第 2 章　基于 Socket 通信

网络通信是一种进程间通信（IPC：Inter-Process Communication），要求位于不同网络节点不同进程的通信双方必须遵循统一的通信协议方可实现。基于套接字（Socket）通信是应用在不同节点上的进程间通信的典型方法。

本章介绍了基于Socket通信的基本方式：数据包Socket和流式Socket，详细说明其基本原理与实现方法。针对因特网典型应用与应用层协议开发，本章介绍简单的Daytime协议应用开发，以及文件传输协议（FTP：File Transfer Protocol）、超文本传输协议（HTTP：Hypertext Transfer Protocol）基本原理与开发过程。

§ 2.1　基本原理

2.1.1　Socket API 基本概念

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。以上Socket API都有相同的消息模型和类似的语法。Socket API 的概念模型如 图 2-1 所示。

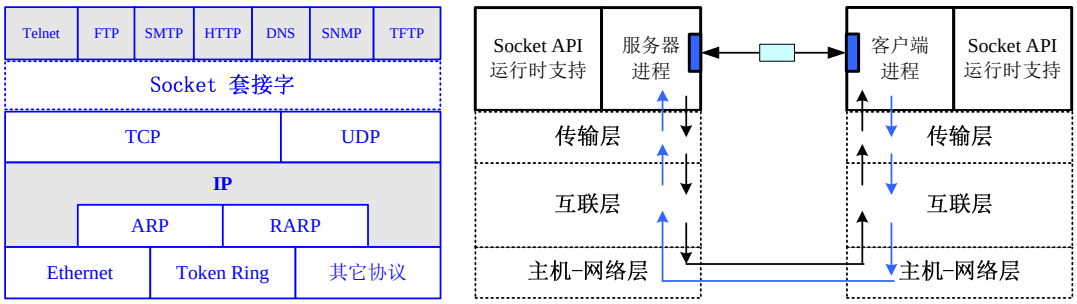

图 2-1　Socket API 的概念模型

Socket是物理网络地址和逻辑端口号的一个集合，通过这个集合可以向另外一个位置的与它具有相同定义的Socket进行数据传输。由于Socket是由机器地址和端口号来识别的，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被定位。套接字类型有二种：一个是流式Socket，它提供进程之间的逻辑连接，并且支持可靠的数据交换；另一个就是数据包Socket，它是无连接的并且不可靠。

【提示】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 非常类似于电

话插座，电话的通话双方相当于相互通信的 2 个进程，区号是它的网络地址；区内一个单位的交换机相当于一台主机，主机分配给每个用户的局内号码相当于 socket 号。任何用户在通话之前，首先要占有一部电话机，相当于申请一个 socket；同时要知道对方的号码，相当于对方有一个固定的 socket。

2.1.2 JAVA 对网络通信的支持

JDK（Java Developer's Kit）提供的预定义类库支持Java程序直接发送与接收TCP数据段或UDP数据报，并帮助程序员以更直接的方式处理建立在TCP之上的应用层协议HTTP、FTP、FILE、HTTPS等。程序员使用这些预定义的类可轻松实现基于TCP和UDP的编程，或更直接地通过HTTP、FTP、FILE等协议访问以URL定位的网上资源。以面向连接的TCP为基础，程序员还可实现常见的Telnet、FTP、SMTP等应用层协议的客户程序或服务程序。

JDK预定义的类均存放在程序包java.net中，使用其中的哪些类取决于所需处理的通信协议。例如，基于TCP的应用程序可使用Socket、ServerSocket等类；基于UDP的应用程序则使用DatagramPacket、DatagramSocket、MulticastSocket等类；基于HTTP和FTP等协议直接访问URL资源的应用程序可使用URL、URLConnection等类；更注重网络通信安全的HTTPS应用程序则使用SSLSocketFactory、SSLSocket、SSLSession等类。

【提示】JDK 目前提供的预定义类库并不支持网络层 IP 协议，因而程序员无法使用预定义的类直接在 Java 程序中发送或接收原始 IP 数据报。倘若应用程序需处理 TCP 和 UDP 之外的传输层协议，或者是直接处理网络层或链路层的任何协议（例如 IP、ICMP、IGMP、ARP/RARP 等协议），可借助于 Java 语言的地代码（native code）机制。

2.1.3 TCP、UDP 与端口

尽管位于传输层的协议TCP和UDP均使用相同的网络层协议IP，但这两种传输层协议却为应用层提供了完全不同的服务：TCP提供一种面向连接的、可靠的数据流服务，而UDP则提供一种面向独立数据报的高效传输服务。

(1) TCP 传输控制协议

传输控制协议（TCP：Transfer Control Protocol）是一种面向连接的传输层协议，可为两台不同主机上的应用程序提供可靠的数据流连接。所谓“面向连接”，意味着两个使用TCP通信的应用程序在交换数据之前，必须先建立一个TCP连接，待通信结束后须关闭该连接。这一过程与电话通信相似：先拨号振铃，等待对方摘机应答后再表明身份，通话完毕后挂机。

TCP执行的任务包括把应用层传来的数据分解为合适的数据段交给网络层，确认接收到的数据分组，设置发送最后确认分组的超时时间等，为应用层屏蔽了实现端到端可靠通信的细节。TCP可保证数据从连接的一端送到另一端时仍能保持原来发送时的次序，否则将一个传输错误。TCP为需要可靠通信的应用程序提供了一种点对点信道，在因特网上常见的FTP、Telnet、SMTP等大多数应用层协议都建立在TCP的基础上。

(2) UDP 用户数据报协议

与TCP协议不同的是，用户数据报协议（UDP：User Datagram Protocol）不是基于连接的，而是为应用层提供一种非常简单、高效的传输服务。UDP从一个应

用程序向另一应用程序发送独立的数据报，但并不保证这些数据报一定能到达另一方，并且这些数据报的传输次序无保障，后发送的数据报可能先到达目的地。

使用UDP协议时，任何必需的可靠性都须由应用层自己提供。UDP适用于对通信可靠性要求低且对通信性能要求高的应用，诸如域名系统DNS（Domain Name System）、路由信息协议RIP（Routing Information Protocol）、简单网络管理协议SNMP（Simple Network Management Protocol）、普通文件传送协议TFTP（Trivial File Transfer Protocol）等应用层协议都建立在UDP的基础上。

虽然UDP不如TCP那样常用，但UDP因其固有的特点而在某些应用场合可更好地发挥特长。例如在一个TCP连接中仅允许两方参与通信，故广播和多播方式不能用于TCP协议，但UDP却能有效地支持广播和多播通信；又如，对一个播发时间的服务程序而言，重发丢失的数据已毫无意义，因为知道数据丢失后原来的时间已经过去；使用ping方式测试两个应用序的通信效果时，反而需要计算数据的丢失率；在一些协同工作的网络应用程序之间，还可利用UDP实现心跳消息（heartbeat message）彼此通知各自的存在。注意在适合UDP协议的各种应用中，要么是因为TCP协议提供的可靠性服务是多余的，要么由应用程序自己实现了所需的可靠性。

(3) 端口

由于现代计算机大多运行多任务操作系统，故一台主机上可能同时运行多个应用程序进程，并且一个进程还可能使用多个不同的连接，因而仅用主机名或IP地址无法惟一地标识数据包的源或目标。端口为标识参与通信的主机、进程和连接提供了一种统一的、惟一的方法，

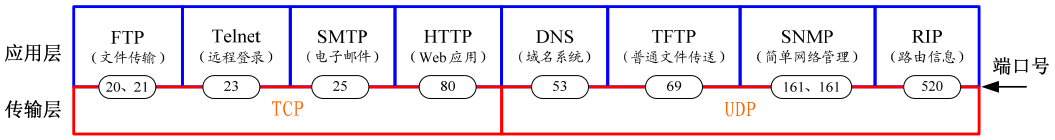

图 2-2 应用层协议及其对应端口

【思考】请与并行口或串行口等具体物理实体相对比，理解端口是一个物理概念，还仅仅是一个程序设计层面的逻辑概念？

因特网上数据的传输目标由主机和端口号组成，例如UDP这类基于数据报通信的每一数据报中均含有端口号信息。主机可由32位的IP地址标识（假设采用IPv4），端口则采用16位数字标识，故端口号的取值范围为0-65535。编号为0-1023的端口保留给系统服务使用，包括HTTP、FTP、Telnet、SMTP等常见服务。应用程序则使用剩余的其他端口，例如Java远程方法调用（RMI）使用端口1099，Microsoft SQL Server默认使用端口1433，BEA WebLogic Server默认使用端口7001和7002等。（如 图 2-2 所示。）

【提示】对于基于连接的通信（例如 TCP 协议）而言，端口为不同应用程序提供了虚拟的专用连接，每一个 socket 都绑定到特定的端口号。

§ 2.2　数据报 Socket

2.2.1　基本编程原理

UDP采用数据报进行通信，数据报是否可到达目标、以什么次序到达目标、到达目标时内容是否依然正确等均是未经校验的。因而UDP是一种不可靠的点对点通信，适合对通信性能要求高、但通信可靠性要求低的应用，并可支持广播和多播通信方式。

与基于TCP的通信类似，基于UDP的单播通信是将数据报从一个发送方传输给单个接收方。java.net程序包为实现UDP单播通信主要提供了两个类：类DatagramPacket代表一个被传送的UDP数据包，该类封装了被传送数据报的内容、源主机与端口号、目标主机与端口号等信息；类DatagramSocket代表一个用于传送UDP数据包的UDPsocket。

【提示】由于 UDP 并不是一种基于连接的协议，因而对 UDP 而言没有类似 TCP 的 I/O 流机制。

(1)　类 DatagramPacket

从类DatagramPacket的构造方法可看出，创建一个DatagramPacket实例时必须提供被封装的数据报详细信息：

```
DatagramPacket(byte buf[], int offset, int length)
DatagramPacket(byte buf[], int length)
DatagramPacket(byte buf[], int offset, int length, InetAddress address, int port)
DatagramPacket(byte buf[], int offset, int length, SocketAddress address) throws SocketException
DatagramPacket(byte buf[], int length, InetAddress address, int port)
DatagramPacket(byte buf[], int length, SocketAddress address) throws SocketException
```

其中，参数buff指定一个字节数组作为缓冲区，用于存放所接收的UDP数据报；参数offset指定缓冲区中的偏移量；参数length表示以字节计算的数据报长度，该参数不可大于缓冲区的大小；参数address和port指定数据报传送目标的地址与端口号，可调用该实例的setAddress()和setPort()方法重新设定目标地址和端口号。

类DatagramPacket既可描述客户程序发送的一个UDP数据报，也可描述服务程序接收的一个UDP数据报。该类提供了多个方法用于设置或访问UDP数据报的状态，例如接收或发送该数据报的主机（IP地址与端口号）、存放在内部缓冲区中的数据报内容以及缓冲区的偏移量与数据报长度等。这些设置和访问方法包括：

```
//设置接收该数据报的主机地址
public synchronized void setAddress(InetAddress iaddr)
//返回发送或接收数据报的主机地址
public synchronized InetAddress getAddress()
//设置接收该数据报的主机端口号
public synchronized void setPort(int iport)
//返回发送或接收数据报的主机端口号
public synchronized int getPort()
//设置接收该数据报的远程主机地址（SocketAddress 实例中已含 IP 地址和端口号）
public synchronized void setSocketAddress(SocketAddress address)
//返回接收或发送该数据报的远程主机地址
public synchronized SocketAddress getSocketAddress()
//设置数据报中的数据缓冲区（偏移量为 0 且长度为缓冲区的长度）
public synchronized void setData(byte[] buf)
//设置缓冲区中的数据、偏移量与长度
public synchronized void setData(byte[] buf, int offset, int length)
```

```
// 返回缓冲区中从偏移量 offset 开始、长度为 length 的数据
public synchronized byte[] getData()
// 返回缓冲区的偏移量设置
public synchronized int getOffset()
// 设置数据报的长度
public synchronized void setLength(int length)
// 返回发送或接收的数据报的长度
public synchronized int getLength()
```

(2) 类 DatagramSocket

基于TCP 的通信是一种面向连接的socket，而UDP socket则面向一个个独立的数据报。一个UDP socket既可用于发送UDP数据报，也可用于接收UDP数据报。类DatagramSocket封装了一个UDP socket绑定的本地主机地址与端口号，及其连接的远程主机地址与端口号，并且支持通过该UDP socket发送和接收UDP数据报。

在创建一个DatagramSocket实例时，可通过不同形式的构造方法指定该UDP socket绑定的主机地址与端口号：

```
DatagramSocket() throws SocketException
DatagramSocket(SocketAddress bindAddr) throws SocketException
DatagramSocket(int port) throws SocketException
DatagramSocket(int port, InetAddress addr) throws SocketException
```

其中，参数bindAddr指定新实例绑定的本地socket地址，参数address和port指定新实例绑定的本地主机地址和端口号。

类DatagramSocket负责管理UDP socket的绑定和连接状态，执行发送和接收UDP数据报的操作。当一个UDP socket连接到一个由地址和端口号指定的远程主机时，该socket只能向该地址发送或从该地址接收UDP数据报。类DatagramSocket提供的主要方法如下：

```
// 将当前 UDP socket 绑定到一个本地主机地址与端口号
public synchronized void bind(SocketAddress addr) throws SocketException
// 返回当前绑定的本地主机地址和端口号
public SocketAddress getLocalSocketAddress()
// 返回当前绑定的本地主机地址
public InetAddress getLocalAddress()
// 返回当前绑定的本地主机端口号
public int getLocalPort()
// 返回当前的绑定状态
public boolean isBound()
// 将当前 UDP socket 连接到一个远程主机地址与端口号
public void connect(InetAddress address, int port)
public void connect(SocketAddress addr) throws SocketException
// 断开当前 UDP socket 与远程主机的连接
public void disconnect()
// 返回当前连接的远程主机地址和端口号
public SocketAddress getRemoteSocketAddress()
// 返回当前连接的远程主机地址
public InetAddress getInetAddress()
// 返回当前连接的远程主机端口号
public int getPort()
// 返回当前的连接状态
public boolean isConnected()
// 利用当前 UDP socket 发送一个 UDP 数据报
```

```
public void send(DatagramPacket p) throws IOException
// 利用当前 UDP socket 接收一个 UDP 数据报
public synchronized void receive(DatagramPacket p) throws IOException
// 关闭当前 UDP socket
public void close()
// 判断当前 UDP socket 是否已关闭
public boolean isClosed()
```

【提示】除上述主要方法外，类 DatagramSocket 还提供了 UDP socket 选项的设置与访问方法。例如，setSoTimeout()和 getSoTimeout()可用于设置或访问 UDP socket 的超时选项 SO_TIMEOUT，类似地还可设置或访问 UDP socket 的 SO_SNDBUF、SO_RCVBUF、SO_REUSEADDR、SO_BROADCAST 等选项。

2.2.2 面向无连接数据包 UDP

图 2-3 显示UDP与TCP协议之间的不同在于UDP不是一种基于稳定连接的通讯协议，UDP协议使用数据报式套接字，UDP在数据传输之前不需要先建立连接，UDP没有组装和重传请求的功能，并不保证接收方能够接收到该数据包，也不保证接收方所接收到的数据和发送方所发送的数据在内容和顺序上是完全一致的。其主要工作是将应用程序传输过来的数据分块交给网络层，确认接受到分组信息。

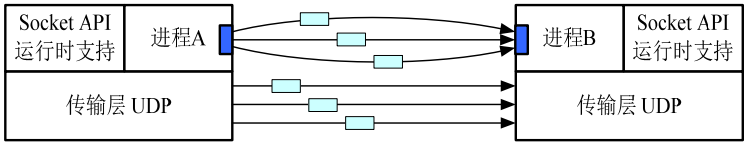

图 2-3 无连接数据包 Socket

在Java.net包中，DatagramPacket类用于实现数据报的接收和发送、读取报文信息等。在图 2-4 建立的模型中，使用UDP数据报协议，以收发数据报作为通信方式。每个数据报文有独立的源地址和目的地址，服务器和客户端不建立连接，通信的内容数据报短信息的形式实现，但传输时不能保证对方一定能收到，也不能保证收到的报文次序。

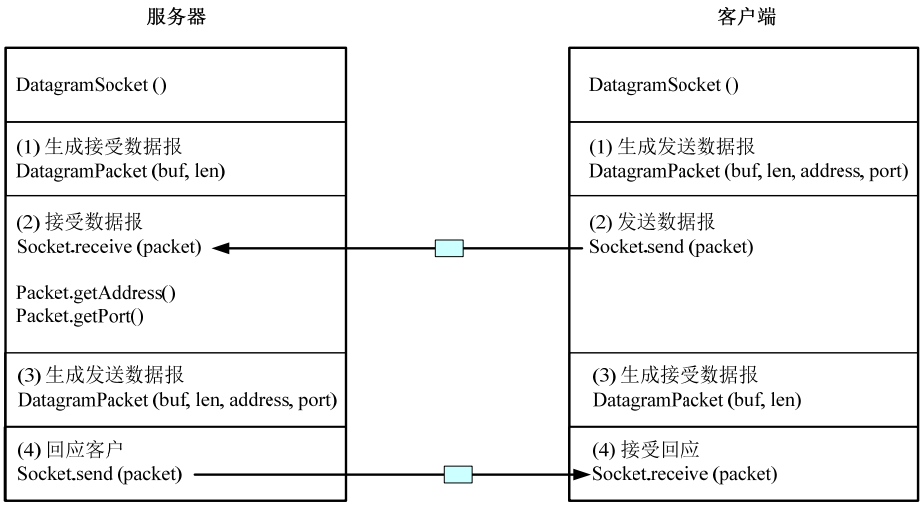

图 2-4 数据报 Socket 通信过程

以下例程 ***UDPConnectless*** 实现无连接数据包传递数据。实现由一方发送消息给另一方，另一方通过接收到消息后，再返回消息给发送方。（如 图 2-5 和 表 2-1 所示）

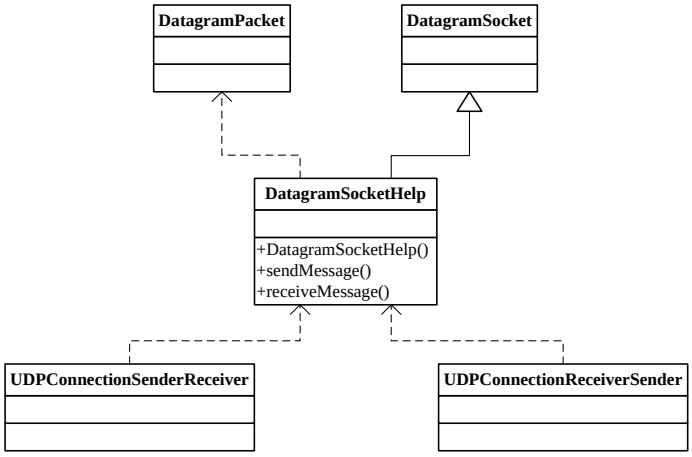

图 2-5 无连接数据包程序 UDPConnectless 类图

表 2-1 无连接数据包例程 UDPConnectless

UDPConnectionSenderReceiver.java	发送接收类	程序 2-1
UDPConnectionReceiverSender.java	接收返回类	程序 2-2
DatagramSocketHelp.java	数据报帮助类	程序 2-3

程序 2-1 发送接收类 UDPConnectionSenderReceiver.java

```
// 发送接收类
import java.net.*;

public class UDPConnectionSenderReceiver {
  public static void main(String[] args) {
    if (args.length != 4)
      System.out.println(
        "命令行参数: 接收方 IP 地址，接收方端口，发送方端口，发送字符串");
    else {
      try {
        InetAddress receiverHost = InetAddress.getByName(args[0]);
        int receiverPort = Integer.parseInt(args[1]);
        int myPort = Integer.parseInt(args[2]);
        String message = args[3];
        // 产生数据报用于发送和接收数据
        DatagramSocketHelp mySocket = new DatagramSocketHelp(myPort);
        // 发送数据报
        mySocket.sendMessage(receiverHost, receiverPort, message);
        // 等待返回数据
        System.out.println(mySocket.receiveMessage());
        mySocket.close();
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

程序 2-1 通过实例化 DatagramSocketHelp 类，由 DatagramSocketHelp 对象产生数据报用于发送和接收数据。由 DatagramSocketHelp 对象的 sendMessage() 方法发送数据报，由.receiveMessage()接收数据，接收数据后则销毁 DatagramSocketHelp 对象。

程序 2-2 接收返回类 UDPConnectionReceiverSender.java

```
//接收返回类
import java.net.*;

public class UDPConnectionReceiverSender {
   public static void main(String[] args) {
      if (args.length != 4)
         System.out.println(
           "命令行参数: 发送方 IP 地址，发送方端口，接收方端口，发送字符串");
      else {
         try {
            InetAddress receiverHost = InetAddress.getByName(args[0]);
            int receiverPort = Integer.parseInt(args[1]);
            int myPort = Integer.parseInt(args[2]);
            String message = args[3];
            // 产生数据报用于接收和返回数据
            DatagramSocketHelp mySocket = new DatagramSocketHelp(myPort);
            //接收数据报
            System.out.println(mySocket.receiveMessage());
            //返回数据报
            mySocket.sendMessage(receiverHost, receiverPort, message);
            mySocket.close();
         } catch (Exception ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

程序 2-2 通过实例化 DatagramSocketHelp 类，由 DatagramSocketHelp 对象产生数据报用于接收和返回数据。由 DatagramSocketHelp 对象的 receiveMessage()先接收数据，接收数据后再通过 sendMessage()方法发送数据报，最后销毁 DatagramSocketHelp 对象。

程序 2-3 数据报帮助类 DatagramSocketHelp.java

```
//数据报帮助类
import java.net.*;
import java.io.*;

public class DatagramSocketHelp extends DatagramSocket {
   static final int MAX_LEN = 100;
   DatagramSocketHelp(int portNo)throws SocketException {
      super(portNo);
   }
   //发送消息方法
   public void sendMessage(InetAddress receiverHost, int receiverPort, String
      message)throws IOException {
      byte[] sendBuffer = message.getBytes();
      DatagramPacket datagram = new DatagramPacket(sendBuffer,
         sendBuffer.length, receiverHost, receiverPort);
      this.send(datagram);
```

```
  }
  // 接收消息方法
  public String receiveMessage()throws IOException {
    byte[] receiveBuffer = new byte[MAX_LEN];
    DatagramPacket datagram = new DatagramPacket(receiveBuffer, MAX_LEN);
    this.receive(datagram);
    String message = new String(receiveBuffer);
    return message;
  }
}
```

数据报帮助类 DatagramSocketHelp 继承 DatagramSocket 类，DatagramSocketHelp 类中有二个主要的方法，即发送消息方法 sendMessage()与接收消息方法 receiveMessage()，通过 DatagramPacket 类的实例化对象，实现数据报的接收和发送、读取报文信息等。

2.2.3 面向连接数据包 UDP

一般来说，很少用数据报 Socket 实现面向连接通信，因为此 API 提供的连接非常简单，通常难以满足应用要求。但我们可以从这个例子中可以看出“分层”思想的体现。

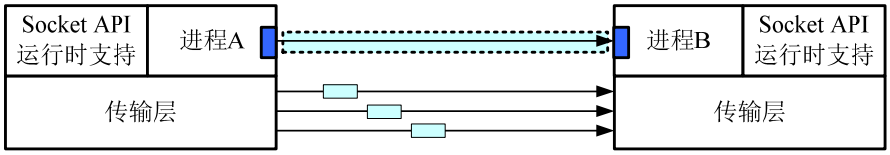

图 2-6 面向连接数据包 Socket

例程 UDPConnection 实现面向连接数据包传递数据。类 DatagramSocketHelp 继承了 DatagramSocket，DatagramSocket 中有用于创建和终止连接的两个方法。Socket 连接通过指定远程 Socket 地址建立。一旦连接建立后，Socket 被用来与远程 Socket 交换数据报文。在 Send 操作中，如果数据报与另一端地址不匹配，将引发异常。如果发送到 Socket 的数据来源于某个发送源，而不是源于与之连接的远程 Socket，此数据就被忽略。因而，连接一旦与数据报绑定后，此 Socket 将不能与任何其他 Socket 进行通信，直到此连接终止。（如 表 2-2 所示）

表 2-2 面向连接数据包例程 UDPConnection

UDPConnectionSender.java	面向连接数据包发送类	程序 2-4
UDPConnectionReceiver.java	面向连接数据包接收类	程序 2-5
DatagramSocketHelp.java	数据报帮助类	程序 2-3

程序 2-4 面向连接数据包发送类 UDPConnectionSender.java

```
// 面向连接数据包发送类
import java.net.*;

public class UDPConnectionSender {
  public static void main(String[] args) {
    if (args.length != 4)
      System.out.println(
        "命令行参数: 接收方 IP 地址, 接收方端口, 发送方端口, 发送字符串");
    else {
      try {
        InetAddress receiverHost = InetAddress.getByName(args[0]);
        int receiverPort = Integer.parseInt(args[1]);
        int myPort = Integer.parseInt(args[2]);
        String message = args[3];
```

```
        // 创建面向连接的数据报
        DatagramSocketHelp mySocket = new DatagramSocketHelp(myPort);
        // 建立连接
        mySocket.connect(receiverHost, receiverPort);
        for (int i = 0; i < 10; i++)
          mySocket.sendMessage(receiverHost, receiverPort, message);
        // 接收从接收方的返回数据
        System.out.println(mySocket.receiveMessage());
        // 断开连接，关闭 Socket
        mySocket.disconnect();
        mySocket.close();
      } catch (Exception ex) {
        System.out.println(ex);
      }
    }
  }
}
```

程序 2-4 通过 DatagramSocketHelp 类的 connect()方法创建面向连接的数据报，connect()方法的参数包括接收方的主机地址与接收方端口，建立连接后，再通过 sendMessage()方法向接收方发送数据，利用 receiveMessage()方法接收数据。

程序 2-5 面向连接数据包接收类 UDPConnectionReceiver.java

```
// 面向连接数据包接收类
import java.net.*;

public class UDPConnectionReceiver {
  public static void main(String[] args) {
    if (args.length != 4)
      System.out.println(
        "命令行参数: 发送方 IP 地址，发送方端口，接收方端口，发送字符串");
    else {
      try {
        InetAddress senderHost = InetAddress.getByName(args[0]);
        int senderPort = Integer.parseInt(args[1]);
        int myPort = Integer.parseInt(args[2]);
        String message = args[3];
        // 创建面向连接的数据报
        DatagramSocketHelp mySocket = new DatagramSocketHelp(myPort);
        // 建立连接，接收数据
        mySocket.connect(senderHost, senderPort);
        for (int i = 0; i < 100; i++)
          System.out.println(mySocket.receiveMessage());
        // 向对方发送数据
        mySocket.sendMessage(senderHost, senderPort, message);
        mySocket.close();
      } catch (Exception ex) {
        System.out.println("An exception has occured: " + ex);
      }
    }
  }
}
```

程序 2-5 与 程序 2-4 类似，通过 DatagramSocketHelp 类的 connect()方法创建面向连接的数

据报，connect()方法的参数包括发送方的主机地址与接收方端口，建立连接后，利用receiveMessage()方法接收数据，再通过 sendMessage()方法向接收方发送数据。

【思考】请从第一章介绍的隐式地与显式地角度出发，理解例程 UDPConnection 基于数据包 socket 实现面向连接通信过程。

§ 2.3　流式 Socket

2.3.1　基本编程原理

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 2-7 所示）。

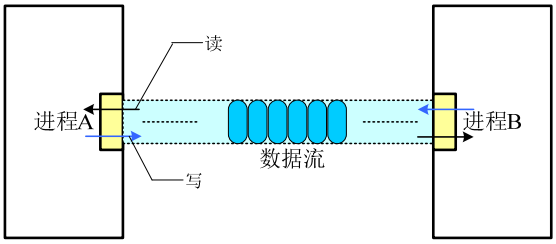

图 2-7 流式 Socket 连接控制方式

在这个过程中，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。

在基于 TCP 的通信中，应用程序需直接使用 IP 地址或域名指定运行在因特网上的某一台主机。java.net 程序包中定义的 InetAddress 类是一个 IP 地址或域名的抽象。创建 InetAddress 类的一个实例时既可使用字符串表示的域名，也可使用字节数组表示的 IP 地址。该类的主要内容如下所示：

```
//// 工厂方法（类方法）
// 利用主机名创建一个实例
static InetAddress getByName(String host) throws UnknownHostException
// 利用 IP 地址创建一个实例
static InetAddress getByAddress(byte[] addr) throws UnknownHostException
// 利用主机名和 IP 地址创建一个实例
static InetAddress getByAddress(String host, byte[] addr) throws UnknownHostException
// 根据主机名返回该主机所有 IP 地址的实例数组（例如多接口主机可绑定多个 IP 地址）
static InetAddress[] getAllByName(String host) throws UnknownHostException
// 返回本地主机的一个实例
static InetAddress getLocalHost() throws UnknownHostException

//// 属性访问方法（实例方法）
// 取出当前实例的主机名（参数 check 指定是否执行安全检查，默认值为 true）
String getHostName()
String getHostName(boolean check)
// 取出当前实例的完整的域名
String getCanonicalHostName()
// 取出当前实例的 IP 地址
byte[] getAddress()
// 取出当前实例的 IP 地址字符串
String getHostAddress()
```

```
//// 检测当前实例的 IP 地址所属的范围（实例方法）
// 判断是否一个多播地址（在 IPv4 中即 224.0.0.0 至 239.255.255.255 范围的所谓 D 类 IP 地址）
boolean isMulticastAddress()
// 判断是否一个通配地址（在 IPv4 中即 0.0.0.0）
boolean isAnyLocalAddress()
// 判断是否一个环回地址（在 IPv4 中即 localhost 的 IP 地址 127.*.*.*）
boolean isLoopbackAddress()
// 判断单播地址是否本地链路范围（在 IPv4 中即 169.254/16 为前缀的地址）
boolean isLinkLocalAddress()
// 判断单播地址是否本地站点范围（在 IPv4 中即 10/8、172.16/12 和 192.168/16 为前缀的地址）
boolean isSiteLocalAddress()
// 判断多播地址是否全局范围（在 IPv4 中即 224.0.1.0 至 238.255.255.255，MC 指 Multicast）
boolean isMCGlobal()
// 判断多播地址是否本地结点范围（仅 IPv6 使用）
boolean isMCNodeLocal()
// 判断多播地址是否本地链路范围（在 IPv4 中即 224.0.0/24 为前缀的地址）
boolean isMCLinkLocal()
// 判断多播地址是否本地站点范围（在 IPv4 中即 239.255/16 为前缀的地址）
boolean isMCSiteLocal()
// 判断多播地址是否本地机构范围（在 IPv4 中即 239.192/14 为前缀的地址）
boolean isMCOrgLocal()
```

注意在该类的设计中采用了 GoF 的工厂方法（Factory Method）设计模式，即不使用普通的构造方法创建实例，而是代之以静态工厂方法 getByName()、getByAddress()、getAllByName()、getLocalHost()等。

【提示】 工厂方法模式（Factory Method）：定义一个用于创建对象的接口，让子类决定实例化哪一个类；该模式允许一个类将实例化工作推迟到其子类中。类 InetAddress 采用工厂方法设计模式的好处是使客户程序可透明地使用 IPv4 协议和 IPv6 协议。采用 IPv4 协议时，工厂方法返回 Inet4Address 的实例；采用 IPv6 协议时，工厂方法返回 Inet6Address 的实例。Inet4Address 和 Inet6Address 都是 InetAddress 的子类。

2.3.2 单线程服务程序与客户程序

使用基于 TCP 协议的双向通信时，网络中的两个应用程序之间必须首先建立一个连接，这一连接的两个端点分别被称为 socket。由于 socket 被绑定到某一固定的端口号，故 TCP 可将数据传输给正确的应用程序。从应用编程的角度看，应用程序可将一个输入流或一个输出流绑定到某一 socket，读写这些输入／输出流即可实现基于 TCP 的通信。

使用网络通信的应用程序普遍采用客户机／服务器计算模型，其中客户程序作为通信的发起者，向服务程序提出信息或服务请求；服务程序则负责提供这种信息或服务，服务程序经常在一个无限循环中等待客户程序的请求并执行相应的服务。

(1) 使用 socket 的通信模式

在 Java 程序中使用 socket 进行通信可支持在一台主机上运行的单个服务程序为多个不同的客户程序提供服务。为实现这一目标，服务程序将选定一个固定的端口号对外发布服务客户程序则必须先按约定的主机与端口号向服务程序发送一个要求建立连接的请求（这些请求称为连接请求），申请建立一个到服务程序的连接。（如 图 2-8 所示）

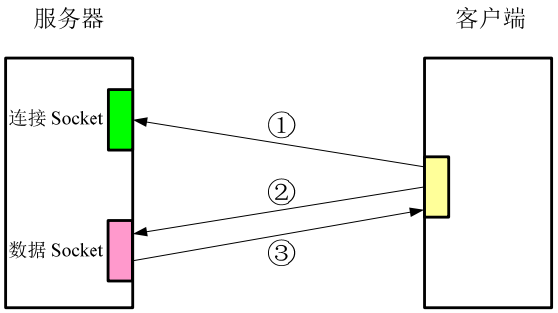

图 2-8 使用流式 socket 的通信模式

服务程序在收到某一客户程序的连接请求后，并不是利用对外发布的那个端口号建立与该客户程序的连接，而是另外分配一个新的端口号建立与客户程序之间的连接，原端口号仍用于监听其他客户程序的连接请求。这一通信模式保证了服务程序对外公布的端口号仅用于处理客户程序要求建立连接的请求，不会因为该端口因长期处理客户程序的服务请求而导致其他客户程序的阻塞。

针对这一通信模式，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。

(2) 客户端编程模式

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。

Socket 类提供了多种重载的构造方法以方便在应用程序中创建 Socket 类的实例，这些构造方法包括：

```
Socket()
Socket(String host, int port) throws UnknownHostException, IOException
Socket(InetAddress addr, int port) throws IOException
Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException
Socket(InetAddress addr, int port, InetAddress localAddr, int localPort) throws IOException
Socket(String host, int port, boolean isStream) throws IOException
Socket(InetAddress host, int port, boolean isStream) throws IOException
```

其中，参数 host 和 addr 用于指定远程主机，port 用于指定远程主机的端口号；参数 localAddr 和 localPort 分别指定新 socket 绑定的本地主机名和端口号；参数 isStream 为 true 表示创建一个面向连接的 socket（又称 TCP socket），否则创建一个面向数据报的 socket（又称 UDP socket），不提供该参数时的默认值为 true。

上述无参数的构造方法之所以不会抛出任何异常，是因为创建 Socket 实例时并未建立与远程主机的连接，这表明一个 socket 实例可在创建之后才建立连接。Socket 类提供了如下方法用于连接远程 socket、绑定本地 socket 或检查 socket 的连接和绑定状态：

```
// 连接远程 socket（注意此时采用的术语是“连接”）
void connect(SocketAddress endpoint) throws IOException
void connect(SocketAddress endpoint, int timeout) throws IOException
boolean isConnected()
InetAddress getInetAddress()
int getPort()
```

```
SocketAddress getRemoteSocketAddress()

// 绑定本地 socket（注意此时采用的术语是“绑定”）
void bind(SocketAddress bindpoint) throws IOException
boolean isBound()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()
```

其中，参数 endpoint 用于指定远程主机的主机名与端口号；参数 bindpoint 指定本地主机的主机名与端口号；参数 timeout 指定建立连接的超时限制，该参数为 0 表示无超时约束。

【提示】除上述主要方法外，类 Socket 还提供了 socket 选项（Socket Option，简称 SO）的设置方法（setter）与访问方法（getter）。例如，setSoTimeout()和 getSoTimeout()可用于设置或访问 socket 的超时选项 SO_TIMEOUT，类似地还可设置或访问一个 socket 的 SO_LINGER、SO_SNDBUF、SO_SNDBUF、SO_REUSEADDR、SO_KEEPALIVE、TCP_NODELAY 等选项。

程序 TCPEcho 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 2-3 基于 TCP 通信的单线程客户程序与服务器程序 TCPEcho

EchoClient.java	客户端程序	程序 2-6
EchoServer.java	单线程服务程序	程序 2-7

程序 2-6 客户端程序 EchoClient.java

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

public class EchoClient {
  public static void main(String[] args)throws Exception {
    if (args.length != 2) {
      System.out.println("用法：EchoClient <主机名> <端口号>");
      return ;
    }

    // 建立连接并打开相关联的输入流和输出流
    Socket socket = new Socket(args[0], Integer.parseInt(args[1]));
    System.out.println("当前 socket 信息：" + socket);
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

    // 将控制台输入的字符串发送给服务端，并显示从服务端获取的处理结果
    BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
    String userInput;
    while ((userInput = stdIn.readLine()) != null) {
      out.println(userInput);
      System.out.println("返回：" + in.readLine());
    }
    stdIn.close();
    // 关闭连接
    out.close();
    in.close();
    socket.close();
  }
```

```
}
```

程序 2-6 是客户端程序，建立连接并打开相关联的输入流和输出流 BufferedReader0，将控制台输入的字符串发送给服务端，并显示从服务端获取的处理结果，值得注意的是 BufferedReader(new InputStreamReader(System.in)使用了设计模式的装饰模式。

【提示】装饰模式（Decorator）：动态地为一个对象添加额外责任；在扩展功能时，装饰模式比使用继承机制更灵活。

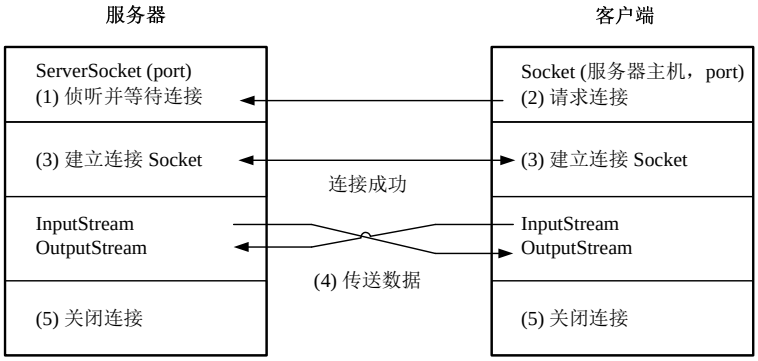

图 2-9 流式 Socket 通信过程

(3) 服务端编程模式

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。

由于 ServerSocket 专用于监听客户程序的连接请求，故这种类型的 socket 与 Socket 的实例有截然不同的语义。类似 Socket 类，ServerSocket 类也提供了多种重载的构造方法以方便在应用程序中创建 ServerSocket 类的实例，包括：

```
ServerSocket() throws IOException
ServerSocket(int port) throws IOException
ServerSocket(int port, int backlog) throws IOException
ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

其中，参数 port 指定服务程序将监听的本地主机端口号，该参数为 0 表示随机选择一个当前未用的端口号；参数 backlog 指定接入连接请求的最大队列长度；参数 bindAddr 指定绑定到本地主机的地址。

上述无参数的构造方法在创建 ServerSocket 实例时并未建立连接，这些实例可调用以下方法建立连接或获取连接的相关信息：

```
void bind(SocketAddress endpoint) throws IOException
void bind(SocketAddress endpoint, int backlog) throws IOException
boolean isBound()
InetAddress getInetAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()
```

其中，参数 endpoint 指定服务程序本地主机的地址，包括主机名与端口号；参数 backlog 指定监听请求的队列长度。在 ServerSocket 类中最重要的方法当属 accept()。该方法建立并返回一个已与客户程序连接的 Socket 实例，其接口为：Socket accept()

程序 2-7 单线程服务程序 EchoServer.java

```
// 单线程服务端程序
import java.net.*;
import java.io.*;

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

      // 监听客户程序的连接请求
      ServerSocket listenSocket = new ServerSocket(Integer.parseInt(args[0]));
      System.out.println("服务程序正在监听端口" + args[0]);
      Socket socket = listenSocket.accept();
      // 从与客户程序的新建连接获取输入流和输出流
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader in = new BufferedReader(new InputStreamReader
         (socket.getInputStream()));
      // 从客户端读取数据，并写回数据的加工结果
      String message;
      while ((message = in.readLine()) != null) {
         System.out.println("收到请求：  " + message);
         out.println(message.toUpperCase());
      }

      // 关闭连接
      out.close();
      in.close();
      socket.close();
      listenSocket.close();
   }
}
```

程序 2-7 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。

2.3.3 多线程服务程序

通过上机运行上一小节的例子程序可发现，应用程序的服务端在同一时刻只能处理一个客户连接。一旦服务程序的 accept()方法被调用，服务程序主线程将持续执行客户程序发来的服务请求，再无其他线程监听服务程序对外发布的端口，导致后续的客户连接请求失败。

一种简单的改进途径是当服务程序处理完一个客户连接后再次循环执行 accept()。这种改进虽然可使服务程序在处理完一个客户程序的所有服务请求后，还可继续建立与下一客户程序的连接并处理其服务请求，但在一个客户程序提交的服务请求需占用服务程序较长时间的情况下，其他客户程序的连接请求将进入队列等待，甚至可能因队列溢出而丢失。

【注意】改进的途径是使用多线程编程方式，让服务程序的主线程执行监听客户程序连接请求的任务，而处理

客户程序服务请求的任务则交由另一个新建的线程负责。

程序 TCPMTEcho 演示了这种简单的多线程处理模型。

表 2-4 基于 TCP 通信的单线程客户程序与多线程服务器程序 TCPMTEcho

EchoClient.java	客户端程序	程序 2-6
MTEchoServer.java	多线程服务程序	程序 2-8

程序 2-8 多线程服务程序 MTEchoServer.java

```
// 采用多线程方式实现的服务端程序
import java.io.*;
import java.net.*;
public class MTEchoServer {
   public static void main(String[] args)throws IOException {
      if (args.length != 1) {
         System.out.println("用法： MTServer <端口号>");
         return ;
      }
      ServerSocket ss = new ServerSocket(Integer.parseInt(args[0]));
      System.out.println("服务程序正在监听端口： " + args[0]);
      for (;;)
         new EchoThread(ss.accept()).start();
   }
}

class EchoThread extends Thread {
   Socket socket;
   EchoThread(Socket s) {
      socket = s;
   }
   public void run() {
      System.out.println("正在为客户程序提供服务: " + socket);
      try {
         PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
         BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
         String message;
         while ((message = in.readLine()) != null) {
            System.out.println(socket + "请求： " + message);
            out.println(message.toUpperCase());
         }
         out.close();
         in.close();
         socket.close();
      } catch (IOException exc) {
         exc.printStackTrace();
      }
   }
}
```

程序 2-8 改进单线程服务程序程序 2-7，EchoThread 类通过继承 Thread 实现多线程机制，当主程序的 ServerSocket 接到客户端的请求后，由 EchoThread 类的 run()方法为每个请求连接的客户端单独建立一个响应处理线程。

【思考】请思考如何利用其它多线程机制实现程序 TCPMTEcho？ 并比较其优缺点。

§ 2.4　应用层协议开发

2.4.1　应用层协议理解

在前二节我们讨论并分析了数据报 Socket 和流式 Socket 编程原理与实现方式，但在我们具体开发应用层的程序时，常常会遇到网络协议的概念。因此，我们这一节有必要详细讨论网络协议（Protocol）的概念和典型应用层协议的开发原理和方法。

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。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，网络协议一般由 RFC（Requests for Comments，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）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。接下来，我们分别介绍如何开发三个应用层协议：Daytime 协议（RFC867）、FTP 协议（RFC959 等）、HTTP 协议（RFC2616 等）。

【提示】关于因特网的正式标准均以 RFC 文档的形式出版，从因特网工程任务组 IETF（Internet Engineering Task Force）的官方网站 http://www.ietf.org/可免费下载这些文档。每一 RFC 文档均以数字标识，数字越大说明文档的内容越新。注意有大量的 RFC 文档并非正式的标准，这些文档仅用于提供信息。

2.4.2　Daytime 协议开发

日期查询协议（Daytime Protocol）是时间传输协议，被广泛的被运行 MS-DOS 和类似的操作系统的小型计算机使用，该协议不指定固定的传输格式，只要求按照 ASCII 标准发送数据。它的作用就是返回当前时间和日期，格式是字符串格式。此协议篇幅很小，容易让初学者理解协议内容和开发过程。

【提示】Daytime 协议规定当客户端发送消息给服务器后，服务器按下面两种格式返回当前时间和日期给客户端：一种格式是：Weekday, Month Day, Year Time-Zone，如：Tuesday, February 22, 1982 17:37:43-PST；另一种格式用于 SMTP 中：dd mmm yy hh:mm:ss zzz，如：02 FEB 82 07:59:01 PST。

基于 UDP 的 Daytime 服务的端口也是 13，不过 UDP 是用数据报传送当前时间的。接收到的数据被忽略。

【思考】请思考如何参考 Daytime 协议分别实现基于 UDP 的 daytime 服务与基于 TCP 的 Daytime 服务

程序 DaytimeUDP 是基于 UDP、符合 Daytime Protocol 的客户端与服务器程序。

(1)　基于 UDP 实现 Daytime 协议

程序 DaytimeUDP 是基于 UDP、符合 Daytime Protocol 的客户端与服务器程序，服务器在端口 13 侦听，一旦有客户端请求服务端就返回 ASCII 形式的日期和时间（接收到的任何数据被忽略），在传送完后关闭连接。

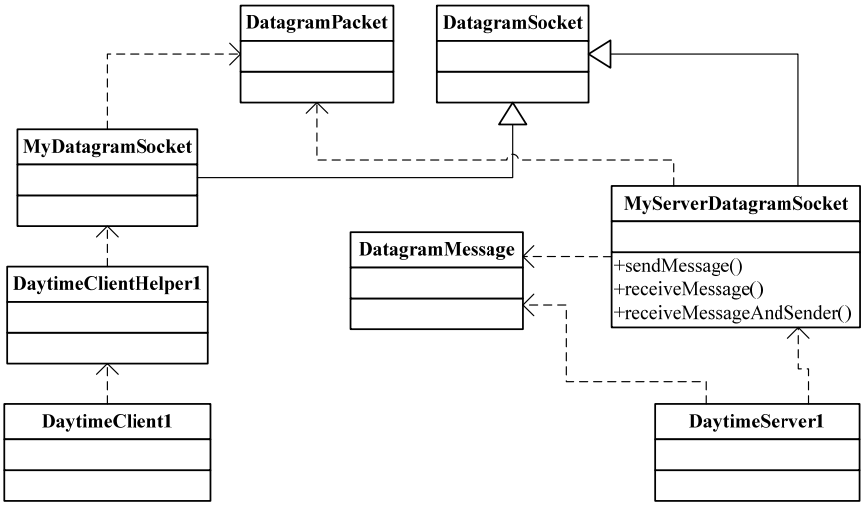

图 2-10 程序 DaytimeUDP 客户端与服务器程序类图

表 2-5 基于 UDP、符合 Daytime Protocol 的客户端与服务器程序 DaytimeUDP

DaytimeServer1.java	基于 UDPDaytime 服务器程序	程序 2-9
MyServerDatagramSocket.java	服务端数据报类	程序 2-10
DatagramMessage.java	数据报消息类	程序 2-11
DaytimeClient1.java	基于 UDPDaytime 客户端程序	程序 2-12
DaytimeClientHelper1.java	客户端帮助类	程序 2-13
MyDatagramSocket.java	客户端数据报类	程序 2-14

程序 2-9 基于 UDPDaytime 服务器程序 DaytimeServer1.java

```
// 基于 UDP 的 Daytime 服务器程序
import java.io.*;
import java.util.Date;

public class DaytimeServer1 {
   public static void main(String[] args) {
            // 服务端缺省端口为 13
      int serverPort = 13;
      if (args.length == 1) serverPort = Integer.parseInt(args[0]);
      try {
        // 创建数据报用于发送与接收
        MyServerDatagramSocket mySocket = new MyServerDatagramSocket(serverPort);
        System.out.println("服务端准备就绪.");
        while (true) {
          DatagramMessage request = mySocket.receiveMessageAndSender();
          System.out.println("收到客户端请求");
          // 接收到消息后，取得服务器时间与日期
          Date timestamp = new Date();
          System.out.println("timestamp sent: " + timestamp.toString());
          // 将服务器时间与日期返回给请求客户端
          mySocket.sendMessage(request.getAddress(), request.getPort(), timestamp.toString());
        }
      } catch (Exception ex) {
        System.out.println("问题:  " + ex);
      }
   }
```

```
}
```

基于 UDPDaytime 服务器程序（程序 2-9），通过 MyServerDatagramSocket 类产生数据报用于发送与接收，当接收到消息后，服务端程序取得服务器时间与日期，并将服务器时间与日期通过 MyServerDatagramSocket 类的 sendMessage()返回给请求客户端。

程序 2-10 服务端数据报类 MyServerDatagramSocket.java

```
// 服务端数据报类
import java.net.*;
import java.io.*;

public class MyServerDatagramSocket extends DatagramSocket {
   static final int MAX_LEN = 100;
   MyServerDatagramSocket(int portNo)throws SocketException {
      super(portNo);
   }
   // 发送消息方法
   public void sendMessage(InetAddress receiverHost, int receiverPort, String message)throws IOException {
      byte[] sendBuffer = message.getBytes();
      DatagramPacket datagram = new DatagramPacket(sendBuffer, sendBuffer.length, receiverHost, receiverPort);
      this.send(datagram);
   }
   // 接收消息方法
   public String receiveMessage()throws IOException {
      byte[] receiveBuffer = new byte[MAX_LEN];
      DatagramPacket datagram = new DatagramPacket(receiveBuffer, MAX_LEN);
      this.receive(datagram);
      String message = new String(receiveBuffer);
      return message;
   }
   // 接收消息后再返回消息方法
   public DatagramMessage receiveMessageAndSender()throws IOException {
      byte[] receiveBuffer = new byte[MAX_LEN];
      DatagramPacket datagram = new DatagramPacket(receiveBuffer, MAX_LEN);
      this.receive(datagram);
      DatagramMessage returnVal = new DatagramMessage();
      returnVal.putVal(new String(receiveBuffer), datagram.getAddress(), datagram.getPort());
      return returnVal;
   }
}
```

程序 2-10 是服务端数据报类，MyServerDatagramSocket 继承 DatagramSocket，实现发送消息方法 sendMessage()、接收消息方法 receiveMessage()、以及接收再返回消息方法 receiveMessageAndSender()，receiveMessageAndSender()返回 DatagramMessage 对象。

程序 2-11 数据报消息类 DatagramMessage.java

```
// 数据报消息类
import java.net.*;

public class DatagramMessage {
   private String message;
   private InetAddress senderAddress;
   private int senderPort;
   public void putVal(String message, InetAddress addr, int port) {
```

```
    this.message = message;
    this.senderAddress = addr;
    this.senderPort = port;
  }
  public String getMessage() {
    return this.message;
  }
  public InetAddress getAddress() {
    return this.senderAddress;
  }
  public int getPort() {
    return this.senderPort;
  }
}
```

程序 2-11 是数据报消息类 DatagramMessage，封装有三个私有成员，message 是传送的消息，senderAddress 是发送方主机地址，senderPort 是发送方端口。

程序 2-12 基于 UDPDaytime 客户端程序 DaytimeClient1.java

```
// 基于 UDP 的 Daytime 客户端程序
import java.io.*;

public class DaytimeClient1 {
  public static void main(String[] args) {
    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(is);
    try {
      System.out.println("请输入服务器主机名： ");
      String hostName = br.readLine();
      if (hostName.length() == 0) hostName = "localhost";
      System.out.println("请输入服务器端口： ");
      String portNum = br.readLine();
      if (portNum.length() == 0) portNum = "13";
      System.out.println(
        // 获取服务端时间
        DaytimeClientHelper1.getTimestamp(hostName, portNum));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

程序 2-12 是基于 UDP 协议的 Daytime 客户端程序，通过命令行获得服务端地址与端口中，再通过 getTimestamp()方法向服务端发送请求，并将响应显示在客户端的界面上。

程序 2-13 客户端帮助类 DaytimeClientHelper1.java

```
// 客户端帮助类
import java.net.*;

public class DaytimeClientHelper1 {
  public static String getTimestamp(String hostName, String portNum) {
    String timestamp = "";
    try {
      InetAddress serverHost = InetAddress.getByName(hostName);
```

```
        int serverPort = Integer.parseInt(portNum);
        MyDatagramSocket mySocket = new MyDatagramSocket();
        mySocket.sendMessage(serverHost, serverPort, "");
        timestamp = mySocket.receiveMessage();
        mySocket.close();
      } catch (Exception ex) {
        System.out.println("问题: " + ex);
      }
      return timestamp;
    }
}
```

程序 2-13 是客户端帮助类，首先通过 MyDatagramSocket 类创建数据报，并由 MyDatagramSocket 类的 sendMessage()方法向服务端发送请求，由 mySocket.receiveMessage()方法接收从服务端返回消息。

程序 2-14 客户端数据报类 MyDatagramSocket.java

```
// 客户端数据报类
import java.net.*;
import java.io.*;

public class MyDatagramSocket extends DatagramSocket {
  static final int MAX_LEN = 100;
  MyDatagramSocket()throws SocketException {
    super();
  }
  MyDatagramSocket(int portNo)throws SocketException {
    super(portNo);
  }
  // 发送消息方法
  public void sendMessage(InetAddress receiverHost, int receiverPort, String
    message)throws IOException {
    byte[] sendBuffer = message.getBytes();
    DatagramPacket datagram = new DatagramPacket(sendBuffer,
      sendBuffer.length, receiverHost, receiverPort);
    this.send(datagram);
  }
  // 接收消息方法
  public String receiveMessage()throws IOException {
    byte[] receiveBuffer = new byte[MAX_LEN];
    DatagramPacket datagram = new DatagramPacket(receiveBuffer, MAX_LEN);
    this.receive(datagram);
    String message = new String(receiveBuffer);
    return message;
  }
}
```

程序 2-14 是客户端数据报类，MyDatagramSocket 类继承 DatagramSocket，sendMessage()方法的参数包括接收方地址 receiverHost、接收方端口 receiverPort 与消息 message，通过 DatagramPacket 类的实例对象发送到服务端。receiveMessage()方法接收服务端响应消息。

(2) 基于 TCP 实现 Daytime 协议

程序 DaytimeTCP 是基于 TCP、符合 Daytime Protocol 的客户端与服务器程序，服务器在

TCP 端口 13 侦听，一旦有连接建立就返回 ASCII 形式的日期和时间（接收到的任何数据被忽略），在传送完后关闭连接。

表 2-6 基于 TCP、符合 Daytime Protocol 的客户端与服务器程序 DaytimeTCP

DaytimeServer2.java	基于 TCPDaytime 服务器程序	程序 2-15
MyStreamSocket.java	服务端流式 Socket 类	程序 2-16
DaytimeClient2.java	基于 TCPDaytime 客户端程序	程序 2-17
DaytimeClientHelper2.java	客户端帮助类	程序 2-18

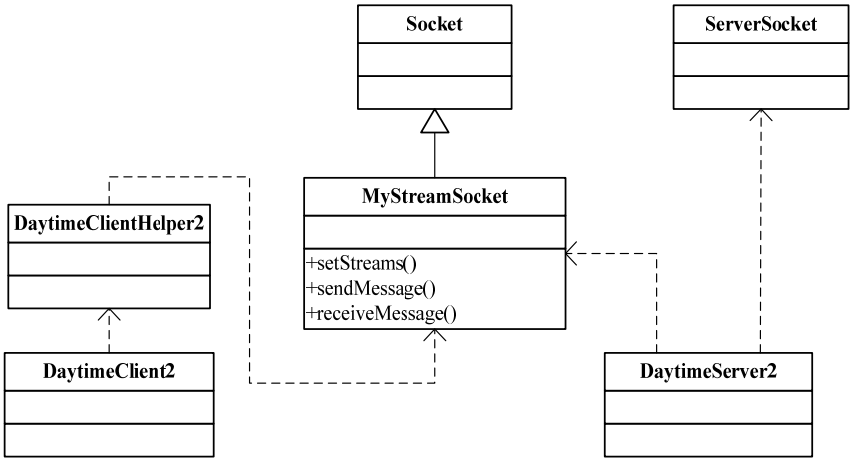

图 2-11 程序 DaytimeTCP 客户端与服务器程序类图

程序 2-15 基于 Daytime 服务器程序 DaytimeServer2.java

```
// 基于 Daytime 服务器程序
import java.io.*;
import java.net.*;
import java.util.Date;

public class DaytimeServer2 {
  public static void main(String[] args) {
                    // 服务端缺省端口为 13
    int serverPort = 13;
    if (args.length == 1) serverPort = Integer.parseInt(args[0]);
    try {
      // 创建流 Socket
      ServerSocket myConnectionSocket = new ServerSocket(serverPort);
      System.out.println("服务端准备就绪.");
      while (true) {
                    // 等待连接请求
        System.out.println("等待客户端请求连接");
        MyStreamSocket myDataSocket = new MyStreamSocket(myConnectionSocket.accept());
        System.out.println("客户端已经建立连接");
        Date timestamp = new Date();
        System.out.println("timestamp sent: " + timestamp.toString());
        // 响应客户端
        myDataSocket.sendMessage(timestamp.toString());
        myDataSocket.close();
      }
    } catch (Exception ex) {
      ex.printStackTrace();
```

```
    }
  }
}
```

程序 2-15 是基于 TCPDaytime 服务器程序，首先通过 ServerSocket 类的实例化产生流 Socket，并由 MyStreamSocket 类的实例对象的 accept()方法监听客户端请求，等待客户端连接。当收到客户端请求后，由 sendMessage()方法返回消息给客户端。

程序 2-16 服务端流式 Socket 类 MyStreamSocket.java

```
// 服务端流式 Socket 类
import java.net.*;
import java.io.*;

public class MyStreamSocket extends Socket {
  private Socket socket;
  private BufferedReader input;
  private PrintWriter output;

  MyStreamSocket(InetAddress acceptorHost, int acceptorPort)throws
    SocketException, IOException {
    socket = new Socket(acceptorHost, acceptorPort);
    setStreams();
  }
  MyStreamSocket(Socket socket)throws IOException {
    this.socket = socket;
    setStreams();
  }
  private void setStreams()throws IOException {
    // 创建数据输入流
    InputStream inStream = socket.getInputStream();
    input = new BufferedReader(new InputStreamReader(inStream));
    // 创建数据输出流
    OutputStream outStream = socket.getOutputStream();
    output = new PrintWriter(new OutputStreamWriter(outStream));
  }
  // 发送消息方法
  public void sendMessage(String message)throws IOException {
    output.println(message);
    output.flush();
  }
  // 接收消息方法
  public String receiveMessage()throws IOException {
    String message = input.readLine();
    return message;
  }
}
```

程序 2-16 是服务端流式 Socket 类，MyStreamSocket 类继承 Socket 类，setStreams()方法得到数据输入流 InputStream 与输出流 OutputStream，sendMessage()方法发送数据流，receiveMessage() 接收数据流。

程序 2-17 基于 Daytime 客户端程序 DaytimeClient2.java

```
// 基于 TCP 的 Daytime 客户端程序
import java.io.*;
```

```
public class DaytimeClient2 {
  public static void main(String[] args) {
    InputStreamReader is = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(is);
    try {
      System.out.println("请输入服务器主机名： ");
      String hostName = br.readLine();
      if (hostName.length() == 0) hostName = "localhost";
      System.out.println("请输入服务器端口： ");
      String portNum = br.readLine();
      if (portNum.length() == 0) portNum = "13";
      System.out.println(
        // 获取服务端时间
        DaytimeClientHelper2.getTimestamp(hostName, portNum));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

程序 2-17 是基于 TCPDaytime 客户端程序，客户端程序从命令行取得服务端的地址与端口，通过客户端帮助类 DaytimeClientHelper2 的 getTimestamp()方法向服务端发送请求并接收响应消息。

程序 2-18 客户端帮助类 DaytimeClientHelper2.java

```
// 客户端帮助类
import java.net.*;

public class DaytimeClientHelper2 {
  public static String getTimestamp(String hostName, String portNum)throws
    Exception {
    String timestamp = "";
    InetAddress serverHost = InetAddress.getByName(hostName);
    int serverPort = Integer.parseInt(portNum);
    MyStreamSocket mySocket = new MyStreamSocket(serverHost, serverPort);
    timestamp = mySocket.receiveMessage();
    mySocket.close();
    return timestamp;
  }
}
```

程序 2-18 客户端帮助类创建 MyStreamSocket 类的实例对象 mySocket，由实例对象的 receiveMessage()方法接收服务端的响应消息。

2.4.3 FTP 协议开发

文件传输协议 FTP（File Transfer Protocol）是一个用于在两台装有不同操作系统的计算机之间传输计算机文件的软件标准，工作于应用层。FTP 服务一般运行在 20 和 21 两个端口。端口 20 用于在客户端和服务器之间传输数据流，而端口 21 用于传输控制流。当数据通过数据流传输时，控制流处于空闲状态。（如 图 2-12 所示）

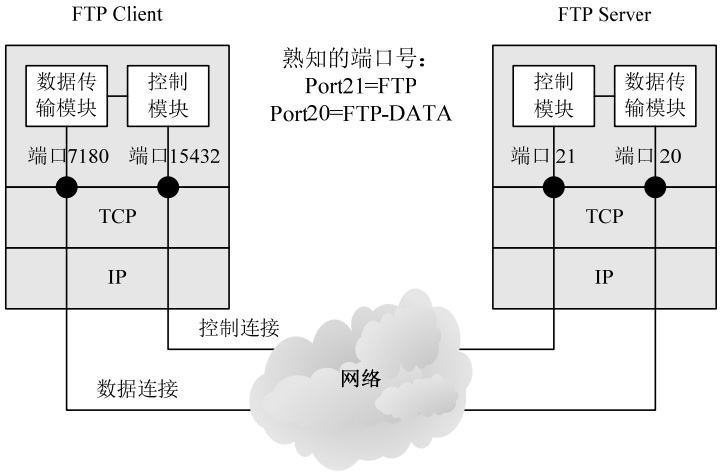

图 2-12 FTP协议通信过程

FTP 协议有两种工作方式：PORT 方式和 PASV 方式，即为主动模式和被动式。主动模式要求客户端和服务器端同时打开并且监听一个端口以建立连接，主动模式的连接过程是：客户端向服务器的 FTP 端口（默认是 21）发送连接请求，服务器接受连接，建立一条命令链路。当需要传送数据时，服务器从 20 端口向客户端的空闲端口发送连接请求，建立一条数据链路来传送数据。

在主动模式下，客户端可以由于安装了防火墙会产生一些问题，所以创立了被动模式。被动模式只要求服务器端产生一个监听相应端口的进程，这样就可以绕过客户端安装了防火墙的问题。被动模式的连接过程是：客户端向服务器的 FTP 端口（默认是 21）发送连接请求，服务器接受连接，建立一条命令链路。当需要传送数据时，客户端向服务器的空闲端口发送连接请求，建立一条数据链路来传送数据。

【提示】一个主动模式的 FTP 连接建立要遵循以下步骤：客户端打开一个随机的端口（端口号大于 1024，在这里，称它为 x），同时一个 FTP 进程连接至服务器的 21 号命令端口。此时，源端口为随机端口 x，在客户端，远程端口为 21。客户端开始监听端口（x+1），同时向服务器发送一个端口命令（通过服务器的 21 号命令端口），此命令告诉服务器客户端正在监听的端口号并且已准备好从此端口接收数据，这个端口就是数据端口。服务器打开 20 号源端口并且建立和客户端数据端口的连接。此时，源端口为 20，远程数据端口为（x+1）。客户端通过本地的数据端口建立一个和服务器 20 号端口的连接，然后向服务器发送一个应答，告诉服务器它已经建立好了一个连接。

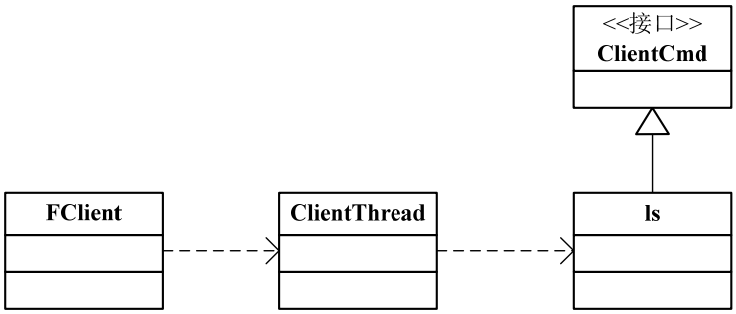

图 2-13 符合FTP协议的客户端程序类结构图

表 2-7 符合 FTP 协议的客户端程序 FTPClient

ClientCmd.java	客户端命令抽象接口	程序 2-19
FClient.java	客户端主程序	程序 2-20
ClientThread.java	客户端线程类	程序 2-21

ls.java	客户端浏览目录命令	程序 2-22

程序 2-19 客户端命令抽象接口 ClientCmd.java

```
import java.net.Socket;
public abstract class ClientCmd {
        public abstract String func(Socket s,String[] cmd);
}
```

程序 2-19 是客户端命令抽象接口，使用设计模式中的抽象工厂模式，从而可以方便地扩展 FTP 客户端命令。

【提示】抽象工厂模式（Abstract Factory）：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

程序 2-20 客户端主程序 FClient.java

```
// FTP 客户端主程序
import java.net.*;
import java.util.Scanner;

public class FClient {
  public static void main(String[] args) {
    System.out.println("欢迎使用 FTP 客户端");
    while (true) {
      System.out.print("ftp>");
      Scanner in = new Scanner(System.in);
      String input = in.nextLine();
      String[] cmd = input.split(" ");
      if (cmd[0].equalsIgnoreCase("open")) {
        try {
          InetAddress addr = InetAddress.getByName(new String(cmd[1]));
          int num = Integer.parseInt(cmd[2]);
          ClientThread ct = new ClientThread();
          ct.testConnection(addr, num);
          break;
        } catch (UnknownHostException e) {
          System.out.println("500 can not found the host!");
        } catch (Exception e) {
          System.out.println("500 incorrect command!");
          continue;
        }
      } else {
        System.out.println("you should open the host first!");
        System.out.println("open [host address] [port]");
      }
    }
  }
}
```

程序 2-20 是客户端主程序，FClient 从命令行读入打开 FTP 服务器命令 open，以及服务端地址，通过 ClientThread 类创建客户端线程，并利用 ClientThread 类 testConnection()方法向服务端建立连接。

程序 2-21 客户端线程类 ClientThread.java 例程

```
// 客户端线程类
import java.io.*;
import java.net.*;
```

```
import java.util.Scanner;

public class ClientThread extends Thread {
   private static Socket connectToServer;
   private Thread thread;
   public void testConnection(InetAddress add, int port) {
      try {
         connectToServer = new Socket(add, port);
         System.out.println("200 connect to " + connectToServer.getInetAddress() + " server success!");
      } catch (IOException e) {
         System.out.println("connection incorrect!");
      }
      thread = new Thread(this);
      thread.start();
   }
   public void run() {
      while (true) {
         try {
            System.out.print("ftp>");
            Scanner i = new Scanner(System.in);
            String temp = i.nextLine();
            String[] cmd = temp.split(" ");
            try {
                // 采用 Java 反射机制获得处理对象
               ClientCmd cm = (ClientCmd)Class.forName("client." + cmd[0]).newInstance();
               String str = cm.func(connectToServer, cmd);
               System.out.println(str);
            } catch (InstantiationException e) {
               System.out.println("Instantiation！ ");
            } catch (IllegalAccessException e) {
               System.out.println("Illegal access！ ");
            } catch (ClassNotFoundException e) {
               System.out.println("Class not found！ ");
            }
         } catch (Exception e) {
            socketClosing();
            break;
         }
      }
   }
   public static void socketClosing() {
      try {
         connectToServer.close();
      } catch (Exception e) {
         System.out.println("close socket fail!");
      }
   }
}
```

程序 2-21 是客户端线程类，通过继承 Thread 类实现客户端多线程程序。为了提高客户端的可扩展性，采用 Java 反射机制获得处理对象。

程序 2-22 客户端浏览目录命令 ls.java

```
// 处理浏览目录命令
```

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;

public class ls extends ClientCmd {
   private static Socket connectToServer;
   private static DataOutputStream sendData;
   private static DataInputStream receiveData;

   public String func(Socket s, String[] cmd) {
      connectToServer = s;
      String str = null;
      try {
         receiveData = new DataInputStream(connectToServer.getInputStream());
         sendData = new DataOutputStream(connectToServer.getOutputStream());
         sendData.writeUTF(cmd[0]);
         str = receiveData.readUTF();
      } catch (IOException e) {
         e.printStackTrace();
      }
      return str;
   }
}
```

程序 2-22 处理客户端的 ls 命令，用于向服务端发送浏览目录请求，ls 类继承 ClientCmd 类，通过 DataOutputStream 与 DataInputStream 创建输出流与输入流，并将从客户端命令行读入的命令参数发送到服务端。

2.4.4 HTTP 协议开发

超文本传输协议 HTTP（HyperText Transfer Protocol）是专门用于传输 HTML 文档的协议，1996 年发布的 RFC 1945 和 1999 年发布的 RFC 2616 分别定义了 HTTP/1.0 和 HTTP/1.1。HTTP 协议采用典型的“请求－答复”通信模型：客户程序建立与服务程序的连接后，向服务程序发送一个服务请求；服务程序根据请求获取相应的 HTML 文档作为答复送回客户程序，最后关闭连接。（如 图 2-14 所示）

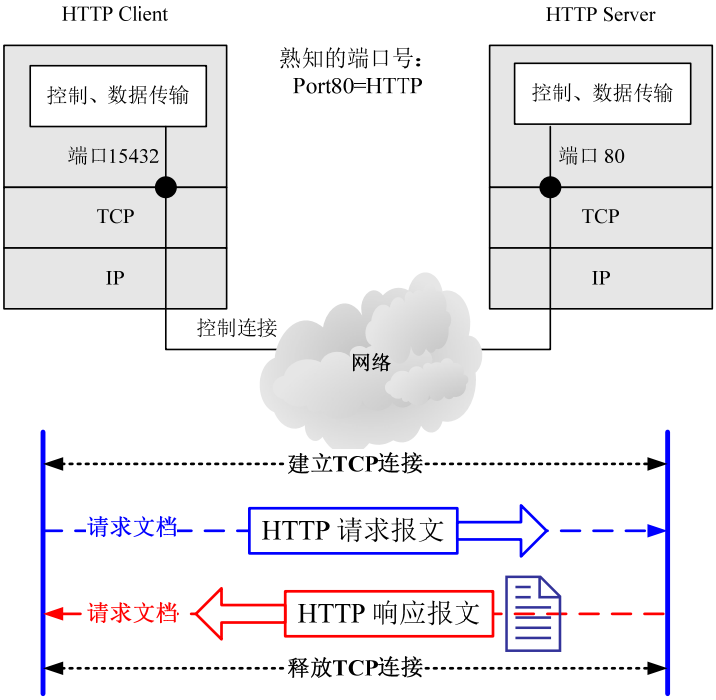

图 2-14 HTTP协议通信过程

由于在客户程序与服务程序的整个会话过程中可能需要建立多个连接（例如，为获取两个不同 URL 指定的资源而分别建立两个 TCP 连接），而不是持久地使用同一 TCP 连接，故 HTTP 是一种无状态的协议。

【提示】在 HTTP 协议的实现中可通过减少 TCP 连接的建立与关闭次数以提高通信效率，HTTP/1.1 的持久连接（persistent connection）机制甚至还保证了客户程序与服务程序之间同一类型的元素持久地使用同一连接，但 HTTP 仍不能被视作一种可保持会话状态的协议。

HTTP 服务程序负责客户程序的服务请求，返回给客户程序的答复既可能是从服务程序本地文件系统中取出的一个 HTML 文件，也可能是调用一个服务端脚本后动态生成的一个 HTML 文档。HTTP 客户程序通常是一个 Web 浏览器，负责格式化和显示取得的 HTML 文并完成与用户的交互。

【提示】一个实用的 Web 浏览器除检索和解释 HTML 文档外，通常还以图形用户界面同时支持文件传输（FTP）、电子邮件（SMTP/POP3/IMAP）等服务。

程序 2-23 演示了一个简单的多线程 HTTP 服务程序。HTTP 协议最简单、最常见的消息是 GET url HTTP/1.1。该服务程序只处理这一种形式的客户请求，将 GET 命令中由 url 指定的文档送回客户程序。HTTP 协议的另一重要消息是 POST 命令，用于向服务程序提交表单中的数据。

程序 2-23 多线程 HTTP 服务程序 HttpServer

```
// 简单的 HTTP 服务器
import java.net.*;
import java.io.*;
import java.util.*;

public class HttpServer {
   public static void main(String[] args)throws IOException {
      ServerSocket ss = new ServerSocket((args.length == 0) ? 80 : Integer.parseInt(args[0]));
      System.out.println("HTTP 服务程序已就绪 ...");
```

```
    while (true) new HttpdThread(ss.accept()).start();
  }
}

class HttpdThread extends Thread {
  Socket socket;
  HttpdThread(Socket s) {
    socket = s;
  }
  public void run() {
    try {
      // 打开与连接绑定的输入流和输出流
      BufferedReader in = new BufferedReader(new InputStreamReader (socket.getInputStream(), "GBK"));
      OutputStream out = socket.getOutputStream();
      // 读取客户请求
      String request = in.readLine();
      System.out.println("收到请求: " + request);
      // 根据 HTTP 协议分析客户请求内容（只处理 GET 命令）
      StringTokenizer st = new StringTokenizer(request);
      if ((st.countTokens() >= 2) && st.nextToken().equals("GET")) {
        // 从请求中取出文档的文件名，支持默认索引文件
        String filename = st.nextToken();
        if (filename.startsWith("/")) filename = filename.substring(1);
        if (filename.endsWith("/")) filename += "index.html";
        if (filename.equals("")) filename += "index.html";
        try {
          // 读取文件中的内容并写到 socket 的输出流
          InputStream file = new FileInputStream(filename);
          byte[] data = new byte[file.available()];
          file.read(data);
          out.write(data);
          out.flush();
        } catch (FileNotFoundException exc) {
          PrintWriter pout = new PrintWriter(new OutputStreamWriter (out, "GBK"), true);
          pout.println("错误代码 404: 未发现目标！");
        }
      } else {
        PrintWriter pout = new PrintWriter(new OutputStreamWriter(out, "GBK"), true);
        pout.println("错误代码 400: 错误的请求！");
      }
      // 关闭连接
      socket.close();
    } catch (IOException exc) {
      System.out.println("I/O 错误: " + exc);
    }
  }
}
```

假设在上述服务程序的同一子目录中放置一个index.html 文档及其链接的图片或其它资源，并在主机*hostname*（若是本地主机则域名为*localhost*）上启动该服务程序，则在客户机的Microsoft IE 或Netscape Communicator 等浏览器的地址栏输入：

```
    http://hostname
```

```
http://hostname/
http://hostname/index.html
```

等地址均可以超文本、多媒体形式浏览index.html 文档，若输入地址：

```
http://hostname/filepath
```

则可在浏览器中显示*filepath*指定的文件。