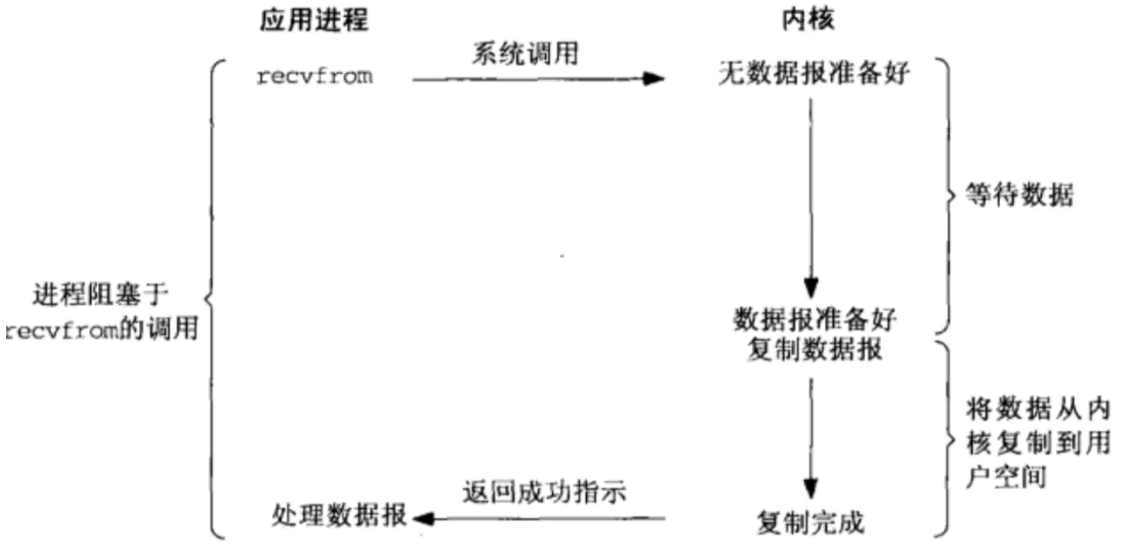
Netty

> 同步异步，阻塞非阻塞
>
> **1.同步与异步** 关注结果/响应是如何通知给调用方的
> 同步和异步关注的是**消息通信机制** (synchronous communication/ asynchronous communication)
> 所谓同步，就是在发出一个*调用*时，在没有得到结果之前，该*调用*就不返回。但是一旦调用返回，就得到返回值了。
> 换句话说，就是由*调用者*主动等待这个*调用*的结果。
>
> 而异步则是相反，***调用*在发出之后，这个调用就直接返回了，所以没有返回结果**。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在*调用*发出后，*被调用者*通过状态、通知来通知调用者，或通过回调函数处理这个调用。
>
> 同步与异步的最大区别在于想要获取执行结果，同步的话需要轮询查询执行结果，而异步的话通过事件回调机制在执行完成之后主动通知给调用者。
>
> **2.阻塞与非阻塞** 关注的是调用者在执行调用的过程中能否执行其他的动作
>
> 阻塞和非阻塞关注的是**程序在等待调用结果（消息，返回值）时的状态.**
>
> 阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
> 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。
>
> 以烧水为例：
>
> 同步就是烧开水，要自己来看开没开；异步就是水开了，然后水壶响了通知你水开了。阻塞是烧开水的过程中，你不能干其他事情（即你被阻塞住了）；非阻塞是烧开水的过程里可以干其他事情。同步与异步说的是你获得水开了的方式不同。阻塞与非阻塞说的是你得到结果之前能不能干其他事情。两组概念描述的是不同的内容。
>
> 阻塞与非阻塞的最大区别在于线程是否被卡在一个地方，必须等待该线程处理完之后才能去做其他事情。

IO模型(https://juejin.im/post/6844903728718462990)

对于一次IO访问（以read为例），数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。

> ## 普通输入操作包含的步骤
>
> - 等待数据准备好 此阶段由内核完成
> - 从内核向进程复制数据 此阶段从内核态切换回用户态
>
> ## 阻塞式IO

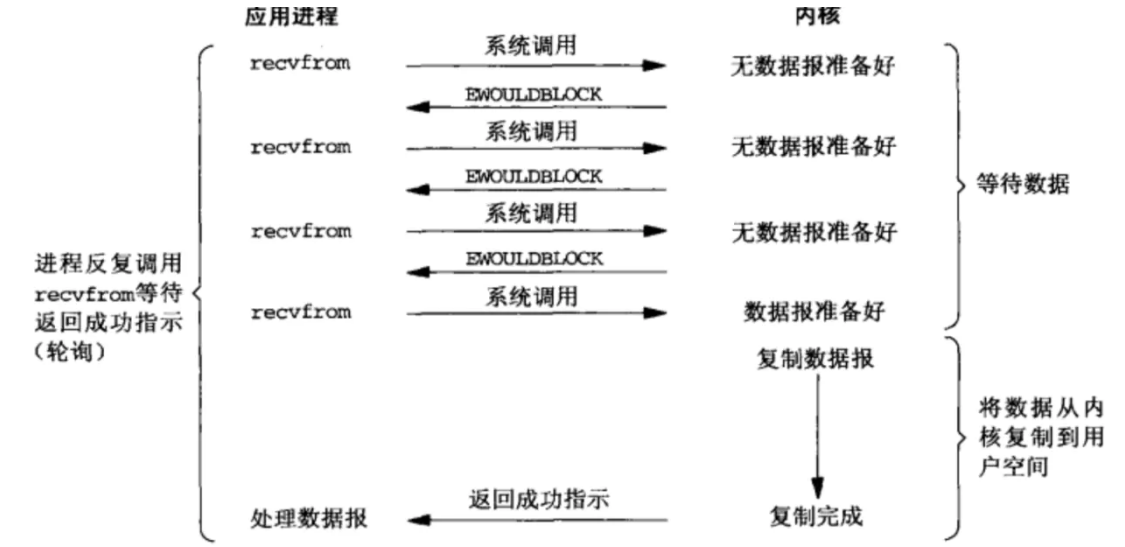


(A拿着一支鱼竿在河边钓鱼，并且一直在鱼竿前等，在等的时候不做其他的事情，十分专心。只有鱼上钩的时，才结束掉等的动作，把鱼钓上来。)

使用系统调用，并一直阻塞直到内核将数据准备好，之后再由内核缓冲区复制到用户态，在等待内核准备的这段时间什么也干不了

函数调用期间，一直被阻塞，直到数据准备好且从内核复制到用户程序才返回，这种IO模型为阻塞式IO

非阻塞式IO



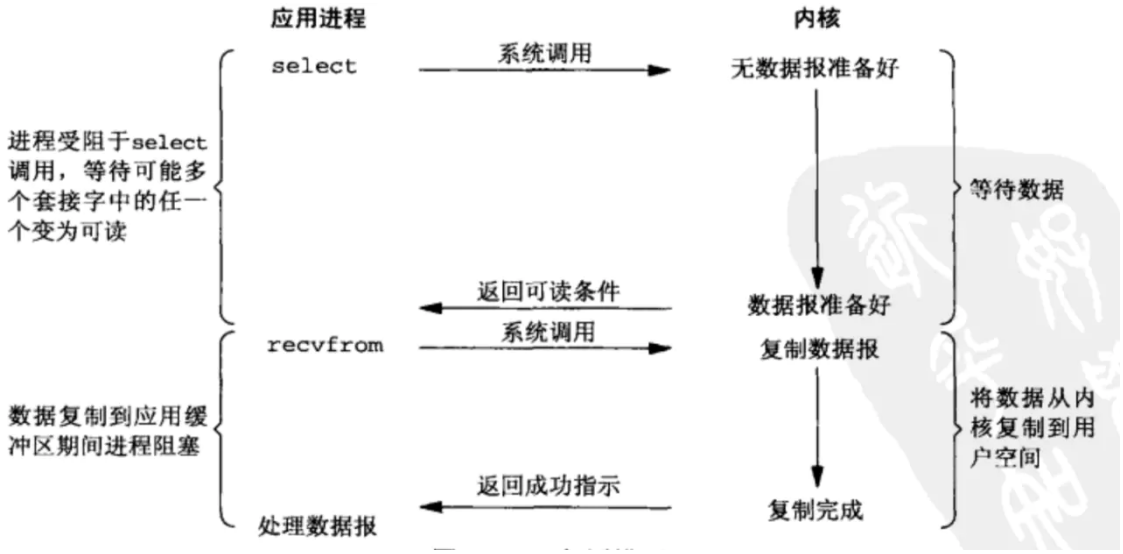
(B也在河边钓鱼，但是B不想将自己的所有时间都花费在钓鱼上，在等鱼上钩这个时间段中，B也在做其他的事情（一会看看书，一会读读报纸，一会又去看其他人的钓鱼等），但B在做这些事情的时候，每隔一个固定的时间检查鱼是否上钩。一旦检查到有鱼上钩，就停下手中的事情，把鱼钓上来。)

内核在没有准备好数据的时候会返回错误码，而调用程序不会休眠，而是不断轮询询问内核数据是否准备好

如果数据没有准备好，不像阻塞式IO那样一直被阻塞，而是返回一个错误码。数据准备好时，函数成功返回。

非阻塞式IO的轮询会耗费大量cpu，通常在专门提供某一功能的系统中才会使用。

IO多路复用



(C同样也在河边钓鱼，但是C生活水平比较好，C拿了很多的鱼竿，一次性有很多鱼竿在等，C不断的查看每个鱼竿是否有鱼上钩。增加了效率，减少了等待的时间。一旦检查到有鱼上钩，就把鱼钓上来。)

类似与非阻塞，**只不过轮询不是由用户线程去执行，而是由内核去轮询**，内核监听程序监听到数据准备好后，调用内核函数复制数据到用户态

select这个系统调用，充当代理类的角色，不断轮询注册到它这里的所有需要IO的文件描述符，有结果时，把结果告诉被代理的recvfrom函数，它本尊再亲自出马去拿数据

IO多路复用至少有两次系统调用，如果只有一个代理对象，性能上是不如前面的IO模型的，但是由于它可以同时监听很多套接字，所以性能比前两者高

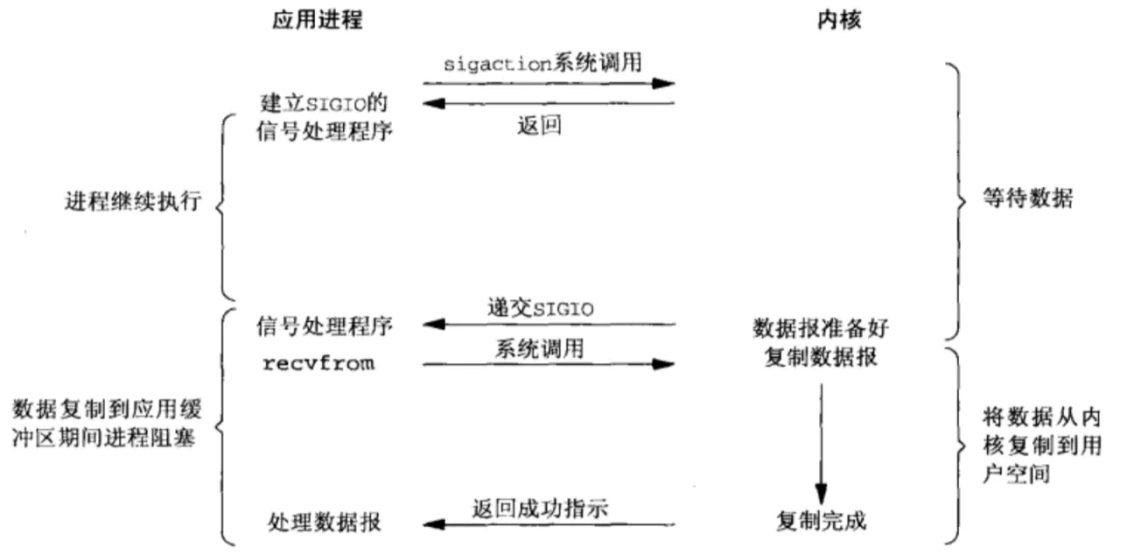
多路复用：

select：线性扫描所有监听的文件描述符，不管他们是不是活跃的。有最大数量限制

poll：同select，不过数据结构不同，需要分配一个pollfd结构数组，维护在内核中。它没有大小限制，不过需要很多复制操作

epoll：用于代替poll和select，没有大小限制。使用一个文件描述符管理多个文件描述符，使用红黑树存储。同时用事件驱动代替了轮询。epoll_ctl中注册的文件描述符在事件触发的时候会通过回调机制激活该文件描述符。epoll_wait便会收到通知。

信号驱动式IO

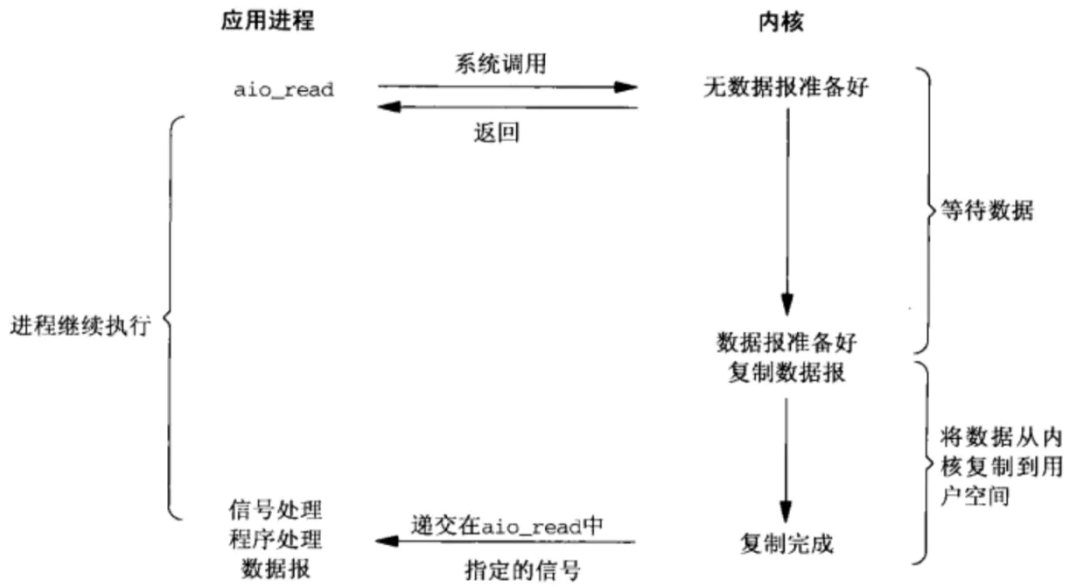


(D也在河边钓鱼，但与A、B, C不同的是，D比较聪明，他给鱼竿上挂一个铃铛，当有鱼上钩的时候，这个铃铛就会被碰响，D就会将鱼钓上来。)

使用信号，内核在数据准备就绪时通过信号来进行通知

数据准备好时，内核会发送SIGIO信号，收到信号后开始进行io操作

异步IO



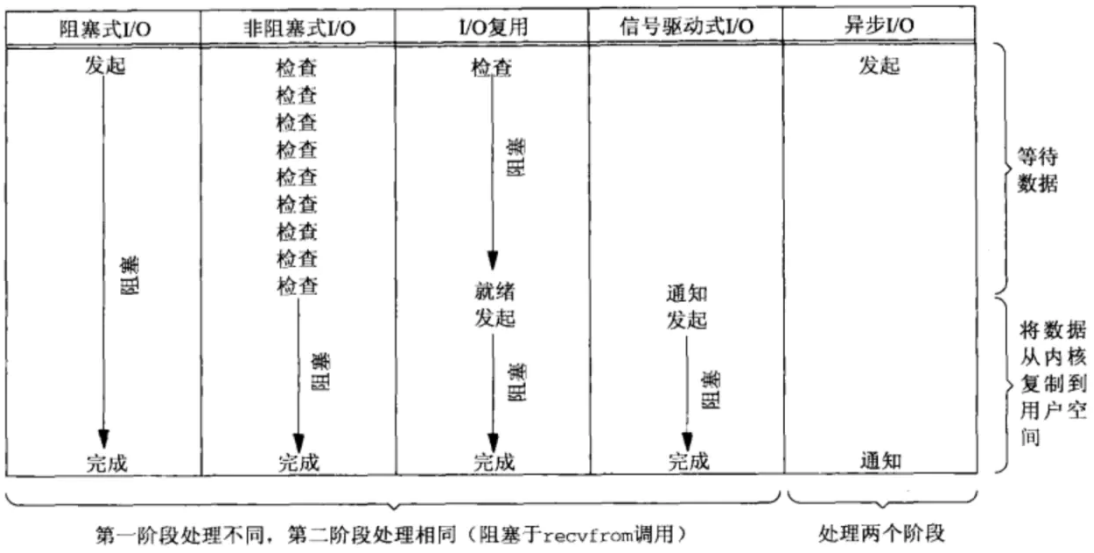
(E也想钓鱼，但E有事情，于是他雇来了F，让F帮他等待鱼上钩，一旦有鱼上钩，F就打电话给E，E就会将鱼钓上去。)

异步IO依赖信号处理程序来进行通知

不过异步IO与前面IO模型不同的是：前面的都是数据准备阶段的阻塞与非阻塞，异步IO模型通知的是IO操作已经完成，而不是数据准备完成（复制操作已经完成）

异步IO才是真正的非阻塞，主进程只负责做自己的事情，等IO操作完成(数据成功从内核缓存区复制到应用程序缓冲区)时通过回调函数对数据进行处理

各种IO模型对比

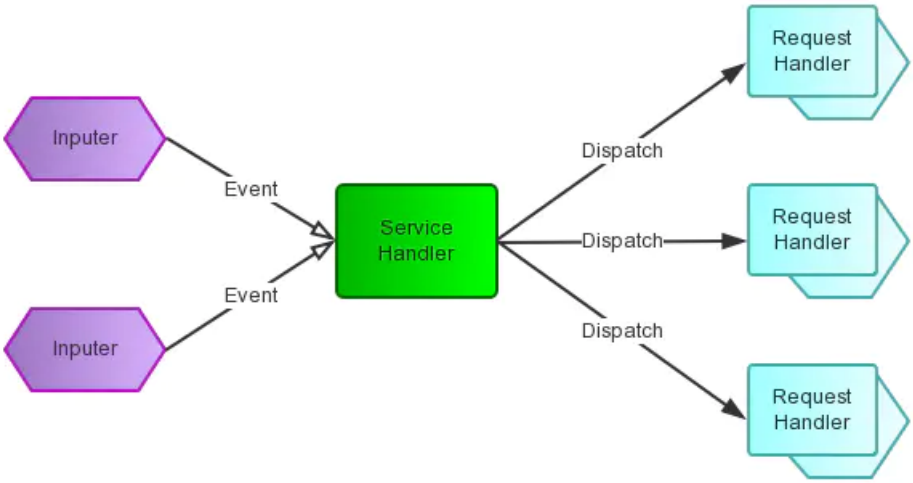


除了异步IO模型之外的其他四种模型，主要区别在第一阶段（是否阻塞），第二阶段都是将数据从内核复制到用户空间，这个复制的过程会阻塞线程；

> 除了异步IO模型之外的其他四种模型都是同步IO，在等待数据复制的时候是不能去做其他事情的，必须同步等待数据复制完成。而异步IO可以在复制期间做别的事情，因为复制完成之后会通知到线程。

Reactor模式(https://juejin.im/post/6844903682509635598，https://www.cnblogs.com/winner-0715/p/8733787.html)

> - 一种事件驱动模型
> - 处理多个输入
> - 采用多路复用将事件分发给相应的Handler处理



Reactor实际上采用了**分而治之**和**事件驱动**的思想：

分而治之： 一个连接里完整的网络处理过程一般分为 accept，read，decode，process，encode，send这几步。而Reactor模式将每个步骤映射为 一个Task，服务端线程执行的最小逻辑单元不再是一个完整的网络请求，而是 Task，且采用非阻塞方式执行。

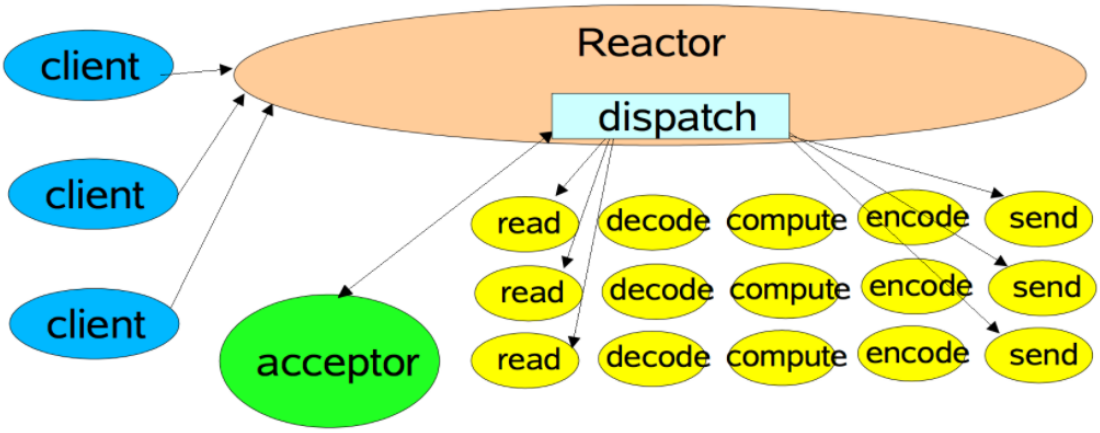
事件驱动： 每个Task 对应特定的网络事件，当Task 准备就绪时，Reactor 收到对应的网络事件通知，并将Task 分发给绑定了对应网络事件的 Handler 执行。

Reactor模式就是 指一个或多个事件输入同时传递给服务处理器(Reactor)，服务处理器负责监听各事件的状态，当任意一个事件准备就绪时，服务处理器收到该事件通知，并将事件发送给绑定了该对应网络事件的事件处理器(Handler)执行

Reactor模式也叫做Dispatcher模式，即 I/O多路复用统一监听事件，收到事件后再分发(Dispatch)给相应的处理线程。

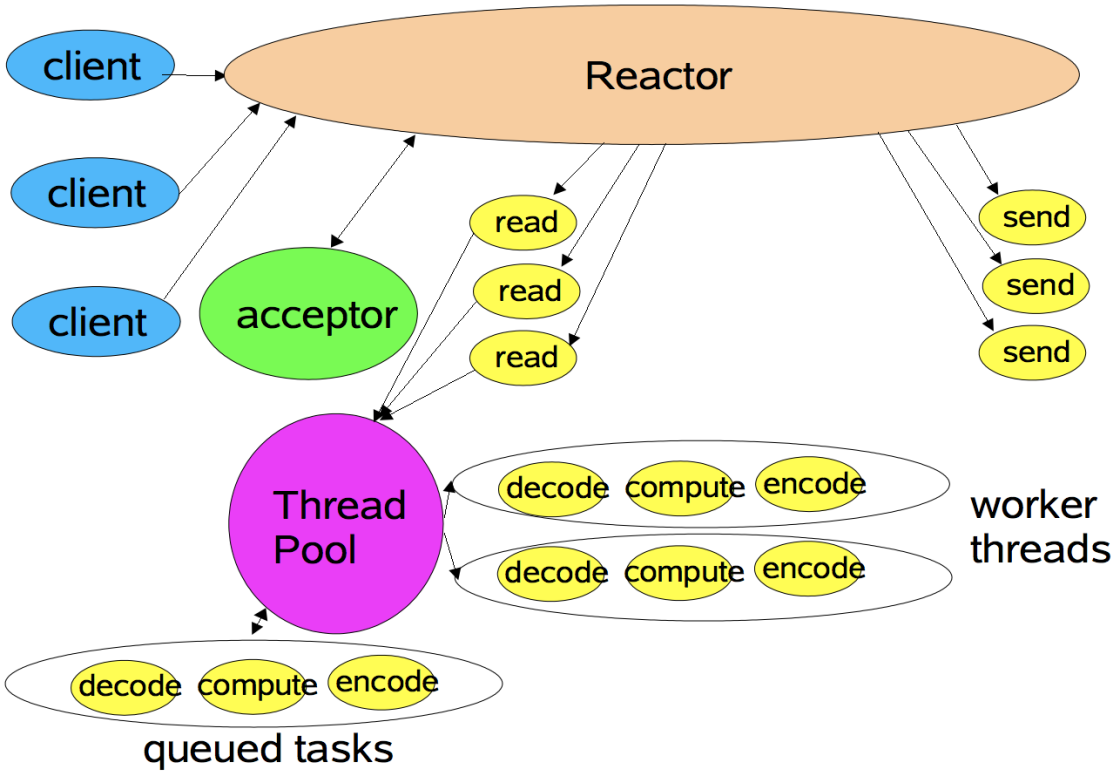
Reactor模式有三种典型的实现方案：

- 单Reactor单线程



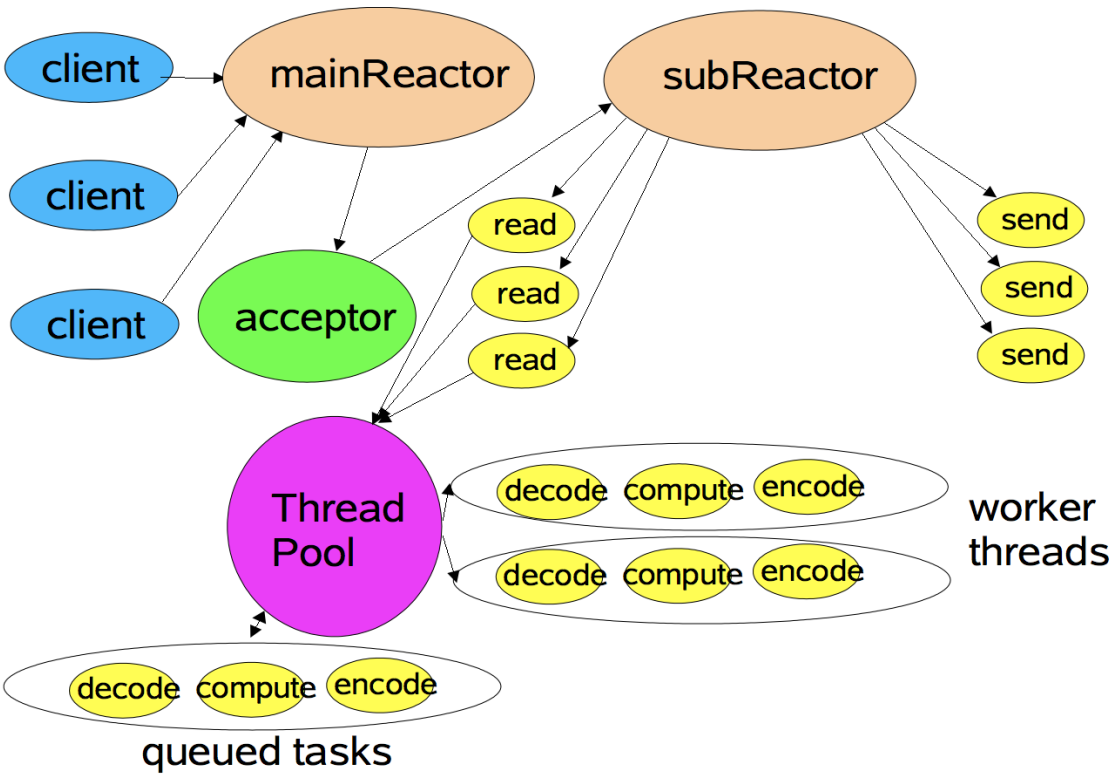
Reactor的单线程模式的单线程主要是针对于I/O操作而言，也就是所有的I/O的accept()、read()、write()以及connect()操作都在一个线程上完成的。并且连非I/O的业务操作也在该线程上进行处理了(例如decode，compute，encode)

- 单Reactor多线程



与单线程Reactor模式不同的是，添加了一个工作者线程池，并将非I/O操作从Reactor线程中移出转交给工作者线程池来执行。这样能够提高Reactor线程的I/O响应，不至于因为一些耗时的业务逻辑而延迟对后面I/O请求的处理。但是所有的I/O操作依旧由一个Reactor来完成，包括I/O的accept()、read()、write()以及connect()操作。

- 主从Reactor多线程



多Reactor线程模式将"接受客户端的连接请求"和"与该客户端的通信"分在了两个Reactor线程来完成。mainReactor完成接收客户端连接请求的操作，它不负责与客户端的通信，而是将建立好的连接转交给subReactor线程来完成与客户端的通信，这样一来就不会因为read()数据量太大而导致后面的客户端连接请求得不到即时处理的情况。并且多Reactor线程模式在海量的客户端并发请求的情况下，还可以通过实现subReactor线程池来将海量的连接分发给多个subReactor线程，在多核的操作系统中这能大大提升应用的负载和吞吐量。

注意：当有I/O事件就绪时，相关的subReactor就将事件派发给响应的处理器处理。注意，这里subReactor线程只负责完成I/O的read()操作，在读取到数据后将业务逻辑的处理放入到线程池中完成，若完成业务逻辑后需要返回数据给客户端，则相关的I/O的write操作还是会被提交回subReactor线程来完成。

注意，所以的I/O操作(包括，I/O的accept()、read()、write()以及connect()操作)依旧还是在Reactor线程(mainReactor线程 或 subReactor线程)中完成的。Thread Pool(线程池)仅用来处理非I/O操作（decode，compute，encode）的逻辑。

Netty(https://www.zhihu.com/search?type=content&q=netty)

Netty 是一个基于NIO的客户、服务器端的编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户、服务端应用。Netty相当于简化和流线化了网络应用的编程开发过程，例如：基于TCP和UDP的socket服务开发。

Netty是建立在NIO基础之上，Netty在NIO之上又提供了更高层次的抽象。

在Netty里面，Accept连接可以使用单独的线程池去处理，读写操作又是另外的线程池来处理。

Accept连接和读写操作也可以使用同一个线程池来进行处理。而请求处理逻辑既可以使用单独的线程池进行处理，也可以跟放在读写线程一块处理。线程池中的每一个线程都是NIO线程。用户可以根据实际情况进行组装，构造出满足系统需求的高性能并发模型。

一句话，Netty出现的目的就是解决网络通信(socket)的痛点，他是建立在Java NIO基础之上的一个框架（其实就是一个jar包），通过这个框架，我们不需要编写复杂的代码去实现客户端与服务端的连接通信。

为什么不用NIO?

NIO的类库和API繁杂，学习成本高；

需要熟悉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络编程非常熟悉，才能写出高质量的NIO程序。

臭名昭著的epoll bug。它会导致Selector空轮询，最终导致CPU 100%。直到JDK1.7版本依然没得到根本性的解决。

Selector BUG出现的原因

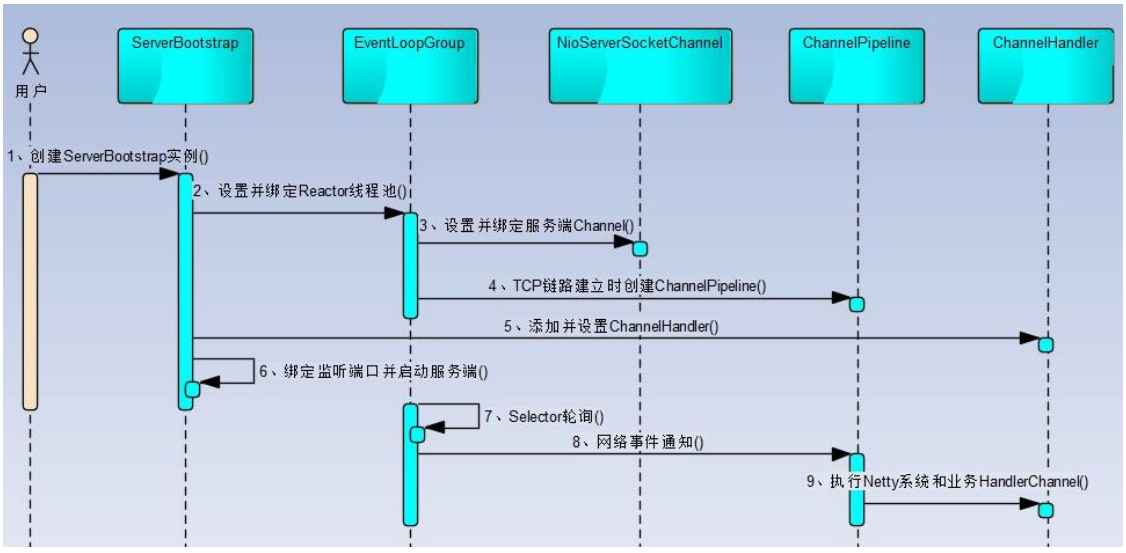
若Selector的轮询结果为空，也没有wakeup或新消息处理，则发生空轮询，CPU使用率100%，

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。

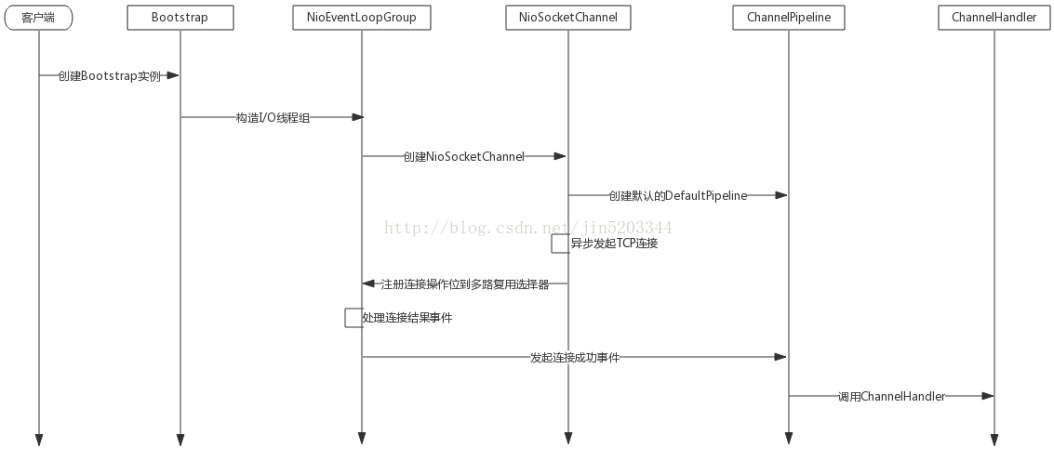
Netty 执行流程

服务端



1、创建ServerBootStrap实例
2、设置并绑定Reactor线程池：EventLoopGroup，EventLoop就是处理所有注册到本线程的Selector上面的Channel
3、设置并绑定服务端的channel(NioServerSocketChannel)
4、创建处理网络事件的ChannelPipeline和handler，网络时间以流的形式在其中流转，handler完成多数的功能定制：比如编解码 SSl安全认证
5、绑定并启动监听端口
6、当轮训到准备就绪的channel后，由Reactor线程：NioEventLoop执行pipline中的方法，最终调度并执行channelHandler

客户端



1.用户线程创建Bootstrap实例

2.创建处理客户端连接，I/O读写Reactor线程组NioEventLoopGroup

3.创建NioSocketChannel

4.创建默认的ChannelHandlerPipeline,用户调度和执行网络事件

5.异步发起TCP连接，如果成功将NioSocketChannel注册到多路复用选择器上,监听读操作位,用于数据读取和消息发送,如果失败，注册连接操作位到多路复用选择 器，等待连接结果

6.注册对应的网络监听状态位到多路复用选择器

7.由多路复用选择器轮询Channel,处理连接结果

8.如果连接成功，设置Future结果，发送连接成功事件，触发ChannelHandlerPipeline执行

9.由ChannelHandlerPipeline调度和执行系统和用户ChannelHandler

Netty核心组件(https://blog.csdn.net/thinking_fioa/article/details/80588138)

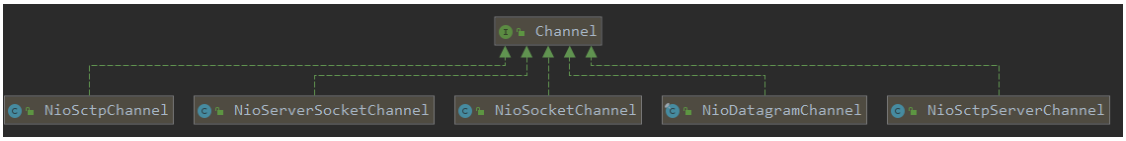
Channel、EventLoop和ChannelFuture

Channel、EventLoop和ChannelFuture是Netty用于对网络进行的抽象

1. Channel ------ Socket
2. EventLoop ------ 控制流、多线程和并发
3. ChannelFuture ------ 异步通知

- channel是Netty对网络操作的抽象类，包括基本的I/O操作，比如bind，connect，read，write

同时在netty中也可以通过channel获取eventLoop；



- Channel 为Netty 网络操作抽象类，EventLoop 主要是为Channel 处理 I/O 操作，两者配合参与 I/O 操作。

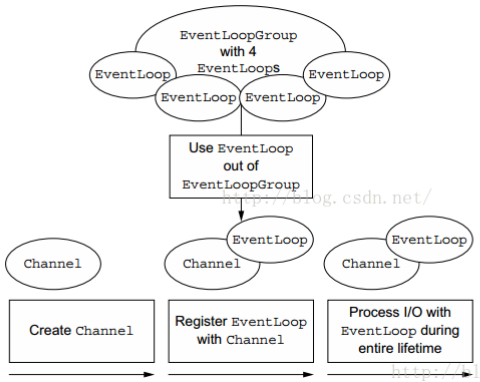


Figure 3.1 Channels, EventLoops, and EventLoopGroups

1)客户端连接成功，新建一个channel与客户端绑定；

2）channel从EventLoopGroup获取一个EventLoop，并注册至该EventLoop，channel生命周期内都与该EventLoop绑定在一起(注册时获得selectionKey)

3）channel同用户端进行网络连接、关闭和读写，生成相对应的event（改变selectinKey信息），触发eventloop调度线程进行执行

约定俗成的关系(非常重要):

1. 一个EventLoopGroup包含一个或多个EventLoop
2. 一个EventLoop在其生命周期内只能和一个Thread绑定
3. 由EventLoop处理的I/O事件都由它绑定的Thread处理
4. 一个Channel在其生命周期内，只能注册于一个EventLoop
5. 一个EventLoop可能被分配处理多个Channel。也就是EventLoop与Channel是1:n的关系
6. **一个Channel上的所有ChannelHandler的事件由绑定的EventLoop中的I/O线程处理**
7. 不要阻塞Channel的I/O线程，可能会影响该EventLoop中其他Channel事件处理

NioEventLoopGroup初始化时未指定线程数，那么会使用默认线程数，即`线程数 = CPU核心数 * 2`;

每个NioEventLoopGroup对象内部都有一组可执行的`NioEventLoop数组`，其大小是 nThreads, 这样就构成了一个线程池， `一个NIOEventLoop可以理解成就是一个线程`。

- Netty 为异步非阻塞，即所有的 I/O 操作都为异步的，因此，我们不能立刻得知消息是否已经被处理了。Netty 提供了 ChannelFuture 接口，通过该接口的 addListener() 方法注册一个 ChannelFutureListener，当操作执行成功或者失败时，监听就会自动触发返回结果。

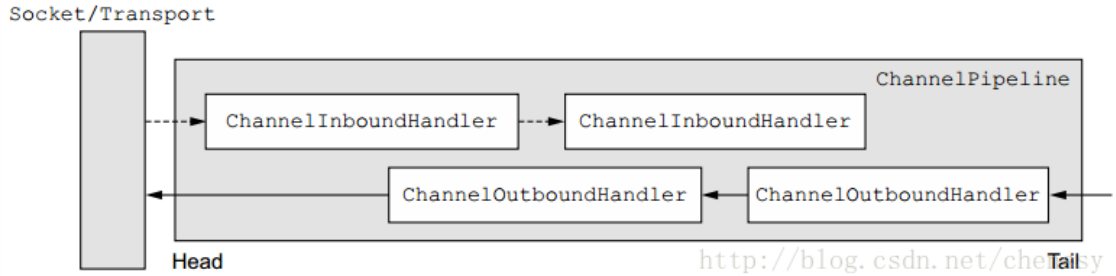
Netty 中的 I/O 操作是异步的，包括 Bind、Write、Connect 等操作会简单的返回一个 ChannelFuture。

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。

ChannelHandler 和 ChannelPipeline

channel绑定至某一个EventLoop之后，当channel与客户端进行网络连接/关闭、读/写等操作时，EventLoop会调度自己绑定的线程执行相应的业务逻辑。而这个时候就会调用对应的 ChannelHandler 来处理不同的事件。

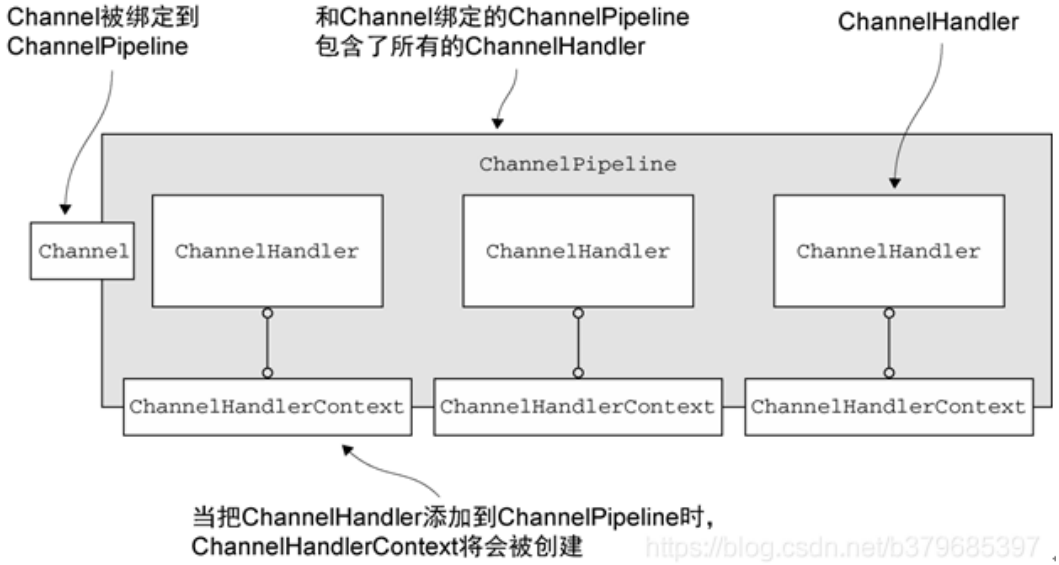
ChannelHandler 有两个核心子类 ChannelInboundHandler 和 ChannelOutboundHandler，其中 ChannelInboundHandler 用于接收、处理入站数据和事件，而 ChannelOutboundHandler 则相反。



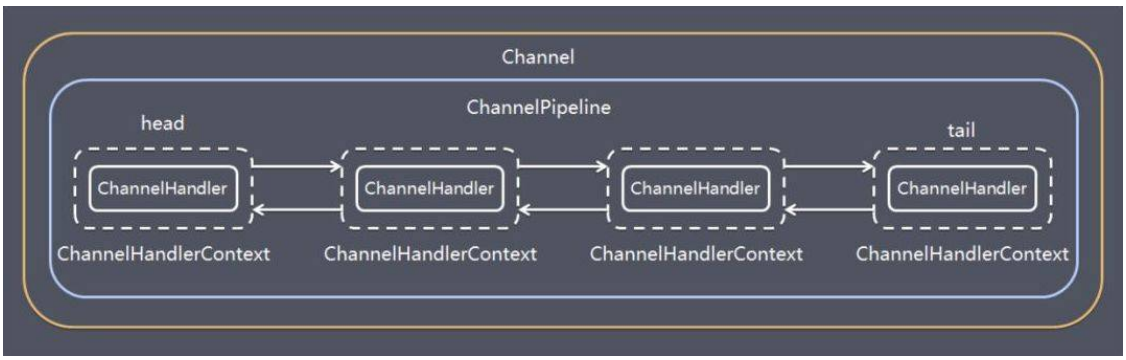
ChannelPipeline本质上是ChannelHandler链的容器，当一个数据流进入 ChannlePipeline 时，它会从 ChannelPipeline 头部开始传给第一个 ChannelInboundHandler ，当第一个处理完后再传给下一个，一直传递到管道的尾部。与之相对应的是，当数据被写出时，它会从管道的尾部开始，先经过管道尾部的 “最后” 一个ChannelOutboundHandler，当它处理完成后会传递给前一个 ChannelOutboundHandler 。

ChannelHandlerContext

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ChannelHandlerContext。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Channel-Pipeline 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（入站下一个，出站上一个）能够处理该事件的ChannelHandler。



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。

Bootstrap与ServerBootstrap

bootstrap用于引导Netty的启动，Bootstrap是客户端的引导类，ServerBootstrap是服务端引导类

服务端需要两个EventLoopGroup

Netty的服务端负责两项任务:

1. 监听本地端口，等待客户端连接。
2. 建立客户端通信的临时分配的端口。所以服务端有两个EventLoopGroup，通常称为: bossEventLoopGroup + workerEventLoopGroup.

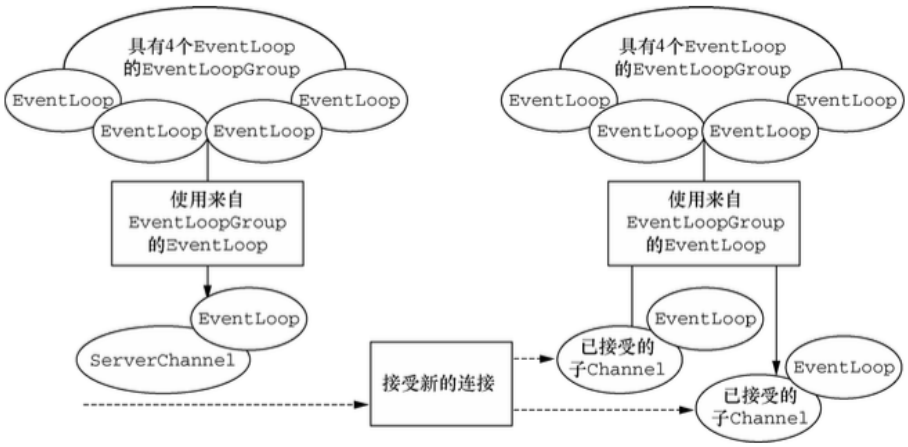
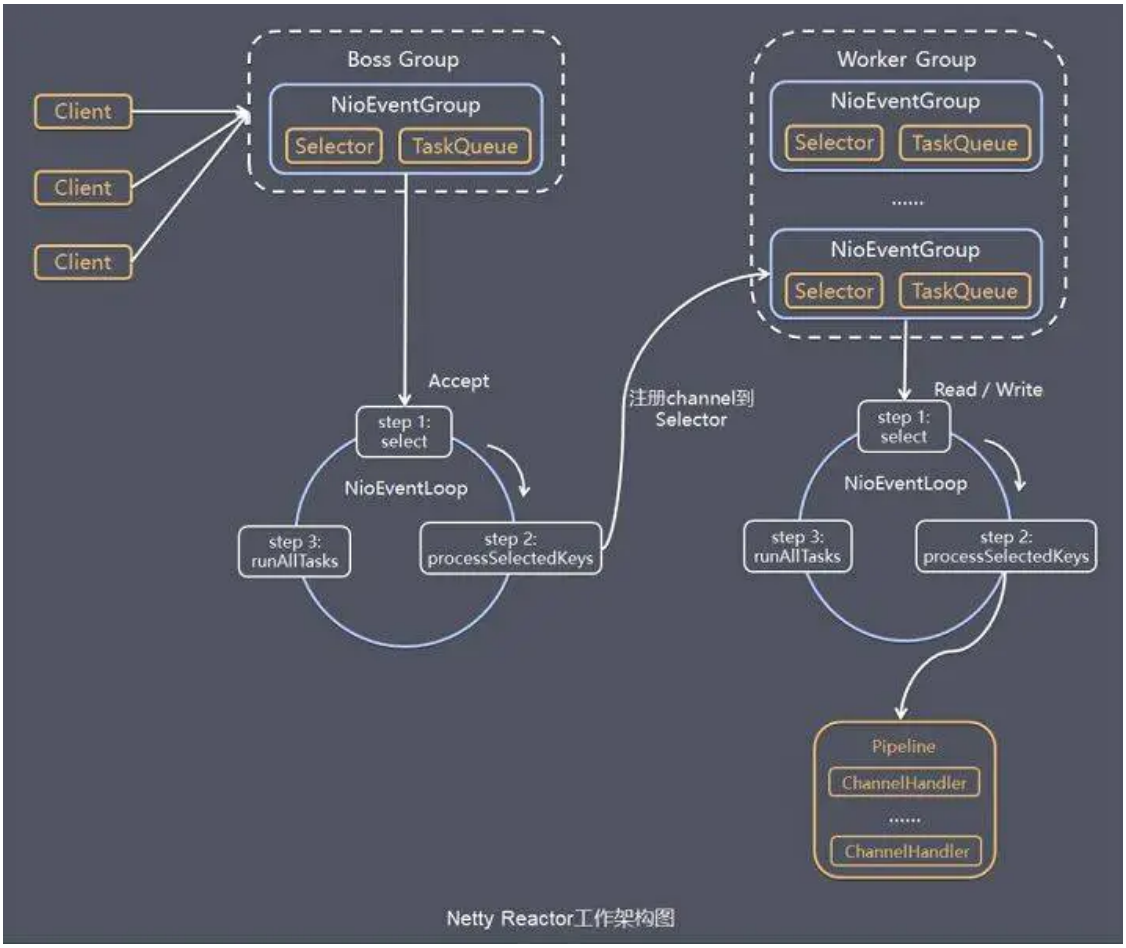


图 3-4　具有两个 EventLoopGroup 的服务器

左边对应bossEventLoopGroup，只包含一个ServerChannel，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。

右边是workerEventLoopGroup，在收到bossEventLoopGroup创建的新的子channel之后，EventLoopGroup会给他分配一个EventLoop，用户与客户端进行数据通信。

Netty工作原理



Server 端包含 1 个 Boss NioEventLoopGroup 和 1 个 Worker NioEventLoopGroup。

NioEventLoopGroup 相当于 1 个事件循环组，这个组里包含多个事件循环 NioEventLoop，**每个 NioEventLoop 包含 1 个 Selector 和 1 个事件循环线程。**

boss NioEventLoop循环任务

- 轮询Accept事件。
- 处理Accept IO事件，与Client建立连接，生成NioSocketChannel,并将NioSocketChannel注册到某个work NioEventLoop的Selector上。
- 处理任务队列中的任务。

work NioEventLoop循环任务

- 轮询Read、Write事件。
- 处理IO事件，在NioSocketChannel可读、可写事件发生时进行处理。
- 处理任务队列中的任务。

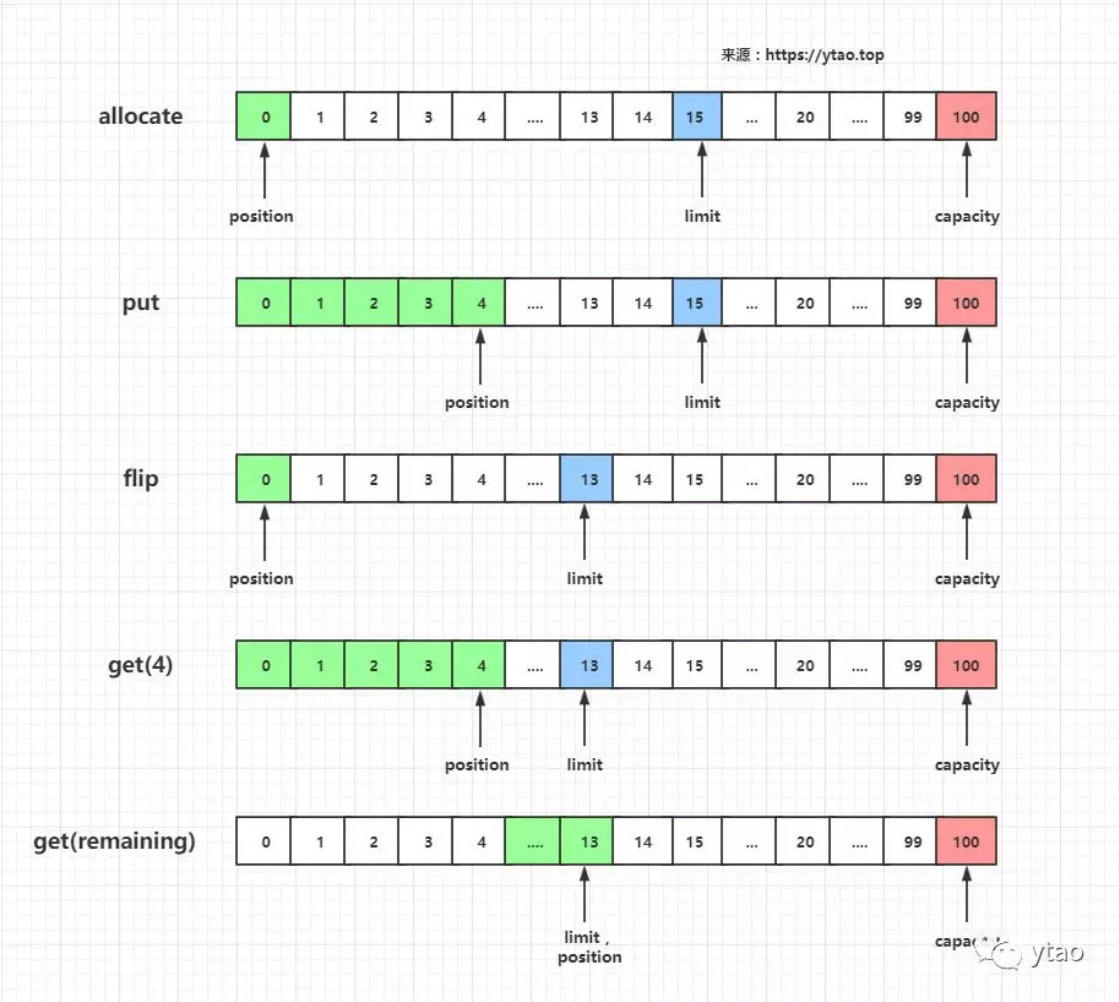
NIO ByteBuffer与Netty ByteBuf

NIO ByteBuffer主要包括三个参数：

- position 读写指针位置
- limit 当前缓存区大小限制
- capacity 缓冲区大小

`ByteBuffer#flip`相关源码：

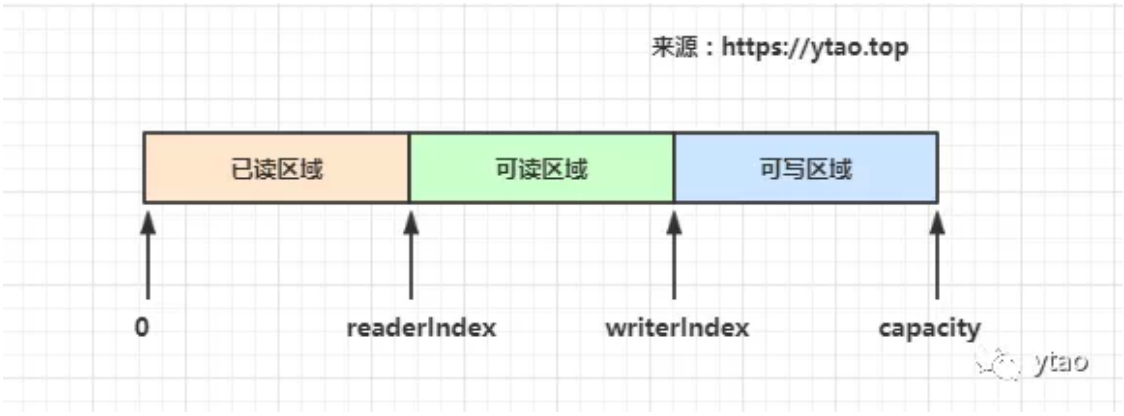
```
public final Buffer flip() {
    limit = position;
    position = 0;
    mark = -1;
    return this;
}
```
ytao



ByteBuf 有读写指针是分开的，分别是 `buf#readerIndex` 和 `buf#writerIndex`，当前缓冲器大小 `buf#capacity`。

这里缓冲区被两个指针索引和容量划分为三个区域：

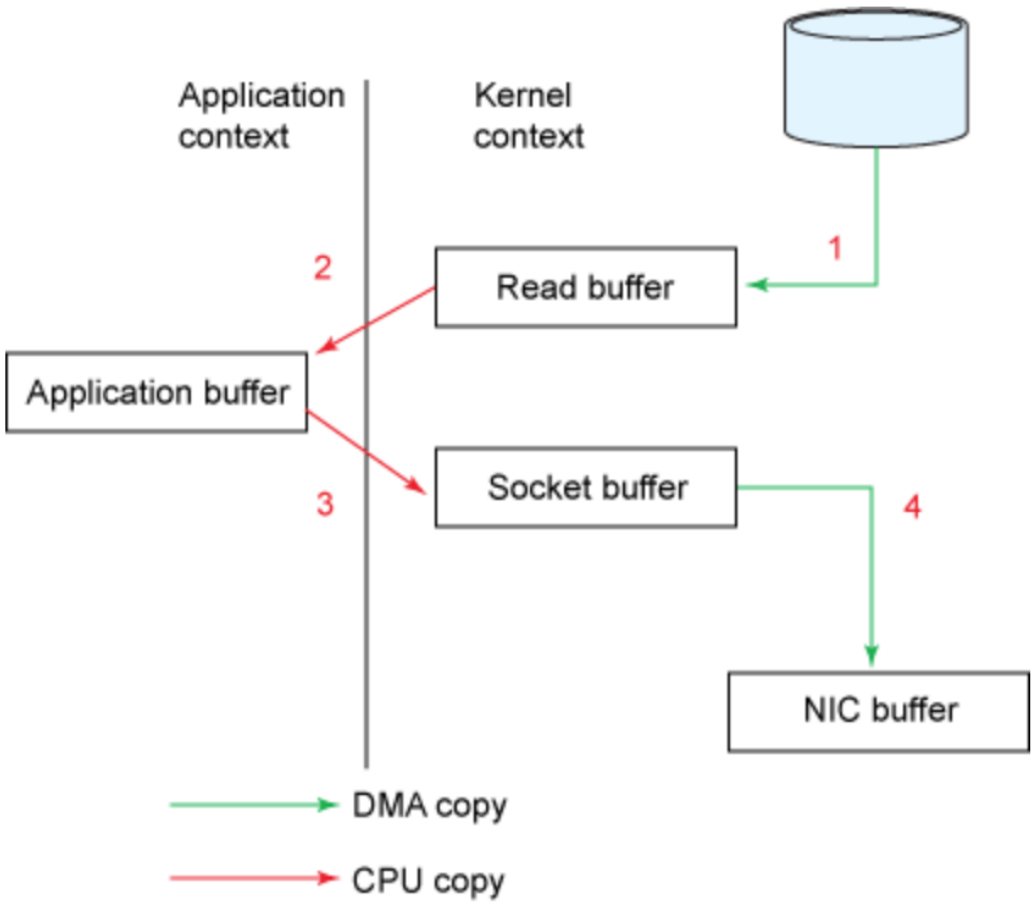
- 0 -> readerIndex 为已读缓冲区域，已读区域可重用节约内存，readerIndex 值大于或等于 0
- readerIndex -> writerIndex 为可读缓冲区域，writerIndex 值大于或等于 readerIndex
- writerIndex -> capacity 为可写缓冲区域，capacity 值大于或等于 writerIndex



零拷贝

DMA是指外部设备不通过CPU而直接与系统内存交换数据的接口技术。这样数据的传送速度就取决于存储器和外设的工作速度。

图 1. 传统的数据拷贝方法



1.首先，调用read方法，文件从user模式拷贝到了kernel模式；（用户模式->内核模式的上下文切换，在内部发送sys_read() 从文件中读取数据，存储到一个内核地址空间缓存区中）

2.之后CPU控制将kernel模式数据拷贝到user模式下；（内核模式-> 用户模式的上下文切换，read()调用返回，数据被存储到用户地址空间的缓存区中）

3.调用write时候，先将user模式下的内容copy到kernel模式下的socket的buffer中（用户模式->内核模式，数据再次被放置在内核缓存区中，send（）套接字调用）

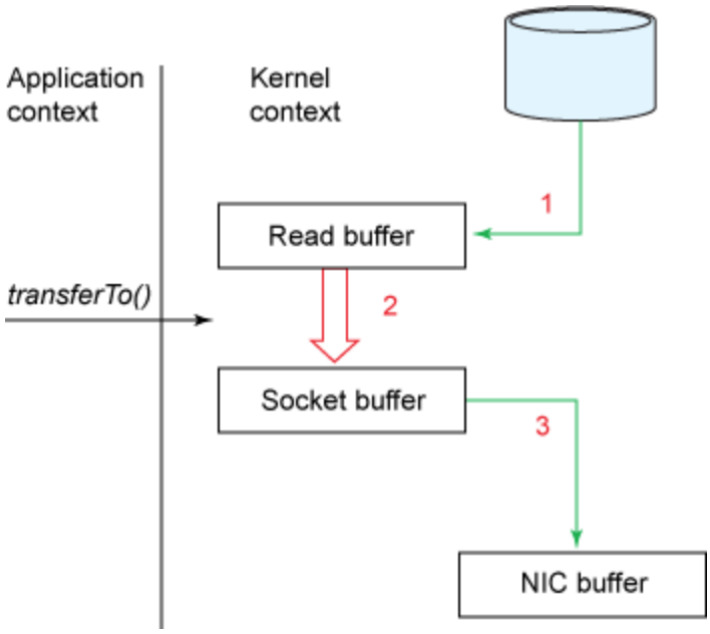
4.最后将kernel模式下的socket buffer的数据copy到网卡设备中；（send套接字调用返回）

用户模式->内核模式->用户模式->内核模式->用户模式

涉及到了 4 次的上下文切换和 4 次的数据拷贝操作，其中磁盘与内核态进行的拷贝操作应用了 DMA 技术（全称 Direct Memory Access，它是一项由硬件设备支持的 IO 技术，应用这项技术可以更好的利用 CPU 资源，在此期间 CPU 可以去做其他事情），而内核态缓冲区到应用程序传输数据需要 CPU 的参与，在此期间 CPU 不能做其他工作。

mmap 提升拷贝性能

图 3. 使用 `transferTo()` 方法的数据拷贝



mmap 是一种内存映射的方法，它可以将磁盘上的文件映射进内存。用户程序可以直接访问内存即可达到访问磁盘文件的效果，这种数据传输方法的性能高于将数据在内核空间和用户之间来回拷贝操作，通常用于高性能要求的应用程序中。

1.transferTo()方法使得文件的内容直接copy到了一个read buffer（kernel buffer）中

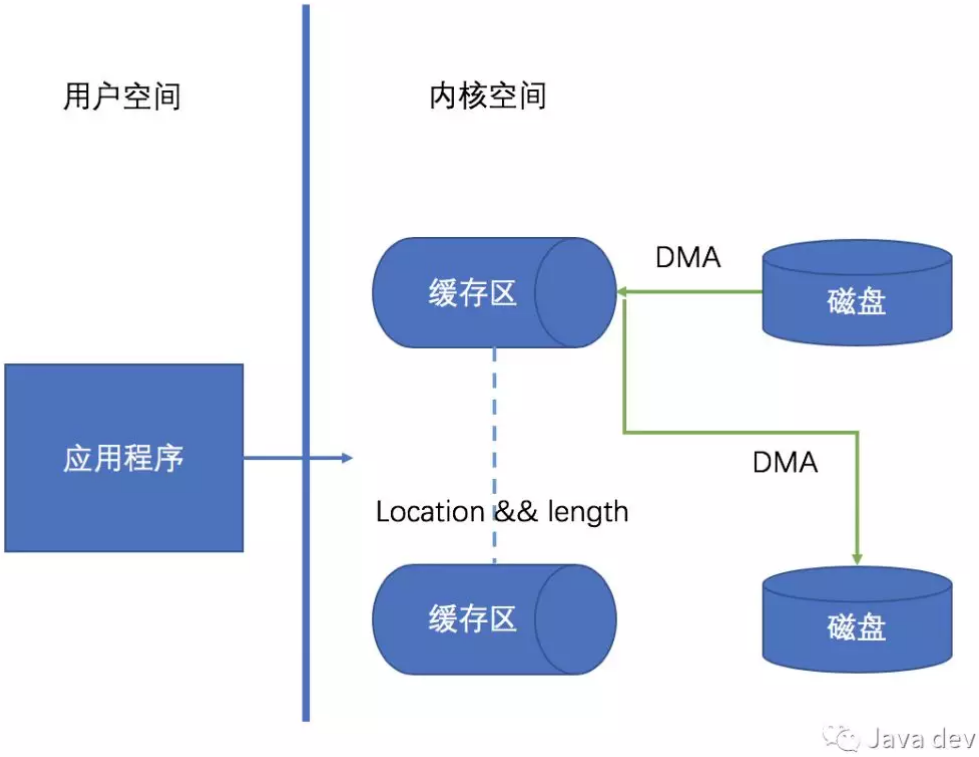
2.然后数据（kernel buffer）copy到socket buffer中

3.最后将socket buffer中的数据copy到网卡设备（protocol engine）中传输；

上下文切换从4次减少到2次，同时把数据copy的次数从4次降低到3次(两次DMA拷贝一次CPU拷贝)；

零拷贝技术

核心：去除CPU拷贝



在 Linux 内核 2.1 版本中引入了 sendfile 系统调用，采用这种方式后内核态的缓冲区之间不再进行 CPU 拷贝操作，只需要将源数据的地址和长度告诉目标缓冲区，然后直接采用 DMA 技术将数据传输到目的地

Netty 零拷贝(广义上的零拷贝，减少内存之间的复制操作)

1) Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

2) Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer。

3) Netty的文件传输采用了transferTo方法，它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。

粘包与拆包(https://www.cnblogs.com/coding-diary/p/11650686.html)

粘包和拆包

同样的，操作系统通过TCP协议发送数据的时候，也会先将数据存放在缓冲区中，假设缓冲区的大小为1024个字节

粘包

如果发送的数据包比较小，远小于缓冲区的大小，TCP会将多个数据包合并为一个数据包发送，这就发生了粘包

拆包

如果发送的数据包比较大，远大于缓冲区的大小，TCP会将数据包拆分为多个数据包发送，这就发生了拆包

有以下4种解决办法

(1) 消息定长，例如每个数据包的大小都为128字节，如果不够，空位补空格

(2) 客户端在每个包的末尾使用固定的分隔符，例如\r\n，如果一个包被拆分了，则等待下一个包发送过来之后找到其中的\r\n，然后对其拆分后的头部部分与前一个包的剩余部分进行合并，这样就得到了一个完整的包

(3) 将消息分为头部和消息体，在头部中保存有当前整个消息的长度，只有在读取到足够长度的消息之后才算是读到了一个完整的消息

(4) 通过自定义协议进行粘包和拆包的处理

Netty中的粘包和拆包解决方案

1. 固定长度的拆包器 FixedLengthFrameDecoder，每个应用层数据包的都拆分成都是固定长度的大小
2. 行拆包器 LineBasedFrameDecoder，每个应用层数据包，都以换行符作为分隔符，进行分割拆分
3. 分隔符拆包器 DelimiterBasedFrameDecoder，每个应用层数据包，都通过自定义的分隔符，进行分割拆分
4. 基于数据包长度的拆包器 LengthFieldBasedFrameDecoder，将应用层数据包的长度，作为接收端应用层数据包的拆分依据。按照应用层数据包的大小，拆包。这个拆包器，有一个要求，就是应用层协议中包含数据包的长度