1.Netty学习前需要掌握知识

-- 1.牢固的JAVA基础

-- 2.熟悉Linux服务器

-- 3.有基础的网络知识

2.异步事件驱动NIO框架Netty介绍

-- 1.Netty是由JBOSS提供的一个java开源框架,是业界最流行的NIO框架，整合了多种协议(包括FTP,SMTP,HTTP等各种二进制文本协议)的实现经验，

API使用简单，框架成熟稳定，社区活跃，互联网、大数据、网络游戏、电信通信行业进过验证。

-- 2.使用Netty框架的产品

--- 1.搜索引擎框架 elasticsearch

--- 2.Hadopp子项目Avro项目,使用Netty作为底层通信框架

--- 3.阿里巴巴开源的RPC框架 Dubbo

3.Netty开发环境

-- 1.IDEA + JDK8 + Netty4.x

4.使用JDK自带BIO编写client-server通信

-- 1.BIO模式:同步阻塞，传统Socket编程，一客户端一线程

-- 2.Server核心代码:

ServerSocket server =new ServerSocket(port);

Socket socket=null;

while(true){

socket=server,accept();

new Thread(new xxxHandler(socket)).start();

}

-- 3.client核心代码:

Socket socket=new Socket(ip,port);

socket.getInputStream();

socket.getOutputStream();

5.BIO编写client-server通信优缺点分析

-- 1.优点:模型简单，编码简单

-- 2.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

-- 3.案例：tomcat7之前，都是使用BIO,7之后使用NIO

-- 4.改进:伪NIO,使用线程池去处理业务逻辑

6.通俗概念讲解 同步异步、阻塞和非阻塞

-- 1.同步阻塞:你把衣服丢到洗衣机，然后看着洗衣机洗完，洗好后再去晾衣服（你就干等着，啥都不做，阻塞在那边）

-- 2.同步非阻塞:你把衣服丢到洗衣机洗，然后回客厅去做其他事情，定时去阳台看洗衣机是不是洗完了，洗好后再去晾衣服（等待期间可做其他事情）

-- 3.异步阻塞:几乎没有这个情况，可以忽略，异步应该就非阻塞

-- 4.异步非阻塞:你把衣服丢到洗衣机，然后回客厅做其他事情，衣服洗好后会自动去晾衣服，晾完衣服后放个音乐通知你洗好衣服并且衣服晾好了

7.Linux网络编程中的五种I/O模型理解

网络IO,用户程序和内核的交互为基础进行理解

IO操作分为两步：发起IO请求等待数据准备，实际IO操作(洗衣服，晾衣服)

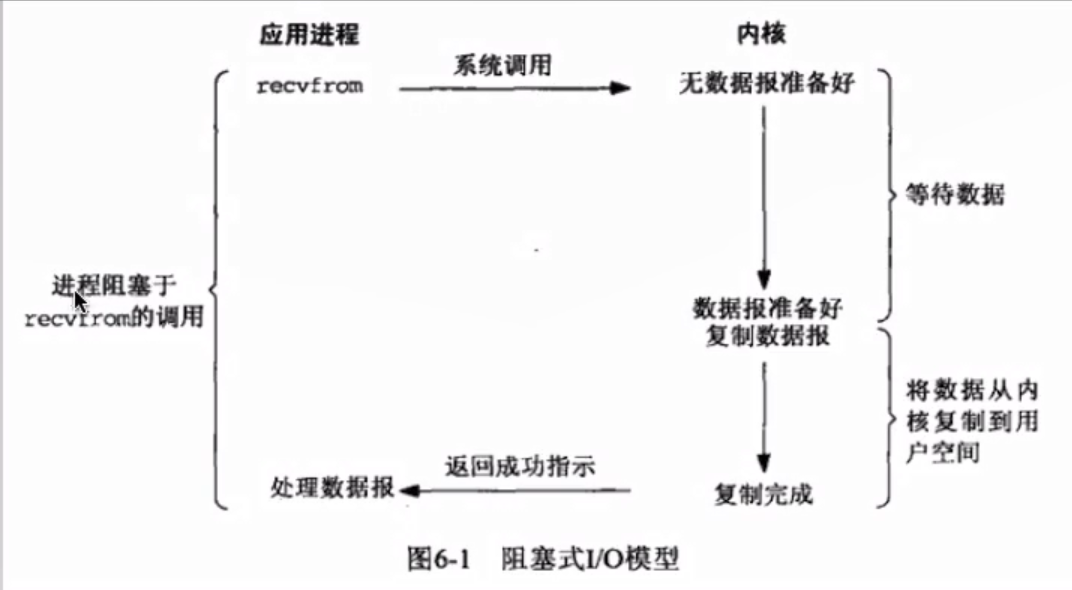
同步需要主动读写数据，在读写数据的过程中还是会阻塞（好比晾衣服阻塞了）

异步仅仅需要IO操作完毕的通知，并不主动读写数据，由操作系统内核完成数据的读写（机器人帮你自动晾衣服）

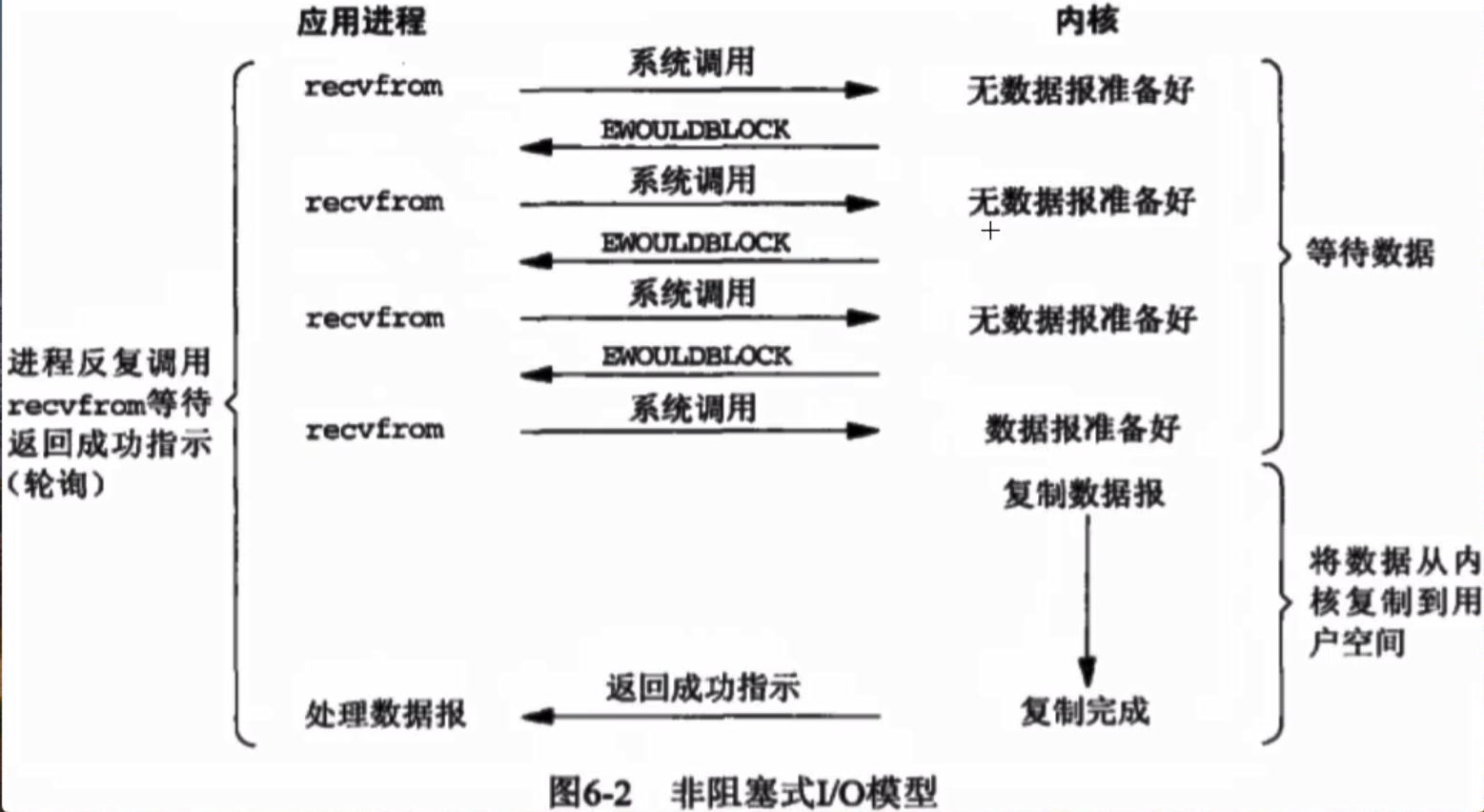
五种IO模型:阻塞IO,非阻塞IO,多路复用IO,信号驱动IO和异步IO

前四种都是同步IO，在内核数据copy到用户控件都是阻塞的

--- 1. 阻塞式IO



--- 2.非阻塞IO



--- 3.IO多路复用(select,poll,epoll……..)

一个线程去处理多个请求

IO多路复用是阻塞在select,epoll这样的系统调用，没有阻塞在真正的IO系统调用如

Recvfrom,进程受阻于select调用，等待可能多个套接口中的任何一个变为可读。

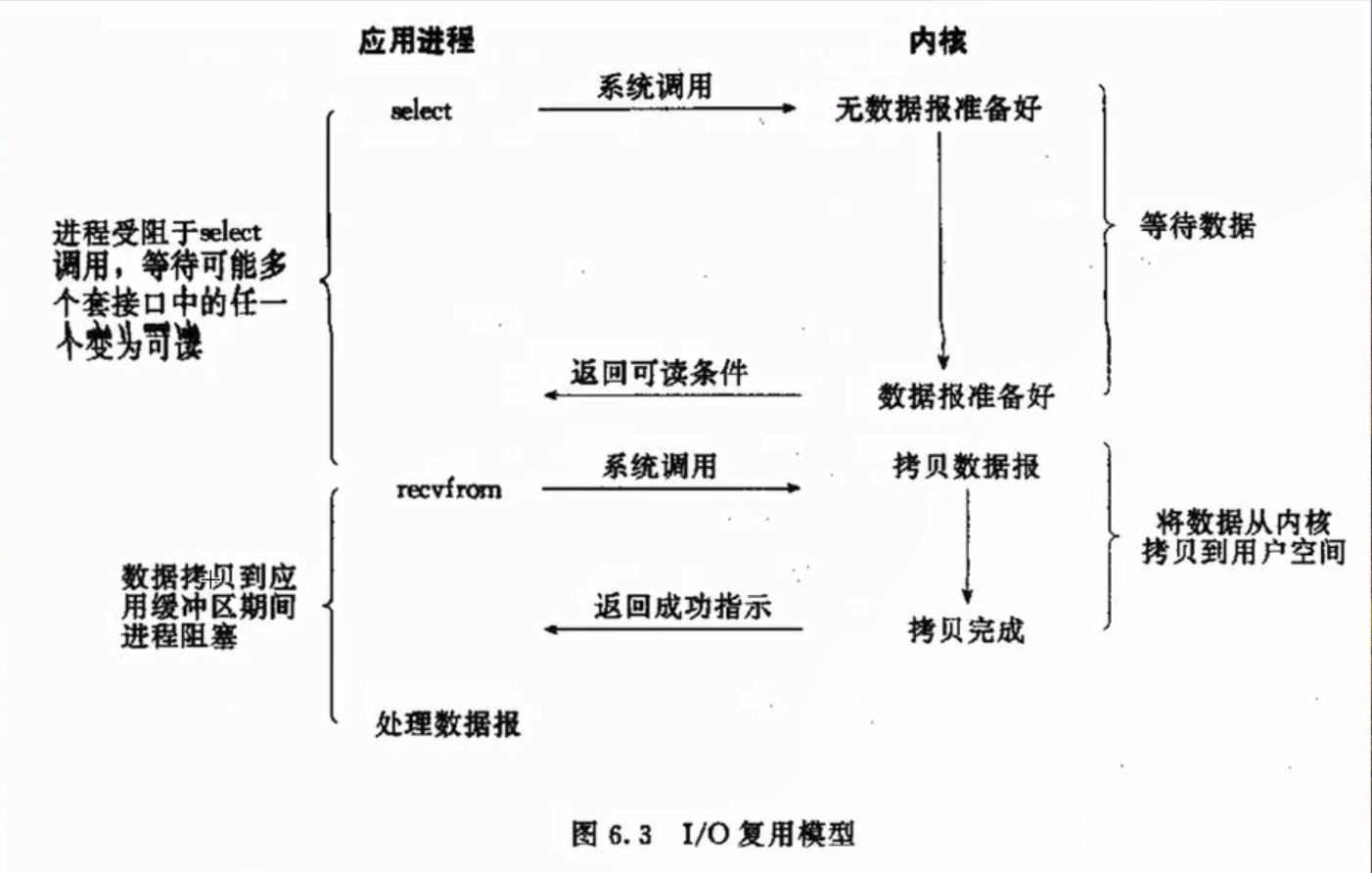
IO多路复用使用两个系统调用(select和recvfrom)

阻塞IO只调用了一个系统调用(recvfrom)

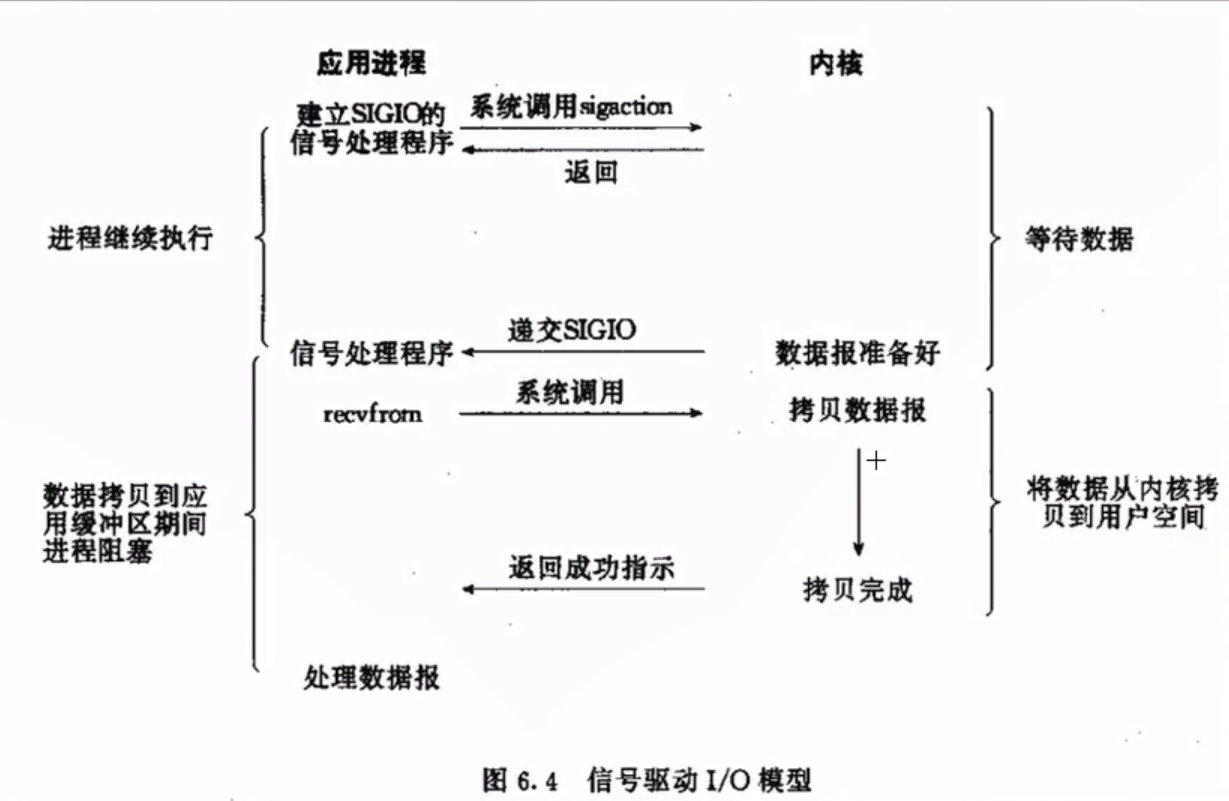
Select/epoll的核心是可以同时处理多个connection,而不是更快，所以连接数不高的话，性能不一定比多线程+阻塞IO好

多路复用模型中，每一个socket,设置为non-blocking,阻塞是select函数block，而不是

Socket阻塞的

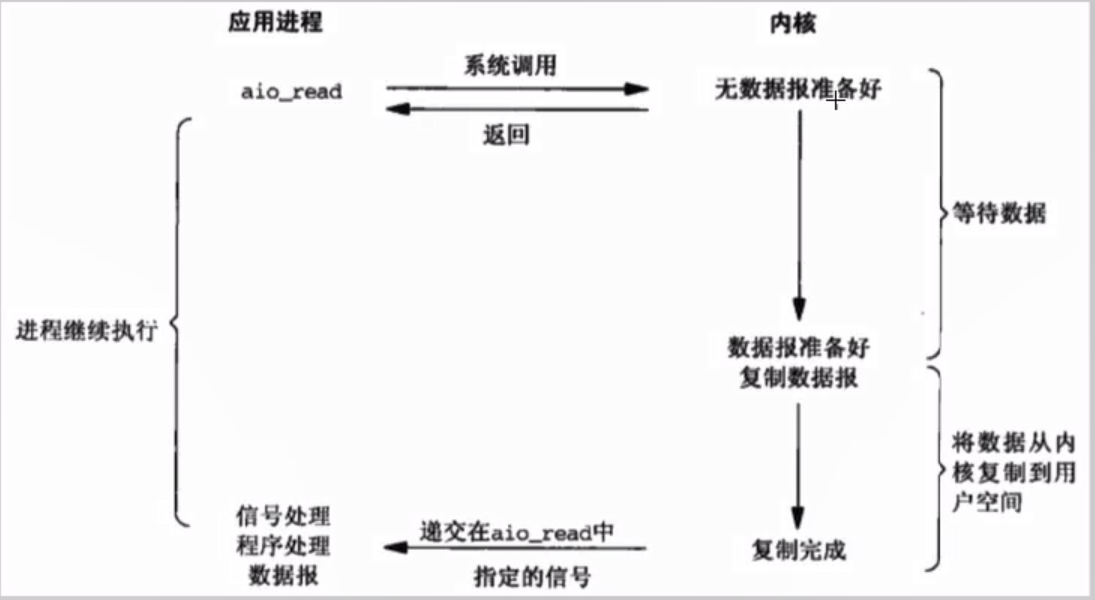


--- 4.信号驱动IO



--- 5.异步IO

Future-listener机制



总结点：

--IO操作分为两步

-----1.发起IO请求，等待数据准备(waiting for the data to be ready)

------2.实际的IO操作，将数据从内核拷贝到进程中(copying the data from the kernel to

the process)

前四种IO模型都是同步IO操作，他们的第二阶段是一样的:数据从内核复制到应用缓

冲期间（用户空间），区别在于第一阶段，进程阻塞于recvfrom调用或者select()函数。

异步IO在这两个阶段都要处理。

阻塞IO和非阻塞IO的区别在于第一步，发起IO请求是否会被阻塞，如果阻塞直到完

成那么就是传统的阻塞IO,如果不阻塞就是非阻塞IO

同步IO和异步IO的区别在于第二步骤是否阻塞，如果实际的IO读写阻塞请求进程，

那么就是同步IO,如果不阻塞，而是操作系统帮你完成IO操作再将结果返回给你，那

么就是异步IO

核心点:

阻塞和非阻塞指的是线程的状态，异步和同步说的是消息的通知机制

同步需要主动读写数据，异步不需要

同步IO和异步IO是针对用户应用程序和内核的交互

8.IO多路复用技术select,poll理解

-- 1.什么是IO多路复用

IO是指网络IO，多路指多个TCP连接（即socket或者channel）,复用指复用一

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，不必创建过多的进程、线程，也不必须维护这些进程、

线程

-- 2.SELECT理解

基本原理:

监视文件3类描述符:writefds/readfds/exceptfds

调用后select函数会阻塞住，等有数据 可读、可写、出异常 或者 超时 就会

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，随着文件扫描符fd数

量增多而下降。

--- 2.每次调用select()，需要把fd集合从用户态拷贝到内核态，并进行遍历

（消息传递都是从内核到用户空间）

--- 3.最大的缺陷就是单个进程打开的fd有限制，默认是1024

(可修改宏定义,但是效率仍然慢,static final int MAX\_FD)

--3.poll 理解

基本流程:selct()和poll()系统调用的大体一样,处理多个描述符也是采用轮询的

方式，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。

最大区别是:poll没有最大文件描述符限制(使用链表的方式存储fd)

-- 4.epoll理解

基本原理:

在2.6内核中提出的，对比select和poll,epoll更加灵活，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fd就绪，内核就会采用callback的

回调机制来激活对应的fd

优点:

----1.没有fd这个限制，所支持的fd上限是操作系统的最大文件句柄数，1G

内存大概支持10万个句柄

----2.效率提高，使用回调通知而不是轮询的方式，不会随着fd数目的增加

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

Epoll-Linux内核核心函数

----epoll\_create() 在linux内核里面申请一个文件系统(B树),返回epoll对象

也是一个fd

----epoll\_ctl() 操作epoll 对象，在这个对象里面修改添加删除对应的连接fd,

绑定一个callback函数

----epoll\_wait() 判断并完成对应的IO操作

9.JAVA的IO演进历史

--1.jdk1.4之前是采用同步阻塞模型，也就是BIO

当时大型服务一般采用c或者c++，因为可以直接操作系统提供的异步IO(AIO)

--2.jdk1.4推出NIO,支持非阻塞IO

Jdk1.7升级推出NIO2.0 ,提供AIO的功能，支持文件和网络套接字的异步IO

10.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，将一个或多个客户的服务请求分离（demultiplex）和调度(dispatch)给应用程序。在

事件驱动的应用中，同步地、有序地处理同时接收的多个服务请求，一般出现在高并发系

统中，比如Netty,Redis等

优点:

响应快，不会因为单个同步而阻塞，虽然Reactor本身仍然是同步的

编程相对简单，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

可扩展性，可以方便的通过增加Reactor实例个数来充分利用CPU资源

缺点：

相比传统的简单模型，Reactor增加了一定的复杂性，因而有一定的复杂性，不易于调试

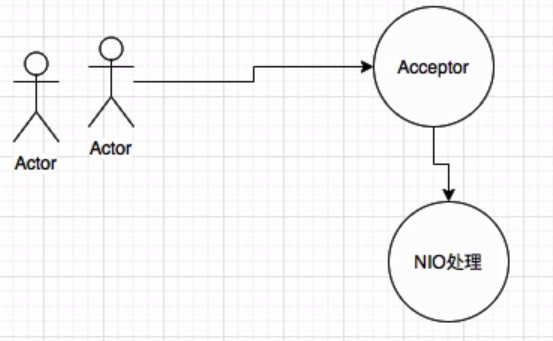
Reactor模式需要系统底层的支持，比如Java中的Selector支持，操作系统的select系统

调用支持

通俗理解:KTV例子 前台接待 服务人员带领去开机器

Reactor模式基于事件驱动，适合处理海量的IO事件，属于同步非阻塞IO(NIO)

Reactor单线程模型（比较少用）



内容:

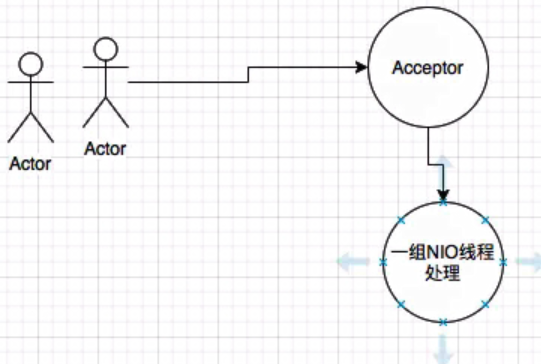
--1.作为NIO服务端，接收客户端的TCP连接；作为NIO客户端，向服务端发起TCP连接

--2.服务端读请求数据并响应；客户端写请求并读取响应

使用场景:

对小业务适合，编码简单；对于高负载、大并发的应用场景不适合，一个NIO线程处理太多请求，则负载过高，并且可能响应变慢，导致大量请求超时，而且万一线程挂了，则不可用了。

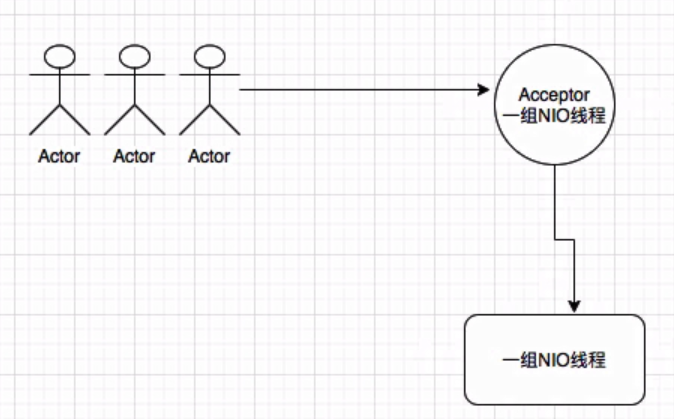
Reactor多线程模型



内容:一个Acceptor线程，一组NIO线程，一般是使用自带的线程池，包含一个任务队列和多个可用的线程

使用场景：可满足大多数场景，但是当Acceptor需要做复杂操作的时候，比如认证等耗时操作，再高并发情况下则也会有性能问题。

Reactor主从线程模型



内容:Acceptor不再是一个线程，而是一组NIO线程；IO线程也是一组NIO线程，这样就是两个线程池去处理接入连接和处理IO。

使用场景：满足目前的大部分场景，也是Netty推荐使用的线程模型。

BossGroup

WorkGroup

11.为什么Netty使用NIO而不是AIO，是同步非阻塞还是异步非阻塞

--1.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整体框架是Reactor模型，采用epoll机制，所以往深的说还是IO多路复用模式,所以也可说netty是同步非阻塞模型(看的层次不一样)

--2.很多人说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，高可用性和高可定制型。

12.EventLoop和EventLoppGroup模块理解

--1.高性能RPC框架的3个要素:IO模型（5种）、数据协议(http/protobuf/Thrift)、线程模型

--2.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服务，可以创建多个eventloop来优化资源的利用，也就是eventloopgroup

--3.EventLoopGroup负责分配eventloop到新创建的channel，里面包含多个eventloop

Eventloopgroup---->>多个eventloop

Eventloop—》维护一个selector

selectors学习资料:http://ifeve.com/selectors/

--4.源码分析默认线程池数量=cpu核数量\*2

13.Channel模块理解

--1.什么是Channel：客户端和服务端建立的一个连接通道(socket连接)

--2.什么是ChannelHandler：负责channel的逻辑处理

--3.什么是ChannelPipeLine：负责管理ChannelHandler的有序容器

--4.它们之间的关系:

一个Channel包含一个ChannelpipeLine,所有channelhandler都会顺序加入到ChannelPipeLine中，创建Channel时会自动创建一个ChannelPipeLine,每个Channel都有一个管理它的channelpipeline,这关联是永久的。

Channel当状态出现变化，就会触发对应的事件

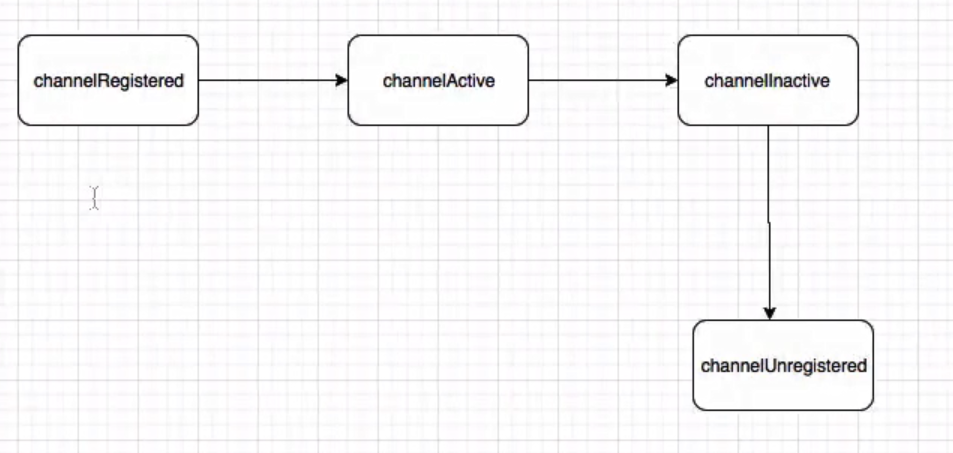
状态：

---1.ChannelRegistered:channel注册到一个EventLoop

---2.ChannelActive:变为活跃状态（连接到了远程主机），可以接收和发送数据

---3.ChannelInactive:channel处于非活跃状态，没有连接到远程主机

---4.channelUnregistered：channel已经创建，但是为注册到一个EventLoop里面，也就是没有和Eventloop中的Selector绑定。



14.Netty启动引导类BootStrap理解

--1.group：设置线程组模型，Reactor线程模型对比EventLoopGroup

---1.单线程

EventLoopGroup bossGroup = new NioEventLoopGroup(1);

try {

ServerBootstrap serverBootstrap = new ServerBootstrap();

serverBootstrap.group(bossGroup).

---2.多线程

EventLoopGroup bossGroup = new NioEventLoopGroup(1);

EventLoopGroup workGroup = new NioEventLoopGroup();

try {

ServerBootstrap serverBootstrap = new ServerBootstrap();

serverBootstrap.group(bossGroup, workGroup).

---3.主从线程

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workGroup = new NioEventLoopGroup();

try {

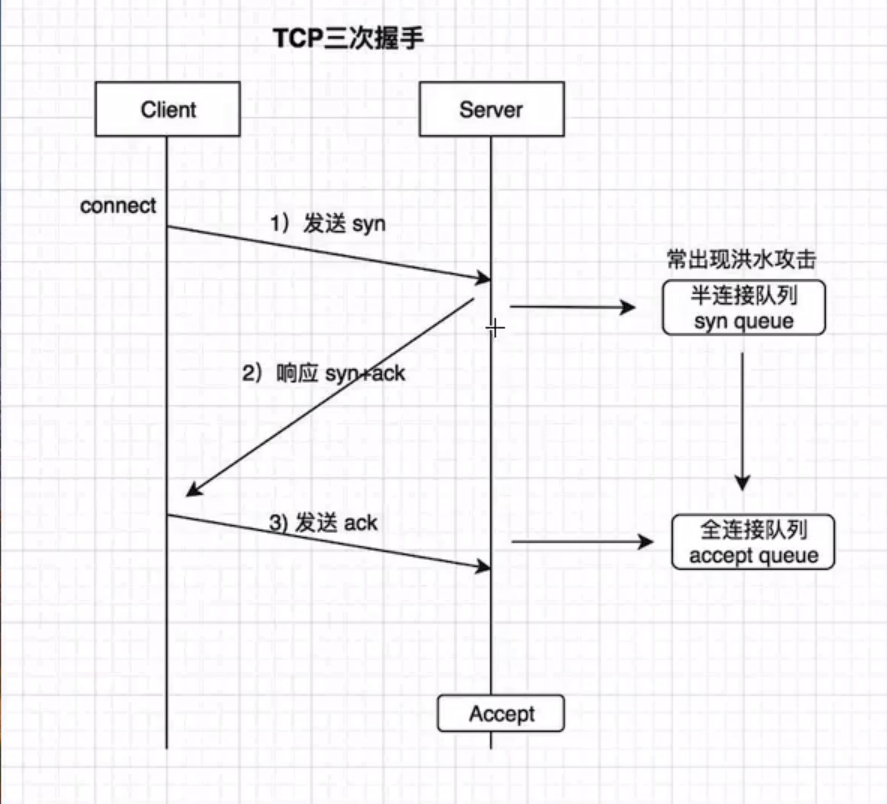
ServerBootstrap serverBootstrap = new ServerBootstrap();

serverBootstrap.group(bossGroup, workGroup).

--2.channel:设置channel通道类型NioServerSocketChannel

--3.option:作用于每个新建立的channel,设置tcp连接中的一些参数，如下：

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



ChannelOption.TCP\_NODELAY : 为了解决Nagle的算法问题，默认是false，要求实时性，有数据时马上发送，就设置true,关闭Nagle算法，如果减少发送次数，就设置为false,会累积到一定大小后再发送。

--4.ChildOption:作用于被accept之后的连接

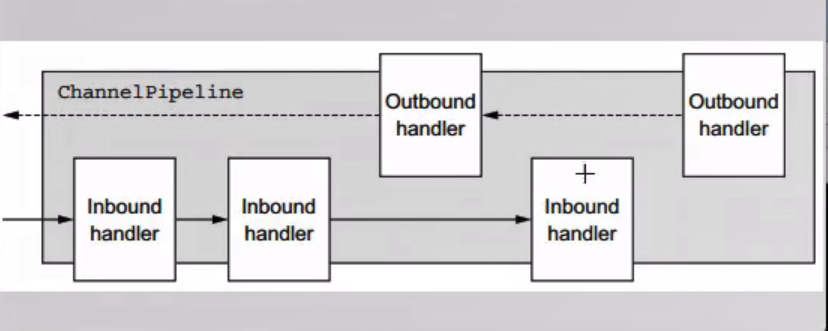
--5.ChildHandler：用于对每个通道里面的数据处理

15.客户端启动引导类BootStrap

--1.remoteAddress:服务端地址

--2.handler：和服务端通信的处理器

16.ChannelHandler和ChannnelPipeLine模块理解



--1.ChannelHandler中的方法

HandlerAdded:当ChannelHandler添加到ChannelPipeLine调用

HadlerRemoved：当ChannelHandler从ChannelPipeLine 移除时调用

ExceptionCaught：执行抛出异常时调用

ChannelHandler下主要有两个子接口:

---1.ChannelInboundHandler:(入站)

处理输入数据和Channel状态类型改变

适配器 ChannelInboundHandlerAdapter

常用的:SimpleChannelInboundHandler

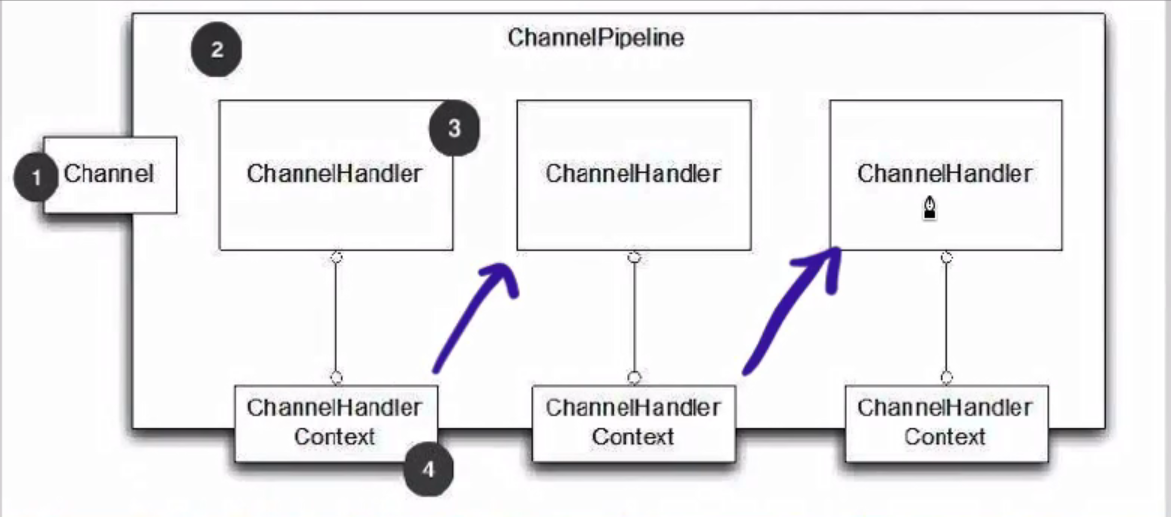
---2.ChannelOutboundHandler：出站

处理输出数据，适配器 ChannelOutboundHandlerAdapter

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。

17.ChannelHandlerContext理解



--1.ChannelHandlerContext是连接ChannelPipeLine和ChannelHandler的桥梁

ChannelHandlerContext部分方法和Channel及ChannelPipeLine重合，好比调用write方法

Channel，ChannelPipeLine，ChannelHandlerContext都可以调用此方法，前两者会在整个管道里传播，而ChannelHandlerContext就只会在后续的Handler传播。

--2.AbstractChannelHandlerContext类

双向链表结构,next/prev分别是后继节点和前驱节点

--3.DefaultChannelHandlerContext是实现类，但大部分都是父AbstractChannelHandlerContext完成，这个只是简单的实现一些方法，主要就是判断Handler类型

--4.ChannelInboundHandler之间的传递，主要通过调用ChannelHandlerContext中的firexxx()方法来实现下个handler的调用。

18.ChannelHandler的执行顺序

--1.一般的项目中，inboundHandler和outboundHandler有多个，在pipeLine中的执行顺序是：InboundHandler顺序执行，OutboundHandler逆序执行

--2.InboundHandler之间传递数据，通过ctx.fireChannelRead(msg)

--3.InboundHandler通过ctx.write(msg),传递到outboundHandler

--4.ctx.write(msg)传递消息，inbound需要放在结尾，在outbound之后，不然outboundHandler不会执行，但是使用channel.write(msg),或者pipeLine.write(msg),情况会不一样，都会执行

--5.outbound和inbound谁先执行，针对客户端和服务端而言，客户端是发送请求再接收数据，先outbound在inbound,服务端则相反。

19.ChannelFuture异步操作模块及注意事项

--1.Netty中的所有IO操作都是异步的，这意味着任何IO调用都会立即返回，而channelFuture会提供IO操作的结果、状态。

--2.ChannelFuture状态:

---1.未完成:当IO操作开始时，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取消，因为IO操作尚未完成。

---2.完成:当IO操作完成时，不管是失败、成功或取消,Futrue标记都是完成的，失败的时候也有具体的信息，例如失败原因，即便失败或取消也属于完成状态。

注意:不要在IO线程内(channelHandler中)调用sync或者await

--3.ChannelPromise:继承channelFuture，进一步拓展用于设置IO操作的结果

20.Netty网络数据传输编解码

最开始接触的编解码：java序列化、反序列化(就是编解码),url编码，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，也就是数据包太大

--3.序列化和反序列化性能差

业界其他编解码框架:google的protobuf,……………….

Netty里面的编解码：

解码器：负责处理“入站 inboundHandler”数据

编码器:负责处理“出站 outboundHandler”数据

Netty里面提供默认的编解码器，也支持自定义编解码器

Encoder:编码器

Decoder:解码器

Codec：编解码器

21.Netty解码器Decoder

Decoder对应的就是channelInboundHandler，主要就是字节数组转为消息对象

主要两个方法:

Decode：一般用这个

DecodeLast：用于最后的几个字节处理，也就是channel关闭的时候，产生的最后一个消息

抽象解码器：

--1.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

--2.ReplayingDecoder:继承ByteToMessageDecoder,不需要检查缓冲区有足够的字节，速度略慢于ByteToMessageDecoder

选择：项目复杂性高则使用ReplayingDecoder,否则使用ByteToMessageDecoder

--3.MessageToMessageDecoder:从一种消息解码到另一种消息（例如POJO到POJO）

解码器的具体实现（主要解决TCP底层的粘包和拆包问题），用的比较多的：

---1.DelimiterBasedFrameDecoder:指定消息分隔符的解码器 1111####2222

---2.LineBasedFrameDecoder:换行符结束标志

---3.FixedLengthFrameDecoder:固定长度

---4.LengthFieldBasedFrameDecoder:message=header+body，基于长度解码的通用解码器

---5.StringDecoder:文本解码，将接收到的消息转换为字符串，一般会与上面的几种进行组合，然后在后面加业务的handler。

22.Netty编码器Encoder

Encoder对应的就是channeloutboundHandler,消息对象转换为字节数组

Netty本身未提供和解码一样的编码器，是因为场景不同，两者非对等的

--1.MessageToByteEncoder

消息转为字节数组，调用write方法，会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

--2.MessageToMessageEncoder

用于从一种消息编码为另一种消息(pojo到pojo)

23.Netty编解码器Codec

组合解码器和编码器，提供对于字节和消息都相同的操作

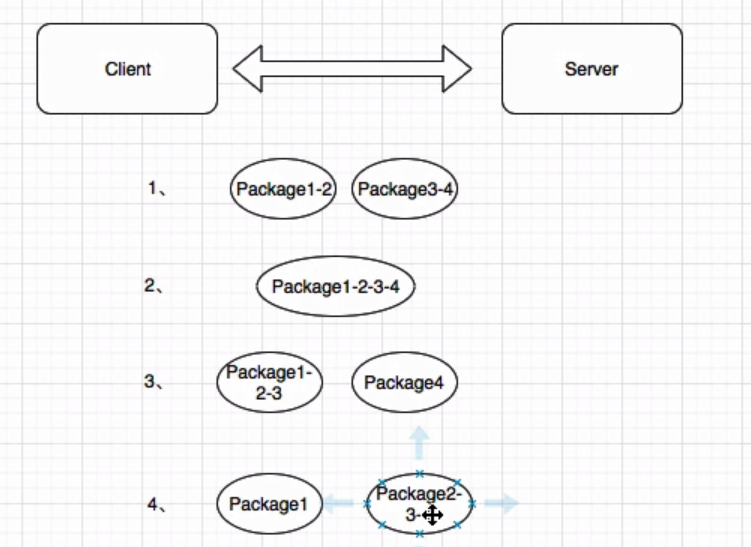
优点：成对出现，编解码都是在一个类里面完成

缺点：耦合在一起，拓展性不佳

24.网络传输TCP粘包拆包

--1.TCP拆包:一个完整的包可能会被TCP拆分为多个包进行发送

--2.TCP粘包:把多个小的包封装成一个大的数据包，client发送的若干数据包，Server接收时粘成一包



--3.原因分析

---1.发送方原因:TCP默认使用nagle算法

---2.接收方原因:TCP接收到数据放置缓存中，应用程序从缓存中读取

补充:UDP没有粘包和拆包的问题，有边界协议

25.TCP半包读写常见的解决方法

--1.发送方:可以关闭nagle算法

--2.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

--3.应用层解决半包读写的方法:

---1.设置定长消息

---2.设置消息的边界(分隔符)

---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

26.Netty自带解决TCP半包读写方案

--1.DelimiterBasedFrameDecoder:指定消息分隔符的解码器

--2.LineBasedFrameDecoder:以换行符为结束标志的解码器

--3.FixedLengthFrameDecoder:固定长度解码器

--4.LengthFieldBasedFrameDecoder：message=header+body,基于长度解码的通用解码器

27.Netty核心模块缓冲ByteBuf

--1.JDK ByteBuffer

共用读写索引，每次读写操作需要flip()

扩容麻烦，容易造成浪费

--2.Netty ByteBuf

读写使用不同的索引，操作便捷

自动扩容，使用便捷

