### Netty的特点

* **高并发**

Netty是一款基于NIO（Nonblocking I/O，非阻塞IO）开发的网络通信框架。

* **传输快**

Netty的传输快其实也是依赖了NIO的一个特性——零拷贝。

* **封装好**

Netty封装了NIO操作的很多细节，提供易于使用的API。

### BIO、NIO和AIO的区别

bio:阻塞读

nio:轮询

aio:读好了，os通知，netty采用channelfuture通知

### NIO的组成是什么

buffer:与channel进行交互,方法：flip,clear,rewind

directbytebuff:堆外缓冲区

channel:双向，io通道，只能访问buff

selector:管理channel的，selectkey

### epoll 和select的区别

I/O多路复用器单个进程可以同时处理多个描述符的I/O，Java应用程序通过调用多路复用器来获取有事件发生的文件描述符，以进行I/O的读/写操作。多路复用器常见的底层实现模型有epoll模型和select模型：

select模型只有一个select函数，每次在调用select函数时，都需要把整个文件描述符集合从用户态拷贝到内核态，当文件描述符很多时，开销会比较大。

每次在调用select函数时，内核都需要遍历所有的文件描述符，这个开销也很大，尤其是当很多文件描述符根本就无状态改变时，也需要遍历，浪费性能。

select可支持的文件描述符有上限，可监控的文件描述符个数取决于sizeOf(fd*set)的值。如果sizeOf(fd*set)=512，那么此服务器最多支持512×8=4096个文件描述符。

epoll模型有三个函数。第一个函数为int epoll*create(int size)，用于创建一个epoll句柄。第二个函数为int epoll*ctl(int epfd,int op,int fd,struct epoll*event\*event)，其中，第一个参数为epoll*create函数调用返回的值；第二个参数表示操作动作，由三个宏（EPOLL*CTL*ADD表示注册新的文件描述符到此epfd上，EPOLL*CTL*MOD表示修改已经注册的文件描述符的监听事件，EPOLL*CTL*DEL表示从epfd中删除一个文件描述符）来表示；第三个参数为需要监听的文件描述符；第四个参数表示要监听的事件类型，事件类型也是几个宏的集合，主要是文件描述符可读、可写、发生错误、被挂断和触发模式设置等。epoll模型的第三个函数为epoll\_wait，表示等待文件描述符就绪。

所有需要监听的文件描述符只需在调用第二个函数int epoll*ctl时拷贝一次即可，当文件描述符状态发生改变时，内核会把文件描述符放入一个就绪队列中，通过调用epoll*wait函数获取就绪的文件描述符。

每次调用epoll\_wait函数只会遍历状态发生改变的文件描述符，无须全部遍历，降低了操作的时间复杂度。

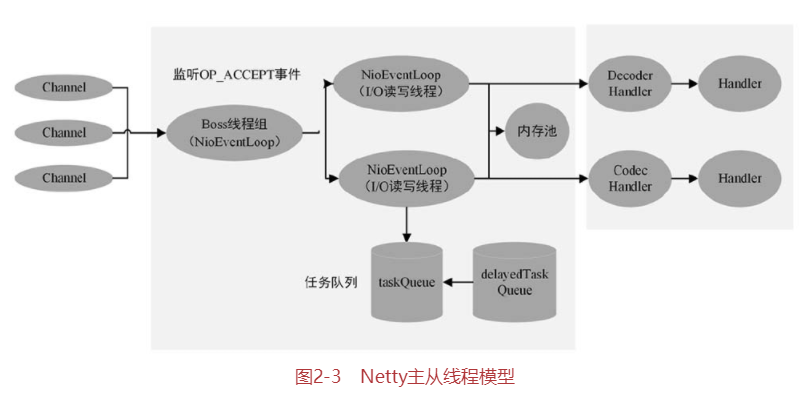
没有文件描述符个数的限制。

采用了内存映射机制，内核直接将就绪队列通过MMAP的方式映射到用户态，避免了内存拷贝带来的额外性能开销。

### netty的线程模型

Boss线程组和Worker线程组。其中，Boss线程组一般只开启一条线程，除非一个Netty服务同时监听多个端口。Worker线程数默认是CPU核数的两倍，Boss线程主要监听SocketChannel的OP\_ACCEPT事件和客户端的连接（主线程）。

当Boss线程监听到有SocketChannel连接接入时，会把SocketChannel包装成NioSocketChannel，并注册到Worker线程的Selector中，同时监听其OP*WRITE和OP*READ事件。当Worker线程监听到某个SocketChannel有就绪的读I/O事件时，会进行以下操作。（1）向内存池中分配内存，读取I/O数据流。（2）将读取后的ByteBuf传递给解码器Handler进行解码，若能解码出完整的请求数据包，就会把请求数据包交给业务逻辑处理Handler。（3）经过业务逻辑处理Handler后，在返回响应结果前，交给编码器进行数据加工。（4）最终写到缓存区，并由I/O Worker线程将缓存区的数据输出到网络中并传输给客户端。



### 序列化

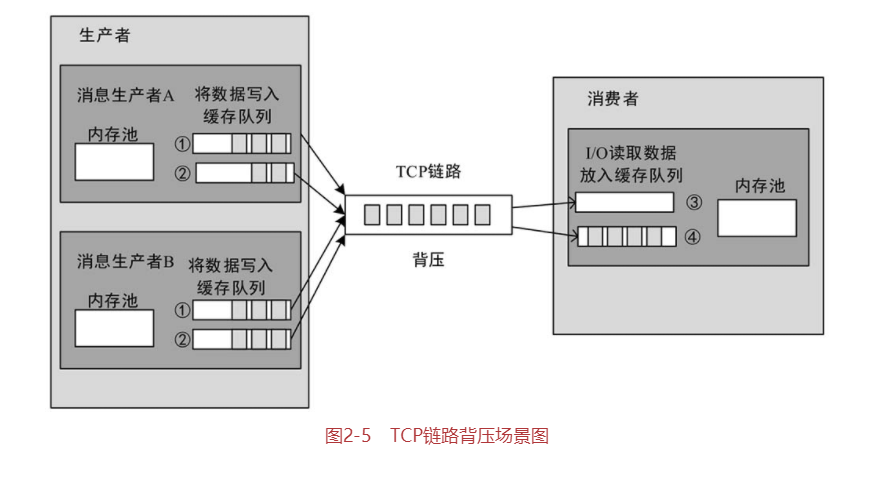
如Protobuf、Kryo、JSON等。由于JSON格式化数据可读性好，而且浏览器对JSON数据的支持性非常好，所以一般的Web应用都会选择它。

### 零拷贝

1. Netty的接收和发送ByteBuffer使用直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用JVM的堆内存进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于使用直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。
2. Netty的文件传输调用FileRegion包装的transferTo方法，可以直接将文件缓冲区的数据发送到目标Channel，避免通过循环write方式导致的内存拷贝问题。
3. Netty提供CompositeByteBuf类, 可以将多个ByteBuf合并为一个逻辑上的ByteBuf, 避免了各个ByteBuf之间的拷贝。
4. 通过wrap操作, 我们可以将byte[]数组、ByteBuf、ByteBuffer等包装成一个Netty ByteBuf对象, 进而避免拷贝操作。
5. ByteBuf支持slice操作，可以将ByteBuf分解为多个共享同一个存储区域的ByteBuf, 避免内存的拷贝。

### 背压

当消费者的消费速率低于生产者的发送速率时，会造成背压，此时消费者无法从TCP缓存区中读取数据，因为它无法再从内存池中获取内存，从而造成TCP通道阻塞。生产者无法把数据发送出去，这就使生产者不再向缓存队列中写入数据，从而降低了生产速率。Netty的背压主要运用TCP的流量控制来完成整个链路的背压效



### 使用 Java NIO 的客户端与服务端

服务端：
  
 Selector selector = Selector.open();
  
   
 ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
  
 serverSocketChannel.socket().bind(new InetSocketAddress(8080), 1024);
  
 serverSocketChannel.configureBlocking(false);
  
 serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);
  
   
   
 while (true) {
  
 selector.selectNow();
  
 Set<SelectionKey> selectionKeys = selector.selectedKeys();
  
 Iterator<SelectionKey> it = selectionKeys.iterator();
  
 while (it.hasNext()) {
  
 SelectionKey selectionKey = it.next();
  
 it.remove();
  
 handleInput(selector, selectionKey);
  
 }
  
 }
  
客户端：
  
 SocketChannel socketChannel = SocketChannel.open();
  
 socketChannel.connect(new InetSocketAddress("0.0.0.0",8090));
  
   
 ByteBuffer writeBuffer = ByteBuffer.allocate(32);
  
 ByteBuffer readBuffer = ByteBuffer.allocate(32);
  
   
 writeBuffer.put("hello".getBytes());
  
 writeBuffer.flip();
  
 while (true){
  
 writeBuffer.rewind();
  
 socketChannel.write(writeBuffer);
  
 readBuffer.clear();
  
 socketChannel.read(readBuffer);
  
 readBuffer.flip();
  
 System.out.println(new String(readBuffer.array()));

### 如何使用netty的客户端与服务端

客户端：
  
 final SslContext sslCtx;
  
 if (SSL) {
  
 sslCtx = SslContextBuilder.forClient()
  
 .trustManager(InsecureTrustManagerFactory.INSTANCE).build();
  
 } else {
  
 sslCtx = null;
  
 }
  
   
 // Configure the client.
  
 EventLoopGroup group = new NioEventLoopGroup();
  
 try {
  
 Bootstrap b = new Bootstrap();
  
 b.group(group)
  
 .channel(NioSocketChannel.class)
  
 .option(ChannelOption.TCP\_NODELAY, true)
  
 .handler(new ChannelInitializer<SocketChannel>() {
  
 @Override
  
 public void initChannel(SocketChannel ch) throws Exception {
  
 ChannelPipeline p = ch.pipeline();
  
 if (sslCtx != null) {
  
 p.addLast(sslCtx.newHandler(ch.alloc(), HOST, PORT));
  
 }
  
 //p.addLast(new LoggingHandler(LogLevel.INFO));
  
 p.addLast(new EchoClientHandler());
  
 }
  
 });
  
   
 // Start the client.
  
 ChannelFuture f = b.connect(HOST, PORT).sync();
  
   
 // Wait until the connection is closed.
  
 f.channel().closeFuture().sync();
  
 } finally {
  
 // Shut down the event loop to terminate all threads.
  
 group.shutdownGracefully();
  
 }
  
服务端：
  
 EventLoopGroup bossGroup = new NioEventLoopGroup(1);
  
 EventLoopGroup workerGroup = new NioEventLoopGroup();
  
 final EchoServerHandler serverHandler = new EchoServerHandler();
  
 try {
  
 ServerBootstrap b = new ServerBootstrap();
  
 b.group(bossGroup, workerGroup)
  
 .channel(NioServerSocketChannel.class)
  
 .option(ChannelOption.SO\_BACKLOG, 100)
  
 .handler(new LoggingHandler(LogLevel.INFO))
  
 .childHandler(new ChannelInitializer<SocketChannel>() {
  
 @Override
  
 public void initChannel(SocketChannel ch) throws Exception {
  
 ChannelPipeline p = ch.pipeline();
  
 if (sslCtx != null) {
  
 p.addLast(sslCtx.newHandler(ch.alloc()));
  
 }
  
 //p.addLast(new LoggingHandler(LogLevel.INFO));
  
 p.addLast(serverHandler);
  
 }
  
 });
  
   
 // Start the server.
  
 ChannelFuture f = b.bind(PORT).sync();
  
   
 // Wait until the server socket is closed.
  
 f.channel().closeFuture().sync();
  
 } finally {
  
 // Shut down all event loops to terminate all threads.
  
 bossGroup.shutdownGracefully();
  
 workerGroup.shutdownGracefully();
  
 }

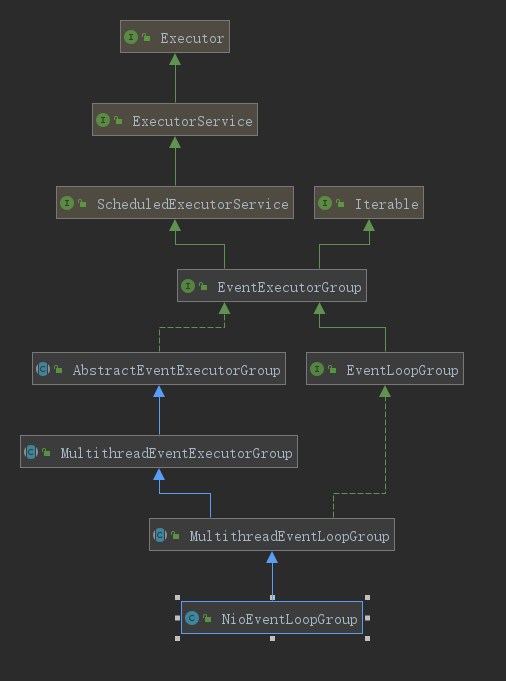
### Netty 底层操作与 Java NIO 操作对应关系

本质是对java nio的封装

NioEventLoopGroup ：SelectorProvider 为select

EpollEventLoopGroup : SelectorProvider 为epoll

### NioEventLoopGroup



NioEventLoopGroup类主要完成以下3件事。

• 创建一定数量的NioEventLoop线程组并初始化。

• 创建线程选择器chooser。当获取线程时，通过选择器来获取。

• 创建线程工厂并构建线程执行器。

在初始化NioEventLoopGroup时，会调用其父类的构造方法。MultithreadEventLoop Group中的DEFAULT*EVENT*LOOP\_THREADS属性决定生成多少NioEventLoop线程，默认为CPU核数的两倍，在构造方法中会把此参数传入，并最终调用MultithreadEventExecutorGroup类的构造方法。此构造方法运用**模板设计模式**来构建线程组生产模板。

线程组的生产分两步：第一步，创建一定数量的EventExecutor数组；第二步，通过调用子类的newChild()方法完成这些EventExecutor数组的初始化。为了提高可扩展性，Netty的线程组除了NioEventLoopGroup，还有Netty通过JNI方式提供的一套由epoll模型实现的EpollEventLoopGroup线程组，以及其他I/O多路复用模型线程组，因此**newChild()方法由具体的线程组子类**来实现。

在newChild()方法中，NioEventLoop的初始化参数有6个：

第1个参数为NioEventLoopGroup线程组本身；

第2个参数为线程执行器，用于启动线程，在SingleThreadEventExecutor的doStartThread()方法中被调用；

第3个参数为NIO的Selector选择器的提供者；

第4个参数主要在NioEventLoop的run()方法中用于控制选择循环；

第5个参数为非I/O任务提交被拒绝时的处理Handler；

第6个参数为队列工厂，在NioEventLoop中，队列读是单线程操作，而队列写则可能是多线程操作，使用支持多生产者、单消费者的队列比较合适，默认为MpscChunkedArrayQueue队列。NioEventLoopGroup通过next()方法获取NioEventLoop线程，最终会调用其父类MultithreadEventExecutorGroup的next()方法，委托父类的选择器EventExecutorChooser。具体使用哪种选择器对象取决于MultithreadEventExecutorGroup的构造方法中使用的**策略模式**。

NioEventLoopGroup通过next()方法获取NioEventLoop线程，最终会调用其父类MultithreadEventExecutorGroup的next()方法，委托父类的选择器EventExecutorChooser。具体使用哪种选择器对象取决于MultithreadEventExecutorGroup的构造方法中使用的策略模式。

由于Netty的NioEventLoop线程被包装成了FastThreadLocalThread线程，同时，NioEventLoop线程的状态由它自身管理，因此每个NioEventLoop线程都需要有一个线程执行器，并且在线程执行前需要通过线程工厂io.netty.util.concurrent.DefaultThreadFactory将其包装成FastThreadLocalThread线程。

### NioEventLoop

NioEventLoop有以下5个核心功能。

• 开启Selector并初始化：openSelector

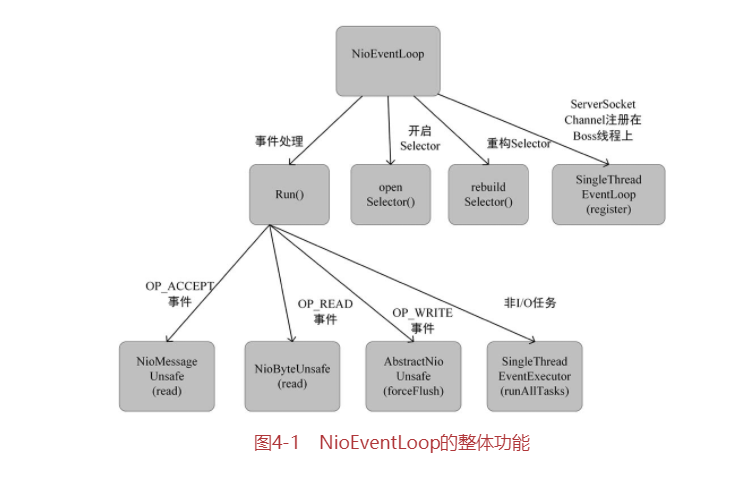
• 把ServerSocketChannel注册到Selector上。

• 处理各种I/O事件，如OP*ACCEPT、OP*CONNECT、OP*READ、OP*WRITE事件。

• 执行定时调度任务。

• 解决JDK空轮询bug。

当select(timeoutMillis)阻塞运行时，在以下4种情况下会正常唤醒线程：其他线程执行了wakeUp唤醒动作、检测到就绪Key、遇上空轮询、超时自动醒来。唤醒线程后，除了空轮询会继续轮询，其他正常情况会跳出循环。



第一部分，select(boolean oldWakenUp)：主要目的是轮询看看是否有准备就绪的Channel。在轮询过程中会调用NIO Selector的selectNow()和select(timeoutMillis)方法。由于对这两个方法的调用进行了很明显的区分，因此调用这两个方法的条件也有所不同，具体逻辑如下。

（1）当定时任务需要触发且之前未轮询过时，会调用selectNow()方法立刻返回。

（2）当定时任务需要触发且之前轮询过（空轮询或阻塞超时轮询）直接返回时，没必要再调用selectNow()方法。（3）若taskQueue队列中有任务，且从EventLoop线程进入select()方法开始后，一直无其他线程触发唤醒动作，则需要调用selectNow()方法，并立刻返回。因为在运行select(booleanoldWakenUp)之前，若有线程触发了wakeUp动作，则需要保证tsakQueue队列中的任务得到了及时处理，防止等待timeoutMillis超时后处理。

（4）当select(timeoutMillis)阻塞运行时，在以下4种情况下会正常唤醒线程：其他线程执行了wakeUp唤醒动作、检测到就绪Key、遇上空轮询、超时自动醒来。唤醒线程后，除了空轮询会继续轮询，其他正常情况会跳出循环。



第三部分，runAllTasks：主要目的是执行taskQueue队列和定时任务队列中的任务，如心跳检测、异步写操作等。由于一个NioEventLoop线程需要管理很多Channel，这些Channel的任务可能非常多，若要都执行完，则I/O事件可能得不到及时处理，因此每执行64个任务后就会检测执行任务的时间是否已用完，如果执行任务的时间用完了，就不再执行后续的任务了。

最后回到NioEventLoop的run()方法，将前面的三个部分结合起来：首先调用select(booleanoldWakenUp)方法轮询就绪的Channel；然后调用processSelectedKeys()方法处理I/O事件；最后运行runAllTasks()方法处理任务队列。

wakeup()方法操作耗性能，因此建议在非复杂处理时，尽量不开额外线程。

从select函数的代码解读中发现，Netty在空轮询次数大于或等于阈值（默认512）时，需要重新构建Selector。重新构建Selector的方式比较巧妙：重新打开一个新的Selector，将旧的Selector上的key和attchment复制过去，同时关闭旧的Selector。

注册方法register()在两个地方被调用：一是在端口绑定前，需要把NioServerSocketChannel注册到Boss线程的Selector上；二是当NioEventLoop监听到有链路接入时，把链路SocketChannel包装成NioSocketChannel，并注册到Woker线程中。最终调用NioSocketChannel的辅助对象unsafe的register()方法，unsafe执行父类AbstractUnsafe的register()模板方法。

NioServerSocketChannel -> AbstractNioChannel中，已经将Netty的Channel和Java NIO的Channel关联起来了

NioServerSocketChannel -> ServerSocketChannel ->ServerChannel -> channel

### channel

Channel是Netty抽象出来的对网络I/O进行读/写的相关接口，与NIO中的Channel接口相似。Channel的主要功能有网络I/O的读/写、客户端发起连接、主动关闭连接、关闭链路、获取通信双方的网络地址等。Channel接口下有一个重要的抽象类——AbstractChannel，一些公共的基础方法都在这个抽象类中实现，一些特定功能可以通过各个不同的实现类去实现。最大限度地实现了功能和接口的重用。（模板设计方法）

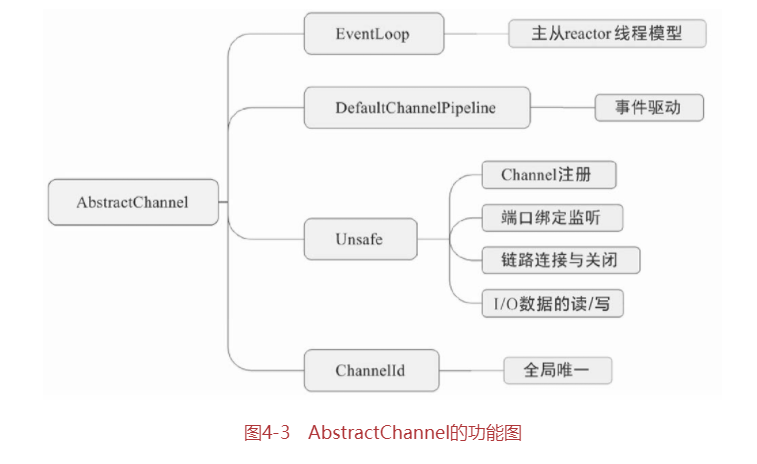
不同协议、不同阻塞类型的连接有不同的Channel类型与之对应，因此AbstractChannel并没有与网络I/O直接相关的操作。每种阻塞与非阻塞Channel在AbstractChannel上都会继续抽象一层。

AbstractChannel抽象类包含以下几个重要属性：

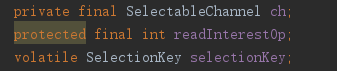
• EventLoop：每个Channel对应一条EventLoop线程。

• DefaultChannelPipeline：一个Handler的容器，也可以将其理解为一个Handler链。Handler主要处理数据的编/解码和业务逻辑。

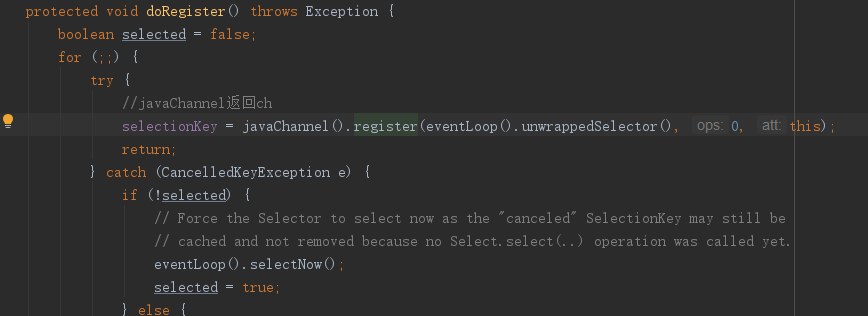
• Unsafe：实现具体的连接与读/写数据，如网络的读/写、链路关闭、发起连接等。命名为Unsafe表示不对外提供使用，并非不安全。



AbstractChannel没有涉及NIO的任何属性和具体方法，包括AbstractUnsafe。AbstractNioChannel有以下3个重要属性：



SelectableChannel是java.nio.SocketChannel和java.nio.ServerSocketChannel公共的抽象类

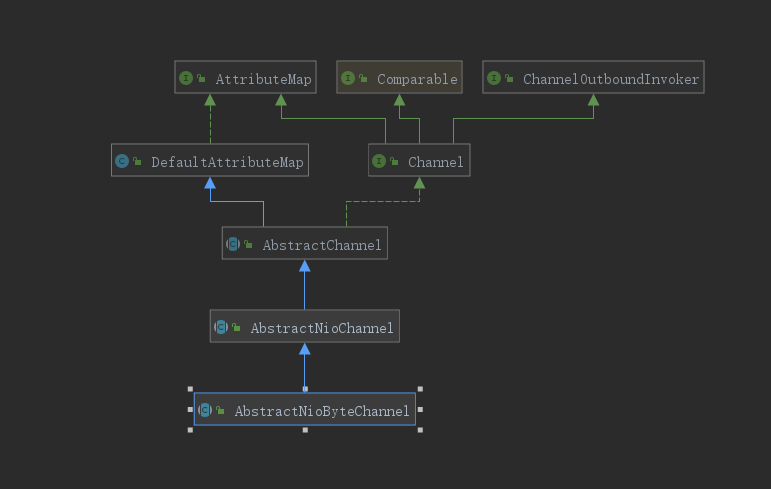


在AbstractNioChannel中有个非常重要的类——AbstractNioUnsafe，是AbstractUnsafe类的NIO实现，主要实现了connect()、flush0()等方法。它还实现了NioUnsafe接口，实现了其finishConnect()、forceFlush()、ch()等方法。其中，forceFlush()与flush0()最终调用的NioSocketChannel的doWrite()方法来完成缓存数据写入Socket的工作，在剖析NioSocketChannel时会详细讲解。connect()和finishConnect()这两个方法只有在Netty客户端中才会使用到。

connect()



AbstractNioChannel拥有NIO的Channel，具备NIO的注册、连接等功能。但I/O的读/写交给了其子类，Netty对I/O的读/写分为POJO对象与ByteBuf和FileRegion，因此在AbstractNioChannel的基础上继续抽象了一层，分为AbstractNioMessageChannel与AbstractNioByteChannel。





属性flushTask为task任务，主要负责刷新发送缓存链表中的数据，由于write的数据没有直接写在Socket中，而是写在了ChannelOutboundBuffer缓存中，所以当调用flush()方法时，会把数据写入Socket中并向网络中发送。因此当缓存中的数据未发送完成时，需要将此任务添加到EventLoop线程中，等待EventLoop线程的再次发送。

doWrite()与doWriteInternal()方法在AbstractChannel的flush0()方法中被调用，主要功能是从ChannelOutboundBuffer缓存中获取待发送的数据，进行循环发送，发送的结果分为以下3种。

（1）发送成功，跳出循环直接返回。

（2）由于TCP缓存区已满，成功发送的字节数为0，跳出循环，并将写操作OP\_WRITE事件添加到选择Key兴趣事件集中。

（3）默认当写了16次数据还未发送完时，把选择Key的OP\_WRITE事件从兴趣事件集中移除，并添加一个flushTask任务，先去执行其他任务，当检测到此任务时再发送。

NioByteUnsafe的read()方法的实现思路大概分为以下3步。

（1）获取Channel的配置对象、内存分配器ByteBufAllocator，并计算内存分配器RecvByteBufAllocator.Handle。

（2）进入for循环。循环体的作用：使用内存分配器获取数据容器ByteBuf，调用doReadBytes()方法将数据读取到容器中，如果本次循环没有读到数据或链路已关闭，则跳出循环。另外，当循环次数达到属性METADATA的defaultMaxMessagesPerRead次数（默认为16）时，也会跳出循环。由于TCP传输会产生粘包问题，因此每次读取都会触发channelRead事件，进而调用业务逻辑处理Handler。

（3）跳出循环后，表示本次读取已完成。调用allocHandle的readComplete()方法，并记录读取记录，用于下次分配合理内存。

AbstractNioMessageChannel写入和读取的数据类型是Object，而不是字节流。在读数据时，AbstractNioMessageChannel数据不存在粘包问题，因此AbstractNioMessageChannel在read()方法中先循环读取数据包，再触发channelRead事件。在写数据时，AbstractNioMessageChannel数据逻辑简单。它把缓存outboundBuffer中的数据包依次写入Channel中。如果Channel写满了，或循环写、默认写的次数为子类Channel属性METADATA中的defaultMaxMessagesPerRead次数，则在Channel的SelectionKey上设置OP*WRITE事件，随后退出，其后OP*WRITE事件处理逻辑和Byte字节流写逻辑一样。

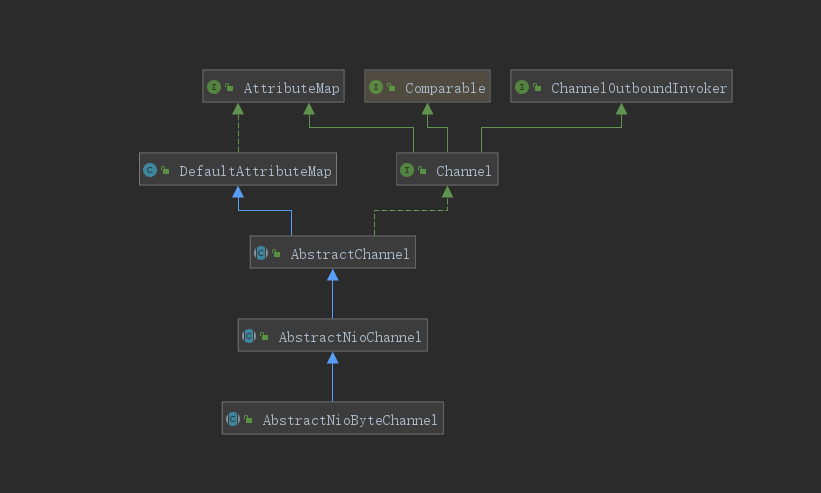
NioSocketChannel是AbstractNioByteChannel的子类，也是io.netty.channel.socket.SocketChannel的实现类。Netty服务的每个Socket连接都会生成一个NioSocketChannel对象。NioSocketChannel在AbstractNioByteChannel的基础上封装了NIO中的SocketChannel，实现了I/O的读/写与连接操作，其核心功能如下。

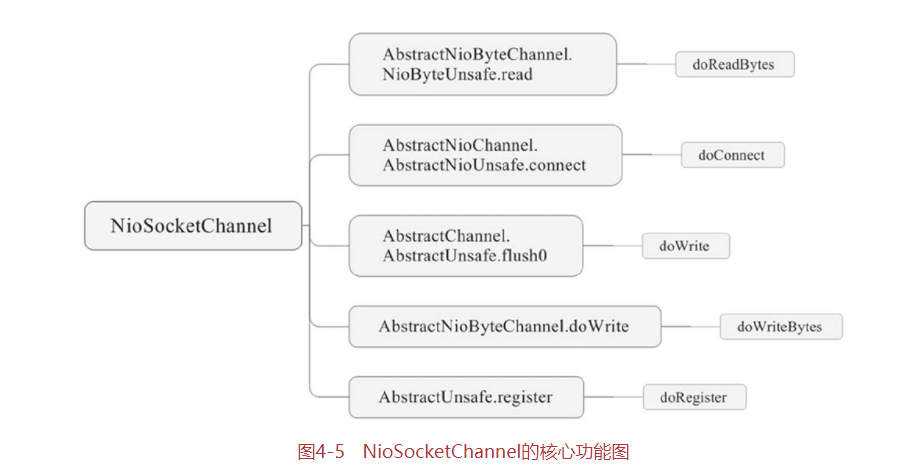
• SocketChannel在NioSocketChannel构造方法中由SelectorProvider.provider().openSocketChannel()创建，提供javaChannel()方法以获取SocketChannel。

• 实现doReadBytes()方法，从SocketChannel中读取数据。

• 重写doWrite()方法、实现doWriteBytes()方法，将数据写入Socket中。

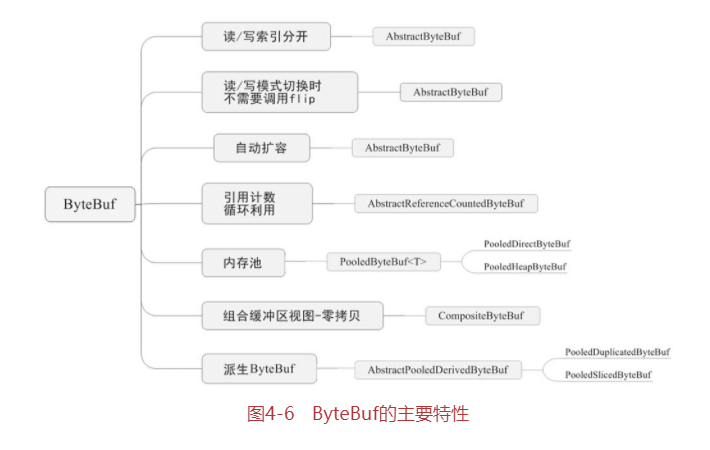
• 实现doConnect()方法和客户端连接。

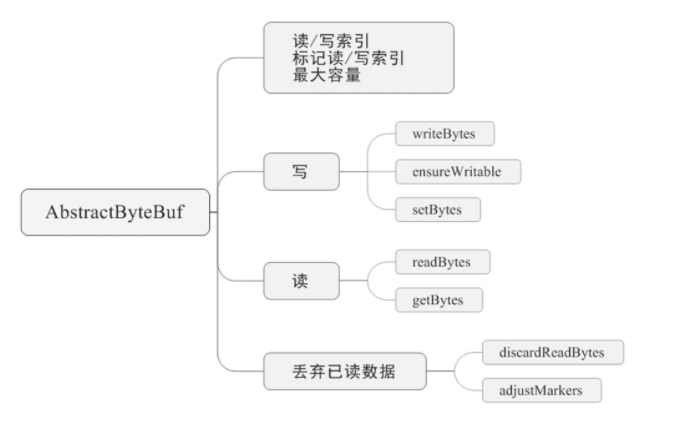




NioServerSocketChannel是AbstractNioMessageChannel的子类，由于NioServerSocketChannel由服务端使用，并且只负责监听Socket的接入，不关心I/O的读/写，所以与NioSocketChannel相比要简单很多。它封装了NIO中的ServerSocketChannel，并通过newSocket()方法打开ServerSocket Channel。它的多路复用器注册与NioSocketChannel的多路复用器注册一样，由父类AbstractNio Channel实现。

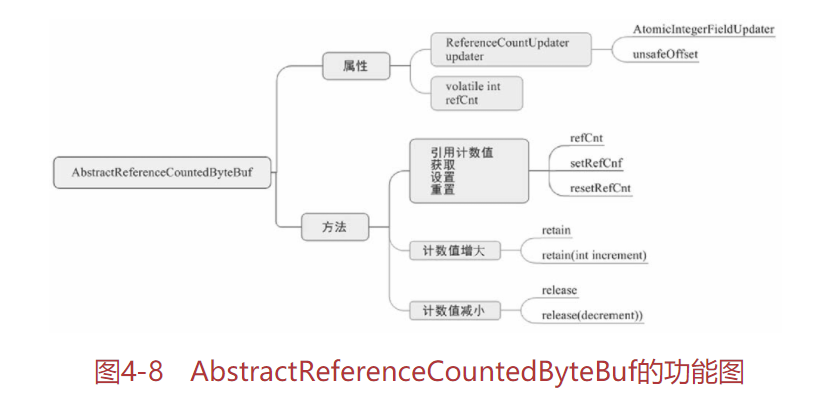
### bytebuf



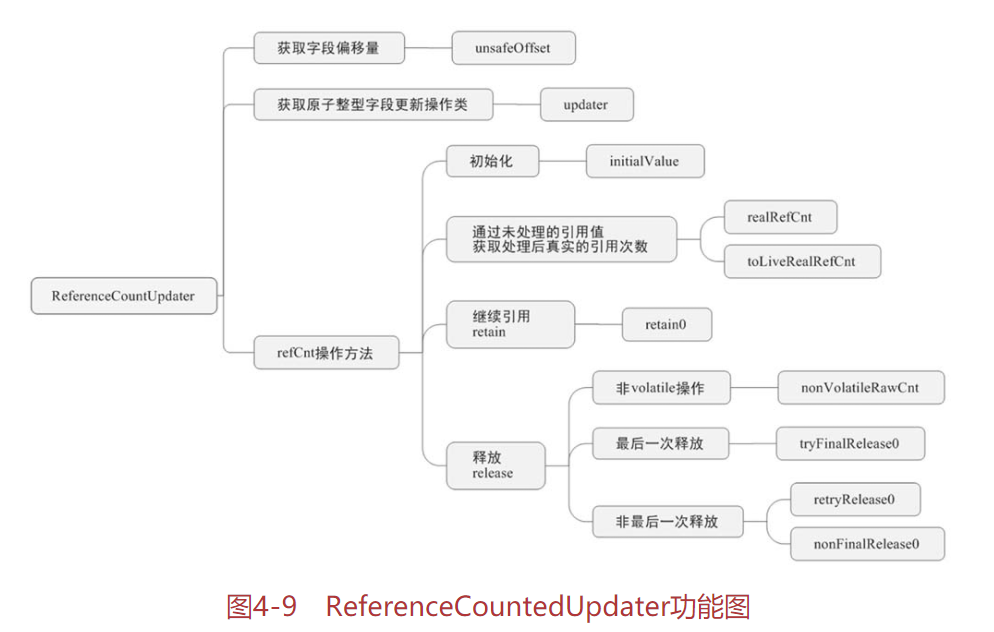


AbstractByteBuf的写操作writeBytes()方法涉及扩容，在扩容时除了合法校验，还需要计算新的容量值，若内存大小为2的整数次幂，则AbstractByteBuf的子类比较好分配内存，因此扩容后的大小必须是2的整数次幂，计算逻辑复杂。

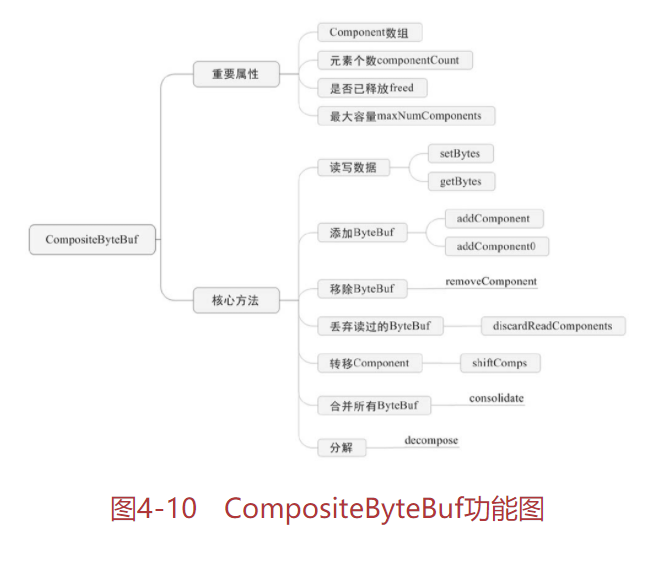
Netty在进行I/O的读/写时使用了堆外直接内存，实现了零拷贝，堆外直接内存Direct Buffer的分配与回收效率要远远低于JVM堆内存上对象的创建与回收速率。Netty使用引用计数法来管理Buffer的引用与释放。Netty采用了内存池设计，先分配一块大内存，然后不断地重复利用这块内存。例如，当从SocketChannel中读取数据时，先在大内存块中切一小部分来使用，由于与大内存共享缓存区，所以需要增加大内存的引用值，当用完小内存后，再将其放回大内存块中，同时减少其引用值。运用到引用计数法的ByteBuf大部分都需要继承AbstractReferenceCountedByteBuf类。该类有个引用值属性——refCnt，其功能大部分与此属性有关。



Netty v4.1.38.Final版本采用了乐观锁方式来修改refCnt，并在修改后进行校验。例如，retain()方法在增加了refCnt后，如果出现了溢出，则回滚并抛异常。在旧版本中，采用的是原子性操作，不断地提前判断，并尝试调用compareAndSet。与之相比，新版本的吞吐量有所提高，但若还是采用refCnt的原有方式，从1开始每次加1或减1，则会引发一些问题，需要重新设计。

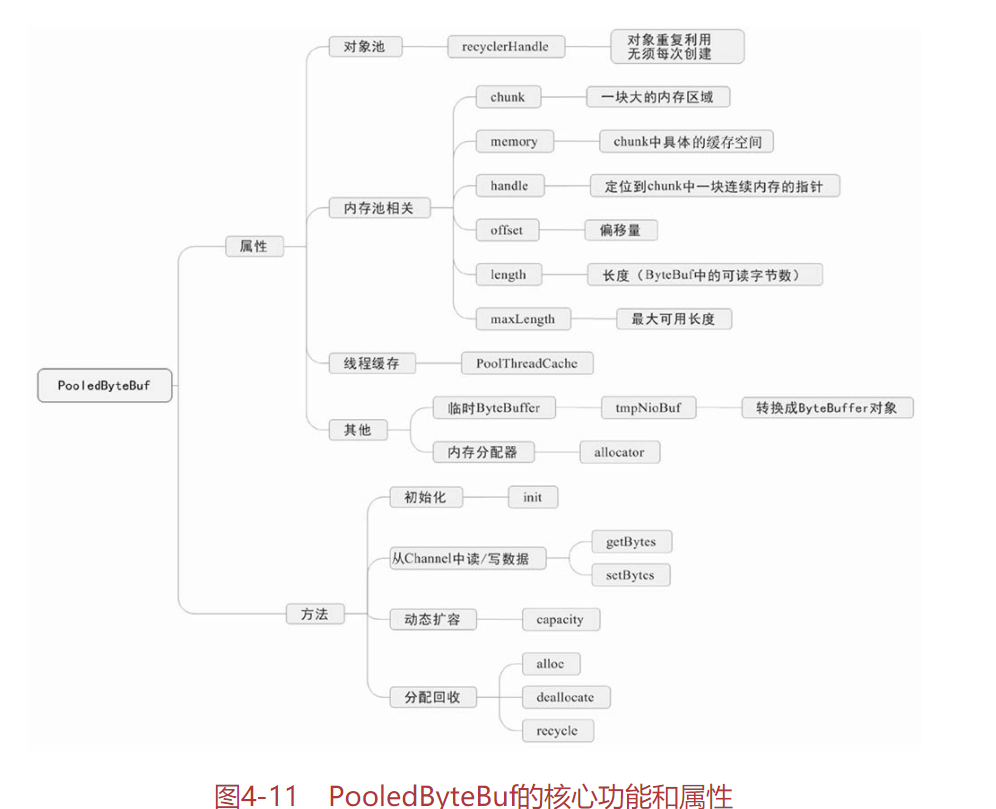


CompositeByteBuf的主要功能是组合多个ByteBuf，对外提供统一的readerIndex和writerIndex。由于它只是将多个ByteBuf的实例组装到一起形成了一个统一的视图，并没有对ByteBuf中的数据进行拷贝，因此也属于Netty零拷贝的一种，主要应用于编码和解码。



PooledByteBuf。这个类继承于AbstractReference CountedByteBuf，其对象主要由内存池分配器PooledByteBufAllocator创建。比较常用的实现类有两种：一种是基于堆外直接内存池构建的PooledDirectByteBuf，是Netty在进行I/O的读/写时的内存分配的默认方式，堆外直接内存可以减少内存数据拷贝的次数；另一种是基于堆内内存池构建的PooledHeapByteBuf。

Netty还使用Java的后门类sun.misc.Unsafe实现了两个缓冲区，即PooledUnsafeDirectByteBuf和PooledUnsafeHeapByteBuf。这个强大的后门类会暴露对象的底层地址，一般不建议使用，Netty为了优化性能引入了Unsafe。



### Channel 与 EventLoop是什么关系

AbstractNioChannel类中

通过javaChannel方法获取nio channel，把channel注册到其eventloop线程的selector上

selectionKey =

javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

### Channel 与 ChannelPipeline是什么关系

<https://www.cnblogs.com/duanxz/p/3724247.html>

ChannelPipeline是一个拦截流经Channel的入站和出站事件的**ChannelHandler实例链**

每一个新创建的 Channel 都将会被分配一个新的 ChannelPipeline。这项关联是永久性 的；Channel 既不能附加另外一个 ChannelPipeline，也不能分离其当前的。在 Netty 组件 的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

EventLoop与EventLoopGroup 是什么关系？

NioEventLoopGroup类主要完成以下3件事。

• 创建一定数量的NioEventLoop线程组并初始化。

• 创建线程选择器chooser。当获取线程时，通过选择器来获取。

• 创建线程工厂并构建线程执行器。

说说Netty 中几个重要的对象是什么，它们之间的关系是什么？

1. Channel
2. ByteBuf
3. ChannelHandler
4. ChannelHandlerContext
5. ChannelPipeline

### Netty 的线程模型是什么

多路复用

Reactor 单线程模型:所有的 IO 操作都在同一个 NIO 线程上面完成

Rector 多线程模型与单线程模型最大的区别就是有一组 NIO 线程处理 IO 操作

主从 Reactor 线程模型的特点是：服务端用于接收客户端连接的不再是个 1 个单独的 NIO 线程，而是一个独立的 NIO 线程池。Acceptor 接收到客户端 TCP 连接请求处理完成后（可能包含接入认证等），将新创建的 SocketChannel 注册到 IO 线程池（sub reactor 线程池）的某个 IO 线程上，由它负责 SocketChannel 的读写和编解码工作。Acceptor 线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端 subReactor 线程池的 IO 线程上，由 IO 线程负责后续的 IO 操作。

服务端： Reactor 的多线程模型

客户端只需要创建一个 EventLoopGroup，因为它不需要独立的线程去监听客户端连接，也没必要通过一个单独的客户端线程去连接服务端

什么是粘包与半包问题?

TCP 是流式协议，消息无边界。

tcp发送数据会进行重新分配，http组装tcp数据

### 粘包的主要原因：

1. 发送方每次写入数据 < 套接字(Socket)缓冲区大小
2. 接收方读取套接字(Socket)缓冲区数据不够及时

半包的主要原因：

1. 发送方每次写入数据 > 套接字(Socket)缓冲区大小
2. 发送的数据大于协议的 MTU (Maximum Transmission Unit，最大传输单元)，因此必须拆包

如何避免粘包与半包问题

封装成帧(Framing)，也就是原本发送消息的单位是缓冲大小，现在换成了帧，这样我们就可以自定义边界了

1. 固定长度：浪费空间
2. 分隔符：当内容本身出现分割符时需要转义，所以无论是发送还是接受，都需要进行整个内容的扫描
3. 专门的length字段：缺点是长度理论上有限制，需要提前限制可能的最大长度从而定义长度占用字节数

### 默认情况 Netty 起多少线程？何时启动

Netty 默认是 CPU 处理器数的两倍，bind 完之后启动。

### Netty 有两种发送消息

Netty 有两种发送消息的方式：

* 直接写入 Channel 中，消息从 ChannelPipeline 当中尾部开始移动；
* 写入和 ChannelHandler 绑定的 ChannelHandlerContext 中，消息从 ChannelPipeline 中的下一个 ChannelHandler 中移动。

### netty服务端如何进行初始化

服务端初始化的步骤 1、创建ServerBootstrap启动辅助类，通过Builder模式进行参数配置； 2、创建并绑定Reactor线程池EventLoopGroup； 4、设置并绑定服务端Channel通道类型； 5、绑定服务端通道数据处理器责任链Handler；

<https://www.jianshu.com/p/f89122e19d4c>

### 何时接受客户端请求

<https://blog.csdn.net/qq_45859054/article/details/102992367>

何时注册接受 Socket 并注册到对应的 EventLoop 管理的 Selector

AbstractChannel抽象类包含以下几个重要属性。• EventLoop：每个Channel对应一条EventLoop线程。• DefaultChannelPipeline：一个Handler的容器，也可以将其理解为一个Handler链。Handler主要处理数据的编/解码和业务逻辑。• Unsafe：实现具体的连接与读/写数据，如网络的读/写、链路关闭、发起连接等。命名为Unsafe表示不对外提供使用，并非不安全。

在AbstractNioChannel中有个非常重要的类——AbstractNioUnsafe，是AbstractUnsafe类的NIO实现，主要实现了connect()、flush0()等方法。

### 客户端如何进行初始化

<https://blog.csdn.net/pzw_0612/article/details/78068194>

<https://xie.infoq.cn/article/a7a2d0faa0117abe6958b81d6>

Netty 客户端初始化时所需的所有内容:

EventLoopGroup：Netty服务端或者客户端，都必须指定EventLoopGroup，客户端指定的是NioEventLoopGroup

Bootstrap: Netty客户端启动类，负责客户端的启动和初始化过程

channel()类型：指定Channel的类型，因为这里是客户端，所以使用的是NioSocketChannel，服务端会使用NioServerSocketChannel

Handler：设置数据的处理器

bootstrap.connect(): 客户端连接netty服务的方法

### 何时创建的 DefaultChannelPipeline

在实例化一个 Channel 时, 会伴随着一个 ChannelPipeline 的实例化, 并且此 Channel 会与这个 ChannelPipeline 相互关联

DefaultChannelPipeline：一个Handler的容器，也可以将其理解为一个Handler链。Handler主要处理数据的编/解码和业务逻辑。

维护了一个以 AbstractChannelHandlerContext 为节点的双向链表

### Netty 支持哪些心跳类型设置

* eaderIdleTime：为读超时时间（即测试端一定时间内未接受到被测试端消息）。
* writerIdleTime：为写超时时间（即测试端一定时间内向被测试端发送消息）。
* allIdleTime：所有类型的超时时间。

### Netty高性能的原因

1. 基于I/O多路复用模型
2. 零拷贝
3. 基于NIO的Buffer
4. 基于内存池的缓冲区重用机制
5. 无锁化的串行设计理念
6. I/O操作的异步处理
7. 提供对protobuf等高性能序列化协议支持
8. 可以对TCP进行更加灵活地配置