



Netty核心技术及源码剖析

尚硅谷研究院





--Netty介绍和应用场景



● 本课程学习要求

- 1) 本课程不适用于 0 基础的学员
- 2) 要求已经掌握了 Java 编程，主要技术构成：Java OOP 编程、Java 多线程编程、Java IO 编程、Java 网络编程、**常用的**Java 设计模式(比如 观察者模式，命令模式，职责链模式)、**常用的**数据结构(比如 链表)
- 3) 本课程的 <<Netty 核心源码剖析章节>> 要求学员**最好有项目开发和阅读源码的经历**。



• Netty的介绍

- 1) Netty 是由 JBOSS 提供的一个 Java 开源框架，现为 Github 上的独立项目。
- 2) Netty 是一个异步的、基于事件驱动的网络应用框架，用以快速开发高性能、高可靠的网络 IO 程序。
- 3) Netty 主要针对在 TCP 协议下，面向 Clients 端的高并发应用，或者 Peer-to-Peer 场景下的大量数据持续传输的应用。
- 4) Netty 本质是一个 NIO 框架，适用于服务器通讯相关的多种应用场景
- 5) 要透彻理解 Netty，需要先学习 NIO，这样我们才能阅读 Netty 的源码。

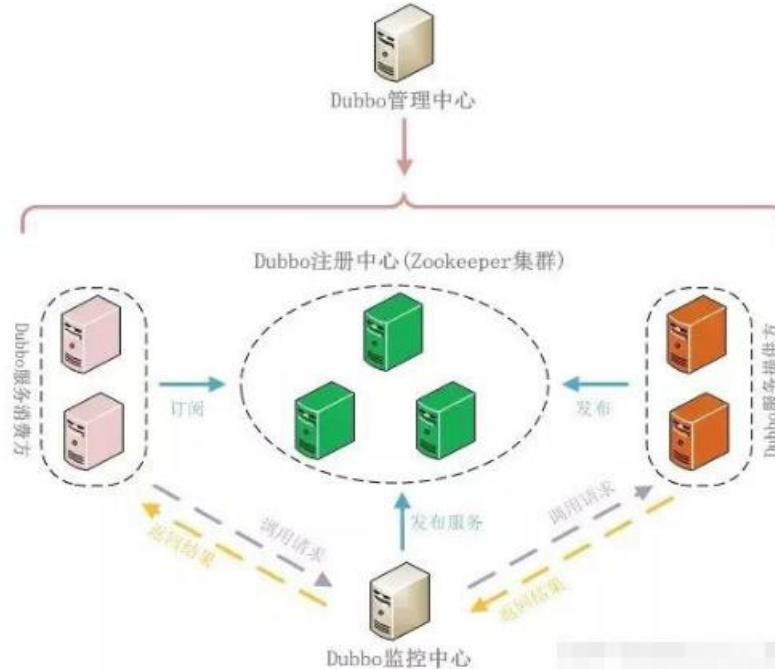


Netty project

- Netty的应用场景

互联网行业

- 1) 互联网行业：在分布式系统中，各个节点之间需要远程服务调用，高性能的 RPC 框架必不可少，Netty 作为异步高性能的通信框架，往往作为基础通信组件被这些 RPC 框架使用。
- 2) 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





• Netty的应用场景

游戏行业

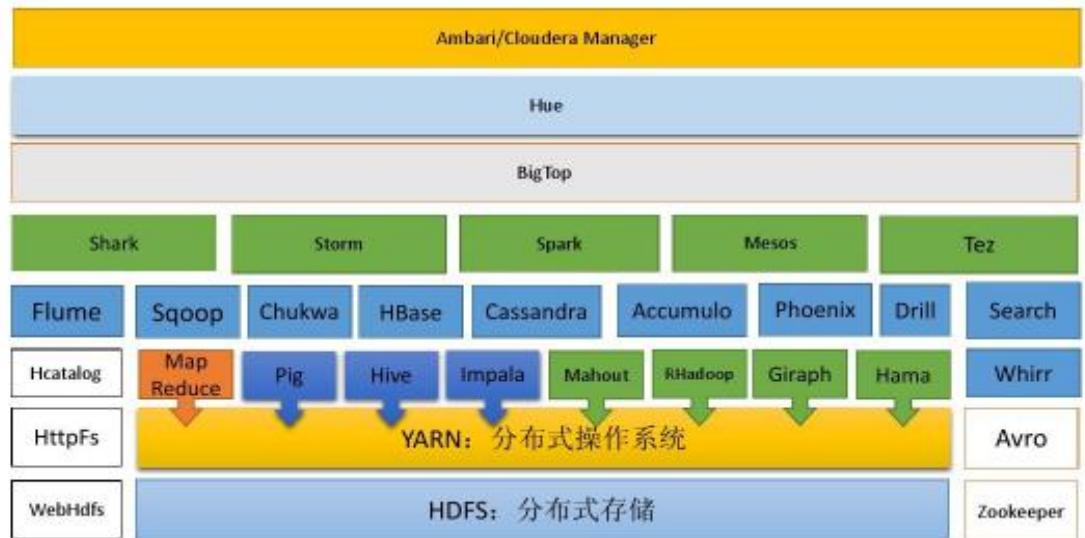
- 1) 无论是手游服务端还是大型的网络游戏，Java 语言得到了越来越广泛的应用
- 2) Netty 作为高性能的基础通信组件，提供了 TCP/UDP 和 HTTP 协议栈，方便定制和开发私有协议栈，账号登录服务器
- 3) 地图服务器之间可以方便的通过 Netty 进行高性能的通信



- Netty的应用场景

大数据领域

- 1) 经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨界点通信
- 2) 它的 Netty Service 基于 Netty 框架二次封装实现。





- Netty的应用场景

其它开源项目使用到Netty

网址: <https://netty.io/wiki/related-projects.html>

Related projects

Did you know this page is automatically generated from a Github Wiki page?

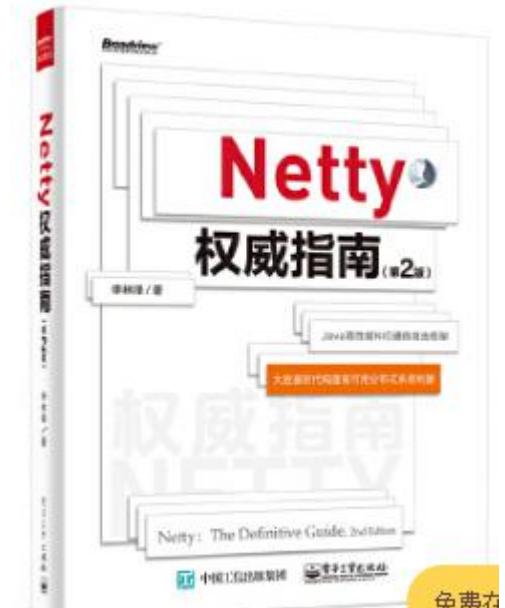
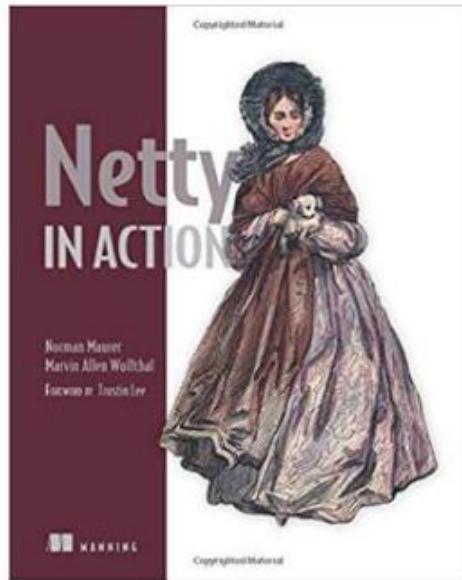
Make sure the list contains only open source projects documented in English.
months.)

- Akka is a Scala-based platform that provides simpler scalability, fault-tolerance, and concurrency management.
- Apache BookKeeper is a scalable, fault-tolerant, and low-latency log storage system.
- Apache Cassandra is a column oriented distributed database.
- Apache Flink is a distributed, stateful stream processing framework.
- Apache James Server is a modular e-mail server platform that integrates with various protocols.
- Apache Pulsar is an open-source distributed pub-sub messaging system.
- Apache Spark is a fast and general purpose cluster compute framework.
- Apache Tair is a distributed fault tolerance, low latency, and high throughput key-value store.



- Netty的学习参考资料

书籍





--Java BIO编程



- I/O模型

I/O 模型基本说明

- 1) I/O 模型简单的理解：就是用什么样的通道进行数据的发送和接收，很大程度上决定了程序通信的性能
- 2) Java 共支持3种网络编程模型/I/O模式： BIO、NIO、AIO
- 3) Java BIO： 同步并阻塞(传统阻塞型)，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销 【简单示意图】
- 4) Java NIO： 同步非阻塞，服务器实现模式为一个线程处理多个请求(连接)，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求就进行处理 【简单示意图】
- 5) Java AIO(NIO.2)： 异步非阻塞，AIO 引入异步通道的概念，采用了 Proactor 模式，简化了程序编写，有效的请求才启动线程，它的特点是先由操作系统完成后才通知服务端程



• I/O模型

BIO、NIO、AIO适用场景分析

- 1) BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，但程序简单易理解。
- 2) NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，弹幕系统，服务器间通讯等。编程比较复杂，JDK1.4开始支持。
- 3) AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。



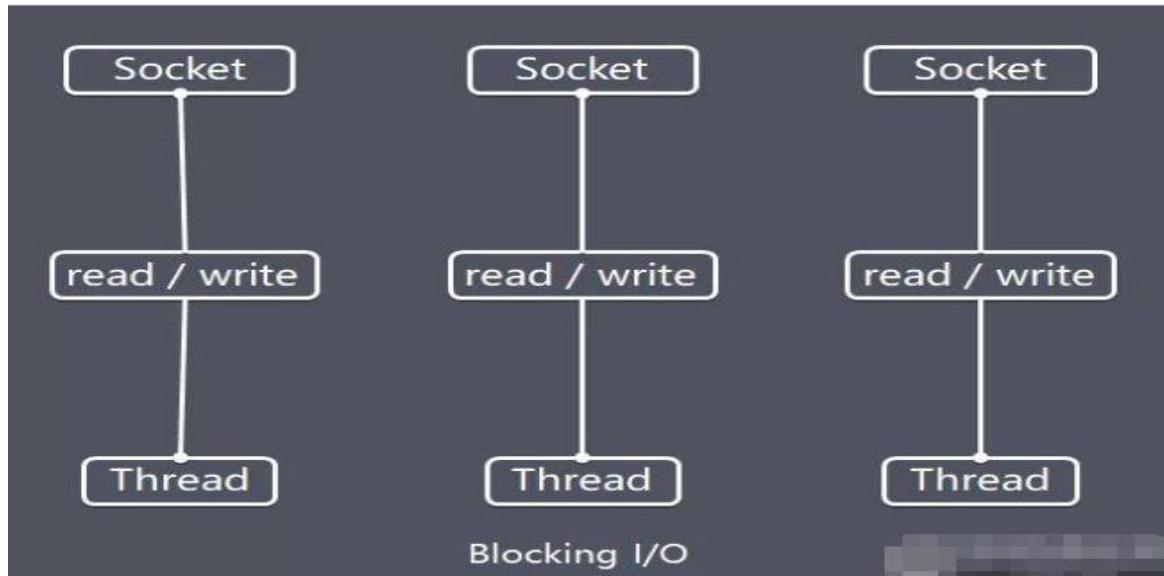
• Java BIO 基本介绍

- 1) Java BIO 就是传统的**java io** 编程，其相关的类和接口在 `java.io`
- 2) BIO(**blocking I/O**)： 同步阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，可以通过**线程池机制改善**(实现多个客户连接服务器)。【[后有应用实例](#)】
- 3) BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4以前的唯一选择，程序简单易理解



• Java BIO 工作机制

工作原理图



BIO编程简单流程

- 1) 服务器端启动一个`ServerSocket`
- 2) 客户端启动`Socket`对服务器进行通信，默认情况下服务器端需要对每个客户建立一个线程与之通讯
- 3) 客户端发出请求后，先咨询服务器是否有线程响应，如果没有则会等待，或者被拒绝
- 4) 如果有响应，客户端线程会等待请求结束后，在继续执行



• Java BIO 应用实例

实例说明：

- 1) 使用BIO模型编写一个服务器端，监听6666端口，当有客户端连接时，就启动一个线程与之通讯。
- 2) 要求使用线程池机制改善，可以连接多个客户端.
- 3) 服务器端可以接收客户端发送的数据(telnet 方式即可)。



BIOServer.zip

```
tomcat服务器启动...
连接到一个客户端!
hello
ok
连接到一个客户端!
hello2
关闭和client的连接..
关闭和client的连接..
```



• Java BIO 问题分析

- 1) 每个请求都需要创建独立的线程，与对应的客户端进行数据 **Read**，业务处理，数据 **Write**。
- 2) 当并发数较大时，需要**创建大量线程来处理连接**，系统资源占用较大。
- 3) 连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在 **Read** 操作上，造成线程资源浪费





--Java NIO编程



• Java NIO 基本介绍

- 1) Java NIO 全称 java non-blocking IO, 是指 JDK 提供的新 API。从 JDK1.4 开始, Java 提供了一系列改进的输入/输出的新特性, 被统称为 NIO(即 New IO), 是**同步非阻塞**的
- 2) NIO 相关类都被放在 `java.nio` 包及子包下, 并且对原 `java.io` 包中的很多类进行改写。【基本案例】



NioBasic.zip

- 3) NIO 有三大核心部分: **Channel(通道)**, **Buffer(缓冲区)**, **Selector(选择器)**

- 4) NIO 是面向**缓冲区**, 或者面向**块** 编程的。数据读取到一个它稍后处理的缓冲区, 需要时可在缓冲区中前后移动, 这就增加了处理过程中的灵活性, 使用它可以提供**非阻塞式**的高伸缩性网络

- ▷ `java.nio`
- ▷ `java.nio.channels`
- ▷ `java.nio.channels.spi`
- ▷ `java.nio.charset`
- ▷ `java.nio.charset.spi`
- ▷ `java.nio.file`
- ▷ `java.nio.file.attribute`
- ▷ `java.nio.file.spi`



• Java NIO 基本介绍

- 5) Java NIO的非阻塞模式，使一个线程从某通道发送请求或者读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，而**不是保持线程阻塞**，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此，一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。【[后面有案例说明](#)】
- 6) 通俗理解：NIO是可以做到用一个线程来处理多个操作的。假设有10000个请求过来，根据实际情况，可以分配50或者100个线程来处理。不像之前的阻塞IO那样，非得分配10000个。
- 7) HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。



• NIO 和 BIO 的比较

- 1) BIO 以流的方式处理数据,而 NIO 以块的方式处理数据,块 I/O 的效率比流 I/O 高很多
- 2) BIO 是阻塞的, NIO 则是非阻塞的
- 3) BIO 基于字节流和字符流进行操作, 而 NIO 基于 Channel(通道)和 Buffer(缓冲区)进行操作, 数据总是从通道读取到缓冲区中, 或者从缓冲区写入到通道中。
Selector(选择器)用于监听多个通道的事件 (比如: 连接请求, 数据到达等), 因此使用单个线程就可以监听多个客户端通道



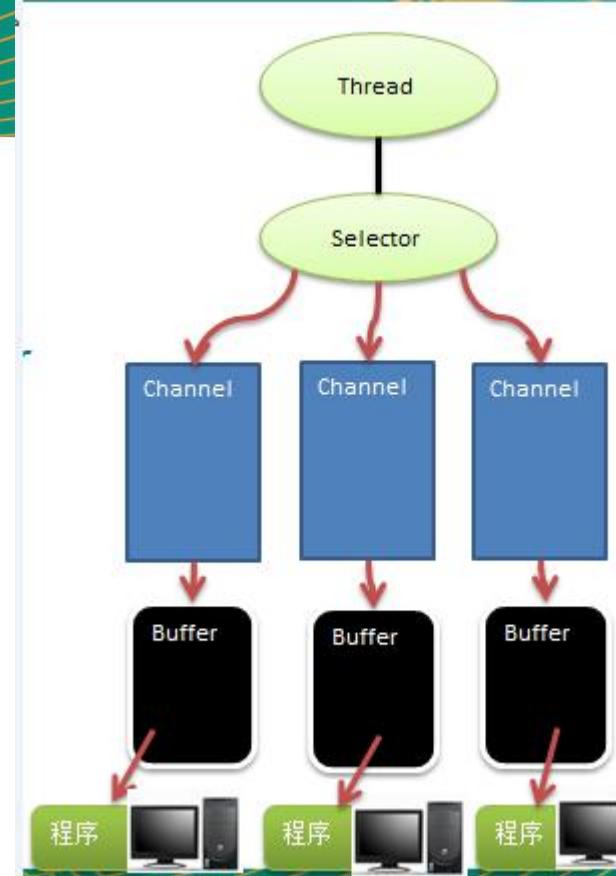
• NIO 三大核心原理示意图

一张图描述NIO 的 Selector 、
Channel 和 Buffer 的关系

Selector 、 Channel 和 Buffer 的关系图(简单版)

关系图的说明:

- 1) 每个channel 都会对应一个Buffer
- 2) Selector 对应一个线程， 一个线程对应多个channel(连接)
- 3) 该图反应了有三个channel 注册到 该selector //程序
- 4) 程序切换到哪个channel 是有事件决定的, Event 就是一个重要的概念
- 5) Selector 会根据不同的事件，在各个通道上切换
- 6) Buffer 就是一个内存块， 底层是有一个数组
- 7) 数据的读取写入是通过Buffer, 这个和BIO , BIO 中要么是输入流， 或者是输出流, 不能双向， 但是NIO的Buffer 是可以读也可以写, 需要 flip 方法切换
- 8) channel 是双向的, 可以返回底层操作系统的情况, 比如Linux , 底层的操作系统通道就是双向的.



关系图.



图100.zi

- 缓冲区(Buffer)

基本介绍

缓冲区（Buffer）：缓冲区本质上是一个可以读写数据的内存块，可以理解成是一个**容器对象(含数组)**，该对象提供了一组方法，可以更轻松地使用内存块，，缓冲区对象内置了一些机制，能够跟踪和记录缓冲区的状态变化情况。Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer，如图：【后面举例说明】

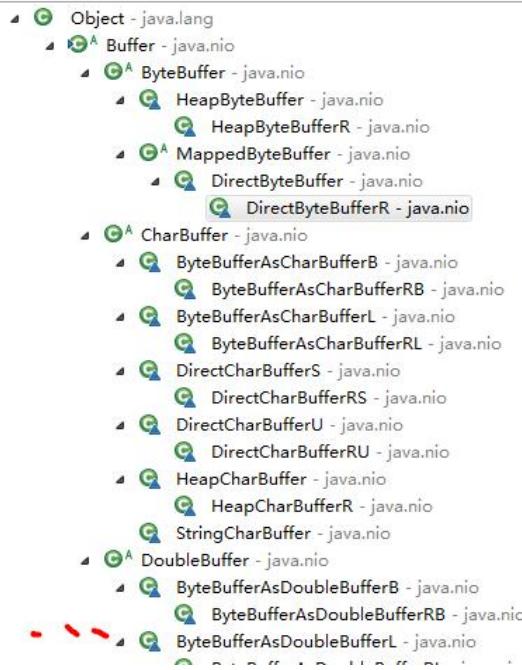




- 缓冲区(Buffer)

Buffer 类及其子类

1) 在 NIO 中, Buffer 是一个顶层父类, 它是一个抽象类, 类的层级关系图:



常用Buffer子类一览

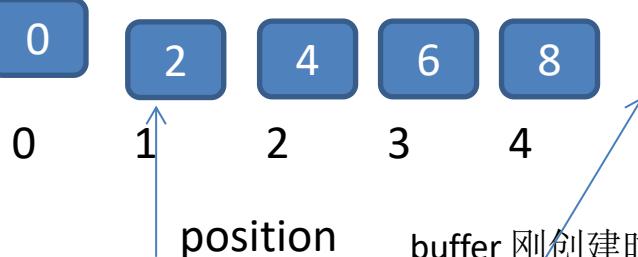
- 1) ByteBuffer, 存储字节数据到缓冲区
- 2) ShortBuffer, 存储字符串数据到缓冲区
- 3) CharBuffer, 存储字符数据到缓冲区
- 4) IntBuffer, 存储整数数据到缓冲区
- 5) LongBuffer, 存储长整型数据到缓冲区
- 6) DoubleBuffer, 存储小数到缓冲区
- 7) FloatBuffer, 存储小数到缓冲区

- 缓冲区(Buffer)

Buffer 类及其子类

- 2) Buffer类定义了所有的缓冲区都具有的四个属性来提供关于其所包含的数据元素的信息:

```
// Invariants: mark <= position <= limit <= capacity
private int mark = -1;
private int position = 0;
private int limit;
private int capacity;
```



buffer 刚创建时
 $\text{capacity} = 5 // \text{不变}$
 $\text{limit} = 5 // \text{不能对缓冲区超过极限的位置 limit 进行读写操作}$

属性	描述
Capacity	容量, 即可以容纳的最大数据量; 在缓冲区创建时被设定并且不能改变
Limit	表示缓冲区的当前终点, 不能对缓冲区超过极限的位置进行读写操作。且极限是可以修改的
Position	位置, 下一个要被读或写的元素的索引 每次读写缓冲区数据时都会改变改值, 为下次读写作准备
Mark	标记



● 缓冲区(Buffer)

Buffer 类及其子类

3) Buffer类相关方法一览

```
public abstract class Buffer {  
    //JDK1.4时，引入的api  
    public final int capacity() //返回此缓冲区的容量  
    public final int position() //返回此缓冲区的位置  
    public final Buffer position (int newPosition) //设置此缓冲区的位置  
    public final int limit() //返回此缓冲区的限制  
    public final Buffer limit (int newLimit) //设置此缓冲区的限制  
    public final Buffer mark() //在此缓冲区的位置设置标记  
    public final Buffer reset() //将此缓冲区的位置重置为以前标记的位置  
    public final Buffer clear() //清除此缓冲区，即将各个标记恢复到初始状态，但是数据并不被释放  
    public final Buffer flip() //反转此缓冲区  
    public final Buffer rewind() //重绕此缓冲区  
    public final int remaining() //返回当前位置与限制之间的元素数  
    public final boolean hasRemaining() //告知在当前位置和限制之间是否有元素  
    public abstract boolean isReadOnly() //告知此缓冲区是否为只读缓冲区  
  
    //JDK1.6时引入的api  
    public abstract boolean hasArray(); //告知此缓冲区是否具有可访问的底层实现数组  
    public abstract Object array(); //返回此缓冲区的底层实现数组  
    public abstract int arrayOffset(); //返回此缓冲区的底层实现数组中第一个缓冲区元素的偏移量  
    public abstract boolean isDirect(); //告知此缓冲区是否为直接缓冲区  
}
```

- 缓冲区(Buffer)

ByteBuffer

从前面可以看出对于 Java 中的基本数据类型(boolean除外), 都有一个 Buffer 类型与之相对应, **最常用**的自然是 ByteBuffer 类 (二进制数据), 该类的主要方法如下:

```
public abstract class ByteBuffer {  
    //缓冲区创建相关api  
    public static ByteBuffer allocateDirect(int capacity)//创建直接缓冲区  
    public static ByteBuffer allocate(int capacity)//设置缓冲区的初始容量  
    public static ByteBuffer wrap(byte[] array)//把一个数组放到缓冲区中使用  
    //构造初始化位置offset和上界length的缓冲区  
    public static ByteBuffer wrap(byte[] array,int offset, int length)  
    //缓存区存取相关API  
    public abstract byte get()//从当前位置position上get, get之后, position会自动+1  
    public abstract byte get (int index);//从绝对位置get  
    public abstract ByteBuffer put (byte b);//从当前位置上添加, put之后, position会自动+1  
    public abstract ByteBuffer put (int index, byte b); //从绝对位置上put  
}
```



- 通道(Channel)

基本介绍

1) NIO的通道类似于流，但有些区别如下：

- 通道可以同时进行读写，而流只能读或者只能写
- 通道可以实现异步读写数据
- 通道可以从缓冲读数据，也可以写数据到缓冲：

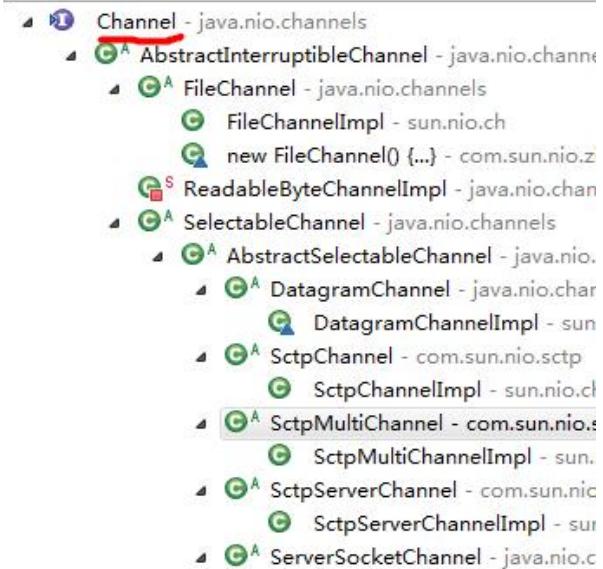




- 通道(Channel)

基本介绍

- 2) BIO 中的 stream 是单向的，例如 FileInputStream 对象只能进行读取数据的操作，而 NIO 中的通道 (Channel)是双向的，可以读操作，也可以写操作。
- 3) Channel在NIO中是一个接口
public interface Channel extends Closeable{}
- 4) 常用的 Channel 类有：FileChannel、 DatagramChannel、ServerSocketChannel 和 SocketChannel。【ServerSocketChanne 类似 ServerSocket , SocketChannel 类似 Socket】
- 5) FileChannel 用于文件的数据读写， DatagramChannel 用于 UDP 的数据读写， ServerSocketChannel 和 SocketChannel 用于 TCP 的数据读写。





- 通道(Channel)

FileChannel 类

FileChannel主要用来对本地文件进行 IO 操作，常见的方法有

- 1) public int read(ByteBuffer dst) , 从通道读取数据并放到缓冲区中
- 2) public int write(ByteBuffer src) , 把缓冲区的数据写到通道中
- 3) public long transferFrom(ReadableByteChannel src, long position, long count), 从目标通道中复制数据到当前通道
- 4) public long transferTo(long position, long count, WritableByteChannel target), 把数据从当前通道复制给目标通道



- 通道(Channel)

应用实例1-本地文件写数据

实例要求：

- 1) 使用前面学习后的ByteBuffer(缓冲) 和 FileChannel(通道)， 将 "hello,尚硅谷" 写入到file01.txt 中
- 2) 文件不存在就创建
- 3) 代码演示



NIOFileOper01.zip



- 通道(Channel)

应用实例2-本地文件读数据

实例要求:

- 1) 使用前面学习后的ByteBuffer(缓冲) 和 FileChannel(通道), 将 file01.txt 中的数据读入到程序, 并显示在控制台屏幕
- 2) 假定文件已经存在
- 3) 代码演示



NIOFileOper02.zip



- 通道(Channel)

应用实例3-使用一个Buffer完成文件读取

实例要求:

- 1) 使用 FileChannel(通道) 和 方法 read , write , 完成文件的拷贝
- 2) 拷贝一个文本文件 **1.txt** , 放在项目下即可
- 3) 代码演示



- 通道(Channel)

应用实例4-拷贝文件transferFrom 方法

实例要求:

- 1) 使用 FileChannel(通道) 和方法 transferFrom , 完成文件的拷贝
- 2) 拷贝一张图片
- 3) 代码演示



NIOFileCopy.zip



- 通道(Channel)

关于Buffer 和 Channel的注意事项和细节

- 1) ByteBuffer 支持类型化的put 和 get, put 放入的是什么数据类型, get就应该使用相应的数据类型来取出, 否则可能有 BufferUnderflowException 异常。【举例说明】
- 2) 可以将一个普通Buffer 转成只读Buffer 【举例说明】
- 3) NIO 还提供了 MappedByteBuffer, 可以让文件直接在内存（堆外的内存）中进行修改, 而如何同步到文件由NIO 来完成. 【举例说明】
- 4) 前面我们讲的读写操作, 都是通过一个Buffer 完成的, NIO 还支持 通过多个 Buffer (即 Buffer 数组) 完成读写操作, 即 Scattering 和 Gathering 【举例说明】



- **Selector(选择器)**

基本介绍

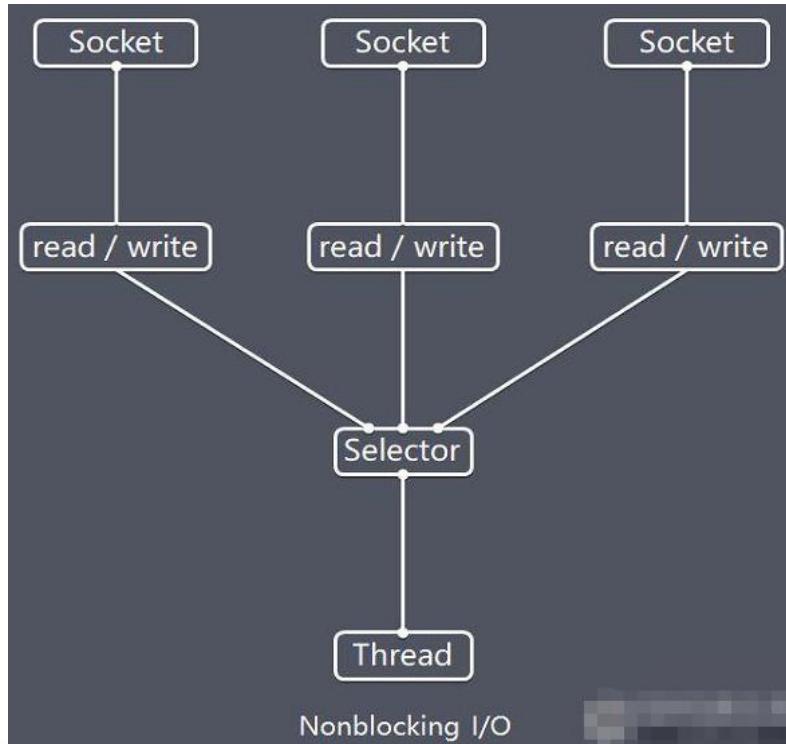
- 1) Java 的 NIO，用非阻塞的 IO 方式。可以用一个线程，处理多个的客户端连接，就会使用到**Selector(选择器)**
- 2) **Selector 能够检测多个注册的通道上是否有事件发生(注意:多个Channel以事件的方式可以注册到同一个Selector)**，如果有事件发生，便获取事件然后针对每个事件进行相应的处理。这样就可以只用一个单线程去管理多个通道，也就是管理多个连接和请求。【示意图】
- 3) 只有在 连接/通道 真正有读写事件发生时，才会进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程
- 4) 避免了多线程之间的上下文切换导致的开销



图片1.z

- Selector(选择器)

Selector示意图和特点说明



特点再说明：

- 1) Netty 的 IO 线程 NioEventLoop 聚合了 Selector(选择器, 也叫多路复用器), 可以同时并发处理成百上千个客户端连接。
- 2) 当线程从某客户端 Socket 通道进行读写数据时, 若没有数据可用时, 该线程可以进行其他任务。
- 3) 线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作, 所以单独的线程可以管理多个输入和输出通道。
- 4) 由于读写操作都是非阻塞的, 这就可以充分提升 IO 线程的运行效率, 避免由于频繁 I/O 阻塞导致的线程挂起。
- 5) 一个 I/O 线程可以并发处理 N 个客户端连接和读写操作, 这从根本上解决了传统同步阻塞 I/O 一连接一线程模型, 架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

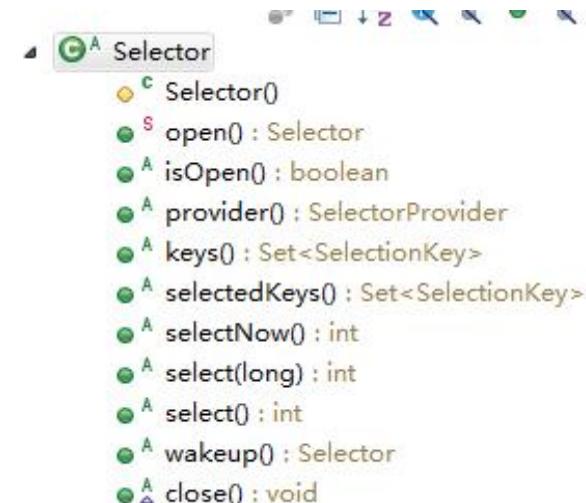


- Selector(选择器)

Selector类相关方法

Selector 类是一个抽象类, 常用方法和说明如下:

```
public abstract class Selector implements Closeable {  
    public static Selector open(); // 得到一个选择器对象  
    public int select(long timeout); // 监控所有注册的通道, 当其  
        中有 IO 操作可以进行时, 将  
        对应的 SelectionKey 加入到内部集合中并返回, 参数用来  
        设置超时时间  
    public Set<SelectionKey> selectedKeys(); // 从内部集合中得  
        到所有的 SelectionKey  
}
```





- **Selector(选择器)**

注意事项

- 1) NIO中的 ServerSocketChannel功能类似ServerSocket, SocketChannel功能类似Socket
- 2) selector 相关方法说明

selector.select()//阻塞

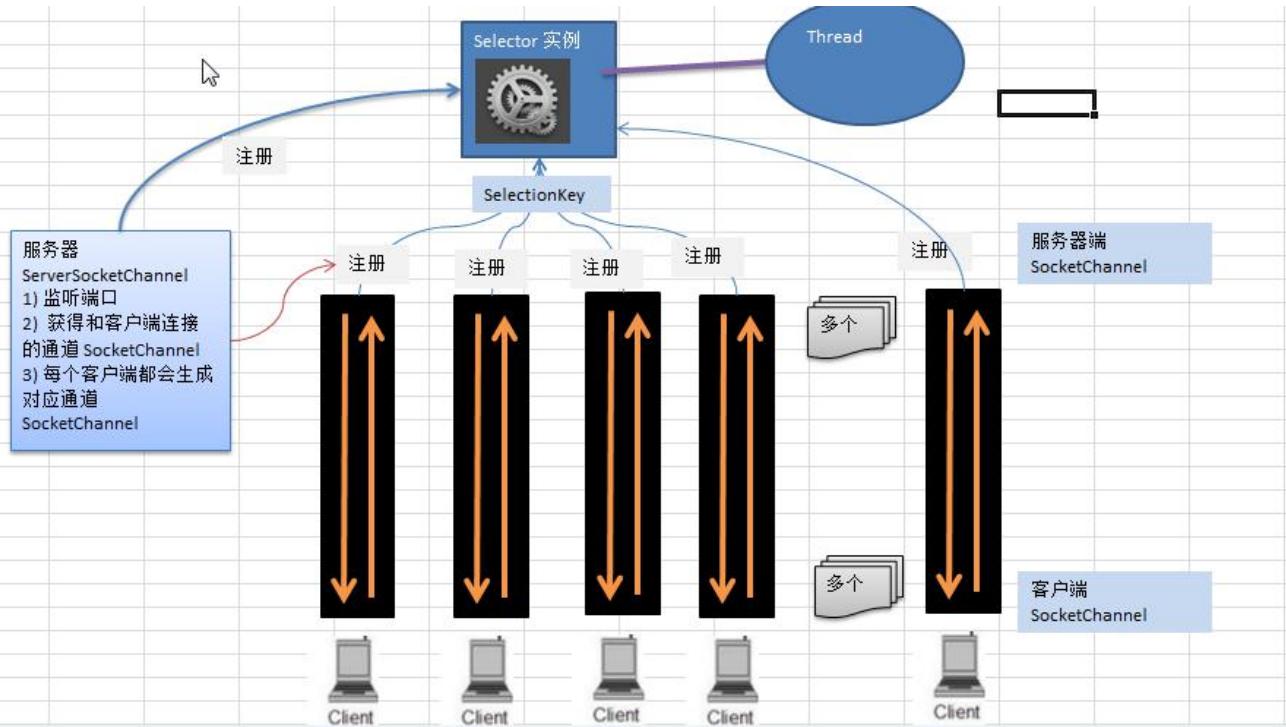
selector.select(1000); //阻塞1000毫秒，在1000毫秒后返回

selector.wakeup(); //唤醒selector

selector.selectNow(); //不阻塞，立马返还

• NIO 非阻塞 网络编程原理分析图

NIO 非阻塞 网络编程相关的(Selector、SelectionKey、ServerScoketChannel和SocketChannel) 关系梳理图



对右图的说明：

- 当客户端连接时，会通过 ServerSocketChannel 得到 SocketChannel
- Selector 进行监听 select 方法，返回有事件发生的通道的个数。
- 将 socketChannel 注册到 Selector 上，register(Selector sel, int ops)，一个 selector 上可以注册多个 SocketChannel
- 注册后返回一个 SelectionKey，会和该 Selector 关联(集合)
- 进一步得到各个 SelectionKey (有事件发生)
- 在通过 SelectionKey 反向获取 SocketChannel，方法 channel()
- 可以通过得到的 channel，完成业务处理
- 代码撑腰。。。



• NIO 非阻塞 网络编程快速入门

案例要求:

- 1) 编写一个 NIO 入门案例，实现服务器端和客户端之间的数据简单通讯（非阻塞）
- 2) 目的：理解NIO非阻塞网络编程机制
- 3) 看老师代码演示



NIOServer.zip



NIOClient.zip



2.zip



- SelectionKey

1) SelectionKey，表示 Selector 和网络通道的注册关系，共四种：

int OP_ACCEPT: 有新的网络连接可以 accept，值为 16

int OP_CONNECT: 代表连接已经建立，值为 8

int OP_READ: 代表读操作，值为 1

int OP_WRITE: 代表写操作，值为 4

源码中：

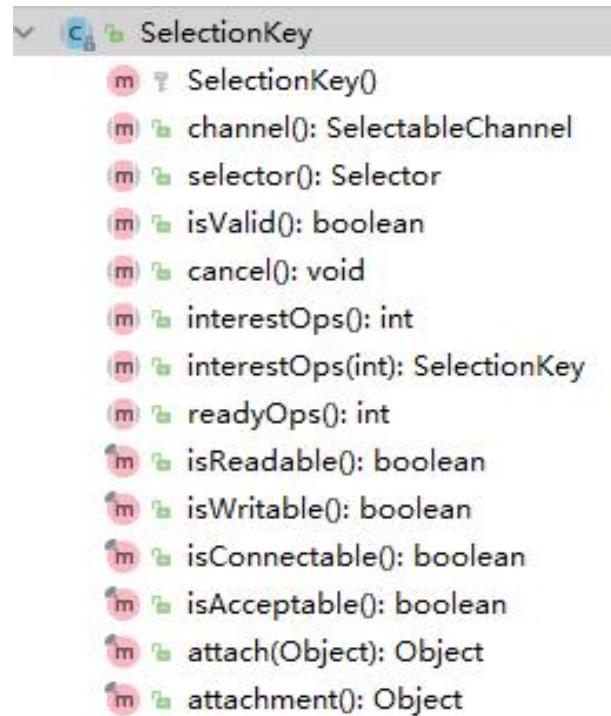
```
public static final int OP_READ = 1 << 0;  
public static final int OP_WRITE = 1 << 2;  
public static final int OP_CONNECT = 1 << 3;  
public static final int OP_ACCEPT = 1 << 4;
```



● SelectionKey

2) SelectionKey相关方法

```
public abstract class SelectionKey {  
    public abstract Selector selector(); //得到与之关联的  
    Selector 对象  
    public abstract SelectableChannel channel(); //得到与之关  
    联的通道  
    public final Object attachment(); //得到与之关联的共享数  
    据  
    public abstract SelectionKey interestOps(int ops); //设置或改  
    变监听事件  
    public final boolean isAcceptable(); //是否可以 accept  
    public final boolean isReadable(); //是否可以读  
    public final boolean isWritable(); //是否可以写  
}
```





● ServerSocketChannel

- 1) ServerSocketChannel 在服务器端监听新的客户端 **Socket** 连接
- 2) 相关方法如下

```
public abstract class ServerSocketChannel
```

```
    extends AbstractSelectableChannel
```

```
    implements NetworkChannel{
```

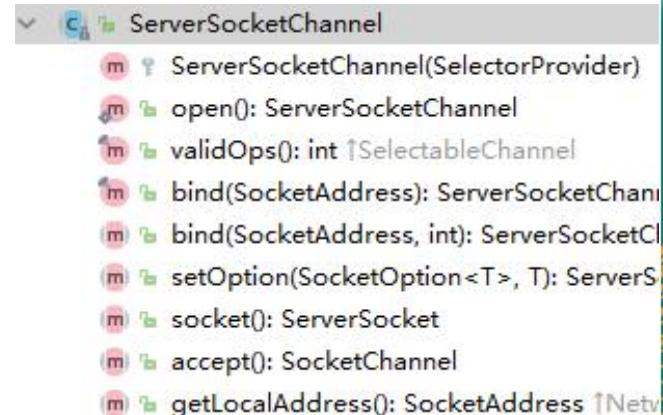
public static ServerSocketChannel open(), 得到一个 ServerSocketChannel 通道
public final ServerSocketChannel bind(SocketAddress local), 设置服务器端口号

public final SelectableChannel configureBlocking(boolean block), 设置阻塞或非阻塞模式, 取值 false 表示采用非阻塞模式

public SocketChannel accept(), 接受一个连接, 返回代表这个连接的通道对象

public final SelectionKey register(Selector sel, int ops), 注册一个选择器并设置监听事件

```
}
```

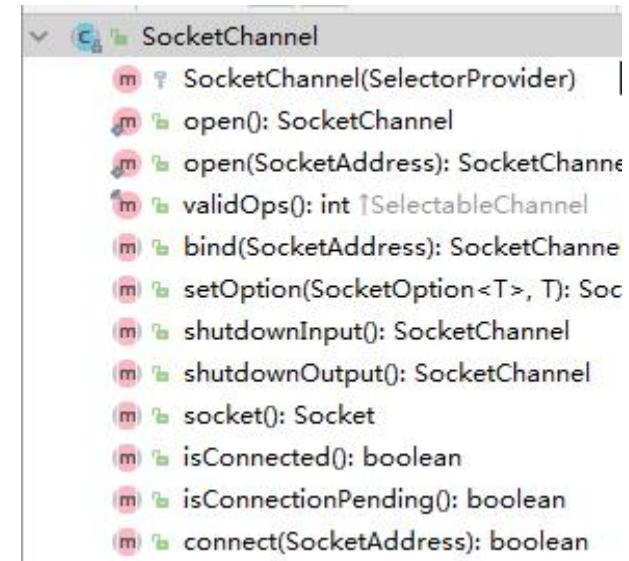




● SocketChannel

- 1) SocketChannel，网络 IO 通道，**具体负责进行读写操作**。NIO 把缓冲区的数据写入通道，或者把通道里的数据读到缓冲区。
- 2) 相关方法如下

```
public abstract class SocketChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel,
NetworkChannel{
    public static SocketChannel open(); //得到一个 SocketChannel 通道
    public final SelectableChannel configureBlocking(boolean block); //设置阻塞或非阻塞
模式，取值 false 表示采用非阻塞模式
    public boolean connect(SocketAddress remote); //连接服务器
    public boolean finishConnect(); //如果上面的方法连接失败，接下来就要通过该方法
完成连接操作
    public int write(ByteBuffer src); //往通道里写数据
    public int read(ByteBuffer dst); //从通道里读数据
    public final SelectionKey register(Selector sel, int ops, Object att); //注册一个选择器并
设置监听事件，最后一个参数可以设置共享数据
    public final void close(); //关闭通道
```





• NIO 网络编程应用实例-群聊系统

实例要求：

- 1) 编写一个 NIO 群聊系统，实现服务器端和客户端之间的数据简单通讯（非阻塞）
- 2) 实现多人群聊
- 3) 服务器端：可以监测用户上线，离线，并实现消息转发功能
- 4) 客户端：通过channel 可以无阻塞发送消息给其它所有用户，同时可以接受其它用户发送的消息(有服务器转发得到)
- 5) 目的：进一步理解NIO非阻塞网络编程机制

```
D:\program\jdk8.0\bin\java ...
服务器接收到消息 时间: [2019-10-22 10:28:11] -> 服务器 ok..
selectionkey=2
/127.0.0.1:50037 上线
selectionkey=3
/127.0.0.1:50046 上线
selectionkey=4
/127.0.0.1:50056 上线
服务器接收到消息 时间: [2019-10-22 10:29:07] -> 127.0.0.1:
我是50056

服务器进行消息转发 ...
取分蚕进1J相忘转反 ...
服务器接收到消息 时间: [2019-10-22 10:30:10] -> /127.0.0.1:50037
我是50046 说: 我是50056
服务器接收到消息 时间: [2019-10-22 10:30:19] -> /127.0.0.1:50046
我是50056 说: 我是50037
服务器接收到消息 时间: [2019-10-22 10:30:29] -> /127.0.0.1:50056
我是50037 说: 我是50046
```

```
D:\program\jdk8.0\bin\java ...
127.0.0.1:50046 is ok ~
127.0.0.1:50056 说: 我是50056
我是50046
127.0.0.1:50037 说: 我是50037
```

```
D:\program\jdk8.0\bin\java ...
127.0.0.1:50037 is ok ~
127.0.0.1:50056 说: 我是50056
127.0.0.1:50046 说: 我是50046
我是50037
```

Process finished with exit code 1

```
D:\program\jdk8.0\bin\java ...
127.0.0.1:50056
我是50056
127.0.0.1:50046
127.0.0.1:50037
```



• NIO 网络编程应用实例-群聊系统

实例要求:

6) 看老师代码演示

Server 端

服务器接收到消息 时间: [2019-09-29 12:02:35] -> 服务器 ok.....

127.0.0.1:56018 上线 ...

127.0.0.1:56029 上线 ...

服务器接收到消息 时间: [2019-09-29 12:02:55] -> 127.0.0.1:56029 说: jack
服务器进行消息转发 ...

服务器接收到消息 时间: [2019-09-29 12:03:07] -> 127.0.0.1:56018 说: tom
服务器进行消息转发 ...

服务器接收到消息 时间: [2019-09-29 12:03:14] -> 127.0.0.1:56029 说: 以前玩
服务器进行消息转发 ...

服务器接收到消息 时间: [2019-09-29 12:03:51] -> 127.0.0.1:56018 离线了 ...

服务器接收到消息 时间: [2019-09-29 12:03:56] -> 127.0.0.1:56029 离线了 ...

Client 端

127.0.0.1:56198 is ok ~

127.0.0.1:56186 说: 我是jack

我是tom

127.0.0.1:56207 说: 我是tom

127.0.0.1:56207 说: 我来自北京

我来自上海

127.0.0.1:56186 说: 我来自天津

bye



GroupChatClient.zip



4.zip



- NIO与零拷贝

零拷贝基本介绍

- 1) 零拷贝是网络编程的关键，很多性能优化都离不开。
- 2) 在 Java 程序中，常用的零拷贝有 `mmap`(内存映射) 和 `sendFile`。那么，他们在 OS 里，到底是怎么样的一个的设计？我们分析 `mmap` 和 `sendFile` 这两个零拷贝
- 3) 另外我们看下NIO 中如何使用零拷贝



- NIO与零拷贝

传统IO数据读写

1) Java 传统 IO 和 网络编程的一段代码

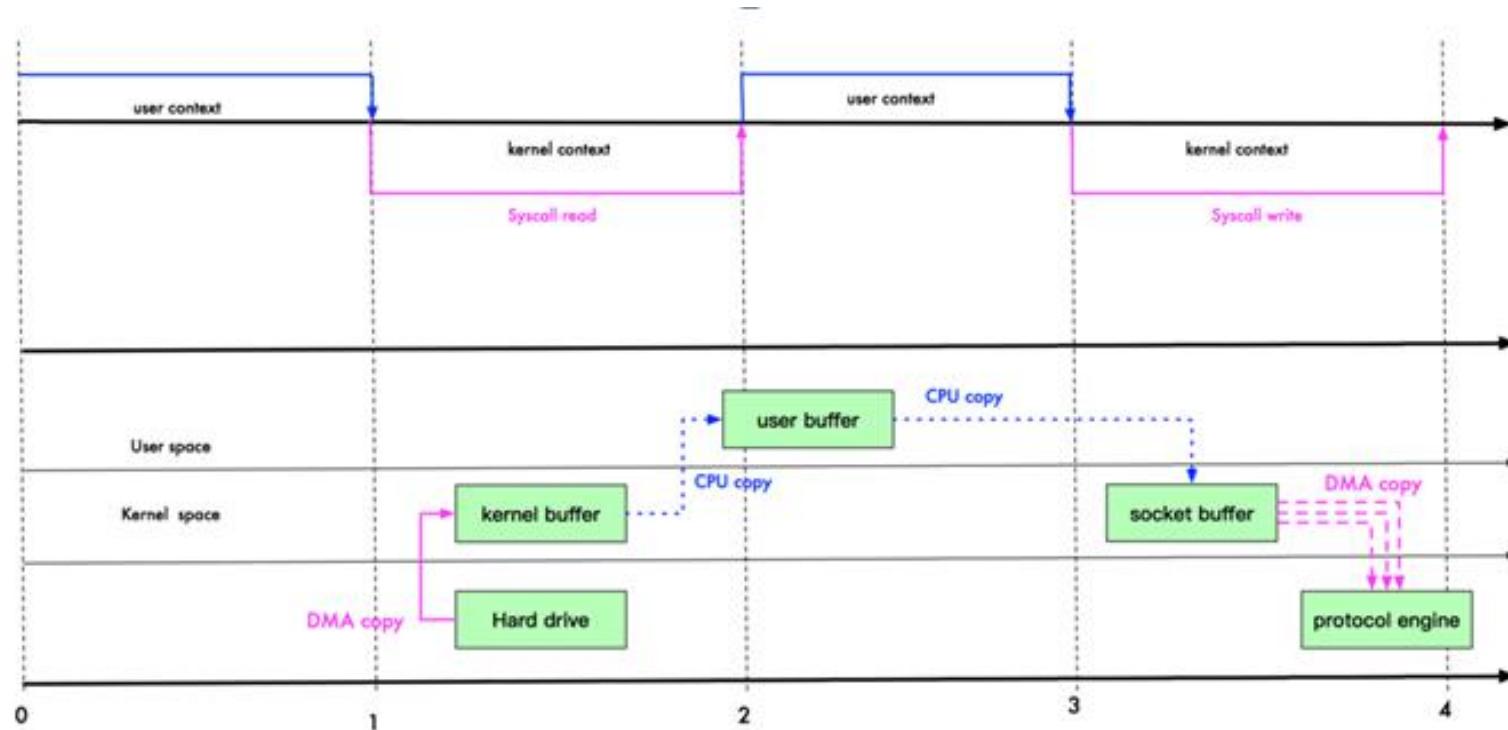
```
File file = new File("test.txt");
RandomAccessFile raf = new RandomAccessFile(file, "rw");

byte[] arr = new byte[(int) file.length()];
raf.read(arr);

Socket socket = new ServerSocket(8080).accept();
socket.getOutputStream().write(arr);
```

- NIO与零拷贝

传统IO

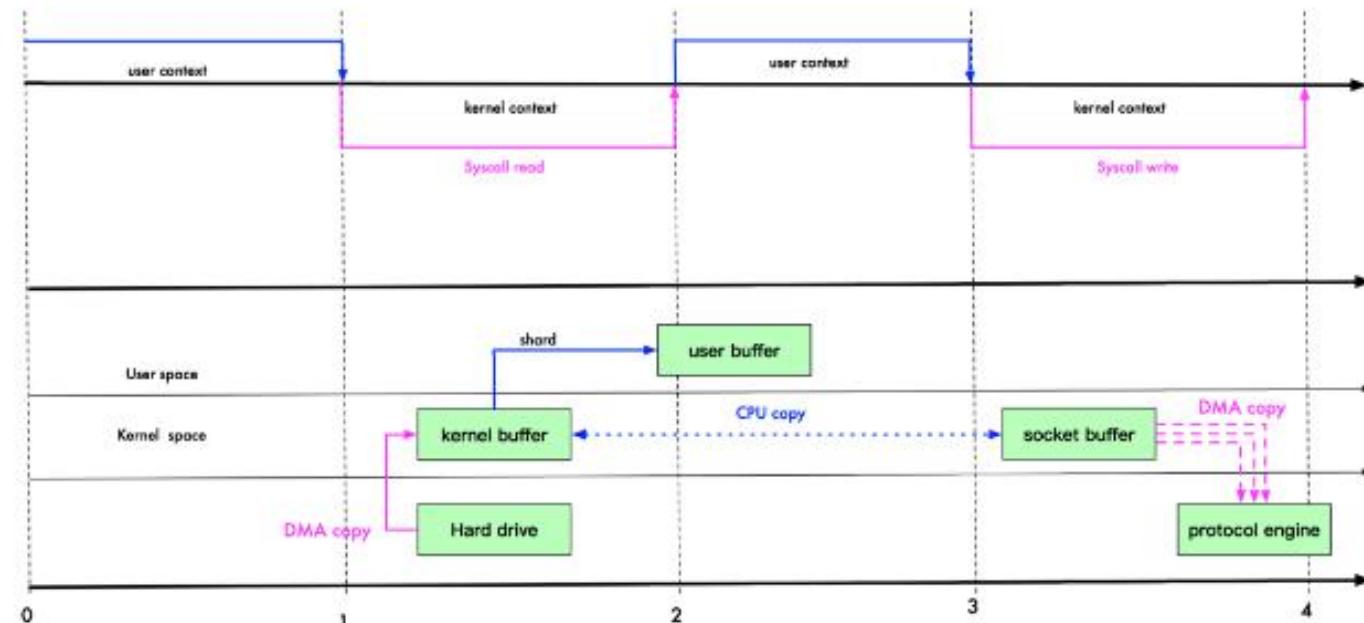


DMA: direct
memory access
直接内存拷贝(

- NIO与零拷贝
mmap 优化

1) mmap 通过内存映射，将文件映射到内核缓冲区，同时，用户空间可以共享内核空间的数据。这样，在进行网络传输时，就可以减少内核空间到用户控件的拷贝次数。如下图

2) mmap示意图

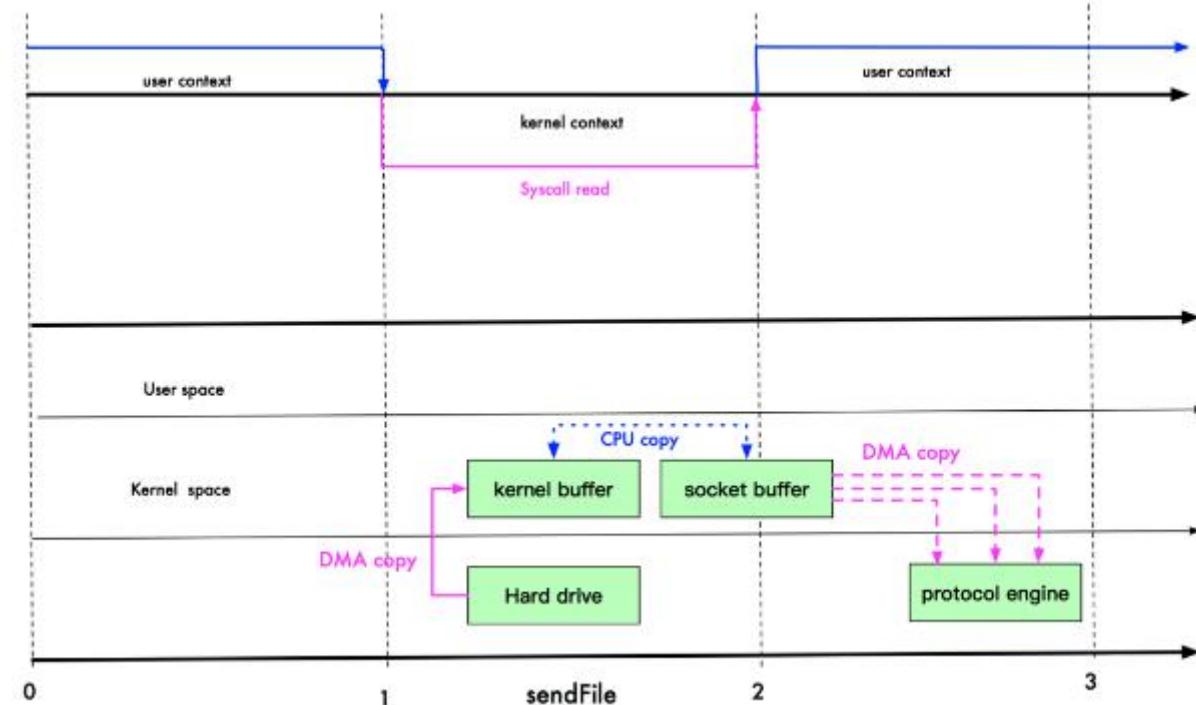


- NIO与零拷贝
sendFile 优化

1) Linux 2.1 版本 提供了 sendFile 函数，其基本原理如下：数据根本不 经过用户态，直接从内 核缓冲区进入到 Socket Buffer，同时，由于和用 户态完全无关，就减少 了一次上下文切换

2) 示意图和小结

提示：零拷贝从操作系统角 度，是没有cpu 拷贝

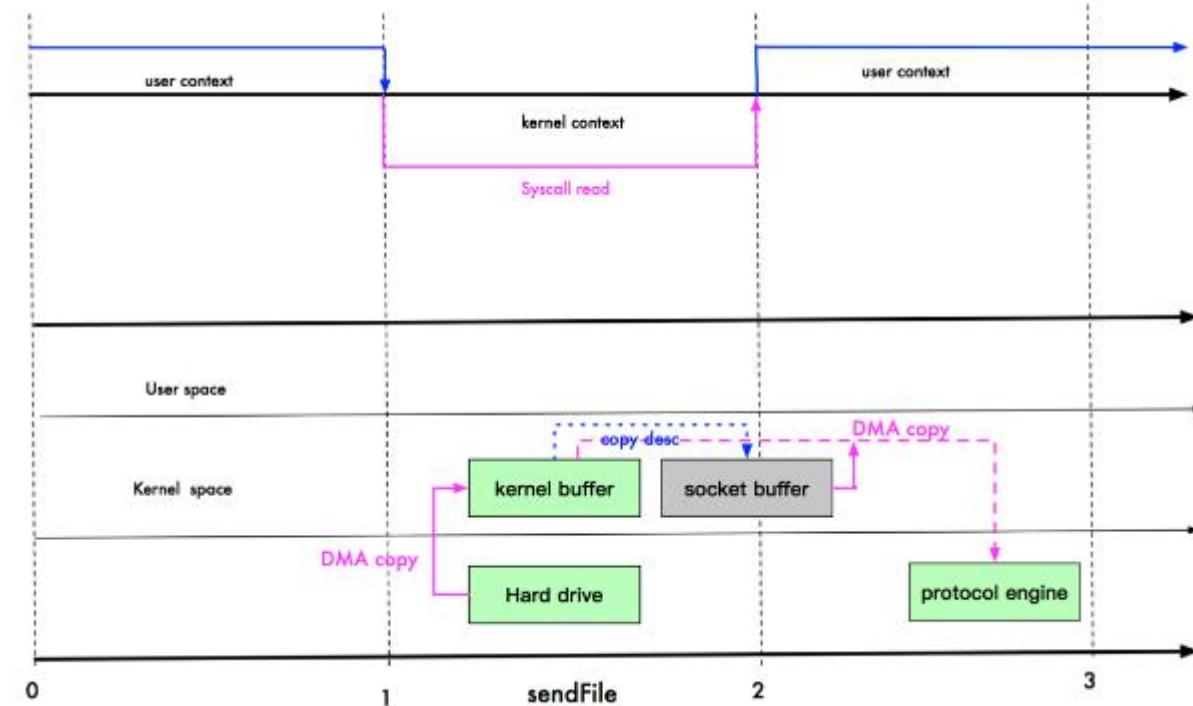


- NIO与零拷贝

sendFile 优化

3) Linux 在 2.4 版本中，做了一些修改，避免了从内核缓冲区拷贝到 Socket buffer 的操作，直接拷贝到协议栈，从而再一次减少了数据拷贝。具体如下图和小结：

这里其实有一次cpu 拷贝
kernel buffer -> socket buffer
但是，拷贝的信息很少，比如
length , offset , 消耗低，可以忽略





- NIO与零拷贝

零拷贝的再次理解

- 1) 我们说零拷贝，是从操作系统的角度来说的。因为内核缓冲区之间，没有数据是重复的（只有 **kernel buffer** 有一份数据）。
- 2) 零拷贝不仅仅带来更少的数据复制，还能带来其他的性能优势，例如更少的上下文切换，更少的 **CPU** 缓存伪共享以及无 **CPU** 校验和计算。



- NIO与零拷贝

mmap 和 sendFile 的区别

- 1) mmap 适合小数据量读写, sendFile 适合大文件传输。
- 2) mmap 需要 4 次上下文切换, 3 次数据拷贝; sendFile 需要 3 次上下文切换, 最少 2 次数据拷贝。
- 3) sendFile 可以利用 DMA 方式, 减少 CPU 拷贝, mmap 则不能 (必须从内核拷贝到 Socket 缓冲区)。



- NIO与零拷贝

NIO 零拷贝案例

案例要求：

- 1) 使用传统的IO 方法传递一个大文件
- 2) 使用NIO 零拷贝方式传递(transferTo)一个大文件
- 3) 看看两种传递方式耗时时间分别是多少



zerocopy.zip



• Java AIO 基本介绍

- 1) JDK 7 引入了 Asynchronous I/O，即 AIO。在进行 I/O 编程中，常用到两种模式：Reactor 和 Proactor。Java 的 NIO 就是 Reactor，当有事件触发时，服务器端得到通知，进行相应的处理
- 2) AIO 即 NIO2.0，叫做异步不阻塞的 IO。AIO 引入异步通道的概念，采用了 Proactor 模式，简化了程序编写，有效的请求才启动线程，它的特点是先由操作系统完成后才通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用
- 3) 目前 AIO 还没有广泛应用，Netty 也是基于NIO, 而不是AIO，因此我们就不详解 AIO了，有兴趣的同学可以参考 <<Java新一代网络编程模型AIO原理及Linux系统 AIO介绍>> <http://www.52im.net/thread-306-1-1.html>

• BIO、NIO、AIO 对比表

	BIO	NIO	AIO
IO 模型	同步阻塞	同步非阻塞（多路复用）	异步非阻塞
编程难度	简单	复杂	复杂
可靠性	差	好	好
吞吐量	低	高	高

举例说明

- 1) 同步阻塞：到理发店理发，就一直等理发师，直到轮到自己理发。
- 2) 同步非阻塞：到理发店理发，发现前面有其它人理发，给理发师说下，先干其他事情，一会过来看是否轮到自己。
- 3) 异步非阻塞：给理发师打电话，让理发师上门服务，自己干其它事情，理发师自己来家给你理发





--Netty 概述



- 原生NIO存在的问题

- 1) NIO 的类库和 API 繁杂，使用麻烦：需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等。
- 2) 需要具备其他的额外技能：要熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网络编程非常熟悉，才能编写出高质量的 NIO 程序。
- 3) 开发工作量和难度都非常大：例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常流的处理等等。
- 4) 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%。直到 JDK 1.7 版本该问题仍旧存在，没有被根本解决。



• Netty官网说明

官网: <https://netty.io/>

Netty is an asynchronous event-driven network application framework
for rapid development of maintainable high performance protocol servers & clients





- Netty官网说明

- 1) Netty 是由 JBOSS 提供的一个 Java 开源框架。Netty 提供异步的、基于事件驱动的网络应用程序框架，用以快速开发高性能、高可靠性的网络 IO 程序
- 2) Netty 可以帮助你快速、简单的开发出一个网络应用，相当于简化和流程化了 NIO 的开发过程
- 3) Netty 是目前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，知名的 Elasticsearch 、Dubbo 框架内部都采用了 Netty。



• Netty的优点

Netty 对 JDK 自带的 NIO 的 API 进行了封装，解决了上述问题。

- 1) 设计优雅：适用于各种传输类型的统一 API 阻塞和非阻塞 Socket；基于灵活且可扩展的事件模型，可以清晰地分离关注点；高度可定制的线程模型 - 单线程，一个或多个线程池。
- 2) 使用方便：详细记录的 Javadoc，用户指南和示例；没有其他依赖项，JDK 5（Netty 3.x）或 6（Netty 4.x）就足够了。
- 3) 高性能、吞吐量更高：延迟更低；减少资源消耗；最小化不必要的内存复制。
- 4) 安全：完整的 SSL/TLS 和 StartTLS 支持。
- 5) 社区活跃、不断更新：社区活跃，版本迭代周期短，发现的 Bug 可以被及时修复，同时，更多的新功能会被加入



• Netty版本说明

- 1) netty版本分为 netty3.x 和 netty4.x、netty5.x
- 2) 因为Netty5出现重大bug，已经被官网废弃了，目前推荐使用的是Netty4.x的稳定版本
- 3) 目前在官网可下载的版本 netty3.x netty4.0.x 和 netty4.1.x
- 4) 在本套课程中，我们讲解 Netty4.1.x 版本
- 5) netty 下载地址： <https://bintray.com/netty/downloads/netty/>





--Netty 高性能架构设计



• 线程模型基本介绍

- 1) 不同的线程模式，对程序的性能有很大影响，为了搞清**Netty** 线程模式，我们来系统的讲解下各个线程模式，最后看看**Netty** 线程模型有什么优越性.
- 2) 目前存在的线程模型有：
传统阻塞 I/O 服务模型
Reactor 模式
- 3) 根据 **Reactor** 的数量和处理资源池线程的数量不同，有 **3** 种典型的实现
 - 单 **Reactor** 单线程；
 - 单 **Reactor** 多线程；
 - 主从 **Reactor** 多线程
- 4) **Netty** 线程模式(**Netty** 主要基于主从 **Reactor** 多线程模型做了一定的改进，其中主从 **Reactor** 多线程模型有多个 **Reactor**)

- 传统阻塞 I/O 服务模型

工作原理图

黄色的框表示对象， 蓝色的框表示线程

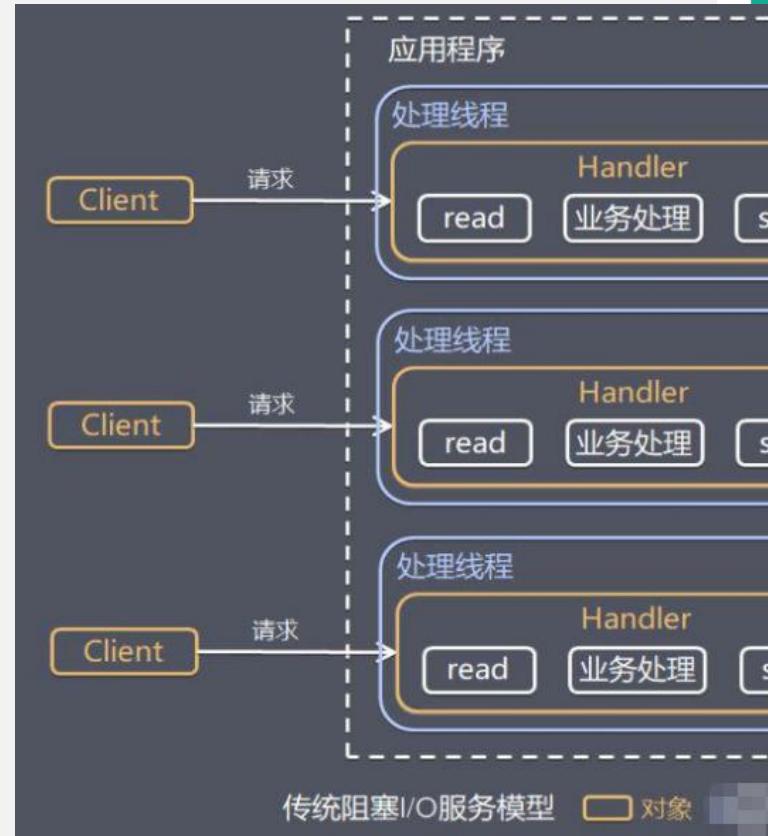
白色的框表示方法(API)

模型特点

- 1) 采用阻塞IO模式获取输入的数据
- 2) 每个连接都需要独立的线程完成数据的输入，业务处理，数据返回

问题分析

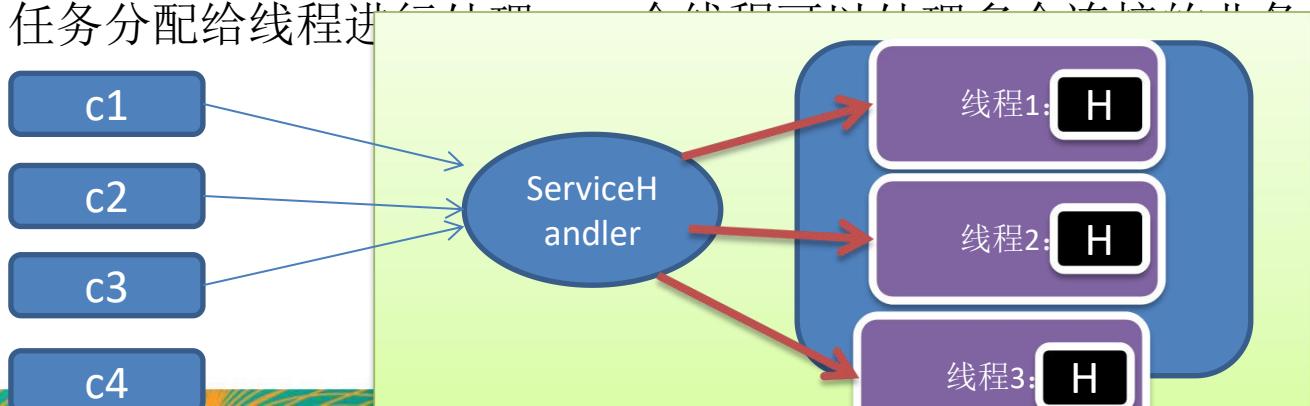
- 1) 当并发数很大，就会创建大量的线程，占用很大系统资源
- 2) 连接创建后，如果当前线程暂时没有数据可读，该线程会阻塞在read 操作，造成线程资源浪费



- Reactor 模式

针对传统阻塞 I/O 服务模型的 2 个缺点，解决方案：

- 1) 基于 I/O 复用模型：多个连接共用一个阻塞对象，应用程序只需要在一个阻塞对象等待，无需阻塞等待所有连接。当某个连接有新的数据可以处理时，操作系统通知应用程序，线程从阻塞状态返回，开始进行业务处理
Reactor 对应的叫法: 1. 反应器模式 2. 分发者模式(Dispatcher) 3. 通知者模式(notifier)
- 2) 基于线程池复用线程资源：不必再为每个连接创建线程，将连接完成后的业务处理任务分配给线程进行处理

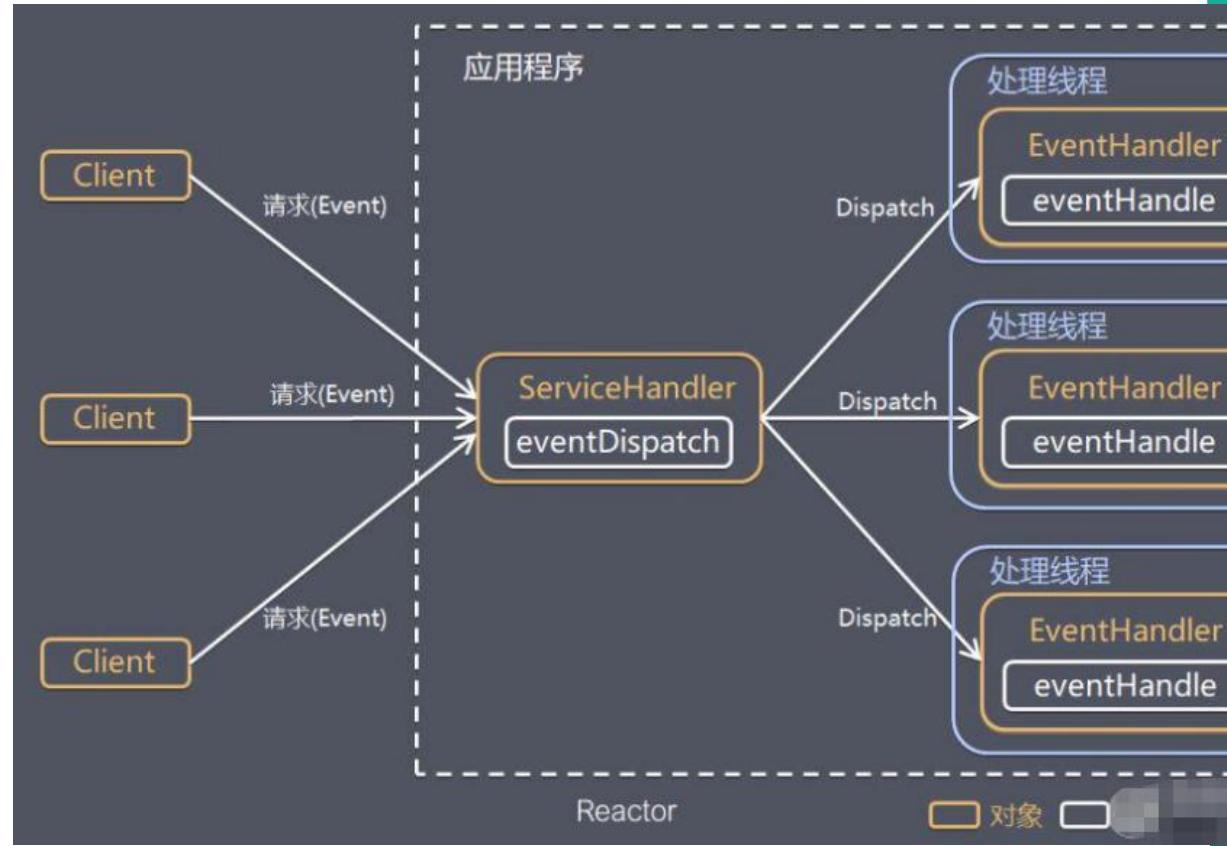


● Reactor 模式

I/O 复用结合线程池，就是 Reactor 模式基本设计思想，如图：

说明：

- 1) Reactor 模式，通过一个或多个输入同时传递给服务处理器的模式（基于事件驱动）
- 2) 服务器端程序处理传入的多个请求，并将它们同步分派到相应的处理线程，因此Reactor模式也叫 Dispatcher模式
- 3) Reactor 模式使用I/O复用监听事件，收到事件后，分发给某个线程(进程)，这点就是网络服务器高并发处理关键





- **Reactor 模式**

Reactor 模式中 核心组成:

- 1) **Reactor:** Reactor 在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对 I/O 事件做出反应。它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人；
- 2) **Handlers:** 处理程序执行 I/O 事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行非阻塞操作。



- Reactor 模式

Reactor 模式分类:

根据 Reactor 的数量和处理资源池线程的数量不同，有 3 种典型的实现

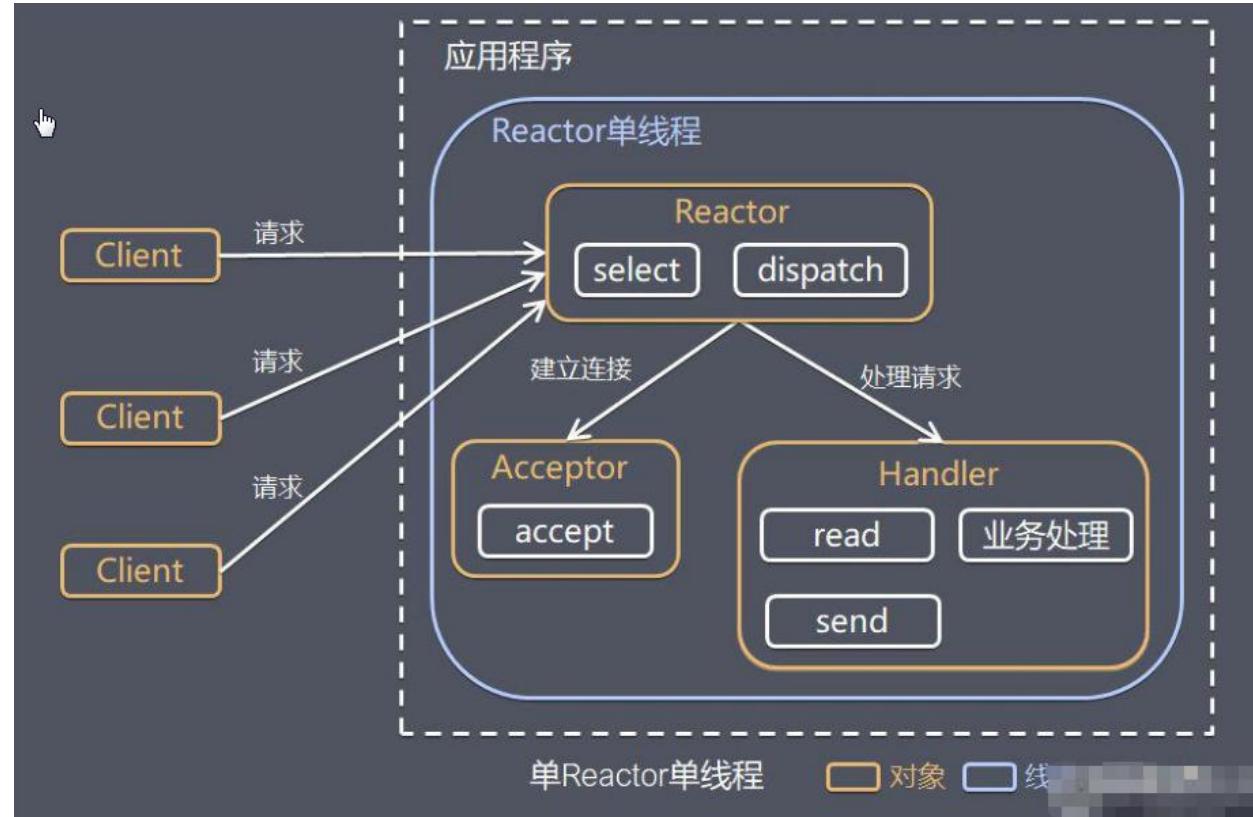
- 1) 单 Reactor 单线程
- 2) 单 Reactor 多线程
- 3) 主从 Reactor 多线程



• 单 Reactor 单线程

工作原理示意图：

演示下NIO 群聊,说明





- 单 Reactor 单线程

方案说明：

- 1) Select 是前面 I/O 复用模型介绍的标准网络编程 API，可以实现应用程序通过一个阻塞对象监听多路连接请求
- 2) Reactor 对象通过 Select 监控客户端请求事件，收到事件后通过 Dispatch 进行分发
- 3) 如果是建立连接请求事件，则由 Acceptor 通过 Accept 处理连接请求，然后创建一个 Handler 对象处理连接完成后的后续业务处理
- 4) 如果不是建立连接事件，则 Reactor 会分发调用连接对应的 Handler 来响应
- 5) Handler 会完成 Read→业务处理→Send 的完整业务流程

结合实例：服务器端用一个线程通过多路复用搞定所有的 IO 操作（包括连接，读、写等），编码简单，清晰明了，但是如果客户端连接数量较多，将无法支撑，前面的 NIO 案例就属于这种模型。



- 单 Reactor 单线程

方案优缺点分析：

- 1) 优点：模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成
- 2) 缺点：性能问题，只有一个线程，无法完全发挥多核 CPU 的性能。Handler 在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶颈
- 3) 缺点：可靠性问题，线程意外终止，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
- 4) 使用场景：客户端的数量有限，业务处理非常快速，比如 Redis 在业务处理的时间复杂度 $O(1)$ 的情况

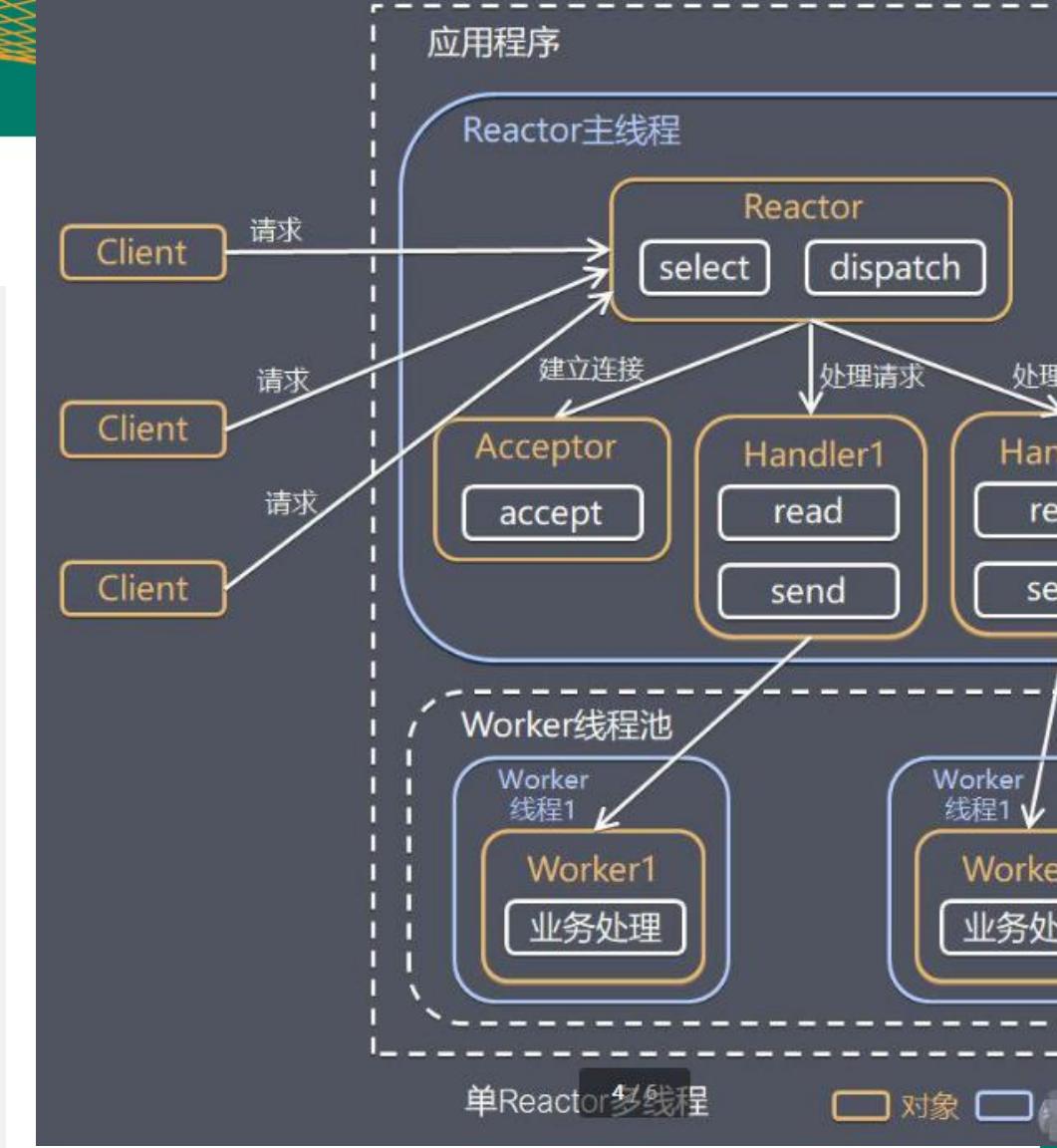


● 单Reactor多线程

工作原理示意图：

方案说明

- 1) Reactor 对象通过select 监控客户端请求事件, 收到事件后, 通过dispatch进行分发
- 2) 如果建立连接请求, 则由Acceptor 通过accept 处理连接请求, 然后创建一个Handler对象处理完成连接后的各种事件
- 3) 如果不是连接请求, 则由reactor分发调用连接对应的handler 来处理
- 4) handler 只负责响应事件, 不做具体的业务处理, 通过read 读取数据后, 会分发给后面的worker线程池的某个线程处理业务
- 5) worker 线程池会分配独立线程完成真正的业务, 并将结果返回给handler
- 6) handler收到响应后, 通过send 将结果返回给client





- 单Reactor多线程

方案优缺点分析:

- 1) 优点: 可以充分的利用多核cpu 的处理能力
- 2) 缺点: 多线程数据共享和访问比较复杂, reactor 处理所有的事件的监听和响应, 在单线程运行, 在高并发场景容易出现性能瓶颈.



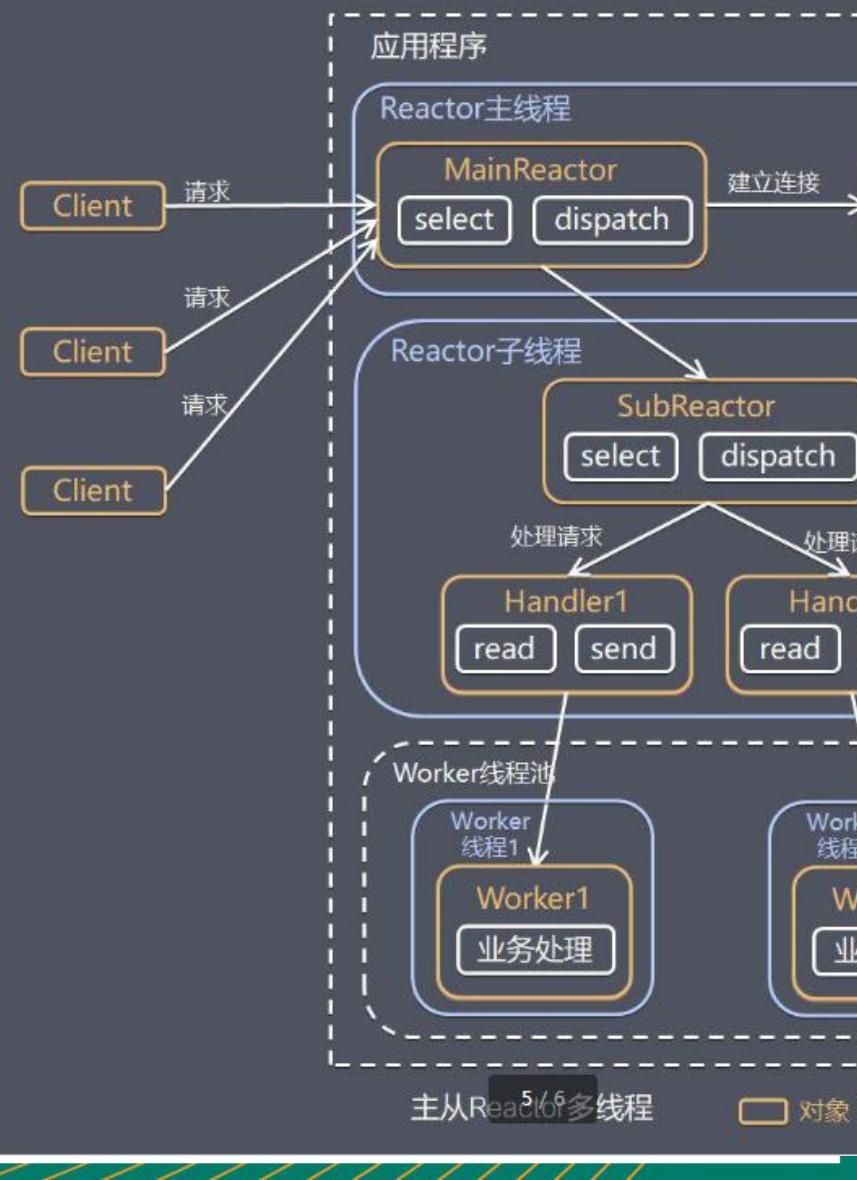
● 主从 Reactor 多线程

工作原理示意图：

针对单 Reactor 多线程模型中，Reactor 在单线程中运行，高并发场景下容易成为性能瓶颈，可以让 Reactor 在多线程中运行

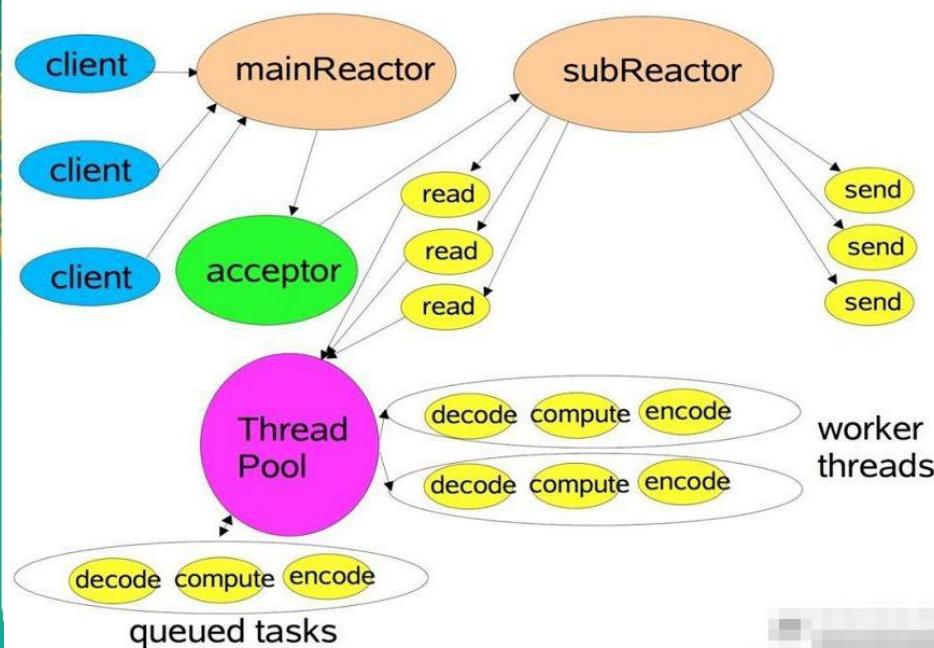
方案说明

- 1) Reactor主线程 MainReactor 对象通过select 监听连接事件, 收到事件后, 通过Acceptor 处理连接事件
- 2) 当 Acceptor 处理连接事件后, MainReactor 将连接分配给 SubReactor
- 3) subreactor 将连接加入到连接队列进行监听,并创建handler 进行各种事件处理
- 4) 当有新事件发生时, subreactor 就会调用对应的handler 处理
- 5) handler 通过read 读取数据, 分发给后面的worker 线程处理
- 6) worker 线程池分配独立的worker 线程进行业务处理, 并返回结果



- 主从 Reactor 多线程

Scalable IO in Java 对 Multiple Reactors 的原理图解:



Doug Lea

[编辑](#)

[讨论](#)

Doug Lea, 中文名为道格·利。美国国籍, 现担任[纽约州立大学Oswego分校](#)教师。

中文名	道格·利	国 籍	美国
外文名	Doug Lea	职 业	教师
隶属	纽约州立大学Oswego分校		

如果IT的历史, 是以人为主体串接起来的话, 那么肯定少不了Doug Lea。这个鼻梁挂着眼镜, 留着德王威廉二世的上永远挂着谦逊腼腆笑容, 服务于[纽约州立大学Oswego分校](#)计算机科学系的老大爷。

说他是这个世界上对Java影响力最大的一个人, 一点也不为过。因为两次Java历史上的大变革, 他都间接或直接的足轻重的角色。2004年所推出的Tiger。Tiger广纳了15项JSRs(Java Specification Requests)的语法及标准, 其中一项便是166。JSR-166是来自于Doug编写的util.concurrent包。

值得一提的是: Doug Lea也是JCP ([Java社区项目](#))中的一员。

Doug是一个无私的人, 他深知分享知识和分享苹果是不一样的, 苹果会越分越少, 而自己的知识并不会因为给了别人了, 知识的分享更能激荡出不一样的火花。《Effective JAVA》这本Java经典之作的作者Joshua Bloch便在书中特别感谢Lea是此书中许多构想的共鸣板, 感谢Doug Lea大方分享丰富而又宝贵的知识。



- 主从 Reactor 多线程

方案优缺点说明：

- 1) 优点：父线程与子线程的数据交互简单职责明确，父线程只需要接收新连接，子线程完成后续的业务处理。
- 2) 优点：父线程与子线程的数据交互简单，Reactor 主线程只需要把新连接传给子线程，子线程无需返回数据。
- 3) 缺点：编程复杂度较高

结合实例：这种模型在许多项目中广泛使用，包括 Nginx 主从 Reactor 多进程模型，Memcached 主从多线程，Netty 主从多线程模型的支持



• Reactor 模式小结

3 种模式用生活案例来理解

- 1) 单 Reactor 单线程，前台接待员和服务员是同一个人，全程为顾客服务
- 2) 单 Reactor 多线程，1 个前台接待员，多个服务员，接待员只负责接待
- 3) 主从 Reactor 多线程，多个前台接待员，多个服务生

Reactor 模式具有如下的优点：

- 1) 响应快，不必为单个同步时间所阻塞，虽然 Reactor 本身依然是同步的
- 2) 可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销
- 3) 扩展性好，可以方便的通过增加 Reactor 实例个数来充分利用 CPU 资源
- 4) 复用性好，Reactor 模型本身与具体事件处理逻辑无关，具有很高的复用性

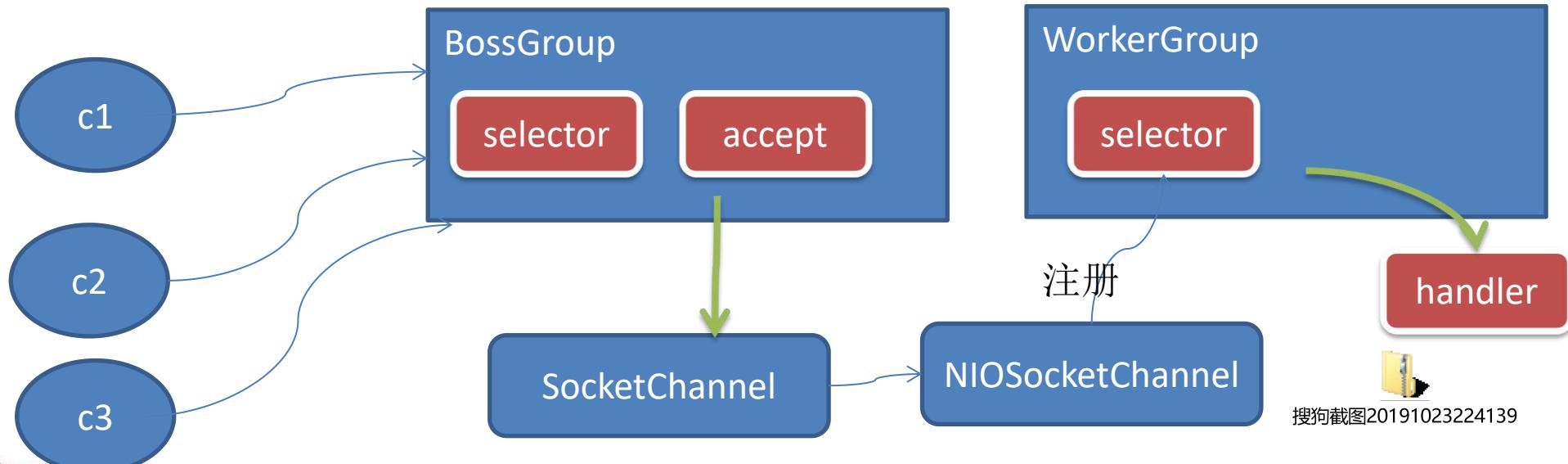


● Netty模型

工作原理示意图1-简单版

Netty 主要基于主从 Reactors 多线程模型 (如图所示)。在图中，展示了 Netty 的多线程模型，其中包含一个 BossGroup 和一个 WorkerGroup。

- 1) BossGroup 线程维护 Selector，只关注 Accept
- 2) 当接收到 Accept 事件，获取到对应的 SocketChannel，封装成 NIOSocketChannel 并注册到 Worker 线程(事件循环)，并进行维护
- 3) 当 Worker 线程监听到 selector 中通道发生自己感兴趣的事件后，就进行处理(就由 handler)，注意 handler 已经加入到通道

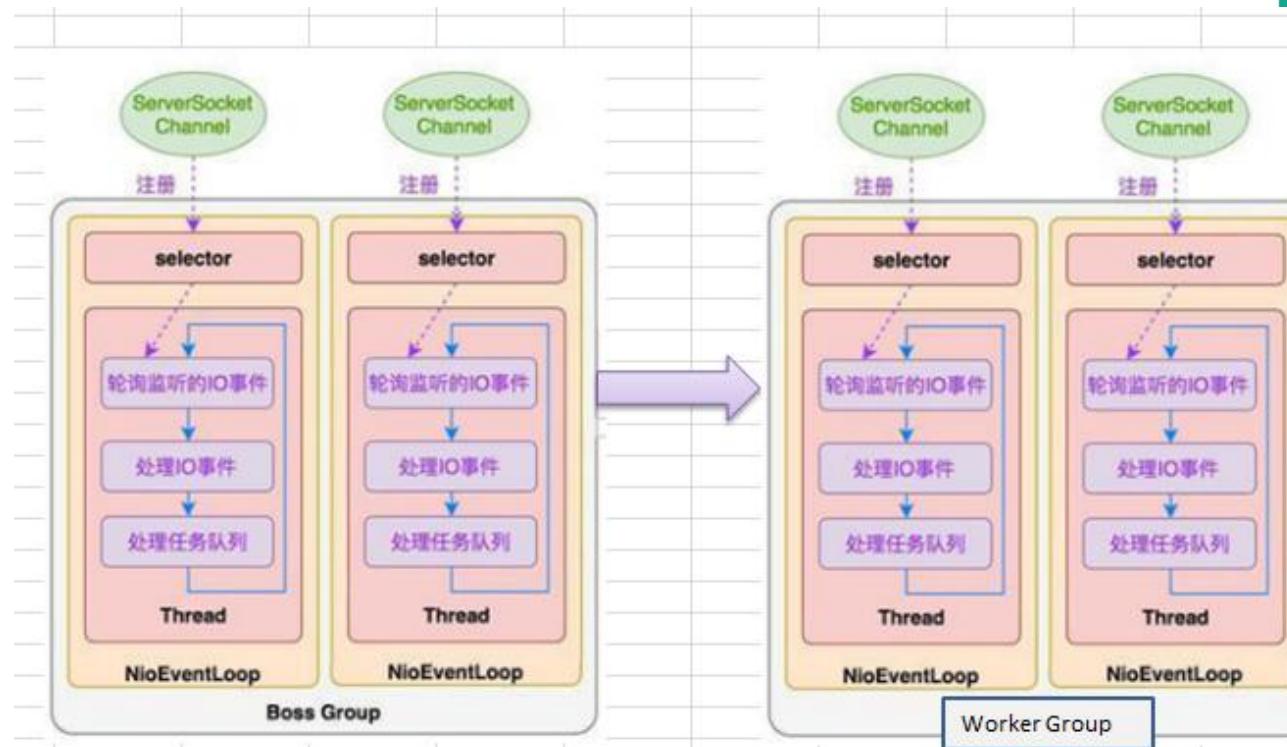




• Netty模型

工作原理示意图2-进阶版

Netty 主要基于主从 Reactors 多线程模型（如图）做了一定的改进，其中主从 Reactor 多线程模型有多个 Reactor

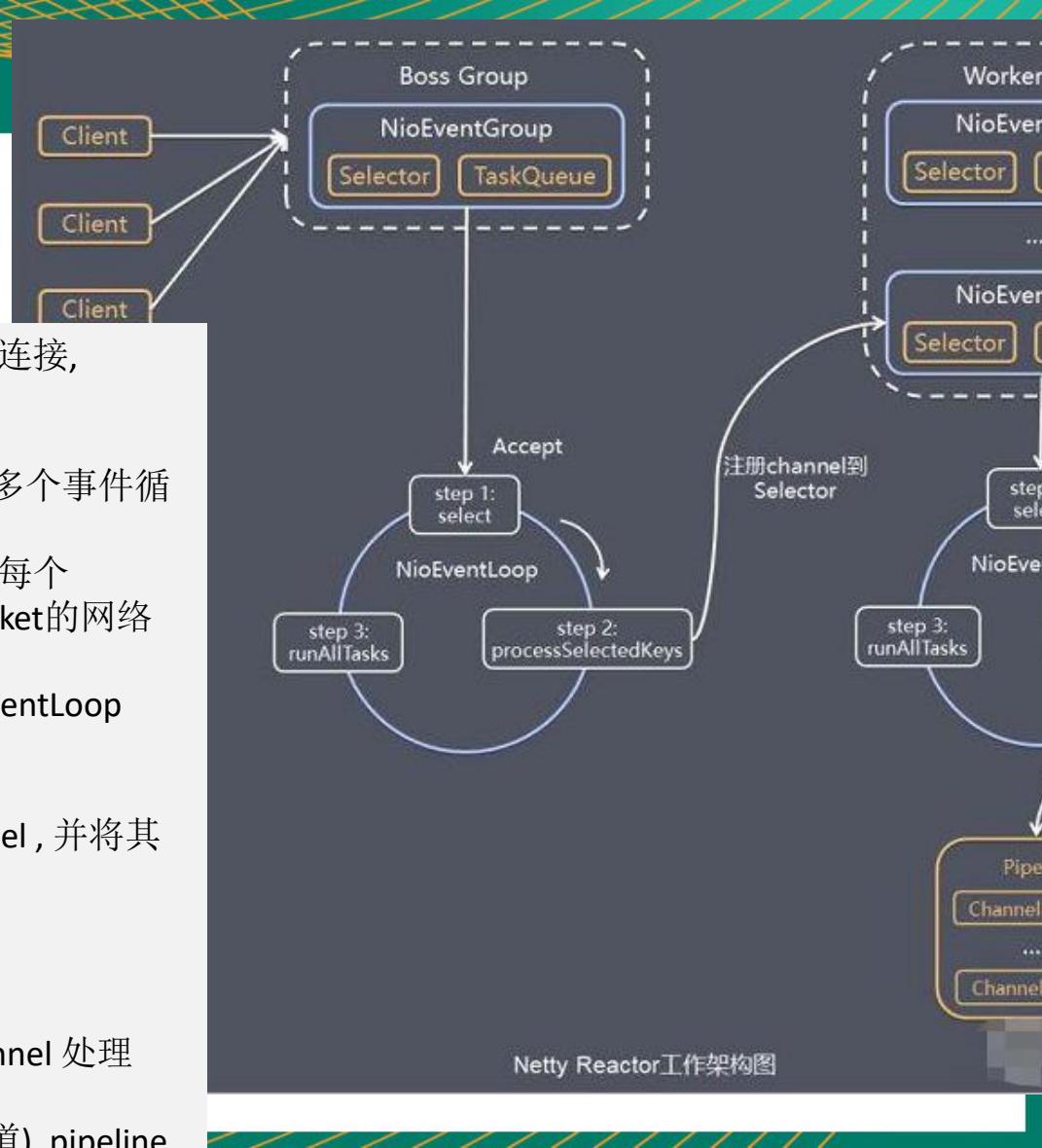




• Netty模型

工作原理示意图-详细版

- 1) Netty抽象出两组线程池 BossGroup 专门负责接收客户端的连接, WorkerGroup 专门负责网络的读写
- 2) BossGroup 和 WorkerGroup 类型都是 NioEventLoopGroup
- 3) NioEventLoopGroup 相当于一个事件循环组, 这个组中含有多个事件循环, 每一个事件循环是 NioEventLoop
- 4) NioEventLoop 表示一个不断循环的执行处理任务的线程, 每个 NioEventLoop 都有一个selector, 用于监听绑定在其上的socket的网络通讯
- 5) NioEventLoopGroup 可以有多个线程, 即可以含有多个NioEventLoop
- 6) 每个Boss NIOEventLoop 循环执行的步骤有3步
 1. 轮询accept 事件
 2. 处理accept 事件, 与client建立连接, 生成NioSocketChannel, 并将其注册到某个worker NIOEventLoop 上的 selector
 3. 处理任务队列的任务, 即 runAllTasks
- 7) 每个 Worker NIOEventLoop 循环执行的步骤
 1. 轮询read, write 事件
 2. 处理i/o事件, 即read, write 事件, 在对应NioSocketChannel 处理
 3. 处理任务队列的任务, 即 runAllTasks
- 8) 每个Worker NIOEventLoop 处理业务时, 会使用pipeline(管道) pipeline



Netty Reactor工作架构图



• Netty快速入门实例-TCP服务

- 1) 实例要求：使用IDEA 创建Netty项目
 - 2) Netty 服务器在 6668 端口监听，客户端能发送消息给服务器 "hello, 服务器~"
 - 3) 服务器可以回复消息给客户端 "hello, 客户端~"
 - 4) 目的：对Netty 线程模型 有一个初步认识，便于理解Netty 模型理论
 - 5) 看老师代码演示
- 5.1 编写服务端 5.2 编写客户端 5.3 对netty 程序进行分析，看看netty模型特点

说明：创建Maven项目，并引入Netty 包



tempsimple.zip



IDEA 项目 导入netty的



- Netty模型

任务队列中的 Task 有 3 种典型使用场景

- 1) 用户程序自定义的普通任务 [举例说明]
- 2) 用户自定义定时任务
- 3) 非当前 Reactor 线程调用 Channel 的各种方法

例如在 **推送系统** 的业务线程里面，根据 **用户的标识**，找到对应的 **Channel 引用**，然后调用 **Write** 类方法向该用户推送消息，就会进入到这种场景。最终的 **Write** 会提交到任务队列中后被 **异步消费**



- Netty模型

方案再说说明

- 1) Netty 抽象出两组线程池，BossGroup 专门负责接收客户端连接，WorkerGroup 专门负责网络读写操作。
 - 2) NioEventLoop 表示一个不断循环执行处理任务的线程，每个 NioEventLoop 都有一个 selector，用于监听绑定在其上的 socket 网络通道。
 - 3) NioEventLoop 内部采用串行化设计，从消息的读取->解码->处理->编码->发送，始终由 IO 线程 NioEventLoop 负责
- NioEventLoopGroup 下包含多个 NioEventLoop
 - 每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue
 - 每个 NioEventLoop 的 Selector 上可以注册监听多个 NioChannel
 - 每个 NioChannel 只会绑定在唯一的 NioEventLoop 上
 - 每个 NioChannel 都绑定有一个自己的 ChannelPipeline



- 异步模型

基本介绍

- 1) 异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的组件在完成后，通过状态、通知和回调来通知调用者。
- 2) Netty 中的 I/O 操作是异步的，包括 Bind、Write、Connect 等操作会简单的返回一个 ChannelFuture。
- 3) 调用者并不能立刻获得结果，而是通过 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
- 4) Netty 的异步模型是建立在 future 和 callback 的之上的。callback 就是回调。重点说 Future，它的核心思想是：假设一个方法 fun，计算过程可能非常耗时，等待 fun 返回显然不合适。那么可以在调用 fun 的时候，立马返回一个 Future，后续可以通过 Future 去监控方法 fun 的处理过程(即： Future-Listener 机制)



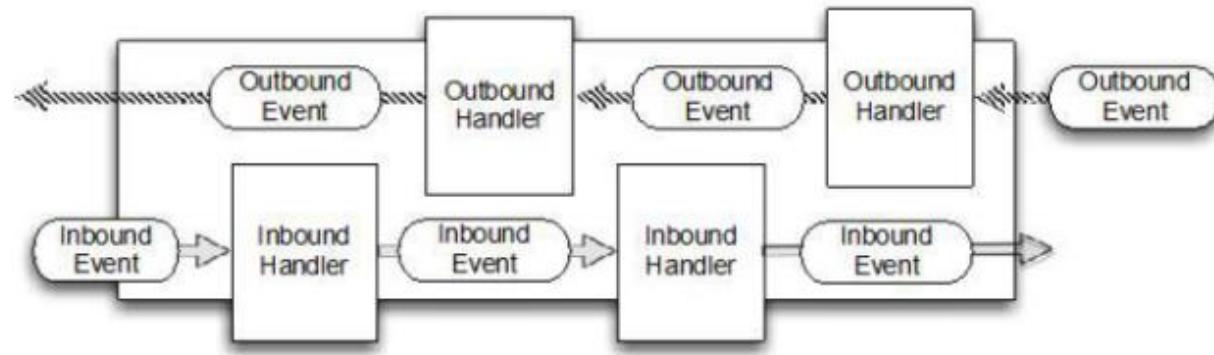
- 异步模型

Future 说明

- 1) 表示异步的执行结果, 可以通过它提供的方法来检测执行是否完成, 比如检索计算等等.
- 2) ChannelFuture 是一个接口 : **public interface ChannelFuture extends Future<Void>**
我们可以添加监听器, 当监听的事件发生时, 就会通知到监听器. 案例说明

- 异步模型

工作原理示意图



链式操作示

说明:

- 1) 在使用 Netty 进行编程时，拦截操作和转换出入站数据只需要您提供 `callback` 或利用 `future` 即可。这使得**链式操作**简单、高效，并有利于编写可重用的、通用的代码。
- 2) Netty 框架的目标就是让你的业务逻辑从网络基础应用编码中分离出来、解脱出来



- 异步模型

Future-Listener 机制

- 1) 当 Future 对象刚刚创建时，处于非完成状态，调用者可以通过返回的 ChannelFuture 来获取操作执行的状态，注册监听函数来执行完成后的操作。
- 2) 常见有如下操作
 - 通过 `isDone` 方法来判断当前操作是否完成；
 - 通过 `isSuccess` 方法来判断已完成的当前操作是否成功；
 - 通过 `getCause` 方法来获取已完成的当前操作失败的原因；
 - 通过 `isCancelled` 方法来判断已完成的当前操作是否被取消；
 - 通过 `addListener` 方法来注册监听器，当操作已完成(`isDone` 方法返回完成)，将会通知指定的监听器；如果 Future 对象已完成，则通知指定的监听器



- 异步模型

Future-Listener 机制

3) 举例说明

演示：绑定端口是异步操作，当绑定操作处理完，将会调用相应的监听器处理逻辑

```
serverBootstrap.bind(port).addListener(future -> {
    if(future.isSuccess()) {
        System.out.println(newDate() + ": 端口["+ port + "]绑定成功!");
    } else{
        System.err.println("端口["+ port + "]绑定失败!");
    }
});
```

小结：相比传统阻塞 I/O，执行 I/O 操作后线程会被阻塞住，直到操作完成；异步处理的好处是不会造成线程阻塞，线程在 I/O 操作期间可以执行别的程序，在高并发情形下会更稳定和更高的吞吐量

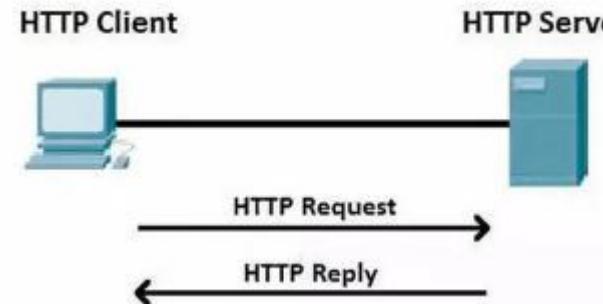


• 快速入门实例-HTTP服务

- 1) 实例要求：使用IDEA 创建Netty项目
- 2) Netty 服务器在 6668 端口监听，浏览器发出请求
"http://localhost:6668/ "
- 3) 服务器可以回复消息给客户端 "Hello! 我是服务器 5 "，并对特定请求资源进行过滤。
- 4) 目的：Netty 可以做Http服务开发，并且理解Handler实例和客户端及其请求的关系。
- 5) 看老师代码演示



HelloWorld! 我是服务器13





--Netty 核心模块组件



• Bootstrap、ServerBootstrap

1) Bootstrap 意思是引导，一个 Netty 应用通常由一个 Bootstrap 开始，主要作用是配置整个 Netty 程序，串联各个组件，Netty 中 Bootstrap 类是客户端程序的启动引导类，ServerBootstrap 是服务端启动引导类

2) 常见的方法有

- public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup)，该方法用于服务器端，用来设置两个 EventLoop
- public B group(EventLoopGroup group)，该方法用于客户端，用来设置一个 EventLoop
- public B channel(Class<? extends C> channelClass)，该方法用来设置一个服务器端的通道实现
- public <T> B option(ChannelOption<T> option, T value)，用来给 ServerChannel 添加配置
- public <T> ServerBootstrap childOption(ChannelOption<T> childOption, T value)，用来给接收到的通道添加配置
- public ServerBootstrap childHandler(ChannelHandler childHandler)，该方法用来设置业务处理类（自定义的 handler）
- public ChannelFuture bind(int inetPort)，该方法用于服务器端，用来设置占用的端口号
- public ChannelFuture connect(String inetHost, int inetPort)，该方法用于客户端，用来连接服务器端



• Future、ChannelFuture

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

2) 常见的方法有

- Channel channel(), 返回当前正在进行 IO 操作的通道
- ChannelFuture sync(), 等待异步操作执行完毕



• Channel

- 1) Netty 网络通信的组件，能够用于执行网络 I/O 操作。
- 2) 通过Channel 可获得当前网络连接的通道的状态
- 3) 通过Channel 可获得 网络连接的配置参数（例如接收缓冲区大小）
- 4) Channel 提供异步的网络 I/O 操作(如建立连接，读写，绑定端口)，异步调用意味着任何 I/O 调用都将立即返回，并且不保证在调用结束时所请求的 I/O 操作已完成
- 5) 调用立即返回一个 ChannelFuture 实例，通过注册监听器到 ChannelFuture 上，可以 I/O 操作成功、失败或取消时回调通知调用方



• Channel

- 6) 支持关联 I/O 操作与对应的处理程序
- 7) 不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，常用的 Channel 类型：

- NioSocketChannel，异步的客户端 TCP Socket 连接。
- NioServerSocketChannel，异步的服务器端 TCP Socket 连接。
- NioDatagramChannel，异步的 UDP 连接。
- NioSctpChannel，异步的客户端 Sctp 连接。
- NioSctpServerChannel，异步的 Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及文件 IO。



• Selector

- 1) Netty 基于 Selector 对象实现 I/O 多路复用，通过 Selector 一个线程可以监听多个连接的 Channel 事件。
- 2) 当向一个 Selector 中注册 Channel 后，Selector 内部的机制就可以自动不断地查询 (Select) 这些注册的 Channel 是否有已就绪的 I/O 事件（例如可读，可写，网络连接完成等），这样程序就可以很简单地使用一个线程高效地管理多个 Channel

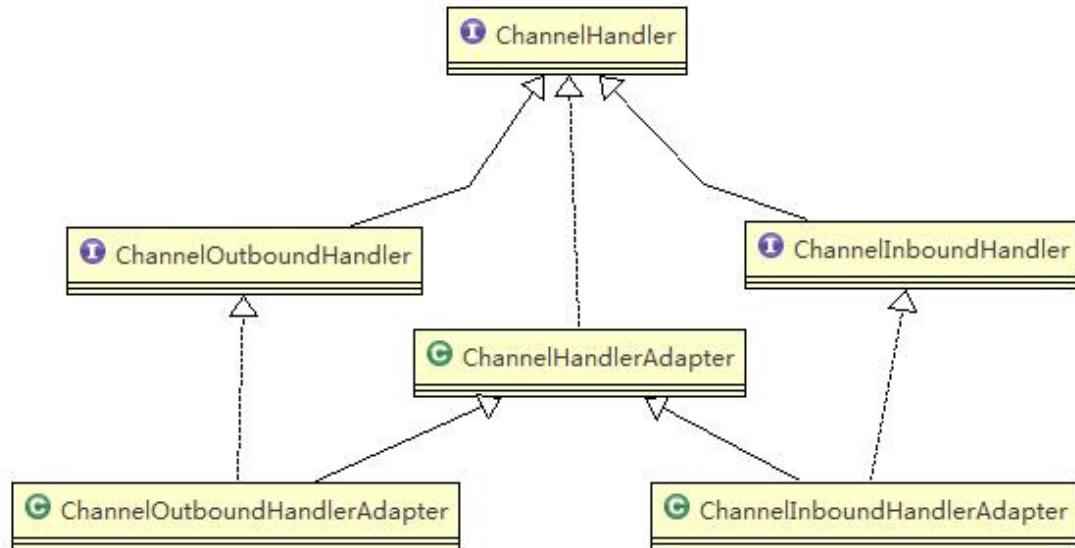


• ChannelHandler 及其实现类

- 1) ChannelHandler 是一个接口，处理 I/O 事件或拦截 I/O 操作，并将其转发到其 ChannelPipeline(业务处理链)中的下一个处理程序。
- 2) ChannelHandler 本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类
- 3) ChannelHandler 及其实现类一览图(后)

• ChannelHandler 及其实现类

相关接口和类一览图



- `ChannelInboundHandler` 用于处理入站 I/O 事件。
- `ChannelOutboundHandler` 用于处理出站 I/O 操作。

//适配器

- `ChannelInboundHandlerAdapter` 用于处理入站 I/O 事件。
- `ChannelOutboundHandlerAdapter` 用于处理出站 I/O 操作。
- `ChannelDuplexHandler` 用于处理入站和出站事件。



● ChannelHandler 及其实现类

- 4) 我们经常需要自定义一个 Handler 类去继承 ChannelInboundHandlerAdapter，然后通过重写相应方法实现业务逻辑，我们接下来看看一般都需要重写哪些方法

```
public class ChannelInboundHandlerAdapter extends ChannelHandlerAdapter
implements ChannelInboundHandler {
    public ChannelInboundHandlerAdapter() { }
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelRegistered();
    }
    public void channelUnregistered(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelUnregistered();
    }
    //通道就绪事件
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelActive();
    }
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelInactive();
    }
    //通道读取数据事件
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
        ctx.fireChannelRead(msg);
    }
}
```



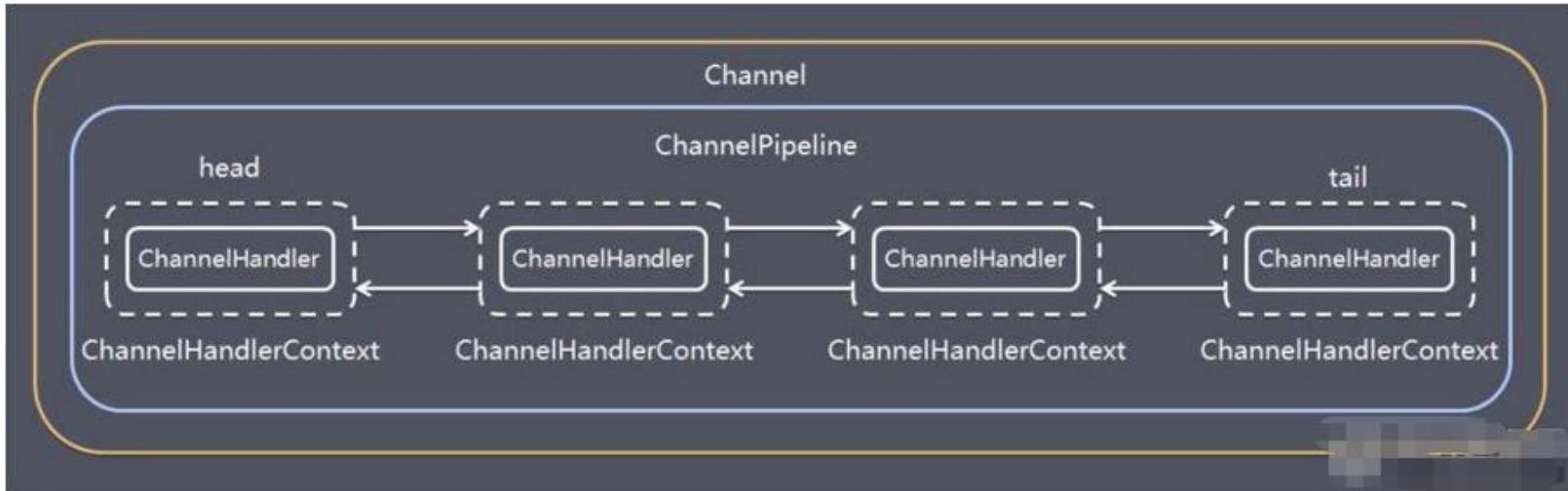
- Pipeline 和 ChannelPipeline

ChannelPipeline 是一个重点：

- 1) ChannelPipeline 是一个 Handler 的集合，它负责处理和拦截 inbound 或者 outbound 的事件和操作，相当于一个贯穿 Netty 的链。**(也可以这样理解：ChannelPipeline 是保存 ChannelHandler 的 List，用于处理或拦截 Channel 的入站事件和出站操作)**
- 2) ChannelPipeline 实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及 Channel 中各个的 ChannelHandler 如何相互交互

• Pipeline 和 ChannelPipeline

3) 在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应，它们的组成关系如下



- 一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表，并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler
- 入站事件和出站事件在一个双向链表中，入站事件会从链表 head 往后传递到最后一个入站的 handler，出站事件会从链表 tail 往前传递到最前一个出站的 handler，两种类型的 handler 互不干扰



- **Pipeline 和 ChannelPipeline**

4) 常用方法

- `ChannelPipeline addFirst(ChannelHandler... handlers)`, 把一个业务处理类（handler）添加到链中的第一个位置
- `ChannelPipeline addLast(ChannelHandler... handlers)`, 把一个业务处理类（handler）添加到链中的最后一个位置



• ChannelHandlerContext

- 1) 保存 Channel 相关的所有上下文信息，同时关联一个 ChannelHandler 对象
- 2) 即 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler，同时 ChannelHandlerContext 中也绑定了对应的 pipeline 和 Channel 的信息，方便对 ChannelHandler 进行调用。
- 3) 常用方法

- ChannelFuture close(), 关闭通道
- ChannelOutboundInvoker flush(), 刷新
- ChannelFuture writeAndFlush(Object msg), 将数据写到 ChannelPipeline 中当前
- ChannelHandler 的下一个 ChannelHandler 开始处理 (出站)



• ChannelOption

- 1) Netty 在创建 Channel 实例后,一般都需要设置 ChannelOption 参数。
- 2) ChannelOption 参数如下:

ChannelOption.SO_BACKLOG

对应 TCP/IP 协议 listen 函数中的 backlog 参数, 用来初始化服务器可连接队列大小。服务端处理客户端连接请求是顺序处理的, 所以同一时间只能处理一个客户端连接。多个客户端来的时候, 服务端将不能处理的客户端连接请求放在队列中等待处理, backlog 参数指定了队列的大小。

ChannelOption.SO_KEEPALIVE

一直保持连接活动状态

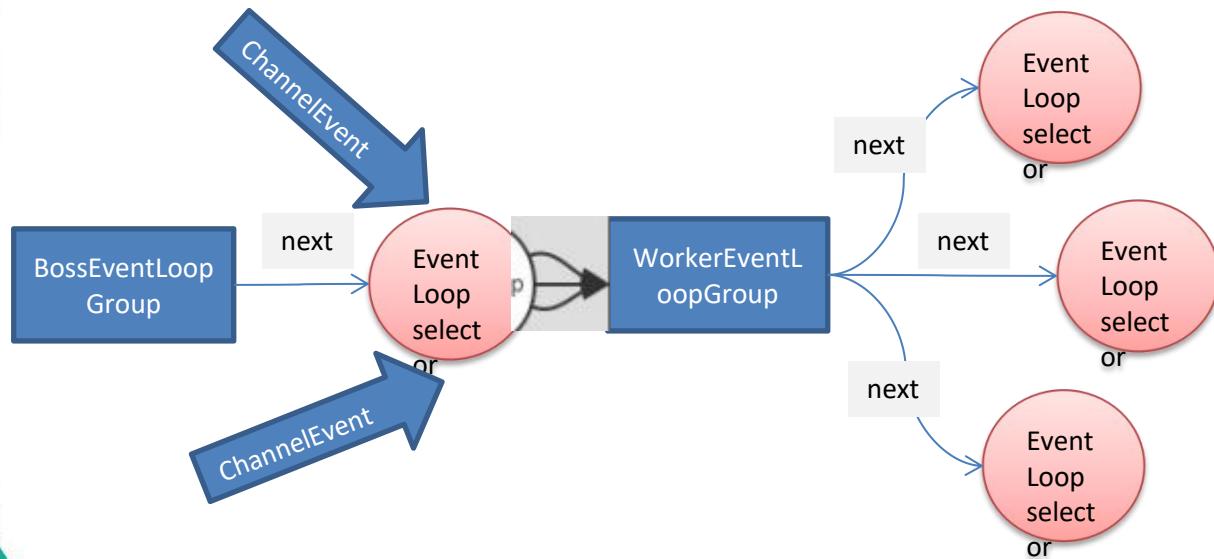


- **EventLoopGroup 和其实现类 NioEventLoopGroup**

- 1) EventLoopGroup 是一组 EventLoop 的抽象，Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，每个 EventLoop 维护着一个 Selector 实例。
- 2) EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。在 Netty 服务器端编程中，我们一般都需要提供两个 EventLoopGroup，例如：BossEventLoopGroup 和 WorkerEventLoopGroup。

- **EventLoopGroup 和其实现类 NioEventLoopGroup**

- 3) 通常一个服务端口即一个 ServerSocketChannel 对应一个 Selector 和一个 EventLoop 线程。 BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理，如下图所示



- BossEventLoopGroup 通常是一个单线程的 EventLoop，EventLoop 维护着一个注册了 ServerSocketChannel 的 Selector 实例。BossEventLoop 不断轮询 Selector 将连接事件分离出来
- 通常是 OP_ACCEPT 事件，然后将接收到的 SocketChannel 交给 WorkerEventLoopGroup
- WorkerEventLoopGroup 会由 next 选择其中一个 EventLoop 来将这个 SocketChannel 注册到其维护的 Selector 并对其后续的 IO 事件进行处理



- **EventLoopGroup 和其实现类 NioEventLoopGroup**

4) 常用方法

- public NioEventLoopGroup(), 构造方法
- public Future<?> shutdownGracefully(), 断开连接, 关闭线程



• Unpooled 类

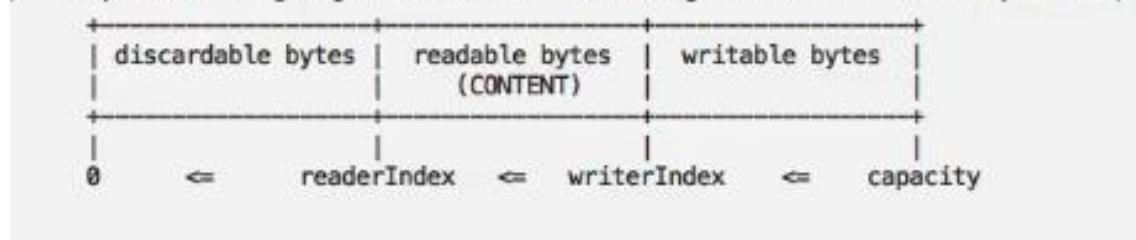
- 1) Netty 提供一个专门用来操作缓冲区(即Netty的数据容器)的工具类
- 2) 常用方法如下所示

//通过给定的数据和字符编码返回一个 ByteBuf 对象 (类似于 NIO 中的 ByteBuffer 但有区别)

```
public static ByteBuf copiedBuffer(CharSequence string, Charset charset)
```

- 3) 举例说明**Unpooled** 获取 Netty的数据容器ByteBuf 的基本使用 【案例演示】

pectively. The following diagram shows how a buffer is segmented into three areas by the two poi



NettyByteBuf.zip



• Netty应用实例-群聊系统

实例要求:

- 1) 编写一个 Netty 群聊系统，实现服务器端和客户端之间的数据简单通讯（非阻塞）
- 2) 实现多人群聊
- 3) 服务器端：可以监测用户上线，离线，并实现消息转发功能
- 4) 客户端：通过channel 可以无阻塞发送消息给其它所有用户，同时可以接受其它用户发送的消息(有服务器转发得到)
- 5) 目的：进一步理解Netty非阻塞网络编程机制



• Netty 网络编程应用实例-群聊系统

6) 看老师代码演示

Server 端

服Netty 群聊服务器 启动 ok.....

[Server]:127.0.0.1:51264上线

服务器接收到消息 时间: [2019-10-05 11:14:55] -> [127.0.0.1:51264]发送的消息: jack

服务器进行消息转发 ...

服务器接收到消息 时间: [2019-10-05 11:15:26] -> [127.0.0.1:51264]发送的消息: tom

服务器进行消息转发 ...

[Server]:127.0.0.1:51307上线

服务器接收到消息 时间: [2019-10-05 11:16:26] -> [127.0.0.1:51307]发送的消息: jack

服务器进行消息转发 ...

[Server]:127.0.0.1:51264离线

Client 端

-----127.0.0.1:51264-----

jack

tom

[127.0.0.1:51307]说: jack

[127.0.0.1:51307]说: ttt

Client 端

-----127.0.0.1:51264-----

jack

tom

[127.0.0.1:51307]说: jack

[127.0.0.1:51307]说: ttt

Client 端

-----127.0.0.1:51264-----

jack

tom

[127.0.0.1:51307]说: jack

[127.0.0.1:51307]说: ttt





- Netty心跳检测机制案例

实例要求:

- 1) 编写一个 Netty心跳检测机制案例, 当服务器超过3秒没有读时, 就提示读空闲
- 2) 当服务器超过5秒没有写操作时, 就提示写空闲
- 3) 实现当服务器超过7秒没有读或者写操作时, 就提示读写空闲



heartbeattest.zip



- Netty 通过WebSocket编程实现服务器和客户端长连接

实例要求:

- 1) Http协议是无状态的, 浏览器和服务器间的请求响应一次, 下一次会重新创建连接.
- 2) 要求: 实现基于WebSocket的长连接的全双工的交互
- 3) 改变Http协议多次请求的约束, 实现长连接了, 服务器可以发送消息给浏览器
- 4) 客户端浏览器和服务器端会相互感知, 比如服务器关闭了, 浏览器会感知, 同样浏览器关闭了, 服务器会感知

吃火锅吧, 冷

发送数据

连接开启~

服务器的时间: 2019-10-26T17:10:13.937 消息 hello
服务器的时间: 2019-10-26T17:10:21.543 消息 hello2
服务器的时间: 2019-10-26T17:10:42.199 消息 吃火锅吧, 冷

清空内容

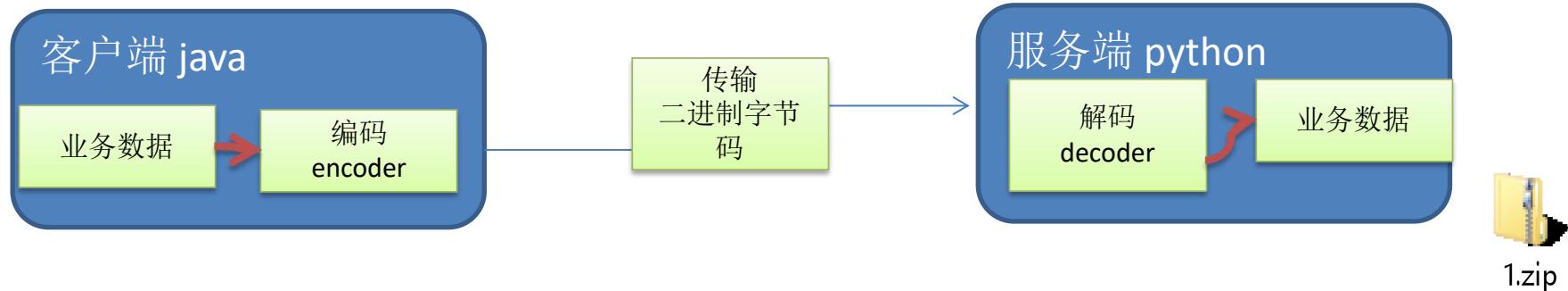


--Google Protobuf



• 编码和解码的基本介绍

- 1) 编写网络应用程序时，因为数据在网络中传输的都是二进制字节码数据，在发送数据时就需要编码，接收数据时就需要解码 [示意图]
- 2) codec(编解码器) 的组成部分有两个：decoder(解码器)和 encoder(编码器)。
encoder 负责把业务数据转换成字节码数据，decoder 负责把字节码数据转换成业务数据





- Netty 本身的编码解码的机制和问题分析

1) Netty 自身提供了一些 codec(编解码器)

2) Netty 提供的编码器

- StringEncoder, 对字符串数据进行编码
- ObjectEncoder, 对 Java 对象进行编码
- ...

3) Netty 提供的解码器

- StringDecoder, 对字符串数据进行解码
- ObjectDecoder, 对 Java 对象进行解码
- ...

4) Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的编码和解码，底层使用的仍是 Java 序列化技术，而 Java 序列化技术本身效率就不高，存在如下问题

- 无法跨语言
- 序列化后的体积太大，是二进制编码的 5 倍多。
- 序列化性能太低



- **Protobuf**

Protobuf基本介绍和使用示意图

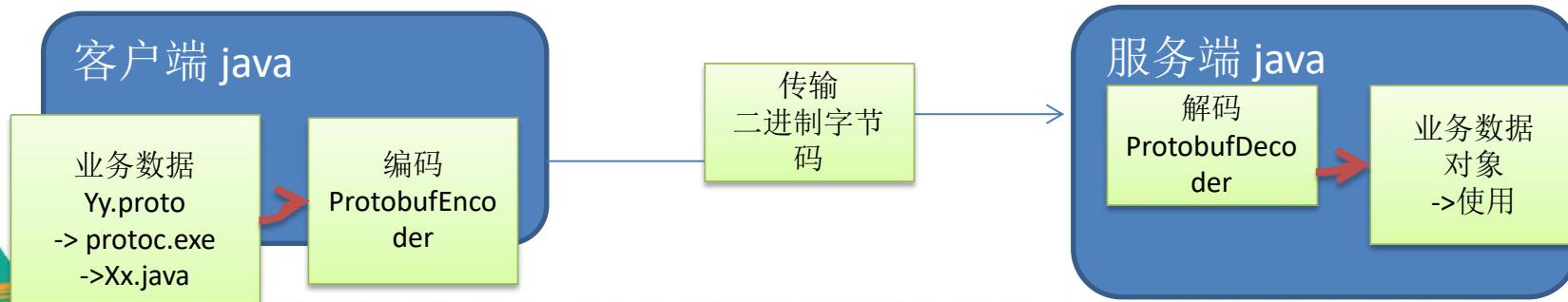
- 1) Protobuf 是 Google 发布的开源项目，全称 Google Protocol Buffers，是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 **RPC[远程过程调用 remote procedure call] 数据交换格式**。
目前很多公司 http+json → tcp+protobuf
- 2) 参考文档：<https://developers.google.com/protocol-buffers/docs/proto> 语言指南
- 3) Protobuf 是以 message 的方式来管理数据的。
- 4) 支持跨平台、**跨语言**，即[客户端和服务器端可以是不同的语言编写的]（**支持目前绝大多数语言**，例如 C++、C#、Java、python 等）



● Protobuf

Protobuf基本介绍和使用示意图

- 5) 高性能，高可靠性
- 6) 使用 protobuf 编译器能自动生成代码，Protobuf 是将类的定义使用.proto 文件进行描述。说明，在idea 中编写 .proto 文件时，会自动提示是否[下载 .ptotot 编写插件](#). 可以让语法高亮。
- 7) 然后通过 protoc.exe 编译器根据.proto 自动生成.java 文件
- 8) protobuf 使用示意图





● Protobuf

Protobuf快速入门实例

编写程序，使用Protobuf完成如下功能

- 1) 客户端可以发送一个Student PoJo 对象到服务器 (通过 Protobuf 编码)
- 2) 服务端能接收Student PoJo 对象，并显示信息(通过 Protobuf 解码)
- 3) 具体看老师演示步骤



coded.zip



Netty中使用ProtoBuf的步

.proto Type	Notes	C++ Type
double		double
float		float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64
uint32	Uses variable-length encoding.	uint32
uint64	Uses variable-length encoding.	uint64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32
sint64	Uses variable-length encoding. Signed int value. These more efficiently	int64
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28} .	uint32
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56} .	uint64
sfixed32	Always four bytes.	int32
sfixed64	Always eight bytes.	int64
bool		bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string



- **Protobuf**

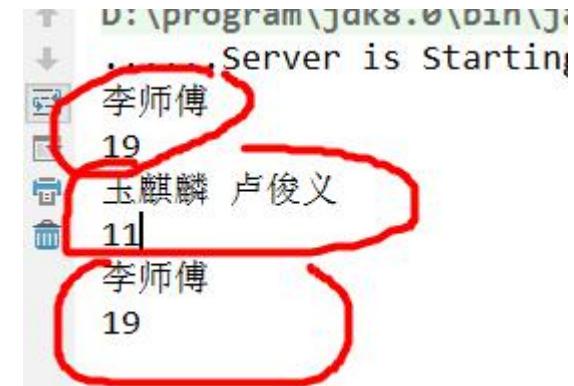
Protobuf快速入门实例2

编写程序，使用**Protobuf**完成如下功能

- 1) 客户端可以随机发送**Student PoJo/ Worker PoJo** 对象到服务器 (通过 Protobuf 编码)
- 2) 服务端能接收**Student PoJo/ Worker PoJo** 对象(需要判断是哪种类型)，并显示信息(通过 Protobuf 解码)
- 3) 具体看老师演示步骤



codec2.zip





--Netty编解码器和handler的调用机制



• 基本说明

- 1) netty的组件设计：Netty的主要组件有Channel、EventLoop、ChannelFuture、ChannelHandler、ChannelPipe等
- 2) ChannelHandler充当了处理入站和出站数据的应用程序逻辑的容器。例如，实现ChannelInboundHandler接口（或ChannelInboundHandlerAdapter），你就可以接收入站事件和数据，这些数据会被业务逻辑处理。当要给客户端发送响应时，也可以从ChannelInboundHandler冲刷数据。业务逻辑通常写在一个或者多个ChannelInboundHandler中。ChannelOutboundHandler原理一样，只不过它是用来处理出站数据的

- 基本说明

- 3) ChannelPipeline提供了ChannelHandler链的容器。以客户端应用程序为例，如果事件的运动方向是从客户端到服务端的，那么我们称这些事件为出站的，即客户端发送给服务端的数据会通过pipeline中的一系列 ChannelOutboundHandler，并被这些Handler处理，反之则称为入站的

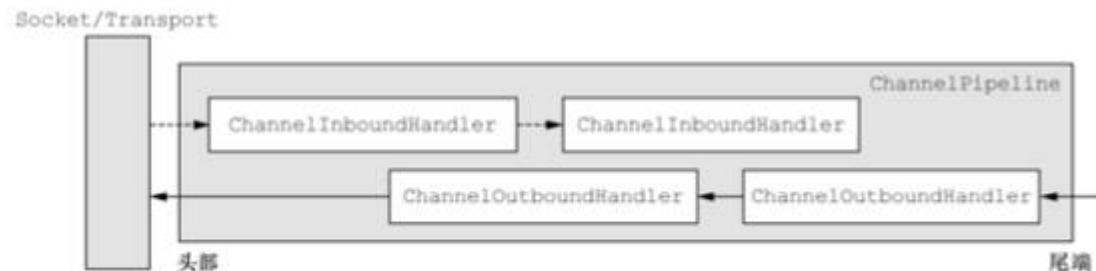


图 3-3 包含入站和出站 ChannelHandler 的 ChannelPipeline

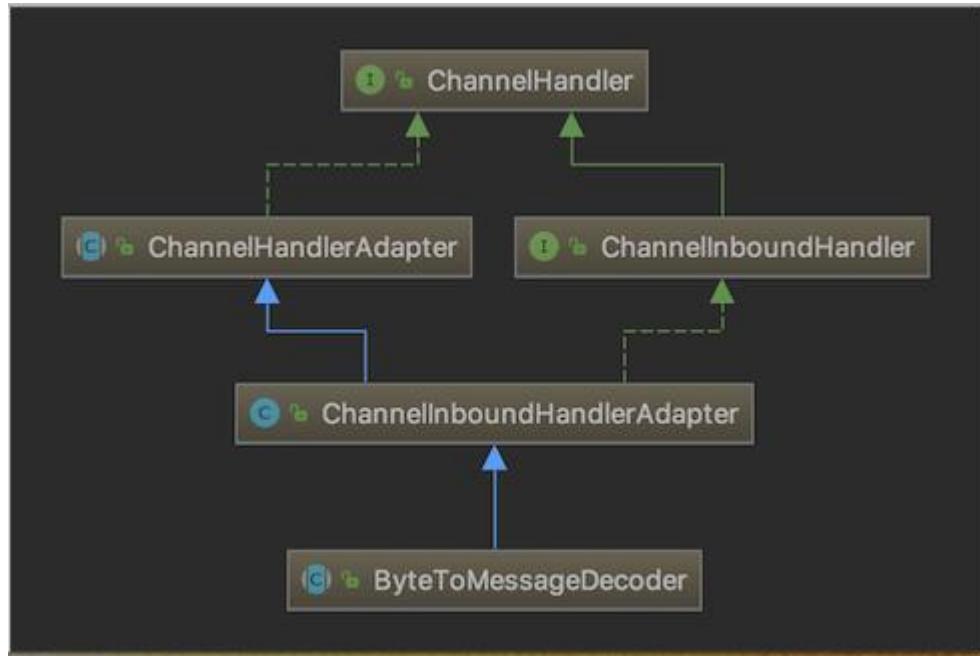


- 编码解码器

- 1) 当**Netty**发送或者接受一个消息的时候，就将会发生一次数据转换。入站消息会被解码：从字节转换为另一种格式（比如**java**对象）；如果是出站消息，它会被编码成字节。
- 2) Netty提供一系列实用的编解码器，他们都实现了**ChannelInboundHandler**或者**ChannelOutboundHandler**接口。在这些类中，**channelRead**方法已经被重写了。以入站为例，对于每个从入站**Channel**读取的消息，这个方法会被调用。随后，它将调用由解码器所提供的**decode()**方法进行解码，并将已经解码的字节转发给**ChannelPipeline**中的下一个**ChannelInboundHandler**。

- 解码器-ByteToMessageDecoder

- 1) 关系继承图
- 2) 由于不可能知道远程节点是否会一次性发送一个完整的信息，tcp有可能出现粘包拆包的问题，这个类会对入站数据进行缓冲，直到它准备好被处理.



- 解码器-ByteToMessageDecoder

3) 一个关于ByteToMessageDecoder实例分析

```
public class TolIntegerDecoder extends ByteToMessageDecoder {  
    @Override  
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception  
{  
    if (in.readableBytes() >= 4) {  
        out.add(in.readInt());  
    }  
}  
}
```

说明：

- 1) 这个例子，每次入站从ByteBuf中读取4字节，将其解码为一个int，然后将它添加到下一个List中。当没有更多元素可以被添加到该List中时，它的内容将会被发送给下一个ChannelInboundHandler。int在被添加到List中时，会被自动装箱为Integer。在调用readInt()方法前必须验证所输入的ByteBuf是否具有足够的数据
- 2) decode 执行分析图 [示意图]



decode 执行分析



• Netty的handler链的调用机制

实例要求:

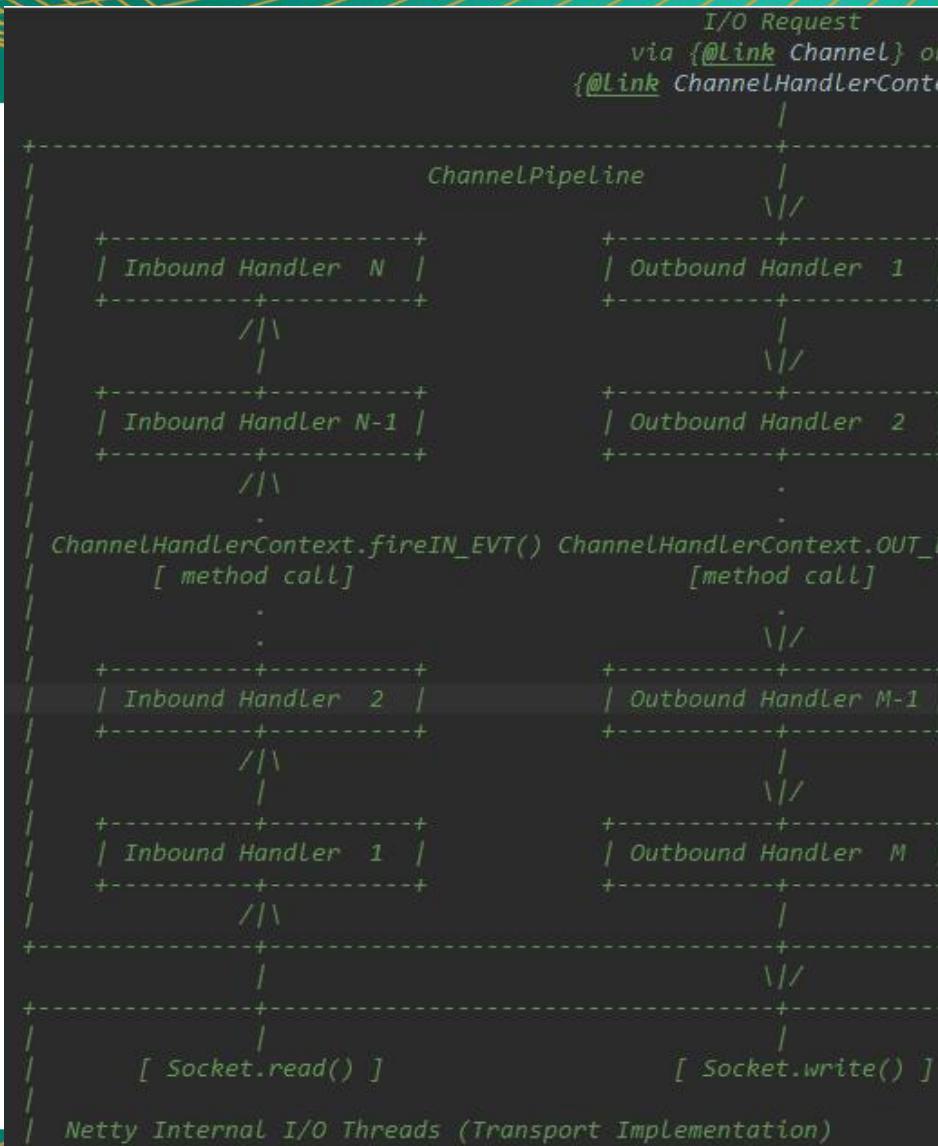
- 1) 使用自定义的编码器和解码器来说明Netty的handler 调用机制
客户端发送long -> 服务器
服务端发送long -> 客户端
- 2) 案例演示



inboundhandlerandoutboundhandler.zip

3) 结论

- 不论解码器handler 还是 编码器handler 即接收的消息类型必须与待处理的消息类型一致，否则该handler不会被执行
- 在解码器 进行数据解码时，需要判断缓存区(ByteBuf)的数据是否足够。否则接收到的





• 解码器-ReplayingDecoder

- 1) public abstract class ReplayingDecoder<S> extends ByteToMessageDecoder
 - 2) ReplayingDecoder扩展了ByteToMessageDecoder类，使用这个类，我们不必调用 readableBytes()方法。参数S指定了用户状态管理的类型，其中Void代表不需要状态管理
- 3) 应用实例：** 使用ReplayingDecoder 编写解码器，对前面的案例进行简化 [案例演示]
- 4) ReplayingDecoder使用方便，但它也有一些局限性：
 - 并不是所有的 ByteBuf 操作都被支持，如果调用了一个不被支持的方法，将会抛出一个 UnsupportedOperationException。
 - ReplayingDecoder 在某些情况下可能稍慢于 ByteToMessageDecoder，例如网络缓慢并且消息格式复杂时，消息会被拆成了多个碎片，速度变慢

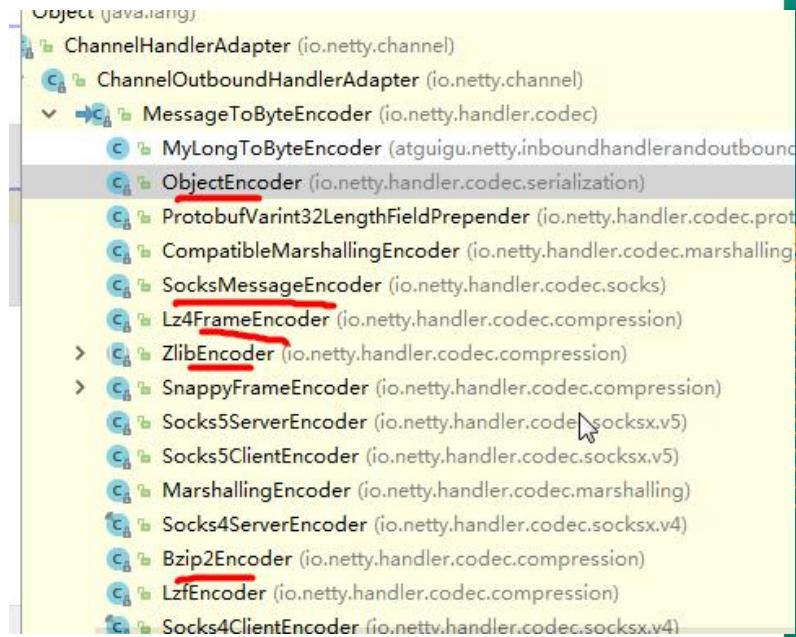


- 其它编解码器

其它解码器

- 1) **LineBasedFrameDecoder**: 这个类在Netty内部也有使用，它使用行尾控制字符（\n或者\r\n）作为分隔符来解析数据。
- 2) **DelimiterBasedFrameDecoder**: 使用自定义的特殊字符作为消息的分隔符。
- 3) **HttpObjectDecoder**: 一个HTTP数据的解码器
- 4) **LengthFieldBasedFrameDecoder**: 通过指定长度来标识整包消息，这样就可以自动的处理黏包和半包消息。

其它编码器





• Log4j 整合到Netty

- 1) 在Maven 中添加对Log4j的依赖 在 pom.xml
- 2) 配置 Log4j , 在 resources/log4j.properties

?

- 3) 演示整合

```
[DEBUG] PlatformDependent0 - -Dio.netty.noUnsafe: false
[DEBUG] PlatformDependent0 - Java version: 8
[DEBUG] PlatformDependent0 - sun.misc.Unsafe.theUnsafe: available
[DEBUG] PlatformDependent0 - sun.misc.Unsafe.copyMemory: available
[DEBUG] PlatformDependent0 - java.nio.Buffer.address: available
```

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.25</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
    <scope>test</scope>
</dependency>
```



--TCP 粘包和拆包 及解决方案

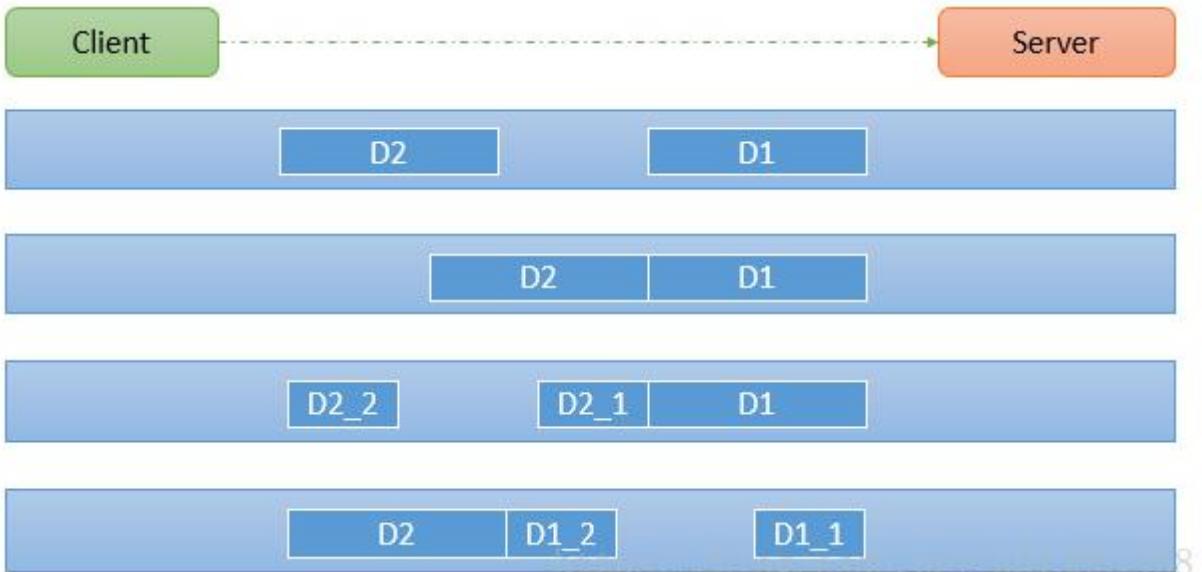


• TCP 粘包和拆包基本介绍

- 1) TCP是面向连接的，面向流的，提供高可靠性服务。收发两端（客户端和服务器端）都要有一一成对的**socket**，因此，发送端为了将多个发给接收端的包，更有效的发给对方，使用了优化方法（**Nagle算法**），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。这样做虽然提高了效率，但是接收端就难于分辨出完整的数据包了，因为**面向流的通信是无消息保护边界**的
- 2) 由于TCP无消息保护边界，需要在接收端处理消息边界问题，也就是我们所说的粘包、拆包问题，看一张图

• TCP 粘包和拆包基本介绍

3) TCP粘包、拆包图解



假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到字节数是不定的，故可能存在以下四种情况：

- 1) 服务端分两次读取到了两个独立的数据分别是D1和D2，没有粘包和拆包
- 2) 服务端一次接受到了两个数据包，D1和D2粘合在一起，称之为TCP粘包
- 3) 服务端分两次读取到了数据包，第一次取到了完整的D1包和D2包的部分内容，二次读取到了D2包的剩余内容，这称之为TCP拆包
- 4) 服务端分两次读取到了数据包，第一次取到了D1包的部分内容D1_1，第二次读取到了D1包的剩余部分内容D1_2和完整的D2包。



- TCP 粘包和拆包现象实例

在编写Netty 程序时，如果没有做处理，就会发生粘包和拆包的问题

看一个具体的实例：





• TCP 粘包和拆包解决方案

- 1) 使用自定义协议 + 编解码器 来解决
- 2) 关键就是要解决 **服务器端每次读取数据长度的问题**, 这个问题解决, 就不会出现服务器多读或少读数据的问题, 从而避免的TCP 粘包、拆包。

看一个具体的实例:

- 1) 要求客户端发送 5 个 Message 对象, 客户端每次发送一个 Message 对象
- 2) 服务器端每次接收一个Message, 分5次进行解码, 每读取到一个Message , 会回复一个Message 对象 给客户端.



protocoltcp.zip

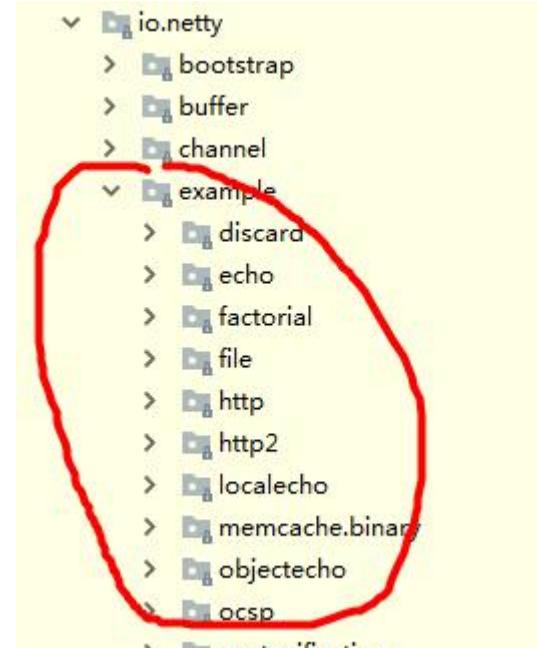


--Netty 核心源码剖析



- 基本说明

- 1) 只有看过**Netty**源码，才能说是真的掌握了**Netty**框架。
- 2) 在 `io.netty.example` 包下，有很多**Netty**源码案例，可以用来分析
- 3) 源码分析章节 是**针对有Java项目经验，并且玩过框架源码的人员讲的**，否则你听起来会有相当的难度。





- Netty 启动过程源码剖析

源码剖析目的

用源码分析的方式走一下 Netty（服务器）的启动过程，更好的理解Netty的整体设计和运行机制。



- Netty 启动过程源码剖析

源码剖析

- 说明:

- 1) 源码需要剖析到Netty 调用doBind方法，追踪到 NioServerSocketChannel的doBind
- 2) 并且要Debug 程序到 NioEventLoop类 的**run代码，无限循环**，在服务器端运行。

```
    @Override
    protected void run() {
        for (;;) {
            try {
                switch (selectStrategy.calculateStrategy(select
                    hasTasks())) {
                    case SelectStrategy.CONTINUE:
                        continue;
                    case SelectStrategy.SELECT:
                        select(wakenUp.getAndSet(false));
                }
            } catch (Exception e) {
                logger.error("Exception in event loop", e);
            }
        }
    }
```

- 源码剖析



Netty 启动过程源码



图片1.z



- Netty 启动过程源码剖析

Netty启动过程梳理

- 1) 创建2个 EventLoopGroup 线程池数组。数组默认大小CPU*2，方便chooser选择线程池时提高性能
- 2) BootStrap 将 boss 设置为 group属性，将 worker 设置为 childer 属性
- 3) 通过 bind 方法启动，内部重要方法为 initAndRegister 和 doBind 方法
- 4) initAndRegister 方法会反射创建 NioServerSocketChannel 及其相关的 NIO 的对象， pipeline , unsafe，同时也为 pipeline 初始了 head 节点和 tail 节点。
- 5) 在register0 方法成功以后调用在 doBind 方法中调用 doBind0 方法，该方法会 调用 NioServerSocketChannel 的 doBind 方法对 JDK 的 channel 和端口进行绑定，完成 Netty 服务器的所有启动，并开始监听连接事件



- Netty 接受请求过程源码剖析

源码剖析目的

- 1) 服务器启动后肯定是要接受客户端请求并返回客户端想要的信息的，下面源码分析 Netty 在启动之后是如何接受客户端请求的
- 2) 在 io.netty.example 包下



- Netty 接受请求过程源码剖析

源码剖析

- 说明：

- 1) 从之前服务器启动的源码中，我们得知，服务器最终注册了一个 Accept 事件等待客户端的连接。我们也知道，`NioServerSocketChannel` 将自己注册到了 boss 单例线程池（reactor 线程）上，也就是 EventLoop。
- 2) 先简单说下 EventLoop 的逻辑(后面我们详细讲解 EventLoop)

- ✓ EventLoop 的作用是一个死循环，而这个循环中做3件事情：
- ✓ 有条件的等待 Nio 事件。
- ✓ 处理 Nio 事件。
- ✓ 处理消息队列中的任务。

- 3) 仍用前面的项目来分析：进入到 `NioEventLoop` 源码中后，在 `private void processSelectedKey(SelectionKey k, AbstractNioChannel ch)` 方法开始调试



- Netty 接受请求过程源码剖析

源码剖析

4) 最终我们要分析到AbstractNioChannel 的 `doBeginRead` 方法，当到这个方法时，
针对于这个客户端的连接就完成了，接下来就可以监听读事件了

- 源码剖析



Netty 接受请求过程源



- Netty 接受请求过程源码剖析

Netty接受请求过程梳理

总体流程：接受连接----->创建一个新的NioSocketChannel----->注册到一个 worker EventLoop 上-----> 注册select Read 事件。

- 1) 服务器轮询 Accept 事件，获取事件后调用 unsafe 的 read 方法，这个 unsafe 是 ServerSocket 的内部类，该方法内部由2部分组成
- 2) doReadMessages 用于创建 NioSocketChannel 对象，该对象包装 JDK 的 Nio Channel 客户端。该方法会像创建 ServerSocketChanel 类似创建相关的 pipeline , unsafe, config
- 3) 随后执行 执行 pipeline.fireChannelRead 方法，并将自己绑定到一个 chooser 选择器选择的 workerGroup 中的一个 EventLoop。并且注册一个0，表示注册成功，但并没有注册读（1）事件



- Pipeline Handler HandlerContext创建源码剖析

源码剖析目的

Netty 中的 **ChannelPipeline**、**ChannelHandler** 和 **ChannelHandlerContext** 是非常核心的组件, 我们从源码来分析 Netty 是如何设计这三个核心组件的, 并分析是如何创建和协调工作的.



- Pipeline Handler HandlerContext创建源码剖析

源码剖析

- 说明

分析过程中，有很多的图形，所以我们准备了一个文档，在文档的基础上来做源码剖析

- 源码剖析



Pipeline Handler HandlerContext创建源码剖



- Pipeline Handler HandlerContext创建源码剖析

Pipeline Handler HandlerContext创建过程梳理

- 1) 每当创建 ChannelSocket 的时候都会创建一个绑定的 pipeline，一对一的关系，创建 pipeline 的时候也会创建 tail 节点和 head 节点，形成最初的链表。
- 2) 在调用 pipeline 的 addLast 方法的时候，会根据给定的 handler 创建一个 Context，然后，将这个 Context 插入到链表的尾端（tail 前面）。
- 3) Context 包装 handler，多个 Context 在 pipeline 中形成了双向链表
- 4) 入站方向叫 inbound，由 head 节点开始，出站方法叫 outbound，由 tail 节点开始



- ChannelPipeline 调度 handler 的源码剖析

源码剖析目的

- 1) 当一个请求进来的时候，ChannelPipeline 是如何调用内部的这些 handler 的呢？
我们一起来分析下。
- 2) 首先，当一个请求进来的时候，会第一个调用 pipeline 的相关方法，如果是入站事件，这些方法由 fire 开头，表示开始管道的流动。让后面的 handler 继续处理



- ChannelPipeline 调度 handler 的源码剖析

源码剖析

- 说明

- 1) 当浏览器输入 `http://localhost:8007`。可以看到会执行handler
- 2) 在Debug时，可以将断点下在 DefaultChannelPipeline 类的

```
public final ChannelPipeline fireChannelActive() {  
    AbstractChannelHandlerContext.invokeChannelActive(head); //断点  
    return this;  
}
```

- 源码分析



ChannelPipeline 是如何调度 handler



示意图.



- ChannelPipeline 调度 handler 的源码剖析

ChannelPipeline 调度 handler 梳理

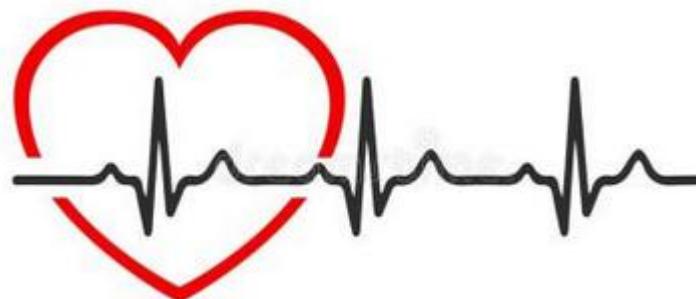
- 1) Context 包装 handler，多个 Context 在 pipeline 中形成了双向链表，入站方向叫 inbound，由 head 节点开始，出站方法叫 outbound，由 tail 节点开始。
- 2) 而节点中间的传递通过 AbstractChannelHandlerContext 类内部的 fire 系列方法，找到当前节点的下一个节点不断的循环传播。是一个过滤器形式完成对handler 的调度



- Netty 心跳(heartbeat)服务源码剖析

源码剖析目的

Netty 作为一个网络框架，提供了诸多功能，比如编码解码等，Netty 还提供了非常重要的一个服务——心跳机制heartbeat。通过心跳检查对方是否有效，这是 RPC 框架中是必不可少的功能。下面我们分析一下Netty内部 心跳服务源码实现。



- Netty 心跳(heartbeat)服务源码剖析

源码剖析

● 说明

1) Netty 提供了 IdleStateHandler , ReadTimeoutHandler, WriteTimeoutHandler 三个 **Handler** 检测连接的有效性，重点分析 **IdleStateHandler** .

2) 如图

序号	名称	作用
1	IdleStateHandler	当连接的空闲时间（读或者写）太长时，将会触发一个 IdleStateEvent 事件。然后，你可以通过你的 ChannelInboundHandler 中重写 userEventTriggered 方法来处理该事件。
2	ReadTimeoutHandler	如果在指定的事件没有发生读事件，就会抛出这个异常，并自动关闭这个连接。你可以在 exceptionCaught 方法中处理这个异常。
3	WriteTimeoutHandler	当一个写操作不能在一定的时间内完成时，抛出此异常，并关闭连接。你同样可以在 exceptionCaught 方法中处理这个异常。



OutputChanged流程



- Netty 心跳(heartbeat)服务源码剖析

源码剖析

- 源码剖析



Netty 心跳(heartbeat)服务源码



- Netty 核心组件 EventLoop 源码剖析

源码剖析目的

Echo第一行代码就是：`EventLoopGroup bossGroup = new NioEventLoopGroup(1);`
下面分析其最核心的组件 EventLoop。



- Netty 核心组件 EventLoop 源码剖析

源码剖析

- 源码剖析



Netty 核心组件 EventLoop 源码



eventloop继承图.



- **handler** 中加入线程池和**Context** 中添加线程池的源码剖析

源码剖析目的

- 1) 在 Netty 中做耗时的，不可预料的操作，比如数据库，网络请求，会严重影响 Netty 对 Socket 的处理速度。
- 2) 而解决方法就是将耗时任务添加到异步线程池中。但就添加线程池这步操作来讲，可以有2种方式，而且这2种方式实现的区别也蛮大的。
- 3) 处理耗时业务的第一种方式---**handler** 中加入线程池
- 4) 处理耗时业务的第二种方式---**Context** 中添加线程池
- 5) 我们就来分析下两种方式



- **handler 中加入线程池和Context 中添加线程池的源码剖析**

源码剖析

- **说明**

演示两种方式的实现，以及从源码来追踪两种方式执行流程

- **源码剖析**



handler 中加入线程池和Context 中添加线程池的源码剖析



流程图.



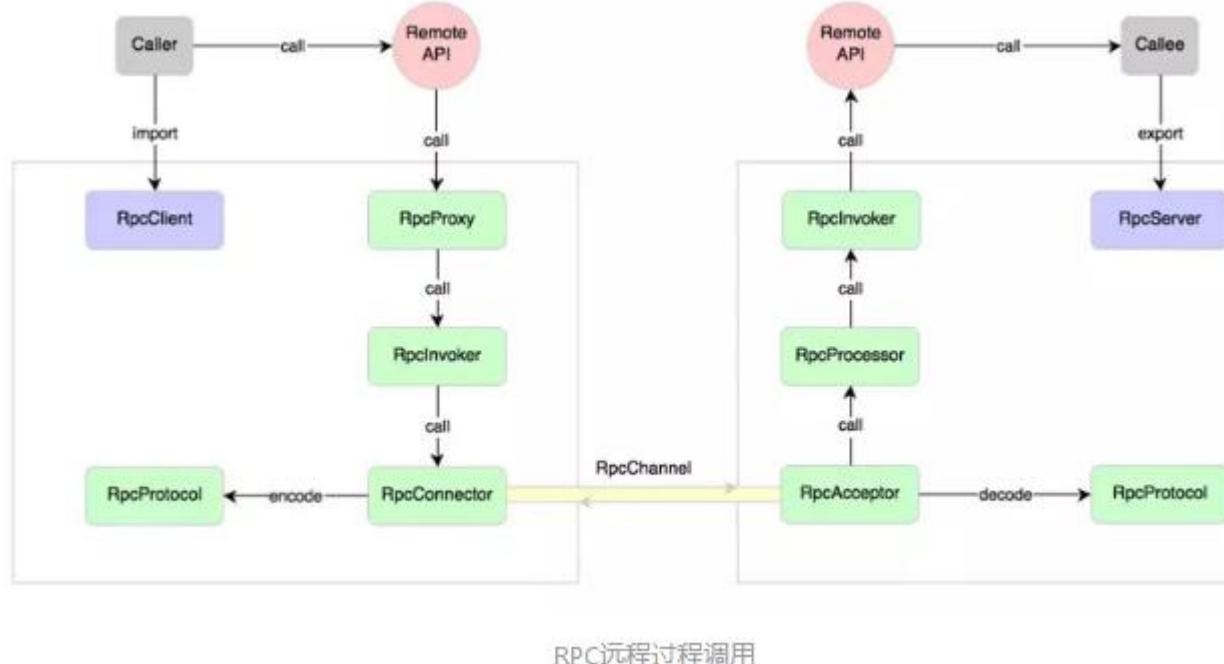
--用Netty自己实现dubbo RPC

- RPC基本介绍

- 1) RPC (Remote Procedure Call)

Procedure Call) — 远程过程调用，是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程

- 2) 两个或多个应用程序都分布在不同的服务器上，它们之间的调用都像是本地方法调用一样(如图)





- **RPC基本介绍**

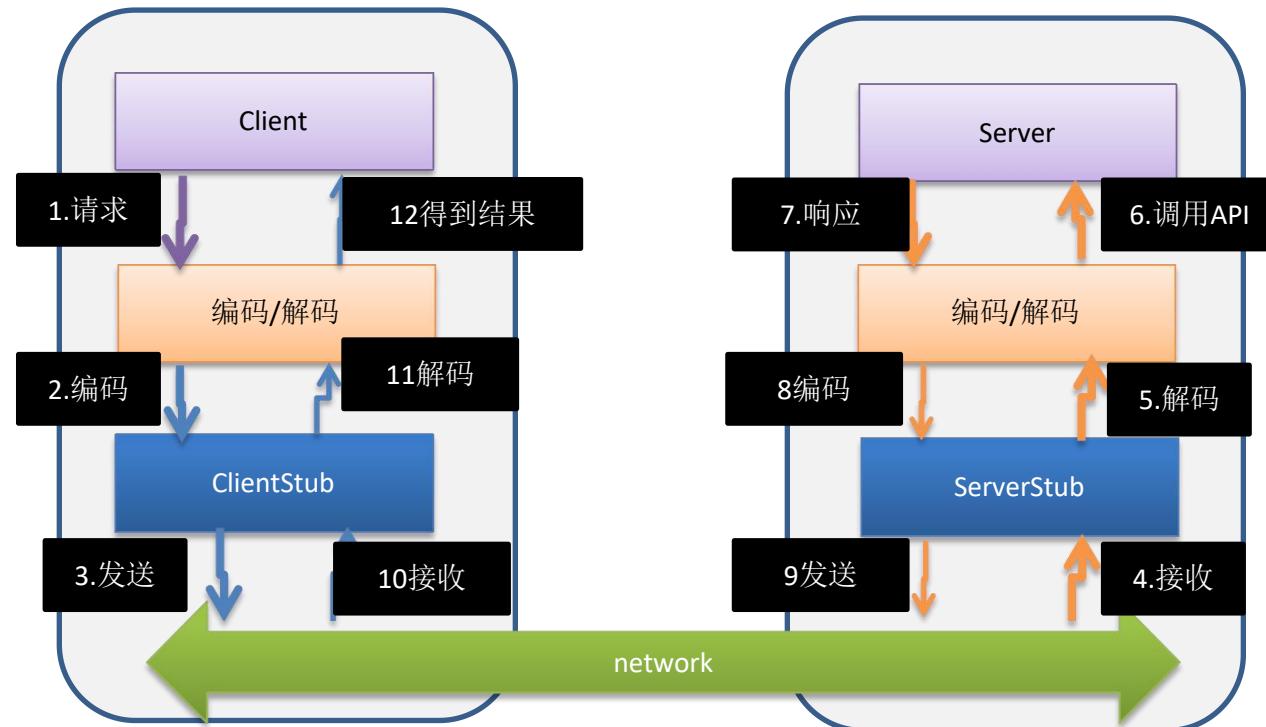
- 3) 常见的 RPC 框架有: 比较知名的如阿里的Dubbo、google的gRPC、Go语言的rpcx、Apache的thrift, Spring 旗下的 Spring Cloud。





• RPC调用流程

RPC调用流程图



术语说明：在RPC中，Client叫服务消费者，Server叫服务提供者



- RPC调用流程

PRC调用流程说明

- 1) **服务消费方(client)**以本地调用方式调用服务
- 2) client stub 接收到调用后负责将方法、参数等封装成能够进行网络传输的消息体
- 3) client stub 将消息进行编码并发送到服务端
- 4) server stub 收到消息后进行解码
- 5) server stub 根据解码结果调用本地的服务
- 6) 本地服务执行并将结果返回给 server stub
- 7) server stub 将返回结果进行编码并发送至消费方
- 8) client stub 接收到消息并进行解码
- 9) **服务消费方(client)得到结果**

小结：RPC 的目标就是将 2-8 这些步骤都封装起来，用户无需关心这些细节，可以像调用本地方法一样即可完成远程服务调用。



- 自己实现 **dubbo RPC**(基于**Netty**)

需求说明

- 1) dubbo 底层使用了 **Netty** 作为网络通讯框架，要求用 **Netty** 实现一个简单的 RPC 框架
- 2) 模仿 **dubbo**，消费者和提供者约定接口和协议，消费者远程调用提供者的服务，提供者返回一个字符串，消费者打印提供者返回的数据。底层网络通信使用 **Netty 4.1.20**

设计说明

- 1) 创建一个接口，定义抽象方法。用于消费者和提供者之间的约定。
- 2) 创建一个提供者，该类需要监听消费者的请求，并按照约定返回数据。
- 3) 创建一个消费者，该类需要透明的调用自己不存在的方法，内部需要使用 **Netty** 请求提供者返回数据



- 自己实现 **dubbo RPC(基于Netty)**

代码实现 **dubbo RPC**

思路分析(图待.)
代码实现



dubborpc.zip



谢谢！ 欢迎收看！