Java 网络 socket 编程详解

7.2 面向套接字编程
我们已经通过了解 Socket 的接口，知其所以然，下面我们就将通过具体的案例，来熟悉 Socket 的具体工作方式

7.2.1 使用套接字实现基于 TCP 协议的服务器和客户机程序
依据 TCP 协议，在 C/S 架构的通讯过程中，客户端和服务器的 Socket 动作如下：

客户端：

1.用服务器的 IP 地址和端口号实例化 Socket 对象。

2.调用 connect 方法，连接到服务器上。

3.将发送到服务器的 IO 流填充到 IO 对象里，比如 BufferedReader/PrintWriter。

4.利用 Socket 提供的 getInputStream 和 getOutputStream 方法，通过 IO 流对象，向服务器发送数据流。

5. 通讯完成后，关闭打开的 IO 对象和 Socket。

服务器：

1. 在服务器，用一个端口来实例化一个 ServerSocket 对象。此时，服务器就可以这个端口时刻监听从客户端发来的连接请求。

2.调用 ServerSocket 的 accept 方法，开始监听连接从端口上发来的连接请求。

3.利用 accept 方法返回的客户端的 Socket 对象，进行读写 IO 的操作

通讯完成后，关闭打开的流和 Socket 对象。

7.2.1.1 开发客户端代码
根据上面描述的通讯流程，我们可以按如下的步骤设计服务器端的代码。

第一步，依次点击 Eclipse 环境里的“文件”|“新建”|“项目”选项，进入“新建项目”的向导对话框，在其中选中“Java 项目”，点击“下一步”按钮，在随后弹出的对话框里，在其中的“项目名”一栏里，输入项目名“TCPSocket”，其它的选项目

选择系统默认值，再按“完成”按钮，结束创建 Java 项目的动作。

第二步，完成创建项目后，选中集成开发环境左侧的项目名“TCPSocket”，点击右键，在随后弹出的菜单里依次选择“新建”！“类”的选项，创建服务器类的代码。

在随后弹出的“新建 Java 类”的对话框里，输入包名“tcp”，输入文件名“ServerCode”，请注意大小写，在“修饰符”里选中“公用”，在“想要创建哪些方法存根”下，选中“public static void main(String[] args)”单选框，同时把其它两项目取消掉，再按“完成”按钮，可以生成代码。

第三步，在生成的代码里，编写引入 Java 包的代码，只有当我们引入这些包后，我们才能调用这些包里提供的 IO 和 Socket 类的方法。

```
package tcp;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.io.PrintWriter;

import java.net.ServerSocket;
```

```
import java.net.Socket;
```

第四步，编写服务器端的主体代码，如下所示。

```
   public class ServerCode

{
// 设置端口号
public static int portNo = 3333;
public static void main(String[] args) throws IOException
{
          ServerSocket s = new ServerSocket(portNo);

                    System.out.println("The Server is start: " + s);

                    // 阻塞,直到有客户端连接

            Socket socket = s.accept();
                        try

{

                              System.out.println("Accept the Client: " + socket);

                              //设置 IO 句柄
BufferedReader in = new BufferedReader(new InputStreamReader(socket

                                                  .getInputStream()));

  PrintWriter out = new PrintWriter(new BufferedWriter(

                             new OutputStreamWriter(socket.getOutputStream())), true);

                              while (true)

{

                                        String str = in.readLine();

                                        if (str.equals("byebye"))

                         {
```

```
                    break;
                }
                System.out.println("In Server reveived the info: " + str);
                out.println(str);
            }
        }
    finally
{
            System.out.println("close the Server socket and the io.");
            socket.close();
            s.close();
        }
    }
}
```

这段代码的主要业务逻辑是：

1. 在上述代码里的 main 函数前，我们设置了通讯所用到的端口号，为 3333。

2. 在 main 函数里，根据给定 3333 端口号，初始化一个 ServerSocket 对象 s，该对象用来承担服务器端监听连接和提供通讯服务的功能。

3. 调用 ServerSocket 对象的 accept 方法，监听从客户端的连接请求。当完成调用 accept 方法后，整段服务器端代码将回阻塞在这里，直到客户端发来 connect 请求。

4. 当客户端发来 connect 请求，或是通过构造函数直接把客户端的 Socket 对象连接到服务器端后，阻塞于此的代码将会继续运行。此时服务器端将会根据 accept 方法的执行结果，用一个 Socket 对象来描述客户端的连接句柄。

5. 创建两个名为 in 和 out 的对象，用来传输和接收通讯时的数据流。

6. 创建一个 while(true)的死循环，在这个循环里，通过 in.readLine()方法，读取从客户端发送来的 IO 流（字符串），并打印出来。如果读到的字符串是“byebye”，那么退出 while 循环。

7. 在 try…catch…finally 语句段里，不论在 try 语句段里是否发生异常，并且不论这些异常的种类，finally 从句都将会被执行到。在 finally 从句里，将关闭描述客户端的连接句柄 socket 对象和 ServerSocket 类型的 s 对象。

7.2.1.2 开发客户端代码

我们可以按以下的步骤，开发客户端的代码。

第一，在 TCPSocket 项目下的 tcp 包下，创建一个名为 ClientCode.java 的文件。在其中编写引入 Java 包的代码，如下所示：

```
    package tcp;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.io.PrintWriter;

import java.net.InetAddress;

import java.net.Socket;
```

第二，编写客户端的主体代码，如下所示：

```
 public class ClientCode

 {

        static String clientName = "Mike";

        //端口号

public static int portNo = 3333;
```

```
        public static void main(String[] args) throws IOException
{
                // 设置连接地址类,连接本地
                InetAddress addr = InetAddress.getByName("localhost");
                //要对应服务器端的 3333 端口号
                Socket socket = new Socket(addr, portNo);
                try
{
              System.out.println("socket = " + socket);
                        // 设置 IO 句柄
 BufferedReader in = new BufferedReader(new InputStreamReader(socket
                                        .getInputStream()));
PrintWrite out = new PrintWriter(BufferedWriter(new
OutputStreamWriter(socket.getOutputStream())), true);
                        out.println("Hello Server,I am " + clientName);
                        String str = in.readLine();
                        System.out.println(str);
                        out.println("byebye");
                }
finally
{
                        System.out.println("close the Client socket and the io.");
                        socket.close();
        }
```

```
        }
}
```

上述客户端代码的主要业务逻辑是：

1. 同样定义了通讯端口号，这里给出的端口号必须要和服务器端的一致。

2. 在 main 函数里，根据地址信息“localhost”，创建一个 InetAddress 类型的对象 addr。这里，因为我们把客户端和服务器端的代码都放在本机运行，所以同样可以用“127.0.0.1”字符串，来创建 InetAddress 对象。

3. 根据 addr 和端口号信息，创建一个 Socket 类型对象，该对象用来同服务器端的 ServerSocket 类型对象交互，共同完成 C/S 通讯流程。

4. 同样地创建 in 和 out 两类 IO 句柄，用来向服务器端发送和接收数据流。

5. 通过 out 对象，向服务器端发送"Hello Server,I am …"的字符串。发送后，同样可以用 in 句柄，接收从服务器端的消息。

6. 利用 out 对象，发送”byebye”字符串，用以告之服务器端，本次通讯结束。

7. 在 finally 从句里，关闭 Socket 对象，断开同服务器端的连接。

7.2.1.3 运行效果演示

在上述两部分里，我们分别讲述了 C/S 通讯过程中服务器端和客户端代码的业务逻辑，下面我们将在集成开发环境里，演示这里通讯流程。

第一步，选中 ServerCode.java 代码，在 eclipse 的“运行”菜单里，选中“运行方式”|“1 Java 应用程序”的菜单，开启服务器端的程序。

开启服务端程序后，会在 eclipse 环境下方的控制台里显示如下的内容：

```
The Server is start: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=3333]
```

在这里，由于 ServerSocket 对象并没监听到客户端的请求，所以 addr 和后面的 port 值都是初始值。

第二步，按同样的方法，打开 ClientCode.java 程序，启动客户端。启动以后，将在客户端的控制台里看到如下的信息：

```
socket = Socket[addr=localhost/127.0.0.1,port=3333,localport=1326]
```

Hello Server,I am Mike

close the Client socket and the io.

从中可以看到，在第一行里，显示客户端 Socket 对象连接的 IP 地址和端口号，在第二行里，可以到到客户端向服务器端发送的字符串，而在第三行里，可以看到通讯结束后，客户端关闭连接 Socket 和 IO 对象的提示语句。

第三步，在 eclipse 下方的控制台里，切换到 ServerCode 服务端的控制台提示信息里，我们可以看到服务器端在接收到客户端连接请求后的响应信息。

响应的信息如下所示：

The Server is start: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=3333]

Accept the Client: Socket[addr=/127.0.0.1,port=1327,localport=3333]

In Server reveived the info: Hello Server,I am Mike

close the Server socket and the io.

其中，第一行是启动服务器程序后显示的信息。在第二行里，显示从客户端发送的连接请求的各项参数。在第三行里，显示了从客户端发送过来的字符串。在第四行里，显示了关闭服务器端 ServerSocket 和 IO 对象的提示信息。从中我们可以看出在服务器端里 accept 阻塞和继续运行的这个过程。

通过上述的操作，我们可以详细地观察到 C/S 通讯的全部流程，请大家务必要注意：一定要先开启服务器端的程序再开启客户端，如果这个步骤做反的话，客户端程序会应找不到服务器端而报异常。

7.2.2 使用套接字连接多个客户机

在 7.1 的代码里，客户端和服务器之间只有一个通讯线程，所以它们之间只有一条 Socket 信道。

如果我们在通过程序里引入多线程的机制，可让一个服务器端同时监听并接收多个客户端的请求，并同步地为它们提供通讯服务。

基于多线程的通讯方式，将大大地提高服务器端的利用效率，并能使服务器端能具备完善的服务功能。

7.2.2.1 开发客户端代码

我们可以按以下的步骤开发基于多线程的服务器端的代码。

第一步，在 3.2 里创建的“TCPSocket”项目里，新建一个名为 ThreadServer.java 的代码文

件，创建文件的方式大家可以参照 3.2 部分的描述。首先编写 package 和 import 部分的代码，用来打包和引入包文件，如下所示：

```
package tcp;

import java.io.*;

import java.net.*;
```

第二步，由于我们在服务器端引入线程机制，所以我们要编写线程代码的主体执行类 ServerThreadCode，这个类的代码如下所示：

```
class ServerThreadCode extends Thread

{

        //客户端的 socket

        private Socket clientSocket;

        //IO 句柄

        private BufferedReader sin;

        private PrintWriter sout;

        //默认的构造函数

        public ServerThreadCode()

        {}

        public ServerThreadCode(Socket s) throws IOException

        {

                clientSocket = s;

                //初始化 sin 和 sout 的句柄

                sin = new BufferedReader(new InputStreamReader(clientSocket

                                .getInputStream()));
```

```
 sout = new PrintWriter(new BufferedWriter(new OutputStreamWriter(
                        clientSocket.getOutputStream())), true);
        //开启线程
        start();
}
//线程执行的主体函数
public void run()
{
        try
        {
                //用循环来监听通讯内容
                for(;;)
                {
           String str = sin.readLine();
                        //如果接收到的是 byebye，退出本次通讯
                        if (str.equals("byebye"))
                        {
                                break;
                        }
                        System.out.println("In Server reveived the info: " + str);
                        sout.println(str);
                }
```

```
                System.out.println("closing the server socket!");
            }
        catch (IOException e)
            {
                e.printStackTrace();
            }
            finally
            {
                System.out.println("close the Server socket and the io.");
                try
            {
                    clientSocket.close();
                }
                catch (IOException e)
                {
                    e.printStackTrace();
                }
            }
        }
}
```

这个类的业务逻辑说明如下：

1. 这个类通过继承 Thread 类来实现线程的功能，也就是说，在其中的 run 方法里，定义了该线程启动后要执行的业务动作。

2. 这个类提供了两种类型的重载函数。在参数类型为 Socket 的构造函数里， 通过参数，初始化了本类里的 Socket 对象，同时实例化了两类 IO 对象。在此基础上，通过 start 方法，启动定义在 run 方法内的本线程的业务逻辑。

3. 在定义线程主体动作的 run 方法里，通过一个 for(;;)类型的循环，根据 IO 句柄，读取从 Socket 信道上传输过来的客户端发送的通讯信息。如果得到的信息为“byebye”，则表明本次通讯结束，退出 for 循环。

4. catch 从句将处理在 try 语句里遇到的 IO 错误等异常，而在 finally 从句里，将在通讯结束后关闭客户端的 Socket 句柄。

上述的线程主体代码将会在 ThreadServer 类里被调用。

第三步，编写服务器端的主体类 ThreadServer，代码如下所示：

```
public class ThreadServer
{
        //端口号
        static final int portNo = 3333;
        public static void main(String[] args) throws IOException
        {
                //服务器端的 socket
                ServerSocket s = new ServerSocket(portNo);
                System.out.println("The Server is start: " + s);
                try
                {
                        for(;;)
                        {
                    //阻塞,直到有客户端连接
```

```
                Socket socket = s.accept();
                //通过构造函数，启动线程
                new ServerThreadCode(socket);
            }
        }
        finally
        {
            s.close();
        }
    }
}
```

这段代码的主要业务逻辑说明如下：

1. 首先定义了通讯所用的端口号，为 3333。

2. 在 main 函数里，根据端口号，创建一个 ServerSocket 类型的服务器端的 Socket，用来同客户端通讯。

3. 在 for(;;)的循环里，调用 accept 方法，监听从客户端请求过来的 socket，请注意这里又是一个阻塞。当客户端有请求过来时，将通过 ServerThreadCode 的构造函数，创建一个线程类，用来接收客户端发送来的字符串。在这里我们可以再一次观察 ServerThreadCode 类，在其中，这个类通过构造函数里的 start 方法，开启 run 方法，而在 run 方法里，是通过 sin 对象来接收字符串，通过 sout 对象来输出。

4. 在 finally 从句里，关闭服务器端的 Socket，从而结束本次通讯。

7.2.2.2 开发客户端代码
我们可以按以下的步骤，编写的基于多线程的客户端代码。

第一步，在 “TCPSocket”项目里，新建一个名为 ThreadClient.java 的代码文件。同样是编写 package 和 import 部分的代码，用来打包和引入包文件，如下所示：

```
package tcp;
```

```
import java.net.*;

import java.io.*;
```

第二步，编写线程执行主体的 ClientThreadCode 类，同样，这个类通过继承 Thread 来实现线程的功能。

```
class ClientThreadCode extends Thread

{

   //客户端的 socket

   private Socket socket;

   //线程统计数，用来给线程编号

   private static int cnt = 0;

   private int clientId = cnt++;

   private BufferedReader in;

   private PrintWriter out;

   //构造函数

   public ClientThreadCode(InetAddress addr)

   {

      try

      {

         socket = new Socket(addr, 3333);

      }

      catch(IOException e)

      {
```

```
            e.printStackTrace();
        }
        //实例化 IO 对象
try
        {
          in = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(
                      new     BufferedWriter(new     OutputStreamWriter(socket.getOutputStream())),
true);
            //开启线程
            start();
         }
        catch(IOException e)
        {
            //出现异常，关闭 socket
              try
              {
                 socket.close();
            }
              catch(IOException e2)
              {
                   e2.printStackTrace();
```

```
                }
            }
        }
    //线程主体方法
public void run()
    {
        try
        {
            out.println("Hello Server,My id is " + clientId );
            String str = in.readLine();
            System.out.println(str);
            out.println("byebye");
        }
        catch(IOException e)
        {
             e.printStackTrace();
        }
        finally
        {
            try
            {
                socket.close();
```

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

这个类的主要业务逻辑是：

1. 在构造函数里， 通过参数类型为 InetAddress 类型参数和 3333，初始化了本类里的 Socket 对象，随后实例化了两类 IO 对象，并通过 start 方法，启动定义在 run 方法内的本线程的业务逻辑。

2. 在定义线程主体动作的 run 方法里，通过 IO 句柄，向 Socket 信道上传输本客户端的 ID 号，发送完毕后，传输”byebye”字符串，向服务器端表示本线程的通讯结束。

3. 同样地，catch 从句将处理在 try 语句里遇到的 IO 错误等异常，而在 finally 从句里，将在通讯结束后关闭客户端的 Socket 句柄。

第三步，编写客户端的主体代码，在这段代码里，将通过 for 循环，根据指定的待创建的线程数量，通过 ClientThreadCode 的构造函数，创建若干个客户端线程，同步地和服务器端通讯。

```
public class ThreadClient
{
  public static void main(String[] args)
        throws IOException, InterruptedException
  {
    int threadNo = 0;
```

```
            InetAddress addr =

            InetAddress.getByName("localhost");

        for(threadNo = 0;threadNo<3;threadNo++)

        {

            new ClientThreadCode(addr);

        }

    }

}
```

这段代码执行以后，在客户端将会有 3 个通讯线程，每个线程首先将先向服务器端发送"Hello Server,My id is "的字符串，然后发送”byebye”，终止该线程的通讯。

7.2.2.3 运行效果演示

接下来，我们来观察一下基于多线程的 C/S 架构的运行效果。

第一步，我们先要启动服务器端的 ThreadServer 代码，启动后，在控制台里会出现如下的提示信息：

```
The Server is start: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=3333]
```

上述的提示信息里，我们同样可以看到，服务器在开启服务后，会阻塞在 accept 这里，直到有客户端请求过来。

第二步，我们在启动完服务器后，运行客户端的 ThreadClient.java 代码，运行后，我们观察服务器端的控制台，会出现如下的信息：

```
The Server is start: ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=3333]

In Server reveived the info: Hello Server,My id is 0

In Server reveived the info: Hello Server,My id is 1

In Server reveived the info: Hello Server,My id is 2

closing the server socket!
```

close the Server socket and the io.

closing the server socket!

close the Server socket and the io.

closing the server socket!

close the Server socket and the io.

其中，第一行是原来就有，在后面的几行里，首先将会输出了从客户端过来的线程请求信息，比如

In Server reveived the info: Hello Server,My id is 0

接下来则会显示关闭 Server 端的 IO 和 Socket 的提示信息。

这里，请大家注意，由于线程运行的不确定性，从第二行开始的打印输出语句的次序是不确定的。但是，不论输出语句的次序如何变化，我们都可以从中看到，客户端有三个线程请求过来，并且，服务器端在处理完请求后，会关闭 Socker 和 IO。

第三步，当我们运行完 ThreadClient.java 的代码后，并切换到 ThreadClient.java 的控制台，我们可以看到如下的输出：

Hello Server,My id is 0

Hello Server,My id is 2

Hello Server,My id is 1

这说明在客户端开启了 3 个线程，并利用这 3 个线程，向服务器端发送字符串。

而在服务器端，用 accept 方法分别监听到了这 3 个线程，并与之对应地也开了 3 个线程与之通讯。

7.2.3 UDP 协议与传输数据报文

UDP 协议一般应用在 “群发信息”的场合，所以它更可以利用多线程的机制，实现多信息的同步发送。

为了改善代码的架构，我们更可以把一些业务逻辑的动作抽象成方法，并封装成类，这样，基于 UDP 功能的类就可以在其它应用项目里被轻易地重用。

7.2.3.1 开发客户端代码

如果我们把客户端的所有代码都写在一个文件中，那么代码的功能很有可能都聚集在一个方

法力，代码的可维护性将会变得很差。

所以我们专门设计了 ClientBean 类，在其中封装了客户端通讯的一些功能方法，在此基础上，通过 UDPClient.java 文件，实现 UDP 客户端的功能。

另外，在这里以及以后的代码里，我们不再详细讲述用 Eclipse 开发和运行 Java 程序的方法，而是重点讲述 Java 代码的业务逻辑和主要工作流程。

首先，我们可以按如下的步骤，设计 ClientBean 这个类。通过 import 语句，引入所用到的类库，代码如下所示。

```
import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;
```

第二，定义 ClientBean 所用到的变量，并给出针对这些变量操作的 get 和 set 类型的方法，代码如下所示。

```
//描述 UDP 通讯的 DatagramSocket 对象

private    DatagramSocket ds;

//用来封装通讯字符串

private    byte buffer[];

//客户端的端口号

private    int clientport ;

//服务器端的端口号

private    int serverport;

//通讯内容
```

```
private    String content;
//描述通讯地址
private    InetAddress ia;
//以下是各属性的 Get 和 Set 类型方法
public byte[] getBuffer()
{
        return buffer;
}
public void setBuffer(byte[] buffer)
{
        this.buffer = buffer;
}
public int getClientport()
{
return clientport;
}
public void setClientport(int clientport)
{
        this.clientport = clientport;
}
public String getContent()
{
```

```
        return content;
}
public void setContent(String content)
{
        this.content = content;
}
public DatagramSocket getDs()
{
        return ds;
}
public void setDs(DatagramSocket ds)
{
        this.ds = ds;
}
public InetAddress getIa()
{
        return ia;
}
public void setIa(InetAddress ia)
{
        this.ia = ia;
}
```

```
public int getServerport()

{

        return serverport;

}

public void setServerport(int serverport)

{

    this.serverport = serverport;

}
```

在上述的代码里，我们定义了描述用来实现 UDP 通讯的 DatagramSocket 类型对象 ds，描述客户端和服务器端的端口号 clientport 和 serverport，用于描述通讯信息的 buffer 和 content 对象，其中，buffer 对象是 byte 数组类型的，可通过 UDP 的数据报文传输，而 content 是 String 类型的，在应用层面表示用户之间的通讯内容，另外还定义了 InetAddress 类型的 ia 变量，用来封装通讯地址信息。

在随后定义的一系列 get 和 set 方法里，给出了设置和获取上述变量的方法。

第三，编写该类的构造函数，代码如下所示。

```
public ClientBean() throws SocketException, UnknownHostException

{

        buffer = new byte[1024];

        clientport = 1985;

        serverport = 1986;

        content = "";

        ds = new DatagramSocket(clientport);

        ia = InetAddress.getByName("localhost");

}
```

在这个构造函数里，我们给各变量赋予了初始值，其中分别设置了客户端和服务器端的端口号分别为 1985 和 1986，设置了通讯连接地址为本地，并根据客户端的端口号初始化了 DatagramSocket 对象。

当程序员初始化 ClientBean 类时，这段构造函数会自动执行，完成设置通讯各参数等工作。

第四，编写向服务器端发送消息的 sendToServer 方法，代码如下所示。

```
public void sendToServer() throws IOException

{

        buffer = content.getBytes();

        ds.send(new DatagramPacket(buffer,content.length(),ia,serverport));

}
```

在这段代码里，根据 String 类型的表示通讯信息的 content 变量，初始化 UDP 数据报文，即 DatagramPacket 对象，并通过调用 DatagramSocket 类型对象的 send 方法，发送该 UDP 报文。

纵观 ClientBean 类，我们可以发现在其中封装了诸如通讯端口、通讯内容和通讯报文等对象以及以 UDP 方式发送信息的 sendToServer 方法。所以，在 UDPClient 类里，可以直接调用其中的接口，方便地实现通讯功能。

其次，我们可以按如下的步骤，设计 UDPClient 这个类。

第一步，通过 import 语句，引入所用到的类库，代码如下所示。

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;
```

第二步，编写线程相关的代码。

由于我们要在 UDP 客户端里通过多线程的机制，同时开多个客户端，向服务器端发送通讯内容，所以我们的 UDPClient 类必须要实现 Runnable 接口，并在其中覆盖掉 Runnable 接口里的 run 方法。定义类和实现 run 方法的代码如下所示。

```
public class UDPClient implements Runnable

{
```

```
public static String content;

public static ClientBean client;

public void run()

{

        try

{

                client.setContent(content);

                client.sendToServer();

        }

catch(Exception ex)

{

                System.err.println(ex.getMessage());

        }

}//end of run

//main  方法

   //…

 }
```

在上述代码的 run 方法里，我们主要通过了 ClientBean 类里封装的方法，设置了 content 内容，并通过了 sentToServer 方法，将 content 内容以数据报文的形式发送到服务器端。

一旦线程被开启，系统会自动执行定义在 run 方法里的动作。

第三步，编写主方法。在步骤(2)里的//main 方法注释的位置，我们可以插入 UDPClient 类的 main 方法代码，具体如下所示。

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

```
{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        client = new ClientBean();
        System.out.println("客户端启动...");
        while(true)
{
                //接收用户输入
content = br.readLine();
                //如果是 end 或空,退出循环
if(content==null||content.equalsIgnoreCase("end")||content.equalsIgnoreCase(""))
{
                        break;
                }
                //开启新线程，发送消息
new Thread(new UDPClient()).start();
        }
}
```

这段代码的主要业务逻辑是，首先初始化了 BufferedReader 类型的 br 对象，该对象可以接收从键盘输入的字符串。随后启动一个 while(true)的循环，在这个循环体里，接收用户从键盘的输入，如果用户输入的字符串不是“end”，或不是为空，则开启一个 UDPClient 类型的线程，并通过定义在 run 方法里的线程主体动作，发送接收到的消息。如果在循环体里，接收到“end”或空字符，则通过 break 语句，退出循环。

从上述代码里，我们可以看出，对于每次 UDP 发送请求，UDPClient 类都将会启动一个线程来发送消息。

7.2.3.2 开发客户端代码

同样，我们把服务器端所需要的一些通用方法以类的形式封装，而在 UDP 的服务器端，通过调用封装在 ServerBean 类里的方法来完成信息的接收工作。

首先，我们可以按如下的步骤，设计 ServerBean 类的代码。

第一步，通过 import 语句，引入所用到的类库，代码如下所示。

```
import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;
```

第二步，同样定义 ServerBean 类里用到的变量，并给出针对这些变量操作的 get 和 set 类型的方法。由于这里的代码和 ClientBean 类里的非常相似，所以不再赘述，代码部分大家可以参考光盘上。

第三步，编写该类的构造函数，在这个构造函数里，给该类里的一些重要属性赋了初值，代码如下所示。

```
public ServerBean() throws SocketException, UnknownHostException

{

        buffer = new byte[1024];

        clientport = 1985;

        serverport = 1986;

        content = "";

        ds = new DatagramSocket(serverport);

        ia = InetAddress.getByName("localhost");

}
```

从中我们可以看到，在 UDP 的服务端里，为了同客户端对应，所以同样把 clientport 和 serverport 值设置为 1985 和 1986，同时初始化了 DatagramSocket 对象，并把服务器的地址也设置成本地。

第四，编写实现监听客户端请求的 listenClient 方法，代码如下所示。

```
public void listenClient() throws IOException

{

        //在循环体里接收消息

while(true)

{

         //初始化 DatagramPacket 类型的变量

DatagramPacket dp = new DatagramPacket(buffer,buffer.length);

                //接收消息，并把消息通过 dp 参数返回

ds.receive(dp);

                content = new String(dp.getData(),0,dp.getLength());

                //打印消息

print();

        }

}
```

在这个方法里，构造了一个 while(true)的循环，在这个循环体内部，调用了封装在 DatagramSocket 类型里的 receive 方法，接收客户端发送过来的 UDP 报文，并通过 print 方法，把报文内容打印出来。

而 print 方法的代码比较简单，只是通过输出语句，打印报文里的字符串。

```
public void print()

{
```

```
        System.out.println(content);
}
```

而 UDP 通讯的服务器端代码相对简单，以下是 UDPServer 类的全部代码。

```
import java.io.IOException;
public class UDPServer
{
        public static void main(String args[]) throws IOException
{
                System.out.println("服务器端启动...");
                //初始化 ServerBean 对象
ServerBean server = new ServerBean();
                //开启监听程序
server.listenClient();
        }
}
```

从上述代码里，我们可以看到，在 UDP 的服务器端里，主要通过 ServerBean 类里提供的 listenClient 方法，监听从客户端发送过来的 UDP 报文，并通过解析得到其中包含的字符串，随后输出。

7.3.2.3 开发客户端代码

由于我们已经讲述过通过 Eclipse 查看代码运行结果的详细步骤，所以这里我们将直接通过命令行的方式，通过 javac 和 java 等命令，查看基于多线程 UDP 通讯的演示效果。

1. 首先我们把刚才编写好的四段 java 代码（即 ClientBean.java、UDPClient.java、ServerBean.java 和 UDPServer.java）放到 D 盘下的 work 目录下（如果没有则新建）。

2. 点击“开始菜单”|“运行”选项，并在“运行程序”的对话框里输入”cmd”命令，进入 DOS 命令界面，并进入到 D:\work 这个目录里。

3. 如果大家已经按照第一章的说明，成功地配置好关于 java 的 path 和 classpath 环境变量，在这里可以直接运行 javac *.java 命令，编译这四个.java 文件，编译后，会在 D:\work 目录下产生同四个 java 文件相对应的.class 文件。

4. 在这个命令窗口里运行 java UDPServer 命令，通过运行 UDPServer 代码，开启 UDP 服务器端程序，开启后，会出现如图 7-3 所示的信息。

图 7-3 启动 UDP 服务端后的效果

5. 在出现上图的效果后，别关闭这个命令窗口，按步骤(2)里说明的流程，新开启一个 DOS 命令窗口，并同样进入到 D:\work 这个目录下。

6. 在新窗口里输入 java UDPClient，开启 UDP 客户端程序。开启后，可通过键盘向服务器端输入通讯字符串，这些字符串将会以数据报文的形式发送到服务器端。

在图 7-4 里，演示了 UDP 客户端向服务器端发送消息的效果。

图 7-4 UDP 客户端发送消息的效果

每当我们在客户端发送一条消息，服务器端会收到并输出这条消息，从代码里我们可以得知，每条消息是通过为之新开启的线程发送到服务器端的。

如果我们在客户端输入”end”或空字符串，客户端的 UDPClient 代码会退出。在图 7-5 里演示了 UDP 服务器端接收并输出通讯字符串的效果。

图 7-5 UDP 服务器端接收到消息的效果

7. 由于 UDPServer.java 代码里，我们通过一个 while(true)的循环来监听客户端的请求，所以当程序运行结束后，可通过 Ctrl+C 的快捷键的方式退出这段程序。