**\*@synchronized 同步锁，防止多个线程同时访问一个内存（方法）， @asnc是异步，往往跟@** [EnableAsync](https://www.cnblogs.com/sweetchildomine/p/8166707.html)允许多线程一起操作这个方法（内存）

\*同步就是，当线程1操作同一个内存A时（需要一定事件），其他线程想要访问时，必须排队，此时线程阻塞（一个操作不能在短时间内完成，线程等待），后边线程干不了活，

\*异步就是，当线程1执行内存a时，还没执行完，其他线程也可以操作同一个内存（程序），不必等到线程1执行完，

\*线程，进程，cpu,

1计算机的核心是**CPU，它承担了所有的计算任务**。它就像一座工厂，时刻在运行。

2假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，**单个CPU一次只能运行一个任务**。

3进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态

4一个车间里，可以有很多工人。他们协同完成一个任务

5线程就好比车间里的工人。**一个进程可以包括多个线程**

**6**车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征**一个进程的内存空间是共享的，每个线程都可以使用这些共享内存**。

7可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所（其实可以进两个人**@asnc异步实现，多线程异步操作就是多人同时使用一个厕所**）。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存

8一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫”**互斥锁**“（Mutual exclusion，缩写**Mutex**），防止多个线程同时读写某一块内存区域

9还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用

**Netty是基于异步的（多个连接服务器的客户端线程都可以访问同一段代码，比如chanleRead()），非阻塞的，(异步还非阻塞是因为，多个线程操作内存时会先向一个（服务器的线程）线程申请调用，此时，netty使用api事件通知机制，可以知道哪些连接可以被操作，读或写，所以一个线程可以处理多个连接不阻塞)**

**异步表示线程的执行方式，不主动查询，等待消息通知，同步一直查询**

**阻塞就是反映线程的状态，一直等待，非阻塞，做其他事情去了**

**\*netty是一个线程处理多任务连接，线程就是**

**1.Netty的核心组件**

这些构建块代表了不同类型的构造：资源、逻辑以及通知。你的应用程序将使用它们来访问 网络以及流经网络的数据。 对于每个组件来说，我们都将提供一个基本的定义，并且在适当的情况下，还会提供一个简 单的示例代码来说明它的用法

1.1Channel:

它代表一个到实体（如一个硬件设备、一个文件、一个网络套接字或者一个能够执 行一个或者多个不同的I/O操作的程序组件）的开放连接，如读操作和写操作 。 目前，可以把Channel看作是传入（入站）或者传出（出站）数据的载体。因此，它可以 被打开或者被关闭，连接或者断开连接

1.2回调； 

Netty 在内部使用了回调来处理事件；当一个回调被触发时，相关的事件可以被一个interface- ChannelHandler的实现处理。代码清单 1-2 展示了一个例子：当一个新的连接已经被建立时， ChannelHandler的channelActive()回调方法将会被调用，并将打印出一条信息。

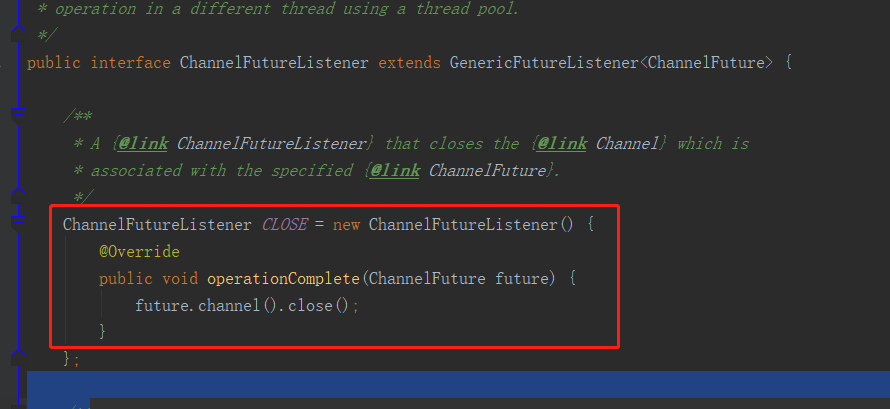
public class ConnectHandler extends ChannelInboundHandlerAdapter {

@Override

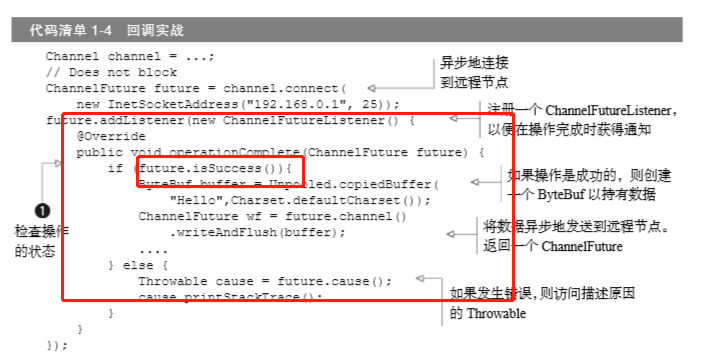
public void channelActive(ChannelHandlerContext ctx) throws Exception {

System.out.println( "Client " + ctx.channel().remoteAddress() + " connected");

} }

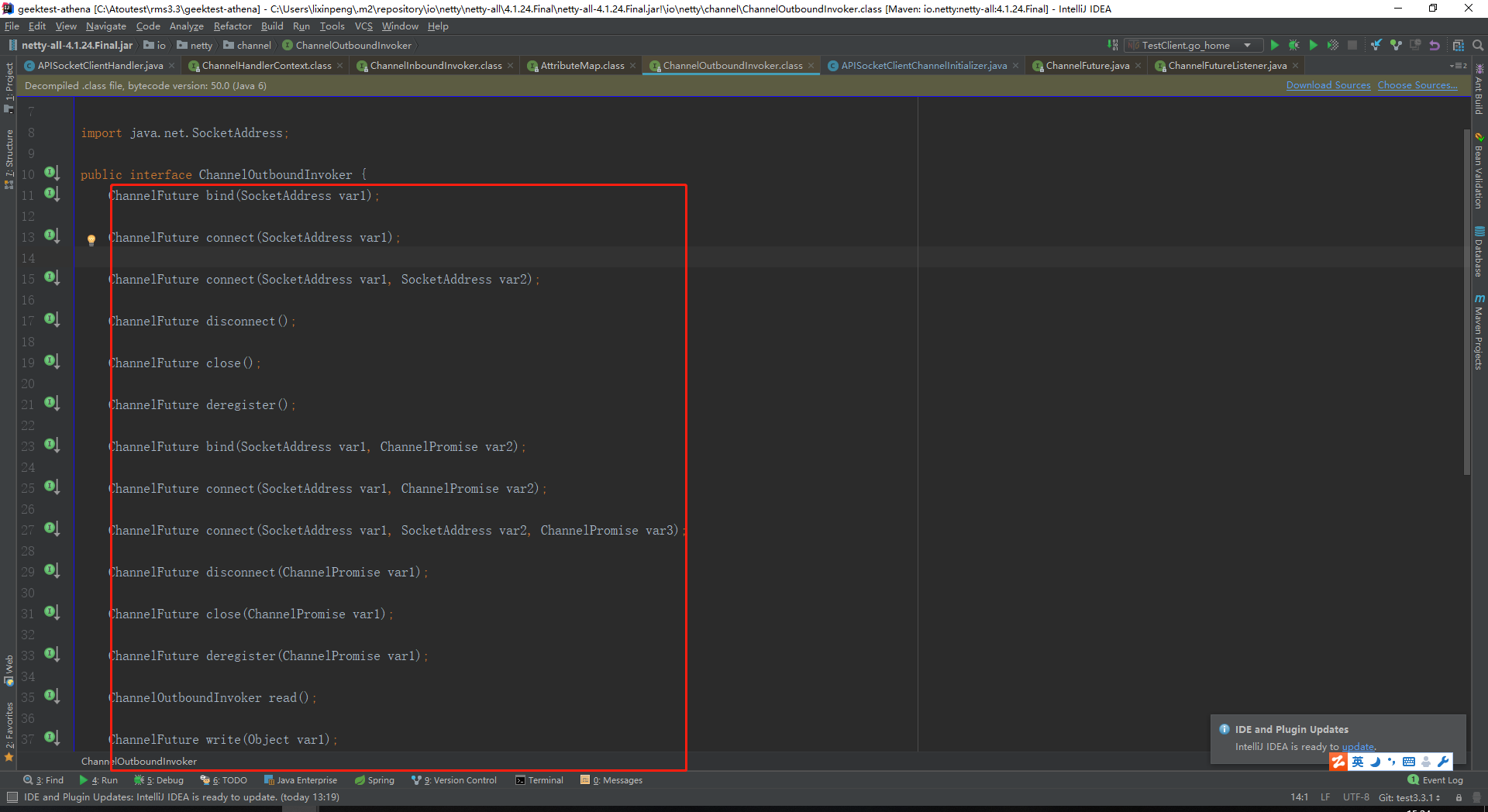
 1.3Future； (chanelFuture如connet()方法会直接返回，具体什么时候完成不知道，但最后完成后会以通知的方式返回到chanelFuture，如果想要根据执行结果作出判断，可以调用方法future.addlisnet(ChannelFutureListener.close.)官方文档如下

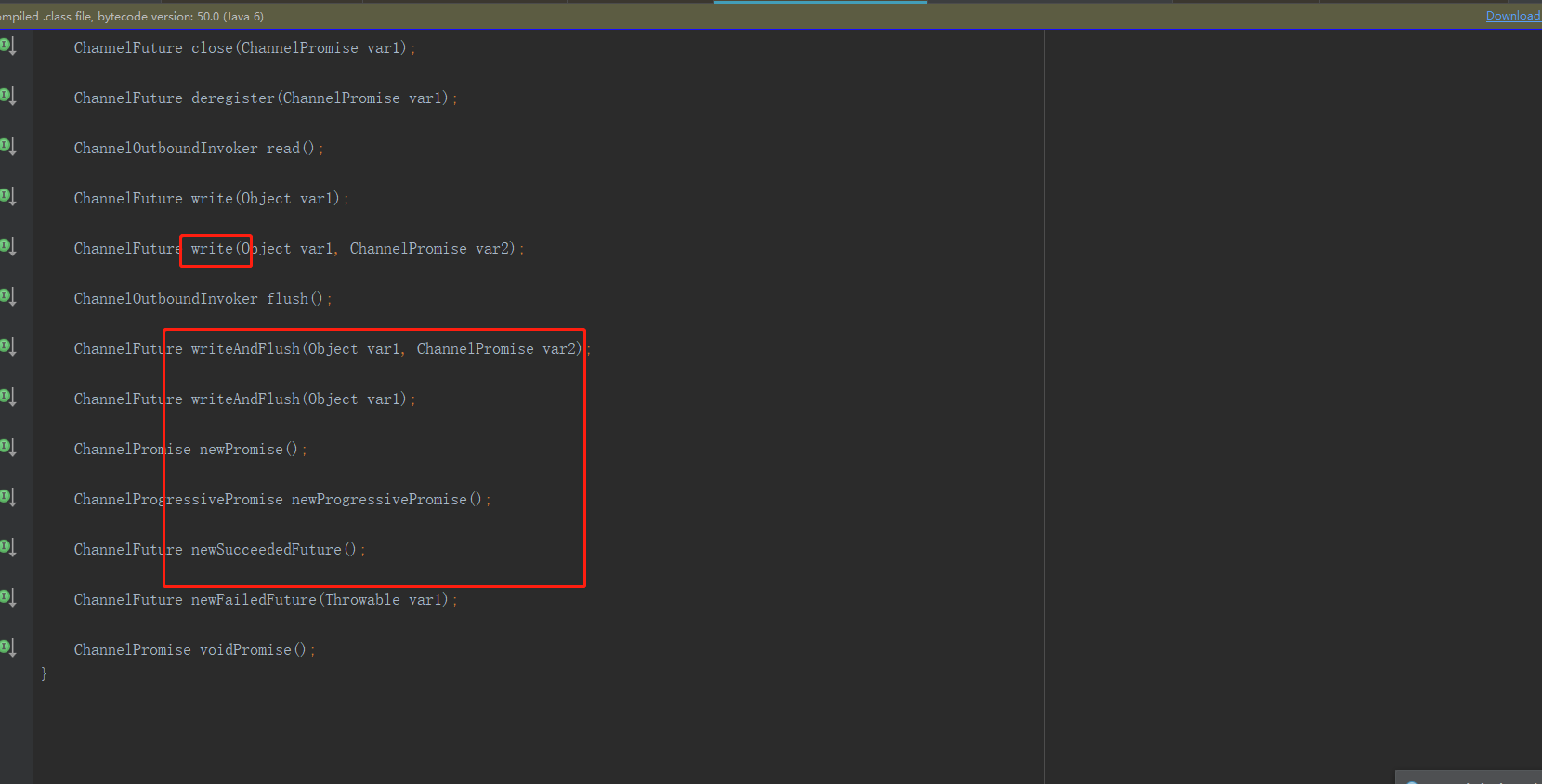
:表示当connet()方法执行完成后，关闭chanel，完成不一定是成功，如果想根据具体执行结果作出操作可以，自定义ChannelFutureListener内部类如下，总之，chanelFuture会在未来某一完成的时刻保存运行后的结果，成功或失败（chanelFuture.isSucceelful）而不会阻塞(任何I/O操作都是非阻塞的，因为返回的都是chanelFuture)。



以下为写操作类（ChannelHandlerContext

.类继承改ChannelOutboundInvoker类，这些方法如writeandflush()）返回都是chanelFuture,都是非阻塞的，会以事件通知（完成时再告诉你，你再在监听器里做你想要做的）



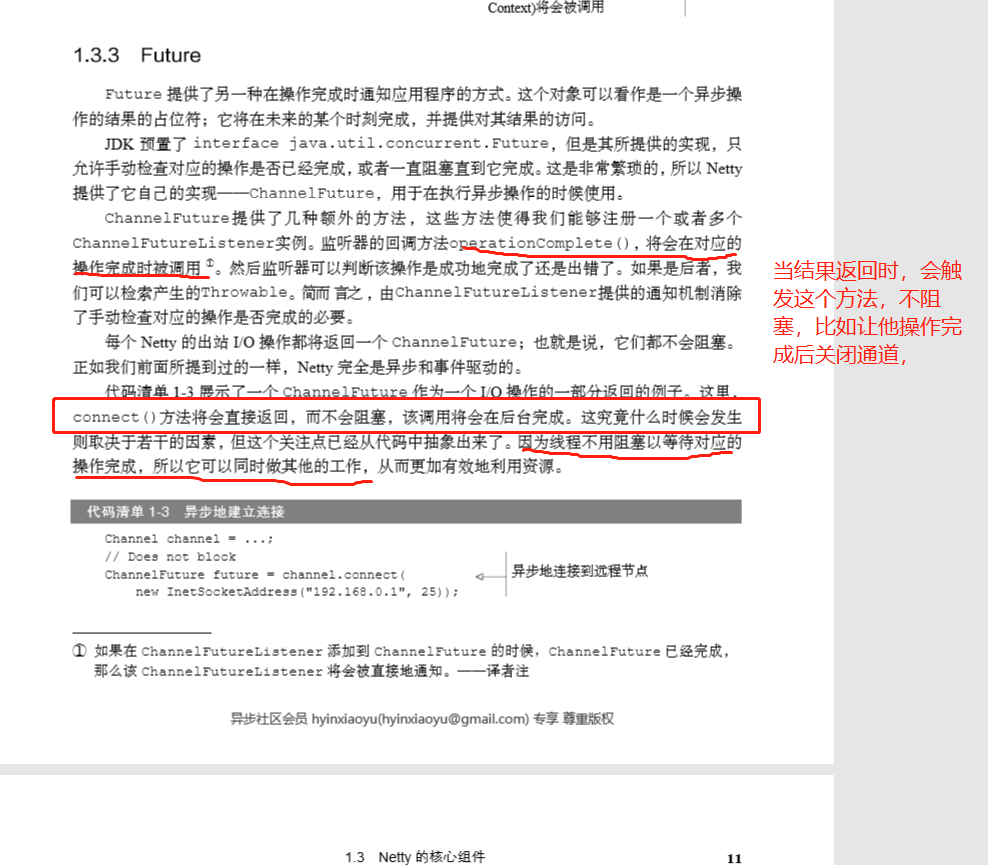


))

Future提供了另一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操 作的结果的占位符；它将在未来的某个时刻完成，并提供对其结果的访问。 JDK 预置了 interface java.util.concurrent.Future，但是其所提供的实现，只 允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以 Netty 提供了它自己的实现——ChannelFuture，用于在执行异步操作的时候使用。 ChannelFuture提供了几种额外的方法，这些方法使得我们能够注册一个或者多个 ChannelFutureListener实例。监听器的回调方法operationComplete()，将会在对应的 操作完成时被调用 ①

// Does not block

。然后监听器可以判断该操作是成功地完成了还是出错了。如果是后者，我 们可以检索产生的Throwable。简而言之，由ChannelFutureListener提供的通知机制消除 了手动检查对应的操作是否完成的必要。 每个 Netty 的出站 I/O 操作都将返回一个ChannelFuture；也就是说，它们都不会阻塞。 正如我们前面所提到过的一样，Netty 完全是异步和事件驱动的。 代码清单 1-3 展示了一个ChannelFuture作为一个 I/O 操作的一部分返回的例子。这里， connect()方法将会直接返回，而不会阻塞，该调用将会在后台完成。这究竟什么时候会发生 则取决于若干的因素，但这个关注点已经从代码中抽象出来了。因为线程不用阻塞以等待对应的 操作完成，所以它可以同时做其他的工作，从而更加有效地利用资源。



代码清单 1-3 异步地建立连接 Channel channel = ...; ChannelFuture future = channel.connect( new InetSocketAddress("192.168.0.1", 25));

① 如果在ChannelFutureListener添加到ChannelFuture的时候，ChannelFuture已经完成， 那么该ChannelFutureListener将会被直接地通知。——译者注

异步地连接到远程节点

当一个新的连接已经被建立时，

channelActive(ChannelHandler Context)将会被调用

异步社区会员 hyinxiaoyu(hyinxiaoyu@gmail.com) 专享 尊重版权

1.3 Netty 的核心组件 11

异步地连接 到远程节点

检查操作 的状态

代码清单 1-4 显示了如何利用 ChannelFutureListener。首先，要连接到远程节点 上。然后，要注册一个新的 ChannelFutureListener 到对 connect()方法的调用所返 回的ChannelFuture上。当该监听器被通知连接已经建立的时候，要检查对应的状态 。 如果该操作是成功的，那么将数据写到该 Channel。否则，要从 ChannelFuture 中检索 对应的Throwable。

代码清单 1-4 回调实战 Channel channel = ...; // Does not block ChannelFuture future = channel.connect( new InetSocketAddress("192.168.0.1", 25)); future.addListener(new ChannelFutureListener() { @Override public void operationComplete(ChannelFuture future) { if (future.isSuccess()){ ByteBuf buffer = Unpooled.copiedBuffer( "Hello",Charset.defaultCharset()); ChannelFuture wf = future.channel() .writeAndFlush(buffer); .... } else { Throwable cause = future.cause(); cause.printStackTrace(); } } }); 需要注意的是，对错误的处理完全取决于你、目标，当然也包括目前任何对于特定类型的错误 加以的限制。例如，如果连接失败，你可以尝试重新连接或者建立一个到另一个远程节点的连接。

如果你把 ChannelFutureListener 看作是回调的一个更加精细的版本，那么你是对的。 事实上，回调和Future是相互补充的机制；它们相互结合，构成了Netty本身的关键构件块之一

1.4事件和ChannelHandler

Netty 使用不同的事件来通知我们状态的改变或者是操作的状态。这使得我们能够基于已经 发生的事件来触发适当的动作。这些动作可能是： 

记录日志； 数据转换； 流控制； 应用程序逻辑。

Netty是一个网络编程框架，所以事件是按照它们与入站或出站数据流的相关性进行分类的。 可能由入站数据或者相关的状态更改而触发的事件包括： 连接已被激活或者连接失活；

注册一个 ChannelFutureListener， 以便在操作完成时获得通知

如果操作是成功的，则创建 一个 ByteBuf 以持有数据

将数据异步地发送到远程节点。 返回一个 ChannelFuture

如果发生错误，则访问描述原因 的 Throwable

数据读取； 用户事件； 错误事件。 出站事件是未来将会触发的某个动作的操作结果，这些动作包括： 打开或者关闭到远程节点的连接； 将数据写到或者冲刷到套接字。 每个事件都可以被分发给ChannelHandler类中的某个用户实现的方法。这是一个很好的 将事件驱动范式直接转换为应用程序构件块的例子。图 1-3 展示了一个事件是如何被一个这样的 ChannelHandler链处理的。

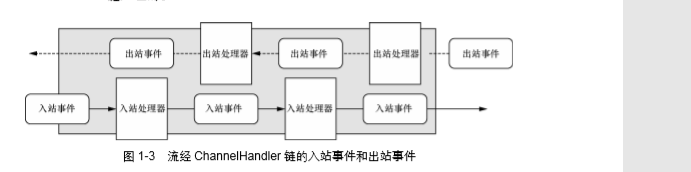
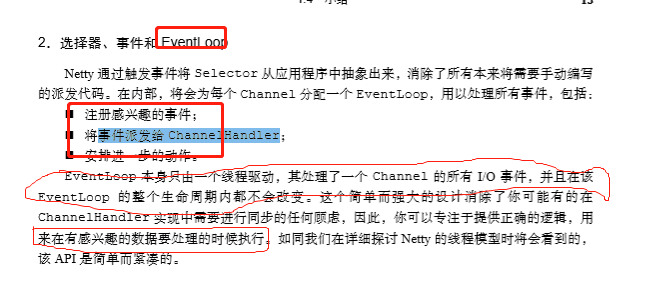


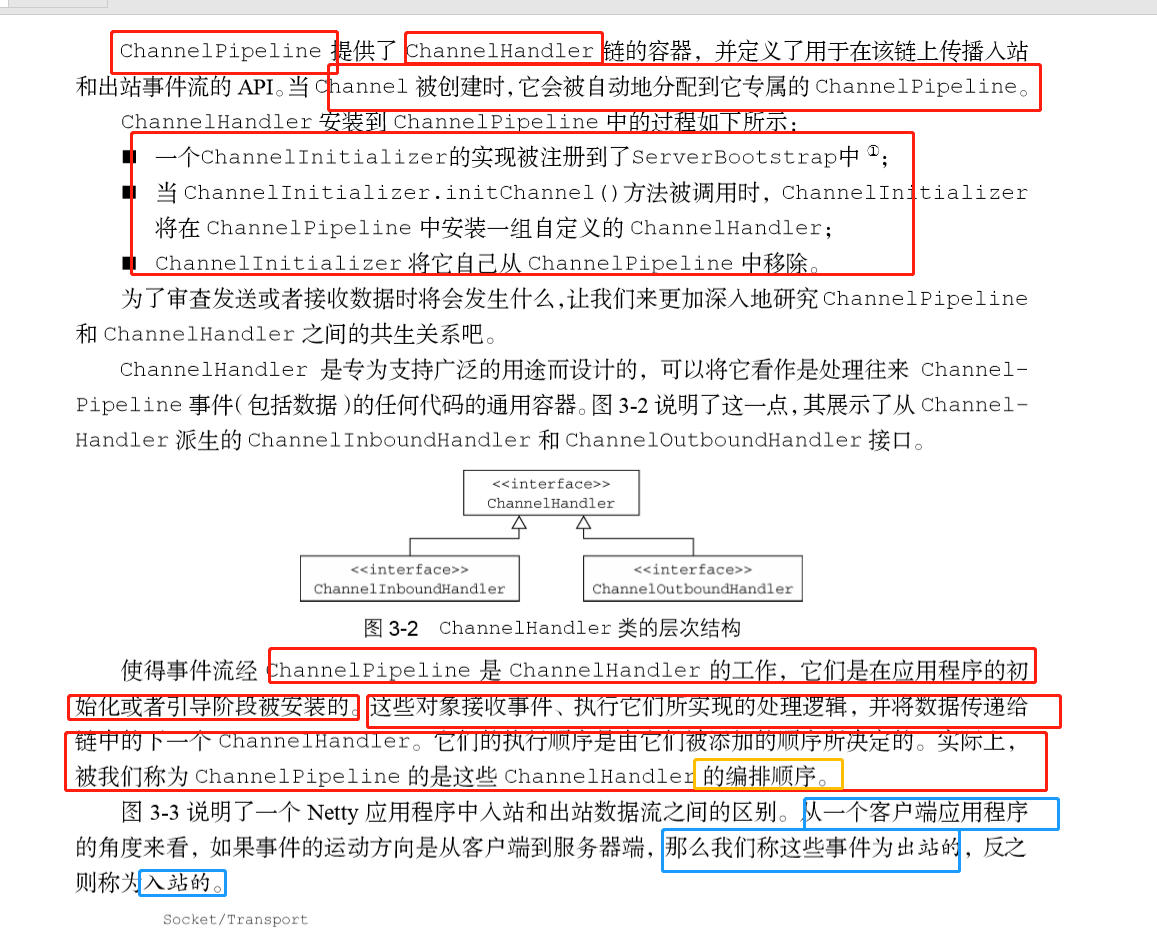
图 1-3 流经 ChannelHandler 链的入站事件和出站事件 Netty 的 ChannelHandler为处理器提供了基本的抽象，如图 1-3 所示的那些。我们会 在适当的时候对 ChannelHandler 进行更多的说明，但是目前你可以认为每个 Channel- Handler的实例都类似于一种为了响应特定事件而被执行的回调。 Netty 提供了大量预定义的可以开箱即用的 ChannelHandler 实现，包括用于各种协议 （如 HTTP 和 SSL/TLS）的ChannelHandler。在内部，ChannelHandler自己也使用了事件 和Future，使得它们也成为了你的应用程序将使用的相同抽象的消费者

1.3evnetLoop

Netty会为每个chanel连接分配一个evnetLoop ：事件循环，处理所有事件的对象（注册特定的事件，事件派发给 ChannelHandler，进一步操作），

**2. ChannelPipeline 提供了 ChannelHandler关系（入站，出战）**

所有事件，即，断开，连接，发送数据，接收数据，这些事件都会触发流经这个chanelpipeline(数据流控制)里的chanledandler,事件触发了，按顺序执行各处理器的功能，编码，解码，接收数据，发送数据，断开，心跳



Chanelread（）所有的回调方法（由事件触发的），执行完后都会传给下一个pipline中的处理器，chanelhandler,下一个不符合，直接再传给一下一个，如下图解释

channelReadComplete(ChannelHandlerContext ctx)，当有数据发过来时，先触发此方法，当读到的字节大小=0或者读到的数据小于缓冲区（bufferSize）大小，会触发完成。并将meg包装成一整条信息，然后将msg作为参数调用channelRead方法，但是如果没有加分隔符，是不会调用Chanelread（）的

\*\*以下为设置缓冲区大小，以及自定义请求的分隔符，

DelimiterBasedFrameDecoder是自定义的分隔符解码，构造函数的第一个参数表示单个消息的最大长度(尽量大，如果单个消息小于等于bufueersize,会一次发送，不会拆包发送，)，当达到该长度后仍然没有查到分隔符，就抛出TooLongFrameException异常，防止由于异常码流缺失分隔符导致的内存溢出。第二个参数为分隔符号，，一次的chanelread()判断的依据，也是channelReadComplete()的依据

//字符串解码器，StringDecoder解码器将ByteBuf解码成字符串对象，这样在ChannelHandlerAdapter中读取消息时就不需要通过ByteBuf获取了

channel.pipeline().addLast(new StringDecoder());  
//字符串编码器，写的时候，如果你发送的sendmessge(string)是string,它会把string转换成bytebuffer传输，  
channel.pipeline().addLast(new StringEncoder());

static final String *DELIMITER* = "\r\n";

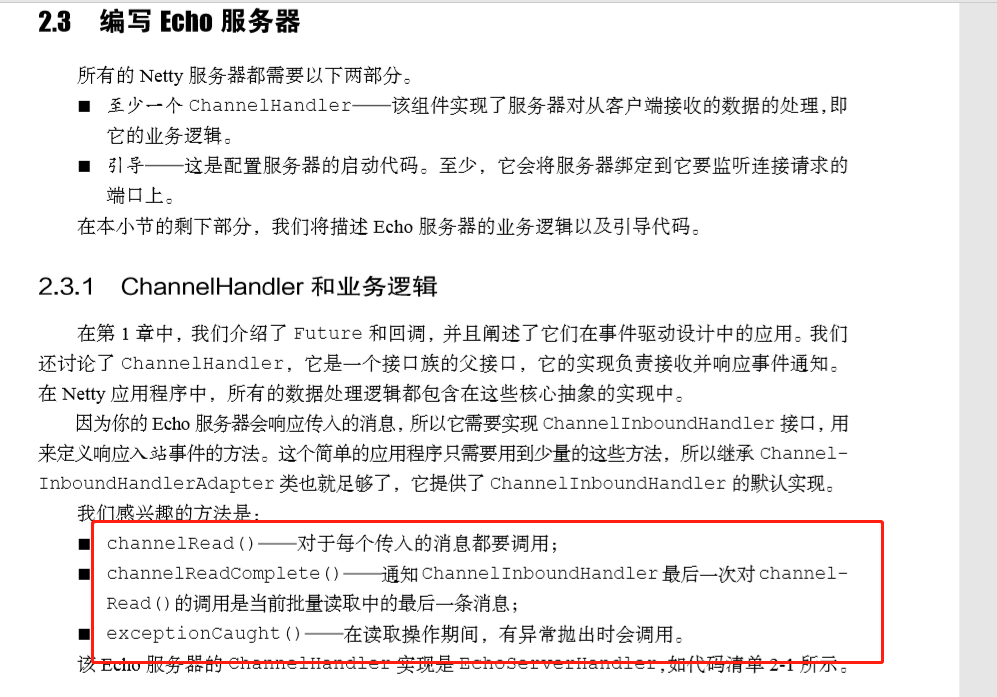
//64X1024byte，一个英文字母占用1byte,足够了

static final int *BUFFER\_SIZE* = 64 \* 1024;

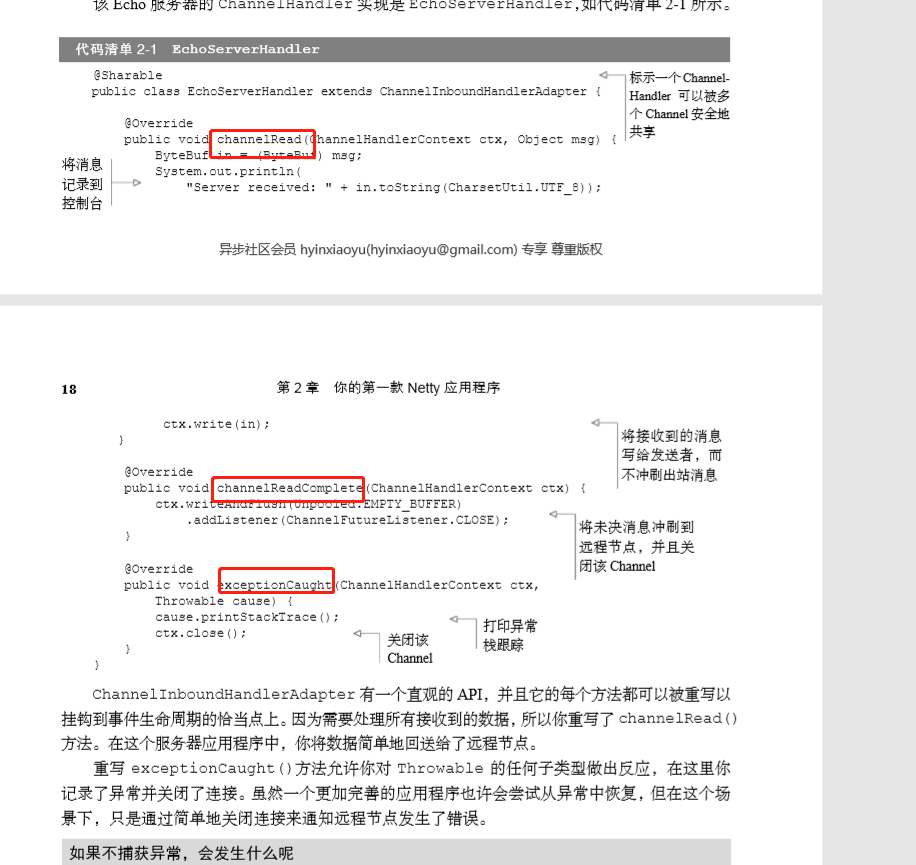
channel.pipeline().addLast(new DelimiterBasedFrameDecoder(*BUFFER\_SIZE*, Unpooled.*copiedBuffer*(*DELIMITER*.getBytes(Charset.*forName*("UTF-8")))));



**3server端代码**



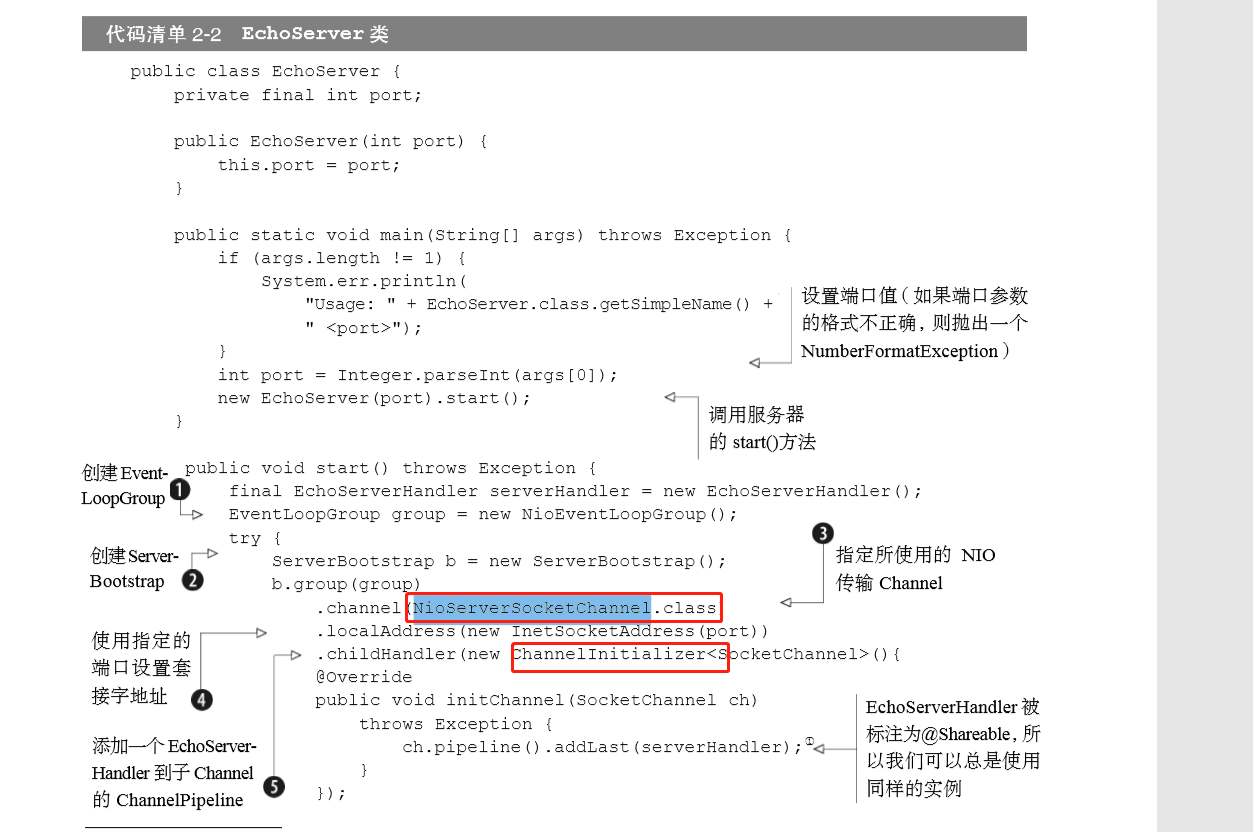
代码如下



EhcoChanelhandler将在inChannelInitializer类里初始化方法中加入pipline中（数据流处理顺序执行），boottrap引导建立连接时，会注册inChannelInitializer

例：boottrap.childHandler(new ChannelInitializer<SocketChannel>(){

**3Boottrap,引导连接(server端)**



其中使用的chanle，连接通道为**NioServerSocketChannel （就是读写的chanel实现类）。**

\*\*Channel 主要有两个实现 NioServerSocketChannel同NioSocketChannel 致于其它不常用不在研究范围内**NioServerSocketChannel 是给server用的，程序由始至终只有一个NioServerSocketChannel**

**NioSocketChannel 是给客户端用的，每个连接生成一个NioSocketChannel 对象**

 详见（https://www.cnblogs.com/solq111/p/7066208.html）解释