# 1、Java 的io 流

## 1、Java支持的三种IO模型

Io模型就是使用什么通道进行数据的接收与发送，Java共支持3种网络编程io模式：BIO，NIO，AIO

### 1.1 BIO（blocking io）

同步阻塞模型，一个客户端连接对应一个窗户里线程

**缺点：**

1、io代码里的read操作是阻塞操作，如果连接不做数据读写操作会导致线程阻塞，浪费资源

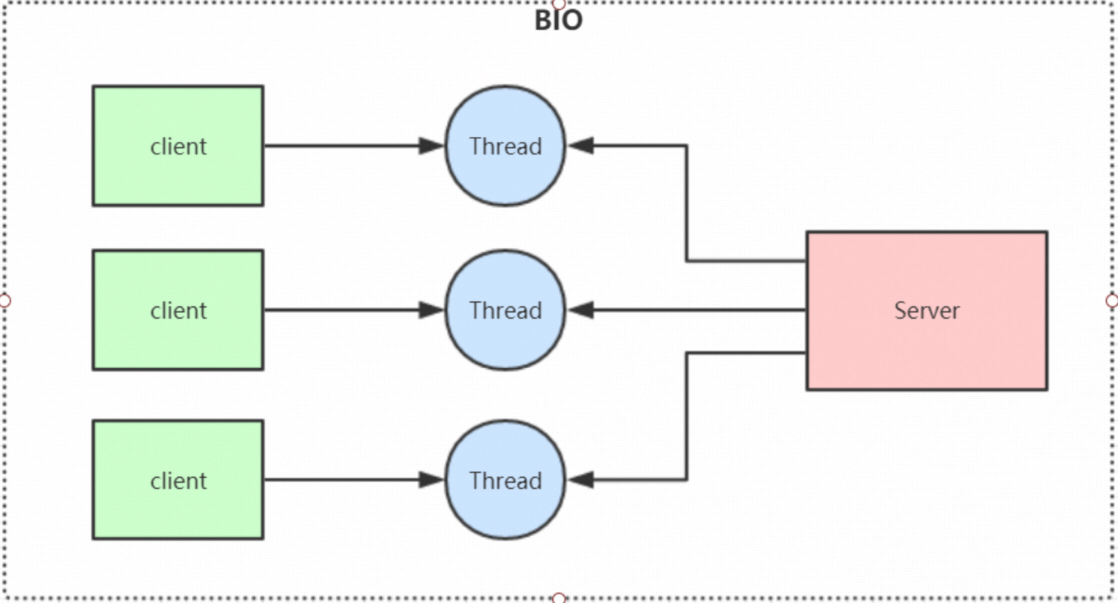
2、如果线程很多，会导致服务器线程太多，压力太大

**解决：**

可以在具体业务处理的地方创建一个线程池，由子线程处理具体业务，主线程返回继续处理新的连接

**应用场景：**

BIO方式适用于连接数目较小的的且固定的架构，这种方式对服务器资源要求比较高，但程序简单易理解

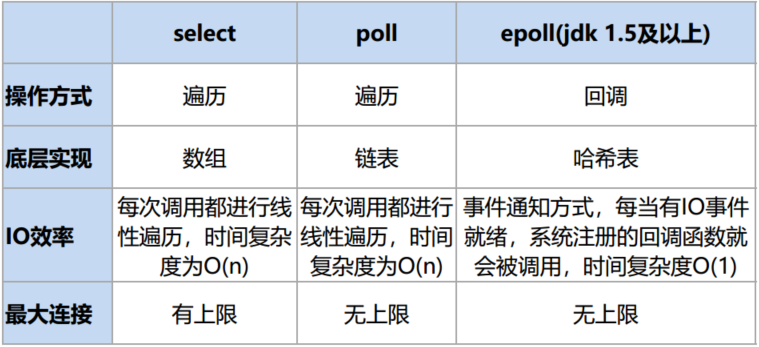


1 //服务端示例  
2 public class SocketServer {  
3 public static void main(String[ ] args) throws IOException {  
4 ServerSocket serverSocket = new ServerSocket(9000) ;  
5 while (true) {  
6 System. out. println("等待连接。 。 ") ;  
7 Socket socket = serverSocket. accept() ; //阻塞方法  
8 System. out. println("有客户端连接了。 。 ") ;  
9 new Thread(new Runnable() {  
10 @Override  
11 public void run() {  
12 try {  
13 handler(socket) ;  
14 } catch (IOException e) {  
15 e. printStackTrace() ;  
16 }  
17 }  
18 }) . start() ;  
19 }  
20 }  
21  
22 private static void handler(Socket socket) throws IOException {  
23 System. out. println("thread id = " + Thread. currentThread() . getId() ) ;  
24 byte[ ] bytes = new byte[ 1024] ;  
25  
26 System. out. println("准备read。 。 ") ;  
27 //接收客户端的数据， 阻塞方法， 没有数据可读时就阻塞  
28 int read = socket. getInputStream() . read(bytes) ;  
29 System. out. println("read完毕。 。 ") ;  
30 if (read ! = ‐1) {  
31 System. out. println("接收到客户端的数据： " + new String(bytes, 0, read) ) ;  
32 System. out. println("thread id = " + Thread. currentThread() . getId() ) ;  
33  
34 }  
35 socket. getOutputStream() . write("HelloClient". getBytes() ) ;  
36 socket. getOutputStream() . flush() ;  
37 }  
38 }  
39  
1 //客户端代码  
2 public class SocketClient {  
3 public static void main(String[ ] args) throws IOException {  
4 Socket socket = new Socket("localhost", 9000) ;  
5 //向服务端发送数据  
6 socket. getOutputStream() . write("HelloServer". getBytes() ) ;  
7 socket. getOutputStream() . flush() ;  
8 System. out. println("向服务端发送数据结束") ;  
9 byte[ ] bytes = new byte[ 1024] ;  
10 //接收服务端回传的数据  
11 socket. getInputStream() . read(bytes) ;  
12 System. out. println("接收到服务端的数据： " + new String(bytes) ) ;  
13 socket. close() ;  
14 }  
15 }

### 1.2 NIO（Non blocking io）

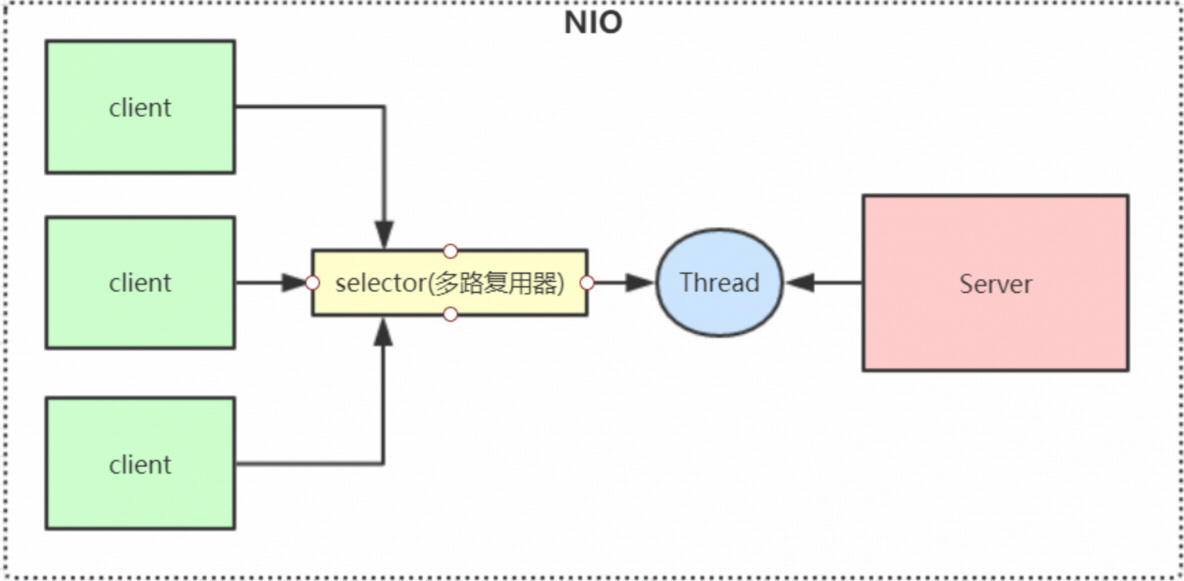
同步非阻塞，服务器实现模式为一个线程可以处理多个请求（连接），客户端发送的连接请求都会注册到多路复用器selector，多路复用器轮询到连接有io请求就进行处理

Io多路复用底层一般用Linux API（select，poll，epoll）来实现

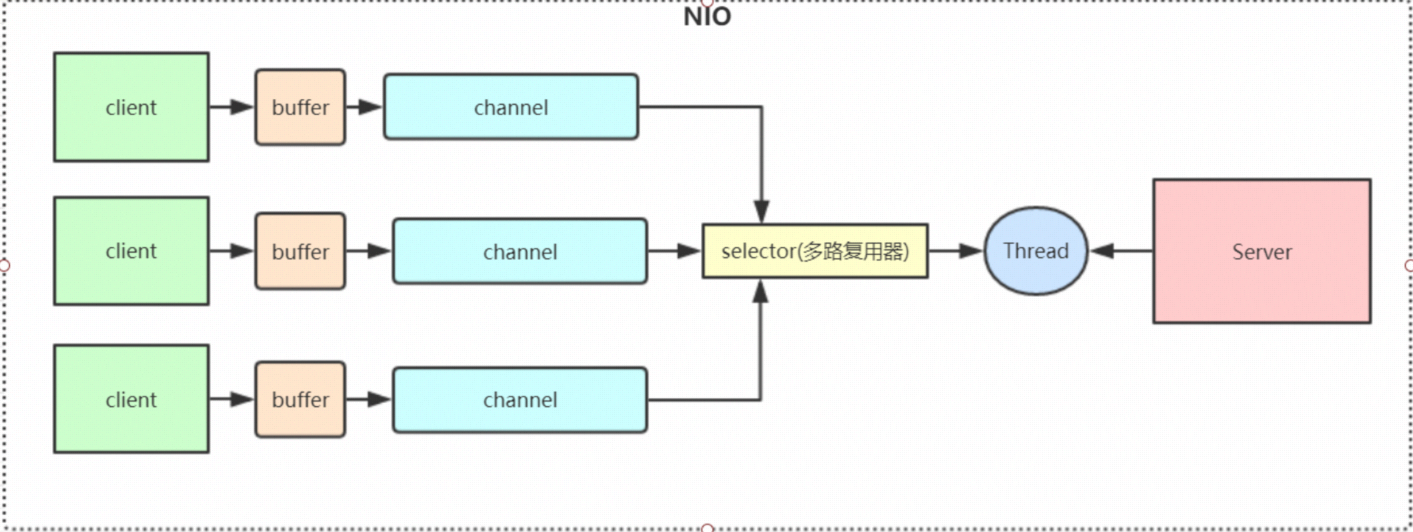


**应用场景**：

NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，弹幕系统，服务器间通讯，编程比较复杂，jdk1.4开始支持



NIO有三大核心组件：channel（通道），Buffer（缓冲区），Selector（选择器）



1、channel类似于流，每个channel对应一个buffer缓冲区，buffer底层就是个数组

2、channel会注册到selector上，由selector根据channel读写事件的发生将其交由某个空闲的线程处理

3、selector可以对应一个或多个线程

4、NIO的Buffer和channel都是即可以读又可以写的

1 //服务端代码  
2 public class NIOServer {  
3 public static void main(String[ ] args) throws IOException {  
4 // 创建一个在本地端口进行监听的服务Socket通道. 并设置为非阻塞方式  
5 ServerSocketChannel ssc = ServerSocketChannel. open() ;  
6 //必须配置为非阻塞才能往selector上注册， 否则会报错， selector模式本身就是非阻塞模式  
7 ssc. configureBlocking(false) ;  
8 ssc. socket() . bind(new InetSocketAddress(9000) ) ;  
91  
0 // 创建一个选择器并将serverSocketChannel注册到它上面  
11 Selector selector = Selector. open() ;  
12 // 把channel注册到selector上， 并且selector对客户端accept连接操作感兴趣  
13 ssc. register(selector, SelectionKey. OP\_ACCEPT) ;  
14  
15 while (true) {  
16 System. out. println("等待事件发生。 。 ") ;  
17 // 轮询监听key， select是阻塞的， accept()也是阻塞的  
18 selector. select() ;  
19 System. out. println("有事件发生了。 。 ") ;  
20 // 有客户端请求， 被轮询监听到  
21 Iterator<SelectionKey> it = selector. selectedKeys() . iterator() ;  
22 while (it. hasNext() ) {  
23 SelectionKey key = it. next() ;  
24 //删除本次已处理的key， 防止下次select重复处理  
25 it. remove() ;  
26 handle(key) ;  
27 }  
28 }  
29  
30 }  
31  
32 private static void handle(SelectionKey key) throws IOException {  
33 if (key. isAcceptable() ) {  
34 System. out. println("有客户端连接事件发生了。 。 ") ;  
35 ServerSocketChannel ssc = (ServerSocketChannel) key. channel() ;  
36 //此处accept方法是阻塞的， 但是这里因为是发生了连接事件， 所以这个方法会马上执行完  
37 SocketChannel sc = ssc. accept() ;  
38 sc. configureBlocking(false) ;  
39 //通过Selector监听Channel时对读事件感兴趣  
40 sc. register(key. selector() , SelectionKey. OP\_READ) ;  
41 } else if (key. isReadable() ) {  
42 System. out. println("有客户端数据可读事件发生了。 。 ") ;  
43 SocketChannel sc = (SocketChannel) key. channel() ;  
44 ByteBuffer buffer = ByteBuffer. allocate(1024) ;  
45 buffer. clear() ;  
46  
47 int len = sc. read(buffer) ;  
48 if (len ! = ‐1) {  
49 System. out. println("读取到客户端发送的数据： " + new String(buffer. array() , 0, len) ) ;  
50 }  
51 ByteBuffer bufferToWrite = ByteBuffer. wrap("HelloClient". getBytes() ) ;  
52 sc. write(bufferToWrite) ;  
53 key. interestOps(SelectionKey. OP\_READ | SelectionKey. OP\_WRITE) ;  
54 sc. close() ;  
55 }  
56 }  
57 }

**NIO服务端程序详细分析：**

1、创建一个ServerSocketChannel和Selector，并将ServerSocketChannel注册到selector上

2、selector通过select()方法监听channel事件，当客户端连接时，selector监听到连接事件，获取到ServerSocketChannel注册时绑定的selectionKey

3、selectionKey通过channel()方法可以获取绑定的ServerSocketChannel

4、ServerSocketChannel通过accept()方法得到SocketChannel

5、将SocketChannel注册到Selector上，关心read事件

6、注册后返回一个SelectionKey，会和该SocketChannel关联

7、selector继续通过select()方法监听事件，当客户端发送数据给服务端，selector监听到read事件，获取到SocketChannel注册时绑定的selectionKey

8、selectionKey通过channel()方法可以获取绑定的SocketChannel

9、将socketChannel里的数据读取出来

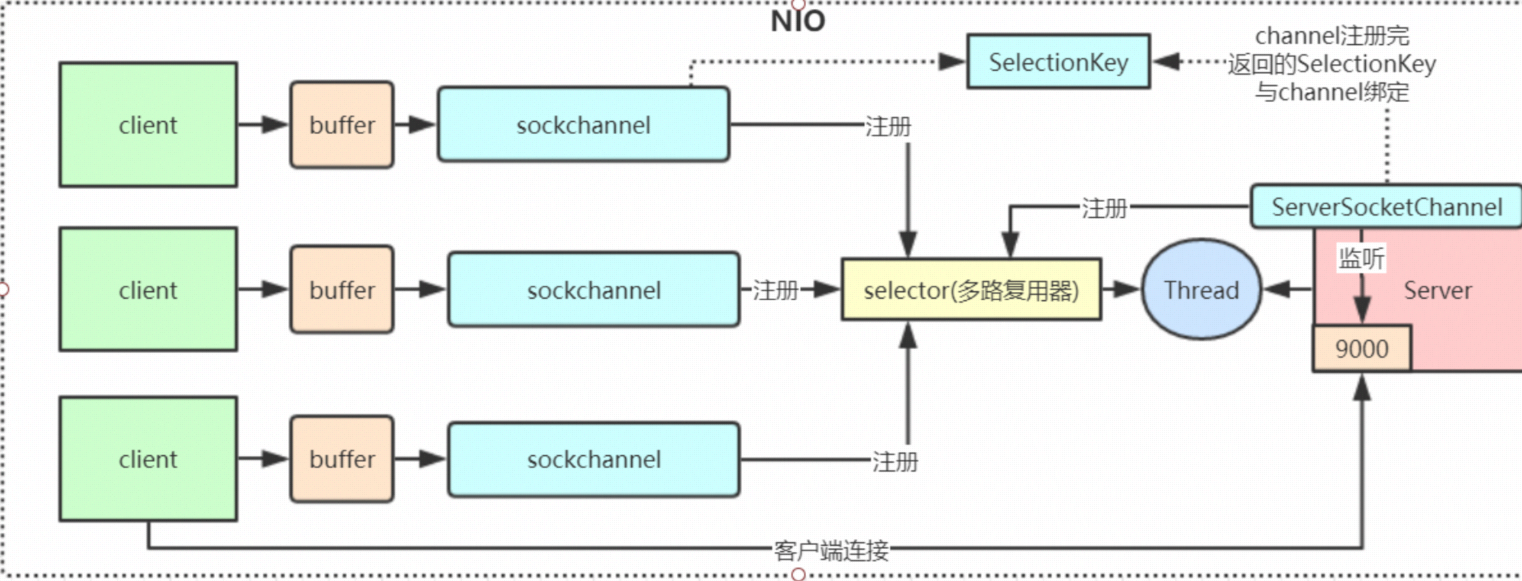
10、用SocketChannel将服务端的数据回写客户端

**总结：**

NIO的selector就像一个大总管，负责监听各种io事件，然后转交给后端线程去处理

NIO相对于BIO的非阻塞就体现在，BIO的后端线程需要阻塞等待客户端写数据（比如read方法），如果客户端不写数据线程就要阻塞，NIO把等待客户端操作的事情交给了大总管selector，selector负责轮询所有已注册的客户端，发现有事件发生了才转交给后端线程处理，后端线程不需要做任何阻塞等待，直接处理客户端的事件数据即可，处理完马上结束或返回线程池供其他客户端事件继续使用，还有就是Channel的读写是非阻塞的

Redis就是典型的NIO线程模型，selector收集所有连接的事件并且转交给后端的线程，线程连续执行所有事件命令并将结果写回客户端



1 //客户端代码  
2 public class NioClient {  
3 //通道管理器  
4 private Selector selector;  
56  
/\*\*  
7 \* 启动客户端测试  
8 \*  
9 \* @throws IOException  
10 \*/  
11 public static void main(String[ ] args) throws IOException {  
12 NioClient client = new NioClient() ;  
13 client. initClient("127. 0. 0. 1", 9000) ;  
14 client. connect() ;  
15 }  
16  
17 /\*\*  
18 \* 获得一个Socket通道， 并对该通道做一些初始化的工作  
19 \*  
20 \* @param ip 连接的服务器的ip  
21 \* @param port 连接的服务器的端口号  
22 \* @throws IOException  
23 \*/  
24 public void initClient(String ip, int port) throws IOException {  
25 // 获得一个Socket通道  
26 SocketChannel channel = SocketChannel. open() ;  
27 // 设置通道为非阻塞  
28 channel. configureBlocking(false) ;  
29 // 获得一个通道管理器  
30 this. selector = Selector. open() ;  
31  
32 // 客户端连接服务器, 其实方法执行并没有实现连接， 需要在listen（） 方法中调  
33 //用channel. finishConnect() ; 才能完成连接  
34 channel. connect(new InetSocketAddress(ip, port) ) ;  
35 //将通道管理器和该通道绑定， 并为该通道注册SelectionKey. OP\_CONNECT事件。  
36 channel. register(selector, SelectionKey. OP\_CONNECT) ;  
37 }  
38  
39 /\*\*  
40 \* 采用轮询的方式监听selector上是否有需要处理的事件， 如果有， 则进行处理  
41 \*  
42 \* @throws IOException  
43 \*/  
44 public void connect() throws IOException {  
45 // 轮询访问selector  
46 while (true) {  
47 // 选择一组可以进行I/O操作的事件， 放在selector中, 客户端的该方法不会阻塞，  
48 //这里和服务端的方法不一样， 查看api注释可以知道， 当至少一个通道被选中时，  
49 //selector的wakeup方法被调用， 方法返回， 而对于客户端来说， 通道一直是被选中的  
50 selector. select() ;  
51 // 获得selector中选中的项的迭代器  
52 Iterator<SelectionKey> it = this. selector. selectedKeys() . iterator() ;  
53 while (it. hasNext() ) {  
54 SelectionKey key = (SelectionKey) it. next() ;  
55 // 删除已选的key, 以防重复处理  
56 it. remove() ;  
57 // 连接事件发生  
58 if (key. isConnectable() ) {  
59 SocketChannel channel = (SocketChannel) key. channel() ;  
60 // 如果正在连接， 则完成连接  
61 if (channel. isConnectionPending() ) {  
62 channel. finishConnect() ;  
63 }  
64 // 设置成非阻塞  
65 channel. configureBlocking(false) ;  
66 //在这里可以给服务端发送信息哦  
67 ByteBuffer buffer = ByteBuffer. wrap("HelloServer". getBytes() ) ;  
68 channel. write(buffer) ;  
69 //在和服务端连接成功之后， 为了可以接收到服务端的信息， 需要给通道设置读的权限。  
70 channel. register(this. selector, SelectionKey. OP\_READ) ; // 获得了可读的事件  
71 } else if (key. isReadable() ) {  
72 read(key) ;  
73 }  
74 }  
75 }  
76 }  
77  
78 /\*\*  
79 \* 处理读取服务端发来的信息 的事件  
80 \*  
81 \* @param key  
82 \* @throws IOException  
83 \*/  
84 public void read(SelectionKey key) throws IOException {  
85 //和服务端的read方法一样  
86 // 服务器可读取消息: 得到事件发生的Socket通道  
87 SocketChannel channel = (SocketChannel) key. channel() ;  
88 // 创建读取的缓冲区  
89 ByteBuffer buffer = ByteBuffer. allocate(512) ;  
90 int len = channel. read(buffer) ;  
91 if (len ! = ‐1) {  
92 System. out. println("客户端收到信息： " + new String(buffer. array() , 0, len) ) ;  
93 }  
94 }  
95 }

### 1.3 AIO（NIO2.0）

异步非阻塞，由操作系统完成后回调通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用

**应用场景：**

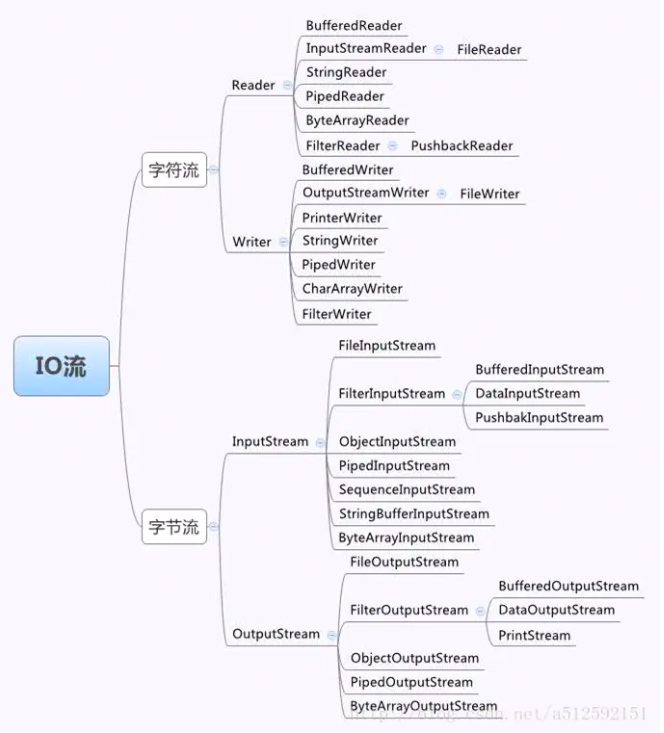
AIO方式适用于连接数目多且连接比较长（重操作）的架构，jdk7开始支持

1 //服务端代码  
2 public class AIOServer {  
3 public static void main(String[ ] args) throws Exception {  
4 final AsynchronousServerSocketChannel serverChannel =  
5 AsynchronousServerSocketChannel. open() . bind(new InetSocketAddress(9000) ) ;  
67  
serverChannel. accept(null, new CompletionHandler<AsynchronousSocketChannel, Object>() {  
8 @Override  
9 public void completed(AsynchronousSocketChannel socketChannel, Object attachment) {  
10 try {  
11 // 再此接收客户端连接， 如果不写这行代码后面的客户端连接连不上服务端  
12 serverChannel. accept(attachment, this) ;  
13 System. out. println(socketChannel. getRemoteAddress() ) ;  
14 ByteBuffer buffer = ByteBuffer. allocate(1024) ;  
15 socketChannel. read(buffer, buffer, new CompletionHandler<Integer, ByteBuffer>() {  
16 @Override  
17 public void completed(Integer result, ByteBuffer buffer) {  
18 buffer. flip() ;  
19 System. out. println(new String(buffer. array() , 0, result) ) ;  
20 socketChannel. write(ByteBuffer. wrap("HelloClient". getBytes() ) ) ;  
21 }  
22  
23 @Override  
24 public void failed(Throwable exc, ByteBuffer buffer) {  
25 exc. printStackTrace() ;  
26 }  
27 }) ;  
28 } catch (IOException e) {  
29 e. printStackTrace() ;  
30 }  
31 }  
32  
33 @Override  
34 public void failed(Throwable exc, Object attachment) {  
35 exc. printStackTrace() ;  
36 }  
37 }) ;  
38  
39 Thread. sleep(Integer. MAX\_VALUE) ;  
40 }  
41 }  
1 //客户端代码  
2 public class AIOClient {  
34  
public static void main(String. . . args) throws Exception {  
5 AsynchronousSocketChannel socketChannel = AsynchronousSocketChannel. open() ;  
6 socketChannel. connect(new InetSocketAddress("127. 0. 0. 1", 9000) ) . get() ;  
7 socketChannel. write(ByteBuffer. wrap("HelloServer". getBytes() ) ) ;  
8 ByteBuffer buffer = ByteBuffer. allocate(512) ;  
9 Integer len = socketChannel. read(buffer) . get() ;  
10 if (len ! = ‐1) {  
11 System. out. println("客户端收到信息： " + new String(buffer. array() , 0, len) ) ;  
12 }  
13 }  
14 }



**同步异步与阻塞非阻塞(段子)**老张爱喝茶， 废话不说， 煮开水。  
出场人物： 老张， 水壶两把（普通水壶， 简称水壶； 会响的水壶， 简称响水壶） 。  
1 老张把水壶放到火上， 立等水开。 （同步阻塞）  
老张觉得自己有点傻  
2 老张把水壶放到火上， 去客厅看电视， 时不时去厨房看看水开没有。 （同步非阻塞）  
老张还是觉得自己有点傻， 于是变高端了， 买了把会响笛的那种水壶。 水开之后， 能大声发出嘀~~~~的噪音。  
3 老张把响水壶放到火上， 立等水开。 （异步阻塞）  
老张觉得这样傻等意义不大  
4 老张把响水壶放到火上， 去客厅看电视， 水壶响之前不再去看它了， 响了再去拿壶。 （异步非阻塞）  
老张觉得自己聪明了。  
所谓同步异步， 只是对于水壶而言。  
普通水壶， 同步； 响水壶， 异步。  
虽然都能干活， 但响水壶可以在自己完工之后， 提示老张水开了。 这是普通水壶所不能及的。  
同步只能让调用者去轮询自己（情况2中） ， 造成老张效率的低下。

## 2、Java io流



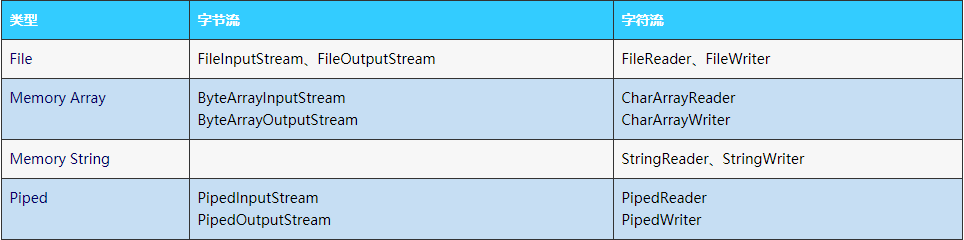
Java io流根据流向分类可分为输入流、输出流，按处理数据不同分为字节流、字符流，按照功能不同分类分为节点流、处理流

字节流：二进制，可处理一切文件，包括纯文本、doc、音频、视频等

字符流：文本文件，只能处理纯文本

节点流：可以从或向一个特定的地方（节点）读写数据

* 文件 FileInputStream FileOutputStrean FileReader FileWriter 文件进行处理的节点流。
* 字符串 StringReader StringWriter 对字符串进行处理的节点流。
* 数组 ByteArrayInputStream ByteArrayOutputStream CharArrayReader CharArrayWriter 对数组进行处理的节点流（对应的不再是文件，而是内存中的一个数组）。
* 管道 PipedInputStream PipedOutputStream PipedReaderPipedWriter对管道进行处理的节点流。



处理流：

* 缓冲流：BufferedInputStrean BufferedOutputStream BufferedReader BufferedWriter---增加缓冲功能，避免频繁读写硬盘。
* 转换流：InputStreamReader OutputStreamReader实现字节流和字符流之间的转换。
* 数据流 DataInputStream DataOutputStream 等-提供将基础数据类型写入到文件中，或者读取出来.

FileInputStream/FileOutputStream 需要逐个字节处理原始二进制流的时候使用，效率低下。  
FileReader/FileWriter 需要组个字符处理的时候使用。  
StringReader/StringWriter 需要处理字符串的时候，可以将字符串保存为字符数组。  
PrintStream/PrintWriter 用来包装FileOutputStream 对象，方便直接将String字符串写入文件 。  
Scanner　用来包装System.in流，很方便地将输入的String字符串转换成需要的数据类型。  
InputStreamReader/OutputStreamReader , 字节和字符的转换桥梁，在网络通信或者处理键盘输入的时候用。  
BufferedReader/BufferedWriter， BufferedInputStream/BufferedOutputStream， 缓冲流用来包装字节流后者字符流，提升IO性能，BufferedReader还可以方便地读取一行，简化编程。  
SequenceInputStream(InputStream s1, InputStream s2)序列流，合并流对象时使用.  
ObjectInputStream、ObjectOutputStream，方法用于序列化对象并将它们写入一个流，另一个方法用于读取流并反序列化对象。  
ByteArrayInputStream、ByteArrayOutputStream，操作数组  
DataInputStream、DataOutputStream操作基本数据类型和字符串

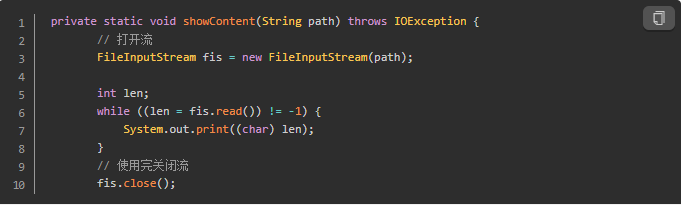
### 2.1 字节流

InputStream有read方法，一次读取一个字节，OutputStream的write方法一次写一个int。这两个类都是抽象类。意味着不能创建对象，那么需要找到具体的子类来使用。操作流的步骤都是：

第一步：1：打开流（即创建流）  
第二步：2：通过流读取内容  
第三步：3：用完后，关闭流资源

#### 1、inputstream

1、使用read方法一次读取一个字节



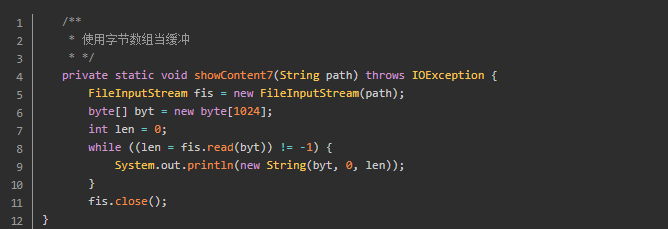
2、使用read方法，可以将读到的数据装入到字节数组中，一次性的操作数组，可以提高效率。



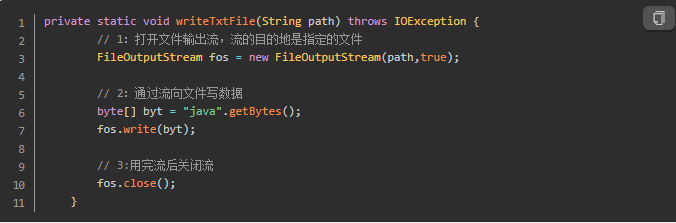
3、使用read(byte[] b,int off,int len)



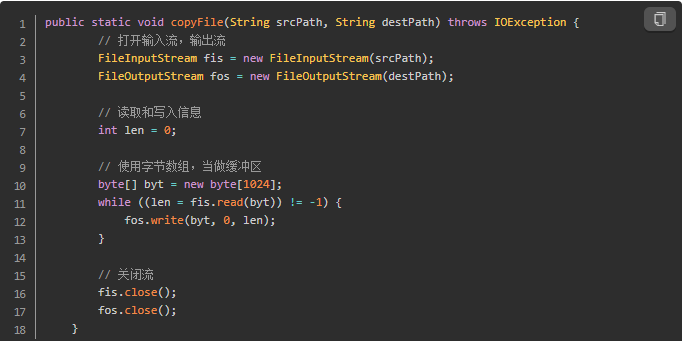
4、使用缓冲(提高效率),并循环读取(读完所有内容)



#### 2、outputStream



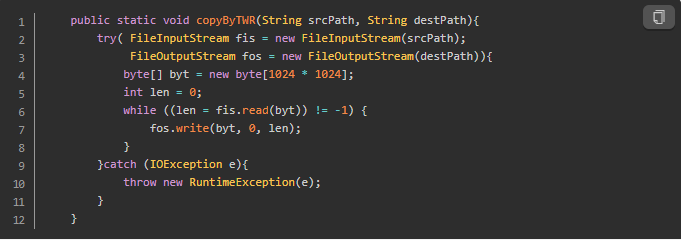
输入输出综合使用



可以根据拷贝的需求调整数组的大小，一般是1024的整数倍。使用缓冲后效率大大提高。目前我们是抛出处理，一旦出现了异常，close就没有执行，也就没有释放资源。那么为了保证close的执行该如何处理呢。那么就需要使用try{} catch(){}finally{}语句。try中放入可能出现异常的语句，catch是捕获异常对象，fianlly是一定要执行的代码。



大量的异常捕获代码使得以上代码变得十分臃肿难看，好在java7提供了TWR（try-with-resource）语法糖，由于很多外部资源类都间接的实现了AutoCloseable接口（单方法回调接口），因此可以利用TWR语法在try结束的时候通过回调的方式自动调用外部资源类的close()方法，避免书写冗长的finally代码块。



当try语句块运行结束时，FileInputStream / FileOutputStream会被自动关闭。这是因为FileInputStream实现了java中的java.lang.AutoCloseable接口。所有实现了这个接口的类都可以在try-with-resources结构中使用上面的例子在try关键字后的括号里创建了两个资源——FileInputStream 和FileOutputStream。当程序运行离开try语句块时，这两个资源都会被自动关闭,这些资源将按照他们被创建顺序的逆序来关闭。首先FileOutputStream会被关闭，然后FileInputStream会被关闭。  
怎么实现对资源的关闭呢？当try-with-resources结构中抛出一个异常，同时FileInputStream被关闭时（调用了其close方法）也抛出一个异常，try-with-resources结构中抛出的异常会向外传播，而FileInputStream被关闭时抛出的异常被抑制了

Try with resource 详解

<http://ifeve.com/java-7%E4%B8%AD%E7%9A%84try-with-resources/>

#### 3、缓冲流

Java其实提供了专门的字节流缓冲来提高效率。BufferedInputStream 和 BufferedOutputStream。BufferedOutputStream和BufferedOutputStream类可以通过减少读写次数来提高输入和输出的速度。它们内部有一个缓冲区，用来提高处理效率。查看API文档，发现可以指定缓冲区的大小。其实内部也是封装了字节数组。没有指定缓冲区大小，默认的字节是8192。显然缓冲区输入流和缓冲区输出流要配合使用。首先缓冲区输入流会将读取到的数据读入缓冲区，当缓冲区满时，或者调用flush方法，缓冲输出流会将数据写出。



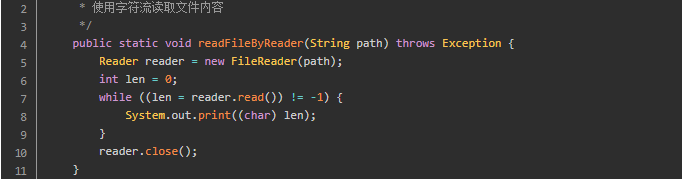
### 2.2 字符流

计算机并不区分二进制文件与文本文件。所有的文件都是以二进制形式来存储的，因此，从本质上说，所有的文件都是二进制文件。所以字符流是建立在字节流之上的，它能够提供字符层次的编码和解码。可以说字符流就是：字节流 + 编码表，为了更便于操作文字数据。字符流的抽象基类：Reader ， Writer。由这些类派生出来的子类名称都是以其父类名作为子类名的后缀，如FileReader、FileWriter。

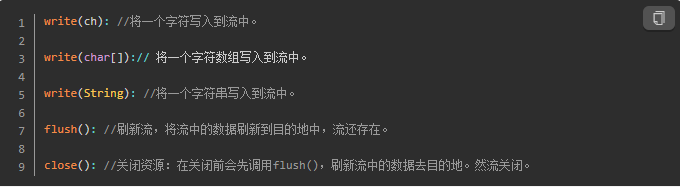
Read方法的几种重载

Read（）：读取一个字符。返回的是读到的那个字符。如果读到流的末尾，返回-1

Read(char[]):将读到的字符存入指定的数组中，返回的是读到的字符个数，也就是往数组里装的元素的个数。如果读到流的末尾，返回-1

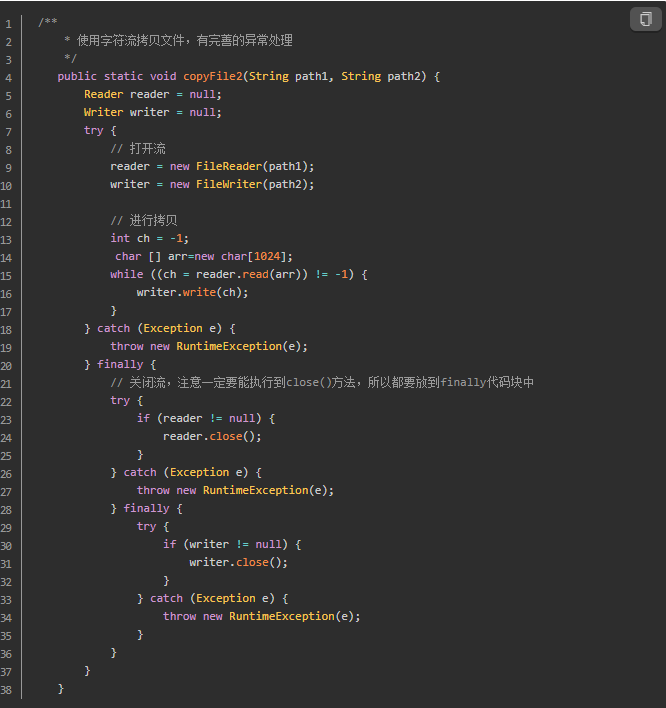


Write



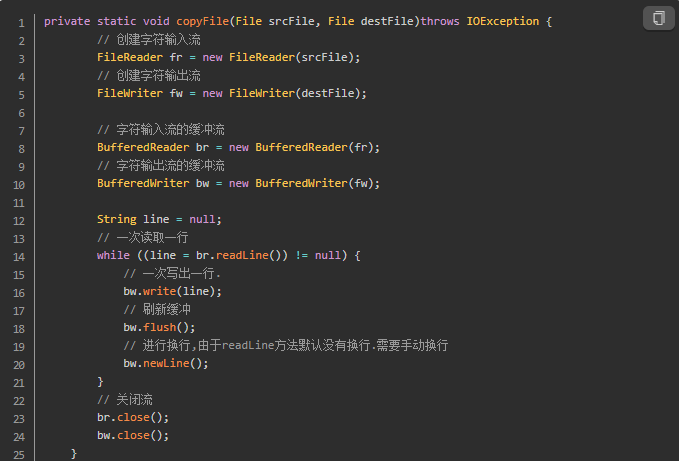
基本方法和OutputStream 类似，有write方法，功能更多一些。可以接收字符串。Writer是抽象类无法创建对象，Writer的实现子类为FileWriter。默认的FileWriter方法新值会覆盖旧值，想要实现追加功能需要使用如下构造函数创建输出流 append值为true了。





#### 缓冲流：

使用字符流缓冲区拷贝文本文件可以提高效率，Reader有一个子类BufferedReader， 子类继承父类显然子类可以重写父类的方法可以增加自己的新方法。例如一次读一行就是常用的操作.那么BufferedReader类就提供了这个方法,可以查看readLine()方法具备 一次读取一个文本行的功能。很显然,该子类可以对功能进行增强。



# 2、netty

NIO的类库和API复杂，使用麻烦：需要熟练掌握selector、ServerSocketChannel、SocketChannel、ByteBuffer等

开发工作量和难度都非常大：例如客户端面临断连重连、网络闪断、心跳处理、半包读写、网络拥塞和异常流的处理等等

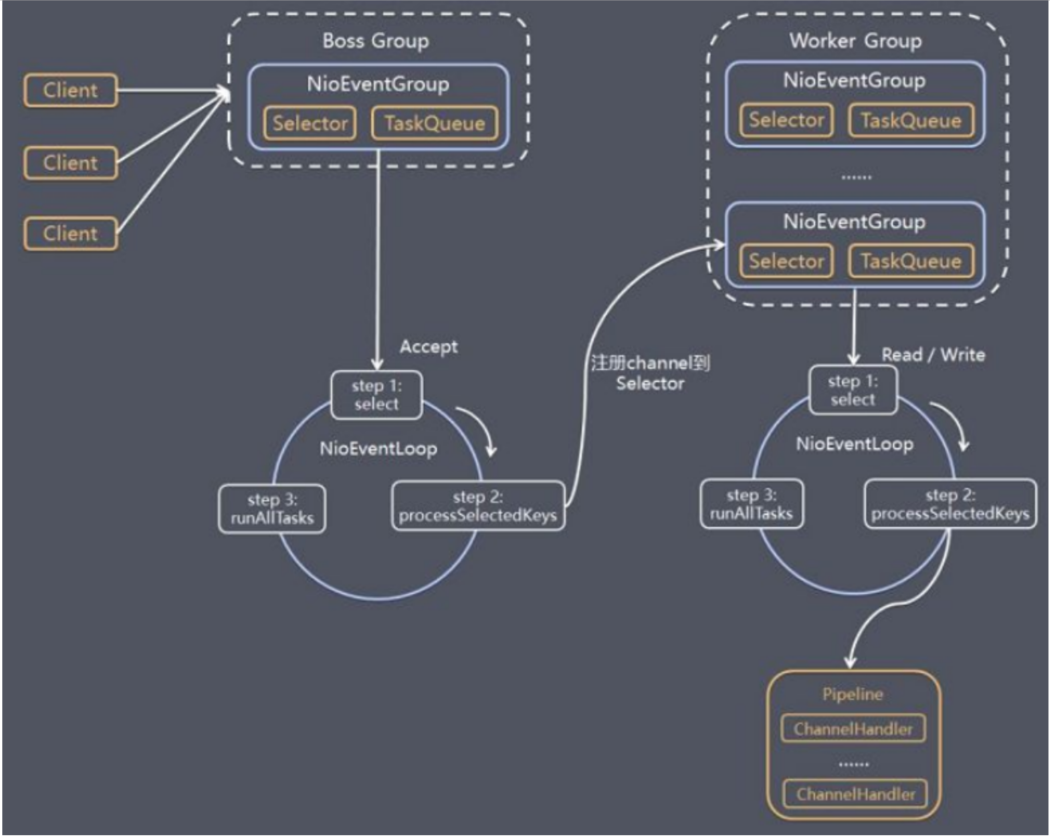
Netty对jdk自带的NIO的API进行了良好的封装，解决了上述问题。且netty拥有高性能、吞吐量更高，延迟更低，减少资源消耗，最小化不必要的内存复制等优点

**Netty的使用场景：**

1） 互联网行业： 在分布式系统中， 各个节点之间需要远程服务调用， 高性能的 RPC 框架必不可少，  
Netty 作为异步高性能的通信框架， 往往作为基础通信组件被这些 RPC 框架使用。 典型的应用有： 阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信， Dubbo 协议默认使用Netty 作为基础通信组件， 用于实现。 各进程节点之间的内部通信。 Rocketmq底层也是用的Netty作为基础通信组件。  
2） 游戏行业： 无论是手游服务端还是大型的网络游戏， Java 语言得到了越来越广泛的应用。 Netty作为高性能的基础通信组件， 它本身提供了 TCP/UDP 和 HTTP 协议栈。  
3） 大数据领域： 经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架， 默认采用 Netty进行跨界点通信， 它的 Netty Service 基于 Netty 框架二次封装实现。  
netty相关开源项目 ：https://netty.io/wiki/related-projects.html

## 2.1 netty的组件与模型

### 2.1.1 netty的线程模型



**模型解释：**

1、netty抽象出两组线程池BossGroup和WorkerGroup，BossGroup专门负责接收客户端的连接，WorkerGroup专门负责网络的读写

2、BossGroup与workerGroup都是NIOEventLoopGroup

3、NioEventLoopGroup相当于一个事件循环线程组，这个组中含有多个事件循环线程，每一个事件循环线程是NIOEventLoop

4、每个NIOEventLoop都有一个selector，用于监听注册在其上的SocketChannel的网络通讯

5、每个boss NioEventLoop线程内部循环执行的步骤有三步

·处理accept事件，与client建立连接，生成NioSocketChannel

·将NioSocketChannel注册到某个worker NioEventLoop上的 selector

·处理任务队列的任务，即runAllTasks

6、每个worker NioEventLoop线程循环执行的步骤

·轮询注册到自己selector上的所有NioSocketChannel的read、write事件

·处理I/O事件，即read、write事件，在对应的NioSocketChannel处理业务

·runAllTasks处理任务队列TaskQueue的任务，一些耗时的业务处理一般可以放入TaskQueue中慢慢处理，这样不影响数据在pipeline中流动处理

7、每个worker NioEventLoop处理NioSocketChannel业务时，会使用pipeline管道，管道中维护了很多 handler处理器用来来处理channel中的数据

### 2.1.2 netty的模块组件

1、BootStrap、ServerBootStrap

Bootstrap意思是引导，一个netty应用通常由一个BootStrap开始，主要作用是配置整个Netty程序，串联各个组件，Netty中的BootStrap类是客户端的启动引导类，serverBootstrap是服务端的启动引导类

2、future、channelFuture

在netty中所有的操作都是异步的，不能立刻得知消息是否被正确处理，但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过future和channelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册和监听事件

3、Channel

Netty的网络通讯组件，能够用于执行网络I/O操作，channel为用户提供：

·当前网络连接的通道状态（例如是否打开，是否已连接）

·网络连接的配置参数（例如接收缓冲区的大小）

·提供异步的网络I/O操作，（例如建立连接，读写，绑定端口），异步调用意味着任何I/O操作都将立即返回，并且不保证在调用结束时所请求的I/O操作已完成

·调用立即返回一个channelFuture实例，通过注册监听器到ChannelFuture上，可以I/O操作成功、失败或取消时回调通知调用方

·支持关联I/O操作与对应的处理程序

不同的协议，不同的阻塞类型的连接都有不同的channel类型与之对应

**NioSocketChannel** 异步的客户端 TCP socket连接

**NioServerSocketChannel** 异步的服务端TCP socket连接

**NioDatagramChannennlel** 异步的UDP连接

**NioSctpChannel** 异步的客户端Sctp连接

**NioSctpServerChannel** 异步的Sctp服务端连接，这些通道涵盖了UDP和TCP网络IO以及文件IO

4、selector

Netty基于selector对象实现I/O多路复用，通过selector一个线程可以监听多个连接的channel事件

当一个selector中注册channel后，selector内部的机制就可以自动不断地查询（select）这些注册的channel是否有已就绪的I/O事件（例如可读，可写，网络连接完成等），这样程序就可以很简单的使用一个线程高效的管理多个channel了

5、NioEventLoop

NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用

NioEventLoop中的run方法，执行I/O任务和非I/O任务

**IO任务**：即selectionKey中ready的事件，如accept、connect、read、write等，由

processSelectedKeys方法触发

**非IO任务**：添加到TaskQueue中的任务，如register0、bind0等任务，由runAllTasks方法触发

6、NioEventLoopGroup

NioEventLoopGroup主要管理eventLoop的生命周期，可以理解为一个线程池，内部维护了一组线程，每个线程NioEventLoop负责处理多个channel上的事件，而一个channel只对应一个线程

7、ChannelHandler

Channelhandler是一个接口，处理IO事件或拦截IO操作，并将其转发到channelPipeline(业务处理链)中的下一个处理程序

ChannelHandler本身并没有提供很多方法，因为这个接口有许多方法需要实现，方便使用期间可以继承它的子类

ChannelInboundHandler 用于处理入站IO事件

ChannelOutboundHandler 用于处理出站IO操作

或者使用一下啊适配器类

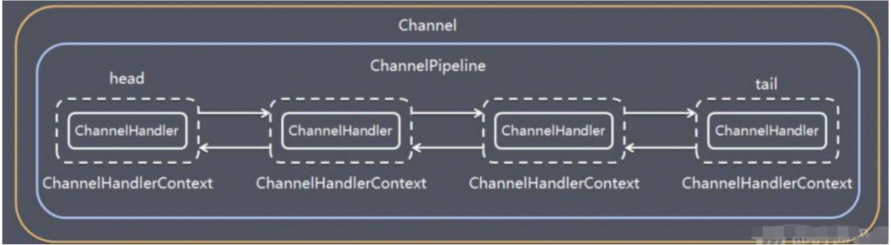
ChannelInboundHandlerAdapter 用于处理入站IO事件

ChannelOutboundHandlerAdapter 用于处理出站的IO事件

8、ChannelPipeline

保存handler的list，用于处理或拦截Channel的入站事件和出站操作，ChannelPipeline实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及Channel中各个channelHandler的如何相互交互

在netty中每个channel都有且仅有一个ChannelPipeline与之对应，它的组成关系如下：



一个channel包含了一个ChannelPipeline，而ChannelPipeline中 又维护了一个由channelHandlerContext组成的双向链表，并且每个channelHandlerContext中国又关联这一个channelHandler

Read事件（入站事件）和write（出站事件）在一个双向链表中，入站事件会从链表head往后传递到最后一个入站的handler，出站事件会从链表tail往前传递到最前一个出站的handler，两种类型的handler互不影响，**即头尾一般为编解码器，后面是自定义的handler处理器**

### 2.1.3 netty简单demo

public class NettyServer {  
23  
public static void main(String[ ] args) throws Exception {  
45  
//创建两个线程组bossGroup和workerGroup, 含有的子线程NioEventLoop的个数默认为cpu核数的两  
倍6  
// bossGroup只是处理连接请求 , 真正的和客户端业务处理， 会交给workerGroup完成  
7 EventLoopGroup bossGroup = new NioEventLoopGroup(1) ;  
8 EventLoopGroup workerGroup = new NioEventLoopGroup() ;  
9 try {  
10 //创建服务器端的启动对象  
11 ServerBootstrap bootstrap = new ServerBootstrap() ;  
12 //使用链式编程来配置参数  
13 bootstrap. group(bossGroup, workerGroup) //设置两个线程组  
14 . channel(NioServerSocketChannel. class) //使用NioServerSocketChannel作为服务器的通道  
实现  
15 // 初始化服务器连接队列大小， 服务端处理客户端连接请求是顺序处理的, 所以同一时间只能处理一个客户端连接。  
16 // 多个客户端同时来的时候, 服务端将不能处理的客户端连接请求放在队列中等待处理  
17 . option(ChannelOption. SO\_BACKLOG, 1024)  
18 . childHandler(new ChannelInitializer<SocketChannel>() {//创建通道初始化对象， 设置初始化参数  
19  
20 @Override  
21 protected void initChannel(SocketChannel ch) throws Exception {  
22 //对workerGroup的SocketChannel设置处理器codec  
23 ch. pipeline() . addLast(new NettyServerHandler() ) ;  
24 }  
25 }) ;  
26 System. out. println("netty server start。 。 ") ;  
27 //绑定一个端口并且同步, 生成了一个ChannelFuture异步对象， 通过isDone()等方法可以判断异步  
事件的执行情况  
28 //启动服务器(并绑定端口 )， bind是异步操作， sync方法是等待异步操作执行完毕  
29 ChannelFuture cf = bootstrap. bind(9000) . sync() ;  
30 //给cf注册监听器， 监听我们关心的事件  
31 /\*cf. addListener(new ChannelFutureListener() {  
32 @Override  
33 public void operationComplete(ChannelFuture future) throws Exception {  
34 if (cf. isSuccess() ) {  
35 System. out. println("监听端口 9000成功") ;  
36 } else {  
37 System. out. println("监听端口 9000失败") ;  
38 }  
39 }  
40 }) ; \*/  
41 //对通道关闭进行监听， closeFuture是异步操作， 监听通道关闭  
42 // 通过sync方法同步等待通道关闭处理完毕， 这里会阻塞等待通道关闭完成  
43 cf. channel() . closeFuture() . sync() ;  
44 } finally {  
45 bossGroup. shutdownGracefully() ;  
46 workerGroup. shutdownGracefully() ;  
47 }  
48 }  
49 }  
50  
51  
52 /\*\*  
53 \* 自 定义Handler需要继承netty规定好的某个HandlerAdapter(规范)  
54 \*/  
55 public class NettyServerHandler extends ChannelInboundHandlerAdapter {  
56  
57 /\*\*  
58 \* 读取客户端发送的数据  
59 \*  
60 \* @param ctx 上下文对象, 含有通道channel， 管道pipeline  
61 \* @param msg 就是客户端发送的数据  
62 \* @throws Exception  
63 \*/  
64 @Override  
65 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
66 System. out. println("服务器读取线程 " + Thread. currentThread() . getName() ) ;  
67 //Channel channel = ctx. channel() ;  
68 //ChannelPipeline pipeline = ctx. pipeline() ; //本质是一个双向链接, 出站入站  
69 //将 msg 转成一个 ByteBuf， 类似NIO 的 ByteBuffer  
70 ByteBuf buf = (ByteBuf) msg;  
71 System. out. println("客户端发送消息是: " + buf. toString(CharsetUtil. UTF\_8) ) ;  
72 }  
73  
74 /\*\*  
75 \* 数据读取完毕处理方法  
76 \*  
77 \* @param ctx  
78 \* @throws Exception  
79 \*/  
80 @Override  
81 public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  
82 ByteBuf buf = Unpooled. copiedBuffer("HelloClient", CharsetUtil. UTF\_8) ;  
83 ctx. writeAndFlush(buf) ;  
84 }  
85  
86 /\*\*  
87 \* 处理异常, 一般是需要关闭通道  
88 \*  
89 \* @param ctx  
90 \* @param cause  
91 \* @throws Exception  
92 \*/  
93 @Override  
94 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exc  
eption {  
95 ctx. close() ;  
96 }  
97 }

客户端代码：  
1 public class NettyClient {  
2 public static void main(String[ ] args) throws Exception {  
3 //客户端需要一个事件循环组  
4 EventLoopGroup group = new NioEventLoopGroup() ;  
5 try {  
6 //创建客户端启动对象  
7 //注意客户端使用的不是 ServerBootstrap 而是 Bootstrap  
8 Bootstrap bootstrap = new Bootstrap() ;  
9 //设置相关参数  
10 bootstrap. group(group) //设置线程组  
11 . channel(NioSocketChannel. class) // 使用 NioSocketChannel 作为客户端的通道实现  
12 . handler(new ChannelInitializer<SocketChannel>() {  
13 @Override  
14 protected void initChannel(SocketChannel channel) throws Exception {  
15 //加入处理器  
16 channel. pipeline() . addLast(new NettyClientHandler() ) ;  
17 }  
18 }) ;  
19 System. out. println("netty client start") ;  
20 //启动客户端去连接服务器端  
21 ChannelFuture channelFuture = bootstrap. connect("127. 0. 0. 1", 9000) . sync() ;  
22 //对关闭通道进行监听  
23 channelFuture. channel() . closeFuture() . sync() ;  
24 } finally {  
25 group. shutdownGracefully() ;  
26 }  
27 }  
28 }  
29  
30 public class NettyClientHandler extends ChannelInboundHandlerAdapter {  
31  
32 /\*\*  
33 \* 当客户端连接服务器完成就会触发该方法  
34 \*  
35 \* @param ctx  
36 \* @throws Exception  
37 \*/  
38 @Override  
39 public void channelActive(ChannelHandlerContext ctx) throws Exception {  
40 ByteBuf buf = Unpooled. copiedBuffer("HelloServer", CharsetUtil. UTF\_8) ;  
41 ctx. writeAndFlush(buf) ;  
42 }  
43  
44 //当通道有读取事件时会触发， 即服务端发送数据给客户端  
45 @Override  
46 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
47 ByteBuf buf = (ByteBuf) msg;  
48 System. out. println("收到服务端的消息: " + buf. toString(CharsetUtil. UTF\_8) ) ;  
49 System. out. println("服务端的地址： " + ctx. channel() . remoteAddress() ) ;  
50 }  
51  
52 @Override  
53 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
54 cause. printStackTrace() ;  
55 ctx. close() ;  
56 }  
57 }

### 2.1.5 netty的编解码

Netty涉及到编解码的组件有channel、channelHandler、ChannelPipeline等

**channelHandler**充当了处理入站和出站数据的应用程序逻辑容器，例如

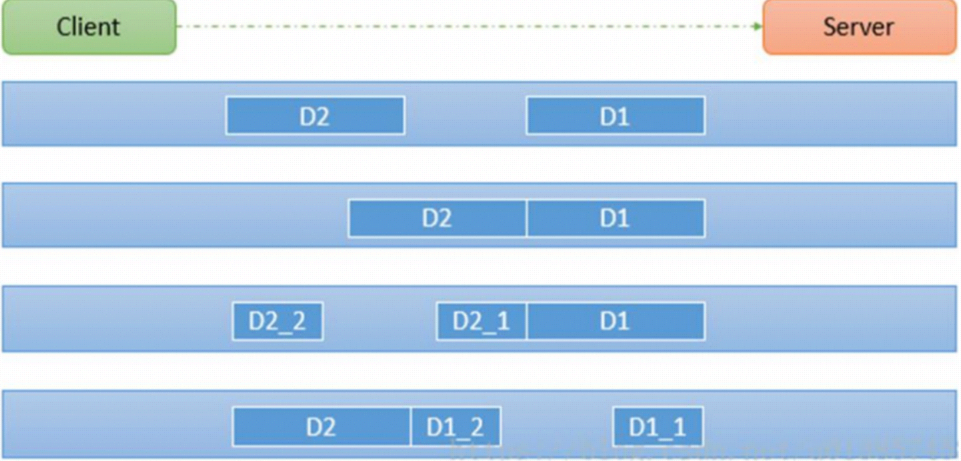
ChannelInboundHandler接口或者ChannelInboundHandlerAdapter，就可以接收入站事件和数据，这些数据随后被应用程序的业务逻辑处理，当要给连接的客户端发送响应时，也可以从ChannelInboundHandler冲刷数据，通常业务逻辑写在一个或者多个ChannelInboundHandler中，ChannelOutboundHandler原理相同，用来处理出站事件

ChannelPipeline提供了channelHandler的链容器以客户端应用程序为例， 如果事件的运动方向是从客户端到服务端的， 那么我们称这些事件为出站的， 即客户端发送给服务端的数据会通过pipeline中的一系列ChannelOutboundHandler(ChannelOutboundHandler调用是从tail到head方向逐个调用每个handler的逻辑)， 并被这些Handler处理， 反之则称为入站的， 入站只调用pipeline里的ChannelInboundHandler逻辑(ChannelInboundHandler调用是从head到tail方向逐个调用每个handler的逻辑)

**编解码器：**

当你通过Netty发送或者接受一个消息的时候， 就将会发生一次数据转换。 入站消息会被解码（decoder）：从字节转换为另一种格式（比如java对象） ； 如果是出站消息，它会被编码成字节（encoder）。Netty提供了一系列实用的编码解码器， 他们都实现了ChannelInboundHadnler或者ChannelOutcoundHandler接口。 在这些类中，channelRead方法已经被重写了。 以入站为例， 对于每个从入站Channel读取的消息， 这个方法会被调用。 随后， 它将调用由已知解码器所提供的decode()方法进行解码， 并将已经解码的字节转发给ChannelPipeline中的下一个ChannelInboundHandler。Netty提供了很多编解码器， 比如编解码字符串的StringEncoder和StringDecoder， 编解码对象的ObjectEncoder和ObjectDecoder等。

### 2.1.6 netty的拆包粘包

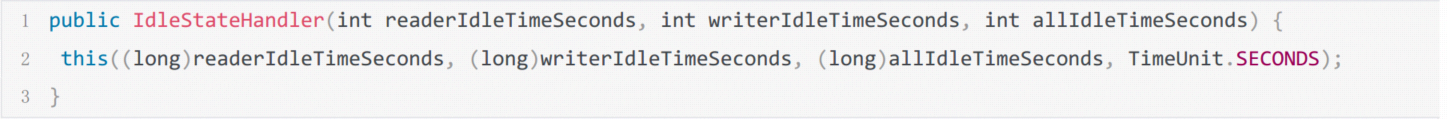


TCP 是面向连接的， 面向流的， 提供高可靠性服务。 收发两端（客户端和服务器端） 都要有成对的 socket， 因此， 发送端为了将多个发给接收端的包， 更有效的发给对方， 使用了优化方法（Nagle 算法） ， 将多次间隔较小且数据量小的数据， 合并成一个大的数据块，然后进行封包。 这样做虽然提高了效率， 但是接收端就难于分辨出完整的数据包了， 因为面向流的通信是无消息保护边界的。

解决方案  
1 ） 格式化数据： 每条数据有固定的格式（开始符、 结束符） ， 这种方法简单易行， 但选择开始符和结束符的时候一定要注意每条数据的内部一定不能出现开始符或结束符。  
2） 发送长度： 发送每条数据的时候， 将数据的长度一并发送， 比如可以选择每条数据的前4位是数据的长度， 应用层处理时可以根据长度来判断每条数据的开始和结束。  
第二种方案更稳妥， 参见项目示例com.tuling.netty.split包下代码

### 2.1.7 netty心跳检测机制

所谓心跳, 即在 TCP 长连接中, 客户端和服务器之间定期发送的一种特殊的数据包, 通知对方自己还在线, 以确保 TCP 连接的有效性.在 Netty 中, 实现心跳机制的关键是 IdleStateHandler, 看下它的构造器：



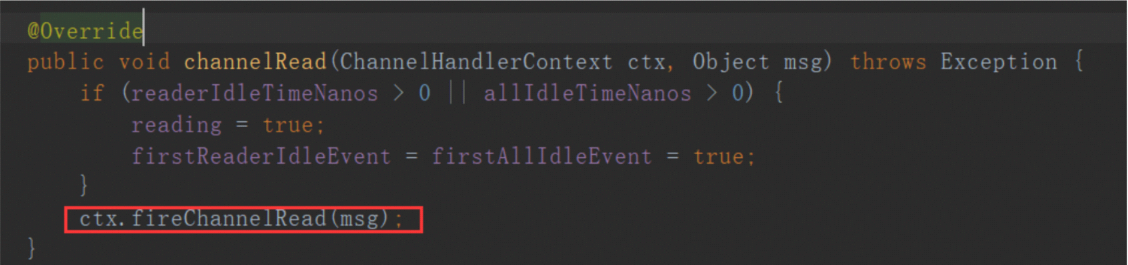
这里解释下三个参数的含义：  
readerIdleTimeSeconds: 读超时. 即当在指定的时间间隔内没有从 Channel 读取到数据时, 会触发一个 READER\_IDLE 的IdleStateEvent 事件.  
writerIdleTimeSeconds: 写超时. 即当在指定的时间间隔内没有数据写入到 Channel 时, 会触发一个 WRITER\_IDLE 的IdleStateEvent 事件.  
allIdleTimeSeconds: 读/写超时. 即当在指定的时间间隔内没有读或写操作时, 会触发一个 ALL\_IDLE 的 IdleStateEvent 事件.  
注：这三个参数默认的时间单位是秒。若需要指定其他时间单位，可以使用另一个构造方法：



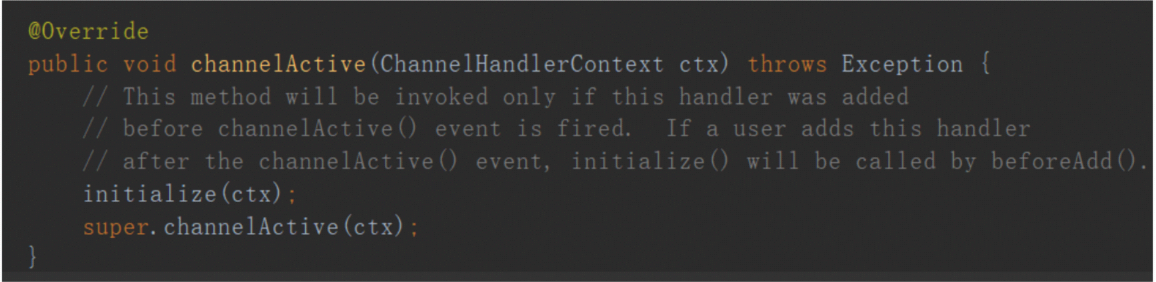
要实现Netty服务端心跳检测机制需要在服务器端的ChannelInitializer中加入如下的代码：



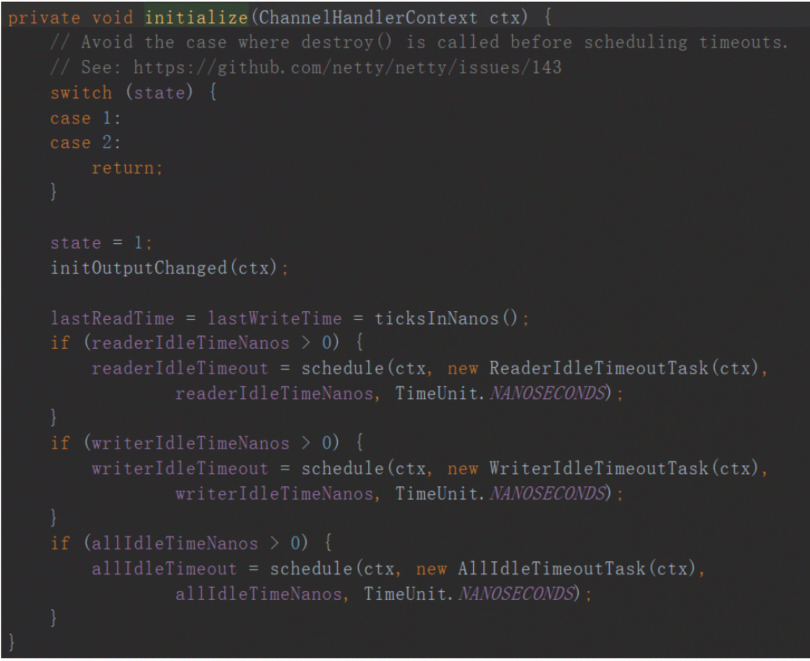
初步地看下IdleStateHandler源码， 先看下IdleStateHandler中的channelRead方法：



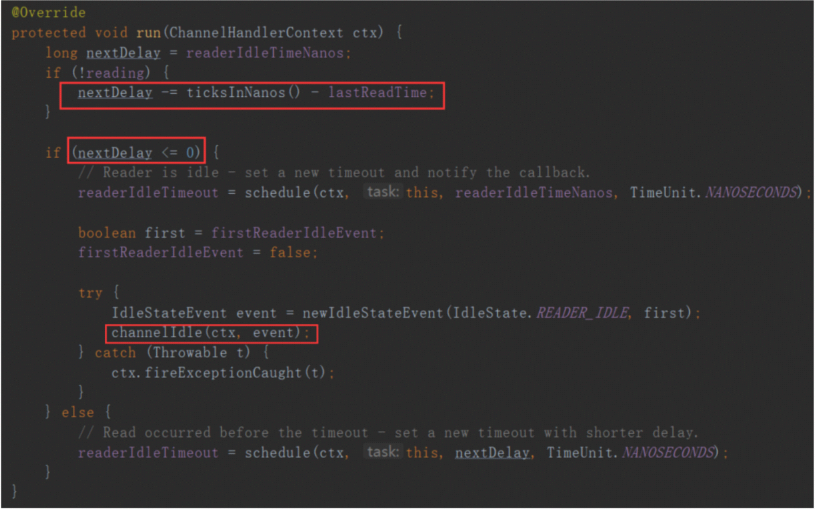
红框代码其实表示该方法只是进行了透传， 不做任何业务逻辑处理， 让channelPipe中的下一个handler处理channelRead方法我们再看看channelActive方法：



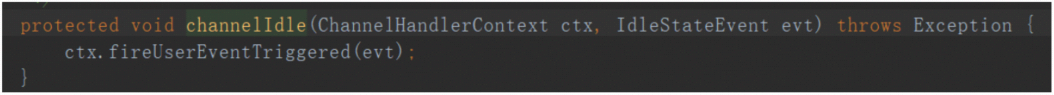
这里有个initialize的方法， 这是IdleStateHandler的精髓， 接着探究：



这边会触发一个Task， ReaderIdleTimeoutTask， 这个task里的run方法源码是这样的：



第一个红框代码是用当前时间减去最后一次channelRead方法调用的时间，假如这个结果是6s， 说明最后一次调用channelRead已经是6s之前的事情了，你设置的是5s，那么nextDelay则为-1，说明超时了，那么第二个红框代码则会触发下一个handler的userEventTriggered方法：

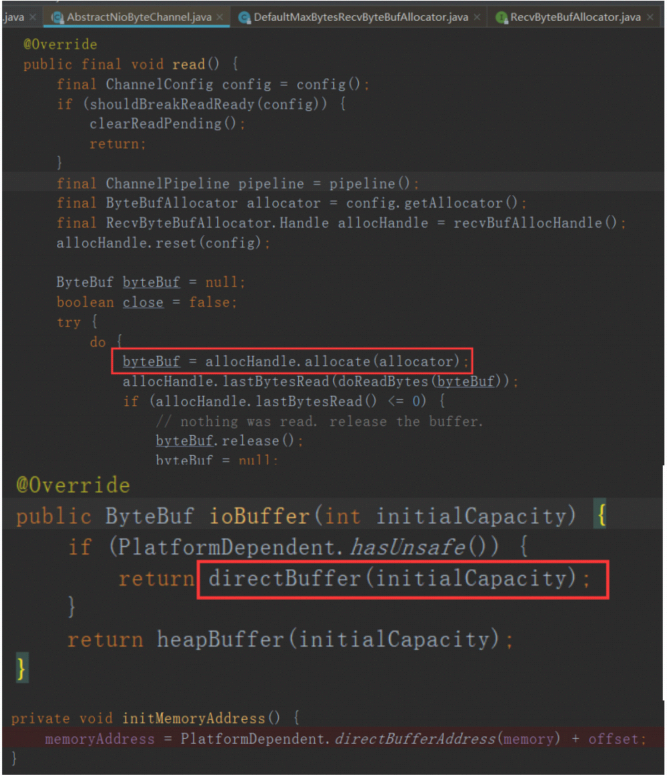


如果没有超时则不触发userEventTriggered方法

1 //服务端代码  
2 public class HeartBeatServer {  
34  
public static void main(String[ ] args) throws Exception {  
5 EventLoopGroup boss = new NioEventLoopGroup() ;  
6 EventLoopGroup worker = new NioEventLoopGroup() ;  
7 try {  
8 ServerBootstrap bootstrap = new ServerBootstrap() ;  
9 bootstrap. group(boss, worker)  
10 . channel(NioServerSocketChannel. class)  
11 . childHandler(new ChannelInitializer<SocketChannel>() {  
12 @Override  
13 protected void initChannel(SocketChannel ch) throws Exception {  
14 ChannelPipeline pipeline = ch. pipeline() ;  
15 pipeline. addLast("decoder", new StringDecoder() ) ;  
16 pipeline. addLast("encoder", new StringEncoder() ) ;  
17 //IdleStateHandler的readerIdleTime参数指定超过3秒还没收到客户端的连接，  
18 //会触发IdleStateEvent事件并且交给下一个handler处理， 下一个handler必须  
19 //实现userEventTriggered方法处理对应事件  
20 pipeline. addLast(new IdleStateHandler(3, 0, 0, TimeUnit. SECONDS) ) ;  
21 pipeline. addLast(new HeartBeatHandler() ) ;  
22 }  
23 }) ;  
24 System. out. println("netty server start。 。 ") ;  
25 ChannelFuture future = bootstrap. bind(9000) . sync() ;  
26 future. channel() . closeFuture() . sync() ;  
27 } catch (Exception e) {  
28 e. printStackTrace() ;  
29 } finally {  
30 worker. shutdownGracefully() ;  
31 boss. shutdownGracefully() ;  
32 }  
33 }  
34 }  
35  
1 //服务端处理handler  
2 public class HeartBeatServerHandler extends SimpleChannelInboundHandler<String> {  
34  
int readIdleTimes = 0;  
56  
@Override  
7 protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {  
8 System. out. println(" ====== > [ server] message received : " + s) ;  
9 if ("Heartbeat Packet". equals(s) ) {  
10 ctx. channel() . writeAndFlush("ok") ;  
11 } else {  
12 System. out. println(" 其他信息处理 . . . ") ;  
13 }  
14 }  
15  
16 @Override  
17 public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {  
18 IdleStateEvent event = (IdleStateEvent) evt;  
19  
20 String eventType = null;  
21 switch (event. state() ) {  
22 case READER\_IDLE:  
23 eventType = "读空闲";  
24 readIdleTimes++; // 读空闲的计数加1  
25 break;  
26 case WRITER\_IDLE:  
27 eventType = "写空闲";  
28 // 不处理  
29 break;  
30 case ALL\_IDLE:  
31 eventType = "读写空闲";  
32 // 不处理  
33 break;  
34 }  
35 System. out. println(ctx. channel() . remoteAddress() + "超时事件： " + eventType) ;  
36 if (readIdleTimes > 3) {  
37 System. out. println(" [ server] 读空闲超过3次， 关闭连接， 释放更多资源") ;  
38 ctx. channel() . writeAndFlush("idle close") ;  
39 ctx. channel() . close() ;  
40 }  
41 }  
42  
43 @Override  
44 public void channelActive(ChannelHandlerContext ctx) throws Exception {  
45 System. err. println("=== " + ctx. channel() . remoteAddress() + " is active ===") ;  
46 }  
47 }  
1 //客户端代码  
2 public class HeartBeatClient {  
3 public static void main(String[ ] args) throws Exception {  
4 EventLoopGroup eventLoopGroup = new NioEventLoopGroup() ;  
5 try {  
6 Bootstrap bootstrap = new Bootstrap() ;  
7 bootstrap. group(eventLoopGroup) . channel(NioSocketChannel. class)  
8 . handler(new ChannelInitializer<SocketChannel>() {  
9 @Override  
10 protected void initChannel(SocketChannel ch) throws Exception {  
11 ChannelPipeline pipeline = ch. pipeline() ;  
12 pipeline. addLast("decoder", new StringDecoder() ) ;  
13 pipeline. addLast("encoder", new StringEncoder() ) ;  
14 pipeline. addLast(new HeartBeatClientHandler() ) ;  
15 }  
16 }) ;  
17  
18 System. out. println("netty client start。 。 ") ;  
19 Channel channel = bootstrap. connect("127. 0. 0. 1", 9000) . sync() . channel() ;  
20 String text = "Heartbeat Packet";  
21 Random random = new Random() ;  
22 while (channel. isActive() ) {  
23 int num = random. nextInt(10) ;  
24 Thread. sleep(num \* 1000) ;  
25 channel. writeAndFlush(text) ;  
26 }  
27 } catch (Exception e) {  
28 e. printStackTrace() ;  
29 } finally {  
30 eventLoopGroup. shutdownGracefully() ;  
31 }  
32 }  
33  
34 static class HeartBeatClientHandler extends SimpleChannelInboundHandler<String> {  
35  
36 @Override  
37 protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {  
38 System. out. println(" client received : " + msg) ;  
39 if (msg ! = null && msg. equals("idle close") ) {  
40 System. out. println(" 服务端关闭连接， 客户端也关闭") ;  
41 ctx. channel() . closeFuture() ;  
42 }  
43 }  
44 }  
45 }

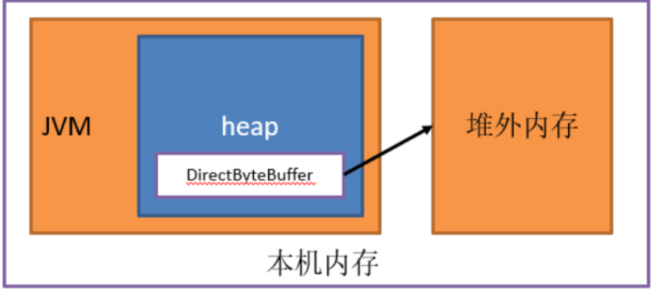
### 2.18 netty零拷贝

Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写， 不需要进行字节缓冲区的二次拷贝。如果使用传统的JVM堆内存（HEAP BUFFERS） 进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中， 然后才能写入Socket中。 JVM堆内存的数据是不能直接写入Socket中的。相比于堆外直接内存， 消息在发送过程中多了一次缓冲区的内存拷贝。可以看下netty的读写源码，比如read源码NioByteUnsafe.read()



**直接内存：**

直接内存（Direct Memory） 并不是虚拟机运行时数据区的一部分， 也不是Java虚拟机规范中定义的内存区域， 某些情况下这部分内存也会被频繁地使用， 而且也可能导致OutOfMemoryError异常出现。 Java里用DirectByteBuffer可以分配一块直接内存(堆外内存)， 元空间对应的内存也叫作直接内存， 它们对应的都是机器的物理内存。



\* 直接内存与堆内存的区别  
3 \*/  
4 public class MemoryTest {  
56  
public static void heapAccess() {  
7 long startTime = System. currentTimeMillis() ;  
8 //分配堆内存  
9 ByteBuffer buffer = ByteBuffer. allocate(1000) ;  
10 for (int i = 0; i < 100000; i++) {  
11 for (int j = 0; j < 200; j++) {  
12 buffer. putInt(j) ;  
13 }  
14 buffer. flip() ;  
15 for (int j = 0; j < 200; j++) {  
16 buffer. getInt() ;  
17 }  
18 buffer. clear() ;  
19 }  
20 long endTime = System. currentTimeMillis() ;  
21 System. out. println("堆内存访问: " + (endTime ‐ startTime) ) ;  
22 }  
23  
24 public static void directAccess() {  
25 long startTime = System. currentTimeMillis() ;  
26 //分配直接内存  
27 ByteBuffer buffer = ByteBuffer. allocateDirect(1000) ;  
28 for (int i = 0; i < 100000; i++) {  
29 for (int j = 0; j < 200; j++) {  
30 buffer. putInt(j) ;  
31 }  
32 buffer. flip() ;  
33 for (int j = 0; j < 200; j++) {  
34 buffer. getInt() ;  
35 }  
36 buffer. clear() ;  
37 }  
38 long endTime = System. currentTimeMillis() ;  
39 System. out. println("直接内存访问: " + (endTime ‐ startTime) ) ;  
40 }  
41  
42 public static void heapAllocate() {  
43 long startTime = System. currentTimeMillis() ;  
44 for (int i = 0; i < 100000; i++) {  
45 ByteBuffer. allocate(100) ;  
46 }  
47 long endTime = System. currentTimeMillis() ;  
48 System. out. println("堆内存申请: " + (endTime ‐ startTime) ) ;  
49 }  
50  
51 public static void directAllocate() {  
52 long startTime = System. currentTimeMillis() ;  
53 for (int i = 0; i < 100000; i++) {  
54 ByteBuffer. allocateDirect(100) ;  
55 }  
56 long endTime = System. currentTimeMillis() ;  
57 System. out. println("直接内存申请: " + (endTime ‐ startTime) ) ;  
58 }  
59  
60 public static void main(String args[ ] ) {  
61 for (int i = 0; i < 10; i ++) {  
62 heapAccess() ;  
63 directAccess() ;  
64 }  
65  
66 System. out. println() ;  
67  
68 for (int i = 0; i < 10; i ++) {  
69 heapAllocate() ;  
70 directAllocate() ;  
71 }  
72 }  
73 }

从程序运行结果看出直接内存申请较慢，但访问效率高。在java虚拟机实现上，本地IO会直接操作直接内存（直接内存=> 系统调用=> 硬盘/网卡），而非直接内存则需要二次拷贝（堆内存=> 直接内存=> 系统调用=> 硬盘/网卡），即多了一步从核心态到用户态的拷贝。

直接内存分配源码分析：  
1 public static ByteBuffer allocateDirect(int capacity) {  
2 return new DirectByteBuffer(capacity) ;  
3 }  
456  
DirectByteBuffer(int cap) { // package‐private  
7 super(‐1, 0, cap, cap) ;  
8 boolean pa = VM. isDirectMemoryPageAligned() ;  
9 int ps = Bits. pageSize() ;  
10 long size = Math. max(1L, (long) cap + (pa ? ps : 0) ) ;  
11 //判断是否有足够的直接内存空间分配， 可通过‐XX: MaxDirectMemorySize=<size>参数指定直接内存最大可分配空间， 如果不指定默认为最大堆内存大小，  
12 //在分配直接内存时如果发现空间不够会显示调用System. gc()触发一次full gc回收掉一部分无用的直接内存的引 用对象， 同时直接内存也会  
被释放掉  
13 //如果释放完分配空间还是不够会抛出异常java. lang. OutOfMemoryError  
14 Bits. reserveMemory(size, cap) ;  
15  
16 long base = 0;  
17 try {  
18 // 调用unsafe本地方法分配直接内存  
19 base = unsafe. allocateMemory(size) ;  
20 } catch (OutOfMemoryError x) {  
21 // 分配失败， 释放内存  
22 Bits. unreserveMemory(size, cap) ;  
23 throw x;  
24 }  
25 unsafe. setMemory(base, size, (byte) 0) ;  
26 if (pa && (base % ps ! = 0) ) {  
27 // Round up to page boundary  
28 address = base + ps ‐ (base & (ps ‐ 1) ) ;  
29 } else {  
30 address = base;  
31 }  
32  
33 // 使用Cleaner机制注册内存回收处理函数， 当直接内存引 用对象被GC清理掉时，  
34 // 会提前调用这里注册的释放直接内存的Deallocator线程对象的run方法  
35 cleaner = Cleaner. create(this, new Deallocator(base, size, cap) ) ;  
36 att = null;  
37 }  
38  
39  
40 // 申请一块本地内存。 内存空间是未初始化的， 其内容是无法预期的。  
41 // 使用freeMemory释放内存， 使用reallocateMemory修改内存大小  
42 public native long allocateMemory(long bytes) ;  
43  
44 // openjdk8/hotspot/src/share/vm/prims/unsafe. cpp  
45 UNSAFE\_ENTRY(jlong, Unsafe\_AllocateMemory(JNIEnv \*env, jobject unsafe, jlong size) )  
46 UnsafeWrapper("Unsafe\_AllocateMemory") ;  
47 size\_t sz = (size\_t) size;  
48 if (sz ! = (julong) size | | size < 0) {  
49 THROW\_0(vmSymbols: : java\_lang\_IllegalArgumentException() ) ;  
50 }  
51 if (sz == 0) {  
52 return 0;  
53 }  
54 sz = round\_to(sz, HeapWordSize) ;  
55 // 调用os: : malloc申请内存， 内部使用malloc这个C标准库的函数申请内存  
56 void\* x = os: : malloc(sz, mtInternal) ;  
57 if (x == NULL) {  
58 THROW\_0(vmSymbols: : java\_lang\_OutOfMemoryError() ) ;  
59 }  
60 //Copy: : fill\_to\_words((HeapWord\*) x, sz / HeapWordSize) ;  
61 return addr\_to\_java(x) ;  
62 UNSAFE\_END

使用直接内存的优点：

1、不占用堆内存空间，减少发生gc的可能

2、Java虚拟机实现上，本地io会直接操作直接内存（直接内存=> 系统调用=> 硬盘/网卡），而非直接内存则需要二次拷贝（堆内存=> 直接内存=> 系统调用=> 硬盘/网卡）

缺点：

1、初始分配较慢

2、没有JVM直接帮助管理内存， 容易发生内存溢出。 为了避免一直没有FULL GC， 最终导致直接内存把物理内存被耗完。 我们可以指定直接内存的最大值，通过-XX： MaxDirectMemorySize来指定， 当达到阈值的时候， 调用system.gc来进行一次FULL GC，间接把那些没有被使用的直接内存回收掉

**对比Java读写与零拷贝读写**

**JVM发出read() 系统调用。**

OS上下文切换到内核模式（第一次上下文切换）并将数据读取到内核空间缓冲区。(第一次拷贝：hardware ----> kernel buffer）

OS内核然后将数据复制到用户空间缓冲区(第二次拷贝: kernel buffer ——> user buffer)，然后read系统调用返回。而系统调用的返回又会导致一次内核空间到用户空间的上下文切换(第二次上下文切换)。

JVM处理代码逻辑并发送write（）系统调用。

OS上下文切换到内核模式(第三次上下文切换)并从用户空间缓冲区复制数据到内核空间缓冲区(第三次拷贝: user buffer ——> kernel buffer)。

write系统调用返回，导致内核空间到用户空间的再次上下文切换(第四次上下文切换)。将内核空间缓冲区中的数据写到hardware(第四次拷贝: kernel buffer ——> hardware)。

总的来说，传统的I/O操作进行了4次用户空间与内核空间的上下文切换，以及4次数据拷贝。显然在这个用例中，从内核空间到用户空间内存的复制是完全不必要的，因为除了将数据转储到不同的buffer之外，我们没有做任何其他的事情。所以，我们能不能直接从hardware读取数据到kernel buffer后，再从kernel buffer写到目标地点不就好了。为了解决这种不必要的数据复制，操作系统出现了零拷贝的概念。注意，不同的操作系统对零拷贝的实现各不相同。在这里我们介绍linux下的零拷贝实现。

**通过sendfile实现的零拷贝I/O**

发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard driver ——> kernel buffer)。

然后再将数据从内核空间缓冲区拷贝到内核中与socket相关的缓冲区中(第二次拷贝: kernel buffer ——> socket buffer)。

sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。通过DMA引擎将内核空间socket缓冲区中的数据传递到协议引擎(第三次拷贝: socket buffer ——> protocol engine)。

通过sendfile实现的零拷贝I/O只使用了2次用户空间与内核空间的上下文切换，以及3次数据的拷贝。

你可能会说操作系统仍然需要在内核内存空间中复制数据（kernel buffer —>socket buffer）。 是的，但从操作系统的角度来看，这已经是零拷贝，因为没有数据从内核空间复制到用户空间。 内核需要复制的原因是因为通用硬件DMA访问需要连续的内存空间（因此需要缓冲区）。 但是，如果硬件支持scatter-and-gather，这是可以避免的。

**带有DMA收集拷贝功能的sendfile实现的I/O**

发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard drive ——> kernel buffer)。

没有数据拷贝到socket缓冲区。取而代之的是只有相应的描述符信息会被拷贝到相应的socket缓冲区当中。该描述符包含了两方面的信息：a)kernel buffer的内存地址；b)kernel buffer的偏移量。

sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。DMA gather copy根据socket缓冲区中描述符提供的位置和偏移量信息直接将内核空间缓冲区中的数据拷贝到协议引擎上(第二次拷贝: kernel buffer ——> protocol engine)，这样就避免了最后一次CPU数据拷贝。

带有DMA收集拷贝功能的sendfile实现的I/O只使用了2次用户空间与内核空间的上下文切换，以及2次数据的拷贝，而且这2次的数据拷贝都是非CPU拷贝。这样一来我们就实现了最理想的零拷贝I/O传输了，不需要任何一次的CPU拷贝，以及最少的上下文切换。

## 2.2 ByteBuf

NIO的byteBuffer与netty的byteBuf的对比

ByteBuffer缓冲区的长度固定，分多了会浪费内存，分少了存放大的数据时会索引越界，所以使用ByteBuffer时，为了解决这个问题，我们一般每次put操作时，都会对可用空间进行校检，如果剩余空间不足，需要重新创建一个新的ByteBuffer，然后将旧的ByteBuffer复制到新的ByteBuffer中去。

而ByteBuf则对其进行了改进，它会自动扩展，具体的做法是，写入数据时，会调用ensureWritable方法，传入我们需要写的字节长度，判断是否需要扩容

### 2.2.1 ByteBuf的详解

从结构上来说，ByteBuf 由一串字节数组构成。数组中每个字节用来存放信息。  
ByteBuf 提供了两个索引， 一个用于读取数据， 一个用于写入数据。 这两个索引通过在字节数组中移动， 来定位需要读或者写信息的位置。  
当从ByteBuf读取时，它的readerIndex（读索引）将会根据读取的字节数递增。  
同样，当写ByteBuf 时，它的 writerIndex 也会根据写入的字节数进行递增



需要注意的是极限的情况是 readerIndex 刚好读到了writerIndex 写入的地方。  
如果 readerIndex 超过了 writerIndex 的时候，Netty 会抛出 IndexOutOf-BoundsException  
异常。

**方法：**

readInt()方法，读取前四个字节的数据并转为int值，readerindex向后移动四位

public class NettyByteBuf {  
2 public static void main(String[ ] args) {  
3 // 创建byteBuf对象， 该对象内部包含一个字节数组byte[ 10]  
4 // 通过readerindex和writerIndex和capacity， 将buffer分成三个区域  
5 // 已经读取的区域： [ 0, readerindex)  
6 // 可读取的区域： [ readerindex, writerIndex)  
7 // 可写的区域: [ writerIndex, capacity)  
8 ByteBuf byteBuf = Unpooled. buffer(10) ;  
9 System. out. println("byteBuf=" + byteBuf) ;  
10  
11 for (int i = 0; i < 8; i++) {  
12 byteBuf. writeByte(i) ;  
13 }  
14 System. out. println("byteBuf=" + byteBuf) ;  
15  
16 for (int i = 0; i < 5; i++) {  
17 System. out. println(byteBuf. getByte(i) ) ;  
18 }  
19 System. out. println("byteBuf=" + byteBuf) ;  
20  
21 for (int i = 0; i < 5; i++) {  
22 System. out. println(byteBuf. readByte() ) ;  
23 }  
24 System. out. println("byteBuf=" + byteBuf) ;  
25  
26  
27 //用Unpooled工具类创建ByteBuf  
28 ByteBuf byteBuf2 = Unpooled. copiedBuffer("hello, zhuge! ", CharsetUtil. UTF\_8) ;  
29 //使用相关的方法  
30 if (byteBuf2. hasArray() ) {  
31 byte[ ] content = byteBuf2. array() ;  
32 //将 content 转成字符串  
33 System. out. println(new String(content, CharsetUtil. UTF\_8) ) ;  
34 System. out. println("byteBuf=" + byteBuf2) ;  
35  
36 System. out. println(byteBuf2. readerIndex() ) ; // 0  
37 System. out. println(byteBuf2. writerIndex() ) ; // 12  
38 System. out. println(byteBuf2. capacity() ) ; // 36  
39  
40 System. out. println(byteBuf2. getByte(0) ) ; // 获取数组0这个位置的字符h的ascii码， h=104  
41  
42 int len = byteBuf2. readableBytes() ; //可读的字节数 12  
43 System. out. println("len=" + len) ;  
44  
45 //使用for取出各个字节  
46 for (int i = 0; i < len; i++) {  
47 System. out. println((char) byteBuf2. getByte(i) ) ;  
48 }  
49  
50 //范围读取  
51 System. out. println(byteBuf2. getCharSequence(0, 6, CharsetUtil. UTF\_8) ) ;  
52 System. out. println(byteBuf2. getCharSequence(6, 6, CharsetUtil. UTF\_8) ) ;  
53 }  
54 }  
55 }

### 2.2.2 byteBuf的扩容缩容

**1、扩容**

public ByteBuf ensureWritable(int minWritableBytes) {

if(minWritableBytes < 0) {

throw Exception

} else if(minWritableBytes <= this.writableBytes()) {

return this;

} else if(minWritableBytes > this.maxCapacity - this.writerIndex) {

throw Exception

} else {

//扩容

int newCapacity = this.calculateNewCapacity(this.writerIndex + minWritableBytes);

this.capacity(newCapacity);

return this;

}

}

可以看到，具体新容量的计算在calculateNewCapacity方法中：

private int calculateNewCapacity(int minNewCapacity) {

int maxCapacity = this.maxCapacity;

int threshold = 4194304; //4mb阀值

if(minNewCapacity == 4194304) {//如果新容量为阀值，直接返回

return 4194304;

} else {

int newCapacity;

if(minNewCapacity > 4194304) {//如果传入的新容量大于阀值，进行计算

newCapacity = minNewCapacity / 4194304 \* 4194304;

if(newCapacity > maxCapacity - 4194304) {//如果大于最大容量，新容量为最大容量

newCapacity = maxCapacity;

} else {//否则新容量 + 阀值 4mb，按照阀值扩容

newCapacity += 4194304;

}

return newCapacity;

} else {//如果小于阀值，则以64为计数倍增，知道倍增的结果>=需要的容量值

for(newCapacity = 64; newCapacity < minNewCapacity; newCapacity <<= 1) {

;

}

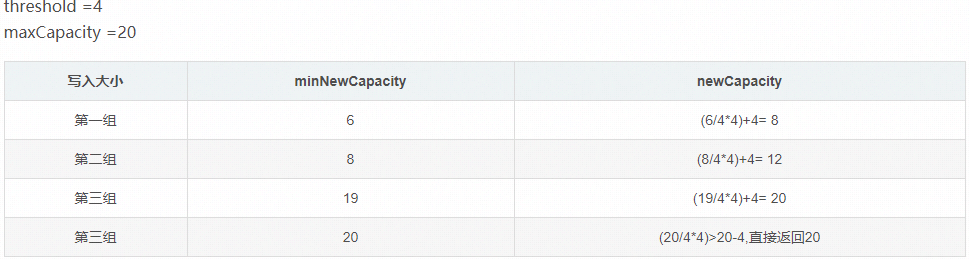
return Math.min(newCapacity, maxCapacity);

}

}

}

如果申请最小的容量刚好等于4MB，则直接返回4MB.如果申请的大小超过了4MB,则需要重新计算新容量大小= minNewCapacity / threshold \* threshold;也就是不会翻倍的扩容，只会以4MB为基数，在申请的容量上，进行扩容，如果小于阀值，则每次翻倍扩容直到满足容量要求



**2、 缩容UnpooledDirectByteBuf#capacity**

@Override

public ByteBuf capacity(int newCapacity) {

checkNewCapacity(newCapacity);

int readerIndex = readerIndex();

int writerIndex = writerIndex();

int oldCapacity = capacity;

if (newCapacity > oldCapacity) {@1

ByteBuffer oldBuffer = buffer;

ByteBuffer newBuffer = allocateDirect(newCapacity);

oldBuffer.position(0).limit(oldBuffer.capacity());

newBuffer.position(0).limit(oldBuffer.capacity());

newBuffer.put(oldBuffer);

newBuffer.clear();

setByteBuffer(newBuffer);

} else if (newCapacity < oldCapacity) {@2

ByteBuffer oldBuffer = buffer;

ByteBuffer newBuffer = allocateDirect(newCapacity);

if (readerIndex < newCapacity) {

if (writerIndex > newCapacity) {

writerIndex(writerIndex = newCapacity);

}

oldBuffer.position(readerIndex).limit(writerIndex);

newBuffer.position(readerIndex).limit(writerIndex);

newBuffer.put(oldBuffer);

newBuffer.clear();

} else {

setIndex(newCapacity, newCapacity);

}

setByteBuffer(newBuffer);

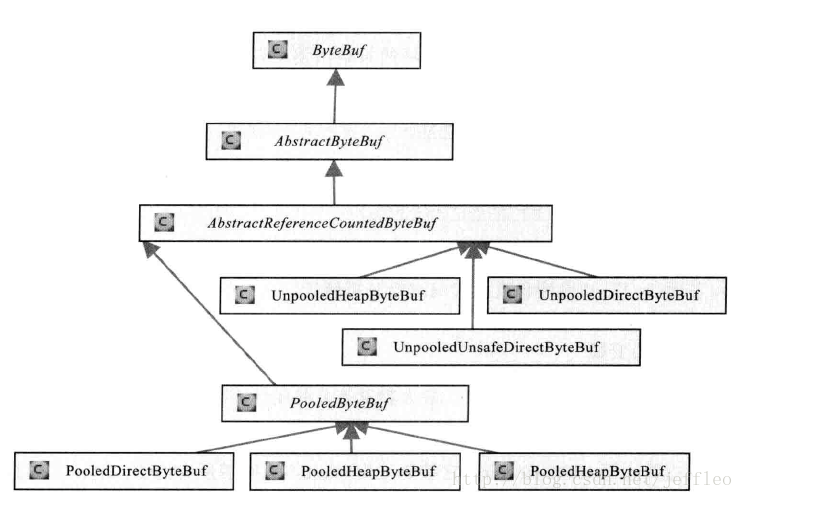
}

return this;

}

如果有未读数据，readerIndex< newCapacity < writerIndex，则更新writeIndex的值为新缩容后的大小，并将未读数据拷贝到新的byteBuf中，如果满足readerIndex < writerIndex < newCapacity 关系，说明缩容后，没有影响到之前的读写区域。 直接更新readerIndex=writerIndex=newCapacity

### 2.2.3 netty的内存回收管理



Netty会通过 引用计数法 及时申请释放不再被引用的对象 ，实现上是通过

AbstractReferenceCountedByteBuf来实现的，我们看上面的结构图，可以看到AbstractReferenceCountedByteBuf是AbstractByteBuf的直接子类，所有具体的实现ByteBuf（堆，非堆等）都是继承自AbstractReferenceCountedByteBuf，也就是说，Netty的具体的实现ByteBuf，都是具有内存回收管理的功能的

AbstractReferenceCountedByteBuf有两个重要的成员变量：

1、AtomicIntegerFieldUpdater< AbstractReferenceCountedByteBuf> refCntUpdater

用来更新引用数，使用原子类，达到线程安全

2、volatile int refCnt = 1

用来记录引用数，保证可见性

**引用+1**

public ByteBuf retain() {

int refCnt;

do {

refCnt = this.refCnt;

if(refCnt == 0) {

throw new IllegalReferenceCountException(0, 1);

}

if(refCnt == 2147483647) {

throw new IllegalReferenceCountException(2147483647, 1);

}

} while(!refCntUpdater.compareAndSet(this, refCnt, refCnt + 1));

return this;

}

**引用-1**

public final boolean release() {

int refCnt;

do {

refCnt = this.refCnt;

if(refCnt == 0) {

throw new IllegalReferenceCountException(0, -1);

}

} while(!refCntUpdater.compareAndSet(this, refCnt, refCnt - 1));

if(refCnt == 1) {

this.deallocate();

return true;

} else {

return false;

}

}

对象时this，自身，也就是说，统计的是自身的引用数，例如对于UnPoolHeapByteBuf来说，它具有统计有多少对象引用着它，当引用数refCnt == 1时，表示此事已经没有对象引用它了，此时便调用deallocate来释放内存

# 3、网络知识

## 3.1 TCP/IP

TCP/IP（Transmission Control Protocol/Internet Protocol，传输控制协议/网际协议）是指能够在多个不同网络间实现信息传输的协议簇。TCP/IP协议不仅仅指的是[TCP](https://baike.baidu.com/item/TCP/33012) 和[IP](https://baike.baidu.com/item/IP/224599)两个协议，而是指一个由[FTP](https://baike.baidu.com/item/FTP/13839)、[SMTP](https://baike.baidu.com/item/SMTP/175887)、TCP、[UDP](https://baike.baidu.com/item/UDP/571511)、IP等协议构成的协议簇， 只是因为在TCP/IP协议中TCP协议和IP协议最具代表性，所以被称为TCP/IP协议

TCP/IP传输协议是严格来说是一个四层的体系结构，应用层、传输层、网络层和数据链路层都包含其中

Osi网络七层模型：物理层、数据链路层、网络层、运输层、会话层、表示层、和应用层

分别介绍TCP/IP协议中的四个层次。

应用层：应用层是TCP/IP协议的第一层，是直接为应用进程提供服务的。

（1）对不同种类的应用程序它们会根据自己的需要来使用应用层的不同协议，邮件传输应用使用了[SMTP](https://baike.baidu.com/item/SMTP/175887)协议、万维网应用使用了[HTTP](https://baike.baidu.com/item/HTTP/243074)协议、远程登录服务应用使用了有[TELNET](https://baike.baidu.com/item/TELNET/810597)协议。

（2）应用层还能加密、解密、格式化数据。

（3）应用层可以建立或解除与其他节点的联系，这样可以充分节省网络资源。 [1]

运输层：作为TCP/IP协议的第二层，运输层在整个TCP/IP协议中起到了中流砥柱的作用。且在运输层中，TCP和UDP也同样起到了中流砥柱的作用。

网络层：网络层在TCP/IP协议中的位于第三层。在TCP/IP协议中网络层可以进行网络连接的建立和终止以及IP地址的寻找等功能。

网络接口层：在TCP/IP协议中，网络接口层位于第四层。由于网络接口层兼并了[物理层](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%B1%82/4329158)和[数据链路层](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%93%BE%E8%B7%AF%E5%B1%82/4329290)所以，网络接口层既是传输数据的物理媒介，也可以为网络层提供一条准确无误的线路。

TCP标志位（位码），有以下六种状态

| **标志位** | **英文** | **含义** |
| --- | --- | --- |
| **SYN** | synchronouse | 建立连接 |
| **ACK** | acknowledgement | 确认标志 |
| **PSH** | push | 传送标志 |
| **FIN** | finish | 结束标志 |
| **RST** | reset | 重置标志 |
| **URG** | urgent | 紧急标志 |

Sequence number 顺序号码

Acknowledgement 确认号码

**连接状态：**

LISTENING： 提供某种服务，侦听远方TCP端口的连接请求，当提供的服务没有被连接时，处于LISTENING状态，端口是开放的，等待被连接。

SYN\_SENT： (客户端状态) 客户端调用connect，发送一个SYN请求建立一个连接，在发送连接请求后等待匹配的连接请求，此时状态为SYN\_SENT。

SYN\_RCVD ：(服务端状态) 在收到和发送一个连接请求后，等待对方对连接请求的确认，当服务器收到客户端发送的同步信号时，将标志位ACK和SYN置1发送给客户端，此时服务器端处于SYN\_RCVD状态，如果连接成功了就变为ESTABLISHED，正常情况下SYN\_RCVD状态非常短暂。

ESTABLISHED：establised状态是表示两台机器正在传输数据。

FIN-WAIT-1：等待TCP连接中断请求的确认，主动关闭端应用程序调用close，TCP发出FIN请求主动关闭连接，之后进入FIN\_WAIT1状态。

FIN-WAIT-2：

1、主动关闭端接到FIN ack后，便进入FIN-WAIT-2，这是在关闭连接时，客户端和服务器两次挥手之后的半关闭状态，在这个状态下，客户端应用程序还有接收数据的能力，但是已经无法发送数据；

2、另一种情况是主动关闭端收到了服务端的FIN请求

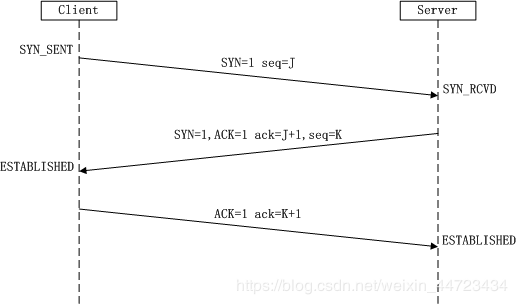
CLOSE-WAIT：等待从本地用户发来的连接中断请求 ，被动关闭端TCP接到FIN后，就发出ACK以回应FIN请求(它的接收也作为文件结束符传递给上层应用程序),并进入CLOSE\_WAIT

CLOSING：等待远程TCP对连接中断的确认,处于此种状态比较少见

LAST-ACK：等待原来的发向远程TCP的连接中断请求的确认,被动关闭端一段时间后，接收到文件结束符的应用程序将调用CLOSE关闭连接,TCP也发送一个 FIN,等待对方的ACK.进入LAST-ACK。

TIME-WAIT：在主动关闭端接收到FIN后，TCP就发送ACK包，并进入TIME-WAIT状态,等待足够的时间以确保远程TCP接收到连接中断请求的确认,很大程度上保证了双方都可以正常结束。

CLOSED：被动关闭端在接受到ACK包后，就进入了closed的状态，连接结束，没有任何连接状态。



第一次握手： Client端发送位码为SYN=1,随机产生seq number=J的数据包到服务器，Server端收到数据包后，由SYN=1判断出 Client端要求连接；此时Client端处于SYN\_SENT的状态。

第二次握手： Server端收到请求后要向Client端发送确认连接的信息，于是，Server端向Client端发送一个ACK=1,SYN=1,ack number=J+1(即Client端的seq number +1),随机生成的seq number=K，此时服务端处于SYN\_RCVD；

第三次握手：

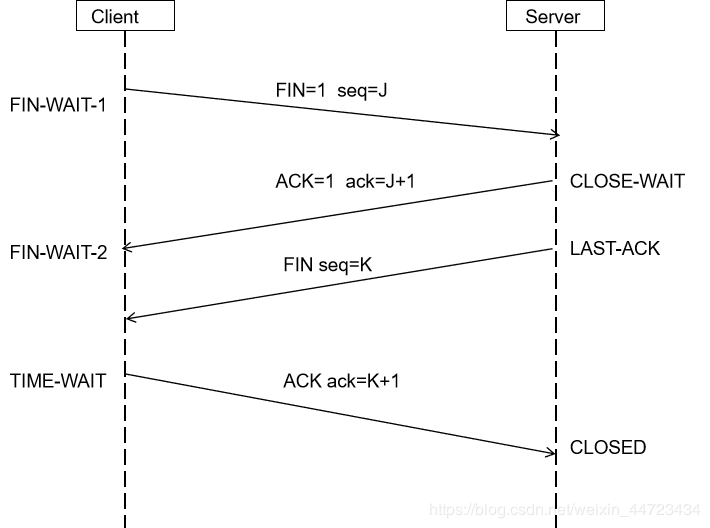
Client端收到后检查两点 ：

1、ack number 是否正确（是否等于J+1）；

2、位码ACK是否等于1。

若以上两点都正确，Client端会再次发送ack num=K+1(第二次机握手中 Server端发送的seq number + 1)，位码ACK=1，Server端收到后确认ack number值是否正确，ACK是否为1 ，若均正确则连接建立成功。

此时双方处于ESTABLISHED的状态。



第一次挥手： Client端发送位码为FIN=1,随机产生seq number=J的数据包到服务器，Server端收到数据包后，由FIN=1判断出 Client端要求断开连接；此时Client端处于FIN-WAIT-1的状态。

第二次挥手： Server端收到请求后要向Client端发送确认断开连接的信息，于是，Server端向Client端发送一个ACK=1,ack number=J+1(即Client端的seq number +1)，此时服务端处于CLOSE\_WAIT的状态，Client端收到这个信号后，由FIN-WAIT-1变成FIN-WAIT-2的状态，此时Client端可以接受Server端的数据但是不能向Server端传输数据。

第三次挥手： Server端主动向Client端发送一个位码为FIN=1,随机产生seq number=K 的数据包到服务器，Client端收到数据包后，由FIN=1判断出Server端要断开连接，此时Server端处于LAST-ACK的状态。

第四次挥手： Client端接受到Server端的请求后，要向Server端发送确认端口连接的信息，于是，Client端向Server端发送了一个ACK=1，ack num=K+1(即Server端的seq number +1)，发送后Client端处于TIME-WAIT的状态，等待2MSL后变成CLOSED，而Server端收到Client端的最后一个ACK后便会变成CLOSED

MSL：Maximum Segment Lifetime最长报文段寿命，默认两分钟，客户端最后等待两个msl时间为了在出现错误数据丢失后能够重新ACK，防止新建立的连接接收到上一个连接的残余数据

TCP/IP与socket的关系：

socket跟TCP/IP并没有必然的联系。Socket编程接口在设计的时候，就希望也能适应其他的网络协议。所以，socket的出现只是可以更方便的使用TCP/IP协议栈而已，其对TCP/IP进行了抽象，形成了几个最基本的函数接口。比如create，listen，accept，connect，read和write等

如果一个程序创建了一个socket，并让其监听80端口，其实是向TCP/IP协议栈声明了其对80端口的占有。以后，所有目标是80端口的TCP数据包都会转发给该程序（这里的程序，因为使用的是Socket编程接口，所以首先由Socket层来处理）。所谓accept函数，其实抽象的是TCP的连接建立过程。accept函数返回的新socket其实指代的是本次创建的连接，而一个连接是包括两部分信息的，一个是源IP和源端口，另一个是宿IP和宿端口。所以，accept可以产生多个不同的socket，而这些socket里包含的宿IP和宿端口是不变的，变化的只是源IP和源端口。这样的话，这些socket宿端口就可以都是80，而Socket层还是能根据源/宿对来准确地分辨出IP包和socket的归属关系，从而完成对TCP/IP协议的操作封装！而同时，防火墙的对IP包的处理规则也是清晰明了，不存在前面设想的种种复杂的情形

## 3.2 websocket协议

### 2.1 webSocket 协议简介

webSocket 是html5 开始提供的一种浏览器与服务器间进行全双工二进制通信协议，其基于TCP双向全双工作进行消息传递，同一时刻即可以发又可以接收消息，相比Http的半双工协议性能有很大的提升，websocket使用了自定义的协议，未加密的连接不再是http://,而是ws://，默认端口号为80，加密的连接也不是https://，而是wss://，端口号为443

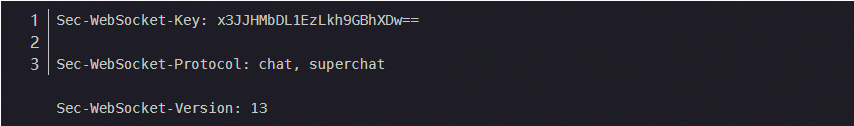
websocket协议在握手阶段通过Upgrade与Connection来声明连接需要升级为websocket协议



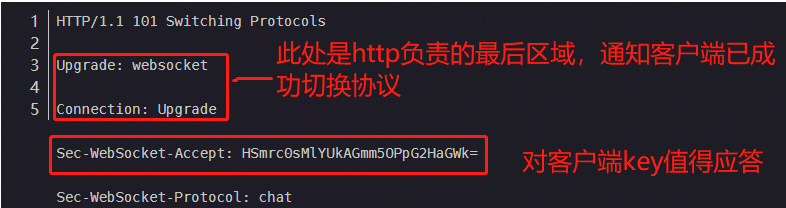
Sec-WebSocket-Key 是一个Base64 encode的值，这个是浏览器随机生成的，服务器受到后会进行应答，如果服务器没有使用websocket协议应答，则客户端接收到的应答格式不正确，握手不成功

Sec\_WebSocket-Protocol 是一个用户定义的字符串，用来区分同URL下，不同的服务所需要的协议。简单理解：今晚我要服务A，别搞错啦~

Sec-WebSocket-Version 是告诉服务器所使用的Websocket Draft（协议版本），在最初的时候，Websocket协议还在 Draft 阶段，各种奇奇怪怪的协议都有，而且还有很多期奇奇怪怪不同的东西，什么Firefox和Chrome用的不是一个版本之类的，当初Websocket协议太多可是一个大难题。。不过现在还好，已经定下来啦大家都使用的一个东西



响应客户端



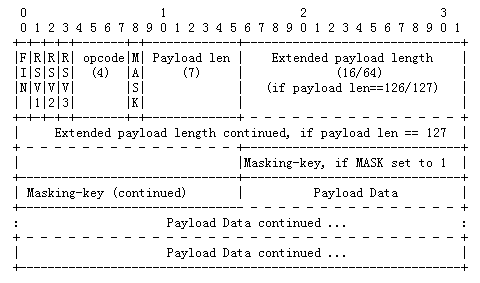
WebSocket是类似Socket的TCP长连接通讯模式。一旦WebSocket连接建立后，后续数据都以帧序列的形式传输。在客户端断开WebSocket连接或Server端中断连接前，不需要客户端和服务端重新发起连接请求。在海量并发及客户端与服务器交互负载流量大的情况下，极大的节省了网络带宽资源的消耗，有明显的性能优势，且客户端发送和接受消息是在同一个持久连接上发起，实时性优势明显。

webSocket特点如下：

1. 单一TCP长连接，采用全双工通信模式
2. 对代理、防火墙透明
3. 无头部信息、消息更精简
4. 通过ping/pong 来保活
5. 服务器可以主动推送消息给客户端，不在需要客户轮询

WebSocket 协议报文格式：

我们知道，任何应用协议都有其特有的报文格式，比如Http协议通过 空格 换行组成其报文。如http 协议不同在于WebSocket属于二进制协议，通过规范进二进位来组成其报文。具体组成如下图：



**报文说明：**

FIN

标识是否为此消息的最后一个数据包，占 1 bit

RSV1, RSV2, RSV3: 用于扩展协议，一般为0，各占1bit

Opcode

数据包类型（frame type），占4bits

0x0：标识一个中间数据包

0x1：标识一个text类型数据包

0x2：标识一个binary类型数据包

0x3-7：保留

0x8：标识一个断开连接类型数据包

0x9：标识一个ping类型数据包

0xA：表示一个pong类型数据包

0xB-F：保留

MASK：占1bits

用于标识PayloadData是否经过掩码处理。如果是1，Masking-key域的数据即是掩码密钥，用于解码PayloadData。客户端发出的数据帧需要进行掩码处理，所以此位是1。

Payload length

Payload data的长度，占7bits，7+16bits，7+64bits：

如果其值在0-125，则是payload的真实长度。

如果值是126，则后面2个字节形成的16bits无符号整型数的值是payload的真实长度。注意，网络字节序，需要转换。

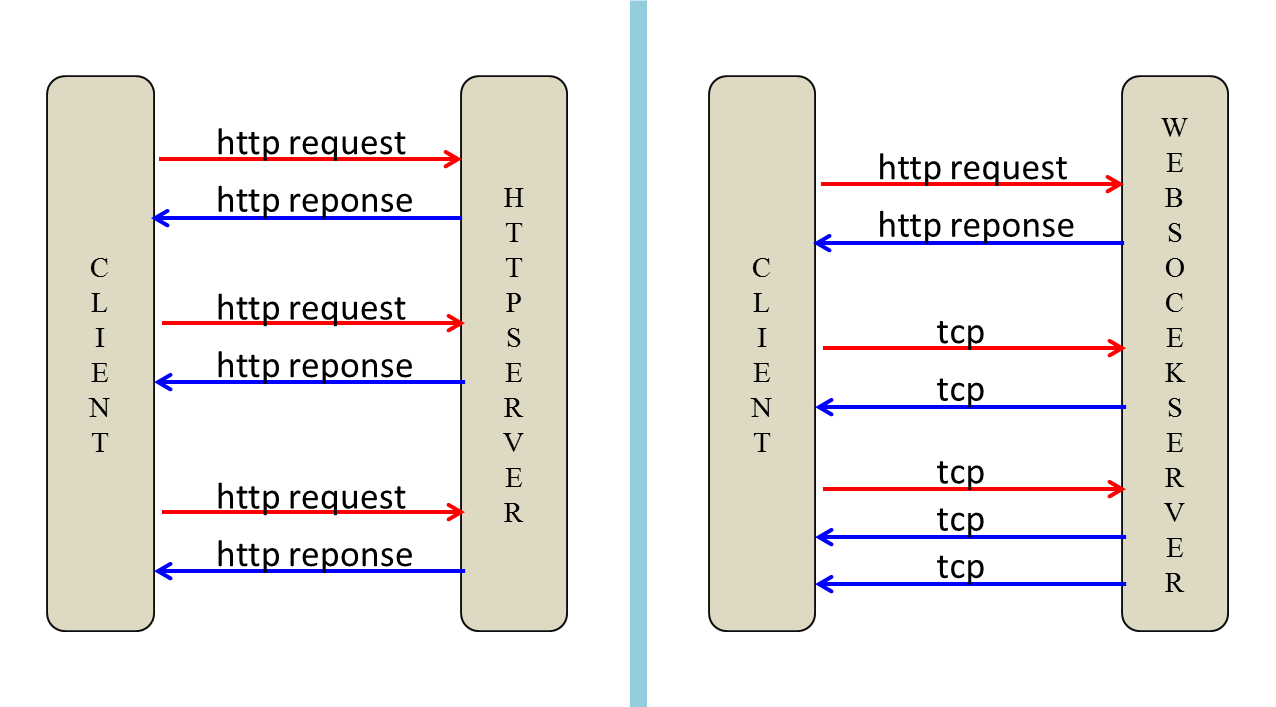
如果值是127，则后面8个字节形成的64bits无符号整型数的值是payload的真实长度。注意，网络字节序，需要转换。

Payload data

应用层数据

WebSocket 在浏览当中的使用

Http 连接与webSocket 连接建立示意图：



通过javaScript 中的API可以直接操作WebSocket 对象，其示例如下：

var ws = new WebSocket(“ws://localhost:8080”);

ws.onopen = function()// 建立成功之后触发的事件 {

  console.log(“打开连接”);   ws.send("ddd"); // 发送消息

  };

  ws.onmessage = function(evt)  { // 接收服务器消息

    console.log(evt.data);

    };

    ws.onclose   = function(evt)  {

      console.log(“WebSocketClosed!”); // 关闭连接  };

      ws.onerror   = function(evt)  {

      console.log(“WebSocketError!”); // 连接异常

 };