# Netty Source

# Reactor模式（具有分发功能的Selector)

* 两种IO多路复用方案:Reactor and Proactor。
* Reactor模式是基于同步I/O的，而Proactor模式是和异步I/O相关的. 在Reactor模式中，事件分离者等待某个事件或者可应用或个操作的状态发生（比如文件描述符可读写，或者是socket可读写),事件分离者就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。
* 类似于生产者消费者模式，但没有缓冲区Queue，每个事件出现后会立即分发给相应的Handler处理。

# 经典Reactor模式

* 在Reactor模式中，包含如下角色：  
  - Reactor 将I/O事件发派给对应的Handler  
  - Acceptor 处理客户端连接请求  
  - Handlers 执行非阻塞读/写
* 一般情况下Reactor指的是一个Acceptor/Reactor线程去循环监听IO事件（ServerSocketChannel在Selector上注册了Accept事件)，如果是连接事件，那么在本线程中建立连接，在建立连接时，SocketChannel在Selector上注册Read事件；
* 如果是读就绪事件，那么将读这个IO任务交给Worker线程池处理。
* 全程只有一个Selector，上面注册了ServerSocket的Accept事件和各个Socket的读事件。

# Netty Reactor模式

* 而Netty中使用的Reactor模式，引入了多Reactor，也即一个主Reactor负责监控所有的连接请求，多个子Reactor负责监控并处理读/写请求，减轻了主Reactor的压力，降低了主Reactor压力太大而造成的延迟。并且每个子Reactor分别属于一个独立的线程，每个成功连接后的Channel的所有操作由同一个线程处理。这样保证了同一请求的所有状态和上下文在同一个线程中，避免了不必要的上下文切换，同时也方便了监控请求响应状态。
* 多Reactor下，mainReactor是一个，subReactor是多个。mainReqactor对应着一个Selector，也是注册了ServerSocket的Accept事件，当接收到连接事件时，接收到Socket，把它交给subReactor，每个subReactor对应着自己的Selector，把Socket的读事件注册到自己的Selector中。
* mainReactor上有一个Selector，注册了ServerSocketChannel的Accept事件；各个subReactor各自对应着自己的Selector，注册了自己对应的SocketChannel的Read事件。

# 分类

* Reactor模式基于事件驱动，特别适合处理海量的I/O事件。Reactor模型主要可以分为：  
  - 单线程模型（所有IO操作都在同一个NIO线程上面完成)  
  - 多线程模型（有一组NIO线程处理IO操作)  
  - 主从多线程模型（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)

## 单线程模型

* Reactor单线程模型，指的是所有的IO操作都在同一个NIO线程上面完成，NIO线程的职责如下：
  + 1)作为NIO服务端，接收客户端的TCP连接；
  + 2)作为NIO客户端，向服务端发起TCP连接；
  + 3)读取通信对端的请求或者应答消息；
  + 4)向通信对端发送消息请求或者应答消息。
* 从架构层面看，一个NIO线程确实可以完成其承担的职责。例如，通过Acceptor类接收客户端的TCP连接请求消息，链路建立成功之后，通过Dispatch将对应的ByteBuffer派发到指定的Handler上进行消息解码。用户线程可以通过消息编码通过NIO线程将消息发送给客户端。
* 对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发的应用场景却不合适，主要原因如下：
  + 1)一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；
  + 2)当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈；
  + 3)可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。
* 为了解决这些问题，演进出了Reactor多线程模型。

## 多线程模型

* 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。
* Reactor多线程模型的特点：
  + 1)有专门一个NIO线程-Acceptor线程用于监听服务端，接收客户端的TCP连接请求；
  + 2)网络IO操作-读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、解码、编码和发送；
  + 3)1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题。
* 在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极个别特殊场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如并发百万客户端连接，或者服务端需要对客户端握手进行安全认证，但是认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型-主从Reactor多线程模型。

### 示例

public class Reactor {  
 public static void main(String[] args) throws IOException {  
 Selector selector = Selector.open();  
 ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
 serverSocketChannel.configureBlocking(false);  
 serverSocketChannel.bind(new InetSocketAddress(9000));  
 serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  
   
 while (true) {  
 if (selector.selectNow() < 0) {  
 continue;  
 }  
 Set<SelectionKey> keys = selector.selectedKeys();  
 Iterator<SelectionKey> iterator = keys.iterator();  
 while (iterator.hasNext()) {  
 SelectionKey key = iterator.next();  
 iterator.remove();  
 if (key.isAcceptable()) {  
 ServerSocketChannel acceptServerSocketChannel = (ServerSocketChannel) key.channel();  
 SocketChannel socketChannel = acceptServerSocketChannel.accept();  
 socketChannel.configureBlocking(false);  
 System.out.println("Accept request from " + socketChannel.getRemoteAddress());  
 SelectionKey readKey = socketChannel.register(selector, SelectionKey.OP\_READ);  
 readKey.attach(new Processor());  
 } else if (key.isReadable()) {  
 Processor processor = (Processor) key.attachment();  
 processor.process(key);  
 }  
 }  
 }  
 }  
}

\*/  
public class Processor {  
 private static final ExecutorService service = Executors.newFixedThreadPool(16);  
  
 public void process(SelectionKey selectionKey) {  
 service.submit(() -> {  
 ByteBuffer buffer = ByteBuffer.allocate(1024);  
 SocketChannel socketChannel = (SocketChannel) selectionKey.channel();  
 int count = socketChannel.read(buffer);  
 if (count < 0) {  
 socketChannel.close();  
 selectionKey.cancel();  
 System.out.println(socketChannel + "\t Read ended");  
 return null;  
 } else if (count == 0) {  
 return null;  
 }  
 System.out.println(socketChannel + "\t Read message " + new String(buffer.array()));  
 return null;  
 });  
 }  
}

## 主从多线程模型

* 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。
* 利用主从NIO线程模型，可以解决1个服务端监听线程无法有效处理所有客户端连接的性能不足问题。
* 它的工作流程总结如下：  
  1.- 从主线程池中随机选择一个Reactor线程作为Acceptor线程，用于绑定监听端口，接收客户端连接；  
  2.- Acceptor线程接收客户端连接请求之后创建新的SocketChannel，将其注册到主线程池的其它Reactor线程上，由其负责接入认证、IP黑白名单过滤、握手等操作；  
  3.- 步骤2完成之后，业务层的链路正式建立，将SocketChannel从主线程池的Reactor线程的多路复用器上摘除，重新注册到Sub线程池的线程上，用于处理I/O的读写操作。

### 示例

public class Reactor {  
  
 public static void main(String[] args) throws IOException {  
 Selector selector = Selector.open();  
 ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();  
 serverSocketChannel.configureBlocking(false);  
 serverSocketChannel.bind(new InetSocketAddress(9000));  
 serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  
  
 int coreNum = Runtime.getRuntime().availableProcessors();  
 Processor[] processors = new Processor[coreNum];  
 for (int i = 0; i < processors.length; i++) {  
 processors[i] = new Processor();  
 }  
 System.out.println("initialized ...");  
 int index = 0;  
 while (selector.select() > 0) {  
 Set<SelectionKey> keys = selector.selectedKeys();  
 for (SelectionKey key : keys) {  
 keys.remove(key);  
 if (key.isAcceptable()) {  
 ServerSocketChannel acceptServerSocketChannel = (ServerSocketChannel) key.channel();  
 SocketChannel socketChannel = acceptServerSocketChannel.accept();  
 socketChannel.configureBlocking(false);  
 System.out.println("Accept request from " + socketChannel.getRemoteAddress());  
 Processor processor = processors[(int) ((index++) / coreNum)];  
 processor.addChannel(socketChannel);  
 }  
 }  
 }  
 }  
}

public class Processor {  
 private static final ExecutorService executor =  
 Executors.newFixedThreadPool(2 \* Runtime.getRuntime().availableProcessors());  
  
 private Selector selector;  
  
 public Processor() throws IOException {  
 this.selector = Selector.open();  
 start();  
 }  
  
 public void addChannel(SocketChannel socketChannel) throws ClosedChannelException {  
 socketChannel.register(this.selector, SelectionKey.OP\_READ);  
 }  
   
 public void start() {  
 executor.submit(() -> {  
 while (true) {  
 if (selector.selectNow() <= 0) {  
 continue;  
 }  
 Set<SelectionKey> keys = selector.selectedKeys();  
 Iterator<SelectionKey> iterator = keys.iterator();  
 while (iterator.hasNext()) {  
 SelectionKey key = iterator.next();  
 iterator.remove();  
 if (key.isReadable()) {

- // 这部分可以交给用户设置的业务线程池处理，在Netty里应该是对应着ChannelHandler的channelRead方法  
 ByteBuffer buffer = ByteBuffer.allocate(1024);  
 SocketChannel socketChannel = (SocketChannel) key.channel();  
 int count = socketChannel.read(buffer);  
 if (count < 0) {  
 socketChannel.close();  
 key.cancel();  
 System.out.println(socketChannel + "\t Read ended");  
 continue;  
 } else if (count == 0) {  
 System.out.println(socketChannel + "\t Message size is 0");  
 continue;  
 } else {  
 System.out.println(socketChannel + "\t Read message " + new String(buffer.array()));  
 }  
 }  
 }  
 }  
 });  
}

}

# Netty Reactor实现

* 服务端启动时创建了两个NioEventLoopGroup，一个是boss，一个是worker。实际上他们是两个独立的Reactor线程池。
* Boss线程池职责如下：
  + （1)接收客户端的连接，初始化Channel参数
  + （2)将链路状态变更时间通知给ChannelPipeline
* Worker线程池作用是：
  + （1)异步读取通信对端的数据报，发送读事件到ChannelPipeline
  + （2)异步发送消息到通信对端，调用ChannelPipeline的消息发送接口
  + （3)执行系统调用Task;
  + （4)执行定时任务Task；
* Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead(Object msg)，只要用户不主动切换线程，一直都是由NioEventLoop调用用户的Handler，期间不进行线程切换，这种串行化设计避免了多线程操作导致的锁竞争，性能角度看是最优的。

# Netty 最佳实践

* Netty是个异步高性能的NIO框架，它并不是个业务运行容器，因此它不需要也不应该提供业务容器和业务线程。合理的设计模式是Netty只负责提供和管理NIO线程，其它的业务层线程模型由用户自己集成，Netty不应该提供此类功能，只要将分层划分清楚，就会更有利于用户集成和扩展。
* 消息队列也可以不设置，Handler直接将Task交给用户设置的业务线程池也是可以的。
  + （1)创建两个NioEventLoopGroup，隔离NIO Acceptor和NIO的IO线程。
  + （2)尽量不要在ChannelHandler中启动用户线程（解码之后，将POJO消息派发到后端的业务线程池除外)。
  + （3)解码要放在NIO线程调用的Handler中，不要放在用户线程中解码。
  + （4)如果IO操作非常简单，不涉及复杂的业务逻辑计算，没有可能导致阻塞的磁盘操作、数据库操作、网络操作等，可以再NIO线程调用的Handler中完成业务逻辑，不要切换到用户线程。
  + （5)如果IO业务操作比较复杂，就不要在NIO线程上完成，因为阻塞可能会导致NIO线程假死，严重降低性能。这时候可以把POJO封装成Task，派发到业务线程池中由业务线程处理，以保证NIO，线程被尽快的释放，处理其余的IO操作。

# Netty 使用

# 为什么选择 Netty

- NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。  
- 需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序。  
- 可靠性能力补齐，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、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。  
- JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK 1.6版本的update18修复了该问题，但是直到JDK 1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有得到根本性解决。

# 简介

* Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如Hadoop的RPC框架Avro就使用了Netty作为底层通信框架，其他还有业界主流的RPC框架，也使用Netty来构建高性能的异步通信能力。
* Netty是一个高性能、异步事件驱动的NIO框架，提供了对TCP、UDP和文件传输的支持，作为一个异步NIO框架，Netty的所有IO操作都是异步（编程模型上的)非阻塞的，通过Future-Listener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。
* 通过对Netty的分析，我们将它的优点总结如下。

- API使用简单，开发门槛低；  
- 功能强大，预置了多种编解码功能，支持多种主流协议；  
- 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；  
- 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；  
- 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；  
- 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；  
- 经历了大规模的商业应用考验，质量得到验证。Netty在互联网、大数据、网络游戏、企业应用、电信软件等众多行业已经得到了成功商用，证明它已经完全能够满足不同行业的商业应用了。

* 正是因为这些优点，Netty逐渐成为了Java NIO编程的首选框架。

# Netty与NIO

* 不选择bio模型我们知道，那么为什么不选择aio模式呢？而还是选择nio模式呢？这是一个值得思考的问题，我就一直很好奇，因为在网络 I/O模型里面介绍的，明显AIO要比NIO模型还要好。
* 为何不使用AIO的官方解释：

# Netty 的使用场景

- 互联网行业：随着网站规模的不断扩大，系统并发访问量也越来越高，传统基于 Tomcat 等 Web 容器的垂直架构已经无法满足需求，需要拆分应用进行服务化，以提高开发和维护效率。从组网情况看，垂直的架构拆分之后，系统采用分布式部署，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。

- 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，非常方便定制和开发私有协议栈。账号登陆服务器、地图服务器之间可以方便的通过 Netty 进行高性能的通信。  
- 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 进行跨节点通信，它的 Netty Service 基于 Netty 框架二次封装实现。  
- 企业软件：企业和 IT 集成需要 ESB，Netty 对多协议支持、私有协议定制的简洁性和高性能是 ESB RPC 框架的首选通信组件。事实上，很多企业总线厂商会选择 Netty 作为基础通信组件，用于企业的 IT 集成。  
- 通信行业：Netty 的异步高性能、高可靠性和高成熟度的优点，使它在通信行业得到了大量的应用。

# NIO BUG

* 原生的 NIO 在 JDK 1.7 版本存在 epoll bug
* 若Selector的轮询结果为空，也没有wakeup或新消息处理，则发生空轮询，CPU使用率100%
* Netty的解决办法
* 对Selector的select操作周期进行统计，每完成一次空的select操作进行一次计数，
* 若在某个周期内连续发生N次空轮询，则触发了epoll死循环bug。
* 重建Selector，判断是否是其他线程发起的重建请求，若不是则将原SocketChannel从旧的Selector上去除注册，重新注册到新的Selector上，并将原来的Selector关闭。

# 缓冲

* Netty 使用自建的 buffer API，而不是使用 NIO 的 ByteBuffer 来表示一个连续的字节序列。与 ByteBuffer 相比这种方式拥有明显的优势。Netty 使用新的 buffer 类型 ByteBuf，被设计为一个可从底层解决 ByteBuffer 问题，并可满足日常网络应用开发需要的缓冲类型。这些很酷的特性包括：
* 如果需要，允许使用自定义的缓冲类型。
* 复合缓冲类型中内置的透明的零拷贝实现。
* 开箱即用的动态缓冲类型，具有像 StringBuffer 一样的动态缓冲能力。
* 不再需要调用的flip()方法。
* 正常情况下具有比 ByteBuffer 更快的响应速度。

## 零拷贝

* 为什么叫零拷贝？因为在数据传输时，最终处理的数据会需要对单个传输层的报文，进行组合或者拆分。NIO原生的ByteBuffer要做到这件事，需要对ByteBuffer内容进行拷贝，产生新的ByteBuffer，而Netty通过提供Composite(组合)和Slice(切分)两种Buffer来实现零拷贝。这部分代码在org.jboss.netty.buffer包中。
* 例如，一个信息可以由两部分组成；header 和 body。在一个模块化的应用，当消息发送出去时，这两部分可以由不同的模块生产和装配。
* 如果你使用的是 ByteBuffer ，你必须要创建一个新的大缓存区用来拷贝这两部分到这个新缓存区中。或者，你可以在 NiO做一个收集写操作，但限制你将复合缓冲类型作为 ByteBuffer 的数组而不是一个单一的缓冲区，打破了抽象，并且引入了复杂的状态管理。此外，如果你不从 NIO channel 读或写，它是没有用的。
* // 复合类型与组件类型不兼容。
* ByteBuffer[] message = new ByteBuffer[] { header, body };
* 通过对比， ByteBuf 不会有警告，因为它是完全可扩展并有一个内置的复合缓冲区。
* // 复合类型与组件类型是兼容的。
* ByteBuf message = Unpooled.wrappedBuffer(header, body);
* // 因此，你甚至可以通过混合复合类型与普通缓冲区来创建一个复合类型。
* ByteBuf messageWithFooter = Unpooled.wrappedBuffer(message, footer);
* // 由于复合类型仍是 ByteBuf，访问其内容很容易，
* //并且访问方法的行为就像是访问一个单独的缓冲区，
* //即使你想访问的区域是跨多个组件。
* //这里的无符号整数读取位于 body 和 footer
* messageWithFooter.getUnsignedInt(
  + messageWithFooter.readableBytes() - footer.readableBytes() - 1);

## 自动容量扩展

* // 一种新的动态缓冲区被创建。在内部，实际缓冲区是被“懒”创建，从而避免潜在的浪费内存空间。
  + ByteBuf b = Unpooled.buffer(4);
* // 当第一个执行写尝试，内部指定初始容量 4 的缓冲区被创建
* b.writeByte('1');
* b.writeByte('2');
* b.writeByte('3');
* b.writeByte('4');
* // 当写入的字节数超过初始容量 4 时，
* //内部缓冲区自动分配具有较大的容量
* b.writeByte('5');

## 更好的性能

* 最频繁使用的缓冲区 ByteBuf 的实现是一个非常薄的字节数组包装器（比如，一个字节)。与 ByteBuffer 不同，它没有复杂的边界和索引检查补偿，因此对于 JVM 优化缓冲区的访问更加简单。更多复杂的缓冲区实现是用于拆分或者组合缓存，并且比 ByteBuffer 拥有更好的性能。

# 统一的IO API

* OIO就是指BIO。
* 传统的 Java I/O API 在应对不同的传输协议时需要使用不同的类型和方法。例如：java.net.Socket 和 java.net.DatagramSocket 它们并不具有相同的超类型，因此，这就需要使用不同的调用方式执行 socket 操作。
* 这种模式上的不匹配使得在更换一个网络应用的传输协议时变得繁杂和困难。由于（Java I/O API)缺乏协议间的移植性，当你试图在不修改网络传输层的前提下增加多种协议的支持，这时便会产生问题。并且理论上讲，多种应用层协议可运行在多种传输层协议之上例如TCP/IP,UDP/IP,SCTP和串口通信。
* 让这种情况变得更糟的是，Java 新的 I/O（NIO)API与原有的阻塞式的I/O（OIO)API 并不兼容，NIO.2(AIO)也是如此。由于所有的API无论是在其设计上还是性能上的特性都与彼此不同，在进入开发阶段，你常常会被迫的选择一种你需要的API。
* 例如，在用户数较小的时候你可能会选择使用传统的 OIO(Old I/O) API，毕竟与 NIO 相比使用 OIO 将更加容易一些。然而，当你的业务呈指数增长并且服务器需要同时处理成千上万的客户连接时你便会遇到问题。这种情况下你可能会尝试使用 NIO，但是复杂的 NIO Selector 编程接口又会耗费你大量时间并最终会阻碍你的快速开发。
* Netty 有一个叫做 Channel 的统一的异步 I/O 编程接口，这个编程接口抽象了所有点对点的通信操作。也就是说，如果你的应用是基于 Netty 的某一种传输实现，那么同样的，你的应用也可以运行在 Netty 的另一种传输实现上。Netty 提供了几种拥有相同编程接口的基本传输实现：
* 基于 NIO 的 TCP/IP 传输 (见 io.netty.channel.nio),
* 基于 OIO 的 TCP/IP 传输 (见 io.netty.channel.oio),
* 基于 OIO 的 UDP/IP 传输, 和
* 本地传输 (见 io.netty.channel.local).
* 切换不同的传输实现通常只需对代码进行几行的修改调整，例如选择一个不同的 ChannelFactory 实现。
* 此外，你甚至可以利用新的传输实现没有写入的优势，只需替换一些构造器的调用方法即可，例如串口通信。而且由于核心 API 具有高度的可扩展性，你还可以完成自己的传输实现。

# 事件驱动

* 一个定义良好并具有扩展能力的事件模型是事件驱动开发的必要条件。Netty 具有定义良好的 I/O 事件模型。由于严格的层次结构区分了不同的事件类型，因此 Netty 也允许你在不破坏现有代码的情况下实现自己的事件类型。这是与其他框架相比另一个不同的地方。很多 NIO 框架没有或者仅有有限的事件模型概念；在你试图添加一个新的事件类型的时候常常需要修改已有的代码，或者根本就不允许你进行这种扩展。
* 在一个 ChannelPipeline 内部一个 ChannelEvent 被一组ChannelHandler 处理。这个管道是 Intercepting Filter (拦截过滤器)模式的一种高级形式的实现，因此对于一个事件如何被处理以及管道内部处理器间的交互过程，你都将拥有绝对的控制力。

# Netty服务器创建过程

public void run() {  
 //两个事件循环器，第一个用于接收客户端连接，第二个用于处理客户端的读写请求  
 //是线程组，持有一组线程  
 EventLoopGroup bossGroup = new NioEventLoopGroup();  
 EventLoopGroup workerGroup = new NioEventLoopGroup();  
 try {  
 //服务器辅助类，用于配置服务器  
 ServerBootstrap bootstrap = new ServerBootstrap();  
 //配置服务器参数  
 bootstrap.group(bossGroup, workerGroup)  
 //使用这种类型的NIO通道，现在是基于TCP协议的  
 .channel(NioServerSocketChannel.class)  
 //对Channel进行初始化，绑定实际的事件处理器，要么实现ChannelHandler接口，要么继承ChannelHandlerAdapter类  
 .childHandler(new ChannelInitializer<SocketChannel>() {  
 protected void initChannel(SocketChannel ch) throws Exception {  
 //编码是其他格式转为字节  
 //解码是从字节转到其他格式  
 //服务器是把先请求转为POJO（解码)，再把响应转为字节（编码)  
 //而客户端是先把请求转为字节（编码)，再把响应转为POJO（解码)  
 ch.pipeline()  
 .addLast(new IdleStateHandler(10, 0, 0))  
 // 将 RPC 请求进行解码（为了处理请求)  
 .addLast(new RPCDecoder(RPCRequest.class))  
 // 将 RPC 响应进行编码（为了返回响应)  
 .addLast(new RPCEncoder(RPCResponse.class))  
 // 处理 RPC 请求  
 .addLast(new RPCServerHandler());  
 }  
 })  
 //服务器配置项  
 //BACKLOG  
 //TCP维护有两个队列，分别称为A和B  
 //客户端发送SYN，服务器接收到后发送SYN ACK，将客户端放入到A队列  
 //客户端接收到后再次发送ACK，服务器接收到后将客户端从A队列移至B队列，服务器的accept返回。  
 //A和B队列长度之和为backlog  
 //当A和B队列长度之和大于backlog时，新的连接会被TCP内核拒绝  
 //注意：backlog对程序的连接数并无影响，影响的只是还没有被accept取出的连接数。  
 .option(ChannelOption.SO\_BACKLOG, 128)  
 //指定发送缓冲区大小  
 .option(ChannelOption.SO\_SNDBUF, 32 \* 1024)  
 //指定接收缓冲区大小  
 .option(ChannelOption.SO\_RCVBUF, 32 \* 1024)  
 //这里的option是针对于上面的NioServerSocketChannel  
 //复杂的时候可能会设置多个Channel  
 .childOption(ChannelOption.SO\_KEEPALIVE, true);  
 //.sync表示是一个同步阻塞执行，普通的Netty的IO操作都是异步执行的  
 //一个ChannelFuture代表了一个还没有发生的I/O操作。这意味着任何一个请求操作都不会马上被执行  
 //Netty强烈建议直接通过添加监听器的方式获取I/O结果，而不是通过同步等待(.sync)的方式  
 //如果用户操作调用了sync或者await方法，会在对应的future对象上阻塞用户线程  
  
 String address = PropertyUtil.getProperty("server.address");  
 if (address == null) {  
 throw new IllegalStateException("server.address未找到");  
 }  
  
 String host = address.split(":")[0];  
 Integer port = Integer.parseInt(address.split(":")[1]);  
 //绑定端口，开始监听  
 //注意这里可以绑定多个端口，每个端口都针对某一种类型的数据（控制消息，数据消息)  
 ChannelFuture future = bootstrap.bind(host, port).sync();  
 log.info("服务器启动");  
  
 registry = new ServiceRegistry();  
 registry.register(address);  
 log.info("服务器向Zookeeper注册完毕");  
  
 //应用程序会一直等待，直到channel关闭  
 future.channel().closeFuture().sync();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 registry.close();  
 workerGroup.shutdownGracefully();  
 bossGroup.shutdownGracefully();  
 }  
}

* 1、NioEventLoopGroup 是用来处理I/O操作的线程池，Netty对 EventLoopGroup 接口针对不同的传输协议提供了不同的实现。在本例子中，需要实例化两个NioEventLoopGroup，通常第一个称为“boss”，用来accept客户端连接，另一个称为“worker”，处理客户端数据的读写操作。
* 2、ServerBootstrap 是启动服务的辅助类，有关socket的参数可以通过ServerBootstrap进行设置。
* 3、这里指定NioServerSocketChannel类初始化channel用来接受客户端请求。
* 4、通常会为新SocketChannel通过添加一些handler，来设置ChannelPipeline。ChannelInitializer 是一个特殊的handler，其中initChannel方法可以为SocketChannel 的pipeline添加指定handler。
* 5、通过绑定端口8080，就可以对外提供服务了。
* 通常情况下，服务端的创建是在用户进程启动的时候进行，因此一般由Main函数或者启动类负责创建，服务端的创建由业务线程负责完成。在创建服务端的时候实例化了2个EventLoopGroup，1个EventLoopGroup实际就是一个EventLoop线程组，负责管理EventLoop的申请和释放。
* EventLoopGroup管理的线程数可以通过构造函数设置，如果没有设置，默认取-Dio.netty.eventLoopThreads，如果该系统参数也没有指定，则为可用的CPU内核数 × 2。
* bossGroup线程组实际就是Acceptor线程池，负责处理客户端的TCP连接请求，如果系统只有一个服务端端口需要监听，则建议bossGroup线程组线程数设置为1。
* workerGroup是真正负责I/O读写操作的线程组，通过ServerBootstrap的group方法进行设置，用于后续的Channel绑定。

# Netty客户端创建过程

* 只有一个eventloopgroup

public RPCClient() {  
 log.info("初始化RPC客户端");  
 this.discovery = new ServiceDiscovery();  
 this.responses = new ConcurrentHashMap<>();  
 this.group = new NioEventLoopGroup();  
 this.bootstrap = new Bootstrap();  
 this.bootstrap.group(group).channel(NioSocketChannel.class)  
 .handler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 public void initChannel(SocketChannel channel) throws Exception {  
 channel.pipeline()  
 .addLast(new IdleStateHandler(0, 0, 5))  
 // 将 RPC 请求进行编码（为了发送请求)  
 .addLast(new RPCEncoder(Message.class, RPCRequest.class))  
 // 将 RPC 响应进行解码（为了处理响应)  
 .addLast(new RPCDecoder(Message.class, RPCResponse.class))  
 // 使用 RpcClient 发送 RPC 请求  
 .addLast(new RPCClientHandler(RPCClient.this,responses));  
 }  
 })  
 .option(ChannelOption.SO\_KEEPALIVE, true);  
  
 try {  
 this.futureChannel = connect();  
 log.info("客户端初始化完毕");  
 } catch (Exception e) {  
 log.error("与服务器的连接出现故障");  
 handleException();  
 }  
}

private Channel connect() throws Exception {  
 log.info("向ZK查询服务器地址中...");  
 String serverAddress = discovery.discover();  
 if (serverAddress == null) {  
 throw new ServerNotAvailableException();  
 }  
 String host = serverAddress.split(":")[0];  
 Integer port = Integer.parseInt(serverAddress.split(":")[1]);  
 ChannelFuture future = bootstrap.connect(host, port).sync();  
 log.info("客户端已连接");  
 return future.channel();  
}

1.- 由用户线程负责初始化客户端资源，发起连接操作；  
2.- 如果连接成功，将SocketChannel注册到IO线程组的NioEventLoop线程中，监听读操作位；  
3.- 如果没有立即连接成功，将SocketChannel注册到IO线程组的NioEventLoop线程中，监听连接操作位；  
4.- 连接成功之后，修改监听位为READ，但是不需要切换线程。

# Netty 概念

# EventLoop

* 运行任务来处理在连接的生命周期内发生的事件是任何网络框架的基本功能。与之相应的编
* 程上的构造通常被称为事件循环—一个Netty 使用了interface io.netty.channel.EventLoop 来适配的术语。
* 事件/任务的执行顺序 事件和任务是以先进先出（FIFO)的顺序执行的。这样可以通过保证字节内容总是按正确的顺序被处理，消除潜在的数据损坏的可能性。

## 继承体系

* 一个EventLoop 将由一个永远都不会改变的Thread 驱动，同时任务
* （Runnable 或者Callable)可以直接提交给EventLoop 实现，以立即执行或者调度执行。
* 根据配置和可用核心的不同，可能会创建多个EventLoop 实例用以优化资源的使用，并且单个EventLoop 可能会被指派用于服务多个Channel。
* 事件的性质通常决定了它将被如何处理；它可能将数据从网络栈中传递到你的应用程序中，
* 或者进行逆向操作，或者执行一些截然不同的操作。但是事件的处理逻辑必须足够的通用和灵活，以处理所有可能的用例。因此，在Netty 4 中，所有的I/O操作和事件都由已经被分配给了EventLoop的那个Thread来处理。

## 任务调度

* ScheduledExecutorService 的实现具有局限性，例如，事实上作为线程池管理的一部
* 分，将会有额外的线程创建。如果有大量任务被紧凑地调度，那么这将成为一个瓶颈。Netty 通过Channel 的EventLoop 实现任务调度解决了这一问题。

## EventLoopGroup

* Netty的服务端使用了两个EventLoopGroup，而第一个EventLoopGroup通常只有一个EventLoop，通常叫做bossGroup，负责客户端的连接请求，然后打开Channel，交给后面的EventLoopGroup（称为workerGroup)中的一个EventLoop来负责这个Channel上的所有读写事件，一个Channel只会被一个EventLoop处理，而一个EventLoop可能会被分配给多个Channel来负责上面的事件，当然，Netty不仅支持NI/O，还支持OI/O，所以两者的EventLoop分配方式有所区别。
* 服务器端的 ServerSocketChannel 只绑定到了 bossGroup 中的一个线程, 因此在调用 Java NIO 的 Selector.select 处理客户端的连接请求时, 实际上是在一个线程中的, 所以对只有一个服务的应用来说, bossGroup 设置多个线程是没有什么作用的, 反而还会造成资源浪费。

### NIO（EventLoop多路复用，与Channel一对多)

* 在NIO非阻塞模式下，Netty将负责为每个Channel分配一个EventLoop，一旦一个EventLoop被分配给了一个Channel，那么在它的整个生命周期中都使用这个EventLoop，但是多个Channel将可能共享一个EventLoop，所以和Thread相关的ThreadLocal的使用就要特别注意，因为有多个Channel在使用该Thread来处理读写时间。在阻塞IO模式下，考虑到一个Channel将会阻塞，所以不太可能将一个EventLoop共用于多个Channel之间，所以，每一个Channel都将被分配一个EventLoop，并且反过来也成立，也就是一个EventLoop将只会被绑定到一个Channel上来处理这个Channel上的读写事件。无论是非阻塞模式还是阻塞模式，一个Channel都将会保证一个Channel上的所有读写事件都只会在一个EventLoop上被处理。
* 每一个连接对应一个 Channel（多路指多个 Channel，复用指多个连接复用了一个线程或少量线程，在 Netty 指 EventLoop)，一个 Channel 对应唯一的 ChannelPipeline，多个 Handler 串行的加入到 Pipeline 中，每个 Handler 关联唯一的 ChannelHandlerContext。
* 在默认的情况下，ChannelHandler会把对它的方法的调用转发给链中的下一个ChannelHandler实例链。因此，如果exceptionCaught方法没有被该链中的某处实现，那么所接收到的异常将会被传递到ChannelPipeline的尾端并被记录。为此，你的应用程序应该提供至少有一个实现了exceptionCaught方法的ChannelHandler。
* 总结：
  + 1)一个EventLoopGroup包含一个或者多个EventLoop
  + 2)一个EventLoop在其生命周期内只和一个线程绑定
  + 3)所有由EventLoop处理的IO事件都将在它专有的线程中被处理
  + 4)一个Channel在它的生命周期只注册于一个EventLoop
  + 5)一个EventLoop可能会被分配给一个或多个Channel
  + 6)一个 Channel 对应唯一的 ChannelPipeline
  + 7)多个 ChannelHandler 串行地加入到 ChannelPipeline 中

### OIO（EventLoop与Channel一对一)

* 得到的保证是每个Channel 的I/O 事件都将只会被一个Thread（用于支撑该Channel 的EventLoop 的那个Thread)处理。

## 线程模型

* EventLoopGroup类似于一个线程池，EventLoop类似于一个线程的封装。
* EventLoop继承了Java的ScheduledExecutorService，也就是调度线程池，所以，EventLoop应当有ScheduledExecutorService提供的所有功能。那为什么需要继承ScheduledExecutorService呢，也就是为什么需要延时调度功能，那是因为，在Netty中，有可能用户线程和Netty的I/O线程同时操作网络资源，而为了减少并发锁竞争，Netty将用户线程的任务包装成Netty的task，然后向Netty的I/O任务一样去执行它们。有些时候我们需要延时执行任务，或者周期性执行任务，那么就需要调度功能。这是Netty在设计上的考虑，为我们极大的简化的编程方法。
* Netty线程模型的卓越性能取决于对于当前执行的Thread的身份的确定，也就是说，确定
* 它是否是分配给当前Channel以及它的EventLoop的那一个线程。
* 如果（当前)调用线程正是支撑EventLoop 的线程，那么所提交的代码块将会被（直接)
* 执行。否则，EventLoop 将调度该任务以便稍后执行，并将它放入到内部队列中。当EventLoop下次处理它的事件时，它会执行队列中的那些任务/事件。这也就解释了任何的Thread 是如何与Channel 直接交互而无需在ChannelHandler 中进行额外同步的。
* 注意，每个EventLoop 都有它自已的任务队列，独立于任何其他的EventLoop。
* “永远不要将一个长时间运行的任务放入到执行队列中，因为它将阻塞需要在同一线程上执行的任其他任务。”如果必须要进行阻塞调用或者执行长时间运行的任务，我们建议使用一个专门的EventExecutor。

# Channel

* 每个Channel 都将会被分配一个ChannelPipeline 和ChannelConfig。
* ChannelConfig 包含了该Channel 的所有配置设置，并且支持热更新。由于特定的传输可能
* 具有独特的设置，所以它可能会实现一个ChannelConfig 的子类型。
* 由于Channel 是独一无二的，所以为了保证顺序将Channel 声明为java.lang.
* Comparable 的一个子接口。因此，如果两个不同的Channel 实例都返回了相同的散列码，那么AbstractChannel 中的compareTo()方法的实现将会抛出一个Error。
* ChannelPipeline 持有所有将应用于入站和出站数据以及事件的ChannelHandler 实例，这些ChannelHandler 实现了应用程序用于处理状态变化以及数据处理的逻辑。
* ChannelHandler 的典型用途包括：
* 将数据从一种格式转换为另一种格式；
* 提供异常的通知；
* 提供Channel 变为活动的或者非活动的通知；
* 提供当Channel 注册到EventLoop 或者从EventLoop 注销时的通知；
* 提供有关用户自定义事件的通知。
* ChannelPipeline使用了责任链模式。
* Netty 的Channel 实现是线程安全的，因此你可以存储一个到Channel 的引用，并且每当
* 你需要向远程节点写数据时，都可以使用它，即使当时许多线程都在使用它。
* 需要注意的是，消息将会被保证按顺序发送。

## Channel的生命周期

# ChannelFuture

* 正如我们已经解释过的那样，Netty 中所有的I/O 操作都是异步的。因为一个操作可能不会
* 立即返回，所以我们需要一种用于在之后的某个时间点确定其结果的方法。为此，Netty 提供了
* ChannelFuture 接口，其addListener()方法注册了一个ChannelFutureListener，以
* 便在某个操作完成时（无论是否成功)得到通知。

# ChannelHandler&ChannelPipeline

* 从应用程序开发人员的角度来看，Netty 的主要组件是ChannelHandler，它充当了所有
* 处理入站和出站数据的应用程序逻辑的容器。这是可行的，因为ChannelHandler 的方法是
* 由网络事件（其中术语“事件”的使用非常广泛)触发的。事实上，ChannelHandler 可专
* 门用于几乎任何类型的动作，例如将数据从一种格式转换为另外一种格式，或者处理转换过程中所抛出的异常。

## ChannelHandler的生命周期

## ChannelInboundHandler 接口

* ChannelInboundHandler 是一个你将会经常实现的子接口。这种类型的ChannelHandler 接收入站事件和数据，这些数据随后将会被你的应用程序的业务逻辑所处理。当你要给连接的客户端发送响应时，也可以从ChannelInboundHandler 冲刷数据。你的应用程序的业务逻辑通常驻留在一个或者多个ChannelInboundHandler 中。
* ChannelPipeline 提供了ChannelHandler 链的容器，并定义了用于在该链上传播入站和出站事件流的API。当Channel 被创建时，它会被自动地分配到它专属的ChannelPipeline。
* ChannelHandler 安装到ChannelPipeline 中的过程如下所示：  
  1.- 一个ChannelInitializer的实现被注册到了ServerBootstrap中；  
  2.- 当ChannelInitializer.initChannel()方法被调用时，ChannelInitializer将在ChannelPipeline 中安装一组自定义的ChannelHandler；  
  3.- ChannelInitializer 将它自己从ChannelPipeline 中移除。
* ChannelHandler 是专为支持广泛的用途而设计的，可以将它看作是处理往来Channel-
* Pipeline 事件（包括数据)的任何代码的通用容器。图3-2 说明了这一点，其展示了从ChannelHandler 派生的ChannelInboundHandler 和ChannelOutboundHandler 接口。
* 使得事件流经ChannelPipeline 是ChannelHandler 的工作，它们是在应用程序的初
* 始化或者引导阶段被安装的。这些对象接收事件、执行它们所实现的处理逻辑，并将数据传递给链中的下一个ChannelHandler。它们的执行顺序是由它们被添加的顺序所决定的。实际上，被我们称为ChannelPipeline 的是这些ChannelHandler 的编排顺序。
* 如果一个消息或者任何其他的入站事件被读取，那么它会从ChannelPipeline 的头部开始流动，并被传递给第一个ChannelInboundHandler。这个ChannelHandler 不一定会实际地修改数据，具体取决于它的具体功能，在这之后，数据将会被传递给链中的下一个ChannelInboundHandler。最终，数据将会到达ChannelPipeline 的尾端，届时，所有处理就都结束了。
* 数据的出站运动（即正在被写的数据)在概念上也是一样的。在这种情况下，数据将从ChannelOutboundHandler 链的尾端开始流动，直到它到达链的头部为止。在这之后，出站数据将会到达网络传输层，这里显示为Socket。通常情况下，这将触发一个写操作。
* 鉴于出站操作和入站操作是不同的，你可能会想知道如果将两个类别的ChannelHandler都混合添加到同一个ChannelPipeline 中会发生什么。虽然ChannelInboundHandle 和ChannelOutboundHandle 都扩展自ChannelHandler，但是Netty 能区分ChannelInboundHandler
* 实现和ChannelOutboundHandler 实现，并确保数据只会在具有相同定向类型的两个ChannelHandler 之间传递。
* 当ChannelHandler 被添加到ChannelPipeline 时，它将会被分配一个ChannelHandler-Context，其代表了ChannelHandler 和ChannelPipeline 之间的绑定。虽然这个对象可以被用于获取底层的Channel，但是它主要还是被用于写出站数据。
* 在Netty 中，有两种发送消息的方式。你可以直接写到Channel 中，也可以写到和ChannelHandler相关联的ChannelHandlerContext 对象中。前一种方式将会导致消息从ChannelPipeline 的尾端开始流动，而后者将导致消息从ChannelPipeline 中的下一个ChannelHandler 开始流动。
* 编程时注意：
* 1、ChannelInboundHandler之间的传递，通过调用 ctx.fireChannelRead(msg) 实现；调用ctx.write(msg) 将传递到ChannelOutboundHandler。
* 2、ctx.write()方法执行后，需要调用flush()方法才能令它立即执行。
* 3、ChannelOutboundHandler 在注册的时候需要放在最后一个ChannelInboundHandler之前，否则将无法传递到ChannelOutboundHandler。
* 4、Handler的消费处理放在最后一个处理。
* Netty 以适配器类的形式提供了大量默认的ChannelHandler 实现，其旨在简化应用程序处理逻辑的开发过程。你已经看到了，ChannelPipeline中的每个ChannelHandler将负责把事件转发到链中的下一个ChannelHandler。这些适配器类（及它们的子类)将自动执行这个操作，所以你可以只重写那些你想要特殊处理的方法和事件。
* 有一些适配器类可以将编写自定义的ChannelHandler 所需要的努力降到最低限度，因为它们提供了定义在对应接口中的所有方法的默认实现。
* 下面这些是编写自定义ChannelHandler 时经常会用到的适配器类：
*  ChannelHandlerAdapter
*  ChannelInboundHandlerAdapter
*  ChannelOutboundHandlerAdapter
*  ChannelDuplexHandler

### SimpleChannelInboundHandler

* 最常见的情况是，你的应用程序会利用一个ChannelHandler 来接收解码消息，并对该数据应用业务逻辑。要创建一个这样的ChannelHandler，你只需要扩展基类SimpleChannelInboundHandler，其中T 是你要处理的消息的Java 类型。在这个ChannelHandler 中，你将需要重写基类的一个或者多个方法，并且获取一个到ChannelHandlerContext 的引用，这个引用将作为输入参数传递给ChannelHandler 的所有方法。在这种类型的ChannelHandler 中， 最重要的方法是channelRead0(ChannelHandlerContext,T)。除了要求不要阻塞当前的I/O 线程之外，其具体实现完全取决于你。
* 当某个ChannelInboundHandler 的实现重写channelRead()方法时，它将负责显式地
* 释放与池化的ByteBuf 实例相关的内存。Netty 为此提供了一个实用方法ReferenceCount-
* Util.release()。
* Netty 将使用WARN 级别的日志消息记录未释放的资源，使得可以非常简单地在代码中发现违规的实例。但是以这种方式管理资源可能很繁琐。一个更加简单的方式是使用Simple-
* ChannelInboundHandler。由于SimpleChannelInboundHandler 会自动释放资源，所以你不应该存储指向任何消息的引用供将来使用，因为这些引用都将会失效。

## ChannelOutboundHandler 接口

* 出站操作和数据将由ChannelOutboundHandler 处理。它的方法将被Channel、Channel-
* Pipeline 以及ChannelHandlerContext 调用。
* ChannelOutboundHandler 的一个强大的功能是可以按需推迟操作或者事件，这使得可
* 以通过一些复杂的方法来处理请求。例如，如果到远程节点的写入被暂停了，那么你可以推迟冲刷操作并在稍后继续。
* ChannelPromise与ChannelFuture ChannelOutboundHandler中的大部分方法都需要一个
* ChannelPromise参数，以便在操作完成时得到通知。ChannelPromise是ChannelFuture的一个子类，其定义了一些可写的方法，如setSuccess()和setFailure()，从而使ChannelFuture不可变。

## ChannelHandlerAdapter

* 你可以使用ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter
* 类作为自己的ChannelHandler 的起始点。这两个适配器分别提供了ChannelInboundHandler和ChannelOutboundHandler 的基本实现。通过扩展抽象类ChannelHandlerAdapter，它们获得了它们共同的超接口ChannelHandler 的方法。
* ChannelHandlerAdapter 还提供了实用方法isSharable()。如果其对应的实现被标
* 注为Sharable，那么这个方法将返回true，表示它可以被添加到多个ChannelPipeline
* 中。
* 在ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter 中所
* 提供的方法体调用了其相关联的ChannelHandlerContext 上的等效方法，从而将事件转发到
* 了ChannelPipeline 中的下一个ChannelHandler 中。
* 你要想在自己的ChannelHandler 中使用这些适配器类，只需要简单地扩展它们，并且重
* 写那些你想要自定义的方法。

## 避免内存泄露

* class ResourceLeakDetector它将对你应用程序的缓冲区分配做大约1%的采样来检测内存泄露，其利用了JDK 提供的PhantomReference类来实现这一点。
* 实现ChannelInboundHandler.channelRead()和ChannelOutboundHandler.write()
* 方法时，应该如何使用这个诊断工具来防止泄露呢？

## ChannelPipeline

* 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。根据事件的起源，事件将会被ChannelInboundHandler 或者ChannelOutboundHandler处理。随后，通过调用ChannelHandlerContext 实现，它将被转发给同一超类型的下一个ChannelHandler。
* ChannelHandlerContext使得ChannelHandler能够和它的ChannelPipeline以及其他的
* ChannelHandler 交互。ChannelHandler 可以通知其所属的ChannelPipeline 中的下一个
* ChannelHandler，甚至可以动态修改它所属的ChannelPipeline。
* 在ChannelPipeline 传播事件时，它会测试ChannelPipeline 中的下一个ChannelHandler 的类型是否和事件的运动方向相匹配。如果不匹配，ChannelPipeline 将跳过该ChannelHandler 并前进到下一个，直到它找到和该事件所期望的方向相匹配的为止。
* ChannelHandler 可以通过添加、删除或者替换其他的ChannelHandler 来实时地修改
* ChannelPipeline 的布局。（它也可以将它自己从ChannelPipeline 中移除。)这是Channel-
* Handler 最重要的能力之一，所以我们将仔细地来看看它是如何做到的。

### ChannelHandler 的执行和阻塞

* 通常ChannelPipeline 中的每一个ChannelHandler 都是通过它的EventLoop（I/O 线程)来处理传递给它的事件的。所以至关重要的是不要阻塞这个线程，因为这会对整体的I/O 处理产生负面的影响。
* 但有时可能需要与那些使用阻塞API 的遗留代码进行交互。对于这种情况，ChannelPipeline 有一些接受一个EventExecutorGroup 的add()方法。如果一个事件被传递给一个自定义的EventExecutorGroup，它将被包含在这个EventExecutorGroup 中的某个EventExecutor 所处理，从而被从该Channel 本身的EventLoop 中移除。对于这种用例，Netty 提供了一个叫DefaultEventExecutorGroup 的默认实现。

### 触发事件

* 总结一下：
* ChannelPipeline 保存了与Channel 相关联的ChannelHandler；
* ChannelPipeline 可以根据需要，通过添加或者删除ChannelHandler 来动态地修改；
* ChannelPipeline 有着丰富的API 用以被调用，以响应入站和出站事件。

## ChannelHandlerContext 接口

* ChannelHandlerContext 代表了ChannelHandler 和ChannelPipeline 之间的关
* 联，每当有ChannelHandler 添加到ChannelPipeline 中时，都会创建ChannelHandler-
* Context。ChannelHandlerContext 的主要功能是管理它所关联的ChannelHandler 和在
* 同一个ChannelPipeline 中的其他ChannelHandler 之间的交互。
* ChannelHandlerContext 有很多的方法，其中一些方法也存在于Channel 和ChannelPipeline 本身上，但是有一点重要的不同。如果调用Channel 或者ChannelPipeline 上的这
* 些方法，它们将沿着整个ChannelPipeline 进行传播。而调用位于ChannelHandlerContext
* 上的相同方法，则将从当前所关联的ChannelHandler 开始，并且只会传播给位于该
* ChannelPipeline 中的下一个能够处理该事件的ChannelHandler。
* 当使用ChannelHandlerContext 的API 的时候，请牢记以下两点：
* ChannelHandlerContext 和ChannelHandler 之间的关联（绑定)是永远不会改
* 变的，所以缓存对它的引用是安全的；
* 相对于其他类的同名方法，ChannelHandler Context的方法将产生更短的事件流，应该尽可能地利用这个特性来获得最大的性能。
* 为什么会想要从ChannelPipeline 中的某个特定点开始传播事件呢？
* 为了减少将事件传经对它不感兴趣的ChannelHandler 所带来的开销。
* 为了避免将事件传经那些可能会对它感兴趣的ChannelHandler。
* 要想调用从某个特定的ChannelHandler 开始的处理过程，必须获取到在（Channel-
* Pipeline)该ChannelHandler 之前的ChannelHandler 所关联的ChannelHandler-
* Context。这个ChannelHandlerContext 将调用和它所关联的ChannelHandler 之后的
* ChannelHandler。

## ChannelHandler 和ChannelHandlerContext 的高级用法

* 可以通过调用ChannelHandlerContext 上的pipeline()方法来获得被封闭的ChannelPipeline 的引用。这使得运行时得以操作ChannelPipeline 的ChannelHandler，我们可以利用这一点来实现一些复杂的设计。例如，你可以通过将ChannelHandler 添加到ChannelPipeline 中来实现动态的协议切换。
* 另一种高级的用法是缓存到ChannelHandlerContext 的引用以供稍后使用，这可能会发
* 生在任何的ChannelHandler 方法之外，甚至来自于不同的线程。
* 因为一个ChannelHandler 可以从属于多个ChannelPipeline，所以它也可以绑定到多
* 个ChannelHandlerContext 实例。对于这种用法指在多个ChannelPipeline 中共享同一
* 个ChannelHandler，对应的ChannelHandler 必须要使用@Sharable 注解标注；否则，
* 试图将它添加到多个ChannelPipeline 时将会触发异常。显而易见，为了安全地被用于多个
* 并发的Channel（即连接)，这样的ChannelHandler 必须是线程安全的。
* 只应该在确定了你的ChannelHandler 是线程安全的时才使用@Sharable 注解。
* 为何要共享同一个ChannelHandler 在多个ChannelPipeline中安装同一个ChannelHandler的一个常见的原因是用于收集跨越多个Channel 的统计信息。

## 异常处理

### 处理入站异常

* 如果在处理入站事件的过程中有异常被抛出，那么它将从它在ChannelInboundHandler
* 里被触发的那一点开始流经ChannelPipeline。要想处理这种类型的入站异常，你需要在你
* 的ChannelInboundHandler 实现中重写下面的方法。

public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception

* 因为异常将会继续按照入站方向流动（就像所有的入站事件一样)，所以实现了前面所示逻
* 辑的ChannelInboundHandler 通常位于ChannelPipeline 的最后。这确保了所有的入站
* 异常都总是会被处理，无论它们可能会发生在ChannelPipeline 中的什么位置。
* 你应该如何响应异常，可能很大程度上取决于你的应用程序。你可能想要关闭Channel（和
* 连接)，也可能会尝试进行恢复。如果你不实现任何处理入站异常的逻辑（或者没有消费该异常)，那么Netty将会记录该异常没有被处理的事实。
* 总结一下：
* ChannelHandler.exceptionCaught()的默认实现是简单地将当前异常转发给ChannelPipeline 中的下一个ChannelHandler；
* 如果异常到达了ChannelPipeline 的尾端，它将会被记录为未被处理；
* 要想定义自定义的处理逻辑，你需要重写exceptionCaught()方法。然后你需要决定
* 是否需要将该异常传播出去。

### 处理出站异常

* 用于处理出站操作中的正常完成以及异常的选项，都基于以下的通知机制。
* 每个出站操作都将返回一个ChannelFuture。注册到ChannelFuture 的Channel-
* FutureListener 将在操作完成时被通知该操作是成功了还是出错了。
* 几乎所有的ChannelOutboundHandler 上的方法都会传入一个ChannelPromise
* 的实例。作为ChannelFuture 的子类，ChannelPromise 也可以被分配用于异步通
* 知的监听器。但是，ChannelPromise 还具有提供立即通知的可写方法：
* ChannelPromise setSuccess();
* Chan nelPromise setFailure(Throwable cause);
* 添加ChannelFutureListener 只需要调用ChannelFuture 实例上的addListener
* (ChannelFutureListener)方法，并且有两种不同的方式可以做到这一点。其中最常用的方式是，
* 调用出站操作（如write()方法)所返回的ChannelFuture 上的addListener()方法。
* 第二种方式是将ChannelFutureListener 添加到即将作为参数传递给ChannelOutboundHandler的方法的ChannelPromise。
* ChannelPromise 的可写方法
* 通过调用ChannelPromise 上的setSuccess()和setFailure()方法，可以使一个操作的状
* 态在ChannelHandler 的方法返回给其调用者时便即刻被感知到。
* 为何选择一种方式而不是另一种呢？对于细致的异常处理，你可能会发现，在调用出站操
* 作时添加ChannelFutureListener 更合适。而对于一般的异常处理，你可能会发现，代码清单6-14 所示的自定义的ChannelOutboundHandler 实现的方式更加的简单。
* 如果你的ChannelOutboundHandler 本身抛出了异常会发生什么呢？在这种情况下，
* Netty 本身会通知任何已经注册到对应ChannelPromise 的监听器。

# Bootstrap

* 有两种类型的引导：一种用于客户端（简单地称为Bootstrap)，而另一种（ServerBootstrap)用于服务器。无论你的应用程序使用哪种协议或者处理哪种类型的数据，唯一决定它使用哪种引导类的是它是作为一个客户端还是作为一个服务器。
* 实际上，ServerBootstrap 类也可以只使用一个EventLoopGroup，此时其将在两个场景下共用同一个EventLoopGroup。
* 这两种类型的引导类之间的第一个区别已经讨论过了：ServerBootstrap 将绑定到一个
* 端口，因为服务器必须要监听连接，而Bootstrap 则是由想要连接到远程节点的客户端应用程序所使用的。
* 第二个区别可能更加明显。引导一个客户端只需要一个EventLoopGroup，但是一个
* ServerBootstrap 则需要两个（也可以是同一个实例)。为什么呢？
* 因为服务器需要两组不同的Channel。第一组将只包含一个ServerChannel，代表服务
* 器自身的已绑定到某个本地端口的正在监听的套接字。而第二组将包含所有已创建的用来处理传入客户端连接（对于每个服务器已经接受的连接都有一个)的Channel。
* 与ServerChannel 相关联的EventLoopGroup 将分配一个负责为传入连接请求创建
* Channel 的EventLoop。一旦连接被接受，第二个EventLoopGroup 就会给它的Channel分配一个EventLoop。

## Bootstrap客户端

* Bootstrap 类被用于客户端或者使用了无连接协议的应用程序中。表8-1 提供了该类的一
* 个概览，其中许多方法都继承自AbstractBootstrap 类。

## ServerBoostrap服务器

* 表8-2 中列出了一些在表8-1 中不存在的方法：childHandler()、
* childAttr()和childOption()。这些调用支持特别用于服务器应用程序的操作。具体来说，
* ServerChannel 的实现负责创建子Channel，这些子Channel 代表了已被接受的连接。因此，负责引导ServerChannel 的ServerBootstrap 提供了这些方法，以简化将设置应用到已被接受的子Channel 的ChannelConfig 的任务。
* protected abstract void initChannel(C ch) throws Exception;
* 这个方法提供了一种将多个ChannelHandler 添加到一个ChannelPipeline 中的简便
* 方法。你只需要简单地向Bootstrap 或ServerBootstrap 的实例提供你的Channel-
* Initializer 实现即可，并且一旦Channel 被注册到了它的EventLoop 之后，就会调用你的
* initChannel()版本。在该方法返回之后，ChannelInitializer 的实例将会从Channel-
* Pipeline 中移除它自己。

## 关闭

* 最重要的是，你需要关闭EventLoopGroup，它将处理任何挂起的事件和任务，并且随后
* 释放所有活动的线程。这就是调用EventLoopGroup.shutdownGracefully()方法的作用。
* 这个方法调用将会返回一个Future，这个Future 将在关闭完成时接收到通知。需要注意的是，shutdownGracefully()方法也是一个异步的操作，所以你需要阻塞等待直到它完成，或者向所返回的Future 注册一个监听器以在关闭完成时获得通知。

# 编解码器

* 当你通过Netty 发送或者接收一个消息的时候，就将会发生一次数据转换。入站消息会被解
* 码；也就是说，从字节转换为另一种格式，通常是一个Java 对象。如果是出站消息，则会发生相反方向的转换：它将从它的当前格式被编码为字节。这两种方向的转换的原因很简单：网络数据总是一系列的字节。
* 对应于特定的需要，Netty 为编码器和解码器提供了不同类型的抽象类。例如，你的应用程
* 序可能使用了一种中间格式，而不需要立即将消息转换成字节。你将仍然需要一个编码器，但是
* 它将派生自一个不同的超类。为了确定合适的编码器类型，你可以应用一个简单的命名约定。
* 通常来说，这些基类的名称将类似于ByteToMessageDecoder 或MessageToByteEncoder。
* 严格地说，其他的处理器也可以完成编码器和解码器的功能。但是，正如有用来简化ChannelHandler 的创建的适配器类一样，所有由Netty 提供的编码器/解码器适配器类都实现了ChannelOutboundHandler 或者ChannelInboundHandler 接口。
* 你将会发现对于入站数据来说，channelRead 方法/事件已经被重写了。对于每个从入站
* Channel 读取的消息，这个方法都将会被调用。随后，它将调用由预置解码器所提供的decode()方法，并将已解码的字节转发给ChannelPipeline 中的下一个ChannelInboundHandler。
* 出站消息的模式是相反方向的：编码器将消息转换为字节，并将它们转发给下一个
* ChannelOutboundHandler。

## 抽象类ByteToMessageDecoder

* 将字节解码为消息（或者另一个字节序列)是一项如此常见的任务，以至于Netty 为它提供了一个抽象的基类：ByteToMessageDecoder。由于你不可能知道远程节点是否会一次性地发送一个完整的消息，所以这个类会对入站数据进行缓冲，直到它准备好处理。
* 引用计数需要特别的注意。对于编码器和解码器来说，其过程
* 也是相当的简单：一旦消息被编码或者解码，它就会被ReferenceCountUtil.release(message)调用自动释放。如果你需要保留引用以便稍后使用，那么你可以调用ReferenceCountUtil.retain(message)方法。这将会增加该引用计数，从而防止该消息被释放。
* 由于Netty 是一个异步框架，所以需要在字节可以解码之前在内存中缓冲它们。因此，不能
* 让解码器缓冲大量的数据以至于耗尽可用的内存。为了解除这个常见的顾虑，Netty 提供了
* TooLongFrameException 类，其将由解码器在帧超出指定的大小限制时抛出。
* 为了避免这种情况，你可以设置一个最大字节数的阈值，如果超出该阈值，则会导致抛出一
* 个TooLongFrameException（随后会被ChannelHandler.exceptionCaught()方法捕
* 获)。然后，如何处理该异常则完全取决于该解码器的用户。某些协议（如HTTP)可能允许你返回一个特殊的响应。而在其他的情况下，唯一的选择可能就是关闭对应的连接。

## 抽象类MessageToByteEncoder

* 这个类只有一个方法，而解码器有两个。原因是解码器通常需要在Channel 关闭之后产生最后一个消息（因此也就有了decodeLast()方法)。这显然不适用于编码器的场景——在连接被关闭之后仍然产生一个消息是毫无意义的。

# 空闲的连接和超时

# TCP粘包与半包

* 粘包：
* 应用程序Write写入的字节大小大于套接口发送缓冲区的大小。
* 进行MSS大小的TCP分段。
* 以太网帧的payload大于MTU进行IP分片。

## 基于分隔符的协议

## 基于长度的协议

# 序列化

# 传输方式

* 注意Epoll！
* Linux作为高性能网络编程的平台，其重要性与日俱增，这催生了大量先进特性的开发，其
  + 中包括epoll——一个高度可扩展的I/O事件通知特性。这个API自Linux内核版本2.5.44（2002)被引入，提供了比旧的POSIX select和poll系统调用更好的性能，同时现在也是Linux上非阻塞网络编程的事实标准。Linux JDK NIO API使用了这些epoll调用。
* Netty为Linux提供了一组NIO API，其以一种和它本身的设计更加一致的方式使用epoll，并且以一种更加轻量的方式使用中断。
* Netty是如何能够使用和用于异步传输相同的API来支持OIO的呢。
* 答案就是，Netty利用了SO\_TIMEOUT这个Socket标志，它指定了等待一个I/O操作完成的最大毫秒数。如果操作在指定的时间间隔内没有完成，则将会抛出一个SocketTimeout Exception。Netty将捕获这个异常并继续处理循环。在EventLoop下一次运行时，它将再次尝试。这实际上也是类似于Netty这样的异步框架能够支持OIO的唯一方式。

# ByteBuf

* Java NIO 提供了ByteBuffer 作为它的字节容器，但是这个类使用起来过于复杂，而且也有些繁琐。
* Netty 的ByteBuffer 替代品是ByteBuf，一个强大的实现，既解决了JDK API 的局限性，
* 又为网络应用程序的开发者提供了更好的API。
* Netty 的数据处理API 通过两个组件暴露——abstract class ByteBuf 和interface
* ByteBufHolder。
* 下面是一些ByteBuf API 的优点：
* 它可以被用户自定义的缓冲区类型扩展；
* 通过内置的复合缓冲区类型实现了透明的零拷贝；
* 容量可以按需增长（类似于JDK 的StringBuilder)；
* 在读和写这两种模式之间切换不需要调用ByteBuffer 的flip()方法；
* 读和写使用了不同的索引；
* 支持方法的链式调用；
* 支持引用计数；
* 支持池化。

## 索引操作

* ByteBuf 维护了两个不同的索引：一个用于读取，一个用于写入。当你从ByteBuf 读取时，
* 它的readerIndex 将会被递增已经被读取的字节数。同样地，当你写入ByteBuf 时，它的
* writerIndex 也会被递增。图5-1 展示了一个空ByteBuf 的布局结构和状态。
* 虽然ByteBuf 同时具有读索引和写索引，但是JDK 的ByteBuffer 却只有一个索引，这
* 也就是为什么必须调用flip()方法来在读模式和写模式之间进行切换的原因。
* 如果打算读取字节直到readerIndex 达到和writerIndex 同样的值时会发生什么。在那时，你将会到达“可以读取的”数据的末尾。就如同试图读取超出数组末尾的数据一样，试图读取超出该点的数据将会触发一个IndexOutOfBoundsException。
* 名称以read 或者write 开头的ByteBuf 方法，将会推进其对应的索引，而名称以set 或
* 者get 开头的操作则不会。后面的这些方法将在作为一个参数传入的一个相对索引上执行操作。
* 可以指定ByteBuf 的最大容量。试图移动写索引（即writerIndex)超过这个值将会触
* 发一个异常。（默认的限制是Integer.MAX\_VALUE。)
* 通过调用discardReadBytes()方法，可以丢弃它们并回收空间。这个分段的初始大小为0，存储在readerIndex 中，会随着read 操作的执行而增加（get\*操作不会移动readerIndex)。
* 可丢弃字节分段中的空间已经变为可写的了。注意，在调用discardReadBytes()之后，对
* 可写分段的内容并没有任何的保证。
* 虽然你可能会倾向于频繁地调用discardReadBytes()方法以确保可写分段的最大化，但是
* 请注意，这将极有可能会导致内存复制，因为可读字节（图中标记为CONTENT 的部分)必须被移动到缓冲区的开始位置。我们建议只在有真正需要的时候才这样做，例如，当内存非常宝贵的时候。
* JDK 的InputStream 定义了mark(int readlimit)和reset()方法，这些方法分别
* 被用来将流中的当前位置标记为指定的值，以及将流重置到该位置。
* 同样，可以通过调用markReaderIndex()、markWriterIndex()、resetWriterIndex()
* 和resetReaderIndex()来标记和重置ByteBuf 的readerIndex 和writerIndex。这些和
* InputStream 上的调用类似，只是没有readlimit 参数来指定标记什么时候失效。
* 也可以通过调用readerIndex(int)或者writerIndex(int)来将索引移动到指定位置。试
* 图将任何一个索引设置到一个无效的位置都将导致一个IndexOutOfBoundsException。
* 可以通过调用clear()方法来将readerIndex 和writerIndex 都设置为0。注意，这
* 并不会清除内存中的内容。
* 查找操作：indexOf或者ByteProcessor#process

## 缓冲区

### 堆缓冲区

* 最常用的ByteBuf 模式是将数据存储在JVM 的堆空间中。这种模式被称为支撑数组
* （backing array)，它能在没有使用池化的情况下提供快速的分配和释放。这种方式非常适合于有遗留的数据需要处理的情况。

### 直接缓冲区

* 直接缓冲区是另外一种ByteBuf 模式。我们期望用于对象创建的内存分配永远都来自于堆
* 中，但这并不是必须的——NIO 在JDK 1.4 中引入的ByteBuffer 类允许JVM 实现通过本地调用来分配内存。这主要是为了避免在每次调用本地I/O 操作之前（或者之后)将缓冲区的内容复制到一个中间缓冲区（或者从中间缓冲区把内容复制到缓冲区)。

### 复合缓冲区

* 为多个ByteBuf 提供一个聚合视图。在这里你可以根据需要添加或者删除ByteBuf 实例。
* Netty 通过一个ByteBuf 子类——CompositeByteBuf——实现了这个模式，它提供了一
* 个将多个缓冲区表示为单个合并缓冲区的虚拟表示。
* 警告：CompositeByteBuf 中的ByteBuf 实例可能同时包含直接内存分配和非直接内存分配。
* 如果其中只有一个实例，那么对CompositeByteBuf 上的hasArray()方法的调用将返回该组
* 件上的hasArray()方法的值；否则它将返回false。

### 派生缓冲区

* 派生缓冲区为ByteBuf 提供了以专门的方式来呈现其内容的视图。这类视图是通过以下方
* 法被创建的：
* duplicate()；
* slice()；
* slice(int, int)；
* Unpooled.unmodifiableBuffer(…)；
* order(ByteOrder)；
* readSlice(int)。
* 每个这些方法都将返回一个新的ByteBuf 实例，它具有自己的读索引、写索引和标记
* 索引。其内部存储和JDK 的ByteBuffer 一样也是共享的。这使得派生缓冲区的创建成本
* 是很低廉的，但是这也意味着，如果你修改了它的内容，也同时修改了其对应的源实例，所以要小心。
* ByteBuf 复制 如果需要一个现有缓冲区的真实副本，请使用copy()或者copy(int, int)方
* 法。不同于派生缓冲区，由这个调用所返回的ByteBuf 拥有独立的数据副本。

## 读写操作

* get()和set()操作，从给定的索引开始，并且保持索引不变；
* read()和write()操作，从给定的索引开始，并且会根据已经访问过的字节数对索引进行调整。

## ByteBufHolder

* 我们经常发现，除了实际的数据负载之外，我们还需要存储各种属性值。HTTP 响应便是一个很好的例子，除了表示为字节的内容，还包括状态码、cookie 等。
* 为了处理这种常见的用例，Netty 提供了ByteBufHolder。ByteBufHolder 也为Netty 的
* 高级特性提供了支持，如缓冲区池化，其中可以从池中借用ByteBuf，并且在需要时自动释放。
* ByteBufHolder 只有几种用于访问底层数据和引用计数的方法。

## 分配

### ByteBufAllocator 接口

* 为了降低分配和释放内存的开销，Netty 通过interface ByteBufAllocator 实现了
* （ByteBuf 的)池化，它可以用来分配我们所描述过的任意类型的ByteBuf 实例。使用池化是特定于应用程序的决定，其并不会以任何方式改变ByteBuf API（的语义)。
* 可以通过Channel（每个都可以有一个不同的ByteBufAllocator 实例)或者绑定到
* ChannelHandler 的ChannelHandlerContext 获取一个到ByteBufAllocator 的引用。
* Netty提供了两种ByteBufAllocator的实现：PooledByteBufAllocator和Unpooled-
* ByteBufAllocator。前者池化了ByteBuf的实例以提高性能并最大限度地减少内存碎片。此实现使用了一种称为jemalloc的已被大量现代操作系统所采用的高效方法来分配内存。后者的实现不池化ByteBuf实例，并且在每次它被调用时都会返回一个新的实例。
* 虽然Netty默认使用了PooledByteBufAllocator，但这可以很容易地通过Channel-
* Config API或者在引导你的应用程序时指定一个不同的分配器来更改

### Unpooled 缓冲区

* 可能某些情况下，你未能获取一个到ByteBufAllocator 的引用。对于这种情况，Netty 提
* 供了一个简单的称为Unpooled 的工具类，它提供了静态的辅助方法来创建未池化的ByteBuf实例。

### ByteBufUtil 类

* ByteBufUtil 提供了用于操作ByteBuf 的静态的辅助方法。因为这个API 是通用的，并
* 且和池化无关，所以这些方法已然在分配类的外部实现。
* 这些静态方法中最有价值的可能就是hexdump()方法，它以十六进制的表示形式打印
* ByteBuf 的内容。这在各种情况下都很有用，例如，出于调试的目的记录ByteBuf 的内容。十六进制的表示通常会提供一个比字节值的直接表示形式更加有用的日志条目，此外，十六进制的版本还可以很容易地转换回实际的字节表示。
* 另一个有用的方法是boolean equals(ByteBuf, ByteBuf)，它被用来判断两个ByteBuf
* 实例的相等性。如果你实现自己的ByteBuf 子类，你可能会发现ByteBufUtil 的其他有用方法。

## 引用计数

* 引用计数是一种通过在某个对象所持有的资源不再被其他对象引用时，释放该对象所持有的资源来优化内存使用和性能的技术。Netty 在第4 版中为ByteBuf 和ByteBufHolder 引入了
* 引用计数技术，它们都实现了interface ReferenceCounted。
* 引用计数背后的想法并不是特别的复杂；它主要涉及跟踪到某个特定对象的活动引用的数
* 量。一个ReferenceCounted 实现的实例将通常以活动的引用计数为1 作为开始。只要引用计数大于0，就能保证对象不会被释放。当活动引用的数量减少到0 时，该实例就会被释放。注意，虽然释放的确切语义可能是特定于实现的，但是至少已经释放的对象应该不可再用了。
* 引用计数对于池化实现（如PooledByteBufAllocator)来说是至关重要的，它降低了内存分配的开销。
* 试图访问一个已经被释放的引用计数的对象，将会导致一个IllegalReferenceCount-
* Exception。
* 注意，一个特定的（ReferenceCounted 的实现)类，可以用它自己的独特方式来定义它
* 的引用计数规则。例如，我们可以设想一个类，其release()方法的实现总是将引用计数设为零，而不用关心它的当前值，从而一次性地使所有的活动引用都失效。

## 零拷贝

1.- Netty 提供了 CompositeByteBuf 类, 它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了各个 ByteBuf 之间的拷贝.  
2.- 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, 进而避免了拷贝操作.  
3.- ByteBuf 支持 slice 操作, 因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf, 避免了内存的拷贝.  
4.- 通过 FileRegion 包装的FileChannel.tranferTo 实现文件传输, 可以直接将文件缓冲区的数据发送到目标 Channel, 避免了传统通过循环 write 方式导致的内存拷贝问题.

### CompositeByteBuf

* 除了上面直接使用 CompositeByteBuf 类外, 我们还可以使用 Unpooled.wrappedBuffer 方法, 它底层封装了 CompositeByteBuf 操作, 因此使用起来更加方便:

# 客户端启动过程

1.- EventLoopGroup: 不论是服务器端还是客户端, 都必须指定 EventLoopGroup. 在这个例子中, 指定了 NioEventLoopGroup, 表示一个 NIO 的EventLoopGroup.  
2.- ChannelType: 指定 Channel 的类型. 因为是客户端, 因此使用了 NioSocketChannel.  
3.- Handler: 设置数据的处理器.

- 1) NioEventLoopGroup#constructor

1.- EventLoopGroup(其实是MultithreadEventExecutorGroup) 内部维护一个类型为 EventExecutor children 数组, 其大小是 nThreads, 这样就构成了一个线程池  
2.- 如果我们在实例化 NioEventLoopGroup 时, 如果指定线程池大小, 则 nThreads 就是指定的值, 反之是处理器核心数 \* 2  
3.- MultithreadEventExecutorGroup 中会调用 newChild 抽象方法来初始化 children 数组  
4.- 抽象方法 newChild 是在 NioEventLoopGroup 中实现的, 它返回一个 NioEventLoop 实例.  
5.- NioEventLoop 属性:  
6.- SelectorProvider provider 属性: NioEventLoopGroup 构造器中通过 SelectorProvider.provider() 获取一个 SelectorProvider  
7.- Selector selector 属性: NioEventLoop 构造器中通过调用通过 selector = provider.openSelector() 获取一个 selector 对象.

public NioEventLoopGroup() {  
 this(0);  
}

public NioEventLoopGroup(int nThreads) {  
 this(nThreads, (Executor) null);  
}

public NioEventLoopGroup(int nThreads, Executor executor) {  
 this(nThreads, executor, SelectorProvider.provider());  
}

public NioEventLoopGroup(  
 int nThreads, Executor executor, final SelectorProvider selectorProvider) {  
 this(nThreads, executor, selectorProvider, DefaultSelectStrategyFactory.INSTANCE);  
}

public NioEventLoopGroup(int nThreads, Executor executor, final SelectorProvider selectorProvider,  
 final SelectStrategyFactory selectStrategyFactory) {  
 super(nThreads, executor, selectorProvider, selectStrategyFactory, RejectedExecutionHandlers.reject());  
}

* protected MultithreadEventLoopGroup(int nThreads, Executor executor, Object... args) {  
  super(nThreads == 0 ? DEFAULT*EVENT*LOOP\_THREADS : nThreads, executor, args);  
  }

private static final int DEFAULT\_EVENT\_LOOP\_THREADS;  
  
static {  
 DEFAULT\_EVENT\_LOOP\_THREADS = Math.max(1, SystemPropertyUtil.getInt(  
 "io.netty.eventLoopThreads", NettyRuntime.availableProcessors() \* 2));  
  
 if (logger.isDebugEnabled()) {  
 logger.debug("-Dio.netty.eventLoopThreads: {}", DEFAULT\_EVENT\_LOOP\_THREADS);  
 }  
}

* protected MultithreadEventExecutorGroup(int nThreads, Executor executor, Object... args) {  
  this(nThreads, executor, DefaultEventExecutorChooserFactory.INSTANCE, args);  
  }

public abstract class MultithreadEventExecutorGroup extends AbstractEventExecutorGroup {  
  
 private final EventExecutor[] children;  
 private final Set<EventExecutor> readonlyChildren;  
 private final AtomicInteger terminatedChildren = new AtomicInteger();  
 private final Promise<?> terminationFuture = new DefaultPromise(GlobalEventExecutor.INSTANCE);  
 private final EventExecutorChooserFactory.EventExecutorChooser chooser;

* }

/\*\*  
 \* Create a new instance.  
 \*  
 \* @param nThreads the number of threads that will be used by this instance.  
 \* @param executor the Executor to use, or {@code null} if the default should be used.  
 \* @param chooserFactory the {@link EventExecutorChooserFactory} to use.  
 \* @param args arguments which will passed to each {@link #newChild(Executor, Object...)} call  
 \*/  
protected MultithreadEventExecutorGroup(int nThreads, Executor executor,  
 EventExecutorChooserFactory chooserFactory, Object... args) {  
 if (nThreads <= 0) {  
 throw new IllegalArgumentException(String.format("nThreads: %d (expected: > 0)", nThreads));  
 }  
  
 if (executor == null) {  
 executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());  
 }  
  
 children = new EventExecutor[nThreads];  
  
 for (int i = 0; i < nThreads; i ++) {  
 boolean success = false;  
 try {  
 children[i] = newChild(executor, args);  
 success = true;  
 } catch (Exception e) {  
 // TODO: Think about if this is a good exception type  
 throw new IllegalStateException("failed to create a child event loop", e);  
 } finally {  
 if (!success) {  
 for (int j = 0; j < i; j ++) {  
 children[j].shutdownGracefully();  
 }  
  
 for (int j = 0; j < i; j ++) {  
 EventExecutor e = children[j];  
 try {  
 while (!e.isTerminated()) {  
 e.awaitTermination(Integer.MAX\_VALUE, TimeUnit.SECONDS);  
 }  
 } catch (InterruptedException interrupted) {  
 // Let the caller handle the interruption.  
 Thread.currentThread().interrupt();  
 break;  
 }  
 }  
 }  
 }  
 }  
  
 chooser = chooserFactory.newChooser(children);  
  
 final FutureListener<Object> terminationListener = new FutureListener<Object>() {  
 @Override  
 public void operationComplete(Future<Object> future) throws Exception {  
 if (terminatedChildren.incrementAndGet() == children.length) {  
 terminationFuture.setSuccess(null);  
 }  
 }  
 };  
  
 for (EventExecutor e: children) {  
 e.terminationFuture().addListener(terminationListener);  
 }  
  
 Set<EventExecutor> childrenSet = new LinkedHashSet<EventExecutor>(children.length);  
 Collections.addAll(childrenSet, children);  
 readonlyChildren = Collections.unmodifiableSet(childrenSet);  
}

- 1.1) ThreadPerTaskExecutor#constructor（EventLoopGroup所依赖的)

* protected ThreadFactory newDefaultThreadFactory() {  
  return new DefaultThreadFactory(getClass());  
  }

public final class ThreadPerTaskExecutor implements Executor {  
 private final ThreadFactory threadFactory;  
  
 public ThreadPerTaskExecutor(ThreadFactory threadFactory) {  
 if (threadFactory == null) {  
 throw new NullPointerException("threadFactory");  
 }  
 this.threadFactory = threadFactory;  
 }  
  
 @Override  
 public void execute(Runnable command) {  
 threadFactory.newThread(command).start();  
 }  
}

- 1.2) NioEventLoopGroup#newChild(executor, args)

* protected EventLoop newChild(Executor executor, Object... args) throws Exception {  
  return new NioEventLoop(this, executor, (SelectorProvider) args[0],  
   ((SelectStrategyFactory) args[1]).newSelectStrategy(), (RejectedExecutionHandler) args[2]);  
  }
  + 1.2.1) NioEventLoop#constructor

private Selector selector;  
private Selector unwrappedSelector;  
private SelectedSelectionKeySet selectedKeys;  
  
private final SelectorProvider provider;  
  
/\*\*  
 \* Boolean that controls determines if a blocked Selector.select should  
 \* break out of its selection process. In our case we use a timeout for  
 \* the select method and the select method will block for that time unless  
 \* waken up.  
 \*/  
private final AtomicBoolean wakenUp = new AtomicBoolean();  
  
private final SelectStrategy selectStrategy;  
  
private volatile int ioRatio = 50;  
private int cancelledKeys;  
private boolean needsToSelectAgain;

* NioEventLoop(NioEventLoopGroup parent, Executor executor, SelectorProvider selectorProvider,  
   SelectStrategy strategy, RejectedExecutionHandler rejectedExecutionHandler) {  
  super(parent, executor, false, DEFAULT*MAX*PENDING\_TASKS, rejectedExecutionHandler);  
  if (selectorProvider == null) {  
   throw new NullPointerException("selectorProvider");  
  }  
  if (strategy == null) {  
   throw new NullPointerException("selectStrategy");  
  }  
  provider = selectorProvider;  
  final SelectorTuple selectorTuple = openSelector();  
  selector = selectorTuple.selector;  
  unwrappedSelector = selectorTuple.unwrappedSelector;  
  selectStrategy = strategy;  
  }
  + 1.2.1.1) SingleThreadEventLoop#constructor
* protected SingleThreadEventLoop(EventLoopGroup parent, Executor executor,  
   boolean addTaskWakesUp, int maxPendingTasks,  
   RejectedExecutionHandler rejectedExecutionHandler) {  
  super(parent, executor, addTaskWakesUp, maxPendingTasks, rejectedExecutionHandler);  
  tailTasks = newTaskQueue(maxPendingTasks);  
  }
  + 1.2.1.1.1) SingleThreadEventExecutor#constructor
* protected SingleThreadEventExecutor(EventExecutorGroup parent, Executor executor, boolean addTaskWakesUp, int maxPendingTasks, RejectedExecutionHandler rejectedHandler) {  
  super(parent);  
  this.addTaskWakesUp = addTaskWakesUp;  
  this.maxPendingTasks = Math.max(16, maxPendingTasks);  
  this.executor = ObjectUtil.checkNotNull(executor, "executor");  
  taskQueue = newTaskQueue(this.maxPendingTasks);  
  rejectedExecutionHandler = ObjectUtil.checkNotNull(rejectedHandler, "rejectedHandler");  
  }
* protected AbstractScheduledEventExecutor(EventExecutorGroup parent) {  
  super(parent);  
  }
* protected AbstractEventExecutor(EventExecutorGroup parent) {  
  this.parent = parent;  
  }
  + 1.2.1.1.2) SingleThreadEventExecutor#newTaskQueue

private final Queue<Runnable> tailTasks;

* protected Queue newTaskQueue(int maxPendingTasks) {  
  return new LinkedBlockingQueue(maxPendingTasks);  
  }

#### 1.2.1.2) openSelector

private SelectorTuple openSelector() {  
 final Selector unwrappedSelector;  
 try {  
 unwrappedSelector = provider.openSelector();  
 } catch (IOException e) {  
 throw new ChannelException("failed to open a new selector", e);  
 }  
  
 if (DISABLE\_KEYSET\_OPTIMIZATION) {  
 return new SelectorTuple(unwrappedSelector);  
 }  
  
 final SelectedSelectionKeySet selectedKeySet = new SelectedSelectionKeySet();  
  
 Object maybeSelectorImplClass = AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 try {  
 return Class.forName(  
 "sun.nio.ch.SelectorImpl",  
 false,  
 PlatformDependent.getSystemClassLoader());  
 } catch (Throwable cause) {  
 return cause;  
 }  
 }  
 });  
  
 if (!(maybeSelectorImplClass instanceof Class) ||  
 // ensure the current selector implementation is what we can instrument.  
 !((Class<?>) maybeSelectorImplClass).isAssignableFrom(unwrappedSelector.getClass())) {  
 if (maybeSelectorImplClass instanceof Throwable) {  
 Throwable t = (Throwable) maybeSelectorImplClass;  
 logger.trace("failed to instrument a special java.util.Set into: {}", unwrappedSelector, t);  
 }  
 return new SelectorTuple(unwrappedSelector);  
 }  
  
 final Class<?> selectorImplClass = (Class<?>) maybeSelectorImplClass;  
  
 Object maybeException = AccessController.doPrivileged(new PrivilegedAction<Object>() {  
 @Override  
 public Object run() {  
 try {  
 Field selectedKeysField = selectorImplClass.getDeclaredField("selectedKeys");  
 Field publicSelectedKeysField = selectorImplClass.getDeclaredField("publicSelectedKeys");  
  
 Throwable cause = ReflectionUtil.trySetAccessible(selectedKeysField, true);  
 if (cause != null) {  
 return cause;  
 }  
 cause = ReflectionUtil.trySetAccessible(publicSelectedKeysField, true);  
 if (cause != null) {  
 return cause;  
 }  
  
 selectedKeysField.set(unwrappedSelector, selectedKeySet);  
 publicSelectedKeysField.set(unwrappedSelector, selectedKeySet);  
 return null;  
 } catch (NoSuchFieldException e) {  
 return e;  
 } catch (IllegalAccessException e) {  
 return e;  
 }  
 }  
 });  
  
 if (maybeException instanceof Exception) {  
 selectedKeys = null;  
 Exception e = (Exception) maybeException;  
 logger.trace("failed to instrument a special java.util.Set into: {}", unwrappedSelector, e);  
 return new SelectorTuple(unwrappedSelector);  
 }  
 selectedKeys = selectedKeySet;  
 logger.trace("instrumented a special java.util.Set into: {}", unwrappedSelector);  
 return new SelectorTuple(unwrappedSelector,  
 new SelectedSelectionKeySetSelector(unwrappedSelector, selectedKeySet));  
}

- 1.3) DefaultEventExecutorChooserFactory#newChooser

public EventExecutorChooser newChooser(EventExecutor[] executors) {  
 if (isPowerOfTwo(executors.length)) {  
 return new PowerOfTwoEventExecutorChooser(executors);  
 } else {  
 return new GenericEventExecutorChooser(executors);  
 }  
}

private static final class PowerOfTwoEventExecutorChooser implements EventExecutorChooser {  
 private final AtomicInteger idx = new AtomicInteger();  
 private final EventExecutor[] executors;  
  
 PowerOfTwoEventExecutorChooser(EventExecutor[] executors) {  
 this.executors = executors;  
 }  
  
 @Override  
 public EventExecutor next() {  
 return executors[idx.getAndIncrement() & executors.length - 1];  
 }  
}  
  
private static final class GenericEventExecutorChooser implements EventExecutorChooser {  
 private final AtomicInteger idx = new AtomicInteger();  
 private final EventExecutor[] executors;  
  
 GenericEventExecutorChooser(EventExecutor[] executors) {  
 this.executors = executors;  
 }  
  
 @Override  
 public EventExecutor next() {  
 return executors[Math.abs(idx.getAndIncrement() % executors.length)];  
 }  
}

- 2) AbstractBootstrap#group

* AbstractBootstrap

volatile EventLoopGroup group;  
@SuppressWarnings("deprecation")  
private volatile ChannelFactory<? extends C> channelFactory;  
private volatile SocketAddress localAddress;  
private final Map<ChannelOption<?>, Object> options = new LinkedHashMap<ChannelOption<?>, Object>();  
private final Map<AttributeKey<?>, Object> attrs = new LinkedHashMap<AttributeKey<?>, Object>();  
private volatile ChannelHandler handler;

public B group(EventLoopGroup group) {  
 if (group == null) {  
 throw new NullPointerException("group");  
 }  
 if (this.group != null) {  
 throw new IllegalStateException("group set already");  
 }  
 this.group = group;  
 return self();  
}

- 3) AbstractBootstrap#channel(NioSocketChannel)

public B channel(Class<? extends C> channelClass) {  
 if (channelClass == null) {  
 throw new NullPointerException("channelClass");  
 }  
 return channelFactory(new ReflectiveChannelFactory<C>(channelClass));  
}

* 3.1

public ReflectiveChannelFactory(Class<? extends T> clazz) {  
 if (clazz == null) {  
 throw new NullPointerException("clazz");  
 }  
 this.clazz = clazz;  
}

* 3.2

public B channelFactory(io.netty.channel.ChannelFactory<? extends C> channelFactory) {  
 return channelFactory((ChannelFactory<C>) channelFactory);  
}

* 3.2.1

public B channelFactory(ChannelFactory<? extends C> channelFactory) {  
 if (channelFactory == null) {  
 throw new NullPointerException("channelFactory");  
 }  
 if (this.channelFactory != null) {  
 throw new IllegalStateException("channelFactory set already");  
 }  
  
 this.channelFactory = channelFactory;  
 return self();  
}

- 4) AbstractBootstrap#handler

public B handler(ChannelHandler handler) {  
 if (handler == null) {  
 throw new NullPointerException("handler");  
 }  
 this.handler = handler;  
 return self();  
}

- 5) AbstractBootstrap#option

private final Map<ChannelOption<?>, Object> options = new LinkedHashMap<ChannelOption<?>, Object>();

public <T> B option(ChannelOption<T> option, T value) {  
 if (option == null) {  
 throw new NullPointerException("option");  
 }  
 if (value == null) {  
 synchronized (options) {  
 options.remove(option);  
 }  
 } else {  
 synchronized (options) {  
 options.put(option, value);  
 }  
 }  
 return self();  
}

- 6) Bootstrap#connect

public ChannelFuture connect(String inetHost, int inetPort) {  
 return connect(InetSocketAddress.createUnresolved(inetHost, inetPort));  
}

public static InetSocketAddress createUnresolved(String host, int port) {  
 return new InetSocketAddress(checkPort(port), checkHost(host));  
}

public ChannelFuture connect(SocketAddress remoteAddress) {  
 if (remoteAddress == null) {  
 throw new NullPointerException("remoteAddress");  
 }  
  
 validate();  
 return doResolveAndConnect(remoteAddress, config.localAddress());  
}

- 6.1) Bootstrap#doResolveAndConnect

private ChannelFuture doResolveAndConnect(final SocketAddress remoteAddress, final SocketAddress localAddress) {  
 final ChannelFuture regFuture = initAndRegister();  
 final Channel channel = regFuture.channel();  
  
 if (regFuture.isDone()) {  
 if (!regFuture.isSuccess()) {  
 return regFuture;  
 }  
 return doResolveAndConnect0(channel, remoteAddress, localAddress, channel.newPromise());  
 } else {  
 // Registration future is almost always fulfilled already, but just in case it's not.  
 final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);  
 regFuture.addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 // Directly obtain the cause and do a null check so we only need one volatile read in case of a  
 // failure.  
 Throwable cause = future.cause();  
 if (cause != null) {  
 // Registration on the EventLoop failed so fail the ChannelPromise directly to not cause an  
 // IllegalStateException once we try to access the EventLoop of the Channel.  
 promise.setFailure(cause);  
 } else {  
 // Registration was successful, so set the correct executor to use.  
 // See https://github.com/netty/netty/issues/2586  
 promise.registered();  
 doResolveAndConnect0(channel, remoteAddress, localAddress, promise);  
 }  
 }  
 });  
 return promise;  
 }  
}

- 6.1.1) AbstractBoostrap#initAndRegister（返回值是DefaultChannelPromise)  
- final ChannelFuture initAndRegister() {  
Channel channel = null;  
try {  
 channel = channelFactory.newChannel();  
 init(channel);  
} catch (Throwable t) {  
 if (channel != null) {  
 // channel can be null if newChannel crashed (eg SocketException("too many open files"))  
 channel.unsafe().closeForcibly();  
 // as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor  
 return new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE).setFailure(t);  
 }  
 // as the Channel is not registered yet we need to force the usage of the GlobalEventExecutor  
 return new DefaultChannelPromise(new FailedChannel(), GlobalEventExecutor.INSTANCE).setFailure(t);  
}  
  
ChannelFuture regFuture = config().group().register(channel);  
if (regFuture.cause() != null) {  
 if (channel.isRegistered()) {  
 channel.close();  
 } else {  
 channel.unsafe().closeForcibly();  
 }  
}  
  
// If we are here and the promise is not failed, it's one of the following cases:  
// 1) If we attempted registration from the event loop, the registration has been completed at this point.  
// i.e. It's safe to attempt bind() or connect() now because the channel has been registered.  
// 2) If we attempted registration from the other thread, the registration request has been successfully  
// added to the event loop's task queue for later execution.  
// i.e. It's safe to attempt bind() or connect() now:  
// because bind() or connect() will be executed \*after\* the scheduled registration task is executed  
// because register(), bind(), and connect() are all bound to the same thread.  
  
return regFuture;

}

- 6.1.1.1) （创建Channel和对应的pipeline)ReflectiveChannelFactory#newChannel

public T newChannel() {  
 try {  
 return clazz.getConstructor().newInstance();  
 } catch (Throwable t) {  
 throw new ChannelException("Unable to create Channel from class " + clazz, t);  
 }  
}

- 6.1.1.1.1) NioSocketChannel#construactor

private static final SelectorProvider DEFAULT\_SELECTOR\_PROVIDER = SelectorProvider.provider();

public NioSocketChannel() {  
 this(DEFAULT\_SELECTOR\_PROVIDER);  
}

public NioSocketChannel(SelectorProvider provider) {  
 this(newSocket(provider));  
}

private static SocketChannel newSocket(SelectorProvider provider) {  
 try {  
 /\*\*  
 \* Use the {@link SelectorProvider} to open {@link SocketChannel} and so remove condition in  
 \* {@link SelectorProvider#provider()} which is called by each SocketChannel.open() otherwise.  
 \*  
 \* See <a href="https://github.com/netty/netty/issues/2308">#2308</a>.  
 \*/  
 return provider.openSocketChannel();  
 } catch (IOException e) {  
 throw new ChannelException("Failed to open a socket.", e);  
 }  
}

public NioSocketChannel(SocketChannel socket) {  
 this(null, socket);  
}

public NioSocketChannel(Channel parent, SocketChannel socket) {  
 super(parent, socket);  
 config = new NioSocketChannelConfig(this, socket.socket());  
}

* protected AbstractNioByteChannel(Channel parent, SelectableChannel ch) {  
  super(parent, ch, SelectionKey.OP\_READ);

/\*\*  
 \* Create a new instance  
 \*  
 \* @param parent the parent {@link Channel} by which this instance was created. May be {@code null}  
 \* @param ch the underlying {@link SelectableChannel} on which it operates  
 \* @param readInterestOp the ops to set to receive data from the {@link SelectableChannel}  
 \*/  
protected AbstractNioChannel(Channel parent, SelectableChannel ch, int readInterestOp) {  
 super(parent);  
 this.ch = ch;  
 this.readInterestOp = readInterestOp;  
 try {  
 ch.configureBlocking(false);  
 } catch (IOException e) {  
 try {  
 ch.close();  
 } catch (IOException e2) {  
 if (logger.isWarnEnabled()) {  
 logger.warn(  
 "Failed to close a partially initialized socket.", e2);  
 }  
 }  
  
 throw new ChannelException("Failed to enter non-blocking mode.", e);  
 }  
}

* protected AbstractChannel(Channel parent) {  
  this.parent = parent;  
  id = newId();  
  unsafe = newUnsafe();  
  pipeline = newChannelPipeline();  
  }
* protected ChannelId newId() {  
  return DefaultChannelId.newInstance();  
  }
* protected DefaultChannelPipeline newChannelPipeline() {  
  return new DefaultChannelPipeline(this);  
  }
* 双向链表结构：
* final AbstractChannelHandlerContext head;  
  final AbstractChannelHandlerContext tail;
* protected DefaultChannelPipeline(Channel channel) {  
  this.channel = ObjectUtil.checkNotNull(channel, "channel");  
  succeededFuture = new SucceededChannelFuture(channel, null);  
  voidPromise = new VoidChannelPromise(channel, true);
* tail = new TailContext(this);  
  head = new HeadContext(this);
* head.next = tail;  
  tail.prev = head;  
  }
  + 6.1.1.2) （向pipeline中添加handler，并创建其对应的ChannelHandlerContext)Boostrap#init
* void init(Channel channel) throws Exception {  
  ChannelPipeline p = channel.pipeline();  
  p.addLast(config.handler());
* final Map<ChannelOption<?>, Object> options = options0();  
  synchronized (options) {  
   setChannelOptions(channel, options, logger);  
  }
* final Map<AttributeKey<?>, Object> attrs = attrs0();  
  synchronized (attrs) {  
   for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {  
   channel.attr((AttributeKey) e.getKey()).set(e.getValue());  
   }  
  }  
  }
  + 6.1.1.2.1) DefaultChannelPipeline#addLast
* AbstractBootstrapConfig#handler

public final ChannelHandler handler() {  
 return bootstrap.handler();  
}

public final ChannelPipeline addLast(ChannelHandler... handlers) {  
 return addLast(null, handlers);  
}

public final ChannelPipeline addLast(EventExecutorGroup executor, ChannelHandler... handlers) {  
 if (handlers == null) {  
 throw new NullPointerException("handlers");  
 }  
  
 for (ChannelHandler h: handlers) {  
 if (h == null) {  
 break;  
 }  
 addLast(executor, null, h);  
 }  
  
 return this;  
}

public final ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler) {  
 final AbstractChannelHandlerContext newCtx;  
 synchronized (this) {  
 checkMultiplicity(handler);  
  
 newCtx = newContext(group, filterName(name, handler), handler);  
  
 addLast0(newCtx);  
  
 // If the registered is false it means that the channel was not registered on an eventloop yet.  
 // In this case we add the context to the pipeline and add a task that will call  
 // ChannelHandler.handlerAdded(...) once the channel is registered.  
 if (!registered) {  
 newCtx.setAddPending();  
 callHandlerCallbackLater(newCtx, true);  
 return this;  
 }  
  
 EventExecutor executor = newCtx.executor();  
 if (!executor.inEventLoop()) {  
 newCtx.setAddPending();  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 callHandlerAdded0(newCtx);  
 }  
 });  
 return this;  
 }  
 }

* return this;  
  }
  + 6.1.1.2.1.1) （创建context)DefaultChannelPipeline#newContext

private String filterName(String name, ChannelHandler handler) {  
 if (name == null) {  
 return generateName(handler);  
 }  
 checkDuplicateName(name);  
 return name;  
}

private AbstractChannelHandlerContext newContext(EventExecutorGroup group, String name, ChannelHandler handler) {  
 return new DefaultChannelHandlerContext(this, childExecutor(group), name, handler);  
}

private EventExecutor childExecutor(EventExecutorGroup group) {  
 if (group == null) {  
 return null;  
 }  
 Boolean pinEventExecutor = channel.config().getOption(ChannelOption.SINGLE\_EVENTEXECUTOR\_PER\_GROUP);  
 if (pinEventExecutor != null && !pinEventExecutor) {  
 return group.next();  
 }  
 Map<EventExecutorGroup, EventExecutor> childExecutors = this.childExecutors;  
 if (childExecutors == null) {  
 // Use size of 4 as most people only use one extra EventExecutor.  
 childExecutors = this.childExecutors = new IdentityHashMap<EventExecutorGroup, EventExecutor>(4);  
 }  
 // Pin one of the child executors once and remember it so that the same child executor  
 // is used to fire events for the same channel.  
 EventExecutor childExecutor = childExecutors.get(group);  
 if (childExecutor == null) {  
 childExecutor = group.next();  
 childExecutors.put(group, childExecutor);  
 }  
 return childExecutor;  
}

- 6.1.1.2.1.1.1) DefaultChannelHandlerContext#constructor

* DefaultChannelHandlerContext(  
   DefaultChannelPipeline pipeline, EventExecutor executor, String name, ChannelHandler handler) {  
  super(pipeline, executor, name, isInbound(handler), isOutbound(handler));  
  if (handler == null) {  
   throw new NullPointerException("handler");  
  }  
  this.handler = handler;  
  }

private static boolean isInbound(ChannelHandler handler) {  
 return handler instanceof ChannelInboundHandler;  
}

private static boolean isOutbound(ChannelHandler handler) {  
 return handler instanceof ChannelOutboundHandler;  
}

* AbstractChannelHandlerContext(DefaultChannelPipeline pipeline, EventExecutor executor, String name,  
   boolean inbound, boolean outbound) {  
  this.name = ObjectUtil.checkNotNull(name, "name");  
  this.pipeline = pipeline;  
  this.executor = executor;  
  this.inbound = inbound;  
  this.outbound = outbound;  
  // Its ordered if its driven by the EventLoop or the given Executor is an instanceof OrderedEventExecutor.  
  ordered = executor == null || executor instanceof OrderedEventExecutor;  
  }

- 6.1.1.2.1.2) DefaultChannelPipeline#addLast0

private void addLast0(AbstractChannelHandlerContext newCtx) {  
 AbstractChannelHandlerContext prev = tail.prev;  
 newCtx.prev = prev;  
 newCtx.next = tail;  
 prev.next = newCtx;  
 tail.prev = newCtx;  
}

- 6.1.1.2.1.3) DefaultChannelPipeline#callHandlerCallbackLater

* added=true

private PendingHandlerCallback pendingHandlerCallbackHead;

* 挂起handler的callback是一个链表结构

private void callHandlerCallbackLater(AbstractChannelHandlerContext ctx, boolean added) {  
 assert !registered;  
  
 PendingHandlerCallback task = added ? new PendingHandlerAddedTask(ctx) : new PendingHandlerRemovedTask(ctx);  
 PendingHandlerCallback pending = pendingHandlerCallbackHead;  
 if (pending == null) {  
 pendingHandlerCallbackHead = task;  
 } else {  
 // Find the tail of the linked-list.  
 while (pending.next != null) {  
 pending = pending.next;  
 }  
 pending.next = task;  
 }  
}

* PendingHandlerAddedTask(AbstractChannelHandlerContext ctx) {  
  super(ctx);  
  }
* PendingHandlerCallback(AbstractChannelHandlerContext ctx) {  
  this.ctx = ctx;  
  }
  + 6.1.1.3) MultithreadEventLoopGroup#register  
    1.- 首先在 AbstractBootstrap.initAndRegister中, 通过 group().register(channel), 调用 MultithreadEventLoopGroup.register 方法  
    2.- 在MultithreadEventLoopGroup.register 中, 通过 next() 获取一个可用的 SingleThreadEventLoop, 然后调用它的 register  
    3.- 在 SingleThreadEventLoop.register 中, 通过 channel.unsafe().register(this, promise) 来获取 channel 的 unsafe() 底层操作对象, 然后调用它的 register.  
    4.- 在 AbstractUnsafe.register 方法中, 调用 register0 方法注册 Channel  
    5.- 在 AbstractUnsafe.register0 中, 调用 AbstractNioChannel.doRegister 方法  
    6.- AbstractNioChannel.doRegister 方法通过 javaChannel().register(eventLoop().selector, 0, this) 将 Channel 对应的 Java NIO SockerChannel 注册到一个 eventLoop 的 Selector 中, 并且将当前 Channel 作为 attachment.

public ChannelFuture register(Channel channel) {  
 return next().register(channel);  
}

- 6.1.1.3.1) MultithreadEventLoopGroup#next

public EventLoop next() {  
 return (EventLoop) super.next();  
}

public EventExecutor next() {  
 return chooser.next();  
}

* DefaultEventExecutorChooserFactory#next（RR)

public EventExecutor next() {  
 return executors[idx.getAndIncrement() & executors.length - 1];  
}

- 6.1.1.3.2) SingleThreadEventLoop#register（向EventLoop添加一个注册任务)

* 返回值是一个DefaultChannelPromise

public ChannelFuture register(Channel channel) {  
 return register(new DefaultChannelPromise(channel, this));  
}

public DefaultChannelPromise(Channel channel, EventExecutor executor) {  
 super(executor);  
 this.channel = checkNotNull(channel, "channel");  
}

public DefaultPromise(EventExecutor executor) {  
 this.executor = checkNotNull(executor, "executor");  
}

public ChannelFuture register(final ChannelPromise promise) {  
 ObjectUtil.checkNotNull(promise, "promise");  
 promise.channel().unsafe().register(this, promise);  
 return promise;  
}

- 6.1.1.3.2.1) AbstractChannel#register

public final void register(EventLoop eventLoop, final ChannelPromise promise) {  
 if (eventLoop == null) {  
 throw new NullPointerException("eventLoop");  
 }  
 if (isRegistered()) {  
 promise.setFailure(new IllegalStateException("registered to an event loop already"));  
 return;  
 }  
 if (!isCompatible(eventLoop)) {  
 promise.setFailure(  
 new IllegalStateException("incompatible event loop type: " + eventLoop.getClass().getName()));  
 return;  
 }  
  
 AbstractChannel.this.eventLoop = eventLoop;  
  
 if (eventLoop.inEventLoop()) {  
 register0(promise);  
 } else {  
 try {  
 eventLoop.execute(new Runnable() {  
 @Override  
 public void run() {  
 register0(promise);  
 }  
 });  
 } catch (Throwable t) {  
 logger.warn(  
 "Force-closing a channel whose registration task was not accepted by an event loop: {}",  
 AbstractChannel.this, t);  
 closeForcibly();  
 closeFuture.setClosed();  
 safeSetFailure(promise, t);  
 }  
 }  
}

- 6.1.1.3.2.1.1) AbstractChannel#register0（将Channel注册到Selector中，初始化EventLoop线程，注册自定义Handler)

private void register0(ChannelPromise promise) {  
 try {  
 // check if the channel is still open as it could be closed in the mean time when the register  
 // call was outside of the eventLoop  
 if (!promise.setUncancellable() || !ensureOpen(promise)) {  
 return;  
 }  
 boolean firstRegistration = neverRegistered;  
 doRegister();  
 neverRegistered = false;  
 registered = true;  
  
 // Ensure we call handlerAdded(...) before we actually notify the promise. This is needed as the  
 // user may already fire events through the pipeline in the ChannelFutureListener.  
 pipeline.invokeHandlerAddedIfNeeded();  
  
 safeSetSuccess(promise);

* // 添加自定义Handler  
   pipeline.fireChannelRegistered();  
   // Only fire a channelActive if the channel has never been registered. This prevents firing  
   // multiple channel actives if the channel is deregistered and re-registered.  
   if (isActive()) {  
   if (firstRegistration) {  
   pipeline.fireChannelActive();  
   } else if (config().isAutoRead()) {  
   // This channel was registered before and autoRead() is set. This means we need to begin read  
   // again so that we process inbound data.  
   //  
   // See https://github.com/netty/netty/issues/4805  
   beginRead();  
   }  
   }  
  } catch (Throwable t) {  
   // Close the channel directly to avoid FD leak.  
   closeForcibly();  
   closeFuture.setClosed();  
   safeSetFailure(promise, t);  
  }  
  }
  + 6.1.1.3.2.1.1.1) AbstractNioChannel#doRegister（将Channel注册到Selector中)
* protected void doRegister() throws Exception {  
  boolean selected = false;  
  for (;;) {  
   try {  
   selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);  
   return;  
   } catch (CancelledKeyException e) {  
   if (!selected) {  
   // Force the Selector to select now as the "canceled" SelectionKey may still be  
   // cached and not removed because no Select.select(..) operation was called yet.  
   eventLoop().selectNow();  
   selected = true;  
   } else {  
   // We forced a select operation on the selector before but the SelectionKey is still cached  
   // for whatever reason. JDK bug ?  
   throw e;  
   }  
   }  
  }  
  }
  + 6.1.1.3.2.1.1.1.1) AbstractSeletableChannel#register

public final SelectionKey register(Selector sel, int ops,  
 Object att)  
 throws ClosedChannelException  
{  
 synchronized (regLock) {  
 if (!isOpen())  
 throw new ClosedChannelException();  
 if ((ops & ~validOps()) != 0)  
 throw new IllegalArgumentException();  
 if (blocking)  
 throw new IllegalBlockingModeException();  
 SelectionKey k = findKey(sel);  
 if (k != null) {  
 k.interestOps(ops);  
 k.attach(att);  
 }  
 if (k == null) {  
 // New registration  
 synchronized (keyLock) {  
 if (!isOpen())  
 throw new ClosedChannelException();  
 k = ((AbstractSelector)sel).register(this, ops, att);  
 addKey(k);  
 }  
 }  
 return k;  
 }  
}

- 6.1.1.3.2.1.2) DefaultChannelPipeline#invokeHandlerAddedIfNeeded

* final void invokeHandlerAddedIfNeeded() {  
  assert channel.eventLoop().inEventLoop();  
  if (firstRegistration) {  
   firstRegistration = false;  
   // We are now registered to the EventLoop. It's time to call the callbacks for the ChannelHandlers,  
   // that were added before the registration was done.  
   callHandlerAddedForAllHandlers();  
  }  
  }

private void callHandlerAddedForAllHandlers() {  
 final PendingHandlerCallback pendingHandlerCallbackHead;  
 synchronized (this) {  
 assert !registered;  
  
 // This Channel itself was registered.  
 registered = true;  
  
 pendingHandlerCallbackHead = this.pendingHandlerCallbackHead;  
 // Null out so it can be GC'ed.  
 this.pendingHandlerCallbackHead = null;  
 }  
  
 // This must happen outside of the synchronized(...) block as otherwise handlerAdded(...) may be called while  
 // holding the lock and so produce a deadlock if handlerAdded(...) will try to add another handler from outside  
 // the EventLoop.  
 PendingHandlerCallback task = pendingHandlerCallbackHead;  
 while (task != null) {  
 task.execute();  
 task = task.next;  
 }  
}

- 6.1.1.3.2.1.3) DefaultChannelPipeline#fireChannelRegistered（添加自定义Handler)

public final ChannelPipeline fireChannelRegistered() {  
 AbstractChannelHandlerContext.invokeChannelRegistered(head);  
 return this;  
}

static void invokeChannelRegistered(final AbstractChannelHandlerContext next) {  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 next.invokeChannelRegistered();  
 } else {  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 next.invokeChannelRegistered();  
 }  
 });  
 }  
}

private void invokeChannelRegistered() {  
 if (invokeHandler()) {  
 try {  
 ((ChannelInboundHandler) handler()).channelRegistered(this);  
 } catch (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } else {  
 fireChannelRegistered();  
 }  
}

* 下面要着重关注ChannelInitializer
* ChannelInitializer#channelRegistered

public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {  
 // Normally this method will never be called as handlerAdded(...) should call initChannel(...) and remove  
 // the handler.  
 if (initChannel(ctx)) {  
 // we called initChannel(...) so we need to call now pipeline.fireChannelRegistered() to ensure we not  
 // miss an event.  
 ctx.pipeline().fireChannelRegistered();  
 } else {  
 // Called initChannel(...) before which is the expected behavior, so just forward the event.  
 ctx.fireChannelRegistered();  
 }  
}

private boolean initChannel(ChannelHandlerContext ctx) throws Exception {  
 if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) { // Guard against re-entrance.  
 try {  
 initChannel((C) ctx.channel());  
 } catch (Throwable cause) {  
 // Explicitly call exceptionCaught(...) as we removed the handler before calling initChannel(...).  
 // We do so to prevent multiple calls to initChannel(...).  
 exceptionCaught(ctx, cause);  
 } finally {

* // 将ChannelInitializer从pipeline中移除  
   remove(ctx);  
  }  
  return true;  
  }  
  return false;  
  }

private void remove(ChannelHandlerContext ctx) {  
 try {  
 ChannelPipeline pipeline = ctx.pipeline();  
 if (pipeline.context(this) != null) {  
 pipeline.remove(this);  
 }  
 } finally {  
 initMap.remove(ctx);  
 }  
}

- 6.1.2) DefaultPromise#isDone

public boolean isDone() {  
 return isDone0(result);  
}

private static boolean isDone0(Object result) {  
 return result != null && result != UNCANCELLABLE;  
}

- 6.1.3) Boostrap#doResolveAndConnect0（真正连接)

public final ChannelPromise newPromise() {  
 return new DefaultChannelPromise(channel);  
}

public DefaultChannelPromise(Channel channel) {  
 this.channel = checkNotNull(channel, "channel");  
}

private ChannelFuture doResolveAndConnect0(final Channel channel, SocketAddress remoteAddress,  
 final SocketAddress localAddress, final ChannelPromise promise) {  
 try {  
 final EventLoop eventLoop = channel.eventLoop();  
 final AddressResolver<SocketAddress> resolver = this.resolver.getResolver(eventLoop);  
  
 if (!resolver.isSupported(remoteAddress) || resolver.isResolved(remoteAddress)) {  
 // Resolver has no idea about what to do with the specified remote address or it's resolved already.  
 doConnect(remoteAddress, localAddress, promise);  
 return promise;  
 }  
  
 final Future<SocketAddress> resolveFuture = resolver.resolve(remoteAddress);  
  
 if (resolveFuture.isDone()) {  
 final Throwable resolveFailureCause = resolveFuture.cause();  
  
 if (resolveFailureCause != null) {  
 // Failed to resolve immediately  
 channel.close();  
 promise.setFailure(resolveFailureCause);  
 } else {  
 // Succeeded to resolve immediately; cached? (or did a blocking lookup)  
 doConnect(resolveFuture.getNow(), localAddress, promise);  
 }  
 return promise;  
 }  
  
 // Wait until the name resolution is finished.  
 resolveFuture.addListener(new FutureListener<SocketAddress>() {  
 @Override  
 public void operationComplete(Future<SocketAddress> future) throws Exception {  
 if (future.cause() != null) {  
 channel.close();  
 promise.setFailure(future.cause());  
 } else {  
 doConnect(future.getNow(), localAddress, promise);  
 }  
 }  
 });  
 } catch (Throwable cause) {  
 promise.tryFailure(cause);  
 }  
 return promise;  
}

- 6.1.3.1) AbstractAddressResolver#resolve

public final Future<T> resolve(SocketAddress address) {  
 if (!isSupported(checkNotNull(address, "address"))) {  
 // Address type not supported by the resolver  
 return executor().newFailedFuture(new UnsupportedAddressTypeException());  
 }  
  
 if (isResolved(address)) {  
 // Resolved already; no need to perform a lookup  
 @SuppressWarnings("unchecked")  
 final T cast = (T) address;  
 return executor.newSucceededFuture(cast);  
 }  
  
 try {  
 @SuppressWarnings("unchecked")  
 final T cast = (T) address;  
 final Promise<T> promise = executor().newPromise();  
 doResolve(cast, promise);  
 return promise;  
 } catch (Exception e) {  
 return executor().newFailedFuture(e);  
 }  
}

- 6.1.3.1.1) InetSocketAddressResolver#doResolve

protected void doResolve(final InetSocketAddress unresolvedAddress, final Promise<InetSocketAddress> promise)  
 throws Exception {  
 // Note that InetSocketAddress.getHostName() will never incur a reverse lookup here,  
 // because an unresolved address always has a host name.  
 nameResolver.resolve(unresolvedAddress.getHostName())  
 .addListener(new FutureListener<InetAddress>() {  
 @Override  
 public void operationComplete(Future<InetAddress> future) throws Exception {  
 if (future.isSuccess()) {  
 promise.setSuccess(new InetSocketAddress(future.getNow(), unresolvedAddress.getPort()));  
 } else {  
 promise.setFailure(future.cause());  
 }  
 }  
 });  
}

* SimpleNameResolver#resolve

public final Future<T> resolve(String inetHost) {  
 final Promise<T> promise = executor().newPromise();  
 return resolve(inetHost, promise);  
}

public Future<T> resolve(String inetHost, Promise<T> promise) {  
 checkNotNull(promise, "promise");  
  
 try {  
 doResolve(inetHost, promise);  
 return promise;  
 } catch (Exception e) {  
 return promise.setFailure(e);  
 }  
}

* protected void doResolve(String inetHost, Promise promise) throws Exception {  
  try {  
   promise.setSuccess(SocketUtils.addressByName(inetHost));  
  } catch (UnknownHostException e) {  
   promise.setFailure(e);  
  }  
  }

public static InetAddress addressByName(final String hostname) throws UnknownHostException {  
 try {  
 return AccessController.doPrivileged(new PrivilegedExceptionAction<InetAddress>() {  
 @Override  
 public InetAddress run() throws UnknownHostException {  
 return InetAddress.getByName(hostname);  
 }  
 });  
 } catch (PrivilegedActionException e) {  
 throw (UnknownHostException) e.getCause();  
 }  
}

- 6.1.3.2) Bootstrap#doConnect（向eventLoop中添加一个connect的任务)

private static void doConnect(  
 final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise connectPromise) {  
  
 // This method is invoked before channelRegistered() is triggered. Give user handlers a chance to set up  
 // the pipeline in its channelRegistered() implementation.  
 final Channel channel = connectPromise.channel();  
 channel.eventLoop().execute(new Runnable() {  
 @Override  
 public void run() {  
 if (localAddress == null) {  
 channel.connect(remoteAddress, connectPromise);  
 } else {  
 channel.connect(remoteAddress, localAddress, connectPromise);  
 }  
 connectPromise.addListener(ChannelFutureListener.CLOSE\_ON\_FAILURE);  
 }  
 });  
}

- 6.1.3.2.1) AbstractChannel#connect

public ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {  
 return pipeline.connect(remoteAddress, promise);  
}

* DefaultChannelPipeline#connect

public final ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {  
 return tail.connect(remoteAddress, promise);  
}

- 6.1.3.2.1.1) AbstractChannelHandlerContext#connect

public ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {  
 return connect(remoteAddress, null, promise);  
}

public ChannelFuture connect(  
 final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise) {  
  
 if (remoteAddress == null) {  
 throw new NullPointerException("remoteAddress");  
 }  
 if (isNotValidPromise(promise, false)) {  
 // cancelled  
 return promise;  
 }  
 // 从pipeline的后面往前面遍历，找到第一个outBoundHandler  
 final AbstractChannelHandlerContext next = findContextOutbound();  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 next.invokeConnect(remoteAddress, localAddress, promise);  
 } else {  
 safeExecute(executor, new Runnable() {  
 @Override  
 public void run() {  
 next.invokeConnect(remoteAddress, localAddress, promise);  
 }  
 }, promise, null);  
 }  
 return promise;  
}

- 6.1.3.2.1.1.1) AbstractChannelHandlerContext#invokeConnect

* HeadContext 重写了 connect 方法, 因此实际上调用的是 HeadContext.connect.

private void invokeConnect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise) {  
 if (invokeHandler()) {  
 try {  
 ((ChannelOutboundHandler) handler()).connect(this, remoteAddress, localAddress, promise);  
 } catch (Throwable t) {  
 notifyOutboundHandlerException(t, promise);  
 }  
 } else {  
 connect(remoteAddress, localAddress, promise);  
 }  
}

- 6.1.3.2.1.1.1.1) DefaultChannelPipeline#connect

* Channel 的 unsafe 字段是 AbstractNioByteChannel.NioByteUnsafe 内部类.

public void connect(  
 ChannelHandlerContext ctx,  
 SocketAddress remoteAddress, SocketAddress localAddress,  
 ChannelPromise promise) throws Exception {  
 unsafe.connect(remoteAddress, localAddress, promise);  
}

- 6.1.3.2.1.1.1.1.1) NioSocketChannel#connect

public final void connect(  
 final SocketAddress remoteAddress, final SocketAddress localAddress, final ChannelPromise promise) {  
 if (!promise.setUncancellable() || !ensureOpen(promise)) {  
 return;  
 }  
  
 try {  
 if (connectPromise != null) {  
 // Already a connect in process.  
 throw new ConnectionPendingException();  
 }  
  
 boolean wasActive = isActive();  
 if (doConnect(remoteAddress, localAddress)) {  
 fulfillConnectPromise(promise, wasActive);  
 } else {  
 connectPromise = promise;  
 requestedRemoteAddress = remoteAddress;  
  
 // Schedule connect timeout.  
 int connectTimeoutMillis = config().getConnectTimeoutMillis();  
 if (connectTimeoutMillis > 0) {  
 connectTimeoutFuture = eventLoop().schedule(new Runnable() {  
 @Override  
 public void run() {  
 ChannelPromise connectPromise = AbstractNioChannel.this.connectPromise;  
 ConnectTimeoutException cause =  
 new ConnectTimeoutException("connection timed out: " + remoteAddress);  
 if (connectPromise != null && connectPromise.tryFailure(cause)) {  
 close(voidPromise());  
 }  
 }  
 }, connectTimeoutMillis, TimeUnit.MILLISECONDS);  
 }  
  
 promise.addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isCancelled()) {  
 if (connectTimeoutFuture != null) {  
 connectTimeoutFuture.cancel(false);  
 }  
 connectPromise = null;  
 close(voidPromise());  
 }  
 }  
 });  
 }  
 } catch (Throwable t) {  
 promise.tryFailure(annotateConnectException(t, remoteAddress));  
 closeIfClosed();  
 }  
}

* protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress) throws Exception {  
  if (localAddress != null) {  
   doBind0(localAddress);  
  }
* boolean success = false;  
  try {  
   boolean connected = SocketUtils.connect(javaChannel(), remoteAddress);  
   if (!connected) {  
   selectionKey().interestOps(SelectionKey.OP\_CONNECT);  
   }  
   success = true;  
   return connected;  
  } finally {  
   if (!success) {  
   doClose();  
   }  
  }  
  }
  + 6.1.3.2.1.1.1.1.1.1) SocketUtils#connect

public static boolean connect(final SocketChannel socketChannel, final SocketAddress remoteAddress)  
 throws IOException {  
 try {  
 return AccessController.doPrivileged(new PrivilegedExceptionAction<Boolean>() {  
 @Override  
 public Boolean run() throws IOException {  
 return socketChannel.connect(remoteAddress);  
 }  
 });  
 } catch (PrivilegedActionException e) {  
 throw (IOException) e.getCause();  
 }  
}

# 服务器启动过程

- 1) NioEventLoopGroup#constructor

* 同客户端的这部分
  + 2) ServerBootstrap#group

private final Map<ChannelOption<?>, Object> childOptions = new LinkedHashMap<ChannelOption<?>, Object>();  
private final Map<AttributeKey<?>, Object> childAttrs = new LinkedHashMap<AttributeKey<?>, Object>();  
private final ServerBootstrapConfig config = new ServerBootstrapConfig(this);  
private volatile EventLoopGroup childGroup;  
private volatile ChannelHandler childHandler;

public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {  
 super.group(parentGroup);  
 if (childGroup == null) {  
 throw new NullPointerException("childGroup");  
 }  
 if (this.childGroup != null) {  
 throw new IllegalStateException("childGroup set already");  
 }  
 this.childGroup = childGroup;  
 return this;  
}

- 2.1) AbstractGroup#group

public B group(EventLoopGroup group) {  
 if (group == null) {  
 throw new NullPointerException("group");  
 }  
 if (this.group != null) {  
 throw new IllegalStateException("group set already");  
 }  
 this.group = group;  
 return self();  
}

- 3) AbstractBoostrap#channel

* 同客户端的这部分
  + 4) ServerBoostrap#childHandler

public ServerBootstrap childHandler(ChannelHandler childHandler) {  
 if (childHandler == null) {  
 throw new NullPointerException("childHandler");  
 }  
 this.childHandler = childHandler;  
 return this;  
}

- 5) AbstractBootstrap#option

* 同客户端的这部分
  + 6) AbstractBoostrap#bind

public ChannelFuture bind(String inetHost, int inetPort) {  
 return bind(SocketUtils.socketAddress(inetHost, inetPort));  
}

public ChannelFuture bind(SocketAddress localAddress) {  
 validate();  
 if (localAddress == null) {  
 throw new NullPointerException("localAddress");  
 }  
 return doBind(localAddress);  
}

- 6.1) AbstractBoostrap#doBind

private ChannelFuture doBind(final SocketAddress localAddress) {

// 同客户端的6.1.1，但部分代码有重写  
 final ChannelFuture regFuture = initAndRegister();  
 final Channel channel = regFuture.channel();  
 if (regFuture.cause() != null) {  
 return regFuture;  
 }  
  
 if (regFuture.isDone()) {  
 // At this point we know that the registration was complete and successful.  
 ChannelPromise promise = channel.newPromise();  
 doBind0(regFuture, channel, localAddress, promise);  
 return promise;  
 } else {  
 // Registration future is almost always fulfilled already, but just in case it's not.  
 final PendingRegistrationPromise promise = new PendingRegistrationPromise(channel);  
 regFuture.addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 Throwable cause = future.cause();  
 if (cause != null) {  
 // Registration on the EventLoop failed so fail the ChannelPromise directly to not cause an  
 // IllegalStateException once we try to access the EventLoop of the Channel.  
 promise.setFailure(cause);  
 } else {  
 // Registration was successful, so set the correct executor to use.  
 // See https://github.com/netty/netty/issues/2586  
 promise.registered();  
  
 doBind0(regFuture, channel, localAddress, promise);  
 }  
 }  
 });  
 return promise;  
 }  
}

- 6.1.1) AbstractBootstrap#initAndRegister

* ServerBootstrap重写了init方法

void init(Channel channel) throws Exception {  
 final Map<ChannelOption<?>, Object> options = options0();  
 synchronized (options) {  
 setChannelOptions(channel, options, logger);  
 }  
  
 final Map<AttributeKey<?>, Object> attrs = attrs0();  
 synchronized (attrs) {  
 for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {  
 @SuppressWarnings("unchecked")  
 AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();  
 channel.attr(key).set(e.getValue());  
 }  
 }  
  
 ChannelPipeline p = channel.pipeline();  
  
 final EventLoopGroup currentChildGroup = childGroup;  
 final ChannelHandler currentChildHandler = childHandler;  
 final Entry<ChannelOption<?>, Object>[] currentChildOptions;  
 final Entry<AttributeKey<?>, Object>[] currentChildAttrs;  
 synchronized (childOptions) {  
 currentChildOptions = childOptions.entrySet().toArray(newOptionArray(childOptions.size()));  
 }  
 synchronized (childAttrs) {  
 currentChildAttrs = childAttrs.entrySet().toArray(newAttrArray(childAttrs.size()));  
 }  
  
 p.addLast(new ChannelInitializer<Channel>() {  
 @Override  
 public void initChannel(final Channel ch) throws Exception {  
 final ChannelPipeline pipeline = ch.pipeline();  
 ChannelHandler handler = config.handler();  
 if (handler != null) {  
 pipeline.addLast(handler);  
 }  
  
 ch.eventLoop().execute(new Runnable() {  
 @Override  
 public void run() {  
 pipeline.addLast(new ServerBootstrapAcceptor(  
 ch, currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  
 }  
 });  
 }  
 });  
}

* 这里为pipeline添加了一个ServerBootstrapAcceptor。

#### ServerBootstrapAcceptor

* ServerBootstrapAcceptor 中的 childGroup 是构造此对象是传入的 currentChildGroup, 即我们的 workerGroup, 而 Channel 是一个 NioSocketChannel 的实例, 因此这里的 childGroup.register 就是将 workerGroup 中的每个 EventLoop 和 NioSocketChannel 关联了。
* 在服务器 NioServerSocketChannel 的 pipeline 中添加的是 handler 与 ServerBootstrapAcceptor.
* 当有新的客户端连接请求时, ServerBootstrapAcceptor.channelRead 中负责新建此连接的 NioSocketChannel 并添加 childHandler 到 NioSocketChannel 对应的 pipeline 中, 并将此 channel 绑定到 workerGroup 中的某个 eventLoop 中.
* handler 是在 accept 阶段起作用, 它处理客户端的连接请求.
* childHandler 是在客户端连接建立以后起作用, 它负责客户端连接的 IO 交互.

private static class ServerBootstrapAcceptor extends ChannelInboundHandlerAdapter {  
  
 private final EventLoopGroup childGroup;  
 private final ChannelHandler childHandler;  
 private final Entry<ChannelOption<?>, Object>[] childOptions;  
 private final Entry<AttributeKey<?>, Object>[] childAttrs;  
 private final Runnable enableAutoReadTask;  
  
 ServerBootstrapAcceptor(  
 final Channel channel, EventLoopGroup childGroup, ChannelHandler childHandler,  
 Entry<ChannelOption<?>, Object>[] childOptions, Entry<AttributeKey<?>, Object>[] childAttrs) {  
 this.childGroup = childGroup;  
 this.childHandler = childHandler;  
 this.childOptions = childOptions;  
 this.childAttrs = childAttrs;  
  
 // Task which is scheduled to re-enable auto-read.  
 // It's important to create this Runnable before we try to submit it as otherwise the URLClassLoader may  
 // not be able to load the class because of the file limit it already reached.  
 //  
 // See https://github.com/netty/netty/issues/1328  
 enableAutoReadTask = new Runnable() {  
 @Override  
 public void run() {  
 channel.config().setAutoRead(true);  
 }  
 };  
 }  
  
 @Override  
 @SuppressWarnings("unchecked")  
 public void channelRead(ChannelHandlerContext ctx, Object msg) {

* // 客户端的连接  
   final Channel child = (Channel) msg;  
   // 为其加入handler  
   child.pipeline().addLast(childHandler);
* setChannelOptions(child, childOptions, logger);  
    
  for (Entry<AttributeKey<?>, Object> e: childAttrs) {  
   child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());  
  }  
    
  try {
* // 类似于客户端中的将客户端连接注册到eventLoopGroup中，实际上是将channel绑定到一个eventLoop中

// 见客户端部分的6.1.1.3  
 childGroup.register(child).addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (!future.isSuccess()) {  
 forceClose(child, future.cause());  
 }  
 }  
 });  
 } catch (Throwable t) {  
 forceClose(child, t);  
 }  
 }  
  
 private static void forceClose(Channel child, Throwable t) {  
 child.unsafe().closeForcibly();  
 logger.warn("Failed to register an accepted channel: {}", child, t);  
 }  
  
 @Override  
 public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {  
 final ChannelConfig config = ctx.channel().config();  
 if (config.isAutoRead()) {  
 // stop accept new connections for 1 second to allow the channel to recover  
 // See https://github.com/netty/netty/issues/1328  
 config.setAutoRead(false);  
 ctx.channel().eventLoop().schedule(enableAutoReadTask, 1, TimeUnit.SECONDS);  
 }  
 // still let the exceptionCaught event flow through the pipeline to give the user  
 // a chance to do something with it  
 ctx.fireExceptionCaught(cause);  
 }  
}

* 那么现在的问题是, ServerBootstrapAcceptor.channelRead 方法是怎么被调用的呢? 其实当一个 client 连接到 server 时, Java 底层的 NIO ServerSocketChannel 会有一个 SelectionKey.OP\_ACCEPT 就绪事件, 接着就会调用到 NioServerSocketChannel.doReadMessages。
* 在 doReadMessages 中, 通过 javaChannel().accept() 获取到客户端新连接的 SocketChannel, 接着就实例化一个 NioSocketChannel, 并且传入 NioServerSocketChannel 对象(即 this), 由此可知, 我们创建的这个 NioSocketChannel 的父 Channel 就是 NioServerSocketChannel 实例。
* 接下来就经由 Netty 的 ChannelPipeline 机制, 将读取事件逐级发送到各个 handler 中, 于是就会触发前面我们提到的 ServerBootstrapAcceptor.channelRead 方法。
  + 6.1.2) AbstractBootstrap#doBind0

private static void doBind0(  
 final ChannelFuture regFuture, final Channel channel,  
 final SocketAddress localAddress, final ChannelPromise promise) {  
  
 // This method is invoked before channelRegistered() is triggered. Give user handlers a chance to set up  
 // the pipeline in its channelRegistered() implementation.  
 channel.eventLoop().execute(new Runnable() {  
 @Override  
 public void run() {  
 if (regFuture.isSuccess()) {  
 channel.bind(localAddress, promise).addListener(ChannelFutureListener.CLOSE\_ON\_FAILURE);  
 } else {  
 promise.setFailure(regFuture.cause());  
 }  
 }  
 });  
}

- 6.1.2.1) DefaultChannelPipeline#bind

public final ChannelFuture bind(SocketAddress localAddress, ChannelPromise promise) {  
 return tail.bind(localAddress, promise);  
}

- 6.1.2.1.1) AbstractChannelHandlerContext#bind

public ChannelFuture bind(final SocketAddress localAddress, final ChannelPromise promise) {  
 if (localAddress == null) {  
 throw new NullPointerException("localAddress");  
 }  
 if (isNotValidPromise(promise, false)) {  
 // cancelled  
 return promise;  
 }  
  
 final AbstractChannelHandlerContext next = findContextOutbound();  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 next.invokeBind(localAddress, promise);  
 } else {  
 safeExecute(executor, new Runnable() {  
 @Override  
 public void run() {  
 next.invokeBind(localAddress, promise);  
 }  
 }, promise, null);  
 }  
 return promise;  
}

- 6.1.2.1.1.1) DefaultChannelPipne#bind

public void bind(  
 ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)  
 throws Exception {  
 unsafe.bind(localAddress, promise);  
}

- 6.1.2.1.1.1.1) AbstractChannel#bind

public final void bind(final SocketAddress localAddress, final ChannelPromise promise) {  
 assertEventLoop();  
  
 if (!promise.setUncancellable() || !ensureOpen(promise)) {  
 return;  
 }  
  
 // See: https://github.com/netty/netty/issues/576  
 if (Boolean.TRUE.equals(config().getOption(ChannelOption.SO\_BROADCAST)) &&  
 localAddress instanceof InetSocketAddress &&  
 !((InetSocketAddress) localAddress).getAddress().isAnyLocalAddress() &&  
 !PlatformDependent.isWindows() && !PlatformDependent.maybeSuperUser()) {  
 // Warn a user about the fact that a non-root user can't receive a  
 // broadcast packet on \*nix if the socket is bound on non-wildcard address.  
 logger.warn(  
 "A non-root user can't receive a broadcast packet if the socket " +  
 "is not bound to a wildcard address; binding to a non-wildcard " +  
 "address (" + localAddress + ") anyway as requested.");  
 }  
  
 boolean wasActive = isActive();  
 try {  
 doBind(localAddress);  
 } catch (Throwable t) {  
 safeSetFailure(promise, t);  
 closeIfClosed();  
 return;  
 }  
  
 if (!wasActive && isActive()) {  
 invokeLater(new Runnable() {  
 @Override  
 public void run() {  
 pipeline.fireChannelActive();  
 }  
 });  
 }  
  
 safeSetSuccess(promise);  
}

- 6.1.2.1.1.1.1.1) NioServerSocketChannel#doBind  
- protected void doBind(SocketAddress localAddress) throws Exception {  
if (PlatformDependent.javaVersion() >= 7) {  
 javaChannel().bind(localAddress, config.getBacklog());  
} else {  
 javaChannel().socket().bind(localAddress, config.getBacklog());  
}

}

# EventLoop

* NioEventLoop在其父类SingleThreadEventExecutor中有一个thread属性，代表了该EventLoop所绑定的线程。
* 那么该线程是如何启动的呢？

# 启动

- 6.1.1.3.2.1) AbstractChannel#register中调用了executor.execute()，这里的executor就是NioEventLoop。  
- 1) SingleThreadEventExecutor#execute

* 操作系统会给每个线程分配一小段CPU时间，得到调度的线程可以占有一段时间的CPU时间，而时间用完了就需要再次排队等候调度，SingleThreadEventExecutor将分配给他的线程存储起来，只是作为一种标志，在进行任务执行之前，它都会进行一次判断，当前线程是否处于分配给自己的线程之中，如果不在，那么就不能执行。当然这种管理是在存在多个SingleThreadEventExecutor的时候进行的，在当个SingleThreadEventExecutor内部，如果在执行任务之前判断发现当前线程不在分配给自己的线程之中，那么只有一种可能，那就是当前的SingleThreadEventExecutor还没有被分配线程，那么就执行上面提到的startThread方法来分配一个线程给自己。

public void execute(Runnable task) {  
 if (task == null) {  
 throw new NullPointerException("task");  
 }  
  
 boolean inEventLoop = inEventLoop();  
 if (inEventLoop) {  
 addTask(task);  
 } else {  
 startThread();  
 addTask(task);  
 if (isShutdown() && removeTask(task)) {  
 reject();  
 }  
 }  
  
 if (!addTaskWakesUp && wakesUpForTask(task)) {  
 wakeup(inEventLoop);  
 }  
}

public boolean inEventLoop() {  
 return inEventLoop(Thread.currentThread());  
}

public boolean inEventLoop(Thread thread) {  
 return thread == this.thread;  
}

- 1.1) SingleThreadEventExecutor#startThread（进入事件循环)

private void startThread() {  
 if (state == ST\_NOT\_STARTED) {  
 if (STATE\_UPDATER.compareAndSet(this, ST\_NOT\_STARTED, ST\_STARTED)) {  
 try {  
 doStartThread();  
 } catch (Throwable cause) {  
 STATE\_UPDATER.set(this, ST\_NOT\_STARTED);  
 PlatformDependent.throwException(cause);  
 }  
 }  
 }  
}

private void doStartThread() {  
 assert thread == null;

// 在一个新线程中执行该runnable  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {

* // 得到线程  
   thread = Thread.currentThread();  
   if (interrupted) {  
   thread.interrupt();  
   }
* boolean success = false;  
  updateLastExecutionTime();  
  try {
* // 开始事件循环  
   SingleThreadEventExecutor.this.run();  
   success = true;  
  } catch (Throwable t) {  
   logger.warn("Unexpected exception from an event executor: ", t);  
  } finally {  
   for (;;) {  
   int oldState = state;  
   if (oldState >= ST*SHUTTING*DOWN || STATE*UPDATER.compareAndSet(*  
   *SingleThreadEventExecutor.this, oldState, ST*SHUTTING\_DOWN)) {  
   break;  
   }  
   }
* // Check if confirmShutdown() was called at the end of the loop.  
  if (success && gracefulShutdownStartTime == 0) {  
   logger.error("Buggy " + EventExecutor.class.getSimpleName() + " implementation; " +  
   SingleThreadEventExecutor.class.getSimpleName() + ".confirmShutdown() must be called " +  
   "before run() implementation terminates.");  
  }  
    
  try {  
   // Run all remaining tasks and shutdown hooks.  
   for (;;) {  
   if (confirmShutdown()) {  
   break;  
   }  
   }  
  } finally {  
   try {  
   cleanup();  
   } finally {  
   STATE\_UPDATER.set(SingleThreadEventExecutor.this, ST\_TERMINATED);  
   threadLock.release();  
   if (!taskQueue.isEmpty()) {  
   logger.warn(  
   "An event executor terminated with " +  
   "non-empty task queue (" + taskQueue.size() + ')');  
   }  
    
   terminationFuture.setSuccess(null);  
   }  
  }
* }  
  }  
  });  
  }
  + 1.1.1) ThreadPerTaskExecutor#execute（创建EventLoop所对应线程)

public void execute(Runnable command) {  
 threadFactory.newThread(command).start();  
}

* DefaultThreadFactory#newThread

public Thread newThread(Runnable r) {  
 Thread t = newThread(FastThreadLocalRunnable.wrap(r), prefix + nextId.incrementAndGet());  
 try {  
 if (t.isDaemon() != daemon) {  
 t.setDaemon(daemon);  
 }  
  
 if (t.getPriority() != priority) {  
 t.setPriority(priority);  
 }  
 } catch (Exception ignored) {  
 // Doesn't matter even if failed to set.  
 }  
 return t;  
}

* static Runnable wrap(Runnable runnable) {  
  return runnable instanceof FastThreadLocalRunnable ? runnable : new FastThreadLocalRunnable(runnable);  
  }
* protected Thread newThread(Runnable r, String name) {  
  return new FastThreadLocalThread(threadGroup, r, name);  
  }

public FastThreadLocalThread(ThreadGroup group, Runnable target, String name) {  
 super(group, FastThreadLocalRunnable.wrap(target), name);  
 cleanupFastThreadLocals = true;  
}

public Thread(ThreadGroup group, Runnable target, String name) {  
 init(group, target, name, 0);  
}

* 在该新线程执行时，会调用NioEventLoop#run方法
  + 1.1.1) NioEventLoop#run（开始事件循环)
  + protected void run() {  
    for (;;) {  
     try {  
     switch (selectStrategy.calculateStrategy(selectNowSupplier, hasTasks())) {  
     case SelectStrategy.CONTINUE:  
     continue;  
     case SelectStrategy.SELECT:  
     select(wakenUp.getAndSet(false));
  + // 'wakenUp.compareAndSet(false, true)' is always evaluated  
     // before calling 'selector.wakeup()' to reduce the wake-up  
     // overhead. (Selector.wakeup() is an expensive operation.)  
     //  
     // However, there is a race condition in this approach.  
     // The race condition is triggered when 'wakenUp' is set to  
     // true too early.  
     //  
     // 'wakenUp' is set to true too early if:  
     // 1) Selector is waken up between 'wakenUp.set(false)' and  
     // 'selector.select(...)'. (BAD)  
     // 2) Selector is waken up between 'selector.select(...)' and  
     // 'if (wakenUp.get()) { ... }'. (OK)  
     //  
     // In the first case, 'wakenUp' is set to true and the  
     // following 'selector.select(...)' will wake up immediately.  
     // Until 'wakenUp' is set to false again in the next round,  
     // 'wakenUp.compareAndSet(false, true)' will fail, and therefore  
     // any attempt to wake up the Selector will fail, too, causing  
     // the following 'selector.select(...)' call to block  
     // unnecessarily.  
     //  
     // To fix this problem, we wake up the selector again if wakenUp  
     // is true immediately after selector.select(...).  
     // It is inefficient in that it wakes up the selector for both  
     // the first case (BAD - wake-up required) and the second case  
     // (OK - no wake-up required).  
      
     if (wakenUp.get()) {  
     selector.wakeup();  
     }  
     // fall through  
     default:  
     }  
      
     cancelledKeys = 0;  
     needsToSelectAgain = false;  
     final int ioRatio = this.ioRatio;
* // Netty中可以设置用于I/O操作和非I/O操作的时间占比，默认各位50%，也就是说，如果某次I/O操作的时间花了100ms，那么这次循环中非I/O得任务也可以花费100ms。Netty中的I/O时间处理通过processSelectedKeys方法来进行，而非I/O操作通过runAllTasks反复来进行
* if (ioRatio == 100) {  
   try {  
   processSelectedKeys();  
   } finally {  
   // Ensure we always run tasks.  
   runAllTasks();  
   }  
  } else {  
   final long ioStartTime = System.nanoTime();  
   try {  
   processSelectedKeys();  
   } finally {  
   // Ensure we always run tasks.  
   final long ioTime = System.nanoTime() - ioStartTime;  
   runAllTasks(ioTime \* (100 - ioRatio) / ioRatio);  
   }  
  }  
  } catch (Throwable t) {  
   handleLoopException(t);  
  }  
  // Always handle shutdown even if the loop processing threw an exception.  
  try {  
   if (isShuttingDown()) {  
   closeAll();  
   if (confirmShutdown()) {  
   return;  
   }  
   }  
  } catch (Throwable t) {  
   handleLoopException(t);  
  }  
  }  
  }

# 任务添加

* 每个SingleThreadEventExecutor仅有一个线程来进行事件处理，而每个SingleThreadEventExecutor包含了一个任务队列，所有需要该SingleThreadEventExecutor处理的任务都需要添加到该任务队列中去，添加的任务会在事件循环的过程中被不断消费，下面来看一下一个任务时如何被SingleThreadEventExecutor执行的。
* 其中持有了一个任务队列，类型为LinkedBlockingQueue。
* SingleThreadEventExecutor#addTask
* protected void addTask(Runnable task) {  
  if (task == null) {  
   throw new NullPointerException("task");  
  }  
  if (!offerTask(task)) {  
   reject(task);  
  }  
  }
* final boolean offerTask(Runnable task) {  
  if (isShutdown()) {  
   reject();  
  }  
  return taskQueue.offer(task);  
  }
* protected final void reject(Runnable task) {  
  rejectedExecutionHandler.rejected(task, this);  
  }

# 任务执行

* 通过 Selector.open() 打开一个 Selector.
* 将 Channel 注册到 Selector 中, 并设置需要监听的事件(interest set)
* 不断重复:
* 调用 select() 方法
* 调用 selector.selectedKeys() 获取 selected keys
* 迭代每个 selected key:
  + 1) 从 selected key 中获取 对应的 Channel 和附加信息(如果有的话)
  + 2) 判断是哪些 IO 事件已经就绪了, 然后处理它们. 如果是 OP\_ACCEPT 事件, 则调用 "SocketChannel clientChannel = ((ServerSocketChannel) key.channel()).accept()" 获取 SocketChannel, 并将它设置为 非阻塞的, 然后将这个 Channel 注册到 Selector 中.
  + 3) 根据需要更改 selected key 的监听事件.
  + 4) 将已经处理过的 key 从 selected keys 集合中删除.
* 在NioEventLoop#run中，会先processSelectedKeys，然后runAllTasks。
  + 1) NioEventLoop#processSelectedKeys（处理IO事件)

private void processSelectedKeys() {  
 if (selectedKeys != null) {  
 processSelectedKeysOptimized();  
 } else {  
 processSelectedKeysPlain(selector.selectedKeys());  
 }  
}

private void processSelectedKeysPlain(Set<SelectionKey> selectedKeys) {  
 // check if the set is empty and if so just return to not create garbage by  
 // creating a new Iterator every time even if there is nothing to process.  
 // See https://github.com/netty/netty/issues/597  
 if (selectedKeys.isEmpty()) {  
 return;  
 }  
  
 Iterator<SelectionKey> i = selectedKeys.iterator();  
 for (;;) {  
 final SelectionKey k = i.next();  
 final Object a = k.attachment();  
 i.remove();  
  
 if (a instanceof AbstractNioChannel) {  
 processSelectedKey(k, (AbstractNioChannel) a);  
 } else {  
 @SuppressWarnings("unchecked")  
 NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;  
 processSelectedKey(k, task);  
 }  
  
 if (!i.hasNext()) {  
 break;  
 }  
  
 if (needsToSelectAgain) {  
 selectAgain();  
 selectedKeys = selector.selectedKeys();  
  
 // Create the iterator again to avoid ConcurrentModificationException  
 if (selectedKeys.isEmpty()) {  
 break;  
 } else {  
 i = selectedKeys.iterator();  
 }  
 }  
 }  
}

private void processSelectedKey(SelectionKey k, AbstractNioChannel ch) {  
 final AbstractNioChannel.NioUnsafe unsafe = ch.unsafe();  
 if (!k.isValid()) {  
 final EventLoop eventLoop;  
 try {  
 eventLoop = ch.eventLoop();  
 } catch (Throwable ignored) {  
 // If the channel implementation throws an exception because there is no event loop, we ignore this  
 // because we are only trying to determine if ch is registered to this event loop and thus has authority  
 // to close ch.  
 return;  
 }  
 // Only close ch if ch is still registered to this EventLoop. ch could have deregistered from the event loop  
 // and thus the SelectionKey could be cancelled as part of the deregistration process, but the channel is  
 // still healthy and should not be closed.  
 // See https://github.com/netty/netty/issues/5125  
 if (eventLoop != this || eventLoop == null) {  
 return;  
 }  
 // close the channel if the key is not valid anymore  
 unsafe.close(unsafe.voidPromise());  
 return;  
 }  
  
 try {  
 int readyOps = k.readyOps();  
 // We first need to call finishConnect() before try to trigger a read(...) or write(...) as otherwise  
 // the NIO JDK channel implementation may throw a NotYetConnectedException.  
 if ((readyOps & SelectionKey.OP\_CONNECT) != 0) {  
 // remove OP\_CONNECT as otherwise Selector.select(..) will always return without blocking  
 // See https://github.com/netty/netty/issues/924  
 int ops = k.interestOps();  
 ops &= ~SelectionKey.OP\_CONNECT;  
 k.interestOps(ops);  
  
 unsafe.finishConnect();  
 }  
  
 // Process OP\_WRITE first as we may be able to write some queued buffers and so free memory.  
 if ((readyOps & SelectionKey.OP\_WRITE) != 0) {  
 // Call forceFlush which will also take care of clear the OP\_WRITE once there is nothing left to write  
 ch.unsafe().forceFlush();  
 }  
  
 // Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead  
 // to a spin loop  
 if ((readyOps & (SelectionKey.OP\_READ | SelectionKey.OP\_ACCEPT)) != 0 || readyOps == 0) {  
 unsafe.read();  
 }  
 } catch (CancelledKeyException ignored) {  
 unsafe.close(unsafe.voidPromise());  
 }  
}

- 1.1) AbstractNioChannel#finishConnect（关闭连接)

public final void finishConnect() {  
 // Note this method is invoked by the event loop only if the connection attempt was  
 // neither cancelled nor timed out.  
  
 assert eventLoop().inEventLoop();  
  
 try {  
 boolean wasActive = isActive();  
 doFinishConnect();  
 fulfillConnectPromise(connectPromise, wasActive);  
 } catch (Throwable t) {  
 fulfillConnectPromise(connectPromise, annotateConnectException(t, requestedRemoteAddress));  
 } finally {  
 // Check for null as the connectTimeoutFuture is only created if a connectTimeoutMillis > 0 is used  
 // See https://github.com/netty/netty/issues/1770  
 if (connectTimeoutFuture != null) {  
 connectTimeoutFuture.cancel(false);  
 }  
 connectPromise = null;  
 }  
}

- 1.1.1) NioSocketChannel#doFinishConnect

* protected void doFinishConnect() throws Exception {  
  if (!javaChannel().finishConnect()) {  
   throw new Error();  
  }  
  }
  + 1.1.2) AbstractNioChannel#fulfillConnectPromise

private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {  
 if (promise == null) {  
 // Closed via cancellation and the promise has been notified already.  
 return;  
 }  
  
 // Get the state as trySuccess() may trigger an ChannelFutureListener that will close the Channel.  
 // We still need to ensure we call fireChannelActive() in this case.  
 boolean active = isActive();  
  
 // trySuccess() will return false if a user cancelled the connection attempt.  
 boolean promiseSet = promise.trySuccess();  
  
 // Regardless if the connection attempt was cancelled, channelActive() event should be triggered,  
 // because what happened is what happened.  
 if (!wasActive && active) {  
 pipeline().fireChannelActive();  
 }  
  
 // If a user cancelled the connection attempt, close the channel, which is followed by channelInactive().  
 if (!promiseSet) {  
 close(voidPromise());  
 }  
}

- 1.2) AbstractNioChannel#forceFlush（处理写事件)

public final void forceFlush() {  
 // directly call super.flush0() to force a flush now  
 super.flush0();  
}

protected void flush0() {  
 if (inFlush0) {  
 // Avoid re-entrance  
 return;  
 }  
  
 final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
 if (outboundBuffer == null || outboundBuffer.isEmpty()) {  
 return;  
 }  
  
 inFlush0 = true;  
  
 // Mark all pending write requests as failure if the channel is inactive.  
 if (!isActive()) {  
 try {  
 if (isOpen()) {  
 outboundBuffer.failFlushed(FLUSH0\_NOT\_YET\_CONNECTED\_EXCEPTION, true);  
 } else {  
 // Do not trigger channelWritabilityChanged because the channel is closed already.  
 outboundBuffer.failFlushed(FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 }  
 } finally {  
 inFlush0 = false;  
 }  
 return;  
 }  
  
 try {  
 doWrite(outboundBuffer);  
 } catch (Throwable t) {  
 if (t instanceof IOException && config().isAutoClose()) {  
 /\*\*  
 \* Just call {@link #close(ChannelPromise, Throwable, boolean)} here which will take care of  
 \* failing all flushed messages and also ensure the actual close of the underlying transport  
 \* will happen before the promises are notified.  
 \*  
 \* This is needed as otherwise {@link #isActive()} , {@link #isOpen()} and {@link #isWritable()}  
 \* may still return {@code true} even if the channel should be closed as result of the exception.  
 \*/  
 close(voidPromise(), t, FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 } else {  
 try {  
 shutdownOutput(voidPromise(), t);  
 } catch (Throwable t2) {  
 close(voidPromise(), t2, FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 }  
 }  
 } finally {  
 inFlush0 = false;  
 }  
}

- 1.2.1) NioSocketChannel#doWrite

* protected void doWrite(ChannelOutboundBuffer in) throws Exception {  
  SocketChannel ch = javaChannel();  
  int writeSpinCount = config().getWriteSpinCount();  
  do {  
   if (in.isEmpty()) {  
   // All written so clear OP\_WRITE  
   clearOpWrite();  
   // Directly return here so incompleteWrite(...) is not called.  
   return;  
   }
* // Ensure the pending writes are made of ByteBufs only.  
  int maxBytesPerGatheringWrite = ((NioSocketChannelConfig) config).getMaxBytesPerGatheringWrite();  
  ByteBuffer[] nioBuffers = in.nioBuffers(1024, maxBytesPerGatheringWrite);  
  int nioBufferCnt = in.nioBufferCount();  
    
  // Always us nioBuffers() to workaround data-corruption.  
  // See https://github.com/netty/netty/issues/2761  
  switch (nioBufferCnt) {  
   case 0:  
   // We have something else beside ByteBuffers to write so fallback to normal writes.  
   writeSpinCount -= doWrite0(in);  
   break;  
   case 1: {  
   // Only one ByteBuf so use non-gathering write  
   // Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
   // to check if the total size of all the buffers is non-zero.  
   ByteBuffer buffer = nioBuffers[0];  
   int attemptedBytes = buffer.remaining();  
   final int localWrittenBytes = ch.write(buffer);  
   if (localWrittenBytes <= 0) {  
   incompleteWrite(true);  
   return;  
   }  
   adjustMaxBytesPerGatheringWrite(attemptedBytes, localWrittenBytes, maxBytesPerGatheringWrite);  
   in.removeBytes(localWrittenBytes);  
   --writeSpinCount;  
   break;  
   }  
   default: {  
   // Zero length buffers are not added to nioBuffers by ChannelOutboundBuffer, so there is no need  
   // to check if the total size of all the buffers is non-zero.  
   // We limit the max amount to int above so cast is safe  
   long attemptedBytes = in.nioBufferSize();  
   final long localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);  
   if (localWrittenBytes <= 0) {  
   incompleteWrite(true);  
   return;  
   }  
   // Casting to int is safe because we limit the total amount of data in the nioBuffers to int above.  
   adjustMaxBytesPerGatheringWrite((int) attemptedBytes, (int) localWrittenBytes,  
   maxBytesPerGatheringWrite);  
   in.removeBytes(localWrittenBytes);  
   --writeSpinCount;  
   break;  
   }  
  }
* } while (writeSpinCount > 0);
* incompleteWrite(writeSpinCount < 0);  
  }
* 1.3-a) （服务器接收连接，从属于bossGroup，read只是表面上的)NioMessageUnsafe#read

public void read() {  
 assert eventLoop().inEventLoop();  
 final ChannelConfig config = config();  
 final ChannelPipeline pipeline = pipeline();  
 final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();  
 allocHandle.reset(config);  
  
 boolean closed = false;  
 Throwable exception = null;  
 try {  
 try {  
 do {  
 int localRead = doReadMessages(readBuf);  
 if (localRead == 0) {  
 break;  
 }  
 if (localRead < 0) {  
 closed = true;  
 break;  
 }  
  
 allocHandle.incMessagesRead(localRead);  
 } while (allocHandle.continueReading());  
 } catch (Throwable t) {  
 exception = t;  
 }  
  
 int size = readBuf.size();  
 for (int i = 0; i < size; i ++) {  
 readPending = false;

* //对于server端来说，第一个handler为io.netty.bootstrap.ServerBootstrap$ServerBootstrapAcceptor  
   pipeline.fireChannelRead(readBuf.get(i));  
  }  
  readBuf.clear();  
  allocHandle.readComplete();  
  pipeline.fireChannelReadComplete();
* if (exception != null) {  
   closed = closeOnReadError(exception);
* pipeline.fireExceptionCaught(exception);
* }
* if (closed) {  
   inputShutdown = true;  
   if (isOpen()) {  
   close(voidPromise());  
   }  
  }  
  } finally {  
   // Check if there is a readPending which was not processed yet.  
   // This could be for two reasons:  
   // \* The user called Channel.read() or ChannelHandlerContext.read() in channelRead(...) method  
   // \* The user called Channel.read() or ChannelHandlerContext.read() in channelReadComplete(...) method  
   //  
   // See https://github.com/netty/netty/issues/2254  
   if (!readPending && !config.isAutoRead()) {  
   removeReadOp();  
   }  
  }  
  }
  + 1.3.1) NioServerSocketChannel#doReadMessages（接收连接)
  + protected int doReadMessages(List buf) throws Exception {  
    SocketChannel ch = SocketUtils.accept(javaChannel());
* try {  
   if (ch != null) {  
   buf.add(new NioSocketChannel(this, ch));  
   return 1;  
   }  
  } catch (Throwable t) {  
   logger.warn("Failed to create a new channel from an accepted socket.", t);
* try {  
   ch.close();  
  } catch (Throwable t2) {  
   logger.warn("Failed to close a socket.", t2);  
  }
* }
* return 0;  
  }
  + 1.3.1.1) SocketUtils#accept

public static SocketChannel accept(final ServerSocketChannel serverSocketChannel) throws IOException {  
 try {  
 return AccessController.doPrivileged(new PrivilegedExceptionAction<SocketChannel>() {  
 @Override  
 public SocketChannel run() throws IOException {  
 return serverSocketChannel.accept();  
 }  
 });  
 } catch (PrivilegedActionException e) {  
 throw (IOException) e.getCause();  
 }  
}

* 1.3-b) （服务器接收数据，从属于workerGroup)AbstractNioByteChannel#read  
  1.- 分配 ByteBuf  
  2.- 从 SocketChannel 中读取数据  
  3.- 调用 pipeline.fireChannelRead 发送一个 inbound 事件，沿着pipeline不断传递，经过解码器，最终到达用户handler。

public final void read() {  
 final ChannelConfig config = config();  
 final ChannelPipeline pipeline = pipeline();  
 final ByteBufAllocator allocator = config.getAllocator();  
 final RecvByteBufAllocator.Handle allocHandle = recvBufAllocHandle();  
 allocHandle.reset(config);  
  
 ByteBuf byteBuf = null;  
 boolean close = false;  
 try {  
 do {  
 byteBuf = allocHandle.allocate(allocator);  
 allocHandle.lastBytesRead(doReadBytes(byteBuf));  
 if (allocHandle.lastBytesRead() <= 0) {  
 // nothing was read. release the buffer.  
 byteBuf.release();  
 byteBuf = null;  
 close = allocHandle.lastBytesRead() < 0;  
 if (close) {  
 // There is nothing left to read as we received an EOF.  
 readPending = false;  
 }  
 break;  
 }  
  
 allocHandle.incMessagesRead(1);  
 readPending = false;  
 pipeline.fireChannelRead(byteBuf);  
 byteBuf = null;  
 } while (allocHandle.continueReading());  
  
 allocHandle.readComplete();  
 pipeline.fireChannelReadComplete();  
  
 if (close) {  
 closeOnRead(pipeline);  
 }  
 } catch (Throwable t) {  
 handleReadException(pipeline, byteBuf, t, close, allocHandle);  
 } finally {  
 // Check if there is a readPending which was not processed yet.  
 // This could be for two reasons:  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelRead(...) method  
 // \* The user called Channel.read() or ChannelHandlerContext.read() in channelReadComplete(...) method  
 //  
 // See https://github.com/netty/netty/issues/2254  
 if (!readPending && !config.isAutoRead()) {  
 removeReadOp();  
 }  
 }  
}

- 1.3.1) NioSocketChannel#doReadBytes

* protected int doReadBytes(ByteBuf byteBuf) throws Exception {  
  final RecvByteBufAllocator.Handle allocHandle = unsafe().recvBufAllocHandle();  
  allocHandle.attemptedBytesRead(byteBuf.writableBytes());  
  return byteBuf.writeBytes(javaChannel(), allocHandle.attemptedBytesRead());  
  }
  + 2) SingleThreadEventExecutor#runAllTasks
* protected boolean runAllTasks(long timeoutNanos) {  
  fetchFromScheduledTaskQueue();  
  Runnable task = pollTask();  
  if (task == null) {  
   afterRunningAllTasks();  
   return false;  
  }
* final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;  
  long runTasks = 0;  
  long lastExecutionTime;  
  for (;;) {  
   safeExecute(task);
* runTasks ++;  
    
  // Check timeout every 64 tasks because nanoTime() is relatively expensive.   
  // XXX: Hard-coded value - will make it configurable if it is really a problem.  
  if ((runTasks & 0x3F) == 0) {  
   lastExecutionTime = ScheduledFutureTask.nanoTime();  
   if (lastExecutionTime >= deadline) {  
   break;  
   }  
  }  
    
  task = pollTask();  
  if (task == null) {  
   lastExecutionTime = ScheduledFutureTask.nanoTime();  
   break;  
  }
* }
* afterRunningAllTasks();  
  this.lastExecutionTime = lastExecutionTime;  
  return true;  
  }

### 2.1) fetchFromScheduledTaskQueue

private boolean fetchFromScheduledTaskQueue() {  
 long nanoTime = AbstractScheduledEventExecutor.nanoTime();  
 Runnable scheduledTask = pollScheduledTask(nanoTime);  
 while (scheduledTask != null) {  
 if (!taskQueue.offer(scheduledTask)) {  
 // No space left in the task queue add it back to the scheduledTaskQueue so we pick it up again.  
 scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);  
 return false;  
 }  
 scheduledTask = pollScheduledTask(nanoTime);  
 }  
 return true;  
}

* protected final Runnable pollScheduledTask(long nanoTime) {  
  assert inEventLoop();
* Queue<ScheduledFutureTask<?>> scheduledTaskQueue = this.scheduledTaskQueue;  
  ScheduledFutureTask<?> scheduledTask = scheduledTaskQueue == null ? null : scheduledTaskQueue.peek();  
  if (scheduledTask == null) {  
   return null;  
  }
* if (scheduledTask.deadlineNanos() <= nanoTime) {  
   scheduledTaskQueue.remove();  
   return scheduledTask;  
  }  
  return null;  
  }

### 2.2) pollTask

* protected Runnable pollTask() {  
  assert inEventLoop();  
  return pollTaskFrom(taskQueue);  
  }
* protected static Runnable pollTaskFrom(Queue taskQueue) {  
  for (;;) {  
   Runnable task = taskQueue.poll();  
   if (task == WAKEUP\_TASK) {  
   continue;  
   }  
   return task;  
  }  
  }

### 2.3) safeExecute

* protected static void safeExecute(Runnable task) {  
  try {  
   task.run();  
  } catch (Throwable t) {  
   logger.warn("A task raised an exception. Task: {}", task, t);  
  }  
  }

# 请求处理：服务器先读后写

* 在EventLoop的processSelectedKeys方法中，检测到读事件，于是调用ServerSocketChannel的accept方法，获得客户端的SocketChannel，然后调用pipeline的fireChannelRead方法，传播读事件。
* 注意，pipeline中的链表元素类型为ChannelHandlerContext，并不是ChannelHandler。
* Pipeline流水线的开始就是HeadContxt，流水线的结束就是TailConext，HeadContxt中调用Unsafe做具体的操作，TailConext中用于向用户抛出pipeline中未处理异常以及对未处理消息的警告

# 一、服务器接收连接

* I/O Request
* via Channel or
* ChannelHandlerContext
* |
* +---------------------------------------------------+---------------+
* | ChannelPipeline | |
* | |/ |
* | +---------------------+ +-----------+----------+ |
* | | Inbound Handler N | | Outbound Handler 1 | |
* | +----------+----------+ +-----------+----------+ |
* | /|\ | |
* | | |/ |
* | +----------+----------+ +-----------+----------+ |
* | | Inbound Handler N-1 | | Outbound Handler 2 | |
* | +----------+----------+ +-----------+----------+ |
* | /|\ . |
* | . . |
* | ChannelHandlerContext.fireIN*EVT() ChannelHandlerContext.OUT*EVT()|
* | [ method call] [method call] |
* | . . |
* | . |/ |
* | +----------+----------+ +-----------+----------+ |
* | | Inbound Handler 2 | | Outbound Handler M-1 | |
* | +----------+----------+ +-----------+----------+ |
* | /|\ | |
* | | |/ |
* | +----------+----------+ +-----------+----------+ |
* | | Inbound Handler 1 | | Outbound Handler M | |
* | +----------+----------+ +-----------+----------+ |
* | /|\ | |
* +---------------+-----------------------------------+---------------+
* | |/
* +---------------+-----------------------------------+---------------+
* | | | |
* | [ Socket.read() ] [ Socket.write() ] |
* | |
* | Netty Internal I/O Threads (Transport Implementation) |
* +-------------------------------------------------------------------+
* 对于 Inbound 事件:
* Inbound 事件是通知事件, 当某件事情已经就绪后, 通知上层.
* Inbound 事件发起者是 unsafe
* Inbound 事件的处理者是 Channel, 如果用户没有实现自定义的处理方法, 那么Inbound 事件默认的处理者是 TailContext, 并且其处理方法是空实现.
* Inbound 事件在 Pipeline 中传输方向是 head -> tail
* 在 ChannelHandler 中处理事件时, 如果这个 Handler 不是最后一个 Hnalder, 则需要调用 ctx.fireIN\_EVT (例如 ctx.fireChannelActive) 将此事件继续传播下去. 如果不这样做, 那么此事件的传播会提前终止.
* Outbound 事件流: Context.fireIN*EVT -> Connect.findContextInbound -> nextContext.invokeIN*EVT -> nextHandler.IN*EVT -> nextContext.fireIN*EVT
* NioEventLoop#processSelectedKeys->read
* 发生在NioMessageUnsafe#read处理完毕之后
  + 1) DefaultChannelPipeline#fireChannelRead

public final ChannelPipeline fireChannelRead(Object msg) {  
 AbstractChannelHandlerContext.invokeChannelRead(head, msg);  
 return this;  
}

* 从head开始沿着链表方向传递。
* head是HeadContext
  + 2)（head)AbstractChannelHandlerContext#invokeChannelRead（静态)

static void invokeChannelRead(final AbstractChannelHandlerContext next, Object msg) {  
 final Object m = next.pipeline.touch(ObjectUtil.checkNotNull(msg, "msg"), next);  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 next.invokeChannelRead(m);  
 } else {  
 executor.execute(new Runnable() {  
 @Override  
 public void run() {  
 next.invokeChannelRead(m);  
 }  
 });  
 }  
}

- 3)（head)AbstractChannelHandlerContext#invokeChannelRead（实例，调用其对应Handler的channelRead)

private void invokeChannelRead(Object msg) {  
 if (invokeHandler()) {  
 try {  
 ((ChannelInboundHandler) handler()).channelRead(this, msg);  
 } catch (Throwable t) {  
 notifyHandlerException(t);  
 }  
 } else {  
 fireChannelRead(msg);  
 }  
}

- 4)（head，其handler就是this)DefaultChannelPipeline#channelRead

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
 ctx.fireChannelRead(msg);  
}

- 5)（head，传播)AbstractChannelHandlerContext#fireChannelRead

public ChannelHandlerContext fireChannelRead(final Object msg) {  
 invokeChannelRead(findContextInbound(), msg);  
 return this;  
}

- 5.1) AbstractChannelHandlerContext#findContextInbound（得到下一个Inbound的ctx)

private AbstractChannelHandlerContext findContextInbound() {  
 AbstractChannelHandlerContext ctx = this;  
 do {  
 ctx = ctx.next;  
 } while (!ctx.inbound);  
 return ctx;  
}

- 之后再去调用2)，此时ctx就不是head了，而是head.next，是自定义handler对应的ctx。

* 就服务器而言，head.next可能是ServerBootstrapAcceptor。
  + 6) (ServerBootstrapAcceptor) AbstractChannelHandlerContext#invokeChannelRead（静态)
* 1.boos reactor线程轮询到有新的连接进入
* 2.通过封装jdk底层的channel创建 NioSocketChannel以及一系列的netty核心组件
* 3.将该条连接通过chooser，选择一条worker reactor线程绑定上去
* 4.注册读事件，开始新连接的读写
  + 6.1) ServerBootstrapAcceptor#channelRead（将接收到的连接注册到childGroup中，类似于客户端的6.1.1.3)
  + 1)将childHandler放入新连接的pipeline中（ChannelInitializer)
  + 2)将Channel注册到Selector中，初始化EventLoop线程，注册自定义Handler（调用ChannelInitializer的initChannel，再将自己移除)

public void channelRead(ChannelHandlerContext ctx, Object msg) {  
 final Channel child = (Channel) msg;  
  
 child.pipeline().addLast(childHandler);  
  
 setChannelOptions(child, childOptions, logger);  
  
 for (Entry<AttributeKey<?>, Object> e: childAttrs) {  
 child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());  
 }  
  
 try {  
 childGroup.register(child).addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (!future.isSuccess()) {  
 forceClose(child, future.cause());  
 }  
 }  
 });  
 } catch (Throwable t) {  
 forceClose(child, t);  
 }  
}

# 二、服务器接收数据

* 发生在AbstractNioByteChannel#read处理完毕之后
* 同样也是先经过HeadContext，只不过之后的handler就是用户指定的handler了，比如Decoder、自定义Handler等InboundHandler。
* pipeline中handler依次处理，直至遇到TailContext。其处理方法是空实现。

# 三、服务器发送数据

* 在用户自定义Handler处理完毕后，会调用对应的ctx的writeAndFlush方法。
* AbstractChannelHandlerContext#writeAndFlush
* 此时当前线程可能是在用户自定义的线程池中，也可能就是在IO线程中。
* 假设当前线程是用户自定义线程池中的某个线程，并不在EventLoop线程中。
* 以下将AbstractChannelHandlerContext简写为CTX

public ChannelFuture writeAndFlush(Object msg) {  
 return writeAndFlush(msg, newPromise());  
}

public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {  
 if (msg == null) {  
 throw new NullPointerException("msg");  
 }  
  
 if (isNotValidPromise(promise, true)) {  
 ReferenceCountUtil.release(msg);  
 // cancelled  
 return promise;  
 }  
  
 write(msg, true, promise);  
  
 return promise;  
}

- 1) （next为首个Outbound Handler)CTX#write(msg,flush,promise)

private void write(Object msg, boolean flush, ChannelPromise promise) {  
 AbstractChannelHandlerContext next = findContextOutbound();  
 final Object m = pipeline.touch(msg, next);  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 if (flush) {  
 next.invokeWriteAndFlush(m, promise);  
 } else {  
 next.invokeWrite(m, promise);  
 }  
 } else {  
 AbstractWriteTask task;  
 if (flush) {  
 task = WriteAndFlushTask.newInstance(next, m, promise);  
 } else {  
 task = WriteTask.newInstance(next, m, promise);  
 }  
 safeExecute(executor, task, promise, m);  
 }  
}

- 2) WriteAndFlushTask#newInstance

private static WriteAndFlushTask newInstance(  
 AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise) {  
 WriteAndFlushTask task = RECYCLER.get();  
 init(task, ctx, msg, promise);  
 return task;  
}

* protected static void init(AbstractWriteTask task, AbstractChannelHandlerContext ctx,  
   Object msg, ChannelPromise promise) {  
  task.ctx = ctx;  
  task.msg = msg;  
  task.promise = promise;
* if (ESTIMATE*TASK*SIZE*ON*SUBMIT) {  
   task.size = ctx.pipeline.estimatorHandle().size(msg) + WRITE*TASK*OVERHEAD;  
   ctx.pipeline.incrementPendingOutboundBytes(task.size);  
  } else {  
   task.size = 0;  
  }  
  }
* 该runnable的run方法为：
  + 3) CTX#safeExecute（将写数据封装为任务放到EventLoop的队列中)

private static void safeExecute(EventExecutor executor, Runnable runnable, ChannelPromise promise, Object msg) {  
 try {  
 executor.execute(runnable);  
 } catch (Throwable cause) {  
 try {  
 promise.setFailure(cause);  
 } finally {  
 if (msg != null) {  
 ReferenceCountUtil.release(msg);  
 }  
 }  
 }  
}

- 4) AbstractWriteTask#run

public final void run() {  
 try {  
 // Check for null as it may be set to null if the channel is closed already  
 if (ESTIMATE\_TASK\_SIZE\_ON\_SUBMIT) {  
 ctx.pipeline.decrementPendingOutboundBytes(size);  
 }  
 write(ctx, msg, promise);  
 } finally {  
 // Set to null so the GC can collect them directly  
 ctx = null;  
 msg = null;  
 promise = null;  
 handle.recycle(this);  
 }  
}

### 5) WriteAndFlushTask

public void write(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise) {  
 super.write(ctx, msg, promise);  
 ctx.invokeFlush();  
}

* protected void write(AbstractChannelHandlerContext ctx, Object msg, ChannelPromise promise) {  
  ctx.invokeWrite(msg, promise);  
  }

### WRITE

- 6) CTX#invokeWrite

private void invokeWrite(Object msg, ChannelPromise promise) {  
 if (invokeHandler()) {  
 invokeWrite0(msg, promise);  
 } else {  
 write(msg, promise);  
 }  
}

private void invokeWrite0(Object msg, ChannelPromise promise) {  
 try {  
 ((ChannelOutboundHandler) handler()).write(this, msg, promise);  
 } catch (Throwable t) {  
 notifyOutboundHandlerException(t, promise);  
 }  
}

* 此时会调用用户指定的Outbound Handler写数据，比如Encoder。
  + 7) （首个Outbound Handler)MessageToByteEncoder#write

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {  
 ByteBuf buf = null;  
 try {  
 if (acceptOutboundMessage(msg)) {  
 @SuppressWarnings("unchecked")  
 I cast = (I) msg;  
 buf = allocateBuffer(ctx, cast, preferDirect);  
 try {  
 encode(ctx, cast, buf);  
 } finally {  
 ReferenceCountUtil.release(cast);  
 }  
  
 if (buf.isReadable()) {  
 ctx.write(buf, promise);  
 } else {  
 buf.release();  
 ctx.write(Unpooled.EMPTY\_BUFFER, promise);  
 }  
 buf = null;  
 } else {  
 ctx.write(msg, promise);  
 }  
 } catch (EncoderException e) {  
 throw e;  
 } catch (Throwable e) {  
 throw new EncoderException(e);  
 } finally {  
 if (buf != null) {  
 buf.release();  
 }  
 }

- 8) CTX#write(msg,promise)

public ChannelFuture write(final Object msg, final ChannelPromise promise) {  
 if (msg == null) {  
 throw new NullPointerException("msg");  
 }  
  
 try {  
 if (isNotValidPromise(promise, true)) {  
 ReferenceCountUtil.release(msg);  
 // cancelled  
 return promise;  
 }  
 } catch (RuntimeException e) {  
 ReferenceCountUtil.release(msg);  
 throw e;  
 }  
 write(msg, false, promise);  
  
 return promise;  
}

- 9) （next为第二个Outbound Handler)CTX#write(msg,flush,promise)   
- 同1)，但是此时：  
- 1)flush为false  
- 2)已经处于EventLoop中

* 继续调用第二个Outbound Handler的write
* 10)（第二个Outbound Handler) MessageToMessageEncoder#write

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {  
 CodecOutputList out = null;  
 try {  
 if (acceptOutboundMessage(msg)) {  
 out = CodecOutputList.newInstance();  
 @SuppressWarnings("unchecked")  
 I cast = (I) msg;  
 try {  
 encode(ctx, cast, out);  
 } finally {  
 ReferenceCountUtil.release(cast);  
 }  
  
 if (out.isEmpty()) {  
 out.recycle();  
 out = null;  
  
 throw new EncoderException(  
 StringUtil.simpleClassName(this) + " must produce at least one message.");  
 }  
 } else {  
 ctx.write(msg, promise);  
 }  
 } catch (EncoderException e) {  
 throw e;  
 } catch (Throwable t) {  
 throw new EncoderException(t);  
 } finally {  
 if (out != null) {  
 final int sizeMinusOne = out.size() - 1;  
 if (sizeMinusOne == 0) {  
 ctx.write(out.get(0), promise);  
 } else if (sizeMinusOne > 0) {  
 // Check if we can use a voidPromise for our extra writes to reduce GC-Pressure  
 // See https://github.com/netty/netty/issues/2525  
 ChannelPromise voidPromise = ctx.voidPromise();  
 boolean isVoidPromise = promise == voidPromise;  
 for (int i = 0; i < sizeMinusOne; i ++) {  
 ChannelPromise p;  
 if (isVoidPromise) {  
 p = voidPromise;  
 } else {  
 p = ctx.newPromise();  
 }  
 ctx.write(out.getUnsafe(i), p);  
 }  
 ctx.write(out.getUnsafe(sizeMinusOne), promise);  
 }  
 out.recycle();  
 }  
 }  
}

- 在这里会调用8)，如此循环直至next为HeadContext。  
- 11) （head)DefaultChannelPipeline#write

public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {  
 unsafe.write(msg, promise);  
}

- 12) （unsafe)AbstractChannel#write（放在tailEntry链表中)

public final void write(Object msg, ChannelPromise promise) {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
 if (outboundBuffer == null) {  
 // If the outboundBuffer is null we know the channel was closed and so  
 // need to fail the future right away. If it is not null the handling of the rest  
 // will be done in flush0()  
 // See https://github.com/netty/netty/issues/2362  
 safeSetFailure(promise, WRITE\_CLOSED\_CHANNEL\_EXCEPTION);  
 // release message now to prevent resource-leak  
 ReferenceCountUtil.release(msg);  
 return;  
 }  
  
 int size;  
 try {  
 msg = filterOutboundMessage(msg);  
 size = pipeline.estimatorHandle().size(msg);  
 if (size < 0) {  
 size = 0;  
 }  
 } catch (Throwable t) {  
 safeSetFailure(promise, t);  
 ReferenceCountUtil.release(msg);  
 return;  
 }  
  
 outboundBuffer.addMessage(msg, size, promise);  
}

public void addMessage(Object msg, int size, ChannelPromise promise) {  
 Entry entry = Entry.newInstance(msg, size, total(msg), promise);  
 if (tailEntry == null) {  
 flushedEntry = null;  
 tailEntry = entry;  
 } else {  
 Entry tail = tailEntry;  
 tail.next = entry;  
 tailEntry = entry;  
 }  
 if (unflushedEntry == null) {  
 unflushedEntry = entry;  
 }  
  
 // increment pending bytes after adding message to the unflushed arrays.  
 // See https://github.com/netty/netty/issues/1619  
 incrementPendingOutboundBytes(entry.pendingSize, false);  
}

* 把数据串在一个Entry链中。

### FLUSH

- 13) CTX#invokeFlush

private void invokeFlush() {  
 if (invokeHandler()) {  
 invokeFlush0();  
 } else {  
 flush();  
 }  
}

- 14) CTX#invokeFlush0

private void invokeFlush0() {  
 try {  
 ((ChannelOutboundHandler) handler()).flush(this);  
 } catch (Throwable t) {  
 notifyHandlerException(t);  
 }  
}

- 15) ChannelOutboundHandlerAdatper#flush

public void flush(ChannelHandlerContext ctx) throws Exception {  
 ctx.flush();  
}

- 16) CTX#flush（从后往前flush，直至head)

public ChannelHandlerContext flush() {  
 final AbstractChannelHandlerContext next = findContextOutbound();  
 EventExecutor executor = next.executor();  
 if (executor.inEventLoop()) {  
 next.invokeFlush();  
 } else {  
 Runnable task = next.invokeFlushTask;  
 if (task == null) {  
 next.invokeFlushTask = task = new Runnable() {  
 @Override  
 public void run() {  
 next.invokeFlush();  
 }  
 };  
 }  
 safeExecute(executor, task, channel().voidPromise(), null);  
 }  
  
 return this;  
}

- 17)（head)DefaultChannelPipeline#flush

public void flush(ChannelHandlerContext ctx) throws Exception {  
 unsafe.flush();  
}

- 18) （unsafe)AbstractChannel#flush

public final void flush() {  
 assertEventLoop();  
  
 ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
 if (outboundBuffer == null) {  
 return;  
 }  
  
 outboundBuffer.addFlush();  
 flush0();  
}

- 19.1) ChannelOutboundBuffer#addFlush

/\*\*  
 \* Add a flush to this {@link ChannelOutboundBuffer}. This means all previous added messages are marked as flushed  
 \* and so you will be able to handle them.  
 \*/  
public void addFlush() {  
 // There is no need to process all entries if there was already a flush before and no new messages  
 // where added in the meantime.  
 //  
 // See https://github.com/netty/netty/issues/2577  
 Entry entry = unflushedEntry;  
 if (entry != null) {  
 if (flushedEntry == null) {  
 // there is no flushedEntry yet, so start with the entry  
 flushedEntry = entry;  
 }  
 do {  
 flushed ++;  
 if (!entry.promise.setUncancellable()) {  
 // Was cancelled so make sure we free up memory and notify about the freed bytes  
 int pending = entry.cancel();  
 decrementPendingOutboundBytes(pending, false, true);  
 }  
 entry = entry.next;  
 } while (entry != null);  
  
 // All flushed so reset unflushedEntry  
 unflushedEntry = null;  
 }  
}

- 19.2) AbstractChannel#flush0

* protected final void flush0() {  
  // Flush immediately only when there's no pending flush.  
  // If there's a pending flush operation, event loop will call forceFlush() later,  
  // and thus there's no need to call it now.  
  if (!isFlushPending()) {  
   super.flush0();  
  }  
  }

protected void flush0() {  
 if (inFlush0) {  
 // Avoid re-entrance  
 return;  
 }  
  
 final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;  
 if (outboundBuffer == null || outboundBuffer.isEmpty()) {  
 return;  
 }  
  
 inFlush0 = true;  
  
 // Mark all pending write requests as failure if the channel is inactive.  
 if (!isActive()) {  
 try {  
 if (isOpen()) {  
 outboundBuffer.failFlushed(FLUSH0\_NOT\_YET\_CONNECTED\_EXCEPTION, true);  
 } else {  
 // Do not trigger channelWritabilityChanged because the channel is closed already.  
 outboundBuffer.failFlushed(FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 }  
 } finally {  
 inFlush0 = false;  
 }  
 return;  
 }  
  
 try {  
 doWrite(outboundBuffer);  
 } catch (Throwable t) {  
 if (t instanceof IOException && config().isAutoClose()) {  
 /\*\*  
 \* Just call {@link #close(ChannelPromise, Throwable, boolean)} here which will take care of  
 \* failing all flushed messages and also ensure the actual close of the underlying transport  
 \* will happen before the promises are notified.  
 \*  
 \* This is needed as otherwise {@link #isActive()} , {@link #isOpen()} and {@link #isWritable()}  
 \* may still return {@code true} even if the channel should be closed as result of the exception.  
 \*/  
 close(voidPromise(), t, FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 } else {  
 try {  
 shutdownOutput(voidPromise(), t);  
 } catch (Throwable t2) {  
 close(voidPromise(), t2, FLUSH0\_CLOSED\_CHANNEL\_EXCEPTION, false);  
 }  
 }  
 } finally {  
 inFlush0 = false;  
 }  
}

- 19.2.1) NioSocketChannel#doWrite  
- 代码同EventLoop 1.2.1) NioSocketChannel#doWrite

* 底层是调用的SocketChannel的write方法。

# ByteBuf

# IdleStateHandler

* channelActive()方法调用Initialize()方法,根据配置的readerIdleTime，WriteIdleTIme等超时事件参数往任务队列taskQueue中添加定时任务task ；
* 定时任务添加到对应线程EventLoopExecutor对应的任务队列taskQueue中，在对应线程的run()方法中循环执行
* 用当前时间减去最后一次channelRead方法调用的时间判断是否空闲超时；
* 如果空闲超时则创建空闲超时事件并传递到channelPipeline中；
* IdleStateHandler心跳检测主要是通过向线程任务队列中添加定时任务，判断channelRead()方法或write()方法是否调用空闲超时，如果超时则触发超时事件执行自定义userEventTrigger()方法；
* Netty通过IdleStateHandler实现最常见的心跳机制不是一种双向心跳的PING-PONG模式，而是客户端发送心跳数据包，服务端接收心跳但不回复，因为如果服务端同时有上千个连接，心跳的回复需要消耗大量网络资源；如果服务端一段时间内内有收到客户端的心跳数据包则认为客户端已经下线，将通道关闭避免资源的浪费；在这种心跳模式下服务端可以感知客户端的存活情况，无论是宕机的正常下线还是网络问题的非正常下线，服务端都能感知到，而客户端不能感知到服务端的非正常下线；
* 要想实现客户端感知服务端的存活情况，需要进行双向的心跳；Netty中的channelInactive()方法是通过Socket连接关闭时挥手数据包触发的，因此可以通过channelInactive()方法感知正常的下线情况，但是因为网络异常等非正常下线则无法感知；