9.3 TCP 通信

在 9.2 节中学习了 UDP 通信，本节将学习 TCP 通信。TCP 通信与 UDP 通信一样，也能实现两台计算机之间的通信，但 TCP 通信的两端需要创建 Socket 对象。TCP 通信与 UDP 通信的区别在于，UDP 中只有发送端和接收端，不区分客户端与服务器端，计算机之间可以任意发送数据；而 TCP 通信是严格区分客户端与服务器端的，在通信时，必须先由客户端去连接服务器端才能实现通信，服务器端不可以主动连接客户端，并且服务器端程序需要事先启动，等待客户端的连接。

Java 提供了两个用于实现 TCP 程序的类，一个是 ServerSocket 类，用于表示服务器端；另一个是 Socket 类，用于表示客户端。通信时，首先要创建代表服务器端的 ServerSocket 对象，创建该对象相当于开启一个服务，此服务会等待客户端的连接；然后创建代表客户端的 Socket 对象，使用该对象向服务器端发出连接请求，服务器端响应请求后，两者才建立连接，开始通信。整个通信过程如图 9-13 所示。

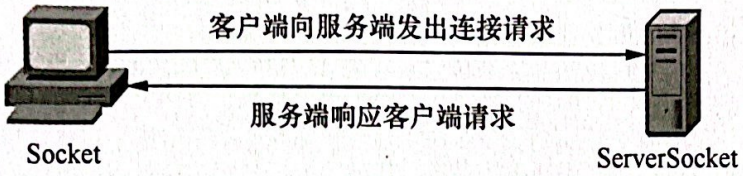

图9-13　Socket和ServerSocket的通信过程

了解了 ServerSocket 类、Socket 类在服务器端与客户端的通信过程后，本节将对 ServerSocket 类和 Socket 类进行详细讲解。

9.3.1 ServerSocket

通过前面的学习可知，在开发 TCP 程序时，首先需要创建服务器端程序。ServerSocket 类的实例对象可以实

现一个服务器端的程序。通过查阅 API 文档可知，ServerSocket 类提供了多种构造方法。下面就对 ServerSocket 的构造方法逐一进行讲解。

（1）ServerSocket（）

ServerSocket 有一个不带参数的默认构造方法。通过该方法创建的 ServerSocket 对象不与任何端口绑定，这样的 ServerSocket 对象所创建的服务器端没有监听任何端口，不能直接使用，还需要继续调用 bind（SocketAddress endpoint）方法将其绑定到指定的端口号上，才可以正常使用。

（2）ServerSocket（int port）

使用该构造方法在创建 ServerSocket 对象时，可以将其绑定到一个指定的端口号上（参数 port 就是端口号）。端口号可以指定为 0，此时系统就会分配一个还没有被其他网络程序使用的端口号。由于客户端需要根据指定的端口号来访问服务器端程序，因此端口号随机分配的情况并不常用，通常都会让服务器端程序监听一个指定的端口号。

（3）ServerSocket（int port, int backlog）

该构造方法是在第二个构造方法的基础上增加了一个 backlog 参数。该参数用于指定在服务器忙时，可以与之保持连接请求的等待客户数量，如果没有指定这个参数，默认为 50。

（4）ServerSocket（int port, int backlog, InetAddress bindAddr）

该构造方法是在第三个构造方法的基础上增加了一个 bindAddr 参数，该参数用于指定相关的 IP 地址。该构造方法的使用适用于计算机上有多块网卡和多个 IP 的情况，使用时可以明确规定 ServerSocket 在哪块网卡或 IP 地址上等待客户的连接请求。显然，对于一般只有一块网卡的情况，就不用专门指定了。

在以上介绍的构造方法中，第二个构造方法是最常使用的。了解了如何通过 ServerSocket 的构造方法创建对象后，下面学习 ServerSocket 的常用方法，如表 9-4 所示。

表 9-4 ServerSocket 的常用方法

方法声明	功能描述
Socket accept（）	该方法用于等待客户端的连接，在客户端连接之前会一直处于阻塞状态，如果有客户端连接，就会返回一个与之对应的 Socket 对象
InetAddress getInetAddress（）	该方法用于返回一个 InetAddress 对象，该对象中封装了 ServerSocket 绑定的 IP 地址
boolean is Closed（）	该方法用于判断 ServerSocket 对象是否为关闭状态，如果是关闭状态则返回 true，反之则返回 false
void bind（SocketAddress endpoint）	该方法用于将 ServerSocket 对象绑定到指定的 IP 地址和端口号，其中参数 endpoint 封装了 IP 地址和端口号

ServerSocket 对象负责监听某台计算机的某个端口号，在创建 ServerSocket 对象后，需要继续调用该对象的 accept（）方法，接收来自客户端的请求。当执行了 accept（）方法之后，服务器端程序会发生阻塞，直到客户端发出连接请求时，accept（）方法才会返回一个 Socket 对象用于与客户端实现通信，程序才能继续向下执行。

9.3.2 Socket

9.3.1 小节讲解了 ServerSocket 对象，它可以实现服务器端程序，但只实现服务器端程序还不能完成通信，此时还需要一个客户端程序与之交互，为此 Java 提供了一个 Socket 类，用于实现 TCP 客户端程序。通过查阅 API 文档可知，Socket 类同样提供了多种构造方法。下面对 Socket 的常用构造方法进行详细讲解。

（1）Socket（）

使用该构造方法在创建 Socket 对象时，并没有指定 IP 地址和端口号，也就意味着只创建了客户端对象，

没有去连接任何服务器。通过该构造方法创建对象后还需调用 connect（SocketAddress endpoint）方法，才能完成与指定服务器端的连接，其中参数 endpoint 用于封装 IP 地址和端口号。

（2）Socket（String host，int port）

使用该构造方法在创建 Socket 对象时，会根据参数去连接在指定地址和端口上运行的服务器程序，其中参数 host 接收的是一个字符串类型的 IP 地址。

（3）Socket（InetAddress address，int port）

该构造方法在使用上与第二个构造方法类似，参数 address 用于接收一个 InetAddress 类型的对象，该对象用于封装一个 IP 地址。

在以上 Socket 的构造方法中，最常用的是第一个构造方法。了解了 Socket 的构造方法后，下面学习 Socket 的常用方法，如表 9-5 所示。

表 9-5　Socket 的常用方法

方法声明	功能描述
int getPort（）	该方法返回一个 int 类型对象，该对象是 Socket 对象与服务器端连接的端口号
InetAddress getLocalAddress（）	该方法用于获取 Socket 对象绑定的本地 IP 地址，并将 IP 地址封装成 InetAddress 类型的对象返回
void close（）	该方法用于关闭 Socket 连接，结束本次通信。在关闭 Socket 之前，应将与 Socket 相关的所有的输入/输出流全部关闭，这是因为一个良好的程序应该在执行完毕时释放所有的资源
InputStream getInputStream（）	该方法返回一个 InputStream 类型的输入流对象，如果该对象是由服务器端的 Socket 返回，就用于读取客户端发送的数据，反之，用于读取服务器端发送的数据
OutputStream getOutputStream（）	该方法返回一个 OutputStream 类型的输出流对象，如果该对象是由服务器端的 Socket 返回，就用于向客户端发送数据，反之，用于向服务器端发送数据

表 9-5 中，getInputStream（）和 getOutputStream（）方法分别用于获取输入流和输出流。当客户端和服务器端建立连接后，数据是以 I/O 流的形式进行交互的，从而实现通信。下面通过一张图描述服务器端和客户端的数据传输，如图 9-14 所示。

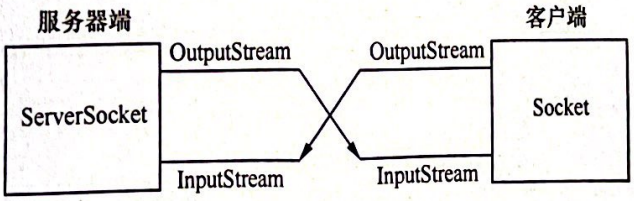

图9-14　服务器端和客户端的数据传输

9.3.3　简单的 TCP 网络程序

通过 9.3.1 小节和 9.3.2 小节的讲解，读者已经了解了 ServerSocket、Socket 类的基本用法。为了让初学者好地掌握这两个类的使用，下面通过一个 TCP 通信的案例来进一步学习这两个类的用法。要实现 TCP 通需要创建一个服务器端程序和一个客户端程序，为了保证数据传输的安全性，首先需要实现服务器端程序。务器端程序实现如文件 9-5 所示。

文件 9-5　Server.java

```
1  import java.io.*;
2  import java.net.*;
3  public class Server {
4      public static void main (String[] args) throws Exception {
5          new TCPServer () .listen () ; // 创建 TCPServer 对象，并调用 listen () 方法
6      }
7  }
```

```
// TCP 服务器端
class TCPServer {
    private static final int PORT = 7788;   // 定义一个端口号
    public void listen () throws Exception { // 定义一个 listen () 方法，抛出异常
        ServerSocket serverSocket = new ServerSocket (PORT) ;
      // 调用 ServerSocket 的 accept () 方法接收数据
        Socket client = serverSocket.accept () ;
        OutputStream os = client.getOutputStream () ;// 获取客户端的输出流
        System.out.println ("开始与客户端交互数据") ;
      // 当客户端连接到服务器端时，向客户端输出数据
        os.write ( ("传智播客欢迎你！") .getBytes () ) ;
        Thread.sleep (5000) ;// 模拟执行其他功能占用的时间
        System.out.println ("结束与客户端交互数据") ;
        os.close () ;
        client.close () ;
    }
}
```

文件 9-5 的运行结果如图 9-15 所示。

在文件 9-5 中，第 9～24 行代码封装了一个 TCP 服务端的方法。其中，第 12 行代码创建 ServerSocket 对象时指定了端口号（7788）；第 14 代码调用 ServerSocket 对象的 accept（）方法用于接收数据；第 15 行代码使用 OutputStream 获取客户端的输出流；第 19 行代码使用线程的 sleep（）方法使线程休眠 5000 毫秒，用于模拟执行其他功能占用的时间；最后在第 21～22 行代码中分别使用 OutputStream 和 Socket 的 close（）方法关闭了 OutputStream 和 Socket。

从图 9-15 的运行结果可以看出，控制台中的光标一直在闪动，这是因为 accept（）方法发生阻塞，程序暂时停止运行，直到有客户端来访问时才会结束这种阻塞状态。这时该方法会返回一个 Socket 类型的对象用于表示客户端，通过该对象获取与客户端关联的输出流并向客户端发送信息，同时执行 Thread.sleep（5000）语句模拟服务器执行其他功能占用的时间。最后，调用 Socket 对象的 close（）方法结束通信。

文件 9-5 完成了服务器端程序的编写，下面编写客户端程序，如文件 9-6 所示。

文件 9-6　Client.java

```
import java.io.*;
import java.net.*;
public class Client {
    public static void main (String[] args) throws Exception {
        new TCPClient () .connect () ;// 创建 TCPClient 对象，并调用 connect () 方法
    }
}
//TCP 客户端
class TCPClient {
    private static final int PORT = 7788; // 服务器端的端口号
    public void connect () throws Exception {
        //创建一个 Socket 并连接到给出地址和端口号的计算机
        Socket client = new Socket (InetAddress.getLocalHost () , PORT) ;
        InputStream is = client.getInputStream () ;   // 得到接收数据的流
        byte[] buf = new byte[1024];   // 定义 1024 个字节数组的缓冲区
        int len = is.read (buf) ;                    // 将数据读到缓冲区中
        System.out.println (new String (buf, 0, len) ) ;// 将缓冲区中的数据输出
        client.close () ;   // 关闭 Socket 对象，释放资源
    }
}
```

文件 9-6 的运行结果如图 9-16 所示。

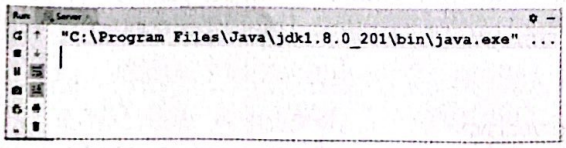

图9-15　文件9-5的运行结果

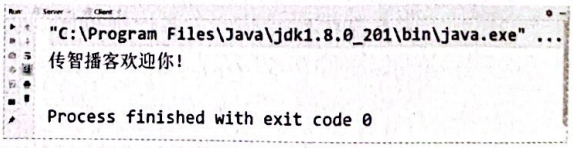

图9-16　文件9-6的运行结果

在文件 9-6 中，第 9～20 行代码封装了一个 TCP 客户端的方法。其中，第 13 行代码创建了一个 Socket 并连接到指定地址和端口号的计算机；第 14 行代码使用 InputStream 接收得到的数据流；第 15 行代码定义 1024 个字节数组的缓冲区；第 16 行代码将 InputStream 接收到的数据读到缓冲区中；最后在第 18 行代码中

使用 Socket 的 close ( ) 方法关闭 Socket。

在客户端创建的 Socket 对象与服务器端建立连接后，通过 Socket 对象获得输入流读取服务器端发来的数据，并打印出图 9-16 所示的结果。同时文件 9-5 中的服务器端程序会结束阻塞状态，并在控制台中打印出“开始与客户端交互数据”，然后向客户端发出数据“传智播客欢迎你!”，在休眠 5 秒后会在控制台打印出“结束与客户端交互数据”，此时，本次通信才结束，如图 9-17 所示。

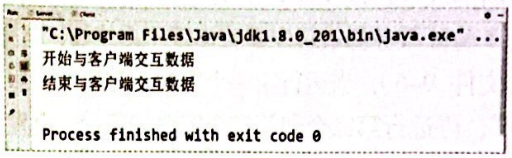

图9-17　服务器端与客户端交互时服务器端的运行结果

9.3.4　多线程的 TCP 网络程序

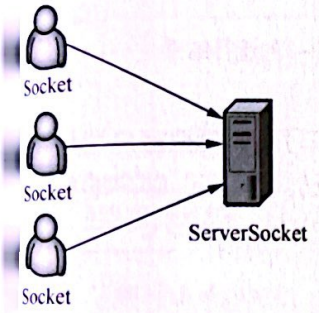

图9-18　多个用户访问服务器

在文件 9-5 和文件 9-6 中，分别实现了服务器端程序和客户端程序，当一个客户端程序请求服务器端时，服务器端就会结束阻塞状态，完成程序的运行。实际上，很多服务器端程序都是允许被多个应用程序访问的，例如，门户网站可以被多个用户同时访问，因此服务器都是多线程的。下面就通过一个图例来表示多个用户访问同一个服务器，如图 9-18 所示。

在图 9-18 中，服务器端为每个客户端创建一个对应的 Socket，并且开启一个新的线程使两个 Socket 建立专线进行通信。下面根据图 9-18 所示的通信方式对文件 9-5 的服务器端程序进行改进，如文件 9-7 所示。

文件 9-7　Server.java

```
1  import java.io.*;
2  import java.net.*;
3  public class Server {
4       public static void main (String[] args) throws Exception {
5            new TCPServer () .listen () ;   // 创建 TCPServer 对象，并调用 listen () 方法
6       }
7  }
8  // TCP 服务器端
9  class TCPServer {
10      private static final int PORT = 7788; // 定义一个静态常量作为端口号
11      public void listen () throws Exception {
12          // 创建 ServerSocket 对象，监听指定的端口
13          ServerSocket serverSocket = new ServerSocket (PORT) ;
14        // 使用 while 循环不停地接收客户端发送的请求
15          while (true) {
16            // 调用 ServerSocket 的 accept () 方法与客户端建立连接
17              final Socket client = serverSocket.accept () ;
18            // 下面的代码用来开启一个新的线程
19              new Thread () {
20                   public void run () {
21                        OutputStream os;  // 定义一个输出流对象
22                        try {
23                             os = client.getOutputStream () ;  // 获取客户端的输出流
24                             System.out.println ("开始与客户端交互数据") ;
25                             os.write ( ("传智播客欢迎你! ") .getBytes () ) ;
26                             Thread.sleep (5000) ;  // 使线程休眠 5000 毫秒
27                             System.out.println ("结束与客户端交互数据") ;
28                             os.close () ;                    // 关闭输出流
29                             client.close () ;                // 关闭 Socket 对象
30                        } catch (Exception e) {
31                             e.printStackTrace () ;
32                        }
33                   };
34              }.start () ;
35          }
36      }
37 }
```

在文件 9-7 中，第 9～37 行代码使用多线程的方式创建了一个服务器端程序。其中，第 15～35 行代码

通过在 while 循环中调用 accept（）方法，不停地接收客户端发送的请求，当与客户端建立连接后，就会启一个新的线程，该线程会去处理客户端发送的数据，而主线程仍处于继续等待状态。

为了验证服务器端程序是否实现了多线程，首先运行服务器端程序（文件 9-7），之后运行 3 个客户程序（文件 9-6），当运行第一个客户端程序时，服务器端马上就进行数据处理，打印出“开始与客户端互数据”，再运行第二个和第三个客户端程序，会发现服务器端也立刻做出回应，3 个客户端会话结束后别打印各自的结束信息，如图 9-19 所示。这说明通过多线程的方式，可以实现多个用户对同一个服务器端程序的访问。

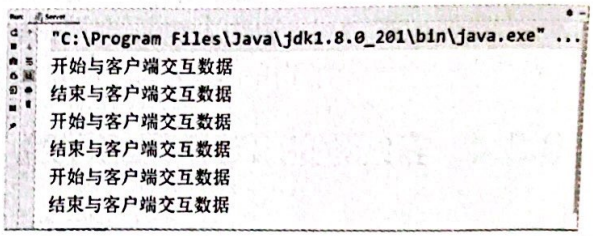

图9-19 文件9-7的运行结果

【案例 9-2】 字符串反转

在使用软件或浏览网页时，总会查询一些数据，查询数据的过程其实就是客户端与服务器交互的过程。用户（客户端）将查询信息发送给服务器，服务器接收到查询消息后进行处理，将查询结果返回给用户（客户端）。本案例要求编写一个模拟客户端与服务器交互的程序，客户端向服务器传递一个字符串（键盘录入），服务器将字符串反转后写回，客户端再次接收到的是反转后的字符串。本案例要求使用多线程与 TCP 通信相关知识完成数据交互。

【案例 9-3】 上传文件

在日常工作和生活中，我们总会将工作成果或生活照片等上传到某一个软件，其实这个上传过程就是将数据保存到了软件服务端。本案例要求编写一个程序模拟向服务端上传文件，在本地机器中输入一个路径，将该路径下的文件上传到服务端 D 盘中名称为 upload 的文件夹中。在上传时，把客户端的 IP 地址加上 count 标识作为上传后文件的名称，即 IP（count）的形式。其中，count 随着文件的增多而增大，如 127.0.0.（1）.jpg、127.0.0.（2）.jpg。本案例要求使用多线程与 TCP 通信相关知识完成数据上传。

9.4 本章小结

本章讲解了 Java 网络编程的相关知识。首先简要介绍了网络通信协议的相关知识，然后着重介绍了与 UDP 网络编程相关的 DatagramPacket 类、DatagramSocket 类，并通过两个案例实现了 UDP 通信。最后讲解了 TCP 网络编程中相关的 ServerSocket 类、Socket 类，并通过两个案例实现了 TCP 通信。通过学习本章的内容，读者能够了解网络编程相关知识，并能够掌握 UDP 网络程序和 TCP 网络程序的编写。

9.5 本章习题

本章习题可以扫描右侧二维码查看。