射到你自定义的实现。就 Ice 整编代码而言，它所期望的是，不管你使用的是何种类型，该类型必须实现 `java.util.List` 接口。

如果你给序列定义增加元数据指令，你是在改变该序列类型的缺省映射。通过其他元数据指令，你还可以重新定义特定的结构、类，或异常成员的映射方式。例如：

```
enum Fruit { Apple, Pear, Orange };

sequence<Fruit> Breakfast;
["java: type: java.util.LinkedList"] sequence<Fruit> Dessert;

struct Meal1 {
    Breakfast    b;
    Dessert      d;
};

struct Meal2 {
    ["java: type: java.util.LinkedList"] Breakfast b;
    ["java: type: java.util.Vector"]      Dessert   d;
};
```

根据这个定义，`Breakfast` 被映射到 Java 数组，`Dessert` 被映射到 `java.util.LinkedList`，而 `Meal1` 的两个数据成员也会进行相应的映

射。对于 `Meal 2`，缺省映射被改换了，所以 `Meal 2::b` 被映射到 `java.util.LinkedList`，而 `Meal 2::d` 被映射到 `java.util.Vector`。

注意，你只能以这种方式改换结构、类，或异常成员的序列映射（你不能改换序列元素、词典值、参数，或返回值的映射）。

8.7.4 词典的映射

下面是我们在 4.7.4 节见过的 `EmployeeMap` 的定义：

```
dictionary<long, Employee> EmployeeMap;
```

和序列一样，Java 映射不会为这个定义创建单独命名的类型。相反，所有词典的类型都是 `java.util.Map`，所以我们可以像使用其他任何 Java 映射表一样使用雇员结构映射表（当然，因为 `java.util.Map` 是一个抽象类，我们必须使用具体的类，比如 `java.util.HashMap`，来充当实际的映射表）。例如：

```
java.util.Map em = new java.util.HashMap();

Employee e = new Employee();
e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(new Long(e.number), e);
```

8.8 常量的映射

下面是我们在第 68 页的 4.7.5 节见过的常量定义：

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

下面是为这些常量生成的定义：

```
public interface AppendByDefault {
    boolean value = true;
}

public interface LowerNibble {
    byte value = 15;
}

public interface Advice {
    String value = "Don't Panic!";
}

public interface TheAnswer {
    short value = 42;
}

public interface PI {
    double value = 3.1416;
}

public interface FavoriteFruit {
    Fruit value = Fruit.Pear;
}
```

你可以看到，每个 Slice 常量都被映射到一个同名的 Java 接口。接口含有一个名叫 value 的成员，这个成员持有常量的值。

8.9 异常的映射

下面是我们在第 82 页的 4.8.5 节看到过的世界时间服务器的部分 Slice 定义：

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

这些异常定义会映射到：

```
public class GenericError extends Ice.UserException {
    public String reason;

    public String ice_name() {
```

```
        return "GenericError";
    }
}

public class BadTimeVal extends GenericError {
    public String ice_name() {
        return "BadTimeVal";
    }
}

public class BadZoneName extends GenericError {
    public String ice_name() {
        return "BadZoneName";
    }
}
```

每个 Slice 异常 都会映射到一个同名的 Java 类。对于每个异常成员，对应的类都会包含一个 **public** 数据成员（显然，因为 `BadTimeVal` 和 `BadZoneName` 没有成员，为这两个异常生成的类也没有成员）。

在生成的类中，Slice 异常的继承结构得到了保持，所以 `BadTimeVal` 和 `BadZoneName` 都是从 `GenericError` 继承的。

每个异常还定义了 `ice_name` 成员函数，这个函数返回的是异常的名字。

所有用户异常都派生自基类 `Ice.UserException`。所以针对 `Ice.UserException` 安装一个处理器，你可以一般化地捕捉所有用户异常。`Ice.UserException` 又派生自 `java.lang.Exception`。

注意，生成的异常类含有其他一些成员函数，在这里没有给出。这些类是供 Java 映射内部使用的，应用代码不应该调用它们。

8.10 运行时异常的映射

在遇到一些预先定义的错误情况时，Ice run time 会抛出运行时异常。所有运行时异常都直接或间接地派生自 `Ice::LocalException`（而这个异常又派生自 `java.lang.RuntimeException`）。

在第 80 页的图 4.4 中给出了用户和运行时异常的继承图。通过在该继承层次的适当点上捕捉异常，你可以根据这些异常所指示的错误范畴来处理异常：

- `Ice.LocalException`

这是运行时异常继承树的根。

- `Ice.UserException`
这是用户异常继承树的根。
- `Ice.TimeoutException`
这既是操作所用超时、也是连接建立超时的基异常。
- `Ice.ConnectionTimeoutException`

如果在初次尝试建立与服务器的连接时超时，就会引发这个异常。

你可能很少需要按范畴来捕捉异常；对异常层次的余下部分所提供的细粒度异常处理感兴趣的，主要是 Ice run time 实现。但有一个特别的异常你可能会感兴趣：`Ice.ObjectNotExistException`。如果客户端所用已不存在的 Ice 对象上的操作，就会引发这个异常。换句话说，客户端持有的是一个悬空的引用，它所指向的对象在过去可能存在，但已经被永久地销毁了。

8.11 接口的映射

Slice 接口映射到客户端的代理。一个代理就是一个 Java 接口，其操作对应于 Slice 接口中所定义的操作。

编译器为每个 Slice 接口生成的源文件相当少。一般而言，对于接口 `<interface-name>`，编译器会创建以下源文件：

- `<interface-name>.java`
这个源文件声明 `<interface-name>` Java 接口。
- `<interface-name>Holder.java`
这个源文件为接口定义 holder 类型（参见第 215 页）。
- `<interface-name>Prx.java`
这个源文件定义 `<interface-name>Prx` 接口（参见第 208 页）。
- `<interface-name>PrxHelper.java`
这个源文件定义为接口的代理定义助手类型（参见第 211 页）。
- `<interface-name>PrxHolder.java`
这个源文件为接口的代理定义 holder 类型（参见第 215 页）。
- `<interface-name>Operations.java`
这个源文件定义一个接口，其操作与 Slice 接口的操作相对应。

这些文件包含的是与客户端有关的代码。编译器还生成了一个服务器端专用的文件，再加上三个其他文件：

- `_<interface-name>Disp.java`
这个文件包含的是服务器端骨架类的定义。
- `_<interface-name>Del.java`
- `_<interface-name>DelD.java`
- `_<interface-name>DelM.java`

这些文件包含的是供 Java 映射内部使用的代码；它们包含的功能与应用程序员无关。

8.11.1 代理接口

在客户端，Slice 映射到 Java 接口，后者的成员函数与前者的操作相对应。考虑下面的简单接口：

```
interface Simple {
    void op();
};
```

Slice 编译器生成以下定义，供客户使用：

```
public interface SimplePrx extends Ice.ObjectPrx {
    public void op();
    public void op(java.util.Map __context);
}
```

你可以看到，编译器生成了一个代理接口 `SimplePrx`。一般而言，生成的代理接口的名字是 `<interface-name>Prx`。如果接口是嵌在模块 `M` 中的，生成的类就是 `package M` 的一部分，所以受到完全限定的名字是 `M.<interface-name>Prx`。

在客户的地址空间中，`SimplePrx` 的实例是“远端的服务器中的 `Simple` 接口的实例”的“本地大使”，叫作代理实例。与服务器端对象有关的所有细节，比如其地址、所用协议、对象标识，都封装在该实例中。

注意，`SimplePrx` 继承自 `Ice.ObjectPrx`。这反映了这样一个事实：所有的 `Ice` 接口都隐式地继承自 `Ice::Object`。

对于接口中的每个操作，代理类都有一个同名的成员函数。就前面的例子而言，我们会发现操作 `op` 已经被映射到成员函数 `op`。还要注意，`op` 是重载的：`op` 的第二个版本有一个参数 `__context`，类型是 `java.util.Map`。`Ice` run time 用这个参数存储关于请求的递送方式的信息；你通常并不需要使用它（我们将在第 16 章详细考察 `__context` 参数。`IceStorm` 使用了这个参数——参见第 26 章）。

因为所有的 `<interface-name>Prx` 类型都是接口，你不能实例化这种类型的对象。相反，代理实例总是由 Ice run time 替客户实例化，所以客户代码永远都不需要直接实例化代理。Ice run time 给出的代理引用的类型总是 `<interface-name>Prx`；这种接口的具体实现是 Ice run time 的一部分，与应用代码无关。

8.11.2 Ice.ObjectPrx 接口

所有 Ice 对象的最终祖先都是 `Object`，所以所有代理都继承自 `Ice.ObjectPrx`。`ObjectPrx` 提供了一些方法：

```
package Ice;

public interface ObjectPrx {
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    int ice_hash();
    boolean ice_isA(String __id);
    String ice_id();
    void ice_ping();
    // ...
}
```

这些方法的行为如下：

- `equals`

这个操作比较两个代理是否相等。注意，这个操作会比较代理的所有方面，比如代理的通信端点。这意味着，一般而言，如果两个代理不相等，那并不说明它们代表的是不同的对象。例如，如果两个代理代表的是同一个 Ice 对象，但所用传输端点不同，那么即使它们代表的是同一个对象，`equals` 也会返回 `false`。

- `ice_getIdentity`

这个对象返回的是代理所代表的对象的标识。Ice 对象标识的 `Slice` 类型是：

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
};
```

要想知道两个代理代表的是否是同一个对象，首先获取每个对象的标识，然后对其进行比较：

```
Ice.ObjectPrx o1 = ...;
Ice.ObjectPrx o2 = ...;
Ice.Identity i1 = o1.ice_getidentity();
Ice.Identity i2 = o2.ice_getidentity();

if (i1.equals(i2))
    // o1 and o2 denote the same object
else
    // o1 and o2 denote different objects
```

- `ice_hash`

这个方法返回代理的哈希键。

- `ice_isA`

这个方法确定代理所代表的对象是否支持特定接口。`ice_isA` 的参数是一个类型 ID（参见 4.12 节）。例如，要想知道一个 `ObjectPrx` 类型的代理代表的是否是 `Printer` 对象，我们可以编写：

```
Ice.ObjectPrx o = ...;
if (o != null && o.ice_isA("::Printer"))
    // o denotes a Printer object
else
    // o denotes some other type of object
```

注意，在调用 `ice_isA` 方法之前，我们先测试了代理是否为 `null`。这样，如果代理为 `null`，就不会引发 `NullPointerException` 了。

- `ice_id`

这个方法返回代理所代表的对象的类型 ID。注意，返回的类型是实际对象的类型，其派生层次可能比代理的静态类型更深。例如，如果我们有一个 `BasePrx` 类型的代理，其静态的类型 ID 是 `::Base`，`ice_id` 的返回值可能会是 `::Base`，也可能是派生层次更深的对象的类型 ID，比如 `::Derived`。

- ice_ping

这个方法为对象提供了基本的可到达测试。如果对象可以从物理上联系到（也就是说，对象存在，它的服务器在运行，并且可到达），调用就会正常完成；否则，它就会抛出异常，说明对象为何不能到达，比如 `ObjectNotExistException` or `ConnectTimeoutException`。

注意，`ObjectPrx` 还有另外一些方法，在此没有给出。这些方法提供了不同的调用分派方式，同时还可以访问对象的 `facets`；我们将在第 16 章和 XREF 中讨论这些方法。

8.11.3 代理助手

对于每个 `Slice` 接口，除了代理接口以外，`Slice-to-Java` 编译器还会创建一个助手类：对于 `Simple` 接口，所生成的助手类的名字是 `SimplePrxHelper`。助手类含有两个有意思的方法：

```
public final class SimplePrxHelper
    extends Ice.ObjectPrxHelper implements SimplePrx {
    public static SimplePrx checkedCast(Ice.ObjectPrx b) {
        // ...
    }

    public static SimplePrx uncheckedCast(Ice.ObjectPrx b) {
        // ...
    }

    // ...
}
```

`checkedCast` 和 `uncheckedCast` 方法实现的都是向下转换：如果传入的代理是 `Simple` 类型的对象的代理，或者是 `Simple` 的派生类型的对象的代理，这两个方法就会返回一个非 `null` 引用，指向的是一个 `SimplePrx` 类型的代理；而如果传入的代理代表的是其他类型的对象（或者传入的代理为 `null`），返回的就是 `null` 引用。

对于任何类型的代理，你都可以 `checkedCast` 来确定其对应的对象是否支持指定的类型，例如：

```
Ice.ObjectPrx obj = ...;           // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

注意，`checkedCast` 会联系服务器。这是必要的，因为只有服务器情况中的代理实现确切地知道某个对象的类型。所以，`checkedCast` 可能会抛出 `ConnectTimeoutException` 或 `ObjectNotExistException`（这也解释了为何需要助手类：Ice run time 必须联系服务器，所以我们不能使用 Java 的向下转换）。

与此相反，`uncheckedCast` 不会联系服务器，而是会无条件地返回具有所请求的类型的代理。但是，如果你要使用 `uncheckedCast`，你必须确定这个代理真的支持你想要转换到的类型；而如果你弄错了，你很可能在祈用代理上的操作时，引发运行时异常。对于这样的类型失配，最后可能会引发 `OperationNotExistException`，但也有可能引发其他异常，比如整编异常。而且，如果对象碰巧有一个同名的操作，但参数类型不同，则有可能根本不产生异常，你最后就会把祈用发送给类型错误的对象；这个对象可能会做出非常糟糕的事情。为了说明这种情况，考虑下面的两个接口：

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

假定你期望收到的是 `Process` 对象的代理，并且要用 `uncheckedCast` 对这个代理进行向下转换：

```
Ice.ObjectPrx obj = ...; // Get proxy...
ProcessPrx process
    = ProcessPrxHelper::uncheckedCast(obj); // No worries...
process.launch(40, 60); // Oops...
```

如果你收到的代理实际上代表 `Rocket` 对象，Ice run time 无法检测到这个错误：因为 `int` 和 `float` 的尺寸相同，而 Ice 协议在线路上没有标记数据的类型，于是 `Rocket::launch` 的实现就会误把传入的整数当作浮点数。

公平地说，这个例子是人为制造的。要让这样的错误在运行时不被注意到，两个对象都必须有一个同名的操作，而且，传给操作的运行时参数整编后的总尺寸，必须与服务器端的解编代码所期望的字节数相吻合。在实践中，这相当罕见，错误的 `uncheckedCast` 通常会导致运行时异常。

关于向下转换的最后一个警告：你必须使用 `checkedCast` 或 `uncheckedCast` 对代理进行向下转换。如果你使用了 Java 的强制转换，其行为是不确定的。

8.12 操作的映射

我们在 8.11 节已经看到，对于接口上的每个操作，代理类都有一个对应的同名成员函数。要祈用某个操作，你要通过代理调用它。例如，下面是 5.4 节给出的文件系统的部分定义：

```
module Filesystem {
    interface Node {
        nonmutating string name();
    };
    // ...
};
```

`name` 操作返回的是类型为 `string` 的值。假定我们有一个代理，指向的是 `Node` 类型的对象，客户可以这样祈用操作：

```
NodePrx node = ...;           // Initialize proxy
String name = node.name();     // Get name via RPC
```

这是典型的返回值接收模式：对于复杂的类型，返回值通过引用返回，对于简单的类型（比如 `int` 或 `double`）。

8.12.1 普通的、`idempotent`，以及 `nonmutating` 操作

你可以给 `Slice` 操作增加 `idempotent` 或 `nonmutating` 限定符。就对应的代理方法的型构而言，`idempotent` 或 `nonmutating` 没有任何效果。例如，考虑下面的接口：

```
interface Example {
    string op1();
    idempotent string op2();
    nonmutating string op3();
};
```

这个接口的代理类是：

```
public interface ExamplePrx extends Ice.ObjectPrx {
    public String op1();
    public String op2();
    public String op3();
}
```

因为 `idempotent` 和 `nonmutating` 影响的是调用分派（`call dispatch`）、而不是接口的某个方面，这三个方法看起来一样，是有道理的。

8.12.2 传递参数

in 参数

Java 映射的参数传递规则非常简单：参数或者通过值传递（对于小的值），或者通过引用传递（对于复杂类型和 `String` 类型）。在语义上，这两种传递参数的方式是等价的：祈用保证不会改变参数的值（XREF 给出了一些警告）。

下面的接口有一些操作，会把各种类型的参数从客户传给服务器：

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

Slice 编译器为这个定义生成这样的代码：

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(int i, float f, boolean b, String s);
    public void op2(NumberAndString ns,
        String[] ss,
        java.util.Map st);
    public void op3(ClientToServerPrx proxy);
}
```

假定我们有一个代理，指向的是 `ClientToServer` 接口，客户代码可以这样传递参数：

```
ClientToServerPrx p = ...; // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s); // Pass simple variables
```

```
NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.HashMap st = new java.util.HashMap();
st.put(new Long(0), ns);
p.op2(ns, ss, st);                                // Pass complex variables

p.op3(p);                                          // Pass proxy
```

out 参数

Java 没有 “pass-by-reference” 语义：参数总是通过值传递的。要让一个函数能修改其参数，我们必须把引用（通过值）传给对象（we must pass a reference (by value) to an object）；然后，被调用的函数可以通过传入的引用修改对象的内容。

为了让被调用的函数能修改参数，Java 映射提供了所谓的 *holder* 类。例如，对于每种内建的 Slice 类型，比如 `int` 和 `string`，Ice package 都含有一个对应的 *holder* 类。下面是 *holder* 类 `Ice.IntHolder` 和 `Ice.StringHolder` 的定义：

```
package Ice;

public final class IntHolder {
    public IntHolder() {}
    public IntHolder(int value)
        this.value = value;
    }
    public int value;
}

public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

holder 类有一个 `public` 的 `value` 成员，用于存储参数的值；通过对该成员赋值，被调用的函数可以对值进行修改。这个类还有一个缺省构造器，以及一个以初始值为参数的构造器。

对于用户定义的类型，比如结构，Slice-to-Java 编译器会生成对应的 holder 类型。例如，下面是为我们在第 214 页定义的 NumberAndString 结构生成的 holder 类型：

```
public final class NumberAndStringHolder {
    public NumberAndStringHolder() {}

    public NumberAndStringHolder(NumberAndString value) {
        this.value = value;
    }

    public NumberAndString value;
}
```

这看起来和内建类型的 holder 类完全一样：我们得到了一个缺省构造器，一个以初始值为参数的构造器，以及 public 的 value 成员。

注意，编译器会为你定义的每一种 Slice 类型生成 holder 类。例如，对于序列，比如我们在第 203 页上看到过的 FruitPlatter 序列，编译器不会生成特殊的 Java FruitPlatter 类型，因为序列会映射到 Java 数组。但编译器 *会* 生成 FruitPlatterHolder 类，所以我们可以把 FruitPlatter 数组用作 out 参数。

要把 out 参数传给操作，我们只需传递 holder 类的实例，并在调用完成时检查每个 out 参数的 value 成员。下面是我们在第 214 页看到过的 Slice 定义，但所有的参数都是作为 out 参数传递的：

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

Slice 编译器为这个定义生成了以下代码：


```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(Ice.IntHolder i, Ice.FloatHolder f,
        Ice.BooleanHolder b, Ice.StringHolder s);
    public void op2(NumberAndStringHolder ns,
        StringSeqHolder ss, StringTableHolder st);
    public void op3(ClientToServerPrxHolder proxy);
}
```

假定我们有一个代理，指向的是 `ServerToClient` 接口，客户代码可以这样传递参数：

```
ClientToServerPrx p = ...;           // Get proxy...

Ice.IntHolder ih = new Ice.IntHolder();
Ice.FloatHolder fh = new Ice.FloatHolder();
Ice.BooleanHolder bh = new Ice.BooleanHolder();
Ice.StringHolder sh = new Ice.StringHolder();
p.op1(ih, fh, bh, sh);

NumberAndStringHolder nsh = new NumberAndString();
StringSeqHolder ssh = new StringSeqHolder();
StringTableHolder sth = new StringTableHolder();
p.op2(nsh, ssh, sth);

ServerToClientPrxHolder stcph = new ServerToClientPrxHolder();
p.op3(stcph);

System.out.println(ih.value); // Show one of the values
```

这段代码同样没有什么让人惊讶之处：一旦操作祈用完成，值就会出现各个 `holder` 实例中，你可以通过每个实例的 `value` 成员访问这些值。

参数类型失配

Java 映射中的参数是静态地类型安全的，只有一个例外：词典映射到 `java.util.Map`，而 `java.util.Map` 是从 `java.lang.Object` 到 `java.lang.Object` 的映射表。也就是说，如果你在客户和服务端之间传递映射表，你必须保证，你插入映射表的值对（pairs of values）的类型是正确的。例如：

```
dictionary<string, long> AgeTable;

interface Ages {
    void put(AgeTable ages);
};
```

假定我们有一个代理，指向的是 Ages 对象，客户可能像是这样：

```
AgesPrx ages = ...;      // Get proxy...

java.util.HashMap ageTable = new java.util.HashMap();
String name = "Michi Henning";
Long age = new Long(42);
ageTable.put(age, name);      // Oops...
ages.put(ageTable);          // ClassCastException!
```

这段代码的问题是，ageTable.put 的参数的次序反了。当代理实现试图整编数据时，它会注意到类型的失配，并抛出 ClassCastException。

Null 参数

有些 Slice 类型自然地就有“空”或“不存在”语义。比如，代理、序列、词典，以及串都可以为 null：

- 对于代理，Java null 引用用于代表 null 代理。null 是一个专用值，表示代理哪里也不指向（不指向哪个对象）。
- 序列和词典不能为 null，但可以是空的。为了让这些类型的使用更容易，只要你把 Java null 引用用作参数、或序列或词典类型的值，Ice run time 都会自动把空的序列或词典发给接收者。
- Java 串可以为 null，但 Slice 串不能（因为 Slice 串不支持 null 串的概念）。只要你把 Java null 串用作参数或返回值，Ice run time 都会自动把空串发给接收者。

作为一种方便的特性，这种行为很有用，特别是对于很深地嵌套的数据类型，其成员中的序列、词典，或串会自动作为空值到达接收端。例如，这样一来，在把一个大序列发送出去之前，你无需显式地初始化每一个串元素，也不会产生 NullPointerExceptions。注意，以这种方式使用 null 参数并没有为序列、词典，或串创建 null 语义。就对象模型而言，它们的 null 语义并不存在（存在的只是空的序列、词典，以及串）。例如，不管你是通过 null 引用、还是通过空串发送串，对接收者都没有区别；不管是哪种方式，接收者看到的都是空串。

8.13 异常处理

任何操作祈用都可以抛出运行时异常（参见第 206 页的 8.10 节），而且，如果操作有异常规范，还可以抛出用户异常（参见第 205 页的 8.9 节）。假定我们有这样一个简单的接口：

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice 异常是作为 Java 异常抛出的，所以你可以把一个或更多操作祈用放在 try-catch 块中：

```
ChildPrx child = ...;    // Get child proxy...

try {
    child.askToCleanUp();
} catch (Tantrum t) {
    System.out.write("The child says: ");
    System.out.println(t.reason);
}
```

在典型情况下，你只需针对操作祈用捕捉你感兴趣的一些异常；其他异常，比如意料之外的运行时错误，通常会由更高层次的异常处理器来处理。例如：

```
public class Client {
    static void run() {
        ChildPrx child = ...;    // Get child proxy...
        try {
            child.askToCleanUp();
        } catch (Tantrum t) {
            System.out.print("The child says: ");
            System.out.println(t.reason);
            p.scold();                // Recover from error...
        }
        p.praise();                // Give positive feedback...
    }

    public static void
    main(String[] args)
    {
```

```

    try {
        // ...
        run();
        // ...
    } catch (Ice.LocalException e) {
        e.printStackTrace();
    } catch (Ice.UserException e) {
        System.err.println(e.getMessage());
    }
}
}

```

出于局部的考虑，这段代码会在调用的地方处理一个具体的异常，并且一般化地处理其他异常（这也是我们在第 3 章的第一个简单应用所采用的策略）。

异常与 out 参数

当操作抛出异常时，Ice run time 不保证 out 参数的状态：参数的值可能没有变，也可能已经被目标对象中的操作实现改变了。换句话说，对于输出参数，Ice 提供的是弱异常保证 [19]，没有提供强异常保证²。

8.14 类的映射

Slice 类映射到同名的 Java 类。对于每一个 Slice 数据成员，生成的类都有一个对应的 public 数据成员（就像结构和异常的情况），而每一个操作都有一个对应的成员函数。考虑下面的类定义：

```

class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
    string format();      // Return time as hh:mm:ss
};

```

Slice 编译器为这个定义生成这样的代码：

2. 这样做是出于效率上的考虑：提供强异常保证会产生更多并不值得的开销。

```
public interface TimeOfDayOperations {
    String format(Ice.Current current);
}

public abstract class TimeOfDay extends Ice.ObjectImpl
    implements TimeOfDayOperations {
    public short hour;
    public short minute;
    public short second;
    // ...
}
```

关于生成的代码，注意以下几点：

- 编译器生成了一个叫作 `TimeOfDayOperations` 的接口。对于类中的每一个 `Slice` 操作，在这个 `<interface-name>Operations` 接口中都有一个对应的方法。
- 编译器生成一个与 `Slice` 类同名的抽象基类（在这个例子中即 `TimeOfDay`）。对于 `Slice` 类中的每一个数据成员，在这个类中都有一个对应的 `public` 数据成员。

如果类只有数据成员，没有操作，编译器生成一个非抽象类。

在这个例子中，`TimeOfDay` 是抽象类，这是有意义的：`Slice-to-Java` 编译器无法知道 `format` 操作的实现应该做什么，所以唯一的选择就是让 `TimeOfDay` 成为抽象类。要创建一个能工作的 `TimeOfDay` 实现，你必须从 `TimeOfDay` 派生一个类，提供缺少的 `format` 实现，例如：

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df
            = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
            df.format(minute) + ":" +
            df.format(second));
    }
}
```

8.14.1 类工厂

创建了这样的类之后，我们有了一个实现，可以实例化 `TimeOfDayI` 类，但我们不能把它作为返回值进行接收，也不能用作某个操作所需的输出参数。要想知道为什么，考虑下面的简单接口：

```
interface Time {
    TimeOfDay get();
};
```

当客户祈用 `get` 操作时，Ice run time 必须实例化 `TimeOfDay` 类，返回它的一个实例。但 `TimeOfDay` 是一个抽象类，不能实例化。除非我们告诉它，否则 Ice run time 不可能魔法般地知道我们创建了一个 `TimeOfDayI` 类，实现了 `TimeOfDay` 抽象类的 `format` 抽象操作。换句话说，我们必须给 Ice run time 提供一个工厂，这个工厂知道 `TimeOfDay` 抽象类有一个 `TimeOfDayI` 具体实现。Ice::Communicator 接口为我们提供了所需的操作：

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        void removeObjectFactory(string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

要把我们的 `TimeOfDayI` 类的工厂 提供给 Ice run time，我们必须实现 `ObjectFactory` 接口：

```
class ObjectFactory
    extends Ice.LocalObjectImpl
    implements Ice.ObjectFactory {
public Ice.Object create(String type) {
    if(type.equals("::TimeOfDay")) {
        return new TimeOfDayI();
    }
    assert(false);
    return null;
}

public void destroy() {
    // Nothing to do
}
}
```

Ice run time 会在需要实例化 `TimeOfDay` 类时调用对象工厂的 `create` 操作；并且会在工厂解除登记、或其 `Communicator` 销毁时调用工厂的 `destroy` 操作。

`create` 方法的参数是要实例化的类的类型 ID（参见 4.12 节）。对于我们的 `TimeOfDay` 类，其类型 ID 是 `"::TimeOfDay"`。我们的 `create` 实现会检查类型 ID：如果是 `"::TimeOfDay"`，就实例化并返回一个 `TimeOfDayI` 对象。如果是其他类型 ID，断言就会失败，因为它不知道怎样实例化其他类型的对象。

假定我们有一个工厂实现，比如我们的 `ObjectFactory`，我们必须把这个工厂的存在告知 Ice run time：

```
Ice.Communicator ic = ...;
ic.addObjectFactory(new ObjectFactory(), "::TimeOfDay");
```

现在，每当 Ice run time 需要实例化类型 ID 是 `"::TimeOfDay"` 的类时，它就会调用已登记的 `ObjectFactory` 实例的 `create` 方法。

当你调用 `Communicator::removeObjectFactory` 时，或者 `Communicator` 销毁时，Ice run time 就会调用对象工厂的 `destroy` 操作。这样，你就有了清理你的工厂使用的任何资源的机会。当工厂仍然登记在 `Communicator` 上时，不要调用工厂的 `destroy`——如果你这样做了，Ice run time 不知道这件事情的发生，取决于你的 `destroy` 实现所做的事情，当 Ice run time 下一次使用工厂时，这可能会导致不确定的行为。

注意，你不能针对同一个类型 ID 两次登记工厂：如果你这样做了，Ice run time 就会抛出 `AlreadyRegisteredException`。与此类似，如果你试图移除一个并没有登记过的工厂，Ice run time 就会抛出 `NotRegisteredException`。

最后，要记住，如果一个类只有数据成员，没有操作，你无需为了传送这样的类的实例而创建并登记对象工厂。只有当类有操作时，你才需要定义并登记对象工厂。

8.14.2 从 LocalObject 继承

注意，前面的例子中的工厂实现扩展了 `Ice.LocalObjectImpl`。对象工厂是一个本地对象，从 `Ice.LocalObject` 继承了一些操作，比如 `ice_hash`（参见第 210 页）。`Ice.LocalObjectImpl` 类提供了这些操作的实现，所以你无需自己实现它们。

一般而言，所有的本地接口都是从 `Ice.LocalObject` 继承的，而所有的本地接口（比如对象工厂或 `servant` 定位器（参见 16.6 节））的实现，都必须从 `Ice.LocalObjectImpl` 继承。

8.15 Package

在缺省情况下，Slice 定义所在的作用域会决定它所映射到的 Java 成分所在的 package。如果 Slice 类型是在任何模块之外定义的，它所映射到的 Java 类型会放在无名的全局 package 中。与此类似，在某个模块层次中定义的 Slice 类型所映射到的类会放在等价的 Java package 中（更多关于模块映射的消息，参见 8.4 节）

但有时，应用需要更多地控制生成的 Java 类的“packaging”。例如，某个公司可能有自己的软件开发指导方针，要求把所有 Java 类都放在指定的 package 中。要满足这一要求，一种做法是修改 Slice 模块层次，让生成的代码缺省地使用指定的 package。在下面的例子中，我们把原来的 `Workflow::Document` 定义放在模块 `com::acme` 中，以使编译器把创建出的类放在 `com.acme` package 中：

```
module com {
  module acme {
    module Workflow {
      class Document {
        // ...
      };
    };
  };
};
```

这种“权宜之计”有两个问题：

1. 它把实现语言（implementation language）的需求混合进了应用的接口规范中。
2. 使用其他语言（比如 C++）的开发者也会受影响。

Slice-to-Java 编译器提供了一种更好的办法来控制生成的代码的 package：使用全局元数据（4.17 节）。可以把上面的例子转换成这样：

```
[[ "java: package: com.acme" ]]
module Workflow {
  class Document {
    // ...
  };
};
```

全局的元数据指令 `java:package:com.acme` 指示编译器，让它根据这个 Slice 文件生成的类都放进 Java package `com.acme` 中。其实际效果是一样的：为 `Document` 生成的类放进了 package `com.acme.Workflow` 中。但是，通过减少对接口规范的影响，我们消除了第一种解决方案的两

个缺点：Slice-to-Java 编译器会识别 `package` 元数据指令，相应地修改其动作，而其他语言映射的编译器会简单地忽略它。

8.16 slice2java 命令行选项

除了 4.18 节描述的标准选项，Slice-to-Java 编译器还提供了以下命令行选项：

- **--tie**
生成 `tie` 类（参见 12.7 节）。
- **--impl**
生成示范性的实现文件。这个选项不会覆盖已有文件。
- **--impl-tie**
生成使用 `tie` 的示范性实现文件（参见 12.7 节）。这个选项不会覆盖已有文件。

第 9 章

开发 Java 文件系统客户

9.1 本章综述

在这一章，我们将给出一个 Java 客户的源码，用于访问我们在第 5 章开发的文件系统（C++ 版本的客户见第 7 章）。

9.2 Java 客户

我们现在所知道的客户端 Java 映射，已经足以让我们开发一个完整的客户，用于访问我们的远地文件系统。为方便参考起见，这里再把文件系统的 Slice 定义给出一次：

```
module Filesystem {  
    interface Node {  
        nonmutating string name();  
    };  
  
    exception GenericError {  
        string reason;  
    };  
  
    sequence<string> Lines;  
  
    interface File extends Node {
```

```

        nonmutating Lines read();
        idempotent void write(Lines text) throws GenericError;
    };

    sequence<Node*> NodeSeq;

    interface Directory extends Node {
        nonmutating NodeSeq list();
    };

    interface Filesys {
        nonmutating Directory* getRoot();
    };
};

```

为了演练文件系统，客户会从根目录开始，递归地列出文件系统的内容。对于文件系统中的一个节点，客户都会显示节点名，以及该节点是文件还是目录。如果节点是文件，客户就获取文件的内容，并打印它们。

下面是客户代码的主体：

```

import Filesystem.*;

public class Client {

    // Recursively print the contents of directory "dir" in
    // tree fashion.  For files, show the contents of each file.
    // The "depth" parameter is the current nesting level
    // (for indentation).

    static void
    listRecursive(DirectoryPrx dir, int depth)
    {
        char[] indentCh = new char[++depth];
        java.util.Arrays.fill(indentCh, '\t');
        String indent = new String(indentCh);

        NodePrx[] contents = dir.list();

        for (int i = 0; i < contents.length; ++i) {
            DirectoryPrx subdir
                = DirectoryPrxHelper.checkedCast(contents[i]);
            FilePrx file
                = FilePrxHelper.uncheckedCast(contents[i]);
            System.out.println(indent + contents[i].name() +
                               (subdir != null ? " (directory):" : " (file):"));
            if (subdir != null) {

```

```
        listRecursive(subdir, depth);
    } else {
        String[] text = file.read();
        for (int j = 0; j < text.length; ++j)
            System.out.println(indent + "\t" + text[j]);
    }
}

}

public static void
main(String[] args)
{
    int status = 0;
    Ice.Communicator ic = null;
    try {
        // Create a communicator
        //
        ic = Ice.Util.initialize(args);

        // Create a proxy for the root directory
        //
        Ice.ObjectPrx base
            = ic.stringToProxy("RootDir:default -p 10000");
        if (base == null)
            throw new RuntimeException("Cannot create proxy");

        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir
            = DirectoryPrxHelper.checkedCast(base);
        if (rootDir == null)
            throw new RuntimeException("Invalid proxy");

        // Recursively list the contents of the root directory
        //
        System.out.println("Contents of root directory:");
        listRecursive(rootDir, 0);
    } catch (Ice.LocalException e) {
        e.printStackTrace();
        status = 1;
    } catch (Exception e) {
        System.err.println(e.getMessage());
        status = 1;
    } finally {
        // Clean up
        //
    }
}
```

```

        if (ic != null)
            ic.destroy();
    }

    System.exit(status);
}
}

```

在导入了 `Filesystem package` 之后，`Client` 类定义了两个方法：`listRecursive`，这是用于打印文件系统内容的助手方法；`main`，这是主程序。让我们先看一看 `main`：

1. `main` 的代码结构遵循了我们在第 3 章看到过的结构。在初始化 `run time` 之后，客户创建一个代理，指向文件系统的根目录。在这个例子中，我们假定服务器运行在本地主机上，并且使用缺省协议（`TCP/IP`）在 10000 端口处侦听。根目录的对象标识叫作 `RootDir`。
2. 客户把代理向下转换成 `DirectoryPrx`，并把这个代理传给 `listRecursive`，由它打印出文件系统的内容。

大多数工作都是在 `listRecursive` 中完成的。这个函数收到的参数是一个代理，指向要列出的目录；另外还有一个缩进层次参数（缩进层次随着每一次递归调用增加，这样，打印每个节点名时的缩进层次就会与该节点的树深度对应）。`listRecursive` 调用目录的 `list` 操作，并且遍历所返回的节点序列：

1. 代码调用 `checkedCast`，把 `Node` 代理窄化成 `Directory` 代理；并且调用 `uncheckedCast`，把 `Node` 代理窄化成 `File` 代理。在这两个转换中只有、而且肯定会有一个成功，所以不需要两次调用 `checkedCast`：如果节点是一个 `Directory`，代理就使用 `checkedCast` 返回的 `DirectoryPrx`；如果 `checkedCast` 失败，我们知道了这个节点是一个 `File`，因此，要获得 `FilePrx`，使用 `uncheckedCast` 就足够了。

一般而言，如果你知道向下转换到特定类型能成功，就最好使用 `uncheckedCast`，而不是 `checkedCast`，因为 `uncheckedCast` 不需要进行任何网络通信。

2. 代码打印文件或目录的名字，然后，取决于成功的转换是哪一个，在名字的后面打印 "`directory`" 或 "`file`"。
3. 代码检查节点的类型：
 - 如果是目录，代码就会递归，同时增加缩进层次。
 - 如果是文件，代表就调用文件的 `read` 操作，取回文件内容，然后遍历返回的内容行序列，打印每一行。

假定我们有一个很小的文件系统，由两个文件和一个目录组成：

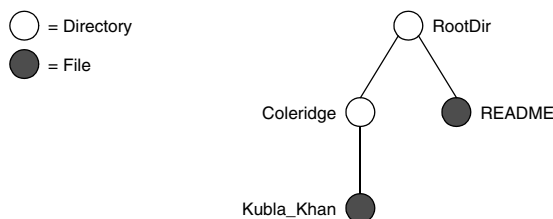


图 9.1. 一个小文件系统

这个文件的客户产生的输出是：

```
Contents of root directory:
  README (file):
    This file system contains a collection of poetry.
  Coleridge (directory):
    Kubla_Khan (file):
      In Xanadu did Kubla Khan
      A stately pleasure-dome decree:
      Where Alph, the sacred river, ran
      Through caverns measureless to man
      Down to a sunless sea.
```

注意，迄今为止，我们的客户（以及服务器）并不很成熟：

- 协议和地址信息是硬写在代码中的。
- 客户进行了一些并非绝对必要的远地过程调用；只要稍稍重新设计一下 Slice 定义，就可以避免许多这样的调用。

我们将在第 20 章和 XREF 中看到怎样消除这些缺点。

9.3 总结

这一章介绍了一个非常简单的客户，用于访问我们在第 5 章开发的文件系统服务器。你可以看到，你很难把这些 Java 代码和一个普通的 Java 程序区分开来。这是 Ice 的最大的优点之一：访问远地对象就和访问普通的本地 Java 对象一样容易。这样，你就可以把精力放在该放的地方，也就是说，集中精力开发你的应用逻辑，而不用去和晦涩的网络 APIs 作斗争。我们将在第 13 章看到，对服务器端来说也同样是如此，也就是说，你可以轻松而高效地开发分布式应用。

第 10 章

服务器端的 Slice-to-C++ 映射

10.1 本章综述

在这一章，我们将介绍服务器端的 Slice-to-C++ 映射（服务器端的 Slice-to-Java 映射见第 12 章）。10.3 节讨论怎样初始化和结束服务器端 run time，10.4 节到 10.6 节说明怎样实现接口和操作，10.7 节讨论怎样向服务器端 Ice run time 登记对象。

10.2 引言

在客户端和服务器端，Slice 数据类型映射到 C++ 的方式是一样的。这意味，第 6 章的所有内容也适用于服务器端。但关于服务器端，你需要了解另外一些内容，其中有：

- 怎样初始化和结束服务器端 run time。
- 怎样实现 servants。
- 怎样传递参数和抛出异常。
- 怎样创建 servants，并向 Ice run time 登记它们。

我们将在本章的余下部分讨论这些主题。

10.3 服务器端 main 函数

Ice run time 的主要进入点是由本地接口 `Ice::Communicator` 来表示的。和在客户端一样，在你在服务器中做任何别的事情之前，你必须调用 `Ice::initialize`，对 Ice run time 进行初始化。`Ice::initialize` 返回一个智能指针，指向一个 `Ice::Communicator` 实例：

```
int
main(int argc, char * argv[])
{
    Ice::CommunicatorPtr ic
        = Ice::initialize(argc, argv);
    // ...
}
```

`Ice::initialize` 接受的参数是 C++ 的 `argc` 和 `argv` 的引用。这个函数扫描参数向量，查找任何与 Ice run time 有关的命令行选项；任何这样的选项都会从参数向量中被移除，这样，当 `Ice::initialize` 返回时，剩余的选项和参数都与你的应用有关。如果在初始化过程中出了任何问题，`initialize` 会抛出异常。

在离开你的 `main` 函数之前，你必须调用 `Communicator::destroy`。`destroy` 操作负责结束 Ice run time。特别地，`destroy` 会等待任何还在运行的操作祈用完成。此外，`destroy` 还会确保任何还未完成的线程都得以汇合（`joined`），并收回一些操作系统资源，比如文件描述符和内存。决不要让你的 `main` 函数不先调用 `destroy` 就终止；这样做会导致不确定的行为。

因此，我们的服务器端 `main` 函数大体上像是这样：

```
#include <Ice/Ice.h>

int
main(int argc, char * argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);

        // Server code here...

    } catch (const Ice::Exception & e) {
        cerr << e << endl;
        status = 1;
    } catch (const std::string & msg) {
```

```

        cerr << msg << endl;
        status = 1;
    } catch (const char * msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}

```

注意，这段代码把对 `Ice::initialize` 的调用放在了 `try` 块中，并且会负责把正确的退出状态返回给操作系统。还要注意，只有在初始化曾经成功的情况下，代码才会尝试销毁通信器。

`const std::string &`和`const char *`的 `catch` 处理服务器是作为一种方便的特性出现的：如果我们在服务器代码的任何地方遇到致命的错误情况，我们可以直接抛出含有出错消息的串或串直接量；这会使栈回退到 `main`，由 `main` 打印出错消息，销毁通信器，然后返回非零的退出状态，继而终止。

10.3.1 Ice::Application 类

前面的 `main` 函数所用的结构很常用，所以 `Ice` 提供 `Ice::Application` 类，封装了所有正确的初始化和结束活动。下面是这个类的定义（省略了一些细节）：

```

namespace Ice {
    class Application /* ... */ {
    public:
        Application();
        virtual ~Application();

        int main(int, char * [], const char * = 0);
        virtual int run(int, char * []) = 0;

        static const char * appName();
        static CommunicatorPtr communicator();
        // ...
    };
}

```

这个类的意图是，你对 `Ice::Application` 进行特化，在你的派生类中实现 `run` 纯虚方法。你通常会放在 `main` 中的代码，都要放进 `run` 方法。使用 `Ice::Application`，我们的程序看起来像这样：

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {

        // Server code here...

        return 0;
    }
};

int
main(int argc, char * argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

Application::main 函数会做这样一些事情:

1. 针对 Ice::Exception 安装一个异常处理器。如果你的代码没有处理某个 Ice 异常, Application::main 会先在 stderr 上打印异常的详细情况, 然后返回非零的返回值。
2. 针对 const std::string & 和 const char * 安装一个异常处理器。这样, 在对致命的错误情况进行响应时, 你可以抛出一个 std::string 或串直接量, 从而终止你的服务器。Application::main 会在 stderr 上打印这个串, 然后返回非零的返回值。
3. 初始化 (通过调用 Ice::initialize) 和结束 (通过调用 Communicator::destroy) 通信器。你可以调用静态的 communicator() 成员, 访问你的服务器的通信器。
4. 扫描参数向量, 查找与 Ice run time 有关的选项, 并移除这样的选项。因此, 在传给你的 run 方法的参数向量中, 不再有与 Ice 有关的选项, 而只有针对你的应用的选项和参数。
5. 通过静态的 appName 成员函数, 提供你的应用的名字。这个调用的返回值是 argv[0], 所以, 你可以在你的代码的任何地方调用 Ice::Application::appName, 从而获得 argv[0] (通常在打印错误消息时需要这一信息)。
6. 创建一个 IceUtil::CtrlCHandler, 适当地关闭通信器。

`Ice::Application` 能够确保你的程序适当地结束 Ice run time，不管你的服务器是正常终止的，还是因为对异常或信号作出响应而终止的。我们建议你在所有程序中都使用这个类；这样做能够让你的生活更轻松。此外，`Ice::Application` 还提供了信号处理以及配置特性，当你使用这个类时，你无需自己实现这些特性。

在客户端使用 `Ice::Application`

你也可以把 `Ice::Application` 用于你的客户：只需从 `Ice::Application` 派生一个类，把客户代码放进它的 `run` 方法，就可以了。这种做法带来的好处与在服务器端一样：即使是在发生异常的情况下，`Ice::Application` 也能确保正确销毁通信器。

捕捉信号

我们在第 3 章开发的服务器无法干净地关闭自己：我们简单地从命令行中断服务器，迫使它退出。对于许多现实应用而言，以这样的方式终止服务器是不可接受的：在典型情况下，服务器在终止之前必须进行一些清理工作，比如刷出数据库缓冲区，或者关闭网络连接。要想在收到信号或键盘中断时，防止数据库文件或其他持久数据损坏，这样的清理工作特别重要。

为了让信号处理更容易一点，`Ice::Application` 在 `IceUtil::CtrlCHandler` 类中封装了平台相关的信号处理能力（参见 15.11 节）。这样，你就能在收到信号时干净地关闭应用，而且，不管底层使用的是哪种操作系统和线程库，你都能使用同样的源码。

```
namespace Ice {
    class Application : /* ... */ {
    public:
        // ...
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();
    };
}
```

你既可以在 Windows 上、也可以在 UNIX 上使用 `Ice::Application`：对于 UNIX，各成员函数控制的是你的应用在收到 **SIGINT**、**SIGHUP**，以及 **SIGTERM** 时的行为；对于 Windows，各成员函数控制的是你的应用在收到 **CTRL_C_EVENT**、**CTRL_BREAK_EVENT**、

CTRL_CLOSE_EVENT、**CTRL_LOGOFF_EVENT**，以及
CTRL_SHUTDOWN_EVENT 时的行为。

下面是各成员函数的行为：

- **shutdownOnInterrupt**

这个函数创建一个 `IceUtil::CtrlCHandler`，用于干净地关闭你的应用。这是缺省行为。

- **ignoreInterrupt**

这个函数使信号被忽略。

- **holdInterrupt**

这个函数临时阻塞信号递送（`signal delivery`）。

- **releaseInterrupt**

这个函数使信号递送恢复成先前的安排。当你调用 `releaseInterrupt` 时，在 `holdInterrupt` 被调用之后到达的任何信号都会被递送。

- **interrupted**

如果此前是信号造成了通信器的关闭，这个函数返回 `true`，否则返回 `false`。据此，我们可以区分有意的关闭和信号造成的被迫关闭。例如，这可以用于日志记录。

在缺省情况下，`Ice::Application` 的表现就好像 `shutdownOnInterrupt` 被祈用过一样，因此，要确保程序在收到信号时干净地终止，我们的服务器的 `main` 函数不需要变动。但我们增加了一个诊断功能，报告信号的发生，所以我们的 `main` 函数现在看起来像这样：

```
#include <Ice/Ice.h>

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {

        // Server code here...

        if (interrupted())
            cerr << appName() << ": terminating" << endl;

        return 0;
    }
};

int
```



```
main(int argc, char * argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

注意，如果你的服务器被信号中断，Ice run time 会等待目前正在执行的所有操作完成。这意味着，如果一个操作正在对持久状态进行更新，它不会因被中断而发生部分更新（partial update）问题。

Ice::Application 和属性

除了在这一节给出的功能，Ice::Application 还负责用属性值初始化 Ice run time。通过属性，你能够以各种方式配置 run time。例如，你可以用属性控制线程池尺寸或服务器端口号。我们将在第 14 章更详细地讨论属性。

Ice::Application 的局限

Ice::Application 是一个单体（singleton）类，会创建单个通信器。如果你要使用多个通信器，你不能使用 Ice::Application。相反，你必须像我们在第 3 章看到的那样安排你的代码结构（一定要记得销毁通信器）。

10.3.2 Ice::Service 类

一般而言，10.3.1 节描述的 Ice::Application 类对于 Ice 客户和服务端来说非常方便。但在有些情况下，应用可能需要作为 Unix 看守（daemon）或 Win32 服务运行在系统一级。对于这样的情况，Ice 提供了 Ice::Service，一个可与 Ice::Application 相比的单体类，但它还封装了低级的、针对特定平台的初始化和关闭步骤——系统服务常常需要使用这样的步骤。下面是 Ice::Service 类的定义：

```
namespace Ice {
    class Service {
    public:
        Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int, char * []);
        Ice::CommunicatorPtr communicator() const;
```

```

        static Service * instance();

protected:
    virtual bool start(int, char * []) = 0;
    virtual void waitForShutdown();
    virtual bool stop();
    virtual Ice::CommunicatorPtr initializeCommunicator(int &,
char * []);

    void enableInterrupt();
    void disableInterrupt();

    void syserror(const std::string &) const;
    void error(const std::string &) const;
    void warning(const std::string &) const;
    void trace(const std::string &) const;

    bool win9x() const;

    // ...
};
}

```

使用 `Ice::Service` 类的 `Ice` 应用至少要定义一个子类，并重新定义 `start` 成员函数；服务必须在这个函数里执行其启动活动，比如处理命令行参数、创建对象适配器、登记 `servants`。应用的 `main` 函数必须实例化这个子类，祈用其 `main` 成员函数，把程序的参数向量作为参数传给它。下面的例子给出了一个最小限度的 `Ice::Service` 子类：

```

#include <Ice/Ice.h>
#include <Ice/Service.h>

class MyService : public Ice::Service {
protected:
    virtual bool start(int, char * []);
private:
    Ice::ObjectAdapterPtr _adapter;
};

bool
MyService::start(int argc, char * argv[])
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(new MyServantI);
    _adapter->activate();
    return true;
}

```

```
}

int
main(int argc, char * argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

`Service::main` 成员函数会执行下面的任务序列：

1. 扫描参数向量，看其中是否有保留的选项、让程序作为系统服务运行。还有一些保留选项可用于管理任务。
2. 让程序准备好作为系统服务运行（如果有必要的话）。
3. 按照一个 `IceUtil::CtrlCHandler`（参见 15.11 节），以进行适当的信号处理。
4. 祈用 `initializeCommunicator` 成员函数，获取一个通信器。你可以使用 `communicator` 成员函数来访问通信器实例。
5. 祈用 `start` 成员函数。如果 `start` 返回表示失败的 `false`，`main` 就销毁通信器，并立即返回。
6. 祈用 `waitForShutdown` 成员函数，这个函数会阻塞到 `shutdown` 被祈用。
7. 祈用 `stop` 成员函数。如果 `stop` 返回 `true`，`main` 就认为应用已经成功终止。
8. 销毁通信器。
9. 得体地（`gracefully`）终止系统服务（如果有必要的话）。

如果 `Service::main` 捕捉到未被处理的异常，在日志中就会记载一条描述性的消息，然后通信器销毁，服务终止。

Ice::Service 成员函数

子类可以用 `Ice::Service` 中的虚成员函数来拦截 `main` 成员函数的活动。所有这些虚成员函数（除了 `start`）都有缺省实现。

- `Ice::CommunicatorPtr`
`initializeCommunicator(int & argc, char * argv[])`

初始化通信器。缺省实现祈用 `Ice::initialize`，并把指定的参数向量传给它。

- `void interrupt()`

信号处理器祈用它来表示收到了信号。缺省实现祈用 `shutdown` 成员函数。

- `bool shutdown()`

让服务器开始关闭的过程。缺省实现祈用通信器的 `shutdown`。如果关闭已成功开始，子类必须返回 `true`，否则返回 `false`。

- `bool start(int argc, char * argv[])`

允许子类执行它的启动活动，比如扫描所提供的参数向量、识别命令行选项，创建对象适配器，以及登记 `servants`。如果启动成功，子类必须返回 `true`，否则返回 `false`。

- `bool stop()`

允许子类在终止之前进行清理。缺省实现什么也不做，只是返回 `true`。如果服务成功停止，子类必须返回 `true`，否则返回 `false`。

- `void waitForShutdown()`

无限期地等待服务关闭。缺省实现会祈用通信器的 `waitForShutdown`。

下面描述类定义中的非虚成员函数：

- `void disableInterrupt()`

禁用 `Ice::Service` 的信号处理行为。禁用之后，信号会被忽略。

- `void enableInterrupt()`

启用 `Ice::Service` 的信号处理行为。启用之后，如果有信号发生，`interrupt` 成员函数会被调用。

- `static Service * instance()`

返回 `Ice::Service` 单体实例。

- `int main(int argc, char * argv[])`

提供 `Ice::Service` 类的主逻辑。在本节的前面已经描述了这个函数所执行的任务。如果成功，它返回 `EXIT_SUCCESS`，否则返回 `EXIT_FAILURE`。

- `void syserror(const std::string & msg) const`

- `void error(const std::string & msg) const`

- `void warning(const std::string & msg) const`

- `void trace(const std::string & msg) const`

为方便使用通信器的日志记录器而提供的函数。`syserror` 成员函数包括了系统的当前错误代码的描述。

- `bool win9x() const`

如果程序是在 Windows 95/98/ME 上运行，就返回真。这个函数只能在 Windows 平台上使用。

Unix 看守

在 Unix 平台上，`Ice::Service` 能识别以下命令行选项：

- **--daemon**

指明程序应该作为看守运行。这涉及到创建一个后台子进程，`Service::main` 将在这个子进程中执行其任务。在子进程成功析用 `start` 成员函数之前¹，父进程不会终止。除非另外收到指示，否则 `Ice::Service` 会把子进程的当前工作目录变更为根目录，并关闭所有无用的文件描述符。注意，在通信器初始化之前，各文件描述符不会关闭，也就是说，在这段时间里，标准输入、标准输出，以及标准错误都可以使用。例如，`IceSSL` 插件可能需要在标准输入上提示输入口令，而如果设置了 `Ice.PrintProcessId`，`Ice` 可能要在标准输出上打印子进程 id。

- **--noclose**

阻止 `Ice::Service` 关闭无用的文件描述符。在调试和诊断过程中，这可能会很有用，因为这样一来，你就可以通过看守的标准输出和标准错误进行输出了。

- **--nochdir**

阻止 `Ice::Service` 变更当前工作目录。

--noclose 和 **--nochdir** 选项只能和 **--daemon** 一起指定。在传给 `start` 成员函数的参数向量中，这些选项会被移除。

Win32 服务

在 Win32 平台上²，如果指定了 **--service** 选项，`Ice::Service` 会把应用作为 Windows 服务启动：

- **--service NAME**

作为名叫 **NAME** 的 Windows 服务启动。在传给 `start` 成员函数的参数向量中，这个选项会被移除。

1. 用 shell 脚本启动看守常常会带来不确定性，上述行为消除了这一不确定性，因为它确保了命令析用不会在看守准备好接收请求之前就完成。

2. 在 Windows 95/98/ME 上不支持 Windows 服务。

但是，在应用作为 Windows 服务运行之前，它必须先被安装，因此，Ice::Service 类还支持另外一些的命令行选项，用于执行管理活动：

- **--install NAME [--display DISP] [--executable EXEC] [ARG ...]**

安装 **NAME** 服务。如果指定了 **--display** 选项，就把 **DISP** 用作服务的显示名，否则就使用 **NAME**。如果指定了 **--executable** 选项，就把 **EXEC** 用作服务的可执行路径名，否则就使用可执行文件的路径名来祈用 **--install**。其他任何参数都会不加改变地传给 Service::start 成员函数。注意，在启动时传给服务的参数集中，这个命令会自动增加命令行参数 **--service NAME**，因此，你不需要显式地指定这些选项。

- **--uninstall NAME**

移除 **NAME** 服务。如果服务目前是活动的，在反安装之前，必须先使它停止。

- **--start NAME [ARG ...]**

启动 **NAME** 服务。其他任何参数都会不加改变地传给 Service::start 成员函数。

- **--stop NAME**

停止 **NAME** 服务。

如果指定的管理命令不止一个，或者在使用 **--service** 的同时还使用了管理命令，就会发生错误。在执行了管理命令之后，程序会立即终止。

Ice::Service 类支持 Windows 服务控制代码 SERVICE_CONTROL_INTERROGATE 和 SERVICE_CONTROL_STOP。在收到 SERVICE_CONTROL_STOP 时，Ice::Service 会祈用 shutdown 成员函数。

日志记录考虑事项

当应用作为 Unix 看守或 Windows 服务运行时，Ice::Logger 通常并不适用，因为它的输出会发往标准错误，从而会丢失。应用可以实现定制的日志记录器，也可以使用 Ice 提供的其他日志记录器：

- 在 Unix 上，通过 Ice.UseSyslog 属性，你可以选用一种使用 syslog 设施的日志记录器实现。
- 在 Windows 上，通过 Ice.UseEventLog 属性，你可以让日志消息被记录到 Windows 事件消息中。这个日志记录器实现会自动在注册表中增加必要的键，以让服务使用事件日志。

注意，在为 Windows 服务的启动做准备时，`Ice::Service` 会创建一个临时的 Windows 事件日志记录器实例，在通信器成功初始化、为通信器配置的日志记录器可用之前，都会使用这个临时的日志记录器。因此，即使一个失败的应用被配置成使用另外的日志记录器实现，在 Windows 事件日志中也有可能已经记录了有用的诊断信息。

如果 `Ice::Service` 的一个子类需要手工安装日志记录器实现，这个子类应该重新定义 `initializeCommunicator` 成员函数。

10.4 接口的映射

服务器端的接口映射为 Ice run time 提供了一个向上调用（up-call）API：通过在 `servant` 类中实现虚函数，你提供的挂钩可以把控制线程从服务器端的 Ice run time 引到你的应用代码中。

10.4.1 骨架类

在客户端，接口映射到代理类（参见 5.12 节）。在服务器端，接口映射到骨架类。对于相应的接口上的每个操作，骨架类都有一个对应的纯虚方法。例如，再次考虑一下我们在第 5 章定义的 Node 接口的 Slice 定义：

```
module Filesystem {  
    interface Node {  
        nonmutating string name();  
    };  
    // ...  
};
```

Slice 编译器为这个接口生成这样的定义：

```
namespace Filesystem {  
  
    class Node : virtual public Ice::Object {  
    public:  
        virtual std::string name(const Ice::Current & =  
                                Ice::Current()) const = 0;  
  
        // ...  
    };  
    // ...  
}
```

目前，我们将忽略这个类的其他一些成员函数。要注意的要点是：

- 和客户端一样，Slice 模块映射到名字相同的 C++ 名字空间，所以骨架类定义会放在名字空间 `Filesystem` 中。
- 骨架类的名字与 Slice 接口的名字（`Node`）相同。
- 对于 Slice 接口中的每个操作，骨架类都有一个对应的纯虚成员函数。
- 骨架类是抽象基类，因为它的成员函数是纯虚函数。
- 骨架类继承自 `Ice::Object`（这个类形成了 Ice 对象层次的根）。

10.4.2 Servant 类

要给 Ice 对象提供实现，你必须创建 `servant` 类，继承对应的骨架类。例如，要为 `Node` 接口创建 `servant`，你可以编写：

```
#include <Filesystem.h> // Slice-generated header

class NodeI : public virtual Filesystem::Node {
public:
    NodeI(const std::string &);
    virtual std::string name(const Ice::Current &) const;
private:
    std::string _name;
};
```

按照惯例，`servant` 类的名字是它们接口的名字加上后缀 `I`，所以 `Node` 接口的 `servant` 类叫作 `NodeI`（这只是一个惯例：从 Ice run time 的角度来说，你可以为你的 `servant` 类选用任何你喜欢的名字）。

注意，`NodeI` 继承自 `Filesystem::Node`，也就是说，它派生自它的骨架类。在定义 `servant` 类时总是使用虚继承是个好主意。严格地说，只有其实现的接口使用了多继承的 `servant` 才必须使用虚继承；但 `virtual` 关键字并无害处，同时，如果你在开发的中途给接口层次增加多继承，你无需回去给你的所有 `servant` 类增加 `virtual` 关键字。

从 Ice 的角度来说，`NodeI` 类只须实现一个成员函数：继承自骨架的 `name` 纯虚函数。这使得 `servant` 类成了一个能实例化的具体类。你可以按照你的实现的需要，增加其他成员函数和数据成员。例如，在前面的定义中，我们增加了一个 `_name` 成员和一个构造器。显然，构造器用于初始化 `_name` 成员，而 `name` 函数用于返回这个成员的值：

```
NodeI::NodeI(const std::string & name) : _name(name)
{
}

std::string
```



```
NodeI::name(const Ice::Current &) const
{
    return _name;
}
```

普通的、idempotent，以及 nonmutating 操作

第 248 页上的 NodeI 骨架的 name 成员函数是一个 const 成员函数。const 关键字是 Slice 编译器加上的，因为 name 是一个 nonmutating 操作（参见 3.8.1 节）。与之相反，普通操作和 idempotent 操作是非 const 成员函数。例如，下面的接口含有一个普通操作，一个 idempotent 操作，以及一个 nonmutating 操作：

```
interface Example {
    void normalOp();
    idempotent void idempotentOp();
    nonmutating void nonmutatingOp();
};
```

下面是这个接口的骨架类：

```
class Example : virtual public Ice::Object {
public:
    virtual void normalOp(const Ice::Current &
                          = Ice::Current()) = 0;
    virtual void idempotentOp(const Ice::Current &
                              = Ice::Current()) = 0;
    virtual void nonmutatingOp(const Ice::Current &
                               = Ice::Current()) const = 0;
    // ...
};
```

注意，只有 nonmutating 操作会映射成 const 成员函数；普通操作和 idempotent 操作会映射成平常的函数。

10.5 参数传递

对于一个 Slice 操作的每一个参数，C++ 映射都会为骨架中对应的虚成员函数生成一个对应的参数。此外，每一个操作都有一个额外的、放在最后的参数，类型是 Ice::Current。例如，Node 接口的 name 操作没有参数，但 Node 骨架类的 name 成员函数却有一个类型为 Ice::Current 的参数。我们将在 16.5 节解释这个参数的用途，而现在会暂时忽略它。

服务器端的参数传递所遵循的规则和客户端一样：

- in 参数通过值或 const 引用传递。
- out 参数通过引用传递。
- 返回值通过值传递。

为了说明这些规则，考虑下面的接口，它在所有可能的方向上传递串参数：

```
interface Example {
    string op(string sin, out string sout);
};
```

下面是为这个接口生成的骨架类：

```
class Example : virtual public ::Ice::Object {
public:
    virtual std::string
        op(const std::string &, std::string &,
           const Ice::Current & = Ice::Current()) = 0;

    // ...
};
```

你可以看到，这里并无让人惊奇之处。例如，我们可以这样实现 op：

```
std::string
ExampleI::op2(const std::string & sin,
              std::string & sout,
              const Ice::Current &)
{
    cout << sin << endl;           // In parameters are initialized
    sout = "Hello World!";         // Assign out parameter
    return "Done";                 // Return a string
}
```

与你通常编写的把串传入和传出函数的代码相比，这段代码没有任何不同；尽管牵涉到了远地过程调用，这些代码却并没有受到任何影响。对于其他类型（比如代理、类，或词典）而言同样也是如此：参数传递规则和普通的 C++ 规则一样，不需要特殊的 API 调用或内存管理³。

3. 这与 CORBA C++ 映射形成了鲜明的对比，后者的参数传递规则非常复杂，太容易造成内存泄漏，或产生不确定的行为。

10.6 引发异常

要从操作实现中抛出异常，你只需实例化异常，初始化，然后抛出它。例如：

```
void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    // Try to write the file contents here...
    // Assume we are out of space...
    if (error) {
        Filesystem::GenericError e;
        e.reason = "file too large";
        throw e;
    }
};
```

在发生异常时，不会出现内存管理问题。

注意，无论对应的 Slice 操作定义是否有异常规范，Slice 编译器都不会为操作生成异常规范。这出于慎重的思考：C++ 异常规范不能给我们增加任何价值，所以 Ice C++ 映射没有使用它（对 C++ 异常规范的问题的极好讨论，参见 [20]）。

如果你随意抛出异常（比如 `int` 或其他意料之外的类型），Ice run time 会捕捉该异常，然后向客户返回 `UnknownLocal Exception`。与此类似，如果你抛出“不可能的”用户异常（没有在操作的异常规范中列出的用户异常），客户会收到 `UnknownUserException`。

抛出 Ice 系统异常也一样：例如，如果你抛出 `MemoryLimitException`，客户会收到 `UnknownLocal Exception`⁴。因此，你决不应该从操作实现中抛出系统异常。如果你这样做了，客户所看到的只是 `UnknownLocal Exception`，这并不能给客户带来任何有用的信息。

4. 在把异常返回给客户时，有三种系统异常不会变成 `UnknownLocal Exception`：`ObjectNotExistException`、`OperationNotExistException`，以及 `FacetNotExistException`。我们将在 XREF 中更详细地讨论这些异常。

10.7 对象体现

在创建了像 10.4.2 节的 `NodeI` 类那样的 `servant` 类之后，你可以实例化这样的类，创建具体的 `servant`，用以接收来自客户的祈用。但是，只是实例化 `servant` 类并不足以体现对象。明确地说，要提供 `Ice` 对象的实现，你必须采取遵循以下步骤：

1. 实例化 `servant` 类。
2. 为这个 `servant` 所体现的 `Ice` 对象创建标识。
3. 向 `Ice` run time 告知这个 `servant` 的存在。
4. 把这个对象的代理传给客户，以让客户访问它。

10.7.1 实例化 `Servant`

实例化 `servant` 意味着在栈上分配其实例：

```
NodePtr servant = new NodeI("Fred");
```

这行代码在堆上创建一个新的 `NodeI` 实例，把它的地址赋给类型为 `NodePtr` 的智能指针（参见第 180 页）。这之所以可行，是因为 `NodeI` 派生自 `Node`，所以类型为 `NodePtr` 的智能指针可以照管类型为 `NodeI` 的实例。但是，如果这时我们想要祈用 `NodeI` 类的成员函数，我们就会遇到问题：我们无法通过 `NodePtr` 智能指针访问 `NodeI` 类的成员函数（C++ 类型规则不允许我们通过指向基类的指针访问派生类的成员）。为了解决这一问题，我们可以这样修改代码：

```
typedef IceUtil::Handle<NodeI> NodeIPtr;  
NodeIPtr servant = new NodeI("Fred");
```

这两行代码利用我们在 6.14.5 节介绍的智能指针模板，把 `NodeIPtr` 定义为指向 `NodeI` 实例的智能指针。你是要使用 `NodePtr` 还是 `NodeIPtr` 类型的智能指针，完全取决于你是否想祈用 `NodeI` 派生类的成员函数；如果不是这样，使用 `NodePtr` 就足够了，你不需要定义 `NodeIPtr` 类型。

无论你使用 `NodePtr` 还是 `NodeIPtr`，按照 6.14.5 节的讨论，使用智能指针类的好处都很明显：使用它们，不再可能发生偶然的内存泄漏。

10.7.2 创建标识

每个 Ice 对象都需要一个标识。在使用同一个对象适配器的所有 servant 中，该标识必须是唯一的⁵。Ice 对象标识是一种结构，下面是它的 Slice 定义：

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

对象的完整标识由 Identity 的 name 和 category 域组成。我们将暂时让 category 域仍为空串，而只是使用 name 域（关于 category 域的讨论，参见 16.6 节）。

要创建标识，我们只需把用于标识 servant 的键赋给 Identity 结构的 name 域：

```
Ice::Identity id;
id.name = "Fred"; // Not unique, but good enough for now
```

10.7.3 激活 Servant

只是创建 servant 实例并没有用处：只有在你显式地把 servant 告知对象适配器之后，Ice run time 才会知道这个 servant 的存在。要激活 servant，你要祈用对象适配器的 add 操作。假定我们能够访问 _adapter 变量中的对象适配器，我们可以编写：

```
_adapter->add(servant, id);
```

注意 add 的两个参数：指向 servant 的智能指针，以及对象标识。调用对象适配器的 add 操作，会把 servant 指针和 servant 的标识增加到适配器的 servant 映射表中，并把“Ice 对象的代理”与“服务器内存中的正确的 servant 实例”链接在一起，如下所示：

1. 除了寻址信息，Ice 对象的代理还含有该对象的标识。当客户祈用操作时，对象标识会随同请求一起发给服务器。
2. 对象适配器接收请求，取得标识，把该标识用作 servant 映射表的索引。

5. Ice 对象模型假定所有对象（不管它们是否使用同一个适配器）的标识都是全局唯一的。进一步的讨论，参见 XREF。

3. 如果具有该标识的 `servant` 是活动的，对象适配器就从 `servant` 映射表中取得 `servant` 指针，把到来的请求分派给 `servant` 上正确的成员函数。

假定对象适配器处在活动状态（参见 16.3.5 节），一旦你调用 `add`，客户请求就会被分派给 `servant`。

Servant 的生命期和引用计数

综合前面所讲的内容，我们可以编写一个简单的函数，实例化并激活一个 `NodeI` `servant`。为了这个例子，我们使用了一个简单的叫作 `activateServant` 的助手函数，用它来创建并激活具有指定标识的 `servant`：

```
void
activateServant(const string & name)
{
    NodePtr servant = new NodeI(name);           // Refcount == 1
    Ice::Identity id;
    id.name = name;
    _adapter->add(servant, id);                   // Refcount == 2
}                                                  // Refcount == 1
```

注意，我们是在堆上创建 `servant` 的，一旦 `activateServant` 返回，我们就会失去指向该 `servant` 的最后一个句柄（因为 `servant` 变量出了作用域）。问题是，这个在堆上分配的 `servant` 实例会怎样？答案在于智能指针语义：

- 当有新的 `servant` 被实例化操作时，它的引用计数被初始化成 0。
- 把 `servant` 的地址赋给 `servant`，智能指针会使 `servant` 的引用计数增加到 1。
- 在调用 `add`、把 `servant` 智能指针传给对象适配器操作时，对象适配器会在内部保留该句柄的一个副本。这会使 `servant` 的引用计数增加到 2。
- 当 `activateServant` 返回时，`servant` 变量的析构器会使 `servant` 的引用计数减到 1。

实际效果就是，只要 `servant` 还在它的对象适配器的 `servant` 映射表中，它就仍然会保留在堆上，其引用计数为 1（如果我们解除这个 `servant` 的激活，也就是说，从 `servant` 映射表中移除它，引用计数就会降到零，它所占用的内存就会被回收；我们将在 XREF 中讨论这些生命周期问题）。

10.7.4 用 UUID 做标识

我们在 2.5.1 节讨论过，Ice 对象模型假定对象标识是全局唯一的。确保这样的唯一性的一种途径是使用 UUID（Universally Unique Identifiers）[14] 做标识。IceUtil 名字空间含有一个助手函数，可以创建这样的标识：

```
#include <IceUtil/UUID.h>
#include <iostream>

int
main()
{
    cout << IceUtil::generateUUID() << endl;
}
```

这个程序在执行时会打印出一个唯一的串，比如 5029a22c-e333-4f87-86b1-cd5e0fcce509。对 generateUUID 的每一次调用都会创建一个和以往不同的串⁶。你可以用这样的 UUID 来创建对象标识。为方便起见，对象适配器提供了 addWithUUID 操作，只要一步，就可以生成一个 UUID、并把 servant 增加到 servant 映射表中。我们可以使用这个操作，重写第 254 页上的代码：

```
void
activateServant(const string & name)
{
    NodePtr servant = new NodeI(name);
    _adapter->addWithUUID(servant);
}
```

10.7.5 创建代理

一旦我们激活了 Ice 对象的 servant，服务器就可以处理针对这个对象的客户请求了。但是，只有拥有了对象的代理，客户才能访问该对象。如果客户知道服务器的详细的地址信息及对象标识，它可以根据一个串来创建代理，就像我们在第 3 章的第一个例子中看到的那样。但是，客户以这种方式创建代理，通常只是为了访问用于引导的初始对象。一旦客户拥有了初始代理，它通常会调用一些操作来进一步获取其他代理。

6. 喔，几乎是这样——到最后，UUID 算法会绕回去重复自己，但这大概要 3400 年才会发生。

对象适配器含有创建代理所需的全部详细资料：寻址信息和协议信息，还有对象标识。Ice run time 提供了几种创建代理的途径。一经创建，你可以把代理当作返回值、或者当作操作祈用的 out 参数传给客户。

代理与 Servant 激活

对象适配器的 add 和 addWithUUID servant 激活操作会返回对应的 Ice 对象的一个代理。这意味着，我们可以编写：

```
typedef IceUtil::Handle<NodeI> NodeIPtr;
NodeIPtr servant = new NodeI(name);
NodePrx proxy = NodePrx::uncheckedCast(
    _adapter->addWithUUID(servant));

// Pass proxy to client...
```

在这段代码中，addWithUUID 会在一步之内激活 servant、并返回这个 servant 所体现的 Ice 对象的一个代理。

注意，在此我们需要使用 uncheckedCast，因为 addWithUUID 返回的代理的类型是 Ice::ObjectPrx。

直接的代理创建

对象适配器提供了一个操作，可以根据指定的标识创建代理：

```
module Ice {
    local interface ObjectAdapter {
        Object* createProxy(Identity id);
        // ...
    };
};
```

注意，不管具有该标识的 servant 是否已被激活，createProxy 都会根据指定标识创建一个代理。换句话说，代理自身的生命周期与 servant 的生命周期完全无关：

```
Ice::Identity id;
id.name = IceUtil::generateUUID();
ObjectPrx o = _adapter->createProxy(id);
```

这段代码会为一个 Ice 对象创建一个代理，这个对象的标识是由 generateUUID 生成的。显然，该对象还没有 servant 存在，如果我们把这个代理返回给客户，而客户祈用了代理上的操作，客户就会收到 ObjectNotExistException（我们将在 XREF 中更详细地考察这些生命周期问题）。

10.8 总结

这一章介绍了服务器端的 C++ 映射。因为对客户和服务器而言，Slice 数据类型的映射是一样的，与客户端相比，服务器端映射只额外增加了几种机制：一个用于初始化和结束 run time 的小 API，再加上一些规则，用于处理怎样从骨架派生 servant 类，以及怎样向服务器端 run time 登记 servant。

尽管这一章的例子非常简单，它们准确地反映了 Ice 服务器的基本编写方式。当然，对于更加复杂的服务器而言（我们将在第 16 章加以讨论），你还需要使用另外一些 API——例如，为了改善性能或可伸缩性。但这些 API 都是用 Slice 描述的，所以，要使用这些 API，除了我们在这里描述的 C++ 映射规则以外，你不需要再学习其他规则。

第 11 章

开发 C++ 文件系统服务器

11.1 本章综述

在这一章，我们将给出一个 C++ 服务器的源码，实现我们在第 5 章开发的文件系统（Java 版本的服务器见第 13 章）。除了必需的线程互锁，我们在这里给出的代码完全能工作（我们将在第 15 章详细考察线程问题）。

11.2 实现文件系统服务器

我们现在所知道的服务器端 C++ 映射，已经足以让我们为第 5 章开发的文件系统实现一个服务器（在研究源码之前，你可以先回顾一下第 5 章的文件系统的 Slice 定义）。

我们的服务器器由两个源文件组成：

- `Server.cpp`

这个文件含有服务器主程序。

- `FilesystemI.cpp`

这个文件含有文件系统 servants 的实现。

11.2.1 服务器的 main 程序

Server.cpp 文件中的服务器主程序使用了我们在 10.3.1 节讨论过的 Ice::Application。run 方法安装信号处理器、创建对象适配器、为文件系统里的目录和文件创建一些 servants，然后激活适配器。这样，main 程序看起来就像：

```
#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Create an object adapter (stored in the NodeI::_adapter
        // static member)
        //
        NodeI::_adapter =
            communicator()->createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI("/", 0);

        // Create a file called "README" in the root directory
        //
        FilePtr file = new FileI("README", root);
        Lines text;
        text.push_back("This file system contains"
            "a collection of poetry.");
        file->write(text);

        // Create a directory called "Coleridge" in
        // the root directory
        //
        DirectoryIPtr coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file called "Kubla_Khan" in the
        // Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
```

```

        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);

        // All objects are created, allow client requests now
        //
        NodeI::_adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName()
                << ": received signal, shutting down" << endl;
        }
        return 0;
    };
};

int
main(int argc, char* argv[])
{
    FilesystemApp app;
    return app.main(argc, argv);
}

```

这些代码不算少，所以让我们来详细考察每一个部分：

```

#include <FilesystemI.h>
#include <Ice/Application.h>

using namespace std;
using namespace Filesystem;

```

这段代码包括了头文件 `FilesystemI.h`（参见第 271 页）。后者会包括 `Ice/Ice.h`，以及 `Slice` 编译器生成的头文件 `Filesystem.h`。因为我们在使用 `Ice::Application`，我们还需要包括 `Ice/Application.h`。

我们使用了两个针对名字空间 `std` 和 `Filesystem` 的 `using` 声明，让我们的源码不那么繁琐。

源码接下来的部分是 `FilesystemApp` 的定义，这个类派生自 `Ice::Application`，在其 `run` 方法中含有主应用逻辑：

```

class FilesystemApp : virtual public Ice::Application {
public:
    virtual int run(int, char * []) {
        // Create an object adapter (stored in the NodeI::_adapter
        // static member)
        //
        NodeI::_adapter =
            communicator()->createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI("/", 0);

        // Create a file called "README" in the root directory
        //
        FilePtr file = new FileI("README", root);
        Lines text;
        text.push_back("This file system contains"
            "a collection of poetry.");
        file->write(text);

        // Create a directory called "Coleridge" in
        // the root directory
        //
        DirectoryIPtr coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file called "Kubla_Khan" in the
        // Coleridge directory
        //
        file = new FileI("Kubla_Khan", coleridge);
        text.erase(text.begin(), text.end());
        text.push_back("In Xanadu did Kubla Khan");
        text.push_back("A stately pleasure-dome decree:");
        text.push_back("Where Alph, the sacred river, ran");
        text.push_back("Through caverns measureless to man");
        text.push_back("Down to a sunless sea.");
        file->write(text);

        // All objects are created, allow client requests now
        //
        NodeI::_adapter->activate();

        // Wait until we are done
        //
    }
};

```



```

communicator()->waitForShutdown();
if (interrupted()) {
    cerr << appName()
        << ": received signal, shutting down" << endl;
}
return 0;
};
};

```

这里的大部分代码都是我们先前见过的公式化代码：我们创建对象适配器，然后到最后，激活对象适配器，并调用 `waitForShutdown`。

有意思的代码是适配器创建代码：服务器实例化我们的文件系统的几个节点，创建了图 11.1 所示的结构。

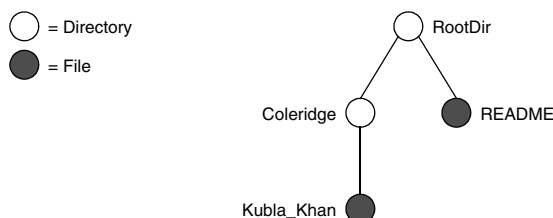


图 11.1. 一个小文件系统

我们很快就会看到，我们的目录和文件的 `servant` 的类型分别是 `DirectoryI` 和 `FileI`。这两种类型的 `servant` 的构造器都接受两个参数：要创建的目录或文件的名称，以及指向父目录的 `servant` 的句柄（对于没有父目录的根目录，我们会传递 `null` 父句柄）。因此，下面的语句

```
DirectoryIPtr root = new DirectoryI("/", 0);
```

会创建根目录，名字是 `"/"`，没有父目录。注意，我们使用了在 6.14.5 节讨论过的智能指针类来存放 `new` 的返回值；这样，我们就不会再有任何内存管理问题。`DirectoryIPtr` 类型在头文件 `FilesystemI.h`（参见第 271 页）中定义，如下所示：

```
typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;
```

下面的代码建立图 11.1 中的结构：

```

// Create the root directory (with name "/" and no parent)
//
DirectoryIPtr root = new DirectoryI("/", 0);

// Create a file called "README" in the root directory

```

```
//
FilePtr file = new FileI("README", root);
Lines text;
text.push_back("This file system contains"
               "a collection of poetry.");
file->write(text);

// Create a directory called "Coleridge" in
// the root directory
//
DirectoryIPtr coleridge
    = new DirectoryI("Coleridge", root);

// Create a file called "Kubla_Khan" in the
// Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text.erase(text.begin(), text.end());
text.push_back("In Xanadu did Kubla Khan");
text.push_back("A stately pleasure-dome decree:");
text.push_back("Where Alph, the sacred river, ran");
text.push_back("Through caverns measureless to man");
text.push_back("Down to a sunless sea.");
file->write(text);
```

我们首先创建根目录，并在根目录中创建 README 文件（注意，当我们创建类型为 FileI 的新节点时，我们传递的父引用是指向根目录的引用）。

下一步是用文本填充文件：

```
FilePtr file = new FileI("README", root);
Lines text;
text.push_back("This file system contains"
               "a collection of poetry.");
file->write(text);
```

回想一下 6.7.3 节的内容，Slice 序列会映射到 STL 向量。Slice 类型 Lines 是串的序列，所以 C++ 类型 Lines 是串的向量；我们调用该向量的 push_back，给我们的 README 文件增加一行文件。

最后，我们调用我们的 FileI servant 的 Slice write 操作：

```
file->write(text);
```

这条语句很有意思：服务器代码祈用它自己的 servant 上的操作。因为这个调用是通过智能类指针（类型是 FilePtr）、而不是代理（类型是 FilePrx）进行的，Ice run time 甚至不知道发生了这个调用——像这样对

servant 的直接调用，不会由 Ice run time 居中协调，而是会作为平常的 C++ 函数调用进行分派。

余下的代码以类似的方式，创建叫作 Coleridge 的子目录，并在该目录中创建叫作 Kubla_Khan 的文件，从而完成了图 11.1 中的结构。

11.2.2 Servant 类定义

我们必须为我们的 Slice 规范中的具体接口提供 servant，也就是说，我们必须在 C++ 类 FileI 和 DirectoryI 中为 File 和 Directory 接口提供 servant。这意味着，我们的 servant 类看起来可能像这样：

```
namespace Filesystem {  
    class FileI : virtual public File {  
        // ...  
    };  
  
    class DirectoryI : virtual public Directory {  
        // ...  
    };  
}
```

于是就有了图 11.2 所示的 C++ 类结构。

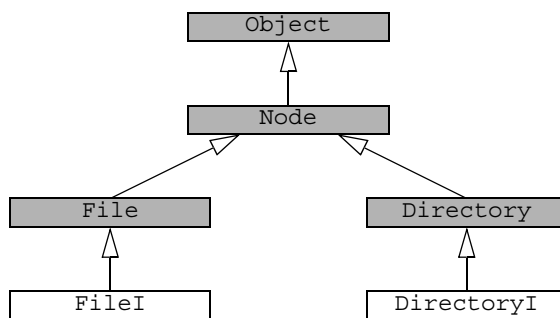


图 11.2. 使用接口继承的文件系统 servants

图 11.2 中有阴影的类是骨架类，无阴影的是我们的 servant 实现。如果我们像这样实现我们的 servant，FileI 必须实现从 File 骨架继承的纯虚操作（read 和 write），以及从 Node 骨架继承的操作（name）。与此类似，DirectoryI 必须实现从 Directory 骨架继承的纯虚操作（list），以及从 Node 骨架继承的操作（name）。以这种方式实现

servant 使用的是对 Node 的接口继承（interface inheritance），因为没有从这个类继承实现代码。

换一种方法，我们还可以用下面的定义实现我们的 servant：

```
namespace Filesystem {  
    class NodeI : virtual public Node {  
        // ...  
    };  
  
    class FileI : virtual public File,  
                 virtual public NodeI {  
        // ...  
    };  
  
    class DirectoryI : virtual public Directory,  
                      virtual public NodeI {  
        // ...  
    };  
}
```

于是就有了图 11.3 所示的 C++ 类结构。

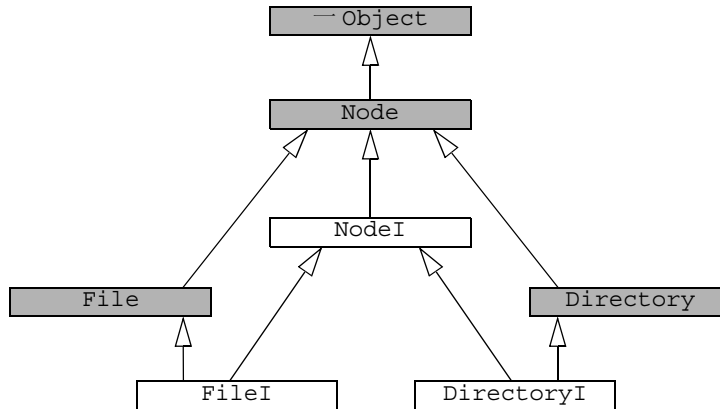


图 11.3. 使用实现继承的文件系统

在这种实现里，`NodeI` 是一个具体基类，它实现了从 `Node` 骨架继承的 `name` 操作。`FileI` 和 `DirectoryI` 对 `NodeI` 和它们各自的骨架进行了多继承，也就是说，`FileI` 和 `DirectoryI` 对它们的 `NodeI` 基类进行了实现继承（implementation inheritance）。

这两种实现都同等有效。究竟选择哪一种，取决于我们是否想复用 `NodeI` 提供的共有代码。我们为下面的实现选择了第二种途径，即使用实现继承。

假如我们使用图 11.3 中的结构，以及我们在文件系统的 `Slice` 定义中定义的操作，我们可以给我们的 `servants` 的类定义增加这些操作：

```
namespace Filesystem {
    class NodeI : virtual public Node {
    public:
        virtual std::string name(const Ice::Current &) const;
    };

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::Lines read(const Ice::Current &) const;
        virtual void write(const Filesystem::Lines &,
                           const Ice::Current &);
    };

    class DirectoryI : virtual public Directory,
                       virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::NodeSeq list(const Ice::Current &) const;
    };
}
```

这会把操作实现的型构增加到每个类（注意，这些型构必须与生成的骨架类中的操作型构完全相符——否则你就会重载基类中的纯虚函数，而不是重新定义它，也就是说，`servant` 类将无法实例化，因为它仍然是抽象的。为了避免发生型构失配，你可以从生成的头文件（`Filesystem.h`）中复制型构）。

现在我们已经有了基本的结构，我们需要思考一下用以支持我们的 `servant` 实现的其他方法和数据成员。在典型情况下，每个 `servant` 类都会隐藏复制构造器和赋值构造器，并且有一个用以给数据成员提供初始状态的构造器。假定我们的文件系统的所有节点都有名字和父目录，这意味着，`NodeI` 类应该实现这样的功能：跟踪每个节点的名字，以及父 - 子关系：

```
namespace Filesystem {
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : virtual public Node {
    public:
```

```

        virtual std::string name(const Ice::Current &) const;
        NodeI(const std::string &, const DirectoryIPtr & parent);
        static Ice::ObjectAdapterPtr _adapter;
    private:
        const std::string _name;
        DirectoryIPtr _parent;
        NodeI(const NodeI &);                // Copy forbidden
        void operator=(const NodeI &);       // Assignment forbidden
    };
}

```

NodeI 类有两个 **private** 数据成员，用于存储它的名字（类型是 `std::string`）和它的父目录（类型是 `DirectoryIPtr`）。构造器的参数用于设置这些数据成员的值。按照惯例，对于根目录，我们会把一个 **null** 句柄传给构造器，说明根目录没有父目录。我们还增加了一个 **public** 静态变量，用以存放一个智能指针，指向的是我们在服务器中使用的（单个）对象适配器；这个变量由第 264 页的 `Filesystem::run` 方法初始化。

`FileI servant` 类必须存储其文件的内容，所以它需要有一个用于此目的的数据成员。我们可以方便地用生成的 `Lines` 类型（`std::vector<std::string>`）来存放文件内容，一个串存一行。因为 `FileI` 继承自 `NodeI`，它还需要有一个构造器，参数是文件名和父目录。于是就有了下面的类定义：

```

namespace Filesystem {
    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::Lines read(const Ice::Current &) const;
        virtual void write(const Filesystem::Lines &,
                           const Ice::Current &);
        FileI(const std::string &, const DirectoryIPtr &);
    private:
        Lines _lines;
    };
}

```

就目录而言，每个目录都必须存储它的子节点的列表。我们可以方便地使用生成的 `NodeSeq` 类型（`vector<NodePrx>`）来做到这一点。因为 `DirectoryI` 继承自 `NodeI`，我们需要增加一个构造器，用以初始化目录名及其父目录。我们很快就会看到，为了能更容易地把新创建的目录和它的父目录连接起来，我们还需要一个 **private** 助手 `addChild` 函数。于是就有了这样的类定义：

```

namespace Filesystem {
    class DirectoryI : virtual public Directory,
                      virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::NodeSeq list(const Ice::Current &) const;
        DirectoryI(const std::string &, const DirectoryIPtr &);
        void addChild(NodePrx child);
    private:
        NodeSeq _contents;
    };
}

```

把这些代码综合在一起，我们最后得到了一个 servant 头文件 `FilesystemI.h`，如下所示：

```

#include <Ice/Ice.h>
#include <Filesystem.h>

namespace Filesystem {
    class DirectoryI;
    typedef IceUtil::Handle<DirectoryI> DirectoryIPtr;

    class NodeI : virtual public Node {
    public:
        virtual std::string name(const Ice::Current &) const;
        NodeI(const std::string &, const DirectoryIPtr & parent);
        static Ice::ObjectAdapterPtr _adapter;
    private:
        const std::string _name;
        DirectoryIPtr _parent;
        NodeI(const NodeI &);           // Copy forbidden
        void operator=(const NodeI &); // Assignment forbidden
    };

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        virtual Filesystem::Lines read(const Ice::Current &) const;
        virtual void write(const Filesystem::Lines &,
                           const Ice::Current &);
        FileI(const std::string &, const DirectoryIPtr &);
    private:
        Lines _lines;
    };

    class DirectoryI : virtual public Directory,

```

```

        virtual public Filesystem::NodeI {
public:
    virtual Filesystem::NodeSeq list(const Ice::Current &) const;
    DirectoryI(const std::string &, const DirectoryIPtr &);
    void addChild(NodePrx child);
private:
    NodeSeq _contents;
};
}

```

11.2.3 Servant 实现

遵循我们的 `FilesystemI.h` 头文件中的类定义，我们的大部分 servant 实现都很平常。

实现 `FileI`

文件的 `read` 和 `write` 操作很平常：我们只是把传入的文件内容存储在 `_lines` 数据成员中。构造器同样也很平常，只是把它的参数传给 `NodeI` 基类构造器：

```

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    _lines = text;
}

Filesystem::FileI::FileI(const string & name,
                        const DirectoryIPtr & parent
                        ) : NodeI(name, parent)
{
}

```

实现 `DirectoryI`

`DirectoryI` 的实现同样也很平常：`list` 操作简单地返回 `_contents` 数据成员，构造器把它的参数传给 `NodeI` 基类构造器：


```

Filesystem::NodeSeq
Filesystem::DirectoryI::list(const Ice::Current &) const
{
    return _contents;
}

Filesystem::DirectoryI::DirectoryI(const string & name,
                                     const DirectoryIPtr & parent
                                     ) : NodeI(name, parent)
{
}

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}

```

唯一需要注意的是 `addChild` 的实现：当有新的目录或文件被创建时，`NodeI` 基类的构造器会调用它自己的父亲的 `addChild`，把指向新创建的孩子的代理传给它。`addChild` 的实现会把传入的引用添加到目录（也就是父目录）的内容列表中。

实现 NodeI

我们的 `NodeI` 类的 `name` 操作也很平常：它简单地返回 `_name` 数据成员：

```
std::string
Filesystem::NodeI::name(const Ice::Current &) const
{
    return _name;
}
```

我们实现的大部分内容在 `NodeI` 构造器中。在这里，我们会创建每个节点的代理，并把父节点和子节点连接起来：

```
Filesystem::NodeI::NodeI(const string & name,
                        const DirectoryIPtr & parent
                        ) : _name(name), _parent(parent)
{
    // Create an identity. The parent has the fixed identity "/"
    //
    Ice::Identity myID = Ice::stringToIdentity(parent
        ? IceUtil::generateUUID()
        : "RootDir");
}
```

```

// Create a proxy for the new node and add it as
// a child to the parent
//
NodePrx thisNode
    = NodePrx::uncheckedCast(_adapter->createProxy(myID));
if (parent)
    parent->addChild(thisNode);

// Activate the servant
//
_adapter->add(this, myID);
}

```

第一步是创建每个节点的唯一标识，对于根目录，我们使用固定标识 "RootDir"。这使得客户能够创建根目录的代理（参见 7.2 节）。对于其他非根目录的目录，我们用一个 UUID 做标识（第 255 页）。

第二步是创建子节点的代理，并调用 `addChild`，把子节点增加到父目录的内容列表中。这会把子节点与父节点连接在一起。

最后，我们需要激活 `servant`，从而让 Ice run time 知道 `servant` 的存在，于是我们调用了对象适配器的 `add`。

我们的 `servant` 的实现就完成了。下面再一次给出完整的源码：

```

#include <FilesystemI.h>
#include <IceUtil/UUID.h>
#include <time.h>

using namespace std;

Ice::ObjectAdapterPtr Filesystem::NodeI::_adapter;

// Slice Node::name() operation

std::string
Filesystem::NodeI::name(const Ice::Current &) const
{
    return _name;
}

// NodeI constructor

Filesystem::NodeI::NodeI(const string & name,
                        const DirectoryIPtr & parent
                        ) : _name(name), _parent(parent)
{
    // Create an identity. The parent has the fixed identity "/"

```

```

//
Ice::Identity myID = Ice::stringToIdentity(parent
                                           ? IceUtil::generateUUID()
                                           : "RootDir");

// Create a proxy for the new node and add it
// as a child to the parent
//
NodePrx thisNode
    = NodePrx::uncheckedCast(_adapter->createProxy(myID));
if (parent)
    parent->addChild(thisNode);

// Activate the servant
//
_adapter->add(this, myID);
}

// Slice File::read() operation

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    return _lines;
}

// Slice File::write() operation

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    _lines = text;
}

// FileI constructor

Filesystem::FileI::FileI(const string & name,
                        const DirectoryIPtr & parent
                        ) : NodeI(name, parent)
{
}

// Slice Directory::list() operation

Filesystem::NodeSeq

```

```
Filesystem::DirectoryI::list(const Ice::Current &) const
{
    return _contents;
}

// DirectoryI constructor

Filesystem::DirectoryI::DirectoryI(const string & name,
                                   const DirectoryIPtr & parent
                                   ) : NodeI(name, parent)
{
}

// addChild is called by the child in order to add
// itself to the _contents member of the parent

void
Filesystem::DirectoryI::addChild(const NodePrx child)
{
    _contents.push_back(child);
}
```

11.3 总结

这一章说明了怎样为我们在第 5 章定义的文件系统实现一个完整的服务器。注意，这个服务器引人注目的特点是，它没有多少与分布处理有关的代码：服务器的大部分代码都只是应用逻辑，如果你编写一个非分布式的版本，同样也要编写这样的代码。这又是 Ice 的重要好处之一：应用代码不需要考虑分布事务，所以你可以专注于应用逻辑、而不是网络基础设施的开发。

注意，按照现在的情况，我们在这里给出的服务器代码并不十分正确：如果有两个客户分别通过不同的线程、同时访问同一个文件，一个线程可能在读取 `_lines` 数据成员，而另一个线程在更新它。显然，如果发生这样的情况，我们可能会写入或返回垃圾数据，或者更糟糕，使服务器崩溃。要让 `read` 和 `write` 操作成为线程安全的其实很容易，只要一个数据成员和两行代码就足够了。我们将在第 15 章讨论怎样编写线程安全的 `servant` 实现。

第 12 章

服务器端的 Slice-to-Java 映射

12.1 Chapter Overview

在这一章，我们将介绍服务器端的 Slice-to-Java 映射（服务器端的 Slice-to-C++ 映射见第 10 章）。12.3 节讨论怎样初始化和结束服务器端 run time，12.4 节到 12.7 节说明怎样实现接口和操作，12.8 节讨论怎样向服务器端 Ice run time 登记对象。

12.2 引言

在客户端和服务端，Slice 数据类型映射到 C++ 的方式是一样的。这意味着，第 8 章的所有内容也适用于服务器端。但关于服务器端，你需要了解另外一些内容，其中有：

- 怎样初始化和结束服务器端 run time
- 怎样实现 servants
- 怎样传递参数和抛出异常
- 怎样创建 servants，并向 Ice run time 登记它们。

我们将在本章的余下部分讨论这些主题。

12.3 服务器端 main 函数

Ice run time 的主要进入点是由本地接口 `Ice::Communicator` 来表示的。和在客户端一样，在你在服务器中做任何别的事情之前，你必须调用 `Ice.Util.initialize`，对 Ice run time 进行初始化。`Ice.Util.initialize` 返回一个引用，指向一个 `Ice.Communicator` 实例：

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        // ...
    }
}
```

`Ice.Util.initialize` 接受的参数向量是由操作系统传给 `main` 的。这个函数扫描参数向量，查找任何与 Ice run time 有关的命令行选项，但不会移除这些选项¹。如果在初始化过程中出了任何问题，`initialize` 会抛出异常。

在离开你的 `main` 函数之前，你必须调用 `Communicator::destroy`。`destroy` 操作负责结束 Ice run time。特别地，`destroy` 会等待任何还在运行的操作祈用完成。此外，`destroy` 还会确保任何还未完成的线程都得以汇合（joined），并收回一些操作系统资源，比如文件描述符和内存。决不要让你的 `main` 函数不先调用 `destroy` 就终止；这样做会导致不确定的行为。

因此，我们的服务器端 `main` 函数大体上像是这样：

1. Java 数组不允许 `Ice.Util.initialize` 修改参数向量的尺寸。但是，Ice 提供了另一个重载的 `Ice.Util.initialize`，允许应用获取一个移除了 Ice 选项的新参数向量。


```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        if (ic != null)
            ic.destroy();
        System.exit(status);
    }
}
```

注意，这段代码把对 `Ice::initialize` 的调用放在了 `try` 块中，并且会负责把正确的退出状态返回给操作系统。还要注意，只有在初始化曾经成功的情况下，代码才会尝试销毁通信器。

12.3.1 Ice.Application 类

前面的 `main` 函数所用的结构很常用，所以 `Ice` 提供 `Ice.Application` 类，封装了所有正确的初始化和结束活动。下面是这个类的概况（省略了一些细节）：

```
package Ice;

public abstract class Application {
    public Application()

    public final int main(String appName, String[] args)

    public final int
        main(String appName, String[] args, String configFile)

    public abstract int run(String[] args);

    public static String appName()
```

```
        public static Communicator communicator()

        // ...
    }
```

这个类的意图是，你对 `Ice.Application` 进行特化，在你的派生类中实现 `run` 抽象方法。你通常会放在 `main` 中的代码，都要放进 `run` 方法。使用 `Ice.Application`，我们的程序看起来像这样：

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

The `Application.main` 函数会做这样一些事情：

1. 针对 `java.lang.Exception` 安装一个异常处理器。如果你的代码没有处理某个 `Ice` 异常，`Application.main` 会先在 `System.err` 上打印异常的名字和栈踪迹，然后返回非零的返回值。
2. 初始化（通过调用 `Ice.Util.initialize`）和结束（通过调用 `Communicator.destroy`）通信器。你可以调用静态的 `communicator` 访问器，访问你的服务器的通信器。
3. 扫描参数向量，查找与 `Ice run time` 有关的选项，并移除这样的选项。因此，在传给你的 `run` 方法的参数向量中，不再有与 `Ice` 有关的选项，而只有针对你的应用的选项和参数。
4. 通过静态的 `appName` 成员函数，提供你的应用的名字。这个调用的返回值是调用 `Application.main` 时用的第一个参数，所以，你可以在你的代码的任何地方调用 `Ice.Application.appName`，从而获得这个名字（通常在打印错误消息时需要这一信息）。
5. 安装一个关闭挂钩，适当地关闭通信器。

`Ice.Application` 能够确保你的程序适当地结束 `Ice run time`，不管你的服务器是正常终止的，还是因为对异常或信号作出响应而终止的。我们建议你在所有程序中都使用这个类；这样做能够让你的生活更轻松。此外，`Ice.Application` 还提供了信号处理以及配置特性，当你使用这个类时，你无需自己实现这些特性。

在客户端使用 `Ice.Application`

你也可以把 `Ice.Application` 用于你的客户：只需从 `Ice.Application` 派生一个类，把客户代码放进它的 `run` 方法，就可以了。这种做法带来的好处与在服务器端一样：即使是在发生异常的情况下，`Ice.Application` 也能确保正确销毁通信器。

Catching Signals

我们在第 3 章开发的服务器无法干净地关闭自己：我们简单地从命令行中断服务器，迫使它退出。对于许多现实应用而言，以这样的方式终止服务器是不可接受的：在典型情况下，服务器在终止之前必须进行一些清理工作，比如刷出数据库缓冲区，或者关闭网络连接。要想在收到信号或键盘中断时，防止数据库文件或其他持久数据损坏，这样的清理工作特别重要。

Java 没有提供对信号的直接支持，但它 允许应用登记 *关闭挂钩*，当 JVM 关闭时会调用这样的关闭挂钩。有若干事件能够触发 JVM 关闭，比如调用 `System.exit`、或者操作系统发出了中断信号，但关闭挂钩不会收到对关闭原因的说明。

在缺省情况下，`Ice.Application` 会登记一个关闭挂钩，从而允许你在 JVM 关闭之前干净地终止你的应用：

```
package Ice;

public abstract class Application {
    // ...

    synchronized public static void shutdownOnInterrupt()

    synchronized public static void defaultInterrupt()

    synchronized public static boolean interrupted()
}
```

下面是各成员函数的行为：

- `shutdownOnInterrupt`

这个函数安装一个关闭挂钩，它会调用通信器的 `shutdown` 干净地关闭你的应用。这是缺省行为。

- `defaultInterrupt`

这个函数移除关闭挂钩。

- `interrupted`

如果此前是信号造成了通信器的关闭，这个函数返回真，否则返回假。据此，我们可以区分有意的关闭和 JVM 造成的被迫关闭。例如，这可以用于日志记录。

在缺省情况下，`Ice.Application` 的表现就好像 `shutdownOnInterrupt` 被祈用过一样，因此，要确保程序在 JVM 关闭时干净地终止，我们的服务器的 `main` 函数不需要变动。但我们增加了一个诊断功能，报告这件事情的发生，所以我们的 `main` 函数现在看起来像这样：

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        if (interrupted())
            System.err.println(appName() + ": terminating");

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

Ice.Application 和属性

除了在这一节给出的功能，`Ice.Application` 还负责用属性值初始化 Ice run time。通过属性，你能够以各种方式配置 run time。例如，你可以用属性控制线程池尺寸或服务器端口号。`Ice.Application` 的 `main`

函数是重载的；第二个版本允许你指定一个配置文件名，这个文件将在初始化过程中被处理。我们将在第 14 章更详细地讨论属性。

Ice.Application 的局限

Ice.Application 是一个单体（singleton）类，会创建单个通信器。如果你要使用多个通信器，你不能使用 Ice.Application。相反，你必须像我们在第 3 章看到的那样安排你的代码结构（一定要记得销毁通信器）。

12.4 接口的映射

服务器端的接口映射为 Ice run time 提供了一个向上调用（up-call）API：通过在 servant 类中实现成员函数，你提供的挂钩可以把控制线程从服务器端的 Ice run time 引到你的应用代码中。

12.4.1 骨架类

在客户端，接口映射到代理类（参见 5.12 节）。在服务器端，接口映射到骨架类。对于相应的接口上的每个操作，骨架类都有一个对应的纯虚方法。例如，再次考虑一下我们在第 5 章定义的 Node 接口的 Slice 定义：

```
module Filesystem {  
    interface Node {  
        nonmutating string name();  
    };  
    // ...  
};
```

Slice 编译器为这个接口生成这样的定义：

```
package Filesystem;  
  
public interface _NodeOperations  
{  
    String name(Ice.Current current);  
}  
  
public interface Node extends Ice.Object, _NodeOperations {}  
  
public abstract class _NodeDisp extends Ice.ObjectImpl
```

```

                                implements Node
{
    // Mapping-internal code here...
}

```

这里要注意的要点是：

- 和客户端一样，Slice 模块映射到名字相同的 Java packages，所以骨架类定义是 Filesystem package 的一部分。
- 对于每个 Slice 接口 *<interface-name>*，编译器都生成一个 Java 接口 *_<interface-name>Operations*（就这个例子而言是 *_NodeOperations*）。对于 Slice 接口中的每个操作，这个接口都含有一个对应的成员函数。
- 对于每个 Slice 接口 *<interface-name>*，编译器都生成一个 Java 接口 *<interface-name>*（就这个例子而言是 *Node*）。这个接口既扩展 *Ice.Object*，又扩展 *_<interface-name>Operations*。
- 对于每个 Slice 接口 *<interface-name>*，编译器都生成一个抽象类 *_<interface-name>Disp*（就这个例子而言是 *_NodeDisp*）。这个抽象类是实际的骨架类；你要从这个基类派生你的 *servant* 类。

12.4.2 Servant 类

要给 Ice 对象提供实现，你必须创建 *servant* 类，继承对应的骨架类。例如，要为 *Node* 接口创建 *servant*，你可以编写：

```

package Filesystem;

public final class NodeI extends _NodeDisp {

    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}

```

按照惯例，servant 类的名字是它们接口的名字加上后缀 I，所以 Node 接口的 servant 类叫作 NodeI（这只是一个惯例：从 Ice run time 的角度来说，你可以为你的 servant 类选用任何你喜欢的名字）。注意，NodeI 扩展了 _NodeDisp，也就是说，它派生自它的骨架类。

从 Ice 的角度来说，NodeI 类只须实现一个成员函数：继承自骨架的 name 纯虚函数。这使得 servant 类成了一个能实例化的具体类。你可以按照你的实现的需要，增加其他成员函数和数据成员。例如，在前面的定义中，我们增加了一个 _name 成员和一个构造器（显然，构造器用于初始化 _name 成员，而 name 函数用于返回它的值）。

普通的、idempotent，以及 nonmutating 操作

一个操作是平常的操作、还是 idempotent 或 nonmutating 操作，对操作的映射方式没有影响。为了说明这一点，考虑下面的接口：

```
interface Example {
    void normalOp();
    idempotent void idempotentOp();
    nonmutating void nonmutatingOp();
};
```

下面是这个接口的操作类：

```
public interface _ExampleOperations
{
    void normalOp(Ice.Current current);
    void idempotentOp(Ice.Current current);
    void nonmutatingOp(Ice.Current current);
}
```

注意，成员函数的型构没有受到 idempotent 和 nonmutating 限定符的影响（在 C++ 中，nonmutating 限定符会生成 C++ const 成员函数）。

12.5 参数传递

对于一个 Slice 操作中的每一个参数，Java 映射都会为 `<interface-name>Operations` 中的对应方法生成一个对应的参数。此外，每一个操作都有一个额外的、放在最后的参数，类型是 `Ice.Current`。例如，Node 接口的 name 操作没有参数，但 `_NodeOperations` 接口的 name 成员函数却有一个类型为 `Ice.Current` 的参数。我们将在 16.5 节解释这个参数的用途，而现在会暂时忽略它。

为了说明这些规则，考虑下面的接口，它在所有可能的方向上传递串参数：

```
interface Example {  
    string op(string sin, out string sout);  
};
```

下面是为这个接口生成的骨架类：

```
public interface _ExampleOperations  
{  
    String op(String sin, Ice.StringHolder sout,  
              Ice.Current current);  
}
```

你可以看到，这里并无让人惊奇之处。例如，我们可以这样实现 op：

```
public final class ExampleI extends _ExampleDisp {  
  
    public String op(String sin, Ice.StringHolder sout,  
                    Ice.Current current)  
    {  
        System.out.println(sin);    // In params are initialized  
        sout.value = "Hello World!"; // Assign out param  
        return "Done";  
    }  
}
```

与你通常编写的把串传入和传出函数的代码相比，这段代码没有任何不同；尽管牵涉到了远地过程调用，这些代码却并没有受到任何影响。对于其他类型（比如代理、类，或词典）而言同样也是如此：参数传递规则和普通的 Java 规则一样，不需要特殊的 API 调用。

12.6 引发异常

要从操作实现中抛出异常，你只需实例化异常，初始化，然后抛出它。例如：

```
// ...  
  
public void  
write(String[] text, Ice.Current current)  
    throws GenericError  
{
```



```
// Try to write file contents here...
// Assume we are out of space...
if (error) {
    GenericError e = new GenericError();
    e.reason = "file too large";
    throw e;
}
}
```

如果你随意抛出异常，Ice run time 会捕捉该异常，然后向客户返回 UnknownLocal Exception。如果你抛出 Ice 系统异常，事情同样也是如此：例如，如果你抛出 MemoryLimitException²，客户会收到 UnknownLocal Exception。因此，你决不应该从操作实现中抛出系统异常。如果你这样做了，客户所看到的只是 UnknownLocal Exception，这并不能给客户带来任何有用的信息。

12.7 Tie 类

我们在 12.4 节看到，在映射到骨架类时，servant 类需要从它的骨架类继承。有时这会带来问题：为了访问有些类库提供的功能，你需要从某个基类继承；因为 Java 不支持多继承，这意味着你不能使用这样的类库来实现你的 servant，因为你的 servant 不能同时继承库的类和骨架类。

为了使你能继续使用这样的类库，Ice 提供了一种编写 servants 的途径，用委托（delegation）取代继承。tie 类提供了对这种途径的支持。其思想是，你不是从骨架类继承，而是创建一个类（称为实现类或委托类），其中的方法与一个接口的操作相对应。在使用 slice2java 编译器时，你通过 --tie 选项来创建 tie 类。例如，对于我们在 12.4.1 节见过的 Node 接口，--tie 选项会让编译器创建和我们先前看到的代码完全一样的代码，但同时还产生一个额外的 tie 类。对于 <interface-name> 接口，生成的 tie 类的名字是 _<interface-name>Tie:

```
package Filesystem;

public class _NodeTie extends _NodeDisp {

    public _NodeTie() {}
}
```

-
2. 在把异常返回给客户时，有三种系统异常不会变成 UnknownLocal Exception: ObjectNotExistException、OperationNotExistException，以及 FacetNotExistException。我们将在 XREF 中更详细地讨论这些异常。

```
public
_NodeTie(_NodeOperations delegate)
{
    _ice_delegate = delegate;
}

public _NodeOperations
ice_delegate()
{
    return _ice_delegate;
}

public void
ice_delegate(_NodeOperations delegate)
{
    _ice_delegate = delegate;
}

public boolean
equals(java.lang.Object rhs)
{
    if(this == rhs)
    {
        return true;
    }
    if(!(rhs instanceof _NodeTie))
    {
        return false;
    }

    return _ice_delegate.equals((( _NodeTie)rhs)._ice_delegate)
;
}

public int
hashCode()
{
    return _ice_delegate.hashCode();
}

public String
name(Ice.Current current)
{
    return _ice_delegate.name(current);
}
```

```

    }

    private _NodeOperations _ice_delegate;
}

```

这看起来很吓人，实则不然：在本质上，生成的 tie 类就是一个 servant 类（它扩展了 `_NodeDisp`），负责把“对 Slice 操作的对应方法的调用”委托给你的实现类（参见图 12.1）。

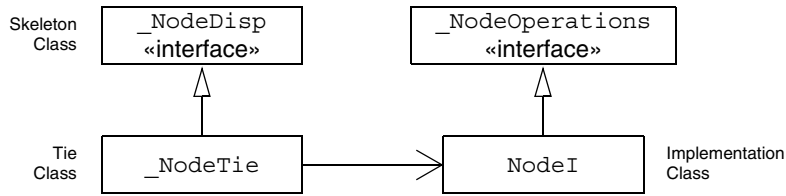


图 12.1. 一个骨架类、tie 类及实现类

有了这样的机制，我们可以这样为我们的 Node 接口创建一个实现类：

```

package Filesystem;

public final class NodeI implements _NodeOperations {
{
    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
}

```

注意，这个类和我们先前的实现是一样的，只是它实现了 `_NodeOperations` 接口，但没有扩展 `_NodeDisp`（这意味着，你现在可以随意扩展其他类来支持你的实现）。

要创建 servant，你要实例化你的实现类和 tie 类，把指向实现实例的引用传给 tie 构造器：

```

NodeI fred = new NodeI("Fred");           // Create implementation
_NodeTie servant = new _NodeTie(fred);     // Create tie

```

另外一种做法是，通过缺省方式构造 `tie` 类，在后面再调用 `ice_delegate` 设置它的委托实例：

```
_NodeTie servant = new _NodeTie();           // Create tie
// ...
NodeI fred = new NodeI("Fred");             // Create implementation
// ...
servant.ice_delegate(fred);                  // Set delegate
```

在使用 `tie` 类时，记住这样一点很重要：是 `tie` 实例、而不是你的委托实例是 `servant`。而且，在 `tie` 实例获得委托之前，你不能用它来体现（参见 12.8 节）Ice 对象。一旦你设置了委托实例，在 `tie` 实例的生命期内你不能改变它；否则就会产生不确定的结果。

你应该只在有此需要的情况下使用这种 `tie` 做法，也就是说，在你需要为实现你的 `servant` 而扩展某个基类的情况下——使用这种做法消耗的内存更多，因为每个 Ice 对象都要由两个（`tie` 对象和委托对象）、而不是一个 Java 对象体现。此外，`tie` 对象的调用分派或多或少要比平常的 `servant` 慢，因为 `tie` 把每个操作都转交给委托对象，也就是，每个操作祈用需要两个函数调用，而不是一个。

还要注意，除非你加以安排，否则你不能从委托对象回到 `tie` 对象。如果你有这样的需要，你可以把指向 `tie` 的引用存放在委托对象的一个成员中（例如，这个引用可以由委托实例的构造器初始化）。

12.8 对象体现

在创建了像 12.4.2 节的 `NodeI` 类那样的 `servant` 类之后，你可以实例化这样的类，创建具体的 `servant`，用以接收来自客户的祈用。但是，只是实例化 `servant` 类并不足以体现对象。明确地说，要提供 Ice 对象的实现，你必须采取遵循以下步骤：

1. 实例化 `servant` 类。
2. 为这个 `servant` 所体现的 Ice 对象创建标识。
3. 向 Ice run time 告知这个 `servant` 的存在。
4. 把这个对象的代理传给客户，以让客户访问它。

12.8.1 Instantiating a Servant

实例化 `servant` 意味着分配其实例：

```
Node servant = new NodeI("Fred");
```

这行代码创建一个新的 `NodeI` 实例，把它的地址赋给类型为 `Node` 的引用。这之所以可行，是因为 `NodeI` 派生自 `Node`，所以一个 `Node` 引用可以指向类型为 `NodeI` 的实例。但是，如果这时我们想要调用 `NodeI` 类的成员函数，我们必须使用 `NodeI` 引用：

```
NodeI servant = new NodeI("Fred");
```

你是要使用 `Node` 还是 `NodeI` 引用，完全取决于你是否想调用 `NodeI` 类的成员函数；如果不是这样，`Node` 所起的作用和 `NodeI` 引用完全一样。

12.8.2 创建标识

每个 `Ice` 对象都需要一个标识。在使用同一个对象适配器的所有 `servant` 中，该标识必须是唯一的³。`Ice` 对象标识是一种结构，下面是它的 `Slice` 定义：

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
    // ...
};
```

对象的完整标识由 `Identity` 的 `name` 和 `category` 域组成。我们将暂时让 `category` 域仍为空串，而只是使用 `name` 域（关于 `category` 域的讨论，参见 16.5 节）。

要创建标识，我们只需把用于标识 `servant` 的键赋给 `Identity` 结构的 `name` 域：

```
Ice.Identity id = new Ice.Identity();
id.name = "Fred"; // Not unique, but good enough for now
```

12.8.3 激活 Servant

只是创建 `servant` 实例并没有用处：只有在你显式地把 `servant` 告知对象适配器之后，`Ice` run time 才会知道这个 `servant` 的存在。要激活 `servant`，你

3. `Ice` 对象模型假定所有对象（不管它们是否使用同一个适配器）的标识都是全局唯一的。进一步的讨论，参见 XREF。

要祈用对象适配器的 `add` 操作。假定我们能够访问 `_adapter` 变量中的对象适配器，我们可以编写：

```
_adapter.add(servant, id);
```

注意 `add` 的两个参数：`servant` 和对象标识。对象适配器的 `add` 操作会把 `servant` 和 `servant` 的标识增加到适配器的 `servant` 映射表中，并把“Ice 对象的代理”与“服务器内存中的正确的 `servant` 实例”链接在一起，如下所示：

1. 除了寻址信息，Ice 对象的代理还含有该对象的标识。当客户祈用操作时，对象标识会随同请求一起发给服务器。
2. 对象适配器接收请求，取得标识，把该标识用作 `servant` 映射表的索引。
3. 如果具有该标识的 `servant` 是活动的，对象适配器就从 `servant` 映射表中取得 `servant`，把到来的请求分派给 `servant` 上正确的成员函数。

假定对象适配器处在活动状态（参见 16.3 节），一旦你调用 `add`，客户请求就会被分派给 `servant`。

12.8.4 用 UUID 做标识

我们在 2.5.1 节讨论过，Ice 对象模型假定对象标识是全局唯一的。确保这样的唯一性的一种途径是使用 UUID（Universally Unique Identifiers）[14] 做标识。Ice.Util package 含有一个助手函数，可以创建这样的标识：

```
public class Example {
    public static void
    main(String[] args)
    {
        System.out.println(Ice.Util.generateUUID());
    }
}
```

这个程序在执行时会打印出一个唯一的串，比如 5029a22c-e333-4f87-86b1-cd5e0fcce509。对 `generateUUID` 的每一次调用都会创建一个和以往不同的串⁴。你可以用这样的 UUID 来创建对象标识。为方便起见，对象适配器提供了 `addWithUUID` 操作，只要一步，

4. 喔，几乎是这样——到最后，UUID 算法会绕回去重复自己，但这大概要 3400 年才会发生。

就可以生成一个 UUID、并把 servant 增加到 servant 映射表中。使用这个操作，我们可以一步就创建一个标识、登记具有该标识的 servant：

```
_adapter.addWithUUID(new NodeI("Fred"));
```

12.8.5 创建代理

一旦我们激活了 Ice 对象的 servant，服务器就可以处理针对这个对象的客户请求了。但是，只有拥有了对象的代理，客户才能访问该对象。如果客户知道服务器的详细的地址信息及对象标识，它可以根据一个串来创建代理，就像我们在第 3 章的第一个例子中看到的那样。但是，客户以这种方式创建代理，通常只是为了访问用于引导的初始对象。一旦客户拥有了初始代理，它通常会调用一些操作来进一步获取其他代理。

对象适配器含有创建代理所需的全部详细资料：寻址信息和协议信息，还有对象标识。Ice run time 提供了几种创建代理的途径。一经创建，你可以把代理当作返回值、或者当作操作祈用的 out 参数传给客户。

代理与 Servant 激活

对象适配器的 add 和 addWithUUID servant 激活操作会返回对应的 Ice 对象的一个代理。这意味着，我们可以编写：

```
NodePrx proxy = NodePrxHelper.uncheckedCast(  
    _adapter.addWithUUID(new NodeI("Fred")));
```

在这段代码中，addWithUUID 会在一步之内激活 servant、并返回这个 servant 所体现的 Ice 对象的一个代理。

注意，在此我们需要使用 uncheckedCast，因为 addWithUUID 返回的代理的类型是 Ice.ObjectPrx。

直接的代理创建

对象适配器提供了一个操作，可以根据指定的标识创建代理：

```
module Ice {  
    local interface ObjectAdapter {  
        Object* createProxy(Identity id);  
        // ...  
    };  
};
```

注意，不管具有该标识的 servant 是否已被激活，createProxy 都会根据指定标识创建一个代理。换句话说，代理自身的生命周期与 servant 的生命周期完全无关：

```
Ice.Identity id = new Ice.Identity();  
id.name = Ice.Util.generateUUID();  
Ice.ObjectPrx o = _adapter.createProxy(id);
```

这段代码会为一个 Ice 对象创建一个代理，这个对象的标识是由 generateUUID 生成的。显然，该对象还没有 servant 存在，如果我们把这个代理返回给客户，而客户祈用了代理上的操作，客户就会收到 ObjectNotExistException（我们将在 XREF 中更详细地考察这些生命周期问题）。

12.9 总结

这一章介绍了服务器端的 Java 映射。因为对客户和服务器而言，Slice 数据类型的映射是一样的，与客户端相比，服务器端映射只额外增加了几种机制：一个用于初始化和结束 run time 的小 API，再加上一些规则，用于处理怎样从骨架派生 servant 类，以及怎样向服务器端 run time 登记 servant。

尽管这一章的例子非常简单，它们准确地反映了 Ice 服务器的基本编写方式。当然，对于更加复杂的服务器而言（我们将在第 16 章加以讨论），你还需要使用另外一些 API——例如，为了改善性能或可伸缩性。但这些 API 都是用 Slice 描述的，所以，要使用这些 API，除了我们在这里描述的 C++ 映射规则以外，你不需要再学习其他规则。

第 13 章

开发 Java 文件系统服务器

13.1 本章综述

在这一章，我们将给出一个 Java 服务器的源码，实现我们在第 5 章开发的文件系统（C++ 版本的服务器见第 11 章）。除了必需的线程互锁，我们在这里给出的代码完全能工作（我们将在第 15 章详细考察线程问题）。

13.2 实现文件系统服务器

我们现在所知道的服务器端 Java 映射，已经足以让我们为第 5 章开发的文件系统实现一个服务器（在研究源码之前，你可以先回顾一下第 5 章的文件系统的 Slice 定义）。

我们的服务器由三个源文件组成：

- `Server.java`

这个文件含有服务器主程序。

- `Filesystem/DirectoryI.java`

这个文件含有 `Directory servant` 的实现。

- `Filesystem/FileI.java`

这个文件含有 `File servant` 的实现。

13.2.1 服务器的 main 程序

Server.java 中的服务器主程序使用了我们在 12.3.1 节讨论过的 Ice.Application 类。run 方法安装关闭挂钩、创建对象适配器、为文件系统里的目录和文件创建一些 servants，然后激活适配器。这使得 main 程序看起来像是这样：

```
import FileSystem.*;

public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Create an object adapter (stored in the _adapter
        // static members)
        //
        Ice.ObjectAdapter adapter
            = communicator().createObjectAdapterWithEndpoints(
                "SimpleFilesystem", "default -p 10000");
        DirectoryI._adapter = adapter;
        FileI._adapter = adapter;

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryI root = new DirectoryI("/", null);

        // Create a file "README" in the root directory
        //
        File file = new FileI("README", root);
        String[] text;
        text = new String[] {
            "This file system contains a collection of poetry."
        };
        try {
            file.write(text, null);
        } catch (GenericError e) {
            System.err.println(e.reason);
        }

        // Create a directory "Coleridge" in the root directory
        //
        DirectoryI coleridge
            = new DirectoryI("Coleridge", root);

        // Create a file "Kubla_Khan" in the Coleridge directory
        //
```

```
file = new FileI("Kubla_Khan", coleridge);
text = new String[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };

try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}

// All objects are created, allow client requests now
//
adapter.activate();

// Wait until we are done
//
communicator().waitForShutdown();

return 0;
}

public static void
main(String[] args)
{
    Server app = new Server();
    System.exit(app.main("Server", args));
}
}
```

这段代码导入了 `Filesystem package` 的内容。这样，我们就不必总是通过 `Filesystem.` 前缀来使用进行完全限定的标识符了。

源码接下来的部分是 `Server` 类的定义，这个类派生自 `Ice.Application`，在其 `run` 方法中含有主应用逻辑。这里的大部分代码都是我们先前见过的公式化代码：我们创建对象适配器，然后到最后，激活对象适配器，并调用 `waitForShutdown`。

有意思的代码是适配器创建代码：服务器实例化我们的文件系统的几个节点，创建了图 13.1 所示的结构。

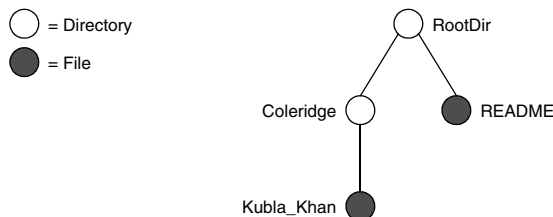


图 13.1. 一个小文件系统

我们很快就会看到，我们的目录和文件的 `servant` 的类型分别是 `DirectoryI` 和 `FileI`。这两种类型的 `servant` 的构造器都接受两个参数：要创建的目录或文件的名称，以及指向父目录的 `servant` 的引用（对于没有父目录的根目录，我们会传递 `null` 父引用）。因此，下面的语句

```
DirectoryI root = new DirectoryI("/", null);
```

会创建根目录，名字是 `"/"`，没有父目录。

下面的代码建立图 13.1 中的结构：

```
// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI("/", null);

// Create a file "README" in the root directory
//
File file = new FileI("README", root);
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}

// Create a directory "Coleridge" in the root directory
//
DirectoryI coleridge
    = new DirectoryI("Coleridge", root);
```

```
// Create a file "Kubla_Khan" in the Coleridge directory
//
file = new FileI("Kubla_Khan", coleridge);
text = new String[]{ "In Xanadu did Kubla Khan",
                     "A stately pleasure-dome decree:",
                     "Where Alph, the sacred river, ran",
                     "Through caverns measureless to man",
                     "Down to a sunless sea." };

try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}
```

我们首先创建根目录，并在根目录中创建 README 文件（注意，当我们创建类型为 FileI 的新节点时，我们传递的父引用是指向根目录的引用）。

下一步是用文本填充文件：

```
String[] text;
text = new String[] {
    "This file system contains a collection of poetry."
};
try {
    file.write(text, null);
} catch (GenericError e) {
    System.err.println(e.reason);
}
```

回想一下 8.7.3 节的内容，在缺省情况下，Slice 序列映射到 Java 数组。Slice 类型 Lines 就是串数组；我们初始化 text 数组，让它包含一个元素，从而给我们的 README 文件增加了一行文本。

最后，我们调用我们的 FileI servant 的 Slice write 操作：

```
file.write(text, null);
```

这条语句很有意思：服务器代码祈用它自己的 servant 上的操作。因为这个调用是通过指向 servant 的引用（类型是 FileI）、而不是代理（类型是 FilePrx）进行的，Ice run time 甚至不知道发生了这个调用——像这样对 servant 的直接调用，完全不会由 Ice run time 居中协调，而是会作为平常的 Java 函数调用进行分派。

余下的代码以类似的方式，创建叫作 Coleridge 的子目录，并在该目录中创建叫作 Kubla_Khan 的文件，从而完成了图 13.1 中的结构。

13.2.2 FileI Servant 类

我们的 FileI servant 类具有这样的基本结构：

```
public class FileI extends _FileDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private String[] _lines;
}
```

这个类有一些数据成员：

- `_adapter`
这个静态成员存储的是一个引用，指向我们在服务器中使用的唯一一个对象适配器。
- `_name`
这个成员存储的是 servant 所体现的文件的名字。
- `_parent`
这个成员存储的是一个引用，指向文件的父目录的 servant。
- `_lines`
这个成员存放文件的内容。

`_name` 和 `_parent` 数据成员由构造器初始化：

```
public
FileI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    assert(_parent != null);

    // Create an identity
    //
    Ice.Identity myID
        = Ice.Util.stringToIdentity(Ice.Util.generateUUID());

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);
}
```

```
// Create a proxy for the new node and
// add it as a child to the parent
//
NodePrx thisNode
    = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
_parent.addChild(thisNode);
}
```

在初始化 `_name` 和 `_parent` 成员之后，这段代码核实指向父目录的引用不是 `null`，因为每个文件都必须有父目录。构造器随即调用 `Ice.Util.generateUUID`，为这个文件生成一个标识，并调用 `ObjectAdapter.add`，把自己增加到 `servant` 映射表中。最后，构造器为这个文件创建代理，并调用它的父目录的 `addChild` 方法。`addChild` 是一个助手函数，子目录或文件可以调用它、把自己增加到父目录的后代节点列表中。我们将在第 305 页看到这个函数的实现。

`FileI` 类的其他方法实现了我们在 `Node` 和 `File` 这两个 `Slice` 接口中定义的操作：

```
// Slice Node::name() operation

public String
name(Ice.Current current)
{
    return _name;
}

// Slice File::read() operation

public String[]
read(Ice.Current current)
{
    return _lines;
}

// Slice File::write() operation

public void
write(String[] text, Ice.Current current)
    throws GenericError
{
    _lines = text;
}
```

`name` 方法继承自生成的 `Node` 接口（这个接口是 `_FileDisp` 类的基接口，而 `FileI` 派生自 `_FileDisp` 类）。它简单地返回 `_name` 成员的值。

`read` 和 `write` 方法继承自生成的 `File` 接口（这个接口是 `_FileDisp` 类的基接口，而 `FileI` 派生自 `_FileDisp` 类）。它们简单地返回和设置 `_lines` 成员。

13.2.3 DirectoryI Servant 类

`DirectoryI` 类具有这样的基本结构：

```
package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    // Constructor and operations here...

    public static Ice.ObjectAdapter _adapter;
    private String _name;
    private DirectoryI _parent;
    private java.util.ArrayList _contents
        = new java.util.ArrayList();
}
```

和 `FileI` 类的情况一样，我们拥有一些数据成员，用于存储对象适配器、名字，以及父目录（对于根目录，`_parent` 成员存放的是 `null` 引用）。此外，我们还有一个 `_contents` 数据成员，存储的是子目录列表。这些数据成员都由构造器初始化：

```
public
DirectoryI(String name, DirectoryI parent)
{
    _name = name;
    _parent = parent;

    // Create an identity. The parent has the fixed identity "/"
    //
    Ice.Identity myID
        = Ice.Util.stringToIdentity(_parent != null ?
                                    Ice.Util.generateUUID() : "RootDir");

    // Add the identity to the object adapter
    //
    _adapter.add(this, myID);
}
```



```
// Create a proxy for the new node and add it as a
// child to the parent
//
NodePrx thisNode
    = NodePrxHelper.uncheckedCast(_adapter.createProxy(myID));
if (_parent != null)
    _parent.addChild(thisNode);
}
```

这个构造器调用 `Ice.Util.generateUUID`，为新目录创建标识（对于根目录，我们使用固定的 "RootDir" 标识）。`servant` 调用 `ObjectAdapter.add`，把自己增加到 `servant` 映射表中，然后创建一个指向自身的引用，传给 `addChild` 助手函数。

`addChild` 简单地把传入的引用增加到 `_contents` 列表：

```
void
addChild(NodePrx child)
{
    _contents.add(child);
}
```

剩下的两个操作 `name` 和 `list` 非常简单：

```
public String
name(Ice.Current current)
{
    return _name;
}

// Slice Directory::list() operation

public NodePrx[]
list(Ice.Current current)
{
    NodePrx[] result = new NodePrx[_contents.size()];
    _contents.toArray(result);
    return result;
}
```

注意，`_contents` 的类型是 `java.util.ArrayList`，这对于 `addChild` 方法的实现而言很方便。但为了从 `list` 操作中返回它，我们需要把列表转换成 Java 数组。

13.3 总结

这一章说明了怎样为我们在第 5 章定义的文件系统实现一个完整的服务器。注意，这个服务器引人注目的特点是，它没有多少与分布处理有关的代码：服务器的大部分代码都只是应用逻辑，如果你编写一个非分布式的版本，同样也要编写这样的代码。这又是 Ice 的重要好处之一：应用代码不需要考虑分布事务，所以你可以专注于应用逻辑、而不是网络基础设施的开发。

注意，按照现在的情况，我们在这里给出的服务器代码并不十分正确：如果有两个客户分别通过不同的线程、同时访问同一个文件，一个线程可能在读取 `_lines` 数据成员，而另一个线程在更新它。显然，如果发生这样的情况，我们可能会写入或返回垃圾数据，或者更糟糕，使服务器崩溃。要让 `read` 和 `write` 操作成为线程安全的其实很容易，只要一个数据成员和两行代码就足够了。我们将在第 15 章讨论怎样编写线程安全的 `servant` 实现。

第 14 章

Ice 属性与配置

14.1 本章综述

Ice 使用了一种配置机制，允许你控制你的 Ice 应用在运行时的许多行为，比如最大消息尺寸、线程数、是否产生网络跟踪消息。这种机制不仅能用于配置 Ice，你还可以用它来给你自己的应用提供配置参数。它的 API 非常小，使用起来很简单，但又灵活得足以满足大多数应用的需要。

14.2 节到 14.7 节描述这种配置机制的基本要素，并解释怎样通过配置文件和命令行选项来配置 Ice。14.8 节说明怎样创建你的应用专用的属性，以及怎样在程序中访问它们的值。

14.2 属性

Ice 及其各子系统是通过属性（property）配置的。一个属性就是一个名-值对（name-value pair），例如：

```
Ice.UDP.SndSize=65535
```

在这个例子中，属性名是 Ice.UDP.SndSize，属性值是 65535。

在 Appendix C 中，你可以找到用于配置 Ice 的属性的完整列表。

14.2.1 属性范畴

按照惯例，Ice 属性使用的是下面的命名方案：

`<application>.<category>[.<sub-category>]`

注意，子范畴是可选的，并不是所有 Ice 属性都会使用它。

这个由两个、或三个部分组成的命名方案只是一种惯例——如果你用属性配置你自己的应用，你可以使用具有任意多的范畴的属性名。

14.2.2 保留的前缀

Ice 保留具有以下前缀的属性：Ice、IceBox、IcePack、IcePatch、IceSSL、IceStorm、Freeze，以及 Glacier。你不能使用以这些前缀起头的属性来配置你自己的应用。

14.2.3 属性语法

属性名由任何多个非空白字符（除了 # 和 = 字符）组成。例如，下面的属性名是有效的：

```
foo
Foo
foo.bar
.
```

注意，属性名中的句点并无特殊含义（句点用于使属性名更可读，属性解析器并不会对它进行特殊处理）。

下面的属性名是无效的：

```
foo bar      Illegal white space
foo=bar      Illegal =
foo#bar      Illegal #
```

14.2.4 值语法

属性值由任意多个字符组成。起头和结尾的空白字符会被忽略。属性值不能包含 # 字符。下面的属性值是合法的：

```
65535
yes
This is a = property value.
../../config
```

下面的属性值是非法的：

```
foo # bar      Property values cannot contain a # character
```

14.3 配置文件

属性通常在配置文件中设置。配置文件含有一些名 - 值对，每一对都在单独的行上。空行及完全由空白字符组成的行会被忽略。# 字符的后面是注释，直到当前行的末尾。

下面是一个简单的配置文件：

```
# Example config file for Ice

Ice.MessageSizeMax = 2048      # Largest message size is 2MB
Ice.Trace.Network=3           # Highest level of tracing for network
Ice.Trace.Protocol=           # Disable protocol tracing
```

如果你多次设置同一属性，最后一次设置会生效，取代先前的设置。注意，如果你把空值赋给属性，就会清除该属性。

对于 C++，Ice 会在你创建通信器时读取配置文件的内容。在缺省情况下，配置文件名由 **ICE_CONFIG** 环境变量的内容决定¹。你可以把这个变量设成配置文件的相对或绝对路径名，例如：

```
$ ICE_CONFIG=/opt/Ice/default_config
$ export ICE_CONFIG
$ ./server
```

这使得服务器从配置文件 /opt/Ice/default_config 中读取它的属性设置。

14.4 在命令行上设置属性

除了在配置文件中设置属性，你还可以在命令行上设置属性，例如：

```
$ ./server --Ice.UDP.SndSize=65535 --IceSSL.Trace.Security=2
```

任何以 -- 起头、并且后跟某个保留前缀（参见第 308 页）的命令行选项，都会在你创建通信器时被读取、并转换成属性设置。命令行上的属性

1. Java run time 不会读取这个环境变量：在使用 Java 时，你必须用 --Ice.Config 属性设置配置文件的名字（参见 14.5 节）。

设置会覆盖配置文件中的设置。如果你在同一命令行上多次设置同一属性，最后的设置会覆盖前面的任何设置。

为方便起见，任何没有明确设置的属性都会被设置为值 1。例如，

```
$ ./server --Ice.Trace.Protocol
```

等价于

```
$ ./server --Ice.Trace.Protocol=1
```

注意，这个特性只适用于在命令行上设置的属性，而不适用于在配置文件中设置的属性。

在命令行上，你还可以这样清除属性：

```
$ ./server --Ice.Trace.Protocol=
```

和在配置文件中设置的属性一样，把空值赋给属性会清除该属性。

14.5 Ice.Config 属性

对 Ice run time 而言，Ice.Config 属性有着特殊含义：它确定用于读取属性设置的配置文件的路径名。例如：

```
$ ./server --Ice.Config=/usr/local/filesystem/config
```

这使得配置设置从 /usr/local/filesystem/config 配置文件中被读出。

对于 C++，`--Ice.Config` 命令行选项会覆盖 `ICE_CONFIG` 环境变量的任何设置，也就是说，如果你设置了 `ICE_CONFIG` 环境变量，同时又使用 `--Ice.Config` 命令行选项，`ICE_CONFIG` 环境变量所指定的配置文件会被忽略。

如果你在使用 `--Ice.Config` 命令行选项的同时还进行了其他属性设置，命令行上的设置会覆盖配置文件中的设置。例如：

```
$ ./server --Ice.Config=/usr/local/filesystem/config \  
> --Ice.MessageSizeMax=4096
```

不管在 /usr/local/filesystem/config 中作了何种设置，这个设置都会把 Ice.MessageSizeMax 属性的值设为 4096。命令行上放置的 `--Ice.Config` 选项对这一优先级没有影响。例如，下面的命令与前面的命令是等价的：

```
$ ./server --Ice.MessageSizeMax=4096 \  
> --Ice.Config=/usr/local/filesystem/config
```

配置文件中的 `Ice.Config` 属性设置会被忽略，也就是说，你只能在命令行上设置 `Ice.Config`。

如果你多次使用 `--Ice.Config` 选项，只有最后的选项设置被使用，前面的会被忽略。例如：

```
$ ./server --Ice.Config=file1 --Ice.Config=file2
```

等价于使用：

```
$ ./server --Ice.Config=file2
```

通过指定用逗号分隔的配置文件名列表，你可以使用多个配置文件。例如：

```
$ ./server --Ice.Config=/usr/local/filesystem/config,./config
```

属性设置会从 `/usr/local/filesystem/config` 中读取，然后是当前目录中的 `config` 文件中的任何设置；`./config` 中的设置会覆盖 `/usr/local/filesystem/config` 中的设置。对于 C++，这种机制还适用于通过 `ICE_CONFIG` 环境变量指定的配置文件。

14.6 命令行解析与初始化

当你调用 `Ice::initialize` (C++) 或 `Ice.Util.initialize` (Java)、初始化 Ice run time 时，你要传一个参数向量给初始化调用。

对于 C++，`Ice::initialize` 接受的参数是一个指向 `argc` 的 C++ 引用：

```
namespace Ice {  
    CommunicatorPtr initialize(int &argc, char *argv[]);  
};
```

`Ice::initialize` 解析参数向量，并相应地对其属性设置进行初始化。此外，它还把属性设置参数从 `argv` 中移除。例如，假定我们祈用一个服务器：

```
$ ./server --myoption --Ice.Config=config -x a \  
--Ice.Trace.Network=3 -y opt file
```

一开始，`argc` 的值是 9，而 `argv` 有十个元素：前九个元素含有程序名和各个参数，最后一个元素 `argv[argc]` 含有一个 `null` 指针（这是 ISO

C++ 标准的要求)。当 `Ice::initialize` 返回时, `argc` 的值是 7, 而 `argv` 含有以下元素:

```
./server
--myoption
-x
a
-y
opt
file
0                # Terminating null pointer
```

这意味着, 你应该在解析命令行、处理应用专用的参数之前, 先初始化 Ice run time。这样, 与 Ice 有关的选项就会从参数向量中去除, 你也就无需再显式地跳过它们了。如果你使用了 `Ice::Application` 助手类 (参见 10.3.1 节), `run` 成员函数收到的参数也是清理过的参数向量。

对于 Java, `Ice.Util.initialize` 是重载的。其型构是:

```
package Ice;
public final class Util {

    public static Communicator
    initialize(String[] args);

    public static Communicator
    initialize(StringSeqHolder args);

    // ...
}
```

第一个版本没有为你去除与 Ice 有关的选项, 所以, 如果你使用该版本, 你需要忽略以保留前缀

(`--Ice`、`--IceBox`、`--IcePack`、`--IcePatch`、`--IceSSL`、`--IceStorm`、`--Freeze`, 以及 `--Glacier`) 起头的选项。第二个版本的行为方式与 C++ 版本类似, 会在传入的参数向量中去除与 Ice 有关的选项。

如果你使用了 `Ice.Application` 助手类 (参见 12.4.1 节), `run` 方法收到的参数是清理过的参数向量。

14.7 Ice.ProgramName 属性

在 C++ 中, `initialize` 把 `Ice.ProgramName` 属性设成当前程序的名字 (`argv[0]`)。Ice 把程序名用于日志消息。你的应用代码可以读取

这个属性，把它用于类似的目的，例如，用在诊断或跟踪消息中（关于怎样在你的程序中访问属性值，参见 14.8.1 节）。

即使 `Ice.ProgramName` 已经为你做了初始化，你仍然可以在配置文件中重新定义它的值，也可以在命令行上进行设置。

在 Java 中，程序名没有作为参数向量的一部分提供给你——如果你想让程序名出现在 Ice 日志消息中，你必须在初始化 Java 通信器之前设置 `Ice.ProgramName`。

14.8 在程序中使用属性

Ice 属性机制不仅可用于配置 Ice，你还可以把它用作你自己的应用的配置机制。你可以用同样的配置文件和命令行机制来设置应用专用的属性。例如，我们可以引入一个属性，控制我们的文件系统应用的最大文件尺寸：

```
# Configuration file for file system application

Filesystem.MaxFileSize=1024    # Max file size in kB
```

Ice run time 像存储其他任何属性一样存储 `Filesystem.MaxFileSize` 属性，并且让你能通过 `Properties` 接口访问它。

要在你的程序里访问属性值，你需要调用 `getProperties`，获取通信器的各个属性：

```
module Ice {

    local interface Properties; // Forward declaration

    local interface Communicator {

        Properties getProperties();

        // ...
    };
};
```

`Properties` 接口提供了用于读写属性设置的方法：

```
module Ice {
    local dictionary<string, string> PropertyDict;

    local interface Properties {
```

```

    string getProperty(string key);
    string getPropertyWithDefault(string key, string value);
    int getPropertyAsInt(string key);
    int getPropertyAsIntWithDefault(string key, int value);
    PropertyDict getPropertyForPrefix(string prefix);

    void setProperty(string key, string value);

    StringSeq getCommandLineOptions();
    StringSeq parseCommandLineOptions(string prefix,
                                      StringSeq options);
    StringSeq parseIceCommandLineOptions(StringSeq options);

    void load(string file);

    Properties clone();
};
};

```

14.8.1 读取属性

用于读取属性值的操作的行为是：

- **getProperty**
这个操作返回指定属性的值。如果该属性没有设置，操作返回空串。
- **getPropertyWithDefault**
这个操作返回指定属性的值。如果该属性没有设置，操作返回你提供的缺省值。
- **getPropertyAsInt**
这个操作把指定属性的值作为整数返回。如果属性没有设置，或者包含的是不能解析成整数的串，操作返回零。
- **getPropertyAsIntWithDefault**
这个操作把指定属性的值作为整数返回。如果属性没有设置，或者包含的是不能解析成整数的串，操作返回你提供的缺省值。
- **getPropertyForPrefix**
这个操作把以指定前缀起头的所有属性、作为 **PropertyDict** 类型的词典返回。如果你想要提取指定的子系统的各个属性，这个操作很有用。例如，

```
getPropertiesForPrefix("Filesystem")
```

返回以前缀 `Filesystem` 起头的所有属性，比如 `Filesystem.MaxFileSize`。随后，你可以用通常的词典查找操作，从返回的词典中提取你感兴趣的属性。

有了这些查找操作，使用应用专用的属性现在变得简单了，你需要做的只是像平常一样初始化通信器、获得对通信器属性的访问，然后检查你需要的属性值。例如（在 C++ 中）：

```
// ...

Ice::CommunicatorPtr ic;

// ...

ic = Ice::initialize(argc, argv);

// Get the maximum file size.
//
Ice::PropertiesPtr props = ic->getProperties();
Ice::Int maxSize
    = props->getPropertyAsIntWithDefault("Filesystem.MaxFileSize",
                                         1024);

// ...
```

假定你创建了一个配置文件，用于设置 `Filesystem.MaxFileSize` 属性（并相应地设置了 **ICE_CONFIG** 变量或 **--Ice.Config** 选项），你的应用 将会获得所配置的属性值。

14.8.2 设置属性

`setProperty` 操作把某个属性设成指定的值（只要把属性设成空串，你就可以清除它）。只有在你调用 `initialize` 之前，这个操作才有用。这是因为，**Ice run time**（通常）只在你调用 `initialize` 时，对属性值进行一次读取。在你初始化通信器之后，**Ice run time** 不保证它会关注属性值的变化。当然，这带来了这样一个问题：你怎样才能设置属性值、并让通信器认可它？

为了允许你在初始化通信器之前设置属性，**Ice run time** 提供了一个重载的助手函数，叫作 `getDefaultProperties`。在 C++ 里，这个函数处在 `Ice` 名字空间中：

```
namespace Ice {
    PropertiesPtr getDefaultProperties();
    PropertiesPtr getDefaultProperties(int &argc, char *argv[]);
};
```

`getDefaultProperties` 返回一个进行了初始化的属性集，初始化所用的或是 **ICE_CONFIG** 环境变量所指定的配置文件的内容，或者，在你调用的是第二个版本的情况下，是 **ICE_CONFIG** 环境变量或 **--Ice.Config** 选项所指定的配置文件的内容——同时，在命令行上指定的属性设置会覆盖配置文件中的设置。

在 Java 里，该函数是 `Util` 类的一个静态方法，这个类处在 `Ice` package 中：

```
package Ice;

public final class Util {
    public static Properties getDefaultProperties();
    public static Properties getDefaultProperties(
        StringSeqHolder args);
    // ...
}
```

第一个版本返回的是一个空属性集，第二个版本返回的是通过 **--Ice.Config** 选项进行了初始化的属性集，同时，命令行上的其他任何属性设置都会覆盖配置文件中的设置。

如果不管配置文件中的设置是什么，你都想要确保某个属性被设成特定的值，`getDefaultProperties` 会很有用。例如：

```
// Get the initialized property set.
//
Ice::PropertiesPtr props = Ice::getDefaultProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);

// ...
```

在 Java 里，等价的代码是：

```
Ice.StringSeqHolder argsH = new Ice.StringSeqHolder(args);
Ice.Properties properties = Ice.Util.getDefaultProperties(argsH);
properties.setProperty("Ice.Warn.Connections", "0");
communicator = Ice.Util.initialize(argsH);
```

注意，这里有一个额外的步骤：我们首先把参数数组转换成初始化过的 `StringSeqHolder`。这是必需的，这样 `getDefaultProperties` 才能够去除 Ice 专用的设置。以这样的方式，我们首先获取一个初始化过的属性集，然后覆盖两个跟踪属性的设置，再把去除了 Ice 专用设置的参数向量传给 `initialize`²。

14.8.3 解析属性

`Properties` 接口提供了三个用于转换和解析 属性的操作：

- `getCommandLineOptions`

这个操作把初始化过的属性集转换成等价的命令行选项序列。例如，如果你把 `Filesystem.MaxFileSize` 属性设成 1024，并调用 `getCommandLineOptions`，这个设置就会作为 `"Filesystem.MaxFileSize=1024"` 串返回。这个操作可用于诊断目的，例如，把所有属性的设置倾卸到日志设施中（参见 16.14 节），也可以在派生新进程时，通过这个操作、使用与当前进程相同的属性设置。

- `parseCommandLineOptions`

这个操作检查传入的参数向量，查找具有指定前缀的命令行选项。任何与该前缀匹配的选项都会被转换成属性设置（也就是说，它们会初始化对应的属性）。这个操作返回一个参数向量，其中含有所有没有被转换的选项（也就是说，那些与前缀不匹配的选项）。

因为 `parseCommandLineOptions` 期望的参数是串序列，而 C++ 程序习惯于处理 `argc` 和 `argv`，Ice 提供了两个实用函数，用于把 `argc/argv` 向量转换成串序列，或进行相反的转变：

```
namespace Ice {

    StringSeq argsToStringSeq(int argc, char* argv[]);
```

2. 回想一下第 312 页的内容，如果我们把 `args` 直接传给 `getDefaultProperties`（没有先把 `args` 转换成 `StringSeqHolder`），`getDefaultProperties` 就不会去除与 Ice 有关的选项，所以两个跟踪属性的设置就不会起任何作用，因为原来传给 `initialize` 的参数数组会覆盖显式的设置。

```
void stringSeqToArgs(const StringSeq& args,
                    int& argc, char* argv[]);
}
```

如果你想要在命令行上设置应用专用的属性，你需要使用 `parseCommandLineOptions`（以及上面的两个实用函数）。例如，要想在命令行上设置 `--Filesystem.MaxFileSize` 选项，我们需要这样初始化我们的程序：

```
int
main(int argc, char * argv[])
{
    // Get the initialized property set.
    //
    Ice::PropertiesPtr props = Ice::getDefaultProperties();

    // Convert argc/argv to a string sequence.
    //
    Ice::StringSeq args = Ice::argsToStringSeq(argc, argv);

    // Strip out all options beginning with --Filesystem.
    //
    args = props->parseCommandLineOptions("Filesystem", args);

    // args now contains only those options that were not
    // stripped. Any options beginning with --Filesystem have
    // been converted to properties.

    // Convert remaining arguments back to argc/argv vector.
    //
    Ice::stringSeqToArgs(args, argc, argv);

    // Initialize communicator.
    //
    Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);

    // At this point, argc/argv only contain options that
    // set neither an Ice property nor a Filesystem property,
    // so we can parse these options as usual.
    //
    // ...
}
```

```
}
```

使用这段代码，任何以 **--Filesystem** 起头的选项都会被转换成属性，并且像往常一样，可以通过属性查找操作访问。随后，对 `initialize` 的访问会移除任何 Ice 专用的命令行选项，这样，一旦通信器被创建，`argc/argv` 所包含的将只是与文件系统或 Ice 属性设置无关的选项和参数。

- `parseIceCommandLineOptions`

这个操作的行为与 `parseCommandLineOptions` 类似，但它会从参数向量中移除保留的 Ice 专用选项（参见 14.2.2 节）。Ice run time 会在内部使用它，用于在 `initialize` 中解析 Ice 专用的选项。

14.8.4 实用操作

`Properties` 接口提供了两个实用操作：

- `clone`

这个操作制作一个现有属性集的副本。副本包含的属性及值和原来的属性集完全一样。如果你需要处理多个属性集，这个操作很有用（参见 14.8.5 节）。

- `load`

这个操作的参数是一个配置文件的路径名，它会根据这个文件初始化属性集。如果无法读取指定的文件（例如，因为它不存在，或者调用者没有读取权限），这个操作就会抛出 `SyscallException`。如果你需要处理多个属性集的不同配置文件，这个操作很有用（参见 14.8.5 节）。

14.8.5 处理多个属性集

有时，你可能会需要使用多个通信器，因而需要使用多个属性集。在缺省情况下，`initialize` 使用的是 `getDefaultProperties` 返回的属性集。这意味着，如果你通过 `initialize` 创建两个通信器，它们是用同样的属性集创建的：

```
// Create a communicator.
//
Ice::CommunicatorPtr ic1 = initialize(argc, argv);

// ...
```

```
// Create another communicator.
Ice::CommunicatorPtr ic2 = initialize(argc, argv);

// ic1 and ic2 are now initialized with the same property set.
```

在内部，`initialize` 使用的是单个静态属性集。这意味着，如果你想要使用两个具有不同属性集的通信器，下面的做法不一定可行：

```
// Create a communicator.
//
Ice::CommunicatorPtr ic1 = initialize(argc, argv);

// Set a property.
//
Ice::PropertiesPtr props = getDefaultProperties();
props->setProperty("Ice.Trace.Network", "1");

// Create another communicator.
Ice::CommunicatorPtr ic2 = initialize(argc, argv);

// ic1 and ic2 both may have tracing enabled!
```

这里的问题是，两个通信器在内部共享了 `getDefaultProperties` 返回的属性集。就这个例子而言，两个通信器最后都会启用网络跟踪功能。但是，特定属性的设置是会影响两个通信器，还是只影响第二个通信器，这取决于 **Ice run time** 是否在内部缓存了该属性值：如果值做了缓存，变化了的设置只会影响第二个通信器；如果值没有缓存，两个通信器都会受影响。实际上，前面的代码的行为是不确定的。

为了让你能使用独立的属性集，**Ice** 提供了用于创建新属性集的实用函数。对于 **C++**，这些函数处在 **Ice** 名字空间中：

```
namespace Ice {
    PropertiesPtr createProperties();
    PropertiesPtr createProperties(int &argc, char *argv[]);
}
```

第一个版本的 `createProperties` 创建一个属性集，用 **ICE_CONFIG** 环境变量所指定的配置文件的内容进行初始化。第二个版本也创建一个属性集，用 **ICE_CONFIG** 环境变量、或 **--Ice.Config** 命令行选项所指定的配置文件的内容进行初始化。

对于 **Java**，这两个函数是在 **Util** 类中提供的：

```
package Ice;
public final class Util {

    public static Properties createProperties();
```



```

        public static Properties createProperties(
                                StringSeqHolder args);

        // ...
    }

```

第一个版本的 `createProperties` 创建一个空属性集。第二个版本创建一个属性集，用 **--Ice.Config** 所指定的配置文件的内容进行初始化。

Ice 还提供了另外一个用于初始化通信器的助手函数：`initializeWithProperties`。在 C++ 里，其声明是：

```

namespace Ice {
    initializeWithProperties(int &argc, char *argv[],
                           const PropertiesPtr &props);
}

```

在 Java 里，`initializeWithProperties` 是 `Ice.Util` 类的一部分：

```

package Ice;
public final class Util {

    public static Communicator
        initializeWithProperties(String[] args, Properties properties)
    ;

    // ...
}

```

你可以看到，`initializeWithProperties` 允许你显式地传入一个属性集，并优先于缺省属性集使用你传入的属性集。这使得你能够为不同的通信器创建单独的属性集，例如：

```

// Create a property set for the first communicator.
//
Ice::PropertiesPtr props1 = createProperties();

// Make sure that network tracing is off.
//
props1->setProperty("Ice.Trace.Network", "0");

// Initialize a communicator with this property set.
//
Ice::CommunicatorPtr ic1
    = Ice::initializeWithProperties(argc, argv, props1);

```

```
// Create a property set for a second communicator.
//
Ice::PropertiesPtr props2 = createProperties();

// Make sure that network tracing is on.
//
props2->setProperty("Ice.Trace.Network", "1");

// Initialize a communicator with this property set.
//
Ice::CommunicatorPtr ic2
    = Ice::initializeWithProperties(argc, argv, props2);
```

上面的代码例子能正确地用不同的属性配置两个通信器，避免了第 320 页上的不正确的例子的问题。如果你想要禁止命令行选项覆盖属性设置，你可以把一个虚设的参数向量传给 `initializeWithProperties`。

如果你想创建的多个通信器只有少量属性不同，一种可能的做法是使用属性集的 `clone` 操作。下面的例子的效果与前面的例子一样。但是，第一个通信器是用缺省属性集创建的，而第二个通信器是用缺省属性值的副本创建的：

```
// Get the default property set for the first communicator.
//
Ice::PropertiesPtr props1 = getDefaultProperties(argc, argv);

// Make sure that network tracing is off.
//
props1->setProperty("Ice.Trace.Network", "0");

// Initialize a communicator with the default property set.
//
Ice::CommunicatorPtr ic1 = Ice::initialize(argv, argc);

// Make a copy of the default property set.
//
Ice::PropertiesPtr props2 = props1->clone();

// Make sure that network tracing is on.
//
props2->setProperty("Ice.Trace.Network", "1");

// Initialize a communicator with this property set.
//
Ice::CommunicatorPtr ic2
    = Ice::initializeWithProperties(argc, argv, props2);
```

如果你想要为不同的通信器使用不同的配置文件，你可以使用相似的代码，但调用 `load` 操作（参见 14.8.4 节）、初始化你用 `createProperties` 创建的不同的属性集。

注意，你在任何时候都可以调用 `Communicator::getProperties`，获得特定通信器的属性集。

14.9 总结

Ice 属性机制提供了一种简单的配置 Ice 的途径，你可以在配置文件中、或在命令行上设置属性。这也适用于你自己的应用：你可以轻松地使用 `Properties` 接口，访问你为自己的需要而创建的、应用专用的属性。用于访问属性值的 API 小而简单，所以要在运行时用它获取属性值很容易；这个 API 还很灵活，如果有需要，它能让你使用多个不同的属性集和配置文件。

第 15 章

C++ 线程与并发

15.1 本章综述

这一章介绍 Ice 提供的 C++ 线程和信号处理抽象（请注意，Ice 没有为 Java 提供等价的线程 API，而是使用了 Java 内建的线程和同步设施）。我们将简要地概述 Ice 使用的线程模型，并说明怎样使用各种可用的同步原语（互斥体和监控器）。随后我们涵盖线程的创建、控制，以及销毁。我们将给出一个简单的例子，以此结束对线程的讨论；这个例子说明了怎样创建一个线程安全的生产者 - 消费者多线程应用。最后，我们将介绍一个可移植的抽象，用于处理信号及与信号类似的事件。

15.2 引言

Ice 天生就是一个多线程平台。在 Ice 中，并没有单线程服务器这样的东西。所以，你必须考虑各种并发问题：如果在一个线程读取某个数据结构的同时，另一个线程正在更新同一个数据结构，除非你用适当的锁保护这个数据结构，否则就会发生严重的混乱。

随操作系统不同，线程和并发控制也会发生很大的变化。为了让线程编程变得更轻松，并提高可移植性，Ice 提供了一个简单的线程抽象层，藉此，不管底层是什么样的平台，你都可以编写可移植的源码。在这一章，我们将详尽地考察 Ice 的线程和并发控制机制。

请注意，我们假定你已经熟悉轻量级线程和并发控制（在 [8] 中你可以找到对线程编程的非常好的讨论）。

15.3 Ice 线程模型

Ice 服务器天生是多线程的。服务器端 `run time` 维护有一个线程池，用于处理到来的请求。通过领导者 - 跟随者（`leader-follower`）线程模型 [17]，客户发来的每个操作祈用都会在其自己的线程中被分派。在服务器中，如果所有线程都在执行操作祈用，耗尽了线程池，随后到来的客户请求就会“透明地”延迟处理，直到服务器中的某个操作完成、放弃其线程；这个线程随即被用于分派下一个待处理的客户请求。

通过 `Ice.ThreadPool.Server.Size` 属性，可以配置服务器中的线程池的尺寸（参见 Appendix C）。这个属性的缺省值是 1。

多线程意味着，来自客户的多个祈用可以在服务器中并发执行。事实上，在同一个 `servant` 中，以及在同一 `servant` 的同一个操作中，都可以有多个请求在并行执行。因此，如果在操作实现中，涉及到对非栈存储的操纵（比如 `servant` 的成员变量、全局变量，或静态变量），你必须对数据访问进行互锁，以防止数据损坏。Ice 线程库提供了许多同步原语，比如简单互斥体、读写锁，以及监控器。这些同步原语允许你实现不同粒度的并发控制。此外，Ice 还允许你创建你自己的线程。例如，你可以创建单独的线程，响应 GUI 事件或其他异步事件。

15.4 线程库综述

Ice 线程库提供了这样一些与线程有关的抽象：

- 互斥体
- 递归互斥体
- 读写递归互斥体
- 监控器
- 一个线程抽象，允许你创建、控制、销毁线程。

所有的线程 API 都是 `IceUtil` 名字空间的一部分。

15.5 互斥体

IceUtil::Mutex 类（在 IceUtil/Mutex.h 中定义）和 IceUtil::StaticMutex（在 IceUtil/StaticMutex.h 中定义）提供了简单的非递归互斥机制：

```
namespace IceUtil {

    class Mutex {
    public:
        Mutex();
        ~Mutex();
        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };

    struct StaticMutex {
        void lock() const;
        bool tryLock() const;
        void unlock() const;

        typedef LockT<StaticMutex> Lock;
        typedef TryLockT<StaticMutex> TryLock;
    };
}
```

IceUtil::Mutex 和 IceUtil::StaticMutex 具有相同的行为，但 IceUtil::StaticMutex 被实现为简单的数据结构¹，所以其实例可以静态声明，并在编译过程中初始化。如下所示：

```
static IceUtil::StaticMutex myStaticMutex =
    ICE_STATIC_MUTEX_INITIALIZER;
```

预处理器宏 ICE_STATIC_MUTEX_INITIALIZER 的用途是正确地初始化 IceUtil::StaticMutex 的各成员。IceUtil::StaticMutex 的实例永远不会被销毁。

1. 用 ISO C++ 的术语说，StaticMutex 是“plain old data”（POD）。

而 `IceUtil::Mutex` 被实现为类，因此会由其构造器初始化，由其析构器销毁。

这两个类的成员函数的行为如下：

- `lock`

`lock` 函数尝试获取互斥体。如果互斥体已经锁住，它就会挂起发出调用的线程（calling thread），直到互斥体变得可用为止。一旦发出调用的线程获得了互斥体，调用就会立即返回。

- `tryLock`

`tryLock` 函数尝试获取互斥体。如果互斥体可用，互斥体就会锁住，而调用就会返回 `true`。如果其他线程锁住了互斥体，调用返回 `false`。

- `unlock`

`unlock` 函数解除互斥体的加锁。

请注意，`IceUtil::Mutex` 和 `IceUtil::StaticMutex` 是非递归的互斥体实现。这意味着，你必须遵守以下规则：

- 不要从同一个线程多次针对同一个互斥体调用 `lock`。这些互斥体不是递归的，所以，如果互斥体的所有者试图第二次锁住它，其行为将是不确定的。
- 除非发出调用的线程持有某个互斥体，否则不要针对该互斥体调用 `unlock`。如果目前没有线程持有某个互斥体，或者持有它的是另外的线程，针对它调用 `unlock` 就会导致不确定的行为。

15.5.1 文件系统应用的线程安全的文件访问

回想一下，在 11.2.3 节中，我们的文件系统服务器的 `read` 和 `write` 操作的实现不是线程安全的：

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    return _lines;          // Not thread safe!
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    _lines = text;          // Not thread safe!
}
```


这些代码的问题在于，如果我们收到对 `read` 和 `write` 的并发祈用，一个线程就会向 `_lines` 向量赋值，而另一个线程正读取该向量。这样的并发数据访问产生的结果是不确定的；要避免发生这样的问题，我们需要通过互斥体，使对 `_lines` 成员的访问序列化。我们可以让这个互斥体成为 `FileI` 类的数据成员，并在 `read` 和 `write` 操作中对它进行加锁和解锁：

```
#include <IceUtil/Mutex.h>
// ...

namespace Filesystem {
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::Mutex _fileMutex;
    };
    // ...
}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    _fileMutex.lock();
    Lines l = _lines;
    _fileMutex.unlock();
    return l;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    _fileMutex.lock();
    _lines = text;
    _fileMutex.unlock();
}
```

除了我们增加的 `_fileMutex` 数据成员，这个 `FileI` 类与 11.2.2 节的实现是一样的。`read` 和 `write` 操作对互斥体进行加锁和解锁，以确保同一时刻、只有一个线程在读文件或写文件。请注意，我们为每个 `FileI` 实

例使用了单独的互斥体，这样，只要多个线程访问的是不同的文件，它们就仍然能够并发地读写文件。只有对同一文件的并发访问会被序列化。

这里的 `read` 实现有点尴尬：我们必须在持有锁时、创建文件内容的局部副本，然后返回该副本。这样做是必要的，因为我们必须在从函数返回之前解除互斥体的加锁。但我们将在下一节看到，通过使用一个助手类，我们可以不必再创建副本——这个类会在函数返回时自动解除互斥体的加锁。

15.5.2 保证互斥体的解锁

使用互斥体原本的 `lock` 和 `unlock` 操作有一个固有的问题：如果你忘记解除互斥体的加锁，你的程序就会死锁。忘记解锁可能比你想像的要容易，例如：

```
Filesystem::Lines
Filesystem::File::read(const Ice::Current &) const
{
    _fileMutex.lock();                // Lock the mutex
    Lines l = readFileContents();      // Read from database
    _fileMutex.unlock();              // Unlock the mutex
    return l;
}
```

假定我们的文件内容存放在辅助存储器（比如数据库）中，而 `readFileContents` 函数用于访问该文件。这段代码和前面的例子几乎是一样的，但现在有了一个潜伏的 **bug**：如果 `readFileContents` 抛出异常，`read` 函数就会在没有解除互斥体加锁的情况下终止。换句话说，`read` 的这种实现不是异常安全的。

如果你有一个更大的函数，有多条返回路径，很容易发生同样的问题。例如：

```
void
SomeClass::someFunction(/* params here... */)
{
    _mutex.lock();                    // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return;                      // Oops!!!
    }
}
```

```

        // More code here...

        _mutex.unlock();           // Unlock the mutex
    }

```

在这个例子中，位于函数中部的提前返回语句会使互斥体留在加锁状态。尽管这个例子中的问题相当明显，在大型的复杂代码中，异常和提前返回都可能造成难以跟踪的死锁问题。为了避免发生这样的问题，`Mutex` 类含有两个助手类的类型定义，叫作 `Lock` 和 `TryLock`：

```

namespace IceUtil {

    class Mutex {
        // ...

        typedef LockT<Mutex> Lock;
        typedef TryLockT<Mutex> TryLock;
    };
}

```

`LockT` 和 `TryLockT` 是简单的模板，主要由构造器和析构器组成；构造器针对它的参数调用 `lock`，而析构器调用 `unlock`。通过实例化类型为 `Lock` 或 `TryLock` 的局部变量，我们可以彻底消除死锁问题：

```

void
SomeClass::someFunction(/* params here... */)
{
    IceUtil::Mutex::Lock lock(_mutex); // Lock a mutex

    // Lots of complex code here...

    if (someCondition) {
        // More complex code here...
        return; // No problem
    }

    // More code here...
} // Destructor of lock unlocks the mutex

```

在 `someFunction` 的一开始，我们实例化了局部变量 `lock`，其类型是 `IceUtil::Mutex::Lock`。`lock` 的构造器针对互斥体调用 `lock`，使函数的余下部分都处在临界区中。最后，`someFunction` 或者通过普通的方式返回（在函数的中部或尾部），或者因为在函数中有异常抛出而返回。不管函数是怎样终止的，C++ run time 都会解开栈，调用 `lock` 的析构

器，从而解除互斥体的加锁，这样，我们就不会陷入我们先前遇到的死锁问题了。

你应该养成这样的习惯：总是使用 Lock 和 TryLock 助手类，而不是直接调用 lock 和 unlock。这样所得到的代码更容易理解和维护。

我们可以使用 Lock 助手类，重写 read 和 write 操作的实现：

```
Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    IceUtil::Mutex::Lock lock(_fileMutex);
    _lines = text;
}
```

请注意，我们无需再创建 _lines 数据成员的副本：返回值是在互斥体的保护之下初始化的，一旦 lock 的析构器解除互斥体的加锁，其他线程不可能再修改该返回值。

15.6 递归互斥体

我们在第 328 页已经，非递归互斥体不能被多次加锁，即使是持有锁的线程也不行。这在有些情况下会成为问题：程序有多个函数，每个函数都必须获取一个互斥体，而你想要在一个函数的实现中调用另一个函数：

```
IceUtil::Mutex _mutex;

void
f1()
{
    IceUtil::Mutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
```

```
IceUtil::Mutex::Lock lock(_mutex);  
// Some code here...  
  
// Call f1 as a helper function  
f1(); // Deadlock!  
  
// More code here...  
}
```

在操纵数据之前，`f1` 和 `f2` 都会正确地锁住互斥体，但作为其实现的一部分，`f2` 调用了 `f1`。程序会在这里发生死锁，因为 `f2` 已经持有 `f1` 试图获取的锁。就这个简单的例子而言，问题很明显。但在复杂的系统中，如果有许多函数获取和释放锁，要想追踪到这种情况就会变得很困难：加锁约定只出现在源码中，而在调用某个函数之前，每个调用者必须了解要获取（或不获取）哪些锁。这样，复杂性很快就会变得难以控制。

为了消除这个问题，Ice 提供了递归互斥体类 `RecMutex`（在 `IceUtil/RecMutex.h` 中定义）：

```
namespace IceUtil {  
  
    class RecMutex {  
    public:  
        void lock() const;  
        bool tryLock() const;  
        void unlock() const;  
  
        typedef LockT<RecMutex> Lock;  
        typedef TryLockT<RecMutex> TryLock;  
    };  
}
```

请注意，这些操作的型构和 `IceUtil::Mutex` 的是一样的。但是，`RecMutex` 实现的是递归互斥体：

- `lock`

`lock` 函数尝试获取互斥体。如果互斥体已被另一个线程锁住，它就会挂起发出调用的线程，直到互斥体变得可用为止。如果互斥体可用、或者已经被发出调用的线程锁住，这个调用就会锁住互斥体，并立即返回。

- `tryLock`

`tryLock` 函数的功能与 `lock` 类似，但如果互斥体已被另一个线程锁住，它不会阻塞调用者，而会返回 `false`。否则返回值是 `true`。

- `unlock`

`unlock` 函数解除互斥体的加锁。

和非递归互斥体的使用一样，在使用递归互斥体时，你必须遵守一些简单的规则：

- 除非发出调用的线程持有锁，否则不要针对某个互斥体调用 `unlock`。
- 要让互斥体能够被其他线程获取，你调用 `unlock` 的次数必须和你调用 `lock` 的次数相同（在递归互斥体的内部实现中，有一个初始化成零的计数器。每次调用 `lock`，计数器就会加一，每次调用 `unlock`，计数器就会减一；当计数器回到零时，另外的线程就可以获取互斥体了）。

使用递归互斥体，第 332 页的代码片段就能够正确工作了：

```
#include <IceUtil/RecMutex.h>
// ...

IceUtil::RecMutex _mutex;          // Recursive mutex

void
f1()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // ...
}

void
f2()
{
    IceUtil::RecMutex::Lock lock(_mutex);
    // Some code here...

    // Call f1 as a helper function
    f1();                                // Fine

    // More code here...
}
```

请注意，现在互斥体的类型是 `RecMutex`，而不是 `Mutex`，同时，我们使用的是 `RecMutex` 类提供的 `Lock` 类型定义，而不是 `Mutex` 类提供的类型定义。

15.7 读写递归互斥体

在加锁方式上，第 332 页的 `read` 和 `write` 操作的实现较为保守，并非绝对必需：在同一时刻，只有一个线程能处在 `read` 或 `write` 操作中。但是，只有在这样的情况下，我们的并发文件访问才会遇到问题：同一文件有并发的写入者，或者有并发的读取者和写入者。而如果我们只有读取者，我们无需使所有读取线程的访问序列化，因为在它们之中，没有哪个线程会更新文件内容。

Ice 提供了一个读写递归互斥体类 `RWRecMutex`（在 `IceUtil/RWRecMutex.h` 中定义），实现了读取者 - 写入者锁：

```
namespace IceUtil {

    class RWRecMutex {
    public:
        void readLock() const;
        bool tryReadLock() const;
        bool timedReadLock(const Time &) const;

        void writeLock() const;
        bool tryWriteLock() const;
        bool timedWriteLock(const Time &) const;

        void unlock() const;

        void upgrade() const;
        bool timedUpgrade(const Time &) const;

        typedef RLockT<RWRecMutex> RLock;
        typedef TryRLockT<RWRecMutex> TryRLock;
        typedef WLockT<RWRecMutex> WLock;
        typedef TryWLockT<RWRecMutex> TryWLock;
    };
}
```

读写递归锁把通常的单个 `lock` 操作划分成了 `readLock` 和 `writeLock` 操作。多个读取者可以各自并行地获取互斥体。但是，在任一时刻，只有一个写入者能够持有互斥体（既不能有别的读取者，也不能有别的写入者）。`RWRecMutex` 是递归的，也就是说，你可以在同一个线程中多次调用 `readLock` 或 `writeLock`。

下面是各成员函数的行为：

- `readLock`

这个函数尝试获取读锁。如果目前有写入者持有互斥体，调用者就会挂起，直到互斥体变得可用于读取为止。如果可以获取互斥体，或者目前只有读取者持有互斥体，这个调用就会锁住互斥体，并立即返回。

- `tryReadLock`

这个函数尝试获取读锁。如果锁目前由写入者持有，这个函数就会返回 `false`。否则，它获取锁，并返回 `true`。

- `timedReadLock`

这个函数尝试获取读锁。如果锁目前由写入者持有，函数就会等待指定的时长，直到发生超时。如果在超时之前获取了锁，函数就会返回 `true`。否则，一旦发生超时，这个函数就返回 `false`。（关于超时值的构造，参见 15.8 节）。

- `writeLock`

这个函数获取写锁。如果目前有读取者或写入者持有互斥体，调用者就会挂起，直到互斥体可用于写为止。如果可以获取互斥体，这个调用就获取锁，并立即返回。

- `tryWriteLock`

这个函数尝试获取写锁。如果锁目前由读取者或写入者持有，这个函数返回 `false`。否则，它获取锁，返回 `true`。

- `timedWriteLock`

这个函数尝试获取写锁。如果锁目前由读取者或写入者持有，函数就等待指定的时长，直到发生超时。如果在超时之前获取了锁，函数就会返回 `true`。否则，一旦发生超时，这个函数就返回 `false`。（关于超时值的构造，参见 15.8 节）。

- `unlock`

这个函数解除互斥体的加锁（不管目前持有锁的是读取者还是写入者）。

- `upgrade`

这个函数使一个读锁升级成写锁。如果目前有其他读取者持有互斥体，调用者就会挂起，直到互斥体变得可用于写为止。如果可以升级互斥体，调用者就会获取锁，并立即返回。

请注意，`upgrade` 是非递归的。不要在同一个线程中多次调用它。

- `timedUpgrade`

这个函数尝试把读锁升级成写锁。如果目前锁由其他读取者持有，函数就会等待指定的时长，直到发生超时。如果在超时之前获取了锁，这个函数就会返回 `true`。否则，一旦发生超时，函数就返回 `false`。（关于超时值的构造，参见 15.8 节）。

请注意，`timedUpgrade` 是非递归的。不要在同一线程中多次调用它。

和非递归及递归互斥体的使用一样，要想正确使用读写锁，你必须遵守一些规则：

- 除非发出调用的线程持有锁，否则不要针对某个互斥体调用 `unlock`。
- 要让互斥体能够被其他线程获取，你调用 `unlock` 的次数必须和你调用 `lock` 的次数相同。
- 如果你没有持有其读锁，不要针对某个互斥体调用 `upgrade` 或 `timedUpgrade`。
- `upgrade` 和 `timedUpgrade` 是非递归的（因为使它们成为递归的会带来不可接受的性能开销）。不要在同一线程中多次调用这些方法。

读写递归互斥体的实现会偏向写入者：如果有写入者在等待获取锁，就不会允许新的读取者获取锁；该实现会等待目前的所有读取者放弃锁，然后为等待中的写入者锁住互斥体。请注意，互斥体并没有实现任何一种公平性：如果有多个写入者在持续地等待获取写锁，接下来获得锁的是哪一个线程取决于底层的线程实现。在该实现中没有写入者等待队列，有些写入者有可能永远也无法获得对互斥体的访问。

使用 `RWRecMutex`，我们可以实现我们的 `read` 和 `write` 操作，允许多个读取者进行并行读取，或者让单个写入者进行写入：

```
#include <IceUtil/RWRecMutex.h>
// ...

namespace Filesystem {
    // ...

    class FileI : virtual public File,
                  virtual public Filesystem::NodeI {
    public:
        // As before...
    private:
        Lines _lines;
        IceUtil::RWRecMutex _fileMutex; // Read-write mutex
    };
    // ...
}
```

```

}

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::RWRecMutex::RLock lock(_fileMutex);    // Read lock
    return _lines;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    IceUtil::RWRecMutex::WLock lock(_fileMutex);    // Write lock
    _lines = text;
}

```

这段代码和第 329 页的非递归版本几乎是一样的。请注意，唯一的变动是，我们把 `servant` 中的互斥体类型改成了 `RWRecMutex`，并且使用了 `RLock` 和 `WLock` 助手来保证解锁，而不是直接调用 `readLock` 和 `writeLock`。

15.8 定时锁

我们在第 335 页已经看到，读写锁提供了一些可使用超时的成员函数。等待的时间量是通过 `IceUtil::Time` 类的实例指定的（这个类在 `IceUtil/Time.h` 中定义）：

```

namespace IceUtil {

    class Time {
    public:
        Time();
        static Time now();
        static Time seconds(long);
        static Time milliSeconds(long);
        static Time microSeconds(long long);

        Time operator-() const;
        Time operator-(const Time&) const;
        Time operator+(const Time&) const;
        Time& operator-=(const Time&);
        Time& operator+=(const Time&);
    };
}

```

```

        bool operator<(const Time&) const;
        bool operator<=(const Time&) const;
        bool operator>(const Time&) const;
        bool operator>=(const Time&) const;
        bool operator==(const Time&) const;
        bool operator!=(const Time&) const;

        operator timeval() const;
        operator double() const;
    };
}

```

Time 类提供了一些基本的设施，用于获取当前时间、构造时间间隔、加减时间，以及比较时间：

- Time

在内部，Time 类以微秒为单位存储“嘀嗒”数。对于绝对时间，这是自 UNIX 新纪元（1970 年 1 月 1 日的 00:00:00，UTC 时间）以来的微秒数。对于持续时间（duration），这是持续时间对应的微秒数。缺省构造器把嘀嗒计数初始化为零。

- now

这个函数构造一个 Time 对象，并把它初始化为当前时间。

- seconds

```

milliseconds
microseconds

```

这些函数以指定的单位、根据参数构造 Time 对象。例如，下面的代码片段创建持续时间为一分钟的 Time 对象：

```
IceUtil::Time t = IceUtil::Time::seconds(60);
```

- operator-

```

operator+
operator-=
operator+=

```

这些操作符允许你加减 Time 对象。例如：

```

IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
IceUtil::Time oneMinuteAgo = IceUtil::Time::now() - oneMinute;

```

- 比较操作符允许你对时间及时间间隔进行比较，例如：

```

IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
IceUtil::Time twoMinutes = IceUtil::Time::seconds(120);

```

```
assert(oneMinute < twoMinutes);
```

- operator timeval

这个操作符把 Time 对象转换成 struct timeval，后者的定义如下所示：

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

对于需要 struct timeval 参数的 API 调用，比如 select，这种转换是有用的。要把一段持续时间转换进 timeval 结构，只需把 Time 对象赋给 timeval 结构：

```
IceUtil::Time oneMinute = IceUtil::Time::seconds(60);
struct timeval tv;
tv = t;
```

- operator double

这个操作符把 Time 对象转换成按秒表示时间的 double 值：

```
IceUtil::Time t = IceUtil::Time::milliseconds(500);
double d = t;
cerr << "Duration: " << d << " seconds" << endl;
```

运行这段代码，得到的输出是：

```
Duration: 0.5 seconds
```

要把 Time 对象用于定时的锁操作非常简单，例如：

```
#include <IceUtil/RWRecMutex.h>

// ...
IceUtil::RWRecMutex _mutex;

// ...

try {
    // Wait for up to two seconds to get a write lock...
    //
    IceUtil::RWRecMutex::TryWLock
        lock(_mutex, IceUtil::Time::seconds(2));

    // Got the lock -- destructor of lock will unlock
```

```
} catch (const IceUtil::ThreadLockedException &) {  
  
    // Waited for two seconds without getting the lock...  
}
```

请注意，`TryRLock` 和 `TryWLock` 构造器是重载的：如果你只提供一个互斥体作为参数，构造器会调用 `tryReadLock` 或 `tryWriteLock`；如果你既提供了互斥体，也提供了超时，构造器会调用 `timedReadLock` 或 `timedWriteLock`。

15.9 监控器

互斥体实现的是一种简单的互斥机制，在任一时刻，只允许一个线程临界区中活动（在使用读写互斥体的情况下，是一个写入者线程或多个读取者线程）。特别地，要让一个线程进入临界区，另一个线程就必须离开它。这意味着，在使用互斥体时，要做到这样的事情是不可能的：在临界区内挂起一个线程，过一段时间再唤醒它（例如，在某个条件变成真时）。

为了解决这一问题，Ice 提供了监控器。简单地说，监控器是一种用于保护临界区的同步机制：和互斥体一样，同一时刻在临界区内，智能有一个线程在活动。但是，监控器允许你在临界区内挂起线程；这样，另一个线程就能进入临界区。第二个线程可以离开监控器（从而解除监控器的加锁），或者在监控器内挂起自己；不管是哪一种情况，原来的线程都会被唤醒，继续在监控器内执行。这样的行为可以扩展到任意数目的线程，所以在监控器中可以有好几个线程挂起²。

与互斥体相比，监控器提供的互斥机制更为灵活，因为它们允许线程检查某个条件，如果条件为假，就让自己休眠；该线程会由其他某个改变了该条件的线程唤醒。

15.9.1 Monitor 类

Ice 通过 `IceUtil::Monitor` 类提供了监控器（在 `IceUtil/Monitor.h` 中定义）：

2. Ice 提供的监控器具有 *Mesa* 语义，之所以这样称呼，是因为这种监控器最初是由 *Mesa* 编程语言实现的 [12]。许多语言都提供了 *Mesa* 监控器，包括 *Java* 和 *Ada*。按照 *Mesa* 语义，发出信号的线程会继续运行，只有当发出信号的线程挂起自身、或离开监控器时，另外的线程才能得以运行。

```
namespace IceUtil {

    template <class T>
    class Monitor {
    public:
        void lock() const;
        void unlock() const;
        bool tryLock() const;

        void wait() const;
        bool timedWait(const Time&) const;
        void notify();
        void notifyAll();

        typedef LockT<Monitor<T> > Lock;
        typedef TryLockT<Monitor<T> > TryLock;
    };
}
```

请注意，`Monitor` 是一个模板类，需要你用 `Mutex` 或 `RecMutex` 做它的模板参数（用 `RecMutex` 实例化 `Monitor`，得到的监控器是递归的）。

下面是各成员函数的行为：

- `lock`

这个函数尝试锁住监控器。如果监控器已被另外的线程锁住，发出调用的线程就会挂起，直到监控器可用为止。在调用返回时，监控器已被它锁住。

- `tryLock`

这个函数尝试锁住监控器。如果监控器可用，调用就锁住监控器，返回 `true`。如果监控器已被另外的线程锁住，调用返回 `false`。

- `unlock`

这个函数解除监控器的加锁。如果有另外的线程在等待进入监控器（也就是阻塞在 `lock` 调用中），其中一个线程会被唤醒，并锁住监控器。

- `wait`

这个函数挂起发出调用的线程，同时释放监控器上的锁。其他线程可以调用 `notify` 或 `notifyAll` 来唤醒在 `wait` 调用中挂起的线程。当 `wait` 调用返回时，监控器重被锁住，而挂起的线程会恢复执行。

- `timedWait`

这个函数挂起调用它的线程，直到指定的时间流逝。如果有另外的线程调用 `notify` 或 `notifyAll`，在发生超时之前唤醒挂起的线程，这个调用返回 `true`，监控器重被锁住，挂起的线程恢复执行。而如果发生超时，函数返回 `false`。

- `notify`

这个函数唤醒目前在 `wait` 调用中挂起的一个线程。如果在调用 `notify` 时没有这样的线程，通知就会丢失（也就是说，如果没有线程能被唤醒，对 `notify` 的调用不会被记住）。

请注意，发出通知并不会致使另外的线程立即运行。只有当发出通知的线程调用 `wait`、或者解除监控器的加锁时，另外的线程才会得以运行（Mesa 语义）。

- `notifyAll`

这个函数唤醒目前在 `wait` 调用中挂起的所有线程。和 `notify` 一样，如果这时没有挂起的线程，对 `notifyAll` 的调用就会丢失。

和 `notify` 的情况一样，在使用 `notifyAll` 时，只有当发出通知的线程调用 `wait`、或者解除监控器的加锁时，其他线程才会得以运行（Mesa 语义）。

要让监控器正确工作，你必须遵守一些规则：

- 除非你持有锁，否则不要调用 `unlock`。如果你用递归互斥体实例化监控器，你将得到递归的语义，也就是说，要让监控器变得可用，你调用 `unlock` 的次数必须与你调用 `lock`（或 `tryLock`）的次数相同。
- 除非你持有锁，否则不要调用 `wait` 或 `timedWait`。
- 除非你持有锁，否则不要调用 `notify` 或 `notifyAll`。
- 在 `wait` 调用返回时，你必须在继续往前执行之前重新测试条件（参见第 345 页）。

15.9.2 使用监控器

为了说明监控器的使用方法，考虑一个简单的无界队列。一些生产者线程往队列中增加数据项，一些消费者线程从队列中移除数据项。如果队列变空，消费者必须等待生产者把新的数据项放入队列。队列自身是一个临界区，也就是说，当消费者在移除数据项时，我们不能允许生产者把数据项放入队列。下面是这种队列的一个非常简单的实现：

```

template<class T> class Queue {
public:
    void put(const T & item) {
        _q.push_back(item);
    }

    T get() {
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};

```

你可以看到，生产者调用 `put` 方法，把数据项放入队列，而消费者调用 `get` 方法，从队列中取出数据项。显然，这种队列实现不是线程安全的，没有什么能阻止消费者从空的队列中取出数据项。

下面这种版本的队列使用了监控器，如果队列是空的，它会挂起消费者：

```

#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};

```


请注意，现在 Queue 类继承自

IceUtil::Monitor<IceUtil::Mutex>，也就是说，Queue 是一个监控器。

在被调用者时，put 和 get 方法都会锁住监控器。和互斥体的使用一样，我们没有直接调用 lock 和 unlock，而是使用了 Lock 助手，它会自动地在实例化时锁住监控器，在销毁时解除监控器的加锁。

put 方法首先锁住监控器——现在它独自占有临界区——然后往队列里放入数据项。在返回（从而解除监控器的加锁）之前，put 会调用 notify。对 notify 的调用将唤醒休眠在 wait 调用中的任何消费者线程，通知它们已有数据项可用。

get 方法也会锁住监控器，然后，在试图从队列中取出数据项之前，测试队列是否是空的。如果是，消费者就会调用 wait。这会使消费者在 wait 调用中挂起，并解除监控器的加锁，于是生产者就可以进入监控器，把数据项放入队列了。一旦数据项被放入队列，生产者就会调用 notify，致使消费者的 wait 调用完成，监控器再次为消费者而锁住。现在，消费者会从队列中取出数据项，并且返回（从而解除监控器的加锁）。

为了使这样的机制正确工作，get 的实现要做两件事情：

- 在获取了锁之后，get 测试队列是否是空的。
- 在围绕 wait 的循环中，get 重新测试队列是否是空的；如果在 wait 返回后队列仍然是空的，就会重新进入 wait 调用。

在编写代码时，你必须总是遵循同样的模式：

- 除非你持有锁，否则绝对不要测试某个条件。
- 总是在一个围绕 wait 的循环中重新测试条件。如果测试表明条件仍未满足，就再次调用 wait。

如果你不遵守这些规则，最后就会发生这样的事情：共享数据并不处在预期的状态，却有线程对其进行访问。原因如下：

1. 如果你没有持有锁就测试条件，另一个线程完全有可能进入监控器，在你获取锁之前改变其状态。这意味着，到你去锁住监控器时，监控器的状态与测试的结果可能已不再一致。
2. 有些线程实现会遇到叫作欺骗性苏醒（spurious wake-up）的问题：在 notify 被调用之后，可能会有不止一个线程苏醒过来。所以，每个从 wait 调用中返回的线程都必须重新测试条件，以确保监控器处在预期的状态中：wait 返回了，并不说明条件一定发生了变化。

15.9.3 高效通知

只要写入者把数据项放入队列，第 344 页上的线程安全队列的实现就会无条件地通知一个在等待的读取者。如果没有读取者在等待，通知就会丢失，不会造成危害。但是，除非只有一个读取者和写入者，否则许多通知的发送都是不必要的，从而造成无谓的开销。

下面是改正这一问题的一种办法：

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() == 1)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
};
```

在这段代码和第 344 页上的实现之间的唯一不同是，只有当队列长度刚刚从空变成非空时，写入者才会调用 `notify`。这样，就不会再产生不必要的 `notify` 调用。只有在读取者线程只有一个的情况下，这种做法才可行。要想知道为什么，考虑下面的情况：

1. 假定队列里目前有一些数据项，而我们有五个读取者线程。
2. 这五个读取者线程持续调用 `get`，直到队列变空，所有这五个读取者就会在 `get` 中等待。
3. 调度器调度一个写入者线程。写入者发现队列是空的，放入一个数据项，唤醒一个读取者线程。
4. 被唤醒的读取者从队列中取出一个数据项。

5. 这个读取者第二次调用 `get`，发现队列是空的，就再次进入休眠。

实际效果就是，很有可能总是只有一个读取者是活动的；其他四个读取者最后将永远沉睡在 `get` 方法中。

解决这个问题的一种办法是，一旦队列超过特定长度，就调用 `notifyAll`，而不是 `notify`：

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
: public IceUtil::Monitor<IceUtil::Mutex> {
public:
    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_q.size() >= _wakeupThreshold)
            notifyAll();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0)
            wait();
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    const int _wakeupThreshold = 100;
};
```

在这段代码中，我们增加了一个 `private` 数据成员 `_wakeupThreshold`；一旦队列长度超过阈值，写入者就会唤醒所有在等待的读取者，期望所有读取者消费数据项的速度能够比数据项的生产速度更快，从而使队列长度再次降到阈值以下。

这种做法是可行的，但也有缺点：

- `_wakeupThreshold` 的适当的值难以确定，并且易受处理器的速度及数目、以及 I/O 带宽的影响。
- 如果有多个读取者在休眠，一旦写入者调用 `notifyAll`，所有读取者都会变得可以运行。在多处理器机器上，这可能会造成所有读取者都同时运行（每一个读取者在一个 CPU 上运行）。但是，一旦这些线程变得可以运行，在从 `wait` 中返回以前，每个线程都会试图重新获取用于

保护监控器的互斥体。当然，只有一个读取者能实际获取它，其他的读取者会再次挂起，等待互斥体变得可用。实际结果就是，会发生大量线程上下文切换，并且重复而不必要地锁住系统总线。

与调用 `notifyAll` 相比，一种更好的做法是一次唤醒一个在等待的读取者。为此，我们要记住在等待的读取者的数目，只有在有读取者需要唤醒的情况下，才调用 `notify`：

```
#include <IceUtil/Monitor.h>

template<class T> class Queue
    : public IceUtil::Monitor<IceUtil::Mutex> {
public:
    Queue() : _waitingReaders(0) {}

    void put(const T & item) {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        _q.push_back(item);
        if (_waitingReaders)
            notify();
    }

    T get() {
        IceUtil::Monitor<IceUtil::Mutex>::Lock lock(*this);
        while (_q.size() == 0) {
            try {
                ++_waitingReaders;
                wait();
                --_waitingReaders;
            } catch (...) {
                --_waitingReaders;
                throw;
            }
        }
        T item = _q.front();
        _q.pop_front();
        return item;
    }

private:
    list<T> _q;
    short _waitingReaders;
};
```

这个实现用成员变量 `_waitingReaders` 来记住挂起的读取者的数目。构造器把这个变量初始化为零，`get` 的实现在调用 `wait` 之前和之后

使这个变量增大和减小。请注意，这些语句处在 `try-catch` 块中；这样，即使 `wait` 抛出异常，在等待的读取者的计数也仍然会保持准确。最后，只有在有读取者在等待的情况下，`put` 才会调用 `notify`。

这种实现的优点是，它使发生在监控器互斥体之上的竞争降到了最低限度：写入者每次都唤醒一个读取者，所以不会发生多个读取者同时尝试锁住互斥体的情况。而且，监控器的 `notify` 只有在解除了互斥体的加锁之后，才会向等待中的线程发出信号。这意味着，当线程从 `wait` 中醒来、重新尝试获取互斥体时，互斥体很可能处在未加锁状态。这会使随后的操作更高效，因为获取未加锁的互斥体通常会非常高效，而强迫线程在锁住的互斥体上休眠很昂贵（因为必须进行线程上下文切换）。

15.9.4 效率考虑事项

Ice 提供的各种互斥机制在尺寸和速度上各不相同。表 15.1 给出了它们在 Windows 和 Linux 上的相对尺寸（都是在 Intel 架构上）。

Table 15.1. 各种互斥原语的尺寸

原语	按字节计算的尺寸 (Windows)	按字节计算的尺寸 (Linux)
Mutex	24	24
RecMutex	28	28
RWRecMutex	124	60
Monitor<Mutex>	72	40
Monitor<RecMutex>	76	44

表 15.2 给出了在没有加锁竞争的情况下，不同的同步原语的相对性能。

Table 15.2. 各种互斥机制的相对性能

原语	速度 (Windows)	速度 (Linux)
Mutex	1.00	1.00
RecMutex	1.02	1.25
RWRecMutex	10.45 (读锁) 22.45 (写锁)	3.23 (读锁) 3.40 (写锁)
Monitor<Mutex>	0.86	1.09
Monitor<RecMutex>	0.90	1.30

请注意，互斥体和递归互斥体之间的尺寸和性能差异微不足道，所以只有当你的临界区非常小、并且在一个紧密的循环中重复访问时，你才应该使用非递归互斥体。与此类似，监控器的性能和互斥体也很接近（但前者尺寸要大很多）。

读写锁在尺寸和速度上都造成了相当的开销（特别是在 Windows 上），所以只有当你真的要利用多个读取者提供的额外的并行性、同时在临界区内有大量工作要做时，你才应该使用读写锁。

15.10 线程

我们在引言中提到，服务器端 Ice run time 为你创建一个线程池，自动地在每个到来的请求自己的线程中分派它们。因此，当你实现服务器时，你通常只需担心线程间的同步，对临界区进行保护。但是，你也可能想要创建自己的线程。例如，你也许需要一个专用线程，负责响应来自用户界面的输入。同时，如果你有一些复杂的、长时间运行的操作，它们能够利用并行性，你可以把多个线程用于该操作的实现。

Ice 提供了一种简单的线程抽象，不管原生平台是什么，你都可以用它编写可移植的源码。这能把你和原生的底层线程 API 屏蔽开来，而且不管你使用哪一种部署平台，都能够保证统一的语义。

15.10.1 Thread 类

Ice 中的基本线程抽象是由两个类提供的：ThreadControl 和 Thread（在 IceUtil/Thread.h 中定义）：

```
namespace IceUtil {

    class Time;

    typedef ... ThreadId;          // OS-specific definition

    class ThreadControl {
    public:
        ThreadControl();

        ThreadId id() const;
        void join();
        void detach();
        bool isAlive() const;
        static void sleep(const Time &);
        static void yield();

        bool operator==(const ThreadControl &) const;
        bool operator!=(const ThreadControl &) const;
        bool operator<(const ThreadControl &) const;
    };

    class Thread {
    public:
        ThreadId id() const;
        virtual void run() = 0;
        ThreadControl start();
        ThreadControl getThreadControl() const;

        bool operator==(const Thread &) const;
        bool operator!=(const Thread &) const;
        bool operator<(const Thread &) const;
    };
    typedef Handle<Thread> ThreadPtr;
}
```

Thread 类是一个抽象基类，拥有一个纯虚方法 run。要创建线程，你必须特化 Thread 类，并实现 run 方法（这个方法将成为新线程的启动栈帧（starting stack frame））。下面是其他成员函数的行为：

- id

这个函数返回每个线程的唯一标识符，类型是 `ThreadId` (`ThreadId` 是平台相关的 `typedef`。Thread ID 是整数)。对于调试和跟踪，Thread ID 很有用。在调用 `start` 之前调用这个方法，会引发 `ThreadNotStartedException`。

- start

这个成员函数启动新创建的线程（也就是说，调用 `run` 方法）。

- getThreadControl

这个成员函数返回它所在线程的线程控制对象（参见 15.10.4 节）。在调用 `start` 之前调用这个方法，会引发 `ThreadNotStartedException`。

- operator==
operator!=
operator<

这些成员函数比较两个线程的 ID。提供它们的目的，是使你能把 Thread 对象用于有序的 STL 容器。在两个线程启动之前（也就是在获得有效的线程 ID 之前）调用这些方法，会引发 `ThreadNotStartedException`。

请注意，IceUtil 还定义了 `ThreadPtr` 类型。这是常见的引用计数智能指针（参见 6.14.5 节），用以保证自动进行清理：一旦它的引用计数降到零，它的析构器就调用 `delete`，释放动态分配的 Thread 对象。

15.10.2 实现线程

为了说明怎样实现线程，考虑下面的代码片段：

```
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};

class WriterThread : public IceUtil::Thread {
```



```
virtual void run() {  
    for (int i = 0; i < 100; ++i)  
        q.put(i);  
}  
};
```

这段代码定义了两个类：ReaderThread 和 WriterThread，它们都继承自 IceUtil::Thread。每个类都实现了从基类继承的纯虚方法 run。这个简单例子所做的事情是，一个写入者线程把 1 到 100 的数放入 15.9 节定义的线程安全的 Queue 类中，而一个读取者线程从该队列中取出 100 个数，把它们打印到 stdout。

15.10.3 创建线程

要创建新线程，我们只需实例化线程，调用它的 start 方法：

```
IceUtil::ThreadPtr t = new ReaderThread;  
t->start();  
// ...
```

请注意，我们把 new 的返回值赋给类型为 ThreadPtr 的智能指针。这确保了我们不会造成内存泄漏：

1. 在线程创建时，其引用计数被设成零。
2. 在调用 run 之前（run 由 start 方法调用），start 把线程的引用计数加到 1。
3. 线程每多一个 ThreadPtr，它的引用计数都会加 1；每有一个 ThreadPtr 销毁，引用计数都会减 1。
4. 当 run 完成时，start 再次使引用计数减 1，然后检查它的值：如果这时值为零，Thread 对象就会调用 delete 释放自身；如果值不为零，那么就有其他智能指针在引用这个 Thread 对象，当最后一个智能指针出作用域时，才会进行删除活动。

请注意，要让这一切能够工作，你必须在堆上分配你的 Thread 对象——在栈上分配的 Thread 对象会造成释放错误：

```
ReaderThread thread;  
IceUtil::ThreadPtr t = &thread; // Bad news!!!
```

这是错的，因为 t 的析构器最后会调用 delete，对于在栈上分配的对象，其行为是不确定的。

15.10.4 ThreadControl 类

start 方法返回的是类型为 ThreadControl 的对象（参见第 352 页）。下面是 ThreadControl 的成员函数的行为：

- ThreadControl

缺省构造器返回一个 ThreadControl 对象，指向发出调用的线程。这样，即使你先前没有保存当前（发出调用的）线程的句柄，你也能获得该句柄。例如：

```
IceUtil::ThreadControl self;    // Get handle to self
self.yield();                  // Let another thread run
```

这段代码使发出调用的线程让出 CPU，让其他线程运行。这个例子还解释了我们为何有两个类 Thread 和 ThreadControl：没有单独的 ThreadControl，我们就不可能获得任意线程的句柄（请注意，即使发出调用的线程不是由 Ice run time 创建的，这段代码也能工作；例如，你可以为操作系统创建的线程创建 ThreadControl 对象）。

- id

这个函数返回每个线程的唯一标识符，类型是 ThreadId（ThreadId 是平台相关的 typedef。Thread ID 是整数）。对于调试和跟踪，Thread ID 很有用。

- join

这个方法挂起发出调用的线程，直到 join 所针对的线程终止为止。例如：

```
IceUtil::ThreadPtr t = new ReaderThread; // Create a thread
IceUtil::ThreadControl tc = t->start(); // Start it
tc.join();                               // Wait for it
```

如果在进行创建的线程调用 join 时，读取者线程已经结束，对 join 的调用就会立即返回；否则，进行创建的线程就会挂起，直到读取者线程终止为止。

请注意，你只能在一个线程中调用另一个线程的 join 方法，也就是说，只有一个线程能够等待另一个线程终止。如果你在多个线程中调用某个线程的 join 方法，就会产生不确定的行为。

如果你针对先前已经汇合的线程、或是分离的（detached）线程调用 join，也会产生不确定的行为。

- detach

这个方法分离一个线程。一旦线程分离，你不能再与它汇合。

如果你针对已经分离的线程、或是已经汇合的线程调用 `detach`，会产生不确定的行为。

请注意，如果你分离了一个线程，你必须确保这个线程在你的程序离开 `main` 函数之前终止。这意味着，它们的生命期必须比主线程的生命期短，因为分离的线程不能再汇合。

- isAlive

如果底层的线程还没有退出（也就是说，还没有离开它的 `run` 方法），这个方法返回真；否则返回假。如果你要实现非阻塞式的 `join`，`isAlive` 很有用。

- sleep

这个方法挂起线程，时间长度由 `Time` 参数指定（参见 15.8 节）。

- yield

这个方法使得它所针对的线程放弃 CPU，让其他线程运行。

- operator==
operator!=
operator<

和 `Thread` 类的情况一样，这些操作符用于比较线程 ID。于是你就可以创建 `ThreadControl` 对象的有序的 STL 容器了。

和所有同步原语的情况一样，在使用线程时，为了避免产生不确定的行为，你必须遵守一些规则：

- 不要汇合或分离不是你自己创建的线程。
- 对于你创建的每个线程，你必须严格地进行一次汇合或分离；如果你没有这样做，就可能造成资源泄漏。
- 不要在多个线程中针对某个线程调用 `join`。
- 在你创建的其他所有线程终止之前，不要离开 `main`。
- 在你销毁你创建的所有 `Ice::Communicator` 对象之前，不要离开 `main`（或者使用 `Ice::Application` 类——参见第 237 页的 10.3.1 节）。
- 在临界区里调用 `yield` 是一个常见错误。这样做常常是没有意义的，因为 `yield` 调用会寻找另外一个能运行的线程，但当该线程运行时，它很可能会尝试进入由调用 `yield` 的线程持有的临界区，继而再次休

眠。在最好的情况下，这什么也没有达成；而在最坏的情况下，它会造成许多多余的上下文切换，却一无所获。

只有在这样的情况下，你才应该调用 `yield`：另外的线程至少应该有机会实际运行，并做一点有用的事情。

15.10.5 一个小例子

下面是一个小例子，它使用了我们在 15.9 节定义的 `Queue` 类。我们创建五个写入者和五个读取者线程。每个写入者线程往队列里放入 100 个数，而每个读取者线程取回 100 个数，把它们打印到 `stdout`：

```
#include <vector>
#include <IceUtil/Thread.h>
// ...

Queue q;

class ReaderThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            cout << q.get() << endl;
    }
};

class WriterThread : public IceUtil::Thread {
    virtual void run() {
        for (int i = 0; i < 100; ++i)
            q.put(i);
    }
};

int
main()
{
    vector<IceUtil::ThreadControl> threads;
    int i;

    // Create five reader threads and start them
    //
    for (i = 0; i < 5; ++i) {
        IceUtil::ThreadPtr t = new ReaderThread;
        threads.push_back(t->start());
    }
}
```

```
// Create five writer threads and start them
//
for (i = 0; i < 5; ++i) {
    IceUtil::ThreadPtr t = new WriterThread;
    threads.push_back(t->start());
}

// Wait for all threads to finish
//
for (vector<IceUtil::ThreadControl>::iterator i
     = threads.begin(); i != threads.end(); ++i) {
    i->join();
}
}
```

这段代码使用了 `threads` 变量来跟踪所创建的线程，这个变量的类型是 `vector<IceUtil::ThreadControl>`。代码创建五个读取者和五个写入者线程，在 `threads` 向量中存储每个线程的 `ThreadControl` 对象。一旦所有线程都得以创建并运行，代码就会在从 `main` 返回之前与每个线程汇合。

请注意，在与你创建的线程汇合之前，你 *不能* 离开 `main`：如果你从 `main` 返回时，仍有其他线程在运行，许多线程库都会崩溃（这也是你为什么不能在调用 `Communicator::destroy` 之前终止程序的原因（参见第 236 页）；`destroy` 的实现会在返回之前，与所有仍在运行的线程汇合）。

15.11 可移植的信号处理

`IceUtil::CtrlCHandler` 类提供了一种可移植的机制，用以处理 `Ctrl+C` 及其他类似的发给 C++ 进程的信号。在 Windows 上，`IceUtil::CtrlCHandler` 是一个封装 `SetConsoleCtrlHandler` 的包装；在 POSIX 平台上，它会用一个专用线程处理 `SIGHUP`、`SIGTERM` 及 `SIGINT`，这个线程使用 `sigwait` 等待这些信号。信号由用户实现并登记的一个回调函数负责处理。这个回调是一个简单的函数，参数是一个 `int`（信号代码），返回的是 `void`；它不应该抛出任何异常：

```
namespace IceUtil {

    typedef void (*CtrlCHandlerCallback)(int);

    class CtrlCHandler {
```

```
public:
    CtrlCHandler(CtrlCHandlerCallback = 0);
    ~CtrlCHandler();

    void setCallback(CtrlCHandlerCallback);
    CtrlCHandlerCallback getCallback() const;
};
```

下面是 CtrlCHandler 的成员函数的行为：

- 构造器

用一个回调函数构造一个实例。在任一时刻，在一个进程中只能有一个 CtrlCHandler 实例。在 POSIX 平台上，构造器设置 SIGHUP、SIGTERM 及 SIGINT 的掩码，然后启动一个线程，使用 sigwait 等待这些信号。要让信号掩码正确工作，CtrlCHandler 实例必须在启动任何线程之前创建，特别是必须在初始化 Ice 通信器之前创建。

- 析构器

销毁实例，在此之后，在 Windows 上将会恢复缺省的信号处理行为（TerminateProcess）。在 POSIX 平台上，“sigwait”线程将被取消和汇合，但信号掩码保持不变，所以后续信号会被忽略。

- setCallback

设置新的回调函数。

- getCallback

获得当前回调函数。

用零（0）来指定回调函数是合法的，在这种情况下，信号会被捕捉并忽略，直到设置了非零回调函数为止。

CtrlCHandler 的典型用途是关闭 Ice 服务器中的通信器（参见 10.3.1 节）。

15.12 总结

这一章阐释了 Ice 提供的各种线程抽象：互斥体、监控器，以及线程。使用这些 API，你可以让代码变得线程安全，并且创建自己的线程，而无需使用语法或语义随平台不同而不同的、不可移植的 API：Ice 不仅提供了可移植的 API，同时还会保证各种函数的语义在不同平台上是相同的。这

样，创建线程安全的应用就会变得更容易，而且，只需简单的重编译，你就可以在平台之间移动你的代码。

Part III

Advanced Ice



Chapter 16

The Ice Run Time in Detail

16.1 Introduction

Now that we have seen the basics of implementing clients and servers, it is time to look at the Ice run time in more detail. This chapter presents the server-side APIs of the Ice run time in detail for synchronous, oneway, and datagram invocations in detail. (We cover the asynchronous interfaces in Chapter 17.)

Section 16.2 describes the functionality associated with Ice communicators, which are the main handle to the Ice run time. Sections 16.3 to 16.5 describe object adapters and the role they play for call dispatch, and show the relationship between proxies, Ice objects, servants, and object identities. Section 16.6 describes servant locators, which are a major mechanism in Ice for controlling the trade-off between performance and memory consumption. Section 16.7 describes the most common implementation techniques that are used by servers. We suggest that you read this section in detail because knowledge of these techniques is crucial to building systems that perform and scale well. Section 16.8 describes implicit transmission of parameters from client to server and Section 16.9 discusses invocation timeouts. Sections 16.10 to 16.13 describe oneway, datagram, and batched invocations, and Sections 16.14 to 16.16 deal with logging, statistics collection, and location transparency. Finally, Section 16.17 compares the Ice server-side run time with the corresponding CORBA approach.

16.2 Communicators

The main entry point to the Ice run time is represented by the local interface `Ice::Communicator`. An instance of `Ice::Communicator` is associated with a number of run-time resources:

- client-side thread pool

The client-side thread pool ensures that at least one thread is available on the client side to receive replies for outstanding requests. This ensures that no deadlock can occur. For example, if a server calls back into the client from within an operation implementation, the client-side receiver thread can process the request from the server even though the client is waiting for a reply for its request from the same server.

The client-side thread pool is also used for asynchronous method invocation (AMI), to avoid deadlocks in callbacks (see Chapter 17).

- server-side thread pool

Threads in this pool accept incoming connections and handle requests from clients.

- configuration properties

Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties (see Chapter 14).

- object factories

In order to instantiate classes that are derived from a known base type, the communicator maintains a set of object factories that can instantiate the class on behalf of the Ice run time (see Section 6.14.4 and Section 8.14.1).

- a logger object

A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled (see Section 16.14).

- a statistics object

A statistics object implements the `Ice::Stats` interface and is informed about the amount of traffic (bytes sent and received) that is handled by a communicator (see Section 16.15).

- a default router

A router implements the `Ice::Router` interface. Routers are used by Glacier (see Chapter 24) to implement the firewall functionality of Ice.

- a default locator

A locator is an object that resolves an object identity to a proxy. Locator objects are used to build location services, such as IcePack (see Chapter 20).

- a plug-in manager

Plug-ins are objects that add features to a communicator. For example, IceSSL (see Chapter 23) is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.

- object adapters

Object adapters dispatch incoming requests and take care of passing each request to the correct servant.

Object adapters and objects that use different communicators are completely independent from each other. Specifically:

- Each communicator uses its own thread pool. This means that if, for example, one communicator runs out of threads for incoming requests, only objects using that communicator are affected. Objects using other communicators have their own thread pool and are therefore unaffected.
- Collocated invocations across different communicators are not optimized, whereas collocated invocations using the same communicator bypass much of the overhead of call dispatch.

Typically, servers use only a single communicator but, occasionally, multiple communicators can be useful. For example, IceBox (see Chapter 25) uses a separate communicator for each Ice service it loads to ensure that different services cannot interfere with each other. Multiple communicators are also useful to avoid thread starvation: if one service runs out of threads, this leaves the remaining services unaffected.

The interface of the communicator is defined in `Slice`. Part of this interface looks as follows:

```
module Ice {
    local interface Communicator {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);
        ObjectAdapter createObjectAdapter(string name);
    };
};
```

```

        ObjectAdapter createObjectAdapterWithEndpoints(
                                                    string name,
                                                    string endpoints);

        void shutdown();
        void waitForShutdown();
        void destroy();
        // ...
    };
    // ...
};

```

The communicator offers a number of operations:

- proxyToString
- stringToProxy

These operations allow you to convert a proxy into its stringified representation and vice versa.

- createObjectAdapter
- createObjectAdapterWithEndpoints

These operations create a new object adapter. Each object adapter is associated with one or more transport endpoints. Typically, an object adapter has a single transport endpoint. However, an object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.

Whereas `createObjectAdapter` determines which endpoints to bind itself to from configuration information (see Chapter 20), `createObjectAdapterWithEndpoints` allows you to specify the transport endpoints for the new adapter. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property (and so avoid recompilation when a transport endpoint needs to be changed).

- shutdown

This operation shuts down the server side of the Ice run time:

- Operation invocations that are in progress at the time shutdown is called are allowed to complete normally. shutdown does *not* wait for these operations to complete; when shutdown returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at the time you called shutdown may still be running. You can wait for still executing operations to complete by calling `waitForShutdown`.
- Operation invocations that arrive after the server has called shutdown either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (see Chapter 20).

- `waitForShutdown`

This operation suspends the calling thread until the communicator has shutdown (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.

- `destroy`

This operation destroys the communicator and all its associated resources, such as threads, communication endpoints, and memory resources. Once you have destroyed the communicator (and therefore destroyed the run time for that communicator), you must not call any other Ice operation (other than to create another communicator).

It is imperative that you call `destroy` before you leave the `main` function of your program. Failure to do so results in undefined behavior.

Calling `destroy` before leaving `main` is necessary because `destroy` waits for all running threads to terminate before it returns. If you leave `main` without calling `destroy`, you will leave `main` with other threads still running; many threading packages do not allow you to do this and end up crashing your program.

If you call `destroy` without calling `shutdown`, the call waits for all executing operation invocations to complete before it returns (that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivate all object adapters that are associated with the communicator.

On the client side, calling `shutdown` while operations are still executing causes those operations to terminate with a `CommunicatorDestroyedException`.

16.3 Object Adapters

A communicator contains one or more object adapters. An object adapter sits at the boundary between the Ice run time and the server application code and has a number of responsibilities:

- It maps Ice objects to servants for incoming requests and dispatches the requests to the application code in each servant (that is, an object adapter implements an up-call interface that connects the Ice run time and the application code in the server).
- It assists in life cycle operations so Ice objects and servants can be created and existing destroyed without race conditions.
- It provides one or more transport endpoints. Clients access the Ice objects provided by the adapter via those endpoints. (It is also possible to create an object adapter without endpoints. In this case the adapter is used for bidirectional callbacks—see Chapter 24.)

Each object adapter has one or more servants that incarnate Ice objects, as well as one or more transport endpoints. If an object adapter has more than one endpoint, all servants registered with that adapter respond to incoming requests on any of the endpoints. In other words, if an object adapter has multiple transport endpoints, those endpoints represent alternative communication paths to the same set of objects (for example, via different transports).

Each object adapters belongs to exactly one communicator (but a single communicator can have many object adapters).

Each object adapter can optionally have its own thread pool, enabled via the `<adapter-name>.ThreadPool.Size` property (see Appendix C). If so, client invocations for that adapter are dispatched in a thread taken from the adapter's thread pool instead of using a thread from the communicator's server thread pool.

16.3.1 The Active Servant Map

Each object adapter maintains a data structure known as the *active servant map*. The active servant map (or *ASM*, for short) is a lookup table that maps object identities to servants: for C++, the lookup value is a smart pointer to the corresponding servant's location in memory; for Java, the lookup value is a Java reference to the servant. When a client sends an operation invocation to the server, the request is targeted at a specific transport endpoint. Implicitly, the transport endpoint identi-

fies the object adapter that is the target of the request (because no two object adapters can be bound to the same endpoint). The proxy via which the client sends its request contains the object identity for the corresponding object, and the client-side run time sends this object identity over the wire with the invocation. In turn, the object adapter uses that object identity to look in its ASM for the correct servant to dispatch the call to, as shown in Figure 16.1.

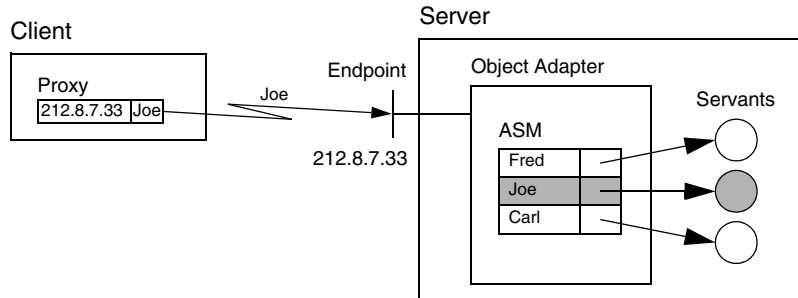


Figure 16.1. Binding a request to the correct servant.

The process of associating a request via a proxy to the correct servant is known as *binding*. The scenario depicted in Figure 16.1 shows direct binding, in which the transport endpoint is embedded in the proxy. Ice also supports an indirect binding mode, in which the correct transport endpoints are provided by the IcePack service (see Chapter 20 for details).

If a client request contains an object identity for which there is no entry in the adapter's ASM, the adapter returns an `ObjectNotExistException` to the client (unless you use a servant locator—see Section 16.6).

16.3.2 Servants

As mentioned in Section 2.2.2, servants are the physical manifestation of an Ice object, that is, they are entities that are implemented in a concrete programming language and instantiated in the server's address space. Servants provide the server-side behavior for operation invocations sent by clients.

The same servant can be registered with one or more object adapters.

16.3.3 Object Adapter Interface

Object adapters are local interfaces:

```

module Ice {
    local interface ObjectAdapter {
        string getName();

        Communicator getCommunicator();

        // ...
    };
};

```

The operations behave as follows:

- The `getName` operation returns the name of the adapter as passed to `Communicator::createObjectAdapter` or `Communicator::createObjectAdapterWithEndpoints`.
- The `getCommunicator` operation returns the communicator that was used to create the adapter.

Note that there are other operations in the `ObjectAdapter` interface; we will explore these throughout the remainder of this chapter.

16.3.4 Servant Activation and Deactivation

The term *servant activation* refers to making the presence of a servant for a particular Ice object known to the Ice run time. Activating a servant adds an entry to the active servant map shown in Figure 16.1. Another way of looking at servant activation is to think of it as creating a link between the identity of an Ice object and the corresponding programming-language servant that handles requests for that Ice object. Once the Ice run time has knowledge of this link, it can dispatch incoming requests to the correct servant. Without this link, that is, without a corresponding entry in the ASM, an incoming request for the identity results in an `ObjectNotExistException`. While a servant is activated, it is said to *incarnate* the corresponding Ice object.

The inverse operation is known as *servant deactivation*. Deactivating a servant removes an entry for a particular identity from the ASM. Thereafter, incoming requests for that identity are no longer dispatched to the servant and result in an `ObjectNotExistException`.

The object adapter offers a number of operations to manage servant activation and deactivation:

```

module Ice {
    local interface ObjectAdapter {
        // ...

        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
        void remove(Identity id);

        // ...
    };
};

```

The operations behave as follows:

- add

The add operation adds a servant with the given identity to the ASM. Requests are dispatched to that servant as soon as add is called. The return value is the proxy for the Ice object incarnated by that servant. The proxy embeds the identity passed to add.

You cannot call add with the same identity more than once: attempts to add an already existing identity to the ASM result in an `AlreadyRegisteredException`. (It does not make sense to add two servants with the same identity because that would make it ambiguous as to which servant should handle incoming requests for that identity.)

Note that it is possible to activate the same servant multiple times with different identities. In that case, the same single servant incarnates multiple Ice objects. We explore the ramifications of this in more detail in Section 16.7.2.

- addWithUUID

The addWithUUID operation behaves the same way as the add operation but does not require you to supply an identity for the servant. Instead, addWithUUID generates a UUID (see [14]) as the identity for the corresponding Ice object. You can retrieve the generated identity by calling the `ice_getIdentity` operation on the returned proxy. addWithUUID is useful to create identities for temporary objects, such as short-lived session objects. (You can also use addWithUUID for persistent objects that do not have a natural identity, as we have done for the filesystem application.)

- remove

The remove operation breaks the association between an identity and its servant by removing the corresponding entry from the ASM. Once the servant

is deactivated, new incoming requests for the removed identity result in an `ObjectNotExistException`. Requests that are executing inside the servant at the time `remove` is called are allowed to complete normally. Once the last request for the servant is complete, the object adapter drops its reference (or smart pointer, for C++) to the servant. At that point, the servant becomes available for garbage collection (or is destroyed, for C++), provided that you do not hold references or smart pointers to the servant elsewhere. The net effect is that a deactivated servant is destroyed once it becomes idle.

Deactivating an object adapter (see Section 16.3.5) implicitly calls `remove` on its active servants.

16.3.5 Adapter States

An object adapter has a number of processing states:

- holding

In this state, any incoming requests for the adapter are held, that is, not dispatched to servants.

For TCP/IP (and other stream-oriented protocols), the server-side run time stops reading from the corresponding transport endpoint while the adapter is in the holding state. In addition, it also does not accept incoming connection requests from clients. This means that if a client sends a request to an adapter that is in the holding state, the client eventually receives a `TimeoutException` or `ConnectTimeoutException` (unless the adapter is placed into the active state before the timer expires).

For UDP, client requests that arrive at an adapter that is in the holding state are thrown away.

Immediately after creation of an adapter, the adapter is in the holding state. This means that requests are not dispatched until you place the adapter into the active state.

- active

In this state, the adapter accepts incoming requests and dispatches them to servants. A newly-created adapter is initially in the holding state. The adapter begins dispatching requests as soon as you place it into the active state.

You can transition between the active and the holding state as many times as you wish.

- inactive

In this state, the adapter has conceptually been destroyed (or is in the process of being destroyed). Deactivating an adapter destroys all transport endpoints that are associated with the adapter. Requests that are executing at the time the adapter is placed into the inactive state are allowed to complete, but no new requests are accepted. (New requests are rejected with an exception). Once an adapter has been deactivated, you cannot reactivate it again in the same process.

The `ObjectAdapter` interface offers operations that allow you to change the adapter state, as well as to wait for a state change to be complete:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        void activate();
        void hold();
        void waitForHold();
        void deactivate();
        void waitForDeactivate();

        // ...
    };
};
```

The operations behave as follows:

- activate

The `activate` operation places the adapter into the active state. Activating an adapter that is already active has no effect. The Ice run time starts dispatching requests to servants for the adapter as soon as `activate` is called.

- hold

The `hold` operation places the adapter into the holding state. Requests that arrive after calling `hold` are held as detailed on page 376. Requests that are in progress at the time `hold` is called are allowed to complete normally. Note that `hold` returns immediately without waiting for currently executing requests to complete.

- waitForHold

The `waitForHold` operation suspends the calling thread until the adapter has completed its transition to the holding state, that is, until all currently executing requests have finished. You can call `waitForHold` from multiple

threads, and you can call `waitForHold` while the adapter is in the active state. Calling `waitForHold` on an adapter in the inactive state does nothing and returns immediately.

- `deactivate`

The `deactivate` operation places the adapter into the inactive state. Requests that arrive after calling `deactivate` are rejected, but currently executing requests are allowed to complete. Once an adapter is in the inactive state, it cannot be reactivated again. Calling `activate`, `hold`, `waitForHold`, or `deactivate` on an adapter that is in the inactive state has no effect. Any servants associated with the adapter are destroyed once they become idle. Note that `deactivate` returns immediately without waiting for the currently executing requests to complete.

- `waitForDeactivate`

The `waitForDeactivate` operation suspends the calling thread until the adapter has completed its transition to the inactive state, that is, until all currently executing requests have completed, all transport endpoints have been closed, and all associated servants have been destroyed. You can call `waitForDeactivate` from multiple threads, and you can call `waitForDeactivate` while the adapter is in the active or holding state. Calling `waitForDeactivate` on an adapter that is in the inactive state does nothing and returns immediately.

Placing an adapter into the holding state is useful, for example, if you need to make state changes in the server that require the server (or a group of servants) to be idle. For example, you could place the implementation of your servants into a dynamic library and upgrade the implementation by loading a newer version of the library at run time without having to shut down the server.

Similarly, waiting for an adapter to complete its transition to the inactive state is useful if your server needs to perform some final clean-up work that cannot be carried out until all executing request have completed.

16.4 Object Identity

Each Ice object has an object identity defined as follows:


```
module Ice {  
    struct Identity {  
        string name;  
        string category;  
    };  
};
```

As you can see, an object identity consists of a pair of strings, a name and a category. Either the name or the category can be empty. (If a proxy contains an identity in which both name and category are empty, Ice interprets that proxy as a null proxy.) The complete object identity is the combination of name and category, that is, for two identities to be equal, both name and category must be the same. The category member is usually the empty string, unless you are using servant locators (see Section 16.6).¹

Object identities can be represented as strings; the category part appears first and is followed by the name; the two components are separated by a / character, for example:

Factory/File

In this example, Factory is the category, and File is the name. If the name or category member themselves contain a / character, the stringified representation escapes the / character with a \, for example:

Factories\Factory/Node/File

In this example, the category member is Factories/Factory and the name member is Node/File.

There are a quite a number of other characters that must be escaped in stringified identities. To make conversion of identities to and from strings easier, the Ice run time provides utility functions that encode and decode identities.

For C++, the utility functions are defined as follows:

```
namespace Ice {  
    std::string  identityToString(const Ice::Identity id);  
    Ice::Identity stringToIdentity(const std::string& s);  
};
```

For Java, the utility functions are in the Ice.Util class and are defined as:

1. Glacier (see Chapter 24) also uses the category member for filtering.

```
package Ice;

public final class Util {
    public static String  identityToString(Identity id);
    public static Identity stringToIdentity(String s);
}
```

These functions correctly encode and decode characters that might otherwise cause problems (such as control characters or white space).

As mentioned on page 375, each entry in the ASM for an object adapter must be unique: you cannot add two servants with the same identity to the ASM.

16.5 The Ice: : Current Object

Up to now, we have tacitly ignored the trailing parameter of type `Ice: : Current` that is passed to each skeleton operation on the server side. The `Current` object is defined as follows:

```
module Ice {
    local dictionary<string, string> Context;

    enum OperationMode { Normal, \Nonmutating, \Idempotent };

    local struct Current {
        ObjectAdapter  adapter;
        Identity       id;
        FacetPath      facet;
        string         operation;
        OperationMode  mode;
        Context        ctx;
    };
};
```

Note that the `Current` object provides access to information about the currently executing request to the implementation of an operation in the server:

- `adapter`

The `adapter` member provides access to the object adapter via which the request is being dispatched. In turn, the adapter provides access to its communicator (via the `getCommunicator` operation).

- `i d`
The `i d` member provides the object identity for the current request (see Section 16.4).
- `facet`
The `facet` member provides access to the facet for the request (see XREF).
- `operati on`
The `operati on` member contains the name of the operation that is being invoked. Note that the operation name may indicate one of the operations on `Ice: : Obj ect`, such as `ice_pi ng` or `ice_i sA`.
- `mode`
The `mode` member contains the invocation mode for the operation (Normal , Idempotent, or Nonmutati ng).
- `ctx`
The `ctx` member contains the current context for the invocation (see Section 16.8).

If you implement your server such that it uses a separate servant for each Ice object, the contents of the `Current` object are not particularly interesting. (You would most likely access the `Current` object to read the `adapter` member, for example, to activate or deactivate a servant.) However, as we will see in Section 16.6, the `Current` object is essential for more sophisticated (and more scalable) servant implementations.

16.6 Servant Locators

Using an adapter's ASM to map Ice objects to servants has a number of design implications:

- Each Ice object is represented by a different servant.²
- All servants for all Ice objects are permanently in memory.

Using a separate servant for each Ice object in this fashion is common to many server implementations: the technique is simple to implement and provides a

2. It is possible to register a single servant with multiple identities. However, there is little point in doing so because a default servant (see Section 16.7.2) achieves the same thing.

natural mapping from Ice objects to servants. Typically, on start-up, the server instantiates a separate servant for each Ice object, activates each servant, and then calls `activate` on the object adapter to start the flow of requests.

There is nothing wrong with the above design, provided that two criteria are met:

1. The server has sufficient memory available to keep a separate servant instantiated for each Ice object at all times.
2. The time required to initialize all the servants on start-up is acceptable.

For many servers, neither criterion presents a problem: provided that the number of servants is small enough and that the servants can be initialized quickly, this is a perfectly acceptable design. However, the design does not scale well: the memory requirements of the server grow linearly with the number of Ice objects so, if the number of objects gets too large (or if each servant stores too much state), the server runs out of memory.

Ice uses *servant locators* to allow you to scale servers to larger numbers of objects.

16.6.1 Overview

In a nutshell, a servant locator is a local object that you implement and attach to an object adapter. Once an adapter has a servant locator, it consults its ASM to locate a servant for an incoming request as usual. If a servant for the request can be found in the ASM, the request is dispatched to that servant. However, if the ASM does not have an entry for the object identity of the request, the object adapter calls back into the servant locator to ask it to provide a servant for the request. The servant locator either

- instantiates a servant and passes it to the Ice run time, in which case the request is dispatched to that newly instantiated servant, or
- the servant locator indicates failure to locate a servant to the Ice run time, in which case the client receives an `ObjectNotExistException`.

This simple mechanism allows us to scale servers to provide access to an unlimited number of Ice objects: instead of instantiating a separate servant for each and every Ice object in existence, the server can instantiate servants for only a subset of Ice objects, namely those that are actually used by clients.

Servant locators are most commonly used by servers that provide access to databases: typically, the number of entries in the database is far larger than what

the server can hold in memory. Servant locators allow the server to only instantiate servants for those Ice objects that are actually used by clients.

Another common use for servant locators is in servers that are used for process control or network management: in that case, there is no database but, instead, there is a potentially very large number of devices or network elements that must be controlled via the server. Otherwise, this scenario is the same as for large databases: the number of Ice objects exceeds the number of servants that the server can hold in memory and, therefore, requires an approach that allows the number of instantiated servants to be less than the number of Ice objects.

16.6.2 Servant Locator Interface

A servant locator has the following interface:

```
module Ice {
    local interface ServantLocator {
        Object locate(    Current      curr,
                       out Local Object cookie);

        void finished(    Current      curr,
                        Object      servant,
                        Local Object cookie);

        void deactivate();
    };
};
```

Note that `ServantLocator` is a local interface. To create an actual implementation of a servant locator, you must define a class that is derived from

`Ice::ServantLocator` and provide implementations of the `locate`, `finished`, and `deactivate` operations. The Ice run time invokes the operations on your derived class as follows:

- `locate`

Whenever a request arrives for which no entry exists in the ASM, the Ice run time calls `locate`. The implementation of `locate` (which you provide as part of the derived class) is supposed to return a servant that can process the incoming request. Your implementation of `locate` can behave in three possible ways:

1. Instantiate and return a servant for the current request. In this case, the Ice run time dispatches the request to the newly instantiated servant.

2. Return null. In this case, the Ice run time raises an `ObjectNotExistException` in the client.
3. Throw an exception. In this case, the Ice run time propagates the thrown exception back to the client. (Keep in mind that all exceptions, apart from `ObjectNotExistException`, `OperationNotExistException`, and `FacetNotExistException`, are presented as `UnknownLocalException` to the client.)

The `cookie` out-parameter to `locate` allows you return a local object to the object adapter. The object adapter does not care about the contents of that object (and it is legal to return a null cookie). Instead, the Ice run time passes whatever cookie you return from `locate` back to you when it calls `finish`. This allows you to pass an arbitrary amount of state from `locate` to the corresponding call to `finish`.

- `finish`

If a call to `locate` has returned a servant to the Ice run time, the Ice run time dispatches the incoming request to the servant. Once the request is complete (that is, the operation being invoked has completed), the Ice run time calls `finish`, passing the servant whose operation has completed, the `Current` object for the request, and the cookie that was initially created by `locate`. This means that every call to `locate` is balanced by a corresponding call to `finish` (provided that `locate` actually returned a servant).

You cannot throw exceptions from `finish`—if you do, the Ice run time logs an error messages and (for connection-oriented transports) closes the connection to the client.

- `deactivate`

The `deactivate` operation is called by the Ice run time when the object adapter to which the servant locator is attached is deactivated. This allows the servant locator to clean up (for example, the locator might close a database connection).

It is important to realize that the Ice run time does not “remember” the servant that is returned by a particular call to `locate`. Instead, the Ice run time simply dispatches an incoming request to the servant returned by `locate` and, once the request is complete, calls `finish`. In particular, if two requests for the same servant arrive more or less simultaneously, the Ice run time calls `locate` and `finish` once for each request. In other words, `locate` establishes the association between an object identity and a servant; that association is valid only for a single request and is never used by the Ice run time to dispatch a different request.

16.6.3 Threading Guarantees for Servant Locators

The Ice run time guarantees that every operation invocation that involves a servant locator is bracketed by calls to `locate` and `finish`, that is, every call to `locate` is balanced by a corresponding call to `finish` (assuming that the call to `locate` actually returned a servant, of course).

In addition, the Ice run time guarantees that `locate`, the operation, and `finish` are called by the same thread. This guarantee is important because it allows you to use `locate` and `finish` to implement thread-specific pre- and post-processing around operation invocations. (For example, you can start a transaction in `locate` and commit or rollback that transaction in `finish`, or you can acquire a lock in `locate` and release the lock in `finish`.³)

Note that, if you are using asynchronous method dispatch (see Chapter 17), the thread that starts a call is not necessarily the thread that finishes it. In that case, `finish` is called by whatever thread executes the operation implementation, which is may be different thread than the one that called `locate`.

The Ice run time also guarantees that `deactivate` is called when you deactivate the object adapter to which the servant locator is attached. The `deactivate` call is made only once all operations that involved the servant locator are finished, that is, `deactivate` is guaranteed not to run concurrently with `locate` or `finish`, and is guaranteed to be *last* call made to a servant locator.

Beyond this, the Ice run time provides no threading guarantees for servant locators. In particular:

- It is possible for invocations of `locate` to proceed concurrently (for the same object identity or for different object identities).
- It is possible for invocations of `finish` to proceed concurrently (for the same object identity or for different object identities).
- It is possible for invocations of `locate` and `finish` to proceed concurrently (for the same object identity or for different object identities).

These semantics allow you to extract the maximum amount of parallelism from your application code (because the Ice run time does not serialize invocations when serialization may not be necessary). Of course, this means that you must protect access to shared data from `locate` and `finish` with mutual exclusion primitives as necessary.

3. Both transactions and locks usually are thread-specific, that is, only the thread that started a transaction can commit it or roll it back, and only the thread that acquired a lock can release the lock.

16.6.4 Servant Locator Registration

An object adapter does not automatically know when you create a servant locator. Instead, you must explicitly register servant locators with the object adapter:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        void addServantLocator(ServantLocator locator,
                               string category);

        ServantLocator findServantLocator(string category);

        // ...
    };
};
```

As you can see, the object adapter allows you add and find servant locators. Note that, when you register a servant locator, you must provide an argument for the category parameter. The value of the category parameter controls which object identities the servant locator is responsible for: only object identities with a matching category member (see page 378) trigger a corresponding call to `locate`. An incoming request for which no explicit entry exists in the ASM and with a category for which no servant locator is registered returns an `ObjectNotExistException` to the client.

`addServantLocator` has the following semantics:

- You can register exactly one servant locator for a specific category. Attempts to call `addServantLocator` for the same category more than once raise an `AlreadyRegisteredException`.
- You can register different servant locators for different categories, or you can register the same single servant locator multiple times (each time for a different category). In the former case, the category is implicit in the servant locator instance that is called by the Ice run time; in the latter case, the implementation of `locate` can find out which category the incoming request is for by examining the object identity member of the `Current` object that is passed to `locate`.
- It is legal to register a servant locator for the empty category. Such a servant locator is known as a *default servant locator*: if a request comes in for which no entry exists in the ASM, and whose category does not match the category

of any other registered servant locator, the Ice run time calls `locate` on the default servant locator.

Note that, once registered, you cannot change or remove the servant locator for a category. The life cycle of the servant locators for an object adapter ends with the life cycle of the adapter: when the object adapter is deactivated, so are its servant locators.

The `findServantLocator` operation allows you retrieve the servant locator for a specific category (including the empty category). If no servant locator is registered for the specified category, `findServantLocator` returns null.

Call Dispatch Semantics

The preceding rules may seem complicated, so here is a summary of the actions taken by the Ice run time to locate a servant for an incoming request.

Every incoming request implicitly identifies a specific object adapter for the request (because the request arrives at a specific transport endpoint and, therefore, identifies a particular object adapter). The incoming request carries an object identity that must be mapped to a servant. To locate a servant, the Ice run time goes through the following steps, in the order shown:

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request to the corresponding servant.
2. If the category of the incoming object identity is non-empty, look for a servant locator that is registered for that category. If such a servant locator is registered, call `locate` on the servant locator and, if `locate` returns a servant, dispatch the request to that servant, followed by a call to `finished`; otherwise, if the call to `locate` returns null, raise `ObjectNotExistException` in the client.
3. If the category of the incoming object identity is empty, or no servant locator could be found for the category in step 2, look for a default servant locator (that is, a servant locator that is registered for the empty category). If a default servant locator is registered, dispatch the request as for step 2.
4. Raise `ObjectNotExistException` in the client.

It is important to keep these call dispatch semantics in mind because they enable a number of powerful implementation techniques. Each technique allows you to streamline your server implementation and to precisely control the trade-off between performance, memory consumption, and scalability. To illustrate the possibilities, we outline a number of the most common implementation techniques in the following section.

16.6.5 Implementing a Simple Servant Locator

To illustrate the concepts outlined in the previous sections, let us examine a (very simple) implementation of a servant locator. Consider that we want to create an electronic phone book for the entire world's telephone system (which, clearly, involves a very large number of entries, certainly too many to hold the entire phone book in memory). The actual phone book entries are kept in a large database. Also assume that we have a search operation that returns the details of a phone book entry. The Slice definitions for this application might look something like the following:

```
struct Details {  
    // Lots of details about the entry here...  
};  
  
interface PhoneEntry {  
    nonmutating Details getDetails();  
    idempotent void updateDetails(Details d);  
    // ...  
};  
  
struct SearchCriteria {  
    // Fields to permit searching...  
};  
  
interface PhoneBook {  
    nonmutating PhoneEntry* search(SearchCriteria c);  
    // ...  
};
```

The details of the application do not really matter here; the important point to note is that each phone book entry is represented as an interface for which we need to create a servant eventually, but we cannot afford to keep servants for all entries permanently in memory.

Each entry in the phone database has a unique identifier. This identifier might be an internal database identifier, or a combination of field values, depending on exactly how the database is constructed. The important point is that we can use this database identifier to link the proxy for an Ice object to its persistent state: we simply use the database identifier as the object identity. This means that each proxy contains the primary access key that is required to locate the persistent state of each Ice object and, therefore, instantiate a servant for that Ice object.

What follows is an outline implementation in C++. The class definition of our servant locator looks as follows:

```

class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr & cookie);

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie);

    virtual void deactivate(const std::string& category);
};

```

Note that `MyServantLocator` inherits from `Ice::ServantLocator` and implements the pure virtual functions that are generated by the `slice2cpp` compiler for the `Ice::ServantLocator` interface. Of course, as always, you can add additional member functions, such as a constructor and destructor, and you can add private data members as necessary to support your implementation.

In C++, you can implement the `locate` member function along the following lines:

```

Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                          Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error &)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}

```

For the time being, the implementations of `finished` and `deactivate` are empty and do nothing.

The `DB_lookup` call in the preceding example is assumed to access the database. If the lookup fails (presumably, because no matching record could be found), `DB_lookup` throws a `DB_error` exception. The code catches that exception and throws an `Ice::ObjectNotExistException` in its stead; that exception is returned to the client to indicate that the client used a proxy to a no-longer existent Ice object.

Note that `locate` instantiates the servant on the heap and returns it to the Ice run time. This raises the question of when the servant will be destroyed. The answer is that the Ice run time holds onto the servant for as long as necessary, that is, long enough to invoke the operation on the returned servant and to call `finished` once the operation has completed. Thereafter, the servant is no longer needed and the Ice run time destroys the smart pointer that was returned by `locate`. In turn, because no other smart pointers exist for the same servant, this causes the destructor of the `PhoneEntryI` instance to be called, and the servant to be destroyed.

The upshot of this design is that, for every incoming request, we instantiate a servant and allow the Ice run time to destroy the servant once the request is complete. Depending on your application, this may be exactly what is needed, or it may be prohibitively expensive—we will explore designs that avoid creation and destruction of a servant for every request shortly.

In Java, the implementation of our servant locator looks very similar:

```
public class MyServantLocator
    extends Ice.LocalObjectImpl
    implements Ice.ServantLocator {

    public Ice.Object locate(Ice.Current c,
                           Ice.LocalObjectHolder cookie)
    {
        // Get the object identity. (We use the name member
        // as the database key.
        String name = c.id.name;

        // Use the identity to retrieve the state
        // from the database.
        //
        ServantDetails d;
        try {
            d = DB.lookup(name);
        } catch (DB.error e) {
```

```

        throw new Ice.ObjectNotExistException();
    }

    // We have the state, instantiate a servant and return it.
    //
    return new PhoneEntryI(d);
}

public void finished(Ice.Current c,
                    Ice.Object servant,
                    Ice.LocalObject cookie)
{
}

public void deactivate(String category)
{
}
}

```

Note that the `MyServantLocator` class extends `Ice.LocalObjectImpl`. `LocalObjectImpl` is provided by the Ice run time and contains the implementation of methods that are common to all local objects (namely, `equals`, `clone`, and `ice_hash`).

A word of warning about the Java mapping: if you want to keep a copy of one of the members of `Ice.Current`, or a copy of `Ice.Current` itself beyond the scope of `locate` or `finished`, you *must* make a copy of the value instead of just storing a reference: for efficiency reasons, the Java Ice run time does not allocate separate objects for each member of a `Current` object. Instead, a pool of `Current` objects is reused for different invocations and the members of the various `Current` objects are assigned to just prior to call dispatch. This means that, if you store a reference to, for example, the identity inside a `Current` object, that identity may unexpectedly change later when the Ice run-time reuses that `Current` object.⁴

All implementations of `locate` follow the pattern illustrated by the previous pseudo-code:

1. Use the `id` member of the passed `Current` object to obtain the object identity. Typically, only the `name` member of the identity is used to retrieve servant

4. The `Ice.Current` object is unique in this respect: for all other parameters, it is safe to store a reference to a parameter beyond the scope of the current invocations.

state. The category member is normally used to select a servant locator. (We will explore use of the category member shortly.)

2. Retrieve the state of the Ice object from secondary storage (or the network) using the object identity as a key.
 - If the lookup succeeds, you have retrieved the state of the Ice object.
 - If the lookup fails, throw an `ObjectNotExistException`. In that case, the Ice object for the client's request truly does not exist, presumably, because that Ice object was deleted earlier, but the client still has a proxy to the now-deleted object.
3. Instantiate a servant and use the state retrieved from the database to initialize the servant. (In this example, we pass the retrieved state to the servant constructor.)
4. Return the servant.

Of course, before we can use our servant locator, we must inform the adapter of its existence prior to activating the adapter, for example (in Java):

```
MyServantLocator sl = new MyServantLocator();
adapter.addServantLocator(sl, "");
```

Note that, in this example, we have installed the servant locator for the empty category. This means that `locate` on our servant locator will be called for invocations to any of our Ice objects (because the empty category acts as the default). In effect, with this design, we are not using the category member of the object identity. This is fine, as long as all our servants all have the same, single interface. However, if we need to support several different interfaces in the same server, this simple strategy is no longer sufficient.

16.6.6 Using the category Member of the Object Identity

The simple example in the preceding section always instantiates a servant of type `PhoneEntryI`. In other words, the servant locator implicitly is aware of the type of servant the incoming request is for. This is not a very realistic assumption for most servers because, usually, a server provides access to objects with several different interfaces. This poses a problem for our `locate` implementation: somehow, we need to decide inside `locate` what type of servant to instantiate. You have several options for solving this problem:

- Use a separate object adapter for each interface type and use a separate servant locator for each object adapter.

This technique works fine, but has the down-side that each object adapter requires a separate transport endpoint, which is wasteful.

- Mangle a type identifier into the name component of the object identity.

This technique uses part of the object identity to denote what type of object to instantiate. For example, in our file system application, we have directory and file objects. By convention, we could prepend a 'd' to the identity of every directory and prepend an 'f' to the identity of every file. The servant locator then can use the first letter of the identity to decide what type of servant to instantiate:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                        Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;
    std::string realId = c.id.name.substr(1);
    try {
        if(name[0] == 'd') {
            // The request is for a directory.
            //
            DirectoryDetails d = DB_lookup(realId);
            return new DirectoryI(d);
        } else {
            // The request is for a file.
            //
            FileDetails d = DB_lookup(realId);
            return new FileI(d);
        }
    } catch (DatabaseNotFoundException &) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

While this works, it is awkward: not only do we need to parse the name member to work out what type of object to instantiate, but we also need to modify the implementation of `locate` whenever we add a new type to our application.

- Use the category member of the object identity to denote the type of servant to instantiate.

This is the recommended approach: for every interface type, we assign a separate identifier as the value of the category member of the object identity. (For example, we can use ‘d’ for directories and ‘f’ for files.) Instead of registering a single servant locator, we create two different servant locator implementations, one for directories and one for files, and then register each locator for the appropriate category:

```
class DirectoryLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                   Ice::LocalObjectPtr & cookie)
    {
        // Code to locate and instantiate a directory here...
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie)
    {
    }

    virtual void deactivate(const std::string& category)
    {
    }
};

class FileLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                   Ice::LocalObjectPtr & cookie)
    {
        // Code to locate and instantiate a file here...
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie)
    {
    }
};
```



```

    }

    virtual void deactivate(const std::string& category)
    {
    }

};

// ...

// Register two locators, one for directories and
// one for files.
//
adapter->addServantLocator(new DirectoryLocator(), "d");
adapter->addServantLocator(new FileLocator(), "f");

```

Yet another option is to use the category member of the object identity, but to use a single default servant locator (that is, a locator for the empty category). With this approach, all invocations go to the single default servant locator, and you can switch on the category value inside the implementation of the `locate` operation to determine which type of servant to instantiate. However, this approach is harder to maintain than the previous one; the category member of the Ice object identity exists specifically to support servant locators, so you might as well use it as intended.

16.6.7 Using Cookies

Occasionally, it can be useful to be able to pass information between `locate` and `finished`. For example, the implementation of `locate` could choose among a number of alternative database backends, depending on load or availability and, to properly finalize state, the implementation of `finished` might need to know which database was used by `locate`. To support such scenarios, you can create a cookie in your `locate` implementation; the Ice run time passes the value of the cookie to `finished` after the operation invocation has completed. The cookie must be derived from `Ice::LocalObject` and can contain whatever state and member functions are useful to your implementation:

```

class MyCookie : public virtual Ice::LocalObject {
public:
    // Whatever is useful here...
};

typedef IceUtil::Handle<MyCookie> MyCookiePtr;

```

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                   Ice::LocalObjectPtr & cookie)
    {
        // Code as before...

        // Allocate and initialize a cookie.
        //
        cookie = new MyCookie(...);

        return new PhoneEntryI();
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie)
    {
        // Down-cast cookie to actual type.
        //
        MyCookiePtr mc = MyCookiePtr::dynamicCast(cookie);

        // Use information in cookie to clean up...
        //
        // ...
    }

    virtual void deactivate(const std::string& category);
};
```

16.7 Server Implementation Techniques

As we mentioned on page 382, instantiating a servant for each Ice object on server start-up is a viable design, provided that you can afford the amount of memory required by the servants, as well as the delay in start-up of the server. However, Ice supports more flexible mappings between Ice objects and servants; these alternate mappings allow you to precisely control the trade-off between memory consumption, scalability, and performance. We outline a few of the more common implementation techniques in this section.

16.7.1 Incremental Initialization

If you use a servant locator, the servant returned by `locate` is used only for the current request, that is, the Ice run time does not add the servant to the Active Servant Map. Of course, this means that if another request comes in for the same Ice object, `locate` must again retrieve the object state and instantiate a servant. A common implementation technique is to add each servant to the ASM as part of `locate`. This means that only the first request for each Ice object triggers a call to `locate`; thereafter, the servant for the corresponding Ice object can be found in the ASM and the Ice run time can immediately dispatch another incoming request for the same Ice object without having to call the servant locator.

An implementation of `locate` to do this would look something like the following:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                        Ice::LocalObjectPtr & cookie)
{
    // Get the object identity. (We use the name member
    // as the database key.)
    //
    std::string name = c.id.name;

    // Use the identity to retrieve the state from the database.
    //
    ServantDetails d;
    try {
        d = DB_lookup(name);
    } catch (const DB_error &)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // We have the state, instantiate a servant.
    //
    Ice::ObjectPtr servant = new PhoneEntryI(d);

    // Add the servant to the ASM.
    //
    c.adapter->add(servant, c.id);      // NOTE: Incorrect!

    return servant;
}
```

This is almost identical to the implementation on page 389—the only difference is that we also add the servant to the ASM by calling `ObjectAdapter::add`. Unfor-

unately, this implementation is wrong because it suffers from a race condition. Consider the situation where we do not have a servant for a particular Ice object in the ASM, and two clients more or less simultaneously send a request for the same Ice object. It is entirely possible for the thread scheduler to schedule the two incoming requests such that the Ice run time completes the lookup in the ASM for both requests and, for each request, concludes that no servant is in memory. The net effect is that `locate` will be called twice for the same Ice object, and our servant locator will instantiate two servants instead of a single servant. Because the second call to `ObjectAdapter::add` will raise an `AlreadyRegisteredException`, only one of the two servants will be added to the ASM.

Of course, this is hardly the behavior we expect. To avoid the race condition, our implementation of `locate` must check whether a concurrent invocation has already instantiated a servant for the incoming request and, if so, return that servant instead of instantiating a new one. The Ice run time provides the `ObjectAdapter::identityToServant` operation to allow us to test whether an entry for a specific identity already exists in the ASM:

```
module Ice {
    local interface ObjectAdapter {
        // ...

        Object identityToServant(Identity id);

        // ...
    };
};
```

`identityToServant` returns the servant if it exists in the ASM and null, otherwise. Using this lookup function, together with a mutex, allows us to correctly implement `locate`. The class definition of our servant locator now has a private mutex so we can establish a critical region inside `locate`:

```
class MyServantLocator : public virtual Ice::ServantLocator {
public:

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                  Ice::LocalObjectPtr &);

    // Declaration of finished() and deactivate() here...

private:
    IceUtil::Mutex _m;
};
```

The `locate` member function locks the mutex and tests whether a servant is already in the ASM: if so, it returns that servant; otherwise, it instantiates a new servant and adds it to the ASM as before:

```
Ice::ObjectPtr
MyServantLocator::locate(const Ice::Current & c,
                        Ice::LocalObjectPtr &)
{
    IceUtil::Mutex::Lock lock(_m);

    // Check if we have instantiated a servant already.
    //
    Ice::ObjectPtr servant = c.adapter.identityToServant(c.id);

    if(!servant) {          // We don't have a servant already

        // Instantiate a servant.
        //
        ServantDetails d;
        try {
            d = DB_lookup(c.id.name);
        } catch (const DB_error &) {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
        servant = new PhoneEntryI(d);

        // Add the servant to the ASM.
        //
        c.adapter->add(servant, c.id);
    }

    return servant;
}
```

The Java version of this locator is almost identical, but we use the `synchronized` qualifier instead of a mutex to make `locate` a critical region:

```
synchronized public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Check if we have instantiated a servant already.
    //
    Ice.Object servant = c.adapter.identityToServant(c.id);

    if(servant == null) { // We don't have a servant already
```

```
        // Instantiate a servant
        //
        ServantDetails d;
        try {
            d = DB.lookup(c.id.name);
        } catch (DB.error &) {
            throw new Ice.ObjectNotExistException();
        }
        servant = new PhoneEntryI(d);
    }

    return servant;
}
```

Using a servant locator that adds the servant to the ASM has a number of advantages:

- Servants are instantiated on demand, so the cost of initializing the servants is spread out over many invocations instead of being incurred all at once during server start-up.
- The memory requirements for the server are reduced because servants are instantiated only for those Ice objects that are actually accessed by clients. If clients only access a subset of the total number of Ice objects, the memory savings can be substantial.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers, chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

16.7.2 Default Servants

A common problem with object-oriented middleware is scalability: servers frequently are used as front ends to large databases that are accessed remotely by clients. The servers' job is to present an object-oriented view to clients of a very large number of records in the database. Typically, the number of records is far too large to instantiate servants for even a fraction of the database records.

A common technique for solving this problem is to use *default servants*. A default servant is a servant that, for each request, takes on the persona of a different Ice object. In other words, the servant changes its behavior according to

the object identity that is accessed by a request, on a per-request basis. In this way, it is possible to allow clients access to an unlimited number of Ice objects with only a single servant in memory.

Default servant implementations are attractive not only because of the memory savings they offer, but also because of the simplicity of implementation: in essence, a default servant is a facade [2] to the persistent state of an object in the database. This means that the programming required to implement a default servant is typically minimal: it simply consists of the code required to read and write the corresponding database records.

To create a default servant implementation, we need as many locators as there are non-abstract interfaces in the system. For example, for our file system application, we require two locators, one for directories and one for files. In addition, the object identities we create use the category member of the object identity to encode the type of interface of the corresponding Ice object. The value of the category field can be anything that identifies the interface, such as the 'd' and 'f' convention we used on page 394. Alternatively, you could use "Directory" and "File", or use the type ID of the corresponding interface, such as "::Filesystem::Directory" and "::Filesystem::File". The name member of the object identity must be set to whatever identifier we can use to retrieve the persistent state of each directory and file from secondary storage. (For our filesystem application, we used a UUID as a unique identifier.)

The servant locators each return a singleton servant from their respective locate implementations. Here is the servant locator for directories:

```
class DirectoryLocator : public virtual Ice::ServantLocator {
public:
    DirectoryLocator() : _servant(new DirectoryI)
    {
    }

    virtual Ice::ObjectPtr locate(const Ice::Current & c,
                                   Ice::LocalObjectPtr & cookie)
    {
        return _servant;
    }

    virtual void finished(const Ice::Current & c,
                          const Ice::ObjectPtr & servant,
                          const Ice::LocalObjectPtr & cookie);

    virtual void deactivate(const std::string& category);
```

```
private:
    Ice::ObjectPtr _servant;
};
```

Note that constructor of the locator instantiates a servant and returns that same servant from every call to `locate`. The implementation of the file locator is analogous to the one for directories. Registration of the servant locators proceeds as usual:

```
_adapter->addServantLocator(new DirectoryLocator, "d");
_adapter->addServantLocator(new FileLocator, "f");
```

All the action happens in the implementation of the operations, using the following steps for each operation:

1. Use the passed `Current` object to get the identity for the current request.
2. Use the name member of the identity to locate the persistent state of the servant on secondary storage. If no record can be found for the identity, throw an `ObjectNotExistException`.
3. Implement the operation to operate on that retrieved state (returning the state or updating the state as appropriate for the operation).

In pseudo-code, this might look something like the following:

```
FileSystem::NodeSeq
FileSystem::DirectoryI::list(const Ice::Current &c) const
{
    // Use the identity of the directory to retrieve
    // its contents.
    DirectoryContents dc;
    try {
        dc = DB_getDirectory(c.id.name);
    } catch(const DB_error &) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    // Use the records retrieved from the database to
    // initialize return value.
    //
    FileSystem::NodeSeq ns;
    // ...

    return ns;
}
```


Note that the servant implementation is completely stateless: the only state it operates on is the identity of the Ice object for the current request (and that identity is passed as part of the `Current` parameter). The price we have to pay for the unlimited scalability and reduced memory footprint is performance: default servants make a database access for every invoked operation which is obviously slower than caching state in memory, as part of a servant that has been added to the ASM. However, this does not mean that default servants carry an unacceptable performance penalty: database often provide sophisticated caching, so even though the operation implementations read and write the database, as long as they access cached state, performance may be entirely acceptable.

Overriding `ice_ping`

One issue you need to be aware of with default servants is the need to override `ice_ping`: the default implementation of `ice_ping` that the servant inherits from its skeleton class always succeeds. For servants that are registered with the ASM, this is exactly what we want; however, for default servants, `ice_ping` must fail if a client uses a proxy to a no-longer existent Ice object. To avoid getting successful `ice_ping` invocations for non-existent Ice objects, you must override `ice_ping` in the default servant. The implementation must check whether the object identity for the request denotes a still-existing Ice object and, if not, throw `ObjectNotExistException`:

```
void
FilesystemDirectoryI::ice_ping(const Ice::Current &c)
{
    try {
        d = DB_lookup(c.id.name);
    } catch (const DB_error &) {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
}
```

It is good practice to override `ice_ping` if you are using default servants.

16.7.3 Hybrid Approaches and Caching

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the

state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less-frequently accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the UNIX kernel [10]. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

16.7.4 Servant Evictors

A variation on the previous theme and particularly interesting use of a servant locator is as an *evictor* [4]. An evictor is a servant locator that maintains a cache of servants:

- Whenever a request arrives (that is, `locate` is called by the Ice run time), the evictor checks to see whether it can find a servant for the request in its cache. If so, it returns the servant that is already instantiated in the cache; otherwise, it instantiates a servant and adds it to the cache.
- The cache is a queue that is maintained in least-recently used (LRU) order: the least-recently used servant is at the tail of the queue, and the most-recently used servant is at the head of the queue. Whenever a servant is returned from or added to the cache, it is moved from its current queue position to the head of the queue, that is, the “newest” servant is always at the head, and the “oldest” servant is always at the tail.
- The queue has a configurable length that corresponds to how many servants will be held in the cache; if a request arrives for an Ice object that does not have a servant in memory and the cache is full, the evictor removes the least-recently used servant at the tail of the queue from the cache in order to make room for the servant about to be instantiated at the head of the queue.

Figure 16.2 illustrates an evictor with a cache size of five after five invocations have been made, for object identities 1 to 5, in that order.

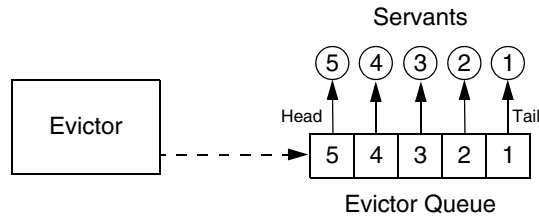


Figure 16.2. An evictor after five invocations for object identities 1 to 5.

At this point, the evictor has instantiated five servants, and has placed each servant onto the evictor queue. Because requests were sent by the client for object identities 1 to 5 (in that order), servant 5 ends up at the head of the queue (at the most-recently used position), and servant 1 ends up at the tail of the queue (at the least-recently used position).

Assume that the client now sends a request for servant 3. In this case, the servant is found on the evictor queue and moved to the head position. The resulting ordering is shown in Figure 16.3.

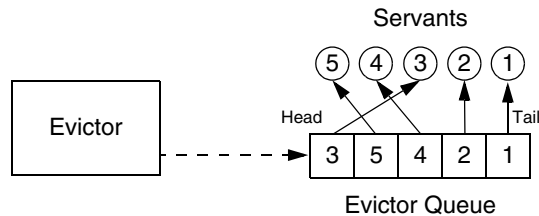


Figure 16.3. The evictor from Figure 16.2 after accessing servant 3.

Assume that the next client request is for object identity 6. The evictor queue is fully populated, so the evictor creates a servant for object identity 6, places that

servant at the head of the queue, and evicts the servant with identity 1 (the least-recently used servant) at the tail of the queue, as shown in Figure 16.4.

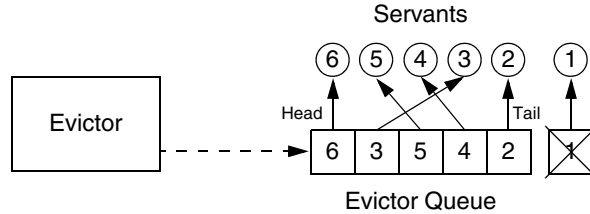


Figure 16.4. The evictor from Figure 16.3 after evicting servant 1.

The evictor pattern combines the advantages of the ASM with the advantages of a default servant: provided that the cache size is sufficient to hold the working set of servants in memory, most requests are served by an already instantiated servant, without incurring the overhead of creating a servant and accessing the database to initialize servant state. By setting the cache size, you can control the trade-off between performance and memory consumption as appropriate for your application.

The following sections show how to implement an evictor in both C++ and Java. (You can also find the source code for the evictor with the code examples for this book in the Ice distribution.)

Creating an Evictor Implementation in C++

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
class EvictorBase : public Ice::ServantLocator {
public:

    EvictorBase(int size = 1000);

    virtual Ice::ObjectPtr locate(const Ice::Current &c,
                                  Ice::LocalObjectPtr &cookie);
    virtual void finished(const Ice::Current &c,
                          const Ice::ObjectPtr &servant,
                          const Ice::LocalObjectPtr &cookie);
    virtual void deactivate(const std::string &category);
```

```
virtual Ice::ObjectPtr add(const Ice::Current &c,
                          Ice::LocalObjectPtr &cookie) = 0;
virtual void evict(const Ice::ObjectPtr &servant,
                  const Ice::LocalObjectPtr &cookie) = 0;

private:
    // ...
};
typedef IceUtil::Handle<EvictorBase> EvictorBasePtr;
```

Note that the evictor has a constructor that sets the size of the queue, with a default argument to set the size to 1000.

The `locate`, `finished`, and `deactivate` functions are inherited from the `ServantLocator` base class; these functions implement the logic to maintain the queue in LRU order and to add and evict servants as needed.

The `add` and `evict` functions are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are pure virtual, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue. Storing the queue position is not strictly necessary—we store the position for efficiency reasons because it allows us to locate a servant’s position in the queue in constant time instead of having to search through the queue in order to maintain its LRU property.

3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The need for the use count deserves some extra explanation: suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

The evictor queue does not store the identity of the servant. Instead, the entries on the queue are iterators into the evictor map. This is useful when the time comes to evict a servant: instead of having to search the map for the identity of the servant to be evicted, we can simply delete the map entry that is pointed at by the iterator at the tail of the queue. We can get away with storing an iterator into the evictor queue as part of the map, and storing an iterator into the evictor map as part of the queue because both `std::list` and `std::map` do not invalidate iterators when we add or delete entries (except for invalidating iterators that point at a deleted entry, of course).

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

The leads to the following definitions in the private section of our evictor:

```

class EvictorBase : public Ice::ServantLocator {
public:
    // ...

private:
    struct EvictorEntry;
    typedef IceUtil::Handle<EvictorEntry> EvictorEntryPtr;

    typedef std::map<Ice::Identity, EvictorEntryPtr> EvictorMap;
    typedef std::list<EvictorMap::iterator> EvictorQueue;

    struct EvictorEntry : public Ice::LocalObject {
        Ice::ObjectPtr servant;
        Ice::LocalObjectPtr userCookie;
        EvictorQueue::iterator pos;
        int useCount;
    };

    struct EvictorCookie : public Ice::LocalObject {
        EvictorEntryPtr entry;
    };
    typedef IceUtil::Handle<EvictorCookie> EvictorCookiePtr;

    EvictorMap _map;
    EvictorQueue _queue;
    Ice::Int _size;

    IceUtil::Mutex _mutex;

    void evictServants();
};

```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. In addition, we use a private `_mutex` data member so we can correctly serialize access to the evictor's data structures.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

The implementation of the constructor is trivial. The only point of note is that we ignore negative sizes:⁵

```

EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}

```

Almost all the action of the evictor takes place in the implementation of `locate`:

```

Ice::ObjectPtr
EvictorBase::locate(const Ice::Current &c,
                    Ice::LocalObjectPtr &cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    // Create a cookie.
    //
    EvictorCookiePtr ec = new EvictorCookie;
    cookie = ec;

    // Check if we have a servant in the map already.
    //
    EvictorMap::iterator i = _map.find(c.id);
    bool newEntry = i == _map.end();
    if(!newEntry) {
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        ec->entry = i->second;
        _queue.erase(ec->entry->pos);
    } else {
        // We do not have an entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map.
        //
        ec->entry = new EvictorEntry;
        ec->entry->servant
            = add(c, ec->entry->userCookie); // Down-call
        if(!ec->entry->servant)
        {
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);
        }
        ec->entry->useCount = 0;
    }
}

```

-
5. We could have stored the size as a `size_t` instead. However, for consistency with the Java implementation, which cannot use unsigned integers, we use `Ice::Int` to store the size.


```

        i = _map.insert(std::make_pair(c.id, ec->entry)).first;
    }

    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(ec->entry->useCount);
    ec->entry->pos = _queue.insert(_queue.begin(), i);

    return ec->entry->servant;
}

```

The first step in `locate` is to lock the `_mutex` data member. This protects the evictor's data structures from concurrent access. The next step is to create the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. The cookie contains a smart pointer to an evictor entry, of type `EvictorEntryPtr`. This is also the value type of our map entries, so we do not store two copies of the same information redundantly—instead, smart pointers ensure that a single copy is shared by both the cookie and the map.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we initialize the cookie's smart pointer with that entry and remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we throw an `ObjectNotExistException` to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, and finally return the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```

void
EvictorBase::finished(const Ice::Current &,
                      const Ice::ObjectPtr &,
                      const Ice::LocalObjectPtr &cookie)
{
    IceUtil::Mutex::Lock lock(_mutex);

    EvictorCookiePtr ec = EvictorCookiePtr::dynamicCast(cookie);

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec->entry->useCount);
    evictServants();
}

```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```

void
EvictorBase::evictServants()
{
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    for (int i = static_cast<int>(_map.size() - _size);
         i > 0; --i) {
        EvictorQueue::reverse_iterator p = _queue.rbegin();
        if ((*p)->second->useCount == 0) {
            evict((*p)->second->servant, (*p)->second->userCookie);
            EvictorMap::iterator pos = *p;
            _queue.erase((*p)->second->pos);
            _map.erase(pos);
        }
    }
}

```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run

time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
void
EvictorBase::deactivate(const std::string& category)
{
    IceUtil::Mutex::Lock lock(_mutex);

    _size = 0;
    evictServants();
}
```

Creating an Evictor Implementation in Java

The evictor we show here is designed as an abstract base class: in order to use it, you derive an object from the `EvictorBase` base class and implement two methods that are called by the evictor when it needs to add or evict a servant. This leads to a class definitions as follows:

```
package Evictor;

public abstract class EvictorBase
    extends Ice.LocalObjectImpl implements Ice.ServantLocator
{
    public
    EvictorBase()
    {
        _size = 1000;
    }

    public
    EvictorBase(int size)
    {
        _size = size < 0 ? 1000 : size;
    }

    public abstract Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie);

    public abstract void
    evict(Ice.Object servant, Ice.LocalObject cookie);

    synchronized public final Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
```

```
        // ...
    }

    synchronized public final void
    finished(Ice.Current c, Ice.Object o, Ice.LocalObject cookie)
    {
        // ...
    }

    synchronized public final void
    deactivate(String category)
    {
        // ...
    }

    // ...

    private int _size;
}
```

Note that the evictor has constructors to set the size of the queue, with a default size of 1000.

The `locate`, `finished`, and `deactivate` methods are inherited from the `ServantLocator` base class; these methods implement the logic to maintain the queue in LRU order and to add and evict servants as needed. The methods are synchronized, so the evictor's internal data structures are protected from concurrent access.

The `add` and `evict` methods are called by the evictor when it needs to add a new servant to the queue and when it evicts a servant from the queue. Note that these functions are abstract, so they must be implemented in a derived class. The job of `add` is to instantiate and initialize a servant for use by the evictor. The `evict` function is called by the evictor when it evicts a servant. This allows `evict` to perform any cleanup. Note that `add` can return a cookie that the evictor passes to `evict`, so you can move context information from `add` to `evict`.

Next, we need to consider the data structures that are needed to support our evictor implementation. We require two main data structures:

1. A map that maps object identities to servants, so we can efficiently decide whether we have a servant for an incoming request in memory or not.
2. A list that implements the evictor queue. The list is kept in LRU order at all times.

The evictor map does not only store servants but also keeps track of some administrative information:

1. The map stores the cookie that is returned from `add`, so we can pass that same cookie to `evict`.
2. The map stores an iterator into the evictor queue that marks the position of the servant in the queue.
3. The map stores a use count that is incremented whenever an operation is dispatched into a servant, and decremented whenever an operation completes.

The last two points deserve some extra explanation.

- The evictor queue must be maintained in least-recently used order, that is, every time an invocation arrives and we find an entry for the identity in the evictor map, we also must locate the servant's identity on the evictor queue and move it to the front of the queue. However, scanning for that entry is inefficient because it requires $O(n)$ time. To get around this, we store an iterator in the evictor map that marks the corresponding entry's position in the evictor queue. This allows us to dequeue the entry from its current position and enqueue it at the head of the queue in $O(1)$ time.

Unfortunately, the various lists provided by `java.util` do not allow us to keep an iterator to a list position without invalidating that iterator as the list is updated. To deal with this, we use a special-purpose linked list implementation, `Evictor.LinkedList`, that does not have this limitation.

`LinkedList` has an interface similar to `java.util.LinkedList` but does not invalidate iterators other than iterators that point at an element that is removed. For brevity, we do not show the implementation of this list here—you can find the implementation in the code examples for this book in the Ice distribution.

- We maintain a use count as part of the map in order to avoid incorrect eviction of servants. Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still

executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*.

However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Finally, our `locate` and `finished` implementations will need to exchange a cookie that contains a smart pointer to the entry in the evictor map. This is necessary so that `finished` can decrement the servant's use count.

This leads to the following definitions in the private section of our evictor:

```
package Evictor;

public abstract class EvictorBase
    extends Ice.LocalObjectImpl implements Ice.ServantLocator
{
    // ...

    private class EvictorEntry
    {
        Ice.Object servant;
        Ice.LocalObject userCookie;
        java.util.Iterator pos;
        int useCount;
    }

    private class EvictorCookie extends Ice.LocalObjectImpl
    {
        public EvictorEntry entry;
    }

    private void evictServants()
    {
        // ...
    }
}
```

```

    }

    private java.util.Map _map = new java.util.HashMap();
    private Evictor.LinkedList _queue = new Evictor.LinkedList();
    private int _size;
}

```

Note that the evictor stores the evictor map, queue, and the queue size in the private data members `_map`, `_queue`, and `_size`. The map key is the identity of the Ice object, and the lookup value is of type `EvictorEntry`. The queue simply stores identities, of type `Ice::Identity`.

The `evictServants` member function takes care of evicting servants when the queue length exceeds its limit—we will discuss this function in more detail shortly.

Almost all the action of the evictor takes place in the implementation of `locate`:

```

synchronized public final Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    // Make a copy of the ID. We need to do this because
    // Ice.Current.id is the same reference on every call: its
    // contents change, but it is the same object every time. We
    // need a copy to insert into the queue and the map.
    //
    Ice.Identity idCopy = new Ice.Identity();
    idCopy.name = c.id.name;
    idCopy.category = c.id.category;

    // Create a cookie.
    //
    EvictorCookie ec = new EvictorCookie();
    cookie.value = ec;

    // Check if we a servant in the map already.
    //
    ec.entry = (EvictorEntry)_map.get(idCopy);
    boolean newEntry = ec.entry == null;
    if(!newEntry) {
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        ec.entry.pos.remove();
    } else {
        // We do not have entry. Ask the derived class to

```

```

        // instantiate a servant and add a new entry to the map.
        //
        ec.entry = new EvictorEntry();
        Ice.LocalObjectHolder cookieHolder
            = new Ice.LocalObjectHolder();
        ec.entry.servant = add(c, cookieHolder); // Down-call
        if(ec.entry.servant == null) {
            throw new Ice.ObjectNotExistException();
        }
        ec.entry.userCookie = cookieHolder.value;
        ec.entry.useCount = 0;
        _map.put(idCopy, ec.entry);
    }

    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(ec.entry.useCount);
    _queue.addFirst(idCopy);
    ec.entry.pos = _queue.iterator();
    ec.entry.pos.next(); // Position the iterator on the element.

    return ec.entry.servant;
}

```

The first step is to create a copy of the object identity that is passed in `Ice.Current`. (As mentioned on page 391, `Ice.Current` and members of `Ice.Current` are assigned to prior to passing them to application code instead of allocating a new object for each call for efficiency reasons.)

Next, the code creates a cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. The cookie contains an evictor entry, of type `EvictorEntry`.

We now look for an existing entry in the evictor map, using the object identity as the key. If we have an entry in the map already, we dequeue the corresponding identity from the evictor queue. (The `pos` member of `EvictorEntry` is an iterator that marks that entry's position in the evictor queue.)

Otherwise, we do not have an entry in the map, so we create a new one and call the `add` method. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we throw an `ObjectNotExistException` to let the Ice run time know that no servant could be found for the current request. If

add succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The final few lines of code increment the entry's use count, add the entry at the head of the evictor queue, and store the entry's position in the queue before returning the servant to the Ice run time.

The implementation of `finished` is comparatively simple. It decrements the use count of the entry and then calls `evictServants` to get rid of any servants that might need to be evicted:

```
synchronized public final void
finished(Ice.Current c, Ice.Object o, Ice.LocalObject cookie)
{
    EvictorCookie ec = (EvictorCookie)cookie;

    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec.entry.useCount);
    evictServants();
}
```

In turn, `evictServants` examines the evictor queue: if the queue length exceeds the evictor's size, the excess entries are scanned. Any entries with a zero use count are then evicted:

```
private void evictServants()
{
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.
    //
    for(int i = _map.size() - _size; i > 0; --i) {
        java.util.Iterator p = _queue.riterator();
        Ice.Identity id = (Ice.Identity)p.next();
        EvictorEntry e = (EvictorEntry)_map.get(id);
        if(e.useCount == 0) {
            evict(e.servant, e.userCookie); // Down-call
            p.remove();
            _map.remove(id);
        }
    }
}
```

The code scans the excess entries, starting at the tail of the evictor queue. If an entry has a zero use count, it is evicted: after calling the `evict` member function

in the derived class, the code removes the evicted entry from both the map and the queue.

Finally, the implementation of `deactivate` sets the evictor size to zero and then calls `evictServants`. This results in eviction of all servants. The Ice run time guarantees to call `deactivate` only once no more requests are executing in an object adapter; as a result, it is guaranteed that all entries in the evictor will be idle and therefore will be evicted.

```
synchronized public final void
deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

Using Servant Evictors

Using a servant evictor is simply a matter of deriving a class from `EvictorBase` and implementing the `add` and `evict` methods. You can turn a servant locator into an evictor by simply taking the code that you wrote for `locate` and placing it into `add`—`EvictorBase` then takes care of maintaining the cache in least-recently used order and evicting servants as necessary. Unless you have clean-up requirements for your servants (such as closing network connections or database handles), the implementation of `evict` can be left empty.

One of the nice aspects of evictors is that you do not need to change anything in your servant implementation: the servants are ignorant of the fact that an evictor is in use. This makes it very easy to add an evictor to an already existing code base with little disturbance of the source code.

Evictors can provide substantial performance improvements over default servants: especially if initialization of servants is expensive (for example, because servant state must be initialized by reading from a network), an evictor performs much better than a default servant, while keeping memory requirements low.

As we will see in XREF, evictors can also be useful to get rid of servants that have been abandoned by clients.

16.8 The Ice::Context Parameter

In Sections 6.11.1 and 8.11.1, methods on a proxy are overloaded with a trailing parameter of type `const Ice::Context &` (C++) or `java.util.Map` (Java). The Slice definition of this parameter is as follows:

```
module Ice {
    local dictionary<string, string> Context;
};
```

As you can see, a context is a dictionary that maps strings to strings or, conceptually, a context is a collection of name–value pairs. The contents of this dictionary (if any) are implicitly marshaled with every request to the server, that is, if the client populates a context with a number of name–value pairs and uses that context for an invocation, the name–value pairs that are sent by the client are available to the server.

On the server side, the operation implementation can access the received Context via the `ctx` member of the `Ice: : Current` parameter (see Section 16.5) and extract the name–value pairs that were sent by the client.

16.8.1 Passing Context Explicitly

Contexts provide a means of sending an unlimited number of parameters from client to server without having to mention those parameters in the signature of an operation. For example, consider the following definition:

```
struct Address {
    // ...
};

interface Person {
    string setAddress(Address a);
    // ...
};
```

Assuming that the client has a proxy to a `Person` object, it could do something along the following lines:

```
PersonPrx p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

In Java, the same code would look as follows:

```

Person p = ...;
Address a = ...;

Ice.Context ctx = new java.util.HashMap();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);

```

On the server side, we can extract the policy value set by the client to influence how the implementation of `setAddress` works. A C++ implementation might look like:

```

void
PersonI::setAddress(const Address &a, const Ice::Current &c)
{
    Ice::Context::const_iterator i = c.ctx.find("write policy");
    if (i != c.ctx.end() && i->second == "immediate") {

        // Update the address details and write through to the
        // data base immediately...

    } else {

        // Write policy was not set (or had a bad value), use
        // some other database write strategy.

    }
}

```

For this example, the server examines the value of the context with the key "write policy" and, if that value is "immediate", writes the update sent by the client straight away; if the write policy is not set or contains a value that is not recognized, the server presumably applies a more lenient write policy (such as caching the update in memory and writing it later). The Java version of the operation implementation is essentially identical, so we do not show it here.

16.8.2 Passing Context Implicitly

Instead of passing a context explicitly with an invocation, you can also use an *implicit* context. Implicit contexts allow you to set a context on a particular proxy once and, thereafter, whenever you use that proxy to invoke an operation, the previously-set context is sent with each invocation. The proxy base class provides a member function, `ice_newContext`, to do this:

```

namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx
                ice_newContext(const Ice::Context &) const;
            // ...
        };
    }
}

```

For Java, the corresponding function is:

```

package Ice;

public interface ObjectPrx {
    ObjectPrx ice_newContext(java.util.Map newContext);
    // ...
}

```

`ice_newContext` creates a new proxy that implicitly stores the passed context. Note that the return type of `ice_newContext` is `ObjectPrx`, that is, before you can use the newly-created proxy, you must down-cast it to the correct type. For example, in C++:

```

Ice::Context ctx;
ctx["write policy"] = "immediate";

PersonPrx p1 = ...;
PersonPrx p2 = PersonPrx::uncheckedCast(p1->ice_newContext(ctx));

Address a = ...;

p1->setAddress(a);           // Sends no context

p2->setAddress(a);           // Sends ctx implicitly

Ice::Context ctx2;
ctx2["write policy"] = "delayed";

p2->setAddress(a, ctx2); // Sends ctx2

```

As the example illustrates, once we have created the `p2` proxy, any invocation via `p2` implicitly sends the previously-set context. The final line of the example illustrates that it is possible to explicitly send a context for an invocation even if the

proxy has an implicit context—an explicit context always overrides any implicit context.

16.8.3 Retrieving Implicit Context

You can retrieve the implicit context of a proxy by calling `ice_getContext`. The call returns the currently-set context of the proxy. (If a proxy has no implicit context, the returned dictionary is empty.) For C++, the signature is:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::Context ice_getContext() const;
            // ...
        };
    }
}
```

For Java, the signature is:

```
package Ice;

public interface ObjectPrx {
    java.util.Map ice_getContext();
    // ...
}
```

16.8.4 Context Use Cases

The purpose of `Ice::Context` is to permit services to be added to Ice that require some contextual information with every request. Such contextual information can be used by services such as a transaction service (to provide the context of a currently established transaction) or a security service (to provide an authorization token to the server). IceStorm (see Chapter 26) uses the context to provide an optional cost parameter to the service that influences how the service propagates messages to down-stream subscribers.

In general, services that require such contextual information can be implemented much more elegantly using contexts because this hides explicit Slice parameters that would otherwise have to be supplied by the application programmer with every call.

In addition, contexts, because they are optional, permit a single Slice definition to apply to implementations that use the context as well as to implementations

that do not use it. In this way, to add transactional semantics to an existing service, you do not need to modify the Slice definitions to add extra parameters to all operations. This not only is convenient for clients but also prevents the type system from being split into two halves: without contexts, we would need different Slice definitions for transactional and non-transactional implementations of what, conceptually, is a single service.

Finally, implicit contexts permit context information to be passed by through intermediate parts of a system without cooperation of those intermediate parts. For example, suppose you set an implicit context on a proxy and then pass that proxy to another system component. When that component uses the proxy to invoke an operation, the implicit context will still be sent. In other words, implicit contexts allow you to transparently propagate information via intermediaries that are ignorant of the presence of any context.

16.8.5 A Word of Warning

Contexts are a powerful mechanism for transparent propagation of context information, *if used correctly*. In particular, you may be tempted to use contexts as a means of versioning an application as it evolves over time. For example, version 2 of your application may accept two parameters on an operation that, in version 1, used to accept only a single parameter. Using contexts, you could supply the second parameter as a name–value pair to the server and avoid changing the Slice definition of the operation in order to maintain backward compatibility.

We *strongly* urge you to resist any temptation to use contexts in this manner. The strategy is fraught with problems:

- Missing context

There is nothing that would compel a client to actually send a context when the server expects to receive a context: if a client forgets to send a context, the server, somehow, has to make do without it (or throw an exception).

- Missing or incorrect keys

Even if the client does send a context, there is no guarantee that it has set the correct key. (For example, a simple spelling error can cause the client to send a value with the wrong key.)

- Incorrect values

The value of a context is a string, but the application data that is to be sent might be a number, or it might be something more complex, such as a structure with several members. This forces you to encode the value into a string

and decode the value again on the server side. Such parsing is tedious and error prone, and far less efficient than sending strongly-typed parameters. In addition, the server has to deal with string values that fail to decode correctly (for example, because of a encoding error made by the client).

None of the preceding problems can arise if you use proper Slice parameters: parameters cannot be accidentally omitted and they are strongly typed, making it much less likely for the client to accidentally send a meaningless value.

If you are concerned about how to evolve an application over time without breaking backward compatibility, Ice offers a number of mechanisms, such as facets, that are better suited to this task (see XREF). Contexts are meant to be used to transmit simple tokens (such as a transaction identifier) for services that cannot be reasonably implemented without them; you should restrict your use of contexts to that purpose and resist any temptation to use contexts for any other purpose.

16.9 Invocation Timeouts

Remote invocations made by clients are synchronous and blocking: the invocation does not complete on the client side until the server has finished processing it. Occasionally, it is useful to be able to force an invocation to terminate after some time, even if it has not completed. Proxies provide the `ice_timeout` operation for this:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_timeout(Ice::Int t) const;
            // ...
        };
    }
}
```

For Java, the corresponding method is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_timeout(int t);
    // ...
}
```


The `ice_timeout` operation creates a proxy with a timeout from an existing proxy. For example:

```

FileSystem::FilePrx myFile = ...;
FileSystem::FilePrx timeoutFile
    = FileSystem::FilePrx::uncheckedCast(
        myFile->ice_timeout(5000));

try {
    Lines text = timeoutFile->read();    // Read with timeout
} catch(const Ice::TimeoutException &) {
    cerr << "invocation timed out" << endl;
}

Lines text = myFile->read();            // Read without timeout

```

The parameter to `ice_timeout` determines the timeout value in milliseconds. A value of 0 indicates no timeout. In the preceding example, the timeout is set to five seconds; if an invocation of `read` via the `timeoutFile` proxy does not complete within five seconds, the operation terminates with an `Ice::TimeoutException`. On the other hand, invocations via the `myFile` proxy are unaffected by the timeout, that is, `ice_timeout` sets the timeout on a per-proxy basis.

The timeout value set on a proxy affects all networking operations: reading and writing of data as well as opening and closing of connections. If any of these operations does not complete within the timeout, the client receives an exception. Timeouts that expire during reading or writing of data are indicated by a `TimeoutException`. For opening and closing of connections, the Ice run time reserves separate exceptions:

- `ConnectTimeoutException`

This exception indicates that a connection could not be established within the specified time.

- `CloseTimeoutException`

This exception indicates that a connection could not be closed within the specified time.

The Ice run time provides the **`Ice.Override.Timeout`** property (see Appendix C). This property allows you to replace the default timeout value of (no timeout) with a different value. Setting this property affects all proxies for which you have not otherwise set a timeout with `ice_timeout`.

Note that timeouts are “soft” timeouts, in the sense that they are not precise, real-time timeouts. (The precision is limited by the capabilities of the underlying

operating system.) You should also be aware that timeouts are considered fatal error conditions by the Ice run time: for example, timeouts result in connection closure on the client side. Timeouts are meant to be used to prevent a client from blocking indefinitely in case something has gone wrong with the server; they are not meant as a mechanism to routinely abort requests that take longer than intended.

16.10 Oneway Invocations

As mentioned in Chapter 2, the Ice run time supports *oneway* invocations. A oneway invocation is sent on the client side by writing the request to the client's local transport buffers; the invocation completes and returns control to the application code as soon as it has been accepted by the local transport. Of course, this means that a oneway invocation is unreliable: it may never be sent (for example, because of a network failure) or it may not be accepted in the server (for example, because the target object does not exist). If anything goes wrong, the client-side application code does not receive any notification of the failure; the only errors that are reported to the client are local errors that occur on the client side during call invocation (such as failure to establish a connection, for example).

Oneway invocations are received and processed on the server side like any other incoming request. In particular, there is no way for the server-side application code to distinguish a oneway invocation from a twoway invocation, that is, oneway invocation is transparent on the server side.

Oneway invocations do not incur any return traffic from the server to the client: the server never sends a reply message in response to a oneway invocation (see Chapter 18). This means that oneway invocations can result in large efficiency gains, especially for large numbers of small messages, because the client does not have to wait for the reply to each message to arrive before it can send the next message.

In order to be able to invoke an operation as oneway, two conditions must be met:

- The operation must have a `void` return type, must not have any out-parameters, and must not have an exception specification.

This requirement reflects the fact that the server does not send a reply for a oneway invocation to the client: without such a reply, there is no way to return any values or exceptions to the client.

If you attempt to invoke an operation that returns values to the client as a oneway operation, the Ice run time throws a `TwoWayOnlyException`.

- The proxy on which the operation is invoked must support a stream-oriented transport (such as TCP or SSL).

Oneway invocations require a stream-oriented transport. (To get something like a oneway invocation for datagram transports, you need to use a datagram invocation—see Section 16.11.)

If you attempt to create a oneway proxy for an object that does not offer a stream-oriented transport, the Ice run time throws a `NoEndpointException`.

Despite their theoretical unreliability, in practice, oneway invocations are reliable (but not infallible): they are sent via a stream-oriented transport, so they cannot get lost unless the connection fails entirely. In particular, the transport uses its usual flow control, so the client cannot overrun the server with messages. On the client-side, the Ice run time will block if the client's transport buffers fill up, so the client-side application code cannot overrun its local transport.

Consequently, oneway invocations normally do not block the client-side application code and return immediately, provided that the client does not consistently generate messages faster than the server can process them. If the rate at which the client invokes operations exceeds the rate at which the server can process them, the client-side application code will eventually block in an operation invocation until sufficient room is available in the client's transport buffers to accept the invocation.

Regardless of whether the client exceeds the rate at which the server can process incoming oneway invocations, the execution of oneway invocations in the server proceeds asynchronously: the client's invocation completes before the message even arrives at the server.

One thing you need to keep in mind about oneway invocations is that they may appear to be reordered in the server: because oneway invocations are sent via a stream-oriented transport, they are guaranteed to be received in the order in which they were sent. However, the server dispatches each invocation in its own thread; because threads are scheduled preemptively, this may cause an invocation sent

later by the client to be dispatched and executed before an invocation that was sent earlier.

For this reason, oneway invocations are usually best suited to simple updates that are otherwise stateless (that is, do not depend on the surrounding context or the state established by previous invocations).

Creating Oneway Proxies

Ice selects between twoway, oneway, and datagram (see Section 16.11) invocations via the proxy that is used to invoke the operation. By default, all proxies are created as twoway proxies. To invoke an operation as oneway, you must create a separate proxy for oneway dispatch from a twoway proxy.

For C++, all proxies are derived from a common `IceProxy::Ice::Object` class (see Section 6.11.1). The proxy base class contains a method to create a oneway proxy, called `ice_oneway`:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_oneway() const;
            // ...
        };
    }
}
```

For Java, proxies are derived from the `Ice.ObjectPrx` interface (see Section 8.11.2) and the definition of `ice_oneway` is:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_oneway();
    // ...
}
```

We can call `ice_oneway` to create a oneway proxy and then use the proxy to invoke an operation as follows. (We show the C++ version here—the Java version is analogous.)

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a oneway proxy.
//
Ice::ObjectPrx oneway;
```

```

try {
    oneway = o->ice_oneway();
} catch (const Ice::NoEndPointException &) {
    cerr << "No endpoint for oneway invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx onewayPerson = PersonPrx::uncheckedCast(oneway);

// Invoke an operation as oneway.
//
try {
    onewayPerson->someOp();
} catch (const Ice::TwowayOnlyException &) {
    cerr << "someOp() is not oneway" << endl;
}

```

Note that we use an `uncheckedCast` to down-cast the proxy from `ObjectPrx` to `PersonPrx`: for a oneway proxy, we cannot use a `checkedCast` because a `checkedCast` requires a reply from the server but, of course, a oneway proxy does not permit that reply. If instead you want to use a safe down-cast, you can first down-cast the twoway proxy to the actual object type and then obtain the oneway proxy:

```

Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Safe down-cast to actual type.
//
PersonPrx person = PersonPrx::checkedCast(o);

if (person) {
    // Get a oneway proxy.
    //
    PersonPrx onewayPerson;
    try {
        onewayPerson
            = PersonPrx::uncheckedCast(person->ice_oneway());
    } catch (const Ice::NoEndPointException &) {
        cerr << "No endpoint for oneway invocations" << endl;
    }

    // Invoke an operation as oneway.
    //
    try {

```

```

        onewayPerson->someOp();
    } catch (const Ice::TwowayOnlyException &) {
        cerr << "someOp() is not oneway" << endl;
    }
}

```

Note that, while the second version of this code is somewhat safer (because it uses a safe down-cast), it is also slower (because the safe down-cast incurs the cost of an additional twoway message).

16.11 Datagram Invocations

Datagram invocations are the equivalent of oneway invocations for datagram transports. As for oneway invocations, datagram invocations can be sent only for operations that have a `void` return type and do not have out-parameters or an exception specification. (Attempts to use a datagram invocation with an operation that does not meet these criteria result in a `TwowayOnlyException`.) In addition, datagram invocations can only be used if the proxy's endpoints include at least one UDP transport; otherwise, the Ice run time throws a `NoEndpointException`.

The semantics of datagram invocations are similar to oneway invocations: no return traffic flows from the server to the client and proceed asynchronously with respect to the client; a datagram invocation completes as soon as the client's transport has accepted the invocation into its buffers. However, datagram invocations have additional error semantics:

- Individual invocations may be lost or received out of order.

On the wire, datagram invocations are sent as true datagrams, that is, individual datagrams may be lost, or arrive at the server out of order. As a result, not only may operations be dispatched out of order, an individual invocation out of a series of invocations may be lost. (This cannot happen for oneway invocations because, if a connection fails, *all* invocations are lost once the connection breaks down.)

- UDP packets may be duplicated by the transport.

Because of the nature of UDP routing, it is possible for datagrams to arrive in duplicate at the server. This means that, for datagram invocations, Ice does *not* guarantee at-most-once semantics (see page 14): if UDP datagrams are duplicated, the same invocation may be dispatched more than once in the server.

- UDP packets are limited in size.

The maximum size of an IP datagram is 65,535 bytes. Of that, the IP header consumes 20 bytes, and the UDP header consumes 8 bytes, leaving 65,507 bytes as the maximum payload. If the marshaled form of an invocation, including the Ice request header (see Chapter 18) exceeds that size, the invocation is lost. (Exceeding the size limit for a UDP datagram is indicated to the application by a `DatagramLimitException`.)

Because of their unreliable nature, datagram invocations are best suited to simply update messages that are otherwise stateless. In addition, due to the high probability of loss of datagram invocations over wide area networks, you should restrict use of datagram invocations to local area networks, where they are less like to be lost. (Of course, regardless of the probability of loss, you must design your application such that it can tolerate lost or duplicated messages.)

Creating Datagram Proxies

To create a datagram proxy, you must call `ice_datagram` on the proxy, for example:

```
Ice::ObjectPrx o = communicator->stringToProxy(/* ... */);

// Get a datagram proxy.
//
Ice::ObjectPrx datagram;
try {
    datagram = o->ice_datagram();
} catch (const Ice::NoEndPointException &) {
    cerr << "No endpoint for datagram invocations" << endl;
}

// Down-cast to actual type.
//
PersonPrx datagramPerson = PersonPrx::uncheckedCast(datagram);

// Invoke an operation as a datagram.
//
try {
    datagramPerson->someOp();
} catch (const Ice::TwowayOnlyException &) {
    cerr << "someOp() is not oneway" << endl;
}
```

As for the oneway example in Section 16.10, you can choose to first do a safe down-cast to the actual type of interface and then obtain the datagram proxy, rather than relying on an unsafe down-cast, as shown here. However, doing so may be disadvantageous for two reasons:

- Safe down-casts are sent via a stream-oriented transport. This means that using a safe down-cast will result in opening a connection for the sole purpose of verifying that the target object has the correct type. This is expensive if all the other traffic to the object is sent via datagrams.
- If the proxy does not offer a stream-oriented transport, the `checkedCast` fails with a `NoEndpointException`, so you can use this approach only for proxies that offer both a UDP endpoint and a TCP/IP and/or SSL endpoint.

16.12 Batched Invocations

Oneway and datagram invocations are normally sent as a separate message, that is, the Ice run time sends the oneway or datagram invocation to the server immediately, as soon as the client makes the call. If a client sends a number of oneway or datagram invocations in succession, the client-side run time traps into the OS kernel for each message, which is expensive. In addition, each message is sent with its own message header (see Chapter 18), that is, for n messages, the bandwidth for n message headers is consumed. In situations where a client sends a number of oneway or datagram invocations, the additional overhead can be considerable.

To avoid the overhead of sending many small messages, you can send oneway and datagram invocations in a batch: instead of being sent as a separate message, a batch invocation is placed into a client-side buffer by the Ice run time. Successive batch invocations are added to the buffer and accumulated on the client side until the client explicitly flushes them. The relevant APIs are part of the proxy interface:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_batchOneway() const;
            Ice::ObjectPrx ice_batchDatagram() const;
```



```

        // ...
    };
}

```

The `ice_batchOneway` and `ice_batchDatagram` methods convert a proxy to a batch proxy. Once you obtain a batch proxy, messages sent via that proxy are buffered in the client-side run time instead of being sent immediately. Once the client has invoked one or more operations on batch proxies, it can explicitly flush the batched invocations by calling `Communicator::flushBatchRequests`:

```

module Ice {
    local interface Communicator {
        void flushBatchRequests();
        // ...
    };
};

```

This causes the batched messages to be sent “in bulk”, preceded by a single message header. On the server side, batched messages are dispatched by a single thread, in the order in which they were written into the batch. This means that messages from a single batch cannot appear to be reordered in the server. Moreover, either all messages in a batch are delivered or none of them. (This is true even for batched datagrams.)

For batched datagram invocations, you need to keep in mind that, if the data for the invocations in a batch substantially exceeds the PDU size of the network, it becomes increasingly likely for an individual UDP packet to get lost due to fragmentation. In turn, loss of even a single packet causes the entire batch to be lost. For this reason, batched datagram invocations are most suitable for simple interfaces with a number of operations that each set an attribute of the target object (or interfaces with similar semantics). (Batched oneway invocations do not suffer from this risk because they are sent over stream-oriented transports, so individual packets cannot be lost.)

Batched invocations are more efficient if you also enable compression for the transport: many isolated and small messages are unlikely to compress well, whereas batched messages are likely to provide better compression because the compression algorithm has more data to work with.⁶

16.13 Testing Proxies for Dispatch Type

The proxy interface offers a number of operations that allow you test the dispatch mode of a proxy. (The Java version of these methods is analogous, so we do not show it here.)

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            bool ice_isTwoway() const;
            bool ice_isOneway() const;
            bool ice_isDatagram() const;
            bool ice_isBatchOneway() const;
            bool ice_isBatchDatagram() const;
            // ...
        };
    }
}
```

These operations allow you to test the dispatch mode of an individual proxy.

16.14 The Ice::Logger Interface

Depending on the setting of various properties (see Chapter 14), the Ice run time produces trace, warning, or error messages. These messages are written via the Ice::Logger interface:

```
module Ice {
    local interface Logger {
        void trace(string category, string message);
        void warning(string message);
        void error(string message);
    };
};
```

6. Regardless of whether you used batched messages or not, you should enable compression only on lower-speed links. For high-speed LAN connections, the CPU time spent doing the compression and decompression is typically longer than the time it takes to just transmit the uncompressed data.

A default logger is instantiated when you create a communicator. The default logger logs to the standard error output. The `trace` operation accepts a category parameter in addition to the error message; this allows you to separate trace output from different subsystems by sending the output through a filter.

You can set and get the logger that is attached to a communicator:

```
module Ice {  
    local interface Communicator {  
        void setLogger(Logger log);  
        Logger getLogger();  
    };  
};
```

The `setLogger` operation installs a different logger for a communicator. The Ice run time caches the logger that is set for a communicator. This means that you should set the logger once, after you create the communicator and not change it thereafter. (If you change the logger later, some messages may appear via the old logger and some messages via the new logger.) The `getLogger` operation returns the current logger.

Changing the Logger object that is attached to a communicator allows you to integrate Ice messages into your own message handling system. For example, for a complex application, you might have an already existing logging framework. To integrate Ice messages into that framework, you can create your own Logger implementation that logs messages to the existing framework.

When you destroy a communicator, its logger is *not* destroyed. This means that you can safely use a logger even beyond the lifetime of its communicator.

For convenience, Ice provides two platform-specific logger implementations: one that logs its messages to the Unix **syslog** facility, and another that uses the Windows event log. You can activate the **syslog** implementation by setting the `Ice.UseSyslog` property, and the Windows event log implementation by setting the `Ice.UseEventLog` property. Both implementations use the value of the `Ice.ProgramName` property to identify the application to the log subsystem.

The **syslog** logger implementation is available in both C++ and Java, whereas the Windows event log implementation is only available in C++. See Section 10.3.2 for additional information on using these logger implementations in C++.

16.15 The Ice::Stats Interface

The Ice run time reports bytes sent and received over the wire on every operation invocation via the Ice::Stats interface:

```
module Ice {
    local interface Stats {
        void bytesSent(string protocol, int num);
        void bytesReceived(string protocol, int num);
    };

    local interface Communicator {
        setStats(Stats st);
        // ...
    };
};
```

The Ice run time calls bytesReceived as it reads from the network and bytesSent as it writes to the network. A very simple implementation of the Stats interface could look like the following:

```
class MyStats : public virtual Ice::Stats {
public:
    virtual void bytesSent(const string &prot, Ice::Int num)
    {
        cerr << prot << ": sent " << num << "bytes" << endl;
    }

    virtual void bytesReceived(const string &prot, Ice::Int)
    {
        cerr << prot << ": received " << num << "bytes" << endl;
    }
};
```

To register your implementation, you must instantiate the MyStats object and call setStats on the communicator:

```
Ice::StatsPtr stats = new MyStats;
communicator->setStats(stats);
```

You can install a Stats object on either the client or the server side (or both). Here is some example output produced by installing a MyStats object in a simple server:

```
tcp: received 14 bytes
tcp: received 32 bytes
tcp: sent 26 bytes
tcp: received 14 bytes
tcp: received 33 bytes
tcp: sent 25 bytes
...
```

In practice, your Stats implementation will probably be a bit more sophisticated: for example, the object can accumulate statistics in member variables and make the accumulated statistics available via member functions, instead of simply printing everything to the standard error output.

16.16 Location Transparency

One of the useful features of the Ice run time is that is *location transparent*: the client does not need to know where the implementation of an Ice object resides; an invocation on an object automatically is directed to the correct target, whether the object is implemented in the local address space, in another address space on the same machine, or in another address space on a remote machine. Location transparency is important because it allows us to change the location of an object implementation without breaking client programs and, by using IcePack (see Chapter 20), addressing information such as domain names and port numbers can be externalized so they do not appear in stringified proxies.

For invocations that cross address space boundaries (or more accurately, cross communicator boundaries), the Ice run time dispatches requests via the appropriate transport. However, for invocations that are via proxies for which the proxies and the servants that process the invocation share the same communicator (so-called *collocated* invocations), the Ice run, by default, does not send the invocation via the transport specified in the proxy. Instead, collocated invocations are short-cut inside the Ice run time and dispatched directly.⁷

The reason for this is efficiency: if collocated invocations were sent via TCP/IP, for example, invocations would still be sent via the operating system kernel (using the back plane instead of a network) and would incur the full cost of creating TCP/IP connections, marshaling requests into packets, trapping in and

7. Note that if the proxy and the servant do not use the same communicator, the invocation is *not* collocated, even though caller and callee are in the same address space.

out of the kernel, and so on. By optimizing collocated requests, much of this overhead can be avoided, so collocated invocations are almost as fast as a local function call.

For efficiency reasons, collocated invocations are not completely location transparent, that is, a collocated call has semantics that differ in some ways from calls that cross address-space boundaries. Specifically, collocated invocations differ from ordinary invocations in the following respects:

- Collocated invocations are dispatched in the calling thread instead of being dispatched by a separate thread taken from the server's thread pool.
- The object adapter holding state is ignored: collocated invocations proceed normally even if the target object's adapter is in the holding state.
- For collocated invocations, classes and exceptions are never sliced. Instead, the receiver always receives a class or exception as the derived type that was sent by the sender.
- If a collocated invocation throws an exception that is not in an operation's exception specification, the original exception is raised in the client instead of `UnknownUserException`. (This applies to the C++ mapping only.)
- Asynchronous method invocation (AMI) and asynchronous method dispatch (AMD) cannot be used with collocated invocations.
- Class factories are ignored for collocated invocations.
- Timeouts on invocations are ignored.

In practice, these differences rarely matter. The most likely cause of surprises with collocated invocations is dispatch in the calling thread, that is, a collocated invocation behaves like a local, synchronous procedure call. This can cause problems if, for example, the calling thread acquires a lock that an operation implementation tries to acquire as well: unless you use recursive mutexes (see Chapter 15), this will cause deadlock.

16.17 A Comparison of the Ice and CORBA Run Time

The CORBA equivalent of the server-side functionality of Ice is the Portable Object Adapter (POA). The most striking difference between Ice and the POA is the simplicity of the Ice APIs: Ice is just as fully featured as the POA but achieves this functionality with much simpler interfaces and far fewer operations. Here are a few of the more notable differences:

- Ice object adapters are not hierarchical, whereas POAs are arranged into a hierarchy. It is unclear why CORBA chose to put its adapters into a hierarchy. The hierarchy complicates the POA interfaces but the feature does not provide any apparent benefit: the inheritance of object adapters has no meaning. In particular, POA policies are *not* inherited from the parent adapter. POA hierarchies can be used to control the order of destruction of POAs. However, it is simpler and easier to explicitly destroy adapters in the correct order, as is done in Ice.
- The POA uses a complex policy mechanism to control the behavior of object adapters. The policies can be combined in numerous ways, despite the fact that most combinations do not make sense. This not only is a frequent source of programming errors, but also complicates the API with additional exception semantics for many of its operations.
- The CORBA run time does not provide access to the ORB object (the equivalent of the Ice communicator) during method dispatch. Yet, access to the ORB object is frequently necessary inside operation implementations. As a result, programmers are forced to keep the ORB object in global variable and, if multiple ORBs are used, there is no way to identify the correct ORB for a particular request. The Ice run time eliminates these problems by always providing access to the communicator via the adapter that is passed as part of the Current object.
- The POA attaches implementation techniques to object adapters. For example, an object adapter that uses a servant locator cannot also use an active servant map.

The POA also distinguishes between servant locators (which are similar to Ice servant locators) and servant activators (which work like the incrementally initializing servant locator in Section 16.7.1). Yet, there is no need to distinguish the two concepts: a servant activator is simply a special case of a servant locator that can be implemented trivially.

Similarly, default servants must be registered with a POA by making a special API call when, in fact, there is no need to cater for default servants as a separate concept. As shown in Section 16.7.2, with Ice, you can use a trivial servant locator to achieve the same effect.

- The POA uses separate POA manager objects to control adapter states. This not only complicates the APIs considerably, it also makes it possible to combine hierarchies of object adapters with groupings of POA managers in meaningless ways, leading to undefined behavior. Ice does not use separate

objects to control adapter states and so eliminates the associated complexities without loss of functionality.

- The POA interfaces have the notion of a default Root POA that is used unless the programmer overrides it explicitly. This misfeature is a frequent source of errors, due to inappropriate policies that were chosen for the Root POA. (Users of CORBA will probably have been bitten by implicit activation of objects on the Root POA, instead of the intended POA.)
- The POA provides the concept of implicit activation as well as the notion of system-generated object identities as part of the object adapter policies. This design is a frequent source of programming errors because it leads to many implicit activities behind the scenes, some of them with surprising side effects. (It is sad to see that all this complexity was added to avoid a single line of code during object activation.) Ice does not provide a notion of implicit activation or implicit generation of object identities. Instead, servants are given an identity explicitly and are activated explicitly, which avoids the complexity and confusion.
- CORBA has no notion of datagrams, or of batched invocations, both of which can provide substantial performance gains.
- CORBA provides no standardized way to integrate ORB messages into existing logging frameworks and does not provide access to network statistics.

In summary, the Ice run time provides all the functionality of the POA (and more) with an API that is a fraction in size. By cleanly separating orthogonal concepts and by providing a minimal but sufficient API, Ice not only provides a simpler and easier-to-use API, but also results in binaries that are smaller in size. This not only reduces the memory requirements of Ice binaries, but also contributes to better performance due to reduced working set size.

16.18 Summary

In this chapter, we explored the server-side run time in detail. Communicators are the main handle to the Ice run time. They provide access to a number of run time resources and allow you to control the life cycle of a server. Object adapters provide a mapping between abstract Ice objects and concrete servants. Various implementation techniques are at your disposal to control the trade-off between performance and scalability; in particular, servant locators are a central mecha-

nism that permits you to choose an implementation technique that matches the requirements of your application.

Ice provides both oneway and datagram invocations. These provide performance gains in situations where an application needs to provide numerous stateless updates. Batching such invocations permits you to increase performance even further.

The Ice logging mechanism is user extensible, so you can integrate Ice message into arbitrary logging frameworks, and the `Ice::Stats` interface permits you to collect statistics for network bandwidth consumption.

Finally, even though Ice is location transparent, in the interest of efficiency, collocated invocations do not behave in all respects like remote invocations. You need to be aware of these differences, especially for applications that are sensitive to thread context.

Chapter 17

Asynchronous Programming

17.1 Chapter Overview

This chapter describes the asynchronous programming facilities in Ice. Section 17.2 gives a brief overview of the capabilities and demonstrates how to modify Slice definitions to enable asynchronous support in language mappings. The client-side facilities are presented in Section 17.3 and are followed by a discussion of the server-side facilities in Section 17.4.

17.2 Introduction

Modern middleware technologies attempt to ease the programmer's transition to distributed application development by making remote invocations as easy to use as traditional method calls: a method is invoked on an object and, when the method completes, the results are returned or an exception is raised. Of course, in a distributed system the object's implementation may reside on another host, and consequently there are some semantic differences that the programmer must be aware of, such as the overhead of remote invocations and the possibility of network-related errors. Despite those issues, the programmer's experience with object-oriented programming is still relevant, and this *synchronous* programming

model, in which the calling thread is blocked until the operation returns, is familiar and easily understood.

Ice is inherently an asynchronous middleware platform that simulates synchronous behavior for the benefit of applications (and their programmers). When an Ice application makes a synchronous twoway invocation on a proxy for a remote object, the operation's parameters are marshaled into a message that is written to a transport, and the calling thread is blocked in order to simulate a synchronous method call. Meanwhile, the Ice run-time operates in the background, processing messages until the desired reply is received and the calling thread can be unblocked to unmarshal the results.

There are many cases, however, in which the blocking nature of synchronous programming is too restrictive. For example, the application may have useful work it can do while it awaits the response to a remote invocation; using a synchronous invocation in this case forces the application to either postpone the work until the response is received, or perform this work in a separate thread. When neither of these alternatives are acceptable, the asynchronous facilities provided with Ice are an effective solution for improving performance and scalability, or simplifying complex application tasks.

17.2.1 Asynchronous Method Invocation

Asynchronous Method Invocation (AMI) is the term used to describe the client-side support for the asynchronous programming model. Using AMI, a remote invocation does not block the calling thread while the Ice run-time awaits the reply. Instead, the calling thread can continue its activities, and the application is notified by the Ice run-time when the reply eventually arrives. Notification occurs via a callback to an application-supplied programming-language object¹. AMI is described in detail in Section 17.3.

17.2.2 Asynchronous Method Dispatch

The number of simultaneous synchronous requests a server is capable of supporting is limited by the size of the Ice run-time's server thread pool (see Section 15.3). If all of the threads are busy dispatching long-running operations,

1. Polling for a response is not supported by the Ice run-time, but it can be implemented easily by the application if desired.

then no threads are available to process new requests and therefore clients may experience an unacceptable lack of responsiveness.

Asynchronous Method Dispatch (AMD), the server-side equivalent of AMI, addresses this scalability issue. Using AMD, a server can receive a request but then suspend its processing in order to release the dispatch thread as soon as possible. When processing resumes and the results are available, the server sends a response explicitly using a callback object provided by the Ice run-time.

In practical terms, an AMD operation typically queues the request data (i.e., the callback object and operation arguments) for later processing by an application thread (or thread pool). In this way, the server minimizes the use of dispatch threads and becomes capable of efficiently supporting thousands of simultaneous clients.

An alternate use case for AMD is an operation that requires further processing after completing the client's request. In order to minimize the client's delay, the operation returns the results while still in the dispatch thread, and then continues using the dispatch thread for additional work.

See Section 17.4 for more information on AMD.

17.2.3 Controlling Code Generation using Metadata

A programmer indicates a desire to use an asynchronous model (AMI, AMD, or both) by annotating Slice definitions with metadata (Section 4.17). The programmer can specify this metadata at two levels: for an interface or class, or for an individual operation. If specified for an interface or class, then asynchronous support is generated for all of its operations. Alternatively, if asynchronous support is needed only for certain operations, then the generated code can be minimized by specifying the metadata only for those operations that require it.

Synchronous invocation methods are always generated in a proxy; specifying AMI metadata merely adds asynchronous invocation methods. In contrast, specifying AMD metadata causes the synchronous dispatch methods to be *replaced* with their asynchronous counterparts. This semantic difference between AMI and AMD is ultimately practical: it is beneficial to provide a client with synchronous and asynchronous versions of an invocation method, but doing the equivalent in a server would require the programmer to implement both versions of the dispatch method, which has no tangible benefits and several potential pitfalls.

Consider the following Slice definitions:

```
["ami "] interface I {  
    bool isValid();  
    float computeRate();  
};  
  
interface J {  
    ["amd"] void startProcess();  
    ["ami ", "amd"] int endProcess();  
};
```

In this example, all proxy methods of interface I are generated with support for synchronous and asynchronous invocations. In interface J, the `startProcess` operation uses asynchronous dispatch, and the `endProcess` operation supports asynchronous invocation and dispatch.

Specifying metadata at the operation level, rather than at the interface or class level, not only minimizes the amount of generated code, but more importantly, it minimizes complexity. Although the asynchronous model is more flexible, it is also more complicated to use. It is therefore in your best interest to limit the use of the asynchronous model to those operations for which it provides a particular advantage, while using the simpler synchronous model for the rest.

17.2.4 Transparency

The use of an asynchronous model does not affect what is sent “on the wire.” Specifically, the invocation model used by a client is transparent to the server, and the dispatch model used by a server is transparent to the client. Therefore, a server has no way to distinguish a client’s synchronous invocation from an asynchronous invocation, and a client has no way to distinguish a server’s synchronous reply from an asynchronous reply.

17.3 Using AMI

In this section, we describe the Ice implementation of AMI and how to use it. We begin by discussing a way to (partially) simulate AMI using oneway invocations. This is not a technique that we recommend, but it is an informative exercise that highlights the benefits of AMI and illustrates how it works. Next, we explain the AMI mapping and illustrate its use with examples.

17.3.1 Simulating AMI using Oneways

As we discussed at the beginning of the chapter, synchronous invocations are not appropriate for certain types of applications. For example, an application with a graphical user interface typically must avoid blocking the window system's event dispatch thread because blocking makes the application unresponsive to user commands. In this situation, making a synchronous remote invocation is asking for trouble.

The application could avoid this situation using oneway invocations (see page 15), which by definition cannot return a value or have any out parameters. Since the Ice run-time does not expect a reply, the invocation blocks only as long as it takes to marshal and copy the message into the local transport buffer. However, the use of oneway invocations may require unacceptable changes to the interface definitions. For example, a twoway invocation that returns results or raises user exceptions must be converted into at least two operations: one for the client to invoke with oneway semantics that contains only in parameters, and one (or more) for the server to invoke to notify the client of the results.

To illustrate these changes, suppose that we have the following Slice definition:

```
interface I {
    int op(string s, out long l);
};
```

In its current form, the operation `op` is not suitable for a oneway invocation because it has an out parameter and a non-void return type. In order to accommodate a oneway invocation of `op`, we can change the Slice definitions as shown below:

```
interface ICallback {
    void opResults(int result, long l);
};

interface I {
    void op(ICallback* cb, string s);
};
```

We made several modifications to the original definition:

- We added interface `ICallback`, containing an operation `opResults` whose arguments represent the results of the original twoway operation. The server invokes this operation to notify the client of the completion of the operation.

- We modified `I : : op` to be compliant with oneway semantics: it now has a void return type, and takes only in parameters.
- We added a parameter to `I : : op` that allows the client to supply a proxy for its callback object.

As you can see, we have made significant changes to our interface definitions to accommodate the implementation requirements of the client. One ramification of these changes is that the client must now also be a server, because it must create an instance of `I Cal I back` and register it with an object adapter in order to receive notifications of completed operations.

A more severe ramification, however, is the impact these changes have on the type system, and therefore on the server. Whether a client invokes an operation synchronously or asynchronously should be irrelevant to the server; this is an artifact of behavior that should have no impact on the type system. By changing the type system as shown above, we have tightly coupled the server to the client, and eliminated the ability for `op` to be invoked synchronously.

To make matters even worse, consider what would happen if `op` could raise user exceptions. In this case, `I Cal I back` would have to be expanded with additional operations that allow the server to notify the client of the occurrence of each exception. Since exceptions cannot be used as parameter or member types in `Slice`, this quickly becomes a difficult endeavor, and the results are likely to be equally difficult to use.

At this point, you will hopefully agree that this technique is flawed in many ways, so why do we bother describing it in such detail? The reason is that the Ice implementation of AMI uses a strategy similar to the one described above, with several important differences:

1. No changes to the type system are required in order to use AMI. The on-the-wire representation of the data is identical, therefore synchronous and asynchronous clients and servers can coexist in the same system, using the same operations.
2. The AMI solution accommodates exceptions in a reasonable way.
3. Using AMI does not require the client to also be a server.

17.3.2 Language Mappings

An operation for which the AMI metadata has been specified supports both synchronous and asynchronous invocation models. In addition to the proxy method for synchronous invocation, the code generator creates a proxy method for

asynchronous invocation, plus a supporting callback class. The generated code uses a pattern similar to the Slice modifications we made to the example in Section 17.3.1: the out parameters and return value are removed, leaving only in parameters for the invocation; the application supplies a callback object that is invoked with the results of the operation. In this case, however, the callback object is a purely local entity that is invoked by the Ice run-time in the client. The name of the callback class is constructed so that it cannot conflict with a user-defined Slice identifier.

C++ Mapping

The C++ code generator emits the following code for each AMI operation:

1. An abstract callback class used by the Ice run-time to notify the application about the completion of an operation. The name of the class is formed using the pattern `AMI_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods are provided:

```
void ice_response(<params>);
```

Indicates that the operation completed successfully. The parameters represent the return value and out parameters of the operation. If the operation has a non-void return type, then the first parameter of the `ice_response` method supplies the return value of the operation. Any out parameters present in the operation follow the return value, in the order of declaration.

```
void ice_exception(const Ice::Exception &);
```

Indicates that a local or user exception was raised.

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
class AMI_I_foo : public ... {
public:
    virtual void ice_response(Ice::Int, Ice::Long) = 0;
    virtual void ice_exception(const Ice::Exception &) = 0;
};
```

```
typedef IceUtil::Handle<AMI_I_foo> AMI_I_fooPtr;
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(const AMI_I_fooPtr &, Ice::Short);
```

Java Mapping

The Java code generator emits the following code for each AMI operation:

1. An abstract callback class used by the Ice run-time to notify the application about the completion of an operation. The name of the class is formed using the pattern `AMI_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Three methods are provided:

```
public void ice_response(<params>);
```

Indicates that the operation completed successfully. The parameters represent the return value and out parameters of the operation. If the operation has a non-void return type, then the first parameter of the `ice_response` method supplies the return value of the operation. Any out parameters present in the operation follow the return value, in the order of declaration.

```
public void ice_exception(Ice.LocalException ex);
```

Indicates that a local exception was raised.

```
public void ice_exception(Ice.UserException ex);
```

Indicates that a user exception was raised.

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation foo is shown below:

```
public abstract class AMI_I_foo extends ... {
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.LocalException ex);
    public abstract void ice_exception(Ice.UserException ex);
}
```

The proxy method for asynchronous invocation of operation foo is generated as follows:

```
public void foo_async(AMI_I_foo __cb, short s);
```

17.3.3 Example

To demonstrate the use of AMI in Ice, let us define the Slice interface for a simple computational engine:

```
sequence<float> Row;
sequence<Row> Grid;

exception RangeError {};

interface Model {
    ["ami"] Grid interpolate(Grid data, float factor)
        throws RangeError;
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way. In the sections below, we present C++ and Java clients that invoke `interpolate` using AMI.

C++ Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```
class AMI_Model_interpolateI : public AMI_Model_interpolate {
public:
    virtual void ice_response(const Grid & result)
    {
        cout << "received the grid" << endl;
    }
};
```

```

        // ... postprocessing ...
    }

    virtual void ice_exception(const Ice::Exception & ex)
    {
        try {
            ex.ice_throw();
        } catch (const RangeError & e) {
            cerr << "interpolate failed: range error" << endl;
        } catch (const Ice::LocalException & e) {
            cerr << "interpolate failed: " << e << endl;
        }
    }
};

```

The implementation of `ice_response` reports a successful result, and `ice_exception` displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```

ModelPrx model = ...;
AMI_Model_interpolatePtr cb = new AMI_Model_interpolateI;
Grid grid;
initializeGrid(grid);
model->interpolate_async(cb, grid, 0.5);

```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run-time receives the response to this request, it invokes the callback object supplied by the client.

Java Client

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

```

class AMI_Model_interpolateI extends AMI_Model_interpolate {
    public void ice_response(float[] [] result)
    {
        System.out.println("received the grid");
        // ... postprocessing ...
    }

    public void ice_exception(Ice.UserException ex)
    {
        assert(ex instanceof RangeError);
        System.err.println("interpolate failed: range error");
    }
}

```

```

    }

    public void ice_exception(Ice.LocalException ex)
    {
        System.err.println("interpolate failed: " + ex);
    }
}

```

The implementation of `ice_response` reports a successful result, and the `ice_exception` methods display a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

```

ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);

```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run-time receives the response to this request, it invokes the callback object supplied by the client.

17.3.4 Concurrency Issues

Support for asynchronous invocations in Ice is enabled by the client thread pool (Chapter 15), whose threads are primarily responsible for processing reply messages. It is important to understand the concurrency issues associated with asynchronous invocations:

- A callback object must not be used for multiple simultaneous invocations. An application that needs to aggregate information from multiple replies can create a separate object to which the callback objects delegate.
- Calls to the callback object are made by threads from the Ice run time's client thread pool, therefore synchronization may be necessary if the application might interact with the callback object at the same time as the reply arrives.
- The number of threads in the client thread pool determines the maximum number of simultaneous callbacks possible for asynchronous invocations. The default size of the client thread pool is one, meaning invocations on callback objects are serialized. If the size of the thread pool is increased, the application may require synchronization if the same callback object is used for multiple invocations.

17.4 Using AMD

This section describes the language mappings for AMD and continues the example introduced in Section 17.3.

17.4.1 Language Mappings

As we discussed in Section 17.3.2, the language mappings for AMI continue to allow applications to use the synchronous invocation model if desired: specifying the AMI metadata for an operation leaves the proxy method for synchronous invocation intact, and causes an additional proxy method to be generated in support of asynchronous invocation.

The language mappings for an AMD operation, however, do not allow the implementation to use both dispatch models. Specifying the AMD metadata causes the method for synchronous dispatch to be replaced with a method for asynchronous dispatch.

The asynchronous dispatch method has a signature similar to that of AMI: the return type is `void`, and the arguments consist of a callback object and the operation's parameters. In AMI the callback object is supplied by the application, but in AMD the callback object is supplied by the Ice run-time and provides methods for returning the operation's results or reporting an exception. The implementation is not required to invoke the callback object before the dispatch method returns; the callback object can be invoked at any time by any thread, but may only be invoked once. The name of the callback class is constructed so that it cannot conflict with a user-defined Slice identifier.

The details for each language mapping are provided below.

C++ Mapping

The C++ code generator emits the following code for each AMD operation:

1. A callback class used by the implementation to notify the Ice run-time about the completion of an operation. The name of this class is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in a class named `AMD_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Several methods are provided:

```
void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-void return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's out parameters follow the return value, in the order of declaration.

```
void ice_exception(const Ice::Exception &);
```

This version of `ice_exception` allows the server to report a user or local exception.

```
void ice_exception(const std::exception &);
```

This version of `ice_exception` allows the server to report a standard exception.

```
void ice_exception();
```

This version of `ice_exception` allows the server to report an unknown exception.

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["amd"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

```
class AMD_I_foo : public ... {
public:
    void ice_response(Ice::Int, Ice::Long);
    void ice_exception(const Ice::Exception &);
    void ice_exception(const std::exception &);
    void ice_exception();
};
```

The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(const AMD_I_fooPtr &, Ice::Short);
```

Java Mapping

The Java code generator emits the following code for each AMD operation:

1. A callback interface used by the implementation to notify the Ice run-time about the completion of an operation. The name of this interface is formed using the pattern `AMD_class_op`. For example, an operation named `foo` defined in interface `I` results in an interface named `AMD_I_foo`. The interface is generated in the same scope as the interface or class containing the operation. Two methods are provided:

```
public void ice_response(<params>);
```

The `ice_response` method allows the server to report the successful completion of the operation. If the operation has a non-void return type, the first parameter to `ice_response` is the return value. Parameters corresponding to the operation's out parameters follow the return value, in the order of declaration.

```
public void ice_exception(java.lang.Exception ex);
```

The `ice_exception` method allows the server to report an exception. With respect to exceptions, there is less compile-time type safety in an AMD implementation because there is no `throws` clause on the dispatch method and any exception type could conceivably be passed to `ice_exception`. However, the Ice run-time validates the exception value using the same semantics as for synchronous dispatch (see Section 4.8.4).

2. The dispatch method, whose name has the suffix `_async`. This method has a `void` return type. The first parameter is a reference to an instance of the callback interface described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

```
interface I {
    ["amd"] int foo(short s, out long l);
};
```

The callback interface generated for operation `foo` is shown below:

```
public interface AMD_I_foo {
    void ice_response(int __ret, long l);
    void ice_exception(java.lang.Exception ex);
}
```


The dispatch method for asynchronous invocation of operation `foo` is generated as follows:

```
void foo_async(AMD_I_foo __cb, short s);
```

17.4.2 Exceptions

There are two processing contexts in which the logical implementation of an AMD operation may need to report an exception: the dispatch thread (i.e., the thread that receives the invocation), and the response thread (i.e., the thread that sends the response)². Although we recommend that the callback object be used to report all exceptions to the client, it is legal for the implementation to raise an exception instead, but only from the dispatch thread.

As you would expect, an exception raised from a response thread cannot be caught by the Ice run-time; the application's run-time environment determines how such an exception is handled. Therefore, a response thread must ensure that it traps all exceptions and sends the appropriate response using the callback object. Otherwise, if a response thread is terminated by an uncaught exception, the request may never be completed and the client might wait indefinitely for a response.

Whether raised in a dispatch thread or reported via the callback object, user exceptions are validated as described in Section 4.8.2, and local exceptions may undergo the translation described in Section 4.8.4.

17.4.3 Example

In this section, we continue the example we started in Section 17.3.3, but first we must modify the operation to add AMD metadata:

```
sequence<float> Row;  
sequence<Row> Grid;  
  
exception RangeError {};
```

2. These are not necessarily two different threads: the response can also be sent from the dispatch thread if desired.

```
interface Model {
    ["ami", "amd"] Grid interpolate(Grid data, float factor)
        throws RangeError;
};
```

The sections below provide implementations of the `Model` interface in C++ and Java.

C++ Servant

Our servant class derives from `Model` and supplies a definition for the `interpolate_async` method:

```
class ModelI : virtual public Model,
               virtual public IceUtil::Mutex {
public:
    virtual void interpolate_async(
        const AMD_Model_interpolatePtr &,
        const Grid &,
        Ice::Float,
        const Ice::Current &);

private:
    std::list<JobPtr> _jobs;
};
```

The implementation of `interpolate_async` uses synchronization to safely record the callback object and arguments in a `Job` that is added to a queue:

```
void ModelI::interpolate_async(
    const AMD_Model_interpolatePtr & cb,
    const Grid & data,
    Ice::Float factor,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock sync(*this);
    JobPtr job = new Job(cb, data, factor);
    _jobs.push_back(job);
}
```

After queuing the information, the operation returns control to the Ice run-time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute` to perform the interpolation. `Job` is defined as follows:

```

class Job : public IceUtil::Shared {
public:
    Job(
        const AMD_Model_interpolatePtr &,
        const Grid &,
        Ice::Float);
    void execute();

private:
    bool interpolateGrid();

    AMD_Model_interpolatePtr _cb;
    Grid _grid;
    Ice::Float _factor;
};
typedef IceUtil::Handle<Job> JobPtr;

```

The implementation of `execute` uses `interpolateGrid` (not shown) to perform the computational work:

```

Job::Job(
    const AMD_Model_interpolatePtr & cb,
    const Grid & grid,
    Ice::Float factor) :
    _cb(cb), _grid(grid), _factor(factor)
{
}

void Job::execute()
{
    if(!interpolateGrid()) {
        _cb->ice_exception(RangeError());
        return;
    }
    _cb->ice_response(_grid);
}

```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

Java Servant

Our servant class derives from `_ModelDisp` and supplies a definition for the `interpolate_async` method that creates a `Job` to hold the callback object and arguments, and adds the `Job` to a queue. The method is synchronized to guard access to the queue:

```
public final class ModelI extends _ModelDisp {
    synchronized public void interpolate_async(
        AMD_Model_interpolate cb,
        float[] [] data,
        float factor,
        Ice.Current current)
        throws RangeError
    {
        _jobs.add(new Job(cb, data, factor));
    }

    java.util.LinkedList _jobs = new java.util.LinkedList();
}
```

After queuing the information, the operation returns control to the Ice run-time, making the dispatch thread available to process another request. An application thread removes the next `Job` from the queue and invokes `execute`, which uses `interpolateGrid` (not shown) to perform the computational work:

```
class Job {
    Job(AMD_Model_interpolate cb,
        float[] [] grid,
        float factor)
    {
        _cb = cb;
        _grid = grid;
        _factor = factor;
    }

    void execute()
    {
        if(!interpolateGrid()) {
            _cb.ice_exception(new RangeError());
            return;
        }
        _cb.ice_response(_grid);
    }

    private boolean interpolateGrid() {
```

```
        // ...  
    }  
  
    private AMD_Model_interpolate _cb;  
    private float[] [] _grid;  
    private float _factor;  
}
```

If `interpolateGrid` returns `false`, then `ice_exception` is invoked to indicate that a range error has occurred. The `return` statement following the call to `ice_exception` is necessary because `ice_exception` does not throw an exception; it only marshals the exception argument and sends it to the client.

If interpolation was successful, `ice_response` is called to send the modified grid back to the client.

17.5 Summary

Synchronous remote invocations are a natural extension of local method calls that leverage the programmer's experience with object-oriented programming and ease the learning curve for programmers who are new to distributed application development. However, the blocking nature of synchronous invocations makes some application tasks more difficult, or even impossible, therefore Ice provides a straightforward interface to its asynchronous facilities.

Using asynchronous method invocation, a calling thread is able to invoke an operation and regain control immediately, without blocking while the operation is in progress. When the results are received, the Ice run-time notifies the application via a callback.

Similarly, asynchronous method dispatch allows a servant to send the results of an operation at any time, not necessarily within the operation implementation. A servant can improve scalability and conserve thread resources by queuing time-consuming requests for processing at a later time.

Chapter 18

The Ice Protocol

18.1 Chapter Overview

The Ice protocol definition consists of three major parts:

- a set of data encoding rules that determine how the various data types are serialized
- a number of message types that are interchanged between client and server, together with rules as to what message is to be sent under what circumstances
- a set of rules that determine how client and server agree on a particular protocol and encoding version

Section 18.2 describes the encoding rules, Section 18.3 describes the various protocol messages, Section 18.4 describes compression, and Section 18.5 explains how the protocol and encoding are versioned and how client and server agree on a common version. (Both encoding and protocol specifications are currently at version 1.0.) Finally, Section 18.6 provides a comparison of the Ice protocol and encoding with those used by CORBA.

18.2 Data Encoding

The key goals of the Ice data encoding are simplicity and efficiency. In keeping with these principles, the encoding does not align primitive types on word boundaries and therefore eliminates the wasted space and additional complexity that alignment requires. The Ice data encoding simply produces a stream of contiguous bytes; data contains no padding bytes and need not be aligned on word boundaries.

Data is always encoded using little-endian byte order for numeric types. (Most machines use a little-endian byte order, so the Ice data encoding is “right” more often than not.) Ice does not use a “receiver makes it right” scheme because of the additional complexity this would introduce. Consider, for example, a chain of receivers that merely forward data along the chain until that data arrives at an ultimate receiver. (Such topologies are common for event distribution services.) The Ice protocol permits all the intermediates to forward the data without requiring it to be unmarshaled: the intermediates can forward requests by simply copying blocks of binary data. With a “receiver makes it right” scheme, the intermediates would have to unmarshal and remarshal the data whenever the byte order of the next receiver in the chain differs from the byte order of the sender, which is inefficient.

Ice requires clients and servers that run on big-endian machines to incur the extra cost of byte swapping data into little-endian layout, but that cost is insignificant compared to the overall cost of sending or receiving a request.

18.2.1 Sizes

Many of the types involved in the data encoding, as well as several protocol message components, have an associated size or count. A size is a non-negative number. Sizes and counts are encoded in one of two ways:

1. If the number of elements is less than 255, the size is encoded as a single byte indicating the number of elements.
2. If the number of elements is greater than or equal to 255, the size is encoded as a byte with value 255, followed by an `int` indicating the number of elements.

Using this encoding to indicate sizes is significantly cheaper than always using an `int` to store the size, especially when marshaling sequences of short strings: counts of up to 254 require only a single byte instead of four. This comes at the expense of counts greater than 254, which require five bytes instead of four.

However, for sequences or strings of length greater than 254, the extra byte is insignificant.

18.2.2 Encapsulations

An encapsulation is used to contain variable-length data that an intermediate receiver may not be able to decode, but that the receiver can forward to another recipient for eventual decoding. An encapsulation is encoded as if it were the following structure:

```
struct Encapsulation {  
    int size;  
    byte major;  
    byte minor;  
    // [... size - 6 bytes ...]  
};
```

The `size` member specifies the size of the encapsulation in bytes (including the `size`, `major`, and `minor` fields). The `major` and `minor` fields specify the encoding version of the data contained in the encapsulation (see Section 18.5.2). The version information is followed by `size-6` bytes of encoded data.

All the data in an encapsulation is context-free, that is, nothing inside an encapsulation can refer to anything outside the encapsulation. This property allows encapsulations to be forwarded among address spaces as a blob of data.

Encapsulations can be nested, that is, contain other encapsulations.

An encapsulations can be empty, in which case its byte count is 6.

18.2.3 Slices

Exceptions and classes are subject to slicing if the receiver of a value only partially understands the received value (that is, only has knowledge of a base type, but not of the actual run-time derived type). To allow the receiver of an exception or class to ignore those parts of a value that it does not understand, exception and class values are marshaled as a sequence of slices (one slice for each level of the inheritance hierarchy). A slice is a byte count encoded as a fixed-length four-byte integer, followed by the data for the slice. (The byte count includes the four bytes occupied by the count itself, so an empty slice has a byte count of four and no data.) The receiver of a value can skip over a slice by reading the byte count b , and then discarding the next $b - 4$ bytes in the input stream.

A slice can be empty, in which case its byte count is 4.

18.2.4 Basic Types

The basic types are encoded as shown in Table 18.1. Integer types (`short`, `int`, `long`) are represented as two's complement numbers, and floating point types (`float`, `double`) use the IEEE standard formats [6]. All numeric types use a little-endian byte order.

Table 18.1. Encoding for basic types.

Type	Encoding
<code>bool</code>	A single byte with value 1 for <code>true</code> , 0 for <code>false</code>
<code>byte</code>	An uninterpreted byte
<code>short</code>	Two bytes (LSB, MSB)
<code>int</code>	Four bytes (LSB .. MSB)
<code>long</code>	Eight bytes (LSB .. MSB)
<code>float</code>	Four bytes (23-bit fractional mantissa, 8-bit exponent, sign bit)
<code>double</code>	Eight bytes (52-bit fractional mantissa, 11-bit exponent, sign bit)

18.2.5 Strings

Strings are encoded as a size (see Section 18.2.1), followed by the string contents in UTF8 format [21]. Strings are not NUL-terminated. An empty string is encoded with a size of zero.

18.2.6 Sequences

Sequences are encoded as a size (see Section 18.2.1) representing the number of elements in the sequence, followed by the elements encoded as specified for their type.

18.2.7 Dictionaries

Dictionaries are encoded as a size (see Section 18.2.1) representing the number of key–value pairs in the dictionary, followed by the pairs. Each key–value pair is

encoded as if it were a struct containing the key and value as members, in that order.

18.2.8 Enumerators

Enumerated values are encoded depending on the number of enumerators:

- If the enumeration has 1 - 127 enumerators, the value is marshaled as a byte.
- If the enumeration has 128 - 32767 members, the value is marshaled as a short.
- If the enumeration has more than 32767 members, the value is marshaled as an int.

The value is the ordinal value of the corresponding enumerator, with the first enumerator value encoded as zero.

18.2.9 Structures

The members of a structure are encoded in the order they appear in the struct declaration, as specified for their types.

18.2.10 Exceptions

Exceptions are marshaled as shown in Figure 18.1

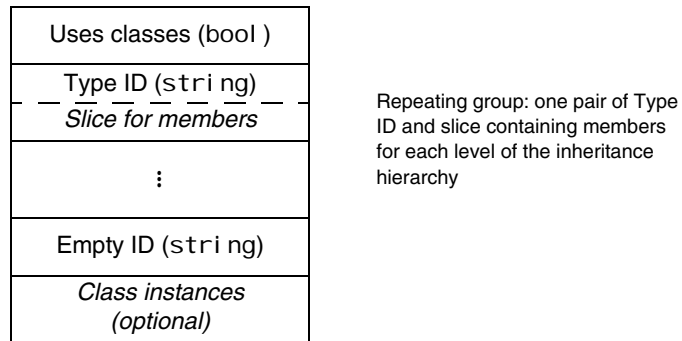


Figure 18.1. Marshaling format for exceptions.

Every exception instance is preceded by a single byte that indicates whether the exception uses class members: the byte value is 1 if any of the exception members

are classes (or if any of the exception members, recursively, contain class members) and 0, otherwise.

Following the header byte, the exception is marshaled as a sequence of pairs: the first member of each pair is the type ID for an exception slice, and the second member of the pair is a slice containing the marshaled members of that slice. The sequence of pairs is marshaled in derived-to-base order, with the most-derived slice first, and ending with the least-derived slice. The sequence of pairs is terminated by a single empty string. Within each slice, data members are marshaled as for structures: in the order in which they are defined in the Slice definition.

Following the sequence of pairs, any class instances that are used by the members of the exception are marshaled. This final part is optional: it is present only if the header byte is 1. (See Section 18.2.11 for a detailed explanation of how class instances are marshaled.)

To illustrate the marshaling, consider the following exception hierarchy:

```
exception Base {
    int baseInt;
    string baseString;
};

exception Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Assume that the exception members are initialized to the values shown in Table 18.2.

Table 18.2. Member values of an exception of type `Derived`.

Member	Type	Value	Marshaled Size (in bytes)
baseInt	int	99	4
baseString	string	"Hello"	6
derivedBool	bool	true	1
derivedString	string	"World!"	7
derivedDouble	double	3.14	8

From Table 18.2, we can see that the total size of the members of `Base` is 10 bytes, and the total size of the members of `Derived` is 16 bytes. None of the exception members are classes. An instance of this exception has the on-the-wire representation shown in Table 18.3. (The size, type, and byte offset of the marshaled representation is indicated for each component.)

Table 18.3. Marshaled representation of the exception in Table 18.2.

Marshaled Value	Size in Bytes	Type	Byte offset
0 (<i>no class members</i>)	1	bool	0
"::Derived" (<i>type ID</i>)	10	string	1
20 (<i>byte count for slice</i>)	4	int	11
1 (<i>derivedBool</i>)	1	bool	15
"World!" (<i>derivedString</i>)	7	string	16
3.14 (<i>derivedDouble</i>)	8	double	23
"::Base" (<i>type ID</i>)	7	string	31
14 (<i>byte count for slice</i>)	4	int	38
99 (<i>baseInt</i>)	4	int	42
"Hello" (<i>baseString</i>)	6	string	46
" " (<i>empty string</i>)	1	string	52

Note that the size of each string is one larger than the actual string length. This is because each string is preceded by a count of its number of bytes, as explained in Section 18.2.5.

The receiver of this sequence of values uses the header byte to decide whether it eventually must unmarshal any class instances contained in the exception (none in this example) and then examines the first type ID (`::Derived`). If the receiver recognizes that type ID, it can unmarshal the contents of the first slice, followed by the remaining slices; otherwise, the receiver reads the byte count that follows the unknown type (20) and then skips 20 + 4 bytes in the input stream, which is the start of the type ID for the second slice (`::Base`). If the receiver does not recognize that type ID either, it again reads the byte count following the type ID (14), skips 14 + 4 bytes, and reads the empty type ID. This empty ID indicates the end of

the exception. If a receiver receives an exception without recognizing any of its type IDs, it throws an `UnknownUserException` to the application.

18.2.11 Classes

The marshaling for classes is complex, due to the need to deal with the pointer semantics for graphs of classes, as well as the need for the receiver to slice classes of unknown derived type. In addition, the marshaling for classes uses a type ID compression scheme to avoid repeatedly marshaling the same type IDs for large graphs of class instances.

Basic Marshaling Format

Classes are marshaled similar to exceptions: each instance is divided into a number of pairs containing a type ID and a slice (one pair for each level of the inheritance hierarchy) and marshaled in derived-to-base order. Only data members are marshaled—no information is sent that would relate to operations. Unlike exceptions, no header byte precedes a class. Instead, each marshaled class instance is preceded by a (non-zero) positive integer that provides an identity for the instance. The sender assigns this identity during marshaling such that each marshaled instance has a different identity. The receiver uses that identity to correctly reconstruct graphs of classes. Unlike for exceptions, the sequence of type ID and slice pairs is *not* terminated by an empty ID. The overall marshaling format for classes is shown in Figure 18.2.

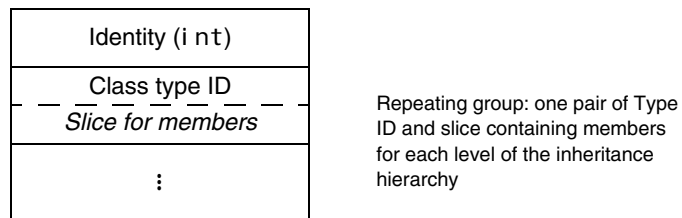


Figure 18.2. Marshaling format for classes.

Class Type IDs

Unlike for exception type IDs, class type IDs are not simple strings. Instead, a class type ID is marshaled as a boolean followed by either a string or a size, to conserve bandwidth. To illustrate this, consider the following class hierarchy:

```
class Base {  
    // ...  
};  
  
class Derived extends Base {  
    // ...  
};
```

The type IDs for the class slices are `::Derived` and `::Base`. Suppose the sender marshals three instances of `::Derived` as part of a single request. (For example, two instances could be out-parameters and one instance could be the return value.)

The first instance that is sent on the wire contains the type IDs `::Derived` and `::Base` preceding their respective slices. Because marshaling proceeds in derived-to-base order, the first type ID that is sent is `::Derived`. Every time the sender sends a type ID that it has not sent previously in the same request, it sends the boolean value `false`, followed by the type ID. Internally, the sender also assigns a unique positive number to each type ID. These numbers start at 1 and increment by one for each type ID that has not been marshaled previously. This means that the first type ID is encoded as the boolean value `false`, followed by `::Derived`, and the second type ID is encoded as the boolean value `false`, followed by `::Base`.

When the sender marshals the remaining two instances, it consults a lookup table of previously-marshaled type IDs. Because both type IDs were sent previously in the same request (or reply), the sender encodes all further occurrences of `::Derived` as the value `true` followed by the number 1 encoded as a size (see Section 18.2.1), and it encodes all further occurrences of `::Base` as the value `true` followed by the number 2 encoded as a size.

When the receiver reads a type ID, it first reads its boolean marker:

- If the boolean is `false`, the receiver reads a string and enters that string into a lookup table that maps integers to strings. The first new class type ID received in a request is numbered 1, the second new class type ID is numbered 2, and so on.
- If the boolean value is `true`, the receiver reads a number encoded as a size and uses that number to index into the lookup table to retrieve the corresponding class type ID.

Note that this numbering scheme is re-established for each new encapsulation. (As we will see in Section 18.3, parameters, return values, and exceptions are always marshaled inside an enclosing encapsulation.) For subsequent or nested encapsu-

lation, the numbering scheme restarts, with the first new type ID being assigned the value 1. In other words, each encapsulation uses its own independent numbering scheme for class type IDs to satisfy the constraint that encapsulations must not depend on their surrounding context.

Encoding class type IDs in this way provides significant savings in bandwidth: whenever an ID is marshaled a second and subsequent time, it is marshaled as a two-byte value (assuming no more than 254 distinct type IDs per request) instead of as a string. Because type IDs can be long, especially if you are using nested modules, the savings are considerable.

Simple Class Marshaling Example

To make the preceding discussion more concrete, consider the following class definitions:

```
interface SomeInterface {
    void op1();
};

class Base {
    int baseInt;
    void op2();
    string baseString;
};

class Derived extends Base implements SomeInterface {
    bool derivedBool;
    string derivedString;
    void op3();
    double derivedDouble;
};
```

Note that `Base` and `Derived` have operations, and that `Derived` also implements the interface `SomeInterface`. Because marshaling of classes is concerned with state, not behavior, the operations `op1`, `op2`, and `op3` are simply ignored during marshaling and the on-the-wire representation is as if the classes had been defined as follows:

```
class Base {
    int baseInt;
    string baseString;
};

class Derived extends Base {
```



```

    bool derivedBool;
    string derivedString;
    double derivedDouble;
};

```

Suppose the sender marshals two instances of `Derived` (for example, as two in-parameters in the same request). The member values are as shown in Table 18.4.

Table 18.4. Member values for two instances of class `Derived`.

	Member	Type	Value	Marshaled Size (in bytes)
First instance	baseInt	int	99	4
	baseString	string	"Hello"	6
	derivedBool	bool	true	1
	derivedString	string	"World!"	7
	derivedDouble	double	3.14	8
Second instance	baseInt	int	115	4
	baseString	string	"Cave"	5
	derivedBool	bool	false	1
	derivedString	string	"Canem"	6
	derivedDouble	double	6.32	8

The sender arbitrarily assigns a non-zero identity (see page 470) to each instance. Typically, the sender will simply consecutively number the instances starting at 1. For this example, assume that the two instances have the identities 1 and 2. The

marshaled representation for the two instances (assuming that they are marshaled immediately following each other) is shown in Table 18.5.

Table 18.5. Marshaled representation of the two instances in Table 18.4.

Marshaled Value	Size in Bytes	Type	Byte offset
1 (<i>identity</i>)	4	int	0
0 (<i>marker for class type ID</i>)	1	bool	4
"::Derived" (<i>class type ID</i>)	10	string	5
20 (<i>byte count for slice</i>)	4	int	15
1 (<i>derivedBool</i>)	1	bool	19
"World!" (<i>derivedString</i>)	7	string	20
3.14 (<i>derivedDouble</i>)	8	double	27
0 (<i>marker for class type ID</i>)	1	bool	35
"::Base" (<i>type ID</i>)	7	string	36
14 (<i>byte count for slice</i>)	4	int	43
99 (<i>baseInt</i>)	4	int	47
"Hello" (<i>baseString</i>)	6	string	51
0 (<i>marker for class type ID</i>)	1	bool	57
"::Ice::Object" (<i>class type ID</i>)	14	string	58
5 (<i>byte count for slice</i>)	4	int	72
0 (<i>number of dictionary entries</i>)	1	size	76
2 (<i>identity</i>)	4	int	77
1 (<i>marker for class type ID</i>)	1	bool	81
1 (<i>class type ID</i>)	1	size	82
19 (<i>byte count for slice</i>)	4	int	83
0 (<i>derivedBool</i>)	1	bool	87
"Canem" (<i>derivedString</i>)	6	string	88

Table 18.5. Marshaled representation of the two instances in Table 18.4.

Marshaled Value	Size in Bytes	Type	Byte offset
6.32 (<i>derivedDouble</i>)	8	double	94
1 (<i>marker for class type ID</i>)	1	bool	102
9 (<i>byte count for slice</i>)	4	int	103
2 (<i>class type ID</i>)	1	size	107
115 (<i>baseInt</i>)	4	int	108
"Cave" (<i>baseString</i>)	5	string	112
1 (<i>marker for class type ID</i>)	1	bool	117
3 (<i>class type ID</i>)	1	size	118
5 (<i>byte count for slice</i>)	4	int	119
0 (<i>number of dictionary entries</i>)	1	size	123

Note that, because classes (like exceptions) are sent as a sequence of slices, the receiver of a class can slice off any derived parts of a class it does not understand. Also note that (as shown in Table 18.5) each class instance contains three slices. The third slice is for the type `::Ice::Object`, which is the base type of all classes. The class type ID `::Ice::Object` has the number 3 in this example because it is the third distinct type ID that is marshaled by the sender. (See entries at byte offsets 58 and 118 in Table 18.5.) All class instances have this final slice of type `::Ice::Object`.

It is necessary to marshal a separate slice for `::Ice::Object` because `::Ice::Object` contains the facet map for each object (see XREF). As a built-in type, `::Ice::Object` has no Slice definition. However, it is marshaled as if it were defined as follows:

```
module Ice {
    class Object;

    dictionary<string, Object> FacetMap;
```

```

class Object {
    FacetMap facets;
};

```

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Marshaling Pointers

Classes support pointer semantics, that is, you can construct graphs of classes. It follows that classes can arbitrarily point at each other. The class identity (see page 470) is used to distinguish instances and pointers as follows:

- A class identity of 0 denotes a null pointer.
- A class identity > 0 precedes the marshaled contents of an instance (see page 470).
- A class identity < 0 denotes a pointer to an instance.

Identity values less than zero are pointers. For example, if the receiver receives the identity 57, this means that the corresponding class member that is currently being unmarshaled will eventually point at the instance with identity 57.

For structures, classes, exceptions, sequences, and dictionary members that do not contain class members, the Ice protocol uses a simple depth-first traversal algorithm to marshal the members. For example, structure members are marshaled in the order of their Slice definition; if a structure member itself is of complex type, such as a sequence, the sequence is marshaled in toto where it appears inside its enclosing structure. For complex types that contain class members, this depth-first marshaling is suspended: instead of marshaling the actual class instance at this point, a negative identity is marshaled that indicates which class instance that member must eventually denote. For example, consider the following definitions:

```

class C {
    // ...
};

struct S {
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
};

```

Suppose we initialize a structure of type *S* as follows:

```
S myS;
myS.i = 99;
myS.firstC = new C;           // New instance
myS.secondC = 0;             // null
myS.thirdC = myS.firstC;     // Same instance as previously
myS.j = 100;
```

When this structure is marshaled, the contents of the three class members are not marshaled in-line. Instead, the sender marshals the negative identities of the corresponding instances. Assuming that the sender has assigned the identity 78 to the instance assigned to *myS.firstC*, *myS* is marshaled as shown in Table 18.6.

Table 18.6. Marshaled representation of *myS*.

Marshaled Value	Size in Bytes	Type	Byte offset
99 (<i>myS.i</i>)	4	int	0
-78 (<i>myS.firstC</i>)	4	int	4
0 (<i>myS.secondC</i>)	4	int	8
-78 (<i>mys.thirdC</i>)	4	int	12
100 (<i>myS.j</i>)	4	int	16

Note that *myS.firstC* and *myS.thirdC* both use the identity -78. This allows the receiver to recognize that *firstC* and *thirdC* point at the same class instance (rather than at two different instances that happen to have the same contents).

Marshaling the negative identities instead of the contents of an instance allows the receiver to accurately reconstruct the class graph that was sent by the sender. However, this begs the question of *when* the actual instances are to be marshaled as described at the beginning of this section. As we will see in Section 18.3, parameters and return values are marshaled as if they were members of a structure. For example, if an operation invocation has five input parameters, the client marshals the five parameters end-to-end as if they were members of a single structure. If any of the five parameters are class instances, or are of complex type (recursively) containing class instances, the sender marshals the parameters in multiple passes: the first pass marshals the parameters end-to-end, using the usual depth-first algorithm:

- If the sender encounters a class member during marshaling, it checks whether it has marshaled the same instance previously for the current request or reply:
 - If the instance has not been marshaled before, the sender assigns a new identity to the instance and marshals the negative identity.
 - Otherwise, if the instance was marshaled previously, the sender sends the same negative identity that is previously sent for that instance.

In effect, during marshaling, the sender builds an identity table that is indexed by the address of each instance; the lookup value for the instance is its identity.

Once the first pass ends, the sender has marshaled all the parameters, but has not yet marshaled any of the class instances that may be pointed at by various parameters or members. The identity table at this point contains all those instances for which negative identities (pointers) were marshaled, so whatever is in the identity table at this point are the classes that the receiver still needs. The sender now marshals those instances in the identity table, but with positive identities and followed by their contents, as described on page 472. The outstanding instances are marshaled as a sequence, that is, the sender marshals the number of instances as a size (see Section 18.2.1), followed by the actual instances.

In turn, the instances just sent may themselves contain class members; when those class members are marshaled, the sender assigns an identity to new instances or uses a negative identity for previously marshaled instances as usual. This means that, by the end of the second pass, the identity table may have grown, necessitating a third pass. That third pass again marshals the outstanding class instances as a size followed by the actual instances. The third pass contains all those instances that were not marshaled in the second pass. Of course, the third pass may trigger yet more passes until, finally, the sender has sent all outstanding instances, that is, marshaling is complete. At this point, the sender terminates the sequence of passes by marshaling an empty sequence (the value 0 encoded as a size).

To illustrate this with an example, consider the definitions shown in Section 4.9.7 on page 99 once more:

```
enum UnaryOp { UnaryPlus, UnaryMinus, Not };
enum BinaryOp { Plus, Minus, Multiply, Divide, And, Or };

class Node {
    idempotent long eval();
};
```

```

class UnaryOperator extends Node {
    UnaryOp operator;
    Node operand;
};

class BinaryOperator extends Node {
    BinaryOp op;
    Node operand1;
    Node operand2;
};

class Operand {
    long val;
};

```

These definitions allow us to construct expression trees. Suppose the client initializes a tree to the shape shown in Figure 18.3, representing the expression $(a + b) * c + d$. The values outside the nodes are the identities assigned by the client.

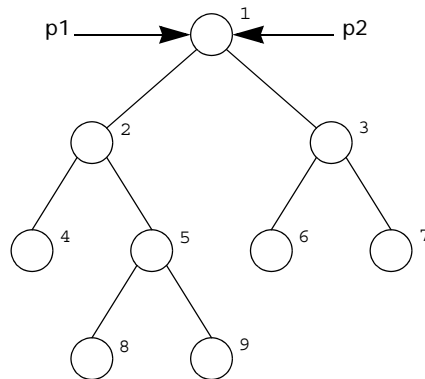


Figure 18.3. Expression tree for the expression $(a + b) * c + d$.

Both p1 and p2 denote the root node.

The client passes the root of the tree to the following operation in the parameters p1 and p2, as shown on page 479. (Even though it does not make sense to pass the same parameter value twice, we do it here for illustration purposes):

```

interface Tree {
    void sendTree(Node p1, Node p2);
};

```

The client now marshals the two parameters p1 and p2 to the server, resulting in the value -1 being sent twice in succession. (The client arbitrarily assigns an identity to each node. The value of the identity does not matter, as long as each node has a unique identity. For simplicity, the Ice implementation numbers instances with a counter that starts counting at 1 and increments by one for each unique instance.) This completes the marshaling of the parameters and results in a single instance with identity 1 in the identity table. The client now marshals a sequence containing a single element, node 1, as described on page 472. In turn, node 1 results in nodes 2 and 3 being added to the identity table, so the next sequence of nodes contains two elements, nodes 2 and 3. The next sequence of nodes contains nodes 4, 5, 6, and 7, followed by another sequence containing nodes 8 and 9. At this point, no more class instances are outstanding, and the client marshals an empty sequence to indicate to the receiver that the final sequence has been marshaled.

Within each sequence, the order in which class instances are marshaled is irrelevant. For example, the third sequence could equally contain nodes 7, 6, 4, and 5, in that order. What is important here is that each sequence contains nodes that are an equal number of “hops” away from the initial node: the first sequence contains the initial node(s), the second sequence contains all nodes that can be reached by traversing a single link from the initial node(s), the third sequence contains all nodes that can be reached by traversing two links from the initial node(s), and so on.

Now consider the same example once more, but with different parameter values for sendTree: p1 denotes the root of the tree, and p2 denotes the operator of the right-hand sub-tree, as shown in Figure 18.4.

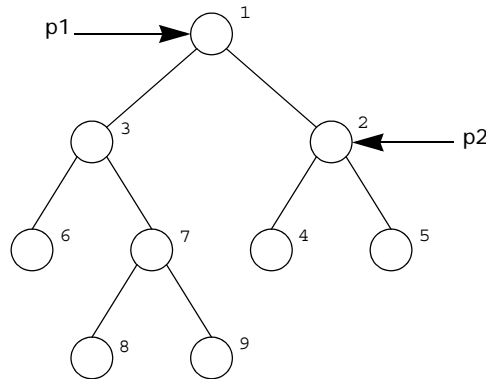


Figure 18.4. The expression tree of Figure 18.3, with p1 and p2 denoting different nodes.

The graph that is marshaled is exactly the same, but instances are marshaled in a different order and with different identities:

- During the first pass, the client sends the identities -1 and -2 for the parameter values.
- The second pass marshals a sequence containing nodes 1 and 2.
- The third pass marshals a sequence containing nodes 3, 4, and 5.
- The fourth pass marshals a sequence containing nodes 6 and 7.
- The fifth pass marshals a sequence containing nodes 8 and 9.
- The final pass marshals an empty sequence.

In this way, any graph of nodes can be transmitted (including graphs that contain cycles). The receiver reconstructs the graph by filling in a patch table during unmarshaling:

- Whenever the receiver unmarshals a negative identity, it adds that identity to a patch table; the lookup value is the memory address of the parameter or member that eventually will point at the corresponding instance.
- Whenever the receiver unmarshals an actual instance, it adds the instance to an unmarshaled table; the lookup value is the memory address of the instantiated class. The receiver then uses the address of the instance to patch any parameters or members with the actual memory address.

Note that the receiver may receive negative identities that denote class instances that have been unmarshaled already (that is, point “backward” in the unmarshaling stream), as well as instances that are yet to be unmarshaled (that is, point “forward” in the unmarshaling stream). Both scenarios are possible, depending on the order in which instances are marshaled, as well as their in-degree.

To provide another example, consider the following definition:

```
class C {
    // ...
};

sequence<C> CSeq;
```

Suppose the client marshals a sequence of 100 C instances to the server, with each instance being distinct. (That is, the sequence contains 100 pointers to 100 different instances, not 100 pointers to the same single instance.) In that case, the sequence is marshaled as a size of 100, followed by 100 negative identities, -1 to -100. Following that, the client marshals a single sequence containing the 100 instances, each instance with its positive identity in the range 1 to 100, and completes by marshaling an empty sequence.

On the other hand, if the client sends a sequence of 100 elements that all point to the same single class instance, the client marshals the sequence as a size of 100, followed by 100 negative identities, all with the value -1. The client then marshals a sequence containing a single element, namely instance 1, and completes by marshaling an empty sequence.

Class Graphs and Slicing

It is important to note that when a graph of class instances is sent, it always forms a connected graph. However, when the receiver rebuilds the graph, it may end up with a disconnected graph, due to slicing. Consider:

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
    Base b;
};
```

```
interface Example {
    void op(Base p);
};
```

Suppose the client has complete type knowledge, that is, understands both types `Base` and `Derived`, but the server only understands type `Base`, so the derived part of a `Derived` instance is sliced. The client can instantiate classes to be sent as parameter `p` as follows:

```
DerivedPtr p = new Derived;
p->b = new Derived;
ExamplePrx e = ...;
e->op(p);
```

As far as the client is concerned, the graph looks like the one shown in Figure 18.5.

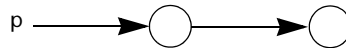


Figure 18.5. Sender-side view of a graph containing derived instances.

However, the server does not understand the derived part of the instances and slices them. Yet, the server unmarshals all the class instances, leading to the situation where the class graph has become disconnected, as shown in Figure 18.6.



Figure 18.6. Receiver-side view of the graph in Figure 18.5.

Of course, more complex situations are possible, such that the receiver ends up with multiple disconnected graphs, each containing many instances.

Exceptions with Class Members

If an exception contains class members, its header byte (see page 467) is 1 and the exception members are followed by the outstanding class instances as described on the preceding pages, that is, the actual exception members are followed by one or more sequences that contain the outstanding class instances, followed by an empty sequence that serves as an end marker.

18.2.12 Proxies

The first component of an encoded proxy is a value of type `Ice::Identity`. If the proxy is a nil value, the category and name members are empty strings, and no additional data is encoded. The encoding for a non-nil proxy consists of general parameters followed by endpoint parameters.

General Proxy Parameters

The general proxy parameters are encoded as if they were members of the following structure:

```
struct ProxyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
};
```

The general proxy parameters are described in Table 18.7.

Table 18.7. General proxy parameters.

Parameter	Description
id	The object identity
facet	The facet (or facet path)
mode	The proxy mode (0=twoway, 1=oneway, 2=batch oneway, 3=datagram, 4=batch datagram)
secure	true if secure endpoints are required, otherwise false

Endpoint Parameters

A proxy optionally contains an endpoint list (see XREF) or an adapter identifier, but not both.

- If a proxy contains endpoints, they are encoded immediately following the general parameters. A size specifying the number of endpoints is encoded first (see Section 18.2.1), followed by the endpoints. Each endpoint is encoded as a short specifying the endpoint type (1=TCP, 2=SSL, 3=UDP), followed by an encapsulation (see Section 18.2.2) of type-specific parameters. The type-

specific parameters for TCP, UDP, and SSL are presented in the sections that follow.

- If a proxy does not have endpoints, a single byte with value 0 immediately follows the general parameters and a string representing the object adapter identifier is encoded immediately following the zero byte.

Type-specific endpoint parameters are encapsulated because a receiver may not be capable of decoding them. For example, a receiver can only decode SSL endpoint parameters if it is configured with the SSL plug-in (see Chapter 23). However, the receiver must be able to re-encode the proxy with all of its original endpoints, in the order they were received, even if the receiver does not understand the type-specific parameters for an endpoint. Encapsulation of the parameters allows the receiver to do this.

TCP Endpoint Parameters

A TCP endpoint is encoded as an encapsulation containing the following structure:

```
struct TCPEndpointData {  
    string host;  
    int port;  
    int timeout;  
    bool compress;  
};
```

The endpoint parameters are described in Table 18.8.

Table 18.8. TCP endpoint parameters.

Parameter	Description
host	The server host (a host name or IP address)
port	The server port (1-65535)
timeout	The timeout in milliseconds for socket operations
compress	true if compression should be used (if possible), otherwise false

See Section 18.4 for more information on compression.

UDP Endpoint Parameters

A UDP endpoint is encoded as an encapsulation containing the following structure:

```
struct UDPEndpointData {
    string host;
    int port;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    bool compress;
};
```

The endpoint parameters are described in Table 18.9.

Table 18.9. UDP endpoint parameters.

Parameter	Description
host	The server host (a host name or IP address)
port	The server port (1-65535)
protocolMajor	The major protocol version supported by the endpoint
protocolMinor	The highest minor protocol version supported by the endpoint
encodingMajor	The major encoding version supported by the endpoint
encodingMinor	The highest minor encoding version supported by the endpoint
compress	true if compression should be used (if possible), otherwise false

See Section 18.4 for more information on compression.

SSL Endpoint Parameters

An SSL endpoint is encoded as an encapsulation containing the following structure:

```
struct SSLEndpointData {  
    string host;  
    int port;  
    int timeout;  
    bool compress;  
};
```

The endpoint parameters are described in Table 18.10.

Table 18.10. SSL endpoint parameters.

Parameter	Description
host	The server host (a host name or IP address)
port	The server port (1-65535)
timeout	The timeout in milliseconds for socket operations
compress	true if compression should be used (if possible), otherwise false

See Section 18.4 for more information on compression.

18.3 Protocol Messages

The Ice protocol uses five protocol messages:

- Request (from client to server)
- Batch request (from client to server)
- Reply (from server to client)
- Validate connection (from server to client)
- Close connection (client to server or server to client)

Of these messages, validate and close connection only apply to connection-oriented transports.

As with the data encoding described in Section 18.2, protocol messages have no alignment restrictions. Each message consists of a message header and (except for validate and close connection) a message body that immediately follows the header.

18.3.1 Message Header

Each protocol message has a 14-byte header that is encoded as if it were the following structure:

```
struct HeaderData {
    int  magic;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    byte messageType;
    byte compressionStatus;
    int  messageSize;
};
```

The members are described in Table 18.11.

Table 18.11. Message header members.

Member	Description
magic	A four-byte magic number consisting of the ASCII-encoded values of 'I', 'c', 'e', 'P' (0x49, 0x63, 0x65, 0x50)
protocolMajor	The protocol major version number
protocolMinor	The protocol minor version number
encodingMajor	The encoding major version number
encodingMinor	The encoding minor version number
messageType	The message type
compressionStatus	The compression status of the message (see Section 18.4)
messageSize	The size of the message in bytes, including the header

Currently, both the protocol and the encoding are at version 1.0. The valid message types are shown in Table 18.12.

Table 18.12. Message types.

Message Type	Encoding
Request	0
Batch request	1
Reply	2
Validate connection	3
Close connection	4

The encoding for these message bodies of each of these message types is described in the sections that follow.

18.3.2 Request Message Body

A request message contains the data necessary to perform an invocation on an object, including the identity of the object, the operation name, and input parameters. A request message is encoded as if it were the following structure:

```
struct RequestData {  
    int requestId;  
    Ice::Identity id;  
    Ice::StringSeq facet;  
    string operation;  
    byte mode;  
    Ice::Context context;  
    Encapsulation params;  
};
```

The members are described in Table 18.13.

Table 18.13. Request members.

Member	Description
requestId	The request identifier
id	The object identity
facet	The facet (or facet path)
operation	The operation name
mode	A byte representation of Ice: : OperationMode (0=normal, 1=nonmutating, 2=idempotent)
context	The invocation context
params	The encapsulated input parameters, in order of declaration

The request identifier zero (0) is reserved for use in oneway requests and indicates that the server must not send a reply to the client. A non-zero request identifier must uniquely identify the request on a connection, and must not be reused while a reply for the identifier is outstanding.

18.3.3 Batch Request Message Body

A batch request message contains one or more oneway requests, bundled together for the sake of efficiency. A batch request message is encoded as if it were a sequence of the following structure:

```
struct BatchRequestData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

The members are described in Table 18.14.

Table 18.14. Batch request members.

Member	Description
i d	The object identity
facet	The facet (or facet path)
operati on	The operation name
mode	A byte representation of Ice : Operati onMode
context	The invocation context
params	The encapsulated input parameters, in order of declaration

Note that no request ID is necessary for batch requests because only oneway invocations can be batched.

18.3.4 Reply Message Body

A reply message body contains the results of a twoway invocation, including any return value, out-parameters, or exception. A reply message body is encoded as if it were the following structure:

```
struct ReplyData {  
    int requestId;  
    byte replyStatus;  
    // [... messageSize - 19 bytes ...]  
};
```

The first four bytes of a reply message body contain a request ID. The request ID matches an outgoing request and allows the requester to associate the reply with the original request (see Section 18.3.2).

The byte following the request ID indicates the status of the request; the remainder of the reply message body following the status byte depends on the status value. The possible status values are shown in Table 18.15.

Table 18.15. Reply status.

Reply status	Encoding
Success	0
User exception	1
Object does not exist	2
Facet does not exist	3
Operation does not exist	4
Unknown Ice local exception	5
Unknown Ice user exception	6
Unknown exception	7

Reply Status 0: Success

A successful reply message is encoded as an encapsulation containing out-parameters (in the order of declaration), followed by the return value for the invocation, encoded according to their types as specified in Section 18.2. If an operation declares a `void` return type and no out-parameters, an empty encapsulation is encoded.

Reply Status 1: User exception

A user exception reply message contains an encapsulation containing the user exception, encoded as described in Section 18.2.10.

Reply Status 2: Object does not exist

If the target object does not exist, the reply message is encoded as if it were the following structure:

```
struct ReplyData {  
    Ice::Identity id;  
    Ice::StringSeq facet;  
    string operation;  
};
```

The members are described in Table 18.16.

Table 18.16. Invalid object reply members.

Member	Description
id	The object identity
facet	The facet (or facet path)
operation	The operation name

Reply Status 3: Facet does not exist

If the target object does not support the facet encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 4: Operation does not exist

If the target object does not support the operation encoded in the request message, the reply message is encoded as for reply status 2.

Reply Status 5: Unknown Ice local exception

The reply message for an unknown Ice local exception is encoded as a single string that describes the exception.

Reply Status 6: Unknown Ice user exception

The reply message for an unknown Ice user exception is encoded as a single string that describes the exception.

Reply Status 7: Unknown exception

The reply message for an unknown exception is encoded as a single string that describes the exception.

18.3.5 Validate Connection Message

A server sends a validate connection message when it receives a new connection.¹ The message indicates that the server is ready to receive requests; the client must not send any messages on the connection until it has received the validate connection message from the server. No reply to the message is expected by the server.

The purpose of the validate connection message is two-fold:

- It informs the client of the protocol and encoding versions that are supported by the server (see Section 18.5.3).
- It prevents the client from writing a request message to its local transport buffers until after the server has acknowledged that it can actually process the request. This avoids a race condition caused by the server's TCP/IP stack accepting connections in its backlog while the server is in the process of shutting down: if the client were to send a request in this situation, the request would be lost but the client could not safely re-issue the request because that might violate at-most-once semantics.

The validate connection message guarantees that a server is not in the middle of shutting down when the server's TCP/IP stack accepts an incoming connection and so avoids the race condition.

The message header described in Section 18.3.1 on page 488 comprises the entire validate connection message. The compression status of a validate connection message is always 0.

18.3.6 Close Connection Message

A close connection message is sent when a peer is about to gracefully shutdown a connection.² The message header described in Section 18.3.1 comprises the entire close connection message. The compression status of a close connection message is always 0.

Either client or server can initiate connection closure. On the client side, connection closure is triggered by *Active Connection Management (ACM)* (see XREF), which automatically reclaims connections that have been idle for some time.

1. Validate connection messages are only used for connection-oriented transports.

2. Close connection messages are only used for connection-oriented transports.

This means that connection closure can be initiated at will by either end of a connection; most importantly, no state is associated with a connection as far as the object model or application semantics are concerned.

The client side can close a connection whenever no reply for a request is outstanding on the connection. The sequence of events is:

1. The client sends a close connection message.
2. The client closes the writing end of the connection.
3. The server responds to the client's close connection message by closing the connection.

The server side can close a connection whenever no operation invocation is in progress that was invoked via that connection. This guarantees that the server will not violate at-most-once semantics: an operation, once invoked in a servant, is allowed to complete and its results are returned to the client. Note that the server can close a connection even after it has received a request from the client, provided that the request has not yet been passed to a servant. In other words, if the server decides that it wants to close a connection, the sequence of events is:

1. The server discards all incoming requests on the connection.
2. The server waits until all still executing requests have completed and their results have been returned to the client.
3. The server sends a close connection message to the client.
4. The server closes its writing end of the connection.
5. The client responds to the server's close connection message by closing both its reading and writing ends of the connection.
6. If the client has outstanding requests at the time it receives the close connection message, it re-issues these requests on a new connection. Doing so is guaranteed not to violate at-most-once semantics because the server guarantees not to close a connection while requests are still in progress on the server side.

18.3.7 Protocol State Machine

From a client's perspective, the Ice protocol behaves according to the state machine shown in Figure 18.7.

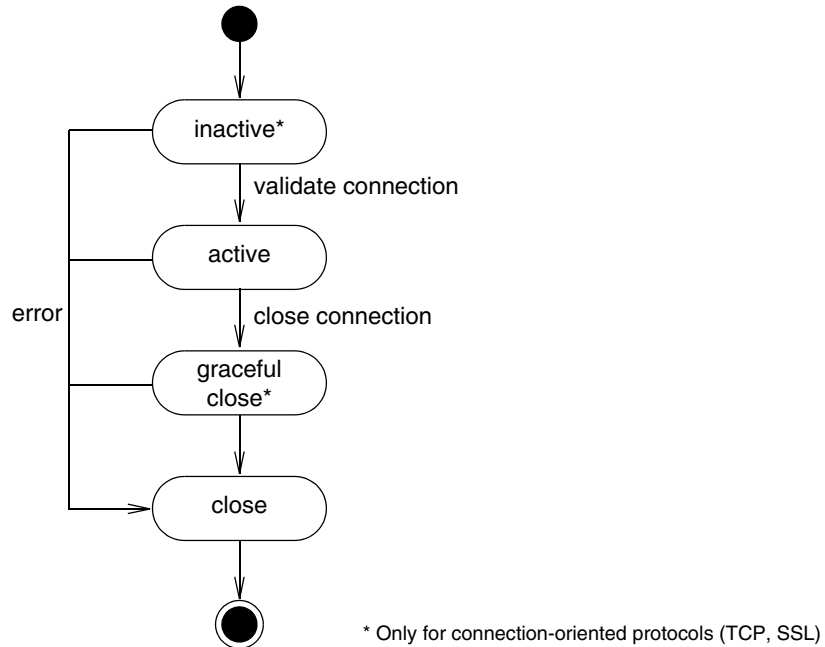


Figure 18.7. Protocol state machine.

To summarize, a new connection is inactive until a validate connection message (see Section 18.3.5) has been received by the client, at which point the active state is entered. The connection remains in the active state until it is shut down, which can occur when there are no more proxies using the connection, or after the connection has been idle for a while. At this point, the connection is gracefully closed, meaning that a close connection message is sent (see Section 18.3.6), and the connection is closed.

18.3.8 Disorderly Connection Closure

Any violation of the protocol or encoding rules results in a disorderly connection closure: the side of the connection that detects a violation unceremoniously closes it (without sending a close connection message or similar). There are many poten-

tial error conditions that can lead to disorderly connection closure; for example, the receiver might detect that a message has a bad magic number or incompatible version, receive a reply with an ID that does not match that of an outstanding request, receive a validate connection message when it should not, or find illegal data in a request (such as a negative size, or a size that disagrees with the actual data that was unmarshaled).

18.4 Compression

Compression is an optional feature of the Ice protocol; whether it is used for a particular message is determined by several factors:

1. Compression may not be supported on all platforms or in all language mappings. For example, compression is currently not supported by Ice for Java.
2. Compression can be used in a request or batch request only if the proxy specifies it (see Section 18.2.12).
3. A reply message is compressed only if its request was compressed.
4. For efficiency reasons, the Ice protocol engine does not compress messages smaller than 100 bytes.

If compression is used, the entire message excluding the header is compressed using the bzip2 algorithm [16]. The `messageSize` member of the message header therefore reflects the size of the compressed message, including the uncompressed header.

The `compressionStatus` field of the message header (see Section 18.3.1) indicates whether a message is compressed and whether the sender can accept a compressed reply, as shown in Table 18.17.

Table 18.17. Compression status values.

Reply status	Encoding	Applies to
Message is uncompressed, sender cannot accept a compressed reply.	0	Request, Batch Request, Reply, Validate Connection, Close Connection
Message is uncompressed, sender can accept a compressed reply.	1	Request, Batch Request
Message is compressed and sender can accept a compressed reply	2	Request, Batch Request, Reply

- A compression status of 0 indicates that the message is not compressed and, moreover, that the sender of this message cannot accept a compressed reply. A client that does not support compression always uses this value. A client that supports compression sets the value to 0 if the endpoint via which the request is dispatched indicates that it does not support compression.

A server uses this value for uncompressed replies.

- A compression status of 1 indicates that the message is not compressed, but that the server is free to return a compressed reply. A client uses this value if the endpoint via which the request is dispatched indicates that it supports compression, but the client has decided not to use compression for this particular request (presumably because the request is too small, so compression does not provide any saving).

This value applies only to request and batch request messages.

- A compression status of 2 indicates that the message is compressed and that the server is free to reply with a compressed message (but need not reply with a compressed message). A client that supports compression (obviously) sets this value only if the endpoint via which the request is dispatched indicates that it supports compression.

A server uses this value for compressed replies.

The message body of compressed request, batch request, or reply message is consists of the compressed version of the encodings specified in Sections 18.3.2 through 18.3.4.

Note that compression is likely to improve performance only over lower-speed links, for which bandwidth is the overall limiting factor. Over high-speed LAN links, the CPU time spent on compressing and uncompressing messages is longer than the time it takes to just send the uncompressed data.

18.5 Protocol and Encoding Versions

As we saw in the preceding sections, both the Ice protocol and encoding have separate major and minor version numbers. Separate versioning of protocol and encoding has the advantage that neither depends on the other: any version of the Ice protocol can be used with any version of the encoding, so they can evolve independently. (For example, Ice protocol version 1.1 could use encoding version 2.3, and vice versa.)

The Ice versioning mechanism provides the maximum possible amount of interoperability between clients and servers that use different versions of the Ice run time. In particular, older deployed clients can communicate with newer deployed servers and vice versa, provided that the message contents use types that are understandable to both sides.

For an example, assume that a later version of Ice were to introduce a new Slice keyword and data type, such as `complex`, for complex numbers. This would require a new minor version number for the encoding; let us assume that version 1.1 of the encoding is identical to the 1.0 encoding but, in addition,

supports the `complex` type. We now have four possible combinations of client and server encoding versions:

Table 18.18. Interoperability for different versions.

Client Version	Server Version	Operation with <code>complex</code> Parameter	Operation without <code>complex</code> Parameter
1.0	1.0	N/A	✓
1.1	1.0	N/A	✓
1.0	1.1	N/A	✓
1.1	1.1	✓	✓

As you can see, interoperability is provided to the maximum extent possible. If both client and server are at version 1.1, they can obviously exchange messages and will use encoding version 1.1. For version 1.0 clients and servers, obviously only operations that do not involve `complex` parameters can be invoked (because at least one of client and server do not know about the new `complex` type) and messages are exchanged using encoding version 1.0.

18.5.1 Version Ground Rules

For versioning of the protocol and encoding to be possible, *all* versions (present and future) of the Ice run time adhere to a few ground rules:

1. Encapsulations always have a six-byte header; the first four bytes are the size of the encapsulation (including the size of the header), followed by two bytes that indicate the major and minor version. How to interpret the remainder of the encapsulation depends on the major and minor version.
2. The first eight bytes of a message header always contain the magic number ‘I’, ‘c’, ‘e’, ‘P’, followed by four bytes of version information (two bytes for the protocol major and minor number, and two bytes of the encoding major and minor number). How to interpret the remainder of the header and the message body depends on the major and minor version.

These ground rules ensure that all current and future versions of the Ice run time can at least identify the version and size of an encapsulation and a message. This is particularly important for message switches such as IceStorm (see Chapter 26);

by keeping the version and size information in a fixed format, it is possible to forward messages that are, for example, at version 2.0, even though the message switch itself may still be at version 1.0.

18.5.2 Version Compatibility Rules

To establish whether a particular protocol version is compatible with another protocol version (or a particular encoding version is compatible with another encoding version), the following rules apply:

1. Different major versions are incompatible. There is no obligation on either clients or servers to support more than a single major version. For example, a server with major version 2 is under no obligation to also support major version 1.

This rule exists to permit the Ice run time to eventually get rid of old versions—without such a rule, all future releases of Ice would have to support all previous major versions forever. In plain language, the rule means that clients and servers that use different major versions simply cannot communicate with each other.

2. A receiver that advertises minor version n guarantees to be able to successfully decode all minor versions less than n . Note that this does *not* imply that messages using version $n - 1$ can be decoded as if they were version n : as far as their physical representation is concerned, two adjacent minor versions can be completely incompatible. However, because any receiver advertising version n is also obliged to correctly deal with version $n - 1$, minor version upgrades are *semantically* backward compatible, even though their physical representation may be incompatible.
3. A sender that supports minor version n guarantees to be able to send messages using all minor versions less than n . Moreover, the sender guarantees that if it receives a request using minor version k (with $k < n$), it will send the reply for that request using minor version k .

18.5.3 Version Negotiation

Client and server must somehow agree on which version to use to exchange messages. Depending on whether the underlying transport is connection-oriented or connection-less, different mechanisms are used to negotiate a common version.

Negotiation for Connection-Oriented Transports

For connection-oriented transports, the client opens a connection to the server and then waits for a validate connection message (see page 494). The validate connection message sent by the server indicates the server's major and highest supported minor version numbers for both protocol and encoding. If the server's and client's major version numbers do not match, the client side raises an `UnsupportedProtocolException` or `UnsupportedEncodingException`.

Assuming that the client has received a validate connection message from the server that matches the client's major version, the client knows the highest minor version number that is supported by the server. Thereafter, the client is obliged to send no message with a minor version number higher than the server's limit. However, the client is free to send a message with a minor version number that is less than the server's limit.

The server does not have a-priori knowledge of the highest minor version that is supported by the client (because there is no validate connection message from client to server). Instead, the server learns about the client version number in each individual message, by looking at the message header. That minor version indicates the minor version number that the client can accept. The scope of that minor version number is a single request-reply interaction. For example, if the client sends a request with minor version 3, the server must reply to that request with minor version 3 as well. However, the next client request might be with minor version 2, and the server must reply to that request with minor version 2.

For orderly connection closure via a close connection message, the server can use any minor version, but that minor version must not be higher than the highest minor version number that was received from the client while the connection was open.

Negotiation for Connection-Less Transports

For connection-less transports, no validate connection message exists, so the client must learn about the highest supported minor version number of the server via other means. The mechanism for this depends on whether a proxy for a connection-less endpoint is bound directly or indirectly (see Section 20.3.2):

- For direct proxies, the version information is part of the endpoint contained in the proxy. In this case, the client simply sends its messages with a minor version number that is not greater than the minor version number of the endpoint in the proxy.
- For indirect proxies, the proxy itself contains no version information at all (because the proxy contains no endpoints). Instead, the client obtains the

version information when it resolves the adapter name to one or more endpoints (via IcePack or an equivalent service). The version information of the endpoints determines the highest minor version number that is available to the client.

18.6 A Comparison with IIOP

It is interesting to compare the Ice protocol and encoding with CORBA's Inter-ORB Interoperability Protocol (IIOP) and Common Data Representation (CDR) encoding. The Ice protocol and encoding differ from IIOP and CDR in a number of important respects:

- Fewer data types

CORBA IDL has data types that are not provided by Ice, such as characters and wide characters, fixed-point numbers, arrays, and unions, each of which require a separate set of encoding rules.

Obviously, fewer data types makes the Ice encoding more efficient and less complex than CDR.

- Fixed byte-order marshaling

CDR supports both big-endian and little-endian encoding and provides a "receiver-makes-it-right" approach to byte ordering. The advantage of this is that, if sender and receiver have matching byte order, no reordering of the data is required at either end. However, the disadvantage is the additional complexity this creates in the marshaling logic. Moreover, the "receiver-makes-it-right" scheme carries a severe performance penalty if messages are to be forwarded via a number of intermediaries because this can require repeated unmarshaling, reordering, and remarshaling of messages. (Encapsulating the data would avoid this problem but, unfortunately, IIOP does not encapsulate most of the data that would benefit from it.)

The Ice encoding uses fixed little-endian data layout and avoids both the complexity and performance penalty.

- No padding

CDR has a complex set of rules for inserting alignment bytes into a data stream in an attempt to keep data aligned in a way that suits the native data layout of the underlying hardware. Unfortunately, this approach is severely flawed for a number of reasons:

- The CDR padding rules actually do not achieve any layout on natural word boundaries. For example, depending on its relative position in the enclosing data stream, the same structure value can differ in size and padding by up to seven bytes. There is not a single hardware platform in existence whose layout rules are the same as those of CDR. As a result, the padding bytes that are inserted serve no purpose other than to waste bandwidth.
- With “receiver-makes-it-right”, the receiver is obliged to re-order data if it arrives in the incorrect byte order anyway. This means that any alignment of the data (even if it were correct) is useless to, on average, half of all receivers because re-ordering of the data requires copying all of the data anyway.
- Padding and alignment rules differ depending on the hardware platform as well as the compiler. (For example, many compilers provide options that change the way data is packed in memory, to allow the developer to control the time vs. space trade-off.) This means that even the best set of layout rules can suit only a minority of platforms.
- Data alignment rules greatly complicate data forwarding. For example, if the receiver of a data item wants to forward that data item to another downstream receiver, it cannot just block-copy the received data into its transmit buffer because the padding of the data is sensitive to its relative position in the enclosing byte stream and a block copy would most likely result in an illegal encoding.

Ice avoids all the complexity (and concomitant inefficiency) by aligning all data on byte boundaries.

- More compact encoding

CDR encoding rules are wasteful of bandwidth, especially for sequences of short data items. For example, with CDR encoding, an empty string occupies eight bytes: four bytes to store the string length, plus a single terminating NUL byte, plus three bytes of padding. Table 18.19 compares the encoding sizes of

CDR and the Ice encoding for sequences of 100 strings for different string lengths.

Table 18.19. Sizes of CDR and Ice sequences of 100 strings of different lengths.

Sequence of 100 Strings of Length	CDR Size	Ice Size	Extra cost of CDR
0	804	101	696%
4	1204	501	140%
8	1604	901	78%
16	2004	1701	18%

Similar savings are achieved for small structures. Depending on the order and type of the structure members, CDR padding bytes can almost double the structure size, which becomes significant when sending sequences of such structures.

- Simple proxy encoding

Due to the inability of CORBA vendors to agree on a common encoding for object references (the equivalent of Ice proxies), CORBA object references have a complex internal structure that is partially standardized and partially opaque, allowing vendors to add proprietary information to object references. In addition, to avoid object references getting too large, references that support more than one transport have a scheme for sharing the object identity among several transports instead of carrying multiple copies of the same identity. The encoding that is required to support all this machinery is quite complex and results in extremely poor marshaling performance when large number of object references are exchanged (for example, when using a trading service).

In contrast, Ice proxies are simple and straight-forward to marshal and do not incur this loss of performance.

- Proper version negotiation

Versioning for IIOP and CDR was never designed properly, with the result that version negotiation in IIOP simply does not work. In particular, CDR encapsulations do not carry a separate version number. As a result, it is possible for data in encapsulations to travel to receivers that cannot decode the contents of the encapsulation, with no mechanism at the protocol level to detect the problem. Lack of correct versioning has been an ongoing problem with CORBA for years and the problem has historically been dealt with by pretending that it does not exist (meaning that different CORBA versions cannot interoperate with each other in many circumstances).

The Ice protocol and encoding have well-defined versioning rules that avoid such problems, allow both protocol and encoding to be extended, and reliably detect version mismatches.

- Fewer message types

IIOP has more message types than Ice. For example, IIOP has both a cancel request and a message error message. The cancel request is meant to cancel an invocation in progress but, of course, cannot do that because there is no way to abort an executing invocation in the server. At best, a cancel request allows the server to avoid marshaling the results of an invocation back to the client. However, the additional complexity introduced into the protocol is not worth the saving. (And, at any rate, neither is there a way for an application developer to send a cancel request, nor can the run time decide on its own when it would be appropriate to send one; despite that, every compliant CORBA implementation is burdened by the need to correctly respond to a useless request.)

IIOP's message error message constitutes similar baggage: the receiver of a malformed message is obliged to respond with a message error message before closing its connection. However, the message carries no useful information and, due to the nature of TCP/IP implementations, is usually lost instead of being delivered. This means that a compliant CORBA implementation is forced to send a useless message that will not be received in most cases when, instead, it should simply close the connection.

Ice avoids any such baggage in its protocol: the messages that are used actually serve a useful purpose.

- Request batching

IIOP has no notion of request batching. The advantages of request batching are particularly noticeable for event forwarding mechanisms, such as IceS-

form (see Chapter 26), as well as for fine-grained interfaces that provide modifier operations for a number of attributes. Request batching significantly reduces networking overhead for such applications.

- **Reliable connection establishment**

IIOP is vulnerable to connection loss during connection establishment. In particular, when a client opens a connection to a server, the client has no way of knowing whether the server is about to shut down and will not be able to process an incoming request. This means that the client has no choice but to send a request on a newly-established connection in the hope that the request will actually be processed by the server. If the server can process the request, there is no problem. However, if the server cannot process the request because it is on its way down, the client is confronted with a broken connection and cannot re-issue the request because that might violate at-most-once semantics. Ice does not have this problem because the validate connection message ensures that the client will not send a request to a server that is about to shut down. Moreover, any requests with outstanding replies can safely be re-issued by the client without violating at-most-once semantics.

- **Reliable endpoint resolution**

Indirect binding (see Section 20.3.2) in IIOP relies on a locate forward reply. Briefly, endpoint resolution is transparent to clients using IIOP. If an object reference uses indirect binding, the client issues the request as usual and receives a locate forward reply containing the endpoint of the actual server. In response to that reply, the client issues the request a second time to contact the actual server. There are a number of problems with this scheme:

- The physical address of the location service is written into each indirectly bound object reference. This makes it impossible to move the location service without invalidating all the references held by clients.³

Ice does not have this problem because the location service is known to clients through configuration. If the location service is moved, clients can be updated by changing their configuration and there is no need to track down a potentially unlimited number of proxies that might be out of date. Moreover,

3. IIOP version 1.2 supported a message to indicate permanent location forwarding. That message was meant to ease migration of location services. However, the semantics of that message broke the object model elsewhere, with the result that IIOP version 1.3 has deprecated that message again. (Unfortunately, reversals such as this are all too common in OMG specifications.)

there is typically no need to move the location service. Instead, it is possible to construct federated location services that work similar to the Internet Domain Name Service (DNS) and internally forward requests to the correct resolver.

- In order to get a location forward message, clients send the actual request to the location service. This is expensive if the request parameters are large because they are marshaled twice: once to the location service and once to the actual server. IIOP adds a locate request message that allows a client to explicitly resolve the location of a server. However, CORBA object references carry no indication as to whether they are bound directly or indirectly. This means that, no matter what the client does, it is wrong some of the time: if the client always uses a locate request with a directly bound reference, it ends up incurring the cost of two remote messages instead of one; if the client sends the request directly with an indirectly bound reference, it incurs the cost of marshaling the parameters twice instead of once.

Ice makes location resolution a step that is explicitly visible to the client-side run time because proxies either carry endpoint information (for direct proxies) or an adapter name (for indirect proxies). This allows the client-side run time to select the correct resolution mechanism without having to play guessing games and to avoid the overhead incurred by location forwarding.

- With IIOP, location resolution is built into the protocol itself. This complicates the protocol with two additional message types and (until 2002, when additional APIs were added to CORBA) made it impossible to implement a location service using standard APIs.

With Ice, no special protocol support is required for location resolution. Instead, the location service is an ordinary Ice server that defines an interface as usual and is contacted using operation invocations like any other server.

- No codeset negotiation

IIOP uses a codeset negotiation feature that (supposedly) permits use of arbitrary character encodings for transmission of wide characters and strings, provided sender and receiver have at least one codeset in common. Unfortunately, this feature was never properly thought through and has repeatedly led to interoperability problems. (Every single version of the CORBA specification, including the latest 3.0 version, has made corrections to the way codeset negotiation is supposed to work. Whether the latest set of corrections will

succeed in dealing with the problem remains to be seen. Regardless, many pages of complexity (and many lines of ORB source code) are devoted to this (mis)feature.)

The UTF-8 encoded Unicode used by Ice avoids all these problems without losing any functionality.

- No fragmentation

IIOP provides a fragment message whose purpose it is to reduce memory overhead in the server during marshaling of the results of an operation invocation. Unfortunately, the feature is quite complex (having been mis-specified and mis-implemented repeatedly in the past). The savings to be gained through fragmentation are quite limited, yet every client-side ORB implementation is forced to provide support for the feature. (In other words, the cure is worse than the disease.)

Ice simply does not use a fragmentation scheme, avoiding both the complexity and the resulting code bloat.

- Support for connection-less transports

In 2001, the OMG adopted a multi-cast specification. To the best of our knowledge, as of early 2003, no implementations of that specification are available.

Ice offers UDP as an alternative to TCP/IP. For messaging services, such as IceStorm, the performance benefits of UDP are considerable, allowing the service to scale well beyond what could be achieved with TCP/IP.

Chapter 19

Ice Extension for PHP

19.1 Chapter Overview

This chapter describes IcePHP, the Ice extension for the PHP scripting language. Section 19.2 provides an overview of IcePHP, including its design goals, capabilities, and limitations. IcePHP configuration is discussed in Section 19.3, and the PHP language mapping is specified in Section 19.4.

19.2 Introduction

PHP is a general-purpose scripting language that is used primarily in Web development. The PHP interpreter is typically installed as a Web server plug-in, and PHP itself also supports plug-ins known as “extensions.” PHP extensions, by definition, extend the interpreter’s run-time environment by adding new functions and data types.

The Ice extension for PHP, *IcePHP*, provides PHP scripts with access to Ice facilities. IcePHP is a thin integration layer implemented in C++ using the Ice C++ run-time library. This implementation technique has a number of advantages over a native PHP implementation of the Ice run-time:

1. Speed

The majority of the time-consuming work involved in making remote invocations, such as marshaling and unmarshaling, is performed in compiled C++, instead of in interpreted PHP.

2. Integration

IcePHP is fully self-contained. Its installation is performed once as an administrative step, and scripts have no dependencies on external PHP code.

3. Reliability

By leveraging the well-tested Ice C++ run-time library, there is less likelihood that new bugs will be introduced into the PHP extension.

4. Flexibility

IcePHP inherits all of the flexibility provided by the Ice C++ run-time, such as support for SSL, protocol compression, etc.

19.2.1 Capabilities

IcePHP supplies a robust subset of the Ice run-time facilities. PHP scripts are able to use all of the Slice data types in a natural way (see Section 19.4), make remote invocations, and use all of the advanced Ice services such as routers, locators and protocol plug-ins.

19.2.2 Limitations

The primary design goal of IcePHP was to provide PHP scripts with a simple and efficient interface to the Ice run-time. To that end, the feature set supported by IcePHP was carefully selected to address the requirements of typical PHP applications. As a result, IcePHP does not support the following Ice features:

- Servers

Given PHP's primary role as a scripting language for dynamic Web pages, the ability to implement an Ice server in PHP was deemed unnecessary for the majority of PHP applications.

- Asynchronous method invocation

The lack of synchronization primitives in PHP greatly reduces the utility of asynchronous invocations.

- Multiple communicators

A script has access to only one instance of `Ice::Communicator`, and is not able to manually create or destroy a communicator. See Section 19.4.11 for more information.

19.2.3 Design

The traditional design for a language mapping requires the intermediate step of translating Slice definitions into the target programming language before they can be used in an application.

IcePHP takes a different approach, made possible by the flexibility of PHP's extension interfaces. In IcePHP, no intermediate code generation step is necessary. Instead, the extension is configured with the application's Slice definitions, which are used to drive the extension's run-time behavior (see Section 19.3).

The Slice definitions are made available to PHP scripts as specified by the mapping in Section 19.4, just as if the Slice definitions had first gone through the traditional code-generation step and then been imported by the script.

There are several advantages to this design:

- The development process is simplified by the elimination of the intermediate code-generation step.
- The lack of machine-generated PHP code reduces the risk that the application's type definitions become outdated.
- Although PHP is a loosely-typed programming language, the Slice definitions enable IcePHP to validate the arguments of remote invocations.

19.3 Configuration

This section defines the PHP configuration directives supported by IcePHP. For installation instructions, please refer to the `INSTALL` file included in the IcePHP distribution.

19.3.1 Profiles

IcePHP allows any number of PHP applications to run independently in the same PHP interpreter, without risk of conflicts caused by Slice definitions that happened to use the same identifiers. IcePHP uses the term *profile* to describe an

application's configuration, including its Slice definitions and Ice configuration properties.

19.3.2 Default Profile

A default profile is supported, which is convenient during development, or when only one Ice application is running in a PHP interpreter. Table 19.1 describes the PHP configuration directives¹ for the default profile.

Table 19.1. PHP configuration directive for the default profile.

Name	Description
<code>ice.config</code>	Specifies the pathname of an Ice configuration file.
<code>ice.options</code>	Specifies command-line options for Ice configuration properties. For example, <code>--Ice.Trace.Network=1</code> . This is a convenient alternative to an Ice configuration file if only a few configuration properties are required.
<code>ice.profiles</code>	Specifies the pathname of a profile configuration file. See Section 19.3.3.
<code>ice.slice</code>	Specifies preprocessor options and the pathnames of Slice files to be loaded.

Here is a simple example:

```
ice.options="--Ice.Trace.Network=1 --Ice.Warn.Connections=1"
ice.slice="-I/myapp/include /myapp/include/MyApp.ice"
```

19.3.3 Named Profiles

If a filename is specified by the `ice.profiles` configuration directive, the file is expected to have the standard INI-file format. The section names identify

1. These directives are typically defined in the `php.ini` file, but can also be defined using Web-server specific directives.

profiles, where each profile supports the `ice.config`, `ice.options` and `ice.slice` directives defined in Section 19.3.2. For example:

```
[Profile1]
ice.config=/profile1/ice.cfg
ice.slice=/profile1/App.ice

[Profile2]
ice.config=/profile2/ice.cfg
ice.slice=/profile2/App.ice
```

This file defines two named profiles, `Profile1` and `Profile2`, having separate Ice configurations and Slice definitions.

19.3.4 Profile Functions

Two global functions are provided for profile activities:

```
Ice_loadProfile(/* string */ $name = null);
Ice_dumpProfile();
```

The `Ice_loadProfile` function must be invoked by a script in order to make the Slice types available and to configure the script's communicator. If no profile name is supplied, the default profile is loaded. A script is not allowed to load more than one profile.

For example, here is a script that loads `Profile1` shown in Section 19.3.3:

```
<?php
Ice_loadProfile("Profile1");
...
?>
```

For troubleshooting purposes, the `Ice_dumpProfile` function can be called by a script. This function displays all of the relevant information about the profile loaded by the script, including the communicator's configuration properties as well as the PHP mappings for all of the Slice definitions loaded by the profile.

19.3.5 Slice Semantics

The expense of parsing Slice files is incurred once, when the PHP interpreter initializes the IcePHP extension. For this reason, we recommend that the IcePHP extension be statically configured into the PHP interpreter, either by compiling the extension directly into the interpreter, or configuring PHP to load the extension dynamically at startup. For the same reason, we discourage the use of IcePHP in a

CGI context, in which a new PHP interpreter is created for every HTTP request. Furthermore, we strongly discourage scripts from dynamically loading the IcePHP extension using PHP's `dl` function.

19.4 Client-Side Slice-to-PHP Mapping

This section describes the PHP mapping for Slice types.

19.4.1 Mapping for Identifiers

Slice identifiers map to PHP identifiers of the same name, unless the Slice identifier conflicts with a PHP reserved word, in which case the mapped identifier is prefixed with an underscore. For example, the Slice identifier `echo` is mapped as `_echo`.

A flattened mapping is used for identifiers defined within Slice modules because PHP does not have an equivalent to C++ namespaces or Java packages. The flattened mapping uses underscores to separate the components of a fully-scoped name. For example, consider the following Slice definition:

```
module M {  
    enum E { one, two, three };  
};
```

In this case, the Slice identifier `M::E` is flattened to the PHP identifier `M_E`.

Note that when checking for a conflict with a PHP reserved word, only the fully-scoped, flattened identifier is considered. For example, the Slice identifier `M::function` is mapped as `M_function`, despite the fact that `function` is a PHP reserved word. There is no need to map it as `M__function` (with two underscores) because `M_function` does not conflict with a PHP reserved word.

However, it is still possible for the flattened mapping to generate identifiers that conflict with PHP reserved words. For instance, the Slice identifier `require::once` must be mapped as `_require_once` in order to avoid conflict with the PHP reserved word `require_once`.

19.4.2 Mapping for Simple Built-in Types

PHP has a limited set of primitive types: `boolean`, `integer`, `double`, and `string`. The Slice built-in types are mapped to PHP types as shown in Table 19.2.

Table 19.2. Mapping of Slice built-in types to PHP.

Slice	PHP
<code>bool</code>	<code>boolean</code>
<code>byte</code>	<code>integer</code>
<code>short</code>	<code>integer</code>
<code>int</code>	<code>integer</code>
<code>long</code>	<code>integer</code>
<code>float</code>	<code>double</code>
<code>double</code>	<code>double</code>
<code>string</code>	<code>string</code>

PHP's `integer` type may not accommodate the range of values supported by Slice's `long` type, therefore `long` values that are outside this range are mapped as strings. Scripts must be prepared to receive an integer or string from any operation that returns a `long` value.

19.4.3 Mapping for User-Defined Types

Slice supports user-defined types: enumerations, structures, sequences, and dictionaries.

Mapping for Enumerations

Enumerations map to a PHP class containing a constant definition for each enumerator. For example:

```
enum Fruit { Apple, Pear, Orange };
```

The PHP mapping is shown below:

```
class Fruit {  
    const Apple = 0;  
    const Pear = 1;  
    const Orange = 2;  
}
```

The enumerators can be accessed in PHP code as `Fruit::Apple`, etc.

Mapping for Structures

Slice structures map to PHP classes. For each Slice data member, the PHP class contains a variable of the same name. For example, here is our `Employee` structure from Section 4.7.4 yet again:

```
struct Employee {  
    long number;  
    string firstName;  
    string lastName;  
};
```

This structure is mapped to the following PHP class:

```
class Employee {  
    var $number;  
    var $firstName;  
    var $lastName;  
}
```

Mapping for Sequences

Slice sequences are mapped to native PHP indexed arrays. The first element of the Slice sequence is contained at index 0 (zero) of the PHP array, followed by the remaining elements in ascending index order.

Here is the definition of our `FruitPlatter` sequence from Section 4.7.3:

```
sequence<Fruit> FruitPlatter;
```

You can create an instance of this sequence as shown below:

```
// Make a small platter with one Apple and one Orange  
//  
$platter = array(Fruit::Apple, Fruit::Orange);
```

Mapping for Dictionaries

Slice dictionaries map to native PHP associative arrays. The PHP mapping does not currently support all Slice dictionary types, however, because native PHP associative arrays support only integer and string key types. A Slice dictionary whose key type is `boolean`, `byte`, `short`, `int` or `long` is mapped as an associative array with an integer key.² A Slice dictionary with a string key type is mapped as associative array with a string key. All other key types cause a warning to be generated.

Here is the definition of our `EmployeeMap` from Section 4.7.4:

```
dictionary<long, Employee> EmployeeMap;
```

You can create an instance of this dictionary as shown below:

```
$e1 = new Employee;
$e1->number = 42;
$e1->firstName = "Stan";
$e1->lastName = "Lipmann";

$e2 = new Employee;
$e2->number = 77;
$e2->firstName = "Herb";
$e2->lastName = "Sutter";

$em = array($e1->number => $e1, $e2->number => $e2);
```

19.4.4 Mapping for Constants

Slice constant definitions map to corresponding calls to the PHP function `define`. Here are the constant definitions we saw in Section 4.7.5:

```
const bool      AppendByDefault = true;
const byte      LowerNibble = 0x0f;
const string    Advice = "Don't Panic!";
const short     TheAnswer = 42;
const double    PI = 3.1416;

enum Fruit { Apple, Pear, Orange };
const Fruit     FavoriteFruit = Pear;
```

2. Boolean values are treated as integers, with false equivalent to 0 (zero) and true equivalent to 1 (one).

Here are the equivalent PHP definitions for these constants:

```
define("AppendByDefault", true);
define("LowerNibble", 15);
define("Advice", "Don't Panic!");
define("TheAnswer", 42);
define("PI", 3.1416);

class Fruit {
    const Apple = 0;
    const Pear = 1;
    const Orange = 2;
}
define("FavoriteFruit", Fruit::Pear);
```

19.4.5 Mapping for Exceptions

A Slice exception maps to a PHP class. For each exception member, the corresponding class contains a variable of the same name. All user exceptions ultimately derive from `Ice_UserException` (which, in turn, derives from `Ice_Exception`, which derives from PHP's base `Exception` class):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }
}

abstract class Ice_UserException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

The optional string argument to the constructor is passed unmodified to the `Exception` constructor.

If the exception derives from a base exception, the corresponding PHP class derives from the mapped class for the base exception. Otherwise, if no base exception is specified, the corresponding class derives from `Ice_UserException`.

Here is a fragment of the Slice definition for our world time server from Section 4.8.5:


```

exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};

```

These exceptions are mapped as the following PHP classes:

```

class GenericError extends Ice_UserException {
    function __construct($message = '') {
        ...
    }

    var $reason;
}

class BadTimeVal extends GenericError {
    function __construct($message = '') {
        ...
    }
}

class BadZoneName extends GenericError {
    function __construct($message = '') {
        ...
    }
}

```

An application can catch these exceptions as shown below:

```

try {
    ...
} catch(BadZoneName $ex) {
    // Handle BadZoneName
} catch(GenericError $ex) {
    // Handle GenericError
} catch(Ice_Exception $ex) {
    // Handle all other Ice exceptions
    print_r($ex);
}

```

19.4.6 Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from

`Ice_LocalException` (which, in turn, derives from `Ice_Exception`, which derives from PHP's base `Exception` class):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }
}

abstract class Ice_LocalException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

An inheritance diagram for user and run-time exceptions appears in Figure 4.4 on page 80. Note however that the PHP mapping only defines classes for the local exceptions listed below:

- `Ice_LocalException`
- `Ice_UnknownException`
- `Ice_UnknownLocalException`
- `Ice_UnknownUserException`
- `Ice_RequestFailedException`
- `Ice_ObjectNotExistException`
- `Ice_FacetNotExistException`
- `Ice_OperationNotExistException`
- `Ice_ProtocolException`
- `Ice_MarshalException`
- `Ice_NoObjectFactoryException`

Instances of all remaining local exceptions are converted to the class `Ice_UnknownLocalException`. The `unknown` member of this class contains a string representation of the original exception.

19.4.7 Mapping for Interfaces

A Slice interface maps to a PHP interface. Each operation in the interface maps to a method of the same name, as described in Section 19.4.9. The inheritance structure of the Slice interface is preserved in the PHP mapping, and all interfaces ultimately derive from `Ice_Object`:

```
interface Ice_Object {};
```

For example, consider the following Slice definitions:

```
interface A {
    void opA();
};
interface B extends A {
    void opB();
};
```

These interfaces are mapped to PHP interfaces as shown below:

```
interface A implements Ice_Object
{
    function opA();
}
interface B implements A
{
    function opB();
}
```

19.4.8 Mapping for Classes

A Slice class maps to an abstract PHP class. Each operation in the class maps to an abstract method of the same name, as described in Section 19.4.9. For each Slice data member, the PHP class contains a variable of the same name. The inheritance structure of the Slice class is preserved in the PHP mapping, and all classes ultimately derive from `Ice_ObjectImpl` (which, in turn, implements `Ice_Object`):

```
class Ice_ObjectImpl implements Ice_Object
{
    var $ice_facets = array();
}
```

Consider the following class definition:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;         // 0 - 59
    short second;         // 0 - 59
    string format();      // Return time as hh:mm:ss
};
```

The PHP mapping for this class is shown below:

```

abstract class TimeOfDay extends Ice_ObjectImpl
{
    var $hour;
    var $minute;
    var $second;
    abstract function format();
}

```

Facets

As shown in the mapping for `Ice_ObjectImpl` on page 523, an Ice object's facets are represented as a native PHP array. More specifically, the facets are contained in the member `ice_facets`, which is an associative array with a key type of `string` and a value type of `Ice_Object`. The `ice_facets` member is public and therefore can be manipulated directly by applications.

Class Factories

Class factories are installed by invoking `addObjectFactory` on the communicator (see Section 19.4.11). A factory must implement the interface `Ice_ObjectFactory`, defined as follows:

```

interface Ice_ObjectFactory implements Ice_LocalObject
{
    function create(/* string */ $id);
    function destroy();
}

```

For example, we can define and install a factory for the `TimeOfDay` class as shown below:

```

class TimeOfDayI extends TimeOfDay {
    function format()
    {
        return sprintf("%02d:%02d:%02d", $this->hour,
            $this->minute, $this->second);
    }
}

class TimeOfDayFactory extends Ice_LocalObjectImpl
    implements Ice_ObjectFactory {
    function create($id)
    {
        return new TimeOfDayI;
    }
}

```

```

        function destroy() {}
    }

    $ICE->addObjectFactory(new TimeOfDayFactory, "::TimeOfDay");

```

19.4.9 Mapping for Operations

Each operation defined in a Slice class or interface is mapped to a PHP function of the same name. Furthermore, each parameter of an operation is mapped to a PHP parameter of the same name, with out parameters passed by reference. Since PHP is a loosely-typed language, no parameter types are specified.³

Consider the following interface:

```

interface I {
    float op(string s, out int i);
};

```

The PHP mapping for this interface is shown below:

```

interface I {
    function op($s, &$i);
}

```

19.4.10 Mapping for Proxies

The PHP mapping for Slice proxies uses a single interface, `Ice_ObjectPrx`, to represent all proxy types.

Typed vs. Untyped Proxies

A proxy can be typed or untyped. All proxies, whether they are typed or untyped, support the core proxy methods described in the next section.

An *untyped proxy* is equivalent to the Slice type `Object*`. The communicator operation `stringToProxy` returns an untyped proxy, as do several of the core proxy methods. A script cannot invoke user-defined operations on an untyped

3. PHP5 introduces the notion of “type hints” that allow you to specify the formal type of object parameters. This would enable the Slice mapping to specify type hints for parameters of type struct, interface, class and proxy. Unfortunately, PHP5 does not currently allow a parameter defined with a type hint to receive a null value, therefore the Slice mapping does not use type hints for parameters.

proxy, nor can an untyped proxy be passed as an argument where a typed proxy is expected.

A *typed proxy* is one that has been associated with a Slice class or interface type. There are two ways a script can obtain a typed proxy:

1. By receiving it as the result of a remote invocation that returns a typed proxy.
2. By using the core proxy methods `ice_checkedCast` or `ice_uncheckedCast`.

For example, suppose our script needs to obtain a typed proxy for interface A, shown below:

```
interface A {
    void opA();
};
```

Here are the steps our script performs:

```
$obj = $ICE->stringToProxy("a:tcp -p 12345");
$obj->opA(); // WRONG!
$a = $obj->ice_checkedCast("::A");
$a->opA(); // OK
```

Attempting to invoke `opA` on `$obj` would result in a fatal error, because `$obj` is an untyped proxy.

Core Proxy Methods

The `Ice_ObjectPrx` interface provides a number of core proxy methods:

```
interface Ice_ObjectPrx {
    function ice_isA(/* string */ $id, /* array */ $ctx = null);
    function ice_ping(/* array */ $ctx = null);
    function ice_ids(/* array */ $ctx = null);
    function ice_id(/* array */ $ctx = null);
    function ice_facets(/* array */ $ctx = null);
    function ice_getIdentity();
    function ice_newIdentity(/* Ice_Identity */ $id);
    function ice_getFacet();
    function ice_newFacet(/* array */ $path);
    function ice_appendFacet(/* string */ $facet);
    function ice_twoway();
    function ice_isTwoway();
    function ice_oneway();
    function ice_isOneway();
    function ice_batchOneway();
    function ice_isBatchOneway();
```

```

function ice_datagram();
function ice_isDatagram();
function ice_batchDatagram();
function ice_isBatchDatagram();
function ice_secure(/* boolean */ $b);
function ice_compress(/* boolean */ $b);
function ice_timeout(/* integer */ $t);
function ice_default();
function ice_uncheckedCast(/* string */ $type,
                          /* string */ $facet = null);
function ice_checkedCast(/* string */ $type,
                        /* string */ $facet = null);
}

```

These methods can be categorized as follows:

- Remote inspection: methods that return information about the remote object.
- Local inspection: methods that return information about the proxy's local configuration.
- Factory: methods that return new proxy instances configured with different features.

Proxies are immutable, so factory methods allow an application to obtain a new proxy with the desired configuration. Factory methods essentially clone the original proxy and modify one or more features of the new proxy.

To demonstrate the use of proxy factory methods, consider this example:

```

$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
$p = $p->ice_uncheckedCast("::A");

```

The script receives an untyped proxy as the return value of `stringToProxy`. Since the stringified proxy did not contain any proxy options, it is configured by default as a twoway, insecure proxy with no timeout. However, our goal is to obtain a secure oneway proxy for interface A, therefore we invoke `ice_oneway`, followed by `ice_secure`. At this point, we have an untyped proxy configured for secure oneway invocations. Finally, we call `ice_uncheckedCast` to obtain a typed proxy.⁴

4. We cannot use `ice_checkedCast` on a proxy configured for oneway invocations.

Note that this process can be simplified by chaining the proxy factory methods, as shown below:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway()->ice_secure(true)->ice_uncheckedCast("::A");
```

The order that the proxy factory methods are invoked is usually not important, except in the case of `ice_checkedCast` and `ice_uncheckedCast`. For example, if the script above had invoked `ice_uncheckedCast` first, followed by the other factory methods, then the result would have been an untyped proxy:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_uncheckedCast("::A"); // WRONG!
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
```

The reason for this unexpected behavior is that most of the factory methods return an untyped proxy, thereby discarding any type that might have been associated with the original proxy. By invoking `ice_oneway` on the typed proxy returned from `ice_uncheckedCast`, the script has lost its typed proxy.

The core proxy methods are explained in greater detail in Table 19.3.

Table 19.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_isA</code>	Returns <code>true</code> if the remote object supports the type indicated by the <code>id</code> argument, otherwise <code>false</code> . This method can only be invoked on a twoway proxy.	Yes
<code>ice_ping</code>	Determines whether the remote object is reachable. Does not return a value.	Yes
<code>ice_ids</code>	Returns the type ids of the types supported by the remote object. The return value is an array of strings. This method can only be invoked on a twoway proxy.	Yes
<code>ice_id</code>	Returns the type id of the most-derived type supported by the remote object. This method can only be invoked on a twoway proxy.	Yes
<code>ice_facets</code>	Returns the names of the facets supported by the remote object. The return value is an array of strings. This method can only be invoked on a twoway proxy.	Yes

Table 19.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_getIdentity</code>	Returns the Ice object identity associated with the proxy. The return value is an instance of <code>Ice_Identity</code> .	No
<code>ice_newIdentity</code>	Returns a new untyped proxy having the given identity.	No
<code>ice_getFacet</code>	Returns the name of the facet associated with the proxy, or an empty string if no facet has been set.	No
<code>ice_newFacet</code>	Returns a new untyped proxy having the given facet path. The path must be an array of strings.	No
<code>ice_appendFacet</code>	Returns a new untyped proxy with the given facet name appended to the facet path.	No
<code>ice_twoway</code>	Returns a new untyped proxy for making twoway invocations.	No
<code>ice_isTwoway</code>	Returns <code>true</code> if the proxy uses twoway invocations, otherwise <code>false</code> .	No
<code>ice_oneway</code>	Returns a new untyped proxy for making oneway invocations.	No
<code>ice_isOneway</code>	Returns <code>true</code> if the proxy uses oneway invocations, otherwise <code>false</code> .	No
<code>ice_batchOneway</code>	Returns a new untyped proxy for making batch oneway invocations.	No
<code>ice_isBatchOneway</code>	Returns <code>true</code> if the proxy uses batch oneway invocations, otherwise <code>false</code> .	No
<code>ice_datagram</code>	Returns a new untyped proxy for making datagram invocations.	No
<code>ice_isDatagram</code>	Returns <code>true</code> if the proxy uses datagram invocations, otherwise <code>false</code> .	No
<code>ice_batchDatagram</code>	Returns a new untyped proxy for making batch datagram invocations.	No
<code>ice_isBatchDatagram</code>	Returns <code>true</code> if the proxy uses batch datagram invocations, otherwise <code>false</code> .	No

Table 19.3. Description of core proxy methods.

Method	Description	Remote
<code>ice_secure</code>	Returns a new untyped proxy whose endpoints may be filtered depending on the boolean argument. If <code>true</code> , only endpoints using secure transports are allowed, otherwise all endpoints are allowed.	No
<code>ice_compress</code>	Returns a new untyped proxy whose protocol compression capability is determined by the boolean argument. If <code>true</code> , the proxy uses protocol compression if it is supported by the endpoint. If <code>false</code> , protocol compression is never used.	No
<code>ice_timeout</code>	Returns a new untyped proxy with the given timeout value in milliseconds. A value of <code>-1</code> disables timeouts.	No
<code>ice_default</code>	Returns a new untyped proxy having the default proxy configuration.	No
<code>ice_uncheckedCast</code>	Returns a new typed proxy associated with the given type id. If a facet name is provided, the returned proxy has the facet appended to its path. The Slice definition for the given type id must be available in the current profile.	No
<code>ice_checkedCast</code>	If the remote object supports the type indicated by the given type id, this method returns a new typed proxy associated with the type, otherwise it returns null. If a facet name is provided, the returned proxy has the facet appended to its path. The Slice definition for the given type id must be available in the current profile.	Yes

Request Context

All remote operations on a proxy support an optional final parameter representing the request context (see Section 16.8). The PHP mapping for the request context is an associative array in which the keys and values are strings. For example, the code below illustrates how to invoke `ice_ping` with a request context:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$ctx = array("theKey" => "theValue");
$p->ice_ping($ctx);
```

Identity

Certain core proxy operations use the type `Ice_Identity`, which is the PHP mapping for the Slice type `Ice::Identity`. This type is mapped using the standard rules for Slice structures, therefore it is defined as follows:

```
class Ice_Identity {
    var $name;
    var $category;
}
```

Two global functions are provided for converting `Ice_Identity` values to and from a string representation:

```
function Ice_stringToIdentity(/* string */ $str);
function Ice_identityToString(/* Ice_Identity */ $id);
```

19.4.11 Mapping for `Ice::Communicator`

Since the Ice extension for PHP provides only client-side facilities, many of the operations provided by `Ice::Communicator` operations are not relevant, therefore the PHP mapping supports a subset of the communicator operations. The mapping for `Ice::Communicator` is shown below:

```
interface Ice_Communicator {
    function stringToProxy(/* string */ $str);
    function proxyToString(/* Ice_ObjectPrx */ $prx);
    function addObjectFactory(/* Ice_ObjectFactory */ $factory,
                             /* string */ $id);
    function removeObjectFactory(/* string */ $id);
    function findObjectFactory(/* string */ $id);
    function flushBatchRequests();
}
```

See Appendix B for a description of these operations.

Communicator Lifecycle

PHP scripts are not allowed to create or destroy communicators. Rather, a communicator is created prior to each PHP request, and is destroyed after the request completes.

Accessing the Communicator

The communicator instance created for a request is available to the script via the global variable `$ICE`. Scripts should not attempt to assign a different value to this variable. As with any global variable, scripts that need to use `$ICE` from within a function must declare the variable as global:

```
function printProxy($prx) {  
    global $ICE;  
  
    print $ICE->proxyToString($prx);  
}
```

Communicator Configuration

The profile loaded by the script determines the communicator's configuration. See Section 19.3 for more information.

Part IV

Ice Services

Chapter 20

IcePack

20.1 Chapter Overview

In this chapter we present IcePack. Section 20.2 provides a brief introduction to IcePack's features, and Sections 20.3 and 20.4 review some concepts which are helpful in understanding how IcePack works. The purpose and use of the IcePack components are described in Sections 20.5 through 20.7. Deploying applications with IcePack is discussed in Sections 20.8 and 20.9, and some troubleshooting tips are provided in Section 20.10.

20.2 Introduction

IcePack provides several essential services that simplify the development and deployment of a complex distributed application:

- a locator service to locate objects and ensure location transparency,
- a server activation and monitoring service to manage server processes,
- and a powerful server deployment mechanism.

IcePack is comprised of two main components:

- An IcePack *registry* manages all of the information about the applications deployed in a particular domain.

- An IcePack *node* activates and monitors server processes.

There is only one IcePack registry in a domain, but there can be many IcePack nodes (typically one per host).

20.3 Concepts

Throughout most of this book we have presented examples that intentionally avoid (for the sake of simplicity) some important issues of distributed application development, including location transparency, binding, and deployment. IcePack addresses these issues with minimal impact on Ice applications.

20.3.1 Location Transparency

Location transparency means the location of an object implementation is irrelevant to the client; the object could be implemented within the client's process, or in a process on another host. Regardless of where the object is implemented, the client interacts with the object in exactly the same way, and with the same semantics. Location transparency simplifies the development of distributed applications and is the foundation for the services that IcePack provides.

20.3.2 Binding

As described in Section 2.2, binding is the process of connecting a proxy with its servant. In practice, this typically involves opening a socket connection to the server's endpoint. However, there are two binding modes that the Ice run time can use to obtain the server's endpoint information: direct binding and indirect binding.

Direct Binding

We have seen endpoint details repeated many times in the examples in this book: the server calls `createObjectAdapterWithEndpoints`, specifying a protocol and port number, and the client invokes `stringToProxy` on a stringified proxy containing the same protocol and port number. The Ice run time in the client opens a socket directly to the server's endpoint; if the server is not currently running, or is running on a different host or port, then the client receives an exception. This is known as *direct binding*, and while it is easy to understand and quite convenient, it is not appropriate for many enterprise applications.

We can improve this situation somewhat by externalizing the endpoint details using configuration properties (see Appendix C). For example, the endpoints of an object adapter named `MyAdapter` can be specified using a configuration property such as:

```
MyAdapter.Endpoints=tcp -p 9999
```

Similarly, the client can define a property containing the stringified proxy:

```
MyProxy=theObject:tcp -p 9999
```

This is a step in the right direction; at least the endpoint details are no longer contained in the code. The server can now be relocated to a different host or port without requiring changes to either client or server programs.

This is still not an ideal solution, however. For example, if proxies were stored in a database, they would all have to be updated when the server endpoints change. Similarly, if there are multiple client and server applications running on multiple hosts, the administrative work involved in updating configuration files to reflect evolving server configurations can quickly become unwieldy.

Indirect Binding

As its name implies, *indirect binding* adds a layer of indirection such that the client no longer needs to specify endpoint details for a proxy. Rather, a proxy's endpoints are determined dynamically by the Ice run time. We discuss how Ice makes this possible in Section 20.4, but for now we can illustrate its use with more proxy examples.

First, we replace the endpoint details with an object adapter identifier:

```
MyProxy=theObject@theAdapter
```

The symbol `theAdapter` identifies an object adapter whose endpoints are retrieved automatically.

Next, we can use only the object identity:

```
MyProxy=theObject
```

In this case, the object identity alone is sufficient to locate an object.

20.3.3 Deployment

Application deployment is a broad term with plenty of application-specific requirements. With respect to Ice applications, this can involve the creation and installation of configuration files, initializing IceBox services (see Chapter 25), and writing scripts to handle startup and shutdown procedures. IcePack simplifies

these tasks using an administrative tool that processes XML descriptions of your application's services and deploys them automatically.

20.4 Ice Locator Facility

The Ice run time supports a delegation facility, represented by the `Ice::Locator` interface, that allows endpoint information to be obtained dynamically. This facility enables the flexibility of indirect binding described in Section 20.3.2 and is a fundamental component of the services provided by IcePack. The details of the `Ice::Locator` interface definition are not relevant to this discussion, but its capabilities are worth exploring in order to understand how IcePack simplifies the design and development of Ice applications.

20.4.1 Objects and Object Adapters

Recall from Section 20.3.2 the two stringified proxy examples that demonstrate indirect binding:

```
MyProxy=theObject@theAdapter
```

and

```
MyProxy=theObject
```

These proxies show the two methods of obtaining endpoint information: by object adapter identifier, and by object identity. In either case, the Ice run time in the client obtains the endpoint information by making requests on an `Ice::Locator` object. This activity is transparent to the application and occurs whenever a proxy using indirect binding is created, such as by calling `stringToProxy` or when receiving a proxy as the result of an operation. However, the Ice run time is careful to minimize the number of locator requests (see Section 20.4.7).

A locator is essentially a repository containing two lookup tables: one that associates endpoints with object identities, and another that associates endpoints with object adapter identifiers. Figure 20.1 illustrates the role played by a locator:

1. At startup, the server registers its well-known objects and object adapters with the locator.
2. When creating a proxy using indirect binding, the Ice run time in the client invokes on the locator to obtain the endpoints associated with an object identity or object adapter identifier.

3. After obtaining the endpoints for the proxy, the client establishes a connection directly to the server. The locator is not consulted again for that proxy unless a connection error occurs.

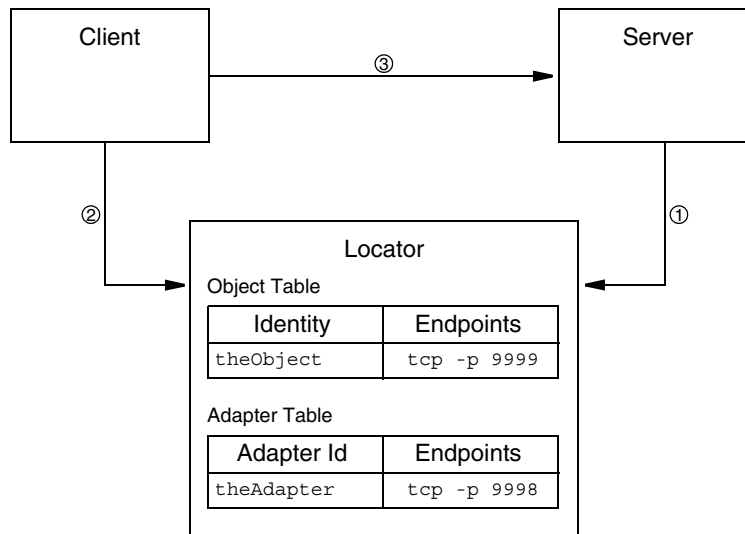


Figure 20.1. Obtaining endpoint information from a locator.

20.4.2 Design Considerations

Although both proxy styles shown in Section 20.4.1 are syntactically similar in that they eliminate the dependency on endpoint details, they have distinctly different design considerations. Specifically, the object identity style is intended to be used primarily for bootstrapping purposes; i.e., the identities of a few well-known objects are made available via the locator facility, and clients use these well-known objects to obtain proxies for other objects as necessary.

The object adapter style, on the other hand, scales to many objects but is still dependent on the server configuration (i.e., the object adapter identifier).

The best situation is often a combination of the two methods: the client uses the object identity style to bootstrap a principal object that provides proxies to subordinate objects using the object adapter style.

20.4.3 Using the Ice Locator Facility in Clients

Integrating the locator functionality into a client application is merely a matter of configuration: the value of the property `Ice.Default.Locator` must specify a stringified proxy for the locator object. Once the Ice run time has this proxy, it can convert object adapter identifiers and object identities into physical endpoint attributes. See Section 20.5.4 for more information on defining the locator property.

20.4.4 Using the Ice Locator Facility in Servers

An object adapter automatically registers its endpoints with the locator when the adapter is activated. Automatic registration is enabled by configuring the object adapter with an identifier¹ and a locator proxy. Of course, the object adapter must also have one or more endpoints defined, but this requirement is not specific to the locator.

For example, an object adapter named `MyAdapter` could be configured with the following properties:

```
Ice.Default.Locator=... // the stringified locator proxy
MyAdapter.AdapterId=theAdapter
MyAdapter.Endpoints=default
```

This configuration assigns the locator identifier `theAdapter` to the object adapter named `MyAdapter`. The endpoint for this object adapter will use the default protocol and a system-assigned port number. Upon activation, the object adapter registers this endpoint with the locator whose proxy is provided in `Ice.Default.Locator`, thereby enabling clients to use the object adapter proxy style with this adapter.

As its name implies, the `Ice.Default.Locator` property supplies the default locator proxy for all object adapters in a server. However, each object adapter whose endpoints should be registered with the locator must also be configured with an identifier. A server can therefore create as many object adapters as necessary, with complete control over which ones are registered with a locator.

The locator proxy can also be configured for a particular object adapter using a property such as:

1. There is a potential for conflicts if object adapters in a domain do not use unique identifiers; it is the developer's responsibility to ensure that this does not occur.

```
MyAdapter.Locator=... // the stringified locator proxy
```

This property has precedence over the `Ice.Default.Locator` property. See Section 20.5.4 for more information on defining the locator properties.

To facilitate use of the proxy style of object identity, each object's identity and proxy must be registered. This can be done programmatically by the server, administratively using the IcePack command-line tool described in Section 20.7, or automatically using the IcePack deployment mechanism discussed in Section 20.8.

20.4.5 Deployment

The locator facility simplifies application deployment, because we no longer need to bother assigning unique endpoints to object adapters. Since proxies contain only a symbolic reference to the endpoint information, object adapters are free to use system-assigned (as opposed to user-assigned) ports.

Although an object adapter would likely be assigned a different port each time it is activated, its automatic registration with the locator ensures that clients receive the correct endpoint information. If a client holds a proxy to an object in an object adapter that is deactivated and then reactivated on a different port, the retry behavior in the client's Ice run time will automatically refresh the proxy's endpoints.

20.4.6 Dependencies

The locator facility is represented by `Ice::Locator`, a generic Slice interface that can have multiple implementations. When the Ice run time is configured with an `Ice::Locator` proxy, it automatically uses the locator to perform indirect binding. The Ice run time therefore has a dependency on the `Ice::Locator` interface, but not on IcePack; the IcePack registry simply implements the `Ice::Locator` interface, thus allowing clients to use indirect binding to servers that are activated by IcePack nodes.

20.4.7 Overhead of Indirect Binding

As you might expect, indirect binding does create some additional overhead, since the Ice run time must make a proxy invocation on the locator to obtain endpoint information. To minimize the number of locator invocations, the Ice run time caches the results; subsequent attempts to bind to the same identity or object

adapter name will use the cached endpoint information. For the majority of applications, the flexibility provided by indirect binding far outweighs the minimal overhead incurred by its use.

20.5 IcePack Registry

The IcePack registry is a centralized repository of information, including

- IcePack nodes,
- deployed servers and object adapters,
- object registrations.

The registry also implements the `Ice::Locator` interface described in Section 20.4, and supports additional `Slice` interfaces for querying and administrative tasks. A registry service can optionally be colocated with an IcePack node, which conserves resources and can be convenient during development and testing.

20.5.1 Endpoints

The IcePack registry creates up to four sets of endpoints, configured with the following properties:

- `IcePack.Registry.Client.Endpoints`
Client-side endpoints supporting the `Ice::Locator` and `IcePack::Query` interfaces.
- `IcePack.Registry.Server.Endpoints`
Server-side endpoints for object and object adapter registration.
- `IcePack.Registry.Admin.Endpoints`
Administrative endpoints supporting the `IcePack::Admin` interface (optional).
- `IcePack.Registry.Internal.Endpoints`
Internal endpoints used by IcePack nodes to communicate with the registry (optional). This property is optional only if the registry is being used without any nodes.

See Appendix C for more information on these properties.

20.5.2 Security Considerations

Access to an IcePack registry's administrative interface should be restricted if the registry is running on an insecure host, such as an application firewall. The `IcePack.Registry.Admin.Endpoints` property (described in the previous section) offers several alternatives:

- Disable administrative access completely by not defining the property.
- Block external access to the administrative endpoints defined by the property.
- Use only SSL endpoints in the property (see Chapter 23).

The method you choose depends on your deployment requirements. For example, if no administrative access is necessary, then the first technique is the most secure. If that is not feasible, then physically blocking (via a firewall) external access to the administrative endpoints is recommended. Finally, if a registry must be administered externally, then SSL endpoints should be used to ensure that only authenticated clients gain access.

20.5.3 Registry Object Identities

The identities of the primary IcePack registry objects are shown below:

- `IcePack/Locator` for the `Ice:Locator` implementation.
- `IcePack/Query` for the `IcePack:Query` implementation.
- `IcePack/Admin` for the `IcePack:Admin` implementation.

These identities, together with the endpoints defined in the properties from Section 20.5.1, enable you to construct proxies for the registry objects.

20.5.4 Ice Locator Properties

Now that we know the identity of the registry's `Ice:Locator` implementation, and the endpoints defined in the `IcePack.Registry.Client.Endpoints` property, we can define the configuration property for Ice clients and servers that enables the locator functionality.

In general, clients and servers can both use the `Ice.Default.Locator` property. For example, if the IcePack registry client endpoint is `tcp -p 12000`, then the locator property can be defined as follows:

```
Ice.Default.Locator=IcePack/Locator:tcp -p 12000
```

In servers, a locator proxy can also be specified individually for each object adapter using a property definition such as:

```
MyAdapter.Locator=IcePack/Locator:tcp -p 12000
```

If the locator property is not defined for an adapter, the value of `Ice.Default.Locator` will be used. See Appendix C for more information on these properties.

20.5.5 Usage

The IcePack registry supports the following command-line options:

```
$ icepackregistry -h
Usage: icepackregistry [options]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
--nowarn             Don't print any security warnings.
```

The registry can optionally be run as a Unix daemon or Win32 service. On Unix, the following additional commands are supported:

```
--daemon           Run as a daemon.
--noclose           Do not close open file descriptors.
--nochdir           Do not change the current working directory.
```

On Windows, the following additional commands are supported:

```
--service NAME      Run as the Windows service NAME.

--install NAME [--display DISP] [--executable EXEC] [args]
                    Install as Windows service NAME. If DISP is
                    provided, use it as the display name,
                    otherwise NAME is used. If EXEC is provided,
                    use it as the service executable, otherwise
                    this executable is used. Any additional
                    arguments are passed unchanged to the
                    service at startup.

--uninstall NAME     Uninstall Windows service NAME.
--start NAME [args]  Start Windows service NAME. Any additional
                    arguments are passed unchanged to the
                    service.
--stop NAME          Stop Windows service NAME.
```

Please see Section 10.3.2 for more information on these options.

20.5.6 Configuration

IcePack registry configuration properties have the prefix `IcePack.Registry` and are described in Appendix C.

20.5.7 Location Service

An IcePack registry can be used by itself as a location service for applications that need indirect binding but do not require the server activation features of an IcePack node.

For example, the following configuration file defines the minimal set of properties that the registry requires to operate in this mode:

```
IcePack.Registry.Client.Endpoints=default -p 12000
IcePack.Registry.Server.Endpoints=default
IcePack.Registry.Admin.Endpoints=default
IcePack.Registry.Data=db
IcePack.Registry.DynamicRegistration=1
```

Since no IcePack nodes will be running with this registry, it is not necessary to define the `IcePack.Registry.Internal.Endpoints` property. The `IcePack.Registry.DynamicRegistration` property is required to allow servers to dynamically register their object adapters without having previously configured them via deployment.

We can initialize the registry's database directory and start it using the following commands:

```
$ mkdir db
$ icepackregistry --Ice.Config=config
```

The last command assumes the configuration file exists in the current directory with the name `config`.

Clients and servers can now use this registry by defining the default locator property using the endpoint in the registry's configuration file:

```
Ice.Default.Locator=IcePack/Locator:default -p 12000
```

A manually-activated server that is configured with this property will register each of its object adapters that has an `AdapterId` property, allowing clients to use the object adapter proxy style. Furthermore, you can use the administration tool (see Section 20.7) to advertise well-known objects in the registry and allow clients to use the object proxy style.

20.6 IcePack Node

An IcePack node is a process that activates, monitors, and deactivates registered server processes. Servers are registered with IcePack nodes via the IcePack deployment mechanism described in Section 20.8.

You can run any number of IcePack nodes in a domain, but typically there is one node per host. An IcePack node must be running on each host on which servers are activated automatically, and nodes cannot run without an IcePack registry.

20.6.1 Server Activation

On-demand server activation is a valuable feature of distributed computing architectures for a number of reasons:

- it minimizes application startup times by avoiding the need to pre-start all servers;
- it allows administrators to use their computing resources more efficiently because only those servers that are actually needed are running;
- it provides more reliability in the case of some server failure scenarios, e.g., the server is reactivated after a failure and may still be capable of providing some services to clients until the failure is resolved;
- it allows remote activation and deactivation.

Activation in Detail

Activation occurs when an Ice client requests the endpoints of one of the server's object adapters via the Ice locator facility (see Section 20.4). If the server is not active at the time the client issues the request, the node activates the server and waits for the server's object adapter to register its endpoints. Once the object adapter endpoints are registered, the node sends the endpoint information back to the client. This sequence ensures that the client receives the endpoint information *after* the server is ready to receive requests.

Requirements

In order to use on-demand activation for an object adapter, the adapter must have an identifier and be entered in the IcePack registry.

Efficiency

One of the advantages of on-demand activation listed above is the ability to manage computing resources more efficiently. Of course there are many aspects to this, but Ice makes one technique particularly simple: servers can be configured to terminate gracefully after they have been idle for a certain amount of time.

A typical scenario involves a server that is activated on demand, used for a while by one or more clients, and then terminated automatically when no requests have been made for a configurable number of seconds. All that is necessary is setting the configuration property `Ice.ServerIdleTime` to the desired idle time.

20.6.2 Endpoint Registration

Section 20.4.4 discusses the configuration requirements for enabling automatic endpoint registration in servers. It should be noted however that IcePack simplifies the configuration process in two ways:

1. A server that is activated automatically by an IcePack node does not need to explicitly configure a proxy for the locator because the IcePack node defines it on the server's command-line.
2. The IcePack deployment mechanism automates the creation of a configuration file for the server, including the definition of object adapter identifiers and endpoints (see Section 20.8.5).

20.6.3 Server Registration and Deactivation

After activating a server, the IcePack node does not normally deactivate the server unless it is explicitly requested via an administrative command. The IcePack node first attempts to deactivate the server gracefully, and then waits for the server to perform an orderly shutdown. If the server does not terminate on its own after a certain period of time, the IcePack node terminates the process.

There are two ways for an IcePack node to request an orderly shutdown of a server:

1. By invoking on the proxy of an `Ice::Process` object residing in the server.
2. Using platform-specific mechanisms, such as POSIX signals.

The latter works well on POSIX platforms but requires the server to be prepared to intercept signals (see Section 15.11). On Windows platforms, it works less reliably for C++ servers, and not at all for Java servers. For these reasons, the former

alternative is preferred because it is portable and reliable, and an IcePack node only uses the latter alternative if no `Ice: : Process` proxy is supplied.

The simplest and most-common way for a server to register an `Ice: : Process` proxy is to set the object adapter property `RegisterProcess` to a non-zero value. This setting causes a newly-activated object adapter to create a servant implementing the `Ice: : Process` interface, add that servant to the active servant map using a UUID for the identity, and invoke an operation on the Ice locator facility to register the proxy.

The object adapter also requires a value for the `Ice . ServerId` property, which should be set to the server's name. The IcePack node defines this property for automatically-activated servers.

A server should configure at most one of its object adapters with the `RegisterProcess` property. When using IcePack deployment descriptors, the `RegisterProcess` property can be defined in the server's configuration automatically by setting the `register` attribute of the `adapter` element to `true` (see Section 20.9).

The `Ice: : Process` servant responds to the IcePack node's deactivation request by invoking `shutdown` on the communicator. This presents a potential security hole: if a hostile client obtained the proxy, it could make a denial-of-service attack by shutting down the server. It is therefore important that careful consideration be given to the server's endpoints in order to minimize the risk of an attack. Specifically, an object adapter can be configured with secure endpoints, or a separate object adapter can be dedicated to the `Ice: : Process` servant and configured with an endpoint that is accessible only to the IcePack node.

20.6.4 Usage

The IcePack node supports the following command-line options:

```
$ icepacknode -h
Usage: icepacknode [options]
Options:
  -h, --help                Show this message.
  -v, --version              Display the Ice version.
  --nowarn                  Don't print any security warnings.

--deploy DESCRIPTOR [TARGET1 [TARGET2 ...]]
                           Deploy descriptor in file DESCRIPTOR, with
                           optional targets.
```

The `--deploy` option allows an application to be deployed automatically as the node process starts, which can be especially useful during testing. The name of a descriptor file should be given, and optionally the names of individual targets within the descriptor file.

The node can optionally be run as a Unix daemon or Win32 service. On Unix, the following additional commands are supported:

```
--daemon           Run as a daemon.
--noclose          Do not close open file descriptors.
--nochdir          Do not change the current working directory.
```

On Windows, the following additional commands are supported:

```
--service NAME      Run as the Windows service NAME.

--install NAME [--display DISP] [--executable EXEC] [args]
                    Install as Windows service NAME. If DISP is
                    provided, use it as the display name,
                    otherwise NAME is used. If EXEC is provided,
                    use it as the service executable, otherwise
                    this executable is used. Any additional
                    arguments are passed unchanged to the
                    service at startup.

--uninstall NAME     Uninstall Windows service NAME.
--start NAME [args]  Start Windows service NAME. Any additional
                    arguments are passed unchanged to the
                    service.
--stop NAME          Stop Windows service NAME.
```

Please see Section 10.3.2 for more information on these options.

20.6.5 Configuration

IcePack node configuration properties have the prefix `IcePack.Node` and are described in Appendix C.

20.7 IcePack Administration Tool

The IcePack administration tool is a command-line program that provides administrative control over all aspects of IcePack operation, including the deployment of applications and servers, and querying for information about registered entities.

The tool requires that the `Ice.Default.Locator` property be specified as described in Section 20.5.4.

20.7.1 Usage

The IcePack administration tool supports the following command-line options:

```
$ icepackadmin -h
Usage: icepackadmin [options] [file...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-DNAME               Define NAME as 1.
-DNAME=DEF           Define NAME as DEF.
-UNAME               Remove any definition for NAME.
-IDIR                Put DIR in the include file search path.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.
```

The `-e` option causes the tool to execute the given commands and then exit without entering an interactive session. Otherwise, the tool enters an interactive session; the **help** command displays the following usage information:

help

Print this message.

exit, quit

Exit this program.

application add DESC [TARGET1 [TARGET2 ...]]

Add servers described in application descriptor DESC. If specified the optional targets will be deployed.

application remove DESC

Remove servers described in application descriptor DESC.

node list

List all registered nodes.

node ping NAME

Ping node NAME.

node shutdown NAME

Shutdown node NAME.

server list

List all registered servers.

**server add NODE NAME DESC [PATH [LIBPATH [TARGET1
[TARGET2 ...]]]]**

Add server NAME to node NODE with deployment descriptor DESC, optional PATH and LIBPATH. If specified the optional targets will be deployed.

server remove NAME

Remove server NAME.

server describe NAME

Get server NAME description.

server state NAME

Get server NAME state.

server pid NAME

Get server NAME pid.

server start NAME

Start server NAME.

server stop NAME

Stop server NAME.

server signal NAME SIGNAL

Send SIGNAL (e.g. SIGTERM or 15) to server NAME.

server stdout NAME MESSAGE

Write MESSAGE on server NAME's stdout.

server stderr NAME MESSAGE

Write MESSAGE on server NAME's stderr.

server activation NAME [on-demand | manual]

Set the server activation mode to on-demand or manual activation.

adapter list

List all registered adapters.

adapter endpoints NAME

Get adapter NAME endpoints.

object add PROXY [TYPE]

Add an object to the object registry, optionally specifies its type.

object remove IDENTITY

Remove an object from the object registry.

object find TYPE

Find all objects with the type TYPE.

shutdown

Shut the IcePack registry down.

20.8 Deployment

Deployment, with respect to IcePack, means configuring an IcePack domain for an application. Specifically, deployment involves adding well-known objects and object adapters to the IcePack registry, configuring servers on IcePack nodes, and creating server configuration files.

IcePack provides a data-driven deployment service in which *descriptors* are written in XML and deployed as often as necessary. There are three types of deployment descriptors:

- Application descriptors define servers to be deployed, and the nodes on which to deploy them.
- Server descriptors define Ice server applications written in C++ or Java. Server descriptors can also define IceBox servers (see Chapter 25).
- Service descriptors define IceBox services.

During deployment, IcePack configures its various components as directed by the descriptors. Each descriptor must reside in its own file.

20.8.1 Application Descriptors

With respect to IcePack, an *application* is defined as a collection of servers and/or IceBox services running on one or more IcePack nodes. An application descriptor is structured similarly, as illustrated by the following example:

```
<application>
  <node name="Firewall" basedir="${basedir}">
    <server name="AppGateway" binpath="./appgateway"
      descriptor="appgateway.xml"/>
  </node>
  <node name="Backend" basedir="${basedir}">
    <server name="Account" binpath="/opt/Ice/bin/icebox"
      descriptor="account.xml"/>
  </node>
</application>
```

Inside the outer `application` element are two `node` elements defining IcePack nodes: `Firewall` and `Backend`. The server `AppGateway` will be deployed on `Firewall`, and the server `Account` will be deployed on `Backend`. Paths to the executables, as well as server descriptor files, are supplied in the attributes of each `server` element.

20.8.2 Server Descriptors

A server descriptor defines a server's object adapters, well-known objects, and configuration properties.

Continuing our example from the preceding section, the contents of `appgateway.xml` are shown below:

```
<server kind="cpp">
  <adapters>
    <adapter name="GatewayAdapter" endpoints="default">
      <object identity="${name}" type="::AppGateway"/>
    </adapter>
  </adapters>

  <properties>
    <property name="Account.Proxy" value="AccountManager"/>
  </properties>
</server>
```

This server descriptor defines a C++ server with one object adapter named `GatewayAdapter` with a default endpoint configuration. The adapter defines a well-known object of type `::AppGateway` and an identity with the value of the vari-

able `${name}`. In this example, `${name}` is substituted with `AppGateway`, which is the name of the server in the application descriptor.

The server descriptor also defines a property named `Account.Proxy` having the value `AccountManager`, representing a proxy for a well-known object that will be resolved via the Ice locator facility.

The descriptor in `account.xml` is shown below:

```
<server kind="cpp-icebox" endpoints="default">
  <service name="AccountService" descriptor="accountsvc.xml"/>
</server>
```

This server descriptor defines a C++ IceBox server with a default endpoint configuration. The server is configured with an IceBox service named `AccountService`, whose descriptor is located in the file `accountsvc.xml`.

For more information on IceBox, see Chapter 25.

20.8.3 Service Descriptors

A service descriptor defines an IceBox service, including its object adapters, well-known objects, and configuration properties.

Continuing our example from the preceding sections, here are the contents of `accountsvc.xml`:

```
<service kind="standard" entry="AccountService:create">
  <adapters>
    <adapter name="AccountAdapter" endpoints="default">
      <object identity="AccountManager"
        type="::AccountManager"/>
    </adapter>
  </adapters>
</service>
```

This service descriptor defines an IceBox service with one object adapter named `AccountAdapter` with a default endpoint configuration. The adapter defines a well-known object of type `::AccountManager` and the identity `AccountManager`. This is the well-known object referred to by the `AppGateway` server descriptor in Section 20.8.2.

20.8.4 Requirements

To deploy an application, the IcePack registry must be running, and all IcePack nodes on which servers will be deployed must be running.

20.8.5 Configuring Servers and Services

IcePack generates a configuration file for each deployed server and service. The file contains the following properties:

- AdapterId and Endpoints properties for each adapter element
- RegisterProcess property for a server's adapter, if necessary
- an equivalent definition for each property element
- an IceBox.DBEnvName.Name property for a Freeze service Name
- an IceBox.ServiceManager.Identity property for an IceBox server
- an IceBox.LoadOrder property for an IceBox server, listing the IceBox services in the order they appear in the <server> descriptor

These files, located in the IcePack node's data directory, generally should not be edited by hand, because any changes will be lost if the application is redeployed.

20.8.6 Deployment Example

To illustrate all this, let us deploy a very simple application. The application descriptor is contained in the file `application.xml`, shown below:

```
<application>
  <node name="node" basedir="${basedir}">
    <server name="SimpleServer" binpath="./server"
      descriptor="server.xml"/>
  </node>
</application>
```

Here's the server descriptor in the file `server.xml`:

```
<server kind="cpp">
  <adapters>
    <adapter name="SimpleAdapter" endpoints="default">
      <object identity="${name}" type="::Simple"/>
    </adapter>
  </adapters>
</server>
```

We can deploy this application in a few simple steps.

1. Create a configuration file called `config` for the IcePack node. The file should contain the following properties:

```
#
# The IcePack locator proxy.
#
```

```
Ice.Default.Locator=IcePack/Locator:default -p 12000

#
# IcePack registry configuration.
#
IcePack.Registry.Client.Endpoints=default -p 12000
IcePack.Registry.Server.Endpoints=default
IcePack.Registry.Internal.Endpoints=default
IcePack.Registry.Admin.Endpoints=default
IcePack.Registry.Data=db/registry

#
# IcePack node configuration.
#
IcePack.Node.Name=node
IcePack.Node.Endpoints=default
IcePack.Node.Data=db/node
IcePack.Node.CollocateRegistry=1
```

2. Create the database directory structure for the IcePack node:

```
$ mkdir db
$ mkdir db/registry
$ mkdir db/node
```

3. Start the IcePack node. Note that the node will have a collocated IcePack registry service, because the `IcePack.Node.CollocateRegistry` property is defined in the configuration file.

```
$ icepacknode --Ice.Config=config
```

4. In another window, start the administration tool and deploy the application. We assume that the application descriptor resides in the file `application.xml`. When starting **icepackadmin**, we use the same configuration file as for **icepacknode**. This is done merely for convenience; the administration tool requires only the `Ice.Default.Locator` property.

```
$ icepackadmin --Ice.Config=config
>>> application add "application.xml"
```

The application should now be deployed. Note that `application.xml` must be enclosed in quotes because `application` is a keyword in the **icepack-admin** command language.

You can verify the configuration using the following commands:

```
>>> server list
SimpleServer
```

```
>>> adapter list
IcePack.Node-node
IcePack.Registry.Internal
SimpleAdapter-SimpleServer
>>> object find ::Simple
SimpleServer -t @ SimpleAdapter-SimpleServer
```

Notice that we did not explicitly assign an identifier to the object adapter in our server descriptor, so the deployer combined the adapter name with the server name to create the identifier `SimpleAdapter-SimpleServer`.

20.8.7 Deploying with Targets

Deployment descriptors allow you to define elements called *targets* that are deployed only when explicitly requested. This is useful in a number of situations, but one of the most common is debugging. We illustrate the benefits of using targets with two examples.

Example 1

Let's modify the server descriptor of our simple deployment example in Section 20.8.6 to include a target element:

```
<server kind="cpp">
  <adapters>
    <adapter name="SimpleAdapter" endpoints="default">
      <object identity="{name}" type="::Simple"/>
    </adapter>
  </adapters>

  <target name="debug">
    <properties>
      <property name="{name}.Debug" value="1"/>
    </properties>
  </target>
</server>
```

We have added a target named `debug` containing a `properties` element which defines one property. If this descriptor is deployed without specifying a target, the contents of the target element are simply ignored. When deployed with the `debug` target, all of the other non-target elements are processed as usual, and in addition, a property is added to the server's configuration file (see Section 20.8.5). The property name is composed using a descriptor variable, as described in

Section 20.9.6. In this example, the property name evaluates to `SimpleServer.Debug`.

Using **icepackadmin**, we can deploy this application normally as follows:

```
>>> application add "application.xml"
```

Alternatively, when we need to run the application with the debugging property defined, we can deploy it with the debugging target like this:

```
>>> application add "application.xml" debug
```

Notice that we can specify the target name as `debug`, even though the application descriptor has no target with that name (we added the target to the server descriptor). An unqualified name such as `debug` will be applied to every descriptor in the application. If there were multiple servers, and we wanted to ensure that debugging was enabled in only one of them, we would need to use a fully-qualified target name:

```
>>> application add "application.xml"
      "node.SimpleServer.debug"
```

Example 2

Now let us consider a slightly more complicated example. We have expanded the target element in our server descriptor:

```
<server kind="cpp">
  <adapters>
    <adapter name="SimpleAdapter" endpoints="default">
      <object identity="{name}" type="::Simple"/>
    </adapter>
  </adapters>

  <target name="debug">
    <adapters>
      <adapter name="DebugAdapter" endpoints="default">
        <object identity="DebugController"
          type="::DebugController"/>
      </adapter>
    </adapters>
    <properties>
      <property name="{name}.Debug" value="1"/>
    </properties>
  </target>
</server>
```

In addition to the property introduced in the previous example, we have defined a new object adapter named `DebugAdapter` and registered a new object with the identity `DebugController`. The server creates these entities when it sees the `debug` property is set, providing clients with access to debugging facilities.

20.9 Descriptor Reference

This section describes each of the XML elements that comprise the IcePack deployment descriptors.

20.9.1 Application Elements

An application descriptor consists of a single `application` element containing one or more `node` elements. Each `node` element contains one or more `server` elements describing the servers to be deployed on that node.

Application

The `application` element supports the following attributes:

Table 20.1. Attributes for application element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No

Node

The `node` element supports the following attributes:

Table 20.2. Attributes for node element.

Attribute	Description	Required
<code>name</code>	The name of the node as defined by the <code>IcePack.Node.Name</code> configuration property.	Yes
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the IcePack node's current working directory is used. The directory must be accessible to the IcePack node.	No

Server

The `server` element supports the following attributes:

Table 20.3. Attributes for server element.

Attribute	Description	Required
<code>name</code>	The name of the server. Server names uniquely identify servers in the IcePack registry.	Yes
<code>descriptor</code>	The pathname of the descriptor.	Yes
<code>binpath</code>	The pathname of the server executable. This attribute is required for C++ Ice servers. For C++ IceBox servers, the value <code>icebox</code> is used if not specified. For Java Ice and IceBox servers, the value <code>java</code> is used if not specified.	Only for C++ Ice servers
<code>libpath</code>	For Java Ice and IceBox servers, this attribute specifies the Java classpath. The value of this attribute is passed directly to the Java virtual machine. This attribute is not currently used for C++ servers.	No
<code>targets</code>	The names of targets used to deploy the server, separated by white space.	No

20.9.2 Server Elements

A server descriptor consists of a single `server` element. The element may contain the following:

- `service` elements defining IceBox services
- an `adapters` element defining object adapters to be registered in the IcePack registry
- a `properties` element specifying configuration properties for the server
- a `classname` element defining the name of the class containing the Java server's main method
- a `pwd` element specifying the pathname of the server's working directory
- `option` elements providing server command-line options
- `vm-option` elements providing command-line options for the Java Virtual Machine
- `env` elements providing environment variable definitions
- `target` elements defining deployment targets.

Server

The `server` element supports the following attributes:

Table 20.4. Attributes for server element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No
<code>kind</code>	The type of server described by this descriptor. Possible values are: <code>cpp</code> , <code>java</code> , <code>cpp-icebox</code> and <code>java-icebox</code> .	Yes
<code>endpoints</code>	For C++ or Java IceBox servers, this attribute defines the endpoints for the IceBox service manager interface and is equivalent to the <code>IceBox.ServiceManager.Endpoints</code> configuration property.	Only for IceBox servers

Service

The `service` element is only valid in the scope of a `server` element if the server's `kind` attribute is `java-icebox` or `cpp-icebox`. The following attributes are supported:

Table 20.5. Attributes for service element.

Attribute	Description	Required
<code>name</code>	The name of the service. Service names uniquely identify services in an IceBox server.	Yes
<code>descriptor</code>	The pathname of the descriptor.	Yes
<code>targets</code>	The names of targets used to deploy the service, separated by white space.	No

Classname

The `classname` element is only valid in the scope of a `server` element if the server's `kind` attribute is `java`. The element value specifies the name of the class containing the server's main method. The element has no attributes.

Env

The `env` element defines an environment variable to be present in the server's environment upon activation. The element value should be of the form *name=value*. The element has no attributes.

Pwd

The `pwd` element value specifies the pathname of the server's working directory. The element has no attributes.

Option

The `option` element value defines a command-line option that is passed to the server upon activation. The element has no attributes.

Vm-option

The `vm-option` element is only valid in the scope of a `server` element if the server's `kind` attribute is `java` or `java-icebox`. The element value defines a

command-line option for the Java Virtual Machine, therefore it appears on the JVM command line prior to the server classname. The element has no attributes.

20.9.3 Service Elements

A service descriptor defines an IceBox service and consists of a single `service` element. The element may contain an `adapters` element defining object adapters to be registered in the IcePack registry, a `properties` element specifying configuration properties for the service, or `target` elements defining deployment targets.

Service

The `service` element supports the following attributes:

Table 20.6. Attributes for service element.

Attribute	Description	Required
<code>basedir</code>	The base directory from which all pathnames in this element are resolved. If not specified, the directory in which the descriptor file resides is used.	No
<code>kind</code>	The type of IceBox service defined by this descriptor. Possible values are <code>standard</code> or <code>freeze</code> .	Yes
<code>entry</code>	The service entry point. In C++, the entry point has the form <i>library:symbol</i> , where <i>library</i> is the name of a shared library or DLL, and <i>symbol</i> is the name of a factory function used to create the service. In Java, the entry point is the name of the service implementation class.	Yes
<code>dbenv</code>	If the value of the <code>kind</code> attribute is <code>freeze</code> , this attribute defines the service database environment directory. If not specified, the node provides a default database environment directory.	No

20.9.4 Common Elements

Deployment descriptors support a number of common elements.

Target

A `target` element encloses other elements that will be deployed only if the user specifies the target's name during deployment. A `target` element can be used in application, server, and service descriptors. Furthermore, a `target` element can only contain those elements that may legally appear in its enclosing descriptor. For example, a `node` element is legal inside a `target` element of an application descriptor, but not inside a `target` element of a server descriptor. The following attributes are supported:

Table 20.7. Attributes for target element.

Attribute	Description	Required
name	The target name.	Yes

Adapters

The `adapters` element contains one or more `adapter` elements and can be used in server and service descriptors. The element has no attributes.

Adapter

An `adapter` element instructs the deployer to register the adapter with the IcePack registry, which associates the adapter with its server and thereby enables

on-demand activation. The `adapter` element may only be used within an `adapters` element. The following attributes are supported:

Table 20.8. Attributes for adapter element.

Attribute	Description	Required
<code>name</code>	The object adapter name. This is the same name that an application passes to the <code>createObject - Adapter</code> operation.	Yes
<code>id</code>	The object adapter identifier. This identifier is used by the IcePack registry to uniquely identify an object adapter. If not specified, the identifier is created based on the adapter name and server name (and service name if the adapter is defined in a service descriptor).	No
<code>endpoints</code>	Endpoints for the object adapter.	Yes
<code>register</code>	If true, the <code>RegisterProcess</code> property is defined for this adapter. Only one adapter should be registered per server. See Section 20.6.3 for more information. If not defined, the default value is false.	No

Object

An `object` element instructs the deployer to register the object with the IcePack registry, to allow clients to lookup objects by identity or type. The `object` element may only be used in an `adapter` element. The following attributes are supported:

Table 20.9. Attributes for object element.

Attribute	Description	Required
<code>identity</code>	A stringified object identity.	Yes
<code>type</code>	The fully-scoped Slice type of the object, such as <code>::Hello</code> .	No

Properties

The `properties` element contains one or more `property` elements and can be used in server and service descriptors. The element has no attributes.

Property

Every deployed server or service has a configuration file located in the data directory of the IcePack node, and the `property` element instructs the deployer to add a property to this configuration file. The `property` element may only be used within a `properties` element. The following attributes are supported:

Table 20.10. Attributes for property element.

Attribute	Description	Required
<code>name</code>	The property name.	Yes
<code>value</code>	The property value.	If <code>location</code> is not specified
<code>location</code>	The property value interpreted as a pathname. A relative pathname is relative to the descriptor base directory.	If <code>value</code> is not specified

The `value` and `location` attributes are mutually exclusive.

20.9.5 Pathname Semantics

Relative pathnames in deployment descriptors are resolved using the base directory of the enclosing element. The default base directory is the directory in which the descriptor file resides, unless overridden by a `basedir` attribute as shown in the application descriptor in Section 20.8.1.

It is important to understand how these pathnames are used during deployment:

1. The **icepackadmin** tool (described in Section 20.7) sends the application descriptor pathname to the IcePack registry, which reads the file. The file must therefore be accessible to the registry, and relative pathnames will be resolved relative to the current working directory of the registry process.
2. The IcePack registry processes each node element by contacting the designated IcePack node to deploy its servers.
3. The IcePack node reads the server descriptor files, as well as any IceBox service descriptor files, therefore the pathnames of server and service descriptors must be accessible to the IcePack node on which they will be deployed,

and relative pathnames will be resolved relative to the current working directory of the node process. Furthermore, the pathnames for binaries and libraries must also be accessible to the IcePack node.

20.9.6 Descriptor Variables

A number of variables can be used in deployment descriptors. Variables have the form $\${name}$, where *name* is the variable name. You can use variables to compose an attribute value, as shown below:

```
<object id="Factory- $\${name}$ -Object" type=""/>
```

During deployment, $\${name}$ will be substituted with the descriptor name of the server or service.

These are the variables you can use in IcePack descriptors:

- `basedir`

The base directory of the descriptor. If not overridden by the top-level element `basedir` attribute, the base directory is the directory in which the descriptor resides. See Section 20.9.5 for more information on descriptor pathnames.

- `name`

The component name. In a server descriptor, this variable is set to the server name defined in the application descriptor. For an IceBox service, this variable is set to the service name defined in the IceBox server descriptor. This variable cannot be used in an application descriptor.

- `parent`

The name of the parent component. In a server descriptor, this variable is substituted with the name of the IcePack node where the server is deployed. In a service descriptor, this variable is substituted by the name of the IceBox server where the service is deployed. This variable cannot be used in an application descriptor.

- `fqn`

The fully qualified name of the component. In a server descriptor, this variable has the form *node_name.server_name*, where *node_name* is the name of the node where the server is deployed and *server_name* is the name of the server. In a service descriptor, this variable has the form *node_name.server_name.service_name*. This variable cannot be used in an application descriptor.

20.10 Troubleshooting

This section provides suggestions on how to resolve several common IcePack issues.

20.10.1 Activation Failure

Automatic server activation can fail for a number of reasons, but the most likely cause is incorrect configuration. For example, an IcePack node may fail to activate a server because the server's executable file or associated shared libraries could not be found. There are several steps you can take in this case:

1. Enable activation tracing in the IcePack node by setting the configuration property `IcePack.Node.Trace.Activation=3`.
2. Examine the tracing output and verify the server's executable pathname and working directory are correct.
3. If the executable pathname is correct, and it is a relative pathname, then it may not be correct relative to the IcePack node's current working directory (see Section 20.9.5). Either replace the relative pathname with an absolute pathname, or restart the IcePack node in the proper working directory.
4. Verify that the IcePack node is started with the correct `PATH` or `LD_LIBRARY_PATH` settings for the server's shared libraries.

Another cause of activation failure is a server fault during startup. After you have confirmed that the IcePack node is successfully spawning the server process using the steps above, you should then check for signs of a server fault (e.g., on UNIX, look for a `core` file in the IcePack node's current working directory). See Section 20.10.4 for more information on server failures.

20.10.2 Proxy Failure

A client may receive `Ice::NotRegisteredException` if binding fails for an indirect proxy (see Section 20.3.2). This exception indicates that the proxy's object identity or object adapter is not registered with the Ice locator facility. The following steps may help you discover the cause of the exception:

1. Use **icepackadmin** (see Section 20.7) to verify that the object identity or object adapter identifier is actually registered, and that it matches what is used by the proxy:

```
>>> adapter list
```



```
...  
>>> object find ::Hello  
...
```

2. If the problem persists, review your configuration to ensure that the locator proxy used by the client matches the IcePack registry's client endpoints, and that those endpoints are accessible to the client (i.e., are not blocked by a firewall).
3. Finally, enable locator tracing in the client by setting the configuration property `Ice.Trace.Locator=1`, then run the client again to see if any log messages are emitted which may indicate the problem.

20.10.3 Deployment Failure

Pathnames in descriptors, especially relative pathnames, are a common source of deployment problems. When writing descriptor files, you should be familiar with the deployment process explained in Section 20.9.5. In short, server and service descriptor files are read by the IcePack node on which they are deployed, therefore the pathnames of these descriptors must be specified such that the IcePack node can access them. In practice, this means the descriptor files for a node's servers and services either must be copied to the host on which the node is running, or must be accessible via a network file system. Furthermore, the use of relative pathnames requires knowledge of the IcePack node's current working directory and therefore should be used with care.

20.10.4 Server Failure

Diagnosing a server failure can be difficult, especially when servers are activated automatically on remote hosts. Here are a few suggestions:

1. If the server is running on a UNIX host, check the current working directory of the IcePack node process for signs of a server failure, such as a `core` file.
2. Judicious use of tracing can help to narrow the search. For example, if the failure occurs as a result of an operation invocation, enable protocol tracing in the Ice run time by setting the configuration property `Ice.Trace.Protocol=1` to discover the object identity and operation name of all requests.

Of course, the default log output channels (standard out and standard error) will probably be lost if the server is activated automatically, so either start the

server manually (see below) or redirect the log output (see Appendix C for a description of the `Ice.UseSyslog` property).

You can also use the `Ice: : Logger` interface to emit your own trace messages.

3. Run the server in a debugger; a server configured for automatic activation can also be started manually if necessary. However, since the IcePack node did not activate the server, it cannot monitor the server process and therefore will not know when the server terminates. This will prevent subsequent activation unless you clean up the IcePack state when you have finished debugging and terminated the server. You can do this by starting the server using `icepack-admin` (see Section 20.7):

```
>>> server start TheServer
```

This will cause the IcePack node to activate (and therefore monitor) the server process. If you do not want to leave the server running, you can stop it with the equivalent `icepackadmin` command.

4. After the server is activated as is in a quiescent state, attach your debugger to the running server process. This avoids the issues associated with starting the server manually (as described in the previous step), but does not provide as much flexibility in customizing the server's startup environment.

20.11 Summary

This chapter provided a detailed discussion of IcePack, including the modifications required to incorporate IcePack into client and server applications, as well as the configuration and administration of IcePack components. Once you understand the basic concepts on which IcePack is founded, you quickly begin to appreciate the flexibility, power, and convenience of IcePack's capabilities:

- automatic server activation increases reliability and makes more efficient use of processing resources
- the locator facility simplifies administrative requirements and minimizes coupling between clients and servers
- the deployment mechanism reduces mundane configuration tasks into a set of easily-understood, reusable XML files

In short, IcePack provides the tools you need to develop robust, enterprise-class Ice applications.

Chapter 21

Freeze

21.1 Chapter Overview

This chapter describes how to use Freeze to add persistence to Ice applications. Section 21.3 discusses the Freeze map and shows how to use it in a simple example. Section 21.4 examines an implementation of the file system using a Freeze map. Section 21.5 presents the Freeze evictor, and Section 21.6 demonstrates the Freeze evictor in another file system implementation.

21.2 Introduction

Freeze represents a set of persistence services, as shown in Figure 21.1.

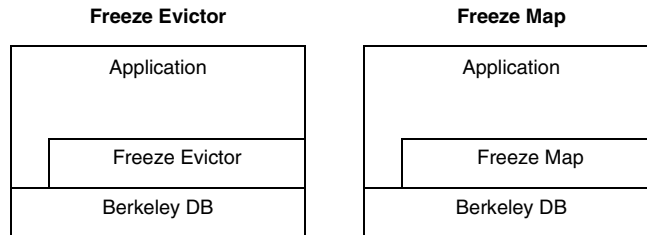


Figure 21.1. Layer diagram for Freeze persistence services.

The Freeze persistence services are described below:

- The Freeze evictor is a highly-scalable implementation of an Ice servant locator that provides automatic persistence and eviction of Ice objects with only minimal application code.
- The Freeze map is a generic associative container. Code generators are provided which produce type-specific maps for any Slice key and value types. Applications interact with a Freeze map just like any other associative container, except the keys and values of a Freeze map are persistent.

As you will see from the examples in this chapter, integrating a Freeze map or evictor into your Ice application is quite straightforward: once you define your persistent data in Slice, Freeze manages the mundane details of persistence.

Freeze is implemented using Berkeley DB, a compact and high-performance embedded database. The Freeze map and evictor APIs insulate applications from the Berkeley DB API, but do not prevent applications from interacting directly with Berkeley DB if necessary.

21.3 The Freeze Map

A Freeze map is a persistent, associative container in which the key and value types can be any primitive or user-defined Slice types. For each pair of key and value types, the developer uses a code-generation tool to produce a language-specific class that conforms to the standard conventions for maps in that language.

For example, in C++ the generated class resembles an STL map, and in Java it implements the `java.util.Map` interface. Most of the logic for storing and retrieving state to and from the database is implemented in a Freeze base class. The generated map classes derive from this base class, so they contain little code and therefore are efficient in terms of code size.

You can only store data types that are defined in Slice in a Freeze map. Types without a Slice definition (that is, arbitrary C++ or Java types) cannot be stored because a Freeze map reuses the Ice-generated marshalling code to create the persistent representation of the data in the database. This is especially important to remember when defining a Slice class whose instances will be stored in a Freeze map; only the “public” (Slice-defined) data members will be stored, not the private state members of any derived implementation class.

21.3.1 Freeze Connections

In order to create a Freeze Map object, you first need to obtain a Freeze Connection object by connecting to a database environment.

As illustrated in Figure 21.2, a Freeze Map is associated with a single connection and a single database file. Connection and map objects are “single threaded”: if you want to use a connection or any of its associated maps from multiple threads, you must serialize access to them. If your application requires concurrent access to the same database file (persistent Map), you must create several connections and associated maps.

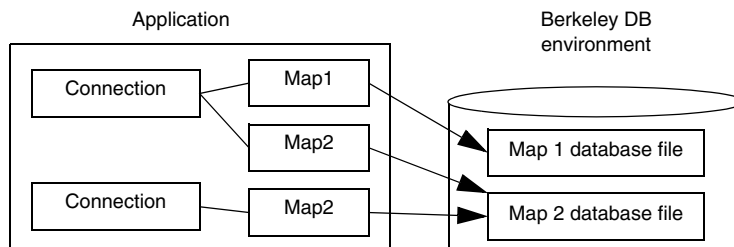


Figure 21.2. Freeze Connections and Maps.

21.3.2 Transactions

You may optionally use transactions with Freeze maps. Freeze transactions provide the usual ACID (atomicity, concurrency, isolation, durability) properties. For example, a transaction allows you to group several database updates in one atomic unit: either all or none of the updates within the transaction occur.

A transaction is started using the `beginTransaction` operation on the `Connection` object. Once a connection has an associated transaction, all operations on the map objects associated with this connection use this transaction. Eventually, you end the transaction by calling `commit` or `rollback`: `commit` saves all your updates while `rollback` undoes them.

```
module Freeze {

  local interface Transaction {
    void commit();
    void rollback();
  };

  local interface Connection {
    Transaction beginTransaction();
    nonmutating Transaction currentTransaction();
    // ...
  };
};
```

If you do not use transactions, every non-iterator update is enclosed in its own internal transaction, and every read-write iterator has an associated internal transaction which is committed when the iterator is closed.

21.3.3 Iterators

Iterators allow you to traverse the contents of a Freeze map. Iterators are implemented using Berkeley DB cursors and acquire locks on the underlying database page files. In C++, both read-only (`const_iterator`) and read-write iterators (`iterator`) are available; in Java only read-write iterators are supported.

Locks held by an iterator are released when the iterator is closed (if you do not use transactions) or when the enclosing transaction is ended. Releasing locks held by iterators is very important to let other threads access the database file through other connection and map objects. Occasionally it is even necessary to release locks to avoid self-deadlock (waiting forever for a lock held by an iterator created by the same thread).

To improve ease of use and make self-deadlocks less likely, Freeze often closes iterators automatically. When you start or when you end a transaction, Freeze closes all the iterators associated with the corresponding maps. If you do not use transactions, any write operation on a map (such as inserting a new element) automatically closes all iterators opened on the same map object, except for the current iterator when the write operation is performed through that iterator. In Java, iterators are also closed when the enclosing map or connection is closed by the application.

There is, however, one situation where an explicit iterator close is needed to avoid self-deadlock:

- you do not use transactions, and
- you have an opened iterator that was used to update a map (it holds a write lock), and
- in the same thread, you read that map.

Read operations never close iterators automatically. In that situation, you need to either use transactions or explicitly close the iterator that holds the write lock.

21.3.4 Recovering from Deadlock Exceptions

If you use multiple threads to access a database file, Berkeley DB may acquire locks in conflicting orders (on behalf of different transactions or iterators). For example, an iterator could have a read-lock on page P1 and attempt to acquire a write-lock on page P2, while another iterator (on a different map object associated with the same database file) could have a read-lock on P2 and attempt to acquire a write-lock on P1.

When this occurs, Berkeley DB detects a deadlock and resolves it by returning a “deadlock” error to one or more threads. For all non-iterator operations performed outside any transaction, such as an insertion into a map, Freeze catches such errors and automatically retries the operation until it succeeds. For other operations, Freeze reports this deadlock by raising `Freeze::DeadlockException`. In that case, the associated transaction or iterator is also automatically rolled back or closed. A properly written application is expected to catch deadlock exceptions and retry the transaction or iteration.

21.3.5 Using a Simple Map in C++

As an example, the following command generates a simple map:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

The **slice2freeze** compiler creates C++ classes for Freeze maps. The command above directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. The final argument is the base name for the output files, to which the compiler appends the `.h` and `.cpp` suffixes. Therefore, this command produces two C++ source files: `StringIntMap.h` and `StringIntMap.cpp`.

Here is a simple program that demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit read or write operations called by the program; instead, simply using the map has the side effect of accessing the database.

```
#include <Freeze/Freeze.h>
#include <StringIntMap.h>

int
main(int argc, char* argv[])
{
    // Initialize the Communicator.
    //
    Ice::CommunicatorPtr communicator =
        Ice::initialize(argc, argv);

    // Create a Freeze database connection.
    //
    Freeze::ConnectionPtr connection =
        Freeze::createConnection(communicator, "db");

    // Instantiate the map.
    //
    StringIntMap map(connection, "simple");

    // Clear the map.
    //
    map.clear();

    Ice::Int i;
    StringIntMap::iterator p;

    // Populate the map.
    //
    for (i = 0; i < 26; i++) {
        std::string key(1, 'a' + i);
        map.insert(make_pair(key, i));
    }
}
```



```

    }

    // Iterate over the map and change the values.
    //
    for (p = map.begin(); p != map.end(); ++p)
        p.set(p->second + 1);

    // Find and erase the last element.
    //
    p = map.find("z");
    assert(p != map.end());
    map.erase(p);

    // Clean up.
    //
    connection->close();
    communicator->destroy();

    return 0;
}

```

Prior to instantiating a Freeze map, the application must connect to a Berkeley DB database environment.

```

Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator, "db");

```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, which by default is created if it does not exist. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

```
StringIntMap map(connection, "simple");
```

Next, we clear the map. This ensures we have an empty map (and therefore an empty database) in case the program is run more than once.

```
map.clear();
```

We populate the map, using a single-character string as the key. The Freeze map supports several `insert` methods for adding entries, similar to an STL map. Insertion via `operator[]` is not supported.

```

for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}

```

Iterating over the map will look familiar to STL users. However, to modify a value at the iterator's current position, you must use the nonstandard `set` method:

```

for (p = map.begin(); p != map.end(); ++p)
    p.set(p->second + 1);

```

Next, the program obtains an iterator positioned at the element with key `z`, and erases it.

```

p = map.find("z");
assert(p != map.end());
map.erase(p);

```

Finally, the program closes the database connection, destroys its communicator and terminates.

```

connection->close();

```

It is not necessary to explicitly close the database connection, but we demonstrate it here for the sake of completeness.

21.3.6 Using a Simple Map in Java

As an example, the following command generates a simple map:

```
$ slice2freezej --dict StringIntMap,string,int
```

The `slice2freezej` compiler creates Java classes for Freeze maps. The command above directs the compiler to create a map named `StringIntMap`, with the Slice key type `string` and the Slice value type `int`. This command produces one Java source file: `StringIntMap.java`.

Here is a simple program that demonstrates how to use a `StringIntMap` to store `<string, int>` pairs in a database. You will notice that there are no explicit read or write operations called by the program; instead, simply using the map has the side effect of accessing the database.

```

public class Client
{
    public static void
    main(String[] args)
    {
        // Initialize the Communicator.
    }
}

```

```
//
Ice.Communicator communicator = Ice.Util.initialize(args);

// Create a Freeze database connection.
//
Freeze.Connection connection =
    Freeze.Util.createConnection(communicator, "db");

// Instantiate the map.
//
StringIntMap map =
    new StringIntMap(connection, "simple", true);

// Clear the map.
//
map.clear();

int i;
java.util.Iterator p;

// Populate the map.
//
for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), new Integer(i));
}

// Iterate over the map and change the values.
//
p = map.entrySet().iterator();
while (p.hasNext()) {
    java.util.Map.Entry e = (java.util.Map.Entry)p.next();
    Integer in = (Integer)e.getValue();
    e.setValue(new Integer(in.intValue() + 1));
}

// Find and erase the last element.
//
boolean b;
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);

// Clean up.
//
```

```

        map.close();
        connection.close();
        communicator.destroy();

        System.exit(0);
    }
}

```

Prior to instantiating a Freeze map, the application must connect to the Berkeley DB database environment.

```

Freeze.Connection connection =
    Freeze.Util.createConnection(communicator, "db");

```

The second argument is the name of a Berkeley DB database environment; by default, this is also the file system directory in which Berkeley DB creates all database and administrative files.

Next, the code instantiates the `StringIntMap` on the connection. The constructor's second argument supplies the name of the database file, and the third argument is a flag indicating whether the database should be created if it does not exist. Note that a database can only contain the persistent state of one map type. Any attempt to instantiate maps of different types on the same database results in undefined behavior.

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

Next, we clear the map. This ensures we have an empty map (and therefore an empty database) in case the program is run more than once.

```
map.clear();
```

We populate the map, using a single-character string as the key. As with `java.util.Map`, the key and value types must be Java objects.

```

for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), new Integer(i));
}

```

Iterating over the map is no different from iterating over any other map that implements the `java.util.Map` interface:

```
p = map.entrySet().iterator();
while (p.hasNext()) {
    java.util.Map.Entry e =
        (java.util.Map.Entry)p.next();
    Integer in = (Integer)e.getValue();
    e.setValue(new Integer(in.intValue() + 1));
}
```

Next, the program verifies that an element exists with key `z`, and then removes it. Note that the program uses a non-standard method for removing the element. The `fastRemove` method differs from the standard `remove` method in that it does not return the value associated with the removed element; instead it returns a boolean indicating whether the element was found. By eliminating the need to return the element value, the method avoids the overhead of reading the value from the database and decoding it. The standard `remove` method could also be used here; we chose to use `fastRemove` instead simply to demonstrate its use:

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

Finally, the program closes the map and its connection.

```
map.close();
connection.close();
```

21.4 Using a Freeze Map in the File System Server

We can use a Freeze map to add persistence to the file system server, and we present C++ and Java implementations in this section. However, as you will see in Section 21.5, a Freeze evictor is often a better choice for applications (such as the file system server) in which the persistent value is an Ice object.

In general, incorporating a Freeze map into your application requires the following steps:

1. Evaluate your existing Slice definitions for suitable key and value types.
2. If no suitable key or value types are found, define new (possibly derived) types that capture your persistent state requirements. Consider placing these definitions in a separate file: these types are only used by the server for persistence, and therefore do not need to appear in the “public” definitions required by

clients. Also consider placing your persistent types in a separate module to avoid name clashes.

3. Generate a Freeze map for your persistent types using the Freeze compiler.
4. Use the Freeze map in your operation implementations.

21.4.1 Choosing Key and Value Types

Our goal is to implement the file system using a Freeze map for all persistent storage, including files and their contents. Our first step is to select the Slice types we will use for the key and value types of our map. We will keep the same basic design as in XREF, therefore we need a suitable representation for persistent files and directories, as well as a unique identifier for use as a key.

Conveniently enough, Ice objects already have a unique identifier of type `Ice::Identity`, and this will do fine as the key type for our map.

Unfortunately, the selection of a value type is more complicated. Looking over the `Filesystem` module in XREF, we do not find any types that capture all of our persistent state, so we need to extend the module with some new types:

```
module Filesystem {
    class PersistentNode {
        string name;
    };

    class PersistentFile extends PersistentNode {
        Lines text;
    };

    class PersistentDirectory extends PersistentNode {
        NodeDict nodes;
    };
};
```

Our Freeze map will therefore map from `Ice::Identity` to `PersistentNode`, where the values are actually instances of the derived classes `PersistentFile` or `PersistentDirectory`. If we had followed the advice at the beginning of Section 21.4, we would have defined `File` and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

21.4.2 Implementing the File System Server in C++

In this section we present a C++ file system implementation that utilizes a Freeze map for persistent storage. The implementation is based on the one discussed in XREF, and, in this section, we only discuss code that illustrates use of the Freeze map.

Generating the Map

Now that we have selected our key and value types, we can generate the map as follows:

```
$ slice2freeze -I$(ICE_HOME)/slice --dict \
    IdentityNodeMap,Ice::Identity,Filesystem::PersistentNode\
    IdentityNodeMap Filesystem.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map class is named `IdentityNodeMap`.

The Server main Program

The server's main program is responsible for initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice::Application`, as described in Section 10.3.1. Our server main program has now become the following:

```
#include <FilesystemI.h>
#include <Ice/Application.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : public virtual Ice::Application {
public:
    FilesystemApp(const string & envName) :
        _envName(envName) { }

    virtual int run(int, char * []) {
        // Terminate cleanly on receipt of a signal
        //
        shutdownOnInterrupt();

        // Install object factories
        //
        communicator()->addObjectFactory(
```

```

        PersistentFile::ice_factory(),
        PersistentFile::ice_staticId());

communicator()->addObjectFactory(
    PersistentDirectory::ice_factory(),
    PersistentDirectory::ice_staticId());

// Create an object adapter (stored in the NodeI::_adapter
// static member)
//
NodeI::_adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

//
// Set static members used to create connections and maps
//
NodeI::_communicator = communicator();
NodeI::_envName = _envName;
NodeI::_dbName = "mapfs";

// Find the persistent node for the root directory, or
// create it if not found
//
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator(), _envName);
IdentityNodeMap persistentMap(connection, NodeI::_dbName);

Ice::Identity rootId = Ice::stringToIdentity("RootDir");
PersistentDirectoryPtr pRoot;
{
    IdentityNodeMap::iterator p =
        persistentMap.find(rootId);

    if (p != persistentMap.end()) {
        pRoot =
            PersistentDirectoryPtr::dynamicCast(
                p->second);
        assert(pRoot);
    } else {
        pRoot = new PersistentDirectory;
        pRoot->name = "/";
        persistentMap.insert(
            IdentityNodeMap::value_type(rootId, pRoot));
    }
}

```



```

        // Create the root directory (with name "/" and no parent)
        //
        DirectoryIPtr root = new DirectoryI(rootId, pRoot, 0);

        // Ready to accept requests now
        //
        NodeI::_adapter->activate();

        // Wait until we are done
        //
        communicator()->waitForShutdown();
        if (interrupted()) {
            cerr << appName()
                << ": received signal, shutting down" << endl;
        }

        return 0;
    }

private:
    string _envName;

};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}

```

Let us examine the changes in detail. First, we are now including `Freeze/Freeze.h` instead of `Ice/Ice.h`. This `Freeze` header file includes all of the other `Freeze` (and `Ice`) header files this source file requires.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:

```

FilesystemApp(const string & envName) :
    _envName(envName) { }

```

The string argument represents the name of the database environment, and is saved for later use in `run`.

One of the first tasks `run` performs is installing the `Ice` object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not

exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. Since these Slice classes have no operations, we can use their built-in factories.

```
communicator()->addObjectFactory(
    PersistentFile::ice_factory(),
    PersistentFile::ice_staticId());

communicator()->addObjectFactory(
    PersistentDirectory::ice_factory(),
    PersistentDirectory::ice_staticId());
```

Next, we set all the NodeI static members.

```
NodeI::_adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

NodeI::_communicator = communicator();
NodeI::_envName = _envName;
NodeI::_dbName = "mapfs";
```

Then we create a Freeze connection and a Freeze map. When the last connection to a Berkeley DB environment is closed, Freeze automatically closes this environment, so keeping a connection in the main function ensures the underlying Berkeley DB environment remains open. Likewise, we keep a map in the main function to keep the underlying Berkeley DB database open.

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator(), _envName);
IdentityNodeMap persistentMap(connection, NodeI::_dbName);
```

Now we need to initialize the root directory node. We first query the map for the identity of the root directory node; if no match is found, we create a new PersistentDirectory instance and insert it into the map. We use a scope to close the iterator after use; otherwise, this iterator could keep locks and prevent subsequent access to the map through another connection.

```
Ice::Identity rootId = Ice::stringToIdentity("RootDir");
PersistentDirectoryPtr pRoot;
{
    IdentityNodeMap::iterator p =
        persistentMap.find(rootId);
```

```

        if (p != persistentMap.end()) {
            pRoot =
                PersistentDirectoryPtr::dynamicCast(
                    p->second);
            assert(pRoot);
        } else {
            pRoot = new PersistentDirectory;
            pRoot->name = "/";
            persistentMap.insert(
                IdentityNodeMap::value_type(rootId, pRoot));
        }
    }
}

```

Finally, the main function instantiates the `FilesystemApp`, passing `db` as the name of the database environment.

```

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}

```

The Servant Class Definitions

We also must change the servant classes to incorporate the Freeze map. We are maintaining the multiple-inheritance design from XREF, but we have added some methods and changed the constructor arguments and state members.

Let us examine the definition of `NodeI` first. You will notice the addition of the `getPersistentNode` method, which allows `NodeI` to gain access to the persistent node in order to implement the `Node` operations. Another alternative would have been to add a `PersistentNodePtr` member to `NodeI`, but that would have forced the `FileI` and `DirectoryI` classes to downcast this member to the appropriate subclass.

```

namespace Filesystem {
    class NodeI : virtual public Node {
    public:
        // ... Ice operations ...
        static Ice::ObjectAdapterPtr _adapter;
        static Ice::CommunicatorPtr _communicator;
        static std::string _envName;
        static std::string _dbName;
    protected:
        NodeI(const Ice::Identity &, const DirectoryIPtr &);
        virtual PersistentNodePtr getPersistentNode() const = 0;
    };
}

```

```

        PersistentNodePtr find(const Ice::Identity &) const;
        IdentityNodeMap _map;
        DirectoryIPtr _parent;
        IceUtil::RecMutex _nodeMutex;
        bool _destroyed;
    public:
        const Ice::Identity _ID;
    };
}

```

Other changes of interest in NodeI are the addition of the members `_map` and `_destroyed`, and we have changed the NodeI constructor to accept an `Ice::Identity` argument.

The FileI class now has a single state member of type `PersistentFilePtr`, representing the persistent state of this file. Its constructor has also changed to accept `Ice::Identity` and `PersistentFilePtr`.

```

namespace Filesystem {
    class FileI : virtual public File,
                  virtual public NodeI {
    public:
        // ... Ice operations ...
        FileI(const Ice::Identity &, const PersistentFilePtr &,
              const DirectoryIPtr &);
    protected:
        virtual PersistentNodePtr getPersistentNode() const;
    private:
        PersistentFilePtr _file;
    };
}

```

The DirectoryI class has undergone a similar transformation.

```

namespace Filesystem {
    class DirectoryI : virtual public Directory,
                      virtual public NodeI {
    public:
        // ... Ice operations ...
        DirectoryI(const Ice::Identity &,
                  const PersistentDirectoryPtr&,
                  const DirectoryIPtr &);

        // ...
    protected:
        virtual PersistentNodePtr getPersistentNode() const;
        // ...
    };
}

```

```

private:
    // ...
    PersistentDirectoryPtr _dir;
};
}

```

Implementing FileI

Let us examine how the implementations have changed. The `FileI` methods are still fairly trivial, but there are a few aspects that need discussion.

First, each operation now checks the `_destroyed` member and raises `Ice::ObjectNotExistException` if the member is `true`. This is necessary in order to ensure that the Freeze map is kept in a consistent state. For example, if we allowed the `write` operation to proceed after the file node had been destroyed, then we would have mistakenly added an entry back into the Freeze map for that file. Previous file system implementations ignored this issue because it was relatively harmless, but that is no longer true in this version.

Next, notice that the `write` operation calls `put` on the map after changing the `text` member of its `PersistentFile` object. Although the file's `PersistentFilePtr` member points to a value in the Freeze map, changing that value has no effect on the persistent state of the map until the value is reinserted into the map, thus overwriting the previous value.

Finally, the constructor now accepts an `Ice::Identity`. This differs from previous implementations in that the identity used to be created by the `NodeI` constructor. However, as we will see later, the caller needs to determine the identity prior to invoking the subclass constructors. Similarly, the constructor is not responsible for creating a `PersistentFile` object, but rather is given one. This accommodates our two use cases: creating a new file, and restoring an existing file from the map.

```

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::RWRecMutex::RLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    return _file->text;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,

```

```

        const Ice::Current &)
    {
        IceUtil::RWRecMutex::WLock lock(_nodeMutex);

        if (_destroyed)
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);

        _file->text = text;
        _map.put(IdentityNodeMap::value_type(_ID, _file));
    }

    Filesystem::FileI::FileI(const Ice::Identity & id,
                             const PersistentFilePtr & file,
                             const DirectoryIPtr & parent) :
        NodeI(id, parent), _file(file)
    {
    }

    Filesystem::PersistentNodePtr
    Filesystem::FileI::getPersistentNode() const
    {
        return _file;
    }

```

Implementing DirectoryI

The DirectoryI implementation requires more substantial changes. We begin our discussion with the createDirectory operation.

```

    Filesystem::DirectoryI::createDirectory(
        const std::string & name,
        const Ice::Current & current)
    {
        IceUtil::RWRecMutex::WLock lock(_nodeMutex);

        if (_destroyed)
            throw Ice::ObjectNotExistException(__FILE__, __LINE__);

        checkName(name);

        PersistentDirectoryPtr persistentDir
            = new PersistentDirectory;
        persistentDir->name = name;
        DirectoryIPtr dir = new DirectoryI(
            Ice::stringToIdentity(IceUtil::generateUUID()),
            persistentDir, this);
    }

```

```

    assert(find(dir->_ID) == 0);
    _map.put(make_pair(dir->_ID, persistentDir));

    DirectoryPrx proxy = DirectoryPrx::uncheckedCast(
        current.adapter->createProxy(dir->_ID));

    NodeDesc nd;
    nd.name = name;
    nd.type = DirType;
    nd.proxy = proxy;
    _dir->nodes[name] = nd;
    _map.put(IdentityNodeMap::value_type(_ID, _dir));

    return proxy;
}

```

After validating the node name, the operation creates a `PersistentDirectory`¹ object for the child directory, which is passed to the `DirectoryI` constructor along with a unique identity. Next, we store the child's `PersistentDirectory` object in the Freeze map. Finally, we initialize a new `NodeDesc` value and insert it into the parent's node table and then reinsert the parent's `PersistentDirectory` object into the Freeze map.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```

Filesystem::FilePrx
Filesystem::DirectoryI::createFile(const std::string & name,
                                   const Ice::Current & current)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    checkName(name);

    PersistentFilePtr persistentFile = new PersistentFile;
    persistentFile->name = name;
    FileIPtr file = new FileI(
        Ice::stringToIdentity(IceUtil::generateUUID()),

```

1. Since this Slice class has no operations, the compiler generates a concrete class that an application can instantiate.

```

        persistentFile, this);
assert(find(file->_ID) == 0);
_map.put(make_pair(file->_ID, persistentFile));

FilePrx proxy = FilePrx::uncheckedCast(
    current.adapter->createProxy(file->_ID));

NodeDesc nd;
nd.name = name;
nd.type = FileType;
nd.proxy = proxy;
_dir->nodes[name] = nd;
_map.put(IdentityNodeMap::value_type(_ID, _dir));

return proxy;
}

```

The next significant change is in the `DirectoryI` constructor. The body of the constructor now instantiates all of its immediate children, which effectively causes all nodes to be instantiated recursively.

For each entry in the directory's node table, the constructor locates the matching entry in the Freeze map. The key type of the Freeze map is `Ice::Identity`, so the constructor obtains the key by invoking `ice_getIdentity` on the child's proxy.

```

Filesystem::DirectoryI::DirectoryI(
    const Ice::Identity & id,
    const PersistentDirectoryPtr & dir,
    const DirectoryIPtr & parent) :
    NodeI(id, parent), _dir(dir)
{
    // Instantiate the child nodes
    //
    for (NodeDict::iterator p = dir->nodes.begin();
        p != dir->nodes.end(); ++p) {
        Ice::Identity id = p->second.proxy->ice_getIdentity();
        PersistentNodePtr node = find(id);
        assert(node != 0);
        if (p->second.type == DirType) {
            PersistentDirectoryPtr pDir =
                PersistentDirectoryPtr::dynamicCast(node);
            assert(pDir);
            DirectoryIPtr d = new DirectoryI(id, pDir, this);
        } else {
            PersistentFilePtr pFile =

```



```

        PersistentFilePtr::dynamicCast(node);
        assert(pFile);
        FileIPtr f = new FileI(id, pFile, this);
    }
}

```

If it seems inefficient for the `DirectoryI` constructor to immediately instantiate all of its children, you are right. This clearly will not scale well to large node trees, so why are we doing it?

Previous implementations of the file system service returned transient proxies from `createDirectory` and `createFile`. In other words, if the server was stopped and restarted, any existing child proxies returned by the old instance of the server would no longer work. However, now that we have a persistent store, we should endeavor to ensure that proxies will remain valid across server restarts. There are a couple of implementation techniques that satisfy this requirement:

1. Instantiate all of the servants in advance, as shown in the `DirectoryI` constructor.
2. Use a servant locator.

We chose not to include a servant locator in this example because it complicates the implementation and, as we will see in Section 21.5, a Freeze evictor is ideally suited for this application and a better choice than writing a servant locator.

The last `DirectoryI` method we discuss is `removeChild`, which removes the entry from the node table, and then reinserts the `PersistentDirectory` object into the map to make the change persistent.

```

void
Filesystem::DirectoryI::removeChild(const string & name)
{
    IceUtil::RecMutex::Lock lock(_nodeMutex);

    _dir->nodes.erase(name);
    _map.put(IdentityNodeMap::value_type(_ID, _dir));
}

```

Implementing `NodeI`

There are a few changes to the `NodeI` implementation that should be mentioned. First, you will notice the use of `getPersistentNode` in order to obtain the node's name. We could have simply invoked the `name` operation, but that would incur the overhead of another mutex lock.

Then, the destroy operation removes the node from the Freeze map and sets the `_destroyed` member to true.

The `NodeI` constructor no longer computes a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The `NodeI` constructor therefore must not compute a new identity, but rather simply remember the identity that is given to it. The `NodeI` constructor also constructs the map object (`_map` data member). Remember this object is single-threaded: we use it only in constructors or when we have a write lock on the recursive read-write `_nodeMutex`.

Finally, the `find` function illustrates what needs to be done when iterating over a database that multiple threads can use concurrently. `find` catches `Freeze::DeadlockException` and retries.

```
std::string
Filesystem::NodeI::name(const Ice::Current &) const
{
    IceUtil::RecMutex::Lock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    return getPersistentNode()->name;
}

void
Filesystem::NodeI::destroy(const Ice::Current & current)
{
    IceUtil::RWRecMutex::WLock lock(_nodeMutex);

    if (_destroyed)
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);

    if (!_parent) {
        Filesystem::PermissionDenied e;
        e.reason = "cannot remove root directory";
        throw e;
    }

    _parent->removeChild(getPersistentNode()->name);
    _map.erase(current.id);
    current.adapter->remove(current.id);
    _destroyed = true;
}
```

```

Filesystem::NodeI::NodeI(const Ice::Identity & id,
                        const DirectoryIPtr & parent)
    : _map(Freeze::createConnection(_communicator, _envName),
          _dbName),
      _parent(parent), _destroyed(false), _ID(id)
{
    // Add the identity of self to the object adapter
    //
    _adapter->add(this, _ID);
}

Filesystem::PersistentNodePtr
Filesystem::NodeI::find(const Ice::Identity & id) const
{
    for(;;) {
        try {
            IdentityNodeMap::const_iterator p = _map.find(id);
            if(p == _map.end())
                return 0;
            else
                return p->second;
        }
        catch(const Freeze::DeadlockException &) {
            // Try again
            //
        }
    }
}

```

21.4.3 Implementing the File System Server in Java

In this section we present a Java file system implementation that utilizes a Freeze map for persistent storage. The implementation is based on the one discussed in XREF; in this section we only discuss code that illustrates use of the Freeze map.

Generating the Map

Now that we have selected our key and value types, we can generate the map as follows:

```
$ slice2freezej -I$(ICE_HOME)/slice --dict \
    Filesystem.IdentityNodeMap,Ice::Identity,\
    Filesystem::PersistentNode\
    Filesystem.ice $(ICE_HOME)/slice/Ice/Identity.ice
```

The resulting map class is named `IdentityNodeMap` and is defined in the package `Filesystem`².

The Server main Program

The server's main program is responsible for initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice.Application` as described in Section 12.3.1. Our server main program has now become the following:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public
    Server(String envName)
    {
        _envName = envName;
    }

    public int
    run(String[] args)
    {
        // Install object factories
        //
        communicator().addObjectFactory(
            PersistentFile.ice_factory(),
            PersistentFile.ice_staticId());
        communicator().addObjectFactory(
            PersistentDirectory.ice_factory(),
            PersistentDirectory.ice_staticId());

        // Create an object adapter (stored in the _adapter
        // static member)
        //
        Ice.ObjectAdapter adapter =
```

2. We cannot generate `IdentityNodeMap` in the unnamed (top-level) package because Java's name resolution rules would prevent the implementation classes in the `Filesystem` package from using it. See the Java Language Specification for more information.

```

        communicator().createObjectAdapterWithEndpoints(
            "FreezeFilesystem", "default -p 10000");
DirectoryI._adapter = adapter;
FileI._adapter = adapter;

//
// Set static members used to create connections and maps
//
String dbName = "mapfs";
DirectoryI._communicator = communicator();
DirectoryI._envName = _envName;
DirectoryI._dbName = dbName;
FileI._communicator = communicator();
FileI._envName = _envName;
FileI._dbName = dbName;

// Find the persistent node for the root directory. If
// it doesn't exist, then create it.
Freeze.Connection connection =
    Freeze.Util.createConnection(
        communicator(), _envName);
IdentityNodeMap persistentMap =
    new IdentityNodeMap(connection, dbName, true);

Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
PersistentDirectory pRoot =
    (PersistentDirectory)persistentMap.get(rootId);
if(pRoot == null)
{
    pRoot = new PersistentDirectory();
    pRoot.name = "/";
    pRoot.nodes = new java.util.HashMap();
    persistentMap.put(rootId, pRoot);
}

// Create the root directory (with name "/" and no parent)
//
DirectoryI root = new DirectoryI(rootId, pRoot, null);

// Ready to accept requests now
//
adapter.activate();

// Wait until we are done
//

```

```

        communicator().waitForShutdown();

        // Clean up
        //
        connection.close();

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        app.main("Server", args);
        System.exit(0);
    }

    private String _envName;
}

```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```

public
Server(String envName)
{
    _envName = envName;
}

```

The string argument represents the name of the database environment, and is saved for later use in run.

One of the first tasks run performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. Since these Slice classes have no operations, we can use their built-in factories.

```

communicator().addObjectFactory(
    PersistentFile.ice_factory(),
    PersistentFile.ice_staticId());
communicator().addObjectFactory(
    PersistentDirectory.ice_factory(),
    PersistentDirectory.ice_staticId());

```

Next, we set all the `DirectoryI` and `FileI` static members.

```
Ice.ObjectAdapter adapter =
    communicator().createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");
DirectoryI._adapter = adapter;
FileI._adapter = adapter;

String dbName = "mapfs";
DirectoryI._communicator = communicator();
DirectoryI._envName = _envName;
DirectoryI._dbName = dbName;
FileI._communicator = communicator();
FileI._envName = _envName;
FileI._dbName = dbName;
```

Then we create a Freeze connection and a Freeze map. When the last connection to a Berkeley DB environment is closed, Freeze automatically closes this environment, so keeping a connection in the main function ensures the underlying Berkeley DB environment remains open. Likewise, we keep a map in the main function to keep the underlying Berkeley DB database open.

```
Freeze.Connection connection =
    Freeze.Util.createConnection(
        communicator(), _envName);
IdentityNodeMap persistentMap =
    new IdentityNodeMap(connection, dbName, true);
```

Now we need to initialize the root directory node. We first query the map for the identity of the root directory node; if no match is found, we create a new `PersistentDirectory` instance and insert it into the map.

```
Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
PersistentDirectory pRoot =
    (PersistentDirectory)persistentMap.get(rootId);
if(pRoot == null)
{
    pRoot = new PersistentDirectory();
    pRoot.name = "/";
    pRoot.nodes = new java.util.HashMap();
    persistentMap.put(rootId, pRoot);
}
```

Finally, the main function instantiates the `Server` class, passing `db` as the name of the database environment.

```

public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}

```

The Servant Class Definitions

We also must change the servant classes to incorporate the Freeze map. We are maintaining the design from XREF, but we have added some methods and changed the constructor arguments and state members.

The `FileI` class has three new static members, `_communicator`, `_envName` and `_dbName`, that are used to instantiate the new `_connection` (a Freeze connection) and `_map` (an `IdentityNodeMap`) members in each `FileI` object. We keep the connection so that we can close it explicitly when we no longer need it.

The class also has a new instance member of type `PersistentFile`, representing the persistent state of this file, and a boolean member to indicate whether the node has been destroyed. Finally, we have changed its constructor to accept `Ice.Identity` and `PersistentFile`.

```

package Filesystem;

public class FileI extends _FileDisp
{
    public
    FileI(Ice.Identity id, PersistentFile file, DirectoryI parent)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Ice.Communicator _communicator;
    public static String _envName;
    public static String _dbName;
    public Ice.Identity _ID;
    private Freeze.Connection _connection;
    private IdentityNodeMap _map;
}

```



```

        private PersistentFile _file;
        private DirectoryI _parent;
        private boolean _destroyed;
    }

```

The `DirectoryI` class has undergone a similar transformation.

```

package Filesystem;

public final class DirectoryI extends _DirectoryDisp
{
    public
    DirectoryI(Ice.Identity id, PersistentDirectory dir,
               DirectoryI parent)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Ice.Communicator _communicator;
    public static String _envName;
    public static String _dbName;
    public Ice.Identity _ID;
    private Freeze.Connection _connection;
    private IdentityNodeMap _map;
    private PersistentDirectory _dir;
    private DirectoryI _parent;
    private boolean _destroyed;
}

```

Implementing `FileI`

Let us examine how the implementations have changed. The `FileI` methods are still fairly trivial, but there are a few aspects that need discussion.

First, each operation now checks the `_destroyed` member and raises `Ice::ObjectNotExistException` if the member is true. This is necessary in order to ensure that the Freeze map is kept in a consistent state. For example, if we allowed the `write` operation to proceed after the file node had been destroyed, then we would have mistakenly added an entry back into the Freeze map for that file. Previous file system implementations ignored this issue because it was relatively harmless, but that is no longer true in this version.

Next, notice that the `write` operation calls `put` on the map after changing the `text` member of its `PersistentFile` object. Although this object is a value

in the Freeze map, changing the `text` member has no effect on the persistent state of the map until the value is reinserted into the map, thus overwriting the previous value.

Finally, the constructor now accepts an `Ice::Identity`. This differs from previous implementations in that the identity used to be computed by the constructor. In order to make our servants truly persistent, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The `FileI` constructor therefore must not compute a new identity, but rather simply remember the identity that is given to it.

Similarly, the constructor is not responsible for creating a `PersistentFile` object, but rather is given one. This accommodates our two use cases: creating a new file, and restoring an existing file from the map.

```
public
FileI(Ice.Identity id, PersistentFile file,
      DirectoryI parent)
{
    _connection =
        Freeze.Util.createConnection(_communicator, _envName);
    _map =
        new IdentityNodeMap(_connection, _dbName, false);
    _ID = id;
    _file = file;
    _parent = parent;
    _destroyed = false;

    assert(_parent != null);

    // Add the identity of self to the object adapter
    //
    _adapter.add(this, _ID);
}

public synchronized String
name(Ice.Current current)
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    return _file.name;
}

public synchronized void
destroy(Ice.Current current)
```

```

        throws PermissionDenied
    {
        if (_destroyed)
            throw new Ice.ObjectNotExistException();

        _parent.removeChild(_file.name);
        _map.remove(current.id);
        current.adapter.remove(current.id);
        _map.close();
        _connection.close();
        _destroyed = true;
    }

    public synchronized String[]
    read(Ice.Current current)
    {
        if (_destroyed)
            throw new Ice.ObjectNotExistException();

        return _file.text;
    }

    public synchronized void
    write(String[] text, Ice.Current current)
        throws GenericError
    {
        if (_destroyed)
            throw new Ice.ObjectNotExistException();

        _file.text = text;
        _map.put(_ID, _file);
    }

```

Implementing DirectoryI

The DirectoryI implementation requires more substantial changes. We begin our discussion with the createDirectory operation.

```

    public synchronized DirectoryPrx
    createDirectory(String name, Ice.Current current)
        throws NameInUse, IllegalName
    {
        if (_destroyed)
            throw new Ice.ObjectNotExistException();

        checkName(name);

```

```

    PersistentDirectory persistentDir
        = new PersistentDirectory();
    persistentDir.name = name;
    persistentDir.nodes = new java.util.HashMap();
    DirectoryI dir = new DirectoryI(
        Ice.Util.stringToIdentity(Ice.Util.generateUUID()),
        persistentDir, this);
    assert(_map.get(dir._ID) == null);
    _map.put(dir._ID, persistentDir);

    DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(dir._ID));

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.DirType;
    nd.proxy = proxy;
    _dir.nodes.put(name, nd);
    _map.put(_ID, _dir);

    return proxy;
}

```

After validating the node name, the operation creates a `PersistentDirectory`³ object for the child directory, which is passed to the `DirectoryI` constructor along with a unique identity. Next, we store the child's `PersistentDirectory` object in the Freeze map. Finally, we initialize a new `NodeDesc` value and insert it into the parent's node table and then reinsert the parent's `PersistentDirectory` object into the Freeze map.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```

public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();
}

```

3. Since this Slice class has no operations, the compiler generates a concrete class that an application can instantiate.

```

        checkName(name);

        PersistentFile persistentFile = new PersistentFile();
        persistentFile.name = name;
        FileI file = new FileI(Ice.Util.stringToIdentity(
            Ice.Util.generateUUID()), persistentFile, this);
        assert(_map.get(file._ID) == null);
        _map.put(file._ID, persistentFile);

        FilePrx proxy = FilePrxHelper.uncheckedCast(
            current.adapter.createProxy(file._ID));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.FileType;
        nd.proxy = proxy;
        _dir.nodes.put(name, nd);
        _map.put(_ID, _dir);

        return proxy;
    }

```

The next significant change is in the `DirectoryI` constructor. The body of the constructor now instantiates all of its immediate children, which effectively causes all nodes to be instantiated recursively.

For each entry in the directory's node table, the constructor locates the matching entry in the Freeze map. The key type of the Freeze map is `Ice::Identity`, so the constructor obtains the key by invoking `ice_getIdentity` on the child's proxy.

```

    public
    DirectoryI(Ice.Identity id, PersistentDirectory dir,
               DirectoryI parent)
    {
        _connection =
            Freeze.Util.createConnection(_communicator, _envName);
        _map =
            new IdentityNodeMap(_connection, _dbName, false);
        _ID = id;
        _dir = dir;
        _parent = parent;
        _destroyed = false;

        // Add the identity of self to the object adapter
        //

```

```

        _adapter.add(this, _ID);

        // Instantiate the child nodes
        //
        java.util.Iterator p = dir.nodes.values().iterator();
        while (p.hasNext()) {
            NodeDesc desc = (NodeDesc)p.next();
            Ice.Identity ident = desc.proxy.ice_getIdentity();
            PersistentNode node = (PersistentNode)_map.get(ident);
            assert(node != null);
            if (desc.type == NodeType.DirType) {
                PersistentDirectory pDir
                    = (PersistentDirectory)node;
                DirectoryI d = new DirectoryI(ident, pDir, this);
            } else {
                PersistentFile pFile = (PersistentFile)node;
                FileI f = new FileI(ident, pFile, this);
            }
        }
    }
}

```

If it seems inefficient for the `DirectoryI` constructor to immediately instantiate all of its children, you are right. This clearly will not scale well to large node trees, so why are we doing it?

Previous implementations of the file system service returned transient proxies from `createDirectory` and `createFile`. In other words, if the server was stopped and restarted, any existing child proxies returned by the old instance of the server would no longer work. However, now that we have a persistent store, we should endeavor to ensure that proxies will remain valid across server restarts. There are a couple of implementation techniques that satisfy this requirement:

1. Instantiate all of the servants in advance, as shown in the `DirectoryI` constructor.
2. Use a servant locator.

We chose not to include a servant locator in this example because it complicates the implementation and, as we will see in Section 21.5, a Freeze evictor is ideally suited for this application and a better choice than writing a servant locator.

The last `DirectoryI` method we discuss is `removeChild`, which removes the entry from the node table, and then reinserts the `PersistentDirectory` object into the map to make the change persistent.

```
synchronized void  
removeChild(String name)  
{  
    _dir.nodes.remove(name);  
    _map.put(_ID, _dir);  
}
```

21.5 The Freeze Evictor

The Freeze evictor combines persistence and scalability features into a single facility that is easily incorporated into Ice applications.

As an implementation of the `ServantLocator` interface (see Section 16.6), the Freeze evictor takes advantage of the fundamental separation between Ice object and servant to activate servants on-demand from persistent storage, and to deactivate them again using customized eviction constraints. Although an application may have thousands of Ice objects in its database, it is not practical to have servants for all of those Ice objects resident in memory simultaneously. The application can conserve resources and gain greater scalability by setting an upper limit on the number of active servants, and letting the Freeze evictor handle the details of servant activation, persistence, and deactivation.

The Freeze evictor maintains a queue of active servants, ordered using a “least recently used” eviction algorithm: if the queue is full, the least recently used servant is evicted to make room for a new servant.

Here is the sequence of events for activating a servant as shown in Figure 21.3. Let us assume that we have configured the evictor with a size of five, that the queue is full, and that a request has arrived for a servant that is not currently active.

1. A client invokes an operation.
2. The object adapter invokes on the evictor to locate the servant.
3. The evictor first checks its active servant queue and fails to find the servant, so it instantiates the servant and restores its persistent state from the database.
4. The evictor adds an item for the servant (servant 1) at the head of the queue.
5. The queue’s length now exceeds the configured maximum, so the evictor removes servant 6 from the queue as soon as it is eligible for eviction. This occurs when there are no outstanding requests pending on servant 6, and its state has been safely stored in the database.

6. The object adapter dispatches the request to the new servant.

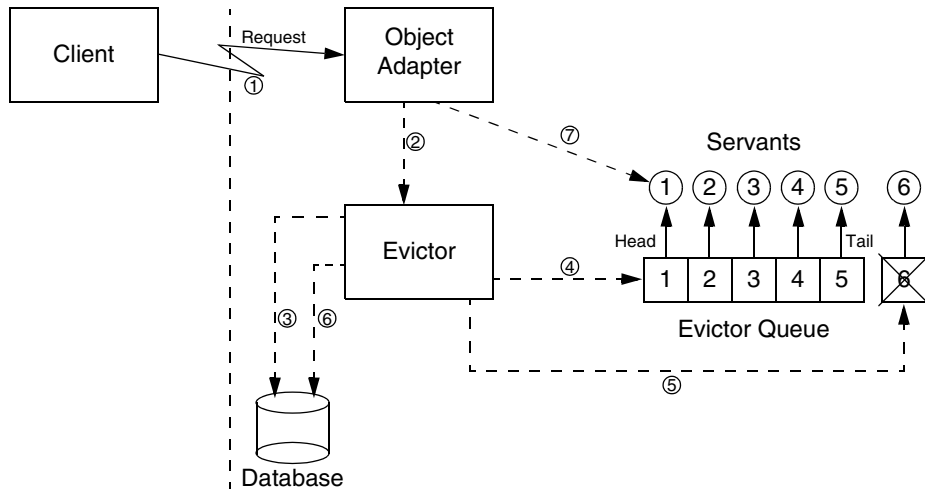


Figure 21.3. An evictor queue after restoring servant 1 and evicting servant 6.

21.5.1 Object Factories

The Freeze evictor is a generic facility in that it manages instances of Object subclasses. Applications are therefore free to use as many object types as necessary, with one requirement: an Ice object factory must be registered for each type.

21.5.2 Facets

A facet is an Ice object of an arbitrary type that is attached to another Ice object. Facets can also have facets, enabling the creation of an object tree with the *main object* at its root. Although a client can make invocations on facets of the main object, only the main object is considered to be a servant. In other words, if an application uses an object adapter, only the main object is registered, and not any of its facets.

Facets require special consideration where persistence is concerned. An inefficient approach would save the state of the main object and all of its facets whenever any one of the facets had been modified. The Freeze evictor, on the other

hand, treats facets as discrete entities when saving state: each facet occupies its own record in the database and is capable of being saved independently of the main object and the other facets.

In other situations, however, the Freeze evictor treats the main object and its facets as a logical unit:

- When first registering an Ice object with the evictor using `createObject`, the evictor saves the persistent state of the main object, as well as that of any facets currently attached to the main object.
- When activating an object in response to an incoming request, the main object and all of its facets are restored from persistent state, regardless of whether the incoming request is for the main object or one of the facets.
- Facets are only evicted when the main object is evicted.

For more information on Ice facets, see XREF.

21.5.3 Detecting Updates

The Freeze Evictor considers that a facet has been modified when a mutating operation on this facet completes. (See Section 4.8 for more information on mutating operations.) Updates made internally, through calls that are not dispatched through the Freeze Evictor, are not detected and can be lost.

21.5.4 Saving Thread

All persistence activity of a Freeze evictor is handled in a background thread created by the evictor. This thread wakes up periodically and saves the state of all newly-registered, modified, and destroyed Ice facets in the evictor's queue.

For applications that experience bursts of activity, resulting in a large number of modified Ice facets in a short period of time, the evictor's thread can also be configured to begin saving facet state as soon as the number of modified facets reaches a certain threshold.

21.5.5 Synchronization

When the saving thread takes a snapshot of a facet it is about to save, it is necessary to prevent the application from modifying the facet's persistent data members at the same time.

The Freeze evictor and the application need to use a common synchronization to ensure correct behavior. In Java, this common synchronization is the facet

itself: the Freeze evictor locks the facet (a Java object) while taking the snapshot. In C++, the facet is required to inherit from the class

`IceUtil::AbstractMutex`: the Freeze evictor locks the facet through this interface while taking a snapshot. On the application side, the facet's implementation is required to synchronize all operations that access the facet's persistent data members using the same mechanism.

21.5.6 Index

The Freeze evictor supports the use of indices to quickly find persistent objects using the value of a data member as the search criteria. Only the data members of the main Ice object can be indexed, and they are restricted to the same types allowed for Slice dictionary keys (see Section 4.7.4).

The `slice2freeze` and `slicefreezej` tools can generate an Index class when passed the `--index` option:

- `--index CLASS,TYPE,MEMBER[,case-sensitive|case-insensitive]`

CLASS is the name of the class to be generated. **TYPE** denotes the type of class to be indexed (objects of different classes are not included in this index). **MEMBER** is the name of the data member in **TYPE** to index. When **MEMBER** has type `string`, it is possible to specify whether the index is case-sensitive or not. The default is case-sensitive.

The generated Index class supplies three methods:

- `sequence<Ice::Identity> findFirst(member-type index, int firstN)`
Returns up to `firstN` objects of **TYPE** whose **MEMBER** is equal to `index`. This is useful to avoid running out of memory if the number of objects matching the criteria can be very large.
- `sequence<Ice::Identity> find(member-type index)`
Returns all the objects of **TYPE** whose **MEMBER** is equal to `index`.
- `int count(<type> index)`
Returns the number of objects of **TYPE** having **MEMBER** equal to `index`.

Indices are associated with a Freeze evictor during evictor creation. See the definition of the `createEvictor` methods for details.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every single update trig-

gers a read from the database to get the old index entry and, if necessary, replace it.

21.5.7 Using a Servant_INITIALIZER

In some applications it may be necessary to initialize a servant after it is instantiated by the evictor but before an operation is dispatched to it. The Freeze evictor allows an application to specify a servant initializer for this purpose.

To clarify the sequence of events, let us assume that a request has arrived for an Ice object that is not currently active:

1. The evictor restores a servant for the Ice object from the database. This involves two steps:
 1. The Ice run time locates and invokes the factory for the Ice object's type, thereby obtaining a new instance with uninitialized data members. Attached facets are also recreated using the registered factories.
 2. The data members are populated from the persistent state.
2. The evictor invokes the application's servant initializer (if any) for the servant.
3. The evictor adds the servant to its least-recently-used queue.
4. The evictor dispatches the operation.

The servant initializer is called on the servant while the evictor holds an internal lock on its data structures. Therefore, the servant initializer should not make remote invocations that could come back to the same Freeze evictor.

The file system implementation presented in Section 21.6 on page 613 demonstrates the use of a servant initializer.

21.5.8 Application Design Considerations

The Freeze evictor creates a snapshot of an Ice object's state for persistent storage by marshaling the Ice object, just as if the Ice object were being sent "over the wire" as a parameter to a remote invocation. Therefore, the Slice definitions for an object type must include the data members comprising the object's persistent state.

For example, we could define a Slice class as follows:

```
class Statel ess {  
    void calc();  
};
```

However, without data members, there will not be any persistent state in the database for objects of this type, and hence there is little value in using the Freeze evictor for this type.

Obviously, Slice object types need to define data members, but there are other design considerations as well. For example, suppose we define a simple application as follows:

```
class Account {
    void withdraw(int amount);
    void deposit(int amount);

    int balance;
};

interface Bank {
    Account* createAccount();
};
```

In this application, we would use the Freeze evictor to manage Account objects that have a data member `balance` representing the persistent state of an account.

From an object-oriented design perspective, there is a glaring problem with these Slice definitions: implementation details (the persistent state) are exposed in the client-server contract. The client cannot directly manipulate the `balance` member because the Bank interface returns Account proxies, not Account instances. However, the presence of the data member may cause unnecessary confusion for client developers.

A better alternative is to clearly separate the persistent state as shown below:

```
interface Account {
    void withdraw(int amount);
    void deposit(int amount);
};

interface Bank {
    Account* createAccount();
};

class PersistentAccount implements Account {
    int balance;
};
```

Now the Freeze evictor can manage PersistentAccount objects, while clients interact with Account proxies. (Ideally, PersistentAccount would be defined in a different source file and inside a separate module.)

21.6 Using the Freeze Evictor in a File System Server

In this section, we present file system implementations that utilize a Freeze evictor. The implementations are based on the ones discussed in XREF, and in this section we only discuss code that illustrates use of the Freeze evictor.

In general, incorporating a Freeze evictor into your application requires the following steps:

1. Evaluate your existing Slice definitions for a suitable persistent object type.
2. If no suitable type is found, you typically define a new derived class which captures your persistent state requirements. Consider placing these definitions in a separate file: they are only used by the server for persistence, and therefore do not need to appear in the “public” definitions required by clients. Also consider placing your persistent types in a separate module to avoid name clashes.
3. Generate code (using `slice2freeze` or `slice2freezej`) for your new definitions.
4. Create an evictor and register it as a servant locator with an object adapter.
5. Create instances of your persistent type and register them with the evictor.

21.6.1 Slice Definitions

Fortunately, it is unnecessary for us to change any of the existing file system Slice definitions to incorporate the Freeze evictor. However, we do need to add some definitions to express our persistent state requirements:

```
module Filesystem {
    class PersistentDirectory;

    class PersistentNode implements Node {
        string nodeName;
        PersistentDirectory* parent;
    };

    class PersistentFile extends PersistentNode implements File {
        Lines text;
    };

    class PersistentDirectory extends PersistentNode
        implements Directory {
        void removeNode(string name);
```

```

        NodeDict nodes;
    };
};

```

As you can see, we have subclassed all of the node interfaces. Let us examine each one in turn.

The `PersistentNode` class adds two data members: `nodeName`⁴ and `parent`. The file system implementation requires that a child node know its parent node in order to properly implement the destroy operation. Previous implementations had a state member of type `DirectoryI`, but that is not workable here. It is no longer possible to pass the parent node to the child node's constructor because the evictor may be instantiating the child node (via a factory), and the parent node will not be known. Even if it were known, another factor to consider is that there is no guarantee that the parent node will be active when the child invokes on it, because the evictor may have evicted it. We solve these issues by storing a proxy to the parent node. If the child node invokes on the parent node via the proxy, the evictor automatically activates the parent node if necessary.

The `PersistentFile` class is very straightforward, simply adding a text member representing the contents of the file. Notice that the class extends `PersistentNode`, and therefore inherits the state members declared by the base class.

Finally, the `PersistentDirectory` class defines the `removeNode` operation, and adds the `nodes` state member representing the immediate children of the directory node. Since a child node contains only a proxy for its `PersistentDirectory` parent, and not a reference to an implementation class, there must be a Slice-defined operation that can be invoked when the child is destroyed.

If we had followed the advice at the beginning of Section 21.5, we would have defined `Node`, `File`, and `Directory` classes in a separate `PersistentFilesystem` module, but in this example we use the existing `Filesystem` module for the sake of simplicity.

4. We used `nodeName` instead of `name` because `name` is already used as an operation in the `Node` interface.


```

    Freeze::ServantInitializerPtr init = new NodeInitializer;
    NodeI::_evictor->installServantInitializer(init);
    NodeI::_adapter->addServantLocator(NodeI::_evictor, "");

    // Create the root node if it doesn't exist
    //
    Ice::Identity rootId = Ice::stringToIdentity("RootDir");
    if (!NodeI::_evictor->hasObject(rootId)) {
        PersistentDirectoryPtr root = new DirectoryI(rootId);
        root->nodeName = "/";
        NodeI::_evictor->createObject(rootId, root);
    }

    // Ready to accept requests now
    //
    NodeI::_adapter->activate();

    // Wait until we are done
    //
    communicator()->waitForShutdown();
    if (interrupted()) {
        cerr << appName()
              << ": received signal, shutting down" << endl;
    }

    return 0;
}

private:
    string _dbEnvName;
};

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}

```

Let us examine the changes in detail. First, we are now including `Freeze/Freeze.h` instead of `Ice/Ice.h`. This `Freeze` header file includes all of the other `Freeze` (and `Ice`) header files this source file requires.

Next, we define the class `FilesystemApp` as a subclass of `Ice::Application`, and provide a constructor taking a string argument:


```
FilesystemApp(const string & dbEnvName) :
    _dbEnvName(dbEnvName) { }
```

The string argument represents the name of the database environment, and is saved for later use in run.

One of the first tasks run performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```
Ice::ObjectFactoryPtr factory = new NodeFactory;
communicator()->addObjectFactory(
    factory,
    PersistentFile::ice_staticId());
communicator()->addObjectFactory(
    factory,
    PersistentDirectory::ice_staticId());
```

After creating the object adapter, the program initializes a Freeze evictor by invoking `createEvictor`. The third argument to `createEvictor` is the name of the database, which by default is created if it does not exist. A servant initializer is then installed, and the evictor is added to the object adapter as a servant locator for the default category.

```
NodeI::_evictor =
    Freeze::createEvictor(communicator(), _dbEnvName,
                          "evictorfs");
Freeze::ServantInitializerPtr init = new NodeInitializer;
NodeI::_evictor->installServantInitializer(init);
NodeI::_adapter->addServantLocator(NodeI::_evictor, "");
```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice::Identity rootId = Ice::stringToIdentity("RootDir");
if (!NodeI::_evictor->hasObject(rootId)) {
    PersistentDirectoryPtr root = new DirectoryI(rootId);
    root->nodeName = "/";
    NodeI::_evictor->createObject(rootId, root);
}
```

Finally, the main function instantiates the `FilesystemApp`, passing `db` as the name of the database environment.

```

int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}

```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. We are maintaining the multiple-inheritance design from XREF, but we have changed the constructors and state members. In particular, each node implementation class has two constructors, one taking no parameters and one taking an `Ice::Identity`. The former is needed by the factory, and the latter is used when the node is first created. To comply with the evictor requirements, `NodeI` implements `IceUtil::AbstractMutex` by deriving from the template class `IceUtil::AbstractMutexI`. The only other change of interest is a new state member in `NodeI` named `_evictor`.

```

namespace Filesystem {
    class NodeI : virtual public PersistentNode,
                  public IceUtil::AbstractMutexI<IceUtil::Mutex> {
    public:
        virtual std::string name(const Ice::Current &) const;
        virtual void destroy(const Ice::Current &);
        static Ice::ObjectAdapterPtr _adapter;
        static Freeze::EvictorPtr _evictor;
    protected:
        NodeI();
        NodeI(const Ice::Identity &);
    public:
        const Ice::Identity _ID;
    };

    class FileI : virtual public PersistentFile,
                  virtual public NodeI {
    public:
        virtual Lines read(const Ice::Current &) const;
        virtual void write(const Lines &,
                           const Ice::Current &);

        FileI();
        FileI(const Ice::Identity &);
    };

    class DirectoryI : virtual public PersistentDirectory,

```

```

        virtual public NodeI {
public:
    virtual NodeDict list(ListMode,
                        const Ice::Current &) const;
    virtual NodeDesc resolve(const std::string &,
                        const Ice::Current &) const;
    virtual DirectoryPrx createDirectory(const std::string &,
                        const Ice::Current&);
    virtual FilePrx createFile(const std::string &,
                        const Ice::Current &);
    virtual void destroy(const Ice::Current &);
    virtual void removeNode(const std::string &,
                        const Ice::Current &);

    DirectoryI();
    DirectoryI(const Ice::Identity &);
protected:
    void listRecursive(const std::string &,
                    const DirectoryPrx &,
                    NodeDict &) const;
private:
    void checkName(const std::string &) const;
    };
}

```

In addition to the node implementation classes, we have also declared implementations of an object factory and a servant initializer:

```

namespace Filesystem {
    class NodeFactory : virtual public Ice::ObjectFactory {
public:
    virtual Ice::ObjectPtr create(const std::string &);
    virtual void destroy();
    };

    class NodeInitializer :
        virtual public Freeze::ServantInitializer {
public:
    virtual void initialize(const Ice::ObjectAdapterPtr &,
                        const Ice::Identity &,
                        const Ice::ObjectPtr &);
    };
}

```

Implementing NodeI

Notice that the NodeI constructor no longer computes a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. The NodeI constructor therefore must not compute a new identity, but rather remember the identity that is given to it.

```
string
Filesystem::NodeI::name(const Ice::Current &) const
{
    return nodeName;
}

void
Filesystem::NodeI::destroy(const Ice::Current & current)
{
    IceUtil::Mutex::Lock lock(*this);

    if (!parent) {
        Filesystem::PermissionDenied e;
        e.reason = "cannot remove root directory";
        throw e;
    }

    parent->removeNode(nodeName);
    _evictor->destroyObject(_ID);
}

Filesystem::NodeI::NodeI()
{
}

Filesystem::NodeI::NodeI(const Ice::Identity & id)
    : _ID(id)
{
}
```

Implementing FileI

The FileI methods are still fairly trivial, because the Freeze evictor is handling persistence for us.

```

Filesystem::Lines
Filesystem::FileI::read(const Ice::Current &) const
{
    IceUtil::Mutex::Lock lock(*this);

    return text;
}

void
Filesystem::FileI::write(const Filesystem::Lines & text,
                        const Ice::Current &)
{
    IceUtil::Mutex::Lock lock(*this);

    this->text = text;
}

Filesystem::FileI::FileI()
{
}

Filesystem::FileI::FileI(const Ice::Identity & id)
    : NodeI(id)
{
}

```

Implementing DirectoryI

The DirectoryI implementation requires more substantial changes. We begin our discussion with the createDirectory operation.

```

Filesystem::DirectoryPrx
Filesystem::DirectoryI::createDirectory(
    const string & name,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock lock(*this);

    checkName(name);

    Ice::Identity id =
        Ice::stringToIdentity(IceUtil::generateUUID());
    PersistentDirectoryPtr dir = new DirectoryI(id);
    dir->nodeName = name;
    dir->parent = PersistentDirectoryPrx::uncheckedCast(
        current.adapter->createProxy(_ID));
}

```

```

        _evictor->createObject(id, dir);

        DirectoryPrx proxy = DirectoryPrx::uncheckedCast(
            current.adapter->createProxy(id));

        NodeDesc nd;
        nd.name = name;
        nd.type = DirType;
        nd.proxy = proxy;
        nodes[name] = nd;

        return proxy;
    }

```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the `createFile` operation has the same structure as `createDirectory`.

```

Filesystem::FilePrx
Filesystem::DirectoryI::createFile(
    const string & name,
    const Ice::Current & current)
{
    IceUtil::Mutex::Lock lock(*this);

    checkName(name);

    Ice::Identity id =
        Ice::stringToIdentity(IceUtil::generateUUID());
    PersistentFilePtr file = new FileI(id);
    file->nodeName = name;
    file->parent = PersistentDirectoryPrx::uncheckedCast(
        current.adapter->createProxy(_ID));
    _evictor->createObject(id, file);

    FilePrx proxy = FilePrx::uncheckedCast(
        current.adapter->createProxy(id));

    NodeDesc nd;
    nd.name = name;
    nd.type = FileType;
    nd.proxy = proxy;
}

```

```

        nodes[name] = nd;

        return proxy;
    }

```

Implementing NodeFactory

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```

Ice::ObjectPtr
Filesystem::NodeFactory::create(const string & type)
{
    if (type == "::Filesystem::PersistentFile")
        return new FileI;
    else if (type == "::Filesystem::PersistentDirectory")
        return new DirectoryI;
    else {
        assert(false);
        return 0;
    }
}

void
Filesystem::NodeFactory::destroy()
{
}

```

Implementing NodeInitializer

`NodeInitializer` is a trivial implementation of the `Freeze::ServantInitializer` interface whose only responsibility is setting the `_ID` member of the node implementation. The evictor invokes the `initialize` operation after the evictor has created a servant and restored its persistent state from the database, but before any operations are dispatched to it.

```

void
Filesystem::NodeInitializer::initialize(
    const Ice::ObjectAdapterPtr &,
    const Ice::Identity & id,
    const Ice::ObjectPtr & obj)
{
}

```

```

        NodeIPtr node = NodeIPtr::dynamicCast(obj);
        assert(node);
        const_cast<Ice::Identity&>(node->_ID) = id;
    }

```

21.6.3 Implementing the File System Server in Java

The Server main Program

The server's main program is responsible for creating the evictor and initializing the root directory node. Many of the administrative duties, such as creating and destroying a communicator, are handled by the class `Ice.Application` as described in Section 12.3.1. Our server main program has now become the following:

```

import Filesystem.*;

public class Server extends Ice.Application {
    public
    Server(String dbEnvName)
    {
        _dbEnvName = dbEnvName;
    }

    public int
    run(String[] args)
    {
        // Install object factories
        //
        Ice.ObjectFactory factory = new NodeFactory();
        communicator().addObjectFactory(
            factory,
            PersistentFile.ice_staticId());
        communicator().addObjectFactory(
            factory,
            PersistentDirectory.ice_staticId());

        // Create an object adapter (stored in the _adapter
        // static member)
        //
        Ice.ObjectAdapter adapter =
            communicator().createObjectAdapterWithEndpoints(
                "FreezeFilesystem", "default -p 10000");
        DirectoryI._adapter = adapter;
    }
}

```

```

    FileI._adapter = adapter;

    // Create the Freeze evictor (stored in the _evictor
    // static member)
    //
    Freeze.Evictor evictor =
        Freeze.Util.createEvictor(communicator(), _dbEnvName,
                                "evictorfs", null, true);
    DirectoryI._evictor = evictor;
    FileI._evictor = evictor;
    Freeze.ServantInitializer init = new NodeInitializer();
    evictor.installServantInitializer(init);
    adapter.addServantLocator(evictor, "");

    // Create the root node if it doesn't exist
    //
    Ice.Identity rootId =
        Ice.Util.stringToIdentity("RootDir");
    if(!evictor.hasObject(rootId))
    {
        PersistentDirectory root = new DirectoryI(rootId);
        root.nodeName = "/";
        root.nodes = new java.util.HashMap();
        evictor.createObject(rootId, root);
    }

    // Ready to accept requests now
    //
    adapter.activate();

    // Wait until we are done
    //
    communicator().waitForShutdown();

    return 0;
}

public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}

```

```

    }

    private String _dbEnvName;
}

```

Let us examine the changes in detail. First, we define the class `Server` as a subclass of `Ice.Application`, and provide a constructor taking a string argument:

```

public
Server(String dbEnvName)
{
    _dbEnvName = dbEnvName;
}

```

The string argument represents the name of the database environment, and is saved for later use in run.

One of the first tasks run performs is installing the Ice object factories for `PersistentFile` and `PersistentDirectory`. Although these classes are not exchanged via Slice operations, they are marshalled and unmarshalled in exactly the same way when saved to and loaded from the database, therefore factories are required. A single instance of `NodeFactory` is installed for both types.

```

Ice.ObjectFactory factory = new NodeFactory();
communicator().addObjectFactory(
    factory,
    PersistentFile.ice_staticId());
communicator().addObjectFactory(
    factory,
    PersistentDirectory.ice_staticId());

```

After creating the object adapter, the program initializes a Freeze evictor by invoking `createEvictor`. The third argument to `createEvictor` is the name of the database, the null argument indicates no indices are in use, and the true argument requests that the database be created if it does not exist. A servant initializer is then installed, and the evictor is added to the object adapter as a servant locator for the default category.

```

Freeze.Evictor evictor =
    Freeze.Util.createEvictor(communicator(), _dbEnvName,
                             "evictorfs", null, true);

DirectoryI._evictor = evictor;
FileI._evictor = evictor;
Freeze.ServantInitializer init = new NodeInitializer();
evictor.installServantInitializer(init);
adapter.addServantLocator(evictor, "");

```

Next, the program creates the root directory node if it is not already being managed by the evictor.

```
Ice.Identity rootId =
    Ice.Util.stringToIdentity("RootDir");
if(!evictor.hasObject(rootId))
{
    PersistentDirectory root = new DirectoryI(rootId);
    root.nodeName = "/";
    root.nodes = new java.util.HashMap();
    evictor.createObject(rootId, root);
}
```

Finally, the main function instantiates the `Server` class, passing `db` as the name of the database environment.

```
public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}
```

The Servant Class Definitions

The servant classes must also be changed to incorporate the Freeze evictor. We are maintaining the design from XREF, but we have changed the constructors and state members. In particular, each node implementation class has two constructors, one taking no parameters and one taking an `Ice::Identity`. The former is needed by the factory, and the latter is used when the node is first created. The only other change of interest is the new state member `_evictor`.

The `FileI` class has a new static member of type `IdentityNodeMap`, which avoids the need to duplicate this reference in each node. The class has a new instance member of type `PersistentFile`, representing the persistent state of this file. Finally, we have changed the constructor to accept an `Ice.Identity` and a `PersistentFile`.

```
package FileSystem;

public class FileI extends PersistentFile
{
    public
    FileI()
    {
```

```

        // ...
    }

    public
    FileI(Ice.Identity id)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Freeze.Evictor _evictor;
    public Ice.Identity _ID;
}

```

The `DirectoryI` class has undergone a similar transformation.

```

package Filesystem;

public final class DirectoryI extends PersistentDirectory
{
    public
    DirectoryI()
    {
        // ...
    }

    public
    DirectoryI(Ice.Identity id)
    {
        // ...
    }

    // ... Ice operations ...

    public static Ice.ObjectAdapter _adapter;
    public static Freeze.Evictor _evictor;
    public Ice.Identity _ID;
}

```

Notice that the constructors no longer compute a value for the identity. This is necessary in order to make our servants truly persistent. Specifically, the identity for a node is computed once when that node is created, and must remain the same for the lifetime of the node. A constructor therefore must not compute a new identity, but rather remember the identity given to it.

Implementing FileI

The FileI methods are still fairly trivial, because the Freeze evictor is handling persistence for us.

```
public
FileI()
{
}

public
FileI(Ice.Identity id)
{
    _ID = id;
}

public String
name(Ice.Current current)
{
    return nodeName;
}

public synchronized void
destroy(Ice.Current current)
    throws PermissionDenied
{
    parent.removeNode(nodeName);
    _evictor.destroyObject(_ID);
}

public synchronized String[]
read(Ice.Current current)
{
    return text;
}

public synchronized void
write(String[] text, Ice.Current current)
    throws GenericError
{
    this.text = text;
}
```

Implementing DirectoryI

The DirectoryI implementation requires more substantial changes. We begin our discussion with the createDirectory operation.

```
public synchronized DirectoryPrx
createDirectory(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    checkName(name);

    Ice.Identity id =
        Ice.Util.stringToIdentity(Ice.Util.generateUUID());
    PersistentDirectory dir = new DirectoryI(id);
    dir.nodeName = name;
    dir.parent = PersistentDirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(_ID));
    dir.nodes = new java.util.HashMap();
    _evictor.createObject(id, dir);

    DirectoryPrx proxy = DirectoryPrxHelper.uncheckedCast(
        current.adapter.createProxy(id));

    NodeDesc nd = new NodeDesc();
    nd.name = name;
    nd.type = NodeType.DirType;
    nd.proxy = proxy;
    nodes.put(name, nd);

    return proxy;
}
```

After validating the node name, the operation obtains a unique identity for the child directory, instantiates the servant, and registers it with the Freeze evictor. Finally, the operation creates a proxy for the child and adds the child to its node table.

The implementation of the createFile operation has the same structure as createDirectory.

```
public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    checkName(name);

    Ice.Identity id =
```

```

        Ice.Util.stringToIdentity(Ice.Util.generateUUID());
        PersistentFile file = new FileI(id);
        file.nodeName = name;
        file.parent = PersistentDirectoryPrxHelper.uncheckedCast(
            current.adapter.createProxy(_ID));
        _evictor.createObject(id, file);

        FilePrx proxy = FilePrxHelper.uncheckedCast(
            current.adapter.createProxy(id));

        NodeDesc nd = new NodeDesc();
        nd.name = name;
        nd.type = NodeType.FileType;
        nd.proxy = proxy;
        nodes.put(name, nd);

        return proxy;
    }

```

Implementing NodeFactory

We use a single factory implementation for creating two types of Ice objects: `PersistentFile` and `PersistentDirectory`. These are the only two types that the Freeze evictor will be restoring from its database.

```

package Filesystem;

public class NodeFactory extends Ice.LocalObjectImpl
    implements Ice.ObjectFactory
{
    public Ice.Object
    create(String type)
    {
        if (type.equals("::Filesystem::PersistentFile"))
            return new FileI();
        else if (type.equals("::Filesystem::PersistentDirectory"))
            return new DirectoryI();
        else {
            assert(false);
            return null;
        }
    }

    public void

```

```

        destroy()
    {
    }
}

```

Implementing NodeInitializer

NodeInitializer is a trivial implementation of the Freeze::ServantInitializer interface whose only responsibility is setting the `_ID` member of the node implementation. The evictor invokes the `initialize` operation after the evictor has created a servant and restored its persistent state from the database, but before any operations are dispatched to it.

```

package Filesystem;

public class NodeInitializer extends Ice.LocalObjectImpl
    implements Freeze.ServantInitializer {
    public void
    initialize(Ice.ObjectAdapter adapter, Ice.Identity id,
               Ice.Object obj)
    {
        if (obj instanceof FileI)
            ((FileI)obj)._ID = id;
        else
            ((DirectoryI)obj)._ID = id;
    }
}

```

21.7 Summary

Freeze is a collection of services that simplify the use of persistence in Ice applications. The Freeze map is an associative container mapping any Slice key and value types, providing a convenient and familiar interface to a persistent map. The Freeze evictor is an especially powerful facility for supporting persistent Ice objects in a highly-scalable implementation.

Chapter 22

FreezeScript

22.1 Chapter Overview

This chapter describes the FreezeScript tools for migrating and inspecting the databases created by Freeze maps and evictors. The discussion of database migration begins in Section 22.3 and continues through Section 22.5. Database inspection is presented in Section 22.6 and Section 22.7. Finally, Section 22.8 describes the expression language supported by the FreezeScript tools.

22.2 Introduction

As described in Chapter 21, Freeze supplies a valuable set of services for simplifying the use of persistence in Ice applications. However, while Freeze makes it easy for an application to manage its persistent state, there are additional administrative responsibilities that must also be addressed:

- Migration

As an application evolves, it is not unusual for the types describing its persistent state to evolve as well. When these changes occur, a great deal of time can be saved if existing databases can be migrated to the new format while preserving as much information as possible.

- Inspection

The ability to examine a database can be helpful during every stage of the application’s lifecycle, from development to deployment.

FreezeScript provides tools for performing both of these activities on Freeze map and evictor databases. These databases have a well-defined structure because the key and value of each record consist of the marshaled bytes of their respective Slice types. This design allows the FreezeScript tools to operate on any Freeze database using only the Slice definitions for the database types.

22.3 Database Migration

The FreezeScript tool **transformdb** migrates a database created by a Freeze map or evictor. It accomplishes this by comparing the “old” Slice definitions (i.e., the ones that describe the current contents of the database) with the “new” Slice definitions, and making whatever modifications are necessary to ensure that the transformed database is compatible with the new definitions.

This would be difficult to achieve by writing a custom transformation program because that program would require static knowledge of the old and new types, which frequently define many of the same symbols and would therefore prevent the program from being loaded. The **transformdb** tool avoids this issue using an interpretive approach: the Slice definitions are parsed and used to drive the extraction of the database records.

The tool supports two modes of operation:

1. automatic migration, in which the database is migrated in a single step using only the default set of transformations, and
2. custom migration, in which you supply a script to augment or override the default transformations.

22.3.1 Default Transformations

The default transformations performed by **transformdb** preserve as much information as possible. However, there are practical limits to the tool’s capabilities, since the only information it has is obtained by performing a comparison of the Slice definitions.

For example, suppose our old definition for a structure is the following:

```
struct AStruct {  
    int i;  
};
```

We want to migrate instances of this struct to the following revised definition:

```
struct AStruct {  
    int j;  
};
```

As the developers, we know that the `int` member has been renamed from `i` to `j`, but to **transformdb** it appears that member `i` was removed and member `j` was added. The default transformation results in exactly that behavior: the value of `i` is lost, and `j` is initialized to a default value. If we need to preserve the value of `i` and transfer it to `j`, then we need to use custom migration (see Section 22.3.5).

The changes that occur as a type system evolves can be grouped into three categories:

- Data members

The data members of class and structure types are added, removed, or renamed. As discussed above, the default transformations initialize new and renamed data members to default values (see Section 22.3.3).

- Type names

Types are added, removed, or renamed. New types do not pose a problem for database migration when used to define a new data member; the member is initialized with default values as usual. On the other hand, if the new type replaces the type of an existing data member, then type compatibility becomes a factor (see the following item).

Removed types generally do not cause problems either, because any uses of that type must have been removed from the new Slice definitions (e.g., by removing data members of that type). There is one case, however, where removed types become an issue, and that is for polymorphic classes (see Section 22.5.5).

Renamed types are a concern, just like renamed data members, because of the potential for losing information during migration. This is another situation for which custom migration is recommended.

- Type content

Examples of changes of type content include the key type of a dictionary, the element type of a sequence, or the type of a data member. If the old and new types are not compatible (as defined in Section 22.3.2), then the default trans-

formation emits a warning, discards the current value, and reinitializes the value as described in Section 22.3.3.

22.3.2 Type Compatibility

Changes in the type of a value are restricted to certain sets of compatible changes. This section describes the type changes supported by the default transformations. All incompatible type changes result in a warning indicating that the current value is being discarded and a default value for the new type assigned in its place. Additional flexibility is provided by custom migration, as described in Section 22.3.5.

Boolean

A value of type `bool` can be transformed to and from `string`. The legal string values for a `bool` value are `"true"` and `"false"`.

Integer

The integer types `byte`, `short`, `int`, and `long` can be transformed into each other, but only if the current value is within range of the new type. These integer types can also be transformed into `string`.

Floating Point

The floating-point types `float` and `double` can be transformed into each other, as well as to `string`. No attempt is made to detect a loss of precision during transformation.

String

A `string` value can be transformed into any of the primitive types, as well as into enumeration and proxy types, but only if the value is a legal string representation of the new type. For example, the string value `"Pear"` can be transformed into the enumeration `Fruit`, but only if `Pear` is an enumerator of `Fruit`.

Enum

An enumeration can be transformed into an enumeration with the same type id, or into a string. Transformation between enumerations is performed symbolically. For example, consider our old type below:

```
enum Fruit { Apple, Orange, Pear };
```

Suppose the enumerator `Pear` is being transformed into the following new type:

```
enum Fruit { Apple, Pear };
```

The transformed value in the new enumeration is also `Pear`, despite the fact that `Pear` has changed positions in the new type. However, if the old value had been `Orange`, then the default transformation emits a warning because that enumerator no longer exists, and initializes the new value to `Apple` (the default value).

If an enumerator has been renamed, then custom migration is required to convert enumerators from the old name to the new one.

Sequence

A sequence can be transformed into another sequence type, even if the new sequence type does not have the same type id as the old type, but only if the element types are compatible. For example, `sequence<short>` can be transformed into `sequence<int>`, regardless of the names given to the sequence types.

Dictionary

A dictionary can be transformed into another dictionary type, even if the new dictionary type does not have the same type id as the old type, but only if the key and value types are compatible. For example, `dictionary<int, string>` can be transformed into `dictionary<long, string>`, regardless of the names given to the dictionary types.

Caution is required when changing the key type of a dictionary, because the default transformation of keys could result in duplication. For example, if the key type changes from `int` to `short`, any `int` value outside the range of `short` results in the key being initialized to a default value (namely zero). If zero is already used as a key in the dictionary, or another out-of-range key is encountered, then a duplication occurs. The transformation handles key duplication by removing the duplicate element from the transformed dictionary. (Custom migration can be useful in these situations if the default behavior is not acceptable.)

Structure

A struct type can only be transformed into another struct type with the same type id. Data members are transformed as appropriate for their types.

Proxy

A proxy value can be transformed into another proxy type, or into `string`. Transformation into another proxy type is done with the same semantics as in a

language mapping: if the new type does not match the old type, then the new type must be a base type of the old type (that is, the proxy is widened).

Class

A class type can only be transformed into another class type with the same type id. A data member of a class type is allowed to be widened to a base type. Data members are transformed as appropriate for their types. See Section 22.5.5 for more information on transforming classes.

22.3.3 Default Values

Data types are initialized with default values, as shown in Table 22.1.

Table 22.1. Default values for Slice types.

Type	Default Value
Boolean	false
Numeric	Zero (0)
String	Empty string
Enumeration	The first enumerator
Sequence	Empty sequence
Dictionary	Empty dictionary
Struct	Data members are initialized recursively
Proxy	Nil
Class	Nil

22.3.4 Running an Automatic Transformation

In order to use automatic transformation, we need to supply the following information to **transformdb**:

- The old and new Slice definitions
- The old and new types for the database key and value
- The database environment directory, the database filename, and the name of a new database environment directory to hold the transformed database

Here is an example of a **transformdb** command:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice \
  --key int,string --value ::Employee db emp.db newdb
```

Briefly, the `--old` and `--new` options specify the old and new Slice definitions, respectively. These options can be specified as many times as necessary in order to load all of the relevant definitions. The `--key` option indicates that the database key is evolving from `int` to `string`. The `--value` option specifies that `::Employee` is used as the database value type in both old and new type definitions, and therefore only needs to be specified once. Finally, we provide the pathname of the database environment directory (`db`), the filename of the database (`emp.db`), and the pathname of the database environment directory for the transformed database (`newdb`).

See Section 22.5 for more information on using `transformdb`.

22.3.5 Custom Migration

Custom migration is useful when your types have changed in ways that make automatic migration difficult or impossible. It is also convenient to use custom migration when you have complex initialization requirements for new types or new data members, because custom migration enables you to perform many of the same tasks that would otherwise require you to write a throwaway program.

Custom migration operates in conjunction with automatic migration, allowing you to inject your own transformation rules at well-defined intercept points in the automatic migration process. These rules are called *transformation descriptors*, and are written in XML.

A Simple Example

We can use a simple example to demonstrate the utility of custom migration. Suppose our application uses a Freeze map whose type is `string` and whose value is an enumeration, defined as follows:

```
enum BigThree { Ford, Chrysler, GeneralMotors };
```

We now wish to rename the enumerator `Chrysler`, as shown in our new definition:

```
enum BigThree { Ford, Dai ml erChrysler, GeneralMotors };
```

As explained in Section 22.3.2, the default transformation results in all occurrences of the `Chrysler` enumerator being transformed into `Ford`, because

Chrysler no longer exists in the new definition and therefore the default value Ford is used instead.

To remedy this situation, we use the following transformation descriptors:

```
<transformdb>
  <database key="string" value="::BigThree">
    <record>
      <if test="oldvalue == ::Old::Chrysler">
        <set target="newvalue"
          value="::New::DaimlerChrysler"/>
      </if>
    </record>
  </database>
</transformdb>
```

When executed, these descriptors convert occurrences of Chrysler in the old type system into DaimlerChrysler in the transformed database's new type system. Transformation descriptors are described in detail in Section 22.4.

22.4 Transformation Descriptors

This section describes the XML elements comprising the FreezeScript transformation descriptors.

22.4.1 Overview

A transformation descriptor file has a well-defined structure. The top-level descriptor in the file is `<transformdb>`. A `<database>` descriptor must be present within `<transformdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers the transformation process. See Section 22.3.5 for an example that demonstrates the structure of a minimal descriptor file.

During transformation, type-specific actions are supported by the `<transform>` and `<init>` descriptors, both of which are children of `<transformdb>`. One `<transform>` descriptor and one `<init>` descriptor may be defined for each type in the new Slice definitions. Each time **transformdb** creates a new instance of a type, it executes the `<init>` descriptor for that type, if one is defined. Similarly, each time **transformdb** transforms an instance of an old type into a new type, the `<transform>` descriptor for the new type is executed.

The `<database>`, `<record>`, `<transform>`, and `<init>` descriptors may contain general-purpose action descriptors such as `<if>`, `<set>`, and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 22.8.

22.4.2 Flow of Execution

The transformation descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present, database transformation occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until transformation completes.
- During transformation of each record, **transformdb** creates instances of the new key and value types, which includes the execution of the `<init>` descriptors for those types. Next, the old key and value are transformed into the new key and value, in the following manner:
 1. Locate the `<transform>` descriptor for the type.
 2. If no descriptor is found, or the descriptor exists and it does not preclude default transformation, then transform the data as described in Section 22.3.1.
 3. If the `<transform>` descriptor exists, execute it.
 4. Finally, execute the child descriptors of `<record>`.

See Section 22.4.4 for detailed information on the transformation descriptors.

22.4.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor¹. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<transform>` descriptor creates

1. In order for a global symbol to be available to a `<transform>` or `<init>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

a local scope and defines the symbols `old` and `new` to represent a value in its old and new forms. Child descriptors of `<transform>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during transformation. This can be accomplished as shown below:

```
<transformdb>
  <database key="string" value="::Ice::Identity">
    <define name="categoryCount" type="int" value="0"/>
    <record/>
    <echo message="categoryCount = " value="categoryCount"/>
  </database>
  <transform type="::Ice::Identity">
    <if test="new.category == 'Accounting'">
      <set target="categoryCount" value="categoryCount + 1"/>
    </if>
  </transform>
</transformdb>
```

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes transformation to proceed. Each occurrence of the type `Ice::Identity` causes its `<transform>` descriptor to be executed, which examines the category member and increases `categoryCount` if necessary. Finally, after transformation completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 22.1. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<init>` descriptor's scope². In

2. This situation can be avoided by assigning a different symbol name to the element value.

addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<init>` and `<database>` scopes.

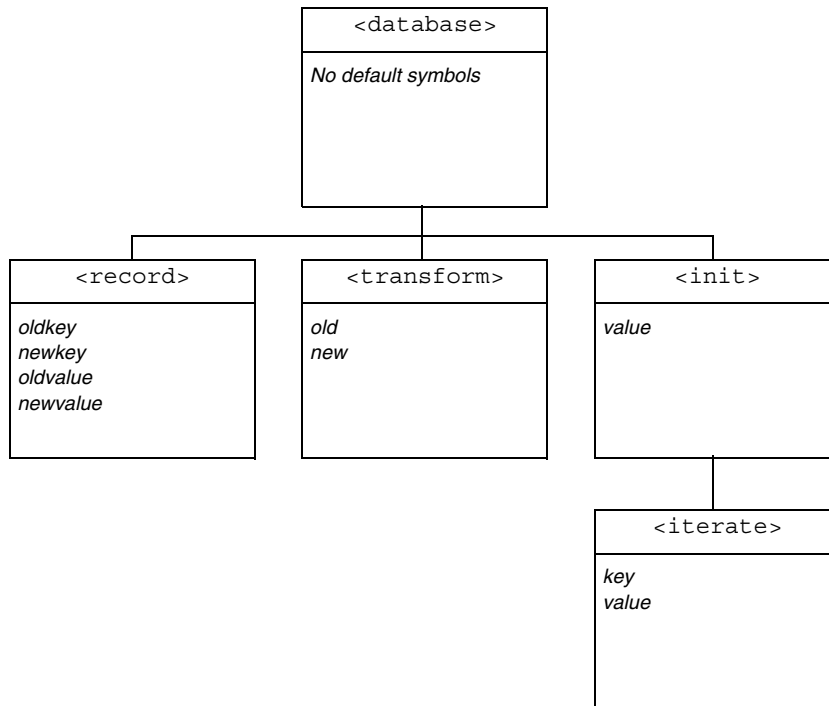


Figure 22.1. Relationship between descriptors and scopes.

22.4.4 Descriptor Reference

`<transformdb>`

The top-level descriptor in a descriptor file. It requires one child descriptor, `<database>`, and supports any number of `<transform>` and `<init>` descriptors. This descriptor has no attributes.

`<database>`

The attributes of this descriptor define the old and new key and value types for the database to be transformed. It supports any number of child descriptors, but at

most one `<record>` descriptor. The `<database>` descriptor also creates a global scope for user-defined symbols (see Section 22.4.3).

The attributes supported by the `<database>` descriptor are described in Table 22.2.

Table 22.2. Attributes for `<database>` descriptor.

Name	Description
key	Specifies the Slice types of the old and new keys. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.
value	Specifies the Slice types of the old and new values. If the types are the same, only one needs to be specified. Otherwise, the types are separated by a comma.

As an example, consider the following `<database>` descriptor. In this case, the Freeze map to be transformed currently has key type `int` and value type `::Employee`, and is migrating to a key type of `string`:

```
<database key="int,string" value="::Employee">
```

`<record>`

Commences the transformation. Child descriptors are executed for each record in the database, providing the user with an opportunity to examine the record's old key and value, and optionally modify the new key and value. Default transformations, as well as `<transform>` and `<init>` descriptors, are executed before the child descriptors. The `<record>` descriptor introduces the following symbols into a local scope: `oldkey`, `newkey`, `oldvalue`, `newvalue`. These symbols are accessible to child descriptors, but not to `<transform>` or `<init>` descriptors. The `oldkey` and `oldvalue` symbols are read-only.

Use caution when modifying database keys to ensure that duplicate keys do not occur. If a duplicate database key is encountered, transformation fails immediately.

Note that database transformation only occurs if a `<record>` descriptor is present.

<transform>

Customizes the transformation for all instances of a type in the new Slice definitions. The children of this descriptor are executed after the optional default transformation has been performed, as described in Section 22.3.1. Only one **<transform>** descriptor can be specified for a type, but a **<transform>** descriptor is not required for every type. The symbols **old** and **new** are introduced into a local scope and represent the old and new values, respectively. The **old** symbol is read-only. The attributes supported by this descriptor are described in Table 22.3.

Table 22.3. Attributes for **<transform>** descriptor.

Name	Description
type	Specifies the Slice type id.
default	If false , no default transformation is performed on values of this type. If not specified, the default value is true .
base	This attribute determines whether <transform> descriptors of base class types are executed. If true , the <transform> descriptor of the immediate base class is invoked. If no descriptor is found for the immediate base class, the class hierarchy is searched until a descriptor is found. The execution of any base class descriptors occurs after execution of this descriptor's children. If not specified, the default value is true .
rename	Indicates that a type in the old Slice definitions has been renamed to the new type identified by the type attribute. The value of this attribute is the type id of the old Slice definition. Specifying this attribute relaxes the strict compatibility rules defined in Section 22.3.2 for enum , struct and class types.

Below is an example of a **<transform>** descriptor that initializes a new data member:

```
<transform type="::Product">
  <set target="new.salePrice"
    value="old.listPrice * old.discount"/>
</transform>
```

For class types, **transformdb** first attempts to locate a **<transform>** descriptor for the object's most-derived type. If no descriptor is found,

transformdb proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<transform>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<transform>` descriptors are executed recursively. For example, consider the following Slice definitions:

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

When **transformdb** is performing the default transformation on a value of type `Outer`, it recursively performs the default transformation on the `Inner` member, then executes the `<transform>` descriptor for `Inner`, and finally executes the `<transform>` descriptor for `Outer`. However, if default transformation is disabled for `Outer`, then no transformation is performed on the `Inner` member and therefore the `<transform>` descriptor for `Inner` is not executed.

<init>

Defines custom initialization rules for all instances of a type. Child descriptors are executed each time the type is instantiated. The typical use case for this descriptor is for types that have been introduced in the new Slice definitions and whose instances require default values different than what **transformdb** supplies. The symbol `value` is introduced into a local scope to represent the instance. The attributes supported by this descriptor are described in Table 22.4.

Table 22.4. Attributes for `<init>` descriptor.

Name	Description
<code>type</code>	Specifies the Slice type id.

Here is a simple example of an `<init>` descriptor:

```
<init type="::Player">
    <set target="value.currency" value="100"/>
</init>
```

Note that, like `<transform>`, `<init>` descriptors are executed recursively. For example, if an `<init>` descriptor is defined for a struct type, the `<init>` descriptors of the struct's members are executed before the struct's descriptor.

`<iterate>`

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 22.5.

Table 22.5. Attributes for `<iterate>` descriptor.

Name	Description
<code>target</code>	The sequence or dictionary.
<code>index</code>	The symbol name used for the sequence index. If not specified, the default symbol is <code>i</code> .
<code>element</code>	The symbol name used for the sequence element. If not specified, the default symbol is <code>elem</code> .
<code>key</code>	The symbol name used for the dictionary key. If not specified, the default symbol is <code>key</code> .
<code>value</code>	The symbol name used for the dictionary value. If not specified, the default symbol is <code>value</code> .

Shown below is an example of an `<iterate>` descriptor that sets the new data member `reviewSalary` to `true` if the employee's salary is greater than \$3000.

```
<iterate target="new.employeeMap" key="id" value="emp">
  <if test="emp.salary > 3000">
    <set target="emp.reviewSalary" value="true"/>
  </if>
</iterate>
```

<if>

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 22.6.

Table 22.6. Attributes for `<if>` descriptor.

Name	Description
<code>test</code>	A boolean expression.

See Section 22.8 for more information on the descriptor expression language.

<set>

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., `<set>` cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 22.7.

Table 22.7. Attributes for `<set>` descriptor.

Name	Description
<code>target</code>	An expression that must select a modifiable value.
<code>value</code>	An expression that must evaluate to a value compatible with the target's type.
<code>type</code>	The Slice type id of a class to be instantiated. The class must be compatible with the target's type.
<code>length</code>	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If <code>value</code> or <code>type</code> is also specified, it is used to initialize each new element.
<code>convert</code>	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
    value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
    value="'P105J3'"/>
```

<add>

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The key and index attributes are mutually exclusive, as are the value and type attributes. If neither value nor type is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 22.8.

Table 22.8. Attributes for `<add>` descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the target dictionary's key type.
index	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before index. The value must not exceed the length of the target sequence.
value	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
convert	If true, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is false.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="'P105J4'"/>
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

<define>

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 22.9.

Table 22.9. Attributes for `<define>` descriptor.

Name	Description
name	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
type	The name of the symbol's formal Slice type. For user-defined types, the name should be prefixed with <code>:Old</code> or <code>:New</code> to indicate the source of the type. The prefix can be omitted for primitive types.
value	An expression that must evaluate to a value compatible with the symbol's type.
convert	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

```
<define name="identity" type=":New::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in Section 22.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type=":New::BigThree"
value=":New::DaimlerChrysler"/>
```

<remove>

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 22.10.

Table 22.10. Attributes for `<remove>` descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the key type of the target dictionary.
index	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

<fail>

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in Table 22.11.

Table 22.11. Attributes for `<fail>` descriptor.

Name	Description
message	A message to display upon transformation failure.
test	A boolean expression.

The following `<fail>` descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"  
      test="old.ticketCount > 32767"/>
```

<delete>

Causes transformation of the current database record to cease, and removes the record from the transformed database. This descriptor has no attributes.

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 22.12.

Table 22.12. Attributes for <echo> descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an <echo> descriptor that uses both message and value attributes:

```
<if test="old.ticketCount > 32767">
  <echo message="deleting record with invalid ticket count: "
    value="old.ticketCount"/>
  <delete/>
</if>
```

22.4.5 Descriptor Guidelines

There are three points at which you can intercept the transformation process: when transforming a record (<record>), when transforming an instance of a type (<transform>), and when creating an instance of a type (<init>).

In general, <record> is used when your modifications require access to both the key and value of the record. For example, if the database key is needed as a factor in an equation, or to identify an element in a dictionary, then <record> is the only descriptor in which this type of modification is possible. The <record> descriptor is also convenient to use when the number of changes to be made is small, and does not warrant the effort of writing separate <transform> or <init> descriptors.

The `<transform>` descriptor has a more limited scope than `<record>`. It is used when changes must potentially be made to all instances of a type (regardless of the context in which that type is used) and access to the old value is necessary. The `<transform>` descriptor does not have access to the database key and value, therefore decisions can only be made based on the old and new instances of the type in question.

Finally, the `<init>` descriptor is useful when access to the old instance is not required in order to properly initialize a type. In most cases, this activity could also be performed by a `<transform>` descriptor that simply ignored the old instance, so `<init>` may seem redundant. However, there is one situation where `<init>` is required: when it is necessary to initialize an instance of a type that is introduced by the new Slice definitions. Since there are no instances of this type in the current database, a `<transform>` descriptor for that type would never be executed.

22.5 Using `transformdb`

This section describes the invocation of the **`transformdb`** tool, and provides advice on how best to use it. The tool supports the standard command-line options common to all Slice processors listed in Section 4.18, with the exception of the include directory (`-I`) option. The options specific to **`transformdb`** are described in the subsections below.

22.5.1 General Options

The following options are used in both automatic and custom migration:

- `--old SLICE`
`--new SLICE`

Loads the old or new Slice definitions contained in the file *SLICE*. These options may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice containing only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- **--include-old *DIR***
--include-new *DIR*

Adds the directory *DIR* to the set of include paths for the old or new Slice definitions.

- **--key *TYPE* [, *TYPE*]**
--value *TYPE* [, *TYPE*]

Specifies the Slice type(s) of the database key and value. If the type does not change, then the type only needs to be specified once. Otherwise, the old type is specified first, followed by a comma and the new type. For example, the option **--key *int*, *string*** indicates that the database key is migrating from *int* to *string*. On the other hand, the option **--key *int*, *int*** indicates that the key type does not change, and could be given simply as **--key *int***. Type changes are restricted to those allowed by the compatibility rules defined in Section 22.3.2, but custom migration provides additional flexibility.

- **-e**

Indicates that a Freeze evictor database is being migrated. This option is provided as a convenience, in that it automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **--key** and **--value** options are not necessary. Specifically, the key type of a Freeze evictor database is `Freeze: : EvictorStorageKey`, and the value type is `Freeze: : ObjectRecord`. These types are defined in the Slice file `Freeze/EvictorStorage.ice`; however, this file does not need to be loaded into your old and new Slice definitions.

- **-i**

Requests that **transformdb** ignore type changes that violate the compatibility rules defined in Section 22.3.2. If this option is not specified, transformation fails immediately if such a violation occurs. With this option, a warning is displayed but transformation continues.

- **-p**

Requests that **transformdb** purge object instances whose type is no longer found in the new Slice definitions. See Section 22.5.5 for more information.

- **-c**

Use catastrophic recovery on the old BerkeleyDB database environment.

- **-w**
Suppress duplicate warnings during transformation.

22.5.2 Database Arguments

If **transformdb** is invoked to transform a database, it requires three arguments:

- **dbenv**
The pathname of the database environment directory.
- **db**
The name of the existing database file. **transformdb** never modifies this database.
- **newdbenv**
The pathname of the database environment directory to contain the transformed database. This directory must exist, and must not contain an existing database whose name matches the **db** argument.

Upon successful transformation, a database of the same name is created in the new environment directory.

22.5.3 Automatic Migration

No additional arguments are necessary to use automatic migration. The standard options from Section 4.18, the general options described in Section 22.5.1, and the database arguments from Section 22.5.2 are all you need. For example, consider the following command, which uses automatic migration to transform a database with a key type of `int` and value type of `string` into a database with the same key type and a value type of `long`:

```
$ transformdb --key int --value string,long dbhome data.db newdbhome
```

Note that we did not need to specify the **--old** or **--new** options because our key and value types are primitives. Upon successful completion, the file **newdbhome/data.db** contains our transformed database.

22.5.4 Custom Migration

Analysis

Custom migration is a two-step process: your first write the transformation descriptors, and then execute them to transform a database. To assist you in the process of creating a descriptor file, **transformdb** can generate a default set of transformation descriptors by comparing your old and new Slice definitions. This feature is called *analysis*, and is enabled by specifying the following option:

- **-o FILE**

Specifies the descriptor file **FILE** to be created during analysis.

No database arguments are required, because transformation does not occur when the **-o** option is specified. The generated file contains a `<transform>` descriptor for each type that appears in both old and new Slice definitions, and an `<init>` descriptor for types that appear only in the new Slice definitions. In most cases, these descriptors are empty. However, they may contain XML comments describing changes detected by **transformdb** that may require action on your part.

For example, let us revisit the enumeration we defined in Section 22.3.5:

```
enum Bi gThree { Ford, Chrysl er, General Motors };
```

This enumeration has evolved into the one shown below. In particular, the `Chrysl er` enumerator has been renamed to reflect a corporate merger:

```
enum Bi gThree { Ford, Dai ml erChrysl er, General Motors };
```

Next we run **transformdb** in analysis mode:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
  --key string --value ::BigThree -o transform.xml
```

The generated file **transform.xml** contains the following descriptor for the enumeration `Bi gThree`:

```
<transform type="::BigThree">
  <!-- NOTICE: enumerator `Chrysler' has been removed -->
</transform>
```

The comment indicates that enumerator `Chrysl er` is no longer present in the new definition, reminding us that we need to add logic in this `<transform>` descriptor to change all occurrences of `Chrysl er` to `Dai ml erChrysl er`.

The descriptor file generated by `transformdb` is well-formed and does not require any manual intervention prior to being executed. However, executing an unmodified descriptor file is simply the equivalent of automatic migration.

Transformation

After preparing a descriptor file, either by writing one completely yourself, or modifying one generated by the analysis phase, you are ready to transform a database. One additional option is provided for transformation:

- **-f *FILE***

Specifies the descriptor file *FILE* to be executed during transformation.

It is not necessary to provide the **-e**, **--key** or **--value** options during transformation, because the key and value types are already specified in the file's `<database>` descriptor.

Continuing our enumeration example from the analysis discussion above, assume we have modified `transform.xml` to convert the Chrysler enumerator, and are now ready to execute the transformation:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
  -f transform.xml dbhome bigthree.db newbigthree.db
```

Strategies

If it becomes necessary for you to transform a Freeze database, we generally recommend that you attempt to use automatic migration first, unless you already know that custom migration is necessary. Since transformation is a non-destructive process, there is no harm in attempting an automatic migration, and it is a good way to perform a sanity check on your `transformdb` arguments (for example, to ensure that all the necessary Slice files are being loaded), as well as on the database itself. If `transformdb` detects any incompatible type changes, it displays an error message for each incompatible change and terminates without doing any transformation. In this case, you may want to run `transformdb` again with the **-i** option, which ignores incompatible changes and causes transformation to proceed.

Pay careful attention to any warnings that `transformdb` emits, as these may indicate the need for using custom migration. For example, if we had attempted to transform the database containing the `BigThree` enumeration from previous sections using automatic migration, any occurrences of the Chrysler enumerator would display the following warning:

```
warning: unable to convert 'Chrysler' to ::BigThree
```

If custom migration appears to be necessary, use analysis to generate a default descriptor file, then review it for NOTICE comments and edit as necessary. Liberal use of the `<echo>` descriptor can be beneficial when testing your descriptor file, especially from within the `<record>` descriptor where you can display old and new keys and values.

22.5.5 Transforming Objects

The polymorphic nature of Slice classes can cause problems for database migration. As an example, the Slice parser can ensure that a set of Slice definitions loaded into **transformdb** is complete for all types but classes (and exceptions, but we ignore those because they are not persistent). **transformdb** cannot know that a database may contain instances of a subclass derived from one of the loaded classes, but itself is not loaded. Alternatively, the type of a class instance may have been renamed and cannot be found in the new Slice definitions.

By default, these situations result in immediate transformation failure. However, the **-p** option is a (potentially drastic) way to handle these situations: if a class instance has no equivalent in the new Slice definitions and this option is specified, **transformdb** removes the instance any way it can. If the instance appears in a sequence or dictionary element, that element is removed. Otherwise, the database record containing the instance is deleted.

Now, the case of a class type being renamed is handled easily enough using custom migration and the `rename` attribute of the `<transform>` descriptor. However, there are legitimate cases where the destructive nature of the **-p** option can be useful. For example, if a class type has been removed and it is simply easier to start with a database that is guaranteed not to contain any instances of that type, then the **-p** option may simplify the broader migration effort.

This is another situation in which running an automatic migration first can help point out the trouble spots in a potential migration. Using the **-p** option, **transformdb** emits a warning about the missing class type and continues, rather than halting at the first occurrence, enabling you to discover whether you have forgotten to load some Slice definitions, or need to rename a type.

22.6 Database Inspection

The FreezeScript tool **dumpdb** is used to examine a Freeze database. Its simplest invocation displays every record of the database, but the tool also supports more

selective activities. In fact, **dumpdb** supports a scripted mode that shares many of the same XML descriptors as **transformdb** (see Section 22.4), enabling sophisticated filtering and reporting.

22.6.1 Descriptor Overview

A **dumpdb** descriptor file has a well-defined structure. The top-level descriptor in the file is `<dumpdb>`. A `<database>` descriptor must be present within `<dumpdb>` to define the key and value types used by the database. Inside `<database>`, the `<record>` descriptor triggers database traversal. Shown below is an example that demonstrates the structure of a minimal descriptor file:

```
<dumpdb>
  <database key="string" value="::Employee">
    <record>
      <echo message="Key: " value="key"/>
      <echo message="Value: " value="value"/>
    </record>
  </database>
</dumpdb>
```

During traversal, type-specific actions are supported by the `<dump>` descriptor, which is a child of `<dumpdb>`. One `<dump>` descriptor may be defined for each type in the new Slice definitions. Each time **dumpdb** encounters an instance of a type, the `<dump>` descriptor for the type is executed.

The `<database>`, `<record>`, and `<dump>` descriptors may contain general-purpose action descriptors such as `<if>` and `<echo>`. These actions resemble statements in programming languages like C++ and Java, in that they are executed in the order of definition and their effects are cumulative. Actions make use of the expression language described in Section 22.8.

Although **dumpdb** descriptors are not allowed to modify the database, they can still define local symbols for scripting purposes. Once a symbol is defined by the `<define>` descriptor, other descriptors such as `<set>`, `<add>`, and `<remove>` can be used to manipulate the symbol's value.

22.6.2 Flow of Execution

The descriptors are executed as described below.

- `<database>` is executed first. Each child descriptor of `<database>` is executed in the order of definition. If a `<record>` descriptor is present,

database traversal occurs at that point. Any child descriptors of `<database>` that follow `<record>` are not executed until traversal completes.

- For each record, **dumpdb** interprets the key and value, invoking `<dump>` descriptors for each type it encounters. For example, if the value type of the database is a struct, then **dumpdb** first attempts to invoke a `<dump>` descriptor for the struct type, and then recursively interprets the structure's members in the same fashion.

See Section 22.6.4 for detailed information on the **dumpdb** descriptors.

22.6.3 Scopes

The `<database>` descriptor creates a global scope, allowing child descriptors of `<database>` to define symbols that are accessible in any descriptor³. Furthermore, certain other descriptors create local scopes that exist only for the duration of the descriptor's execution. For example, the `<dump>` descriptor creates a local scope and defines the symbol `value` to represent a value of the specified type. Child descriptors of `<dump>` can also define new symbols in the local scope, as long as those symbols do not clash with an existing symbol in that scope. It is legal to add a new symbol with the same name as a symbol in an outer scope, but the outer symbol will not be accessible during the descriptor's execution.

The global scope is useful in many situations. For example, suppose you want to track the number of times a certain value was encountered during database traversal. This can be accomplished as shown below:

```
<dumpdb>
  <database key="string" value="::Ice::Identity">
    <define name="categoryCount" type="int" value="0"/>
    <record/>
    <echo message="categoryCount = " value="categoryCount"/>
  </database>
  <dump type="::Ice::Identity">
    <if test="new.category == 'Accounting'">
      <set target="categoryCount" value="categoryCount + 1"/>
    </if>
  </dump>
</dumpdb>
```

3. In order for a global symbol to be available to a `<dump>` descriptor, the symbol must be defined before the `<record>` descriptor is executed.

In this example, the `<define>` descriptor introduces the symbol `categoryCount` into the global scope, defining it as type `int` with an initial value of zero. Next, the `<record>` descriptor causes traversal to proceed. Each occurrence of the type `Ice::Identity` causes its `<dump>` descriptor to be executed, which examines the category member and increases `categoryCount` if necessary. Finally, after traversal completes, the `<echo>` descriptor displays the final value of `categoryCount`.

To reinforce the relationships between descriptors and scopes, consider the diagram in Figure 22.2. Several descriptors are shown, including the symbols they define in their local scopes. In this example, the `<iterate>` descriptor has a dictionary target and therefore the default symbol for the element value, `value`, hides the symbol of the same name in the parent `<dump>` descriptor's scope⁴. In addition to symbols in the `<iterate>` scope, child descriptors of `<iterate>` can also refer to symbols from the `<dump>` and `<database>` scopes.

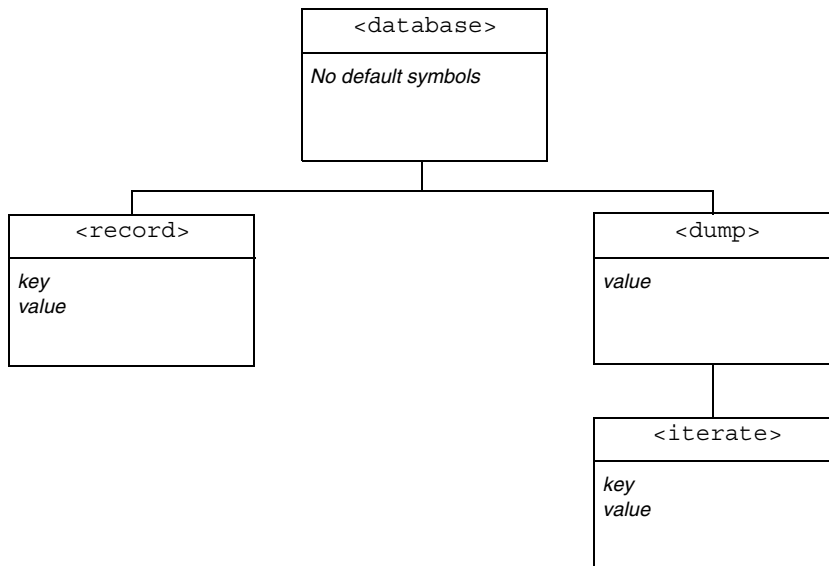


Figure 22.2. Relationship between descriptors and scopes.

4. This situation can be avoided by assigning a different symbol name to the element value.

22.6.4 Descriptor Reference

<dumpdb>

The top-level descriptor in a descriptor file. It requires one child descriptor, <database>, and supports any number of <dump> descriptors. This descriptor has no attributes.

<database>

The attributes of this descriptor define the key and value types of the database. It supports any number of child descriptors, but at most one <record> descriptor. The <database> descriptor also creates a global scope for user-defined symbols (see Section 22.6.3).

The attributes supported by the <database> descriptor are described in Table 22.13.

Table 22.13. Attributes for <database> descriptor.

Name	Description
key	Specifies the Slice type of the database key.
value	Specifies the Slice type of the database value.

As an example, consider the following <database> descriptor. In this case, the Freeze map to be examined has key type `int` and value type `::Employee`:

```
<database key="int" value="::Employee">
```

<record>

Commences the database traversal. Child descriptors are executed for each record in the database, but after any <dump> descriptors are executed. The <record> descriptor introduces the read-only symbols `key` and `value` into a local scope. These symbols are accessible to child descriptors, but not to <dump> descriptors.

Note that database traversal only occurs if a <record> descriptor is present.

<dump>

Executed for all instances of a Slice type. Only one <dump> descriptor can be specified for a type, but a <dump> descriptor is not required for every type. The

read-only symbol `value` is introduced into a local scope. The attributes supported by this descriptor are described in Table 22.14.

Table 22.14. Attributes for `<dump>` descriptor.

Name	Description
<code>type</code>	Specifies the Slice type id.
<code>base</code>	If <code>type</code> denotes a Slice class, this attribute determines whether the <code><dump></code> descriptor of the base class is invoked. If <code>true</code> , the base class descriptor is invoked after executing the child descriptors. If not specified, the default value is <code>true</code> .
<code>contents</code>	For <code>class</code> and <code>struct</code> types, this attribute determines whether descriptors are executed for members of the value. For <code>sequence</code> and <code>dictionary</code> types, this attribute determines whether descriptors are executed for elements. If not specified, the default value is <code>true</code> .

Below is an example of a `<dump>` descriptor that searches for certain products:

```
<dump type="::Product">
  <if test="value.description.find('scanner') != -1">
    <echo message="Scanner SKU: " value="value.SKU"/>
  </if>
</dump>
```

For class types, **dumpdb** first attempts to locate a `<dump>` descriptor for the object's most-derived type. If no descriptor is found, **dumpdb** proceeds up the class hierarchy in an attempt to find a descriptor. The base object type, `Object`, is the root of every class hierarchy and is included in the search for descriptors. It is therefore possible to define a `<dump>` descriptor for type `Object`, which will be invoked for every class instance.

Note that `<dump>` descriptors are executed recursively. For example, consider the following Slice definitions:

```
struct Inner {
  int sum;
};
struct Outer {
  Inner i;
};
```

When **dumpdb** is interpreting a value of type **Outer**, it executes the `<dump>` descriptor for **Outer**, then recursively executes the `<dump>` descriptor for the **Inner** member, but only if the `contents` attribute of the **Outer** descriptor has the value `true`.

<iterate>

Iterates over a dictionary or sequence, executing child descriptors for each element. The symbol names selected to represent the element information may conflict with existing symbols in the enclosing scope, in which case those outer symbols are not accessible to child descriptors. The attributes supported by this descriptor are described in Table 22.15.

Table 22.15. Attributes for `<iterate>` descriptor.

Name	Description
<code>target</code>	The sequence or dictionary.
<code>index</code>	The symbol name used for the sequence index. If not specified, the default symbol is <code>i</code> .
<code>element</code>	The symbol name used for the sequence element. If not specified, the default symbol is <code>elem</code> .
<code>key</code>	The symbol name used for the dictionary key. If not specified, the default symbol is <code>key</code> .
<code>value</code>	The symbol name used for the dictionary value. If not specified, the default symbol is <code>value</code> .

Shown below is an example of an `<iterate>` descriptor that displays the name of an employee if the employee's salary is greater than \$3000.

```
<iterate target="value.employeeMap" key="id" value="emp">
  <if test="emp.salary > 3000">
    <echo message="Employee: " value="emp.name"/>
  </if>
</iterate>
```


<if>

Conditionally executes child descriptors. The attributes supported by this descriptor are described in Table 22.16.

Table 22.16. Attributes for <if> descriptor.

Name	Description
test	A boolean expression.

See Section 22.8 for more information on the descriptor expression language.

<set>

Modifies a value. The `value` and `type` attributes are mutually exclusive. If `target` denotes a dictionary element, that element must already exist (i.e., <set> cannot be used to add an element to a dictionary). The attributes supported by this descriptor are described in Table 22.17.

Table 22.17. Attributes for <set> descriptor.

Name	Description
target	An expression that must select a modifiable value.
value	An expression that must evaluate to a value compatible with the target's type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target's type.
length	An integer expression representing the desired new length of a sequence. If the new length is less than the current size of the sequence, elements are removed from the end of the sequence. If the new length is greater than the current size, new elements are added to the end of the sequence. If <code>value</code> or <code>type</code> is also specified, it is used to initialize each new element.
convert	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

The `<set>` descriptor below modifies a member of a dictionary element:

```
<set target="new.parts['P105J3'].cost"
    value="new.parts['P105J3'].cost * 1.05"/>
```

This `<set>` descriptor adds an element to a sequence and initializes its value:

```
<set target="new.partsList" length="new.partsList.length + 1"
    value="'P105J3'"/>
```

<add>

Adds a new element to a sequence or dictionary. It is legal to add an element while traversing the sequence or dictionary using `<iterate>`, however the traversal order after the addition is undefined. The key and index attributes are mutually exclusive, as are the value and type attributes. If neither value nor type is specified, the new element is initialized with a default value. The attributes supported by this descriptor are described in Table 22.18.

Table 22.18. Attributes for `<add>` descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the target dictionary's key type.
index	An expression that must evaluate to an integer value representing the insertion position. The new element is inserted before index. The value must not exceed the length of the target sequence.
value	An expression that must evaluate to a value compatible with the target dictionary's value type, or the target sequence's element type.
type	The Slice type id of a class to be instantiated. The class must be compatible with the target dictionary's value type, or the target sequence's element type.
convert	If true, additional type conversions are supported: between integer and floating point, and between integer and enumeration. Transformation fails immediately if a range error occurs. If not specified, the default value is false.

Below is an example of an `<add>` descriptor that adds a new dictionary element and then initializes its member:

```
<add target="new.parts" key="'P105J4'" />
<set target="new.parts['P105J4'].cost" value="3.15"/>
```

<define>

Defines a new symbol in the current scope. The attributes supported by this descriptor are described in Table 22.19.

Table 22.19. Attributes for `<define>` descriptor.

Name	Description
name	The name of the new symbol. An error occurs if the name matches an existing symbol in the current scope.
type	The name of the symbol's formal Slice type.
value	An expression that must evaluate to a value compatible with the symbol's type.
convert	If <code>true</code> , additional type conversions are supported: between integer and floating point, and between integer and enumeration. Execution fails immediately if a range error occurs. If not specified, the default value is <code>false</code> .

Below are two examples of the `<define>` descriptor. The first example defines the symbol `identity` to have type `Ice::Identity`, and proceeds to initialize its members using `<set>`:

```
<define name="identity" type="::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

The second example uses the enumeration we first saw in Section 22.3.5 to define the symbol `manufacturer` and assign it a default value:

```
<define name="manufacturer" type="::BigThree"
  value="::DaimlerChrysler"/>
```

<remove>

Removes an element from a sequence or dictionary. It is legal to remove an element while traversing a sequence or dictionary using `<iterate>`, however

the traversal order after removal is undefined. The attributes supported by this descriptor are described in Table 22.20.

Table 22.20. Attributes for `<remove>` descriptor.

Name	Description
target	An expression that must select a modifiable sequence or dictionary.
key	An expression that must evaluate to a value compatible with the key type of the target dictionary.
index	An expression that must evaluate to an integer value representing the index of the sequence element to be removed.

`<fail>`

Causes transformation to fail immediately. If `test` is specified, transformation fails only if the expression evaluates to `true`. The attributes supported by this descriptor are described in Table 22.21.

Table 22.21. Attributes for `<fail>` descriptor.

Name	Description
message	A message to display upon transformation failure.
test	A boolean expression.

The following `<fail>` descriptor terminates the transformation if a range error is detected:

```
<fail message="range error occurred in ticket count!"
    test="value.ticketCount > 32767"/>
```

<echo>

Displays values and informational messages. If no attributes are specified, only a newline is printed. The attributes supported by this descriptor are described in Table 22.22.

Table 22.22. Attributes for `<echo>` descriptor.

Name	Description
message	A message to display.
value	An expression. The value of the expression is displayed in a structured format.

Shown below is an `<echo>` descriptor that uses both `message` and `value` attributes:

```
<if test="value.ticketCount > 32767">
  <echo message="range error occurred in ticket count: "
    value="value.ticketCount"/>
</if>
```

22.7 Using `dumpdb`

This section describes the invocation of `dumpdb` and provides advice on how to best use it.

22.7.1 Options

The tool supports the standard command-line options common to all Slice processors listed in Section 4.18. The options specific to `dumpdb` are described below:

- **--load *SLICE***

Loads the Slice definitions contained in the file *SLICE*. This option may be specified multiple times if several files must be loaded. However, it is the user's responsibility to ensure that duplicate definitions do not occur (which is possible when two files are loaded that share a common include file). One strategy for avoiding duplicate definitions is to load a single Slice containing

only `#include` statements for each of the Slice files to be loaded. No duplication is possible in this case if the included files use include guards correctly.

- **--key TYPE**
--value TYPE

Specifies the Slice type of the database key and value.

- **-e**

Indicates that a Freeze evictor database is being examined. This option is provided as a convenience, in that it automatically sets the database key and value types to those appropriate for the Freeze evictor, and therefore the **--key** and **--value** options are not necessary. Specifically, the key type of a Freeze evictor database is `Freeze::EvictorStorageKey`, and the value type is `Freeze::ObjectRecord`. These types are defined in the Slice file `Freeze/EvictorStorage.ice`, however this file does not need to be explicitly loaded.

- **-o FILE**

Create a file named **FILE** containing sample descriptors for the loaded Slice definitions. The key and value types must be specified using the **--key** and **--value** options, or using the **-e** option. If the **--select** option is used, its expression is included in the sample descriptors. The database arguments are not necessary because database traversal is not performed when the **-o** option is used.

- **-f FILE**

Execute the descriptors in the file named **FILE**. The file's `<database>` descriptor specifies the key and value types, therefore the **--key**, **--value** and **-e** options are not necessary when **-f** is used.

- **--select EXPR**

Only display those records for which the expression **EXPR** is true. The expression can refer to the symbols `key` and `value`.

22.7.2 Database Arguments

If **dumpdb** is invoked to examine a database, it requires two arguments:

- **dbenv**

The pathname of the database environment directory.

- **db**

The name of the database file. **dumpdb** opens this database as read-only, and traversal occurs within a transaction.

22.7.3 Use Cases

The command line options described in Section 22.7.1 support several modes of operation:

- Dump an entire database.
- Dump selected records of a database.
- Emit a sample descriptor file.
- Execute a descriptor file.

These use cases are described in the following sections.

Dump an Entire Database

The simplest way to examine a database with **dumpdb** is to dump its entire contents. You must specify the database key and value types, load the necessary Slice definitions, and supply the names of the database environment directory and database file. For example, this command dumps a Freeze map database whose key type is `string` and value type is `Employee`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \  
db emp.db
```

Dump Selected Records

If only certain records are of interest to you, the `--select` option provides a convenient way to filter the output of **dumpdb**. In the following example, we select employees from the accounting department:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \  
--select "value.dept == 'Accounting'" db emp.db
```

In cases where the database records contain polymorphic class instances, you must be careful to specify an expression that can be successfully evaluated against all records. For example, **dumpdb** fails immediately if, while evaluating the expression, a data member is encountered that does not exist in the class instance. The safest way to write an expression in this case is to check the type of the class instance before referring to any of its data members.

In the example below, we assume that a Freeze evictor database contains instances of various classes in a class hierarchy, and we are only interested in instances of `Manager` whose employee count is greater than 10:

```
$ dumpdb -e --load Employee.ice \  
  --select "value.servant.ice_id == '::Manager' and \  
  value.servant.group.length > 10" db emp.db
```

Alternatively, if `Manager` has derived classes, then the expression can be written in a different way so that instances of `Manager` and any of its derived classes are considered:

```
$ dumpdb -e --load Employee.ice \  
  --select "value.servant.ice_isA('::Manager') and \  
  value.servant.group.length > 10" db emp.db
```

Creating a Sample Descriptor File

If you require more sophisticated filtering or scripting capabilities, then you must use a descriptor file. The easiest way to get started with a descriptor file is to generate a template using `dumpdb`:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \  
  -o dump.xml
```

The output file `dump.xml` is complete and can be executed immediately if desired, but typically the file is used as a starting point for further customization.

If the `--select` option is specified, its expression is included in the generated `<record>` descriptor as the value of the `test` attribute in an `<if>` descriptor.

Notice that database arguments are not required when `-o` is specified, because `dumpdb` terminates immediately after generating the sample file.

Executing a Descriptor File

Use the `-f` option when you are ready to execute a descriptor file. For example, we can execute the descriptor we generated in the previous section using this command:

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

22.8 Descriptor Expression Language

An expression language is provided for use in FreezeScript descriptors.

22.8.1 Operators

The language supports the usual complement of operators: `and`, `or`, `not`, `+`, `-`, `/`, `*`, `%`, `<`, `>`, `==`, `!=`, `<=`, `>=`, `(`, `)`. Note that the `<` character must be escaped as `<` in order to comply with XML syntax restrictions.

22.8.2 Literals

Literal values can be specified for integer, floating point, boolean, and string. The expression language supports the same syntax for literal values as that of Slice (see Section 4.7.5), with one exception: string literals must be enclosed in single quotes.

22.8.3 Symbols

Certain descriptors introduce symbols that can be used in expressions. These symbols must comply with the naming rules for Slice identifiers (i.e., a leading letter followed by zero or more alphanumeric characters). Data members are accessed using dotted notation, such as `value.memberA.memberB`.

Expressions can refer to Slice constants and enumerators using scoped names. In a **transformdb** descriptor, there are two sets of Slice definitions, therefore the expression must indicate which set of definitions it is accessing by prefixing the scoped name with `::Old` or `::New`. For example, the expression `old.fruitMember == ::Old::Pear` evaluates to `true` if the data member `fruitMember` has the enumerated value `Pear`. In **dumpdb**, only one set of Slice definitions is present and therefore the constant or enumerator can be identified without any special prefix.

22.8.4 Nil

The keyword `nil` represents a nil value of type `Object`. This keyword can be used in expressions to test for a nil object value, and can also be used to set an object value to nil.

22.8.5 Elements

Dictionary and sequence elements are accessed using array notation, such as `userMap['marc']` or `stringSeq[5]`. An error occurs if an expression attempts to access a dictionary or sequence element that does not exist. For dictio-

naries, the recommended practice is to check for the presence of a key before accessing it:

```
<if test="userMap.containsKey('marc') and userMap['marc'].active">
```

See Section 22.8.8 for more information on the `containsKey` function.

Similarly, expressions involving sequences should check the length of the sequence:

```
<if test="stringSeq.length > 5 and stringSeq[5] == 'fruit'">
```

See Section 22.8.7 for details on the `length` member.

22.8.6 Reserved Keywords

The following keywords are reserved: `and`, `or`, `not`, `true`, `false`, `nil`.

22.8.7 Implicit Members

Certain `Slice` types support implicit data members:

- Dictionary and sequence instances have a member `length` representing the number of elements.
- Object instances have a member `ice_id` denoting the actual type of the object, and a member `ice_facets` containing the object's facets. The `ice_facets` member can be used like any other dictionary value.

22.8.8 Functions

The expression language supports two forms of function invocation: member functions and global functions. A member function is invoked on a particular data value, whereas global functions are not bound to a data value. For instance, here is an expression that invokes the `find` member function of a `string` value:

```
old.stringValue.find('theSubstring') != -1
```

And here is an example that invokes the global function `stringToIdentity`:

```
stringToIdentity(old.stringValue)
```

If a function takes multiple arguments, the arguments must be separated with commas.

String Member Functions

The `string` data type supports the following member functions:

- `int find(string match[, int start])`
Returns the index of the substring, or `-1` if not found. A starting position can optionally be supplied.
- `string replace(int start, int len, string str)`
Replaces a given portion of the string with a new substring, and returns the modified string.
- `string substr(int start[, int len])`
Returns a substring beginning at the given start position. If the optional length argument is supplied, the substring contains at most `len` characters, otherwise the substring contains the remainder of the string.

Dictionary Member Functions

The `dictionary` data type supports the following member function:

- `bool containsKey(key)`
Returns `true` if the dictionary contains an element with the given key, or `false` otherwise. The `key` argument must have a value that is compatible with the dictionary's key type.

Object Member Functions

Object instances support the following member function:

- `bool ice_isA(string id)`
Returns `true` if the object implements the given interface type, or `false` otherwise. This function cannot be invoked on a `nil` object.

Global Functions

The following global functions are provided:

- `string generateUUID()`
Returns a new UUID.
- `string identityToString(Ice::Identity id)`
Converts an identity into its string representation.
- `string lowercase(string str)`
Returns a new string converted to lowercase.

- `string proxyToString(Ice::ObjectPrx prx)`
Returns the string representation of the given proxy.
- `Ice::Identity stringToIdentity(string str)`
Converts a string into an `Ice::Identity`.
- `Ice::ObjectPrx stringToProxy(string str)`
Converts a string into a proxy.
- `string typeOf(val)`
Returns the formal Slice type of the argument.

22.9 Summary

FreezeScript provides tools that ease the maintenance of Freeze databases. The **transformdb** tool simplifies the task of migrating a database when its persistent types have changed, offering an automatic mode requiring no manual intervention, and a custom mode in which scripted changes are possible. Database inspection and reporting is accomplished using the **dumpdb** tool, which supports a number of operational modes including a scripting capability.

Chapter 23

IceSSL

23.1 Chapter Overview

In this chapter we present IceSSL, an optional security component for Ice applications. Section 23.2 provides an overview of the SSL protocol and the infrastructure required to support it. Section 23.3 and Section 23.4 discuss the configuration aspects of IceSSL, while Section 23.5 details the configuration file syntax. Finally, Section 23.6 shows how an application can interact directly with IceSSL.

23.2 Introduction

Security is an important consideration for many distributed applications, both within corporate intranets as well as over untrusted networks, such as the Internet. The ability to protect sensitive information, ensure its integrity, and verify the identities of the communicating parties is essential for developing secure applications. With those goals in mind, Ice includes the IceSSL *plug-in* that provides these capabilities using the Secure Socket Layer (SSL) protocol.¹

1. IceSSL is currently only available for C++ applications.

23.2.1 SSL Overview

SSL is the protocol that enables Web browsers to conduct secure transactions and therefore is one of the most commonly used protocols for secure network communication. You do not need to know the technical details of the SSL protocol in order to use IceSSL successfully (and those details are outside the scope of this text). However, it would be helpful to have a high-level understanding of how the protocol works and the infrastructure required to support it. (For more information on the SSL protocol, see [22].)

SSL provides a secure environment for communication (without sacrificing too much performance) by combining a number of cryptographic techniques:

- public key encryption
- symmetric (shared key) encryption
- message authentication codes
- digital certificates

When a client establishes an SSL connection to a server, a *handshake* is performed. During a typical handshake, digital certificates that identify the communicating parties are validated, and symmetric keys are exchanged for encrypting the session traffic. Public key encryption, which is too slow to be used for the bulk of a session's data transfer, is used heavily during the handshaking phase. Once the handshake is complete, SSL uses message authentication codes to ensure data integrity, allowing the client and server to communicate at will with reasonable assurance that their messages are secure.

23.2.2 Public Key Infrastructure

Security requires trust, and public key cryptography by itself does nothing to establish trust. SSL addresses the issue of trust using Public Key Infrastructure (PKI), which binds public keys to identities using certificates. A Certification Authority (CA) is often used to issue certificates and verify the identities of certificate owners.

Smaller applications often require very little in the way of infrastructure; private certificates created for the client and server using freely-available tools may suffice. However, enterprises have more elaborate security requirements, in which setting up a private CA² or using a third-party CA (such as Verisign) is necessary.

It is also possible to avoid the use of certificates altogether; encryption is still used to obscure the session traffic, but the benefits of authentication are sacrificed in favor of reduced complexity and administration.

For more information on PKI, see [5].

23.2.3 Requirements

Integrating IceSSL into your application generally requires no changes to your source code, but does involve the following administrative tasks:

- creating a public key infrastructure (if necessary)
- writing the XML configuration files for the IceSSL plug-in
- modifying your application's configuration to install the IceSSL plug-in and use secure connections

The remainder of this chapter primarily discusses the configuration aspects of IceSSL.

23.3 Configuring IceSSL

IceSSL is configured separately for client and server functionality, reflecting the roles an Ice application can play: client only, server only, or mixed client-server. The configuration data is written in XML and provides the information IceSSL needs to initialize the SSL protocol, including the selection of cryptographic algorithms and the names of certificate and key files. This section presents example configuration files, while Section 23.5 provides a detailed reference of the XML elements comprising the IceSSL configuration.

23.3.1 RSA Example

Ice uses the configuration file shown below for its IceSSL-related tests and sample programs (see the file `certs/sslconfig.xml` in the Ice source distribution).

-
2. OpenSSL, the open-source SSL toolkit used by IceSSL, provides tools for setting up a minimal Certification Authority that may be suitable for development purposes or for a small organization.

The top-level element in every IceSSL configuration file is `SSLConfig`. Nested within this element is a `client` or `server` element (or in this case, both):

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE SSLConfig SYSTEM "sslconfig.dtd">
<SSLConfig>
  <client>
    <general version="SSLv23" cipherlist="RC4-MD5"
      verifymode="peer" verifydepth="10" />
    <certauthority file="cacert.pem" />
    <basecerts>
      <rsacert keysize="1024">
        <public encoding="PEM"
          filename="c_rsa1024_pub.pem" />
        <private encoding="PEM"
          filename="c_rsa1024_priv.pem" />
      </rsacert>
    </basecerts>
  </client>
  <server>
    <general version="SSLv23" cipherlist="RC4-MD5"
      verifymode="peer" verifydepth="10" />
    <certauthority file="cacert.pem" />
    <basecerts>
      <rsacert keysize="1024">
        <public encoding="PEM"
          filename="s_rsa1024_pub.pem" />
        <private encoding="PEM"
          filename="s_rsa1024_priv.pem" />
      </rsacert>
    </basecerts>
  </server>
</SSLConfig>
```

As you can see, the `client` and `server` elements are very similar. Let us examine the contents of `client` in more detail (the discussion applies equally well to the `server` element). The first element we encounter is `general`, which selects cryptographic algorithms and controls certificate verification:

```
<general version="SSLv23" cipherlist="RC4-MD5"
  verifymode="peer" verifydepth="10" />
```

The `version` attribute chooses a protocol version; in this case, the value `SSLv23` is really a compatibility mode that includes SSL version 2, SSL version

3, and Transport Layer Security (TLS) version 1. Note however that SSL version 2 has several security flaws and therefore is not supported by IceSSL.

A cipher suite is selected using the `cipherlist` attribute. The value `RC4-MD5` chooses RC4 as the encryption algorithm and MD5 as the message digest algorithm.

The `verifymode` and `verifydepth` attributes affect certificate authentication. A verification mode of `peer` requires the client to authenticate the server's certificate before allowing the connection to proceed, and allows the client to send its certificate to the server upon request. A certificate may have an arbitrarily long chain of signing certificates that must be searched for a trusted CA; the `verify-depth` attribute allows you to establish a limit on the depth of the search. If this limit is exceeded, the connection fails.

Next, the `certauthority` element specifies the filename of a trusted CA certificate (or certificate chain):

```
<certauthority file="cacert.pem" />
```

As the certificate filename implies, it is encoded in the Privacy Enhanced Mail (PEM) format, which is the format IceSSL requires for all certificates and keys.

Finally, the `basecerts` element provides the public certificate and private key needed to identify the peer and authenticate certificates:

```
<basecerts>
  <rsacert keysize="1024">
    <public encoding="PEM"
      filename="c_rsa1024_pub.pem" />
    <private encoding="PEM"
      filename="c_rsa1024_priv.pem" />
  </rsacert>
</basecerts>
```

The `rsacert` element encapsulates the `public` and `private` elements and supplies the bit strength (1024) of the encryption key. The `public` element defines the filename and encoding of the public certificate. Similarly, the `private` element defines the filename and encoding of the private key. Although the encoding format is provided in an attribute, currently only `PEM` is supported.

23.3.2 ADH Example

The following example uses ADH (the Anonymous Diffie-Hellman cipher). ADH is not a good choice in most cases because, as its name implies, there is no authentication of the communicating parties, and it is vulnerable to man-in-the-middle

attacks. However, it still provides encryption of the session traffic and requires very little administration and therefore may be useful in certain situations. The IceSSL configuration file shown below demonstrates how to use ADH:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE SSLConfig SYSTEM "sslconfig.dtd">
<SSLConfig>
  <client>
    <general version="SSLv23" cipherlist="ADH"/>
    <basecerts>
      <dhparams keysize="512" filename="dh512.pem"/>
    </basecerts>
  </client>
  <server>
    <general version="SSLv23" cipherlist="ADH"/>
    <basecerts>
      <dhparams keysize="512" filename="dh512.pem"/>
    </basecerts>
  </server>
</SSLConfig>
```

The important aspects of this configuration are the selection of the ADH cipher and the use of a `dhparams` element to provide the Diffie-Hellman parameter file. We specify the same parameter file for both client and server, but they are not required to have the same parameters. During the handshake, the Diffie-Hellman parameters of one peer are shared with the other, so only one set of parameters is actually used. It is up to the SSL implementation to determine which peer's parameters are used, but usually it is the server's parameters. We do not specify any certificates, because there is no peer authentication.

23.3.3 Configuration File Strategies

The IceSSL configuration file provides a lot of flexibility. For example, if a client and a server have access to the same filesystem, keeping their IceSSL configurations in the same file can simplify administrative duties. However, it is also perfectly reasonable (and often necessary) to create separate files for each client and server, with the client's configuration file containing only a `client` element, and the server's configuration file containing only a `server` element.

If either the client or the server is a mixed-mode application (i.e., both sends and receives requests), then care must be taken when attempting to share IceSSL configurations: a mixed-mode application's configuration requires both `client`

and server elements, and it may not be appropriate for another peer to use the same configuration (keys, certificates, etc.).

23.4 Configuring Applications

Configuring an application to use IceSSL requires installing the plug-in and creating SSL endpoints.

23.4.1 Installing IceSSL

Ice supports a generic plug-in facility that allows extensions (such as IceSSL) to be installed dynamically without changing the application source code. The executable code for the IceSSL plug-in resides in a shared library on Unix and a dynamic link library (DLL) on Windows. The plug-in is installed using a configuration property:

```
Ice.Plugin.IceSSL=IceSSL:create
```

The last component of the property name (`IceSSL`) becomes the plug-in's official identifier for configuration purposes, but the IceSSL plug-in requires its identifier to be `IceSSL`. The property value `IceSSL:create` is sufficient to allow the Ice run time to locate the IceSSL library (on both Unix and Windows) and initialize the plug-in. The only requirement is that the library reside in a directory that appears in the library path (`LD_LIBRARY_PATH` on Unix, `PATH` on Windows). For more information on the `Ice.Plugin` property, see Appendix C.

The next step is to supply the plug-in with its configuration file(s), which is also accomplished via configuration properties:

```
IceSSL.Client.CertPath=/opt/certs  
IceSSL.Client.Config=sslconfig.xml  
IceSSL.Server.CertPath=/opt/certs  
IceSSL.Server.Config=sslconfig.xml
```

The properties are split into client and server versions, similar to the configuration data described in Section 23.3. A client need only specify the `IceSSL.Client` properties, and likewise for a server. When initializing the plug-in, the client configuration is read from the file specified by the `IceSSL.Client.Config` property, and the server configuration will be read from the file specified by the `IceSSL.Server.Config` property. (It is legal to use the same file for both

properties.) The `CertPath` properties specify a default directory in which the respective configuration and certificate files can be found.

The plug-in supports a number of additional configuration properties that are described in Appendix C, but the ones described above are sufficient for many applications.

23.4.2 Creating SSL Endpoints

Installing the IceSSL plug-in enables you to use a new protocol, `ssl`, in your endpoints. For example, the following endpoint list creates a TCP endpoint, an SSL endpoint, and a UDP endpoint:

```
MyAdapter.Endpoints=tcp -p 8000:ssl -p 8001:udp -p 8000
```

As this example demonstrates, it is possible for a UDP endpoint to use the same port number as a TCP or SSL endpoint, because UDP is a different protocol and therefore has its own set of ports. It is not possible for a TCP endpoint and an SSL endpoint to use the same port number, because SSL is essentially a layer over TCP.

Using SSL in stringified proxies is equally straightforward:

```
MyProxy=MyObject:tcp -p 8000:ssl -p 8001:udp -p 8000
```

For more information on proxies and endpoints, see Appendix D.

23.4.3 Security Considerations

Defining an object adapter's endpoints to use multiple protocols, as shown in the example in Section 23.4.2, has obvious security implications. If your intent is to use SSL to protect session traffic and/or restrict access to the server, then you should only define SSL endpoints.

There can be situations, however, in which insecure endpoint protocols are advantageous. Figure 23.1 illustrates an environment in which TCP endpoints are allowed behind the firewall, but external clients are required to use SSL.

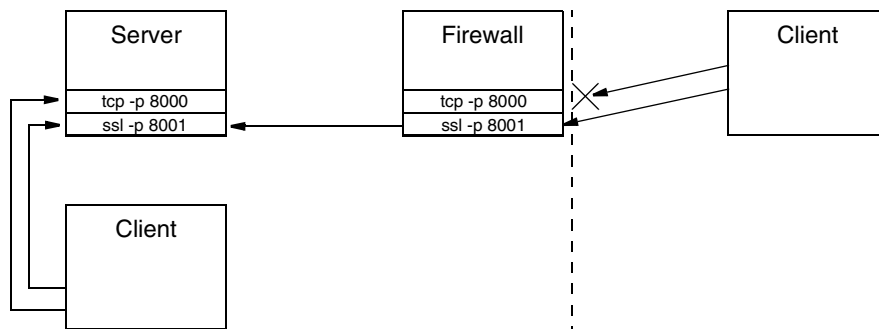


Figure 23.1. An application of multiple protocol endpoints.

The firewall in Figure 23.1 is configured to block external access to TCP port 8000 and to forward connections to port 8001 to the server machine.

One reason for using TCP behind the firewall is that it is more efficient than SSL and requires less administrative work to use. Of course, this scenario assumes that internal clients can be trusted, which is not true in many environments.

For more information on using SSL in complex network architectures, see Chapter 24.

23.5 Configuration Reference

This section describes each of the XML elements comprising the IceSSL configuration file.

23.5.1 Structure

The IceSSL configuration file has the following structure:

```
<SSLConfig>
  <client>
    <general .../>
    <certauthority .../>
    <basecerts>
      <rsacert ...>
```

```

        <public .../>
        <private .../>
    </rsacert>
    <dsacert ...>
        <public .../>
        <private .../>
    </dsacert>
    <dhparams .../>
</basecerts>
</client>
<server>
    <general .../>
    <certauthority .../>
    <basecerts>
        <rsacert ...>
            <public .../>
            <private .../>
        </rsacert>
        <dsacert ...>
            <public .../>
            <private .../>
        </dsacert>
        <dhparams .../>
    </basecerts>
    <tempcerts>
        <rsacert ...>
            <public .../>
            <private .../>
        </rsacert>
        <dhparams .../>
    </tempcerts>
</server>
</SSLConfig>

```

Some of the elements shown above are optional; see the element descriptions for details. A DTD for the configuration file is provided in the Ice source distribution as `certs/sslconfig.dtd`.

23.5.2 basecerts

The `basecerts` element contains certificate and key specifications. It is a mandatory child of `client` and `server`, and may contain one `rsacert` element, one `dsacert` element, and one `dhparams` element. In particular, `basecerts` must contain an `rsacert` element or a `dsacert` element (or

both). If a `dsacert` element is provided, a `dhparams` element normally accompanies it.

23.5.3 **certauthority**

The `certauthority` element specifies the file containing the chain of trusted CA certificates, as well as a directory in which certificates can be located. It is an optional child of `client` and `server`, and supports the following attributes:

Table 23.1. Attributes for `certauthority` element.

Attribute	Description	Required
<code>file</code>	A file (in PEM format) containing a chain of trusted CA certificates. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	No
<code>path</code>	The absolute pathname of a directory containing trusted CA certificates. This directory needs to be prepared before use by running the OpenSSL utility c_rehash in the directory.	No

Both attributes are marked as optional, but at least one of the attributes should be defined. A significant difference between the attributes is the time at which the certificates are loaded. The certificate(s) indicated by the `file` attribute are loaded immediately, whereas the certificates residing in the directory indicated by the `path` attribute are loaded as necessary during the SSL handshake.

23.5.4 **client**

The `client` element establishes the configuration for the client-side components of the IceSSL plug-in. It is an optional child of `SSLConfig` and, if specified, it must contain a `general` element and a `basecerts` element, and may contain a `certauthority` element.

23.5.5 dhparams

The `dhparams` element specifies parameters for the Diffie-Hellman key agreement algorithm. It is an optional child of `basecerts` and `tempcerts`, and supports the following attributes:

Table 23.2. Attributes for `dhparams` element.

Attribute	Description	Required
<code>keysize</code>	The number of bits in the prime number parameter.	Yes
<code>encoding</code>	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
<code>filename</code>	A file containing Diffie-Hellman parameters. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	Yes

If this element is specified but the parameters cannot be read from the file specified by the `filename` attribute, IceSSL generates temporary Diffie-Hellman parameters using a 512-bit prime number.

23.5.6 dsacert

The `dsacert` element defines the public certificate and private key for the Digital Signature Algorithm (DSA). It is an optional child of `basecerts` and `tempcerts`, and must contain a `public` element and a `private` element. It supports the following attribute:

Table 23.3. Attributes for `dsacert` element.

Attribute	Description	Required
<code>keysize</code>	The bit strength of the keys.	Yes

23.5.7 general

The `general` element configures the SSL protocol. It is a mandatory child of `client` and `server`, and supports the following attributes:

Table 23.4. Attributes for general element.

Attribute	Description	Required
version	The SSL protocol version. Possible values are <code>SSLv23</code> , <code>SSLv3</code> , and <code>TLSv1</code> . The value <code>SSLv23</code> is a compatibility mode that includes SSL version 2, SSL version 3, and TLS version 1. Note that SSL version 2 has several security flaws and therefore is not supported by IceSSL. If not specified, the default value is <code>SSLv23</code> .	No
cipherlist	The list of cipher suites that SSL is allowed to use, separated by colons. If not specified, a default list of cipher suites is used.	No
context	A session ID context. Specifying this attribute enables session caching in a server.	No
verifymode	The certificate verification behavior. See Table 23.6 for the possible values and their semantics. Multiple values can be specified, separated by white space or a vertical bar (). If not specified, the default value is <code>none</code> .	No
verifydepth	The maximum depth when searching a certificate chain for a trusted CA. If this depth is exceeded, the handshake fails. If not specified, the default value is 10.	No
randombytes	Seeds the pseudo-random number generator with data from one or more files. A pathname may represent a Unix domain socket supporting the EGD protocol. Multiple pathnames must be separated by a semi-colon (;) on Windows and a colon (:) on Unix. At least 512 bytes of data must be provided.	No

cipherlist

The `cipherlist` attribute identifies the cipher suites that SSL is allowed to negotiate during the handshake. A cipher suite is a set of algorithms that satisfies the four requirements for establishing a secure connection: signing and authenti-

cation, key exchange, secure hashing, and encryption. Some algorithms satisfy more than one requirement, and there are many possible combinations.

The value of this attribute is given directly to the low-level OpenSSL library, on which IceSSL is based. Therefore, OpenSSL determines the allowable cipher suites, which in turn depend on how the OpenSSL distribution was compiled. You can obtain a complete list of the supported cipher suites using the **openssl** command:

```
$ openssl ciphers
```

This command will likely generate a long list. To simplify the selection process, OpenSSL supports several classes of ciphers:

Table 23.5. Cipher classes.

Class	Description
ALL	All possible combinations.
ADH	Anonymous ciphers.
LOW	Low bit-strength ciphers.
EXP	Export-crippled ciphers.

Classes and ciphers can be excluded by prefixing them with an exclamation point. The special keyword **@STRENGTH** sorts the cipher list in order of their strength, so that SSL gives preference to the more secure ciphers when negotiating a cipher suite. The **@STRENGTH** keyword must be the last element in the list.

For example, here is a good value for the **cipherlist** attribute:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

This value excludes the ciphers with low bit strength and known problems, and orders the remaining ciphers according to their strength.

Note that no warning is given if an unrecognized cipher is specified.

verifymode

The `verifymode` attribute has the following semantics:

Table 23.6. Semantics of `verifymode` attribute.

Value	Client Semantics	Server Semantics
none	The client verifies the server's certificates, but failure does not terminate the handshake. The client does not send a certificate if one is requested by the server.	The server does not request a certificate from the client.
peer	The client verifies the server's certificates. Verification failure terminates the handshake.	The server requests a certificate from the client, and verifies it if one is sent. Verification failure terminates the handshake.
fail_no_cert	None.	Causes the handshake to fail if the client does not provide a certificate. Must be combined with <code>peer</code> .
client_once	None.	Prevents the server from requesting a certificate from the client during a renegotiation. Must be combined with <code>peer</code> .

The value `none` must not be combined with other values. The value `peer` must not be combined with other values in a client configuration.

23.5.8 private

The `private` element specifies the private key of a signing algorithm. It is a mandatory child of `rsacert` and `dsacert`, and supports the following attributes:

Table 23.7. Attributes for private element.

Attribute	Description	Required
encoding	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
filename	A file containing the private key. A relative path-name is resolved using the directory defined by the <code>CertPath</code> property.	Yes

23.5.9 public

The `public` element specifies the public certificate of a signing algorithm. It is a mandatory child of `rsacert` and `dsacert`, and supports the following attributes:

Table 23.8. Attributes for public element.

Attribute	Description	Required
encoding	The encoding format of the file indicated by the <code>filename</code> attribute. The value must be <code>PEM</code> . If not specified, the default value is <code>PEM</code> .	No
filename	A file containing the public certificate. A relative pathname is resolved using the directory defined by the <code>CertPath</code> property.	Yes

23.5.10 rsacert

The `rsacert` element defines the RSA public certificate and private key. It is an optional child of `basecerts` and `tempcerts`, and must contain a `public` element and a `private` element. It supports the following attribute:

Table 23.9. Attributes for `dsacert` element.

Attribute	Description	Required
<code>keysize</code>	The bit strength of the keys.	Yes

23.5.11 server

The `server` element establishes the configuration for the server-side components of the IceSSL plug-in. It must contain a `general` element and a `basecerts` element, and may contain a `certauthority` element and a `tempcerts` element.

23.5.12 SSLConfig

The `SSLConfig` is the top-level element in an IceSSL configuration file. It must contain one `client` element, or one `server` element, or both.

23.5.13 tempcerts

The `tempcerts` element contains configuration parameters for temporary keys. It is an optional child of `server`, and may contain multiple `rsacert` and `dhparams` elements.

Temporary (also known as *ephemeral*) keys are required when using DSA authentication, and increase security when using RSA authentication (although ephemeral RSA keys are rarely used in practice). The `tempcerts` element allows you to specify files containing RSA public certificates and private keys, as well as Diffie-Hellman parameters. Loading these items from files avoids the need to generate them dynamically, which can be computationally expensive. A file is not read until a matching key size is requested by SSL; if no file is found with a matching key size, the requested data is generated dynamically.

23.6 Programming with IceSSL

Some applications may require IceSSL functionality that is only available programmatically. Here are a few reasons why an application might need direct interaction with the plug-in:

- To load configuration files dynamically
- To add keys and trusted certificates dynamically
- To install custom certificate verification rules

In order to accomplish any of these tasks, an application needs to obtain a reference to the plug-in from the communicator. Ice defines the Slice interface `Ice::Plugin` that represents a generic plug-in, and IceSSL defines a derived interface `IceSSL::Plugin`. These interfaces are described in Appendix B, but the following code example demonstrates how to obtain a reference to the IceSSL plug-in:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

The first step is to obtain a reference to the plugin manager, then ask it for the IceSSL plugin. The argument to `getPlugin` is the plug-in identifier, in this case `IceSSL` (see Section 23.4.1). Finally, we downcast to `IceSSL::Plugin`.

23.7 Summary

The Secure Socket Layer (SSL) protocol is the de facto standard for secure network communication. Its support for authentication, non-repudiation, data integrity, and strong encryption makes it the logical choice for securing Ice applications.

Although security is an optional component of Ice, it is not an afterthought. The IceSSL plug-in integrates easily into existing Ice applications, in most cases requiring nothing more than configuration changes. Naturally, some additional effort is required to create the necessary security infrastructure for an application, but in many enterprises this work will have already been done.

Chapter 24

Glacier

24.1 Chapter Overview

This chapter presents Glacier, the firewall solution for Ice applications. Section 24.2 provides an introduction to Glacier and describes the complex networking issues Glacier is designed to address. Section 24.3 presents the simplest Glacier use case in which clients require only configuration changes. The issue of callbacks, and how Glacier supports callbacks in restricted network environments, is addressed in Section 24.4. The Glacier starter is the subject of Section 24.5, and Section 24.6 details the security aspects of using Glacier.

24.2 Introduction

We have presented many examples of client/server applications in this book, all of which assume the client and server programs are running either on the same host, or on multiple hosts with no network restrictions. We can justify this assumption because this is an instructional text, but a real-world network environment is usually much more complicated: client and server hosts with access to public networks often reside behind protective router-firewalls that not only restrict incoming connections, but also allow the protected networks to run in a private address space using Network Address Translation (NAT). These features, which

are practically mandatory in today's hostile network environments, also disrupt the ideal world in which our examples are running.

24.2.1 Common Scenarios

Let us assume that a client and server need to communicate over an untrusted network, and that the client and server hosts reside in private networks behind firewalls, as shown in Figure 24.1.

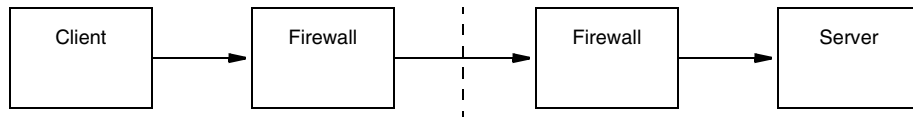


Figure 24.1. Client request in a typical network scenario.

Although the diagram looks fairly straightforward, there are several troublesome issues:

- A dedicated port on the server's firewall must be opened and configured to forward messages to the server.
- If the server uses multiple endpoints (e.g., to support both TCP and SSL), then a firewall port must be dedicated to each endpoint.
- The client's proxy must be configured to use the server's "public" endpoint, which is the hostname and dedicated port of the firewall.
- If the server returns a proxy as the result of a request, the proxy must not contain the server's private endpoint, which is inaccessible to the client.

To complicate the scenario even further, Figure 24.2 adds a callback from the server to the client. Callbacks imply that the client is also a server, therefore all of the issues associated with Figure 24.1 now apply to the client as well.

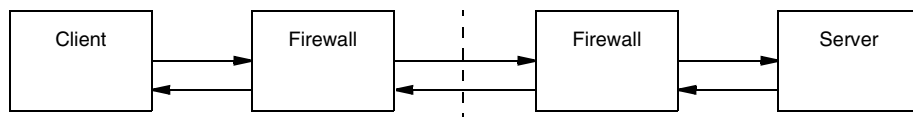


Figure 24.2. Callbacks in a typical network scenario.

As if this was not complicated enough already, Figure 24.3 adds multiple clients and servers. Each additional server (including clients requiring callbacks) adds more work for the firewall administrator as more ports are dedicated to forwarding requests.

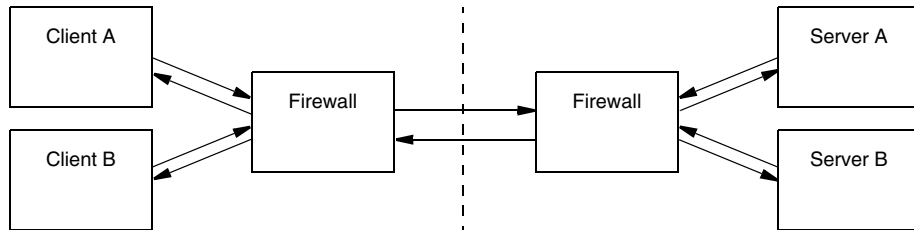


Figure 24.3. Multiple clients and servers with callbacks in a typical network scenario.

Clearly, these scenarios do not scale well, and are unnecessarily complex. Fortunately, Ice provides a solution.

24.2.2 What is Glacier?

Glacier, the router-firewall for Ice applications, addresses the issues raised in Section 24.2.1 with minimal impact on clients or servers (or firewall administrators). In Figure 24.4, Glacier becomes the server firewall for Ice applications. What is not obvious in the diagram, however, is how Glacier eliminates much of the complexity of the previous scenarios.

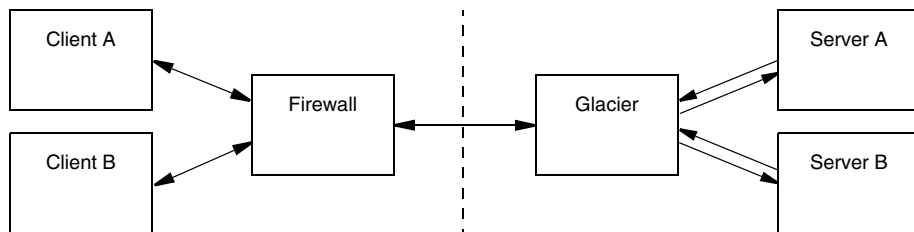


Figure 24.4. Multiple clients and servers with callbacks using Glacier.

In particular, Glacier provides the following advantages:

- Clients often require only configuration changes, not code changes, to use Glacier.
- A single Glacier port supports any number of servers.
- Servers are unaware of Glacier's presence, and require no modifications whatsoever to use Glacier. From a server's perspective, Glacier is just another local client, therefore servers are no longer required to advertise "public" endpoints in the proxies they create. Furthermore, back-end services such as IcePack (see Chapter 20) can continue to be used transparently behind the Glacier firewall.
- Callbacks are supported without requiring new connections from servers to clients. In other words, a callback from a server to a client is sent over an existing connection from the client to the server, thereby eliminating the administrative requirements associated with supporting callbacks in the client firewall.
- Glacier supports all Ice protocols (TCP, SSL, and UDP).
- Glacier requires no application-specific knowledge and therefore is very efficient: it routes request and reply messages without unmarshalling the message contents.

24.2.3 How it works

The Ice core supports a generic router facility, represented by the `Ice::Router` interface, that allows a third-party service to "intercept" requests on a properly-configured proxy and deliver them to the intended server. Glacier is an implementation of this service, although other implementations are certainly possible.

Glacier normally runs on a host with access to both networks: the public network over which clients send requests, and the private network over which those requests are routed to servers. Although it is possible in some cases to run Glacier behind a firewall, Glacier is really intended to *be* the firewall for Ice appli-

cations. It follows that Glacier must have endpoints on each network, as shown in Figure 24.5.

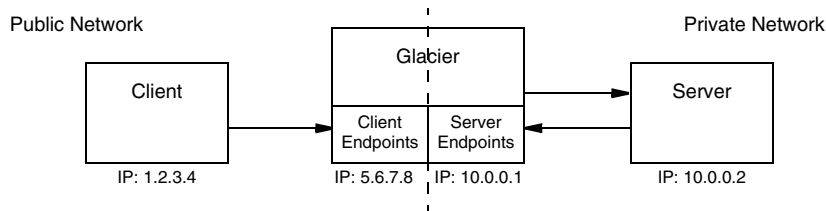


Figure 24.5. Glacier's client and server endpoints.

In the client, proxies must be configured to use Glacier as a router. This configuration can be done statically for all proxies created by a communicator, or programmatically for a particular proxy. A proxy configured to use a router is called a *routed proxy*.

When a client invokes an operation on a routed proxy, the client connects to one of Glacier's client endpoints and sends the request as if Glacier were the server. Glacier then establishes a client connection to the intended server, forwards the request, and returns the reply (if any). Glacier is essentially acting as a local client on behalf of the remote client.

If a server returns a proxy as the result of an operation, that proxy contains the server's endpoints in the private network, as usual. (Remember, the server is unaware of Glacier's presence, and therefore assumes that the proxy is usable by the client that requested it.) Of course, those endpoints are not accessible to the client and, in the absence of a router, the client would receive an exception if it were to use the proxy. When that proxy is configured with a router, however, the client ignores the server's endpoints and only sends requests to the router's client endpoints.

Glacier's server endpoints, which reside in the private network, are only used when a server makes a callback to a client. See Section 24.4 for more information on callbacks.

24.3 Using Glacier

The simplest Glacier configuration uses a single instance of the Glacier router. This configuration is capable of routing requests from multiple clients to multiple servers, but does not support callbacks from servers to clients.

Creating this configuration is straightforward:

1. Write a configuration file for the Glacier router.
2. Start the router on a host with access to the public and private networks.
3. Modify the client configuration to use the Glacier router.

For the sake of example, let us assume that the router's public address is 5.6.7.8 and its private address is 10.0.0.1.

24.3.1 Configuring the Router

The following router configuration properties establish the necessary endpoints:

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
```

The endpoint defined by `Glacier.Router.Endpoints` is called the *router control* endpoint because it is used by the Ice run time in a client to interact directly with the router. The endpoint defined by the property `Glacier.Router.Client.Endpoints` is where requests from routed proxies are sent. Both of these endpoints must be accessible to clients and therefore are defined on the public network interface.¹

The router control endpoint uses a fixed port because clients may be statically configured with a proxy for this endpoint. The client endpoint, on the other hand, does not require a fixed port.

24.3.2 Starting the Router

Assuming the configuration properties shown in Section 24.3.1 are stored in a file named `config`, the router can be started with the following command:

```
$ glacierreouter --Ice.Config=config
```

24.3.3 Configuring the Client

The following property configures all of the proxies in a client to use a Glacier router:

1. This sample configuration uses TCP as the endpoint protocol, although in most cases SSL is preferable (see Section 24.6). One case where a TCP endpoint is necessary is for Java clients, since SSL is not currently supported in Ice for Java.

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

The value of this property is a proxy for the router control object, therefore the endpoint must match the one in `Glacier.Router.Endpoints`.

24.3.4 Example

The `demo/Ice/hello` example can be used to test this configuration. In fact, the example's configuration file already contains the relevant property definitions for using a Glacier router, but the file may require editing to enable some properties that are commented out by default. See the comments in the configuration file for more information.

24.4 Callbacks

Callbacks from servers to clients are commonly used in distributed applications, often for notification purposes (such as the completion of a long-running calculation or a change to a database record). Unfortunately, supporting callbacks in a complicated network environment presents its own set of problems, as described in Section 24.2.1. Ice overcomes these obstacles using a Glacier router and bi-directional connections.

24.4.1 Bi-directional Connections

While a regular unrouted connection only allows requests to flow in one direction (from client to server), a bi-directional connection enables requests to flow in both directions. This capability is necessary to circumvent the network restrictions discussed in Section 24.2.1, namely, client-side firewalls that prevent a server from establishing an independent connection directly to the client. By sending callback requests over the existing connection from the client to the server (more accurately, from the client to the router), we have created a virtual connection

back to the client. Figure 24.6 illustrates the steps involved in making a callback using Glacier.

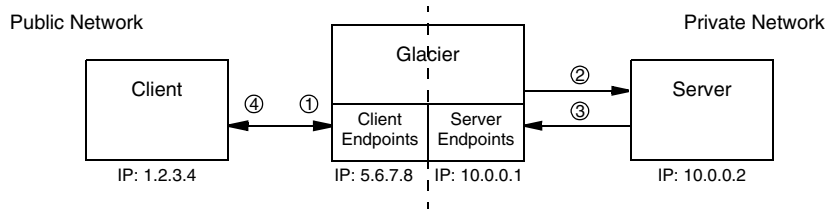


Figure 24.6. A callback via Glacier.

1. The client has a routed proxy for the server and makes an invocation. A connection is established to the router's client endpoint and the request is sent to the router.
2. The router, using information from the client's proxy, establishes a connection to the server and forwards the request. In this example, one of the arguments in the request is a proxy for a callback object in the client.
3. The server makes a callback to the client. For this to succeed, the proxy for the callback object must contain endpoints that are accessible to the server. The only path back to the client is through the router, therefore the proxy contains the router's server endpoints (see Section 24.4.4). The server connects to the router and sends the request.
4. The router forwards the callback request to the client using the bi-directional connection established in step 1.

The arrows in Figure 24.6 indicate the flow of requests; notice that two connections are used between the router and the server. Since the server is unaware of the router, it does not use routed proxies, and therefore does not use bi-directional connections.

24.4.2 Lifetime of a Bi-directional Connection

When a client terminates, it closes its connection to the router. If a server later attempts to make a callback to the client, the attempt fails because the router has no connection to the client over which to forward the request. This situation is no worse than if the server attempted to contact the client directly, which would be prevented by the client firewall. However, this illustrates the inherent limitation of

bi-directional connections: a client is only available for callbacks as long as it is connected to the router.

24.4.3 Callback Requirements

The principal issue we face in using callbacks is that a single instance of a Glacier router cannot forward callback requests to more than one client. Therefore, any client that uses callbacks must use its own instance of a Glacier router. This requirement presents an administrative dilemma: how do you (easily) ensure that each client gets its own router instance? We present the solution for this problem in Section 24.5; for now let us assume that we have only one client, and one router instance.

24.4.4 Configuring the Router

In order for the router to support callbacks from servers, it needs to have endpoints in the private network. The configuration file shown below adds the property `Glacier.Router.Server.Endpoints`:

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
```

Similar to the client endpoints described in Section 24.3.1, the server endpoints do not require fixed ports.

24.4.5 Configuring the Client Object Adapter

A client that receives callbacks is also a server, and therefore must have an object adapter. Typically, an object adapter has endpoints in the local network, but those endpoints are of no use to a server in our restricted network environment. We really want the proxy to use the router's server endpoints, and we accomplish that by configuring the object adapter with a proxy for the router. A single configuration property is all that we need:

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

If you recall from Section 24.3.3, the `Ice.Default.Router` property configures all proxies with a router. In a server context, this property also configures all object adapters with a router. Note, however, that multiple object adapters created by the same communicator cannot use the same router, therefore we have to be

careful when using this property in a server context. Since our client only has one object adapter, this does exactly what we need (Section 24.4.7 discusses strategies for clients that require multiple object adapters).

For each object adapter, the Ice run time maintains a list of endpoints that are embedded in proxies created by that adapter. Normally this list simply contains the local endpoints defined for the object adapter but, when the adapter is configured with a router, the list also contains the router's server endpoints. In our callback example, the client does not need any local endpoints because it does not expect callbacks from local servers, so the proxies only contain the router's server endpoints.

It may seem unusual to create an object adapter that has no endpoints of its own and, in the absence of a router, there is no reason to do so. When using a router, however, the object adapter is necessary in order to service callback requests. Although it does not have local endpoints and therefore cannot receive new local connections, the object adapter does receive a “virtual” connection when the client establishes an outgoing connection to the router.

24.4.6 Active Connection Management

Ice supports *active connection management* (ACM). ACM conserves resources by periodically closing idle connections. This is a very useful feature, but generally should not be used for bi-directional connections, otherwise ACM might close a connection over which a client is still expecting a callback. ACM is enabled by default (see Appendix C for a description of the `Ice.ConnectionIdleTime` property).

24.4.7 Object Adapter Strategies

An application that needs to support callback requests from a router as well as requests from local clients should use multiple object adapters. This strategy ensures that proxies created by these object adapters contain the appropriate endpoints. For example, suppose we have the network configuration shown in

Figure 24.7. Notice that the two local area networks use the same private network addresses, which is not an unrealistic scenario.

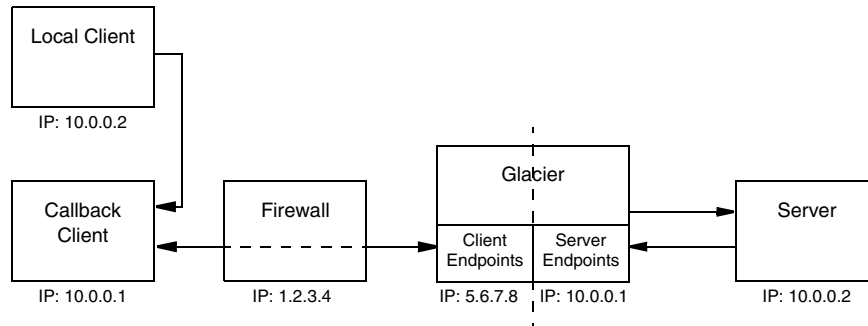


Figure 24.7. Supporting callback and local requests.

Now, if the callback client were to use a single object adapter for handling both callback requests and local requests, then any proxies created by that object adapter would contain the application's local endpoints as well as the router's server endpoints. As you might imagine, this could cause some subtle problems.

1. When the local client attempts to establish a connection to the callback client via one of these proxies, it might arbitrarily select one of the router's server endpoints to try first. Since the router's server endpoints use addresses in the same network, the local client attempts to make a connection over the local network, with two possible outcomes: the connection attempts to those endpoints fail, in which case they are skipped and the real local endpoints are attempted; or, even worse, one of the endpoints might accidentally be valid in the local network, in which case the local client has just connected to some unknown server.
2. The server may encounter similar problems when attempting to establish a local connection to the router in order to make a callback request.

The solution is to dedicate an object adapter solely to handling callback requests, and another one for servicing local clients.

In order to use multiple object adapters in this fashion, you must not define the `Ice.Default.Router` property (see Section 24.4.5). Assuming our callback object adapter is named `CallbackAdapter`, we configure the adapter with a router using the following property:

```
CallbackAdapter.Router=Glacier/router -h 5.6.7.8 -p 8000
```

Of course, by removing the `Ice.Default.Router` property, our client proxies are no longer automatically configured with a router. There are a couple of ways to solve this:

1. Invoke the `ice_router` operation on each proxy we need routed.
2. Use multiple communicators. For example, create one communicator that is configured with the `Ice.Default.Router` proxy, and use that communicator to create the callback object adapter and all routed proxies. Then create another communicator for the local object adapter.

24.4.8 Using Multiple Routers

A client is not limited to using only one router at a time: the proxy operation `ice_router` allows a client to configure its routed proxies as necessary. With respect to callbacks, the Ice core is designed to allow a single object adapter to accept callbacks from multiple routers, but that capability is not currently supported in the Glacier implementation. Therefore, a callback object adapter must be created for each router that can forward callback requests to the application.

24.4.9 Example

The `demo/Ice/callback` example can be used to test callbacks via Glacier. The example's configuration file already contains the relevant property definitions for using a Glacier router, but the file may require editing to enable some properties that are commented out by default. See the comments in the configuration file for more information.

24.5 The Glacier Starter

A Glacier router is capable of routing requests from multiple clients to multiple servers, but only when no callbacks are required (see Section 24.3). If callbacks

are necessary, each client must use its own Glacier router. Ice provides² the Glacier “starter” to launch routers on demand, as shown in Figure 24.8.

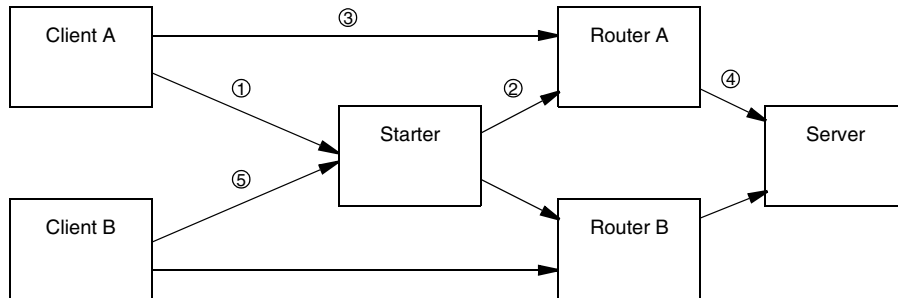


Figure 24.8. Using the Glacier starter.

1. Client A issues a request to the starter, which is listening on a well-known endpoint.
2. The starter spawns a new instance of the router, Router A, and returns the router’s proxy to the client.
3. Client A configures its proxies with the router, and makes an invocation via a routed proxy. From this point on, Client A interacts only with Router A, not the starter.
4. Router A forwards the request to the server.
5. The process is repeated for Client B.

As you can see, the Glacier starter is a very convenient way of solving the router-per-client issue. For security considerations, see Section 24.6.

24.5.1 Configuring the Starter

The starter is a regular Ice server and supports a number of configuration properties, as described in Appendix C. A minimal configuration is shown below:

2. The Glacier starter is currently only provided for Unix platforms.

```
Glacier.Starter.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Starter.RouterPath=/opt/Ice/bin/glacierrouter
Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60
```

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
```

The `Glacier.Starter.Endpoints` property defines the well-known endpoint for the starter. This is the endpoint that a client uses in its starter proxy (see Section 24.5.3).

The `Glacier.Starter.RouterPath` property supplies the pathname of the router executable. If this property is not defined, the default value is `glacierrouter`, implying that the router executable must reside in the starter's executable search path.

The most interesting property is `Glacier.Starter.PropertiesOverride`. A router that is launched by a starter inherits the starter's configuration properties, which is why we included the router's properties in the starter's configuration. However, you may want to override one of the starter's properties, or you may want to augment the properties specifically for the router without applying them to the starter. The `PropertiesOverride` property allows you to do this.

In this example, the starter defines the `Ice.ServerIdleTime` property, which forces a server to terminate gracefully if there is no connection activity for a certain period of time. We do not set this property on the starter because we want it to remain available for new client requests. However, this is a very useful property for the router because it avoids the accumulation of inactive router processes whose clients have terminated or crashed. We therefore use the `PropertiesOverride` property to define the `Ice.ServerIdleTime` property only for the router.

Finally, we define the router's endpoints. These endpoints do not use fixed ports because each router is launched for use by a single client and therefore must use system-assigned ports. If we were to specify fixed ports, then only one router at a time would be able to run.

24.5.2 Starting the Starter

Assuming the configuration properties shown in Section 24.5.1 are stored in a file named `config`, the starter can be started with the following command:

```
$ glacierstarter --Ice.Config=config
```

The starter can also be run as a Unix daemon or Win32 service. See Section 10.3.2 for more information.

24.5.3 Using the Starter

A client can obtain a router by invoking the `startRouter` operation on the `Glacier::Starter` interface.

C++ Example

The example code below illustrates how a C++ client obtains a router.

```
#include <Ice/Ice.h>
#include <Glacier/Glacier.h>

// ...

Ice::CommunicatorPtr communicator = // ...
Ice::ObjectAdapterPtr adapter = // ...

std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);

Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);

communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

The code begins by creating a proxy for the starter. The endpoint in this proxy matches the one we configured using the `Glacier.Starter.Endpoints` property in Section 24.5.1.

```
std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);
```

Next, we invoke the `startRouter` operation, passing values for the user and password arguments. (See Section 24.6 for more information on starter security.)

```
Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);
```

Finally, we configure the communicator and the object adapter with the router proxy we received from the starter.

```
communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

At this point, the communicator is ready to create routed proxies, and the object adapter is ready to receive callback requests.

Java Example

The example code below illustrates how a Java client obtains a router.

```
Ice.Communicator communicator = // ...
Ice.ObjectAdapter adapter = // ...

String ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice.ObjectPrx obj = communicator.stringToProxy(ref);
Glacier.StarterPrx starter =
    Glacier.StarterPrxHelper.checkedCast(obj);

Ice.ByteSeqHolder privateKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder publicKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder routerCert = new Ice.ByteSeqHolder();

Glacier.RouterPrx router =
    starter.startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);

communicator.setDefaultRouter(router);
adapter.addRouter(router);
```

The code begins by creating a proxy for the starter. The endpoint in this proxy matches the one we configured using the `Glacier.Starter.Endpoints` property in Section 24.5.1.

```
String ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice.ObjectPrx obj = communicator.stringToProxy(ref);
Glacier.StarterPrx starter =
    Glacier.StarterPrxHelper.checkedCast(obj);
```

Next, we invoke the `startRouter` operation, passing values for the user and password arguments. (See Section 24.6 for more information on starter security.)

```
Ice.ByteSeqHolder privateKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder publicKey = new Ice.ByteSeqHolder();
Ice.ByteSeqHolder routerCert = new Ice.ByteSeqHolder();

Glacier.RouterPrx router =
    starter.startRouter("user", "passwd", privateKey, publicKey,
                       routerCert);
```

Finally, we configure the communicator and the object adapter with the router proxy we received from the starter.

```
communicator.setDefaultRouter(router);
adapter.addRouter(router);
```

At this point, the communicator is ready to create routed proxies, and the object adapter is ready to receive callback requests.

24.6 Glacier Security

As a firewall, a Glacier router represents a doorway into a private network, and in most cases that doorway should have a good lock. The obvious first step is to use SSL for the router control, router client, and starter endpoints. This allows you to secure the message traffic and restrict access to clients having the proper credentials (see Chapter 23). However, the Glacier starter takes security one step further by providing even stricter access control capabilities. We saw evidence of these capabilities in the sample code presented in Section 24.5.3, and this section explores them in depth.

24.6.1 Access Control

For some applications, the authentication capabilities of SSL are not sufficient to restrict access to the firewall. The certificate validation phase of the SSL handshake verifies that the user is who he says he is, but how do we know that he should be given passage through the firewall?

The Glacier starter addresses this issue through the use of an access control facility. The first two arguments to the `startRouter` operation are a username and password, as shown in Section 24.5.3. These arguments are verified by the starter

before it launches a router for the client. (It is safe for the client to send these arguments “in the clear” because the client connects to the starter via SSL.)

The starter supports two forms of username/password verification: a file-based access control list, or a custom verifier implementation. Configuration properties determine which form of verification a particular starter uses; file-based verification is used by default.

Crypt File

The file-based access control list uses the `crypt` algorithm to protect the passwords, similar to the `passwd` file on a typical Unix system. Each line of the file must contain a username and password, separated by white space. The password must be a 13-character, `crypt`-encoded string. For example, the file used by the test suite in `test/Glacier/starter/passwords` contains the following line:

```
dummy xxMqsnnDcK8tw
```

You can obtain the `crypt` encoding for a password using the `openssl` utility provided with OpenSSL:

```
$ openssl passwd
```

The program prompts you for a password and generates the `crypt`-encoded version for use in the password file.

The `Glacier.Starter.CryptPasswords` property defines the name of the file. If not specified, the default name is `passwords`.

Permissions Verifier

An application that has special requirements can implement the interface `Glacier:PermissionsVerifier` to have programmatic control over access to a starter. This can be especially useful in situations where a repository of account information already exists (such as an LDAP directory), in which case duplicating that information in another file would be tedious and error prone.

The `Slice` definition for the interface contains just one operation:

```
module Glacier {
  interface PermissionsVerifier {
    nonmutating
    bool checkPermissions(string userId, string password,
                          out string reason);
  };
};
```


The starter invokes `checkPermissions` on the verifier object, passing it the username and password arguments that were given to `startRouter`. The operation must return `true` if the arguments are valid, `false` otherwise. If the operation returns `false`, a reason can be provided in the output parameter.

To configure a starter with a custom verifier, set the configuration property `Glacier.Starter.PermissionsVerifier` with the proxy for the object. Since the username and password are sent in the clear, it is advisable to use an SSL endpoint for the verifier.

24.6.2 Filtering

The Glacier router is capable of filtering requests based on object identity, which helps to ensure that clients do not gain access to unintended objects.

As you may recall, the `Ice.Identity` type contains two string members: `category` and `name`. A router can be configured with a list of valid identity categories, in which case it only routes requests for objects in those categories. The configuration property `Glacier.Router.AllowCategories` supplies the category list:

```
Glacier.Router.AllowCategories=cat1 cat2
```

The starter also plays a role in this filtering capability. The configuration property `Glacier.Starter.AddUserToAllowCategories` controls whether the starter, when launching a router, adds the verified username to the router's list of valid categories. This feature accommodates applications in which Ice objects are created for use by a particular client and must not be accessible to other clients. Since the starter is creating a new router for each client, it is in the ideal position to supply a client-specific category (i.e., the username) to the router. This strategy naturally assumes the server is cooperating by using the username as the identity category of the Ice objects it creates for the client. The properties shown below demonstrate how to configure the starter and router for filtering:

```
Glacier.Starter.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Starter.RouterPath=/opt/Ice/bin/glacierrouter
Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60
Glacier.Starter.AddUserToAllowCategories=1
```

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
Glacier.Router.Server.Endpoints=tcp -h 10.0.0.1
Glacier.Router.AllowCategories=Factory
```

Using this configuration, a router created for user `duncanfoo` will allow requests for the identity categories `Factory` and `duncanfoo`. If that client makes a request for any other identity category, it receives an `Ice::ObjectNotExistException`.

24.6.3 Securing the Router

In addition to the access control and filtering features discussed in the previous sections, the Glacier starter takes one more step to secure the router. When the starter is configured for IceSSL and a new client is successfully verified, the starter generates a key pair for the client and a key pair for the router. The starter supplies the client's public key to the router, and the router's public key to the client. The router and client then configure their respective IceSSL plug-ins with their peer's public key, in order to ensure a connection is established only with that peer.

The code for properly configuring the IceSSL plug-in in a client is shown below. We include the steps for invoking `startRouter`, although they are identical to the examples presented in Section 24.5.3 and therefore are not described again. The goal of this example is to demonstrate how to use the three out arguments returned by `startRouter`.

```
#include <Ice/Ice.h>
#include <Glacier/Glacier.h>
#include <IceUtil/Base64.h>
#include <IceSSL/Plugin.h>

// ...

Ice::CommunicatorPtr communicator = // ...
Ice::ObjectAdapterPtr adapter = // ...

std::string ref = "Glacier/starter:default -h 5.6.7.8 -p 8000";
Ice::ObjectPrx obj = communicator->stringToProxy(ref);
Glacier::StarterPrx starter =
    Glacier::StarterPrx::checkedCast(obj);

Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                        routerCert);
```

```

Ice::PropertiesPtr properties = communicator->getProperties();

string cliCfg = properties->getProperty("IceSSL.Client.Config");
string srvCfg = properties->getProperty("IceSSL.Server.Config");

if (!cliCfg.empty() && !srvCfg.empty())
{
    string privateKeyBase64 = IceUtil::Base64::encode(privateKey);
    string publicKeyBase64  = IceUtil::Base64::encode(publicKey);
    string routerCertString = IceUtil::Base64::encode(routerCert);

    Ice::PluginManagerPtr pluginManager =
        communicator->getPluginManager();
    Ice::PluginPtr plugin = pluginManager->getPlugin("IceSSL");
    IceSSL::PluginPtr sslPlugin =
        IceSSL::PluginPtr::dynamicCast(plugin);

    sslPlugin->configure(IceSSL::Server);
    sslPlugin->setCertificateVerifier(
        IceSSL::ClientServer,
        sslPlugin->getSingleCertVerifier(routerCert));

    sslPlugin->setRSAKeysBase64(IceSSL::ClientServer,
                               privateKeyBase64,
                               publicKeyBase64);
    sslPlugin->addTrustedCertificateBase64(IceSSL::ClientServer,
                                           routerCertString);
}

communicator->setDefaultRouter(router);
adapter->addRouter(router);

```

After invoking `startRouter`, the client determines whether it is configured for IceSSL by checking for the IceSSL configuration properties. If both properties are defined, the client continues with the plug-in configuration.

The values returned by `startRouter` are converted to a Base64 encoding, which is the format required by IceSSL:

```

string privateKeyBase64 = IceUtil::Base64::encode(privateKey);
string publicKeyBase64  = IceUtil::Base64::encode(publicKey);
string routerCertString = IceUtil::Base64::encode(routerCert);

```

Next, the client obtains a reference to the IceSSL plug-in using the technique described in Chapter 23.

```
Ice::PluginManagerPtr pluginManager =  
    communicator->getPluginManager();  
Ice::PluginPtr plugin = pluginManager->getPlugin("IceSSL");  
IceSSL::PluginPtr sslPlugin =  
    IceSSL::PluginPtr::dynamicCast(plugin);
```

The client then manually configures the server component of the IceSSL plug-in, which causes the plug-in to load the IceSSL server configuration file, and then creates a certificate verifier using the router's public key.

```
sslPlugin->configure(IceSSL::Server);  
sslPlugin->setCertificateVerifier(  
    IceSSL::ClientServer,  
    sslPlugin->getSingleCertVerifier(routerCert));
```

Finally, the client gives its new RSA key pair to the plug-in, and then adds the router's public key as a trusted certificate.

```
sslPlugin->setRSAKeysBase64(IceSSL::ClientServer,  
    privateKeyBase64,  
    publicKeyBase64);  
sslPlugin->addTrustedCertificateBase64(IceSSL::ClientServer,  
    routerCertString);
```

24.7 Summary

Complex network environments are a fact of life. Unfortunately, the cost of securing an enterprise's network is increased application complexity and administrative overhead. Glacier helps to minimize these costs by providing a low-impact, efficient and secure router-firewall for Ice applications.

Chapter 25

IceBox

25.1 Chapter Overview

In this chapter we present IceBox, an easy-to-use framework for Ice application services. Section 25.2 provides an overview of IceBox and the advantages of using it. Section 25.3 introduces the service manager and describes its role in IceBox. A tutorial on writing and configuring an IceBox service is presented in Section 25.4, and Section 25.5 puts all the pieces together.

25.2 Introduction

The Service Configurator pattern [7] is a useful technique for configuring services and centralizing their administration. In practical terms, this means services are developed as dynamically-loadable components that can be configured into a general purpose “super server” in whatever combinations are necessary. IceBox is an implementation of the Service Configurator pattern for Ice services.

A generic IceBox server replaces the typical monolithic Ice server you normally write. The IceBox server is configured via properties with the application-specific services it is responsible for loading and managing, and it can be administered remotely. There are several advantages in using this architecture:

- Services loaded by the same IceBox server can be configured to take advantage of Ice's collocation optimizations. For example, if one service is a client of another service, and those services reside in the same IceBox server, then invocations between them can be optimized.
- Composing an application consisting of various services is done by configuration, not by compiling and linking. This decouples the service from the server, allowing services to be combined or separated as needed.
- Multiple Java services can be active in a single instance of a Java Virtual Machine (JVM). This conserves operating system resources when compared to running several monolithic servers, each in its own JVM.
- Services implement an IceBox service interface, providing a common framework for developers and a centralized administrative facility.
- IceBox support is integrated into IcePack, the server activation and deployment service (see Chapter 20).

25.3 The Service Manager

In addition to the objects supported by application services, an IceBox server supports an administrative object that implements the `IceBox: : ServiceManager` interface (see Appendix B). This object is responsible for loading and initializing the services, as well as performing administrative actions requested by clients. An object adapter is created for this object so that its endpoint(s) can be secured against unauthorized access.

Currently, the administrative capabilities are rather limited: the only supported operation on the `ServiceManager` interface is shutdown, which terminates the services and shuts down the IceBox server. Additional administrative features may be added in a future release.

25.3.1 Endpoint Configuration

The endpoints for the service manager's object adapter are defined using the `IceBox.ServiceManager.Endpoints` configuration property:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
```

Since a malicious client could make a denial-of-service attack against the service manager, it is advisable to use SSL endpoints (see Chapter 23), or to secure these endpoints with the proper firewall configuration, or both.

25.3.2 Client Configuration

A client requiring administrative access to the service manager can create a proxy using the endpoints defined by the property described in Section 25.3.1. The default identity of the service manager object is `ServiceManager`, but this can be changed using the `IceBox.ServiceManager.Identity` property (see Appendix C). Therefore, a proxy using the default identity and the endpoint from Section 25.3.1 would be constructed as follows:

```
ServiceManager.Proxy=ServiceManager:tcp -p 10000
```

25.3.3 Administrative Utility

An administrative utility is included with IceBox. Given the limited capabilities of the `ServiceManager` interface in its present form, it should come as no surprise that the administrative utility also has limited functionality. IceBox provides C++ and Java implementations of the utility, with the same usage:

```
Usage: iceboxadmin [options] [command...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.

Commands:
shutdown            Shutdown the server.
```

The C++ utility is named `iceboxadmin`, while the Java utility is represented by the class `IceBox.Admin`. Both use the `IceBox.ServiceManager.Endpoints` property described in Section 25.3.1 to create the proxy for the service manager object. Typically, the utility is started using the same configuration file as the IceBox server (see Section 25.5), but that is not mandatory.

25.4 Developing a Service

Writing an IceBox service requires implementing one of the IceBox service interfaces. The C++ and Java sample implementations we present in this section implement `IceBox::Service`, shown below:

```
module IceBox {
    local interface ServiceBase {
        void stop();
    };
};
```

```

local interface Service extends ServiceBase {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args)
        throws FailureException;
};
};

```

As you can see, a service needs to implement only two operations, start and stop. These operations are invoked by the service manager; start is called after the service is loaded, and stop is called when the IceBox server is shutting down.

The start operation is the service's opportunity to initialize itself; this typically includes creating an object adapter and servants. The name and args parameters supply information from the service's configuration (see Section 25.4.3), and the communicator parameter is an `Ice::Communicator` object created by the service manager for use by the service. Depending on the service configuration, this communicator instance may be shared by other services in the same IceBox server, therefore care should be taken to ensure that items such as object adapters are given unique names.

The stop operation must reclaim any resources used by the service. Generally, a service deactivates its object adapter, and may also need to invoke `waitForDeactivate` on the object adapter in order to ensure that all pending requests have been completed before the clean up process can proceed. The service manager is responsible for destroying the communicator.

These interfaces are declared as `local` for a reason: they represent a contract between the service manager and the service, and are not intended to be used by remote clients. Any interaction the service has with remote clients is done via servants created by the service.

25.4.1 C++ Service Example

The example we present here is taken from the `demo/IceBox/hello` sample program provided in the Ice distribution.

The class definition for our service is quite straightforward, but there are a few aspects worth mentioning:

```

#include <IceBox/IceBox.h>

#ifdef _WIN32
#    define HELLO_API __declspec(dllexport)

```



```

#else
#   define HELLO_API /**/
#endif

class HELLO_API HelloServiceI : public IceBox::Service {
public:
    virtual void start(const std::string &,
                      const Ice::CommunicatorPtr &,
                      const Ice::StringSeq &);
    virtual void stop();

private:
    Ice::ObjectAdapterPtr _adapter;
};

```

First, we include the IceBox header file so that we can derive our implementation from IceBox::Service.

Second, the preprocessor definitions are necessary because, on Windows, this service resides in a Dynamic Link Library (DLL), therefore we need to export the class so that the service manager can load it properly.

The member definitions are equally straightforward:

```

#include <Ice/Ice.h>
#include <HelloServiceI.h>
#include <HelloI.h>

using namespace std;

extern "C" {
    HELLO_API IceBox::Service *
    create(Ice::CommunicatorPtr communicator)
    {
        return new HelloServiceI;
    }
}

void
HelloServiceI::start(
    const string & name,
    const Ice::CommunicatorPtr & communicator,
    const Ice::StringSeq & args)
{
    _adapter = communicator->createObjectAdapter(name);
    Ice::ObjectPtr object = new HelloI(communicator);
    _adapter->add(object, Ice::stringToIdentity("hello"));
}

```

```

        _adapter->activate();
    }

    void
    HelloServiceI::stop()
    {
        _adapter->deactivate();
    }

```

You might be wondering about the `create` function we defined. This is the *entry point* for a C++ IceBox service; that is, this function is used by the service manager to obtain an instance of the service, therefore it must have a particular signature. The name of the function is not important, but the function is expected to take a single argument of type `Ice::CommunicatorPtr`, and return a pointer to `IceBox::Service`¹. In this case, we simply return a new instance of `HelloServiceI`. See Section 25.4.3 for more information on entry points.

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

This is obviously a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the code into a shared library or DLL, it can be configured into an IceBox server as described in Section 25.4.3.

25.4.2 Java Service Example

As with the C++ example presented in the previous section, the complete source for the Java example can be found in the `demo/IceBox/hello` directory of the Ice distribution. The class definition for our service looks as follows:

```

public class HelloServiceI extends Ice.LocalObjectImpl
    implements IceBox.Service
{
    public void
    start(String name,
          Ice.Communicator communicator,
          String[] args)
        throws IceBox.FailureException

```

1. A function with C linkage cannot return an object type, such as a smart pointer, therefore the entry point must return a regular pointer value.

```
{
    _adapter = communicator.createObjectAdapter(name);
    Ice.Object object = new HelloI(communicator);
    _adapter.add(object, Ice.Util.stringToIdentity("hello"));
    _adapter.activate();
}

public void
stop()
{
    _adapter.deactivate();
}

private Ice.ObjectAdapter _adapter;
}
```

First, notice that our class extends `Ice.LocalObjectImpl`. This is one of the few occasions when an Ice developer must implement a local interface (other common cases are object factories and servant locators).

The `start` method creates an object adapter with the same name as the service, activates a single servant of type `HelloI` (not shown), and activates the object adapter. The `stop` method simply deactivates the object adapter.

The service manager requires a service implementation to have a default constructor. This is the *entry point* for a Java IceBox service; that is, the service manager dynamically loads the service implementation class and invokes the default constructor to obtain an instance of the service.

This is obviously a trivial service, and yours will likely be much more interesting, but this does demonstrate how easy it is to write an IceBox service. After compiling the service implementation class, it can be configured into an IceBox server as described in Section 25.4.3.

25.4.3 Configuring a Service

A service is configured into an IceBox server using a single property. This property serves several purposes: it defines the name of the service, it provides the service manager with the service entry point, and it defines properties and arguments for the service.

The format of the property is shown below:

```
IceBox.Service.name=entry_point [args]
```

The *name* component of the property key is the service name. This name is passed to the service's `start` operation, and must be unique among all services

configured in the same IceBox server. It is possible, though rarely necessary, to load two or more instances of the same service under different names.

The first argument in the property value is the entry point specification. For C++ services, this must have the form `library:symbol`, where `library` is the simple name of the service shared library or DLL, and `symbol` is the name of the entry point function. By simple name, we mean a name without any platform-specific prefixes or extensions; the service manager appends the appropriate decorations depending on the platform. For example, the simple name `MyService` results in a DLL named `MyService.dll` on Windows, and a shared library named `libMyService.so` on Linux. The shared library or DLL must reside in a directory that appears in `PATH` on Windows or `LD_LIBRARY_PATH` on POSIX systems.

For Java services, the entry point is simply the complete class name (including any package) of the service implementation class. The class must reside in the class path of the IceBox server.

Any arguments following the entry point specification are examined. If an argument has the form `--name=value`, then it is interpreted as a property definition that appears in the property set of the communicator passed to the service start operation. These arguments are removed, and any remaining arguments are passed to the start operation in the `args` parameter.

C++ Example

Here is an example of a configuration for our C++ example from Section 25.4.1:

```
IceBox.Service.Hello=HelloService:create \  
  --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in `HelloService.dll` on Windows or `libHelloService.so` on Linux, and the entry point function `create` is invoked to create an instance of the service. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Java Example

Here is an example of a configuration for our Java example from Section 25.4.2:

```
IceBox.Service.Hello=HelloServiceI \  
  --Ice.Trace.Network=1 hello there
```

This configuration results in the creation of a service named `Hello`. The service is expected to reside in the class `HelloServiceI`. The argument `--Ice.Trace.Network=1` is converted into a property definition, and the arguments `hello` and `there` become the two elements in the `args` sequence parameter that is passed to the `start` method.

Freeze Configuration

A Freeze service supports an additional configuration property that defines the pathname of the directory in which the service manager creates the Freeze database environment:

```
IceBox.DBEnvName.name=path
```

The `name` component of the property key is the service name.

Sharing a Communicator

The service manager can be configured to share a single communicator instance with all of its services using the following property:

```
IceBox.UseSharedCommunicator=1
```

The default behavior if this property is not specified is to create a new communicator instance for each service. However, if collocation optimizations between services are desired, then a shared communicator instance is required.

Loading Services

By default, the service manager loads the configured services in an undefined order, meaning services in the same `IceBox` server should not depend on one another. If services must be loaded in a particular order, the `IceBox.LoadOrder` property can be used:

```
IceBox.LoadOrder=Service1,Service2
```

In this example, `Service1` is loaded first, followed by `Service2`. Any remaining services are loaded after `Service2`, in an undefined order. Each service mentioned in `IceBox.LoadOrder` must have a matching `IceBox.Service` property.

25.4.4 Freeze Service

`IceBox` provides another interface for services that use Freeze (see Chapter 21):

```

module IceBox {
local interface FreezeService extends ServiceBase {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args,
               Freeze::DBEnvironment dbEnv)
    throws FailureException;
};
};

```

By implementing `FreezeService` instead of `Service`, you have implicitly informed the service manager that it should create a `Freeze` database environment for this service and supply it to the `start` operation. The pathname of the database environment directory is specified using a configuration property, as described in Section 25.4.3. The service manager owns the database environment, and is responsible for destroying it after invoking `stop` on the service.

25.5 Starting IceBox

Incorporating everything we discussed in the previous sections, we can now configure and start IceBox servers.

25.5.1 Starting the C++ Server

The configuration file for our example C++ service is shown below:

```

IceBox.ServiceManager.Endpoints=tcp -p 10000
IceBox.Service.Hello=HelloService:create
Hello.Endpoints=tcp -p 10001

```

Notice that we define an endpoint for the object adapter created by the `Hello` service.

Assuming these properties reside in a configuration file named `config`, we can start the C++ IceBox server as follows:

```
$ icebox --Ice.Config=config
```

25.5.2 Starting the Java Server

Our Java configuration is nearly identical to the C++ version, except for the entry point specification:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000  
IceBox.Service.Hello=HelloServiceI  
Hello.Endpoints=tcp -p 10001
```

Assuming these properties reside in a configuration file named `config`, we can start the Java IceBox server as follows:

```
$ java IceBox.Server --Ice.Config=config
```

25.5.3 Initialization Failure

At startup, an IceBox server inspects its configuration for all properties having the prefix `IceBox.Service.` and initializes each service. If initialization fails for a service, the IceBox server invokes the `stop` operation on any initialized services, reports an error, and terminates.

25.6 Summary

IceBox offers a refreshing change of perspective: developers focus on writing services, not applications. The definition of an application changes as well; using IceBox, an application becomes a collection of discrete services whose composition is determined dynamically by configuration, rather than statically by the linker.

Chapter 26

IceStorm

26.1 Chapter Overview

In this chapter we present IceStorm, an efficient publish/subscribe service for Ice applications. Section 26.2 provides an introduction to IceStorm, while Section 26.3 discusses some basic IceStorm concepts. An overview of the IceStorm Slice interfaces is provided in Section 26.4, and Section 26.5 presents an example IceStorm application. The IceStorm administration tool is described in Section 26.6, and the subject of federation is discussed in Section 26.7. IceStorm's quality of service parameters are defined in Section 26.8. Finally, IceStorm configuration is addressed in Section 26.9.

26.2 Introduction

Applications often need to disseminate information to multiple recipients. For example, suppose we are developing a weather monitoring application in which we collect measurements such as wind speed and temperature from a meteorolog-

ical tower and periodically distribute them to weather monitoring stations. We initially consider using the architecture shown in Figure 26.1.

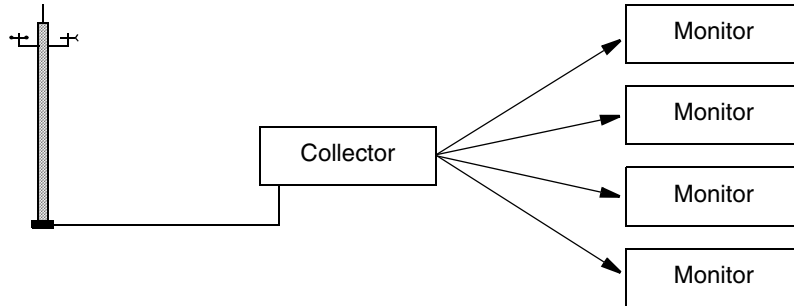


Figure 26.1. Initial design for a weather monitoring application.

However, the primary disadvantage of this architecture is that it tightly couples the collector to its monitors, needlessly complicating the collector implementation by requiring it to manage the details of monitor registration, measurement delivery, and error recovery. We can rid ourselves of these mundane duties by incorporating IceStorm into our architecture, as shown in Figure 26.2.

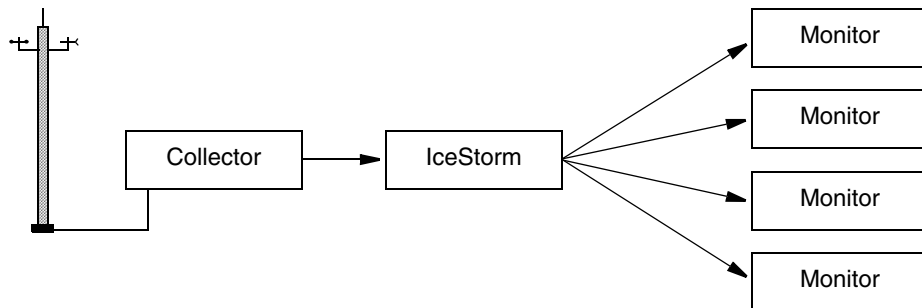


Figure 26.2. A weather monitoring application using IceStorm.

IceStorm simplifies the collector implementation significantly by decoupling it from the monitors. As a publish/subscribe service, IceStorm acts as a mediator between the collector (the publisher) and the monitors (the subscribers), and offers several advantages:

- When the collector is ready to distribute a new set of measurements, it makes a single request to the IceStorm server. The IceStorm server takes responsibility for delivering the request to the monitors, including handling any exceptions caused by ill-behaved or missing subscribers. The collector no longer needs to be aware of its monitors, or whether it even has any monitors at that moment.
- Similarly, monitors interact with the IceStorm server to perform tasks such as subscribing and unsubscribing, thereby allowing the collector to focus on its application-specific responsibilities and not on administrative trivia.
- The collector and monitor applications require very few changes to incorporate IceStorm.

26.3 Concepts

This section discusses several concepts that are important for understanding IceStorm's capabilities.

26.3.1 Message

An IceStorm *message* is strongly typed and is represented by an invocation of a Slice operation: the operation name identifies the type of the message, and the operation parameters define the message contents. A message is published by invoking the operation on an IceStorm proxy in the normal fashion. Similarly, subscribers receive the message as a regular servant upcall. As a result, IceStorm uses the “push” model for message delivery; polling is not supported.

26.3.2 Topic

An application indicates its interest in receiving messages by subscribing to a *topic*. An IceStorm server supports any number of topics, which are created dynamically and distinguished by unique names. Each topic can have multiple publishers and subscribers.

A topic is essentially equivalent to an application-defined Slice interface: the operations of the interface define the types of messages supported by the topic. A publisher uses a proxy for the topic interface to send its messages, and a subscriber implements the topic interface (or an interface derived from the topic interface) in order to receive the messages. This is no different than if the

publisher and subscriber were communicating directly in the traditional client-server style; the interface represents the contract between the client (the publisher) and the server (the subscriber), except IceStorm transparently forwards each message to multiple recipients.

IceStorm does not verify that publishers and subscribers are using compatible interfaces, therefore applications must ensure that topics are used correctly.

26.3.3 Oneway

IceStorm messages have oneway semantics (see Section 2.2.2), therefore a publisher cannot receive replies from its subscribers. Naturally, all of the Ice transports (TCP, SSL, and UDP) can be used to publish and receive messages.

26.3.4 Federation

IceStorm supports the formation of topic graphs, also known as federation. A topic graph is formed by creating links between topics, where a *link* is a unidirectional association from one topic to another. Each link has a *cost* that may restrict message delivery on that link (see Section 26.7.2). A message published on a topic is also published on all of the topic's links for which the message cost does not exceed the link cost.

Once a message has been published on a link, the receiving topic publishes the message to its subscribers, but does not publish it on any of its links. In other words, IceStorm messages propagate at most one hop from the originating topic in a federation (see Section 26.7.1).

Figure 26.3 presents an example of topic federation. Topic T_1 has links to T_2 and T_3 , as indicated by the arrows. The subscribers S_1 and S_2 receive all messages

published on T_2 , as well as those published on T_1 . Subscriber S_3 receives messages only from T_1 , and S_4 receives messages from both T_3 and T_1 .

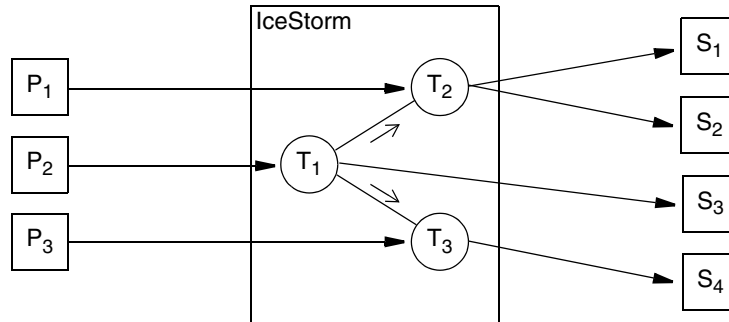


Figure 26.3. Topic federation.

IceStorm makes no attempt to prevent a subscriber from receiving duplicate messages. For example, if a subscriber is subscribed to both T_2 and T_3 , then it would receive two requests for each message published on T_1 .

26.3.5 Quality of Service

IceStorm allows each subscriber to specify its own *quality of service* (QoS) parameters that affect the delivery of its messages. Quality of service parameters are represented as a dictionary of name-value pairs. The supported QoS parameters are described in Section 26.8.

26.3.6 Persistence

IceStorm has a database in which it maintains information about its topics and links. However, a message sent via IceStorm is not stored persistently, but rather is discarded as soon as it is delivered to the topic's current set of subscribers. If an error occurs during delivery to a subscriber, IceStorm does not queue messages for that subscriber.

26.3.7 Subscriber Errors

Since IceStorm messages are delivered with oneway semantics, the only errors IceStorm can detect are connection or timeout failures. If IceStorm encounters

such a failure while attempting to deliver a message to a subscriber, the subscriber is immediately unsubscribed from the topic on which the message was published.

Slow subscribers can also create problems. IceStorm attempts to deliver messages as soon as they are published: when the IceStorm publisher object receives a message, it may immediately make nested invocations to the topic's subscribers. Therefore, if a subscriber is not consuming messages as fast as they are being published, it may cause threads to accumulate in IceStorm. If the number of threads reaches the maximum size of the thread pool, then no new messages can be published. See Chapter 15 for more information on Ice thread pools.

26.4 IceStorm Interface Overview

This section provides a brief introduction to the Slice interfaces comprising the IceStorm service. See Appendix B for the Slice documentation.

26.4.1 TopicManager

The TopicManager is a singleton object that acts as a factory and repository of Topic objects. Its interface and related types are shown below:

```
module IceStorm {
    dictionary<string, Topic*> TopicDict;

    exception TopicExists {
        string name;
    };

    exception NoSuchTopic {
        string name;
    };

    interface TopicManager {
        Topic* create(string name) throws TopicExists;
        nonmutating Topic* retrieve(string name) throws NoSuchTopic;
        nonmutating TopicDict retrieveAll();
    };
};
```

The create operation is used to create a new topic, which must have a unique name. The retrieve operation allows a client to obtain a proxy for an existing topic, and retrieveAll supplies a dictionary of all existing topics.

26.4.2 Topic

The Topic interface represents a topic and provides several administrative operations for configuring links and managing subscribers.

```
module IceStorm {
    struct LinkInfo {
        Topic* theTopic;
        string name;
        int cost;
    };
    sequence<LinkInfo> LinkInfoSeq;

    dictionary<string, string> QoS;

    exception LinkExists {
        string name;
    };

    exception NoSuchLink {
        string name;
    };

    interface Topic {
        nonmutating string getName();
        nonmutating Object* getPublisher();
        void subscribe(QoS theQoS, Object* subscriber);
        idempotent void unsubscribe(Object* subscriber);
        idempotent void link(Topic* linkTo, int cost)
            throws LinkExists;
        idempotent void unlink(Topic* linkTo) throws NoSuchLink;
        nonmutating LinkInfoSeq getLinkInfoSeq();
        void destroy();
    };
};
```

The getName operation returns the name assigned to the topic, while the getPublisher operation returns a proxy for the topic's publisher object (see Section 26.5.2).

The `subscribe` operation adds a subscriber's proxy to the topic; if another subscriber proxy already exists with the same object identity, the subscriber's proxy is replaced with the new one. The `unsubscribe` operation removes a subscriber from the topic.

A link to another topic is created using the `link` operation; if a link already exists to the given topic, the `LinkExists` exception is raised. Links are destroyed using the `unlink` operation.

Finally, the `destroy` operation permanently destroys the topic.

26.5 Using IceStorm

In this section we expand on the weather monitoring example from Section 26.2, demonstrating how to create, subscribe to and publish messages on a topic. We use the following Slice definitions in our example:

```
struct Measurement {
    string tower; // tower id
    float windSpeed; // knots
    short windDirection; // degrees
    float temperature; // degrees Celsius
};

interface Monitor {
    void report(Measurement m);
};
```

`Monitor` is our topic interface. For the sake of simplicity, it defines just one operation, `report`, taking a `Measurement` struct as its only parameter.

26.5.1 Implementing a Publisher

The implementation of our collector application can be summarized easily:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Obtain a proxy for the `Weather` topic, either by creating the topic if it does not exist, or retrieving the proxy for the existing topic.
3. Obtain a proxy for the `Weather` topic's "publisher object." This proxy is provided for the purpose of publishing messages, and therefore is narrowed to the topic interface (`Monitor`).

4. Collect and report measurements.

In the sections below, we present collector implementations in C++ and Java.

C++ Example

As usual, our C++ example begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown above.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);
    IceStorm::TopicPrx topic;
    try {
        topic = topicManager->retrieve("Weather");
    }
    catch (const IceStorm::NoSuchTopic&) {
        topic = topicManager->create("Weather");
    }

    Ice::ObjectPrx pub = topic->getPublisher();
    if (!pub->ice_isDatagram())
        pub = pub->ice_oneway();
    MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
    while (true) {
        Measurement m = getMeasurement();
        monitor->report(m);
    }
    ...
}
```

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```
IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
}
catch (const IceStorm::NoSuchTopic&) {
    topic = topicManager->create("Weather");
}
```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the Monitor interface.

```
Ice::ObjectPrx pub = topic->getPublisher();
if (!pub->ice_isDatagram())
    pub = pub->ice_oneway();
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```
while (true) {
    Measurement m = getMeasurement();
    monitor->report(m);
}
```

Java Example

The equivalent Java version is shown below.

```
public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);
    IceStorm.TopicPrx topic = null;
    try {
        topic = topicManager.retrieve("Weather");
    }
    catch (IceStorm.NoSuchTopic ex) {
        topic = topicManager.create("Weather");
    }

    Ice.ObjectPrx pub = topic.getPublisher();
    if (!pub.ice_isDatagram())
        pub = pub.ice_oneway();
    MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
    while (true) {
```

```

        Measurement m = getMeasurement();
        monitor.report(m);
    }
    ...
}

```

After obtaining a proxy for the topic manager, the collector attempts to retrieve the topic. If the topic does not exist yet, the collector receives a `NoSuchTopic` exception and then creates the topic.

```

IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
}
catch (IceStorm.NoSuchTopic ex) {
    topic = topicManager.create("Weather");
}

```

The next step is obtaining a proxy for the publisher object, which the collector narrows to the `Monitor` interface.

```

Ice.ObjectPrx pub = topic.getPublisher();
if (!pub.ice_isDatagram())
    pub = pub.ice_oneway();
MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);

```

Finally, the collector enters its main loop, collecting measurements and publishing them via the IceStorm publisher object.

```

while (true) {
    Measurement m = getMeasurement();
    monitor.report(m);
}

```

26.5.2 Using a Publisher Object

Each topic creates a publisher object for the express purpose of publishing messages. It is a special object in that it implements an Ice interface that allows the object to receive and forward requests (i.e., IceStorm messages) without requiring knowledge of the operation types.

From the publisher's perspective, the publisher object appears to be an application-specific type. In reality, the publisher object can forward requests for any type, and that introduces a degree of risk: a misbehaving publisher can use `uncheckedCast` to narrow the publisher object to any type and invoke any operation; the publisher object unknowingly forwards those requests to the

subscribers. If the publisher sends a request using an incorrect type, the Ice run time in a subscriber typically responds by raising `OperationNotExistedException`. However, since the subscriber receives its messages as oneway invocations, no response can be sent to the publisher object to indicate this failure, and therefore neither the publisher nor the subscriber is aware of the type-mismatch problem. In short, IceStorm places the burden on the developer to ensure that publishers and subscribers are using it correctly.

Although it is not strictly necessary for a publisher to use a oneway proxy for the publisher object, there are two reasons why it is recommended:

- it reiterates the fact that no replies are possible using this proxy;
- it is more efficient because the Ice run time in the collector does not waste time waiting for a reply to each request.

The second reason is especially important, given how IceStorm delivers its messages. As mentioned in Section 26.3.7, the publisher object attempts to deliver a message immediately upon publication of the message, but does not guarantee that the message is delivered to all subscribers before the publication invocation returns. Therefore, a publisher gains nothing by making a twoway invocation on the publisher object, and risks unnecessarily delaying its own processing due to IceStorm implementation details.

Note that using a oneway proxy does not necessarily imply a loss of reliability. If the publisher endpoints use reliable transports such as TCP or SSL, oneway messages are always delivered reliably to the IceStorm server. However, each subscriber can select its own transport for message delivery, therefore the transport used between the publisher and the IceStorm server has no effect on subscribers.¹

26.5.3 Implementing a Subscriber

Our subscriber implementation takes the following steps:

1. Obtain a proxy for the `TopicManager`. This is the primary IceStorm object, used by both publishers and subscribers.
2. Create an object adapter to host our `Monitor` servant.

1. The UDP transport might be used for a subscriber's endpoint if the possibility of lost messages is acceptable. However, UDP is not typically used for a `TopicManager` endpoint because the publisher generally wants to ensure that messages are delivered reliably to IceStorm, even if they may not be delivered reliably to subscribers.

3. Instantiate the Monitor servant and activate it with the object adapter.
4. Subscribe to the Weather topic.
5. Process report messages until shutdown.
6. Unsubscribe from the Weather topic.

In the sections below, we present monitor implementations in C++ and Java.

C++ Example

Our C++ monitor implementation begins by including the necessary header files. The interesting ones are `IceStorm/IceStorm.h`, which is generated from the IceStorm Slice definitions, and `Monitor.h`, containing the generated code for our monitor definitions shown at the beginning of Section 26.2.

```
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <Monitor.h>

using namespace std;

class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurement& m,
                       const Ice::Current&) {
        cout << "Measurement report:" << endl
              << "  Tower: " << m.tower << endl
              << "  W Spd: " << m.windSpeed << endl
              << "  W Dir: " << m.windDirection << endl
              << "  Temp: " << m.temperature << endl
              << endl;
    }
};

int main(int argc, char* argv[])
{
    ...
    Ice::ObjectPrx obj = communicator->stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm::TopicManagerPrx topicManager =
        IceStorm::TopicManagerPrx::checkedCast(obj);

    Ice::ObjectAdapterPtr adapter =
        communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
```

```

Ice::ObjectPrx proxy = adapter->addWithUUID(monitor);

IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribe(qos, proxy);
}
catch (const IceStorm::NoSuchTopic&) {
    // Error! No topic found!
    ...
}

adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
...
}

```

Our implementation of the Monitor servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```

class MonitorI : virtual public Monitor {
public:
    virtual void report(const Measurement& m,
                       const Ice::Current&) {
        cout << "Measurement report:" << endl
              << "  Tower: " << m.tower << endl
              << "  W Spd: " << m.windSpeed << endl
              << "  W Dir: " << m.windDirection << endl
              << "  Temp: " << m.temperature << endl
              << endl;
    }
};

```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the Monitor servant and activates it.

```

Ice::ObjectAdapterPtr adapter =
    communicator->createObjectAdapter("MonitorAdapter");

MonitorPtr monitor = new MonitorI;
Ice::ObjectPrx proxy = adapter->addWithUUID(monitor);

```

Next, the monitor subscribes to the topic.

```

IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
    IceStorm::QoS qos;
    topic->subscribe(qos, proxy);
}
catch (const IceStorm::NoSuchTopic&) {
    // Error! No topic found!
    ...
}

```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```

adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);

```

Java Example

The Java implementation of the monitor is shown below.

```

class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "  Temp: " + m.temperature + "\n");
    }
}

public static void main(String[] args)
{
    ...
    Ice.ObjectPrx obj = communicator.stringToProxy(
        "IceStorm/TopicManager:tcp -p 9999");
    IceStorm.TopicManagerPrx topicManager =
        IceStorm.TopicManagerPrxHelper.checkedCast(obj);

    Ice.ObjectAdapterPtr adapter =
        communicator.createObjectAdapter("MonitorAdapter");

    Monitor monitor = new MonitorI();

```

```

Ice.ObjectPrx proxy = adapter.addWithUUID(monitor);

IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
    java.util.Map qos = null;
    topic.subscribe(qos, proxy);
}
catch (IceStorm.NoSuchTopic ex) {
    // Error! No topic found!
    ...
}

adapter.activate();
communicator.waitForShutdown();

topic.unsubscribe(proxy);
...
}

```

Our implementation of the Monitor servant is currently quite simple. A real implementation might update a graphical display, or incorporate the measurements into an ongoing calculation.

```

class MonitorI extends _MonitorDisp {
    public void report(Measurement m, Ice.Current curr) {
        System.out.println(
            "Measurement report:\n" +
            "  Tower: " + m.tower + "\n" +
            "  W Spd: " + m.windSpeed + "\n" +
            "  W Dir: " + m.windDirection + "\n" +
            "  Temp: " + m.temperature + "\n");
    }
}

```

After obtaining a proxy for the topic manager, the program creates an object adapter, instantiates the Monitor servant and activates it.

```

Monitor monitor = new MonitorI();
Ice.ObjectPrx proxy = adapter.addWithUUID(monitor);

```

Next, the monitor subscribes to the topic.

```

IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
    java.util.Map qos = null;

```



```

        topic.subscribe(qos, proxy);
    }
    catch (IceStorm.NoSuchTopic ex) {
        // Error! No topic found!
        ...
    }

```

Finally, the monitor activates its object adapter and waits to be shutdown. After `waitForShutdown` returns, the monitor cleans up by unsubscribing from the topic.

```

adapter.activate();
communicator.waitForShutdown();

topic.unsubscribe(proxy);

```

26.6 IceStorm Administration

The IceStorm administration tool is a command-line program that provides administrative control of an IceStorm server. The tool requires that the `IceStorm.TopicManager.Proxy` property be specified as described in Section 26.9.2.

The following command-line options are supported:

```

$ icestormadmin -h
Usage: icestormadmin [options] [file...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.
-DNAME               Define NAME as 1.
-DNAME=DEF           Define NAME as DEF.
-UNAME               Remove any definition for NAME.
-IDIR                Put DIR in the include file search path.
-e COMMANDS          Execute COMMANDS.
-d, --debug          Print debug messages.

```

The tool operates in three modes, depending on the command-line arguments:

1. If one or more `-e` options are specified, the tool executes the given commands and exits.
2. If one or more files are specified, the tool preprocesses each file with the C preprocessor, executes the commands in each file, and exits.
3. Otherwise, the tool enters an interactive session.

The **help** command displays the following usage information:

help

Print this message.

exit, quit

Exit this program.

create TOPICS

Add TOPICS.

destroy TOPICS

Remove TOPICS.

link FROM TO COST

Link FROM to TO with the given COST.

unlink FROM TO

Unlink TO from FROM.

graph DATA COST

Construct the link graph as described in DATA with COST.

list [TOPICS]

Display information on TOPICS or all topics.

Many of the commands accept one or more topic names (**TOPICS**) as arguments. Topic names containing white space or matching a command keyword must be enclosed in single or double quotes.

For more information on the **graph** command, see Section 26.7.3.

26.7 Topic Federation

The ability to link topics together into a federation provides IceStorm applications with a lot of flexibility, while the notion of a “cost” associated with links allows applications to restrict the flow of messages in creative ways. IceStorm applications have complete control of topic federation using the Topi cManager interface described in Appendix B, allowing links to be created and removed dynamically as necessary. For many applications, however, the topic graph is static and therefore can be configured using the administrative tool discussed in Section 26.6.

26.7.1 Message Propagation

IceStorm messages are never propagated over more than one link. For example, consider the topic graph shown in Figure 26.4.

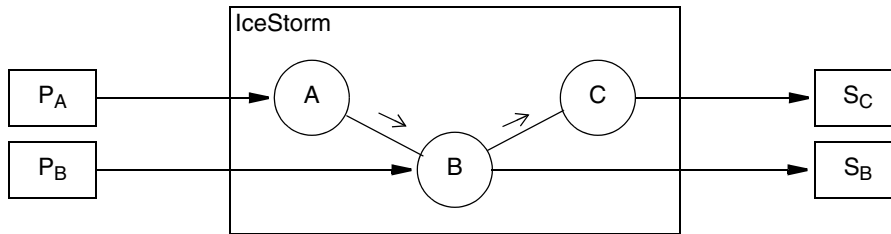


Figure 26.4. Message propagation.

In this case, messages published on A are propagated to B, but B does not propagate A's messages to C. Therefore, subscriber S_B receives messages published on topics A and B, but subscriber S_C only receives messages published on topics B and C. If the application needs messages to propagate from A to C, then a link must be established directly between A and C.

26.7.2 Cost

As described in Section 26.7.1, IceStorm messages are only propagated on the originating topic's immediate links. In addition, applications can use the notion of cost to further restrict message propagation.

A cost is associated with messages and links. When a message is published on a topic, the topic compares the cost associated with each of its links against the message cost, and only propagates the message on those links whose cost equals or exceeds the message cost. A cost value of zero (0) has the following implications:

- messages with a cost value of zero (0) are published on all of the topic's links regardless of the link cost;
- links with a cost value of zero (0) accept all messages regardless of the message cost.

For example, consider the topic graph shown in Figure 26.5.

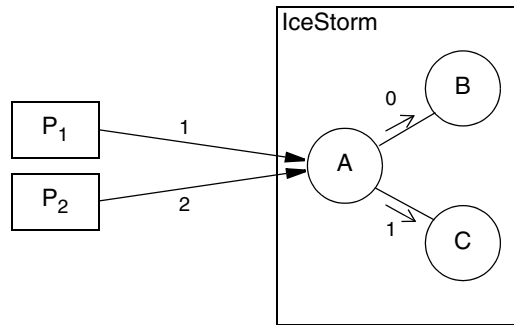


Figure 26.5. Cost semantics.

Publisher P_1 publishes a message on topic A with a cost of 1. This message is propagated on the link to topic B because the link has a cost of 0 and therefore accepts all messages. The message is also propagated on the link to topic C, because the message cost does not exceed the link cost (1). On the other hand, the message published by P_2 with a cost of 2 is only propagated on the link to B.

Request Context

The cost of a message is specified in an Ice request context. Each Ice proxy operation has an implicit argument of type `Ice::Context` representing the request context (see Section 16.8). This argument is rarely used, but it is the ideal location for specifying the cost of an IceStorm message because an application only needs to supply a request context if it actually uses IceStorm's cost feature. If the request context does not contain a cost value, the message is assigned the default cost value of zero (0).

Publishing a Message with a Cost

The code examples below demonstrate how a collector can publish a measurement with a cost value of 5. First, the C++ version:

```
Measurement m = getMeasurement();
Ice::Context ctx;
ctx["cost"] = "5";
monitor->report(m, ctx);
```

And here is the equivalent version in Java:

```
Measurement m = getMeasurement();
java.util.HashMap ctx = new java.util.HashMap();
ctx.put("cost", "5");
monitor.report(m, ctx);
```

Receiving a Message with a Cost

A subscriber can discover the cost of a message by examining the request context supplied in the `Ice::Current` argument. For example, here is a C++ implementation of `Monitor::report` that displays the cost value if it is present:

```
virtual void report(const Measurement& m,
                  const Ice::Current& curr) {
    Ice::Context::const_iterator p = curr.ctx.find("cost");
    cout << "Measurement report:" << endl
         << "  Tower: " << m.tower << endl
         << "  W Spd: " << m.windSpeed << endl
         << "  W Dir: " << m.windDirection << endl
         << "  Temp: " << m.temperature << endl
         << "  Temp: " << m.temperature << endl;
    if (p != curr.ctx.end())
        cout << "    Cost: " << p->second << endl;
    cout << endl;
}
```

And here is the equivalent Java implementation:

```
public void report(Measurement m, Ice.Current curr) {
    String cost = null;
    if (curr.ctx != null)
        cost = curr.ctx.get("cost");
    System.out.println(
        "Measurement report:\n" +
        "  Tower: " + m.tower + "\n" +
        "  W Spd: " + m.windSpeed + "\n" +
        "  W Dir: " + m.windDirection + "\n" +
        "  Temp: " + m.temperature);
    if (cost != null)
        System.out.println("    Cost: " + cost);
    System.out.println();
}
```

For the sake of efficiency, the Ice for Java run time may supply a null value for the request context in `Ice.Current`, therefore an application is required to check for null before using the request context. Furthermore, a non-null request context

is subject to change after the completion of the request, so an application that wishes to retain the request context must make a copy of it (e.g., using `clone`).

26.7.3 Automating Federation

Given the restrictions on message propagation described in the previous sections, creating a complex topic graph can be a tedious endeavor. Of course, creating a topic graph is not typically a common occurrence, since IceStorm keeps a persistent record of the graph. However, there are situations where an automated procedure for creating a topic graph can be valuable, such as during development when the graph might change significantly and often, or when graphs need to be recomputed based on changing costs.

Administration Tool Script

A simple way to automate the creation of a topic graph is to create a text file containing commands to be executed by the IceStorm administration tool. For example, the commands to create the topic graph shown in Figure 26.5 are shown below:

```
create A B C
link A B 0
link A C 1
```

If we store these commands in the file `graph.txt`, we can execute them using the following command:

```
$ icestormadmin --Ice.Config=config graph.txt
```

We assume that the configuration file `config` contains the definition for the property `IceStorm.TopicManager.Proxy`.

XML Graph Descriptor

IceStorm provides an alternative method of configuring topic graphs for applications that use message and link costs. This feature is employed using the administration tool's **graph** command, which accepts an XML graph descriptor and

establishes links between topics based on reachability within a maximum cost. For example, let us start with the graph shown in Figure 26.6.

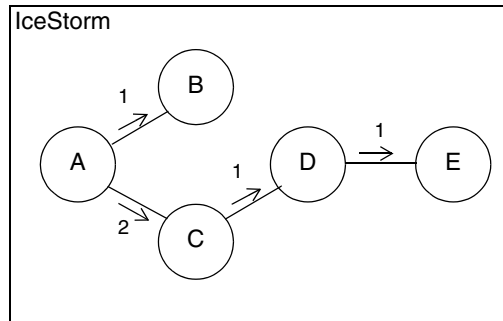


Figure 26.6. Initial graph.

The XML graph descriptor for this topic graph is shown below.

```

<?xml version="1.0" encoding="iso-8859-1"?>
<graph>
  <vertex-set>
    <vertex name="A"/>
    <vertex name="B"/>
    <vertex name="C"/>
    <vertex name="D"/>
    <vertex name="E"/>
  </vertex-set>
  <edge-set>
    <edge source="A" target="B" cost="1"/>
    <edge source="A" target="C" cost="2"/>
    <edge source="C" target="D" cost="1"/>
    <edge source="D" target="E" cost="1"/>
  </edge-set>
</graph>

```

As you can see, the descriptor format is quite straightforward. The outer graph element contains one vertex-set element and one edge-set element. The vertex-set element is comprised of two or more vertex elements having a single attribute, name, providing the name of a topic. The edge-set element contains one or more edge elements that describe links between topics. The source and target attributes supply the topic names, and the cost attribute defines the link cost.

Although this descriptor accurately describes the structure of the graph in Figure 26.6, the graph that is ultimately created by the administration tool's **graph** command is dependent on the specified maximum cost. Suppose we save our graph descriptor as `graph.xml` and execute the following commands:

```
$ icestormadmin --Ice.Config=config  
>>> create A B C  
>>> graph "graph.xml" 1
```

We must first create the topics that are used as vertices in our XML graph descriptor. Next, we instruct the administration tool to create a graph using a maximum cost of 1, which results in the graph shown in Figure 26.7.

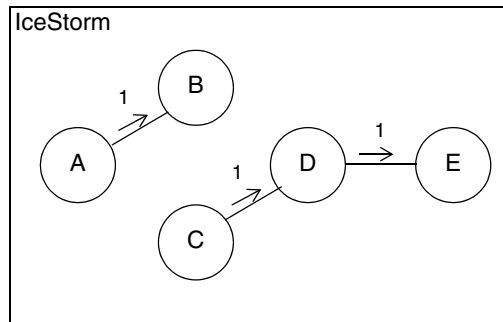


Figure 26.7. Graph with maximum cost of 1.

Notice that the link from A to C was not created, because the cost of that link (2) exceeded the maximum cost. Now consider the outcome of using a maximum cost of 2:

```
>>> graph "graph.xml" 2
```


As shown in Figure 26.8, the link from A to C is now present. In addition, a link from C to E has been established with a cost of 2, because E is reachable from C within the maximum cost.

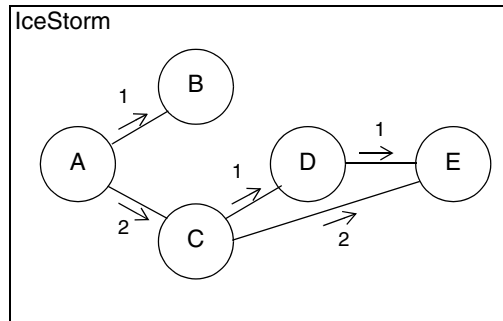


Figure 26.8. Graph with maximum cost of 2.

We can verify that the links are created properly using the **list** command:

```

>>> list A B C D E
A
    B with cost 0
    C with cost 2
B
C
    D with cost 1
    E with cost 2
D
    E with cost 1
E
  
```

The **graph** command can be used repeatedly to experiment with different cost configurations. For each invocation of the **graph** command, the administration tool recomputes the graph based on the specified maximum cost, establishes the necessary links, and removes any links that are no longer valid given the new cost constraints.

26.8 Quality of Service

An IceStorm subscriber specifies QoS parameters at the time of subscription. IceStorm currently supports only one QoS parameter, `reliability`, which is described in the section below.

26.8.1 Reliability

The QoS parameter `reliability` affects message delivery. The default value is `oneway`, in which each published message results in a separate oneway invocation to the subscriber. Alternatively, the value `batch` causes IceStorm to temporarily queue messages for the subscriber and flush them at regular intervals by delivering them as batch oneway requests (see Chapter 18). There is no difference between the two reliability modes as far as the subscriber's servant implementation is concerned; the messages are delivered individually as servant up-calls in the normal fashion. However, there is a difference in terms of safety and performance.

The oneway mode emphasizes safety by delivering its messages as soon as they are received in order to minimize the possibility of lost messages should an IceStorm failure occur.

By contrast, the `batch` mode queues messages for a configurable amount of time, increasing the possibility of lost messages in the event of failure, but providing better performance by minimizing network overhead. The frequency with which the `batch` mode flushes its queues is determined by the configuration property `IceStorm.Flush.Timeout`, as described in Appendix C.

The performance improvement offered by the `batch` mode is especially apparent for topics in which small messages are sent frequently. Rather than making many small requests, as would be done in `oneway` mode, the messages are accumulated and delivered in groups, amortizing the network overhead across several messages. However, the subscriber should only use the `batch` mode if it is willing to wait up to the flush timeout for new messages.

26.8.2 Example

The Slice type `IceStorm: : QoS` is defined as a `dictionary` whose key and value types are both `string`, therefore the QoS parameter name and value are both represented as strings. The example code presented in Section 26.5.3 used an empty dictionary for the QoS argument, meaning default values are used. The

C++ and Java examples shown below illustrate how to set the `reliability` parameter to `batch`.

C++ Example

```
IceStorm::QoS qos;  
qos["reliability"] = "batch";  
topic->subscribe(qos, proxy);
```

Java Example

```
java.util.Map qos = new java.util.HashMap();  
qos.put("reliability", "batch");  
topic.subscribe(qos, proxy);
```

26.9 Configuring IceStorm

IceStorm is a relatively lightweight service in that it requires very little configuration and is implemented as an `IceBox` service (see Chapter 25). The configuration properties supported by IceStorm are described in Appendix C; however, most of them control diagnostic output and are not discussed in this chapter.

26.9.1 Server Configuration

The sample server configuration file shown below presents the properties of primary interest:

```
IceBox.Service.IceStorm=IceStormService,1.0.0:create  
IceBox.DBEnvName.IceStorm=db  
IceStorm.TopicManager.Endpoints=tcp -p 9999  
IceStorm.Publish.Endpoints=tcp
```

The first property defines the entry point for the IceStorm service. The service name (the last component of the `IceBox.Service` property name, `IceStorm` in this example) determines the prefix for the IceStorm configuration properties. It is not mandatory to use `IceStorm` as the service name, but it is recommended if there is no preferred name.

IceStorm uses Freeze to manage the service's persistent state, therefore the second property specifies the pathname of the Freeze database environment directory (see Chapter 21) for the service. In this example, the directory `db` is used, which must already exist in the current working directory.

The final two properties specify the endpoints used by the IceStorm object adapters; notice that their property names begin with `IceStorm`, matching the service name. The `TopicManager` property specifies the endpoints on which the `TopicManager` and `Topic` objects reside; these endpoints typically use a connection-oriented protocol such as TCP or SSL. The `Publish` property specifies the endpoints used by topic publisher objects.

26.9.2 Client Configuration

Clients of the service can define a proxy for the `TopicManager` object as follows:

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 9999
```

The name of the property is not relevant, but the endpoint must match that of the `IceStorm.TopicManager.Endpoints` property, and the object identity must use the IceStorm service name as the identity category and `TopicManager` as the identity name.

26.10 Summary

IceStorm is a publish/subscribe service that offers Ice applications a flexible and efficient means of publishing oneway requests to a group of subscribers. IceStorm simplifies development by relieving the application from the burden of managing subscribers and handling delivery errors, allowing the application to focus on publishing its data and not on the minutiae of distribution. Finally, IceStorm leverages Ice request-forwarding facilities in order to provide a typed interface to publishers and subscribers, minimizing the impact on applications.

Appendices

Appendix A

Slice Keywords

The following identifiers are Slice keywords:

bool	enum	implements	module	string
byte	exception	int	nonmutating	struct
class	extends	interface	Object	throws
const	false	local	out	true
dictionary	float	Local Object	sequence	void
double	idempotent	long	short	

Keywords must be capitalized as shown.

Appendix B

Slice Documentation

B.1 Ice

Overview

`module Ice`

The Ice core library. Among many other features, the Ice core library manages all the communication tasks using an efficient protocol (including protocol compression and support for both TCP and UDP), provides a thread pool for multi-threaded servers, and additional functionality that supports high scalability.

Bool Seq

`sequence<bool> Bool Seq;`

A sequence of bools.

ByteSeq

`sequence<byte> ByteSeq;`

A sequence of bytes.

Used By

```
:: Glacier::Starter::startRouter, :: Glacier::Starter::startRouter,
:: Glacier::Starter::startRouter, BadMagicException::badMagic,
:: IcePatch::FileDesc::md5, :: IcePatch::Regular::getBZ2,
:: IcePatch::Regular::getBZ2MD5, :: IceSSL::Plugin::addTrustedCertificate,
:: IceSSL::Plugin::getSingularCertificate,
:: IceSSL::Plugin::setRSAKeys, :: IceSSL::Plugin::setRSAKeys.
```

DoubleSeq

```
sequence<double> DoubleSeq;
```

A sequence of doubles.

FacetPath

```
sequence<string> FacetPath;
```

The facet path.

Used By

```
:: Freeze::Evictor::addFacet, :: Freeze::Evictor::removeFacet,
:: Freeze::EvictorStorageKey::facet, Current::facet, RequestFailedException::facet.
```

FloatSeq

```
sequence<float> FloatSeq;
```

A sequence of floats.

IdentitySeq

```
local sequence<Identity> IdentitySeq;
```

A sequence of identities.

IntSeq

```
sequence<int> IntSeq;
```

A sequence of ints.

LongSeq

sequence<long> LongSeq;

A sequence of longs.

ObjectProxySeq

sequence<Object*> ObjectProxySeq;

A sequence of object proxies.

Used By

:: IcePack: : Query: : findAllObjectsWithType.

ObjectSeq

sequence<Object> ObjectSeq;

A sequence of objects.

ShortSeq

sequence<short> ShortSeq;

A sequence of shorts.

StringSeq

sequence<string> StringSeq;

A sequence of strings.

Used By

Properties: : getCommandLineOptions, Properties: : parseCommandLineOptions, Properties: : parseCommandLineOptions, Properties: : parseCommandLineOptions, Properties: : parseCommandLineOptions, Properties: : parseCommandLineOptions, Properties: : start, :: IceBox: : FreezeService: : start, :: IceBox: : Service: : start,

```

::IcePack::Admin::addApplication, ::IcePack::Admin::addServer,
::IcePack::Admin::getAdapterIds, ::IcePack::Admin::getAdapterNames,
::IcePack::Admin::getAdapterServerNames, ::IcePack::ServerDescription::args,
::IcePack::ServerDescription::envs,
::IcePack::ServerDescription::targets.

```

Context

```
dictionary<string, string> Context;
```

A request context. Context is used to transmit metadata about a request from the server to the client, such as Quality-of-Service (QoS) parameters. Each operation on the client has a Context as its implicit final parameter.

Used By

Current::ctx.

ObjectDict

```
local dictionary<Identity, Object> ObjectDict;
```

A mapping between identities and Ice objects.

PropertyDict

```
local dictionary<string, string> PropertyDict;
```

A simple collection of properties, represented as a dictionary of key/value pairs. Both key and value are strings.

Used By

Properties::getPropertiesForPrefix.

See Also

Properties::getPropertiesForPrefix.

B.2 Ice: : AbortBatchRequestException

Overview

Local exception AbortBatchRequestException extends ProtocolException

This exception is a specialization of ProtocolException, indicating that a batch request has been aborted.

B.3 Ice: : AdapterAlreadyActiveException

Overview

exception AdapterAlreadyActiveException

This exception is raised if a server tries to set endpoints for an adapter that is already active.

B.4 Ice: : AdapterNotFoundException

Overview

exception AdapterNotFoundException

This exception is raised if an adapter cannot be found.

B.5 Ice: : AlreadyRegisteredException

Overview

Local exception AlreadyRegisteredException

This exception is raised if an attempt is made to register a servant, servant locator, facet, object factory, plug-in, object adapter, object, or user exception factory more than once for the same ID.

`id`

`string id;`

The id (or name) of the object that is registered already.

`kindOfObject`

`string kindOfObject;`

The kind of object that is registered already: "servant", "servant locator", "facet", "object factory", "plug-in", "object adapter", "object", or "user exception factory".

B.6 Ice: : BadMagicException

Overview

Local exception `BadMagicException` extends `ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that a message did not start with the expected magic number ('I', 'c', 'e', 'P').

`badMagic`

`ByteSeq badMagic;`

A sequence containing the first four bytes of the incorrect message.

B.7 Ice: : CloseConnectionException

Overview

Local exception `CloseConnectionException` extends `ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that the connection has been gracefully shut down by the server. The operation call that caused this exception has not been executed by the server. In most cases you will not get this exception, because the client will automatically retry the operation call in case the server shut down the connection. However, if upon retry the server shuts down the connection again, and the retry limit has been reached, then this exception is propagated to the application code.

B.8 `Ice::CloseTimeoutException`

Overview

Local exception `CloseTimeoutException` extends `TimeoutException`

This exception is a specialization of `TimeoutException` for connection closure timeout conditions.

B.9 `Ice::CollocationOptimizationException`

Overview

Local exception `CollocationOptimizationException`

This exception is raised if a feature is requested that is not supported with collocation optimization.

B.10 `Ice::Communicator`

Overview

Local interface `Communicator`

The central object in Ice. One or more communicators can be instantiated for an Ice application. Communicator instantiation is language specific, and not specified in Slice code.

Used By

`:: Freeze:: Connecti on:: getCommuni cator, Obj ectAdapter:: getCommuni -
cator, :: IceBox:: FreezeServi ce:: start, :: IceBox:: Servi ce:: start.`

See Also

Logger, Stats, Obj ectAdapter, Properti es, Obj ectFactory.

addObj ectFactory

```
void addObj ectFactory(Obj ectFactory factory, string i d);
```

Add a servant factory to this communicator. Installing a factory with an id for which a factory is already registered throws `AlreadyRegi steredExcepti on`.

When unmarshaling an Ice object, the Ice run-time reads the most-derived type id off the wire and attempts to create an instance of the type using a factory. If no instance is created, either because no factory was found, or because all factories returned nil, the object is sliced to the next most-derived type and the process repeats. If no factory is found that can create an instance, the Ice run-time will slice the object to the type `Ice::Object`.

The following order is used to locate a factory for a type:

1. The Ice run-time looks for a factory registered specifically for the type.
2. If no instance has been created, the Ice run-time looks for the default factory, which is registered with an empty type id.
3. If no instance has been created by any of the preceding steps, the Ice run-time looks for a factory that may have been statically generated by the language mapping for non-abstract classes.

Parameters

factory

The factory to add.

i d

The type id for which the factory can create instances, or an empty string for the default factory.

See Also

`removeObj ectFactory, fi ndObj ectFactory, Obj ectFactory.`

createObjectAdapter

```
ObjectAdapter createObjectAdapter(string name);
```

Create a new object adapter. The endpoints for the object adapter are taken from the property *name*. Endpoints.

Parameters

name

The object adapter name.

Return Value

The new object adapter.

See Also

createObjectAdapterWithEndpoints, ObjectAdapter, Properties.

createObjectAdapterWithEndpoints

```
ObjectAdapter createObjectAdapterWithEndpoints(string name, string endpoints);
```

Create a new object adapter with endpoints. This method sets the property *name*. Endpoints, and then calls createObjectAdapter. It is provided as a convenience function.

Parameters

name

The object adapter name.

endpoints

The endpoints for the object adapter.

Return Value

The new object adapter.

See Also

createObjectAdapter, ObjectAdapter, Properties.

destroy

```
void destroy();
```

Destroy the communicator. This operation calls shutdown implicitly. Calling destroy cleans up memory, and shuts down this communicator's client functionality. Subsequent calls to destroy are ignored.

See Also

shutdown.

findObjectFactory

```
ObjectFactory findObjectFactory(string id);
```

Find a servant factory registered with this communicator.

Parameters

id

The type id for which the factory can create instances, or an empty string for the default factory.

Return Value

The servant factory, or null if no servant factory was found for the given id.

See Also

addObjectFactory, removeObjectFactory, ObjectFactory.

flushBatchRequests

```
void flushBatchRequests();
```

Flush any pending batch requests for this communicator. This causes all batch requests that were sent via proxies obtained via this communicator to be sent to the server.

getLogger

```
Logger getLogger();
```

Get the logger for this communicator.

Return Value

This communicator's logger.

See Also

Logger.

getPluginManager

PluginManager getPluginManager();

Get the plug-in manager for this communicator.

Return Value

This communicator's plug-in manager.

See Also

PluginManager.

getProperties

Properties getProperties();

Get the properties for this communicator.

Return Value

This communicator's properties.

See Also

Properties.

getStats

Stats getStats();

Get the statistics callback object for this communicator.

Return Value

This communicator's statistics callback object.

See Also

Stats.

proxyToString

```
string proxyToString(Object* obj);
```

Convert a proxy into a string.

Parameters

obj

The proxy to convert into a string.

Return Value

The "stringified" proxy.

See Also

stringToProxy.

removeObjectFactory

```
void removeObjectFactory(string id);
```

Remove a servant factory from this communicator. Removing an id for which no factory is registered throws `NotRegisteredException`.

Parameters

id

The type id for which the factory can create instances, or an empty string for the default factory.

See Also

addObjectFactory, findObjectFactory, ObjectFactory.

setDefaultLocator

```
void setDefaultLocator(Locator* loc);
```

Set a default Ice locator for this communicator. All newly created proxy and object adapters will use this default locator. To disable the default locator, null can be used. Note that this operation has no effect on existing proxies or object adapters.

You can also set a locator for an individual proxy by calling the operation `ice_locator` on the proxy, or for an object adapter by calling the operation `setLocator` on the object adapter.

Parameters

`loc`

The default locator to use for this communicator.

See Also

Locator, ObjectAdapter: : setLocator.

setDefaultRouter

```
void setDefaultRouter(Router* rtr);
```

Set a default router for this communicator. All newly created proxies will use this default router. To disable the default router, null can be used. Note that this operation has no effect on existing proxies.

You can also set a router for an individual proxy by calling the operation `ice_router` on the proxy.

Parameters

`rtr`

The default router to use for this communicator.

See Also

Router, ObjectAdapter: : addRouter.

setLogger

```
void setLogger(Logger log);
```

Set the logger for this communicator.

Parameters

log

The logger to use for this communicator.

See Also

Logger.

setStats

```
void setStats(Stats st);
```

Set the statistics callback object for this communicator.

Parameters

st

The statistics callback object to use for this communicator.

See Also

Stats.

shutdown

```
void shutdown();
```

Shuts down this communicator's server functionality, including the deactivation of all object adapters. Subsequent calls to shutdown are ignored.

After shutdown returns, no new requests are processed. However, requests that have been started before shutdown was called might still be active. You can use `waitForShutdown` to wait for the completion of all requests.

See Also

`destroy`, `waitForShutdown`, `ObjectAdapter::deactivate`.

stringToProxy

```
Object* stringToProxy(string str);
```

Convert a string into a proxy. For example, `MyCategory/MyObject: tcp -h some_host -p 10000` creates a proxy that refers to the Ice object having an identity with a name "MyObject" and a category "MyCategory", with the server running on host "some_host", port 10000.

Parameters

`str`

The string to convert into a proxy.

Return Value

The proxy.

See Also

`proxyToString`.

`waitForShutdown`

`void waitForShutdown();`

Wait until this communicator's server functionality has shut down completely. Calling `shutdown` initiates shutdown, and `waitForShutdown` only returns when all outstanding requests have completed. A typical use of this operation is to call it from the main thread, which then waits until some other thread calls `shutdown`. After shutdown is complete, the main thread returns and can do some cleanup work before it finally calls `destroy` to also shut down the client functionality, and then exits the application.

See Also

`shutdown`, `destroy`, `ObjectAdapter::waitForDeactivate`.

B.11 `Ice::CommunicatorDestroyedException`

Overview

`local_exception CommunicatorDestroyedException`

This exception is raised if the Communicator has been destroyed.

See Also

Communicator::destroy.

B.12 Ice::CompressionException

Overview

Local exception CompressionException extends ProtocolException

This exception is a specialization of ProtocolException that is raised if there is a problem with compressing or uncompressing data.

reason

string reason;

A description of the problem with compress or uncompress.

B.13 Ice::CompressionNotSupportedException

Overview

Local exception CompressionNotSupportedException extends ProtocolException

This exception is a specialization of ProtocolException that is raised if a compressed protocol message has been received by an Ice version that does not support compression.

B.14 Ice::ConnectFailedException

Overview

Local exception ConnectFailedException extends SocketException

This exception is a specialization of `SocketException` for connection failures.

B.15 `Ice::ConnectTimeoutException`

Overview

Local exception `ConnectTimeoutException` extends `TimeoutException`

This exception is a specialization of `TimeoutException` for connection establishment timeout conditions.

B.16 `Ice::ConnectionLostException`

Overview

Local exception `ConnectionLostException` extends `SocketException`

This exception is a specialization of `SocketException`, indicating a lost connection.

B.17 `Ice::ConnectionNotValidatedException`

Overview

Local exception `ConnectionNotValidatedException` extends `ProtocolException`

This exception is a specialization of `ProtocolException`, that is raised if a message is received over a connection that is not yet validated.

B.18 Ice: : Connecti onTi meoutExcepti on

Overview

Local exception `Connecti onTi meoutExcepti on` extends `Ti meoutExcepti on`

This exception is a specialization of `Ti meoutExcepti on`, and indicates that a connection has been shut down because it has been idle for some time.

B.19 Ice: : Current

Overview

Local struct `Current`

Information about the current method invocation for servers. Each operation on the server has a `Current` as its implicit final parameter. `Current` is mostly used for Ice services, such as `IceStorm`. Most applications ignore this parameter.

Used By

`ServantLocator: : fi ni shed`, `ServantLocator: : l o c a t e`.

See Also

`: : I c e S t o r m`.

adapter

`ObjectAdapter adapter;`

The object adapter.

ctx

`Context ctx;`

The request context, as received from the client.

`facet`

`FacetPath facet;`

The facet.

`id`

`Identity id;`

The Ice object identity.

`mode`

`OperationMode mode;`

The mode of the operation.

`operation`

`string operation;`

The operation name.

B.20 `Ice::DNSException`

Overview

`local_exception DNSException`

This exception indicates a DNS problem. For details on the cause, error should be inspected.

`error`

`int error;`

The error number describing the DNS problem. For C++ and Unix, this is equivalent to `h_errno`. For C++ and Windows, this is the value returned by `WSAGetLastError()`.

host

string host;

The host name that could not be resolved.

B.21 Ice::DatagramLimitException

Overview

Local exception `DatagramLimitException` extends `ProtocolException`

This exception is a specialization of `ProtocolException` that is raised if a datagram exceeds the configured send or receive buffer size, or exceeds the maximum payload size of a UDP packet (65507 bytes).

B.22 Ice::EncapsulationException

Overview

Local exception `EncapsulationException` extends `MarshalException`

This exception is a specialization of `MarshalException`, indicating a malformed data encapsulation.

B.23 Ice::EndpointParseException

Overview

Local exception `EndpointParseException`

This exception is raised if there was an error while parsing an endpoint.

str

string str;

The string that could not be parsed.

B.24 Ice: : FacetNotExistException

Overview

Local exception FacetNotExistException extends RequestFailedException

This exception is raised if an object does not implement a given facet path.

B.25 Ice: : Identity

Overview

struct Identity

The identity of an Ice object. An empty name denotes a null object.

Used By

```
:: Freeze: : Evi ctor: : addFacet, :: Freeze: : Evi ctor: : createObject,
:: Freeze: : Evi ctor: : destroyObject, :: Freeze: : Evi ctor: : hasObject,
:: Freeze: : Evi ctor: : removeAllFacets, :: Freeze: : Evi ctor: : removeFacet,
:: Freeze: : Evi ctorIterator: : next, :: Freeze: : Evi ctorStorageKey: : identitySeq, :: Freeze: : ServantInitializer: : initialize, Current: : id, IdentitySeq, IllegalIdentityException: : id, Locator: : findObjectById, ObjectAdapter: : add, ObjectAdapter: : createDirectProxy, ObjectAdapter: : createProxy, ObjectAdapter: : createReverseProxy, ObjectAdapter: : identityToServant, ObjectAdapter: : remove, ObjectDict, RequestFailedException: : id, :: IcePack: : Query: : findObjectById.
```

category

string category;

The Ice object category.

See Also

ServantLocator, ObjectAdapter::addServantLocator.

name

string name;

The name of the Ice object.

B.26 Ice::IdentityParseException

Overview

local exception IdentityParseException

This exception is raised if there was an error while parsing a stringified identity.

str

string str;

The string that could not be parsed.

B.27 Ice::IllegalIdentityException

Overview

local exception IllegalIdentityException

This exception is raised if an illegal identity is encountered.

id

Identity id;

The illegal identity.

B.28 Ice: : IllegalIndirectionException

Overview

LocalException IllegalIndirectionException extends MarshalException

This exception is a specialization of MarshalException, indicating an illegal indirection during unmarshaling.

B.29 Ice: : IllegalMessageSizeException

Overview

LocalException IllegalMessageSizeException extends ProtocolException

This exception is a specialization of ProtocolException, indicating that the message size is illegal, i.e., it is less than the minimum required size.

B.30 Ice: : Locator

Overview

interface Locator

The Ice locator interface. This interface is used by clients to lookup adapters and objects. It is also used by servers to get the locator registry proxy.

The Locator interface is intended to be used by Ice internals and by locator implementations. Regular user code should not attempt to use any functionality of this interface directly.

findAdapterById

```
[ "amd" ] Object* findAdapterById(string id) throws
AdapterNotFoundException;
```

Find an adapter by id and return its proxy (a dummy direct proxy created by the adapter).

Parameters

id

The adapter id.

Return Value

The adapter proxy, or null if the adapter is not active.

Exceptions

AdapterNotFoundException

Raised if the adapter cannot be found.

findObjectById

```
[ "amd" ] Object* findObjectById(Identity id) throws  
ObjectNotFoundException;
```

Find an object by identity and return its proxy.

Parameters

id

The identity.

Return Value

The proxy, or null if the object is not active.

Exceptions

ObjectNotFoundException

Raised if the object cannot be found.

getRegistry

```
LocatorRegistry* getRegistry();
```

Get the locator registry.

Return Value

The locator registry.

B.31 Ice: : LocatorRegistry

Overview

interface LocatorRegistry

The Ice locator registry interface. This interface is used by servers to register adapter endpoints with the locator.

The LocatorRegistry interface is intended to be used by Ice internals and by locator implementations. Regular user code should not attempt to use any functionality of this interface directly.

setAdapterDirectProxy

void setAdapterDirectProxy(string id, Object* proxy) throws
AdapterNotFoundException, AdapterAlreadyActiveException;

Set the adapter endpoints with the locator registry.

Parameters

id

The adapter id.

proxy

The adapter proxy (a dummy direct proxy created by the adapter). The direct proxy contains the adapter endpoints.

Exceptions

AdapterNotFoundException

Raised if the adapter cannot be found, or if the locator only allows registered adapters to set their active proxy and the adapter is not registered with the locator.

AdapterAlreadyActive

Raised if an adapter with the same id is already active.

setServerProcessProxy

`void setServerProcessProxy(string id, Process* proxy) throws
ServerNotFoundException;`

Set the process proxy for a server.

Parameters

`id`

The server id.

`proxy`

The process proxy.

Exceptions

`ServerNotFoundException`

Raised if the server cannot be found.

B.32 Ice::Logger

Overview

Local interface `Logger`

The Ice message logger. Applications can provide their own logger by implementing this interface and installing it in a communicator.

Used By

`Communicator::getLogger`, `Communicator::setLogger`.

`error`

`void error(string message);`

Log error messages.

Parameters

message

The error message to log.

See Also

warni ng.

trace

```
void trace(string category, string message);
```

Log trace messages.

Parameters

category

The trace category.

message

The trace message to log.

warni ng

```
void warni ng(string message);
```

Log warning messages.

Parameters

message

The warning message to log.

See Also

error.

B.33 Ice: : Marshal Exception

Overview

Local exception `MarshalException` extends `ProtocolException`

This exception is a specialization of `ProtocolException` that is raised upon an error during marshaling or unmarshaling data.

Derived Exceptions

`EncapsulationException`, `IllegalIndirectionException`, `MemoryLimitException`, `NegativeSizeException`, `NoObjectFactoryException`, `ProxyUnmarshalException`, `UnmarshalOutOfBoundsException`.

B.34 Ice: : MemoryLimitException

Overview

Local exception `MemoryLimitException` extends `MarshalException`

This exception is a specialization of `MarshalException` that is raised if the system-specific memory limit is exceeded during marshaling or unmarshaling.

B.35 Ice: : NegativeSizeException

Overview

Local exception `NegativeSizeException` extends `MarshalException`

This exception is a specialization of `MarshalException` that is raised if a negative size (e.g., a negative sequence size) is received.

B.36 Ice: : NoEndpointException

Overview

Local exception NoEndpointException

This exception is raised if no suitable endpoint is available.

proxy

string proxy;

The stringified proxy for which no suitable endpoint is available.

B.37 Ice: : NoObjectFactoryException

Overview

Local exception NoObjectFactoryException extends MarshalException

This exception is a specialization of MarshalException that is raised if no suitable object factory was found during object unmarshaling.

See Also

ObjectFactory, Communicator: : addObjectFactory, Communicator: : removeObjectFactory, Communicator: : findObjectFactory.

type

string type;

The absolute Slice type name of the object for which we could not find a factory.

B.38 Ice::NotRegisteredException

Overview

Local exception NotRegisteredException

This exception is raised if an attempt is made to remove a servant, facet, object factory, plug-in, object adapter, object, or user exception factory that is not currently registered.

id

string id;

The id (or name) of the object that could not be removed.

kindOfObject

string kindOfObject;

The kind of object that could not be removed: "servant", "facet", "object factory", "plug-in", "object adapter", "object", or "user exception factory".

B.39 Ice::ObjectAdapter

Overview

Local interface ObjectAdapter

The object adapter, which is responsible for receiving requests from endpoints, and for mapping between servants, identities, and proxies.

Used By

:: Freeze:: ServantInitializer:: initialize, Communicator:: createObjectAdapter, Communicator:: createObjectAdapterWithEndpoints, Current:: adapter.

See Also

Communicator, ServantLocator.

activate

```
void activate();
```

Activate all endpoints that belong to this object adapter. After activation, the object adapter can dispatch requests received through its endpoints.

See Also

hold, deactivate.

add

```
Object* add(Object servant, Identity id);
```

Add a servant to this object adapter's Active Servant Map. Note that one servant can implement several Ice objects by registering the servant with multiple identities. Adding a servant with an identity that is in the map already throws `AlreadyRegisteredException`.

Parameters

servant

The servant to add.

id

The identity of the Ice object that is implemented by the servant.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

Identity, addWithUUID, remove.

addRouter

```
void addRouter(Router* rtr);
```

Add a router to this object adapter. By doing so, this object adapter can receive callbacks from this router over connections that are established from this process to the router. This avoids the need for the router to establish a separate connection back to this object adapter.

You can add a particular router to only a single object adapter. Adding the same router to more than one object adapter results in undefined behavior. However, it is possible to add different routers to different object adapters.

Parameters

`rtr`

The router to add to this object adapter.

See Also

Router, Communicator: : setDefaultRouter.

addServantLocator

```
void addServantLocator(ServantLocator locator, string category);
```

Add a Servant Locator to this object adapter. Adding a servant locator for a category for which a servant locator is already registered throws `AlreadyRegisteredException`. To dispatch operation calls on servants, the object adapter tries to find a servant for a given Ice object identity in the following order:

1. The object adapter tries to find a servant for the identity in the Active Servant Map.
2. If no servant has been found in the Active Servant Map, the object adapter tries to find a locator for the category component of the identity. If a locator is found, the object adapter tries to find a servant using this locator.
3. If no servant has been found by any of the preceding steps, the object adapter tries to find a locator for an empty category, regardless of the category contained in the identity. If a locator is found, the object adapter tries to find a servant using this locator.
4. If no servant has been found with any of the preceding steps, the object adapter gives up and the caller receives `ObjectNotExistException`.

Only one locator for the empty category can be installed.

Parameters

locator

The locator to add.

category

The category for which the Servant Locator can locate servants, or an empty string if the Servant Locator does not belong to any specific category.

See Also

Identity, findServantLocator, ServantLocator.

addWithUUID

```
Object* addWithUUID(Object servant);
```

Add a servant to this object adapter's Active Servant Map, using an automatically generated UUID as its identity. Note that the generated UUID identity can be accessed using the proxy's `ice_getIdentity` operation.

Parameters

servant

The servant to add.

Return Value

A proxy that matches the generated UUID identity and this object adapter.

See Also

Identity, add, remove.

createDirectProxy

```
Object* createDirectProxy(Identity id);
```

Create a "direct proxy" that matches this object adapter and the given identity. A direct proxy always contains the current adapter endpoints.

This operation is intended to be used by locator implementations. Regular user code should not attempt to use this operation.

Parameters

`i d`

The identity for which a proxy is to be created.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

`I d e n t i t y`.

createProxy

```
Object* createProxy(I d e n t i t y i d);
```

Create a proxy that matches this object adapter and the given identity.

Parameters

`i d`

The identity for which a proxy is to be created.

Return Value

A proxy that matches the given identity and this object adapter.

See Also

`I d e n t i t y`.

createReverseProxy

```
Object* createReverseProxy(I d e n t i t y i d);
```

Create a "reverse proxy" that matches this object adapter and the given identity. A reverse proxy uses connections that have been established from a client to this object adapter.

Like the Router interface, this operation is intended to be used by router implementations. Regular user code should not attempt to use this operation.

Parameters

`i d`

The identity for which a proxy is to be created.

Return Value

A "reverse proxy" that matches the given identity and this object adapter.

See Also

`I d e n t i t y`.

deactivate

`void deactivate();`

Deactivate all endpoints that belong to this object adapter. After deactivation, the object adapter stops receiving requests through its endpoints. Object adapters that have been deactivated must not be reactivated again, i.e., the deactivation is permanent and `activate` or `hold` must not be called after calling `deactivate`; attempting to do so results in an `ObjectAdapterDeactivatedException` being thrown. Calls to `deactivate` on an already deactivated object adapter are ignored. After `deactivate` returns, no new requests are processed by the object adapter. However, requests that have been started before `deactivate` was called might still be active. You can use `waitForDeactivate` to wait for the completion of all requests for this object adapter.

See Also

`activate`, `hold`, `waitForDeactivate`, `Communicator::shutdown`.

findServantLocator

`ServantLocator findServantLocator(string category);`

Find a Servant Locator installed with this object adapter.

Parameters

`category`

The category for which the Servant Locator can locate servants, or an empty string if the Servant Locator does not belong to any specific category.

Return Value

The Servant Locator, or null if no Servant Locator was found for the given category.

See Also

Identity, addServantLocator, ServantLocator.

getCommunicator

Communicator getCommunicator();

Get the communicator this object adapter belongs to.

Return Value

This object adapter's communicator.

See Also

Communicator.

getLocator

Locator* getLocator();

Get the locator configured for this object adapter.

Return Value

The locator proxy.

See Also

Locator.

getName

string getName();

Get the name of this object adapter.

Return Value

This object adapter's name.

hold

```
void hold();
```

Temporarily hold receiving and dispatching requests. The object adapter can be reactivated with the `activate` operation.

Holding is not immediate, i.e., after `hold` returns, the object adapter might still be active for some time. You can use `waitForHold` to wait until holding is complete.

See Also

`activate`, `deactivate`, `waitForHold`.

identityToServant

```
Object identityToServant(Identity id);
```

Look up a servant in this object adapter's Active Servant Map by the identity of the Ice object it implements.

This operation only tries to lookup a servant in the Active Servant Map. It does not attempt to find a servant by using any installed `ServantLocator`.

Parameters

`id`

The identity of the Ice object for which the servant should be returned.

Return Value

The servant that implements the Ice object with the given identity, or null if no such servant has been found.

See Also

`Identity`, `proxyToServant`.

proxyToServant

```
Object proxyToServant(Object* proxy);
```

Look up a servant in this object adapter's Active Servant Map, given a proxy. This operation only tries to lookup a servant in the Active Servant Map. It does not attempt to find a servant via any installed `ServantLocators`.

Parameters

proxy

The proxy for which the servant should be returned.

Return Value

The servant that matches the proxy, or null if no such servant has been found. see `identityToServant`

remove

```
void remove(Identity id);
```

Remove a servant from the object adapter's Active Servant Map.

Parameters

id

The identity of the Ice object that is implemented by the servant. If the servant implements multiple Ice objects, `remove` has to be called for all those Ice objects. Removing an identity that is not in the map throws `NotRegisteredException`.

See Also

`Identity`, `add`, `addWithUUID`.

setLocator

```
void setLocator(Locator* loc);
```

Set an Ice locator for this object adapter. By doing so, the object adapter will register itself with the locator registry when it is activated for the first time. Furthermore, the proxies created by this object adapter will contain the adapter name instead of its endpoints.

Parameters

`l oc`

The locator used by this object adapter.

See Also

`createDi rectProxy`, `Locator`, `LocatorRegi stry`.

wait tForDeacti vate

`void wait tForDeacti vate();`

Wait until the object adapter has deactivated. Calling `deacti vate` initiates object adapter deactivation, and `wait tForDeacti vate` only returns when deactivation has been completed.

See Also

`deacti vate`, `wait tForHol d`, `Communi cator: : wait tForShutdown`.

wait tForHol d

`void wait tForHol d();`

Wait until the object adapter holds requests. Calling `hol d` initiates holding of requests, and `wait tForHol d` only returns when holding of requests has been completed.

See Also

`hol d`, `wait tForDeacti vate`, `Communi cator: : wait tForShutdown`.

B.40 `Ice: : Obj ectAdapterDeacti vatedExcepti on`

Overview

`local excepti on Obj ectAdapterDeacti vatedExcepti on`

This exception is raised if an attempt is made to use a deactivated `Obj ectAdapter`.

See Also

ObjectAdapter::deactivate, Communicator::shutdown.

name

string name;

Name of the adapter.

B.41 Ice::ObjectAdapterIdInUseException

Overview

Local exception ObjectAdapterIdInUseException

This exception is raised if an ObjectAdapter cannot be activated because the Locator detected another active ObjectAdapter with the same adapter id.

id

string id;

Adapter id.

B.42 Ice::ObjectFactory

Overview

Local interface ObjectFactory

A factory for objects. Object factories are used in several places, for example, when receiving "objects by value" and when ::Freeze restores a persistent object. Object factories must be implemented by the application writer, and registered with the communicator.

Used By

Communicator::addObjectFactory, Communicator::findObjectFactory.

See Also

::Freeze.

create

```
Object create(string type);
```

Create a new object for a given object type. The type is the absolute Slice type name, i.e., the name relative to the unnamed top-level Slice module. For example, the absolute Slice type name for interfaces of type Bar in the module Foo is

```
::Foo::Bar.
```

The leading "::" is required.

Parameters

type

The object type.

Return Value

The object created for the given type, or nil if the factory is unable to create the object.

destroy

```
void destroy();
```

Called when the factory is removed from the communicator, or if the communicator is destroyed.

See Also

Communicator::removeObjectFactory, Communicator::destroy.

B.43 Ice::ObjectNotExistException

Overview

Local exception `ObjectNotExistException` extends `RequestFailedException`

This exception is raised if an object does not exist on the server.

B.44 Ice::ObjectNotFoundException

Overview

exception `ObjectNotFoundException`

This exception is raised if an object cannot be found.

B.45 Ice::OperationMode

Overview

enum `OperationMode`

The `OperationMode` determines the skeleton signature (for C++), as well as the retry behavior of the Ice run time for an operation invocation in case of a (potentially) recoverable error.

Used By

`Current::mode`.

Normal

Normal

Ordinary operations have `Normal` mode. These operations modify object state; invoking such an operation twice in a row has different semantics than invoking it

once. The Ice run time guarantees that it will not violate at-most-once semantics for Normal operations.

Nonmutating

Nonmutating

Operations that use the Slice Nonmutating keyword must not modify object state. For C++, nonmutating operations generate const member functions in the skeleton. In addition, the Ice run time will attempt to transparently recover from certain run-time errors by re-issuing a failed request and propagate the failure to the application only if the second attempt fails.

Idempotent

Idempotent

Operations that use the Slice Idempotent keyword can modify object state, but invoking an operation twice in a row must result in the same object state as invoking it once. For example, $x = 1$ is an idempotent statement, whereas $x += 1$ is not. For idempotent operations, the Ice run-time uses the same retry behavior as for nonmutating operations in case of a potentially recoverable error.

B.46 Ice: : OperationNotExistException

Overview

Local exception OperationNotExistException extends RequestFailedException

This exception is raised if an operation for a given object does not exist on the server. Typically this is caused by either the client or the server using an outdated Slice specification.

B.47 Ice::Plugin

Overview

Local interface Plugin

A communicator plug-in. A plug-in generally adds a feature to a communicator, such as support for a protocol.

Derived Classes and Interfaces

: IceSSL: Plugin.

Used By

PluginManager::addPlugin, PluginManager::getPlugin.

destroy

```
void destroy();
```

Called when the communicator is being destroyed.

B.48 Ice::PluginInitializationException

Overview

Local exception PluginInitializationException

This exception indicates that a failure occurred while initializing a plug-in.

reason

```
string reason;
```

The reason for the failure.

B.49 Ice: :PluginManager

Overview

Local interface PluginManager

Each communicator has a plugin manager to administer the set of plug-ins.

Used By

Communicator: :getPluginManager.

addPlugin

```
void addPlugin(string name, Plugin pi);
```

Install a new plug-in.

Parameters

name

The plug-in's name.

pi

The plug-in.

destroy

```
void destroy();
```

Called when the communicator is being destroyed.

getPlugin

```
Plugin getPlugin(string name);
```

Obtain a plug-in by name.

Parameters

name

The plug-in's name.

Return Value

The plug-in.

B.50 `Ice: : Process`

Overview

`interface Process`

An administrative interface for process management. Managed servers must implement this interface and invoke `ObjectAdapter::setProcess` to register the process proxy.

A servant implementing this interface is a potential target for denial-of-service attacks, therefore proper security precautions should be taken. For example, the servant can use a UUID to make its identity harder to guess, and be registered in an object adapter with a secured endpoint.

`shutdown`

`void shutdown();`

Initiate a graceful shutdown.

See Also

`Communicator::shutdown`.

B.51 `Ice: : Properties`

Overview

`local interface Properties`

A property set used to configure Ice and Ice applications. Properties are key/value pairs, with both keys and values being strings. By convention, property keys should have the form *application-name[.category[.sub-category]].name*.

Used By

Communicator: :getProperties, clone.

clone

Properties clone();

Create a copy of this property set.

Return Value

A copy of this property set.

getCommandLineOptions

StringSeq getCommandLineOptions();

Get a sequence of command-line options that is equivalent to this property set. Each element of the returned sequence is a command-line option of the form *--key=value*.

Return Value

The command line options for this property set.

getPropertiesForPrefix

PropertyDict getPropertiesForPrefix(string prefix);

Get all properties whose keys begins with *prefix*. If *prefix* is an empty string, then all properties are returned.

Return Value

The matching property set.

getProperty

string getProperty(string key);

Get a property by key. If the property does not exist, an empty string is returned.

Parameters

key

The property key.

Return Value

The property value.

See Also

setProperty.

getPropertyAsInt

```
int getPropertyAsInt(string key);
```

Get a property as an integer. If the property does not exist, 0 is returned.

Parameters

key

The property key.

Return Value

The property value interpreted as an integer.

See Also

setProperty.

getPropertyAsIntWithDefault

```
int getPropertyAsIntWithDefault(string key, int value);
```

Get a property as an integer. If the property does not exist, the given default value is returned.

Parameters

key

The property key.

value

The default value to use if the property does not exist.

Return Value

The property value interpreted as an integer, or the default value.

See Also

setProperty.

getPropertyWithDefault

```
string getPropertyWithDefault(string key, string value);
```

Get a property by key. If the property does not exist, the given default value is returned.

Parameters

key

The property key.

value

The default value to use if the property does not exist.

Return Value

The property value or the default value.

See Also

setProperty.

Load

```
void Load(string file);
```

Load properties from a file.

Parameters

file

The property file.

parseCommandLineOptions

```
StringSeq parseCommandLineOptions(String prefix, StringSeq options);
```

Convert a sequence of command-line options into properties. All options that begin with *--prefix* are converted into properties. If the prefix is empty, all options that begin with *--* are converted to properties.

Parameters

prefix

The property prefix, or an empty string to convert all options starting with *--*.

options

The command-line options.

Return Value

The command-line options that do not start with the specified prefix, in their original order.

parseIceCommandLineOptions

```
StringSeq parseIceCommandLineOptions(StringSeq options);
```

Convert a sequence of command-line options into properties. All options that begin with one of the following prefixes are converted into properties: *--Ice*, *--IceBox*, *--IcePack*, *--IcePatch*, *--IceSSL*, *--IceStorm*, *--Freeze*, and *--Glacier*.

Parameters

options

The command-line options.

Return Value

The command-line options that do not start with one of the listed prefixes, in their original order.

setProperty

```
void setProperty(string key, string value);
```

Set a property. To unset a property, set it to the empty string.

Parameters

key

The property key.

value

The property value.

See Also

getProperty.

B.52 Ice: : ProtocolException

Overview

local exception ProtocolException

A generic exception base for all kinds of protocol error conditions.

Derived Exceptions

AbortBatchRequestException, BadMagicException, CloseConnectionException, CompressionException, CompressionNotSupportedException, ConnectionNotValidatedException, DatagramLimitException, IllegalMessageSizeException, MarshalException, UnknownMessageException, UnknownReplyStatusException, UnknownRequestIdException, UnsupportedEncodingException, UnsupportedProtocolException.

B.53 Ice: : ProxyParseException

Overview

local exception ProxyParseException

This exception is raised if there was an error while parsing a stringified proxy.

`str`

`string str;`

The string that could not be parsed.

B.54 Ice: : ProxyUnmarshal Exception

Overview

`local exception ProxyUnmarshalException` extends `MarshalException`

This exception is a specialization of `MarshalException` that is raised if inconsistent data is received while unmarshaling a proxy.

B.55 Ice: : RequestFailedException

Overview

`local exception RequestFailedException`

This exception is raised if a request failed. This exception, and all exceptions derived from `RequestFailedException`, are transmitted by the Ice protocol, even though they are declared `local`.

Derived Exceptions

`FacetNotExistException`, `ObjectNotExistException`, `OperationNotExistException`.

`facet`

`FacetPath facet;`

The facet to which the request was sent.

`id`

`Identity id;`

The identity of the Ice Object to which the request was sent.

`operation`

`string operation;`

The operation name of the request.

B.56 Ice::Router

Overview

`interface Router`

The Ice router interface. Routers can be set either globally with `Communicator::setDefaultRouter`, or with `ice_router` on specific proxies.

The router interface is intended to be used by Ice internals and by router implementations. Regular user code should not attempt to use any functionality of this interface directly.

Derived Classes and Interfaces

`::Glacier::Router`.

`addProxy`

`void addProxy(Object* proxy);`

Add new proxy information to the router's routing table.

Parameters

`proxy`

The proxy to add.

getClientProxy

```
Object* getClientProxy();
```

Get the router's client proxy, i.e., the proxy to use for forwarding requests from the client to the router.

Return Value

The router's client proxy.

getServerProxy

```
Object* getServerProxy();
```

Get the router's server proxy, i.e., the proxy to use for forwarding requests from the server to the router.

Return Value

The router's server proxy.

B.57 Ice: : ServantLocator

Overview

Local interface ServantLocator

The servant locator, which is called by the object adapter to locate a servant that is not found in its active servant map.

Derived Classes and Interfaces

: : Freeze: : Evictor.

Used By

ObjectAdapter: : addServantLocator, ObjectAdapter: : findServantLocator.

See Also

ObjectAdapter, ObjectAdapter: : addServantLocator, ObjectAdapter: : findServantLocator.

deactivate

```
void deactivate(string category);
```

Called when the object adapter in which this servant locator is installed is deactivated.

Parameters

category

Indicates for which category the servant locator is being deactivated.

See Also

ObjectAdapter::deactivate, Communicator::shutdown, Communicator::destroy.

finished

```
void finished(Current curr, Object servant, LocalObject cookie);
```

Called by the object adapter after a request has been made. This operation is only called if locate was called prior to the request and returned a non-null servant. This operation can be used for cleanup purposes after a request.

Parameters

curr

Information about the current operation call for which a servant was located by locate.

servant

The servant that was returned by locate.

cookie

The cookie that was returned by locate.

See Also

ObjectAdapter, Current, locate.

locate

```
Object locate(Current curr, out LocalObject cookie);
```

Called by the object adapter before a request is made when a servant cannot be found in the object adapter's active servant map. Note that the object adapter does not automatically insert the returned servant into its active servant map. This must be done by the servant locator implementation, if this is desired.

Important: If you call `locate` from your own code, you must also call `finished` when you have finished using the servant, provided that a non-null servant was returned. Otherwise you will get undefined behavior if you use Servant Locators such as the `::Freeze::Evictor`.

Parameters

`curr`

Information about the current operation for which a servant is required.

`cookie`

A "cookie" that will be passed to `finished`.

Return Value

The located servant, or null if no suitable servant has been found.

See Also

`ObjectAdapter`, `Current`, `finished`.

B.58 `Ice::ServerNotFoundException`

Overview

`exception ServerNotFoundException`

This exception is raised if a server cannot be found.

B.59 `Ice::SocketException`

Overview

Local exception `SocketException` extends `SyscallException`

This exception is a specialization of `System.IO.IOException` for socket errors.

Derived Exceptions

`ConnectFailedException`, `ConnectionLostException`.

B.60 Ice::Stats

Overview

Local interface `Stats`

An interface `Ice` uses to report statistics, such as how much data is sent or received. Applications must provide their own `Stats` by implementing this interface and installing it in a communicator.

Used By

`Communicator::getStats`, `Communicator::setStats`.

bytesReceived

```
void bytesReceived(string protocol, int num);
```

Callback to report that data has been received.

Parameters

`protocol`

The protocol over which data has been received (for example "tcp", "udp", or "ssl").

`num`

How many bytes have been received.

bytesSent

```
void bytesSent(string protocol, int num);
```

Callback to report that data has been sent.

Parameters

protocol

The protocol over which data has been sent (for example "tcp", "udp", or "ssl").

num

How many bytes have been sent.

B.61 `Ice::SyscallException`

Overview

Local exception `SyscallException`

This exception is raised if a system error occurred in the server or client process. There are many possible causes for such a system exception. For details on the cause, error should be inspected.

Derived Exceptions

`SocketException`.

error

`int error;`

The error number describing the system exception. For C++ and Unix, this is equivalent to `errno`. For C++ and Windows, this is the value returned by `GetLastError()` or `WSAGetLastError()`.

B.62 `Ice::TimeoutException`

Overview

Local exception `TimeoutException`

This exception indicates a timeout condition.

Derived Exceptions

`CloseTimeoutException`, `ConnectTimeoutException`, `ConnectionTimeoutException`.

B.63 `Ice::TwowayOnlyException`

Overview

`LocalException` `TwowayOnlyException`

This exception is raised if an attempt is made to invoke an operation with `ice_oneway`, `ice_batchOneway`, `ice_datagram`, or `ice_batchDatagram` and the operation has a return value, out parameters, or an exception specification.

`operation`

`string operation`;

The name of the operation that was invoked.

B.64 `Ice::UnknownException`

Overview

`LocalException` `UnknownException`

This exception is raised if an operation call on a server raises an unknown exception. For example, for C++, this exception is raised if the server throws a C++ exception that is not directly or indirectly derived from `Ice::LocalException` or `Ice::UserException`.

Derived Exceptions

`UnknownLocalException`, `UnknownUserException`.

unknown

string unknown;

A textual representation of the unknown exception. This field may or may not be set, depending on the security policy of the server. Some servers may give this information to clients for debugging purposes, while others may not wish to disclose information about server internals.

B.65 Ice: : UnknownLocal Exception

Overview

Local exception UnknownLocalException extends UnknownException

This exception is raised if an operation call on a server raises a local exception. Because local exceptions are not transmitted by the Ice protocol, the client receives all local exceptions raised by the server as UnknownLocalException. The only exception to this rule are all exceptions derived from RequestFailedException, which are transmitted by the Ice protocol even though they are declared Local.

B.66 Ice: : UnknownMessageException

Overview

Local exception UnknownMessageException extends ProtocolException

This exception is a specialization of ProtocolException, indicating that an unknown protocol message has been received.

B.67 Ice: : UnknownReplyStatusExcepti on

Overview

Local excepti on UnknownReplyStatusExcepti on extends Protocol Excepti on

This exception is a specialization of Protocol Excepti on, indicating that an unknown reply status has been received.

B.68 Ice: : UnknownRequestIdExcepti on

Overview

Local excepti on UnknownRequestIdExcepti on extends Protocol Excepti on

This exception is a specialization of Protocol Excepti on, indicating that a response for an unknown request id has been received.

B.69 Ice: : UnknownUserExcepti on

Overview

Local excepti on UnknownUserExcepti on extends UnknownExcepti on

This exception is raised if an operation call on a server raises a user exception that is not declared in the exception's throws clause. Such undeclared exceptions are not transmitted from the server to the client by the Ice protocol, but instead the client just gets an UnknownUserExcepti on. This is necessary in order to not violate the contract established by an operation's signature: Only local exceptions and user exceptions declared in the throws clause can be raised.

B.70 Ice: : Unmarshal OutOfBoundsExcepti on

Overview

Local excepti on Unmarshal OutOfBoundsExcepti on extends Marshal Excepti on

This exception is a specialization of Marshal Excepti on that is raised if an out-of-bounds condition occurs during unmarshaling.

B.71 Ice: : UnsupportedEncodi ngExcepti on

Overview

Local excepti on UnsupportedEncodi ngExcepti on extends Protocol Excepti on

This exception is a specialization of Protocol Excepti on, indicating that an unsupported data encoding version has been encountered.

badMaj or

int badMaj or;

The major version number of the unsupported encoding.

badMi nor

int badMi nor;

The minor version number of the unsupported encoding.

maj or

int maj or;

The major version number of the encoding that is supported.

`minor`

`int minor;`

The highest minor version number of the encoding that can be supported.

B.72 `Ice::UnsupportedProtocolException`

Overview

`local exception UnsupportedProtocolException` extends `ProtocolException`

This exception is a specialization of `ProtocolException`, indicating that an unsupported protocol version has been encountered.

`badMajor`

`int badMajor;`

The major version number of the unsupported protocol.

`badMinor`

`int badMinor;`

The minor version number of the unsupported protocol.

`major`

`int major;`

The major version number of the protocol that is supported.

`minor`

`int minor;`

The highest minor version number of the protocol that can be supported.

B.73 Ice: : VersionMismatchException

Overview

Local exception VersionMismatchException

This exception is raised if the Ice library version does not match the Ice header files version.

B.74 Freeze

Overview

module Freeze

Freeze provides automatic persistence for Ice servants. Freeze provides a binary data format for maximum speed, as well as an XML data format for maximum flexibility. Freeze databases using the XML data format can be migrated when the Slice description of the persistent data changes.

Key

sequence<byte> Key;

A database key, represented as a sequence of bytes.

Value

sequence<byte> Value;

A database value, represented as a sequence of bytes.

B.75 Freeze: : Connecti on

Overview

Local i nterface Connecti on

A connection to a database (database environment with Berkeley DB). If you want to use a connection concurrently in multiple threads, you need to serialize access to this connection.

begi nTransacti on

Transacti on begi nTransacti on();

Create a new transaction. Only one transaction at a time can be associated with a connection.

Return Value

The new transaction

Exceptions

rai ses

TransactionAlreadyInProgressException if a transaction is already associated with this connection.

cl ose

voi d cl ose();

Closes this connection. If there is an associated transaction, it is rolled back.

currentTransacti on

Transacti on currentTransacti on();

Returns the transaction associated with this connection.

Return Value

The current transaction if there is one, null otherwise.

getCommunicator

```
::Ice::Communicator getCommunicator();
```

Returns the communicator associated with this connection

getName

```
string getName();
```

The name of the connected system (e.g. Berkeley DB environment)

B.76 Freeze: : DatabaseException

Overview

local exception DatabaseException

A Freeze database exception.

Derived Exceptions

DeadlockException, NotFoundException.

See Also

DB, Evaluator, Connection.

message

```
string message;
```

A message describing the reason for the exception.

B.77 Freeze: : DeadlockException

Overview

local exception DeadlockException extends DatabaseException

A Freeze database deadlock exception. Applications can react to this exception by aborting and trying the transaction again.

B.78 Freeze: : EmptyFacetPathException

Overview

Local exception EmptyFacetPathException

This exception is raised if an empty : : Ice: : FacetPath is passed to Evictor: : addFacet or Evictor: : removeFacet.

B.79 Freeze: : Evictor

Overview

Local interface Evictor extends : : Ice: : ServantLocator

An automatic Ice object persistence manager, based on the evictor pattern. The evictor is a servant locator implementation that stores the persistent state of its objects in a database. Any number of objects can be registered with an evictor, but only a configurable number of servants are active at a time. These active servants reside in a queue; the least recently used servant in the queue is the first to be evicted when a new servant is activated.

See Also

ServantInitializer.

addFacet

```
void addFacet(: : Ice: : Identity identity, : : Ice: : FacetPath facet,
Object servant);
```

Adds a new persistent facet to this object.

Parameters

`i d e n t i t y`

The identity of the target Ice object

`f a c e t`

The facet path.

`s e r v a n t`

The servant for the Ice object.

Exceptions

`D a t a b a s e E x c e p t i o n`

Raised if a database failure occurred.

`E v i c t o r D e a c t i v a t e d E x c e p t i o n`

Raised if a the evictor has been deactivated.

`E m p t y F a c e t P a t h E x c e p t i o n`

Raised if the facet path is empty.

See Also

`::Ice::I d e n t i t y, r e m o v e F a c e t.`

createObject

```
void createObject(::Ice::I d e n t i t y i d e n t i t y, O b j e c t s e r v a n t);
```

Create a new Ice object for this evictor. The state of the servant passed to this operation is saved in the evictor's persistent store. If the object already exists, it is updated.

Parameters

`i d e n t i t y`

The identity of the Ice object to create.

`s e r v a n t`

The servant for the Ice object.

Exceptions

DatabaseExcepti on

Raised if a database failure occurred.

Evi ctorDeacti vatedExcepti on

Raised if a the evictor has been deactivated.

See Also

:: Ice:: I denti ty, destroyObj ect.

destroyObj ect

```
void destroyObject(:: Ice:: I denti ty i denti ty);
```

Permanently destroy an Ice object by removing it from the evictor's persistent store. If the object does not exist, this operation does nothing.

Parameters

i denti ty

The identity of the Ice object to destroy.

Exceptions

DatabaseExcepti on

Raised if a database failure occurred.

Evi ctorDeacti vatedExcepti on

Raised if a the evictor has been deactivated.

See Also

:: Ice:: I denti ty, createObj ect.

getI terator

```
Evi ctorI terator getIterator(int batchSize, bool loadServants);
```

Get an iterator for the identities managed by the evictor.

Parameters

`batchSize`

Internally, the Iterator retrieves the identities in batches of size `batchSize`.
Selecting a small `batchSize` can have an adverse effect on performance.

`loadServants`

If true, attempt to load the corresponding servants in the Evictor. The servants may not be loaded if a save occurs while the batch is retrieved.

Return Value

A new iterator.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

`getSize`

```
int getSize();
```

Get the size of the evictor's servant queue.

Return Value

The size of the servant queue.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

See Also

`setSize`.

`hasObject`

```
bool hasObject(Ice::Identity id);
```

Returns true if the given identity is managed by the evictor.

Return Value

true if the identity is managed by the evictor, false otherwise.

Exceptions

`DatabaseException`

Raised if a database failure occurred.

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

installServantInitializer

```
void installServantInitializer(ServantInitializer initializer);
```

Install a servant initializer for this evictor.

Parameters

`initializer`

The servant initializer to install. Subsequent calls overwrite any previously set value. A null value removes an existing servant initializer.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

See Also

`ServantInitializer`.

removeAllFacets

```
void removeAllFacets(::Ice::Identity identity);
```

Permanently remove all the facets from the object.

Parameters

`identity`

The identity of the target Ice object.

Exceptions

DatabaseExcepti on

 Raised if a database failure occurred.

Evi ctorDeacti vatedExcepti on

 Raised if a the evictor has been deactivated.

See Also

`::Ice::Identi ty, removeFacet.`

removeFacet

Object removeFacet(`::Ice::Identi ty i denti ty`, `::Ice::FacetPath facet`);

Permanently remove this facet from the object.

Parameters

`i denti ty`

 The identity of the target Ice object.

`facet`

 The facet path.

Return Value

The removed facet.

Exceptions

DatabaseExcepti on

 Raised if a database failure occurred.

Evi ctorDeacti vatedExcepti on

 Raised if a the evictor has been deactivated.

EmptyFacetPathExcepti on

 Raised if the facet path is empty.

See Also

`::Ice::Identi ty, addFacet.`

`setSize`

```
void setSize(int sz);
```

Set the size of the evictor's servant queue. This is the maximum number of servants the evictor keeps active. Requests to set the queue size to a value smaller than zero are ignored.

Parameters

`sz`

The size of the servant queue. If the evictor currently holds more than `setSize` servants in its queue, it evicts enough servants to match the new size. Note that this operation can block if the new queue size is smaller than the current number of servants that are servicing requests. In this case, the operation waits until a sufficient number of servants complete their requests.

Exceptions

`EvictorDeactivatedException`

Raised if a the evictor has been deactivated.

See Also

`getSize`.

B.80 Freeze: : `EvictorDeactivatedException`

Overview

Local exception `EvictorDeactivatedException`

This exception is raised if the evictor has been deactivated.

B.81 Freeze: : `EvictorIterator`

Overview

Local interface `EvictorIterator`

An iterator for the objects managed by the evictor. Note that an `EvictorIterator` is not thread-safe: the application needs to serialize access to a given `EvictorIterator`, for example by using it in just one thread.

Used By

`Evictor::getIterator`.

See Also

`Evictor`.

hasNext

```
bool hasNext();
```

Determines if the iteration has more elements.

Return Value

True if the iterator has more elements, false otherwise.

Exceptions

`DatabaseException`

Raised if a database failure occurs while retrieving a batch of objects.

next

```
::Ice::Identity next();
```

Obtains the next identity in the iteration.

Return Value

s The next identity in the iteration.

Exceptions

`NoSuchElementException`

Raised if there is no further elements in the iteration.

`DatabaseException`

Raised if a database failure occurs while retrieving a batch of objects.

B.82 Freeze: : EvictorStorageKey

Overview

struct EvictorStorageKey

The key of the Evictor persistent map.

facet

::Ice::FacetPath facet;

identity

::Ice::Identity identity;

B.83 Freeze: : InvalidPositionException

Overview

local exception InvalidPositionException

This Freeze Iterator is not on a valid position, for example this position has been erased.

B.84 Freeze: : NoSuchElementException

Overview

local exception NoSuchElementException

This exception is raised if there are no further elements in the iteration.

B.85 Freeze: : NotFoundExcepti on

Overview

Local exception `NotFoundExcepti on` extends `DatabaseExcepti on`

A Freeze database exception, indicating that a database record could not be found.

B.86 Freeze: : Obj ectRecord

Overview

struct `Obj ectRecord`

The evictor uses a map of `Evi ctorStorageKey` to `Obj ectRecord` as its persistent storage.

servant

`Obj ect` servant;

stats

`Stati stics` stats;

B.87 Freeze: : ServantIni ti al i zer

Overview

Local interface `ServantIni ti al i zer`

A servant initializer is installed in an evictor and provides the application with an opportunity to perform custom servant initialization.

Used By

Evictor: : installServantInitializer.

See Also

Evictor.

initialize

```
void initialize(::Ice::ObjectAdapter adapter, ::Ice::Identity
identity, Object servant);
```

Called whenever the evictor creates a new servant. This operation allows application code to perform custom servant initialization after the servant has been created by the evictor and its persistent state has been restored.

Parameters

adapter

The object adapter in which the evictor is installed.

identity

The identity of the Ice object for which the servant was created.

servant

The servant to initialize.

See Also

::Ice::Identity.

B.88 Freeze: : Statistics

Overview

```
struct Statistics
```

The evictor maintains statistics about each object.

Used By

ObjectRecord: : stats.

`avgSaveTime`

`long avgSaveTime;`

The average time between saves, in milliseconds.

`creationTime`

`long creationTime;`

The time the object was created, in milliseconds since Jan 1, 1970 0:00.

`lastSaveTime`

`long lastSaveTime;`

The time the object was last saved, in milliseconds relative to `creationTime`.

B.89 Freeze: : Transaction

Overview

`local interface Transaction`

A transaction. If you want to use a transaction concurrently in multiple threads, you need to serialize access to this transaction.

Used By

`Connection: : beginTransaction`, `Connection: : currentTransaction`.

`commit`

`void commit();`

Commit this transaction.

`rollback`

`void rollback();`

Roll back this transaction.

B.90 Freeze: : TransactionAlreadyInProgressException

Overview

Local exception TransactionAlreadyInProgressException

B.91 IceBox

Overview

module IceBox

IceBox is an application server specifically for Ice applications. IceBox can easily run and administer Ice services that are dynamically loaded as a DLL, shared library, or Java class.

B.92 IceBox: : FailureException

Overview

Local exception FailureException

Indicates a failure occurred. For example, if a service encounters an error during initialization, or if the service manager is unable to load a service executable.

reason

string reason;

The reason for the failure.

B.93 IceBox: : FreezeService

Overview

Local interface FreezeService extends ServiceBase

A Freeze application service managed by a ServiceManager.

See Also

ServiceBase.

start

```
void start(string name, ::Ice::Communicator* communicator,  
           ::Ice::StringSeq args, string envName);
```

Start the service. The given communicator is created by the ServiceManager for use by the service. This communicator may also be used by other services, depending on the service configuration. The database environment is created by the ServiceManager for the exclusive use of the service.

The ServiceManager owns the communicator and the database environment, and is responsible for destroying them.

Parameters

name

The service's name, as determined by the configuration.

communicator

A communicator for use by the service.

args

The service arguments that were not converted into properties.

envName

The name of the Freeze database environment.

Exceptions

FailureException

Raised if start failed.

B.94 IceBox: : Service

Overview

Local interface Service extends ServiceBase

A standard application service managed by a ServiceManager.

See Also

ServiceBase.

start

```
void start(string name, ::Ice::Communicator communicator,  
::Ice::StringSeq args);
```

Start the service. The given communicator is created by the ServiceManager for use by the service. This communicator may also be used by other services, depending on the service configuration.

The ServiceManager owns this communicator, and is responsible for destroying it.

Parameters

name

The service's name, as determined by the configuration.

communicator

A communicator for use by the service.

args

The service arguments that were not converted into properties.

Exceptions

FailureException

Raised if start failed.

B.95 IceBox: : ServiceBase

Overview

Local interface ServiceBase

Base interface for an application service managed by a ServiceManager.

Derived Classes and Interfaces

FreezeService, Service.

See Also

ServiceManager, Service, FreezeService.

stop

```
void stop();
```

Stop the service.

B.96 IceBox: : ServiceManager

Overview

interface ServiceManager

Administers a set of Service instances.

See Also

Service.

shutdown

```
void shutdown();
```

Shutdown all services. This will cause Service: : stop to be invoked on all configured services.

B.97 IcePack

Overview

module IcePack

IcePack is a server activation and deployment tool. IcePack, simplifies the complex task of deploying applications in a heterogenous computer network.

B.98 IcePack: : AdapterDeploymentException

Overview

exception AdapterDeploymentException extends DeploymentException

This exception is raised if an error occurs during adapter registration.

id

string id;

The id of the adapter that could not be registered.

B.99 IcePack: : AdapterNotExistException

Overview

exception AdapterNotExistException

This exception is raised if an adapter does not exist.

B.100 IcePack: : Admin

Overview

interface Admin

The IcePack administrative interface.

Warning: Allowing access to this interface is a security risk! Please see the IcePack documentation for further information.

addApplication

void addApplication(string descriptor, ::Ice::StringSeq targets)
throws DeploymentException;

Add an application to IcePack. An application is a set of servers.

Parameters

descriptor

The application descriptor.

targets

The optional targets to deploy. A target is a list of components separated by dots and a target name. For example, the "debug" target of "service1" in "server1" will be deployed if the target "server1.service1.debug" is specified.

Exceptions

DeploymentException

Raised if application deployment failed.

See Also

removeApplication.

addObject

void addObject(Object* obj) throws ObjectExistsException,
ObjectDeploymentException;

Add an object to the object registry. IcePack will get the object type by calling `ice_id` on the given proxy. The object must be reachable.

Parameters

`obj`

The object to be added to the registry.

Exceptions

`ObjectDeploymentException`

Raised if an error occurred while trying to register this object. This can occur if the type of the object cannot be determined because the object is not reachable.

`ObjectExistsException`

Raised if the object is already registered.

`addObjectWithType`

```
void addObjectWithType(Object* obj, string type) throws
ObjectExistsException;
```

Add an object to the object registry and explicitly specify its type.

Parameters

`obj`

The object to be added to the registry.

`type`

The object type.

Exceptions

`ObjectExistsException`

Raised if the object is already registered.

`addServer`

```
void addServer(string node, string name, string path, string
libraryPath, string descriptor, ::Ice::StringSeq targets) throws
DeploymentException, NodeUnreachableException;
```

Add a server to an IcePack node.

Parameters

node

The name of the node where the server will be deployed.

name

The server name.

path

The server path. For C++ servers, this is the path of the executable. For C++ icebox, this is the path of the C++ icebox executable or, if empty, IcePack will rely on the PATH to find it. For a Java server or Java icebox, this is the path of the java command or, if empty, IcePack will rely on the PATH to find it.

librarypath

Specify the LD_LIBRARY_PATH value for C++ servers or the CLASSPATH value for Java servers.

descriptor

The server deployment descriptor.

targets

The optional targets to deploy. A target is a list of components separated by dots and a target name. For example, the "debug" target of "service1" in "server1" will be deployed if the target "server1.service1.debug" is specified.

Exceptions

DeploymentException

Raised if server deployment failed.

NodeUnreachableException

Raised if the node could not be reached.

See Also

removeServer.

getAdapterEndpoints

string getAdapterEndpoints(string id) throws
AdapterNotExistsException, NodeUnreachableException;

Get the list of endpoints for an adapter.

Parameters

`id`

The adapter id.

Return Value

The stringified adapter endpoints.

Exceptions

`AdapterNotExistException`

Raised if the adapter is not found.

getAdapterIds

```
::Ice::StringSeq getAdapterIds();
```

Get all the adapter ids registered with IcePack.

Return Value

The adapter ids.

getNodeNames

```
::Ice::StringSeq getNodeNames();
```

Get all the IcePack nodes currently registered.

Return Value

The node names.

getServerNames

```
::Ice::StringSeq getServerNames();
```

Get all the server names registered with IcePack.

Return Value

The server names.

See Also

getServerDescription, getServerState.

getServerActivation

ServerActivation getServerActivation(string name) throws
ServerNotExistException, NodeUnreachableException;

Get the server's activation mode.

Parameters

name

Must match the name of ServerDescription: : name.

Return Value

The server activation mode.

Exceptions

ServerNotExistException

Raised if the server is not found.

NodeUnreachableException

Raised if the node could not be reached.

See Also

getServerDescription, getServerState, getAllServerNames.

getServerDescription

ServerDescription getServerDescription(string name) throws
ServerNotExistException, NodeUnreachableException;

Get a server's description.

Parameters

name

Must match the name of `ServerDescription`: : name.

Return Value

The server description.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerState`, `getServerPid`, `getAllServerNames`.

getServerPid

`int getServerPid(string name) throws ServerNotExistsException, NodeUnreachableException;`

Get a server's system process id. The process id is operating system dependent.

Parameters

name

Must match the name of `ServerDescription`: : name.

Return Value

The server process id.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerState`, `getAllServerNames`.

getServerState

`ServerState getServerState(string name)` throws `ServerNotExistException`, `NodeUnreachableException`;

Get a server's state.

Parameters

`name`

Must match the name of `ServerDescription`: `name`.

Return Value

The server state.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerPid`, `getAllServerNames`.

pingNode

`bool pingNode(string name)` throws `NodeNotExistException`;

Ping an IcePack node to see if it is active.

Return Value

true if the node ping succeeded, false otherwise.

removeApplication

void removeApplication(string descriptor) throws DeploymentException;

Remove an application from IcePack.

Parameters

descriptor

The application descriptor.

See Also

addApplication.

removeObject

void removeObject(Object* obj) throws ObjectNotExistException;

Remove an object from the object registry.

Parameters

obj

The object to be removed from the registry.

Exceptions

ObjectNotExistException

Raised if the object cannot be found.

removeServer

void removeServer(string name) throws DeploymentException, ServerNotExistException, NodeUnreachableException;

Remove a server from an IcePack node.

Parameters

name

Must match the name of ServerDescription: : name.

Exceptions

`DeploymentException`

Raised if the server deployer failed to remove the server.

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`addServer`.

`sendSignal`

`void sendSignal (string name, string signal)` throws `ServerNotExistException`, `NodeUnreachableException`, `BadSignalException`;

Send signal to a server.

Parameters

`name`

Must match the name of `ServerDescription : name`.

`signal`

The signal, for example `SIGTERM` or `15`.

Exceptions

`ServerNotExistException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

`BadSignalException`

Raised if the signal is not recognized by the target server.

setServerActivation

```
void setServerActivation(string name, ServerActivation mode)
throws ServerNotExistsException, NodeUnreachableException;
```

Set the server's activation mode.

Parameters

name

Must match the name of `ServerDescription`: : name.

Return Value

The server activation mode.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

See Also

`getServerDescription`, `getServerState`, `getAllServerNames`.

shutdown

```
void shutdown();
```

Shut down the IcePack registry.

shutdownNode

```
void shutdownNode(string name) throws NodeNotExistsException;
```

Shutdown an IcePack node.

startServer

```
bool startServer(string name) throws ServerNotExistsException,
NodeUnreachableException;
```

Start a server.

Parameters

name

Must match the name of `ServerDescription`: : name.

Return Value

True if the server was successfully started, false otherwise.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

stopServer

`void stopServer(string name) throws ServerNotExistsException, NodeUnreachableException;`

Stop a server.

Parameters

name

Must match the name of `ServerDescription`: : name.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

writeMessage

`void writeMessage(string name, string message, int fd) throws ServerNotExistsException, NodeUnreachableException;`

Write message on server stdout or stderr

Parameters

name

Must match the name of `ServerDescription`: : name.

message

The message.

fd

1 for stdout, 2 for stderr.

Exceptions

`ServerNotExistsException`

Raised if the server is not found.

`NodeUnreachableException`

Raised if the node could not be reached.

B.101 `IcePack::BadSignalException`

Overview

exception `BadSignalException`

This exception is raised if an unknown signal was sent to to a server.

B.102 `IcePack::DeploymentException`

Overview

exception `DeploymentException`

A generic exception base for all kinds of deployment error exception.

Derived Exceptions

AdapterDeploymentException, ObjectDeploymentException, ParserDeploymentException, ServerDeploymentException.

component

string component;

The path of the component that caused the deployment to fail. The path is a dot-separated list of component names. It always starts with the node name, followed by the server name, and finally the service name.

reason

string reason;

The reason for the failure.

B.103 IcePack: : NodeNotExistException

Overview

exception NodeNotExistException

This exception is raised if a node does not exist.

B.104 IcePack: : NodeUnreachableException

Overview

exception NodeUnreachableException

This exception is raised if a node could not be reached.

B.105 IcePack: : ObjectDeploymentException

Overview

exception ObjectDeploymentException extends DeploymentException

This exception is raised if an error occurs during object registration.

proxy

Object* proxy;

The object that could not be registered.

B.106 IcePack: : ObjectExistsException

Overview

exception ObjectExistsException

This exception is raised if an object already exists.

B.107 IcePack: : ObjectNotExistException

Overview

exception ObjectNotExistException

This exception is raised if an object does not exist.

B.108 IcePack: : ParserDeploymentException

Overview

exception ParserDeploymentException extends DeploymentException

This exception is raised if an error occurs while parsing the XML descriptor of a component.

B.109 IcePack: : Query

Overview

interface Query

The IcePack query interface. This interface is accessible to Ice clients who wish to lookup objects.

findAllObjectsWithType

::Ice::ObjectProxySeq findAllObjectsWithType(string type) throws
ObjectNotExistException;

Find all the objects with the given type.

Parameters

type

The object type.

Return Value

The proxies.

Exceptions

ObjectNotExistException

Raised if no objects can be found.

findObjectById

Object* findObjectById(::Ice::Identity id) throws
ObjectNotExistException;

Find an object by identity.

Parameters

`id`

The identity.

Return Value

The proxy.

Exceptions

`ObjectNotExistException`

Raised if no objects can be found.

findObjectByType

`Object* findObjectByType(string type) throws
ObjectNotExistException;`

Find an object by type.

Parameters

`type`

The object type.

Return Value

The proxy.

Exceptions

`ObjectNotExistException`

Raised if no objects can be found.

B.110 IcePack: : ServerActivation

Overview

`enum ServerActivation`

The server activation mode.

Used By

Admin: : getServerActivation, Admin: : setServerActivation.

OnDemand

OnDemand

The server is activated on demand if a client requests one of the server's adapter endpoints and the server is not already running.

Manual

Manual

The server is activated manually through the administrative interface.

B.111 IcePack: : ServerDeploymentException

Overview

exception ServerDeploymentException extends DeploymentException

This exception is raised if an error occurs while deploying a server.

server

string server;

The name of the server that could not be deployed.

B.112 IcePack: : ServerDescription

Overview

struct ServerDescription

The server description.

Used By

Admin: :getServerDescription.

args

::Ice::StringSeq args;

The server arguments.

descriptor

string descriptor;

The path of the deployment descriptor used to deploy the server.

envs

::Ice::StringSeq envs;

The server environment variables.

name

string name;

Server name.

node

string node;

The name of the node where the server is deployed.

path

string path;

The server path.

`pwd`

`string pwd;`

The server working directory.

`serviceManager`

`::IceBox::ServiceManager* serviceManager;`

The IceBox service manager proxy if the server is an IceBox service, otherwise a null proxy.

`targets`

`::Ice::StringSeq targets;`

Targets used to deploy the server.

B.113 `IcePack::ServerNotExistsException`

Overview

`exception ServerNotExistsException`

This exception is raised if a server does not exist.

B.114 `IcePack::ServerState`

Overview

`enum ServerState`

An enumeration representing the state of the server.

Used By

`Admin::getServerState.`

Inactive

Inactive

The server is not running.

Activating

Activating

The server is being activated and will change to the active state if the server fork succeeded or to the Inactive state if it failed.

Active

Active

The server is running.

Deactivating

Deactivating

The server is being deactivated.

Destroying

Destroying

The server is being destroyed.

Destroyed

Destroyed

The server is destroyed.

B.115 IceSSL

Overview

module IceSSL

IceSSL is a dynamic SSL transport plug-in for the Ice core. It provides authentication, encryption, and message integrity, using the industry-standard SSL protocol.

B.116 IceSSL: : CertificateException

Overview

Local exception CertificateException extends SslException

A root exception class for all exceptions related to public key certificates.

Derived Exceptions

CertificateParseException, CertificateSignatureException, CertificateSigningException.

B.117 IceSSL: : CertificateKeyMatchException

Overview

Local exception CertificateKeyMatchException extends ContextException

When loading a public and private key pair into a Context, the load succeeded, but the private key and public key (certificate) did not match.

B.118 IceSSL: : CertificateLoadException

Overview

Local exception CertificateLoadException extends ContextException

Indicates that a problem occurred while loading a certificate into a Context from either a memory buffer or from a file.

B.119 IceSSL: : CertificateParseException

Overview

Local exception CertificateParseException extends CertificateException

Indicates that IceSSL was unable to parse the provided public key certificate into a form usable by the underlying SSL implementation.

B.120 IceSSL: : CertificateSignatureException

Overview

Local exception CertificateSignatureException extends CertificateException

Indicates that the signature verification of a newly signed temporary RSA certificate has failed.

B.121 IceSSL: : CertificateSigningException

Overview

Local exception CertificateSigningException extends CertificateException

Indicates that a problem occurred while signing certificates during temporary RSA certificate generation.

B.122 IceSSL: : CertificateVerificationException

Overview

Local exception CertificateVerificationException extends ShutdownException

Indicates a problem occurred during the certificate verification phase of the SSL handshake. This is currently only thrown by server connections.

B.123 IceSSL: : CertificateVerifier

Overview

Local interface CertificateVerifier

The CertificateVerifier is the base interface for all classes that define additional application-specific certificate verification rules. These rules are evaluated during the SSL handshake by an instance of a class derived from CertificateVerifier. The methods defined in derived interfaces will depend upon the requirements of the underlying SSL implementation. Default certificate verifier implementations can be obtained via the Plugin. As this is simply a base class for purposes of derivation, no methods are defined.

Used By

Plugin: getDefaultCertificateVerifier, Plugin: getSingleCertificateVerifier,
Plugin: setCertificateVerifier.

See Also

Plugin.

setContext

```
void setContext(ContextType type);
```

Set the context type of this Certificate Verifier.

Parameters

type

The type of context that is using this CertificateVerifier, Client, Server or ClientServer.

B.124 IceSSL: : CertificateVerifierTypeException

Overview

Local exception CertificateVerifierTypeException extends SslException

This exception indicates that the provided CertificateVerifier was not derived from the proper base class, and thus, does not provide the appropriate interface.

B.125 IceSSL: : ConfigParseException

Overview

Local exception ConfigParseException extends SslException

This exception indicates that a problem occurred while parsing the SSL configuration file, or while attempting to locate the configuration file. This exception could indicate a problem with the IceSSL.Client.Config, IceSSL.Server.Config, IceSSL.Client.CertPath or IceSSL.Server.CertPath properties for your ::Ice::Communicator.

B.126 IceSSL: : ConfigurationLoadingException

Overview

Local exception ConfigurationLoadingException extends SslException

This exception indicates that an attempt was made to load the configuration for a Context, but the property specifying the indicated Context's SSL configuration file was not set. Check the values for the appropriate property, either IceSSL.Client.Config or IceSSL.Server.Config.

B.127 IceSSL: : ContextException

Overview

Local exception ContextException extends SslException

A problem was encountered while setting up the Context. This can include problems related to loading certificates and keys or calling methods on a Context that has not been initialized as of yet.

Derived Exceptions

CertificateKeyMatchException, CertificateLoadException, ContextInitializationException, ContextNotConfiguredException, PrivateKeyLoadException, TrustedCertificateAddException, UnsupportedContextException.

B.128 IceSSL: : ContextInitializationException

Overview

Local exception ContextInitializationException extends ContextException

Indicates that a problem occurred while initializing the context structure of the underlying SSL implementation.

B.129 IceSSL: : ContextNotConfiguredException

Overview

Local exception ContextNotConfiguredException extends ContextException

This exception is raised when an attempt is made to make use of a Context that has not been configured yet.

B.130 IceSSL: : ContextType

Overview

enum ContextType

A Plugin may serve as a Client, Server or both (ClientServer). A Context is set up inside the Plugin in order to handle either Client or Server roles. The Context represents a role-specific configuration. Some Plugin operations require a ContextType argument to identify the Context.

Used By

CertificateVerifier: : setContext, Plugin: : addTrustedCertificate,
 Plugin: : addTrustedCertificateBase64, Plugin: : configure,
 Plugin: : loadConfig, Plugin: : setCertificateVerifier, Plugin: : setR-
 SAKeys, Plugin: : setRSAKeysBase64.

Client

Client

Select only the Client Context, no modifications to the Server.

Server

Server

Select only the Server Context, no modifications to the Client.

ClientServer

ClientServer

Select and affect changes on both the Client and Server Contexts.

B.131 IceSSL: : Plugin

Overview

Local interface Plugin extends ::Ice::Plugin

The interface for the SSL plug-in. This interface is typically used to perform programmatic configuration of the plug-in.

addTrustedCertificate

```
void addTrustedCertificate(ContextType cType, ::Ice::ByteSeq
certificate);
```

Add a trusted certificate to the plug-in's default certificate store. The provided certificate (passed in binary DER format) is added to the trust list so that the certificate, and all certificates signed by its private key, are trusted. This method only affects new connections -- existing connections are left unchanged.

Parameters

contextType

The Context(s) in which to add the trusted certificate.

certificate

The certificate, in binary DER format, to be trusted.

addTrustedCertificateBase64

```
void addTrustedCertificateBase64(ContextType cType, string
certificate);
```

Add a trusted certificate to the plug-in's default certificate store. The provided certificate (passed in Base64-encoded binary DER format, as per the PEM format)

is added to the trust list so that the certificate, and all certificates signed by its private key, are trusted. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The Context(s) in which to add the trusted certificate.

`certi fi cate`

The certificate to be trusted, in Base64-encoded binary DER format.

`confi gure`

```
void confi gure(ContextType cType);
```

Configure the plug-in. If the plug-in is left in an unconfigured state, it will load its configuration from the properties `IceSSL.Server.Confi g` or `IceSSL.Client.Confi g`, depending on the context type. Configuration property settings will also be loaded during this operation, with the property values overriding those of the configuration file.

Parameters

`contextType`

The Context(s) to configure.

`getDefaultCertVeri fi er`

```
Certi fi cateVeri fi er getDefaultCertVeri fi er();
```

Retrieves an instance of the `Certi fi cateVeri fi er` that is installed by default in all plug-in instances.

Return Value

CertificateVerifier

`getSi ngl eCertVeri fi er`

```
Certi fi cateVeri fi er getSi ngl eCertVeri fi er(: : Ice : : ByteSeq  
certi fi cate);
```

Returns an instance of a `CertificateVerifier` that only accepts a single certificate, that being the RSA certificate represented by the binary DER encoding contained in the provided byte sequence. This is useful if you wish your application to accept connections from one party.

Be sure to use the `peer verifymode` in your SSL configuration file.

Parameters

`certificate`

A DER encoded RSA certificate.

Return Value

`CertificateVerifier`

loadConfig

```
void loadConfig(ContextType cType, string configFile, string certPath);
```

Configure the plug-in for the given Context using the settings in the given configuration file. If the plug-in is left in an unconfigured state, it will load its configuration from the property `IceSSL.Server.Config` or `IceSSL.Client.Config`, depending on the context type. Configuration property settings will also be loaded as part of this operation, with the property values overriding those of the configuration file.

Parameters

`contextType`

The Context to configure.

`configFile`

The file containing the SSL configuration information.

`certPath`

The path where certificates referenced in `loadConfig` may be found.

setCertificateVerifier

```
void setCertificateVerifier(ContextType cType, CertificateVerifier certVerifier);
```


Set the CertificateVerifier used for the indicated ContextType role. All plug-in Contexts are created with default CertificateVerifier objects installed. Replacement CertificateVerifiers can be specified using this operation. This operation only affects new connections -- existing connections are left unchanged.

Parameters

contextType

The Context(s) in which to install the Certificate Verifier.

certVerifier

The CertificateVerifier to install.

See Also

CertificateVerifier.

setRSAKeys

```
void setRSAKeys(ContextType cType, ::Ice::ByteSeq privateKey,
::Ice::ByteSeq publicKey);
```

Set the RSA keys to be used by the plug-in when operating in the context mode specified by ContextType. This method only affects new connections -- existing connections are left unchanged.

Parameters

contextType

The Context(s) in which to set/replace the RSA keys.

privateKey

The RSA private key, in binary DER format.

publicKey

The RSA public key, in binary DER format.

setRSAKeysBase64

```
void setRSAKeysBase64(ContextType cType, string privateKey, string
publicKey);
```

Set the RSA keys to be used by the plug-in when operating in the context mode specified by `ContextType`. This method only affects new connections -- existing connections are left unchanged.

Parameters

`contextType`

The Context(s) in which to set/replace the RSA keys.

`privateKey`

The RSA private key, in Base64-encoded binary DER format.

`publicKey`

The RSA public key, in Base64-encoded binary DER format.

B.132 IceSSL: : PrivateKeyException

Overview

Local exception `PrivateKeyException` extends `SslException`

A root exception class for all exceptions related to private keys.

Derived Exceptions

`PrivateKeyParseException`.

B.133 IceSSL: : PrivateKeyLoadException

Overview

Local exception `PrivateKeyLoadException` extends `ContextException`

Indicates that a problem occurred while loading a private key into a Context from either a memory buffer or from a file.

B.134 IceSSL: : PrivateKeyParseException

Overview

Local exception PrivateKeyParseException extends PrivateKeyException

Indicates that IceSSL was unable to parse the provided private key into a form usable by the underlying SSL implementation.

B.135 IceSSL: : ProtocolException

Overview

Local exception ProtocolException extends ShutdownException

Indicates that a problem occurred that violates the SSL protocol, causing the connection to be shutdown.

B.136 IceSSL: : ShutdownException

Overview

Local exception ShutdownException extends SslException

This exception generally indicates that a problem occurred that caused the shutdown of an SSL connection.

Derived Exceptions

CertificateVerificationException, ProtocolException.

B.137 IceSSL: : Ssl Excepti on

Overview

Local excepti on Ssl Excepti on

This exception represents the base of all security related exceptions in Ice. It is a local exception because, usually, a problem with security precludes a proper secure connection over which to transmit exceptions. In addition, many exceptions would contain information that is of no use to external clients/servers.

Derived Exceptions

Certi fi cateExcepti on, Certi fi cateVeri fi erTypeExcepti on, Confi gParse-Excepti on, Confi gurati onLoadi ngExcepti on, ContextExcepti on, Pri vateKeyExcepti on, ShutdownExcepti on.

message

string message;

Contains pertinent information from the security system to help explain the nature of the exception in greater detail. In some instances, it contains information from the underlying security implementation and/or debugging trace.

B.138 IceSSL: : TrustedCerti fi cateAddExcepti on

Overview

Local excepti on TrustedCerti fi cateAddExcepti on extends ContextExcepti on

An attempt to add a certificate to the Context's trusted certificate store has failed.

B.139 IceSSL: : UnsupportedOperationException

Overview

Local exception UnsupportedOperationException extends ContextException

An attempt was made to call a method that references a ContextType that is not supported for that operation.

B.140 Glacier

Overview

module Glacier

Glacier is a firewall solution for Ice. Glacier authenticates and filters client requests and allows callbacks to the client in a secure fashion. In combination with IceSSL, Glacier provides a security solution that is both non-intrusive and easy to configure.

B.141 Glacier: : CannotStartRouterException

Overview

exception CannotStartRouterException

This exception is raised if the router cannot be started.

reason

string reason;

Details as to why the router could not be started.

B.142 Glacier: : NoSessionManagerException

Overview

exception NoSessionManagerException

This exception is raised if no SessionManager object has been configured.

B.143 Glacier: : PermissionDeniedException

Overview

exception PermissionDeniedException

This exception is raised if router access is denied.

reason

string reason;

Details as to why access was denied.

B.144 Glacier: : PermissionsVerifier

Overview

interface PermissionsVerifier

The Glacier router starter permissions verifier.

checkPermissions

bool checkPermissions(string userId, string password, out string reason);

Check whether user has permission to access router.

Parameters

`userId`

The user id for which to check permissions.

`password`

The user's password.

`reason`

The reason access was denied.

Return Value

true if access is allowed, or false otherwise.

B.145 `Glacier::Router`

Overview

`interface Router extends ::Ice::Router`

The Glacier router interface.

createSession

`Session* createSession() throws NoSessionManagerException;`

Create a new session. The session is automatically shutdown when the Router terminates.

Return Value

A proxy to the new session.

Exceptions

`NoSessionManagerException`

if there is no configured `SessionManager`.

shutdown

`void shutdown();`

Shutdown the router.

B.146 `Glacier::Session`

Overview

`interface Session`

A session object, which is tied to the lifecycle of a Router.

See Also

Router, SessionManager.

`destroy`

`void destroy();`

Destroy the session. This is called automatically when the Router is destroyed.

B.147 `Glacier::SessionManager`

Overview

`interface SessionManager`

The session manager, which is responsible for managing Session objects. New session objects are created by the Router object.

See Also

Session.

`create`

`Session* create(string userId);`

Create a new session object.

Parameters`userId`

The user id for the session.

Return Value

A proxy to the newly created session.

B.148 `Glacier::Starter`

Overview`interface Starter`

The Glacier router starter.

`startRouter`

```
Router* startRouter(string userId, string password, out
::Ice::ByteSeq privateKey, out ::Ice::ByteSeq publicKey, out
::Ice::ByteSeq routerCert) throws Permi ssi onDeni edExcepti on,
CannotStartRouterExcepti on;
```

Start a new Glacier router. If the password for the given user id is incorrect, or if the user isn't allowed access, an `Permi ssi onDeni edExcepti on` is raised. Otherwise a new router is started, and a proxy to that router is returned to the caller.

Parameters`userId`

The user id for which to check the password.

`password`

The password for the given user id.

`privateKey`

The RSA Private Key (DER encoded) for the client to use. (Only for SSL.)

`publ i cKey`

The RSA Public Key (DER encoded) for the client to use. (Only for SSL.)

routerCert

The trusted certificate of the router. (Only for SSL.)

Return Value

A proxy to the router that has been started.

Exceptions

Permi ssi onDeni edExcepti on

Raised if the password for the given user id is not correct or if the user isn't allowed access.

B.149 IceStorm

Overview

modul e IceStorm

A messaging service with support for federation. In contrast to most other messaging or event services, IceStorm supports typed events, meaning that broadcasting a message over a federation is as easy as invoking a method on an interface.

Li nkI nfoSeq

sequence<Li nkI nfo> Li nkI nfoSeq;

A sequence of Li nkI nfo objects.

Used By

Topi c: : getLi nkI nfoSeq.

QoS

di cti onary<stri ng, stri ng> QoS;

This dictionary represents Quality of service parameters.

Used By

Topic : subscribe.

See Also

Topic : subscribe.

TopicDict

dictionary<string, Topic*> TopicDict;

Mapping of topic name to topic proxy.

Used By

TopicManager : retrieveAll.

B.150 IceStorm: : LinkExists

Overview

exception LinkExists

This exception indicates that an attempt was made to create a link that already exists.

name

string name;

The name of the linked topic.

B.151 IceStorm: : LinkInfo

Overview

struct LinkInfo

Information on the topic links.

Used By

LinkInfoSeq.

cost

int cost;

The cost of traversing this link.

name

string name;

The name of the linked topic.

theTopic

Topic* theTopic;

The linked topic.

B.152 IceStorm: : NoSuchLink

Overview

exception NoSuchLink

This exception indicates that an attempt was made to remove a link that does not exist.

name

string name;

The name of the link that does not exist.

B.153 IceStorm: : NoSuchTopic

Overview

exception NoSuchTopic

This exception indicates that an attempt was made to retrieve a topic that does not exist.

name

string name;

The name of the topic that does not exist.

B.154 IceStorm: : Topic

Overview

interface Topic

Publishers publish information on a particular topic. A topic logically represents a type.

See Also

TopicManager.

destroy

void destroy();

Destroy the topic.

getLinkInfoSeq

LinkInfoSeq getLinkInfoSeq();

Retrieve information on the current links.

Return Value

A sequence of LinkInfo objects.

getName

```
string getName();
```

Get the name of this topic.

Return Value

The name of the topic.

See Also

TopicManager::create.

getPublisher

```
Object* getPublisher();
```

Get a proxy to a publisher object for this topic. To publish data to a topic, the publisher calls getPublisher and then casts to the topic type. An unchecked cast must be used on this proxy.

Return Value

A proxy to publish data on this topic.

Link

```
void Link(Topic* linkTo, int cost) throws LinkExists;
```

Create a link to the given topic. All events originating on this topic will also be sent to link.

Parameters

linkTo

The topic to link to.

cost

The cost to the linked topic.

Exceptions

`LinkExists`

Raised if a link to the same topic already exists.

`subscribe`

```
void subscribe(QoS theQoS, Object* subscriber);
```

Subscribe with the given QoS to this topic. If the given subscribe proxy has already been registered, it will be replaced.

Parameters

`qos`

The quality of service parameters for this subscription.

`subscriber`

The subscriber's proxy.

See Also

`unsubscribe`.

`unlink`

```
void unlink(Topic* linkTo) throws NoSuchLink;
```

Destroy the link from this topic to the given topic `unlink`.

Parameters

`link`

The topic to destroy the link to.

Exceptions

`NoSuchLink`

Raised if a link to the topic does not exist.

`unsubscribe`

```
void unsubscribe(Object* subscriber);
```

Unsubscribe the given `unsubscribe`.

Parameters

`subscriber`

The proxy of an existing subscriber.

See Also

`subscribe`.

B.155 `IceStorm: : TopicExists`

Overview

exception `TopicExists`

This exception indicates that an attempt was made to create a topic that already exists.

`name`

`string name;`

The name of the topic that already exists.

B.156 `IceStorm: : TopicManager`

Overview

interface `TopicManager`

A topic manager manages topics, and subscribers to topics.

See Also

`Topic`.

create

Topic* create(string name) throws TopicExists;

Create a new topic. The topic name must be unique, otherwise TopicExists is raised.

Parameters

name

The name of the topic.

Return Value

A proxy to the topic instance.

Exceptions

TopicExists

Raised if a topic with the same name already exists.

retrieve

Topic* retrieve(string name) throws NoSuchTopic;

Retrieve a topic by name.

Parameters

name

The name of the topic.

Return Value

A proxy to the topic instance.

Exceptions

NoSuchTopic

Raised if the topic does not exist.

retrieveAll

TopicDict retrieveAll();

Retrieve all topics managed by this topic manager.

Return Value

A dictionary of string, topic proxy pairs.

B.157 IcePatch

Overview

module IcePatch

A patching service for software distributions. IcePatch automates updating of individual files as well as complete directory hierarchies. Only files that have changed are downloaded to the client machine, using efficient compression algorithms.

FileDescSeq

sequence<FileDesc> FileDescSeq;

Used By

Directory::getContents.

B.158 IcePatch::BusyException

Overview

exception BusyException

B.159 IcePatch: : Directory

Overview

interface Directory extends File

getContents

FileDescSeq getContents() throws FileAccessException,
BusyException;

B.160 IcePatch: : DirectoryDesc

Overview

class DirectoryDesc extends FileDesc

dir

Directory* dir;

B.161 IcePatch: : File

Overview

interface File

Derived Classes and Interfaces

Directory, Regular.

describe

FileDesc describe() throws FileAccessException, BusyException;

B.162 IcePatch::FileAccessException

Overview

exception FileAccessException

reason

string reason;

B.163 IcePatch::FileDesc

Overview

class FileDesc

Derived Classes and Interfaces

DirectoryDesc, RegularDesc.

Used By

File::describe, FileDescSeq.

md5

::Ice::ByteSeq md5;

B.164 IcePatch::Regular

Overview

interface Regular extends File

getBZ2

::Ice::ByteSeq getBZ2(int pos, int num) throws
FileAccessException, BusyException;

getBZ2MD5

::Ice::ByteSeq getBZ2MD5(int size) throws FileAccessException,
BusyException;

getBZ2Size

int getBZ2Size() throws FileAccessException, BusyException;

B.165 IcePatch::RegularDesc

Overview

class RegularDesc extends FileDesc

reg

Regular* reg;

Appendix C

Properties

If not stated otherwise in the description of the individual properties, the default value for all properties is the empty string. If the property takes a numeric value, the empty string is interpreted as zero.

C.1 Ice Configuration Property

Ice.Config

Synopsis

```
--Ice.Config --Ice.Config=1 --Ice.Config=config_file
```

Description

This property must be set from the command line with the `--Ice.Config`, `--Ice.Config=1`, or `--Ice.Config=config_file` option.

If the `Ice.Config` property is empty or set to 1, the Ice run time examines the contents of the `ICE_CONFIG` environment variable to retrieve the pathname of a configuration file. Otherwise, `Ice.Config` must be set to the pathname of a configuration file. (Pathnames can be relative or absolute.) Further property values are read from the configuration file thus specified.

Configuration File Syntax

A configuration file contains a number of property settings, one setting per line. Property settings have one of the forms

property_name= # Set property to the empty string or zero

property_name=*value* # Assign value to property

The # character indicates a comment: the # character and anything following the # character on the same line are ignored. A line that has the # character as its first non-white space character is ignored in its entirety.

A configuration file is free-form: blank, tab, and newline characters serve as token delimiters and are otherwise ignored.

Any setting of the `Ice.Config` property inside the configuration file itself is ignored.

C.2 Ice Trace Properties

Ice.Trace.GC

Synopsis

`Ice.Trace.GC=num`

Description

The garbage collector trace level:

0	No garbage collector trace. (default)
1	Show the total number of instances collected, the total number of instances examined, the time spent in the collector in milliseconds, and the total number of runs of the collector.
2	Like 1, but also produces a trace message for each run of the collector.

Ice.Trace.Network

Synopsis

Ice.Trace.Network=*num*

Description

The network trace level:

0	No network trace. (default)
1	Trace connection establishment and closure.
2	Like 1, but more detailed.
3	Like 2, but also trace data transfer.

Ice.Trace.Protocol

Synopsis

Ice.Trace.Protocol=*num*

Description

The protocol trace level:

0	No protocol trace. (default)
1	Trace Ice protocol messages.

Ice.Trace.Retry

Synopsis

Ice.Trace.Retry=*num*

Description

The request retry trace level:

0	No request retry trace. (default)
1	Trace Ice operation call retries.
2	Also trace Ice endpoint usage.

Ice.Trace.Slicing**Synopsis**

`Ice.Trace.Slicing=num`

Description

The slicing trace level:

0	No trace of slicing activity. (default)
1	Trace all exception and class types that are unknown to the receiver and therefore sliced.

C.3 Ice Warning Properties

Ice.Warn.Connections**Synopsis**

`Ice.Warn.Connections=num`

Description

If *num* is set to a value larger than zero, Ice applications will print warning messages for certain exceptional conditions in connections. The default value is 0.

Ice.Warn.Datagrams

Synopsis

Ice.Warn.Datagrams=*num*

Description

If *num* is set to a value larger than zero, servers will print a warning message if they receive a datagram that exceeds the servers' receive buffer size. (Note that this condition is not detected by all UDP implementations -- some implementations silently drop received datagrams that are too large.) The default value is 0.

Ice.Warn.Dispatch

Synopsis

Ice.Warn.Di spatch=*num*

Description

If *num* is set to a value larger than zero, Ice applications will print warning messages for certain exceptions that are raised while an incoming request is dispatched.

0	No warnings.
1	Print warnings for unexpected Ice: : Local Excepti on, Ice: : UserExcepti on, C++ exceptions, and Java runtime exceptions. (default)
2	Like 1, but also issue warnings for Ice: : Obj ectNotExi stExcepti on, Ice: : FacetNotExi stExcepti on, and Ice: : Operati onNotExi stExcepti on.

Ice.Warn.AMICallback

Synopsis

Ice.Warn. AMI Cal l back=*num*

Description

If *num* is set to a value larger than zero, warnings are printed if an AMI callback raises an exception. The default value is 1.

Ice.Warn.Leaks**Synopsis**

Ice.Warn.Leaks=num

Description

If *num* is set to a value larger than zero, the `Ice::Communicator` destructor will print a warning if some other Ice-related C++ objects are still in memory. The default value is 1. (C++ only.)

C.4 Ice Object Adapter Properties

name.AdapterId**Synopsis**

name.AdapterId=id

Description

Specifies an identifier for the object adapter with the name *name*. This identifier must be unique among all object adapters using the same locator instance. If a locator proxy is defined using *name.Locator* or `Ice.Default.Locator`, this object adapter will set its endpoints with the locator registry upon activation.

name.Endpoints**Synopsis**

name.Endpoints=endpoints

Description

Sets the endpoints for the object adapter *name* to *endpoints*.

***name*.Locator**

Synopsis

name.Locator=*locator*

Description

Specifies a locator for the object adapter with the name *name*. The value is a stringified proxy to the Ice locator interface.

***name*.RegisterProcess**

Synopsis

name.RegisterProcess=*num*

Description

If *num* is set to a value larger than zero, the object adapter with the name *name* registers the server with the locator registry. Registration occurs upon the object adapter's initial activation, during which the object adapter creates a servant implementing the Ice: : Process interface, adds the servant using a UUID, and registers its proxy with the locator registry using the value of the Ice. ServerId property for the server id. The servant implements the shutdown operation by invoking shutdown on the object adapter's communicator.

It is important for a server to be registered with the locator registry so that services such as IcePack can request a graceful shutdown when necessary. If a server is not registered, then platform-specific techniques are used to request a shutdown, and these techniques are not always effective (especially on Windows platforms).

Only one object adapter in a server should register with a locator registry.

The Ice: : Process servant represents a potential target for denial-of-service attacks. The object adapter uses a UUID to make the proxy more difficult to guess, but the object adapter should be configured with secure endpoints if the server operates in a potentially hostile environment. Alternatively, a dedicated object adapter can be created specifically to provide a restricted access point for services such as IcePack.

name.Router

Synopsis

name.Router=router

Description

Specifies a router for the object adapter with the name *name*. The value is a stringified proxy to the Glacier router control interface. Defining a router allows the object adapter to receive callbacks from the router over outgoing connections from this process to the router, thereby avoiding the need for the router to establish a connection back to the object adapter.

A router can only be assigned to one object adapter. Specifying the same router for more than one object adapter will result in undefined behavior. The default value is no router.

Ice.PrintAdapterReady

Synopsis

Ice.PrintAdapterReady=num

Description

If *num* is set to a value larger than zero, an object adapter prints "*adapter_name* ready" on standard output after initialization is complete. This is useful for scripts that need to wait until an object adapter is ready to be used.

C.5 Ice Plug-in Properties

Ice.Plugin.*name*

Synopsis

Ice.Plugin.name=entry_point [args]

Description

Defines a plug-in to be installed during communicator initialization.

In C++, *entry_point* has the form `basename[, version]: function`. The `basename` and optional `version` components are used to construct the name of a DLL or shared library. If no version is supplied, the Ice version is used. The `function` component is the name of a function with C linkage.

For example, the entry point `MyPlugin.2.3:create` would imply a shared library name of `libMyPlugin.so.2.3` on Unix and `MyPlugin23.dll` on Windows. Furthermore, if Ice is built on Windows with debugging, `ad` will be automatically appended to the version (e.g., `MyPlugin23d.dll`).

In Java, *entry_point* is the name of a class.

C.6 Ice Thread Pool Properties

name.Size

Synopsis

name.Size=num

Description

name is the name of the thread pool. The name of the client-side thread pool is `Ice.ThreadPool.Client`, the name of the default server-side thread pool is `Ice.ThreadPool.Server`. In addition, individual object adapters can have separate thread pools. In this case, the name of the thread pool is *adapter_name*.`.ThreadPool`. Having a separate thread pool for an object adapter is useful to ensure that a minimum number of threads is available for dispatching requests on certain Ice objects, in order to avoid deadlocks because of thread starvation.

num is the initial and also minimum number of threads in the thread pool. The default is one for `Ice.ThreadPool.Client` and `Ice.ThreadPool.Server`, and zero for object adapter thread pools, meaning that by default, object adapters use the default server-side thread pool.

Multiple threads for the client side thread pool are only required for nested AMI invocations. If AMI is not used, or AMI calls are not nested (i.e., AMI callbacks do not call any other methods on Ice objects), then there is no need to set the number of threads in the client thread pool to a value larger than one.

name*.SizeMax*Synopsis**

name. Si zeMax=*num*

Description

num is the maximum number of threads for the thread pool *name*. Thread pools in Ice can grow and shrink dynamically, based on an average load factor. Thread pools do not grow larger than the parameter specified by Si zeMax, and they do not shrink to a number of threads smaller than the value specified by Si ze.

The default value for Si zeMax is the value of Si ze, meaning that by default, thread pools do not grow dynamically.

name*.SizeWarn*Synopsis**

name. Si zeWarn=*num*

Description

Whenever *num* threads are active in a thread pool, a "low on threads" warning is printed. The default value for Si zeWarn is 80% of the value specified by Si zeMax.

C.7 Ice Default and Override Properties

Ice.Default.Protocol**Synopsis**

Ice. Default t. Protocol =*protocol*

Description

Sets the protocol that is being used if an endpoint uses default t as the protocol specification. The default value is tcp.

Ice.Default.Host

Synopsis

Ice.Default.Host=*host*

Description

If an endpoint is specified without a host name (i.e., without a `-h host` option), the *host* value from this property is used instead. The default value is the local host name.

Ice.Default.Router

Synopsis

Ice.Default.Router=*router*

Description

Specifies the default router for all proxies. The value is a stringified proxy to the Glacier router control interface. The default router can be overridden on a proxy using the `ice_router()` operation. The default value is no router.

Ice.Default.Locator

Synopsis

Ice.Default.Locator=*locator*

Description

Specifies a default locator for all proxies and object adapters. The value is a stringified proxy to the IcePack locator interface. The default locator can be overridden on a proxy using the `ice_locator()` operation. The default value is no locator.

The IcePack locator's object identity is `IcePack/Locator`. It is listening on the IcePack client endpoints. For example if `IcePack.Registry.Client.Endpoints` is set to `tcp -p 12000 -h local host`, the stringified proxy for the IcePack locator will be `IcePack/Locator: tcp -p 12000 -h local host`.

Ice.Override.Timeout

Synopsis

Ice.Override.Timeout=*num*

Description

If set, this property overrides timeout settings in all endpoints. *num* is the timeout value in milliseconds, or -1 for no timeout.

Ice.Override.ConnectTimeout

Synopsis

Ice.Override.ConnectTimeout=*num*

Description

This property overrides timeout settings used to establish connections. *num* is the timeout value in milliseconds, or -1 for no timeout. If this property is not set, then Ice.Override.Timeout is used.

Ice.Override.Compress

Synopsis

Ice.Override.Compress=*num*

Description

If set, this property overrides compression settings in all proxies. If *num* is set to a value larger than zero, compression is enabled. If zero, compression is disabled.

C.8 Ice Miscellaneous Properties

Ice.GC.Interval

Synopsis

Ice.GC.Interval=*num*

Description

This property determines the frequency with which the class garbage collector runs. If the interval is set to zero (the default), no collector thread is created. Otherwise, the collector thread runs every *num* seconds.

Ice.RetryIntervals

Synopsis

`Ice.RetryIntervals=num [num ...]`

Description

This property defines the number of times an operation is re-tried and the delay between each retry. For example, if the property is set to `0 100 500`, the operation will be re-tried 3 times: immediately re-tried upon the first failure, after waiting 100 (ms) upon the second failure, after waiting 500 (ms) upon the third failure. The default value is to retry once immediately (0). If set to -1, no retry will occur.

Ice.MessageSizeMax

Synopsis

`Ice.MessageSizeMax=num`

Description

This property controls the maximum size (in kilobytes) of a protocol message that will be accepted or sent by the Ice run time. The size includes the size of the Ice protocol header. Messages larger than this size cause a `[MemoryLimitException]`. The default size is 1024 (1 Megabyte). Settings with a value less than 1 are ignored.

This property adjusts the value of `Ice.UDP.RcvSize` and `Ice.UDP.SndSize`, that is, if `Ice.UDP.RcvSize` or `Ice.UDP.SndSize` are larger than `Ice.MessageSizeMax * 1024 + 28`, they are adjusted to `Ice.MessageSizeMax * 1024 + 28`.

Ice.ChangeUser

Synopsis

`Ice.ChangeUser=user`

Description

If set, Ice will change the user and group id to the respective ids of *user* in `/etc/passwd`. This only works if the Ice application is executed by the super-user. (Unix only.)

Ice.ConnectionIdleTime

Synopsis

`Ice.Connecti onl dl eTi me=num`

Description

If *num* is set to a value larger than zero, ACM (Active Connection Management) is enabled. This means that connections are automatically closed after they have been idle for *num* seconds. This is transparent to user code, i.e., closed connections are automatically reestablished in case they are needed again. The default value is 60, meaning that idle connections are automatically closed after one minute.

Ice.MonitorConnections

Synopsis

`Ice.Moni torConnecti ons=num`

Description

If *num* is set to a value larger than zero, Ice will start a thread that monitors connections. This thread is responsible for shutting down idle connections (see `Ice.Connecti onl dl eTi me`), as well as for enforcing AMI timeouts. The default value is the value of `Ice.Connecti onl dl eTi me`.

Ice.PrintProcessId

Synopsis

Ice.PrintProcessId=*num*

Description

If *num* is set to a value larger than zero, the process ID is printed on standard output upon startup.

Ice.ProgramName

Synopsis

Ice.ProgramName=*name*

Description

name is the program name, which is set automatically from argv[0] during initialization. However, a different name can be used by setting this property.

Ice.ServerId

Synopsis

Ice.ServerId=*id*

Description

The value *id* is used as the server id when an object adapter registers the server with the locator registry. Refer to the description of the object adapter property *adapter.RegisterProcess* for more information.

Ice.ServerIdleTime

Synopsis

Ice.ServerIdleTime=*num*

Description

If *num* is set to a value larger than zero, Ice will automatically call Communicator::shutdown after the Communicator has been idle for *num* seconds. This will shut down the Communicator's server side, and all threads waiting in Communicator::waitForShutdown will return. After that, a server will typically do some cleanup work, and then exit. The default value is zero, meaning that the server will not shut down automatically.

Ice.UseEventLog**Synopsis**

Ice.UseEventLog=*num*

Description

If *num* is set to a value larger than zero, a special logger is installed that logs to the Windows Event Log instead of standard error. The event source name is the value of Ice.ProgramName. (Windows 2000/XP only.)

Ice.UseSyslog**Synopsis**

Ice.UseSyslog=*num*

Description

If *num* is set to a value larger than zero, a special logger is installed that logs to the syslog facility instead of standard error. The identifier for syslog is the value of Ice.ProgramName. (Unix only.)

Ice.Logger.Timestamp**Synopsis**

Ice.Logger.Timestamp=*num*

Description

If *num* is set to a value larger than zero, the output of the default logger will include timestamps.

Ice.NullHandleAbort**Synopsis**

`Ice.NullHandleAbort=num`

Description

If *num* is set to a value larger than zero, invoking an operation using a null smart pointer (i.e., a handle) causes the program to abort immediately instead of raising `IceUtil::NullHandleException`. (C++ only.)

Ice.Nohup**Synopsis**

`Ice.Nohup=num`

Description

If *num* is set to a value larger than zero, the C++ `Ice::Application` class ignores `SIGHUP` (for UNIX) and `CTRL_LOGOFF_EVENT` (for Windows). As a result, an application that sets `Ice.Nohup` continues to run if the user that started the application logs off. (C++ only.)

Ice.UDP.RcvSize, Ice.UDP.SndSize**Synopsis**

`Ice.UDP.RcvSize=num Ice.UDP.SndSize=num`

Description

These properties set the UDP receive and send buffer sizes to the specified value in bytes. Ice messages larger than *num* - 28 bytes cause a `DatagramLimitException`. The default value depends on the configuration of the local UDP stack. (Common default values are 65535 and 8192 bytes.)

The OS may impose lower and upper limits on the receive and send buffer sizes or otherwise adjust the buffer sizes. If a limit is requested that is lower than the OS-imposed minimum, the value is silently adjusted to the OS-imposed minimum. If a limit is requested that is larger than the OS-imposed maximum, the value is adjusted to the OS-imposed maximum; in addition, Ice logs a warning showing the requested size and the adjusted size.

Settings of these properties less than 28 are ignored.

Note that, on many operating systems, it is possible to set buffer sizes greater than 65535. Such settings do not change the hard limit of 65507 bytes for the payload of a UDP packet, but merely affect how much data can be buffered by the kernel.

Settings less than 65535 limit the size of Ice datagrams as well as adjust the kernel buffer sizes.

If `Ice.MessageSizeMax` is set and `Ice.MessageSizeMax * 1024 + 28` is smaller than the UDP receive or send buffer size, the corresponding UDP buffer size is reduced to `Ice.MessageSizeMax * 1024 + 28`.

C.9 IceSSL Properties

IceSSL.Trace.Security

Synopsis

`IceSSL.Trace.Security=num`

Description

The SSL plug-in trace level:

0	No security trace. (default)
1	Trace security warnings.
2	Like 1, but more verbose, including warnings during configuration file parsing.

IceSSL.Client.CertPath, IceSSL.Server.CertPath

Synopsis

`IceSSL.Client.CertPath=`*path* `IceSSL.Server.CertPath=`*path*

Description

Defines the path (relative or absolute) where the SSL plug-in can find PEM format certificate files (RSA and DSA) and Diffie-Hellman group parameter files.

In the case that the `IceSSL.Client.Config` or `IceSSL.Server.Config` is a relative path, it will be relative to the value of `IceSSL.Client.CertPath` or `IceSSL.Server.CertPath`.

If not specified, the application will use the current working directory as the certificate path.

IceSSL.Client.Config, IceSSL.Server.Config

Synopsis

`IceSSL.Client.Config=`*config_file* `IceSSL.Server.Config=`*config_file*

Description

Defines the XML-based configuration file from which the SSL plug-in will load initialization information and certificates. If the property contains a relative path, the path will be interpreted relative to the certificate path defined by `IceSSL.Client.CertPath` or `IceSSL.Server.CertPath`.

Xerces-c, the XML parser used to read this file, will look for the DTD file in the same directory that it finds the XML configuration file.

Depending on whether the application is running in client mode, server mode or both modes, a valid value for one or both of these parameters must be specified for the proper operation of the IceSSL plug-in.

IceSSL.Client.Handshake.Retries

Synopsis

`IceSSL.Client.Handshake.Retries=`*num*

Description

IceSSL clients attempt to perform an entire SSL handshake in the connection phase. When attempting this handshake, it is possible that the client will timeout waiting for a response from the server. This property specifies the number of handshake retries the client attempts before throwing a `Ice::Connecti on Fai l e- dExcepti on`.

If not specified, the default value for this property is 10 retries.

**IceSSL.Client.Passphrase.Retries,
IceSSL.Server.Passphrase.Retries**
Synopsis

`IceSSL. Cl i ent. Passphrase. Retri es=num`
`IceSSL. Server. Passphrase. Retri es=num`

Description

When IceSSL is directed to use a private key in a PEM file that has been encrypted, a prompt is displayed `Enter PEM pass phrase: .` If the passphrase is entered incorrectly, these properties determine how many retries the user will be allowed before IceSSL shuts down.

If not specified, the default value for these properties is 5 retries.

**IceSSL.Server.Overrides.RSA.PrivateKey,
IceSSL.Server.Overrides.RSA.Certificate**
Synopsis

`IceSSL. Server. Overri des. RSA. Pri vateKey=Base64 encoded DER string`
`IceSSL. Server. Overri des. RSA. Certi fi cate=Base64 encoded DER string`

Description

These properties override the RSA private key and public key (certificate) specified in the config file (`IceSSL. Server. Confi g`) for the Server context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

IceSSL.Server.Overrides.DSA.PrivateKey,

IceSSL.Server.Overrides.DSA.Certificate

Synopsis

`IceSSL.Server.Overrides.DSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Server.Overrides.DSA.Certificate=Base64 encoded DER string`

Description

These properties override the DSA private key and public key (certificate) specified in the config file (`IceSSL.Server.Config`) for the Server context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

IceSSL.Client.Overrides.RSA.PrivateKey, IceSSL.Client.Overrides.RSA.Certificate

Synopsis

`IceSSL.Client.Overrides.RSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Client.Overrides.RSA.Certificate=Base64 encoded DER string`

Description

These properties provides a method by which the RSA private key and public key (certificate) used by the Client context may be overridden from those specified in the config file (specified in `IceSSL.Client.Config`). The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

IceSSL.Client.Overrides.DSA.PrivateKey, IceSSL.Client.Overrides.DSA.Certificate

Synopsis

`IceSSL.Client.Overrides.DSA.PrivateKey=Base64 encoded DER string`
`IceSSL.Client.Overrides.DSA.Certificate=Base64 encoded DER string`

Description

These properties override the RSA private key and public key (certificate) specified in the config file (`IceSSL.Client.Config`) for the Client context. The value must be the DER representation of the private and public keys, base64 encoded.

There are no default values for these properties.

IceSSL.Client.Overrides.CACertificate, IceSSL.Server.Overrides.CACertificate

Synopsis

`IceSSL.Client.Overrides.CACertificate=Base64 encoded DER string`
`IceSSL.Server.Overrides.CACertificate=Base64 encoded DER string`

Description

These properties override any trusted Certificate Authority (CA) certificates specified in `IceSSL.Server.Config` or `IceSSL.Client.Config`. The new certificate is represented as the base64 encoding of the DER binary representation of the certificate.

There are no default values for these properties.

IceSSL.Client.IgnoreValidPeriod, IceSSL.Server.IgnoreValidPeriod

Synopsis

`IceSSL.Client.IgnoreValidPeriod=num`
`IceSSL.Server.IgnoreValidPeriod=num`

Description

These properties will cause the default certificate verifier to ignore the certificate validity period on peer certificates if set to 1. The default value for these properties is 0, meaning that the certificate validity period is not ignored.

C.10 IceBox Properties

IceBox.ServiceManager.Endpoints

Synopsis

IceBox.ServiceManager.Endpoints=*endpoints*

Description

Defines the endpoints of the IceBox service manager interface. The service manager endpoints must be accessible to the IceBox administration tool to shut-down the IceBox server.

IceBox.ServiceManager.Identity

Synopsis

IceBox.ServiceManager.Identity=*identity*

Description

The identity of the service manager interface. If not specified the default value ServiceManager is used.

IceBox.PrintServicesReady

Synopsis

IceBox.PrintServicesReady=*token*

Description

The service manager will print "*token* ready" on standard output after initialization of all the services is done. This is useful for scripts that wish to wait until all services are ready to be used.

IceBox.LoadOrder

Synopsis

IceBox. LoadOrder=*names*

Description

Determines the order in which services are loaded. The service manager loads the services in the order they appear in *names*, where each service name is separated by a comma or whitespace. Any services not mentioned in *names* are loaded afterward, in an undefined order.

IceBox.Service.*name*

Synopsis

IceBox. Servi ce. *name*=*entry_point* [*args*]

Description

Defines a service to be loaded during IceBox initialization.

In C++, *entry_point* has the form `library: symbol`. The `library` component is the name of a shared library or DLL. The `symbol` component is the name of a factory function used to create the service.

In Java, *entry_point* is the name of the service implementation class.

IceBox.DBEnvName.*name*

Synopsis

IceBox. DBEnvName. *name*=*db*

Description

Defines the database environment directory for the Freeze service with the name *name*.

IceBox.UseSharedCommunicator.*name*

Synopsis

IceBox.UseSharedCommunicator.*name=num*

Description

If *num* is set to a value larger than zero, the service manager will supply the service with the name *name* a communicator which might be shared by other services.

C.11 IcePack Properties

IcePack.Registry.Client.Endpoints

Synopsis

IcePack.Registry.Client.Endpoints=*endpoints*

Description

Defines the endpoints of the IcePack client interface. The client endpoints must be accessible to Ice clients that are using IcePack to locate objects (see Ice.DefaultLocator).

IcePack.Registry.Server.Endpoints

Synopsis

IcePack.Registry.Server.Endpoints=*endpoints*

Description

Defines the endpoints of the IcePack server interface. The server endpoints must be accessible to Ice servers that are using IcePack to register their object adapter endpoints.

IcePack.Registry.Admin.Endpoints

Synopsis

`IcePack.Registry.Admin.Endpoints=endpoints`

Description

Defines the optional administrative endpoints of the IcePack admin interface. The administrative endpoints must be accessible to clients which are using the IcePack administrative interface, such as the IcePack administrative tool.

Allowing access to the IcePack admin interface is a security risk! If this property is not defined, the admin interface will be disabled.

IcePack.Registry.Internal.Endpoints

Synopsis

`IcePack.Registry.Internal.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack internal interface. The internal endpoints must be accessible to IcePack nodes. Nodes use this interface to communicate with the registry.

IcePack.Registry.Data

Synopsis

`IcePack.Registry.Data=path`

Description

Defines the path of the IcePack registry data directory.

IcePack.Registry.DynamicRegistration

Synopsis

`IcePack.Registry.DynamicRegistration=num`

Description

If *num* is set to a value larger than zero, the locator registry will allow Ice servers to set endpoints for object adapters which have not been previously registered.

IcePack.Registry.Trace.ServerRegistry**Synopsis**

IcePack.Registry.Trace.ServerRegistry=*num*

Description

The server registry trace level:

0	No server registry trace. (default)
1	Trace server registration, removal.

IcePack.Registry.Trace.AdapterRegistry**Synopsis**

IcePack.Registry.Trace.AdapterRegistry=*num*

Description

The object adapter registry trace level:

0	No object adapter registry trace. (default)
1	Trace object adapter registration, removal.

IcePack.Registry.Trace.NodeRegistry**Synopsis**

IcePack.Registry.Trace.NodeRegistry=*num*

Description

The node registry trace level:

0	No node registry trace. (default)
1	Trace node registration, removal.

IcePack.Registry.Trace.ObjectRegistry**Synopsis**

`IcePack.Registry.Trace.ObjectRegistry=num`

Description

The object registry trace level:

0	No object registry trace. (default)
1	Trace object registration, removal.

IcePack.Node.Endpoints**Synopsis**

`IcePack.Node.Endpoints=endpoints`

Description

Defines the endpoints of the IcePack node interface. The node endpoints must be accessible to the IcePack registry. The registry uses this interface to communicate with the node.

IcePack.Node.Name**Synopsis**

`IcePack.Node.Name=name`

Description

Defines the name of the IcePack node. All nodes using the same registry must have unique names. The node will refuse to start if there is a node with the same name already running.

The default value is the hostname as returned by `gethostname()`.

IcePack.Node.Data**Synopsis**

`IcePack. Node. Data=`*path*

Description

Defines the path of the IcePack node data directory. The node will create `db` and `servers` subdirectories in this directory if they do not already exist. The `db` directory contains the node database. The `servers` directory contains configuration data for each deployed server.

IcePack.Node.Output**Synopsis**

`IcePack. Node. Output=`*path*

Description

Defines the path of the IcePack node output directory. If set, the node will redirect the `stdout` and `stderr` output of the started servers to files named *server.out* and *server.err* in this directory. Otherwise, the started servers share the `stdout` and `stderr` of the IcePack node.

IcePack.Node.PropertiesOverride**Synopsis**

`IcePack. Node. Properti esOverri de=`*overri des*

Description

Defines a list of properties which override the properties defined in server deployment descriptors. For example, in some cases it is desirable to set the property `Ice.Default.Host` for servers, but not in server deployment descriptors. The property definitions should be separated by white space.

IcePack.Node.RedirectErrToOut**Synopsis**

`IcePack.Node.RedirectErrToOut=num`

Description

If *num* is set to a value larger than zero, the stderr of each started server is redirected to the server's stdout.

IcePack.Node.WaitTime**Synopsis**

`IcePack.Node.WaitTime=num`

Description

Defines the interval in seconds that IcePack will wait for server activation and deactivation.

If a server is automatically activated and does not register its object adapter endpoints within this time interval, the node will assume there is a problem with the server and return an empty set of endpoints to the client.

If a server is being gracefully deactivated and IcePack does not detect the server deactivation during this time interval, IcePack will kill the server. The default value is 60 seconds.

IcePack.Node.CollocateRegistry**Synopsis**

`IcePack.Node.CollocateRegistry=num`

Description

If *num* is set to a value larger than zero, the node will also collocate the IcePack registry.

The collocated registry is configured with the same properties as the standalone IcePack registry.

IcePack.Node.PrintServersReady

Synopsis

`IcePack.Node.PrintServersReady=token`

Description

The IcePack node will print "*token* ready" on standard output after all the servers managed by the node are ready. This is useful for scripts that wish to wait until all servers are ready to be used.

IcePack.Node.Trace.Server

Synopsis

`IcePack.Node.Trace.Server=num`

Description

The server trace level:

0	No server trace. (default)
1	Trace server addition, removal.
2	Like 1, but more verbose, including server activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including server transitional state change (activating and deactivating).

IcePack.Node.Trace.Adapter

Synopsis

IcePack. Node. Trace. Adapter=*num*

Description

The object adapter trace level:

0	No object adapter trace. (default)
1	Trace object adapter addition, removal.
2	Like 1, but more verbose, including object adapter activation and deactivation and more diagnostic messages.
3	Like 2, but more verbose, including object adapter transitional state change (e.g., `waiting for activation').

IcePack.Node.Trace.Activator

Synopsis

IcePack. Node. Trace. Activator=*num*

Description

The activator trace level:

0	No activator trace. (default)
1	Trace process activation, termination.
2	Like 1, but more verbose, including process signaling and more diagnostic messages on process activation.
3	Like 2, but more verbose, including more diagnostic messages on process activation (e.g., path, working directory and arguments of the activated process).

C.12 IceStorm Properties

IceStorm.TopicManager.Endpoints

Synopsis

IceStorm.TopicManager.Endpoints=*endpoints*

Description

Defines the endpoints for the IceStorm topic manager and topic objects.

IceStorm.Publish.Endpoints

Synopsis

IceStorm.Publish.Endpoints=*endpoints*

Description

Defines the endpoints for the IceStorm publisher objects.

IceStorm.Trace.TopicManager

Synopsis

IceStorm.Trace.TopicManager=*num*

Description

The topic manager trace level:

0	No topic manager trace. (default)
1	Trace topic creation.

IceStorm.Trace.Topic

Synopsis

IceStorm.Trace.Topic=*num*

Description

The topic trace level:

0	No topic trace. (default)
1	Trace topic links, subscription, and unsubscription.
2	Like 1, but more verbose, including QoS information, and other diagnostic information.

IceStorm.Trace.Flush**Synopsis**

IceStorm.Trace.Flush=*num*

Description

Trace information on the thread that flushes batch reliability events to subscribers:

0	No flush trace. (default)
1	Trace each flush.

IceStorm.Trace.Subscriber**Synopsis**

IceStorm.Trace.Subscriber=*num*

Description

The subscriber trace level:

0	No subscriber trace. (default)
1	Trace topic diagnostic information on subscription and unsubscription.

IceStorm.Flush.Timeout

Synopsis

IceStorm.Flush.Timeout=*num*

Description

Defines the interval in milliseconds that batch reliability events are sent to subscribers. The default is 1000 ms. The minimum value of this property is 100 ms.

IceStorm.TopicManager.Proxy

Synopsis

IceStorm.TopicManager.Proxy=*proxy*

Description

Defines the proxy for the IceStorm topic manager. This property is used by the IceStorm administration tool, and may also be used by applications.

C.13 Glacier Router Properties

Glacier.Router.Endpoints, Glacier.Router.Client.Endpoints, Glacier.Router.Server.Endpoints

Synopsis

Glacier.Router.Endpoints=*endpoints*
Glacier.Router.Client.Endpoints=*endpoints*
Glacier.Router.Server.Endpoints=*endpoints*

Description

Defines the endpoints of the Glacier router control interface, the client interface, and the server interface. The router endpoints and the client endpoints must be accessible to Glacier clients from which the router forwards requests, and to which the router sends callbacks. The server endpoints must be accessible to

Glacier servers to which the router forwards requests, and from which the router accepts callbacks for the client.

Glacier.Router.Identity

Synopsis

`Glacier.Router.Identity=identity`

Description

The identity of the router control interface. If not specified, the default value `router` is used.

Glacier.Router.PrintProxyOnFd

Synopsis

`Glacier.Router.PrintProxyOnFd=fd`

Description

If set, print the stringified router proxy on the filedescriptor *fd*, and close this file-descriptor. (Unix only.)

This operation is intended to be used by the Glacier router starter only. It should not be set manually.

Glacier.Router.Trace.Client

Synopsis

`Glacier.Router.Trace.Client=num`

Description

The client interface trace level:

0	No client interface trace. (default)
1	Trace exceptions during request forwarding from the client to the server.

2	Also trace detailed forward routing information from the client to the server.
---	--

Glacier.Router.Trace.Server

Synopsis

Glacier.Router.Trace.Server=*num*

Description

The server interface trace level:

0	No server interface trace. (default)
1	Trace exceptions during callbacks from the server back to the client.
2	Also trace detailed reverse routing information for callbacks from the server to the client.

Glacier.Router.Trace.RoutingTable

Synopsis

Glacier.Router.Trace.RoutingTable=*num*

Description

The routing table trace level:

0	No routing table trace. (default)
1	Trace additions to the Glacier routing table.

Glacier.Router.Trace.Throttle

Synopsis

Glacier.Router.Trace.Throttle=*num*

Description

The request throttle trace level:

0	No request throttle tracing. (default)
1	Trace throttled requests.

Glacier.Router.Client.ForwardContext, Glacier.Router.Server.ForwardContext

Synopsis

Glacier.Router.Client.ForwardContext=*num*
Glacier.Router.Server.ForwardContext=*num*

Description

If *num* is set to a value larger than zero, the context parameter is forwarded unmodified as received from the client or server, respectively. Otherwise an empty context is forwarded. Default is no context forwarding.

Glacier.Router.Client.SleepTime, Glacier.Router.Server.SleepTime

Synopsis

Glacier.Router.Client.SleepTime=*num*
Glacier.Router.Server.SleepTime=*num*

Description

num is the sleep time (delay) in milliseconds after a request or message batch has been forwarded. The default value is zero, meaning no delay. Setting these values avoids message flooding. This is useful to avoid denial-of-service attacks, or to allow a minimum time for collecting messages for batching.

Glacier.Router.Client.Throttle.Twoways, Glacier.Router.Server.Throttle.Twoways

Synopsis

Glacier.Router.Client.Throttle.Twoways=*num*
Glacier.Router.Server.Throttle.Twoways=*num*

Description

num is the maximum number of twoway requests that the router will forward simultaneously. The default is zero, meaning no limit. Enabling twoway message throttling is useful to prevent a router from consuming too many resources of backend services.

Glacier.Router.SessionManager

Synopsis

Glacier.Router.SessionManager=*proxy*

Description

A stringified proxy to a session manager. If not specified, it is not possible to use the Router::createSession() method.

Glacier.Router.UserId

Synopsis

Glacier.Router.UserId=*name*

Description

The authenticated user id. This is usually passed from the Glacier router starter. The user id is used as an argument to Router::createSession().

Glacier.Router.AllowCategories

Synopsis

Glacier.Router.AllowCategories=*list*

Description

A white space separated list of categories. If this property is set, then requests are only permitted to Ice objects with an identity that matches one of the categories from this list.

Glacier.Router.AcceptCert**Synopsis**

`Glacier.Router.AcceptCert=base64 encoded certificate string`

Description

A base64 encoded certificate (which can be obtained by calling `certToBase64()` on an existing `IceSSL::RSAKeyPair`). The Glacier Router will use this certificate, when in SSL mode, to restrict those clients that may connect to it. Only clients that use this certificate may connect, others will be rejected.

C.14 Glacier Router Starter Properties

Glacier.Starter.Endpoints**Synopsis**

`Glacier.Starter.Endpoints=endpoints`

Description

Defines the endpoints of the Glacier router starter. (Unix only.)

Glacier.Starter.PasswordVerifier**Synopsis**

`Glacier.Starter.PasswordVerifier=proxy`

Description

If set, the specified password verifier will be used. If not set, a built-in crypt-based password verifier will be used.

Glacier.Starter.CryptPasswords

Synopsis

`Glacier.Starter.CryptPasswords=file`

Description

The pathname of the file that contains user-id / password pairs, with the passwords being encrypted by the crypt algorithm. The default pathname is "passwords". This file is only used for the built-in crypt-based password verifier, meaning that the property is ignored if `Glacier.Starter.PasswordVerifier` is set.

Glacier.Starter.RouterPath

Synopsis

`Glacier.Starter.RouterPath=path`

Description

Sets the path of the Glacier router executable to be started. The default is `glacier`. (Unix only.)

Glacier.Starter.PropertiesOverride

Synopsis

`Glacier.Starter.PropertiesOverride=overrides`

Description

By default, the Glacier router starter starts new routers with the exact same property set as for the router starter. *overrides* can contain a list of properties for the router, which are used in addition to the router starter's properties, or which override the router starter's properties. The property definitions should be separated by white space.

For example, in many cases it is desirable to set the property `Ice.ServerIdleTime` for the router, but not for the router starter. For an idle time of 60 seconds, this can be done by setting `Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60`. (Unix only.)

Glacier.Starter.StartupTimeout

Synopsis

`Glacier.Starter.StartupTimeout=num`

Description

num is the number of seconds the Glacier router starter will wait for the router to start up. If this timeout expires, a `Glacier: : CannotStartRouterException` is returned to the caller. The default value is 10 seconds. Timeout values smaller than one second are silently changed to 1 second. (Unix only.)

Glacier.Starter.AddUserToAllowCategories

Synopsis

`Glacier.Starter.AddUserToAllowCategories=num`

Description

Control the addition of the user id authenticated by the Glacier router starter to the `Glacier.Router.AllowCategories` property upon router startup:

0	Do not add user id. (default)
1	Add user id.
2	Add user id with prepended underscore.

Glacier.Starter.Trace

Synopsis

`Glacier.Starter.Trace=num`

Description

The router starter trace level: (Unix only.)

0	No router starter trace. (default)
---	------------------------------------

1	Trace router startup exceptions.
2	Also trace each successful router startup.

Glacier.Starter.Certificate.Country

Synopsis

Glacier.Starter.Certificate.Country=*country code*

Description

This property specifies the country code portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. Examples of valid values for this field are "US" for the United States and "CA" for Canada. The default value is "US".

Glacier.Starter.Certificate.StateProvince

Synopsis

Glacier.Starter.Certificate.StateProvince=*state/province code*

Description

This property specifies the state or province code portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. Examples of valid values for this field are "CA" for California and "British Columbia" for British Columbia, Canada. The default value is "DC".

Glacier.Starter.Certificate.Locality

Synopsis

Glacier.Starter.Certificate.Locality=*city or town name*

Description

This property specifies the locality portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The locality is usually the name of the city or town. The default value is "Washington".

Glacier.Starter.Certificate.Organization**Synopsis**

Glacier.Starter.Certificate.Organization=*organization or company name*

Description

This property specifies the organization portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The organization is usually the name of the company or organization to which the certificate has been granted. The default value is "Some Company Inc.".

Glacier.Starter.Certificate.OrganizationalUnit**Synopsis**

Glacier.Starter.Certificate.OrganizationalUnit=*department*

Description

This property specifies the organizational unit portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The organization unit is usually the name of the department within the company or organization structure to which the certificate has been granted. The default value is "Sales".

Glacier.Starter.Certificate.CommonName**Synopsis**

Glacier.Starter.Certificate.CommonName=*contact name*

Description

This property specifies the common name portion of the Distinguished Name (DN) values that will be present in certificates generated by the Glacier Router Starter for client applications and the Glacier Router itself. The common name is usually the name of the contact (a person's name) within the company and department to which the certificate has been granted. The default value is "John Doe".

Glacier.Starter.Certificate.BitStrength**Synopsis**

Glacier.Starter.Certificate.BitStrength=*number of bits*

Description

This property specifies the bit strength which will be used in the generation of certificates by the Glacier Router Starter for client applications and the Glacier Router itself. This value is the modulus size of the RSA key.

Although modulus size is determined by the needs of the particular application, care should be taken not to specify a size that is too large, as certificate generation is an expensive operation. Sizes less than 512 bits are not supported, and sizes that exceed 2048 bits should be chosen with generation times in mind. The default value is 1024 bits.

Glacier.Starter.Certificate.SecondsValid**Synopsis**

Glacier.Starter.Certificate.SecondsValid=*seconds*

Description

This property specifies the number of seconds that certificates generated by the Glacier Router Starter are valid before they will expire. The default value of 1 day (86,400 seconds).

Glacier.Starter.Certificate.IssuedAdjust

Synopsis

Glacier.Starter.Certificate.IssuedAdjust=(+/-) *seconds*

Description

This property adjusts the issued timestamp on dynamically generated certificates for the Glacier Router. The default value is 0, which sets the issued time based on the system time when the certificate is actually created. Positive adjustments will adjust the timestamp into the future, and negative adjustments will cause the timestamp to be in the past by the number of seconds indicated. This adjustment is relative to server time.

C.15 Freeze Properties

Freeze.DbEnv.*env-name*.DbCheckpointPeriod

Synopsis

Freeze.DbEnv.*env-name*.DbCheckpointPeriod=*num*

Description

Every Berkeley DB environment created by Freeze has an associated thread that checkpoints this environment every *num* seconds. Defaults to 120 seconds.

Freeze.DbEnv.*env-name*.DbHome

Synopsis

Freeze.DbEnv.*env-name*.DbHome=*db-home*

Description

Defines the home directory of this Freeze database environment. Defaults to *env-name*.

Freeze.DbEnv.*env-name*.DbPrivate

Synopsis

Freeze.DbEnv. *env-name*. DbPri vate=*num*

Description

If *num* is set to a value larger than zero, Freeze will instruct Berkeley DB to use process private memory instead of shared memory. The default value is 1. Set it to 0 in order to run db_archive (or another Berkeley DB utility) on a running environment.

Freeze.DbEnv.*env-name*.DbRecoverFatal

Synopsis

Freeze.DbEnv. *env-name*. DbRecoverFatal =*num*

Description

If *num* is set to a value larger than zero, "fatal" recovery will be performed when the environment is opened. The default value is 0.

Freeze.DbEnv.*env-name*.OldLogsAutoDelete

Synopsis

Freeze.DbEnv. *env-name*. Ol dLogsAutoDel ete=*num*

Description

If *num* is set to a value larger than zero, old transactional logs no longer in use are deleted after each periodic checkpoint (see Freeze.DbEnv. *env-name*. DbCheckpoi ntPeri od). The default value is 1.

Freeze.DbEnv.*env-name*.PeriodicCheckpointMinSize

Synopsis

Freeze.DbEnv. *env-name*. Peri odi cCheckpoi ntMi nSi ze=*num*

Description

num is the minimum size in kbytes for the periodic checkpoint (see Freeze.DbEnv. *env-name*. DbCheckpointPeriod). This value is passed to Berkeley DB's checkpoint function. Defaults to 0 (which means no minimum).

Freeze.Evictor.*env-name*.*db-name*.MaxTxSize**Synopsis**

Freeze.Evictor. *env-name*. *db-name*. MaxTxSize=*num*

Description

Freeze uses a background thread to save updates to the database. Transactions are used to save many facets together. *num* defines the maximum number of facets saved per transaction. Defaults to 10 * SaveSizeTrigger (see Freeze.Evictor. *env-name*. *db-name*. SaveSizeTrigger); if this value is negative, the actual value is set to 100.

Freeze.Evictor.*env-name*.*db-name*.SavePeriod**Synopsis**

Freeze.Evictor. *env-name*. *db-name*. SavePeriod=*num*

Description

Freeze uses a background thread to save updates to the database. After *num* milliseconds without saving, if there is any facet created, modified or destroyed, this background thread wakes up to save these facets. When *num* is 0, there is no periodic saving. Defaults to 60,000.

Freeze.Evictor.*env-name*.*db-name*.SaveSizeTrigger**Synopsis**

Freeze.Evictor. *env-name*. *db-name*. SaveSizeTrigger=*num*

Description

Freeze uses a background thread to save updates to the database. When *num* is 0 or positive, as soon as *num* or more facets have been created, modified or destroyed, this background thread wakes up to save them. When *num* is negative, there is no size trigger. Defaults to 10.

Freeze.Trace.DbEnv

Synopsis

Freeze.Trace.DbEnv=*num*

Description

The Freeze database environment activity trace level:

0	No database environment activity trace. (default)
1	Trace database open and close.
2	Also trace checkpoints and the removal of old log files.

Freeze.Trace.Evictor

Synopsis

Freeze.Trace.Evictor=*num*

Description

The Freeze evictor activity trace level:

0	No evictor activity trace. (default)
1	Trace Ice object and facet creation and destruction, facet streaming time, facet saving time, object eviction (every 50 objects) and evictor deactivation.
2	Also trace object lookups, and all object evictions.

3	Also trace object retrieval from the database.
---	--

Freeze.Trace.Map

Synopsis

Freeze.Trace.Map=*num*

Description

The Freeze map activity trace level:

0	No map activity trace. (default)
1	Trace database open and close.
2	Also trace iterator and transaction operations, and reference counting of the underlying database.

Appendix D

Proxies and Endpoints

D.1 Proxies

Synopsis

```
identity -f facet -t -o -O -d -D -s @ adapter_id :  
endpoints
```

Description

A stringified proxy consists of an identity, proxy options, and an optional object adapter identifier or endpoint list. A proxy containing an identity with no endpoints, or an identity with an object adapter identifier, represents an indirect proxy that will be resolved using the Ice locator (see the `Ice.Default.Locator` property).

Proxy options configure the invocation mode:

-f <i>facet</i>	Select a facet of the Ice object.
-t	Configures the proxy for twoway invocations. (default)

-o	Configures the proxy for oneway invocations.
-O	Configures the proxy for batch oneway invocations.
-d	Configures the proxy for datagram invocations.
-D	Configures the proxy for batch datagram invocations.
-s	Configures the proxy for secure invocations.

The proxy options `-t`, `-o`, `-O`, `-d`, and `-D` are mutually exclusive.

The object identity *identity* is structured as `[category/]name`, where the *category* component and slash separator are optional. If *identity* contains white space or either of the characters `:` or `@`, it must be enclosed in single or double quotes. The *category* and *name* components are UTF8 strings that use the encoding described below. Any occurrence of a slash (`/`) in *category* or *name* must be escaped with a backslash (i.e., `\`).

The *facet* argument of the `-f` option represents a facet path comprising one or more facets separated by a slash (`/`). If *facet* contains white space, it must be enclosed in single or double quotes. Each component of the facet path is a UTF8 string that uses the encoding described below. Any occurrence of a slash (`/`) in a facet path component must be escaped with a backslash (i.e., `\`).

The object adapter identifier *adapter_id* is a UTF8 string that uses the encoding described below. If *adapter_id* contains white space, it must be enclosed in single or double quotes.

UTF8 strings are encoded using ASCII characters for the ordinal range 32-126 (inclusive). Characters outside this range must be encoded using escape sequences (`\b`, `\f`, `\n`, `\r`, `\t`) or octal notation (e.g., `\007`). Quotation marks used to enclose a string can be escaped using a backslash, as can the backslash itself (`\\`).

If endpoints are specified, they must be separated with a colon (`:`) and formatted as described in Endpoints. The order of endpoints in the stringified proxy is not necessarily the order in which connections are attempted during binding: when a stringified proxy is converted into a proxy instance, the endpoint list is randomized as a form of load balancing.

If the `-s` option is specified, only those endpoints that support secure invocations are considered during binding. If no valid endpoints are found, the application receives `Ice::NoEndpointException`.

Otherwise, if the `-s` option is not specified, the endpoint list is ordered so that non-secure endpoints have priority over secure endpoints during binding. In other

words, connections are attempted on all non-secure endpoints before any secure endpoints are attempted.

If an unknown option is specified, or the stringified proxy is malformed, the application receives `Ice: : ProxyParseException`. If an endpoint is malformed, the application receives `Ice: : EndpointParseException`.

D.2 Endpoints

Synopsis

```
endpoint : endpoint
```

Description

An endpoint list comprises one or more endpoints separated by a colon (:). An endpoint has the following format: ***protocol option*** The supported protocols are `tcp`, `udp`, `ssl`, and `default`. If `default` is used, it is replaced by the value of the `Ice.Default.Protocol` property. If an endpoint is malformed, or an unknown protocol is specified, the application receives `Ice: : EndpointParseException`.

The `ssl` protocol is only available if the `IceSSL` plug-in is installed.

The protocols and their supported options are described in the sections that follow.

TCP Endpoint

Synopsis

```
tcp -h host -p port -t timeout -z
```

Description

A tcp endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
<i>-h host</i>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>Ice.Default.Host</code> is used instead.	Determines the hostname or IP address to which a connection attempt is made.	Determines the network interface on which the object adapter listens for connections, as well as the hostname that is advertised in proxies created by the adapter.
<i>-p port</i>	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made. (required)	The port will be selected by the operating system if this option is not specified or <i>port</i> is zero.

Option	Description	Client Semantics	Server Semantics
<i>-t timeout</i>	Specifies the endpoint timeout in milliseconds.	If <i>timeout</i> is greater than zero, it sets a maximum delay for binding and proxy invocations. If a timeout occurs, the application receives <code>Ice::TimeoutException</code> .	Determines the default timeout that is advertised in proxies created by the object adapter.
<i>-z</i>	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compressed responses are sent, as well as whether compression is advertised in proxies created by the adapter.

UDP Endpoint

Synopsis

```
udp -v major.minor -e major.minor -h host -p port -c -z
```

Description

A udp endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
<i>-v major.minor</i>	Specifies the protocol major and highest minor version number to be used for this endpoint. If not specified, the protocol major version and highest supported minor version of the client-side Ice run time is used.	Determines the protocol major version and highest minor version used by the client side when sending messages to this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.	Determines the protocol major version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.

Option	Description	Client Semantics	Server Semantics
<i>-e major.minor</i>	Specifies the encoding major and highest minor version number to be used for this endpoint. If not specified, the encoding major version and highest supported minor version of the client-side Ice run time is used.	Determines the encoding major version and highest minor version used by the client side when sending messages to this endpoint. The encoding major version number must match the encoding major version number of the server; the encoding minor version number must not be higher than the highest minor version number supported by the server.	Determines the encoding version and highest minor version advertised by the server side for this endpoint. The protocol major version number must match the protocol major version number of the server; the protocol minor version number must not be higher than the highest minor version number supported by the server.

Option	Description	Client Semantics	Server Semantics
-h <i>host</i>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>DefaultHost</code> is used instead.	Determines the hostname or IP address to which datagrams are sent.	Determines the network interface on which the object adapter listens for datagrams, as well as the hostname that is advertised in proxies created by the adapter.
-p <i>port</i>	Specifies the port number of the endpoint.	Determines the port to which datagrams are sent. (required)	The port will be selected by the operating system if this option is not specified or <i>port</i> is zero.
-c	Specifies that a connected UDP socket should be used.	None.	Causes the server to connect to the socket of the first peer that sends a datagram to this endpoint.

Option	Description	Client Semantics	Server Semantics
-z	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compression is advertised in proxies created by the adapter.

SSL Endpoint

Synopsis

```
ssl -h host -p port -t timeout -z
```

Description

An ssl endpoint supports the following options:

Option	Description	Client Semantics	Server Semantics
-h <i>host</i>	Specifies the hostname or IP address of the endpoint. If not specified, the value of <code>Ice.Default.Host</code> is used instead.	Determines the hostname or IP address to which a connection attempt is made.	Determines the network interface on which the object adapter listens for connections, as well as the hostname that is advertised in proxies created by the adapter.

Option	Description	Client Semantics	Server Semantics
<code>-p port</code>	Specifies the port number of the endpoint.	Determines the port to which a connection attempt is made. (required)	The port will be selected by the operating system if this option is not specified or <i>port</i> is zero.
<code>-t timeout</code>	Specifies the endpoint timeout in milliseconds.	If <i>timeout</i> is greater than zero, it sets a maximum delay for binding and proxy invocations. If a timeout occurs, the application receives <code>Ice::TimeoutException</code> .	Determines the default timeout that is advertised in proxies created by the object adapter.
<code>-z</code>	Specifies bzip2 compression.	Determines whether compressed requests are sent.	Determines whether compressed responses are sent, as well as whether compression is advertised in proxies created by the adapter.

Bibliography

References

1. Booch, G., et al. 1998. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.
2. Gamma, E., et al. 1994. *Design Patterns*. Reading, MA: Addison-Wesley.
3. Grimes, R. 1998. *Professional DCOM Programming*. Chicago, IL: Wrox Press.
4. Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.
5. Housley, R., and T. Polk. 2001. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. Hoboken, NJ: Wiley.
6. Institute of Electrical and Electronics Engineers. 1985. *IEEE 754-1985 Standard for Binary Floating-Point Arithmetic*. Piscataway, NJ: Institute of Electrical and Electronic Engineers.
7. Jain, P., et al. 1997. "Dynamically Configuring Communication Services with the Service Configurator Pattern." *C++ Report* 9 (6).
8. Kleiman, S., et al. 1995. *Programming With Threads*. Englewood, NJ: Prentice Hall.
9. Lakos, J. 1996. *Large-Scale C++ Software Design*. Reading, MA: Addison-Wesley.

10. McKusick, M. K., et al. 1996. *The Design and Implementation of the 4.4BSD Operating System*. Reading, MA: Addison-Wesley.
11. Microsoft. 2002. *.NET Infrastructure & Services*.
<http://www.microsoft.com/technet/treeview/default.asp?url=/technet/itsolutions/net/Default.asp>. Bellevue, WA: Microsoft.
12. Mitchell, J. G., et al. 1979. *Mesa Language Manual*. CSL-793. Palo Alto, CA: Xerox PARC.
13. Object Management Group. 2001. *Unified Modeling Language Specification*.
<http://www.omg.org/technology/documents/formal/uml.htm>. Framingham, MA: Object Management Group.
14. The Open Group. 1997. *DCE 1.1: Remote Procedure Call*. Technical Standard C706. <http://www.opengroup.org/publications/catalog/c706.htm>. Cambridge, MA: The Open Group.
15. Prescod, P. 2002. *Some Thoughts About SOAP Versus REST on Security*.
<http://www.prescod.net/rest/security.html>.
16. Red Hat, Inc. 2003. *The bzip2 and libbzip2 Home Page*.
<http://sources.redhat.com/bzip2>. Raleigh, NC: Red Hat, Inc.
17. Schmidt, D. C. et al. 2000. "Leader/Followers: A Design Pattern for Efficient Multi-Threaded Event Demultiplexing and Dispatching". In *Proceedings of the 7th Pattern Languages of Programs Conference*, WUCS-00-29, Seattle, WA: University of Washington. <http://www.cs.wustl.edu/~schmidt/PDF/lf.pdf>.
18. Sleepycat Software, Inc. 2003. *Berkeley DB Technical Articles*.
<http://www.sleepycat.com/company/technical.shtml>. Lincoln, MA: Sleepycat Software, Inc.
19. Sutter, H. 1999. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.
20. Sutter, H. 2002. "A Pragmatic Look at Exception Specifications." *C/C++ Users Journal* 20 (7): 59–64. <http://www.gotw.ca/publications/mill22.htm>.
21. Unicode Consortium, ed. 2000. *The Unicode Standard, Version 3.0*. Reading, MA: Addison-Wesley. <http://www.unicode.org/unicode/uni2book/u2.html>.
22. Viega, J., et al. 2002. *Network Security with OpenSSL*. Sebastopol, CA: O'Reilly.
23. World Wide Web Consortium. 2002. *SOAP Version 1.2 Specification*.
<http://www.w3.org/2000/xml/Group/#soap12>. Boston, MA: World Wide Web Consortium.

24. World Wide Web Consortium. 2002. *Web Services Activity*.
<http://www.w3.org/2002/ws>. Boston, MA: World Wide Web Consortium.

