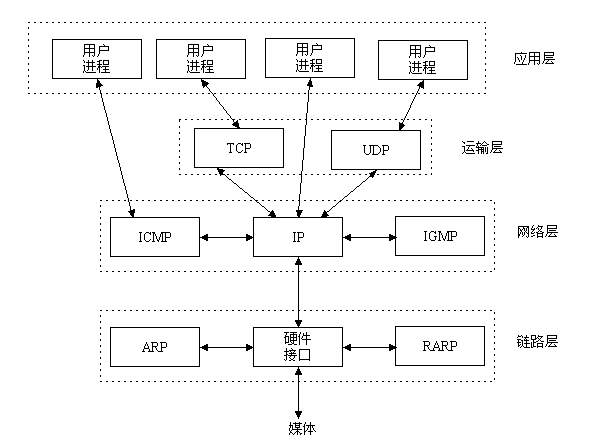
<https://blog.csdn.net/lxd8731247769/article/details/53589067>

对TCP/IP、UDP、Socket编程这些词你不会很陌生吧？随着网络技术的发展，这些词充斥着我们的耳朵。那么我想问：  
  
1.         什么是TCP/IP、UDP？  
2.         Socket在哪里呢？  
3.         Socket是什么呢？  
4.         你会使用它们吗？  
  
**什么是TCP/IP、UDP？**

         TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WANs）设计的。  
         UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议。它是属于TCP/IP协议族中的一种。  
        这里有一张图，表明了这些协议的关系。  
  
                                                                                  
  
                                                                        图1

       TCP/IP协议族包括运输层、网络层、链路层。现在你知道TCP/IP与UDP的关系了吧。  
**Socket在哪里呢？**在图1中，我们没有看到Socket的影子，那么它到底在哪里呢？还是用图来说话，一目了然。

  
图2

       原来Socket在这里。  
**Socket是什么呢？**       Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。  
**你会使用它们吗？**前人已经给我们做了好多的事了，网络间的通信也就简单了许多，但毕竟还是有挺多工作要做的。以前听到Socket编程，觉得它是比较高深的编程知识，但是只要弄清Socket编程的工作原理，神秘的面纱也就揭开了。  
       一个生活中的场景。你要打电话给一个朋友，先拨号，朋友听到电话铃声后提起电话，这时你和你的朋友就建立起了连接，就可以讲话了。等交流结束，挂断电话结束此次交谈。    生活中的场景就解释了这工作原理，也许TCP/IP协议族就是诞生于生活中，这也不一定。



图3

       先从服务器端说起。服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

============================================

我们深谙信息交流的价值，那网络中进程之间如何通信，如我们每天打开浏览器浏览网页 时，浏览器的进程怎么与web服务器通信的？当你用QQ聊天时，QQ进程怎么与服务器或你好友所在的QQ进程通信？这些都得靠socket？那什么是 socket？socket的类型有哪些？还有socket的基本函数，这些都是本文想介绍的。本文的主要内容如下：

1、网络中进程之间如何通信？

2、Socket是什么？

3、socket的基本操作

3.1、socket()函数

3.2、bind()函数

3.3、listen()、connect()函数

3.4、accept()函数

3.5、read()、write()函数等

3.6、close()函数

4、socket中TCP的三次握手建立连接详解

5、socket中TCP的四次握手释放连接详解

6、一个例子

# 1、网络中进程之间如何通信？

本地的进程间通信（IPC）有很多种方式，但可以总结为下面4类：

消息传递（管道、FIFO、消息队列）

同步（互斥量、条件变量、读写锁、文件和写记录锁、信号量）

共享内存（匿名的和具名的）

远程过程调用（Solaris门和Sun RPC）

但这些都不是本文的主题！我们要讨论的是网络中进程之间如何通信？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！在本地可以通过进程PID来唯一标识一个进程，但是在网络中这是行不通的。其实TCP/IP协议族已经帮我们解决了这个问题，网络层的“**ip地址**”可以唯一标识网络中的主机，而传输层的“**协议+端口**”可以唯一标识主机中的应用程序（进程）。这样利用三元组（ip地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互。

使用TCP/IP协议的应用程序通常采用应用编程接口：UNIX BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来实现网络进程之间的通信。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，网络中进程通信是无处不在，这就是我为什么说“一切皆socket”。

# 2、什么是Socket？

上面我们已经知道网络中的进程是通过socket来通信的，那什么是socket呢？socket起源于Unix，而Unix/Linux基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。我的理解就是Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭），这些函数我们在后面进行介绍。

### socket一词的起源

在组网领域的首次使用是在1970年2月12日发布的文献[IETF RFC33](http://datatracker.ietf.org/doc/rfc33/" \t "https://www.cnblogs.com/wangcq/p/_blank)中发现的，撰写者为Stephen Carr、Steve Crocker和Vint Cerf。根据美国计算机历史博物馆的记载，Croker写道：“命名空间的元素都可称为套接字接口。一个套接字接口构成一个连接的一端，而一个连接可完全由一对套接字接口规定。”计算机历史博物馆补充道：“这比BSD的套接字接口定义早了大约12年。”

# 3、socket的基本操作

既然socket是“open—write/read—close”模式的一种实现，那么socket就提供了这些操作对应的函数接口。下面以TCP为例，介绍几个基本的socket接口函数。

## 3.1、socket()函数

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

socket函数对应于普通文件的打开操作。普通文件的打开操作返回一个文件描述字，而**socket()**用于创建一个socket描述符（socket descriptor），它唯一标识一个socket。这个socket描述字跟文件描述字一样，后续的操作都有用到它，把它作为参数，通过它来进行一些读写操作。

正如可以给fopen的传入不同参数值，以打开不同的文件。创建socket的时候，也可以指定不同的参数创建不同的socket描述符，socket函数的三个参数分别为：

* domain：即协议域，又称为协议族（family）。常用的协议族有，AF\_INET、AF\_INET6、AF\_LOCAL（或称AF\_UNIX，Unix域socket）、AF\_ROUTE等等。协议族决定了socket的地址类型，在通信中必须采用对应的地址，如AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合、AF\_UNIX决定了要用一个绝对路径名作为地址。
* type：指定socket类型。常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等等（socket的类型有哪些？）。
* protocol：故名思意，就是指定协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等，它们分别对应TCP传输协议、UDP传输协议、STCP传输协议、TIPC传输协议（这个协议我将会单独开篇讨论！）。

注意：并不是上面的type和protocol可以随意组合的，如SOCK\_STREAM不可以跟IPPROTO\_UDP组合。当protocol为0时，会自动选择type类型对应的默认协议。

当我们调用**socket**创建一个socket时，返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址。如果想要给它赋值一个地址，就必须调用bind()函数，否则就当调用connect()、listen()时系统会自动随机分配一个端口。

## 3.2、bind()函数

正如上面所说bind()函数把一个地址族中的特定地址赋给socket。例如对应AF\_INET、AF\_INET6就是把一个ipv4或ipv6地址和端口号组合赋给socket。

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

函数的三个参数分别为：

* sockfd：即socket描述字，它是通过socket()函数创建了，唯一标识一个socket。bind()函数就是将给这个描述字绑定一个名字。
* addr：一个const struct sockaddr \*指针，指向要绑定给sockfd的协议地址。这个地址结构根据地址创建socket时的地址协议族的不同而不同，如ipv4对应的是：

struct sockaddr\_in {

sa\_family\_t sin\_family;

in\_port\_t sin\_port;

struct in\_addr sin\_addr;

};

struct in\_addr {

uint32\_t s\_addr;

};

* ipv6对应的是：

struct sockaddr\_in6 {

sa\_family\_t sin6\_family;

in\_port\_t sin6\_port;

uint32\_t sin6\_flowinfo;

struct in6\_addr sin6\_addr;

uint32\_t sin6\_scope\_id;

};

struct in6\_addr {

unsigned char s6\_addr[16];

};

* Unix域对应的是：

#define UNIX\_PATH\_MAX 108

struct sockaddr\_un {

sa\_family\_t sun\_family;

char sun\_path[UNIX\_PATH\_MAX];

};

* addrlen：对应的是地址的长度。

通常服务器在启动的时候都会绑定一个众所周知的地址（如ip地址+端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的ip地址组合。这就是为什么通常服务器端在listen之前会调用bind()，而客户端就不会调用，而是在connect()时由系统随机生成一个。

### 网络字节序与主机字节序

**主机字节序**就是我们平常说的大端和小端模式：不同的CPU有不同的字节序类型，这些字节序是指整数在内存中保存的顺序，这个叫做主机序。引用标准的Big-Endian和Little-Endian的定义如下：

　　a) Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

　　b) Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

**网络字节序**：4个字节的32 bit值以下面的次序传输：首先是0～7bit，其次8～15bit，然后16～23bit，最后是24~31bit。这种传输次序称作大端字节序。**由于TCP/IP首部中所有的二进制整数在网络中传输时都要求以这种次序，因此它又称作网络字节序。**字节序，顾名思义字节的顺序，就是大于一个字节类型的数据在内存中的存放顺序，一个字节的数据没有顺序的问题了。

所以： 在将一个地址绑定到socket的时候，请先将主机字节序转换成为网络字节序，而不要假定主机字节序跟网络字节序一样使用的是Big-Endian。由于 这个问题曾引发过血案！公司项目代码中由于存在这个问题，导致了很多莫名其妙的问题，所以请谨记对主机字节序不要做任何假定，务必将其转化为网络字节序再 赋给socket。

## 3.3、listen()、connect()函数

如果作为一个服务器，在调用socket()、bind()之后就会调用listen()来监听这个socket，如果客户端这时调用connect()发出连接请求，服务器端就会接收到这个请求。

int listen(int sockfd, int backlog);int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

listen函数的第一个参数即为要监听的socket描述字，第二个参数为相应socket可以排队的最大连接个数。socket()函数创建的socket默认是一个主动类型的，listen函数将socket变为被动类型的，等待客户的连接请求。

connect函数的第一个参数即为客户端的socket描述字，第二参数为服务器的socket地址，第三个参数为socket地址的长度。客户端通过调用connect函数来建立与TCP服务器的连接。

## 3.4、accept()函数

TCP服务器端依次调用socket()、bind()、listen()之后，就会监听指定的socket地址了。TCP客户端依次调用socket()、connect()之后就想TCP服务器发送了一个连接请求。TCP服务器监听到这个请求之后，就会调用accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络I/O操作了，即类同于普通文件的读写I/O操作。

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

accept函数的第一个参数为服务器的socket描述字，第二个参数为指向struct sockaddr \*的指针，用于返回客户端的协议地址，第三个参数为协议地址的长度。如果accpet成功，那么其返回值是由内核自动生成的一个全新的描述字，代表与返回客户的TCP连接。

注意：accept的第一个参数为服务器的socket描述字，是服务器开始调用socket()函数生成的，称为监听socket描述字；而accept函数返回的是已连接的socket描述字。一个服务器通常通常仅仅只创建一个监听socket描述字，它在该服务器的生命周期内一直存在。内核为每个由服务器进程接受的客户连接创建了一个已连接socket描述字，当服务器完成了对某个客户的服务，相应的已连接socket描述字就被关闭。

## 3.5、read()、write()等函数

万事具备只欠东风，至此服务器与客户已经建立好连接了。可以调用网络I/O进行读写操作了，即实现了网咯中不同进程之间的通信！网络I/O操作有下面几组：

* read()/write()
* recv()/send()
* readv()/writev()
* recvmsg()/sendmsg()
* recvfrom()/sendto()

我推荐使用recvmsg()/sendmsg()函数，这两个函数是最通用的I/O函数，实际上可以把上面的其它函数都替换成这两个函数。它们的声明如下：

#include

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, const void \*buf, size\_t count);

#include

#include

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,

const struct sockaddr \*dest\_addr, socklen\_t addrlen);

ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,

struct sockaddr \*src\_addr, socklen\_t \*addrlen);

ssize\_t sendmsg(int sockfd, const struct msghdr \*msg, int flags);

ssize\_t recvmsg(int sockfd, struct msghdr \*msg, int flags);

read函数是负责从fd中读取内容.当读成功时，read返回实际所读的字节数，如果返回的值是0表示已经读到文件的结束了，小于0表示出现了错误。如果错误为EINTR说明读是由中断引起的，如果是ECONNREST表示网络连接出了问题。

write函数将buf中的nbytes字节内容写入文件描述符fd.成功时返回写的字节 数。失败时返回-1，并设置errno变量。在网络程序中，当我们向套接字文件描述符写时有俩种可能。1)write的返回值大于0，表示写了部分或者是 全部的数据。2)返回的值小于0，此时出现了错误。我们要根据错误类型来处理。如果错误为EINTR表示在写的时候出现了中断错误。如果为EPIPE表示 网络连接出现了问题(对方已经关闭了连接)。

其它的我就不一一介绍这几对I/O函数了，具体参见man文档或者baidu、Google，下面的例子中将使用到send/recv。

## 3.6、close()函数

在服务器与客户端建立连接之后，会进行一些读写操作，完成了读写操作就要关闭相应的socket描述字，好比操作完打开的文件要调用fclose关闭打开的文件。

#include int close(int fd);

close一个TCP socket的缺省行为时把该socket标记为以关闭，然后立即返回到调用进程。该描述字不能再由调用进程使用，也就是说不能再作为read或write的第一个参数。

注意：close操作只是使相应socket描述字的引用计数-1，只有当引用计数为0的时候，才会触发TCP客户端向服务器发送终止连接请求。

# 4、socket中TCP的三次握手建立连接详解

我们知道tcp建立连接要进行“三次握手”，即交换三个分组。大致流程如下：

* 客户端向服务器发送一个SYN J
* 服务器向客户端响应一个SYN K，并对SYN J进行确认ACK J+1
* 客户端再想服务器发一个确认ACK K+1

只有就完了三次握手，但是这个三次握手发生在socket的那几个函数中呢？请看下图：

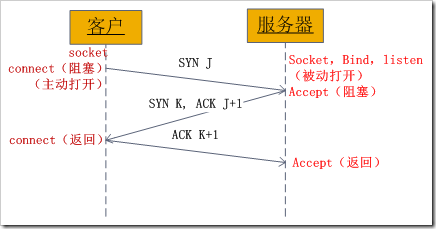
[](http://images.cnblogs.com/cnblogs_com/skynet/201012/201012122157467258.png)

图1、socket中发送的TCP三次握手

从图中可以看出，当客户端调用connect时，触发了连接请求，向服务器发送了SYN J包，这时connect进入阻塞状态；服务器监听到连接请求，即收到SYN J包，调用accept函数接收请求向客户端发送SYN K ，ACK J+1，这时accept进入阻塞状态；客户端收到服务器的SYN K ，ACK J+1之后，这时connect返回，并对SYN K进行确认；服务器收到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

总结：客户端的connect在三次握手的第二个次返回，而服务器端的accept在三次握手的第三次返回。

# 5、socket中TCP的四次握手释放连接详解

上面介绍了socket中TCP的三次握手建立过程，及其涉及的socket函数。现在我们介绍socket中的四次握手释放连接的过程，请看下图：

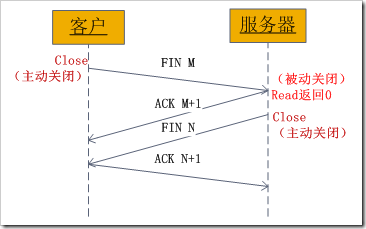
[](http://images.cnblogs.com/cnblogs_com/skynet/201012/201012122157487616.png)

图2、socket中发送的TCP四次握手

图示过程如下：

某个应用进程首先调用close主动关闭连接，这时TCP发送一个FIN M；

另一端接收到FIN M之后，执行被动关闭，对这个FIN进行确认。它的接收也作为文件结束符传递给应用进程，因为FIN的接收意味着应用进程在相应的连接上再也接收不到额外数据；

一段时间之后，接收到文件结束符的应用进程调用close关闭它的socket。这导致它的TCP也发送一个FIN N；

接收到这个FIN的源发送端TCP对它进行确认。

这样每个方向上都有一个FIN和ACK。

1. 下面给出实现的一个实例

服务器端：

                 ① 创建ServerSocket对象，绑定监听端口

                 ② 通过accept()方法监听客户端请求

                 ③ 连接建立后，通过输入流读取客户端发送的请求信息

                 ④ 通过输出流向客户端发送乡音信息

                 ⑤ 关闭相关资源

/\*\*

\* 基于TCP协议的Socket通信，实现用户登录，服务端

\*/

//1、创建一个服务器端Socket，即ServerSocket，指定绑定的端口，并监听此端口

ServerSocket serverSocket =new ServerSocket(10086);//1024-65535的某个端口

//2、调用accept()方法开始监听，等待客户端的连接

Socket socket = serverSocket.accept();

//3、获取输入流，并读取客户端信息

InputStream is = socket.getInputStream();

InputStreamReader isr =new InputStreamReader(is);

BufferedReader br =new BufferedReader(isr);

String info =null;

while((info=br.readLine())!=null){

System.out.println("我是服务器，客户端说："+info)；

}

socket.shutdownInput();//关闭输入流

//4、获取输出流，响应客户端的请求

OutputStream os = socket.getOutputStream();

PrintWriter pw = new PrintWriter(os);

pw.write("欢迎您！");

pw.flush();

//5、关闭资源

pw.close();

os.close();

br.close();

isr.close();

is.close();

socket.close();

serverSocket.close();

客户端：

                 ① 创建Socket对象，指明需要连接的服务器的地址和端口号

                 ② 连接建立后，通过输出流想服务器端发送请求信息

                 ③ 通过输入流获取服务器响应的信息

                 ④ 关闭响应资源

//客户端

//1、创建客户端Socket，指定服务器地址和端口

Socket socket =newSocket("localhost",10086);

//2、获取输出流，向服务器端发送信息

OutputStream os = socket.getOutputStream();//字节输出流

PrintWriter pw =newPrintWriter(os);//将输出流包装成打印流

pw.write("用户名：admin；密码：123");

pw.flush();

socket.shutdownOutput();

//3、获取输入流，并读取服务器端的响应信息

InputStream is = socket.getInputStream();

BufferedReader br = new BufferedReader(new InputStreamReader(is));

String info = null;

while((info=br.readLine())!null){

System.out.println("我是客户端，服务器说："+info);

}

//4、关闭资源

br.close();

is.close();

pw.close();

os.close();

socket.close();

应用多线程实现服务器与多客户端之间的通信

               ① 服务器端创建ServerSocket，循环调用accept()等待客户端连接

               ② 客户端创建一个socket并请求和服务器端连接

               ③ 服务器端接受苦读段请求，创建socket与该客户建立专线连接

               ④ 建立连接的两个socket在一个单独的线程上对话

               ⑤ 服务器端继续等待新的连接

//服务器线程处理

//和本线程相关的socket

Socket socket =null;

//

public serverThread(Socket socket){

this.socket = socket;

}

publicvoid run(){

//服务器处理代码

}

//============================================

//服务器代码

ServerSocket serverSocket =newServerSocket(10086);

Socket socket =null;

int count =0;//记录客户端的数量

while(true){

socket = serverScoket.accept();

ServerThread serverThread =new ServerThread(socket);

serverThread.start();

count++;

System.out.println("客户端连接的数量："+count);

}

具体代码：

**public** **class** SocketHandler **implements** Runnable {

Context context;

**public** SocketHandler(Context context) {

**this**.context = context;

}

@Override

**public** **void** run() {

Log.*d*("BLACK", "server go");

ServerSocket server = **null**;

**try** {

server = **new** ServerSocket(54321);

**while** (**true**) {

Log.*d*("BLACK", "open server");

Socket accept = server.accept();

**new** Thread(**new** MessageHandler(accept, context)).start();

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**try** {

server.close();

} **catch** (Exception e2) {

e2.printStackTrace();

}

}

}

}

**class** MessageHandler **implements** Runnable {

Socket socket = **null**;

Context context;

**public** MessageHandler(Socket socket, Context context) {

**this**.socket = socket;

**this**.context = context;

}

@Override

**public** **void** run() {

InputStream in = **null**;

OutputStream out = **null**;

**try** {

in = socket.getInputStream();

out = socket.getOutputStream();

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(

in, "UTF-8"));

String msg = **null**;

**while** ((msg = reader.readLine()) != **null**) {

// Thread.sleep(500);

Log.*d*("BLACK", "上报数据：" + msg);

String result = BussinessResource.*doBusiness*(msg,

context.getContentResolver(),context) + "\r\n";

Log.*d*("BLACK", "返回结果：" + result);

out.write(result.getBytes("UTF-8"));

}

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

**try** {

in.close();

} **catch** (Exception e2) {

e2.printStackTrace();

}

**try** {

out.close();

} **catch** (Exception e2) {

e2.printStackTrace();

}

}

}

}

**五、UDP编程**

**UDP协议（用户数据报协议）是无连接的、不可靠的、无序的,速度快**

**进行数据传输时，首先将要传输的数据定义成数据报（Datagram），大小限制在64k，在数据报中指明数据索要达到的Socket（主机地址和端口号），然后再将数据报发送出去**

**DatagramPacket类:表示数据报包**

**DatagramSocket类：进行端到端通信的类**

**1、服务器端实现步骤**

**① 创建DatagramSocket，指定端口号**

**② 创建DatagramPacket**

**③ 接受客户端发送的数据信息**

**④ 读取数据**

1. //服务器端，实现基于UDP的用户登录
2. //1、创建服务器端DatagramSocket，指定端口
3. DatagramSocket socket =new datagramSocket(10010);
4. //2、创建数据报，用于接受客户端发送的数据
5. byte[] data =newbyte[1024];//
6. DatagramPacket packet =newDatagramPacket(data,data.length);
7. //3、接受客户端发送的数据
8. socket.receive(packet);//此方法在接受数据报之前会一致阻塞
9. //4、读取数据
10. String info =newString(data,o,data.length);
11. System.out.println("我是服务器，客户端告诉我"+info)
12. //=========================================================
13. //向客户端响应数据
14. //1、定义客户端的地址、端口号、数据
15. InetAddress address = packet.getAddress();
16. int port = packet.getPort();
17. byte[] data2 = "欢迎您！".geyBytes();
18. //2、创建数据报，包含响应的数据信息
19. DatagramPacket packet2 = new DatagramPacket(data2,data2.length,address,port);
20. //3、响应客户端
21. socket.send(packet2);
22. //4、关闭资源
23. socket.close();

**2、客户端实现步骤**

**① 定义发送信息**

**② 创建DatagramPacket，包含将要发送的信息**

**③ 创建DatagramSocket**

**④ 发送数据**

1. //客户端
2. //1、定义服务器的地址、端口号、数据
3. InetAddress address =InetAddress.getByName("localhost");
4. int port =10010;
5. byte[] data ="用户名：admin;密码：123".getBytes();
6. //2、创建数据报，包含发送的数据信息
7. DatagramPacket packet = newDatagramPacket(data,data,length,address,port);
8. //3、创建DatagramSocket对象
9. DatagramSocket socket =newDatagramSocket();
10. //4、向服务器发送数据
11. socket.send(packet);
12. //接受服务器端响应数据
13. //======================================
14. //1、创建数据报，用于接受服务器端响应数据
15. byte[] data2 = new byte[1024];
16. DatagramPacket packet2 = new DatagramPacket(data2,data2.length);
17. //2、接受服务器响应的数据
18. socket.receive(packet2);
19. String raply = new String(data2,0,packet2.getLenth());
20. System.out.println("我是客户端，服务器说："+reply);
21. //4、关闭资源
22. socket.close();

**六、注意问题：**

**1、多线程的优先级问题：**

**根据实际的经验，适当的降低优先级，否侧可能会有程序运行效率低的情况**

**2、是否关闭输出流和输入流：**

**对于同一个socket，如果关闭了输出流，则与该输出流关联的socket也会被关闭，所以一般不用关闭流，直接关闭socket即可**

**3、使用TCP通信传输对象，IO中序列化部分**

**4、socket编程传递文件，IO流部分**