# 聊天室

## 基于socket的网络编程

董业 201400130044

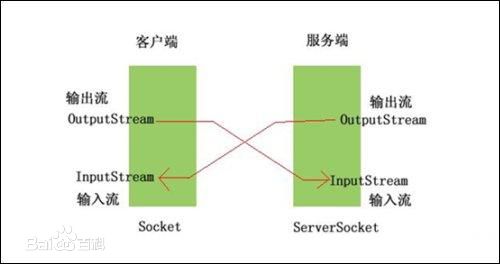
QQ:1342156335

### Socket简介：

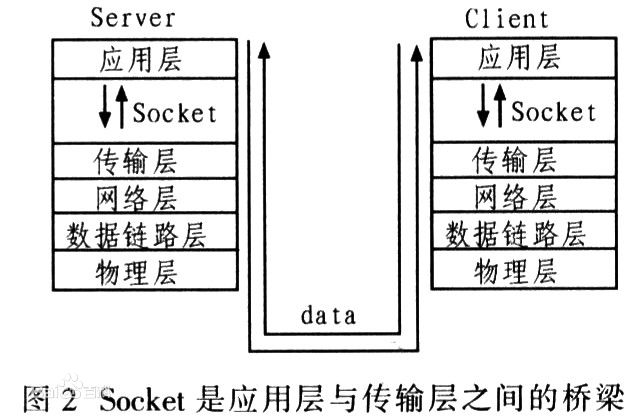
网络上两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个socket。

建立网络通信连接至少要一对端口号（socket）。Socket的本质事编程接口（API），对TCP/IP的封装，TCP/IP也要提供可供程序员做网络开发使用的接口，这就是socket编程接口。一个比较形象的比喻就是：HTTP是轿车，提供了封装或者显示数据的具体形式；socket是发动机，提供网络通信的能力。

Socket的英文原意是“孔”或者“插座”。作为BSD UNIX的进程机制，取后一种意思。通畅也称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄，可以用来实现不同虚拟机或者不同计算机之间的通信。再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，并绑定一个端口，不同的端口对应着不同的服务。Socket正像英文中所说的那样，像一个多孔插座。一台主机犹如布满各种插座的房间，每隔插座都有一个编号，有的插座提供220伏特的交流电，有的提供110伏特的交流电，有的则提供有线电视节目。而这些编号都是由操作系统分配的。客户端软件将插到不同编号的插座上，就可以的到不同的服务。



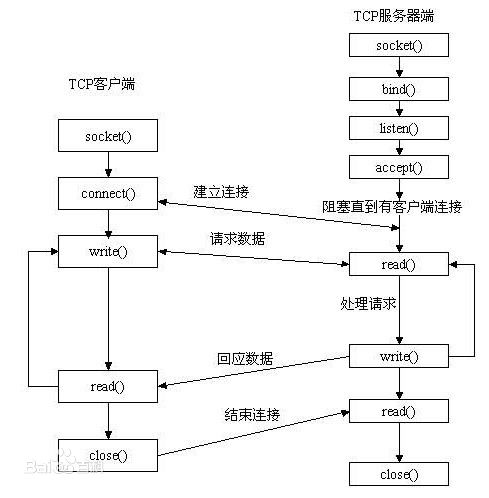
也就是说，套接字是应用层与传输层之间的桥梁，传输层通过套接字，将来自底层的数据信息以可见的方式传向上传递给应用层的应用软件，或者应用层通过使用套接字将信息传递给传输层，再由传输层封装信息传递给底层进行数据信息的传输。形象话展示如图：

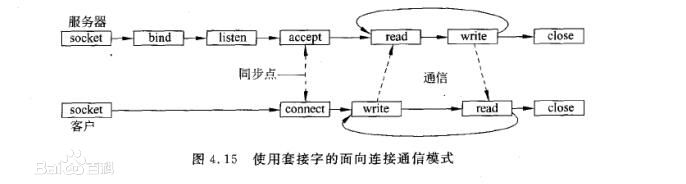


### 连接过程：

根据链接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求和连接确认。

1. 服务器监听：是服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态；
2. 客户端请求：是指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器的套接字的地址和端口号，然后就向服务器端套接字提出连接请求；
3. 连接确认：是指当服务器端套接字监听到或者说收到客户端套接字的连接请求，它就响应客户端的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其它客户端套接字的连接请求。





### 常用的函数：

1、创建

函数原型：

Int socket(int domain,int type,int protocol);

参数说明：

Domain：协议域，又称协议族。常用的协议族有AF\_INET、AF\_INET6、AF\_LOCAL（或称AF\_UNIX，UNIX域Socket）、AF\_ROUTE等。协议族决定了socket的地址类型，在通信重必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位）与端口号(16位)的组合、AF\_UNIX决定了要用一个绝对路径作为地址。

Type：指定socket类型。常用的socket类型有SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等。流式Socket（SOCK\_STREAM）是一种面向连接的Socket，针对于面向连接的TCP服务应用。数据报式Socket（SOCK\_DGRAM）是一种无连接的Socket，对应于无连接的UDP服务应用。

protocol：指定协议。常用协议有IPPROTO\_TCP、IPPROTO\_UDP、IPPROTO\_STCP、IPPROTO\_TIPC等，分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议。

注意：

（1）type和protocol不可以随意组合，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当第三个参数为0时，会自动选择第二个参数类型对应的默认协议。

（2）WindowsSocket下protocol参数中不存在IPPROTO\_STCP

返回值：

如果调用成功就返回新创建的套接字的描述符，如果失败就返回INVALID\_SOCKET（Linux下失败返回-1）。套接字描述符是一个整数类型的值。每个进程的进程空间里都有一个套接字描述符表，该表中存放着套接字描述符和套接字数据结构的对应关系。该表中有一个字段存放新创建的套接字的描述符，另一个字段存放套接字数据结构的地址，因此根据套接字描述符就可以找到其对应的套接字数据结构。每个进程在自己的进程空间里都有一个套接字描述符表但是套接字数据结构都是在操作系统的内核缓冲里。

2、绑定

函数原型：

int bind(SOCKET socket, const struct sockaddr\* address, socklen\_t address\_len);

参数说明：

socket：是一个套接字描述符。

address：是一个sockaddr结构指针，该结构中包含了要结合的地址和端口号。

address\_len：确定address缓冲区的长度。

返回值：

如果函数执行成功，返回值为0，否则为SOCKET\_ERROR。

3、接收

函数原型：

int recv(SOCKET socket, char FAR\* buf, int len, int flags);

参数说明：

socket：一个标识已连接套接口的描述字。

buf：用于接收数据的缓冲区。

len：缓冲区长度。

flags：指定调用方式。取值：MSG\_PEEK 查看当前数据，数据将被复制到缓冲区中，但并不从输入队列中删除；MSG\_OOB 处理带外数据。

返回值：

若无错误发生，recv()返回读入的字节数。如果连接已中止，返回0。否则的话，返回SOCKET\_ERROR错误，应用程序可通过WSAGetLastError()获取相应错误代码。

函数原型：

ssize\_t recvfrom(int sockfd, void buf, int len, unsigned int flags, struct socketaddr\* from, socket\_t\* fromlen);

参数说明：

sockfd：标识一个已连接套接口的描述字。

buf：接收数据缓冲区。

len：缓冲区长度。

flags：调用操作方式。是以下一个或者多个标志的组合体，可通过or操作连在一起：

（1）MSG\_DONTWAIT：操作不会被阻塞；

（2）MSG\_ERRQUEUE： 指示应该从套接字的错误队列上接收错误值，依据不同的协议，错误值以某种辅佐性消息的方式传递进来，使用者应该提供足够大的缓冲区。导致错误的原封包通过msg\_iovec作为一般的数据来传递。导致错误的数据报原目标地址作为msg\_name被提供。错误以sock\_extended\_err结构形态被使用。

（3）MSG\_PEEK：指示数据接收后，在接收队列中保留原数据，不将其删除，随后的读操作还可以接收相同的数据。

（4）MSG\_TRUNC：返回封包的实际长度，即使它比所提供的缓冲区更长， 只对packet套接字有效。

（5）MSG\_WAITALL：要求阻塞操作，直到请求得到完整的满足。然而，如果捕捉到信号，错误或者连接断开发生，或者下次被接收的数据类型不同，仍会返回少于请求量的数据。

（6）MSG\_EOR：指示记录的结束，返回的数据完成一个记录。

（7）MSG\_TRUNC：指明数据报尾部数据已被丢弃，因为它比所提供的缓冲区需要更多的空间。(MSG\_TRUNC使用错误,4才是MSG\_TRUNC的正确解释)

（8）MSG\_CTRUNC：指明由于缓冲区空间不足，一些控制数据已被丢弃。

（9）MSG\_OOB：指示接收到out-of-band数据(即需要优先处理的数据)。

（10）MSG\_ERRQUEUE：指示除了来自套接字错误队列的错误外，没有接收到其它数据。

from：（可选）指针，指向装有源地址的缓冲区。

fromlen：（可选）指针，指向from缓冲区长度值。

5、发送

函数原型：

int sendto( SOCKET s, const char FAR\* buf, int size, int flags, const struct sockaddr FAR\* to, int tolen);

参数说明：

s：套接字

buf：待发送数据的缓冲区

size：缓冲区长度

flags：调用方式标志位, 一般为0, 改变Flags，将会改变Sendto发送的形式

addr：（可选）指针，指向目的套接字的地址

tolen：addr所指地址的长度

返回值：

如果成功，则返回发送的字节数，失败则返回SOCKET\_ERROR。

接收连接请求

函数原型：

int accept( int fd, struct socketaddr\* addr, socklen\_t\* len);

参数说明：

fd：套接字描述符。

addr：返回连接着的地址

len：接收返回地址的缓冲区长度

返回值：

成功返回客户端的文件描述符，失败返回-1。

### 基于socket开发的聊天室网络程序：

客户端：  
连接服务器的关键代码如下：

**publicvoid** connectToServer() {

**try** {

s = **new** Socket("127.0.0.1", 8888);

dos = **new** DataOutputStream(s.getOutputStream());

dis = **new** DataInputStream(s.getInputStream());

bConnected = **true**;

} **catch** (BindException e) {

System.***out***.println("找不到指定的服务器");

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

因为是在本地测试，所以服务器的ip位127.0.0.1

如果连接服务器失败，就会输出响应的提示信息。

断开连接的关键代码如下：

**publicvoid** disConnect() {

**try** {

**if** (s != **null**) {

s.close();

}

**if** (dos != **null**) {

dos.close();

}

**if** (dis != **null**) {

dis.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

接下来几个内部类是对有关接口的实现：

**privateclass** Connect **implements** ActionListener {

**publicvoid** actionPerformed(ActionEvent e) {

**if** (e.getActionCommand() == "连接") {

connectToServer();

**try** {

t.start();

} **catch** (IllegalThreadStateException ex) {

}

connect.setText("断开连接");

} **elseif** (e.getActionCommand() == "断开连接") {

disConnect();

connect.setText("连接");

}

}

}

**privateclass** SendMsg **implements** ActionListener {

**publicvoid** actionPerformed(ActionEvent e) {

**if** (connect.getActionCommand() == "连接") {

JOptionPane.*showMessageDialog*(TCPClient.**this**,

"没有找到指定的服务器", "错误提示", 1);

} **else** {

String str = tfTxt.getText();

tfTxt.setText("");

**try** {

dos.writeUTF(str);

dos.flush();

} **catch** (SocketException ex) {

System.***out***.println("没有找到指定的服务器");

JOptionPane.*showMessageDialog*(TCPClient.**this**,

"没有找到指定的服务器", "错误提示", 1);

} **catch** (IOException ex) {

ex.printStackTrace();

}

}

}

}

**privateclass** RecToServer **implements** Runnable {

**publicvoid** run() {

**try** {

**while** (bConnected) {

String str = dis.readUTF();

// System.out.println(str);

taContent.append(str + "\n");

}

} **catch** (SocketException e) {

System.***out***.println("服务器已关闭");

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

上述就是客户端的关键部分，接下来介绍服务器端的关键部分：  
首先实现一个监视器，监听是否有客户端发来连接的请求：  
 **publicvoid** tcpMonitor() {

**try** {

ss = **new** ServerSocket(8888);

bStart = **true**;

} **catch** (IOException e) {

e.printStackTrace();

}

**try** {

**while** (bStart) {

index++;

Socket s = ss.accept();

Client c = **new** Client(s);

clients.add(c);

taContent.append(s.getInetAddress().getHostAddress()

+ " connected " + index + " clients\n");

**new** Thread(c).start();

}

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

ss.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

监听8888号端口，当一个客户端连接上服务器之后，index就会自增1，进而存储相关的信息。

实现一个内部类，用来在客户端和服务器之间交换信息：

**privateclass** Client **implements** Runnable {

DataInputStream dis = **null**;

DataOutputStream dos = **null**;

Socket s = **null**;

**boolean**bStart = **false**;

Client(Socket s) {

**this**.s = s;

**try** {

dis = **new** DataInputStream(s.getInputStream());

dos = **new** DataOutputStream(s.getOutputStream());

} **catch** (IOException e) {

e.printStackTrace();

}

bStart = **true**;

}

**publicvoid** sendToEveryClient(String str) {

**try** {

dos.writeUTF(str);

dos.flush();

} **catch** (IOException e) {

index--;

clients.remove(**this**);

taContent.append(s.getInetAddress().getHostAddress()

+ " exited " + index + " clients\n");

System.***out***.println("对方退出了！我从List里面去掉了！");

}

}

**publicvoid** run() {

**try** {

**while** (bStart) {

String str = dis.readUTF();

System.***out***.println(str);

**for** (**int**i = 0; i<clients.size(); i++) {

Client c = clients.get(i);

c.sendToEveryClient(str);

}

}

} **catch** (EOFException e) {

clients.remove(**this**);

taContent.append(s.getInetAddress().getHostAddress()

+ " exited " + clients.size() + " clients\n");

System.***out***.println("client closed");

} **catch** (SocketException e) {

System.***out***.println("client closed");

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

**if** (s != **null**)

s.close();

**if** (dis != **null**)

dis.close();

**if** (dos != **null**)

dos.close();

} **catch** (IOException e) {

e.printStackTrace();

}

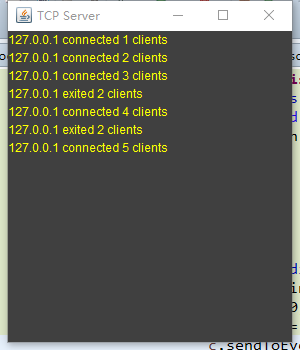
}

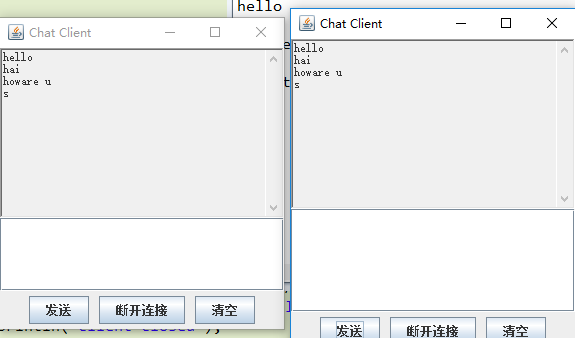
}

}

上述就是聊天室程序的关键代码部分，界面和其它部分在附件中。

程序执行的效果如下：

服务器端

客户端：

至此，整个程序设计完成

### 不足：

此次开发的聊天程序还有很多可以改善的地方：

从传统软件的角度出发，这个程序在界面等方面还有很大的差距。但是已经基本实现了socket的功能，达到了这次实验的目的。

除此之外，这个程序是否可以做成不基于服务器的，而是直接用户对用户的模式，如同P2P。

### 附件

：