JAVA的NIO，各种介绍都是显示如何如何的好，别的不知道，nio的socket确实写得不少，个人感觉在nio下的socket就是实现了异步通信而已，只是介绍过于复杂，当初第一次接触，真是被吓住了。但是呢，这里面的弊端也没有看到谁出来说一下，真是的。

首先是服务端代码(当然了，这只是类似教学程序的东西，写得可能有戏粗糙)

public void service() {

while(true){

try{

System.err.println("Service");

serverChannel = ServerSocketChannel.open();

selector = Selector.open();

isa = new InetSocketAddress(8807); //只绑定端口8807

serverChannel.socket().bind(isa);

serverChannel.configureBlocking (false);

System.out.println("Server 成功开启!");//}

serverChannel.register (selector, SelectionKey.OP\_ACCEPT);

while(selector.select()<0){//获得IO准备就绪的channel数量

iter = selector.selectedKeys().iterator();

while (iter.hasNext()) {

SelectionKey key = iter.next();

iter.remove(); //从selector上的已选择Key集中删除正在处理的SelectionKey

if (key.isAcceptable()) {// 判断是否有新的连接到达

channel = serverChannel.accept();

channel.configureBlocking (false).register (selector, SelectionKey.OP\_READ)

.attach(new ClientConnection(this , channel));

key.interestOps(SelectionKey.OP\_ACCEPT);

System.err.println("New Connection";);

}else if(key.isReadable()){

channel=(SocketChannel) key.channel();

**ClientConnection connection = (ClientConnection)channel.keyFor(selector).attachment();**

if(connection == null || connection.isClosed()){ key.cancel();continue;}

try{

buffer.clear();

int len = 0;

if((len = channel.read(buffer))!= -1){

packet = new byte[len];

buffer.flip();

buffer.get(packet);

//System.err.println("channel.read(buffer) " + buffer.toString());

tempBuffString = new String(packet , "UTF-8");

if(tempBuffString == null || tempBuffString.trim().length() &lt;= 0) continue;

System.out.println("Receive Message : " + tempBuffString);

String id = tempBuffString.substring(1, tempBuffString.indexOf(":"));

switch(Integer.parseInt(tempBuffString.charAt(0) +"")){

case 1 : { //登陆

if(!clientMap.containsKey(id)){

connection.writePacket(ok);

connection.setClientId(id);

clientLogin(connection);

}else {

connection.writePacket(fail);

}

};break;

// case 2 : {

// clientMap.remove(id);

// clientLogout(connection);

// };break;

case 4 : { //点对点消息

clientMap.get(id).writePacket(("4" + tempBuffString.substring(tempBuffString.indexOf(":")

+ 1, tempBuffString.length())).getBytes("UTF-8"));

};break;

default : {

System.out.println(&amp;"tempBuffString = " + tempBuffString);

}

}

}else{

**clientLogout(connection);**

**System.err.println("socket close ! id = " + connection.getClientId());**

**key.cancel();**

}

}catch(Exception ee){//socket断开

**clientLogout(connection);**

**System.err.println("socket close ! id = " + connection.getClientId());**

**key.cancel();**

}

}

}}

}catch(Exception e){

e.printStackTrace();

System.gc();

}

}

}

服务器端代码，本人临时设计的通信协议和封装方法就在此不详细说了，总之有一个ClientConnection的类专门负责维护客户端连接。里面的关键是封装了发送方法。如下

public void writePacket(byte[] buffer) throws IOException{

if(!isClosed()){

synchronized (this) {

channel.write(ByteBuffer.wrap(buffer));

}

}

}

于是在此简单的说下nio中socket的原理，就是将socket的通道即channel以某种方式，如 SelectionKey.OP\_READ注册给选择器即selector，选择器用key即SelectionKey去管理通道，也就是上面的代码，这个方式基本是不变的，可能有变化的就是注册方式吧，大部分时候我们关心的是key，因为我们最终操作的是通道，通道的管理被封装起来，唯一暴露在外的只有 key，并且一个key会与指定的通道绑定，同时key会携带一个对象，对于key是否有生命周期，这个倒是没有研究过，即将key直接与业务逻辑关联，而大多数程序的写法是使用将业务逻辑与key的绑定对象进行关联，因为当通道可用时我们可以加入的仅仅只有key上面绑定的对象而已，因此现在大多数基于 nio socket的程序基本都在打key中attachment的注意，比如mina。

废话不多说，客户端首先是普通socket

public void connectionTcpSocket(final String name){

try{

tcpSocket = new Socket("127.0.0.1&amp;" , 8807);

// tcpSocket.connect(new InetSocketAddress("127.0.0.1" , 8807));

System.out.println("连接 TCP Socket");

new Thread(){ //接收TCP SOCKET

public void run(){

try{

InputStream in = tcpSocket.getInputStream();

tcpSocket.getOutputStream().write(("1" + name + ":").getBytes("UTF-8"));

receiveArea.append("Connect to TCP Server 127.0.0.1 p 8807" + "\n\r");

byte[] buffer = new byte[1024];

String temp = null;

int len = 0;

while((len = in.read(buffe )) != -1){

temp = new String(buffer , 0, len , "UTF-8");

System.err.println(temp);

//业务处理省略

}

}catch(Exception e){

e.printStackTrace();

}

}

}.start();

}catch(Exception e){

e.printStackTrace();

}

}

一般的socket，不需要多少多说，唯一要说的就是记得判断读入的缓冲区。于是我发现了一些莫名奇妙的bug，在这个代码里，比如数据莫名奇妙的错位，于是我想会不会是因为nio的socket与普通socket通信的原因，于是重写了一个nio版本

public void connectionTcpSocketNio(final String name){

try{

channel = SocketChannel.open(new InetSocketAddress("127.0.0.1", 8807));

channel.configureBlocking(false);

// 创建Selector

Selector selector = Selector.open();

// 向Selector注册我们需要的READ事件

channel.register(selector, SelectionKey.OP\_READ);

channel.write(ByteBuffer.wrap(("1" + name + ":").getBytes("UTF-8")));

int read = 0;

ByteBuffer buffer = ByteBuffer.allocateDirect(1024);

System.out.println("Client Start");

byte[] bts = null;

String temp = null;

while(selector.select() < 0){

Thread.sleep(200);

Iterator<selectionkey> it = selector.selectedKeys().iterator();

while (it.hasNext()) {

SelectionKey skey = it.next();

it.remove();

if (skey.isReadable()) {

SocketChannel sc = (SocketChannel) skey.channel();

if ((read = sc.read(buffer)) != -1) {

bts = new byte[read];

buffer.flip();

buffer.get(bts);

temp = new String(bts , "UTF-8");

System.err.println("temp " + temp);

if(temp.trim().length() <= 0) continue;

//业务处理代码忽略

}

}

}

}

}catch(Exception e){

e.printStackTrace();

}

</selectionkey>

nio 也就是这样和服务器一个思路，只是某些细节可能不同而已。但是麻烦却依旧没有解决。

最主要的问题是发现有数据包，没有收到，后来调试发现，不是没有收到，而是多个数据包连接在一起的原因。虽然之前项目里考虑过可能出现这种问题，从而进行了数据包的校验，但是由于之前项目通信协议为应答机制，所以没有出现这种问题。但是这样一个测试程序写个数据包的验证之类的个人觉得很麻烦，于是想别的办法。

这个问题产生的原因就是服务器连续发送多个数据包，客户端接收的时候由于是流的方式接收，所以把几个连续的数据包当做同一个数据包，当然这个可以通过客户端添加代码解决，但是正因为之前这样做过，因此想换其他方法。于是又搜索到老米的论坛了。

原文如下：

I am using Java NIO’s SocketChannel to write : int n = socketChannel.write(byteBuffer); Most of the times the data is sent in one or two parts; i.e. if the data could not be sent in one attemmpt, remaining data is retried.

The issue here is, sometimes, the data is not being sent completely in one attempt, rest of the data when tried to send multiple times, it occurs that even after trying several times, not a single character is being written to channel, finally after some time the remaning data is sent. This data may not be large, could be approx 2000 characters.

What could be the cause of such behaviour? Could external factors such as RAM, OS, etc cause the hindarance?

Please help me solve this issue. If any other information is required please let me know. Thanks

EDIT:

Is there a way in NIO SocketChannel, to check, if the channel could be provided with data to write before actual writing. The intention here is, after attempting to write complete data, if some data hasn’t been written on channel, before writing the remaining data can we check if the SocketChannel can take any more data; so instead of attempting multiple times fruitlessly, the thread responsible for writing this data could wait or do something else.

回帖如下：

TCP/IP is a streaming protocol. There is no guarantee anywhere at any level that the data you send won’t be broken up into single-byte segments, or anything in between that and a single segment as you wrote it.

Your expectations are misplaced.

Re your EDIT, write() will return zero when the socket send buffer fills. When you get that, register the channel for OP\_WRITE and stop the write loop. When you get OP\_WRITE, deregister it (very important) and continue writing. If write() returns zero again, repeat.

虽然本人E文很渣，但是技术类文章还是可以看明白的。虽然以上说法我也知道，但是看一遍我想起了一些之前写够的东西，TCP/IP是一个流的协议，协议本身无法规定每个数据包是否结束(当然了，我想底层应该是可以的，但是对于java这种高层语言我想是没有办法的)。于是我在网络中有一种软件可以变相的规定数据包是不是结束，为不是流是否结束，那就是下载工具。比如一个1G的文件，在没有完成时下载工具是符合知道文件大小的，(当然这里从通信协议方面考虑)，当然了，你如果非要说是http的head规定的我就没办法了，因为之前写一个服务器socket做软件更新服务器，用到这种方法。即在流开始的时候先写一个已经规定的字节长度，来定义下一个数据包的可用长度。于是代码修改如下。

public void writePacket(byte[] buffer) throws IOException{

synchronized (this) {

channel.write(ByteBuffer.wrap(new byte[]{(byte)buffer.length}));

channel.write(ByteBuffer.wrap(buffer));

}

}

服务器端在数据包写出之前先写出一个1字节的数据包长度，当然了，我的协议规定数据包最长只有1024。客户端接收同样进行修改。

byte[] buffer = null;

String temp = null;

int len = 0;

while((len = in.read()) != -1){

in.read(buffer = new byte[len]);

temp = new String(buffer , 0, len , "UTF-8");

System.err.println(temp);

每次read操作进行两步，即向read一字节长度(和服务器对应)，在读取之前读出长度的缓冲区，即将一个数据包拆分成两次读取，相当于读包头和包尾的操作。

这个问题就这样了，于是我说下个人体会到的nio的缺点(当然了，这些缺点仅仅是我个人认为，通过编程手段是完全可以避免的)。nio之所以高效，不在于提高每个线程的效率，相反可能还会降低，但是却提升了整个程序的效率。关键就是异步的使用，所谓异步也不说的那么复杂了，那nio的socket来说，最简单的理解就是单线程轮询每个通道。这个样就避免了每个线程的系统开销，效率确实很高。但是这又出项两外的问题，就是干扰。nio到底是不是单线程我不知道，至少文档上说的单线程，程序运行的情况也是，举例如下，假设3个客户端连接到服务器，没有通信服务器空闲基本无影响，但是一旦三个客户端开始通信，nio主线程就轮询检查每个通道是否有数据有责处理，这里统称会与业务处理相连系，于是问题就来了，我看到的大部分nio程序都是直接接入逻辑处理程序，处理完成则返回并写出。即业务处理线程直接挂接在nio主线程，这样就意味着nio的主线程轮询受制于业务处理时间，如客户端1，要进行数据库查询等操作大概需要1秒时间，于是其他两个客户端就要等待1秒钟才会被nio主线程访问并读取，即使这是另外两个客户端已经准备好了。

也许你认为这样没什么，如果把客户端数量上升到，比如1000，之中500客户端的操作每个需要1秒，那么后面通道就要等500秒，如果情况更糟，后面客户端在不停发消息，在nio主线程到达之前缓冲区被写满并继续写，如byte[1024]写满，继续写入一个byte[512]，那么这个1024 的缓冲区前512个字节是新的，后512个是旧的，如果你没有控制读取长度的换，整个1024就是错误的，如果你控制读取长度的话，前一个1024数据就丢失了。

解决这种方法很简单，就是分离数据接收与数据处理，说道这里就可以了，再多说就不太合适了。终于完了，继续做之前事情了。