目　录

第1章 Java 语 言

1.1 Java发展简史

Java语言（简称Java）是由Sun公司的James Gosling发明的，当时他是一个开发消费者电子工业工程的小组成员之一。为了实用，这种语言要产生简短、高效的执行代码。用它编写的程序必须容易地运行于不同类型和处理器上，为了减少使用这种语言开发和维护程序的成本，该语言必须是简单且面向对象的。

Java的突破是由Internet产生的，早期对可移植性的强调在于要求开发在Internet上下载的程序能够运行。因为这些程序可以不加修改地运行在任何能够解释Java语言的计算机上。用Java进行开发，意味着一个Web站点的设计者不必再编制同一程序的多个版本。

为了证明Java在软件开发上的价值，Sun公司使用Java开发了HotJava浏览器。尽管HotJava的最早版本非常粗糙且速度缓慢，但它说明了一个重要事实，即开发真正跨平台的程序是可能的。Java就是完成它的语言工具。然后Netscape把Java集成到了它的浏览器中，Java就像一枚火箭一样起飞了。

从一开始，Java的发展就得到来自于许多硬件和软件计算机的推动。比如，Netscape、IBM和Borland在许多领域对Java语言都做出了巨大的贡献。由于Sun公司对Java抱有无限的信心，使得它把Java的详细说明甚至源代码都公布于众，这样也就创造了第一个真正的标准的跨平台语言。

1.2 Java的语言特性

Java是一种跨平台的、适合于分布式计算机环境的面向对象的编程语言。它具有可移植、安全、面向对象、动态、高性能、简单、与体系结构无关性、动态执行等特性。

1.2.1 面向对象

面向对象是Java最重要的特性。Java语言的设计是完全面向对象的，它不是支持类似C语言那样的面向过程的程序设计技术。Java支持静态和动态风格的代码继承重用。单从面向对象的特性来看，Java类似于SmallTalk，但它的其他特性，尤其是适用分布式计算环境的特性却远远超越了SmallTalk。

“面向对象”目前是一个非常流行的术语，其影响领域从操作系统、编程语言及其开发环境、数据库管理系统直至软件工程。可以说，不管适合不适合，也不管是否真正具有面向对象特性，人们已经习惯了给自己的产品贴上“面向对象”的标签。但事实上，这些

产品有许多并不是真正面向对象的，或者说不是完全面向对象的。

何谓“面向对象”？面向对象其实就是现实世界模型的一个自然延伸。现实世界中的任何实体，都可以看作是对象。对象之间通过消息相互作用。传统过程式编程语言支持一个公式：

程序 = 算法 + 数据

面向对象编程语言也有一个公式：

程序 = 对象 + 消息

所有面向对象的编程语言支持3个概念：封装、多态性和继承。现实世界中的对象均具有属性和行为，映射到计算机程序，属性表示为数据，行为表示为程序代码。所谓封装，就是用一个自主式框架把代码和数据联编在一起，形成一个对象。也就是说，对象是支持封装的手段，是封装的基本单位。对象内的数据和代码，可以是共有的。私有代码和数据只能被对象其他部分访问，公有代码和数据则可以被其他对象访问。一般情况下，对象的公用部分是对象之间交互的机制。

多态性是指“一个对外接口，多个内在实现形式”的表示。多态性的一个典型例子是计算机中的堆栈。堆栈可以用来存储各种格式的数据，包括整数、浮点数和字符。不管存储的是何种数据，堆栈的算法实现却是一样的。在过程式编程中，必须针对每种数据类型专门实现一套堆栈表示和算法，名称和外部接口均必须不同。而在面向对象编程中，只需用一个对外接口和实现即可。针对不同数据类型，编程人员不必手工选择，只需使用统一接口名，系统可自动选择。

继承是指一个对象直接使用另一对象的所有属性和方法的过程。理解继承的一个很好的模型是现实世界中的家族树。事实上，很多实体都可以自上而下进行管理。例如，把个人计算机(PC)看作是一个实体，可以分为多个子实体，如Macintosh,IBM PC,Motorola 68000系列，等等。这些子实体均具有PC机的特性，PC机是这些子实体的父对象，这些子实体为PC机的子对象。继承可简化对象的定义，利用继承，在定义一个对象时，就可以只定义该对象独特于其父对象的特殊品质。

面向对象可以说是一种编程方法学。从这个意义上说，利用传统的面向对象编程语言，同样可以编写出具有面向对象的程序。经常使用的一些所谓面向对象编程语言，包括C++,Object Pascal(Borland Dephi中所用的编程语言)、Perl 5.0和Object C等，实际上只是一种混合型语言，是过程式语言加面向对象的扩展。这些语言支持对象的使用，但同样也支持过程式编程，事实上你几乎总是有意无意地使用一些过程式编程特征。

Java则不然，它是一门“纯”面向对象的编程语言。在一门“纯”面向对象的语言中，语言的任何方面都是基于消息或基于对象的；所有数据类型，无论简单还是复杂，均为对象类。Java实现了标准C语言中的所有数据类型，如整型、字符型和浮点型，这些基本数据类型可以作为对象，也可以不作为对象处理，除此之外的其他任何内容均为对象。这一方面保证了Java的高性能，另一方面又使得Java仍然是一门纯面向对象的编程语言。

1.2.2 可移植性

程序的可移植性指的是程序不经修改而在不同硬件或软件平台上运行的特性。在目前这种标准和开放系统备受推崇的时代，可移植性在一定程度上决定了程序的可应用性。反

过来说，标准的制定以及所谓开放系统的推出，其目的之一也是为了增强程序的可移植性。可移植性包括两种层次：源代码级可移植性和二进制级可移植性。C语言（本书简称为C）和C++语言（本书简称为C++）只具有一定程度的源代码级可移植性，这表明C或C++源程序要能够在不同平台（如DOS和Unix平台）运行，必须重新编译。

Java采用了多种机制来保证可移植性，其中最主要的有两条：

1）Java既是编译型又是解释型的。因此，Java编程人员在进行软件开发时，不必考虑软件运行平台。不仅开发的源代码是可移植的，甚至源代码经过编译之后形成的二进制代码——字节码，也同样是可移植的。任何一台机器只要配备了Java解释器，就可以运行Java二进制代码，而不管这种字节码是在何种平台上生成的。

2）Java采用的是基于国际标准的数据类型。Java的数据类型在任何机器上都是一致的，它不支持特定于具体的硬件环境的数据类型。Java的数据类型采用的是IEEE标准。比方说，Java的浮点数遵循的是IEEE 754标准。

如上所述，Java程序的最终执行需经过两个步骤：编译和解释。Java编译器所生成的可执行代码并不基于任何具体的硬件平台，而是基于一种抽象的机器——Java虚拟机（Java Virtual Machine——JVM）。JVM实际上是一个可运行Java代码的虚拟操作平台。通过预先把Java源程序编译成字节码，Java避免了传统的解释型语言的性能瓶颈，并确保了其可移植性

除给出基于JVM的虚拟机器码外，Java语言还规定同一种数据类型在所有各种实现中必须占据相同的空间大小。C++的数据类型在不同的硬件环境或操作系统下占据的空间是不同的，如整数类型在Windows 3.11下为16位，而在Windows 95下却是32位。通过在数据类型的空间大小方面采用统一标准，Java成功地保证了其程序的平台独立性，或者所谓的“体系结构中立性”（Architecture Neutral）。

此外，Java的可移植性还体现在Java的运行环境上。Java编译器是用Java语言本身所编写的，而其他运行时环境则是用ANSI C编写的，整个运行时环境体现了一个定义良好的可移植性接口。最后，Java语言规范遵循POSIX标准，这也是使Java具有良好可移植性的重要原因。

Java的可移植性具有深远意义。长期以来，无论是对网络软件还是单机软件而言，使应用软件能够在任意平台上运行一直都是编程人员所梦寐以求的事情，如今Java终于使大家的梦想成真。有了Java开发工具，今后的编程人员在开发应用软件时，再不用分别开发IBM PC机版本、Macintosh版本和工作站版本等等，而只需开发一个“通用”的最终软件，这将大大加快软件产品的开发。

更进一步地讲，Java的可移植性还迎合了目前很时兴的“网络计算机”的思想。所谓“网络计算机”，就是指未来的专门用来连接到网络环境的计算机。这种机器使用廉价的芯片，没有硬盘或硬盘空间非常有限，依靠从网络服务器下载应用程序和内容来运行，运行结果也存放在服务器上。可以想象，如果各种应用软件都能用Java重新编写，并且放到某个Internet服务器中，那么未来的网络计算机的用户（同时也是Internet用户）将不再需要安装那些需占用大量空间的软件。他们只需有一个Java解释器，每当需要使用某种应用软件时，从Internet下载该软件的字节码即可，运行结果也可以发回到服务器。这在各种应用软件所需空间越来越大的今天无疑具有非凡的意义。

此外，这同时也给未来的软件产业带来了一种新的思路。今后的软件销售也许将不仅仅是今天这样的一次性销售的方式，到时候将可能出现一种“计时收费”的软件销售方式，用户在需要时才下载某个软件的字节码，并且只需要为自己的使用时间付费。这样，诸如版本更新、软件费用昂贵等问题，都将迎刃而解。

总之，Java 的可移植性决定了这种新型语言将成为未来的网络环境上的“世界语”。

1.2.3 稳定性和安全性

分布式计算环境要求软件具有高度的稳定性和安全性。熟练的C++程序员，可能已经知道：C++程序在稳定性方面的最大问题，在于其指针的使用和缺乏主动的内存管理。这意味着：C++程序员完全可以编写出在语法和语义上均正确，但却能对系统产生巨大破坏作用的软件。

鉴于这些原因，为保证稳定性，Java采用了 3 个措施，首先，Java不支持指针数据类型，这样，程序员便不再能够凭借指针在任意内存空间，甚至是操作系统的内存空间中“遨游”；其次，它提供了数据下标的检查机制，从而使网络“黑客”们无法构造出类似C和C++语言所支持的那种指针；最后，Java还提供了自动内存管理机制，即一个自动的“内存垃圾”搜集程序。

Java运行时，环境保证字节码的传输过程中使用公开密钥加密机制(PKC)外，还提供了如下四级安全保障机制：

（1）字节码校验器（ByteCode Verifier）；

（2）类装载器（Class Loader）；

（3）运行时内存布局；

（4）文件访问限制。

当Java字节码进入解释器时，首先必须经过字节码校验器的检查，这是非常重要的。即使Java编译器生成的是完全正确的字节码，解释器也必须再次对其进行检查，因为在从Java程序的编译直到解释执行这段时间内，字节码可能被有意无意地改动过。

然后，Java解释器将决定程序中类的内存布局，这意味着“黑客”们将无法预先得知一个类的内存布局结构，从而也就无法利用该信息来“刺探”或破坏系统。随后，类装载器负责把来自网络的类封装到其单独的内存区域，避免应用程序之间的相互干扰或破坏作用。

最后，客户机一端管理员还可以限制从网络上装载的类只能访问某些被允许的文件系统。上述机制综合在一起，使得Java成了最安全的编程语言和环境之一，并且保证了Java代码无法成为类似特洛伊木马、病毒和蠕虫等具有潜在破坏作用的东西。

1.2.4 简单性

Java语言简单而有效，这与Java的起源有很大关系。Java最初是为对家用电器进行集成控制而设计的一种语言，因此它必须具有简单明了的特性。在一开始，Gosling等人曾经考虑过直接用C++编程语言，但由于C++过于复杂，存在大量冗余，再加上诸如安全性等

各种因素，才不得不忍痛割爱，并决定重新设计一种新的语言，也就是现在的Java。

Java的主要设计目标之一就是尽可能类似于C++，尽可能采用面向对象的编程风格。另一种设计目标则是希望从C++中消除其可能给软件开发、实现和维护带来麻烦的地方，包括其二义性、冗余和存在安全隐患之处，如操作符重载、多继承和数据类型自动转换等。综合来说，Java语言的简单性主要出于如下几种因素：

（1）Java的风格类似于C++，因而对C++程序员而言是非常熟悉的；从某种意义上来说，Java语言本身是C及C++的一个变种，因此，C++程序员可以很快掌握Java编程技术；

（2）Java摒弃了C++种容易引发程序错误的地方，如指针和内存管理；

（3）Java提供了自动内存垃圾搜集机制，从而减轻了编程人员的进行内存管理的负担，有助于减少软件错误；

（4）Java是完全面向对象的，它是最容易学习的面向对象编程语言之一；同时它还提供了大量可重用的类库。

Java的简单性是以增加运行时系统的复杂性为代价的，以内存管理为例，自动内存垃圾处理减轻了面向对象编程的负担，但Java运行时系统却必须内嵌一个内存管理模块。但无论如何，对编程人员而言，Java的简单性只会是一个优点，它可以使我们的学习曲线更趋合理化，加快我们的开发进度，减少程序出错的可能性。

1.2.5 高性能

正常情况下，可移植性、稳定性和安全性几乎总是以牺牲性能为代价的，解释型语言的执行效率一般也要低于直接执行源码的速度。但Java所采用的措施却很好地弥补了这些性能差距。这些措施如下。

1）多线程

线程也被称作“轻量(lightweight)进程”，线程的概念提高了程序执行的并发度，从而可提高系统效率。C和C++采用的是单线程的体系结构，均未提供对线程的语言级支持。与此相反，Java却提供了完全意义的多线程支持。

Java的多线程支持体现在两个方面。首先，Java环境本身就是多线程的，它可以利用系统的空闲时间来执行诸如必要的垃圾清除和一般性的系统维护等操作；其次，Java还提供了对多线程的语言支持。利用其多线程编程接口，编程人员可以很方便地写出支持多线程的应用程序，提高程序的执行效率。

必须注意的是，从Java语言规范可以看出，Java的多线程支持在一定程度上可能会受其运行时支持平台的限制，并且依赖于其他一些与平台相关的特性。比方说，如果操作系统本身不支持多线程，Java多线程就可能只是“受限”的或不完全的多线程。

2）高效的字节码

Java编译器生成的字节码和机器码的执行效率相差无几。Java字节码格式的设计充分考虑了性能因素，其字节码的格式非常简单，这使得经由Java解释器解释执行时可产生高效的机器码。据统计，Java字节码的执行效率非常接近于由C和C++生成的机器码的执行效率。

最后，Java运行时环境还提供了另外两种可选的性能提高措施：及时编译和嵌入C代码。及时编译是指在运行时把字节码编译成机器码，这意味着代码仍然是可移植的，但在一开始会有一个编译字节码的延迟过程。嵌入C代码在运行速度方面效果当然是最理想的，但会给编程人员带来额外负担，同时将降低代码的可移植性。不过不管怎么说，利用这些性能提高措施，Java的确可以达到与C++几乎相同的执行效率。

1.2.6 动态特性

Java的动态特性是其面向对象设计的延伸。这种特性使得Java程序能够适应不断变化的执行环境。类库的使用是最明显的一个例子。如果你曾经编写过C++程序，那么对类库的概念应该不会陌生。用C++编写的应用程序经常会用到各种类库，不仅仅是基础类库。很多时候还需要一些从第三方厂商处购买的类库。销售应用程序时，类库有时是单独出售的。对于C++应用程序，这就会导致一个问题：类库一旦升级，用这些类库编写的应用程序就必须重新编译，并且重新发送到用户手中，否则就无法利用升级后类库的新增功能。

Java的“滞后联编”却避免了这个问题。“滞后联编”机制使得Java完全利用了面向对象编程模式的优点。如前所述，Java程序的基本组成单元为类，这些类是在运行过程中动态装载的。因此，Java可以在分布式环境中动态地维护应用程序及其支持类库之间的一致性。这样，对于Java而言，其支持类库升级之后，相应的应用程序不必重新编译，也一样可以利用升级后类库的新增功能。

除此之外，Java的动态性还体现在其对动态数据类型和动态协议的支持上。利用一种特殊的Applet，即内容句柄，编程人员可很方便地使HotJava支持新的数据类型。类似地，通过编写协议句柄，可以使HotJava支持新的、自定义的传输协议。

Java的动态性有何价值？简而言之，Java的动态性使你能够真正拥有“即插即用”（plug-and-play）的软件模块功能。

1.2.7 分布式

分布式包括数据分布和操作分布。数据分布是指数据可以分散存放于网络上的不同主机，操作分布则指把计算分散由不同主机进行处理。Java支持WWW客户机/服务器计算模式。因此，它可以支持所有这两种分布性。

对于数据分布，Java提供了一个URL对象，利用此对象可以打开并访问网络上的对象，其访问方式与访问本地文件系统几乎完全相同。对于操作分布，Java客户机/服务器模式可以把运算从服务器分散到客户一端，提高整个系统的执行效率，避免瓶颈制约，增加动态可扩充性。

对于编程人员来说，Java的网络类库是对分布式编程的最好支持。Java网络类库是支持TCP/IP协议的子例程库。目前支持的协议有HTTP和FTP等。同时，通过编写协议句柄，编程人员还可以扩充Java支持的协议集合。

以上介绍了Java的主要特征。由这些特征可以看出，Java的确是一门适合Internet和分布式环境的编程语言。

1.3 Java语言的特点

1.3.1 包、接口和类

1）包

包是一组类和接口，它是用来管理庞大的命名空间和避免冲突的工具。每个类名和接口名都包含在某个包中。包名由分离的字段组成，段与段之间用“.”隔开。

Java语言提供一种机制，它可以使类和接口的定义和实现跨包有效。import关键字用于标志那些被当前包所引用的类。编译单元可以自动地引用本包中的每个类和接口。

使用import语句引用类、接口或者是包含它们的包的方法如下：

```
import Java.net.*;
```

它表示每个Java.net中的公共类都被引用。

2）接口

接口是方法定义和常量值的集合。Java通过接口可使处于不同层次的类具有相同行为。由于接口中只进行方法的说明，而不提供方法的实现，因此接口中所说明的方法，尚未实现方法体。接口可以说是抽象类的一种极端形式。接口可以提供对方法协议的封装，这些方法的实现没有限制在同一继承树上。当一个类实现接口时，它通常要实现接口中所说明的所有方法体（如果实现接口的类是抽象的——从来不实现——它将给它的子类留下部分或全部接口方法）。接口的一般定义格式如下：

```
interface InterfaceName [extends surperInterfaceList]          //接口说明
{
    type constantName=Value;            //常量说明，可有多个
    return Type methodName([paramList]); //方法说明，可有多个
}
```

接口能够解决一些多重继承性能处理的问题，相对比较起来在运行时开销较小。然而，由于接口涉及动态方法连接，在使用时必然有一些小的性能损失。

在使用接口时，允许多个类共享一个程序接口，使它们具有相同的行为，而无须考虑这些类之间的继承关系。

一个类也可以实现一到多个接口。当需要在类中实现接口时，必须借助implements关键字。如下所示：

```
public class ClassName extends superClass implements interface[,option interface]
{
  //class body
}
```

接口的概念给Java带来了许多好处：通过创建一个接口，可以先定义一个抽象类的协议，而不管其具体实现，即把具体实现放到以后再行定义，这在有些场合是非常有用的；其次，有了接口的概念，多个类可以共享相同的接口，相互之间不用管其他类是如何定义

该接口的方法。如果继承了接口的属性，其他用户就可以了解到如何调用这个类的方法。比如说，如果希望所有的Shapes类都有一个draw()方法，就可以创建一个接口，并把它命名为Shape。如下所示：

```
public interface Shape
{
    void draw();
}
public class Triangle implements Shape
{
    void draw()
    {
        //draw implementation
        }
    ...
}
```

这样，每当需要使用Shape时，就可以知道这个类有一个draw()方法，同时还可以确定这个方法都有哪些参数。

3）类

“类”(Class)是基于对象之上的一个概念，类本质上可以认为是对象的描述，是创建对象的“模板”。

类是组成Java程序的基本要素，类代表了经典的面向对象程序设计模型。在Java中，一个类的通常定义形式为：

```
[类的修饰字] class 类名 [extends 父类名]
{
.....
}
```

建立一个新类，程序员必须基于已有旧类的基础，新类是由已存在的类衍生出来的。衍生的类被称为子类，已有的类被称为其父类。类的衍生是可传递的。如果B是A的子类，C是B的子类，则C也是A的子类。

在类的说明中，用关键字extends来指明一个类的直接父类。

所有的类都是由单一的根类——Object衍生出来的，除Object类外，每个类只有一个直接父类。如果在说明一个类时没有指明它的直接父类，那么，就认为Object类是它的直接父类。

Java语言只支持单一继承性。但通过接口性能，它可以支持某些在其他语言中通过多继承性支持的性能。

1.3.2 出错与异常处理

对计算机程序来说，错误和异常情况都是不可避免的。因此，一门编程语言如果不提

供良好的出错与异常情况处理机制，将会是难以想象的。Java提供了丰富的出错与异常情况处理措施。作为一个纯面向对象的语言，它把这些都封装到了各种出错处理类中。

对于编程领域来说，错误和异常情况之间的差别非常微妙，两者经常被互换使用，但仍然存在一些差别。“错误”(Error)通常是指程序（或Java解释器）认为非法的情形，这些情形常常是因为代码本身存在的问题而发生的。而且，如果编程人员可以对程序进行更仔细的检查，这些情形理论上讲是可以避免的。

“异常情况”（Exception）则经常表示另外一种“非同寻常”的错误，也就是说，这种错误通常是不可预测的。常见的异常情况有内存不足、找不到所需的文件，等等。

无论是错误还是异常情况，我们关注的都是出现这些问题之后该如何处理。对于编程语言来说，通常有两种方法。第一种是传统的非面向对象的语言，如Fortran、C和Pascal所采取的方法。在这些程序设计中，出错处理的任务几乎全在编程人员自身，编程人员必须考虑程序中可能出现的错误，并且自行决定如何处理这些错误。

这种处理方式存在两个缺点：首先是编程人员的负担过重，出错处理要求编程人员有广博的知识面，如果编写的是大系统，还要求编程人员对整个系统有较为全面的了解；其次是出错处理不规范，久而久之，每个编程人员均形成了自己的一套出错处理方式，这些出错处理对他自己来说是非常明显的，但对其他编程人员和用户却不然，这无疑影响程序的可读性和可应用性。

第二种方法则是面向对象编程语言中所采取的方式，即所谓catch-throw（捕捉－抛出）方式，运行时系统和应用程序抛出各种标准类型的错误，程序捕捉这些错误，并进行相应处理。这时，异常情况处理（Exception Handling）变得标准化，编程人员得以能够以一致的方式对错误进行处理。C++和Java都采用了这种方式。其中Java又增加了一项新内容：throw，catch和try。

Java的出错与异常情况（以下统称“异常情况”）处理采用“throw、catch和try”的方式。它们对应Java的3条语句：

1）throw

throw表示抛出一个异常情况。Java的异常情况一般由运行时系统抛出，但是编程人员也可以用throw语句自行定义异常情况。throw语句的作用是改变执行流程，使程序转而执行相应的异常情况处理句柄。如下面的例子：

```
Class My Exception extends Exception
 {
     ...
 }
 class Myclass
 {
     void oops()
     {
         if(/*no error occurred*/)
         {
```

```
            ...
        }
        else
        {                    //error occurred
            throw new MyException();
        }
    }
}
```

由这个例子可以看出，throw语句由throw关键字后接对象名构成。这个对象必须是类Exception的实例或子类。throw语句的格式如下：

throw subclass-or-instance-of-Exception

throw语句执行时，其后面的语句将不被执行，执行流程将直接寻找后面的catch语句，执行相应的异常处理程序。

2）try

try语句用于启动异常情况处理语句，其标准格式如下：

try statements

catch (subclass-or-instance-of-Exception e) statement;

每个try语句后面必须至少接一个catch语句。try和第一个catch语句之间括起的是需要捕捉异常情况的代码段。一旦抛出一个异常情况，系统将直接跳到catch语句，看看是否存在相匹配的异常情况类。即使try和第一个catch语句之间代码段包括方法调用，在方法调用内部被抛出的异常情况，也将使程序转而执行相应的异常情况处理程序。

3) catch

catch语句是和try语句一同出现的，一个try语句可接多个catch语句。因此，catch语句的调用格式和try语句相同，如下所示：

try statements;

catch (subclass-or-instance-of-Exception e) statements;

当一个异常情况被抛出时，系统将执行与一个参数相匹配的catch语句。当且仅当出现下面3种情况时，参数才被认为是与异常情况相匹配的：

（1）该参数是与被抛出的例外情形同属于一个类；

（2）该参数是被抛出的例外情形类的超类；

（3）该参数是一个接口，被抛出的异常情况实现了这个接口。

catch语句执行结束后，接下来执行的是try-catch之后的语句，而不是从出现异常情况的地方恢复执行。try-catch语句除用于出错处理外，事实上还可用作流程控制的一种手段。不过就一般情形而言，把try-catch语句用于流程控制的做法并不值得推荐，因为这毫无必要，只会损坏程序的可读性。

与异常处理情况相关的另一语句是finally语句。这条语句通常紧接于try-catch结构中的最后一条catch语句，结构如下所示：

try statement;

catch (Throwable-subclass e) statement;

finally statement:

finally语句表明：无论是否出现异常情况，也不管出现的是哪一种异常情况，均必须执行finally关键字后面的语句。finally语句其实并非必要，它可以用别的结构来替换。

异常情况处理是应用程序非常重要的一个方面。缺少了异常情况处理，应用程序将频繁退出运行，甚至给不出任何提示，这无论是对编程人员还是对最终用户，显然都是不可想象的。作为一门完全面向对象的编程语言，Java把异常处理封装到类中，实现了对异常情况处理的标准化处理，从而使编程人员更快地编写出更与用户友好的应用程序。

1.3.3 多线程机制

线程是现代操作系统所提出的一个新概念。线程，也被称作“轻量(lightweight)进程”，是比传统进程更小的一种可并发执行的执行单位。线程的概念提高了程序执行的并发度，从而可提高系统效率。C和C++采用的是单线程的体系结构，均未提供对线程的语言级支持。与此相反，Java却提供了完全意义的多线程支持。

Java的多线程支持体现在两个方面。首先，Java环境本身就是多线程的，它可以利用系统的空闲时间来执行诸如必要的垃圾清除和一般性的系统维护等操作；其次，Java还提供了对多线程的语言支持。利用其多线程编程接口，编程人员可以很方便地写出支持多线程的应用程序，提高程序的执行效率。

必须注意的是，从Java语言规范可以看出，Java的多线程支持在一定程度上可能会受其运行时支持平台的限制，并且依赖于其他一些与平台相关的特性。比方说，如果操作系统本身不支持多线程，Java多线程就可能只是“受限”的或不完全的多线程。

1.3.4 Java虚拟机

Java解决各机器不同机器码限制的方法是采用半编译、半解释的方式，定义出了Java自己的虚拟机。所以，Java虚拟机（Java Virtual Machine）是可以运行Java代码的虚拟操作平台，它的设计目标是提供一个基于抽象规格描述的计算机模型，为解释程序开发人员提供很好的灵活性，同时也保证Java代码可在符合该规范的任何系统上运行。这是Java平台独立性的基础。

Java虚拟机由5个部分组成：JVM指令系统、JVM寄存器、JVM栈结构、JVM碎片回收堆、JVM存储区。

下面分别介绍这5部分。

1) JVM指令系统

JVM指令系统同其他计算机的指令系统极其相似。Java指令也是由操作系统和操作数两部分组成。操作码为8位二进制数，操作数紧随在操作码的后面，其长度根据需要而不同。操作码用于指定一条指令操作的性质（在这里用汇编符号的形式进行说明），如iload表示从存储器中装入一个整数。当长度大于8位时，操作数被分为两个以上字节存放。JVM采用了“big endian”的编码方式来处理这种情况，即高位bits存放在低字节中。这同Motorola及其他的RISC CPU采用的编码方式是一致的，而与Intel采用的“little endian”的编码方式，即低位bits存放在低字节中的方法不同。

Java指令系统是以Java语言的实现为目的设计的，其中包含了用于调用方法和监视多线程系统的指令。Java的8位操作码的长度使得JVM最多有256种指令，目前已使用了160多种操作码。

2) JVM寄存器

所有CPU均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，如果虚拟机中的寄存器比实际CPU的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器，这反而会降低虚拟机的效率。针对这种情况，JVM只设置了4个最为常用的存储器。它们是：

pc	程序计数器
optop	操作数栈顶指针
frame	当前执行环境指针
vars	指向当前执行环境中的一个局部变量的指针

所有寄存器均为32位。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

3）JVM栈结构

作为基于栈结构的计算机，Java栈是JVM存储信息的主要方法。当JVM得到一个Java字节码应用程序后，便为该代码中一个类中的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架中包含以下三类信息：局部变量、执行环境、操作数栈。

局部变量用于存储一个类的方法中所用到的局部变量。vars寄存器指向该变量表中的第一个局部变量。

执行环境用于保存解释器对Java字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如，如果解释器要执行iadd（整数加法），首先要从frame寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。

操作数栈用于存储运算所需操作数及运算的结果。

4）JVM碎片回收堆

Java类的实例所需的存储空间是在堆上分配的。解释器具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。

在Java语言中，除了new语句外没有其他方法为一对象申请和释放内存。对内存进行释放或回收的工作是由Java运行系统承担的。这允许Java运行系统的设计者自己决定碎片回收的方法。在SUN公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

5）JVM存储区

JVM有两类存储区：常量缓冲区（constant pool area）和方法区（method area）。常量缓冲区用于存储类名称、方法和字段名称以及串常量。方法区则用于存储Java方法的字节

码。对于这两种存储区域具体实现方式在JVM规格描述中没有明确规定。这使得Java应用程序的存储器布局必须在运行过程中确定，依赖于具体平台的实现方式。

JVM是为Java字节码定义的一种独立于具体平台的规格描述，是Java平台独立性的基础。目前的JVM还存在一些限制和不足，有待于进一步完善，但无论如何，JVM的思想是成功的。

1.3.5 网络功能

Java语言目前日渐走红的一个主要原因，是充当了“网络上的世界语”这个举足轻重的角色，与这个角色相对应，Java语言本身便具有了独具特色的网络优势与网络功能。

由Java语言编写的应用程序（application）特别是小应用程序（applet）具有与操作平台无关的性能，而Java也得到了许多商用浏览器的支持，这首先就为它在网络上的应用奠定了基础。出于对安全性的考虑，Java的编译器和解释器要分别对Java的源码和字节码进行检查，保证了Java的程序设计人员没有暗藏破坏的企图，从而避免了毁灭系统资源、消耗系统资源、挖掘系统或个人秘密等事件的发生。另外，从小应用程序的角度考虑安全性，Java还提供了下述要求：不允许小应用程序读写用户系统的磁盘，但用户特别指定的目录除外；Java小应用程序不能够在用户系统上直接执行其他任何程序；除了小应用程序原来所处的位置外，通常不能通过网络与其他的WWW服务器联系；小应用程序无法取得任何有关系统设置的信息等。正因如此，Java语言才以其安全性及平台无关性等网络优势成为网络世界语的首选。

Java.net包提供了涉及网络运作的各种类，这些类又为简单的网络操作提供了跨平台的抽象性，利用Java.net包与输入输出流相结合，就能够做到从网络上读取文件、数据或向网络写文件数据时，像在本地磁盘上读写文件数据时一样的容易和方便。

在网络上，应用程序或小应用程序与系统之间的通信方法有三种，即：

（1）showDocument()，它能够使小应用程序applet去通知浏览器完成在Web上加载和连接另一页的任务。

（2）openStream()，它能够打开一个到某个URL上的连接，并从这个连接中获得数据。

（3）有关套接字的各个类：这些类可以打开到主机的标准套接字连接，并从这种连接中读取数据或向其中写入数据。

这3种方法都包含于Java.net包中，如果从网络功能的从低到高的级别来看，这个包所提供的网络功能又可以划分为如下3类：

1）Datagram(数据报)

Datagram是最低级的一种网络功能，这是因为其他网络数据传送方式都是假想在程序执行时，建立一条安全稳定的通信通道，而以数据报的方式传送数据时，只是把数据的目的地记录在数据分组中，之后便直接放到了网上，系统并不保证数据是否能安全送到，或者什么时候可以送到。

2）Socket(套接口)

所谓Socket，可以想成是两个不同的程序通过网络的通信管道，而这也是传统网络程序最常用的方式。一般在TCP/IP网络协议下的客户服务器软件，通常也使用socket来作为它们交流信息的方式。

3）URL

这是三大类功能中最高级的一种。通过URL的网络资源表达方式，就可以很容易确定网络上数据，以及把自己的数据传送到网络的另一端。

1.4 Java语言的基本表示法

本节介绍Java语言对标识符、注释、关键字、数据结构、运算符及其基本语句的具体规定。这些规定是Java语言基本表示法，相当于建筑一栋楼房所需要的砖头瓦块，它们是Java语言的基础。

1.4.1 标识符

标识符在程序设计语言中是常用的一种表示法，变量的定义、类名、域名等都需要由标识符来表示。标识符由程序员命名，由数字、字母、下划线或美元符组成。标识符必须以字母、下划线或美元符为第一个字母，在随后的字符串中也可以含有数字。在Java语言中，标识符可以任意长，在机器内部用Unicode字符集表示。

由于Java语言考虑到国际化的要求，所以采用16位的Unicode字符集作为字符的内部表示形式，每个字符用16个二进制位表示。在其他语言中，如C语言，字符只用8个二进制位来表示，也就是大多数计算机系统中采用ASCII码集。ASCII（American Standard Code for Information Interchange）以7个二进制位分别表示大小写字母、0～9的数字以及约12个左右的控制字符。随着计算机技术在西欧的普及，用户都需要表示自己国家语言的字母表。ASCII码集顺应需求，扩充为用8个二进制位表示，增加了128个字符。但是，渐渐地，用户需要一个更广泛的解决方案以支持亚洲语言中数千个表意文字。在这种情况下就选用了一种新的方案——Unicode字符集。Unicode字符集以16个二进制位表示，支持了65536个不同的字符，其中21000个用于汉语、日语、朝文等表意文字。Unicode前256个值含有ASCII码中的表示，使ASCII码成为Unicode一个子集，具有很好的兼容性。

Java语言的内部用Unicode表示字符，而字符码的外部表示，即打印时所表示的或在屏幕上呈现的状态，完全依赖于主机操作系统的服务。在日常使用的unix、windows等操作系统上，字符集都是基于8个二进制位，当Java得到这些系统的字符时，立即将其存储为16个二进制位的数据类型，并总是将它作为16个比特位处理。

1.4.2 注释

注释也是程序员常用的一种辅助手段，良好注释的习惯将有助于程序设计清晰度的提高，以及可读性的加强。

Java语言有3种注释方式：

（1）以“//”开始的注释结束于行尾；

（2）以“/*”开始的注释结束于下一个“*/”；

（3）以“/**”开始的注释结束于下一个“*/”。

程序员可能会对前面两种注释方式很熟悉，这也是C++语言中使用的注释方式。在Java

语言中，注释不能被嵌套。程序员希望注释一大段代码，可以在每一行前都加上“//”，也可以用“/*”与“*/”将其封闭起来。

注释（3）表示此种注释将被Java的自动文档生成器Javadoc拾取文字部分，即根据在Java源代码中的线索创建HTML文件。实际上，Javadoc就是分析说明在“/**…*/”中的特殊注释，将其规范化并提取成一系列在HTML中描述API的Web页面。Javadoc作用于*.Java文件，而不是*.class文件。如：

Javadoc filename.Java

执行结果生成一个filename.html的文件，程序员可以在Web页面上浏览它。在filename.html中将会显示出filename.Java类中所有的公有域以及类的继承链

1.4.3 关键字

关键字就是保留字，不能再被用做标识符。在Java语言中有近60个关键字，还包括6个保留日后使用的关键字。根据关键字的应用将其分为以下7类：

1）内构类型关键字

boolean true false char byte short int long float double void

2）表达式关键字

null this new super

3）语句关键字

（1）选择语句

if else switch case break default

（2）循环语句

for do while continue

（3）控制转移语句

return throw

（4）防卫语句（用于线程及例外中）

synchronized try catch finally

4）修饰符关键字（可用于说明范围、可见性、共享性等）

static abstract final private protected public transient volatile

5）用于类、方法及其相关目的的关键字

class instanceof throws native

6）用于扩展类构筑模块的关键字

extends interface implements package import

7）保留日后使用的关键字

cast const future generic goto inner operator outer rest var

在这些关键字中，有许多是读者所熟悉的，有一些是Java语言特有的。在以后的章节会更详细讲解其用途

1.4.4 基本数据类型

与C语言相比，Java语言多了两个基本数据类型：byte和boolean，同时，Java语言亦

严格地规定了数据类型的大小与标识性质。另外，在Java语言中，所有的变量都必须有初始值或缺省值。

Java语言的基本数据类型列于表1.1。

表1.1　基本数据类型

类型	取值	缺省值	大小	取值范围
boolean	true或false	false	1 bit	\u0000~\uffff
char	Unicode的子集中的字符	\u0000	16bit	
byte	有标号整数	0	8bit	-128~127
short	有标号整数	0	16bit	-32768~32767
int	有标号整数	0	32bit	-2147483648~2147483647
long	有标号整数	0	64bit	-9223372036854775808~9223372036854775807
float	有标号整数	0.0	32bit	+-3.40282347E+38~+-1.40239846E-45
double	浮点数	0.0	64bit	+-1.79769313486231570E+308~+-4.94065645841246544E-324

由于字符皆用16位二进制位表示，所以Java语言增加了一个数据类型byte，用8个二进制位来表示。另外，在Java语言中，布尔型（Boolean）数据不再与整数相关，而独立出来作为一种新的类型，并且也不能与整数有任何自动转换关系。但这并不阻碍布尔型与整型的转换。如：

```
boolean b=false;
int i=4;
b=(i!=0);//i=4 → b=true
i=(b)? 1:0;//b=false → i=1
```

其中，表达式1为整型转换为布尔型，表达式2为布尔型转换为整型。

另外，在Java语言中，char是唯一的无称号表示的数据类型。如果将char转换为byte或是short，很可能得到一个负数。

值得一提的是：Java语言中所有的基本数据类型在被说明之后，都会从内存中分配到相应大小的空间，用以存放初始值或缺省值，并且当方法引用此数据时，值就会传到方法中去。换一种说法也可以这样认为，基本数据类型是由值控制的。与基本数据类型相对的是引用数据类型，包括对象、数组（数组实际也是对象）。Java语言通过类库中定义的String类与StringBuffer类处理字符串，字符串也具有引用变量的特征。引用变量是由地址参数来控制的。当建立一个对象后，内存只分配一个变量空间用来存放此对象的地址；当方法调用它时，只是将此对象的引用即存有地址的变量传过去。只有当对象用new命令正式向内存申请空间或直接初始化时，才会得到用来存放值的空间。Java语言中除了基本数据类型外的数据类型就是类。Java中定义了大量用户可重用的类，如：字符串、数组、向量、哈希表、栈等。它们包含在java.lang包中。Java还定义了大量类为用户使用系统功能方便，这些内容将在稍后介绍。

1.4.5 运算符及其执行顺序

Java语言特别强调运算符执行顺序的概念。在任何一种算法语言中，影响一个表达式最终结果的因素有3个，并且它们以这样的顺序起作用：优先级、结合性、求值顺序。在Java语言中，对运算符的优先级、结合性与求值顺序作了明确的规定，从而根除了表达式上的二义性以及结果不统一的弊端。在Java语言中共有17种优先级，每个运算符分属于各个优先级。同时，每个运算符还有自己的界结合性，有的具有左结合性，即为自左至右的结合原则；有的具有右结合性，即为自右至左的结合原则。运算符在表达式中的执行顺序为：首先遵循优先级原则，优先级高的运算符先执行，之后，在同优先级的运算符之间遵循结合性原则，或自左至右，或自右至左，最后，Java语言严格地按自左至右的求值顺序完成表达式的求值。

正是由于规定了这一自左至右的执行顺序，才可以实现在一表达式上不止一次地修改同一个值。例如对于表达式“(i=4)*i++”，变量i首先被赋值为4，后做乘法，表达式的值为16，同时由于i+1使变量i值为5。这个表达式在C与C++语言中都没有定义。但在Java语言中，它是合法的。可见，求值顺序处理的是一个运算符周围的操作数的求值问题。

Java语言中共有44个运算符，其表示与作用列于表1.2中，Java的运算符可分为以下几类：

算术运算符（+ - * / %）
关系运算符（> < == >= <= != instance of）
逻辑运算符（! && ||）
位运算符（<< >> >>> ~ & ^ |）
赋值运算符（=及其扩展赋值运算符）
条件运算符（?:）
强制类型转换运算符（类型）
其他

表1.2 Java语言运算符总览

优先级	运算符	含义	要求操作数的个数	结合性
17	名称、标记new	单一标识为对象分配空间	N/a	自右至左
16	A[i]	下标运算符	后缀	自左至右
	M(…)	方法和引用	后缀	自左至右
	M(…)	域选择符	后缀	自左至右
	++ --	自增、自减	前缀	自左至右
15	++ --	自增、自减	后缀	自右至左
14	~	按位取反	单目运算	自右至左
	!	逻辑非	单目运算	自右至左
	+	正号运算符	单目运算	自右至左
	-	负号运算符	单目运算	自右至左

(续)

优先级	运算符	含义	要求操作数的个数	结合性
13	(类型名)	类型转换	单目运算	自右至左
12	*	乘法运算符	双目运算	自左至右
	/	除法运算符	双目运算	自左至右
	%	求余运算符	双目运算	自左至右
11	-	减法运算符	双目运算	自左至右
	+	加法运算符	双目运算	自左至右
10	<<	左移运算符	双目运算	自左至右
	>>	左移运算符	双目运算	自左至右
	>>>	右移比零填充运算符	双目运算	自左至右
9	Instance of <<= >>=	关系运算符	双目运算	自左至右
8	==	等于运算符	双目运算	自左至右
	!=	不等于运算符	双目运算	自左至右
7	&	按位与运算符	双目运算	自左至右
6	^	按位异或运算符	双目运算	自左至右
5	|	按位或运算符	双目运算	自左至右
4	&&	按位与运算符	双目运算	自左至右
3	||	逻辑非运算符	双目运算	自左至右
2	?:	条件运算符	三目运算	自右至左
1	= *= /= %= += -= <<= >>= >>>= &= ^= |=	赋值运算符	双目运算	自右至左

1.4.6 程序流程控制

Java语言中的所有语句都可以归之为以下4类：选择语句、循环语句、控制转移语句、防护语句。

前3类语句同C语言中对应的语句很相似，任何熟悉C语言的程序员对之都不会感到陌生。

防护语句是Java语言所特有的，主要用于例外与线程中。

1.4.6.1 选择语句

选择语句包括if语句和switch语句。

1）if 选择语句

if 选择语句的基本形式是：

if （表达式）语句体 [else 语句体]

其中，“else 语句体”可选。

在 if 语句中的表达式的值必须是一个布尔型，这一点与 C 中不同，Java 语言中不允许出现如“if (a=b)”这类的语句。正是由于布尔型独立于整型，才会使 if 选择语句的结构更加严整与清晰。如果表达式的值是非布尔型，系统将会报错。

另外，语句体可以用一条语句，或是用“{}”括起来的一组语句。语句体中可包括 Java 语言中的任何语句

从表面上看，if 选择语句与条件运算符的功能很相似。为什么还要提供两种方法呢？下面，对 if 选择语句和条件运算符来作一下比较和分析。

if 选择语句：if（表达式）值为 true 时执行的语句，else 值为 false 时执行的语句

条件运算符：（表达式）？值为 true 时执行的语句，值为 false 时执行的语句

首先，条件运算符仅仅可以处理表达式，它不能处理语句。当需要在两者之间做快速选择时，条件运算符极具优势，不仅运算速度快，而且程序清晰，一目了然。但当需要在若干值之间选择时，条件运算符也可以完成，但这时的整个式子显得臃肿，且难于让人很快读懂。所以这两者各有各的长处，各自适用于不同的条件。

2）switch 选择语句

switch 选择语句的基本形式为：

```
switch (表达式) {
case 常量 _1: {语句体 1}; break;
case 常量 _2: {语句体 2}; break;
case 常量 _3: {语句体 3}; break;
…
case 常量 _n: {语句体 n}; break;
      default: {语句体 n+1}; break;
}
```

switch 选择语句的常用目的就是为了从众多情况中选择所希望的一种去执行，故而，每一分支语句中都用 break 语句作为结束。如果忽略掉 break 语句，程序将执行下一分支，直到遇到一 break 语句或当前 switch 选择语句体结束，这往往不是程序员所希望的。

switch 选择语句中如果有，应只有一个 default 语句，作为其他情况都不匹配时的出口。

switch 选择语句的表达式的值应为一个常量。

1.4.6.2 循环语句

Java 语言中有 3 种形式的循环语句（这与 C 很一致），它们是 for 语句、while 语句，以及 do…while 语句。在 Java 语言中可以随时说明局部变量加以利用，并且仅在说明的程序块中有效。循环语句中常用此局部变量作为循环变量，当循环结束时，局部变量亦被内存空间释放掉，这种动态的分配内存的机制很方便，且节约内存（这与 C++ 中不同）。

1）for 循环语句

for 循环语句的一般格式为

for(循环初始值；循环测试条件；循环增长量)

{循环体}

for循环语句中的初始值，测试条件及增长量皆由表达式组成，三者皆为可选的，即可以在for循环语句之前设定好循环初始值，也可以在for循环语句体内对增长量加以设定。如果连测试条件都没有，那么将是无穷循环。

for（；；）{}

这三者之间用分号“；”隔开，故每个部分可以有几个表达式，每个表达式之间用逗号“，”隔开。如：

for (i=0, j=0; i+j<100; i++, j+=2)

如前所述，for 语句可临时说明局部变量且使用之，如：

for(int i=0; i<10; i++) {}

当 for 循环结束时，i 亦失效。

2）while 循环语句

while 循环语句的表达形式为：

while（表达式）语句体；

while循环语句先计算布尔表达式的值，如果为真，执行语句体；为假，转入执行下一条语句。也就是说，语句体可能不被执行或执行多次。同样，while语句的表达式应为布尔表达式。

3）do …while 循环语句

do …while 循环语句的表达形式为：

do 语句体 while （表达式）；

与while循环语句不同的是，do… while循环语句的语句体至少被执行一次。也就是说，do…while循环语句先执行一次语句体，再计算布尔表达式的值，如果为真，再去执行语句体；为假，执行下一段程序。

除了这 3 种形式的循环语句，还须介绍用于控制循环的两条语句：continue 和 break。

4）continue 语句

continue 语句有两种形式：

continue；

continue 标号；

continue语句的作用就是继续下一个循环。“continue；”被执行时，它使得程序进入下一次循环，即现在是第 n 次循环，增加循环变量，测试循环条件，然后，如果符合，进入第 n+1 次循环。

“continue 标号；”的作用是，如果希望结束内部循环，而继续外部循环时，同时不希望外部循环又从头开始。就事先在外部循环处加上标号，用“continue 标号；”语句来控制。

5）break 语句

break 语句亦有两种形式：

break ；

break 标号;

break 语句的作用与 continue 语句正好相反，它是使程序跳出循环语句，从而中断循环。break 语句仅用于 for，while，do …while 与 switch 语句中，当它被执行时，使程序流马上指向以上语句块的结束处，执行下面的语句。

如果是“break 标号;”语句，那么程序中必须已定义了一匹配的标号，且此标号标识在一个完整程序块处。这一完整程序块可以为任何语句，不仅局限于循环语句或switch语句。这与“continue 标号;”中的标号位置不同，在continue 中标号在外部循环开始处。还有一点很值得注意：标号是标识在完整程序块的开始处，而 break 语句使程序转入完整程序块的结束处。

1.4.6.3 控制转移语句

控制转移语句有以下两种表达形式：

return ;

return （表达式）;

控制转移语句用于返回调用方的时候。只有在调用方需要返回值时才使用第2种表达方式，对于一个返回值为“void”类型的方法，永远不会有返回值表达式的形式。另外，处理例外情况的“throw”语句也属于控制语句，它将引发一个例外。

需要提醒的是，Java语言中虽然将 goto作为关键字，但并没有goto语句，只不过为防用户将其定义为一个变量名，才将其作为关键字保留起来。Java 语言认为goto语句除了使程序流混乱之外，一无是处，并且Java语言现有的语句完全可以很轻松地完成以往goto语句的工作，故而将其摒弃掉。

1.4.6.4 防护语句

Java 防护语句用于出错与异常情况处理，详细内容请参见本书 1.3.2 节。

1.5 使用 Java 语言进行面向对象程序设计

1.5.1 Java 是如何工作的

Java 语言看起来很像广大程序员熟悉的 C/C++ 语言。Java 保留了 C/C++ 中使得它们成为功能强大的编程语言的许多特点，但是它也添加了许多增强措施，使得用Java编写程序更加有趣。许多程序员发现比起他们以前使用的语言来，使用 Java 使他们变得更加有创造性。这也是 Java 吸引人的地方之一。然而 Java 语言去掉了它的前任语言中许多严重易犯的错误。更为重要的是，Java 语言编写的代码可以运行在 Java 语言所支持的任何平台上。另外，Java 语言从一开始就是为网络编程而准备的，用它来存取网络与存取本机文件系统一样容易。至于说到Java小程序，实际上，它更容易些！这是因为 Java 语言的内置安全性检查机制所致，使得从网络上下载的 Java 小程序对本机系统不可能造成任何危害。

1.5.1.1 Java 虚拟机

Java 可移植性的秘密在于Java虚拟机（即 JVM）。JVM 位于Java程序和用户的计算机系统之间（如图 1.1）。

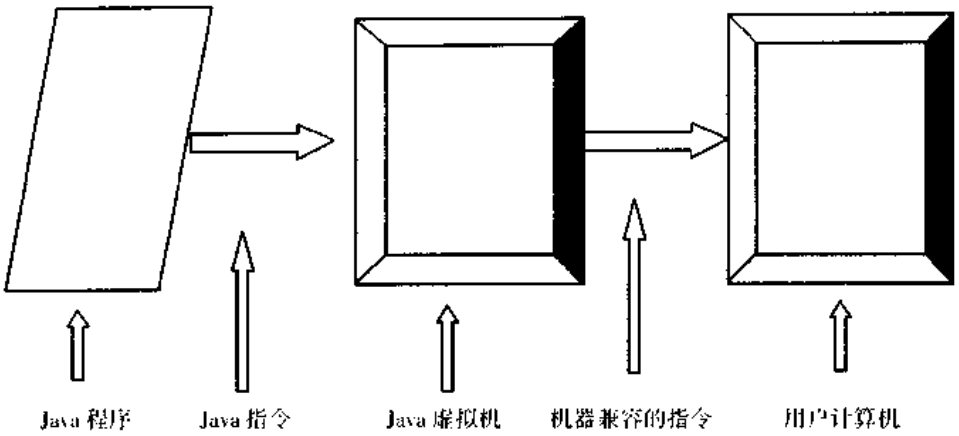

图 1.1　Java 虚拟机

虚拟机（VM）是计算机的一种抽象设计，现实中并不存在，但它可由程序仿真实现。顾名思义，Java 虚拟机，是一种“抽象计算机”，它知道 Java 和各种指令并把 Java 指令翻译成用户计算机所能理解的指令。

Java 语言被说成是又被解释又被编译，这是 JVM 结构的自然反映。用户写的 Java 程序首先被编译成虚拟指令（在 Java 虚拟机上运行的指令）——对应的字节代码。然后像用其他语言编写的其他任何程序一样，这种编译过的 Java 程序被当作可执行文件来分发。Java 与其他语言不同之处在于 Java 的可执行文件要运行于 JVM 之上。JVM 在任何计算机上都以同样的方式工作，所以 Java 程序不必为每种运行平台都单独地进行编译。当 JVM 运行 Java 程序时，它为所运行于上的平台解释（翻译每条指令并立即执行该指令）每个字节码命令。JVM 的最大优点在于用户的 Java 程序可以不加修改地运行于 JVM 存在的任何平台上。所有和移植性有关的问题全部在 JVM 中得到了解决，用户不必再为此头疼不已。

随着 Internet 和 World Wide Web 的出现，可移植性成为了必需。同时，当前高速硬件使 JVM 实用化。另外，虚拟机仿真器设计中的主要进步极大提高了仿真性能。实时（JIT）编译器通过在程序解释时编译它们并进行初步的优化，取得了和传统编译过的文件接近的性能。

另外，新的可执行智能的自适应优化的虚拟机正在 Sun 和其他公司的开发中。这些仿真指出了在什么地方程序消耗了大部分的运行时间，然后，在那些地方做广泛的优化。运行于这些虚拟机的 Java 程序有希望比某些编译过的程序更快些，然而仍保持平台无关性。

1.5.1.2　编译 Java 程序

如前所述，Java 程序被编译成字节代码，在 JVM 上解释执行。图 1.2 说明当 Java 程序编译和执行时发生了什么。

JVM 实际执行的是 .class 文件。当 JVM 运行在用户机器上时，它把 .class 文件中包含的虚拟机指令翻译成用户计算机可以理解的指令。

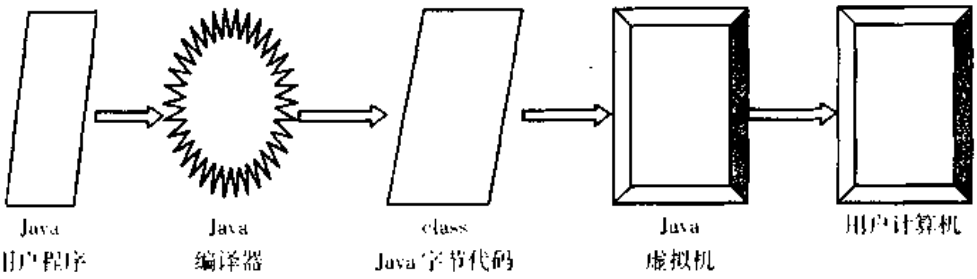

图 1.2 Java 程序的编译运行过程

Java (Javac.exe)和Java解释器 (Java.exe)都是与操作系统有关的程序。用户不能在 UNIX 平台上运行 Windows 执行文件，也不能在 UNIX 平台上运行 Windows 版本的 Java 解释器 (Java.exe)。然而，.class 文件中包含的字节码却是完全不同的两回事。因为它们是平台无关的，所以可以放在任何有 JVM 的平台上，它们可以在网络上下载并运行于用户的机器上，而与开发这些程序所赖依的平台无关，这就是 Java 语言的本质美。

1.5.1.3 小程序与应用程序

应用程序是由用户系统就地装入的正常程序。一个典型的应用程序可以获得用户系统上所有可得到的功能，包括向本机硬盘读出和写入数据，获得其他的系统资源如打印机。

另一方面，Java 小程序是可从网络上下载并在用户机器上（在用户的 Web 浏览器中）运行的短小而有限制的程序。一个正常的小程序可以与下载所在处的服务器对话，但不能影响用户系统。然而，一个可信赖的小程序，与就地装入的应用程序一样，被赋予了可同样访问用户系统的能力。

当 Java 开始发布时，它特别地限制在创建小程序上。然而，今天，Java 也越来越多地被用来开发大型应用程序。

1.5.2 面向对象编程

OOP就是使用对象进行编程的过程，所谓对象就是协调数据存储以及作用于数据之上操作的独立实体。对象把数据保存在属性中，对象中也包括作用于属性之上的操作，称之为方法（方法亦称作函数、过程、子例程或行为）。比如，设想把一辆汽车作为一个对象。一些信息或数据描述了许多汽车中的这一辆的品牌模型、颜色、最高速度，这些都是汽车对象的属性。汽车也可以完成事情：启动、加速、行驶、绕行、减速和停车。这些是汽车对象的操作，用 Java 语言的术语说，是它的方法。

用户可以通过定义一个对象集合以及它们之间的相互作用来创建一个面向对象程序。对象们协同工作来定义一个完成用户需要的程序。

1.5.2.1 类和对象

那么当用户创建一个面向对象程序时，如何建立对象呢？可以通过定义类，然后使用类来创建用户需要的对象。当用户编写自己的Java程序时，所涉及到的主要工作就是编写类定义。当程序运行时，类定义用来创建新对象。由类定义创建对象的过程称为实例化

(instantiation)。每个对象是类的一个新实例(instance)。图 1.3 显示了类和对象的不同之处。Car类是对什么是汽车的一个定义，而 Ferrari、Volvo 和 Minivan 是对象，是汽车类的实例。

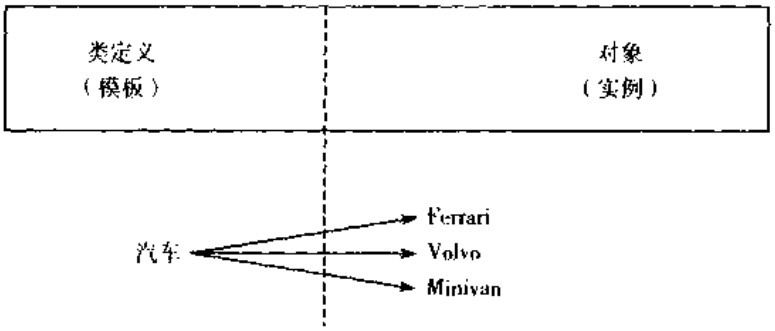

图 1.3　类和对象

1）属性

类定义中的属性定义了一个对象区别于其他对象的特征值。比如，在Car类的定义中包括汽车的品牌、颜色、最高速度这些属性，如图 1.4 所示。每个对象的这些属性都有自己的值。所有的由类定义建立的对象都共享类的方法。但是它们都拥有在类方法中定义的所有变量的副本（copy）。

2）方法

对象的操作由Java中的方法来指定。要使一个对象做某件事情，就要调用它的相应方法。在用户程序中，这由一行给出了方法名及参数列表的代码来完成。假如用户想要改变汽车的颜色，在程序中，可以编写如下代码：

myCar.setColor（yellow）

这行代码包括了对象名 myCar；方法名 setColor；参数 yellow（包含于（）号中）。可

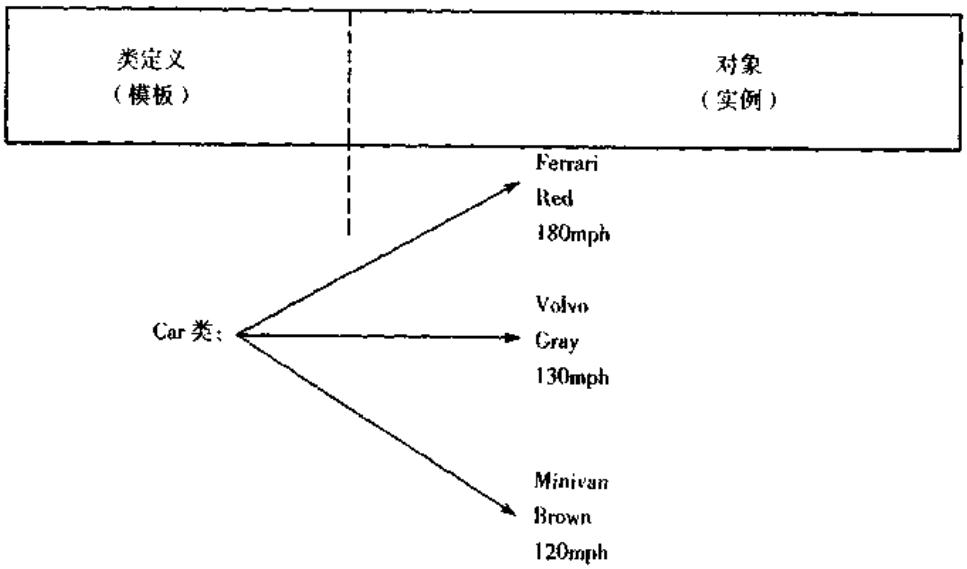

图 1.4　拥有属性的类和对象

以看到，一个方法的参数是传送给方法的数值，这些数值在方法执行中被用到。在这个例子中，setColor方法把数值yellow赋给汽车对象的color属性，然后改变汽车的颜色。这个过程也被称为向对象发送一条消息。在本例中，用户向汽车对象发送了一条消息，要求改变颜色属性，并指定了新的颜色值。

为了分清到底是一个对象的属性还是方法，在本书中方法后均带（）号，比如stop（）号。当参数传递给方法时，它们在（）中给出。方法也可以返回一个数据值。下面这个方法：

```
myCar.getColor ( );
```

将返回汽车的颜色值，在本例中为黄色。

1.5.2.2 封装

类定义封装构成类的属性和方法。这是因为在类定义中属性和方法能够不为程序的其他部分见到，它们“隐藏”了起来。它们只能由所定义的类使用。从表面上看，这是一个非常简单的概念。但它在被公认为是一个好主意之前，已经花去了30多年的程序设计实践和思考。

在封装出现以前，程序的任何部分都能存取其他的任何部分，这使得改动变得十分困难，因为程序员永远也不知道这种改动会产生什么样的影响。当发现一个变量拥有一个糟糕的值时，用户永远也不知道如何变成这样的情况。封装简化了事物因为改动的影响，往往局限于程序中一个小的区域内，极大地缩小了潜在问题的影响范围。所以封装的确是个好主意。它只是看起来简单，但是到现在我们才懂得它。

封装的根本目的是保证对象的属性只能通过对象的方法进行存取。这种实现需要额外的编码，但它保证了任何使用该对象的编码都独立于该对象执行的实现细节。这使得用户可以按个人意愿改变对象的执行过程。只要对象的程序设计接口（API）不变，也就是对象方法的结构不变，那么任何使用该对象的代码都能像以前一样正常地工作。

1.5.2.3 继承

图1.5显示了面向对象编程的另一重要方面——继承性。注意Vehicle定义了属性make、color和topSpeed。当定义子类Car时，它自动继承基类Vehicle的属性和方法。类Car称为一个子类，派生类或继承类。类Vehicle称为Car的基类、父类或超类（superclass）。

继承节省了在定义新类中的大量工作，因为编程者可以方便地重用代码。比如，当创建派生类时，Car.color和Car.topSpeed属性被自动地定义，引用方法Car.start（）时会自动调用在类Vehicle中定义的方法start（）。

但一个子类不必非得使用继承下来的属性和方法。一个子类可以选择重载（override）已存在的属性和方法，或添加新的属性和方法。有代表性的是重载方法，而非属性。用户也可添加其他的属性和方法来满足具体的需要——比如说摩托的撑脚架。

只有当用户想向自己的新类定义中添加新的操作，或者把已存在类的缺省行为融合进自己的新类中时，才需要继承一个已存在的类定义。在Java中，这称为派生一个类。要添加新的方法和属性，只需定义它们即可。要重载一个已存在的方法，就要在子类中（Car）定义一个新方法，该方法与基类中（Vehicle）被重载的方法有相同的名字和参数列表。

图1.6显示了在Vehicle的派生类中重载stop( )方法的过程。

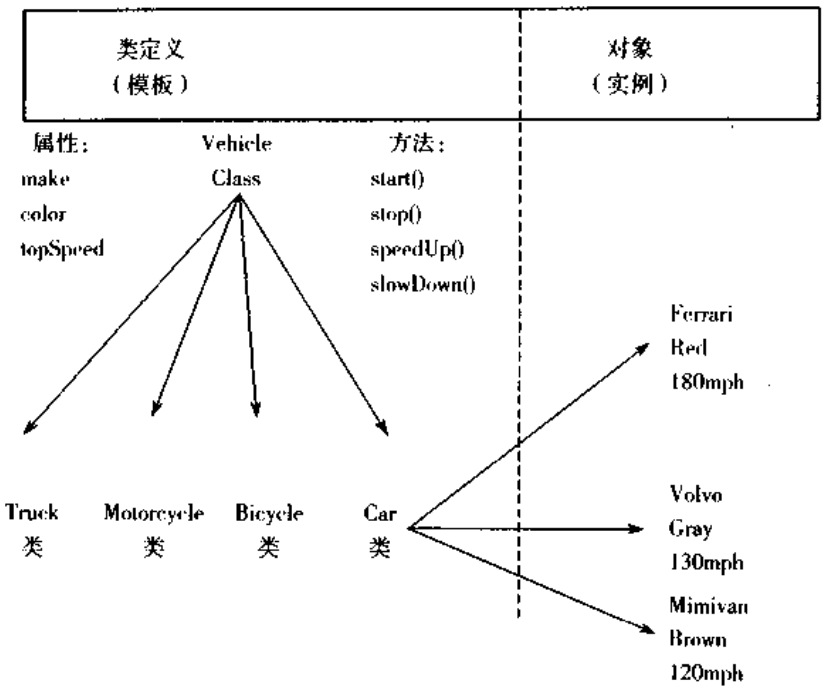

图 1.5　vehicle 类是 Truck、Motorcycle、Bicycle 和 Car 的基类

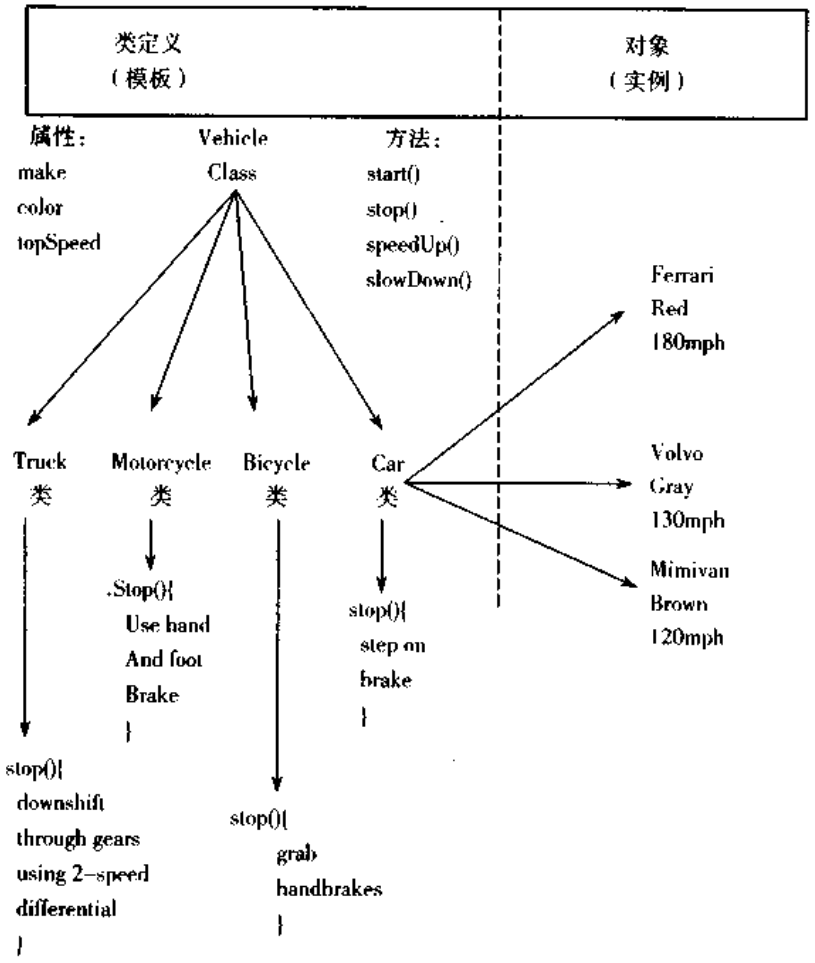

图 1.6　Vehicle 的派生类以不同方式重载了它的 stop()方法

如果子类中没有重载stop()方法，那么基类Vehicle中的缺省stop()方法就被调用。该方法可能关掉发动机。

1.5.2.4 多态性

同样的方法名因为调用的方式不同而执行完全不同的例程的能力，带来了面向对象编程中另一个强大的方面——多态性（polymorphism）。“Poly”的意思是“许多”，“morph”意为“形状”。所有“polymorphism”从词义上说就是许多形态的意义。它实际的含义就是不同的对象有相同的一般的轮廓或形态，但具体执行的过程却大相径庭。

多态性使用户可以写更加通用的过程。用户可以编写一个过程来控制Vehicle对象，说“当看到一个停车标志时，执行stop（）方法”。用户可以让子类去关心如何完成stop（）方法，而继续在更高的抽象级别上编写自己的通用过程。比如说，即使Car对象的stop（）方法与Truck对象的stop()方法完全不一样（Truck在18个档之间来回切换），用户也可以编写Vehicle.stop（），并知道哪一个合适的方法将被调用。

多态性也使得用户在以后不费多大力气就可以派生程序。假设用户在为小汽车和卡车构建应用程序。用户知道还存在摩托车和自行车，但在当时并不太在意它们。用户可以写一个处理Vehicle类的程序，把Truck和Car子类定义为Vehicle的特例。以后，如果需要，不需多大努力就可派生Vehicle类来定义Motorcycle和Bicycle类，并相信为Vehicle类编写的所有代码都可以为这些新类工作。

1.5.2.5 方法重载

方法重载（method overloading）提供了类似于多态性的好处。方法重载意味着两个方法有相同的名字但参数不同。方法的名字与它参数的个数及类型决定了该方法的特征（signature）。一个类可以拥有多个同名的方法，只要每个方法都有不同的特征即可。

与多态性一样，方法重载赋予了用户某种程度的概括能力。使用不同的方法特征使用户可以指定同名但细节不同的多种操作。比如，Car类拥有一个标准的stop（）方法，即脚踏刹车闸。它也可以有一定义为“双脚猛踩车闸”的stop（emergency）方法，和定义为“轻微地踩刹车闸但一有滑动就松开刹车闸”的stop（ice）方法。使用相同的名字和不同的参数来调用不同的方法，使用户不必为同一类操作的不同变体而绞尽脑汁取新的名字。这也使类的使用者可以更容易地记住方法的名称。

1.5.2.6 接口

Java对接口的定义提供了一种与封装、继承、多态性和方法重载同等重要的能力。然而对这种能力至今没有一个标准的术语称呼它。用户可以把它称作等态性（isomorphism）或“相同形态（same shape）”，这是一种把来自完全不相关类的若干对象统一处理的能力。Java接口确实简单，但又异常强大。Java接口可用来定义：

1）对象所执行的一种方法；

2）一组有用的常量。

所有在Java接口中定义的变量都为自动常量。

从类定义的起点出发，接口指定了类提供哪些方法和属性，但用户也可以定义一个使用接口的变量，正如用类名定义一个实例一样。当这样做时，变量可以拥有执行所指定接口的任何类的实例，比如，如果前面提到的stop（）方法是Stoppable接口的一部分，那么用户可以定义一个Stoppable类型的变量，它可以含有类Car、Truck或Sailboat的一

个实例。

从按此途使用一个对象的起点出发，接口指定了使用该对象的代码可以应用哪些方法和数据。为了帮助用户更好地理解这些，让我们看一个对象是如何被使用的。

1）使用对象

假设用户在编写一个赛车程序。赛车程序含有几个对象，包括一个Car对象、一个Course对象及一个Weather对象。程序含有可以从这些对象中提取的数据，调用它们的方法来管理比赛。

程序除了有代表赛道和天气的对象外，还有一个称为raceCar的Car类对象。程序赋予raceCar对象诸如speedUp（）、slowDown（）、pass（）和draft（）之类的命令。图1.7显示了这些类的包含层次。

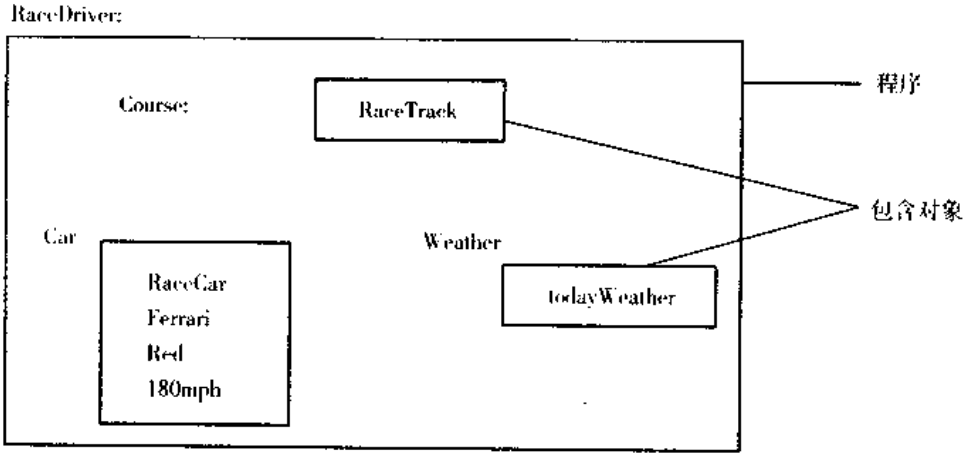

图1.7 使用raceCar对象的赛车程序

下面是用伪代码编写的赛车程序的粗线条轮廓。它显示在对象名和属性或方法间加一点号（·）的传统写法，如raceCar.speedUp（）。这传统写法称为点表示方法（dot notation）。

```
if on straightaway:          raceCar.speedUP ( )
if curve ahead:              raceCar.slowDown()
if car ahead going slower:   raceCar.pass()
if car ahead at same speed:  raceCar.draft()
```

用户已知道一种方法来概括这个程序。不是使用Car类的一个对象，而是使用Vehicle类的派生对象。只要所有的Vehicle类的派生都提供了正确的方法，赛车程序就可以用于比赛的摩托车、自行车，甚至还可以用于比赛的卡车。图1.8显示了经概括后重写的程序段。

下面是重写的伪代码，现在它应用于比赛摩托车、自行车以及小汽车：

```
if on straightaway:          raceVehicle.speedUp ( )
if curve ahead:              raceVehicle.slowDown ( )
if vehicle ahead going slower:   raceVehicle.pass ( )
if vehicle ahead at same speed:  raceVehicle.draft ( )
```

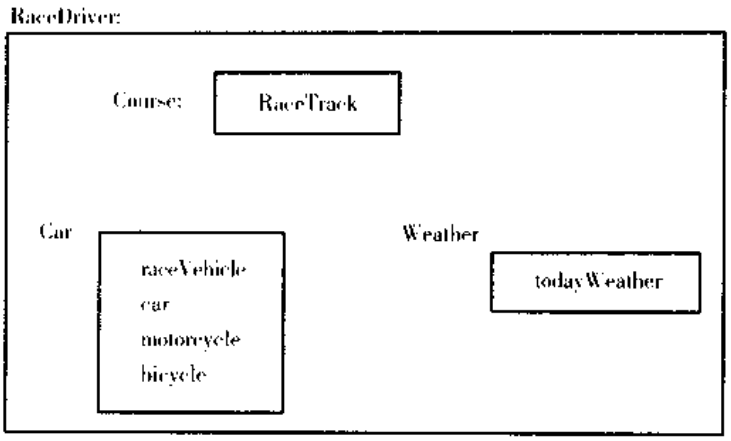

图 1.8　使用 raceVehical 对象的 RaceDriver 程序

但用户要注意上面的重要警告“只要所有的Vehicle类的派生类都提供了正确的方法”，并没有措施保证它们都能这样做！由于卡车通常不用来比赛，因此在Truck类中可能就没有定义draft（）方法　由于在Vehicle类中也没有定义draft（）方法，结果Truck对象将根本没有draft（）方法，这个程序将运行失败。

2）补救接口

解决上面问题的一种办法是保证参赛的对象执行恰当的方法。在Java中，可通过指定一个抽象接口来达到。接口并不实现一套方法集，它只是指定了这些方法的存在（见图1.9）

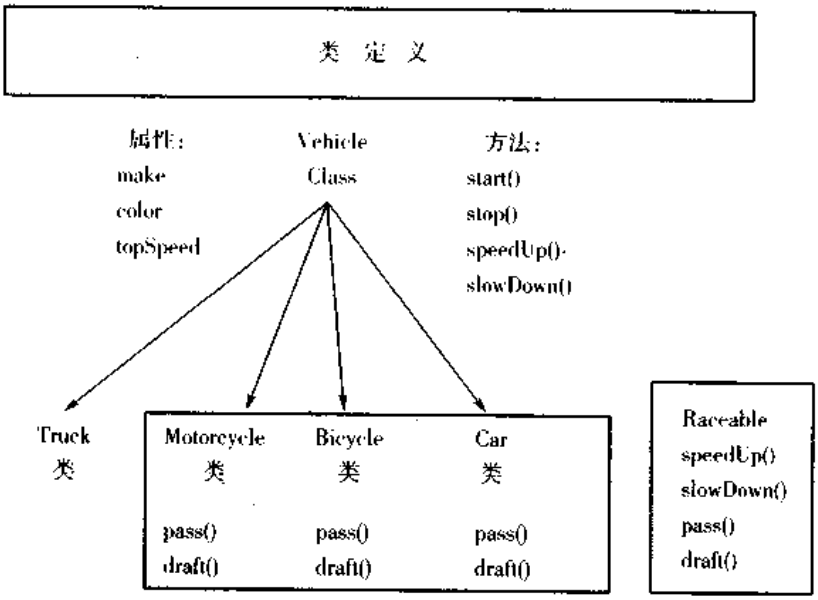

图 1.9　实现 Raceable 接口的类

拥有 Raceable 接口的 Motorcycle、Bicycle 和 Car 必须实现该接口指定的所有方法。它们已经从 Vehicle 类中继承了 speedUp（）和 slowDown（）方法，所以它们只需执行 pass（）和 draft（）方法来成为 Raceable 对象。图 1.10 显示了与早先所见相同的程序，仅修改为运

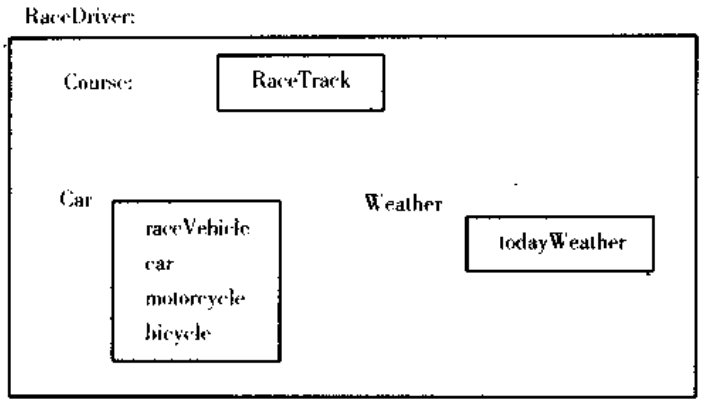

图 1.10　使用 raceable 对象的 RaceDriver 程序(实现 Raceable 接口的 Raceable 对象)

行在 Raceable 对象上。

对 RaceDriver 程序的唯一改变就是将 raceVehicle 对象定义为 raceable 类型而不是 Vehicle 类型。其他东西都没有，甚至伪代码都是一样的：

if on straightaway：raceVehicle.speedUp（）

if curve ahead：raceVehicle.slowDown（）

if vehicle ahead going slower：raceVehicle.pass（）

if vehicle ahead at same speed：raceVehicle.draft（）

但是一个改变导致了很大的区别，因为现在的赛车程序适用于许多不同的Raceable对象。

接口的力量源自于它是作为数据类型使用的这一事实。任何实现 Raceable接口的类都可以用于赛车程序。这意味着同样的程序可以用于 RaceHorse、Skater、Sailboat 或 Runner 这些类型的比赛对象，只要这些类能够提供 speedUp（）、slowDown（）、pass（）和 draft（）方法即可。这种区别如图 1.11 所示。

用户也可以看到Java接口提供了整个一种新的水平上的概括能力。程序有了多得多的自由。它不再关注比赛对象属于什么类，而仅仅集中于该比赛对象要做些什么！使用这种方法允许来自广泛的不相关类的若干对象在一起工作。

在 Java 中，一个只能派生于其他的一个类，但它可以定义自己需要的任意多的接口。

3）多态性接口小结

用户见到许多Java代码都利用了接口的优点，当程序员们学会利用接口提供的强大的概括能力后必定在将来会更多地使用。

接口把多态性提升到一个全新的水平上。多态要依靠于接口。多态性是说，派生于同一个基类的两个类可以拥有以不同方式定义的同名方法。用户也可以编写使用一个基类对象的过程，而不必担心该过程实际包含的是哪一个派生类对象。在该对象中调用基类的方法仍然适用，不过派生类重载了这一方法，提供了一个不同的定义。

然而，接口是说两个完全无关的类可以拥有不同方式定义的同一组方法。用户可以编写使用一个接口对象的过程，而不必担心此过程包含的是哪一个对象。调用该类的对象上的接口方法仍然适用，尽管每个类只提供了该方法的一个不同的实现。

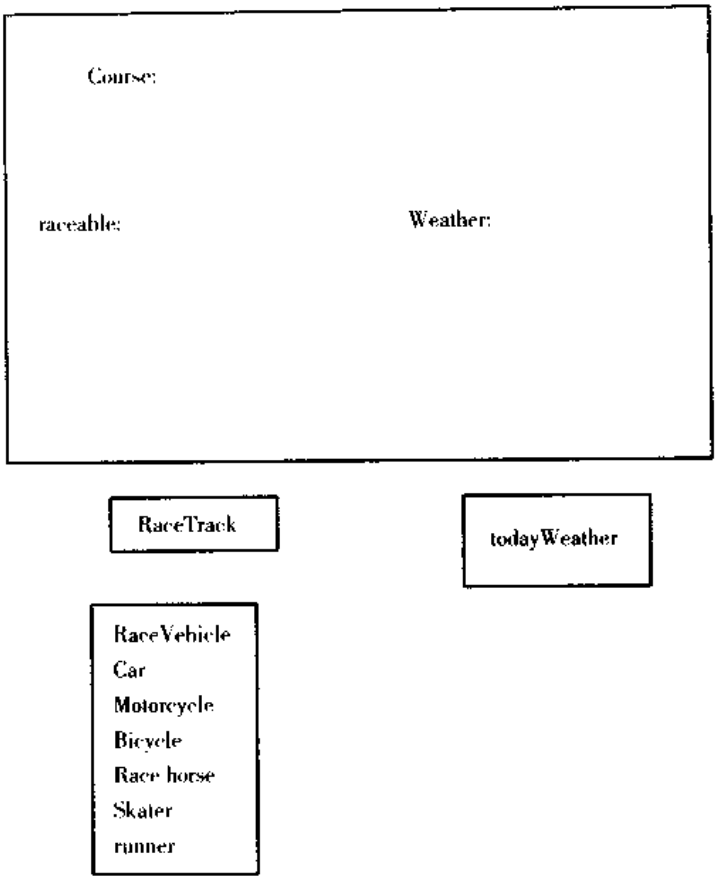

图 1.11　使用了其他 raceable 对象的赛车程序

接口不是多重继承，多重继承就是从不只一个类上继承属性和方法的能力。理论上，它允许用户定义一个多父类的对象。然而，实际上多重继承要取得成功十分困难。要求编译器对多重继承进行正确管理是十分困难的，它也几乎不能提高程序的设计效率。所以即使 C++ 语言允许多重继承，在 Java 中也不允许这样做。当用户需要同多种继承相似的效果时，Java 程序员必须使用其他的机制。

在 Java 中，接口往往是作为提供多重继承的方法被谈及的。然而它们不是，主要原因就是它们仅是对方法特征的抽象定义。它说明了接口中所提供方法的类型，但并不亲自实现这些方法。而另一方面，通过多重继承，类定义获得来自基类的方法特征和方法的实现过程。

继承的整个想法就是通过重用已有的代码来节省编程工作。接口并非如此。为了在 Java 中获得与多重继承同样的效果，用户需要建立自己的新类以使新类中包含自己要用的代码。然后通过它们的方法来代表用户希望这些对象所具有的行为。不管怎样，这是一个更好的设计途径，它不带有多态继承中性能和维护方面易犯的错误。用户经常把接口作为过程的一部分，以保证代表者和被代表者的方法特征相匹配，但这与拥有多个实例不是一回事。

在以上两种情况下，程序都获得了在一种抽象水平上处理对象集的能力，而让每个对象自己去关注具体的细节。基类和接口都确保了能够定义程序员所能依赖的一组特定方法。

接口是通过一种更普遍的途径来实现它的，所以在某种意义上它们更加强大，尽管在接口用户不能继承已存在的方法。

1.5.3 面向对象编程和基于组件的编程

遗憾的是，面向对象引起的革命并没有使程序设计更加简化，使用面向对象方法的程序更加灵活，可维护性更好，但它们并不因此而更加容易设计和编写。实际上，在许多方面，面向对象的构造更加困难，就好像代码的编写比搭积木更难。

然而，面向对象的构造被证实为是设计软件组件的极好基础。所谓组件，就是可以很容易地被定制并使用它来向程序中添加功能的对象。比如，在一个GUI程序中的按钮和窗口都是组件，正是大多数的可视化组件构成了这样的程序。用户添加到HelloWorld程序中的标签也是一个组件，通过告诉它，显示的文本和使用的字体可以对它进行定制。也有一些非可视化组件。例如，用户使用的用来连接数据库的特定组件，通过告诉它要连接的数据库的名字以及数据库的地址来定制它们。

正在出现的软件组件工业大有希望，最终使得通过把组件接插到一起来创建应用程序的方法成为可能，在这方面已取得了一定的成绩（Java Beans标准的开发是在这方面迈出的坚实一步）。除了GUI组件和数据库接口组件外，在JBuilder中附带的Java通用库（JGL）中有用于分类和过滤数据结构的“策略型”组件。今后的几年中，很可能会取得更多的成就。最终，用户可以通过组合预制的组件，使用最少量的特殊代码把它们连在一起构成应用程序。

Borland公司的JBuilder使基于组件的编程尽可能简化。它允许用户使用自己创建的新组件或从他人那里获得的组件来扩充开发环境，于是用户可以以最小的代价来把它们融入自己的设计中。像JBuilder这样的可扩展集成开发环境就有助于创建组件开发市场。

然而，纯基于组件的程序设计至今也没有可能。要使任意但是十分繁琐的程序工作需要太多的编码。到今天，面向对象的开发技术是经过实践验证的最好的软件构造技术，Java也顺理成章地成为实现这种方法的最好语言。

1.5.4 定义Java类

本节学习如何创建和使用对象来建立一个比Hello World程序稍微复杂一点的程序。

在Jbuilder环境下选择File~Open/Create菜单项。双击hello文件，然后双击工程文件hello.jpr。工程现在打开了，用户可以准备工作了。

1.5.4.1 定义一个新类

开始时在Structure面板中单击Helloworld.Java文件。下面的代码出现在Content面板中：

```
package hello;
import com.sun.Java.swing.UIManager;
public class helloWorld {
  boolean packFrame=false;
  //Construct the application
    public HelloWorld(){
```

```
        Frame1 frame=new Frame1();
        //Validate frames that have preset sizes
        //pack frames that have useful preferred size info.
        //e.g.from their layout
      if(packframe)
      frame.pack();
      else
      frame.validate();
      frame.setVisible(true);
      }
      //main method
      public static void main(string[] args){
      try {
        UIManager.setLookAndFeel(new
            com.sun.Java.swing.plaf.windows.windowsLookAndFeel());
        //UIManager.setLookAndFeel(new
            com.sun.Java.swing.plaf.motif.MotifLookAndeel());
        //UIManager.SetlookAndFeel(new
            com.sun.Java.swing.plaf.metal.MetalLookAndFeel());
}
catch (Exception e){
}
      new HelloWorld();
   }
}
```

下面这行代码：

```
class HelloWorld{
```

使用了关键字 class 来定义 HelloWorld 类——程序的主要类。在类定义中左大括号“{”与右大括号“}”之间的内容是类定义的一部分。

按照惯例，类名首字母大写，所以 HelloWorld 类中 H 是大写的。

1.5.4.2 派生现存的类

现在单击 Structure 面板中的 Frame1.Java，然后浏览出现在 Content 面板上的代码：

```
package hello;
import Java.awt.*;
import Java.awt.event.*;
import borland.jbcl.control.*;
import borland.jbcl.layout.*;

public class Frame1  extends DecoratedFrame{
```

```
BorderLayout borderLayout1=new BorderLayout();
XYLayout xyLaout2=new XYLayout();
BevelPanel bevelPanel1=new BevelPanel();
Label  label1=new Label();
//Construct the frame
public Frame1(){
  try{
    jbInit();
  }
  catch (Exception e){
    e.printStackTrace();
  }
}
//Component initialization
  public void jbInit () throws Exception{
  this.setLayout(borderLayout1);
  this.setSize(new Dimension(400,300));
  this.setTitle("Hello World");
  label1.setFont(new Font("Dialog",1,32));
  lab11.setAlignment(1);
  label1..setText("Hello  world");
  bevelPanel1.setLayout(xYLayout2);
  this.add(bevelPanel1,BorderLayout.CENTER);
  bevelPanel1.add(label1,new XYConstraints(86,95,-1,-1));
 }
}
```

这个文件中以下代码定义了 Frame1 类：

```
class Frame1 extends DecoratedFrame{
```

但这个类定义建立在一个已有类DecoratedFrame 的基础上，而不是从头创建的一个新类。关键字 Extends 说明 DecoratedFrame 是基类，而 Frame1 是派生类。

Extends 子句是可选的。只有从不同于 Java 根类（Object）在派生时才需要该子句。所以下两个类定义是等价的：

```
class HelloWorld {
  ...
}
class HelloWorld excends Object {
  ...
}
```

同样，用来定义类的属性和方法含在大括号内（{}）。

1.5.4.3 把类定义存储在源文件中

HelloWorld 类包含在源文件 HelloWorld.Java 中。一个 Java 文件可以包含多个类定义。当该源文件编译时，会创建多个 .class 文件，每一文件对应一个类。

缺省情况下，当源文件存于 C：\JBuilder2\myprojects 中时，JBuilder 把 .class 文件放在 C：\JBuilder2\myclasses 目录下。把 class 保存在一个单独目录下使得管理用户的源目录更加容易，也容易分发拷贝 .class 文件。

HelloWorld 程序的文件名与它的类名相同，这并不是偶然的。这是由类定义中 public 关键字带来的后果之一：

```
public class HelloWorld{
```

Java 中要求任何以 public 关键字定义的类的文件名与类名相同。这意味着一个源文件中只能有一个 public 类。这也适用于 Frame1.Java，它包含了一个名为 Frame1 的 public 类。public 类可以被程序中的其他任何部分使用。public 类总是以与文件名相同的名字定义，使得用户可很容易地找到程序中使用的一个已存在类的代码。

1.5.4.4 把类组合为包

一起工作的若干个类的集合存储在一个普通包（package）中。包就是具有相同名的一个目录中所含有的类的集合。HelloWorld.Java 和 Frame1.Java 都以下面这行代码开始：

```
package hello;
```

package 关键字的意思是当前包被命名为 hello，因此从这个文件中创建的类都存于名为 hello 的目录下。按照惯例，包名字都是小写字母，所以 hello 的 h 为小写。

1.5.4.5 向应用程序中导入包

HelloWorld.Java 中的下面这行代码引用了包含在 AWT(Java Abstract Windowing Toolkit) 中已存在的类集合：

```
import Java.awt.*;
```

实际上，单词“import”是误称。实际上没有任何东西导入到程序中。import 关键字只是简单地像 MS—DOS 中的环境变量 PATH 一样，告诉 Java 编译器去哪儿寻找在程序中找不到的类。

AWT 被称为抽象窗口化工具箱，是因为它是许多不同窗口化操作系统——Microsoft Window、MacOS、UNIX OpenLook 等等的抽象。它定义了一个用户可以用来编写运行于许多不同系统上的 GUI 程序的“抽象层”。

Frame1.Java 中还包含下面这行代码：

```
import Java.awt.*;
```

因为这样做了，所以用户的 Frame1 类就可以使用 Label 类来定义 Label1 变量：

```
Label label1=new laebel ( );
```

名为 lable11 的变量实际上是计算机中一个存储单元，它包含了指向一个用 Label 类定义建立的对象的引用（或指针）。用户脑中应保特这种清醒的认识：一个如 Label1 的对象变量实际上包含的是指向对象的引用，而不是对象本身，那么 Java 中许多微妙之处也就清晰可懂了。

用户也可以使用完整的路径名来指定 Label 类，正如在 MS—DOS 中指定完整的路径名。如果不是有了 import 语句，这行代码必须写成：

Java.awt.Label label1=new Java.awt.Label();

1）通配符

import语句中的星号（*）是一个通配符，代表Java.awt包中的所有类：

import Java.awt.*;

用户可能还会指定单独一个类，方法如下：

import Java.awt.Label.;

这也将允许用户不必完全说明引用路径就可使用Label类。不过，理解当导入Java.awt.*时，只有那些程序中实际用到的类成为程序的一部分，它们只在程序运行时才被使用！用户的程序不会因为使用了“*”引用而变得臃肿起来。

以前，程序员习惯使用整个例程库作为可执行程序的一部分。当他们这样做时，库中的每一个例程都成了程序的一部分，不管用到还是没用到。同样在C/C++中，include语句引用了一个头文件成为程序的一部分，不管实际上使用了该头文件的多少。

随着动态链接库（DLL）的到来，大的例程库存放在许多程序可以得到的中心地带。只有当这些例程被实际用到时才申请内存，因为多个程序可以读取相同的DLL，磁盘空间也节省了。Java的import语句，不管它的名字如何，工作的方法与DLL是一样的。没有例程的拷贝发生。类在使用时只是简单地被引用。

2）点和目录

Java.awt.*中的点分隔指定路径中的子目录。当用户看到Java.awt.*时，可以肯定存在一个名为Java的目录，在其下有一个名为awt的目录，在awt下存放着用户要存取的.class文件。

3）类路径

用户也许已注意到，Java.awt不是一个完整的路径名。路径名的其他部分包含在工程的类路径中，它有些像MS—DOS中的PATH环境变量。当Java运行时，它知道按什么路径去寻找像Java.awt这样的标准库中的类。当使用其他类库中的类时，用户需要告知Java寻找这些类库的路径。

要查看HelloWorld工程用到了哪些类库，右击Navigation面板中的hello.jpr，然后选择Properties，或选择File → Project Properties菜单项。

Java库的设置中包括以下类库：Swing 1.0，JBCL 2.0，JGL 3.1，VisiBroker 3.1。

该类库列表中的各表项不是类库的目录设置或类库的zip文件（使用zip文件允许把多类库放在一个类库的压缩文件中）。创建一个工程时，这些设置初始化为JBuilder中的缺省类库路径设置。用户可以选择Tools → Default Project Properties来改动它。

查看文件Frame1.Java的开头，会看到以下这些import语句：

```
import  Java.awt.*;
import  Java.awt.event.*;
import  borland.jbcl.control.*;
import  borland.jbcl.layout.*;
```

每个import语句都指定了一个部分路径，以当前系统中的目录分隔符来代替import中各字中的点号可得到该路径。

当输入Java.awt.*类时，这一步并不包含event子目录下的类。必须单独输入

Java.awt.event.* 类

在 MS—DOS 下，目录分隔符为一反斜杠（\），所以当编译器寻找组件类时，比如寻找 Label 类时，它知道按下列的任一部分路径寻找：

import Java \ awt \ Label;

import Java \ awt \ event \ Label;

import borland \ jbcl \ control \ Label;

import borlnnd \ jbcl \ layout \ Label;

为建立一个完全路径，编译器把可能的部分路径与类库列表中的每一项都组合起来。当从全命令行运行 Java.exe 时，用户可以用 classpath 选项来指定类库及路径，指定多个时用分号（；）隔开。比如，下边是用户以命令行方式执行 Java.exe 时包含两个类路径的情形：

…C: \ JBUILDER\ classes;C: \ jbuilder \ Java \ classes

当 Java 虚拟机寻找 Label 类时，它把每一个类库路径与 import 语句指定的每一个部分路径进行组合，产生一个与下表相似的查找表：

C:\ jbuilder \ classes \ Java \ awt \ Label

C:\ jbuilder \ Java \ classes \ Java \ awt \ event \ Label

C:\ jbuilder \ classes\ borland \ jbcl \ control \ Label

C:\ jbuilder \ classes \ borland \ jbcl \ layout \ Label

每个代表一个部分路径与类库路径的可能组合。值得庆幸的是，这样的查找非常的好，遇上第一个匹配时，查找过程结束。

以 Java.awt 路径名指定的类实际存储于 C：\ JBuilder2 \ Java \ Lib \ classes.zip 中。其中的 AWT 类拥有路径 Java \ awt。

4）完全合乎要求的类名

package 语句指定了一个完全的目录路径，对该路径必须准确匹配指定一个完全合乎要求的类名。比如，Label 类的源代码表示其路径说明符为：

package Java.awt;

因此，Label 类完全合乎要求的类名为 Java.awt.label。有些 Java 工具要求用户提供一个合乎要求的类名。比如，当以命令方式运行编译器 Java.exe 时，用户要给出一个完全合乎要求的名字。

类路径（或类库列表）中的条目之一以及包名称能产生一个完全的路径名，指向本机系统上的一个类。因为完全合乎要求的名字总是指定了一个完全的路径名，因此，在类路径中包含 C：\ JBuilder2 \ Java，而后去尝试指定 awt.Label 来提供一个完全合乎要求的名字，这是不可能的。

把包说明符看作为一个长链，说明符中的每个包/目录都是链上的一环，但链不允许断开。不管用户何时提供一个包名，他都提供整个的一条链。不是这样，就是什么也没提供。用户也可以把类路径看成是墙上的一套挂衣钩。每一个类路径就是一个可以用来挂一个包链的挂钩 在链的末端就是要找的类。用户也可以在同一挂钩上挂多个链（多个包）。用户甚至可以拥有某一条链的多个拷贝，亦即同一包中的一些入口悬挂于不同的挂钩（类

路径）上。

使用基于URL的目录名，由于Java能保证指向一个类的目录路径，因此Sun公司已建议使用基于URL的包名，以使软件开发商能单独保存他们的类库。比如，在JDK1.1中，Sun使用了com.sun.Java.swing指定Swing类的路径，因此这些类被添加到了Java1.1的包中。而在JDK1.2中，包的名字变为Java.awt.swing，它们是与标准Java类一起分发的。

在指定Swing类的package语句中，用户看到com.sun.Java.swing。打开这些类时，类路径指向一个“挂有”com.sun.Java.swing.的目录“挂钩”。如果用户查看包含swing类的包文件（jar file）时，会在文件中看到每一个类前面的com\sun\Java\swing路径。

如果用户开发的是大范围使用的类，使用一个类似的命名策略，能帮助防止用户自己的类与其他开发商开发的类之间的命名冲突。

举个更具体的例子，用户可以把Java.awt看成是指定了一个目录结构，该结构挂住了用户系统中的某些目录。使用类路径，用户可以设置挂钩来挂住系统中的任何目录。但挂钩的下面，该结构包含了一个Java目录以及其中的一个awt子目录和该子目录中的Label类。进而，用户知道开发一类时所在的目录结构与使用该类时所在的目录结构是准确相同的。

1.5.4.6 定义变量

Frame1.Java中的如下代码行定义了一个名为label1的变量：

```
Label label1=new Label ( );
```

数据类型的名字位于左边。在本例中，数据类型是Label类，因此Label1是一个含Label对象的变量。数据类型可以不是一个类名，而是一种基本数据类型，或者如String的通用类型，如int的数据类型（带符号的32位整数）。

一个基本数据类型拥有一个简单值（数字、字符或布尔值，如true或false），而不是对一个对象的引用。基本数据变量除了它的一个值以外没有方法，也没有其他属性。从某个角度来看，基本数据变量像用户所定义过的最简单的对象。

在本例中，label1变量在定义的同时被赋予一个值。但变量在定义的同时也可以不给它赋值，具体方法如下：

```
Label label1;
```

当用户以这种方式定义变量时，需要在后面的程序中用一个赋值语句给变量赋值：

```
label1=new label1();
```

等号（=）为赋值操作符。它使 符号右边的指定值存放在等号左边指定的变量中。因为label1已被说明为Label类型，所以这个赋值语句中不包含数据类型说明符。

如用户第二次包含了Label类型，将会返回一条“重复定义（duplicate definition）”出错信息，说明变量label1已经被定义了（用户可以尝试这么做，仅仅是为了在运行程序时看到这条出错信息）。另一方面，也有用户可以重定义变量而不产生错误的时候。这种能力依赖于一个概念“变量作用域（variable scope）”。

按照惯例，变量名以小写字母开头，以便与以大写字母开头的类名区别开来。所以label1中的头字符“l”为小写形式。

1.5.4.7 定义常量

关键字final通过确保一次只赋给变量一个值使之成为一个常量。这是对“最终变量

(final variable)"的一个特别奇怪的解释——换言之，"最终变量"是不变化的"变量"。Label类在下行代码中定义了一个常量LEFT：

```
public static final int LEET=0;
```

这行代码定义了一个int(整型)型常量LEFT，用来指定标签组件的文本应该是左对齐的。如果调用例程时指定"0"，也能达到同样的效果，但LEFT更加易读，也易被记住。

关键字static的意思是说，变量作为一个整体附属于类，而不是附属于任何具体的对象。该变量的存储空间只在类定义对象中分配一次，而不是在每一个使用创建的对象中都为它分配空间。因此，在用户工程中，类的每一个对象都被看成是相同的变量，而不是同名的多个变量。

按照惯例，最终变量（final variable）用全部大写字母形式指定。

用户也可以向一个方法或类的定义添加关键字final。当一个方法说明为final时，不允许派生类对该方法重载。当一个类说明为final时，该类不允许被派生。然而，与常量不同，一个说明为final的类或方法的名字并不总是大写形式。

1.5.4.8 定义方法

再看看HelloWorld.Java，名为main()的方法用下列代码行定义：

```
static public void main(String[ ] args){
        new HelloWorld( );
        }
```

main()是程序运行最先执行的方法（Java小程序没有main()方法，运行小程序的Web浏览器中有）。main()被定义的方法完全包含在HelloWorld类中。大括号内含有方法调用时要执行的代码。在本例中，只有简单的一行代码：

```
new HelloWorld( );
```

和变量名一样，按照惯例，方法名以小写字母打头，因此main()方法名的首字母m要小写。

1）方法返回值

像C语言一样，Java假定每个方法都返回一个值。方法返回值的数据类型以与指定变量的数据类型相同的方式指定，即通过把数据类型名放在方法名的前面实现。

如果一个方法没有返回值，可以使用关键字void。正如HelloWorld中的main()方法那样，关键字void指明了方法没有返回值：

```
static public void man(String[ ] args) {
```

Frame1.Java中包含另一个例子：

```
Public void jbInit( ) throws Exception {
```

在此，方法jbInit()定义为没有返回值。没有返回值的方法总是被它们自己调用。比如，用户可以看一下在构造函数Frame1（）中的jbInit()方法的调用：

```
        try{
    jbInit ( );
  }
```

这行代码在没有参数的情况下调用jbInit（）方法（没有数据返回）。

目前，请读者暂不管以前没有见过的关键字，在以后的章节中读者会逐渐了解它们。

它们出现在这里仅仅是为了帮助读者找到方法被定义和被调用的地方。

● 有返回值的方法

方法前面的类名或基本数据类型名表明该方法为一个返回指定类型值的函数。要见到一个这样的例子，选择 Frame1.Java 的 Structure 面板中的 Label1(图 1.12)，并双击（图 1.13）。

图 1.12　structure 面板中的 Label1 项

双击后，Label 类的源代码显示在 Content 面板上，使 getText（）方法高亮度显示（图 1.13）。

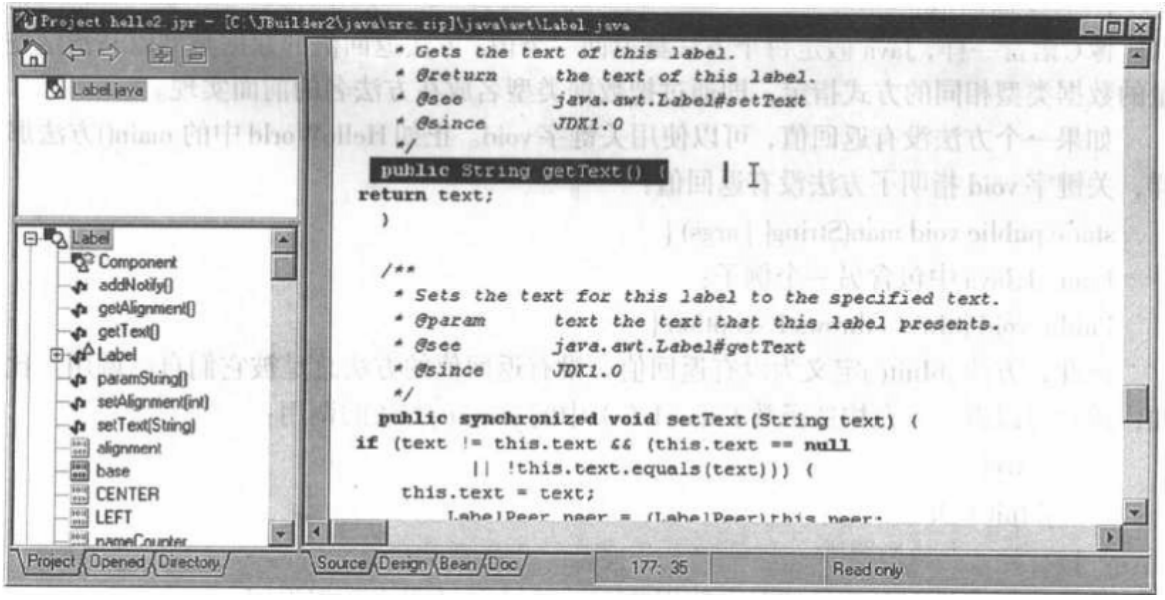

图 1.13　在 Content 面板中浏览 getText()方法

Content 面板现在显示如下代码：

```
public String getText ( ){
    return text;
}
```

关键字 String 说明 getText ( ) 方法返回一个 String 对象，它是 String 类的一个成员。实际的返回值是标签的文本。

return 语句：

return 语句结束方法的执行并指定要返回的值。返回值可以是明确的量（指用户写到程序的一个指定值）、变量或表达式，但它的类型必须与方法的返回类型一样。比如，在下面的方法中：

```
public String getText ( ){
    return text;
}
```

return 语句的意思是，把 String 变量 text 的值返回给调用 getText()方法的代码。在返回一个值的方法中需要 return 语句。对于 void 返回类型的方法，return 语句是可选 的，因为在方法的最后一条指令执行完后，方法自动返回。对于有返回值和无返回值的两类方法，return 语句可以放在方法中的任何地方，以在没执行到最后一条指令之前就从方法中退出。

2）定义方法参数

除了返回值以外，方法也可以处理作为参数值传递过来的数据，以控制它们的操作用户在()指定传送给方法的参数。比如，选择 Structure 面板中 Label 类的 setText (String) 方法，显示以下代码：

```
public synchronized setText ( String text ){
  …
}
```

这行代码的意思是，setText ( ) 方法接受一个 String 类型的参数。在该方法中用名字 text 来引用该参数。

参数说明看起来好像变量说明。每个参数以数据类型后跟变量名字的方式指定。唯一的不同之处在于参数的说明不能够指定一个初值。

● 多参数

多个参数间以逗号（,）分开。如果 setText ( ) 方法接受两个参数，它的定义看起来像以下代码：

```
public synchronized void setText( String text1,  String text2 ){
…
}
```

● 方法特征

用户可以以相同的名字定义多个方法，只要它们有不同的特征（方法名与变量的数目和类型）就行。比如，单击 Structure 面板中紧挨 Label 项的“+”号，显示以下三个不同的方法变种：

Label ( )

Label (String)

Label (String, int)

用到以上哪一个取决于调用时指定的参数。

3）定义构造函数

构造函数是一种特殊的方法。用类定义创建对象时，构造函数初始化对象中的变量。实际上，对象只能通过使用类的构造函数来创建。

下面是 HelloWorld 类的构造函数：

```
//Construct the application
public HelloWorld ( ){
...
}
```

除了以下两点外，构造函数看起来像方法：

- 构造函数的名字总是与类名相同
- 没有返回值

4）定义多个构造函数

因为构造函数是方法，所以它可以接收参数来控制对象的初始化。请回想一下：多个方法只要它们拥有不同的特征就可以取相同的名字。同理，用户也可以拥有一个类的多个构造函数，只要它们有不同的特征就行。因为构造函数的名字都相同，所以参数的数目和类型可以变化。下面再次列出 Label 类的构造函数：

Label ( )

Label (String)

Label (String, int)

每个类都自动定义一个缺省的构造函数，所以当用户不想作特别的初始化时，不必担心是否定义它。缺省的构造函数与类名相同，没有参数。缺省的构造函数把类中的每一个参数都初始化为它的缺省值。

Java 自动为没有定义构造函数的类定义缺省的构造函数。Label 类的缺省构造函数名为 Label ( )，但如定义了其他构造函数，如 Label (String)，就必须具体定义一个形如 Label ( ) 的无参数的函数，否则 Label()将不会存在。

1.5.4.9 浏览接口定义

用户使用关键字 interface 来指定接口，使用关键字 implements 来指定类属于那个接口。下面将进行一些浏览。

1）在 Label.Java 中，在 Structure 面板中选取 Component。

Content 面板上（图 1.14）显示 Lable 类是 Component 类的派生类。

Structure 面板上紧挨 Component 图标上的黄色星形说明 Component 是 Label 类的基类。基类项总是类名下的第一项。

2）双击 Structure 面板中的 Component。

Content 面板上（图 1.15）显示 Component 类的定义。

3）在 Structure 面板中的 Component 下，单击 Object。

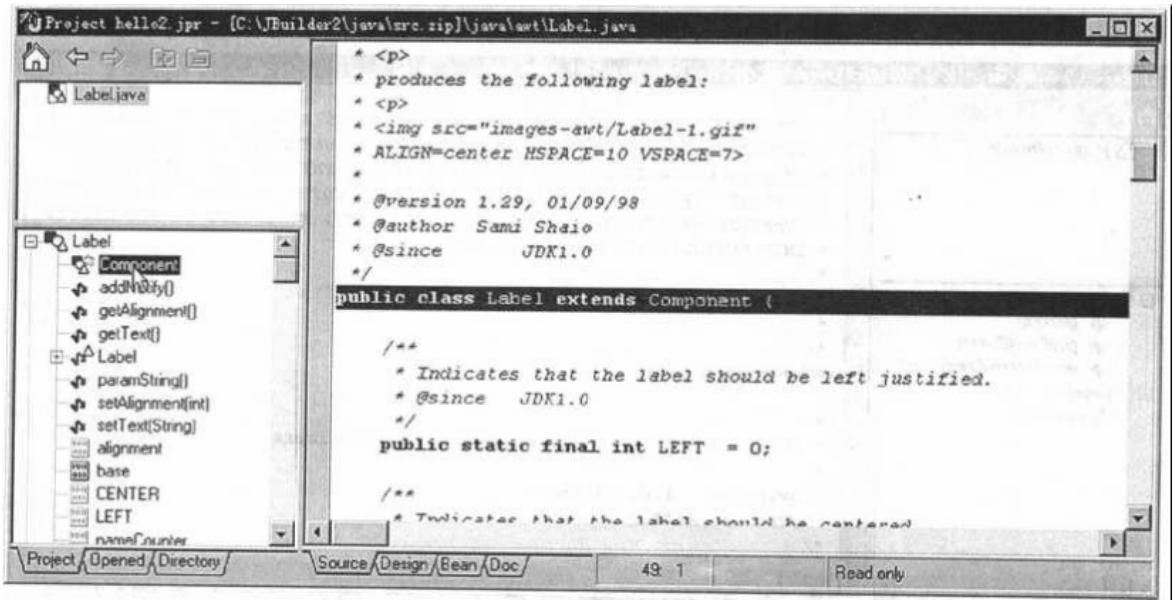

图 1.14　在 Label.Java 中浏览 Component 项

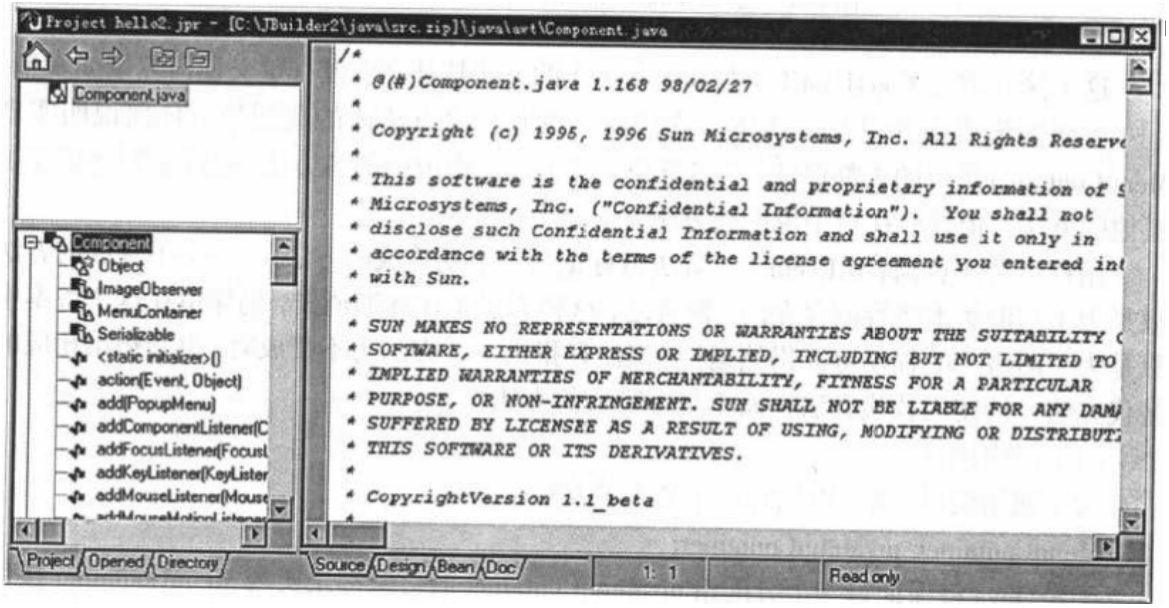

图 1.15　显示 Component.Java 的 Content 面板

用户看到像下面这样开始的 Component 类的定义：

public abstract class Component implements ImageObserver, …

关键字 implements 位于 Component 类所提供的接口列表的前面，这些接口包括 ImageObserver、MenuContainer 和 Serializable。在 Structure 面板中，紧挨 ImageObserver 的图标上含有一个黄色的、风格化的“i”，以帮助用户把这些项标识接口定义。

4）双击 Structure 面板中的 Menu Container 项，并向下滚动直至看到 MenuContainer 的整个接口定义。

Content 面板（图 1.16）现在显示了 MenuContainer 接口的定义。

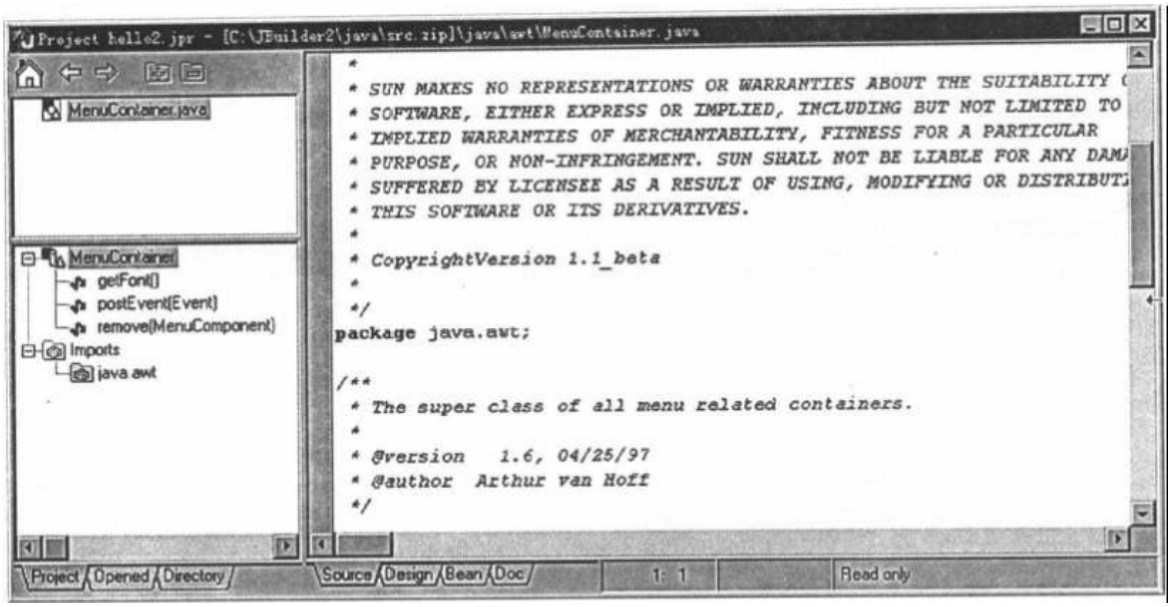

图 1.16　显示 MenuContainer.Java 的 Content 面板

这个接口定义了 getFont（）与 remove（）的方法特征。它没有定义任何变量或常量。

方法的说明中并没有实现体，它们甚至没有包含方法体的花括号。任何说明实现 MenuContainer接口的类都保证：它们至少定义了接口中说明的各方法。也许它们还定义了其他的方法，但这些接口中说明的方法是必须有的。

用户将会注意到 postEvent（）方法被标记为“不受赞成的方法”。一个不赞成的方法只是为了与旧版本兼容而存在的。换言之，这种方法正在逐渐消失，仍保留在这个版本中只是为了旧版本的程序可以仍然运行。当注释指出一个方法为不赞成时，也会指出相应的替代方法。请使用替代方法，而不是不赞成的方法。

（1）使用接口

为了使用接口，用户需创建一个如下的变量：

MenuContainer myMenuContainer;

于是，Java 将保证每一个分配给 myMenuContainer 的对象都是实现 MenuContainer 接口的对象。然后，用户就可以调用 getFont()和 remove()方法，可以肯定这些方法是由 MyMenuContainer 提供的，而不管包含这些方法的对象是 Label 组件还是其他组件，或是一个完全不同的对象(该对象实现 MenuContainer 接口)。

作为惯例，包名全部小写，变量名与方法名以小写字母开头，类名以大写字母开头，最终变量（常量）名全部大写。这些惯例不是必须如此，但它们有助于用户阅读源代码时一目了然，因为几乎所有的程序员都遵循它们，因此它们也能帮助用户阅读其他程序员编写的代码。

两个另外的命名惯例不是这样的标准。一个是接口的命名。因为接口是相对较新的东西，在接口命名上有一对不同的惯例。一个是使用大写字母 I 开头，后跟大写的名字串。

IRace 就属于这种命名。另一个命名惯例使用后缀 -able 或 -ible，产生一个像 Raceable 这样的接口名。

表 1.3 和表 1.4 总结了这些命名惯例。

表 1.3　标准命名惯例

程序元素	大 小 写	例 子
包名	全部小写	schduler
类名	首字符大写，其他不限	MyClass
方法名	首字符小写，其他不限	myMethod
变量名	首字符小写，其他不限	myNumber
常量（最终变量）名	全部大写	PI

表 1.4　其他命名惯例

程序元素	命 名 惯 例	例 子
接口	首字符大写，以 -able 或 -ible 结尾	Raceable
接口	首字母为 I，后跟名称	IRace
参数	首字符小写，以 a- 或 an- 开头	avariable,anObject

（2）派生接口

像类定义那样，接口定义也可以使用关键字 extends，来从基本接口进行继承。这主要是一种方便的机制，使用户说明他们实现一个接口而不是多个接口。当用户执行这样的一个接口时，需要在定义派生接口中说明的方法的同时，定义基本接口中说明的每个方法。

1.5.4.10　定义抽象类

再看看 Component 类，注意该类定义中的关键字 abstract：

public abstract class Component implements ImageObserver，…

关键字 abstract 说明 Component 类不能用来创建对象。它的存在仅仅是为其他从此派生的类提供方法和属性。比如，标签（Label）是一种组件，所以 Label 是由 Component 派生来的。用户可以创建一个 Label 对象，所以 Label 有时被称为具体类（concrete class）。

组件有许多种，包括列表框、按钮和文本框。如果让用户创建一个组件，用户就会问“哪一种？”用户这样问是因为组件是用来谈论一类事物用到抽象概念。它不指向一个实际的事物本身，所以组件是抽象类。

抽象类可以说明抽象方法，也就是用了关键字 abstract 的方法特征的说明（方法特征的说明不带方法实现体）。在抽象类中定义的抽象方法像在接口中定义的方法那样工作。任

何由抽象类派生的类都必须执行抽象的方法，或者类本身是一个抽象类，它把抽象方法传送给第一个实现该方法的具体类。

比如，如果本节前面的Vehicle类被定义为抽象类，那么stop（）方法也可以被定义为抽象方法。

这使得所有由此抽象类派生的具体类必须执行stop（）方法本身，而不继承缺省的例程。由于缺乏每种具体的Vehicle的针对性的操作，Vehicle类可能会碰到一些情况，所以在Vehicle类中定义stop（）方法也许会好一些。

1.5.5 创建和使用对象

1.5.5.1 创建对象

在Framel.Java中，下面这行代码创建了一个新的Label对象：

```
Label  labell=new Label ( );
```

语句的第一部分为类型说明部分。它说明用户定义了属于Label类的一个名为Labell的变量。紧接着为赋值操作符“=”。关键字new说明用户正在创建一个新对象。最后是Label类构造函数的名字。在本例中，没有指定参数，关键字new调用对象的构造函数来创建对象及初始化对象的属性。

1.5.5.2 定义变量与创建对象

注意下面两行代码的不同之处：

```
Label  labell;
Label  labell = new  Label ( );
```

第一行定义了变量labell，但没有创建对象。这说明labell可以成为Label类的对象，但此时它的值为Null。如果程序的后面部分没有对它赋一个对象就引用它，用户在运行程序时会碰到“Null Pointer Exception（空指针异常）”的错误消息，意思是说labell这个变量没有初始化。上面的第二行代码定义了变量，创建了一个新对象赋给它。

串的创建是一个例外，并不要求用户用关键字new来创建一个对象。要创建一个String对象，可使用下面的语句：

```
String s =  “This  is  your  string。”;
```

Java程序由于创建串的操作非常频繁，且它的语法对C程序员说又是非常熟悉，所以不执行要求用以下的语句：

```
String s = new  String ( This  is  your  string。);
```

尽管String是一个类，但Java让用户把它当成了一个基本数据类型。

1.5.5.3 调用方法

用户可以通过指定名字来调用同一类中定义的一个方法。比如，在Framel.Java中，构造函数Framel（）以这种方式调用了jbInit（）方法：

```
pubic  Framel ( ){
  try{
    jbInit ( );
}
```

```
catch ( Exception e ){
  e.printStackTrace ( );
};
}
```

通过使用Java的点表示法来在另一个类中访问方法或变量。例如，Framel类中的jbInit ()初始化方法这样调用了 label1 的 setText ( ) 方法：

```
public void jbInit ( ) throws Exception{
...
label1.setText(Hello World);
}
```

用户还可以通过使用关键字this来引用当前对象，从而访问作为当前类组成部分的一个方法：

```
public void jbInit ( ) throws Exception{
...
this.setTitle ( Hello World );
...
}
```

1.5.5.4 访问对象变量

关键字this也经常用来区分局部变量与在类中定义的对象变量。一般情况下，当一个参数与类变量拥有相同的名字时，会进行这样的操作：

```
class myClass{
  int value;
   ...
  void setValue ( int value ){
  this.value=value;
```

在这种情况下，this.value引用属于类的值，但value引用属于setValue ( ) 方法的局部参数值。

关键字this只是在方法的参数与对象变量同名时才要求使用。

1.5.5.5 调用超类方法

用户会注意到setTitle ( ) 方法并没有在Framel.Java中定义。这是因为Framel类是派生的（中间隔着DecoratedFrame类），在Frame类中定义了setTitle ( ) 方法。图1.17显示了这种继承层次。

用户也可以使用关键字super来指明从超类（superclass）中得到这个方法。于是，Java编译器不是首先从Framel中寻找seTitle ( ) 方法，而是首先在DecoratedFrame类中寻找。此时，它不区分用户指定的是super.setTitle()还是this.setTitle()，因为只有一个名为setTitle()方法。但如果用户通过在Framel中定义了一个setTitle()方法来重载Frame中的setTitle()，那么，从Frame类中获得setTitle ( ) 方法的唯一途径就是使用super. setTitle ( )。在用户定义的setTitle ( ) 中可能会出现这种情况，为的是使用原始方法的功能，如图1.18所示。

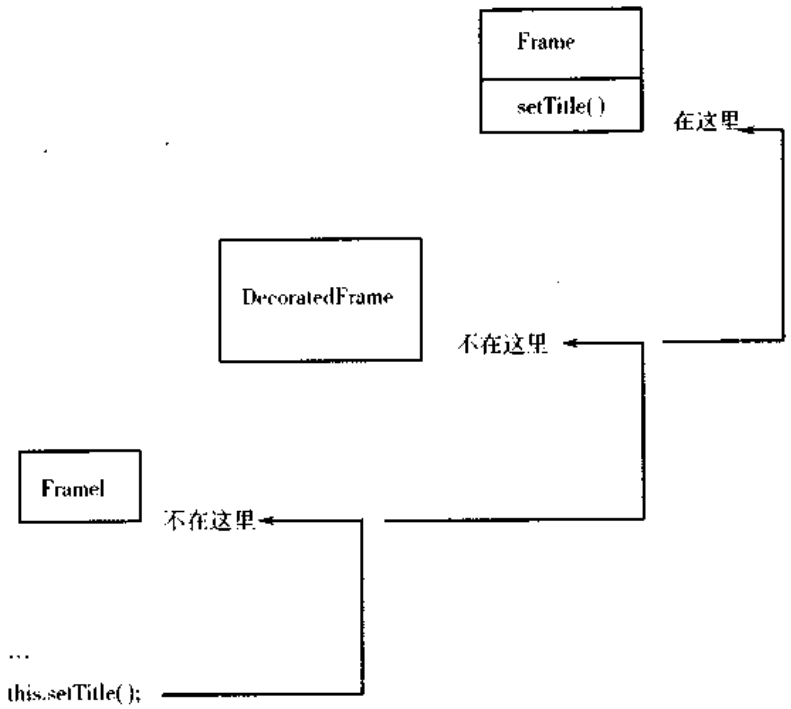

图 1.17　Frame 类的层次

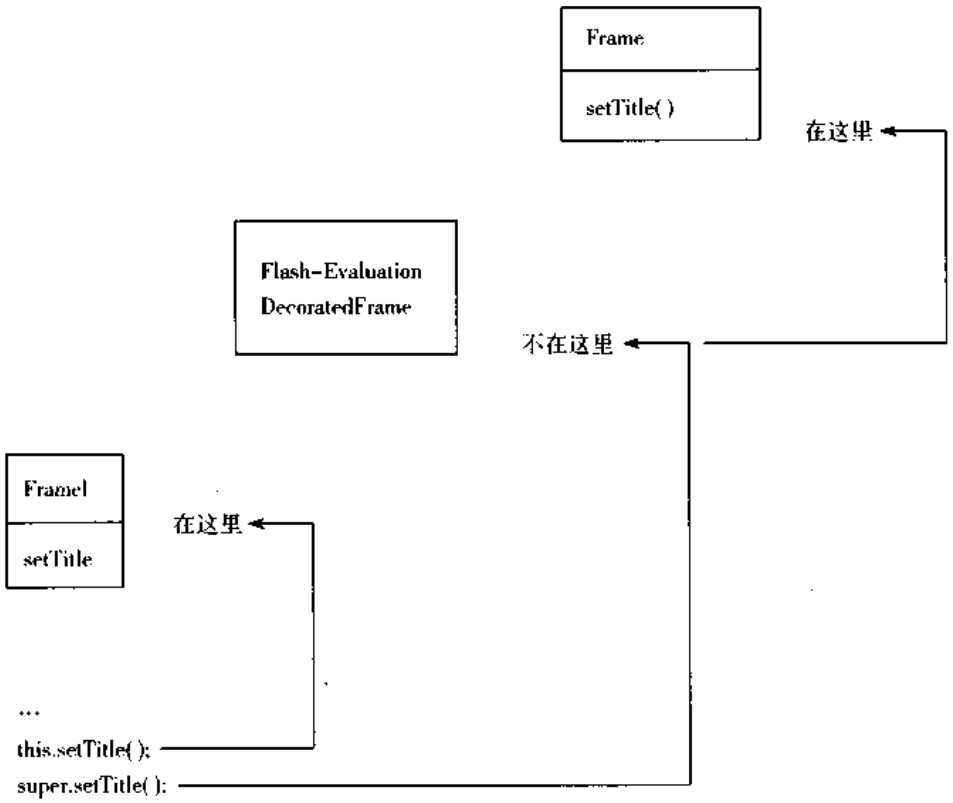

图 1.18　含有重载方法的 Frame 类层次

1.5.5.6 调用构造函数

关键字 this 和 super 也可以用来调用构造函数，但这样的调用只能在另一个构造函数中进行，而且这种调用必须是构造函数要做的第一件事。这样的调用被用于引发构造函数的一个不同版本，同时传递那个版本所要求使用的参数。例如，下列代码显示了 Label 类的缺省构造函数：

```
public Label ( ){
  this ( "", LEFT );
  }
```

这个版本的 L abel构造函数需要两个参数，一个为初始化文本值指定空字符串，另一个把“左对齐”作为缺省的对齐方式。关键字 super 同样以相同的方式从基类中引发构造函数，具体方法如下：

```
super ( );
```

如果一个构造函数没有明确调用另一个构造函数，那么会自动调用缺省的基类构造函数。这就是为什么超类中定义的变量都得到了初始化的缘故。

1.5.5.7 把方法的返回值赋给变量

可以使用返回值的方法把一个值赋给变量，方法如下：

```
String s = label1.getText ( );
```

像在C语言中那样，用户也可以通过在一行代码中使用方法本身来忽略返回值。在下行代码中，返回值就被简单忽略了：

```
label1.getText ( );
```

1.5.5.8 指定方法的参数

调用方法时，在（）中指定参数，参数以逗号（，）隔开。例如，label1 的 setText()方法调用时只带唯--的参数，即字符串 "HelloWorld"：

```
label1.setText ( "Hello world" );
```

定义方法时，用户必须指定参数的数据类型以及在方法定义中使用的参数名字。但当调用方法时，用户只要指定一个值或一个变量即可。

1.5.5.9 访问变量

通过使用变量的名字来访问当前类或当前方法中定义的变量。下面是 Helloworld.Java 中的几行代码：

```
//Construct the application
pubic HelloWorld( ) {
  Framel frame = new Framel( );
...
  Dimension screenSize =
  Toolkit.getDefaultToolkit( ).getScreenSize( );
Dimension frameSize = frame.getSize( );
...
```

这段代码创建了两 个Dimension类型的变量（也就是说，使用Dimension类）。变量名为 screeSize 和 frameSize：

```
Dimension  frameSize = frame.getSize ( );
```

第二个变量定义调用Frame对象中的一个方法，该返回值表达了Frame对象的尺寸大小。因为Frame是在当前方法中定义的，所以通过简单地指定它的名字来引用它。

与引用对象的方法一样，用户可以使用点号(.)来访问对象的变量。下面的代码得到在sereenSize对象中定义的height变量的值，并将其赋给frameSize对象中的height变量：

```
frameSize.height = screeSize.height;
```

变量的引用与方法的引用看起来是一样的，不过方法名后面跟有括号。

1）控制变量的作用域

在Framel.Java中定义Labell的代码如下：

```
public class HelloWorld{
…
Label labell = new Label( );
…
//Construct the application
public HelloWorld( ){
  …
```

变量Labell在类中定义，而类中的任何方法都在类之外定义。这意味着Labell可以被类中的任何一方法使用。

回想一下，一个变量定义作用的代码范围称为变量的作用域。超出作用域，变量便不存在。对于在类一级定义的变量，比如上面那个例子，缺省作用域实际上是整个包。变量作用域也被叫做变量的可见性，它决定了程序的哪些部分能够“看到”这个变量。

对比之下，HelloWorld.Java中如下定义了变量frame：

```
public class HelloWorld{
…
//Construct the application
public HelloWorld( ){
  Framel frame = new Framel( );
…
}
```

在这个例子中，变量frame在构造函数HelloWorld（）中定义，所以这个变量只能为该方法中的代码所访问。

2）控制变量的可见性

面向对象编程的一个重要宗旨就是提供通过封装来隐藏对象的实现部分。另一方面，要使对象有用，必须使它的一部分可以被访问。Java让用户决定一个方法的什么部分对程序的其他部分是可见的。

变量和方法有四种级别的可视性。这些不同的可视性级别决定了程序的什么部分可以访问类定义的变量和方法。从最开放到最限制，四种级别依次是：公共（public）、包局部（package local）（亦称包保护）、保护（protected）和私有（private）。

关键字public、protected或private跟变量或方法的定义连用，来指定相应的可视性级

别。如果这三个关键字都没被指定，那么类变量或方法的缺省可视性就是包局部。表 1.5 描述了这些可视性级别。

表 1.5　可视性级别

可见性级别	从中可访问的部分
public	程序中任何其他类
package local	同一包中的任何其他类
protected	派生这个类的任何类
private	仅在当前类中

图 1.19 显示了“另一个类”访问“用户类”中定义的变量与方法，取决于用户赋予的可见性级别。

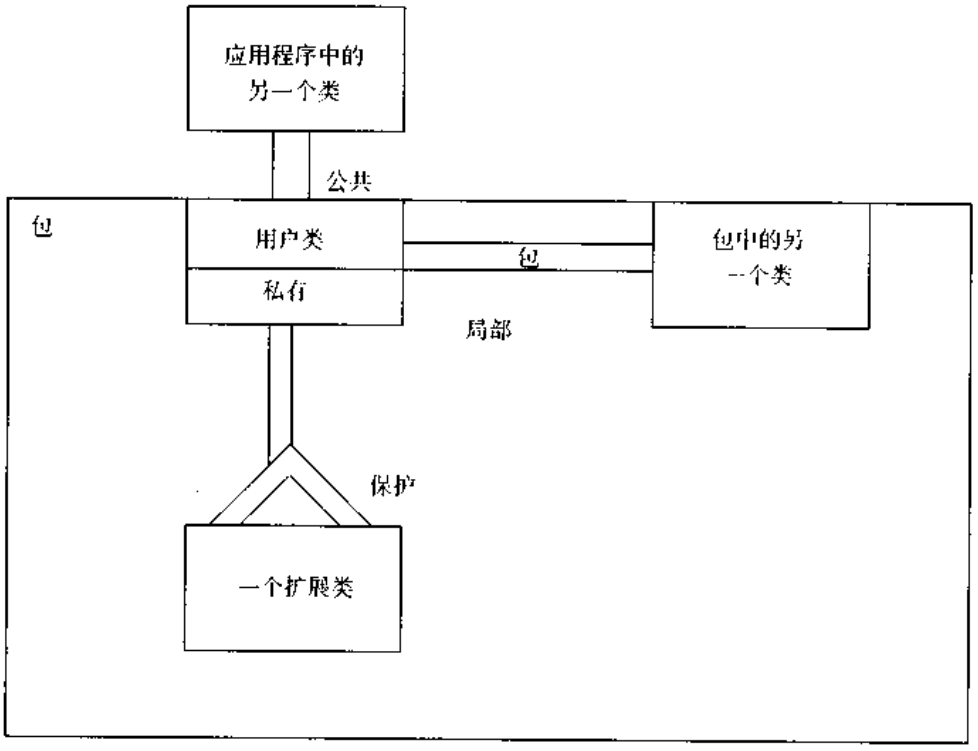

图 1.19　变量与方法的可见性级别

图 1.19 中部分说明举例如下：

```
public class Helloworld…          //public visibility
class MYClass…                    //package local visibility
public void main…                 //public visibility
```

```
void myMethod                    //package local visibility
int i;                           //package local visibility
protected int j                  //protected visibility
private int k;                   //private visibility
```

回想一下，包就等价于用户机器上文件系统的目录。所以“包局部访问”就意味着同一包中定义的任何类都可以访问它。一般情况下，协同工作的类赋给它的方法包局部可见性级别，以使它们之间可以方便地互相交流。对外界公开的形成包API的方法的可见性级别为“公开的（public）”。

有些变量与方法设定为不对其他类公开，但派生类想对它们重载或访问，那它们的可见性级别就定为保护级别（protected）。理想情况下，变量与方法不是private就是protected，这样才能仅通过方法来访问它们。这把一个类与剩余的代码尽最大可能分隔开，使类可以对其他代码毫无影响地进行改动。

然而，实际上，数据变量经常被赋予包局部可见性级别，来最小化编码工作量，并使之尽可能简单地与其他类互相交流。这样的缺点就是：如果一个类用不同的数据结构重新实现，那么同一包中使用现在数据的其他类也需要进行修正。然而，这个缺陷经常值得克服。只在极少情况下一个变量处于公开可见性级别。通常，那是需要避免的，因为如果改变了那个类，那么潜在的无数用户可能会受到冲击。

强制使用方法来访问变量，应用程序有可能保证内部的变量保持同步。比如，一个日期类包含了星期域和日域，那针对每个域的set( )方法应保证同时改变对应的域。在将来，如果添加了额外的域，它们也应该能同步。如果一个变量为公共变量的话，可访问该类的代码也许会直接改变这些数据。于是，接下来就无法保证内部数据保持同步了。

1.6 AWT类库及其应用

1.6.1 Java用户接口

Java提供了创建与用户接口的基本方法。所谓用户接口（User Interface，简称UI）是一个程序中与程序的使用者交互的那部分。用户接口有许多种形式，可从简单的命令行接口到由许多现代程序提供的图形用户接口，形式复杂度各异，它不仅仅包括用户所看到的，还包括用户所听到的；甚至一个程序与用户的交互速度也是用户接口一个很重要的部分。

Java环境提供了一些类来实现以下与用户接口功能：

1）用标准的输入输出显示文本。标准的输入输出是提供用户接口的一种传统形式，标准的输入输出对于测试和调试程序是十分有用的。

2）播放声音。

3）用属性保存用户的选项。用户可以用选项来保存applet和Application的信息。

4）获取applet的属性。属性是applet用来获取用户选项的唯一途径。

5）提供图形用户接口(Graphic User Interface，简称GUI)。

本节介绍抽象窗口工具箱(AWT)，它包含了用于编写GUI程序的全部类的集合。其他用户接口在以后章节介绍。

1.6.1.1 抽象窗口工具箱——AWT组件

在AWT（Abstract Windowing Toolkit）的概念中，窗口系统中所显示的各种对象都统称为"组件"（Component）。每种组件各有各的用途，如最常使用的按钮（Button）、列表框（List）等等，都是组件。AWT中，每一个组件都是通过继承AWT的Component类来实现的。

Component类定义了各种窗口对象中最基本最重要的方法和性质。除了菜单类外，所有代表窗口对象的类，都是component类所派生的子类（菜单类继承的是MenuComponent类）。除了每种组件自己的特殊方法以外，大部分基本方法都定义在Component类中。

1）基本控制

按钮（Button）、检查框（Checkbox）、选择框（Choice）、列表框（List）、菜单（Menu）和单行文本域(textField)

Button类、Checkbox类、Choice类、List类、MenuItem类和TextFied类提供了基本的控制。这些控制是用户向Java程序发出命令的最常用的方法。当用户激活一个控制时（例如点中鼠标或在文本域中按回车键），它将传递一个事件（ACTION—EVENT），包含这个控制的对象将通过执行action（）方法来对此事件作出反应。

2）实现用户输入的真正方法

滑动器(Slider)滚动棒(Scrollbar)和多行文本域(TextArea)。当基本的控制不能满足用户需要时，用户可以使用滚动棒和多行文本域来实现用户输入，Scrollbar类用作实现滑动器和滚动棒的功能，TextArea类提供了一个区域来显示或编辑多行文本（文本域自身就带有滚动棒）。

3）用Canvas类创建自定义的组件

Canvas类允许用户创建自定义的组件。通过继承Canvas类，用户可以在一个绘画程序（图像处理器或游戏）中在屏幕上画出自定义的图形，而且还可以进行各种事件处理。

4）标签

标签只能显示不可编辑的、不可选择行数的文本。

5）容器

在AWT组件中，有一种组件比较特别，这种组件是用来包含其他组件的，称之为容器（container）。用户可以把各种组件放入到容器中，也可以把容器放到另一个容器中（因为容器本身也是组件，它们都是Component类的子类），从而形成具有层次的组件结构。

AWT用Container类来定义最基本的容器，所有可以作为容器的窗口对象都是Container类的实例或子类所生成的对象。

AWT提供了四个容器类。它们是Window类及其两个子类，Frame类和Dialog类，以及Panel类。除了AWT提供的容器外，Applet类也是一个容器，它是Panel类的一个子类。

表1.6列出了AWT提供的每个容器类的简要说明。

表 1.6　AWT 四种容器类说明

Window 类	最高级的显示面窗口。Window 类的实例不能附加或嵌入到另一个容器中。Window 类的实例没有边框和标题
Frame 类	具有边框或标题的最高级显示面（窗口）。Frame 类的实例可以有一个菜单条，否则它非常类似于 Window 类的实例
Dialog 类	具有边框或标题的最高级显示面（窗口）。Dialog 类的实例只有在一个相关的 Frame 类的实例存在时才能存在
Panel 类	容器组件的通用容器。Panel 类实例提供了一个可以加入组件的容器

1.6.1.2　AWT 中的其他类

Java.awt 软件包不仅仅包含组件，还包含与绘图和事件处理有关的类。正如我们前面所讲到的，组件是集中放置在容器中的。但我们没有提到每个容器如何使用布局管理器（Layout manager）来控制它所包含的组件尺寸和排放位置。Java.awt 软件提供了几个布局管理器类（BorderLayout 类、CardLaout 类等）。用户可根据需要选择适当的布局管理器对组件进行布局。

Java.awt 软件包还提供了几个类来表示尺寸和形状。例如：Dimension 类，它指定了一个矩形区域的尺寸；Insets 类，它通常用来指定容器的周边与容器的显示区域之间的空白大小；Shape 类则包含了点、矩形和多边形。

AWT 中的 Color 类对于表示和调整颜色是十分有用的。它定义了一些常量来表示常用的颜色，比如，Color.black 表示黑色，虽然它使用的颜色是 RGB(red-green-blue)格式的，但它也可以理解 HSB（hue-saturation-brightness）格式。

AWT 中的 Image 类提供了图像数据的途径。Applet 可以通过 Applet 的 getImage（）方法获得 GIF(Graphic Information format)和 JPEG(Joint photographic Expert Group)图像的对象。非 applet 类则可以使用 Toolkit 类获得。Toolkit 类针对 AWT 的平台独立性，提供了一个非独立于平台的接口。但是，大部分程序除了获得图像外，不能直接使用 Toolkit 对象。Java 中，图像的载入可以是异步的，也就是说用户可以拥有一个有效的图像对象，而图像数据可以不载入（或图像数据还不存在）。用 MediaTracker 对象，用户可以跟踪图像载入的状态。MediaTracker 类虽然仅能处理图像，但也可以处理其他类型媒体，比如声音。

用户还可以使用 Font 和 FontMetrics 对象来控制所画文本的外观。Font 类使用户可以获得关于字体的信息并创建不同的字体对象。用 FonMetrics 对象，用户可以获得一些特定字体在尺寸上的细节信息。用户可以用 Component 类和 Graphics 类的 setFont（）方法设置一个组件所使用的字体。

最后要提到的是 Graphics 类和 Event 类。Graphics 类和 setFont 类对于 AWT 的绘画及事件处理系统是非常重要的。没有 Graphics 对象，程序就不能往屏幕上画任何东西。而一个事件对象则表示一个用户的动作，比如点中鼠标。

1.6.1.3 事件处理

用户接口起到了使程序的使用者与程序之间进行交互的作用，如果使用户接口具有交互能力，就必须涉及到事件处理。所谓“事件”，就是指在系统中有某些我们所关心的事情（如：鼠标移动，用户按下了某个键等）发生了，然后系统便通知去处理这些事情。这样的概念在一般采用窗口用户界面环境的操作系统中，是十分常见的。当用户在组件上作出动作时（如：在组件上点鼠标或按回车键），就创建了一个 Event 对象。AWT 的事件处理系统会自动将这个Event对象沿组件的继承层次结构向上传递，每一个组件在windows系统完全处理这个 Event 对象以前，有机会对此事件作出反应。每个组件的事件处理程序既可以忽略此事件，也可以用下面的方法之一对此事件作出反应：

- 在Event对象沿组件的继承层次结构向上传递以前修改它。例如：对于一个只能以大写方式显示字符的TextField类的子类，我们通过修改事件，使其对小写字母的按键也能作出反应。
- 以其他方式对事件作出反应。例如，一个 TextField 类的子类可以通过调用处理文本域内容的方法来对按下回车键作出反应。
- 终止进一步处理事件。例如：如果一个无效字符被输入到文本域中，事件处理程序可以终止此事件，使其不能被进一步处理。

从组件的角度来看，AWT的事件处理系统更像一个事件过滤系统。概括起来讲，依赖于window，系统的代码生成了一个事件，但在此代码完全处理这个事件以前，组件有机会对此事件进行修改，作出反应或破坏此事件。

1）Event 对象

每个事件发生后，都会创建一个 Event 对象。一个 Event 对象包含了以下信息：

（1）id ——事件的类型。或是按键，或是点鼠标。

（2）target ——发生事件的对象。

（3）when ——时间印迹。指出事件发生的时间。

（4）x，y ——发生事件的坐标。

（5）key——键盘事件中被按下的键。

（6）arg ——一个与事件相关的任意参数。比如：在组件上显示的字符串。

（7）medifier ——修饰键的状态（即 Alt,Ctrl 键的状态）。

2）如何实现事件处理程序

组件可以通过执行handleEvent()方法或通过执行针对某种事件的特定方法来对事件作出反应。后面一种方法与前一种是等价的，因为handeEvent()方法的缺省定义是调用处理某种事件的特定方法。这些特定方法包括：mouseEnter()，mouseExit()，mouseMove()，mouseUP()，mouseDown()，mouseDrag()，keyDown()和 action()。

3）典型事件处理

（1）鼠标点中事件。当在其 applet 中点鼠标时，事件就产生了，可以利用鼠标点中事件来做一些简单的事情。比如，触发声音开关、清除屏幕等。也可以将鼠标点中和鼠标移动两种事件合起来使用，这样可以完成一些复杂的功能。

当点中鼠标一次，AWT 会生成两个事件：mouseDown 事件和 MouseUp 事件。当鼠标键被按下时，AWT 生成 mouseDown 事件；当鼠标键松开时，AWT 生成 mousUP 事件。

那么一个鼠标动作为什么要用两个事件呢？因为，针对鼠标键的按下和松开状态，用户可能想要做不同的事情。例如，在下拉式菜单中，mouseDown 事件将菜单展开，而mouseUP事件可以用来选中一个菜单项（其中还要涉及到mouseDrag事件，将在以后介绍）。如果只用一个事件来处理mouseUP和mouseDown两个动作，用户是不可能做到上面所提到的那种交互方式的。

在applet中处理鼠标事件是非常容易的，所要做的只是在自己的applet中重载正确的方法。当特定的事件发生时，相应的方法会被自动调用。下面是处理mouseDown事件的方法：

```
public boolean mouseDown( Event evt,  int x,  int y )
    {
        ......
    }
```

mouseDown方法（以及mouseUp方法）带有三个参数：事件本身和事件发生所处的坐标值(x 和 y)。

事件参数（上例中为evt）是Event类的一个实例。所有系统生成的事件都是Event类的一个实例，它包含了事件在何处、何时发生以及事件类型的信息。它还包含了用户想知道的关于事件的一些其他信息。

事件发生所在的坐标值可以通过参数x, y获得。利用它们，可以精确地知道在什么位置上发生了鼠标点中事件。例如，下面的例子在mouseDown事件发生时，将显示mouseDown事件发生所处的x, y 坐标值。

```
poblic boolean mouseDown ( Event evt,  int x,  int y )
 {
    system.out.println ( "mouse down at" + x + "," +y );
    return  true;
 }
```

如果在applet中加入此方法，每当在applet中点中鼠标时，都会显示一条信息。上述方法的返回值是布尔类型，这一点对于创建用户接口和管理接口输入都是非常重要的。因为事件处理程序返回真值还是假值决定了一个指定的UI组件是否能够截获此事件，或此UI组件是否要将此事件传递给包含它的组件。通常的规则是：如果用户的事件处理程序处理了该事件，它就应该返回真值。

由于mouseUp（）方法在用户松开鼠标键时被调用，因此要想处理mouseUp（）事件，用户只要将 mouseUp（）方法加到自己的applet中就可以了。

```
public boolean mouseUp ( Event evt,  int x,  int y )
    {
.....
    }
```

（2）鼠标移动。每当鼠标向任何方向移动时，一个鼠标移动事件就会生成。Java中有两种鼠标移动事件：mouseDrag事件和mouseMove事件。当按下鼠标键后，鼠标再移动时，就会产生mouseDrag事件。如果没有按下鼠标键而单独移动鼠标则会产生mouseMove事件。

为了管理鼠标移动事件，可以使用 mouseDrag()和 mouseMove（ ）方法。

● mouseDrag 和 mouseMove

当将 mouseDrag( )和 mouseMove( )方法包含在自己的 applet 代码中时，就可以截获并处理鼠标移动事件。mouseMove（ ）方法用于鼠标键没有被按下而移动鼠标的情况：

```
public boolean mouseMove(Event evt,int x,int y)
{
  ......
}
```

mouseDrag（ ）方法用于按下鼠标键并移动鼠标的情况（一个完整的鼠标拖动包括一个 mouseDown 事件，一系列的 mouseDrag 事件，以及鼠标键被抬起的一个 mouseUp 事件）。

```
public boolean mouseDrag(Event evt，int x，int y)
 {
  ......
 }
```

● mouseEnter 和 mouseExit

mouseEnter( )和 mouseExit（ ）这两个方法在鼠标的指示箭头进入 applet：或离开 applet 时，分别被调用。

mouseEnter( )方法和mouseExit( )方法都有三个参数：事件对象和鼠标进入或离开applet 时，所处的 x，y 坐标值。

```
public boolean mouseEnter（Event evt，int x，int y）
{
 ......
 }
  public boolean mouseExit(Event evt，int x,int y)
  {
  ......
  }
```

（3）键盘事件。无论何时，当按下键盘上的某一个键时，就会产生键盘事件。利用键盘事件，可以获得他所按下键的值或者字符。

● keyDown( )和 keyUp( )方法

使用的 keyDown( )方法，可以获取键盘事件：

```
public boolean keyDown（Event evt，int key）
    {
     ......
      }
```

在 keyDown 事件中，被按下键的值将作为 key 参数传递给 keyDown（ ）方法。它们是整型的 ASCII 码值，包括字母、数字、字符、功能键、Tab 键、回车键等。如果要以字符的形式使用它们（例如，将它们打印出来），必须将它们强行转化为字符型：

```
currentchar=（char）key;
```

下面是一个简单的例子，它将用户所按的键以 ASCII 码和字符的形式显示出来：

```
public boolean keyDown(Event evt, int key)
{
  System.out.println("ASCII Value:" +key );
  System.out.println("character:"+(char)key);
        return true;
  }
```

每个 keyDown 事件有相对应的 keyUp 事件。为了获取 keyUp 事件，应使用 keyUp ( ) 方法：

```
public booleankeyUp(Event evt, int key )
   {
    ......
   }
```

● 缺省键

Event 类提供了一个类变量集。这些类变量与几种标准的非字母数字键有关。如果在接口中使用了这些键，可以在 keyDown ( ) 方法中检测这些按键的名字而不用检测它们的数值，从而使代码的可读性提高。例如，判断键是否被按下，可以使用下面的代码：

```
if (key==Event.Up)
  {
   ......
   }
```

因为这些类变量的值是整型，因此也可以使用 switch 表达式来检测它们。

表 1.7 列出用于不同键的标准事件类变量和它们所表示的实际键。

表 1.7　由 Event 类定义的标准键

Class variable(类变量)	Represented key(实际键)
Event.HOME	The Home key(home 键)
Event.END	The End key(End 键)
Event.PGUP	The PageUp key(pageUp 键)
Event.PGDN	The PageDown key(pageDown 键)
Event.UP	the up arrow(向上箭头键)
Event.DOWN	the down arrow(向下箭头键)
Event.LEFT	the left arrow(向左箭键)
Event.RIGHT	the right arrow(向右箭头键)

● 检测修饰键

shift键、Control键和meta键是修饰键。它们自己不会生成按键事件。但是，当获得一个普通的鼠标或键盘事件时，可以检测在事件发生时这些修饰键是否按下。shift键和字母数字键一起按下，与单独按下字母数字键是不同的，这会生成另一个不同的事件。对于其他事件，尤其是鼠标事件，也可以处理修饰键同时被按下的事件，与常规不同。

Event类提供了三个方法用于测试修饰键是否被按下：shiftDown（）方法，metaDown（）方法和ControlDown（）方法。它们都返回布尔值。返回的是真值还是假值反映了修饰键是否真正按下。可以在任何事件处理方法中（鼠标事件或键盘事件）使用这三个方法。

```
public boolean mouseDown(Event evt, int x, int t)
    {
     if (evt.shiftDown())
          {
     //handle shift-click
    {
     else
         {
   //handle regular click
     }
   }
```

● Awt的事件处理程序

在Java中，用于处理基本事件的特定方法（例如前面我们所学的mouseDown（），keyDown（）等）是由事件处理程序handleEvent（）调用的。

```
puhlic boolean handleEvent(Event evt)
   {
       ......
   }
```

为了知道特定的事件，应检查Event对象的实例变量ID。事件的ID是一个整数。Event定义了一个完整的事件ID集作为类变量，可以在自己的handleEvent（）方法内检测这些类变量，从而可以检测相应的事件。由于这些类变量是整型常量，因此可以用switch表达式。

可以在handleEvent（）方法中检测并处理以下键盘事件：

当按下键盘键时，生成Event.KEY_PRESS事件（同mouseDown（）方法一样）。

当按下的键松开时，生成Event.RELEASE事件。

当键被按下后又松开，会生成Event. KEY_ACTION和Event.KEY_ACTION_RELEASE事件。

用户还可以在handleEvent（）方法中检测并处理以下鼠标事件：

当鼠标键被按下时，生成Event.MOUSE_DOWN事件（同mouseDown（）方法）。

当鼠标键松开时，生成Event.MOUSE_UP事件（同mouseUp（）方法）。

当鼠标移动时，会产生Event.MOUSE_MOVE事件（同mouseMove（）方法）。

当鼠标键按下，鼠标移动时，会产生Event.MOUSE_DRAG事件（同mouseDrag（）方法）。

当鼠标进入applet或applet中的一个组件时，会产生Event. MOUSE_ENTER事件。同样，用mouseEnter（）方法，也可以检测并处理此事件。

当鼠标指示离开applet时，会生成Event. MOUSE_EXIT事件。同样，用mouseExit（）方法，也可以检测并处理此事件。

值得注意的是，如果在自己的类中重载了handleEvent（）方法，处理事件的方法（如mouseDown.keyDown（）等）将不会被调用，除非在handleEvent()方法中直接调用它们。因此，在决定使用handleEveni（）方法时，对此要十分注意。

一种解决方法是当检测自己所感兴趣的事件时，如果无此事件，就调用Super.handleEvent()。这样做是为了让超类定义的handleEvent()方法可以处理此事件。下面的例子表示出了此方法是如何来实现的：

```
public boolean handleEvent ( Event evt )
{
  if(evt.id==Event.MOUSE_DOWN){
    //process the mouse down
     return true;
   }
  else
   {
     return super.handleEvent(evt);
   }
}
```

1.6.2 用组件构造用户接口

1.6.2.1 使用组件的一般规则

本节将介绍如何将组件加到容器中，组件从Component类中继承的功能以及如何改变组件的外观。

1）将组件加到容器中

任何组件（除了某种类型的窗口外）要在屏幕上显示出来，必须要将它加到容器中。当然，容器本身也是组件，它们也可以加到另一个容器中。而像Frame和Dialog这样的窗口，它们是最高级的容器，不能将它们加到其他容器中去。

Container类中定义了两种类型的add( )方法用来加入组件：单参数的add（）方法和双参数的add( )方法。使用哪一种方法要依赖于容器所使用的布局管理器。

单参数的add（）方法中的参数是用户要加入的组件。布局管理器FlowLayout类、GridLayout类和GridBagLayout类将同单参数的add( )方法一起使用。

双参数的add( )方法中的第二个参数指定了要加入的组件。第一个参数是依赖于布局管理器的字符串：BorderLayout类（缺省的布局管理程序）需要通过指定“North”“South”“East”“West”或“Center”字符串来指出组件放置的位置，CardLayout类需要用户写出标

识被加入的组件的字符串。

2）类提供的功能

除了菜单以外的所有组件都是作为Component类的子类来实现的。从Component类，它们继承了大量的功能，Component类为实现所有的绘画功能及事件处理提供了基础。下面将详细地列出Component类所提供的功能：

（1）支持基本的绘画功能。Component类提供了paint（）、update（）和repaint（）方法，这些方法使组件能够在屏幕上将自己画出来。

（2）事件处理。Component类定义了handleEvent（）方法等一组方便的方法，例如：action（）方法，它用来处理指定类型的事件，Component类也提供了设置并获得键盘光标，使键盘能够对组件进行控制等一些方法。

（3）对显示字体的控制。Component类提供了用于获得和设置当前字体的方法，以及获得当前字体信息的方法。

（4）对显示颜色的控制。Component类提供了用于获得和设置前景颜色和背景颜色的方法。

（5）图像处理。Component类提供了对ImageObserver接口的执行并定义了帮助组件显示图像的方法。要注意的是，大部分组件不能显示图像，因为它们外观的显示是以执行特定平台的代码获得的。而组件Canvas和容器可以显示图像。

（6）组件在屏幕上的显示尺寸和位置的控制。所有组件的显示尺寸和位置是由布局管理器控制的。组件本身也提供了一些方法改变组件的尺寸，将组件放到适当的位置以及报告组件的合适的和最小尺寸。组件还提供了一些方法用于返回关于组件当前尺寸和位置的信息。

3）如何改变组件的外观和行为

大多数组件的外观取决于它们运行的平台。除了由组件的类和其超类所提供的影响组件外观的方法和变量以外，用户很难改变组件的外观，尤其是不能靠创建某个组件类的子类来改变组件的外观。要改变组件的外观，必须继承Canvas类，然后让它具有自己所要的外观，具有用户所期望的行为。

尽管很难对组件的外观作较大的改变，但可以改变组件的行为。例如，如果想在文本域中只有数字才有效的话，他可以继承TextField类，然后在这个子类中检测所有键盘事件，并只截获有效的事件（即按下数字键所产生的事件）。

1.6.2.2 如何使用AWT组件

下面我们将逐一介绍如何使用AWT所提供的组件：

1）如何使用按钮

按钮是简单的UI组件。在用户接口中可以利用它被按下时激发一些动作。例如一个计算器applet会有许多按钮用于数字和操作符。又如：一个对话框会有“ok”“Cancel”按钮。要创建按钮，可以使用以下构造函数：

（1）Button（）方法创建了一个不带卷标的按钮。

（2）Button（String）创建了一个带有以参数string为卷标的按钮。

一旦创建并拥有了按钮对象，那么就可以用getLabel（）方法获得按钮的卷标，setLabel（string）方法设置按钮的卷标。

下面的代码创建了3个按钮：

```
bl=new Button ( );
b1. setLabel("Button1");
b2 = new Button ( "Button2" );
b3 = new ButtOn("Button3");
```

Button类提供了一个缺省的按钮工具。按钮的外观取决于他们正在运行的平台。如果用户想使其程序中的按钮在任何平台上看上去相同或有一个特殊的外观，他只能通过创建一个Canvas子类来实现。在使用Button类的子类时，不能改变其外观，只能改变按钮显示的文本字体及其前景和背景颜色。

2）如何使用画布(Canvas)

Java类是用于被继承的。它本身并不做任何事情，而仅仅是实现一个特定的组件提供方法。例如：画布常作为图像或图形的显示区域使用。

```
import Java.awt.*;
public class ImageSquence extends Applet implements Runnable{
int frameNumber;
int delay;
Thread animatorTread;
Dimension offDimension;
Image offImage;
Graphics offGraphics;
Image images[ ];
public void init( ){
        String str;
        str=getParameter("fps");
        int fps=(str!=null)?Inter.parseInt(str):10;
        delay=(fps>0)?(100/fps):100;
        images=new Image[10];
        for(int i=1;i<=10;i++){
            images[i-1]=getImage(getCodeBase( ),../../../images/duke/T"+i+".gif");
        }
}
  public void start( ){
    if (animatorThread==null){
      animatorThread=new Thread(this);
      animatorThread.start( );
     }
    public void stop( ){
      animatorThread=null;
      offImage=null;
```

```
            offGraphics=null;
          }
        public boolean mouseDown(Event e,int x,int y){
              if(animatorThread==null){
                   start( );
              }
              else{
                 stop( );
              }
          return false;
        }
      public void run( ){
          long startTime=System.currentTimeMillis( );
          while(Thread.currentThread( )==animatorThread){
              //Display the next frame of animation.
              repaint( );
              try{
                 startTime+=delay;
                 Thread.sleep(Math.max(0,startTime-System.currentTimemillis( )));
                 }
                 catch(InterruptesException e){
                       break;
                 }
                 frameNumber++;
              }
           }
      public void paint(Graphics g){
            if(offImage!=null){
              g.drawImage(offImage,0,this);
              }
           }
      public void update(Graphics g){
            Dimension d=size( );

if((offGraphics==null)||(d.width!=offDimension.width)||(d.height!=offDimension.height)){
    offDimension=d;
    offImage=createImage(d.width,d.height);
    offGraphics=offImage.getGraphics( );
  }
```

```
offGraphics.setColor(getBackground( ));
offGraphics.fillRect(0,0,d.width,d.height);
offGraphics.setColor(Color.black);
offGraphics.drawImage(images[frameNumber%10],0,0,this);
g.drawImage(offImage,0,this);
  }
}
```

当需要一个组件，例如一个按钮，而且想要让它外观看上去与组件的缺省实现不一样时，就要用到画布。由于不能靠继承相应组件的类来改变标准组件的外观，因此必须通过继承 Canvas 类来使其既具有用户所希望的外观，又同标准组件的缺省实现具有同样的行为。

继承 Canvas 类时，应注意用 minimumSize ( ) 方法和 preferredSize ( ) 方法来恰当地反应出其画布的尺寸。否则，依靠画布所在容器使用的布局，画布会变得很小，甚至无法看见。

下面是一个 Canvas 子类的例子，它显示了一个图像：

```
class ImageCanvas extends Canvas{
      ImageCanvas(Image img,Dimension prefsize){
      image=img;
      preferredsize=prefsize;
  }
  public Dimension minimumsize( ){
       return preferredsize;
       }
  public Dimension preferredsize( ){
       return preferrsize;
  }
public void paint(Graphics g0{
  g.drawImage(image,0,0,this);
  }
}
```

3）如何使用检查框

检查框是一种用户接口组件。它具有两种状态：打开和关闭（或被检查和未被检查；或被选中和未被选中；真值和假值等）。不像按钮，检查框在 UI 中通常不激发直接动作，而是被用做指示一些动作可选择的特点。

检查框可以两种方式使用：

（1）非单一方式。这意味着在给定的一系列检查框中，它们中任何一个都可被选中。

（2）单一方式。这意味着在给定的一系列检查框中，在同一时刻，它们中只有一个能被选中。

单一方式的检查框叫做检查框组，或叫做 Radio 按钮。

非单一方式的检查框可以通过 Checkbox 类创建。可以用下列构造函数之一创建检查框：

- Checkbox( )方法创建一个空的，非选中的检查框。
- Checkbox(string)方法创建一个以给定字符串作为卷标的检查框。
- Checkbox (string, null, boolean) 方法创建一个根据布尔型参数是真值还是假值来决定其被选中还是非选中的检查框。

下列代码创建了几个检查框：

```
Checkbox cb1,cb2,cb3;
cb1=new Checkbox( );
cb1.setLabel("Checkbox1");
cb2=new Checkbox("Checkbox2");
cb3=new Checkbox("Checkbox3");
cb3.setState(true);
```

检查框组同检查框不太一样。它们具有相同的外观，但检查框组中在同一时刻，只有一个检查框能被选中。要创建检查框组，首先应创建一个 CheckboxGroup 类的实例：

```
CheckboxGroup cbg = new CheckboxGroup( ) ;
```

然后创建并加入单个检查框。这要将检查框组的实例作为第二个参数，并以真值或假值作为第三个参数（真值或假值决定了检查框是否被选中，但只能有一个检查框被选中）。

```
add(new Checkbox("yes", cbg, true));
add(new Checkbox("no",cgb,true));
```

下面是一个简单的例子：

```
CheckboxGroup cbg=new CheckboxGroup ( );
add(new Checkbox("Red",cbg,false));
add(new Checkbox("Yellow",cbg,false));
add(new Checkbox("Green",cbg,false));
add(new Checkbox("orange",cbg,true));
add(new Checkbox("purple",cbg,false));
```

前面表中列出的方法，可以用于检查框组中的检查框。另外，可以使用 getCheckboxGroup( )和 setCheckboxGroup ( ) 方法来访问和改变任何给定检查框的组。

在检查框组中定义的 getCurrent ( ) 和 setCurrent (Checkbox) 方法，可以用来获得或设置当前被选中的检查框。

4）如何使用选择框（Choice）

选择框是一个弹出式（或下拉式）菜单，它允许从菜单中选中一项，然后将选中的菜单项在屏幕上显示。

要创建选择框，应创建 Choice 类的一个实例。然后用 addItem( )方法将每个选项加入。选项的显示顺序同加入的顺序一样。

```
Choice C=new Choice( );
C.addItem("Apples");
```

```
C.dddItem("Oranges");
C.addItem("Bananas");
```

最后，经常使用的方法是将整个选择框加到面板（panel）上。

```
add(C);
```

一旦选择框被创建，无论它是否被加到面板上，可以继续用addItem（）方法向其中加入选择。

5）如何使用标签(label)

标签是用来标注其他UI组件的有效的文本字符串，它优于一般的文本字符串的方面在于它遵循给定面板的布局，而且在每次面板重画时，不用去重画标签。在面板中，很容易对标签进行排放，而且可以将标签附加给其他的UI组件，并不需要知道具体的像素位置。

可以用下列构造函数之一创建标签：

（1）Label（）方法创建一个空的标签，它的文本被安排在左侧。

（2）Label（String）创建了一个具有给定文本字符串的标签，它的文本也被安排在左侧。

（3）Label（string，int）创建了一个具有给定字符串的标签，它的文本被安排在给定位置。标签中可用的放置位置被存储在类变量中，它们很容易记：Label.RIGHT,Label.LEFT和Label.CENTER。

标签的字体由用于组件的全局字体决定。

下面是创建几个标签的简单例子：

```
add （new Label（"aligned left"））;
add（new Label("aligned center",Label.CENTER));
add（new Label("aligned right",Label.RIGHT));
```

一旦用户拥有了一个标签对象，那么就可以使用类中定义的方法获得或设置标签的文本。

6）如何使用列表（List）

列表框功能上与选择框（Choice）很相似，只是在列表框中允许在同一时间内从中选择多项选择项。列表框与选择框有两个重要的区别：

（1）列表框不是弹出式菜单。它只是选项的列表，其中可以选择一项或多项。如果选项的数目超出列表框的范围，滚动条会自动生成，以使用户可以看见其他的选项。

（2）列表框可以定义成在同一时刻只接受一个选项（单一方式）或在同一时刻可接受多个选项（非单一方式）。

要创建列表框，首先应创建List类的一个实例，然后再将选项逐个加入。List类有两个构造函数：

- List（）方法创建了一个空的列表框。它在同一时刻只允许一个选择。
- List（int,boolean）方法创建了一个具有给定数目可见行的列表框。布尔型参数指示出此列表框是否允许多项选择（true表示允许，false表示不允许）。

创建了一个列表框对象后，可以用addItem（）方法将选项加入，然后将列表框加到包含它的面板中。下面是一个例子：

```
List lst = new list ( 5,  true );
lst.addItem1 ( "Hamlet" );
lst.addItem ( "claudius" );
lst.addItem("Gertrude");
lst.addItem("Polonius");
lst.addItem("Laertes");
lst.addItem("Laertes");
lst.addItem("ophelia");
add(lst);
```

7）如何使用单行文本域(TextField)

单行文本域允许用户在单行范围内输入编辑文本。

可以用下列构造函数之一创建单行文本域：

（1）TextField 方法创建一个不含任何字符的空单行文本域。

（2）TextField(int)方法创建一个以字符为单位指定宽度的空单行文本域。

（3）TextField(string)方法创建一个具有指定初始化字符串 0 字符宽度的单行文本域。

（4）TextField(String,int)方法创建一个包含指定初始化字符串，并具有指定宽度的单行文本域。

例如：下列代码创建了一个具有多个字符、并以 "Enter your name" 为初始化内容的单行文本域。

```
TextField tf=new TextField("Enter your name",30);
add(tf);
```

注:单行文本域只提供了一个可编辑文本的区域。因此，它通常需要一个标签来指示其内容。单行文本域不同于多行文本域。它在尺寸上有限制，常用于单行文本的情况。而多行文本域具有滚动条，它适合于较大的文本窗口。无论单行文本域还是多行文本域，它们都是可编辑的，可选择的。

也可以创建一个可以隐藏被敲入字符串的单行文本域，例如：口令域等。要做到这一点，首先应创建一个单行文本域，然后用setEchoCharacter( )方法设置在屏幕上重复显示的字符(此重复显示的字符代替了用户敲入的字符在屏幕上显示)。见下例:

```
TextField tf=new TextField(30);
tf.setEchoCharacter('*');
```

单行文本域继承了 TextComponent 类并拥有一整套可应用的方法。这些方法有继承 TextComponent 类的，有 TextField 类自己定义的。它们对于编写 Java 程序都十分有用。

8）如何使用多行文本域(TextArea)

多行文本域除了具有更多的功能处理大量文本以外，它同单行文本域（TextField）是一样的。单行文本域在尺寸上有限制而且不能卷动，因此它最好用于单行文本；多行文本域可以指定其高度和宽度，而且具有滚动棒，因此它可以处理较大数量的文本。

可以用下列构造函数之一创建多行文本域：

（1）TextArea（ ）方法创建了一个 0 行长，0 列宽的空多行文本域。

（2）TextArea（int，int）方法创建了一个具有给定行数，给定列数的空多行文本域。

（3）TextArea(String)方法创建了一个具有给定字符串的多行文本域。

（4）TextArea(String,int,int）创建了一个具有给定字符串，具有给定行数、列数的多行文本域。

下面是一个简单的例子：

```
String str= ("once upon a midnight dreary, while I pondered");
add （new TextArea（str,10,60））;
```

因为TextField类和TextArea类都继承了TextComponent类，所以许多用于单行文本域的方法，对于多行文本域也同样适用。当然，多行文本域也拥有自己特有的方法。

9）如何使用滚动棒（Scrollbar）

多行文本域和列表框都有自己的滚动棒。滚动棒已被做到这些UI组件中，成为它们的一部分。也可以创建自己的滚动棒。滚动棒通常用在一个最大值和最小值之间。

（1）用滚动棒中间的滚动区域，可以较大地增加或减少滚动棒的值（缺省为加10或减10）。

（2）滚动棒中有个滚动块，它的位置表示出滚动棒当前值在其最大值、最小值范围内所处的位置。用鼠标移动滚动块，可以改变滚动棒的值。

（3）有了滚动棒之后，只需要给出最大值和最小值，使用滚动棒中可以改变滚动棒的值的部分。不需要更新任何事情、处理任何事件。因为剩下的工作由Java为用户完成。

用户可以用下列构造函数之一创建滚动棒：

（4）Scrollbar（）方法创建了一个垂直方向，最大值和最小值为0的滚动棒。

（5）Scrollbar（int）方法创建了一个最大值和最小值为0的滚动条。参数表示了一个方向，可以使用类变量Scrollbar.HoRizoNTAL和Scrollbar.vERTicAL来表示。

（6）Scrollbar（int，int，int，int）方法创建了一个具有下列参数的滚动棒（每个参数是整型，而且必须按下面提供的顺序给出）：

- 第一个参数指定了滚动棒的方向。可以用类参数Scrollbar.HORIZONTAL和Scrollbar.VERTICAL指定。
- 第二个参数是滚动棒的值。此值必须在滚动棒的最大值和最小值之间。
- 第三个参数是滚动块的总宽度（或总高度，这要根据滚动棒的方向）。在用户接口设计中，滚动块越大表示现在显示出的内容占全部内容范围的比例越大。

下面是一个滚动棒增值的例子。滚动棒左侧的标签，在每次滚动棒的值改变时被更新：

```
import Java.awt.*;
public  class slidetest extends Java.applet.Applet{
        Label l;
public void init( ) {
 l=new Label("0");
 add(l);
 add(new Scrollbar(Scrollbar.HORIZONTAL,1,0,1,100));
          }
public boolean handleEvent(Event evt){
```

```
        if(evt.target instance  of Scrollbar){
          int V=((Scrollbar)evt.target).getvalue();
          l.setText(String.valueof(v));
         }
         return true;
        }
       }
```

10）如何使用面板（Panel）

面板是能在屏幕上实际显示的组件。面板继承了Container类，它提供了容纳其他组件的功能。Applet类是Panel类的一个子类。要在一个applet中加入其他面板，只能创建一个新的applet，并把它加到applet中，就像用户加入其他UI组件：

```
SetLayout（new Gridiayout（1,2,10,10））;
Panel Panel1 = new Panel（ ）;
Panel Panel2 = new Panel（ ）;
add（Panel1）;
add（Panel2）;
```

可以为那些子面板建立独立的布局，并通过调用add()方法将AWT组件加到它们中去（它们中仍可包含子面板）。

```
Panel1. SetLayout（FlowLayout（ ））;
Panel1. add （new Button（"Up"））;
Panel1. add （new Button（"Down"））;
```

11）如何使用框架（Frame）

AWT的Window类使用户可以创建独立于包含applet的测览器窗口的窗口。

Window类提供了用于窗口的一些基本功能。通常，使用Window类的子类Frame类和Dialog类。Frame类使用户可以创建带有菜单条的全功能窗口。Dialog类用于创建对话框。

可以用下列构造函数之一创建框架：

（1）new Frame( )创建了一个不带标题的框架。

（2）new Frame（String）创建了一个带有指定标题的框架。

框架是容器，就像面板一样，可以用add()方法将其他组件加到其中。用于窗口的缺省布局是BorderLayout：

```
win = new Frame（"My cool window"）;
win.SetLayout(new BorderLayout(10,20));
win.add("North",new Button("start"));
win.add("Center",new Button("Move"));
```

要设置新窗口的尺寸，用resize（）方法。要设置窗口的显示位置，用move（）方法。用location（）方法、可以知道applet窗口在屏幕上的位置。

```
win.resize(100,200);
```

```
Dimension d=location();
win.move(d.width+50,d.height+50);
```

当创建一个窗口时，窗口是不可见的。需要用show()方法将窗口在屏幕上显示出来（还可以用 hide（）方法将其隐藏）。

```
win.show（）;
```

12）如何使用菜单(Menu)

用户创建的每个新的窗口都能有它们自己的菜单条。每个菜单条可以有一定数量的菜单。菜单又可以有自己的菜单项。AWT提供了一些类用于这些组件。它们分别是MenuBar类、Menu类和MenuItem类。

（1）菜单和菜单条。要为指定窗口创建一个菜单条，首先应创建MenuBar类的一个实例：

```
MenuBar mb=new MenuBar();
```

然后可以用 setMenuBar（）方法将这个菜单条设置成窗口的缺省菜单：

```
Window.setMenuBar(mb);
```

最后，可以创建菜单并把它们加到菜单条上：

```
Menu m = new Menu（"My"）;
mb.add(m);
```

一些系统允许指定一个特定的帮助菜单，这个菜单将被放置在菜单条的最右侧。setHelpMenu（）方法可以使用户达到目的。但要指定为帮助菜单的菜单，必须已经加到菜单条上了。这是一个先决条件。

```
Menu hm = new Menu（"help"）;
mb.add（hm）;
mb.setHelpMenu（hm）;
```

如果出于某些原因，想阻止其他人选择菜单，那么他可以用disable（）方法使其无效（用enable（）方法可以使其重新有效）。

```
m.disable（）;
```

（2）菜单项。有四种类型菜单项用户可以将其加到菜单中去：

- MenuItem类的实例，用于常规的菜单项；
- CheckBoxMenuItem类的实例，用于触发式菜单项；
- 带有菜单项的其他菜单；
- 分割线，用于将菜单中的菜单项分成组。

常规菜单项可以 add( )方法加入：

```
Menu m = new Menu（"Tools"）;
m.add（new MenuItem（"Info"））;
m.add （new MenuItem（"Colors"））;
```

子菜单可以通过创建 Menu 类的一个新的实例创建，然后将它们加到其他菜单中。

```
Menu sb = new Menu（"sizes"）;
m.add(sb);
sb.add(new MenuItem("Small"));
```

```
sb.add(new MenuItem("Medium"));
sb.add(new MenuItem("Large"));
```

CheckBoxMenuItem类创建了一个带有检查框的菜单项，并使菜单可以处于被打开和关闭状态（选中检查框一次，使其处于被选中状态；再选择它一次，则使其处于非选中状态）。创建和加入检查框菜单项的方法同常规菜单项一样：

```
CheckBoxMenuItem coords=new CheckBoxMenuItem ( "show coordinates" );
m.add ( coords );
```

可以用下面的代码将分割线加到菜单中：

```
MenuItem msep = new MenuItem ( "_" )
m.add(msep);
```

任何菜单项都可以用disable（）方法使菜单项无效，用enable（）方法使其有效。无效的菜单项是不能被选中的。

```
MenuItem mi = new MenuItem ( "Fill" );
mi.disable();
```

选中菜单项会生成一个动作事件。可以重载action（）方法来处理这些动作事件。选中常规菜单项和检查框菜单项都会生成一个特殊的参数，这个参数代表了用于那个菜单的标签。可以用这个标签来决定该做什么。注意，因为CheckBoxMenuItem类是MenuItem的子类，因此可以不对检查框菜单项特殊对待。

```
public boolean action ( Event evt, Object arg )
{
  if(evt.target instance of menuItem){
    String label=(String)arg;
  if(Label.equals("Show Coordinates")toggleCoords( );
  else if(label.equals("Fill"))fillcurrentArea( );
   return true;
  }
  else return false;
}
```

13）如何使用对话框（Diaiog Boxes）

对话框功能上与框架（Frame）很相似，它们都在屏幕上弹出新窗口。但对话框用于短暂的窗口，例如：提示警告信息的窗口；向用户询问特定信息的窗口等等。通常对话框没有标题条或窗口所具有的许多特点。用户可以使它们不能改变尺寸或使它们模式化。

有模式对话框在消失以前，不允许屏幕上的其他窗口进行输入工作。

AWT提供了两种对话框：Dialog类可以生成普通对话框；FileDialog类可以生成依赖于平台的，用于选择存储或打开文件的对话框。

可以用下列构造函数之一创建普通对话框：

（1）Dialog(Frame，boolean）方法创建了一个起初看不见的对话框，这个对话框附属于当前框架。布尔值决定了此对话框是否为有模式对话框（真值为有模式，假值为无模式）。

（2）Dialog（Frame，String,boolean）方法功能上同前面的方法，只不过此方法创建的对话框带有标题条和字符串参数所示的标题。

像框架窗口一样，对话框窗口也是一个面板。在它的上面，可以画UI组件，安排组件布局，进行图形处理。对话框像其他窗口一样，起初创建时是不可见的，要用show（）方法使其可见或用hide（）方法再将其隐藏。

```
Dialog d=new Dialog(this,"Evter Text",true);
  d.setLayout(new GridLayout(2.1.30,30));
  tf=new TextField();
  d.add(tf);
  d.add(new Button("ok"));
  d.resize(150,75);
  d.show();
```

FileDialog类提供了基本的文件打开／存储对话框。从而很方便地访问文件系统。FileDialog类独立于系统，但不独立于平台。

可以用下列构造函数之一创建文件对话框：

● FileDialog（Frame，String）方法创建了一个打开文件的对话框，此文件对话框附属于给定的框架，具有给定的标题。

● FileDialog（Frame，String,int）方法同上面的方法功能基本一样。整型参数用来决定此文件对话框是用来打开文件还是保存文件。可选择的类型参数为FileDialog.LOAD和FileDialog.SAVE。

在创建了文件对话框实例之后，用户必须用show（）方法将其显示出来：

```
FileDialog fd=new  FileDialog(this，"FileDialog");
fd.show（）;
```

1.6.3 组件在容器中的布局

本部分将向读者介绍如何使用AWT提供的布局管理器；介绍如何编写用户自己的布局管理器。

1.6.3.1 使用布局管理器

在缺省情况下，每个容器中都有布局管理器。如果容器的缺省布局管理器不能满足用户的需要，用户可以使用别的布局管理器来代替缺省的。AWT提供了五种布局管理器：FlowLayout,GridLayout,BorderLayout,CardLayout和GridBagLayout。下面我们介绍管理器的基本规则以及各种布局管理器的特点和使用方法。

1）使用布局管理器的基本规则

如果用户不指定容器所使用的布局管理器，那么容器将使用它们自己缺省的布局管理器。每当容器需要改变外观时，这些缺省的布局管理器都会发挥作用。大多数布局管理器的方法都不需要程序直接调用。

（1）如何选择布局管理器

AWT提供的布局管理器各有各的长处和缺点。下面将讨论几种方案及适合每种方案的AWT布局管理器。

方案1：用户需要尽其所有的空间来显示组件。

用户可以考虑使用BorderLayout和GridBaglayout。如果使用BorderLayout，用户应该将占用空间最大的组件放在中心部位。如果使用GridLayout，用户需要为组件设置限制条件。如果用户不想让同一容器中的其他组件与占用空间最大的组件一样大，则可以用GridLayout。

方案2：如果用户需要在紧凑的一行中以组件的自然尺寸显示较少组件时，用户可以考虑用面板容纳组件，并使用面板的缺省布局管理器：FlowLayout。

方案3：用户需要在多行或多列中显示一些同样尺寸的组件。GridLayout最适合此情况。如果有必要的话，可以使用面板来容纳组件。

（2）如何创建布局管理器

每个容器都有与其相关的缺省的布局管理器。所有面板（包括Applets）的缺省布局管理是FlowLayout。所有窗口（除了一些特殊目的窗口，像FileDialog）的缺省布局管理器是BorderLayout。

如果用户想使用容器的缺省布局管理器，那么用户不需要做任何事情。因为容器的构造函数创建了布局管理程序的实例并将容器初始化为使用此实例。

如果用户不想使用容器缺省的布局管理器，那么用户需要创建他想使用的布局管理器的实例并让容器使用此实例。下面的代码创建了CardLayout布局管理器，并将它设置为容器的布局管理器。

```
Container.setLayout(new CardLayout( ));
```

2）如何使用BorderLayout

BorderLayout有五个区域：北、南、东、西和中心。如果用户扩大窗口，用户将会看到中心区域会尽可能的扩大，而其他区域只扩大到容纳组件所需空间的大小。

下面的例子创建了BorderLayout布局管理器及它所管理的组件：

```
setLayout(new BorderLayout( ));
setFont(new Font("helvetica",Font,PLAIN,14));
add("North",new Button("North"));
add("East",new Button("East"));
add("West",new Button("West"));
add ( "Center",  new  Button ( "Center" ) );
```

当用户向使用BouderLayout的容器中加入组件时，用户必须使用两个参数的add()方法，而且第一个参数必须为“North”，“South”，“East”，“West”或“Center”。如果用户使用一个参数的add()方法或指定的第一个参数无效，那么该组件将不能显示出来。

在缺省情况下，BorderLayout将使它所管理的组件之间没有空隙，用户可以用下面的构造函数指定间隙（以像素为单位）：

```
public  Borderlayout ( int  horizontalGap,  int  verticalGap )
```

3）如何使用CardLayout

CardLayout类可使用户管理共享同一显示空间的两个或更多个组件。

从概念上讲，CardLayout所管理的每个组件就像在纸盒里放的纸牌，在某一时刻只有最

上面一张可见，用户可以通过指定组件的名字或指定最后或第一个组件（组件的顺序按照它们被加入到容器中的顺序）来选择要显示的组件。

下面的代码创建了 CarLayout 布局管理器及它所管理的组件：

```
Panel cards;
final static string BUTTONPANEL="Panel With Buttons";
final static string TEXTPANEL="Panel With TextField";
Cards=new Panel( );
Cards.setLayout(new CardLayout( ));
Panel P1=new Panel( );
P1.add(new Button("Button1");
p1.add(new Button("Button2");
p1.add(new Button("Button3");
Panel P2=new Panel( );
P2.add(new TextField("TextField",20));
 Cards.add(BUTTONPANEL,p1);
 Cards.add(TEXTPANEL,P2);
```

当向 CardLayout 所管理的容器中加入组件时，必须用带两个参数的 add（）方法：add（String name，Component comp）。第一个参数是标识加入组件的字符串，第二个参数是要加入的组件。

为了选择出 CardLayout 显示的组件，需要再加入一些代码：

```
.....
Panel cp = new Panel;
Choice C = new Choice ( );
C.addItem （BUTTONPANEL );
C.addItem ( TEXTPANEL );
cp.add(C);
add ( "North", cp );
public boolean action （Event evt, Object arg )
 {
   if ( evt.target instanceof Choice )
        {
    ((Cardlayout)Cards.getLayout)).show （Cards, ( String ) arg );
                  return true;
        }
       return false;
   }
```

正如上面例子所示，可以用 CardLayout 的 show（）方法来设置当前要显示的组件。show()方法中的第一个参数是 CardLayout 所控制的容器（换句话说，容器中的组件排

放由 CardLayout 管理）。第二个参数是标识要显示组件的字符串。

下面列出的是CardLayout所提供的，允许选择组件的方法。对于每个方法，第一个参数是由 CardLayout 控制的容器。

```
public void first （Container parent）
public void next （Container parent）
public void previous （Container parent）
public void last （Container parent）
public void show(Container parent,String name)
```

4）如何使用 FlowLayout

FlowLayout将组件以它们适合的尺寸放在一行中，如果容器中的水平空间对于将所有组件放在一行中太小，那么FlowLayout将用多行显示。每一行组件将根据FlowLayout创建时 alignment 参数的指定放在屏幕的中心位置（缺省）左侧或右侧。

下面的代码创建了 FlowLayout 及它所管理的组件。

```
setLayout(new FlowLayout());
setFont(new Font("Helvetica",Font.PLAIN,14));
add(new Button("Button1"));
add(new Button("Button2"));
add(new Button("Button3"));
add(new Button("Long-Named Button4"));
add(new Button("Button5"));
```

FlowLayout 类有三个构造函数：

```
public FlowLayout()
public FlowLayout(int alignment)
public FlowLayout(int alignment,int horizontalGap,int VerticalGap)
```

lignment 参数的值必须是 FlowLayout.LEFT,Flowlayout.CENTER 或 FlowLayout.RIGHT。horizontalGap 和 verticalGap 参数指定了组件间隔距离(以像素为单位)。如果没有指定间隔值，FlowLayout 将自动指定其为 5。

5）如何使用 GridLayout

GridLayout 将组件放置在格栅的单元空间中。每个组件占据其单元空间的所有空间，而且每个单元空间具有相同的尺寸。如果改变GridLayout所有管理的窗口的尺寸，用户将会看到 GridLayout 改变了单元空间的尺寸，使它们在容器的可用空间中尽可能的大。

下面的代码创建了 GridLayout 和它所管理的组件。

```
setLayout(new GridLayout(0, 2));
setFont(new Font("Helvetica", Font. PLAIN,14));
add(new Button("Button1"));
add(new Button("2"));
add(new Button("Button3"));
add(new Button("long - Named Button4"));
```

```
add ( new Button ( "Button5" ));
```

GridLayout 类有两个构造函数，下面是它们的定义：

```
public GridLayout( int rows, int columns )
public GridLayout(int rows, int cloumns, int horizontalGap, int verticalGap)
```

第一个构造函数的参数指定了行数和列数，它们中必须至少有一个为非零值。第二个构造函数中的 horizontalGap 和 verticalGap 参数指定了单元空间之间的间隔（以像素为单位）。如果用户不指定间隔值，它们的缺省值为 0。

6）如何使用 GridBagLayout

GridBagLayout 是 AWT 提供的最灵活、最复杂的布局管理程序。GridBagLayout 将组件以多行多列放置，允许指定的组件跨多行或多列。如果用户增大 applet 的窗口，用户会发现最后一行会占有所有新的垂直方向的空间，所有新的水平方向的空间被所有的列分割。这个改变尺寸的动作是以 applet 分配给 GridBagLayout 中单个组件的权为依据的。用户还会看到每个组件会占据尽可能多的空间。这个动作也是由 applet 指定的。

applet 是通过为每个组件指定限制因素（constraint）来指定组件的尺寸和位置特点的。要指定限制因素，用户在 GridBagConstraints 对象中设定实例变量，并用 setConstraints（）方法告诉 GridBagLayout 组件相关的限制因素。

下面我们将介绍用户可以设定的限制因素并提供一个例子。

（1）指定限制因素。下面是使用 GridBagLayout 的容器中常见的典型代码：

```
GridBagLayout gridbag=new GridBagLayout();
GridBagConstraints C=new GridBagConstraints();
setLayout(gridbag);
... 创建组件
... 设置 GridBagConstraints 实例中的实例变量
    gridbag.setConstraints(theComponent,C);
    add(theComponent);
```

多种组件可以使用相同的 GridBagConstraints 实例，甚至于具有不同的限制因素的组件也可以使用相同的 GridBagConstraints 实例，GridBagLayout 引用了限制因素的值，就不再使用 GridBagConstraints。对此用户要十分小心，必要时要将实例变量重新设置成它们的缺省值。

用户可以设置下列 GridBagConstraints 实例变量：

```
gridx,gridy
```

指定了组件显示区域左上角处的行值和列值。最左侧一列为 gridx=0，最上面一行为 gridy=0。用户可以使用缺省值 GridBagConstraints.RELATIVE 来将此组件放置在事先已加入到容器中的组件的右侧（对于 Gridx 来说）或下侧（对于 Gridy 来说）。

```
gridwidth,gridheight
```

指定了组件显示区域的列数（对于 GridWidth 来说）或行数（对于 GridHeight 来说）。它们的缺省值是 1。用户可以用 GridBagConstraints.REMAINDER 将组件放在所在行最后一个组件位置或所在列最后一个位置。用户还可以用 GridBagConstraints.RELATIVE 值将组件放置在相邻于所在行或所在列最后一个组件的位置。

此参数在组件的显示区域大于组件所需尺寸时用来决定是否改变组件的尺寸及如何改变组件的尺寸。此参数的有效值为：GridBagConstraints.NONE（缺省）、GridBagConstraints.HORIZONTAL（使组件足够宽，从而填充其显示区域的全部水平空间，但不改变组件的高度）、GridBagConstraints.VERTICAL 使组件足够高，从而填充其显示区域的全部垂直空间，但不改变组件的宽度）、GridBagConstraints.BOTH（使组件填充其全部显示区域）。

参数 ipadx,ipady 指定了在组件最小尺寸的基础上的增加量。其缺省值为 0。组件的宽度至少为其最小宽度加上 ipadx × 2 个像素。同样，组件的高度至少为其最小高度加上 ipady × 2 个像素

insets 指定了组件和其显示区域边界之间的空间大小。Insets 对象决定了 insets 值。

anchor参数在组件小于其显示区域时用于决定在显示区域何处放置此组件。有效值可以是：GridBagConstraints.CENTER(缺省),GridBagConstraints.NORTH,GridBagConstraints.NORTHEAST, GridBagConstraints.EAST, GridBagConstraints.SOUTHEAST, GridBagConstraints.SOUTH. GridBagConstraints.SOUTHWEST, GridBagConstraints.WEST, GridBagConstraints. NORTHWEST。

weightx,weighty这两个参数所指定的权值明显地影响着GriddBagLayout所控制组件的外观。权值用于决定在列与列之间和行与行之间如何分配空间。

除非在 weightx 和 weighty 两者之中至少指定了一个为非 0 值，所有的组件将集中在一起，处于容器的中央部位。这是因为当权值为 0 时（缺省），GridBagLayout 将任何额外的空间合并到单元空间和容器边缘之间的空间中去。

通常权值被指定为 0.0 和 1.0，1.0 作为最大值；或者指定为在 0.0 和 1.0 之间所需要用的值。较大的数值意味着组件的行或列应获得较大的空间。对于每一列，达到权值等于那列组件指定的最大weightx。同样，每行的权值等于为那行中的组件指定的最大额外空间都归并到最右一列和最下面一行中。

（2）一个典型例子的分析。下面的代码创建了 GridBagLayout 和它所管理的组件。

```
protected void makebutton(String name,GridBagLayout gridbag,GridBagConstraints C){
                Button button=new Button(name);
                gridbag.setConstraints(button,C);
                add(button);
                }
public GridBagWindow(){
                GridBagLayout gridbag=new GridBagLayout();
                GridBagContraints C=new GridBagConstraints();
                setFont(new font("Helvetica",Font.PLAIN,14));
                setLayout(gridbag);
                C.fill=GridBagConstraints.BOTH;
                C.Weightx=1.0;
                makebutton("Button1",gridbag,C);
                makebutton("Button2",gridbag,C);
```

```
makebutton("Button3",gridbag,C);
C.gridwidth=GridBagConstraints.REMAINDER;//end row
makebutton("Button4",gridbag,C);
C.weightx=0.0;
makebutton("Button5",gridbag,C);
C.gridwidth=GridBagConstraints.RELATIVE;//next-to-last in row
makebutton("Button6",gridbag,C);
C.gridwidth=GridBagConstraints.REMAINDER;//end row
makebutton("Button7",gridbag,C);
C.gridwidth=1;
C.gridheight=2;
C.weighty=1.0;
makebutton("Button8",gridbag,C);
C.weighty=0.0;
C.gridwidth=GridBagConstraints.REMAINDER;//END ROW
C.gridheight=1;
makebutton("Button9",gridbag,C);
makebutton("Button10",gridbag,C);
```

这个例子把一个GridBagConstraints实例用于gridBagLayout管理的所有组件。在每个组件加到容器以前，这些代码在GridBagConstraints对象中设置了合适的实例变量。然后，运用setConstraints（）方法为组件记录下它的所有限制因素的值。

比如，在将一组件放到一行末尾时，用户将用下列代码：

```
C.gridwidth=GridBagConstraints.REMAINDER;
```

为了便于分析，我们用一个显示了为每个组件设定的所有限制因素的值。缺省值用粗体显示。

容器中的所有的组件（Buttonl ~ Button9）在它们自己的行、列范围内都尽可能大的显示出来。这是因为程序中将GridBagConstraints的fill事件变量设置成了GridBagConstraints.BOTH。如果程序中不设置fill变量，所有的按钮将以它们自然的尺寸显示。

这个程序中有四个组件跨了多列（Button5，Button6，Button9和Buttonl0），有一个组件跨了多行（Button8）。程序中只明确地指定了一个组件（Buiton8）的宽度或高度。其他组件的宽度被指定为GridBagConstiaints.RELATIVE或GridBagConstiaints.REMAINDER。这样就是由GridBagLayout来决定组件的尺寸，而且要考虑本行中其他组件的尺寸。

显示窗口增大时，用户会发现各列的宽度也在增大，但总是保持相等。这是因为第一行中的每个组件（每个组件为一列宽）都有weightx=1.0的限制因素。这些组件的weightx的值并不重要，关键的是第一行中的所有组件具有相同的、大于0的权值。如果GridBagLayout所管理的组件没有设置weightx；那么当组件所在的容器扩大时，所有组件会聚集在一起，位于容器的中央部位。

当显示窗口增大时，用户还会发现只有最后一行的组件变高了。这是因为只有Button8

的weighty大于0。Button8占有2行高度，而且GridBagLayout恰巧将Button8的权值分配给了Button8所占有的最后一行。

1.6.3.2 创建特定的布局管理器

要创建特定的布局管理器，用户必须创建一个执行LayoutManager接口的类。同时，LayoutManager接口需要其附属的5个方法：

（1）public void addLayoutComponent（String name，Component comp）

这个方法由Containter类的add(name,component)方法调用。

（2）public void removeLayout Component（Component comp）

这个方法由Containter类的remove()和removeAll()方法调用。

（3）public Dimension preferredSize（Container parent）

这个方法由Container类的PreferredSize（）方法调用。

（4）public Dimension minimumLayoutSize（Container parent）

此方法由Container类的minimumSize（）方法调用。

（5）public void layoutContainer(Container parent)

当容器第一次显示时，调用此方法。在每次容器改变尺寸时，也调用此方法。一个布局管理器的LayoutContainer（）方法实际上并不能完成画组件的功能，它只是通过调用每个组件的resize（），move（）和reshape（）方法来设定组件的尺寸和位置。

除了以上五个方法以外，布局管理器还需要至少一个公用的构造函数和toString（）方法。

1.7 输 入 输 出

1.7.1 输入

Java语言在实现输入与输出的过程中，用到了流（stream）这个概念。所谓“流”，就是指数据通信通道。我们通常提到的输入输出，都是立足于编程者的角度上的，因此，所谓输入，就是将编程者所设计的程序看作流的终点；所谓输出，则是将程序看作流的起点。这一节中，将结合Java.io包中与输入输出有关的类与方法，对流的输入输出进行详细的阐述。Java语言的输入输出类也是网络编程的基础。

1.7.1.1 InputStream类

InputStream类是所有输入流类的祖先，它直接的下一级子类是ByteArrayInputStream类，FileInputStream类，FilterInputStream类，PipedInputStream类，SequenceInputStream类和StringBufferInputStream类。对于这些类，我们将在后面进行讨论。根据面向对象程序设计的原理以及类之间的继承关系，我们很容易想到InputStream类必然要对以上各子类共性的东西进行抽象和概括，进而规范出输入流的最基本的行为（即方法）。此外，InputStream作为一个抽象类，并不能直接对它进行实例化，但可以通过其子类的构造方法来生成一个输入流对象，进而引用它。例如：

```
InputStream object_InputStream=new BufferedInputStream
(new ByteArrayStream(byteArray))
```

其中byteArray是一个元素类型为byte的数组，于是，用户就可以以自身设计的程序为终点，向object_InputStream索取数据，而流的实际起点则是字节数组byteArray。

作为所有输入流的共同祖先，InputStream必然提供read方法，以便能够使用户从建立的InputStream流对象中读取数据。read方法本身，为了适应用户不同情况下的要求，又有三种形式，现分别举例说明如下：

（1）byte test_read;

test_read=(byte)object_InputStream.read();

这个read方法的作用是从建立好的输入流对象object_InputStream中读取一个字节的数据，而读取的内容作为返回值返回，于是此处的变量test_read中就存放了读入的字节数据。另外，由于这个read方法本身返回的数据为int型，而变量test_read为byte型，于是用到了强制类型转换。

（2）byte test_read[]=new byte[4];

int return_value;

return_value=object_ InputStream.read(test_read);

这个read方法，实现了一次输入一个字节数组的功能，输入的内容存放在字节数组test_read中。变量return_value中存放的是这次read操作中，实际读到字节数组test_read中的字节个数。另外，这个return_value变量也可能等于-1，等于-1则说明在执行这个read操作时，流中已经没有数据可供读取，也就是说已经到达了流的尾部。

（3）.byte test_read[]=new byte[8];

int offset=1;

int len=2;

int return_value;

return_value=object_ InputStream.read(test_read,offset,len);

在这个例子中，通过方法read，我们可以从object_ InputStream流对象中读取len(2)个字节的数据，并把它们存放在字节数组test_read中，另外读取的第一个字节数据存于test_read[offset]（即test_read[1]）中，读取的其他字节数据依次存放下去。关于返回值return_value，它的情形与第二种read方式中的返回值一样，这里就不再重述了。

需要注意的是，在这三种read方式运行时，如果无法从流中获取数据，就会导致程序流程的阻塞，并且一直阻塞到可读取需要的数据为止。

在进行输入工作时，往往并不需要流起点的全部内容，而只是对其中的某段后某几段内容感兴趣，于是为适应这方面的应用需要，这个所有输入流的共同祖先InputStream类提供给了用户一个skip方法，通过给定一个参数，来略过不需要的内容，从而实现了有选择地读取数据。例如流的起点是一个字节数组，并且在读第一个数据之前使用方法skip(3)，那么在skip之后读取的数据实际上是字节数组中的第四个元素，而头三个元素则被略过了。

在输入流InputStream类中，比较有特色的两个方法是mark和rest。通过mark方法的使用我们可以在输入流的当前位置上作一个标记，在通过在此之后的某个时刻使用rest方法，将输入点重新置到输入流最近一次mark标记的位置上，来实现对同一段内容的重新读取。但并不是每一个用于输入的类都支持mark和rest方法，为了保险起见，往往要在使用它们之前进行一下测试，这个测试由方法markSupport来完成。

在这一部分结合InputStream类，系统地介绍了输入流中的最基本的内容，在后续各部分，将对 InputStream 类的子类分别进行介绍、以建立各种类型的输入流。

1.7.1.2 ByteArrayInputStream 类

ByteArrayInputStream类可以建立起一个起点为某个字节数组的输入流，从而使用户可以通过这个流来读取字节数组中的内容。在这个类中，构造这个实体对象的方法有两种，即

1）ByteArrayInputStream（byteArray）

以指定字节数组 byteArray 作为流的起点来构造一个 ByteArrayInputStream 实体对象，也就是建立起一个输入流，用户通过这个输入流来读取 byteArray 中的内容。

2）ByteArrayInputStream（byteArray，offset,length）

在这个构造方法中，byteArray仍是所构造输入流的起点，但它并不是向流提供其全部内容，而是仅仅允许用户通过输入流读取其部分内容，即允许读取byteArray字节数组中从offset 下标起的 length 个字节的数据。

在这个ByteArrayInputStream输入流类中，除了构造方法以外，还提供了与其父类意义相同的skip方法和read方法，但read方法仅有上述所介绍的第一和第三种形式。Available 方法在这个类中的作用是告知用户还有多少个字节数据没有读取。另外，这个类并不支持 mark 方法，但却支持 rest 方法，它的作用是将输入点重新设置到输入流数据的字节数组的起点，即在使用第一种构造方法的情况下，重新设置输入点到整个字节数组byteArray的起点处，而在使用第二种构造方法的情况下，则是重新设置输入点到整个字节数组byteArray的 offset 处，通过这个 resr 方法，用户可以在任何需要的时候，重新来读取一次数据。

1.7.1.3 FileInputStream 类

经常用做输入源的，除键盘之外，就是文件了。特别是在对大型数据进行处理的过程中，所要处理的数据都是存在文件中的，用户在程序中使用这些数据时，只需从文件中读取即可。FileInputStream 类正是为满足这种从文件获取数据的需要而存在的，它所构造的输入流的起点就是文件。根据已知条件的不同，FileInputStream 类提供了三种不同的构造方法，大大方便了用户对文件输入流的构建，对它们分别举例说明如下：

（1）FileInputStream object_ FileInputStream=new FileInputStream("example");

通过这个方法构造的输入流，它的起点是文件名为"example"的文件。在已知文件名的情况下，就可以简单地利用这个构造方法来构造一个文件输入流，但是这个文件名却是依赖于系统的。

（2）File file=new File("c:/test/example");

FileInputStream object_ FileInputStream=new FileInputStream(file);

通过这个构造方法，建立一个 file 指定的文件输入流。

（3）FileInputStream object_ FileInputStream=new FileInputStream(file Descriptor);

在已知文件描述符的情况下，可以利用这个构造方法来构建一个文件输入流，其起点就是文件描述符为 Descriptor 的文件。

在 FileInputStream 类中，同样存在三种形式的 read 方法，available 方法，skip 方法和 close 方法，但它对 mark 和 reset 方法不提供支持。此外，由于文件存在着文件描述符，而这个类有可以通过文件描述符来构造一个实体对象，所以此类自然提供了一个与文件描述

符有关的方法 getFD(). 通过这个方法，可以获得作为输入流起点的文件的文件描述符。

1.7.1.4 FilterInputStream 类

FilterInputStream 类通过其构造方法可以创建一个过滤器输入流实体对象，即

FilterInputStream object_ FilterInputStream=new FilterInputStream (in);

由于这个输入流实体对象必需依附在另外一个输入流之上，所以在构造该类的实体对象时，必须以参数的形式指明它所依附的输入流对象，当向该类的实体对象索取数据时，它就会向它所依附的输入流对象索取数据。如果将数个 FilterInputStream 类实体对象串起来，如：

FilterInputStream object1=new FilterInputStream (in);

FilterInputStream object2=new FilterInputStream (object1);

FilterInputStream object3=new FilterInputStream (object2);

那么当用户向 object3 索取数据时，object3 就会向 object2 索取数据，而 object2 向 object1 索要数据，object1 再向输入流对象 in 索要数据。当然，这样的串接并没有什么实际意义，正如该类的实体对象本身也没有什么实际上的功能和作用一样，但是该类的几个子类 BufferedInputStream,DataInputStream,LineNumberInputStream,PushbackInputStream却十分具有利用价值，接下来，将分别对它们加以说明。

1）BufferedInputStream 类

BufferedInputStream 类可以构造出一种带有缓冲区的输入流，也就是通过构造函数给真正的输入流加上一个缓冲区，使程序并不是直接从输入流中读取数据，而是从缓冲区中读取数据。

这种读取方式，使得程序每读一次数据时，未必导致一次真正意义上的输入动作，而一般情况也正是先将数据送入到缓冲区中，然后在某个需要的时刻通过一次快速访问缓冲区的动作来读得这些数据，这种访问动作，可以通过 read 方法来完成。

给真正的输入流加上一个缓冲区以构造一个 BufferedInputStream 实体对象的方法有两种，即

（1）InputStream object_ InputStream=new BufferedInputStream(new ByteArrayInputStream (byteArray));

在这个方法中，真正的输入流是一个有 BufferedInputStream 类构造的字节数组输入流，其中 byteArray 是一个字节数组。通过 BufferedInputStream 类的构造方法，给这个输入流加上一个系统默认大小的缓冲区，从而构造出 BufferedInputStream 类的输入流对象。

（2）InputStream object_ InputStream=new BufferedInputStream(new ByteArrayInputStream (byteArray), 512);

这个方法与上一个方法不同之处在于它比上一个方法多出了一个整型值参数，从而允许用户来决定加到真正输入流上的缓冲区的大小。在这个例子中缓冲区的大小是 512 个字节。

给一个真正的输入流加上缓冲区，其目的有二：一是可以减少不同计算机设备之间由于速度上的不同而使程序流程读取数据时导致的阻塞时间，从而提高程序整体的执行效率；二是可以使原来不支持 mark 和 reset 方法的输入流，在加上缓冲区之后，变得具有 mark 和 reset 功能了。

2）DataInputStream 类

前面介绍的几种输入流，只能以字节或字节数组的形式来读取并返回各用户数据，而DataInputStream类则通过对DataInput接口的实现，根据用户的要求向其对象所依附的输入流索取数据，并以适当的类型形式返回给用户，诸如

readBoolean()方法可以读取并返回一个布尔型数据；

readByte()方法可以读取并返回一个 byte 型数据；

readUnisignedByte()方法可以读取并返回 一个 8bit 宽的 int 型数据；

readShort()方法可以读取并返回一个 short 型数据；

readUnisignedShort()方法可以读取并返回一个 16bit 宽的 int 型数据；

readChar()方法可以读取并返回一个 char 型数据；

readInt()方法可以读取并返回一个 int 型数据；

readLong()方法可以读取并返回一个 long 型数据；

readFloat()方法可以读取并返回一个 float 型数据；

readDouble()方法可以读取并返回一个 double 型数据。

此外，该类提供了 readLine()方法来一次读取一行的数据，并以 string 的形式返回，而这里的一行，是指以 ASCII 编码以 \n、\r、\r\n 或 EOF 其中一个为结束符的字符串。而有时，用户需要的却是一种 UTF 格式的字符串，为满足这种需要，DataInputStream 还提供了 readUTF 方法。

DataInputStream类的构造方法十分简单，只需要指明其对象所依附的真正输入流即可，如：

DataInputStream object_ DataInputStream=new DataInputStream(object);

其中 object 可以是任一真正输入流对象，诸如字节输入流等。

3）LineNumberInputStream 类

有时，我们会对输入内容的行号感兴趣，十分关注当前输入到类第几行，而LineNumberInputStream类所生成的输入流实体对象，正好可以对行号进行跟踪，并且在该类中也提供了与行号操作相关的方法setLineNumber(LineNumber)方法和getLineNumber()方法，它们分别用于设置输入内容的当前行号和获取当前行号。

当然，在构造LineNumberInputStream类实体对象时，也必须以参数的形式指明它所依附的真正的输入流，如真正输入流为in，那么生成该类实体对象object_ LineNumberInput-Stream 的方式如下：

LineNumberInputStream object_ LineNumberInputStream=new LineNumberInputStream(in)

另外，该类是支持 mark,reset 和 available 方法的。

4）PushbackInputStream 类

由 PushbackInputStream 类创建的输入流对象，具有把一个已经读取的字节“推”回去的能力，这个能力可以通过 unread 方法来实现。但需引起注意的是，该类的对象允许且仅允许通过 unread 方法推回一个字节的数据，如果试图通过 unread 方法推回多个字节的数据，则会引发 IOException 例外。

该类的构造方法与其他 FilterInputStream 字类的构造方法一样，但需要指明它所依附的真正的输入流。如：

PushbackInputStream object_ PushbackInputStream=new PushbackInputStream(in);

其中 in 是一个真正的输入流。而类中的 read,available 等方法仍与前面介绍的相同。

1.7.1.5 PipedInputStream 类

PipedInputStream 类的实体对象只有与 PipedOutputStream 实体对象相连接才会真正发挥作用。这是因为当我们从一个 PipedInputStream 类实体对象读取数据时，执行这个动作的线程 A 是从另外一个线程 B 处接受数据的，而线程 B 的作用恰恰是用来向一个与该 PipedInputStream 类实体对象相连接的 PipedOutputStream 实体对象输出数据。鉴于此，我们在构造一个PipedOutputStream实体对象时，一般都要以参数的形式指明要与之相连接的PipedOutputStream实体对象，即如果object_ PipedOutputStream是一个PipedOutputStream实体对象的话，构造一个与它相连接 PipedInputStream 实体对象的方式为：

PipedInputStream object_ PipedOutputStream= new PipedInputStream (object_ PipedOutputStream);

有时也可以构造出一个PipedInputStream实体对象，通过connect方法指明与之相连的PipedInputStream 实体对象，即

PipedInputStream object_ PipedOutputStream= new PipedInputStream ( );

object_ PipedOutputStream.connect(object_ PipedOutputStream);

PipedInputStream 实体对象作为一种输入流存在，自然还要具有 read 的功能。

关于 PipedInputStream 实体对象和如何与 PipedOutputStream 实体对象配合使用的，可以参见 PipedOutputStream 类部分及其中的例程。

1.7.1.6 SequenceInputStream 类

前面各节介绍的都是能简单地生成一个输入流的类，在这一节中，让我们来看一个可以将两个甚至数个输入流串起来以构成一个单一输入流--- SequenceInputStream类。在这个类中，根据要组合的已知输入流的不同，存在两种形式的构造方法，其一是将两个已知的输入流串起来的构造方法。如果ins1与ins2是两个已知输入流，那么此构造方法的用法为：

SequenceInputStream object_ SequenceInputStream= new SequenceInputStream(ins1,ins2);

另外一种形式的构造方法，是在要将三个或三个以上的输入流串起来或者是在要输入流的个数未知的情况下使用。在使用这种方法时，需要给定一个参数，这个参数必须是某些实现了接口 Enumeration 的类实体对象，而这种构造方法也正是通过这个参数得知要串接的输入流的信息的。例如ins1,ins2,ins3是已经建好的输入流，那么用第二种构造方法构造一个 SequenceInputStream 实体对象的方式为：

```
Vector object_Vector=new(Vector);
object_Vector.add(ins1);
object_Vector.add(ins2);
object_Vector.add(ins3);
SequenceInputStream object_ SequenceInputStream= new SequenceInputStream
(object_Vector);
```

由于SequenceInputStream类创建的实体对象也是一种输入流的共性，即read的功能，另外，由于它是在将若干已建好的输入流串接起来的基础之上构造出来的，在读上又具有

特殊性，这就是在从它读数据的时候，如果遇到了某个输入流的EOF，而该输入流之后还有其他输入流的话，该SequenceInputStream类实体对象就会自动切至下一个输入流中，并根据情况继续提供数据。在提供数据时，如果到达了整个SequenceInputStream输入流的流尾，它同样会返回一个-1值以便通知编程人员。

1.7.1.7 StringBufferInputStream类

该类与ByteBufferInputStream类十分相似，其提供的方法情况也与ByteBufferInputStream类的一样，唯一不同的地方在于StringBufferInputStream类所创建的输入流，其起点是以参数的形式在StringBufferInputStream类构造方法中指定的字符串，那么StringBufferInputStream实体对象的构造方式如下：

```
StringBufferInputStream object_StringBufferInputStream=new StringBufferInputStream(s);
```

1.7.2 输出

在前文中介绍的是以程序作为输入流流向的目标，而在这一部分，将介绍以程序作为输出流流出的源泉。在Java中，大部分的流类都是"成对出现"的，如有InputStream类，就有OutputStream类；有ByteArrayInputStream类，就有ByteArrayOutputStream类。鉴于此，如果在本节中有与上一节讨论过的相似之处，此处就不再重复了。

1.7.2.1 OutputStream类

在这一部分介绍所有输出流的共同的祖先——OutputStream类，它的最直接的子类是ByteArrayOutputStream类，FilterOutputputStream类，FileOutputStream类，PipedOutputStream类。OutputStream类与InputStream类一样，也就是一个抽象类，因此它提供的方法中不包括构造方法，无法直接产生它的一个实体对象。像上面Java输入中介绍的那样，它作为继承类型，由其子类去构造真正的输出流实体对象，然后就可以把那个输出流当成是一个最抽象的输出流来使用。例如：

```
OutputStream object_OutputStream=new BufferOutputStream(new ByteArrayOutputStream());
```

OutputStream类作为一个抽象类，特别是作为对所有输出流的一种概念上的抽象，必然是要规定出一些应用最广泛、同时又是最基本的输出流的行为模式，即方法。在这个作为输出流祖先的类中，必然少不了的方法是write。与InputStream类中的read方法相对应，此处的write方法也根据不同的情况分成了三种形式。

（1）

```
int b=5;
object_OuptputStream.write(b);
```

通过这个方法，可以向外输出一个字节的数据，而要输出的数据则是以参数b的形式给出的，在这里，我们输出了数据5。

（2）

```
byte[]b={'0','1','2','3','4'};
object_OutputStream.write(b);
```

当我们要一次输出存于一个字节数组中的全部数据时，往往采用这种方法。在此处，输出了字节数组b中的全部内容。

（3）

```
byte[]b={'0','1','2','3','4'};
object_OutputSteam.write(b,1,3);
```

有时，我们仅仅需要一次输出某个字节数组中连续的几个元素，而不是全部的数据，

在这种情况下，就可以通过采用此处这种形式的write方法来实现。在这个具体的例子中，我们输出的是字节数组b中从下标为1起的连续3个字节的数据，即b[1]('1'),b[2]('2')和b[3]('3')。

在用上述三种形式的write方法进行输出时，同样都可能面临阻塞程序流程的问题，这种问题通常是发生在输出方用write送出数据，而对方却没有将数据读走的情况下。既然会对程序流程进行阻塞，那么我们岂不要一直等待对方来读取数据了吗？其实，我们完全可以利用多线程的方式，一面等待对方取走数据，一面又在进行其他方面的工作。关于线程与多线程，将在后续章节中进行介绍。

有些时候，为了获取较高的执行效率，在选定并设计某种输出流时，往往会先将数据送到一个缓冲区中，之后在缓冲区满时或者是使用了某种方法时，才将缓冲区中数据一次送给对方，而不是每一次的数据送出都导致一次真正意义上的向对方输送数据的动作。凡事有利便必有弊，有些时候，数据已送到了缓冲区中，作为程序设计者可以认为已将数据送出了，而对方可能并没有真正地收到数据，从而造成数据的丢失和程序运行的错误。针对这种情况，OutputStream类中提供了一个flush方法，它的作用就是清除流，也就是使用具有缓冲区的输出流对象将其缓冲区中的数据全部强制性的输送出去。当然，如果某个输出流对象根本就没有缓冲区的存在，那么flush方法也就不会起任何作用了。

在Java的优越性介绍中，我们提到过它具有动态内存机制的设计，使得Java在进行垃圾回收时，可以将某个成了垃圾的输出流对象回收。但是有时某个输出流已经没有了存在的价值，而到Java回收它的时刻，就可能存在输出流无意义地占用某些系统资源的情况，同时也可能成为影响程序执行效果的潜在因素。因此，我们还是用close方法在不用某个输出流时关闭它，并释放它所占用的系统资源。当然，这个道理同样适用于InputStream类及所有输入流中的close方法。

1.7.2.2 ByteArrayOutputStream类

ByteArrayOutputStream类所创建的输出流对象，是以字节数组为终点的。作为终点的字节数组大小可以采用系统默认值0,也就是可以由程序员在构造方法中指定，但无论是哪一种方式，在用于存放输出内容的字节数组的内容不够用，Java都会自动帮助我们增大这个容积值。从上述简述中我们不难想到，ByteArrayOutputStream类必然存在着两种形式的构造方法，即

（1）ByteArrayOutputStream object_ ByteArrayOutputStream=new ByteArrayOutputStream( );

通过这个方法构造出来的输出流对象，它的终点——字节数组的初始化大小为零。

（2）ByteArrayOutputStream object_ ByteArrayOutputStream=new ByteArrayOutputStream(512);

有时，我们想直接在构造ByteArrayOutputStream类的输出流对象时指定作为终点的字节数组的初始化大小，就可以采用这种方式的构造方法。在这个例子中，字节数组的初始化大小为512。

ByteArrayOutpuStream类除了提供了write方法之外，还提供了几个在OutputStream抽象类中没有但又十分有用的方法。

size方法让我们可以得知作为输出流终点的字节数组的当前大小。

reset方法与ByteArrayInputStream类中的reset方法相似，其作用是重置输出流，从而使程序可以重新使用作为数据流终点的字节数组。

如果只用write方法，只是向ByteArrayStream类生成的输出流输出数据，可有时我们会需要将它作为一种中介，通过它将数据再送到另外一个输出流中去，恰好ByteArrayOutputStream类提供了writeTo方法来帮助我们实现这种愿望。如os是另外一个输出流对象，那么通过object_ByteArrayOutputStream向它输出数据的方式为：

```
object_ByteArrayOutputStream.write(5);
object_ByteArrayOutputStream.writeTo(os);
```

在ByteArrayOutputStream类中还有三个十分有益的方法，即toByteArray()方法，toString()方法和toString(int)方法，它们的功能都是可以将目前的输出内容复制之后返回，但返回的形式又各有特色。其中toByteArray()方法是以一个字节数组的形式返回，后两者之间的区别则在于：toString()只是简单地进行了复制与返回，而toString(int)不仅是将当前的输出内容复制到了一个字符串之中，而且在复制过程之中将作为输出流终点的字节数组中的数据由8位转换成16位后，由转换后的数据形成了返回字符串；由8位向16位的转换方法是原来的8位数据作为16位数据中的低8位存在，而高八位值则由给定的参数决定。

1.7.2.3 FileOutputStream类

对数据进行输出时，屏幕是我们最常用到的终点站，其次就应该是文件了。FileOutputStream类就为我们提供了将文件作为输出流终点的可能性。与FileInputStream类相似，根据已知条件的不同，我们也可以从该类提供的三种构造方法之中选择其一来创建FileOutputStream输出流对象，现分别对这三种构造方法举例如下：

（1）FileOutputStream object_FileOutputStream=new FileOutputStream("example");

在这里，通过指定一个作为输出流终点的文件名"example"来构造FileOutputStream实体对象，其中"example"与该程序在同一目录下，否则应该指明路径。但需要引起注意的是，作为已知条件的文件名是依赖于系统的。

（2）File file=new File("c:/test/example");

FileOutputStream object_FileOutputStream=new FileOutputStream(file);

当已知条件是输出流终点文件的File实体对象时，就可以通过这种方式来构造一个文件输出流。

（3）FileOutputStream object_FileOutputStream=new FileOutputStream(fileDescriptor);

其中fileDescriptor是作为输出流终点的文件的文件描述符。从这种方式的构造方法可以看出，我们完全可以根据已知的文件描述符来创建一个FileOutputStream实体对象。

在利用上述三种形式的构造方法来创建文件输出流时，都可能遇到一个相同的问题，即没有找到作为文件流终点的文件。在这种情况下，系统会产生出一个FileNotFoundException例外，如果我们想对这种情况进行处理，就可以借助try...catch形式，编上一段自己的处理程序，如：

```
try{
FileOutputStream object_FileOutputStream=new FileOutputStream("example");
}
catch(FileNotFoundException fnfe){
System.out.println(fnfe);
}
```

除三个最基本的构造方法之外，FileOutputStream类还提供了三种write方法和close方法，它们的形式与用法与其父类中的一致；此外，还有与FileInputStream类中功能相同的getFD()方法。

1.7.2.4 FilterOutputStream类

FilterOutputStream类与FilterInputStream类极其相似，它所创建的实体对象也必须依附于另外一个输出流上，因此在它的构造方法中，也不可避免地需要一个输出流对象作为参数，但不同之处在于，FilterOutputStream类的构造方法是public的，而FilterInputStream类的却是protected的，假如outStream是一个真正意义上的输出流，那么构造一个FilterOutputStream类实体对象的方法如下：

FilterOutputStream object_FilterOutputStream=new FilterOutputStream(outStream);

当我们向object_FilterOutputStream输出数据时，它就必须向所依附的输出流outStream输出数据。当然，outStream也未必一定是一个真正意义上的输出流对象，因为我们可以将数个FilterOutputStream实体对象串起来，只需保证最后一个FilterOutputStream实体对象所依附的输出流是真正意义上的即可。如：

FilterOutputStream object1=new FilterOutputStream(out);

FilterOutputStream object2=new FilterOutputStram(object1);

FilterOutputStream object3=new FilterOutputStram(object2);

其中，out是一个真正意义上的输出流。当我们用write方法向object3输出流写数据时，它就会将数据输送给object2，而object2再送给object1，最后由object1将数据送给了真正的输出流对象out。

当然，FilteOutputStream类与FilterInputStream类一样，本身并不具有什么实际意义上的功能和作用，真正引起我们注意的，同样是它的几个子类：BufferedOutputStream类、DataOutputStream类和PrintSteam类。接下来，让我们分别来讨论和学习它们。

1）BufferedOutputStream类

在操作系统课中，我们曾学过向磁盘写数据时是以块为单位的，因此无论是一次写一个字节的数据还是写一个块的数据，所需的时间几乎是一样的，那么我们为什么不采用缓冲区技术，待块满时一次写出呢？这样不仅可以提高效率，还可以避免为等对方将数据取走而造成的程序阻塞。这样的道理也同样适用于那些输出流终点是以块为存储单位的设备。在Java中，为满足这种加缓冲区的需要，不仅提供了BufferedInputStream类，还提供了BufferedOutputStream类。通过BufferedOutputStream类，可以给一个真正的输出流加上一个缓冲区，这样在每次向输出流写数据时，都是写到了缓冲区中。只有在缓冲区满了时，或者是通过flush()方法清除输出流时，才会将缓冲区中的数据一次性的写给真正的输出流，从而避免了每一次的write调用都导致一次真正意义上的输出动作。

给一个真正的输出流加上缓冲区的方式有两种，即

（1）OutputStream object=new BufferedOutputStream(new FileOutputStream("example"));

给在参数中指定的真正意义上的输出流——以文件"example"作为终点的文件输出流加上一个系统默认大小的缓冲区，从而构造出了一个BufferedOutputStream实体对象。

（2）OutputStream object=new BufferedOutputStream(new FileOutputStream("example"),1024);

在这个构造方法中，我们不难发现它比前者多了一个参数1024，正是通过这个参数，

我们可以在程序设计中人为地设定缓冲区的大小，此例中，加给文件输出流的缓冲区的大小为 1024 个字节。

2）DataOutputStream 类

DataOutputStream 类的实体对象在输出数据的形式上，比其他输出流有所突破，它已经不仅仅局限于以字节或字节组的形式写数据，而是可以以多种独立于机器的格式进行写输出。要想利用该类的这种优越性，首先要通过构造方法创建它的一个实体对象，即

DataOutputStream object_DataOutputStream=new DataOutputStream(os);

其中，os 是该实体对象 object_DataOutputStream 所依附的真正的输出流。在此之后，就可以调用与类 DataInputStream 中提供的读方法相对应的各种写方法来实现各种编程要求。这些独具特色的方法是：

writeBoolean(boolean)方法输出一个布尔型数据；

writeByte(int)方法输出一个 8bit 的 int 型数据；

writeShort(int)方法输出一个 16bit 的 int 数据；

writeChar(int)方法输出一个 16bit 的字符数据；

writeInt(int)方法输出一个 32bit 的整型数据；

writeLong(long)方法输出一个 64bit 的 long 型数据；

writeFloat(float)方法输出一个 32bit 的单精度浮点型数据；

writeDouble(double)方法输出一个 64bit 的双精度浮点型数据；

writeBytes(String)方法输出一个 String 类的对象实体，并且每个字符的宽度为 8bit；

writeChar(String)方法输出一个 String 类的对象实体，但每个字符的宽度为 16bit；

writeUTF(String)方法允许我们以 UTF 的格式来输出一个 String 类的实体对象。

此外，该类中的 size()方法还可以让我们获知已经输出的字节个数。

3）PrintStream 类

在所有的例程中，我们几乎都用到了 System.out，并且利用它的 print 和 println 方法向屏幕输出数据，这个 system.out 就是 PrintStream 类的一个输出流对象。在 System 中还有一个 system.err 也是 PrintStream 类的输出流对象，这两个对象在程序执行时是分别连到了该程序的标准输出流（C 语言中的 stdout）和标准错误流（C 语言中的 stderr）上，因此我们在用 System.out.println 方法时可以向屏幕打印数据。

PrintStream 类的构造有两种形式：

PrintStream object_PrintStream=new PrintStream(out);

（1）其中 out 是 PrintStream 类的实体对象 object_PrintStream 所依附的真正的输出流对象，因为 PrintStream 类毕竟是 FilterOutputStream 类的子类。

（2）PrintStream object_PrintStream=new PrintStream(out,true);

在这个构造方法中，比前者多了一个布尔型的参数 true，这表明由此构造方法创建的依赖于 out 的 PrintStream 输出流对象在打印时，如果遇到了换行字符，就会自动地清除缓冲区；反之，如果布尔型参数不是 true，而是 false，那么就不会发生自动清除缓冲区的情况。

PrintStream 类提供的方法中，除最普通的 write 方法之外，还有许多与 DataOutputStream 中的特殊写方法十分相似的方法，不同之处在于在此类中，方法名被冠以 print 或 println，例如 DataOutputStream 中要输出一个单精度浮点数时用 writeFloat(float)方法，而在此处则用

print(float)方法或者是println(float)方法。但这里的print和println又不完全与DataOutputStream中的各种有特色的写方法相等效，因为它们二者的职能只是将数据以字符串的方式表现出来，而不是输出数据本身的存储形式。另外，print与println的差别只在于后者在输出数据之后，比前者在最后多加上了一个换行字符，而前者则不会加上换行符。

在使用PrintStream类的实体对象输出字符串时，有一点是必须引起注意的，那就是字符串中的每个字符的较高八位都会被忽略掉。

1.7.2.5 PipedOutputStream 类

现在，让我们来看一下OutputStream类的最后一个子类——PipedOutputStream类。该类与PipedInputStream类是一对，它们二者的实体对象必须一并使用，利用它们可以建立起一个供不同线程之间通信的管道线。关于将PipedOutputStream类的实体对象与PipedInputStream类的实体对象结合起来的途径，在这里给出另外两种：

（1）PipedOutputStream object_ PipedOutputStream=new PipedOutputStream(object_ PipedInputStream);

通过PipedOutputStream类的构造方法中指明与生成的输出流对象相连接的PipedInputStream实体对象，来建立二者之间的连接关系。

（2）PipedOutputStream object_ PipedOutputStream=new PipedOutputStream（）

object_ PipedOutputStream.connect(object_ PipedInputStream);

在这种方式中，是通过利用PipedOutputStream类中的connect方法，来建立PipedOutputStream实体对象object_ PipedOutputStream与PipedInputStream实体对象object_ PipedInputStream之间的连接的。

1.7.3 与输入，输出相关的类

前面介绍的是Java.io包中与流有直接关系的类，在这一部分，则着重讨论一下Java.io包中与输入输出有关部分的，并且又是十分有用的几个类。

1.7.3.1 File 类

File类为我们提供了表示一个“文件”或者“目录”对象的方法。通过为文件建立File对象，我们可以利用该类所提供的各种方法来对“文件”或者“目录”进行处理。创建File对象的方法有三种形式：

（1）File file1=new File("c:/test/file1.txt");

根据参数指定的文件路径来构造一个File实体对象。

（2）File file2=new File("c:/test","file1.txt");

在这方式中，根据给定的目录来构造一个File实体对象，其中"c:/test"为目录的路径，"file1.txt"为文件的名称。

（3）File file3=new File(file,"file.txt");

通过这个构造方法，可以根据给定的目录文件对象file来构造一个新的File实体对象。

在利用构造方法生成File实体对象时，该对象所描述的文件名可以是相对的，也可以是绝对的，但必须遵循主机平台的文件名约定。

针对程序远在设计中可能对文件进行的操作情况，该类提供了用于获取File对象所描

述的文件的文件名的方法 getName( )、获得文件路径的方法 getPath( )、获得文件绝对路径的方法 getAbsolutePath( )、以及判断文件是否可读写的方法 canWrite( )和 canRead( )等等。关于这方面的方法，可以从第 3 章中获得，这里不再重述。

1.7.3.2 RandomAccessFile 类

RandomAccessFile 类同时实现了 DataInput 接口和 DataOutput 接口，从而使我们可以很容易地直接读取和写入各种类型的数据到要处理的文件之中。正由于该类同时实现了两个接口，所以它也就具有了 DataInputStream 类和 DataOutStream 类中的一些相似功能的方法，诸如 readLong 方法和 writeLong 方法等，但是，RandomAccessFile 类的构造方法却与 DataInputStream 类和 DataOutStream 类的不同。在 RandomAccessFile 构造方法中，作为参数给出的是要处理的文件及对文件的读写权的设定，其形式有两种，即

（1）RandomAccessFile object=new RandomAccessFile ("example"， "r");

在这个构造方法中，以字符串参数指明要处理的文件为“example”，同时根据参数“r”可以得知这个文件的访问模式是“只读的”，其中“r”也可以由“rw”取代，表示这个文件的访问模式是“读写的”。

（2）File file=new file("c:/test/example");

RandomAccessFile object=new RandomAccessFile (file， "r");

这个构造方法与前者不同之处仅在于是以 File 对象指定了要处理的文件。

此外，RandomAccessFile 类是一种随机访问文件类，当然要提供一种有助于随机访问的方法，这就是方法——seek(long)。通过它，可以将文件的读写指针移动到希望读写的位置上。而另一种方法 getFilePointer( )则意义相反，返回到文件读写指针的当前位置。正是由于该类拥有这两种方法，又同时拥有 DataInputStream 类和 DataOutStream 类中的部分方法，因此在某些情况下，具有其他流类不可比拟的优越性。

1.7.4 示例程序

下面我们来看一段示例程序，以便能够更好的理解与输入输出有关的类的各种方法以及它们的用法，该程序的主要功能是将输入的文本内容和数值内容存入一个您指定的文件，若该文件不存在，则询问您是否新创建该文件，若该文件存在，则先将该文件的内容显示出来，并询问您是否覆盖该文件的所有内容。

```
import Java.io.*;
public class Fileio {
  public static void main(String[ ] args)throws Exception{
       boolean t=true;
      do{
       byte ba[ ]=new byte[50];
       System.out.println("这是一个简单的文件输入输出流的例子,"+'\n'+"您可以将下列您输入的字符串和数字存入文件，然后将它显示出来。");
       System.out.println(" 请输入字符串:");
       System.in.read(ba);
       String baa=new String(ba,0);
```

```
byte bd[ ]=new byte[50];
System.out.println(" 请输入数字:");
System.in.read(bd);
String bdd=new String(bd,0);
byte buff[ ]=new byte[10];
int i;
System.out.println(" 请输入您存取的文件名 .(Exit 为退出)");
System.in.read(buff);
String str=new String(buff,0);
if (str.trim( ).equals("Exit"))
    t=false;
    File input=new File(str.trim( )+".txt");
    System.out.println(input.getAbsolutePath( ));
    if(input.exists( )){
    System.out.println(" 文件已存在 .");
    System.out.println(input.getName( ));
    System.out.println(input.getPath( ));
    System.out.println(input.getAbsolutePath( ));
    System.out.println(input.getParent( ));
     FileInputStream is=new FileInputStream(str.trim( )+".txt");
     int n,ch,d=0;
     for(n=0;(ch=is.read( ))!=-1;n++){
     if (ch>='0'&&ch<='9')
     d++;
     System.out.print((char)ch);
     }
     n=n-d;
     System.out.println(" ");
     System.out.println(n+" 个字符 ");
     System.out.println(d+" 个数字 ");
     byte buf[ ]=new byte[5];
     System.out.println(" 您真的想覆盖吗?");
     System.in.read(buf);
     String st=new String(buf,0);
     if (st.trim( ).equals("y")){
   FileOutputStream  fo=new FileOutputStream(str.trim( )+".txt",false);
       OutputStream aos=new BufferedOutputStream(fo,1024);
       aos.write(ba,0,baa.trim( ).length( ));
       aos.write(bd,0,bdd.trim( ).length( ));
```

```
    aos.close( );
    FileInputStream sis=new FileInputStream(str.trim( )+".txt");
                    d=0;
for(n=0;(ch=sis.read( ))!=-1;n++){
if (ch>='0'&&ch<='9')
d++;
System.out.print((char)ch);
}
n=n-d;
System.out.println(" ");
System.out.println(n+" 个字符 ");
System.out.println(d+" 个数字 ");


}
else {
System.out.println("File Not found,Do you want new it?");
   byte buf[ ]=new byte[5];
System.in.read(buf);
String st=new String(buf,0);
if  (st.trim( ).equals("y")){
   FileOutputStream fo
              =new FileOutputStream(str.trim( )+".txt",false);
   OutputStream aos=new BufferedOutputStream(fo,1024);
   aos.write(ba,0,baa.trim( ).length( ));
   aos.write(bd,0,bdd.trim( ).length( ));
   aos.close( );
FileInputStream is=new FileInputStream(str.trim( )+".txt");
int n,ch,d=0;
for(n=0;(ch=is.read( ))!=-1;n++){
if (ch>='0'&&ch<='9')
d++;
System.out.print((char)ch);
}
n=n-d;
System.out.println(" ");
System.out.println(n+" 个字符 ");
System.out.println(d+" 个数字 ");
```

```
}
}
        int n=0;
        for (n=0;n<ba.length;n++)
            ba[n]=' ';
        for (n=0;n<bd.length;n++)
            bd[n]=' ';
          }while (t);
       }
    }
```

运行结果如图 1.20 所示。

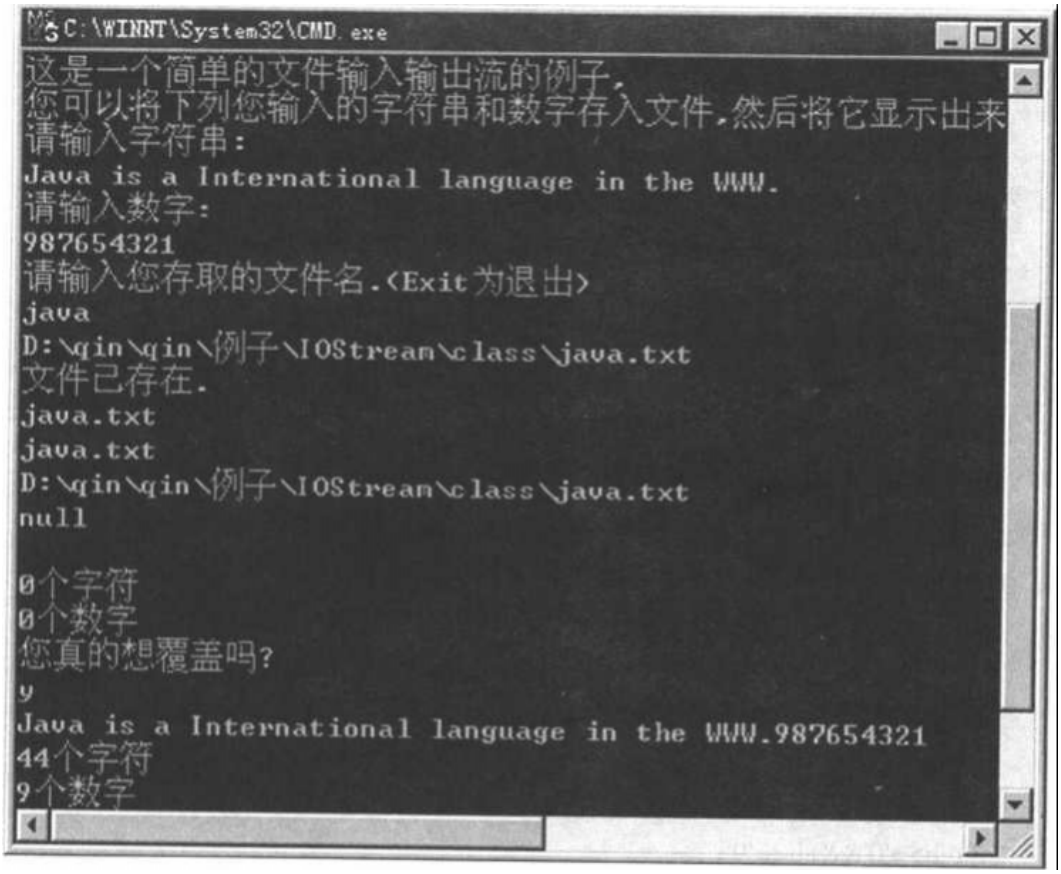

图 1.20　文件输入输出结果图

1.8　典型数据结构有关的 Java 类库及其使用

Java 提供了常用的与复杂数据结构有关的类，它包括字符串、数组、向量（Vector）、哈希表（Hashtable）、栈(Stack)、日期和时间（date）等。它们包含在 Java.lang 类库中。本

节介绍较为重要的类：字符串、数组、日期时间类（date）、向量类（Vector）、哈希表类（Hashtable）和堆栈类(Stack)。

1.8.1 字符串类

Java.lang 提供了两种字符串类：String 类和 StringBuffer 类。

1）String 类

String 类提供几种字符串创建方法：

String s="Hello, World!"; //使用字符串常量自动创建 String 实例，如 C++ 中一样。

String s= new String(String s); //通过 String 对象或字符串常量传递给构造方法。

public String(char value[ ]); //整个字符数组赋给 String 构造方法。

public String(char value[ ],int offset,int count);//字符数组一部分赋给String构造方法，offset //为起始下标，count 为子数组长度。

String 类提供了丰富的字符串操作方法，其中重要的列举如下：

public int length( ); //返回字符串的长度。

public char charAt(int index);; //返回字符串位置 index 处的字符。

public boolean equals(Object o); //比较两个字符串对象，相等返回 True，反之，返回 False。

public int compareTo(String s); //比较两个字符串字典顺序，相等返回 0，s 大于当前串 //返回一个负值，s 小于当前串返回一个正值。

public boolean regionMatches(int toffset,String Other,int ooffset,int len); //从当前字符串位 //置 toffset 开始寻找字符串 Other，起始位置为 //ooffset，长度为 len 的子串。如发现匹配，返回 //true，否则，返回 false。

public boolean startsWith(String prefix); //从当前字符串位起始位置 t 开始寻找字符串 //prefix。如发现匹配，返回 //true，否则，返回 false。

public boolean startsWith(String prefix); //从当前字符串位起始位置 t 开始寻找字符串 //prefix。如发现匹配，返回 //true，否则，返回 false。

public boolean endsWith(String suffix); //如当前字符串的结尾子串与 suffix 匹配，返回 //true，否则，返回 false。

public int indexOf(String str); //在当前字符串中寻找与 str 匹配的子串，返回首次匹配的 //起始下标值，无匹配返回 -1。

public String substring(int beginIndex,int endIndex); //从当前字符串起始位置 beginIndex //到结束位置 endIndex 的子串。

public String concat(String str); //当前字符串与 str 连接，返回连接后的字符串。

public String toLowerCase( ); //当前字符串全转换为小写形式。

public String toUpperCase( ); //当前字符串全转换为大写形式。

public Char toCharArray( ); /// 当前字符串转换为字符数组。

public Static String valueOf(type variable); // 把 variable 转换为字符串。

2）StringBuffer 类

String 类实现一种不能改变的静态字符串，StringBuffer 类实现一种动态可改变的字符串。StringBuffer 类主要用于创建 String 类，StringBuffer 一旦建好，用 toString( )方法将其转换为 String 类。以后，就可以使用 String 类方法来操作。

StringBuffer 类提供三种创建方法：

public StringBuffer( ); // 创建一个空的 StringBuffer 类。

public StringBuffer(int length); // 创建一个大小为 length 的 StringBuffer 类。

public StringBuffer(String str); // 按 str 创建一个动态可变的 StringBuffer 类。

StringBuffer 类提供的方法主要用于把输入数据转换为 String 类。输入数据可来自键盘或其他数据源。类型可以是字符、字符数组、整数、浮点数和 Object 类型等。下面介绍 StringBuffer 的主要方法：

public int length( ); // 返回缓冲区的字符数。

public int capacity( ); // 返回缓冲区剩余空间。

public synchronized StringBuffer append(type variable); // 把 variable 转换为字符串，然后
// 与当前串连。

public synchronized StringBuffer append(Char(char ch)); // 把字符 ch 连接到当前串尾。

public synchronized StringBuffer insert(int offset,type variable); // 把 variable 转换为字符串，
// 然后插入到当前串由 offset 指定的位置。

public synchronized StringBuffer insert(int offset,char ch); // 把字符 ch 插入到当前串由
//offset 指定的位置。

public synchronized String toString( ); // 把 StringBuffer 转换为字符串 String。

下面举一个简单的例子，说明用 StringBuffer 类将盘输入的数据建立一个字符串实例。

```
System.out.println("Enter Width!");
{int length =System.in.read( );
 StringBuffer str=new StringBuffer(length);
 while (ch=System.in.read( ))!='\n')
 {
  str.append(ch)
 }
}
```

1.8.2 数组类

Java 语言中的数组是一种专门类型，它们通过索引方式组织一种对象列表。Java 程序中数组说明如下例所示：

```
int count[ ];
char ch[ ][ ];
float f[ ];
```

上述数组说明也可以用另外一种等价形式描述如下：

int[ ] count;

char[ ][ ] ch;

float[ ] f;

Java 数组说明中不需要指明数组大小，其内存单元分配必须通过 new 操作，如下例所示：

int count[ ]=new int[10];或下面等价形式：

int count[ ];

count[ ]=new int[10];

Java数组有些操作可以很方便地通过系统类(java.lang.System)的方法来实现，如数组拷贝就可以用 arraycopy( )方法实现。有关详情，请参阅参考文献[10]。

1.8.3 Utility 类库的大致构成

日期时间类、向量类、哈希表类和栈类 都包含在 Java.Utility 类库中。Utility 类库是一个由一些实用工具类组成的集合，有些类还是 Java 语言所不可缺少的。Utility 类库的大致层次结构如图 1.21 所示。

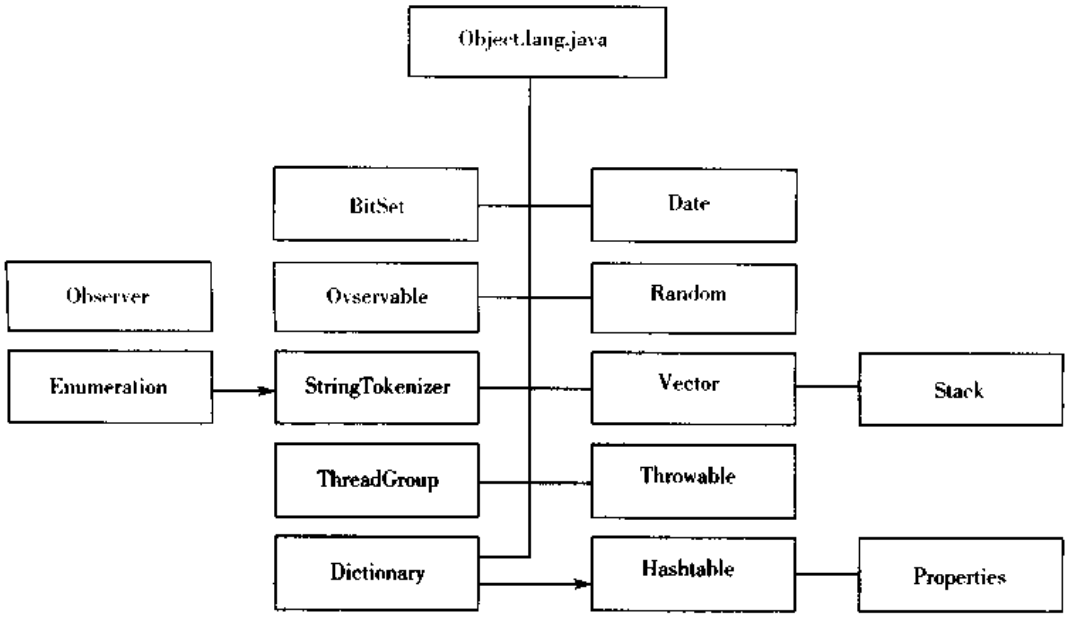

图 1.21 Utility 类库的层次结构

这里还需要说明几点：图中 Dictionary 是抽象类，Enumeration 和 Observer 是接口，其余属于普通类；Hashtable 除继承自 Dictionary 类外，还是 Java.lang 类库中 Cloneable 的一个实现；类似地，BitSet 和 Vector 除继承了类 Object 外，同时也是 Java.lang.Cloneable 的实现；BitSet 是最终类，不能用以创建子类。下面介绍 Utility 中重要的与数据结构有关的类。

1.8.4 日期时间类

日期时间类是一个相对简单但使用频繁的类，它提供了独立于具体系统的日期/时间

的表示形式。利用日期时间类提供的方法，你可以获取当前日期和时间，创建日期和时间参数，计算某年或某月的天数，或者对当前日期进行比较。日期时间类的实例可通过如下几种方法予以创建：

（1）public Date( )

这个构造方法将把当前日期和时间保存于所创建的Date实例。

（2）public Date(int year, int month, int date)

这个构造方法将根据所给定的year，month，date参数创建一个Date实例。也就是说，利用此构造方法，所创建的Date实例表示的日期与年、月、日参数表示的日期等同。

（3）public Date(int year, int month, int date, int hours, int minutes)

这个构造方法类似于第（2）个方法，只不过这里还提供了具体的小时（hours）和分钟（minutes）参数，因而时间要更为精确。

（4）public Date(int year, int month, int date,int minutes,int seconds)

这个构造方法类似于第（3）个方法，所不同的是这里还提供了具体的（seconds）参数，时间又精确了一些。

一旦创建了日期时间类的一个实例，就可以用它的方法来检索有关日期和时间的信息。下面列出了日期时间类的方法：

```
public int getYear( ); //返回当前日期中的年份。
public int getMonth( ); //返回当前日期中的月份。
public int getDate( ); //返回当前日期中的日期（0-31）。
public int getDay( ); //确认某天是星期几。
public int getHours( ); //返回当前日期中的小时数。
public int getMinutes( ); //返回当前日期中的分钟数。
public int getSeconds( ); //返回当前日期中的秒数。
public  boolean  befor ( Date when ); // 对日期实例所代表的时间和 when 进行比较。
                                      // 若比 when 早，返回 true; 否则返回 false。
public boolean after(Date when); // 对日期实例所代表的时间和 when 进行比较。若比
                                      //when/ 晚，返回 true;否则返回 false。
public boolean equal(Object obj); //比较两个日期对象。若相等则返回true;否则返回false。
public String toString( ); //返回当前日期参数的字符串表示形式。注意：不同主机系统的
                                      // 日期表示形式不尽相同。
```

日期时间类的使用非常简单，只需要创建一个类实例不带任何参数，即可以生成一代表当前日期的类。如下所示：

```
Date today = new Date( )
System. out. println(today. ToString( ));
```

或者也可用一更为简单的方式：

```
System. out. println(today);
```

后一种方式中，println将自动调用toString方法，因而无需显式调用。

1.8.5 向量类及其使用

向量这种数据结构并不是必不可少的，它的功能可通过创造性地使用数组来实现，但向量提供了一种更为优化的操作，它封装了许多复杂的操作。

1.8.5.1 向量和数组的异同

向量和数组存在许多相似之处，它们均可用以保存列表对象。数组只能保存固定大小的列表，它必须一次申请所有的存储单元。如果事先无法确定列表内究竟要存放多少对象，Java的数组就无能为力了，向量数据类型却提供了一种与“动态数组”相近的概念。

尽管向量与数组相比有许多重要的优点，但它也有一些不足之处，其中之一是它不能直接存储简单数据类型。向量和数组分别适用于不同的场合。一般而言，下列场合更适合于使用向量：

（1）如果你需要频繁进行对象的插入和删除工作，或者因为需要处理的对象数目不定；

（2）列表成员全部都是对象，或者可以用对象方便地表示；

（3）需要很快确定列表内是否存在某一特定对象，并且希望很快了解到对象的存放位置。

这主要是因为向量作为一个对象，提供了比数组更多的方法。另一方面，由于向量只能存储对象，如果需要把简单数据类型保存到向量，必须使用Java的数据类型类，因此，有些场合下使用数组反而比使用向量要方便一些。下面这些场合适合于使用数组：

（1）所需处理的对象数目固定，或大致可以确定，尽管具体哪些对象经常发生变化；

（2）所需处理的是简单数据类型。

1.8.5.2 向量类的构造方法、属性和方法

要使用向量，首先必须创建一个Vector类实例，这通过向量的构造方法创建。向量类共有3种形式的构造方法，如下所示：

```
Vector(int capacity, int capacityIncrement); //用指定的向量容量及其增量参数，创建一个
                                             //空向量。
Vector(int capacity);                        //用给定的向量容量参数，创建一空向量。
Vector( );                                   //创建一空向量。
```

这里需要对所谓的“向量容量”和“容量增量”参数作一说明。为优化存储管理，Java的每个向量均可以使用一个“向量容量”参数和一个“容量增量”参数。向量容量通常是一个大于向量实际元素个数的整数；容量增量则规定了每当向量元素个数达到极限时，需一次性扩充的向量容量大小。也就是说，当新增向量元素时，如果向量内元素个数已经达到了向量的极限大小，向量并不是申请一个空间，而是多个空间，数目由容量增量参数确定。如果容量增量参数为0，则每次向量容量都将增加一倍大小。设置这两个参数可增加向量类的使用效率。你也可以忽略这两个参数，这时Java将自动为你维护向量的使用，包括必要时向量的增容等。除构造方法外，向量类还提供了3个属性变量，如下所示：

```
protected int capacityIncrement;    //当向量大小不足时，所用的增量大小
protected int elementCount;         //向量的元素个数
protected Object elementData[ ];    //向量成员数据所用的缓冲
```

一旦创建了向量类的实例，就可以用它的方法来执行插入、删除及查找对象等操作。向量类提供了极为丰富的方法，下面是其中一些主要的方法：

```
public final synchronized void copyInto(Object anArray[]; //把向量元素拷贝到指定数组
public final synchronized void trimToSize( ); //把向量容量调整到正好等于向量元素个数
        //以压缩向量的存储空间。这样，随后的插入操作将导致重新申请内存空间。
public final synchronized void setSize(int newSize); //设置向量大小。如果向量被缩小，
        // 多余的元素将被截断；如果被放大，新增的向量元素将置为 null。
public final int capacity( ); //返回向量容量。
public final int size( ); //返回向量的元素个数，注意和 capacity( )之间的区别。
public final boolean isEmpty( ); //若向量不包括任何元素，返回 true;否则返回 false。
public final synchronized Enumeration elements( ); //返回向量元素所对应的枚举值，以便
        //随后用 Enumeration( )方法获取该向量元素。
public final boolean contains (Object elem); //若向量中包括了对象 elem，返回 true ；否
        //则返回 false。
public final int indexOf (Object elem); //返回向量下标；若对象不存在，返回 -1。
public final synchronized int indexOf (Object elem,int index); //从指定位置（index）开始
        //搜索向量，返回对象所对应的向量下标值。若未找到对象，返回 -1。
public final int lastIndexOf (Object elem); //从向量末尾向前搜索向量，返回对象的下标
        //值。
public final synchronized int lastIndexOf (Object elem,int index); //从指定位置开始向前搜
        //索向量，返回给定对象的下标值。
public final synchronized Object elementAt (int index); //返回指定下标处的对象。若下标
        //值非法，抛出一 ArrayIndexOutOfBoundsException 异常情况。
public final synchronized Object firstElement ( ); //返回向量的第一个元素。若向量为空，
        //抛出一 NoSuchElementException 异常情况。
public final synchronized Object lastElement ( ); //返回向量的最后一个元素。若向量为空，
        //抛出一 NoSuchElementException 异常情况。
public final synchronized void setElementAt (Object obj, int index); //把给定对象存放到给
        //定下标处，该下标处的原有对象丢失。若下标值非法，抛出异常情况
        //ArrayIndexOutOfBoundsException。
public final synchronized void removeElementAt (int index); //删除给定下标处的向量元
        //素，后面元素前移一个位置。若下标值非法，抛出异常情况
        //ArrayIndexOutOfBoundsException。
public final synchronized void insertElementAt (Object obj,int index); //把给定对象插入到
        //指定下标处。该下标之后的元素后移一个位置。若下标值非法，抛出
        //异常情况 ArrayIndexOutOfBoundsException。
public final synchronized void addElement (Object obj); //把给定对象增加到向量末尾。
public final synchronized boolean removeElement (Object obj); //从向量中删除指定对象。
        //若给定对象在向量中保存多次，则只删除其第一个实例。若向量中没
```

```
                    //有这个对象，返回 false。
    public final synchronized void removeAllElements ( ); //删除向量中的所有对象。这时向
                    //量将变成空向量。
    public final synchronized String toString ( ); //把向量转换成字符串。请注意：这个方法
                    //实际上覆盖了 Object 类中的 toString( )方法。
```

1.8.5.3 建立向量实例

和数组一样，向量的内存空间是通过new操作符实现的。一个向量在被创建后，将自行在系统内部维护一个专用数组，并在必要时对数组大小进行动态调整。比方说，如果插入一个向量元素时，向量的空间需求超过了这个内部数组的大小，向量将向系统申请一些新的内存空间。

```
Vector theVector = new Vector
```

1.8.5.4 向量维护

创建向量实例之后，就可以把对象插入到向量。这时将用到addElement( )方法。如下面的程序：

```
Vector theVector = new Vector( )
for( int i = 0; i<10; ++ )
{
    Integer newInteger = new Integer(i);
    theVector. addElement( newInteger );
}
```

addElement( )方法把对象插入到向量末尾。

向量可用以保存任何类型的对象，同一个向量实例甚至还可以保存多种不同类型的对象。下面我们给出另外一个例子，这个例子中，给向量交错插入一些浮点数和字符串。

```
Vector theVector = new Vector( );
new testString;
testString = " pai" ;
theVector.addElement( testString );
Integer testFloat = new Float( 3.14 );
theVector.addElement( testFloat );
testString = " Faill" ;
theVector.addElement(testString);
Integer testFloat = new Float( 59 );
theVector.addElement( testFloat );
```

执行这段代码，向量将包括如下内容：{" pai" ,3.14," Fail" ,59}。

增加向量元素的另一方式是通过insertElementAt( )方法。该方法把对象插入到向量的给定位置。

另一经常用到的维护操作是从向量中删除一个对象。这时有3 个方法可用：removeElement( ), removeElementAt( ), removeAllElement( )。3个方法分别用于不同场合。removeElement( )删除指定对象，removeElementAt( )删除给定位置处的对象，

removeAllElement( )则删除所有向量成员。对于前两个方法需要注意的是删除之后整个向量的变化：删除一个对象后，后面的对象将前移一个位置。对于前面例子中介绍的向量，下面语句将删除其中的3.14和60，而不是3.14和“pass”。

```
Integer deleteFloat = new Float(3.14);
theVector.removeElement(deleteFloat);
theVector.removeElementAt(2);
```

1.8.5.5 对象查找

与数组相比，向量的优势之一在于它提供了丰富的方法，使我们能够方便快捷地从向量中查找对象。对象查找中最常遇见的可能是给定一个下标值，希望由此确定该下标处的对象。这时可使用elementAt( )方法。如下面这行程序：

```
Object tempObj = theVector.elementAt(0);
```

这行程序返回一对象引用，该引用指向向量成员“pail”。

有时我们可能需要确定向量中是否包括了某一确定的对象，这如果使用数组是不太容易实现的。使用向量就方便得多，因为它有一个contains( )方法，可直接实现此项功能。我们再来看一下这个方法：

```
public final boolean contains(Object elem);
```

这个方法返回一布尔值。若向量包含了对象elem，返回true，否则返回false。

我们经常用下标来访问数组元素，这是使用数组的最有效方法。向量成员也可用下标进行访问，其中最简单的方法是使用刚刚介绍的elementAt( )方法，它可以返回给定下标处的对象。除这个方法外，其他两个用下标操作的方法是indexOf( )和lastIndexOf( )。下面我们再来看一下这两个方法：

```
public final synchronized int indexOf( Object elem, int start_index )
public final synchronized int lastIndexOf( Object elem, int start_index )
```

这两个方法均用以从给定下标开始搜索向量，确定向量中是否包含了指定的对象。如果是，则返回该对象的下标值，否则返回-1。差别在于indexOf( )是从前往后搜索，lastIndexOf( )是从后向前搜索。

向量的其他方法读者可参查相关资料。

1.8.6 哈希表类及其应用

Java的哈希表用于存储对象。

1.8.6.1 哈希表类的构造方法、属性变量和成员方法

1）构造方法

哈希表类（Hashtable）共有三种形式的构造方法：

```
public Hashtable(int initialCapacity, float loadFactor);// 参数 initialCapacity 给定哈希表创建初
                    // 始可容纳的元素数，loadFactor)为装载因子，取值0.0到1.0之间。
public Hashtable(int initialCapacity);// 参数 initialCapacity 给定哈希表创建初始可容纳的元
                    // 素数，loadFactor)为缺省值。
public Hashtable( ); // 参数 initialCapacity 和 loadFactor)为缺省值。
```

2）哈希表类没有定义属性变量

3）成员方法

哈希表类定义的主要方法如下：

```
public int size( ); //返回哈希表大小（即表中元素个数）。
public boolean isEmpty( ); //确认哈希表是否为空。
public synchronized Enumeration keys( ); //返回哈希表示内关键字枚举值。
public synchronized Enumeration elememnts( ); //返回有关内元素的一个枚举值对象，随后
                                              //Emumeration 类方法顺序获取对象。
public synchronized boolean contains(Object value); //确认哈希表内是否包括了给定的对象，
                                                    //与 containKey( )类似。
public synchronized boolean contains(Object key); //确认哈希表内是否包括了给定的关键字。
public synchronized Object get(Object key); //获取对应关键字的对象，如不存在返回 null。
protected void rehash( ); //扩充哈希表使之可以保存更多的元素，当哈希表达到饱和时，
                          //系统将自动调用此方法。
public synchronized Object put(Object key, Object value); //用给定关键字把对象保存到哈希
                                                          //表。随后该对象即可通过同样关键字由 get 方法获取。这里的关键
                                                          //字和元素均不可为空。
public synchronized Object remove(Object key); //从哈希表中删除与给定关键字相对应的对
                                               //象，如该对象不存在返回 null。
public synchronized void clear( ); //从哈希表中删除所有对象，使其为空。
public synchronized String toString( ); //把哈希表内容转换为一字符串，此方法覆盖了 Object
                                        //类的 toString( )方法。
```

1.8.6.2 哈希表的使用

与 Java 中其他对象一样哈希表创建需使用 new 操作，如下例所示：

```
HashTable hash_1=new HashTable( );
```

为了把对象保存到哈希表，必须为每一个对象分配一关键字。关键字可以为任意对象，但必须实现了 hashCode( )和 equals( )方法。Java 提供的类几乎都提供了这两个方法。建好了哈希表后，现在利用上述 put 方法把数据存放到哈希表中去。

```
hash_1.put("one",new Integer(1));
hash_1.put("two",new Integer(2));
hash_1.put("three",new Integer(3));
```

为了把对象从哈希表中删除，须使用 remove( )方法。例如：

```
hash_1.remove("two");
```

这时对象 Integer(2)将从哈希表删除，同时返回这个被删除的对象。如指定的关键字不存在，remove( )返回 null。

使用哈希表类的 get( )和 containKey( )访问可以查找关键字。例如：

```
Integer n=(Integer) hash_1.get("three");
```

```
if (n!=null {
    System.out.println("three="+n);
      }
```

如果查找的关键字不存在，get 返回 null。

1.8.7 栈类

Java 语言规范中，栈类是向量类的子类，它满足 FIFO(先进后出)的要求。栈类定义的主要方法如下：

```
public Stack( ); // 栈类构造方法。
public Object push (Object item);// 把对象压入栈。
public Object pop ( );// 从栈顶弹出一个对象。
public Object peek( ); // 读栈顶一个对象，但不弹出。
public boolean empty( );// 测试栈是否为空，如是，返回 true，否则，返回 false。
```

介绍完 Java 语言的基础知识、Java 面向对象的编程方法、常用的可重用类，读者就可以编写单机程序了。

第2章　Java网络编程基础

2.1　Java网络编程基本概念

Java的特性之一是灵活方便的网络支持，Java用专门的类来处理较低层次的TCP/IP网络连接。程序员即使从来没有编过网络程序，也能在很短的时间内利用Java方便地访问Internet上的资源。

在介绍Java的网络类之前，我们先简单介绍一下有关TCP/IP协议的基本知识和UNIX下的socket编程，这些内容显得有些繁琐，而Java有关的Socket、ServerSocket类正是封装了这些技术上的细节，为Java用户带来方便。2.1节的目的是使读者对Java有关的Socket、ServerSocket类及最基础的网络编程技术有较深刻的理解。感到有困难的读者可以跳过2.1节，直接阅读2.2节。

2.1.1　TCP/IP协议组

TCP/IP是一组在Internet网络上的不同计算机之间进行通信的协议群，它由TCP（传输控制协议）、IP（Internet协议）、UDP（用户数据报协议）、HTTP(超文本传输协议)、FTP(文件传输协议)、SMTP(简单邮件传输协议)等一系列协议组成。它们从下往上可分为四层结构：

- 物理层　（以太网、令牌环、X.25等）
- 网络层　（IP协议）
- 传输层　（TCP协议、UDP协议）
- 应用层　（HTTP协议、FTP协议、SMTP协议等）

从一台计算机发往另一台计算机的数据都是按以下顺序发送和接收的：

发送数据的计算机先将数据传到应用层，由上往下传送，数据经过每一层时，所使用的协议都给数据加上一个协议头，然后将加上协议头的数据传到下一层，下一层所使用的协议再给它加上一个协议头，继续向下传递，最后由物理层经硬件设备发送到网络上。接收数据的计算机则相反，数据是由下往上传递的，每经过一层时，都剥去相应的协议头，然后继续向上传递。最后传给用户的数据将是剥去所有协议头的，即最原始的数据。

假设你的浏览器想从网络上运行的Web服务器上获取数据，它先将一个HTTP请求发送到TCP层，加上一个TCP端口地址后发往IP层，IP层再给它加上一个IP地址，通过物理层将此请求发往网络上指定的计算机。

在接收数据的一方，每一层都剥去相应的地址信息，然后决定下一步该做什么。物理层在接收到信息后，将数据发往IP层，IP层发现这是一个包后，将剥去IP头的数据发往TCP层，TCP层再将剥去TCP头的数据发送给计算机上运行的HTTP服务代理（即在服务器上运行的服务进程），HTTP代理根据这个请求进行相应的处理，然后将结果按相应的路

径返回给请求者。

2.1.2 套接字（Socket）

套接字用于实现网络上客户程序和服务程序之间的连接。也就是说，网络上两个以双工方式通信的进程之间总有一个连接，这个连接的端点成为套接字。套接字是在一个较低层次上进行通信的。

套接字负责网络上进程之间的通信，客户程序可以向套接字里写入请求，然后服务器会处理这个请求，并把处理结果通过套接口送回。具体来说，一个服务器应用程序一般侦听一个特定端口以等待客户的连接请求，当一个连接请求到达时，客户和服务器建立一个通信连接，在连接过程中，客户被分配一个本地端口号并且与一个Socket连接，客户通过写Socket来通知服务器，以读Socket来获取信息。类似地，服务器也获取一个本地端口号，它需要一个新的端口号来侦听原始端口上的其他连接请求。服务器也给它的本地端口连接一个Socket，通过读写它来与客户通信。

Socket可以根据通信性质分类；这种性质对于用户是可见的。应用程序一般仅在同一类的套接字之间通信。不过只要底层的通信协议允许，不同类型的套接字之间也可以通信。

用户目前可以使用两种套接字，即流套接字和数据报套接字。流套接字提供了双向的、有序的、无重复并且无记录边界的数据流服务。TCP即是一种流套接字协议；数据报套接字支持双向的数据流，但并不保证是可靠、有序、无重复的，也就是说，一个以数据报套接字接收的信息的进程有可能发现信息重复了，或者和发出的顺序不同。数据报套接字的一个重要特点是它保留了记录边界。UDP即是一种数据报套接字协议。

下面我们来介绍一下套接字Socket接口：

1) 创建一个套接字

系统调用创建套接字需有三个整数参数，调用返回一个整数结果：

result = socket (pf, type, protocol)

参数pf指定套接字使用的协议族。也就是说，它指出如何解释所提供的地址。当前的协议族包括TCP/IP互连网(PF_INET)，Xerox公司的PUP互连网(PF_PUP)，Apple计算机公司的Appletalk网络(PF_APPLETALK)和UNIX文件系统(PF_UNIX)以及许多其他协议族。

Type参数指定所需的通信类型。可能的类型包括可靠数据流服务(SOCK_STREAM)和无连接数据报服务(SOCK_DGRAM)，以及允许有特权的用户访问底层协议或网络接口的原始(raw)类型(SOCK_RAW)。另外还有两种类型已列入规划，但还未被实现。

虽然区分协议族和类型的一般方法似乎可以轻易应付各种情况，但事实并非如此。首先，有可能某个指定的协议族不能支持一种或多种可能的协议族类型。例如，UNIX家族有一个被称为管道(pipe)的进程间通信机制，它使用可靠数据流服务，但它却没有用于按序分组投递的机制。因此，并不是所有的协议族和服务类型的结合都具有意义的。其次，一些协协议族具有支持一种服务类型的多种协议。例如，可能一个协议族具有两种无连接数据报服务。为了能在一个协议族中提供多种协议，套接字调用就有了用于选择协议的第三个参数。为使用第三个参数，编程人员必须充分了解协议族，以便知道每个协议提供的服务类型。

由于设计人员竭力在套接字设计中捕捉传统的UNIX操作，他们需要一种模拟UNIX

管理机制的方法，我们没必要了解管道的细节，只需了解管道的一个显著特征：管道与标准网络操作的区别在于管道同时创建了用于通信的两个端点。为提供管道，设计人员增加了一个 socketpair 系统调用，其调用格式如下：

socketpair(pf, type, protocol, sarray)

socketpair比套接字过程多了一个参数sarray。此参数给出了一个二元整数数组的地址

socketpair同时创建两个套接字，并将两个套接字描述符放入 sarray的两个元素中。读者应当明白在使用TCP/IP协议族时，socketpair是没有意义的（这里提到它只是为了我们对套接字的描述完整。）

2) 套接字的继承与终止

UNIX 使用 fork 和 exec 系统调用开始一个新应用程序。这是一个两步过程。第一步，fork创建当前运行程序的一个独立的副本。第二步，用一个所需要的应用程序代替新副本。当程序调用fork时，新建副本继承了对所有打开套接字的访问，正如它继承了对所有打开文件访问。当程序调用 exec 时，新应用程序仍维持对所有打开套接字的访问。我们将看到在主服务器创建从服务器以处理某个连接时就使用了套接字继承。操作系统内部保存了对每个套接字的引用计数器，因而它知道有多少个应用程序（进程）访问了套接字。

新老进程对现有的套接字都有同样的访问权，且都能访问套接字。因此，应当由编程人员负责确保两个进程能有效使用共享套接字。

当进程完成了对套接字的使用就调用 close。close 具有如下形式：

close(socket)

此处，socket 参数指明要关闭的套接字描述符。当进程在任何情况下终止时，系统关闭所有打开的套接字。在系统内部，close调用将减少套接字的引用计数，并在计数为零时删除套接字。

3) 指明本地地址

一个套接字初始创建时并未与任何本地或目的地址关联。对于TCP/IP 协议，这意味着没有分配本地协议端口号，也没有指定目的端口或IP地址。在很多情况下，应用程序并不关心它们使用的本地地址，而希望协议软件为它们选择一个。但是，在某个知名端口上操作的服务进程必须能对系统指定端口。一旦创建了一个套接字，服务器使用 bind 系统调用为套接字建立一个本地地址。bind 具有如下形式：

bind(socket, localaddr, addrlen)

socket 参数是要连接的套接字描述符。参数localaddr是一个指定套接字要连接的本地地址的结构，参数 addrlen 是一个指定地址长度（字节数）的整数。设计人员选择用图 2.1 所示的结构表示地址，而不是将地址简单表示成一个字节序列。

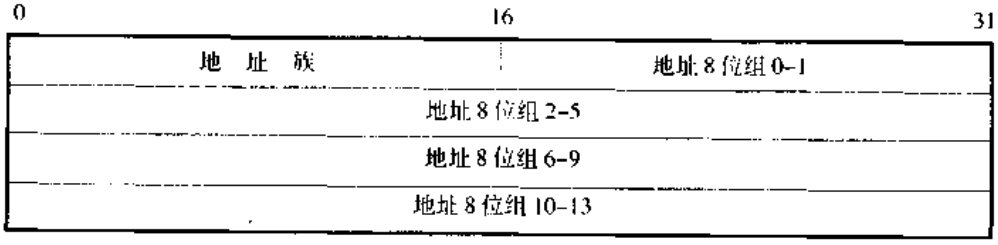

图 2.1 在给套接字接口传递 TCP/IP 地址时使用的 sockaddr 结构

此结构通常被称为sockaddr，它开始是一个识别地址所属协议族的16bit的地址族(ADDRESS FAMILY)字段。随后是一个不超过14个8位组的地址。套接字地址结构在C语言中说明为一个用于所有可能的地址族的结构的联合(union)。

ADDRESS FAMILY字段中的值决定剩下的地址八位组的格式。例如，ADDRESS FAMILY字段中值为2，意味着后续地址的8位组包含一个TCP/IP地址。每个协议族规定它如何使用地址字段中的八位组。对于TCP/IP地址，套接字地址是sockaddr_in。它包括一个IP地址和此地址上的端口号（即一个互连网套接字地址结构可能含有一个IP地址和该地址的一个协议端口）。图2.2说明了一个TCP/IP套接字地址的确切结构。

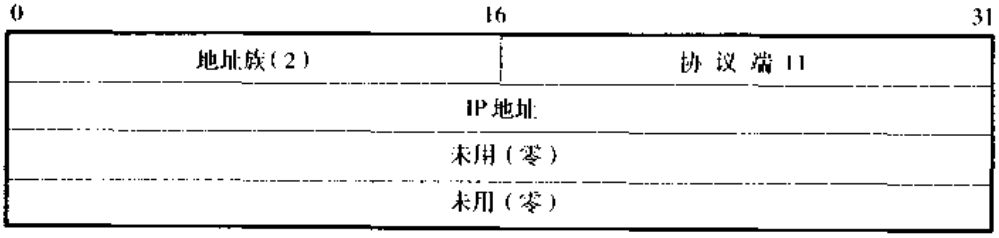

图2.2 使用TCP/IP地址时一个套接字地址结构（sockaddr_in）的格式。此结构包括一个IP地址和此地址上的端口号

虽然可以在调用bind时在地址结构中指定任意值，但不是所有可能的连接都是有效的。例如，调用者可能申请了一个已被另一个程序使用的本地协议端口，或者它可能申请了一个无效的IP地址。在那种情况下，bind调用失败并返回一个差错代码。

4) 将套接字连接到目的地址

初始创建的套接字处于一种未连接的状态，即套接字并未与任何外地目的地址关联。系统调用connect为套接字连接一个永久的目的地址，将它置于连接状态。应用程序在通过可靠数据流套接字传输数据前必须调用connect建立一个连接。用于无连接数据报服务的套接字不需要在使用前建立连接，但如果建立了就使传输数据时可以不必每次都指定目的地址。

connect系统调用具有如下形式：

connect(socket, destaddr, addrlen)

参数socket是要连接的套接字描述符整数。参数destaddr是一个套接字地址结构，它指定要与套接字连接的目的地址。参数addrlen指定按字计算的目的地址长度。

connet的语法与底层协议有关。在PF_INET协议族中选择可靠数据流传输服务意味着选择了TCP。在此情况下，connect构造一个与目的地的TCP连接，并在不能构造连接时返回一个差错代码。如果是无连接服务，connect只在本地存储目的地址，不再完成其他功能。

5) 通过套接字发送数据

一旦一个应用程序已建立了一个套接字，它就可以使用套接字传输数据。有五个可能的操作系统调用可供选择：send，sendto，sendmsg，write和writev。send，write和writev只用于已建连的套接字，因为他们不允许调用者指定目的地址。这三个调用间的区别很小。write带有三个参数：

write(socket, buffer, length)

参数socket包含一个整数套接字描述符(write也可用于其他类型的描述符)。参数buffer包含要发送数据的地址。参数length指明要发送数据的长度。对write调用在数据能被传输前一直阻断(例如,若套接字使用的内部系统缓冲区装满就阻断)。与UNIX中许多系统调用一样,write给调用它的应用程序返回一个差错代码,使得编程人员知道操作是否成功。

writev系统调用与write类似,只是writev使用"收集写(gather write)"的形式,这使应用程序写一个报文时可以不将报文拷贝到内存的相邻字节区域。writev具有如下格式:

writev(socket, iovector, vectorlen)

参数iovector给出一个iovec类型的数组地址,它含有一个指向各个构成报文的字节块的指针序列。如图2.3所示,一个块长度伴有一个块指针。参数vectorlen指定iovector中指针项的数目

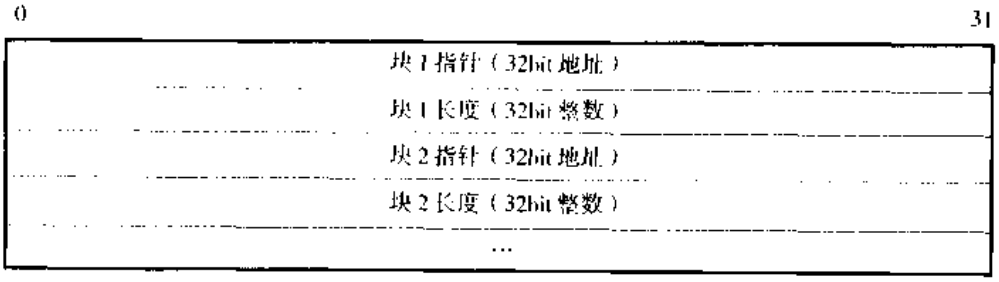

图2.3 用于writev和readv的iovec类型的iovector的格式

send系统调用具有如下形式:

send(socket, message, length, flags)

此处的socket参数指定所用的套接字描述符,参数message给出要发送数据的地址,参数length说明要发送数据的字节数,参数flags控制传输。flags的一个值允许发送者指定报文必须在支持带外传输的套接字上带外发送。flags的另一个值允许调用者请求不使用本地路由发送报文。目的是允许调用者控制路由选择,使得用户可以编写网络调试软件。当然,并不是所有的套接字都支持来自任一程序的所有请求。一些请求需要调用程序具有特殊的权限;另一些请求简单地被处理为不在任何套接字上支持它们。

系统调用sendto和sendmsg允许调用者在无连接的套接字上发送报文,因为他们都需要调用者指定目的地址。sendto将目的地址作为参数,它具有如下格式:

sendto(socket, message, length, flags, destaddr, sddrlen)

前四个参数与send系统调用使用的完全相同。最后两个参数指定一个目的地址并给出地址长度。参数destaddr使用图2.2中定义的sockaddr_in结构指定目的地址。

编程人员可能选择使用sendmsg系统调用,以免sendto所需的一长串参数使得程序效率不高且难读。sendmsg具有如下格式:

semdmsg(socket, messagestruct, flags)

此处参数messagestruct是图2.4所示的结构。该结构包含要发送报文的信息及其长度、目的地址和地址长度。这一调用尤其有用,因为有一个相应的输入操作将产生完全相同格式的报文结构。

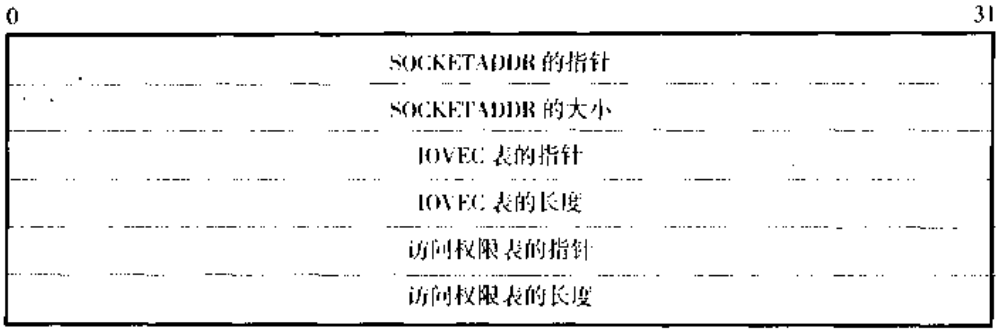

图 2.4　sendmsg 使用的 messagestructure 报文结构的格式

6) 通过套接字接收数据

与五个不同的输出操作类似，BSD UNIX 提供五个系统调用，进程可使用它们通过套接字接收数据：read，recv，recvfrom 和 recvmsg。传统的 UNIX 输入操作 read 只能在套接字已建连接时使用。它具有如下形式：

read(descriptor、buffer、length)

此处参数 descriptor 给出了从中读取数据的一个套接字的整数描述符，buffer 指定存储数据的存储区的地址，参数 length 指定要读取的最大字节数。

read 的替代形式是 readv，它允许调用者使用一种“分散读取”（scatter read）方式的接口，即可将要读的数据放在不相邻的位置。readv 的格式如下：

readv(descriptor、iovector、vectorlen)

参数 iovector 给出 iovec 类型结构的地址，它包含一组指向存储待读数据的内存块的指针。参数 vectorlen 指定 iovector 中数据项的数目。

除了传统的输入操作，还有三种用于网络报文输入的系统调用。进程调用 recv 从已建立的套接字接收数据。它的格式如下：

recv(socket, buffer, length, flags)

参数 socket 指定从中接收数据的套接字描述符。参数 buffer 指定存放报文的内存缓冲区地址，参数 length 指定缓冲区和长度。参数 flags 的可能值之一允许调用者通过提取下一个到来报文的副本预览数据，但不从套接字中删除此报文。

系统调用 recvfrom 允许调用者指定从一个无连接套接字输入数据。它包括允许调用者指定在何处记录发送者地址的其余参数。格式如下：

recvfrom(socket、buffer、length、flags、fromaddr、addrlen)

两个额外参数 fromaddr 和 addrlen 分别是套接字地址结构的指针和一个整数。操作系统使用 fromaddr 记录报文发送者的地址，并使用 addrlen 记录发送者地址的长度。注意上面讨论过的输出操作 sendto，其地址格式与 recvfrom 完全一样。因此，很容易与发送响应。

用于输入的最后一个系统调用是 recvmsg，它与 sendmsg 输出操作类似。recvmsg 操作类似 recvfrom，但需要的参数更少。它的格式为：

recvmsg(socket、messagestruct、flags)

此处参数 messagestruct 给出存放到来的报文和发送者地址的结构的地址。recvmsg 生成的结构与 sendmsg 使用的结构完全一样，这使得它们很好地可成对操作。

7) 获得本地和远地套接字地址

我们说过新创建的进程从创建它们的进程处继承打开的套接字。有时，新创建的进程需要判断套接字连到了哪个目的地址。进程也可能希望判断套接字的本地地址。有两个系统调用提供这些消息：getpeername 和 getsockname（尽管名字如此，其实两个都是处理我们意识中的“地址”）。

进程调用 getpeername 判断套接字连接的对端（也就是远端）地址。它具有如下格式：

getpeername(socket, desaddr, addrlen)

参数 socket 指定需要判定对端地址的套接字。参数 destaddr 是接收的套接字地址的 sockaddr结构类型的指针（见图2.1）。最后，参数 addrlen 是接收的套接字地址长度的一个整数指针。getpeername只用于已建连的套接字。

系统调用 getsockname 返回套接字相连的本地地址。它具有的格式如下：

getsockname(socket, localaddr, addrlen)

参数 socket 指定需要判定本地地址的套接字。参数 localaddr 是包含套接字地址的 sockaddr 结构类型的指针。最后，参数 addrlen 是包含套接字地址长度的一个整数指针。getpeername 只用于已建连的套接字。

8) 获得并设置套接字选项

除了将套接字连接到本地地址或连接到目的地址，还需要有一种允许应用程序控制套接字的机制。例如，当使用有超时和重传输机制的协议时，应用程序可能要设置超时参数。它也可能想控制缓冲区空间的分配，判断是否允许发送广播，或控制带外数据的处理。编程人员决定与其给每个新控制操作增加一个新系统调用，还不如构造一个单一的控制机制。该控制机制有两个操作：getsockopt 和 setsockopt。

系统调用getsockopt允许应用程序请求获取套接字有关的信息。调用者指定套接字、感兴趣的选项和存放请求信息的位置。操作系统检查套接字用的内部数据结构，并将被请求的信息传给调用者。调用格式如下：

getsockopt(socket, level, optionid, optionval, length)

参数 socket 指定需要信息的套接字。参数 level 标识操作是用于套接字本身还是底层协议。参数 optionid 指定申请的是哪一个选项。optionval 和 length 一对参数定义两个指针。第一个给出系统存放请求值的缓冲区地址，第二个给出系统存放选项值长度的一个整数的地址。

系统调用setsockopt 允许应用程序使用通过 getsockopt 获取一组值去设置套接字选项。调用者指定一个要设置选项的套接字、要改变的选项和选项值。其调用格式如下：

setsockopt(socket, level, optionid, optionval, length)

此处的参数与 getsockopt( )的很像，只是 setsockopt( )的参数 length 是传给系统的选项长度。调用者必须提供选项的一个合理值和该值的正确长度。当然不是所有的选项都可用于所有的套接字。各个请求的正确性和语义与套接字的当前状态和使用的底层协议有关。

9) 指明服务器的队列长度

套接字使用的选项中有一个使用得很频繁，以至于专门为它构造了一个系统调用。为理解它产生的原因，我们来考虑一个服务器。服务器创建一个套接字，将它连接到一个知名的协议端口，并等候请求。如果一个服务器使用可靠数据流传输，或者如果计算一个响

应花费的时间不算少，可能在服务器完成对前一个请求的响应前又到达一个新请求。为避免协议拒绝或丢弃到来的请求，服务器必须告知底层协议软件它希望把来不及处理的请求排队。

系统调用listen允许服务器为到来的请求准备一个套接字。根据底层协议，listen将套接字置于被动准备接受连接的模式。当服务器发出listen时，它也通知操作系统协议软件应将同时到达套接字的请求排队。格式是：

listen(socket, qlength)

参数socket给出服务器准备的套接字描述符，参数qlength指定套接字上请求队列的长度。调用listen后，系统至多将qlength个连接请求排队。如果当一个请求到达时队列已满，操作系统将通过丢弃此请求拒绝连接。listen只用于已选择可靠传输服务的套接字。

10) 服务进程中怎样接受连接

正如我们所见，服务进程使用系统调用socket，bind和listen创建一个套接字，将它连接到知名的端口，并指定连接请求的排队长度。请注意，当调用bind将套接字与一个知名的协议端口连接时，套接字并没有连接到外部目的地址。实际上，外部目的地址必须指定一个通配符，从而允许套接字可接收来自任意客户机的连接请求。

一旦套接字已建立，服务进程需要等待一个连接，它使用系统调用accept做此工作。调用accept将被阻断直到到达一个连接请求。它的格式如下：

newsock = accept(socket, addr, addrlen)

参数socket指定等候连接的套接字描述符。参数addr是sockaddr结构类型的指针，参数addrlen是一个整数的指针。当一个请求到达时，系统将发出请求的客户机地址填入参数addr，将addrlen设为地址的长度。最后，系统创建一个新套接字，其目的地址已连接到发请求的客户机，并给调用者返回一个新的套接字描述符。原来那个套接字的目的地址仍为通配符，并保持打开。因此，主服务器可继续在原套接字上接受其他的连接。

accept在一个建立连接请求到达时返回。服务进程可交互地处理或并行处理请求。在交互方式中，服务进程自己处理请求，关闭新套接字，然后调用accept获取新连接请求。在并行方式中，调用accept返回后，主服务进程创建一个从服务进程处理请求（UNIX术语表达就是它创建一个子进程处理请求）。从进程继承了新套接字的副本，因而它可对请求进行服务。从进程完成任务后关闭套接字并终止。原（主）服务进程在启动从进程后关闭新套接字的副本。然后它调用accept获取下一个连接请求。

服务进程的并发设计似乎令人困惑，因为多个进程使用了同一个本地协议端口。理解这一机制的关键在于底层协议处理协议端口的方式。因此，有多少个进程使用同一个本地协议端口无关紧要，只要它们连接到了不同的目的地。在一个并发服务进程中，每个客户机对应一个进程，并有一个额外进程用来接受连接。主服务器进程使用的套接字将通配符用作外部目的地址，这允许它连接到任意外部结点。一旦一个TCP报文到达，它就被送到与报文来源连接的套接字。如果不存在此套接字，报文将被送到用通配符作外部目的地址的套接字。然而由于通配符作外部目的地址的套接字没有一个打开的连接，它只能处理请求连接的TCP报文。

11) 处理多重服务和服务器

BSD UNIX接口提供另一种可能令人感兴趣的服务进程设计，因为它允许单个过程在

多个套接字上等待连接。使这一设计成为可能的系统调用是select，它不仅用于套接字上的通讯，通常还用于I/O。select具有格式：

nready = select(ndesc, indesc, outdesc, exedesc, timeout)

一般情况下，select调用将被阻断直到一组文件描述符中某一个已准备好。参数ndesc指定有多少个描述符应被检查（被检查的描述符总是在0和ndesc-1之间）。参数indesc是指向一个屏蔽位的指针，该屏蔽位指定要进行输入检查的文件描述符，参数outdesc是一个指定要进行输出检查的文件描述符的屏蔽位的指针，参数exedesc是一个指定要进行例外检查的文件描述符的屏蔽位的指针。最后，如果参数timeout非零，它就是一个指定在返回调用者前等待连接的最长时间值的整数指针。若timeout为零，就使得调用阻断直到有一个描述符准备好。由于timeout参数包含超时值的指针而本身并不是整数，进程可通过传递值为零的整数地址申请零时延（即一个进程可轮询查看I/O是否准备好）。

select调用返回指定组中I/O已准备好的描述符数目。它也可改变由indesc，outdesc和exedesc指定的屏蔽位，通知应用程序哪一个已选择的文件描述符已准备好。因此，调用select之前，调用者必须打开与检查描述符对应的那些屏蔽位。在调用过程中，所有保持为1的位与I/O准备好的文件描述符相对应。

为了在多个套接字上通信，一个进程首先创建它所需的所有套接字，然后使用select判断哪一个套接字最先为I/O准备好。一旦发现有一个套接字已准备好，进程就使用上面定义的输入或输出过程进行通信。

12) 获取与设置主机名字

BSD UNIX操作系统保留了一个内部主机名。对于Internet上的机器，内部名经常被选作该机主要网络接口的域名。系统调用gethostname允许用户进程访问主机名，系统调用sethostname允许有特权的进程设置主机名。gethostname格式为：

gethostname(name,length)

参数name给出存放名字的字节数组的地址，参数length是一个指定名字数组长度的整数。为设置主机名，有特权的进程发出的调用格式为：

sethostname(name,length)

参数name给出存放名字的字节数组的地址，参数length是一个给出名字数组长度的整数。

13) 获取与设置内部主机域

操作系统保留了一个指定机器所属命名域的字符串。当1000个网点获得部分域名空间的访问权限时，它生成一个识别它所在空间部分的字符串，并使用此串作为域名。例如，在如下面的域：

.dragon.cumt.edu

有特权的进程使用系统调用setdomainname，其格式如下：

setdomainname(name, length)

参数name给出包含域名的字节数组的地址，参数length是给出名字长度的一个整数。用户进程调用getdomainname向系统查询域名。它的格式如下：

getdomainname(name, length)

参数name给出存放域名的字节数组的地址，参数length中给出数组长度的一个整数。

14) BSD UNIX 网络库的调用

除了上面描述的系统调用，BSD UNIX 还提供一组例程来获得组网信息。图 2.5 显示了系统调用和库例程间的区别。系统调用将控制传给计算机操作系统，而例程库与其他过程类似，编程人员将他们捆绑到一个程序中。

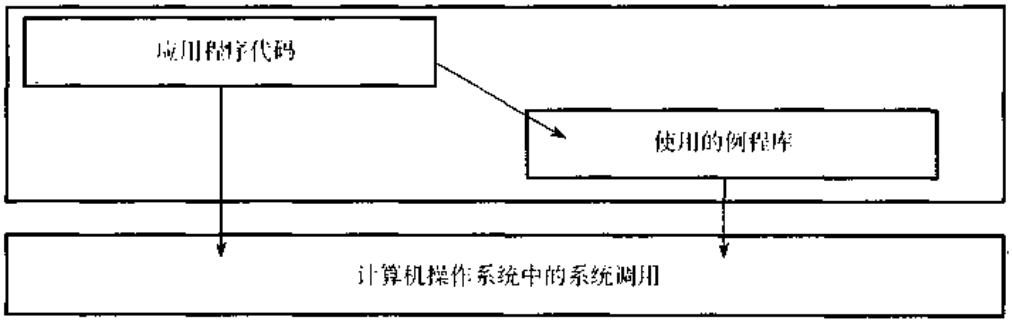

图 2.5　系统调用与库例程

BSD UNIX的许多例程提供系统信息服务，从而允许一个进程获得机器名和网络服务，协议端口号与其他相关信息。例如，一组库例程提供对网络服务的数据库访问。我们将服务数据库中的每一项看成一个三元组，每个三元组包括（人们可读的）网络名、支持服务的协议和服务用的协议端口。库例程允许进程获取任何一项的信息。

以下几节将介绍一组库例程，描述其使用目的以及如何使用这些例程。正如我们所见，各组库例程按一种模式提供对顺序数据库的访问。

库例程被捆绑到应用程序中，而系统调用是操作系统的一部分，这是二者的区别。一个程序可调用这两者；库例程可调用其他库例程或系统调用。这些例程操作顺序为：建立到数据库（数据文件）的连接、一次获取一项内容，和关闭连接。用于这三个操作的例程被命名为setXent、getXent 和 endXent，此处 X 表示数据库的名字。例如，用于主机数据库的库例程被命名为 sethostent、gethostent 和 endhostent。

15) 网络字节序转换程序

BSD UNIX 提供了四个库函数用于本地机器字节序和网络标准字节间的转换。为使程序可移植，编写程序时应在每次从本地机器拷贝一个整数值到网络分组时，或每次从网络分组中拷贝一个值到本地机器中，都要调用转换例程。

所有这四个转换例程都是函数，他们用一个值作参数，并返回字节重排后的一个新值。例如，为将一个短整数（2 个字节）从网络字节序转换为本地主机字节序，编程人员调用 ntohs(网络到主机短整数)。格式为：

localshort = ntohs(netshort)

参数 netshort 是一个网络标准字节序的 2 个字节（16bit）整数，结果 localshort 是本地主机字节序的短整数

· UNIX 称 4 字节（32bit）整数为 longs（长整数）。函数 ntohl（网络到主机长整数）将 4字节的长整数从网络标准序转换到本地主机字节序。编程人员调用ntohl 函数，将网络字

节的长整数作为参数：

```
locallong = ntohl(netlong)
```

还有两个相似的函数允许编程人员完成本地主机字节到网络字节序的转换。函数htons将一个本地主机字节的短整数（2个字节）转换为网络标准字节的短整数。程序调用htons函数的格式为：

```
netshort = htons (localshort)
```

最后一个转换函数是htonl，它转换长整数到网络标准字节序。与其他一样，htonl是一个函数：

```
netlong = htonl(locallong)
```

很明显，转换函数能保持如下的数学关系：

```
netshort = htons(ntohs(netshort))
```

和

```
localshort = ntohs(htons(localshort))
```

(6) IP地址处理程序

由于许多程序要在32bit IP地址和对应的点分十进制表示间进行转换，BSD UNIX库包括完成此转换的实用例程。过程inet_addr和inet_network都将点分十进制格式转换为网络字节序的32bit IP地址。Inet_addr形成一个32bit主机IP地址；Inet_network形成主机地址部分为零的网络地址。它们具有如下格式：

```
address = inet_addr(string    )
```

和

```
address = inet_network(string        )
```

此处参数string给出一个ASCII字符串的地址，其中包含以点分十进制格式表示的数字。点分十进制格式有四个数字单元，中间用点隔开。如果4个数字都出现，每个数字对应其表示的32bit IP地址中的各个字节。如果少于4个数字，就扩展最后一个数字单元以填充剩余的字节。

过程inet_ntoa完成一个32bit整数到一个点分十进制格式的ASCII字符串的映射。它的格式为：

```
str = inet_ntoa(internetaddr)
```

此处参数internetaddr是一个网络字节序的32bit IP地址，str是相应ASCII字符串地址。

通常处理IP地址的程序必须将网络地址与网络上本地址结合。过程inet_makeaddr完成这种结合。它具有格式：

```
internetaddr = inet_makeaddr(net, local)
```

参数net是主机字节序的32比特IP地址，参数local是代表网络上本地主机地址的整数，local也是主机字节序的。

过程inet_netof和inet_lnaof提供inet_makeaddr的反向功能，即将一个IP地址的网络部分和本地部分分开。它们的格式为：

```
net = inet_netof(internetaddr)
```

和

```
local = inet_lnaof(internetaddr)
```

此处参数 internetaddr 是网络字节序的 32bit IP 地址，返回结果是主机字节序的。

17) 访问域名系统

BSD UNIX 中 TCP/IP 域名系统的接口由五个库过程组成。调用这些例程的应用程序成为一个域名系统的客户机，它发送一个或多个服务器请求并等待响应。

一般思路是一个程序形成一个查询请求，发送到一个服务器，并等待响应。由于存在许多选项，例程只有几个基本参数并使用一个用于存放其他参数的全局结构 res 。例如，res 中的一个字段允许调试消息，而另一个字段控制查询代码使用 UDP 还是 TCP。res 中的大多数字段有合理的默认值，因此可在使用例程时不改变 res。

一个程序在使用其他过程之前调用 res_init。该调用没有参数：

res_init ( )

res_init 读取一个文件，它包含诸如运行域名服务器的机器名信息，并将结果存入全局结构 res。

过程 res_mkquery 形成域名查询，并将查询放到一个内存缓冲区中。调用格式是：

res_mkquery(op,dname, class, type, data, datalen, newrr, buffer, buflen)

前七个参数直接与域名查询中的字段对应。参数 op 指定请求的操作，dname 给出包含域名的字符数组成的地址，class 给出查询种类的整数，type 给出查询类型的整数， data 给出包含在查询中的数据数组的地址，datalen 给出数据长度的整数。除了库过程， UNIX 还为应用程序提供为重要值定义符号常量的能力，因此，编程人员不必了解协议细节就可使用域名系统。最后两个参数 buffer 和 buflen，分别给出存放查询的缓冲区地址和长度。最后，在当前实现中，参数 newrr 没有使用。

一旦程序形成一个查询，它就调用 res_send 将其发送给一个名字服务器并获取响应。格式为：

res_send(buffer, buflen, answer, anslen)

参数 buffer 是保存待发送报文的存储区指针（可以推测，应用程序调用过程 res_mkquery 形成报文）。参数 buflen 是指出报文长度的整数。参数 answer 给出将写入响应的存储的地址，整数参数 anslen 指定响应区域的长度。

除了形成和发送查询的例程，BSD UNIX 库例程还包含两个传统 ASCII 串与查询中使用的压缩格式间转换域名的例程。过程 dn_expand 将一个压缩域名扩充为完整的 ASCII 串。它具有格式：

dn_expand(msg, eom, compressed, full, fullen)

参数 msg 给出包含要扩充名字的域名报文的地址，eom 指定报文结束（ end -of-message ）的界限，超出此限就不能再扩充。参数 compressed 是压缩名字的首字节指针，参数 full 是将写入扩充名的数组的指针，参数 fullen 是指定数组长度的整数。

产生压缩名比扩充压缩名更复杂，因为压缩涉及去掉公共的后缀。在压缩名字时，客户机必须保存以前出现过的后缀的记录。过程 dn_comp 通过比较后缀与以前用过的后缀列表并去掉可能最长的后缀来压缩一个完整的域名。调用格式为：

dn_comp(full, compressed, cmprlen, prevptrs, lastptr)

参数 full 给出一个完整域名的地址。参数 compressed 指向存放压缩名的字节数组，参数 cmprlen 指定数组的长度。参数 prevptrs 是指向以前各个压缩后缀的指针的数组的地址，

lastptr指向此指针数组尾。通常，如果使用了一个新后缀才调用dn_comp压缩域名和更新prevptrs。

我们也使用过程dn_comp将域名从ASCII字符串转换为非压缩的内部格式(也就是说不去掉后缀)。为此，进程调用dn_comp时只需将参数prevptrs设为NULL(即为0)。

18) 获取主机信息

有些库过程允许进程给出主机的域名或IP地址就可查询此主机有关信息。当一个可访问域名服务器的机器使用这些例程时，进程可通过发出一个请求到服务器并等待响应成为一个客户机。当这些例程用于不能访问域名系统的系统上时(例如不在Internet上的主机)，例程就从存放于辅助存储器的数据库获取所需的信息。

函数gethostname用于主机域名作参数，并返回指向该主机信息的结构的指针。调用格式如下：

```
ptr = gethostname(namestr)
```

参数namestr是包含主机域名的字符串的指针。返回值ptr指向包含下列信息的一个结构：正式主机名、主机登记的别名列表、主机地址类型(即地址是否是IP地址)、地址长和主机的一个或多个地址的列表。更多细节可见UNIX编程人员手册。

函数gethostbyaddr与gethostbyname产生同样的信息。两者的区别在于gethostbyaddr采用一个主机地址作参数：

```
ptr = gethostbyaddr(addr, len, type)
```

参数addr是包含一个主机地址的字节序列的指针。参数len是给出地址长度的整数，参数type是指定地址类型(如它是一个IP地址)的一个整数。

正如前面提到的，过程sethost，gethostent和endhostent提供对主机数据库的顺序访问。

19) 获取网络信息

运行BSD UNIX的主机或者使用域名系统，或者保留一个其互连网上的简单网络数据库。网络库例程包括五个允许进程访问网络数据库的例程。例程getnetbyname根据给出的网络域名获取或格式化数据库中某项的内容。调用格式为：

```
ptr = getnetbyname(name)
```

此处参数name是包含需获取信息的网络名的字符串的指针。返回值是一个结构的指针，该结构内容包含网络和正式名字、登记别名列表、一个整数地址类型和一个32位网络地址(即将主机部分设为零的IP地址)。

一个进程在它需要查询有关给定地址网络的信息时调用库例程getnetbyaddr。调用格式为：

```
ptr = getnetbyaddr(netaddr, addrtype)
```

参数netaddr是一个32 bit网络地址，参数addrtype是给出netaddr类型的一个整数。过程setnetent，getnetent和endnetent提供对网络数据库的顺序访问。

20) 获取协议信息

还有五个库例程提供对某个机器上使用的协议数据库的访问。每个协议都有一个正式名、登记名和一个正式协议号。过程getprotobyname允许调用者获取给定名字的协议的有关信息：

```
ptr = getprotobyname(name)
```

此处参数name是包含要获取信息的协议名的字符串的指针，返回值是一个结构的指针，该结构内容包含协议的正式名字、登记别名列表、分配给此协议的唯一整数值。

过程getprotobynumber允许进程使用协议号作关键字搜索协议信息：

```
ptr = getprotobynumber(number)
```

最后，过程setprotoent，getprotoent和endprotoent提供对协议数据库的顺序访问。

21) 获取网络服务信息

有一些端口号是为网络服务保留的。例如，TCP端口43是为whois服务保留的。whois允许一个机器上的客户与另一个机器上的服务器联系，并获取在服务器的有帐号的用户的有关信息。服务例程库中whois项指定服务名为whois，协议为TCP，且协议端口号为43。有五个服务例程可用于获取有关服务和使用协议端口的信息。

例程getserverbyname映射一个命名服务到一个端口上：

```
ptr = getservbyname( name, proto)
```

此处参数name指定包含需获取信息的服务名的字符串的地址，且整型参数proto指定该服务使用的协议。典型情况下，协议仅限于TCP和UDP。返回值是一个结构的指针，该结构内容包含服务名、别名列表、服务使用协议的标识和分配给此服务的协议端口号（整数）。

例程getservbyaddr允许调用者给出分配给某服务的端口号就可以从服务例程库获取记录项。调用格式为：

```
ptr = getservbyaddr(port, proto)
```

参数port是分配给服务的整型协议端口号，且参数proto指定服务所用的协议。过程setservent、getservent和endservent为进程提供对服务数据库的顺序访问。

2.1.3 端口

端口是一个逻辑概念。主机每一服务者运行在该主机的一个对外开放的端口上。一个主机上可以有多种服务者，也就是有多个端口。程序员可以在创建自己的服务程序时使用其他端口（即除了系统默认的端口）。端口常以数字编号，作为客户可指定一个端口号，以这个端口号码连接服务代理以接收服务。例如telnet 202.114.0.241 5555，通过这条命令，即可远程登录到华中理工大学的一台服务代理，通过5555端口参与一个网络mud游戏。

2.2 使用TCP协议的Socket网络编程

2.2.1 一对一的Socket C/S通信

TCP是一种可靠的、基于连接的网络协议，在Internet上大都使用TCP/IP协议进行互联。网络上的两进程采用C/S模式进行通信。当两台主机准备进行交谈时，都必须建立一个Socket，其中一方作为服务器打开一个Socket并侦听来自网络的连接请求，另一方作为客户，它向网络上的服务器发送请求，通过Socket与服务器传递信息，要建立连接，只需指定主机的IP地址和端口号即可。

2.2.2 TCP协议通信的服务方实现

服务程序工作在服务器的某个端口上，一旦启动服务，它将在这个端口上倾听，等待客户程序发来的请求。服务器的套接字用服务器套接字类（ServerSocket）来建立。假设服务器工作在端口8000上，以下命令：

```
ServerSocket svrsoc=new ServerSocket(8000);
```

建立了一个服务者svrsoc，它监视端口8000。命令

```
Socket soc=svrsoc.accept( );
```

让服务者永远等待，直到客户连接到该端口。一旦有客户送来正确请求，连接至该端口，accept( )方法就返回一个Socket对象，表示已建立好连接，可用Socket的对象soc获得一个输入／输出流，如下所示：

```
DataInputStream in=new DataInputStream(soc.getInputStream( ));
PrintStream    out=new PrintStream(soc.getOutputStream( ));
```

这里创建了数据输入流类的实例in和输出流类的实例out，是服务者用于从客户接受输入信息和向客户程序发送信息所用，同样在客户端，也应该建立这两个实例，用来与服务程序进行双向通信。服务者向输出流发送的所有信息都成为客户的输入信息，而客户程序的输出都送入服务者的输入流。

获取客户机的IP地址，并在服务者窗口中显示客户机的地址信息：

```
clientIP=soc.getInetAddress( );
System.out.println("Client's IP address:"+clientIP);
```

println( )是输出流类的一个方法，下行向客户送一句问候：

```
out.println("Welcome!...");
```

当用远程登录通过端口8000连接到该服务者时，客户终端屏幕上将接受到上述信息。

该简单服务中，每次只读入一行客户的输入，并回显该行，以表明服务者接收了客户的输入

readLine( )是数据输入流类中的一个方法，用于服务者或客户从对方读入一个输入流信息：

```
String str=in.readLine( );
while (!str.equals("quit"))
{
  System.out.println("Client said:"+str0);
  str=in.readLine( );
}
```

不断循环以上代码，直到客户输入“quit”或者str为null为止。最后在退出前，关闭输入输出流及Socket：

```
    System.out.println("Client want to leave.");
    finally {
        in.close( );
```

```
            out.close( );
            soc.close( );
            svrsoc.close( );
        }
```

这就是一个简单的回应服务者的工作过程。每个服务者，如HTTP Web服务者，都是不停地执行下列循环：

(1) 通过输入流从客户获得命令；

(2) 以某种方式获取该信息；

(3) 通过输出流将信息送给客户。

2.2.3 TCP协议通信的客户方实现

客户机先创建一个指向固定主机的固定端口的Socket，假如上述服务者程序在本机"localhost"上，则以下命令：

```
Socket soc=new Socket("localhost",8000);
```

建立了客户到服务者的连接，两端进行通信的通路即建立。当服务者接收该连接请求时，Socket实例soc即建立，同样，从该Socket实例中获取输入和输出流：

```
        in=new DataInputStream(svrsoc.getInputStream( ));
    out=new PrintStream(svrsoc.getOutputStream( ));
```

输入/输出流建立后，客户首先从服务器读取发来的"Welcome!..."信息，显示在窗口中：

```
strin=in.readLine( );
System.out.println("Server said:"+strin);
```

这两行命令执行后，窗口中应显示出服务者的欢迎信息和客户机系统输出的信息。

客户向服务者发送的数据流从键盘获取：

```
sysin=new DataInputStream(System.in);
strout=sysin.readLine( );
```

当键盘输入不是"quit"时，将键盘输入的数据写入输出流中、并发送出去，然后继续从键盘获取输入数据：

```
out.println(strout);
strout=sysin.readLine( );
```

不断循环上述两行命令，直到键盘输入"quit"时，先将其传送给服务者，然后关闭输入/输出流和Socket：

```
out.println(strout);
in.close( );
out.close( );
sysin.close( );
svrsoc.close( );
```

该客户是与同一台主机上的回应服务者进行通信，先运行服务程序，再运行客户程

序，运行结果如下：

在服务进程窗口中显示：

Client's IP address:127.0.0.1

Client said:Hello!

Client want to leave.

在客户进程窗口中显示：

Connecting to the Server...

Server said:Welcome!...

Hello!

quit

其中后面三行是客户的键盘输入。如果要在网络中的两台不同计算机之间进行通信，只需将 Socket( )中传递的参数“localhost”改成相应的主机名或 IP 即可。

图 2.6 是基于无连接的服务者客户程序流程图。

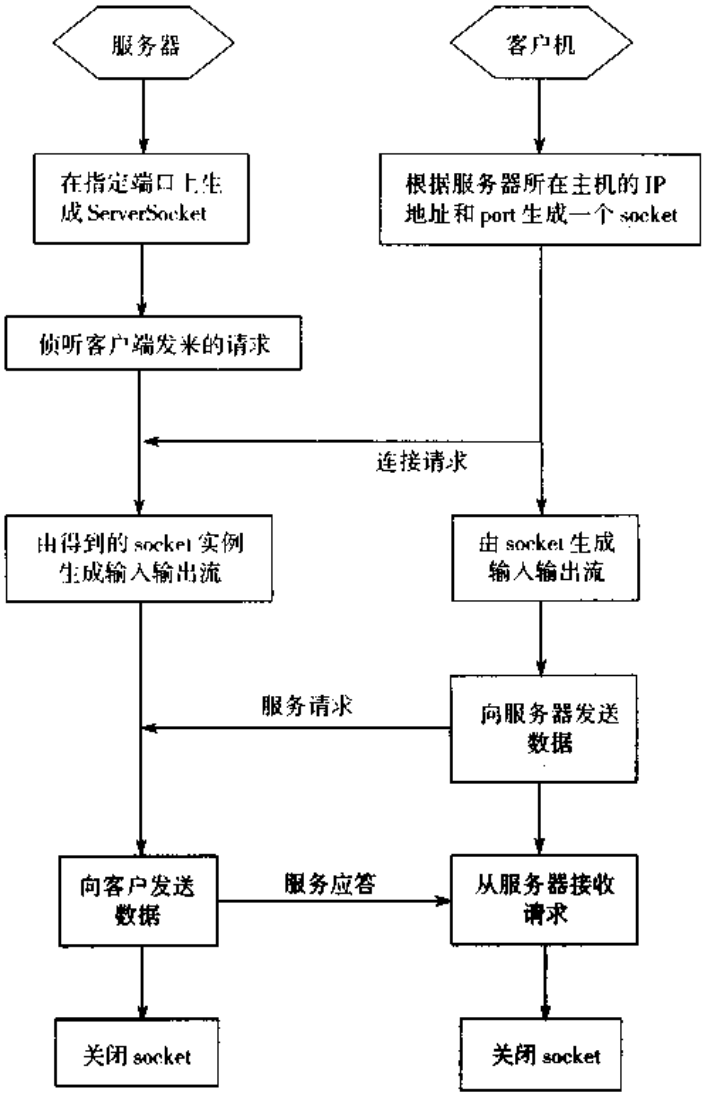

图 2.6　基于无连接的服务者客户流程图

2.2.3.1 API 注释：套接口类(java.net.Socket)

1) Socket(String host, int port) throws UnknownHostException,IOException

创建一个流套接口（即 Socket 实体对象），并将其连接至特定主机的特定端口上。

参数：host　主机名

　　port　端口号

2) Socket(String host.int port.boolean stream)

构造一个套接口（即 Socket 实体对象），并把它连接到特定主机的特定端口上。而此套接口是流套接口还是数据报（datagram）套接口，则是由最后一个参数 stream 决定的。

参数：host　主机名

　　port　端口号

　　stream　用于决定生成的套接口是流套接口还是数据报套接口（datagram socket）。

3) Socket(InetAddress address,int port)

构造一个流套接口（即 Socket 实体对象），并把它连接到特定主机的特定地址上。

参数：address　特定的地址

　　port　端口

4) Socket(InetAddress address,int port boolean stream)

构造一个流套接口（即 Socket 实体对象），并把它连接到特定端口的特定地址上。而此套接口是流套接口还是数据报（datagram）套接口，则是由最后一个参数 stream 决定的。

参数：host　主机名

　　port　端口号

　　stream　用于决定生成的套接口是流套接口还是数据报套接口（datagram socket）。

5) InetAddress getInetAddress( )

返回该套接口（socket）所连接的地址。

6) int getPort( )

返回该套接口（socket）所连接的远程端口。

7) synchronized void close( ) throws IOException

关闭套接口

8) InputStream getInputStream( ) throws IOExcepiton

获得从套接口读入数据的输入流。

注：DataInputStream 为 InputStream 的子类。

9) OutputStream getOutputStream( ) throws IOException

获得向套接口进行写操作的输出流。

注：PrintStream 为 OutputStream 的子类。

2.2.3.2 API 注释：服务者套接口类（java.net.ServerSocket）

1) ServerSocket(int port) throws IOException

在指定的端口上构造一个服务者套接口，即构造一个 ServerSocket 实体对象。

参数：port　端口号

2) ServerSocket(int port, int count)

构造一个服务者套接口，即构造一个ServerSocket实体对象，并且该对象是与指定的当地端口相连接的。此外，可以对它进行监听。用户也可以通过将port设为0来将该对象与一个匿名端口相连接。

参数：port: 端口号

count 对该ServerSocket实体对象与端口间的连接进行监听的次数。

3) Socket accept( ) throws IOException

等待一个连接，该方法将阻塞当前线程，直到连接成功。该方法返回一个套接口类(Socket)对象，通过该对象，程序与连接的客户进行通信。

4) void close( ) throws IOException

关闭套接口。

2.2.3.3 源程序清单及注释：

TCP服务程序：

```
import java.io.*;
import java.net.*;
public class tcpserver
{
    static public void main(String args[ ])
            throws IOException
    {
        ServerSocket svrsoc=null;
        Socket soc=null;
        DataInputStream in=null;
        PrintStream out=null;
        InetAddress clientIP=null;
        String str=null;
        try
        {
            svrsoc=new ServerSocket(8000);
            soc=svrsoc.accept( );
            in=new DataInputStream(soc.getInputStream( ));
            out=new PrintStream(soc.getOutputStream( ));
            clientIP=soc.getInetAddress( );
            System.out.println("Client's IP address:"+clientIP);
            out.println("Welcome!...");
            str=in.readLine( );
            while(!str.equals("quit"))
            {
                System.out.println("Client said:"+str);
                str=in.readLine( );
```

```
                }
                System.out.println("Client want to leave.");
            }
            catch(Exception e)
            {
                System.out.println("Error:"+e);
            }
            finally
            {
                in.close( );
                out.close( );
                soc.close( );
                svrsoc.close( );
                System.exit(0);
            }
        }
    }
```

TCP 客户程序：

```
import java.net.*;
import java.io.*;
public class tcpclient
{
  static public void main(String args[ ])
            throws IOException
  {
            Socket soc=null;
            DataInputStream in=null;
            PrintStream out=null;
            DataInputStream sysin=null;
            String strin=null;
            String strout=null;

            try
            {
                soc=new Socket(args[0],8000);
                System.out.println("Connecting to the Server...");
                in=new DataInputStream(soc.getInputStream( ));
                out=new PrintStream(soc.getOutputStream( ));
                strin=in.readLine( );
                System.out.println("Server said:"+strin);
```

```
            sysin=new DataInputStream(System.in);
            strout=sysin.readLine( );
            while(!strout.equals("quit"))
            {
                out.println(strout);
                strout=sysin.readLine( );
            }
            out.println(strout);
        }
        catch(Exception e)
        {
            System.out.println("Error:"+e);
        }
        finally
        {
            in.close( );
            out.close( );
            sysin.close( );
            soc.close( );
            System.exit(0);
        }
    }
}
```

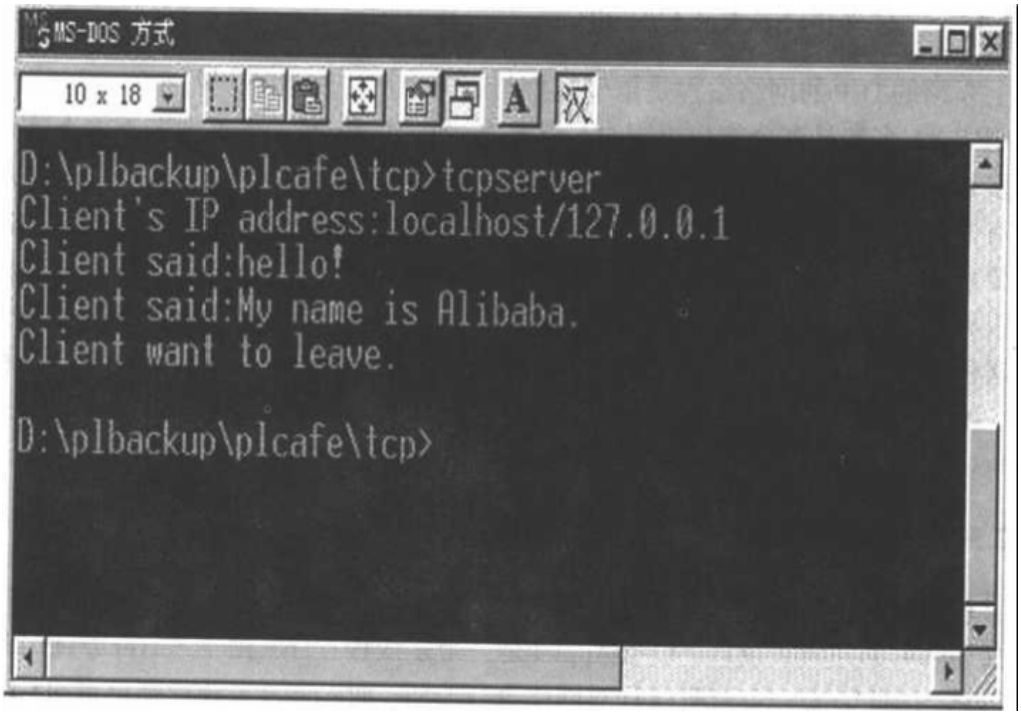

图 2.7　TCP 服务程序运行结果

客户程序在这里作了一些改动，将服务程序所在主机的IP地址作为args[0]参数传递，这样就可以实现两机通信。在运行了服务程序后，如果服务程序在本机上，用dos命令运行：

tcpclient 127.0.0.1

即在客户程序与服务程序间建立了连接。图2.7和图2.8为运行结果：其中图2.7dos窗口中运行的是服务程序，图2.8dos窗口中运行的是客户程序。

图2.8 TCP客户程序运行结果

2.2.4 一对多的Socket C/S通信

以上介绍的TCP的网络编程是单个客户与服务者进行联络的。然而在Internet的许多实际应用中，大多数是多个客户机同时与服务者进行联系，即一个服务者为多个客户服务的一对多C/S通信。在这种情况下，服务者将因为要同时处理几个客户的请求而变得复杂些。当有多个客户同时连接到服务者上时，服务者要建立与每一个客户相对应的Socket，并且还要建立多个线程，每个线程用于处理每一个客户的请求。为了便于对多个客户进行管理，服务者采用向量类（Vector）用于存放客户的编号信息，根据向量下标访问相应的客户。这种一对多的C/S通信在通信机制上与上述一对一的C/S通信是相同的，采用TCP/IP协议、数据流的形式进行传输。只是在服务者方要采用多线程同步机制。下面用一个模拟Internet上的聊天程序（Chat）来具体讲述客户方和服务方的实现。

2.2.5 一对多通信的客户方实现

Chat Client是一个Applet，它在浏览器中运行，Applet一旦被浏览器下载，就与Server建立连接，然后创建聊天窗口。Chat Applet是一个多线程程序。由于Server方随时都有可能传来信息，因此需要一个监听线程对网络进行监听。而Applet主线程则负责对GUI事件进行处理。Chat Client的流程图如图2.9所示。

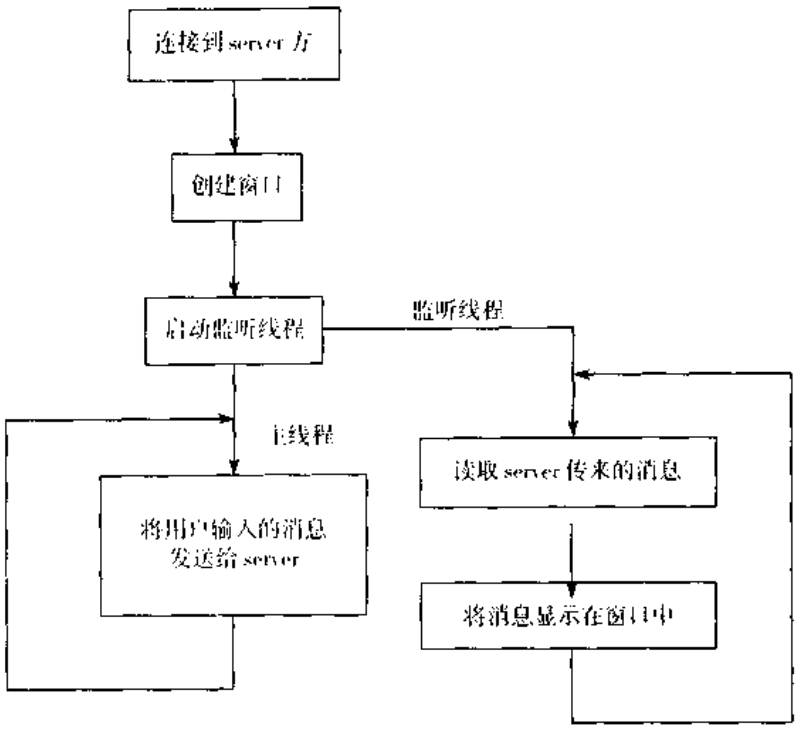

图 2.9　Chat Client 的流程图

2.2.6　一对多通信的服务方实现

Chat Server 也是一个多线程程序，如果聊天室有 n 个网友在聊天，Chat Server 就应该为每一个人连入的Chat Applet创建一个对应的线程，该线程监听对应的Applet是否有信息传来。如果有则向所有的 Chat Applet 广播该消息，也就是说，这时 Chat Server 应该创建有 n 个线程。Chat Server 的流程图如图 2.10 所示。

顾客 Client 程序如下：

```
/*
      A basic extension of the java.applet.Applet class
 */

import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;

public class chatApplet1 extends Applet implements Runnable{
        TextArea  m_textarea;// 信息显示窗口
        TextField  m_textfield;// 信息输入窗口
        DataInputStream  m_in;
        DataOutputStream  m_out;
```

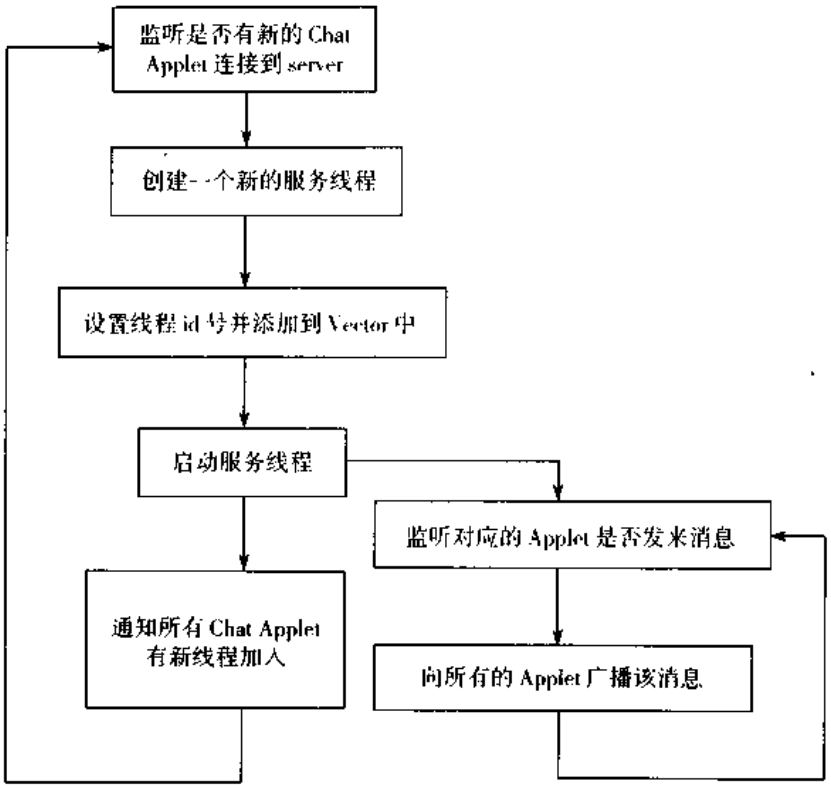

图 2.10　Chat Server 的流程图

```
    /*
    *Applet 的初始化方法
    */
public void init( )
{
        //{{创建窗口
        setLayout(null);
        setSize(426,266);
        //}}
         m_textarea=new TextArea(10,10);
         m_textfield=new TextField( );
         m_in=null;
         m_out=null;
         //初始化网络，并连接到 Server 方
        try{
          URL url=getCodeBase( ); //获取 applet 的 URL 值。
          InetAddress inetaddr=InetAddress.getByName(url.getHost( ));
          Socket m_socket;
```

```
            System.out.println("Server:"+inetaddr+" "+url.getHost( )+" "+url.getProtocol( ));
            m_socket=new Socket(inetaddr,5555);
            m_in=new DataInputStream(m_socket.getInputStream( ));
            m_out=new DataOutputStream(m_socket.getOutputStream( ));
        }
        catch (Exception e)
        {
             System.out.println("Error:"+e);
        }
        setLayout(new BorderLayout( ));
        add("Center",m_textarea);
        add("South",m_textfield);
        m_textarea.setEditable(false);
          // 启动监听线程
        new Thread(this).start( );
    }

    /* 当用户在信息输入域输入回车后,
    * 读取字符串,发送给服务者方。
    */

    public boolean handleEvent(Event event)
    {
         String b=m_textfield.getText( );

         if((event.target==m_textfield)&&(event.id==Event.ACTION_EVENT))
        { m_textfield.setText("");
         // 将用户输入的消息发送给 Chat Server
         try{
              m_out.writeUTF(b); // 写一 UTF 格式字符串。
         }
         catch(IOException e){}

         return true;
         }
         else
         return super.handleEvent(event);
    }
      /* 监听线程在这里读取 Chat Server 传来的消息,
```

```
    * 并显示在通信显示窗口中。
    */

    public void run( )
    {
        try
        {
            while(true)
            {
                // 监听服务者发来的消息，线程将阻塞在该语句中，
                // 直到消息到来。
                String s=m_in.readUTF( ); // 读一个 UTF 格式字符串。
                if(s!=null)
                // 将消息显示在信息显示窗口中。
                m_textarea.append(s+"\n");
            }
      }
            catch(Exception e)
            {
                m_textarea.append("Network problem or Sever down.\n");
                m_textfield.setVisible(false);
            }
      }
      public void stop( )
      {
            try
            {
                m_out.writeUTF("leave");
            }
            catch(IOException e){}
      }

    }
```

服务者源程序 Chat Server 如下：

```
/*
        A basic Java class stub for a Win32 Console Application.
```

```
 */
import java.net.*;
import java.io.*;
import java.util.*;


public class chatserver {
/*
* m_threads 是一个 Vector 静态变量，它维护所有 Server 方的
* ServerThread 实例，通过该变量能向所有连入 Internet
* 的 Applet 广播信息
*/
    //Chat Server 的主方法入口。
    // 该方法监听 Chat Applet 的请求，并为新连接的
    //Applet 创建一个服务线程
    public static void main(String args[ ])
    {
        ServerSocket socket=null;
        Vector m_threads=new Vector( );
        System.out.println("listen...");

        try
        {
            // 设置 Server 监听端口号为 5555，这个数字必须
            // 和程序 Chat Client 中的 port 参数一致。
            socket=new ServerSocket(5555);
        }
        catch(Exception e)
        {
            System.out.println("new ServerSocket( )failed!");
            return;
        }
        try
        {
            int nid=0;
            while(true)
            {
                // 监听是否有新 Chat Applet 连接到 Server，
                // 程序会陷入到该语句，直到有新的连接产生。
```

```
                Socket s=socket.accept( );
                System.out.println("accepted");
                // 创建一个新的 Server Thread.
                ServerThread  st=new ServerThread(s,m_threads);
                // 为该线程设置一个 ID 号。
                st.setID(nid++);
                // 将该线程加入到 m_threads Vector 中。
                m_threads.addElement(st);
                // 启动服务线程。
                new Thread(st). start( );
                // 通知所有 Chat Applet 有一个新的网友加入。
                for(int i=0;i<m_threads.size( );i++)
                {
                    ServerThread st1=(ServerThread)m_threads.elementAt(i);
                    st1.write("<#>welcome "+st.getID( )+" to enter chatroom!");
                }
                System.out.println("Listen again...");
            }
        }
        catch(Exception e)
        {
            System.out.println("Server is down...");
        }

    }
}

/*
* 监听线程，监听对应的 Chat Applet 是否有信息传来。
*/
    class ServerThread implements Runnable
    {
        Vector m_threads;
        Socket m_socket=null;
        DataInputStream m_in=null;
        DataOutputStream m_out=null;
        int m_nid;
        // 初始化该线程。
        public ServerThread(Socket s,Vector threads)
```

```
{
   m_socket=s;
   m_threads=threads;
   try
   {
        m_in=new DataInputStream(m_socket.getInputStream( ));
        m_out=new DataOutputStream(m_socket.getOutputStream( ));
   }
   catch(Exception e)
   {
   }
}

//线程的执行体。
public void run( )
{
    System.out.println("thread is running");
    try
    {
        while(true)
        {
            // 监听对应的 Applet 是否传来消息
            // 程序陷入到 m_in.readUTP( )中，直到有信息传来才返回。
            String s=m_in.readUTF( );
            If (s==null)  break;
            // 如果 Chat Applet 传来的信息为 "leave",
            // 则通知所有其他的的 Chat Applet 自己退出了。
            If (s.trim( ).equals ("leave"))
              For (int i=0;i<m_threads.size( );i++)
                 {
                    ServerThread st=(ServerThread)m_threads.elementAt(i);
                    st.write("*** "+getID( )+" leave..."+"***");
                 }
            else

            // 向所有 Chat Applet 广播该信息。
            for(int i=0;i<m_threads.size( );i++)
            {
                ServerThread st=(ServerThread)m_threads.elementAt(i);
```

```
                        st.write("<"+getID( )+">"+s);
                    }
                }
            }
            catch(Exception e)
            {
                e.printStackTrace( );
            }
            // 从 m_threads Vector 中删除该线程，表示该线程
            // 已经离开 chat room.
            m_threads.removeElement(this);
            try
            {
                m_socket.close( );
            }
            catch (Exception e){}
        }
        // 将 msg 送回对应的 Applet
        public void write (String msg)
        {
            synchronized(m_out)
            {
                try
                {
                    m_out.writeUTF(msg);
                }
                catch(IOException e){}
            }
        }

        // 获得该线程的 ID.
        public int getID( )
        {
            return m_nid;
        }

        // 设置线程的 ID.
        public void setID(int nid)
```

```
    {
        m_nid=nid;
    }
}
```

2.3 使用UDP协议的Socket C/S通信

2.3.1 UDP与TCP的对比

UDP是一种不可靠的，无连接的网络协议。在Java中由DatagramSocket类和DatagramPacket类来实现UDP通信。与可靠连接协议TCP相比，它不能保证发送的数据能被对方收到，如果数据在传输过程中丢失，它也不会重发，但是TCP协议发送数据需要建立一个特定的传输通道来对数据的传输进行控制，UDP协议的数据本身就自带了传输控制信息，因此，用UDP传输数据能节省系统开销。而且，在数据的传输效率上，UDP比TCP要高。所以，在网络上传输某些重要性较低的数据，或相隔一定时间要重发的数据（比如天气预报），经常使用到UDP，还有，测试网络是否通的ping程序，也是使用UDP。

2.3.1.1 通信协议端口

多数计算机的操作系统支持多程序并发功能，即允许多个应用程序同时执行。使用操作系统的术语，我们把每个正在运行的程序称为进程或任务；这种系统称为多任务系统。很自然地，进程是一条报文发送者或接受者。但是，把一台机器上运行的特定的进程当作某个数据报的最终目的地有点让人误解。首先，进程的生成和消失都是动态的，发送者难以了解其他机器上的进程的具体情况；其次，我们可以在不通知所有的发送者前提下改换接改数据报的进程（例如重新启动计算机之后，所有的进程都改变了，而发送方对新的进程并不知晓）；还有，我们需要从接收方所实现的功能来识别目的地，而不需要知道实现这个功能的进程（例如，允许发送者与一个文件服务者通信而不必知道这个目的机上到底由哪个进程来实现文件服务功能）。最重要的是，在那些允许由一个进程完成多个功能的系统中，我们一定要让进程能够知道发送方到底要求何种功能服务。

正因如此，我们不应该把进程看作通信的最终的目的地，而是应该把每台机器看作是一些抽象的访问目的对象即协议端口（protocol port）的集合。每个协议端口由一个正整数标识。本地的操作系统提供了一个接口机制，进程通过它来指定和接入到协议端口。

多数操作系统提供对端口的同步接入能力。从特定的进程的角度来看，同步接入意味着在接入端口时阻塞计算的运行。例如，一个进程在数据到达某端口之前需要从该端口提取数据时，操作系统就在数据到达前将该进程暂时挂起。一旦数据到来，操作系统把数据送给这个进程并激活它。通常各端口都有缓冲区，数据首先进入缓冲区。操作系统中的协议软件模块将到达某个端口的报文分组在缓冲区排成队列，进程需要数据时就到这个队列来取。

为了能够与外部端口通信，发送者不仅要知道目的机的IP地址，还要知道对应协议端口的标识号。每个报文必须带有目的机上的目的端口号，同时还带有发送者自身的端口号，这样就使得接收进程可以作出回应。

2.3.1.2 用户数据报协议

在TCP/IP协议族中，用户数据报协议UDP提供应用程序之间传送数据报的基本机制。UDP提供的协议端口能够区分在一台机器上运行的多个程序。也就是说，每个UDP报文不仅传送用户数据，还包括发送方和接收方的协议端口号，这使得接收方的UDP软件能够把报文送到正确的接收进程，而接收进程也能回送应答报文。

UDP使用底层的互连网络协议来送报文，提供和IP一样的不可靠的无连接数据报传输服务。它不使用确认信息对报文的到达进行确认，不对收到的报文进行排序，也不提供反馈信息来控制机器之间传输的信息流量。这就是说，UDP报文可能会出现丢失、重复、乱序的现象。而且分组到达的速率可能大于接收处理的速率。归纳一下：

通过使用IP在机器之间传送报文，用户数据报协议UDP提供了不可靠的无连接传输服务。它使用IP来携带报文，该协议要求相关主机增加区别多个通信目标的能力。

一个使用UDP的应用程序要承担可靠性方面的全部工作，包括报文的丢失、重复、时延、乱序以及连接失效等问题。不幸的是，应用程序员在编制程序时常常忽略了这些问题。此外，由于程序员常常在可靠性好、传输时延小的局域网上进行网络软件的测试，所以潜在的失效问题难以暴露出来。因而许多基于UDP的应用程序在局域网上工作得很好，而在大型的TCP/IP互连网络上运行却会出现各种戏剧性的错误。

2.3.1.3 UDP的报文格式

每个UDP报文称为一个用户数据报。用户数据报在概念上分为两个部分：UDP首部和UDP数据区。如图2.11所示，首部被分为四个长为16bit的字段，分别说明了报文是从哪个端口来、到哪个端口去、报文的长度以及UDP的校验和。

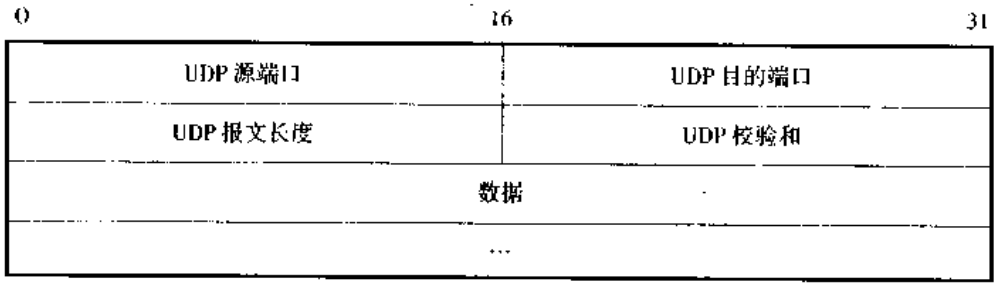

图2.11 UDP数据报的字段格式

源端口（SOURCE PORT）字段和目的端口（DESTINATION PORT）字段包含了16bit的UDP协议端口号，以便在各个等待接收报文的进程之间对数据报进行区分。其中源端口（SOURCE PORT）字段是可选的。若选用，则字段值是应答报文应该回送的目的端口值，若不选用时其值为零。

长度（LENGTH）字段记录了该UDP数所报的八位组数，这个长度包括了首部和用户数据区。因此长度（LENGTH）字段的最小值是8，即首部的长度。

UDP的校验和（CHECKSUM）字段是可选的，如果该字段值为零就说明不进行校验。

设计者把这个字段作为可选项的目的，是为了那些在可靠性很好的局域网上使用UDP的实现者能够尽量减少开销。IP协议对IP报文中的数据部分并不计算校验和，所以UDP的校验和字段提供了唯一的、对数据是否原封不动到达的监督手段，因此使用这个字段是很有必要的。

初学者往往会担心UDP报文的校验和计算出来确实是零的时候该怎么办。UDP使用和IP协议同样的校验和计算方法，即将数据分为16bit的单元，对每个单元计算其二进制反码，再对这些值进行异或得到校验和。幸运的是，零的二进制反码可以有两种表示方法，全0或全1，所以校验和为零并不会引起问题。当计算出的校验和为零时，UDP就使用全1来表示这种情况。

2.3.1.4 UDP的伪首部

UDP校验和覆盖的内容超出了UDP数据报本身的范围。为了计算校验和，UDP把伪首部（pesudo—header）引入到数据之中，在伪首部中有一个值为零的填充八位组用于保证整个数据报的长度为16bit的整数倍，这样才好计算校验和。填充八位组和伪首部并不随着UDP数据报一起传输，也不计算在数据报长度之内。为了计算校验和，我们先把零值赋予校验和（CHECKSUM）字段，然后对整个对象，包括伪首部、UDP的首部和用户数据，算出一个16bit的二进制反码和。

使用伪首部的目的是检验UDP数据报已到达正确的目的地。理解伪首部的关键在于认识到：正确的目的地包括了特定的主机和机器上特定的协议端口。UDP报文的首部仅仅指定了使用的协议端口号。因此为了确保报文能够正确到达目的地，发送UDP数据的机器在计算校验和的时候把目的机的IP地址和应有的数据都包括在内。在最终的接收端，UDP协议软件对校验和进行检验的时候要用到携带UDP报文的IP数据报的首部中的IP地址。如果校验和正确，那么说明UDP数据报达到了正确主机上的正确端口。

在UDP校验和的计算过程中用到的伪首部长度为12八位组，其结构如图2.12所示。伪首部的源IP地址（SOURCE IP ADDRESS）字段和目的IP地址（DESTINATION IP ADDRESS）字段记录了发送UDP报文时所使用的源IP地址和目的IP地址。协议（PROTO）字段指明了所使用的协议类型代码（UDP是17），而UDP长度（UDP LENGTH）字段是UDP数据报的长度（伪首部的长度不计算在内）。接收方进行正确性验证的时候，必须要把这些字段的信息从IP报文的伪首部中抽取出来，以伪首部的格式进行装配，然后再重新计算校验和。

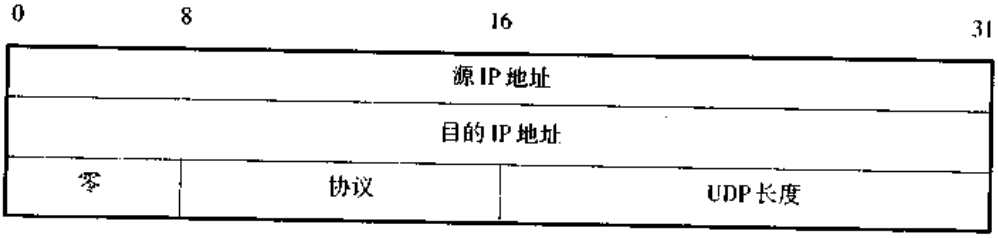

图2.12 校验和使用的12个八位组长的UDP伪首部格式

2.3.1.5 UDP的封装与协议的分层

UDP提供了本书讨论的第一种运输协议。在层次结构模型中，UDP位于Internet层之

上。从概念上说，应用程序访问UDP层，然后使用IP层来传送数据报。在图2.13中表示了这种结构。

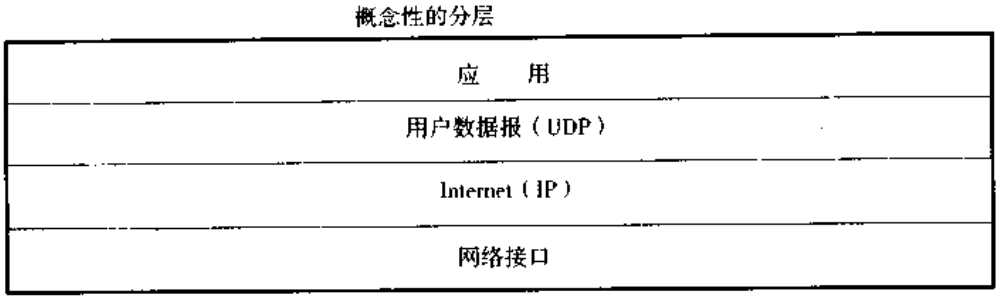

图2.13　UDP在TCP/IP协议层中的位置

将UDP层放到IP层之上意味着一个包括UDP首部和数据的完整UDP报文，在互连网络中传输时要封装到IP数据报中，图2.14给出了示意图，图中IP数据报在具体的网络上传输时又被进一步封装成为帧。

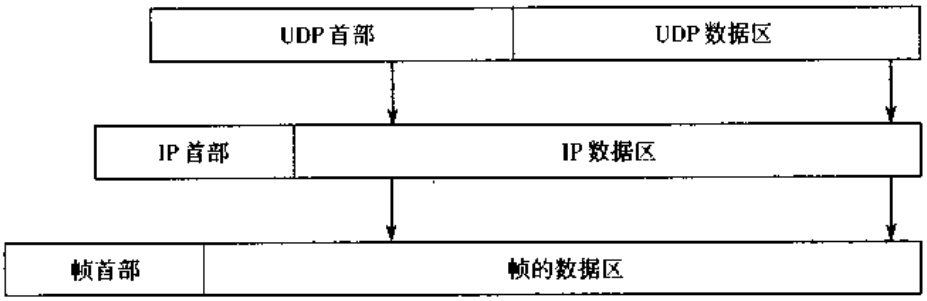

图2.14　UDP数据报协议

对于所讨论的协议，封装意味着UDP给想要传输的（从应用层接收到的）数据加上一个首部后再交给IP层。IP层又给从UDP层接收到的数据加上一个首部（继续交给下一层）。最后，网络接口层把数据报封装到一个帧里，再进行机器之间的传输。帧的结构根据下层的网络技术来确定。通常网络的帧结构包括另外的首部。

在接收端，最低层的网络软件接收到一个分组后把它提交给上一层模块。每一层都在向上送交数据之前剥去本层的首部，因此当最高层的协议软件把数据送到相应的接收进程的时候，所有附加的首部都被剥去了。也就是说，最外层首部对应的是最低层的协议，而最内层的首部对应的是最高层的协议。研究首部的生成与剥除时，可从协议的分层原则得到启发。当把分层原则具体地应用于UDP协议时，可清楚地知道在目的机上的由IP层送交给UDP层的数据报就等同于发送机上的UDP层交给IP层的数据报。同样的，接收方的UDP层上交给用户进程的数据也就是发送方的用户进程送到UDP层的数据。

在多层协议之间，职责的划分是清楚而明确的：IP层只负责在互连网络上的一对主机之间进行数据传输，而UDP层只负责对一台主机上的复用的多个源端口或目的端口进行区分。

因此。只有IP层的首部指明了源主机和目的主机地址；只有UDP层指明了主机上的

源端口和目的端口。

2.3.1.6 层次的划分及UDP校验和的计算

细心的读者可能注意到，分层原则与UDP校验和的计算过程看来存在着冲突。回忆一下，UDP的校验和覆盖了一个伪首部，而这个伪首部中包括了源IP地址和目的IP地址。我们可以认为进行发送的用户必须知道目的IP地址，在发送数据时把目的IP地址告诉了UDP层，这样UDP层就可直接得知目的IP地址而不必从IP层去寻找。但是，源IP地址要根据IP层对路由的选择而定，因为源IP地址会表示出数据报传输时所经过的网络接口。因此，如果不与IP进行交互，UDP层是无法得知源IP地址的。

假定UDP软件要求IP层提供源IP地址和（可能还有）目的IP地址，再用这些信息去生成伪首部、计算校验和、抛弃伪首部以及将UDP数据报交给IP层传送。一种可选的提高软件效率的方法是让UDP层把UDP数据报封装到一个IP数据报中，从IP层获取源地址，把源地址和目的地址填到数据报首部中的相应字段中，计算UDP的校验和，最后把这个IP数据报交给IP层，而IP层只需将IP报文首部中剩下的字段填充好即可。

UDP与IP这样密切的交互操作会不会破坏我们的基本前提，即划分层次要对应于独立的功能呢？回答是肯定的。这种方案明显是一种对严格的层次结构的折衷，完全是出于实践的需要。之所以忽略违反分层结构的情况，是因为几乎不可能在不指定目的机的前提下识别目的应用程序，而是我们希望对UDP所使用的地址和IP所使用的地址进行有效的映射。

2.3.1.7 UDP的复用、去复用和端口

我们看到协议体系结构的各层中的软件都要对临近层的多个对象进行复用和去复用操作。UDP软件提供了复用和去复用的又一个例子。它接收多个应用程序送来的数据报，把它们送给IP层去传输，同时它接收IP层送来的UDP数据报，把它们送给对应的应用程序。

从概念上来说，所有的UDP软件与应用程序之间的复用和去复用都要通过端口机制来实现。实际上每个应用程序在发送数据之前必须与操作系统进行协商以获得协议端口和相应的端口号。当指定了端口之后，凡是利用这个端口发送数据报的应用程序都要把端口号放入UDP报文中的源端口（SOURCE PORT）字段中。

当UDP从IP层软件接收了数据报之后，根据UDP的目的端口号进行去复用，图2.15对此进行了描述。

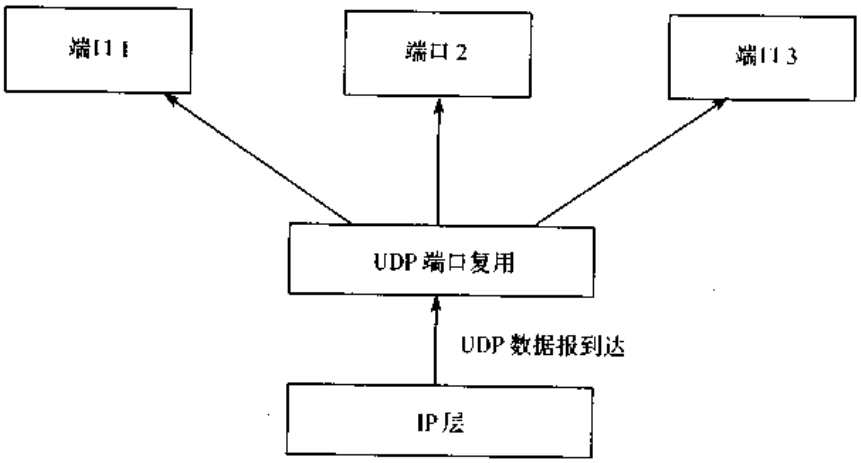

图2.15 IP层之上的端口复用举例

UDP端口的最简单的实现方法是一个队列。在大多数的实现中，当应用程序与操作系统协商，企图发送数据报到某个端口的时候，操作系统就创建一个内部队列来接收收到的数据报。通常应用程序可以指定和修改这个队列的长度。当UDP收到数据报时，先去检查当前使用的端口是否就是该数据报的目的端口。如果不能匹配，它就送一个ICMP的端口不可达的错误信息报文并抛弃这个数据报。如果匹配，它就把这处数据报送到相应的队列之中，等待应用程序的访问。当然在端口已满的情况下也会出错，UDP也要抛弃这个数据报。

2.3.1.8 保留的和可用的UDP端口编号

如何指定协议端口号呢？这个问题很重要，因为两台计算机之间在交互操作之前必须确认一个端口号。例如A机希望从B机那里获得一个文件，它就需要知道B机上的文件传输程序所使用的端口号。端口号的指派有两种基本方式。一种是使用集中式管理机构。大家服从一个管理机构对端口的指派，这个机构负责发布这些指派。所有的软件在设计时都要遵从这些指派的规定。这种方式又称为统一指派（universal assignment），这些被管理机构指定的端口指派也叫知名端口（well—known port）的指派。

由当前指定的UDP端口的指派以及相应的Internet标准关键字和UNIX关键字组成的表格如下。本表并不全面。出于扩展的方便，其他的传输层协议所提供的同等功能也使用与UDP规定相同的端口号。

十进制数字	关键字	UNIX关键字	描述
0	—	—	保留
7	ECHO	echo	回送
9	DISCARO	discard	丢弃
11	USERS	systat	用户
13	DAYTIME	daytime	日期时间
15	—	netstat	网络状态 NETSTAT
17	QUOTE	qotd	日期的引用
19	CHARGEN	chargen	字符发生器
37	TIME	time	时间
42	NAMESERVER	name	主机名字服务器
43	NICNAME	whois	是谁
53	DOMAIN	nameeserver	域名服务器
67	BOOTPS	bootps	引导协议服务器
68	BOOTPC	bootpc	引导协议客户机
69	TFTP	tftp	简单文件传送
111	SUNRPC	sunrpc	Sun Microsystems RPC
123	NTP	ntp	网络时间协议
161	—	snmp	简单网络管理协议
162	—	snmp—trap	SNMP陷阱
512	—	biff	UNIX comsat
513	—	who	UNIX rwho daemon
514	—	syslog	系统记录
525	—	timed	时间 daemon

第二种指派方式是动态连接（dynamic binding）。在使用动态连接时，端口并非为所有机器知晓。当一个应用程序需要使用端口时，网络软件就指定一个端口。为了知道另一台机器上的当前端口号，必须送出一个请求报文，提出类似于“当前的文件传输服务所使用的端口号是什么？”的问题，然后目的主机进行回答，把正确的端口号送回来。

TCP/IP的设计者们采纳了一种混合方式以端口地址进行管理，他们对某些端口进行指派，但为本地站点和应用程序留下了很大的端口取值范围。对一些固定端口分配了由小到大的值，而把较高的值留待进行动态指派。上表中给出了一些指定的UDP端口值，其中第二列是Internet的标准关键字，而第三列是多数UNIX系统中用来表示这些端口的关键字。

2.3.2 UDP协议通信的服务方实现

服务程序是一直在服务器上运行的，它在一定的端口上侦听来自网络上的连接请求。

DatagramPacket类用于存放数据，DatagramSocket类用于实现数据的发送和接受，它相当于服务器程序中的ServerSocket。

下面介绍使用这两个类的一个UDP服务程序。同TCP服务程序一样，它只是简单地将客户输入的字符显示在服务器窗口中，当客户输入"BYE"字符串后，该应用程序停止并退出。

在程序中，先在端口9000上建立一个DatagramSocket：

```
svrsoc=new DatagramSocket(9000);
```

然后，创建一个DatagramPacket，用于接收数据并从DatagramSocket中读取客户传过来的数据，将其存放在DatagramPacket中：

```
svrpac=new DatagramPacket(buf,buf.length);
svrsoc.receive(svrpac);
```

将DatagramPacket的数据转换成字符串：

```
data=new String(buf,0,0.svrpac.getLength( ));
```

获取客户机的IP地址和DatagramSocket端口号：

```
clientIP=svrpac.getAddress( );
clientPort=svrpac.getPort( );
```

在任意端口上创建一个DatagramSocket，用于向客户发送数据：

```
soc=new DatagramSocket( );
```

判断客户传送过来的数据，如果是"quit"，则跳出循环，否则，将用户传送的数据显示在服务器的窗口中，并将字符串"I have received:"及用户传送的信息返回给用户：

```
System.out.println("Client said:"+data);
strToSend="I have received:"+data;
msg=new byte[strToSend.length( )];
strToSend.getByte(0,msg.length,msg,0);
pac=new DatagramPacket(msg,msg.length,clientIP,clientPort);
soc.send(pac);
```

在程序中，我们用到了DatagramPacket的两个重要方法：getAddress( )和getPort( )，利用它们可获得远端客户机的IP地址和端口号，并利用它们作为参数向客户发送数据。

以上是服务者的实现，下面介绍客户方的实现。

2.3.3 UDP协议通信的客户方实现

在客户程序中，用户将在窗口中输入的一行字符传送给服务器，再接收服务器发送的信息，并将其显示在窗口中。

首先，客户程序在任意空闲的端口上创建DatagramSocket对象：

```
soc=new DatagramSocket( );
```

然后接收用户的键盘输入，并判断如果为"quit"将退出循环：

```
sysin=new DatagramStream(Syetem.in);
        strToSend=sysin.readLine( );
        while(!strToSend.equals("quit"))
```

将键盘数据存放在DatagramPacket对象中，指定远端主机名和端口号，假如是在一台计算机中运行的服务者程序和客户程序之间通信，则主机名为"localhost"：

```
bufToSend=new byte[strToSend.length( )];
strToSend.getByte(0,bufToSend.length,bufToSend,0);
pacToSend=new DatagramPacket(bufToSend,bufToSend.length,
            InetAddress.getByName("localhost"),9000);
```

将数据发送给服务者：

```
  soc.send(pacToSend);
```

然后接收服务者返回的信息，并将信息显示在窗口中：

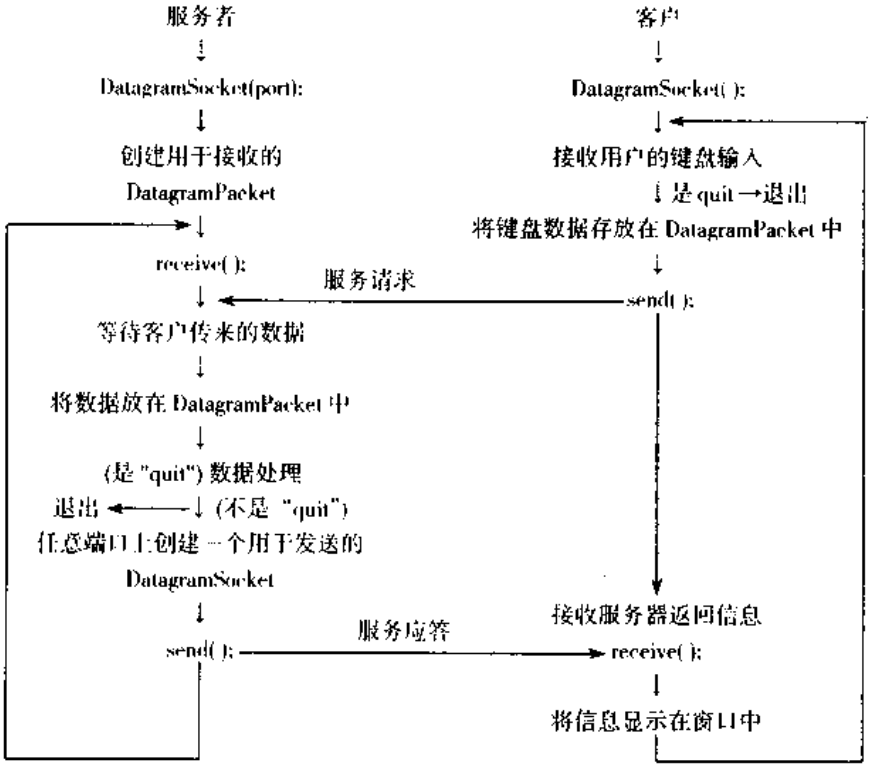

图2.16 基于无连接(UDP)的服务者客户流程图

```
pacToRec=new  DatagramPacket(bufToRec,bufToRec.length);
soc.receive(pacToRec);
data=new  String(bufToRec,0,0,pacToRec.getLength( ));
System.out.println("Server said:"+data);
```

当用户输入为"quit"时，先将其传送给服务者，然后退出：

```
bufToSend=new byte[strToSend.length( )];
strToSend.getByte(0,bufToSend.length,bufToSend,0);
pacToSend=new DatagramPacket(bufToSend,bufToSend.length,
            InetAddress.getByName("localhost"),9000);
soc.send(pacToSend);
```

当服务者和客户程序编译后，先运行服务程序，后运行客户程序，就可检验UDP的通信效果。也可以在两台计算机之间通信，只需将客户程序中指定的服务者主机名改为相应的主机名就可以了。

图2.16是基于无连接的服务者、客户程序流程图：

由本章介绍可以看出，Java有关Socket、ServerSocket的网络通信编程是网络应用的基础。

第3章　Java中URL类的应用

3.1　Java URL类简介

URL类封装了使用统一资源定位器(Uniform Resource Locator)访问一个WWW上的资源的方法。这个类可以生成一个寻址或指向某个资源的对象。URL类生成的对象指向WWW资源（Web页、文本文件、图形文件、声频片段等等）。

URL的基本表示方法是：

protocol://hostname:port/resourcename#anchor

其中：

- protocol制订使用的协议，它可以是http、ftp、gopher、news、telnet等
- 主机名hostname指定DNS服务器能访问到的WWW上的计算机名称，如www.sun.com。
- 端口号port是可选的，表示所联的端口，只在要覆盖协议的缺省端口时才有用，如果忽略端口号，将连接到协议缺省的端口，例如http协议的缺省端口为80。
- 资源名resourcename是主机上能访问到的目录或文件。
- 标记anchor也是可选的，它指定在资源文件中的有特定标记的位置

下面是几个合法的URL例子：

http://www.ncsa.uiuc.edu/demoweb/url-primer.htm

http://www.ncsa.uiuc.edu:8080/demoweb/url-primer.htm

http://local/demo/information#myinfo

ftp://local/demo/readme.txt

第二个URL把标准Web服务器端口80覆盖成不常用的8080端口，第三个URL加上符号"#"，用于指定在文件information中标记为myinfo的部分。

构造了的URL类实例，就可以用getPort()、getProtocol、getHost()、getFile()、和getRef()方法提取URL的任何构成字段。URL的核心方法是getContent()，不必显式指定寻找的资源类型，就可以取回资源并返回相应的形式(例如GIF或JPEG图形资源会返回一个Image对象)。URL类提供的一个重要方法是openStream()，顾名思义，这个方法的作用是打开一个输入流，返回类型是InputStream，而这个输入流的起点是URL实体对象的内容所代表的资源位置处，终点则是使用了该URL实体对象及方法openStream()的程序。在输入流建好了之后，我们就可以从输入流中读取数据了，而这些信息数据的实际来源，则是为输入流起点的网上资源文件。

3.2 用 URL 获取文本和图像

利用URL可以方便地获取文本和图像。文本数据源可以是网上或者本机上的任何文本文件，只要该文本文件的地址表示符合URL的标准位置表示法。如果要利用URL来获取图像数据的话，就不能使用openStream( )方法了，而是要用到方法getImage(URL)。这个方法会立即生成一个Image对象，并且返回程序对象的引用，但这并不意味着图像文件的数据已经读到了内存之中，而是系统在这同时，产生了另一个线程去读取图像文件的数据，因此就可能存在程序已经执行到了getImage( )后面的语句部分，而系统还正在读图像文件数据的情形。下面是一个利用URL来获取文本文件（.txt）和图像文件（.jpeg,.gif）的例子：

```
import java.net.*;
import java.awt.*;
import java.io.*;
//这个类继承了Frame类，生成一个带标题的窗口
public class getDataByURL extends Frame
{
    MenuBar menuBar;
    boolean drawImage=false;
    DataInputStream dataInputStream;
    int i=0;
    String line_str;
    boolean first=true;
    Font font;
    //类getDataByURL的构造函数
    public getDataByURL( )
    {
        //生成一个菜单条
        menuBar=new MenuBar( );
        setMenuBar(menuBar);
        //为菜单条第一个菜单取名 "display"
        Menu display=new Menu("display");
        menuBar.add(display);
        //生成 "display" 菜单下的两个菜单项
        MenuItem beauty_display=new MenuItem("display beauty");
        MenuItem text_display=new MenuItem("display text");
        display.add(beauty_display);
```

```
        display.add(text_display);
        //设置背景颜色和文本的字体
        setBackground(Color.white);
        font=new Font("System",Font.BOLD,20);
        //设置带有菜单的窗口的标题
        setTitle("sample:use URL get data");
        resize(400,300);
        //显示窗口
        show( );
    }
    //处理窗口中的菜单事件
    public boolean action(Event evt,Object what)
    {
        if(evt.target=instanceof MenuItem)
    {
        String message=(String)what;
        if(message=="display beauty")
        {
            drawImage=true;
            doDrawImage( );
        }
        else
        {
            drawImage=false;
            first=true;
            if(message=="display text")
            doWrite("file://d://plbackup/tt.txt");
        }
    }
    return true;
}
//处理窗体事件
public boolean handleEvent(Event evt)
{
    switch(evt.id)
    {
        case Event.WINDOW_DESTROY:
        dispose( );
```

```
            System.exit(0);
            default:
            return super.handleEvent(evt);
        }
}
static public void main(String args[])
{
    new getDataByURL( );
}
public void paint(Graphics g)
{
    if (drawImage)
    {
        try
        {
            //生成一个 URL 对象，它指向本机上
            //的一个类型为 .jpeg 的图形文件
            URL image_URL=newURL("file://d:/plbackup/zy4.jpeg");
            Toolkit object_Toolkit=Toolkit.getDefaultToolkit();
            Image object_Image=object_Toolkit.getImage(image_URL);
            g.setColor(Color.white);
            g.fillRect(0,0,300,400);
            g.drawImage(object_Image,40,80,160,200,this);
        }
        catch(MalformedURLException e){}
  }
  else
  {
        if(first)
        {
            first=false;
            g.setColor(Color.white);
            g.fillRect(0,0,400,300);
            g.setFont(font);
        }
   if(line_str!=null)
   g.drawString(line_str,10,i*20);
   i++;
```

```
        }
    }
    //画图像函数
    private void doDrawImage( )
    {
        drawImage=true;
        repaint( );
    }
//写文本函数，它的参数是一个指向绝对 URL 的字符串
    private void doWrite(String url_str)
    {
        try
        {
            //用参数 url_str 生成一个绝对的 URL,它指
            //向本机上的一个文本文件
            URL url=new URL(url_str);
            dataInputStream=new DataInputStream(url.openStream( ));
            try
            {
                i=1;
                line_str=dataInputStream.readLine( );
                while(line_str!=null)
                {
                    paint(getGraphics( ));
                    line_str=dataInputStream.readLine( );
                }
            }
            catch(IOException e){}
            dataInputStream.close( );
        }
        catch(MalformedURLException e1){}
        catch(IOException e2){}
    }
}
```

这个例子用到了菜单棒（MenuBar）、菜单（Menu）以及子菜单项（MenuItem）。有了菜单之后，便于用户通过菜单来决定想要获得的内容，其执行情况如图 3.1 所示。

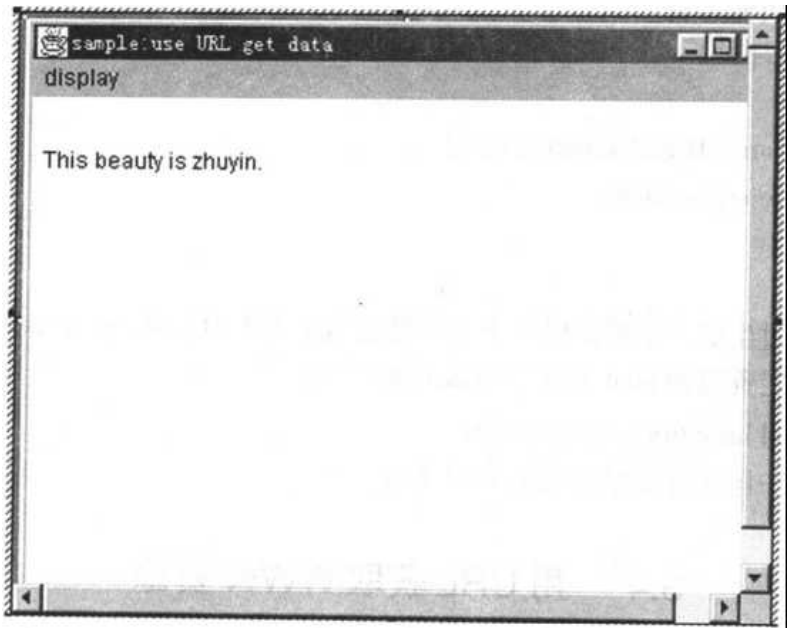

图 3.1　选中 display beauty 菜单项的状态显示

3.3　用 URL 获取网上 HTML 文件

这个例子是利用 URL 获取网络上资源的 html 文件：

```
import java.net.*;
import java.io.*;
public class urlhtml
{
    static public void main(String args[])
    {
        try
        {
            //根据参数 args[0]生成一个绝对的 URL 对象
            URL url=new URL(args[0]);
            //用 URL 对象打开一个输入流
            DataInputStream d=new DataInputStream(url.openStream( ));
            String inputLine;
            //从输入流读入数据
            while ((inputLine=d.readLine( ))!=null)
            {
                //将数据信息显示到系统标准输出上
                System.out.println(inputLine);
            }
```

```
            //关闭输入流
            d.close( );
        }
        catch(MalformedURLException me){}
        catch(IOException ioe){}
    }
}
```

这个例子通过生成一个指向网络上一个特定资源的URL，用openStream打开其输入流，从而读取指定资源的html文件。用dos命令运行

urlhtml http://www.cumt.edu.cn

就可以看到中国矿业大学主页的html文件。

3.4 用URL获取WWW资源

在上节的例子中，虽然可以非常方便地抓取网上资源的html文件，但是看到的仅仅是html文件本身的内容。实际上在许多应用中，还需要获得Web主页，或者FTP到网络上的一个FPT服务器上下载文件。下面这个例子就可以实现这些功能：

```
import java.io.*;
import java.net.*;
public class URL_FTP
{
    static public void main(String args[ ])
    {
        byte data=0;
        URL obj1;
        File obj2;
        DataInputStream inf=null;
        FileOutputStream outf=null;
        if(args.length!=2)
        {
            System.out.println("Download file!");
            System.out.println("Usage:java URL_FTP file1 file2");
            return;
        }
        try
        {
            //根据参数args[0]构造一个绝对的URL对象
             obj1=new URL(args[0]);
```

```
 }
 catch(MalformedURLException e)
 {
System.out.println("Open URL "+args[0]+" Error");
      return;
 }
//根据参数 args[1]构造一个 File 实体对象（文件）
 obj2=new File(args[1]);
 //系统输出输入文件的有关描述
 System.out.println("Input File Description:");
 System.out.println("\tProtocol:"+obj1.getProtocol( ));
 System.out.println("\tHost    :"+obj1.getHost( ));
 System.out.println("\tPort    :"+obj1.getPort( ));
 System.out.println("\tFile    :"+obj1.getFile( ));
 System.out.println("\tExternal:"+obj1.toExternalForm( ));
 System.out.println("\ttoString:"+obj1.toString( ));
 //得到输入文件的文件名称
 String s=obj2.getName( );
 System.out.println(s);
 try
 {
    //用一个 URL 的对象 obj1 打开一个输入流 inf
    inf=new DataInputStream(obj1.openStream( ));
 }
 catch(FileNotFoundException e)
 {
    System.out.println("file1 not found!");
 }
 catch(IOException e)
 {
    System.out.println("io error");
 }
 try
 {
 //用一个 File 的对象来构造一个文件输出流 outf
    outf=new FileOutputStream(obj2);
 }
 catch(FileNotFoundException e)
 {
```

```
            System.out.println("file2 not found!");
        }
        catch(IOException e)
        {
         System.out.println("Open Data Stream Error!");
            return;
        }
        try
        {
            do
            {
            //根据输入流 inf 生成一个用于输出的字节数组
            data=(byte)inf.readByte( );
            //输出字节数组
            outf.write(data);
            //循环直至将输入数据全部输完为止
            }while(true);
        }
        catch(EOFException e)
        {
        //一旦输出了所有的数据，提示文件已下载完毕
           System.out.println("File Download Complete!");
        }
        catch(IOException e)
        {
            System.out.println("File Download Error!");
            return;
        }
        try
        {
            //关闭输入流
            inf.close( );
            //关闭文件输出流
            outf.close( );
        }
        catch(IOException e){ }
    }
}
```

这个例子实际上实现了一个简单的拷贝功能，在这个拷贝过程中的源文件是网络上的某一资源，比如一台主机的WWW服务器或者FTP服务器的主页，目标文件是下载到的本机上的文件。当这个程序执行成功时，本机上就得到了一个下载文件。用浏览器打开这个文件，就得到了我们想要的WWW或FTP服务。

例如，在dos状态下输入下列命令行：

URL_FTP http://www.cumt.edu.cn d:/file1

就可以看到如图3.2所示的以下信息：

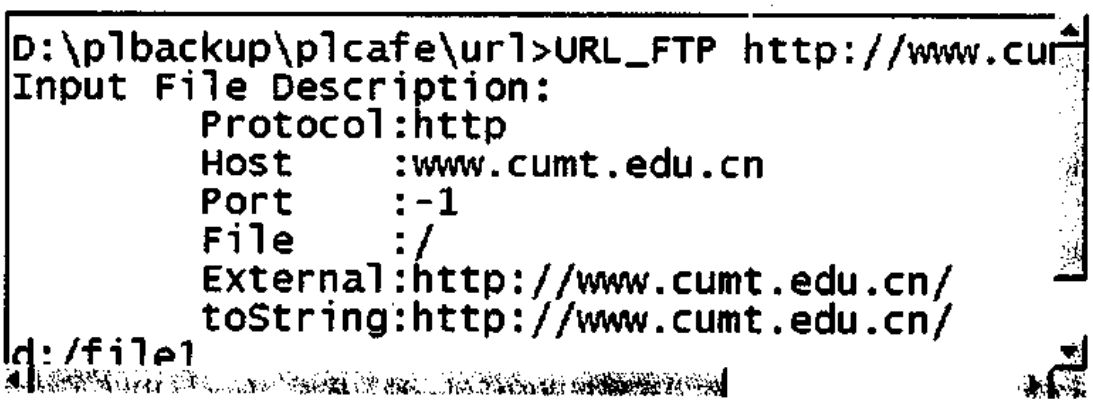

图3.2 WWW主页下载命令行的显示

这时打开d盘上的file1文件，选择IE浏览器为打开这个文件的程序，就可以在浏览器中看到中国矿业大学WWW的主页。

再例如，在dos状态下输入下列命令行：URL_FTP ftp://ftp.cumt.edu.cn d:/file2就可以看到如图3.3所示的信息。

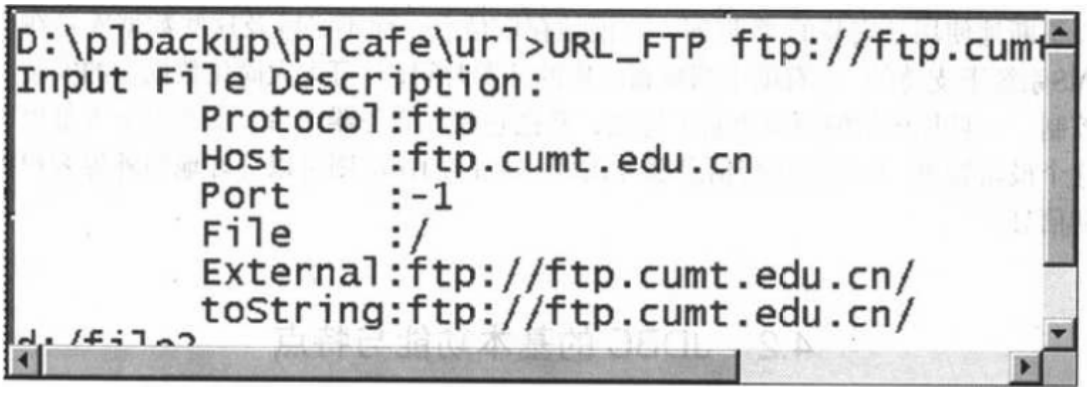

图3.3 ftp下载命令行的显示

第4章　Java与数据库的连结
——JDBC技术

4.1　概　　述

JDBC本身是一个产品的商标名，但它也可被看作为“Java Database Connectivity(Java数据库连接)”。它由一组用Java语言编写的类接口组成。JDBC已成为一种供数据库开发者使用的标准API，用户可以用纯Java API来编写数据库应用。

使用JDBC可以很容易地把SQL语句传送到任何关系型数据库中。用户可以用JDBC API写出一个程序，将SQL语句发送到相应的任何一种数据库。Java与JDBC的结合，使程序员可以只写一次数据库应用软件后，就可以在任何一种数据库系统上运行。由于Java语言具有健壮性、安全、易使用、易理解和自动下载到网络等优点，因此，它是数据库应用的一个极好的基础语言。JDBC又实现Java应用与各种不同数据库对话，扩充了Java的应用范围。例如，用Java与JDBC API可以发布一种包含远程数据库信息的Applet的Web页面。企业使用JDBC可以把它的所有雇员信息通过Intranet连接到一个或多个内部数据库里。毫无疑问，随着Java程序员的不断增加，对Java数据库的访问会越来越迫切。MIS管理员希望Java与JDBC结合，因为这有助于更容易、更经济地发布企业信息。商界可以不断地使用已安装的 数据库，方便地存取信息，而不必顾及这些数据库是在何种DBMS系统下支持的。它有助于缩短新应用的开发时间，并可大大简化数据库的安装与版本控制。一旦程序员编写或更新了应用，并把它放入服务器之后，那么每个人都可以获得这个最新版本。在商业销售信息服务领域，Java与JDBC则可以更好地向外界客户提供最新信息。

4.2　JDBC的基本功能与特点

4.2.1　JDBC的基本功能

简单的说，JDBC可以做三件事：

(1) 建立与数据库的连接。

(2) 发送SQL语句。

(3) 处理数据库操作结果。

为说明JDBC的基本功能，以下面的程序为例实现上述的三种基本功能。

```
Connection con=DriveManager.GetConnection("jdbc:odbc:wombat","Login", "Password" )
// 建立与数据库的连接
Statement stmt=con.createStatement( ); // 发送 SQL 语句
ResultSet rs=stmt.executeQuery("SELECT a,b,c FROM Table1"); // 数据库操作结果
While (rs.next( )){
int x=getInt("a");
string s=getString("b");
float f=getFloat("c");
}
```

4.2.2 JDBC API特点

1) 在SQL水平上的API

JDBC是为Java语言定义的一个SQL调用级界面（CLI），也就是说其关键在于执行基本的SQL说明和取回结果。在此基础上可以形成更高层次的API，其中的接口包括直接将基本表与Java中的类相对应，提供更多的通用查询的语义树表示。

2) 与SQL的一致性

一般数据库系统在很大范围内支持SQL的语法语义，但所支持的一般只能是SQL语法全集中的一个子集，并且在许多更强的功能，例如，它的外部连接及过程存储等方面彼此不能一致，而现在的标准SQL扩展了更多的功能。那么JDBC是怎么保证与SQL一致性的呢？

(1) JDBC允许使用从属于DBMS的系统的任何查询语句，因此一个应用程序，可以使用尽可能多的DBMS功能，尽管在一些DBMS中可能会出现错误。也就是说，不限制用户使用所有的SQL语法，而是通过驱动程序来加以限制。可以通过使用JDBC中提供的一些功能函数来获取有关数据库的基本信息，再对数据库进行操作避免错误。实际上，一个应用程序的查询甚至不必是SQL形式的，它可以是SQL特定演化形式，例如，为特定DBMS提供的文本或是图片查询等。

(2) 一般认为ANSI SQL 92 Entry Lever标准比较完备，并且被广泛支持，所以为了使JDBC与SQL一致，我们要求用户使用至少ANSI SQL 92 Entry Lever以上标准，这样就给那些要求广泛的可移植性的应用程序提供了至少共同命名的保证。

3) 可在现有数据库接口之上实现

JDBC SQL API保证能在普通的SQL API上实现，特别是ODBC。这种要求使JDBC的功能变得更加丰富，尤其是在处理SQL说明中的OUT参数及有关大的数据块上。

4) 提供与其他Java系统一致的Java界面

JDBC提供与Java系统其他部分一致的Java界面，这对Java语言来说有着非常重要的意义。在很大程度上，这意味着Java语言与标准运行系统被认为是一致的，简单化，并且是功能强大的。

5) JDBC 的基本 API 在最大可能上简单化

这也是体现在大多数情况下采用简单的结构来实现特定的任务，而不是提供复杂的结构。也就是说，对某个特定的任务，提供一种方案，而不是多种复杂的方案。JDBC 的 API 以后还将不断的扩展以实现更完善的功能。

6) 使用健壮的、静态的通用数据类型

JDBC API 使用健壮的数据类型，并且很多的类型信息采用静态表达，这就使很多的错误能在编译时捕获。但是，由于SQL本身是动态数据类型，所以在程序运行时，就可能会碰到类型不匹配的问题，例如，在对所操作的数据库基本信息未知的情况下，就可能发生程序员希望一个SELECT语句返回整数型结果，而数据库返回是字符串类型的情况。这时，程序员可以在操作前使用API中定义的一些基本方法，来对数据库的数据类型进行查询，这样程序员在编译时提出他们所希望的数据类型，就能尽可能地对数据类型进行静态检查。同时，在需要时，也能支持动态数据类型的界面。

7) 使一般任务简单化

所谓一般任务指的是程序员执行一个简单的没有参数的 SQL 说明（例如 SELECT、INSERT 、UPDATE 或 DELETE），然后获得简单的结果数据集。带有传入 IN 的参数的 SQL 说明，也属于一般任务。

8) 多种方法、多种功能

ODBC 中定义的界面过程少，利用过程的标志参数，使它们选择不同的操作。而 Java 核心的类定义的界面方法多，方法不带有标志项参数，使用基本接口时不必被与复杂功能相关的参数所困扰。

4.2.3 JDBC 是低级的 API 与高级 API 的基础

JDBC 是一种“低级”的接口，因为它直接调用 SQL 命令，但它又可作为构造高级接口与工具的基础。高级接口是“用户友善的”、更易理解和更为方便的 API，由后台将它翻译成如 JDBC 这样的低级接口。有两种基于 JDBC 的高级接口正处在开发之中：

一种是嵌入 SQL 的 Java。JDBC 要求把 SQL 语句作为字符串传递给 Java 方法。嵌入 SQL 预处理程序，可让程序员把 SQL 和 Java 直接混合使用，例如 SQL 语句中，可用 Java 变量接收或提供 SQL 值。嵌入 SQL 预处理程序，把这种混合的 Java/SQL 翻译成带 JDBC 调用的 Java。

另一种是把关系数据库表直接映射 Java 类。在“对象/关系”的映射中，表中的每一行变成类的一个实例，每一列的值对应于该实例的一个属性。然后程序员可直接操作 Java 对象，并自动生成存、取数据的 SQL 调用。另外它还提供了更高级的映射，例如 Java 类中把多个表的行结合起来。

4.2.4 JDBC 与 ODBC 和其他 API 的比较

目前，微软的 ODBC API 是访问关系型数据库中运用最广的编程接口，它几乎能将所有平台的所有数据库连接起来。可否通过 Java 来使用 ODBC 呢？回答是肯定的，但最好的做法是在 JDBC 的帮助下采用 JDBC-ODBC 桥接方式实现。需要 JDBC 理由如下：

1) Java 不能直接使用 ODBC。因为 ODBC 使用 C 语言接口，如果让 Java 调用本机 C 代码的话，那么会在安全、属性、健壮性、应用的可移植性方面带来困难。

2) 不希望把 ODBC CAPI 逐字翻译成 Java API。例如，ODBC 使用了大量的易于出错的指针，而 Java 取消了这种不安全的指针。现在通用的 JDBC，把 ODBC 翻译成具有 Java 风格的面向对象的接口。

3) ODBC 难于学习。ODBC 把简单功能与高级功能混杂在一起，即使是简单的查询也会带来复杂任选项。而 JDBC 的设计使得简单的事情用简单的做法，仅在必要时才让用户使用高级功能。

4) JDBC 的 JDBC API 提供“纯 Java”的解决办法。当使用 ODBC 时，ODBC 驱动器管理程序与驱动器必须手工地装入到每台客户机上。而 JDBC 驱动器全部是用 Java 编写的，JDBC 代码则在所有 Java 平台上都可自动安装，并且是可移植的和安全的。

总之，JDBC API 是一种基于 SQL 的抽象与 Java 接口，它是基于 ODBC 的，熟悉 ODBC 的程序员很容易学习 JDBC，JDBC 保留基本设计功能。而且两种接口都是基于 X/Open SQL CLI(Call Lever Interface)。最大的区别是 JDBC 保持了 Java 自身的风格与优点。最近，Microsoft 引入了 ODBC 以外的新的 API，包括 RDO、DAO 和 OLE DB。其设计策略在许多方面与 JDBC 类似，也是一种基于 ODBC 类的面向对象的数据库接口。

4.2.5 在数据库存取的二层与三层模型上的应用

两层模型中 Applet 或 Application 直接和数据库进行对话，这需要一个能够和特定数据库管理系统通信的 JDBC 驱动程序，数据库可以运行在本地或网络上另一台机器上，这属于两层的客户 / 服务方式（如图 4.1 所示）。

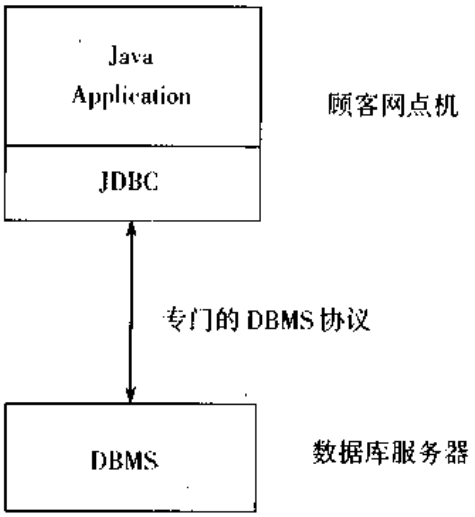

图 4.1 两层的客户 / 服务方式

在三层模型中，命令被发送到服务器端的一个“中间层”，然后发送到数据库，数据库处理的结果返回到中间层再到用户（如图 4.2 所示），三层结构的灵活性好、易于维护，而且可以提供一个更容易使用的高级用户接口，由中间层负责将其翻译成底层调用。

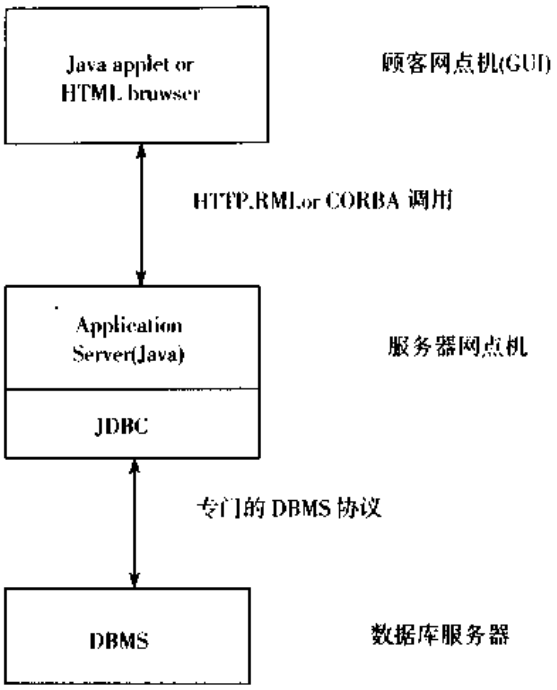

图 4.2　三层结构的客户 / 服务方式

4.3　JDBC 产 品

4.3.1　JavaSoft 框架

JavaSoft 提供了三种 JDBC 产品组件，并作为 JDK 的组成部分（见图 4.3）：

(1) JDBC 驱动器管理程序。

(2) JDBC 驱动器测试集。

(3) JDBC-ODBC 桥接。

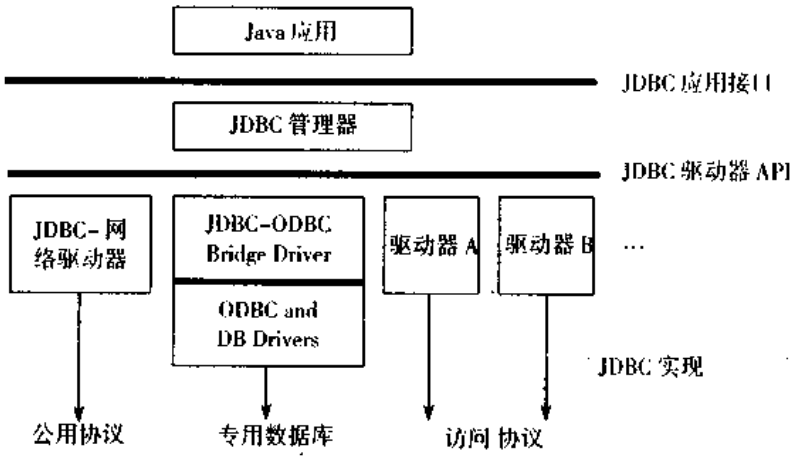

图 4.3　JDBC 总体结构

JDBC驱动器管理程序是JDBC组件的支柱，它的主要功能是把Java应用联结到JDBC驱动器。

JDBC驱动器测试集测试JDBC驱动器的可信度。只有通过JDBC测试集的驱动器才可以得到JDBC-COMPLIANT（JDBC兼容）标记。

JDBC-ODBC桥接允许ODBC驱动器作为JDBC驱动器使用。这是使JDBC迅速投入使用的一种现实方式。长远考虑，它用来存取专用的DBMS系统。

4.3.2 JDBC驱动器类型

SUN公司将JDBC驱动程序分为以下4种类型，分别适用于不同的场合，如图4.4所示。

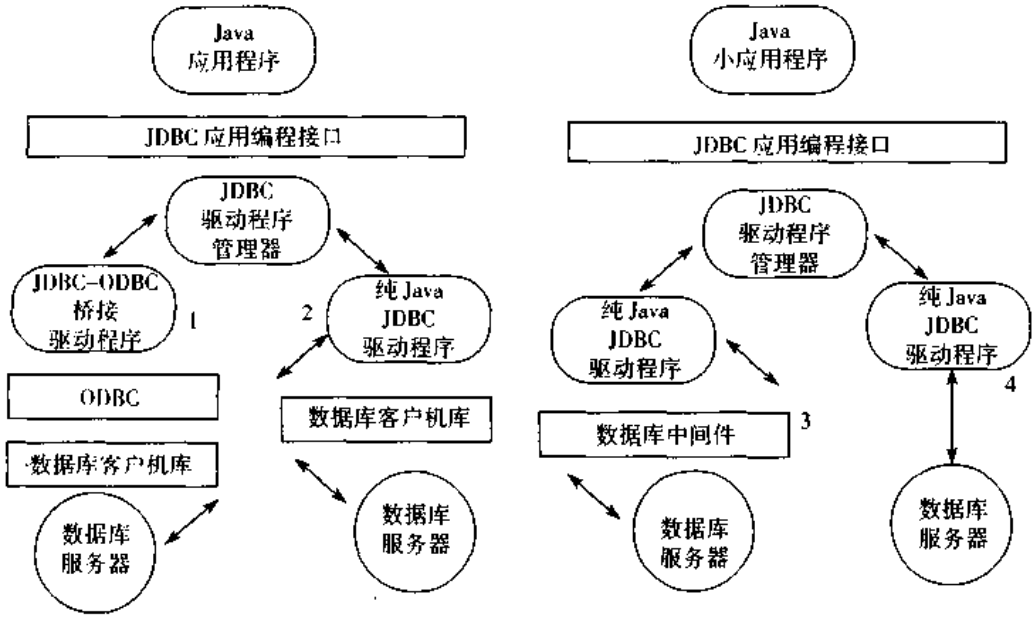

图4.4 驱动程序的4种类型

(1) JDBC-ODBC桥接驱动程序：Sun桥接产品通过ODBC驱动程序提供JDBC存取。经过ODBC驱动程序访问数据库，在大多数情况下，ODBC二进制代码必须在每个使用该数据库驱动程序的客户端安装，所以这种驱动程序主要适用于公司内部网络，或者在3层结构中用Java编写应用服务器代码。

(2) 本机应用编程接口部分Java驱动程序：此类驱动程序转换JDBC调用到客户端针对特定数据库系统的API，如Oracle, Sybase, Informix, DB2, 或其他的DBMS，像桥接驱动程序一样，这种类型的驱动程序要求一些二进制代码在每一个客户机上被安装。

(3) 数据库中间件的纯Java驱动程序：此类驱动程序将JDBC调用转换成为中间件供应商的协议，然后通过中间件服务器转换成为DBMS协议。中间件为许多不同数据库提供连接。这个网络服务器中间件连接所有Java客户端到各种不同的数据库，特定的协议取决于供应商。通常这种方式是JDBC最方便的选择，提供商将为Internet用户提供产品套件。为了使这些产品能够支持Internet访问，必须处理另外的安全、防火墙穿越等需求，各

个提供商正在增加JDBC驱动程序到它们现存的中间件产品中。

(4) 直接连接数据库的纯Java驱动程序：本地协议纯Java驱动程序，此类驱动程序转换JDBC调用到由DBMS直接使用的网络协议。这种方式允许从客户机到DBMS服务器的直接调用，是Intranet访问的一种行之有效的解决方案，因为多数这些协议是专用的，因此数据库提供商自己将成为这种驱动程序的主要来源。

4.4 JDBC API

4.4.1 使用方法

通常有两种常用的方法：applets 和 applications。

1) Applets

目前Java使用最多的是从网络中下载的applet，它们作为web文件的一个部分。其中有数据库存取 applet 和能够使用JDBC来接触数据库的applet。

最一般的情况下，对applet的使用是通过不可靠的边界的。例如从另外一个公司或者Internet上获得这些applet。于是称这个情况为"Internet"场合。然而applet也可能通过局域网下载。典型的applet在几个方面与传统的数据库应用程序有所不同：

(1) 不可靠的applet被严格地限制在它们被允许执行的的操作上。特别地，不允许它们存取本地的文件、不允许它们对任意的数据库建立网络连接。

(2) 就标识和连接网上数据库来说，Internet环境下的applet而临新的问题。当数据库与客户相隔很远的时候，效率的考虑也有所不同了。与局域网相比，Internet上数据库applet可能会碰到十分不同的反应时间。

(3) 已验证的applet（Trusted applets）是指那些已经被Java虚拟机器认定是可以信赖的applet。它们之所以被认为是可信的是因为它们已经对上了特定的密钥，或者用户认为从特定来源来的applet是可信的。在安全上它们与应用（appliction）相同，但是其他方面（例如定位一个数据库）与则与applet相似。

2) Applications

Java语言也可以用来构造常规性应用程序。随着Jaba语言开发工具完善，以及人们逐渐认识到这种发展中的程序语言的创造性和Jaba应用程序开发的优势所在，Java的这种应用将越来越普遍。

在应用Applications的情况下，Java的代码是可信的，并且同其他常用应用程序代码一样，能读写文件，进行网络连接等。

也许Java的Applications的最广泛的用途是应用于公司或企业内部网上，所以也可以称为商业网（Intranet）应用。例如一个公司可以通过GUI构造工具来用Java语言实现它的所有的应用程序，而GUI构造工具可以为基于共享数据库的形式产生Java的代码。这些Applications可以访问本地或是任何网络上的共享数据库服务器，当然它也能通过Internet来访问数据库，图4.5表示了Java Application访问网络数据库的情况。

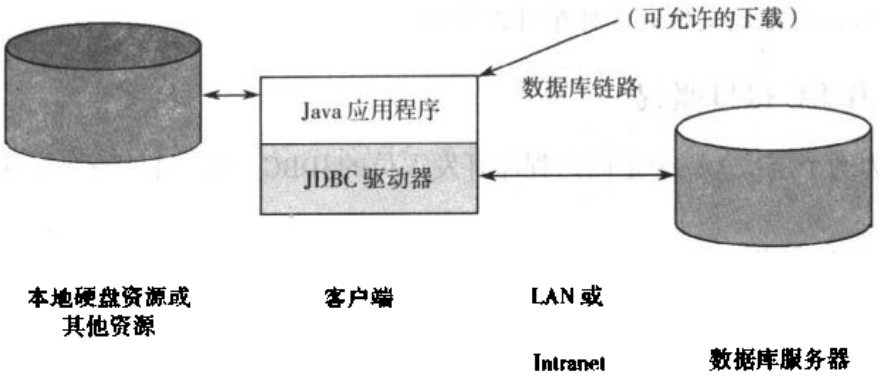

图 4.5　Application 访问网络数据库

4.4.2　安全性问题

作为网络应用就要考虑相应的安全问题，这在 JDBC 的两种方法中有以下体现：

1) Java Application 使用方法下，Java 的代码是本地的，因此是可信任的。同样出于安全考虑，可信任的 applet 也可归于此类。

2) 相反，如一个 Java applet 是不可信任的，就不能允许存取本地机上的文件或是其他网络数据。

下面介绍具体问题。

1) JDBC 和不可信任的 applets

(1) JDBC 必须遵从标准的 applet 的安全模式；

(2) JDBC 必须保证普通的无标志的 applets 是不可信任的；

(3) JDBC 将不允许不可信任的 applet 存取本地的数据库系统的数据；

(4) 若下载了一个已向JDBCDriveManager注册的JDBC驱动程序，则只有与该驱动程序从同一个服务器上下载的代码中提出连接要求，JDBC 才能利用它来进行连接；

(5) 一个不可信任的 applet 通常只能对它所下载的服务器进行连接；

(6) 在与远程数据库服务器进行连接时，JDBC应当避免自动或盲目使用本地机上的私有信息。

若一个JDBC的驱动层能完全保证进行数据库的网络连接时，不会发生权限被网上的其他主机或程序所窃取，则它会允许一个 applet 进行这种连接。

这种对于不可信任的applet的限制的确很繁琐，但这与一般applet安全模式是一致的，若取消的话，会带来很多不便。

2) JDBC 和 Java 的应用程序

对于一个普通的Java应用程序（不包含 applet 的Java代码），JDBC将很迅速地从本地路径上装载驱动程序，并允许自由地存取文件及远程服务器上的资源等。

而对于 applet 来说，若从远程服务器上下载一个不可信任的 sun.sql.Drive 类，那么这个 Drive 只能与从同一源服务器上下的代码一起使用。

3) 驱动程序在安全方面的职责

(1) 共享 TCP 连接；

(2) 检查所有的本地文件通路作最坏的准备。

4.4.3 JDBC接口概貌

接口分为两个层次，一个是面向程序开发人员的JDBC API。另外一个是底层的JDBC Driver API。

1) JDBC接口定义

(1) JDBC API。JDBC API被描述成为一族抽象的Java接口，使得应用程序可以对某个数据库打开连接，执行SQL语句并且处理结果。图4.6表示了JDBC API接口之间的关系。

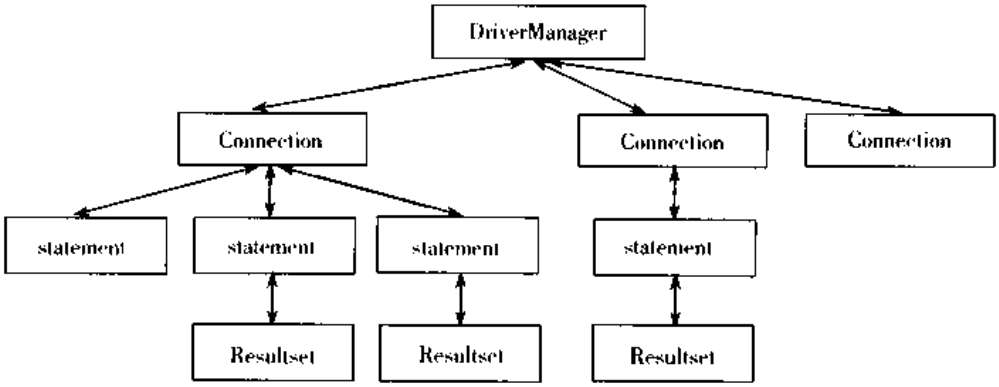

图4.6 JDBC API的类

JDBC API被定义在java.sql包中，其中定义了JDBC API用到的所有类、接口和方法，主要的类和接口有：

类DriverManager——处理驱动程序的装入，为新的数据库连接提供支持，驱动程序要向该类注册后才能被使用，当进行连接时该类根据JDBC URL选择匹配的驱动程序；

接口java.sql.Driver——驱动程序接口，负责确认URL与驱动程序的匹配、建立到数据库的连接等，其中的方法需要有相应的驱动程序实现；

接口java.sql.Connection——表示到特定数据库的连接，其中的方法需要有相应的驱动程序实现；

接口java.sql.Statement——为SQL语句提供一个容器，包括执行SQL语句、取得查询结果等方法；

接口java.sql.ResultSet——提供对结果集进行处理的手段。

其中java.sql.Statement又有两个子类型：

- java.sql.PreparedStatement　用于执行预编译的SQL语句。
- java.sql.CallableStatement　用于执行对一个数据库内嵌过程的调用。

(2) JDBC Driver API。大部分JDBC驱动器只需要完成上述这些JDBC API所定义的抽象类就可以了。特别是，所有的driver必须提供对java.sql.Connection，java.sql. Statement，java.sql.PreparedStatement，java.sql.ResultSet的实现。如果目标DBMS提供有OUT参数的内嵌过程，还必须提供java.sql.CallableStatement 接口。 每个database driver必须提供

java.sql.Driver 类，以使得系统可以由 java.sql.DriverManager 来管理。

一个显然的driver是在ODBC之上提供对JDBC的实现，从而提供与ODBC接口的JDBC-ODBC 桥，就像图 4.3 所显示的。由于 JDBC 放在 ODBC 之后，所以实现起来简单而且高效。

2) 数据库连接

(1) 为用户访问数据库建立一个连接。使用 JDBC 管理层 java.sql.DriverManager.getConection 方法，产生一个 java.sql.Conection 对象。该方法使用一个 URL 串作为参数。

(2) 通过指定驱动程序的名称，(或缺省)，选择合适的驱动程序。

(3) 由 URL 指定要连接的数据库，此时可称为 JDBC URL，其格式为：

jdbc:< 子协议 >:< 子名称 >

如果对网络数据库访问，建议使用标准 URL 作为子名称的一部分。

例如对数据资源名为 fred 访问的 URL 可能是：

jdbc:odbc:fred 或 jdbc:dbnet://wombat:356/fred

子协议 odbc 表示对 ODBC 数据源的访问，其格式为：

jdbc:odbc< 数据资源名 >[：< 属性名 >=< 属性值 >]

连接参数由 java.util.Properties 对象指出。建议大多数参数不要在此处给出，而在协议中指出。

支持多连接：一个应用程序可以使用一个或多个驱动程序建立与多个数据库连接。

驱动程序注册：有两种方法，一是在 JDBC java.sql.DriverManager 类初始化时查找“sql.driver”特性，对每一个驱动程序自动注册；二是由标准 Class.forName 方法显示加载一个驱动程序，参数为驱动程序名。

3) 数据传递和结果接收

(1) 查询：执行一条查询语句后，返回的是由 java.sql.ResultSet 对象访问的行的集合。在该对象中提供了一系列“get”方法，访问当前的每一列，ResultSet.next 方法可实现结果集的行之间移动，可以使用列索引或列名指定相应的列。

(2) 传递 IN 参数：java.sql.PreparedStatement 接口提供了一系列 setXXX 方法向 SQL 语句传递参数，实现动态的 SQL 语句。

在传递参数时必须满足数据类型一致的要求。因此，必须预先调用类型转换方法完成数据转换，同时也提供了传递 SQL 空值和常数给 IN 参数的方法。

(3) 接受 OUT 参数：在调用一个存储过程时，可用 setXXX 方法传递 IN 参数，使用 OUT 参数接受返回结果。在使用时必须先调用 CallableStatement.registerOutParameter 方法为每一个 OUT 参数进行类型注册，然后执行该过程调用语句，最后使用 getXXX 方法取出 OUT 参数的结果。

(4) 数据截断：在某种情况下，可能读或写数据时出现数据截断，如当由 Connection.setMaxFieldSize 设置了一个域的最大长度时，超过设置的最大长度就被截断。

4) SQL 数据类型到 Java 类型的转换

由于 SQL 数据类型与 Java 类型的转换差异较大，相互转换的类型之间还是存在一些差异，因此 JDBC 提供了详细的从 SQL 类型到 Java 类型标准转换表和从 Java 类型到 SQl 类型的标准转换表。

4.4.4 进一步了解 JDBC API

1) 异步、线程和交易

(1) 异步：某些请求数据库 API 提供了 SQL 语句异步执行的机制，这样可以使一个数据库的操作在后台运行的同时，前台一边等待一边处理其他操作。由于Java提供了多线程机制，因此并不真正需要实现异步SQL语句执行。当需要异步执行时可通过新建一个线程来实现数据库操作。

(2) 多线程：对Java.sql的所有对象的操作是多线程安全的，并且当多线程同时访问一个对象时，也保证操作正确性。尽管不同的驱动程序并发的程度不同，仍可假定为完全并发执行的。因为驱动程序若需要某种形式的同步操作，则一定会提供相应的实现机制。

(3) 交易：每一个新的JDBC连接都初始化为“自动提交”模式，即意味着每一个语句作为一个分开的交易来执行。当需要把多个语句作为一个交易来执行时，可以调用Connection.setAutoCommit(false)方法，取消自动提交。执行一个完整的交易后，调用Connection.commit显示完成提交，或调用Connection.rollback卷回整个交易操作。当一个交易被提交或卷回后，关闭所有在此连接上的 PreparedStatments，CallableStatements 和 ResultSets，只有简单的 Statements 是打开的状态。

2) 游标

JDBC 支持简单游标，这里所说的游标是指 SQL 数据库里的概念。应用程序可以用 ResultSet.getCursorName()方法，取得与当前 ResultSet 相关联的游标，利用该游标可以对当前行进行修改和删除。游标的有效期是到 ResultSet 或其父语句结束。

3) 对 SQL 的扩充

JDBC 全部支持 SQL-2 基本（Entry）级规范，部分支持 SQL-2 过渡（Transitional）级规范。对 SQL-2 基本级扩充有两点：一是支持 DROP TABLE 命令；二是选定的过渡级语义必须通过 Escape 语法来支持，以便一个驱动程序可以方便地 扫描和翻译特定的 DBMS 语法。

(1) SQLE scape 语法：在存储过程、标量过程、标量函数、日期、时间、输出连接等方面，JDBC 支持与 ODBC 相同的与 DBMS 无关的 Escape 转义语法，格式为：

{关键字 ..... 参数 .....}

(2) 存储过程：JDBC 激活一个存储过程语法格式为：

{call 过程名[参数 1, 参数 2，.....]}

或者是带返回结果参数的过程：

{?=call 过程名[参数 1, 参数 2，.....]}

(3) 日期和时间：JDBC 支持标准格式日期和时间的字符串表示，用 Escape 转义语句表示日期和时间，如{d'yyy-mm-dd'}或{t'nn:mm:s'}分别表示日期和时间。

(4) 标量函数：JDBC支持标量值的数值、串、时间、日期、系统和转换函数，如：{fnconcat ("Hot ","Java")}。

(5) 输出连接：语法格式为

{oj outer-join}

其中 outer-join 形式为

table LEFOUTRJOIN{ outer-join }ON search-condition

4.5 JDBC 应 用

4.5.1 数据库建立连接

所有JDBC的程序的第一步都是与数据库建立连接，得到一个java.sql.Condition类的对象，对这个数据库的所有操作都是基于这个对象。

1) 加载驱动程序

为了与数据库建立连接，JDBC必须加载驱动程序。驱动程序可以是JDBC-ODBC桥驱动程序，JDBC 到通用网络协议的驱动程序，或者由数据库厂商提供的驱动程序；或可以通过设置Java属性中的sql.driver来指定驱动程序列表。这个属性是一系列冒号隔开的driver类 的名称。如 cme.wonder.Driver:foobaz.openNet. Driver:vender.Our. Driver。JDBC 将按照列表搜索驱动程序，并使用第一个能成功地与给定的 URL 相连的驱动程序。

2) 建立连接

DriverMannager类的getConnection方法用于建立与某个数据源的连接。来与url对象指定的数据源建立连接，若连接成功，则返回一个 Connection 类的对象 con。之后对于这个数据源的操作都是基于这个对象。getConnection 方法是 DriverMannager 类中的静态方法。所以使用时不产生DriverMannager类的对象，直接使用类名DriverMannager调用就可以了。

3) 检查警告信息

若连接失败，则产生SQLException 例外。成功时也可能产生一些警告信息，使用前检查这些警告信息是一个好习惯。Connection类的getWarning方法返回一个SQLWarning类的对象，这是一个警告信息。在我们定义的 checkForWarning 方法中，将检查所有的警告信息。如果 warn 参数不为 null，说明警告存在。将 warn 的一些信息输出，然后获取下一个警告，直至没有。

getSQLState方法给出目前的SQL状态，getMessage给出对警告的说明，getErrorCode则给出数据库厂商提供的错误代码。

4) 获得关于数据库源的信息

建立连接之后就可以对数据库进行各种操作。语句 DatabaseMetaData dma= con.getMetaData( )返回一个 DatabaseMetaData 类的对象，从中可以获得许多关于数据的信息。

5) 关闭连接

对于任何一个连接，当不再对数据库操作时，应该将其关闭。语句 con.close()关闭连接 con。

6) 捕获例外

在建立连接和对数据库进行操作过程中，都可能产生例外。在JDBC中经常遇到的例外是 SQLException。

7) 获取和设置连接选项

对于一个连接可以设置一些连接选项，如：是否只读，是否自动提交，是否自动关闭等等。有一组方法可以设置这些连接选项，还有一组方法用于获得连接选项的值。如setReadOnly用于设置是否只读，而isReadOnly用于获得连接是否只读。

4.5.2 执行查询语句

JDBC中查询语句执行方法可分为三类：Statement,PreparedStatement和CallableStatement对象。

1) Statement

(1) 创建Statement 对象。Statement 对象主要用于一般语句的查询，若执行一个SQL查询语句，必须建立一个Statement 对象。Connection 类的createStatement 对象方法用于建立一个Statement 对象。Statement stmt=con.create Statement ();

(2) 执行查询语句。在Statement 对象上，可以使用execQuery 方法来执行一查询语句。execQuery 的参数是一个String对象，即一个SELECT语句。它的返回值是一个Resultset类的对象。

ResultSet rs= stmt. execQuery("SELECT*FROM Customer");

该语句将返回表中的所有行。

ResultSet rs= stmt. execQuery("SELECT FirstName,Adress from Customer WHERE CustomerID<5");

上面的语句将在结果集RS中返回CustomerID小于5的FIRSTNAME列和Address列。

ExecQuery 方法一般用于执行一个SELECT语句，它只返回一个结果集。而Execute方法可用于返回多个结果集的情况。一般情况下是不会返回多个结果集的。

JDBC在编译时不对将要执行的SQL语句作任何检查，只是将其作为一个String对象。直到驱动程序执行SQL语句时才知道其正确与否。对于错误的SQL语句，在执行时会产生Sqlexception。

一个Statement对象在同一时间只能打开一个结果集，对第二个结果集的打开隐含着对第一个结果集的关闭。如果想对多个结果集同时进行操作，必须创建多个Statement对象，在每一个Statement对象上执行SQL语句获得相应的结果集。如果不需要同时处理多个结果集，那么可以在一个Statement对象上顺序地执行多个SQL语句，对获得的结果集进行顺序操作。

(3) 获取和设置选项。对于Statement对象也可以设置许多选项，下面给出其中的几个，详细说明请参见有关JDBC API资料。

SetMaxRows设置结果能容纳的最多行数，超过此行数的结果将被丢弃且不用通知用户。它的参数是一个int值。GetMaxRdws方法返回目前结果集能容纳的最多行数。

getQueryTimeOut设置一个语句的执行等待时间，它的参数是以秒为单位的INT值。若驱动程序等待一个语句执行的时间超过了此值，则产生一个SqlExeption。getQueryTimeOut返回目前这个选项的值。

Setescapeprocessing通知驱动程序如何处理转义字符。它的参数是一个Boolean值。若为TRUE，则驱动程序在将SQL语句送至数据库之前进行转义字符的替换（这也是缺省动作）。若为False则驱动程序不作替换，由数据库自己负责替换转义字符。

(4) 关闭 Statement。

stmt.close ( );

一个无用的Statement对象在无用单元收集时将会被自动关闭，但是在使用完Statement对象后立即关闭是一个好习惯。在关闭Statement对象时，如果其上还有结果集，该结果集也将被关闭。

2) PreparedStatement

Statement 对象在每次执行 SQL 语句时都将该语句转给数据库，在多次执行同一语句时，这样做效率较低。这时可以使用PrepareStatement对象。如果数据库支持预编译，它可以将 SQL 语句传给数据库做预编译，以后每次执行这个 SQL 语句时，速度就可以提高很多。如果数据库不支持预编译，则在语句执行时，才将其传给数据库。这对用户来说是透明的。

PreparedStatement对象的SQL语句还可以接受参数。在语句中指出需要接受哪些参数，然后进行预编译。在每一次执行时执行可以给SQL语句传输不同的参数，这样就大大提高了灵活性。

PreparedStatement类是Statement类派生的子类，因此它可以使用Statement类中的方法。

(1) 创建PreparedStatement对象。从一个Connection对象可以创建一个PreparedStatement对象。在创建时，应该给出要预编译的 SQL 语句。例如：

OreoareStatement pstmt=con.PjrepareStatement ("SLECT*FROM-Customer");

(2) 执行查询语句。 PreparedStatement 对象也使用 execQuery 方法来执行语句。与 Statement 类不同的是该方法没有参数。这是创建 PreparedStatement 对象时，已经给出了要执行的 SQL 语句，并进行了预编译。执行时，只需要执行已编译好的语句就可以了。

ResultSet rs=pstmt.execQuery();

上述语句可以被执行多次，无须重新给出 SQL 语句。同 Statement 类一样它返回一个 Result 对象。

PreparedStatement 类也有一个不带参数的 Execute 方法，用于可能 返回多个 Resultset 的语句的执行。它的返回值和处理方法同Statement类的 Execute 方法一样，这将有下节中讲述。

(3) 获取和设置选项。PreparedStatement类是使用它的父类Statement类的相应方法来获取和设置选项的。

(4) 关闭对象。PreparedStatement 对象也是 Close 方法来关闭的，实际上，它是使用父类 Statement 的 Close 方法。

3) CallableStatement

CallableStatement 对象用于执行数据库中的存储过程。存储过程是数据库中已经存在的SQL查询语句。执行该存储过程的结果同执行相应的SQL语句是一样的。存储过程可以有输入参数，也可以有输出参数。在 Access 中存储过程称为 QUERY。

(1) 创建 CallableStatement 对象。Connection 类的 Preparecall 方法可以创建一个 Callablestatement 对象。它的参数是一个 String 对象，一般格式为 "{CALL Procedurename ( )}"，其中 Procedurename 是存储过程的名称，在 Access 中，就是 QUERY 的名称。

例如，要执行数据库例子中的QUERY1，可以用以下语句创建CallableStatement对象：

Callablestatement cstmt=con. PrepareCall（"{CALL QUERY1（)}"）;

(2) 执行存储过程。CallableStatement 类是使用父类 PreparedStatement 类的 Exequery 方法或 Execute 方法来执行存储过程的。例如：

Resultset rs=cstmt.execteQuery( );

(3) 获取和设置选项。Callable Statement 类是使用 Statement 类的相应方法来获取和设置选项的。

(4) 关闭 CallableStatement。CallableStatement 对象也是使用 Close 方法来关闭的，实际上，它是使用 Statement 的 Close 方法。

4.5.3 检索结果集

SQL查询语句执行的结果是一个ResultSet类的对象，要想把查询结果返回给用户，必须对对象进行处理。

1) ResultSet 的基本处理方法

ResultSet 对象包括一个由查询语句返回的一个表，这个表中包含所有的查询结果。对 ResultSet 对象的处理必须逐行进行，而对每一行中的各个列，可以按任何顺序处理。

ResultSet 对象维持一个指向当前的指针。最初，这个指针指向第一行之前，ResultSet 类的next方法使这个指针指向下一行。因此，第一次使用next方法将指针指向结果集的第一行，这时可以对第一行的数据进行处理。处理完毕后，使用next方法，使这个指针指向下一行，继续处理第一行的数据。next方法返回的是一个boolean值，若为TRUE，则说明指针成功地移向下一行，可以对该行的数据进行处理，相反若为FALSE，则说明没有下一行，即结果集处理完毕。

在对每一行进行处理时，可以对各个列按任意顺序进行处理，不过，按从左至右的顺序可以获得较高的执行效率。ResultSet 类的 getXXX 方法可以从某一列中获得结果。其中 XXX 是 JDBC 中的 Java 数据类型，如 getInt、getString、getData 等。getXXX 方法需要指定要检索的列。有两种指定列的方法：一种是以一个 int 值作为列的索引，另一种是以一个 String 对象作为列名来索引。

以下是一个简单的检索结果的一段程序：

```
Statement stmt=con.creatStatement();
    ResultSet=stmt.executeQuery("SELECT a, b, c,from table");
while(r.next()){
            int i=r.getInt(1);
            String s=r.getstring("namer");
            byte b[]=r.getBytes(3);
System.out.println("i+" "+s+""+B[0]" );
        }
```

2) 处理多个结果集

通常情况下，使用 executeQuery 或 executeUpdate 来执行 SQL 语句。executeQuery 用来执行查询语句，它返回一个 ResultSet 对象。executeUpdate 用来执行收据更新语句，它返回更新的行数。然而在某些情况下，在语句执行前，并不知道是否返回一个结果集。另外，

某些存储过程可能返回多个结果集和更新计数。这就需要有种机制来处理多个结果集和更新计数。这时使用execute方法，并使用getResultSet, getUpdateCount和getMoreResults方法来处理返回结果集和更新计数。

对于Statement对象，可以使用execute（String sql）方法，而对于PreparedStatement对象和CallableStatement对象，可以使用execute()方法，它们都返回一个boolean值，指示第一个结果的类型。如果第一个结果是ResultSet类型，则返回true，否则返回false。

如果当前结果是ResultSet类型，要使用getResultSet方法获取当前结果集。然后如前所述，对结果集进行操作。

如果当前结果是一个更新计数，可以使用getUpdateCount方法来获取其值。如果当前结果是一个ResultSet对象或不再有结果，该方法返回-1。应该首先检验一个结果是否是ResultSet类型，若不是则可能是更新计数，或不再有结果。

getMoreResults返回一个boolean值，如果下一个结果是ResultSet类型，则返回true，否则返回false。判断不再有结果的条件是（!getMoreResults()&&(getUpdateCount==-1)）。

通过上述三种方法交替使用，就可以依次处理各个结果集和更新计数。

4.5.4 更新数据库操作

1) 对表中记录的操作

对一个表中的记录可以进行修改、插入和删除操作。分别对应于SQL的UPDATE，INSERT，DELETE操作，同SELECT操作类似，executeUpdate方法的参数是String对象，即要执行的SQL语句。它返回的不是ResultSet对象，而是一个整数。对于UPDATE，INSERT，DELECT操作，这个整数是操作所应影响的记录数。对于其他不返回值的SQL语句，executeUpdate的方法的返回值为零。

例如，下面的语句将Customer表中FirsrName为"Li"的记录的Address项改为Beijing:

```
stmt.executeUpdate("UPDATE Customer SET Address='Beijing' WHERE FristName="Li");
```

而下面的两个语句在Customer表中增添和删除记录：

```
stmt.executeUpdate("DELETE FROM Customer WHERE Address='Beijing'");
stmt.executeUpdate( "INSERT INTO Customer (Customer(CustomerID,First Name)
VALUES(9,'Liu')");
```

在增添记录时，如果写上列名，则VALUES中的值赋给相应的列，对于没有写出的列，其值为NULL。如果INSERT语句不写列名，则VALUES中的值按顺序赋给每个列。

2) 创建和删除表

创建和删除一个表对应于SQL的CREATETABLE和DROPTABLE语句，也是使用statement对象的executeUpfdate方法来完成。

下面语句创建一个表another，它有两列：列ID为整形值，列Name为字符串：

```
stmt.executeUpdate("CREATETABLE another(ID INTEGER, Name VARCHAR(20)");
```

CREATETABLE语句只是给出了表中的列名和数据类型，并没有加入记录。要加入记录，应使用上面讲到的INSERT语句。删除一个表，要用DROPTABLE语句：

```
stmt.executeUpdate("DROPTABLE another");
```

3) 增加和删除表中的列

对一个表的列进行更新操作是使用SQL的ALTERTABLE语句。对列进行的更新操作要影响到表中的所有的行。

下面语句在 another 表中增加一列 Address，数据类型为字符串：

```
stmt.executeUpdate("ALTERTABLE another ADDCOLUMN Address VARCHAR(50)");
```

在增加一列以后，表中以前存在的行的列值为NULL。可以使用UIPDATE 语句设计此列的值，还可以使用 INSERT 语句增加新的记录。

删除表中的一列语句如下：

```
stmt.executeUpdate("ALTERTABLE another DROPCOLUMN Address");
```

4) 使用 PreparedStatement 对象

同 SQL 查询语句一样，数据更新语句也可以在 PreparedStatement 对象上执行。使用 PreparedStatement 对象，只需要传递一次 SQL 语句，可以多次执行它。并且可以利用数据库的预编技术，提高报告效率。

使用 PreparedStatement 对象的另一个好处是可以接受参数。

4.5.5 参数的输入和输出

JDBC 允许在要执行的 SQL 语句中设置参数，这样就给数据库操作带来很大的方便。要想使用SQL 语句的输入参数或输出参数，必须在PREPAREDSTATEMENT对象上进行操作。由于 CallableStatement 类是 PreparedStatement 类的子类，所以在 CallableStatement 对象上的操作也可以使用输入参数和输出参数。在生成 PreparedStatement 对象或 CallableStatement 对象时，SQL语句中指定输入或输出参数。在执行这个SQL 语句之前，要对输入参数进行赋值。

1) 使用 PreparedStatement 对象

PreparedStatement 对象上的查询语句和更新语句都可以设置输入参数 。在生成 PREPAREDSTATEMENT对象时，在SQL语句使用之前，使用setXXX 方法给参数赋值，然后使用 executeQuery 或 executeUpdate 来执行这个SQL语句。每一次执行SQL语句之前，可以给参数重新赋值。

setXXX 方法用于给相应的输入参数赋值。其中 XXX 是 JDBC 的数据类型，如：int，String的第一个参数的位置为1；第二个参数的位置为2；依次类推。setXXX 的第二个参数是要传递的值，如：100，“BEIJING”等，随 XXX 的不同而有不同的类型。

下面的一段程序在一个 SQLUpdate 语句中设置了二个参数，类型分别为 Steing 和 int。在循环中，反复执行这个SQLUpdate 语句，每次执行时，赋予第二个参数不同的整数值：

```
PreparedStatement pstmt=con.PrepareStatement("UPDATE table SET m=?WHETE x=?");
c.setString(1,"text");
for (int I=0;I<10;I++);{
pstmt.setInt(2,I);
Pstmt.executeUpdate();
}
```

2) 使用 CallableStatement 对象

有一些存储过程要求输入参数，例如给出的例子为数据库中的 Query2 和 Query3。这时可以在生成 CallableStatement 对象的存储过程调用语句中设置输入参数。在执行这个存储过程之前使用 setXXX 方法给参数赋值，然后再执行这个存储过程。

3) 输入参数的数据转换

setXXX 方法对数据类型不做任何转换，只是将 Java 的数据类型映射到相应的 SQL 数据类型。程序员应该保证参数的数据类型与数据库所需要的一致。JDBC 数据类型与 SQL 数据类型的关系请参见有关“JDBC 与 SQL 的数据转换”技术资料。如果必须进行数据类型转换，可以使用 setOBJECT 方法将一个 Java OBJECT 转换为 SQL 数据类型，然后再传递给数据库。

JDBC 允许将 SQL NULL 作为输入参数传给数据库。这要使用 setNull 方法。setNull 方法的第二个参数是一个整数，它代表这个参数所要求的 SQL 数据类型。这些 SQL 数据类型在 java.sql.Types 中有定义。另外，如果将 Java 的 NULL 值传递给 setNull 方法，那么传递给数据库的将是 SQL NULL。

JDBC 中的 setBytes 方法和 setString 方法并没有限制所传递数据的上限。但是当传递非常长的数据时，最好使用流机制，JDBC 中可以将一个 Java 输入输出流作为 SQL 语句的参数。有三种方法可以将输入输出流作为 SQL 语句的参数：setBinaryStream 用于二进制流，set ASCII Stream 用于 ASCII 码流，而 setUnicode Stream 则用于 UNICODE 码流。下面是一个使用二进制流的例子：

```
File file=new File("foo");
    int filelLength=file.length();
      InputStream fin=new FileInputStream(file);
        PreparedStatement pstmt=con.PreparedStatement("SELECT another SET stuff=?WHERE
index=4");
stmt.setBinaryStream(1,fin,fileLength);
stmt.executeUpdate();
```

4) 接收输出参数

某些存储过程可能会返回输出参数。这时在执行这个存储过程之前，必须使用 CallableStatement 的 RegisterOutParameter 登记输出参数。在 RegisterOutParameter 方法中要给出输出参数的相应位置以及输出参数的 SQL 数据类型。SQL 数据类型的值在 Java. SQL. Types 类中有定义。在执行存储过程以后，必须使用 getXXX 方法中要指出获取那一个输出参数的值。如果一个存储过程既返回结果集，又返回输出参数，那么在处理时，最好先处理结果集，然后再获取输出参数。

getXXX 方法不对数据类型作任何转换。在 RegisterOutParameter 方法中已经指定了数据库将返回的 SQL 数据类型。程序员应使用相应的 getXXX 方法来获取输出参数的值。

返回的输出参数的值也可能是 SQL NULL。为了判断输出参数的值是否为 SQLNULL，必须首先读取这个输出参数，然后使用 ISNULL 方法来判断其值是否为 SQLNULL。如果 ISNULL 方法返回 TRUE，则表明输出参数的值为 SQLNULL。这同 ResultSet 的处理方法是类似的。如果使用 getXXX 方法去读一个 SQLNULL 值，可能有以下几种情形：

- 返回值为 Java OBJECT 的 getXXX 方法将返回 Java NULL 值。
- getByte、getShort、getInt、getLong、getFloat 和 getDouble 将返回零值。
- getBoolean 将返回 FALSE 值。

JDBC 不支持将输出参数作为输出流来获取值，最好使用Resultset来返回较长的数据。在 Resultset 类中，可以使用流机制来获得数据的值。

5) 数据截断

在某些情况下，从数据库中读取的数据或向数据库中写入的数据可能被截断。对于数据库被截断的处理依赖于个体环境。一般情况下，在读取数据时发生截断将产生警告。而在写入数据时发生截断将产生 SQLException 例外。

如果程序使用 Connection 类的 setMaxFieldSize 方法设置了一个域的最大长度，那么超过此长度的读或写都将被截断，并且不产生任何SQLException 和 SQLWaring。JDBC 中截断读取数据的情况不常见，因为并不要求程序员指定数据缓冲区，而是按照需要来分配数据缓冲区。然而，在某些情况下，驱动程序可能受到内部实现的限制，从而在读取数据的时候进行数据截断。如果 ResultSet 中的数据被截断，那么一个 DataTruncation(SQLWaring 的子类)对象将被加到ResultSet的警告链表中，并返回尽可能多的数据。如果在输出参数上发生了数据截断，则一个 DataTruncation 对象将被加到 Callablestatement 的警告链表中，并返回尽可能多的收据。

如果驱动程序或数据库没有准备好接受 JDBC 程序传送的数据，就将对写入的数据产生截断，这时将产生一个 SQLException 例外。

4.5.6 动态数据库访问

一般情况下，程序员在JDBC编写程序时，已经了解要访问的数据库的情况，如各个表名，表中的各个列名以及了解要访问的数据库的特定情况，选取相应的方法来访问数据库并检索结果。然而在在某些情况下，JDBC程序需要动态地获得要访问的数据库的情况，然后再采取相应的方法去访问数据库和检索结果。JDBC 支持这种动态的数据库访问，以提高程序的灵活性。

1）数据库元信息的获取

在对数据库进行连接以后，得到一个 Connection 类的对象，可以从这个对象获得有关数据源的各种信息，包括关于数据库中的各种表，表中的各个列，数据类型和存储过程等各方面的信息。根据这些信息，JDBC 程序可以访问一个事先并不了解的数据库。获取这些信息的方法都是在 DatabaseMetaData 类的对象上实现的。

下面的语句可以在一个联接的基础上创建一个对象：

```
DatabaseMetaData dbma=con.getMetaData();
```

DatabaseMetaData 类中提供了许多方法用于获得数据源的各个方面的连系，通过这些可以非常方便地了解数据库。以下给出了一些常用的方法，其他的方法请参见 JDBC API 部分的 DatabaseMetaData 类。

这些方法中有些要用到字符串搜索匹配模式作为参数，这些匹配模式与 ODBC 中一样。字符‘_’匹配任何单个字符，‘%’匹配零个或多个字符。一个值为 null 的 Java String 可以与任何字符串匹配。

这些方法中有些返回的是一个结果集，即ResultSet对象，可以使用结果集的方法来处理它们。

对某些方法，有些驱动程序并不支持，这时会产生一个SQLException例外，例外中会输出消息说明驱动程序并不支持这个方法。

2) 结果集的动态访问

一般情况下程序员对结果集的结构是了解的，可以采用相应的方法来检索结果集。但是，有时对结果集中的结果并不了解，这是可以先通过ResultSetMetaData对象来获取结果集的有关信息。这已在4.5.3节中介绍过。

3) 动态数据访问

以前已经讲过SQL数据类型和Java数据类型之间的映射关系，例如：一个SQL INTEGER类型的数据一般映射为Java的int类型，可以使用一种简单的Java数据类型来读写SQL数据

JDBC为了支持通用的数据访问，提供可以将数据作为Java Object来访问的方法。这就是ResultSet类的get Object方法，PreparedStatement类的setObject方法和CallableStatement和getObject方法。对于两个getObject方法，在获取相应的数据类型之前，必须将Object对象转换为相应的类型。

某些Java的数据类型，如boolean和int并不是Object的子类，因此，从SQL类型到Java Object类型的映射可能稍有不同。

(1) ResultSet.getObject。ResultSet类的getObject方法返回一个Object对象，它的类型由结果集中相应列的SQL类型决定，请参见表4.1。

表4.1　SQL类型到Java Object的映射

SQLtype	Java Object Type	SQLtype	Java Object Type
CHAR	String	BIGINT	long
VARCHAR	String	REAL	float
LONG VARCHAR	String	FLOAT	double
NUMERIC	java.sql.Numeric	DOUBLE	double
DECIMAL	java.sql.numeric	BINARY	byte
BIT	boolean	VARBINARY	byte
TINYINT	integer	VARBINARY	byte
SMALLINT	integer	LONG VARBINARY	byte
INTEGER	integer	DATE	java.sql.Date
TIME	java.sql.Time	TIMESTAMP	java.sql.TimeStamp

例如，如果结果集中的列“a”的SQL类型是CHAR，而列“b”的SQL类型是SMALLINT，可以使用下列语句获取其值：

```
ResultSet rs=stmt.executeQuery("SELECT a,b FROM foo");
```

```
While(rs.next( )){
  Object x=rs.getObject("a");//gets a String
  Object y=rs.getObject("b");//gets an integer
```

(2) PreparedStatement.setObject 方法。PreparedStatement 类的 setObject 方法，可以指定目标 SQL 类型。Java Object 首先被映射为缺省的 SQL 类型，然后转换为指定的 SQL 类型（见表 4.2），再传给数据库。也可以缺省目标 SQL 类型，这时 Java Object 只是简单的转换为缺省的 SQL 类型，再传给数据库。

表 4.2　Java Object 到 SQL 类型的映射

Java Object Type	SQL type	Java Object Type	SQL type
String	LONG or VARCHAR	byte[]	VARBINARY or LONGVARBINARY
Java.sql.numeric	NUMERIC	double	DOUBLE
boolean	BIT	java.sql.Date	DATE
integer	INTEGER	java.sql.Time	TIME
long	BIGINT	java.sql.Timestamp	TIMESTAMP
float	REAL		

(3) CallableStatement.setObject 方法。在调用 CallableStatement 类的 setObject 的方法之前，必须使用 CallableStatement 类 RegisterOutParameter 方法来指定参数的 SQL 类型。驱动程序将根据这个 SQL 类型返回一个 Java Object 类型（参见表 4.1）。

4.5.7　JDBC 中的例外

JDBC 程序在运行时，除了产生一些 Java 例外，进行数据库访问时，还可能产生 SQLException 类型的例外。此外，JDBC 中还定义了 SQLException 的子类：SQLWarning 和 Datatruncation。

1) SQLException

SQLException 是 java.lang. Exception 的子类，它提供了关于数据库范访问错误的信息。每个 SQLException 类提供了以下几种信息：

(1) 一个描述错误的字符串。它作为 Java Exception 的消息，可以通过 getMessage( )方法来获取这个字符串。

(2) 一个SQLstate字符串。它遵循XOPEN的SQLstate标准。SQLstate字符串的值在XOPEN的 SQLstate 标准中有描述。它可以通过 getSQLstate 方法来获得。

(3) 一个由数据库厂商提供的错误代码，它是一个整数值。通常是低层的数据库返回的真正错误代码。可以通过 getErrorCode（ ）方法来获得。

(4) 到下一个例外的连接，可以提供额外的错误信息。它可以通过 getNextExcepton 方法来获得下一个例外，返回的是一个 SQLException 对象。

下面一个程序可用于捕获 SQLException 例外，并显示错误信息：

```
try {
// 数据库操作
}
catch(SQLException ex){
System.out.println("\n*** SQLException caught***\n);
While(ex!=null){
System.out.println("SQLstate:"+ex.get SQLstate());
System.out.println("Message:"+ex.getMessage());
System.out.println("Vendor:"+ex.getErrorCode());
ex=ex.getNextException();
System.out.println(" ");
    }
}
```

2) SQLWarning

SQLWarning 是SQLException的子类,它提供类数据库访问的警告信息。如果一个对象的方法在访问数据库时产生了警告信息，则 SQLWarning 被加到这个对象的链上。与 SQLException 不同，SQLWarning并不产生一个例外来打断程序的运行。可以使用该对象的getWarning方法来获取SQLWarning对象。下面一段程序用来检查数据库连接过程中的警告信息，它使用 checkForWarning 方法来检查并显示警告信息：

```
connection con=DriverManager.getConnection(url);
checkForWarning(con. getWarnings( ));
```

由于 SQLWarning 是 SQLException 的子类，所以 SQLException 类中的方法，如getSQLState( ),getErrorCode( )和getMessage( )都可以使用。下面checkForWarning方法是一个检查 SQLWarning 的一段程序：

```
private static boolean checkForWarning(SQLWarning warn)
    throws SQLException
{
boolean rc=false;
 if (warn!=null){
System.out.println("\n***Warning***\n);
 Rc=true
While(warn!=null){
        System.out.println("SQLstate:"+warn.getSQLstate( ));
System.out.println("Message:"+ warn. getMessage( ));
System.out.println("Vender:"+warn. getErrorCode( ));
   System.out.println("");
Ex=warn. getNextException( );
}
    }
```

```
return rc;
}
```

3) DataTruncation

DataTruncation 是 SQLException 的子类。当 JDBC 例外截断一个数据时，它产生一个 DataTruncation 警告(读操作时)或一个 DataTruncation 例外(写操作时)。DataTruncation 的 SQLstate 被设置为 "01004"。

下面是 DataTruncation 类的几个方法，通过这些方法，可以获得关于数据截断的一些信息。关于 DataTruncation 的详细说明，请参见 JDBC API 中的 Exception 部分：

(1) int getIndex()。返回被截断的列或参数的位置索引。如果不能确定列或参数的位置索引，则返回 -1；

(2) boolean getParameter()。如果被截断的数据是一个参数，则返回True，否则返回False；

(3) boolean getRead()。如果截断发生在读操作，则返回 True，如果截断发生在写操作，则返回 FALSE；

(4) int getDataSize()。获取数据应该传送的字节数。如果数据的大小未知，则返回 -1；

(5) int getTransferSize()。返回数据实际传送的字节数，如果不能确定，则返回 -1。

4.5.8 JDBC中的其他问题

JDBC 完全支持 SQL_2 Entry Level 定义的 SQL 语句，某些超出 SQL_2 Entry Level 但被广泛支持的SQL语句也被支持。然而，ANSI定义的SQL下一个层次：SQL_2 Transitional Level 并没有被广泛支持，它的语法在不同 DBMS 中经常是不同的。

定义了两种扩展，要求的驱动程序必须能支持这两种扩展：

一种是必须支持的 SQL_2 Transitional Level 的语法和语义。目前 JDBC 所要求的这类语句只有一个，即一个 DPO TABLE 语句。

另一种必须支持的 SQL_2 Transitional Level 的语义是通过转义语法来实现的。驱动程序可以很容易地查询并将其翻译成为特定的DBMS要求的语法。只有在下层的数据库支持相应 SQL_2 Transitional Level 的语义时，才需要支持转义语法。

一个支持 Microsoft 定义的ODBCCoreSQL的ODBC驱动程序也支持本节中定义的JDBC SQL：转义语法、存储过程、时间和日期、标题函数、转义字符、外部连接。

1) 异步执行

某些数据库的 API，如 ODBC，允许 SQL 语句异步执行。这允许应用程序在启动一个数据库操作，等待这个操作的结果的时候，可以去处理其他的工作。Java 语言是一个多线程的环境，因此没有必要为语句的异步执行提供 API，程序员可以很方便地创建一个独立的线程去异步执行的 SQL 语句。

各个驱动程序对并发执行的支持并不相同。但是程序员可以假定为完全并发地执行。如果驱动程序需要某种形式的同步，对于程序员来说，唯一的区别就是程序的并行性降低。

例如，在同一个 Connection 上的两个 Statement 可以并发地执行，它们的结果集也可以并发地处理。一些驱动程序可能提供完全的并发支持，而另一些驱动程序可能等到第一个语句执行完毕，才发送第二个语句。

使用多线程可以取消一个长时间运行的语句。一个线程执行这个语句的时候，另一个线程可以使用Statement类的Cancel()方法来取消它的执行。

2) 事务处理

JDBC支持数据库访问中的事务处理。由于事务处理是依赖于底层的数据库实现的，因此不同的驱动程序对事务处理的支持程度可能有所不同。

一个新打开的JDBC连接被初始化为自动提交模式，即每个语句作为一个独立的事务被执行。为了将几个语句作为一个事务来执行，必须调用Connection类的setAutoCommit(false)方法来自动提交。当自动提交被禁止时，总是有一个事务隐含地与连接相连系。可以使用Connection类的Commit方法来提交事务，或使用abort方法来放弃事务。提交或放弃事务总是隐含着启动一个新的事务。

事务的精确语义依赖于底层的数据库实现。使用DatabaseMetaData类的方法来获取当前的缺省状态，可以使用Connection类的方法将一个新打开的连接置为同一缺省状态。

当放弃一个事务时，缺省动作是所有的PreparedStatement,CallabeStatement和连接上的结果集都被关闭，普通的Statement将保持在打开形状。

3) 游标

JDBC支持简单的游标操作。可以使用Result类getCursorName()的方法来获得一个与当前ResultSet结果集相关联的游标，利用该游标可以对当前行进行修改和删除。游标在语句关闭之前一直是合法的。

并不是所有的DBMS都支持定位更新和删除。可以使用DatabaseMateData类supportPositionDelete方法和supportPositionUpdate的方法来了解数据库支持这些操作。详细说明请参见JDBC API部分的DatabaseMateData类。如果获得支持定位更新和删除，数据库或驱动程序应该保证所选取的适当地被加锁，保证定位更新不会导致异常或其他并发性错误。

4.6 应用Java JDBC开发二层C/S数据库应用程序

前面已经论述的1、2、4型JDBC驱动程序都属于以客户端为中心的JDBC驱动程序，可以建立二层C/S数据库访问模式。

4.6.1 JDBC-ODBC桥驱动程序开发数据库应用

ODBC（开放式数据库互连：Open Database Connectivity）是用C语言写的，在多种不同的DBMS（数据库管理系统）中存取数据的标准应用程序接口；目前应用最广的是微软的ODBC，它几乎可将所有平台的所有数据库连接起来。ODBC在应用程序与特定数据库之间插入一个驱动程序管理器，每种数据库引擎都需要向驱动程序管理器注册它自己的ODBC驱动程序，驱动程序管理器将与ODBC兼容的SQL请求从应用程序传给ODBC驱动程序，并由ODBC驱动程序把SQL请求翻译为对数据库的固有调用，从而达到应用程序访

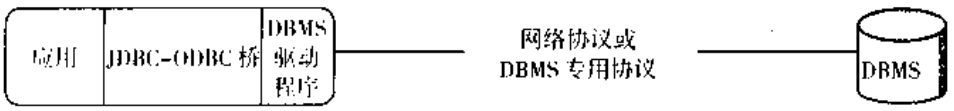

图 4.7　JDBC-ODBC 桥接方式使 Java 应用程序访问数据库

问操作 数据库的目的

JDBC 采用 JDBC-ODBC 桥接方式使 Java 应用程序使用 ODBC，见图 4.7。

用 JDBC 编写访问、操作数据库的 Java 应用程序，一般做下面三件事：

(1) 加载 JDBC-ODBC bridge 驱动程序，可如下使用 Class.forName 方法显式加载驱动程序来完成：

```
Class.forName("sun.jdbc.odbc.JdbsOdbcDriver");
```

(2) 建立数据库的连接、发送访问、操作数据库的 SQL 语句；

(3) 处理对数据库访问操作的结果。

以下给出一个简单的程序实例：

```
//JDBC to ODBC Bridge
import java.net.URL;
import java.sql.*;
class SimpleSelect {
public static void main (String args[]) {
        String url = "jdbc:odbc:FoxPro";
String query = "SELECT * FROM tmp";
try {
    // 调用 jdbc-odbc 桥 driver
    Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
    DriverManager.setLogStream(System.out);
    // 试图连接一个 driver. 每一个以注册的 drivers 被调用直到
    // 找到能处理该 URL 的 Driver 被找到
    Connection con = DriverManager.getConnection (url, "sa", "");
    Statement stmt = con.createStatement ( );
    ResultSet rs = stmt.executeQuery (query);
    ..... // 现实查询结果
    rs.close( );
    stmt.close( );
    con.close( );
 }
 catch (SQLException ex) {
     // 产生 SQLException，捕获例外并显示错误信息
         System.out.println ("\n*** SQLException caught ***\n");
```

```
        }
    }
}
```

ODBC二进制代码必须在每个使用该数据库驱动程序的客户端安装，所以这种驱动程序主要适用于公司内部网络，或者在三层结构中用Java编写应用服务器代码。

Microsoft的ODBC在Java中直接使用ODBC是不适当的，因为ODBC使用了C语言接口，从Java调用本地C代码在安全、实现、健壮性、可自动移动等方面存在许多缺陷，如Java没有指针，另外ODBC难于学习。ODBC在Java环境中使用时必须在每个客户机上进行安装配置，相反如果JDBC驱动程序用纯Java编写，可以具有Java的自动安装、可移动以及所有平台上Java的安全性。

4.6.2 运用纯Java JDBC驱动程序开发数据库应用

利用4型纯JDBC驱动程序可以实现两层C/S模式数据库访问，由于是纯Java语言编写的驱动程序，所以可以被动态下载，客户端可以实现零配置，如图4.8所示。

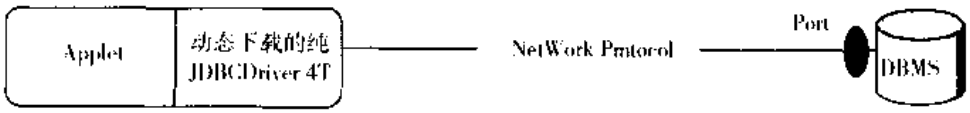

图4.8 利用4型JDBC驱动程序可以实现两层C/S模式数据库访问

下面是运用Java 4型JDBC驱动程序开发数据库应用的程序设计实例。

后台数据库采用MS SQL Server7.0，相应的JDBC Driver为MS SQL Server特定的驱动程序com.inet.tds.TdsDriver（可以在网上下载），Java版本1.1x，JDBC Version：1.22。

首先察看主机名和服务器的端口号，该驱动程序使用TCP/IP协议与Microsoft SQL Server进行通信，察看当前服务器监听的端口号：运行Microsoft SQL Server 7.0SQL Setup程序可以察看到默认的端口号是1433，用户可以进行更改。

如果运行telnet <hostname or ip address> <port number>后访问被拒绝，则主机名或端口号不正确。

驱动程序名：com.inet.tds.TdsDriver（可以在网上下载）

Url语法可以有以下几种格式：

jdbc:inetdae:hostname:portnumber→inetdae是驱动程序子协议名

jdbc:inetdae:hostname:portnumber?database=MyDb&language=deutsch

→带有properties信息

例如：jdbc:inetdae:199.34.57.35:1433

jdbc:inetdae:localHost:1433

有两种途径为驱动程序提供属性描述：

1）增加属性到URL，如

jdbc:inetdae:hostname:portnumber?database=MyDb&language=deutsch

2) 在 Driver Manager 的方法 getConnection(string url. Properties info) 中指定。
程序编制：

```
import java.sql.*;                                  // JDBC package
String url = "jdbc:inetdae:localhost:1433"; // 主机端口号可根据用户实际修改
String login = "sa";                                // 用户名可根据用户实际修改
String password = "";                               // 口令可根据用户实际修改
try{
    //DriverManager.setLogStream(System.out); // to create more info
                                    // for technical support
    //load the class with the driver
    Class.forName("com.inet.tds.TdsDriver");    // JDK.Netscape
    //or
    Class.forName("com.inet.tds.TdsDriver").newInstance();   // JDK.Netscape.JE
    //or
    new com.inet.tds.TdsDriver();  // JDK.Netscape.JE

    //set a timeout for login and query
    DriverManager.setLoginTimeout(10);
    //open a connection to the database
    Connection connection = DriverManager.getConnection(url,login,password);
    //select a database
    connection.setCatalog( "MyDatabase");
    //create a statement
    Statement st = connection.createStatement();

    //execute a query
    ResultSet rs = st.executeQuery("SELECT * FROM xxk1");

    //close the objects
    st.close();
    connection.close();

}catch(Exception e){
    e.printStackTrace();
}
```

程序运行时，首先将该程序编译放到www的可执行目录，解包驱动程序到该目录的子目录，建立HTML文件，嵌入该Java类，客户端使用WWW浏览器浏览相应的主页，即可

下载运行内嵌的Java Applet程序，进行数据库存取等操作。

该类驱动程序是针对特定数据库的二层客户/服务器模式的，使用该类驱动程序适合于后台数据库比较统一的特定环境，并且由于是二层客户服务器模式，程序开发成为以客户端为中心的，无法适应后台数据库的异构、多变，商业规则的变化，无法提供负载平衡、易于维护等好处。

第5章　Java RMI技术

5.1　分布式技术及从RPC到RMI

5.1.1　RPC发展及其不足

程序在网络中运行，使得分布式计算模式得到发展，这种结构允许用户共享网络中的计算机资源。这样，可使一个单机作为多个客户机的集中工数据库服务器；一些对CPU和内存要求很高的客户机不能很好地执行的操作，可以由网络中的服务器执行。在分布式计算环境中，RPC (Remote Procedure Call) 标准及其模型发挥了重要的作用。所谓RPC，即远程过程调用，它可以使客户机应用程序像本地API一样调用网络中某点的一个RPC服务。当服务工作完成，它将数据传回用户，调用返回。

传统程序设计语言中的“过程调用”概念是不难理解的，远程过程调用RPC是把过程调用的概念加以扩充后引入网络环境中的一程形式。RPC的形式和行为与传统的过程调用的形式和行为极其相似，主要的差别在于被调用的过程代码实际运行在与调用者结点不同的另一结点上，见图5.1。因此，需国设计相应在的软件来实现两者之间的连接的信息沟通。

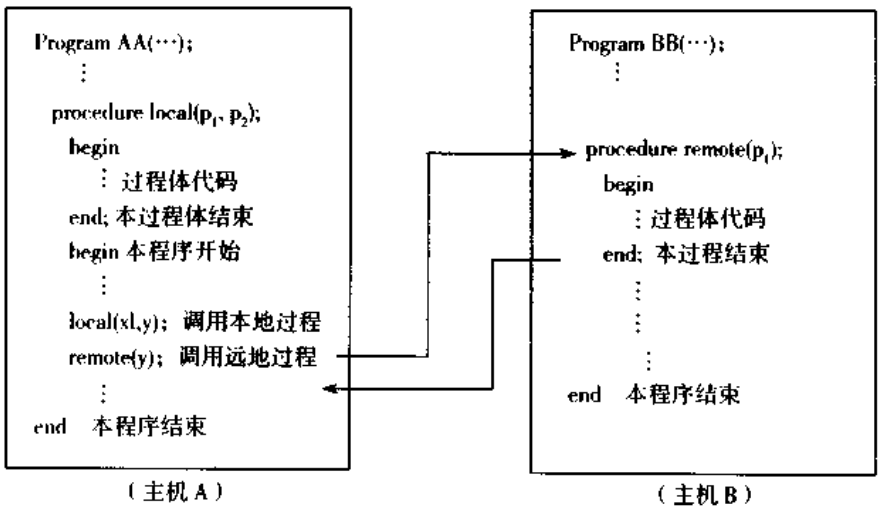

图5.1　主机A调用远地主机B上的过程

RPC机制的实质是实现网络七层协议中会话层的功能——在两个试图进行通信的结点之建立一条逻辑信道（即进行会话连接）并利用这个信道交换信息，不再使用时，负责释放所建立的连接。RPC的通信模型是基于client/server进程间相互通信模型的一种同步通信

形式，它对client提供了远程服务的过程抽象，其底层消息传递操作对client是透明的，在RPC中，client请求服务的调用者，server是执行client的请求调用的程序。

1) RPC实现技术

RPC的实现原理如图5.2。

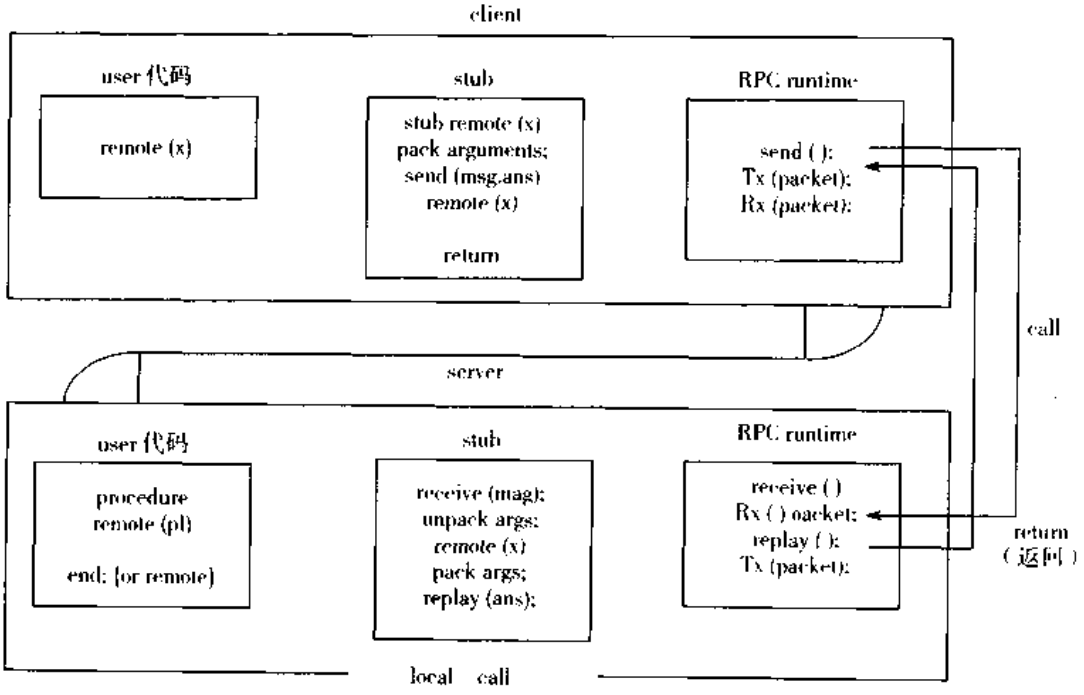

图5.2 RPC的实现原理

图中表明，每个远程过程由若干成分组成：调用者或用户、调用代码段以及被调用者、被调用代码段。这些都可用常规的程序设计语言编写，不需要利用特别的设施，就像它们在同一结点机上执行一样。另一些成分是与调用者相关的stub、与被调用者相关的stub及RPC runtime子程序，后者呆在系统中所有站点上运行。过程调用的主要工作环节如下：

- 调用者用通常方式调用对应C中的一个过程（也可以是其他语言过程）。
- 这个stub过程把有关的参数组装成一个消息包或一组消息包，以形成一条消息。运行此过程的那个站点的“地址”和那个站点上指称此过程的“标识符”都应包含在这条消息中；
- 将这消息发送给对应的RPC runtime子程序，该子程序再转交消息到指定的站点。
- 在接收此消息时，运程runtime子程序引用与被调用者对应的stub中的一个子程序，并让它来处理这条消息；拆卸有关的调用参数，并用通常的过程调用方式调用所需的过程。
- 返回调用结果，整个远程过程调用以与调用者对应stub程序执行“return”语句返回到用户而终止。
- 不难察觉，在上面叙述中，回避了一个重要的问题，即与用户对应的stub如何知道实际运行远程过程的结点之地址呢？例如，给定远程过程调用procX(pal;pa2)，与调用者对联应的stub如何确定procX将运行在哪个结点上呢？已经研究出了一些解决这一问题的

方法：

● 当系统生成与调用者对应stub时，可把该远程结点的地址也一同并入其中，不过这种做法不太灵活。

● 在进行调用之前，与调用者对应的stub向系统中的其他结点进行广播，请求有关的结点通报其地址，这必然引起一系列的消息传递。尤其是，当这种广播是在若干网络间进行时，其传递速度是很慢的。

● 由系统管理一个表，其表项的内容为：结点地址和该点上将运行的远程过程的名字。

● “愿意”产生一个可供其他结点引用的过程的那些站点添加一个表项到这个表中，该表项给出了这些结点的地址和此远程过程的名字。希望引用远程过程的用户有可通过查询此表获取有关信息。

RPC执行时，各部分的关系如图5.3所示，传输部分是RPC的最低层，其主要功能为：

● 提供对网络传输层协议的选择；

● 建立/释放逻辑信道，发送/接收消息等；

● 管理RPC中的消息缓冲区。

控制部分的主要功能是：

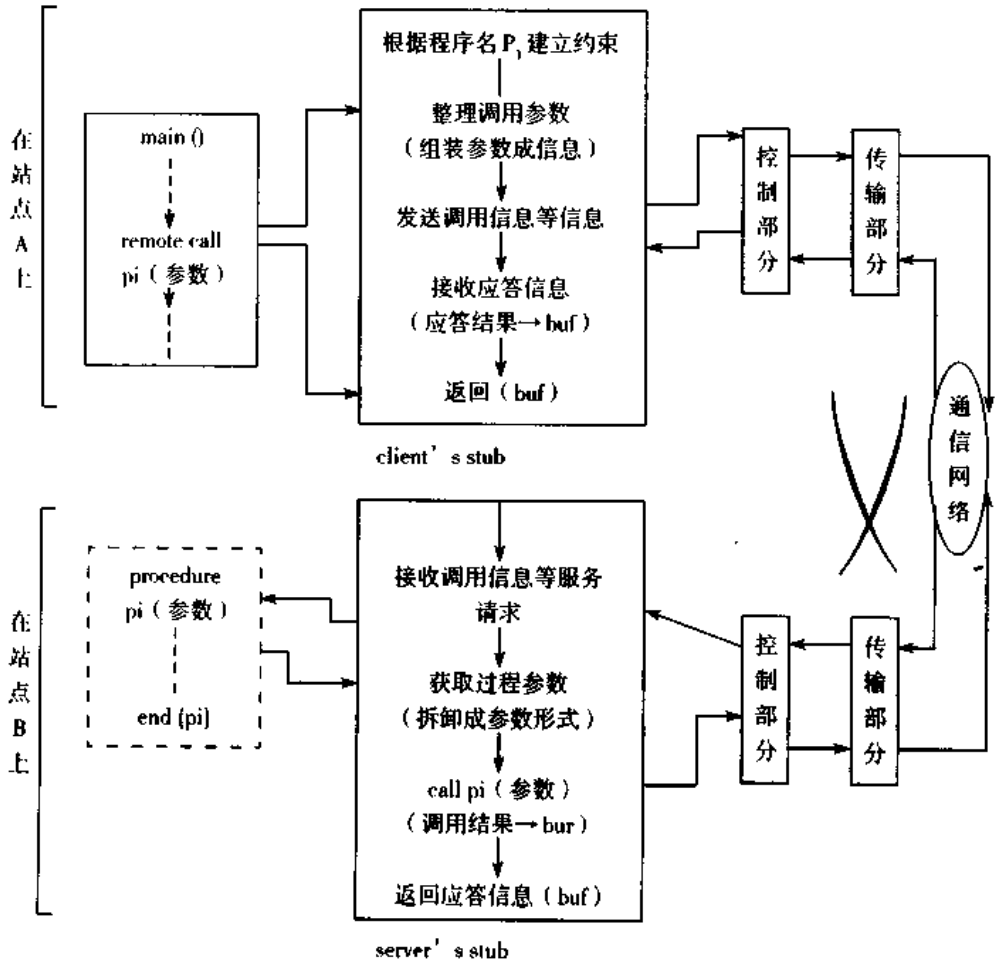

图5.3 RPC执行时各部分的关系图

● 确定RPC中消息的方向（发送或接收）。当client's styb开始线次RPC调用或者server向server's stub返回调用结果时，该部分负责控制传输部分进行传输。

● 站点间会合与进程同步。站点间会合是指为使两个站点间进程同步，它们必须同意“会合”（rendezvous），即早到达的进程要等待晚到达的进程，会合进程通过结点间会合建立一致的起点，并以该起点作为埋程同步点进行对话。

● 若干状态信息的处理。

由上可知，由于client's stub的作用，使得client可用常规过程调用方式去调用远程过程；由于server's stub的作用，使得server程序可能独立于调用者来编程，因而比较灵活。

2）RPC的语义

本地调用和远程调用之间存在许多不同之处。如果远程调用是在两种异型机器间进行，这就存在数据表示问题。例如，这两类机器的字长可能不同，解决这一问题的方法之一是它在传递数据之前，让RPC机制将有关的数据转换成一种统一格式，接收点在接收数据时，再把它们转换成本地所允许的数据格式。

另一问题是如何解释指针。在不具有共享地址空间的情况下，RPC不允许在网络范围内传递指针。因此，在RPC中是不可能用“reference方式”传递参数的。

更严重的问题是调用者的被调用者都可能在调用期间发生故障，而且经常是被调用者故障，致使调用者被挂起。如果发生这种情况，调用者可能不得不夭折，这在本地调用中是决不会出现的。

一个远程过程调用故障之后，调用者很难得知在故障发生之前，该过程调用已经进行到了哪一步。因此，系统必须提供一些基本的保护机制来确保RPC的正确效果。不过，由于通信方面也可能出错以及系统式图进行错误矫正而使这个问题更加复杂。不同的RPC实现方案定义的这种效果或RPC语义是有差别的，几种常用的定度RPC语义的规则是：

● last-of-many对执行一个远程过程调用而言，被调用的过程可能执行若干次，但规定其最后一次执行的结果作为返回结果；

● at-most-once。若调用者收到了回复消息，则称被调用的过程正确地完成了它的一次（仅仅一次）执行。如果调用者没收到回复消息，或者，如果调用者在获得回复消息之前发生故障，那么，这时的调用效果就看作是根本就没有执行相应的过程。

● at-least-once。在站点正常的情况下，远程过程至少执行一次，回复消息可能返回一次或多次。在站点故障时，就不能保证远程过程是否已被执行或曾返回任何回复消息。

● exactly-once。若server正常，则远程过程恰好执行一次，并返回一个调用结果。

RPC也有一些不同的形式。例如可以允许异步远程过程调用，因此，调用者和被调用者可以并行执行，调用者负责稍后某一时刻执行一个所谓的“回合”来获取调用结果。

3）RPC机制的不足

RPC在许多情况下是非常有用的，但随着面向对象编程技术在整个计算机领域的发展，RPC的许多弱点就显示出来了，比如RPC在处理网络节点对象时无能为力，若想在网络中传递特定对象是不可能的；RPC需要显式说明服务到指定节点，RPC运行时需要名服务（name servece）进程，以使客户机可以用它来找到给定接口的服务器所在的位置，增加了程序员的负担；在网络中显式传递指针使得无指针类型的语言不能使用这些服务；开发程序对客户与服务两端的平台及语言要求很严格；开发的程序扩展性（继承服务功能需

要重新编译整个程序）不好等。这都在很大程度上限制了RPC在更大范围内的使用。而RMI(Remote Method Invocation）正好弥补了这一不足。

5.1.2 分布式对象技术及RMI的诞生

分布式对象技术是分布式计算模式与面向对象编程的有机结合，作为一种应用集成技术，其目标是使得网络中任一结点上的应用，可以透明地访问存在于网络上任一结点上使用任何语言编写的远程对象，即可以像在本地地址空间一样对远程对象进行操作。这减轻了网络协议的多样性所带来的开发复杂度。为此，必须将其中的所有分布式处理封装于分布式对象底层基础设施中，而应用开发人员使用提供的接口可以透明地访问对象。

从系统开发者角度看，这种技术有很大的优点：因为只要把处理要求提交网络，便可取得其他对象的服务。对象在某些时候作为提供服务的服务器，在另外一些时候则作为接受服务的客户机。因此客户机同服务器的关系可在网上动态地形成，使得网络整体充分发挥其功能。

各个对象的实现在原则上是自由的。需要明确定义的，只是客户机对象同服务器对象之间通信用的界面规范。因此，尽管服务器对象因版本升级而变为高速、高功能的服务，客户机对象的开发者完全可以不必因此修改程序设计。对服务器对象开发者来说，也只需要遵从界面规格，而在内部如何实现同样可自由变化。这样，可以用对象作为基本单位来扩充系统，这是个重要的优点。因为对象生成过程中，曾经生成的功能可形成供以后再利用的父类，通过对象的继承方法，可以大大减少再次程序设计的工作量，同时也可以有效地提高已有服务的质量。

随着分布式对象技术的发展，SUN公司提出了基于Java语言RMI分布技术解决方案。Java RMI是Java的一种分布处理机制。允许不同虚拟机上不同地址空间的对象彼此调用其他对象的方法。这种机制实际上模拟了当前广泛应用于分布计算中的远程过程调用RPC，但它更具有面向对象特征且又限制在两个Java虚拟机之间。若某对象的方法被远程客户调用，则称为服务器。实际上，对象可具有客户与服务器的双重身份。前一个提供方法的服务器，当对另一机器上的对象调用远程方法时，本身又成为客户，甚至可以形成远程递归调用。

5.1.3 RMI的工作步骤

当客户机代码想对远程对象调用远程方法时，它实际调用了Java方法，该方法封装在一个称为存根(Stub)的代理对象(Surrogate Object)中，存根位于客户机上。存根取得远程方法中使用的参数，并将其打包为字节块。对每个参数的打包使用了独立于机器的编码机制。例如，数字总是按低位在前(big一endian)格式传送。字符串和对象就需采用点技巧：对对象进行编码，编码不能使用该对象的引用，因为对该对象的引用指向用户机上的内存地址，这些内存地址在服务器上毫无用处。以适当格式将参数编码，以便在网上传输，这一过程称为参数序列化（parameter marshalling）。

Java使用对象序列化机制来实现参数序列化。首先，客户机上的存根创建一个信息块，该信息块由下列部分组成：

- 调用的远程对象的标识符；
- 操作，描述被调用的方法；
- 序列化(marshal)的参数。

然后，向服务器发送该信息。在服务器端，有个构架（Skeleton）对象，它能理解包中的信息，并将该信息传递给执行远程方法的实现对象。特别是，对每个方法调用，构架执行下列五个操作：

- 反序列化（unmarshal）参数；
- 对服务器上的远程对象真正调用期望的方法；
- 捕获返回值或服务器上的异常；
- 序列化该返回值；
- 向客户机上的存根发送一个包，该包中含有序列化形成的值。

存根再反序列化从服务器上返回的值(或异常)，就得到了远程方法调用的返回值。如果远程方法抛出一个异常，存根就在调用者的进程空间内重新抛出该异常。

显然，该过程很复杂，但它在很大程度上，对编程者是透明的，并且Java远程对象结构的设计者尽力使远程对象和本地对象看起来一样。尽管如此，远程对象和本地对象间还是有重大区别的。

5.2 Java RMI 中的参数传递

5.2.1 远程对象参数的传输

当Java从服务器向客户传送远程对象时，客户接受到一个引用（object reference），使用该引用，客户就可远程调用服务器对象方法。然而，该服务器对象却保存在服务器上。远程调用除实现远程类接口外，还实现传送和返回任何对象参数。

每当需将远程对象从一个Java虚拟机传递到另一个Java虚拟机上时，Java虚拟机复制该对象的拷贝，然后在网络连接上发送该拷贝。这同本地方法中的参数传递相差很远。当向本地方法传入对象，或从方法中返回对象时，只传递参数的引用。然而，对象引用是其在本地Java虚拟机中的地址。该信息在不同的Java虚拟机中毫无意义。用Java序列化机制，Java可拷贝一般类的实例对象。然而，这是指Java不能拷贝下列类中的信息：该类的语法已显式限制拷贝或该类被Java语言本身禁止序列化。

本章最后的程序例是一个用户远程选购商品的简单程序。顾客在客户机上运行查询程序，服务程序收集接受查询顾客的信息，该例中只有年龄、性别和爱好三项。顾客类(Customer)对象参数被送给服务器，它向服务器上拷贝该对象。服务器程序返回一个产品向量。该向量中含有满足顾客简短描述的那些产品，而且总是含有该顾客喜欢的商品。因为向量类不是远程类，所以，向量被从服务器拷贝回到客户端。例中，所有在服务器端的向量被完整地拷贝回客户端。该例的向量元素较为复杂，向量元素即为远程产品类(Product)对象。因此，商品接受者获得向量的拷贝。

总之，远程对象参数在网络上以对象拷贝的形式被传输。所有这些都是自动进行的，不需程序员干预。每当调用远程方法时，存根就将所有参数值的拷贝打包，发送给服务器，

用对象序列化机制来将参数序列化。服务器再反序列化这些参数。由此可见，在参数很多时，该过程就非常慢了。

5.2.2 远程对象引用

从服务器向客户传递远程对象引用很简单。客户接受对象的引用，然后将该引用保存在一个对象变量中，该变量类型与远程接口相同。通过该变量，客户就可以访问服务器上的实际对象，客户可在自己的本地机器上拷贝该变量，所有这些拷贝只是对同一对象的引用。通过引用只能访问远程接口。远程接口是继承远程类的任何接口。所有未在远程接口中说明的方法，客户端都不能调用，这些方法都只在建立实际服务对象的远程Java虚拟机上运行。只有实现远程接口的类，才产生对象引用，对象引用提供接口中指定的方法。如果基类实现远程接口，派生类不实现远程接口，并且派生类对象被传送给远程方法，则只能访问基类的方法。

5.2.3 远程方法参数类型不正确

假如顾客远程选购产品时，要求看到产品的样图，则其中涉及到类是否可序列化的重要问题。程序中若只是简单的添加一个Paint(Graphics g)方法，会因为Paint参数不可序化而出现异常。实际上，图形类是个抽象类。有两种途径可获得图形类对象，一种是作为组件类的getGraphics方法的返回值，另一种是作为paint方法的参数。图形类属于在具体平台上实现了图形内容的基类。这些对象，需要与本地图形的实现代码交互作用。不同平台之间拷贝图形对象是毫无意义的。

5.2.4 自定义类与RMI的序列化机制

本章最后的程序例定义了一个Customer类，在客户端构造Customer对象，然后将该对象发送至服务端，调用方法后返回。可以发现该类继承了java.io.Seriolizable接口，这是一个可使对象序列化的接口。所谓对象的序列化，就是可以使对象保持固有特性在流中运动。在一般情况下，存储文件数据，传送网络参数使用Java一般类型就可以了，但若需要存储一个类对象，即若将类对象写入硬盘或是传递一个类对象至网络的另一台机器，在再次读出该对象或在另一机器看来也是有意义的一个对象的话，则不仅需要使得数据保留，而且要使得保留的数据特性反映出对象特征，即保留的数据是结构化的和面向对象的。

在一般的Java应用中，通常使用java.io包中的对象输入输出流类，来实现自定义类对象的存储。比如定义了一个学生类，其中包含姓名，性别，年龄等信息，可以这样定义：

```
public class Student{
private String name;
private boolean sex;
private int age;
}
```

若需要保存一个学生信息，可以先打开一个对象输出流类：

```
ObjectOutputStream out=new ObjectOutStream(new FileOutputStream("student.dat"));
```

然后，就可以向该对象输出流类中写入的对象了：

```
Student s=new student("lxb" ,true,22);
out.writeObject(s);
```

当以后需要再次读出该对象时，可以使用 readObject 方法：

```
ObjectInputStream in=new ObjectInputStream(new FileInputStream("student.dat"));
Student s=(Student)in.readObject();
```

读回对象时，必须仔细记录已保存的对象个数，顺序和类型。每调用一次 readObject 方法，就读入另一个类型为对象类的对象。然后将该对象转换为其正确类型。在这一过程中对象内部的具体属性值的保存不需要另外使用操作一般类型的办法，对象的操作已经包含了这些过程，可以看到，Java 在面向对象的思想方面做得非常出色。

RMI 最终操作是最底层的操作，也是在客户与服务之间打开一条输入输出流，在这一通道上，进行远程的操作。不过因为在这条流上不仅传输对象，也传输有关一般类型数据。RMI 对这些底层操作进行了封装，所以也不必像上面所示范的那样针对具体的对象进行操作，但 RMI 还是要保证存在某种机制使得 Java 能够正确地在本地与远程之间传递对象，为此，Java 提供了一个使对象序列化的接口 .io.Seriolizable，该接口使得 RMI 机制能够对定义对象进行序列化。RMI 使用中，可以不必理会网络中类对象究竟是如何传输的，但必须明确一点，它一定要继承 java.io.Seriolizable 接口，否则，自定义的类所构造的对象就无法在网络中传输。

5.2.5 Java RMI 的数据类型

网络程序跨平台尤其是异构平台计算的一个重要条件是参数的标准化，无论是在传统的程序间彼此调用或采取其他的分布式解决方案，均需将客户端与服务端参数格式映射为网络格式参数或将其中的一种参数类型格式转换为另一种参数表达格式。在网络进行分布程序设计时，因为客户端与服务端之间不可见，无法预先了解各端程序设计的工具及其参数格式，因而各自的参数格式无法直接转换，所以在进行远程端调用时往往是强制将参数格式映射为网络标准格式，这样做就使得在网络中各种平台上的各种语言均可以进行互操作，比如 CORBA，就是其中较典型的例子。

在 Java RMI 中，并不存在外部数据表达类型的问题，因为 Java 本身即严格规定了各类型参数的字节数及编码方式，只要运行在 Java 虚拟机上的所有 Java 程序都服从参数规定，在 Java RMI 中，网络传输的参数可是 Java 一般类型、使用 java.io.Serializable 系列化的类和实现某一远程接口的类，无论何种类型在 Java 中最终表示形式都是唯一的，在动态建立的客户与服务双方不会引起二义，这样 Java RMI 参数可以很方便地在网络传递。

5.2.6 Java RMI 的回收机制

Java 虚拟机运行的最大优点是与平台无关，其中引人注目的是垃圾回收功能，在程序运行中可动态地加载类并建立对象，同时与其他程序设计语言相比，较为显著的是建立的

对象无须显式地加以回收，Java虚拟机采取低优先级后台线程运行方式，自动在后台回收那些不再使用的对象，而且一旦程序终止，则后台线程将回收所有在程序中建立但并未回收的对象。

Java虚拟机运行时，垃圾回收机制在后台自动建立对象生存周期表，该表中包含对象标识符，引用次数等必要信息，只要有对象建立，则在表中添加该项对象记录信息，由程序中引用该对象来刷新该对象的生存期。

Java的自动垃圾回收机制在RMI的机制中有力地保证了Java网络编程的透明与简洁。RMI中的垃圾回收处理表也是建立在后台，不过是在远程提供具体方法服务的机器上，其刷新信息指令由Stub与Skel负责连接传递。一般说来，表中所建立的对象是对应具体客户端的，在多用户环境和多线程中，可能会有多个部分对同一个对象进行操作，在这种情况下，只需建立一个对象记录，在调用时只需要添加对象记录引用次数即可。

不过，回收机制只能有效地管理被引用对象的生存期，对于未被客户端引用的对象，则需要程序员显式地打破这些对象的生存期。一般情况下，出现这些的原因是建立了中间对象和临时对象，服务端的虚拟机可以用本地垃圾回收的方式加以管理。

总之，RMI的垃圾回收机制使得网络中传输的均为必要的数据或指令，有效地降低了网络负载，保证了对象传输和建立时不可预见异常的网络蔓延，达到了各网络节点自理。可以说，垃圾回收，有力地保证了Java RMI的分布健壮性。

5.2.7 Java RMI动态类装载与安全机制

当向另一Java程序传入远程对象时，无论是以参数方式，或是远程方法的返回值，这个Java程序都必须能够处理相关的存根对象，即它必须包括关于存根类的代码。存根方法所做工作只是序列化/反序列化参数，然后将该信息与服务器连接起来，当然，这些工作对程序员是透明的。另外还需装载关于参数、返回值和异常对象的类。这将比想象的要复杂得多。例如，远程方法可能被说明为某种返回类型，客户知道该类型，但实际返回的却是派生类对象，而客户却不知道这个派生类，那么，类装载器就必须装入该派生类。正因如此，运行的客户的程序必须能够获得存根类。达到此目的的明显办法是检查本地文件系统上是否有这些类。若没有，Java将从别处装入该类，其装入过程同在浏览器中的小应用程序相同。

对一个小应用程序，浏览器装入小应用程序并检查其字节码的有效性，从远程地点装入的小应用程序类的工作由类装载器完成。小应用程序类装载器限制非常严格，它只装入该小应用程序源主机上的类。并可设置存根类装载器，以装入更多的小应用程序。例如，允许它查找其他网络节点上的存根代码。这对分布式的Java程序非常有用。在这种程序中，多个处理器相互协作执行很复杂的运算，这些处理器都将从中心点获取同一个存根。

类装载器决定从何处装载类。安全管理器决定这些类能做什么，不能做什么。小应用程序安全管理器禁止小应用程序读/写本地文件，或创建与第三方的套接字连接。存根安全管理器比小应用程序安全管理器的限制更严格。因其只控制存根代码的特性，除存根必须执行的操作外，存根安全管理器将阻止其他一切操作。这是个安全机制，用以防止存

根代码中的病毒进入程序，对专用情况，Java程序员可采用自己的类装载器和安全管理器，但RMI系统提供的足以满足通常要求。

5.2.8 Java RMI的连接协议

RMI在传输层使用Java远程方法协议（JRMP,Java Remote Method Protocal），通常也称为RMI连接协议，用来在Java虚拟机之间传递方法调用、连接参数、返回值以及例外处理等，JRMP是由五条消息，外加五条用于多路复用流控制的消息所组成的简单协议。

所有JRMP的对话由一个标题和接下来的一条或几条消息所组成，标题中仅包含字符JRMI，协议的版本和子协议的ASCII代码。有三个子协议：SingleOpProtocol(简单操作协议),StreamProtocol（流协议）和MultiplexProtocol（多路协议）。SingleOpProtocol所描述的是在结束对话前（即连接关闭），一个标题只跟一条消息。StreamProtocol和MultiplexProtocol能传输一条或多条消息，后一协议是当客户机和服务器都通过一个socket进行多重调用时使用的，通信的客户机和服务器通常是各自打开一个socket给对方（即，双方系统都连接和监听连接状况），客户机的socket通常是调用服务器端的方法，而服务器的socket则调用客户机端的对象。

缺省的小应用程序安全限制拒绝小程序打开socket以响应任何服务器，除非是它们的源主机；此外，它还阻止任何监听socket的连接，解决这种情况就需要使用MultiplexProtocol多路复用协议及五条消息组。五条消息组是：Open, Close, CloseAck, Request, Transmit，它们允许客户机和服务器模拟StreamProtocol流协议使用一个socket进行双向通信，可以有多个虚拟连接被同时打开，每一个连接有唯一的ID表示。

5.2.9 Java RMI的分布进程

RMI技术本质上是一种分布对象技术，也就是在不同的网络节点上构造不同功能的对象，通过对这些对象的远程访问及对象方法的调用来完成客户所期望的目标。下面我们从进程的角度考虑RMI机制的有关调度问题。

在集中式系统中，对于进程的管理及调度已有许多较有效的方法能够保证进程的良好运行，比如关于进程的同步与互斥，死锁问题的解决等等，而在一个分布式计算系统中可以从应用程序总进程的角度来分析。因为分布式系统中各节点的进程运行部分依赖于各节点自身的操作系统及资源等，在局部可以认为它们的管理调度是集中的。

首先，可以参照已有的分布模型来具体描述一下RMI的运行及其特征，可以认为RMI是一种计算迁移与作业迁移皆具备的资源共享的远程调用技术。所谓计算迁移，就是客户端向服务端发出指令，通过在服务端调用预设的方法（过程）或建立新的方法（进程）来处理客户端的指令，对服务器本地的文件或其他数据进行操作返回结果的方法。在此过程中客户端的进程与服务端建立的进程可以并发执行。而作业迁移，则是一种完全的分散策略，即若作业到达某一网络节点时，可以将作业分散到网络上的不同节点上来执行。计算迁移的好处，是可以在需要多个网络节点上的局部数据时减轻网络负载，而作业迁移则平衡了网络负载，可以更好地发挥网络各节点的软件与硬件特长，并可以通过这种在不同现

场的执行达到进程并发从而加速计算。

在一个完全分布计算的环境中，解决最基本的进程间的通信，同步及互斥，防止死锁等问题是非常复杂的。RMI机制在这些方面做了大量的工作。

由RMIc编译Java接口类的实现代码生成了两个负责通信的类Stub和类Skel，其中Skel位于服务端，Stub位于客户端，由Stub负责客户端指令及数据的序列化，调用服务端（通过Skel）的方法，该过程建立在最基本的网络协议TCP/IP基础上，因而在底层由TCP/IP协议及其路由算法决定了网络数据及进程间的发送策略。对于RMI的高层来说，需要捕获可能由底层所引起的异常。但对于这些异常的处理却是自由的，即可以不限定程序员是采取尝试连接还是抛出异常立即退出等。

分布系统中要解决进程间的同步与互斥及防止死锁的问题发生，首先要解决进程在系统中的先后顺序问题，这样就可以根据发出申请资源的逻辑时间决定进程是否需要等待。在RMI机制中，由于其是建立在一种分散对象基础上的，每一局部点的对象在其向引导程序注册时已经建立，即该进程已经存在，但此时其具体的远程接口中的方法子进程并未建立，这些方法子进程是在收到Skel传来的调用指令时建立的。所以在这过程中，已经由这种逻辑上的关系规定了进程隐含的时间戳，即完成任务的总进程根据任务需求调度各分散对象的子进程，这样就可以保证了在任务总进程中的各个子进程的先后顺序，在这基础上就可以参照集中管理的方式管理各个子进程了。

在RMI中，管理进程的同步与互斥及其他资源访问问题可以用局部到某一具体的分散对象内部，即方法一级来考虑这些问题。一般情况下，对象进程即父进程是在对象建立与注册时已经建立，客户的访问只是单独地在逻辑上规定了其可以在访问等待队列中排队和排队的先后顺序，对方法的具体调用是可以不限制的，即等待队列中的客户访问进程可以同时对方法进行调用，因而，若资源的属性是非临界的，则这种并发执行是可靠且有效的。但是一旦这种访问具有互斥属性，则只能让访问进程按照次序顺序访问，RMI中实现这种访问的互斥限制是通过加上synchronized修饰符而达到的，若方法中存在这种限制的访问，则该方法的执行具有独占和事务特征。若一个对象中的多个方法存在这种同步限制，对象此时相当于一个监视器，负责管理这条访问临界资源并具有事务特征的等待队列。

分布系统中的另一个重要的问题是防止死锁发生。在一个完全分布的系统中，防止存在死锁主要是定义一个逻辑上的全序，使用资源定序技术来防止死锁发生。即对整个系统中的所有资源都赋予一个唯一的编号，仅当某进程未占有编号大于或等于i的资源时，它才可以申请编号为i的资源。分析RMI系统可以发现，分布程序执行逻辑上的先后关系和分散对象的访问机制使得死锁的可能性消失。首先分散对象的属性必须是公共的，即所有客户进程都可以访问，并且这种访问也不是独占的，因而在对象这一级上，对象这一资源是共享的，因此死锁的互斥条件无法成立；另外对象内部方法访问的限制使客户访问进程在执行完必要的功能后立即撤消因访问所建立的子进程，这样，在方法子进程上，访问进程也不可能做到持久拥有；再者，更广层面的逻辑上执行的先后顺序也使得访问对象是相对静止的访问，即客户在需要访问对象完成需要的功能之后，对该对象方法可能的需要同步的方法进程就予以撤消，因此在RMI的分布计算系统中不存在拥有局部点私有资源再去申请其他节点资源的情况。

5.3 RMI的工作步骤

5.3.1 设置

即使运行最简单的远程对象例程序，也需进行较多的设置。

(1) 有一个特定的查询机制，允许客户为定位服务器上的对象开始编写实际代码。

(2) 对象信息应分为客户端和服务器端实现。

(3) 必须在服务器和客户机上同时运行Java程序。

5.3.7节的例演示了如何满足上述要求。该例将在服务器上建立产品类(product)类型的两个对象。客户机将定位和查询这些对象。

5.3.2 接口

客户程序需要对服务器对象进行控制处理，但实际上却没有该对象的拷贝。对象本身位于服务器上。客户还必须知道能对该对象执行那些操作。即服务端可提供的服务必须对客户是可见的，这在设置RMI时也是一个重要的考虑方面。这种可见的能力通过接口来实现，客户与服务器共享该接口，所以，该接口必须同时存在于客户机与服务器上。远程对象的所有接口必须继承远程类(Remote)接口，远程类是在java.rmi包中定义的。在RMI中程序中所有必须使用远程对象的类都必须继承某远程类。这些接口中的所有方法还必须说明自己将抛出远程异常类(RemoteExceptio)异常。原因很简单，远程方法调用肯定比本地调用不可靠，远程调用随时都可能失败。服务器或网络连接可能暂时失败，或者也可能出现网络的硬件故障等，因此客户代码必须准备处理这些情况，否则，有可能导致客户端程序的崩溃。因此，Java强迫程序员每个远程方法调用在捕获必要的本地异常外，均必须捕获远程异常，并指明失败时就采取的适当操作，这一点也是网络编程与单机顺序编程的重要的不同之处。

远程方法必须说明为Public型，否则客户方在试图访问远程对象时将会发生错误，也就是接口方法对客户与服务双方的可见。另外，在方法说明中，方法中的参数与返回值必须是合法的RMI数据类型，合法的RMI数据类型包括Java一般类型，实现java.io.serializable的类(可使对象序列化的类)和实现某一远程接口的类。一般类型和可序列化类参数在RMI中是按值传递的，在目标地址空间中重新建立，远程对象是按引用传递的，客户方可使用从远程方法调用中得到的远程对象的引用对其进行远程访问。

5.3.3 RMI命名规则

使用RMI时，为保证代码具有可读性和良好的组织性，所有的类都应当采用统一的命名约定，以便容易识别每个类的用途(如表5.1所示)，将服务器的方法放入“...Impl”类的main方法中，可避免使用“...Server”类，在本节的最初举例中，将这两者分开，以便于仔细分析其中的关系。

表 5.1　远程类的标识约定

类的后缀名	该后缀名约定的含义	示　例
无后缀	远程接口	Product
Impl 后缀	实现接口的服务器类	ProductImpl
Server 后缀	创建服务器对象的服务器程序	ProductServer
Client 后缀	调用远程方法的客户程序	ProductClient
-Stub 后缀	由 rmic 程序自动生成的存根类	ProductImpl_Stub
_Skel 后缀	由 rmic 程序自动生成的构架类	ProductImpl_Skel

实际上，所有的服务器都必须继承java.rmi.server包中的远程服务器类(RemoteServer)。它是个抽象类，只定义服务器对象和其远程存根间的通信机制，RMI中的单模远程对象类(UnicastRemoteObject)继承远程对象服务器抽象类，是具体类，所以，不必编写代码就可使用该类。编写服务器类的最快途径是派生单模远程服务器类的子类。

单模远程对象位于服务器上。当用户申请服务时，该对象必须是活动的，并且通过TCP/IP协议是可获得的。这可能也是目前Java版本的限制，在未来的版本中，可能会有多模远程服务器类，用以表示在多个服务器上被复制的对象，还可建立按需激活对象类，可使用其他通信协议的类，如使用UDP协议的类。

在构造函数中，有一Super()调用的是UnicastRemoteObject类的缺省构造函数，它将该对象公布到分布式环境之中。由于Java不允许多继承，当一个远程对象必须继承另一父类时，就必须使用UnicastRemoteObject类的静态方法void exportObject(java.rmi.Remote obj)，显式地将该远程对象公布出去。

5.3.4　启动 RMI 自举注册表

客户要访问服务器上的远程对象，必须有获得远程引用的机制，来访问远程对象。客户代码可通过几种途径访问服务器对象。最常见的作法是，调用一个远程方法，其返回值是个服务器对象。当服务器对象作为方法结果返回给客户时，RMI机制自动送回一个远程引用，而不是实际的远程对象，RMI注册表提供一个简单的Server名字服务功能，它允许客户方从中得到某一远程对象的引用。当然，该注册功能可以返回所有在其注册服务对象的方法。由于Java安全性方面的限制，注册程序rmiregistry只能在服务方所在主机启动，不过每一Java虚拟机都提供了该注册程序。

启动注册程序的方法很简单，只需在服务所在主机上运行rmiregistry [port]，通常注册程序运行在1099端口，也可显式地将服务注册在指定端口，通常默认端口可以满足需要。若指定其他端口，则在程序中需要指定捆绑端口及寻找对象端口。

服务器用注册程序登记可提供的服务对象，而客户检索这些对象的存根。将服务器对象的应用传给注册程序并登记服务，同时给该对象取一个名字，这样就登记了服务器对象，该名字是个唯一的字符串。

进行登记时，要保持名字唯一是极其困难的，一般不用拼接名字的方法登记服务对

象。但在局域网中是可以协调的，在全球范围内，仅有很少几个命名的服务器对象带有引导服务。本例中使用名字拼接可以说明登记和定义对象的机制。

5.3.5 远程方法的基本步骤

调用远程方法的基本步骤如下：

(1) 编写继承远程类的接口并放在服务器和客户机上。

(2) 将继承远程对象类的实现放在服务器上。

(3) 创建实现类的（ProductImpl）的存根和构架。因为构架和存根是服务程序级和客户级的类，它们由RMI机制使用，以便序列化参数和序列化网上方法调用的结果。生成的存根和构架对程序员将是透明的。生成命令如下：

rmic ProductImpl

这就调用 rmic 工具产生两个类文件，其名分别为 ProductImpl_Skel.class（构架）和 ProductImpl_Stub.class(存根)。

(4) 登记注册，运行命令如下：

rmiregistry（可以将 rmiregistry 编入批处理 reg.bat 程序中，运行 reg.bat 即可注册。

(5) 在服务器上启动一个程序，该程序创建并登记实现类对象。

(6) 运行一个程序，查找服务器对象，并调用服务器上的远程方法。

详细步骤请参见 5.3.7 程序示例部分 。

5.3.6 编程的主要接口 APIS

接口是分布式对象处理中的一个重要的概念，客户的远程对象访问和服务方提供的远程对象服务都应遵循相同的接口，方可进行有效的互操作，即接口须对客户方与服务方均是可见的。所有的远程接口需要直接或间接地继承远程接口：java.rmi.Remote.Remote是一个空接口，没有说明任何方法，仅为所有的远程接口定义了共同的父类。远程接口中的任何方法，除了应用本身需处理的例外之外，都应说明抛出一个java.rmi.RemoteException例外，以处理RMI系统在远程调用过程中所发生的异常，如网络连接失败等。

static Remote lookup(String url)

返回 url 指向的远程对象。如果与该名字对象未连接，则抛出未连接(NotBound)异常。

static void bind(String name,Remote obj)

将 name 连接到远程对象 obj。如果该对象已连接，则抛出连接异常类（AlreadyBoundException）异常。

static void unbind(String name)

撤消 name 的连接，如果 name 当前没连接，则抛出未连接类异常。

5.3.7 程序举例

下面用例子介绍如何编写实际代码满足上述要求。在该例子中，将在服务器上建立两个产品类（Product）类型的对象. 客户机将运行定位和查询这些对象的 Java 程序。

该例可在单机或两台联网机器上运行。若想在远程机器上运行服务程序代码，则需在

客户机代码中设置服务器的URL，并重新编译，程序中已注明何处做修改。另外，还需将类文件分布到客户机和服务器上。服务器程序类、接口和构架类必须放在服务器上。客户程序、接口和存根类，必须放在客户机上。即使在单机上运行该代码，也必须有相应的网络服务，特别是要运行TCP/IP协议。若机器上没有网卡，可用Windows下的拨号网络特性建立起TCP/IP连接。

（该例中凡带有“——”的代码为命令，不带“——”的代码为程序代码。）

1) 定义接口

```
import java.rmi.*;
interface Product extends Remote
{
public String getDescription() throws RemoteException;
}
```

2) 实现接口

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductImpl extends UnicastRemoteObject implements Product
{
private String name;
public ProductImpl(String s) throws RemoteException
{
Super();
Name=s;
}
public String getDescription()
{
return “Hello!This is “+name+”!”;
}
}
```

该实现类有一个构造函数和一个方法，该方法即是在远程接口中定义的方法getDescription()。由于它继承了UnicastRemoteObject，这是个服务器类，UnicastRemoteObject是使远程对象可被远程访问的具体的Java类，在早先版本中，是用UnicastRemoteServer实现的。

3) 生成存根与构架/

——rmic ProductImpl

这就调用rmic工具产生两个类文件，其名分别为ProductImpl_Skel.class（构架）和ProductImpl_Stub.class(存根)

4) 登记注册，命令如下

rmiregistry

5) 创建远程对象

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductServer
{
public static void main(String args[])
{
System.setSecurityManager(new RMISecurityManager());
Try
{
ProductImpl p1=new ProductImpl("ColorTV");
ProductImpl p2=new ProductImpl("Rediao");
Naming.rebind("C" ,p1);
Naming.rebind("R" ,p2);
}
catch(Exception e)
{
System.out.println("Error is: "+e);
}
}
}
```

在RMI中，提供的java.rmi.Naming类让客户与服务双方使用注册表，它所有的方法均是静态的：void Naming.bind(String rmiurl,Remote obj))用于将一个Remote对象obj按照rmiurl登记到注册表中，void Naming.unbind(String rmiurl)用于取消某一远程对象的注册信息，Remote Naming.lookup(java.lang.String rmiurl)可得到某一远程对象rmiurl的远程引用，其中rmiurl的语法为"//<host>[:<port>]/<object>"表示运行于host端口port中的注册表中登记名为object的远程对象(port缺省为1099)。

服务方在创建远程对象之前需要安装一个安全管理器（SecurityManager），可以是RMISecurityManager，也可以是程序员自定义的SecurityManager，因为在RMI应用中类装载器需要装载一些对应用开发透明的类，如Skeleton及其使用的接口类，远程方法中的参数等，这些类文件可能要从网络获得，对RMI而言，安全性相对一般应用而言，限制更严格，无论是远程还是本地，不装载安全管理器，则会拒绝加载任何类。

6) 远程对象访问

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductClient
public static void main(String[] args)
{
System.setSecurityManager(new RMISecurityManager());
String rmiurl= " ";
```

```
Try
{
Product c1=(Product)Naming.lookup(rmiurl+" C");
Product c2=(Product)Naming.lookup(rmiurl+" R");
System.out.println(c1.getDescription());
System.out.println(c2.getDescription());
}
catch(Exception e)
{
System.out.println("Error is: "+e);
}
System.exit(0);
}
}
```

运行后显示：

```
Hello!This is ColorTV!
Hello!This is Radio!
```

这个输出虽然简单，但可以仔细考虑一下，Java都执行了那些操作。客户程序用lookup方法，获得对存根对象的引用。客户调用getDescription方法，该方法向服务器的构架对象传送一个网络消息，构架对象对服务器上的产品实现类对象调用getDescription方法。该方法计算一个字符串，该字符串被返回给构架，在网络上传送，被存根接受，从而作为结果返回。

5.4 与JDBC结合构架分布式数据库应用

5.4.1 编制JDBC程序的一般知识

有关JDBC的主要介绍请参见第4章。

首先要了解数据库URL。连接数据库时，必须指定数据源，还需指定附加参数。例如，网络协议驱动程序可能需要端口号，ODBC驱动程序可能需要各种特性，如用户名及口令等。

JDBC使用与普通网络URL相似的语法描述数据源。一般语法如下。

Jdbc：子协议名：其他参数部分

此处的子协议即是JDBC连接数据库的特定驱动程序，其他参数部分的格式取决于所用的子协议。对于ODBC源数据库，在使用JDBC-ODBC桥时可以使用如下这种格式：jdbc：odbc：<数据源名>；User="<用户名>"；PW="<口令>"

对于网络数据库，格式为

jdbc：db2：<主机名:[端口号]/数据源名>

编程首先需装载适当的驱动程序。例如，在使用JDBC-ODBC桥时，需要用

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")装入SUN的JDBC-ODBC驱动程序。

然后与数据库建立连接，该步骤是由驱动程序管理器类（DriverManager）负责装入数据库驱动程序，并建立新的数据库连接。实际编程时，不必太关心驱动程序管理细节。只需要用如下代码打开数据库连接：

Connection con=DriverManager.getConnection("jdbc:odbc:example")

JDBC管理器将使用加入的驱动程序进行连接数据库操作，如在上面不加入一个驱动程序，则JDBC管理器将遍历当前登记的所有驱动程序而实现该步操作。用getConnection方法可获得一个连接对象，该连接对象可使用JDBC驱动程序来管理SQL查询。用户可以执行查询和操作语句，移交或回送交互操作。

第三步创建一个语句类（Statement）对象。调用DriverManager.getConnection方法获得的Connection对象可以创建语句对象。

Statement stmt=con.createStatement()

第四步是向数据库发出SQL指令，执行对数据库的查询维护等。调用上面创建的语句类对象的方法，提供查询用的SQL命令串，即可执行查询。可使用同一个语句对象执行多个无关的查询。

ResultSet rs=stmt.executeQuery("SELECT * FROM Student")

对返回的记录集对象可以用多种方法对其进行操作，如遍历访问某一字段的值等。

最后需要关闭各类打开的连接，如stmt.close()和con.close()等。

5.4.2 运用Java RMI构架三层分布式数据库应用

在三层模型中，来自客户端的SQL指令被送到服务器端的中间层，然后再发送到数据库，数据库处理的结果从中间层再返回到客户端.

三层结构比较灵活并且易于维护，可以为用户提供一个容易使用的高级接口，由中间层负责将客户请求翻译成底层调用。在本文后面，将就一个具体的Java RMI与JDBC结合开发一个完整应用的例子。

5.5 RMI的应用举例

5.5.1 简单调用：单客户单服务

这是一种典型的Java RMI应用，在本例中客户端输入一小写字符串，调用远程方法转换为大写字符串，返回到客户机上。

(1) 接口为upper.java：

```
import java.rmi.*;
public interface upper extends Remote
{
String upper(String s) throws RemoteException;
}
```

该接口中规定了一个远程方法为upper(String s)，即转换小写字符串为大写字符串的方

法。

(2) 接口实现为 upperImpl.java:

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class upperImpl extends UnicastRemoteObject implements upper
{
private String name;
public upperImpl(String s) throws RemoteException {
super();
name = s;
}
public String upper(String s) throws RemoteException {
return s.toUpperCase();
}
public static void main(String args[])
{
// Create and install a security manager
System.setSecurityManager(new RMISecurityManager());
try {
upperImpl obj = new upperImpl("HelloServer");
Naming.rebind("lxb", obj);
// 该捆绑名字中可以包含 ip 和 port 信息。
System.out.println("lxb bound in registry");
} catch (Exception e) {
System.out.println("HelloImpl err: " + e.getMessage());
e.printStackTrace();
}
}
}
```

该接口实现具体实现了 upper(String s)的方法，同时在 main()方法中建立了服务对象“HelloServer”，并捆绑名字为” lxb”，若出现异常则抛出异常，程序终止。

(3) 客户端小应用程序：

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import borland.jbcl.layout.*;
import borland.jbcl.control.*;
import java.rmi.*;
```

```
public class upperclient extends Applet
{
XYLayout xYLayout1 = new XYLayout();
boolean isStandalone = false;
TextField textField1 = new TextField();
TextField textField2 = new TextField();
Button button1 = new Button();
public String getParameter(String key, String def)
{
return isStandalone ? System.getProperty(key, def) :
(getParameter(key) != null ? getParameter(key) : def);
}
//Construct the applet
public upperclient()
{
}
public void init()
{
try
{
jbInit();
}
catch (Exception e)
{
e.printStackTrace();
}
}
private void jbInit() throws Exception
{
xYLayout1.setWidth(400);
xYLayout1.setHeight(300);
button1.setLabel("Convert");
button1.addActionListener(new java.awt.event.ActionListener()
{
  public void actionPerformed(ActionEvent e)
{
  button1_actionPerformed(e);
}
});
```

```
this.setLayout(xYLayout1);
this.add(textField1, new XYConstraints(92, 72, 156, 25));
this.add(textField2, new XYConstraints(96, 139, 156, 25));
this.add(button1, new XYConstraints(131, 209, 87, 29));
}
public String getAppletInfo()
{
return "Applet Information";
}
public String[][] getParameterInfo()
{
return null;
}
void button1_actionPerformed(ActionEvent e)
{
upper obj=null;
try {
String s=this.getParameter("ip").trim();
obj = (upper)Naming.lookup("//202.119.199.118/lxb");
textField2.setText(obj.upper(textField1.getText()));
} catch (Exception ex) {
System.out.println("HelloApplet exception: " +
ex.getMessage());
ex.printStackTrace();
}
}
}
```

编译运行后，在客户端任意输入一小写字符串，则在第二个文本框中显示大写字符串。

5.5.2 分布方法: 单客户多服务

这个例子中，客户端界面有三个输入框，一个输出框，在三个输入框中填入整数值，击按钮，则在输出框中显示最终数值，前两个数的和与第三个数的积。但这个程序中并不是只将三个参数传给一个服务器处理返回值的，而是分别调用两个服务对象实现最终目的的。

(1) 该部分包含一个加法接口，一个乘法接口：

```
import java.rmi.*;
public interface multi extends Remote
```

```
{
int mul(int a,int b) throws RemoteException;
}
import java.rmi.*;
public interface plus extends Remote
{
int add(int a,int b) throws RemoteException;
}
```

(2) 加法接口实现部分：

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class plusImpl extends UnicastRemoteObject implements plus
{
    private String name;
    public plusImpl(String s) throws RemoteException
{
 super();
 name = s;
}
public int add(int a,int b) throws RemoteException
{
    System.out.println("Invoke the add");
    return (a+b); //how to convert a char to ascii function
}
}
```

乘法接口实现

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class multiImpl
extends UnicastRemoteObject
implements multi
{
private String name;
public multiImpl(String s) throws RemoteException {
super();
name = s;
}
public int mul(int a,int b) throws RemoteException
{
```

```
  System.out.println("Invoke the multiply");
  return a*b; //how to convert a char to ascii function
 }
 }
```

(3)（省略了服务对象的建立过程）第一个服务器建立了服务对象pluscxx，第二个服务器建立服务对象multicxx，分别建立了plus与multi的实现对象，在客户端的程序中可以在按钮事件加入如下代码：

```
void CK(MouseEvent e) {
int i1=integer.parseInt(textField1.getText());
int i2=integer.parseInt(textField2.getText());
int i3=integer.parseInt(textField3.getText());
try{
plus obj1=(plus)Naming.lookup("pluscxx");
System.out.println("Found plus object");
multi obj2=(multi)Naming.lookup("multicxx");
System.out.println("Found multi object");
int i4=obj1.add(i1,i2);
System.out.println("Add the first two number");
int i5=obj2.mul(i4,i3);
System.out.println("Multi with the last number");
textField4.setText(""+i5);
System.out.println("OK");
}
catch(Exception ex)
{
System.out.println("This error is :"+ex);
}
}
}
```

由客户端首先查找到两个提供服务的对象，当然完全可以只在初始化部分查找，这样使得代码较紧凑，同时也可以保证确保每一次调用方法对象都存在，然后调用第一个服务器的方法，求出前两数之和，然后调用第二个服务器的方法，求出该和与第三个数的积。注意在该调用过程中是由客户端负责调用对应服务器方法的。

我们在两个服务的程序中加入了必要的提示代码，这样运行客户端程序时可以了解具体调用隐蔽的一些细节。

从客户程序执行的控制台中可以看到，首先在控制台中打印已找到两个服务对象的信息，此时只是返回具体的对象引用，并不调用具体方法，因而在服务控制台中并没有调用信息，而一旦在服务端有被调用信息，则客户端已经打印计算信息，这一过程也是RMI普遍的调用执行过程。

例中的程序是在单机上实验的，若将对象定位的url改变，可以放到网络中其他节点的机器上。

客户程序的执行依然是顺序执行的，所有的对象调用都是对存储在本地机上的对象引用操作，客户程序中定义该变量为接口类型的。将其连接到实际对象中，对远程对象来说就是存根类。客户程序并不知道这些对象的实际类型。

5.5.3 递归方法调用：客户与服务的互调用

一般情况下C/S的构架是静态的，即服务端接受客户端发出的请求信息并处理返回请求结果调用值，客户端仅是一简单的输入输出前台。但可以想象的一条是，可能这样，就使得服务端由于数据的集中与业务处理的庞大而不堪重负，而所谓的客户端却由于崇尚“瘦”型，即零配置与业务规则的简单而浪费了许多宝贵的可计算资源。

现以一个简单的例子来证明网络中的客户与服务是一个相对与动态的概念。不存在绝对的服务也不存在绝对的客户，同时网络中服务与客户是可以相互转化的，例子很简单：客户端有二个文本框，第一个框是想要求其阶乘的整数，第二个框返回该整数的阶乘，当然，在本文所有样例中都忽略了数据有效型检验。不过这并不影响我们理解这些主要观念。

程序运行过程是很好理解的，客户端向服务端发出请求计算阶乘的整数，服务端经检验后，如是1，则直接返回值1，否则减去1，通知客户端以新的值继续调用服务端的方法，当然不是简单的通知服务端摇身一变成为客户，调用客户端（现在客户又变成了服务者）的方法，这样，进行不停的累乘，最后返回阶乘。

不过，可以只用简单调用的方法完成这一步骤，这样做又会增加网络负载，是否有这个必要，可以想象，若不存在一个超级计算机，这样做还是有好处的，至少，充分利用了网络中每个节点的计算能力。另外，可以肯定的是，若服务端是一个智能的并很严格的处理系统，它会在必要的情况下要求用户一次提交信息后输入确认信息(在用户输入不完整信息的情况下)，并具有简单地与客户会话的能力，这时，客户提供一些必要的方法以便服务端在需要时加以使用将是很方便的。

例如，要向一家商店电话购买彩电，刚开始时只想购买某家企业生产的彩电，于是说“我想购买长虹彩电”，但因为是不见面的电话购买，商店有很多长虹的彩电，有显示器大小不一的，有外观颜色不一的，自然商店不能一下确定你的需求目的，于是他会不断的向你提示询问“你需要什么颜色的”，“你需要多大的显示器”等等，你在他的提示之下逐渐表明了更多的具体信息，直到他认为你所有的条件都非常清楚和齐备了，并且这些也是他可以落实到到某一种具体的彩电上时，他才会接受你的购买请求。在这一过程中，客户不能只发出第一次请求信息就等待结果，他必须在服务端的引导之下逐渐提交更多的可以加以综合处理的信息，可以认为此时客户变成了某种意义上的服务。这一系列的步骤就是一个较完整的客户与服务互调用的过程。

(1) 服务端计算阶乘的接口：

```
import java.rmi.*;
public interface function extends Remote
{
```

```
int func(int x) throws RemoteException;
}
```

(2) 客户端查找对象的接口：

```
import java.rmi.*;
public interface lookserver extends Remote
{
int lookandfunc(String s,int x) throws RemoteException;
}
```

很显然，该调用接口含有两个参数，第一个参数为服务对象的标识，第二个参数为服务端回馈的二次参数值。

(3) 服务器端的接口实现：

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class functionImpl extends UnicastRemoteObject
implements function
{
private String name;
public functionImpl(String s) throws RemoteException {
super();
name = s;
}
public int func(int x) throws RemoteException
{
lookserver obj=null;
if (x==1)
return 1;
//若客户端请求值为1，则直接返回结果值1。否则执行else部分
else{
try
{
obj=(lookserver)Naming.lookup("//202.119.199.118/lxb");
//参数lxb部分是一个动态概念，其中包含客户IP信息
//在服务端回调时，作为服务端对象处理并返回一个客户端对象的引用。
}
catch(Exception e)
{
System.out.println("Error is: "+e);
}
System.out.println("This calls x is: "+x);
```

```
// 加入提示信息可以显示具体的回调参数值。
return x*(obj.lookandfunc("//202.119.199.118/cxx",x-1));
// 调用客户端查找对象的方法。
}
}
}
```

客户端实现部分代码：

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class lookserverImpl extends UnicastRemoteObject
implements lookserver
{
private String name;
public lookserverImpl(String s) throws RemoteException
{
super();
name = s;
}
public int lookandfunc(String name,int x) throws RemoteException
{
function obj=null;
try{
obj=(function)Naming.lookup(name);
// 提供的方法用以服务端调用客户端的启调方法。
}
catch(Exception e)
{
System.out.println("Error is: "+e);
}
System.out.println("This time calls x is: "+x);
return obj.func(x);
// 客户端调用服务端的接口方法
}
}
```

(4) 客户端程序负责第一次的查找服务器工作并向服务器发出请求信息，然后在不满足服务条件的情况下，被动地被服务器调用再次的请求信息，直到最终返回结果值。客户与服务端的建立对象代码省略，客户端程序也省略。

5.5.4 级联调用：服务端调用另一服务

级联调用是RMI又一重要的应用。假如客户向服务器发出请求后，服务器经检验后发现，自己不能处理其全部数据，那么服务器将调用能处理该数据的其他服务来共同完成任务。当然可能是服务器原先的设定就是将其中的一部分工作给其他服务器，这样达到了协同工作。如果客户端使用的是服务端几个公布方法中的一种，具体的参数不是很清楚，于是客户向服务端发出了一个不十分完备的信息，若服务端是严格的，它会像上述所讲的递归方法，调用客户端的确认信息，以保证服务端的处理确实能满足客户端的需要，当然服务器端完全可以将其中的一部分工作交给其他服务器处理。

在以下的程序中，客户端的界面与第二个例子的界面完全相同，但在具体的远程调用上有很大区别。三个数被同时传送给第一个服务器，由第一个服务器负责计算前两个数的和然后调用第二个服务器计算出该和与第三个数的积。这里省略了由第一个服务器调用客户端的确认信息方法，而只是由客户端负责传递参数的数目，即在第三个数为0第三个文本框不输入的情况下，向服务器发送两个参数，而若输入了第三个数，则向服务器发送三个参数，在这种情况下服务端将调用第二个服务方法来共同完成所请求的任务。

(1) 乘法接口与第二例中相同，而第一个服务接口中的定义变为：

```
import java.rmi.*;
public interface plusandmulti extends Remote
{
int addandmul(int a,int b) throws RemoteException;
int addandmul(int a,int b,int c) throws RemoteException;
}
```

(2) 该接口实现为：

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class plusandmultiImpl
extends UnicastRemoteObject
implements plusandmulti
{
private String name;
public plusandmultiImpl(String s) throws RemoteException {
super();
name = s;
}
public int addandmul(int a,int b) throws RemoteException
{
return a+b;
}
// 若只传送两个参数，则计算结果返回和。
```

```
public int addandmul(int a,int b,int c) throws RemoteException {
System.out.println("Invoke the addandmul");
int addtmp=a+b;
multi obj=null;
try{
obj=(multi)Naming.lookup("multicxx");
System.out.println("Invoke the multicxx");
// 在此调用第二个服务对象。
}
catch(Exception e)
{
System.out.println("Error is: "+e);
System.exit(1);
}
int last=obj.mul(addtmp,c);
return last;
// 返回最终结果。
}
}
```

(3) 分别建立一个multicxx和plusandmulticxx服务对象，客户端代码与服务对象建立代码省略。

5.5.5 分布数据库应用:与JDBC结合构架网络数据库

通过在JDBC中的论述，可以很容易地构架一个分布环境下的数据库应用模型。

针对客户端远程调用对数据库操作的远程方法，具体实现途径可以采取多种策略。较好区别的两种手段分别是“治标”的或“治本”的。所谓“治标”，是采取已有的JDBC的数据库操作方法，将SQL语句的方法连接到该服务端的实现上，客户端发送SQL语句到服务进程，由服务进程进行查询，将结果包装为可以传递的远程对象返回给客户；而“治本”是指将JDBC API由本地方法重新定义为远程接口，然后给出所定义的JDBC API接口的所有实现，使之继承远程类，连接该类到服务程序指定的端口，客户端就可以仿照编写本地JDBC程序一样调用远程类的方法。处理数据查询的指令大多在客户端，这样的实现可以更灵活，相对也更简单（程序员可以不必要考虑数据的网络传输格式）。下面仅就第一种方式做一个简单的实例。

(1) 定义一个远程接口：

```
import java.rmi.*;
public interface request extends Remote
{
String getresult(String sql) throws RemoteException;
}
```

在该接口中，将返回的客户可操作的数据确定为String类对象，这其实是在服务端将记录集对象经过转换成网络可以传输的格式，因为记录集是一逻辑视图，不可以被序列化，若想在客户端直接使用记录集对象，则就需要重新定义所有的JDBC API接口为远程方法接口，其中所有的方法都需要重新定义。

(2) 实现上述远程接口的程序：

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.sql.*;
import java.net.*;
public class requestImpl extends UnicastRemoteObject implements request
{
private String name;
private String queryresults;
public requestImpl(String s) throws RemoteException
{
super();
name=s;
queryresults="";
}
public String getresult(String sql) throws RemoteException
{
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
}
catch(Exception ex)
{
System.out.println("Can't find Database driver class :"+ex);
return(null);
}
try{
String s="";
Connection con=DriverManager.getConnection("jdbc:odbc:example");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery(sql);
int strlen=0;
while(rs.next())
{
String sb="";
int i=rs.getInt("number");
```

```
String s1=rs.getString("name");
String s2=rs.getString("nation");
boolean b=rs.getBoolean("sex");
if (b==true)
sb=" 男 ";
else
sb=" 女 ";
Date d=rs.getDate("year");
String s3=rs.getString("address");
String s4=rs.getString("phone");
String text=i+" "+s1+" "+s2+" "+sb+" "+d+" "+s3+" "+s4;
s=s+text+"|";
}
queryresults=s;
stmt.close();
con.close();
}
catch(SQLException ex)
{
System.out.println("SQLException: "+ex);
}
return(queryresults);
}
}
```

(3) 服务程序：

```
import java.rmi.*;
public class server
{
public static void main(String args[])
{
System.setSecurityManager(new RMISecurityManager());
try
{
requestImpl obj=new requestImpl("server");
Naming.rebind("lxb",obj);
System.out.println("lxb created and bound");
}
catch(Exception e)
{
```

```
System.out.println("Error is: "+e);
}
}
}
```

(4) 客户端进行查询的小应用程序：

```
import java.awt.*;
import java.rmi.*;
public class client extends java.applet.Applet
{
String rs="";
public void init()
{
try
{
request obj=(request)Naming.lookup("//202.119.199.118/lxb");
rs=obj.getresult("select * from student");
}
catch(Exception e)
{
System.out.println("Error is: "+e);
}
}
public void paint(Graphics g)
{
if(rs.equals(""))
{
g.drawString(" 无记录 ",5,50);
return;
}
g.drawString(" 学号 姓名 民族 性别 出生日期 籍贯 电话号码 ",5,10);
int y=30;
int begin=0;
int end=rs.indexOf("|");
do
{
String line=rs.substring(begin,end);
g.drawString(line,5,y);
begin=end+1;
end=rs.indexOf("|",begin);
```

```
y=y+15:
}
while(begin!=rs.length());
}
}
```

客户端也可以使用一般的 Java 应用程序，则不需要 WEB 浏览器来实现数据库查询，这样使得客户端的应用更加自由、灵活同时适应性也更强。由于Java的跨平台，可以将此程序无修改地移植到其他操作平台上，客户端也不必关心后台的环境结构，利用浏览器可以透明地访问后台数据库。

5.6 用 RMI 技术实现简单的远程产品定购系统

5.6.1 产品定购系统的设计说明

本系统利用 Java RMI模拟实现了一个简单的网上购物模型。整个系统中分成商场、客户、银行三个部分，它们之间的通信关系如图 5.4 和图 5.5 所示。

通常情况下，一个完整的网上购物系统应包括以下几个部分：

- 企业信息发布系统；
- 商品采购系统；
- 电子认证系统；
- 财务管理系统。

当然，具有这样几个系统的网上购物系统还只是顾客－商家模式。本系统中，首先顾客到商家查询其所想购买的产品，这时，商家负责向顾客询问其想购买产品的品牌，通过询问，商家可以最终统计出最受欢迎的产品品牌。接着，商家向顾客发出所有的产品信息，在本演示程序中，商家向顾客发送产品的单价、规格、库存量。

图5.4中，顾客查询完成之后，可以首先完成订货单的填写，然后顾客再将订货单发送到商家。商家接受到订单之后，如果商家此时的库存量大于顾客的定购量，商家将调用银行的认证系统，同时商家传入自己的账号和交易额以待银行认证其身份并结算，然后，

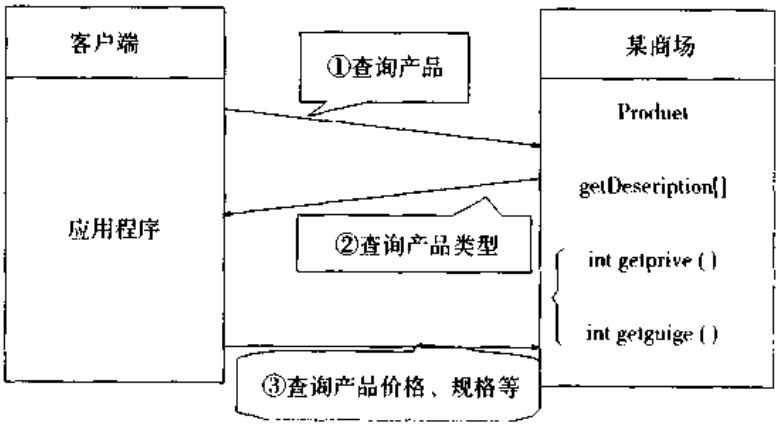

图 5.4 客户查询产品时与商场的递归调用

银行调用顾客端的setmima（）来获取顾客的数字签名。

图5.5中，银行验证身份之后，若双方身份均合法，银行将到结算中心调用zhuanzhang

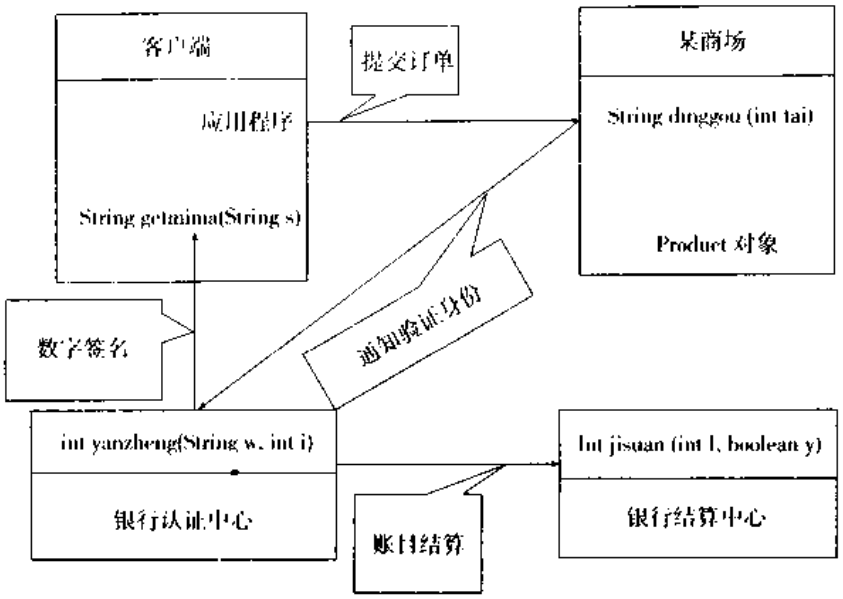

图 5.5　客户定购产品时与银行、商场三者之间的级联调用

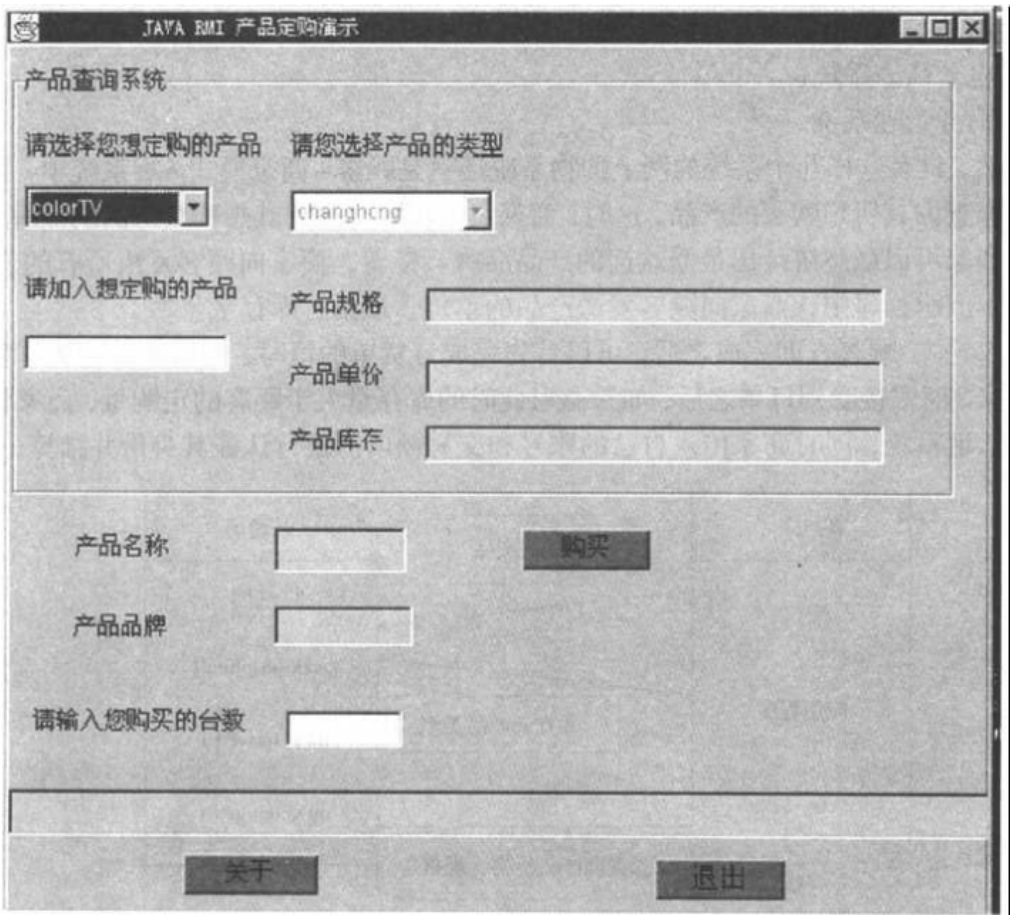

图 5.6　客户运行界面

()来进行顾客和商家之间的转账。银行转完账之后，向商家发送转账信息，商家再向顾客发送送货通知。最后商家将货物送到顾客手中。至此，整个购物过程就结束了。

当然本演示程序真正的目的并不完全是讨论有关电子商务的实现技术细节，而是通过示例来体现Java RMI的优势。在实现整个购物演示的过程中，可以看出顾客、商家和银行三者的地位是平等的。顾客并不是简简单单的将所需产品的信息发送出去就完事了，他在查询和购买的过程中必须与商家和银行进行交互。也就是说顾客的信息并不是一次性的全部发送出给商家。这样顾客所购买的东西以及购买的数量等隐私只有商家知道，而顾客的数字签名是银行系统进行认证的，这就使得商家无法获取认证的有关信息，保证了系统的安全和隐私保护。

首先客户在传送信息之前必须进行加密，通过密钥处理之后再在网上传送，当然在本演示程序中只是简单的按照顾客和银行的约定，首先顾客将传送的信息按照字母顺序向前或向后移动几位，打包送出去，银行接到包之后再按照顾客的约定逆向解包。通过密钥处理，一方面是安全的需要，另一方面使得顾客一旦发送信息后就无法抵赖。

客户运行界面如图5.6所示。

5.6.2 系统的实现有关细节

产品定购系统主要是采用的Java RMI技术，但是在整个实现的过程中，也用到了Java的许多其他技术。

首先，在最初设计的时候，整个流程是处在同一个进程之中，这样就产生了一个问题：顾客这一端调用商家的远程对象的方法时，进程正在处理远程对象的方法，这时顾客这一端的界面就无法得到刷新。为了解决这一问题，可利用Java支持多线程的特性，将调用商家远程方法的部分做成线程，在进程中调用该线程，每隔5 ms将该线程休眠，使得进程有机会来刷新屏幕。

其次，Java语言面向对象的机制的确给编程带来方便。在商场服务端，需要定义一个产品类，它用来描述产品的有关信息。商家必须能够在某一类产品中增加不同的品牌，而顾客又可以通过将某一类产品作为远程对象调用它的getprice()、getname()、getguige()和getkucunliang()获得有关信息。因为Java语言本身就是纯面向对象的语言，每一个完整的程序Java都将它看成一个类，可以定义一个该类的对象，调用其中的公共方法。在系统中，为了描述产品信息和顾客信息，定义了两个类：Product类和Client类。其中，Product类是一个远程类，也就是说它定义的对象可以被远程调用。

第三，Java语言中捕捉异常的机制也给编程带来了方便。在Java的网络编程中出现异常或错误是经常发生的，而Java提供了显式捕捉异常的机制，即凡是在有可能出现异常的地方，Java都要求程序员捕捉异常，这些异常的捕捉以及以及异常信息的提供通常是Java语言本身提供的。也可以重载异常捕捉，来定义自己的异常处理机制。比方说在编程的过程中，有好多地方出现异常的时候，给出了自定义的异常信息，这使得程序的可读性更强，对用户来讲也显得更加友好。

Java语言本身还有很多的优点，如与平台无关性、动态性等等，但是这些所有优点都是以牺牲速度为代价的。随着计算机硬件速度的提升，Java速度的问题将有望得到解决。

5.6.3 实现本系统的意义

作为一项技术，Java RMI的优势并不体现在电子商务系统中顾客－商家这一块，这一块用ASP和Java applet可以将顾客这一端做得相当完善。在电子商务系统中，完全是可以实现顾客端零配置，即只要顾客有一台连上Internet的计算机，并装有浏览器即可。Java RMI技术的优势体现在服务上，也就是说在电子商务系统中商家和银行提供的服务用Java RMI技术来完成是很有意义的。使用RMI技术的两台或多台计算机之间的地位是平等的，不必要将它们分为客户或服务器。因为RMI技术不同于传统的C/S模式，当某一方调用另一方的对象时，调用者是客户端，被调用者是服务器。但这种调用和被调用的关系是处在动态的变化之中的，它是一种典型的点对点的模式。

所演示的RMI的几种调用形式就能说明RMI的优点。不同的服务者之间可以通过不断的相互调用，相互通信最终连为一个整体来为客户提供强大而可靠的服务，这是RMI真正的魅力所在

5.7 关于Java RMI的未来

RMI将Java特性扩展到分布式系统的领域中，由于Java本地模型的易用性，RMI成为一种较简单和快捷的实现分布式对象结构的方式。它的优点是多方面的：可以在服务器与客户端之间来回传递对象数据；支持分布式的垃圾回收机制，使网络节点具有自理能力，减少网络负载；保留并加强了Java的良好安全性；充分利用Java多线程的机制以增强并发处理的要求；可以使用对象序列化机制跨越系统直接发送对象。这些优点使得Java RMI的分布式应用具有良好的平台中立性和健壮性。与其他分布式对象技术如CORBA相比，在伸缩性方面还不够，它只将RMI的应用限制在Java分布式对象中，而不能和其他非Java分布式对象通信。不过最新的技术进展表明，SUN正努力在CORBA/IIOP的顶层实现RMI的小套件，使Java程序通过RMI访问CORBA对象。相信Java RMI将会在未来的跨平台分布解决方案中发挥巨大的作用。

第 6 章　Java 与 CORBA 技术

6.1　CORBA 简 介

OMG（Object Management Group，对象管理组织）是一个致力于开发分布式对象标准的工业协会。CORBA(Common Object Request Broker Architecture，公共对象请求代理结构)是 OMG 制定的一个基于开放标准的分布式计算解决方案，它的主要目的是透明地穿过硬件、程序语言和操作系统开发健壮的、可伸缩的、面向对象的分布式应用。

6.1.1　CORBA 的主要内容

OMA（Object Management Architecture，对象管理体系结构)参考模型(如图 6.1 所示)从总体上抽象地描述了 OMG 组织推出的面向对象技术所包含的内容、以及模型中各组成部分之间的关系。CORBA 规范定义了 IDL 语言（Interface Definition Language,接口定义语言）及其映射、单个 ORB 和 ORB 间互操作机制。

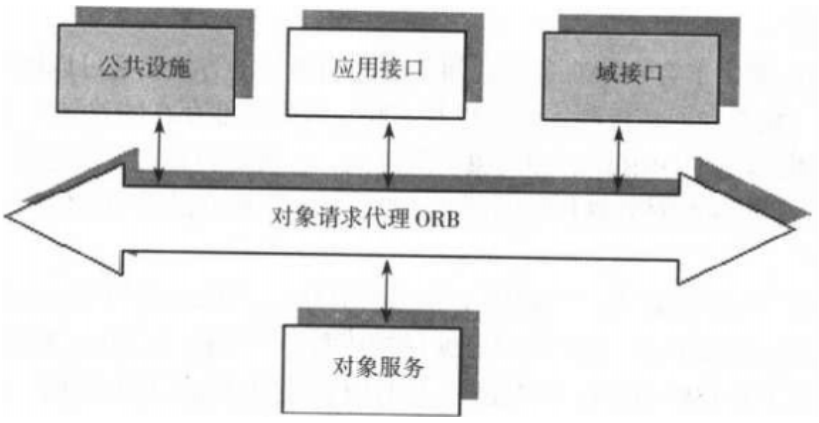

图 6.1　OMA 参考模型

对象请求代理ORB(Object Request Broker)负责对象在分布环境中透明地收发请求和响应，它是构建分布对象应用、在异构或同构环境下实现应用间互操作的基础。

对象服务（Object Services）是为使用和实现对象而提供的基本服务集合。针对对象服务，OMG 组织制订了 CORBA 服务（CORBAServices:Common Object Services Specification）规范，简称 COSS 规范。COSS 规范由一组接口（Interface）和服务行为描述构成，其接口一般使用 OMG IDL 语言描述。

公共设施（Common Facilities）是向终端用户应用提供的一组共享服务接口，但这组服务不像对象服务那样基本。

应用接口（Application Interfaces）是由销售商提供的、可控制其接口的产品。应用接口处于参考模型的最高层。

域接口（Domain Interfaces）是为应用领域服务而提供的接口。

CORBA 规范是针对 OMA 参考模型中的对象请求代理 ORB 而制订的。CORBA 规范定义了 IDL 语言（Interface Definition Language，接口定义语言）及映射、单个 ORB 和 ORB 间互操作机制。

单个 ORB 的体系结构如图 6.2 所示：

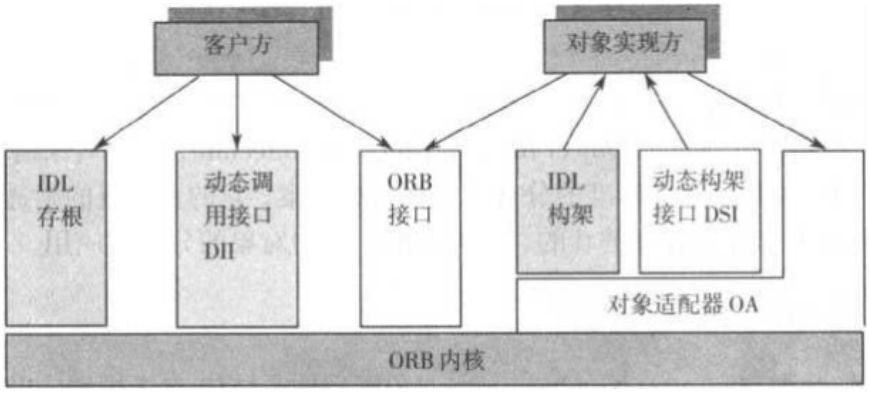

图 6.2　单个 ORB 体系结构

在该体系结构中，描述了以下主要的内容：

(1) IDL语言通过说明对象的接口来定义对象。一个接口包括一组命名的操作和相应于这些操作的参数。

(2) ORB 核心提供了客户—对象实现间实现透明通信的方法，它可以屏蔽对象实现位置、实现方法、状态和通信机制等细节以及不同实现间可能存在的差异。

(3) 对象适配器位于 ORB 核心和对象实现之间，它负责服务对象的注册、对象引用的创建和解释、对象实现的服务进程的激活和去活、对象实现的激活和去活以及客户请求的分发。

(4) IDL存根（IDL Stub）为客户提供了静态调用方式，IDL构架（IDL Skeleton Interface）为客户提供了静态实现方式。IDL编译器编译描述服务对象接口的IDL文件，生成对应于具体编程语言的IDL存根和IDL构架程序。IDL构架对用户请求进行解码，定位所请求的对象的方法，执行该方法，并把执行结果或异常信息编码后发送回客户。

(5) 动态调用接口 DII（Dynamic Invocation Interface）和动态构架接口 DSI（Dynamic Skeleton Interface）提供了动态调用方式和动态实现方式。

6.1.2　CORBA 的技术特色

CORBA 规范充分利用了现今各种技术发展的最新成果，将面向对象的概念揉合到分布计算中，定义了一组与实现无关的接口方式，引入代理机制分离客户和服务器，使得 CORBA 规范成为开放的、基于客户/服务器模式的、面向对象的分布计算的工业标准。

CORBA 规范的技术特点是：

(1) 引入了代理（Broker）的概念。一个代理至少可以有三个方面的作用：完成对客户

方提出的抽象服务请求的映射；自动发现和寻找服务器；自动设定路由，实现到服务器方的执行。代理根据客户方的请求，选择一个或若干个服务器进行处理，使客户方无需考虑服务器方的处理细节，只需最后把服务器处理的结果作为对客户方请求的响应通知客户。客户方和服务器方相互独立无关，使得软件易于修改、移植和维护。

(2) 所实现的客户方程序与服务器方程序的完全分离。客户与服务器之间可以有更加灵活的关系存在。在调用方式保持不变的前提下，服务器可以自由地修改和升级，客户方程序也可以自由地按照要求进行更改，而无需通知对方。

(3) 将分布计算同面向对象的概念相互结合，可以提高软件重用率、控制冗余度等。

(4) 提供软件总线的机制。所谓软件总线是指CORBA系统定义了一组接口规范，任何应用程序、软件系统或工具只要具有与该接口规范相符合的接口定义，就能方便地集成到CORBA系统中，这个接口规范独立于任何实现语言和环境。

(5) 分层的设计原则和实现方式。CORBA规范仅定义了ORB中需要用到的基本对象，封装了相应的属性和方法。而面向应用的对象定义则可以在OMA的应用对象、领域对象或应用开发环境中逐步分层定义和实现，使CORBA系统核心始终是一个精炼的实体，而各种复杂功能和应用可以由核心扩展和延伸。

6.1.3 CORBA产品一览

自从CORBA规范颁布后，许多公司推出了遵从CORBA规范的产品，产生了一定的影响。主要的产品有：

- IONA公司的Orbix
- Inprise公司的VisiBroker
- Digital公司的ObjectBroker
- IBM公司的Component Broker
- Sun Microsystems公司的NEO和JOE
- SunSoft公司的DOE

6.2 ORB系统组成及其运行原理

6.2.1 ORB系统组成结构

在CORBA环境中，一个应用可以由许多对象组成，通过对象间的交互来实现应用的功能，而对象间的交互则是通过ORB来传递。对一个交互过程而言，有客户和服务器之分。ORB系统由IDL存根、动态调用接口（Dynamic Invocation Interface）、ORB核心（ORB Core）、ORB接口（ORB Interface）、IDL构架（IDL Skeleton Interface）、动态构架接口（Dynamic Skeleton Interface）和对象适配器（Object Adaptor）组成。对象实现是通过由目标对象的接口描述自动生成的IDL构架或者动态构架接口的“回调”（Call-back）来接收客户的调用请求。对象接口的定义可以通过以下两种方式实现：

(1) 使用接口描述语言对对象进行静态的定义和描述；

(2) 将对象接口加入到接口库（Interface Repository）中。

尽管这两种定义和描述的方式不同，它们的描述能力是相同的。下面介绍一下ORB系统中的组成成员以及相关的概念。

1) 对象引用(Object Reference)

对象引用是在一ORB范围内用以指定和标识一个目标服务对象的信息。对象引用本身的形式与特定的编程语言和ORB系统的实现相关。尽管不同的ORB实现系统中的对象引用的表示形式不相同，但是它们在相同的编程语言环境中提供给应用（包括客户和对象实现）的使用形式是相同的。这样，就能使得以某种编程语言编制的程序在访问对象引用时与特定的ORB实现（产品）无关。

2) 客户

客户通过访问目标（服务）对象的对象引用来调用并执行目标（服务）对象的操作。客户仅仅了解目标对象的逻辑结构——目标对象的接口及其参数。

3) 对象实现

对象实现通过对象实例化数据的说明和赋值以及对象方法的执行代码来提供对象的语义。在通常情况下，对象实现会使用其他对象或者其他软件来实现对象的行为。

4) ORB核心和ORB接口

ORB核心提供了对象的通用表示和对象间通信的机制，它实现了客户/对象实现间通信的透明性，屏蔽了对象实现的位置、通信机制。根据CORBA的设计思想，ORB核心的功能和机制是非常简单的。这样就可以为客户调用请求和对象实现响应的传输带来较高的效率，而且ORB核心的简单化使得它的实现能够尽可能地可靠。

ORB系统的一些基本功能通过ORB接口的形式为客户方和对象实现方所见，这些接口在所有的ORB实现系统中都是相同的。这些接口一部分表现为ORB对象的操作，另一部分表现为对象引用对象（即Obiect对象类）的操作。但是，无论这些接口和功能表现为哪种对象的操作，它们都由ORB核心实现。

这些接口从功能上划分，可以分为以下几个类别：

(1) 对象引用的操作。

该组接口提供对象引用的串化和反串化功能，以便对象引用的保存和传递。同时还规定了对象引用的复制、删除、比较及检查对象引用存在与否的操作。

CORBA规范定义了对象引用（字符）串化和（字符）反串化的接口。它们表现为ORB对象的操作，具体形式（采用伪IDL语言）如下：

```
module{//                          伪IDL语言
    interface ORB{
        String object_to_string(in Object obj);
        Object string_to_object(in String str);
        Status create_list(in long count,NVList new_list);
        };
    };
```

对于对象引用，还有另外一些基本操作。这些操作为Object对象操作，也就是对象引用本身的操作，它们的具体形式如下：

```
module CORBA{
        interface Object{
            ImplementationDef get_implementation();
            InterfaceDef get_interface();
            boolean is_null();
            Object duplicate();
            void release();
            boolean is_a(in string logical_type_id);
            boolean non_existent();
            boolean is_equivalent(in Object other_object);
            unsigned long hash(in unsigned long maximum);
            Status create_request(in Context ctx,in Identifier operation,in NVList
arg_list,inout NamedValue result,out Request request,in Flags req_flags);
                };
        };
```

(2) ORB 和对象适配器的初始化。

要使一个应用能在CORBA环境中运行，当它进入CORBA环境时，首先就需要通过一种方式来获得ORB和对象适配器的伪对象引用。ORB的初始化操作并不是ORB对象的操作，而是直接位于CORBA 模块（Module）中，其形式如下：

```
module CORBA{
    typedef string ORBid;
    typedef sequence <string> arg_list;
    ORB ORB_init(inout arg_list argv,in ORBid orb_identifier);
    };
```

由于CORBA 规范允许多种形式的对象适配器，因此，很难通过使用一个通用的<OA>_init 操作来返回应用所需要的对象适配器。这样，就有必要针对每种对象适配器类型定义一个初始化操作。在CORBA2.0 规范中，唯一被定义的适配器是基本对象适配器（Basic Object Adapter，简称为BOA），所以，CORBA2.0规范中只定义了BOA_init操作，其形式如下面的伪IDL语言片段所示：

```
module CORBA{
   interface ORB{
     typedef string OAid;
     typedef sequence <string> arg_list;
     BOA BOA_init(inout arg_list argv,in OAid orb_identifier);
     };
}
```

(3) 获取初始对象引用。

当ORB初次启动时，客户并不知道它要调用的对象实现的对象引用。没有对象引用

就不能定位对象实现，也就不能进行调用请求。因此，ORB给出了获取初始对象引用的方法，应用可以通过调用它来获得初始化对象引用。ORB接口获取初始对象引用的操作以ORB对象的操作的形式出现，其具体形式如下面的伪IDL语言片段所示：

```
module CORBA{
    interface ORB{
      typedef string ObjectId;
      typedef sequence <ObjectId> ObjectIdList;
      exception InvalidName{};
      ObjectIdList list_initial_services();
      Object resolve_initial_references (in ObjectIdentifier) rasises(InvalidName);
    }
}
```

5) 对象适配器和基本对象适配器（BOA，见图6.3）

对象适配器OA位于服务对象实现和ORB之间，提供对象登记、对象引用生成、服务激活等服务。为满足大多数对象实现的需要，CORBA 2.0规范中定义了基本对象适配器

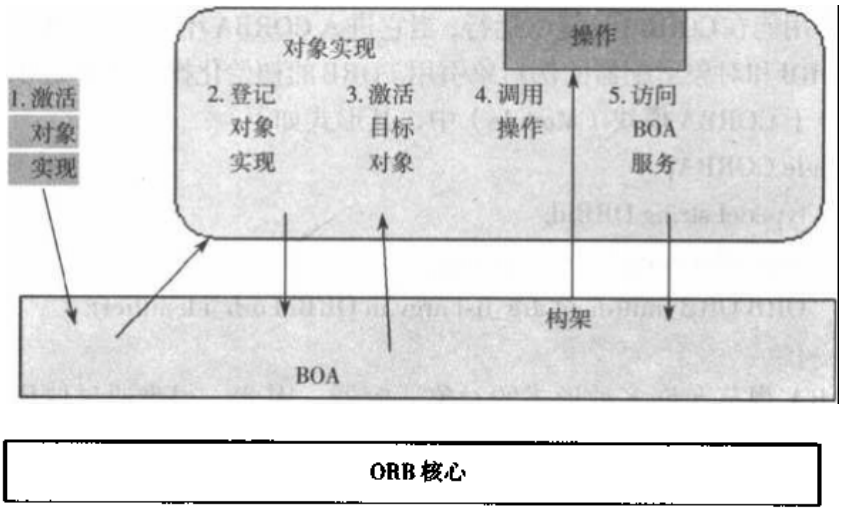

图6.3　BOA相关交互关系

BOA，BOA提供了以下功能：

(1) 产生和解释对象引用；

(2) 对请求进行认证，满足保密性的要求；

(3) 激活和去活实现；

(4) 激活和去活单个对象；

(5) 构架调用方法；

(6) 实现对象的登记。

6) IDL存根和动态调用接口（DII，见图6.4）

IDL存根由特定编程语言的一组函数（或对象）组成，它们构成了由IDL接口定义映

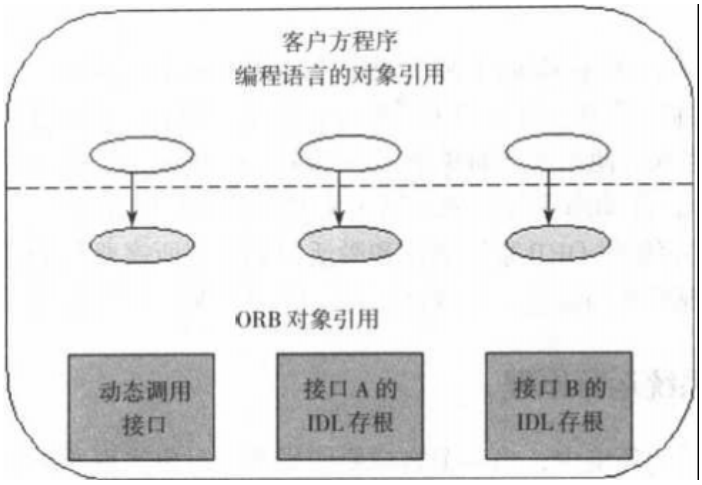

图 6.4　客户方程序的组成

射得到的对象接口定义的本地表示。IDL 存根通常是由 IDL 编译器编译目标对象的 IDL 接口描述文件而自动产生的。IDL 存根通过与客户方程序一起联编，把客户应用和 ORB 连接起来，使得客户调用远程对象就象调用本地对象或者本地函数一样。

动态调用接口（Dynamic Invocation Interface，简称 DII）由一组 CORBA 系统对象组成，这些对象提供客户动态地创建和调用对象请求的功能。在 DII 中主要包含以下三类请求操作：

(1) 同步调用操作（Synchronous Invocation）；

(2) 延迟同步调用操作（Deferred Invocation）；

(3) 单向调用操作（Oneway Invocation）。

7) IDL 构架和动态构架接口（DSI）（见图 6.5）

IDL 构架是根据目标对象接口的 IDL 描述由 IDL 编译器自动产生的对象实现的部件。通过它，可以使 ORB 核心 / 对象适配器调用对象实现中的操作方法。对象实现则根据 IDL 描述的接口给出具体的实现例程。

动态构架接口（Dynamic Skeleton Interface，简称 DSI）允许动态调用对象，它的基本思想是：让 ORB 核心 / 对象适配器对所有的对象调用请求都通过向上调用同一组接口例程来调用实际对象实现中操作，这组例程被称为动态调用例程（Dynamic Invocation Routine，

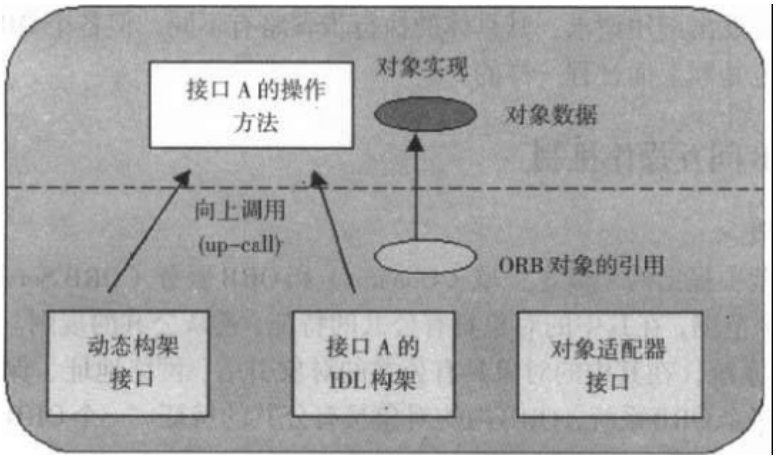

图 6.5　对象实现的组成

简称 DIR)。

8) 接口库 (Interface Repository) 和实现库 (Implement Repository)

接口库是用来存储、发布、管理相关对象接口定义的集合，它所提供的服务对象均为CORBA对象。它在实现时使用永久对象(Persistent Object)的机制，这使得它的用户可以随时查阅接口库的内容，并动态获得所加入的对象实现的IDL接口信息。

实现库的作用是存储供ORB系统定位和激活对象实现所需要的信息，还可以用来保存有关ORB对象实现的附加信息，如调试信息、管理控制、资源分配、安全信息等。

6.2.2 ORB系统运行原理

在单个ORB支撑的环境中，当一个客户要使用某个对象实现所提供的服务时，其操作步骤大致如下：

(1) 客户通过某种方式找到特定对象实现的对象引用，这些查询方式包括：

① 使用ORB接口中提供的操作 resolve_initial_references；

② 使用CORBA对象服务中的名录服务 (Naming Service)；

③ 通过其他接口的输出参数或者返回值；

④ 通过ORB实现系统自身提供的特定的对象引用获取方法。

(2) 如果该对象实现有相应的IDL存根，则客户可以通过该IDL存根向对象实现发出请求，否则，在接口库的协助下，客户可以使用动态调用接口来向对象实现发出请求。

(3) 当对象调用请求通过IDL存根或动态调用接口到达ORB核心以后，ORB核心负责请求的传送，将其送给相应的对象适配器，其具体的请求传递方式由ORB的具体实现决定。

(4) 对象适配器接到该请求后，判断一下所请求的对象实现是否有IDL构架存在。如果有，则对象适配器通过该IDL构架调用执行对象实现中的操作；否则，对象适配器将通过动态构架接口中的动态实现例程 (DIR) 来调用对象实现中的操作。

(5) 对象实现的特定操作方法执行完成后，结果 (包括输出参数、输入输出参数、返回值以及异常信息等) 将按照对象请求传递和执行路径逆向返回给客户对象。至此，一个完整的对象请求调用便完成了。

在包含多个异构ORB系统的环境中，客户可以通过自己所在的ORB系统向其他ORB系统上的对象实现发出调用请求，其具体的执行步骤略有不同，但各个ORB系统内的对象请求处理过程也是和上面过程一样的。

6.2.3 ORB间互操作机制

1) 互操作的概念

在互操作规范中提出两个概念：域 (Domain) 和ORB服务 (ORB Services)。

域，是指一个范围，在其中的对象具有公共的特征，遵从公共的规则。一个域可以是一个单独的ORB系统，在其中的对象具有公共的对象引用、网络地址、保密机制等；一个域也可以跨越多个ORB系统，ORB内的对象具有公共的特征。一个ORB系统内部也可以划分为多个域。

ORB服务的设置是为了使ORB内部结构明确化，在ORB系统中ORB核心主要用来提供对象引用的基本表示和调用请求的传输机制。对象请求中某些隐含属性会影响请求传递方式，如事务属性、安全属性等，它们使ORB核心以不同方式来传递对象调用请求。通过把这些任务交给ORB服务来完成，以后就可以通过增加ORB服务的方式来支持新特性而无需修改ORB体系结构，一个ORB服务可对应于某一具体的隐含属性。ORB服务既可以架在ORB核心之上，也可以和ORB核心结合为一体。使用ORB服务的请求调用过程大致如下：

(1) 客户产生一个对对象实现的目标对象的调用请求，其中包括给定目标对象的对象引用、操作名称、操作参数以及隐含的上下文（context）。

(2) 在客户方，该调用请求由客户的ORB系统的某个特定ORB服务进行处理，然后传递到对象实现方。

(3) 在对象实现方，ORB系统接收对象调用请求之后，ORB系统中的等同ORB服务对请求进行处理，然后再传递给对象适配器，并交给IDL构架或者动态构架进行实际的处理。

(4) 对象实现方的目标服务对象执行具体操作的实现代码。

(5) 操作结果（参数、返回值），包括异常信息以类似方式返回给客户。

2) 互操作桥连机制

为了实现不同ORB域之间的互操作，当对象调用请求/响应的交互穿越互操作域的边界时，必须使用一种映射（Mapping）机制或桥连（Bridging）机制来转换交互中的相关信息元素。在CORBA2.0规范中，主要采用桥连机制来进行这种转化。

从桥接策略上分，桥接机制可以划分为两大类：

(1)间接桥接。在间接桥接方式下，在域边界上那些和域内私有特性有关的交互成员，将在域内私有形式和某一种公共形式之间进行转换。它有以下特点：

- 采用的公共形式，既可以是两个特定的ORB域实现间的公共形式，也可以采用一种公共标准来作为公共形式；
- 可以有不止一种公共形式，每一种公共形式对某一特定应用进行优化；
- 如果有不止一种公共形式，选择的公共形式的方式可以是静态的也可以是动态的。

(2) 直接桥连。在直接桥连方式上，那些和域内私有特性有关的交互成员在通过域边界时，将直接在两个域的内部形式之间进行转换。

从桥连层次上分，桥接机制也可以划分为两类：

(1) 内嵌式桥连（In_line Bridging）方式，如图6.6所示。

所谓内嵌式桥，是指在ORB内部能够执行必要的转换和映射的代码。在这种桥接方式中，所要求的桥接功能可以通过不同层次的ORB组成成员结合来提供；使用低层ORB核心的附加或可选的ORB服务来提供桥接功能；使用附加或可选的存根和构架代码来提供桥接功能。

(2) 请求式桥接（Request_Level Bridging）方式，如图6.7所示。

在ORB系统之外执行请求转换或映射的代码称为请求式桥。请求式桥通过公共的协议在不同执行环境间进行对象调用请求/响应之间的间接转换，在每个域中请求式桥部分称为“半桥”（Half Bridge）。如果转换在同一执行环境内部发生，称为“全桥”。

请求式桥接的一般过程如下：

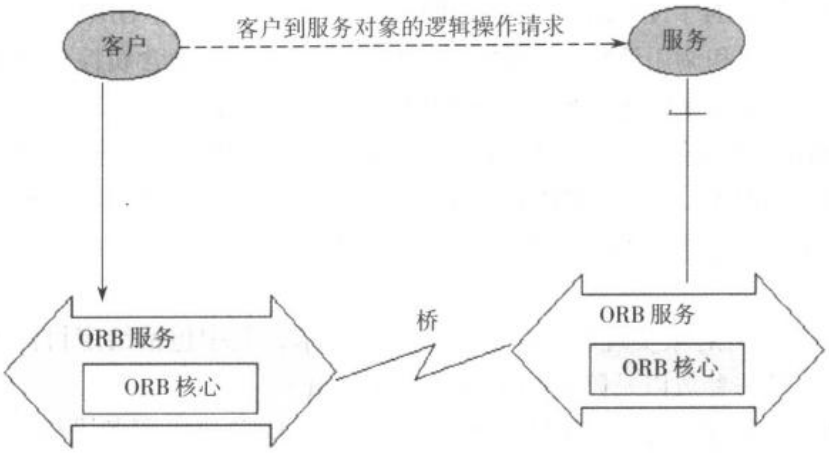

图 6.6　内嵌式桥接机制示意图

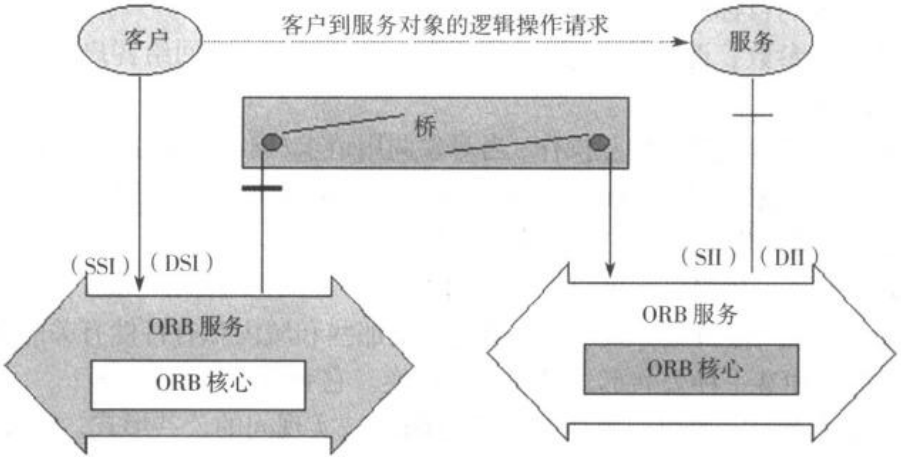

图 6.7　请求式桥接机制示意图

● 客户的目标对象调用请求传递客户所在 ORB 系统中的代理对象；

● 代理对象使用本方半桥把调用请求转换成为能够被对象实现所在的ORB系统理解的形式，然后经过传输交给对象实现所在 ORB 系统中的代理对象；

● 对象实现所在ORB系统中的代理对象使用单一ORB系统的运行机制，执行对对象实现操作的调用；

● 操作结果（参数、返回值）以及异常信息通过相反的路径（也经过类似的转换）返回给客户。

构造一般请求式桥与以下几个 ORB 组成成员有关系：

● 动态调用接口（DII）　请求式桥可能需要通过它来调用某一对象引用的对象实现，此对象实现可以在请求式桥加入 ORB 环境以后才加入 ORB 环境；

● 动态构架接口（DSI）　它产生代理对象，请求式桥可能需要处理代理对象发出的调用请求；

● 接口库　请求式桥可能需要通过它来获得驱动 DII 和 DSI 的信息，如对象类型、支

持的操作、操作参数的类型、返回值、异常信息等；

● 对象适配器　它用来创建代理对象的对象引用，对象引用既可以在请求桥初始化时创建，也可以在映射对象引用时创建；

● CORBA 对象引用　它用来指明具体的对象实现。

3) 可互操作的对象引用 IOR

可互操作的对象引用 IOR（Interoperable Object Reference）由带标签的框架文件（Profile）组成，这些框架文件定义了在一个具体的传输模式下定位一个对象所需的全部信息。框架文件有以下特点：

(1) 框架文件必须是独立、完整和自包含的，它不使用别的框架文件中的信息。

(2) 所有使用 IOR 的客户请求调用仅仅只能使用一个框架文件中的信息。

(3) 在一个框架文件中可以同时存在用来驱动多个 ORB 间协议的信息，它们之间可能存在信息共享。

(4) 在定义一个特定框架文件时，多个带同一标签的框架文件必须包含在一个 IOR 之中。

(5) 数字标签可以是公共的也可以是私有的。

4) 带互操作的请求执行机制

在请求式桥接情况下，如果有请求传送给 DSI，DSI 根据对象引用判断其指向的对象实现是否在本地域内。如果在本地域内，执行方式和在单一 ORB 域内执行情况是相同的；否则，DSI 将产生一个代理对象，代理对象引用由基本对象适配器来产生，以便客户和半桥与代理对象进行交互。代理对象通过本方半桥将请求转换成中间协议的表示形式，传送给目的 ORB 系统，目的 ORB 系统上的半桥将中间协议表示形式转换为本地 DII 请求来完成对象实现中的方法调用，结果按原路返回。

5) ORB 间互操作协议(见图 6.8)

为了达到支持各种 ORB 间互操作的目的，CORBA2.0 规范中定义了 ORB 间通信的标准协议 GIOP（General Inter-ORB Protocol），它定义了用于 ORB 间通信的一种标准传输语法和一组消息格式。GIOP 由以下三个部分组成：

(1) 公共数据表示。公共数据表示（Common Data Representation，简称 CDR）是 GIOP 表示 IDL 数据类型的格式标准，它将 IDL 数据类型映射到低级的二进制流，以便网络传输。

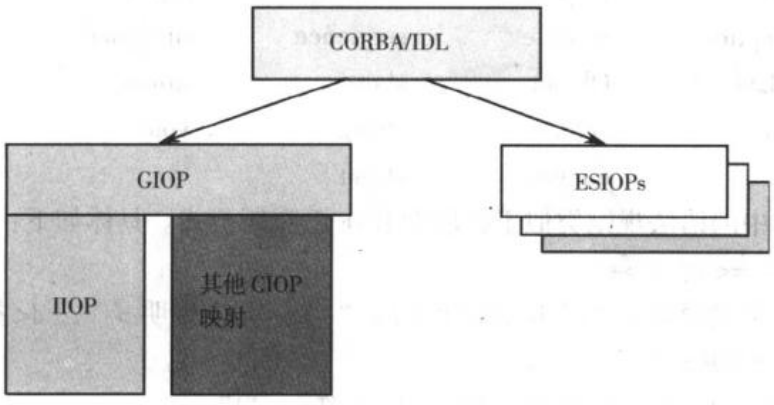

图 6.8　互操作协议之间的关系

(2) GIOP 消息格式。GIOP 协议定义了七种消息格式，用于 ORB 系统间信息交换，从而实现对象请求、对象定位和信道管理等任务。

(3) GIOP 消息传递。GIOP 协议可运行于多种面向连接的运输层协议，还允许 ORB 重用连接资源。

GIOP 协议只是一种抽象协议，在实现时必须映射到具体的运输层协议或者特定的运输机制之上。GIOP 协议到 TCP/IP 协议的映射又称为 IIOP 协议，而 GIOP 协议到 DCE 的映射又称为 ESIOP。

IIOP（Internet Inter-ORB Protocol）是 GIOP 的一种映射，它定义了用于 Internet 之上的一种标准互操作协议，它利用的运输层协议就是 Internet 所采用的 TCP 协议。

CORBA 规范中还规定了使用于特定应用环境的 ORB 间互操作协议（Environment-Specific Inter-ORB Protocol，简称 ESIOP）。当用户已经大规模采用了某种特殊的网络技术或分布式计算技术（如 DCE）时，ESIOP 便可以在这样的特殊应用环境中提供互操作的能力。

6.3 IDL 语言与编译器

6.3.1 IDL 语言

IDL语言是一种接口定义语言。所谓接口定义语言是专门用于描述数据和行为过程的语言。OMG IDL 语言不同于所有已有的程序设计语言，它是一种描述性语言，也就是说，用它描述得到的结果是不能直接被编译执行的。它有完整的语法规则和语义解释，从形式上看，非常类似于 C++ 语言。

OMG IDL 语言采用 ISO Latin-1(8859.1)字符集。该字符集可以分为字母、数字、图形符号、空格符和格式符号。其中字母包括英文 26 个字母的大小写，数字包括 10 个阿拉伯数字 0 到 9。

OMG IDL 语言中还定义了一组关键字，它们是：

any	default	inout	out	switch
attribute	double	interface	raises	TRUE
boolean	enum	long	readonly	typedef
case	exception	module	sequence	unsigned
char	FALSE	Object	short	union
const	float	octet	string	void
context	in	oneway	struct	

OMG IDL 语言中的语法规则类似于扩展的 BNF 巴克斯范式。具体如下：

<规格说明> :: =<定义>+

<定义> :: =<类型说明> ";" |<常量说明> ";" |<异常说明> ";" |<接口> ";"
|<模块> ";"

<模块> :: = "module" <标识符> "{" <标识符>+ "}"

<接口> :: =<接口说明>|<前向说明>

<接口说明> :: =<接口首部> "{" <接口体> "}"
<前向说明> :: = "interface" <标识符>
<接口首部> :: = "interface" <标识符>[<继承规格说明>]
<接口体> :: =<输出>
<输出> :: =<类型说明> ";" |<常量说明> ";" |<异常说明> ";" |<属性说明> ";" |
<操作说明>;
<继承规模说明> :: = ":" <作用域名>{"," <作用域名>}*
<作用域名> :: =<标识符>| "::" <标识符>|<作用域名> "::" <标识符>
<常量说明> :: = "const" <常量类型><标识符> "=" <常量表达式>
<常量类型> :: =<整型>|<字符类型>|<布尔类型>|<浮点类型>|<字符串类型>|<作用域名>
<常量表达式> :: =<位或表达式>
<位或表达> :: =<位异或表达式>|<位或表达式> "|" <位异或表达式>
<位异或表达> :: =<位与表达式>|<位异或表达式> "^" <位与表达式>
<位与表达式> :: =<移位表达式>|<位与表达式> "&" <移位表达式>
<移位表达式> :: =<和表达式>|<移位表达式> ">>" <和表达式>|<移位表达式> "<<"
<和表达式>
<和表达式> :: =<乘表达式>|<和表达式> "+" <乘表达式>|<和表达式> "-" <乘表达式>
<乘表达式> :: =<单操作符表达式>|<乘表达式> "*" <单操作符表达式>|
<乘表达式> "/" <单操作符表达式>|<乘表达式> "%"
<单操作符表达式>
<单操作符表达> :: =<单操作符><主表达式>|<主表达式>
<单操作符> :: = "-" | "+" | "~"
<主表达式> :: =<作用域名>|<文字>| "(" <常量表达式> ")"
<文字> :: =<整型文字>|<字符文字>|<布尔文字>|<浮点文字>|<字符串文字>
<布尔文字> :: = "TRUE" | "FALSE"
<正整数常量> :: =<常量表达式>
<类型说明> :: = "typedef" <类型说明符>|<结构类型>|<联合类型>|<枚举类型>
<类型说明符> :: =<类型规格说明><说明符串>
<类型规格说明> :: =<简单类型规格说明>|<构造类型规格说明>
<简单类型规格说明> :: =<基类型规格说明>|<模板类型规格说明>|<作用域名>
<基类型规格说明> :: =<浮点类型>|<整型>|<字符类型>|<布尔类型>|
<位组类型>|<any 类型>
<模板类型规格说明> :: =<序列类型>|<字符串类型>
<构造类型规格说明> :: =<结构类型>|<联合类型>|<枚举类型>
<说明符串> :: =<说明符>{"," <说明符>}*
<说明符> :: =<简单说明符>|<复杂说明符>
<简单说明符> :: =<标识符>

<复杂说明符> :: =<数组说明符>
<浮点类型> :: = “float” | “double”
<整型> :: =<带符号整数>|<无符号整数>
<带符号整数> :: =<带符号长整数>|<带符号短整数>
<带符号长整数> :: = “long”
<带符号短整数> :: = “short”
<无符号整数> :: =<无符号长整数>|<无符号短整数>
<无符号长整数> :: = “unsigned” “long”
<无符号短整数> :: = “unsigned” “short”
<字符类型> :: = “char”
<布尔类型> :: = “boolean”
<位组类型> :: = “octet”
<any 类型> :: = “any”
<结构类型> :: = “struct” <标识符> “{” <成员表> “}”
<成员表> :: =<成员>+
<成员> :: =<类型规格说明><说明符串> “;”
<联合类型> :: = “union” <标识符> “switch” “(” <选择类型规格说明> “)”
“{” <选择体> “}”
<选择类型规格说明> :: =<整型>|<字符类型>|<布尔类型>|<枚举类型>|<作用域名>
<选择体> :: =<情况>+
<情况> :: =<情况符>+<元素规格说明> “;”
<情况符> :: = “case” <常量表达式> “:” | “default”
<元素规格说明> :: =<类型规格说明><标识符>
<枚举类型> :: = “enum” <标识符> “{” <枚举符>{“,” <枚举符>}* “}”
<枚举符> :: =<标识符>
<序列类型> :: = “sequence” “<” <简单类型规格说明> “,” <正整数常量> “>” |
“sequence” “<” <简单类型规格说明> “>”
<字符串类型> :: = “string” “<” <正整数常量> “>” | “string”
<数组说明符> :: =<标识符><定长数组长度>
<定长数组长度> :: = “[” <正整数常量> “]”
<属性说明> :: =[“readonly”]“attribute” <操作类型规格说明><简单说明符>{“,”
<简单说明符>}*
<异常说明> :: = “exception” <标识符> “{” <成员> “}”
<操作说明> :: =[<操作属性>]<操作类型规格说明><标识符><参数说明串>[
<异常表达式>]
[<上下文表达式>]
<操作属性> :: = “oneway”
<操作类型规格说明> :: =<参数类型规格说明> | “void”

<参数说明串> :: = "(" <参数说明>{"," <参数说明>}* ")" | "(" ")"
<参数说明> :: =<参数属性><参数类型规格说明><简单说明符>
<参数属性> :: = "in" | "inout" | "out"
<异常表达式> :: = "raises" "(" <作用域名>{"," <作用域名>}* ")"
<上下文表达式> :: = "context "(" <字符串文字>{"," <字符串文字>}* ")"
<参数类型规格说明> :: =<基类型规格说明>|<字符串类型>|<作用域名>
在对象的继承属性方面，还需要加以特别的说明：

(1) 一个特定对象中的所有操作都应该有一个唯一可以区别于其他操作的名字。这就限制了不能重新定义从其他对象中继承的操作。

(2) 引用任何的常量、类型和异常都和定义它们的对象相联系在一起。也就是继承对象中的常量、类型和异常不能重新定义，这样就不会影响基类中定义的相应的常量、类型和异常。

6.3.2 IDL/Java映射

6.3.2.1 命名

IDL名字和标识符在映射为Java的名字和标识符时，一般不作改动。如果映射后得到的Java代码中产生命名冲突，解决的方法是在映射后的名字前加下划线（_）。

由于Java语言的特性，一个IDL结构可能被映射为几个不同名字的Java结构。附加的那些名字通过在名字后加一个描述的后缀来实现。例如，IDL接口foo被映射为Java接口foo，以及附加的Java类fooHelper和fooHolder。

在某些特殊情况下，附加的名字会与经过其他映射后得到的名字相冲突，前述的冲突解决方法同样适用于那些经过其他映射后得到的名字。

如果某些IDL名字不作改变就映射为Java标识符，会与Java的保留字冲突的话，则也要对它们运用冲突解决方法。

Java语言的关键字包括：

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

在以下的情况中，也要求映射后得到的名字不能与其他名字发生冲突：

(1) Java类<type>Helper，其中<type>是用户自定义的IDL类型的名字；

(2) Java类<type>Holder，其中<type>是用户自定义的IDL类型的名字（有些例外情况

如，类型定义 typedef 的别名映射后不产生 <type>Holder);

(3) Java 类 <basicJavaType>Holder，其中 <basicJavaType> 是一个 IDL 基本类型所使用的 Java 标准数据类型；

(4) 嵌套的 Java 包名 <interface>Package，其中 <interface> 是 IDL 接口的名字。

6.3.2.2 模块映射

模块是 IDL 中的名字空间机制。Java 中相应的名字空间机制是 Java 包。一个 IDL 模块被映射为同名的一个 Java 包。所有在模块中定义的 IDL 类型被映射为所生成的包里的 Java 类或接口。

```
//IDL 语言描述
module Example{…}
```

将被映射为：

```
// 产生的 Java 代码
package Example;
…
```

6.3.2.3 支持类（Holder Classes）

如果要支持 IDL 语言中 out 类型参数和 inout 类型参数的传递模式，则必须使用附加的支持类。所有 IDL 基本数据类型的支持类都在 org.omg.CORBA 包中，而且所有用户自定义的类型名，除了那些由类型定义（typedef）定义的外，都有其支持类。

对于用户自定义的类型，其支持类的名字是在映射得到的（Java）类型名字后加上 Holder 而构成的。

对于 IDL 基本数据类型，其支持类名字是在映射得到的 Java 类型名后加上 Holder，例如 IntHolder。如有名字冲突，按照命名冲突解决方法实施。

对于一个 IDL 数据类型 type，其支持类包括：一个公共的实例成员 value，其类型是 type 中相应的映射类型；一个默认的构造函数，它把 value 的值置为 Java 语言规定的默认值，比如 boolean 类型设为 false，数值或字符类型为 0，字符串和对象为 null；一个带有参数 initial 的构造函数，它将 value 的初始值设为 initial。

为了支持可移植的客户存根和服务器构架，用户自定义类型的支持类还一定要实现 org.omg.CORBA.portable.Streamable 接口。

基本类型 <type> 的支持类一般定义如下。注意它们不需要实现 org.omg.CORBA.portable.Streamable 接口。

```
//Java 代码
package org.omg.CORBA;
final public class <Type>Holder{
public <Type> value;
public <Type>Holder(){}
public <Type>Holder(<type>initial){
value=initial;
        }
}
```

用户自定义类型的支持类如下所示:

```
// Java 代码
final public class<foo>Holder
    implements org.omg.CORBA.portable.Streamable{
      public<foo>value;
      public<foo>Holder(){}
      public<foo>Holder(<foo>initial){}
      public void_read(org.omg.CORBA.portable.InputStream i)
      {…}
      public void_write(org.omg.CORBA.portable.OutputStream o)
      {…}
      public org.omg.CORBA.TypeCode_type(){…}
}
```

6.3.2.4 助手类(Helper Classes)

所有用户自定义的IDL类型都有一个附加的“助手”类，其名字是生成的类型名字后加Helper后缀。它提供了几种静态方法，用于对该类型的对象进行操作，包括该类型的插入和析取操作、获得库标识（repository id.）操作、获得类型编码（typecode）操作以及把该类型写入流或从流中读出的操作。另外，助手类还定义了narrow操作。IDL枚举类型的助手类还定义了from_int操作。

对所有用户自定义的IDL类型<typename>，将生成如下的Java代码。

```
// 产生的 Java 助手类
public class <typename>Helper{
      public static void insert(org.omg.CORBA.Any a,<typename>t){…}
      public static <typename> extract(Any a) {…}
      public static org.omg.CORBA.TypeCode type(){…}
      public static String id(){…}
      public staic <typename>read(org.omg.CORBA.portable.InputStream istream)
      {…}

      public static void write(org.omg.CORBA.portable.OutputStream ostream,<typename> value)
      {…}

      // 仅用于接口类型助手类
      public static <typename>narrow(org.omg.CORBA.Object obj);

      // 仅用于枚举类型助手类
      public static <enum_name> from_int(int value);
      }
```

6.3.2.5 基本类型映射

基本类型是IDL/Java映射中的基础部分。表6.1说明了基本类型的映射规则。异常(Exceptions)栏列出了当IDL类型与映射的Java类型不匹配时可能出现的异常情况。当一个Java类型的表示范围比IDL“大”的时候，则可能会发生潜在的不匹配。另外，在运行过程中，如果一个对象被作为in或inout参数编码时，应当进行有效性检查。

在使用IDL语言中的无符号（unsigned）类型时要特别注意。由于Java语言不支持无符号类型，客户程序要确保较大的IDL无符号类型的值能够被正确地作为Java的负整数进行处理。

表6.1 基本类型的映射

IDL类型	Java类型	异常
boolean	boolean	--
char	char	CORBA::DATA_CONVERSION
wchar	char	--
octet	byte	--
string	java.lang.String	CORBA::MARSHAL
		CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	--
unsigned	short	--
long	int	--
unsigned long	int	--
long long	long	--
unsigned long long	long	--
float	float	--
double	double	--

目前的Java语言尚不直接支持IDL语言中新的定点类型（fixed）以及长双精度类型（long double） 它们有可能按表6.2所示的方法映射。

表6.2 未来可能支持类型的映射

IDL类型	Java类型	异常
long double	现在还没有对应的类型	--
fixed	java.math.BigDecimal	CORBA::DATA_CONVERSION

还需要说明的是关于Java null值的使用。Java null值只是用来表示“空”的对象引用。例如，空字符串一定要用一个长度为0的字符串来表示，而不能用null来表示。数组也同样如此。

1) 布尔类型映射

IDL 布尔常量 TRUE 和 FALSE 映射为相应的 Java 布尔值 TRUE 和 FALSE。

2)字符类型的映射

IDL 字符是 8 位 ISO8859.1 字符集中的元素，而 Java 字符是 16 位无符号的 Unicode 字符集中的成员。为了保证类型安全，在对方法中的调用参数进行编码时，基于 Java 开发的 CORBA 系统将确保所有从 IDL 字符映射得到的 Java 字符的范围有效性。如果字符在 ISO8859.1 定义的范围之外，将出现一个 CORBA::DATA_CONVERSION 异常。

此外，IDL wchar 字符类型映射为 Java 基本类型 char。

3) 8 位组

8 位组的 IDL 类型 octet，映射为 Java 类型 byte。

4) 字符串类型

IDL语言中有界或无界的字符串都被映射为java.lang.String。对于字符的范围检查以及对字符串的越界检查在对它们进行编码时进行。字符越界问题将导致一个 CORBA::DATA_CONVERSION 异常的出现；同时，字符串越界问题也将导致一个 CORBA::MARSHAL 异常的出现。

IDL语言中有界或无界的wstring类型都被映射为java.lang.String。同样，越界将导致一个 CORBA::MARSHAL 异常的出现。

5) 整型和浮点类型

整型和浮点类型（float 和 double）的映射详见表 6.1。

6) 未来的定点类型和长双精度类型

IDL 语言中的 fixed 类型将映射为 Java 的 java.math.BigDecimal 类型。范围的冲突将引发一个 CORBA::DATA_CONVERSION 异常。

现在，Java中还没有对IDL长双精度类型的支持。这种类型可能作为 java.math.BigFloat 包而加入 Java 中。

6.3.2.6 常量映射

Java 程序员一般把常量定义成类或接口中的静态常量成员。所以，IDL接口中的常量映射为Java中的静态常量成员（public static final）较为自然。在模块域内或全局定义的每个IDL常量都产生一个同名的 Java 类，其中含有一个 public static final 类型的值 value。这样可以方便地支持全局常量。这些类只在编译时使用，因为生成的 Java 二进制码运行时不再依赖它们。

1) 接口中定义的常量的映射

```
//IDL 语言描述
module Example{
    interface Face{
        const long aLongerOne=-321;
            }
        }
```

将被映射为：

```
// 产生的 Java 代码
```

```
package Example;
public interface Face{
        public static final int aLongerOne=(int)(-321L);
        }
```

2) 不在 IDL 接口中定义的常量的映射

```
//IDL 语言描述
module Example{
        const long aLongOne=-123;
};
```

将被映射为：

```
// 产生的 Java 代码
package Example;
public interface aLongOne{
    public static final int value=(int)(-123L);
```

6.3.2.7 枚举类型映射

IDL 语言中的枚举类型映射为同名的 Java final 类，它定义了一个 value 方法，每个标签 (label)有两个静态数据成员和一个私有的构造函数。如果有如下的 IDL 描述：

```
//IDL 语言描述
Enum EnumType{a,b,c};
```

将被映射为：

```
// 产生的 Java 代码
public final class EnumType {
    public static final int_a = 0;
    public static final EnumType a = new EnumType(_a);
    public static final int_b = 1;
    public static final EnumType b = new EnumType(_b);
    public static final int_c = 2;
    public static final EnumType c = new EnumType(_c);
    public int value() {…}
    // 构造函数
    private EnumType(int){…}
};
```

一个成员是与 IDL 枚举标签同名的 public static final 类型；另一个在名字之前带一下划线，在 switch 语句中使用。

Value 方法返回整型值，该值是由 0 开始顺序赋值得到。注意，即使有一个标签名字为 value 也不会和 value 方法冲突。

枚举类型的助手类中除了通常的方法外，还有一个特殊的 from_int 方法，它返回一个整数所指定的枚举值。

6.3.2.8 结构类型映射

IDL语言中的结构类型映射成同名的Java类，包含IDL结构类型中的成员变量以及构造函数。同时提供一个空的构造函数供以后进一步扩展。如果有如下的IDL描述：

```
//IDL语言描述
struct StructType{
        long field1;
        String field2;
        };
```

将被映射为：

```
//产生的Java代码
final public class StructType{
// 变量
  public int field1;
  public String field2;
// 构造函数
public StructType(){}
public StructType(int field1,String field2)
{…}
}
```

6.2.3.9 联合类型映射

IDL语言中的联合类型映射为一个同名的Java类，它包含：

(1) 一个缺省的构造函数；

(2) 一个访问判别子的方法，称为discriminator()；

(3) 每个分支的访问方法；

(4) 每个分支的修改方法；

(5) 含有不止一个标识的分支的修改方法；

(6) 一个默认的修改方法。

分支的访问和修改方法需要重载，它们都使用分支的名字作为方法名。如果要访问的分支还没有置值，那么访问方法将出现CORBA::BAD_OPERATION系统异常。

如果一个分支对应n个情况(case)标签，那么将生成一个访问方法和n个修改方法。其中一个简单的修改方法是把判别子置为第一个情况标签的值。其他的修改方法采用一个显式的判别子参数来指定判别子。

如果分支对应默认标签，那么修改函数把判别子设置为与其他标签所对应的值都不相同的值。对于一个联合类型，如果所列出的情况标签完全地覆盖了判别子的值域，那么就不能再定义默认标签。

如果一个联合类型没有显式地定义默认标签，而且已列出的情况标签没有完全地覆盖判别子的值域，那么将生成一个默认修改方法。

```
//IDL语言描述
union UnionType switch(EnumType){
```

```
case first:long win;
case second:short place;
case third:
case fourth:octet show;
default:boolean other;
};
```

将被映射为：

```
// 产生的 Java 代码
public class UnionType{
  // 构造符
  public UnionType(){…}
  // 差异符的访问函数
  public <switch-type>discriminator(){…}
  //win 操作
  public int win(){…}
  public void win(int value) {…}
  //place 操作
  public short place(){…}
  public void place(short value) {…}
  //show 操作
  public byte show(){…}
  public void show(byte value) {…}
  public void show(int discriminator,byte value) {…}
  //other 操作
  public boolean other(){…}
  public void other(boolean value) {…}
}
```

6.3.2.10 序列类型映射

IDL语言中的序列类型映射为一个同名的Java数组。当它作为IDL操作的参数时，应检查有界序列的界限，如越界将引发 CORBA::MARSHAL 异常。

```
//IDL 语言描述
typedef sequence <long>UnboundedData;
typedef sequence<long,42>BoundedData;
```

将被映射为：

```
// 产生的 Java 代码
public int[] UnboundedData;
public int[] BoundedData;
final public class UnboundedDataHolder
 implements org.omg.CORBA.portable.Streamable{
```

```
        public int[] value;
        public UnboundedDataHolder() {};
        public UnboundedDataHolder(int[] initial) {…};
        public void_read(org.omg.CORBA.portable.InputStream i)
        {…}
        public void_write(org.omg.CORBA.portable.OutputStream o)
        {…}
        public org.omg.CORBA.TypeCode_type(){…}
    }

    final public class BoundedDataHolder
      implements org.omg.CORBA.portable.Streamable{
          public int[] value;
          public BoundedDataHolder() {};
          public BoundedDataHolder(int[] initial) {…};
          public void_read(org.omg.CORBA.portable.InputStream i)
          {…}
          public void_write(org.omg.CORBA.portable.OutputStream o)
          {…}
          public org.omg.CORBA.TypeCode_type(){…}
      }
```

6.3.2.11 数组类型映射

IDL语言中数组映射为一个同名的Java数组。当它作为IDL操作的参数时，应检查数组的界限。如果越界，将引发CORBA::MARSHAL异常。

```
//IDL语言描述
const long arrayBound=42;
typedef long larray[arrayBound];
将被映射为：
// 产生的Java代码
public int[] larray;
final public class larrayHolder
   implements org.omg.CORBA.portable.Streamable{
     public int[] value;
     public larrayHolder(){}
     public larrayHolder(int[] initial) {…}
     public void_read(org.omg.CORBA.portable.InputStream i)
     {…}
     public void_write(org.omg.CORBA.portable.OutputStream o)
     {…}
```

```
    public org.omg.CORBA.TypeCode_type(){…}
  }
```

6.3.2.12 接口类型映射

1) 基本映射

IDL语言中的接口类型映射为一个同名的公共Java接口、一个助手类和一个支持类。Java接口继承了基类org.omg.CORBA.Object。助手类包含一个静态的narrow方法，可以使用它把一个org.omg.CORBA.Object类型的对象转化为一个更精确类型的对象引用。如果narrow失败，将引发CORBA::BAD_PARAM异常。特别地，Java的null值可以传到任何需要对象引用的地方。

接口类型中的属性（Attribute）映射为一对同名的访问和修改方法。如果属性为只读（readonly）属性，则只有访问方法，没有修改方法。

IDL中的接口继承关系直接映射为Java的接口继承关系。如果有以下的IDL描述：

```
//IDL语言描述
module Example{
   interface Face{
      long method(in long arg) raises(e);
      attribute long assignable;
      attribute readonly long nonassignable;
     }
}
```

将被映射为：

```
//产生的Java代码
package Example;
public interface face extends org.omg.CORBA.Object{
      int method(int arg) throws Example.e;
      int assignable();
      void assignable(int i);
      int nonassignable();
 }
public class FaceHelper{
//其他标准的助手类方法
      public static Face narrow(org.omg.CORBA.Object obj)
      {…}
  }

  final public class FaceHolder
    implements org.omg.CORBA.portable.Streamable{
      public Face value;
      public FaceHolder(){}
```

```
    public FaceHolder(Face initial) {…}
    public void_read(org.omg.CORBA.portable.InputStream i)
    {…}
    public void_write(org.omg.CORBA.portable.OutputStream o)
    {…}
    public org.omg.CORBA.TypeCode_type(){…}
}
```

2) 参数传递方式

IDL语言中的int类型参数使用传值的语义，直接映射为普通的Java参数。IDL操作的返回值作为相应的Java方法的返回值返回。

IDL语言中的out类型和inout类型参数使用传结果和传值/传结果的语义，不能直接映射为Java的参数传递机制。IDL/Java映射规范对所有的IDL标准和用户自定义类型定义上附加的支持类，用于在Java中实现这些参数模式。客户为每一个IDL文件中的out和inout类型参数提供一个Java支持类的实例，用于调用改变支持类（而不是实例本身）的内容。客户在调用返回后使用修改过的内容。

```
//IDL语言描述
module Example{
   interface Modes {
      long operation(in long inArg,out long outArg,inout long inoutArg);
           };
      };
```

在上面这个IDL定义中，操作的结果作为普通的结果返回，in参数只作为普通的值。但是对于out和inout参数，一定要构造一个适当的支持类。下面是一个典型的使用示例：

```
//用户编写的Java代码
//选择一个目标对象
Example.Modes target=…;

//得到in参数的实际值
int inArg=57;
//准备接收out参数
IntHolder outHolder=new IntHolder();
//对inout参数赋值
IntHolder inoutHolder=new IntHolder(131);

//实施调用
int result =target.operation(inArg,outHolder,inoutHolder);
```

```
// 使用 outHolder 对象的值
…outHolder.value…

// 使用 inoutHolder 对象的值
…inoutHolder.value…
```

在调用之前，inout参数的输入值一定要设置到将作为实参的支持类实例中去。调用完成后，客户可以使用outHolder.value来访问out参数的值，使用inoutHolder.value来访问inout参数的输出值。IDL操作的返回值由调用的结果返回。

6.3.2.13 异常类型映射

IDL语言中异常类型的映射同结构类型的映射类似。它映射为Java类，提供构造函数，并对异常的属性提供实例变量。CORBA的系统异常是不需要检查的异常，它们（间接地）继承java.lang.RuntimeException类；用户自定义异常是需要检查的异常，它们（间接地）继承java.lang.Exception类。异常类型的继承关系如图6.9所示。

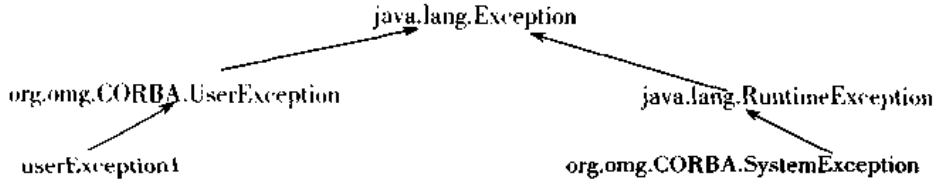

图6.9 异常类型的继承关系

1) 用户自定义异常

用户自定义异常映射为一个final Java类，它继承org.omg.CORBA.UserException。它的映射类似IDL结构类型的映射，包括生成帮助类和支持类。

如果异常定义在一个嵌套的IDL域中（一般在接口中），那么它相应的Java类名就会定义在一个特殊的域中，否则它将定义在该异常所在的IDL模块所对应的Java包中。

如果有如下的IDL描述：

```
//IDL 语言描述
module Example {
    exception ex1{string reason;};
};
```

被映射为：

```
// 产生的 Java 代码
package Example;
final public class ex1 extends org.omg.CORBA.UserException{
    public String reason;// 实例
    public ex1(){…}// 缺省构造函数
    public ex1(String r){…}// 构造函数
```

```
    }
```

2) 系统异常

另一类异常是标准的IDL系统异常，它映射后，成为Java的final类，该类继承了org.omg.CORBA.SystemException。注意org.omg.CORBA.SystemException没有公共的构造函数，只有那些继承了它的类才可以被实例化。

IDL异常的Java类名与它的IDL名字相同，它定义在org.omg.CORBA包中。默认的构造函数把minor code值置为0，把completion code值置为COMPLETED_NO，并把原因字符串值置为空串。当然它还包括其他的构造函数。

```
    // 来自 org.omg.CORBA 包
    package org.omg.CORBA;
    public final class CompletionStatus{
    //Completion Status 常量
    public static final int_COMPLETED_YES=0,
    _COMPLETED_NO=1,
    _COMPLETED_MAYBE=2;
    public static final CompletionStatus COMPLETED_YES=
        new CompletionStatus(_COMPLETED_YES);
    public static final CompletionStatus COMPLETED_NO=
        new CompletionStatus(_COMPLETED_NO);
    public static final CompletionStatus COMPLETED_MAYBE=
        new CompletionStatus(_COMPLETED_MAYBE);
    private CompletionStatus(int){…}
    }
abstract public class
SystemException extends java.lang.RuntimeException
{
     public int minor;
     public CompletionStatus completed;
}
final public class
  UNKNOWN ectends org.omg.CORBA.SystemException{
    public UNKNOWN()…
    public UNKNOWN(int minor,CompletionStatus completed)…
    public UNKNOWN(String reason)…
    public UNKNOWN(String reason,int minor,CompletionStatus completed)…
    }
…
// 对其他的每个IDL系统异常都有一个与上面类似的定义
```

6.3.2.14 any 类型映射

IDL 语言中的 any 类型映射为 Java 类 org.omg.CORBA.Any。这个类具有所有必需的方法用以对预定义类型进行写入和截取操作。如果对一个错误的类型进行操作，将引发 CORBA::BAD_OPERATION 异常。

插入操作把 any 对象设置为特定的值。如果有必要，还可以通过 type()方法设置 any 对象的类型。如果在设置值以前就试图读取它的值，将导致出现 CORBA::BAD_OPERATION 异常。

```
package org.omg.CORBA;
abstract public class Any{
    abstract public org.omg.CORBA.TypeCode type();
    abstract public void type(org.omg.CORBA.TypeCode);
    // 当类型码与值不一致时，在从流中读取或写入的过程中会出现异常
    abstract public void read_value( org.omg.CORBA.portable.InputStream,
    org.omg.CORBA.TypeCode)throws org.omg.CORBA.MARSHAL;

    abstract public void
            write_value(org.omg.CORBA.portable.OutputStream);
    abstract public org.omg.CORBA.portable.OutputStream
            create_output_stream();
    abstract public org.omg.CORBA.portable.InputStream
            create_input_stream();

    //插入和截取每个原始类型
    abstract public short extract_short()
        throws org.omg.CORBA.BAD_OPERATION;
    abstract public void insert_short(short);
    ...
    //其他基本类型与 short 相似，不再列出

    //插入和截取类型码
    abstract public org.omg.CORBA.TypeCode extract_TypeCode(
            org.omg.CORBA.TypeCodeHolder)
            throws org.omg.CORBA.BAD_OPERATION;
    abstract public void insert_TypeCode(
            org.omg.CORBA.TypeCodeHolder);

        插入和截取 principal 对象值
        abstract public org.omg.CORBA.Principal extract_Principal(
                org.omg.CORBA.PrincipalHolder)
```

```
            throws org.omg.CORBA.BAD_OPERATION;
        abstract public void insert_Principal(
            org.omg.CORBA.PrincipalHolder);
        //插入和截取非原始IDL类型
        abstract public org.omg.CORBA.Streamable extract_Streamable(
            org.omg.CORBA.portable.StreamableHolder)
            throws org.omg.CORBA.BAD_OPERATION;
        abstract public void insert_Streamable(
            org.omg.CORBA.portable.StreamableHolder);
```

6.3.2.15 嵌套类型映射

IDL语言允许类型定义嵌套在接口中，而Java不允许这样做。所以，当把那些在接口域范围内的IDL类映射为Java类时，要把它们放在一个特殊的“域”包内。也就是说，包含这些类型定义的IDL接口将生成一个域包，以包含这些映射后的Java类定义。域包的名字是在IDL类型名字后加Package而构成的。

```
//IDL语言描述
module Example{
    interface Foo{
        exception e1{};
        };
    };
```

将被映射为：

```
//产生的Java代码
package Example.FooPackage;
public class e1 extends org.omg.CORBA.UserException{…}
```

6.3.2.16 typedef类型映射

由于Java中不支持IDL中的类型定义typedef，一般把IDL文件中使用typedef定义的类型在映射后直接设置为基本类型或用户定义类型。

```
//IDL语言描述
struct EmpName{
String firstName;
String lastName;
};
typedef EmpName EmpRec;
```

将被映射为：

```
//生成的Java代码
//对EmName结构类型的映射和EmpRec助手类映射
```

```
final public class EmpName{
…
    }
public class EmpRecHelper{
…
    }
```

6.3.3 IDL/Java 编译器

有了IDL/Java映射的知识作为基础，就可以进一步讨论如何实现一个具有这样功能的编译器了。

IDL/Java编译器随着实现的不同而有所不同。在这里仅介绍我们实现的IDL/Java编译器的构成和功能。

CORBA模型不仅能解决异种平台的应用集成问题，而且也能解决异种语言应用的集成问题。编译器的功能如下：

- 生成界面类；
- 生成客户方的Stub文件；
- 生成服务器方的Skeleton文件；
- 生成用于传递参数的Holder文件；
- 生成辅助类的Helper文件。

不同的平台环境和应用开发语言将对应不同的IDL语言编译器，因此开发IDL编译器就成为分布应用开发的一个重点。IDL编译器使客户对象完全独立于具体实现对象所在的位置、使用的语言及其他与对象界面无关的方面。所以说，IDL及其编译器是基于CORBA标准的分布计算环境中最基本的开发工具。

1) IDL编译器在分布式开发中的作用

IDL编译器在分布式开发中的作用如图6.10所示。

2)IDL编译器的开发模型

分布对象计算环境要支持异种平台和异种语言间的应用集成，从而导致IDL编译器具有以下特点：

(1) 分布计算环境不同，IDL语言的编译器不同。

(2) 应用开发使用的编程语言不同，IDL语言的编译器不同。

针对以上的特点，我们提出了输入处理、中间信息表示、目标代码输出三个层次的编译器开发模型，如图6.11所示。

在输入处理中，主要对输入的IDL文件进行词法和语法分析，它为中间信息表示层和目标代码输出层屏蔽了输入文件的词法和语法规则的差别。

中间信息表示层为输入层和目标代码输出层之间提供了通信的机制。利用中间信息表示层，定义并存储了IDL文件的所有语法单位。而目标代码输出层中，则通过这存储信息来得到语法树。

在目标代码输出层中，需要从中间信息表示层中获取信息，生成目标代码。

通过这种分层机制，扩展不同语言的IDL编译器只需在目标代码输出层中作相应的修

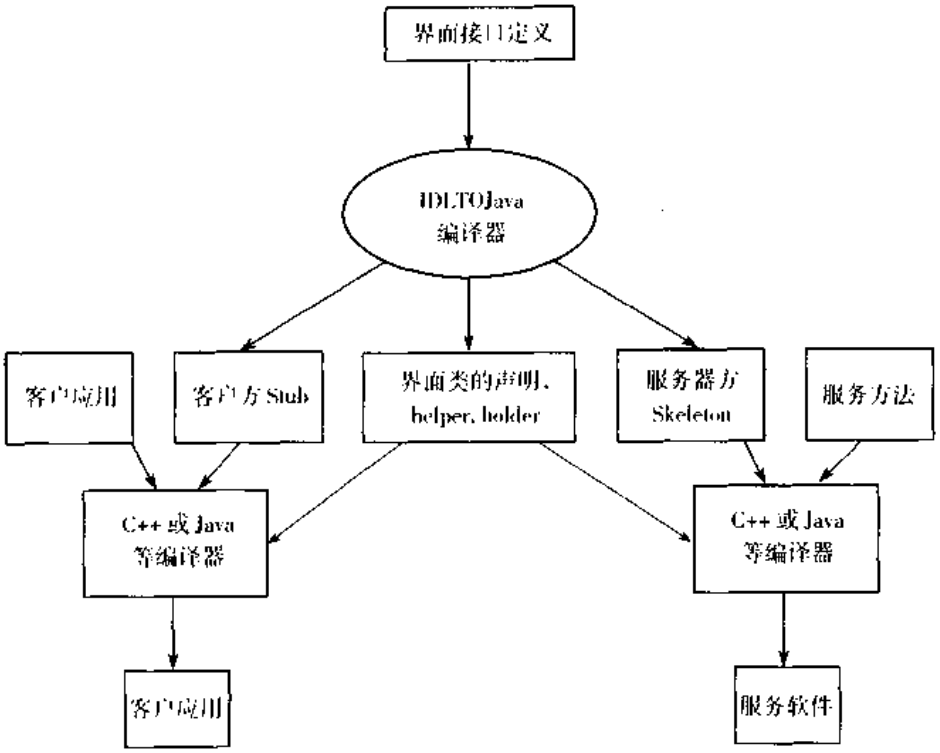

图 6.10　IDL 编译器在分布式开发中的作用

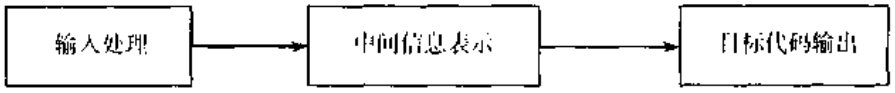

图 6.11　IDL 编译器的开发模型

改即可。

3) 编译器的有关技术问题

(1) 为每一个语法单元建立了一个类，用于封装关于本类的转换信息及转换方法。其中大体包括：module 类、interface 类、exception 类、operation 类、union 类、struct 类、attribute 类、const 类、typedef 类、base 类。

interface 类的基本内容如下：

```
int interfaceName;
Vector interfBody;
InterfaceJava(String na, String np, Vector bas1);
InterfaceHolder(String na);
InterfaceHelper(String na, String np);
```

其他的类与 interface 类大体相同。在此不一一列出。

对于它的扫描部分，完全是按照语法进行的，每个类对应一个函数，它们被封装在gen 类中。

(2) 对IDL源文件的串化及编码，是在第一层处理层进行的，对关键字、数据类型、方法、接口、模块等一一进行编码。在第一、第二阶段处理之后，将生成一棵含有语法单

元的语法树，结构如下：

```
Object node ;
Tree father ;
Tree next;
Tree peer;
```

与它同时产生的还有token向量，token装有处理过的类型和标识符。这一工作完成后将进入第三阶段。

(3) 第三阶段的主要任务是根据语法树以及token向量生成目标文件。我们对语法树进行next优先遍历。通过遍历，我们调用每个语法单元的生成目标代码函数，对每个节点做解码和目标文件的生成工作，最终完成IDL/Java的转换。

在对节点处理时，如果节点为容器类，有可能对它包含的子类先做一一处理再处理它本身。运用这一算法对于IDL的每个语法单元都可以加上描述并最终生成目标文件。

本方案的扩充性很好，在将来如果要生成除上述五种文件外的目标文件时，只要修改相应类的处理函数部分；如果IDL语法发生变化只需修改扫描部分的部分函数。而对于其他语言的转换，如C++,C，只要对生成目标代码部分做修改，即可完成。

6.4 CORBA，WWW和Java

三种流行的Web对象计算体系有ActiveX方案、Java RMI方案和CORBA方案。用CORBA来开发基于WEB的应用有许多优点：支持已有的代码，可以升级，广泛的支持平台，端到端的安全性，与开发语言无关的独立性，与开发者无关的独立性和与操作系统无关的独立性。Java语言作为一门新兴的网络上的世界语，在Web应用开发中占有举足轻重的地位，因此，Java迅速成为CORBA在WWW上的切入点。CORBA与Java有相似性和互补性。

由于CORBA与Web相结合所带来的巨大优势，出现了很多种CORBA的Web技术。

1) CORBA与Java的相似性

CORBA和Java都采用面向对象技术，因此，可以很容易地用Java语言开发CORBA应用，或将Java应用以及JavaBeans对象集成到CORBA应用环境中；CORBA和Java都适用于开发分布式应用，所不同的是：CORBA偏重于通用的分布式应用开发，而Java注重于WWW环境中的分布式应用开发。

2) CORBA与Java的互补性

(1) Java可以在以下方面为CORBA带来好处：

- WWW应用开发的能力。利用Java在Web应用开发中牢不可破的地位，CORBA可以轻松进入WWW领域，从而将CORBA的使用范围进一步扩大。
- 可移植性。Java语言强有力的可移植性使得Java-ORB本身以及在这之上开发的CORBA应用都具有很好的平台无关性，大大方便了CORBA应用的发布和使用。

(2) 采用了CORBA体系结构以后，Java程序员也获益匪浅：

- 分布系统开发能力。CORBA的引入，使Java程序员可以无需考虑网络编程的细节，

便捷地开发真正的异构环境下的分布式应用，大大提高了Java以及Java应用的广泛性。

● IDL的接口描述能力。Java RMI并没有一种专门的方式来描述对象的接口。CORBA/IDL可以满足Java程序员的这种需求，提供了为大型应用系统分析和建模的手段，从而加快系统的设计和开发过程。

● 访问CORBA对象服务。在CORBA系统中开发的Java应用，可以享用CORBA所提供的各种对象服务，包括名录服务、事件服务、事务服务、安全服务等。这些对象服务为Java程序员的开发提供了相当大的便利。

Java_ORB是指完全用Java语言开发的CORBA/ORB系统。它不仅能像C或C++ ORB一样开发分布式的Java应用(Java Applet)，更重要的是，它能够开发基于WWW的CORBA应用，赋予Java Applet更强大的功能。

6.5 采用Java语言在Web上构建CORBA应用

6.5.1 开发的一般过程

使用Java语言进行Web上的CORBA应用开发一般过程如下：

(1) 对所要实现的系统进行分析和对象建模，用IDL语言描述系统中各个对象的属性和对外接口。

(2) 考虑客户方采用的调用策略　如果采用静态调用，需将该系统的IDL描述文件通过IDL/Java编译器进行编译，生成相应的存根和构架文件；如果用动态调用，只需将IDL文件载入到接口库中。

(3) 使用Java语言实现系统中的各个对象。

(4) 编写服务器方主程序，其功能是创建服务方对象实现实例，并向Java-ORB注册，同时等待接收请求。

(5) 编写客户方Applet，其作用是获取服务器方初始对象引用，与用户进行交互，并根据用户指令向服务器方对象实现实例发出请求，并将结果返回给客户。

(6) 将客户方Applet和服务器方主程序分别与存根和构架文件等联编，并将该Applet嵌入到一个HTML页面中。

至此，用户通过浏览器浏览该页面，就可以调用服务器方对象实现的操作了。

6.5.2 实例：基于CORBA的信息集成

CORBA提供独立于客户端和数据库端的多层数据库访问：

(1) 由于采用CORBA体系结构，客户端可以采用任何CORBA支持的语言，如客户端可以采用Java语言，也可以采用C语言或C++语言等；

(2)由于是多层数据库访问方式，服务端和数据库之间可以采用与数据库管理系统相应的驱动程序实现数据库访问，不同的数据库只需要更换不同的驱动程序，不需要改写服务端代码；

(3)客户端不必关心服务端采用什么数据库驱动程序、甚至不必关心后台是什么数据

库，它只需要装入服务端提供的基于CORBA的驱动程序，按照和本地JDBC编程模式一样的方法编写用户代码即可。

利用CORBA 和Java JDBC构造分布式数据库应用：中间层为基于CORBA的应用服务器，中间层调用后台数据库可采用JDBC驱动程序，客户端调用应用服务器提供给用户的接口实现对数据库的访问。其实现原理如图6.12所示。

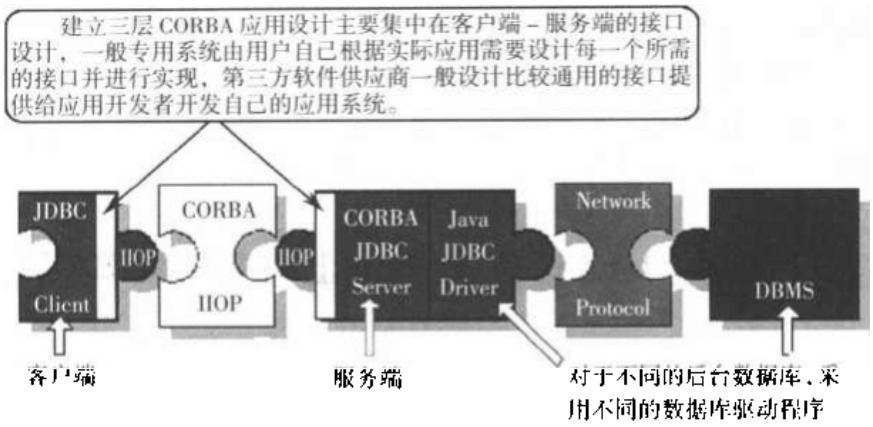

图6.12　基于三层CORBA应用的信息集成原理

JBuilder 2.0提供了Java语言的CORBA 开发支持，利用Jbuilder及其所带的 VisiBroker开发一个分布式的、基于对象的数据库查询应用程序，VisiBroker for Java是一个开发工具，可以用来建立、管理和开发分布式的跨多种平台的、开放的、易用的、可交互的Java 应用程序。这里介绍一个开发可以访问 MS SQL Server 数据库的客户/服务器模式的应用。

利用 VisiBroker 开发 CORBA 应用程序的过程如图6.13所示。

1) 定义 CORBA Objects 和编译 IDL 代码

文件student.idl定义了student模块，其中包含的QueryScore接口提供了一个连接数据库并返回查询结果的方法 QuSc（）：

```
module student {
  interface QueryScore {
    long QuSc(in String name, in String Subject);
  };
};
```

对于该接口利用 VisiBroker IDL 编译器（idl2java）编译产生一个 Java 接口 QueryScore.java 和其他必须的 Java 文件：_st_QueryScore.java 是客户端的 stub 程序，_QueryScoreImplBase.java 是一个服务端的框架（skeleton）程序，QueryScoreHolder.java 为传递参数提供支持，QueryScoreHelper.java 定义了一些辅助的功能。

2) CORBA 接口的实现

对象的实现类必须结合 IDL 编译器产生的框架类来完成。为实现 IDL 定义的 Query_Score 接口，QueryScoreImpl.java 类需要继承由 IDL 编译器产生的框架 _QueryScoreImplBase 类。QuSc() 方法通过 Java JDBC API 实现连接数据库、返回查询结果。

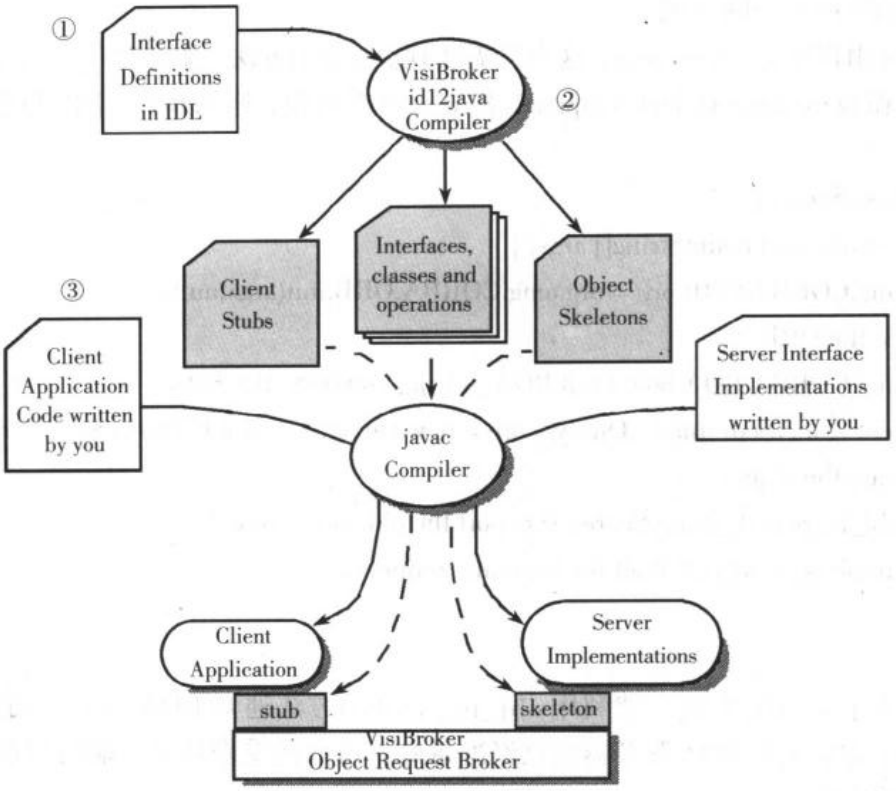

图 6.13　利用 VisiBroker 开发 CORBA 应用程序的过程

```
public class QueryScoreImpl extends student._QueryScoreImplBase {
public QueryScoreImpl(String name) {
        super(name);
}
public long QuSc(String name, string Subject) {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
// load JDBC driver
     java.sql.Connection con = DriverManager.getConnection("jdbc:odbc:SQLServer
db1""jsj"123');      //create connect to database:
   Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery(sql); //execute SQL query:
     String s0 = rs.getString("id");. . . . . //handle result:
     . . . . . //convert code for unicode and GB
    return(QueryResults); //return results:
   }
}
```

需要注意的是unicode码和GB码问题：IDL字符串类型被映射到Java 的java.lang.String 类型，java.lang.Strings 包含的是 Unicode 编码的字符，它比 IDL 字符宽，为了正确显示汉字，在客户端和服务器端都要进行 unicode 码和 GB 码的转换。

3) 服务端应用程序的实现

服务端应用程序类Server.java，这里只需实现其main()方法，服务程序负责下列任务：初始化ORB 和BOA(Basic Object Adapter)，建立服务端对象，通知BOA 对象可用以及准备接受请求。

```
public class Server {
   public static void main(String[] args) {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
//Initialize the ORB
    org.omg.CORBA.BOA boa = orb.BOA_init();// Initialize the BOA
     student.QueryScoreImpl _QueryScore = new QueryScoreImpl("QueryScoreServer");
     //Create the object
    boa.obj_is_ready(_QueryScore); // export the object reference.
     boa.impl_is_ready();// Wait for incoming requests:
   }
}
```

一旦建立了对象的实现，就利用obj_is_ready()方法通知BOA 该对象可用，利用impl_is_ready()方法通知BOA 服务器端已经准备好，可以接受来自客户端的请求。

4) 客户端实现

客户端的实现步骤为：初始化CORBA 环境即初始化ORB，查找定位到QueryScore 对象，获得想要调用其操作的对象的参考，然后调用其操作并处理得到的结果，这里的调用为QuSc() 方法，用来连接数据库，取得查询结果然后进行显示。

```
public class ClientApplet extends java.applet.Applet {
    public void init() {
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);
// Initialize the ORB
      QueryScore = student.QueryScoreHelper.bind(orb, " QueryScoreServer");
// Locate a QueryScore
     rs=_QueryScore.QuSc("select * from xsda");// connects database and get the query results
      . . . . . //convert code for unicode and GB
      }
      public void print(Graphics g) {
      g.drawString(. . . . .) ;//prints results
   }
   }
}
```

将该Applet 类锚接嵌入到ClientApplet.html：

```
<applet code=student.ClientApplet.class width=600 height=300>.
```

5) 编译项目

在菜单中选择Build|Make Project student.jpr 编译项目文件。

6) 程序的运行

● 运行程序前要建立好工作环境：在Windows NT4.0上运行MS SQL Server，正确安装JBuilder 2.0，设置好程序中用到的ODBC数据源，设置好必要的Java环境。

● 启动Smart Agent(OsAgent): Samrt Agent提供了对象查找服务。

● 启动服务器端的应用程序Server.java。

● 启动GateKeeper（默认端口号为15000），GateKeeper使得VisiBroker的Applet可以穿过大型网络与对象服务端通信，GateKeeper也可以作为Web服务器使用。

● 运行客户端程序。

若使用appletviewer，命令行下键入：

prompt>appletviewer http:// <server host name>:15000/clientapplet.html

若使用支持Java 1.1以上版本的浏览器，在地址栏输入：

http:// <server host name>:15000/clientapplet.html 。

由此可见，应用开发者只要编写客户方和服务器方的主驱动程序和应用程序本身的代码，涉及网络通信，表示转换，字节顺序，消息编码和解码，消息发送等等大量的工作均由ORB和IDL编译产生的代码完成。这些能大大简化分布式应用程序的开发过程，提高分布式应用的开发效率。随着CORBA的不断完善，它必将在信息集成中发挥更大的作用。

6.6 移植JDBC API接口到CORBA体系结构

利用Jbuilder及其所带的VisiBroker，Borland Jbuild 2.0提供了Java语言的CORBA开发支持。VisiBroker for Java是一个CORBA ORB开发工具，可以用来建立、管理和开发分布式的跨多种平台的、开放的、易用的、可交互Java应用程序，并且已经被Netscape公司集成到其WWW浏览器Communicator中。

VisiBroker提供的java2iiop编译器支持Java/IDL映射，用于从Java语言编译产生OMG IDL，使得开发人员可以用Java语言替代IDL语言定义CORBA接口，java2iiop编译器可以生成兼容iiop的客户端Stub和服务端的Skeletons。另外，VisiBroker支持对象按值传送。

VisiBroker提供的java2idl编译器转换Java代码到IDL语言，使你可以根据需要把客户端的Stubs映射到其他的语言。移植步骤：

(1) 定义服务器端的基于CORBA的JDBC接口；

(2) 编写上述服务器端的基于CORBA的JDBC接口的Java类实现；

(3) 编写客户端实现JDBC接口的类，其中的方法实现是调用服务器端相应的方法；

(4) 编写服务器端的服务守护进程；

(5) 根据实际应用编写客户端应用程序。

6.6.1 定义服务器端基于CORBA的JDBC接口

首先定义服务器端基于CORBA的JDBC接口，以下列出了几个主要接口的定义：

1) Driver接口：CJDriverInterface

```
package CJDriver;
import java.sql.*;
import org.omg.CORBA.*;
public interface CJDriverInterface extends org.omg.CORBA.Object {
    CJConnectionInterface connect(String url, java.util.Properties info) throws
        org.omg.CORBA.SystemException;
    .....
    }
```

2) Connection 接口：CJConnectionInterface

```
import org.omg.CORBA.*;
public interface CJConnectionInterface extends org.omg.CORBA.Object {
CJStatementInterface createStatement() throws org.omg.CORBA.SystemException;
CJPreparedStatementInterface prepareStatement(String sql) throws org.omg.CORBA.
SystemException;
    CJCallableStatementInterface prepareCall(String sql) throws org.omg.CORBA.SystemException;
.....
}
```

3) Statement 接口：CJStatementInterface

```
package CJDriver;
import java.sql.*;
import org.omg.CORBA.*;
public interface CJStatementInterface extends org.omg.CORBA.Object {
CJResultSetInterface executeQuery(String sql)throws org.omg.CORBA.SystemException;
int executeUpdate(String sql)throws org.omg.CORBA.SystemException;
.....
}
```

4) ResultSet接口：CJResultSetInterface

```
package CJDriver;
import java.lang.*;
import java.math.BigDecimal;
import java.sql.*;
import org.omg.CORBA.*;
public interface CJResultSetInterface extends org.omg.CORBA.Object {
    boolean next()        throws org.omg.CORBA.SystemException;
    ......
public java.lang.Object getObject_str(String columnName)throws org.omg.CORBA.
SystemException;
}
```

以上所有接口定义都继承了org.omg.CORBA.Object，这是向CORBA移植的基础，由此才可以生成客户端Stub和服务端的Skeletons，从而获得ORB的框架支持。其他接口类似

地进行定义。

用java2iiop编译上述接口，生成兼容iiop的客户端Stub和服务端的Skeletons以及一系列辅助类，生成的类用于下一步被服务端和客户端实现来继承和利用。

6.6.2 编写驱动程序的服务器端接口对象实现

服务器端的接口实现需要继承由java2iiop编译器编译生成的服务端的Skeletons，以下编写的代码显示了上述定义的接口中 CJDriverInterface 的实现：

```
package CJDriver;
import java.sql.*;
public class CJDriverServer extends CJDriver._CJDriverInterfaceImplBase {
   public CJDriverServer(String name) {
   super(name);
   }
   public CJConnectionInterface connect(String url, java.util.Properties info)
   throws org.omg.CORBA.SystemException {
     try {
       java.sql.Driver jdbcDriver;
       if((jdbcDriver = DriverManager.getDriver(url)) == null) {
         throw new org.omg.CORBA.TRANSIENT("CJDriverServer::connect: No suitable Driver");
       }
       return new CJConnectionServer(jdbcDriver.connect(url, info));
     } catch(Exception e) {
       throw new org.omg.CORBA.TRANSIENT("CJDriverServer::connect: "+e);
       }
   }

.....
}
```

该接口实现继承了 java2iiop 编译器编译生成的服务端 Skeleton 框架 CJDriver._CJDriverInterfaceImplBase。

其中最重要的 Connect 方法用于实现数据库的连接，由程序代码中（return new CJConnectionServer(jdbcDriver.connect(url, info));）可以看出，它生成了服务端的连接对象（CJConnectionServer）的实例，该实例返回一个真正的到数据库的连接。其他方法功能显而易见，且实现简单。

其他服务器端接口(共 8 个)实现类似，这里不再介绍。

6.6.3 编写驱动程序的客户端接口实现

下面编写驱动程序客户端类CJDriver的实现，并从分析该类入手，追踪系统接口之间

的交互过程，进一步阐述整个程序包的实现原理：

```
package CJDriver;
import java.sql.*;
import java.net.InetAddress;
public class CJDriver implements java.sql.Driver {
    static {
      try {
        // register the driver with the JDBC DriverManager
        DriverManager.registerDriver(new CJDriver()); // 向 DriverManager 注册
        } catch (Exception e) {
        e.printStackTrace();
        }
    }
    public CJDriver(String corbaAddr) throws Exception {
      throw new java.sql.SQLException("Deprecated constructor for CJDriver");
    }
    public CJDriver() throws Exception {
    }
    private static int corba_ADDRESS = 0;
    private static int JDBC_URL = 1;
    private String[] splitURL(String url) {
      /* The url has the format:
.....
    }

      public java.sql.Connection connect(String url, java.util.Properties info)
      throws SQLException {
        if(!acceptsURL(url)) throw new java.sql.SQLException("Unsupported URL");
        try {
          String split[] = splitURL(url);
          return new CJConnection(split[corba_ADDRESS], split[JDBC_URL], info);
        } catch(Exception e) {
          throw new java.sql.SQLException(e.getMessage());
        }
      }
    public boolean acceptsURL(String url) throws SQLException {
      return url.startsWith("jdbc:corba:");
    }
.....
```

};

由程序代码中可以看到：静态构造函数中自动向DriverManager注册自身，这就是当建立驱动程序实例时自动注册的代码实现。其中还建立了一个用于分析URL的函数splitURL (String url)，解析 URL 各部分保存到一个数组中供其他方法使用。注意到 connect 方法以 URL 为参数实例化客户端的连接接口 CJConnection，具体语句：

new CJConnection(split[corba_ADDRESS], split[JDBC_URL], info);

这是建立到数据库连接的开始，为了追踪连接的过程，沿着该语句调用路线进一步进行分析。由于该句是 CJConnection 对象的实例化，来看一下对象 CJConnection 类的实现代码：

```
package CJDriver;
import java.sql.*;
public class CJConnection implements java.sql.Connection {
  CJConnectionInterface corbaConnection_;
  public CJConnection(String corbaAddr, String url, java.util.Properties info)
  throws Exception {
    // Initialize the ORB (using the Applet).
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
     CJDriver.CJDriverInterface theDriver =CJDriver.CJDriverInterfaceHelper.bind(orb,
"CorbaJdbcServer");
    corbaConnection_ = theDriver.connect(url, info);
}
.....// 其他代码略去
};
```

该段代码比较长，注意其中的构造函数，这是建立到服务器端的关键语句段落。

首先初始化 ORB 对象：

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init ();
```

然后根据服务器端的服务对象名称查找服务器端的驱动程序对象(服务器端的驱动程序对象已经按照名字 CorbaJdbcServer 连接到服务器指定的端口上)，并建立该对象的实例 (远程服务器端的驱动程序对象实例)：

```
CJDriver.CJDriverInterface theDriver =CJDriver.CJDriverInterfaceHelper.bind(orb,
"CorbaJdbcServer");
```

紧接着语句是执行远程服务器端的驱动程序的连接方法connect(url, info)，返回服务器端到数据库的实际连接：

```
CJConnectionInterface corbaConnection = theDriver.connect(url, info);
```

继续进行跟踪，进入了对服务器端驱动程序的连接方法的调用，下面是服务器端驱动程序实现代码：

```
package CJDriver;
import java.sql.*;
//import java.net.InetAddress;
```

```
public class CJDriverServer extends CJDriver._CJDriverInterfaceImplBase {
   public CJDriverServer(String name) {
   super(name);
   }
   public CJConnectionInterface connect(String url, java.util.Properties info)
   throws org.omg.CORBA.SystemException {
     try {
       java.sql.Driver jdbcDriver;
       if((jdbcDriver = DriverManager.getDriver(url)) == null) {
         throw new org.omg.CORBA.TRANSIENT("CJDriverServer::connect: No suitable Driver");
       }
       return new CJConnectionServer(jdbcDriver.connect(url, info));
     } catch(Exception e) {
       throw new org.omg.CORBA.TRANSIENT("CJDriverServer::connect: "+e);
     }
   }
.....//其他代码略去
}
```

该段代码中服务器端驱动程序的 connect(String url, java.util.Properties info)方法建立一个服务器端的连接对象实例：

```
new CJConnectionServer(jdbcDriver.connect(url, info));
```

这样跟踪到了服务器端的一个重要类 CJConnectionServer，该类代码如下：

```
package CJDriver;
import java.sql.*;
import java.io.*;
import org.omg.CORBA.*;
public class CJConnectionServer extends _CJConnectionInterfaceImplBase {
   java.sql.Connection jdbcConnection_;
   public CJConnectionServer(java.sql.Connection c)
   throws org.omg.CORBA.TRANSIENT {
     super();
     jdbcConnection_ = c;
   }
   public CJStatementInterface createStatement()
   throws org.omg.CORBA.TRANSIENT {
     try {
       return new CJStatementServer(jdbcConnection_.createStatement());
     } catch(java.sql.SQLException e) {
       throw new org.omg.CORBA.TRANSIENT(
```

```
            "CJConnectionServer::createStatement()"+e);
        }
    }
    public CJPreparedStatementInterface prepareStatement(String sql)
    throws org.omg.CORBA.TRANSIENT {
        try {
            return new CJPreparedStatementServer(
                jdbcConnection_.prepareStatement(sql));
        } catch(java.sql.SQLException e) {
            throw new org.omg.CORBA.TRANSIENT(
                "CJConnectionServer::prepareStatement()"+e);
        }
    }
    public CJCallableStatementInterface prepareCall(String sql)
    throws org.omg.CORBA.TRANSIENT {
        try {
            return new CJCallableStatementServer(jdbcConnection_.prepareCall(sql));
        } catch(java.sql.SQLException e) {
            throw new org.omg.CORBA.TRANSIENT(
                "CJConnectionServer::prepareCall()"+e);
        }
    }
.....
};
```

该类建立了一个到数据库的真正连接，客户端得到的就是该连接对象的实例，客户端所进行数据库操作都是以该连接为基础的。

该类同时提供了建立在该连接基础上的数据库操纵方法，如：

public CJStatementInterface createStatement()
public CJPreparedStatementInterface prepareStatement(String sql)
public CJCallableStatementInterface prepareCall(String sql)等等。

通过对客户端如何借助服务器端建立到数据库的连接过程的跟踪分析，我们进一步对该驱动程序的实现原理进行了深入阐述，其他的对数据库的操作过程如执行查询、返回结果的处理等和建立连接过程原理相同，在此就不一一阐述了。

6.6.4 程序设计中的相关问题

以下是程序编制过程中的一些重要问题及其解决办法。

对象按值传送（Object by Value）：本接口包中的程序需要以对象作为参数，所以需要

支持对象按值传送（Object by Value）的ORB，visibroker支持对象按值传送。

方法重载：由于java2iiop编译器不支持方法重载，所以为解决方法重载问题，修改了服务端接口中方法名称，客户端根据方法调用参数的不同选择调用不同名称的服务端方法。现举例子说明：

```
interface CJCallableStatementInterface
        void registerOutParameter__3(int parameterIndex, int sqlType, int scale) throws
org.omg.CORBA.TRANSIENT;
```

由于interface CJCallableStatementInterface已经存在void registerOutParameter(.....)方法，所以本方法改名为void registerOutParameter__3(.....)，客户端调用时如果是3个参数的则调用void registerOutParameter__3(.....)方法。其他类推。

汉字显示技术：由于Java使用Unicode编码，而CORBA采用ASCII码，所以对汉字的处理存在不一致性，如果不进行转换，无法正确实现处理汉字。采取在服务端程序中主动转换Unicode到ASCII传送到客户端后，客户端再进行逆转换，从而解决了汉字显示问题。

服务器端连接的对象名:服务器端连接的对象名（硬编码）定义为CorbaJdbcServer。

6.6.5 驱动程序服务器端的守护进程的设计及实现

驱动程序服务器端的守护进程用于初始化ORB，建立并注册服务对象，监听客户端的请求调用。

```
package CJDriver;
//import java.net.InetAddress;
public class CJServer {
  public static void main(String[] args) {
    System.out.println("Initializing ORB,BOA ang creating server object .....");
    // 初始化 ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // 初始化 BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // 建立服务对象(服务端驱动程序).
    CJDriver.CJDriverInterface _CJDriverServer = new CJDriverServer("CorbaJdbcServer");
    // 注册一个数据库驱动程序(这里注册 JDBC-ODBC 桥)
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
    } catch(Exception ex) {
      System.out.println("Can't find Database driver class: " + ex);
      return;
    }

    // 输出刚建立的对象
```

```
        boa.obj_is_ready(_CJDriverServer);
        System.out.println("The CorbaJdbcServer is ready.");
        // 等待进入的请求
        boa.impl_is_ready();
    }

};
```

6.6.6 利用该驱动程序开发企业实际应用系统

利用该驱动程序开发企业实际应用系统时，客户端应用程序的编写和标准JDBC应用程序编写具有一样的语法，而且不论后台是什么数据库，该程序都用本CORBA-JDBC驱动程序驱动（参见语句Class.forName("CJDriver.CJDriver").newInstance();），不用更改程序编码（后台数据库驱动程序在服务器端装入），对于熟悉JDBC编程者来讲，和编写标准的JDBC应用程序完全一样。完全可以根据用户实际应用编写应用代码。且如果用户以前用Java JDBC开发过应用，可以直接嵌入该驱动程序建立起三层应用，前端界面不必作任何修改，由于代码和有关JDBC数据库访问编程一样，在此就不重复论述了。

6.6.7 整个系统的设置及运行方法

1) 设置ODBC数据源

2) 关于服务端驱动程序和客户端数据库URL的硬编码(用户可以在使用时进行修改)

(1) 服务器端装入sun.java.JdbcOdbcDriver驱动程序 (也可以改为其他任何可用驱动程序)，参见CJServer.java；

(2) 客户端装入Corba-Jdbc驱动程序，用指定的URL建立连接；

客户端建立连接时传送数据库的URL到服务端驱动程序，服务端驱动程序用此URL调用服务端装入的驱动程序如sun.java.JdbcOdbcDriver，来建立到数据库的连接，以上两段连接一起形成真正的客户—服务端—数据库的连接；

3) 运行Application

(1) 启动Smart Agent: Tools--->Visibroker Smart Agent；

(2) 运行服务端程序CJServer.java，待屏幕提示:The CorbaJdbcServer is ready.表示运行成功；

(3) 运行客户端应用程序ClientApplication.java，将在DOS窗口中显示查询结果；

4) 运行Applet

(1) 启动Smart Agent: Tools--->Visibroker Smart Agent；

(2) 运行服务端程序CJServer.java，待屏幕提示:The CorbaJdbcServer is ready.表示运行成功；

(3) 启动一个DOS窗口，假设Jbuilder安装目录为F:\JB2。

运行Gatekeeper: 在F:\JB2\myclasses目录中运行start Gatekeeper，提供一个HTTP:15000端口的服务，默认主目录即为F:\JB2\myclasses；

拷贝 ClientApplit.htm 到 F:\JB2\myclasses;

该文件嵌入客户端 Applet 程序，在浏览器中运行客户端程序。

程序执行输出结果如图 6.14 所示。

图 6.14　程序运行输出结果

6.7　Java 与 CORBA 相结合的前景

用CORBA来开发基于WEB的应用有许多优点：它支持已有的代码，可以升级，广泛地支持平台，端到端的安全性，与开发语言无关的独立性，与开发者无关的独立性和与操作系统无关的独立性。CORBA的ORB在当前每一种主流操作系统上均有实现。除此之外，CORBA ORB可以访问多种语言实现的对象（包括C++、COBOL、Smalltalk和Java）。借助于IIOP，某一开发者（比如说Visigenic）开发的CORBA ORB能够获取、操作远程的由其他的开发者（比如说IONA）开发的对象。Java ORB允许客户端在没有安装任何特别软件的情况下实现Java客户端应用程序（Java ORB的类可与小应用程序一起动态下载，也可能与浏览器捆绑在一起）。

CORBA最大的优点是对已有代码的支持。对于想把已有的应用向intranet或Internet开放的开发人员来说，CORBA是一个很好的选择，因为它能在利用更新的像Java一类的技术的同时重用现有的代码。CORBA对已有代码的支持能力强。因为CORBA是一种独立于语言的技术，所以用支持CORBA的语言（Java,C++,Ada,SmallTalk,COBOL,等)写的任何一个客户端都能以一种独立于平台和语言的方式访问服务器的对象。

Remote Method Invocation 是最新的 JDK1.1 中的重要特色，是 Sun 的 Java-Only 解决方案，RMI 使得 Java 客户能够访问远地的服务对象。这听起来似乎十分类似于 CORBA，但两者并不一样。其关键在于服务器端的应用程序也必须用 Java 编写，且只能使用 JDK1.1 中提供的工具。你根本无法把过去编制的代码加到新程序中去，除此之外，RMI 还有许多其他缺陷。

与 CORBA 不同，RMI 没有服务这一概念。另外，根据 RMI 写出的 Java 服务器对象往往性能低，这个缺点源于 Java 虚拟机（Java CORBA 服务器比 RMI 服务器表现出更好的性能）。RMI 也不包括像 CORBA ORB 那样的对象激活功能。

实际上，RMI 及 Java 技术正在向 OMG 的标准靠拢，而不是背道而驰。Sun 已经宣布，Java 事务服务（JavaTransactionServeces）将建立在 OMG 的对象事务服务（ObjectTransactionServeces）上。该公司还曾发布其长远计划：使 RMI 对象可以通过 IIOP 互通信。

总而言之，RMI 对于用纯 Java 书写的小规模的应用程序来说，是一种可行方案。但 CORBA 提供了集成的基础，这种集成是指新开发的代码和已有对象的集成，同时允许将来加以扩展。在做出取此舍彼的选择之前，必须权衡上面的各种因素，并仔细审视每种技术的现状。

CORBA 和 RMI 都是被 OMG (Object Management Group) 支持的分布式对象标准。CORBA 是构建异构分布式系统的体系结构，支持不同语言编写的分布式构件，其对象接口使用 OMG 的接口定义语言（Interface Definition Language，IDL)定义，传输层可以使用不同的协议，如其标准协议 IIOP (Internet InterORB Protocol)。RMI 是分布式 Java 系统的 API 标准，使用自己专用的结构和传输层，其标准传输协议为 JRMP，但也可以使用 IIOP 作为传输层。使用 RMI 可以获得 Java 所具有的如分布式废料收集、对象可串行化等优点，其远程对象接口用纯 Java 代码就可被定义。标准的 CORBA 传递对象是按参考传送的，不支持对象的按值的传送，RMI 利用 Java 的对象可串行化可以实现对象的按值传送。

从上述 CORBA 和 RMI 两种方式实现分布式数据库应用可以看出，虽然两种方式协议不兼容，但协议模型相似，所以程序设计思路也极为相似。从语言中立、可伸缩性以及集成能力方面看，CORBA 优于 RMI，但由于 CORBA 被设计的语言无关性，从而也损失了特定语言所具有的优越性，相对讲 RMI 在方便快捷、易用性方面有独特优势。

对比两种方案，CORBA 使得面向对象的软件成员在分布异构环境中可重用、可移植、可互操作，它具有跨平台、语言中立、伸缩性、健壮性好等优点，适用于大型分布式系统开发，特别是开发集成已有旧系统方面更具实用价值。RMI 对于 Java 环境下的应用程序快速开发是一种好的选择，RMI 在 Java 环境下可以为我们提供一个轻松简洁、易实现的分布式应用环境，可以在许多领域特别是开发新系统方面得到很好应用。与 CORBA 规范相比，RMI 存在的主要问题是不像 CORBA 那样伸缩性好，例如 RMI 标准传输协议 JRMP 和当前的 CORBA/IIOP 协议不兼容，只能和其他的 Java RMI 对象通信，不能和其他非 Java 分布对象通信。不过随着技术的发展，RMI 已经可以采用 IIOP 作为传输协议了，而扩展的 CORBA 也可以支持对象按值传送了，RMI 和 CORBA 越走越近了。

参 考 文 献

1 Donald Doherty Michelle, M.Manning. Borland Jbuilder 2. Publishing House of Electronics Industry, 1999.
2 O'Toole, A.Making. Software Work together. Orbix Journal,May1998
3 James Glosling,Bill Joy,Guy Steele. The Java Language Specification. Addison-Wesley Publisher, 1996
4 Object Management Group,CORBA successful Stories. http://www.corba.arg/
5 Curtis D.Java,RMI and Corba. White Paper of Object Management Group,1996
6 张为民 等编著. Java语言及应用. 北京：清华大学出版社, 1997
7 （美）Patrick Naughton 著. Java 使用手册. 谢小兵，于春燕译. 北京：电子工业出版社，1996
8 （美）Gary Cornel 著. Java 核心. 扬秀军，丁兴农等译. 北京：科学出版社，1997
9 （美）Ashton Hobbs 著. 自学 JDBC 数据库编程. 北京：清华大学出版社
10 廖卫东 陈梅编著. Java 程序设计实用指南. 北京：机械工业出版社, 1996
11 王克宏编著. Java 语言 SQL 接口 -JDBC 编程技术. 北京：清华大学出版社,1997
12 王克宏编著. Java 语言 APPLET 编程技术. 北京：清华大学出版社, 1998
13 （美）Doug Lea. Java 并发程序设计. 严伟，陈志兰等译. 艾迪生维理出版有限公司,1998
14 (荷兰) Andrew S. Tanenbaum 著. 计算机网络. 第三版. 王小虎等译. 北京：清华大学出版社,1997
15 汪芸. CORBA 技术及其应用. 东南大学出版社，1998