在客户/服务器通信模式中, 客户端需要主动创建与服务器连接的 Socket(套接字), 服务器端收到了客户端的连接请求, 也会创建与客户连接的 Socket. Socket可看做是通信连接两端的收发器, 服务器与客户端都通过 Socket 来收发数据.

这篇文章首先介绍Socket类的各个构造方法, 以及成员方法的用法, 接着介绍 Socket的一些选项的作用, 这些选项可控制客户建立与服务器的连接, 以及接收和发送数据的行为.

一. 构造Socket

     Socket的构造方法有以下几种重载形式:

Socket()   
Socket(InetAddress address, int port) throws UnknowHostException, IOException   
Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException   
Socket(String host, int port) throws UnknowHostException, IOException   
Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException   
除了第一个不带参数的构造方法以外, 其他构造方法都会试图建立与服务器的连接, 如果连接成功, 就返回 Socket对象; 如果因为某些原因连接失败, 就会抛出IOException .

1.1 使用无参数构造方法, 设定等待建立连接的超时时间

Socket socket = new Socket();  
SocketAddress remoteAddr = new InetSocketAddress("localhost",8000);  
socket.connect(remoteAddr, 60000); //等待建立连接的超时时间为1分钟

      以上代码用于连接到本地机器上的监听8000端口的服务器程序, 等待连接的最长时间为1分钟. 如果在1分钟内连接成功则connet()方法顺利返回; 如果在1分钟内出现某种异常, 则抛出该异常; 如果超过1分钟后, 即没有连接成功, 也没有出现其他异常, 那么会抛出 SocketTimeoutException. Socket 类的 connect(SocketAddress endpoint, int timeout) 方法负责连接服务器, 参数endpoint 指定服务器的地址, 参数timeout 设定超时数据, 以毫秒为单位. 如果参数timeout 设为0, 表示永远不会超时, 默认是不会超时的.

1.2 设定服务器的地址

      除了第一个不带参数的构造方法, 其他构造方法都需要在参数中设定服务器的地址, 包括服务器的IP地址或主机名, 以及端口:

Socket(InetAddress address, int port)              //第一个参数address 表示主机的IP地址  
Socket(String host, int port)                              //第一个参数host 表示主机的名字

      InetAddress 类表示服务器的IP地址, InetAddress 类提供了一系列静态工厂方法, 用于构造自身的实例, 例如:

//返回本地主机的IP地址  
InetAddress addr1 = InetAddress.getLocalHost();  
//返回代表 "222.34.5.7"的 IP地址  
InetAddress addr2 = InetAddress.getByName("222.34.5.7");  
//返回域名为"[www.javathinker.org](http://www.javathinker.org/)"的 IP地址  
InetAddress addr3 = InetAddress.getByName("[www.javathinker.org](http://www.javathinker.org/)");

1.3 设定客户端的地址

      在一个Socket 对象中, 即包含远程服务器的IP 地址和端口信息, 也包含本地客户端的IP 地址和端口信息. 默认情况下, 客户端的IP 地址来自于客户程序所在的主机, 客户端的端口则由操作系统随机分配. Socket类还有两个构造方法允许显式地设置客户端的IP 地址和端口:

//参数localAddr 和 localPort 用来设置客户端的IP 地址和端口  
Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException  
Socket(String host, int port, InetAddress localAddr, int localPort) throws IOException

     如果一个主机同时属于两个以上的网络, 它就可能拥有两个以上的IP 地址. 例如, 一个主机在Internet 网络中的IP 地址为 "222.67.1.34", 在一个局域网中的IP 地址为 "112.5.4.3". 假定这个主机上的客户程序希望和同一个局域网的一个服务器程序(地址为:"112.5.4.45: 8000")通信, 客户端可按照如下方式构造Socket 对象:

InetAddress remoteAddr1 = InetAddress.getByName("112.5.4.45");  
InetAddress localAddr1 = InetAddress.getByName("112.5.4.3");  
Socket socket1 = new Socket(remoteAddr1, 8000, localAddr1, 2345);   //客户端使用端口2345

1.4 客户连接服务器时可能抛出的异常

       当Socket 的构造方法请求连接服务器时, 可能会抛出下面的异常.

UnKnownHostException: 如果无法识别主机的名字或IP 地址, 就会抛出这种异常.   
ConnectException: 如果没有服务器进程监听指定的端口, 或者服务器进程拒绝连接, 就会抛出这种异常.   
SocketTimeoutException: 如果等待连接超时, 就会抛出这种异常.   
BindException: 如果无法把Socket 对象与指定的本地IP 地址或端口绑定, 就会抛出这种异常.  
以上4中异常都是IOException的直接或间接子类.      如图2-1所示.

        IOException------- UnknownHostException

                              |---- InterruptedIOException ----------- SocketTimeoutException

                              |---- SocketException              ----------- BindException

                                                                             |---------- ConnectException

                                    图2-1 客户端连接服务器时可能抛出的异常

二. 获取Socket 的信息

      在一个Socket 对象中同时包含了远程服务器的IP 地址和端口信息, 以及客户本地的IP 地址和端口信息. 此外, 从Socket 对象中还可以获得输出流和输入流, 分别用于向服务器发送数据, 以及接收从服务器端发来的数据. 以下方法用于获取Socket的有关信息.

getInetAddress(): 获得远程服务器的IP 地址.   
getPort(): 获得远程服务器的端口.   
getLocalAddress(): 获得客户本地的IP 地址.   
getLocalPort(): 获得客户本地的端口.   
getInputStream(): 获得输入流. 如果Socket 还没有连接, 或者已经关闭, 或者已经通过 shutdownInput() 方法关闭输入流, 那么此方法会抛出IOException.   
getOutputStream(): 获得输出流, 如果Socket 还没有连接, 或者已经关闭, 或者已经通过 shutdownOutput() 方法关闭输出流, 那么此方法会抛出IOException.   
这里有个HTTPClient 类的例子, 代码我是写好了, 也测试过了, 因为篇幅原因就不贴了. 这个HTTPClient 类用于访问网页 [www.javathinker.org/index.jsp](http://www.javathinker.org/index.jsp). 该网页位于一个主机名(也叫域名)为 [www.javathinker.org](http://www.javathinker.org/) 的远程HTTP服务器上, 它监听 80 端口. 在HTTPClient 类中, 先创建了一个连接到该HTTP服务器的Socket对象, 然后发送符合HTTP 协议的请求, 接着接收从HTTP 服务器上发回的响应结果.

三. 关闭Socket

     当客户与服务器的通信结束, 应该及时关闭Socket , 以释放Socket 占用的包括端口在内的各种资源. Socket 的 close() 方法负责关闭Socket. 当一个Socket对象被关闭, 就不能再通过它的输入流和输出流进行I/O操作, 否则会导致IOException.

      为了确保关闭Socket 的操作总是被执行, 强烈建议把这个操作放在finally 代码块中:

   Socket socket = null;  
try{  
socket = new Socket(www.javathinker.org,80);  
//执行接收和发送数据的操作  
..........  
}catch(IOException e){  
e.printStackTrace();  
}finally{  
try{  
if(socket != null) socket.close();  
}catch(IOException e){e.printStackTrace();}  
}

    Socket 类提供了3 个状态测试方法.

isClosed(): 如果Socket已经连接到远程主机, 并且还没有关闭, 则返回true , 否则返回false .   
isConnected(): 如果Socket曾经连接到远程主机, 则返回true , 否则返回false .   
isBound(): 如果Socket已经与一个本地端口绑定, 则返回true , 否则返回false .   
如果要判断一个Socket 对象当前是否处于连接状态, 可采用以下方式:

      boolean isConnected = socket.isConnected() && !socket.isClosed();

四. 半关闭Socket

     进程A 与进程B 通过Socket 通信, 假定进程A 输出数据, 进程B 读入数据. 进程A 如何告诉进程B 所有数据已经输出完毕? 下文略......

五. 设置Socket 的选项

     Socket 有以下几个选项.

TCP\_NODELAY: 表示立即发送数据.   
SO\_RESUSEADDR: 表示是否允许重用Socket 所绑定的本地地址.   
SO\_TIMEOUT: 表示接收数据时的等待超时数据.   
SO\_LINGER: 表示当执行Socket 的 close()方法时, 是否立即关闭底层的Socket.   
SO\_SNFBUF: 表示发送数据的缓冲区的大小.   
SO\_RCVBUF: 表示接收数据的缓冲区的大小.   
SO\_KEEPALIVE: 表示对于长时间处于空闲状态的Socket , 是否要自动把它关闭.   
OOBINLINE: 表示是否支持发送一个字节的TCP 紧急数据.   
5.1 TCP\_NODELAY 选项

设置该选项: public void setTcpNoDelay(boolean on) throws SocketException   
读取该选项: public boolean getTcpNoDelay() throws SocketException   
默认情况下, 发送数据采用Negale 算法. Negale 算法是指发送方发送的数据不会立即发出, 而是先放在缓冲区, 等缓存区满了再发出. 发送完一批数据后, 会等待接收方对这批数据的回应, 然后再发送下一批数据. Negale 算法适用于发送方需要发送大批量数据, 并且接收方会及时作出回应的场合, 这种算法通过减少传输数据的次数来提高通信效率.

     如果发送方持续地发送小批量的数据, 并且接收方不一定会立即发送响应数据, 那么Negale 算法会使发送方运行很慢. 对于GUI 程序, 如网络游戏程序(服务器需要实时跟踪客户端鼠标的移动), 这个问题尤其突出. 客户端鼠标位置改动的信息需要实时发送到服务器上, 由于Negale 算法采用缓冲, 大大减低了实时响应速度, 导致客户程序运行很慢.

      TCP\_NODELAY 的默认值为 false, 表示采用 Negale 算法. 如果调用setTcpNoDelay(true)方法, 就会关闭 Socket的缓冲, 确保数据及时发送:

       if(!socket.getTcpNoDelay()) socket.setTcpNoDelay(true);

      如果Socket 的底层实现不支持TCP\_NODELAY 选项, 那么getTcpNoDelay() 和 setTcpNoDelay 方法会抛出 SocketException.

5.2 SO\_RESUSEADDR 选项

设置该选项: public void setResuseAddress(boolean on) throws SocketException   
读取该选项: public boolean getResuseAddress() throws SocketException   
当接收方通过Socket 的close() 方法关闭Socket 时, 如果网络上还有发送到这个Socket 的数据, 那么底层的Socket 不会立即释放本地端口, 而是会等待一段时间, 确保接收到了网络上发送过来的延迟数据, 然后再释放端口. Socket接收到延迟数据后, 不会对这些数据作任何处理. Socket 接收延迟数据的目的是, 确保这些数据不会被其他碰巧绑定到同样端口的新进程接收到.

     客户程序一般采用随机端口, 因此出现两个客户程序绑定到同样端口的可能性不大. 许多服务器程序都使用固定的端口. 当服务器程序关闭后, 有可能它的端口还会被占用一段时间, 如果此时立刻在同一个主机上重启服务器程序, 由于端口已经被占用, 使得服务器程序无法绑定到该端口, 启动失败. (第三篇文章会对此作出介绍).

     为了确保一个进程关闭Socket 后, 即使它还没释放端口, 同一个主机上的其他进程还可以立即重用该端口, 可以调用Socket 的setResuseAddress(true) 方法:

        if(!socket.getResuseAddress()) socket.setResuseAddress(true);

    值得注意的是 socket.setResuseAddress(true) 方法必须在 Socket 还没有绑定到一个本地端口之前调用, 否则执行 socket.setResuseAddress(true) 方法无效. 因此必须按照以下方式创建Socket 对象, 然后再连接远程服务器:

Socket socket = new Socket();            //此时Socket对象未绑定本地端口,并且未连接远程服务器  
socket.setReuseAddress(true);  
SocketAddress remoteAddr = new InetSocketAddress("localhost",8000);  
socket.connect(remoteAddr);              //连接远程服务器, 并且绑定匿名的本地端口

    或者:

Socket socket = new Socket();              //此时Socke 对象为绑定本地端口, 并且未连接远程服务器  
socket.setReuseAddress(true);  
SocketAddress localAddr = new InetSocketAddress("localhost",9000);  
SocketAddress remoteAddr = new InetSocketAddress("localhost",8000);  
socket.bind(localAddr);             //与本地端口绑定  
socket.connect(remoteAddr); //连接远程服务器

    此外, 两个共用同一个端口的进程必须都调用 socket.setResuseAddress(true) 方法, 才能使得一个进程关闭 Socket后, 另一个进程的 Socket 能够立即重用相同端口.

5.3 SO\_TIMEOUT 选项

设置该选项: public void setSoTimeout(int milliseconds) throws SocketException   
读取该选项: public int getSoTimeout() throws SocketException   
当通过Socket 的输入流读数据时, 如果还没有数据, 就会等待. 例如, 在以下代码中, in.read(buff) 方法从输入流中读入 1024个字节:

byte[] buff = new byte[1024];  
InputStream in = socket.getInputStream();  
in.read(buff);

     如果输入流中没有数据, in.read(buff) 就会等待发送方发送数据, 直到满足以下情况才结束等待:

     略...............

     Socket 类的 SO\_TIMEOUT 选项用于设定接收数据的等待超时时间, 单位为毫秒, 它的默认值为 0, 表示会无限等待, 永远不会超时. 以下代码把接收数据的等待超时时间设为 3 分钟:

        if(socket.getSoTimeout() == 0) socket.setSoTimeout(60000 \* 3);   //注意, 原书中这里的代码错误, 里面的方法名字都少了"So"

     Socket 的 setSoTimeout() 方法必须在接收数据之前执行才有效. 此外, 当输入流的 read()方法抛出 SocketTimeoutException 后, Socket 仍然是连接的, 可以尝试再次读数据:

socket.setSoTimeout(180000);  
byte[] buff = new byte[1024];  
InputStream in = socket.getInputStream();  
int len = -1;  
do{  
try{  
len = in.read(buff);  
//处理读到的数据  
//.........  
}catch(SocketTimeoutException e){  
//e.printStackTrace();   
System.out.println("等待读超时!");  
len = 0;  
}      
}while(len != -1);

     例子ReceiveServer.java 和 SendClient.java 是一对简单的服务器/客户程序. sendClient 发送字符串 "hello everyone" ,接着睡眠 1 分钟, 然后关闭 Socket. ReceiveServer 读取 SendClient 发送来的数据, 直到抵达输入流的末尾, 最后打印 SendClient 发送来的数据.

     ReceiveServer.java 略....... ,         SendClient.java 略..........

     在 SendClient 发送字符串 "hello everyone" 后, 睡眠 1 分钟. 当 SendClient 在睡眠时, ReceiveServer 在执行 in.read(buff) 方法, 不能读到足够的数据填满 buff 缓冲区, 因此会一直等待 SendClient 发送数据. 如果在 ReceiveServer 类中 socket.setSoTimeout(20000) , 从而把等待接收数据的超时时间设为 20 秒, 那么 ReceiveServer 在等待数据时, 每当超过 20 秒, 就会抛出SocketTimeoutException . 等到 SendClient 睡眠 1 分钟后, SendClient 调用 Socket 的 close() 方法关闭 Socket, 这意味着 ReceiveServer 读到了输入流的末尾, ReceiveServer 立即结束读等待, read() 方法返回 -1 . ReceiveServer最后打印接收到的字符串 "hello everyone", 结果如下:

等待读超时!  
等待读超时!  
hello everyone

5.4 SO\_LINGER 选项

设置该选项: public void setSoLinger(boolean on, int seconds) throws SocketException   
读取该选项: public int getSoLinger() throws SocketException   
SO\_LINGER 选项用来控制 Socket 关闭时的行为. 默认情况下, 执行 Socket 的 close() 方法, 该方法会立即返回, 但底层的 Socket 实际上并不立即关闭, 它会延迟一段时间, 直到发送完所有剩余的数据, 才会真正关闭 Socket, 断开连接.

      如果执行以下方法:

      socket.setSoLinger(true, 0);

      那么执行Socket 的close() 方法, 该方法也会立即返回, 并且底层的 Socket 也会立即关闭, 所有未发送完的剩余数据被丢弃.

      如果执行以下方法:

      socket.setSoLinger(true, 3600);

      那么执行Socket 的 close() 方法, 该方法不会立即返回, 而是进入阻塞状态. 同时, 底层的 Socket 会尝试发送剩余的数据. 只有满足以下两个条件之一, close() 方法才返回:

      ⑴ 底层的 Socket 已经发送完所有的剩余数据;

      ⑵ 尽管底层的 Socket 还没有发送完所有的剩余数据, 但已经阻塞了 3600 秒(注意这里是秒, 而非毫秒), close() 方法的阻塞时间超过 3600 秒, 也会返回, 剩余未发送的数据被丢弃.

      值得注意的是, 在以上两种情况内, 当close() 方法返回后, 底层的 Socket 会被关闭, 断开连接. 此外, setSoLinger(boolean on, int seconds) 方法中的 seconds 参数以秒为单位, 而不是以毫秒为单位.

      如果未设置 SO\_LINGER 选项, getSoLinger() 返回的结果是 -1, 如果设置了 socket.setSoLinger(true, 80) , getSoLinger() 返回的结果是 80.

Tips: 当程序通过输出流写数据时, 仅仅表示程序向网络提交了一批数据, 由网络负责输送到接收方. 当程序关闭 Socket, 有可能这批数据还在网络上传输, 还未到达接收方. 这里所说的 "未发送完的数据" 就是指这种还在网络上传输, 未被接收方接收的数据.

    例子 SimpleClient.java 与 SimpleServer.java 所示是一对简单的客户/服务器程序. SimpleClient 类发送一万个字符给 SimpleServer, 然后调用Socket 的 close() 方法关闭 Socket.

    SimpleServer 通过 ServerSocket 的 accept() 方法接受了 SimpleClient 的连接请求后, 并不立即接收客户发送的数据, 而是睡眠 5 秒钟后再接收数据. 等到 SimpleServer 开始接收数据时, SimpleClient 有可能已经执行了 Socket 的close() 方法, 那么 SimpleServer 还能接收到 SimpleClient 发送的数据吗?

    SimpleClient.java 略..., SimpleServer.java 略......

    SimpleClient.java中

System.out.println("开始关闭 Socket");  
long begin = System.currentTimeMillis();  
socket.close();  
long end = System.currentTimeMillis();  
System.out.println("关闭Socket 所用的时间为:" + (end - begin) + "ms");

    下面分 3 种情况演示 SimpleClient 关闭 Socket 的行为.

    ⑴ 未设置 SO\_LINGER 选项, 当 SimpleClient 执行 Socket 的close() 方法时, 立即返回, SimpleClient 的打印结果如下:

开始关闭 Socket  
关闭Socket 所用的时间为:0ms

     等到 SimpleClient 结束运行, SimpleServer 可能才刚刚结束睡眠, 开始接收 SimpleClient 发送的数据. 此时尽管 SimpleClient 已经执行了 Socket 的 close() 方法, 并且 SimpleClient 程序本身也运行结束了, 但从 SimpleServer 的打印结果可以看出, SimpleServer 仍然接收到了所有的数据. 之所以出现这种情况, 是因为当 SimpleClient 执行了 Socket 的 close() 方法后, 底层的 Socket 实际上并没有真正关闭, 与 SimpleServer 的连接依然存在. 底层的 Socket 会存在一段时间, 直到发送完所有的数据.

     ⑵ 设置SO\_LINGER 选项, socket.setSoLinger(true, 0). 这次当 SimpleClient 执行 Socket 的 close() 方法时, 会强行关闭底层的 Socket, 所有未发送完的数据丢失. SimpleClient 的打印结果如下:

开始关闭 Socket  
关闭Socket 所用的时间为:0ms

     从打印结果看出, SimpleClient 执行 Socket 的 close() 方法时, 也立即返回. 当 SimpleServer 结束睡眠, 开始接收 SimpleClient 发送的数据时, 由于 SimpleClient 已经关闭底层 Socket, 断开连接, 因此 SimpleServer 在读数据时会抛出 SocketException:

        java.net.SocketException: Connection reset

     ⑶ 设置SO\_LINGER 选项, socket.setSoLinger(true, 3600). 这次当 SimpleClient 执行 Socket 的close() 方法时, 会进入阻塞状态, 知道等待了 3600 秒, 或者底层 Socket 已经把所有未发送的剩余数据发送完毕, 才会从 close() 方法返回. SimpleClient 的打印结果如下:

开始关闭 Socket  
关闭Socket 所用的时间为:5648ms

     当 SimpleServer 结束了 5 秒钟的睡眠, 开始接收 SimpleClient 发送的数据时, SimpleClient 还在这些 Socket 的close() 方法, 并且处于阻塞状态. SimpleClient 与 SimpleServer 之间的连接依然存在, 因此 SimpleServer 能够接收到 SimpleClient 发送的所有数据.

5.5 SO\_RCVBUF 选项

设置该选项: public void setReceiveBufferSize(int size) throws SocketException   
读取该选项: public int getReceiveBufferSize() throws SocketException   
SO\_RCVBUF 表示 Socket 的用于输入数据的缓冲区的大小. 一般说来, 传输大的连续的数据块(基于HTTP 或 FTP 协议的通信) 可以使用较大的缓冲区, 这可以减少传输数据的次数, 提高传输数据的效率. 而对于交互频繁且单次传送数据量比较小的通信方式(Telnet 和 网络游戏), 则应该采用小的缓冲区, 确保小批量的数据能及时发送给对方. 这种设定缓冲区大小的原则也同样适用于 Socket 的 SO\_SNDBUF 选项.

      如果底层 Socket 不支持 SO\_RCVBUF 选项, 那么 setReceiveBufferSize() 方法会抛出 SocketException.

5.6 SO\_SNDBUF 选项

设置该选项: public void setSendBufferSize(int size) throws SocketException   
读取该选项: public int getSendBufferSize() throws SocketException   
SO\_SNDBUF 表示 Socket 的用于输出数据的缓冲区的大小. 如果底层 Socket 不支持 SO\_SNDBUF 选项, setSendBufferSize() 方法会抛出 SocketException.

5.7 SO\_KEEPALIVE 选项

设置该选项: public void setKeepAlive(boolean on) throws SocketException   
读取该选项: public boolean getKeepAlive() throws SocketException //原书中这个方法返回的类型是int   
当 SO\_KEEPALIVE 选项为 true 时, 表示底层的TCP 实现会监视该连接是否有效. 当连接处于空闲状态(连接的两端没有互相传送数据) 超过了 2 小时时, 本地的TCP 实现会发送一个数据包给远程的 Socket. 如果远程Socket 没有发回响应, TCP实现就会持续尝试 11 分钟, 直到接收到响应为止. 如果在 12 分钟内未收到响应, TCP 实现就会自动关闭本地Socket, 断开连接. 在不同的网络平台上, TCP实现尝试与远程Socket 对话的时限有所差别.

      SO\_KEEPALIVE 选项的默认值为 false, 表示TCP 不会监视连接是否有效, 不活动的客户端可能会永远存在下去, 而不会注意到服务器已经崩溃.

      以下代码把 SO\_KEEPALIVE 选项设为 true:

        if(!socket.getKeepAlive()) socket.setKeepAlive(true);

5.8 OOBINLINE 选项

设置该选项: public void setOOBInline(boolean on) throws SocketException   
读取该选项: public boolean getOOBInline() throws SocketException //原书中这个方法返回的类型是int   
当 OOBINLINE 为 true 时, 表示支持发送一个字节的 TCP 紧急数据. Socket 类的 sendUrgentData(int data) 方法用于发送一个字节的 TCP紧急数据.

     OOBINLINE 的默认值为 false, 在这种情况下, 当接收方收到紧急数据时不作任何处理, 直接将其丢弃. 如果用户希望发送紧急数据, 应该把 OOBINLINE 设为 true:

          socket.setOOBInline(true);

      此时接收方会把接收到的紧急数据