* TCP通信

**1. Socket原理**

**1.1. Socket简介**

socket通常称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄。在Internet上的主机一般运行了多个服务软件，同时提供几种服务。每种服务都打开一个Socket，并绑定到一个端口上，不同的端口对应于不同的服务。

应用程序通常通过“套接字”向网络发出请求或者应答网络请求。Socket和ServerSocket类库位于java .net包中。ServerSocket用于服务端，Socket是建立网络连接时使用的。在连接成功时，应用程序两端都会产生一个Socket实例，操作这个实例，完成所需的会话。

**1.2. 获取本地地址和端口号**

java.net.Socket为套接字类，其提供了很多方法，其中我们可以通过Socket获取本地的地址以及端口号int getLocalPort()

该方法用于获取本地使用的端口号

InetAddress getLocalAddress()

该方法用于获取套接字绑定的本地地址

使用InetAddress获取本地的地址方法

String getCanonicalHostName()

获取此 IP 地址的完全限定域名。

String getHostAddress()

返回 IP 地址字符串（以文本表现形式）。

代码如下:

public void testSocket()throws Exception {

Socket socket = new Socket("localhost",8088);

InetAddress add = socket.getLocalAddress();

System.out.println(add.getCanonicalHostName());

System.out.println(add.getHostAddress());

System.out.println(socket.getLocalPort());

}

**1.3. 获取远端地址和端口号**

Socket也提供了获取远端的地址以及端口号的方法:

int getPort()

该方法用于获取远端使用的端口

InetAddress .getInetAddress()

该方法用于获取套接字绑定的远端地址 。

代码如下:

public void testSocket()throws Exception {

Socket socket = new Socket("localhost",8088);

InetAddress inetAdd = socket.getInetAddress();

System.out.println(inetAdd.getCanonicalHostName());

System.out.println(inetAdd.getHostAddress());

System.out.println(socket.getPort());

}

**1.4. 获取网络输入流和网络输出流**

通过Socket获取输入流与输出流,这两个方法是使用Socket通讯的关键方法。封装了TCP协议的Socket是基于流进行通讯的，所以我们在创建了双方连接后，只需要获取相应的输入与输出流即可实现通讯。

InputStream.getInputStream()

该方法用于返回此套接字的输入流。

OutputStream.getOutputStream()

该方法用于返回此套接字的输出流。

代码如下：

public void testSocket()throws Exception {

Socket socket = new Socket("localhost",8088);

InputStream in = socket.getInputStream();

OutputStream out = socket.getOutputStream();

}

**1.5. close方法**

当使用Socket进行通讯完毕后，要关闭Socket以释放系统资源。void close()

当关闭了该套接字后也会同时关闭由此获取的输入流与输出流。

**2. Socket通讯模型**

**2.1. Server端ServerSocket监听**

java.net.ServerSocket是运行于服务端应用程序中。通常创建ServerSocket需要指定服务端口号，之后监听Socket的连接。监听方法为:Socket accept()

该方法是一个阻塞方法，直到一个客户端通过Socket连接后，accept会封装一个Socket，该Socket封装与表示该客户端的有关的信息。通过这个Socket与该客户端进行通信。

代码如下:

…

//创建ServerSocket并申请服务端口8088

ServerSocket server = new ServerSocket(8088);

/\*方法会产生阻塞，直到某个Socket连接,并返回请求连接的Socket\*/

Socket socket = server.accept();

…

**2.2. Client端Socket连接**

通过上一节我们已经知道，当服务端ServerSocket调用accept方法阻塞等待客户端连接后，我们可以通过在客户端应用程序中创建Socket来向服务端发起连接。

需要注意的是，创建Socket的同时就发起连接，若连接异常会抛出异常。 我们通常创建Socket时会传入服务端的地址以及端口号。

代码如下:

//参数1：服务端的IP地址，参数2:服务端的服务端口

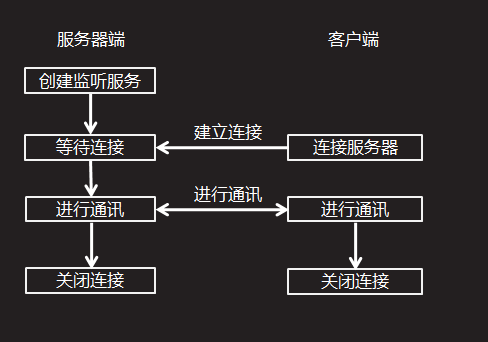
Socket socket = new Socket(“localhost”,8088);

…

**2.3. C-S端通信模型**

C-S的全称为(Client-Server):客户端-服务器端

* 客户端与服务端通信模型如下:
* 服务端创建ServerSocket
* 通过调用ServerSocket的accept方法监听客户端的连接
* 客户端创建Socket并指定服务端的地址以及端口来建立与服务端的连接
* 当服务端accept发现客户端连接后，获取对应该客户端的Socket
* 双方通过Socket分别获取对应的输入与输出流进行数据通讯
* 通讯结束后关闭连接。



* **Server端应用程序**

public class Server {

  public static void main(String[] args) {

    ServerSocket server = null;

      try {

        //创建ServerSocket并申请服务端口为8088

        server = new ServerSocket(8088);

        //侦听客户端的连接

        Socket socket = server.accept();

        //客户端连接后，通过该Socket与客户端交互

        //获取输入流，用于读取客户端发送过来的消息

        InputStream in = socket.getInputStream();

        BufferedReader reader = new BufferedReader( new InputStreamReader( in,"UTF-8" ));

        //获取输出流，用于向该客户端发送消息

        OutputStream out = socket.getOutputStream();

        PrintWriter writer = new PrintWriter(new OutputStreamWriter(out,"UTF-8"),true );

        //读取客户端发送的消息

        String message = reader.readLine();

        System.out.println("客户端说:"+message);

        //向客户端发送消息

        writer.println("你好客户端!");

      } catch (Exception e) {

        e.printStackTrace();

      } finally{

        if(server != null){

           try {

             server.close();

         } catch (IOException e) {

       }

     }

   }

 }

}

* **Client端应用程序**

public class Client {

    public static void main(String[] args) {

      Socket socket = null;

      try {

        socket = new Socket("localhost",8088);

        //获取输入流，用于读取来自服务端的消息

        InputStream in = socket.getInputStream();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in,"UTF-8"));

        //获取输出流，用于向服务端发送消息

        OutputStream out = socket.getOutputStream();

        OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");

        PrintWriter writer = new PrintWriter(osw,true);

        //向服务端发送一个字符串

        writer.println("你好服务器！");

        //读取来自客户端发送的消息

        String message = reader.readLine();

        System.out.println("服务器说:"+message);

      } catch (Exception e) {

        e.printStackTrace();

      } finally{

        try {

          if(socket != null){

          //关闭Socket

          socket.close();

          }

      } catch (IOException e) {

          e.printStackTrace();

       }

     }

   }

}

**3.Socket通讯模型**

**3.1. Server端多线程模型**

通过上一节我们已经知道了如何使用ServerSocket与Socket进行通讯了，但是这里存在着一个问题，就是只能“p2p”点对点。一个服务端对一个客户端。若我们想让一个服务端可以同时支持多个客户端应该怎么做呢？这时我们需要分析之前的代码。我们可以看到，当服务端的ServerSocket通过accept方法侦听到一个客户端Socket连接后，就获取该Socket并与该客户端通过流进行双方的通讯了，这里的问题在于，只有不断的调用accept方法，我们才能侦听到不同客户端的连接。但是若我们循环侦听客户端的连接，又无暇顾及与连接上的客户端交互，这时我们需要做的事情就是并发。我们可以创建一个线程类ClientHandler，并将于客户端交互的工作全部委托线程来处理。这样我们就可以在当一个客户端连接后，启动一个线程来负责与客户端交互，而我们也可以循环侦听客户端的连接了。

我们需要对服务端的代码进行修改:[copytextpop-up](http://pdf7.tarena.com.cn/tts8_source/ttsPage/JAVA/JSD_N_V06/JAVASE02/DAY06/SUPERDOC/01/index.html)

* **Server端应用程序**

public class Server {

  public static void main(String[] args) {

    ServerSocket server = null;

    try {

      //创建ServerSocket并申请服务端口为8088

      server = new ServerSocket(8088);

      while(true){

      //循环侦听客户端的连接

      Socket socket = server.accept();

      //当一个客户端连接后，启动线程来处理该客户端的交互

      new ClientHandler(socket).start();

    }

    } catch (Exception e) {

       e.printStackTrace();

    } finally{

       if(server != null){

          try {

             server.close();

          } catch (IOException e) {

       }

    }

  }

 }

}

class ClientHandler extends Thread{

\* 建立线程类

\* 该线程的作用是并发与客户端进行交互

\* 这里的代码就是原来在Server中客户端连接后交互的代码

  private Socket socket;

    public ClientHandler(Socket socket){

        this.socket = socket;

    }

    public void run(){

      try {

        //获取输入流，用于读取客户端发送过来的消息

        InputStream in = socket.getInputStream();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in,"UTF-8"));

        //获取输出流，用于向该客户端发送消息

        OutputStream out = socket.getOutputStream();

        PrintWriter writer = new PrintWriter(new OutputStreamWriter(out,"UTF-8"),true);

        //读取客户端发送的消息

        String message = reader.readLine();

        System.out.println("客户端说:"+message);

        //向客户端发送消息

        writer.println("你好客户端!");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}

Server端应用程序\*

public class Server {

public static void main(String[] args) {

ServerSocket server = null;

try {

//创建ServerSocket并申请服务端口为8088

server = new ServerSocket(8088);

while(true){

//循环侦听客户端的连接

Socket socket = server.accept();

//当一个客户端连接后，启动线程来处理该客户端的交互

new ClientHandler(socket).start();

}

} catch (Exception e) {

e.printStackTrace();

} finally{

if(server != null){

try {

server.close();

} catch (IOException e) {

}

}

}

}

}

/\*\*

\* 线程类

\* 该线程的作用是并发与客户端进行交互

\* 这里的代码就是原来在Server中客户端连接后交互的代码

\*/

class ClientHandler extends Thread{

private Socket socket;

public ClientHandler(Socket socket){

this.socket = socket;

}

public void run(){

try {

//获取输入流，用于读取客户端发送过来的消息

InputStream in = socket.getInputStream();

BufferedReader reader

= new BufferedReader(

new InputStreamReader(

in,"UTF-8"

)

);

//获取输出流，用于向该客户端发送消息

OutputStream out = socket.getOutputStream();

PrintWriter writer

= new PrintWriter(

new OutputStreamWriter(

out,"UTF-8"

),true

);

//读取客户端发送的消息

String message = reader.readLine();

System.out.println("客户端说:"+message);

//向客户端发送消息

writer.println("你好客户端!");

} catch (Exception e) {

e.printStackTrace();

}

}

}

经过上面的改动，我们再次启动服务端，这个时候我们会发现，我们启动若干客户端都可以被服务器所接受并进行交互了。