第5章 性能要求

在本书的基础部分发现关于性能的一章可能会令读者惊奇。其实，我们觉得性能对于CORBA企业级解决方案是一个很基础的问题，所以我们觉得这里是讨论它的好地方。本章提供了讨论本书余下部分的基础。

用户不时会听到CORBA执行慢的说法。问题在于开发者经常忘记基础网络和分布式基础结构的复杂性，这两样都是CORBA允许用户抽象化的；这种复杂性并没有消失。 CORBA提供了强大的抽象机制，这是极其重要和有帮助的，但这并不是指构建大规模高性能的分布式系统可变为小的任务。抽象是CORBA的主要优点之一，因为它允许开发者着重于商业上的功能而不是系统级的功能。但是抽象并不意味着用户可完全忽略抽象层下面要处理的复杂的分布式系统。纯正的CORBA IDL会导致差的性能和可伸缩性。本章的第一部分看一下 IDL接口设计的性能实质（performance implication）。第二部分从更一般的角度来看一下 CORBA环境中的性能工程。

5.1 IDL设计的性能实质

为分布式对象设计接口比为驻留在单个地址空间的对象设计接口需要更多地考虑性能问题。显然，按重要性顺序，一个简单的内存中方法激发要优于 CORBA远程方法激发 —— 这并不很令人惊奇。从CORBA的角度看，下面的因素对分布式系统的性能有重要的影响：

- 系统中给出的远程方法激发数量。
- 每次远程方法激发中传递的数据量。
- 系统使用的不同IDL数据类型的打包花费。

下面会探讨这些不同的因素如何影响分布式 CORBA系统的性能。例如，在标准的 LAN环境中，远程激发的数量经常有最重大的影响。在低延迟的高速网络中，打包花费就可能变得更显著。本节并不给出准确的性能数字，因为这些对于每个硬件、网络、操作系统、编程语言和对象请求代理的配置来说都是不同的。何况，本节是要帮助读者理解特定 IDL设计对CORBA系统性能影响的程度。因此，本节的性能图表和插图不显示具体的数字，而只显示性能的特征。

5.1.1 访问模式

CORBA IDL接口设计不同于为传统的单个对象模型设计接口。在后一种情况中，所有的对象驻留在一单独的进程空间。在分布式环境中，必须仔细考虑预期的对象访问模式。每个远程方法激发增加一固定的执行开销。忽略这个事实的 IDL设计会导致很差的性能。由于每个远程方法激发的开销，用户必须努力把远程方法激发的数量保持在某一水平，以保证达到需要的性能。

下面使用一个例子来说明这个问题。回忆一下在第 4章中对证券管理器的讨论。用户想设计IDL接口，以输出图4-6所描述的证券管理器对象模型。客户机应用程序将访问证券管理器

组件，以允许用户通过GUI访问其证券。

1. 证券管理器IDL的第1版

忽略访问模式和分布特征，证券和控股的简单接口设计可以如下所示：

```
// CORBA IDL (first version of portfolio example):

#include <StockWatch.idl>

interface Holding;
typedef sequence<Holding> HoldingSeq;

interface Portfolio {
    Money getCurrentValue ();
    HoldingSeq getHoldings ();
};

interface Holding {
    unsigned long getNumberOfShares ();
    Stock getStock ();
};
```

证券管理器GUI显示了一列由实际证券管理器掌管的所有股票。显示代号列表的函数必须得到所有控股的列表，然后再得到每个控制股票的代号。假定 GUI以Java实现，并使用Java ORB来访问远程对象，那么返回并显示所有代号的函数可以实现为：

```
// First version of Java code for Portfolio Manager GUI

public void displaySymbols (_PortfolioRef thePortfolio)
{
    HoldingSeq holdings = thePortfolio.getHoldings ();
    for (int idx=0; idx<holdings.length; idx++) {
      String symbol =
      holdings.buffer[idx].getStock().getSymbol(); // (1)
      holdingList.add (symbol);
    }
}
```

注意(1)中每次for循环执行的时间是很重要的，这里用了两个远程激发： getStock()和getSymbol()。这显示于图5-1中。这个方法的问题在于，如果证券管理器手中有 N只股票，就会有1+2×N个远程方法激发。想一下每个远程激发都要增加可观的开销。如果预计每个证券管理器只持有少量股票，这就不成问题。但如果证券管理器抓着大量的不同的股票，上面设计的系统就会有很长的响应时间。

2. 证券管理器IDL的第2版

下面的证券管理器接口的变化大大改变了证券管理器客户机的访问模式：

```
// CORBA IDL (second version of portfolio example):

#include <StockWatch.idl>

interface Holding;

interface Portfolio {
```

```
    Money getCurrentValue ();
    SymbolSeq getSymbols ();
    Holding getHoldingBySymbol (in Symbol aSymbol);
};

// Holding as before ...
```

这个IDL可由证券管理器GUI客户机像下面那样使用：

```
// Second version of Java code for Portfolio Manager GUI

public void displaySymbols (_PortfolioRef thePortfolio)
{
    SymbolSeq symbols = thePortfolio.getSymbols (); // (1)
    for (int idx=0; idx<symbols.length; idx++) {
      holdingList.add (holdings.buffer[idx]);
    }
}
```

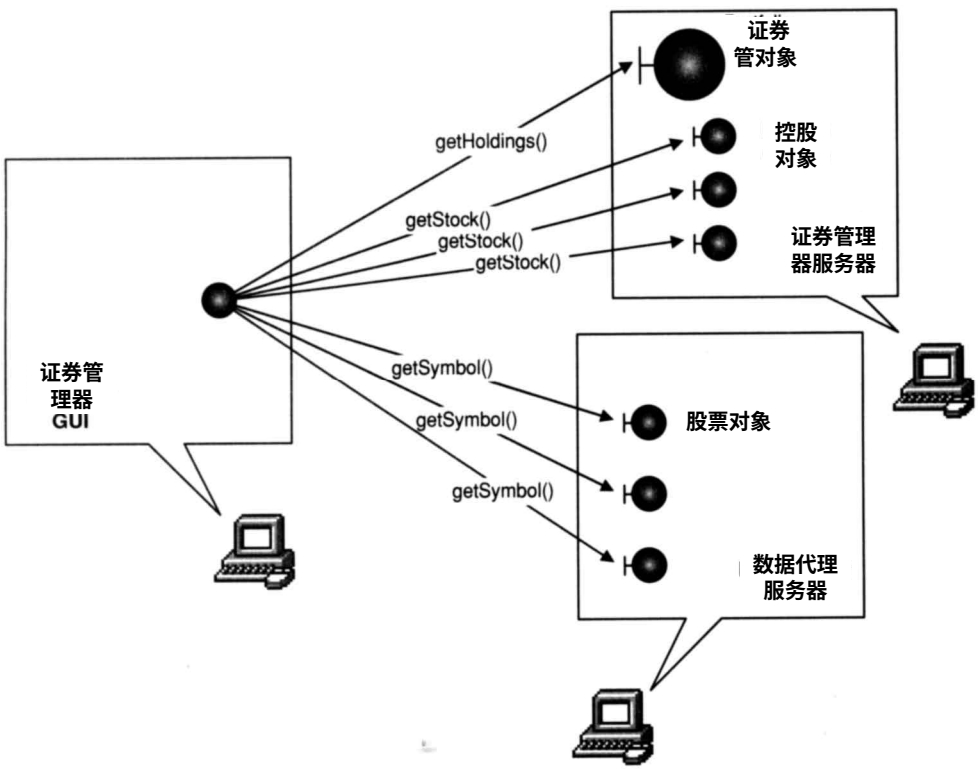

图5-1 displaySymbols()第一版的访问模式

证券管理器IDL的第二版提供了一个操作getSymbols()，它允许用户在一单独的远程激发中得到特定证券管理器持有的所有股票代号的列表。

如果需要，一个控股可在以后通过调用getHoldingBySymbol()得到。例如，如果用户选择一特别的控股以获取更详细的信息，这个操作就会被调用。 IDL第二版的访问模式如图5-2所示。

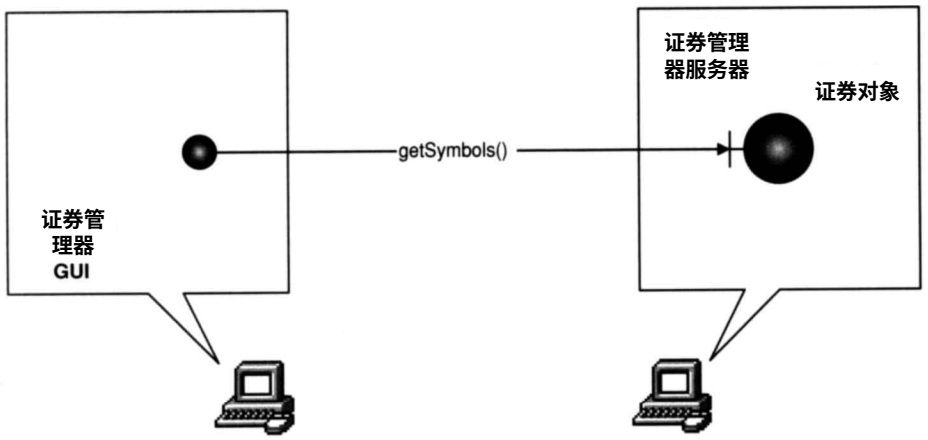

图5-2 displaySymbols()第二版的访问模式

3. 比较

现在从性能的角度来比较两种不同的 IDL版本。图5-3显示了当证券管理器持有大量不同股票类型时两个证券管理器 IDL版本的区别。可以看到第二个版本通过重要性的排序改善了性能。这是因为开始时所确认的三个因素：远程激发的数量、每次远程激发所传递的数据量和请求参数的数据类型。读者早已注意到第一个版本在每次 displaySymbols()中发出1+2×N次远程激发，相对地，第二个版本只发出一个远程激发。那么数据量呢？两个版本中传递相同数量的代号（string），但在第一个版本中还要传递 2×N个对象引用（Holding，Stock）。用户不仅要传递大量的数据，而且还要传递一个更高打包花费的数据类型：对对象引用的打包花费比对简单数据类型高得多。图 5-4显示了第一个版本 displaySymbol()花费的统计分析。下面会更详细讨论不同IDL数据类型打包的开销。

图5-3 两个displaySymbols()版本的比较

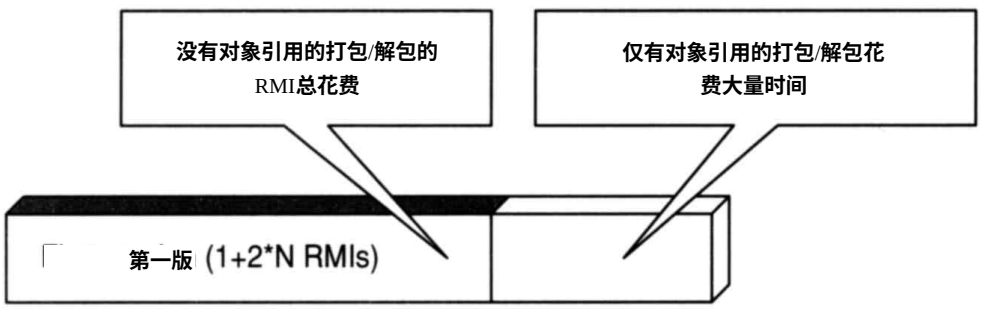

图5-4 displaySymbols()第一版花费的减少

5.1.2 影响性能的重要因素

三个对CORBA应用程序性能有重大影响的因素是远程激发的数量、每次远程激发传递的数据量和伴随激发所传送数据的打包开销。下面会更详细讨论每一个因素。

1. 远程激发数量

每个通过网络连接发送的请求都必有一个延迟，这通常称为“网络延迟”。这个延迟对每个CORBA远程激发增加了相当可观的处理时间，特别是在TCP/IP网络中。每个请求和回答所强加的网络延迟并不是每次远程激发中唯一的消耗因素。还有其他的因素，例如服务器端对象查找，它向每次远程激发贡献一固定的消耗因素。用户会发现这个固定消耗因素是远程激发数量经常比每次请求所传递数据量更为重要的原因。

2. 消息尺寸

从上面可得到一简单的结论：高处理速度可以这样实现：使用少量请求，每次传递大量的数据，而不是用多个请求，每次只传递少量的数据。但在单个消息中传递极其大量的数据又有另一个问题——如果消息太大，处理速度就会下降。下降的原因包括TCP缓冲区问题和进程空间的增加。

图5-5显示了系统平均消息尺寸和系统平均吞吐量之间的依赖关系。这个图形随网络不同而变化，特别是在比较WAN、LAN、低速和高速网络的时候。其他的因素也扮演重要的角色——例如IP缓冲区大小[⊖]的配置。作为一般的抽象，在大多数网络中的吞吐量曲线有三个特征段：

段	消息尺寸	吞吐量
I	小	低
II	相对大	高
III	极大	低

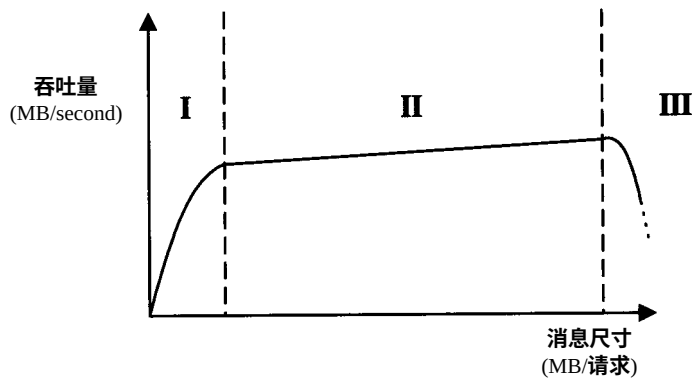

图5-5 吞吐量与消息尺寸的依赖关系

3. 不同IDL数据类型的打包开销

图5-6比较了打包和解包一些不同CORBA数据类型的相对花费（本节余下部分中把打包和解包花费归入术语“打包开销”）。图中所示的相对花费对不同的ORB、编程语言、操作系

⊖ 关于CORBA性能的一些优秀论文可在St.Louis的华盛顿大学计算机科学系的Douglas Schmidt的主页中找到（http://www.cs.wustl.edu）

统、硬件平台和网络类型是不同的。这幅图的目的是帮助读者理解打包不同 IDL数据类型时，相对花费重要性的排序。下面简要讨论每种数据类型：

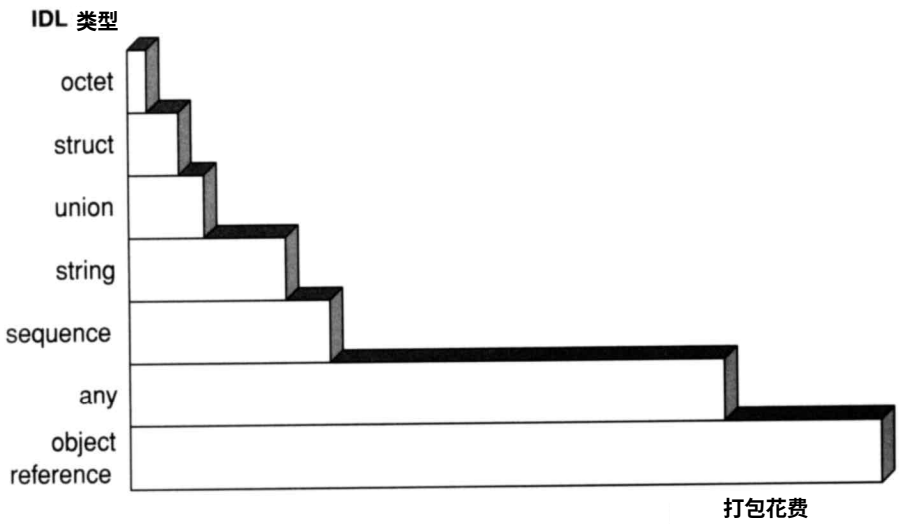

图5-6 打包/解包不同IDL数据类型的相对花费

- Octet octet是CORBA IDL提供的最简单数据类型。octet的打包开销是很少的。ORB甚至不负责octet的字节顺序。
- Struct struct同样也很简单，因此它的打包开销很少。显然， struct的打包开销由struct的成员类型来决定。图中指出了struct本身的开销，而没有包括struct所含的任意数据。
- Union union比struct稍为复杂，因为unions可以存储不同类型的数值。因此，打包开销也比struct稍高。Union的打包开销取决于union中所含数据的类型。图中只是指union本身的开销。
- String string是一长度可变的数据类型。长度可变数据类型的问题是它们通常包括很耗费内存的管理操作。例如，拷贝一 string序列时，ORB不能立即为所有strings预分配内存，而必须为每个string单独分配内存。通常，读者会发现可变长度数据类型比固定长度数据类型更加耗费资源。
- Sequence sequence不仅是可变长度数据类型，它还可存储复杂的 IDL类型。同样，sequence的打包开销取决于它的成员类型。
- Any any类型是IDL提供的最复杂的基本数据类型，因为它本质上是动态类型，而且在打包和解包期间必须处理动态类型代码。 Any类型的打包开销极大地依赖于编程语言（一些语言比其他语言更适合支持动态类型）和ORB实现。通常，any比其他基本数据类型更加耗费资源。
- Object Reference 对象引用提供了强大的抽象机制。它们封装了如主机地址和对象标识等信息，并映射到激活的网络连接。这种抽象的代价是相对高的对象引用打包开销。通常，接收器端对象引用的解包开销特别大，因为它包括代理对象的创建。

可以看到，IDL数据类型的选择严重影响系统的性能表现。高度结构化的数据导致打包和解包开销的增加。不仅是打包的时间增加；表示数据的消息尺寸也增加。打包例程需要加入关于数据结构的信息，这样解包例程就可从平展的消息中重新创建原始结构。

5.1.3 设计例子

在下面的一些例子中，考虑到了这些影响性能的因素。第一个例子解释了次级OID的概念，第二个例子讨论了递归数据结构。

1. 使用次级对象标识（OID）

如果IDL接口提供了一个操作，它能返回类似对象引用序列的数据，我们就必须了解这样做的后果。如果用户希望操作通常只返回少量或适度的对象引用，那就没有问题。但如果操作返回大量的对象引用，这很可能会导致极差的性能。首先，对象引用的打包开销是相当可观的。其次或甚至更糟，传递大量的对象引用经常会伴随大量的远程方法激发，因为对象引用的接收者可能会依次使用每个远程方法激发。

不返回大量的对象引用，而是使用次级对象标识（OID）和数据结构的结合则是一更好的主意。次级对象标识是某种IDL数据类型，唯一地标识CORBA对象，这个对象常常和另一个CORBA对象相关。例如，在StockWatch IDL中，使用Stock代号作为次级OID来标识相对于一个证券管理器的控股。这里不是返回大量的对象引用，而是返回次级 OID，并提供一个方法，例如getStockBySymbol()，以允许客户机以交换次级OID来获取CORBA对象引用。

但仅是这样还不能解决问题。通常，返回大量对象引用的操作由客户机为某种选择过程而使用。例如，displaySymbols()函数向用户显示代号，用户然后选择其中一些代号来得到更详细的信息。为防止客户机向所有可能的对象进行激发，要向客户机提供足够的信息，以让它在本地执行选择过程，并接着处理少量选出的对象。为选择过程返回含有足够信息的数据结构，并加上次级OID来识别真实对象通常是一个好主意。假定用户想实现一系统来管理雇员。一个雇员表示为第一类的CORBA对象（即有IDL接口的对象）。用户需要一管理员接口来选择特定的雇员。最初，这个选择可用雇员的名字和通配符的结合来完成。由于名字不是唯一的，而通配符可能会匹配不同的名字，查询结果就极有可能含有多个雇员。从这个初始的查询结果中，使用称呼、名和姓就可选择一特定雇员。

```
// IDL
typedef unsigned long EmployeeID;
interface Employee;

struct EmployeeData {
     string m_title;
     string m_firstName;
     string m_lastName;
     EmployeeID m_id;
};
typedef sequence<EmployeeData> EmployeeDataSeq;

interface EmployeeAdmin
{
     EmployeeDataSeq queryUsingWildcard (in string name);
     Employee getEmployeeByID (in EmployeeID id);
};
```

EmployeeAdmin接口允许管理员使用通配符得到一系列雇员的数据，然后从结果集合中选出一特定的雇员，并以特定雇员对象来继续处理。

第一个查询结果表示为数据结构的序列。每个数据结构包含一个次级OID（EmployeeID），

并加上一些附加信息。附加信息用于选择过程。这是十分重要的：查询结果包含足够的信息在客户端执行选择，这样就不需要和服务器的交互。只有当选择了一特定的雇员，客户机才能使用getEmployeeByID()来得到真正的雇员对象。

2. 递归数据结构

这个例子基于图元和图，一个图包含一图元集。有不同类型的图元：包括起始点和终止点的直线；一个复合图元是一组图元。这和对象模型一起在图 5-7中显示。

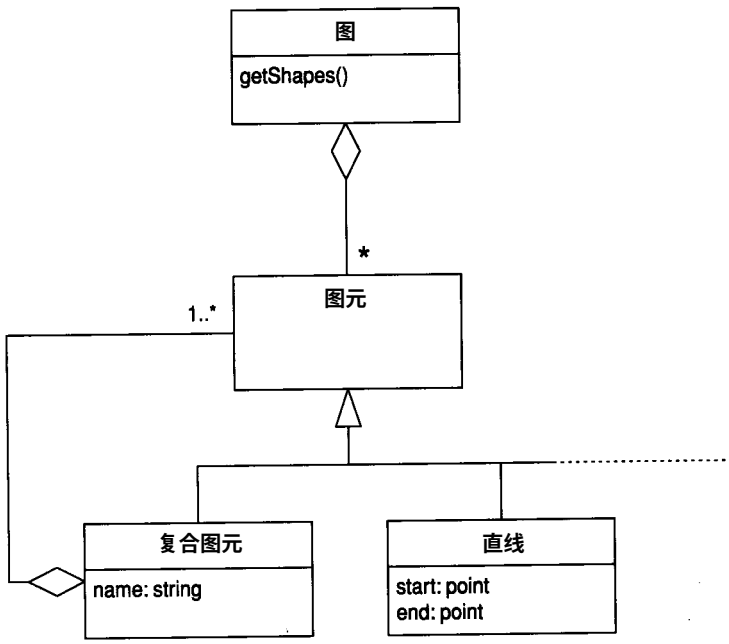

图5-7 递归类层次

这里有一个具有递归聚集的简单继承层次：CompoundShape一方面继承Shape，另一方面聚集了一系列的Shape对象。

我们的任务是把这个简单的对象模型以一种方式映射到 IDL，以允许图元以值来传递，即作为数据而不是对象引用传递。这意味着我们可使用像 struct、union或sequence等IDL数据类型。IDL类型interface不能用于图元。例子中唯一的接口是 Diagram，它提供操作getShapes()。这个操作可被客户机用来获取形状数值的序列。

把对象模型映射到IDL数据类型而不是IDL接口，通常是一个不小的任务。问题是下列限制应用IDL数据类型：

- IDL数据类型不支持继承。
- IDL数据类型不能用来描述递归数据结构。 ⊖

下面接着探讨这些问题的工作区。但是，这些工作区会导致 IDL不够清楚或难于阅读。

把对象模型映射到IDL数据类型的一个方法是把对象模型的类映射到 IDL structs。使用这个方法，聚集关系可以轻易映射到 IDL序列。映射继承更加困难。通常，对象模型中

⊖ 对这些迭代类型声明规则有一个例外 —— 在新类型本身有一迭代类型声明是不可能的，例如，如果新类型作为序列成员使用。下面是正确的迭代类型声明的一个例子： struct A {sequence<A> m_aSeq;};

的继承可映射到IDL级的聚集。这经常是使用IDL类型union来完成的。基类可以用类型union的成员映射到struct，该struct可用来存储任何子类的属性。如果基类没有属性，它就可直接映射到union。第二个限制是递归类型声明。在下面的例子中忽略了这个限制导致的无效IDL：

```
// IDL : invalid version of shape example
struct CompoundShape {
    string          m_name;
    sequence<Shape>       m_shapes;          // (1) Error!
};

struct Line {
    point m_start;
    point m_end;
};

union Shape switch (short) {
    case 1: Line          m_Line;
    case 2: CompoundShapem_CompoundShape;
};
```

问题在于(1)中使用了类型Shape，它在IDL文件中的该处还没有被声明。但是，如果把这个Shape的声明移到IDL的头部，就可能会有类似的问题，因为Shape指类型Line和CompoundShape。用户不能在IDL中递归类型声明，因为IDL不允许IDL数据类型的前置声明。前置声明只能用于IDL接口。

那么如何处理递归数据类型呢？一个方法是把IDL数据类型和IDL接口结合起来。这是指一些数据可按值传递，而其他数据只能远程访问。通常，这不是用户真正想要的。另一个方法是使用一般的IDL类型any，它可表示任意的IDL类型。下面的例子使用了类型any来解决递归类型声明问题：

```
// CORBA IDL (first version of Shape example)
struct Shape {
    any m_details;                    // (2)
};

struct CompoundShape {
    string m_name;
    sequence<Shape> m_shapes;         // (3)
};

struct Line {
    point m_start;
    point m_end;
};
```

(2)中在Shape结构中加入了类型为any的域m_details。这个域总是包含CompoundShape或Line。由于Shape现在是一有效的类型，所以可以在CompoundShape声明(3)中使用它。

但是，使用类型any来解决递归类型声明问题会对系统性能有重要的影响；any是最复杂的类型之一，因此通常也是最低效的类型之一。

可以做些什么来优化IDL的效率呢？问题并不是在使用类型any的IDL中的类型声明数量。问题是在运行期所传递的any的数量。在这个例子中，假定90%的形状实际上是直线，而只有

10%的是复合形状。如果找出一种方法来把对象模型映射到 IDL，只有复合形状才会映射到类型any，这就会极大地增强应用程序的性能。下面的例子显示了这个操作如何完成：

```
// IDL
struct Line {                                    // (4)
    point start;
    point end;
};

union Shape switch (short) {                     // (5)
    case 1: Line m_line;
    case 2: any m_CompoundShape;                 // (6)
};
struct CompoundShape {
    string m_name;
    sequence<Shape> m_shapes;                    // (7)
};
```

我们由声明对象模型中的叶子类数据类型开始，但并不会递归回基类。在这个例子中，这只是一个类：Line(4)。接着我们把基类映射到IDL union(5)。这个联合有相对于每个叶子类的一个成员。只有那些退回基类的类才映射到类型 any(6)。最后可以把这些类定义为正常的结构，并加上对基类的引用，这并没有问题 (7)。

这个解决方案的优点是现在可把形状作为联合序列传递，联合在 90%的情况下包含正常的结构。只有10%的序列成员包含any。这并不一定会在性能上提高 90%，但会极大地帮助改善性能。

5.1.4 所有的对象都是合适的，但其中一些比其他更合适

几年来，用户可能会发现对于设计对象模型问题并没有通用的解决方案，但这个过程又需要大量的抽象、直觉和对已存在类图的重画。有了 CORBA，问题上升为哪些对象可变为第一类的CORBA公民，哪些不能。这个决定会受到系统性能的 IDL设计的重大影响。

要记住第一个用面向对象编程语言写的程序：一个图元有一个 draw()方法。直线和圆从图元中继承。用户可以迭代图元序列，并激发每个图元上的 draw()。很快，用户的屏幕就会被直线和圆所填充。讨论过IDL设计对系统性能的影响后，很明显，把细粒度 (fine-grained)的设计，如本例中的那样，直接映射到 CORBA接口所得到的实现会导致极差的性能。这意味着在CORBA世界中，必须经常改变旧的设计策略，并找出新的策略来发现合适的 CORBA对象。Smalltalk程序员可能会皱眉头，但事实上，在 CORBA世界中并不是每个实体都可变成对象，至少在按值传递对象的规范并不是普遍可用（在本章后面讨论）的情况下是这样。即使那样，问题仍然存在：要远程激发哪个对象，哪个对象要按值传递？

这意味着CORBA设计者必须在好的面向对象设计和性能需求间找到平衡点。读者会发现CORBA世界对象经常有不止一个“服务”特征；即它们经常不能提供细粒度的业务对象，而只能提供一个接口在服务器端操纵它们。另外，读者经常会发现服务对象和更细粒度的业务对象的结合：服务对象可用来操纵和选择特定的业务对象，并在选择了特定对象后，客户机可以通过直接和新选出的对象一起工作来继续处理。这是指服务对象提供能够在服务器端影响大量细粒度对象的服务（例如查询），以避免过多的远程交互。因此要去尝试找出在细粒度

的Smalltalk设计方法和极其笼统的过程化方法之间的折衷。通常，读者会发现 IDL设计是语义丰富性和性能之间的协调。

CORBA对象的分布式本质和IDL设计性能实质对面向对象软件开发过程有很重要的影响。从这个角度观察，在本书中也可看出其他重要的方面，在第 18章“工程化过程的重要性”中将描述面向对象软件开发过程的重要性。

5.2 传递大量数据

在目前对与CORBA相关的性能问题的讨论中，可以看到 ORB并没有作为文件传输程序（FTP）来设计——即ORB没有设计为可传输任意大量的数据。但是，用户可轻易地使用 ORB来构建FTP。下面看一下与使用ORB来传输大量数据相关的各个方面。

5.2.1 迭代器

在传输大量数据时，迭代器是相对普通的技术，用以保证优化的吞吐量（即保证系统操作在图5-5所描述吞吐量图中的第 II 部分）。迭代器允许用户在几个部分中迭代大的数据集合，而不用在一个部分中传递整个数据集合。假定用户想在 StockWatch接口中加入操作 queryPrices()，该操作支持对股票价格历史的查询。操作的结果可能是非常大量的。为了不以股票价格序列直接返回这些数据，这里使用了迭代器。下面的 IDL中显示了一个例子：

```
// CORBA IDL (stock price iterator)

interface StockWatch {
    StockPriceIterator queryPrices(in QueryExpression aQuery);

    // Other operations as before…
};

interface StockPriceIterator {
    void getNextChunk (out StockPriceSeq nextChunk,
                       out boolean hasMore);
};
```

这个方法有几个优点。首先用户可优化请求的平均尺寸，以保证得到最佳的吞吐量。此外，迭代器让客户机在接收数据时得到控制权——例如，客户机可控制定时，并可能向最终用户给出更快的响应。注意 CORBA 迭代器对象的实现很重要，在 14.1节内存管理中有对此的讨论。

5.2.2 按值传递对象

标准IDL数据类型有很多限制，这使得处理复杂数据结构变得困难。当前， OMG正致力于CORBA规范的扩展，使得对象可按值传递。这个扩展会引入新的 IDL类型value，它提供一座从普通CORBA结构到CORBA接口的桥梁。新类型 value可以表示复杂的状态（即任意的对象图，包括迭代和循环）。当然value类型总是按值传递。 value没有标识——它们的值就是它们的标识。所有有效的CORBA类型都可用来作为value成员的类型，而且成员可以是公共或私有的。单继承为 value类型而提供。 value类型也可通过继承 CORBA接口类型以支持接口的集合。这意味着value类型支持值的单继承和接口的多继承。 value可以是抽象的。

由于CORBA服务器不能假定客户机的实现语言（反之亦然），所以只有value对象的数据部分可以传递，实现则不能。规范在接收过程中为定位兼容的实现而定义策略。但是，规范只规定了结构上的兼容性，而没有给出语义上的保证。 Value类型中定义的操作，或是继承自基value的操作，都总是在本地执行。继承自接口或是基 value所支持接口的操作则总是远程执行，即在发送进程中的对象上执行。为保证 values能有效实现， value不必注册到ORB。只有当一个value支持一个接口时， value才必须注册到ORB。如果接口操作在value上被激发，而且该value没有注册到ORB，就会产生OBJ_NOT_EXIST异常。

一方面，新的按值传递对象规范提供了一个标准方案，以解决目前由专有的流方案所提出的问题，这在下一节描述。但是，按值传递对象方法违反了基本的 CORBA封装概念。通常，对象的实现是对它的客户机完全隐藏的。为了按值传递对象，对象的接收器必须能够提供该对象的一个实现。这打破了对象的封装，而封装却是 CORBA体系结构的其中一个优点。由于简单的原因，如性能、分布式对象计算的任一结构都不可能提供完全的位置透明性，除非能够消除网络的延迟。按值传递对象可能是增加位置透明性的机会 —— 例如，通过致力于性能问题，如本地和远程访问。但是，这样做会有危险，按值传递对象对于 ORB供应商、IDL设计者、服务器程序员和甚至客户机程序员来说，会把事情弄复杂。它甚至可以证明减少位置透明性是适得其反的。但在现实世界中建立和调配基于按值传递对象的第一个企业类应用程序时，还可以看到这种情况的发生。

5.2.3 专有的流解决方案

除了CORBA按值传递对象规范，还存在一些按值传递对象的专有解决方案。这些实现已经存在了一段时间，它们可被认为是 CORBA按值传递对象规范的先驱。大多数这些专有流解决方案是基于和 CORBA不一致的ORB特征，如IONA Technologies的opaque数据类型：假如一个类型被声明为opaque，但在IDL级上却一点不知道这个类型。 opaque参数的打包和解包例程并不是由ORB提供，但必须由专有的流库或应用程序提供。 opaque类型可用来实现特别的打包和解包方法，例如按值传递对象。对于按值传递对象，在商业上有一些可用的实现，它们使用Orbix的opaque扩展。两个例子分别是来自 Object Space (http://www.objectspace.com)的Streaming<Toolkit>和Rogue Wave (http://www.roguewave.com)的Tools.h++。

5.2.4 声频/视频流

在对按值传递数据或对象的讨论中使用了术语“流动（ streaming）”，意思是把对象从发送者流动到接收者。但是，术语“流（ stream）”也经常用于连续的数据流，这和前面讨论过的简单按值传递对象根本不同。 CORBA 3.0规范包括了控制和管理流的规范。虽然任意类型的数据都可在两点间流动，但这个规范侧重声频和视频流，以及相关的服务质量。这个规范定义流是数据按特定方向的持续流动 —— 即从数据源到数据槽。

这个规范的重点不是为实际数据流传输定义新协议，而是着重于为处理流提供管理框架。规范定义了流和流量接口，以及建立、调整和释放流的操作，另外还有处理服务质量的函数。其目的是为流管理提供一通用框架，以便和各种低级网络协议一起使用。

5.3 小结

我们在本章中看了一下对 CORBA系统性能有最强烈影响的三个因素：远程激发的频率、

传递的数据量和不同IDL数据类型的开销。我们还看到了分布式系统中不同访问模式的影响，以及IDL如何设计，以得到优化了性能的访问模式。

我们对性能作了一些一般性的定量描述，并使用图表来表示它们的关系。显然，作出明确的性能陈述是困难的，因为有太多的因素影响CORBA系统的性能，这些因素的范围从网络和硬件底层结构到编程语言及ORB实现的质量。

最后研究了在CORBA环境中传输大量数据的方法，包括从新的按值传递对象规范到专有的流解决方案。