第二部分　核心服务

本书这部分着重于分布式对象系统中一些重要的方面。这些核心服务是许多 CORBA系统的中心。

第6章探讨了客户机如何获取对象。每个 CORBA系统都必须使对象可被客户机使用，所以对这个课题的理解是很必要的。基本上，服务器把对象引用公布到对象目录中，而客户机随后在这个目录中查找对象。本书讨论一下系统用来实现这个目标的不同机制，以及每种方法的优点和缺点。

第7章讨论了CORBA中对异步、非耦合通信的支持。正常的CORBA激发会令客户机阻塞，直到从目标对象中接收到回答。客户机和服务器也还是紧密耦合的，在它们之间有直接的网络连接。有时候，系统要求不同的处理方式。我们将探讨一下对 CORBA中这样的消息接发特征的当前和将来的支持。

第8章提供了分布式系统中安全性的一个工作定义，以及可以应用的多个不同层次。探讨了CORBA安全性的标准后，将讨论安全性在现存产品中的实际应用。

第6章　对象定位

每个CORBA系统必须设法回答同样的问题：客户机组件如何获取对象引用？有很多可能的方式来回答这个问题，而每个方法又有自己的优点和缺点。解决方案可能易于使用但缺少灵活性和可伸缩性，或是功能强大但却很复杂。同样地，解决方案可能专属于特定的 ORB实现或遵循CORBA。理想的方法是允许客户机轻易地获取任意主机上的任意对象的引用，又保持遵循CORBA，并足够灵活以扩展到大的CORBA系统。

本章中首先介绍客户机如何获取服务器中对象引用的模型。然后讨论遵循 CORBA的获取对象引用的方法，并接触一些特定 ORB机制。这些机制中的每一个都有自己的一套优点和缺点，最好应用在不同的情况。

6.1　定位对象的模型

在令对象对客户机可用的过程中包括三个步骤，如图 6-1所示。这个抽象模型为下面讨论用于定位对象的实用机制作好了准备。

这个模型由几部分组成。服务器和客户机是用户熟悉的应用程序组件，它们分别实现和使用业务对象。第三个组件是对象目录。这部分负责存储对象引用和一些与引用相关联的可选择性的描述性数据。例如，CORBA命名服务是一个对象目录，它存储对象引用，并把这些引用和名字相关联。同样地，CORBA交易对象服务也是一个对象目录，它存储对象引用，并把它们和一系列的属性相关联。即使如存储在配置文件中简单的对象引用字符串也遵循这个

模型。

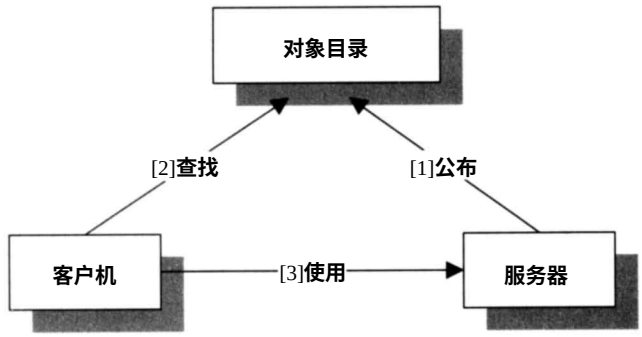

图6-1 输出和定位对象

为了定位对象，系统执行下列步骤：首先，服务器把许多对象公布到目录，提供一些能够以有意义的方式来识别对象的属性。接着，客户机在目录中查找对象。客户机向目录提供一系列所需属性，这样目录就可返回一系列匹配的对象。当客户机从目录处得到一个或多个对象后，它就可以使用这些对象。

当然，服务器如何公布对象以及客户机如何查找对象的细节和特定的对象目录相关。同样地，和对象相关联的属性数量和类型也和每个具体的对象目录实现相关。

什么是对象引用？

这些目录究竟存储什么？服务器实际上没有向目录公布对象本身，它们公布的是对象引用。一个引用含有对象的 IDL类型，以及足够的信息，以让 ORB能够找到在其主机上的服务器进程中的目标对象。当客户机从目录处得到对象引用后， ORB就把这个引用转化为本地编程语言对象 —— 代理 —— 客户机应用程序代码用它来激发远程目标对象。

6.2 CORBA对象定位服务

CORBA规范引入了对象目录的一些实例。 CORBA命名服务和CORBA交易对象服务是最普遍使用的，并且为对象公布和查找提供不同层次的灵活性和复杂性。

CORBA命名服务存储每个对象引用的名字。命名服务还提供层次化的命名空间结构，它允许用户以对业务领域有意义的方式来有条理地组织对象。

有了CORBA交易对象服务，每个对象（交易者术语中的供应者）可以有 any类型的多个属性。交易服务为客户机查找对象提供基于这些属性子集的灵活机制。

这两个服务都有向用户提供把一些附加信息和公布的对象引用关联起来的能力。把特定的应用程序信息依附到对象引用的能力是这些服务的主要价值。它是一抽象层，允许用户在重要信息的基础上定位对象，这些是对用户重要的信息，而不是对 ORB重要的信息。

6.2.1 CORBA命名服务

CORBA命名服务是对象目录的一个简单例子。它在类似于 Unix风格文件系统的层次结构中存储对象引用，所以很多概念都是用户熟悉的。每个对象引用都有一个和它关联的名字。一个名字包含两个字符串域， id和kind。概念上，这些相当于文件系统中的文件名和扩展名。

层次由命名上下文组成，这些上下文包含对象引用和其他命名上下文。在这个意义上，它们相当于文件系统中的目录。命名上下文可存储多个对象引用，而在这些引用的名字结构中，id域和kind域必须有一个与之不同。一个简单的命名服务层次如图 6-2所示。

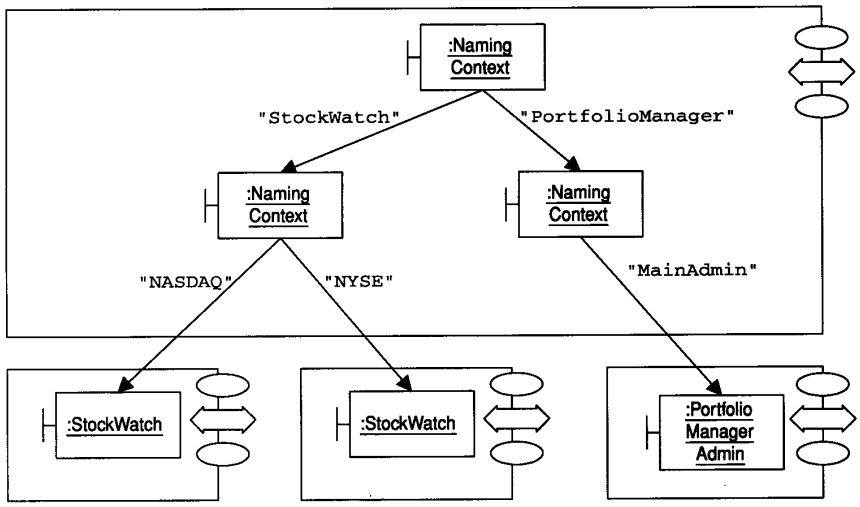

图6-2 命名服务层次范例

根命名上下文对象包含两个元素，这两个元素都是命名上下文对象。每个元素都有和它关联的一个名字；一个是“ StockWatch ”，另一个是“ PortfolioManager ”（在这个例子中只使用名字的id部分，而不用kind域）。“ StockWatch ”命名上下文含有两个元素：“ NASDAQ ”和“ NYSE ”，它们都是服务器公布的应用程序的对象引用。

注意图6-2中显示的命名上下文有一T字架标志，指上下文都是CORBA对象。即实际上应用程序组件通过激发命名上下文对象来使用命名服务。命名上下文支持一些操作，其中只有两个操作对这个讨论是重要的。其图解见图 6-3。这些方法的IDL描述在下面显示。（若要得到命名服务IDL的完整介绍，可参考CORBA规范或命名服务程序员指南。）

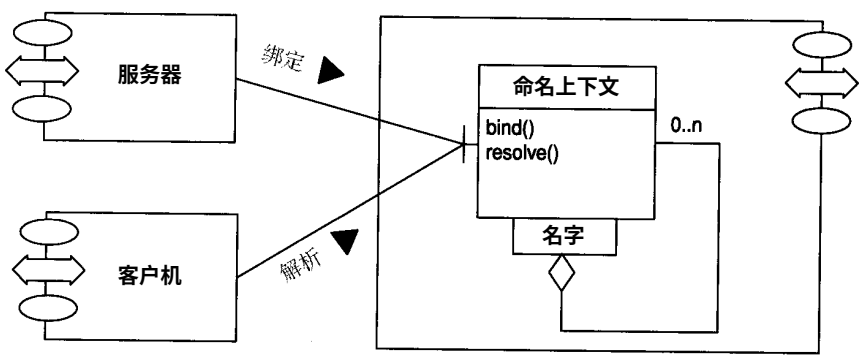

图6-3 命名服务类图

```
// IDL Fragment : CORBA Naming Service
// Simplified IDL : exceptions and typedefs omitted
interface NamingContext
{
```

```
    void bind(in Name n, in Object obj);

    Object resolve (in Name n);

// other methods not shown...
}
```

正如模型所要求的，命名上下文支持允许服务器公布对象的操作以及允许客户机查找对象的操作。下面会讨论这两个方法。

1. 服务器：绑定对象

服务器使用 bind()方法来公布对象。服务器调用命名上下文对象上的这个方法，并提供和该对象关联的Name结构以及对象本身。注意这里实际上是把两部分识别信息和输出的每个对象关联。首先是名字本身，由前面提到的 id和kind域组成。第二部分信息是对象在命名服务层次中的位置。我们可以用这两部分信息来控制名字空间如何组织。这在下面“设计名字层次”中有简要的提及。

注意应用程序对象是作为 Object传递到bind()方法的。这个通用类型允许命名服务用来存储任意的应用程序对象。（回顾一下，Object是IDL中定义的所有CORBA对象的隐含基类。）

2. 客户机：解析对象

客户机通过调用特定命名上下文对象上的 resolve()方法来查找对象。客户机提供命名服务的足够信息来唯一识别一个对象。所要求的名字作为输入参数传递进来，而被调用的方法所在的命名上下文决定服务在层次中的哪个位置执行查找。如果找到一匹配的对象，它就作为通用的Object返回。客户机应用程序简单地把它缩小到恰当的应用程序对象，然后使用它。

3. 设计名字层次

为利用命名服务，要注意的第一件事是层次本身的结构。我们可以把它按需要进行拓展，这取决于业务需求。我们还可选择在命名服务中如何使用输入的 id和kind域。应用这两个方面可以让我们以任意数量的方式来安排层次。通常，我们会使用层次化结构来有条理地分离不同的对象，而相关的对象则组合在一起，用它们唯一的名字来区分。

下面简要地探讨一个例子。想象一下每个 StockWatch服务器应用程序实现了两个接口。一个是用户熟悉的StockWatch接口，它实现用户的业务逻辑。另外一个是 ServerManager，它是一个管理和工具接口。这由系统管理应用程序使用来观察和控制它们正在运行的服务器进程。

名字层次的一个可能结构如图 6-4所示。为了简化名字层次图，这里避免使用正规的记号。作为替换，要使用的是下面所显示的较简单的格式。每个上下文和对象简单地由它的名字标记，名字用id.kind字符串格式。层次中的叶子结点是对象引用。

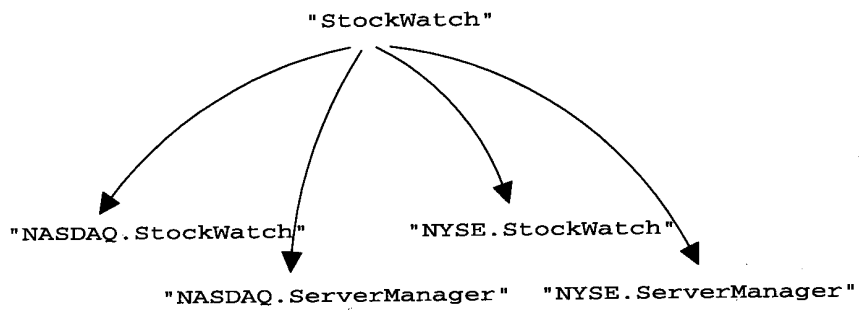

图6-4 平展的名字层次

这里选择了平展的层次，在StockWatch命名上下文中，有两个对象引用具有id NASDAQ。这些引用由它们的kind域区分。同样，在命名上下文中有两个NYSE引用，每个服务器进程支持的接口有一个引用。

作为选择，用户可以选择根本不使用kind域。在这种情况下会有一个更深的名字层次，对于每个股票交易都有一个命名上下文。这些上下文中的每一个都含有两个输出对象。这在图6-5中显示。

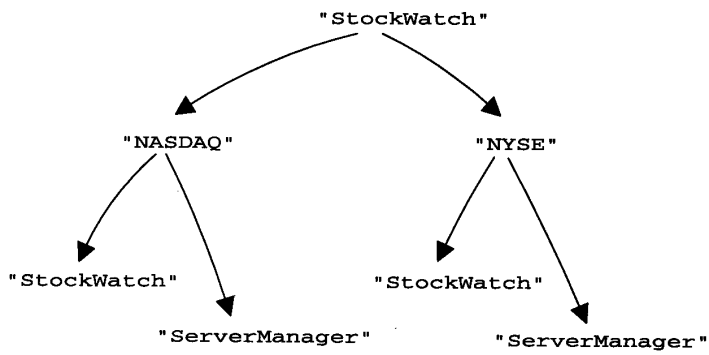

图6-5 深度的名字层次

设计名字层次时，有很多因素要考虑。首先，要考虑应用程序的复杂性，以及该名字层次会覆盖的领域。显然，一个单独部门应用程序的层次比扩展到整个企业的应用程序集合的层次简单得多。还要考虑层次的计划用法。层次的结构是要通过应用程序输出到最终用户吗？或层次是否只能被应用程序和管理器访问？这个因素通常会决定层次是由多个具有描述性名字的上下文组成，还是由具有标准格式的较少名字的上下文组成。

例如，在图6-6和图6-7中显示了两个可能的层次。第一个适合于由最终用户浏览。用户可很容易想象到一个应用程序浏览该层次，并让用户选择他们想用的打印机。把这和第二个层次比较，第二个层次更加紧凑。这个结构最好用于向终端用户隐藏结构的应用程序。应用程序可能曾被管理器配置过，以使用特定的打印机，这样这些名字对最终用户就不可见。图6-7的结构较简单，所以可由更简单的管理和应用代码使用。

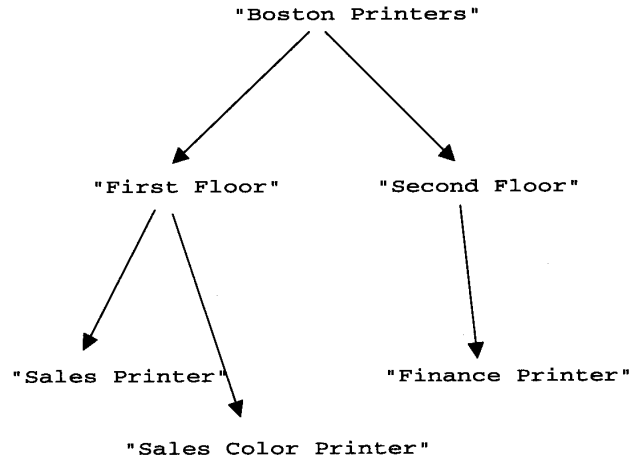

图6-6 描述性命名层次

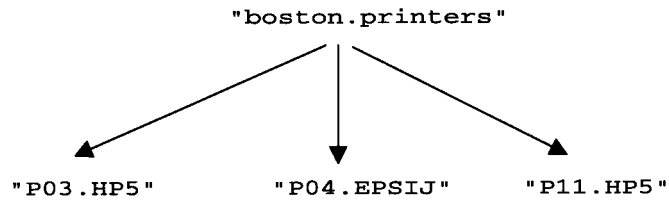

图6-7 紧凑命名层次

还有一个要考虑的因素是服务器应用程序要公布对象的数量。通常可选择输出两种类型的对象。第一，可输出应用程序使用的实际业务对象。图 6-6中的打印机对象就是这样的一个例子。第二，可选择不公布业务对象而公布工厂对象。这些工厂可作为组件中的入口点。它们支持应用程序用来获取业务对象的方法。 PortfolioManager对象就是入口点的好例子。客户机应用程序查找PortfolioManager对象，并用它来得到Portfolio对象，这个对象实现业务功能。在第18章“工程化过程的重要性”中会讨论CORBA组件以及它们服务中的入口点。

另一个要考虑的因素是命名结构要求的服务质量。如果用户想使用该结构来存储大量的对象或很复杂的层次，这就会特别重要。当用户设计好自己的层次后，要评估一下命名服务的能力。考虑机制所支持的持久存储 —— 它是使用基于文件的系统，还是要连接到一个工业用的数据库？用所想要的上下文和对象的数目去评估它的性能。它达到了性能要求吗？考虑一下它的健壮性 —— 例如它支持复制自己的数据存储吗？要保证所选择的产品符合要求。

设计名字层次时，还要考虑命名服务把名字层次联邦（即连接， federate）在一起的能力。要记住命名上下文是简单的对象，在特定的 CORBA服务器中实现。当向层次中插入命名上下文时，实际上是提供新命名上下文的引用。特别地，这个新上下文在相同的服务器中实现，如同所包含的上下文一样。但是，因为这些是简单的 CORBA对象引用，命名上下文实际上可在另一个命名服务进程中实现。联邦的例子在图 6-8中显示。

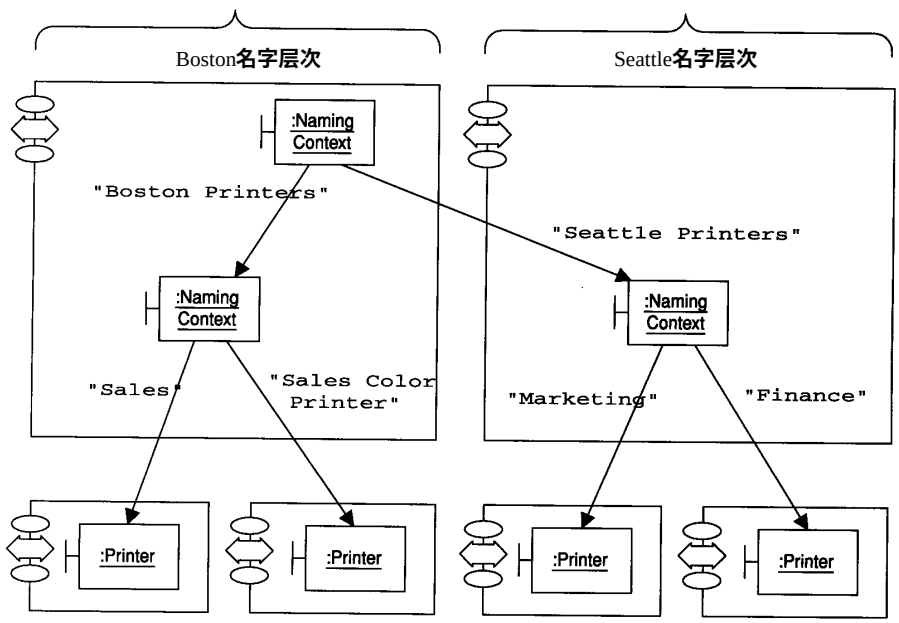

图6-8 联邦的名字层次

这个例子显示了Boston名字层次连接到Seattle办公室中的另一个层次。这些层次由两个独立的命名服务实现存储，运行在独立的主机上。通过联邦它们，可提供对多个层次的透明访问。在这个例子中，基于Boston的最终用户可轻易地使用Seattle办公室中的打印机，这只需简单地导航到名为“ Seattle Printers ”的命名上下文。

联邦命名层次允许用户提供对对象的全局访问，而又保持对这些对象的本地控制。只要这两个办公室同意名字空间的结构和格式，用户就可轻易而又透明地跨组织分享服务。

6.2.2 CORBA交易对象服务

CORBA交易对象服务提供了公布和查找对象的功能强大而又灵活的方式。当服务器公布对象时，它们把任意类型的任意数量的属性和对象相关联。当客户机执行查找时，它们明确规定了一系列所要求属性。交易者评估该寻找查询，并返回一系列匹配对象。

不像名字层次，交易者的对象目录不以任何形式化的方式来组织。相反，交易服务是基于服务类型的概念。一个服务类型含有一个IDL接口标识符，还有一些附加的数据，这些数据定义能和该类型关联的属性。在服务器能实际公布对象之前，必须正确地定义能和该类型对象相关联的属性数量和类型。当定义了服务类型后，服务器就可公布这个类型的对象。在交易者术语中，这称为输出服务供应。服务器输出对象引用，并伴随识别该特定服务供应的属性。这些供应必须遵循由先前定义的服务类型所描述的格式。由服务器向交易者输出的供应集合组成了它的交易空间。

客户机通过执行查询在交易空间中查找对象。客户机明确规定服务类型，以及一系列所要求的属性和限制。交易服务评估供应的准则，并返回一有序的匹配对象集合。

交易对象服务的IDL是复杂的，而且在此不会对它进行完整的解释。相反，这里会用 IDL的抽象来工作，它更易于说明，而且在概念上提供相同的操作集合。要得到一完整而又详细的解释，可参考CORBA规范或交易对象服务编程指南。

图6-9说明了应用程序组件在交易服务中的使用。在下面显示的简化 IDL中包含了必要的参数，用于由交易者实现的每个方法。

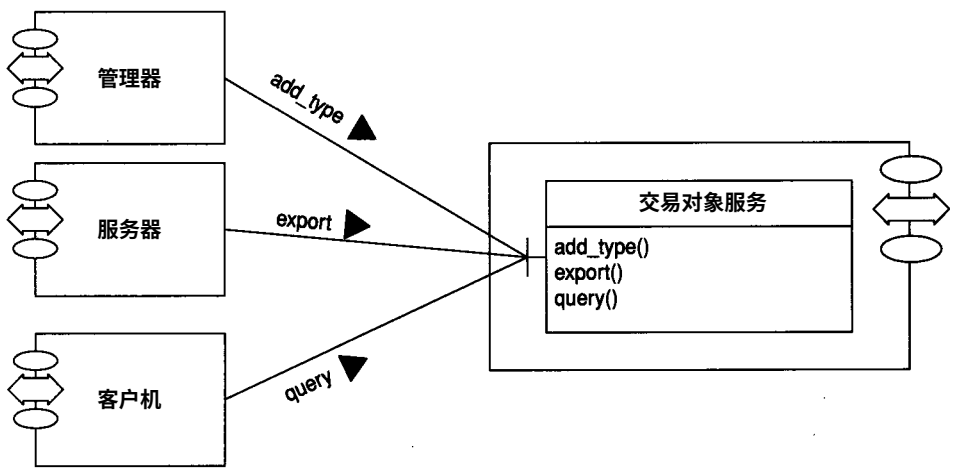

图6-9 交易对象服务图

```
// pseudo-IDL
interface TradingService
{
    // define a new service type
    serviceTypeID add_type(in string serviceTypeName,
                           in string IDLInterfaceIdentifier,
                           in propertyDefinitionSeq definedProperties);
    // publish an offer to the trader
    offerId export(in Object reference,
                   in string serviceTypeName,
                   in propertySeq properties);

    // look up one or more matching offers
    void query (in string serviceTypeName,
                in string constraint,
                in specifiedProperties propertiesToReturn,
                out offerSeq offers);
};
```

1. 管理器：定义服务类型

在使用交易服务时，第一个步骤是定义所支持的服务类型。每个服务类型对应一单独的IDL接口[⊖]，并定义可用来描述这个服务供应的属性集合。下面考察一个简单的例子。它使用OMG CORBA交易对象服务规范中的简单服务类型记号。（注意这不是IDL，它只是用于规定服务类型的方便的记号。）

```
service PrinterService {
    interface Printer;
    mandatory property string          location;
    mandatory property sequence<string>      supportedFormats;
    mandatory property long                  pagesPerMinute;
};
```

这个记号对于严谨地描述服务供应是有用的。它含有IDL接口标识符以及用于这个服务类型的任意数量的属性定义。在这个简单的例子中定义一称为PrinterService的服务类型，它应用于实现Printer接口的对象。前面已定义了和该服务类型相关联的三个属性：location、supportedFormats和pagesPerMinute。所有这些属性规定为强制性的，这是指服务器输出该服务类型的一个供应时，它必须为所有这些属性提供值。被定义的属性可以是任意的IDL数据类型。这个例子使用了一个字符串、一个字符串序列和一长整型。

下面考虑第二个服务类型例子，它更加复杂。

```
service CompanyResearchService {
    interface Company;
    mandatory property string          companyName;
    mandatory property sequence<string>      sectors;
            property float             currentStockPrice;
};
```

这里定义了一个称为CompanyResearchService的服务类型。这个服务类型用来描述实现IDL接口Company的对象，并定义了一些具有不同IDL类型的属性。这些属性中有一个没有定

⊖ 交易服务也支持服务类型的继承，它用于反映相应IDL接口的继承。

义为强制性的，这是指服务器输出的供应不要求向该属性提供值。迄今为止，已提到过的所有属性都是静态属性。静态属性有一个值，这是在服务器端输出其供应时向它提供的。例如，当服务器输出PrinterService供应时，它会向location和supportedFormats属性提供值。

交易服务也支持动态属性。输出供应时，服务器不是提供值，而是提供回调对象。只有在客户机执行查询时，交易者才能获取该属性的值。通过激发这个回调对象就可以完成这个操作。输出该供应的服务器必须决定属性的当前值，并返回它。

一些属性对于客户机能够查询是很重要的，当用户具有这些属性时，动态属性就会很有用，但是动态属性的值会经常改变。在这个例子中， currentStockPrice属性是一动态属性。客户机应用程序要能够基于公司的股票价格找出公司。由于一个公司的股票价格会频繁改变，动态属性可允许用户以标准而有效的方式[⊖]执行这个查询。

2. 服务器：输出供应

当管理员定义了被应用程序使用的服务类型后，服务器就可向交易服务输出其对象。服务器通过激发交易服务中一个对象上的 export()方法来完成这个任务，以提供一个对象、一个识别服务类型的字符串和一个属性的序列。

对象当然是实现了特定IDL接口的对象。特别是，它必须实现和指定服务类型[⊜]相关联的接口。服务器还提供了一属性序列，以名字 -值作为结构。每个属性名字必须匹配在指定服务类型中定义的属性名。服务器也必须为每个属性提供值。这个值可以是具体的（对于静态属性），也可以是回调对象（对于动态属性）。

交易者当然会在供应上施加一些一致性检查。特别地，它会检验输出的对象是否有一个合适的类型，以及是否向所有强制的属性提供了值。

3. 客户机：查询匹配的供应

最后一步是让客户机在交易服务中查找对象。交易者的客户机激发交易者中对象上的query()方法，并提供一些参数。首先，客户机确定所需的服务类型。它必须和前面由管理器定义的服务类型匹配。要记住服务类型定义了一系列属性名和数据类型。当服务器输出供应时，它们会向这些属性中的一些（或全部）提供值。客户机通过提供一系列限制来查询输出供应的集合。交易服务评估该查询，并找出匹配这些特定限制的供应集合。这些供应会返回到客户机。

不像命名服务（它只能返回单独的对象引用作为查找的结果），交易服务被设计为在一个单独的查询中返回多个供应。这些供应并不止包含对象引用，还包含了一些（或全部）供应的属性。客户机对与匹配供应一起返回的属性集合有完全的控制，而不是由交易者自动返回一个（有可能是很多）供应的全部属性。

下面更详细地检验 query()的参数。由客户机规定的限制字符串在概念上类似于 SQL WHERE子句。它规定了属性所需的值，而对应这些值可对特定服务类型的所有供应进行过滤。字符串遵循由OMG规定的格式，以作为交易对象服务规范的一部分。下面看一个限制字符串的示例，在这里客户机查询PrinterService供应。

⊖ 有意思的是，动态属性并不是在服务类型定义时确定。相反，它是在服务器输出每个供应时确定。如果输出服务器选择了相应的属性实现，所有这些属性都可以是动态属性。因此，单独的属性可以在一个供应中具有向它提供的静态值，而在另一个供应中支持动态的评估。

⊜ 输出的对象也可以是继承服务对象接口的类型。

```
(location == 'first floor') and (pagesPerMinute > 10)
```

这个限制字符串检索第一层的供应，而且每分钟打印要多于 10页。

这里是另一个限制字符串示例，在此客户机正在查询 CompanyResearchService供应：

```
('airline' in sectors) and (currentStockPrice < 100.00)
```

这个查询检索属于航空部门的公司，而且它的股票当前价格要小于 100.00。

这个限制语言提供了一灵活的机制以允许客户机规定任意复杂的标准集合，让交易者用来过滤供应。

客户机还规定了应该由交易者返回的属性列表。要记住交易者是向客户机返回匹配供应的序列。这就允许客户机在开始使用一个供应之前，更深入地限定匹配的供应。例如，关于匹配供应的信息可交给用户，然后让用户选择合适的目标。或者，应用程序可看一下返回的值，并在决定使用哪一个供应之前对这些值执行一些额外的处理。客户机提供一specifiedProperties类型，而不是让交易者自动返回一个供应的所有属性的值。这个联合包含一数字类型，指出客户机到底是想要一个供应的所有属性，还是不需要属性，或者想要一些属性。如果客户机想要一些属性，它通过传递一属性名字符串序列来指定这些属性。

交易者向客户机返回一供应序列。每个供应包含对象引用以及所请求属性的名字 -值对序列（也可能是空的）。

4. 设计交易空间

设计交易空间主要是定义服务类型。评估客户机应用程序所需的用来找出对象的方法，并定义组成那些属性的服务类型。然后，服务器简单地输出供应，而客户机则向交易服务查询匹配值。交易服务则完成繁重的任务。

要考虑的一个因素当然是组件逻辑上和地理上的分离。应用程序最好由逻辑上或物理上与它们相近的交易者来服务。但是，一个单独的交易者只能从向该交易者实例直接输出的供应集合中返回匹配值。要克服这个限制，交易服务支持连接的交易者。当交易者连接在一起时，它们可把客户机查询转发到其他交易者，以找出额外的匹配供应。这个连接跟踪行为由交易者的配置和客户机执行查询时所提供的一些额外选项所决定。通过把交易者连接在一起（也称为联邦），可以生成一幅连接交易者图。通过跨领域联邦交易者，可以向客户机提供多样的服务。

联邦当然取决于服务类型的标准化集合。所有连接的交易者必须都同意准确的服务类型、IDL接口和属性名的命名约定。今天，这样的协定只可能存在于特定的组织。但是，由于CORBA的商业接受性变得更加普遍，我们可预见为整个工业领域而定义的标准化交易空间。

6.3 定位对象的其他方法

除了使用CORBA服务来定位对象外，还有其他方法来完成这个任务。这里首先提一下使用对象引用字符串，这是遵循CORBA的方法，符合对象定位模型。然后再讨论工厂模式。工厂是返回其他对象的简单对象。这个模式在 CORBA中普遍使用，而且非常有效。接着，我们会涉及ORB的特定方法。这些专有机制很有用，但经常受到 CORBA服务不曾有过的限制。最后我们讨论自举问题，这是遵循CORBA的方法，它让对象获取一个CORBA服务的引用。

6.3.1 使用对象引用字符串

获取对象引用的另一个遵循 CORBA的方法是使用对象引用字符串。 CORBA规范要求

ORB支持CORBA::ORB::object_to_string()方法。这个方法把提供的对象引用字符串化，把它转化为标准的字符串格式。客户机应用程序激发 CORBA::ORB::string_to_object()方法，而ORB把该字符串再转化为对象引用。

这个方法遵循定位对象模型，虽然这种情况下的对象目录不是一个服务，而只是应用程序设计者选择的一个容器。服务器通过把对象字符串化来公布它的对象，并把字符串写到某个输出地方。逻辑上，这些字符串化的对象引用存储在对象目录中。而实际上，它们会存储在客户端配置文件或外壳脚本中。客户机应用程序通过从字符串的存储地方取得字符串，并把它转化为对象引用来查找对象。

当然，客户机和服务器的程序员必须协调好这些对象引用字符串如何管理。例如，当服务器向一个文件写入对象引用时，客户机开发者必须知道这个文件的名称，以及里面所含的引用指的是什么对象。然后，他们必须把这个字符串传递到客户机应用程序以让它能正确使用该字符串。这个管理上的协作类似于使用命名或交易服务时所需要的协作。在这两种情况下，客户机和服务器开发者必须协调好实体的结构和名字。

6.3.2 使用工厂对象

在CORBA系统中一个普遍使用的模式是工厂对象。工厂是一个对象，该对象的一个方法激发的结果是返回另一个对象引用。读者已经见过工厂对象的一个例子——在证券管理器中的PortfolioManager对象。注意工厂对象并不只是用来创建新的 CORBA对象，它还可用来返回已存在CORBA对象的引用，这是很重要的。如在 PortfolioManager的IDL中所示，下面是其中的一部分。

```
// IDL fragment from PortfolioManager
interface PortfolioManager
{
     // This method creates and returns a new Portfolio
     // object
     Portfolio newPortfolio(in string id,
                            in string password);

     // This method returns a reference to an
     // existing Portfolio object
     Portfolio login(in string id,
                     in string password);
}
```

工厂对象是极其有用的，特别是在客户机能使用大量对象的时候。服务器可以只公布一些工厂对象，而不公布所有伺服对象的引用，客户机可以用这些工厂对象来获取它所需的剩余对象的引用。要注意工厂对于减少某个时间在一个服务器进程中激活的对象数量也是有用的。工厂允许服务器程序员推迟对象的实例化，直至客户机明确地请求该对象，而不需要着急实例化所有可能的对象。这些优点使得工厂对于多数的大规模 CORBA系统都是很重要的。

在上面显示的IDL中，工厂返回一特定 Portfolio类型的对象。另一常见的选择是让工厂返回一般类型Object的对象。例如，这是命名服务 IDL所采用的方法，它允许工厂返回任意类型的对象引用，这只需要求调用者在返回的引用上执行一 _narrow()方法。

6.3.3 特定于ORB的方法

除了支持与CORBA规定一致的方法外，商业上的ORB还典型地支持定位对象的专有机制。

这些机制很容易使用，很受欢迎。

例如IONA Technologies和Inprise的ORB都实现了专有的_bind()方法，这是由IDL编译器为每个接口生成的。这两个机制的详情很不一样，但它们都受到同样的制约——缺少抽象性。

在以上两种专有方法中，客户机程序员都必须提供在ORB领域内能唯一识别目标对象的信息。特别是，当这些信息是与对象实例和系统配置紧密结合在一起的时候。例如，客户机可能会规定一些信息，象伺服对象运行在哪个主机，或ORB的内部对象标识。这里没有什么工具来关联一个对象的高层次信息，客户机只能使用低层次信息来查找对象。

尽管如此，这些专有方法仍然得到广泛使用。在很多场合中，是由于它们的简单性。它们很容易使用，这使得它们成为对象定位机制的第一选择。随着时间的推移，希望对这些机制的使用会减少。新的应用程序将充分利用在这里讨论过的更灵活的方法，同时ORB供应商则会反对这些专有方法。

6.3.4 自举

命名和交易服务本质上是由提供特定接口的伺服对象实现的。组件（包括客户机和服务器）在和这些服务通信时是作为客户机。但是，就像任意的CORBA客户机，用户在使用这些对象之前要获取它们的引用。那么如何得到这些引用呢？推荐的获取引用的方法是使用命名或交易服务。显然，用户需要一些其他的方法来自举程序，这样它们就可以得到这些服务中的某一个服务里的一个对象的初始引用。

一个用于自举且遵循CORBA的方式是使用CORBA::ORB::resolve_initial_references()方法。这个调用输入一单独的字符串参数，并返回一个对象引用。这个字符串识别调用者想从中得到对象引用的服务，该字符串的合法值包括“NameService”和“TradingService”。这些字符串由OMG规定，所有的ORB都必须为相应的服务返回一对象引用。

要得到所需服务的另一个方法是获取以字符串形式表示的对象引用。这个字符串化的对象引用可以是服务本身的输出，也可以由IOR创建工具生成。当客户机得到这个字符串后（例如从配置文件中读取），它们就可以调用CORBA::ORB::string_to_object()来创建对象引用。

不管使用哪个自举机制，当应用程序得到知识库的初始引用后，它们就可简单地使用这个引用来公布或寻找应用程序对象。

6.4 选择对象定位机制

选择使用哪个对象定位机制是一个很重要的决定。选择了不恰当的工具会导致灵活性差、难于维护的系统。虽然各个对象定位机制的复杂度不同，但其中没有一个是真正难使用的。当定义了管理性的基础结构后，所有的方法都可被直接使用（可能由一些为特定环境而定制的客户端封装类来支持）。

这里主要的考虑是环境的复杂性。有多少对象要被公布到对象目录？客户机用来查找对象的标准有多复杂？用户要求什么样的服务质量？确保用户对客户机如何查找对象有一充分的了解是很重要的。

通常，我们建议不要使用与特定ORB相关的机制，原因前面已经提过。它们依赖于专有的机制，并没有工具把高层次信息和对象关联起来，而且不能与其他ORB互操作。

对于很简单的静态环境，使用对象引用字符串是恰当的。这个方法不依赖于任何附加服

务的可用性。客户机只需使用对象引用字符串并连接到合适的服务器。通常，客户机从配置文件中得到一单独的对象引用字符串，然后把这个对象作为工厂使用来得到对所需事务对象的访问。

当然，如果伺服对象移动，客户机就要用新的对象引用字符串来重新配置。对于大量的客户机或是地理上分散的客户机，这样做是困难而且昂贵的。如果这个限制可以接受，那么就可以有效地使用对象引用字符串。

比较命名和交易服务

对于更复杂的环境，命名或交易服务就更加适合。它们要求更多的考虑，但相对于其他方法，它们能向用户提供更多的功能。命名和交易服务在有细小差别的状况下最有用。

交易空间不同于一个名字层次，这是因为后者是多维的。每个服务类型可以有任意数量的属性，而在服务类型中定义的每个属性和所有其他属性正互不相关。另外，可以在服务类型中加入新的属性而不影响已存在的供应。把这和只有两维的名字层次比较。服务器绑定到名字层次的每个对象，只有两部分与其关联的信息 —— 它的名字和它在层次中的位置。

考虑打印机对象，如同在 PrinterService服务类型中所表达的，客户机要能够根据打印机的物理位置和打印速度来定位打印机。在某些场合中，最终用户会用最近的打印机，而在另外一些场合中，他们想找出最快的打印机。正如我们所看到的，支持这些不同类型的查找操作在使用交易者时是很简单的。但是，要用命名服务来完成这个功能却很困难。我们要以两个平行的层次或一深度嵌套的层次来结束。

现在，如果决定在查找对象的客户机中加入第三个属性，考虑一下会发生什么，对于服务类型，增加这个属性，在交易者看来是无足轻重的。但是，向名字层次增加属性则要求对层次进行完全的重构。在这种情况下，显然命名服务不是一个好的选择。

通常，如果客户机只需根据固定的标准集合来查找对象，那么命名服务通常就适合对此建模。另一方面，如果客户机使用变化的标准集合来查找对象，那么交易服务通常就是更好的选择。

6.5 选择公布对象

用户如何决定把哪些伺服对象公布到对象目录呢？答案当然取决于用户的系统是如何构建的，以及客户机是如何使用这些伺服对象的。一些系统包含相对小的、固定的伺服对象集合，所有的客户机都可使用。在这种情况下，公布所有的这些对象是行得通的。其他系统包含相对少的工厂对象，这些工厂对象用来创建客户机临时使用的瞬态对象。在这些情况下应该只公布工厂对象。瞬态对象由工厂返回给客户机，并且由一个客户机专用，含有特定的客户机状态。因此，它们不能普遍可用，没有理由要公布它们。同样，由于这些对象直到客户机请求时才存在，所以公布和查找这些对象是不必要的开销。

现在，考虑一含有大量对象的系统 —— Portfolio Manager系统，它在数据库中有数以千计的Portfolio对象，用户需要决定如何组织命名服务层次。一个方法是为每个 Portfolio对象创建一个入口，由唯一的帐号来命名。另一个方法是用户只需简单地输出 PortfolioManager对象，让客户机使用它来获取特定 Portfolio对象的引用。在某些场合下，这是较好的方法。首先，它使用户不必实例化每个 Portfolio对象，并把它绑定到命名服务。其次，向命名服务加入多个入

口会增加它的数据库规模，并减慢处理速度。第三，通过只输出 Portfolio Manager工厂，就更易于重定位服务器，或是为负载平衡提供一组对象。用户可简单地在命名服务中更换一个入口来暂时（或永远）重定位服务器，而不用列出数以千计的对象。

当系统在一个单独的服务器进程中包含大量的对象时，通常最好是从该进程中输出少量的工厂对象，而不要输出所有的伺服对象。但是，如果系统含有分布在多个服务器进程中的对象，那么输出所有的伺服对象也是可行的。