§ 1.1　异常处理机制

1.1.1　异常处理的基本原则

按照程序错误类型、错误发现时刻及错误处理原则，可以将错误大致分为三类：违反语法规范的错误称为语法错，这可以在程序编译时发现；在程序语义上存在错误，则称为语义错，通常在程序运行时才能被发现；其它就是系统无法发现的逻辑错误，这需要特别的工具与方法才能处理。图 **Error! No text of specified style in document.**-1 描述了Java程序发现错误与异常的处理流程。

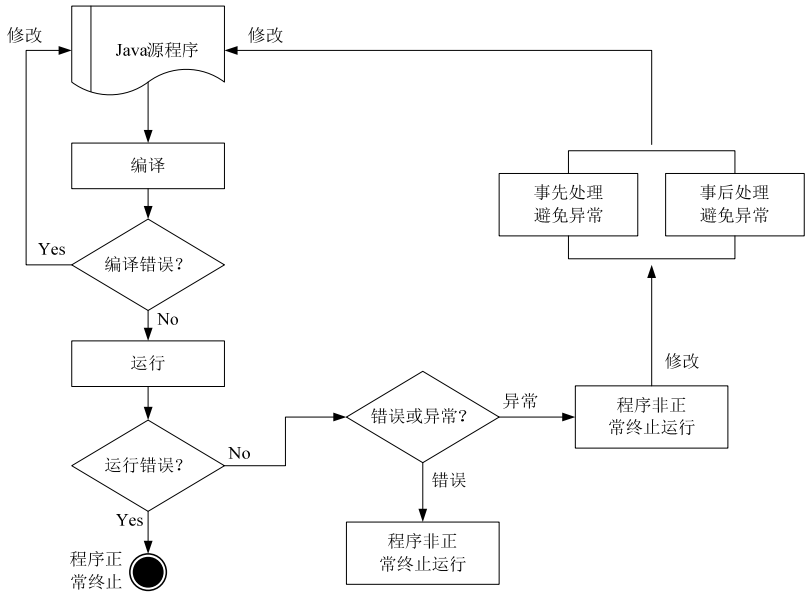

图 Error! No text of specified style in document.-1 Java 程序发现错误与异常示意图

【提示】错误（error）指程序运行时遇到的硬件或操作系统的错误，比如 OutOfMemoryError、ThreadDeath 等。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。异常（exception）指在硬件和操作系统正常时，程序遇到的运行错，程序本身可以处理的异常。

异常可以看做是程序运行过程中出现的错误。区别于面向过程语言错误处理方式（不进行范围检查、采用 if 语句进行事先判断以防止出现错误），面向对象语言异常处理的思想的优越之处体现在以下两方面：从语法上看，异常处理语句将程序正常代码与错误处理代码分开，使程序结构清晰，算法重点突出，可读性强；从运行效果看，异常处理语句使程序具有处理错误的能力。

Java把异常当作对象来处理，并定义一个基类java.lang.Throwable作为所有异常的超类。Throwable类所有异常和错误的超类，有两个子类Error和Exception，分别表示错误和异常。其中异常类Exception又分为运行时异常(RuntimeException)和非运行时异常。Error与运行时异常，也称之为不检查异常（Unchecked Exception）；Exception常称为需要检查异常（Checked Exception）。

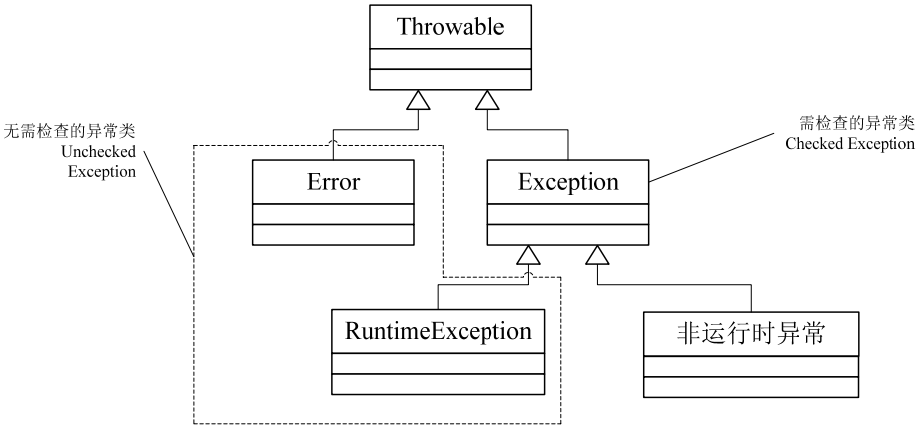

图 **Error! No text of specified style in document.**-2 Java 程序发现错误与异常示意图

【提示】错运行时异常都是 RuntimeException 类及其子类异常，如 NullPointerException、IndexOutOfBoundsException 等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理，但程序应该从逻辑角度尽可能避免这类异常的发生。非运行时异常是 RuntimeException 以外的异常，从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如 IOException、SQLException 等以及用户自定义的 Exception 异常，一般情况下不自定义检查异常。

1.1.2 Java 语言的异常捕获与处理机制

Java 采用将出错处理和正常代码分开的方法来实现异常处理。每当 Java 程序运行过程中发生一个可识别的运行错误时，即该错误有一个异常类与之相对应时，系统都会产生一个相应的该异常类的对象，即产生一个异常。一旦一个异常对象产生了，系统中就一定有相应的机制来处理它，确保不会产生死机、死循环或其他对操作系统的损害，从而保证了整个程序运行的安全性。Java 异常捕获与处理涉及到五个关键字，分别是：try、catch、finally、throw、throws。异常处理的完整语法如下所示：

```
try{
                                    //（尝试运行的）程序代码
 }catch(异常类型 异常的变量名){
     //异常处理代码
 }finally{
     //异常发生，方法返回之前，总是要执行的代码
 }
```

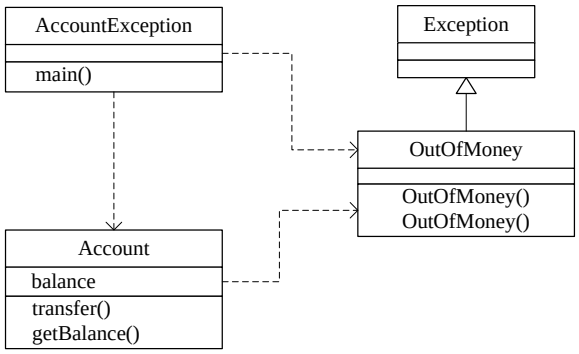

图 **Error! No text of specified style in document.**-3 AccountDemo 例程静态类图

程序 2-1 给出了一个捕获与处理异常的简单例子。

表 **Error! No text of specified style in document.**-1 捕获与处理异常例程

AccountDemo.java	捕获与处理异常	程序 **Error! No text of specified style in document.**-1

程序 **Error! No text of specified style in document.**-1 AccountDemo.java 程序例程

```
//捕获与处理异常例程
class Account {
        private double balance = 1000;
        public void transfer(double amount) throws OutOfMoney {   //在方法中声明抛出异常
                if (balance < amount)
                        //直接抛出异常
                        throw new OutOfMoney("[Balance:" + balance + " < Amount:" + amount + "]");
                balance = balance - amount;
        }
        public double getBalance() {
                return balance;
        }
}

class OutOfMoney extends Exception {
        public OutOfMoney() {                                          //给出该异常的一个缺省的字符串描述
                super("Your account have not enough money!");
        }
        public OutOfMoney(String msg) {   //允许使用者自行给出对异常的一个字符串描述
                super(msg);
        }
}

public class AccountException {
        public static void main(String[] args) {
                Account obj = new Account();
                double amount = 800;
                for (int count = 0; count < 3; count++) {
                        try {
                                obj.transfer(amount);
                                System.out.println("Transfer amount: " + amount + ", and then balance: "+obj.getBalance());
                        } catch (OutOfMoney exc) {
                                exc.printStackTrace();
                        } finally {
                                System.out.println("finally 语句块中的语句总是会执行！");
                        }
                        amount = amount - 300;
                }
        }
}
```

定义的类 OutOfMoney 的第一个构造方法给出该异常的一个缺省的字符串描述，而第二个构造方法允许使用者自行给出对异常的一个字符串描述。这两个构造方法都仅仅简单地调用超类 Exception 的构造方法。类 Account 定义方法 transfer()用于转账，它检测到余额不足时创建

并抛出异常类 OutOfMoney 的一个异常对象，由于该类型的异常是需要检查的异常，该方法又不使用 try...catch...语句对此异常进行捕获和处理，因此该方法要使用 throws 语句声明将抛出该类型的异常。

【注意】在类和方法声明抛出异常用 throws，直接抛出异常就要用到 throw。

类 AccountDemo 演示该异常的捕获与处理，在主方法 main()中，将可能抛出异常的方法调用 obj.transfer(amount)放在 try 语句块，其后跟 catch 语句块处理 try 语句块中的语句可能抛出的各种异常，每一种异常使用一个 catch 语句块，在保留字 catch 后面给出该块所能捕获的异常类型，这里只要捕获类型为 OutOfMoney 的异常，对该异常的处理只是打印创建所捕获的异常对象时的调用栈。

不管try 语句块的语句是否抛出异常，也不管catch 语句块是否捕获到异常，finally 语句块的语句总会执行，所以以上程序中finally 语句块中的打印内容总会被显示出来。

有关异常的转译与异常链，异常声明与方法重载，异常处理与继承机制等内容，大家可以参考《面向对象程序设计与Java语言》一书。

§ 1.2　基于 Socket API 开发的基本原理

1.2.1　Socket API 基本概念

Socket 是从电话通信中借用的一个术语，Socket的英文原义是“孔”或“插座”。

Socket API是TCP/IP网络的API。Socket API最早作为伯克利（Berkeley）UNIX操作系统的程序库，出现于20世纪80年代早期，用于提供IPC（IPC：Inter-Process Communication）通信。目前，所有主流操作系统都支持Socket API，在BSD、Linux等基于UNIX的系统中，Socket API都是操作系统内核的一部分。在MS-DOS、Windows等个人计算机操作系统也是以程序库的形式提供Socket API（其中在Windows系统中，Socket API被称作Winsock）。

JAVA语言在设计之初也考虑到网络编程，也将Socket API作为语言核心类的一部分。

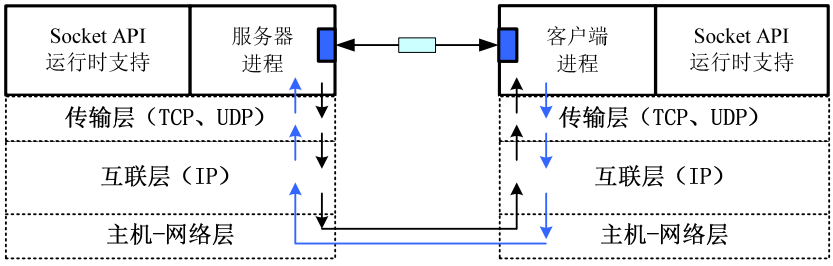

图 **Error! No text of specified style in document.**-4 Socket API 的概念模型

Socket API 的概念模型如 图 Error! **No text of specified style in document.**-4 所示。一个Socket代表了逻辑上的一条通信“通道”的一个方向的最端点。而实际上讲，Socket是物理网络地址和逻辑端口号的一个集合，而这个集合可以向另外一个位置的与他具有相同定义的Socket进行数据传输。

因为Socket是由机器地址和端口号来区分/识别的，那么在一个特定的计算机网络上，每一个Socket都是以此方式被唯一识别的。这就使得应用程序可以唯一地去

定位网络上的另外一个位置的Socket。 对于同一台机器上的两个Socket，他们是完全具备彼此间进行通信的可能的；在这种情况下，两个Socket具有相同的主机地址，但是他们拥有不同的端口号。

1.2.2 套接字 Socket API

套接字 Socket API 提供给用户一种处理通信的方法，使得相关进程可以存在于横跨网络的不同的工作站上。套接字类型有二种方式：

- 流式套接字，它提供进程之间的逻辑连接，并且支持可靠的数据交换。
- 数据报套接字，它是无连接的并且不可靠。

本教程提供的两个基于 Socket API 的实验案例，都要求基于流式套接字，所以本节重点讲解流式套接字的基本原理。具体数据报套接字的原理，读者可以参考《分布式计算》教材。

流式 Socket 所完成的通信是一种基于连接的通信，即在通信开始之前先由通信双方确认身份并建立一条专用的虚拟连接通道，然后它们通过这条通道传送数据信息进行通信，当通信结束时再将原先所建立的连接拆除（如 图 Error! **No text of specified style in document.**-5）。这个过程中，Server 端首先在某端口提供一个监听 Client 请求的监听服务并处于监听状态，当 Client 端向该 Server 的这个端口提出服务请求时，Server 端和 Client 端就建立了一个连接和一条传输数据的通道，当通信结束时，这个连接通道将被同时拆除。

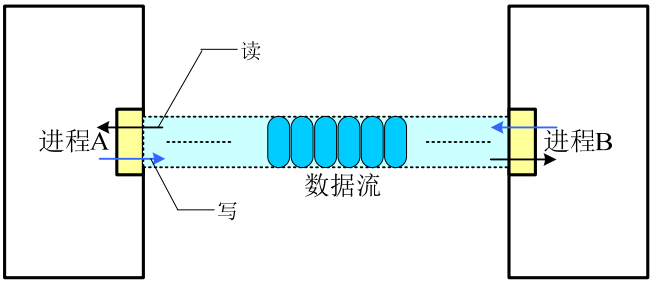

图 Error! No text of specified style in document.-5 流式 Socket 连接控制方式

针对这一流式 Socket 通信模式，java.net 程序包将基于 TCP 通信的 socket 封装为两个类：类 Socket 表达了一个用于建立 TCP 连接的 socket，该 socket 既可由客户程序使用，也可由服务程序使用；类 ServerSocket 则是一个服务端专门监听客户程序连接请求的 socket 的抽象，仅在服务程序中使用。

基于 socket 通信的客户程序首先通过指定主机（主机名或 InetAddress 的实例）和端口号构造一个 socket，然后调用 Socket 类的 getInputStream()和 getOutputStream()分别打开与该 socket 关联的输入流和输出流，依照服务程序约定的协议读取输入流或写入输出流，最后依次关闭输入 / 输出流和 socket。

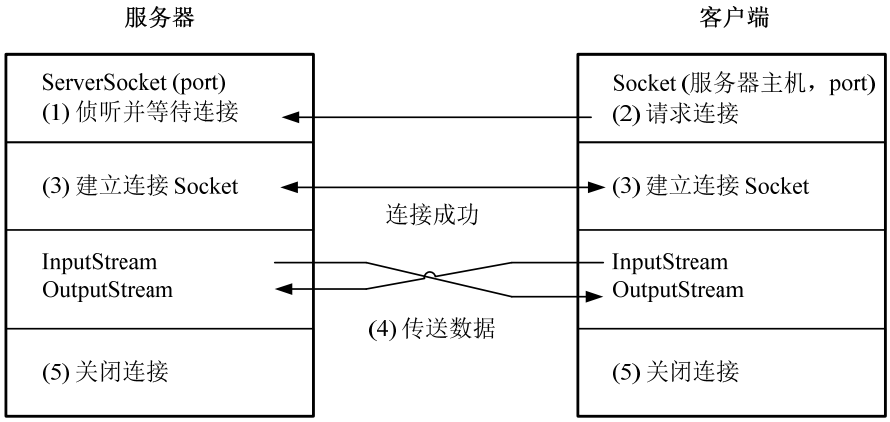

图 **Error! No text of specified style in document.**-6 流式 Socket 通信过程

基于 socket 通信的服务程序负责监听对外发布的端口号，该端口专用于处理客户程序的连接请求。因而服务程序首先通过指定监听的端口号创建一个 ServerSocket 实例，然后调用该实例的 accept()方法；调用 accept()方法会引起阻塞，直至有一个客户程序发送连接请求到服务程序所监听的端口，服务程序收到连接请求后将分配一个新端口号建立与客户程序的连接，并返回该连接的一个 socket。然后，服务程序可调用该 socket 的 getInputStream() 和 getOutputStream()方法获取与客户程序的连接相关联的输入流和输出流，并依照预先约定的协议读输入流或写输出流。完成所有通信后，服务程序必须依次关闭所有输入流和输出流、所有已建立连接的 socket、以及专用于监听的 socket。

以下程序展示了一个典型的基于 TCP 通信的单线程客户程序与服务器程序。从该例子程序不难看出，不同客户程序的编程复杂度主要体现在如何依照客户程序和服务程序双方约定的协议读／写并处理数据。

表 **Error! No text of specified style in document.**-2 基于 TCP 通信的单线程客户程序与服务器程序

MyClient.java	客户端程序	程序 **Error! No text of specified style in document.**-2
MyServer.java	单线程服务程序	程序 **Error! No text of specified style in document.**-3

程序 **Error! No text of specified style in document.**-2 客户端程序 MyClient.java

```
//简单的利用流套接字实现通信的客户端程序
import java.net.*;
import java.io.*;

public class MyClient{
        static Socket server;

        public static void main(String[] args)throws Exception{

                if (args.length != 2) {
```

```
                System.out.println("用法：MyClient <主机名> <端口号>");
                return;
        }

        //获取本地 ip 地址，访问在本地的服务程序，缺省端口是：1234
        Socket server = new Socket(args[0], Integer.parseInt(args[1]));;

        // 建立连接并打开相关联的输入流和输出流
        BufferedReader in=new BufferedReader(new InputStreamReader(server.getInputStream()));
        PrintWriter out=new PrintWriter(server.getOutputStream());
        BufferedReader wt=new BufferedReader(new InputStreamReader(System.in));

        // 将控制台输入的字符串发送给服务端，如果接收到"end"，则退出程序。
        while(true){
                String str=wt.readLine();
                out.println(str);
                out.flush();
                if(str.equals("end")){
                        System.out.println("通信已经终止");
                        break;
                        }
                System.out.println(in.readLine());
                }

        //关闭连接
        wt.close();
        out.close();
        in.close();
        server.close();
    }
}
```

程序 **Error! No text of specified style in document.**-3 展示了一个典型的基于 TCP 通信的服务程序。从该例子程序不难看出，不同服务程序的编程复杂度也是体现在如何依照预先约定的协议读／写数据和加工数据。当然，服务程序通常还须更多地考虑可伸缩性、安全性、可靠性等问题。

程序 Error! No text of specified style in document.-3 单线程服务程序 MyServer.java

```
//简单的利用流套接字实现通信的服务端程序
import java.io.*;
import java.net.*;

public class MyServer{
    public static void main(String[] args) throws IOException{
        if (args.length != 1) {
        System.out.println("用法：EchoServer <端口号>");
        return;
        }

        //创建一个 ServerSocket 实例，建立连接
        ServerSocket server = new ServerSocket(Integer.parseInt(args[0]));
        System.out.println("服务程序正在监听端口" + args[0]);
```

```
                    // 监听客户程序的连接请求
                    Socket client=server.accept();
                    BufferedReader in=new BufferedReader(new InputStreamReader(client.getInputStream()));
                    PrintWriter out=new PrintWriter(client.getOutputStream());

                    // 从客户端读取数据，并打印在屏幕上，如果接收到”End”，则退出程序。
                    String str;
                    System.out.println("客户端已经建立连接");
                    while((str = in.readLine()) != null) {
                              System.out.println(str);
                              System.out.println("收到请求:  " + str);
                              out.println("服务端已经收到请求:  " + str);
                              out.flush();
                              if(str.equals("end")) {
                                        System.out.println("通信已经终止");
                                        break;
                              }
                     }
          // 关闭连接
          out.close();
          in.close();
          client.close();
          }
}
```

1.2.3 网络协议的理解

本教材提供的 Socket API 实验案例都是基于流式套接字的应用层协议的开发，分别开发 HTTP 协议和 FTP 协议的应用。因此，理解网络协议的工作方式是极为重要的。

一般来说，网络协议有三个要素，分别是：语法、语义与规则（时序）。

- 语义规定了通信双方彼此之间准备“讲什么”，即确定协议元素的类型；
- 语法规定通信双方彼此之间“如何讲”，即确定协议元素的格式；
- 变换规则用以规定通信双方彼此之间的“应答关系”，即确定通信过程中的状态变化，通常可用状态变化图来描述。

若针对应用层协议，此三要素的含义是：

- 语法是消息的语法和描写，
- 语义是指消息的解释或含义，
- 规则是进程间通信的顺序。

网络协议是一种特殊的软件，是计算机网络实现其功能的最基本机制。网络协议的本质是规则，即各种硬件和软件必须遵循的共同守则。网络协议并不是一套单独的软件，它融合于其他所有的软件系统中，因此可以说，协议在网络中无所不在。TCP 和 UDP 是 TCP/IP 协议中的两个传输层协议，它们使用 IP 路由功能把数据包发送到目的地，从而为应用程序及应用层协议（包括：HTTP、SMTP、SNMP、FTP 和 Telnet）提供网络服务。

网络协议一般由 RFC（Requests for Comments，请求注释，Internet 标准草案定义组织）来定义，该组织中的 IETF（Internet 工程任务组，Internet Engineering Task Group）负责制定新的标准的或者协议。网络产品供应商（例如 IBM、思科、微软、Novell）就根据这些标准并将这些标准在他们的产品中实现出来。

因此，我们学习开发应用层协议，首先应该从语法、语义和规则三个方面读懂此协议对应的 RFC，再根据 RFC 中所要求的内容进行开发。

在本教材第二部分，我们分别介绍如何开发二个应用层协议： FTP 协议（RFC959 等）、

HTTP 协议（RFC2616 等）。

§ 1.3　多线程机制

1.3.1　多线程基础（同步与通信）

进程是一种重量级任务，而线程则是一种轻量级任务。线程与进程之间的主要区别在于：每一进程占有独立的地址空间，包括代码、数据及其他资源，而一个进程中的多个线程可共享该进程的这些空间；进程之间的通信（简称 IPC）开销较大且受到诸多限制，必须具有明确的对象或操作接口并采用统一的通信协议，而线程之间则可通过共享的公共数据区进行通信，开销较少且比较简单；进程间的切换开销也较大，而线程间的切换开销较小。

在应用程序中使用多线程一方面可满足问题空间本质上对并发性的需求，另一方面可在解空间充分利用多处理器体系结构以提高应用程序的性能。多线程适用于多种应用场合，其典型应用是提高交互式应用程序的响应性和提高分布式计算中服务端程序的性能。

设计多线程应用程序时，除考虑多线程带来的众多好处之外，也不应忽视使用多线程的代价。例如，多线程应用程序远比单线程应用程序复杂，程序员需谨慎处理多个计算任务之间的通信与同步；多线程应用程序更加难以调试和测试，这主要是由多线程应用程序运行结果的不确定性造成的；多线程应用程序更难以实现，要求程序员熟练掌握某一特定线程库的具体用法；多线程应用程序的某些行为还与特定平台有关，降低了应用程序的平台可移植性。

Java 线程机制建立在本地平台的线程库基础上，创建一个 Java 线程有两种途径：要么继承 Thread 类，要么实现 Runnable 接口。每一个线程都有其生存期，可用一个状态转换图描述。线程的可运行状态与正运行状态之间的切换由 JVM 线程调度程序负责，程序员也可通过调用 yield()方法主动放弃 CPU 时间。

线程阻塞是实现线程之间通信与同步的基础，Java 语言为线程提供了多种阻塞机制，其中由 synchronized 标识的同步代码段与 wait()/notify()机制是最重要的两类线程阻塞形式。线程优先级用于影响调度程序对线程的选择，程序员可通过设置优先级使重要的任务优先执行；定时器用于调度与时间有关的计算任务；守护线程是一类特殊的线程，JVM 并不将守护线程视作一个应用程序的核心部分。

1.3.2　Java 语言的多线程机制

Java 语言的线程机制建立在宿主操作系统的线程基础上，它将宿主操作系统提供的线程机制包装为语言一级的机制提供给程序员使用，一方面为程序员提供了简单一致、独立于平台的多线程编程接口，另一方面也为程序员屏蔽了宿主操作系统的线程技术细节，使得 Java 程序员不必关心如何将 Java 语言的线程机制映射到宿主操作系统的线程库，这一任务完全交由 JVM 供应商完成。

编写一个单线程的 Java 应用程序时，程序员无需显式地处理程序中的线程，由 JVM 自动完成对线程的管理，正如本书前面的大多数例子程序所示。编写多单线程的 Java 应用程序则必须由程序员显式地创建线程，并负责管理线程之间的同步与通信。

【启示】JVM 是一个进程，并且无论你编写的是否一个多线程应用程序，JVM 本身总是以多线程方式执行。例如，在JVM 中除你编写的线程外还存在着垃圾收集线程、鼠标与键盘事件分派线程等其他线程以守护线程（daemon thread）形式运行。每一个 JVM 进程都拥有一个堆空间，该进程中的每一个线程都拥有自己的调用栈空间。同一个JVM 中的所有线程可通过共同的堆空间共享或交换信息。

Java 程序中的线程被设计为一个对象，该对象具有自己特定的生存期。程序员利用接口java.lang.Runnable和类java.lang.Thread即可轻松地创建一个新线程。创建一个新线程之前程序员必须编写一个线程类，并将该线程所需执行的任务编写在该类的一个特定方法中。由程序员自定义的线程类要么实现Runnable接口，要么继承

Thread类。无论采用何种方式编写线程类，线程类中均须重定义方法run()，该方法负责完成线程所需执行的任务。

(1) 应用 Runnable 创建线程

Runnable是一个简单的接口，其中定义了所有线程类必须实现的抽象方法run()。实现 Runnable接口的线程类可节约Java 程序中宝贵的单继承指标。例如，一个Applet程序常常以这种方式实现多线程，因为Applet本身需要继承javax.swing.JApplet类。完成一个实现了 Runnable接口的线程类后，可创建该线程类的一个实例，然后再以该实例为参数创建Thread 类的一个实例，最后调用该Thread 实例的start()方法，从而创建了一个新线程。程序 Error! **No text of specified style in document.**-4 演示了如何通过实现Runnable 接口创建一个新的线程。

表 Error! No text of specified style in document.-3 通过实现 Runnable 接口创建一个新的线程

RunnableThread.java	应用 Runnable 创建线程	程序 **Error! No text of specified style in document.**-4

程序 Error! No text of specified style in document.-4 RunnableThread.java 例程

```
// 通过实现 Runnable 接口定义新线程
class Counter implements Runnable {
        public void run() {
                for (int i = 0; i < 100; i++) System.out.println("计数器 = " + i);
        }
}

public class RunnableThread {
        public static void main(String[] args) {
                Counter counter = new Counter();
                Thread thread = new Thread(counter);
                thread.start();
                System.out.println("主程序结束");
        }
}
```

程序 Error! **No text of specified style in document.**-4 利用实现Runnable 接口的线程类Counter 在主程序线程之外创建了一个新线程。读者运行上述程序时会发现，主程序输出提示信息“主程序结束”之后，仍有计数器的输出信息，这是因为虽然主程序线程已运行结束，但以Counter 类创建的线程仍在继续运行，当主程序线程与该线程均运行完毕后JVM 才运行结束。

(2) 继承 Thread 类

继承 Thread 类的线程类可用更简单的代码完成同样的事情。在利用继承Thread 类的线程类创建一个新线程时，可直接创建该线程类的一个实例。程序

Error! **No text of specified style in document.**-5 演示了如何利用Thread类的派生类创建一个新线程。

表 Error! No text of specified style in document.-4 利用 Thread 类的派生类创建一个新线程

SubclassThread.java	利用 Thread 类的派生类创建新线程	程序 **Error! No text of specified style in document.**-5

程序 Error! No text of specified style in document.-5 SubclassThread.java 例程

```
// 通过继承 Thread 类定义新线程
public class SubclassThread extends Thread {
        public void run() {
                while (true) {
                // 执行线程自身的任务
                        try {
                        sleep(5 * 1000);
                        break;
                        } catch (InterruptedException exc) {
                        // 睡眠被中断
                        }
                }
        }

        public static void main(String[] args) {
                Thread thread = new SubclassThread();
                thread.start();
                System.out.println("主程序结束");
        }
}
```

上述程序在主程序中创建了一个新线程，新线程的惟一功能是睡眠5 秒（方法sleep()的时间参数以毫秒为单位）。因而运行上述程序时，在主程序输出提示信息“主程序结束”之后，整个程序仍将持续5 秒后才结束。

当一个 Java 程序中有多个线程同时执行时，由于程序的执行效果受JVM 线程调度的影响，程序运行的一个显著特征是不确定性。

【启示】由于多线程应用程序执行效果的不确定性，不宜将程序的语义正确性建立在线程的执行次序基础上。例如，你的应用程序不应假设线程一定以某种方式调度（轮流执行或抢占执行）才能获得正确的结果。

§ 1.4 Java 序列化与反序列化机制

在分布式计算环境下，进程间是通过相互之间传递消息实现远程通信，消息中包含有各种类型的数据。而无论是何种类型的数据，都是以二进制序列的形式在网络上传送。这就需要发送进程将对象转换为字节序列，才能在网络上传送；接收进程则需要把字节序列再恢复为对象。在进程间消息通信过程中，把Java对象转换为字节序列的过程称为对象的序列化。把字节序列恢复为Java对象的过程称为对象的反序列化。由此，可以看出，对象的序列化主要有两种用途：将对象的字节序列持久化，保存在文件系统中；在网络上传送对象的字节序列。

Java语言中要求只有实现了java.io.Serializable接口的类的对象才能被序列化及反序列化。JDK类库中的有些类（如String类、包装类、Date类等）都实现了Serializable接口。　对象的序列化包括以下步骤：创建一个对象输出流，它可以包装一个其他类型的输出流，比如文件输出流；通过对象输出流写对象。对象的反序列化包括以下步骤：创建一个对象输入流，它可以包装一个其他类型的输入流，比如文件输入流；通过对象输入流读取对象。另外，在对象的序列化和反序列化过程当中，必须注意的是：为了能读出正确的数据，必须保证对象输出流的写对象的顺序与对象输入流读对象的顺序是一致的。

表 Error! No text of specified style in document.-5 实现对象序列化与反序列化例程

AccountSerializable.java	实现对象序列化与反序列化	程序 **Error! No text of specified style in document.-5**

程序 Error! No text of specified style in document.-6 AccountSerializable.java 例程

```
//测试对象的序列化和反序列化
import java.io.*;
public class AccountSerializable{
	public static void main(String[] args) throws Exception {
		//创建本地文件输入流
		File f = new File("objectFile.obj");
		ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(f));

		//序列化对象
		Account Account1 = new Account("Zhang3", 1000);
		Account Account2 = new Account("Li4", 2000);
		out.writeObject(Account1);
		out.writeObject(Account2);
		out.close();

		//反序列化对象
		ObjectInputStream in = new ObjectInputStream(new FileInputStream(f));
		Account obj1 = (Account) in.readObject();
		System.out.println("Account1=" + obj1);
		Account obj2 = (Account) in.readObject();
		System.out.println("Account2=" + obj2);
		in.close();
	}
}
//Account 类实现 java.io.Serializable 接口
class Account implements Serializable {
	private String name;
	private double balance;
	public Account(String name, double balance) {
		this.name = name;
		this.balance = balance;
	}
	public String toString() {
		return "name=" + name + ", balance=" + balance;
```

```
        }
}
```

【提示】ObjectOutputStream 只能对 Serializable 接口的类的对象进行序列化。默认情况下，ObjectOutputStream 按照默认方式序列化，这种序列化方式仅仅对对象的非 transient 的实例变量进行序列化，而不会序列化对象的 transient 的实例变量，也不会序列化静态变量。

当ObjectOutputStream对Account对象进行序列化时，如果该对象具有writeObject()方法，那么就会执行这一方法，否则就按默认方式序列化。在该对象的writeObjectt()方法中，可以先调用ObjectOutputStream的defaultWriteObject()方法，使得对象输出流先执行默认的序列化操作。同理可得出反序列化的情况，不过使用的是defaultReadObject()方法。

§ 1.5　Java 语言的反射机制

1.5.1　Java 语言的反射机制原理

通常将程序运行时，可以程序结构或变量类型的语言为动态语言。从这个观点看，perl、phthon、ruby 是动态语言，C++、JAVA、C#不是动态语言。Java 语言的反射（Reflection）机制则可以在程序运行时判断任意一个对象所属的类，构造任意一个类的对象，判断任意一个类所具有的成员变量和方法，调用任意一个对象的方法，或者生成动态代理。

在 JDK 中，主要由 java.lang.reflect 包中的 Class 类、Field 类、Method 类、Constructor 类、Array 类来实现反射机制，它们分别代表类、类的成员变量类的方法、类的构造方法，动态创建数组以及访问数组元素的静态方法。

如程序 Error! **No text of specified style in document.**-7 所示 AccountReflect 类演示了 Reflection API 的基本使用方法，AccountReflect 类有一个 copy(Object object)方法，这个方法能够创建一个和参数 object 同样类型的对象，然后把 object 对象中的所有属性复制到新建的对象中，并将它返回。此程序复制 Account 类所产生的象，Account 类的每个属性都有 public 类型的 getXXX()和 setXXX()方法。

表 Error! No text of specified style in document.-6 读取命令行参数指定的类名

AccountReflect.java	读取命令行参数指定的类名	程序 **Error! No text of specified style in document.**-7

程序 Error! No text of specified style in document.-7 AccountReflect.java 例程

```
//演示 Reflection 的基本使用方法
import java.lang.reflect.*;
public class AccountReflect {
        public Object copy(Object object) throws Exception{
        //获得对象的类型
        Class classType=object.getClass();
        System.out.println("Class:"+classType.getName());
        //通过默认构造方法创建一个新的对象
        Object objectCopy=classType.getConstructor(new Class[]{}).
```

```
            newInstance(new Object[]{});
            //获得对象的所有属性
            Field fields[]=classType.getDeclaredFields();
            for(int i=0; i<fields.length;i++){
                        Field field=fields[i];
                        String fieldName=field.getName();
                        String firstLetter=fieldName.substring(0,1).toUpperCase();
                        //获得和属性对应的 getXXX()方法的名字
                        String getMethodName="get"+firstLetter+fieldName.substring(1);
                        //获得和属性对应的 setXXX()方法的名字
                        String setMethodName="set"+firstLetter+fieldName.substring(1);
                        //获得和属性对应的 getXXX()方法
                        Method getMethod=classType.getMethod(getMethodName,new Class[]{});
                        //获得和属性对应的 setXXX()方法
                        Method setMethod=classType.getMethod(setMethodName,new Class[]{field.getType()});
                        //调用原对象的 getXXX()方法
                        Object value=getMethod.invoke(object,new Object[]{});
                        System.out.println(fieldName+":"+value);
                        //调用复制对象的 setXXX()方法
                        setMethod.invoke(objectCopy,new Object[]{value});
            }
            return objectCopy;
            }
            public static void main(String[] args) throws Exception{
                        Account Account=new Account("Zhang3",1000);
                        Account AccountCopy=(Account)new AccountReflect().copy(Account);
                        System.out.println("Copy information:"+AccountCopy.getName() + " " + AccountCopy.getBalance());
            }
}
//Account 类
class Account{
            private String name;
            private int balance;
            public Account(){}
            public Account(String name,int balance){
                        this.name=name;
                        this.balance=balance;
            }
            public String getName(){return name;}
            public void setName(String name){this.name=name;}
            public int getBalance(){return balance;}
            public void setBalance(int balance){this.balance=balance;}
}
```

1.5.2 应用反射机制实现远程方法调用

在学习了 Java 语言的反射机制后，我们很自然地会想到如何将这一机制应用于 Java 的网络程序设计中。我们可以试想，通过反射机制，客户端可以将向服务端发出的请求对象进行封装，客户端只需要知道远程对象名及其提供服务的方法名，就可以与服务端实现通信。本节应用反射机制实现远程方法调用例程综合采用了基于流的 Socket 编程，序列化机制、反射机制。事实上，本节例程可以看做是本书第三章、第四章、第五章将讲解的 RMI 与 Corba 的初步构建，希望读者能将这一例程认真掌握，举一反三。

我们在Server服务器端创建了一个AccountServiceImpl对象，它具有getAccount()。AccountServiceImpl类实现了AccountService接口。RemoteClient客户端需要把调用的方法名、方法参数类型、方法参数值，以及方法所属的类名或接口名发送给Server，Server再调用相关对象的方法，然后把方法的返回值发送给RemoteClient。

表 **Error! No text of specified style in document.**-7 应用反射机制实现远程方法调用

AccountService.java	远程对象接口	程序 **Error! No text of specified style in document.**-8
AccountServiceImpl.java	远程对象接口的实现类	程序 **Error! No text of specified style in document.**-9
RemoteCall.java	远程调用对象	程序 **Error! No text of specified style in document.**-10
RemoteClient.java	客户端程序	程序 **Error! No text of specified style in document.**-11
Server.java	服务端程序	程序 **Error! No text of specified style in document.**-12

程序 **Error! No text of specified style in document.**-8 AccountService.java 例程

```
//远程对象接口
public interface AccountService {
        public String getAccount(String Name);
}
```

程序 **Error! No text of specified style in document.**-9 AccountServiceImpl.java 例程

```
//远程对象接口的实现类
public class AccountServiceImpl implements AccountService{
        public String getAccount(String Name){
                return "Account id: "+ Name;
        }
}
```

为了便于按照面向对象的方式来处理客户端与服务器端的通信，可以把要发送的信息用RemoteCall类来封装。RemoteCall类采用序列化机制，实现Serializable接口。一个RemoteCall对象表示客户端发起的一个远程调用，它包括调用的类名或接口名、方法名、方法参数类型、方法参数值和方法执行结果。

程序 **Error! No text of specified style in document.**-10 RemoteCall.java 例程

```
//远程调用对象
import java.io.*;
public class RemoteCall implements Serializable{
	private String className; //表示类名或接口名
	private String methodName; //表示方法名
	private Class[] paramTypes; //表示方法参数类型
	private Object[] params; //表示方法参数值

	//表示方法的执行结果
	//如果方法正常执行，则 result 为方法返回值，如果方法抛出异常，那么 result 为该异常。
	private Object result;
	public RemoteCall(){}
	public RemoteCall(String className,String methodName,Class[] paramTypes, Object[] params){
		this.className=className;
		this.methodName=methodName;
		this.paramTypes=paramTypes;
		this.params=params;
	}

	public String getClassName(){return className;}
	public void setClassName(String className){this.className=className;}

	public String getMethodName(){return methodName;}
	public void setMethodName(String methodName){this.methodName=methodName;}

	public Class[] getParamTypes(){return paramTypes;}
	public void setParamTypes(Class[] paramTypes){this.paramTypes=paramTypes;}

	public Object[] getParams(){return params;}
	public void setParams(Object[] params){this.params=params;}

	public Object getResult(){return result;}
	public void setResult(Object result){this.result=result;}

	public String toString(){
		return "className="+className+" methodName="+methodName;
	}
}
```

程序 **Error! No text of specified style in document.**-11 RemoteClient.java 例程

```
//客户端程序
import java.io.*;
import java.net.*;
import java.util.*;

public class RemoteClient {
	public void invoke()throws Exception{
		Socket socket = new Socket("localhost",8000);
		OutputStream out=socket.getOutputStream();
		ObjectOutputStream oos=new ObjectOutputStream(out);
		InputStream in=socket.getInputStream();
		ObjectInputStream ois=new ObjectInputStream(in);
```

```
                RemoteCall call = new RemoteCall("AccountService", "getAccount", new Class[]{String.class},
                                                                                        new Object[]{"Zhang3"});
                //向服务器发送 Call 对象
                oos.writeObject(call);
                //接收包含了方法执行结果的 Call 对象
                call=(RemoteCall)ois.readObject();
                System.out.println(call.getResult());
                ois.close();
                oos.close();
                socket.close();
        }

        public static void main(String args[])throws Exception {
                new Client().invoke();
        }
}
```

RemoteClient调用Server端的AccountServiceImpl对象的getAccount()方法。首先，RemoteClient创建一个请求对象RemoteCall，它包含了调用AccountService接口的getAccount()。RemoteClient通过对象输出流把Call对象发送给服务端Server。接下来，Server通过对象输入流读取Call对象，运用反射机制调用AccountServiceImpl对象的getAccount()方法，把getAccount()方法的执行结果保存到Call对象中，并通过对象输出流把包含了方法执行结果的Call对象发送给RemoteClient。最后，RemoteClient通过对象输入流读取Call对象，从中获得方法执行结果。

程序 Error! No text of specified style in document.-12 Server.java 例程

```
//服务端程序
import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.reflect.*;

public class Server {
        // 存放远程对象的缓存
        private Map remoteObjects=new HashMap();
        // 把一个远程对象放到缓存中
        public void register(String className,Object remoteObject){
                remoteObjects.put(className,remoteObject);
        }
        public void service()throws Exception{
                // 创建基于流的 Socket,并在 8000 端口监听
                ServerSocket serverSocket = new ServerSocket(8000);
                System.out.println("服务器启动.....");
                while(true){
                        Socket socket=serverSocket.accept();
                        InputStream in=socket.getInputStream();
                        ObjectInputStream ois=new ObjectInputStream(in);
                        OutputStream out=socket.getOutputStream();
                        ObjectOutputStream oos=new ObjectOutputStream(out);
                        RemoteCall remotecallobj=(RemoteCall)ois.readObject(); //接收客户发送的 Call 对象
                        System.out.println(remotecallobj);
                        remotecallobj=invoke(remotecallobj); //调用相关对象的方法
```

```
                    oos.writeObject(remotecallobj);    //向客户发送包含了执行结果的 remotecallobj 对象
                    ois.close();
                    oos.close();
                    socket.close();
                }
        }

        public RemoteCall invoke(RemoteCall call){
                Object result=null;
                try{
                    String className=call.getClassName();
                    String methodName=call.getMethodName();
                    Object[] params=call.getParams();
                    Class classType=Class.forName(className);
                    Class[] paramTypes=call.getParamTypes();
                    Method method=classType.getMethod(methodName,paramTypes);
                    Object remoteObject=remoteObjects.get(className); //从缓存中取出相关的远程对象
                    if(remoteObject==null){
                        throw new Exception(className+"的远程对象不存在");
                    }else{
                        result=method.invoke(remoteObject,params);
                    }
                }catch(Exception e){result=e;}
                    call.setResult(result); //设置方法执行结果
                    return call;
                }

        public static void main(String args[])throws Exception {
                Server server=new Server();
                //把事先创建的 RemoteServiceImpl 对象加入到服务器的缓存中
                server.register("AccountService",new AccountServiceImpl());
                server.service();
        }
}
```

图 **Error! No text of specified style in document.**-7 显示了RemoteClient与Server的通信过程。

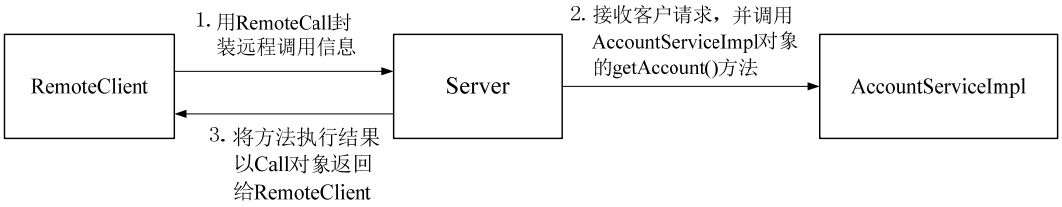

图 **Error! No text of specified style in document.**-7 RemoteClient 与 Server 的通信过程

1.5.3 应用代理模式实现远程方法调用

为了使程序具有更良好的扩展性，我们可以在 2.6.2 节的程序中采用代理模式来实现远程方法调用。代理模式的特征是代理类与委托类有同样的接口。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存

在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。

表 Error! No text of specified style in document.-8 应用代理模式实现远程方法调用

AccountService.java	远程对象接口	程序 **Error! No text of specified style in document.**-8
AccountServiceImpl.java	远程对象接口的实现类	程序 **Error! No text of specified style in document.**-9
RemoteCall.java	远程调用对象	程序 Error! **No text of specified style in document.**-10
Connector.java	建立连接，以及接收和发送 Socket 对象	程序 Error! **No text of specified style in document.**-14
ProxyClient.java	客户端程序	程序 Error! **No text of specified style in document.**-16
StaticServiceProxy.java	静态代理类	程序 Error! **No text of specified style in document.**-13
DynamicProxyFactory.java	动态代理类	程序 Error! **No text of specified style in document.**-15
Server.java	服务端程序	程序 **Error! No text of specified style in document.**-12

如程序 **Error! No text of specified style in document.**-13 StaticServiceProxy 类即为代理类，而程序 **Error! No text of specified style in document.**-9 AccountServiceImpl 类为委托类，这两个类实现了 程序 **Error! No text of specified style in document.**-8 AccountService 所描述的接口。

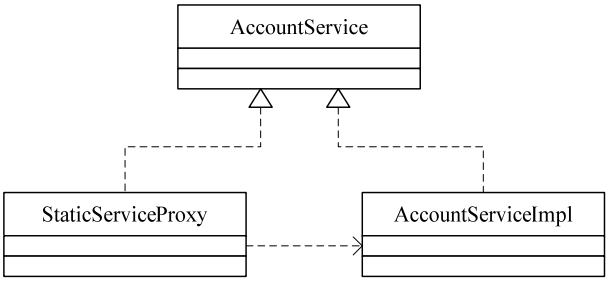

图 **Error! No text of specified style in document.**-8 委托类 AccountServiceImpl 及代理类 StaticServiceProxy 共同实现 AccountService 接口

程序 **Error! No text of specified style in document.**-13 StaticServiceProxy.java 例程

```
//静态代理类，实现 AccountService 接口
public class StaticServiceProxy implements AccountService{
        private String host;
        private int port;

        public StaticServiceProxy(String host,int port){
                this.host=host;
                this.port=port;
        }

        public String getAccount(String Name) {
                Connector connector=null;
                try{
                        connector=new Connector(host,port);
                        RemoteCall call = new RemoteCall("AccountService","getAccount",
                                                                new Class[]{String.class},new
Object[]{Name});
                        connector.send(call);
                        call=(RemoteCall)connector.receive();
                        Object result=RemoteCall.getResult();
                        return (String)result;
                } catch (Exception e) {
                        e.printStackTrace();
                }finally{if(connector!=null)connector.close();}
                return Name;
        }
}
```

【提示】代理(Proxy)模式是设计模式中行为类型的一种，它可以为其他对象提供一种代理以控制对这个对象的访问。按照代理类的创建时期，代理类可分为静态代理与动态代理两类。其中静态代理类由程序员创建或由特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class 文件就已经存在了。而动态代理类是在程序运行时，运用反射机制动态创建而成。

程序 **Error! No text of specified style in document.**-14 Connector.java 例程

```
//负责建立与远程服务器的连接，以及接收和发送 Socket 对象
import java.io.*;
import java.net.*;
public class Connector {
        private String host;
```

```
	private int port;
	private Socket skt;
	private InputStream is;
	private ObjectInputStream ois;
	private OutputStream os;
	private ObjectOutputStream oos;

	public Connector(String host,int port)throws Exception{
		this.host=host;
		this.port=port;
		connect(host,port);
	}
	public void send(Object obj)throws Exception{ //发送对象方法
		oos.writeObject(obj);
	}
	public Object receive() throws Exception{ //接收对象方法
		return ois.readObject();
	}
	public void connect()throws Exception{ //建立与远程服务器的连接
		connect(host,port);
	}
	public void connect(String host,int port)throws Exception{ //建立与远程服务器的连接
		skt=new Socket(host,port);
		os=skt.getOutputStream();
		oos=new ObjectOutputStream(os);
		is=skt.getInputStream();
		ois=new ObjectInputStream(is);
	}
	public void close(){ //关闭连接
		try{
			ois.close();
			oos.close();
			skt.close();
		}catch(Exception e){
			System.out.println("Connector.close: "+e);
		}
	}
}
```

Connector 类负责建立与远程服务器的连接，以及接收和发送Socket对象。

在静态代理模式中，客户端PorxyClient通过AccountService代理类StaticServiceProxy来调用Server服务器端的AccountServiceImpl对象的方法。StaticServiceProxy代理类也实现了AccountService接口，这可以简化客户端PorxyClient的编程。对于客户端而言，与远程服务器的通信的细节被封装到代理类StaticServiceProxy中。

另一种代理类的创建方法是采用动态模式，DynamicProxyFactory类的静态getProxy()方法就负责创建AccountService 的动态代理类，并且返回它的一个实例。动态代理类也是通过Connector类来发送和接收RemoteCall对象。getProxy()方法的第一个参数classType指定代理类实现的接口的类型，如果参数classType的取值为AccountService.class，那么getProxy()方法就创建AccountService动态代理类的实

例。由此可见，getProxy()方法可以创建任意类型的动态代理类的实例，并且它们都具有调用被代理的远程对象的方法的能力。

程序 **Error! No text of specified style in document.**-15 DynamicProxyFactory.java 例程

```
//动态代理类
import java.lang.reflect.*;
public class DynamicProxyFactory {
	public static Object getProxy(final Class classType,final String host,final int port){
		InvocationHandler handler=new InvocationHandler(){
			public Object invoke(Object proxy,Method method,Object args[]) throws Exception{
			Connector connector=null;
			try{
				connector=new Connector(host,port);
				RemoteCall call=new RemoteCall(classType.getName(),
				method.getName(),method.getParameterTypes(),args);
				connector.send(call);
				call=(RemoteCall)connector.receive();
				Object result=call.getResult();
				return result;
			}finally{if(connector!=null)connector.close();}
			}
		};
		return Proxy.newProxyInstance(classType.getClassLoader(),new Class[]{classType},handler);
		}
}
```

因此如果使用静态代理方式，那么对于每一个需要代理的类，都要手工编写静态代理类的源代码；如果使用动态代理方式，那么只要编写一个动态代理工厂类，它就能自动创建各种类型的动态代理类，从而大大简化了编程，并且提高了软件系统的可扩展。

程序 **Error! No text of specified style in document.**-16 ProxyClient.java 例程

```
//客户端程序
public class ProxyClient {
	public static void main(String args[])throws Exception {
		//创建静态代理类实例
		StaticServiceProxy Service1=new StaticServiceProxy("localhost",8000);
		System.out.println(Service1.getAccount("Zhang3"));
		//创建动态代理类实例
		AccountService
Service2=(AccountService)DynamicProxyFactory.getProxy(AccountService.class,"localhost",8000);
		System.out.println(Service2.getAccount("Li4"));
	}
}
```