**tcp 和 udp的区别：**

tcp 和 udp 是 OSI 模型中的运输层中的协议。tcp 提供可靠的通信传输，而 udp 则常被用于让广播和细节控制交给应用的通信传输。

两者的区别大致如下：

tcp 面向连接，udp 面向非连接即发送数据前不需要建立链接；

tcp 提供可靠的服务（数据传输），udp 无法保证；

tcp 面向字节流，udp 面向报文；

tcp 数据传输慢，udp 数据传输快；

**socket长连接短连接，连接出现异常你是怎么处理的：**

短连接：在一次TCP通讯中，先通过三次握手建立连接，然后客户端和服务端通讯，然后断开连接。一次通讯建立一次连接。

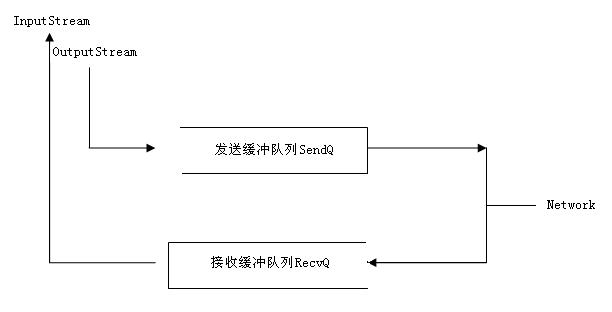
长连接：即是客户端和服务端建立连接，经过一次通讯后不会立即断开。客户端和服务端通过心跳保持连接，在所有读写结束后断开连接。

在一些操作频繁，并发数不是很多的点对点情况下，适合使用长连接。

在一些操作不多，但并发很大的一点对多点的情况下，使用短连接。

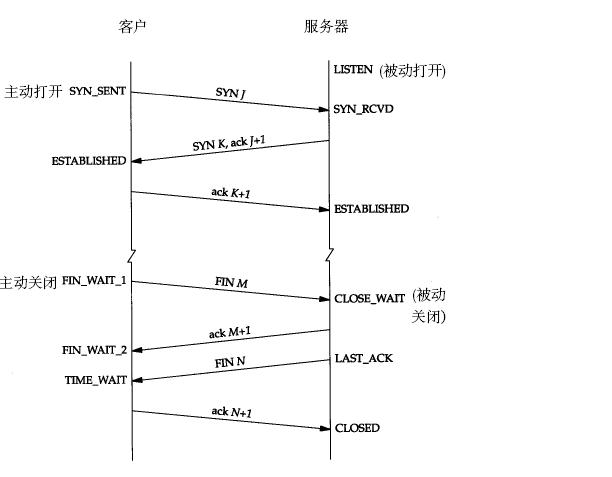
异常处理：Linux socket通信和异常处理：<https://blog.csdn.net/lh2016rocky/article/details/70858859>

▶超时：套接字底层是基于TCP的，所以socket的超时和TCP超时是相同的。



JAVA可以设置读写缓冲区的大小-setReceiveBufferSize(int size), setSendBufferSize(int size)。向输出流写数据并不意味着数据实际上已经被发送，它们只是被复制到了发送缓冲区队列SendQ，就是在Socket的OutputStream上调用flush()方法，也不能保证数据能够立即发送到网络。真正的数据发送是由操作系统的TCP协议栈模块从缓冲区中取数据发送到网络来完成的。当有数据从网络来到时，TCP协议栈模块接收数据并放入接收缓冲区队列RecvQ，输入流InputStream通过read方法从RecvQ中取出数据。

socket连接建立是基于TCP的连接建立过程。TCP的连接需要通过3次握手报文来完成，开始建立TCP连接时需要发送同步SYN报文，然后等待确认报文SYN+ACK，最后再发送确认报文ACK。TCP连接的关闭通过4次挥手来完成，主动关闭TCP连接的一方发送FIN报文，等待对方的确认报文；被动关闭的一方也发送FIN报文，然等待确认报文。



正在等待TCP连接请求的一端有一个固定长度的连接队列，该队列中的连接已经被TCP接受（即三次握手已经完成），但还没有被应用层所接受。TCP接受一个连接是将其放入这个连接队列，而应用层接受连接是将其从该队列中移出。应用层可以通过设置backlog变量来指明该连接队列的最大长度，即已被TCP接受而等待应用层接受的最大连接数。当一个连接请求SYN到达时，TCP确定是否接受这个连接。如果队列中还有空间，TCP模块将对SYN进行确认并完成连接的建立。但应用层只有在三次握手中的第三个报文收到后才会知道这个新连接。如果队列没有空间，TCP将不理会收到的SYN。如果应用层不能及时接受已被TCP接受的连接，这些连接可能占满整个连接队列，新的连接请求可能不被响应而会超时。如果一个连接请求SYN发送后，一段时间后没有收到确认SYN+ACK，TCP会重传这个连接请求SYN两次，每次重传的时间间隔加倍，在规定的时间内仍没有收到SYN+ACK，TCP将放弃这个连接请求，连接建立就超时了。

①socket连接超时：可以设置Socket连接建立的超时时间-

connect(SocketAddress endpoint, int timeout)

如果在timeout内，连接没有建立成功，在TimeoutException异常被抛出。如果timeout的值小于三次握手的时间，那么Socket连接永远也不会建立。

**import java.net.\*;**

**import java.io.\*;**

**public class SocketClientTest{**

**public static final int PORT = [8088](https://www.baidu.com/s?wd=8088&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/yihui8/article/details/_blank);**

**public static void main( String[] args ) throws Exception {**

**InetAddress addr = InetAddress.getByName( "[127.0.0.1](https://www.baidu.com/s?wd=127.0.0.1&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd" \t "https://blog.csdn.net/yihui8/article/details/_blank)" );**

**Socket socket = new Socket();**

**try{**

**socket.connect( new InetSocketAddress( addr, PORT ), 30000 );**

**socket.setSendBufferSize(100);**

**BufferedWriter out = new BufferWriter(new OutputStreamWriter(**

**socket.getOutputStream()));**

**int i = 0;**

**while( true ){**

**System.out.println( "client sent --- hello \*\*\* " + i++ );**

**out.write( "client sent --- hello \*\*\* " + i );**

**out.flush();**

**Thread.sleep( 1000 );**

**}**

**}**

**finally {**

**socket.close();**

**}**

**}**

**}**

②socket读超时：

如果输入缓冲队列RecvQ中没有数据，read操作会一直阻塞而挂起线程，直到有新的数据到来或者有异常产生。调用setSoTimeout(int timeout)可以设置超时时间，如果到了超时时间仍没有数据，read会抛出一个SocketTimeoutException，程序需要捕获这个异常，但是当前的socket连接仍然是有效的。如果对方进程崩溃、对方机器突然重启、网络断开，本端的read会一直阻塞下去，这时设置超时时间是非常重要的，否则调用read的线程会一直挂起。TCP模块把接收到的数据放入RecvQ中，直到应用层调用输入流的read方法来读取。如果RecvQ队列被填满了，这时TCP会根据滑动窗口机制通知对方不要继续发送数据，本端停止接收从对端发送来的数据，直到接收者应用程序调用输入流的read方法后腾出了空间。

**public class Client {**

**public static void main(String args[]) throws Exception {**

**//为了简单起见，所有的异常都直接往外抛**

**String host = "127.0.0.1"; //要连接的服务端IP地址**

**int port = 8899; //要连接的服务端对应的监听端口**

**//与服务端建立连接**

**Socket client = new Socket(host, port);**

**//建立连接后就可以往服务端写数据了**

**Writer writer = new OutputStreamWriter(client.getOutputStream());**

**writer.write("Hello Server.");**

**writer.write("eof\n");**

**writer.flush();**

**//写完以后进行读操作**

**BufferedReader br = new**

**BufferedReader(new InputStreamReader(client.getInputStream()));**

**//设置超时间为10秒**

**client.setSoTimeout(10\*1000);**

**StringBuffer sb = new StringBuffer();**

**String temp;**

**int index;**

**try {**

**while ((temp=br.readLine()) != null) {**

**if ((index = temp.indexOf("eof")) != -1) {**

**sb.append(temp.substring(0, index));**

**break;**

**}**

**sb.append(temp);**

**}**

**} catch (SocketTimeoutException e) {**

**System.out.println("数据读取超时。");**

**}**

**System.out.println("from server: " + sb);**

**writer.close();**

**br.close();**

**client.close();**

**}**

**}**

▶断开连接：TCP Socket连接是双向的，通过四次挥手的方式断开，双方分别调用Socket.close()方法断开连接。连接断开的过程中，一般一方A先断开连接，另一方B发现A断开连接后，也断开连接。为方便表述，将先断开连接的一方A称为“主动断开连接”；后断开的一方B，则为“被动断开连接”。为保证主动断开连接的一方不会阻塞在in.readUTF()方法中，需要先执行socket.shutdownInput()。所以主动断开连接的代码如下。

**socket.shutdownInput();**

**in.close();**

**socket.close();**

被动断开连接的一方，在捕获到in.readUTF()的异常后，断开Socket连接。

**try {**

**String s = in.readUTF();**

**} catch (IOException e) {// 连接被断开(被动)try {**

**in.close();**

**socket.close();**

**in = null;**

**socket = null;**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**}**

**IO NIO区别，NIO原理**：<https://my.oschina.net/kinglaw007/blog/1940757>

①IO是面向字节流和字符流的，而NIO是面向缓冲区的。

②IO是阻塞模式的，NIO是非阻塞模式的。

③NIO新增了选择器的概念，可以通过选择器监听多个通道。

NIO中重要的类：

Buffer（一块连续的内存块，是NIO数据读写的中转地）：ByteBuffer、CharBuffer、IntBuffer、DoubleBuffer、Floatbuffer、LongBuffer、ShortBuffer

Channel：FileChannel、ServerSocketChannel（TCP通信服务端）、SocketChannel（TCP通信客户端）、DatagramChannel（UDP通信）

Charset

Selector

**NOTE--** 通道（channel）和流（stream）：

通道是双向的，既可以读数据由可以写数据；流的读写是单向的。通道的读写总是要把 数据先写入到一个Buffer中。

FileChannel test：

**public void writeFile(){  
 ByteBuffer byteBuffer;  
 FileChannel fileChannel = null;  
 FileOutputStream fileOutputStream = null;  
 try {  
 fileOutputStream = new**

**FileOutputStream(new File("C:\\Users\\LXX\\Desktop\\a.txt"),false);  
 fileChannel = fileOutputStream.getChannel();  
 byteBuffer = ByteBuffer.wrap**

**("hello world,你好".getBytes("utf-8"));//position位置不变  
 //byteBuffer.flip();//读写转换,这里不需要  
 byteBuffer.clear();  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 try {  
 fileChannel.close();  
 fileOutputStream.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 }  
}**

**public void readFile(){  
 FileInputStream fileInputStream = null;  
 FileChannel fileChannel = null;  
 ByteBuffer byteBuffer;  
 try {  
 fileInputStream = new**

**FileInputStream(new File("C:\\Users\\LXX\\Desktop\\a.txt"));  
 fileChannel = fileInputStream.getChannel();  
 byteBuffer = ByteBuffer.allocate(5);  
 //方法一  
// int result = fileChannel.read(byteBuffer);  
// byteBuffer.flip();  
// while (byteBuffer.hasRemaining()){  
// System.out.println(new String(byteBuffer.array()));  
// fileChannel.read(byteBuffer);  
// byteBuffer.flip();  
// }  
 //方法二  
 while (fileChannel.read(byteBuffer) != -1){  
 byteBuffer.flip();  
 System.out.println(new String(byteBuffer.array()));  
 while (byteBuffer.hasRemaining()){  
 System.out.println((char)byteBuffer.get());  
 }  
 byteBuffer.clear();  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
 }**

**BIO,NIO,AIO区别：**<https://blog.csdn.net/caohongshuang/article/details/79455391>

BIO：同步阻塞IO。socket就是BIO，一个socket连接一个处理线程，这个线程处理这个socket连接的一系列数据传输操作。阻塞的原因在于操作系统的线程数量是有限的，多个socket申请与服务端建立连接时，服务端不能提供相应数量的处理线程，没有分配到的连接阻塞等待或被拒绝。

NIO：同步非阻塞IO。基于轮询。NEW IO是对BIO的改进，当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器（selector）上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

AIO：异步IO。基于回调。两种操作方式：Future方式，CallBack方式

▶BIO代码：

服务端：

**public class Server {  
 private static ServerSocket serverSocket;  
 private static ExecutorService executorService**

**= Executors.newFixedThreadPool(5);  
 public synchronized static void start() throws IOException{  
 if (serverSocket != null){  
 return;  
 }  
 try {  
 serverSocket = new ServerSocket(8089);  
 System.out.println("服务器已启动，端口号："+8089);  
 int i = 1;  
 while (true){**

**//等待连接，没有可以接受的连接会阻塞等待  
 Socket socket =**

**serverSocket.accept();**

**BufferedReader bufferedReade =**

**new BufferedReader**

**(new InputStreamReader(socket.getInputStream()));  
PrintWriter printWriter =**

**new PrintWriter(socket.getOutputStream(),true);  
 String expression;  
 String result;  
 if ((expression = bufferedReade.readLine()) == null){  
 break;  
 }  
 System.out.println("服务器收到消息："+expression);  
 printWriter.println(expression+"<>");**

**//newThread(newServerHandler(socket),String.valueOf(i)).start();**

**//对每一个连接都创建一个线程处理（一个线程对应一个连接）  
 // executorService.execute(new ServerHandler(socket));**

**//使用线程池，FixedThreadPool我们就有效的控制了线程的最大数量， //保证了系统有限的资源的控制，实现了N:M的伪异步I/O模型  
 i++;  
 }  
 }finally {  
 }  
 }  
}**

ServerHandler：

**public class ServerHandler implements Runnable {  
 private Socket socket;  
 public ServerHandler(Socket socket) {  
 this.socket = socket;  
 }  
 @Override  
 public void run() {  
 try (BufferedReader bufferedReade = new BufferedReader**

**(new InputStreamReader(socket.getInputStream()));  
 PrintWriter printWriter =**

**new PrintWriter(socket.getOutputStream(),true)){  
String expression;  
String result;  
while (true){  
 if ((expression = bufferedReade.readLine()) == null){  
 break;  
 }  
 System.out.println(Thread.currentThread().getName()+"：服务器收到消息："+expression);  
 printWriter.println(expression+"<>");  
}  
}catch (Exception e){**

**e.printStackTrace();**

**}**

**}**

**}**

客户端：

**public class Client {  
 public static void send(String expression) throws IOException{  
 System.out.println("client generate expression:"+expression);  
 try (Socket socket = new Socket(“127.0.0.1”,8089){  
 BufferedReader bufferedReader = new BufferedReader**

**(new InputStreamReader(socket.getInputStream()));  
 PrintWriter printWriter =**

**new PrintWriter(socket.getOutputStream(),true)){  
 printWriter.println(expression);  
 system.out.println("收到服务器消息："+bufferedReader.readLine());  
 }  
 }**

**}**

▶NIO代码：

服务端：

**public void serviceWithSelector(){  
 try {  
 ServerSocketChannel serverSocketChannel =**

**ServerSocketChannel.open();  
 serverSocketChannel.socket().bind(new InetSocketAddress(8090));  
 serverSocketChannel.configureBlocking(false);  
 Selector selector = Selector.open();  
 //向选择器注册通道的接受事件  
 serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  
 while (true){  
 //获取已经准备好的通道数量  
 int readyChannels = selector.selectNow();  
 //如果没有准备好，重试  
 if (readyChannels == 0){  
 continue;  
 }  
 //获取准备好的通道中的事件集合  
 Set selectKeys = selector.selectedKeys();  
 Iterator iterator = selectKeys.iterator();  
 while (iterator.hasNext()){  
 SelectionKey key = (SelectionKey) iterator.next();  
 if (key.isAcceptable()){  
 //在自己注册的事件中写业务逻辑  
 ServerSocketChannel serverSocketChannel1 =**

**(ServerSocketChannel) key.channel();  
 SocketChannel socketChannel =**

**serverSocketChannel1.accept();  
 ByteBuffer buffer = ByteBuffer.allocate(1024);  
 socketChannel.read(buffer);  
 buffer.flip();  
 if (buffer.hasRemaining()){  
 System.out.println(">>>服务端收到数据："+new String(buffer.array()));  
 }  
 buffer.clear();  
 ByteBuffer header = ByteBuffer.allocate(6);  
 header.put("[head]".getBytes());  
 ByteBuffer body = ByteBuffer.allocate(1024);  
 body.put("i am body!".getBytes());  
 System.out.println(header.mark());  
 header.flip();  
 body.flip();  
 ByteBuffer[] buffers = {header,body};  
 socketChannel.write(buffers);  
 socketChannel.close();  
 }**

**}  
}**

**}catch (Exception e){  
 e.printStackTrace();  
 }  
}**

客户端：

**public void request(){  
 try {  
 SocketChannel socketChannel = SocketChannel.open();  
 socketChannel.connect(new InetSocketAddress("localhost",8090));  
 ByteBuffer buffer = ByteBuffer.allocate(1024);  
 buffer.put("来自客户端的请求".getBytes());  
 buffer.flip();  
 if (buffer.hasRemaining()){  
 socketChannel.write(buffer);  
 }  
 buffer.clear();  
 //接受服务端的返回  
 ByteBuffer header = ByteBuffer.allocate(6);  
 ByteBuffer body = ByteBuffer.allocate(1024);  
 ByteBuffer[] buffers = {header,body};  
 //分散：从一个通道中把内容写入多个缓存中  
 socketChannel.read(buffers);  
 header.flip();  
 body.flip();  
 System.out.println(header.mark());  
 System.out.println(body.mark());  
 if (header.hasRemaining()){  
 System.out.println**

**(">>>客户端接受头部数据："+new String(header.array()));  
 }  
 if (header.hasRemaining()){  
 System.out.println**

**(">>>客户端接受body数据："+new String(body.array()));  
 }  
 socketChannel.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
}**

▶AIO代码：

服务端：

客户端：

http/https,webservice,socket,websocket：<https://www.2cto.com/kf/201805/744083.html>

后续netty网络编程