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:eay@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。

Note: 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	Internet Communications Engine （Ice）	3
	1.3	本书的篇章结构	4
	1.4	排字约定	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 在架构上提供的好处	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 Package 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 `Ice::Context` 413
16.9 调用超时 418
16.10 单向调用 419
16.11 数据报调用 423
16.12 成批的调用 424
16.13 测试代理的分派类型 426
16.14 `Ice::Logger` 接口 426
16.15 `Ice::Stats` 428
16.16 位置透明性 429
16.17 对比 Ice 与 CORBA Run Time 430
16.18 总结 432

第 17 章 异步程序设计 443

17.1 本章综述 443
17.2 引言 443
17.3 使用 AMI 446
17.4 使用 AMD 452
17.5 总结 459

第 18 章 Ice 协议 463

18.1 本章综述 463
18.2 数据编码 463
18.3 协议消息 486
18.4 压缩 494
18.5 协议和编码版本 496
18.6 与 IIOP 的对比 498

第 19 章 Ice 的 PHP 扩展 511

19.1 本章综述 511
19.2 引言 511
19.3 配置 513
19.4 客户端的 Slice-to-PHP 映射 515

第四部分 Ice 服务 533

第 20 章 IcePack 535
20.1 本章综述 535
20.2 介绍 535
20.3 概念 536
20.4 Ice 定位器设施 537
20.5 IcePack 注册表 541
20.6 IcePack 节点 545
20.7 IcePack 管理工具 548
20.8 部署 551
20.9 描述符参考资料 557
20.10 排除故障 565
20.11 总结 568

第 21 章 Freeze 569
21.1 本章综述 569
21.2 引言 570
21.3 Freeze 映射表 570
21.4 在文件系统服务器中使用 Freeze 映射表 578
21.5 Freeze 逐出器 603
21.6 在文件服务器中使用 Freeze 逐出器 608
21.7 总结 627

第 22 章 FreezeScript 629
22.1 本章综述 629
22.2 引言 629
22.3 数据库迁移 630
22.4 转换描述符 635
22.5 使用 **`transformdb`** 647
22.6 数据库审查 652
22.7 使用 **`dumpdb`** 662
22.8 描述符表达式语言 665
22.9 总结 668

第 23 章 IceSSL 669
23.1 本章综述 669
23.2 引言 669
23.3 配置 IceSSL 671
23.4 配置应用 674
23.5 配置参考资料 676
23.6 IceSSL 编程 684
23.7 总结 685

第 24 章	**Glacier**	**687**
	24.1 本章综述	687
	24.2 引言	687
	24.3 使用 Glacier	691
	24.4 回调	692
	24.5 Glacier 启动器	696
	24.6 Glacier 的安全性	701
	24.7 总结	706
第 25 章	**IceBox**	**707**
	25.1 本章综述	707
	25.2 引言	707
	25.3 服务管理器	708
	25.4 开发服务	709
	25.5 启动 IceBox	715
	25.6 总结	716
第 26 章	**IceStorm**	**719**
	26.1 本章综述	719
	26.2 引言	720
	26.3 概念	721
	26.4 IceStorm 接口综述	723
	26.5 使用 IceStorm	725
	26.6 IceStorm 的管理	734
	26.7 主题联盟	735
	26.8 服务质量	742
	26.9 配置 IceStorm	743
	26.10 总结	744

附录 745

附录 A	**Slice 关键字**	**747**

附录 B Slice 文档 749
B.1 Ice 749
B.2 Ice::AbortBatchRequestException 753
B.3 Ice::AdapterAlreadyActiveException 753
B.4 Ice::AdapterNotFoundException 753
B.5 Ice::AlreadyRegisteredException 753
B.6 Ice::BadMagicException 754
B.7 Ice::CloseConnectionException 754
B.8 Ice::CloseTimeoutException 755
B.9 Ice::CollocationOptimizationException 755
B.10 Ice::Communicator 755
B.11 Ice::CommunicatorDestroyedException 763
B.12 Ice::CompressionException 763
B.13 Ice::CompressionNotSupportedException 764
B.14 Ice::ConnectFailedException 764
B.15 Ice::ConnectTimeoutException 764
B.16 Ice::ConnectionLostException 764
B.17 Ice::ConnectionNotValidatedException 765
B.18 Ice::ConnectionTimeoutException 765
B.19 Ice::Current 765
B.20 Ice::DNSException 767
B.21 Ice::DatagramLimitException 767
B.22 Ice::EncapsulationException 768
B.23 Ice::EndpointParseException 768
B.24 Ice::FacetNotExistException 768
B.25 Ice::Identity 769
B.26 Ice::IdentityParseException 770
B.27 Ice::IllegalIdentityException 770
B.28 Ice::IllegalIndirectionException 770
B.29 Ice::IllegalMessageSizeException 771
B.30 Ice::Locator 771
B.31 Ice::LocatorRegistry 772
B.32 Ice::Logger 774
B.33 Ice::MarshalException 775
B.34 Ice::MemoryLimitException 776
B.35 Ice::NegativeSizeException 776
B.36 Ice::NoEndpointException 776
B.37 Ice::NoObjectFactoryException 777
B.38 Ice::NotRegisteredException 777
B.39 Ice::ObjectAdapter 778
B.40 Ice::ObjectAdapterDeactivatedException 786
B.41 Ice::ObjectAdapterIdInUseException 787

B.42 Ice::ObjectFactory 787
B.43 Ice::ObjectNotExistException 788
B.44 Ice::ObjectNotFoundException 789
B.45 Ice::OperationMode 789
B.46 Ice::OperationNotExistException 790
B.47 Ice::Plugin 790
B.48 Ice::PluginInitializationException 791
B.49 Ice::PluginManager 791
B.50 Ice::Process 792
B.51 Ice::Properties 793
B.52 Ice::ProtocolException 797
B.53 Ice::ProxyParseException 798
B.54 Ice::ProxyUnmarshalException 798
B.55 Ice::RequestFailedException 799
B.56 Ice::Router 799
B.57 Ice::ServantLocator 801
B.58 Ice::ServerNotFoundException 803
B.59 Ice::SocketException 803
B.60 Ice::Stats 803
B.61 Ice::SyscallException 804
B.62 Ice::TimeoutException 805
B.63 Ice::TwowayOnlyException 805
B.64 Ice::UnknownException 806
B.65 Ice::UnknownLocalException 806
B.66 Ice::UnknownMessageException 807
B.67 Ice::UnknownReplyStatusException 807
B.68 Ice::UnknownRequestIdException 807
B.69 Ice::UnknownUserException 807
B.70 Ice::UnmarshalOutOfBoundsException 808
B.71 Ice::UnsupportedEncodingException 808
B.72 Ice::UnsupportedProtocolException 809
B.73 Ice::VersionMismatchException 810
B.74 Freeze 810
B.75 Freeze::Connection 811
B.76 Freeze::DatabaseException 812
B.77 Freeze::DeadlockException 812
B.78 Freeze::EmptyFacetPathException 813
B.79 Freeze::Evictor 813
B.80 Freeze::EvictorDeactivatedException 819
B.81 Freeze::EvictorIterator 819
B.82 Freeze::EvictorStorageKey 821
B.83 Freeze::InvalidPositionException 821

B.84 Freeze::NoSuchElementException 821
B.85 Freeze::NotFoundException 822
B.86 Freeze::ObjectRecord 822
B.87 Freeze::ServantInitializer 822
B.88 Freeze::Statistics 823
B.89 Freeze::Transaction 824
B.90 Freeze::TransactionAlreadyInProgressException 825
B.91 IceBox 825
B.92 IceBox::FailureException 825
B.93 IceBox::FreezeService 826
B.94 IceBox::Service 827
B.95 IceBox::ServiceBase 828
B.96 IceBox::ServiceManager 828
B.97 IcePack 829
B.98 IcePack::AdapterDeploymentException 829
B.99 IcePack::AdapterNotExistException 829
B.100 IcePack::Admin 830
B.101 IcePack::BadSignalException 841
B.102 IcePack::DeploymentException 841
B.103 IcePack::NodeNotExistException 842
B.104 IcePack::NodeUnreachableException 842
B.105 IcePack::ObjectDeploymentException 842
B.106 IcePack::ObjectExistsException 843
B.107 IcePack::ObjectNotExistException 843
B.108 IcePack::ParserDeploymentException 843
B.109 IcePack::Query 844
B.110 IcePack::ServerActivation 845
B.111 IcePack::ServerDeploymentException 846
B.112 IcePack::ServerDescription 846
B.113 IcePack::ServerNotExistException 848
B.114 IcePack::ServerState 848
B.115 IceSSL 849
B.116 IceSSL::CertificateException 850
B.117 IceSSL::CertificateKeyMatchException 850
B.118 IceSSL::CertificateLoadException 850
B.119 IceSSL::CertificateParseException 851
B.120 IceSSL::CertificateSignatureException 851
B.121 IceSSL::CertificateSigningException 851
B.122 IceSSL::CertificateVerificationException 851
B.123 IceSSL::CertificateVerifier 852
B.124 IceSSL::CertificateVerifierTypeException 853
B.125 IceSSL::ConfigParseException 853

B.126 IceSSL::ConfigurationLoadingException 853
B.127 IceSSL::ContextException 854
B.128 IceSSL::ContextInitializationException 854
B.129 IceSSL::ContextNotConfiguredException 854
B.130 IceSSL::ContextType 855
B.131 IceSSL::Plugin 856
B.132 IceSSL::PrivateKeyException 860
B.133 IceSSL::PrivateKeyLoadException 860
B.134 IceSSL::PrivateKeyParseException 860
B.135 IceSSL::ProtocolException 860
B.136 IceSSL::ShutdownException 861
B.137 IceSSL::SslException 861
B.138 IceSSL::TrustedCertificateAddException 862
B.139 IceSSL::UnsupportedContextException 862
B.140 Glacier 862
B.141 Glacier::CannotStartRouterException 862
B.142 Glacier::NoSessionManagerException 863
B.143 Glacier::PermissionDeniedException 863
B.144 Glacier::PermissionsVerifier 863
B.145 Glacier::Router 864
B.146 Glacier::Session 865
B.147 Glacier::SessionManager 865
B.148 Glacier::Starter 866
B.149 IceStorm 867
B.150 IceStorm::LinkExists 868
B.151 IceStorm::LinkInfo 869
B.152 IceStorm::NoSuchLink 869
B.153 IceStorm::NoSuchTopic 870
B.154 IceStorm::Topic 870
B.155 IceStorm::TopicExists 873
B.156 IceStorm::TopicManager 874
B.157 IcePatch 875
B.158 IcePatch::BusyException 876
B.159 IcePatch::Directory 876
B.160 IcePatch::DirectoryDesc 876
B.161 IcePatch::File 877
B.162 IcePatch::FileAccessException 877
B.163 IcePatch::FileDesc 877
B.164 IcePatch::Regular 878
B.165 IcePatch::RegularDesc 879

附录 C 属性 881
C.1 Ice 配置属性 881
C.2 Ice 跟踪属性 882
C.3 Ice 警告属性 884
C.4 Ice 对象适配器属性 886
C.5 Ice 插件属性 888
C.6 Ice 线程池属性 888
C.7 Ice 的 Default 和 Override 属性 890
C.8 Ice 杂项属性 892
C.9 IceSSL 属性 897
C.10 IceBox 属性 901
C.11 IcePack 属性 903
C.12 IceStorm 属性 910
C.13 Glacier 路由器属性 913
C.14 Glacier 路由器启动器属性 918
C.15 Freeze 属性 923

附录 D 代理与端点 929
D.1 代理 929
D.2 端点 931

参考文献 935

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：在本书撰写的同时，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，另外还有几个附录:

- 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，以及异步方法调用与分派（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*。客户通过调用操作来发出请求。
- 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：当客户调用代理上的操作时，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，或是进行相反的转换。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 （为了能调用操作），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调用（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调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 进行注册。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布式系统。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。

同步方法调用（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：操作调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。

异步方法调用（Asynchronous Method Invocation）

Ice 还支持*异步方法调用*（AMI）：客户可以*异步地*调用操作，也就是说，客户像平常一样使用代理来调用操作，但除了传递通常的参数以外，还要传递一个*回调对象*，而客户调用会立即返回。一旦操作完成，客户端端 run time 会调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，对操作调用作出响应。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调用。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，当操作调用到达时，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调用中，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，但在操作调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。

单向方法调用 （Oneway Method Invocation）

客户可以把操作当作*单向*操作来调用。单向调用具有 “尽力”语义。在处理单向调用时，客户端 run time 会把调用交给本地传输机制，只要本地传输机制缓冲了该调用，客户端的调用就算完成了。随后，实际的调用会由操作系统异步发送。服务器不会答复单向调用，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。

单向调用是不可靠的。例如，目标对象可能不存在，在这种情况下，调用会丢失。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），才能进行单向调用。

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，没有办法区分双向调用和单向调用。

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），才能使用单向调用。

注意，即使单向操作是在面向流的传输机制上发送的，服务器也可能会不按次序处理它们。之所以会这样，是因为每个调用都在它自己的线程中被分派：即使调用*发起*（initiated）的次序与调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。

成批的单向方法调用 （Batched Oneway Method Invocation）

每个单向调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。

*成批的单向调用*允许你在一条消息中发送一系列单向调用：每次你调用一个这样的单向操作，调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调用，你就发出另外一个 API 调用，一次发送所有调用。客户端 run time 随即在单条消息中发送所有已缓冲的调用，而服务器会在单条消息中收到所有调用。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调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。

数据报调用 （Datagram Invocations）

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

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

数据报和单向调用一样是不可靠的：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调用，但由于客户发送的参数无效，导致调用失败。与单向调用一样，客户也不会收到关于这些错误的通知。

但与单向调用不同，数据报调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，如果你依次调用三个操作，中间的调用可能会丢失（单向调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，不可能发生个别调用丢失的情况）。

- 各个调用可能会不按次序到达。

  这也是 UDP 数据报的本性所致。因为每个调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：各个调用的到达次序与它们的发送次序不一样。

数据报调用很适用于 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的情形，比如快速的交互式 Internet 应用。

成批的数据报调用 （Batched Datagram Invocations）

与成批的单向调用一样，*成批的数据报调用*允许你在缓冲区中累积一批调用，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，成批的数据报调用才有用：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，一个成批消息中的调用或者全部被递送，或者一个也不被递送。不可能发生一个成批消息中的个别调用丢失的情况。

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调用。这保证了各个调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）

任何操作调用都可以引发*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:

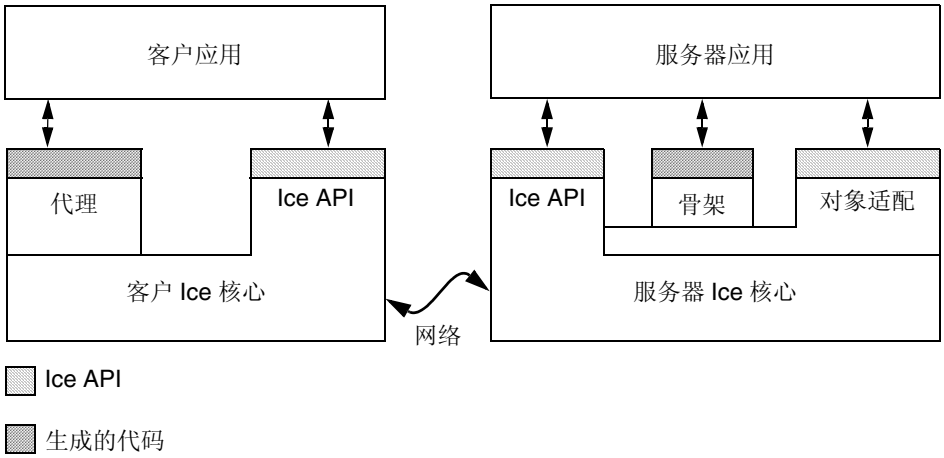

图 2.1. Ice 客户与服务器的结构

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器端运行时支持。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器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。用于客户和服务器的 Ice API 是一样的 （尽管服务器使用的 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，调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 run time 把控制线程转交给你编写的应用代码。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，能让我们更轻松地管理数据库，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：随需启动服务器、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 允许你注册服务器，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，随需启动服务器。
- 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的版本，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。所有的操作调用都使用迟后绑定，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

  客户和服务器可以分别部署，所用语言也可以不同（目前支持 C++、Java，以及 PHP（客户端））。客户和服务器所用的 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

  客户不知道服务器是怎样实现其对象的。这意味着，在客户部署之后，服务器的实现可以改变，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器付出额外努力。

- 传输机制无关性

  Ice 目前采用了 TCP/IP 和 UDP 作为传输协议。客户和服务器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）。

- 位置和服务器透明性

  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。客户与服务器之间的交互显得像是无连接的。如果在客户调用操作时，服务器没有运行，你可以通过 IcePack 让它们随需启动。服务器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器进程上的。

- 安全性

  通过 SSL 强加密，可以使客户和服务器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，但它们在一些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）。

允许客户随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。
- 不需要对串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，你必须调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），这样的比较也至关紧要。

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 支持异步方法调用，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， CORBAO 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，我们现在也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，并让你感受一下 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。我们为我们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。在客户和服务器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器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，以及在服务器端解编数据的代码。

  我们必须编译 Printer.cpp 文件，并把它链接进客户和服务器。

编写和编译服务器

服务器的源码不多，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 编译器会为这个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——或者是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，并向对象适配器注册它。

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
```

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
```

运行客户和服务器

要运行客户和服务器，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器，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，以了解与你的平台相关的各种细节。

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——或者是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，并向对象适配器注册它。

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
```

运行客户和服务器

要运行客户和服务器，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(OutgoingConnectionFac
tory.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器，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）的客户和服务器。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. 编写服务器并编译它。
3. 编写客户并编译它。

如果有人已经编写了服务器，而你只是在编写服务器，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器）。

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

Slice[1] （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`）。

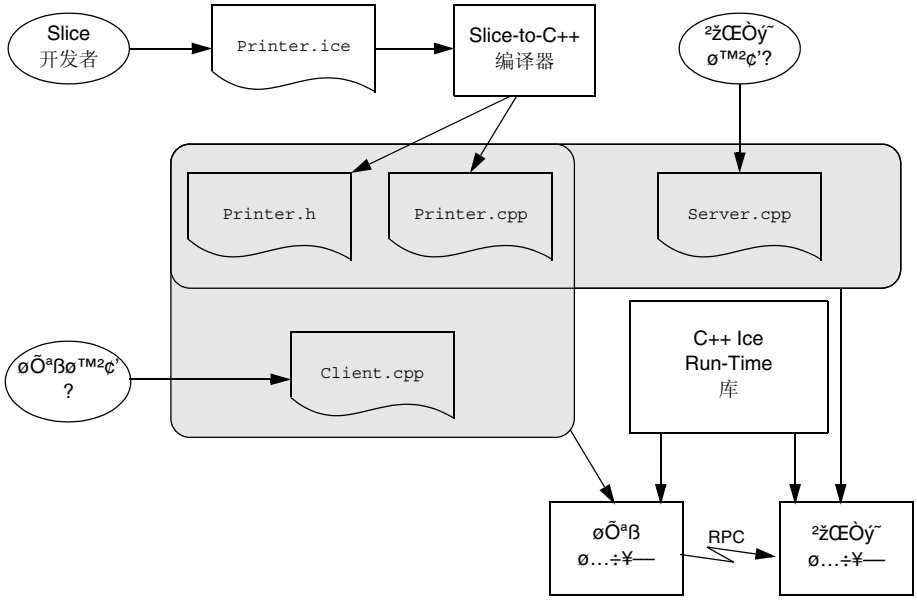

图 4.1. 在客户与服务器共享相同的开发环境时的开发过程

- `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。客户和服务器开发者之间的唯一链接是它们各自使用的 Slice 定义。

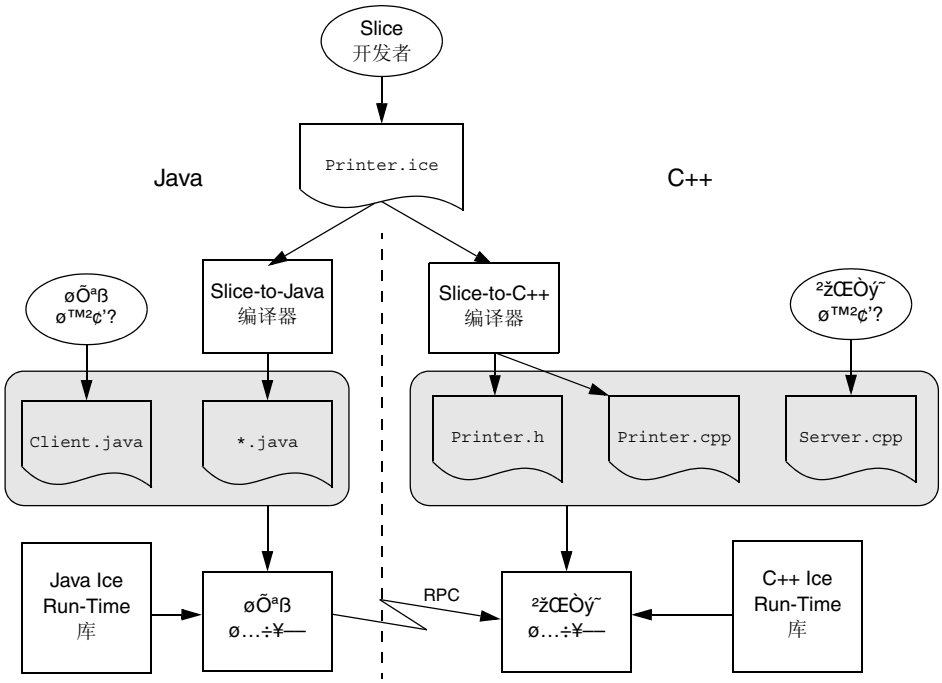

图 4.2. 使用不同的开发环境时的开发过程

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 LocalObject 也是关键字，必须按照所示方式让首字母大写。在附录 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。

表 4.1. Slice 基本类型

¿‡–Õ	²¹Œß	Š?¥Á
`bool`	`false` 或 `true`	未规定
`byte`	-128-127[a]	≥ 8 位
`short`	-2^{15} 到 $2^{15}-1$	≥ 16 位
`int`	-2^{31} 到 $2^{31}-1$	≥ 32 位
`long`	-2^{63} 到 $2^{63}-1$	≥ 64 位
`float`	IEEE 单精度	≥ 32 位
`double`	IEEE 双精度	≥ 64 位
`string`	所有 Unicode 字符，除了 所有位为零的字符	变长

a. ªÚ0-255£¨»°æˆ”/ ”Ô—‘”Š…‰

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，只要你不在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员的结构，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－值对的结构的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－值对的结构的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 Table 4.1 on page 62 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，客户要调用该对象的代理上的操作：要读取当前时间，客户调用 getTime 操作；要设置当前时间，客户调用 setTime 操作，把类型为 TimeOfDay 的参数传给它。

如果你调用代理上的操作，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调用操作；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，你必须调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
```

```
                    TimeOfDay endTime,
                out timeOfDay prevStartTime,
                out TimeOfDay prevEndTime);
};
```

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 getTime，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 TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

nonmutating 关键字说明， getTime 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 会保证操作调用的 “*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调用给服务器，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调用，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调用，其效果与一次调用是一样的，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，如果客户调用 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 RTError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RTError 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器:

```
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，服务器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: 如果服务器从 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器相同的定义构建的，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 FatalApplication-Error 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。.

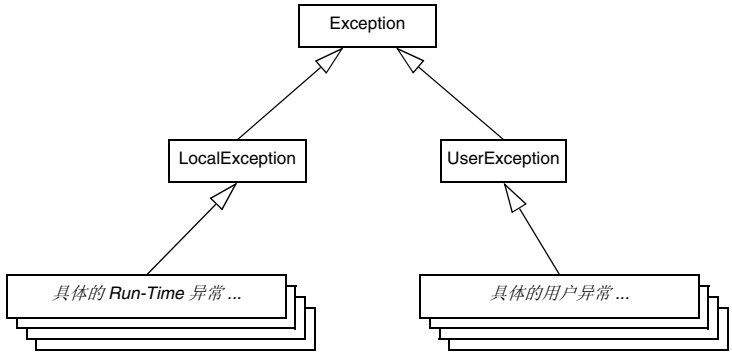

图 4.3. Ice 运行时异常的继承结构

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]。

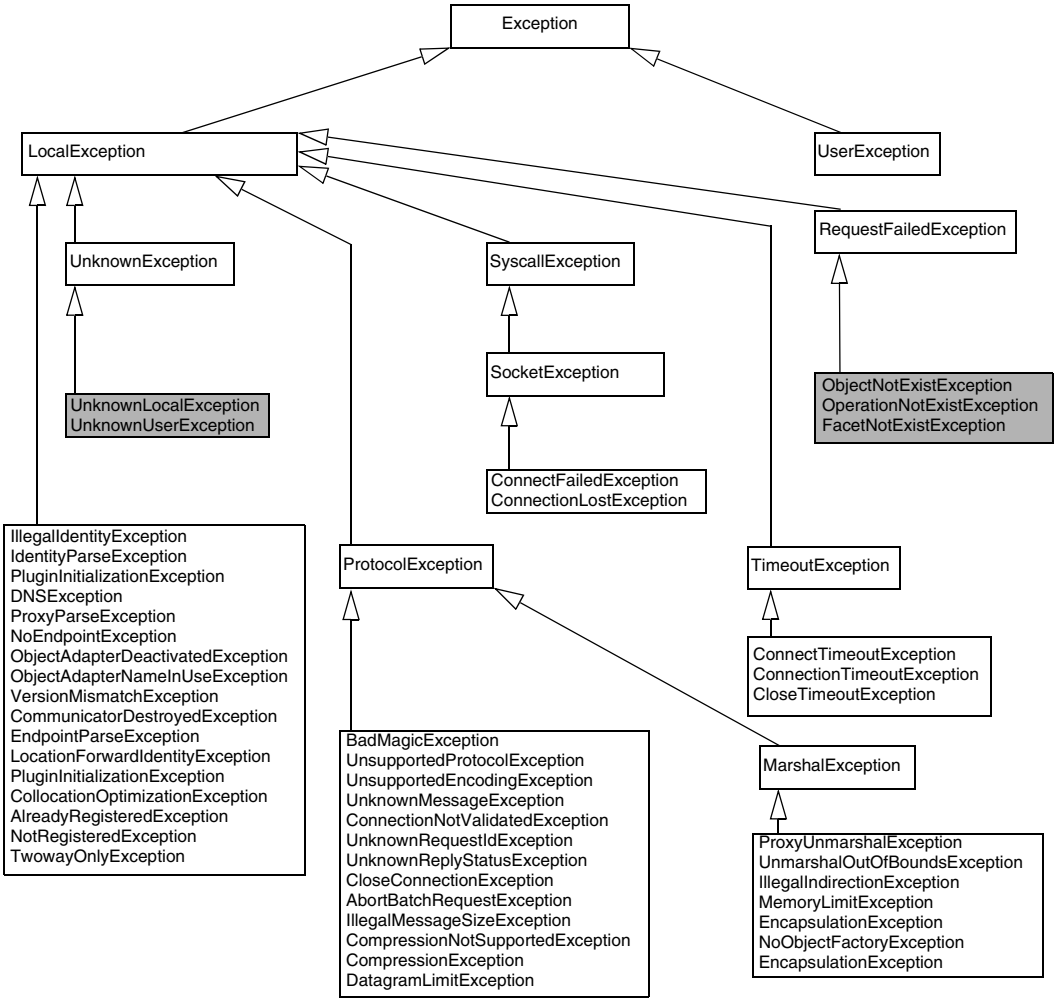

图 4.4. Ice 运行时异常层次 （服务器可能会发送有阴影的异常）

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调用时，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 调用这个操作的地方。 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，调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；如果它们调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”；如果你调用代理上的某个操作，你的调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:

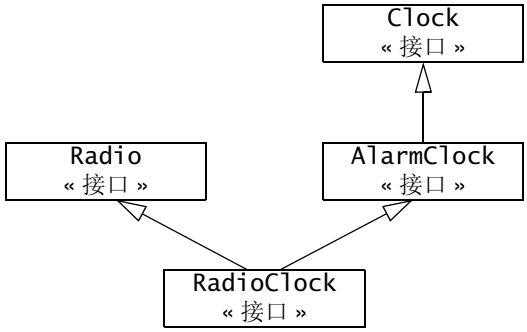

图 4.5. RadioClock 的继承图

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:

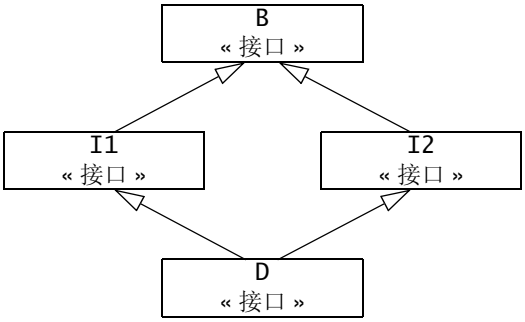

图 4.6. 菱形的继承图

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。

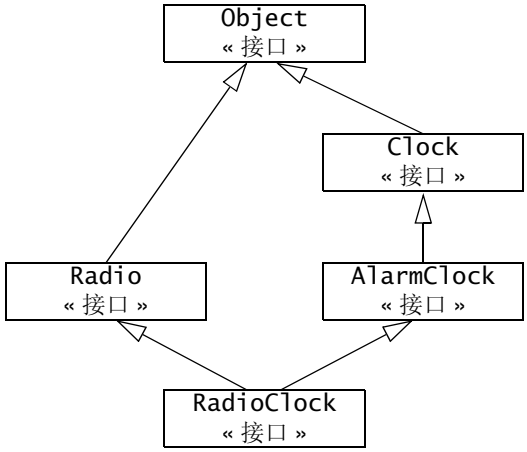

图 4.7. 隐含地继承 Object

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 不可能调用 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，然后调用操作。

   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调用。

  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 and 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*。这很快就会造成碎片化的、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:

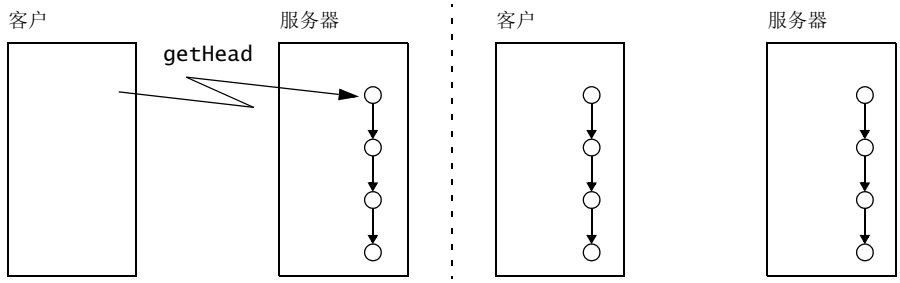

图 4.8. 类版本的 Link 在调用 getHead 之前和之后的情况

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，并且通过调用每个链接上的 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器（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，客户和服务器之间的合约就会由 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 RadioClock implements Time, Radio {
    TimeOfDay time;
    long hertz;
};
```

RadioClock 类既实现 Time 接口，又实现 Radio 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 Clock implements Time {
    TimeOfDay time;
};

interface AlarmClock extends Time {
    nonmutating TimeOfDay getAlarmTime();
    idempotent void setAlarmTime(TimeOfDay alarmTime);
};

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

class RadioAlarmClock extends Clock
                      implements AlarmClock, Radio {
    TimeOfDay alarmTime;
    long hertz;
};
```

这些定义产生了如图 4.9 所示的继承图:

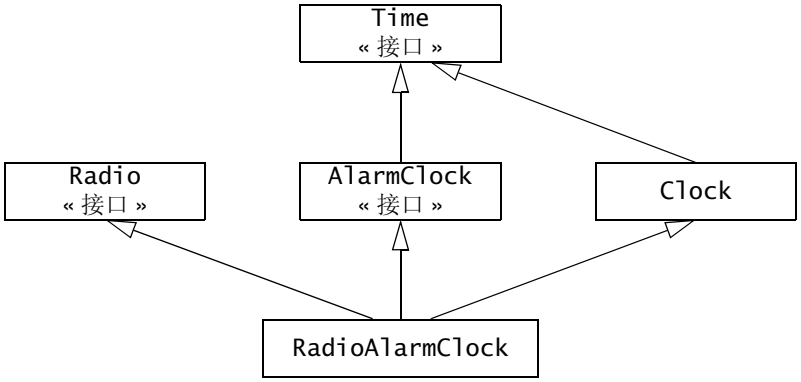

图 4.9. 一个使用实现和接口继承的类

在这个定义中，Radio 和 AlarmClock 是抽象的接口，而 Clock 和 RadioAlarmClock 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调用操作，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调用 format 操作。这个操作的实现由服务器提供，当客户调用 format 时，它发送一条 RPC 消息给服务器，就和它调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 MutableRealms {
    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，所以我们也可以用 `::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 AlarmClock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:

表 4.2. 调用一个代理上的 ice_isA，这个代理代表的是 AlarmClock 类型的对象

Argument	Result
::Ice::Object	true
::Clock	true
::AlarmClock	true
::Radio	false
::RadioClock	false

和我们预期的一样，对于 ::Clock 和 ::AlarmClock， 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 AlarmClock 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、 ::AlarmClock、 ::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器之间传送（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。 .

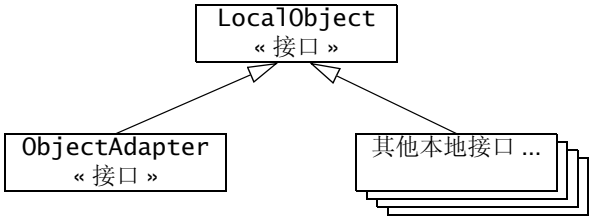

图 4.10. 以 `LocalObject` 为根的继承层次

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:

```
["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， `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 `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*异步方法调用*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:

```
[["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。

- **-D*NAME***[10]

  定义预处理器符号 ***NAME***。

- **-D***NAME*=*DEF*[10]

  定义预处理器符号 ***NAME***，其值为 ***DEF***。

- **-U***NAME*[10]

  解除预处理器符号 ***NAME*** 的定义。

- **-I***DIR*

  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，从远地调用该实例。

  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。异步方法调用（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）。

11.IDL 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，并在客户和服务器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：

- 和普通调用一样，oneway 调用也是通过 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 调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 单向调用，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。对于异步方法调用， 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，客户和服务器之间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－串对（name-string pairs）的序列随每次操作调用一起发送；服务器可以检

查这些名－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－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。这会影响像 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：

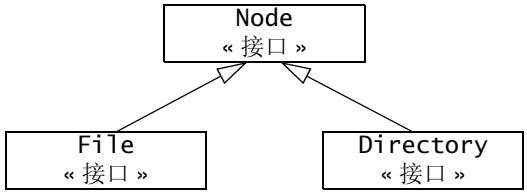

图 5.1. 文件系统的继承图

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调用。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，客户怎样调用操作、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，客户怎样调用操作、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：忽略某个操作调用的返回值，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。

表 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..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会做正确的事情）。

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，可以用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 &) cons
t;
        // ...
    };
    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`

  这既是操作调用超时、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；句柄负责把操作调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，我们通常既会用*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）。如果你调用这样的 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：

```
{                                   // Enter new scope
    SimplePrx s1 = ...;             // Get a proxy from somewhere
    SimplePrx s2(s1);               // Copy-construct s2
    assert(s1 == s2);               // Assertion passes
}                                   // Leave scope; s1, s2, and the
                                    // underlying proxy instance
                                    // are deallocated
```

注意这个例子里的断言：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。要调用某个操作，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，客户可以这样调用操作：

```
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，而后者又把操作调用发给服务器、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：调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。下面是我们在第 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，如果客户和服务器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。

链式调用

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调用：

```
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

任何操作调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调用放在 try-catch 块中：

3. 这值得一提，因为在 CORBA，同样的代码会泄漏内存（就和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，你只需针对操作调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。

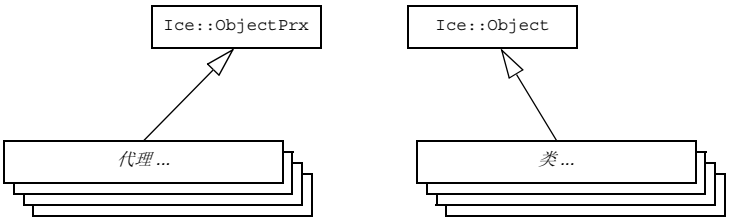

图 6.1. 从 Ice::ObjectPrx 和 Ice::Object 继承

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 TimeOfDaye 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调用的 out 参数。要想知道为什么，考虑下面的简单接口：

```
interface Time {
    TimeOfDay get();
};
```

当客户调用 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；并且会在工厂解除注册、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，它就会调用已注册的 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。当工厂仍然注册在 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 两次注册工厂：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注册过的工厂， 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注册对象工厂。只有当类有操作时，你才需要定义并注册对象工厂。

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。

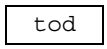

图 6.2. 新初始化的智能指针

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。

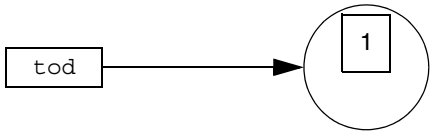

图 6.3. 已初始化的智能指针

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:

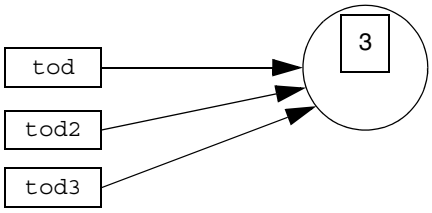

图 6.4. 指向同一类实例的三个智能指针

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:

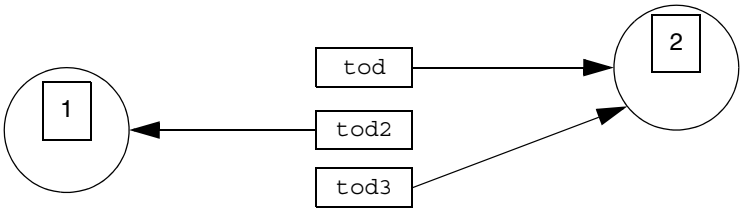

图 6.5. 三个智能指针和两个实例

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：

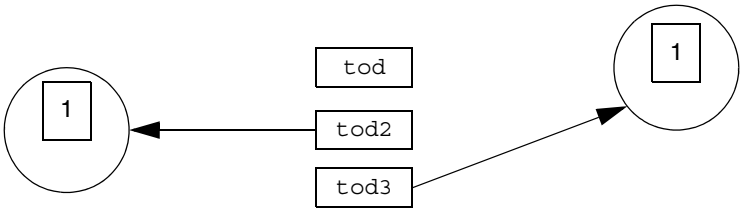

图 6.6. 在使智能指针清零后引用计数减少了

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。

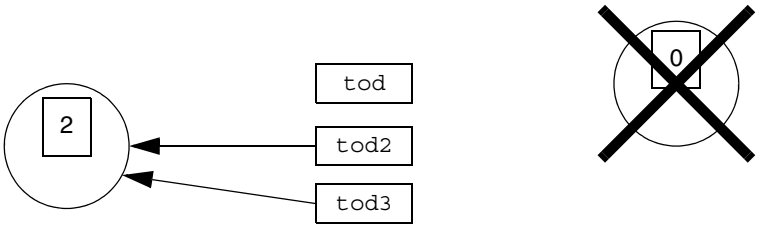

图 6.7. 释放引用计数为零的实例

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，你必须实现并调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。

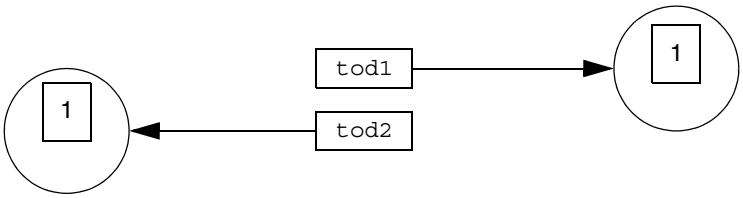

图 6.8. 通过 clone 对类实例进行深度复制

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
```

在栈上分配的类实例

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：

```
{                       // Enter scope
    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：

```
{                               // Enter scope
    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：

```
{                               // Enter scope
    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
    NodePrt 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
    NodePrt 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部方便，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（附录 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。对于文件系统中的每一个节点，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`

      客户和服务器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：

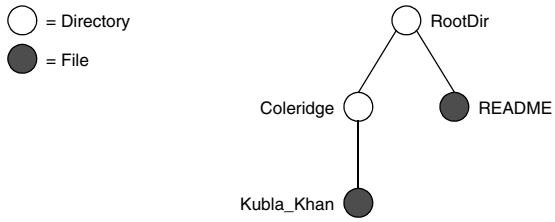

图 7.1. 一个小文件系统

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，客户怎样调用操作、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，客户怎样调用操作、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 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

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。

表 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己定制的实现。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 Meal2，缺省映射被改换了，所以 Meal2::b 被映射到 java.util.LinkedList，而 Meal2::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`

  这既是操作调用超时、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`。如果客户调用已不存在的 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 “远地的服务器中的 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，你很可能会在调用代理上的操作时，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，你最后就会把调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 unchecedCast 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。要调用某个操作，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，客户可以这样调用操作：

```
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：调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*却*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(stch);

System.out.writeln(ih.value);   // Show one of the values
```

这段代码同样没有什么让人惊讶之处：一旦操作调用完成，值就会出现在各个 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器之间传递映射表，你必须要保证，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

任何操作调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调用放在 try-catch 块中：

```
ChildPrx child = ...;   // Get child proxy...

try {
    child.askToCleanUp();
} catch (Tantrum t) {
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

在典型情况下，你只需针对操作调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调用的输出参数。要想知道为什么，考虑下面的简单接口：

```
interface Time {
    TimeOfDay get();
};
```

当客户调用 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；并且会在工厂解除注册、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，它就会调用已注册的 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。当工厂仍然注册在 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 两次注册工厂：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注册过的工厂， 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注册对象工厂。只有当类有操作时，你才需要定义并注册对象工厂。

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，让它把根据这个 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。对于文件系统中的每一个节点，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:

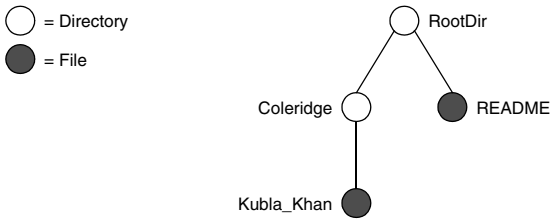

图 9.1. 一个小文件系统

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 注册对象。

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 注册它们。

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调用完成。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 被调用过一样，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 客户和服务器来说非常方便。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、注册 servants。应用的 main 函数必须实例化这个子类，调用其 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. 调用 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. 调用 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. 调用 waitForShutdown 成员函数，这个函数会阻塞到 shutdown 被调用。
7. 调用 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。缺省实现调用 Ice::initialize，并把指定的参数向量传给它。

- `void interrupt()`

  信号处理器调用它来表示收到了信号。缺省实现调用 `shutdown` 成员函数。

- `bool shutdown()`

  让服务器开始关闭的过程。缺省实现调用通信器的 `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，以及注册 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。缺省实现会调用通信器的 `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。在子进程成功调用 `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，因为它确保了命令调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调用 **`--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` 会调用 `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 `UnknownLocalException`。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 `UnknownLocalException`[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 `UnknownLocalException`，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 `UnknownLocalException`：`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调用。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，如果这时我们想要调用 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调用 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，你要调用对象适配器的 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。当客户调用操作时，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、或者当作操作调用的 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，而客户调用了代理上的操作，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 注册 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。

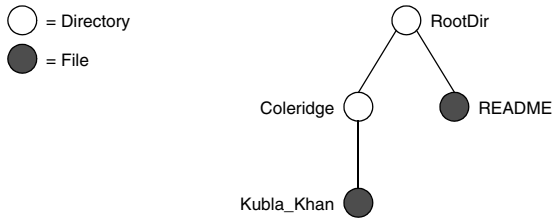

图 11.1. 一个小文件系统

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字，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：服务器代码调用它自己的 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。

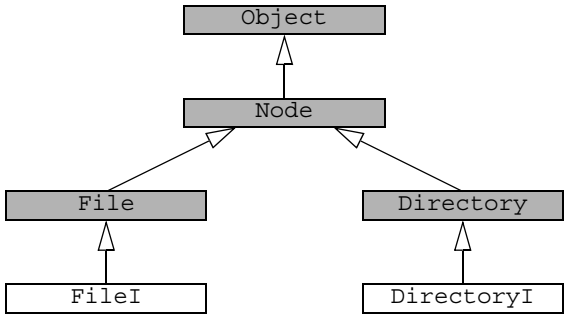

图 11.2. 使用接口继承的文件系统 servants

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。

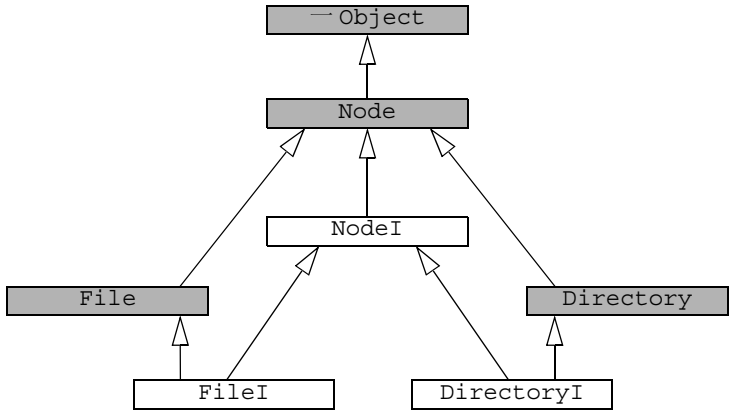

图 11.3. 使用实现继承的文件系统

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 注册对象。

12.2 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 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 注册它们。

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调用完成。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);
    }
}
```

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 允许应用注册*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 会注册一个关闭挂钩，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 被调用过一样，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，你能够以一各种方式配置 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 UnknownLocalException。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 MemoryLimitException[2]，客户会收到 UnknownLocalException。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 UnknownLocalException，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 UnknownLocalException：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）。

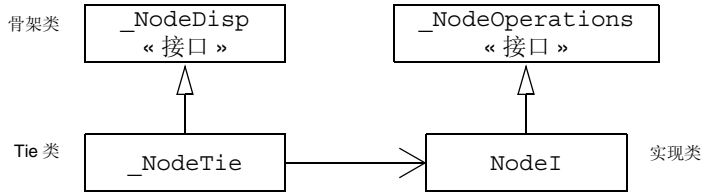

图 12.1. 一个骨架类、tie 类及实现类

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，每个操作调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调用。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 。

要调用对象适配器的 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。当客户调用操作时，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、注册具有该标识的 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、或者当作操作调用的 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，而客户调用了代理上的操作，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 注册 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。

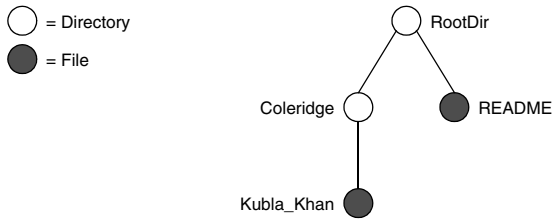

图 13.1. 一个小文件系统

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字，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：服务器代码调用它自己的 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`。

在附录 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都它进行特殊处理）。

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，假定我们调用一个服务器:

```
$ ./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 getPropertiesForPrefix(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。

- getPropertiesForPrefix

  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，或进行相反的转换：

```
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 communcator.
//
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 communcator.
//
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调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调用，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（参见附录 C）。这个属性的缺省值是 1。

多线程意味着，来自客户的多个调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 的并发调用，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。不要在同一个线程中多次调用它。

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；一旦队列长度超过阀值，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阀值以下。

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）。

表 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。

表 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。如果你想要实现非阻塞式的 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：如果在你从 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。信号由用户实现并注册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。

第三部分

高级 Ice

第 16 章
Ice Run Time 详解

16.1 引言

现在，我们已经知道了客户和服务器的基本实现方式，该更详细地看一看 Ice run time 了。这一章将针对同步、单向，以及数据报调用，详细介绍 Ice 的服务器端 API（我们将在第 17 章涵盖异步接口）。

16.2 节将描述与 Ice 通信器（Ice run time 的主句柄）相关联的功能。，16.3 节到 16.5 节描述对象适配器及它们在调用分派方面扮演的角色，并且说明代理、 Ice 对象、 servant，以及对象标识之间的关系。 16.6 节描述 servant 定位器，这是 Ice 的一种重要机制，用于控制性能和内存消耗之间的平衡。 16.7 节描述最为常用的服务器实现技术。我们建议你详细阅读这一节，因为了解这些技术，对于高性能和高伸缩性系统的构建而言是至关紧要的。 16.8 节描述从客户到服务器的隐式参数传送， 16.9 节讨论调用超时。 16.10 到 16.13 描述单向、数据报，以及成批调用， 16.14 到 16.16 讨论日志、统计信息收集，以及位置透明性。最后， 16.17 节对服务器端的 Ice run time 和对应的 CORBA 做法进行了对比。

16.2 通信器

Ice run time 的主进入点由本地接口 `Ice::Communicator` 表示。`Ice::Communicator` 的一个实例与一些运行时资源是关联在一起的:

- 客户端线程池

  客户端线程池会确保，在客户端，至少有一个线程可用于接收对未完成请求的答复。这确保了不会发生死锁。例如，如果服务器在操作实现中回调客户，即使客户正在等待同一服务器答复它的请求，客户端接收者线程也能处理来自服务器的请求。

  客户端线程池还被用于异步方法调用（AMI），用以避免在回调中发生死锁（参见第 17 章）。

- 服务器端线程池

  这个线程池里面的线程负责接受到来的连接，并处理来自客户的请求。

- 配置属性

  Ice run time 的各个方面可以通过属性进行配置。每个通信器都有自己的配置属性集（参见第 14 章）。

- 对象工厂

  为了实例化从已知基类派生的类，通信器维护有一组对象工厂，能够替 Ice run time 对类进行实例化（参见 6.14.4 节和 8.14.1 节）。

- 一个日志记录器对象

  日志记录器对象实现了 `Ice::Logger` 接口，并负责确定 Ice run time 产生的日志消息的处理方式（参见 16.14 节）。

- 一个统计对象

  统计对象实现了 `Ice::Stats` 接口，通信器会把通信流量（发送和接收字节数）告知该对象（参见 16.15 节）。

- 一个缺省路由器

  路由器实现了 `Ice::Router` 接口。 Glacier（参见第 24 章）使用了路由器来实现 Ice 的防火墙功能功能。

- 一个缺省定位器

  定位器是用于把对象标识解析为代理的对象。定位器对象被用于构建定位服务，比如 IcePack（参见第 20 章）。

- 一个插件管理器

  插件是用于给通信器增加特性的对象。例如， IceSSL（参见第 23 章）被实现成插件。每个通信器都有一个插件管理器，这个管理器实现 `Ice::PluginManager` 接口，通过它，你可以访问通信器的插件集。

- 对象适配器

  对象适配器负责分派到来的请求，并把每个请求传给正确的servant。

使用不同通信器的对象适配器和对象完全是相互独立的。特别地:

- 每个通信器都使用自己的线程池。例如，这意味着，如果一个通信器耗尽了用于处理到来请求的线程，只有使用该通信器的对象会受影响。使用其他通信器的对象有自己的线程池，因而不受影响。
- 跨越不同通信器的并置调用不会被优化，而使用同一通信器的并置调用则能避免产生大部分调用分派开销。

服务器通常只使用一个通信器，但有时多个通信器也会很有用。例如，（参见第 25 章）为它加载的每个 Ice 服务使用了单独的通信器，以确保不同的服务不会相互干扰。多个通信器器还能用于避免线程饥饿：如果一个服务耗尽线程，其他的服务不会受影响。

通信器的接口是用 Slice 定义的。下面是其部分接口:

```
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();
        // ...
    };
    // ...
};
```

通信器提供了一些操作:

- proxyToString
- stringToProxy

  这两个操作允许你把代理转换成串化表示，或进行反向转换。

- createObjectAdapter
- createObjectAdapterWithEndpoints

  这两个操作创建新的对象适配器。每个对象适配器都与一个或更多传输端点关联在一起。一个对象适配器通常拥有一个传输端点。但是，一个对象适配器也可以提供多个传输端点。如果是这样，这些端点可以

通往同一组对象，但代表访问这些对象的不同手段。这很有用，例如，服务器在防火墙后面，但必须让内部和外部的客户都能访问它；把适配器同时绑定到内部和外部接口，就可以其中任何一个接口访问在服务器中实现的对象了。

createObjectAdapter 根据配置信息（参见第 20 章）来确定把自己绑定到哪个端点，而 createObjectAdapterWithEndpoints 允许你为新适配器指定传输端点。你通常会优先于 createObjectAdapterWithEndpoints 使用 createObjectAdapter。这样，就能把与传输机制相关的信息（比如主机名和端口号）放在 源码外面，通过改变属性，你可以对应用进行重配置（于是，在传输端点需要改变时，不用重新进行编译）。

- shutdown

  这个操作关闭服务器端的 Ice run time:

  - 在 shutdown 被调用时，仍处在执行过程中的操作调用可以正常完成。shutdown *不会*等待这些操作完成；在 shutdown 返回时，你所知道的是：不会再有新的请求被分派，但在你调用 shutdown 时已经在执行之中的操作可能仍在运行。你可以调用 waitForShutdown，等待仍在执行的操作完成。
  - 在服务器调用 shutdown 之后到达的操作调用或者会失败（抛出 ConnectFailedException），或者会被透明地重定向到服务器的某个新实例（参见第 20 章）。

- waitForShutdown

  这个操作挂起发出调用的线程，直到通信器关闭为止（也就是说，直到在服务器中不再有操作在执行为止）。这样，你可以在销毁通信器之前，等待服务器空闲下来。

- destroy

  这个操作销毁通信器及其相关资源，比如线程、通信端点，以及内存资源。一旦你销毁了通信器（因此也就销毁了该通信器的 run time），你不能再调用其他任何 Ice 操作（除了创建另外的通信器）。

  在离开程序的 main 函数之前要调用 destroy，这是强制性的。不这样做会导致不确定的行为。

  在离开 main 之前调用 destroy 是必需的，因为 destroy 会在返回之前等待所有运行中的线程终止。如果你没有调用 destroy 就离开 main，你就会留下许多仍在运行的线程；许多线程包不允许你这样做，你最终会使程序崩溃。

如果你没有调用 shutdown 就调用 destroy，在这个调用返回之前，它会等待所有执行中的操作调用完成（也就是说， destroy 的实现隐含地调用 shutdown，然后调用 waitForShutdown）。 shutdown （因此，也包括 destroy）会解除与通信器相关联的所有对象适配器的激活。

在客户端，如果你在操作执行过程中调用 shutdown，这些操作会终止，抛出 CommunicatorDestroyedException。

16.3 对象适配器

一个通信器含有一个或更多对象适配器。对象适配器处在 Ice run time 和服务器之间的界线上，负有这样一些责任:

- 它把Ice对象映射到到来请求的servant，并把请求分派给每个servant中的应用代码（也就是说，对象适配器实现了一个向上调用接口，把 Ice run time 与服务器中的应用代码连接在一起）。
- 它协助进行生命周期操作，使得Ice对象和servant在创建和销毁时不会出现竞争状况。
- 它提供一个或更多传输端点。客户通过这些端点访问适配器所提供的 Ice 对象（创建没有端点的对象适配器也是可能的。这样的适配器被用于双向回调——参见第 24 章）。

每个对象适配器都有一个或更多 servant，对 Ice 对象进行体现；同时还有一个或更多传输端点。如果对象适配器拥有的传输端点不止一个，所有向该适配器作了注册的 servant 可以在任何一个端点上响应到来的请求。换句话说，如果对象适配器有多个传输端点，这些端点代表的是通往同一组对象的不同通信路径（例如，通过不同的传输机制）。

每个对象适配器都只属于一个通信器（但一个通信器可以有多个对象适配器）。

每个对象适配器可以非强制性地拥有自己的线程池，你可以通过 <adapter-name>.ThreadPool.Size 属性来使用这个特性（参见附录 C）。如果使用了这个特性，在分派针对该适配器的客户调用时，使用的是适配器的线程池中的线程，而不会使用通信器的服务器线程池中的线程。

16.3.1 活动 Servant 映射表

每个对象适配器都维护有一个叫作*活动 servant 映射表*（active servant map）的数据结构。活动 servant 映射表（简称为 *ASM*）是一个查找表，用

于把对象标识映射到 servant：在 C++ 里，查找值是智能指针，指向对应的 servant 在内存中的位置；在 Java 里，查找值是指向 servant 的 Java 引用。当客户把操作调用发给服务器时，请求的目标是特定的传输端点。传输端点隐含地标识了请求所针对的对象适配器（因为同一个端点只能绑定到一个对象适配器）。客户藉以发送请求的代理含有对应的对象的标识，客户端 run time 会在线路上随调用一起发送这个对象标识。对象适配器继而使用这个对象标识、在它的 ASM 中查找正确的 servant，把调用分派给它。如图 16.1 所示。

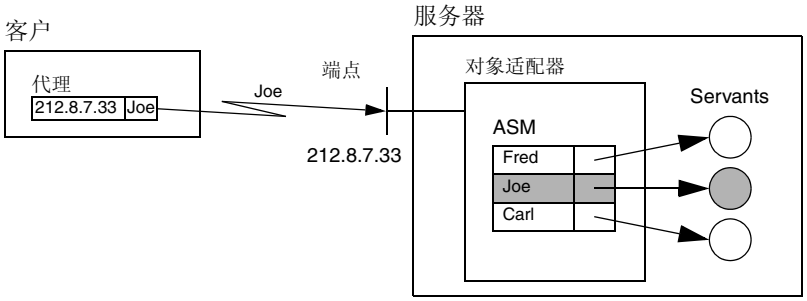

图 16.1. 把请求绑定到正确的 servant

经由代理、把请求关联到正确的 servant 的过程叫作*绑定*。图 16.1 的例子所描述的是直接绑定：传输端点是嵌在代理中的。Ice 还支持间接绑定模式：正确的传输端点由 IcePack 服务提供（详情参见第 20 章）。

如果在适配器的 ASM 中，客户请求所包含的对象标识没有对应的条目，适配器就会把 `ObjectNotExistException` 返回给客户（除非你使用了 servant 定位器——参见 16.6 节）。

16.3.2 Servant

在 2.2.2 节中提到， servant 是 Ice 对象的物理体现，也就是说，它们是用具体的编程语言实现的实体，并且在服务器的地址空间中进行实例化。Servants 为“客户发送的操作调用”提供服务器端行为。

同一个 servant 可以向一个或更多对象适配器注册。

16.3.3 对象适配器接口

对象适配器是本地接口：

```
module Ice {
    local interface ObjectAdapter {
        string getName();

        Communicator getCommunicator();

        // ...
    };
};
```

这两个操作的行为是:

- getName 操作返回适配器的名字，这个名字是在调用 Communicator::createObjectAdapter 或 Communicator::createObjectAdapterWithEndpoints 时传入的。
- getCommunicator 操作返回先前用于创建该适配器的通信器。

注意，在 ObjectAdapter 接口中还有其他操作；我们将在本章余下的各个部分讨论这些操作。

16.3.4 Servant 激活与解除激活

术语 *servant 激活*指的是，向 Ice run time 告知某个 Ice 对象的 servant 的存在。如果你激活一个 servant，在图 16.1 所示的活动 servant 映射表中就会增加一个条目。看待 servant 激活的另一种方式是，把它看成是在 “Ice 对象的标识”与 “对应的用编程语言编写的、负责为该 Ice 对象处理请求的 servant”之间创建链接。一旦 Ice run time 知道了这个链接，它就可以把到来的请求分派给正确的 servant。如果没有这样的链接，也就是说，在 ASM 中没有对应的条目，针对该标识的到来请求就会引发 ObjectNotExistException。一旦 servant 被激活，我们就认为它*体现了*对应的 Ice 对象。

与之相反的操作叫作 *servant 解除激活*。如果你解除某个 servant 的激活，就会从 ASM 中移除针对特定标识的一个条目。自此以后，针对该标识的到来请求就不会再分派给 servant，并且会引发 ObjectNotExistException。

对象适配器提供了一些操作，用于管理 servant 激活和解除激活:

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

        Object* add(Object servant, Identity id);
        Object* addWithUUID(Object servant);
```

```
        void remove(Identity id);

        // ...
    };
};
```

这些操作的行为是:

- add

  add操作把一个具有指定标识的servant增加到ASM中。一旦add被调用，请求就会分派给这个servant。返回值是这个servant所体现的Ice对象的代理。这个代理嵌有传给add的标识。

  你不能用同一标识多次调用add：如果你试图在ASM中增加一个已经存在的标识，就会引发AlreadyRegisteredException（增加两个具有相同标识的servant没有意义，因为这会带来不明确：不知道应该由哪一个servant处理针对该标识的到来请求）。

  注意，通过不同的标识多次激活同一个servant，这是可能的。在这种情况下，同一个servant将体现多个Ice对象。我们将在16.7.2节更详细地探索这个特性的影响。

- addWithUUID

  addWithUUID操作的行为和add操作一样，但你不需要为servant提供标识。addWithUUID会生成一个UUID（参见[14]），作为对应的Ice对象的标识。你可以调用返回的代理的ice_getIdentity操作，取得生成的标识。addWithUUID可用于创建临时对象的标识，比如短期存在的对象（你还可以把addWithUUID用于没有自然标识的持久对象，就像我们为文件系统应用所做的那样）。

- remove

  remove操作中断标识与其servant之间的关联，从ASM中移除对应的条目。一旦servant解除了激活，新到来的针对已移除标识的请求会引发ObjectNotExistException。当remove被调用时，正在servant中执行的请求可以正常完成。一旦servant的最后一个请求完成，对象适配器就会丢弃它的指向servant的引用（在C++里是智能指针）。这时，如果你不再持有指向这个servant的引用或智能指针，它就可以被当作垃圾收集了（在C++里是被销毁）。实际效果就是，一旦解除了激活的servant空闲下来，它就会被销毁。如果你解除对象适配器的激活，将会隐含地调用它的活动servants的remove操作（参见16.3.5节）。

16.3.5 适配器状态

对象适配器具有以下处理状态（processing state）:

- 扣留状态（holding）

  在这种状态下，适配器的任何到来的请求都会被“扣留”，也就是说，不会被分派给 servant。

  在使用 TCP/IP（及其他面向流的协议）当被调用时，如果适配器处在扣留状态，服务器端 run time 就会停止从对应的传输端点读取数据。此外，它也不接受客户发送的连接请求。这意味着，如果客户向处在扣留状态的适配器发送请求，客户最终会收到 `TimeoutException` 或 `ConnectTimeoutException`（除非在定时器超时之前，适配器变成了活动状态）。

  在使用 UDP 时，如果客户请求到达处在扣留状态的适配器，请求将被丢弃。

  刚创建的适配器都处在扣留状态。这意味着，在你让适配器变成活动状态之前，请求不会被分派。

- 活动状态

  在这种状态下，适配器会接受到来的请求，把它们分派给 servant。新创建的适配器一开始处在扣留状态。只要你让适配器变成活动状态，它就会开始分派请求。

  你可以随意在活动和扣留状态之间转换。

- 不活动

  在这种状态下，适配器在概念上已经被销毁（或处在销毁过程中）。解除适配器的激活会销毁与该适配器相关联的所有传输端点。当适配器变成不活动状态时，仍在执行的请求可以完成，但适配器不会再接受新的请求（发出新请求的客户会收到异常）。一旦适配器解除了激活，你不能再在同一进程中重新激活它。

`ObjectAdapter` 接口提供了一些操作，允许你改变适配器状态，或者等待状态改变完成:

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

        void activate();
        void hold();
        void waitForHold();
        void deactivate();
```

```
        void waitForDeactivate();

        // ...
    };
};
```

下面是这些操作的行为:

- activate

  activate 操作让适配器变成活动状态。激活已经处在活动状态的适配器没有任何效果。一旦 activate 被调用， Ice run time 就会开始把请求分派给适配器的 servants。

- hold

  hold 操作让适配器变成扣留状态。在调用 hold 之后到达的请求都会像第 373 页所说的那样被扣留。在 hold 被调用时正在执行的请求可以正常完成。注意， hold 会立即返回，不会等待正在执行的请求完成。

- waitForHold

  waitForHold 操作挂起发出调用的线程，直到适配器迁移到扣留状态为止，也就是说，直到正在执行的所有请求完成为止。你可以从多个线程调用 waitForHold，也可以在适配器处在活动状态时调用 waitForHold。如果适配器处在不活动状态，对它调用 waitForHold 不会做任何事情，并且会立即返回。

- deactivate

  deactivate 操作让适配器变成不活动状态。在调用 deactivate 之后到达的请求会被拒绝，但正在执行的请求可以完成。一旦适配器处在不活动状态，它不能再被重新激活。如果适配器处在不活动状态，对它调用 activate、 hold、 waitForHold，或 deactivate 没有任何效果。一旦变得空闲，与该适配器相关联的任何 servants 都会被销毁。注意，deactivate 会立即返回，不会等待正在执行的请求完成。

- waitForDeactivate

  waitForDeactivate 操作挂起发出调用的线程，直到适配器迁移到不活动状态为止，也就是说，直到所有正在执行的线程完成、所有的传输端点关闭、所有相关联的 servants 销毁为止。你可以从多个线程调用 waitForDeactivate，也可以在适配器处在活动或扣留状态时调用 waitForDeactivate。如果适配器处在不活动状态，对它调用 wait-ForDeactivate 不会做任何事情，并且会立即返回。

让适配器变成扣留状态是一种有用的操作，例如这样的情况：你要改变服务器的状态，需要服务器 （或一组 servants）空闲下来。例如，你可以

把你的 servants 的实现放进动态库中，不必关闭服务器，就可以在运行时加载更新的库版本来使实现升级。

与此类似，如果遇到下述情况，等待适配器迁移到不活动状态也很有用：你的服务器需要进行一些最后的清理工作，在所有正在执行的请求完成之前，你不能去做这些工作。

16.4 对象标识

每个 Ice 对象都有一个对象标识，其定义如下所示：

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
};
```

你可以看到，对象标识由一对串组成：name 和 category。name 或 category 都可以是空串（如果某个代理包含的标识的 name 和 category 都是空的，Ice 就把这个代理解释成 null 代理）。完整的对象标识由 name 和 category 组合而成，也就是说，两个标识要相等，它们的 name 和 category 都必须相同。category 成员通常是空串，除非你在使用 servant 定位器（参见 16.6 节）[1]。

对象标识可以用串来表示；category 部分先出现，然后是 name；两部分之间用 / 字符分隔，例如：

```
Factory/File
```

在这个例子里，Factory 是 category，File 是 name。如果 name 或 category 成员自身含有 / 字符，其串化表示可以用 \ 来使 / 字符转义，例如：

```
Factories\/Factory/Node\/File
```

在这个例子里，category 成员是 Factories/Factory，name 成员是 Node/File。

在串化标识中还有相当一些字符必须转义。为了使标识与串的相互转换更容易，Ice run time 提供了一些实用函数，用于标识编码和解码。

在 C++ 里，这些实用函数的定义是：

1. Glacier（参见第 24 章）还把 category 成员用于过滤。

```
namespace Ice {
    std::string   identityToString(const Ice::Identity id);
    Ice::Identity stringToIdentity(const std::string& s);
};
```

在 Java 里，这些实用程序位于 Ice.Util 类中，其定义是:

```
package Ice;

public final class Util {
    public static String   identityToString(Identity id);
    public static Identity stringToIdentity(String s);
}
```

这些函数能够正确地对那些可能会造成问题的字符进行编码和解码（比如控制字符或空格）。

在第 372 页提到过，对象适配器的 ASM 中的每个条目都必须是唯一的：你不能把两个具有相同标识的 servants 增加到 ASM 中。

16.5 Ice::Current 对象

迄今为止，我们不声不响地忽略了传给服务器端的每个骨架操作的最后一个参数（类型是 Ice::Current）。Current 对象的定义是:

```
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;
    };
};
```

注意，通过 Current 对象，你可以访问“正在执行的请求”和“服务器中的操作的实现”等信息:

- adapter

  通过 adapter 成员，你可以访问负责分派当前请求的对象适配器。通过适配器，你又可以访问它的通信器（通过 getCommunicator 操作）。
- id

  id 成员提供当前请求的对象标识（参见 16.4 节）。
- facet

  通过 facet 成员，你可以访问请求的 facet（参见 XREF）。
- operation

  operation 成员含有正在被调用的操作的名字。注意，这个操作名可能是 Ice::Object 上的操作的名字，比如 ice_ping 或 ice_isA。
- mode

  mode 成员含有操作的调用模式（Normal、Idempotent，或 Nonmutating）。
- ctx

  ctx 成员含有这个调用的当前上下文（参见 16.8 节）。

如果你在实现服务器时，让它为每个 Ice 对象实现一个单独的 servant，Current 对象的内容就并不怎么有意思（你很可能会访问 Current 对象，读取 adapter 成员，并且（举个例子）通过它来激活 servant，或解除其激活）。但是，我们将在 16.6 节看到，更为完备（而且可伸缩性更好）的 servant 实现必须使用 Current 对象。

16.6 Servant 定位器

使用适配器的 ASM 来把 Ice 对象映射到 servants，在设计上有这样一些影响：

- 每个 Ice 对象都由一个不同的 servant 代表[2]。
- 所有 Ice 对象的所有 servants 都永久性地处在内存中。

对于许多服务器而言，以这样的方式、为每个 Ice 对象使用一个单独的 servant 是很常见的做法：这种技术实现起来很简单，而且能够很自然地从 Ice 对象映射到 servants。服务器在启动时，通常会为每个 Ice 对象实例化一

2. 你可以通过多个标识注册同一个 servant。但是，这样做意义不大，因为使用缺省 servant（参见 16.7.2 节）就能获得同样的效果。

个单独的 servant，激活每个 servant，然后调用对象适配器的 `activate`，启动对请求流（the flow of requests）的处理。

只要能满足两个条件，这样的设计没有什么问题：

1. 服务器有足够的内存可用，足以为每个 Ice 对象实例化一个单独的 servant，并一直保留这些 servants。
2. 在启动时初始化所有 servants 所需的时间是可以接受的。

对于许多服务器而言，这两个条件都不会带来什么问题：只要 servants 的数量足够少，同时， servants 能够很快初始化，这样的设计完全可以接受。但是，这种设计的可伸缩性不好：服务器的内存需求会随 Ice 对象数目的增大呈线性增长，所以，如果对象太多（或者每个 servant 存储了太多状态），服务器就会耗尽内存。

Ice 提供了 *servant 定位器*，你可以用它来让服务器支持大量对象。

16.6.1 综述

简言之， servant 定位器是一种本地对象，你负责实现它，并把它系到对象适配器上。适配器一旦有了 servant 定位器，它会和平常一样查询 ASM，对 servant 进行定位。如果在 ASM 中能找到与该请求对应的 servant，请求就会分派给这个 servant。但是，如果在 ASM 没有与该请求的对象标识对应的条目，对象适配器就会回调 servant 定位器，让它为该请求提供 servant。 servant 定位器会做下面两件事情中的一件：

- 初始化一个 servant，把它传给 Ice run time，在这种情况下，请求会被分派给新实例化的 servant。
- servant 定位器告诉 Ice run time，它没有能找到 servant，在这种情况下，客户会收到 `ObjectNotExistException`。

采用这种简单的机制，我们的服务器能够让我们访问数量不限的 Ice 对象：服务器不必为每一个现有的 Ice 对象实例化一个单独的 servant，而是可以只实例化一部分 Ice 对象，也就是客户实际使用的那些对象。

提供数据库访问的服务器常常使用 servant 定位器：数据库中的条目数通常要远远大于服务器能够存放在内存中的条目数。采用 servant 定位器，服务器可以只为客户实际使用的 Ice 对象实例化 servants。

用于进程控制或网络管理的服务器也常常会使用 servant 定位器：在这样的情况下，并没有使用数据库，但可能会有非常多的设备或网络元件，必须通过服务器进行控制。除此而外，这种情况与大数据的情况是一样的：Ice 对象的数量超过了服务器能够存放在内存中的 servants 的数量，因此，我们需要采用某种途径，让被实例化的 servants 的数量少于 Ice 对象的数量。

16.6.2 Servant 定位器接口

下面是 servant 定位器的接口:

```
module Ice {
    local interface ServantLocator {
        Object locate(    Current     curr,
                      out LocalObject cookie);

        void finished(    Current     curr,
                          Object      servant,
                          LocalObject cookie);

        void deactivate();
    };
};
```

注意， ServantLocator 是一个本地接口。为了创建实际的 servant 定位器实现，你必须定义一个从 Ice::ServantLocator 派生的类，并实现 locate、 finished，以及 deactivate 操作。 Ice run time 会这样调用你的派生类上的操作:

- locate

  只要有请求到达，而且它在 ASM 中没有对应的条目， Ice run time 就会调用 locate。 locate 的实现（由你在派生类中提供）应该返回一个能够处理该请求的 servant。你的 locate 实现可以有以下三种行为方式:

  1. 为当前请求实例化并返回一个 servant。在这种情况下， Ice run time 会把请求分派给新实例化的 servant。
  2. 返回 null。在这种情况下， Ice run time 会在客户端引发 ObjectNotExistException。
  3. 抛出一个异常。在这种情况下， Ice run time 会把所抛出的异常传播回客户（要记住，除了 ObjectNotExistException、OperationNotExistException，以及 FacetNotExistException，所有的异常都被当作 UnknownLocalException 报告给客户）。

  通过 locate 的 cookie out 参数，你可以把一个本地对象传给对象适配器。对象适配器并不在乎这个对象的内容（返回 null cookie 也是合法的）。相反，当 Ice run time 调用 finished 时，会把你从 locate 返回的 cookie 传回给你。这样，你可以把任意数量的状态从 locate 传到对应的 finished 调用。

- finished

  如果 locate 调用返回了一个 servant 给 Ice run time， Ice run time 会把到来的请求分派给这个 servant。一旦请求完成（也就是说，被调用的操作完成了）， Ice run time 就会调用 finished，把完成了操作的 servant、该请求的 Current 对象，以及 locate 在一开始创建的 cookie 传给它。这意味着，对 locate 的每一个调用都会与一个 对 finished 的调用相配对（前提条件是 locate 确实返回了 servant）。

  你不能在 finished 里抛出异常——如果你这样做了， Ice run time 会在日志里记录一条错误消息，并且（在使用面向连接的传输机制的情况下）关闭与客户的连接。

- deactivate

  当 servant 定位器所属的对象适配器解除激活时， Ice run time 会调用 deactivate 操作。这样， servant 定位器就有机会进行清理活动（例如，定位器可以关闭数据库连接）。

意识到这一点很重要：Ice run time 并不会 “记住”通过特定的 locate 调用得到的 servant。相反， Ice run time 会简单地把到来的请求分派给 locate 返回的 servant，而且，一旦请求完成，就调用 finished。特别地，如果针对同一个 servant 的两个请求几乎同时到达， Ice run time 会分别为每个请求调用一次 locate 和 finished。换句话说， locate 负责建立对象标识与 servant 之间的关联；该关联只对单个请求有效， Ice run time 不会用它去分派另外的请求。

16.6.3 针对 Servant 定位器的线程保证

Ice run time 保证，涉及到 servant 定位器的每一个操作调用都会配对地调用 locate 和 finished，也就是说，每一次对 locate 的调用，都会有对 finished 的调用与之配对（当然，前提是对 locate 的调用确实返回了 servant）。

此外， Ice run time 还保证，调用 locate、操作，以及 finished 的都是同一个线程。这个保证很重要，因为这样的话，你就能用 locate 和 finished，围绕操作调用实现线程专有的前、后处理（例如，你可以在 locate 中启动一个事务，在 finished 中提交或回滚它；你也可以在 locate 中获取某个锁，在 finished 中释放这个锁[3]）。

3. 事务和锁通常都是线程专有的，也就是说，只有启动事务的线程能够提交或回滚它，只有获取了某个锁的线程能够释放这个锁。

注意，如果你在使用异步方法分派（参见第 17 章），启动某个调用的线程不一定是完成它的线程。在这种情况下，调用 finished 的线程就是执行操作实现的线程，这个线程可能不是调用 locate 的线程。

Ice run time 还保证，当你解除 servant 定位器所属的对象适配器的激活时，deactivate 会被调用。只有在与 servant 定位器有关的所有操作都结束之后，deactivate 才会被调用，也就是说，deactivate 保证不会与 locate 或 finished 同时运行，而且保证是针对 servant 定位器发出的*最后一个*调用。

除了这些保证，Ice run time 不再为 servant 定位器提供任何线程保证。特别地:

- 对 locate 的多个调用可能会同时进行（针对同一个对象标识，或针对不同的对象标识）。
- 对 finished 的多个调用可能会同时进行（针对同一个对象标识，或针对不同的对象标识）。
- 对 locate 和 finished 的调用可能会同时进行（针对同一个对象标识，或针对不同的对象标识）。

通过这些语义，你可以使你的应用代码获得最大限度的并行性（因为在不需要进行序列化时，Ice run time 不会对调用进行序列化）。当然，这意味着，在必要时，你必须使用互斥原语、在 locate 和 finished 中对共享数据进行保护。

16.6.4 Servant 定位器注册

对象适配器不会自动了解到你创建了一个 servant 定位器。相反，你必须显式地向对象适配器注册 servant 定位器:

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

        void addServantLocator(ServantLocator locator,
                               string         category);

        ServantLocator findServantLocator(string category);

        // ...
    };
};
```

你可以看到，对象适配器允许你增加和查找 servant 定位器。注意，在你注册 servant 定位器时，你必须为 category 提供实参。category 参数的值决定这个 servant 定位器要负责的对象标识：只有那些 category 成员（参见第 375 页）与该定位器相匹配的对象标识才会触发对应的 locate 调用。如果到来的请求在 ASM 中没有明确的对应条目，而且也没有 servant 定位器的范畴与之对应，客户就会收到 ObjectNotExistException。

addServantLocator 具有以下语义：

- 针对特定的范畴，你只能注册一个 servant 定位器。如果你针对同一个范畴、多次调用 addServantLocator，就会引发 AlreadyRegisteredException。
- 你可以针对不同的范畴注册不同的 servant 定位器，也可以多次注册同一个 servant 定位器（每次都针对不同的范畴）。在前一种情况下，在 Ice run time 调用的 servant 定位器实例中，范畴是隐含的；在后一种情况下，locate 的实现可以检查传给它的 Current 对象的对象标识成员，找出到来的请求所针对的范畴。
- 针对空范畴注册 servant 定位器是合法的。这样的 servant 定位器叫作*缺省 servant 定位器*：如果到来的请求在 ASM 中没有对应的条目存在，而且其范畴也与任何其他已注册的 servant 定位器的范畴不匹配，Ice run time 就会调用缺省 servant 定位器的 locate。

注意，一旦作了注册，你就无法再改变或移除针对某个范畴的 servant 定位器。对象适配器的 servant 定位器的生命周期最后会和适配器一样：当对象适配器解除激活时，它的 servant 定位器也会如此。

findServantLocator 操作允许你取回针对特定范畴（包括空范畴）的 servant 定位器。如果没有针对该范畴的 servant 定位器，findServantLocator 会返回 null。

调用分派语义

前面的规则可能会显得很复杂，所以下面将总结一下，Ice run time 在为到来的请求定位 servant 时，要采取哪些动作。

每个到来的请求都隐含地标识了一个具体的对象适配器，用于处理该请求（因为请求是在具体的传输端点到达的，因此，标识了特定的对象适配器）。到来的请求携带有一个必须能映射到某个 servant 的对象标识。为了定位 servant，Ice run time 会按照给出的顺序、采取以下步骤：

1. 在 ASM 中查找该标识。如果 ASM 有这样一个条目，就把请求分派到对应的 servant。
2. 如果到来的对象标识的范畴不是空的，就查找针对该范畴注册的 servant 定位器。如果有这样的 servant 定位器，就调用这个定位器的 locate，

如果 locate 返回一个 servant，就把请求分派到该 servant，然后调用 finished；而如果 locate 调用返回 null，就在客户端引发 ObjectNotExistException。

3. 如果到来的对象标识的范畴是空的，或者在第 2 步找不到针对该范畴的 servant 定位器，就去查找缺省的 servant 定位器（也就是，针对空范畴注册的 servant 定位器）。如果有缺省的 servant 定位器，就像第 2 步那样分派请求。
4. 在客户端 ObjectNotExistException。

在头脑里记住这些调用分派语义，这很重要，因为针对它们，可以采用一些强大的实现技术。每种技术都允许你使服务器实现流水线化，并精确地控制性能、内存消耗，以及可伸缩性之间的平衡。为了说明各种可能性，我们将在下一小节里概述一些最为常用的实现技术。

16.6.5 实现一个简单的 Servant 定位器

为了说明我们在前面概述的概念，让我们考察一个（非常简单的）servant 定位器实现。设想一下，我们想要为整个世界创建一个电话簿（显然，这涉及到的条目数量非常多，肯定多得不能整个放在内存里）。实际的电话簿放在一个大型数据库里。我们还假定，我们有一个搜索操作，负责返回一个电话簿条目的详细信息。这个应用的 Slice 定义可能像这样:

```
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);
    // ...
};
```

应用的详细情况并不重要；请注意一个要点：每个电话本条目都由一个接口表示，最终我们需要为一个接口创建一个 servant，但我们不能把所有条目的 servant 都永久性地放在内存中，我们负担不起。

电话数据库中的每个条目都有一个唯一标识符。取决于数据库的构造方式，这个标识符可以是内部数据库标识符，也可以是各个字段值的组合。其要点是，我们可以使用这个数据库标识符，把 Ice 对象的代理与它的持久状态链接在一起：我们可以就用数据库标识符做对象标识。这意味着，每个代理都含有一个主访问键，用于定位每个 Ice 对象的持久状态，继而为这个 Ice 对象实例化 servant。

接下来我们将看到用 C++ 编写的一个简略的实现。我们的 servant 定位器的类定义是：

```
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);
};
```

注意，`MyServantLocator` 从 `Ice::ServantLocator` 继承，并实现了 **`slice2cpp`** 编译器为 `Ice::ServantLocator` 接口生成的一些纯虚函数。当然，和往常一样，你可以增加额外的成员函数，比如构造器和析构器，你也可以增加 private 数据成员，用以支持你的实现。

使用 C++，你可以这样实现 `locate` 成员函数：

```
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);
}
```

目前， finished 和 deactivate 的实现是空的，什么也不做。

在前面的例子中，我们假定 DB_lookup 要访问数据库。如果查找失败（可能是因为没有匹配的记录）， DB_lookup 会抛出 DB_error 异常。这段代码会捕捉该异常，继而抛出 Ice::ObjectNotExistException；这个异常会被返回给客户，说明客户使用的代理指向的 Ice 对象已不再存在。

注意， locate 在堆上实例化 servant，并把它返回给 Ice run time。这引发了这样一个问题：servant 何时被销毁？回答是，只要有必要， Ice run time 就会保留 servant，也就是说，足以在返回的 servant 上调用操作，并在操作完成时调用 finished。此后， servant 就不再需要了， Ice run time 会销毁 locate 所返回的智能指针。然后，因为没有其他智能指针指向这个 servant， PhoneEntryI 实例的析构器就会被调用， servant 就会被销毁。

这种设计的结果是，对于每一个到来的请求，我们都实例化一个 servant，并让 Ice run time 在请求完成时销毁 servant。取决于你的应用，这可能正好是你所需要的，也可能会昂贵得惊人——我们很快将考察一些设计，避免为每一个请求创建和销毁一个 servant。

使用 Java，我们的 servant 定位器的实现非常相似：

```
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)
    {
    }
}
```

注意，MyServantLocator 类扩展了 Ice.LocalObjectImpl。LocalObjectImpl 由 Ice run time 提供，含有所有本地对象共有的方法的实现（即 equals、clone，以及 ice_hash）。

关于 Java 映射的一个警告：如果你想要在 locate 和 finished 的作用域之外，保留 Ice.Current 的某个成员的副本，或 Ice.Current 自身的副本，你必须制作这些值的副本，而不能只是存储它们的引用：出于效率上的考虑，Java Ice run time 没有为一个 Ice.Current 对象的每个成员分配单独的对象，而是会把一个 Current 对象池复用于不同的调用，并在进行调用分派之前，对 Current 对象的成员进行赋值。这意味，如果你存储一个引用，指向的是某个 Current 对象内部的标识，如果 Ice run time 复用这个 Current 对象时，它的值会出乎你意料地发生变化[4]。

locate 的所有实现都遵循前面的伪码所阐述的模式：

1. 要获取对象标识，使用传入的 Current 对象的 id 成员。通常，要获取 servant 的状态，只使用标识的 name 成员。category 成员通常用于选择 servant 定位器（我们很快就会考察 category 成员的用法）。
2. 用对象标识做键，从辅助存储器（或网络）上取回 Ice 对象的状态。

4. 就这方面而言，Ice.Current 对象很独特：对于其他所有参数，存储一个指向某个参数的引用，并在当前调用的作用域之外继续保留，是一种安全的做法。

- 如果查找成功，你已经取回了 Ice 对象的状态。
- 如果查找失败，抛出 ObjectNotExistException。在这种情况下，与客户请求对应的 Ice 对象确实不存在，可能是因为已经被删掉了，而客户却仍然拥有一个代理，指向这个已经被删掉的对象。

3. 实例化一个 servant，用从数据库取回的状态初始化这个 servant（在这个例子中，我们会把取回的状态传给 servant 的构造器）。
4. 返回 servant。

当然，在使用 servant 定位器之前，我们必须先把定位器的存在告知适配器，然后再激活适配器，例如（在 Java 里）：

```
MyServantLocator sl = new MyServantLocator();
adapter.addServantLocator(sl, "");
```

注意，在这个例子中，我们是在针对空范畴安装 servant 定位器。这意味着，针对我们的任何 Ice 对象的调用，都会调用我们的 servant 定位器的 locate（因为空范畴是缺省范畴）。实际上，我们的这种设计没有使用对象标识的 category 成员。只要我们的 servant 具有相同的单一接口，这就没有问题。但是，如果我们需要在同一个服务器中支持若干不同的接口，这种简单的策略就不再够用了。

16.6.6 使用对象标识的 category 成员

前面的小节里的简单例子总是实例化类型为 PhoneEntryI 的 servant。换句话说， servant 定位器总是隐含地知道到来的请求所针对的 servant 的类型。对于大多数服务器而言，这并非是一个很现实的假定，因为在服务器中的对象，通常具有若干不同的类型。这就给我们的 locate 实现带来了一个问题：在 locate 中，我们必须以某种方式决定要实例化何种类型的 servant。要解决这个问题，你有几种选择：

- 为每种接口类型使用单独的对象适配器，同时为每个对象适配器使用单独的 servant 定位器。

  这种技术工作良好，但也有不利的方面：每个对象适配器都要有单独的传输端点，这是一种很浪费的做法。

- 把一种类型标识符合成到对象标识的 name 部分。

  这种技术使用对象标识的一部分来表示要实例化的对象的类型。例如，在我们的文件系统应用中，我们有目录和文件对象。我们可以约定，在每个目录的标识前面加上 ‘d’，在每个文件的前面加上 ‘f’。随后， servant 定位器可以用标识的第一个字母来决定要实例化何种类型的 servant:

```
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__);
    }
}
```

尽管这是一种可行的做法，但也是一种笨办法：我们不仅需要解析 name 成员，找出要实例化的对象的类型，我们还需要在每次给应用增加新类型时，修改 locate 的实现。

- 使用对象标识的 category 成员来表示要实例化的 servant 的类型。

  这是我们推荐的做法：对于每一种接口类型，我们都会指定一个单独的标识符，作为对象标识的 category 成员的值（例如，我们可以用‘d’表示目录，用‘f’表示文件）。除了注册唯一一种 servant 定位器，我们还创建两种不同的 servant 定位器实现，一种用于目录，一种用于文件，然后针对适当的范畴注册每种定位器：

```
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");
```

另一种选择也是使用对象标识的 category 成员，但却使用唯一的一个缺省 servant 定位器（也就是说，一个针对空范畴的定位器）。采用这种做法，所有的调用都会送往这唯一的一个缺省 servant 定位器，你可以在

locate 操作的实现的内部针对范畴值执行 switch 语句，确定要实例化的 servant 的类型。但与前面的做法相比，这种做法更难以维护； Ice 对象标识的 category 成员是专门用来支持 servant 定位器的，所以你最好也按照这样的意图使用它。

16.6.7 使用 Cookies

有时，你也需要在 locate 和 finished 之间传递信息。例如， locate 的实现可以根据负载或可用性，从多种数据库后端中选取一种；同时，为了适当执行结束工作， finish 的实现可能需要知道 locate 使用的是哪一种数据库。为了支持这样的情况，你可以在你的 locate 实现中创建一个 cookie ；在操作调用完成之后， Ice run time 会把这个 cookie 的值传给 finished。cookie 必须从 Ice::LocalObject 派生，可以容纳任何对你的实现有用的状态和成员函数:

```
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 服务器实现技术

我们在第 378 页提到，在服务器启动时为每个 Ice 对象实例化一个 servant，是一种可行的设计，前提是你能够负担这些 servant 所需的内存数量，并承受服务器启动时的延迟。但是，在 Ice 中，你可以通过其他一些方式、更灵活地把 Ice 对象映射到 servant；使用这些映射，你可以精确地控制内存消耗、可伸缩性，以及性能之间的平衡。在这一节，我们将概述一些较为常用的实现技术。

16.7.1 渐进的初始化

如果你使用 servant 定位器，`locate` 返回的 servant 只能用于当前请求，也就是说，Ice run time 不会把这个 servant 增加到 Active Servant Map 中。当然，这意味着，如果有其他针对同一个 Ice 对象的请求到来，`locate` 必须再次取回对象状态，并实例化一个 servant。一种常用的实现技术是，在 `locate` 中，把每个 servant 增加到 ASM 中。这意味着，只有对 Ice 对象的初次请求会触发对 `locate` 的调用；自此以后，与 Ice 对象对应的 servant 就可以在 ASM 中找到，Ice run time 不必调用 servant 定位器，就可以立即分派针对同一个 Ice 对象的到来请求。

采用这种做法的一种 `locate` 实现可能像是这样：

```
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;
}
```

这和第 384 页上的实现几乎是一样的——唯一的不同是，我们还调用 `ObjectAdapter::add`，把 servant 增加到 ASM 中。遗憾的是，这个实现是错误的，因为在它里面有竞争状态。考虑这样的情况：在 ASM 中，没有针对某一特定 Ice 对象的 servant，有两个客户几乎同时针对这个 Ice 对象发出请求。线程调度器完成有可能这样做：它调度两个到来的请求，使得 Ice run time 在 ASM 中为这两个请求进行查找，并且都得出在内存中没有对应的 servant 存在的结论。结果，针对同一个 Ice 对象，`locate` 会被调用两次，我们的 servant 定位器会实例化两个、而不是一个 servant。因为对 `ObjectAdapter::add` 的第二次调用会引发 `AlreadyRegisteredException`，只有其中一个 servant 会增加到 ASM 中。

当然，这大概不是我们期望的行为。为了避免出现竞争状态，我们的 `locate` 实现必须检查某个同时发生的调用是否已经为到来的请求实例化了一个 servant，如果是这样，就返回这个 servant，而不是实例化一个新的。Ice run time 提供了 `ObjectAdapter::identityToServant` 操作，可用于测试 ASM 中是否存在针对特定标识的条目：

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

        Object identityToServant(Identity id);

        // ...
    };
};
```

如果 servant 已经存在于 ASM 中，identityToServant 返回这个 servant，否则返回 null。使用这个查找函数，再加上一个互斥体，我们就可以正确地实现 locate 了。我们的 servant 定位器现在有了一个 private 互斥体，用于在 locate 中建立临界区：

```
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;
};
```

locate 成员函数锁住这个互斥体，测试某个 servant 是否已存在于 ASM 中：如果是这样，就返回这个 servant；否则，实例化一个新的 servant，并像前面一样把它增加到 ASM 中：

```
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;
}
```

这个定位器的 Java 版本几乎是一样的，但我们使用了 synchronized 限定符、而不是互斥体，来使 locate 成为临界区：

```
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;
}
```

使用一个把 servant 增加到 ASM 中的定位器有一些优点：

- Servants 是随需实例化的，所以 servant 的初始化代价分散到了许多次调用中，而不是在服务器启动时同时产生。
- 服务器的内存需求降低了，因为只有当 Ice 对象被客户实际访问时，servant 才会实例化。如果客户只访问全部 Ice 对象中的一部分对象，内存就可能会得到相当的节省。

一般而言，如果在启动时实例化 servant 太慢，渐进地初始化是有益的。内存方面节省也值得我们使用这种初始化方式，但通常只对运行时间相对较短的服务器才有效：对于长期运行的服务器，每个 Ice 对象都可能迟早会被某个客户访问到；在这种情况下，就不会节省内存，因为无论如何，我们最终都会为每一个 Ice 对象实例化一个 servant。

16.7.2 缺省 Servants

面向对象中间件的一个常见问题是可伸缩性：客户要从远地访问大型数据库，服务器常被用作这些数据库的前端。服务器的任务是，就数据库中的大量记录，向客户给出一个面向对象的视图。记录的数量通常会太大，因此，即使只是部分数据库记录，我们也无法为它们实例化 servant。

解决这个问题的一种常见技术是使用*缺省 servants*。缺省 servant 是这样一种 servant，它针对每个请求，充当不同的 Ice 对象的角色。换句话说，缺省 servant 会在处理每个请求时，根据请求所访问的对象标识改变其行为。通过这样的方式，客户可以访问数量不限的 Ice 对象，但却只有一个 servant 在内存里。

缺省 servant 实现之所以有吸引力，不仅是因为它们能够节省内存，还因为这种实现很简单：在本质上，缺省 servant 就是数据库中的对象的持久状态的 facade[2] 。这意味着，要实现缺省 servant，所需的编程量很小：它只需包含读取对应数据库记录的代码就可以了。

要创建缺省 servant 实现，我们需要的定位器的数量和系统中的非抽象接口的数量一样多。例如，就我们的文件系统应用而言，我们需要两个定位器：一个用于目录，一个用于文件。此外，我们创建的对象标识使用它的 `category` 成员、对对应的 Ice 对象的接口类型进行编码。范畴域的值可以是任何能标识该接口的东西，比如我们在第 388 页使用的 ‘d’ 和 ‘f’ 。你还可以使用 `"Directory"` 和 `"File"`，或使用对应接口的类型 ID，比如 `"::Filesystem::Directory"` 和 `"::Filesystem::File"`。不管我们把对象标识的 `name` 成员设置成什么，我们都必须要能用它来从辅助存储器中取回每个目录和文件的持久状态 （在我们的文件系统应用中，我们使用 UUID 做唯一标识）。

这两个 servant 定位器会从它们各自的 `locate` 实现中返回一个单体 servant。下面是用于目录的 servant 定位器：

```
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;
};
```

注意，定位器的构造器实例化一个 servant，并在每次 locate 被调用请求时返回这同一个 servant。用于文件的定位器的实现与用于目录的类似。servant 定位器的注册和往常一样:

```
_adapter->addServantLocator(new DirectoryLocator, "d");
_adapter->addServantLocator(new FileLocator, "f");
```

所有的动作都发生在操作的实现中，针对每个操作使用以下步骤:

1. 使用传入的 Current 对象获得当前请求的标识。
2. 使用标识的 name 成员，在辅助存储器上定位 servant 的持久状态。如果找不到针对该标识的记录，抛出 ObjectNotExistException。
3. 在取回的状态上执行操作（返回该状态，或根据该操作的情况更新状态）。

用伪码来表示，上面的步骤可能像是这样:

```
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;
}
```

注意，这个 servant 实现完全是无状态的：它所处理的唯一状态是当前请求所针对的 Ice 对象的标识（这个标识是作为 Current 参数的一部分传入的）。我们为无限制的可伸缩性、以及更低的内存占有所付出的代价是性能：对于每一个被调用的操作，缺省 servant 都要进行一次数据库访问，这显然比在内存中缓存状态（作为已增加进 ASM 的 servant 的一部分）要慢。但是，这并非意味着缺省 servant 造成的性能降低是不可接受的：数据库常常会提供成熟的缓存机制，所以即使操作实现要读写数据库，只要它们是在访问缓存中的状态，性能可能就完全可以接受。

重新定义 ice_ping

关于缺省 servant，你需要注意的一个问题是，你需要重新定义 ice_ping：servant 从它的骨架类继承的 ice_ping 的缺省实现总是会成功。对于向 ASM 作了注册的 servant，这种行为正是我们所要的；但对于缺省 servant 而言，如果客户使用的代理指向的是不再存在的 Ice 对象，ice_ping 就必须失败——为了避免产生前一种行为，你必须在缺省 servant 中重新定义 ice_ping。你的实现必须检查，该请求的对象标识代表的是否是仍然存在的 Ice 对象，如果不是，就抛出 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__);
    }
}
```

如果你使用缺省 servant，重新定义 ice_ping 是一种好做法。

16.7.3 混合途径及缓存

取决于你的应用的本质，你也许能采取一种中间道路，在提供更好的性能的同时，保持低内存需求：如果你的应用有一些经常访问的对象，是性能的关键，你可以把针对这些对象的 servant 增加到 ASM 中。如果你在 servant 内部的数据成员中存储这些对象的状态，你实际上就是缓存了这些对象。

其他不常被访问的对象可以通过缺省 servant 实现。例如，在我们的文件系统实现中，我们可以选择永久性地实例化目录 servants，但却通过缺省

servant 实现文件对象。这样，你可以高效地在目录树上移动，只在访问文件时（这种情况可能不那么频繁），速度才会减慢。

你可以用“最近访问过的文件”的缓存来增强这种技术，就像 UNIX 内核 [10] 所用的缓冲池。要点在于，你可以把 ASM 与 servant 定位器及缺省 servant 结合起来，精确地控制可伸缩性、内存消耗，以及性能之间的平衡，从而满足你的应用的需要。

16.7.4 Servant 逐出器

逐出器（evictor）[4] 是前面的方案的一种变种，也是 servant 定位器的一种特别有趣的用法。逐出器是维护有 servants 缓存的 servant 定位器：

- 只要有请求到达（也就是说，Ice run time 调用了 `locate`），逐出器就在它的缓存中检查，看是否能找到可用于该请求的 servant。如果是，它就返回这个已经在缓存中实例化的 servant；否则，它实例化一个 servant，并把它增加到缓存中。
- 缓存是一个按照“最近最少使用”（LRU）顺序维护的缓存：最近最少使用的 servant 处在队列的尾部，最近使用最多的 servant 处在队列的头部。如果某个 servant 被从缓存中返回，或增加到缓存中，它就会从当前的队列位置移到队列头部，也就是说，“最新”的 servant 总是在头部，“最老”的 servant 总是在尾部。
- 队列的长度可以配置，并决定会有多少 servant 存放在缓存中；如果针对某个 Ice 对象的请求在内存中没有对应的 servant，且缓存满了，逐出器就会在队尾移除最近最少使用的 servant，给要在队头实例化的 servant 腾出空间。

图 16.2 说明的是一个缓存尺寸为五的逐出器在进行了五次调用之后的情况，所针对的对象标识是从 1 到 5。

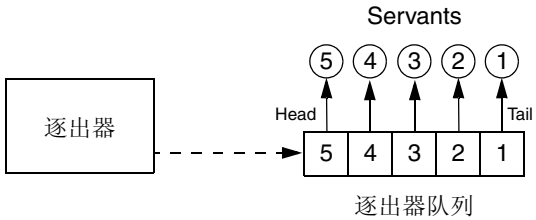

图 16.2. 一个进行了五次调用之后的逐出器，所针对的对象标识是从 1 到 5

这时，逐出器已经实例化了五个 servants，并把每个 servant 放进了逐出器队列。因为客户发出的请求所针对的对象标识是从 1 到 5，servant 5 最后

会处在队列的头部（在最近使用最多的位置），而 servant 1 最后处在队列的尾部（在最近使用最少的位置）。

假定客户现在发出了一个针对 servant 3 的请求。在这种情况下， servant 可以在逐出器队列中找到，并会被移到头部。所产生的顺序如图 16.3 所示。

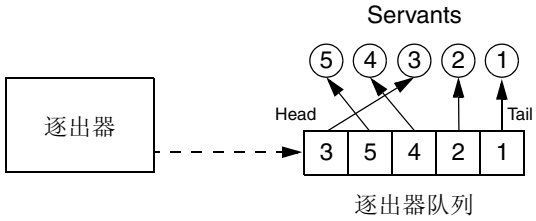

图 16.3. 图 16.2 的逐出器在访问了 servant 3 之后的情况

假定下一个客户请求是针对对象标识 6 的。逐出器队列已经全满了，所以逐出器会为对象标识 6 创建一个 servant，把这个 servant 放在队列的头部，并在队列的尾部逐出标识为 1 的 servant（最近最少使用的 servant）。如图 16.4 所示。

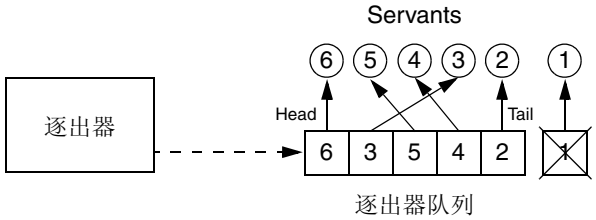

图 16.4. 图 16.3 的逐出器在逐出了 servant 1 之后的情况

逐出器模式结合了 ASM 和缺省 servant 的优点：只要缓存尺寸足以在内存中容纳 servant 工作集，大多数请求都会由已经实例化的 servant 提供服务，而不用创建 servant、并访问数据库来初始化 servant 状态。你可以根据你的应用的情况，通过设置缓存尺寸，控制性能和内存消耗之间的平衡。下面的部分将向你说明，怎样用 C++ 和 Java 实现逐出器（在 Ice 发布包中，你可以找到逐出器的源码，以及本书的代码示例）。

创建 C++ 逐出器实现

我们在这里给出的逐出器被设计成一个抽象基类：要想使用它，你要从 EvictorBase 基类派生一个对象，并实现两个方法，逐出器会在需要增加或逐出 servant 时调用这两个方法。于是产生了这样的类定义：

```
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;
```

注意，这个逐出器的构造器会设置队列尺寸，缺省参数把尺寸设置成 1000。

locate、finished，以及 deactivate 函数派生自 ServantLocator 基类；这些函数实现的逻辑负责以 LRU 顺序维护队列，并按照需要增加和逐出 servant。

逐出器会在需要把新 servant 增加到队列、或从队列中逐出 servant 时调用 add 和 evict 函数。注意，这两个函数都是纯虚函数，所以在派生类中必须实现它们。 add 的任务是，实例化并初始化 servant，供逐出器使用。逐出器会在需要逐出 servant 时调用 evict 函数。这样， evict 可以进行任何清理活动。注意， add 可以返回一个 cookie，由逐出器传给 evict，所以你可以把上下文信息从 add 传到 evict。

接下来，我们需要考虑我们的逐出器实现所需的数据结构。我们需要两个主要的数据结构：

1. 一个映射表，把对象标识映射到 servant，这样我们就能高效地确定、在内存中是否有 servant 可用于到来的请求。

2. 一个列表，实现逐出器队列。这个列表总是保持 LRU 顺序。

逐出器映射表不仅要存储 servant，还要记录一些管理信息：

1. 映射表存储 `add` 返回的 cookie，这样我们才能把这个 cookie 传给 `evict`。
2. 映射表存储一个用于逐出器队列的迭代器，标记这个 servant 在队列中的位置。存储队列位置并非绝对必要——我们之所以存储这个位置，是出于效率上的考虑，因为这样我们在恒定的时间内就能确定 servant 的位置，而不必为了维护队列的 LRU 属性、对整个队列进行搜索。
3. 映射表存储一个使用计数，每当有操作被分派到某个 servant 时，这个计数就会增大，每当操作完成时，这个计数就会减小。

我们应该额外解释一下，为何需要使用计数：假定有一个 Ice 对象，标识是 *I*，客户要调用其上的一个长期运行的操作。作为响应，逐出器把 *I* 的一个 servant 增加到逐出器队列中。在这个调用执行的同时，其他客户也调用多个 Ice 对象上的操作，从而使得针对其他对象标识的更多 servant 也被增加到队列中。结果，标识 *I* 的 servant 会逐渐向队列尾部移动。如果对象 *I* 上的操作仍在执行，同时又有足够的针对其他 Ice 对象的客户请求到来，*I* 的 servant 就可能会被逐出，而同时它又仍在执行原来的请求。

就这种情况自身而言，并没有什么危害。但如果在这个 servant 被逐出后，又有一个客户向对象 *I* 发出请求，逐出器并不知道 *I* 有一个 servant 仍然存在，就会为 *I* 增加第二个 servant。同一个 Ice 对象有两个 servant，很可能会带来问题，特别是在 servant 的操作实现对数据库进行写入的情况下。

使用计数能让我们避开这个问题：我们会记录，在每个 servant 内，有多少请求正在执行，而如果某个 servant 忙，就不逐出它。这样，队列尺寸就不再是一个硬性的上限：长期运行的操作可能会临时造成队列中的 servant 数目大于这个上限。但只要额外的 servant 空闲下来，它们就会像平常一样被逐出。

逐出器队列并不存储 servant 的标识，相反，队列中的条目是用于逐出器映射表的迭代器。当我们要逐出某个 servant 时，这样的迭代器会很有用：我们不必搜索映射表，查找要逐出的 servant 的标识，而是可以简单地删除队尾的迭代器所指向的映射表条目。我们可以侥幸地把用于逐出器队列的迭代器作为映射表的一部分存储起来，把用于逐出器映射表的迭代器作为队列的一部分存储起来，因为当我们增加或删除条目某个时，`std::list` 和 `std::map` 都不会使迭代器失效（当然，要除开指向被删除条目的迭代器）。

最后，我们的 locate 和 finished 实现需要交换一个 cookie，其中含有一个智能指针，指向逐出器映射表中的条目。finished 要想使 servant 的引用计数减小，必须使用这个 cookie。

于是在我们的逐出器的 private 部分，就有了这样的定义:

```
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();
};
```

注意，逐出器在 private 数据成员 _map、_queue，以及 _size 中存储逐出器映射表、队列，以及队列尺寸。此外，我们还有一个 private _mutex 数据成员，用于正确地使“对逐出器的数据结构的访问”序列化。

evictServants 成员函数负责在队列长度超出限制时逐出 servant——我们很快将更详细地讨论这个问题。

构造器的实现非常简单。唯一要注意的是，我们会忽略负的尺寸[5]:

```
EvictorBase::EvictorBase(Ice::Int size) : _size(size)
{
    if (_size < 0)
        _size = 1000;
}
```

逐出器几乎所有的动作都发生在 locate 的实现中：

```
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. 我们可以改用 size_t 变量存储尺寸。但为了与 Java 实现保持一致（Java 实现不能使用无符号整数），我们使用了 Ice::Int 存储尺寸。

```
        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;
}
```

在 locate 中，第一步是锁住 _mutex 数据成员。这能够针对并发访问保护逐出器的数据结构。下一步是创建一个 cookie，由 locate 返回，并由 Ice run time 传给对应的 finished 调用。这个 cookie 含有一个智能指针，指向一个逐出器条目，类型是 EvictorEntryPtr。这也是我们的映射表条目的值类型，所以我们没有多余地存储同一信息的两份副本——相反，智能指针会确保 cookie 和映射表共享一份副本。

下一步是查找逐出器映射表，看我们是否已经拥有针对这个对象标识的条目。如果是，我们就用这个条目初始化 cookie 的智能指针，并在这个条目在队列中的当前位置处移除它。

否则我们就还没有针对这个对象标识的条目，所以我们必须创建一个。接下来的代码创建一个新的逐出器条目，然后调用 add 获取一个新的 servant。这是对 “将从 EvictorBase 派生的具体类”的向下调用。 add 的实现必须通过在 Current 对象中传入的标识、定位具有该标识的 Ice 对象的状态，然后，或者像平常一样返回一个 servant，或者返回 null，或者抛出异常表示失败。如果 add 返回 null，我们就抛出 ObjectNotExistException，让 Ice run time 知道，没有能为当前请求找到 servant。如果 add 成功，我们就把条目的使用计数初始化成零，并把这个条目插入逐出器映射表。

locate 的最后几行代码把针对当前请求的条目增加到逐出器队列的头部，从而维持其 LRU 属性；增加该条目的使用计数；最后把这个 servant 返回给 Ice run time。

finished 的实现比较简单。它减少条目的使用计数，然后调用 evictServants、清除任何应被逐出的 servant:

```
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();
}
```

接着，evictServants 检查逐出器队列：如果队列长度超出逐出器的尺寸，就扫描那些额外的条目。于是使用计数为零的条目就会被逐出：

```
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);
        }
    }
}
```

这些代码从逐出器队列的尾部开始，扫描额外的条目。如果某个条目的使用计数为零，就把它逐出：在调用派生类中的 evict 成员函数之后，这段代码就把被逐出的条目从映射表和队列中移除。

最后，deactivate 的实现把逐出器尺寸设成零，然后调用 evictServants。这将使得所有 servant 被逐出。Ice run time 保证，只有当对象适配器中已经没有请求在执行时，deactivate 才会被调用；因此，这时逐出器中的所有条目都保证是空闲的，将会被逐出：

```
void
EvictorBase::deactivate(const std::string& category)
{
    IceUtil::Mutex::Lock lock(_mutex);
```

```
    _size = 0;
    evictServants();
}
```

创建 Java 逐出器实现

我们在这里给出的逐出器被设计成一个抽象基类：要想使用它，你要从 Evictor.EvictorBase 基类派生一个对象，并实现两个方法，逐出器会在需要增加或逐出 servant 时调用这两个方法。于是产生了这样的类定义：

```
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;
}
```

注意，这个逐出器的几个构造器会设置队列尺寸，缺省参数把尺寸设置成 1000。

locate、finished，以及 deactivate 函数继承自 ServantLocator 基类；这些函数实现的逻辑负责以 LRU 顺序维护队列，并按照需要增加和逐出 servant。这些方法是同步方法，所以逐出器的内部数据结构针对并发访问得到了保护。

逐出器会在需要把新 servant 增加到队列、或从队列中逐出 servant 时调用 add 和 evict 函数。注意，这两个函数都是抽象函数，所以在派生类中必须实现它们。add 的任务是，实例化并初始化 servant，供逐出器使用。逐出器会在需要逐出 servant 时调用 evict 函数。这样，evict 可以进行任何清理活动。注意，add 可以返回一个 cookie，由逐出器传给 evict，所以你可以把上下文信息从 add 传到 evict。

接下来，我们需要考虑我们的逐出器实现所需的数据结构。我们需要两个主要的数据结构:

1. 一个映射表，把对象标识映射到 servant，这样我们就能高效地确定、在内存中是否有 servant 可用于到来的请求。
2. 一个列表，实现逐出器队列。这个列表总是保持 LRU 顺序。

   逐出器映射表不仅要存储 servant，还要记录一些管理信息:

   1. 映射表存储从 add 返回的 cookie，这样我们才能把这个 cookie 传给 evict。
   2. 映射表存储一个用于逐出器队列的迭代器，标记这个 servant 在队列中的位置。
   3. 映射表存储一个使用计数，每当有操作被分派到某个 servant 时，这个计数就会增大，每当操作完成时，这个计数就会减小。

   我们应该对最后两点做一点额外的解释:

- 逐出器队列必须按照最近最少使用的次序排序，也就是说，每当有调用到达时，我们就要在逐出器映射表中找到一个针对相应标识的条目，我们还必须在队列中定位 servant 的标识，把它移到队列的头部。但是，

为找到该标识而扫描队列很低效，因为所需时间是 $O(n)$。为解决这个问题，我们在映射表中存储一个迭代器，标记对应的条目在逐出器 队列中的位置。这样，我们在 $O(1)$ 时间内，就能从条目的当前位置处取出它，把它放到队列的头部去。

遗憾的是，在使用 `java.util` 提供的各种列表时，如果列表被更新，我们保存的指向某个列表位置的迭代器就会失效。为了处理这个问题，我们使用了一种特殊的链表实现 `Evictor.LinkedList`，这种实现没有上述局限。 `LinkedList` 的接口与 `java.util.LinkedList` 类似，但除了所指向的元素已被移除的迭代器，它不会使其他迭代器失效。为简洁起见，在此我们没有给出这个列表的实现—— 你可以在 Ice 源码包附带的本书的代码示例中找到这个实现。

- 我们在映射表中维护有一个使用计数，目的是避免不正确地逐出 servant。假定有一个 Ice 对象，标识是 *I*，客户要调用其上的一个长期运行的操作。作为响应，逐出器把 *I* 的一个 servant 增加到逐出器队列中。在这个调用执行的同时，其他客户也调用多个 Ice 对象上的操作，从而使得针对其他对象标识的更多 servant 也被增加到队列中。结果，标识 *I* 的 servant 会逐渐向队列尾部移动。如果对象 *I* 上的操作仍在执行，同时又有足够的针对其他 Ice 对象的客户请求到来， *I* 的 servant 就可能会被逐出，而同时它又仍在执行原来的请求。

  就这种情况自身而言，并没有什么危害。但如果在这个 servant 被逐出后，又有一个客户向对象 *I* 发出请求，逐出器并不知道 *I* 有一个 servant 仍然存在，就会为 *I* 增加第二个 servant。同一个 Ice 对象有两个 servant，很可能会带来问题，特别是在 servant 的操作实现对数据库进行 写入的情况下。

  使用计数能让我们避开这个问题：我们会记录，在每个 servant 内，有多少请求正在执行，而如果某个 servant 忙，就不逐出它。这样，队列尺寸就不再是一个硬性的上限：长期运行的操作可能会临时造成队列中的 servant 数目大于这个上限。但只要额外的 servant 空闲下来，它们就会像平常一样被逐出。

最后，我们的 `locate` 和 `finished` 实现需要交换一个 cookie，其中含有一个智能指针，指向逐出器映射表中的条目。 `finished` 要想 使 servant 的引用计数减小，必须使用这个 cookie。

于是在我们的逐出器的 private 部分，就有了这样的定义：

```
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;
}
```

注意，逐出器在 private 数据成员 _map、_queue，以及 _size 中存储逐出器映射表、队列，以及队列尺寸。映射表的键是 Ice 对象的标识，值的类型是 `EvictorEntry`。队列存储的是 `Ice::Identity` 类型的标识。

`evictServants` 成员函数负责在队列长度超出限制时逐出 servant——我们很快将更详细地讨论这个问题。

逐出器几乎所有的动作都发生在 `locate` 的实现中：

```
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;
}
```

第一步是创建在 Ice.Current 传入的对象标识的副本（如第 386 页所提到的，出于效率上的考虑，Ice.Current 和 Ice.Current 的成员的赋值发生在传给应用代码之前，而不是为每次调用分配一个新的对象）。

接下来，代码创建一个 cookie，由 locate 返回，并由 Ice run time 传给对应的 finished 调用。这个 cookie 含有一个逐出器条目，其类型是 EvictorEntry。

现在，我们用对象标识做键，在逐出器映射表中查找一个已有条目。如果在映射表中已经有该条目，我们就从逐出器队列中取出对应的标识（EvictorEntry 的 pos 成员是一个迭代器，标记的是该条目在逐出器队列中的位置）。

而如果在映射表中没有找到该条目，我们就创建一个新条目，并调用 add 方法。这是在向下调用“将从 EvictorBase 派生的具体类”。add 的实现必须定位指定的 Ice 对象（其标识已传入 Current 对象）的对象状态，然后像平常一样返回一个 servant，如果失败，则返回 null，或抛出异常。如果 add 返回 null，我们就抛出 ObjectNotExistException，让 Ice run time 知道，找不到与当前请求对应的 servant。如果 add 成功了，我们就把该条目的使用计数初始化成零，并把它插入逐出器映射表中。

最后几行代码使该条目的使用计数加一，在逐出器队列的头部添加该条目，并在把 servant 返回给 Ice run time 之前，存储该条目在队列中的位置。

finished 的实现比较简单。它使条目的使用计数减一，然后调用 evictServants，驱除那些应该被逐出的 servant:

```
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();
}
```

evictServants 继而检查逐出器队列：如果队列长度超过了逐出器的尺寸，超过的条目就会被扫描。任何使用计数为零的条目都将被逐出：

```
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);
        }
    }
}
```

这段代码从逐出器队列的尾部开始，对超出的条目进行扫描。如果某个条目的使用计数为零，它就会被逐出：在调用了派生类的 evict 成员函数之后，这段代码会把被逐出的条目从映射表和队列中移除。

最后， deactivate 的实现把逐出器的尺寸设成零，然后调用 evictServants。这样，所有的 servant 都会被逐出。Ice run time 保证，只有在这样的情况下，它才会调用 deactivate：在对象适配器中已经没有请求在执行；所以，这时在逐出器中的所有条目都将是空闲的，因而将会被逐出。

```
synchronized public final void
deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

使用 Servant 逐出器

要使用 servant 逐出器，你只需从 EvictorBase 派生一个类，实现 add 和 evict 方法。你可以通过这样的方式把 servant 定位器变成逐出器：取出你为 locate 编写的代码，把它放进 add——EvictorBase 会负责按最近最少使用的顺序维护缓存，并在必要时逐出 servant。除非你的 servant 需要进行清理（比如关闭网络连接或数据库句柄），否则 evict 的实现可以是空的。

逐出器的一个让人愉悦的方面是，你不需要对你的 servant 实现做任何改动：servant 不知道你正在使用逐出器。所以，要把逐出器添加到已有代码中，而又不对源码造成很大的干扰，是一件非常容易的事情。

与缺省的 servant 相比，逐出器能够带来相当大的性能改善：如果 servant 的初始化很昂贵（例如，因为 servant 的状态必须从网上读取），逐出器的性能就更比缺省 servant 要好得多，同时内存需求也会保持很低的水平。

我们将在 XREF 中看到，要驱除已被客户抛弃的 servant，逐出器也会很有用。

16.8 Ice::Context

在 6.11.1 和 8.11.1 中，代理的各重载方法都有一个拖尾的参数，类型是 `const Ice::Context &` (C++) 或 `java.util.Map` (Java)。这个参数的 Slice 定义是:

```
module Ice {
    local dictionary<string, string> Context;
};
```

你可以看到，上下文就是一个把串映射到串的词典，或者从概念上说，上下文就是一系列名 - 值对。每当有请求要发往服务器时，这个词典的内容（如果有的话）都会随同请求一起整编，也就是说，如果客户在上下文中放入一些名 - 值对，并在发出调用时使用这个上下文，服务器就将能使用客户所发送的这些名 - 值对。

在服务器端，操作的实现可以通过 `Ice::Current` 的 `ctx` 成员访问接收到的 `Context`（参见 16.5 节），并提取客户所发送的名 - 值对。

16.8.1 显式传递上下文

上下文提供了一种手段，可以把数量不限的参数从客户发往服务器，而不必在操作的型构中提到这些参数。例如，考虑下面的定义:

```
struct Address {
    // ...
};

interface Person {
    string setAddress(Address a);
    // ...
};
```

假定客户有一个指向 `Person` 对象的代理，它可以这样来做一些事情:

```
PersonPrx p = ...;
Address a = ...;

Ice::Context ctx;
ctx["write policy"] = "immediate";

p->setAddress(a, ctx);
```

在 Java 中，同样的代码看起来可能像是这样:

```
Person p = ...;
Address a = ...;

Ice.Context ctx = new java.util.HashMap();
ctx.put("write policy", "immediate");

p.setAddress(a, ctx);
```

在服务器端，我们可以提取客户所设置的策略值，从而影响 setAddress 的实现的工作方式。你可以这样编写 C++ 实现:

```
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.
    }
}
```

在这段代码中，服务器检查键为 "write policy" 的上下文的值，如果这个值是 "immediate"，就马上把客户发来的更新写到数据库中；如果没有设置写策略，或者我们无法识别其中包含的值，服务器就假定要采用一种更宽松的写策略（比如在内存中缓存更新，在后面再把它们写出）。这个操作的实现的 Java 版本在本质上是一样的，我们不再给出它。

16.8.2 隐式传递上下文

除了在进行调用时显式传递上下文，你还可以使用*隐式的*上下文。通过隐式上下文，你可以一次性地针对特定代理设置上下文，然后，每当你使用该代理调用操作时，先前设置的上下文会随同调用一起发送。为此，代理基类提供了一个成员函数 ice_newContext:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx
```

```
                ice_newContext(const Ice::Context &) const;
            // ...
        };
    }
}
```

对应的 Java 函数是:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_newContext(java.util.Map newContext);
    // ...
}
```

ice_newContext 创建一个新代理，在其中隐式地存储了所传入的上下文。注意， ice_newContext 的返回类型是 ObjectPrx，也就是说，在你使用新创建的代理之前，你必须把它向下转换到正确的类型。例如，在 C++ 中:

```
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
```

这个例子说明，一旦我们创建了 p2 代理，任何通过 p2 发出的调用都会隐式地发送先前设置的上下文。这个例子的最后一行代码说明，即使代理已经有了一个隐式上下文，你仍然可以在发出调用时显式地发送上下文——显式上下文总是会覆盖任何隐式的上下文。

16.8.3 取回隐式的上下文

你可以调用 ice_getContext，取回某个代理的隐式上下文。这个调用返回的是目前为该代理所设置的上下文（如果代理没有隐式上下文，返回的词典就是空的）。在 C++ 中，其型构是:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::Context ice_getContext() const;
            // ...
        };
    }
}
```

在 Java 中，其型构是:

```
package Ice;

public interface ObjectPrx {
    java.util.Map ice_getContext();
    // ...
}
```

16.8.4 上下文用例

通过使用 Ice::Context，你可以把这样的服务添加到 Ice 中：在处理每一个请求时，它们需要一些上下文信息。这样的上下文信息可用于像事务服务（为正在建立的事务提供上下文）或安全服务（向服务器提供授权令牌）这样的服务。IceStorm（参见第 26 章）使用了上下文来向服务提供可选的 cost 参数，用于影响服务在向“下游订阅者”发送消息时采用的传播方式。

一般而言，如果服务需要这样的上下文信息，使用上下文来实现会优雅得多，因为这样做隐藏了一些显式的 Slice 参数，应用程序员本来需要在每次发出调用时提供这些参数。

此外，因为上下文是可选的，你既可以把一个 Slice 定义应用于使用了上下文的实现，也可以把同样的定义应用于没有使用上下文的实现。这样，如果你要把事务语义添加到已有的服务，你无需修改 Slice 定义，给所有操作都增加额外的参数。这不仅方便了客户，而且还防止了把类型系统分成两半：如果没有上下文，对于（从概念上讲的）同一个服务的事务实现和非事务实现，我们就需要定义不同的 Slice 定义。

最后，使用隐式上下文，上下文信息可以穿过某个系统的多个中间部分，而无需与这些中间部分进行协作。例如，假定你针对某个代理设置了一个隐式上下文，然后把这个代理传给另一个系统组件。当该组件使用这个代理来调用操作时，隐式的上下文也会被发送出去。换句话说，隐式上下文允许你通过一些中间组件传播信息，这些组件不知道任何上下文的存在。

16.8.5 警告

上下文是一种强大的机制，*如果正确地加以使用*，可用于透明地传播上下文信息。特别地，你可能会被诱惑，把上下文当作一种版本管理的手段，在应用的演化过程中进行版本管理。例如，你的应用的版本 2 中的某个操作有两个参数，而版本 1 只有一个参数。你可以使用上下文，把第二个参数当作名 - 值对传给服务器，避免改动该操作的 Slice 定义，从而维持向后兼容性。

我们*强烈地*建议你，不要以这种方式使用上下文。这种策略有许多问题:

- 缺少上下文

  如果服务器期望接收上下文，没有任何办法能迫使客户实际发送上下文：如果客户忘了发送上下文，服务器必须以某种方式、在没有上下文的情况下应付过去（或抛出异常）。

- 缺少键，或键不正确

  即使客户发送了上下文，也不能保证它设置了正确的键（例如，一个简单的拼写错误就可能导致客户发送出具有错误的键的值）。

- 不正确的值

  上下文的值是串，而要发送的应用数据可能是数，也可能是更为复杂的东西，比如有若干成员的结构。这将迫使你把值编码成串，并在服务器端对串进行解码。这样的解析繁琐而易错，与发送强类型的参数相比，效率要低下得多。此外，服务器还必须处理不能正确解码的串值（例如，因为客户进行了错误的编码）。

如果你使用适当的 Slice 参数，上述问题一个也不可能发生：参数不可能被偶然遗漏，而且，它们是强类型的，所以客户偶然发送无意义的值的可能性很低。

如果你想随时间推移而演化应用，同时又不破坏向后兼容性， Ice 提供了一些可以更适用于这个任务的机制，比如 facets（参见 XREF）。上下文的设计用途是传送简单的 token（比如事务标识符），用于那些非如此就无

法合理实现的服务；你应该只把上下文用于这样的目的，而不要把它用于其他目的。

16.9 调用超时

客户发出的远地调用是同步的和阻塞的：在服务器处理完该调用之前，在客户端，调用不会完成。有时候，在一段时间之后，即使调用还没有完成，也迫使它终止，会很有用处。为此，代理提供了 ice_timeout 操作：

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_timeout(Ice::Int t) const;
            // ...
        };
    }
}
```

对应的 Java 方法是：

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_timeout(int t);
    // ...
}
```

ice_timeout 根据一个已有的代理，创建一个具有超时的代理。例如：

```
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
```

ice_timeout 的参数指定以毫秒为单位的超时值。值 -1 表示没有超时。在前面的例子中，超时被设成五秒；如果你通过 timeoutFile 调用

read，没有在五秒之内完成，该操作就会终止，抛出 Ice::TimeoutException。另一方面，通过 myFile 发出的调用不受超时的影响，也就是说，ice_timeout 设置的超时是针对相应的单个代理的。

给代理设置的超时值会影响所有的网络操作：读写数据，以及打开和关闭连接。如果有任何这样的操作没有在超时之前完成，客户就会收到异常。在数据读写过程中到期的超时由 TimeoutException 指示。对于连接打开和关闭，Ice run time 使用了另外的异常：

- ConnectTimeoutException

  这个异常指示，连接无法在指定时间之内建立。

- CloseTimeoutException

  这个异常指示，连接无法在指定时间之内关闭。

Ice run time 提供了 **Ice.Override.Timeout** 属性（参见附录 C）。通过这个属性，你可以用另外的值取代缺省的超时值 -1（没有超时）。如果你设置了这个属性，所有没有用 ice_timeout 设置超时的代理都会受影响。

注意，这些超时是“软”超时，也就是说，它们不是精确的、实时的超时（精确度受制于底层操作系统的能力）。你还应该知道，Ice run time 把超时当作是严重的出错：例如，超时会导致客户端的连接关闭。超时应该被用于防止客户在“服务器发生错误”的情况下无限期地阻塞；你不应该例行公事地把它们用于这样的情况：请求执行的时间比预期要长，所以你想要中止它。

16.10 单向调用

在第 2 章提到过，Ice run time 支持*单向*调用。在客户端，调用的发送是通过把请求写到客户的本地传输缓冲区来完成的；一旦本地传输机制接受了调用，调用就会完成，并把控制返回给应用代码。当然，这意味着，单向调用是不可靠的：它可能根本没有发送出去（例如，因为网络故障），也可能没有被服务器接受（例如，因为目标对象不存在）。如果出了什么问题，客户端应用代码不会收到失败通知；只有那些在调用过程中，发生在客户端的本地错误，才会报告给客户（例如，没有能建立连接）。

在服务器端，接收和处理单向调用的方式与其他请求是一样的。特别地，服务器端应用代码无法区分单向调用和双向调用，也就是说，在服务器端，单向调用是透明的。

在使用单向调用时，不会有任何数据从服务器返回客户：服务器不会发出答复消息来响应单向调用（参见第 18 章）。这意味着，单向调用可能会

带来很大的效率提升，特别是在发送大量小消息的情况下，因为客户无需等待对每条消息的答复，就可以发送下一条消息了。

要想单向调用某个操作，必须满足两个条件：

- 操作的返回类型必须是 `void`，不能有任何 `out` 参数，也不能有异常规范。

  这个要求反映了这样一个事实：服务器不会向客户发送单向调用答复：没有这样的答复，任何值或异常都无法返回给客户。

  如果你试图把某个有值要返回给客户的操作当作单向操作进行调用， Ice run time 会抛出 `TwowayOnlyException`。

- 操作所属的代理必须支持面向流的传输机制（比如 TCP 或 SSL）。

  单向调用需要使用面向流的传输机制（如果你需要在使用数据报传输机制的情况下，获得某种类似单向调用的功能，你需要使用数据报调用——参见 16.11 节）。

  如果你试图为一个没有提供面向流的传输机制的对象创建单向代理， Ice run time 会抛出 `NoEndpointException`。

尽管单向调用在理论上是不可靠的，在实践中，它们是可靠的（但并非不会失败的）：它们是通过面向流的传输机制发送的，所以它们不可能丢失，除非连接完全失败了。特别地，面向流的传输机制将使用平常所用的流控制，所以客户不会向服务器发送过多的消息。在客户端，如果客户的传输缓冲区满了， Ice run time 就会阻塞，所以客户端应用代码不可能把过多消息发往它的本地传输缓冲区。

因此，只要客户不持续生成消息，比服务器处理它们的速度还要快，单向调用通常就不会阻塞客户端应用代码，并且会立即返回。如果客户调用操作的速率超过了服务器处理它们的速率，客户端应用代码最后就会阻塞在某个操作调用中，直到在客户的传输缓冲区中，有足够的空间接受调用为止。

不管客户是否超过了服务器处理单向调用的速率，在服务器中，单向调用的执行都是异步的：客户的调用在消息到达服务器之前就会完成。

关于单向调用，你需要记住一件事情：它们可能会按照不同的次序在服务器中出现：因为单向调用是通过面向流的传输机制发送的，它们被接收的次序肯定和它们被发送的次序相同。但是，服务器会在每个调用自己的线程分派这些调用；因为各个线程会以占先方式进行调用，客户后发送的调用可能会比先发送的调用要先分派和执行。

因此，单向调用通常最适合用于简单的更新，它们本来是无状态的（也就是说，不依赖于周围的上下文，或先前的调用所建立的状态）。

创建单向代理

Ice 是通过用于调用操作的代理来选择双向、单向，以及数据报（16.11 节）调用的。在缺省情况下，所有的代理都被创建成双向代理。要以单向方式调用某个操作，你必须基于双向代理，另外创建一个单向调用代理。

在 C++ 中，所有的代理都派生自共同的 `IceProxy::Ice::Object` 类（参见 6.11.1 节）。这个代理基类含有一个方法，可用于创建单向代理，叫作 `ice_oneway`:

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_oneway() const;
            // ...
        };
    }
}
```

在 Java 中，代理派生自 `Ice.ObjectPrx` 接口（参见 8.11.2 节），`ice_oneway` 的定义是:

```
package Ice;

public interface ObjectPrx {
    ObjectPrx ice_oneway();
    // ...
}
```

我们可以调用 `ice_oneway`，创建一个单向代理，然后用这个代理来这样调用操作（这里给出的是 C++ 版本——Java 版本与此类似）:

```
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;
}
```

注意，我们使用了 uncheckedCast 来把代理从 ObjectPrx 向下转换成 PersonPrx：对于单向代理，我们不能使用 checkedCast，因为 checkedCast 要求服务器作出答复，而单向代理当然不允许那样的答复。如果你想要使用安全的向下转换，你可以首先把双向代理向下转换成实际的对象类型，然后再获取单向代理：

```
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;
    }
}
```

注意，尽管这段代码的第二个版本更安全一点（因为它使用了安全的向下转换），它的速度也会更慢（因为安全的向下转换需要发送额外的双向消息，造成了额外的开销）。

16.11 数据报调用

数据报调用是与单向调用等价的一种功能，适用于数据报传输机制。和单向调用一样，只有对那些返回类型为 `void`、并且没有 `out` 参数或异常规范的操作，才能进行单向调用（如果操作不满足这些标准，试图使用数据报调用就会引发 `TwowayOnlyException`）。此外，代理的端点必须至少包括一个 UDP 传输机制，才能使用数据报调用；否则，Ice run time 就会抛出 `NoEndpointException`）。

数据报调用的语义与单向调用的语义是类似的：没有答复会从服务器返回客户，同时，相对于客户而言，其处理是异步的；一旦客户的传输机制把调用接受进它的缓冲区中，数据报调用就完成了。但是，数据报调用有还有另外一些出错语义：

- 个别的调用可能会丢失，也可能会不按次序到达。

  在线路上，数据报调用是作为真正的数据报发送的，也就是说，个别的数据报可能会丢失，也可能会不按次序到达服务器。因此，不仅操作可能会不按次序分派，一系列调用中的某一个调用也可能会丢失（在使用单向调用时不可能发生这样的事情，因为，如果连接出了故障，一旦连接中断，*所有的*调用都会丢失）。

- 传输机制可能会造成 UDP 包的重复。

  因为 UDP 路由的本性使然，数据报可能会重复到达服务器。这意味着，对于数据报调用，Ice *不*保证“最多一次”语义（参见第 13 页）：如果 UDP 数据报重复了，在服务器中，同一个调用可能会被多次分派。

- UDP 包的尺寸是有限的。

  IP 数据报的最大尺寸是 65,535 字节。其中 IP 头要占用 20 字节，UDP 头占用 8 字节，剩下 65,507 字节是有效内容。如果某个调用（包括 Ice 请求头，参见第 18 章）整编后的形式超过了该尺寸，调用就会丢失（如果某个 UDP 数据报超过了尺寸限制，应用会收到 `DatagramLimitException`）。

因为数据报有着不可靠的本质，它们最适用于简单的更新消息，这些消息本来没有状态。此外，由于在广域网上数据报调用丢失的可能性很高，你应该只在局域网上使用数据报调用，在局域上网上它们丢失的可能性比较小（当然，不管丢失的可能性有多大，你都必须设计你的应用，让它能容忍消息丢失或重复）。

创建数据报代理

要创建数据报代理，你必须调用代理的 ice_datagram 方法，例如:

```
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;
}
```

和 16.10 节的单向例子一样，你可以先安全地向下转换到实际的接口类型，然后再获取数据报代理，而不是依靠这里所用的不安全的向下转换。但是，这样做可能并不好，原因有两个:

- 安全的向下转换是通过面向流的传输机制发送的。这意味着，如果使用安全的向下转换，就需要打开一个连接，其目的只是为了检验目标对象是否具有正确的类型。如果对这个对象的其他所有访问都是通过数据报进行的，这样做会很昂贵。
- 如果代理没有提供面向流的传输机制， checkedCast 就会失败，抛出 NoEndpointException，所以你只能把这种做法用于既提供 UDP 端点、也提供 TCP/IP 和 / 或 SSL 端点的代理。

16.12 成批的调用

单向调用和数据报调用通常会作为单条消息发送，也就是说，只要客户发出调用， Ice run time 就会立刻把单向或数据报调用发给服务器。如果客

户接连发送多个单向调用或数据报调用，客户端 run time 会因为每条消息陷入（trap into）OS 一次，这是一种很昂贵的做法。此外，在发送每条消息时，还要发送它自己的消息头（参见第 18 章），也就是说，假如有 *n* 条消息，就需要消耗 *n* 个消息头的带宽。如果客户发送了许多单向调用或数据报调用，额外的开销就会很可观。

为了避免这样的开销，你可以成批地发送单向调用和数据报调用：Ice run time 不是把它们作为单条消息发送，而是把一个批调用放在客户端缓冲区中。后继的批调用会添加到这个缓冲区中，在客户端累积起来，直到客户显式地刷出它们。相关的 API 是代理接口的一部分：

```
namespace IceProxy {
    namespace Ice {
        class Object : /* ... */ {
        public:
            Ice::ObjectPrx ice_batchOneway() const;
            Ice::ObjectPrx ice_batchDatagram() const;
            // ...
        };
    }
}
```

`ice_batchOneway` 和 `ice_batchDatagram` 方法把一个代理转换成批代理。一旦你获得了批代理，通过该代理发送的消息就会缓冲在客户端 run time 中，而不会立即发送出去。客户在调用了批代理上的一个或多个操作之后，可以显式地调用 `Communicator::flushBatchRequests`，刷出成批的调用：

```
module Ice {
    local interface Communicator {
        void flushBatchRequests();
        // ...
    };
};
```

这会使成批的消息被“大批”发送出去，在这些消息的前面只有一个消息头。在服务器端，成批的消息由单个线程分派，分派的次序是这些消息被写入批消息时的次序。这意味着，在单条批消息中的各个消息不可能以不同的次序出现在服务器中。而且，一条批消息中的消息或者被全部递送，或者一条也不被递送（即使对于成批的数据报，也同样是如此）。

关于成批的数据报调用，你需要记住，如果批消息中的调用数据远远超过了网络的 PDU 尺寸，个别 UDP 包就很有可能会因为分段而丢失。而即使丢失的只是一个包，也会造成整个批消息丢失。因此，成批的数据报调用最适用于这样的简单接口（或具有类似语义的接口）：它们有一些操

作，每个操作负责设置目标对象的某个属性（成批的单向调用没有这样的危险，因为它们是通过面向流的传输端点发送的，所以不可能丢失个别的包）。

如果你启用传输压缩，成批调用更加高效：许多孤立的小消息的压缩率不可能太好，而成批的消息的压缩率则可能会更好，因为压缩算法有了更多要处理的数据[6]。

16.13 测试代理的分派类型

代理接口提供了一些操作，允许你测试代理的分派模式（这些方法的 Java 版本是类似的，所以我们没有给出它们）。

```
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;
            // ...
        };
    }
}
```

这些操作允许你测试各个代理的分派模式。

16.14 Ice::Logger 接口

取决于各种属性的设置（参见第 14 章），Ice run time 会产生跟踪、警告，或错误消息。这些消息是通过 `Ice::Logger` 接口输出的：

6. 不管你是否使用成批的消息，你都只应该在速度较低的链接上启用压缩。对于高速的 LAN 连接，花在压缩和解压上的 CPU 时间通常会比直接传送未压缩数据要长。

```
module Ice {
    local interface Logger {
        void trace(string category, string message);
        void warning(string message);
        void error(string message);
    };
};
```

在你创建通信器时，会实例化一个缺省的日志记录器。缺省的日志记录器会记录标准的错误输出。除了错误消息以外， trace 操作还有一个 category 参数；这样，你可以把输出发给一个过滤器，把来自不同子系统的跟踪输出分开。

你可以设置和获取与一个通信器相关联的日志记录器：

```
module Ice {
    local interface Communicator {
        void setLogger(Logger log);
        Logger getLogger();
    };
};
```

setLogger 操作为通信器安装一个不同的日志记录器。Ice run time 会缓存为通信器设置的日志记录器。这意味着，在你创建了通信器之后，你应该设置一次日志记录器，然后就不再改变它了（如果你后来改变了日志记录器，有些消息会送往老日志记录器，有些会送往新记录器）。 getLogger 操作会返回当前的日志记录器。

通过改变与通信器相关联的 Logger 对象，你可以把 Ice 消息集成进你自己的消息处理系统。例如，对于一个复杂的应用，你可能已经有了一个日志记录框架。要把 Ice 消息集成进这个框架，你可以创建你自己的 Logger 实现，把消息送往已有的框架。

当你销毁通信器时，它的日志记录器*不会*被销毁。这意味着，你可以安全地使用某个日志记录器，即使是在它所关联的通信器的生命期之外。

为方便起见， Ice 提供了两种针对特定平台的日志记录器：一种实现会把它的消息记录到 Unix **syslog** 设施，另一种则会使用 Windows 时间日志。要激活 **syslog** 实现，你要设置 Ice.UseSyslog 属性；要激活 Windows 事件日志，你要设置 Ice.UseEventLog 属性。在把日志消息发往日志子系统时，这两种实现都使用了 Ice.ProgramName 属性的值来标识应用。

syslog 日志记录器实现既能在 C++ 中使用，也能在 Java 中使用，而 Windows 事件日志实现只能在 C++ 中使用。要进一步了解怎样在 C++ 中使用这两种日志记录器实现，参见 10.3.2 节。

16.15 Ice::Stats

Ice run time 通过 Ice::Stats 接口报告，所有的操作调用在线路上发送和接收的字节数:

```
module Ice {
    local interface Stats {
        void bytesSent(string protocol, int num);
        void bytesReceived(string protocol, int num);
    };

    local interface Communicator {
        setStats(Stats st);
        // ...
    };
};
```

当 Ice run time 从网络读取数据时，它会调用 bytesReceived； 当它把数据写往网络时，它会调用 bytesSent。下面是 Stats 接口的一种非常简单的实现:

```
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;
    }
};
```

要注册你的实现，你必须实例化 MyStats 对象，然后调用通信器的 setStats:

```
Ice::StatsPtr stats = new MyStats;
communicator->setStats(stats);
```

你可以在客户端、也可以在服务器端（或两端）安装 Stats 对象。下面是在一个简单的服务器中安装了 MyStats 对象之后，产生的一些示例输出:

```
tcp: received 14 bytes
tcp: received 32 bytes
tcp: sent 26 bytes
tcp: received 14 bytes
tcp: received 33 bytes
tcp: sent 25 bytes
...
```

在实践中，你的 Stats 实现可能会更加复杂一点：例如，这个对象可以在成员变量中累计统计信息，并且提供成员函数，用于让你访问累计的统计信息，而不是把所有信息都简单地打印到标准错误输出。

16.16 位置透明性

Ice run time 的一个有用的特性是*位置透明性*（location transparent）：客户无需知道 Ice 对象的实现的位置；对某个对象的调用会被自动引导到正确的目标，不管这个对象的实现是在本地地址空间中，在同一台机器上的另一个地址空间中，还是在一台远地机器上的另一个地址空间中。位置透明性十分重要，因为有了它，我们能够改变对象实现的位置，而不会破坏客户程序，同时，通过使用 IcePack（参见第 20 章），像域名和端口号这样的信息可以放在应用的外部，不用出现在串化代理中。

如果调用跨越了地址空间的界线（或者更准确地说，跨越了通信器的界线），Ice run time 会通过适当的传输机制分派请求。但是，如果调用所使用的代理和负责处理该调用的 servant 处在同一个通信器中（所谓的*并置*调用），在缺省情况下，Ice run time 不会通过在代理中指定的传输机制发送调用。相反，在 Ice run time 中，并置的调用会“走捷径”，直接被分派[7]。

之所以要这样做，原因是效率：例如，如果并置的调用是通过 TCP/IP 发送的，那么这些调用仍将通过操作系统内核发送（使用底板而不是网络），并且需要负担全部这样的开销：创建 TCP/IP 连接、把请求整编成包、陷入和陷出内核，等等。通过优化并置的请求，大部分这样的开销都可以避免，让并置的调用几乎和本地函数调用一样快。

出于效率上的考虑，并置的调用不完全是位置透明的，也就是说，在某些方面，并置调用的语义与跨越地址空间界线的调用不同。特别地，并置调用与普通调用在以下一些方面有所不同：

7. 注意，如果代理和 servant 使用的不是同一个通信器，即使调用者和被调用者处在同一个地址空间中，调用也*不是*并置的。

- 并置的调用会在发出调用的线程中分派，而不是由从服务器的线程池中取出的另外一个线程分派。
- 对象适配器的 “扣留”状态会被忽略：即使目标对象的适配器处在“扣留”状态，并置的调用也会被正常处理。
- 对于并置的调用，类和异常不会被切断。相反，接收者所接收到的类或异常的类型总是发送者所发送的派生类型。
- 如果并置的调用抛出的异常不在操作的异常规范中，在客户中引发的就是这个异常，而不是 `UnknownUserException`（这只适用于 C++ 映射）。
- 异步方法调用 (AMI) 和异步方法分派 (AMD) 不能用于并置的调用。
- 对于并置的调用，类工厂会被忽略。
- 调用超时会被忽略。

在实践中，这些差异通常并没有关系。在使用并置的调用时，最有可能让你惊奇的是，分派是在发出调用的线程中进行的，也就是说，并置的调用的行为就像是本地的同步过程调用。例如，如果发出调用的线程获取了一个锁，操作实现也想获取，就有可能发生问题：这会造成死锁，除非你使用递归互斥体（参见第 15 章）。

16.17 对比 Ice 与 CORBA Run Time

在 CORBA 中，与 Ice 的服务器端功能等价的是 Portable Object Adapter (POA)。 Ice 和 POA 最显著的差异是 Ice API 的简单性：Ice 的特性和 POA 一样完备，但其接口简单得多，操作也少得多。下面是一些值得注意的差异：

- Ice 对象适配器不是层次化的，而 POA 则被组织进一个层次体系中。我们不清楚 CORBA 为何要把它的适配器放进层次体系中。这种层次体系使得 POA 接口变得复杂化，却并未带来任何明显的好处：对象适配器的继承没有意义。特别地， POA 的策略（policy）*不是*从父适配器那里继承的。 POA 层次体系可用于控制 POA 的析构次序。但以正确的次序显式销毁适配器要容易得多，就像在 Ice 中所做的那样。
- POA 使用了一种复杂的策略机制来控制对象适配器的行为。多个策略可以通过许多方式组合起来，但大多数组合并没有意义。这不仅是一些编程错误的主要来源，而且还使得其 API 变得复杂化，给许多操作增加了额外的异常语义。
- CORBA run time 没有提供在方法分派过程中访问 ORB 对象（Ice 通信器的等价物）的途径，而在操作实现中，常常需要访问 ORB 对象。结

果，程序员被迫在全局变量中存放 ORB 对象，而如果使用了多个 ORB，你无法确定与特定请求对应的是哪个 ORB。而在 Ice 中没有这样的问题，你总是可以通过作为 `Current` 对象的一部分传入的适配器，访问通信器。

- POA 把一些实现技术与对象适配器系在一起。例如，使用 servant 定位器的对象适配器不能同时使用活动 servant 映射表。

  POA 还对 servant 定位器（与 Ice servant 定位器类似）和 servant 激活器（其工作方式与 16.7.1 节的渐近式 servant 定位器初始化类似）作出了区分。然而，区分这两种概念并不必要：servant 激活器只是 servant 定位器的一种特例，非常容易实现。

  与此类似，缺省的 servant 必须通过特殊的 API 调用、向 POA 注册，而事实上，缺省 servant 无需被当作一种单独的概念。如 16.7.2 节所示，在 Ice 中，你可以使用一个微不足道的 servant 定位器来获得同样的效果。

- POA 使用了另外的 POA 管理器对象来管理适配器状态。这不仅使 API 显著地复杂化了，而且还使得程序员有可能以一种无意义的方式、把对象适配器的层次体系与 POA 管理器的分组组合在一起，从而导致不确定的行为。Ice 没有使用另外的对象来控制适配器状态，从而消除了与之关联的各种复杂性，而又没有损失功能。

- POA 接口有缺省的 Root POA 的概念，程序员只有显式地重新定义它，才能使用它。这个错误的特性常常会造成各种问题，原因是为 Root POA 选用的策略不合适（CORBA 用户可能已经遇到过这样的问题：你隐式地激活了 Root POA 上、而不是你想用的 POA 上的对象）。

- POA 提供了隐式激活的概念，并且还可以把系统生成的对象标识用作对象适配器策略的一部分。这种设计常常会造成编程错误，因为它会使许多活动在幕后隐式地发生，而某些活动有着让人吃惊的副作用（CORBA 之所以增加所有这些复杂性，只是为了减少对象激活过程中的一行代码——这是一件让人悲哀的事情）。Ice 没有隐式激活或隐式生成对象标识的概念。相反， servant 的标识是显式指定的，其激活也是显式的，这种做法避免了复杂和混乱。

- CORBA 没有数据报或成批调用的概念，这两种功能都能带来很大的性能提升。

- CORBA 没有提供一种标准的方式，用以把 ORB 消息集成进已有的日志记录框架，也没有提供网络统计信息。

总而言之， Ice run time 提供了 POA 的全部功能，同时还提供了其他功能，但其 API 的尺寸却小得多。通过把各种正交的概念分解开来，并提供

最低限度的、同时也够用的 API， Ice 不仅提供了一个更简单、更易用的 API，而且还能产生尺寸更小的二进制文件。这不仅降低了 Ice 二进制文件的内存需求，同时还使工作集（working set）的尺寸得以缩减，从而带来了更好的性能。

16.18 总结

在这一章，我们详细探讨了服务器端 run time。通信器是 Ice run time 的主句柄。你可以通过通信器访问许多 run time 资源，并控制服务器的生命周期。对象适配器提供了抽象的 Ice 对象与具体的 servant 的映射。你可以通过各种实现技术来平衡性能和可伸缩性；特别地， servant 定位器是一种重要机制，你可以用它来选择与你的应用需求相吻合的实现技术。

Ice 既提供了单向调用，也提供了数据报调用。如果应用需要进行许多无状态的更新，这两种调用方式能够提高性能。通过成批处理这样的调用，你还可以进一步提高性能。

Ice 日志记录机制可以由用户进行扩展，所以你可以把 Ice 消息集成进任意的日志记录框架，同时， `Ice::Stats` 接口允许你收集网络带宽消耗方面的统计信息。

最后，即使 Ice 是位置透明的，为了提高效率，并置的调用的行为并非与远地调用完全一样。你需要注意这些差异，特别是在对线程上下文敏感的应用中。

第 17 章
异步程序设计

17.1 本章综述

这一章将描述 Ice 中的各种异步编程设施。 17.2 节简要介绍这些设施的能力，并演示怎样修改 Slice 定义，启用语言映射中的异步支持。 17.3 节介绍客户端设施，接着， 17.4 节讨论了服务器端设施。

17.2 引言

通过使远地调用变得像传统的方法调用一样容易，现代中间件技术力图减轻程序员在转向分布式应用开发时的负担：你调用某个对象上的方法，当该方法完成时，就会返回结果，或是抛出异常。当然，在分布式系统中，对象的实现可能会驻留在另一个主机上，从而带来一些语义上的差异，程序员必须加以注意，比如远地调用的开销，以及可能发生与网络有关的错误。尽管有这样一些问题，程序员的面向对象编程经验仍然是有用的，上述的*同步*编程模型（发出调用的线程会阻塞到操作返回）是我们所熟悉和容易理解的。

Ice 在本质上是一个异步的中间件平台，出于对应用（及其程序员）的考虑而模拟了同步的行为。当 Ice 应用通过代理、向远地对象发出同步的双向调用时，发出调用的线程会阻塞起来，以模拟同步的方法调用。在此期

间， Ice run time 在后台运行，处理消息，直到收到所需的答复为止，然后发出调用的线程就可以解除阻塞，解编结果了。

但是，在许多情况下，同步编程的阻塞本质太过受限。例如，在等待远地调用的响应返回时，应用本来可以另外做一些工作；在这样的情况下使用同步调用，应用就会被迫把工作推后、等待响应返回，或是在另外一个线程中执行该工作。如果这些做法都不可接受， Ice 的异步设施提供了一种有效的解决方案，可以改善性能和可伸缩性，或是简化复杂的应用任务。

17.2.1 异步方法调用

异步方法调用 (*AMI*) 这个术语描述的是客户端的异步编程模型支持。如果你使用 AMI 发出远地调用，在 Ice run time 等待答复的同时，发出调用的线程不会阻塞。相反，发出调用的线程可以继续进行各种活动，当答复最终到达时， Ice run time 会通知应用。通知是通过回调发给应用提供的编程语言对象的[1]。 17.3 节将详细描述 AMI。

17.2.2 异步方法分派

一个服务器在同一时刻所能支持的同步请求数受到 Ice run time 的服务器线程池的尺寸限制 （参见 15.3 节）。如果所有线程都在忙于分派长时间运行的操作，那么就没有线程可用于处理新的请求，客户就会经验到不可接受的无响应状态。

异步方法分派 (*AMD*) 是 AMI 的服务器端等价物，能够解决这个可伸缩性问题。在使用 AMD 时，服务器可以接收一个请求，然后挂起其处理，以尽快释放分派线程。当处理恢复、结果已得出时，服务器要使用 Ice run time 提供的回调对象，显式地发送响应。

用实际的术语说， AMD 操作通常会把请求数据 （也就是，回调对象和操作参数）放入队列 ，供应用的某个线程 （或线程池）随后处理用。这样，服务器就使分派线程的使用率降到了最低限度，能够高效地支持数千并发客户。

另外， AMD 还可用于需要在完成了客户的请求之后继续进行处理的操作。为了使客户的延迟降到最低限度，操作在返回结果后，仍留在分派线程中，继续用分派线程执行其他工作。

关于 AMD 的更多信息，参见 17.4 节。

1. Ice run time 不支持对响应进行轮询 （polling），但如果需要，应用很容易实现这个功能。

17.2.3 用元数据控制代码生成

程序员如果想要使用异步模型 (AMI、 AMD，或两者都使用)，需要给 Slice 定义批注上元数据 (4.17 节)。程序员可以在两个层面上指定这种元数据：接口或类的层面，或单个操作的层面。如果你是在为一个接口或类进行指定，那么为它的所有操作生成的代码都将会有异步支持。而如果只有某些操作需要异步支持，那么你可以只为这些操作指定元数据，从而使生成的代码的数据降到最低限度。

在代理中总是会生成同步调用方法；指定了 AMI 元数据，只是会增加异步调用方法。相反，指定了 AMD 元数据，同步分派方法就会被与它们对等的异步分派方法*取代*。 AMI 和 AMD 的这种语义差异完全来自实际的考虑：给客户提供某个调用方法的同步和异步版本是有益的，但如果在服务器中这样做，程序员就必须实现两种版本的分派方法，并不能带来切实的好处，而且还会带来若干潜在的缺陷。

考虑下面的 Slice 定义：

```
["ami"] interface I {
  bool isValid();
  float computeRate();
};

interface J {
  ["amd"]        void startProcess();
  ["ami", "amd"] int endProcess();
};
```

在这个例子中，为接口 I 的所有代理方法生成的代码都将具有同步和异步调用支持。在接口 J 中， startProcess 操作使用的是异步分派，而 endProcess 操作则支持异步的调用和分派。

在操作层面、而不是接口或类的层面指定元数据，不仅能使所生成的代码的数量降到最低限度，而且，更重要的是，它还能使复杂度降到最低限度。尽管异步模型更为灵活，它的使用也更复杂。因此，你最好限制异步模型的使用，只把它用于那些能从中获得某种好处的操作；对其他操作则使用更简单的同步模型。

17.2.4 透明性

使用异步模型并不会影响 “在线路上” 发送的数据。特别地，客户所用的调用模型对服务器而言是透明的，而服务器所用的分派模型对客户而言也是同名的。因此，服务器无法区分客户的同步调用和异步调用，而客户也无法区分服务器的同步答复和异步答复。

17.3 使用 AMI

在这一节，我们将描述 Ice 的 AMI 实现，以及怎样使用它。我们将首先讨论一种做法，用单向调用（部分地）模拟 AMI。我们并不推荐使用这种技术，但这个练习能让你了解 AMI 的各种好处，并了解它的工作方式。然后，我们将解释 AMI 映射，并通过例子说明它的用法。

17.3.1 用单向调用模拟 AMI

我们在本章的一开始讨论过，同步调用不适用于某些类型的应用。例如，有图形用户界面的应用通常必须避免阻塞窗口系统的事件分派线程，因为阻塞会使得应用不响应用户的命令。在这样的情况下，发出同步的远地调用会带来带来麻烦。

应用可以使用单向调用来避免发生这样的事情（参见第 15 页），按照定义，单向调用不能返回值，也不能有任何 out 参数。因为 Ice run time 不期望答复，调用的阻塞时间只是消息整编及复制进本地传输缓冲区所需的时间。但是，使用单向调用可能需要你对接口定义进行不可接受的改动。例如，一个要返回结果或引发用户异常的双向调用，必须被转换成至少两个操作：一个用于让客户发出只含有 in 参数的单向调用，另一个（或更多操作）用于让服务器把结果通知给客户。

为了阐释这些改动，假定我们有这样一个 Slice 定义：

```
interface I {
  int op(string s, out long l);
};
```

就其当前的形式而言，操作 op 不适于进行单向调用，因为它有一个 out 参数，以及一个非 void 返回类型。为了让 op 能用于单向调用，我们可以这样来改变 Slice 定义：

```
interface ICallback {
  void opResults(int result, long l);
};

interface I {
  void op(ICallback* cb, string s);
};
```

我们对原来的定义作了若干修改：

- 我们增加了接口 ICallback，其中含有一个操作 opResults，其参数代表的是原来的双向操作的结果。服务器调用这个操作，以通知客户，操作已完成。
- 我们修改了 I::op，让它遵从单向语义：它现在有了 void 返回类型，并且只有 in 参数。
- 我们给 I::op 增加了一个参数，用于让客户提供它的回调对象的代理。

你可以看到，为了满足客户的实现需要，我们对我们的接口定义做了重大的改动。这些改动还有一个影响：现在客户也必须是一个服务器，因为它必须创建 ICallback 的实例，向对象适配器注册它，以接收操作已完成的通知。

但一个更为严重的结果是，这些改动对类型系统产生了影响，因而也会影响服务器。客户是在同步还是异步地调用操作，应该和服务器没有关系；这种行为不应该对类型系统产生影响。如果像上面那样改动类型系统，我们就把服务器和客户紧密地耦合在一起，并且使得 op 不再能被异步调用。

更糟的是，考虑一下，如果 op 可能抛出用户异常，又会发生什么？在这种情况下，ICallback 必须进行扩展，增加额外的操作，用于让服务器把每一个异常的发生通知给客户。因为在 Slice 中，异常不能用作参数或成员类型，这样的扩展很快就会变得很困难，而所产生的结果也很可能会难以使用。

至此，你很可能会同意，这种技术在许多方面都是有缺陷的，那么我们又为何要如此详细地描述它呢？其原因是，Ice 的 AMI 实现所用的策略与上面描述的策略是类似的，只是有若干重要的不同：

1. 要使用 AMI，不用改变类型系统。数据在线路上的表示方式是相同的，因此，同步和异步的客户及服务器可以共存于同一个系统中，使用相同的操作。
2. 这种 AMI 解决方案以一种合理的方式容纳了异常。
3. 使用 AMI，不要求客户也同时是服务器。

17.3.2 语言映射

标记了 AMI 元数据的操作既支持同步调用模型，也支持异步调用模型。除了用于同步调用的代理方法，代码生成器还会创建一个用于异步调用的代理方法，再加上一个起辅助作用的回调类。生成的代码所用的模式 (pattern) 与我们在 17.3.1 节中对 Slice 定义所做的修改类似：out 参数和返回值被移除了，只留下了 in 参数；应用提供了一个回调对象，在回调时传入的是操作的结果。但是，在这里，这个回调对象是 Ice run time 将在客户

中调用的、纯粹的本地实体。回调类的名字的构造方式使得它不可能与用户定义的 Slice 标识符发生冲突。

C++ 映射

C++ 代码生成器为每个 AMI 操作生成以下代码:

1. 一个抽象的回调类， Ice run time 用它来通知应用，操作已完成。类名是按这样的模式取的: AMI_*class*_*op*。例如，对于在接口 I 中定义的名叫 foo 的操作 ，对应的类的名字是 AMI_I_foo。这个类所在的名字空间与操作所属的接口或类相同。这个类提供了两个方法:

   ```
   void ice_response(<params>);
   ```

   表明操作已成功完成。各个参数代表的是操作的返回值及 out 参数。如果操作的有一个非 void 返回类型， ice_response 方法的第一个参数就是操作的返回值。操作的所有 out 参数都按照声明时的次序，跟在返回值的后面。

   ```
   void ice_exception(const Ice::Exception &);
   ```

   表明抛出了本地或用户异常。

2. 一个额外的代理方法，其名字是操作在映射后的名字，加上 _async。这个方法的返回类型是 void，第一个参数是一个智能指针，指向上面描述的回调类的一个实例。其他的参数由操作的各个 in 参数组成，次序是声明时的次序。

   例如，假定我们定义了下面这个操作:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

下面是为操作 foo 生成的回调类:

```
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;
```

下面是为操作 foo 的异步调用生成的 代理方法:

```
void foo_async(const AMI_I_fooPtr &, Ice::Short);
```

Java 映射

Java 代码生成器为每个 AMI 操作生成以下代码:

1. 一个抽象的回调类， Ice run time 用它来通知应用，操作已完成。类名是按这样的模式取的: AMI_*class_op*。例如，对于在接口 I 中定义的名叫 foo 的操作 ，对应的类的名字是 AMI_I_foo。这个类所在的名字空间与操作所属的接口或类相同。这个类提供了三个方法:

   ```
   public void ice_response(<params>);
   ```

   表明操作已成功完成。各个参数代表的是操作的返回值及 out 参数。如果操作的有一个非 void 返回类型， ice_response 方法的第一个参数就是操作的返回值。操作的所有 out 参数都按照声明时的次序，跟在返回值的后面。

   ```
   public void ice_exception(Ice.LocalException ex);
   ```

   表明抛出了本地异常。

   ```
   public void ice_exception(Ice.UserException ex);
   ```

   表明抛出了用户异常。

2. 一个额外的代理方法，其名字是操作在映射后的名字，加上 _async。这个方法的返回类型是 void，第一个参数是一个引用，指向上面描述的回调类的一个实例。其他的参数由操作的各个 in 参数组成，次序是声明时的次序。

   例如，假定我们定义了下面这个操作:

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

下面是为操作 foo 生成的回调类:

```
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);
}
```

下面是为操作 foo 的异步调用生成的 代理方法:

```
public void foo_async(AMI_I_foo __cb, short s);
```

17.3.3 例子

为了演示 Ice 中的 AMI 的用法，让我们定义一个简单的计算引擎的 Slice 接口：

```
sequence<float> Row;
sequence<Row> Grid;

exception RangeError {};

interface Model {
  ["ami"] Grid interpolate(Grid data, float factor)
    throws RangeError;
};
```

给定一个两维的浮点值栅格和一个因子，interpolate 操作返回的是一个尺寸不变的新栅格，其值以某种有趣的（但却是未指定的）方式发生了改变。在下面的小节里，我们将给出两个 C++ 和 Java 客户，使用 AMI 调用 interpolate。

C++ 客户

我们首先必须定义我们的回调实现类，它派生自生成的 AMI_Model_interpolate 类：

```
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;
    }
  }
};
```

ice_response 报告成功的结果，如果发生异常，ice_exception 显示一条诊断信息。

调用 interpolate 的代码同样直截了当：

```
ModelPrx model = ...;
AMI_Model_interpolatePtr cb = new AMI_Model_interpolateI;
Grid grid;
initializeGrid(grid);
model->interpolate_async(cb, grid, 0.5);
```

在获取了 Model 对象的代理之后，客户实例化一个回调对象，初始化一个栅格，然后调用 interpolate 的异步版本。当 Ice run time 接收到对这个请求的响应时，会调用客户提供的回调对象。

Java 客户

我们首先必须定义我们的回调实现类，它派生自生成的 AMI_Model_interpolate 类：

```
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);
  }
}
```

ice_response 报告成功的结果，如果发生异常，ice_exception 显示一条诊断信息。

调用 interpolate 的代码同样直截了当：

```
ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

在获取了 `Model` 对象的代理之后，客户实例化一个回调对象，初始化一个栅格，然后调用 `interpolate` 的异步版本。当 Ice run time 接收到对这个请求的响应时，会调用客户提供的回调对象。

17.3.4 并发问题

在 Ice 中，异步调用由客户线程池提供支持 (第 15 章)，池中的线程主要负责处理答复消息。理解下列与异步调用相关的并发问题很重要:

- 一个回调对象不能同时用于多个调用。需要聚合来自多个答复的信息的应用可以创建一个单独的对象，让回调对象对它进行委托。
- 对回调对象的调用来自 Ice run time 的客户线程池中的线程，因此，如果在答复到达的同时，应用可能要与回调对象进行交互，就有可能需要进行同步。
- 客户线程池中的线程的数目决定了，同时可以为多少异步调用发出回调。客户线程池的缺省尺寸是一，意味着对回调对象的调用是序列化的。如果线程池的尺寸增大了，而同一回调对象被用于多个调用，应用就可能需要进行同步。

17.4 使用 AMD

这一节描述 AMD 的语言映射，并继续讨论我们在 17.3 节引入的例子。

17.4.1 语言映射

我们在 17.3.2 节讨论过，如果有需要，AMI 的语言映射仍然允许应用使用同步调用模型：如果为某个操作指定 AMI 元数据，用于同步调用的代理方法会完整地得以保留；同时，还会生成一个额外的代理方法，用以支持异步调用。

但是，AMD 操作的语言映射不允许我们的实现同时使用两种分派模型。如果你指定了 AMD 元数据，用于同步分派的方法就会被用于异步分派的方法取代。

异步分派方法的型构与 AMI 方法的类似：返回类型是 `void`，参数由一个回调对象、以及操作的 `in` 参数组成。在 AMI 中，回调对象是由应用提供的，而在 AMD 中，回调对象是由 Ice run time 提供的，同时它还提供了一些方法，用于返回操作的结果，或报告异常。我们的实现不需要在分派方法返回之前调用回调对象；回调对象可以在任何时候，由任何线程调

用，但只能被调用一次。回调类的名字的构造方式使得它不可能与用户定义的 Slice 标识符发生冲突。

下面将详细介绍两种语言映射。

C++ 映射

C++ 代码生成器为每个 AMD 操作生成以下代码:

1. 一个抽象的回调类，实现用它来通知 Ice run time，操作已完成。类名是按这样的模式取的：AMD_*class*_*op*。例如，对于在接口 I 中定义的名叫 foo 的操作，对应的类的名字是 AMD_I_foo。这个类所在的名字空间与操作所属的接口或类相同。这个类提供了若干方法:

   ```
   void ice_response(<params>);
   ```

   服务器可以用 ice_response 方法报告操作已成功完成。如果操作的返回类型不是 void，ice_response 的第一个参数就是返回值。与操作的 out 参数对应的参数跟在返回值后面，其次序是声明时的次序。

   ```
   void ice_exception(const Ice::Exception &);
   ```

   服务器可以用这个版本的 ice_exception 报告用户异常或本地异常。

   ```
   void ice_exception(const std::exception &);
   ```

   服务器可以用这个版本的 ice_exception 报告标准的异常。

   ```
   void ice_exception()
   ```

   服务器可以用这个版本的 ice_exception 报告未知异常。

2. 分派方法，其名字有后缀 _async。这个方法的返回类型是 void，第一个参数是一个智能指针，指向上面描述的回调类的一个实例。其他的参数由操作的各个 in 参数组成，次序是声明时的次序。

   例如，假定我们定义了下面这个操作:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

下面是为操作 foo 生成的回调类:

```
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();
};
```

下面是为操作 foo 的异步调用生成的分派方法:

```
void foo_async(const AMD_I_fooPtr &, Ice::Short);
```

Java 映射

Java 代码生成器为每个 AMD 操作生成以下代码:

1. 一个抽象的回调接口，操作实现用它来通知 Ice run time，操作已完成。接口名是按这样的模式取的: AMD_*class*_*op*。例如，对于在接口 I 中定义的名叫 foo 的操作，对应的类的名字是 AMD_I_foo。这个接口所在的名字空间与操作所属的接口或类相同。这个接口提供了两个方法:

   ```
   public void ice_response(<params>);
   ```

   服务器可以用 ice_response 方法报告操作已成功完成。如果操作的返回类型不是 void， ice_response 的第一个参数就是返回值。与操作的 out 参数对应的参数跟在返回值后面，其次序是声明时的次序。

   ```
   public void ice_exception(java.lang.Exception ex);
   ```

   服务器可以用 ice_exception 方法报告异常。相对于异常而言，AMD 实现的编译时类型安全性更少，因为分派方法没有 throws 子句，任何可以想象得到的异常类型都可以传给 ice_exception。但是， Ice run time 会使用与同步分派相同的语义，对异常值进行验证 (参见 4.8.4 节)。

2. 分派方法，其名字有后缀 _async。这个方法的返回类型是 void，第一个参数是一个引用，指向上面描述的回调接口的一个实例。其他的参数由操作的各个 in 参数组成，次序是声明时的次序。

   例如，假定我们定义了下面这个操作:

```
interface I {
  ["amd"] int foo(short s, out long l);
};
```

下面是为操作 foo 生成的回调接口:

```
public interface AMD_I_foo {
  void ice_response(int __ret, long l);
  void ice_exception(java.lang.Exception ex);
}
```

下面是为操作 foo 的异步调用生成的分派方法:

```
void foo_async(AMD_I_foo __cb, short s);
```

17.4.2 异常

在两种处理上下文中， AMD 操作的逻辑实现可能需要报告异常：分派线程 (也就是，接收调用的线程)，以及响应线程 (也就是，发送响应的线程)[2]。尽管我们建议你用回调对象来把所有异常报告给客户，实现抛出异常也是合法的，但只能在分派线程中抛出。

和你所预期的一样， Ice run time 无法捕捉从响应线程抛出的异常；应用的运行时环境将决定怎样处理这样的异常。因此，响应线程必须确保抓住所有异常，并用回调对象发送适当的响应。否则，如果响应线程被未捕捉的异常终止，请求就可能不会完成，客户将无限期地等待响应。

不管是在分派线程中引发，还是通过回调对象报告，用户异常都会按照 4.8.2 节所描述的方式进行验证，而本地异常可能会按照 4.8.4 节描述的方式进行翻译。

17.4.3 例子

在这一节，我们将继续我们在 17.3.3 节中引入的例子，但首先，我们必须修改操作，增加 AMD 元数据:

```
sequence<float> Row;
sequence<Row> Grid;

exception RangeError {};

interface Model {
  ["ami", "amd"] Grid interpolate(Grid data, float factor)
    throws RangeError;
};
```

下面的部分提供了 Model 接口的 C++ 和 Java 实现

2. 不一定存在两个不同的线程：如果需要，响应也可以从分派线程发出。

C++ Servant

我们的 servant 类派生自 Model，并且提供了 interpolate_async 方法的定义:

```
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;
};
```

interpolate_async 的实现使用了同步来在一个 Job 中安全地记录回调对象，并把它放进一个队列中:

```
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);
}
```

在把信息放进队列之后，该操作把控制返回给 Ice run time，使分派线程能够去处理另外的请求。一个应用线程从队列中移除下一个 Job，并调用 execute 来进行插值。下面是 Job 的定义:

```
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;
```

execute 的实现使用 interpolateGrid（没有给出）来完成计算工作：

```
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);
}
```

如果 interpolateGrid 返回 false， ice_exception 就会被调用，表明发生了范围错误。在调用了 ice_exception 之后，使用 return 语句是必要的，因为 ice_exception 并没有抛出异常；它仅仅是整编了参数，并把它发送给客户。

如果插值成功， ice_response 会被调用，把修改后的栅格返回给客户。

Java Servant

我们的 servant 类派生自 _ModelDisp，提供了 interpolate_async 方法的定义，该方法创建一个 Job，用以存放回调对象及参数，并把它添加到一个队列中。这个方法是同步的，用以保护对队列的访问：

```
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();
}
```

在把信息放进队列之后，该操作把控制返回给 Ice run time，使分派线程能够去处理另外的请求。一个应用线程从队列中移除下一个 Job，并调用 execute, 由它使用 interpolateGrid 来完成计算工作:

```
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;
}
```

如果 interpolateGrid 返回 false， ice_exception 就会被调用，表明发生了范围错误。在调用了 ice_exception 之后，使用 return 语句是必要的，因为 ice_exception 并没有抛出异常；它仅仅是整编了参数，并把它发送给客户。

如果插值成功， ice_response 会被调用，把修改后的栅格返回给客户。

17.5 总结

同步的远地调用是对本地的方法调用的自然扩展，它利用了程序员的面向对象编程经验，使初学分布式应用开发的程序员的学习曲线平缓下来。但是，同步调用的阻塞本质使得有些应用任务的实现变得更为困难，甚至不可能，因此， Ice 提供了一个直截了当的接口，你可以用这个接口来访问 Ice 的异步设施。

如果使用异步方法调用，发出调用的线程可以调用一个操作，然后马上就重获控制，不用阻塞起来等待操作完成。当 Ice run time 收到结果时，它会通过回调通知应用。

与此类似，异步方法分派允许 servant 在任何时候发送操作的结果，而不一定要在操作实现中发送。通过把费时的请求放在队列中，后面再进行处理， servant 可以改善可伸缩性，并节省线程资源。

第 18 章
Ice 协议

18.1 本章综述

Ice 协议定义由三个主要部分组成:

- 一组数据编码规则，确定各种数据类型的序列化方式
- 一些消息类型，在客户和服务器之间进行交换；还有一些规则，确定在何种情况下要发送何种消息
- 一组规则，确定客户与服务器怎样就特定的协议和编码版本达成一致

18.2 节 描述编码规则， 18.3 节描述各种协议消息， 18.4 节描述压缩，18.5 节解释怎样确定协议及编码的版本，并且解释了客户与服务器是怎样就共同的版本达成一致的 (目前，编码与协议规范的版本都是 1.0)。最后，18.6 节对 Ice 的协议和编码与 CORBA 的进行了对比。

18.2 数据编码

Ice 数据编码的关键设计目标是简单和高效。遵循这些原则，这种编码没有在字边界 (word boundary) 上对齐原始类型，因此避免了浪费空间，也消除了对齐所带来的复杂性。 Ice 数据编码产生的就是一个连续的数据流；数据不含填充字节，也无需在字边界上对齐。

数值类型的数据总是使用 "little-endian" 字节序进行编码（大多数机器都使用 "little-endian" 字节序，所以 Ice 数据编码 “正确” 的时候要比不正确的时候多）。 Ice 没有使用 “由接收者负责使其正确” 的方案，因为这种方案会引入额外的复杂性。例如，设想一下，有一个接收者链，每一个接收者都沿着链转发数据，直到数据到达最终的接收者（这样的拓朴结构常见于事件分发服务）。 Ice 协议允许所有在中间的接收者直接转发数据，无需进行解编：中间接收者只需复制二进制数据块，就可以把请求转发出去。而如果使用的是 “由接收者负责使其正确” 的方案，只要链中的下一个接收者的字节序与发送者的字节序不同，中间接收者就必须解编和重整编数据。这是一种低效的方案。

Ice 要求运行在 "big-endian" 机器上的客户和服务器付出额外的代价，对数据的字节进行交换，变成 "little-endian" 布局，但与发送或接收请求的总代价相比，那样的代价无关紧要。

18.2.1 尺寸

在数据编码中涉及到的许多类型，以及若干协议消息的成分，都有一个与之相关联的尺寸或计数。尺寸是非负的数值。尺寸和计数是按下面两种方式中的一种进行编码的：

1. 如果元素的数目少于 255，尺寸就被编码成单个 `byte`，指示元素的数目。
2. 如果元素的数目多于或等于 255，尺寸就被编码成一个值为 `255` 的 `byte`，再跟上一个 `int`，指示元素的数目。

使用这种编码来指示尺寸，比总是用一个 `int` 来存储尺寸要廉价得多，特别是在整编短串序列的情况下：255 以下的计数只需要一个字节，而不是四个字节。相应的代价是，大于 254 的计数需要五个字节，而不是四个字节。但是，对于长度大于 254 的序列或串而言，多出的一个字节无关紧要。

18.2.2 封装

封装 (encapsulation) 被用于容纳这样的变长数据：中间接收者可能无法解码这些数据，但它可以转发给另一个接收者，进行最终的解码。封装被编码成好像具有这样的结构：

```
struct Encapsulation {
    int size;
    byte major;
    byte minor;
    // [... size - 6 bytes ...]
};
```

size 成员以字节为单位指定封装的尺寸（包括 size、major，以及 minor 字段）。major 和 minor 字段指定在该封装中所包含的数据的编码版本（参见 18.5.2 节）。在版本信息后面是编码后的数据，长度是 size-6 个字节。

封装中的所有数据都是无上下文的（context-free），也就是说，在封装中的数据不能引用封装之外的任何数据。这一性质使得封装可以作为一块数据在地址空间之间转发。

封装可以嵌套，也就是说，包含另外的封装。

封装可以是空的，在这种情况下，它的字节计数是 6。

18.2.3 切片（Slice）

如果某个值的接收者只能部分地理解所接收的值（也就是说，只知道基类型，而不知道实际的运行时派生类型），异常和类就会被切断 (slicing)。为了让异常或类的接收者忽略值的那些它不理解的部分，异常和类的值会被整编成切片序列（每一级继承层次有一个切片）。一个切片由这样一些部分组成：一个编码成定长的四字节整数的字节计数，后面跟的是该切片的数据（字节计数包括计数自身所占用的四个字节，所以没有数据的空切片的字节计数是四）。某个值的接收者可以这样跳过一个切片：读取字节计数 b，然后丢弃输入流中接下来的 b-4 个字节。

切片可以是空的，在这种情况下，它的字节计数是 4。

18.2.4 基本类型

基本类型的编码方式如表 18.1 所示。整数类型 (`short`、 `int`、 `long`) 由两个互补的数表示，浮点类型 (`float`、 `double`) 使用 IEEE 标准格式 [6]。所有数值类型都使用 "little-endian" 字节序。

表 18.1. 基本类型的编码

类型	编码
`bool`	单个字节，值 `1` 表示 `true`， `0` 表示 `false`
`byte`	一个不作解释的字节
`short`	两个字节 (LSB、 MSB)
`int`	四个字节 (LSB .. MSB)
`long`	八个字节 (LSB .. MSB)
`float`	四个字节 (23 位小数尾数、8 位指数、符号位)
`double`	八个字节 (52 位小数尾数、11 位指数、符号位)

18.2.5 串

串被编码成：一个尺寸 (参见 18.2.1 节)，后跟以 UTF8 格式表示的串内容 [21]。串不是以 NUL 结尾的。空串被编码成尺寸为零。

18.2.6 序列

序列被编码成：一个尺寸 (参见 18.2.1 节)，表示序列中的元素的数目，后跟按照其类型进行了编码的元素。

18.2.7 词典

词典被编码成：一个尺寸 (参见 18.2.1 节)，表示词典中的键 - 值对的数目，后跟所有的键 - 值对。每个键 - 值对都被编码成像是一个 `struct`，其中含有键和值作为成员，次序不变。

18.2.8 枚举符

枚举值的编码取决于枚举符的数目：

- 如果枚举有 1 - 127 个枚举符，枚举值被整编成一个 `byte`。
- 如果枚举有 128 - 32767 个成员，枚举值被整编成一个 `short`。
- 如果枚举的成员多于 32767 个，枚举值被整编成一个 `int`。

枚举值就是对应的枚举符的序数值，第一个枚举符的值被编码成零。

18.2.9 结构

结构的各个成员按照各自的类型、以它们在 `struct` 声明中的出现次序进行编码。

18.2.10 异常

异常的整编如图 18.1 所示。

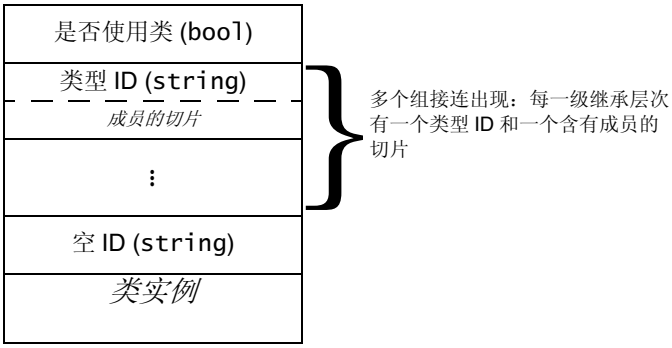

图 18.1. 异常的整编格式

每一个异常实例的前面都有一个字节，指示异常是否使用了类成员：如果有任何异常成员是类（或任何异常成员递归地含有类成员），这个字节的值就是 `1`，否则就是 `0`。

在头部的字节之后，异常被整编成对序列 (a sequence of pairs)：每个对的第一个成员是异常切片的类型 ID，第二个成员是一个切片，含有该切片整编后的成员。对序列是按照“从派生切片到基切片”的次序整编的，派生层次最深的切片在最前面，最后是派生层次最浅的切片。在对序列的末

尾，是一个空串。在每个切片中，数据成员像结构一样被整编，其次序是它们在 Slice 定义中出现的次序。

如果异常的成员使用了类实例，这些实例会在对序列的后面被整编。这个最后的部分是可选的：只有当头字节是 1 时才有这个部分 (关于类实例的详细整编方式，参见 18.2.11 节)。

让我们举个例子阐释一下整编，考虑下面的异常层次：

```
exception Base {
    int baseInt;
    string baseString;
};

exception Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

假定异常的各个成员按表 18.2 所示的值进行了初始化。

表 18.2. 一个类型为 Derived 的 异常的成员值

成员	类型	值	整编后的尺寸 (按字节数计算)
baseInt	int	99	4
baseString	string	"Hello"	6
derivedBool	bool	true	1
derivedString	string	"World!"	7
derivedDouble	double	3.14	8

根据表 18.2 的内容，我们可以看到， Base 的各成员的总尺寸是 10 字节， Derived 的各成员的总尺寸是 16 字节。异常的成员中没有类。这个异

常的一个实例在线路上的表示如表 18.3 所示 (标出了每个成员整编后的表示的尺寸、类型，以及字节偏移量)。

表 18.3. 表 18.2 中的异常整编后的表示

整编后的值	按字节计算的尺寸	类型	字节偏移量
0 *(没有类成员)*	1	bool	0
"::Derived" *(类型 ID)*	10	string	1
20 *(切片的字节计数)*	4	int	11
1 *(derivedBool)*	1	bool	15
"World!" *(derivedString)*	7	string	16
3.14 *(derivedDouble)*	8	double	23
"::Base" *(类型 ID)*	7	string	31
14 *(切片的字节计数)*	4	int	38
99 *(baseInt)*	4	int	42
"Hello" *(baseString)*	6	string	46
"" *(空串)*	1	string	52

注意，每个串的尺寸都比实际串的长度要大 1。这是因为，如 18.2.5 节所解释的那样，每个串的前面都有一个字节计数。

这个值序列的接收者使用头字节来确定，它最终是否必须解编包含在异常中的类实例（在这个例子里没有），然后再检查第一个类型 ID(::Derived)。如果接收者能够识别这个类型 ID，它就可以解编第一个切片的内容（其他的切片跟在后面）；否则，接收者读取跟在这个未知类型之后的字节计数 (20)，跳过输入流中的 20-4 个字节，到达第二个切片 (::Base) 的类型 ID 的起始处。如果接收者也无法识别这个类型 ID，它就再读取跟在这个类型 ID 后的字节计数 (14)，跳过 14-4 个字节，然后读取空的类型 ID。这个空的 ID 表明，异常已经结束。如果接收者没有能够识别所收到的异常的任何类型 ID，它就会把 UnknownUserException 抛给应用。

18.2.11 类

由于需要处理类图（graphs of classes）的指针语义，同时，接收者还需要对未知的派生类型的类进行切断，所以类的整编很复杂。此外，类的整编还使用了一种类型 ID 压缩方案，用以避免反复整编大型的类实例图的相同类型 ID。

基本的整编格式

类的整编与异常类似：每个实例都被划分成一些对，其中含有一个类型 ID 和一个切片（每一级继承层次有一个对），并按照“从派生切片到基切片”的次序进行整编。只有数据成员会被整编——没有与操作有关的信息被发送。与异常不同，在类的前面没有头字节。相反，在每个整编后的类实例前面都有一个（非零）正数，充当该实例的标识。发送者在整编过程中指定这个标识，使每个被整编的实例都有不同的标识。接收者使用该标识来重新正确地构造类图。与异常不同，在类型 ID 和切片对的后面*没有*空 ID。类的总整编格式如图 18.2 所示。

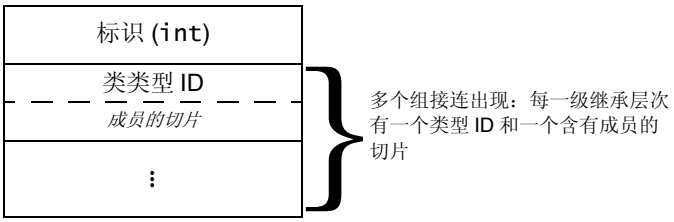

图 18.2. 类的整编格式

类的类型 ID

与异常的类型 ID 不同，类的 ID 不是简单的串。相反，类的类型 ID 被整编成一个布尔值，后面跟着一个串或一个尺寸，从而节省带宽。为了说明这一点，考虑下面的类层次：

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
};
```

这两个类的切片的类型 ID 是 ::Derived 和 ::Base。假定发送者整编了 ::Derived 的三个实例，作为一次请求的组成部分（例如，有两个实例可能是 out 参数，另一个实例可能是返回值）。

在线路上发送的第一个实例含有类型 ID ::Derived 和 ::Base，位于它们各自的切片的前面。因为整编是按照“从派生切片到基切片”的次序进行的，先发送的类型 ID 是 ::Derived。每当发送者发送一个它先前没有在同一请求中发送过的类型 ID 时，它就会发送布尔值 false，后面跟着类型 ID。在内部，发送者还给每个类型 ID 指定一个唯一的正数。这些正数从 1 开始，如果要发送的类型 ID 先前没有整编过，这个数就会加一。这意味着，这个例子中的第一个类型 ID 被编码成布尔值 false，后面跟着 ::Derived，第二个类型 ID 被编码成布尔值 false，后面跟着 ::Base。

当发送者整编余下的两个实例时，它会用先前整编过的类型 ID 构造一个查找表。因为先前发送的两个类型 ID 是在同一个请求（或答复）中，每当 ::Derived 再度出现，发送者就把它编码成值 true，后面跟着编码成尺寸的数值 1(参见 18.2.1 节)；每当 ::Base 再度出现，发送者就把它编码成值 true，后面跟着编码成尺寸的数值 2。

当接收者读取类型 ID 时，它首先会读取它的布尔标记:

- 如果这个布尔值是 false，接收者就读取一个串，并把这个串放入一个把整数映射到串的查找表中。在一次请求中，所收到的第一个新的类型 ID 的编号是 1，第二个新类型 ID 的编号是 2，如此等等。
- 如果这个布尔值是 true，接收者就读取一个编码成尺寸的数值，用这个数值来做查找表的索引，取回对应的类类型 ID。

注意，这种编号方案会针对每一个新的封装重新建立（我们将在 18.3 节看到，参数、返回值及异常总是会被整编进一个封装中）。对于后继的或嵌套的封装，编号方案会重新开始，给第一个新的类型 ID 指定值 1。换句话说，每个封装都会使用自己独立的类类型 ID 编号方案，以满足封装不能依赖周围的上下文的约束。

以这样的方式对类的类型 ID 进行编码，能够显著地节省带宽：只要 ID 被再次或多次整编，它就会被整编成一个两字节的值 (假定一次请求的各不相同的类型 ID 不超过 254 个)，而不是整编成串。因为类型 ID 可能会很长，特别是在使用嵌套的模块的情况下，所以节省的带宽会相当可观。

简单的类整编例子

为了让前面的讨论更具体一点，让我们考虑下面的类定义:

```
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;
};
```

注意，Base 和 Derived 都有操作，而 Derived 还实现了 SomeInterface 接口。因为类的整编所关心的是状态，而不是行为，在整编过程中，操作 op1、op2 及 op3 会被忽略，在线路上的表示就好像这些类是这样定义的一样：

```
class Base {
    int baseInt;
    string baseString;
};

class Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

假定发送者整编了 Derived 的两个实例 (例如，作为同一个请求中的两个 in 参数)。各成员值如表 18.4 所示。

表 18.4. Derived 类的两个实例的成员值

	成员	类型	值	整编后的尺寸 (按字节计算)
第一个实例	baseInt	int	99	4
	baseString	string	"Hello"	6
	derivedBool	bool	true	1
	derivedString	string	"World!"	7
	derivedDouble	double	3.14	8
第二个实例	baseInt	int	115	4
	baseString	string	"Cave"	5
	derivedBool	bool	false	1
	derivedString	string	"Canem"	6
	derivedDouble	double	6.32	8

发送者任意地给每个实例指定一个非零标识 (参见第 470 页)。通常，发送者会简单地从 1 开始给实例连续编号。在这个例子中，我们假定两个

实例的标识是 1 和 2。两个实例整编后的表示如表 18.5 所示（假定它们是紧接着整编的）。

表 18.5. 表 18.4 中的两个实例整编后的表示

整编后的值	按字节计算的尺寸	类型	字节偏移量
1 *(标识)*	4	int	0
0 *(类的类型 ID 的标记)*	1	bool	4
"::Derived" *(类的类型 ID)*	10	string	5
20 *(切片的字节计数)*	4	int	15
1 *(derivedBool)*	1	bool	19
"World!" *(derivedString)*	7	string	20
3.14 *(derivedDouble)*	8	double	27
0 *(类的类型 ID 的标记)*	1	bool	35
"::Base" *(类型 ID)*	7	string	36
14 *(切片的字节计数)*	4	int	43
99 *(baseInt)*	4	int	47
"Hello" *(baseString)*	6	string	51
0 *(类的类型 ID 的标记)*	1	bool	57
"::Ice::Object" *(类的类型ID)*	14	string	58
5 *(切片的字节计数)*	4	int	72
0 *(词典条目数)*	1	size	76
2 *(标识)*	4	int	77
1 *(类的类型 ID 的标记)*	1	bool	81
1 *(类的类型 ID)*	1	size	82
19 *(切片的字节计数)*	4	int	83
0 *(derivedBool)*	1	bool	87
"Canem" *(derivedString)*	6	string	88

表 18.5. 表 18.4 中的两个实例整编后的表示

整编后的值	按字节计算的尺寸	类型	字节偏移量
6.32 *(derivedDouble)*	8	double	94
1 *(类的类型 ID 的标记)*	1	bool	102
9 *(切片的字节计数)*	4	int	103
2 *(类的类型 ID)*	1	size	107
115 *(baseInt)*	4	int	108
"Cave" *(baseString)*	5	string	112
1 *(类的类型 ID 的标记)*	1	bool	117
3 *(类的类型 ID)*	1	size	118
5 *(切片的字节计数)*	4	int	119
0 *(词典条目数)*	1	size	123

注意，因为类（就像异常）是作为切片的序列发送的，类的接收者可以切掉它不理解的任何派生部分。还要注意，（如表 18.5 所示）每个类实例都含有三个切片。第三个是类型 `::Ice::Object` 的切片，这种类型是所有类的基类型。在这个例子中，类类型 ID `::Ice::Object` 的编号是 3，因为它是发送者所整编的第三种与其他 ID 不同的类型 ID（参见表 18.5 中的字节偏移 58 和 118 处）。所有类实例的最后一个切片的类型都是 `::Ice::Object`。

为 `::Ice::Object` 单独整编一个切片是必需的，因为 `::Ice::Object` 含有每个对象的 face 映射表 (参见 XREF)。作为一种内建类型，`::Ice::Object` 没有 Slice 定义。但它在整编时就好像它是这样定义的:

```
module Ice {
    class Object;

    dictionary<string, Object> FacetMap;

    class Object {
        FacetMap facets;
    };
};
```

注意，如果某个类没有数据成员，该类的类型 ID 和切片仍然会进行整编。在这种情况下，切片的字节计数将是 4，表明这个切片没有包含数据。

整编指针

类支持指针语义，也就是说，你可以构造类图 (graphs of classes)。类因而可以任意地互指。类标识 (参见第 470 页) 被用于这样区分实例和指针:

- 类标识为 0，表示 null 指针。
- 类标识 > 0，放在实例整编后的内容之前 (参见第 470 页)。
- 类标识 < 0，表示一个指向某个实例的指针。

小于零的标识值是指针。例如，如果接收者接收到标识 -57，那就意味着，对应的正在被解编的类成员最终将指向标识为 57 的实例。

对于不含类成员的结构、类、异常、序列，以及词典成员， Ice 协议使用了一种简单的深度优先遍历算法来整编成员。例如，结构成员是按照它们的 Slice 定义出现的次序整编的；如果某个结构成员自身的类型是复杂类型，比如序列，序列整编后的内容就会出现在它所属的结构的相应位置处。对于含有类成员的复杂类型，这个深度优先的整编过程会被挂起：此时，实际的类实例不会被整编，而是会整编一个负标识，表明该成员最终必须表示哪一个类实例。例如，考虑下面的定义:

```
class C {
    // ...
};

struct S {
    int i;
    C firstC;
    C secondC;
    C thirdC;
    int j;
};
```

假定我们初始化了一个类型为 S 的结构:

```
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;
```

当这个结构被整编时，三个类成员的内容不会被马上整编。相反，发送者会整编相应的实例的负标识。假定发送者已经把标识 78 指派给赋给 myS.firstC 的实例， myS 的整编如表 18.6 所示。

表 18.6. myS 整编后的表示

整编后的值	按字节计算的尺寸	类型	字节偏移量
99 *(myS.i)*	4	int	0
-78 *(myS.firstC)*	4	int	4
0 *(myS.secondC)*	4	int	8
-78 *(mys.thirdC)*	4	int	12
100 *(myS.j)*	4	int	16

注意， myS.firstC 和 myS.thirdC 都使用了标识 -78。这样，接收者就会知道 firstC 和 thirdC 都指向同一个类实例 (而不是两个内容碰巧相同的不同实例)。

整编负标识，而不是实例的内容，使得接收者能准确地重新构造出发送者发送的类图。但是，这带来了一个问题：实际的实例*何时*像本节一开始描述的那样进行整编？我们将在 18.3 节中看到，参数和返回值就好像它们是结构的成员一样被整编。例如，如果某个操作调用有五个输入参数，客户会依次整编所有这五个参数，就好像它们是同一个结构的成员一样。如果其中的某些参数是类实例，或者是含有类实例的复杂类型，发送者就会对参数进行多遍整编：第一遍使用平常的深度优先算法，依次整编所有参数：

- 如果发送者在整编过程中遇到类成员，它会检查自己是否曾为当前的请求或答复整编过同一个实例：
  - 如果该实例此前没有被整编过，发送者为该实例指定一个新标识，并对负标识进行整编。
  - 如果该实例此前整编过，发送者就会发送先前为该实例发送过的负标识。

  实际上，在整编过程中，发送者会构建一个标识表，其索引是每个实例的地址；为实例查找到的值是它的标识。

一旦第一遍整编结束，发送者就整编完了所有参数，但还没有整编各个参数或成员指向的任何类实例。这时的标识表已经包含了所有那些实例，它们的负标识（指针）已经整编过了，所以这时在标识表中的标识都是接收者仍然需要的类的标识。现在，发送者会整编在标识表中的实例，但使用的是正标识，后面跟着它们的内容（参见第 471 页）。待决的实例会被整编成一个序列，也就是说，发送者先把实例的数目整编成尺寸 (参见 18.2.1 节)，后面跟着实际的实例。

刚刚发送的实例也可能会含有类成员；当这些类成员被整编时，发送者会像平常一样，给新实例指定一个标识，如果该实例先前整编过，就使用它的负标识。这意味着，到第二遍整编结束时，标识表可能已经增大，从而需要第三遍整编。第三遍整编再次把待决的类实例整编成一个尺寸、后跟实际的实例。第三遍整编处理的是在第二遍中待决的实例。当然，第三遍整编也可能会触发更多遍整编，直到最后，发送者发送完所有待决的实例，也就是说，整编完成了。这时，发送者会整编一个空序列（把值 0 编码成尺寸），从而结束整编。

让我们用一个例子来对此加以阐释，再次考虑第 99 页的 4.9.7 节给出的定义：

```
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;
};
```

这些定义允许我们构造表达式树。假定客户初始化一棵树，让它具有图 18.3 所示的形状，代表表达式 (1 +6 / 2) ∗ (9−3)。在节点外面的值是客户指定的标识。

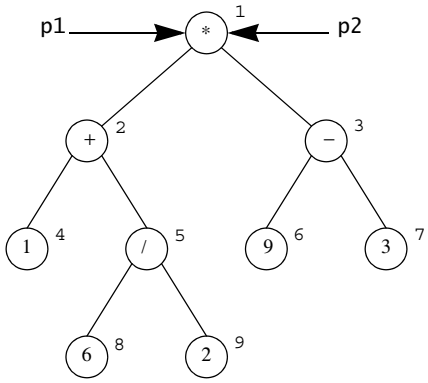

图 18.3. 表达式 (1 +6 / 2) ∗ (9−3) 的表达式树。p1 和 p2 都表示根节点

客户通过参数 p1 和 p2 把树根传给下面的操作 (尽管两次传递同一个参数没有意义，我们在这里这样做是出于讲解的目的):

```
interface Tree {
    void sendTree(Node p1, Node p2);
};
```

现在，客户整编 p1 和 p2 这两个参数，传给服务器，从而 致使值 -1 被连着发送两次 （客户给每个节点任意地指定一个标识。只要每个节点的标识是唯一的，标识的值是多少就无关紧要。为简单起见， Ice 实现用一个从 1 开始的计数器来为实例编号，每有一个与其他实例不同的实例，计数器都会加一)。这样，参数的整编就完成了，而且在标识表中有了一个标识为 1 的 实例。如第 471 页所描述的那样，客户现在整编了一个序列，其中含有一个元素节点 1。节点 1 又致使节点 2 和节点 3 被添加到标识表中，所以下一个节点序列含有两个元素，节点 2 和节点 3。再下一个节点序列含有节点 4、 5、 6、 7，后面跟着另外一个序列，其中含有节点 8 和节点 9。这时，不再有类实例在等待整编，客户会整编一个空序列，向接收者表明，序列已经整编完了。

在每一个序列中，类实例的整编次序无关紧要。例如，第三个序列中的节点的次序同样可以是 7、 6、 4、 5。在这里，重要的是，每个序列所包含的节点距离初始节点的 “跳” 数是相同的：第一个序列包含的是初始节

点，第二个序列包含的节点都可以从初始节点经过一个链接到达，第三个序列包含的节点都可以从初始节点经过两个链接到达，等等。

现在再考虑一下同样的例子，但 sendTree: p1 具有不同的参数值，表示的是树根，而 p2 表示的是右手的子树的 –运算符，如图 18.4 所示。

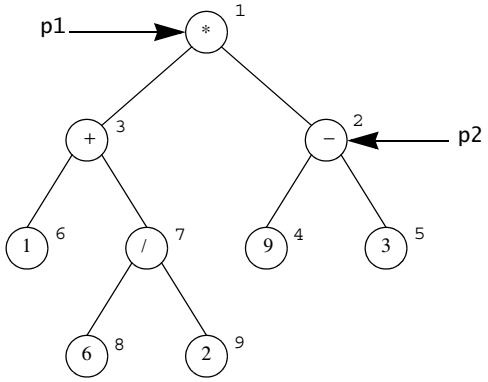

图 18.4. 图 18.3 的表达式树， p1 和 p2 表示的是不同的节点

这个图的整编过程与前面一样，但各个实例是按不同的次序整编的，并且具有不同的标识:

- 在第一遍整编过程中，客户发送参数值的标识-1 和-2。
- 第二遍整编了一个序列，其中含有节点 1 和节点 2。
- 第三遍整编了一个序列，其中含有节点 3、 4、 5。
- 第四遍整编了一个序列，其中含有节点 6 和节点 7。
- 第五遍整编了一个序列，其中含有节点 8 和节点 9。
- 最后一遍整编了一个空序列。

以这样的方式可以传送任何节点图（包括有循环的图）。通过在解编过程中填充一个修补表 (patch table)，接收者可以把图重新构造出来:

- 每当接收者解编负标识，它都会把该标识添加到修补表中；表的查找值 (lookup value) 是参数或成员的内存地址，该地址最终将会指向对应的实例。
- 每当接收者解编实际的实例，它都会把该实例添加到一个解编表中；表的查找值是已实例化的类的内存地址。然后，接收者使用实例的地址来修补任何具有这个实际的内存地址的参数或成员。

注意，接收者可能会收到其所表示的类实例已经解编过的负标识（也就是说，“向后”指向解编流中的某个地方，也可能，其所表示的类实例

还没有解编过（也就是说，“向前”指向解编流中的某个地方）。取决于实例的整编次序，以及它们的深度，这两种情况都有可能。

让我们再来看一个例子，考虑下面的定义：

```
class C {
    // ...
};

sequence<C> CSeq;
```

假定客户整编了一个序列，含有 100 个各不相同的 C 实例，发给服务器（也就是说，序列中含有 100 个指针，指向 100 个不同的实例，而不是 100 个指针，指向同一个实例）。在这种情况下，序列被整编成一个值为 100 的尺寸，后面跟着 100 个负实例，从-1 到-100。紧接着，客户整编了一个含有 100 个实例的序列，每个实例都有一个正标识，其范围是 1 到 100。最后整编一个空序列作为结束。

另一方面，如果客户要发送一个有 100 个元素的序列，这些元素全都指向同一个类实例，客户就会把这个序列整编成一个值为 100 的尺寸，后面跟着 100 个负标识，值全都是-1。然后整编一个只有一个元素（也就是实例 1）的序列。最后整编一个空序列作为结束。

类图与切断

注意到这样一个问题很重要：当类实例的图被发送时，它总是会形成一个连通图 (connected graph)。但当接收者重新构造这个图时，由于切断的缘故，最终得到的可能是非连通图。考虑：

```
class Base {
    // ...
};

class Derived extends Base {
    // ...
    Base b;
};

interface Example {
    void op(Base p);
};
```

假定客户有完整的类型知识，也就是说，既理解 Base 类型、也理解 Derived 类型，而服务器只理解 Base 类型，所以 Derived 实例的派生部分会被切掉。客户可以按照下面的方式实例化要作为参数 p 发送的类：

```
DerivedPtr p = new Derived;
p->b = new Derived;
ExamplePrx e = ...;
e->op(p);
```

就客户而言，图看起来如图 18.5 所示。

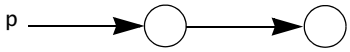

图 18.5. 发送端所看到的含有派生实例的图

但是，服务器端不理解实例的派生部分，会切掉它们。而服务器又会解编所有的客户实例，从而产生非连通的类图，如图 18.6 所示。

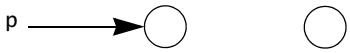

图 18.6. 图 18.5 中的图在接收端的样子

当然，还可能存在更复杂的情况，比如接收者最终得到多个非连通图，每一个图都有许多实例。

有类成员的异常

如果异常含有类成员，它的头字节 (参见第 467 页) 将是 1，而异常成员后面跟的是前面所说的待决实例，也就是说，实际的异常成员后面跟着一个或多个序列，其中含有待决的类实例。在这些序列的后面跟着一个空序列，充当结束标志。

18.2.12 代理

编码后的代理的第一个部分是一个类型为 `Ice::Identity` 的值。如果代理的值是 nil， `category` 和 `name` 成员就是空串，后面不再有其他数据。非 nil 代理由一些一般参数及一些端点参数组成。

一般的代理参数

一般的代理参数在编码时就好像它们是下面的结构的成员一样:

```
struct ProxyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    byte mode;
    bool secure;
};
```

在表 18.7 中描述了一般的代理参数。

表 18.7. 一般的代理参数

参数	描述
id	对象标识
facet	facet (或 facet 路径)
mode	代理模式 (0= 双向、 1= 单向、 2= 成批单向、 3= 数据报、 4= 成批数据报)
secure	true：需要安全端点， false：不需要

端点参数

代理可以包含一个端点列表 (参见 XREF) 或一个适配器标识符，但不能同时包含两者。

- 如果代理含有端点，它们会紧跟在一般参数后面进行编码。首先编码的是一个尺寸，说明端点的数目 (参见 18.2.1 节)，后面跟着各个端点。每个端点都被编码成一个说明端点类型的 short 值 (1=TCP, 2=SSL, 3=UDP)，后面跟着一个封装 (参见 18.2.2 节)，其中的内容是针对特定类型的参数。在后面的小节将会给出针对 TCP、 UDP，以及 SSL 的的参数。
- 如果代理没有端点，在一般参数后面会紧跟着一个值为 0 的字节，在这个字节后面会紧跟着一个进行了编码的串，表示的是对象适配器标识符。

针对特定类型的端点参数之所以被封装起来，是因为接收者可能没有能力解码它们。例如，只有当接收者被配置成要使用 SSL 插件 (参见第 23 章) 之后，它才能对 SSL 端点参数进行解码。但是，即使接收者不理解某个端点的与类型有关的参数，它要必须要能按照代理的所有端点被接收的次序，重新用这些端点对代理进行编码。把参数封装起来，接收者就可以做到这一点了。

TCP 端点参数

一个 TCP 端点被编码成一个封装，其中含有这样的结构:

```
struct TCPEndpointData {
    string host;
    int port;
    int timeout;
    bool compress;
};
```

表 18.8 描述了这些端点参数。

表 18.8. TCP 端点参数

参数	描述
host	服务器主机 (主机名或 IP 地址)
port	服务器端口 (1-65535)
timeout	供 socket 操作使用的以毫秒为单位的超时
compress	true：要使用压缩 (如果有可能)， false：不使用

关于压缩的更多信息，参见 18.4 节。

UDP 端点参数

一个 UDP 端点被编码成一个封装，其中含有这样的结构:

```
struct UDPEndpointData {
    string host;
    int port;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    bool compress;
};
```

表 18.9 描述了这些端点参数。

表 18.9. UDP 端点参数

参数	描述
host	服务器主机 (主机名或 IP 地址)
port	服务器端口 (1-65535)
protocolMajor	该端点所支持的大协议版本
protocolMinor	该端点所支持的最高的小协议版本
encodingMajor	该端点所支持的大编码版本
encodingMinor	该端点所支持的最高的小编码版本
compress	true：要使用压缩 (如果有可能)， false：不使用

关于压缩的更新信息，参见 18.4 节。

SSL 端点参数

一个 SSL 端点被编码成一个封装，其中含有这样的结构:

```
struct SSLEndpointData {
    string host;
    int port;
    int timeout;
    bool compress;
};
```

表 18.10 描述了这些端点参数。

表 18.10. SSL 端点参数

参数	描述
host	服务器主机 (主机名或 IP 地址)
port	服务器端口 (1-65535)
timeout	供 socket 操作使用的以毫秒为单位的超时
compress	true：要使用压缩 (如果有可能)， false：不使用

关于压缩的更多信息，参见 18.4 节。

18.3 协议消息

Ice 使用了五种协议消息:

- 请求 (从客户发到服务器)
- 批请求 (从客户发到服务器)
- 答复 (从服务器发到客户)
- 验证连接 (从服务器发到客户)
- 关闭连接 (从客户发到服务器，或从服务器发到客户)

在这些消息里，验证和关闭连接消息只适用于面向连接的传输机制。

和 18.2 节中描述的数据编码一样，协议消息没有对齐限制。每个消息都由一个消息头和一个紧跟其后的消息体组成（验证和关闭连接消息没有消息体）。

18.3.1 消息头

每种协议消息都有一个 14 字节的头，这个头在编码时就好像是这样的结构:

```
struct HeaderData {
    int  magic;
    byte protocolMajor;
    byte protocolMinor;
    byte encodingMajor;
    byte encodingMinor;
    byte messageType;
    byte compressionStatus;
    int  messageSize;
};
```

表 18.11 描述了这些成员。

表 18.11. 消息头成员

成员	描述
magic	一个四字节的幻数，由 'I'、 'c'、 'e'、 'P' 的 ASCII 码 (0x49, 0x63, 0x65, 0x50) 组成
protocolMajor	协议的大版本号
protocolMinor	协议的小版本号
`encodingMajor`	编码的大版本号
`encodingMinor`	编码的小版本号
`messageType`	消息类型
compressionStatus	消息的压缩状态 (参见 18.4 节)
`messageSize`	按字节计算的消息尺寸，包括头

目前，协议和编码的版本都是 1.0。表 18.12 给出了有效的消息类型。

表 18.12. 消息类型

消息类型	编码
请求	`0`
批请求	`1`
答复	`2`
验证连接	`3`
关闭连接	`4`

在接下来的几节里将描述每种消息的消息体的编码。

18.3.2 请求消息体

请求消息含有调用某个对象所必需的数据，包括对象的标识、操作名，以及输入参数。请求消息在编码时就好像是这样的结构:

```
struct RequestData {
    int requestId;
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

表 18.13 描述了这些成员。

表 18.13. 请求成员

成员	描述
requestId	请求标识符
id	对象标识
facet	facet (或 facet 路径)
operation	操作名
mode	用一个字节表示的 Ice::OperationMode (0= 普通 , 1=nonmutating, 2=idempotent)
context	调用上下文
params	封装了的输入参数，按声明时的次序排列

请求标识符零 (0) 保留用于单向请求，表明服务器不能向客户发送答复。一个非零的请求标识符必须在一个连接上唯一地标识该请求，而且，如果与该标识符对应的答复还为返回，该标识符就不能被重用。

18.3.3 批请求消息体

一个批请求消息含有一个或多个单向请求，出于效率上的考虑而捆绑在一起。批请求消息在编码时就好像是具有下列结构的序列一样:

```
struct BatchRequestData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
```

```
    byte mode;
    Ice::Context context;
    Encapsulation params;
};
```

表 18.14 描述了其中的成员

表 18.14. 批请求成员

成员	描述
id	对象标识
facet	facet (或 facet 路径)
operation	操作名
mode	用一个字节表示的 Ice::OperationMode
context	调用上下文
params	封装了的输入参数，按声明时的次序排列

注意，批请求不需要使用请求 ID，因为只有单向调用能成批发送。

18.3.4 答复消息体

一个答复消息体含有一个双向调用的结果，包括任何返回值、 out 参数，或是异常。答复消息体在编码时就好像是下面的结构一样:

```
struct ReplyData {
    int requestId;
    byte replyStatus;
    // [... messageSize - 19 bytes ...]
};
```

答复消息体的前四个字节是请求 ID。请求 ID 与某个外发的请求相匹配，请求者可以通过它来把答复和原来的请求关联起来 (参见 18.3.2 节)。

跟在请求 ID 后面的字节是请求的状态；跟在状态字节后面的其余部分取决于状态的值。表 18.15 列出了可能的状态值。

表 18.15. 答复状态

答复状态	编码
成功	0
用户异常	1
对象不存在	2
Facet 不存在	3
操作不存在	4
未知的 Ice 本地异常	5
未知的 Ice 用户异常	6
未知的异常	7

答复状态 0：成功

一个成功的答复消息被编码成一个含有 out 参数的封装 (按照声明时的次序排列)，后面跟着该调用的各个返回值，根据它们的类型、按照 18.2 节的规定进行了编码。如果操作声明了 void 返回值，并且没有 out 参数，答复消息就会编码成一个空的封装。

答复状态 1：用户异常

一个用户异常答复消息被编码成一个含有用户异常的封装，按照 18.2.10 节所描述的方式进行了编码。

答复状态 2：对象不存在

如果目标对象不存在，答复消息在编码时就好像是这样的结构：

```
struct ReplyData {
    Ice::Identity id;
    Ice::StringSeq facet;
    string operation;
};
```

表 18.16 描述了其中的成员。

表 18.16. 无效对象答复的成员

成员	描述
`id`	对象标识
`facet`	facet (或 facet 路径)
`operation`	操作名

答复状态 3：Facet 不存在

如果目标对象不支持编码在请求消息中的 facet，答复消息就会像答复状态是 2 一样编码。

答复状态 4：操作不存在

如果目标对象不支持编码在请求消息中的操作，答复消息就会像答复状态是 2 一样编码。

答复状态 5：未知的 Ice 本地异常

未知的 Ice 本地异常的答复消息会作为一个描述该异常的串进行编码。

答复状态 6：未知的 Ice 用户异常

未知的 Ice 用户异常的答复消息会作为一个描述该异常的串进行编码。

答复状态 7：未知的异常

未知的异常的答复消息会作为一个描述该异常的串进行编码。

18.3.5 验证连接消息

服务器在收到新连接时会发送一条验证连接消息[1]。这种消息表明服务器已准备好接收请求；在收到服务器发送的验证连接消息之前，客户不能在连接上发送任何消息。客户无需针对这种消息向服务器发送答复。

验证连接消息的用途有两个:

1. 验证连接消息只用于面向连接的传输机制。

- 它把服务器支持的协议和编码版本告知客户 (参见 18.5.3 节)。
- 它防止客户在服务器确认能实际处理请求之前，把请求消息写到客户本地的传输缓冲区中。这避免了一种竞争状态：在服务器关闭过程中，服务器的 TCP/IP 栈接受了它的 backlog 中的连接——如果客户在这种情况下发送请求，请求就可能会丢失，但客户却无法安全地重发请求，因为这可能会违反 “最多一次”语义。

  验证连接消息能保证，当服务器的 TCP/IP 栈接受进入的连接时，服务器没有处在关闭过程中，从而能避免这种竞争状态。

在第 486 页的 18.3.1 节描述的消息头组成了整个验证连接消息。验证连接消息的压缩状态总是 0。

18.3.6 关闭连接消息

当某一方要得体地 (gracefully) 关闭连接时就会发送关闭连接消息[2]。18.3.1 节描述的消息头组成了整个 关闭连接消息。关闭连接消息的压缩状态总是 0。

无论是客户还是服务器，都可以发起连接关闭。在客户端，连接关闭由 *Active Connection Management* (*ACM*) 触发 (参见 XREF)， ACM 会自动回收空闲了一段时间的连接。

这意味着，连接的任何一端都可以按照自己的意愿发起连接关闭；最重要的是，就对象模型或应用语义而言，没有什么状态与连接关联在一起。

只要在连接上没有在等待某个请求的答复返回，客户端就可以关闭连接。事件的序列是：

1. 客户发送关闭连接消息。
2. 客户关闭连接的写端。
3. 服务器关闭连接，对客户的关闭连接消息作出响应。

只要没有操作正在执行，服务器端就可以关闭连接。这保证了服务器不会违反 “最多一次”语义：操作一旦在 servant 中调用，就肯定可以完成，并把结果返回给客户。注意，服务器甚至可以在接收了客户发来的请求之后关闭连接，只要该请求还没有传给 servant。换句话说，如果服务器决定关闭连接，事件的序列是：

1. 服务器丢弃连接上的所有进入的请求。
2. 服务器等待仍在执行的请求完成，并把结果返回给客户。

2. 关闭连接消息只用于面向连接的传输机制。

3. 服务器向客户发送关闭连接消息。
4. 服务器关闭连接的写端。
5. 客户关闭连接的读端和写端，对服务器的关闭连接消息作出响应。
6. 如果客户在收到关闭连接请求时，仍有未得到答复的请求，它就在一个新的连接上重新发送这些请求。这样做保证了 “最多一次”语义不会被违反，因为如果在服务器端仍有请求在执行，服务器保证不会关闭连接。

18.3.7 协议状态机

从客户的视角来看， Ice 协议的行为如图 18.7 的状态机所示。

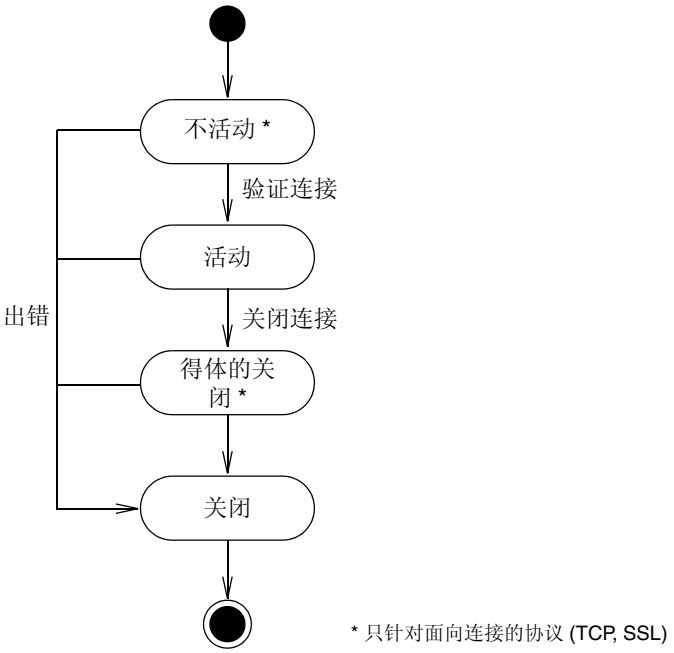

图 18.7. 协议状态机

总而言之，新连接一开始处在不活动状态，直到收到验证连接消息 (参见 18.3.5 节)，这时连接就会进入活动状态。连接保持在活动状态中，直到它被关闭，这可能会在不再有代理使用该连接时发生，也可能会在连接空

闲了一段时间之后发生。这时，连接就会得体地关闭，也就是说，发送一条关闭连接消息 (参见 18.3.6 节)，然后连接就会关闭。

18.3.8 不按次序关闭连接

任何对协议或编码规则的违反都会导致不按次序关闭连接：如果连接的某一端检测到对规则的违反，它可以强行关闭连接 （不用发送连接关闭消息或类似的消息)。有许多潜在的出错情况可能会导致连接关闭失序；例如，接收者可能检测到，某条消息的幻数是错的、或者版本不兼容，接收到的答复的 ID 与任何等待答复的请求都不匹配，在不应收到验证连接消息的情况下收到这种消息，或是在请求中发现了非法数据 (比如负的尺寸，或是与实际解编的数据量不一致的尺寸)。

18.4 压缩

压缩是 Ice 协议的一个可选特性；是否要把它用于特定的消息，取决于若干因素：

1. 并非所有语言或所有映射都支持压缩。例如， Ice for Java 目前不支持压缩。
2. 只有当代理指定要压缩时，压缩才能用于某个请求或批请求 (参见 18.2.12 节)。
3. 只有在对应的请求进行了压缩的情况下，答复消息才能进行压缩。
4. 出于效率上的考虑， Ice 协议引擎不压缩小于 100 字节的消息。

如果要进行压缩，除了头以外的整个消息都会使用 bzip2 算法进行压缩 [16]。因此，消息头的 `messageSize` 成员反映的是压缩后的消息的尺寸，再加上未压缩的头。

消息头的 compressionStatus 字段 (参见 18.3.1 节) 指示消息是否进行了压缩，以及发送者是否接受压缩的答复，如表 18.17 所示。

表 18.17. 压缩状态值

答复状态	编码	适用于
消息未压缩，发送者不能接受压缩答复。	0	请求、批请求、答复、验证连接、关闭连接
消息未压缩，发送者可以接受压缩答复。	1	请求、批请求
消息是压缩的，发送者可以接受压缩答复。	2	请求、批请求、答复

- 压缩状态 0 指示，消息未压缩，而且，这个消息的发送者不能接受压缩答复。不支持压缩的客户总是使用这个值。如果用以分派请求的端点指示它不支持压缩，支持压缩的客户也把该值设成 0。

  服务器把这个值用于未压缩的答复。

- 压缩状态 1 指示，消息未压缩，但服务器可以返回压缩答复。如果用以分派请求的端点指示它支持压缩，但客户决定不对这个请求进行压缩 (可能是因为请求太小，压缩并不能节省空间)。

  这个值只适用于请求和批请求消息。

- 压缩状态 2 指示，消息进行了压缩，服务器也可以用压缩消息来发送答复 (但不一定非如此不可)。(显然) 只有当用以分派请求的端点指示它支持压缩时，支持压缩的客户才设置这个值。

  服务器把这个值用于压缩答复。

压缩的请求、批请求或答复消息的消息体由从 18.3.2 到 18.3.4 的编码方式的压缩版本组成。

注意，只有在速度较低的链接上，压缩才有可能提高性能；在这样的链接上，带宽是主要的限制因素。在高速的 LAN 链接上，花费在消息压缩和解压上的 CPU 时间要比直接发送未压缩数据的时间更长。

18.5 协议和编码版本

我们在前面几节已经看到， Ice 协议和编码有各自的大版本号和小版本号。把协议和编码的版本机制分开，其优点是它们互不依赖：Ice 协议的任何版本都可以和编码的任何版本一起使用，所以它们可以相互独立地演化（例如， Ice 协议版本 1.1 可以使用编码版本 2.3，反之亦然）。

Ice 的版本机制在使用不同的 Ice run time 版本的客户和服务器之间提供了最大的互操作可能性。特别地，只要消息内容使用的是双方都理解的类型，以前部署的客户可以与最近部署的服务器通信，反之亦然。

例如，假定后来的某个 Ice 版本要引入一个新的 Slice 关键字和数据类型，比如 `complex`，用于表示复数。这要求我们为编码增加一个新的小版本号；让我们假定，编码的 1.1 版与 1.0 版是一样的，但另外还支持 `complex` 类型。现在，我们的客户和服务器编码版本有了四种可能的组合：

表 18.18. 不同版本的互操作性

客户版本	服务器版本	有 complex 参数的操作	没有 complex 参数的操作
1.0	1.0	N/A	4
1.1	1.0	N/A	4
1.0	1.1	N/A	4
1.1	1.1	4	4

你可以看到，互操作性已经达到了最大限度。如果客户和服务器的版本都是 1.1，它们显然可以交换消息，并使用编码版本 1.1。对于 1.0 版的客户和服务器，显然只有不涉及 `complex` 参数的操作才能被调用（因为至少有一个客户和服务器不知道新的 `complex` 类型），而消息使用编码版本 1.0 进行交换。

18.5.1 版本基本规则

要使协议和编码的版本机制成为可能， Ice run time 的*所有*版本（现在的和将来的）都必需遵循一些基本规则：

1. 封装的头总是六字节；前四个字节是封装的尺寸 (包括头 的尺寸)，后面跟的两个字节是大小版本 。怎样解释封装的其余内容取决于大小版本。
2. 消息头的前八个字节总是幻数 'I'、 'c'、 'e'、 'P'，再加上四个字节的版本信息 (两个字节的协议大小版本号，两个字节的编码大小版本号)。怎样解释头的其余内容取决于大小版本。

这两个基本规则保证了，所有当前的和未来的 Ice run time 版本都至少能确定封装和消息的版本及尺寸。这对于像 IceStorm(参见第 26 章) 这样的交换服务而言特别重要 ; 通过使版本和尺寸信息保持固定的格式，（举个例子）能够通过版本仍是 1.0 的消息交换服务来转发版本 2.0 的消息。

18.5.2 版本兼容规则

要想确定某个协议版本是否与另一个协议版本兼容 (或某个编码版本是否与另一个编码版本兼容)，要考察以下规则:

1. 不同的大版本是不兼容的。客户和服务器都没有义务支持一种以上的大版本。例如，大版本是 2 的服务器完全没有义务支持大版本 1。

   这条规则之所以存在，是为了让 Ice run time 能最终摆脱旧版本——没有这样的规则，未来的所有版本的 Ice 就必须支持永远支持先前的所有大版本。用 简明的话说，使用不同大版本的客户和服务器不能相互通信。
2. 如果一个接收者说自己能支持小版本 n，它就肯定能成功解码所有小于 n 的小版本。注意，这*并非*是说，使用版本 $n-1$ 的消息能被当成版本 n 的消息进行解码：就它们的物理表示而言，两个相邻的小版本可能完全不兼容。但是，因为任何说自己支持版本 n 的接收者也有义务正确处理版本 $n-1$，小版本升级*在语义上*是向后兼容的，即使它们的物理表示可能并不兼容。
3. 支持小版本 n 的发送者保证能使用所有小于 n 的小版本来发送消息。而且，发送者还保证 ，如果它收到的请求使用的是小版本 k ($k \leq n$)，它将用小版本 k 来发送对该请求的答复。

18.5.3 版本磋商

客户和服务器必须以某种方式、就使用哪个版本来交换消息达成一致。取决于底层传输机制是面向连接的还是无连接的，用于磋商共同版本的机制也不同。

为面向连接传输机制进行磋商

在使用面向连接的传输机制时，客户打开一个通向服务器的连接，然后等待验证连接消息返回 (参见第 491 页)。服务器发送的验证连接消息会指示服务器的协议和编码的大版本号、以及所支持的最高小版本号。如果服务器和客户的大版本号不匹配，客户端将引发 `UnsupportedProtocolException` 或 `UnsupportedEncodingException`。

假定客户收到的来自服务器的验证连接消息与客户的大版本是匹配的，客户就知道了服务器所支持的最高的小版本号。从此以后，客户有义务不发送小版本号更高的消息。但是，服务器可以发送小版本号更低的消息。

服务器一开始不知道客户所支持的最高小版本 (因为客户不会发送验证连接消息给服务器)。相反，服务器会查看消息头，了解每个消息中的客户版本号。这个小版本号指示的是客户所能接受的小版本号，其作用域是一次请求 - 答复交互过程。例如，如果客户发送了一个小版本为 3 的请求，服务器必须也用小版本 3 发送答复。但是，下一次客户请求也许会使用小版本 2，服务器也必需用小版本 2 来答复该请求。

在通过关闭连接消息进行有序的连接关闭时，服务器可以使用任何小版本，但这个小版本不能高于此前从客户那里所接收到最高的小版本号。

为无连接传输机制进行磋商

在使用无连接的传输机制时，不存在验证连接消息，所以客户必须通过其他手段来了解服务器所支持的最高小版本号。采用何种手段取决于无连接端点是直接绑定的还是间接绑定的 (参见 20.3.2 节):

- 对于直接代理，版本信息是代理中所包含的端点的一部分。在这种情况下，客户在发送消息时，使用的小版本号不高于代理中的端点的小版本号。
- 对于间接代理，代理自身完全不含版本信息 (因为代理不含端点)。相反，客户会在把适配器名解析成一个或多个端点时获取版本信息 (通过 IcePack 或等价的服务)。端点的版本信息将决定客户可以使用的最高小版本号。

18.6 与 IIOP 的对比

对 Ice 的协议及编码和 CORBA 的 Inter-ORB Interoperability Protocol (IIOP) 及 Common Data Representation (CDR) 编码进行对比，是一件有意思的事情。 Ice 协议及编码在许多重要方面都与 IIOP 及 CDR 不同:

- 数据类型更少

  CORBA IDL 的有些数据类型在 Ice 中没有提供，比如字节和宽字符、定点数、数组 、联合，这些数据类型都需要有自己的编码规则。

  显然，因为 Ice 的数据类型更少，所以它比 CORBA 更高效，复杂度也更低。

- 固定的字节序整编

  CDR 既支持 "big-endian" 编码，也支持 "little-endian" 编码，并且提供了一种 “由接收者负责使其正确” 的字节序处理方法。其优点是，如果发送者和接收者的字节序是一样的，那么两端都不需要调整数据的次序。但是，其缺点是这种方案会在整编逻辑中制造额外的复杂性。而且，如果消息要通过许多中间节点进行转发，这种方案会付出严重的性能代价，因为消息可能需要反复进行解编、调整次序，重整编 (封装数据可以避免这个问题，但遗憾的是，IIOP 没有封装大多数本可以由此获益的数据)。

  Ice 编码使用的是固定的 "little-endian" 数据布局，从而避免了付出复杂性和性能方面的代价。

- 不进行填充

  CDR 有一组复杂的规则，用于把对齐字节插入数据流中，让数据以某种方式进行对齐，从而适应底层应用的原生数据布局。遗憾的是，由于一些原因，这种做法有严重的缺陷:

  - CDR 填充规则实际上并没有获得任何自然的字边界布局。例如，取决于同样的结构值在它所属的数据流中的位置，这个值的尺寸可能会发生变化，并最多填充七个字节。在现有的硬件平台中，没有一种具有与此类似的布局规则。结果，除了浪费带宽，所插入的填充字节没有任何用处。
  - 在使用 “由接收者负责使其正确” 方案时，如果到达的数据的字节序不正确，接收者要负责调整数据的次序。这意味着，对大约一半的接收者而言，数据的对齐 （即使是正确的）是没有用的，因为调整数据的次序无论如何都需要复制所有数据。
  - 取决于硬件平台以及编译器的不同，填充和对齐规则也有所不同 （例如，为了让开发者平衡时间和空间开销，许多编译器都提供了一些选项，用以改变数据在内存中的打包方式）。这意味着，即使是最好的布局规则，也只适合少数平台。
  - 数据对齐规则使得数据转发变得十分复杂。例如，如果某个数据项的接收者想要把它转发给另一个下游接收者，它不能简单地把接收到的数据块复制到它的传送缓冲区中，因为数据的填充方式会因它在所属

字节流中的位置而发生变化，如果你进行块复制，很可能会产生非法的编码。

通过在字节边界上对齐所有数据， Ice 避免了所有这些复杂性（以及后继的低效）。

- 更紧凑的编码

CDR 编码会浪费带宽，特别是在对短数据项的序列进行编码时。例如，使用 CDR 编码，一个空串会占用八个字节：四个字节存储串长，一个字节用于结尾的 NUL 字节，再加上三个填充字节。表 18.19 对比了使用 CDR 和 Ice、对 100 个长度不同的串的序列进行编码 的编码尺寸。

表 18.19. 100 个长度不同的串的 CDR 和 Ice 序列的尺寸

100 个长度为……的串的序列	CDR 尺寸	Ice 尺寸	CDR 的额外开销
0	804	101	696%
4	1204	501	140%
8	1604	901	78%
16	2004	1701	18%

小结构也能获得类似的节省。取决于结构成员的次序和类型， CDR 填充字节可能会使结构尺寸几乎翻倍，在发送这样的结构的序列时，这样的翻倍影响重大。

- 简单的代理编码

由于 CORBA 供应商无法就对象引用（Ice 代理的等价物）的编码达成一致， CORBA 对象引用的内部结构很复杂，部分进行了标准化，部分是不透明的，这样，供应商才能够给对象引用添加私有信息。此外，为了避免对象引用变得太大，支持多种传输机制的引用使用了一种方案，在若干传输机制间共享对象标识，而不是携带同一标识的多个副

本。支持所有这些机制所需的编码相当复杂，结果在交换大量对象引用时（例如，在使用 trading 服务的情况下），整编性能极其低下。

与此相反， Ice 代理的整编简单、直截了当，不会造成这样的性能下降。

- 适当的版本磋商

  IIOP 和 CDR 的版本机制从未得到适当设计，造成的结果是， IIOP 中的版本磋商完成无法工作。特别地， CDR 封装没有携带单独的版本号。结果，封装的数据有可能会传送到不能正确解码封装的内容的接收者那里，而在协议级没有机制可用于检测该问题。多年来，缺少正确的版本机制一直是 CORBA 的一个问题，而在历史上，这个问题的处理办法一直是，假装它并不存在（也就是说，在许多情况下，不同的 CORBA 版本不能进行互操作）。

  Ice 协议和编码有着明确定义的版本规则，能够避免这样的问题，从而使得协议和编码都能够进行扩展，并且能可靠地检测版本失配。

- 消息类型更少

  IIOP 的消息类型比 Ice 要多。例如， IIOP 既有取消请求消息，也有消息错误消息。取消请求的用途是取消正在执行的调用，但那当然无法做到，因为没有什么办法能中止正在服务器中执行的调用。取消请求最多能让服务器不要再整编调用的结果发回客户。但是，这并不值得在协议中引入那么多额外的复杂性（而且，无论如何，应用开发者既无法发送取消请求， run time 也不能自行决定何时发送取消请求才合适；尽管如此，每一个遵从标准的 CORBA 实现都要承受负担，正确地响应一个无用的请求）。

  IIOP 的消息错误消息也有类似的负担：接收到有问题的消息的接收者有义务在关闭其连接之前、发送一个消息错误消息作为响应。但是，这种消息没有携带有用的信息，而且，由于 TCP/IP 实现的本质，常常会丢失，而不是到达对端。这意味着，一个遵从标准的 CORBA 实现会被迫发送一条无用的消息（它本应该简单地关闭连接），在大多数情况下，这条消息不会被收到。

  在 Ice 协议中没有这样的负担：所用的消息都的确是有用的。

- 批请求

  IIOP 没有批请求的概念。对于像 IceStorm (参见第 26 章) 这样的事件转发机制，以及为许多属性提供修改器操作的细粒度接口，批请求的好处特别值得一提。对于这样的应用，批请求能够显著地降低网络开销。

- 可靠的连接建立

  IIOP 很容易受到连接建立过程中发生的连接丢失的影响。特别地，当客户打开一个通向服务器的连接时，客户无法知道服务器是否正要关闭、将无法处理进入的请求。这意味着，客户别无选择，只能在新建立的连接上发送一个请求，希望这个请求能被服务器实际处理。如果服务器能够处理该请求，没有问题；但如果服务器因为正在关闭而无法处理该请求，客户面临的就是一个坏掉的连接，并且不能再重发该请求，因为那可能会违反 “最多一次”语义。

  Ice 没有这个问题，因为验证连接消息保证了客户不会把请求发给正要关闭的服务器。而且，客户可以安全地重发任何没有得到答复的请求，而不会违反 “最多一次”语义。

- 可靠的端点解析

  IIOP 中的间接绑定 (参见 20.3.2 节) 依赖于定位转发答复。简要地说，端点解析对于使用 IIOP 的客户而言是透明的。如果对象引用使用了间接绑定，客户像平常一样发出请求，将收到一个定位转发答复，其中含有实际服务器的端点。客户对该请求作出响应，再次发出请求，联系实际的服务器。这种方案有一些问题：

  - 定位服务的物理地址是写在每个间接绑定的对象引用中的。这样，如果定位服务的地址变了，客户持有的所有引用都会失效[3]。

    Ice 没有这样的问题，因为客户是通过配置获得定位服务器的地址的。如果定位服务的地址变了，可以通过改变客户的配置来更新客户，无需去追踪那些可能已失效的、数目不定的代理。而且，定位服务通常没有必要改变地址，你可以构造与 Internet Domain Name Service (DNS) 类似的定位服务联盟，在内部把请求转发给正确的解析器。

  - 为了获得定位转发消息，客户要把实际的请求发送给定位服务。如果请求参数很多，这就会很昂贵，因为这些参数会被整编两次：一次是给定位服务，一次是给实际的服务器。 IIOP 增加了一个定位请求消息，允许客户显式地解析服务器的地址。但是， CORBA 对象引用没有携带什么指示，说明它们是直接绑定的，还是间接绑定的。这意味着，不管客户做什么，总有一些时候是错的：如果在使用直接绑定的引用时，客户总是使用定位请求，它最后就会付出发送两个、而不是

3. IIOP 1.2 版支持一种指示永久的定位转发的消息。这种消息意在使定位服务的迁移变得轻松一点。但是，该消息的语义在别处破坏了对象模型，致使 IIOP 1.3 版不赞成使用该消息 (遗憾的是，像这样的反复无常在 OMG 规范中太常见了)。

一个远地消息的代价；如果在使用间接绑定的引用时，客户总是直接发送请求，它就会付出两次、而不是一次整编参数的代价。

Ice 让定位解析变成了客户端 run time 可以明确看到的一个步骤，因为代理不是携带了端点信息 (直接代理)，或者携带了适配器名 (间接代理)。这样，客户端 run time 就能够选择正确的解析机制，而不必进行猜测游戏，从而避免定位转发所造成的开销。

- 在 IIOP 中，定位解析被构建进协议自身之中。这使得协议变复杂了，多了两种消息类型，而且，也造成实现者不可能使用标准 API 来实现定位服务（直到 2002 年， CORBA 增加了另外的 API 之后，这一局面才改变）。

  在 Ice 中，定位解析无需特殊的支持。相反，定位服务是一个普通的 Ice 服务器，和其他服务器一样，它定义了一个接口，要访问它也是使用操作调用。

- 没有代码集磋商

  IIOP 使用了一种代码集磋商机制，（大概）允许 开发者使用任意的字符编码来传送宽字符和宽串，前提是发送者和接收者至少拥有一个共同的 代码集。遗憾的是，这种特性从未得到过适当的思考，而且反复地造成了各种互操作问题（CORBA 规范的每一个版本，包括最新的 3.0 版，都对字符集磋商的工作方式进行了更正。最新的更正是否能成功处理那些问题，还有待于时间来证明。无论如何，为了这个（成问题的）特性， CORBA 规范用了许多页复杂的文档（以及许多行 ORB 源码）来加以处理。

  Ice 使用的以 UTF-8 方式编码的 Unicode 避免了所有这些问题，而且没有损失任何功能。

- 没有分段

  IIOP 提供了一个分段消息，其目的是，在整编操作调用的结果的过程中，降低服务器的内存开销。遗憾的是，这个特性相当复杂（在过去被反复地错误规定和实现）。通过分段所节省的内存相当有限，而每一个客户端 ORB 实现都被迫为这个特性提供支持（换句话说，治疗比疾病还要糟）。

  Ice 没有使用分段方案，既避免了复杂性，也避免了代码膨胀。

- 支持无连接的传输机制

  在 2001 年， OMG 采用了一种多播规范。就我们所知，到 2003 年初，还没有该规范的实现可用。

  Ice 支持 UDP，作为 TCP/IP 的替代品。对于像 IceStorm 这样的消息服务而言， UDP 在性能上的优越性相当可观，所以这样的服务的可伸缩性可以远远超过 TCP/IP。

第 19 章
Ice 的 PHP 扩展

19.1 本章综述

这一章将描述 IcePHP， Ice 的 PHP 脚本语言扩展。 19.2 节对 IcePHP 进行了综述，包括它的设计目标、能力，以及局限。 19.3 节讨论 IcePHP 的配置， 19.4 节规定了 PHP 语言映射。

19.2 引言

PHP 是一种通用的脚本语言，主要用于 Web 开发。 PHP 解释器通常是作为 Web 服务器插件安装的， PHP 自身也支持称为 “扩展” 的插件。按照其定义，PHP 扩展的用途是通过增加新的函数和数据类型，对解释器的 run time 环境进行扩展。

Ice 的 PHP 扩展 (*IcePHP*) 向 PHP 脚本提供了访问各种 Ice 设施的能力。IcePHP 是一个瘦集成层，用 Ice 的 C++ run time 库实现。与原生的 PHP 实现相比，这种实现计数有一些优点:

1. 速度

   在进行远地调用时，大部分耗时的工作，比如整编和解编，都是通过编译方式的 C++ 代码、而不是解释方式的 PHP 代码来完成的。

2. 集成

   IcePHP 完全是自包含的。它的安装是作为一个管理步骤一次完成的，脚本不依赖外部的 PHP 代码。

3. 可靠性

   通过利用进行了详尽测试的 Ice C++ run time 库，在 PHP 扩展中引入新 bug 的可能性很小。

4. 灵活性

   IcePHP 继承了 Ice C++ run time 所提供的所有灵活性，比如 SSL 支持、协议压缩，等等。

19.2.1 能力

IcePHP 提供了一组健壮的 Ice run time 设施。 PHP 脚本能够以自然的方式使用所有 Slice 数据类型 (参见 19.4 节)、进行远地调用，以及使用高级的 Ice 服务，比如路由器、定位器，以及协议插件。

19.2.2 局限

IcePHP 的主要设计目标是，向 PHP 脚本提供一个简单而高效的接口，用以访问 Ice run time。因此， IcePHP 所支持的特性集经过了精心挑选，以满足典型的 PHP 应用的需要。所以 IcePHP 不支持以下 Ice 特性:

- 服务器

  因为 PHP 的主要角色是用于动态 Web 页面的脚本语言，对于大多数 PHP 应用而言，用 PHP 实现 Ice 服务器的能力并无必要。

- 异步方法调用

  PHP 缺少同步原语，这极大地降低了异步调用的效用。

- 多通信器

  一个脚本只能访问 `Ice::Communicator` 的一个实例，而且不能手工创建或销毁通信器。更多信息，参见 19.4.11 节。

19.2.3 设计

传统的语言映射的设计要求采用一个中间步骤，把 Slice 定义翻译成目标编程语言，然后才能把它们用在应用中。

由于 PHP 的扩展接口很灵活， IcePHP 采用了一种不同的途径。在 IcePHP 中，不需要生成中间代码。相反，该扩展会用应用的 Slice 定义进行配置，把这些定义用于驱动扩展的运行时行为（参见 19.3 节）。

Slice 定义会按照 19.4 节规定的映射方式，提供给 PHP 脚本使用，就好像 Slice 定义首先经过了传统的代码生成步骤、然后由脚本导入一样。

这种设计有若干好处:

- 因为消除了中间的代码生成步骤，所以开发过程得到了简化。
- 因为没有机器生成的 PHP 代码，所以应用的类型定义过时的风险降低了。
- 尽管是一种类型松散的编程语言，通过 Slice 定义，IcePHP 能够验证远地调用的参数。

19.3 配置

这一节将定义 IcePHP 所支持的 PHP 配置指令。要了解安装指令，请查看 IcePHP 包里所包含的 `INSTALL` 文件。

19.3.1 Profile

IcePHP 允许任意数目的 PHP 应用在同一 PHP 解释器中相互独立地运行，不会因 Slice 定义碰巧使用了同样的标识符而发生冲突。 IcePHP 使用了术语 *profile* 来描述应用的配置，包括它的 Slice 定义和 Ice 配置属性。

19.3.2 缺省 Profile

IcePHP 支持缺省 profile，在开发过程中、或者在 PHP 解释器中只有一个 Ice 应用在运行时，这十分方便。表 19.1 描述了用于缺省 profile 的 PHP 配置指令[1]。

表 19.1. 用于缺省 profile 的 PHP 配置指令

名字	描述
`ice.config`	指定 Ice 配置文件的路径名。

表 19.1. 用于缺省 profile 的 PHP 配置指令

名字	描述
`ice.options`	指定用作 Ice 配置属性的命令行选项。例如，`--Ice.Trace.Network=1`。如果要使用的配置属性不多，这个指令可以方便地替代 Ice 配置文件。
`ice.profiles`	指定 profile 配置文件的路径名。参见 19.3.3 节。
`ice.slice`	指定预处理器选项，以及要加载的 Slice 文件的路径名

下面是一个简单的例子:

```
ice.options="--Ice.Trace.Network=1 --Ice.Warn.Connections=1"
ice.slice="-I/myapp/include /myapp/include/MyApp.ice"
```

19.3.3 有名字的 Profile

如果 `ice.profiles` 配置指令指定了一个文件名，这个文件应该具有标准的 INI 文件格式。区名 (section name) 标识 profile，在其中可以使用 19.3.2 节定义的 `ice.config`、 `ice.options` 及 `ice.slice` 指令。例如:

```
[Profile1]
ice.config=/profile1/ice.cfg
ice.slice=/profile1/App.ice

[Profile2]
ice.config=/profile2/ice.cfg
ice.slice=/profile2/App.ice
```

这个文件定义了两个有名字的 profile: `Profile1` 和 `Profile2`，它们有不同的 Ice 配置和 Slice 定义。

1. 这些指令通常在 `php.ini` 文件中定义，但也可以用 Web 服务器特有的指令定义。

19.3.4 Profile 函数

IcePHP 提供了两个用于 profile 的全局函数:

```
Ice_loadProfile(/* string */ $name = null);
Ice_dumpProfile();
```

脚本必须调用 `Ice_loadProfile` 函数获得 Slice 类型，并配置该脚本的通信器 。如果 profile 名没有提供，就会加载缺省的 profile。脚本不允许加载多个 profile。

例如，下面的脚本加载 19.3.3 节给出的 `Profile1`:

```
<?php
Ice_loadProfile("Profile1");
...
?>
```

为了进行故障诊断，脚本可以调用 `Ice_dumpProfile` 函数。这个函数显示与脚本加载的 profile 相关的所有信息，包括通信器的配置属性，以及这个 profile 加载的所有 Slice 定义的 PHP 映射。

19.3.5 Slice 语义

Slice 文件的解析是在 PHP 解释器初始化 IcePHP 扩展时、一次性地进行的。因此，我们建议你把 IcePHP 扩展静态地配置进 PHP 解释器中：或者直接把扩展编译进解释器中，或者配置 PHP、让它在启动时加载扩展。因为同样的原因，我们建议你不要在 CGI 上下文中使用 IcePHP，在这种情况下，每有一个 HTTP 请求，都会创建一个新的 PHP 解释器。此外，我们强烈地建议你不要用 PHP 的 `dl` 函数加载 IcePHP 扩展。

19.4 客户端的 Slice-to-PHP 映射

这一节描述各种 Slice 类型的 PHP 映射。

19.4.1 标识符的映射

Slice 标识符映射到名字相同的 PHP 标识符，除非 Slice 标识符与某个 PHP 保留字相冲突，在这种情况下，映射后的标识符会加上一个下划线。例如， Slice 标识符 `echo` 会被映射成 `_echo`。

在 Slice 模块中定义的标识符会使用 “变平的” (flattened) 标识符，因为 PHP 没有 C++ 名字空间或 Java package 的等价成分。变平映射使用下划线来分隔 fully-scoped 名字的各个部分。例如，考虑下面的 Slice 定义:

```
module M {
    enum E { one, two, three };
};
```

在这种情况下， Slice 标识符 `M::E` 变平成了 PHP 标识符 `M_E`。

注意，在检查是否与 PHP 保留字有冲突时，所考虑的只是 fully-scoped、变平的标识符。例如，尽管 `function` 是 PHP 保留字， Slice 标识符 `M::function` 仍会被映射到 `M_function`。它不需要被映射成 `M__function` (有两个下划线)，因为 `M_function` 和 PHP 保留字没有冲突。

但是，变平映射仍有可能生成与 PHP 保留字有冲突的标识符。例如，为了避免与 PHP 保留字 `require_once` 发生冲突， Slice 标识符 `require::once` 必须映射到 `_require_once`。

19.4.2 简单的内建类型的映射

PHP 有一组有限的原始类型: `boolean`、 `integer`、 `double`，以及 `string`。 Slice 内建类型与 PHP 类型的映射如表 19.2 所示。

Table 19.2. 把 Slice 内建类型映射到 PHP.

Slice	PHP
`bool`	`boolean`
`byte`	`integer`
`short`	`integer`
`int`	`integer`
`long`	`integer`
`float`	`double`
`double`	`double`

Table 19.2. 把 Slice 内建类型映射到 PHP.

Slice	PHP
`string`	`string`

PHP 的 `integer` 类型可能不能容纳 Slice 的 `long` 类型的值的范围，因此，超出此范围的 `long` 值会被映射成串。对于返回 `long` 值的操作，脚本必须准备好接收整数或串。

19.4.3 用户定义的类型的映射

Slice 支持用户定义的类型：枚举、结构、序列，以及词典。

枚举的映射

枚举映射到 PHP 的类，其中含有针对每个枚举符的常量定义。例如：

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

下面是相应的 PHP 映射：

```
class Fruit {
    const Apple = 0;
    const Pear = 1;
    const Orange = 2;
}
```

可以用像 `Fruit::Apple` 这样的 PHP 代码来访问各个枚举符。

结构的映射

Slice 结构映射到 PHP 的类。对于每一个 Slice 数据成员，对应的 PHP 类都含有一个同名的变量。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;
};
```

这个结构被映射到下面的 PHP 类：

```
class Employee {
    var $number;
    var $firstName;
    var $lastName;
}
```

序列的映射

Slice 序列映射到原生的 PHP 有索引数据。 Slice 的第一个元素位于 PHP 数组的索引 0（零）处，后面是按 上升的索引序排列的其余元素。

下面是 4.7.3 节的 FruitPlatter 序列的定义:

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

你可以这样创建这个序列的实例:

```
// Make a small platter with one Apple and one Orange
//
$platter = array(Fruit::Apple, Fruit::Orange);
```

词典的映射

Slice 词典映射到原生的 PHP 关联数组。但是， PHP 映射目前不支持所有的 Slice 词典类型，因为原生的 PHP 关联数组只支持整数和串键类型。键类型为 boolean、 byte、 short、 int 或 long 的 Slice 词典被映射成具有整数键的关联数组[2]。键类型为串的 Slice 词典被映射成具有串键的关联数据。所有其他的键类型都会导致 IcePHP 生成一条警告信息。

下面是 4.7.4 节的 EmployeeMap 的定义:

```
dictionary<long, Employee> EmployeeMap;
```

你可以这样创建这个词典的实例:

```
$e1 = new Employee;
$e1->number = 42;
$e1->firstName = "Stan";
$e1->lastName = "Lipmann";

$e2 = new Employee;
$e2->number = 77;
```

2. 布尔值被当作整数，假等价于 0 (零)，真等价于 1 (一).

```
$e2->firstName = "Herb";
$e2->lastName = "Sutter";

$em = array($e1->number => $e1, $e2->number => $e2);
```

19.4.4 常量的映射

Slice 常量定义映射到对相应的 PHP 函数 define 的调用。下面是我们在 4.7.5 节看到过的常量定义:

```
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;
```

下面是与这些常量等价的 PHP 定义:

```
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 异常的映射

Slice 异常映射到 PHP 的类。对于每一个异常成员，对应的类都含有一个同名的变量。所有的用户异常最终都派生自 Ice_UserException(这个类派生自 Ice_Exception，而后者又派生自 PHP 的 Exception 基类):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }
```

```
}

abstract class Ice_UserException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

构造器的可选串参数会被不加修改地传给 Exception 构造器。

如果异常派生自一个基异常，对应的 PHP 类就会从为该基异常映射的类派生。而如果没有指定基异常，对应的类就会从 Ice_UserException 派生。

下面是 4.8.5 节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 {};
```

这些异常被映射成下面的 PHP 类:

```
class GenericError extends Ice_UserException {
    function __construct($message = '') {
        ...
    }

    var $reason;
}

class BadTimeVal extends GenericError {
    function __construct($message = '') {
        ...
    }
}

class BadZoneName extends GenericError {
    function __construct($message = '') {
        ...
    }
}
```

应用可以这样来捕捉这些异常:

```
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 运行时异常的映射

The Ice run time 会因为一些预先定义的错误情况而抛出运行时异常。所有的运行时异常都直接或简单地从 `Ice_LocalException` 派生 (这个异常派生自 `Ice_Exception`，而后者又派生自 PHP 的 `Exception` 基类):

```
abstract class Ice_Exception extends Exception {
    function __construct($message = '') {
        ...
    }
}

abstract class Ice_LocalException extends Ice_Exception {
    function __construct($message = '') {
        ...
    }
}
```

第 80 页的图 4.4 给出了用户异常和运行时异常的继承图。但要注意，PHP 映射只为下面列出的本地异常定义了类:

- `Ice_LocalException`
- `Ice_UnknownException`
- `Ice_UnknownLocalException`
- `Ice_UnknownUserException`
- `Ice_RequestFailedException`
- `Ice_ObjectNotExistException`
- `Ice_FacetNotExistException`
- `Ice_OperationNotExistException`
- `Ice_ProtocolException`
- `Ice_MarshalException`

- Ice_NoObjectFactoryException

其他的本地异常的实例会被转换成 Ice_UnknownLocalException。这个类的 unknown 成员含有原来的异常的串表示。

19.4.7 接口的映射

Slice 接口映射到 PHP 的接口。接口中的每个操作都会按照 19.4.9 节所描述的方式，映射到一个同名的方法。Slice 接口的继承结构在 PHP 映射中得到了保留，所有的接口最终都派生自 Ice_Object:

```
interface Ice_Object {};
```

例如，考虑下面的 Slice 定义:

```
interface A {
    void opA();
};
interface B extends A {
    void opB();
};
```

这些接口被映射到这样的 PHP 接口:

```
interface A implements Ice_Object
{
    function opA();
}
interface B implements A
{
    function opB();
}
```

19.4.8 类的映射

Slice 类映射到抽象的 PHP 类。类中的每个操作都会按照 19.4.9 节所描述的方式，映射到一个同名的抽象方法。对于每个 Slice 数据成员，对应的 PHP 类都含有一个同名的变量。Slice 类的继承结构在 PHP 映射中得到了保留，所有的类最终都派生自 Ice_ObjectImpl（这个类又实现了 Ice_Object):

```
class Ice_ObjectImpl implements Ice_Object
{
    var $ice_facets = array();
}
```

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
};
```

下面是这个类的 PHP 映射:

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    var $hour;
    var $minute;
    var $second;
    abstract function format();
}
```

Facet

如第 522 页上的 `Ice_ObjectImpl` 的映射所示， Ice 对象的 facet 由原生的 PHP 数组表示。更具体地说，各个 facet 包含在 `ice_facets` 成员中，这是一个关联数组，键类型是 `string`，值类型是 `Ice_Object`。`ice_facets` 是 public 成员，因此可以由应用直接操纵。

类工厂

要安装类工厂，要调用通信器的 `addObjectFactory`（参见 19.4.11 节）。工厂必须实现 `Ice_ObjectFactory` 接口，其定义如下:

```
interface Ice_ObjectFactory implements Ice_LocalObject
{
    function create(/* string */ $id);
    function destroy();
}
```

例如，我们可以这样为 `TimeOfDay` 类定义并按照一个工厂:

```
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 操作的映射

在 Slice 类或接口中定义的每个操作都会映射到一个同名的 PHP 函数。此外，操作的每个参数都被映射到一个同名的 PHP 参数，如果是 out 参数，就通过引用传递。因为 PHP 是一种类型松散的语言，所以不会指定参数类型[3]。

考虑下面的接口：

```
interface I {
    float op(string s, out int i);
};
```

下面是这个接口的 PHP 映射：

19.4.10 代理的映射

Slice 代理的 PHP 映射使用了一个接口 Ice_ObjectPrx 来表示所有代理类型。

有类型的 vs. 无类型的代理

代理可以是有类型的或无类型的。所有代理，不管是有类型的还是无类型的，都支持下一节描述的核心代理方法。

*无类型的代理*与 Slice 类型 Object* 等价。通信器的 stringToProxy 操作、以及其他一些核心代理方法返回的是无类型的代理。脚本不能在无类

3. PHP5 引入了“类型提示”(type hint) 的概念，允许你指定对象参数的形式类型。这使得 Slice 映射能够为结构、接口、类、代理类型的参数指定类型提示 。遗憾的是， PHP5 目前不允许有类型提示的参数接收 null 值，因此 Slice 映射没有给参数指定类型提示。

型的代理上调用用户定义的操作，无类型的代理也不能作为参数传给需要有类型代理的地方。

*有类型的代理*是已经与某个 Slice 类或接口类型关联在一起的代理。脚本可以通过两种方式获取有类型的代理:

1. 从返回有类型代理的远地调用那里接收。
2. 使用核心代理方法 ice_checkedCast 或 ice_uncheckedCast。

例如，假定我们的脚本需要获取下面的接口 A 的有类型代理:

```
interface A {
    void opA();
};
```

下面是我们的脚本执行的步骤:

```
$obj = $ICE->stringToProxy("a:tcp -p 12345");
$obj->opA(); // WRONG!
$a = $obj->ice_checkedCast("::A");
$a->opA(); // OK
```

试图在 $obj 上调用 opA 会产生致命的错误，因为 $obj 是无类型的代理。

核心代理方法

Ice_ObjectPrx 接口提供了一些核心代理方法:

```
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);
}
```

这些方法可分类如下：

- 远地审查：这一类方法会返回关于远地对象的信息。
- 本地审查：这一类方法会返回关于代理的本地配置的信息。
- 工厂：这一类方法会返回用不同特性配置的新代理实例。

代理是不可变的，所以通过工厂方法，应用可以获得具有所需配置的代理。在本质上，工厂方法会克隆原来的代理，并修改新代理的一个或多个特性。

为了演示代理工厂方法的用法，考虑下面的例子：

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
$p = $p->ice_uncheckedCast("::A");
```

这个脚本收到 stringToProxy 返回的一个无类型代理。因为串化代理不含任何代理选项，它被配置成双向的不安全代理，而且没有超时。但是，我们的目标是获得接口 A 的一个安全的单向代理，因此我们在调用了 ice_oneway 之后，又调用了 ice_secure。这时，我们已经有了一个无类型的代理，为安全的单向调用作了配置。最后，我们调用 ice_uncheckedCast 获取一个有类型的代理[4]。

注意，通过改变代理工厂方法，可以简化这个过程，如下所示：

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_oneway()->ice_secure(true)->ice_uncheckedCast("::A");
```

代理工厂方法被调用的次序通常并不重要，除非是在使用 ice_checkedCast 和 ice_uncheckedCast 的情况下。例如，如果上面

4. 我们不能在为单向调用配置的代理上使用 ice_checkedCast。

的脚本先调用 ice_uncheckedCast，然后再调用其他工厂方法，那么所得到的就将是一个无类型的代理：

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$p = $p->ice_uncheckedCast("::A"); // WRONG!
$p = $p->ice_oneway();
$p = $p->ice_secure(true);
```

之所以会产生这个意外的行为，是因为大多数工厂方法都返回无类型的代理，从而会丢弃任何已经与原来的代理关联在一起的类型。因为在 ice_uncheckedCast 返回的有类型代理上调用 ice_oneway，脚本失去了它的有类型代理。

表 19.3 更详尽地解释了这些核心代理方法。

表 19.3. 对核心代理方法的描述

方法	描述	远地的
ice_isA	如果远地对象支持 id 参数所指示的类型，返回 true，否则返回 false。这个方法只能在双向代理上调用。	是
ice_ping	确定远地对象是否可到达。没有返回值。	是
ice_ids	返回远地对象所支持的各种类型的 id。返回值是一个串数组。这个方法只能在双向代理上调用。	是
ice_id	返回远地对象所支持的派生层次最深的类型的 id。这个方法只能在双向代理上调用。	是
ice_facets	返回远地对象所支持的 facet 的名字。返回值是一个串数组。这个方法只能在双向代理上调用。	是
ice_getIdentity	返回与代理相关联的 Ice 对象的标识。返回值是一个 Ice_Identity 实例。	否
ice_newIdentity	返回一个新的、具有给定标识的无类型代理。	No
ice_getFacet	返回与代理相关联的 facet 的名字，如果没有设置 facet，就返回空串。	No
ice_newFacet	返回一个新的、具有给定的 facet 路径的无类型代理。路径必须是一个串数组。	No

表 19.3. 对核心代理方法的描述

方法	描述	远地的
ice_appendFacet	返回一个新的、已把给定的 facet 名附加到 facet 路径中的无类型代理。	否
ice_twoway	返回一个新的、用于进行双向调用的无类型代理。	否
ice_isTwoway	如果代理使用的是双向调用，返回 true，否则返回 false。	否
ice_oneway	返回一个新的、用于进行单向调用的无类型代理。	否
ice_isOneway	如果代理使用的是单向调用，返回 true，否则返回 false。	否
ice_batchOneway	返回一个新的、用于进行成批的单向调用的无类型代理。	否
ice_isBatchOneway	如果代理使用的是成批的单向调用，返回 true，否则返回 false。	否
ice_datagram	返回一个新的、用于进行数据报调用的无类型代理。	否
ice_isDatagram	如果代理使用的是数据报调用，返回 true，否则返回 false。	否
ice_batchDatagram	返回一个新的、用于进行成批的数据报调用的无类型代理。	否
ice_isBatchDatagram	如果代理使用的是成批的数据报调用，返回 true，否则返回 false。	否
ice_secure	返回一个新的无类型代理，其端点会根据一个布尔参数进行过滤。如果该参数是 true，只有使用安全传输机制的端点才能被使用，否则所有端点都能被使用。	否
ice_compress	返回一个新的无类型代理，其协议压缩能力由一个布尔参数决定。如果该参数是 true，在端点支持压缩的前提下，代理就会使用协议压缩。如果是 false，就不使用协议压缩。	否
ice_timeout	返回一个新的无类型代理，它具有给定的按毫秒计算的超时值。值为 -1 则禁用超时。	否
ice_default	返回一个新的、具有缺省的代理配置的无类型代理。	否

表 19.3. 对核心代理方法的描述

方法	描述	远地的
ice_uncheckedCast	返回一个新的、与给定的类型 id 关联在一起的有类型代理。如果提供了 facet 名，在返回的代理中这个 facet 已被附加到它的路径中。给定的类型 id 的 Slice 定义必须要能在当前 profile 中找到。	否
ice_checkedCast	如果远地对象支持给定的类型 id 所指示的类型，这个方法就返回一个新的、与该类型相关联 的有类型代理；否则就返回 null。如果提供了 facet 名，在返回的代理中这个 facet 已被附加到它的路径中。给定的类型 id 的 Slice 定义必须要能在当前 profile 中找到。	是

请求上下文

代理上的所有远地操作都支持一个可选的、位于最后的参数，用于表示请求上下文 (参见 16.8 节)。请求上下文会被映射到一个 PHP 关联数组，其中的键和值都是串。例如，下面的代码说明了怎样使用请求上下文来调用 ice_ping:

```
$p = $ICE->stringToProxy("a:tcp -p 12345");
$ctx = array("theKey" => "theValue");
$p->ice_ping($ctx);
```

标识

某些核心代理操作使用了 Ice_Identity 类型，这是 Slice 的 Ice::Identity 类型的 PHP 映射。这种类型的映射使用了用于 Slice 结构的标准规则，因此它是这样定义的:

```
class Ice_Identity {
    var $name;
    var $category;
}
```

IcePHP 提供了两个全局函数，用于把 Ice_Identity 值转换成串表示，或进行相反的转换:

```
function Ice_stringToIdentity(/* string */ $str);
function Ice_identityToString(/* Ice_Identity */ $id);
```

19.4.11 Ice::Communicator 的映射

因为 Ice 的 PHP 扩展只支持客户端的设施，Ice::Communicator 提供的许多操作都与此无关，因此，PHP 映射只支持通信器操作的一个子集。Ice::Communicator 的映射如下所示：

```
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();
}
```

对这些操作的描述，参见附录 B。

通信器的生命周期

PHP 脚本不允许创建或销毁通信器。通信器先于每个 PHP 请求创建，并在请求完成之后销毁。

访问通信器

脚本可以通过全局变量 $ICE 访问为某个请求创建的通信器实例。脚本不应该把另外的值赋给这个变量。和任何全局变量一样，需要在函数内部使用 $ICE 的脚本必须把这个变量声明成全局的：

```
function printProxy($prx) {
    global $ICE;

    print $ICE->proxyToString($prx);
}
```

通信器配置

脚本加载的 profile 将决定通信器的配置 。更多信息，参见 19.3 节。

第四部分

Ice 服务

第 20 章
IcePack

20.1 本章综述

在这一章我们将介绍 IcePack。 20.2 节简要介绍 IcePack 的各种特性，20.3 节和 20.4 节回顾了一些概念，有助于我们理解 IcePack 的工作方式。从 20.5 节到 20.7 节描述 IcePack 各个组件的用途和用法。 20.8 节和 20.9 节讨论怎样通过 IcePack 部署应用。 20.10 节就怎样排除故障给出了一些提示。

20.2 介绍

IcePack 提供了若干基本的服务，可以简化复杂的分布式应用的开发和部署:

- 定位器服务，用于定位对象和保证位置透明性，
- 服务器激活和监控服务，用于管理服务器进程，
- 强大的服务器部署机制。

IcePack 由两个主要组件组成:

- IcePack *注册表*，用于管理部署在特定域 (domain) 中的应用的所有相关信息。
- IcePack 节点，用于激活和监控服务器进程。

在一个域中只能有一个 IcePack 注册表，但可以有多个 IcePack 节点 (通常是一个主机一个)。

20.3 概念

在此前给出的大多数例子中，我们（为了简单起见）有意避开了分布式应用开发的一些重要问题，包括位置透明性、绑定，还有部署。 IcePack 解决了这些问题，而且对 Ice 应用的影响很小。

20.3.1 位置透明性

位置透明性意味着，对象实现的位置与客户无关；对象可以在客户的进程中、或另外的主机上的进程中实现。不管对象是在哪里实现的，客户与之交互的方式都完全是一样的，而且语义也相同。位置透明性简化了分布式应用的开发，是 IcePack 提供的服务的基础。

20.3.2 绑定

如 2.2 节所述，绑定是把代理和它的 servant 连接起来的过程。在实践中，这通常涉及到打开一个通向服务器的端点的 socket 连接。但是， Ice run time 可以采用两种绑定模式来获取服务器的端点信息：直接绑定和间接绑定。

直接绑定

在本书前面的例子中，我们已经多次看到过重复出现的端点信息：服务器调用 `createObjectAdapterWithEndpoints`，指定协议和端口号，而客户调用含有同样的协议和端口号的串化代理上的 `stringToProxy`。客户端的 Ice run time 打开一个直接通向服务器的端点的 socket ；如果服务器没有在运行，或是运行在另外的主机或端口上，那么客户就会收到异常。这叫作*直接绑定*，尽管这种绑定方式易于理解，也相当方便，对于许多企业应用而言并不适用。

通过使用配置属性来使端点信息外部化，我们可以稍微改善一下这种情况 (参见附录 C)。例如，我们可以使用下面的配置属性来指定名为 `MyAdapter` 的对象适配器的端点：

```
MyAdapter.Endpoints=tcp -p 9999
```

与此类似，客户可以定义一个含有串化代理的属性：

```
MyProxy=theObject:tcp -p 9999
```

这样的做法在正确的方向上前进了一步；至少，端点信息不再包含在代码中了。现在，不用改变客户或服务器的程序，就可以把服务器放到另外的主机或端口上去。

但这仍然不是一种理想的解决方案。例如，如果代理是存储在数据库中的，当服务器端点发生变化时，这些代理就必须全部被更新。与此类似，如果有多个客户和服务器应用运行在多个主机上，管理工作要涉及到更新配置文件、以反映不断演化的服务器配置，这很快就会变得难以实行。

间接绑定

顾名思义，*间接绑定*增加了一个间接层面，让客户不再需要为代理指定端点信息。代理的端点可以由 Ice run time 动态确定。我们将在 20.4 节讨论 Ice 是怎样使之成为可能的，但目前我们可以再用一些代理的例子来阐释其用法。

首先，我们用对象适配器标识符取代端点信息:

```
MyProxy=theObject@theAdapter
```

符号 `theAdapter` 标识的是一个对象适配器，其端点将会自动获得。

接下来，我们可以只使用对象标识:

```
MyProxy=theObject
```

在这种情况下，只用对象标识就足以定位对象了。

20.3.3 部署

应用部署是一个宽泛的用语，具有大量针对特定应用的要求。就 Ice 应用而言，应用部署涉及到配置文件的创建和安装、 IceBox 服务 (参见第 25 章) 的初始化，以及编写脚本来处理启动和关闭流程。 IcePack 能够简化上述任务：它使用了一种管理工具 ，处理你的应用的服务的 XML 描述，并自动部署这些服务。

20.4 Ice 定位器设施

Ice run time 支持由 `Ice::Locator` 接口所代表的委托设施，允许端点信息被动态 获取。这种设施带来了 20.3.2 节所描述的间接绑定的灵活性，是 IcePack 所提供的服务的基本组件。 `Ice::Locator` 接口定义的详细情况与这

里的讨论无关，但为了理解 IcePack 是怎样简化 Ice 应用的设计和开发的，它的能力值得我们探究一番。

20.4.1 对象与对象适配器

回想一下 20.3.2 节的内容，两个串化代理的例子演示了怎样进行间接绑定：

```
MyProxy=theObject@theAdapter
```

及

```
MyProxy=theObject
```

这两个代理说明了两种获取端点信息的方法：通过对象适配器标识符，以及通过对象标识。在两种情况下，客户端的 Ice run time 都会向一个 `Ice::Locator` 对象发出请求。这个活动对于应用而言是透明的，只要所创建的代理（调用 `stringToProxy`，或从某个操作那里接收到代理）使用的是间接绑定， Ice run time 就会发出这样的请求。但是， Ice run time 会注意使定位器请求的数量降到最低限度 (参见 20.4.7 节)。

一个定位器在本质上是一个仓库，其中含有两个查找表：一个把端点和对象标识关联在一起，另一个把端点和对象适配器标识符关联在一起。图 20.1 说明了定位器所扮演的角色：

1. 在启动时，服务器向定位器注册它的周知对象和对象适配器。
2. 在创建使用间接绑定的代理时，客户端的 Ice run time 调用定位器上的方法，获取与某个对象标识或对象适配器标识符相关联的端点。

3. 在为代理获取了端点之后，客户直接建立一个通向服务器的连接。客户不会再次询问定位器，除非发生连接错误。

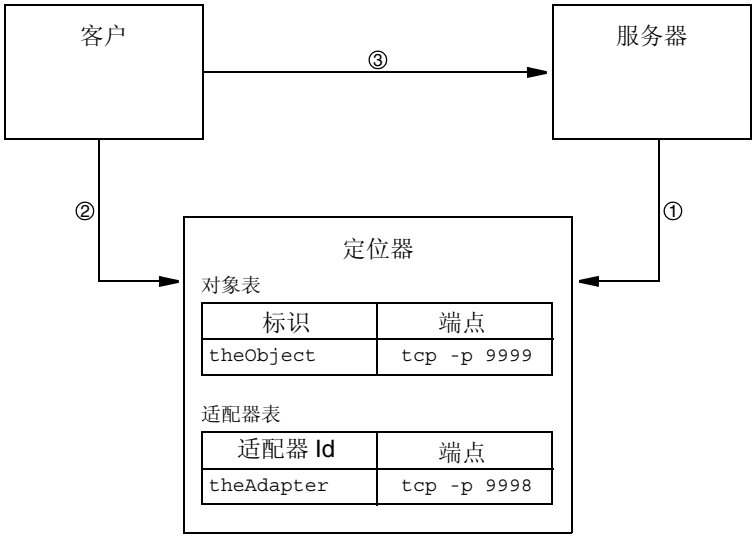

图 20.1. 从定位器那里获取端点信息。

20.4.2 设计考虑事项

尽管在 20.4.1 节给出的两种代理在语法上是类似的：它们消除了对端点信息的依赖，但它们的设计考虑事项有着明显的不同。特别地，对象标识方式的主要用途是进行引导 (bootstrapping)；也就是说，通过定位器设施可以访问一些周知的对象，客户用这些周知的对象来按照需要获取其他对象的代理。

另一方面，对象适配器方式可以用于许多对象，但仍然依赖于服务器配置 (也就是，对象适配器标识符)。

最好的做法常常是把两种方法结合起来：客户使用对象标识方式来引导一个首要对象，由这个对象提供指向 “使用对象适配器方式的次级对象” 的代理。

20.4.3 在客户中使用 Ice 定位器设施

要把定位器功能集成进客户应用，只需进行配置就行了：`Ice.Default.Locator` 属性的值必须指定定位器对象的串化代理。一旦 Ice run time 拥有了这个代理，它就可以把对象适配器标识符和对象标识转换成物理端点属性。关于怎样定义定位器属性的更多信息，参见 20.5.4 节。

20.4.4 在服务器中使用 Ice 定位器设施

适配器在被激活时，会自动向定位器注册它的端点。用一个标识符[1] 和一个定位器代理来配置对象适配器，就能启用自动注册。当然，对象适配器必须定义了一个或多个端点，但这个要求并非只针对定位器。

例如，可以用下面的属性来配置名为 `MyAdapter` 的对象适配器：

```
Ice.Default.Locator=... // the stringified locator proxy
MyAdapter.AdapterId=theAdapter
MyAdapter.Endpoints=default
```

这个配置把定位器标识符 `theAdapter` 指派给名为 `MyAdapter` 的对象适配器。这个对象适配器的端点将使用缺省协议和系统指派的端口号。在激活时，对象适配器向定位器（其代理在 `Ice.Default.Locator` 中提供）注册这个端点，从而使得客户能通过对象适配器代理方式使用这个适配器。

顾名思义，`Ice.Default.Locator` 属性提供的是服务器中的所有对象适配器的缺省定位器代理。但是，每一个其端点要向定位器注册的对象适配器，都应该用一个标识符进行配置。因此，服务器可以按照需要创建任意多个对象适配器，并且可以完全控制哪一些要向定位器注册。

使用下面这样的属性，还可以针对特定的对象适配器来配置定位器代理：

```
MyAdapter.Locator=... // the stringified locator proxy
```

这个属性的优先级高于 `Ice.Default.Locator` 属性。关于怎样定义定位器属性的更多信息，参见 20.5.4 节。

为了推动代理风格的对象标识的使用，每个对象的标识和代理都必须进行注册。这可以由服务器在程序中完成，可以用 20.7 节所描述的 IcePack

1. 如果一个域中的各个对象适配器没有使用唯一的标识符，就有可能发生冲突；开发者要负责确保不会发生这样的事情。

命令行工具来完成，也可以用 20.8 节所讨论的 IcePack 部署机制来自动完成。

20.4.5 部署

定位器设施可以简化应用的部署，因为我们无需再考虑怎样给对象适配器指派唯一的端点。因为代理只含有指向端点信息的符号引用，对象适配器可以使用系统指派的端口 (而不是使用用户指派的端口)。

尽管对象适配器在每次激活时获得的端口可能会不同，但因为它会自动向定位器进行注册，所以能确保客户收到正确的端点信息。如果客户持有一个代理，指向的是某个对象适配器中的一个对象，这个适配器在解除了激活之后又重新在另外的端口上激活了，客户的 Ice run time 的重试行为将会自动更新代理的端点。

20.4.6 依赖

定位器设施由 `Ice::Locator` 接口代表，这是一个普通的 Slice 接口，可以有多种实现。如果 Ice run time 配置了一个 `Ice::Locator` 代理，它会自动使用这个定位器来进行间接绑定。因此， Ice run time 对 `Ice::Locator` 接口有依赖，但不依赖 IcePack； IcePack 注册表只是实现了 `Ice::Locator` 接口，从而允许客户使用间接绑定来访问 IcePack 节点所激活的服务器。

20.4.7 间接绑定的开销

如你可能会期望的一样，间接绑定会带来一些额外的开销，因为 Ice run time 必须通过代理对定位器进行调用，以获取端点信息。为了使定位器尽量少调用定位器， Ice run time 会缓存调用的结果；如果随后要绑定的是相同的标识或对象适配器名，所使用的就将是缓存的端点信息。对于大多数应用而言，间接绑定所提供的灵活性远远超过了使用它所造成的一点开销。

20.5 IcePack 注册表

IcePack 注册表是一个集中式信息仓库，其中含有:

- IcePack 节点，
- 已部署的服务器和对象适配器，
- 对象注册信息。

注册表还实现了 20.4 节所描述的 `Ice::Locator` 接口，并且支持另外一些 Slice 接口，可执行查询和管理任务。注册表服务可以与某个 IcePack 节点并置在一起，从而节省资源，而且在开发和测试过程中也很方便。

20.5.1 端点

IcePack 注册表最多创建四组端点，用下面的属性进行配置:

- `IcePack.Registry.Client.Endpoints`

  支持 `Ice::Locator` 和 `IcePack::Query` 的客户端端点。
- `IcePack.Registry.Server.Endpoints`

  用于对象和对象适配器注册的服务器端端点。
- `IcePack.Registry.Admin.Endpoints`

  支持 `IcePack::Admin` 接口的管理端点 (可选)。
- `IcePack.Registry.Internal.Endpoints`

  IcePack 节点用于与注册表进行通信的内部端点 (可选)。只有当正在使用的注册表中没有任何节点时，这个属性才是可选的。

更多关于这些属性的信息，参见附录 C。

20.5.2 安全考虑事项

如果注册表运行在不安全的主机上 （比如应用防火墙)，对 IcePack 注册的管理接口的访问就应该受到限制。

`IcePack.Registry.Admin.Endpoints` 属性 (在前一节中描述过) 提供了多种选择:

- 不定义该属性，完全禁止进行管理访问。
- 阻塞对该属性定义的管理端点的外部访问。
- 只使用该属性中的 SSL 端点 (参见第 23 章)。

应该选择哪种方法，取决于你的部署需求。例如，如果不需要进行管理访问，那么第一种技术就更安全。如果这种做法不可行，那么我们建议你 (通过防火墙) 从物理上阻塞对管理端点的外部访问。最后，如果注册表必须从外部进行管理，那么就应该使用 SSL 端点来确保只有验证过的客户能够获得访问权。

20.5.3 注册表对象标识

下面是主要的 IcePack 注册表对象的标识:

- `IcePack/Locator`，`Ice::Locator` 实现的标识。
- `IcePack/Query`，`IcePack::Query` 实现的标识。
- `IcePack/Admin`，`IcePack::Admin` 实现的标识

通过这些标识，以及 20.5.1 节所描述的各个属性定义的端点，你可以构造出注册表对象的代理。

20.5.4 Ice 定位器属性

现在，我们知道了注册表的 `Ice::Locator` 实现的标识，以及在 `IcePack.Registry.Client.Endpoints` 属性中定义的端点，我们可以为 Ice 客户和服务器定义配置属性，启用定位器功能了。

一般而言，客户和服务器都要使用 `Ice.Default.Locator` 属性。例如，如果 IcePack 注册表客户端点是 `tcp -p 12000`，那么就可以这样定义定位器属性:

```
Ice.Default.Locator=IcePack/Locator:tcp -p 12000
```

在服务器中，使用下面的属性定义，可以为每个对象适配器个别地指定一个定位器代理:

```
MyAdapter.Locator=IcePack/Locator:tcp -p 12000
```

如果某个适配器的定位器属性没有定义，`Ice.Default.Locator` 的值就会被使用。关于这些属性的更多信息，参见附录 C。

20.5.5 用法

IcePack 注册表支持以下命令行选项:

```
$ icepackregistry -h
Usage: icepackregistry [options]
Options:
-h, --help            Show this message.
-v, --version         Display the Ice version.
--nowarn              Don't print any security warnings.
```

注册表可以作为 Unix 看守 (daemon) 或 Win32 服务运行。在 Unix 上还支持以下额外的命令:

```
--daemon              Run as a daemon.
--noclose             Do not close open file descriptors.
--nochdir             Do not change the current working directory.
```

在 Windows 上还支持以下额外的参数:

```
--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.
```

关于这些选项的更多信息，参见 10.3.2 节。

20.5.6 配置

IcePack 注册表的配置属性具有前缀 `IcePack.Registry`，附录 C 对这些属性进行了描述。

20.5.7 定位服务

如果应用需要间接绑定，但不需要 IcePack 节点的服务器激活特性，IcePack 注册表自身可以用作定位服务。

例如，要想让注册表在这种模式下工作，可以使用下面的配置文件定义的一组最小限度的属性集:

```
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
```

因为没有 IcePack 节点要和这个注册表一起运行，我们不必定义 `IcePack.Registry.Internal.Endpoints` 属性。要想让服务器动态

地注册它们的对象适配器，而不必先通过部署来配置它们，需要使用 IcePack.Registry.DynamicRegistration 属性。

我们可以用下面的命令来初始化注册表的数据库目录并启动它：

```
$ mkdir db
$ icepackregistry --Ice.Config=config
```

后一条命令假定配置文件存在于当前目录中，名为 config。

现在，客户和服务器可以用注册表的配置文件中的端点来定义缺省的定位器属性，从而使用这个注册表了：

```
Ice.Default.Locator=IcePack/Locator:default -p 12000
```

用这个属性配置的服务器在进行手工激活时，会注册自己的每一个有 AdapterId 属性的对象适配器，从而允许客户使用对象适配器代理方式。此外，你也可以使用管理工具 (参见 20.7 节) 来向注册表注册周知的对象，并允许客户使用对象代理方式。

20.6 IcePack 节点

一个 IcePack 节点是一个负责 “已注册的服务器进程的激活、监控、以及激活解除” 的进程。服务器向 IcePack 节点注册是通过 20.8 节描述的 IcePack 部署机制来完成的。

在一个域中，你可以运行任意数目的 IcePack，但通常一个主机只有一个节点。如果一个主机上的服务器要想被自动激活，在这个主机上就必须运行 IcePack 节点；如果没有 IcePack 注册表，节点就无法运行。

20.6.1 Server 激活

随需激活服务器是分布式计算架构的一个很有价值的特性，原因有很多：

- 因为无需预先启动所有服务器，应用启动时间得以降到了最低限度；
- 管理员能够更高效地使用他们的计算资源，因为只有那些确实需要的服务器才在运行；
- 在有服务器出故障的情况下 （例如，服务器在出故障后重新激活，可能仍然有能力给客户提供某些服务，直到故障被修复为止），这种激活方式提供了更高的可靠性；
- 可以进行远地激活和激活解除。

激活详解

激活发生在 Ice 客户通过 Ice 定位器设施请求获取服务器的某个对象适配器的端点时 (参见 20.4 节)。如果在客户发出请求时，服务器不是活动的，节点就会激活该服务器，并等待服务器的对象适配器注册其端点。一旦对象适配器端点注册好，节点就会把端点信息发回给客户。这样的动作序列确保了客户会在服务器准备后接收请求之后收到端点信息。

要求

要让某个对象适配器进行随需激活，该适配器必须有一个标识符，并放进 IcePack 注册表中。

高效率

上面列出的随需激活的一个优点是能够更高效地管理计算资源。当然，这个问题涉及到许多方面，但 Ice 使得一种技术变得特别简单：可以对服务器进行配置，让它们在空闲了一定时间之后得体地关闭。

一种典型的情况：服务器随需激活，由一个或多个客户使用一段时间，然后，如果在一定时间（可配置）之内没有请求，服务器就自动终止。你所需做的只是把 `Ice.ServerIdleTime` 属性设置成你想要的空闲时间。

20.6.2 端点注册

20.4.4 节讨论了要启用服务器的自动端点注册，所需进行的配置。但你要注意， IcePack 以两种方式简化了配置过程：

1. 由 IcePack 节点自动激活的服务器无需显式地配置定位器的代理，因为 IcePack 节点会在服务器的命令行上定义它。
2. IcePack 部署机制使服务器的配置文件的创建得以自动化，包括对象适配器标识符和端点 (参见 20.8.5 节)。

20.6.3 服务器注册与解除激活

在激活某个服务器之后， IcePack 节点通常不会解除该服务器的激活，除非收到通过管理命令发出的显式请求。 IcePack 节点首先会尝试得体地解除该服务器的激活，然后等待服务器有序地关闭。如果在一段时间之后，服务器没有自己终止， IcePack 节点就会终止进程。

IcePack 节点可以通过两种方式来要求服务器有序地关闭：

1. 调用驻留在服务器中的 `Ice::Process` 对象的代理。
2. 使用平台特有的机制，比如 POSIX 信号。

后一种方式在 POSIX 平台上工作良好，但要求服务器为拦截信号做好准备 (参见 15.11 节)。在 Windows 平台上，它在用于 C++ 服务器时不那么可靠，而且完全不能用于 Java 服务器。由于这些原因，前一种选择更可取，因为它是可移植的和可靠的；如果没有提供 `Ice::Process` 代理，IcePack 节点就只使用后一种选择。

服务器要想注册 `Ice::Process` 代理，最简单和最常用的方式是把对象适配器属性 `RegisterProcess` 设成非零值。设置了这个属性之后，新激活的对象适配器会创建一个实现 `Ice::Process` 接口的 servant，使用一个 UUID 做标识、把它添加到活动 servant 映射表中，然后调用 Ice 定位器设施上的操作注册该代理。

对象适配器还需要 `Ice.ServerId` 属性的值，你应该把它设成服务器的名字。 IcePack 节点会为自动激活的服务器定义这个属性。

服务器应该最多为它的一个对象适配器配置 `RegisterProcess` 属性。如果使用了 IcePack 部署描述符，只要把 `adapter` 元素的 `register` 属性设成真， `RegisterProcess` 属性就会在服务器的配置中自动定义。(参见 20.9 节)。

`Ice::Process` servant对IcePack节点的解除激活请求的响应是调用通信器的 `shutdown`。这带来了一个潜在的安全漏洞：如果恶意客户获得了该代理，它可以通过关闭服务器来发起 “拒绝服务” 攻击。因此，仔细考虑服务器的端点，使遭到攻击的风险降至最低，这十分重要。特别地，你可以用安全端点来配置对象适配器，也可以把一个单独的对象适配器专用于 `Ice::Process` servant，并用只有 IcePack 节点能访问的端点来配置它。

20.6.4 用法

IcePack 节点支持以下命令行选项：

```
$ 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.
```

通过 `--deploy` 选项，应用可以在节点进程启动时被自动部署，在测试过程中这特别有用。在其后应给出描述符文件名，然后再给出描述符文件中的各个目标 (target) 的名字 （可选）。

节点可以作为 Unix 看守或 Win32 服务运行。在 Unix 上还支持以下额外的命令:

```
--daemon              Run as a daemon.
--noclose             Do not close open file descriptors.
--nochdir             Do not change the current working directory.
```

在 Windows 上还支持以下额外的命令:

```
--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.
```

更多关于这些选项的信息，参见 10.3.2 节。

20.6.5 配置

IcePack 节点的配置属性具有前缀 `IcePack.Node`，附录 C 对这些属性进行了描述。

20.7 IcePack 管理工具

IcePack 管理工具是一个命令行程序，可用于管理 IcePack 操作的所有方面，包括应用和服务器的部署，以及查询已注册实体的相关信息。

这个工具要求你按照 20.5.4 节所描述的方式设置 `Ice.Default.Locator` 属性。

20.7.1 用法

IcePack 管理工具支持以下命令行选项:

```
$ 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.
```

如果使用了 -e 选项，管理工具会执行给定的命令，然后不进入交互式会话就退出。否则管理工具就会进入一个交互式会话； **help** 命令显示以下使用信息:

help

打印这个消息。

exit, **quit**

退出这个程序。

application add DESC [TARGET1 [TARGET2 ...]]

添加在应用描述符 DESC 中描述的服务器。如果指定了可选的目标，这些目标也将被部署。

application remove DESC

移除在描述符 DESC 中描述的服务器。

node list

列出所有已注册的节点。

node ping NAME

Ping 节点 NAME。

node shutdown NAME

关闭节点 NAME。

server list

列出所有已注册的服务器。

server add NODE NAME DESC [PATH [LIBPATH [TARGET1 [TARGET2 ...]]]]

使用部署描述符 DESC、可选的 PATH 及 LIBPATH，把服务器 NAME 添加到节点 NODE 中。如果指定了可选的目标，这些目标也将被部署。

server remove NAME

移除服务器 NAME。

server describe NAME

获取服务器 NAME 的描述。

server state NAME

获取服务器 NAME 的状态。

server pid NAME

获取服务器 NAME 的。

server start NAME

启动服务器 NAME。

server stop NAME

停止服务器 NAME。

server signal NAME SIGNAL

发送 SIGNAL (例如， SIGTERM 或 15) 给服务器 NAME。

server stdout NAME MESSAGE

把 MESSAGE 写到服务器 NAME 的 stdout。

server stderr NAME MESSAGE

把 MESSAGE 写到服务器 NAME 的 stderr。

server activation NAME [on-demand | manual]

把服务器激活模式设成随需激活或手工激活。

adapter list

列出所有已注册的适配器。

adapter endpoints NAME

获取适配器 NAME 的端点。

object add PROXY [TYPE]

把一个对象添加到对象注册表中，可以指定其类型。

object remove IDENTITY

从对象注册表中移除一个对象。

object find TYPE

找到所有类型为 TYPE 的对象。

shutdown

关闭 IcePack 注册表。

20.8 部署

就 IcePack 而言，部署意味着为某个应用配置 IcePack 域。特别地，部署涉及到把周知的对象和对象适配器添加到 IcePack 注册表中、配置 IcePack 节点上的服务器，以及创建服务器配置文件。

IcePack 提供了一种数据驱动的部署服务，其中的*描述符*用 XML 编写，并按照所需的频度进行部署。有三种类型的部署描述符:

- 应用描述符定义要部署的服务器，以及在哪些节点上部署它们。
- 服务器描述符定义用 C++ 或 Java 编写的 Ice 服务器应用。服务器描述符也可以定义 IceBox 服务器 (参见第 25 章)。
- 服务描述符定义 IceBox 服务。

在部署过程中， IcePack 按照描述符的指示配置它的各种组件。每个描述符都必须放在它自己的文件中。

20.8.1 应用描述符

就 IcePack 而言，*应用*被定义为运行在一个或多个 IcePack 节点上的一组服务器和 / 或 IceBox 服务。应用描述符的结构与下面给出的例子类似:

```
<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>
```

在 application 元素内部有两个 node 元素，定义了 IcePack 节点 Firewall 和 Backend。服务器 AppGateway 将被部署在 Firewall 上，而服务器 Account 将被部署在 Backend 上。可执行文件的路径，以及服务器描述符文件，在每个 server 元素的属性中给出。

20.8.2 服务器描述符

服务器描述符定义服务器的对象适配器、周知对象，以及配置属性。

继续前面的例子，下面是 appgateway.xml 的内容:

```
<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>
```

这个服务器描述符定义了一个 C++ 服务器，具有一个名为 GatewayAdapter 的对象适配器，有一个缺省端点。这个适配器定义了一个周知的对象，类型是 ::AppGateway；以及一个标识，其值是变量 ${name} 的值。在这个例子中，${name} 会被替换成 AppGateway，这是应用描述符中的服务器的名字。

这个服务器描述符还定义了一个名为 Account.Proxy 的属性，其值是 AccountManager，表示的是一个周知对象的代理，这个代理将通过 Ice 定位器设施被解析。

下面是 account.xml 中的描述符：

```
<server kind="cpp-icebox" endpoints="default">
    <service name="AccountService" descriptor="accountsvc.xml"/>
</server>
```

这个服务器描述符定义了一个 C++ IceBox 服务器，具有一个缺省的端点。它配置了一个名为 AccountService 的 IceBox 服务，其描述符位于文件 accountsvc.xml 中。

更多关于 IceBox 的信息，参见第 25 章。

20.8.3 服务描述符

服务描述符定义 IceBox 服务，包括它的对象适配器、周知对象，以及配置属性。

继续前面的例子，下面是 accountsvc.xml 的内容:

```
<service kind="standard" entry="AccountService:create">
    <adapters>
        <adapter name="AccountAdapter" endpoints="default">
            <object identity="AccountManager"
                    type="::AccountManager"/>
        </adapter>
    </adapters>
</service>
```

这个服务描述符定义了一个 IceBox 服务，具有一个名为 AccountAdapter 的对象适配器，有一个缺省端点。这个适配器定义了一个类型为 ::AccountManager 的周知对象，以及标识 AccountManager。这是 20.8.2 节中的 AppGateway 服务器描述符所引用的周知对象。

20.8.4 要求

要部署应用， IcePack 注册表必须在运行，服务器将要部署到的所有 IcePack 节点也必须在运行。

20.8.5 配置服务器和服务

IcePack 为每个被部署的服务器和服务生成一个配置文件。该文件含有以下属性:

- 为每个 adapter 元素生成 AdapterId 和 Endpoints 属性
- 如果有必要，为服务器适配器生成 RegisterProcess 属性

- 每个 property 元素的等价定义
- 为 Freeze 服务 *Name* 生成 IceBox.DBEnvName.*Name* 属性
- 为 IceBox 服务器生成 IceBox.ServiceManager.Identity 属性
- 为 IceBox 服务器生成 IceBox.LoadOrder 属性，按 IceBox 服务在 <server> 描述符中出现的次序列出它们

这些文件位于 IcePack 节点的数据目录中，通常不应该手工编辑，因为如果应用重新部署，任何变动都会丢失。

20.8.6 部署例子

为了举例阐释上述内容，让我们来部署一个非常简单的应用。应用描述符包含在文件 application.xml 中:

```
<application>
    <node name="node" basedir="${basedir}">
        <server name="SimpleServer" binpath="./server"
                descriptor="server.xml"/>
    </node>
</application>
```

下面是文件 server.xml 中的服务器描述符:

```
<server kind="cpp">
    <adapters>
        <adapter name="SimpleAdapter" endpoints="default">
            <object identity="${name}" type="::Simple"/>
        </adapter>
    </adapters>
</server>
```

我们可以通过一些简单的步骤部署这个应用。

1. 为这个 IcePack 节点创建一个叫作 config 的配置文件。该文件应该含有以下属性:

```
#
# 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. 为这个 IcePack 节点创建数据库目录结构:

```
$ mkdir db
$ mkdir db/registry
$ mkdir db/node
```

3. 启动这个 IcePack 节点。注意，该节点将有一个并置的 IcePack 注册表服务，因为在配置文件中定义了 `IcePack.Node.CollocateRegistry` 属性。

```
$ icepacknode --Ice.Config=config
```

4. 在另一个窗口中，启动管理工具，部署该应用。我们假定，应用描述符在文件 `application.xml` 中。在启动 **icepackadmin** 时，我们把同一个配置文件用作 **icepacknode**。这样做只是为了方便；管理工具只需要 `Ice.Default.Locator` 属性。

```
$ icepackadmin --Ice.Config=config
>>> application add "application.xml"
```

现在，这个应用应该已经部署好了。注意， `application.xml` 必须放在引号中，因为在 **icepackadmin** 命令语言中， `application` 是一个关键字。

你可以使用下面的命令来验证配置:

```
>>> server list
SimpleServer
>>> adapter list
IcePack.Node-node
IcePack.Registry.Internal
SimpleAdapter-SimpleServer
>>> object find ::Simple
SimpleServer -t @ SimpleAdapter-SimpleServer
```

注意，我们没有显式地给我们的服务器描述符中的对象适配器指定标识符，所以部署器把适配器名和服务器名结合在一起，创建了标识符 SimpleAdapter-SimpleServer.

20.8.7 目标部署

部署描述符允许你定义叫作*目标*(target) 的元素，这样的元素只有在你显式要求时才会被部署。这在许多情况下都是有用的，但最常见的是用于调试。我们将用两个例子来说明使用目标的好处。

例 1

让我们修改 20.8.6 节的简单部署例子的服务器描述符，在其中包含一个 target 元素:

```
<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>
```

我们增加了一个名为 debug 的目标，其中含有定义了一个属性的 properties 元素。如果在部署这个描述符时没有指定目标， target 元素的内容就会被忽略。如果要部署 debug 目标，所有其他的元素都会像平常一样被处理；此外，有一个属性会被添加到服务器的配置文件中 (参见 20.8.5 节)。如 20.9.6 节所述，这个属性的名字是用描述符变量来生成的。在这个例子中，这个属性的名字是 SimpleServer.Debug。

使用 **icepackadmin**，我们通常可以这样部署这个应用:

```
>>> application add "application.xml"
```

另外，如果我们需要在定义了调试属性的情况下运行应用，我们可以使用调试目标来部署它:

```
>>> application add "application.xml" debug
```

注意，我们可以把目标名指定为 debug，尽管应用描述符没有使用这个名字的目标 (我们把这个目标添加到了服务器描述符中)。像 debug 这样的未受限定 名字将被应用到应用中的每一个描述符。如果我们有多个服务器，又想确保只在一个服务器中启用调试，我们需要使用受到完全限定的目标名：

```
>>> application add "application.xml"
    "node.SimpleServer.debug"
```

例 2

现在，让我们考虑一个稍微复杂一点的例子。我们扩展了服务器描述符中的 target 元素：

```
<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>
```

除了在前面的例子中引入的属性以外，我们还定义了一个新的对象适配器，名为 DebugAdapter，并且注册了一个标识为 DebugController 的新对象。当服务器看到设置了调试属性之后，会创建这些实体，为客户提供访问调试设施的能力。

20.9 描述符参考资料

这一节描述 IcePack 部署描述符中的各个 XML 元素。

20.9.1 应用元素

应用描述符由单个 application 元素组成，其中含有一个或多个 node 元素。每个 node 元素都含有一个或多个 server 元素，描述要在该节点上部署的服务器。

Application

application 元素支持以下属性:

表 20.1. application 元素的属性

属性	描述	必须指定
basedir	在解析这个元素中的路径名时使用的基目录。如果没有指定，就使用描述符文件所在的目录。	否

Node

node 元素支持以下属性:

表 20.2. node 元素的属性

属性	描述	必须指定
name	和 IcePack.Node.Name 配置属性的值相同的节点名。	是
basedir	在解析这个元素中的路径名时使用的基目录。如果没有指定，就使用这个 IcePack 节点的当前工作目录。该目录必须能让这个 IcePack 节点 访问。	否

Server

server 元素支持以下属性:

表 20.3. server 元素的属性

属性	描述	必须指定
name	服务器名。服务器名在 IcePack 注册表中唯一地标识服务器。	是
descriptor	描述符的路径名。	是
binpath	服务器可执行程序的路径名。如果要部署的是 C++ Ice 服务器，必须定义这个属性。对于 C++ IceBox 服务器，如果没有指定这个属性，就会使用 icebox。对于 Java Ice 服务器和 IceBox 服务器，如果没有指定这个属性，就会使用 java。	只有 C++ Ice 服务器才必须定义
libpath	对于 Java Ice 服务器和 IceBox 服务器，这个属性指定的是 Java 类路径。这个属性的值会直接传给 Java 虚拟机。这个属性目前没有用于 C++ 服务器。	否
targets	用于部署服务器的目标名，用空格分隔。	否

20.9.2 Server 元素

服务器描述符由单个 server 元素组成。该元素可以含有以下元素:

- service 元素，定义 IceBox 元素
- 一个 adapters 元素，定义要在 IcePack 注册表中注册的对象适配器
- 一个 properties 元素，指定服务器的配置属性
- 一个 classname 元素，定义含有 Java 服务器的 main 方法的类的名字
- 一个 pwd 元素，指定服务器的工作目录的路径名
- option 元素，提供服务器命令行选项
- vm-option 元素，提供传给 Java 虚拟机的命令行选项
- env 元素，提供环境变量定义
- target 元素，定义部署目标。

Server

server 元素支持以下属性:

表 20.4. server 元素的属性

属性	描述	必须指定
basedir	在解析这个元素中的路径名时使用的基目录。如果没有指定，就使用描述符文件所在的目录。	否
kind	这个描述符所描述的服务器的类型。可能的值是: cpp、 java、 cpp-icebox 和 java-icebox。	是
endpoints	对于 C++ 或 Java IceBox 服务器，这个属性定义的是 IceBox 服务管理器接口的端点，等价于 IceBox.ServiceManager.Endpoints 配置属性。	只有 IceBox 服务器才必须定义

Service

只有当服务器的 kind 属性是 java-icebox 或 cpp-icebox 时，server 元素范围内的 service 元素才有效。 service 元素支持以下属性:

表 20.5. 服务元素的属性

属性	描述	必须指定
name	服务名。服务名在一个 IceBox 服务器中唯一地标识服务。	是
descriptor	描述符的路径名。	是
targets	用于部署服务的目标名，用空格分隔。	否

Classname

只有当服务器的 kind 属性是 java 时， server 元素范围内的 classname 元素才有效。这个元素的值指定的是含有服务器的 main 方法的类的名字。这个元素没有属性。

Env

`env` 元素定义一个环境变量，将在服务器激活时出现在服务器的环境中。这个元素的值的形式应该是 *name*=*value*。这个元素没有属性。

Pwd

`pwd` 元素值指定的是服务器的工作目录的路径名。这个元素没有属性。

Option

`option` 元素值定义一个命令行选项，将在服务器激活时传给服务器。这个元素没有属性。

Vm-option

只有当服务器的 `kind` 属性是 `java` 或 `java-icebox` 时， `server` 元素范围内的 `vm-option` 元素才有效。这个元素的值定义的是用于 Java 虚拟机的命令行选项，因此，它会出现在 JVM 命令行上、在服务器类名的前面。这个元素没有属性。

20.9.3 Service 元素

服务描述符定义的是 IceBox 服务，由单个 `service` 元素组成。这个元素可以包含一个 `adapters` 元素，定义要注册到 IcePack 注册表的对象适配器；还包含一个 `properties` 元素，指定服务的配置属性，或是包含一个 `target` 元素，定义部署的目标。

Service

service 元素支持以下属性:

表 20.6. service 元素的属性

属性	描述	必须指定
basedir	在解析这个元素中的路径名时使用的基目录。如果没有指定，就使用描述符文件所在的目录。	否
kind	这个描述符所定义的 IceBox 服务的类型。可能的值是 standard 或 freeze。	是
entry	服务进入点。在 C++ 中，进入点的形式是 *library*:*symbol*，*library* 是共享库或 DLL 的名字，*symbol* 是用于创建服务的工厂函数的名字。在 Java 中，进入点是服务实现类的名字。	是
dbenv	如果 kind 属性的值是 freeze，这个属性定义的是服务数据库环境目录。如果没有指定，节点会提供一个缺省的数据库环境目录。	否

20.9.4 共有的元素

部署描述符支持一些共有的元素。

Target

只有当用户在部署过程中指定了目标名，放在 target 元素中的才会被部署。 target 元素可用在应用、服务器、以及服务描述符中。此外，target 元素只能包含那些可以合法地出现在它所属的描述符中的元素。例如，在应用描述符的 target 元素中， node 元素是合法的，但在服务器描述符的 target 元素中，就是不合法的。这个元素支持以下属性:

表 20.7. target 元素的属性

属性	描述	必须指定
name	目标名。	是

Adapters

adapters 元素含有一个或多个 adapter 元素，可用在服务器描述符和服务描述符中。这个元素没有属性。

Adapter

adapter 元素指示部署器，向 IcePack 注册表注册这个适配器。注册表会把适配器与它的服务器关联起来，从而启用随需激活。 adapter 元素只能用在 adapters 元素中。这个元素支持以下属性:

表 20.8. adapter 元素的属性。

属性	描述	必须指定
name	对象适配器名。这也是应用传给 `createObjectAdapter` 操作的名字。	是
id	对象适配器标识符。 IcePack 注册表用这个标识符唯一地标识一个对象适配器。如果没有指定，注册表就会基于适配器名和服务器名 (如果适配器是在服务描述符中定义的，则使用服务名) 创建标识符。	否
endpoints	对象适配器的端点。	是
register	如果为真，就会为这个适配器定义 `RegisterProcess` 属性。一个服务器只应该注册一个适配器。更多信息，参见 20.6.3 节。如果没有定义，缺省值是假。	否

Object

object 元素指示部署器，向 IcePack 注册表注册对象，以使客户能通过标识或类型查找对象。 object 元素只能用在 adapter 元素中。object 元素支持以下属性:

表 20.9. object 元素的属性

属性	描述	必须指定
identity	一个串化对象标识。	是
type	范围受到完全限定的对象类型，比如 `::Hello`。	否

Properties

properties 元素含有一个或多个 property 元素，可用在服务器和服务描述符中。这个元素没有属性。

Property

每个被部署的服务器或服务都有一个配置文件，位于 IcePack 节点的数据目录中，而 property 元素指示部署器，给这个配置文件添加一个属性。 property 元素只能用在 properties 元素中。 property 元素支持以下属性:

表 20.10. property 元素的属性

属性	描述	必须指定
name	property 的名字。	是
value	property 的值。	如果没有指定 location
location	被解释成路径名的 property 值。相对路径名是相对于描述符基目录而言的。	如果没有指定 value

value 和 location 属性是互相排斥的。

20.9.5 路径名语义

部署描述符中的相对路径名会使用它所属的元素的基目录来解析。缺省的基目录是描述符文件所在的目录，除非被 basedir 属性取代 (如 20.8.1 节的应用描述符所示)。

重要的是要理解，在部署过程中是怎样使用这些路径名的:

1. **icepackadmin** 工具 (在 20.7 节描述过) 把应用描述符路径名发给 IcePack 注册表，由后者读取该文件。因此，这个文件必须要能被注册表访问；相对路径名会相对于注册表进程的当前工作目录进行解析。
2. IcePack 注册表会处理每个 node 元素：联系指定的 IcePack 节点，部署其服务器。
3. IcePack 节点读取服务器描述符文件，以及任何 IceBox 服务描述符文件。因此，服务器和服务描述符的路径名必须能被它们要部署到的

IcePack 节点访问；相对路径名会相对于节点进程的当前工作目录进行解析。此外，二进制文件和库的路径名也必须能被 IcePack 节点访问。

20.9.6 描述符变量

在部署描述符中可以使用一些变量。这些变量的形式是 ${*name*}，*name* 是变量名。如下所示，你可以在属性值中使用变量：

```
<object id="Factory-${name}-Object" type=""/>
```

在部署过程中，${name} 会被替换成服务器或服务的描述符名。

在 IcePack 描述符中你可以使用以下变量：

- basedir

  描述符的基目录。如果没有被顶层元素 basedir 的属性覆盖，基目录就是描述符所在的目录。更多关于描述符路径名的信息，参见 20.9.5 节。

- name

  组件名。在服务器描述符中，这个变量被设成应用描述符所定义的服务器名。对于 IceBox 服务，这个变量被设成 IceBox 服务器描述所定义的服务名。这个变量不能用在应用描述符中。

- parent

  父组件的名字。在服务器描述符中，这个变量被替换成服务器所部署到的 IcePack 节点的名字。在服务描述符中，这个变量被替换成服务所部署到的 IceBox 服务器的名字。这个变量不能用在应用描述符中。

- fqn

  组件的受到完全限定的名字。在服务器描述符中，这个变量的形式是 *node_name.server_name*，*node_name* 是服务器所部署到的节点的名字，而 *server_name* 是服务器名。在服务描述符中，这个变量的形式是 *node_name.server_name.service_name*。这个变量不能用在应用描述符中。

20.10 排除故障

这一节就怎样解析若干常见的 IcePack 问题给出了一些建议。

20.10.1 激活故障

自动的服务器激活可能会因为许多原因而失败，但可能性最大的原因是配置不正确。例如，某个 IcePack 节点之所以没有能够成功激活某个服务器，是因为服务器的可执行文件或相关的共享库找不到。在这种情况下，你可以采取若干步骤:

1. 设置配置属性 `IcePack.Node.Trace.Activation=3`，启用 IcePack 节点中的激活跟踪。
2. 检查跟踪输出，检验服务器的可执行文件的路径名及工作目录是否正确。
3. 如果可执行文件的路径名是正确的，而且这是一个相对路径名，那么它相对于 IcePack 节点的当前工作目录而言有可能是不正确的 (参见 20.9.5 节)。你可以用绝对路径名替换相对路径名，也可以在正确的工作目录中重启 IcePack 节点。
4. 检验在启动 IcePack 节点时是否为服务器的共享库设置了正确的 `PATH` 或 `LD_LIBRARY_PATH`。

激活失败的另一种原因是服务器在启动过程中的故障。在你使用上述步骤、确认 IcePack 节点成功派生了服务器进程之后，你应该再检查服务器故障的各种迹象 (例如，在 UNIX 上，在 IcePack 节点的当前工作目录中查找 `core` 文件)。关于服务器故障的更多信息，参见 20.10.4 节。

20.10.2 代理故障

如果间接代理的绑定失败，客户可能会收到 `Ice::NotRegisteredException` (参见 20.3.2 节)。这个异常表明，代理的对象标识或对象适配器没有向 Ice 定位器设施注册。下面的步骤可以帮助你找出发生异常的原因:

1. 使用 **`icepackadmin`** (参见 20.7 节) 检验对象标识或对象适配器标识符确实已注册，而且与代理所使用的是匹配的:

```
>>> adapter list
...
>>> object find ::Hello
...
```

2. 如果问题仍继续出现，复查你的配置，确保客户所用的定位器代理与 IcePack 注册表的客户端点是匹配的，而且这些端点可以被客户访问 (也就是说，没有被防火墙阻塞)。

3. 最后，设置配置属性 `Ice.Trace.Locator=1`，启用客户中的定位器跟踪，然后再次运行客户，看是否有日志消息能指出问题。

20.10.3 部署故障

描述符中的路径名，特别是相对路径名，是部署问题的常见来源。在编写描述符文件时，你应该熟悉 20.9.5 节所阐释的部署过程。简而言之，服务器和服务描述符文件会被它们所部署到的 IcePack 节点读取，因此，你指定的描述符路径名必须要能被 IcePack 节点访问。在实践中这意味着，节点的服务器和服务的描述符文件或者要复制到节点所运行的主机上，或者必须能通过网络文件系统访问。此外，要使用相对路径名，需要知道 IcePack 节点的当前工作目录，所以使用时要小心。

20.10.4 服务器故障

要诊断服务器故障可能会很困难，特别是在服务器自动在远地主机上激活的情况下。下面有一些建议:

1. 如果服务器运行在 UNIX 主机上，检查 IcePack 节点进程的当前工作目录，看是否有服务器出故障的迹象，比如 `core` 文件。
2. 明智地使用跟踪功能有助于缩小查找的范围。例如，如果故障是因为发出某个操作调用而发生的，那么你可以设置配置属性 `Ice.Trace.Protocol=1`，启用 Ice run time 中的协议跟踪，从而找出所有请求的对象标识和操作名。

   当然，如果服务器是自动激活的，缺省的日志输出信道 (标准输出和标准错误) 很可能会丢失，所以你应该手工启动服务器 (见下)，或是重定向日志输出 (对 `Ice.UseSyslog` 的描述，参见附录 C)。

   你还可以使用 `Ice::Logger` 接口来记录你自己的跟踪消息。
3. 在某个调试器中运行服务器；如果有必要，被配置成自动激活的服务器也可以手工启动。但是，在这种情况下，因为服务器不是由 IcePack 节点激活的，它无法监控服务器进程，因而也就不知道服务器何时终止。这会使得后续的激活无法进行，除非你在完成调试、终止服务器之后清理 IcePack 的状态。你可以使用 `icepackadmin` (参见 20.7 节) 来完成上述操作 :

   ```
   >>> server start TheServer
   ```

   这将让 IcePack 节点激活 (从而监控) 服务器进程。如果你不想让服务器继续运行，你可以用对等的 **`icepackadmin`** 命令来停止它。

4. 在服务器被激活成好像处在静止状态之后，把你的调试器关联到正在运行的服务器进程。这避开了与手工启动服务器相关联的问题 (就像前一步所描述的那样)，但在定制服务器的启动环境方面提供的灵活性没有那么多。

20.11 总结

这一章详细讨论了 IcePack，包括为了把 IcePack 结合进客户和服务器应用、所需作出的修改，以及 IcePack 组件的配置和管理。一旦你理解了 IcePack 的基础概念，你将很快开始欣赏 IcePack 的各种能力所具有的灵活性、威力，以及便利性:

- 自动服务器激活提高了可靠性，并且能更高效地利用处理资源
- 定位器设施简化了管理的要求，并且使客户和服务器之间的耦合降到了最低限度
- 部署机制把平常的配置任务简化成了一组易于理解的、可复用的 XML 文件

简而言之，IcePack 提供的各种工具正是开发健壮的企业级 Ice 应用所需要的。

第 21 章
Freeze

21.1 本章综述

这一章将描述怎样用 Freeze 给 Ice 应用增加持久能力。 21.3 节讨论 Freeze 映射表，并演示怎样在一个简单的例子中使用它。 21.4 节考察一个使用 Freeze 映射表的文件系统实现。 21.5 节介绍 Freeze 逐出器， 21.6 节在另一个文件系统实现中演示了 Freeze 逐出器。

21.2 引言

如图 21.1 所示， Freeze 代表的是一组持久服务。

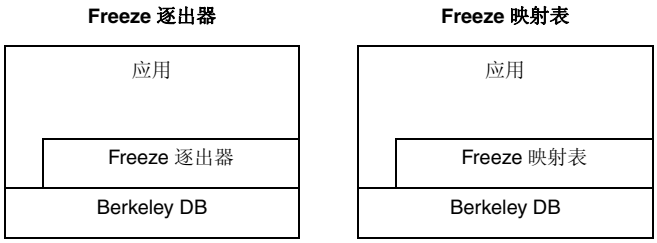

图 21.1. Freeze 持久服务的层次图

下面是对 Freeze 持久服务的描述:

- Freeze 逐出器是 Ice servant 定位器的一种可高度伸缩的实现，采用 Freeze 逐出器，只用很少的应用代码就能进行自动持久和 Ice 对象逐出。
- Freeze 映射表是一种泛型关联容器。你可以使用 Ice 提供的代码生成器、为任何 Slice 键和值类型产生特定类型的映射表。应用与 Freeze 映射表的交互方式就像与其他任何关联容器交互一样，但 Freeze 映射表的键和值是持久的。

你将在本章的例子中看到，要把 Freeze 映射表或逐出器集成进你的 Ice 应用相当直截了当：一旦你用 Slice 定义了你的持久数据， Freeze 将负责管理持久的各种平常的细节。

Freeze 是用 Berkeley DB 实现的，这是一个紧凑的、高性能的嵌入式数据库。 Freeze 映射与逐出器 API 使应用与 Berkeley DB API 隔离开来，但并不会阻止应用在必要时直接与 Berkeley DB 交互。

21.3 Freeze 映射表

Freeze 映射表是一个持久的关联容器，其中的键和值的类型可以是任何原始的或用户定义的 Slice 类型。对于每一对键和值类型，开发者都要使用一个代码生成工具来产生一个使用特定语言的类，这个类将遵从该语言中的映射表的标准约定。例如，在 C++ 中，所生成的类与 STL `map` 类似，而在 Java 中，这个类将实现 `java.util.Map` 接口。大多数在数据库中存储和读取状态的逻辑都是在一个 Freeze 基类中实现的。所生成的映射表类派

生自这个基类，所以它们所含的代码很少，因而在代码尺寸方面的效率很高。

在 Freeze 映射表中，你只能存储用 Slice 定义的数据类型。你不能存储没有 Slice 定义的类型 (也就是说，任意的 C++ 或 Java 类型)，因为 Freeze 映射表要复用 Ice 生成的整编代码来在数据库中创建数据的持久表示。如果你要定义一个 Slice 类，在 Freeze 映射表中存储其实例，记住上面所说的尤其重要；只有 "public" (用 Slice 定义的) 数据成员才会被存储，派生的实现类的 "private" 状态成员则不会被存储。

21.3.1 Freeze 连接

要创建 Freeze Map 对象，你首先要连接到数据库环境，从而获得 Freeze `Connection` 对象。

如图 21.2 所示，一个 Freeze Map 与单个连接和单个数据库文件关联在一起。连接和映射表对象是 “单线程化的”：如果你想要在多个线程中使用某个连接或与它相关联的任何映射表，你必须对访问进行序列化。如果你的应用需要并发访问同一个数据库文件 (持久的 Map)，你必须创建多个连接及相关联的映射表。

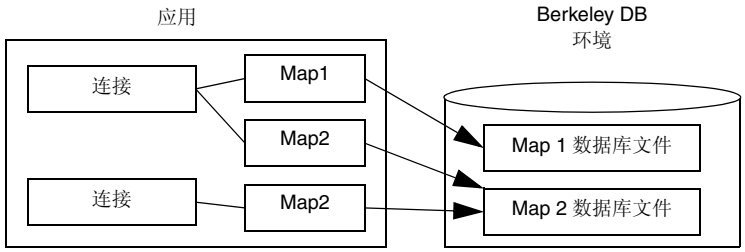

图 21.2. Freeze 连接与映射表

21.3.2 事务

在使用 Freeze 映射表时你也可以使用事务 (transaction)。 Freeze 事务提供了惯常的 ACID (atomicity, concurrency, isolation, durability) 属性。例如，事务允许你把若干数据库更新归总在一个原子单元中：这个事务中的更新或者全部发生，或者一个都不发生。

要启动事务，要使用 `Connection` 对象上的 `beginTransaction` 操作。一旦连接有了一个相关联的事务，在与这个连接相关联的映射表对象上进行

的所有操作都将使用这个事务。最后，你调用 commit 或 rollback 来结束事务：commit 会保存你的所有更新，而 rollback 会撤销它们。

```
module Freeze {

local interface Transaction {
    void commit();
    void rollback();
};

local interface Connection {
    Transaction beginTransaction();
    nonmutating Transaction currentTransaction();
    // ...
};
};
```

如果你没有使用事务，每一次非迭代器更新都会包含在它自己的内部事务中，而每一个读 - 写迭代器都有一个相关联的内部事务，将在迭代器关闭时提交。

21.3.3 迭代器

迭代器允许你访问 Freeze 映射表的内容。迭代器是用 Berkeley DB 的游标 (cursor) 实现的，会获取底层的数据库页面文件上的锁。在 C++ 中，可以使用只读 (const_iterator) 和读 - 写迭代器 (iterator)；在 Java 中，只支持读 - 写迭代器。

迭代器持有的锁会在迭代器关闭时释放 (如果你没有使用事务)，或在它所属的事务结束时释放。释放迭代器所持有的锁非常重要，这样其他线程才能通过其他连接和映射表对象访问相应的数据库文件。有时甚至必须通过释放锁来避免自锁 (永远在等待由同一个线程创建的迭代器持有的锁)。

为了提供易用性，降低自锁的可能性， Freeze 常常会自动关闭迭代器。当你启动或结束一个事务时， Freeze 会关闭与对应的映射表相关联的所有迭代器。如果你没有使用事务，任何对映射表的写操作 (比如插入新元素) 都会自动关闭在该映射表对象上打开的所有迭代器，除了写操作所使用的当前迭代器。在 Java 中，迭代器在其所属的映射表或连接被应用关闭时也会关闭。

但是，在有一种情况下，为了避免自锁，你需要显式关闭迭代器：

- 你没有使用事务
- 你有一个已经打开的迭代器，用于更新映射表 (它持有一个写锁)

- 在同一个线程中，你读取映射表。

读取操作不会自动关闭迭代器。在这种情况下，你需要使用事务，或是显式关闭持有写锁的迭代器。

21.3.4 从死锁异常中恢复

如果你用多个线程访问数据库文件， Berkeley DB 可能会按有冲突的次序获取锁 (代表不同的事务或迭代器进行获取)。例如，某个迭代器拥有页面 P1 上的读锁，并试图获取页面 P2 上的写锁，而另一个迭代器 (在与同一个数据库文件相关联的另一个映射表对象上) 拥有 P2 上的读锁，并试图获取 P1 上的写锁。

在发生这样的情况时， Berkeley DB 会检测到死锁，并通过把 “死锁” 错误返回给一个或多个线程来解决这个问题。对于在任何事务之外执行的所有非迭代器操作，比如在映射表中插入元素， Freeze 会捕捉这样的错误，并自动重试操作，直到成功为止。对于其他操作， Freeze 会引发 `Freeze::DeadlockException`，报告这个死锁。在这种情况下，与之相关联的事务或迭代器也会自动回滚或关闭。编写得当的应用应该捕捉死锁异常，重试事务或迭代。

21.3.5 在 C++ 中使用一个简单的映射表

作为一个例子，下面的命令会生成一个简单的映射表:

```
$ slice2freeze --dict StringIntMap,string,int StringIntMap
```

`slice2freeze` 编译器会为 Freeze 映射表创建 C++ 类。上面的命令指示编译器，创建一个名为 `StringIntMap` 的映射表，其 Slice 键类型是 `string`，Slice 值类型是 `int`。最后一个参数是输出文件的基名字，编译器将在其后附加 `.h` 和 `.cpp` 后缀。因此，这个命令将产生两个 C++ 源文件: `StringIntMap.h` 和 `StringIntMap.cpp`。

下面是一个简单的程序，演示怎样使用 `StringIntMap`、在数据库中存储 `<string, int>` 对。你会注意到，程序没有显式地调用 `read` 或 `write` 操作；相反，使用这个映射表有副作用: 读写数据库。

```
#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;
}
```

在实例化 Freeze 映射表之前，应用必须先连接到 Berkeley DB 数据库环境。

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator, "db");
```

第二个参数是 Berkeley DB 数据库环境的名字；在缺省情况下，这也是 Berkeley DB 用于创建所有数据库和管理文件的文件系统目录。

接下来，程序在连接上实例化 `StringIntMap`。构造器的第二个参数是数据库文件名，在缺省情况下，如果这个文件不存在，就会被创建。注意，数据库只能容纳一种映射表类型的持久状态。如果你试图在同一个数据库上实例化不同类型的映射表，就会导致不确定的行为。

```
StringIntMap map(connection, "simple");
```

接下来，我们清除映射表。这确保了在程序多次运行的情况下，我们的映射表是空的 (因此数据库也是空的)。

```
map.clear();
```

我们用单个字符的串作为键来填充映射表。Freeze 映射表与 STL 映射表类似，支持多个用于添加条目的 `insert` 方法。通过 `operator[]` 进行插入不支持。

```
for (i = 0; i < 26; i++) {
    std::string key(1, 'a' + i);
    map.insert(make_pair(key, i));
}
```

遍历映射表的方式在 STL 用户看来会很熟悉。但是，要修改迭代器的当前位置处的值，你必须使用非标准的 `set` 方法:

```
for (p = map.begin(); p != map.end(); ++p)
    p.set(p->second + 1);
```

接下来，程序获取一个迭代器，其所指向的元素的键为 `z`；然后把它消除掉。

```
p = map.find("z");
assert(p != map.end());
map.erase(p);
```

最后，程序关闭数据库连接，销毁它的通信器，然后终止。

```
connection->close();
```

显式关闭数据库连接并非必要，但为了完整起见，我们仍在这里演示了它。

21.3.6 在 Java 中使用一个简单的映射表

作为一个例子，下面的命令会生成一个简单的映射表:

```
$ slice2freezej --dict StringIntMap,string,int
```

slice2freezej编译器会为Freeze映射表创建Java类。上面的命令指示编译器，创建一个名为 StringIntMap 的映射表，其 Slice 键类型是 string， Slice 值类型是 int。这个命令将产生一个 Java 源文件: StringIntMap.java。

下面是一个简单的程序，演示怎样使用 StringIntMap、在数据库中存储 <string, int> 对。你会注意到，程序没有显式地调用 read 或 write 操作；相反，使用这个映射表有副作用: 读写数据库。

```
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);
    }
}
```

在实例化 Freeze 映射表之前，应用必须先连接到 Berkeley DB 数据库环境。

```
Freeze.Connection connection =
    Freeze.Util.createConnection(communicator, "db");
```

第二个参数是 Berkeley DB 数据库环境的名字；在缺省情况下，这也是 Berkeley DB 用于创建所有数据库和管理文件的文件系统目录。

接下来，程序在连接上实例化 `StringIntMap`。构造器的第二个参数是数据库文件名，第三个参数是一个标志，说明如果数据库不存在，是否应该创建它。注意，数据库只能容纳一种映射表类型的持久状态。如果你试图在同一个数据库上实例化不同类型的映射表，就会导致不确定的行为。

```
StringIntMap map = new StringIntMap(connection, "simple", true);
```

接下来，我们清除映射表。这确保了在程序多次运行的情况下，我们的映射表是空的 (因此数据库也是空的)。

```
map.clear();
```

我们用单个字符的串作为键来填充映射表。和使用 java.util.Map 时一样，键和值的类型必须是 Java 对象。

```
for (i = 0; i < 26; i++) {
    final char[] ch = { (char)('a' + i) };
    map.put(new String(ch), new Integer(i));
}
```

遍历这个映射表和遍历其他任何实现了 java.util.Map 接口的映射表没有区别:

```
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));
}
```

接下来，程序检验有键为 z 的元素存在，然后移除它。注意，程序使用了非标准的方法来移除这个元素。 fastRemove 方法与标准的 remove 方法有所不同: 它不会返回与被移除的元素相关联的值; 相反，它返回的是一个布尔值，表明是否找到了该元素。因为没有返回元素值，所以这个方法避免了从数据库中读取值、并进行解码的开销。在这里也可以使用标准的 remove 方法; 我们选择使用 fastRemove，只是为了演示它的用法:

```
b = map.containsKey("z");
assert(b);
b = map.fastRemove("z");
assert(b);
```

最后，程序关闭映射表及其连接。

```
map.close();
connection.close();
```

21.4 在文件系统服务器中使用 Freeze 映射表

我们可以用 Freeze 映射表来给文件系统服务器增加持久能力，在这一节，我们将给出 C++ 实现和 Java 实现。但是，你将在 21.5 节看到，如果在一个应用中（比如文件系统服务器），要持久的值是 Ice 对象， Freeze 逐出器常常是一种更好的选择。

一般而言，要把 Freeze 映射表结合进你的应用中，需要采取以下步骤:

1. 考察你已有的 Slice 定义，确定合适的键和值类型。
2. 如果没有找到合适的键或值类型，就定义新的（可能是派生的）、能满足你的持久状态需求的类型。考虑把这些定义放在单独的文件里：这些类型只被服务器用于进行持久，因此，无需出现在客户需要的 "public" 定义中。还可以考虑把你的持久类型放在单独的模块中，以避免名字冲突。
3. 使用 Freeze 编译器，为你的持久类型生成 Freeze 映射表。
4. 在您的操作实现中使用 Freeze 映射表。

21.4.1 选择键和值类型

我们的目标是在实现文件系统时、使用 Freeze 映射表来进行所有持久存储，包括文件及其内容。我们的第一步是要选择我们的映射表的 Slice 键类型和值类型。我们将保持 XREF 中的基本设计不变，因此，我们需要一种合适的用于持久文件和目录的表示，以及用作键的唯一标识符。

足够方便的是， Ice 对象已经有一个类型为 `Ice::Identity` 的唯一标识符，也可以很好地用作我们映射表的键。

遗憾的是，值类型的选择更为复杂。Unfortunately, the selection of a value type is more complicated. 查看 XREF 中的 `Filesystem` 模块，我们没有发生任何类型能捕捉我们的所有持久状态，所以我们需要用一些新类型来扩展该模块:

```
module Filesystem {
    class PersistentNode {
        string name;
    };

    class PersistentFile extends PersistentNode {
        Lines text;
    };

    class PersistentDirectory extends PersistentNode {
        NodeDict nodes;
    };
};
```

因此，我们的 Freeze 映射表将把 `Ice::Identity` 映射到 `PersistentNode`，其中的值实际上是派生类 `PersistentFile` 或 `PersistentDirectory` 的实例。如果我们遵循 21.4 节一开头的建议，我们本应该在单独的 `PersistentFilesystem` 模块中定义 `File` 和 `Directory` 类，但在这个例子中，为了简单起见，我们使用了已有的 `Filesystem` 模块。

21.4.2 用 C++ 实现文件系统服务器

在这一节，我们将给出一个 C++ 文件系统实现，利用了 Freeze 映射表进行持久存储。这个实现基于在 XREF 中讨论的实现，而在这一节中，我们将只讨论那些能说明 Freeze 映射表的用法的代码。

生成映射表

现在，我们已经选择了我们的键和值类型，我们可以这样来生成映射表:

```
$ slice2freeze -I$(ICE_HOME)/slice --dict \
    IdentityNodeMap,Ice::Identity,Filesystem::PersistentNode\
    IdentityNodeMap Filesystem.ice \
    $(ICE_HOME)/slice/Ice/Identity.ice
```

生成的映射表类名为 IdentityNodeMap。

服务器 main 程序

服务器的 main 程序负责初始化根目录节点。如 10.3.1 节所述，许多管理任务，比如创建和销毁通信器，都是由 Ice::Application 类来完成的。现在，我们的服务器 main 程序变成了这样:

```
#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);
}
```

让我们详细考察一下所做的变动。首先，我们现在包括的是 `Freeze/Freeze.h`，而不是 `Ice/Ice.h`。这个 Freeze 头文件包括了这个源文件所需的其他全部 Freeze (以及 Ice) 头文件。

接下来，我们把 `FilesystemApp` 类定义成 `Ice::Application` 的子类，并提供了一个构造器，其参数是一个串:

```
  FilesystemApp(const string & envName) :
      _envName(envName) { }
```

串参数表示的是数据库环境名，会被保存下来，以备后面在 `run` 中使用。

`run`首先要执行的任务之一是为`PersistentFile`和`PersistentDirectory`安装 Ice 对象工厂。尽管这两个类不通过 Slice 操作进行交换，当它们被保存到数据库中、或从中加载时，它们的整编和解编是完全一样的，所以需要使用工厂。因为这两个 Slice 类没有操作，我们可以使用它们内建的工厂。

```
communicator()->addObjectFactory(
    PersistentFile::ice_factory(),
    PersistentFile::ice_staticId());

communicator()->addObjectFactory(
    PersistentDirectory::ice_factory(),
    PersistentDirectory::ice_staticId());
```

接下来，我们设置 NodeI 的所有静态成员：

```
NodeI::_adapter = communicator()->
    createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

NodeI::_communicator = communicator();
NodeI::_envName = _envName;
NodeI::_dbName = "mapfs";
```

然后我们创建一个 Freeze 连接和一个 Freeze 映射表。当与某个 Berkeley DB 环境的最后一个连接关闭时， Freeze 会自动关闭该环境，所以，在 main 函数中保存一个连接，能确保底层的 Berkeley DB 环境不被关闭。同样，我们在 main 函数中保存了一个映射表，以使底层的 Berkeley DB 数据库不被关闭。

```
Freeze::ConnectionPtr connection =
    Freeze::createConnection(communicator(), _envName);
IdentityNodeMap persistentMap(connection, NodeI::_dbName);
```

现在，我们需要初始化根目录节点。我们首先在映射表中查询根目录节点的标识；如果没有找到，我们就创建一个新的 PersistentDirectory 实例，并把它插入映射表中。我们使用了一个作用域来在使用完毕后关闭迭代器；否则的话，这个迭代器会一直锁住，使得我们无法再通过其他连接访问映射表。

```
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));
            }
        }
```

最后， main 函数实例化 FilesystemApp，把 db 作为数据库环境名传给它。

```
int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}
```

服务器类定义

我们还必须改动 servant 类，把 Freeze 映射表结合进来。我们维持了 XREF 的多继承设计，但我们增加了一些方法，并改动了构造器参数和状态成员。

首先让我们来考察 NodeI 的定义。你会注意到，我们增加了 getPersistentNode 方法，允许 NodeI 为了实现 Node 操作而访问持久节点。另一种可能的做法是给 NodeI 增加 PersistentNodePtr 成员，但这会迫使 FileI 和 DirectoryI 类把这个成员向下转换成适当的子类。

```
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;
    };
}
```

NodeI 中的另一处有意思的变动是增加了 _map 成员和 _destroyed 成员，同时我们还改动了 NodeI 的构造器，让它接受一个 Ice::Identity 参数。

现在，FileI 类有了一个类型为 PersistentFilePtr 的状态成员，表示的是这个文件的持久状态。其构造器也做了改动，接受 Ice::Identity 和 PersistentFilePtr 参数。

```
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;
    };
}
```

DirectoryI 类也进行了类似的改动。

```
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;
    };
}
```

实现 FileI

让我们考察一下实现发生了哪些变化。FileI 的方法仍然相当简单，但有一些方面需要讨论。

首先，现在每个操作都会检查 _destroyed 成员，如果该成员为 true，就引发 Ice::ObjectNotExistException。这是必要的，目的是确保 Freeze 映射表处在一致的状态中。例如，如果我们允许 write 操作在文件节点销毁之后执行，我们就可能会错误地把一个与该文件对应的条目添加进 Freeze 映射表中。先前的文件系统实现忽略了这个问题，因为它相对而言是无害的，但在这个版本中情况却不再是如此。

接下来，注意 write 操作会在改动了 PersistentFile 对象的 text 成员之后调用映射表的 put 方法。尽管文件的 PersistentFilePtr 成员指向的是 Freeze 映射表中的值，在这个值被重新插入映射表、从而覆盖先前的值之前，改动这个值对映射表的持久状态不起作用。

最后，构造器现在接受一个 Ice::Identity 参数。这与先前的实现不同：标识原来是由 NodeI 构造器创建的。但我们在后面会看到，调用者需要在调用子类构造器之前确定标识 。与此类似，构造器并不负责创建 PersistentFile 对象，而是会接受传入的对象。这适用于我们的两种用例：创建新文件，以及从映射表中恢复 (restore) 已有文件。

```
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;
}
```

实现 DirectoryI

DirectoryI 实现需要进行更实质性的改变。我们将首先讨论 create-Directory 操作。

```
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;
}
```

在验证了节点名之后，这个操作为子节点创建一个 PersistentDirectory[1] 对象，连同一个唯一标识符一起传给 DirectoryI 构造器。接下来，我们把子节点的 PersistentDirectory 对象存储在 Freeze 映射表中。最后，我们初始化一个新的 NodeDesc 值，把它插入父节点的节点表中，然后把父节点的 PersistentDirectory 对象重新插入 Freeze 映射表中。

createFile 操作的实现的结构和 createDirectory 是一样的。

```
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()),
        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;
```

1. 因为这个 Slice 类没有操作，编译器会生成应用能够实例化的具体类。

```
    _map.put(IdentityNodeMap::value_type(_ID, _dir));

    return proxy;
}
```

下一个重大变动是在 DirectoryI 构造器中。现在，这个构造器的主体会实例化它的所有直接的子节点，从而在事实上造成所有节点被递归地实例化。

对于目录的节点表中的每个条目，构造器都会在 Freeze 映射表中定位与之匹配的条目。这个 Freeze 映射表的键类型是 Ice::Identity，所以构造器会通过调用子节点的代理上的 ice_getIdentity 来获取键。

```
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);
        }
    }
}
```

如果你觉得 DirectoryI 构造器立刻实例化它的所有子节点效率不高，你是对的。显然，这种做法不能伸展到有大量节点的树，那么，我们为什么要这么做呢？

文件系统服务的前一种实现从 createDirectory 和 createFile 返回的是暂时的代理。换句话说，如果服务器被停止并重启，服务器的老实例返回的任何已有的子代理都不再有效。但是，现在我们有了一个持久存储

器，我们应该尽力确保，在服务器重启后，代理仍然有效。要满足这一需求，可以使用这样两种实现技术:

1. 像 DirectoryI 构造器那样，预先实例化所有 servant。
2. 使用 servant 定位器。

在这个例子中，我们选择不使用 servant 定位器，因为那会使实现复杂化，同时，我们将在 21.5 节看到， Freeze 逐出器是这种应用的理想解决方案，比使用 servant 定位器更好。

我们讨论的最后一个 DirectoryI 方法是 removeChild，这个方法用于从节点表中移除条目，然后把 PersistentDirectory 对象重新插入映射表，以使变动得以持久。

```
void
Filesystem::DirectoryI::removeChild(const string & name)
{
    IceUtil::RecMutex::Lock lock(_nodeMutex);

    _dir->nodes.erase(name);
    _map.put(IdentityNodeMap::value_type(_ID, _dir));
}
```

实现 NodeI

对 NodeI 实现的一些变动值得一提。首先，你会注意到，为了获取节点名，使用了 getPersistentNode。我们也可以调用 name 操作，但那需要进行另一次互斥体加锁，从而带来额外的开销。

随后， destroy 操作把节点从 Freeze 映射表中移除，并把 _destroyed 成员设成真。

NodeI 构造器不再计算标识的值。为了使我们的 servant 真正持久，这样做是必需的。特别地，某个节点的标识会在该节点创建时一次性计算出来，而且，在节点的生命期内，必须保持不变。因此， NodeI 构造器不能计算新标识，而是要记住传给它的标识。 NodeI 构造器还要构造映射表对象 (_map 数据成员)。要记住，这个对象是单线程化的: 我们只在构造器中、或拥有递归的读 - 写 _nodeMutex 上的写锁的情况下，才使用这个对象。

最后， find 函数 说明了，在遍历可由多个线程同时使用的数据库时，需要做些什么。 find 会捕捉 Freeze::DeadlockException 并重试。

```
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 用 Java 实现文件服务器

在这一节，我们将给出一个 Java 文件系统实现，利用了 Freeze 映射表进行持久存储。这个实现基于在 XREF 中讨论的实现，而在这一节中，我们将只讨论那些能说明 Freeze 映射表的用法的代码。

Generating the Map

现在，我们已经选择了我们的键和值类型，我们可以这样来生成映射表:

```
$ slice2freezej -I$(ICE_HOME)/slice --dict \
    Filesystem.IdentityNodeMap,Ice::Identity,\
    Filesystem::PersistentNode\
    Filesystem.ice $(ICE_HOME)/slice/Ice/Identity.ice
```

生成的映射表类名为 `IdentityNodeMap`，在 `Filesystem`[2] 包中定义。

服务器 `main` 程序

服务器的 `main` 程序负责初始化根目录节点。如 12.3.1 节所述，许多管理任务，比如创建和销毁通信器，都是由 `Ice.Application` 类来完成的。现在，我们的服务器 `main` 程序变成了这样:

```
import Filesystem.*;

public class Server extends Ice.Application {
    public
    Server(String envName)
```

2. 我们不能在无名的 (顶层的) 包中生成 `IdentityNodeMap`，因为 Java 的名字解析规则会阻止 `Filesystem` 包中的实现类使用它。更多信息，参见 Java Language Specification。

```
{
    _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 =
        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;
}
```

让我们详细考察一下所做的变动。首先，我们把 Server 类定义成 Ice.Application 的子类，并提供了一个构造器，其参数是一个串：

```
public
Server(String envName)
{
    _envName = envName;
}
```

串参数表示的是数据库环境名，会被保存下来，以备后面在 run 中使用。

run首先要执行的任务之一是为PersistentFile和PersistentDirectory安装 Ice 对象工厂。尽管这两个类不通过 Slice 操作进行交换，当它们被保存到数据库中、或从中加载时，它们的整编和解编是完全一样的，所以需要使用工厂。因为这两个 Slice 类没有操作，我们可以使用它们内建的工厂。

```
communicator().addObjectFactory(
    PersistentFile.ice_factory(),
    PersistentFile.ice_staticId());
communicator().addObjectFactory(
    PersistentDirectory.ice_factory(),
    PersistentDirectory.ice_staticId());
```

接下来，我们设置 DirectoryI 和 FileI 的所有静态成员。

```
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;
```

然后我们创建一个 Freeze 连接和一个 Freeze 映射表。当与某个 Berkeley DB 环境的最后一个连接关闭时， Freeze 会自动关闭该环境，所以，在 main 函数中保存一个连接，能确保底层的 Berkeley DB 环境不被关闭。同样，我们在 main 函数中保存了一个映射表，以使底层的 Berkeley DB 数据库不被关闭。

```
Freeze.Connection connection =
    Freeze.Util.createConnection(
        communicator(), _envName);
IdentityNodeMap persistentMap =
    new IdentityNodeMap(connection, dbName, true);
```

现在，我们需要初始化根目录节点。我们首先在映射表中查询根目录节点的标识；如果没有找到，我们就创建一个新的 PersistentDirectory 实例，并把它插入映射表中。

```
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);
}
```

最后， main 函数实例化 Server 类，把 db 作为数据库环境名传给它。

```
public static void
main(String[] args)
{
    Server app = new Server("db");
    app.main("Server", args);
    System.exit(0);
}
```

服务器类定义

我们还必须改动 servant 类，把 Freeze 映射表结合进来。我们维持了 XREF 的设计，但我们增加了一些方法，并改动了构造器参数和状态成员。

FileI 类有三个新的静态成员： _communicator、 _envName 和 _dbName，用于在每个 FileI 对象中实例化新的 _connection (Freeze 连接) 和 _map (IdentityNodeMap) 成员。我们会保存该连接，这样，当我们不再需要它时，我们就可以显式地关闭它。

这个类还有了一个新的实例成员，类型是 PersistentFile，代表这个文件的持久状态，还有一个 boolean 成员，表示这个节点是否已被销毁。最后，我们改变了它的构造器，让它接受 Ice.Identity 和 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;
}
```

DirectoryI 类也进行了类似的改动。

```
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;
}
```

实现 FileI

让我们考察一下实现发生了哪些变化。 FileI 的方法仍然相当简单，但有一些方面需要讨论。

首先，现在每个操作都会检查 _destroyed 成员，如果该成员为 true，就引发 Ice::ObjectNotExistException。这是必要的，目的是确保 Freeze 映射表处在一致的状态中。例如，如果我们允许 write 操作在文件节点销毁之后执行，我们就可能会错误地把一个与该文件对应的条目添加进 Freeze 映射表中。先前的文件系统实现忽略了这个问题，因为它相对而言是无害的，但在这个版本中情况却不再是如此。

接下来，注意 write 操作会在改动了 PersistentFile 对象的 text 成员之后调用映射表的 put 方法。尽管这个对象是 Freeze 映射表中的一个值，在这个值被重新插入映射表、从而覆盖先前的值之前，改动 text 成员对映射表的持久状态不起作用。

最后，构造器现在接受一个 Ice::Identity 参数。这与先前的实现不同：标识原来是由构造器创建的。为了使我们的 servant 真正持久，节点的标识会在该节点创建时一次性计算出来，而且，在节点的生命期内，必须保持不变。因此， NodeI 构造器不能计算新标识，而是要记住传给它的标识。

与此类似，构造器并不负责创建 PersistentFile 对象，而是会接受传入的对象。这适用于我们的两种用例：创建新文件，以及从映射表中恢复 (restore) 已有文件。

```
    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);
    }
```

实现 DirectoryI

DirectoryI 实现需要进行更实质性的改变。我们将首先讨论 create-Directory 操作。

```
    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;
    }
```

在验证了节点名之后，这个操作为子节点创建一个 PersistentDirectory[3] 对象，连同一个唯一标识符一起传给 DirectoryI 构造器。接下来，我们把子节点的 PersistentDirectory 对象存储在 Freeze 映射表中。最

后，我们初始化一个新的 NodeDesc 值，把它插入父节点的节点表中，然后把父节点的 PersistentDirectory 对象重新插入 Freeze 映射表中。

createFile 操作的实现的结构和 createDirectory 是一样的。

```
public synchronized FilePrx
createFile(String name, Ice.Current current)
    throws NameInUse, IllegalName
{
    if (_destroyed)
        throw new Ice.ObjectNotExistException();

    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;
}
```

下一个重大变动是在 DirectoryI 构造器中。现在，这个构造器的主体会实例化它的所有直接的子节点，从而在事实上造成所有节点被递归地实例化。

对于目录的节点表中的每个条目，构造器都会在 Freeze 映射表中定位与之匹配的条目。这个 Freeze 映射表的键类型是 Ice::Identity，所以构造器会通过调用子节点的代理上的 ice_getIdentity 来获取键。

3. 因为这个 Slice 类没有操作，编译器会生成应用能够实例化的具体类。

```
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);
        }
    }
}
```

如果你觉得 DirectoryI 构造器立刻实例化它的所有子节点效率不高，你是对的。显然，这种做法不能伸展到有大量节点的树，那么，我们为什么要这么做呢？

文件系统服务的前一种实现从 createDirectory 和 createFile 返回的是暂时的代理。换句话说，如果服务器被停止并重启，服务器的老实例返回的任何已有的子代理都不再有效。但是，现在我们有了一个持久存储器，我们应该尽力确保，在服务器重启后，代理仍然有效。要满足这一需求，可以使用这样两种实现技术：

1. 像 DirectoryI 构造器那样，预先实例化所有 servant。
2. 使用 servant 定位器。

在这个例子中，我们选择不使用 servant 定位器，因为那会使实现复杂化，同时，我们将在 21.5 节看到， Freeze 逐出器是这种应用的理想解决方案，比使用 servant 定位器更好。

我们讨论的最后一个 DirectoryI 方法是 removeChild，这个方法用于从节点表中移除条目，然后把 PersistentDirectory 对象重新插入映射表，以使变动得以持久。

```
synchronized void
removeChild(String name)
{
    _dir.nodes.remove(name);
    _map.put(_ID, _dir);
}
```

21.5 Freeze 逐出器

Freeze 逐出器把持久和可伸缩性特性结合进一种设施中，你可以在 Ice 应用中轻松地使用这种设施。

Freeze 逐出器是 ServantLocator 接口（参见 16.6 节）的一种实现，它利用了 Ice 对象和 servant 之间的基础性分离，在需要时从持久存储器中激活 servant，并使用定制的逐出约束来解除它们的激活。尽管应用在数据库中可能有数千个 Ice 对象，让所有这些 Ice 对象的 servant 都同时驻留在内存中并不现实。通过设置活动 servant 的上限，并让 Freeze 逐出器来处理 servant 的激活、持久、解除激活，应用可以节省资源，并获得更大的可伸缩性。

Freeze 逐出器维护有一个活动 servant 队列，按照 “最近最少使用” 逐出算法来排定次序：如果队列满了，最近最少使用的 servant 就会被逐出，为新的 servant 让出空间。

在图 21.3 中给出了激活 servant 时的事件序列。让我们假定，我们把逐出器的尺寸配置成五，该队列是满的，有一个请求已经到达，相应的 servant 目前不是活动的。

1. 客户调用某个操作。
2. 对象适配器调用逐出器，定位 servant。
3. 逐出器首先检查它的活动 servant 队列，发现找不到所需 servant，所以它实例化该 servant，从数据库中恢复它的持久状态。

4. 逐出器在队头为该 servant (servant 1) 增加一个项。
5. 现在，队列的长度超过了所配置的最大尺寸，所以只要 servant 6 可以被逐出，逐出器就会把 servant 6 从队列中移除。只要 servant 6 上没有正在执行的请求，而且其状态已完全地保存在数据库中，它就可以被逐出。
6. 对象适配器把请求分派给新的 servant。

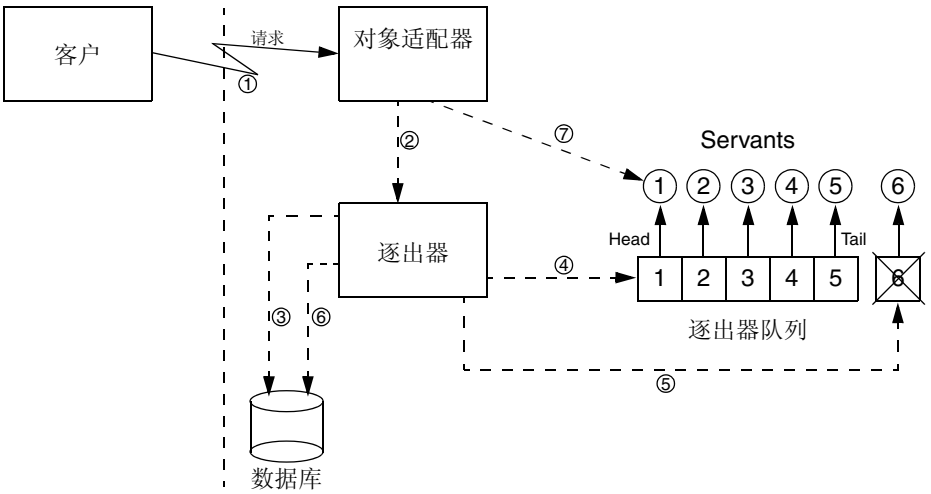

图 21.3. 一个逐出器队列，在恢复 servant 1、逐出 servant 6 之后

21.5.1 对象工厂

Freeze 逐出器是一种泛型设施：它管理的是 `Object` 的子类的实例。因此，应用可以按照需要，自由地使用任意多种对象类型；要求只有一个：必须为每种类型登记一个 Ice 对象工厂。

21.5.2 Facets

一个 facet 是与某个 Ice 对象系在一起的一个任意类型的 Ice 对象。 facet 也可以有 facet，从而可以创建出对象树，用*主对象*做它的根。尽管客户可以调用对对象上的 facet，只有主对象才被认为是 servant。换句话说，如果

应用使用了对象适配器，只有主对象才会被注册，而它的任何 facet 都不会。

在使用持久设施时，需要给予 facet 特殊的考虑。一种低效的做法是，只要某个 facet 被修改了，就保存主对象及其所有 facet 的状态。与此不同，Freeze 逐出器在保存状态时，把 facet 当作离散的实体：在数据库中，每个 facet 都拥有自己的记录，可以独立于主对象和其他 facet 进行保存。

但在有些情况下， Freeze 逐出器会把主对象及其 facet 当作一个逻辑单元：

- 在用 `createObject` 第一次注册某个 Ice 对象时，逐出器会保存主对象的持久状态，以及目前与主对象系在一起的任何 facet 的持久状态。
- 在为了响应进入的请求而激活某个对象时，主对象及其所有 facet 都会根据持久状态进行恢复，不管进入的请求是针对主对象的、还是针对某个 facet 的。
- 只有当主对象被逐出时， facet 才会被逐出。

更多关于 Ice facet 的信息，参见 XREF。

21.5.3 检测更新

如果在某个 facet 上完成了一个 mutating 操作， Freeze 逐出器就认为这个 facet 已被修改过了 (更多关于 mutating 操作的信息，参见 See 4.8 节)。不是通过 Freeze 逐出器分派的、在内部进行的更新，不会被检测到，可能会丢失。

21.5.4 负责进行保存的线程

一个 Freeze 逐出器的所有持久活动都由逐出器创建的一个后台线程处理。这个线程周期性地苏醒过来，保存逐出器队列中所有新注册的、修改过的，以及被销毁的 Ice facet 的状态。

如果应用爆发了大量活动，致使短期内有大量 Ice facet 被修改，你可以对逐出器的线程进行配置，只要被修改的 facet 的数目达到一定的阀值，就马上保存 facet 状态。

21.5.5 同步

当负责进行保存的线程摄取它要保存的 facet 的快照 (snapshot) 时，它必须阻止应用同时也去修改 facet 的持久数据成员。

Freeze 逐出器和应用需要共同进行同步，确保正确的行为。在 Java 中，这个共同的同步机制就是 facet 自身：在摄取快照时， Freeze 逐出器会锁住

(一个 Java 对象 object)。在 C++ 中， facet 必须从 `IceUtil::AbstractMutex` 类继承：Freeze 逐出器会在摄取快照时通过这个接口进行加锁。在应用端， facet 的实现必须使用同样的机制、对访问 facet 的持久数据成员的所有操作进行同步。

21.5.6 索引

Freeze 逐出器支持使用索引：用某个数据成员的值做搜索标准，快速地找到持久对象。只有主 Ice 对象的数据成员能被索引，同时，它们的类型必须是那些能用作 Slice 词典键的类型 (参见 4.7.4 节)。

把 **`--index`** 传给 **`slice2freeze`** 和 **`slicefreezej`** 工具，它们就可以生成 `Index` 类：

- **`--index`** ***`CLASS`***`,`***`TYPE`***`,`***`MEMBER`***`[,case-sensitive|case-insensitive]`**

  `CLASS` 是要生成的类的名字。***`TYPE`*** 表示要进行索引的类的类型 (在这个索引中不包括不同的类的对象)。 ***`MEMBER`*** 是 ***`TYPE`*** 中要索引的数据成员的名字。当 ***`MEMBER`*** 的类型是 `string` 时，可以指定索引是大小写敏感的、还是不敏感的。缺省是大小写敏感。

生成的 `Index` 类提供了三个方法：

- `sequence<Ice::Identity> findFirst(`*`member-type`* `index, int firstN)`

  返回最多 `firstN` 个类型为 ***`TYPE`*** 的对象，其 ***`MEMBER`*** 与 `index` 相等。如果与搜索标准相匹配的对象的数目可能会非常大，这能够避免耗尽内存。

- `sequence<Ice::Identity> find(`*`member-type`* `index)`

  返回所有类型为 ***`TYPE`***、其 ***`MEMBER`*** 与 `index` 相等的对象。

- `int count(<type> index)`

  返回类型为 ***`TYPE`***、其 ***`MEMBER`*** 与 `index` 相等的对象的数目。

索引是在 Freeze 逐出器创建过程中、与逐出器关联在一起的。详情请参见 `createEvictor` 方法的定义。

索引过的搜索易于使用，而且非常高效。但要注意，索引会增加显著的写开销：在 Berkeley DB 中，每一次更新都会触发一次读取，从数据库中获得旧的索引条目，如果有必要，还要替换它。

21.5.7 使用 Servant 初始化器

在有些应用中，在 servant 被逐出器实例化之后、操作分派给它之前，必须对它进行初始化。 Freeze 逐出器允许应用为了这一目的而指定 servant 初始化器。

为了使事件的序列变得更明晰一点，让我们假定，针对某个 Ice 对象的请求已经到达，但这个对象目前不是活动的:

1. 逐出器从数据库中恢复这个 Ice 对象的 servant。这涉及两个步骤:
   1. Ice run time 定位并调用这个 Ice 对象的类型的工厂，从而获得一个新的实例，其数据成员还未初始化。系在上面的 facet 也会使用所注册的工厂重新创建。
   2. 根据持久状态对数据成员进行赋值。
2. 逐出器针对这个 servant 调用应用的 servant 初始化器 (如果有)。
3. 逐出器把这个 servant 添加到它的 “最近最少使用” 队列。
4. 逐出器分派操作。

当 servant 初始化器在对 servant 进行初始化的同时，逐出器会锁住它内部的数据结构。因此， servant 初始化器不应该发出可能回到同一个 Freeze 逐出器的远地调用。

第 608 页的 21.6 节给出的文件系统实现演示了 servant 初始化器的使用。

21.5.8 应用设计考虑事项

为了进行持久存储， Freeze 逐出器会通过整编 Ice 对象来创建这个 Ice 对象的状态的快照，就好像这个对象正在作为远地调用的参数 “在线路上” 发送一样。因此，对象类型的 Slice 定义必须包括一些数据成员，组成该对象的持久状态。

例如，我们可以这样定义 Slice 类:

```
class Stateless {
    void calc();
};
```

但没有数据成员，在数据库中就不会有这种类型的对象的持久状态，因此，把 Freeze 逐出器用于这种类型的价值就很小。

显然， Slice 对象类型需要定义数据成员，但另外还有一些要考虑的设计问题。例如，假定我们这样定义了一个简单应用:

```
class Account {
    void withdraw(int amount);
    void deposit(int amount);

    int balance;
};

interface Bank {
    Account* createAccount();
};
```

在这个应用中，我们要使用 Freeze 逐出器来管理 Account 对象，这种对象有一个数据成员 balance，用于表示账户的持久状态。

从面向对象的设计视角来看，这些 Slice 定义有一个很大的问题：实现细节 (持久状态) 暴露在了客户 - 服务器合约中。客户不能直接操纵 balance 成员，因为 Bank 接口返回的是 Account 代理，而不是 Account 实例。但是，数据成员的存在可能会给客户开发者带来不必要的困惑。

更好的做法是明确地把持久状态分开：

```
interface Account {
    void withdraw(int amount);
    void deposit(int amount);
};

interface Bank {
    Account* createAccount();
};

class PersistentAccount implements Account {
    int balance;
};
```

现在， Freeze 逐出器可以管理 PersistentAccount 对象，而客户可与 Account 代理进行交互 (在理想情况下，PersistentAccount 应该在另外一个源文件中定义，并放在单独的模块中)。

21.6 在文件服务器中使用 Freeze 逐出器

在这一节，我们将给出使用了 Freeze 逐出器的文件系统实现。这些实现的基础是 XREF 中所讨论的实现，而在这一节我们只讨论那些能说明 Freeze 逐出器的用法的代码。

一般而言，要把 Freeze 逐出器结合进你的应用中，需要采取以下步骤：

1. 考察你已有的 Slice 定义，确定合适的键和值类型。
2. 如果没有找到合适的类型，你通常要定义一个新的派生类，用以满足你的持久状态需求。考虑把这些定义放在单独的文件里：这些类型只被服务器用于进行持久，因此，无需出现在客户需要的 "public" 定义中。还可以考虑把你的持久类型放在单独的模块中，以避免名字冲突。
3. (使用 **slice2freeze** 或 **slice2freezej**) 为你的新定义生成代码。
4. 创建逐出器，并把它作为 servant 定位器向对象适配器注册。
5. 创建你的持久类型的实例，向逐出器注册它们。

21.6.1 Slice 定义

幸运的是，我们不必为了结合 Freeze 逐出器而改动任何已有的文件系统 Slice 定义。但是，为了表达我们的持久状态需求，我们需要增加一些定义：

```
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;
    };
};
```

你可以看到，我们为所有接点接口定义了子类。让我们来依次考察它们。

PersistentNode 类增加了两个数据成员：nodeName[4] 和 parent。这个文件系统实现要求子节点知道谁是它的父节点，以适当地实现 destroy 操作。先前的实现有一个类型为 DirectoryI 的状态成员，但那在这里不能工作。把父节点传给子节点的构造器已经不再可能，因为逐出器可能会 (通

过工厂）实例化子节点，这时不知道父节点是谁。即使我们知道父节点，另一个要考虑的因素是，我们无法保证父节点在子节点调用它时是活动的，因为逐出器可能已经把它逐出。我们通过存储指向父节点的代理，解决了这些问题。如果子节点通过代理调用父节点，逐出器会在必要时自动激活父节点。

PersistentFile 类非常直截了当，只是增加了一个 text 成员来表示文件的内容。注意，这个类扩展了 PersistentNode，因此继承了基类所声明的状态成员。

最后，PersistentDirectory 类定义了 removeNode 操作，并增加了 nodes 状态成员来表示目录节点的直接的子节点。因为子节点包含的只是它的 PersistentDirectory 父节点的代理，而不是指向实现类的引用，必须有一个用 Slice 定义的操作，能在子节点销毁时被调用。

如果我们遵循 21.5 节一开头的建议，我们本应该在单独的 Persistent-Filesystem 模块中定义 Node、File 和 Directory 类，但在这个例子中，为了简单起见，我们使用了已有的 Filesystem 模块。

21.6.2 用 C++ 实现文件系统服务器

服务器 main 程序

服务器的 main 程序负责创建逐出器，并初始化根目录节点。如 10.3.1 节所述，许多管理任务，比如创建和销毁通信器，都是由 Ice::Application 类来完成的。现在，我们的服务器 main 程序变成了这样：

```
#include <FilesystemI.h>
#include <Freeze/Freeze.h>

using namespace std;
using namespace Filesystem;

class FilesystemApp : virtual public Ice::Application {
public:
    FilesystemApp(const string & dbEnvName) :
        _dbEnvName(dbEnvName) { }

    virtual int run(int, char * []) {
```

4. 我们使用的是 nodeName，而不是 name，因为在 Node 接口中，name 已经被用作操作。

```
// Install object factories
//
Ice::ObjectFactoryPtr factory = new NodeFactory;
communicator()->addObjectFactory(
    factory,
    PersistentFile::ice_staticId());
communicator()->addObjectFactory(
    factory,
    PersistentDirectory::ice_staticId());

// Create an object adapter (stored in the NodeI::_adapter
// static member)
//
NodeI::_adapter =
    communicator()->createObjectAdapterWithEndpoints(
        "FreezeFilesystem", "default -p 10000");

// Create the Freeze evictor (stored in the
// NodeI::_evictor static member)
//
NodeI::_evictor =
    Freeze::createEvictor(communicator(), _dbEnvName,
                          "evictorfs");
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);
}
```

让我们详细考察一下所做的变动。首先，我们现在包括的是 Freeze/Freeze.h，而不是 Ice/Ice.h。这个 Freeze 头文件包括了这个源文件所需的其他全部 Freeze (以及 Ice) 头文件。

接下来，我们把 FilesystemApp 类定义成 Ice::Application 的子类，并提供了一个构造器，其参数是一个串:

```
    FilesystemApp(const string & dbEnvName) :
        _dbEnvName(dbEnvName) { }
```

串参数表示的是数据库环境名，会被保存下来，以备后面在 run 中使用。

run首先要执行的任务之一是为PersistentFile和PersistentDirectory安装 Ice 对象工厂。尽管这两个类不通过 Slice 操作进行交换，当它们被保存到数据库中、或从中加载时，它们的整编和解编是完全一样的，所以需要使用工厂。为这两种类型安装的工厂是 NodeFactory 的同一个实例。

```
        Ice::ObjectFactoryPtr factory = new NodeFactory;
        communicator()->addObjectFactory(
            factory,
            PersistentFile::ice_staticId());
        communicator()->addObjectFactory(
            factory,
            PersistentDirectory::ice_staticId());
```

在创建了对象适配器之后，程序调用 createEvictor 初始化一个 Freeze 逐出器。给 createEvictor 的第三个参数是数据库名，在缺省情况下，如果数据库不存在，就会创建一个。然后会安装一个 servant 初始化器，并把逐出器作为用于缺省范畴的 servant 定位器添加到对象适配器中。

```
    NodeI::_evictor =
        Freeze::createEvictor(communicator, _dbEnvName,
                              "evictorfs");
    Freeze::ServantInitializerPtr init = new NodeInitializer;
    NodeI::_evictor->installServantInitializer(init);
    NodeI::_adapter->addServantLocator(NodeI::_evictor, "");
```

接下来，如果在逐出器中还没有根目录节点，就创建它。

```
    Ice::Identity rootId = Ice::stringToIdentity("RootDir");
    if (!NodeI::_evictor->hasObject(rootId)) {
        PersistentDirectoryPtr root = new DirectoryI(rootId);
        root->nodeName = "/";
        NodeI::_evictor->createObject(rootId, root);
    }
```

最后， main 函数实例化 FilesystemApp，把 db 作为数据库环境名传给它。

```
int
main(int argc, char* argv[])
{
    FilesystemApp app("db");
    return app.main(argc, argv);
}
```

Servant 类定义

我们还必须改动 servant 类，把 Freeze 映射表结合进来。我们维持了 XREF 的多继承设计，但我们改动了构造器和状态成员。特别地，每个节点实现类都有两个构造器，一个没有参数，另一个的参数是一个 Ice::Identity。前者是工厂所需要的，而后者用于节点初次创建时。为了满足逐出器的要求， NodeI 从模板类 IceUtil::AbstractMutexI 派生，实现了 IceUtil::AbstractMutex。其他的唯一一处有意思的变动是 NodeI 中的一个新状态成员，名为 _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;
    };
}
```

除了节点实现类，我们还声明了一个对象工厂和一个 servant 初始化器的实现:

```
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 &);
    };
}
```

实现 NodeI

注意，NodeI 构造器不再计算标识的值。为了使我们的 servant 真正持久，这样做是必需的。特别地，某个节点的标识会在该节点创建时一次性计算出来，而且，在节点的生命期内，必须保持不变。因此，NodeI 构造器不能计算新标识，而是要记住传给它的标识。

```
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)
{
}
```

实现 FileI

FileI 的方法仍然相当简单，因为 Freeze 住处其会为我们处理持久问题。

```
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)
{
}
```

实现 DirectoryI

DirectoryI 实现需要进行更实质性的改变。我们将首先讨论 create-Directory 操作。

```
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;
}
```

在验证了节点名之后，这个操作获取子目录的唯一标识，实例化 servant，并向 Freeze 逐出器注册它。最后，这个操作为子目录创建代理，并把子目录添加到它的节点表中。

`createFile` 操作的实现的结构和 `createDirectory` 是一样的。

```
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;
}
```

实现 NodeFactory

我们把一种工厂实现用于创建两种类型的 Ice 对象：PersistentFile 和 PersistentDirectory。Freeze 逐出器只从其数据库中恢复这两种类型。

```
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()
{
}
```

实现 NodeInitializer

NodeInitializer 是 Freeze::ServantInitializer 接口的一种简单的实现，这个实现的唯一职责就是设置节点实现的 _ID 成员。逐出器会在创建了 servant、从数据库中恢复了其持久状态之后，但操作还没有分派给它之前，调用 initialize 操作。

```
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 用 Java 实现文件系统服务器

服务器 main 程序

服务器的 main 程序负责创建逐出器，并初始化根目录节点。如 12.3.1 节所述，许多管理任务，比如创建和销毁通信器，都是由 Ice.Application 类来完成的。现在，我们的服务器 main 程序变成了这样：

```
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;
}
```

让我们详细考察一下所做的变动。首先，我们现在把 Server 类定义成 Ice.Application 的子类，并提供了一个构造器，其参数是一个串：

```
public
Server(String dbEnvName)
{
    _dbEnvName = dbEnvName;
}
```

串参数表示的是数据库环境名，会被保存下来，以备后面在 run 中使用。

run首先要执行的任务之一是为PersistentFile和PersistentDirectory安装 Ice 对象工厂。尽管这两个类不通过 Slice 操作进行交换，当它们被保存到数据库中、或从中加载时，它们的整编和解编是完全一样的，所以需要使用工厂。为这两种类型安装的工厂是 NodeFactory 的同一个实例。

```
Ice.ObjectFactory factory = new NodeFactory();
communicator().addObjectFactory(
    factory,
    PersistentFile.ice_staticId());
communicator().addObjectFactory(
    factory,
    PersistentDirectory.ice_staticId());
```

在创建了对象适配器之后，程序调用 createEvictor 初始化一个 Freeze 逐出器。给 createEvictor 的第三个参数是数据库名， null 参数表示没有使用索引， true 表示，如果数据库不存在，就应该创建一个。然后会安装一个 servant 初始化器，并把逐出器作为用于缺省范畴的 servant 定位器添加到对象适配器中。

```
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, "");
```

接下来，如果在逐出器中还没有根目录节点，就创建它。

```
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);
        }
```

最后， main 函数实例化 Server 类，把 db 作为数据库环境名传给它。

```
    public static void
    main(String[] args)
    {
        Server app = new Server("db");
        app.main("Server", args);
        System.exit(0);
    }
```

Servant 类定义

我们还必须改动 servant 类，把 Freeze 映射表结合进来。我们维持了 XREF 的设计，但我们改动了构造器和状态成员。特别地，每个节点实现类都有两个构造器，一个没有参数，另一个的参数是一个 Ice::Identity。前者是工厂所需要的，而后者用于节点初次创建时。其他的唯一一处有意思的变动是新状态成员 _evictor。

FileI 类有一个新的静态成员，类型是 IdentityNodeMap，从而避免了在每个节点中重复这个饮用。这个类有一个新的实例成员，类型是 PersistentFile，用于表示这个文件的持久状态。最后，我们改动了构造器，接受参数 Ice.Identity 和 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;
}
```

DirectoryI 类也进行了类似的改动。

```
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;
}
```

注意，构造器不再计算标识的值。为了使我们的 servant 真正持久，这样做是必需的。特别地，某个节点的标识会在该节点创建时一次性计算出来，而且，在节点的生命期内，必须保持不变。因此， NodeI 构造器不能计算新标识，而是要记住传给它的标识。

实现 FileI

FileI 的方法仍然相当简单，因为 Freeze 住处其会为我们处理持久问题。

```
    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;
    }
```

实现 DirectoryI

DirectoryI 实现需要进行更实质性的改变。我们将首先讨论 create-Directory 操作。

```
    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;
    }
```

在验证了节点名之后，这个操作获取子目录的唯一标识，实例化 servant，并向 Freeze 逐出器注册它。最后，这个操作为子目录创建代理，并把子目录添加到它的节点表中。

`createFile` 操作的实现的结构和 `createDirectory` 是一样的。

```
    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;
    }
```

实现 NodeFactory

我们把一种工厂实现用于创建两种类型的 Ice 对象：PersistentFile 和 PersistentDirectory。 Freeze 逐出器只从其数据库中恢复这两种类型。

```
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()
    {
    }
}
```

实现 NodeInitializer

NodeInitializer 是 Freeze::ServantInitializer 接口的一种简单的实现，这个实现的唯一职责就是设置节点实现的 _ID 成员。逐出器会在创建了 servant、从数据库中恢复了其持久状态之后，但操作还没有分派给它之前，调用 initialize 操作。

```
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 总结

Freeze 是一组服务，可以简化 Ice 应用中的持久功能的使用。 Freeze 映射表是一个关联容器，可用于映射任何 Slice 键和值类型，为你提供了一种方便而熟悉的持久映射表接口。 Freeze 逐出器这种强大的设施尤其适用于在可高度伸缩的实现中支持持久的 Ice 对象。

第 22 章
FreezeScript

22.1 本章综述

本章将描述 FreezeScript，这是用于迁移和审查 (inspect) Freeze 映射表和逐出器所创建的数据库的工具。对数据库迁移的讨论从 22.3 节开始 , 一直延续到 22.5 节。 22.6 节和 22.7 节介绍数据库审查。最后， 22.8 节描述 FreezeScript 工具所支持的表达式语言。

22.2 引言

如第 21 章所述， Freeze 提供了一组很有价值的服务，可以简化 Ice 应用中持久功能的使用。但是，尽管 Freeze 使得应用能轻松地管理它的持久状态，还有另外一些管理上的任务需要得到处理:

- 迁移

  随着应用的演化，描述其持久状态的类型也随之演化，这并非很少见。在发生这样的变化时，如果已有的数据库能够迁移到新的格式，同时又尽可能多地保留信息，将能节省大量时间。

- 审查

  在应用的生命周期的每一个阶段，在一次次的开发之间，如果能对数据库进行检查，将会很有助益。

FreezeScript 提供了一些工具，能在 Freeze 映射表和逐出器数据库上进行这两种活动。这些数据库具有明确定义的结构，因为每条记录的键和值都是由它们各自的 Slice 类型的整编后的字节组成的。这种设计使得 FreezeScript 只使用数据库的类型的 Slice 定义，就能够在任何 Freeze 数据库上进行操作。

22.3 数据库迁移

FreezeScript 工具 **transformdb** 可以迁移 Freeze 映射表或逐出器所创建的数据库。它的迁移方式是：把 “老” 的 Slice 定义（也就是，描述数据库的当前内容的 Slice 定义）与 “新” 的 Slice 定义进行对比，并进行任何必需的修改，确保转换后的数据库与新定义是兼容的。

编写定制的转换程序来进行这样的迁移会很困难，因为那样的程序需要拥有新老类型的静态知识，这两种类型常常会定义许多相同的符号，因此会使程序无法加载。 **transformdb** 工具通过一种解释型的途径避免了上述问题：Slice 定义会被解析，用于驱动数据库记录的提取。

这个工具支持两种操作模式：

1. 自动迁移：只使用缺省的转换集，在一步之内迁移数据库。
2. 定制迁移：你提供一个脚本，用于增强或替代缺省的转换。

22.3.1 缺省转换

transformdb 进行的缺省转换会尽可能多地保留信息。但是，这个工具的能力受到了一些实际的限制，因为它所拥有的信息都是通过对 Slice 定义进行比较获得的。

例如，假定我们的某个结构的老定义是这样的：

```
struct AStruct {
    int i;
};
```

我们想要把上面这个结构的实例迁移到下面这个修改过的定义：

```
struct AStruct {
    int j;
};
```

作为开发者，我们知道这个 int 成员的名字从 i 变成了 j，但对于 **transformdb** 而言，好像移除了成员 i，添加了成员 j。缺省的转换正好会导致上述行为：i 的值丢失了，而 j 被初始化成一个缺省值。如果我们

需要保留 i 的值，并把它转给 j，那么我们就需要使用定制的迁移 (参见 22.3.5 节)。

在类型系统的演化过程中发生的变化可以划分成三种范畴:

- 数据成员

  类和结构类型的数据成员被添加、移除、更名。如上面所讨论的，缺省的转换会把新的和更名后的数据成员初始化为缺省的值 (参见 22.3.3 节)。

- 类型名

  类型被添加、移除、更名。在用于定义新的数据成员时，新类型不会给数据库迁移带来问题；成员会像平常一样用缺省值来初始化。另一方面，如果新类型取代了已有的数据成员的类型，那么类型兼容性就会变成一个要考虑的因素 (参见下面的条目)。

  被移除的类型通常也不会带来问题，因为任何对该类型的使用都应该已经从新的 Slice 定义中移除 (例如，通过把这种类型的数据成员移除掉)。但是，在有一种情况下，被移除的类型会变得成问题：在使用多态类时 (参见 22.5.5 节)。

  你需要注意被移除的类型，就像注意被移除的数据成员一样，因为可能会在迁移过程中丢失信息。在这种情况下，我们也建议进行定制的迁移。

- 类型内容

  变动类型内容的例子有：变动词典的键类型、序列的元素类型，或是数据成员的类型。如果老类型和新类型不兼容 (按照 22.3.2 节的定义)，那么缺省转换就会发出警告、丢弃当前值，并按照 22.3.3 节所描述的方式重新对值进行初始化。

22.3.2 类型兼容性

对值的类型所做的变动受限于一组特定的兼容变动。本节将描述缺省转换所支持的各种类型变动。所有不兼容的类型变动都会收到警告，说明当前值正在被丢弃，并由新类型的缺省值取而代之。如 22.3.5 节所述，定制迁移还提供了更多的灵活性。

布尔值

bool 类型的值可以转换成 string，或进行相反的转换。bool 值的合法串值是 "true" 和 "false"。

整数

整数类型 byte、 short、 int，以及 long 可以进行相互转换，但前提是当前值位于新类型的有效范围之内。这些整数类型也可以转换成 string。

浮点值

浮点 float 和 double 可以进行相互转换，也可以转换成 string。在转换过程中不会检测是否丧失精度。

串

string 值可以转换成其他任何原始类型，也可以转换成枚举和代理类型，但前提是这个值是新类型的合法串表示。例如，串值 "Pear" 可以转换成枚举 Fruit，但前提是 Pear 是 Fruit 的枚举符。

枚举

枚举可以转换成具有相同的类型 id 的枚举，也可以转换成串。枚举之间的转换是按照符号方式进行的。例如，考虑这样的老类型:

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

假定枚举符 Pear 要转换成这样的新类型:

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

尽管 Pear 在新类型中改变了位置，在新枚举中转换后的值仍是 Pear。但是，如果老值是 Orange，那么缺省转换就会发出警告，因为这个枚举符已经不再存在；新值会被初始化成 Apple (缺省值)。

如果有枚举符改了名字，要把枚举符从老的名字转换成新的名字，需要进行定制迁移。

序列

一种序列可以转换成另一种序列，即使新序列类型的类型 id 与老类型不同——但元素类型必须是兼容的。例如， sequence<short> 可以转换成 sequence<int>，不管这两种序列类型的名字是什么。

词典

一种词典可以转换成另一种词典，即使新词典类型的类型 id 与老类型不同——但键和值类型必须是兼容的。例如， dictionary<int, string> 可以转换成 dictionary<long, string>，不管这两种词典类型的名字是什么。

在改变词典的键类型时要小心，因为缺省的键转换可能会造成重复。例如，如果键类型从 int 变成 short，任何在 short 的有效范围以外的 int 值都会使得键被初始化成缺省值 (也就是零)。如果在词典中零已经被用作键，或是又遇到了超出有效范围的键，就会发生重复。缺省转换对键重复的处理办法是，从转换后的词典中移除重复的元素 (在这样的情况下，如果缺省行为不可接受，可以使用定制迁移)。

结构

一种 struct 类型只能被转换成另一种具有相同类型 id 的 struct 类型。数据成员会根据其类型进行适当的转换。

代理

代理值可以转换成另一种代理类型，或是转换成 string。前者是按照语言映射中的语义来完成的：如果新类型与老类型不匹配，那么新类型必须是老类型的基类型 (也就是说，代理宽化了)。

类

一种 class 类型只能被转换成另一种具有相同的类型 id 的 class 类型。一种 class 类型的数据成员可以宽化成一种基类型。数据成员会根据其类型进行适当的转换。更多关于类转换的信息，参见 22.5.5 节。

22.3.3 缺省值

如表 22.1 所示，各种数据类型会被初始化成缺省值。

表 22.1. Slice 类型的缺省值

类型	缺省值
布尔	false
数值	零 (0)
串	空串
枚举	第一个枚举符
序列	空序列
词典	空词典
结构	数据成员进行递归的初始化
代理	Nil

表 22.1. Slice 类型的缺省值

类型	缺省值
类	Nil

22.3.4 进行自动转换

要使用自动转换，我们需要给 **transformdb** 提供以下信息:

- 老的和新的 Slice 定义
- 数据库键和值的老类型和新类型
- 数据库环境目录、数据库文件名，以及新的、用以保存转换后数据库的数据库环境目录的名字

下面是 **transformdb** 命令的一个例子:

```
$ transformdb --old old/MyApp.ice --new new/MyApp.ice \
    --key int,string --value ::Employee db emp.db newdb
```

简要地说， **--old** 和 **--new** 选项分别指定老的和新的 Slice 定义。这两个选项可以根据需要多次指定，以加载所有相关的定义。 **--key** 选项指示，数据库键在从 `int` 演化到 `string`。 **--value** 选项指定把 `::Employee` 同时用作新老类型定义中的数据库值类型，因此只需指定一次。最后，我们提供了数据库环境目录 (**db**) 的路径名、数据库文件名 (**emp.db**)，以及存放转换后数据库的数据库环境目录的路径名 (**newdb**)。

transformdb 的更多使用信息，参见 22.5 节。

22.3.5 定制迁移

如果你的类型的变化方式使得自动迁移很困难、或不可能，定制迁移就会很有用。如果你的新类型或新数据成员的初始化很复杂，使用定制迁移也会很方便，因为定制迁移能让你完成许多任务，如果不使用定制迁移，你就需要编写用过即扔的程序来完成这些任务。

定制迁移与自动迁移协同工作，允许你在自动迁移过程中的一些明确定义的拦截点上注入你自己的转换规则。这些规则称为*转换描述符* (transformation descriptor)，是用 XML 编写的。

一个简单的例子

我们可以用一个简单的例子来演示定制迁移的使用。假定我们的应用使用了一个 Freeze 映射表，其键类型是 string，其值是一个枚举，定义如下：

```
enum BigThree { Ford, Chrysler, GeneralMotors };
```

现在，我们希望更改枚举符 Chrysler 的名字，如我们的新定义所示：

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

按照 22.3.2 节的解释，缺省转换会把所出现的所有 Chrysler 枚举符都转换成 Ford，因为在新定义中已不再有 Chrysler，故而会使用缺省值 Ford。

为了纠正这个问题，我们使用了下面的转换描述符：

```
<transformdb>
    <database key="string" value="::BigThree">
        <record>
            <if test="oldvalue == ::Old::Chrysler”>
                <set target="newvalue"
                     value="::New::DaimlerChrysler"/>
            </if>
        </record>
    </database>
</transformdb>
```

在执行时，这些描述符会把老类型系统中的 Chrysler 转换成新数据库的类型系统中的 DaimlerChrysler。 22.4 节将详细描述转换描述符。

22.4 转换描述符

本节描述 FreezeScript 转换描述符中的 XML 元素。

22.4.1 综述

转换描述符文件具有明确定义的结构。文件中的顶层描述符是 <transformdb>。在 <transformdb> 中必须出现一个 <database> 描述符，定义数据库所用的键和值类型。在 <database> 中， <record> 描述符会触发转换过程。 22.3.5 节给出了一个例子，演示了一个最小的描述符文件的结构。

在转换过程中， <transform> 和 <init> 描述符支持某些类型特有的动作，这两个元素都是 <transformdb> 的子元素。对于新 Slice 定义中的每种类型，可以定义一个 <transform> 描述符和一个 <init> 描述符。 **transformdb** 每次创建某种类型的新实例时，如果这种类型定义了 <init> 描述符，它就会执行该描述符。与此类似， **transformdb** 每次把老类型的实例转换成新类型时，时都会执行这种新类型的 <transform> 描述符。

<database>、 <record>、 <transform>，以及 <init> 描述符可以包含通用的动作描述符，比如 <if>、 <set>，以及 <echo>。这些动作与 C++ 和 Java 这样的语言中的语句在这样一些方面是类似的：它们按照定义的次序执行，它们的效果会累积。这些动作使用了 22.8 节描述的表达式语言。

22.4.2 执行流

转换描述符按照下述方式执行:

- <database> 最先执行。 <database> 的每个子描述符都按照定义的次序执行。如果有 <record> 描述符，在该点就会进行数据库转换。在转换完成之前， <database> 的任何位于 <record> 之后的子描述符都不会执行。
- 在转换每条记录的过程中， **transformdb** 会创建新的键和值类型的实例，包括执行这些类型的 <init> 描述符。接下来，老的键和值会按照下面的方式转换成新的键和值:

  1. 定位该类型的 <transform> 描述符。
  2. 如果没有找到描述符，或者描述符存在、但没有把缺省转换排除在外，就按 22.3.1 节所述，对数据进行转换。
  3. 如果 <transform> 描述符存在，就执行它。
  4. 最后，执行 <record> 的各个子描述符。

  关于转换描述符的详细信息，参见 22.4.4 节。

22.4.3 作用域

<database> 描述符创建了一个全局作用域，允许它的子描述符去定义能在任何描述符中访问的符号[1]。此外，另外的某些描述符会创建局部作用域，只在描述符执行期间存在。例如， <transform> 会创建局部作用域，定义符号 old 和 new 来表示老形式和新形式的值。 <transform> 的子描述符还可以在局部作用域中定义新的符号，只要这些符号与该作用域

中的已有符号没有冲突。添加与外层作用域中的某个符号同名的符号是合法的，但在该描述符执行期间，将无法访问该外层符号。

全局作用域在许多情况下都很有用。例如，假定你想要追踪在转换过程中，遇到某个值的次数是多少，你可以使用这样的描述符：

```
<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>
```

在这个例子中，`<define>` 描述符在全局作用域中引入了符号 `categoryCount`，把它的类型定义为 `int`，初始值是零。接下来，`<record>` 描述符致使转换开始进行。`Ice::Identity` 类型的每次出现都会致使它的 `<transform>` 描述符被执行，这个描述符会检查 `category` 成员，如果有必要就使 `categoryCount` 加一。最后，在转换完成后，`<echo>` 描述符会显示 `categoryCount` 的最终值。

让我们再强调一下描述符和作用域之间的关系。考虑图 22.1 中的图，其中给出了若干描述符，包括它们在其局部作用域中定义的符号。在这个例子中，`<iterate>` 描述符有一个词典目标，因此元素值 `value` 的缺省符号使得父 `<init>` 描述符的作用域中的同名符号隐藏了起来[2]。除了

1. 要让某个全局符号能在 `<transform>` 或 `<init>` 描述符中访问，这个符号必须在 `<record>` 描述符执行之前定义。
2. 通过把另外的符号名赋给这个元素值，可以避免这种情况。

<iterate> 作用域中的符号，<iterate> 的子描述符还可以引用 <init> 和 <database> 作用域中的符号。

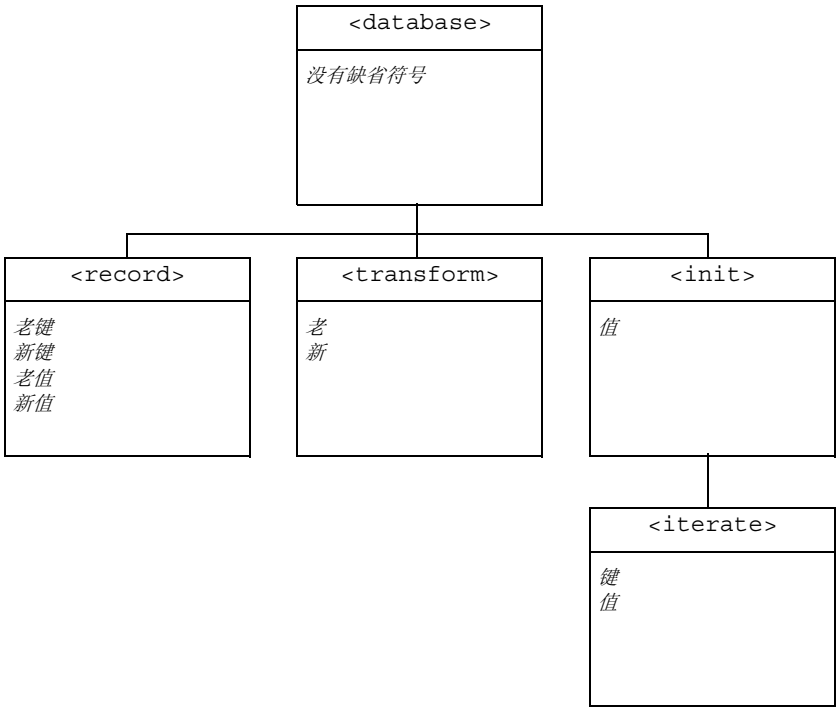

图 22.1. 描述符和作用域之间的关系

22.4.4 描述符参考资料

<transformdb>

描述符文件中的顶层描述符。它需要有一个子描述符 <database>，并且支持任意数目的 <transform> 和 <init> 描述符。这个描述符没有属性。

<database>

这个描述符的各个属性定义要转换的数据库的老的和新的键和值类型。它支持任意数目的子描述符，但最多只能有一个 <record> 描述符。

<database> 描述符还为用户定义的符号创建了一个全局作用域 (参见 22.4.3 节)。

表 22.2 描述了 <database> 描述符支持的属性。

表 22.2. <database> 描述符的属性

名字	描述
key	指定老键和新键的 Slice 类型。如果两者的类型是一样的，只需指定一个就可以了。否则，用逗号把类型分隔开。
value	指定老值和新值的 Slice 类型。如果两者的类型是一样的，只需指定一个就可以了。否则，用逗号把类型分隔开。

例如，考虑下面的 <database> 描述符。在这个例子中，要进行转换的 Freeze 映射表目前的键类型是 int，值类型是 ::Employee，迁移后的键类型是 string:

```
<database key="int,string" value="::Employee">
```

<record>

开始转换。子描述符会针对数据库中的每条记录执行，并给用户提供机会，检查记录的老的键和值，也可以修改新的键和值。缺省转换，以及 <transform> 和 <init> 描述符，会在子描述符执行之前执行。<record> 描述符会把下列符号引入局部作用域：oldkey、 newkey、oldvalue、 newvalue。子描述符可以访问这些符号，但 <transform> 或 <init> 描述符不可以。 oldkey 和 oldvalue 符号是只读的。

在修改数据库键时，要注意确保不会产生重复的键。如果遇到了重复的数据库键，转换会立刻失败。

注意，只有在有 <record> 的情况下，才会进行数据库转换。

<transform>

为新 Slice 定义中的某种类型的所有实例定制转换。这个描述符的子描述符会在可选的缺省转换 （ 参见 22.3.1 节 ）完成之后执行。对一种类型，只能指定一个 <transform> 描述符，但并非所有类型都需要有

<transform> 描述符。符号 old 和 new 会被引入局部作用域，分别表示老值和新值。 old 符号是只读的。表 22.3 描述了这个描述符支持的属性。

表 22.3. <transform> 描述符的属性

名字	描述
type	指定 Slice 类型 id。
default	如果为 false，不对这种类型的值进行缺省转换。如果没有指定，缺省值是 true。
base	这个属性决定是否要执行基类类型的 <transform> 描述符。如果为 true，直接的基类 (immediate base class) 的 <transform> 描述符会被调用。如果没有找到直接基类的描述符，就搜索类层次，直到找到一个描述符为止。任何基类描述符都会在这个描述符的子描述符执行之后才执行。如果没有指定，缺省值是 true。
rename	说明老 Slice 定义中的某个类型已经更名为由 type 属性标识的新类型。这个属性的值是老 Slice 定义的类型 id。指定了这个属性之后，在 22.3.2 节针对 enum、 struct 及 class 类型定义的兼容性规则会放宽。

下面是 <transform> 描述符的一个例子，它对一个新数据成员进行初始化:

```
<transform type="::Product">
    <set target="new.salePrice"
         value="old.listPrice * old.discount"/>
</transform>
```

对于类类型， **transformdb** 会首先尝试定位对象的派生层次最深的类型的 <transform> 描述符。如果没有找到描述符， **transformdb** 就会继续沿着类层次向上搜索描述符。基对象类型 Object 是所有类层次的根，也包含在描述符搜索范围内。因此，为 Object 类型定义一个 <transform> 描述符也是可能的，在转换任何一个类实例时都会调用这个描述符。

注意， <transform> 描述符会递归执行。例如，考虑下面的 Slice 定义:

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

当 **transformdb** 在 Outer 类型的值上进行缺省转换时，它会在 Inner 成员上递归地进行转换，然后执行 Inner 的 <transform> 描述符，最后执行 Outer 的 <transform> 描述符。但是，如果 Outer 禁用缺省转换，那么在 Inner 成员上也不会进行转换，因此 Inner 的 <transform> 描述符不会被执行。

<init>

为某种类型的所有实例定义定制的初始化规则。子描述符会在该类型每次实例化时执行。这个描述符的典型用例：用于已经引入新 Slice 定义的类型，它的实例所需要的缺省值与 **transformdb** 所提供的不同。符号 value 被引入局部作用域中，用于表示实例。表 22.4 描述了这个描述符支持的属性。

表 22.4. <init> 描述符的属性

名字	描述
type	指定 Slice 类型 id

下面是 <init> 描述符的一个简单例子：

```
<init type="::Player">
    <set target="value.currency" value="100"/>
</init>
```

注意，和 <transform> 一样，<init> 描述符的执行也是递归的。例如，如果为一个 struct 类型定义了一个 <init> 描述符，这个 struct 的成员的 <init> 描述符会在 struct 的描述符执行之前执行。

<iterate>

遍历词典或序列，执行每个元素的子描述符。用来表示元素信息的符号名可能会和所属作用域中的已有符号冲突，在这种情况下，子描述符无法访问那些外层符号。表 22.5 描述了这个描述符支持的属性。

表 22.5. <iterate> 描述符的属性

名字	描述符
target	序列或词典。
index	用于序列索引的符号名。如果没有指定，缺省符号是 i。
element	用于序列元素的符号名。如果没有指定，缺省符号是 elem。
key	用于词典键的符号名。如果没有指定，缺省符号是 key。
value	用于词典值的符号名。如果没有指定，缺省符号是 value。

下面是 <iterate> 描述符的一个例子。如果雇员的薪水大于 $3000，就把新数据成员 reviewSalary 设成 true。

```
<iterate target="new.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <set target="emp.reviewSalary" value="true"/>
    </if>
</iterate>
```

<if>

有条件地执行子描述符。表 22.6 描述了这个描述符支持的属性。

表 22.6. <if> 描述符的属性。

名字	描述
test	一个布尔表达式

更多关于描述符表达式语言的信息，参见 22.8 节。

<set>

修改一个值。 value 和 type 属性是互斥的。如果 target 表示的是词典元素，这个元素必须已经存在 (也就是说， <set> 不能用来把元素添加到词典中)。表 22.7 描述了这个描述符支持的属性。

表 22.7. <set> 描述符的属性

名字	描述
target	一个必须选中可修改的值的表达式。
value	一个表达式，对其求值所得的值必须与目标的类型兼容。
type	要实例化的类的类型 id。这个类必须与目标的类型兼容。
length	一个整型表达式，表示所需的序列新长度。如果新长度小于序列的当前尺寸，会从序列的末尾移除元素。如果新长度大于当前尺寸，会添加新元素到序列的末尾。如果同时还指定了 value 或 type，就会用来初始化每个新元素。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，转换会立刻失败。如果没有指定，缺省值是 false。

下面的 <set> 描述符修改词典元素的成员:

```
<set target="new.parts[‘P105J3’].cost"
      value="new.parts[‘P105J3’].cost * 1.05"/>
```

这个 <set> 描述符把一个元素添加到一个序列中，并初始化它的值:

```
<set target="new.partsList"
length="new.partsList.length + 1"
      value="‘P105J3’"/>
```

<add>

把一个新元素添加到序列或词典中。在使用 <iterate> 遍历序列或词典的过程中添加元素是合法的，但添加之后的遍历次序是不确定的。与 value 和 type 属性一样， key 和 index 属性是互斥的。如果 value 和

type 都没有指定，新元素就会用缺省值初始化。表 22.8 描述了这个描述符支持的属性。

表 22.8. <add> 描述符的属性

名字	描述
target	一个必须选中可修改序列或词典的表达式。
key	一个表达式，对其求值所得的值必须与目标词典的键类型兼容。
index	一个表达式，对其求值所得的必须是整数值，表示插入的位置。新元素被插入到 index 之前。这个值不能超过目标序列的长度。
value	一个表达式，对其求值所得的值必须与目标词典的值类型、或目标序列的元素类型兼容。
type	要实例化的类的类型 id。这个类必须与目标词典的值类型、或目标序列的元素类型兼容。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，转换会立刻失败。如果没有指定，缺省值是 false。

下面是 <add> 描述符的一个例子，它添加一个新词典元素，然后初始化其成员：

```
<add target="new.parts" key="‘P105J4’"/>
<set target="new.parts[‘P105J4’].cost" value="3.15"/>
```

<define>

在当前作用域中定义一个新符号。表 22.9 描述了这个描述符支持的属性。

表 22.9. <define> 描述符的属性

名字	描述
name	新符号的名字。如果这个名字与当前作用域中的某个已有符号同名，就会发生错误。

表 22.9. <define> 描述符的属性

名字	描述
type	符号的形式 Slice 类型的名字。对于用户定义的类型，名字应该加上前缀 ::Old 或 ::New，表明该类型的来源。原始类型的前缀可以忽略。
value	一个表达式，对其求值所得的值应该与符号的类型兼容。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，执行会立刻失败。如果没有指定，缺省值是 false。

下面是 <define> 描述符的两个例子。第一个例子把符号 identity 的类型定义为 Ice::Identity，并用 <set> 初始化其成员：

```
<define name="identity" type="::New::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

第二个例子用我们最初在 22.3.5 节看到的枚举来定义符号 manufacturer，并把缺省值赋给它：

```
<define name="manufacturer" type="::New::BigThree"
        value="::New::DaimlerChrysler"/>
```

<remove>

从序列或词典中移除一个元素。在用 <iterate> 遍历序列或词典的过程中移除元素是合法的，但移除之后的遍历次序是不确定的。表 22.10 描述了这个描述符支持的属性。

表 22.10. <remove> 描述符的属性

名字	描述
target	一个必须选中可修改序列或词典的表达式。
key	一个表达式，对其求值所得的值必须与目标词典的键类型兼容。
index	一个表达式，对其求值所得的必须是一个整数值，表示要移除的序列元素的索引。

<fail>

致使转换立刻失败。如果指定了 test，只有当表达式的值为 true 时，转换才会失败。表 22.11 描述了这个描述符支持的属性。

表 22.11. <fail> 描述符的属性

名字	描述
message	一条在转换失败时显示的消息。
test	一个布尔表达式。

如果检测到有效范围错误，下面的 <fail> 描述符就终止转换:

```
<fail message="range error occurred in ticket count!"
      test="old.ticketCount > 32767"/>
```

<delete>

停止当前数据库记录的转换，从转换后的数据库中移除该记录。这个描述符没有属性。

<echo>

显示值和信息。如果没有指定属性，就只打印一个换行符。表 22.12 描述了这个描述符支持的属性。

表 22.12. <echo> 描述符的属性

名字	描述
message	一条要显示的消息。
value	一个表达式。该表达式的值会以结构化的格式显示。

下面是一个 <echo> 描述符，它使用了 message 和 value 属性:

```
<if test="old.ticketCount > 32767">
    <echo message="deleting record with invalid ticket count: "
          value="old.ticketCount"/>
    <delete/>
</if>
```

22.4.5 描述符指导方针

在转换过程中，你可以在三个点上进行拦截：在转换记录时 (<record>)、在转换某种类型的实例时 (<transform>)、在创建某种类型的实例时 (<init>)。

一般而言，<record> 用在你的修改既需要访问记录的键，也需要访问记录的值时。例如，如果数据库键需要用作某个方程式的因子，或是标识词典中的某个记录，那么 <record> 就是唯一一种能让你进行这种修改的描述符。如果所做变动的数量很少，同时不值得编写单独的 <transform> 或 <init> 描述符，<record> 描述符使用起来也很方便。

<transform> 描述符的作用域比 <record> 更有限。它被用在这样的情况下：有可能必须对某种类型的所有实例做出改变 (不管该类型被用在什么样的上下文中)，并且需要访问老值。<transform> 描述符不能访问数据库键和值，因此其决策只能基于所处理的类型的老实例和新实例来做出。

最后，如果为了适当地初始化某种类型、并不需要访问老的实例，可以使用 <init> 描述符。在大多数情况下，这种活动可以由 <transform> 描述符来完成，它只需忽略老的实例就可以了，所以 <init> 似乎是多余的。但是，在下面这种情况下，需要使用 <init>：必须对新 Slice 定义引入的某种类型的实例进行初始化。因为在当前数据库中没有这种类型的实例，所以 <transform> 描述符永远不会针对这种类型执行。

22.5 使用 transformdb

本节将描述怎样调用 **transformdb** 工具，并就怎样最好地利用它给出了建议。除了 include 目录 (**-I**) 选项，4.18 节列出的所有 Slice 处理器共同支持的命令行选项，这个工具都支持。**transformdb** 专有的选项将在下面的小节中描述。

22.5.1 一般选项

下面的选项既可用于自动迁移，也可用于定制迁移：

- **--old *SLICE***
  --new *SLICE*

  加载包含在 ***SLICE*** 文件中的老的或新的 Slice 定义。如果必须加载多个文件，这两个选项可以多次指定。但是，用户要负责确保没有重复的定义 (如果两个被加载的文件共享了同一个 include 文件，就可能发

生这样的事情)。要避免重复定义，一种策略是加载一个 Slice 文件，其中只含有用于把每个要加载的 Slice 文件包括进来的 `#include` 语句。在这种情况下，如果被包括的文件正确地使用了 include 守卫，就不可能发生重复。

- **`--include-old` *`DIR`***
  `--include-new` *`DIR`*

  把 ***`DIR`*** 目录添加到老的或新的 Slice 定义的 include 路径集中。

- **`--key` *`TYPE[,TYPE]`***
  `--value` *`TYPE[,TYPE]`*

  指定数据库键和值的 Slice 类型。如果类型没有改变，就只需指定一次。否则，就首先指定老类型，然后是新类型，中间用逗号分隔。例如，选项 **`--key int,string`** 指示，数据库键要从 `int` 迁移到 `string`。而选项 **`--key int,int`** 指示，键类型没有改变；也可以只用 **`--key int`** 来指定。类型的改变受限于 22.3.2 节定义的兼容性规则所允许的方式，但定制迁移提供了更多的灵活性。

- **`-e`**

  说明正在迁移的是 Freeze 逐出器数据库。这个选项是为了方便起见而提供的：它将按照 Freeze 逐出器的情况，自动设置数据库键和值类型，因而不再需要使用 **`--key`** 和 **`--value`** 选项。特别地，Freeze 逐出器数据库的键类型 是 `Freeze::EvictorStorageKey`，值类型是 `Freeze::ObjectRecord`。这两个类型是在 Slice 文件 `Freeze/EvictorStorage.ice` 中定义的；但是，你无需把这个文件加载进你的老的和新的 Slice 定义中。

- **`-i`**

  要求 **`transformdb`** 忽略那些违反 22.3.2 节定义的兼容性规则的类型改变。如果没有指定这个选项，在发生这样的违反的情况下，转换会立刻失败。指定了这个选项，会显示警告，但转换将继续。

- **`-p`**

  要求 **`transformdb`** 清除那些其类型已不能在新 Slice 定义中找到的对象实例。更多信息，参见 22.5.5 节。

- **`-c`**

  对老的 BerkeleyDB 数据库环境进行灾难恢复。

- **`-w`**

  抑制转换过程中的重复警告。

22.5.2 数据库参数

如果你调用 **transformdb** 来转换数据库，它需要三个参数:

- **dbenv**

  数据库环境目录的路径名。

- **db**

  现有数据库文件的名字。 **transformdb** 决不会修改这个数据库。

- **newdbenv**

  用于容纳转换后数据库的数据库环境目录的路径名。这个目录必须存在，而且不能已经含有名字与 **db** 参数相同的数据库。

  如果转换成功，在新环境目录中会创建一个同名的数据库。

22.5.3 自动迁移

在使用自动迁移时不需要指定额外的参数。你所需要的全部选项就是 4.18 节的标准选项、 22.5.1 节描述的一般选项，以及 22.5.2 节的数据库参数。例如，考虑下面的命令，它使用了自动迁移，把一个键类型为 int、值类型为 string 的数据库转换成键类型不变、值类型为 long 的数据库:

```
$ transformdb --key int --value string,long dbhome data.db newdbhome
```

注意，我们不需要指定 **--old** 或 **--new** 选项，因为我们的键和值类型是原始类型。如果转换成功，在 **newdbhome/data.db** 文件中有我们转换后的数据库。

22.5.4 定制迁移

分析

定制迁移是一个有两个步骤的过程: 你先编写转换描述符，然后执行它们，对数据库进行转换。为了帮助你创建描述符文件， **transformdb** 可以对你的新老 Slice 定义进行比较，生成一组缺省的转换描述符。这个特性称为*分析* (analysis)，可以通过指定下面的选项来调用:

- **-o *FILE***

  指定在分析过程中要创建的描述符文件 ***FILE***。

这时不需要使用数据库选项，因为如果指定了 **-o** 选项，就不会进行转换。在生成的文件中，对于新老 Slice 定义中都有的每一种类型，都会生成一个 <transform> 描述符; 对于只出现在新 Slice 定义中的每一种类型，

都会 生成一个 <init> 描述符。在大多数情况下，这些描述符是空的。但它们也可能含有一些 XML 注释，描述 **transformdb** 所检测到的一些需要你做出的变动。

例如，让我们再看一看我们在 22.3.5 节定义的枚举:

```
enum BigThree { Ford, Chrysler, GeneralMotors };
```

这个枚举已经演化成了下面的枚举。特别地，为了反映公司的合并，Chrysler 枚举符更改了名字:

```
enum BigThree { Ford, DaimlerChrysler, GeneralMotors };
```

接下来，我们在分析模式中运行 **transformdb**:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
    --key string --value ::BigThree -o transform.xml
```

生成的 **transform.xml** 文件含有以下用于 BigThree 枚举的描述符:

```
<transform type="::BigThree">
    <!-- NOTICE: enumerator `Chrysler' has been removed -->
</transform>
```

注释表明，在新定义中已不再有枚举符 Chrysler，提醒我们，要在这个 <transform> 描述符中增加逻辑，把所有出现 Chrysler 的地方改成 DaimlerChrysler。

transformdb 生成的描述符文件是 "well-formed"，在执行之前，不需要任何的手工干预。但是，如果不进行修改就执行这样的描述符文件，就等同于自动迁移。

转换

在准备好描述符文件（或者完全由你自己编写，或者修改分析阶段所生成的文件）之后，你就可以去转换数据库了。还有一个额外的用于转换的选项:

- **-f *FILE***

  指定在转换过程中要执行的描述符文件 ***FILE***。

在转换过程中不必提供 **-e**、 **--key** 或 **--value** 选项，因为在 <database> 描述符中已经指定了键和值类型。

继续我们在上面的分析阶段使用的枚举例子，假定我们有一个修改过的 **transform.xml**，用于转换 Chrysler 枚举符，现在已可以执行转换:

```
$ transformdb --old old/BigThree.ice --new new/BigThree.ice \
    -f transform.xml dbhome bigthree.db newbigthree.db
```

策略

如果你必须转换某个 Freeze 数据库，一般而言，我们建议你首先尝试使用自动转换，除非你知道必须使用定制转换。因为转换是一个非毁坏性的过程，尝试进行自动转换并没有什么危害，这也是对你的 **transformdb** 参数、以及数据库自身进行健全性检查的好方法 (例如，确保所有必需的 Slice 文件被加载)。如果 **transformdb** 检测到任何不兼容的类型改变，它会为每一处不兼容的改变显示一条出错消息，然后不进行任何转换就终止。在这种情况下，你可以再次运行 **transformdb**，指定 **-i** 选项，忽略所有不兼容的改变，让转换得以进行。

你应该注意 **transformdb** 发出的任何警告，因为它们可能表明需要使用定制迁移。例如，如果我们试图使用自动迁移来转换含有 BigThree 枚举的数据库，只要出现 Chrysler 枚举符，就会显示下面的警告：

```
warning: unable to convert 'Chrysler' to ::BigThree
```

如果需要使用定制迁移，你可以用分析来生成缺省的描述符文件，然后查看其中的 NOTICE 注意，根据需要进行编辑。在测试你的描述符文件时，使用 <echo> 描述符可能会有所助益，特别是在 <record> 描述符中，你可以在那里显示老的和新的键和值。

22.5.5 转换对象

在进行数据库迁移时， Slice 类的的多态本质可能会带来问题。例如，Slice 解析器可以确保加载进 **transformdb** 的一组 Slice 定义的所有类型都是完整的，除了类 (还有异常，但我们可以忽略它们，因为它们不能持久)。 **transformdb** 无法知道这样的情况：数据库可能含有某个子类的实例，这个子类派生自某个已加载的类，但这个子类却未加载。某个类实例的类型也可能更改了名字，无法在新的 Slice 定义中找到。

在缺省情况下，这样的情况会立刻导致转换失败。但是， **-p** 选项是一种 (可能很激烈的) 处理这样的情况的办法：如果某个类实例在新 Slice 定义中没有等价物，而你指定了这个选项， **transformdb** 就会尽一切可能移除该实例。如果这个实例出现在序列或词典元素中，该元素就会被移除。否则，含有该实例的数据库记录就会被删除。

现在，使用定制迁移和 <transform> 描述符的 rename 属性，可以足够轻松地处理更改类类型的名称的问题。但是，在有些合理的情况下，也可以利用 **-p** 选项的毁坏性。例如，如果某个类类型已被移除，而且从一个保证不含任何该类型的实例的数据库开始着手工作，会更容易，那么 **-p** 选项也许可以简化更广泛的迁移工作。

这又是这样一种情况的一个例子：首先进行自动迁移，有助于指出潜在的迁移中的问题点。使用 **-p** 选项， **transformdb** 会针对缺失的类类型

发出警告并继续执行，而不会在第一次遇到这样的情况时中止，这样你就可以发现你是否忘记了加载某些 Slice 定义，或是否需要对某个类型进行更名。

22.6 数据库审查

FreezeScript 工具 **dumpdb** 被用于检查 Freeze 数据库。以最简单的方式调用它，会显示出数据库的每一条记录；但这个工具还支持一些更有选择性的活动。事实上， **dumpdb** 支持一种脚本模式，可以和 **transformdb** (参见 22.4 节) 共享许多相同的 XML 描述符，从而让你进行复杂的过滤和报告。

22.6.1 描述符综述

dumpdb 描述符文件具有明确定义的结构。文件中的顶层描述符是 <dumpdb>。在 <dumpdb> 中，必须有一个 <database> 描述符，定义数据库所用的键和值类型。在 <database> 中， <record> 描述符会触发数据库遍历。下面给出的例子说明了一个最小的描述符文件的结构:

```
<dumpdb>
    <database key="string" value="::Employee">
        <record>
            <echo message="Key: " value="key"/>
            <echo message="Value: " value="value"/>
        </record>
    </database>
</dumpdb>
```

在遍历过程中， <dump> 支持某些类型特有的动作； <dump> 是 <dumpdb> 的孩子。对于新 Slice 定义中的每种类型，可以定义一个 <dump> 描述符。每当 **dumpdb** 遇到某种类型的一个实例时，就会执行该类型的 <dump> 描述符。

<database>、 <record>，以及 <dump> 描述符可以包含一些通用的动作描述符，比如 <if> 和 echo>。这些动作与 C++ 和 Java 这样的语言中的语句在这样一些方面是类似的：它们按照定义的次序执行，它们的效果会累积。这些动作使用了 22.8 节描述的表达式语言。

尽管 **dumpdb** 描述符不允许修改数据库，它们仍然可以为了定义脚本而定义局部符号。一旦 <define> 描述符定义了一个符号，其他描述符，比如 <set>、 <add>，以及 <remove>，都可被用来操纵符号的值。

22.6.2 执行流

描述符按照下述方式执行:

- <database> 最先执行。 <database> 的每个子描述符都按照定义的次序执行。如果有 <record> 描述符，在该点就会进行数据库遍历。在遍历完成之前， <database> 的任何位于 <record> 之后的子描述符都不会执行。
- 对于每条记录， **dumpdb** 都会解释其键和值，调用它所遇到的类型的 <dump> 描述符。例如，如果数据库的值类型是一个 struct，那么 **dumpdb** 会首先尝试调用结构类型的 <dump> 描述符，然后以同样的方式递归地解释结构的各个成员。

  关于 **dumpdb** 描述符的详细信息，参见 22.6.4 节。

22.6.3 作用域

<database> 描述符创建了一个全局作用域，允许它的子描述符去定义能在任何描述符中访问的符号[3]。此外，某些其他的描述符会创建局部作用域，只在描述符执行期间存在。例如， <dump> 会创建局部作用域，定义符号 value 来表示指定类型的值。 <dump> 的子描述符还可以在局部作用域中定义新的符号，只要这些符号与该作用域中的已有符号没有冲突。添加与外层作用域中的某个符号同名的符号是合法的，但在该描述符执行期间，将无法访问该外层符号。

全局作用域在许多情况下都很有用。例如，假定你想要追踪在数据库遍历过程中，遇到某个值的次数是多少，你可以使用这样的描述符:

```
<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. 要让某个全局符号能在 <dump> 描述符中访问，这个符号必须在 <record> 描述符执行之前定义。

在这个例子中，<define>描述符在全局作用域中引入了符号 categoryCount，把它的类型定义为 int，初始值是零。接下来，<record>描述符致使数据库遍历开始进行。Ice::Identity 类型的每次出现都会致使它的<dump>描述符被执行，这个描述符会检查 category 成员，如果有必要就使 categoryCount 加一。最后，在遍历完成后，<echo>描述符会显示 categoryCount 的最终值。

让我们再强调一下描述符和作用域之间的关系。考虑图 22.2 中的图，其中给出了若干描述符，包括它们在其局部作用域中定义的符号。在这个例子中，<iterate>描述符有一个词典目标，因此元素值 value 的缺省符号使得父<dump>描述符的作用域中的同名符号隐藏了起来[4]。除了<iterate>作用域中的符号，<iterate>的子描述符还可以引用<dump>和<database>作用域中的符号。

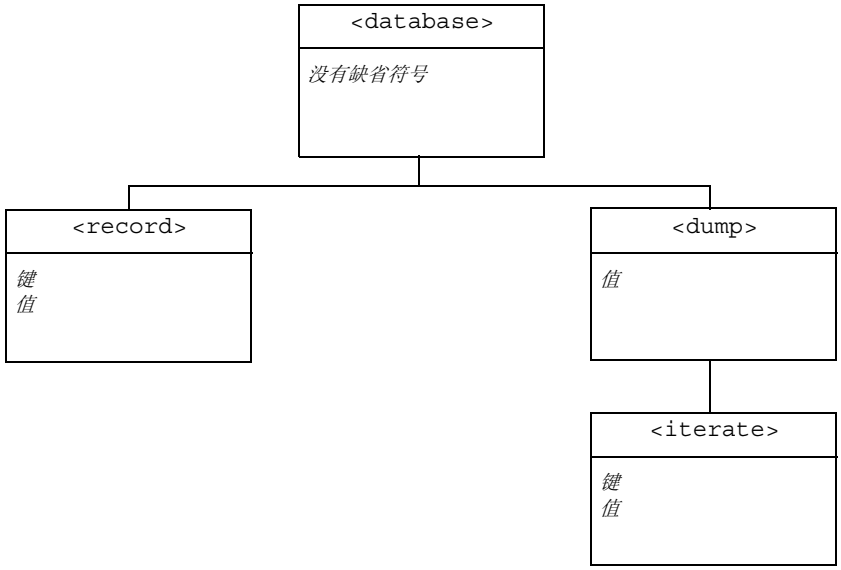

图 22.2. 描述符和作用域之间的关系

4. 通过把另外的符号名赋给这个元素值，可以避免这种情况。

22.6.4 描述符参考资料

<dumpdb>

描述符文件中的顶层描述符。它需要有一个子描述符 <database>，并且支持任意数目的 <dump> 描述符。这个描述符没有属性。

<database>

这个描述符的各个属性定义数据库的键和值类型。它支持任意数目的子描述符，但最多只能有一个 <record> 描述符。 <database> 描述符还为用户定义的符号创建了一个全局作用域 (参见 22.6.3 节)。

表 22.13 描述了 <database> 描述符支持的属性。 .

表 22.13. <database> 描述符的属性

名字	描述
key	指定数据库键的 Slice 类型。
value	指定数据库值的 Slice 类型。

例如，考虑下面的 <database> 描述符。在这个例子中，要进行检查的 Freeze 映射表的键类型是 int，值类型是 ::Employee:

```
<database key="int" value="::Employee">
```

<record>

开始遍历数据库。子描述符会针对数据库中的每条记录执行，但要在 <dump> 描述符执行之后才执行。 <record> 描述符会把只读符号 key 和 value 引入局部作用域。子描述符可以访问这些符号，但 <dump> 描述符不可以。

注意，只有在有 <record> 的情况下，才会进行数据库遍历。

<dump>

针对某种 Slice 类型的所有实例执行。对一种类型，只能指定一个 <dump> 描述符，但并非所有类型都需要有 <dump> 描述符。只读符号 value 会被引入局部作用域中。表 22.14 描述了这个描述符支持的属性。

表 22.14. <dump> 描述符的属性

名字	描述
type	指定 Slice 类型 id。
base	如果 type 指定的是 Slice 类，这个属性决定是否要调用基类类型的 <dump> 描述符。如果为 true，基类描述符会在子描述符执行之后被调用。如果没有指定，缺省值是 true。
contents	对于 class 和 struct 类型，这个属性决定是否要执行值的成员的描述符。对于 sequence 和 dictionary 类型，这个属性决定是否要执行元素的描述符。如果没有指定，缺省值是 true。

下面是 <dump> 描述符的一个例子，它的要搜索特定的产品:

```
<dump type="::Product">
    <if test="value.description.find('scanner') != -1">
        <echo message="Scanner SKU: " value="value.SKU"/>
    </if>
</dump>
```

对于类类型， **dumpdb** 会首先尝试定位对象的派生层次最深的类型的 <dump> 描述符。如果没有找到描述符， **dumpdb** 就会继续沿着类层次向上搜索描述符。基对象类型 Object 是所有类层次的根，也包含在描述符搜索范围内。因此，为 Object 类型定义一个 <dump> 描述符也是可能的，在处理每一个类实例时都会调用这个描述符。

注意， <dump> 描述符会递归执行。例如，考虑下面的 Slice 定义:

```
struct Inner {
    int sum;
};
struct Outer {
    Inner i;
};
```

当 **dumpdb** 在解释 Outer 类型的值时，它会执行 Outer 的 <dump> 描述符，然后递归地执行 Inner 成员的 <dump> 描述符，但只有在 Outer 描述符的 contents 属性的值为 true 时才这么做。

<iterate>

遍历词典或序列，执行每个元素的子描述符。用来表示元素信息的符号名可能会和所属作用域中的已有符号冲突，在这种情况下，子描述符无法访问那些外层符号。表 22.15 描述了这个描述符支持的属性。

表 22.15. <iterate> 描述符的属性

名字	描述
target	序列或词典。
index	用于序列索引的符号名。如果没有指定，缺省符号是 i。
element	用于序列元素的符号名。如果没有指定，缺省符号是 elem。
key	用于词典键的符号名。如果没有指定，缺省符号是 key。
value	用于词典值的符号名。如果没有指定，缺省符号是 value。

下面是 <iterate> 描述符的一个例子。如果雇员的薪水大于 $3000，就显示该雇员的姓名。

```
<iterate target="value.employeeMap" key="id" value="emp">
    <if test="emp.salary > 3000">
        <echo message="Employee: " value="emp.name"/>
    </if>
</iterate>
```

<if>

有条件地执行子描述符。表 22.16 描述了这个描述符支持的属性。

表 22.16. <if> 描述符的属性

名字	描述
test	一个布尔表达式。

更多关于描述符表达式语言的信息，参见 22.8 节。

<set>

修改一个值。 value 和 type 属性是互斥的。如果 target 表示的是词典元素，这个元素必须已经存在 (也就是说， <set> 不能用来把元素添加到词典中)。表 22.17 描述了这个描述符支持的属性。

表 22.17. <set> 描述符的属性

名字	描述
target	一个必须选中可修改的值的表达式。
value	一个表达式，对其求值所得的值必须与目标的类型兼容。
type	要实例化的类的类型 id。这个类必须与目标的类型兼容。
length	一个整型表达式，表示所需的序列新长度。如果新长度小于序列的当前尺寸，会从序列的末尾移除元素。如果新长度大于当前尺寸，会添加新元素到序列的末尾。如果同时还指定了 value 或 type，就会用来初始化每个新元素。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，转换会立刻失败。如果没有指定，缺省值是 false。

下面的 <set> 描述符修改词典元素的成员:

```
<set target="new.parts[‘P105J3’].cost"
      value="new.parts[‘P105J3’].cost * 1.05"/>
```

这个 <set> 描述符把一个元素添加到一个序列中，并初始化它的值:

```
<set target="new.partsList" length="new.partsList.length + 1"
     value="‘P105J3’"/>
```

<add>

把一个新元素添加到序列或词典中。在使用 <iterate> 遍历序列或词典的过程中添加元素是合法的，但添加之后的遍历次序是不确定的。与 value 和 type 属性一样， key 和 index 属性是互斥的。如果 value 和

type 都没有指定，新元素就会用缺省值初始化。表 22.18 描述了这个描述符支持的属性。

表 22.18. <add> 描述符的属性

名字	描述
target	一个必须选中可修改序列或词典的表达式。
key	一个表达式，对其求值所得的值必须与目标词典的键类型兼容。
index	一个表达式，对其求值所得的必须是整数值，表示插入的位置。新元素被插入到 index 之前。这个值不能超过目标序列的长度。
value	一个表达式，对其求值所得的值必须与目标词典的值类型、或目标序列的元素类型兼容。
type	要实例化的类的类型 id。这个类必须与目标词典的值类型、或目标序列的元素类型兼容。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，转换会立刻失败。如果没有指定，缺省值是 false。

下面是 <add> 描述符的一个例子，它添加一个新词典元素，然后初始化其成员：

```
<add target="new.parts" key="‘P105J4’"/>
<set target="new.parts[‘P105J4’].cost" value="3.15"/>
```

<define>

在当前作用域中定义一个新符号。表 22.19 描述了这个描述符支持的属性。

表 22.19. <define> 描述符的属性

名字	描述
name	新符号的名字。如果这个名字与当前作用域中的某个已有符号同名，就会发生错误。

表 22.19. <define> 描述符的属性

名字	描述
type	符号的形式 Slice 类型的名字。
value	一个表达式，对其求值所得的值应该与符号的类型兼容。
convert	如果为 true，支持额外的类型转换：在整数和浮点数之间，以及在整数和枚举之间。如果发生有效范围错误，执行会立刻失败。如果没有指定，缺省值是 false。

下面是 <define> 描述符的两个例子。第一个例子把符号 identity 的类型定义为 Ice::Identity，并用 <set> 初始化其成员：

```
<define name="identity" type="::Ice::Identity"/>
<set target="identity.name" value="steve"/>
<set target="identity.category" value="Admin"/>
```

第二个例子用我们最初在 22.3.5 节看到的枚举来定义符号 manufacturer，并把缺省值赋给它：

```
<define name="manufacturer" type="::BigThree"
    value="::DaimlerChrysler"/>
```

<remove>

从序列或词典中移除一个元素。在用 <iterate> 遍历序列或词典的过程中移除元素是合法的，但移除之后的遍历次序是不确定的。表 22.20 描述了这个描述符支持的属性。

表 22.20. <remove> 描述符的属性

名字	描述
target	一个必须选中可修改序列或词典的表达式。
key	一个表达式，对其求值所得的值必须与目标词典的键类型兼容。
index	一个表达式，对其求值所得的必须是一个整数值，表示要移除的序列元素的索引。

<fail>

致使转换立刻失败。如果指定了 test，只有当表达式的值为 true 时，转换才会失败。表 22.21 描述了这个描述符支持的属性。

表 22.21. <fail> 描述符的属性

名字	描述
message	一条在转换失败时显示的消息。
test	一个布尔表达式。

如果检测到有效范围错误，下面的 <fail> 描述符就终止转换：

```
<fail message="range error occurred in ticket count!"
      test="value.ticketCount > 32767"/>
```

<echo>

显示值和信息。如果没有指定属性，就只打印一个换行符。表 22.22 描述了这个描述符支持的属性。

表 22.22. <echo> 描述符的属性

名字	描述
message	一条要显示的消息。
value	一个表达式。该表达式的值会以结构化的格式显示。

下面是一个 <echo> 描述符，它使用了 message 和 value 属性：

```
<if test="value.ticketCount > 32767">
    <echo message="range error occurred in ticket count: "
          value="value.ticketCount"/>
</if>
```

22.7 使用 dumpdb

本节将描述怎样调用 **dumpdb** 工具，并就怎样最好地利用它给出了建议。

22.7.1 选项

这个工具支持 4.18 节列出的所有 Slice 处理器都支持的标准命令行选项。下面描述的是 **dumpdb** 专有的选项:

- **--load *SLICE***

  加载包含在文件 ***SLICE*** 中的 Slice 定义。如果必须加载多个文件，这个选项可以多次指定。但是，用户要负责确保没有重复的定义 (如果两个被加载的文件共享了同一个 include 文件，就可能发生这样的事情)。要避免重复定义，一种策略是加载一个 Slice 文件，其中只含有用于把每个要加载的 Slice 文件包括进来的 #include 语句。在这种情况下，如果被包括的文件正确地使用了 include 守卫，就不可能发生重复。

- **--key *TYPE***
  --value *TYPE*

  指定数据库键和值的 Slice 类型。

- **-e**

  说明正在检查的是 Freeze 逐出器数据库。这个选项是为了方便起见而提供的: 它将按照 Freeze 逐出器的情况，自动设置数据库键和值类型，因而不再需要使用 **--key** 和 **--value** 选项。特别地，Freeze 逐出器数据库的键类型 是 `Freeze::EvictorStorageKey`，值类型是 `Freeze::ObjectRecord`。这两个类型是在 Slice 文件 `Freeze/EvictorStorage.ice` 中定义的; 但是，你无需显式加载这个文件。

- **-o *FILE***

  创建一个名为 ***FILE*** 的文件，其中含有用于被加载的 Slice 定义的示例描述符。你必须用 **--key** 和 **--value** 选项、或 **-e** 选项，指定键和值的类型。如果使用了 **--select** 选项，它的表达式将包括在示例描述符中。你不需要指定数据库参数，因为在使用了 **-o** 选项的情况下，不会进行数据库遍历。

- **-f *FILE***

  执行文件 ***FILE*** 中的描述符。文件的 <database> 描述符会指定键和值类型，因此，在使用了 **-f** 的情况下，无需指定 **--key**、 **--value** 和 **-e** 选项。

- **--select EXPR**

  只显示表达式 ***EXPR*** 对其而言为真的那些记录。这个表达式可以引用符号 key 和 value。

22.7.2 数据库参数

如果你调用 **dumpdb** 检查数据库，它需要两个参数:

- **dbenv**

  数据库环境目录的路径名。

- **db**

  数据库文件的名字。 **dumpdb** 以只读方式打开这个数据库，而遍历发生会在一个事务中进行。

22.7.3 用例

22.7.1 节描述的命令行参数支持若干操作模式:

- 倾卸整个数据库。
- 倾斜数据库的部分被选中的记录。
- 生成示例描述符文件。
- 执行描述符文件。

在下面的部分将描述这些用例。

倾卸整个数据库

用 **dumpdb** 检查一个数据库的最简单的方式就是倾卸它的整个内容。你必须指定数据库键和值类型，加载必要的 Slice 定义，并提供数据库环境目录和数据库文件的名字。例如，下面这条命令将倾卸一个 Freeze 映射表数据库，其键类型是 string，值类型是 Employee:

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
  db emp.db
```

倾卸被选中的记录

如果你只对某些记录感兴趣，**--select** 选项提供了一种方便的方式来过滤 **dumpdb** 的输出。在下面的例子中，我们选择了来自会计部门的雇员：

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
  --select "value.dept == 'Accounting'" db emp.db
```

如果数据库记录含有多态的类实例，你在指定表达式时要小心：对于所有记录，这个表达式必须都能成功求值。例如，如果在对表达式进行求值时，遇到的某个数据成员在类实例中并不存在，**dumpdb** 就会立刻失败。在这种情况下，最安全的表达式编写方式是，在引用类实例的任何数据成员之前，都对这个实例的类型进行检查。

在下面的例子中，我们假定一个 Freeze 逐出器数据库含有一个类层次中的各种类的实例，而我们只对雇员数大于 10 的 Manager 的实例感兴趣：

```
$ dumpdb -e --load Employee.ice \
  --select "value.servant.ice_id == '::Manager' and \
  value.servant.group.length > 10" db emp.db
```

另外，如果 Manager 有派生类，那么也可以用不同的方式来编写这个表达式，让 Manager 和它的任何派生类的实例都被考虑进来：

```
$ dumpdb -e --load Employee.ice \
  --select "value.servant.ice_isA('::Manager') and \
  value.servant.group.length > 10" db emp.db
```

创建示例描述符文件

如果你需要更复杂的过滤或脚本能力，你必须使用描述符文件。最容易的着手使用描述符文件的途径，是用 **dumpdb** 来生成模板：

```
$ dumpdb --key string --value ::Employee --load Employee.ice \
  -o dump.xml
```

输出文件 dump.xml 是完整的，如果需要，可以立即执行，但通常你会把这个文件用作进一步定制的出发点。

如果指定了 **--select** 选项，它的表达式会包括在生成的 <record> 描述符中，作为一个 <if> 描述符的 test 属性的值。

注意，如果指定了 **-o**，不需要再指定数据库参数，因为 **dumpdb** 会在生成了示例文件之后立刻终止。

执行描述符文件

你可以使用 **-f** 选项来执行描述符文件。例如，我们可以用下面的命令来执行我们在前面生成的描述符文件：

```
$ dumpdb -f dump.xml --load Employee.ice db emp.db
```

22.8 描述符表达式语言

在 FreezeScript 中可以使用一种表达式语言。

22.8.1 运算符

这种语言支持常见的一些运算符：`and`、`or`、`not`、`+`、`-`、`/`、`*`、`%`、`<`、`>`、`==`、`!=`、`<=`、`>=`、`(`、`)`。注意，为了遵从 XML 语法限制，`<` 字符必须转码成 `<`。

22.8.2 直接量

你可以为整数、浮点数、布尔值，以及串指定直接量值。就直接量值的语法而言，表达式语言支持的语法与 Slice 支持的相同 (参见 4.7.5 节)，只有一个例外：串直接量必须放在单引号中。

22.8.3 符号

有些描述符会引入可在表达式中使用的符号。这些符号必须遵从 Slice 标识符的命名规则 (也就是说，在一个起头的字母后面跟有零个或更多个字母或数字字符)。访问数据成员要用用点分隔的方式，比如 `value.memberA.memberB`。

表达式可以用 "scoped name" 来引用 Slice 常量和枚举。在 **transformdb** 的描述符中，有两组 Slice 定义，因此，表达式必须指明它要访问的是哪一组定义：在 "scoped name" 前加上 `::Old` 或 `::New`。例如，如果数据成员 `fruitMember` 的枚举值是 `Pear`，表达式 `old.fruitMember == ::Old::Pear` 求值所得就是 `true`。在 **dumpdb** 中，只有一组 Slice 定义，因此，不用任何特殊前缀就可以标识出前缀或枚举符。

22.8.4 Nil

关键字 `nil` 表示的是类型为 `Object` 的 nil 值。这个关键字可用于在表达式中对 nil 对象值进行测试，也可用于把某个对象值设成 nil。

22.8.5 元素

你可以通过数组的形式来访问词典和序列元素，比如 `userMap['marc']` 或 `stringSeq[5]`。如果表达式试图获取的词典或序列元素不存在，就会发生错误。对于词典而言，我们建议你在访问某个键之前先检查它是否存在：

```
<if test="userMap.containsKey('marc') and userMap['marc'].active">
```

更多关于 `containsKey` 函数的信息，参见 22.8.8 节。

与此类似，涉及到序列的表达式应该检查序列的长度：

```
<if test="stringSeq.length > 5 and stringSeq[5] == 'fruit'">
```

关于 `length` 成员的详细情况，参见 22.8.7 节。

22.8.6 保留的关键字

以下关键字是保留的：`and`、`or`、`not`、`true`、`false`、`nil`。

22.8.7 隐含的成员

有些 Slice 类型支持隐含的数据成员：

- 词典和序列实例具有成员 `length`，表示元素的数目。
- 对象实例具有成员 `ice_id`，表示对象的实际类型；还有成员 `ice_facets`，其中含有对象的各个 facet。你可以像使用其他词典值一样使用 `ice_facets` 成员。

22.8.8 函数

表达式语言支持两种形式的函数调用：成员函数和全局函数。成员函数针对特定的数据值被调用，而全局函数不受限于哪个数据值。例如，下面的表达式调用了一个 `string` 值的 `find` 成员函数：

```
old.stringValue.find('theSubstring') != -1
```

而下面的例子调用了全局函数 `stringToIdentity`：

```
stringToIdentity(old.stringValue)
```

如果某个函数有多个参数，这些参数必须用逗号分隔。

String 成员函数

`string` 数据类型支持以下成员函数：

- `int find(string match[, int start])`

  返回子串的下标，如果没有找到，就返回 -1。你可以指定起始位置。
- `string replace(int start, int len, string str)`

  用一个新的子串替换串的给定部分，并返回修改后的串。
- `string substr(int start[, int len])`

  返回一个从给定位置开始的子串。如果提供了可选的长度参数，子串最多包含 `len` 个字符，否则子串就会包含串的余下部分。

词典成员函数

`dictionary` 数据类型支持以下成员函数:

- `bool containsKey(key)`

  如果词典中有某个元素具有给定的键，就返回 `true`，否则返回 `false`。 `key` 参数的值必须与词典的键类型兼容。

对象成员函数

对象实例支持以下成员函数:

- `bool ice_isA(string id)`

  如果对象实现了给定的接口类型，就返回 `true`，否则返回 `false`。这个函数不能在 nil 对象上调用。

全局函数

你可以使用以下全局函数:

- `string generateUUID()`

  返回一个新的 UUID。
- `string identityToString(Ice::Identity id)`

  把标识转换成它的串表示。
- `string lowercase(string str)`

  返回一个已转换成小写的新串。
- `string proxyToString(Ice::ObjectPrx prx)`

  返回给定代理的串表示。
- `Ice::Identity stringToIdentity(string str)`

  把串转换成一个 `Ice::Identity`。
- `Ice::ObjectPrx stringToProxy(string str)`

  把串转换成代理。

- `string typeOf(val)`

  返回参数的形式 Slice 类型。

22.9 总结

FreezeScript 提供的一些工具能使 Freeze 数据库的维护变得更轻松。在数据的持久类型发生变化时，**transformdb** 工具能够简化迁移任务。它提供了一种自动模式，不需要你进行手工干预；还提供了一种定制模式，能够让你编写脚本来控制改变。你可以使用 **dumpdb** 工具来进行数据库审查和报告，这种工具支持多种操作模式，包括可以执行脚本。

第 23 章
IceSSL

23.1 本章综述

在这一章我们将介绍 IceSSL，一个可选的 Ice 应用安全组件。 23.2 节将对 SSL 协议及支持它所需的基础设施进行综述。 23.3 节和 23.4 节将讨论 IceSSL 的配置问题，而 23.5 节将给出详细的配置文件语法。最后， 23.6 节演示怎样让应用与 IceSSL 直接交互。

23.2 引言

无论是在企业内部网中，还是在像 Internet 这样的非受信 (untrusted) 网络上，安全性对于许多分布式应用而言，都是一个重要的考虑事项。保护敏感信息、确保其完整性、检验各通信方的标识，这些能力对开发安全的应用而言必不可少。考虑到这些目标， Ice 提供了 IceSSL *插件*，能够通过 Secure Socket Layer (SSL) 协议为你提供这些能力[1]。

1. IceSSL 目前只能用在 C++ 应用中。

23.2.1 SSL 综述

SSL 协议能够让 Web 服务器执行安全的事务，因此是安全的网络通信中使用最多的协议之一。要想成功地使用 IceSSL，你无需知道 SSL 协议的技术细节 (这些细节超出了本书的范围)。但是，在高级层面上了解该协议的工作方式、以及支持它所需的基础设施，会对你有所帮助 (更多关于 SSL 协议的信息，参见 [22])。

通过结合多种密码技术， SSL 提供了一个安全的通信环境 (并且没有牺牲太多性能):

- 公钥加密
- 对称 (共享钥) 加密
- 消息认证代码
- 数字证书

客户会在建立与服务器的 SSL 连接时进行*握手*。在典型的握手过程中，会对标识各通信方的数字证书进行验证，并交换用于加密会话通信的对称钥。在握手阶段大量使用了公钥加密 （对大量的会话数据传送而言，公钥加密太慢了)。一旦握手完成， SSL 会使用消息认证代码来确保数据完整性，从而让客户和服务器在合理地相信它们的消息的安全的情况下，任意进行通信。

23.2.2 公钥基础设施

安全性需要的是信任，而公钥密码系统自身无助于建立信任。 SSL 使用了 Public Key Infrastructure(PKI) 来处理信任问题，PKI 使用证书把公钥绑定到标识。 Certification Authority (CA) 常被用来发放证书，并检验证书所有者的标识。

较小的应用在基础设施方面的需求常常很少；使用自由工具为客户和服务器创建私有证书可能就足够了。但是，企业具有更为精细的安全性需求，必须设置私有 CA[2] 或使用第三方 CA (比如 Verisign)。

完全避免使用证书也是有可能的；加密仍然被用来遮掩会话通信，但为了降低复杂度和管理开销，验证的好处被牺牲掉了。

关于 PKI 的更多信息，参见 [5]。

2. OpenSSL 是 IceSSL 所使用的开放源码 SSL 工具包，它提供的工具可以设置最低限度的 Certification Authority，也许适用于开发目的，或是小型组织。

23.2.3 要求

要把 IceSSL 集成进你的应用，通常不需要改动你的源代码，但会涉及到以下管理任务:

- 创建一个公钥基础设施 (如果有必要)
- 编写用于 IceSSL 插件的 XML 配置文件
- 修改你的应用的配置，以安装 IceSSL、并使用安全的连接

本章的余下部分将主要讨论 IceSSL 的配置。

23.3 配置 IceSSL

你可以分别配置 IceSSL 的客户功能和服务器功能，这反映了一个 Ice 应用可以充当的角色：只充当客户、只充当服务器、混合的客户 - 服务器。配置数据用 XML 编写，提供了 IceSSL 初始化 SSL 协议所需的信息，包括加密算法的选择，以及证书和键文件名。本节将给出配置文件的例子，而 23.5 节将给出详细的参考资料，对 IceSSL 配置的 XML 元素加以说明。

23.3.1 RSA 例子

Ice 把下面给出的文件用于与 IceSSL 有关的测试及示例程序 (参见 Ice 源码包中的 `certs/sslconfig.xml` 文件)。

每一个 IceSSL 配置文件的顶层元素都是 `SSLConfig`。嵌套在这个元素中的是一个 `client` 或 `server` 元素 (在这个例子中，两个元素都有):

```
<?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>
```

你可以看到， client 和 server 元素非常类似，让我们来更详细地检查一下 client 的内容 (这些讨论也适用于 server 元素)。我们遇到的第一个元素是 general，它选择加密算法，并控制证书验证:

```
        <general version="SSLv23" cipherlist="RC4-MD5"
                 verifymode="peer" verifydepth="10" />
```

version 属性选择协议版本; 在这里， SSLv23 实际上是一种兼容模式，包括了 SSL 版本 2、 SSL 版本 3，以及 Transport Layer Security (TLS) 版本 1。但要注意， SSL 版本 2 有若干安全缺陷，所以 IceSSL 不支持它。

cipherlist 属性选择的是密码套件。 RC4-MD5 选择 RC4 作为加密算法，选择 MD5 作为消息摘要算法。

verifymode 和 verifydepth 属性会影响证书认证。 peer 的验证模式要求客户在让连接继续建立之前，认证服务器的证书，并允许客户在被要求的情况下把它的证书发送给服务器。一个证书可以拥有一个任意长的签名证书链，必须对其进行搜索以找到受信的 CA ; verifydepth 属性允许你设置搜索深度的限度。如果超过了这个限度，连接就会失败。

接下来， certauthority 元素指定一个 受信 CA 证书 (或证书链) 的文件名:

```
        <certauthority file="cacert.pem" />
```

顾名思义，证书文件名的编码格式是 Privacy Enhanced Mail (PEM)，IceSSL 要求所有证书和密钥都使用这种格式。

最后， basecerts 元素提供了标识对端和认证证书所需的公开证书和私钥:

```
        <basecerts>
            <rsacert keysize="1024">
                <public encoding="PEM"
                        filename="c_rsa1024_pub.pem" />
                <private encoding="PEM"
                        filename="c_rsa1024_priv.pem" />
            </rsacert>
        </basecerts>
```

rsacert 元素封装了 public 和 private 元素，并提供了密钥的位长度 (1024)。 public 元素定义了公开证书的文件名和编码方式。与此类似， private 定义了私钥的文件名和编码方式。尽管编码格式是在一个属性中提供的，目前支持的编码格式只有 PEM。

23.3.2 ADH 例子

下面的例子使用了 ADH (Anonymous Diffie-Hellman 密码)。在大多数情况下 ADH 都不是一个好的选择，因为，顾名思义，它不会对各通信方进行认证，并且易于受到 "man-in-middle" 攻击 (参见 http://sun.soci.niu.edu/~rslade/secgloss.htm#man-in-the-middle——译注)。但是，它仍然提供了会话通信加密，而且所需的管理非常少，因此可以用在某些情况下。下面给出的 IceSSL 配置文件演示了怎样使用 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>
```

在这个配置中，重要的是 ADH 密码的选择，以及使用了一个 dhparams 元素来提供 Diffie-Hellman 参数文件。我们为客户和服务器指定了同一个参数文件，但它们的参数不需要相同。在握手过程中，一端的 Diffie-Hellman 参数会与另一端共享，所以实际上只使用了一组参数。使用

哪一端的参数要由 SSL 实现来决定，但通常会用服务器的参数。我们没有指定任何证书，因为不存在对端认证。

23.3.3 配置文件策略

IceSSL 配置文件提供了许多灵活性。例如，如果客户和服务器可以访问同一个文件系统，把它们的 IceSSL 配置放在同一个文件中可以简化管理工作。但是，为客户和服务器分别创建文件，让客户的配置文件只包含一个 `client` 元素，让服务器的配置文件只包含一个 `server` 元素，这也是完全合理的 (而且常常也是必要的)。

如果客户或服务器是混合模式的应用 (也就是说，既发送请求，也接收请求)，那么在共享 IceSSL 配置时你必须要小心：混合模式的应用的配置既需要有 `client` 元素，也需要有 `server` 元素，而同样的配置 (密钥、证书，等等) 并一定适用于另一端。

23.4 配置应用

配置一个应用、让它使用 IceSSL，需要安装插件，并创建 SSL 端点。

23.4.1 安装 IceSSL

Ice 支持一种通用的插件设施，允许 IceSSL 这样的扩展进行动态安装，而不用改动应用的源码。 IceSSL 的可执行代码驻留在 Unix 的共享库和 Windows 的动态链接库 (DLL) 中。要安装这个插件，使用：

```
Ice.Plugin.IceSSL=IceSSL:create
```

属性名的最后一部分 (`IceSSL`) 将变成插件在配置时的正式名称，但 IceSSL 插件要求它的标识符是 `IceSSL`。属性值 `IceSSL:create` 已足以让 Ice run time 定位 IceSSL 库 (在 Unix 和 Windows 上都是如此)，并对插件进行初始化。唯一的要求是库所在的目录必须出现在库路径中 (在 Unix 上是 `LD_LIBRARY_PATH`，在 Windows 上是 `PATH`)。关于 `Ice.Plugin` 属性的更多信息，参见附录 C。

下一步是要向插件提供它的配置文件，这也要通过配置属性来完成：

```
IceSSL.Client.CertPath=/opt/certs
IceSSL.Client.Config=sslconfig.xml
IceSSL.Server.CertPath=/opt/certs
IceSSL.Server.Config=sslconfig.xml
```

与 23.3 节描述的配置数据类似，这些属性被划分成客户版本和服务器版本。客户只需指定 IceSSL.Client 属性，服务器只需指定 IceSSL.Server 属性。在初始化插件时，客户配置会从 IceSSL.Client.Config 属性所指定的文件中读取，而服务器配置将从 IceSSL.Server.Config 属性所指定的文件中读取。(在这两个属性中指定同一个文件是合法的)。 CertPath 属性指定一个缺省目录，在其中可以找到客户和服务器的配置及证书文件。

这个插件还支持更多的配置属性，在附录 C 中对它们进行了描述，但上面的这些属性对许多应用而言已经足够了。

23.4.2 创建 SSL 端点

安装了 IceSSL 插件之后，你就可以在你的端点中使用一种新协议 ssl 了。例如，下面的端点列表创建了一个 TCP 端点、一个 SSL 端点，以及一个 UDP 端点:

```
MyAdapter.Endpoints=tcp -p 8000:ssl -p 8001:udp -p 8000
```

如这个例子所演示的， UDP 端点可以使用和 TCP 或 SSL 端点相同的端口号，因为 UDP 是一种不同的协议，有自己的端口集。但 TCP 端点和 SSL 端点不能使用同一个端口号，因为 SSL 在本质上是位于 TCP 之上的一个层面。

在串化代理中使用 SSL 同样直截了当:

```
MyProxy=MyObject:tcp -p 8000:ssl -p 8001:udp -p 8000
```

关于代理和端点的更多信息，参见附录 D。

23.4.3 安全考虑事项

像 23.4.2 节中的例子那样，让对象适配器的端点使用多种协议，对安全有一些明显的影响。如果你的意图是用 SSL 来保护会话通信，并且 / 或者限制对服务器的访问，那么你应该只定义 SSL 端点。

但在有些情况下，使用不安全的端点协议也有好处。图 23.1 阐释了一种环境，在防火墙以内可以使用 TCP，但外部客户必须使用 SSL。

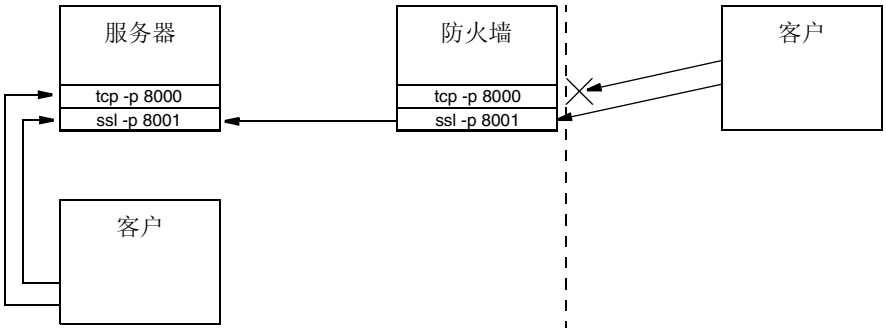

图 23.1. 一个多协议端点应用

图 23.1 中的防火墙被配置成阻塞外部对 TCP 端口 8000 的访问，并把与端口 8001 的连接转到服务器的机器。

在防火墙以内使用 TCP 的一个原因是，它比 SSL 效率更高，使用时时所需的管理工作更少。当然，这里的例子假定内部客户是可信的，在许多环境中事情未必如此。

关于怎样在复杂的网络架构中使用 SSL 的更多信息，参见第 24 章。

23.5 配置参考资料

本节描述 Ice 配置文件的每一个 XML 元素。

23.5.1 结构

IceSSL 配置文件具有这样的结构:

```
<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>
```

有些元素是可选的；详情参见元素描述。在 Ice 源码包中的 `certs/sslconfig.dtd` 文件中提供了配置文件的 DTD。

23.5.2 basecerts

`basecerts` 元素含有证书和密钥规范。在 `client` 和 `server` 中必须有这个子元素，而在这个元素中可以含有一个 `rsacert` 元素、一个 `dsacert` 元素，以及一个 `dhparams` 元素。特别地，`basecerts` 必须含有 `rsacert` 元素或 `dsacert` 元素(或两者都有)。如果提供了 `dsacert` 元素，通常也要提供 `dhparams` 元素。

23.5.3 certauthority

`certauthority`元素指定的是含有受信的CA证书链的文件，以及证书所在的目录。它是 `client` 和 `server` 的可选子元素，支持以下属性:

表 23.1. certauthority 元素的属性

属性	描述	必须指定
file	一个含有受信 CA 证书链的文件 (PEM 格式)。如果是相对路径名，将使用 `CertPath` 属性定义的目录进行解析。	否
path	含有受信的 CA 证书的目录的绝对路径名。在使用之前，需要在这个目录中运行 OpenSSL 实用程序 **`c_rehash`**，进行准备。	否

这两个属性都被标记为可选的，但你至少应该定义其中一个属性。这两个属性的一个重要差异是证书加载的时间。 `file` 属性所指示的证书会立刻加载，而位于 `path` 属性所指示的目录中的证书则会在 SSL 握手过程中、按照需要加载。

23.5.4 client

`client`元素为IceSSL插件的客户端组件建立配置。它是`SSLConfig`的可选子元素，如果你指定了它，它必须包含一个 `general` 元素和一个 `basecerts` 元素，还可以包含一个 `certauthority` 元素。

23.5.5 dhparams

dhparams 元素指定 Diffie-Hellman 密钥约定算法的参数。它是 basecerts 和 tempcerts 的可选子元素，支持以下属性:

表 23.2. dhparams 元素的属性

属性	描述	必须指定
keysize	质数参数中的位数	是
encoding	filename 属性所指示的文件的编码格式。其值必须是 PEM。如果没有指定，缺省值是 PEM。	否
filename	一个含有 Diffie-Hellman 参数的文件。如果是相对路径名，将使用 CertPath 属性定义的目录进行解析。	是

如果指定了这个元素，但无法从 filename 属性指定的文件中读取参数， IceSSL 会使用 512 位的质数来生成临时的 Diffie-Hellman 参数。

23.5.6 dsacert

dsacert 元素定义Digital Signature Algorithm (DSA)的公开证书和私钥。它是 basecerts 和 tempcerts 的可选子元素，必须包含一个 public 元素和一个 private 元素。它支持以下属性:

表 23.3. dsacert 元素的属性

属性	描述	必须指定
keysize	密钥的位强度。	是

23.5.7 general

`general` 元素对 SSL 协议进行配置。`client` 和 `server` 必须包含这个子元素。它支持以下属性:

表 23.4. general 元素的属性

属性	描述	必须指定
version	SSL 协议版本。可能的值有 `SSLv23`、 `SSLv3`，以及 `TLSv1`。值 `SSLv23` 是一种兼容模式，包括了 SSL 版本 2、 SSL 版本 3，以及 TLS 版本 1。注意， SSL 版本 2 有若干安全缺陷，所以 IceSSL 不支持它。如果没有指定，其缺省值是 `SSLv23`。	否
cipherlist	允许 SSL 使用的密码套件列表，用冒号分隔。如果没有指定，会使用一个缺省的密码套件列表。	否
context	一个会话 ID 上下文。如果指定了这个属性，在服务器中会缓存会话。	否
verifymode	证书检验行为。可能的值及其语义，参见表 23.6。可以指定多个值，用空格或竖线 (|) 分隔。如果没有指定，其缺省值是 `none`。	否
verifydepth	在为找到受信 CA 而搜索证书链时的最大深度。如果超过了这个深度，握手就会失败。如果没有指定，其缺省值是 `10`。	否
randombytes	用来自一个或多个文件的数据给伪随机数发生器提供种子。可以用路径名表示支持 EGD 协议的 UNIX 域 socket。多个路径名必须用分号 (`;`)(在 Windows 上) 或冒号 (`:`) (在 Unix 上)。必须至少提供 512 字节的数据。	否

cipherlist

`cipherlist` 属性标识的是允许 SSL 在握手过程中磋商的密码套件。密码套件是一组能够满足四个建立安全连接的要求的算法: 签名和认证、密钥交换、安全哈希，以及加密。有些算法能满足不止一个要求，而且存在着许多可能的组合。

这个属性的值会直接送给低级的 OpenSSL 库，这个库是 IceSSL 的基础。因此， OpenSSL 将会决定可以使用哪些密码套件，而这又取决于 OpenSSL 包的编译方式。你可以用下面的 **`openssl`** 命令来获取所支持的密码套件的完整列表:

```
$ openssl ciphers
```

这个命令很可能会生成很长的列表。为了简化选择过程，OpenSSL 支持若干密码类:

表 23.5. 密码类

类	描述
ALL	所有可能的组合。
ADH	匿名密码。
LOW	低位强度密码。
EXP	"Export-crippled" 密码。

你可以通过在类和密码前面加上感叹号来排除它们。特殊的关键字 @STRENGTH 会按照密码的强度对密码列表进行排序，让 SSL 在磋商密码套件时优先使用更安全的密码。 @STRENGTH 关键字必须是列表中的最后一个元素。

例如，下面是 cipherlist 属性的一个好值:

```
ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH
```

这个值排除了低强度的和有问题的密码，并按照强度对余下的密码进行排序。

注意，如果指定的密码无法识别，不会有警告。

verifymode

verifymode 属性具有以下语义:

表 23.6. verifymode 属性的语义

值	客户语义	服务器语义
none	客户检验服务器的证书，但如果失败，并不会终止握手。如果服务器要求发送证书，客户不会发送。	服务器不要求客户发送证书。
peer	客户检验服务器的证书。如果检验失败，握手会终止。	服务器要求客户发送证书，并在收到证书的情况下对其进行检验。如果检验失败，握手会终止。
fail_no_cert	无。	如果客户没有提供证书，握手就会失败。必须与 peer 结合使用。
client_once	无。	阻止服务器在重新磋商的过程中要求客户发送证书。必须与 peer 结合使用。

值 none 不能与其他值结合使用。在客户配置中，值 peer 不能与其他值结合使用。

23.5.8 private

private 元素指定的是签名算法的私钥。rsacert 和 dsacert 必须包含这个子元素。它支持以下属性:

表 23.7. private 元素的属性

属性	描述	必须指定
encoding	filename 属性所指定的文件的编码格式。其值必须是 PEM。如果没有指定，缺省值是 PEM。	否
filename	一个含有私钥的文件。如果是相对路径名，将使用 CertPath 属性定义的目录进行解析。	是

23.5.9 public

public 元素指定的是签名算法的公开证书。 rsacert 和 dsacert 必须包含这个子元素。它支持以下属性:

表 23.8. public 元素的属性

属性	描述	必须指定
encoding	filename 属性所指定的文件的编码格式。其值必须是 PEM。如果没有指定，缺省值是 PEM。	否
filename	一个含有公开证书的文件。如果是相对路径名，将使用 CertPath 属性定义的目录进行解析。	是

23.5.10 rsacert

rsacert 元素定义 RSA 公开证书和私钥。它是 basecerts 和 tempcerts 的可选子元素，必须含有一个 public 元素和一个 private 元素。它支持以下属性:

表 23.9. dsacert 元素的属性

属性	描述	必须指定
keysize	密钥的位强度。	是

23.5.11 server

server 元素为 IceSSL 插件的服务器端组件建立配置。它必须包含一个 general 元素和 basecerts 元素，还可以包含一个 certauthority 元素和一个 tempcerts 元素。

23.5.12 SSLConfig

SSLConfig 是 IceSSL 配置文件中的顶层元素。它必须包含一个 client 元素，或一个 server 元素，或两者都包含。

23.5.13 tempcerts

`tempcerts` 元素含有临时密钥的配置参数参数。它是 `server` 的可选子元素，还可以包含多个 `rsacert` 和 `dhparams` 元素。

在使用 DSA 验证时需要使用临时 (也称为 "*ephemeral*") 密钥，并在使用 RSA 验证时提高安全性 (尽管临时的 RSA 密钥很少用在实践中)。`tempcerts` 元素允许你指定含有 RSA 公开证书和私钥、以及 Diffie-Hellman 参数的文件。从文件中加载这些内容，可以不必动态地生成它们，后者在计算上可能会很昂贵。在 SSL 请求某个密钥尺寸之前，与之相配的文件不会被读取；如果没有找到具有相配的密钥尺寸的文件，所请求的数据就会动态生成。

23.6 IceSSL 编程

有些应用可能需要使用只能通过程序方式使用的 IceSSL 功能。下面是应用为何需要与插件直接交互的一些原因:

- 为了动态加载配置文件
- 为了动态地添加密钥和受信证书
- 为了安装定制的证书检验规则

为了完成上面的任何一项任务，应用都需要从通信器那里获取一个指向 IceSSL 插件的引用。 Ice 定义了 Slice 接口 `Ice::Plugin` 来代表一个一般的插件，而 IceSSL 定义了一个派生接口 `IceSSL::Plugin`。这些接口将在附录 B 中描述，但下面的代码例子演示了怎样获取指向 IceSSL 插件的引用:

```
Ice::CommunicatorPtr communicator = // ...
Ice::PluginManagerPtr pluginMgr =
    communicator->getPluginManager();
Ice::PluginPtr plugin = pluginMgr->getPlugin("IceSSL");
IceSSL::PluginPtr sslPlugin =
    IceSSL::PluginPtr::dynamicCast(plugin);
```

第一步是获取指向插件管理器的引用，然后向它请求 IceSSL 插件。`getPlugin` 的参数是插件标识符，在这里是 `IceSSL` (参见 23.4.1 节)。最后，我们向下转换到 `IceSSL::Plugin`。

23.7 总结

Secure Socket Layer (SSL) 协议是安全的网络通信的事实标准。它支持验证、不可否认性、数据完整性，以及强加密，从而使得它成了安全的 Ice 应用的当然选择。

尽管安全只是 Ice 的一个可选的成分，它并非是事后才追加的。 IceSSL 插件能够轻松地集成进已有的 Ice 应用中，而且在大多数情况下只需要改动配置。很自然，要为应用创建必需的安全基础设施，需要付出一些额外的努力，但在许多企业中，这项工作已经完成了。

第 24 章
Glacier

24.1 本章综述

这一章将介绍 Glacier，用于 Ice 应用的防火墙解决方案。 24.2 节将简要介绍 Glacier，并描述 Glacier 的设计想要解决的复杂的网络问题。 24.3 节将给出最简单的 Glacier 用例，其中的客户只需做出配置上的改动。 24.4 节解决的是回调的问题，以及 Glacier 怎样在受限的网络环境中支持回调。 24.5 节的主题是 Glacier 启动器， 24.6 节将详细介绍使用 Glacier 时需要考虑的安全问题。

24.2 引言

在本书中我们给出过许多客户 / 服务器的例子，所有这些例子都假定客户和服务器程序或者运行在同一个主机上，或者运行在没有网络限制的多个主机上。我们可以证明这样的假定是正当的，因为这是一本介绍性的书，但现实世界的网络环境通常要复杂得多：能够访问公共网络的客户和服务器主机常常位于保护性的路由器 - 防火墙之后，它们不仅要限制进入的连接，而且还要允许受保护的网络使用 Network Address Translation (NAT)、在私有的地址空间中运行。实际上，这些特性在今天的危险的网络环境中是必需的；它也破坏了我们的例子所假定的理想世界。

24.2.1 常见情况

让我们假定，有客户和服务器需要在非受信的网络上进行通信，它们位于防火墙后面的私有网络中。如图 24.1 所示:

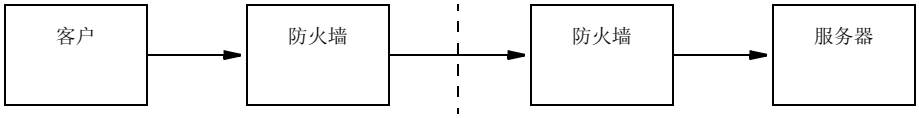

图 24.1. 典型的网络环境下的客户请求

尽管这幅图看起来相当直截了当，有一些麻烦的问题需要解决:

- 在服务器上必须打开一个专用端口，并进行配置，把消息转发给服务器。
- 如果服务器使用多个端点 (例如，既支持 TCP，也支持 SSL)，那么每个端点都要有专用的防火墙端口。
- 客户的代理必须进行配置，使用服务器的 “公开” 端点，也就是防火墙的主机名和专用端口。
- 如果服务器把一个代理作为请求的结果返回，这个代理包含的不能是服务器的私有端点，因为客户无法访问这个端点。

图 24.2 使得这种情况变得更加复杂化，它增加了一个从服务器到客户的回调。回调意味着客户也是服务器，因此与图 24.1 相关联的所有问题现在也适用于客户。

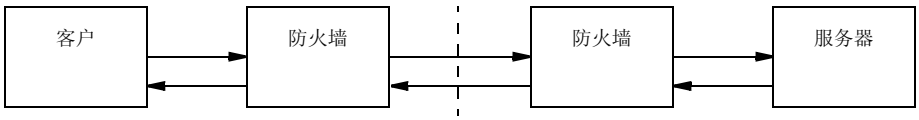

图 24.2. 典型的网络环境中的回调

就好像这还不够复杂，图 24.3 又增加了多个客户和服务器。随着更多的端口专用于转发请求，每个额外的服务器 (包括需要回调的客户) 都给防火墙管理员增加了更多的工作。

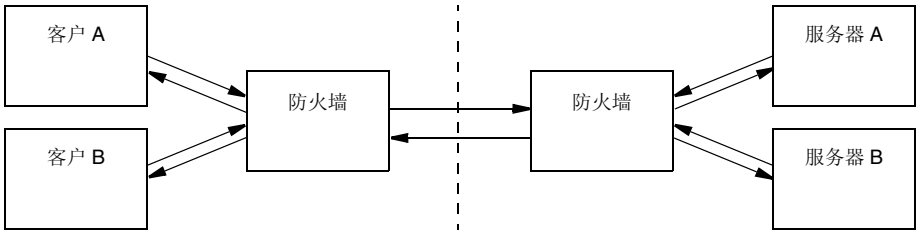

图 24.3. 典型网络环境中的有回调的多个客户和服务器

显然，这些做法的可伸缩性不好，而且有着过度的复杂性。幸运的是，Ice 提供了一种解决方案。

24.2.2 什么是 Glacier?

Glacier，用于 Ice 应用的路由器 - 防火墙，解决了 24.2.1 节提出的问题，而且对客户和服务器 (或防火墙管理员) 影响也很小。在图 24.4 中，Glacier 变成了 Ice 应用的服务器防火墙。但在图中并不明显的是， Glacier 是怎样消除前面的做法的大部分复杂性的。

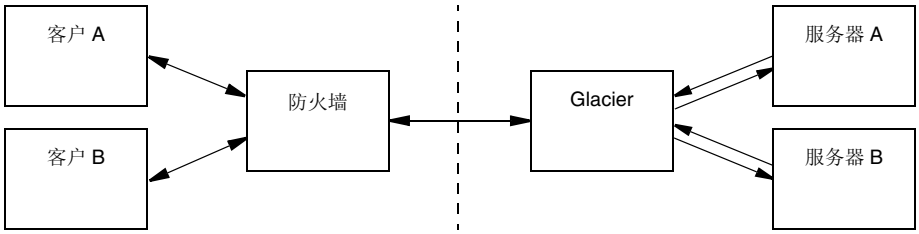

图 24.4. 使用 Glacier 的有回调的多个客户和服务器

特别地， Glacier 提供了以下好处:

- 要使用 Glacier，客户常常只需改动配置，无需改动代码。
- 单个 Glacier 端口就能支持任意数目的服务器。

- 服务器不知道 Glacier 的存在，也无需为了使用 Glacier 而做出任何修改。从服务器的角度看， Glacier 只是一个本地客户 (local client)，因此服务器再也不需要 通告它们创建的代理中的 “公开”端点了。此外，在 Glacier 防火墙的后面，还可以继续透明地使用像 IcePack (参见第 20 章) 这样的服务。
- 服务器无需为了进行回调而建立与客户的新连接。换句话说，服务器对客户的回调会在客户与服务器之间的已有连接上发送，从而消除了在客户防火墙中支持回调所需的管理工作。
- Glacier 支持所有 Ice 协议 (TCP、 SSL，以及 UDP)。
- Glacier 不需要对应用有特别的了解，因此非常高效：它无需解编消息内容，就可以对请求和答复消息进行路由。

24.2.3 工作方式

Ice 核心支持一种一般化的路由设施，由 `Ice::Router` 接口表示，允许第三方服务 “拦截”进行了适当配置的代理上的请求，并把这些请求递送给指定的服务器。 Glacier 是这个服务的一种实现，但你肯定也可以编写其他的实现。

Glacier 通常会运行在能够同时访问两个网络的主机上：客户发送请求所用的公共网络，以及把这些请求路由给服务器所用的私有网络。尽管在有些情况下，在防火墙后面使用 Glacier 也是可能的， Glacier 的实际设计目标就是要成为 Ice 应用的防火墙。所以 Glacier 在两个网络上都必须有端点，如图 24.5 所示。

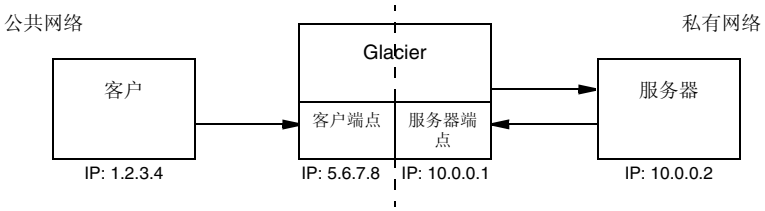

图 24.5. Glacier 的客户和服务器端点

在客户中，代理必须进行配置，把 Glacier 用作路由器。你可以针对某个通信器创建的所有代理静态进行这样的配置，也可以在程序中针对特定的代理进行配置。被配置成要使用路由器的代理被称为 “*有路由的代理*” (routed proxy)。

当客户调用有路由的代理上的操作时，客户会连接到 Glacier 的某一个客户端点，并发送请求，就好像 Glacier 是服务器一样。 Glacier 随即建立一个通向指定服务器的客户连接，转发请求，并返回答复（如果有）。在本质上， Glacier 充当的是代表远地客户的本地客户。

如果服务器把一个代理作为操作的结果返回，这个代理和平常一样，含有服务器在私有网络中的服务器端点（记住，服务器不知道 Glacier 的存在，因此假定发送请求的客户是能够使用这个代理的）。当然，客户无法访问这些端点，如果它在没有路由器的情况下使用该代理，就会收到异常。但如果该代理配置了路由器，客户就会忽略服务器的端点，只向路由器的客户端点发送请求。

Glacier 的服务器端点位于私有网络中，只能用于服务器向客户进行回调。更多关于回调的信息，参见 24.4 节 。

24.3 使用 Glacier

最简单的 Glacier 配置将使用 Glacier 路由器的单个实例。这种配置能够把请求从多个客户路由到多个服务器，但不支持服务器对客户进行回调。

这种配置的创建十分直截了当:

1. 为 Glacier 编写一个配置文件。
2. 在能够访问公共和私有网络的主机上启动路由器。
3. 修改客户配置，让它使用 Glacier 路由器。

在下面的例子中，让我们假定路由器的公开地址是 `5.6.7.8`，私有地址是 `10.0.0.1`。

24.3.1 配置路由器

下面的路由器配置属性建立了必需的端点:

```
Glacier.Router.Endpoints=tcp -h 5.6.7.8 -p 8000
Glacier.Router.Client.Endpoints=tcp -h 5.6.7.8
```

`Glacier.Router.Endpoints` 所定义的端点叫做*路由器控制*端点，因为 Ice run time 会在客户中使用它，直接与路由器交互。`Glacier.Router.Client.Endpoints` 所定义的端点是有路由的代理发送的请求的目的地。这些端点都必须能被客户访问，因此会在公共网络接口上定义[1]。

路由器控制端点使用了一个固定的端口，因为客户可能静态配置了针对这个端点的代理。而客户端点不需要使用固定端口。

24.3.2 启动路由器

假定 24.3.1 节中给出的配置属性存储在名为 `config` 的文件中，你可以用下面的命令启动路由器:

```
$ glacierrouter --Ice.Config=config
```

24.3.3 配置客户

下面的属性配置客户中的所有代理，以使用 Glacier 路由器:

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

这个属性的值是一个路由器控制对象的代理，因此这个端点必须与 `Glacier.Router.Endpoints` 中的某个端点匹配。

24.3.4 例子

`demo/Ice/hello` 例子可用于测试这个配置。事实上，这个例子的配置文件已经包含了与使用 Glacier 路由器有关的属性定义，但你可能需要对这个文件进行编辑，启用某些在缺省情况下被放在注释里的属性。更多的信息，参见配置文件中的注释。

24.4 回调

在分布式应用中经常要从服务器回调客户，目的常常是发送通知 (比如长时间进行的计算已完成，或对数据库记录进行了改动)。遗憾的是，如 24.2.1 节所述，在复杂的网络环境中支持回调会带来一组与此相关的问题。Ice 使用 Glacier 路由器和双向连接克服了这些困难。

24.4.1 双向连接

常规的无路由的连接只允许请求单向流动 (从客户到服务器)，而双向连接能够让请求双向流动。为了避开 24.2.1 节所讨论的网络限制 （也就是，客户端防火墙不允许服务器直接建立与客户的独立连接)，这是一种必须具备的能力。通过在已有的从客户到服务器 （更准确地说，是从客户

1. 这个示例配置使用了 TCP 作为端点协议，尽管在大多数情况下 SSL 更可取 (参见 24.6 节)。必须使用 TCP 端点的一种情况是 Java 客户，因为 Ice for Java 目前不支持 SSL。

到路由器）的连接上发送回调请求，我们创建了一个回到客户的虚拟连接。图 24.6 阐释了用 Glacier 进行回调所涉及的步骤。

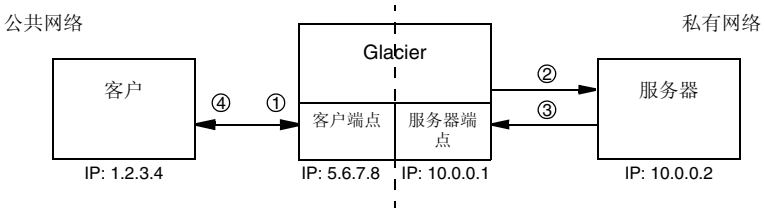

图 24.6. 通过 Glacier 进行回调

1. 客户有一个有路由的指向服务器的代理，并发出了一个调用。一个通往路由器的客户端点的连接被建立起来，请求被发往路由器。
2. 路由器使用来自客户的代理的信息，建立一个通往服务器的连接，并且转发请求。在这个例子中，请求中的参数之一是一个代理，指向客户中的回调对象。
3. 服务器对客户进行回调。要想成功回调，回调对象的代理必须含有能被服务器访问的端点。回到客户的唯一路径是经过路由器，因此代理含有路由器的服务器端点 (参见 24.4.4 节)。服务器连接到路由器，并发送请求。
4. 路由器用在步骤 1 中建立的双向连接把回调请求转发到客户。

图 24.6 中的箭头表示的是请求的流向；注意，在路由器和服务器之间使用了两个连接。因为服务器不知道有路由器，它没有使用有路由的代理，因此没有使用双向连接。

24.4.2 双向连接的生命期

客户在终止时，会关闭通往路由器的连接。如果随后服务器试图回调客户，就会失败，因为路由器没有通往客户的连接可用来转发请求。这种情况并不比服务器试图直接联系客户更糟，后面这种做法会被客户防火墙阻止。但是，这种情况说明了双向连接的固有局限：只有在客户与路由器还有连接的情况下，它才能被回调。

24.4.3 回调要求

在使用回调时，我们面临的首要问题是，单个 Glacier 路由器实例无法把回调请求转发给多个客户 。因此，任何使用回调的客户都必须使用它自

己的 Glacier 路由器。这个要求造成了管理上的困境：你怎样才能（轻松地）确保每个客户都获得它自己的路由器实例？我们将在 24.5 节给出这个问题的解决方案；目前，让我们假定我们只有一个客户和一个路由器实例。

24.4.4 配置路由器

要让路由器支持来自服务器的回调，它需要拥有在私有网络中的端点。下面给出的配置文件增加了 `Glacier.Router.Server.Endpoints` 属性：

```
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
```

与 24.3.1 节描述的客户端点类似，服务器端点不需要使用固定的端口。

24.4.5 配置客户对象适配器

收到回调的客户也是服务器，因此必须有对象适配器。在典型情况下，对象适配器有一些端点在局域网中，但那些端点对我们的受限的网络环境中的服务器没有用处。我们实际上想让代理使用路由器的服务器端点，为此，我们会用路由器的代理来配置配置对象适配器。我们只需使用一个配置属性：

```
Ice.Default.Router=Glacier/router:tcp -h 5.6.7.8 -p 8000
```

回想一下 24.3.3 节的内容，`Ice.Default.Router` 属性会给所有代理配置一个路由器。在服务器上下文中，这个属性还会给所有对象适配器配置一个路由器。但要注意，同一个通信器创建的多个对象适配器不能使用同一个路由器，因此，当我们在服务器上下文中使用这个属性时，必须要小心。因为我们的客户只有一个对象适配器，这正好是我们所需要的 (24.4.7 节讨论了需要多个对象适配器的客户可以采取的策略)。

对于每一个对象适配器，Ice run time 维护有一个端点列表，其中包含了该适配器所创建的各个代理的端点。通常，这个列表只含有为对象适配器定义的本地端点，但在适配器配置了路由器的情况下，这个列表也含有路由器的服务器端点。在我们的回调例子中，客户不需要任何本地端点，因为它并不期望收到来自本地服务器的回调，所以代理只含有路由器的服务器端点。

创建没有自己的端点的对象适配器似乎并不寻常，而且，在没有路由器的情况下，也没有理由这样做。但是，在使用路由器时，为了对回调请求

进行服务，必须使用对象适配器。尽管对象适配器没有本地端点，因此无法接受新的本地连接，但它的确会在客户建立与路由器的外出连接时，收到“虚拟的”连接。

24.4.6 活动连接管理

Ice 支持*活动连接管理*(ACM)。ACM 通过周期性地关闭空闲连接来节省资源。这是一个非常有用的特性，但通常不应该用于双向连接，否则 ACM 就可能会关闭仍然在等待回调的连接。在缺省情况下 ACM 是启用的 (对 `Ice.ConnectionIdleTime` 属性的描述，参见附录 C)。

24.4.7 对象适配器策略

需要支持来自路由器及本地客户的回调请求的应用应该使用多个对象适配器。这种策略能确保这些对象适配器所创建的代理含有适当的端点。例如，假定我们的网络配置如图 24.7 所示。要注意两个局域网使用的是相同的私有网络地址，这并非是一种不切实际的安排。

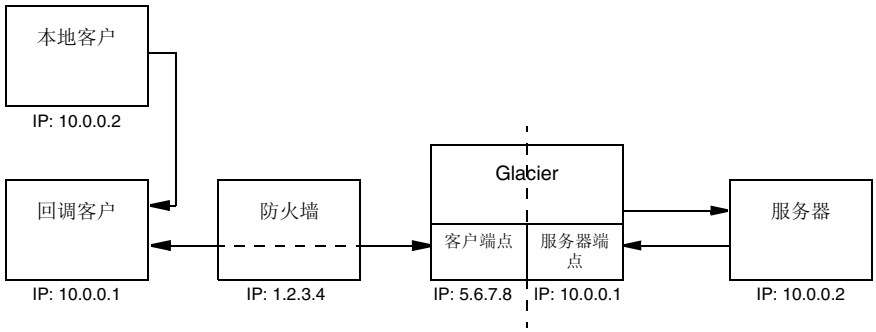

图 24.7. 支持回调和本地请求

现在，如果回调客户是使用单个对象适配器处理回调请求和本地请求，那么这个对象适配器创建的任何代理都将含有应用的本地端点，以及路由器的服务器端点。你可以想象，这会造成一些微妙的问题:

1. 当本地客户试图通过其中某一个代理建立与回调客户的连接时，它可能会任意地选择先尝试路由器的某个服务器端点。因为路由器的服务器端点使用的是同一个网络中的地址，本地客户会尝试在局域网上建立连接，可能的结果有两种：与这些端点建立连接的尝试失败了，在这种情况下，它们会被略过，继续尝试真实的本地端点；更糟的可能是，在局

域网中，某个端点碰巧是有效的，在这种情况下本地客户会连接到某个未知的服务器。

2. 服务器在为了发出回调请求而尝试建立与路由器的局域连接时，可能会遇到类似的问题。

这个问题的解决方案是专门用一个对象适配器来处理回调请求，用另一个来为本地客户服务。

要以这种方式使用多个对象适配器，你不能定义 `Ice.Default.Router` 属性 (参见 24.4.5 节)。假定我们的回调对象适配器名为 `CallbackAdapter`，我们用下面的属性来为适配器配置路由器:

```
CallbackAdapter.Router=Glacier/router -h 5.6.7.8 -p 8000
```

当然，在移除了 `Ice.Default.Router` 属性之后，我们的客户代理不再会自动配置路由器。要解决这个问题，有两种做法:

1. 在我们需要进行路由的代理上调用 `ice_router` 操作。
2. 使用多个通信器。例如，创建一个配置了 `Ice.Default.Router` 代理的通信器，并使用这个通信器来创建回调对象适配器和所有有路由的代理。然后创建另一个通信器，用于本地对象适配器。

24.4.8 使用多个路由器

客户并没有同时只能使用一个路由器的限制: 代理操作 `ice_router` 允许客户按照需要配置它的有路由代理。就回调而言， Ice 核心被设计成允许单个对象适配器接受来自多个路由器的回调，但 Glacier 实现目前不支持这种能力。因此，必须为每个能把回调请求转发给应用的路由器创建一个回调对象适配器。

24.4.9 例子

`demo/Ice/callback` 例子能用于测试通过Glacier进行的回调。这个例子的配置文件已经含有与 Glacier 路由器的使用有关的属性定义，但你可能需要编辑这个文件，启用某些在缺省情况下放在注释里的属性。更多的信息，参见配置文件中的注释。

24.5 Glacier 启动器

一个 Glacier 路由器有能力把来自多个客户的请求路由到多个服务器，但前提是不需要进行回调 (参见 24.3 节)。如果必须进行回调，每个客户都

必须使用它自己的 Glacier 路由器。如图 24.8 所示， Ice 提供了[2] Glacier “启动器”来随需启动路由器:

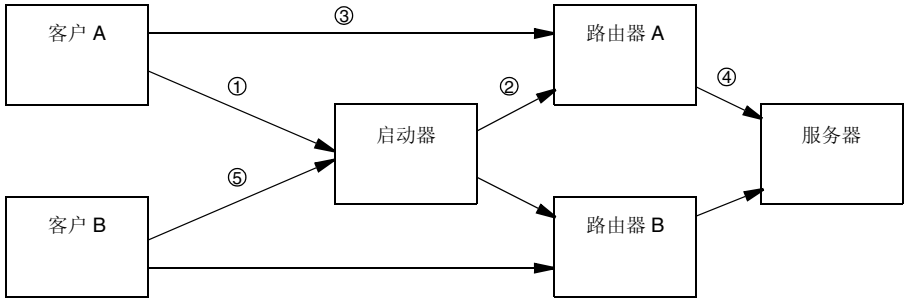

图 24.8. 使用 Glacier 启动器

1. 客户 A 向启动器发出一个请求，后者在周知的端点上侦听。
2. 启动器派生一个新路由器实例，路由器 A，并把这个路由器的代理返回给客户。
3. 客户 A 为它的代理配置路由器，并通过有路由的代理发出调用。从这时起，客户 A 将只与路由器 A、而不是启动器交互。
4. 路由器 A 把请求转发给服务器。
5. 客户 B 重复这个过程。

你可以看到，要解决 “每个客户一个路由器” 的问题， Glacier 启动器是一种非常方便的途径。关于安全方面的考虑，参见 24.6 节。

24.5.1 配置启动器

启动器是一个常规的 Ice 服务器，支持许多配置属性（参见附录 C）。最低限度的配置如下所示:

2. 目前只有在 Unix 平台上才支持 Glacier 启动器。

```
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
```

`Glacier.Starter.Endpoints` 属性定义启动器的周知端点。这是客户用在其启动器代理中的端点 (参见 24.5.3 节)。

`Glacier.Starter.RouterPath` 属性提供路由器可执行程序的路径名。如果这个属性没有定义，缺省值是 `glacierrouter`，意味着路由器可执行程序必须位于启动器的可执行程序搜索路径中。

最有意思的属性是 `Glacier.Starter.PropertiesOverride`。路由器是由继承了路由器的配置属性的启动器启动的，这也是我们为什么要在启动器的配置中包括路由器的属性的原因。但是，你可能会想要替换启动器的某个属性，你可能会想要专门为路由器增加一些特性，而不把它们应用于启动器。 `PropertiesOverride` 属性能够让你做到这一点。

在这个例子中，启动器定义了 `Ice.ServerIdleTime` 属性，这个属性迫使服务器在连接空闲了一段时间之后得体地终止。我们没有在启动器上设置这个属性，因为我们想让它保持连接，用于新的客户请求。但是，这对路由器来说是一个非常有用的属性，因为它避免了累积不活动的、其客户已终止或崩溃的路由器进程。因此，我们使用了 `PropertiesOverride` 属性来专门为路由器定义 `Ice.ServerIdleTime` 属性。

最后，我们定义了路由器的端点。这些端点没有使用固定的端点，因为每个路由器都由同一个客户启动，因此必须使用系统分配的端点。如果我们使用的是固定的端点，那么同时就只有一个路由器能运行。

24.5.2 启动启动器

假定 24.5.1 节给出的配置属性存储在名为 `config` 的文件中，你可以用下面的命令来启动启动器:

```
$ glacierstarter --Ice.Config=config
```

启动器也可以作为 Unix 看守或 Win32 服务运行。更多信息，参见 10.3.2 节。

24.5.3 使用启动器

客户可以调用 Glacier::Starter 接口上的 startRouter 操作来获得路由器。

C++ 例子

下面的例子代码说明了 C++ 客户怎样获得路由器。

```
#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);
```

这段代码首先创建启动器的代理。这个代理中的端点与我们用 24.5.1 节描述的 Glacier.Starter.Endpoints 属性配置的端点是匹配的。

```
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);
```

接下来，我们调用 startRouter 操作，传入用户和密码参数的值 (关于启动器安全性的更多信息，参见 24.6 节)。

```
Ice::ByteSeq privateKey, publicKey, routerCert;

Glacier::RouterPrx router =
    starter->startRouter("user", "passwd", privateKey, publicKey,
                         routerCert);
```

最后，我们用从启动器那里接收到的路由器代理配置通信器和对象适配器。

```
communicator->setDefaultRouter(router);
adapter->addRouter(router);
```

至此，通信器已经准备好创建有路由的代理，而对象适配器也已准备好接收回调请求。

Java 例子

下面的例子代码说明了 Java 客户怎样获取路由器。

```
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);
```

这段代码首先创建启动器的代理。这个代理中的端点与我们用 24.5.1 节描述的 `Glacier.Starter.Endpoints` 属性配置的端点是匹配的。

```
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);
```

接下来，我们调用 `startRouter` 操作，传入用户和密码参数的值 (关于启动器安全性的更多信息，参见 24.6 节)。

```
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);
```

最后，我们用从启动器那里接收到的路由器代理配置通信器和对象适配器。

```
communicator.setDefaultRouter(router);
adapter.addRouter(router);
```

至此，通信器已经准备好创建有路由的代理，而对象适配器也已准备好接收回调请求。

24.6 Glacier 的安全性

作为防火墙，Glacier 路由器是通往私有网络的一道门，而在大多数情况下，这道门应该有一把好锁。很明显，第一步是要把 SSL 用于路由器控制、路由器客户，以及启动器端点。这使得你能够保护消息通信，并限制对 “拥有适当证书的客户” 的访问 (参见第 23 章)。但 Glacier 启动器还进一步加强了安全性：它能够进行更严格的访问控制。我们在 24.5.3 节给出的示例代码中看到过这些能力的迹象，我们将在这一节深入地探索它们。

24.6.1 访问控制

对于有些应用而言， SSL 的验证能力并不足以限制对防火墙的访问。SSL 握手的证书检验阶段会核实用户的确是他所自称的人，但我们怎么知道他有权通过防火墙？

通过使用一种访问控制设施， Glacier 启动器解决了这个问题。如 24.5.3 节所示， `startRouter` 操作的前两个参数是用户名和密码。启动器会在为客户启动路由器之前校验这两个参数 (客户用 “明文” 发送这些参数是安全的，因为客户是通过 SSL 连接到启动器的)。

启动器支持两种形式的用户名 / 校验：基于文件的访问控制列表，或定制的校验器实现。特定的启动器使用那种校验形式，要由配置属性来决定；缺省情况使用的是基于文件的校验。

Crypt 文件

和 Unix 系统上的 passwd 文件类型，基于文件的访问列表使用了 crypt 算法来保护密码。这个文件的每一行都必须含有一个用户名和一个密码，用空格分开。密码必须是由 crypt 编码的、 13 字节长的串。例如，test/Glacier/starter/passwords 中的测试套件所用的文件含有这样一行内容:

```
dummy xxMqsnnDcK8tw
```

你可以用 OpenSSL 提供的 **openssl** 实用程序来获得 crypt 编码:

```
$ openssl passwd
```

这个程序会提示你输入密码，然后生成用 crypt 编码过的版本，供你用在密码文件中。

Glacier.Starter.CryptPasswords 属性定义这个文件的名字。如果没有指定，缺省的名字是 passwords。

许可校验器

有特殊要求的应用可以实现 Glacier::PermissionsVerifier 接口，在程序中控制对启动器的访问。这种做法在下面的情况下特别有用: 账户信息仓库已经存在 (比如一个 LDAP 目录)，在这样的情况下，在另一个文件中重复那些信息既繁琐、又易错。

这个接口的 Slice 定义 只有一个操作:

```
module Glacier {
interface PermissionsVerifier {
    nonmutating
    bool checkPermissions(string userId, string password,
                          out string reason);
};
};
```

启动器调用校验器对象的 checkPermissions，把先前传入 startRouter 的用户和密码参数传给它 。如果参数是有效的，操作必须返回真，否则返回假。如果操作返回假，可以在输出参数中指出原因。

要为启动器配置定制的校验器，把配置属性 Glacier.Starter.PermissionsVerifier 设成校验器对象的代理。因为用户名和密码是用明文发送的，所以把 SSL 端点用于校验器是一种明智的做法。

24.6.2 过滤

Glacier 路由器能够根据对象标识来过滤请求，这有助于确保客户无法访问它不该访问的对象。

回想一下，Ice::Identity 类型含有两个串成员：category 和 name。你可以用一个有效标识范畴列表来配置路由器，在这种情况下，路由器将只对那些在这些范畴中的请求进行路由。配置属性 Glacier.Router.AllowCategories 用于指定范畴列表：

```
Glacier.Router.AllowCategories=cat1 cat2
```

在过滤功能中启动器也扮演了一个角色。配置属性 Glacier.Starter.AddUserToAllowCategories 用于控制启动器在启动某个路由器时，是否要把校验过的用户名添加到路由器的有效范畴列表中。这个特性能用于这样的应用：各个 Ice 对象由一个特定的客户创建，而且不能让其他客户访问。因为启动器会为每个客户创建一个新路由器，要把客户特有的范畴（也就是用户名）提供给路由器，启动器处在最佳位置上。很自然，这种策略假定服务器会这样进行协作：使用用户名来作为它为客户创建的 Ice 对象的标识范畴。下面给出的属性演示了怎样为了过滤而对启动器和路由器进行配置：

```
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
```

使用这种配置，为用户 duncanfoo 创建的路由器将允许针对标识范畴 Factory 和 duncanfoo 的请求通过。如果客户发送的请求针对的是其他任何标识范畴，它就会收到 Ice::ObjectNotExistException。

24.6.3 保护路由器

除了前面讨论的访问控制和过滤特性，Glacier 启动器还采取了进一步的步骤来保护路由器。如果启动器为使用 IceSSL 做了配置，同时成功校验了某个新客户，启动器会为客户生成一个密钥对，并为路由器生成一个密钥对。启动器把客户的公钥提供给路由器，把路由器的公钥提供给客户。随后，路由器和客户用它们的对端的公钥来分别配置自己的 IceSSL 插件，以确保确实是在与该对端建立连接。

下面给出了在客户中适当配置 IceSSL 插件所需的代码。我们在其中包括了调用 startRouter 的步骤，尽管它们和 24.5.3 节给出的例子是相同的。这个例子的目的是演示怎样使用 startRouter 返回的三个 out 参数。

```
#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);
```

在调用了 `startRouter` 之后，客户会通过检查 IceSSL 配置属性来确定它是否被配置成要使用 IceSSL。如果两个属性都定义了，客户就会继续进行插件配置。

`startRouter` 返回的值被转换成 Base64 编码，这是 IceSSL 要求的格式:

```
    string privateKeyBase64 = IceUtil::Base64::encode(privateKey);
    string publicKeyBase64  = IceUtil::Base64::encode(publicKey);
    string routerCertString = IceUtil::Base64::encode(routerCert);
```

接下来，客户使用第 23 章描述的技术获得 IceSSL 插件的一个引用。

```
    Ice::PluginManagerPtr pluginManager =
        communicator->getPluginManager();
    Ice::PluginPtr plugin = pluginManager->getPlugin("IceSSL");
    IceSSL::PluginPtr sslPlugin =
        IceSSL::PluginPtr::dynamicCast(plugin);
```

客户随即手工配置 IceSSL 插件的服务器组件，让插件夹在 IceSSL 配置文件，然后用路由器的公钥创建证书校验器。

```
    sslPlugin->configure(IceSSL::Server);
    sslPlugin->setCertificateVerifier(
        IceSSL::ClientServer,
        sslPlugin->getSingleCertVerifier(routerCert));
```

最后，客户把它的新 RSA 密钥对传给插件，然后把路由器的公钥添加为受信证书。

```
    sslPlugin->setRSAKeysBase64(IceSSL::ClientServer,
                                privateKeyBase64,
                                publicKeyBase64);
    sslPlugin->addTrustedCertificateBase64(IceSSL::ClientServer,
                                           routerCertString);
```

24.7 总结

复杂的网络环境是生活的一部分。遗憾的是，保护企业网络所需付出的代价增加了应用的复杂性和管理开销。 Glacier 为 Ice 应用提供了低影响、高效而安全的路由器 - 防火墙，有助于使上述开销降到最低限度。

第 25 章
IceBox

25.1 本章综述

在这一章我们将介绍 IceBox，一种易于使用的 Ice 应用服务框架。25.2 节将对 IceBox 及使用它的好处进行综述。25.3 节介绍服务管理器，并描述了它在 IceBox 中的角色。25.4 节将向你介绍怎样编写和配置 IceBox 服务，25.5 节把所有步骤汇总在一起。

25.2 引言

Service Configurator 模式 [7] 这种技术可用于配置服务、并把对它们的管理集中在一起。用实践中的术语说，这意味着服务被开发成可动态加载的组件，并且可以按照需要，以任何一种组合方式配置进一个通用的 “超级服务器”中。IceBox 是 Service Configurator 模式的一种用于 Ice 服务的实现。

一个通用的 IceBox 服务器取代了你通常编写的整体式 Ice 服务器。你通过属性为 IceBox 服务配置它负责加载和管理的应用特有的服务，并且可以对它进行远地管理。使用这种架构有若干好处：

- 你可以对同一个IceBox服务器加载的多个服务进行配置，利用Ice的并置优化。例如，如果一个服务是另一个服务的客户，而这两个服务驻留在同一个 IceBox 服务器中，那么在它们之间进行的调用就可以优化。

- 要把多个服务组合成一个应用，可以通过配置、而不是编译和链接来完成。这解除了服务和服务器的耦合，允许你按照需要组合服务或分离服务。
- 在 Java Virtual Machine (JVM) 的单个实例中，可以有多个活动的 Java 服务。与在多个 JVM 中分别运行一个整体式的服务器相比，上面的这种做法能够节省操作系统资源。
- 各个服务要实现 IceBox 服务接口，为开发者提供了一个公共框架，以及一个集中式的管理设施。
- 在服务器激活和部署服务 IcePack 中集成了 IceBox 支持 (参见第 20 章)。

25.3 服务管理器

除了应用服务所支持的对象， IceBox 服务器还支持一个实现了 `IceBox::ServiceManager` 接口 (参见附录 B) 的管理对象。这个对象负责加载和初始化服务，并执行客户所要求的管理动作。为了保护这个对象的端点，不让它们受到未经授权的访问， IceBox 服务器还会为这个对象创建一个对象适配器。

目前， `ServiceManager` 接口的管理能力相当有限：该接口只支持 `shutdown`，用于终止服务、关闭 IceBox 服务器。在未来的版本中可能会增加更多的管理特性。

25.3.1 端点配置

服务管理器的对象适配器的端点是用 `IceBox.ServiceManager.Endpoints` 配置属性定义的：

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
```

因为有恶意的客户可能会向服务管理器发动 “拒绝服务” 攻击，明智的做法是使用 SSL 端点 (参见第 23 章)，或用适当的防火墙配置来保护这些端点，或者两者都使用。

25.3.2 客户配置

需要访问服务管理器的管理功能的客户可以使用 25.3.1 节描述的属性所定义的端点来创建代理。服务管理器对象的缺省标识是 `ServiceManager`，但你可以使用

IceBox.ServiceManager.Identity 属性来改变它 (参见附录 C)。因此，使用缺省标识和 25.3.1 节的端点的代理可以构造如下:

```
ServiceManager.Proxy=ServiceManager:tcp -p 10000
```

25.3.3 管理实用程序

随同 IceBox 提供了一个管理实用程序。我们已经说过， ServiceManager 接口目前的能力很有限，所以不奇怪，这个管理实用程序的功能也很有限。IceBox 提供了这个实用程序的 C++ 和 Java 实现，其使用方法是一样的:

```
Usage: iceboxadmin [options] [command...]
Options:
-h, --help           Show this message.
-v, --version        Display the Ice version.

Commands:
shutdown             Shutdown the server.
```

这个 C++ 实用程序名为 iceboxadmin，而 Java 实用程序由 IceBox.Admin 类表示。它们都使用了 25.3.1 节描述的 IceBox.ServiceManager.Endpoints 属性来创建服务管理器对象的代理。在典型情况下，在启动这个实用程序时使用的配置文件与 IceBox 服务器的相同 (参见 25.5 节)，但这并非是强制性的。

25.4 开发服务

要编写 IceBox 服务，需要实现某个 IceBox 服务接口。我们在这一节给出的 C++ 和 Java 例子实现了 IceBox::Service:

```
module IceBox {
local interface ServiceBase {
    void stop();
};

local interface Service extends ServiceBase {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args)
        throws FailureException;
};
};
```

你可以看到，服务只需实现两个操作：start 和 stop。这些操作由服务管理器调用； start 在服务加载后被调用，而 stop 在服务关闭时被调用。

服务可以在 start 操作中初始化自身；这通常包括创建对象适配器和 servant。 name 和 args 参数提供了来自服务的配置 (参见 25.4.3 节) 的信息，而 communicator 参数是服务管理器为供服务使用而创建的 Ice::Communicator 对象。取决于服务的配置，这个通信器实例可能会由同一个 IceBox 服务器中的其他服务共享，因此，你需要注意确保像对象适配器这样的对象的名字是唯一的。

stop 操作必须回收服务所使用的任何资源。一般而言，服务会解除其对象适配器的激活，可能还需要调用对象适配器的 waitForDeactivate，以确保在继续进行清理之前，所有待处理的请求都已完成。服务管理器会负责销毁通信器。

由于下面的原因，这些接口被声明为 local 接口：它们代表的是服务管理器和服务之间的合约，而不是要供远地客户使用。服务与远地客户进行的任何交互都要通过服务所创建的 servant 来完成。

25.4.1 C++ 服务例子

我们在这里给出的例子取自 Ice 包中提供的 demo/IceBox/hello 实例程序。

我们的服务的类定义相当直截了当，但有一些方面值得一提：

```
#include <IceBox/IceBox.h>

#if defined(_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;
};
```

首先，我们包括了 IceBox 头文件，以使我们能从 `IceBox::Service` 派生我们的实现。

其次，那些预处理器定义是必需的，因为在 Windows 上，这个服务驻留在一个 Dynamic Link Library (DLL) 中，因此我们需要输出这个类，让服务管理器能适当地加载它。

成员定义同样直截了当：

```
#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();
}
```

你可能会对我们定义的 `create` 函数感到奇怪。这是 C++ IceBox 服务的*进入点*；也就是说，服务管理器用这个函数来获得服务的实例，所以它必须具有特定的型构。函数名并不重要，但这个函数应该接受一个参数，类型是 `Ice::CommunicatorPtr`，并且返回一个指向 `IceBox::Service`[1]

的指针。在这里，我们简单地返回了 HelloServiceI 的一个新实例。关于进入点的更多信息，参见 25.4.3 节。

start 方法用和服务名相同的名字创建一个对象适配器，激活一个类型为 HelloI (没有给出) 的 servant，并激活对象适配器。 stop 简单地解除对象适配器的激活。

这显然是一个微不足道的服务，你的兴趣很可能要大得多，但这个例子确实向你演示了要编写 IceBox 服务是多么容易。在把这些代码编译进共享库或 DLL 中之后，你可以像 25.4.3 节描述的那样把它配置进 IceBox 服务器中。

25.4.2 Java 服务例子

与上一节给出的 C++ 例子一样，你可以在 Ice 包的 demo/IceBox/hello 目录中找到这个 Java 例子的完整源码。我们的服务的类定义是:

```
public class HelloServiceI extends Ice.LocalObjectImpl
    implements IceBox.Service
{
    public void
    start(String name,
          Ice.Communicator communicator,
          String[] args)
        throws IceBox.FailureException
    {
        _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;
}
```

1. C 链接方式的函数不能返回对象类型，比如智能指针，因此进入点函数必须返回常规的指针值。

首先要注意，我们的类扩展了 `Ice.LocalObjectImpl`。这是 Ice 开发者必须实现本地接口的极少数情况之一 (其他常见的情况是对象工厂和 servant 定位器)。

`start` 方法用和服务名相同的名字创建一个对象适配器，激活一个类型为 `HelloI` (没有给出) 的 servant，并激活对象适配器。 `stop` 简单地解除对象适配器的激活。

服务管理器要求服务的实现拥有缺省构造器。这是 Java IceBox 服务的*进入点*；也就是说，服务管理器会动态加载服务实现类，并调用构造器来获取服务的实例。

这显然是一个微不足道的服务，你的兴趣很可能要大得多，但这个例子确实向你演示了要编写 IceBox 服务是多么容易。在编译了服务实现类之后，你可以像 25.4.3 节描述的那样把它配置进 IceBox 服务器中。

25.4.3 配置服务器

要把服务配置进 IceBox 服务器中，只需使用一个属性。这个属性的用途有好几个：它定义服务的名字，它向服务管理器提供服务进入点，它还定义用于服务的属性和参数。

下面是这个属性的格式：

```
IceBox.Service.name=entry_point [args]
```

属性键的 *name* 部分是服务名。这个名字会传给服务的 `start` 操作，在配置进同一个 IceBox 服务器的所有服务中必须是唯一的。用不同的名字加载同一个服务的两个或多个实例是可能的，尽管你很少需要这么做。

属性值的第一个参数用于指定进入点。对于 C++ 服务，其形式必须是 `library:symbol`， `library` 是服务共享库或 DLL 的简单名字，而 `symbol` 是进入点函数的名字。我们所说的 “简单名字” 是指，没有任何与特定平台有关的前缀或扩展名的名字；服务管理器会根据平台附加适当的前缀或扩展名。例如，简单名字 `MyService` 在 Windows 上对应的是名为 `MyService.dll` 的 DLL，在 Linux 上对应的是名为 `libMyService.so` 的共享库。共享库或 DLL 所在的目录必须出现在 Windows 上的 `PATH` 或 POSIX 系统上的 `LD_LIBRARY_PATH` 中。

对于 Java 服务，进入点就是服务实现类的完整类名 (包括任何 package)。这个类必须位于 IceBox 服务器的类路径上。

跟在 `entry_point` 后面的任何参数都会被检查。如果某个参数的形式是 `--name=value`，它就会被解释为属性定义，将会出现在传给服务的 `start` 操作的通信器的属性集中。这些参数将被移除，剩下的参数会放在 `args` 参数中传给 `start` 操作。

C++ 例子

下面是对 25.4.1 节的 C++ 例子进行配置的一个例子:

```
IceBox.Service.Hello=HelloService:create \
    --Ice.Trace.Network=1 hello there
```

使用这个配置，将会创建一个名为 `Hello` 的服务。这个服务应当位于 `HelloService.dll`（Windows）或 `libHelloService.so`（Linux）中，进入点函数 `create` 会被调用，创建服务的一个实例。参数 `--Ice.Trace.Network=1` 被转换成属性定义，参数 `hello` 和 `there` 则会成为传给 `start` 方法的 `args` 序列参数中的两个元素。

Java 例子

下面是对 25.4.2 节的 Java 例子进行配置的一个例子:

```
IceBox.Service.Hello=HelloServiceI \
    --Ice.Trace.Network=1 hello there
```

使用这个配置，将会创建一个名为 `Hello` 的服务。这个服务应当位于 `HelloServiceI` 类中。参数 `--Ice.Trace.Network=1` 被转换成属性定义，参数 `hello` 和 `there` 则会成为传给 `start` 方法的 `args` 序列参数中的两个元素。

Freeze 配置

Freeze 服务支持一个额外的配置属性，用于定义服务管理器创建 Freeze 数据库环境所用的目录的路径名:

`IceBox.DBEnvName.`*`name`*`=`*`path`*

属性键的 *`name`* 部分是服务名。

共享一个通信器

你可以用下面的属性对服务管理器进行配置，让它的所有服务共享一个通信器实例:

```
IceBox.UseSharedCommunicator=1
```

如果没有定义这个属性，缺省的行为是为每个服务创建一个新通信器。但是，如果你想要对服务进行并置优化，就必须使用一个共享的通信器实例。

加载服务

在缺省情况下，服务管理器加载服务的次序是不确定的，也就是说，同一个 IceBox 服务器中的各个服务不应该相互依赖。如果多个服务必须按照特定的次序加载，可以使用 IceBox.LoadOrder 属性:

```
IceBox.LoadOrder=Service1,Service2
```

在这个例子中， Service1 首先加载，接着是 Service2。剩下的其他服务在 Service2 之后以不确定的次序加载。在 IceBox.LoadOrder 中提到的每个服务都必须有相匹配的 IceBox.Service 属性。

25.4.4 Freeze 服务

IceBox 为使用 Freeze (参见第 21 章) 的服务提供了另一个接口 :

```
module IceBox {
local interface FreezeService extends ServiceBase {
    void start(string name,
               Ice::Communicator communicator,
               Ice::StringSeq args,
               Freeze::DBEnvironment dbEnv)
        throws FailureException;
};
};
```

通过实现 FreezeService、而不是 Service，你隐含地告诉服务管理器，它应该为这个服务创建一个 Freeze 数据库环境，并把它提供给 start 操作。如 25.4.3 节所述，数据路环境目录的路径名使用一个配置属性指定。服务管理器拥有这个数据库环境，并会负责在调用服务的 stop 之后销毁它。

25.5 启动 IceBox

把我们在前面讨论的内容结合在一起，我们现在可以配置并启动 IceBox 服务器了。

25.5.1 启动 C++ 服务器

下面是用于我们的 C++ 服务例子的配置文件:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
IceBox.Service.Hello=HelloService:create
Hello.Endpoints=tcp -p 10001
```

注意，我们为 Hello 服务创建的对象适配器定义了一个端点。

假定这些属性位于名为 config 的配置文件中，我们可以这样启动 C++ IceBox 服务器:

```
$ icebox --Ice.Config=config
```

25.5.2 启动 Java 服务器

我们的 Java 配置和 C++ 版本几乎是一样的，除了进入点的指定:

```
IceBox.ServiceManager.Endpoints=tcp -p 10000
IceBox.Service.Hello=HelloServiceI
Hello.Endpoints=tcp -p 10001
```

假定这些属性位于名为 config 的配置文件中，我们可以这样启动 Java IceBox 服务器:

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 初始化失败

在启动时， IceBox 服务器会检查它的配置，查找所有具有前缀 IceBox.Service 的属性，并初始化每个服务。如果某个服务的初始化失败了， IceBox 服务器会调用任何已初始化的服务的 stop 操作，报告错误并终止。

25.6 总结

IceBox 提供了一种让人振奋的视角变化：开发者专注于编写服务，而不是应用。应用的定义变了；使用 IceBox，应用变成了一组离散的服务，其组合将由配置动态决定，而不是由链接器静态决定。

第 26 章
IceStorm

26.1 本章综述

在这一章我们将介绍 IceStorm，一个高效的用于 Ice 应用的发布 / 订阅服务。 26.2 节简要介绍 IceStorm， 26.3 节讨论一些基本的 IceStorm 概念。 26.4 节对 IceStorm 的 Slice 接口进行综述， 26.5 节给出了 IceStorm 应用的一个例子。 26.6 节描述 IceStorm 的管理工具， 26.7 节将讨论联盟这个主题。最后， IceStorm 的配置问题将在 26.9 节讨论。

26.2 引言

应用常常需要把信息分发给多个接收者。例如，假定我们在开发一个气候监测应用，用于收集来自气象塔的观测数据，比如风速和温度，并把它们周期性地发布给气候监视站。我们一开始考虑使用图 26.1 中的架构。

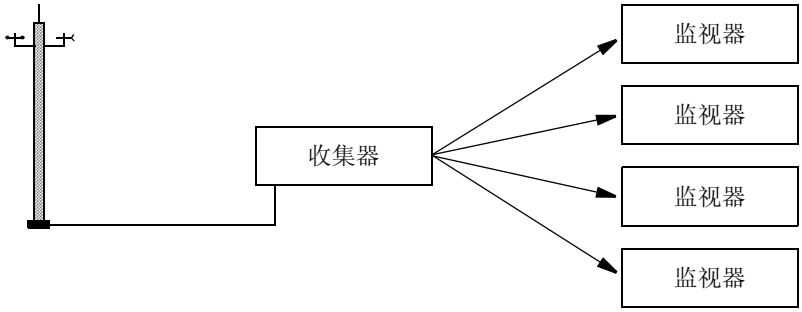

图 26.1. 气候监视应用的初始设计

但这种构架有一个很大的缺点，它把收集器及其监视器紧密地耦合在一起，要求收集器实现去管理监视器的注册细节、观测数据递送，以及出错恢复，从而变得无谓的复杂化。如图 26.2 所示，我们可以把 IceStorm 结合进我们的应用中，使我们摆脱这些琐碎的工作。

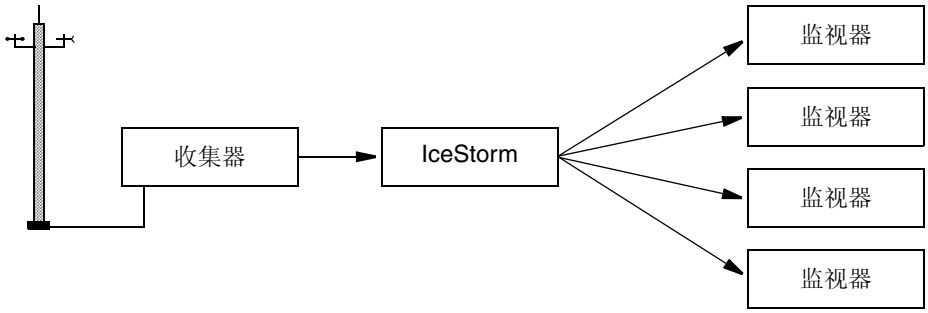

图 26.2. 使用 IceStorm 的气候监视应用

IceStorm 使收集器与监视器解除耦合，极大地简化了收集器实现。作为发布 / 订阅服务， IceStorm 充当的是收集器 (发布者) 与监视器 (订阅者) 之间的 “中间人”，它提供了若干好处:

- 当收集器准备好分发一组新的观测数据时，它会向 IceStorm 服务器发出一个请求。 IceStorm 服务器会负责把请求递送给监视器，包括处理由于订阅者的行为有问题或订阅者不存在所造成的异常。收集器不再需要关注它的监视器，甚至不需要知道此时是否有监视器。
- 与此类似，监视器会与 IceStorm 服务器交互，完成像订阅和取消订阅这样的任务，从而让收集器专注于它自己的应用特有的职责，而不是管理上的琐事。
- 要把 IceStorm 结合进来，收集器和监视器应用所需做出的改动非常少。

26.3 概念

这一节将讨论几个概念，要想理解 IceStorm 的能力，这些概念十分重要。

26.3.1 消息

IceStorm 的*消息*是强类型的，由对某个 Slice 操作的调用代表：操作名标识消息的类型，操作参数定义消息内容。要发布消息，可以按普通的方式调用某个 IceStorm 代理上的操作。与此类似，订阅者会像收到常规的向上调用 （upcall）一样收到消息。所以 IceStorm 的消息递送使用的是“推”模式；轮询模式不支持。

26.3.2 主题

应用要通过订阅某个*主题*（topic）来表明自己有兴趣接收某些消息。IceStorm 服务器能够支持任意数量的主题，这些主题是动态创建的，通过唯一的名字来区分。每个主题都可以有多个发布者和订阅者。

在本质上，一个主题和一个由应用定义的 Slice 接口是等价的：该接口的各操作定义该主题所支持的消息的类型。发布者使用主题接口的代理来发送它的消息，而订阅者则为了接收消息而实现主题接口（或派生自该主题接口的接口）。这和传统的客户 - 服务器方式没有不同：发布者和订阅者就像是在直接通信；接口代表的是客户 （发布者）和服务器 （订阅者）之间的合约，但 IceStorm 会透明地把每个消息转发给多个接收者。

IceStorm 不会对发布者和接收者使用的是否是兼容的接口进行检验，因此应用必须确保正确地使用主题。

26.3.3 单向语义

IceStorm 消息具有单向语义 (参见 2.2.2 节)，因此发布者无法接收来自其订阅者的答复。很自然，所有的 Ice 传输机制 (TCP、 SSL，以及 UDP) 都能用于发布和接收消息。

26.3.4 联盟

使用 IceStorm，你可以构造主题图 （topic of graphs），也称为联盟（federation）。主题图是通过在主题间创建链接而形成的，*链接*是一个主题与另一个主题的无向关联。每个链接都有*成本*（cost），可能会限制消息在该链接上递送 (参见 26.7.2 节)。在某个主题上发布的消息也会在该主题的所有这样的链接上发布：消息成本没有超过链接成本。

一旦某条消息在某个链接上发布，收到该消息的主题就会把它发布给自己的订阅者，但不会把它发布给自己的任何链接。换句话说，在某个联盟中，从最初的主题开始计算， IceStorm 消息的传播最多只有一跳 (参见 26.7.1 节)。

图 26.3 给出了主题联盟的一个例子。如箭头所示，主题 T_1 具有与 T_2 和 T_3 的链接。订阅者 S_1 和 S_2 会收到在 T_2 上发布的所有消息，以及在 T_1 上发布的消息。订阅者 S_3 只会收到来自 T_1 的消息，而 S_4 会收到来自 T_3 和 T_1 的消息。

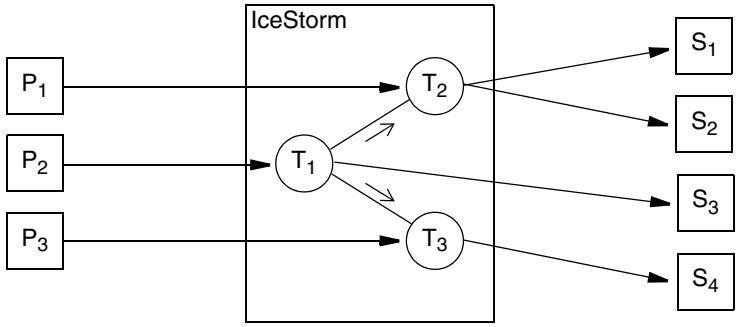

图 26.3. 主题联盟

IceStorm 不会设法防止订阅者收到重复的消息。例如，如果某个订阅者既订阅了 T_2，也订阅了 T_3，那么在 T_1 上每发布一个消息，它就会收到两个请求。

26.3.5 服务质量

IceStorm 允许每个订阅者指定它自己的*服务质量*(QoS) 参数，对其消息的递送施加影响。服务质量参数由一个含有名 - 值对的词典表示。 26.8 节描述了所支持的 QoS 参数。

26.3.6 持久

IceStorm 拥有一个数据库，里面维护的是关于其主题和链接的信息。但是，通过 IceStorm 发送的消息不会被持久地存储，而是会在递送给主题目前的订阅者集之后，马上被丢弃。如果在把消息递送给某个订阅者的过程中发生错误， IceStorm 不会为该订阅者进行消息排队。

26.3.7 订阅者出错

因为 IceStorm 消息是采用单向语义递送的， IceStorm 只能检测到连接或超时错误。如果在把消息递送给订阅者的过程中， IceStorm 遇到这样的错误，该订阅者就会立刻被解除与该消息对应的主题的订阅。

慢速的订阅者也可能会造成问题。一旦消息发布， IceStorm 会立刻尝试递送它们: IceStorm 发布者对象可能会一收到消息就嵌套地调用主题的订阅者。因此，如果订阅者消费消息的速度没有发布的速度快，它就可能造成 IceStorm 中的线程数不断增加。如果线程数到达了线程池的最大尺寸，那么就不再能发布新消息了。关于 Ice 线程池的更多信息，参见第 15 章。

26.4 IceStorm 接口综述

这一节简要地介绍组成 IceStorm 服务的各个 Slice 接口。 Slice 文档见附录 B。

26.4.1 TopicManager

`TopicManager` 是一个单体对象，充当的是 `Topic` 对象的工厂和仓库。它的接口和相关类型如下所示:

```
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();
};
};
```

create 操作用于创建一个新主题，该主题必须具有唯一的名字。retrieve 操作允许客户获取某个已有主题的代理，retrieveAll 返回一个词典，内含所有已有的主题。

26.4.2 Topic

Topic 接口表示一个主题，它提供了若干管理操作，可用于配置链接、管理订阅者。

```
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();
};
};
```

getName 操作返回的是指派给该主题的名字，而 getPublisher 操作返回的是该主题的发布者对象的代理 (参见 26.5.2 节)。

subscribe 操作把一个订阅者的代理添加到主题中；如果已经有一个订阅者代理具有相同的对象标识，新的订阅者代理就会取代旧的。 unsubscribe 操作从主题中移除一个订阅者。

要创建与另一个主题的链接，要使用 link 操作；如果与给定的主题已经有链接，就会引发 LinkExists 异常。要销毁链接，使用 unlink 操作。

最后， destroy 操作用于永久性地销毁主题。

26.5 使用 IceStorm

在这一节，我们将扩展 26.2 节的气候监测例子，演示怎样就某个主题创建、订阅和发布消息。在我们的例子中，我们将使用下面的 Slice 定义:

```
struct Measurement {
    string tower; // tower id
    float windSpeed; // knots
    short windDirection; // degrees
    float temperature; // degrees Celsius
};

interface Monitor {
    void report(Measurement m);
};
```

Monitor 是我们的主题接口。为简单起见，它只定义了一个操作 report，其唯一的参数是一个 Measurement 结构。

26.5.1 实现发布订阅者

我们很容易对我们的收集器应用的实现进行总结:

1. 获取 TopicManager 的一个代理。这是 IceStorm 的主要对象，发布者和订阅者都会使用它。

2. 获取 Weather 主题的一个代理——如果主题不存在，就创建它，否则就获取已有主题的代理。
3. 获取 Weather 主题的 “发布者对象” 的代理。这个代理的用途是发布消息，因此会被窄化成主题接口 (Monitor)。
4. 收集并报告观测数据。

在下面的小节里，我们将给出用 C++ 和 Java 编写的收集器实现。

C++ 例子

和往常一样，我们的 C++ 例子首先会包括必需的头文件。有意思的头文件是 IceStorm/IceStorm.h，这是根据 IceStorm Slice 定义生成的；还有 Monitor.h，含有根据上面给出的监视器定义生成的代码。

```
#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);
    }
    ...
}
```

在获取了主题管理器的代理之后，收集器会尝试取得主题。如果主题还不存在，收集器就会收到 NoSuchTopic 异常，并随即创建该主题。

```
IceStorm::TopicPrx topic;
try {
    topic = topicManager->retrieve("Weather");
}
catch (const IceStorm::NoSuchTopic&) {
    topic = topicManager->create("Weather");
}
```

下一步是创建发布者对象的代理，收集器会把它窄化成 Monitor 接口。

```
Ice::ObjectPrx pub = topic->getPublisher();
if (!pub->ice_isDatagram())
    pub = pub->ice_oneway();
MonitorPrx monitor = MonitorPrx::uncheckedCast(pub);
```

最后，收集器进入它的主循环，收集观测数据，并通过 IceStorm 发布者对象来发布它们。

```
while (true) {
    Measurement m = getMeasurement();
    monitor->report(m);
}
```

Java 例子

下面是等价的 Java 版本。

```
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);
    }
    ...
}
```

在获取了主题管理器的代理之后，收集器会尝试取得主题。如果主题还不存在，收集器就会收到 NoSuchTopic 异常，并随即创建该主题。

```
IceStorm.TopicPrx topic = null;
try {
    topic = topicManager.retrieve("Weather");
}
catch (IceStorm.NoSuchTopic ex) {
    topic = topicManager.create("Weather");
}
```

下一步是创建发布者对象的代理，收集器会把它窄化成 Monitor 接口。

```
Ice.ObjectPrx pub = topic.getPublisher();
if (!pub.ice_isDatagram())
    pub = pub.ice_oneway();
MonitorPrx monitor = MonitorPrxHelper.uncheckedCast(pub);
```

最后，收集器进入它的主循环，收集观测数据，并通过 IceStorm 发布者对象来发布它们。

```
while (true) {
    Measurement m = getMeasurement();
    monitor.report(m);
}
```

26.5.2 使用发布者对象

每个主题会为了发布消息的明确目的而创建一个发布者对象。这是一个特殊的对象：它实现了一个 Ice 接口，从而无需知道操作的类型，就可以接收和转发请求 (也就是， IceStorm 消息)。

从发布者的角度看，发布者对象显得像是应用特有的类型。实际上，发布者对象可以转发任何类型的请求；这带来了一定程度的危险：一个有问题的发布者可以使用 uncheckedCast 把发布者对象窄化为任何类型，并调用任何操作；发布者对象在一种不知情地状态下把这些请求转发给订阅者。如果发布者用不正确的类型发送了一个请求，发布者的 Ice run time 通常就会引发 OperationNotExistException。但是，因为订阅者是通过单向

调用接收消息的，所以无法向发布者对象发送响应来报告错误，因而发布者或订阅者都不会注意到出现了类型失配问题。简而言之， IceStorm 给开发者施加了这样的负担：确保发布者和订阅者在正确地使用 IceStorm。

尽管发布者并非一定要使用发布者对象的单向代理，这是我们建议的方式，理由有二：

- 它重申了这样一个事实：使用这个代理不可能收到回复；
- 它的效率更高，因为收集器的 Ice run time 不会把时间浪费在等待每个请求的答复上。

考虑到 IceStorm 递送消息的方式，第二个理由尤其重要。如 26.3.7 节所提到的，一旦消息发布，发布者对象会立刻尝试递送它，但不保证该消息能在发布调用返回之前到达所有订阅者。因此，如果在发布者对象上进行双向调用，发布者并不能获得任何好处，而且，由于 IceStorm 实现的一些具体细节，发布者还有耽搁自己的处理的危险。

注意，使用单向代理并非一定就会损失可靠性。如果发布者端点使用的是像 TCP 或 SSL 这样的可靠传输机制，单向消息总是会被可靠地递送给 IceStorm 服务器。但是，每个订阅者都可以自行选择用于消息递送的传输机制，因此发布者和 IceStorm 服务器之间所用的传输机制对订阅者没有影响[1]。

26.5.3 实现订阅者

我们的订阅者实现采取了以下步骤：

1. 获取 `TopicManager` 的一个代理，这是 IceStorm 的主要对象，发布者和订阅者都要使用它。
2. 创建一个对象适配器，充当我们的 `Monitor` servant 的宿主。
3. 实例化 `Monitor` servant，通过对象适配器激活它。
4. 订阅 `Weather` 主题。
5. 处理 `report` 消息，直到关闭为止。
6. 取消 `Weather` 主题的订阅。

在下面的小节里，我们将给出用 C++ 和 Java 编写的监视器实现。

1. 如果消息丢失率可以接受，可以把 UDP 用于订阅者的端点。但是， TopicManager 端点一般不会使用 UDP，因为发布者通常要确保消息被可靠地递送给 IceStorm，即使它们可能不会被可靠地递送给订阅者。

C++ 例子

我们的 C++ 监视器实现首先会包括必需的头文件。有意思的头文件是 IceStorm/IceStorm.h，这是根据 IceStorm Slice 定义生成的；还有 Monitor.h，含有根据 26.2 节的起始处给出的监视器定义生成的代码。

```
#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);
    ...
}
```

目前，我们的 Monitor servant 的实现相当简单。真实的实现可能会更新图形显示，或是把观测数据合并进正在进行的计算中。

```
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;
    }
};
```

在获取了主题管理器的代理之后，程序会创建一个对象适配器，实例化 Monitor servant，并激活它。

```
    Ice::ObjectAdapterPtr adapter =
        communicator->createObjectAdapter("MonitorAdapter");

    MonitorPtr monitor = new MonitorI;
    Ice::ObjectPrx proxy = adapter->addWithUUID(monitor);
```

接下来，监视器订阅主题。

```
    IceStorm::TopicPrx topic;
    try {
        topic = topicManager->retrieve("Weather");
        IceStorm::QoS qos;
        topic->subscribe(qos, proxy);
    }
    catch (const IceStorm::NoSuchTopic&) {
        // Error! No topic found!
        ...
    }
```

最后，监视器激活它的对象适配器，并等待被关闭。在 waitForShutdown 返回后，监视器会通过取消该主题的订阅来完成清理工作。

```
adapter->activate();
communicator->waitForShutdown();

topic->unsubscribe(proxy);
```

Java Example

下面是监视器的 Java 实现。

```
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);
    ...
}
```

目前，我们的 Monitor servant 的实现相当简单。真实的实现可能会更新图形显示，或是把观测数据合并进正在进行的计算中。

```
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");
    }
}
```

在获取了主题管理器的代理之后，程序会创建一个对象适配器，实例化 Monitor servant，并激活它。

```
    Monitor monitor = new MonitorI();
    Ice.ObjectPrx proxy = adapter.addWithUUID(monitor);
```

接下来，监视器订阅主题。

```
    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!
        ...
    }
```

最后，监视器激活它的对象适配器，并等待被关闭。在 waitForShutdown 返回后，监视器会通过取消该主题的订阅来完成清理工作。

```
    adapter.activate();
    communicator.waitForShutdown();

    topic.unsubscribe(proxy);
```

26.6 IceStorm 的管理

IceStorm 的管理工具是一个命令行程序，提供了对 IceStorm 服务器的管理控制。这个工具要求你按照 26.9.2 节所描述的方式指定 `IceStorm.TopicManager.Proxy` 属性。

这个工具支持以下命令行选项:

```
$ 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.
```

取决于命令行参数，这个工具有三种操作模式:

1. 如果指定了一个或多个 `-e` 选项，这个工具会执行给定的命令并退出。
2. 如果指定了一个或多个文件，这个工具会用 C 预处理器对每个文件进行预处理，执行每个文件中的命令，然后退出。
3. 否则，这个工具就会进入交互式会话。

help 命令会显示以下使用信息:

help

打印本消息。

exit, **quit**

退出本程序。

create TOPICS

添加 TOPICS。

destroy TOPICS

移除 TOPICS。

link FROM TO COST

采用给定的 COST 把 FROM 链接到 TO。

unlink FROM TO

解除 TO 和 FROM 的链接。

graph DATA COST

按照 DATA 中的描述，采用 COST 构造链接图。

list [TOPICS]

显示 TOPICS 或所有主题上的信息。

你可以用一个或多个主题名 (**TOPICS**) 来做许多命令的参数。含有空格的主题名、或者与某个命令关键字相同的主题名，必须放在单引号或双引号中。

关于 **graph** 命令的更多信息，参见 26.7.3 节。

26.7 主题联盟

在一个联盟中把多个主题链接在一起的能力为 IceStorm 应用提供了许多灵活性，而与链接相关联的 “成本” 的概念又允许应用以创造性的方式对消息流进行限制。 IceStorm 应用可以用附录 B 描述的 `TopicManager` 接口来完全控制主题联盟，从而使你能根据需要动态地创建和移除链接。但对于许多应用而言，主题图是静态的，因此可以用 26.6 节所讨论的管理工具来配置。

26.7.1 消息传播

IceStorm 消息的传播不会超过一个链接。例如，考虑图 26.4 中的主题图。

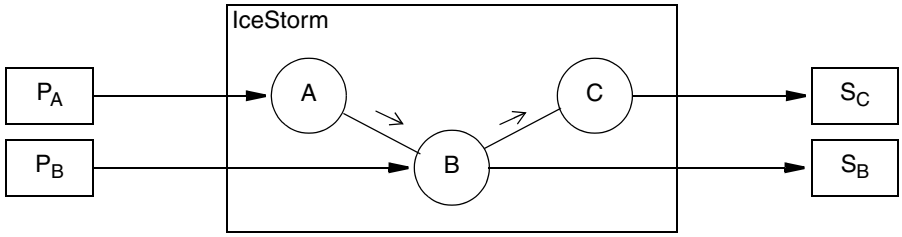

图 26.4. 消息传播

在这种情况下，在 A 上发布的消息会传播到 B，但 B 不会把 A 的消息传播到 C。因此，订阅者 S_B 会收到在主题 A 和 B 上发布的消息，但订阅者 S_C 只会收到在主题 B 和 C 上发布的消息。如果应用需要让消息从 A 传播到 C，那么在 A 和 C 之间必须直接建立一个链接。

26.7.2 成本

如 26.7.1 节所述， IceStorm 消息只会在与发出消息的主题直接连接的链接上传播。此外，应用还可以用成本的概念来进一步对消息进行限制。

成本是与消息和链接关联在一起的。当你在某个主题上发布消息时，主题会拿与它的每个链接相关联的成本与消息的成本进行比较，并且只在那些成本等于或超过消息成本的链接上传播该消息。成本值为零 (0) 意味着:

- 不管链接成本是多少，具有零 (0) 成本值的消息会在主题的所有链接上发布;
- 不管消息成本是多少，具有零 (0) 成本值的链接接受所有消息。

例如，考虑图 26.5 中的主题图。

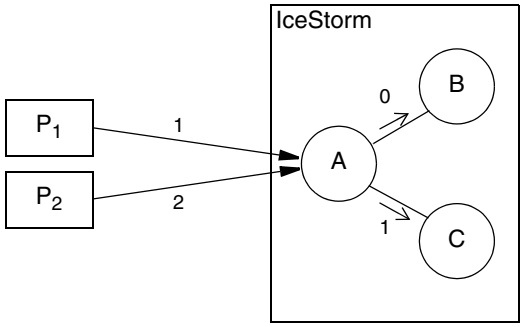

图 26.5. 成本语义

发布者 P_1 在主题 A 上发布成本为 1 的消息。这条消息会在通往主题 B 的链接上传播，因为该链接的成本为 0，因而接受所有消息。这条消息也会在通往主题 C 的链接上传播，因为消息成本没有超过链接成本 (1)。而 P_2 发布的成本为 2 的消息只会在通往 B 的链接上传播。

请求上下文

消息的成本在 Ice 请求上下文中指定。每个 Ice 代理操作都有一个隐含的参数，类型是 `Ice::Context`，用于表示请求上下文 (参见 16.8 节)。这个参数很少使用，但用来指定 IceStorm 消息的成本很理想，因为如果应用确实要使用 IceStorm 的成本特性，它只需提供一个请求上下文。如果请求上下文没有包含成本值，消息就会被赋予缺省的成本值零 (0)。

发布具有成本的消息

下面的代码例子演示了收集器是怎样发布成本值为 5 的观测数据的。首先是 C++ 版本:

```
Measurement m = getMeasurement();
Ice::Context ctx;
ctx["cost"] = "5";
monitor->report(m, ctx);
```

下面是等价的 Java 版本:

```
Measurement m = getMeasurement();
java.util.HashMap ctx = new java.util.HashMap();
ctx.put("cost", "5");
monitor.report(m, ctx);
```

接收具有成本的消息

订阅者可以通过检查 Ice::Current 参数所提供的请求上下文来找出消息的成本。例如，下面是 Monitor::report 的 C++ 实现，如果有成本值，它就会把该值显示出来:

```
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;
}
```

下面是等价的 Java 实现:

```
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();
}
```

为了效率的缘故， Ice for Java run time 可能会给 Ice.Current 中的请求上下文提供 null 值，因此，在使用请求上下文之前，应用必须检查其是否为 null。此外，非 null 请求上下文可能会在请求完成后发生变化，所以想要保留请求上下文的应用必须制作一个副本 (例如，使用 clone)。

26.7.3 联盟自动化

考虑到前面所描述的对消息传播的限制，要创建复杂的主题图可能会很繁琐。当然，创建主题图并不会频繁发生，因为 IceStorm 会保存主题图的

持久记录。但是，在有些情况下，通过自动的过程来创建主题图可能更有价值，比如在开发过程中，主题图可能会经常发生显著的变化，又比如有时主题图需要根据变化的成本重新进行计算。

管理工具脚本

使主题图的创建自动化的一种途径是，创建一个文本文件，其中含有要由 IceStorm 管理工具执行的命令。例如，可以用下面的命令来创建图 26.5 中的主题图:

```
create A B C
link A B 0
link A C 1
```

如果我们把这些命令存储在文件 `graph.txt` 中，我们可以用下面的命令来执行它们:

```
$ icestormadmin --Ice.Config=config graph.txt
```

我们假定，配置文件 `config` 中含有属性 `IceStorm.TopicManager.Proxy` 的定义。

XML 图描述符

IceStorm 为要使用消息和链接成本的应用提供了另外一种配置主题图的方法。管理工具 **`graph`** 命令就采用了这个特性，它可以接受一个 XML 图描述符，基于可到达性建立主题之间的链接，并使成本最小化。例如，让我们先考虑图 26.6 中的主题图。

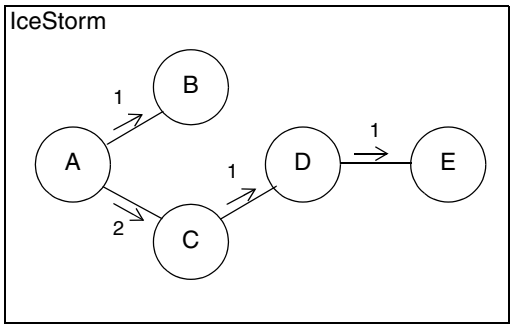

图 26.6. 最初的图

下面是用于这个主题图的 XML 图描述符。

```
<?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>
```

你可以看到，描述符的格式相当直截了当。外层的 graph 元素含有一个 vertex-set 元素和一个 edge-set 元素。 vertex-set 元素由两个或多个 vertex 元素组成，这些 vertex 元素具有单个属性 name，用于提供主题名。 edge-set 元素含有一个或多个 edge 元素，用于描述主题之间的链接。 source 和 target 属性提供主题名， cost 属性定义链接成本。

尽管这个描述符准确地描述了图 26.6 中的主题图的结构，管理工具的 **graph** 命令最终创建的图取决于指定的最大成本。假定我们把我们的图描述符保存为 graph.xml，并执行下面的命令:

```
$ icestormadmin --Ice.Config=config
>>> create A B C
>>> graph "graph.xml" 1
```

我们必须先创建用作 “我们的 XML 图描述符中的顶点” 的主题。接下来，我们让管理工具用最大成本 1 创建一个图，从而得到了图 26.7 中的图。

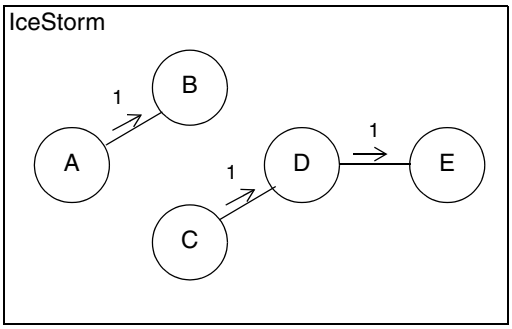

图 26.7. 最大成本为 1 的图

注意，从 A 到 C 的链接没有创建，因为该链接的成本 (2) 超过了最大成本。现在，看一看使用最大成本 2 所得到的图:

```
>>> graph "graph.xml" 2
```

如图 26.8 所示，现在从 A 到 C 的链接出现了。此外，还建立了从 C 到 E 的链接，其成本为 2，因为在最大成本之内就可以从 C 到达 E。

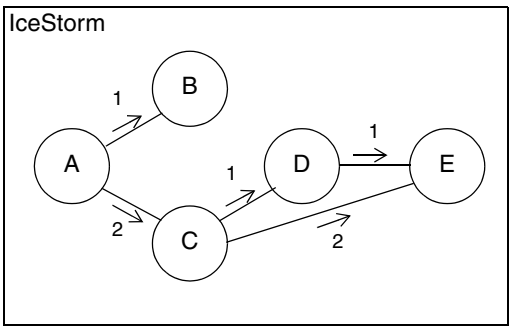

图 26.8. 最大成本为 2 的图

我们可以用 **list** 命令来检验各个链接是否得到了适当创建:

```
>>> 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
```

你可以反复使用 **graph** 命令来试验不同的成本配置。你每一次调用 **graph** 命令，管理工具都会基于指定的最大成本重新对图进行计算，然后建立必需的链接，并移除那些在新的成本约束下不再有效的链接。

26.8 服务质量

IceStorm 订阅者可以在订阅时指定 QoS 参数。IceStorm 目前只支持一个 QoS 参数 `reliability`，下面的小节将描述这个参数。

26.8.1 可靠性

QoS 参数 `reliability` 会对消息递送产生影响。其缺省值是 `oneway`，所发布的每一个消息都会造成对订阅者的一次单向调用。而 `batch` 则让 IceStorm 暂时把要递送给订阅者的消息放在队列里，并每隔一段时间把它们作为成批的单向请求递送出去 (参见第 18 章)。就订阅者的 servant 实现而言，这两种可靠性模式没有区别；消息都会以普通的方式、逐个地作为对 servant 的向上调用进行递送。但是，它们在安全性和性能方面有所不同。

`oneway` 模式强调的是安全性：只要一收到消息，它就会把它们递送出去，从而使消息因为 IceStorm 故障而丢失的可能性降到最低程度。

与此相反， `batch` 模式会把消息放在队列里，其存放时间可以配置，这增加了消息因为发生故障而丢失的可能性，但却通过使网络开销最小化而提供了更好的性能。 `batch` 模式递送队列中的消息的频度由配置属性 `IceStorm.Flush.Timeout` 决定 （参见附录 C）。

如果主题中经常发送小消息， `batch` 模式带来的性能提升尤其明显。`batch` 模式不会像 `oneway` 模式那样，发送许多小的请求，而是会把消息累

积起来，成组递送，从而在多条消息上分摊网络开销。但是，只有在情愿让消息等待一段时间才递送的情况下，订阅者才应该使用 batch 模式。

26.8.2 例子

Slice 类型 IceStorm::QoS 被定义为一个 dictionary，其键和值类型都是 string，因此 QoS 参数的名字和值都是用串来表示的。 26.5.3 节给出的例子代码把一个空词典用作 QoS 参数，意味着所用的是缺省值。下面给出的 C++ 和 Java 例子阐述了怎样把 reliability 参数设为 batch。

C++ 例子

```
IceStorm::QoS qos;
qos["reliability"] = "batch";
topic->subscribe(qos, proxy);
```

Java 例子

```
java.util.Map qos = new java.util.HashMap();
qos.put("reliability", "batch");
topic.subscribe(qos, proxy);
```

26.9 配置 IceStorm

IceStorm 是一个相对较轻的服务，所需的配置非常少，并且被实现为一个 IceBox 服务 (参见第 25 章)。在附录 C 中将描述 IceStorm 所支持的配置属性；但是，其中大部份属性都用于控制诊断输出，在本章中将不予讨论。

26.9.1 服务器配置

下面的服务器示例配置文件给出了最有意思的属性:

```
IceBox.Service.IceStorm=IceStormService,1.0.0:create
IceBox.DBEnvName.IceStorm=db
IceStorm.TopicManager.Endpoints=tcp -p 9999
IceStorm.Publish.Endpoints=tcp
```

第一个属性定义 IceStorm 服务的进入点。服务名 (IceBox.Service 属性名的最后一部分，在这个例子中也就是 IceStorm) 决定了 IceStorm

配置属性的前缀。使用 IceStorm 作为服务名并非是强制性的，但如果你没有更偏爱的名字，我们建议你就使用这个名字。

IceStorm 使用了 Freeze 来管理服务的持久状态，因此第二个属性指定了用于这个服务的 Freeze 数据库环境目录的路径名 (参见第 21 章)。在这个例子中使用的是 db 目录，在当前的工作目录中必须已经存在该目录。

最后的两个属性指定的是 IceStorm 对象适配器所使用的端点；注意，它们的属性名是以 IceStorm 起头的，与服务名相吻合。 TopicManager 属性指定的是 TopicManager 和 Topic 对象所在的端点；通常这些端点使用的是面向连接的协议，比如 TCP 或 SSL。 Publish 属性指定的是主题发布者对象使用的端点。

26.9.2 客户配置

服务的客户可以这样定义 TopicManager 对象的代理：

```
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 9999
```

属性的名字并不要紧，但端点必须与 IceStorm.TopicManager.Endpoints 属性定义的端点相吻合，而对象标识必须使用 IceStorm 服务名来做标识范畴，并用 TopicManager 做标识名。

26.10 总结

IceStorm 是一个发布 / 订阅服务，为 Ice 应用提供了一种灵活而高效的手段，用以把 “单向请求” 发布给一组订阅者。通过减轻应用管理订阅者和处理递送错误的负担， IceStorm 简化了开发，从而使得应用能够专注于数据的发布，而不是各种琐碎的分布事务。最后，为了给发布者和订阅者提供一个类型化的接口， IceStorm 还利用了 Ice 的请求 - 转发设施，从而使其对应用的影响得以降到了最低限度。

附录

附录 A
Slice 关键字

以下标识符是 Slice 关键字:

```
bool        enum        implements  module      string
byte        exception   int         nonmutating struct
class       extends     interface   Object      throws
const       false       local       out         true
dictionary  float       LocalObject sequence    void
double      idempotent  long        short
```

关键字的大小写必须按照给出的方式拼写。

附录 B
Slice 文档

B.1 Ice

综述

```
module Ice
```

Ice 核心库。Ice 核心库具有许多特性，下面是其部分功能：使用一种高效的协议（能进行协议压缩，并支持 TCP 和 UDP）来管理所有通信任务、为多线程服务器提供线程池，并且还具有其他一些能支持很高的可伸缩性的功能。

BoolSeq

```
sequence<bool> BoolSeq;
```

bool 序列。

ByteSeq

```
sequence<byte> ByteSeq;
```

byte 序列。

用于

::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::getSingleCertVerifier, ::IceSSL::Plugin::setRSAKeys, ::IceSSL::Plugin::setRSAKeys.

DoubleSeq

```
sequence<double> DoubleSeq;
```

double 序列。

FacetPath

```
sequence<string> FacetPath;
```

facet 路径。

用于

::Freeze::Evictor::addFacet, ::Freeze::Evictor::removeFacet, ::Freeze::EvictorStorageKey::facet, Current::facet, RequestFailedException::facet.

FloatSeq

```
sequence<float> FloatSeq;
```

float 序列。

IdentitySeq

```
local sequence<Identity> IdentitySeq;
```

标识序列。

IntSeq

```
sequence<int> IntSeq;
```

int 序列。

LongSeq

```
sequence<long> LongSeq;
```

long 序列。

ObjectProxySeq

```
sequence<Object*> ObjectProxySeq;
```

对象代理序列。

用于

::IcePack::Query::findAllObjectsWithType.

ObjectSeq

```
sequence<Object> ObjectSeq;
```

对象序列。

ShortSeq

```
sequence<short> ShortSeq;
```

short 序列。

StringSeq

```
sequence<string> StringSeq;
```

string 序列。

用于

Properties::getCommandLineOptions, Properties::parseCommandLineOptions, Properties::parseCommandLineOptions, Properties::parseIceCommandLineOptions, Properties::parseIceCommandLineOptions, ::IceBox::FreezeService::start, ::IceBox::Service::start, ::IcePack::Admin::addApplication, ::IcePack::Admin::addServer, ::IcePack::Admin::getAllAdapterIds, ::IcePack::Admin::getAllNodeNames, ::IcePack::Admin::getAllServerNames, ::IcePack::ServerDescription::args, ::IcePack::ServerDescription::envs, ::IcePack::ServerDescription::targets.

Context

```
dictionary<string, string> Context;
```

请求上下文。上下文的用途是把关于某个请求的元数据从服务器传到客户，比如服务质量 (QoS) 参数。客户的每个操作都有一个 Context，作为其隐含的最后一个参数。

用于

Current::ctx.

ObjectDict

```
local dictionary<Identity, Object> ObjectDict;
```

标识与 Ice 对象之间的映射。

PropertyDict

```
local dictionary<string, string> PropertyDict;
```

简单的属性集合，表示成键 / 值对组成的词典。键和值都是 string。

用于

Properties::getPropertiesForPrefix.

参见

Properties::getPropertiesForPrefix.

B.2 Ice::AbortBatchRequestException

综述

```
local exception AbortBatchRequestException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，用于表示某个批请求已被中止。

B.3 Ice::AdapterAlreadyActiveException

Overview

```
exception AdapterAlreadyActiveException
```

如果服务器试图为已经在活动的适配器设置端点，就会引发该异常。

B.4 Ice::AdapterNotFoundException

综述

```
exception AdapterNotFoundException
```

如果找不到适配器，就会引发该异常。

B.5 Ice::AlreadyRegisteredException

综述

```
local exception AlreadyRegisteredException
```

如果 servant、 servant 定位器、 facet、对象工厂、插件、对象适配器、对象、或用户异常工厂想要多次使用同一 ID 进行注册，就会引发该异常。

id

```
string id;
```

已经注册的对象的 id (或名字)。

kindOfObject

```
string kindOfObject;
```

已经注册的对象的种类："servant"、"servant locator"、"facet"、"object factory"、"plug-in"、"object adapter"、"object"，或 "user exception factory"。

B.6 Ice::BadMagicException

综述

```
local exception BadMagicException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，用于表示某条消息没有用预期的幻数起头 ('I', 'c', 'e', 'P')。

badMagic

```
ByteSeq badMagic;
```

含有不正确的消息的前四个字节的序列。

B.7 Ice::CloseConnectionException

综述

```
local exception CloseConnectionException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，用于表示连接已经被服务器得体地关闭。服务器还没有执行造成该异常的操作调用。在大多数情况下，你不会收到这个异常，因为在服务器关闭了连接的情况下，客户将自

动重试操作调用。但如果在重试时服务器又关闭了连接，而重试的次数限制已到，那么这个异常会转播给应用代码。

B.8 Ice::CloseTimeoutException

综述

`local exception CloseTimeoutException extends TimeoutException`

这个异常是对 `TimeoutException` 的特化，用于关闭连接超时的情况。

B.9 Ice::CollocationOptimizationException

综述

`local exception CollocationOptimizationException`

如果并置优化不支持所要的特性，就会引发该异常。

B.10 Ice::Communicator

综述

`local interface Communicator`

Ice 的核心对象。一个 Ice 应用可以实例化一个或多个通信器。通信器实例化是具体于某种语言的，没有在 Slice 代码中指定。

用于

`::Freeze::Connection::getCommunicator`, `ObjectAdapter::getCommunicator`, `::IceBox::FreezeService::start`, `::IceBox::Service::start`.

参见

`Logger`, `Stats`, `ObjectAdapter`, `Properties`, `ObjectFactory`.

addObjectFactory

```
void addObjectFactory(ObjectFactory factory, string id);
```

给这个通信器添加一个 servant 工厂。如果在安装工厂时使用的 id 已被用于某个已注册的工厂，就会抛出 AlreadyRegisteredException。

在解编 Ice 对象时， Ice run-time 会从线路上读取派生层次最深的类型 id，并尝试用工厂来创建该类型的实例。如果因为没有找到工厂，或所有的工厂都返回 nil，而没有能创建实例，对象就会被切成下一种派生层次最深的类型，然后再重复上述过程。如果没有找到能创建实例的工厂， Ice run-time 将把该对象切成 Ice::Object 类型。

在定位用于某种类型的工厂时采用了如下次序:

1. Ice run-time 查找专为该类型注册的工厂。
2. 如果还没有创建过实例， Ice run-time 就会查找缺省工厂，即用空类型 id 注册的工厂。
3. 如果还没有通过前面的步骤创建过实例，Ice run-time 就会查找由语言映射为非抽象类静态生成的工厂。

参数

factory

要添加的工厂。

id

该工厂所针对的类型的 id，空串表示是缺省工厂。

参见

removeObjectFactory, findObjectFactory, ObjectFactory.

createObjectAdapter

```
ObjectAdapter createObjectAdapter(string name);
```

创建一个新的对象适配器。这个对象适配器的端点取自 name.Endpoints 属性。

参数

name

对象适配器的名字。

返回值

新创建的对象适配器。

参见

createObjectAdapterWithEndpoints, ObjectAdapter, Properties.

createObjectAdapterWithEndpoints

```
ObjectAdapter createObjectAdapterWithEndpoints(string name, string
endpoints);
```

创建一个带端点的新对象适配器。这个方法会设置 name.Endpoints 属性，然后调用 createObjectAdapter。提供这个函数是为了方便。

参数

name

对象适配器的名字。

endpoints

对象适配器的端点。

返回值

新创建的对象适配器。

参见

createObjectAdapter, ObjectAdapter, Properties.

destroy

```
void destroy();
```

销毁通信器。这个操作会隐含调用 shutdown。调用 destroy 将清理内存，并关闭这个通信器的客户功能。对 destroy 的后继调用会被忽略。

参见

shutdown.

findObjectFactory

```
ObjectFactory findObjectFactory(string id);
```

查找一个向这个通信器注册过的 servant 工厂。

参数

id

该工厂所针对的类型的 id，空串表示是缺省工厂。

返回值

servant 工厂，如果没有找到使用给定 id 的 servant 工厂，则返回 null。

参见

addObjectFactory, removeObjectFactory, ObjectFactory.

flushBatchRequests

```
void flushBatchRequests();
```

刷出这个通信器的任何还未发出的批请求。所有的批请求都将使用“通过这个通信器获得的代理”发给服务器。

getLogger

```
Logger getLogger();
```

获取这个通信器的日志记录器。

返回值

这个通信器的日志记录器。

参见

Logger.

getPluginManager

```
PluginManager getPluginManager();
```

获取这个通信器的插件管理器。

返回值

这个通信器的插件管理器。

参见

PluginManager.

getProperties

```
Properties getProperties();
```

获取这个通信器的属性。

返回值

这个通信器的属性。

参见

Properties.

getStats

```
Stats getStats();
```

获取这个通信器的统计回调对象。

返回值

这个通信器的统计回调对象。

参见

Stats.

proxyToString

```
string proxyToString(Object* obj);
```

把代理转换成串。

参数

obj

要转换成串的代理。

返回值

" 串化的 " 代理。

参见

stringToProxy.

removeObjectFactory

```
void removeObjectFactory(string id);
```

从这个通信器中移除指定的 servant 工厂。如果没有工厂与你指定的 id 吻合，就引发 NotRegisteredException。

参数

id

该工厂所针对的类型的 id，空串表示是缺省工厂。

参见

addObjectFactory, findObjectFactory, ObjectFactory.

setDefaultLocator

```
void setDefaultLocator(Locator* loc);
```

为这个通信器创建一个缺省的 Ice 定位器。所有新创建的代理和对象适配器都将使用这个缺省定位器。要禁用缺省定位器，可以使用 null。注意，这个操作对已有的代理或对象适配器不起作用。

你可以调用代理的 ice_locator 操作，为个别的代理设置定位器，或调用对象适配器的 setLocator 操作，为其设置定位器。

参数

loc

要用于这个通信器的缺省定位器。

参见

Locator, ObjectAdapter::setLocator.

setDefaultRouter

```
void setDefaultRouter(Router* rtr);
```

为这个通信器设置一个缺省的路由器。所有新创建的代理都将使用这个缺省路由器。要禁用缺省路由器，可以使用 null。注意，这个操作对已有的代理不起作用。

你可以调用代理的 ice_router 操作，为个别的代理设置路由器。

参数

rtr

要使用这个通信器的缺省路由器。

参见

Router, ObjectAdapter::addRouter.

setLogger

```
void setLogger(Logger log);
```

为这个通信器设置日志记录器。

参数

log

要用于这个通信器的日志记录器。

参见

Logger.

setStats

```
void setStats(Stats st);
```

为这个通信器设置统计回调对象。

参数

st

要用于这个通信器的统计回调对象。

参见

Stats.

shutdown

```
void shutdown();
```

关闭这个通信器的服务器功能，包括解除所有对象适配器的激活。对 shutdown 的后续调用将被忽略。

在 shutdown 返回后，不再会处理新的请求。但是，在 shutdown 被调用之前开始处理的请求可能仍然是活动的。你可以使用 waitForShutdown 来等待所有请求完成。

参见

destroy, waitForShutdown, ObjectAdapter::deactivate.

stringToProxy

```
Object* stringToProxy(string str);
```

把串转换成代理。例如， MyCategory/MyObject:tcp -h some_host -p 10000将创建指向某个Ice对象的代理，其标识的名字是"MyObject"、范畴是"MyCategory"，服务器将运行在主机 "some_host" 的端口 10000 上。

参数

str

要转换成代理的串。

返回值

代理。

参见

proxyToString.

waitForShutdown

```
void waitForShutdown();
```

等待这个通信器的服务器功能完全关闭。要发起关闭，调用 shutdown；只有在所有仍在处理的请求完成后，waitForShutdown 才会返回。你通常会在主线程中调用这个操作，等待其他线程 shutdown。在关闭完成后，主线程将返回，并可以在最终调用 destroy 关闭客户功能之前，完成一些清理工作，然后再退出应用。

参见

shutdown, destroy, ObjectAdapter::waitForDeactivate.

B.11 Ice::CommunicatorDestroyedException

综述

```
local exception CommunicatorDestroyedException
```

如果 Communicator 已被销毁，就会引发该异常。

参见

Communicator::destroy.

B.12 Ice::CompressionException

综述

```
local exception CompressionException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，如果在压缩或解压数据时出了问题，就会引发该异常。

reason

```
string reason;
```

描述压缩或解压时遇到的问题。

B.13 Ice::CompressionNotSupportedException

综述

```
local exception CompressionNotSupportedException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，如果某个不支持压缩的 Ice 版本收到压缩的协议消息，就会引发该异常。

B.14 Ice::ConnectFailedException

综述

```
local exception ConnectFailedException extends SocketException
```

这个异常是对 SocketException 的特化，用于表示连接失败。

B.15 Ice::ConnectTimeoutException

综述

```
local exception ConnectTimeoutException extends TimeoutException
```

这个异常是对 TimeoutException 的特化，用于表示连接建立超时的情况。

B.16 Ice::ConnectionLostException

综述

```
local exception ConnectionLostException extends SocketException
```

这个异常是对 SocketException 的特化，用于表示连接已丢失。

B.17 Ice::ConnectionNotValidatedException

综述

```
local exception ConnectionNotValidatedException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，如果在还没有验证的连接上收到消息，就会引发该异常。

B.18 Ice::ConnectionTimeoutException

综述

```
local exception ConnectionTimeoutException extends
TimeoutException
```

这个异常是对 TimeoutException 的特化，用于表示某个连接因为空闲了一段时间，已经关闭。

B.19 Ice::Current

综述

```
local struct Current
```

关于服务器的当前方法调用的信息。服务器上的每个操作的最后一个参数都是隐含的 Current 参数。 Current 主要用于 Ice 服务，比如 IceStorm。大多数应用都会忽略这个参数。

用于

ServantLocator::finished, ServantLocator::locate.

参见

::IceStorm.

adapter

ObjectAdapter adapter;

对象适配器。

ctx

Context ctx;

请求上下文，其内容是从客户接收来的。

facet

FacetPath facet;

facet。

id

Identity id;

Ice 对象标识。

mode

OperationMode mode;

操作的模式。

operation

string operation;

操作的名字。

B.20 Ice::DNSException

综述

```
local exception DNSException
```

这个异常说明出了 DNS 问题。要了解详细的原因，应该检查 error。

error

```
int error;
```

DNS 问题的错误号。就 C++ 和 UNIX 而言，这个错误号和 h_errno 是等价的。就 C++ 和 Windows 而言，这个错误号是 WSAGetLastError() 返回的值。

host

```
string host;
```

无法解析的主机名。

B.21 Ice::DatagramLimitException

综述

```
local exception DatagramLimitException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，如果数据报的尺寸超过了先前配置的发送或接收缓冲区的尺寸，或是超过了 UDP 包的最大有效负载尺寸 (65507 bytes)，就会引发该异常。

B.22 Ice::EncapsulationException

综述

```
local exception EncapsulationException extends MarshalException
```

这个异常是对 MarshalException 的特化，用于表示数据封装有问题。

B.23 Ice::EndpointParseException

综述

```
local exception EndpointParseException
```

如果在解析端点的过程中出了错，就会引发该异常。

str

```
string str;
```

无法解析的串。

B.24 Ice::FacetNotExistException

综述

```
local exception FacetNotExistException extends
RequestFailedException
```

如果对象没有实现指定的 facet 路径，就会引发该异常。

B.25 Ice::Identity

综述

struct Identity

Ice 对象的标识。空的 name 表示 null 对象。

用于

::Freeze::Evictor::addFacet, ::Freeze::Evictor::createObject, ::Freeze::Evictor::destroyObject, ::Freeze::Evictor::hasObject, ::Freeze::Evictor::removeAllFacets, ::Freeze::Evictor::removeFacet, ::Freeze::EvictorIterator::next, ::Freeze::EvictorStorageKey::identity, ::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;

Ice 对象的范畴。

参见

ServantLocator, ObjectAdapter::addServantLocator.

name

string name;

Ice 对象的名字。

B.26 Ice::IdentityParseException

综述

```
local exception IdentityParseException
```

如果在解析串化标识的过程中出了错，就会引发该异常。

str

```
string str;
```

无法解析的串。

B.27 Ice::IllegalIdentityException

综述

```
local exception IllegalIdentityException
```

如果遇到非法标识，就会引发该异常。

id

```
Identity id;
```

非法标识。

B.28 Ice::IllegalIndirectionException

综述

```
local exception IllegalIndirectionException extends
MarshalException
```

这个异常是对 MarshalException 的特化，用于表示解编过程中出现了非法的间接引用。

B.29 Ice::IllegalMessageSizeException

综述

```
local exception IllegalMessageSizeException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，用于表示消息的尺寸是非法的，也就是说，它比规定的最小尺寸还小。

B.30 Ice::Locator

综述

```
interface Locator
```

Ice 定位器接口。客户用这个接口来查找适配器和对象。服务器还用它来获取定位器注册表代理。

Locator 接口的设计用途是供 Ice 内部使用，也可用于定位器实现。常规的用户代码不应该直接使用这个接口的任何功能。

findAdapterById

```
[ "amd" ] Object* findAdapterById(string id) throws
AdapterNotFoundException;
```

根据 id 查找适配器，并返回其代理 (适配器创建的虚设的直接代理)。

参数

id

适配器 id。

返回值

适配器代理，如果适配器不是活动的，就返回 null。

异常

AdapterNotFoundException

如果找不到适配器，就会引发该异常。

findObjectById

```
[ "amd" ] Object* findObjectById(Identity id) throws
ObjectNotFoundException;
```

根据标识查找对象，并返回其代理。

参数

id

标识。

返回值

该对象的代理，如果对象不是活动的，就返回 null。

异常

ObjectNotFoundException

如果找不到对象，就会引发该异常。

getRegistry

```
LocatorRegistry* getRegistry();
```

获取定位器注册表。

返回值

定位器注册表。

B.31 Ice::LocatorRegistry

综述

```
interface LocatorRegistry
```

Ice 定位器注册表接口。服务器用这个接口来向定位器注册适配器的端点。

LocatorRegistry 接口的设计用途是供 Ice 内部使用，也可用于定位器实现。常规的用户代码不应该直接使用这个接口的任何功能。

setAdapterDirectProxy

```
void setAdapterDirectProxy(string id, Object* proxy) throws
AdapterNotFoundException, AdapterAlreadyActiveException;
```

通过定位器注册表设置适配器端点。

Parameters

id

适配器 id。

proxy

适配器代理 (适配器创建的虚设的直接代理)。这个直接代理含有适配器的端点。

异常

AdapterNotFoundException

如果找不不到适配器，或者定位器只允许已注册的适配器设置其活动的代理、而这个适配器没有向定位器注册，就会引发该异常。

AdapterAlreadyActive

如果已经有具有相同 id 的适配器在活动，就会引发该异常。

setServerProcessProxy

```
void setServerProcessProxy(string id, Process* proxy) throws
ServerNotFoundException;
```

设置服务器的进程代理。

参数

id

服务器 id。

proxy

进程代理。

异常

ServerNotFoundException

如果找不到服务器，就会引发该异常。

B.32 Ice::Logger

综述

local interface Logger

Ice 消息日志记录器。通过实现该接口、并在通信器中安装它，应用可以提供自己的日志记录器。

用于

Communicator::getLogger, Communicator::setLogger.

error

void error(string message);

把出错消息记入日志。

参数

message

要记录的出错消息。

参见

warning.

trace

void trace(string category, string message);

把跟踪消息记入日志。

参数

category

跟踪范畴。

message

要记录的跟踪消息。

warning

```
void warning(string message);
```

把警告消息记入日志。

参数

message

要记录的警告消息。

参见

error.

B.33 Ice::MarshalException

综述

```
local exception MarshalException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，如果数据整编或解编出错，就会引发该异常。

派生异常

EncapsulationException, IllegalIndirectionException, MemoryLimitException, NegativeSizeException, NoObjectFactoryException, ProxyUnmarshalException, UnmarshalOutOfBoundsException.

B.34 Ice::MemoryLimitException

综述

```
local exception MemoryLimitException extends MarshalException
```

这个异常是对 MarshalException 的特化。如果在整编或解编过程中超过了相应系统的内存限制，就会引发该异常。

B.35 Ice::NegativeSizeException

综述

```
local exception NegativeSizeException extends MarshalException
```

这个异常是对 MarshalException 的特化，如果收到负的尺寸 (例如，负的序列尺寸)，就会引发该异常。

B.36 Ice::NoEndpointException

综述

```
local exception NoEndpointException
```

如果没有合适的端点可用，就会引发该异常。

proxy

```
string proxy;
```

没有合适端点可用的串化代理。

B.37 Ice::NoObjectFactoryException

综述

```
local exception NoObjectFactoryException extends MarshalException
```

这个异常是对 MarshalException 的特化，如果在对象解编过程中，没有找到合适的对象工厂，就会引发该异常。

参见

ObjectFactory, Communicator::addObjectFactory, Communicator::removeObjectFactory, Communicator::findObjectFactory.

type

```
string type;
```

我们无法为之找到工厂的对象的绝对的 Slice 类型名。

B.38 Ice::NotRegisteredException

综述

```
local exception NotRegisteredException
```

如果要移除的 servant、 facet、对象工厂、插件、对象适配器、对象、或是用户异常工厂并没有注册，就会引发该异常。

id

```
string id;
```

无法移除的对象的 id (或名字)。

kindOfObject

```
string kindOfObject;
```

无法移除的对象的种类："servant"、"facet"、"object factory"、"plug-in"、"object adapter"、"object"，或 "user exception factory"。

B.39 Ice::ObjectAdapter

综述

`local interface ObjectAdapter`

对象适配器，负责接收来自端点的请求、并在 servant、标识和代理之间进行映射。

用于

`::Freeze::ServantInitializer::initialize`, `Communicator::createObjectAdapter`, `Communicator::createObjectAdapterWithEndpoints`, `Current::adapter`.

参见

`Communicator`, `ServantLocator`.

activate

```
void activate();
```

激活属于这个对象适配器的所有端点。在激活之后，对象适配器就可以分派通过其端点接收的请求了。

参见

`hold`, `deactivate`.

add

```
Object* add(Object servant, Identity id);
```

把一个 servant 添加到这个对象适配器的 Active Servant Map。注意，通过用多个标识注册同一个 servant，可以让一个 servant 实现多个 Ice 对象。如果所添加的 servant 的标识在映射表中已经存在，就引发 `AlreadyRegisteredException`。

参数

`servant`

要添加的 servant。

`id`

这个 servant 所实现的 Ice 对象的标识。

返回值

与给定的标识及这个对象适配器相匹配的代理。

参见

`Identity`, `addWithUUID`, `remove`.

addRouter

```
void addRouter(Router* rtr);
```

给这个对象适配器添加一个路由器。藉此，这个对象适配器器可以接收来自这个路由器的回调，其所用连接就是从这个进程到路由器的连接。这样，路由器就无需再另外建一条回到这个对象适配器的连接了。

对于一个特定的路由器，你只能把它添加到一个对象适配器。如果你把同一个路由器添加到多个对象适配器，就会产生不确定的行为。但是，把不同的路由器添加到不同的对象适配器是可能的。

参数

`rtr`

要添加到这个对象适配器的路由器。

参见

`Router`, `Communicator::setDefaultRouter`.

addServantLocator

```
void addServantLocator(ServantLocator locator, string category);
```

把一个 Servant Locator 添加到 这个对象适配器。如果你添加的 servant 定位器所针对的范畴已经注册了 servan 定位器，就引发

`AlreadyRegisteredException`。为了分派 servant 上的操作调用，对象适配器会尝试按下列顺序查找与给定的 Ice 对象标识相对应的 servant:

1. 对象适配器尝试在 Active Servant Map 中查找与该标识对应的 servant。
2. 如果在 Active Servant Map 中没有找到 servant，对象适配器会尝试查找与该标识的范畴部分相对应的定位器。如果找到了定位器，对象适配器就会尝试用这个定位器来查找 servant。
3. 如果上述步骤都没有找到 servant，对象适配器就会尝试查找用于空范畴的定位器，而不去管标识中包含的范畴是什么。如果找到了定位器，对象适配器就会尝试用这个定位器来查找 servant。
4. 如果上述步骤都没有找到 servant，对象适配器就会放弃，调用者将会收到 `ObjectNotExistException`。

针对空范畴，只能安装一个定位器。

参数

`locator`

要添加的定位器。

`category`

这个 Servant Locator 所针对的范畴，空串表示这个 Servant Locator 不属于任何特定的范畴。

参见

`Identity`, `findServantLocator`, `ServantLocator`.

addWithUUID

```
Object* addWithUUID(Object servant);
```

把一个 servant 添加到这个对象适配器的 Active Servant Map 中，用自动生成的 UUID 作为其标识。注意，可以用代理的 `ice_getIdentity` 操作来访问生成的 UUID 标识。

参数

`servant`

要添加的 servant。

返回值

与生成的 UUID 标识及这个对象适配器相匹配的代理。

参见

Identity, add, remove.

createDirectProxy

```
Object* createDirectProxy(Identity id);
```

创建一个与这个对象适配器及给定标识相匹配的 " 直接代理 "。直接代理总是包含有当前的适配器端点。

这个操作的设计用途是供定位器实现使用。常规的用户代码不应该使用这个操作。

参数

id

要创建的代理的标识。

返回值

与给定的标识及这个对象适配器相匹配的代理。

参见

Identity.

createProxy

```
Object* createProxy(Identity id);
```

创建一个与这个对象适配器及给定标识相匹配的代理。

Parameters

id

要创建的代理的标识。

返回值

与给定标识及这个对象适配器相匹配的代理。

参见

Identity.

createReverseProxy

```
Object* createReverseProxy(Identity id);
```

创建一个与这个对象适配器及给定标识相匹配的 " 反向代理 "。反向代理使用的连接是已经建立的从客户到这个对象适配器的连接。

和 Router 接口一样，这个操作的设计用途是供路由器实现使用。常规的用户代码不应该使用这个操作。

参数

id

要创建的代理的标识。

返回值

与给定标识及这个对象适配器相匹配的 " 反向代理 "。

参见

Identity.

deactivate

```
void deactivate();
```

解除所有属于这个对象适配器的端点的激活。在解除激活之后，对象适配器将停止通过其端点接收请求。已经解除激活的对象适配器不能再重新激活，也就是说，解除激活是永久性的，在调用了 deactivate 之后，不能再调用 activate 或 hold；如果你这样做，就引发 ObjectAdapterDeactivatedException。在已经解除了激活的对象适配器上调用 deactivate，将会被忽略。

在 deactivate 返回之后，对象适配器不会再处理新的请求。但是，在 deactivate 被调用之前已经开始处理的请求可能仍然在活动。你可以用 waitForDeactivate 来等待这个对象适配器的所有请求完成。

参见

activate, hold, waitForDeactivate, Communicator::shutdown.

findServantLocator

```
ServantLocator findServantLocator(string category);
```

查找一个已经安装到这个对象适配器中的 Servant Locator。

参数

`category`

这个 Servant Locator 所针对的范畴，空串表示这个 Servant Locator 不属于任何特定的范畴。

返回值

Servant Locator，如果找不到针对给定范畴的 Servant Locator，就返回 null。

参见

`Identity`, `addServantLocator`, `ServantLocator`.

getCommunicator

```
Communicator getCommunicator();
```

获取这个对象适配器所属的通信器。

返回值

这个对象适配器的通信器。

参见

`Communicator`.

getLocator

```
Locator* getLocator();
```

获取为这个对象适配器配置的定位器。

返回值

定位器的代理。

参见

Locator.

getName

```
string getName();
```

获取这个对象适配器的名字。

返回值

这个对象适配器的名字。

hold

```
void hold();
```

暂时停止接收和分派请求。可以用 activate 操作重新激活对象适配器。

"扣留"并非是即时的，也就是说，在 hold 返回之后，对象适配器可能仍然会活动一段时间。你可以用 waitForHold 来等待"扣留"完成。

参见

activate, deactivate, waitForHold.

identityToServant

```
Object identityToServant(Identity id);
```

根据 servant 所实现的 Ice 对象的标识来在这个对象适配器的 Active Servant Map 中查找该 servant。

这个操作只在 Active Servant Map 中查找 servant。它不会使用任何已安装的 ServantLocator 来查找 servant。

参数

id

你想要的 servant 所实现的 Ice 对象的标识。

返回值

实现了“具有给定标识的 Ice 对象”的 servant，如果没有找到这样的 servant，就返回 null。

参见

Identity, proxyToServant.

proxyToServant

```
Object proxyToServant(Object* proxy);
```

根据给定的代理，在这个对象适配器的 Active Servant Map 中查找 servant。这个操作只在 Active Servant Map 中查找 servant。它不会使用任何已安装的 ServantLocator 来查找 servant。

参数

proxy

你想要的 servant 的代理。

返回值

与代理相匹配的 servant，如果没有找到这样的 servant，就返回 null。参见 identityToServant。

remove

```
void remove(Identity id);
```

从对象适配器的 Active Servant Map 中移除一个 servant。

参数

id

该 servant 所实现的 Ice 对象的标识。如果 servant 实现了多个 Ice 对象，你必须针对所有这些 Ice 对象调用 remove。如果你想要移除的标识不在映射表中，就引发 NotRegisteredException。

参见

Identity, add, addWithUUID.

setLocator

```
void setLocator(Locator* loc);
```

为这个对象适配器设置一个 Ice 定位器。藉此，对象适配器将在初次激活时，向定位器注册表注册自身。此外，这个对象适配器所创建的代理将会包含适配器名，而非其端点。

参数

loc

这个对象适配器要使用的定位器。

参见

createDirectProxy, Locator, LocatorRegistry.

waitForDeactivate

```
void waitForDeactivate();
```

等待对象适配器解除激活。要发起对象适配器的激活解除，调用 deactivate；要等待激活解除的完成，调用 waitForDeactivate。

参见

deactivate, waitForHold, Communicator::waitForShutdown.

waitForHold

```
void waitForHold();
```

等待对象适配器扣留请求。要发起请求的扣留，调用 hold；要等待请求扣留的完成，调用 waitForHold。

参见

hold, waitForDeactivate, Communicator::waitForShutdown.

B.40 Ice::ObjectAdapterDeactivatedException

综述

```
local exception ObjectAdapterDeactivatedException
```

如果试图使用已解除激活的 ObjectAdapter，就会引发该异常。

参见

ObjectAdapter::deactivate, Communicator::shutdown.

name

```
string name;
```

适配器的名字。

B.41 Ice::ObjectAdapterIdInUseException

综述

```
local exception ObjectAdapterIdInUseException
```

如果因为 Locator 发现已经有一个 id 相同的 ObjectAdapter 在活动，所以无法激活某个 ObjectAdapter，就会引发该异常。

id

```
string id;
```

适配器 id。

B.42 Ice::ObjectFactory

Overview

```
local interface ObjectFactory
```

对象工厂，可以用在若干地方，例如，在接收 " 通过值传递的对象 " 时，以及在 ::Freeze 取回持久对象时。对象工厂必须由应用的编写者实现，并向通信器注册。

用于

Communicator::addObjectFactory, Communicator::findObjectFactory.

参见

::Freeze.

create

```
Object create(string type);
```

为给定的对象类型创建一个新对象。类型是绝对的 Slice 类型名，也就是说，相对于无名的顶层 Slice 模块的名字。例如，在模块 Foo 中的 Bar 类型的接口的绝对 Slice 类型名是 ::Foo::Bar。

起首处的 "::" 是必需的。

参数

type

对象类型。

返回值

为给定类型创建的对象，如果工厂无法创建该对象，就返回 nil。

destroy

```
void destroy();
```

会在工厂从通信器中移除时、或通信器销毁时被调用。

参见

Communicator::removeObjectFactory, Communicator::destroy.

B.43 Ice::ObjectNotExistException

综述

```
local exception ObjectNotExistException extends
RequestFailedException
```

如果对象在服务器上不存在，就会引发该异常。

B.44 Ice::ObjectNotFoundException

综述

exception ObjectNotFoundException

如果找不到对象，就会引发该异常。

B.45 Ice::OperationMode

综述

enum OperationMode

OperationMode 将决定骨架的型构 (C++)，以及在发生可恢复的错误时，Ice run time 重试操作调用的次数。

用于

Current::mode.

Normal

Normal

平常的操作的模式是 Normal。这些操作会修改对象状态；接连两次调用这样的操作所具有的语义和调用一次不同。 Ice run time 将保证，对于 Normal 操作，它不会违反 “最多一次”语义。

Nonmutating

Nonmutating

使用 Slice 的 Nonmutating 关键字的操作不能修改对象状态。在 C++ 里， nonmutating 操作将会在骨架里生成 const 成员函数。此外， Ice run

time 会尝试通过重新发出失败了的请求，透明地从某些运行时错误中恢复，并只在第二次尝试也失败的情况下，才把失败告知应用。

Idempotent

`Idempotent`

使用了 Slice 的 `Idempotent` 关键字的操作可以修改对象状态，接连两次调用这样的操作之后的对象状态和调用一次必须是一样的。例如，`x = 1` 是一个 idempotent 语句，而 `x += 1` 则不是。对于 idempotent 操作，在发生可恢复的错误时，Ice run-time 采用的重试行为和用于 nonmutating 操作的是一样的。

B.46 Ice::OperationNotExistException

综述

```
local exception OperationNotExistException extends
RequestFailedException
```

如果在服务器上，给定对象的某个操作不存在，就引发这个异常。通常，这是由客户或服务器使用了过时的 Slice 规范引起的。

B.47 Ice::Plugin

综述

```
local interface Plugin
```

通信器插件。插件通常用于给通信器增加特性，比如支持某种协议。

派生类与接口

`::IceSSL::Plugin`.

用于

`PluginManager::addPlugin`, `PluginManager::getPlugin`.

destroy

```
void destroy();
```

在通信器正在销毁时被调用。

B.48 Ice::PluginInitializationException

综述

```
local exception PluginInitializationException
```

这个异常表示在插件初始化过程中发生了失败。

reason

```
string reason;
```

失败原因。

B.49 Ice::PluginManager

综述

```
local interface PluginManager
```

每个通信器都有一个插件管理器，用于管理插件集。

用于

Communicator::getPluginManager.

addPlugin

```
void addPlugin(string name, Plugin pi);
```

安装一个新插件。

参数

name

插件名。

pi

插件。

destroy

```
void destroy();
```

在通信器正在销毁时被调用。

getPlugin

```
Plugin getPlugin(string name);
```

根据插件名获取插件。

参数

name

插件名。

返回值

插件。

B.50 Ice::Process

综述

```
interface Process
```

用于管理进程的管理接口。被管理的服务器必须实现这个接口，并调用 ObjectAdapter::setProcess 注册进程代理。

实现了这个接口的 servant 可能会受到 “拒绝服务” 攻击，因此，必须采取适当的安全预防措施。例如， servant 可以使用 UUID 来使其标识难以被猜出，并使用具有安全端点的对象适配器。

shutdown

```
void shutdown();
```

发起得体的关闭。

参见

Communicator::shutdown.

B.51 Ice::Properties

综述

```
local interface Properties
```

用于配置 Ice 和 Ice 应用的属性集。属性是键 / 值对，键和值都是 string。按照惯例，属性键应该具有这样的形式：*application-name*[.*category*[.*sub-category*]].*name*。

用于

Communicator::getProperties, clone.

clone

```
Properties clone();
```

创建这个属性集的副本。

返回值

这个属性集的副本。

getCommandLineOptions

```
StringSeq getCommandLineOptions();
```

获取与这个属性集等价的命令行选项序列。返回的序列的每个元素都是一个命令行选项，其形式是：--key=value。

返回值

与这个属性集等价的命令行选项。

getPropertiesForPrefix

```
PropertyDict getPropertiesForPrefix(string prefix);
```

获取所有其键以 *prefix* 起头的属性。如果 *prefix* 是空串，那么将返回所有属性。

返回值

具有指定前缀的属性集。

getProperty

```
string getProperty(string key);
```

根据键获取属性。如果不存在这样的属性，就返回空串。

参数

key

属性键。

返回值

属性值。

参见

setProperty.

getPropertyAsInt

```
int getPropertyAsInt(string key);
```

把某个属性转换成整数。如果该属性不存在，就返回 0。

参数

key

属性键。

返回值

被解释成整数的属性值。

参见

setProperty.

getPropertyAsIntWithDefault

```
int getPropertyAsIntWithDefault(string key, int value);
```

把某个属性转换成整数。如果该属性不存在，就返回给定的缺省值。

参数

key

属性键。

value

在属性不存在的情况下使用的缺省值。

返回值

被解释成整数的属性值，或缺省值。

参见

setProperty.

getPropertyWithDefault

```
string getPropertyWithDefault(string key, string value);
```

根据键获取属性。如果该属性不存在，就返回给定的缺省值。

Parameters

key

属性键。

value

在属性不存在的情况下使用的缺省值。

返回值

属性值或缺省值。

参见

setProperty.

load

```
void load(string file);
```

从文件中加载属性。

参数

file

属性文件。

parseCommandLineOptions

```
StringSeq parseCommandLineOptions(string prefix, StringSeq
options);
```

把一个命令行选项的序列转换成属性。所有以 *--prefix* 起头的选项都将被转换成属性。如果前缀是空的，所有以 -- 起头的选项都将被转换成属性。

参数

prefix

属性前缀，空串表示要转换所有以 -- 起头的选项。

options

命令行选项。

返回值

不是以指定前缀起头的命令行选项，保持原来的次序不变。

parseIceCommandLineOptions

```
StringSeq parseIceCommandLineOptions(StringSeq options);
```

把一个命令行选项的序列转换成属性。所有以下列前缀起头的选项都将被转换成属性：--Ice、 --IceBox、 --IcePack、 --IcePatch、 --IceSSL、 --IceStorm、 --Freeze，以及 --Glacier。

参数

options

命令行选项。

返回值

不是以所列出的前缀起头的命令行选项，保持原来的次序不变。

setProperty

```
void setProperty(string key, string value);
```

设置属性。要想取消属性的设置，把它设成空串。

参数

key

属性键。

value

属性值。

参见

getProperty.

B.52 Ice::ProtocolException

Overview

```
local exception ProtocolException
```

所有协议错误的基异常。

派生

AbortBatchRequestException, BadMagicException, CloseConnectionException, CompressionException, CompressionNotSupportedException, ConnectionNotValidatedException, DatagramLimitException, IllegalMessageSizeException, MarshalException, UnknownMessageException, UnknownReplyStatusException, UnknownRequestIdException, UnsupportedEncodingException, UnsupportedProtocolException.

B.53 Ice::ProxyParseException

综述

```
local exception ProxyParseException
```

如果在解析串化代理时出错，就会引发该异常。

str

```
string str;
```

无法解析的串。

B.54 Ice::ProxyUnmarshalException

Overview

```
local exception ProxyUnmarshalException extends MarshalException
```

这个异常是对 MarshalException 的特化，如果在解编代理时收到不一致的数据，就会引发该异常。

B.55 Ice::RequestFailedException

综述

```
local exception RequestFailedException
```

如果某个请求失败，就会引发该异常。这个异常及所有从它派生的异常，都由 Ice 协议进行传送，即使它们被声明成 local 异常。

派生异常

FacetNotExistException, ObjectNotExistException, OperationNotExistException.

facet

```
FacetPath facet;
```

请求所针对的 facet。

id

```
Identity id;
```

请求所针对的 Ice 对象的标识。

operation

```
string operation;
```

请求的操作名。

B.56 Ice::Router

综述

```
interface Router
```

Ice 路由器接口。可以用 Communicator::setDefaultRouter 设置全局的路由器，也可以用 ice_router 设置具体代理的路由器。

路由器的设计用途是供 “Ice 内部和路由器实现” 使用。常规的用户代码不应该直接使用这个接口的任何功能。

派生类与接口

::Glacier::Router.

addProxy

```
void addProxy(Object* proxy);
```

给路由器的路由表添加新的代理信息。

参数

proxy

要添加的代理。

getClientProxy

```
Object* getClientProxy();
```

获取路由器的客户代理，也就是，用于把请求从客户转发到路由器的代理。

返回值

路由器的客户代理。

getServerProxy

```
Object* getServerProxy();
```

获取路由器的服务器代理，也就是，用于把请求从服务器转发到路由器的代理。

返回值

路由器的服务器代理。

B.57 Ice::ServantLocator

综述

```
local interface ServantLocator
```

servant 定位器，对象适配器会调用它来定位在其活动 servant 映射表中找不到的 servant。

派生类与接口

::Freeze::Evictor.

用于

ObjectAdapter::addServantLocator, ObjectAdapter::findServantLocator.

参见

ObjectAdapter, ObjectAdapter::addServantLocator, ObjectAdapter::findServantLocator.

deactivate

```
void deactivate(string category);
```

这个操作将在下述情况下被调用：这个 servant 定位器所在的对象适配器被解除激活。

参数

category

说明 servant 定位器正在针对哪个范畴解除激活。

参见

ObjectAdapter::deactivate, Communicator::shutdown, Communicator::destroy.

finished

```
void finished(Current curr, Object servant, LocalObject cookie);
```

对象适配器在处理完请求之后会调用这个操作。只有在下述情况下，这个操作才会被调用：在处理请求之前调用了 locate，并返回了非 null servant。这个操作可以用于在处理请求之后进行清理。

参数

curr

与当前操作调用有关的信息，locate 所找到的 servant 提供了这个操作。

servant

locate 返回的 servant。

cookie

locate 返回的 cookie。

参见

ObjectAdapter, Current, locate.

locate

```
Object locate(Current curr, out LocalObject cookie);
```

当在对象适配器的活动 servant 映射表中无法找到 servant 时，对象适配器会在处理请求之前调用 locate。注意，对象适配器不会自动把返回的 servant 插入其活动 servant 映射表中。如果需要，这必须由 servant 定位器实现来完成。

重要提示：如果你从自己的代码中调用 locate*，在你使用完 servant 之后，你必须调用* finished - *前提是返回的是非 null servant。否则，如果你使用像* ::Freeze::Evictor *那样的 Servant Locator，就会产生不确定的行为。*

参数

curr

与当前操作有关的信息，locate 必须找到一个提供了该操作的 servant。

cookie

要传给 finished 的 "cookie"。

返回值

找到的 servant，如果没有找到合适的 servant，就返回 null。

参见

ObjectAdapter, Current, finished.

B.58 Ice::ServerNotFoundException

综述

exception ServerNotFoundException

如果无法找到服务器，就会引发该异常。

B.59 Ice::SocketException

综述

local exception SocketException extends SyscallException

这个异常是对 SyscallException 的特化，用于 socket 错误。

派生异常

ConnectFailedException, ConnectionLostException.

B.60 Ice::Stats

综述

local interface Stats

Ice 用来报告统计信息的接口，比如发送或接收了多少数据。应用必须提供自己的 Stats：实现这个接口，并在通信器中安装它。

用于

Communicator::getStats, Communicator::setStats.

bytesReceived

```
void bytesReceived(string protocol, int num);
```

用于报告已收到多少数据的回调操作。

参数

protocol

是通过什么协议收到数据的 (例如， "tcp"、 "udp" 或 "ssl")。

num

收到了多少字节。

bytesSent

```
void bytesSent(string protocol, int num);
```

用于报告发送了多少数据的回调操作。

参数

protocol

是通过什么协议发送数据的 (例如， "tcp"、 "udp" 或 "ssl")。

num

发送了多少字节。

B.61 Ice::SyscallException

综述

```
local exception SyscallException
```

如果在服务器或客户进程中发生错误，就会引发该异常。发生系统异常的可能原因有很多。要了解详细的原因，应该检查 error。

派生异常

SocketException.

error

```
int error;
```

与系统异常对应的错误号。对于 C++ 和 Unix，这和 errno 是等价的。对于 C++ 和 Windows，这是 GetLastError() 或 WSAGetLastError() 返回的值。

B.62 Ice::TimeoutException

综述

```
local exception TimeoutException
```

超时异常。

派生异常

CloseTimeoutException, ConnectTimeoutException, ConnectionTimeoutException.

B.63 Ice::TwowayOnlyException

综述

```
local exception TwowayOnlyException
```

如果试图通过 ice_oneway、 ice_batchOneway、 ice_datagram 或 ice_batchDatagram 调用某操作，而该操作有返回值、 out 参数或异常规范，就会引发该异常。

operation

```
string operation;
```

被调用的操作的名字。

B.64 Ice::UnknownException

综述

```
local exception UnknownException
```

如果在调用服务器上的操作时引发了未知异常，就会引发该异常。例如，在 C++ 里，如果服务器抛出了不是直接或间接派生自 Ice::LocalException 或 Ice::UserException 的异常，就引发这个异常。

派生异常

UnknownLocalException, UnknownUserException.

unknown

```
string unknown;
```

未知异常的文本表示。取决于服务器的安全策略，可能会、也可能不会设置这个域。有些服务器可能会出于调试的目的而把这个信息给客户，而另一些服务器则可能不希望透露服务器的内部信息。

B.65 Ice::UnknownLocalException

Overview

```
local exception UnknownLocalException extends UnknownException
```

如果在调用服务器上的操作时引发了本地异常，就会引发该异常。因为 Ice 协议不会传送本地异常，对于服务器引发的所有本地异常，客户都将收到 UnknownLocalException。这条规则的唯一一个例外是所有派生自 RequestFailedException 的异常，即使它们被声明成 local 异常， Ice 协议也仍然会传送它们。

B.66 Ice::UnknownMessageException

综述

```
local exception UnknownMessageException extends ProtocolException
```

这个异常是对 ProtocolException 的特化，表明收到了未知协议的消息。

B.67 Ice::UnknownReplyStatusException

综述

```
local exception UnknownReplyStatusException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，表明收到了未知的答复状态。

B.68 Ice::UnknownRequestIdException

综述

```
local exception UnknownRequestIdException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，表明收到的响应含有未知的请求 id。

B.69 Ice::UnknownUserException

综述

```
local exception UnknownUserException extends UnknownException
```

如果在调用服务器上的操作时引发的用户异常没有在异常的 throws 子句中声明，就会引发该异常。 Ice 不会把这样的未声明异常从服务器传到客户，客户收到的只是 UnknownUserException。为了避免违反操作的型构所建立的合约，上述做法是必需的：只有在 throws 子句中声明了的本地异常和用户异常才能被引发。

B.70 Ice::UnmarshalOutOfBoundsException

综述

```
local exception UnmarshalOutOfBoundsException extends
MarshalException
```

这个异常是对 MarshalException 的特化，如果在解编过程中出现了越界情况，就会引发该异常。

B.71 Ice::UnsupportedEncodingException

综述

```
local exception UnsupportedEncodingException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，表明遇到了不支持的数据编码版本。

badMajor

```
int badMajor;
```

不支持的编码的大版本号。

badMinor

```
int badMinor;
```

不支持的编码的小版本号。

major

```
int major;
```

这个 Ice 版本支持的编码的大版本号。

minor

```
int minor;
```

这个 Ice 版本能支持的编码的最高小版本号。

B.72 Ice::UnsupportedProtocolException

综述

```
local exception UnsupportedProtocolException extends
ProtocolException
```

这个异常是对 ProtocolException 的特化，表明遇到了不支持的协议版本。

badMajor

```
int badMajor;
```

不支持的协议的大版本号。

badMinor

```
int badMinor;
```

不支持的协议的小版本号。

major

```
int major;
```

这个 Ice 版本支持的协议的大版本号。

minor

```
int minor;
```

这个 Ice 版本能支持的编码的最高小版本号。

B.73 Ice::VersionMismatchException

综述

```
local exception VersionMismatchException
```

如果 Ice 库版本与 Ice 头文件版本不匹配，就会引发该异常。

B.74 Freeze

综述

```
module Freeze
```

Freeze 为 Ice servant 提供了自动持久功能。为使速度最大化， Freeze 提供了一种二进制格式，为使灵活性最大化， Freeze 还提供了一种 XML 数据格式。当持久数据的 Slice 描述发生变化时，可以迁移用 XML 数据格式描述的 Freeze 数据库。

Key

```
sequence<byte> Key;
```

数据库键，用字节序列表示。

Value

```
sequence<byte> Value;
```

数据库值，用字节序列表示。

B.75 Freeze::Connection

综述

```
local interface Connection
```

与数据库 (Berkeley DB 的数据库环境) 的连接。如果你想要在多个线程中并发使用某个连接，你需要序列化对这个连接的访问。

beginTransaction

```
Transaction beginTransaction();
```

创建一个新事务。同时只能有一个事务与一个连接相关联。

返回值

新创建的事务。

异常

如果已经有事务与这个连接相关联，就引发 TransactionAlreadyInProgressException。

close

```
void close();
```

关闭这个连接。如果有相关联的事务，该事务将被回滚。

currentTransaction

```
Transaction currentTransaction();
```

返回与这个连接相关联的事务。

返回值

当前事务（如果有），否则返回 null。

getCommunicator

```
::Ice::Communicator getCommunicator();
```

返回与这个连接相关联的通信器。

getName

```
string getName();
```

连接到的系统的名字 (例如， Berkeley DB environment)。

B.76 Freeze::DatabaseException

综述

```
local exception DatabaseException
```

Freeze 数据库异常。

派生异常

DeadlockException, NotFoundException.

参见

DB, Evictor, Connection.

message

```
string message;
```

描述异常原因的消息。

B.77 Freeze::DeadlockException

综述

```
local exception DeadlockException extends DatabaseException
```

Freeze 数据库死锁异常。应用可以通过中止并重试事务来对这个异常做出反应。

B.78 Freeze::EmptyFacetPathException

综述

```
local exception EmptyFacetPathException
```

如果传给 Evictor::addFacet 或 Evictor::removeFacet 的是空的 ::Ice::FacetPath，就会引发该异常。

B.79 Freeze::Evictor

综述

```
local interface Evictor extends ::Ice::ServantLocator
```

一个自动的 Ice 对象持久管理器，基于逐出器模式。逐出器是一种 servant 定位器实现，会把其对象的持久状态存储在数据库中。向住处器注册的对象可以有任意多个，但同一时刻只能有一定数目的活动 servant（该数目可以配置）。这些活动的 servant 驻留在队列中；当有新的 servant 被激活时，最近最少使用的 servant 将最先被逐出。

参见

ServantInitializer.

addFacet

```
void addFacet(::Ice::Identity identity, ::Ice::FacetPath facet,
Object servant);
```

给这个对象添加一个新的持久 facet。

参数

`identity`

目标 Ice 对象的标识。

`facet`

facet 路径。

`servant`

Ice 对象的 servant。

异常

`DatabaseException`

如果数据库出错，就会引发该异常。

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

`EmptyFacetPathException`

如果 facet 路径是空的，就会引发该异常。

参见

`::Ice::Identity`, `removeFacet`.

createObject

```
void createObject(::Ice::Identity identity, Object servant);
```

为这个逐出器创建一个新的 Ice 对象。传给这个操作的 servant 的状态将保存在逐出器的持久存储器中。如果对象已经存在，它将会被更新。

参数

`identity`

要创建的 Ice 对象的标识。

`servant`

Ice 对象的 servant。

异常

`DatabaseException`

如果数据库出错，就会引发该异常。

EvictorDeactivatedException

如果逐出器已解除激活，就会引发该异常。

参见

::Ice::Identity, destroyObject.

destroyObject

```
void destroyObject(::Ice::Identity identity);
```

把某个 Ice 对象从逐出器的持久存储器中移除，永久性地销毁它。如果该对象不存在，这个操作就什么也不做。

参数

identity

要销毁的 Ice 对象的标识。

异常

DatabaseException

如果数据库出错，就会引发该异常。

EvictorDeactivatedException

如果逐出器已解除激活，就会引发该异常。

参见

::Ice::Identity, createObject.

getIterator

```
EvictorIterator getIterator(int batchSize, bool loadServants);
```

获取逐出器管理的标识的迭代器。

参数

batchSize

在内部，迭代器会按照 batchSize 指定的尺寸，成批获取标识。选择很小的 batchSize，可能会对性能起反作用。

`loadServants`

如果为真，就尝试在逐出器中加载对应的 servant。如果在获取成批标识时，逐出器在进行保存， servant 就有可能不会被加载。

返回值

新的迭代器。

异常

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

getSize

```
int getSize();
```

获取逐出器的 servant 队列的尺寸。

返回值

servant 队列的尺寸。

Exceptions

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

参见

`setSize`.

hasObject

```
bool hasObject(::Ice::Identity ident);
```

如果逐出器在管理给定的标识，就返回真。

返回值

如果逐出器在管理给定的标识，就返回真，否则返回假。

异常

DatabaseException

如果数据库出错，就会引发该异常。

EvictorDeactivatedException

如果逐出器已解除激活，就会引发该异常。

installServantInitializer

```
void installServantInitializer(ServantInitializer initializer);
```

为这个逐出器安装 servant 初始化器。

参数

initializer

要安装的 servant 初始化器。后续的调用会替代先前设置的值。 null 值将移除已经安装的 servant 初始化器。

异常

EvictorDeactivatedException

如果逐出器已解除激活，就会引发该异常。

参见

ServantInitializer.

removeAllFacets

```
void removeAllFacets(::Ice::Identity identity);
```

永久性地从这个对象中移除所有 facet。

参数

identity

目标 Ice 对象的标识。

异常

DatabaseException

如果数据库出错，就会引发该异常。

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

参见

`::Ice::Identity`, `removeFacet`.

removeFacet

```
Object removeFacet(::Ice::Identity identity, ::Ice::FacetPath
facet);
```

永久性地从该对象中移除这个 facet。

参数

`identity`

目标 Ice 对象的标识。

`facet`

facet 路径。

返回值

被移除的 facet。

异常

`DatabaseException`

如果数据库出错，就会引发该异常。

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

`EmptyFacetPathException`

如果 facet 路径是空的，就会引发该异常。

参见

`::Ice::Identity`, `addFacet`.

setSize

```
void setSize(int sz);
```

设置逐出器的 servant 队列的尺寸。这是逐出器中的活动 servant 的最大数目。如果你想把队列尺寸设成负值，你的请求会被忽略。

参数

`sz`

servant 队列的尺寸。如果逐出器目前在其队列里持有的 servant 多于 `setSize`，它会逐出足够的 servant，以适应新的尺寸。注意，如果新的队列尺寸小于正在对请求进行服务的 servant 的数目，这个操作可能会阻塞。在这种情况下，这个操作会进行等待，直到有足够多的 servant 完成其请求。

异常

`EvictorDeactivatedException`

如果逐出器已解除激活，就会引发该异常。

参见

`getSize`.

B.80 Freeze::EvictorDeactivatedException

综述

```
local exception EvictorDeactivatedException
```

如果逐出器已解除激活，就会引发该异常。

B.81 Freeze::EvictorIterator

综述

```
local interface EvictorIterator
```

逐出器管理的对象的迭代器。注意， EvictorIterator 不是线程安全的：应用需要序列化对给定的 EvictorIterator 的访问，例如，只在一个线程中使用它。

用于

Evictor::getIterator.

参见

Evictor.

hasNext

```
bool hasNext();
```

确定迭代器是否还有更多元素。

返回值

如果迭代器是否还有更多元素，返回真，否则返回假。

异常

DatabaseException

如果在获取成批对象的过程中数据库出错，就会引发该异常。

next

```
::Ice::Identity next();
```

获取迭代过程中的下一个标识。

返回值

迭代过程中的下一个标识。

异常

NoSuchElementException

如果在迭代过程中已没有元素，就会引发该异常。

DatabaseException

如果在获取成批对象的过程中数据库出错，就会引发该异常。

B.82 Freeze::EvictorStorageKey

综述

```
struct EvictorStorageKey
```

逐出器持久映射表的键。

facet

```
::Ice::FacetPath facet;
```

identity

```
::Ice::Identity identity;
```

B.83 Freeze::InvalidPositionException

综述

```
local exception InvalidPositionException
```

这个 Freeze 迭代器没有处在有效的位置上，例如，该位置已被去除。

B.84 Freeze::NoSuchElementException

综述

```
local exception NoSuchElementException
```

如果在迭代过程中已没有元素，就会引发该异常。

B.85 Freeze::NotFoundException

综述

```
local exception NotFoundException extends DatabaseException
```

Freeze 数据库异常，表明没有找到某条数据库记录。

B.86 Freeze::ObjectRecord

综述

```
struct ObjectRecord
```

逐出器使用了一个把 EvictorStorageKey 映射到 ObjectRecord 的映射表作为其持久存储器。

servant

```
Object servant;
```

stats

```
Statistics stats;
```

B.87 Freeze::ServantInitializer

综述

```
local interface ServantInitializer
```

servant 初始化器要安装在逐出器中，向应用提供 “进行定制的 servant 初始化” 的机会。

用于

Evictor::installServantInitializer.

参见

Evictor.

initialize

```
void initialize(::Ice::ObjectAdapter adapter, ::Ice::Identity
identity, Object servant);
```

只要逐出器创建新的 servant，就会调用这个操作。藉此，应用代码可以在逐出器创建 servant 并恢复其持久状态之后，进行定制的 servant 初始化。

参数

adapter

逐出器被安装到这个适配器。

identity

被创建的 servant 的 Ice 对象的标识。

servant

要初始化的 servant。

参见

::Ice::Identity.

B.88 Freeze::Statistics

综述

struct Statistics

逐出器会维护关于每个对象的统计信息。

用于

ObjectRecord::stats.

avgSaveTime

```
long avgSaveTime;
```

保存所用的平均时间，以毫秒为单位。

creationTime

```
long creationTime;
```

对象的创建时间，以毫秒为单位，从 1970 年 1 月 1 日 0:00 开始计算。

lastSaveTime

```
long lastSaveTime;
```

对象的最后保存时间，以毫秒为单位，相对于 creationTime 计算。

B.89 Freeze::Transaction

综述

```
local interface Transaction
```

事务。如果你想要在多个线程中并发地使用事务，你需要序列化对该事务的访问。

用于

Connection::beginTransaction, Connection::currentTransaction.

commit

```
void commit();
```

提交这个事务。

rollback

```
void rollback();
```

回滚这个事务。

B.90 Freeze::TransactionAlreadyInProgressException

综述

```
local exception TransactionAlreadyInProgressException
```

B.91 IceBox

综述

```
module IceBox
```

IceBox 是一个专门用于 Ice 应用的应用服务器。 Ice 可以轻松地运行和管理作为 DLL、共享库或 Java 类而动态加载的 Ice 服务。

B.92 IceBox::FailureException

综述

```
local exception FailureException
```

表明失败了。例如，服务在初始化过程中遇到了错误，或是服务管理器无法加载某个服务的可执行程序。

reason

```
string reason;
```

失败原因。

B.93 IceBox::FreezeService

综述

```
local interface FreezeService extends ServiceBase
```

由 ServiceManager 管理的 Freeze 应用服务。

参见

ServiceBase.

start

```
void start(string name, ::Ice::Communicator communicator,
::Ice::StringSeq args, string envName);
```

启动服务。给定的通信器由 ServiceManager 创建，供服务使用。取决于服务配置，其他服务也可能会使用这个通信器。数据库环境由 ServiceManager 创建，供这个服务专用。

ServiceManager 拥有该通信器和数据库环境，并会负责销毁它们。

参数

name

服务名，由配置确定。

communicator

供该服务使用的通信器。

args

没有转换成属性的服务参数。

envName

Freeze 数据库环境的名字。

异常

FailureException

如果 start 失败，就会引发该异常。

B.94 IceBox::Service

综述

```
local interface Service extends ServiceBase
```

标准的应用服务，由 ServiceManager 管理。

参见

ServiceBase.

start

```
void start(string name, ::Ice::Communicator communicator,
::Ice::StringSeq args);
```

启动服务。给定的通信器由 ServiceManager 创建，供服务使用。取决于服务配置，其他服务也可能会使用这个通信器。

ServiceManager 拥有该通信器，并会负责销毁它。

参数

name

服务名，由配置确定。

communicator

供该服务使用的通信器。

args

没有转换成属性的服务参数。

异常

FailureException

如果 start 失败，就会引发该异常。

B.95 IceBox::ServiceBase

综述

`local interface ServiceBase`

由 ServiceManager 管理的应用服务的基接口。

派生类与接口

FreezeService, Service.

参见

ServiceManager, Service, FreezeService.

stop

`void stop();`

停止服务。

B.96 IceBox::ServiceManager

综述

`interface ServiceManager`

管理一组 Service 实例。

参见

Service.

shutdown

`void shutdown();`

关闭所有服务。这将致使所有已配置的服务的 Service::stop 被调用。

B.97 IcePack

综述

```
module IcePack
```

IcePack 是一个服务器激活和部署工具。在异种计算机网络中部署应用是一项复杂的任务， IcePack 能够对其进行简化。

B.98 IcePack::AdapterDeploymentException

综述

```
exception AdapterDeploymentException extends DeploymentException
```

如果在适配器注册过程中出错，就会引发该异常。

id

```
string id;
```

无法注册的适配器的 id。

B.99 IcePack::AdapterNotExistException

综述

```
exception AdapterNotExistException
```

如果适配器不存在，就会引发该异常。

B.100 IcePack::Admin

Overview

`interface Admin`

IcePack 管理接口。

警告: 允许访问这个接口会带来安全风险！进一步的信息，请参考 IcePack 的文档。

addApplication

```
void addApplication(string descriptor, ::Ice::StringSeq targets)
throws DeploymentException;
```

把一个应用添加到 IcePack。一个应用是一组服务器。

参数

`descriptor`

应用描述符。

`targets`

要部署的可选目标。目标是由多个目标名组成的列表，用句点分隔开。例如，如果指定的是 "server1.service1.debug"，就会部署 "server1" 中的 "service1" 的 "debug" 目标。

异常

`DeploymentException`

如果应用部署失败，就会引发该异常。

参见

`removeApplication`.

addObject

```
void addObject(Object* obj) throws ObjectExistsException,
ObjectDeploymentException;
```

把一个对象添加到对象注册表。 IcePack 会调用给定代理上的 `ice_id` 来获取对象类型。对象必须要能到达。

参数

obj

要添加到注册表中的对象。

异常

ObjectDeploymentException

如果在注册这个对象时出错，就会引发该异常。如果因为对象无法到达而造成对象的类型无法确定，就可能会发生这样的情况。

ObjectExistsException

如果对象已经注册，就会引发该异常。

addObjectWithType

```
void addObjectWithType(Object* obj, string type) throws
ObjectExistsException;
```

把一个对象添加到对象注册表，并显式地指定其类型。

参数

obj

要添加到注册表中的对象。

type

对象类型。

异常

ObjectExistsException

如果对象已经注册，就会引发该异常。

addServer

```
void addServer(string node, string name, string path, string
libraryPath, string descriptor, ::Ice::StringSeq targets) throws
DeploymentException, NodeUnreachableException;
```

把一个服务器添加到 IcePack 节点。

参数

`node`

服务器将被部署到这个节点。

`name`

服务器名。

`path`

服务器路径。对于 C++ 服务器，这是可执行程序的路径。对于 C++ icebox，这是 C++ icebox 可执行程序的路径——如果该路径是空的， IcePack 将依靠 PATH 来找到它。对于 Java 服务器或 Java icebox，这是 `java` 命令的路径—— 如果该路径是空的，IcePack 将依靠 PATH 来找到它。

`librarypath`

为 C++ 服务器指定 `LD_LIBRARY_PATH` 的值，或为 Java 服务器指定 `CLASSPATH` 的值。

`descriptor`

服务器部署描述符。

`targets`

要部署的可选目标。目标是由多个目标名组成的列表，用句点分隔开。例如，如果指定的是 "server1.service1.debug"，就会部署 "server1" 中的 "service1" 的 "debug" 目标。

异常

`DeploymentException`

如果服务器部署失败，就会引发该异常。

`NodeUnreachableException`

如果节点无法到达，就会引发该异常。

参见

`removeServer`.

getAdapterEndpoints

```
string getAdapterEndpoints(string id) throws
AdapterNotExistException, NodeUnreachableException;
```

获取适配器的端点列表。

参数

`id`

适配器 id。

返回值

串化的适配器端点。

异常

`AdapterNotExistException`

如果找不到适配器，就会引发该异常。

getAllAdapterIds

```
::Ice::StringSeq getAllAdapterIds();
```

获取向 IcePack 注册过的所有适配器 id。

返回值

适配器的所有 ids。

getAllNodeNames

```
::Ice::StringSeq getAllNodeNames();
```

获取目前已经注册的所有 IcePack 节点。

返回值

节点名。

getAllServerNames

```
::Ice::StringSeq getAllServerNames();
```

获取向 IcePack 注册过的所有服务器的名字。

返回值

服务器名。

参见

getServerDescription, getServerState.

getServerActivation

```
ServerActivation getServerActivation(string name) throws
ServerNotExistException, NodeUnreachableException;
```

获取服务器的激活模式。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

服务器的激活模式。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

getServerDescription, getServerState, getAllServerNames.

getServerDescription

```
ServerDescription getServerDescription(string name) throws
ServerNotExistException, NodeUnreachableException;
```

获取服务器的描述。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

服务器的描述。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

getServerState, getServerPid, getAllServerNames.

getServerPid

```
int getServerPid(string name) throws ServerNotExistException,
NodeUnreachableException;
```

获取服务器的系统进程 id。进程 id 和操作系统是相关的。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

服务器进程 id。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

getServerDescription, getServerState, getAllServerNames.

getServerState

```
ServerState getServerState(string name) throws
ServerNotExistException, NodeUnreachableException;
```

获取服务器的状态。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

服务器状态。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

getServerDescription, getServerPid, getAllServerNames.

pingNode

```
bool pingNode(string name) throws NodeNotExistException;
```

Ping 某个 IcePack 节点，看它是否是活动的。

返回值

如果 ping 成功，返回真，否则返回假。

removeApplication

```
void removeApplication(string descriptor) throws
DeploymentException;
```

从 IcePack 中移除指定应用。

参数

descriptor

应用描述符。

参见

addApplication.

removeObject

```
void removeObject(Object* obj) throws ObjectNotExistException;
```

从对象注册表中移除指定对象。

参数

obj

要从注册表中移除的对象。

异常

ObjectNotExistException

如果找不到这个对象，就会引发该异常。

removeServer

```
void removeServer(string name) throws DeploymentException,
ServerNotExistException, NodeUnreachableException;
```

从 IcePack 节点中移除指定的服务器。

参数

name

必须与 ServerDescription::name 的值吻合。

异常

DeploymentException

如果服务器部署器没有能移除服务器，就会引发该异常。

ServerNotExistException

如果找不到这个服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

addServer.

sendSignal

```
void sendSignal(string name, string signal) throws
ServerNotExistException, NodeUnreachableException,
BadSignalException;
```

发信号给服务器。

Parameters

name

必须与 ServerDescription::name 的值吻合。

signal

要发出的信号，例如， SIGTERM 或 15。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

BadSignalException

如果目标服务器无法识别这个信号，就会引发该异常。

setServerActivation

```
void setServerActivation(string name, ServerActivation mode)
throws ServerNotExistException, NodeUnreachableException;
```

设置服务器的激活模式。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

服务器激活模式。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

参见

getServerDescription, getServerState, getAllServerNames.

shutdown

```
void shutdown();
```

关闭 IcePack 注册表。

shutdownNode

```
void shutdownNode(string name) throws NodeNotExistException;
```

关闭指定的 IcePack 节点。

startServer

```
bool startServer(string name) throws ServerNotExistException,
NodeUnreachableException;
```

启动指定的服务器。

参数

name

必须与 ServerDescription::name 的值吻合。

返回值

如果服务器成功启动，就返回真，否则返回假。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

stopServer

```
void stopServer(string name) throws ServerNotExistException,
NodeUnreachableException;
```

停止指定的服务器。

参数

name

必须与 ServerDescription::name 的值吻合。

异常

ServerNotExistException

如果找不到服务器，就会引发该异常。

NodeUnreachableException

如果节点无法到达，就会引发该异常。

writeMessage

```
void writeMessage(string name, string message, int fd) throws
ServerNotExistException, NodeUnreachableException;
```

把消息写往服务器的 stdout 或 stderr。

参数

name

必须与 ServerDescription::name 的值吻合。

`message`

要写出的消息。

`fd`

1 是 stdout， 2 是 stderr。

异常

`ServerNotExistException`

如果找不到服务器，就会引发该异常。

`NodeUnreachableException`

如果节点无法到达，就会引发该异常。

B.101 IcePack::BadSignalException

综述

`exception BadSignalException`

如果发往服务器的是未知信号，就会引发该异常。

B.102 IcePack::DeploymentException

综述

`exception DeploymentException`

所有种类的部署错误的基异常。

派生异常

`AdapterDeploymentException`, `ObjectDeploymentException`, `ParserDeploymentException`, `ServerDeploymentException`.

component

```
string component;
```

造成部署失败的组件的路径。路径由组件名组成，用句点分隔。起头总是节点名，后面跟着服务器名，最后是服务名。

reason

```
string reason;
```

失败原因。

B.103 IcePack::NodeNotExistException

综述

```
exception NodeNotExistException
```

如果节点不存在，就会引发该异常。

B.104 IcePack::NodeUnreachableException

综述

```
exception NodeUnreachableException
```

如果节点无法到达，就会引发该异常。

B.105 IcePack::ObjectDeploymentException

综述

```
exception ObjectDeploymentException extends DeploymentException
```

如果在对象注册过程中出错，就会引发该异常。

proxy

```
Object* proxy;
```

无法注册的对象。

B.106 IcePack::ObjectExistsException

综述

```
exception ObjectExistsException
```

如果对象已经存在，就会引发该异常。

B.107 IcePack::ObjectNotExistException

综述

```
exception ObjectNotExistException
```

如果对象不存在，就会引发该异常。

B.108 IcePack::ParserDeploymentException

综述

```
exception ParserDeploymentException extends DeploymentException
```

如果在解析某个组件的 XML 描述符时出错，就会引发该异常。

B.109 IcePack::Query

综述

```
interface Query
```

IcePack 查询接口。想要查找对象的 Ice 客户可以访问这个接口。

findAllObjectsWithType

```
::Ice::ObjectProxySeq findAllObjectsWithType(string type) throws
ObjectNotExistException;
```

找出所有具有指定类型的对象。

参数

`type`

对象类型。

返回值

各对象的代理。

异常

`ObjectNotExistException`

如果找不到对象，就会引发该异常。

findObjectById

```
Object* findObjectById(::Ice::Identity id) throws
ObjectNotExistException;
```

根据标识查找对象。

参数

`id`

标识。

返回值

对象代理。

异常

ObjectNotExistException

如果找不到对象，就会引发该异常。

findObjectByType

```
Object* findObjectByType(string type) throws
ObjectNotExistException;
```

根据类型查找对象。

参数

type

对象类型。

返回值

对象代理。

异常

ObjectNotExistException

如果找不到对象，就会引发该异常。

B.110 IcePack::ServerActivation

综述

```
enum ServerActivation
```

服务器的激活模式。

用于

Admin::getServerActivation, Admin::setServerActivation.

OnDemand

OnDemand

如果客户请求使用服务器的某一个适配器端点，而该服务器还没有运行，它就会被随需激活。

Manual

Manual

服务器通过管理接口手工激活。

B.111 IcePack::ServerDeploymentException

综述

exception ServerDeploymentException extends DeploymentException

如果在部署服务器时出错，就会引发该异常。

server

string server;

无法部署的服务器的名字。

B.112 IcePack::ServerDescription

综述

struct ServerDescription

服务器的描述。

用于

Admin::getServerDescription.

args

```
::Ice::StringSeq args;
```

服务器的参数。

descriptor

```
string descriptor;
```

用于部署服务器的部署描述符的路径。

envs

```
::Ice::StringSeq envs;
```

服务器的环境变量。

name

```
string name;
```

服务器名。

node

```
string node;
```

服务器要部署到的节点的名字。

path

```
string path;
```

服务器的路径。

pwd

```
string pwd;
```

服务器的工作目录。

serviceManager

```
::IceBox::ServiceManager* serviceManager;
```

如果服务器是 IceBox 服务，这是 IceBox 服务管理器的代理，否则就是 null 代理。

targets

```
::Ice::StringSeq targets;
```

用于部署服务器的目标。

B.113 IcePack::ServerNotExistException

综述

```
exception ServerNotExistException
```

如果服务器不存在，就会引发该异常。

B.114 IcePack::ServerState

综述

```
enum ServerState
```

用于表示服务器状态的枚举。

用于

Admin::getServerState.

Inactive

```
Inactive
```

服务器没有在运行。

Activating

`Activating`

服务器正在激活，如果服务器 fork 成功，将变成活动状态，如果失败，将变成 Inactive 状态。

Active

`Active`

服务器在运行中。

Deactivating

`Deactivating`

服务器正在解除激活。

Destroying

`Destroying`

服务器正在销毁。

Destroyed

`Destroyed`

服务器已被销毁。

B.115 IceSSL

综述

`module IceSSL`

IceSSL 是 Ice 核心的一种动态的 SSL 传输插件。它使用工业标准 SSL 协议，提供了身份认证、加密、消息完整性等特性。

B.116 IceSSL::CertificateException

综述

`local exception CertificateException extends SslException`

与公钥证书相关的所有异常的根异常。

派生异常

`CertificateParseException`, `CertificateSignatureException`, `CertificateSigningException`.

B.117 IceSSL::CertificateKeyMatchException

综述

```
local exception CertificateKeyMatchException extends
ContextException
```

在把公钥和私钥对加载进 `Context` 时，加载已成功，但私钥和公钥（证书）不匹配。

B.118 IceSSL::CertificateLoadException

综述

`local exception CertificateLoadException extends ContextException`

在把来自内存缓冲区或文件的证书加载进 `Context` 时出了问题。

B.119 IceSSL::CertificateParseException

综述

```
local exception CertificateParseException extends
CertificateException
```

IceSSL 无法把公钥证书解析成底层 SSL 实现可以使用的形式。

B.120 IceSSL::CertificateSignatureException

综述

```
local exception CertificateSignatureException extends
CertificateException
```

对新签署的临时 RSA 证书的签名校验失败了。

B.121 IceSSL::CertificateSigningException

综述

```
local exception CertificateSigningException extends
CertificateException
```

在生成临时 RSA 证书的过程中，证书签署出了问题。

B.122 IceSSL::CertificateVerificationException

综述

```
local exception CertificateVerificationException extends
ShutdownException
```

在 SSL 握手的证书校验阶段出了问题。目前，只有服务器连接会抛出这个异常。

B.123 IceSSL::CertificateVerifier

综述

```
local interface CertificateVerifier
```

所有要定义应用特有的证书校验规则的类都要从 CertificateVerifier 派生。派生自 CertificateVerifier 的类实例会在 SSL 握手过程中评估这些规则。在派生接口中定义哪些接口取决于底层 SSL 实现的要求。通过 Plugin 可以获取缺省的证书校验器实现。因为这只是一个供继承用的基类，所以它没有定义方法。

用于

Plugin::getDefaultCertVerifier, Plugin::getSingleCertVerifier, Plugin::setCertificateVerifier.

参见

Plugin.

setContext

```
void setContext(ContextType type);
```

设置这个证书校验器的上下文类型。

参数

type

正在使用这个证书校验器的上下文， Client、 Server、或 ClientServer。

B.124 IceSSL::CertificateVerifierTypeException

综述

```
local exception CertificateVerifierTypeException extends
SslException
```

这个异常表明，所提供的证书校验器不是从适当的基类派生的，因此，没有提供适当的接口。

B.125 IceSSL::ConfigParseException

综述

```
local exception ConfigParseException extends SslException
```

这个异常表明，在解析 SSL 配置文件时、或尝试定位配置文件时出了问题。这个异常可用于表明你的 ::Ice::Communicator 的 IceSSL.Client.Config、 IceSSL.Server.Config、IceSSL.Client.CertPath 或 IceSSL.Server.CertPath 属性有问题。

B.126 IceSSL::ConfigurationLoadingException

综述

```
local exception ConfigurationLoadingException extends SslException
```

这个异常表明，你试图加载某个 Context 的配置，但却没有设置属性，指定这个 Context 的 SSL 配置文件。检查适当的属性值（IceSSL.Client.Config 或 IceSSL.Server.Config）。

B.127 IceSSL::ContextException

综述

```
local exception ContextException extends SslException
```

在设置 Context 时遇到了问题，比如在加载证书和密钥时有问题，或是调用了还没有初始化的 Context 上的问题。

派生异常

CertificateKeyMatchException, CertificateLoadException, ContextInitializationException, ContextNotConfiguredException, PrivateKeyLoadException, TrustedCertificateAddException, UnsupportedContextException.

B.128 IceSSL::ContextInitializationException

综述

```
local exception ContextInitializationException extends
ContextException
```

在初始化底层 SSL 实现的上下文结构时出了问题。

B.129 IceSSL::ContextNotConfiguredException

综述

```
local exception ContextNotConfiguredException extends
ContextException
```

如果你试图使用还没有进行过配置的 Context，就会引发该异常。

B.130 IceSSL::ContextType

综述

enum ContextType

一个 Plugin 可以充当 Client、 Server，或两者都是（ClientServer）。为了应付 Client 或 Server 的角色，在 Plugin 内部设置了一个 Context。这个 Context 表示的是与插件的角色相关的配置。为了标识 Context，有些 Plugin 操作需要有一个 ContextType 参数。

用于

CertificateVerifier::setContext, Plugin::addTrustedCertificate, Plugin::addTrustedCertificateBase64, Plugin::configure, Plugin::loadConfig, Plugin::setCertificateVerifier, Plugin::setRSAKeys, Plugin::setRSAKeysBase64.

Client

Client

只选择 Client Context，不修改 Server。

Server

Server

只选择 Server Context，不修改 Client。

ClientServer

ClientServer

选择并改动 Client 和 Server Context。

B.131 IceSSL::Plugin

综述

```
local interface Plugin extends ::Ice::Plugin
```

SSL 插件的接口。这个接口通常用于在程序中完成对该插件的配置。

addTrustedCertificate

```
void addTrustedCertificate(ContextType cType, ::Ice::ByteSeq
certificate);
```

把一个受信的证书添加到插件的缺省证书存储器中。所提供的证书（以二进制 DER 格式传递）将被添加到信任列表中，这样，该证书、还有用其私钥签署的所有证书都会得到信任。这个方法只影响新的连接——已有的连接不受影响。

参数

`contextType`

受信证书将添加到该 `Context`(s)。

`certificate`

想要得到信任的证书，二进制 DER 格式。

addTrustedCertificateBase64

```
void addTrustedCertificateBase64(ContextType cType, string
certificate);
```

把一个受信的证书添加到插件的缺省证书存储器中。所提供的证书（按照 PEM 格式，以经过 Base64 编码的二进制 DER 格式传递）将被添加到信任列表中，这样，该证书、还有用其私钥签署的所有证书都会得到信任。这个方法只影响新的连接——已有的连接不受影响。

参数

`contextType`

受信证书将添加到该 `Context`(s)。

certificate

想要得到信任的证书，经过 Base64 编码的二进制 DER 格式。

configure

```
void configure(ContextType cType);
```

配置插件。如果插件处在没有配置的状态，取决于上下文的类型，它将会从 IceSSL.Server.Config 或 IceSSL.Client.Config 属性中加载其配置。在这个操作中，还会加载配置属性设置，其属性值将会替代配置文件中的属性值。

参数

contextType

要配置的 Context(s)。

getDefaultCertVerifier

```
CertificateVerifier getDefaultCertVerifier();
```

所有插件实例都会安装一个缺省的 CertificateVerifier 实例；这个操作用于获取该实例。

返回值

证书校验器。

getSingleCertVerifier

```
CertificateVerifier getSingleCertVerifier(::Ice::ByteSeq
certificate);
```

返回一个 CertificateVerifier 实例，它只接受一种证书，即包含在字节序列参数中的二进制 DER 编码所表示的 RSA 证书。如果你希望你的应用接受来自一方的连接，这个操作很有用。

在你的 SSL 配置文件中一定要使用 peer verifymode。

参数

certificate

DER 编码的 RSA 证书。

返回值

证书校验器。

loadConfig

```
void loadConfig(ContextType cType, string configFile, string
certPath);
```

使用给定的配置文件中的设置，为给定的 Context 配置插件。如果插件处在没有配置的状态，取决于上下文的类型，它将会从 IceSSL.Server.Config 或 IceSSL.Client.Config 属性中加载其配置。在这个操作中，还会加载配置属性设置，其属性值将会替代配置文件中的属性值。

参数

contextType

要配置的 Context。

configFile

含有 SSL 配置信息的文件。

certPath

loadConfig 中引用的证书的路径。

setCertificateVerifier

```
void setCertificateVerifier(ContextType cType, CertificateVerifier
certVerifier);
```

设置用于指定的 ContextType 角色的 CertificateVerifier。创建的所有插件 Context 都会安装缺省的 CertificateVerifier 对象。用这个操作可以换用别的 CertificateVerifier。这个方法只影响新的连接——已有的连接不受影响。

参数

contextType

在该 Context(s) 中安装证书校验器。

certVerifier

要安装的 CertificateVerifier。

参见

CertificateVerifier.

setRSAKeys

```
void setRSAKeys(ContextType cType, ::Ice::ByteSeq privateKey,
::Ice::ByteSeq publicKey);
```

设置 RSA 密钥，当在 ContextType 指定的上下文模式中进行操作时，插件将使用这些密钥。这个方法只影响新的连接——已有的连接不受影响。

参数

contextType

在该 Context(s) 中设置 / 替换 RSA 密钥。

privateKey

RSA 私钥，二进制 DER 格式。

publicKey

RSA 公钥，二进制 DER 格式。

setRSAKeysBase64

```
void setRSAKeysBase64(ContextType cType, string privateKey, string
publicKey);
```

设置 RSA 密钥，当在 ContextType 指定的上下文模式中进行操作时，插件将使用这些密钥。这个方法只影响新的连接——已有的连接不受影响。

参数

contextType

在该 Context(s) 中设置 / 替换 RSA 密钥。

privateKey

RSA 私钥， Base64 编码的二进制 DER 格式。

publicKey

RSA 公钥， Base64 编码的二进制 DER 格式。

B.132 IceSSL::PrivateKeyException

综述

`local exception PrivateKeyException extends SslException`

与私钥相关的所有异常的根异常。

派生异常

`PrivateKeyParseException`.

B.133 IceSSL::PrivateKeyLoadException

综述

`local exception PrivateKeyLoadException extends ContextException`

在把来自内存缓冲区或文件的私钥加载进 `Context` 时出了问题。

B.134 IceSSL::PrivateKeyParseException

综述

```
local exception PrivateKeyParseException extends
PrivateKeyException
```

IceSSL 无法把私钥解析成底层 SSL 实现可以使用的形式。

B.135 IceSSL::ProtocolException

综述

`local exception ProtocolException extends ShutdownException`

发生了违反 SSL 协议的问题，造成连接被关闭。

B.136 IceSSL::ShutdownException

综述

```
local exception ShutdownException extends SslException
```

这个异常表明，发生了造成 SSL 连接关闭的问题。

派生异常

CertificateVerificationException, ProtocolException.

B.137 IceSSL::SslException

综述

```
local exception SslException
```

这个异常是 Ice 中所有与安全有关的异常的基异常。它之所以是一个本地异常，是因为，如果安全出了问题，就不能再在连接上传送异常了。此外，许多异常包含的信息对外部的客户 / 服务器没有用处。

派生异常

CertificateException, CertificateVerifierTypeException, ConfigParseException, ConfigurationLoadingException, ContextException, PrivateKeyException, ShutdownException.

message

```
string message;
```

来自安全系统的相关信息，有助于更详细地解释该异常的性质。在有些情况下，它含有来自底层安全实现的信息，以及 / 或者调试跟踪信息。

B.138 IceSSL::TrustedCertificateAddException

综述

```
local exception TrustedCertificateAddException extends
ContextException
```

把证书添加到 Context 的受信证书存储器的尝试失败了。

B.139 IceSSL::UnsupportedContextException

综述

```
local exception UnsupportedContextException extends
ContextException
```

你调用的方法引用了该操作不支持的 ContextType。

B.140 Glacier

综述

```
module Glacier
```

Glacier 是 Ice 的防火墙解决方案。 Glacier 负责认证和过滤客户请求，并允许对客户进行安全的回调。与 IceSSL 相结合， Glacier 提供了一种既没有侵入性、又易于配置的安全解决方案。

B.141 Glacier::CannotStartRouterException

综述

```
exception CannotStartRouterException
```

如果路由器无法启动，就会引发该异常。

reason

```
string reason;
```

路由器无法启动的详细原因。

B.142 Glacier::NoSessionManagerException

综述

```
exception NoSessionManagerException
```

如果没有配置过 SessionManager 对象，就会引发该异常。

B.143 Glacier::PermissionDeniedException

综述

```
exception PermissionDeniedException
```

如果对路由器的访问被拒绝，就会引发该异常。

reason

```
string reason;
```

访问被拒绝的详细原因。

B.144 Glacier::PermissionsVerifier

综述

```
interface PermissionsVerifier
```

Glacier 路由器启动器的权限校验器。

checkPermissions

```
bool checkPermissions(string userId, string password, out string
reason);
```

检查用户是否有权访问路由器。

参数

`userId`

用户 id。

`password`

用户密码。

`reason`

访问被拒绝的原因。

返回值

如果允许访问，返回真，否则返回假。

B.145 Glacier::Router

综述

```
interface Router extends ::Ice::Router
```

Glacier 路由器接口。

createSession

```
Session* createSession() throws NoSessionManagerException;
```

创建一个新会话。路由器终止时会自动关闭会话。

返回值

新会话的代理。

异常

NoSessionManagerException

如果没有配置好的 SessionManager。

shutdown

```
void shutdown();
```

关闭路由器。

B.146 Glacier::Session

综述

```
interface Session
```

会话对象，其生命周期与 Router 连在一起。

参见

Router, SessionManager.

destroy

```
void destroy();
```

销毁会话。 Router 会在销毁时自动调用这个操作。

B.147 Glacier::SessionManager

综述

```
interface SessionManager
```

会话管理器，负责管理 Session 对象。新会话对象由 Router 对象创建。

参见

Session.

create

```
Session* create(string userId);
```

创建一个新的会话对象。

参数

userId

会话所用的用户 id。

返回值

新创建的会话的代理。

B.148 Glacier::Starter

综述

```
interface Starter
```

Glacier 路由器启动器。

startRouter

```
Router* startRouter(string userId, string password, out
::Ice::ByteSeq privateKey, out ::Ice::ByteSeq publicKey, out
::Ice::ByteSeq routerCert) throws PermissionDeniedException,
CannotStartRouterException;
```

启动一个新的 Glacier 路由器。如果密码不正确，或者用户没有访问权，就会引发 PermissionDeniedException。否则就会启动一个新路由器，并把该路由器的代理返回给调用者。

参数

`userId`

用户 id。

`password`

用户密码。

`privateKey`

供客户使用的 RSA 私钥 (DER 编码)(只适用于 SSL)。

`publicKey`

供客户使用的 RSA 公钥 (DER 编码)(只适用于 SSL)。

`routerCert`

路由器的受信证书 (只适用于 SSL)。

返回值

已经启动的路由器的代理。

异常

`PermissionDeniedException`

如果密码不正确，或者用户没有访问权，就会引发该异常。

B.149 IceStorm

Overview

`module IceStorm`

能够支持联盟的消息服务。与其他消息或事件服务不同， Ice 支持有类型的事件，也就是说，只需调用某个接口上的方法，就能在整个联盟中进行消息广播。

LinkInfoSeq

```
sequence<LinkInfo> LinkInfoSeq;
```

`LinkInfo` 对象序列。

用于

Topic::getLinkInfoSeq.

QoS

```
dictionary<string, string> QoS;
```

这个词典表示的是服务质量参数。

用于

Topic::subscribe.

参见

Topic::subscribe.

TopicDict

```
dictionary<string, Topic*> TopicDict;
```

从主题名到主题代理的映射。

Used By

TopicManager::retrieveAll.

B.150 IceStorm::LinkExists

综述

```
exception LinkExists
```

如果要创建的链接已经存在，就会引发该异常。

name

```
string name;
```

被链接的主题的名字。

B.151 IceStorm::LinkInfo

综述

struct LinkInfo

关于主题链接的信息。

用于

LinkInfoSeq.

cost

int cost;

通过这个链接所需的成本。

name

string name;

被链接的主题的名字。

theTopic

Topic* theTopic;

被链接的主题。

B.152 IceStorm::NoSuchLink

综述

exception NoSuchLink

如果想要移除的链接不存在，就会引发该异常。

name

```
string name;
```

不存在的连接的名字。

B.153 IceStorm::NoSuchTopic

Overview

```
exception NoSuchTopic
```

如果想要获取的主题不存在，就会引发该异常。

name

```
string name;
```

不存在的主题的名字。

B.154 IceStorm::Topic

综述

```
interface Topic
```

发布者会在特定的主题上发布信息。一个主题在逻辑上代表一种类型。

参见

TopicManager.

destroy

```
void destroy();
```

销毁主题。

getLinkInfoSeq

```
LinkInfoSeq getLinkInfoSeq();
```

获取关于当前链接的信息。

返回值

LinkInfo 对象序列。

getName

```
string getName();
```

获取这个主题的名字。

返回值

主题名。

参见

TopicManager::create.

getPublisher

```
Object* getPublisher();
```

获取这个主题的发布者对象的代理。为了把数据发布到某个主题，发布者要调用 getPublisher，然后转换成主题类型。对这个代理必须使用不进行检查的转换。

返回值

用于在这个主题上发布数据的代理。

link

```
void link(Topic* linkTo, int cost) throws LinkExists;
```

创建一个通向给定主题的链接。所有发给这个主题的事件也会发给 link。

参数

linkTo

要链接到的主题。

cost

被链接的主题的成本。

异常

LinkExists

如果通往相同主题的链接已经存在，就会引发该异常。

subscribe

```
void subscribe(QoS theQoS, Object* subscriber);
```

用给定的 QoS 参数订阅这个主题。如果给定的 subscribe 代理已经注册过，它将会被替换。

参数

qos

本次订阅的服务质量参数。

subscriber

订阅者的代理。

参见

unsubscribe.

unlink

```
void unlink(Topic* linkTo) throws NoSuchLink;
```

销毁从这个主题到给定主题的链接。

参数

linkTo

要销毁的链接所通向的主题。

异常

NoSuchLink

如果该链接不存在，就会引发该异常。

unsubscribe

```
void unsubscribe(Object* subscriber);
```

取消订阅。

参数

subscriber

要取消的订阅者的代理。

参见

subscribe.

B.155 IceStorm::TopicExists

综述

```
exception TopicExists
```

如果试图创建的主题已经存在，就会引发该异常。

name

```
string name;
```

已经存在的主题的名字。

B.156 IceStorm::TopicManager

综述

interface TopicManager

主题管理器负责管理主题和订阅主题。

参见

Topic.

create

Topic* create(string name) throws TopicExists;

创建新主题。主题名必须是唯一的，否则就会引发 TopicExists 。

参数

name

主题名。

返回值

主题实例的代理。

异常

TopicExists

如果已有同名主题存在，就会引发该异常。

retrieve

Topic* retrieve(string name) throws NoSuchTopic;

根据主题名获取主题。

参数

name

主题名。

返回值

主题实例的代理。

异常

NoSuchTopic

如果主题不存在，就会引发该异常。

retrieveAll

```
TopicDict retrieveAll();
```

获取这个主题管理器所管理的所有主题。

返回值

串 - 主题代理对的词典。

B.157 IcePatch

综述

```
module IcePatch
```

用于软件发布的修补服务。 IcePatch 能自动更新单个的文件，以及完整的目录层次。只有那些改动过的文件才会下载到客户机器上（使用压缩算法）。

FileDescSeq

```
sequence<FileDesc> FileDescSeq;
```

用于

Directory::getContents.

B.158 IcePatch::BusyException

综述

```
exception BusyException
```

B.159 IcePatch::Directory

综述

```
interface Directory extends File
```

getContents

```
FileDescSeq getContents() throws FileAccessException,
BusyException;
```

B.160 IcePatch::DirectoryDesc

综述

```
class DirectoryDesc extends FileDesc
```

dir

```
Directory* dir;
```

B.161 IcePatch::File

综述

interface File

派生类与接口

Directory, Regular.

describe

FileDesc describe() throws FileAccessException, BusyException;

B.162 IcePatch::FileAccessException

综述

exception FileAccessException

reason

string reason;

B.163 IcePatch::FileDesc

综述

class FileDesc

派生类与接口

DirectoryDesc, RegularDesc.

用于

File::describe, FileDescSeq.

md5

```
::Ice::ByteSeq md5;
```

B.164 IcePatch::Regular

综述

```
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

综述

```
class RegularDesc extends FileDesc
```

reg

```
Regular* reg;
```

附录 C
属性

如果在对属性的描述中没有另作陈述，所有属性的缺省值都是空串。如果该属性需要的是数字值，空串会被解释成零。

C.1 Ice 配置属性

Ice.Config

概要

`--Ice.Config --Ice.Config=1 --Ice.Config=`*`config_file`*

描述

这个属性必须用 `--Ice.Config`、`--Ice.Config=1`、或 `--Ice.Config=`*`config_file`* 选项在命令行上设置。

如果 `Ice.Config` 属性是空的，或被设成 1， Ice run time 会检查 `ICE_CONFIG` 环境变量的内容，取得配置文件的路径名。否则， `Ice.Config` 必须被设成配置文件的路径名 (可以是相对或绝对路径名)。其他的属性值会从指定的配置文件中读取。

配置文件语法

配置文件含有一些属性设置，每个设置占一行。属性设置的格式是

property_name= # Set property to the empty string or zero

property_name=*value* # Assign value to property

字符是注释的标志：# 字符及其后在同一行上的所有字符都会被忽略。如果某一行的第一个非空格字符是 #，这一整行都会被忽略。

配置文件的格式很自由：空格、制表符、换行符或者充当 token 分隔符，或者被忽略。

如果你在配置文件里面设置 Ice.Config 属性，你的设置会被忽略。

C.2 Ice 跟踪属性

Ice.Trace.GC

概要

Ice.Trace.GC=*num*

描述

垃圾收集器跟踪级别：

0	不跟踪垃圾收集器 (缺省)。
1	显示已收集的实例的总数、已检查的实例的总数、在收集器中化肥的时间 （以毫秒为单位），以及收集器的运行次数。
2	和 1 一样，但还在每次运行收集器时生成一条跟踪消息。

Ice.Trace.Network

概要

Ice.Trace.Network=*num*

描述

网络跟踪级别:

0	不跟踪网络 (缺省)。
1	跟踪连接的建立和关闭。
2	和 1 一样，但更详细。
3	和 2 一样，但另外还跟踪数据的传送。

Ice.Trace.Protocol

概要

`Ice.Trace.Protocol=`*`num`*

描述

协议跟踪级别:

0	不跟踪协议 (缺省)。
1	跟踪 Ice 的协议消息。

Ice.Trace.Retry

概要

`Ice.Trace.Retry=`*`num`*

描述

请求重试 （ request retry ）的跟踪级别:

0	不跟踪请求重试 (缺省)。
1	跟踪 Ice 操作调用的重试。
2	另外也跟踪 Ice 端点的使用。

Ice.Trace.Slicing

概要

`Ice.Trace.Slicing=num`

描述

切断的跟踪级别:

0	不跟踪切断活动 (缺省)。
1	跟踪接收者不知道的、因而将被切断的所有异常和类类型。

C.3 Ice 警告属性

Ice.Warn.Connections

概要

`Ice.Warn.Connections=num`

描述

如果 *num* 被设成大于零的值，Ice 应用将针对连接中的某些异常情况，打印出警告消息。缺省值是 0。

Ice.Warn.Datagrams

概要

`Ice.Warn.Datagrams=num`

描述

如果 *num* 被设成大于零的值，服务器将在所收到的数据报超过了服务器的接收缓冲区尺寸时打印警告消息 (注意，并非所有 UDP 实现都能检测到这种情况——有些实现会不声不响地扔掉所收到的太大的数据报)。缺省值是 0。

Ice.Warn.Dispatch

概要

`Ice.Warn.Dispatch=`*num*

描述

如果 *num* 被设成大于零的值， Ice 应用将针对分派请求时发生的某些异常，打印出警告消息。

0	不打印警告。
1	针对意料之外的 `Ice::LocalException`、`Ice::UserException`、 C++ 异常、 Java runtime 异常，打印警告 (缺省)。
2	和 1 一样，但还要针对 `Ice::ObjectNotExistException`、`Ice::FacetNotExistException`、 `Ice::OperationNotExistException` 发出警告。

Ice.Warn.AMICallback

概要

`Ice.Warn.AMICallback=`*num*

描述

如果 *num* 被设成大于零的值，在某个 AMI 回调引发了异常的情况下，打印警告。缺省值是 1。

Ice.Warn.Leaks

概要

`Ice.Warn.Leaks=`*num*

描述

如果 *num* 被设成大于零的值，在仍有与 Ice 有关的 C++ 对象在内存中的情况下， `Ice::Communicator` 析构器 将打印警告。缺省值是 1 （只适用于 C++)。

C.4 Ice 对象适配器属性

name.AdapterId

概要

name.AdapterId=*id*

描述

为名为 *name* 的对象适配器指定标识符。在使用同一个定位器实例的所有对象适配器的范围内，这个标识符必须是唯一的。如果用 name.Locator 或 Ice.Default.Locator 定义了定位器代理，这个对象适配器将在激活时用定位器注册表来设置其端点。

name.Endpoints

概要

name.Endpoints=*endpoints*

描述

把对象适配器 *name* 的端点设成 *endpoints*。

name.Locator

概要

name.Locator=*locator*

描述

为名为 *name* 的对象适配器指定一个定位器。其值是指向 Ice 定位器接口的串化代理。

name.RegisterProcess

概要

name.RegisterProcess=*num*

描述

如果 *num* 被设成大于零的值，名为 *name* 的对象适配器将向定位器注册表注册服务器。注册发生在对象适配器的初始激活阶段，在这个阶段对象适配器会创建一个实现 `Ice::Process` 接口的 servant，用一个 UUID 添加这个 servant，并用 `Ice.ServerId` 属性的值做服务器 id，向定位器注册表注册其代理。 servant 通过调用对象适配器的通信器的 `shutdown` 来实现 `shutdown` 操作。

向定位器注册表注册服务器很重要，于是像 IcePack 这样的服务就可以在必要时请求进行得体的关闭。如果服务器没有注册，那么就会使用平台特有的技术来请求关闭，而这些技术并非总是有效（特别是在 Windows 平台上）。

一个服务器应当只向定位器注册表注册一个对象适配器。

`Ice::Process` servant 可能会成为 “拒绝服务” 攻击的对象。对象适配器使用了 UUID 来使代理更难以被猜出，但如果服务器在不安全的环境中运行，应该为对象适配器配置安全的端点。也可以采用另一种做法：专门创建一个对象适配器，用于为 IcePack 这样的服务提供受限的访问点。

name.Router

概要

name`.Router=`*router*

描述

为名为 *name* 的对象适配器指定一个路由器。其值是指向 Glacier 路由器控制接口的串化代理。在定义了路由器之后，对象适配器可以就用原来的连接接收来自路由器的回调，这样路由器就不必再建立回到对象适配器的连接了。

一个路由器只能被指派给一个对象适配器。如果你为多个对象适配器指定同一个路由器，就会产生不确定的行为。缺省值是没有路由器。

Ice.PrintAdapterReady

概要

`Ice.PrintAdapterReady=`*num*

描述

如果 *num* 被设成大于零的值，对象适配器会在初始化完成之后，在标准输出上打印 "*adapter_name* ready"。如果有脚本需要等待对象适配器准备就绪，这个属性很有用。

C.5 Ice 插件属性

Ice.Plugin.*name*

概要

`Ice.Plugin.`*`name`*`=`*`entry_point [args]`*

描述

定义一个要在通信器初始化过程中安装的插件。

在 C++ 里， *entry_point* 的形式是 `basename[,version]:function`。`basename` 和可选的 `version` 用于构造 DLL 或共享库的名字。如果没有提供版本，就会使用 Ice 的版本。 `function` 是具有 C 链接方式的某个函数的名字。

例如，进入点 `MyPlugin,2.3:create` 意味着名为 `libMyPlugin.so.2.3` 的共享库名 （Unix）或 `MyPlugin23.dll` （Windows）。此外，如果你在 Windows 上构建的是 Ice 的调试版本，在版本的后面会自动加上一个 `d` （例如， `MyPlugin23d.dll`）。

在 Java 里， *entry_point* 是某个类的名字。

C.6 Ice 线程池属性

name.Size

概要

`name``.Size=`*`num`*

描述

name 是线程池的名字。客户端线程池的名字是 `Ice.ThreadPool.Client`，缺省的服务器端线程池的名字是 `Ice.ThreadPool.Server`。此外，每个对象适配器可以有自己的线程池。在这种情况下，线程池的名字是 *adapter_name*`.ThreadPool`。让对象适配器有自己的线程池，可以确保在分派某些 Ice 对象上的请求时，总有一定数量的线程可用，从而避免因为线程饥饿而发生死锁。

num 是线程池一开始的线程数，也是最少线程数。`Ice.ThreadPool.Client` 和 `Ice.ThreadPool.Server` 的缺省值是 1，对象适配器线程池的缺省值是零，也就是说，在缺省情况下，对象适配器使用的是缺省的服务器端线程池。

只有在进行嵌套的 AMI 调用时，客户端线程池才需要有多个线程。如果没有使用 AMI，或者 AMI 调用不是嵌套的（也就是说，AMI 回调方法没有调用 Ice 对象的其他任何方法），那么就不需要把客户线程池中的线程数设成大于一的值。

name.SizeMax

概要

name`.SizeMax=`*num*

描述

num 是线程池 *name* 的最大线程数。Ice 中的线程池可以基于平均负载因数、动态地增长和缩减。线程池最大只会增长到 `SizeMax` 所指定的数目，最小只会缩减到 `Size` 所指定的数目。

`SizeMax` 的缺省值是 `Size` 的值，意味着在缺省情况下，线程池不会动态增长。

name.SizeWarn

概要

name`.SizeWarn=`*num*

描述

当线程中只有 *num* 个活动线程时，会打印 "low on threads" 警告。`SizeWarn` 的缺省值是 `SizeMax` 指定的值的 80%。

C.7 Ice 的 Default 和 Override 属性

Ice.Default.Protocol

概要

`Ice.Default.Protocol=`*`protocol`*

描述

如果某个端点指定使用的协议是 `default`，就使用该属性指定的协议。缺省值是 `tcp`。

Ice.Default.Host

概要

`Ice.Default.Host=`*`host`*

描述

如果某个端点没有指定主机名（也就是，没有 `-h host` 选项），就会使用这个属性的 *host* 值来做主机名。缺省值是本地主机名。

Ice.Default.Router

概要

`Ice.Default.Router=`*`router`*

描述

指定所有代理的缺省路由器。其值是一个指向 Glacier 路由控制接口的串化代理。可以用 `ice_router()` 操作来替换某个代理的缺省路由器。缺省值是没有路由器。

Ice.Default.Locator

概要

`Ice.Default.Locator=`*`locator`*

描述

为所有代理和对象适配器指定一个缺省定位器。其值是一个指向 IcePack 定位器接口的串化代理。可以用 `ice_locator()` 操作来替换（override）某个代理的缺省定位器。缺省值是没有定位器。

IcePack 定位器的对象标识是 `IcePack/Locator`。它在 IcePack 的客户端点上侦听。例如，如果 `IcePack.Registry.Client.Endpoints` 被设成 `tcp -p 12000 -h localhost`，IcePack 定位器的串化代理将是 `IcePack/Locator:tcp -p 12000 -h localhost`。

Ice.Override.Timeout

概要

`Ice.Override.Timeout=`*num*

描述

如果设置了这个属性，它就会替换所有端点中的超时设置。 *num* 是以毫秒为单位的超时值， -1 是没有超时。

Ice.Override.ConnectTimeout

概要

`Ice.Override.ConnectTimeout=`*num*

描述

这个属性会替换建立连接时用的超时设置。*num* 是以毫秒为单位的超时值，-1 是没有超时。如果没有设置这个属性，就会使用 `Ice.Override.Timeout`。

Ice.Override.Compress

概要

`Ice.Override.Compress=`*num*

描述

如果设置了该属性，就会替换所有代理中的压缩设置。如果 *num* 被设成大于零的值，就启用压缩。如果设成零，就禁用压缩。

C.8 Ice 杂项属性

Ice.GC.Interval

概要

`Ice.GC.Interval=`*num*

描述

这个属性决定类垃圾收集器的运行频度。如果间隔被设成零（缺省），就不创建收集器线程。否则，收集器每 *num* 秒运行一次。

Ice.RetryIntervals

概要

`Ice.RetryIntervals=`*num [num ...]*

描述

这个属性定义操作的重试次数，以及每次重试之前的延时。例如，如果该属性被设成 *0 100 500*，操作将重试 3 次：在第一次失败时立刻重试，在第二次失败后等待 100 (ms)，在第三次失败后等待 500 (ms)。缺省值是立刻重试 (0)。如果设成 -1，就不会进行重试。

Ice.MessageSizeMax

概要

`Ice.MessageSizeMax=`*num*

描述

这个属性控制 Ice run time 将要接受或发送的协议消息的最大尺寸（以 kb 为单位）。该尺寸包括 Ice 协议头的尺寸。大于该尺寸的消息会引发

[MemoryLimitException]。缺省尺寸是 1024 (1 M 字节)。设成小于 1 的值会被忽略。

这个属性会调整 `Ice.UDP.RcvSize` 和 `Ice.UDP.SndSize` 的值，也就是说，如果 `Ice.UDP.RcvSize` 或 `Ice.UDP.SndSize` 大于 `Ice.MessageSizeMax * 1024 + 28`，它们会被调整成 `Ice.MessageSizeMax * 1024 + 28`。

Ice.ChangeUser

概要

`Ice.ChangeUser=`*`user`*

描述

如果设置了该属性， Ice 将把用户和组 id 改成 `/etc/passwd` 中的 *user* 的用户和组 id。要让该属性生效， Ice 应用必须由超级用户执行 (只适用于 Unix)。

Ice.ConnectionIdleTime

概要

`Ice.ConnectionIdleTime=`*`num`*

描述

如果 *num* 被设成大于零的值，就启用 ACM (Active Connection Management)。这意味着，连接会在空闲 *num* 秒后自动关闭。对用户代码来说这是透明的，也就是说，如果需要再次使用已关闭的连接，它们就会自动重新建立。缺省值是 60，意味着空闲连接会在一分钟后自动关闭。

Ice.MonitorConnections

概要

`Ice.MonitorConnections=`*`num`*

描述

如果 *num* 被设成大于零的值，Ice 将启动一个线程来监视连接。这个线程负责关闭空闲的连接 (参见 `Ice.ConnectionIdleTime`)，并强制实现 AMI 超时。缺省值是 `Ice.ConnectionIdleTime` 的值。

Ice.PrintProcessId

概要

`Ice.PrintProcessId=`*num*

描述

如果 *num* 被设成大于零的值，在启动时将在标准输出上打印进程 ID。

Ice.ProgramName

概要

`Ice.ProgramName=`*name*

描述

name 是程序名，会在初始化过程中根据 `argv[0]` 自动设置。但只要设置这个值，就可以使用另外的名字。

Ice.ServerId

概要

`Ice.ServerId=`*id*

描述

当对象适配器向定位器注册表注册服务器时，值 *id* 被用作服务器 id。更多信息，参见对象适配器属性 `adapter.RegisterProcess` 的说明。

Ice.ServerIdleTime

概要

`Ice.ServerIdleTime=`*num*

描述

如果 *num* 被设成大于零的值， Ice 将在通信器空闲 *num* 秒之后自动调用 `Communicator::shutdown`。这将会关闭通信器的服务器端，所有在 `Communicator::waitForShutdown` 中等待的线程都将返回。在此之后，服务器通常会做一些清理工作，然后退出。缺省值是零，意味着服务器不会自动关闭。

Ice.UseEventLog

概要

`Ice.UseEventLog=`*num*

描述

如果 *num* 被设成大于零的值，就会安装一个特殊的日志记录器，把日志消息送往 Windows Event Log，而不是标准错误。事件源的名字是 `Ice.ProgramName` 的值 (只适用于 Windows 2000/XP)。

Ice.UseSyslog

概要

`Ice.UseSyslog=`*num*

描述

如果 *num* 被设成大于零的值，就会安装一个特殊的日志记录器，把日志消息送往 syslog 设施，而不是标准错误。 syslog 的标识符是 `Ice.ProgramName` 的值 (只适用于 Unix)。

Ice.Logger.Timestamp

概要

`Ice.Logger.Timestamp=`*num*

描述

如果 *num* 被设成大于零的值，缺省日志记录器的输出将包含时间戳。

Ice.NullHandleAbort

概要

`Ice.NullHandleAbort=`*num*

描述

如果 *num* 被设成大于零的值，使用 null 智能指针（也就是句柄）调用操作将致使程序立刻中止，而不是引发 `IceUtil::NullHandleException` (只适用于 C++)。

Ice.Nohup

概要

`Ice.Nohup=`*num*

描述

如果 *num* 被设成大于零的值， C++ `Ice::Application` 类会忽略 `SIGHUP` (UNIX) 和 `CTRL_LOGOFF_EVENT` (Windows)。因此，如果启动应用的用户注销了，设置了 `Ice.Nohup` 的应用将会继续运行 (只适用于 C++)。

Ice.UDP.RcvSize, Ice.UDP.SndSize

概要

`Ice.UDP.RcvSize=`*num* `Ice.UDP.SndSize=`*num*

描述

这些属性把 UDP 的接收和发送缓冲区的尺寸设成指定的值（以字节为单位）。大于 `num - 28` 的 Ice 消息会引发 `DatagramLimitException`。缺省值取决于本地 UDP 栈的配置（常见的缺省值是 65535 和 8192 字节）。

OS 可能会对接收或发送缓冲区施加更低或更高的限制，也可能会调整缓冲区的尺寸。如果所请求的尺寸低于 OS 规定的最小值，这个尺寸就会被调整成 OS 规定的最小值。如果所请求的尺寸大于 OS 规定的最大值，这个值就会被调整成 OS 规定的最大值；此外， Ice 还会在日志中记录一条警告，记载所请求的尺寸和调整后的尺寸。

小于 28 的属性设置会被忽略。

注意，在许多操作系统上，可以把缓冲区尺寸设成大于 65535。这样的设置不会改变 UDP 包的有效负载不能超过 65507 的硬性限制，而只会影响内核所能缓冲的数据量。

小于 65535 的设置会限制 Ice 数据报的尺寸，并调整内核缓冲区的尺寸。

如果 `Ice.MessageSizeMax` 已被设置，而 `Ice.MessageSizeMax * 1024 + 28` 小于 UDP 接收或发送缓冲区的尺寸，对应的 UDP 缓冲区的尺寸就会减小到 `Ice.MessageSizeMax * 1024 + 28`。

C.9 IceSSL 属性

IceSSL.Trace.Security

概要

`IceSSL.Trace.Security=`*`num`*

描述

SSL 插件跟踪级别:

0	不进行安全性跟踪 (缺省)。
1	跟踪安全性警告。
2	和 1 一样，但更详细，包括在配置文件的解析过程中发出的警告。

IceSSL.Client.CertPath, IceSSL.Server.CertPath

概要

`IceSSL.Client.CertPath=`*`path`* `IceSSL.Server.CertPath=`*`path`*

描述

定义 PEM 格式的证书文件（RSA 和 DSA）和 Diffie-Hellman 组参数文件的路径（相对路径或绝对路径）。

如果 `IceSSL.Client.Config` 或 `IceSSL.Server.Config` 是相对路径，它相对的是 `IceSSL.Client.CertPath` 或 `IceSSL.Server.CertPath` 的值。

如果没有指定这两个属性，应用会用把当前工作目录当作证书目录。

IceSSL.Client.Config, IceSSL.Server.Config

概要

`IceSSL.Client.Config=`*`config_file`* `IceSSL.Server.Config=`*`config_file`*

描述

定义基于 XML 的配置文件，SSL 插件将从中夹在初始化信息和证书。如果该属性包含的是相对路径，它相对的是 `IceSSL.Client.CertPath` 或 `IceSSL.Server.CertPath` 定义的证书路径。

用于读取这个文件的 XML 解析器是 Xerces-c，它将在 XML 配置文件所在的目录中查找 DTD 文件。

取决于应用是运行在客户模式、服务器模式，还是这两种模式，要想让 IceSSL 插件正常运作，必须为这两个参数中的一个或两个指定有效的值。

IceSSL.Client.Handshake.Retries

概要

`IceSSL.Client.Handshake.Retries=`*`num`*

描述

IceSSL 客户试图在连接阶段完成整个的 SSL 握手。在进行这样的握手时，客户有可能会在等待服务器发回响应时超时。这个属性指定的是，在抛出 `Ice::ConnectionFailedException` 之前，客户要重试多少次握手。

如果这个属性没有指定，其缺省值是 10 次重试。

IceSSL.Client.Passphrase.Retries, IceSSL.Server.Passphrase.Retries

概要

```
IceSSL.Client.Passphrase.Retries=num
IceSSL.Server.Passphrase.Retries=num
```

描述

如果 IceSSL 被告知要使用某个已加密的 PEM 文件中的私钥，会显示一行提示：`Enter PEM pass phrase:`。如果输入的密码不正确，这两个属性将决定，在 IceSSL 关闭之前，用户能重试多少次。

如果没有指定这两个属性，其缺省值是 5 次重试。

IceSSL.Server.Overrides.RSA.PrivateKey, IceSSL.Server.Overrides.RSA.Certificate

概要

```
IceSSL.Server.Overrides.RSA.PrivateKey=Base64 encoded DER string
IceSSL.Server.Overrides.RSA.Certificate=Base64 encoded DER string
```

描述

这两个属性会替换在 Server 上下文的配置文件 (`IceSSL.Server.Config`) 中指定的 RSA 私钥和公钥（证书）。其值必须是私钥和公钥的 DER 表示，并用 base64 进行了编码。

这两个属性没有缺省值。

IceSSL.Server.Overrides.DSA.PrivateKey, IceSSL.Server.Overrides.DSA.Certificate

概要

```
IceSSL.Server.Overrides.DSA.PrivateKey=Base64 encoded DER string
IceSSL.Server.Overrides.DSA.Certificate=Base64 encoded DER string
```

描述

这两个属性会替换在 Server 上下文的配置文件 (`IceSSL.Server.Config`) 中指定的 DSA 私钥和公钥（证书）。其值必须是私钥和公钥的 DER 表示，并用 base64 进行了编码。

这两个属性没有缺省值。

IceSSL.Client.Overrides.RSA.PrivateKey, IceSSL.Client.Overrides.RSA.Certificate

概要

```
IceSSL.Client.Overrides.RSA.PrivateKey=Base64 encoded DER string
IceSSL.Client.Overrides.RSA.Certificate=Base64 encoded DER string
```

描述

通过这两个属性，可以用配置文件（由 `IceSSL.Client.Config` 指定）中指定的 RSA 私钥和公钥（证书）来替换 Client 上下文所用的私钥和公钥。其值必须是私钥和公钥的 DER 表示，并用 base64 进行了编码。

这两个属性没有缺省值。

IceSSL.Client.Overrides.DSA.PrivateKey, IceSSL.Client.Overrides.DSA.Certificate

概要

```
IceSSL.Client.Overrides.DSA.PrivateKey=Base64 encoded DER string
IceSSL.Client.Overrides.DSA.Certificate=Base64 encoded DER string
```

描述

这两个属性会替换 Client 上下文的配置文件 (`IceSSL.Client.Config`) 中指定的 RSA 私钥和公钥（证书）。其值必须是私钥和公钥的 DER 表示，并用 base64 进行了编码。

这两个属性没有缺省值。

IceSSL.Client.Overrides.CACertificate,

IceSSL.Server.Overrides.CACertificate

概要

```
IceSSL.Client.Overrides.CACertificate=Base64 encoded DER string
IceSSL.Server.Overrides.CACertificate=Base64 encoded DER string
```

描述

这两个属性会替换在 `IceSSL.Server.Config` 或 `IceSSL.Client.Config` 中指定的任何受信的 Certificate Authority (CA) 证书。新的证书被表示成证书的 DER 二进制形式的 base64 编码形式。

这两个属性没有缺省值。

IceSSL.Client.IgnoreValidPeriod, IceSSL.Server.IgnoreValidPeriod

概要

```
IceSSL.Client.IgnoreValidPeriod=num
IceSSL.Server.IgnoreValidPeriod=num
```

描述

如果这两个属性被设成 1，缺省证书校验器会忽略对对端的证书进行有效性校验的阶段。这两个属性的缺省值是 0，意味着不忽略该阶段。

C.10 IceBox 属性

IceBox.ServiceManager.Endpoints

概要

```
IceBox.ServiceManager.Endpoints=endpoints
```

描述

定义 IceBox 服务管理器接口的端点。服务管理器端点必须能被 IceBox 管理工具访问到，以关闭 IceBox 服务器。

IceBox.ServiceManager.Identity

概要

`IceBox.ServiceManager.Identity=`*`identity`*

描述

服务管理器接口的标识。如果没有指定，缺省值是 `ServiceManager`。

IceBox.PrintServicesReady

概要

`IceBox.PrintServicesReady=`*`token`*

描述

在所有服务的初始化完成之后，服务管理器将打印 "*token* ready"。如果有脚本想要等待所有服务准备就绪，这项特性很有用。

IceBox.LoadOrder

概要

`IceBox.LoadOrder=`*`names`*

描述

确定服务加载的次序。服务管理器会按照服务在 *names* 中出现的次序加载它们，在 *names* 中，服务名用逗号或空格分隔。没有在 *names* 中提到的服务都在后面加载，次序不确定。

IceBox.Service.*name*

概要

`IceBox.Service.`*`name=entry_point [args]`*

描述

定义要在 IceBox 初始化过程中加载的服务。

在 C++ 里， *entry_point* 的形式是 `library:symbol`。 `library` 是一个共享库或 DLL 的名字。 `symbol` 是用于创建服务的工厂函数的名字。

在 Java 里， *entry_point* 是服务实现类的名字。

IceBox.DBEnvName.*name*

概要

`IceBox.DBEnvName.`*`name`*`=`*`db`*

描述

为名为 *name* 的 Freeze 服务定义数据库环境目录。

IceBox.UseSharedCommunicator.*name*

概要

`IceBox.UseSharedCommunicator.`*`name`*`=`*`num`*

描述

如果 *num* 被设成大于零的数，服务管理器将给名为 *name* 的服务提供一个通信器，其他服务也可能会使用这个通信器。

C.11 IcePack 属性

IcePack.Registry.Client.Endpoints

概要

`IcePack.Registry.Client.Endpoints=`*`endpoints`*

描述

定义 IcePack 客户接口的端点。这些客户端点必须能被 “用 IcePack 来定位对象的 Ice 客户” 访问到 （参见 `Ice.Default.Locator`）。

IcePack.Registry.Server.Endpoints

概要

`IcePack.Registry.Server.Endpoints=`*`endpoints`*

描述

定义 IcePack 服务器接口的端点。这些服务器端点必须能被 “用 IcePack 来注册其对象适配器端点的 Ice 服务器” 访问到。

IcePack.Registry.Admin.Endpoints

概要

`IcePack.Registry.Admin.Endpoints=`*`endpoints`*

描述

定义可选的 IcePack admin 接口的管理端点。这些管理端点必须能被 “使用了 IcePack 管理接口的客户” 访问到，比如 IcePack 管理工具。
允许访问 IcePack admin 接口在安全上有风险！如果这个属性没有指定，admin 接口将被禁用。

IcePack.Registry.Internal.Endpoints

概要

`IcePack.Registry.Internal.Endpoints=`*`endpoints`*

描述

定义 IcePack 内部接口的端点。这些内部端点必须能被 IcePack 节点访问。节点会用这个接口来和注册表通信。

IcePack.Registry.Data

概要

`IcePack.Registry.Data=`*`path`*

描述

定义 IcePack 注册表数据目录的路径。

IcePack.Registry.DynamicRegistration

概要

`IcePack.Registry.DynamicRegistration=`*`num`*

描述

如果 *num* 被设成大于零的值，定位器注册表将允许 Ice 服务器为先前没有注册过的对象适配器设置端点。

IcePack.Registry.Trace.ServerRegistry

概要

`IcePack.Registry.Trace.ServerRegistry=`*`num`*

描述

服务器注册表跟踪级别:

0	不跟踪服务器注册表 (缺省)。
1	跟踪服务器的注册、移除。

IcePack.Registry.Trace.AdapterRegistry

概要

`IcePack.Registry.Trace.AdapterRegistry=`*`num`*

描述

对象适配器注册表跟踪级别:

0	不跟踪对象适配器注册表 (缺省)。
1	跟踪对象适配器的注册、移除。

IcePack.Registry.Trace.NodeRegistry

概要

`IcePack.Registry.Trace.NodeRegistry=`*`num`*

描述

节点注册表跟踪级别:

0	不跟踪节点注册表 (缺省)。
1	跟踪节点的注册、移除。

IcePack.Registry.Trace.ObjectRegistry

概要

`IcePack.Registry.Trace.ObjectRegistry=`*`num`*

描述

对象注册表跟踪级别:

0	不跟踪对象注册表 (缺省)。
1	跟踪对象的注册、移除。

IcePack.Node.Endpoints

概要

`IcePack.Node.Endpoints=`*`endpoints`*

描述

定义 IcePack 节点接口的端点。这些节点端点必须能被 IcePack 注册表访问。注册表使用这个接口来和节点通信。

IcePack.Node.Name

概要

`IcePack.Node.Name=`*`name`*

描述

定义 IcePack 节点的名字。使用同一个注册的节点的名字必须是唯一的。如果已经在同名的节点在运行，节点将拒绝启动。

缺省值和 `gethostname()` 返回的主机名一样。

IcePack.Node.Data

概要

`IcePack.Node.Data=`*`path`*

描述

定义 IcePack 节点数据目录的路径。如果在该目录中没有 db 和 `servers` 子目录，节点将在该目录中创建它们。db 目录含有节点数据库。 `servers` 目录含有每个已部署服务器的配置数据。

IcePack.Node.Output

概要

`IcePack.Node.Output=`*`path`*

描述

定义 IcePack 节点的输出目录的路径。如果设置了该属性，节点将把被启动的服务器的 stdout 和 stderr 重定向到该目录中名为 *server*.out 和 *server*.err 的文件。否则，被启动的服务器就会共享 IcePack 节点的 stdout 和 stderr。

IcePack.Node.PropertiesOverride

概要

`IcePack.Node.PropertiesOverride=`*`overrides`*

描述

定义一个属性列表，用于替换在服务器部署描述符中定义的属性。例如，在有些情况下，为服务器设置 `Ice.Default.Host` 属性，比在服务器部署描述符中设置更好。各属性定义应该用空格分隔。

IcePack.Node.RedirectErrToOut

概要

`IcePack.Node.RedirectErrToOut=`*num*

描述

如果 *num* 被设成大于零的值，每个被启动的服务器的 stderr 就会被重定向到服务器的 stdout。

IcePack.Node.WaitTime

概要

`IcePack.Node.WaitTime=`*num*

描述

以秒为单位，定义 IcePack 等待服务器激活和解除激活的时间长度。

如果服务器是自动激活的，而且没有在这段时间里注册其对象适配器端点，节点就会认为服务器出了问题，返回一个空的端点集给客户。

如果服务器正在得体地解除激活，而 IcePack 在这段时间里没有检测到服务器已经解除激活， IcePacck 就会杀死该服务器。

缺省值是 60 秒。

IcePack.Node.CollocateRegistry

概要

`IcePack.Node.CollocateRegistry=`*num*

描述

如果 *num* 被设成大于零的值，该节点将会和 IcePack 注册表并置在一起。并置的注册表所配置的属性和独立的 IcePack 注册表是一样的。

IcePack.Node.PrintServersReady

概要

`IcePack.Node.PrintServersReady=`*`token`*

描述

在节点所管理的所有服务器都就绪之后， IcePack 节点将在标准输出上打印 "*token* ready"。如果有脚本希望等待所有服务器就绪，这个属性很有用。

IcePack.Node.Trace.Server

概要

`IcePack.Node.Trace.Server=`*`num`*

描述

服务器跟踪级别:

0	不跟踪服务器 (缺省)。
1	跟踪服务器的添加、移除。
2	和 1 一样，但更详细，包括服务器的激活和解除激活，以及更多的诊断消息。
3	和 2 一样，但更详细，包括服务器的过渡性的状态变化 (正在激活和解除激活)。

IcePack.Node.Trace.Adapter

概要

`IcePack.Node.Trace.Adapter=`*`num`*

描述

对象适配器跟踪级别:

0	不跟踪对象适配器 (缺省)。

1	跟踪对象适配器的添加、移除。
2	和 1 一样，但更详细，包括对象适配器的激活和解除激活，以及更多的诊断消息。
3	和 2 一样，但更详细，包括对象适配器的过渡性的状态变化 (例如，“正等待激活”)。

IcePack.Node.Trace.Activator

概要

`IcePack.Node.Trace.Activator=`*`num`*

描述

激活器跟踪级别:

0	不跟踪激活器 (缺省)。
1	跟踪进程的激活、终止。
2	和 1 一样，但更详细，包括进程的信号处理，以及更多的关于进程激活的诊断消息。
3	和 2 一样，但更详细，包括更多的关于进程激活的诊断消息（例如，被激活的进程的路径、工作目录，以及参数）。

C.12 IceStorm 属性

IceStorm.TopicManager.Endpoints

概要

`IceStorm.TopicManager.Endpoints=`*`endpoints`*

描述

定义 IceStorm 主题管理器和主题对象的端点。

IceStorm.Publish.Endpoints

概要

`IceStorm.Publish.Endpoints=`*`endpoints`*

描述

定义 IceStorm 发布者对象的端点。

IceStorm.Trace.TopicManager

概要

`IceStorm.Trace.TopicManager=`*`num`*

描述

主题管理器跟踪级别:

0	不跟踪主题管理器 (缺省)。
1	跟踪主题的创建。

IceStorm.Trace.Topic

概要

`IceStorm.Trace.Topic=`*`num`*

描述

主题跟踪级别:

0	不跟踪主题 (缺省)。
1	跟踪主题的链接、订阅、解除订阅。
2	和 1 一样，但更详细，包括 QoS 信息和其他的诊断信息。

IceStorm.Trace.Flush

概要

`IceStorm.Trace.Flush=num`

描述

某个线程负责把批可靠性事件 (batch reliability events) 刷出给订阅者，这个属性指定是否要跟踪该线程上的消息:

0	不跟踪刷出 (缺省)。
1	跟踪每次刷出。

IceStorm.Trace.Subscriber

概要

`IceStorm.Trace.Subscriber=num`

描述

订阅者跟踪级别:

0	不跟踪订阅者 (缺省)。
1	跟踪关于 “订阅和取消订阅” 的主题诊断信息。

IceStorm.Flush.Timeout

概要

`IceStorm.Flush.Timeout=num`

描述

以毫秒为单位，定义发送批可靠性事件的时间间隔。缺省是 1000 ms。这个属性的最小值是 100 ms。

IceStorm.TopicManager.Proxy

概要

`IceStorm.TopicManager.Proxy=`*`proxy`*

描述

定义 IceStorm 主题管理器的代理。 IceStorm 管理工具将使用这个属性，其他应用也可以使用。

C.13 Glacier 路由器属性

Glacier.Router.Endpoints, Glacier.Router.Client.Endpoints, Glacier.Router.Server.Endpoints

概要

```
Glacier.Router.Endpoints=endpoints
Glacier.Router.Client.Endpoints=endpoints
Glacier.Router.Server.Endpoints=endpoints
```

描述

定义 Glacier 路由器控制接口、客户接口、服务器接口的端点。路由器端点和客户端点必须能被 “路由器为之转发请求的 Glacier 客户、以及对其进行回调的 Glacier 客户” 访问到。服务器端点必须能被 “路由器转发的请求所到达的 Glacier 服务器、以及向路由器发送对客户的回调的 Glacier 客户” 访问到。

Glacier.Router.Identity

概要

`Glacier.Router.Identity=`*`identity`*

描述

路由器控制接口的标识。如果没有指定，缺省值是 `router`。

Glacier.Router.PrintProxyOnFd

概要

`Glacier.Router.PrintProxyOnFd=`*`fd`*

描述

如果设置了该属性，就在文件描述符 *fd* 上打印串化的路由器代理，然后关闭这个文件描述符 (只适用于 Unix)。
这个操作归 Glacier 路由器的启动器专用，不应该手工设置它。

Glacier.Router.Trace.Client

概要

`Glacier.Router.Trace.Client=`*`num`*

描述

客户接口跟踪级别:

0	不跟踪客户接口 (缺省)。
1	在把请求从客户转发到服务器的过程中跟踪异常。
2	还要跟踪 “把请求从客户转发到服务器的过程中” 的详细路由信息。

Glacier.Router.Trace.Server

概要

`Glacier.Router.Trace.Server=`*`num`*

描述

服务器接口跟踪级别:

0	不跟踪服务器接口 (缺省)。
1	在服务器回调客户的过程中跟踪异常。

2	还要跟踪服务器回调客户过程中的详细的反向路由信息。

Glacier.Router.Trace.RoutingTable

概要

`Glacier.Router.Trace.RoutingTable=`*`num`*

描述

路由表跟踪级别:

0	不跟踪路由表 (缺省)。
1	跟踪对路由表所做的添加。

Glacier.Router.Trace.Throttle

概要

`Glacier.Router.Trace.Throttle=`*`num`*

描述

请求阀门 (request throttle) 的跟踪级别:

0	不跟踪请求阀门 (缺省)。
1	跟踪被阀门堵住的请求。

Glacier.Router.Client.ForwardContext, Glacier.Router.Server.ForwardContext

概要

`Glacier.Router.Client.ForwardContext=`*`num`*
`Glacier.Router.Server.ForwardContext=`*`num`*

描述

如果 *num* 被设成大于零的值，上下文参数就会按照从客户或服务器接收到的内容，不做修改就转发出去。否则，转发的就是一个空上下文。缺省是不转发上下文。

Glacier.Router.Client.SleepTime, Glacier.Router.Server.SleepTime

概要

```
Glacier.Router.Client.SleepTime=num
Glacier.Router.Server.SleepTime=num
```

描述

num 是批请求或消息被转发之后的休眠时间（延迟），单位是毫秒。缺省值是零，意味着没有延迟。设置这两个值可以避免消息“泛滥”(flooding)。要想避免“拒绝服务攻击”，或规定收集批消息的最小时间间隔，这两个属性很有用。

Glacier.Router.Client.Throttle.Twoways, Glacier.Router.Server.Throttle.Twoways

概要

```
Glacier.Router.Client.Throttle.Twoways=num
Glacier.Router.Server.Throttle.Twoways=num
```

描述

num 路由器同时转发的双向请求的最大数目。缺省值是零，意味着没有限制。启用双向消息阀门可以防止路由器消耗后端服务的太多资源。

Glacier.Router.SessionManager

概要

```
Glacier.Router.SessionManager=proxy
```

描述

指向会话管理器的串化代理。如果没有指定，就不能使用 `Router::createSession()` 方法。

Glacier.Router.UserId

概要

`Glacier.Router.UserId=`*`name`*

描述

被验证的用户的 id。这通常由 Glacier 路由器启动器传来。用户 id 将被用作传给 `Router::createSession()` 的参数。

Glacier.Router.AllowCategories

概要

`Glacier.Router.AllowCategories=`*`list`*

描述

用空格分隔的范畴列表。如果设置了这个属性，那么只有哪些标识与列表中的某个范畴相吻合的 Ice 对象才能接收请求。

Glacier.Router.AcceptCert

概要

`Glacier.Router.AcceptCert=`*`base64 encoded certificate string`*

描述

一个用 base64 编码的证书（可以针对已有的 IceSSL::RSAKeyPair 调用 certToBase64()，从而获得这种形式的证书）。在 SSL 模式中， Glacier Router 将使用这个证书来确定哪些客户可以连接它。只有使用了这个证书的客户能进行连接，其他的将会被拒绝。

C.14 Glacier 路由器启动器属性

Glacier.Starter.Endpoints

概要

`Glacier.Starter.Endpoints=`*`endpoints`*

描述

定义 Glacier 路由器启动器的端点 (只适用于 Unix)。

Glacier.Starter.PasswordVerifier

概要

`Glacier.Starter.PasswordVerifier=`*`proxy`*

描述

如果设置了该属性，将会使用指定的密码校验器。如果没有设置，将会使用内建的基于 `crypt` 的密码校验器。

Glacier.Starter.CryptPasswords

概要

`Glacier.Starter.CryptPasswords=`*`file`*

描述

含有用户 id/ 密码对的文件的路径名，密码用 `crypt` 算法加过密。缺省路径名是 "`passwords`"。这个文件仅用于内建的基于 `crypt` 的密码校验器，也就是说，如果 `Glacier.Starter.PasswordVerifier` 被设置，这个属性就会被忽略。

Glacier.Starter.RouterPath

概要

`Glacier.Starter.RouterPath=`*`path`*

描述

设置将要启动的 Glacier 路由器的可执行程序的路径。缺省值是 `glacier`(只适用于 Unix)。

Glacier.Starter.PropertiesOverride

概要

`Glacier.Starter.PropertiesOverride=`*`overrides`*

描述

在缺省情况下， Glacier 路由器启动器在启动新路由器时所用的属性集和启动器自身所用的完全一样。 *overrides* 可以包含一个用于路由器的属性列表，可以和启动器的属性一起使用，也可以替换启动器的属性。属性定义应该用空格分隔。

例如，在许多情况下，为路由器设置 `Ice.ServerIdleTime` 属性，但不为启动器设置该属性，是一种更好的做法。要把空闲时间设成 60 秒，可以这样进行设置:
`Glacier.Starter.PropertiesOverride=Ice.ServerIdleTime=60`。（只适用于 Unix)。

Glacier.Starter.StartupTimeout

概要

`Glacier.Starter.StartupTimeout=`*`num`*

描述

num 是 Glacier 路由器启动器等待路由器启动的秒数。如果发生超时，就会返回 `Glacier::CannotStartRouterException` 给调用者。缺省值是 10 秒。小于一秒的超时值会被自动改成一秒（只适用于 Unix）。

Glacier.Starter.AddUserToAllowCategories

概要

`Glacier.Starter.AddUserToAllowCategories=`*`num`*

描述

控制 Glacier 路由器启动器在路由器启动时的行为，即是否及怎样把认证过的用户 id 添加到 `Glacier.Router.AllowCategories` 属性:

0	不添加用户 id(缺省)。
1	添加用户 id。
2	在用户 id 前面加上下划线，然后再添加。

Glacier.Starter.Trace

概要

`Glacier.Starter.Trace=`*`num`*

描述

路由器启动器的跟踪级别 (只适用于 Unix):

0	不跟踪路由器启动器 (缺省)。
1	跟踪路由器的启动异常。
2	还要跟踪每一次成功的路由器启动。

Glacier.Starter.Certificate.Country

概要

`Glacier.Starter.Certificate.Country=`*`country code`*

描述 Description

这个属性指定的是 Distinguished Name (DN) 值的国家代码部分，这个值将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。关于这个代码的有效值，下面有一些例子：美国的代码是 "US"，加拿大的代码是 "CA"。缺省值是 "US"。

Glacier.Starter.Certificate.StateProvince

概要

`Glacier.Starter.Certificate.StateProvince=`*`state/province code`*

描述

这个属性指定的是 Distinguished Name (DN) 值的州或省代码部分，这个值将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。关于这个代码的有效值，下面有一些例子：加州的代码是 "CA"，加拿大的不列颠哥伦比亚省是 "British Columbia"。缺省值是 "DC"。

Glacier.Starter.Certificate.Locality

概要

`Glacier.Starter.Certificate.Locality=`*`city or town name`*

描述

这个属性指定的是 Distinguished Name (DN) 值的本地名称部分，这个值将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。本地名称通常是城市或城镇的名称。缺省值是 "Washington"。

Glacier.Starter.Certificate.Organization

概要

`Glacier.Starter.Certificate.Organization=`*`organization or company name`*

Description

这个属性指定的是 Distinguished Name (DN) 值的机构名称部分，这个值将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。机构名称通常是证书所针对的公司或机构的名称。缺省值是 "Some Company Inc."。

Glacier.Starter.Certificate.OrganizationalUnit

概要

`Glacier.Starter.Certificate.OrganizationalUnit=`*`department`*

描述

这个属性指定的是 Distinguished Name (DN) 值的机构单位部分，这个值将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。机构单位通常是证书所针对的公司或机构中的部门的名称。缺省值是 "Sales"。

Glacier.Starter.Certificate.CommonName

概要

`Glacier.Starter.Certificate.CommonName=`*`contact name`*

描述

这个属性指定的是 Distinguished Name (DN) 值的公共名称 (common name) 部分，将出现在 Glacier Router Starter 为客户应用和 Glacier Router 自身生成的证书中。公共名称通常是证书所针对的公司和部门中联系人。缺省值是 "John Doe"。

Glacier.Starter.Certificate.BitStrength

概要

`Glacier.Starter.Certificate.BitStrength=`*`number of bits`*

描述

这个属性指定的是 Glacier Router Starter 为客户应用和 Glacier Router 自身生成证书时使用的位强度。这个值是 RSA 密钥的模数尺寸 (modulus size)。

尽管模数尺寸是根据特定应用的需要决定的，但要小心不要指定太大的尺寸，因为生成证书是一种昂贵的操作。不能指定小于 512 位的尺寸，而在选用超过 2048 位的尺寸时要考虑生成所需的时间。缺省值是 1024 位。

Glacier.Starter.Certificate.SecondsValid

概要

`Glacier.Starter.Certificate.SecondsValid=`*`seconds`*

描述

这个属性指定的是 Glacier Router Starter 所生成的证书的有效时间，以秒为单位。缺省值是 1 天 (86,400 秒)。

Glacier.Starter.Certificate.IssuedAdjust

概要

`Glacier.Starter.Certificate.IssuedAdjust=`*`(+/-) seconds`*

描述

这个属性将调整为 Glacier Router 动态生成的证书上的发放时间戳。缺省值是 0，即基于证书实际创建的系统时间来设置发放时间。按照所设的秒数，正值将把时间戳调整到未来的时间，而负值将把时间戳调整到过去的时间。调整将相对于服务器时间做出。

C.15 Freeze 属性

Freeze.DbEnv.*env-name*.DbCheckpointPeriod

概要

`Freeze.DbEnv.`*`env-name`*`.DbCheckpointPeriod=`*`num`*

描述

Freeze 创建的每一个 Berkeley DB 环境都有一个与其相关联的线程，每隔 *num* 秒检查一次该环境。缺省值是 120 秒。

Freeze.DbEnv.*env-name*.DbHome

概要

`Freeze.DbEnv.`*`env-name`*`.DbHome=`*`db-home`*

描述

定义这个 Freeze 数据库环境的主目录。缺省是 *env-name*。

Freeze.DbEnv.*env-name*.DbPrivate

概要

`Freeze.DbEnv.`*`env-name`*`.DbPrivate=`*`num`*

描述

如果 *num* 被设成大于零的值， Freeze 会让 Berkeley DB 适用进程私有的内存，而不是共享内存。缺省值是 1。要针对正在使用的环境运行 db_archive（或其他 Berkeley DB 实用程序），把这个属性设成零。

Freeze.DbEnv.*env-name*.DbRecoverFatal

概要

`Freeze.DbEnv.`*`env-name`*`.DbRecoverFatal=`*`num`*

描述

如果 *num* 被设成大于零的值，当环境被打开时，将进行 “fatal” 恢复。缺省值是 0。

Freeze.DbEnv.*env-name*.OldLogsAutoDelete

概要

`Freeze.DbEnv.`*`env-name`*`.OldLogsAutoDelete=`*`num`*

描述

如果 *num* 被设成大于零的值，在每次遇到周期性的检查点时 (参见 `Freeze.DbEnv.env-name.DbCheckpointPeriod`)，不再使用的老事务日志将会被删除。缺省值是 1。

Freeze.DbEnv.*env-name*.PeriodicCheckpointMinSize

概要

`Freeze.DbEnv.`*`env-name`*`.PeriodicCheckpointMinSize=`*`num`*

描述

num 是周期性的检查点的最小尺寸 (参见 `Freeze.DbEnv.env-name.DbCheckpointPeriod`)，以 kb 为单位。这个值将传给 Berkeley DB 的检查点函数。缺省值是 0(也就是说，没有最小尺寸)。

Freeze.Evictor.*env-name*.*db-name*.MaxTxSize

概要

`Freeze.Evictor.`*`env-name`*`.`*`db-name`*`.MaxTxSize=`*`num`*

描述

Freeze 使用了一个后台线程来保存对数据库的更新。在把许多 facet 合起来保存时使用了事务。 *num* 定义的是在每个事务中所保存的 facet 的最大数目。缺省值是 10 * SaveSizeTrigger (参见 `Freeze.Evictor.env-name.db-name.SaveSizeTrigger`)；如果这个值为负，实际的值将被设成 100。

Freeze.Evictor.*env-name*.*db-name*.SavePeriod

概要

`Freeze.Evictor.`*`env-name`*`.`*`db-name`*`.SavePeriod=`*`num`*

描述

Freeze 使用了一个后台线程来保存对数据库的更新。在上一次保存的 *num* 毫秒之后，如果有任何 facet 被创建、修改或销毁，这个后台线程就会醒来保存这些 facet。如果 *num*，就不进行周期性地保存。缺省值是 60,000。

Freeze.Evictor.*env-name*.*db-name*.SaveSizeTrigger

概要

`Freeze.Evictor.`*`env-name`*`.`*`db-name`*`.SaveSizeTrigger=`*`num`*

描述

Freeze 使用了一个后台线程来保存对数据库的更新。如果 *num* 是 0 或正数，只要有 *num* 个或更多的 facet 被创建、修改或销毁，后台线程就会醒来保存它们。如果 *num* 为负，后台线程就不会因上述变化而被唤醒。缺省值是 10。

Freeze.Trace.DbEnv

概要

`Freeze.Trace.DbEnv=`*`num`*

描述

Freeze 数据库环境活动的跟踪级别:

0	不跟踪数据库环境的活动 (缺省)。
1	跟踪数据库的打开和关闭。
2	还要跟踪检查点，以及老日志文件的移除。

Freeze.Trace.Evictor

概要

`Freeze.Trace.Evictor=`*`num`*

描述

Freeze 逐出器活动的跟踪级别:

0	不跟踪逐出器的活动 (缺省)。
1	跟踪 Ice 对象和 facet 的创建和析构、 facet 的流动时间、 facet 的保存时间、对象逐出 （每 50 个对象）和逐出器的解除激活。

2	还要跟踪对象查找，以及所有对象的逐出。
3	还要跟踪从数据库取回对象的活动。

Freeze.Trace.Map

概要

`Freeze.Trace.Map=`*`num`*

描述

Freeze 映射表活动的跟踪级别:

0	不跟踪映射表的活动 (缺省)。
1	跟踪数据库的打开和关闭。
2	还要跟踪迭代器和事务操作，以及底层数据库的引用计数。

附录 D
代理与端点

D.1 代理

纲要

identity* -f *facet* -t -o -O -d -D -s @ *adapter_id* : *endpoints

描述

一个串化代理由一个标识符、代理选项、以及一个可选的对象适配器标识符或端点列表组成。如果代理含有没有端点的标识、或有对象适配器标识 符的标识，它表示的就是将使用 Ice 定位器解析的间接代理 (参见 `Ice.Default.Locator` 属性)。

代理选项用于配置调用模式:

-f *facet*	选择 Ice 对象的某个 facet。
-t	配置代理，使用双向调用 (缺省)
-o	配置代理，使用单向调用。

-O	配置代理，使用成批的单向调用。
-d	配置代理，使用数据报调用。
-D	配置代理，使用成批的数据报调用。
-s	配置代理，使用安全的调用。

代理选项 `-t`、`-o`、`-O`、`-d`、还有 `-D` 是互斥的。

对象标识 *identity* 的结构是 `[category/]name`， *category* 部分及斜杠分隔符是可选的。如果 *identity* 含有空格、`:` 或 `@`，它必须放在单引号或双引号中。 *category* 和 *name* 部分是 UTF8 串，所用编码方式将在下面描述。在 *category* 或 *name* 中，只要出现斜杆 (/)，就要用反斜杠来转义 (也就是 \/)。

`-f` 选项的 *facet* 参数代表的是 facet 路径，由一个或多个用斜杠 (/) 分隔的 facet 组成。如果 *facet* 含有空格，它必须放在单引号或双引号中。 facet 路径的每个部分都是一个 UTF8 串，所用编码方式将在下面描述。在 facet 路径的某个部分中，只要出现斜杠 (/)，就要用反斜杠来转义 (也就是 \/)。

对象适配器标识符 *adapter_id* 是一个 UTF8 串，所用编码方式将在下面描述。如果 *adapter_id* 含有空格，它必须放在单引号或双引号中。

UTF8 串的编码方式是这样的：对于在 32-126 （包括这两个数）范围内的字符，使用 ASCII 字符。在这个范围以外的字符必须用转义序列进行编码 (`\b`、`\f`、`\n`、`\r`、`\t`) 或八进制表示法 (例如， `\007`)。你可以用反斜杠来使引号转义，也可以用反斜杠来使其自身转义 (\\)。

如果指定了端点，它们必须用冒号 (`:`) 分隔，并按照 “端点” 一节中的描述格式化。串化代理中各个端点的次序并不一定就是绑定过程中的端点使用次序：当串化代理被转换成代理实例时，作为负载均衡的一种形式，端点列表会被打乱。

如果指定了 `-s` 选项，在绑定过程中，将只考虑那些支持安全调用的端点。如果没有找到有效的端点，应用就会收到 `Ice::NoEndpointException`。

如果没有指定 `-s` 选项，端点列表就会进行排序，以在绑定过程中，优先于安全端点使用非安全端点。换句话说，首先会尝试在所有非安全端点上建立连接，然后才尝试安全端点。

如果指定了未知选项，或者串化代理的格式有问题，应用会收到 `Ice::ProxyParseException`。如果端点的格式有问题，应用会收到 `Ice::EndpointParseException`。

D.2 端点

纲要

endpoint* : *endpoint

描述

端点列表由一个或多个用冒号 (:) 分隔的端点组成。端点的格式如下所示: ***protocol option***。所支持的协议有 tcp、 udp、 ssl，以及 default。如果使用了 default，它会被 Ice.Default.Protocol 属性的值替代。如果端点的格式有问题，或者指定了未知的协议，应用会收到 Ice::EndpointParseException。

只有安装了 IceSSL 插件，才能使用 ssl 协议。

各个协议及其所支持的选项将在下面描述。

TCP 端点

纲要

tcp -h *host* -p *port* -t *timeout* -z

描述

tcp 端点支持以下选项:

选项	描述	客户语义	服务器语义
-h *host*	指定端点的主机名或 IP 地址。如果没有指定，将使用 Ice.Default.Host 的值。	确定要连接到的主机名或 IP 地址。	确定对象适配器用于侦听连接的网络接口，以及在适配器所创建的代理中向外公布的主机名。
-p *port*	指定端点的端口号。	确定要连接到的端口 (必须指定)。	如果没有指定这个选项，或是 *port* 为零，端口将由操作系统选择。

选项	描述	客户语义	服务器语义
-t *timeout*	以毫秒为单位指定端点超时。	如果 *timeout* 大于零，它会把绑定和代理调用的延迟设为最大。如果发生超时，应用会收到 `Ice::TimeoutException`。	确定在对象适配器所创建的代理中向往公布的缺省超时。
-z	指定使用 bzip2 进行压缩。	确定是否要发送压缩请求。	确定是否要发送压缩响应，以及是否要在适配器所创建的代理中向外公布使用了压缩。

UDP 端点

纲要

`udp -v major.minor -e major.minor -h host -p port -c -z`

描述

udp 端点支持以下选项：

选项	描述	客户语义	服务器语义
-v *major.minor*	指定这个端点要使用的协议大版本号和最高小版本号。如果没有指定，就会使用客户端 Ice run time 的大协议版本及所支持的最高的小版本。	确定客户端在把消息发送到这个端点时所使用的大协议版本和最高小版本。大协议版本号必须和服务器的大协议版本号吻合；小协议版本号不能高于服务器所支持的最高小版本号。	确定服务器端为这个端点公布的大版本和最高小版本。大协议版本号必须和服务器的大协议版本号吻合；小协议版本号不能高于服务器所支持的最高小版本号。

选项	描述	客户语义	服务器语义
-e *major.minor*	指定这个端点要使用的编码大版本号和最高小版本号。如果没有指定，就会使用客户端 Ice run time 的大协议版本及所支持的最高的小版本。	确定客户端在把消息发送到这个端点时所使用的大编码版本和最高小版本。大编码版本号必须和服务器的大编码版本号吻合；小编码版本号不能高于服务器所支持的最高小版本号。	确定服务器端为这个端点公布的大编码版本和最高小版本。大编码版本号必须和服务器的大编码版本号吻合；小编码版本号不能高于服务器所支持的最高小版本号。
-h *host*	指定端点的主机名或 IP 地址。如果没有指定，将使用 `Ice.Default.Host` 的值。	确定要把数据报发往哪个主机或 IP 地址。	确定对象适配器用于侦听数据报的网络接口，以及在适配器所创建的代理中向外公布的主机名。
-p *port*	指定端点的端口号。	确定要把数据报发往哪个端口 (必须指定)	如果没有指定这个选项，或是 *port* 为零，端口将由操作系统选择。
-c	指定应该使用某个已连接的 UDP socket。	无。	让服务器连接到第一个把数据报发到这个端点的对端的 socket。
-z	指定使用 bzip2 进行压缩。	确定是否要发送压缩请求。	确定是否要在适配器所创建的代理中向外公布使用了压缩。

SSL 端点

纲要

`ssl -h host -p port -t timeout -z`

描述

`ssl` 端点支持以下选项：

选项	描述	客户语义	服务器语义
-h *host*	指定端点的主机名或 IP 地址。如果没有指定，将使用 `Ice.Default.Host` 的值。	确定要连接到的主机名或 IP 地址。	确定对象适配器用于侦听连接的网络接口，以及在适配器所创建的代理中向外公布的主机名。
-p *port*	指定端点的端口号。	确定要连接到的端口 (必须指定)。	如果没有指定这个选项，或是 *port* 为零，端口将由操作系统选择。
-t *timeout*	以毫秒为单位指定端点超时。	如果 *timeout* 大于零，它会把绑定和代理调用的延迟设为最大。如果发生超时，应用会收到 `Ice::TimeoutException`。	确定在对象适配器所创建的代理中向往公布的缺省超时。
-z	指定使用 bzip2 进行压缩。	确定是否要发送压缩请求。	确定是否要在适配器所创建的代理中向外公布使用了压缩。

参考文献

参考资料

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: Institue of Electrical and Electronic Engineers.
7. Jain, P., et al. 1997. 褶 ynamically 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. 襆 eader/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. 襶 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 詩 eilly.
23. World Wide Web Consortium. 2002. *SOAP Version 1.2 Specification*. http://www.w3.org/2000/xp/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.