

# 1.Netty

---

最流行的 NIO 框架，由 JBOSS 提供的，整合了FTP,SMTP,HTTP协议

1. API 简单
2. 成熟稳定
3. 社区活跃
4. 经过大规模验证（互联网、大数据、网络游戏、电信通信）  
Elasticsearch、Hadoop 子项目 avro项目、阿里开源框架 Dubbo、使用 Netty

## BIO

优点：模型简单，编码简单缺点：性能瓶颈，请求数和线程数 N:N 关系高并发情况下 ,CPU 切换线程上下文损耗大案例：Tomcat 7之前，都是 BIO，7 之后是 NIO改进：伪 NIO，使用线程池去处理逻辑

## IO 模式

---

同步阻塞：丢衣服->等洗衣机洗完->再去晾衣服同步非阻塞：丢衣服->去做其他事情，定时去看衣服是否洗完->洗完后自己去晾衣服异步非阻塞：丢衣服-> 去做其他事情不管了，衣服洗好会自动晾好，并且通知你晾好了

## 2. 五种 I/O 模型

五种 I/O 模型：阻塞 IO、非阻塞 IO、多路复用 IO、信号驱动 IO、异步 IO，前 4 种是同步 IO，在内核数据 copy 到用户空间时是阻塞的

## 3. 阻塞 IO

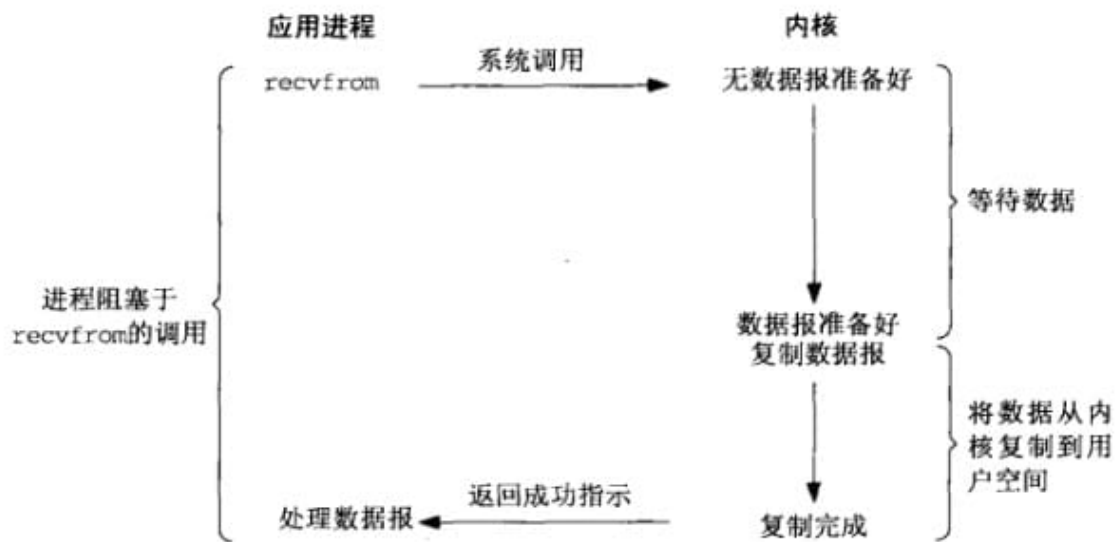


图6-1 阻塞式I/O模型

## 4. 非阻塞 IO

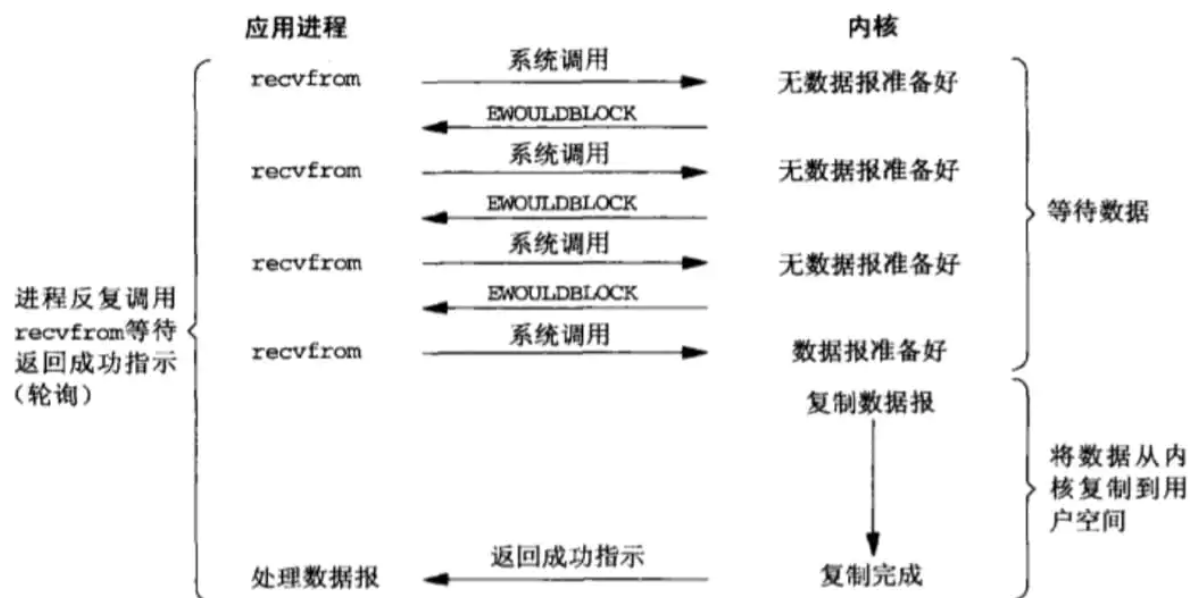


图6-2 非阻塞式I/O模型

## 5. IO 多路复用

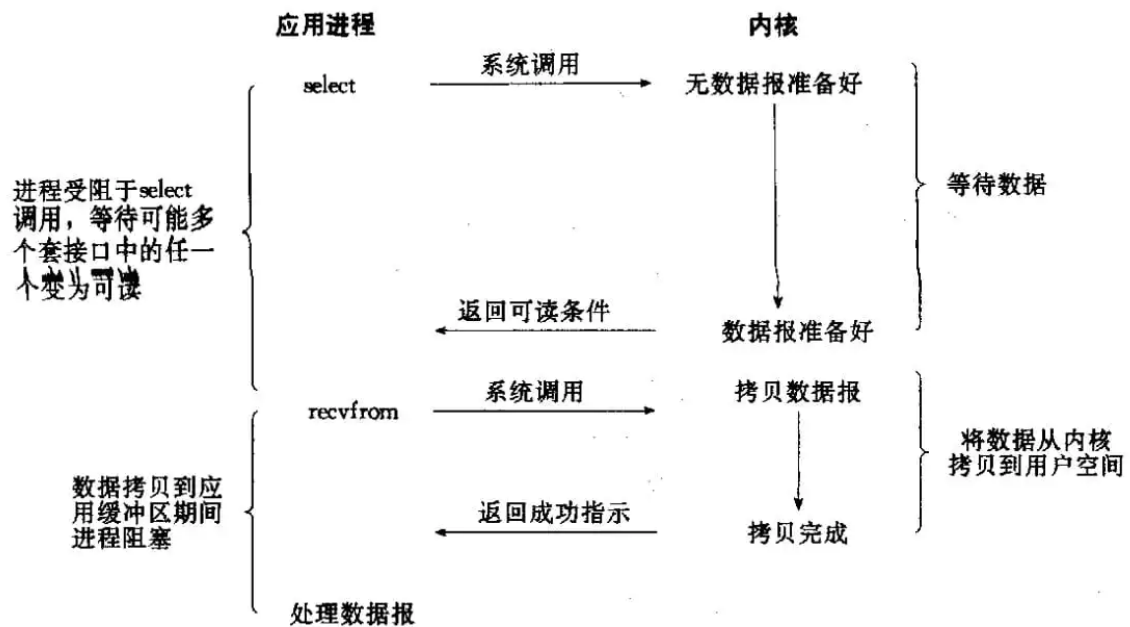


图 6.3 I/O 复用模型

核心：可以同时处理多个 connection，调用系统 select 和 recvfrom 函数每一个 socket 设置为 non-blocking 阻塞是被 select 这个函数 block 而不是 socket 阻塞  
 缺点：连接数不高的情况下，性能不一定比多线程+阻塞 IO 好（多调用一个 select 函数）

## 信号驱动

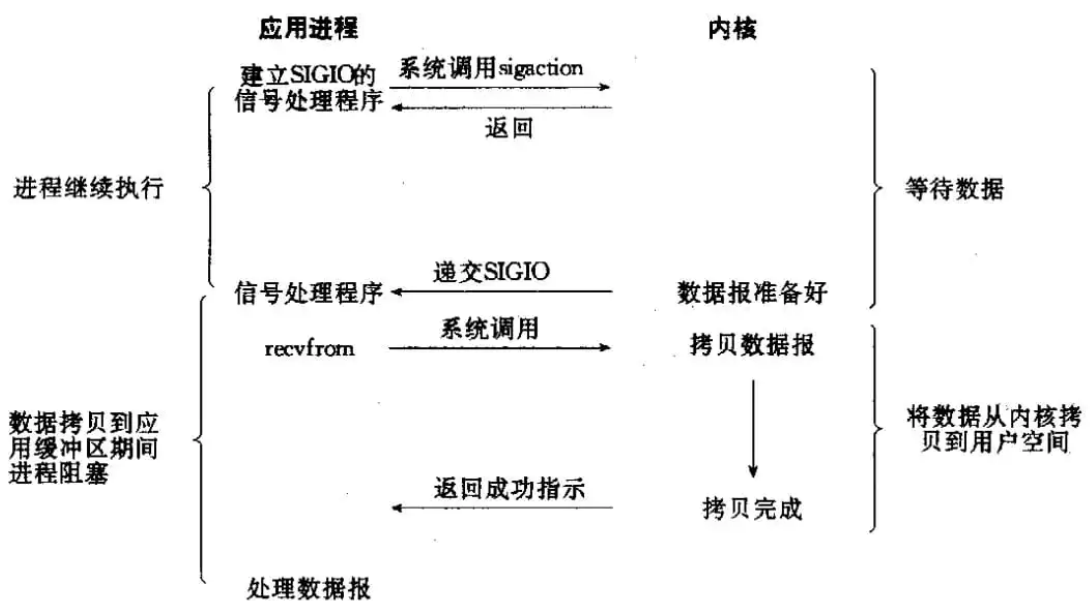
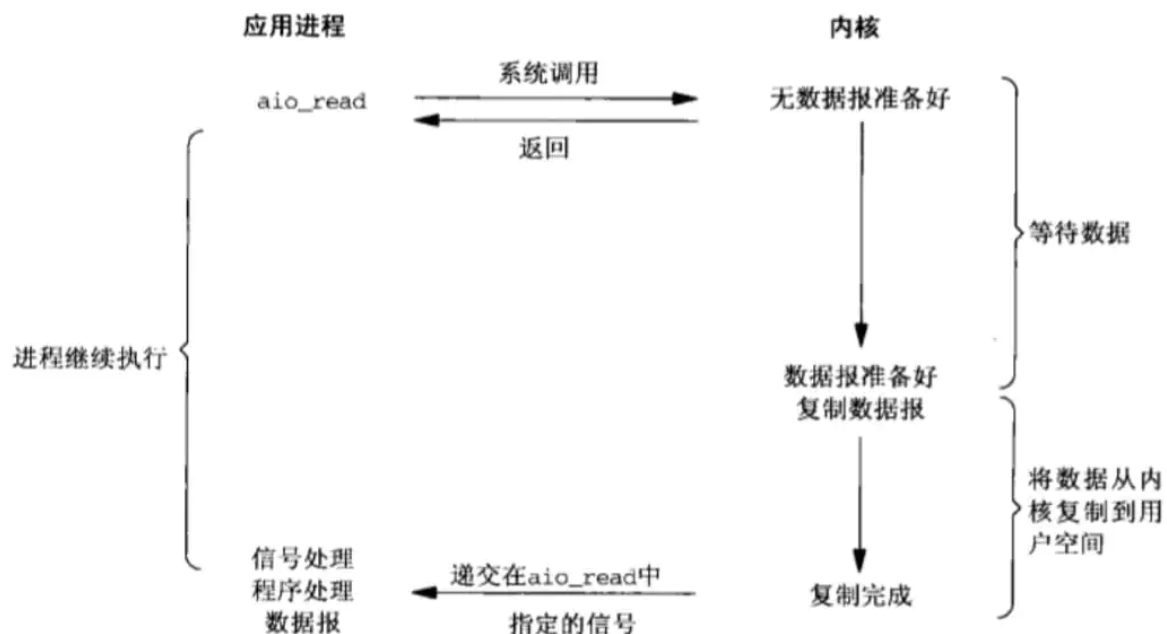


图 6.4 信号驱动 I/O 模型

## 异步 IO

采用 Future-Listener 机制



IO 操作分为 2 步：

1. 发起 IO 请求，等待数据准备
2. 实际的 IO 操作，将数据从内核拷贝到进程中
  - 阻塞 IO、非阻塞 IO 区别在于发起 IO 请求是否被阻塞
  - 同步 IO、异步 IO 在于实际的 IO 读写是否阻塞请求进程
  - 阻塞非阻塞是线程的状态
  - 同步和异步是消息的通知机制
  - 同步需要主动读写数据，异步不需要主动读写数据
  - 同步 IO 和异步 IO 是针对用户应用程序和内核的交互

### 3.IO 多路复用

I/O 是指网络 I/O，多路指多个 TCP 连接，复用指一个或几个线程。简单来说：就是使用一个或者几个线程处理多个 TCP 连接，最大优势是减少系统开销，不必创建过多的线程进程，也不必维护这些线程进程

#### select

文件描述符 `wrtefds`、`readdfs`、`exceptfds` 30w 个连接会阻塞住，等数据可读、可写、出异常、或者超时返回 `select` 函数正常返回后，通过遍历 `fdset` 整个数组才能发现哪些句柄发生了事件，来找到就绪的描述符 `fd`，然后进行对应的 IO 操作，几乎在所有的平台上支持，跨平台支持性好缺点：

1. `select` 采用轮询的方式扫描文件描述符，全部扫描，随着文件描述符 `FD` 数量增多而性能下降。
2. 每次调用 `slect()`，需要把 `fd` 集合从用户态拷贝到内核态，并进行遍历（消息传递都是内核到用户空间）
3. 最大缺陷就是单个进程打开的 `FD` 有限制，默认是 1024

#### poll

基本流程和 `select` 差不多，处理多个描述符也是轮询，根据描述符的状态进行处理，一样需要把 `fd` 集合从用户态拷贝到内核态，并进行遍历。区别是 `poll` 没有最大文件描述符限制（使用链表方式存储 `fd`）

#### epoll

没有描述符限制，用户态拷贝到内核态只需要一次使用事件通知，通过 `epoll_ctl` 注册 `fd`，一旦该 `fd` 就绪，内核就采用 `callback` 机制激活对应的 `fd` 优点：

1. 没有fd限制，所支持的 FD 上限是操作系统的最大文件句柄数（65535），1G 内存大概支持 10W 句柄，支持百万连接的话，16G 内存就可以搞定
2. 效率高，使用回调通知而不是轮询方式，不会随着 FD 数目增加效率下降
3. 通过 callback 机制通知，内核和用户空间 mmap 同一块内存实现

缺点：

编程模型比 select / poll 复杂

linux内核核心函数

4. epoll\_create() 系统启动时，会向linux内核申请一个文件系统，b+树，返回epoll 对象，也是一个 fd
5. epoll\_ctl() 操作epoll对象，在这个对象里面修改添加删除对应的链接fd，绑定一个callback函数
6. epoll\_wait() 判断并完成对应的 IO 操作

例子：100W 个连接，1W 个活跃，在 select ,poll,epoll中怎么样表现

select :不修改宏定义，需要 1000 个进程才能支持 100W 连接

poll:100W连接，遍历都响应不过来，还有空间的拷贝消耗大量的资源

epoll: 不用遍历fd,不用内核空间和用户空间数据的拷贝

如果 100W 个连接中，95W 活跃，则 poll 和 epoll差不多

## 4.Java的i/o

---

1. jdk1.4之前是采用同步阻塞模型（BIO）  
大型服务一般采用 C/C++,因为可以直接操作系统提供的异步 IO（AIO）
2. jdk1.4之后推出NIO,支持非阻塞 IO，jdk1.7 升级推出 NIO2.0，提供了AIO 功能，支持文件和网络套接字的异步 IO

## 5.Netty 线程模型和 Reactor 模式

---

Reactor模式（反应器设计模式），是一种基于事件驱动的设计模式，在事件驱动的应用中，将一个或者多个客户的请求进行分离和调度。在事件驱动的应用中，同步地，有序地处理接受多个服务请求。属于同步非阻塞 IO优点：

1. 响应快，不会因为单个同步而阻塞，虽然 reactor本身是同步的
  2. 编程相对简单，最大程度避免复杂的多线程以及同步问题，避免了多线程、进程切换开销
  3. 可扩展性，可以方便的通过 reactor实例个数充分利用 CPU 资源
- 缺点：
4. 相对复杂，不易于调试
  5. reactor模式需要系统底层的支持。比如java中的selector支持，操作系统select系统调用支持

### Reactor 单线程模型

1. 作为 NIO 服务器，接受客户端 TCP 连接，作为 NIO 客户端，向服务端发起 TCP 连接
2. 服务端读请求数据并响应，客户端写请求并读取响应

场景：

对应小业务则适合，编码简单，对于高负载，高并发不合适。一个 NIO 线程处理太多请求，负载很高，并且响应变慢，导致大量请求超时，万一线程挂了，则不可用

### Reactor 多线程模型

一个 Acceptor线程，一组 NIO 线程，一般是使用自带线程池，包含一个任务队列和多个可用线程场景：可满足大多数场景，当Acceptor需要做负责操作的时候，比如认证等耗时操作，在高并发情况下也会有性能问题

### Reactor 主从线程模型

Acceptor不在是一个线程，而是一组 NIO 线程，IO 线程也是一组 NIO 线程，这样就是 2 个线程池去处理接入和处理 IO场景：满足目前大部分场景，也是 Netty推荐使用的线程模型BossGroup 处理连接的 WorkGroup 处理业务的

## Netty 使用 NIO 而不是 AIO

在 linux系统上，AIO 的底层实现仍然使用 epoll，与 NIO 相同，因此在性能上没有明显的优势Netty 整体架构是 reactor 模型，采用 epoll机制，IO 多路复用，同步非阻塞模型Netty是基于 Java NIO 类库实现的异步通讯框架特点：异步非阻塞，基于事件驱动，性能高，高可靠性，高可定制性。

## 6.Echo服务

---

回显服务，用于调试和检测的服务

## 7.源码剖析

---

### EventLoop和EventLoopGroup

高性能 RPC框架 3 个要素：IO 模型、数据协议（http,brotobuf/thrift）、线程模型EventLoop 好比一个线程，一个 EventLoop可以服务多个Channel,一个Channel只有一个EventLoop，可以创建多个 EventLoop来优化资源的利用，也就是EventLoopGroup一个Channel 一个连接，EventLoopGroup 负责 EventLoop

NIO（单线程处理多个Channels） BIO（一个线程处理一个Channels） 事件：  
accept,connect,read,writeEventLoopGroup 默认创建线程数是 CPU 核数 \* 2

### Bootstrap

1. group:设置线程中模型，Reactor线程模型对比EventLoopGroup
  - 1) 单线程

```
EventLoopGroup g = new NioEventLoopGroup(1);
ServerBootstrap strap = new ServerBootstrap();
strap.group(g)复制代码
```

2) 多线程3) 主从线程

## channel

NioServerSocketChannelOioServerSocketChannelEpollServerSocketChannelKQueueServerSocketChannel

## childHandler


用于对每个通道里面的数据处理

## childOption

作用于被 accept 之后的连接

## option

作用于每个新建的 channel，设置 TCP 连接中的一些参数

- ChannelOption.SO\_BACKLOG  
存放已完成三次握手的请求的等待队列的最大长度  
Linux 服务器 TCP 连接底层知识：  
syn queue: 半连接队列，洪水攻击（伪造 IP 海量发送第一个握手包），tcp\_max\_syn\_backlog  
(修改半连接 vi /etc/sysctl.conf)  
accept queue: 全连接队列 net.core.somaxconn 当前机器最大连接数  
系统默认的 somaxconn 参数要足够大，如果 backlog 比 somaxconn 大，则会优先用后者  

- ChannelOption.TCP\_NODELAY  
默认是 false，要求高实时性，有数据时马上发送，就将该值改为 true 关闭 Nagle 算法（Nagle 算法会积累一定大小后再发送，为了减少发送次数）Nagle 算法只允许一个未被 ACK 的包存在于网络  
(tcp\_synack\_retries = 0 加快回收半连接，如果收不到第三个握手包 ACK，不进行重试，默认值是 5，每次等待 30s，半连接会 hold 住大约 180s, tcp\_syn\_retries 默认值是 5，客户端没收到 SYN+ACK 包，客户端也会重试 5 次发送 SYN 包)

## childOption

作用于被 accept 之后的链接

## childHandler

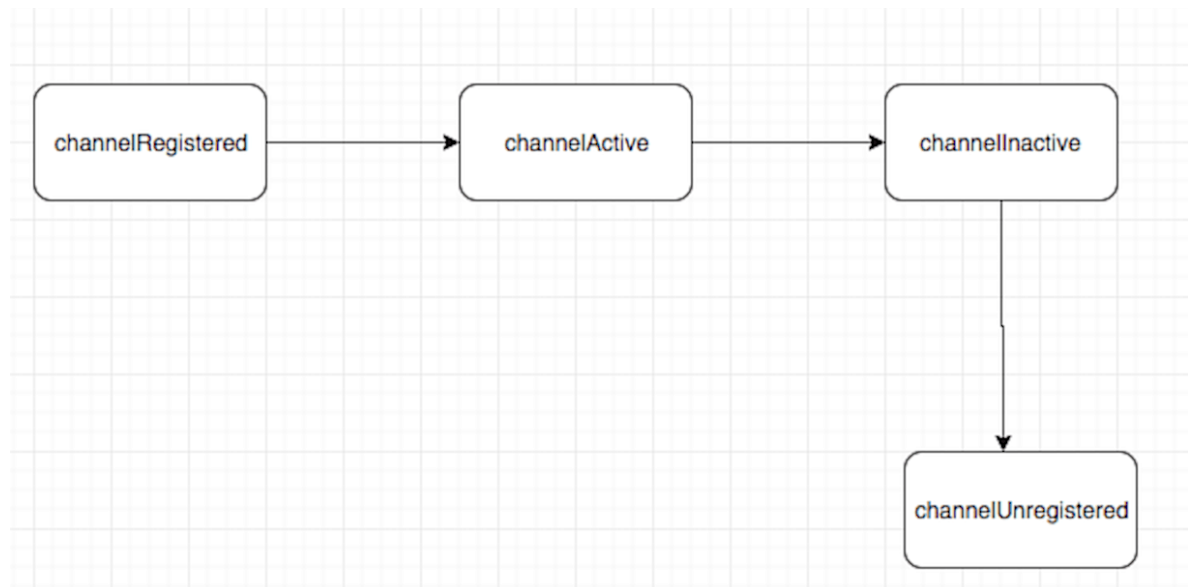
用于对每个通道里面的数据处理

## Channel

- Channel  
客户端和服务端建立的一个连接通道
- ChannelHandler  
负责 Channel 的逻辑处理

- ChannelPipeline  
负责管理 ChannelHandler的有序容器

一个Channel包含一个ChannelPipeline，所有 ChannelHandler都会顺序加入到ChannelPipeline中。  
Channel当状态出现变化，对触发对应的事件

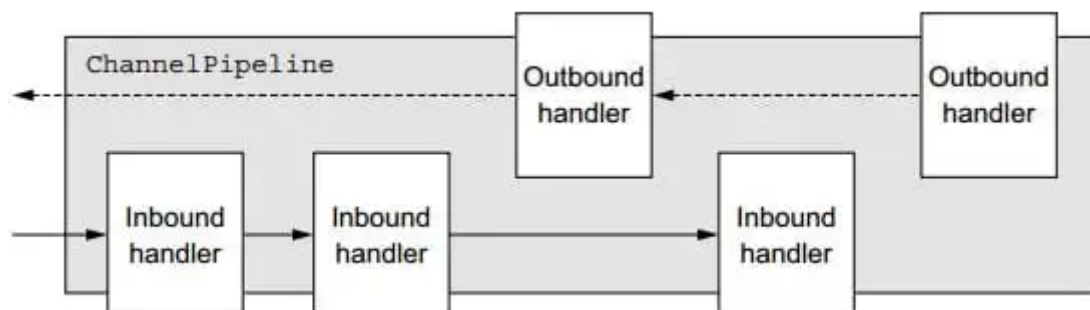


状态：

- channelRegistered  
channel注册到一个EventLoop，和Selector绑定
- channelUnRegistered  
channel已创建，但是未注册到一个EventLoop里面，也就是没有和Selector绑定
- channelActive  
变为活跃状态，连接到了远程主机，可以接受和发送数据
- channelInactive  
channel处于非活跃状态，没有连接到远程主机

## ChannelHandler和ChannelPipeline

ChannelHandler生命周期： handlerAdded:当ChannelHandler添加到ChannelPipeline调用  
handlerRemoved:当ChannelHandler从ChannelPipeline移除时调用exceptionCaught:执行抛出异常  
时调用ChannelHandler有 2 个子接口：



ChannelInboundHandler（入站）： 处理输入数据和Channel状态类型改变，适配器

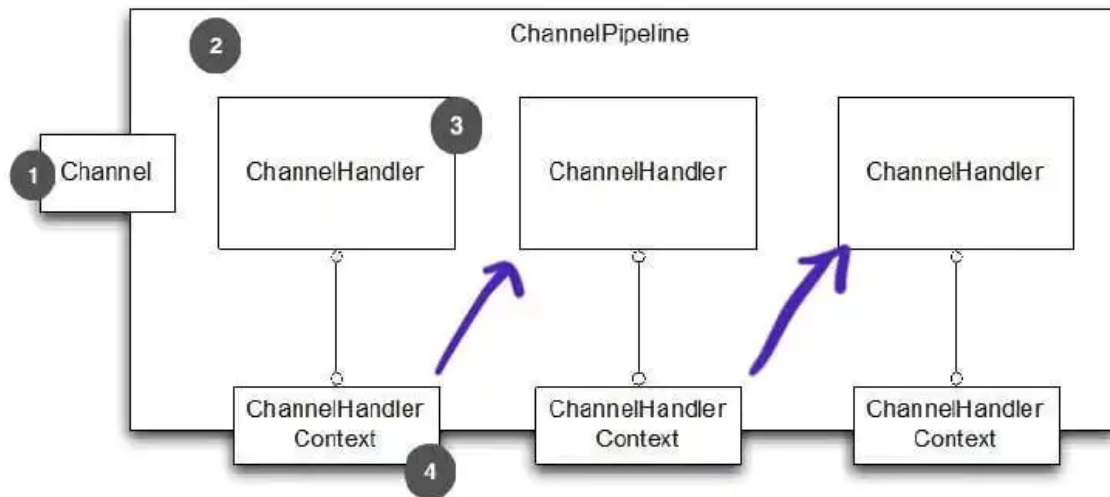
ChannelInboundHandlerAdapter（适配器设计模式），常用

SimpleChannelInboundHandlerChannelOutboundHandler（出站）： 处理输出数据，适配器  
Channel

ChannelPipeline:好比厂里的流水线一样，可以在上面添加多个ChannelHandler，也可以看成是一串  
ChannelHandler 实例，拦截穿过Channel的输入输出 event，ChannelPipeline实现了拦截器的一种高级形式，使得用户可以对事件的处理以及ChannelHandler之间交互获得完全的控制权



## ChannelHandlerContext



1. channelHandlerContext 是连接 ChannelHandler 和 ChannelPipeline 的桥梁  
ChannelHandlerContext 部分方法是和 Channel以及ChannelPipeline重合，好比调用 write方法  
Channel,ChannelPipeline,ChannelHandlerContext都可以调用写方法，前 2 者会在整个管道流  
里传播，而 ChannelHandlerContext只会在后续的 Handler里传播
2. AbstractChannelHandlerContext  
双向链表结构,next/prev 后继、前驱节点
3. DefaultChannelHandlerContext 是实现类，但是大部分都是父类完成，整个只是简单的实现一些  
方法，主要就是判断 Handler的类型  
fire调用下一个 handler，不fire就不调用

## Handler执行顺序

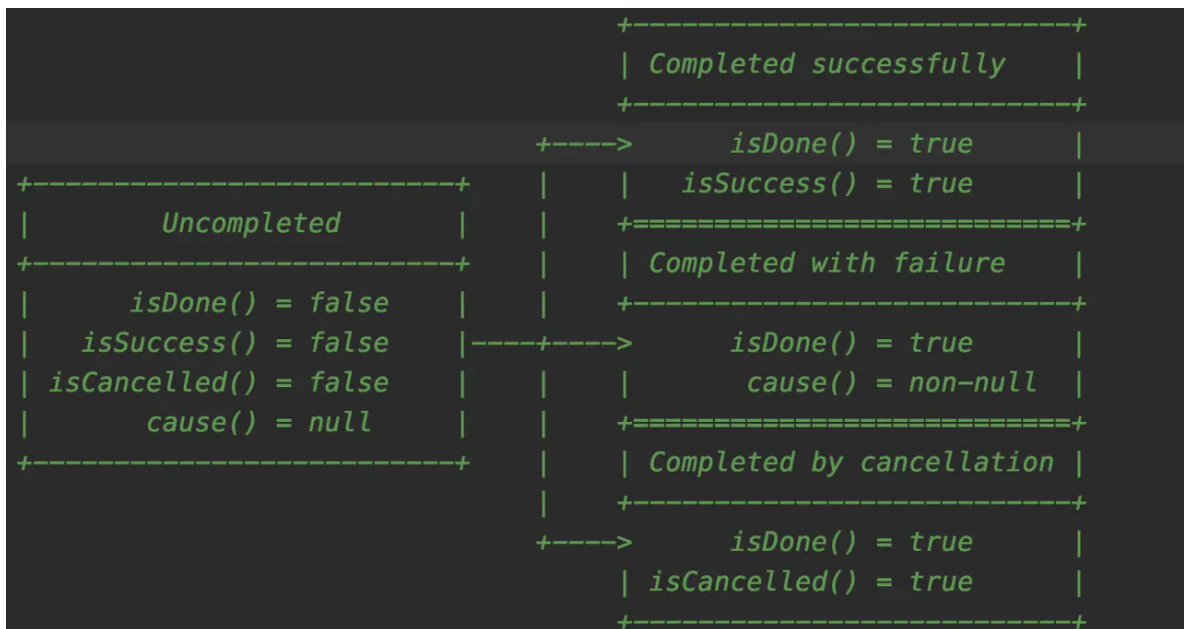
InboundHandler顺序执行，OutboundHandler逆序执行

```
channel.pipeline().addLast(new OutboundHandler1());  
channel.pipeline().addLast(new OutboundHandler2());  
channel.pipeline().addLast(new InboundHandler1());  
channel.pipeline().addLast(new InboundHandler2());复制代码
```

InboundHandler1 InboundHandler2 OutboundHandler2 OutboundHandler1InboundHandler1之  
间通过 fireChannelRead()方法调用InboundHandler通过ctx.write(msg),传递到  
OutboundHandlerctx.write(msg)传递消息，Inbound需要放在结尾，在 outbound之后，不然  
outboundHandler不会执行，使用 channel.write(msg),或者 pipeline.write(msg),就不用考虑（传播  
机制）客户端：发起请求再接受请求，先 outbound再inbound服务端：先接受请求再发送请求，先  
inbound再outbound

## ChannelFuture

netty中所有 I/O 操作都是异步的，意味着任何 I/O 调用都会立即返回，而ChannelFuture会提供有关的信息 I/O 操作的结果或状态未完成：当 I/O 操作开始时，将创建一个新的对象，新的最初是未完成的，它既没有成功，也没有被取消，因为 I/O 操作尚未完成。已完成：当 I/O 操作完成，不管是成功、失败还是取消，Future都是标记为已完成的，失败的时候也有具体的信息，例如原因失败，但请注意，即使失败和取消属于完成状态。



注意：不要在 IO 线程内调用Future对象的sync和await方法，不能在 channelhandler中调用 sync 和 await

## ChannelPromise

继承 ChannelFuture，进一步扩展用于设置 IO 操作的结果

## 编解码

java序列化/反序列化，url编解码,base64编解码java自带序列化的缺点：

1. 无法跨语言
2. 序列化后的码流太大，数据包太大
3. 序列化和反序列化性能比较差

业界其他编解码框架：PB，Thrift，Marshalling，Kyro

Netty里面的编解码：

- 解码器：主要负责处理入站 InboundHandler
  - 编码器：主要负责处理出站 OutBoundHandler
- Netty默认编解码器，也支持自定义编解码器  
Encoder（编码器），Decoder（解码器），Codec（编解码器）

## Netty解码器 Decoder

Decoder对应 ChannelInboundHandler，主要就是字节数组转换成消息对象方法：

- decode :常用
  - decodeLast: 用于最后的几个字节处理，也就是 cahnnel 关闭的时候，产生的最后一个消息
- 解码器：
- ByteToMessageDecoder  
用于将字节转为消息，需要检查缓冲区是否有足够的字节
  - ReplayingDecoder  
继承ByteToMessageDecoder，不需要检查缓冲区是否有足够多的数据，速度略慢于 ByteToMessageDecoder
  - MessageToMessageDecoder  
用于将一种消息解码到另外一种消息（例如 POJO 到 POJO）
- 常用的解码器：（主要解决 TCP 底层的粘包和拆包问题）

- DelimiterBasedFrameDecoder: 执行消息分隔符的解码器
- LineBasedFrameDecoder: 以换行符为结束标志的解码器
- FixedLengthFrameDecoder: 固定长度的解码器
- LengthFieldBasedFrameDecoder: message = header + body,基于长度解码的通用解码器
- StringDecoder:文本解码器，将接收到的消息转为字符串，一般会与上面的几种进行组合，然后再后面加业务的 handler

## Netty 编码器 Encoder

Encoder 对应就是 ChannelOutboundHandler，消息对象转换成字节数组编码器：

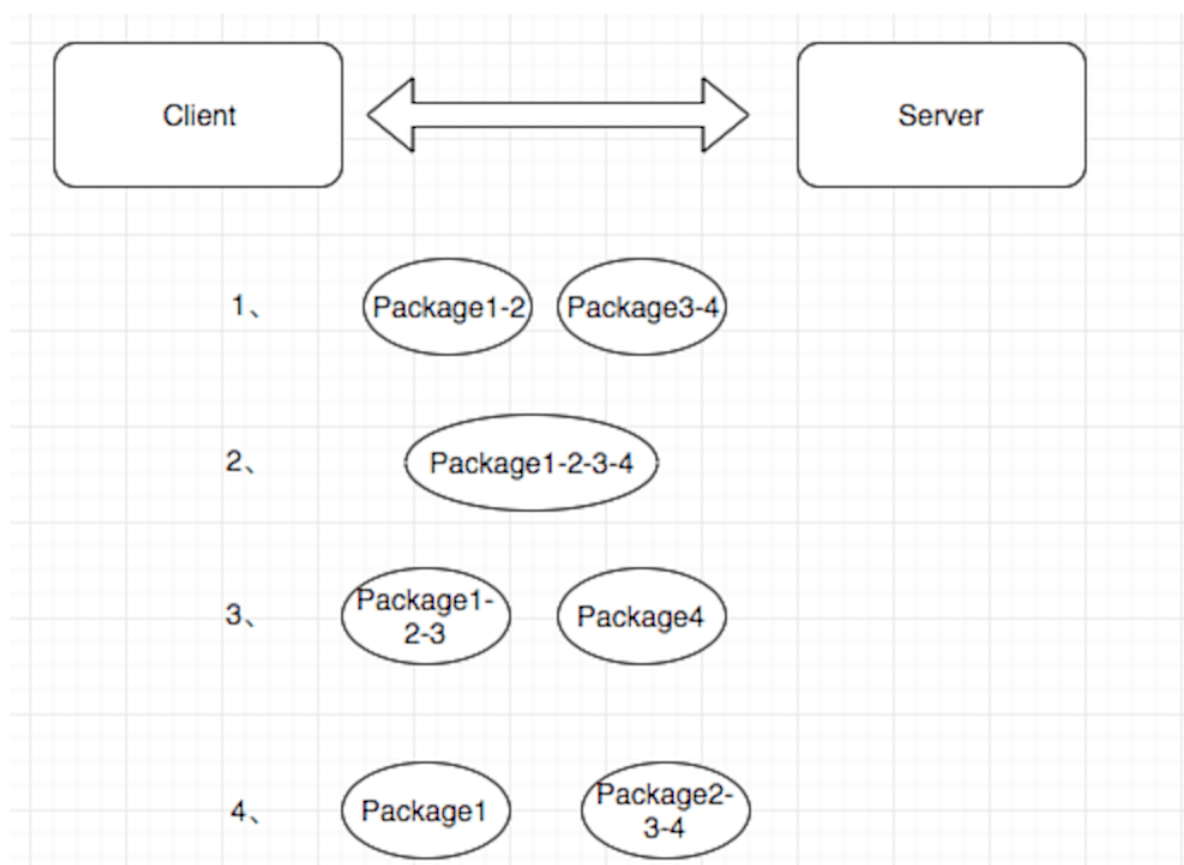
- MessageToByteEncoder  
消息转为字节数组，调用 write方法，会先判断当前编码器是否支持需要发送的消息类型，如果不支持，则透传
- MessageToMessageEncoder 从一种消息编码为另外一种消息

## Netty 组合编解码器 Codec

优点：成对出现，编解码都是在一个类里完成缺点：耦合，扩展性不佳

- ByteToMessageCodec
- MessageToMessageCode

## TCP 粘包，拆包



TCP 拆包：一个完整的包可能被 TCP 拆分成多个包进行发送TCP 粘包：把多个小的包封装成一个大的数据包发送,client发送的若干数据包，server接收时粘在一个包发送方和接收方都可能出现这个原因发送方的原因：TCP 默认会使用 Nagle算法接收方的原因：TCP 接收到数据放置缓存中，应用程序从缓存中读取比较慢UDP 无粘包、拆包问题，有边界协议

## TCP 半包读写解决方案

发送方：关闭 Nagle 算法接收方：TCP 是无界的数据流，并没有处理粘包现象的机制，且协议本身无法避免粘包，半包读写的发生需要在应用层进行处理应用层解决半包读写方法：

1. 设置定长消息 (10 个字符)  
abcdefgh11abcdefgh11abcdefgh11
2. 设置消息边界 (?切割)  
dfdsfdfsdf?dsfdfsdf\$dsfdfsdf
3. 使用带消息头的协议，消息头存储消息开始标识及消息的长度信息  
header + body

## Netty 自带解决 TCP 半包读写方案

- DelimiterBasedFrameDecoder:指定消息分隔符的解码器
- LineBasedFrameDecoder:以换行符为结束标志的解码器
- FixedLengthFrameDecoder:固定长度解码器
- LengthFieldBasedFrameDecoder : message = header + body，基于长度解码的通用解码器

## 实战半包读写

LineBasedFrameDecoder:以换行符为结束标志的解码器StringDecoder 解码器将对象转成字符串

## 自定义分隔符解决 TCP 读写问题

DelimiterBasedFrameDecodermaxLength: 表示一行最大的长度，超过长度依然没检测自定义分隔符，抛出TooLongFrameExceptionfailFast: 如果为true,则超过 maxLength后立即抛出 TooLongFrameException,不进行继续解码，如果为 false,则等到完整消息被解码后，再抛出 TooLongFrameExceptionstripDelimiter:解码后的消息是否去除分隔符delimiters:分隔符，ByteBuf类型

## 自定义长度半包读写器 LengthFieldBasedFrameDecoder

maxFrameLength 数据包最大长度lengthFieldOffset 长度字段的偏移量，长度字段开始的地方（跳过指定长度个字节之后的才是消息体字段）lengthFieldLength 长度字段占的字节数，帧数据长度的字段本身的长度lengthAdjustment一般 Header + Body ,添加到长度字段的补偿值，如果为负数，开发人员认为这个Header的长度字段是整个消息包的长度，，则Netty应该减去对应的数字initialBytesToStrip 从解码帧中第一次去除的字节数，获取完一个完整的数据包之后，忽略前面的指定位数的长度字节，应用解码器拿到的就是不带长度域的数据包

## ByteBuf



字节容器，

- JDK 中原生 ByteBuffer  
读和写公用一个索引，每次换操作都需要Flip()  
扩容麻烦，而且扩容后容易造成浪费
- Netty ByteBuf  
读写使用不同的索引，所以操作便捷  
自动扩容，便捷

## ByteBuf 创建方法与常见的模式

ByteBuf:传递字节数据的容器ByteBuf的创建方法:

1. ByteBufAllocator  
Netty 4.x之后默认使用池化 (PooledByteBufAllocator) 提高性能，最大程度减少内存碎片  
非池化: UnPooledByteBufAllocator 每次返回一个新的实例
2. Unpooled:提供静态方法创建未池化的ByteBuf，可以创建堆内存和直接内存缓冲区

ByteBuf使用模式:

1. 堆缓存区  
优点: heap buffer 存储在 jvm的堆空间中，快速的分配和释放  
缺点: 每次使用前会拷贝到直接缓存区 (堆外内存)
2. 直接缓存区  
Direct buffer  
优点: 不用占用 JVM 的堆内存，存储在堆外内存  
缺点: 内存的分配和释放，比在堆缓存区更复杂
3. 复合缓冲区  
创建多个不同的 ByteBuf,然后放在一起，但是只是一个视图

选择: 大量 IO 数据读写，用直接缓存区，业务消息编解码用堆缓存区

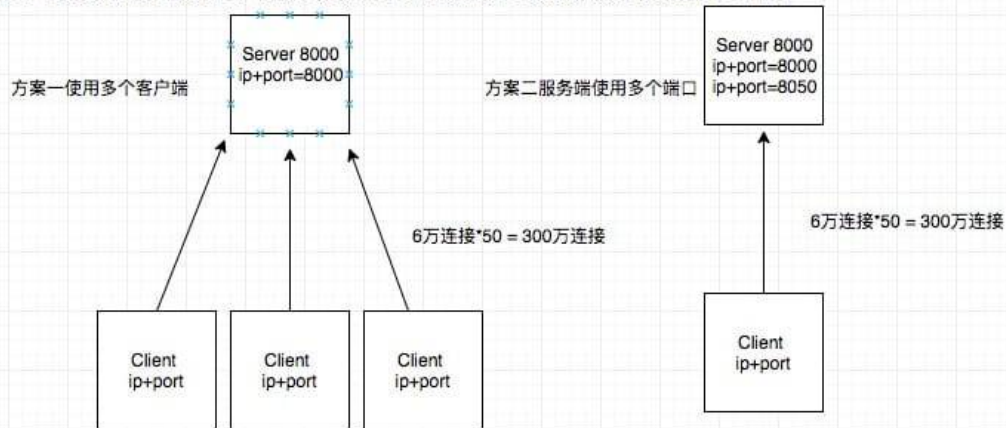
## Netty 设计模式

Builder 构造器模式: ServerBootstrap责任链设计模式: pipeline的事件传播工厂模式: 创建 channel  
适配器模式: HandlerAdapter

## Netty 单机百万实战

1. 网络 IO模型
2. Linux文件描述符  
单进程文件描述符 (句柄数)，每个进程都有最大的文件描述符限制  
全局文件句柄数，也有默认值，不同系统版本会不一样
3. 如何确定唯一 TCP 连接  
TCP 四元组: 源 IP，源端口，目标 IP，目标端口  
服务端端口范围 (1024~65535)

65535个端口，由于其他应用占用了端口，而且系统默认1024以下的端口是不被使用，大致可以数64000个端口



65545

优化:

4. `sudo vim /etc/security/limits.conf` 修改局部 fd数目，修改后要重启，`ulimit -n` 查看当前这个用户每个进程最大 FD 数

```
root soft nofile 1000000
root hard nofile 1000000
* soft nofile 1000000
* hard nofile 1000000复制代码
```

1. `sudo vim /etc/sysctl.conf` 修改全局 fd 数目

```
fs.file-max=1000000复制代码
```

`sysctl -p` 重启生效参数`cat /proc/sys/fs/file-max` 查看全局fd数目

1. 重启生效 `reboot`

`-Xms5g -Xmx5g -XX:NewSize=3g -XX:MaxNewSize=3g`

## 数据链路

浏览器同域名下资源加载有并发数限制，建议不同资源用不同域名输入域名-》浏览器内核调度-》本地 DNS 解析-》远程 DNS解析-》IP-》路由多层跳转-》目的服务器-》服务器内核-》应用程序