



JavaTCP/IP Socket编程

极客学院出版

前言

本书由浅入深，全面讲解了 Java Socket 方面的网络编程知识，包括 TCP、UDP、自定义协议、协议成帧、解析、多线程、线程池、NIO、死锁、Socket 套接字的底层实现机制等。通过本书的学习，能够让读者了解通过 Socket 技术实现服务器与客户端的通信，全面了解 Java 网络编程中的技术难点。

适用人群

本书适合作为 Java Socket 编程的入门教程，可供从事网络编程的技术人员参考。

学习前提

在学习本书之前，我们假定你已经对 TCP/UDP 协议及 Java 编程有了一定的了解。

目录

前言	1
第 1 章 Socket 简介	3
第 2 章 Java TCP Socket 编程	6
第 3 章 UDP Socket 编程	12
第 4 章 应用程序协议中消息的成帧与解析	19
第 5 章 构建和解析自定义协议消息	24
第 6 章 基于线程池的 TCP 服务器	32
第 7 章 Socket 通信中由 read 返回值造成的死锁问题	39
第 8 章 Java NIO Socket VS 标准 IO Socket	50
第 9 章 基于 NIO 的 TCP 通信	53
第 10 章 深入剖析 Socket——数据传输的底层实现	62
第 11 章 深入剖析 Socket——TCP 通信中由于底层队列填满而造成的死锁问题	67
第 12 章 深入剖析 Socket——TCP 套接字的生命周期	72



Socket 简介



协议简介

1. 协议相当于相互通信的程序间达成的一种约定，它规定了分组报文的结构、交换方式、包含的意义以及怎样对报文所包含的信息进行解析。
2. TCP/IP 协议族有 IP 协议、TCP 协议和 UDP 协议。
3. TCP 协议和 UDP 协议使用的地址叫做端口号，用来区分同一主机上的不同应用程序。TCP 协议和 UDP 协议也叫端到端传输协议，因为他们将数据从一个应用程序传输到另一个应用程序，而 IP 协议只是将数据从一个主机传输到另一个主机。
4. 在 TCP/IP 协议中，有两部分信息用来确定一个指定的程序：互联网地址和端口号：其中互联网地址由 IP 协议使用，而附加的端口地址信息则由传输协议（TCP 或 UDP 协议）对其进行解析。
5. 现在 TCP/IP 协议族中的主要 socket 类型为流套接字（使用 TCP 协议）和数据报套接字（使用 UDP 协议），其中通过数据报套接字，应用程序一次只能发送最长 65507 个字节长度的信息。
6. 一个 TCP/IP 套接字由一个互联网地址，一个端对端协议（TCP 协议或 UDP 协议）以及一个端口号唯一确定。
7. 每个端口都标识了一台主机上的一个应用程序，实际上，一个端口确定了一个主机上的一个套接字。主机中的多个程序可以同时访问同一个套接字，在实际应用中，访问相同套接字的不同程序通常都属于一个应用（如 web 服务程序的多个副本），但从理论上讲，它们可以属于不同的应用。

基本套接字

1. 编写 TCP 客户端程序，在实例化 Socket 类时，要注意，底层的 TCP 协议只能处理 IP 协议，如果传递的第一个参数是主机名字而不是你 IP 地址，Socket 类具体实现的时候会将其解析成相应的地址，若因为某些原因连接失败，构造函数会抛出一个 IOException 异常。
2. TCP 协议读写数据时，read()方法在没有可读数据时会阻塞等待，直到有新的数据可读。另外，TCP 协议并不能确定在 read()和 write()方法中所发送信息的界限，接收或发送的数据可能被 TCP 协议分割成了多个部分。
3. 编写 TCP 服务器端的程序将在 accept()方法处阻塞，以等待客户端的连接请求，一旦取得连接，便要为每个客户端的连接建立一个 Socket 实例来进行数据通信。

4. 在 UDP 程序中，创建 DatagramPacket 实例时，如果没有指定远程主机地址和端口，则该实例用来接收数据（尽管可以调用 setXXX()等方法指定），如果指定了远程主机地址和端口，则该实例用来发送数据。
5. UDP 程序在 receive()方法处阻塞，直到收到一个数据报文或等待超时。由于 UDP 协议是不可靠协议，如果数据报在传输过程中发生丢失，那么程序将会一直阻塞在 receive()方法处，这对客户端来说是肯定不行的，为了避免这个问题，我们在客户端使用 DatagramSocket 类的 setSoTimeout()方法来制定 receive()方法的最长阻塞时间，并指定重发数据报的次数，如果每次阻塞都超时，并且重发次数达到了设置的上限，则关闭客户端。
6. UDP 服务器为所有通信使用同一套接字，这点与 TCP 服务器不同，TCP 服务器则为每个成功返回的 accept()方法创建一个新的套接字。
7. 在 UDP 程序中，DatagramSocket 的每一次 receive()调用最多只能接收调用一次 send()方法所发送的数据，而且，不同的 receive()方法调用绝对不会返回同一个 send()方法所发送的数据。
8. 在 UDP 套接字编程中，如果 receive()方法在一个缓冲区大小为 n 的 DatagramPacket 实例中调用，而接受队列中的第一个消息长度大于 n，则 receive()方法只返回这条消息的前 n 个字节，超出的其他字节部分将自动被丢弃，而且也没有任何消息丢失的提示。因此，接受者应该提供一个足够大的缓存空间的 DatagramPacket 实例，以完整地存放调用 receive()方法时应用程序协议所允许的最大长度的消息。一个 DatagramPacket 实例中所运行传输的最大数据量为 65507 个字节，即 UDP 数据报文所能负载的最多数据，因此，使用一个有 65600 字节左右缓存数组的数据总是安全的。
9. 在 UDP 套接字编程中，每一个 DatagramPacket 实例都包含一个内部消息长度值，而该实例一接收到新消息，这个长度值便可能改变（以反映实际接收的消息的字节数）。如果一个应用程序使用同一个 DatagramPacket 实例多次调用 receive()方法，每次调用前就必须显式地将消息的内部长度重置为缓冲区的实际长度。
10. 另一个潜在问题的根源是 DatagramPacket 类的 getData()方法,该方法总是返回缓冲区的原始大小，忽略了实际数据的内部偏移量和长度信息。



T

2



Java TCP Socket 编程



TCP 的 Java 支持

协议相当于相互通信的程序间达成的一种约定，它规定了分组报文的结构、交换方式、包含的意义以及怎样对报文所包含的信息进行解析，TCP/IP 协议族有 IP 协议、TCP 协议和 UDP 协议。现在 TCP/IP 协议族中的主要 socket 类型为流套接字（使用 TCP 协议）和数据报套接字（使用 UDP 协议）。

TCP 协议提供面向连接的服务，通过它建立的是可靠地连接。Java 为 TCP 协议提供了两个类：Socket 类和 ServerSocket 类。一个 Socket 实例代表了 TCP 连接的一个客户端，而一个 ServerSocket 实例代表了 TCP 连接的一个服务器端，一般在 TCP Socket 编程中，客户端有多个，而服务器端只有一个，客户端 TCP 向服务器端 TCP 发送连接请求，服务器端的 ServerSocket 实例则监听来自客户端的 TCP 连接请求，并为每个请求创建新的 Socket 实例，由于服务端在调用 accept（）等待客户端的连接请求时会阻塞，直到收到客户端发送的连接请求才会继续往下执行代码，因此要为每个 Socket 连接开启一个线程。服务器端要同时处理 ServerSocket 实例和 Socket 实例，而客户端只需要使用 Socket 实例。另外，每个 Socket 实例会关联一个 InputStream 和 OutputStream 对象，我们通过将字节写入套接字的 OutputStream 来发送数据，并通过从 InputStream 来接收数据。

TCP 连接的建立步骤

客户端向服务器端发送连接请求后，就被动地等待服务器的响应。典型的 TCP 客户端要经过下面三步操作：

- 创建一个 Socket 实例：构造函数向指定的远程主机和端口建立一个 TCP 连接；
- 通过套接字的 I/O 流与服务端通信；
- 使用 Socket 类的 close 方法关闭连接。

服务端的工作是建立一个通信终端，并被动地等待客户端的连接。

典型的 TCP 服务端执行如下两步操作：

1. 创建一个 ServerSocket 实例并指定本地端口，用来监听客户端在该端口发送的 TCP 连接请求；
2. 重复执行：
 - 调用 ServerSocket 的 accept（）方法以获取客户端连接，并通过其返回值创建一个 Socket 实例；
 - 为返回的 Socket 实例开启新的线程，并使用返回的 Socket 实例的 I/O 流与客户端通信；通信完成后，使用 Socket 类的 close（）方法关闭该客户端的套接字连接。

TCP Socket Demo

下面给出一个客户端服务端 TCP 通信的 Demo，该客户端在 20006 端口请求与服务端建立 TCP 连接，客户端不断接收键盘输入，并将其发送到服务端，服务端在接收到的数据前面加上“echo”字符串，并将组合后的字符串发回给客户端，如此循环，直到客户端接收到键盘输入“bye”为止。

客户端代码如下：

```
package zyb.org.client;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
import java.net.SocketTimeoutException;

public class Client1 {
    public static void main(String[] args) throws IOException {
        //客户端请求与本机在20006端口建立TCP连接
        Socket client = new Socket("127.0.0.1", 20006);
        client.setSoTimeout(10000);
        //获取键盘输入
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        //获取Socket的输出流，用来发送数据到服务端
        PrintStream out = new PrintStream(client.getOutputStream());
        //获取Socket的输入流，用来接收从服务端发送过来的数据
        BufferedReader buf = new BufferedReader(new InputStreamReader(client.getInputStream()));
        boolean flag = true;
        while(flag){
            System.out.print("输入信息: ");
            String str = input.readLine();
            //发送数据到服务端
            out.println(str);
            if("bye".equals(str)){
                flag = false;
            }else{
                try{
                    //从服务器端接收数据有个时间限制（系统自设，也可以自己设置），超过了这个时间，便会抛出该异常
                    String echo = buf.readLine();
                    System.out.println(echo);
                }catch(SocketTimeoutException e){
                    System.out.println("Time out, No response");
                }
            }
        }
    }
}
```

```

        }
    }
}
input.close();
if(client != null){
    //如果构造函数建立起了连接，则关闭套接字，如果没有建立起连接，自然不用关闭
    client.close(); //只关闭socket，其关联的输入输出流也会被关闭
}
}
}
}

```

服务端需要用到多线程，这里单独写了一个多线程类，代码如下：

```

package zyb.org.server;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;

/**
 * 该类为多线程类，用于服务端
 */
public class ServerThread implements Runnable {

    private Socket client = null;
    public ServerThread(Socket client){
        this.client = client;
    }

    @Override
    public void run() {
        try{
            //获取Socket的输出流，用来向客户端发送数据
            PrintStream out = new PrintStream(client.getOutputStream());
            //获取Socket的输入流，用来接收从客户端发送过来的数据
            BufferedReader buf = new BufferedReader(new InputStreamReader(client.getInputStream()));
            boolean flag = true;
            while(flag){
                //接收从客户端发送过来的数据
                String str = buf.readLine();
                if(str == null || "".equals(str)){
                    flag = false;
                }else{
                    if("bye".equals(str)){
                        flag = false;
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

        }else{
            //将接收到的字符串前面加上echo，发送到对应的客户端
            out.println("echo:" + str);
        }
    }
}
out.close();
client.close();
}catch(Exception e){
    e.printStackTrace();
}
}
}

```

服务端处理 TCP 连接请求的代码如下：

```

package zyb.org.server;

import java.net.ServerSocket;
import java.net.Socket;

public class Server1 {
    public static void main(String[] args) throws Exception{
        //服务端在20006端口监听客户端请求的TCP连接
        ServerSocket server = new ServerSocket(20006);
        Socket client = null;
        boolean f = true;
        while(f){
            //等待客户端的连接，如果没有获取连接
            client = server.accept();
            System.out.println("与客户端连接成功！");
            //为每个客户端连接开启一个线程
            new Thread(new ServerThread(client)).start();
        }
        server.close();
    }
}

```

执行结果截图如下：

```
Console X
<terminated> Client1 [Java Application] D:\JAVA\jre\bin\java
输入信息: java编程思想
echo:java编程思想
输入信息: java网络编程
echo:java网络编程
输入信息: bye
```

```
Console X
Server1 [Java Application] D:\JAVA\jre\bin\javaw.exe (2013-1
与客户端连接成功!

http://blog.csdn.net/ns_code
```



3

UDP Socket 编程



UDP 的 Java 支持

UDP 协议提供的服务不同于 TCP 协议的端到端服务，它是面向非连接的，属不可靠协议，UDP 套接字在使用前不需要进行连接。实际上，UDP 协议只实现了两个功能：

- 在 IP 协议的基础上添加了端口；
- 对传输过程中可能产生的数据错误进行了检测，并抛弃已经损坏的数据。

Java 通过 `DatagramPacket` 类和 `DatagramSocket` 类来使用 UDP 套接字，客户端和服务端都通过 `DatagramSocket` 的 `send()` 方法和 `receive()` 方法来发送和接收数据，用 `DatagramPacket` 来包装需要发送或者接收到的数据。发送信息时，Java 创建一个包含待发送信息的 `DatagramPacket` 实例，并将其作为参数传递给 `DatagramSocket` 实例的 `send()` 方法；接收信息时，Java 程序首先创建一个 `DatagramPacket` 实例，该实例预先分配了一些空间，并将接收到的信息存放在该空间中，然后把该实例作为参数传递给 `DatagramSocket` 实例的 `receive()` 方法。在创建 `DatagramPacket` 实例时，要注意：如果该实例用来包装待接收的数据，则不指定数据来源的远程主机和端口，只需指定一个缓存数据的 byte 数组即可（在调用 `receive()` 方法接收到数据后，源地址和端口等信息会自动包含在 `DatagramPacket` 实例中），而如果该实例用来包装待发送的数据，则要指定要发送到的目的主机和端口。

UDP 的通信建立的步骤

UDP 客户端首先向被动等待联系的服务器发送一个数据报文。一个典型的 UDP 客户端要经过下面三步操作：

- 创建一个 `DatagramSocket` 实例，可以有选择对本地地址和端口号进行设置，如果设置了端口号，则客户端会在该端口号上监听从服务器端发送来的数据；
- 使用 `DatagramSocket` 实例的 `send()` 和 `receive()` 方法来发送和接收 `DatagramPacket` 实例，进行通信；
- 通信完成后，调用 `DatagramSocket` 实例的 `close()` 方法来关闭该套接字。

由于 UDP 是无连接的，因此 UDP 服务端不需要等待客户端的请求以建立连接。另外，UDP 服务器为所有通信使用同一套接字，这点与 TCP 服务器不同，TCP 服务器则为每个成功返回的 `accept()` 方法创建一个新的套接字。一个典型的 UDP 服务端要经过下面三步操作：

- 创建一个 `DatagramSocket` 实例，指定本地端口号，并可以有选择地指定本地地址，此时，服务器已经准备好从任何客户端接收数据报文；

- 使用 DatagramSocket 实例的 receive()方法接收一个 DatagramPacket 实例，当 receive()方法返回时，数据报文就包含了客户端的地址，这样就知道了回复信息应该发送到什么地方；
- 使用 DatagramSocket 实例的 send()方法向服务器端返回 DatagramPacket 实例。

UDP Socket Demo

这里有一点需要注意：UDP 程序在 receive()方法处阻塞，直到收到一个数据报文或等待超时。由于 UDP 协议是不可靠协议，如果数据报在传输过程中发生丢失，那么程序将会一直阻塞在 receive()方法处，这样客户端将永远都接收不到服务器端发送回来的数据，但是又没有任何提示。为了避免这个问题，我们在客户端使用 DatagramSocket 类的 setSoTimeout()方法来制定 receive()方法的最长阻塞时间，并指定重发数据报的次数，如果每次阻塞都超时，并且重发次数达到了设置的上限，则关闭客户端。

下面给出一个客户端服务端 UDP 通信的 Demo（没有用多线程），该客户端在本地 9000 端口监听接收到的数据，并将字符串"Hello UDPserver"发送到本地服务器的 3000 端口，服务端在本地 3000 端口监听接收到的数据，如果接收到数据，则返回字符串"Hello UDPclient"到该客户端的 9000 端口。在客户端，由于程序可能会一直阻塞在 receive()方法处，因此这里我们在客户端用 DatagramSocket 实例的 setSoTimeout()方法来指定 receive（）的最长阻塞时间，并设置重发数据的次数，如果最终依然没有接收到从服务端发送回来的数据，我们就关闭客户端。

客户端代码如下：

```
package zyb.org.UDP;

import java.io.IOException;
import java.io.InterruptedIOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class UDPClient {
    private static final int TIMEOUT = 5000; //设置接收数据的超时时间
    private static final int MAXNUM = 5;    //设置重发数据的最多次数
    public static void main(String args[])throws IOException{
        String str_send = "Hello UDPserver";
        byte[] buf = new byte[1024];
        //客户端在9000端口监听接收到的数据
        DatagramSocket ds = new DatagramSocket(9000);
        InetAddress loc = InetAddress.getLocalHost();
        //定义用来发送数据的DatagramPacket实例
        DatagramPacket dp_send= new DatagramPacket(str_send.getBytes(),str_send.length(),loc,3000);
```

```

//定义用来接收数据的DatagramPacket实例
DatagramPacket dp_receive = new DatagramPacket(buf, 1024);
//数据发向本地3000端口
ds.setSoTimeout(TIMEOUT);          //设置接收数据时阻塞的最长时间
int tries = 0;                      //重发数据的次数
boolean receivedResponse = false;   //是否接收到数据的标志位
//直到接收到数据，或者重发次数达到预定值，则退出循环
while(!receivedResponse && tries<MAXNUM){
    //发送数据
    ds.send(dp_send);
    try{
        //接收从服务端发送回来的数据
        ds.receive(dp_receive);
        //如果接收到的数据不是来自目标地址，则抛出异常
        if(!dp_receive.getAddress().equals(loc)){
            throw new IOException("Received packet from an unknown source");
        }
        //如果接收到数据。则将receivedResponse标志位改为true，从而退出循环
        receivedResponse = true;
    }catch(InterruptedException e){
        //如果接收数据时阻塞超时，重发并减少一次重发的次数
        tries += 1;
        System.out.println("Time out," + (MAXNUM - tries) + " more tries..." );
    }
}
if(receivedResponse){
    //如果收到数据，则打印出来
    System.out.println("client received data from server: ");
    String str_receive = new String(dp_receive.getData(),0,dp_receive.getLength()) +
        " from " + dp_receive.getAddress().getHostAddress() + ":" + dp_receive.getPort();
    System.out.println(str_receive);
    //由于dp_receive在接收了数据之后，其内部消息长度值会变为实际接收的消息的字节数，
    //所以这里要将dp_receive的内部消息长度重新置为1024
    dp_receive.setLength(1024);
}else{
    //如果重发MAXNUM次数数据后，仍未获得服务器发送回来的数据，则打印如下信息
    System.out.println("No response -- give up.");
}
ds.close();
}
}

```

服务端代码如下：

```
package zyb.org.UDP;
```



```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class UDPServer {
    public static void main(String[] args) throws IOException {
        String str_send = "Hello UDPClient";
        byte[] buf = new byte[1024];
        //服务端在3000端口监听接收到的数据
        DatagramSocket ds = new DatagramSocket(3000);
        //接收从客户端发送过来的数据
        DatagramPacket dp_receive = new DatagramPacket(buf, 1024);
        System.out.println("server is on, waiting for client to send data.....");
        boolean f = true;
        while(f){
            //服务器端接收来自客户端的数据
            ds.receive(dp_receive);
            System.out.println("server received data from client: ");
            String str_receive = new String(dp_receive.getData(), 0, dp_receive.getLength()) +
                " from " + dp_receive.getAddress().getHostAddress() + ":" + dp_receive.getPort();
            System.out.println(str_receive);
            //数据发动到客户端的3000端口
            DatagramPacket dp_send = new DatagramPacket(str_send.getBytes(), str_send.length(), dp_receive.getAddress());
            ds.send(dp_send);
            //由于dp_receive在接收了数据之后，其内部消息长度值会变为实际接收的消息的字节数，
            //所以这里要将dp_receive的内部消息长度重新置为1024
            dp_receive.setLength(1024);
        }
        ds.close();
    }
}

```

如果服务器端没有运行，则 receive () 会失败，此时运行结果如下图所示：

```

<terminated> UDPClient [Java Application] D:\jre\bin\javaw
Time out,4 more tries...
Time out,3 more tries...
Time out,2 more tries...
Time out,1 more tries...
Time out,0 more tries...
No response -- give up.

```

如果服务器端先运行，而客户端还没有运行，则服务端运行结果如下图所示：

```
UDPServer [Java Application] D:\jre\bin\javaw.exe (2013-11-4 下午7:40:48)
server is on, waiting for client to send data.....
```

http://blog.csdn.net/ns_code

此时，如果客户端运行，将向服务端发送数据，并接受从服务端发送回来的数据，此时运行结果如下图所示：

```
UDPServer [Java Application] D:\jre\bin\javaw.exe (2013-11-4 下午7:4
server is on, waiting for client to send data.....
server received data from client:
Hello UDPServer from 172.31.88.152:9000
```

http://blog.csdn.net/ns_code

```
<terminated> UDPClient [Java Application] D:\jre\bin\javaw.exe (2013
client received data from server:
Hello UDPclient from 172.31.88.152:3000
```

http://blog.csdn.net/ns_code

需要注意的地方

UDP 套接字和 TCP 套接字的一个微小但重要的差别：UDP 协议保留了消息的边界信息。

DatagramSocket 的每一次 receive()调用最多只能接收调用一次 send()方法所发送的数据，而且，不同的 receive()方法调用绝对不会返回同一个 send()方法所发送的数据。

当在 TCP 套接字的输出流上调用 write()方法返回后，所有调用者都知道数据已经被复制到一个传输缓存区中，实际上此时数据可能已经被发送，也有可能还没有被传送，而 UDP 协议没有提供从网络错误中恢复的机制，因此，并不对可能需要重传的数据进行缓存。这就意味着，当 send () 方法调用返回时，消息已经被发送到了底层的传输信道中。

UDP 数据报文所能负载的最多数据，亦及一次传送的最大数据为 65507 个字节。

当消息从网络中到达后，其所包含的数据被 TCP 的 `read()` 方法或 UDP 的 `receive()` 方法返回前，数据存储在一个先进先出的接收数据队列中。对于已经建立连接的 TCP 套接字来说，所有已接受但还未传送的字节都看作是一个连续的字节序列。然而，对于 UDP 套接字来说，接收到的数据可能来自不同的发送者，一个 UDP 套接字所接受的数据存放在一个消息队列中，每个消息都关联了其源地地址信息，每次 `receive()` 调用只返回一条消息。如果 `receive()` 方法在一个缓存区大小为 `n` 的 `DatagramPacket` 实例中调用，而接受队里中的第一条消息的长度大于 `n`，则 `receive()` 方法只返回这条消息的前 `n` 个字节，超出部分会被自动放弃，而且对接收程序没有任何消息丢失的提示！

出于这个原因，接受者应该提供一个有足够大的缓存空间的 `DatagramPacket` 实例，以完整地存放调用 `receive()` 方法时应用程序协议所允许的最大长度的消息。一个 `DatagramPacket` 实例中所允许传输的最大数据量为 65507 个字节，也即是 UDP 数据报文所能负载的最多数据。因此，可以用一个 65600 字节左右的缓存数组来接受数据。

`DatagramPacket` 的内部消息长度值在接收数据后会发生改变，变为实际接收到的数据的长度值。

每一个 `DatagramPacket` 实例都包含一个内部消息长度值，其初始值为 `byte` 缓存数组的长度值，而该实例一旦接收到消息，这个长度值便会变为接收到的消息的实际长度值，这一点可以用 `DatagramPacket` 类的 `getLength()` 方法来测试。如果一个应用程序使用同一个 `DatagramPacket` 实例多次调用 `receive()` 方法，每次调用前就必须显式地将其内部消息长度重置为缓存区的实际长度，以免接受的数据发生丢失。

以上面的程序为例，若在服务端的 `receiver()` 后加入如下代码：

```
System.out.println(dp_receive.getLength());
```

则得到的输出结果为：15，即接收到的字符串数据“Hello UDPserver”的长度。

`DatagramPacket` 的 `getData()` 方法总是返回缓冲区的原始大小，忽略了实际数据的内部偏移量和长度信息。

由于 `DatagramPacket` 的 `getData()` 方法总是返回缓冲数组的原始大小，即刚开始创建缓冲数组时指定的大小，在上面程序中，该长度为 1024，因此如果我们要获取接收到的数据，就必须截取 `getData()` 方法返回的数组中只含接收到的数据的那一部分。在 Java1.6 之后，我们可以使用 `Arrays.copyOfRange()` 方法来实现，只需一步便可实现以上功能：

```
byte[] destbuf = Arrays.copyOfRange(dp_receive.getData(), dp_receive.getOffset(),
dp_receive.getOffset() + dp_receive.getLength());
```

当然，如果要将接收到的字节数组转换为字符串的话，也可以采用本程序中直接 `new` 一个 `String` 对象的方法：

```
new String(dp_receive.getData(), dp_receive.getOffset(),
dp_receive.getOffset() + dp_receive.getLength());
```



4



应用程序协议中消息的成帧与解析



程序间达成的某种包含了信息交换的形式和意义的共识称为协议，用来实现特定应用程序的协议叫做应用程序协议。大部分应用程序协议是根据由字段序列组成的离散信息定义的，其中每个字段中都包含了一段以位序列编码（即二进制字节编码，也可以使用基于文本编码的方式，但常用协议如：TCP、UDP、HTTP 等在传输数据时，都是以位序列编码的）的特定信息。应用程序协议中明确定义了信息的发送者应该如何排列和解释这些位序列，同时还要定义接收者应该如何解析，这样才能使信息的接收者能够抽取出每个字段的意义。TCP/IP 协议唯一的约束：信息必须在块中发送和接收，而块的长度必须是 8 位的倍数，因此，我们可以认为 TCP/IP 协议中传输的信息是字节序列。

由于协议通常处理的是由一组字段组成的离散的信息，因此应用程序协议必须指定消息的接收者如何确定何时消息已被完整接收。成帧技术就是解决接收端如何定位消息首尾位置问题的，由于协议通常处理的是由一组字段组成的离散的信息，因此应用程序协议必须指定消息的接收者如何确定何时消息已被完整。主要有两种技术使接收者能够准确地找到消息的结束位置：

- 基于定界符：消息的结束由一个唯一的标记指出，即发送者在传输完数据后显式添加的一个特定字节序列，这个特殊标记不能在传输的数据中出现（这也不是绝对的，应用填充技术能够对消息中出现的定界符进行修改，从而使接收者不将其识别为定界符）。该方法通常用在以文本方式编码的消息中。
- 显式长度：在变长字段或消息前附加一个固定大小的字段，用来指示该字段或消息中包含了多少字节。该方法主要用在以二进制字节方式编码的消息中。

由于 UDP 套接字保留了消息的边界信息，因此不需要进行成帧处理（实际上，主要是 DatagramPacket 负载的数据有一个确定的长度，接收者能够准确地知道消息的结束位置），而 TCP 协议中没有消息边界的概念，因此，在使用 TCP 套接字时，成帧就是一个非常重要的考虑因素（在 TCP 连接中，接收者读取完最后一条消息的最后一个字节后，将受到一个流结束标记，即 read（）返回-1，该标记指示出已经读取到了消息的末尾，非严格意义上讲，这也算是基于定界符方法的一种特殊情况）。

下面给出一个自定义实现上面两种成帧技术的 Demo（书上的例子），先定义一个 Framer 接口，它由两个方法：frameMsg（）方法用来添加成帧信息并将指定消息输出到指定流，nextMsg（）方法则扫描指定的流，从中取出下一条消息。

```
import java.io.IOException;
import java.io.OutputStream;

public interface Framer {
    void frameMsg(byte[] message, OutputStream out) throws IOException;
    byte[] nextMsg() throws IOException;
}
```

下面的代码实现了基于定界符的成帧方法，定界符为换行符“\n”，frameMsg（）方法并没有实现填充，当成帧的字节序列中包含有定界符时，它只是简单地抛出异常；nextMsg（）方法扫描流，直到读取到了定界符，并

返回定界符前面所有的字符，如果流为空则返回 null，如果直到流结束也没找到定界符，程序将抛出一个异常来指示成帧错误。

```
import java.io.ByteArrayOutputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class DelimFramer implements Framer {

    private InputStream in;    // 数据来源
    private static final byte DELIMITER = '\n'; // 定界符

    public DelimFramer(InputStream in) {
        this.in = in;
    }

    public void frameMsg(byte[] message, OutputStream out) throws IOException {
        for (byte b : message) {
            if (b == DELIMITER) {
                //如果在消息中检查到界定符，则抛出异常
                throw new IOException("Message contains delimiter");
            }
        }
        out.write(message);
        out.write(DELIMITER);
        out.flush();
    }

    public byte[] nextMsg() throws IOException {
        ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
        int nextByte;

        while ((nextByte = in.read()) != DELIMITER) {
            //如果流已经结束还没有读取到定界符
            if (nextByte == -1) {
                //如果读取到的流为空，则返回null
                if (messageBuffer.size() == 0) {
                    return null;
                } else {
                    //如果读取到的流不为空，则抛出异常
                    throw new EOFException("Non-empty message without delimiter");
                }
            }
        }
    }
}
```

```

    messageBuffer.write(nextByte);
}

return messageBuffer.toByteArray();
}
}

```

下面的代码实现了基于长度的成帧方法，适用于长度小于 65535 个字节的消息。发送者首先给出指定消息的长度，并将长度信息以 big-endian 顺序（从左边开始，由高位到低位发送）存入 2 个字节的整数中，再将这两个字节存放在完整的消息内容前，连同消息一起写入输出流；在接收端，使用 `DataInputStream` 读取整型的长度信息，`readFully()` 方法将阻塞等待，直到给定的数组完全填满。使用这种成帧方法，发送者不需要检查要成帧的消息内容，而只需要检查消息的长度是否超出了限制。

```

import java.io.DataInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class LengthFramer implements Framer {
    public static final int MAXMESSAGELENGTH = 65535;
    public static final int BYTEMASK = 0xff;
    public static final int SHORTMASK = 0xffff;
    public static final int BYTESHIFT = 8;

    private DataInputStream in;

    public LengthFramer(InputStream in) throws IOException {
        this.in = new DataInputStream(in); //数据来源
    }

    //对字节流message添加成帧信息，并输出到指定流
    public void frameMsg(byte[] message, OutputStream out) throws IOException {
        //消息的长度不能超过65535
        if (message.length > MAXMESSAGELENGTH) {
            throw new IOException("message too long");
        }
        out.write((message.length >> BYTESHIFT) & BYTEMASK);
        out.write(message.length & BYTEMASK);
        out.write(message);
        out.flush();
    }

    public byte[] nextMsg() throws IOException {
        int length;
    }
}

```

```
try {  
    //该方法读取2个字节，将它们作为big-endian整数进行解释，并以int型整数返回它们的值  
    length = in.readUnsignedShort();  
} catch (EOFException e) { // no (or 1 byte) message  
    return null;  
}  
// 0 <= length <= 65535  
byte[] msg = new byte[length];  
//该方法处阻塞等待，直到接收到足够的字节来填满指定的数组  
in.readFully(msg);  
return msg;  
}  
}
```




T

5



构建和解析自定义协议消息



在传输消息时，用 Java 内置的方法和工具确实很用，如：对象序列化，RMI 远程调用等。但有时候，针对要传输的特定类型的数据，实现自己的方法可能更简单、容易或有效。下面给出一个实现了自定义构建和解析协议消息的 Demo。

该例子是一个简单的投票协议。这里，一个客户端向服务器发送一个请求消息，消息中包含了一个候选人的 ID，范围在 0~1000。程序支持两种请求：一种是查询请求，即向服务器询问候选人当前获得的投票总数，服务器发回一个响应消息，包含了原来的候选人 ID 和该候选人当前获得的选票总数；另一种是投票请求，即向指定候选人投一票，服务器对这种请求也发回响应消息，包含了候选人 ID 和获得的选票数（包含了刚刚投的一票）。

在实现一个协议时，一般会定义一个专门的类来存放消息中所包含的信息。在我们的例子中，客户端和服务端发送的消息都很简单，唯一的区别是服务端发送的消息还包含了选票总数和一个表示相应消息的标志。因此，可以用一个类来表示客户端和服务端的两种消息。下面的 VoteMsg.java 类展示了每条消息中的基本信息：

- 布尔值 isInquiry, true 表示该消息是查询请求, false 表示该消息是投票请求;
- 布尔值 isResponse, true 表示该消息是服务器发送的相应消息, false 表示该消息为客户端发送的请求消息;
- 整型变量 candidateID, 指示了候选人的 ID;
- 长整型变量 voteCount, 指示出所查询的候选人获得的总选票数。

另外，注意一下几点：

- candidateID 的范围在 0~1000;
- voteCount 在请求消息中必须为 0;
- voteCount 不能为负数。

VoteMsg 代码如下：

```
public class VoteMsg {
    private boolean isInquiry; // true if inquiry; false if vote
    private boolean isResponse; // true if response from server
    private int candidateID; // in [0,1000]
    private long voteCount; // nonzero only in response

    public static final int MAX_CANDIDATE_ID = 1000;

    public VoteMsg(boolean isResponse, boolean isInquiry, int candidateID, long voteCount)
        throws IllegalArgumentException {
        // check invariants
        if (voteCount != 0 && !isResponse) {
            throw new IllegalArgumentException("Request vote count must be zero");
        }
    }
}
```

```

    if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
        throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
    }
    if (voteCount < 0) {
        throw new IllegalArgumentException("Total must be >= zero");
    }
    this.candidateID = candidateID;
    this.isResponse = isResponse;
    this.isInquiry = isInquiry;
    this.voteCount = voteCount;
}

public void setInquiry(boolean isInquiry) {
    this.isInquiry = isInquiry;
}

public void setResponse(boolean isResponse) {
    this.isResponse = isResponse;
}

public boolean isInquiry() {
    return isInquiry;
}

public boolean isResponse() {
    return isResponse;
}

public void setCandidateID(int candidateID) throws IllegalArgumentException {
    if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
        throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
    }
    this.candidateID = candidateID;
}

public int getCandidateID() {
    return candidateID;
}

public void setVoteCount(long count) {
    if ((count != 0 && !isResponse) || count < 0) {
        throw new IllegalArgumentException("Bad vote count");
    }
    voteCount = count;
}

```

```

public long getVoteCount() {
    return voteCount;
}

public String toString() {
    String res = (isInquiry ? "inquiry" : "vote") + " for candidate " + candidateID;
    if (isResponse) {
        res = "response to " + res + " who now has " + voteCount + " vote(s)";
    }
    return res;
}
}

```

接下来，我们要根据一定的协议来对其进行编解码，我们定义一个 `VoteMsgCoder` 接口，它提供了对投票消息进行序列化和反序列化的方法。`toWire()` 方法用于根据一个特定的协议，将投票消息转换成一个字节序列，`fromWire()` 方法则根据相同的协议，对给定的字节序列进行解析，并根据信息的内容返回一个该消息类的实例。

```

import java.io.IOException;

public interface VoteMsgCoder {
    byte[] toWire(VoteMsg msg) throws IOException;
    VoteMsg fromWire(byte[] input) throws IOException;
}

```

下面给出两个实现了 `VoteMsgCoder` 接口的类，一个实现的是基于文本的编码方式，一个实现的是基于二进制的编码方式。

首先是用文本方式对消息进行编码的程序。该协议指定使用 ASCII 字符集对文本进行编码。消息的开头是一个所谓的“魔术字符串”，即一个字符序列，用于快速将投票协议的消息和网络中随机到来的垃圾消息区分开，投票/查询布尔值被编码为字符形似，‘v’ 代表投票消息，‘i’ 代表查询消息。是否为服务器发送的响应消息，由字符‘R’ 指示，状态标记后面是候选人 ID，其后跟的是选票总数，它们都编码成十进制字符串。

```

import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class VoteMsgTextCoder implements VoteMsgCoder {
    /*
     * Wire Format "VOTEPROTO" <"v" | "i"> [<RESPFLAG>] <CANDIDATE> [<VOTECNT>]
     * Charset is fixed by the wire format.
     */
}

```

```

// Manifest constants for encoding
public static final String MAGIC = "Voting";
public static final String VOTESTR = "v";
public static final String INQSTR = "i";
public static final String RESPONSESTR = "R";

public static final String CHARSETNAME = "US-ASCII";
public static final String DELIMSTR = " ";
public static final int MAX_WIRE_LENGTH = 2000;

public byte[] toWire(VoteMsg msg) throws IOException {
    String msgString = MAGIC + DELIMSTR + (msg.isInquiry() ? INQSTR : VOTESTR)
        + DELIMSTR + (msg.isResponse() ? RESPONSESTR + DELIMSTR : "")
        + Integer.toString(msg.getCandidateID()) + DELIMSTR
        + Long.toString(msg.getVoteCount());
    byte data[] = msgString.getBytes(CHARSETNAME);
    return data;
}

public VoteMsg fromWire(byte[] message) throws IOException {
    ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
    Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
    boolean isInquiry;
    boolean isResponse;
    int candidateID;
    long voteCount;
    String token;

    try {
        token = s.next();
        if (!token.equals(MAGIC)) {
            throw new IOException("Bad magic string: " + token);
        }
        token = s.next();
        if (token.equals(VOTESTR)) {
            isInquiry = false;
        } else if (!token.equals(INQSTR)) {
            throw new IOException("Bad vote/inq indicator: " + token);
        } else {
            isInquiry = true;
        }

        token = s.next();
        if (token.equals(RESPONSESTR)) {
            isResponse = true;

```

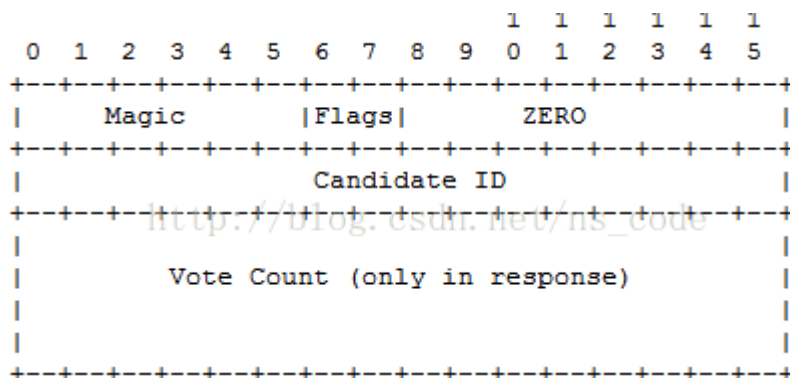
```

        token = s.next();
    } else {
        isResponse = false;
    }
    // Current token is candidateID
    // Note: isResponse now valid
    candidateID = Integer.parseInt(token);
    if (isResponse) {
        token = s.next();
        voteCount = Long.parseLong(token);
    } else {
        voteCount = 0;
    }
} catch (IOException ioe) {
    throw new IOException("Parse error...");
}
return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
}
}

```

toWire () 方法简单地创建一个字符串，该字符串中包含了消息的所有字段，并由空白符隔开。fromWire () 方法首先检查”魔术字符串“，如果在消息最前面没有魔术字符串，则抛出一个异常。在理说明了在实现协议时非常重要的一点：永远不要对从网络中来的任何输入进行任何假设。你的程序必须时刻为任何可能的输入做好准备，并能很好的对其进行处理。

下面将展示基于二进制格式对消息进行编码的程序。与基于文本的格式相反，二进制格式使用固定大小的消息，每条消息由一个特殊字节开始，该字节的最高六位为一个”魔术值“010101，该字节的最低两位对两个布尔值进行了编码，消息的第二个字节总是 0，第三、四个字节包含了 candidateID 值，只有响应消息的最后 8 个字节才包含了选票总数信息。字节序列格式如下图所示：



代码如下：

```

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;

```

```

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;

public class VoteMsgBinCoder implements VoteMsgCoder {

    // manifest constants for encoding
    public static final int MIN_WIRE_LENGTH = 4;
    public static final int MAX_WIRE_LENGTH = 16;
    public static final int MAGIC = 0x5400;
    public static final int MAGIC_MASK = 0xfc00;
    public static final int MAGIC_SHIFT = 8;
    public static final int RESPONSE_FLAG = 0x0200;
    public static final int INQUIRE_FLAG = 0x0100;

    public byte[] toWire(VoteMsg msg) throws IOException {
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        DataOutputStream out = new DataOutputStream(byteStream); // converts ints

        short magicAndFlags = MAGIC;
        if (msg.isInquiry()) {
            magicAndFlags |= INQUIRE_FLAG;
        }
        if (msg.isResponse()) {
            magicAndFlags |= RESPONSE_FLAG;
        }
        out.writeShort(magicAndFlags);
        // We know the candidate ID will fit in a short: it's > 0 && < 1000
        out.writeShort((short) msg.getCandidateID());
        if (msg.isResponse()) {
            out.writeLong(msg.getVoteCount());
        }
        out.flush();
        byte[] data = byteStream.toByteArray();
        return data;
    }

    public VoteMsg fromWire(byte[] input) throws IOException {
        // sanity checks
        if (input.length < MIN_WIRE_LENGTH) {
            throw new IOException("Runt message");
        }
        ByteArrayInputStream bs = new ByteArrayInputStream(input);
        DataInputStream in = new DataInputStream(bs);
        int magic = in.readShort();
    }

```

```
if ((magic & MAGIC_MASK) != MAGIC) {
    throw new IOException("Bad Magic #: " +
        ((magic & MAGIC_MASK) >> MAGIC_SHIFT));
}
boolean resp = ((magic & RESPONSE_FLAG) != 0);
boolean inq = ((magic & INQUIRE_FLAG) != 0);
int candidateID = in.readShort();
if (candidateID < 0 || candidateID > 1000) {
    throw new IOException("Bad candidate ID: " + candidateID);
}
long count = 0;
if (resp) {
    count = in.readLong();
    if (count < 0) {
        throw new IOException("Bad vote count: " + count);
    }
}
// Ignore any extra bytes
return new VoteMsg(resp, inq, candidateID, count);
}
```




T



6

基于线程池的 TCP 服务器



线程池

在 [Java TCP Socket 编程 \(\)](#) 这篇文章中，服务器端采用的实现方式是：一个客户端对应一个线程。但是，每个新线程都会消耗系统资源：创建一个线程会占用 CPU 周期，而且每个线程都会建立自己的数据结构（如，栈），也要消耗系统内存，另外，当一个线程阻塞时，JVM 将保存其状态，选择另外一个线程运行，并在上下文转换（context switch）时恢复阻塞线程的状态。随着线程数的增加，线程将消耗越来越多的系统资源，这将最终导致系统花费更多的时间来处理上下文转换和线程管理，更少的时间来对连接进行服务。在这种情况下，加入一个额外的线程实际上可能增加客户端总服务的时间。

我们可以通过限制线程总数并重复使用线程来避免这个问题。我们让服务器在启动时创建一个由固定线程数量组成的线程池，当一个新的客户端连接请求传入服务器，它将交给线程池中的一个线程处理，该线程处理完这个客户端之后，又返回线程池，继续等待下一次请求。如果连接请求到达服务器时，线程池中所有的线程都已经被占用，它们则在一个队列中等待，直到有空闲的线程可用。

实现步骤

- 与一客户一线程服务器一样，线程池服务器首先创建一个 ServerSocket 实例。
- 然后创建 N 个线程，每个线程反复循环，从（共享的）ServerSocket 实例接收客户端连接。当多个线程同时调用一个 ServerSocket 实例的 accept() 方法时，它们都将阻塞等待，直到一个新的连接成功建立，然后系统选择一个线程，为建立起的连接提供服务，其他线程则继续阻塞等待。
- 线程在完成对一个客户端的服务后，继续等待其他的连接请求，而不终止。如果在一个客户端连接被创建时，没有线程在 accept() 方法上阻塞（即所有的线程都在为其他连接服务），系统则将新的连接排列在一个队列中，直到下一次调用 accept() 方法。

示例代码

我们依然实现 [Java TCP Socket 编程 \(\)](#) 这篇文章中的功能，客户端代码相同，服务器端代码在其基础上改为基于线程池的实现，为了方便在匿名线程中调用处理通信细节的方法，我们对多线程类 ServerThread 做了一些微小的改动，如下：

```
package zyb.org.server;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.net.Socket;
```

```

/**
 * 该类为多线程类，用于服务端
 */
public class ServerThread implements Runnable {

    private Socket client = null;
    public ServerThread(Socket client){
        this.client = client;
    }

    //处理通信细节的静态方法，这里主要是方便线程池服务器的调用
    public static void execute(Socket client){
        try{
            //获取Socket的输出流，用来向客户端发送数据
            PrintStream out = new PrintStream(client.getOutputStream());
            //获取Socket的输入流，用来接收从客户端发送过来的数据
            BufferedReader buf = new BufferedReader(new InputStreamReader(client.getInputStream()));
            boolean flag =true;
            while(flag){
                //接收从客户端发送过来的数据
                String str = buf.readLine();
                if(str == null || "".equals(str)){
                    flag = false;
                }else{
                    if("bye".equals(str)){
                        flag = false;
                    }else{
                        //将接收到的字符串前面加上echo，发送到对应的客户端
                        out.println("echo:" + str);
                    }
                }
            }
            out.close();
            buf.close();
            client.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        execute(client);
    }
}

```

这样我们就可以很方便地在匿名线程中调用处理通信细节的方法，改进后的服务器端代码如下：

```
package zyb.org.server;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 该类实现基于线程池的服务器
 */
public class serverPool {

    private static final int THREADPOOLSIZE = 2;

    public static void main(String[] args) throws IOException{
        //服务端在20006端口监听客户端请求的TCP连接
        final ServerSocket server = new ServerSocket(20006);

        //在线程池中一共只有THREADPOOLSIZE个线程，
        //最多有THREADPOOLSIZE个线程在accept()方法上阻塞等待连接请求
        for(int i=0;i<THREADPOOLSIZE;i++){
            //匿名内部类，当前线程为匿名线程，还没有为任何客户端连接提供服务
            Thread thread = new Thread(){
                public void run(){
                    //线程为某连接提供完服务后，循环等待其他的连接请求
                    while(true){
                        try {
                            //等待客户端的连接
                            Socket client = server.accept();
                            System.out.println("与客户端连接成功！");
                            //一旦连接成功，则在该线程中与客户端通信
                            ServerThread.execute(client);
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            };
            //先将所有的线程开启
            thread.start();
        }
    }
}
```

结果分析

为了便于测试，程序中，我们将线程池中的线程总数设置为 2，这样，服务器端最多只能同时连接 2 个客户端，如果已有 2 个客户端与服务器建立了连接，当我们打开第 3 个客户端的时候，便无法再建立连接，服务器端不会打印出第 3 个“与客户端连接成功！”的字样。

```
serverPool [Java Application] D:\jre\bin\javaw.exe (2013-11-7 下午4:16:18)
与客户端连接成功！
与客户端连接成功！
```

http://blog.csdn.net/ns_code

这第 3 个客户端如果过了一段时间还没接收到服务端发回的数据，便会抛出一个 `SocketTimeoutException` 异常，从而打印出如下信息：

```
Client1 (1) [Java Application] D:\jre\bin\javaw.exe (2013-11-
输入信息: 分布式系统
Time out, No response
输入信息:
```

如果在抛出 `SocketTimeoutException` 异常之前，有一个客户端的连接关掉了，则第 3 个客户端便会与服务器端建立起连接，从而收到返回的数据。

```
Client1 (1) [Java Application] D:\jre\bin\javaw.exe (2013-11-7 下午4:34:15)
输入信息: 分布式系统
echo:分布式系统
输入信息: |
```

改进

在创建线程池时，线程池的大小是个很重要的考虑因素，如果创建的线程太多（空闲线程太多），则会消耗掉很多系统资源，如果创建的线程太少，客户端还是有可能等很长时间才能获得服务。因此，线程池的大小需要根据负载情况进行调整，以使客户端连接的时间最短，理想的情况是有一个调度的工具，可以在系统负载增加时扩展线程池的大小（低于大上限值），负载减轻时缩减线程池的大小。一种解决的方案便是使用 Java 中的 Executor 接口。

Executor 接口代表了一个根据某种策略来执行 Runnable 实例的对象，其中可能包括了排队和调度等细节，或如何选择要执行的任务。Executor 接口只定义了一个方法：

```
interface Executor{

    void execute(Runnable task);

}
```

Java 提供了大量的内置 Executor 接口实现，它们都可以简单方便地使用，ExecutorService 接口继承于 Executor 接口，它提供了一个更高级的工具来关闭服务器，包括正常的关闭和突然的关闭。我们可以通过调用 Executors 类的各种静态工厂方法来获取 ExecutorService 实例，而后通过调用 execute（）方法来为需要处理的任务分配线程，它首先会尝试使用已有的线程，但如果有必要，它会创建一个新的线程来处理任务，另外，如果一个线程空闲了 60 秒以上，则将其移出线程池，而且任务是在 Executor 的内部排队，而不像之前的服务器那样是在网络系统中排队，因此，这个策略几乎总是比前面两种方式实现的 TCP 服务器效率要高。

改进的代码如下：

```
package zyb.org.server;

import java.io.IOException;
```

```
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

/**
 * 该类通过Executor接口实现服务器
 */
public class ServerExecutor {

    public static void main(String[] args) throws IOException{
        //服务端在20006端口监听客户端请求的TCP连接
        ServerSocket server = new ServerSocket(20006);
        Socket client = null;
        //通过调用Executors类的静态方法，创建一个ExecutorService实例
        //ExecutorService接口是Executor接口的子接口
        Executor service = Executors.newCachedThreadPool();
        boolean f = true;
        while(f){
            //等待客户端的连接
            client = server.accept();
            System.out.println("与客户端连接成功！");
            //调用execute()方法时，如果必要，会创建一个新的线程来处理任务，但它首先会尝试使用已有的线程，
            //如果一个线程空闲60秒以上，则将其移除线程池；
            //另外，任务是在Executor的内部排队，而不是在网络中排队
            service.execute(new ServerThread(client));
        }
        server.close();
    }
}
```

7

Socket 通信中由 read 返回值造成的的死锁问题

示例

在第一章中，作者给出了一个 TCP Socket 通信的例子——反馈服务器，即服务器端直接把从客户端接收到的数据原原本本地反馈回去。

示例客户端代码如下：

```
import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class TCPEchoClient {

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

        if ((args.length < 2) || (args.length > 3)) // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");

        String server = args[0];    // Server name or IP address
        // Convert argument String to bytes using the default character encoding
        byte[] data = args[1].getBytes();

        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;

        // Create socket that is connected to server on specified port
        Socket socket = new Socket(server, servPort);
        System.out.println("Connected to server...sending echo string");

        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();

        out.write(data); // Send the encoded string to the server

        // Receive the same string back from the server
        int totalBytesRcvd = 0; // Total bytes received so far
        int bytesRcvd;         // Bytes received in last read
        while (totalBytesRcvd < data.length) {
            if ((bytesRcvd = in.read(data, totalBytesRcvd, data.length - totalBytesRcvd)) == -1)
                throw new SocketException("Connection closed prematurely");
            totalBytesRcvd += bytesRcvd;
        } // data array is full
```

```

        System.out.println("Received: " + new String(data));

        socket.close(); // Close the socket and its streams
    }
}

```

示例服务器端代码如下：

```

import java.net.*; // for Socket, ServerSocket, and InetAddress
import java.io.*; // for IOException and Input/OutputStream

public class TCPEchoServer {

    private static final int BUFSIZE = 32; // Size of receive buffer

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

        if (args.length != 1) // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Port>");

        int servPort = Integer.parseInt(args[0]);

        // Create a server socket to accept client connection requests
        ServerSocket servSock = new ServerSocket(servPort);

        int recvMsgSize; // Size of received message
        byte[] receiveBuf = new byte[BUFSIZE]; // Receive buffer

        while (true) { // Run forever, accepting and servicing connections
            Socket clntSock = servSock.accept(); // Get client connection

            SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
            System.out.println("Handling client at " + clientAddress);

            InputStream in = clntSock.getInputStream();
            OutputStream out = clntSock.getOutputStream();

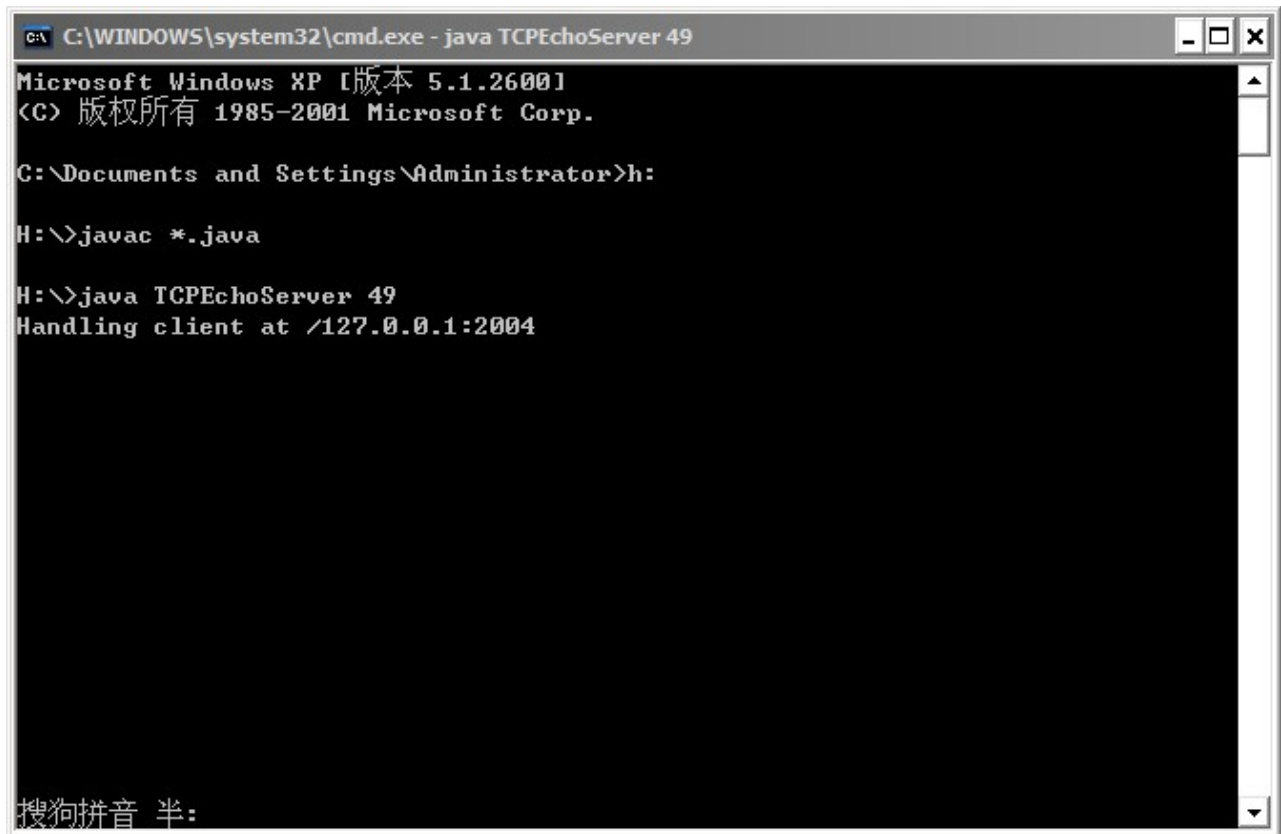
            // Receive until client closes connection, indicated by -1 return
            while ((recvMsgSize = in.read(receiveBuf)) != -1) {
                out.write(receiveBuf, 0, recvMsgSize);
            }

            clntSock.close(); // Close the socket. We are done with this client!
        }
    }
}

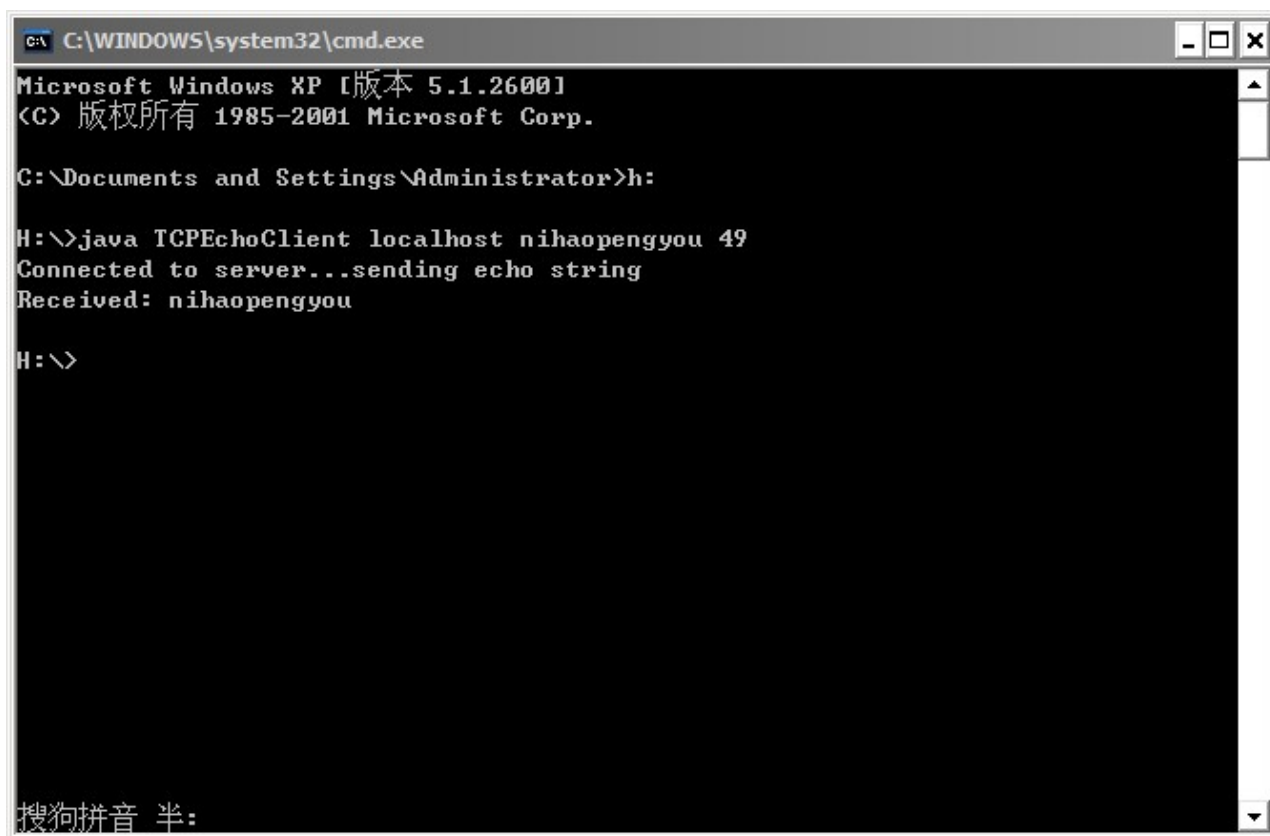
```

```
/* NOT REACHED */  
}  
}
```

运行结果如下：



```
C:\WINDOWS\system32\cmd.exe - java TCPEchoServer 49  
Microsoft Windows XP [版本 5.1.2600]  
<C> 版权所有 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\Administrator>h:  
  
H:\>javac *.java  
  
H:\>java TCPEchoServer 49  
Handling client at /127.0.0.1:2004  
  
搜狗拼音 半:
```



问题的引出

明确问题

- 客户端与服务器端在接收和发送数据时，`read()`和`write()`方法不一定要对应，比如，其中一方可以一次发送多个字节的数据，而另一方可以一个字节一个字节地接收，也可以一个字节一个字节地发送，而多个字节多个字节地接收。因为TCP协议会将数据分成多个块进行发送，而后在另一端会从多个块进行接收，再组合在一起，它并不仅能确定`read()`和`write()`方法中所发送信息的界限。
- `read()`方法会在没有数据可读时发生阻塞，直到有新的数据可读。

注意客户端中下面部分代码。

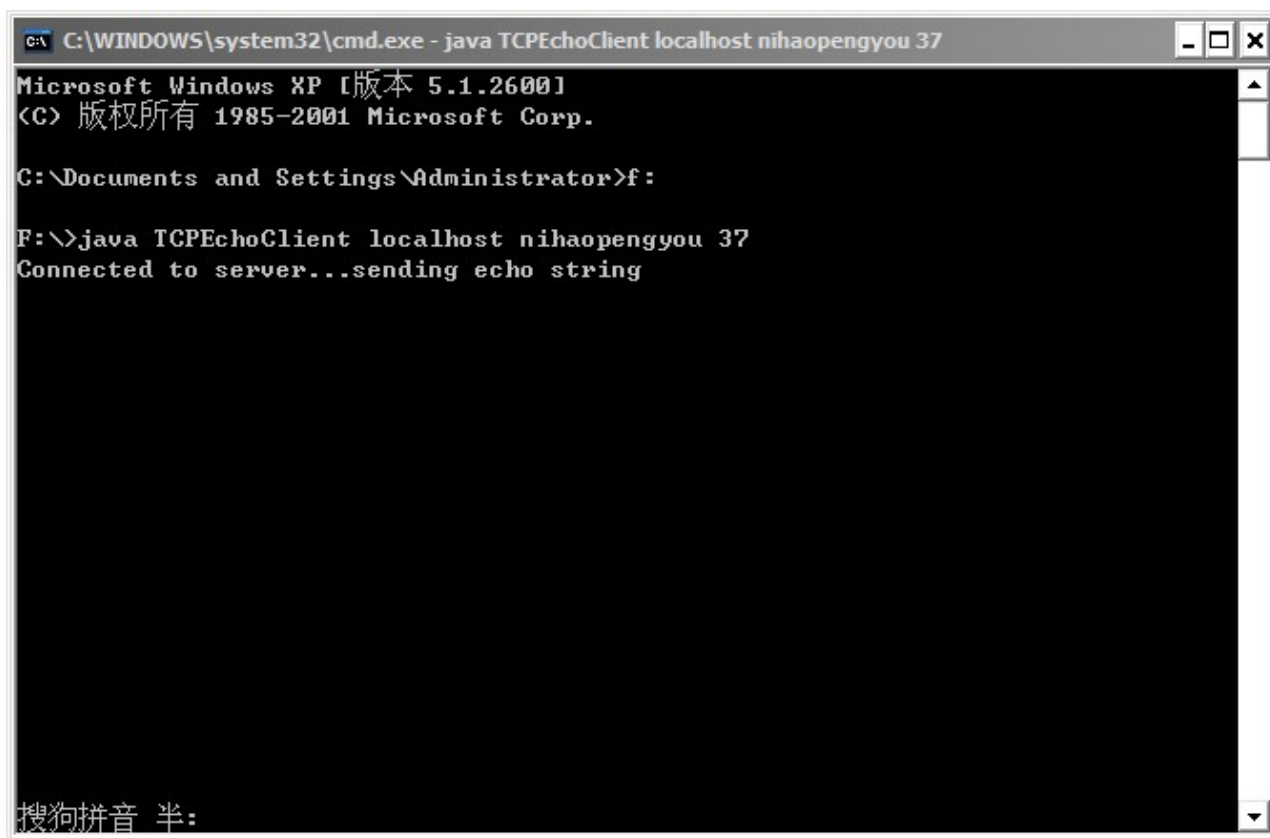
```
while (totalBytesRcvd < data.length) {
    if ((bytesRcvd = in.read(data, totalBytesRcvd, data.length - totalBytesRcvd)) == -1)
        throw new SocketException("Connection closed prematurely");
    totalBytesRcvd += bytesRcvd;
} // data array is full
```

客户端从 Socket 套接字中读取数据，直到收到的数据的字节长度和原来发送的数据的字节长度相同为止，这里的前提是已经知道了要从服务器端接收的数据的大小，如果现在我们不知道要反馈回来的数据的大小，那么我们

只能用 read 方法不断读取，直到 read() 返回 -1，说明接收到了所有的数据。我这里采用一个字节一个字节读取的方式，代码改为如下：

```
while((bytesRcvd = in.read()) != -1){  
    data[totalBytesRcvd] = (byte)bytesRcvd;  
    totalBytesRcvd++;  
}
```

这时问题就来了，输出结果如下：



```
C:\WINDOWS\system32\cmd.exe - java TCPEchoClient localhost nihaopengyou 37  
Microsoft Windows XP [版本 5.1.2600]  
<C> 版权所有 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\Administrator>f:  
  
F:\>java TCPEchoClient localhost nihaopengyou 37  
Connected to server...sending echo string  
  
搜狗拼音 半:
```

```

C:\WINDOWS\system32\cmd.exe - java TCPEchoServer 37
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>f:

F:\>javac *.java

F:\>java TCPEchoServer 37
Handling client at /127.0.0.1:2281

搜狗拼音 半:

```

问题的分析

客户端没有数据打印出来，初步推断应该是 `read()` 方法始终没有返回 `-1`，导致程序一直无法往下运行，我在客户端执行窗口中按下 `CTRL+C`，强制结束运行，在服务器端抛出如下异常：

```

Exception in thread "main" java.net.SocketException: Connection reset
    at java.net.SocketInputStream.read(Unknown Source)
    at java.net.SocketInputStream.read(Unknown Source)
    at TCPEchoServer.main(TCPEchoServer.java:32)

```

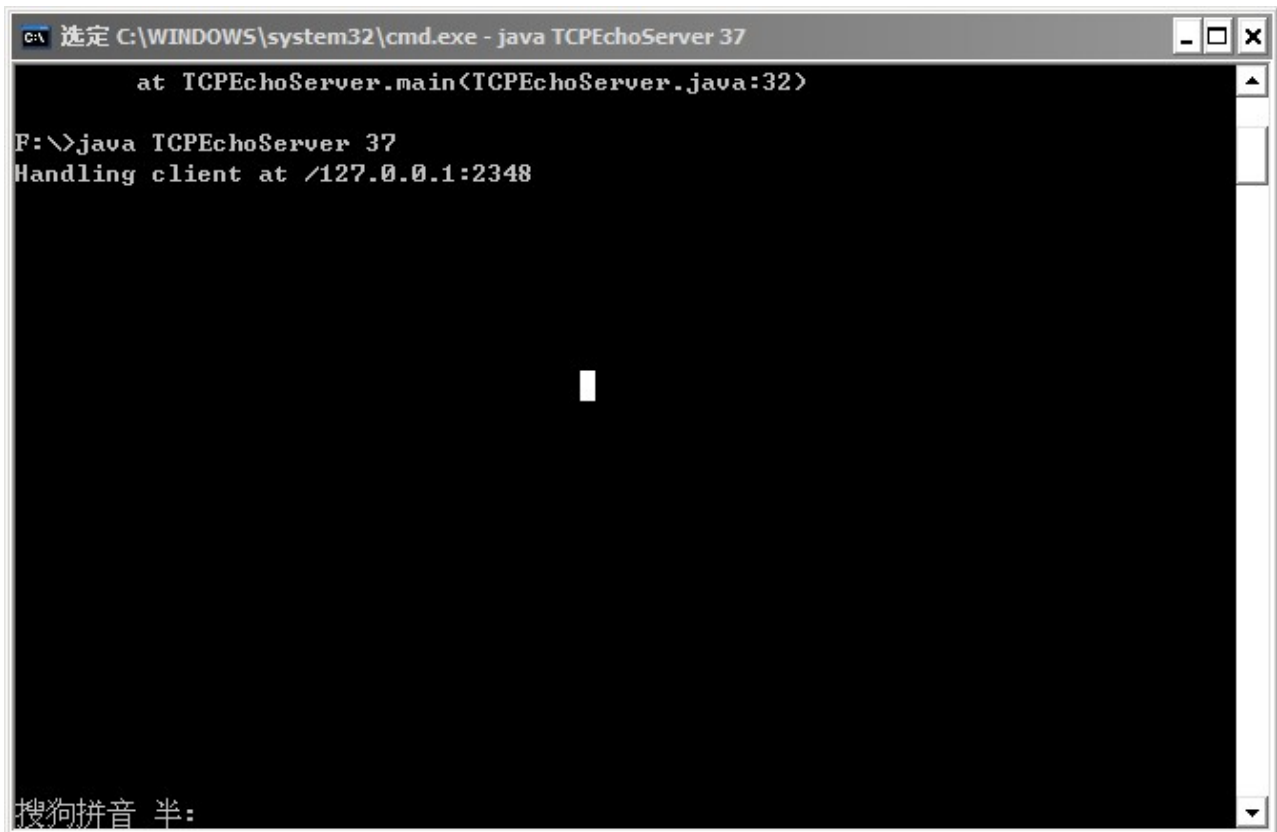
异常显示，问题出现在服务端的 32 行，没有资源可读，现在很有可能便是由于 `read()` 方法始终没有返回 `-1` 所致，为了验证，我在客户端读取字节的代码中加入了一行打印读取的单个字符的代码，如下：

```

while((bytesRcvd = in.read())!= -1){
    data[totalBytesRcvd] = (byte)bytesRcvd;
    System.out.println((char)data[totalBytesRcvd]);
    totalBytesRcvd++;
}

```

此时运行结果如下：

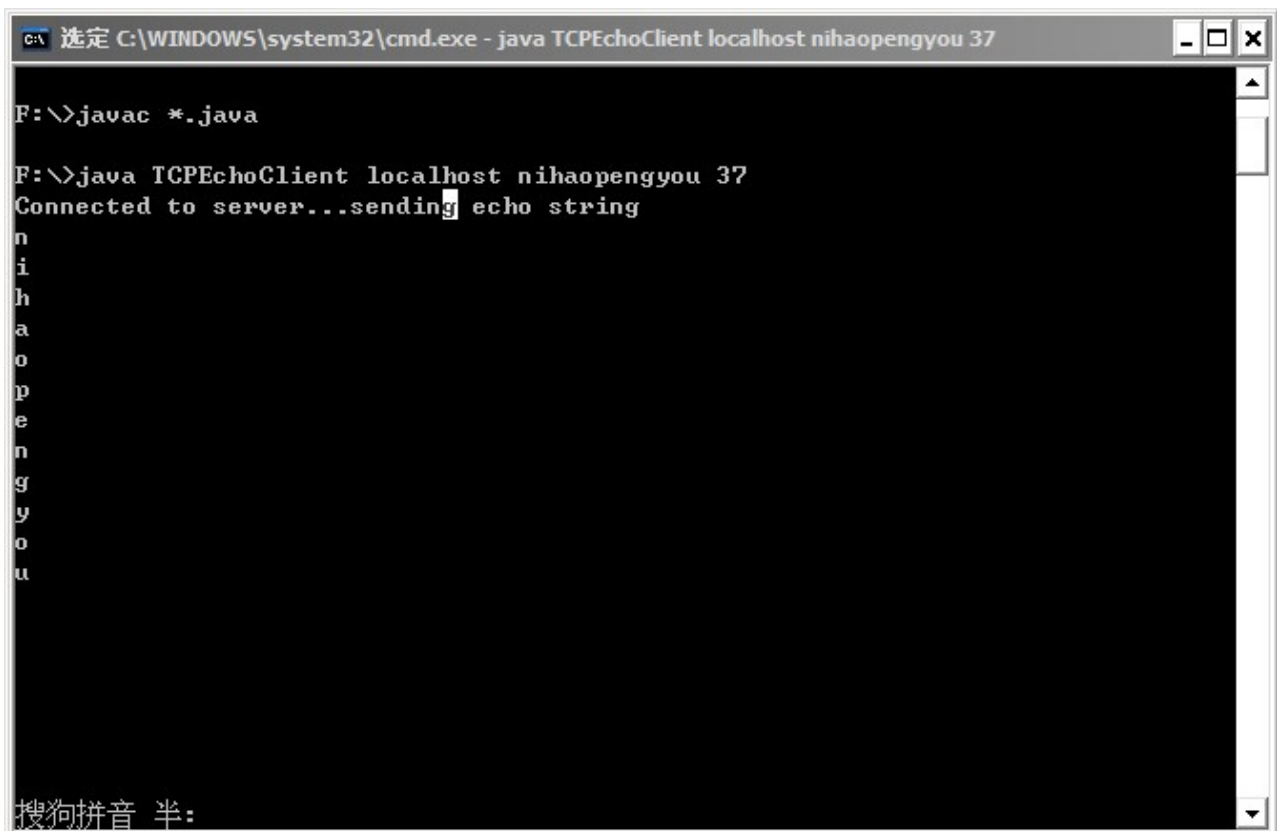


```
C:\ 选定 C:\WINDOWS\system32\cmd.exe - java TCPEchoServer 37

    at TCPEchoServer.main<TCPEchoServer.java:32>

F:\>java TCPEchoServer 37
Handling client at /127.0.0.1:2348

搜狗拼音 半:
```



```
C:\ 选定 C:\WINDOWS\system32\cmd.exe - java TCPEchoClient localhost nihaopengyou 37

F:\>javac *.java

F:\>java TCPEchoClient localhost nihaopengyou 37
Connected to server...sending echo string
n
i
h
a
o
p
e
n
g
y
o
u

搜狗拼音 半:
```

很明显，客户端程序在打印出最有一个字节后不再往下执行，没有执行其后面的 `System.out.println("Received: " + new String(data))`；这行代码，这是因为 `read()` 方法已经将数据读完，没有数据可读，但又没有返回 `-1`，因此在此处产生了阻塞。这便造成了 TCP Socket 通信的死锁问题。

问题的解决

问题就出现在 `read()` 方法上，这里的重点是 `read()` 方法何时返回 `-1`，在一般的文件读取中，这代表流的结束，亦即读取到了文件的末尾，但是在 Socket 套接字中，这样的概念很模糊，因为套接字中数据的末尾并没有所谓的结束标记，无法通过其自身表示传输的数据已经结束，那么究竟什么时候 `read()` 会返回 `-1` 呢？答案是：当 TCP 通信连接的一方关闭了套接字时。

再次分析改过后的代码，客户端用到了 `read()` 返回 `-1` 这个条件，而服务端也用到了，只有二者有一方关闭了 Socket，另一方的 `read()` 方法才会返回 `-1`，而在客户端打印输出前，二者都没有关闭 Socket，因此，二者的 `read()` 方法都不会返回 `-1`，程序便阻塞在此处，都不往下执行，这便造成了死锁。

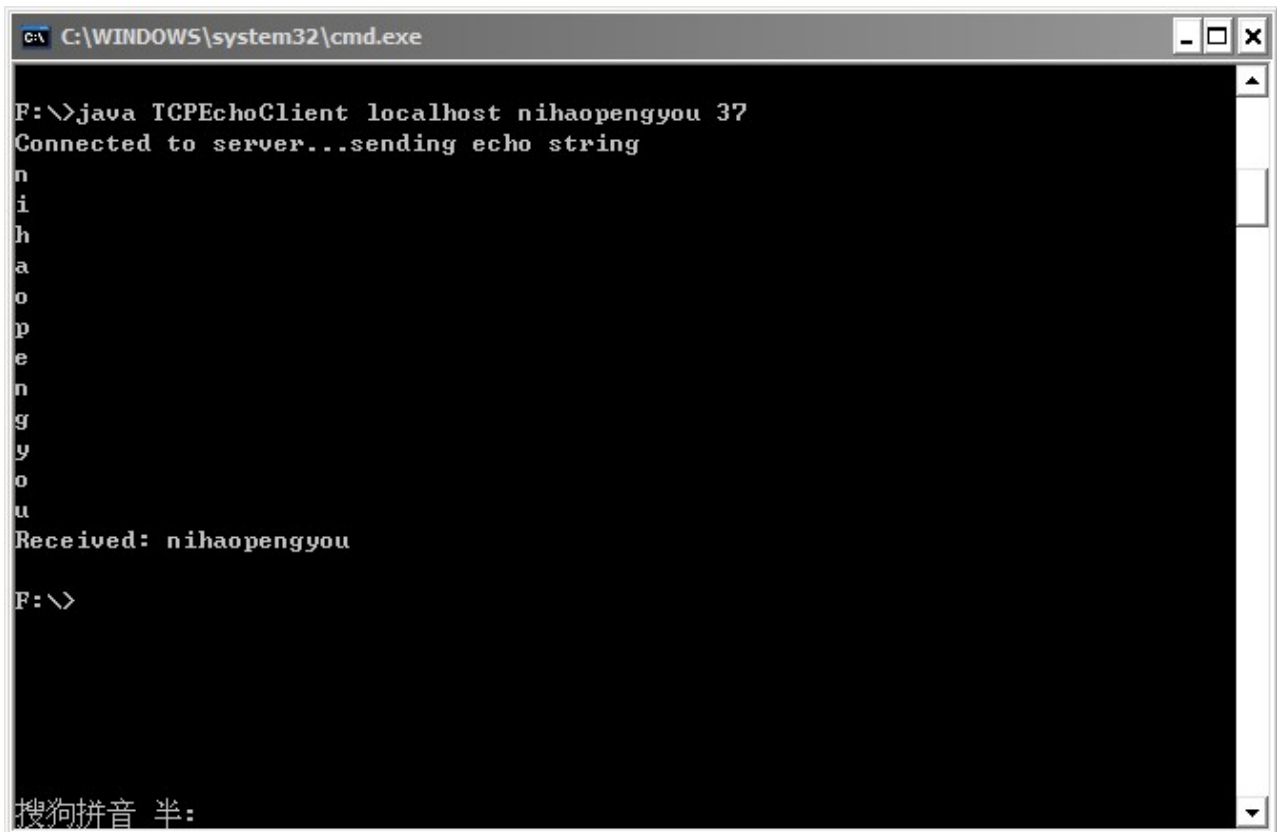
反过来，再看书上的给出的代码，在客户端代码的 while 循环中，我们的条件是 `totalBytesRcvd < data.length`，而不是 `(bytesRcvd = in.read()) != -1`，这样，客户端在收到与其发送相同的字节数之后便会退出 while 循环，再往下执行，便是关闭套接字，此时服务端的 `read()` 方法检测到客户端的关闭，便会返回 `-1`，从而继续往下执行，也将套接字关闭。因此，不会产生死锁。

那么，如果在客户端不知道反馈回来的数据的情况下，该如何避免死锁呢？Java 的 Socket 类提供了 `shutdownOutput()` 和 `shutdownInput()` 另个方法，用来分别只关闭 Socket 的输出流和输入流，而不影响其对应的输入流和输出流，那么我们便可以在客户端发送完数据后，调用 `shutdownOutput()` 方法将套接字的输出流关闭，这样，服务端的 `read()` 方法便会返回 `-1`，继续往下执行，最后关闭服务端的套接字，而后客户端的 `read()` 方法也会返回 `-1`，继续往下执行，直到关闭套接字。

客户端改变后的代码部分如下：

```
out.write(data); // Send the encoded string to the server
socket.shutdownOutput();
```

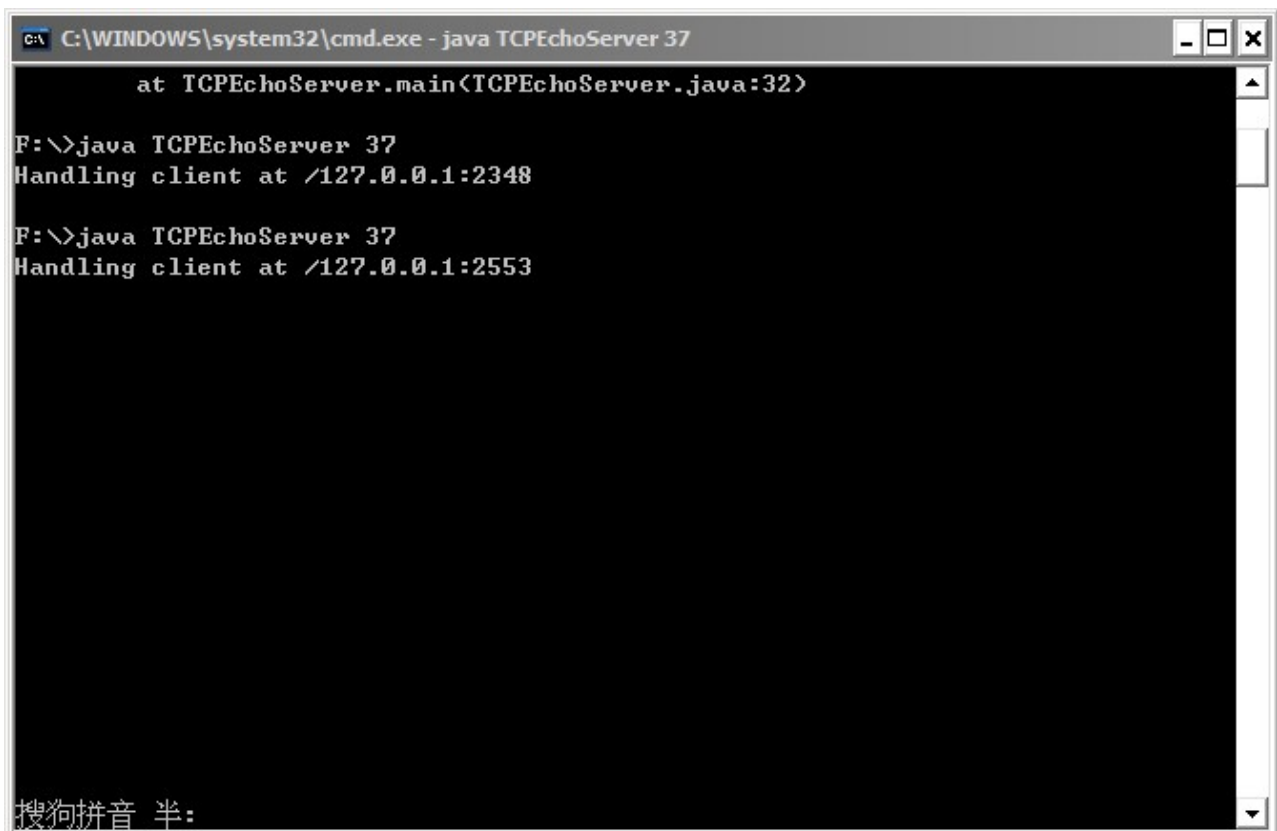
这样，便得到了预期的运行结果，如下：



```
C:\WINDOWS\system32\cmd.exe

F:\>java TCPEchoClient localhost nihaopengyou 37
Connected to server...sending echo string
n
i
h
a
o
p
e
n
g
y
o
u
Received: nihaopengyou

F:\>
```



```
C:\WINDOWS\system32\cmd.exe - java TCPEchoServer 37
at TCPEchoServer.main(TCPEchoServer.java:32)

F:\>java TCPEchoServer 37
Handling client at /127.0.0.1:2348

F:\>java TCPEchoServer 37
Handling client at /127.0.0.1:2553
```

总结

由于 read()方法只有在另一端关闭套接字的输出流时，才会返回 -1，而有时候由于我们不知道所要接收数据的大小，因此不得不用 read()方法返回 -1 这一判断条件，那么此时，合理的程序设计应该是先关闭网络输出流（亦即套接字的输出流），再关闭套接字。

8

Java NIO Socket VS 标准 IO Socket

简介

Java NIO 从 JDK1.4 引入，它提供了与标准 IO 完全不同的工作方式。

NIO 包（`java.nio.*`）引入了四个关键的抽象数据类型，它们共同解决传统的 I/O 类中的一些问题。

- Buffer：它是包含数据且用于读写的线形表结构。其中还提供了一个特殊类用于内存映射文件的 I/O 操作。
- Charset：它提供 Unicode 字符串影射到字节序列以及逆影射的操作。
- Channels：包含 socket, file 和 pipe 三种管道，它实际上是双向交流的通道。
- Selector：它将多元异步 I/O 操作集中到一个或多个线程中。

比较

数据的读写操作

标准的 IO 是基于字节流和字符流进行操作的，它不能前后移动流中的数据，而 NIO 是基于通道（Channel）和缓冲区（Buffer）进行操作的，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中，需要时可以在缓冲区中前后移动所保存的数据。

非阻塞

在标准 IO 的 Socket 编程中，套接字的某些操作可能会造成阻塞：`accept()`方法的调用可能会因为等待一个客户端连接而阻塞，`read()`方法也可能会因为没有数据可读而阻塞，`write()`方法在数据没有完全写入时也可能发生阻塞，阻塞发生时，该线程被挂起，什么也干不了。

阻塞式网络 IO 的特点

多线程处理多个连接。每个线程拥有自己的栈空间并且占用一些 CPU 时间。

每个线程遇在外部未准备好的时候，都会阻塞。阻塞的结果就是会带来大量的上下文切换。且大部分上下文切换可能是无意义的。比如假设一个线程监听一个端口，一天只会几次请求进来，但是该 cpu 不得不为该线程不断做上下文切换尝试，大部分的切换以阻塞告终。

NIO 则具有非阻塞的特性，可以通过对 channel 的阻塞行为的配置，实现非阻塞式的信道。在非阻塞情况下，线程在等待连接，写数据等（标准 IO 中的阻塞操作）的同时，也可以做其他事情，这便实现了线程的异步操作。

非阻塞式网络 IO 的特点

- 把整个过程切换成小的任务，通过任务间协作完成。

- 由一个专门的线程来处理所有的 IO 事件，并负责分发。
- 事件驱动机制：事件到的时候触发，而不是同步的去监视事件。
- 线程通讯：线程之间通过 wait,notify 等方式通讯。保证每次上下文切换都是有意义的。减少无谓的进程切换。

选择器

Java NIO 引入了选择器的概念，选择器可以监听多个通道的事件（比如：连接打开，数据到达）。因此，单个的线程可以监听多个数据通道，这也是非阻塞 IO 的核心。而在标准 IO 的 Socket 编程中，单个线程则只能在一个端口监听。



9

基于 NIO 的 TCP 通信



NIO 主要原理及使用

NIO 采取通道 (Channel) 和缓冲区(Buffer)来传输和保存数据,它是非阻塞式的 I/O,即在等待连接、读写数据(这些都是在一线程以客户端的程序中会阻塞线程的操作)的时候,程序也可以做其他事情,以实现线程的异步操作。

考虑一个即时消息服务器,可能有上千个客户端同时连接到服务器,但是在任何时刻只有非常少量的消息需要读取和分发(如果采用线程池或者一线程一客户端方式,则会非常浪费资源),这就需要一种方法能阻塞等待,直到有一个信道可以进行 I/O 操作。NIO 的 Selector 选择器就实现了这样的功能,一个 Selector 实例可以同时检查一组信道的 I/O 状态,它就类似一个观察者,只要我们把需要探知的 SocketChannel 告诉 Selector,我们接着做别的事情,当有事件(比如,连接打开、数据到达等)发生时,它会通知我们,传回一组 SelectionKey,我们读取这些 Key,就会获得我们刚刚注册过的 SocketChannel,然后,我们从这个 Channel 中读取数据,接着我们可以处理这些数据。

Selector 内部原理实际是在做一个对所注册的 Channel 的轮询访问,不断的轮询(目前就这一个算法),一旦轮询到一个 Channel 有所注册的事情发生,比如数据来了,它就会读取 Channel 中的数据,并对其进行处理。

要使用选择器,需要创建一个 Selector 实例,并将其注册到想要监控的信道上(通过 Channel 的方法实现)。最后调用选择器的 select()方法,该方法会阻塞等待,直到有一个或多个信道准备好了 I/O 操作或等待超时,或另一个线程调用了该选择器的 wakeup()方法。现在,在一个单独的线程中,通过调用 select()方法,就能检查多个信道是否准备好进行 I/O 操作,由于非阻塞 I/O 的异步特性,在检查的同时,我们也可以执行其他任务。

基于 NIO 的 TCP 连接的建立步骤

服务端

- 创建一个 Selector 实例;
- 将其注册到各种信道,并指定每个信道上感兴趣的 I/O 操作;
- 重复执行:
 - 调用一种 select()方法;
 - 获取选取的键列表;
 - 对于已选键集中的每个键:

- 获取信道，并从键中获取附件（如果为信道及其相关的 key 添加了附件的话）；
- 确定准备就绪的操纵并执行，如果是 accept 操作，将接收的信道设置为非阻塞模式，并注册到选择器；
- 如果需要，修改键的兴趣操作集；
- 从已选键集中移除键。

客户端

与基于多线程的 TCP 客户端大致相同，只是这里是通过信道建立连接，但在等待连接建立及读写时，我们可以异步地执行其他任务。

基于 NIO 的 TCP 通信 Demo

下面给出一个基于 NIO 的 TCP 通信的 Demo，客户端发送一串字符串到服务端，服务端将该字符串原原本本地反馈给客户端。

客户端代码及其详细注释如下：

```
import java.net.InetSocketAddress;
import java.net.SocketException;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class TCPEchoClientNonblocking {
    public static void main(String args[]) throws Exception{
        if ((args.length < 2) || (args.length > 3))
            throw new IllegalArgumentException("参数不正确");
        //第一个参数作为要连接的服务端的主机名或IP
        String server = args[0];
        //第二个参数为要发送到服务端的字符串
        byte[] argument = args[1].getBytes();
        //如果有第三个参数，则作为端口号，如果没有，则端口号设为7
        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
        //创建一个信道，并设为非阻塞模式
        SocketChannel clntChan = SocketChannel.open();
        clntChan.configureBlocking(false);
        //向服务端发起连接
        if (!clntChan.connect(new InetSocketAddress(server, servPort))){
            //不断地轮询连接状态，直到完成连接
```



```

while (!clntChan.finishConnect()){
    //在等待连接的时间里，可以执行其他任务，以充分发挥非阻塞IO的异步特性
    //这里为了演示该方法的使用，只是一直打印"."
    System.out.print(".");
}
}
//为了与后面打印的"."区别开来，这里输出换行符
System.out.print("\n");
//分别实例化用来读写的缓冲区
ByteBuffer writeBuf = ByteBuffer.wrap(argument);
ByteBuffer readBuf = ByteBuffer.allocate(argument.length);
//接收到的总的字节数
int totalBytesRcvd = 0;
//每一次调用read（）方法接收到的字节数
int bytesRcvd;
//循环执行，直到接收到的字节数与发送的字符串的字节数相等
while (totalBytesRcvd < argument.length){
    //如果用来向通道中写数据的缓冲区中还有剩余的字节，则继续将数据写入信道
    if (writeBuf.hasRemaining()){
        clntChan.write(writeBuf);
    }
    //如果read（）接收到-1，表明服务端关闭，抛出异常
    if ((bytesRcvd = clntChan.read(readBuf)) == -1){
        throw new SocketException("Connection closed prematurely");
    }
    //计算接收到的总字节数
    totalBytesRcvd += bytesRcvd;
    //在等待通信完成的过程中，程序可以执行其他任务，以体现非阻塞IO的异步特性
    //这里为了演示该方法的使用，同样只是一直打印"."
    System.out.print(".");
}
//打印出接收到的数据
System.out.println("Received: " + new String(readBuf.array(), 0, totalBytesRcvd));
//关闭信道
clntChan.close();
}
}

```

服务端用单个线程监控一组信道，代码如下：

```

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.util.Iterator;

```

```

public class TCPServerSelector{
    //缓冲区的长度
    private static final int BUFSIZE = 256;
    //select方法等待信道准备好的最长时间
    private static final int TIMEOUT = 3000;
    public static void main(String[] args) throws IOException {
        if (args.length < 1){
            throw new IllegalArgumentException("Parameter(s): <Port> ...");
        }
        //创建一个选择器
        Selector selector = Selector.open();
        for (String arg : args){
            //实例化一个信道
            ServerSocketChannel listnChannel = ServerSocketChannel.open();
            //将该信道绑定到指定端口
            listnChannel.socket().bind(new InetSocketAddress(Integer.parseInt(arg)));
            //配置信道为非阻塞模式
            listnChannel.configureBlocking(false);
            //将选择器注册到各个信道
            listnChannel.register(selector, SelectionKey.OP_ACCEPT);
        }
        //创建一个实现了协议接口的对象
        TCPProtocol protocol = new EchoSelectorProtocol(BUFSIZE);
        //不断轮询select方法，获取准备好的信道所关联的Key集
        while (true){
            //一直等待,直至有信道准备好了I/O操作
            if (selector.select(TIMEOUT) == 0){
                //在等待信道准备的同时，也可以异步地执行其他任务，
                //这里只是简单地打印"."
                System.out.print(".");
                continue;
            }
            //获取准备好的信道所关联的Key集合的iterator实例
            Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
            //循环取得集合中的每个键值
            while (keyIter.hasNext()){
                SelectionKey key = keyIter.next();
                //如果服务端信道感兴趣的I/O操作为accept
                if (key.isAcceptable()){
                    protocol.handleAccept(key);
                }
                //如果客户端信道感兴趣的I/O操作为read
                if (key.isReadable()){
                    protocol.handleRead(key);
                }
            }
        }
    }
}

```

```

    }
    //如果该键值有效，并且其对应的客户端信道感兴趣的I/O操作为write
    if (key.isValid() && key.isWritable()) {
        protocol.handleWrite(key);
    }
    //这里需要手动从键集中移除当前的key
    keyIter.remove();
}
}
}
}

```

这里为了使不同协议都能方便地使用这个基本的服务模式，我们把信道中与具体协议相关的处理各种 I/O 的操作分离了出来，定义了一个接口，如下：

```

import java.nio.channels.SelectionKey;
import java.io.IOException;

/**
 *该接口定义了通用TCPSelectorServer类与特定协议之间的接口，
 *它把与具体协议相关的处理各种I/O的操作分离了出来，
 *以使不同协议都能方便地使用这个基本的服务模式。
 */
public interface TCPProtocol{
    //accept I/O形式
    void handleAccept(SelectionKey key) throws IOException;
    //read I/O形式
    void handleRead(SelectionKey key) throws IOException;
    //write I/O形式
    void handleWrite(SelectionKey key) throws IOException;
}

```

接口的实现类代码如下：

```

import java.nio.channels.SelectionKey;
import java.nio.channels.SocketChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.ByteBuffer;
import java.io.IOException;

public class EchoSelectorProtocol implements TCPProtocol {
    private int bufSize; // 缓冲区的长度
    public EchoSelectorProtocol(int bufSize){
        this.bufSize = bufSize;
    }
}

```

```

//服务端信道已经准备好了接收新的客户端连接
public void handleAccept(SelectionKey key) throws IOException {
    SocketChannel clntChan = ((ServerSocketChannel) key.channel()).accept();
    clntChan.configureBlocking(false);
    //将选择器注册到连接到的客户端信道，并指定该信道key值的属性为OP_READ，同时为该信道指定关联的附件
    clntChan.register(key.selector(), SelectionKey.OP_READ, ByteBuffer.allocate(bufSize));
}

//客户端信道已经准备好了从信道中读取数据到缓冲区
public void handleRead(SelectionKey key) throws IOException{
    SocketChannel clntChan = (SocketChannel) key.channel();
    //获取该信道所关联的附件，这里为缓冲区
    ByteBuffer buf = (ByteBuffer) key.attachment();
    long bytesRead = clntChan.read(buf);
    //如果read ( ) 方法返回-1，说明客户端关闭了连接，那么客户端已经接收到了与自己发送字节数相等的数据，可以安全地关闭
    if (bytesRead == -1){
        clntChan.close();
    }else if(bytesRead > 0){
        //如果缓冲区总读入了数据，则将该信道感兴趣的操作设置为为可读可写
        key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
    }
}

//客户端信道已经准备好了将数据从缓冲区写入信道
public void handleWrite(SelectionKey key) throws IOException {
    //获取与该信道关联的缓冲区，里面有之前读取到的数据
    ByteBuffer buf = (ByteBuffer) key.attachment();
    //重置缓冲区，准备将数据写入信道
    buf.flip();
    SocketChannel clntChan = (SocketChannel) key.channel();
    //将数据写入到信道中
    clntChan.write(buf);
    if (!buf.hasRemaining()){
        //如果缓冲区中的数据已经全部写入了信道，则将该信道感兴趣的操作设置为可读
        key.interestOps(SelectionKey.OP_READ);
    }
    //为读入更多的数据腾出空间
    buf.compact();
}
}

```

执行结果如下：

```

C:\ 选定 C:\WINDOWS\system32\cmd.exe

E:\>
E:\>
E:\>
E:\>
E:\>
E:\>
E:\>
E:\>java TCPEchoClientNonblocking localhost nihaopengyou 2002
..Received: nihaopengyou
E:\>java TCPEchoClientNonblocking localhost nihaopengyou 2005
..Received: nihaopengyou
E:\>
搜狗拼音 半:

```

```

C:\ 选定 C:\WINDOWS\system32\cmd.exe - java TCPSelector 2000 2001 2002 2003 20...
或批处理文件。
E:\>n.nio.ch.IOUtil.readIntoNativeBuffer(Unkno
'n.nio.ch.IOUtil.readIntoNativeBuffer' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
E:\>n.nio.ch.IOUtil.read(Unknown Source)
'n.nio.ch.IOUtil.read' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
E:\>
E:\>javac *.java
E:\>java TCPSelector 2000 2001 2002 2003 2004 2005
.....
搜狗拼音 半:

```

说明：以上的服务端程序，select()方法第一次能选择出来的准备好的信道都是服务端信道，其关联键值的属性都为 OP_ACCEPT，亦及有效操作都为 accept，在执行 handleAccept 方法时，为取得连接的客户端信道也进

行了注册，属性为 `OP_READ`，这样下次轮询调用 `select()` 方法时，便会检查到对 `read` 操作感兴趣的客户端信道（当然也有可能有关联 `accept` 操作兴趣集的信道），从而调用 `handleRead` 方法，在该方法中又注册了 `OP_WRITE` 属性，那么第三次调用 `select()` 方法时，便会检测到对 `write` 操作感兴趣的客户端信道（当然也有可能有关联 `read` 操作兴趣集的信道），从而调用 `handleWrite` 方法。

结果：从结果中很明显地可以看出，服务器端在等待信道准备好的时候，线程没有阻塞，而是可以执行其他任务，这里只是简单的打印".", 客户端在等待连接和等待数据读写完成的时候，线程没有阻塞，也可以执行其他任务，这里也正是简单的打印".".

需要注意的地方

1. 对于非阻塞 `SocketChannel` 来说，一旦已经调用 `connect()` 方法发起连接，底层套接字可能既不是已经连接，也不是没有连接，而是正在连接。由于底层协议的工作机制，套接字可能会在这个状态一直保持下去，这时候就需要循环地调用 `finishConnect()` 方法来检查是否完成连接，在等待连接的同时，线程也可以做其他事情，这便实现了线程的异步操作。
2. `write()` 方法的非阻塞调用只会写出其能够发送的数据，而不会阻塞等待所有数据，而后一起发送，因此在调用 `write()` 方法将数据写入信道时，一般要用到 `while` 循环，如：

```
while ( buf.hasRemaining() )
    channel.write(buf);
```

1. 任何对 `key`（信道）所关联的兴趣操作集的改变，都只在下次调用了 `select()` 方法后才会生效。
2. `selectedKeys()` 方法返回的键集是可修改的，实际上在两次调用 `select()` 方法之间，都必须手动将其清空，否则，它就会在下次调用 `select()` 方法时仍然保留在集合中，而且可能会有无用的操作来调用它，换句话说，`select()` 方法只会在已有的所选键集上添加键，它们不会创建新的键集。
3. 对于 `ServerSocketChannel` 来说，`accept` 是唯一的有效操作，而对于 `SocketChannel` 来说，有效操作包括读、写和连接，另外，对于 `DatagramChannel`，只有读写操作是有效的。



10

深入剖析 Socket——数据传输的底层实现



底层数据结构

如果不理解套接字的具体实现所关联的数据结构和底层协议的工作细节，就很难抓住网络编程的精妙之处，对于 TCP 套接字来说，更是如此。套接字所关联的底层的数据结构集包含了特定 Socket 实例所关联的信息。比附，套接字结构除其他信息外还包含：

- 该套接字所关联的本地和远程互联网地址和端口号。
- 一个 FIFO（First In First Out）队列，用于存放接收到的等待分配的数据，以及一个用于存放等待传输的数据的队列。
- 对于 TCP 套接字，还包含了与打开和关闭 TCP 握手相关的额定协议状态信息。

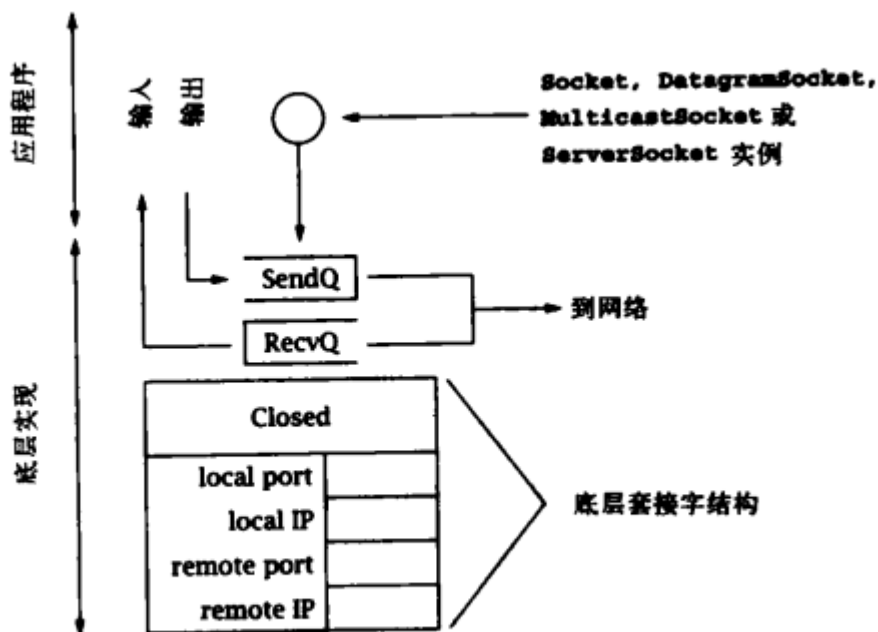


图6-1 套接字关联的数据结构

了解这些数据结构，以及底层协议如何对其进行影响是非常有用的，因为它们控制了各种 Socket 对象行为的各个方面。例如，由于 TCP 提供了一种可信赖的字节流服务，任何写入 Socket 和 OutputStream 的数据副本都必须保留，直到连接的另一端将这些数据成功接收。向输出流写数据并不意味着数据实际上已经被发送——它们只是被复制到了本地缓冲区，就算在 Socket 的 OutputStream 上进行 flush() 操作，也不能保证数据能够立即发送到信道。此外，字节流服务的自身属性决定了其无法保留输入流中消息的边界信息。

数据传输的底层实现

在使用 TCP 套接字时，需要记住的最重要的一点是：不能假设在连接的一端将数据写入输出流和在另一端从输入流读出数据之间有任何的一致性。尤其是在发送端由单个输出流的 `write()` 方法传输的数据，可能会通过另一端的多个输入流的 `read()` 方法获取，而一个 `read()` 方法可能会返回多个 `write()` 方法传输的数据。

一般来讲，我们可以认为 TCP 连接上发送的所有字节序列在某一瞬间被分成了 3 个 FIFO 队列：

- **SendQ**：在发送端底层实现中缓存的字节，这些字节已经写入输出流，但还没在接收端成功接收。它占用大约 37KB 内存。
- **RecvQ**：在接收端底层实现中缓存的字节，这些字节等待分配到接收程序——即从输入流中读取。它占用大约 25KB 内存。
- **Delivered**：接收者从输入流已经读取到的字节。

当我们调用 `OutputStream` 的 `write()` 方法时，将向 **SendQ** 追加字节。

TCP 协议负责将字节按顺序从 **SendQ** 移动到 **RecvQ**。这里有重要的一点需要明确：这个转移过程无法由用户程序控制或直接观察到，并且在块中发生，这些块的大小在一定程度上独立于传递给 `write()` 方法的缓冲区大小。

接收程序从 `Socket` 的 `InputStream` 读取数据时，字节就从 **RecvQ** 移动到 **Delivered** 中，而转移的块的大小依赖于 **RecvQ** 中的数据量和传递给 `read()` 方法的缓冲区的大小。

示例分析

为了展示这种情况，考虑如下程序：

```
byte[] buffer0 = new byte[1000];
byte[] buffer1 = new byte[2000];
byte[] buffer2 = new byte[5000];
...
Socket s = new Socket(destAddr, destPort);
OutputStream out = s.getOutputStream();
...
out.write(buffer0);
...
out.write(buffer1);
...
out.write(buffer2);
...
s.close();
```

其中，圆点代表了设置缓冲区数据的代码，但不包含对 `out.write()` 方法的调用。这个 TCP 连接向接收端传输 8000 字节，在连接的接收端，这 8000 字节的分组方式取决于连接两端的 `out.write()` 方法和 `in.read()` 方法的调用时间差，以及提供给 `in.read()` 方法的缓冲区的大小。

下图展示了 3 次调用 `out.write()` 方法后，另一端调用 `in.read()` 方法前，以上 3 个队列的一种可能状态。不同的阴影效果分别代表了上文中 3 次调用 `write()` 方法传输的不同数据：

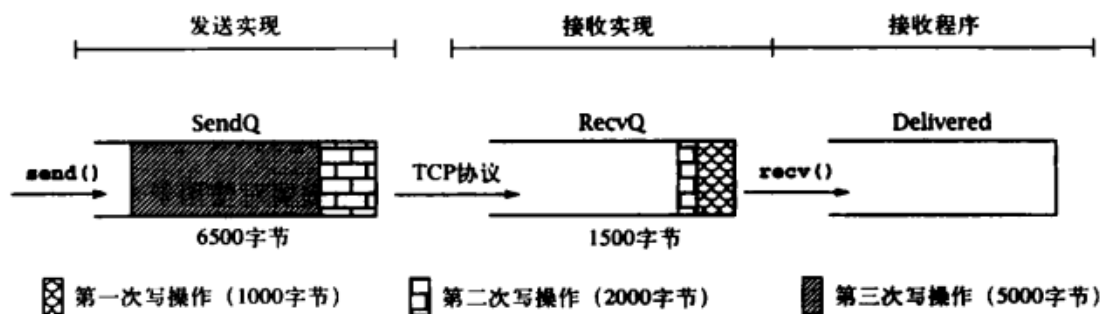


图6-2 3次调用`write()`方法后3个队列的状态

现在假设接收者调用 `read()` 方法时使用的缓冲区数组大小为 2000 字节，`read()` 调用则会将把 `RecvQ` 中的 1500 字节全部移动到数组中，返回值为 1500。注意，这些数据中包含了第一次和第二次调用 `write()` 方法时传输的字节，再过一段时间，当 TCP 连接传完更多数据后，这三部分的状态可能如下图所示：

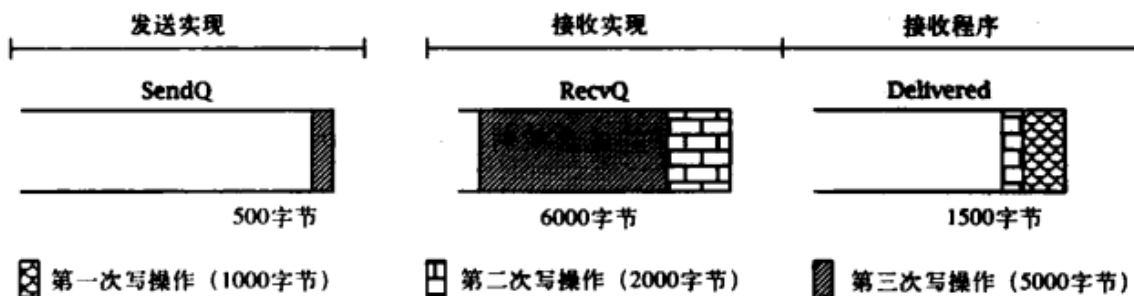


图6-3 第一次调用`read()`方法后

如果接收者现在调用 `read()` 方法时使用 4000 字节的缓冲区数组，将有很多字节从 `RecvQ` 队列转移到 `Delivered` 队列中，这包括第二次调用 `write()` 方法时剩下的 1500 字节加上第三次调用 `write()` 方法的 2500 字节。此时，队列的状态如下图：

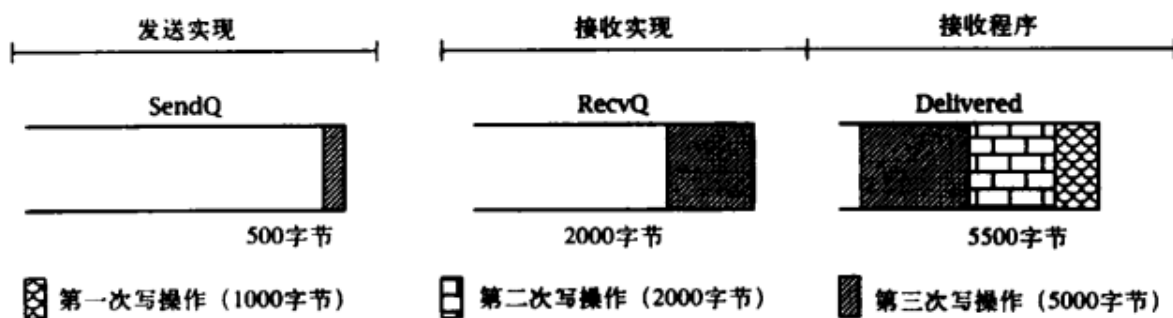


图6-4 另一次调用`read()`后

下次调用 `read()` 方法返回的字节数，取决于缓冲区数组的大小，亦及发送方套接字通过网络向接收方实现传输数据的时机。数据从 `sendQ` 到 `RecvQ` 缓冲区的移动过程对应用程序协议的设计有重要的指导性。



T

11



深入剖析 Socket——TCP 通信中由于底层队列填满而造成的死锁问题



基础准备

首先需要明白数据传输的底层实现机制，在上一章中有详细的介绍，我们提到了 SendQ 和 RecvQ 缓冲队列，这两个缓冲区的容量在具体实现时会受一定的限制，虽然它们使用的实际内存大小会动态地增长和收缩，但还是需要一个硬性的限制，以防止行为异常的程序所控制的单一 TCP 连接将系统的内存全部消耗。正式由于缓冲区的容量有限，它们可能会被填满，事实也正是如此，如果与 TCP 的流量控制机制结合使用，则可能导致一种形式的死锁。

一旦 RecvQ 已满，TCP 流控制机制就会产生作用（使用流控制机制的目的是为了保证发送者不会传输太多数据，从而超出了接收系统的处理能力），它将阻止传输发送端主机的 SendQ 中的任何数据，直到接收者调用输入流的 read()方法将 RecvQ 中的数据移除一部分到 Delivered 中，从而腾出了空间。发送端可以持续地写出数据，直到 SendQ 队列被填满，如果 SendQ 队列已满时调用输出流的 write()方法，则会阻塞等待，直到有一些字节被传输到 RecvQ 队列中，如果此时 RecvQ 队列也被填满了，所有的操作都将停止，直到接收端调用了输入流的 read()方法将一些字节传输到了 Delivered 队列中。

引出问题

我们假设 SendQ 队列和 RecvQ 队列的大小分别为 SQS 和 RQS。将一个大小为 n 的字节数组传递给发送端 write()方法调用，其中 $n > SQS$ ，直到有至少 $n - SQS$ 字节的数据传递到接收端主机的 RecvQ 队列后，该方法才返回。如果 n 的大小超过了 $SQS + RQS$ ，write()方法将在接收端从输入流读取了至少 $n - (SQS + RQS)$ 字节后才会返回。如果接收端没有调用 read()方法，大数据量的发送是无法成功的。特别是连接的两端同时分别调用它的输出流的 write()方法，而他们的缓冲区大小又大于 $SQS + RQS$ 时，将会发生死锁：两个 write 操作都不能完成，两个程序都将永远保持阻塞状态。

下面考虑一个具体的例子，即主机 A 上的程序和主机 B 上的程序之间的 TCP 连接。假设 A 和 B 上的 SQS 和 RQS 都是 500 字节，下图展示了两个程序试图同时发送 1500 字节时的情况。主机 A 上的程序中的前 500 字节已经传输到另一端，另外 500 字节已经复制到了主机 A 的 SendQ 队列中，余下的 500 字节则无法发送，write()方法将无法返回，直到主机 B 上程序的 RecvQ 队列有空间空出来，然而不幸的是 B 上的程序也遇到了同样的情况，而二者都没有及时调用 read()方法从自己的 RecvQ 队列中读取数据到 Delivered 队列中。因此，两个程序的 write()方法调用都永远无法返回，产生死锁。因此，在写程序时，要仔细设计协议，以避免在两个方向上传输大量数据时产生死锁。

示例分析

回顾前面几篇文章中的 TCP 通信的示例代码，基本都是只调用一次 `write()` 方法将所有的数据写出，而且我们测试的数据量也不大。考虑一个压缩字节的 Demo，客户端从文件中读取字节，发送到服务端，服务端将受到的文件压缩后反馈给客户端。

这里先给出代码，客户端代码如下：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class CompressClientNoDeadlock {

    public static final int BUFSIZE = 256; // Size of read buffer

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

        if (args.length != 3) // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Server> <Port> <File>");

        String server = args[0];          // Server name or IP address
        int port = Integer.parseInt(args[1]); // Server port
        String filename = args[2];         // File to read data from

        // Open input and output file (named input.gz)
        final FileInputStream fileIn = new FileInputStream(filename);
        FileOutputStream fileOut = new FileOutputStream(filename + ".gz");

        // Create socket connected to server on specified port
        final Socket sock = new Socket(server, port);

        // Send uncompressed byte stream to server
        Thread thread = new Thread() {
            public void run() {
                try {
                    SendBytes(sock, fileIn);
                } catch (Exception ignored) {}
            }
        };
    }
};
```

```

thread.start();

// Receive compressed byte stream from server
InputStream sockIn = sock.getInputStream();
int bytesRead;           // Number of bytes read
byte[] buffer = new byte[BUFSIZE]; // Byte buffer
while ((bytesRead = sockIn.read(buffer)) != -1) {
    fileOut.write(buffer, 0, bytesRead);
    System.out.print("R"); // Reading progress indicator
}
System.out.println();    // End progress indicator line

sock.close(); // Close the socket and its streams
fileIn.close(); // Close file streams
fileOut.close();
}

public static void SendBytes(Socket sock, InputStream fileIn)
    throws IOException {

    OutputStream sockOut = sock.getOutputStream();
    int bytesRead;           // Number of bytes read
    byte[] buffer = new byte[BUFSIZE]; // Byte buffer
    while ((bytesRead = fileIn.read(buffer)) != -1) {
        sockOut.write(buffer, 0, bytesRead);
        System.out.print("W"); // Writing progress indicator
    }
    sock.shutdownOutput(); // Done sending
}
}

```

死锁问题的产生原因在客户端上，因此，服务端的具体代码我们不再给出，服务端采取边读边写的策略。

下面我们边对上面可能产生的问题进行分析。对该示例而言，当需要传递的文件容量不是很大时，程序运行正常，也能得到预期的结果，但如果尝试运行该客户端并传递给它一个大文件，该文件压缩后仍然很大（在此，大的精确定义取决于程序运行的系统，不过压缩后依然超过 2MB 的文件应该就可以使该程序产生死锁问题），那么客户端将打印出一堆 W 后停止，而且不会打印出任何 R，程序也不会终止。

为什么会产生这种情况呢？我们来看程序，客户端很明显是一边读取本地文件中的数据，一边调用输出流的 write() 方法，将数据送入客户端主机的 SendQ 队列，直到文件中的数据被读取完，客户端才调用输入流的 read() 方法，读取服务端发送回来的数据。

考虑这种情况：客户端和服务端的 SendQ 队列和 RecvQ 队列中都有 500 字节的数据空间，而客户端发送了一个 10000 字节的文件，同时假设对于这个文件，服务端读取 1000 字节并返回 500 字节，即压缩比为 2:1，当客

户端发送了 2000 字节后，服务端将最终全部读取这些字节，并发回 1000 字节，由于客户端此时并没有调用输入流的 `read()` 方法从客户端主机的 `RecvQ` 队列中移出数据到 `Delivered`，因此，此时客户端的 `RecvQ` 队列和服务端的 `SendQ` 队列都被填满了，此时客户端还在继续发送数据，又发送了 1000 字节的数据，并且被服务端全部读取，但此时服务端的 `write` 操作尝试都已被阻塞，不能继续发送数据给客户端，当客户端再发送了另外的 1000 字节数据后，客户端的 `SendQ` 队列和服务端的 `RecvQ` 队列都将被填满，后续的客户端 `write` 操作也将阻塞，从而形成死锁。

解决方案

如何解决这个问题呢？造成死锁产生的原因是因为客户端在发送数据的同时，没有及时读取反馈回来的数据，从而使数据都阻塞在了底层的传输队列中。

方案一是在编写客户端程序时，使客户端一边循环调用输出流的 `write()` 方法向服务端发送数据，一边循环调用输入流的 `read()` 方法读取从服务端反馈回来的数据，但这也不能完全保证不会产生死锁。

更好的解决方案是在不同的线程中执行客户端的 `write` 循环和 `read` 循环。一个线程从文件中反复读取未压缩的字节并将其发送给服务器，直到文件的结尾，然后调用该套接字的 `shutdownOutput()` 方法。另一个线程从服务端的输入流中不断读取压缩后的字节，并将其写入输出文件，直到到达了输入流的结尾（服务器关闭了套接字）。这样，便可以实现一边发送，一边读取，而且如果一个线程阻塞了，另一个线程仍然可以独立执行。这样我们可以对客户端代码进行简单的修改，将 `SendBytes()` 方法调用放到一个线程中：

```
Thread thread = new Thread() {
    public void run() {
        try {
            SendBytes(sock, fileIn);
        } catch (Exception ignored) {}
    }
};
thread.start();
```

当然，解决这个问题也可以不使用多线程，而是使用 NIO 机制（`Channel` 和 `Selector`）。



12

深入剖析 Socket——TCP 套接字的生命周期



建立 TCP 连接

新的 Socket 实例创建后，就立即能用于发送和接收数据。也就是说，当 Socket 实例返回时，它已经连接到了一个远程终端，并通过协议的底层实现完成了 TCP 消息或握手信息的交换。

客户端连接的建立

Socket 构造函数的调用与客户端连接建立时所关联的协议事件之间的关系下图所示：

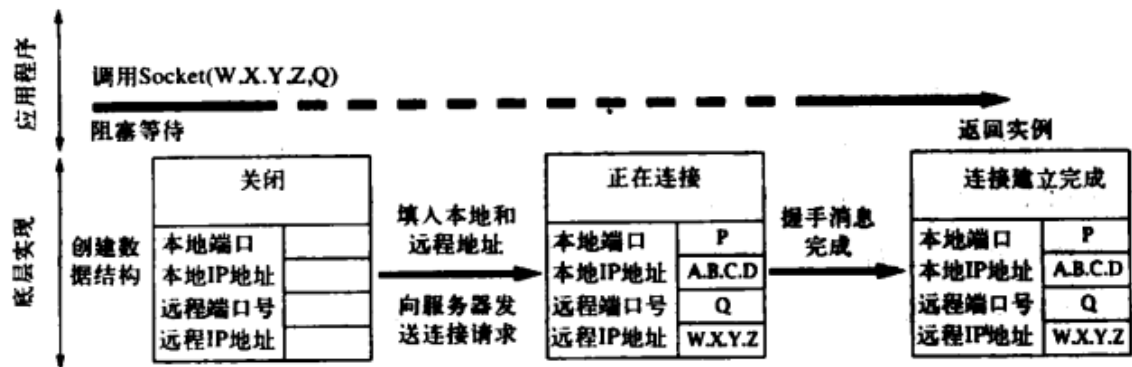


图6-6 客户端连接建立

当客户端以服务器端的互联网地址 W.X.Y.Z 和端口号 Q 作为参数，调用 Socket 的构造函数时，底层实现将创建一个套接字实例，该实例的初始状态是关闭的。TCP 开放握手也称为3次握手，这通常包括 3 条消息：一条从客户端到服务端的连接请求，一条从服务端到客户端的确认消息，以及另一条从客户端到服务端的确认消息。对客户端而言，一旦它收到了服务端发来的确认消息，就立即认为连接已经建立。通常这个过程发生的很快，但连接请求消息或服务端的回复消息都有可能传输过程中丢失，因此 TCP 协议实现将以递增的时间间隔重复发送几次握手消息。如果 TCP 客户端在一段时间后还没有收到服务端的回复消息，则发生超时并放弃连接。如果服务端并没有接收连接，则服务端的 TCP 将发送一条拒绝消息而不是确认消息。

服务端连接的建立

当客户端的事件序列则有所不同。服务端首先创建一个 ServerSocket 实例，并将其与已知端口相关联（在此为 Q），套接字实现为新的 ServerSocket 实例创建一个底层数据结构，并就 Q 赋给本地端口，并将特定的通配符（*）赋给本地 IP 地址（服务器可能有多个 IP 地址，不过通常不会指定该参数），如下图所示：

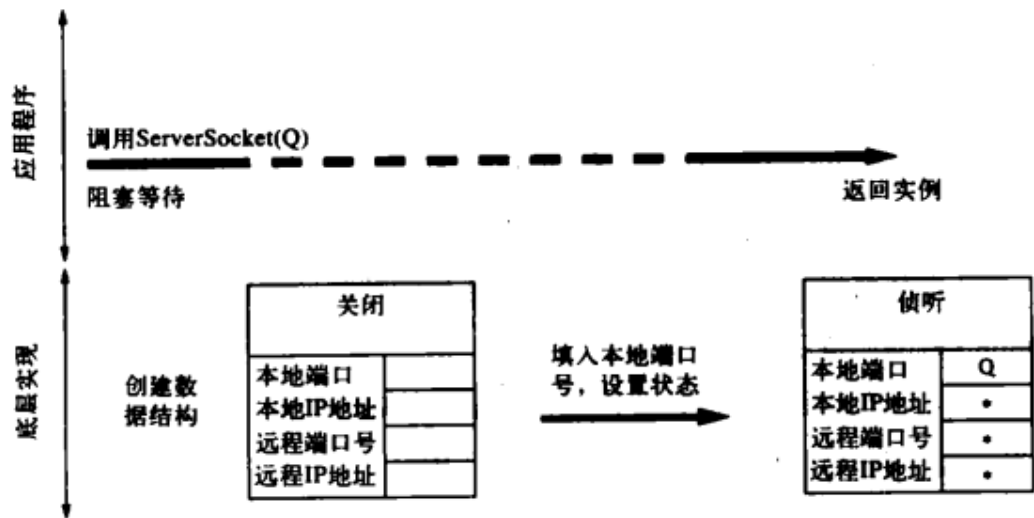


图6-7 服务器端的套接字设置

现在服务端可以调用 ServerSocket 的 accept()方法, 来将阻塞等待客户端连接请求的到来。当客户端的连接请求到来时, 将为连接创建一个新的套接字数据结构。该套接字的地址根据到来的分组报文设置: 分组报文的目标互联网地址和端口号成为该套接字的本地互联网地址和端口号; 而分组报文的源地址和端口号则成为改套接字的远程互联网地址和端口号。注意, 新套接字的本地端口号总是与 ServerSocket 的端口号一致。除了要创建一个新的底层套接字数据结构外, 服务端的 TCP 实现还要向客户端发送一个 TCP 握手确认消息。如下图所示:

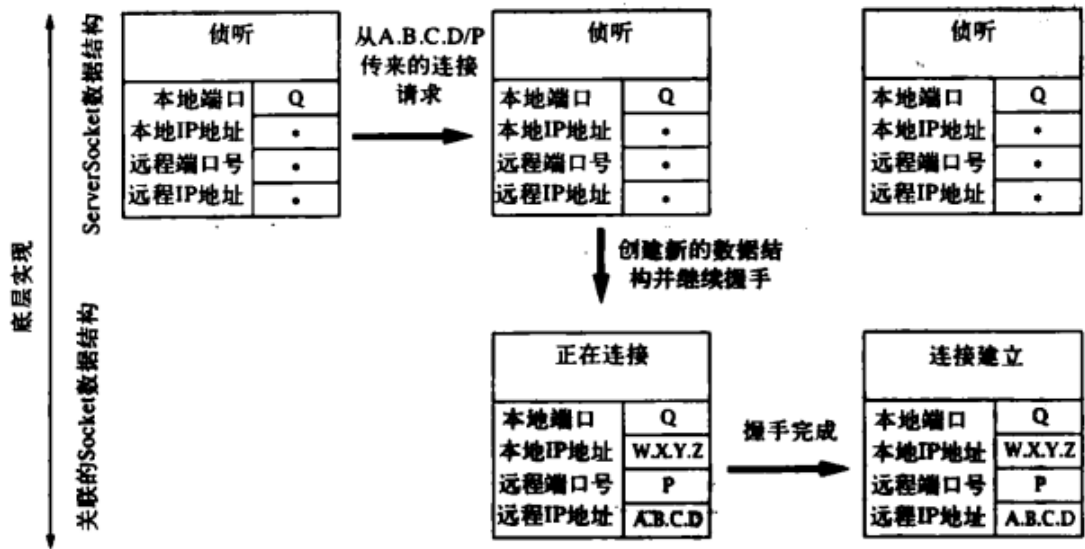
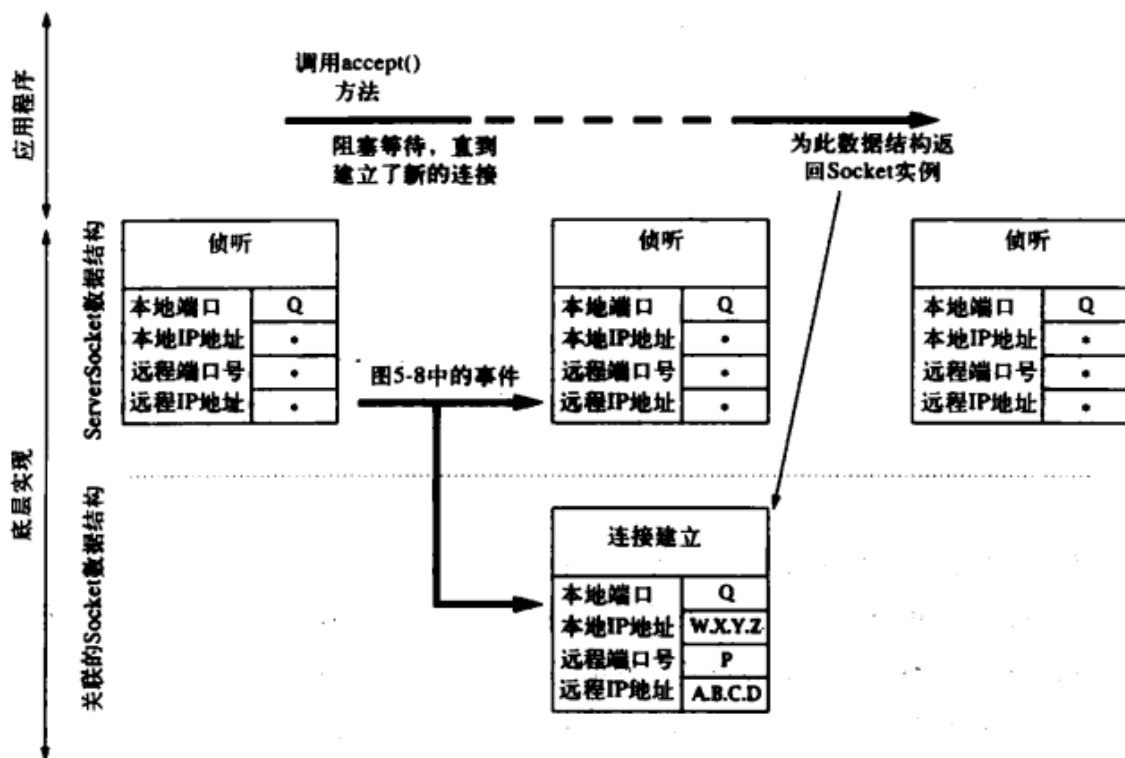


图6-8 处理传入的连接请求

但是, 对于服务端来说, 在接收到客户端发来的第3条消息之前, 服务端 TCP 并不会认为握手消息已经完成。一旦收到客户端发来的第 3 条消息, 则表示连接已建立, 此时一个新的数据结构将从服务端所关联的列表中移除, 并为创建一个 Socket 实例, 作为 accept()方法的返回值。如下图所示:

图 6-9 `accept()` 处理

这里有非常重要的一点需要注意，在 `ServerSocket` 关联的列表中的每个数据结构，都代表了一个与另一端的客户端已经完成建立的 TCP 连接。实际上，客户只要收到了开放握手的第 2 条消息，就可以立即发送数据——这可能比服务端调用 `accept()` 方法为其获取一个 `Socket` 实例要早很长时间。

关闭 TCP 连接

TCP 协议有一个优雅的关闭机制，以保证应用程序在关闭时不必担心正在传输的数据会丢失，这个机制还可以设计为允许两个方向的数据传输相互独立地终止。关闭机制的工作流程是：应用程序通过调用连接套接字的 `close()` 方法或 `shutdownOutput()` 方法表明数据已经发送完毕。底层 TCP 实现首先将留在 `SendQ` 队列中的数据传输出去（这还要依赖于另一端的 `RecvQ` 队列的剩余空间），然后向另一端发送一个关闭 TCP 连接的握手消息。该关闭握手消息可以看做流结束的标志：它告诉接收端 TCP 不会再有新的数据传入 `RecvQ` 队列了。注意：关闭握手消息本身并没有传递给接收端应用程序，而是通过 `read()` 方法返回 `-1` 来指示其在字节流中的位置。而正在关闭的 TCP 将等待其关闭握手消息的确认消息，该确认消息表明在连接上传输的所有数据已经安全地传输到了 `RecvQ` 中。只要收到了确认消息，该连接变成了“半关闭”状态。直到连接的另一个方向上收到了对称的握手消息后，连接才完全关闭——也就是说，连接的两端都表明它们没有数据发送了。

TCP 连接的关闭事件序列可能以两种方式发生：一种方式是先由一个应用程序调用 `close()` 方法或 `shutdownOutput()` 方法，并在另一端调用 `close()` 方法之前完成其关闭握手消息；另一种方式是两端同时调用 `close()` 方法，他

们的关闭握手消息在网络上交叉传输。下图展示了以第一种方式关闭连接时，发起关闭的一端底层实现中的事件序列：

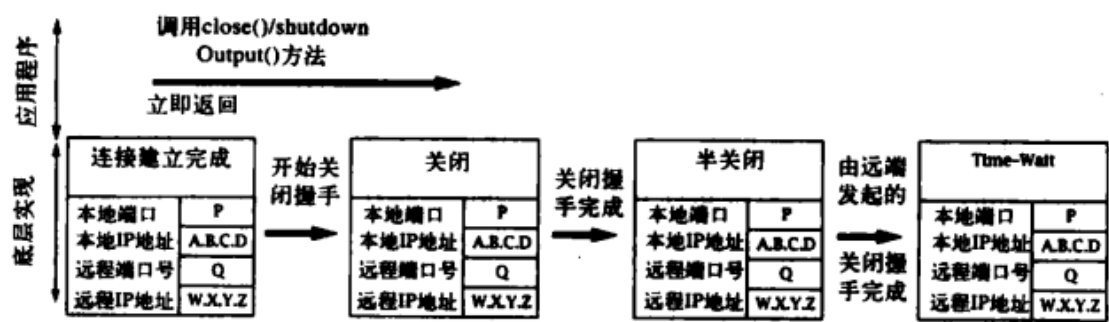


图6-10 首先关闭一端的TCP连接

注意，如果连接处于半关闭状态时，远程终端已经离开，那么本地底层数据结构则无限期地保持在该状态。当另一端的关闭握手消息到达后，则发回一条确认消息并将状态改为“Time—Wait”。虽然应用程序中相应的 Socket 实例可能早已消失，与之关联的底层数据结构还将在底层实现中继续存留几分钟。

对于没有首先发起关闭的一端，关闭握手消息达到后，它立即发回一个确认消息，并将连接状态改为“Close—Wait”。此时，只需要等待应用程序调用 Socket 的 close()方法。调用该方法后，将发起最终的关闭消息，并释放底层套接字数据结构。下图展示了没有首先发起关闭的一端底层实现中的事件序列：

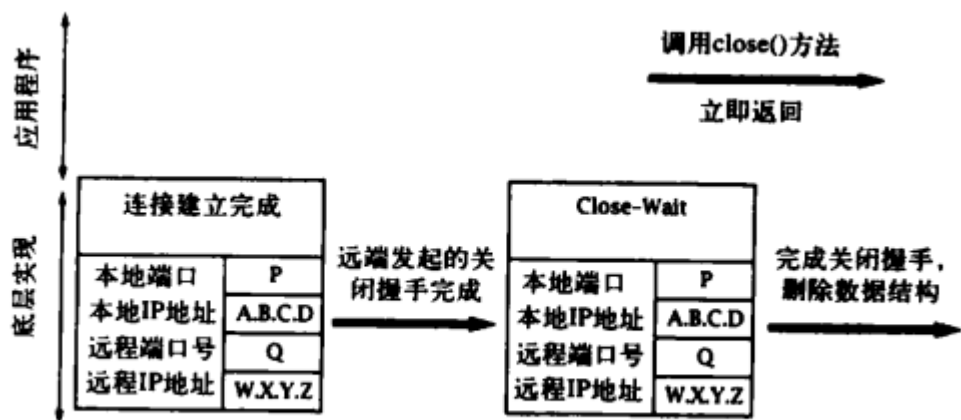


图6-11 在另一端关闭后关闭

注意这样一个事实：close()方法和 shutdownOutput()方法都没有等待关闭握手的完成，而是调用后立即返回，这样，当应用程序调用 close()方法或 shutdownOutput()方法并成功关闭连接时，有可能还有数据留在 SendQ 队列中。如果连接的任何一端在数据传输到 RecvQ 队列之前崩溃，数据将丢失，而发送端应用程序却不知道。

最好的解决方案是设计一种应用程序协议，以使首先调用 `close()` 方法的一方在接收到了应用程序的数据已接收保证后，才真正执行关闭操作。例如，在《[Socket 通信中由 read 返回值造成的死锁问题](#)》() 这篇文章的分析示例中，客户端程序确认其接收到的字节数与其发送的字节数相等后，它能够知道此时在连接的两个方向上都没有数据在传输，因此可以安全地关闭连接。

关闭 TCP 连接的最后微妙之处在于对 Time—Wait 状态的需要。TCP 规范要求终止连接时，两端的关闭握手都完成后，至少要有套接字在 Time—Wait 状态保持一段时间。这个要求的提出是由于消息在网络中传输时可能延迟。如果在连接两端都完成了关闭握手后，它们都移除了其底层数据结构，而此时在同样一对套接字地址之间又建立了新的连接，那么前一个连接在网络上传输时延迟的消息就可能在新建立连接后到达。由于包含了相同的源地址和目的地址，旧消息就会被错误地认为是属于新连接的，其包含的数据就可能被错误地分配到应用程序中。虽然这种情况很少发生，TCP 还是使用了包括 Time—Write 状态在内的多种机制对其进行防范。

Time—Wait 状态最重要的作用是：只要底层套接字数据结构还存在，就不允许在相同的本地端口上关联其他套接字，尤其试图使用该端口创建新的 Socket 实例时，将抛出 `IOException` 异常。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/java-socket/>