**NIO&Netty**

一、**BIO和NIO和AIO**

**1 同步&异步&阻塞&非阻塞**

**同步 ：** 自己亲自出马持银行卡到银行取钱（使用同步IO时，Java自己处理IO读写）；

**异步 ：** 委托一小弟拿银行卡到银行取钱，然后给你（使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS(银行卡和密码)，OS需要支持异步IO操作API）；

**阻塞 ：** ATM排队取款，你只能等待（使用阻塞IO时，Java调用会一直阻塞到读写完成才返回）；

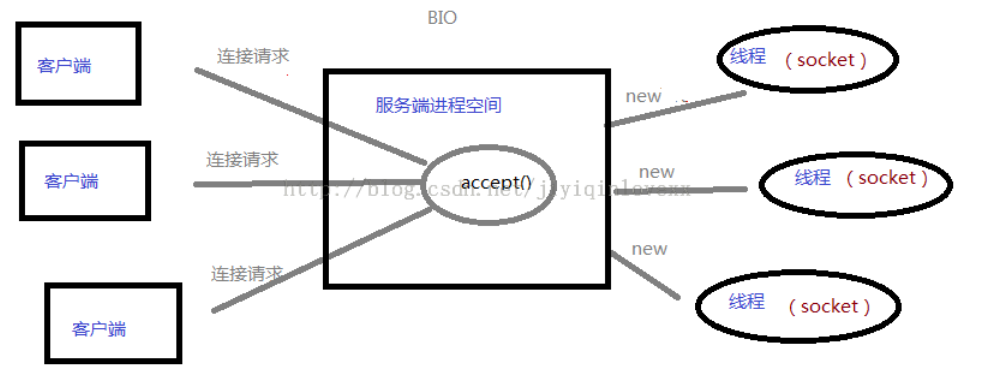
**非阻塞 ：** 柜台取款，取个号，然后坐在椅子上做其它事，等号广播会通知你办理，没到号你就不能去，你可以不断问大堂经理排到了没有，大堂经理如果说还没到你就不能去

（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

**2 IO的方式通常分为几种，同步阻塞的BIO、同步非阻塞的NIO、异步非阻塞的AIO。**

**3 BIO**

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，**默认情况下服务端需要对每个请求建立一堆线程等待请求**，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。

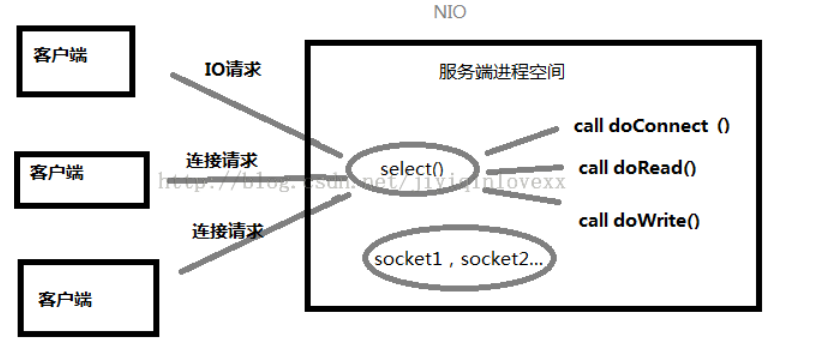


**4 NIO**

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题：在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

**NIO的最重要的地方是当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定**，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。

在NIO的处理方式中，当一个请求来的话，开启线程进行处理，可能会等待后端应用的资源(JDBC连接等)，其实这个线程就被阻塞了，当并发上来的话，还是会有BIO一样的问题。



**5 AIO**

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。 即可以理解为，**read/write方法都是异步的，完成后会主动调用回调函数。**  在JDK1.7中，这部分内容被称作NIO.2，主要在java.nio.channels包下增加了下面四个异步通道：

AsynchronousSocketChannel

AsynchronousServerSocketChannel

AsynchronousFileChannel

AsynchronousDatagramChannel

其中的read/write方法，会返回一个带回调函数的对象，当执行完读取/写入操作后，直接调用回调函数。

**6 Java对BIO、NIO、AIO的支持**

Java BIO ： 同步并阻塞，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，当然可以通过线程池机制改善。

Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，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请求时才启动一个线程进行处理。

Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

**7 BIO、NIO、AIO适用场景分析**

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，但程序直观简单易理解。

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，并发局限于应用中，编程比较复杂，JDK1.4开始支持。服务器需要支持超大量的长时间连接。比如10000个连接以上，并且每个客户端并不会频繁地发送太多数据。例如总公司的一个中心服务器需要收集全国便利店各个收银机的交易信息，**只需要少量线程按需处理维护的大量长期连接。Jetty、Mina、Netty、ZooKeeper等都是基于NIO方式实现。**

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。

**8 NIO三种模型**

**（1）Reactor单线程模型：**

单个线程完成所有事情包括接收客户端的TCP连接请求，读取和写入套接字数据等。对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发的应用却不合适，主要原因如下：

1) 一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；

2) 当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，NIO线程会成为系统的性能瓶颈；

3) 可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

为了解决这些问题，演进出了Reactor多线程模型。

**（2）Reactor多线程模型：**

Rector多线程模型与单线程模型最大的区别就是有一组NIO线程处理真实的IO操作。

1) 有专门一个NIO线程-Acceptor线程用于监听服务端，接收客户端的TCP连接请求；

2) 网络IO操作-读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、解码、编码和发送；

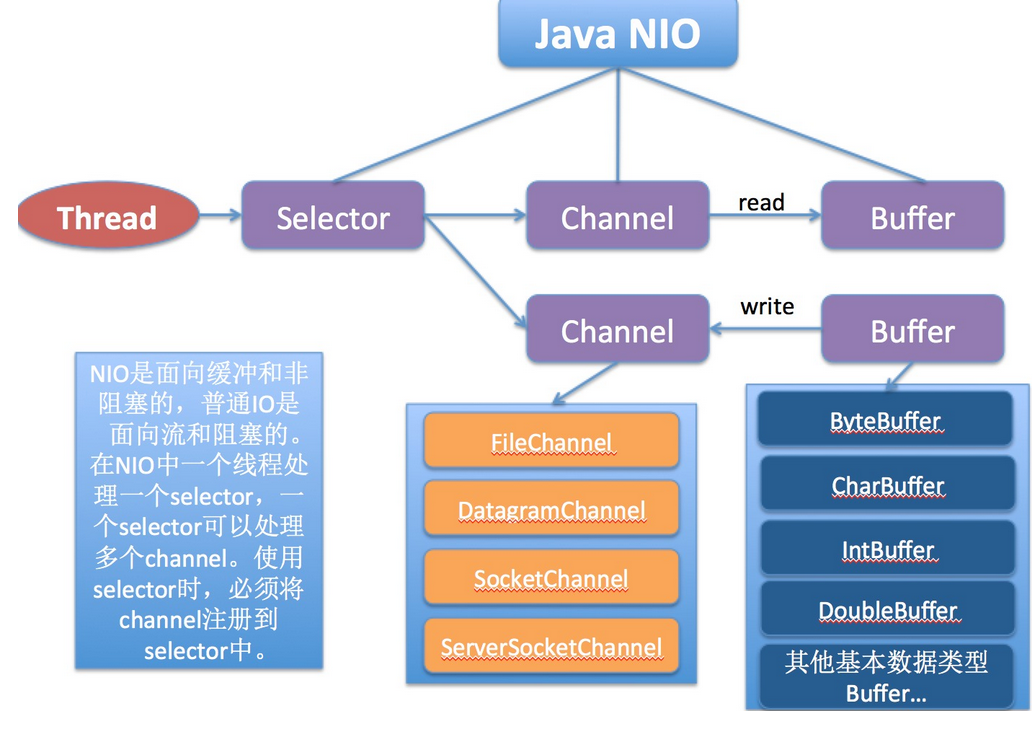
3) 1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题。

在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型-主从Reactor多线程模型。即从单线程中由一个线程即监听连接事件、读写事件、由完成数据读写，拆分为由一个线程专门监听各种事件，再由专门的线程池负责处理真正的IO数据读写。

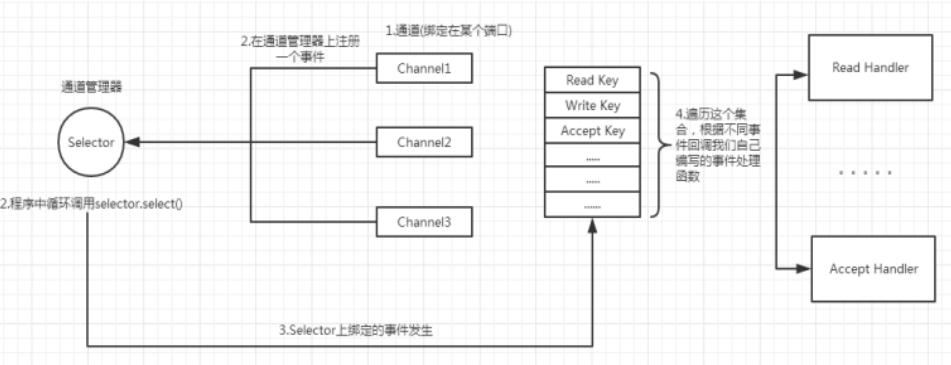
**（3）主从Reactor多线程模型**

主从Reactor线程模型与Reactor多线程模型的最大区别就是有一组NIO线程处理连接、读写事件。主从Reactor线程模型的特点是：服务端用于接收客户端连接的不再是个1个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端TCP连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到IO线程池（sub reactor线程池）的某个IO线程上，由它负责SocketChannel的读写和编解码工作。Acceptor线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作。

即从多线程模型中由一个线程来监听连接事件和数据读写事件，拆分为一个线程监听连接事件，线程池的多个线程监听已经建立连接的套接字的数据读写事件，另外和多线程模型一样有专门的线程池处理真正的IO操作。



**9 原生NIO开发大致流程**



**本文大致讲述了使用NIO进行服务器端开发的大致流程，但代码显然仍然存在问题，其一是我们只使用了一个线程执行所有操作，包括接收客户端连接，读取数据，返回数据，对于这个简单的Demo来说已经足够了，但在实际的服务器开发中，例如你想使用NIO开发自己的HTTP服务器，服务器本地需要做大量操作，包括解析用户请求，根据请求路由到某一个Action执行业务逻辑，这其中又很可能某些数据从数据库读取，渲染模板等操作，十分耗时，这无疑又称为系统的瓶颈，再者，使用单一线程不能充分利用多核CPU提供的计算能力。下一篇中会看到，在基于Reactor模型的Netty中，会使用一个Boss线程接收客户端请求，使用多个Worker线程执行具体的业务逻辑。**

**10 NIO主要有三大核心部分：Channel(通道)，Buffer(缓冲区), Selector。**

Channel

国内大多翻译成“通道”。Channel和IO中的Stream(流)是差不多一个等级的。只不过Stream是单向的，譬如：InputStream, OutputStream.而Channel是双向的，既可以用来进行读操作，又可以用来进行写操作。

NIO中的Channel的主要实现有：

**FileChannel**

**DatagramChannel**

**SocketChannel**

**ServerSocketChannel**

这里看名字就可以猜出个所以然来：分别可以对应文件IO、UDP和TCP（Server和Client）。下面演示的案例基本上就是围绕这4个类型的Channel进行陈述的。

**Buffer**

NIO中的关键Buffer实现有：ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer，

分别对应基本数据类型: byte, char, double, float, int, long, short。

当然NIO中还有MappedByteBuffer, HeapByteBuffer, DirectByteBuffer等这里先不进行陈述。

**Selector**

Selector运行单线程处理多个Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用Selector就会很方便。例如在一个聊天服务器中。要使用Selector, 得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新的连接进来、数据接收等。

**11 FileChannel**

首先，案例1是采用FileInputStream读取文件内容的：

public static void method2(){

InputStream in = null;

try{

in = new BufferedInputStream(new FileInputStream("src/nomal\_io.txt"));

byte [] buf = new byte[1024];

int bytesRead = in.read(buf);

while(bytesRead != -1)

{

for(int i=0;i<bytesRead;i++)

System.out.print((char)buf[i]);

bytesRead = in.read(buf);

}

}catch (IOException e)

{

e.printStackTrace();

}finally{

try{

if(in != null){

in.close();

}

}catch (IOException e){

e.printStackTrace();

}

}

}

案例是对应的NIO（这里通过RandomAccessFile进行操作，当然也可以通过FileInputStream.getChannel()进行操作）：

public static void method1(){

RandomAccessFile aFile = null;

try{

aFile = new RandomAccessFile("src/nio.txt","rw");

FileChannel fileChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocate(1024);

int bytesRead = fileChannel.read(buf);

System.out.println(bytesRead);

while(bytesRead != -1)

{

buf.flip();

while(buf.hasRemaining())

{

System.out.print((char)buf.get());

}

buf.compact();

bytesRead = fileChannel.read(buf);

}

}catch (IOException e){

e.printStackTrace();

}finally{

try{

if(aFile != null){

aFile.close();

}

}catch (IOException e){

e.printStackTrace();

}

}

}

输出结果：（略）

通过仔细对比案例1和案例2，应该能看出个大概，最起码能发现NIO的实现方式比叫复杂。有了一个大概的印象可以进入下一步了。

Buffer的使用一般遵循下面几个步骤：

@分配空间（ByteBuffer buf = ByteBuffer.allocate(1024); 还有一种allocateDirector后面再陈述）

@写入数据到Buffer(int bytesRead = fileChannel.read(buf);)

@调用filp()方法（ buf.flip();）

@从Buffer中读取数据（System.out.print((char)buf.get());）

@调用clear()方法或者compact()方法

向Buffer中写数据：

从Channel写到Buffer (fileChannel.read(buf))

通过Buffer的put()方法 （buf.put(…)）

从Buffer中读取数据：

从Buffer读取到Channel (channel.write(buf))

使用get()方法从Buffer中读取数据 （buf.get()）

**12 SocketChannel**

NIO的channel抽象的一个重要特征就是可以通过配置它的阻塞行为，以实现非阻塞式的信道。

**channel.configureBlocking(false)**

在非阻塞式信道上调用一个方法总是会立即返回。这种调用的返回值指示了所请求的操作完成的程度。例如，在一个非阻塞式ServerSocketChannel上调用accept()方法，如果有连接请求来了，则返回客户端SocketChannel，否则返回null。这里先举一个TCP应用案例，客户端采用NIO实现，而服务端依旧使用IO实现。

**13 客户端代码（案例3）：**

public static void client(){

ByteBuffer buffer = ByteBuffer.allocate(1024);

SocketChannel socketChannel = null;

try

{

socketChannel = SocketChannel.open();

socketChannel.configureBlocking(false);

socketChannel.connect(new InetSocketAddress("10.10.195.115",8080));

if(socketChannel.finishConnect())

{

int i=0;

while(true)

{

TimeUnit.SECONDS.sleep(1);

String info = "I'm "+i+++"-th information from client";

buffer.clear();

buffer.put(info.getBytes());

buffer.flip();

while(buffer.hasRemaining()){

System.out.println(buffer);

socketChannel.write(buffer);

}

}

}

}

catch (IOException | InterruptedException e)

{

e.printStackTrace();

}

finally{

try{

if(socketChannel!=null){

socketChannel.close();

}

}catch(IOException e){

e.printStackTrace();

}

}

}

**14 服务端代码（案例4）：**

public class ServerConnect

{

private static final int BUF\_SIZE=1024;

private static final int PORT = 8080;

private static final int TIMEOUT = 3000;

public static void main(String[] args)

{

selector();

}

public static void handleAccept(SelectionKey key) throws IOException{

ServerSocketChannel ssChannel = (ServerSocketChannel)key.channel();

SocketChannel sc = ssChannel.accept();

sc.configureBlocking(false);

sc.register(key.selector(), SelectionKey.OP\_READ,ByteBuffer.allocateDirect(BUF\_SIZE));

}

public static void handleRead(SelectionKey key) throws IOException{

SocketChannel sc = (SocketChannel)key.channel();

ByteBuffer buf = (ByteBuffer)key.attachment();

long bytesRead = sc.read(buf);

while(bytesRead>0){

buf.flip();

while(buf.hasRemaining()){

System.out.print((char)buf.get());

}

System.out.println();

buf.clear();

bytesRead = sc.read(buf);

}

if(bytesRead == -1){

sc.close();

}

}

public static void handleWrite(SelectionKey key) throws IOException{

ByteBuffer buf = (ByteBuffer)key.attachment();

buf.flip();

SocketChannel sc = (SocketChannel) key.channel();

while(buf.hasRemaining()){

sc.write(buf);

}

buf.compact();

}

public static void selector() {

Selector selector = null;

ServerSocketChannel ssc = null;

try{

selector = Selector.open();

ssc= ServerSocketChannel.open();

ssc.socket().bind(new InetSocketAddress(PORT));

ssc.configureBlocking(false);

ssc.register(selector, SelectionKey.OP\_ACCEPT);

while(true){

if(selector.select(TIMEOUT) == 0){

System.out.println("==");

continue;

}

Iterator<SelectionKey> iter = selector.selectedKeys().iterator();

while(iter.hasNext()){

SelectionKey key = iter.next();

if(key.isAcceptable()){

handleAccept(key);

}

if(key.isReadable()){

handleRead(key);

}

if(key.isWritable() && key.isValid()){

handleWrite(key);

}

if(key.isConnectable()){

System.out.println("isConnectable = true");

}

iter.remove();

}

}

}catch(IOException e){

e.printStackTrace();

}finally{

try{

if(selector!=null){

selector.close();

}

if(ssc!=null){

ssc.close();

}

}catch(IOException e){

e.printStackTrace();

}

}

}

}

## 二、netty

**1 Netty 是一个基于 JAVA NIO 类库的异步通信框架**，它的架构特点是：异步非阻塞、基于事件驱动、高性能、高可靠性和高可定制性。换句话说，Netty是一个NIO框架，使用它可以简单快速地开发网络应用程序，比如客户端和服务端的协议。Netty大大简化了网络程序的开发过程比如TCP和UDP的 Socket的开发。Netty 已逐渐成为 Java NIO 编程的首选框架

**Netty是最流行的NIO框架，它的健壮性、功能、性能、可定制性和可扩展性在同类框架都是首屈一指的。它已经得到成百上千的商业/商用项目验证，如Hadoop的RPC框架Avro、RocketMQ以及主流的分布式通信框架Dubbox等等。**

**2 Netty 的优点**

**API 使用简单**，开发门槛低；

**功能强大，**预置了多种编解码功能，支持多种主流协议；

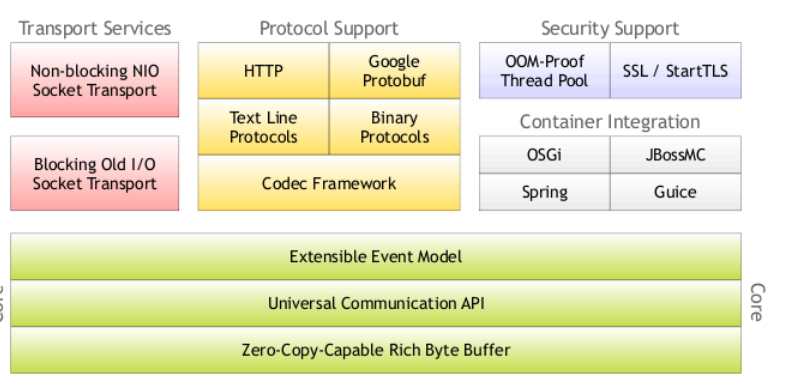
**定制能力强**，可以通过 ChannelHandler 对通信框架进行灵活的扩展；

**性能高**，通过与其它业界主流的 NIO 框架对比，Netty 的综合性能最优；

**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；

**经历了大规模的商业应用考验，质量得到验证**。在互联网、大数据、网络游戏、企业应用、电信软件等众多行业得到成功商用，证明了它完全满足不同行业的商用标准。

**3 Netty架构组成**



**4 NIO的通信步骤**

①创建ServerSocketChannel，为其配置非阻塞模式。

②绑定监听，配置TCP参数，录入backlog大小等。

③创建一个独立的IO线程，用于轮询多路复用器Selector。

④创建Selector，将之前创建的ServerSocketChannel注册到Selector上，并设置监听标识位SelectionKey.OP\_ACCEPT。

⑤启动IO线程，在循环体中执行Selector.select()方法，轮询就绪的通道。

⑥当轮询到处于就绪状态的通道时，需要进行操作位判断，如果是ACCEPT状态，说明是新的客户端接入，则调用accept方法接收新的客户端。

⑦设置新接入客户端的一些参数，如非阻塞，并将其继续注册到Selector上，设置监听标识位等。

⑧如果轮询的通道标识位是READ，则进行读取，构造Buffer对象等。

⑨更细节的问题还有数据没发送完成继续发送的问题......

**5 Netty通信的步骤**

①创建两个NIO线程组，一个专门用于网络事件处理（接受客户端的连接），另一个则进行网络通信的读写。

②创建一个ServerBootstrap对象，配置Netty的一系列参数，例如接受传出数据的缓存大小等。

③创建一个用于实际处理数据的类ChannelInitializer，进行初始化的准备工作，比如设置接受传出数据的字符集、格式以及实际处理数据的接口。

④绑定端口，执行同步阻塞方法等待服务器端启动即可。

**6 TCP粘包、拆包问题**

熟悉TCP编程的可能都知道，无论是服务器端还是客户端，当我们读取或者发送数据的时候，都需要考虑TCP底层的粘包/拆包机制。TCP是一个“流”协议，所谓流就是没有界限的遗传数据。大家可以想象一下，如果河水就好比数据，他们是连成一片的，没有分界线，TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的具体情况进行包的划分，也就是说，在业务上一个完整的包可能会被TCP分成多个包进行发送，也可能把多个小包封装成一个大的数据包发送出去，这就是所谓的粘包/拆包问题。

解决方案：

①消息定长，例如每个报文的大小固定为200个字节，如果不够，空位补空格。

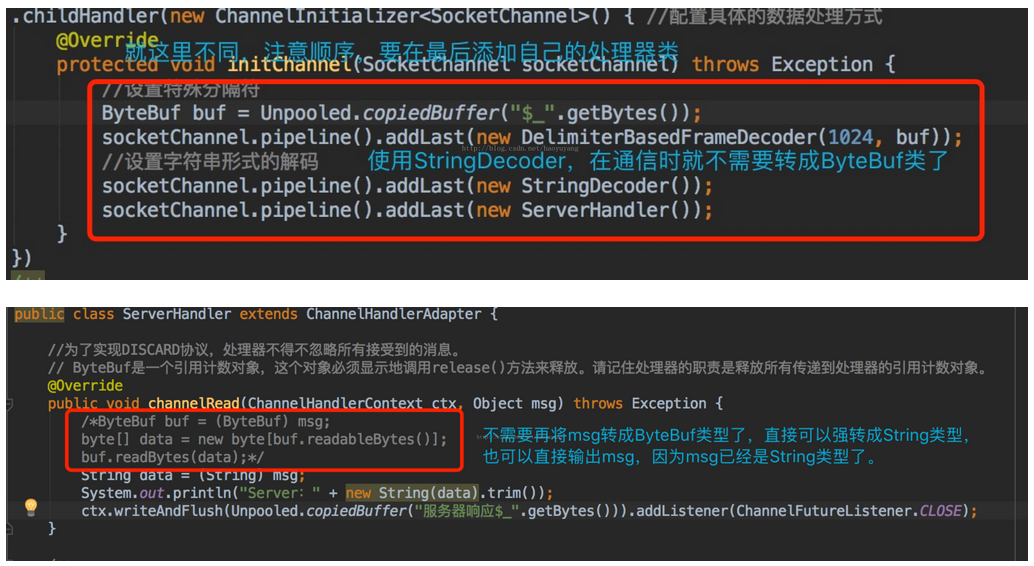
②在包尾部增加特殊字符进行分割，例如加回车等。

③将消息分为消息头和消息体，在消息头中包含表示消息总长度的字段，然后进行业务逻辑的处理。

Netty中解决TCP粘包/拆包的方法：

①分隔符类：DelimiterBasedFrameDecoder（自定义分隔符）

②定长：FixedLengthFrameDecoder



## 三、Netty最佳实战

**1 数据通信**

我们需要了解在真正项目中如何使用Netty，大体上对于一些参数设置都是根据服务器性能决定的。我们需要考虑的问题是两台机器（甚至多台）使用Netty怎样进行通信。

**大体上分为三种**

①使用长连接通道不断开的形式进行通信，也就是服务器和客户端的通道一直处于开启状态，如果服务器性能足够好，并且客户端数量也比较上的情况下，推荐这种方式。

②一次性批量提交数据，采用短连接方式。也就是说先把数据保存到本地临时缓存区或者临时表，当达到界值时进行一次性批量提交，又或者根据定时任务轮询提交，这种情况的弊端是做不到实时性传输，对实时性要求不高的应用程序中推荐使用。

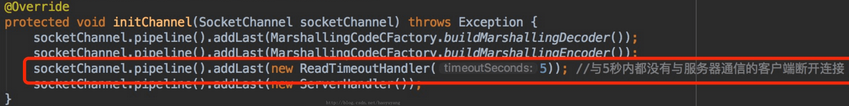
③使用一种特殊的长连接，在某一指定时间段内，服务器与某台客户端没有任何通信，则断开连接。下次连接则是客户端向服务器发送请求的时候，再次建立连接。

**使用Netty实现第三种方式的连接，但是我们需要考虑两个因素**：

①如何在超时（即服务器和客户端没有任何通信）后关闭通道？关闭通道后又如何再次建立连接？

②客户端宕机时，我们无需考虑，下次重启客户端之后就可以与服务器建立连接，但服务器宕机时，客户端如何与服务器端通信

**2 服务端**





**3 客户端**：

**public** **class** ClientChannelTimeoutInitializer **extends** ChannelInitializer<SocketChannel>

{

// 配置具体的数据处理方式

@Override

**protected** **void** initChannel(SocketChannel socketChannel) **throws** Exception

{

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

socketChannel.pipeline().addLast(**new** ReadTimeoutHandler(5)); //5秒后未与服务器通信，则断开连接。

socketChannel.pipeline().addLast(**new** ClientHandler());

}

}

**public** **class** DataCommunicationClient

{

**private** **static** **class** SingleHodler

{

**static** **final** DataCommunicationClient ***DataCommunicationClient*** = **new** DataCommunicationClient();

}

**public** **static** DataCommunicationClient getInstance()

{

**return** SingleHodler.***DataCommunicationClient***;

}

**private** EventLoopGroup workerGroup;

**private** Bootstrap bootstrap;

**private** ChannelFuture future;

**private** DataCommunicationClient()

{

workerGroup = **new** NioEventLoopGroup();

bootstrap = **new** Bootstrap();

bootstrap.group(workerGroup) .channel(NioSocketChannel.**class**).handler(**new** ClientChannelTimeoutInitializer());

}

**public** **void** connect()

{

**try**

{

future = bootstrap.connect("127.0.0.1", 8765).sync();

}

**catch** (InterruptedException e)

{

e.printStackTrace();

}

}

**public** ChannelFuture getFuture()

{

**if** (future == **null** || !future.channel().isActive())

{

**this**.connect();

}

**return** future;

}

**public** **static** **void** main(String[] args) **throws** InterruptedException

{

DataCommunicationClient dataCommunicationClient = *getInstance*();

ChannelFuture future = dataCommunicationClient.getFuture();

**for** (**int** i = 1; i <= 3; i++)

{

Message message = **new** Message(i, "pro" + i, "数据信息" + i);

future.channel().writeAndFlush(message);

Thread.*sleep*(4000); // 休眠4秒后再发送数据

}

future.channel().closeFuture().sync();

**new** Thread(() ->

{

**try**

{

System.***out***.println("子线程开始....");

ChannelFuture f = dataCommunicationClient.getFuture();

Message message = **new** Message(4, "pro" + 4, "数据信息" + 4);

f.channel().writeAndFlush(message);

f.channel().closeFuture().sync();

} **catch** (Exception e)

{

e.printStackTrace();

}

}).start();

System.***out***.println("主线程退出......");

}

}

**4 心跳检测**

我们使用Socket通信一般经常会处理多个服务器之间的心跳检测，一般来讲我们去维护服务器集群，肯定要有一台或多台服务器主机（Master），然后还应该有N台（Slave），那么我们的主机肯定要时时刻刻知道自己下面的从服务器的各方面情况，然后进行实时监控的功能。这个在分布式架构里交做心跳检测或者心跳监控。最佳处理方案是使用一些通信框架进行实现，Netty就可以做这样的事。

package com.hiya.nio.netty.heartBeat;

import java.net.InetSocketAddress;

import io.netty.bootstrap.Bootstrap;

import io.netty.channel.ChannelFuture;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.nio.NioSocketChannel;

public class HeartBeatClient

{

public static void main(String[] args)

{

EventLoopGroup workerGroup = new NioEventLoopGroup();

try

{

Bootstrap bootstrap = new Bootstrap();

bootstrap.group(workerGroup).channel(NioSocketChannel.class).handler(new HeartBeatClientInitializer());

ChannelFuture future = bootstrap.connect(new InetSocketAddress("127.0.0.1", 8765)).sync();

future.channel().closeFuture().sync();

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

workerGroup.shutdownGracefully();

}

}

}

package com.hiya.nio.netty.heartBeat;

import java.net.InetAddress;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

import io.netty.util.ReferenceCountUtil;

import io.netty.util.concurrent.ScheduledFuture;

public class HeartBeatClientHandler extends ChannelHandlerAdapter

{

private ScheduledExecutorService scheduled = Executors.newScheduledThreadPool(1);

private ScheduledFuture<?> heartBeat;

private InetAddress address;

private static final String SUCCESS\_KEY = "auth\_success\_key";

public void channelActive(ChannelHandlerContext ctx) throws Exception

{

address = InetAddress.getLocalHost();

String ip = address.getHostAddress();

String key = "1234";

String auth = ip + "," + key;

ctx.writeAndFlush(auth);

}

@Override

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

{

cause.printStackTrace();

if (heartBeat != null)

{

heartBeat.cancel(true);

heartBeat = null;

}

ctx.fireExceptionCaught(cause);

}

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception

{

try

{

if (msg instanceof String)

{

String data = (String) msg;

if (SUCCESS\_KEY.equals(data))

{

heartBeat = (ScheduledFuture<?>) scheduled.scheduleWithFixedDelay(new HeartBeatClientTask(ctx), 0, 5, TimeUnit.SECONDS);

System.out.println(msg);

} else

{

System.out.println(msg);

}

}

} finally

{

ReferenceCountUtil.release(msg);

}

}

}

package com.hiya.nio.netty.heartBeat;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.socket.SocketChannel;

public class HeartBeatClientInitializer extends ChannelInitializer<SocketChannel>

{

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception

{

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

socketChannel.pipeline().addLast(new HeartBeatClientHandler());

}

}

package com.hiya.nio.netty.heartBeat;

import java.net.InetAddress;

import java.net.UnknownHostException;

import java.util.HashMap;

import java.util.Map;

import org.hyperic.sigar.CpuPerc;

import org.hyperic.sigar.Mem;

import org.hyperic.sigar.Sigar;

import org.hyperic.sigar.SigarException;

import io.netty.channel.ChannelHandlerContext;

public class HeartBeatClientTask implements Runnable

{

private final ChannelHandlerContext ctx;

InetAddress address = null;

public HeartBeatClientTask(ChannelHandlerContext ctx) throws UnknownHostException

{

this.ctx = ctx;

address = InetAddress.getLocalHost();

}

@Override

public void run()

{

try

{

RequestInfo requestInfo = new RequestInfo();

requestInfo.setIp(address.getHostAddress());

Sigar sigar = new Sigar();

CpuPerc cpuPerc = sigar.getCpuPerc();

Map<String, Object> cpuPercMap = new HashMap<>();

cpuPercMap.put("combined", cpuPerc.getCombined());

cpuPercMap.put("user", cpuPerc.getUser());

cpuPercMap.put("sys", cpuPerc.getSys());

cpuPercMap.put("wait", cpuPerc.getWait());

cpuPercMap.put("idle", cpuPerc.getIdle());

Mem mem = sigar.getMem();

Map<String, Object> memoryMap = new HashMap<>();

memoryMap.put("total", mem.getTotal() / (1024 \* 1024));

memoryMap.put("used", mem.getUsed() / (1024 \* 1024));

memoryMap.put("free", mem.getFree() / (1024 \* 1024));

requestInfo.setCpuPercMap(cpuPercMap);

requestInfo.setMemoryMap(memoryMap);

ctx.writeAndFlush(requestInfo);

} catch (Exception e)

{

e.printStackTrace();

}

}

/\*\*

\*

JAVA web项目报错no sigar-x86-winnt.dll in java.library.path

no sigar-amd64-winnt.dll in java.library.path

org.hyperic.sigar.SigarException: no sigar-amd64-winnt.dll in java.library.path

at org.hyperic.sigar.Sigar.loadLibrary(Sigar.java:172)

at org.hyperic.sigar.Sigar.<clinit>(Sigar.java:100)

解决此报错：

一、需要下载hyperic-sigar-1.6.2.zip，提前hyperic-sigar-1.6.2\sigar-bin\lib下的sigar-x86-winnt.dll，然后放到JAVA JDK安装目录下的bin目录下即可。

二、需要注意的是：本次调试使用的KDE是myeclipse 8.5，其自带有JDK 1.6，默认情况下Myeclipse 8.5运行不使用外部安装的JDK，需要将sigar-x86-winnt.dll放到Myeclipse 8.5自带JDK环境下的bin目录下，如果项目编译使用手动安装的外部JDK则将sigar-x86-winnt.dll放入外部JDK的bin目录下。

\* @param args

\* @throws UnknownHostException

\* @throws SigarException

\*/

public static void main(String[] args) throws UnknownHostException, SigarException

{

InetAddress address = InetAddress.getLocalHost();

RequestInfo requestInfo = new RequestInfo();

requestInfo.setIp(address.getHostAddress());

Sigar sigar = new Sigar();

CpuPerc cpuPerc = sigar.getCpuPerc();

Map<String, Object> cpuPercMap = new HashMap<>();

cpuPercMap.put("combined", cpuPerc.getCombined());

cpuPercMap.put("user", cpuPerc.getUser());

cpuPercMap.put("sys", cpuPerc.getSys());

cpuPercMap.put("wait", cpuPerc.getWait());

cpuPercMap.put("idle", cpuPerc.getIdle());

Mem mem = sigar.getMem();

Map<String, Object> memoryMap = new HashMap<>();

memoryMap.put("total", mem.getTotal() / (1024 \* 1024));

memoryMap.put("used", mem.getUsed() / (1024 \* 1024));

memoryMap.put("free", mem.getFree() / (1024 \* 1024));

requestInfo.setCpuPercMap(cpuPercMap);

requestInfo.setMemoryMap(memoryMap);

System.out.println(address.getHostAddress());

System.out.println(cpuPercMap);

System.out.println(memoryMap);

}

}

package com.hiya.nio.netty.heartBeat;

import java.net.InetSocketAddress;

import io.netty.bootstrap.ServerBootstrap;

import io.netty.channel.ChannelFuture;

import io.netty.channel.ChannelOption;

import io.netty.channel.EventLoopGroup;

import io.netty.channel.nio.NioEventLoopGroup;

import io.netty.channel.socket.nio.NioServerSocketChannel;

import io.netty.handler.logging.LogLevel;

import io.netty.handler.logging.LoggingHandler;

public class HeartBeatServer

{

public HeartBeatServer(int port)

{

EventLoopGroup bossGroup = new NioEventLoopGroup();

EventLoopGroup workerGroup = new NioEventLoopGroup();

try

{

ServerBootstrap bootstrap = new ServerBootstrap();

bootstrap.group(bossGroup, workerGroup)

.channel(NioServerSocketChannel.class)

.childHandler(new HeartBeatServerInitializer())

.handler(new LoggingHandler(LogLevel.INFO))

.option(ChannelOption.SO\_BACKLOG, 1024);

ChannelFuture future = bootstrap.bind(new InetSocketAddress("127.0.0.1", port)).sync();

future.channel().closeFuture().sync();

}

catch (Exception e)

{

e.printStackTrace();

}

finally

{

bossGroup.shutdownGracefully();

workerGroup.shutdownGracefully();

}

}

public static void main(String[] args)

{

new HeartBeatServer(8765);

}

}

package com.hiya.nio.netty.heartBeat;

import java.util.HashMap;

import java.util.Map;

import io.netty.channel.ChannelFutureListener;

import io.netty.channel.ChannelHandlerAdapter;

import io.netty.channel.ChannelHandlerContext;

public class HeartBeatServerHandler extends ChannelHandlerAdapter

{

private static Map<String, String> AUTH\_IP\_MAP = new HashMap<>();

private static final String SUCCESS\_KEY = "auth\_success\_key";

static

{

AUTH\_IP\_MAP.put("192.168.3.176", "1234");

}

private boolean auth(ChannelHandlerContext ctx, Object msg)

{

String[] rets = ((String) msg).split(",");

String auth = AUTH\_IP\_MAP.get(rets[0]);

if (auth != null && auth.equals(rets[1]))

{

ctx.writeAndFlush(SUCCESS\_KEY);

return true;

} else

{

ctx.writeAndFlush("authfailure!").addListener(ChannelFutureListener.CLOSE);

return false;

}

}

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception

{

if (msg instanceof String)

{

auth(ctx, msg);

} else if (msg instanceof RequestInfo)

{

RequestInfo info = (RequestInfo) msg;

System.out.println("----------------------------------------------");

System.out.println("当前主机ip：" + info.getIp());

System.out.println("当前主机cpu：情况");

Map<String, Object> cpuMap = info.getCpuPercMap();

System.out.println("总使用率：" + cpuMap.get("combined"));

System.out.println("用户使用率：" + cpuMap.get("user"));

System.out.println("系统使用率：" + cpuMap.get("sys"));

System.out.println("等待率：" + cpuMap.get("wait"));

System.out.println("空闲率：" + cpuMap.get("idle"));

System.out.println("当前主机memory情况：");

Map<String, Object> memMap = info.getMemoryMap();

System.out.println("内存总量：" + memMap.get("total"));

System.out.println("当前内存使用量：" + memMap.get("used"));

System.out.println("当前内存剩余量：" + memMap.get("free"));

System.out.println("-----------------------------------------------");

ctx.writeAndFlush("info received!");

}

else

{

ctx.writeAndFlush("connectfailure").addListener(ChannelFutureListener.CLOSE);

}

}

}

package com.hiya.nio.netty.heartBeat;

import io.netty.channel.ChannelInitializer;

import io.netty.channel.socket.SocketChannel;

public class HeartBeatServerInitializer extends ChannelInitializer<SocketChannel>

{

@Override

protected void initChannel(SocketChannel socketChannel) throws Exception

{

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder());

//socketChannel.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingEncoder());

socketChannel.pipeline().addLast(new HeartBeatServerHandler());

}

}

package com.hiya.nio.netty.heartBeat;

import java.io.Serializable;

import java.util.Map;

public class RequestInfo implements Serializable

{

private static final long serialVersionUID = 133455L;

private String ip;

private Map<String, Object> cpuPercMap;

private Map<String, Object> memoryMap;

public String getIp()

{

return ip;

}

public void setIp(String ip)

{

this.ip = ip;

}

public Map<String, Object> getCpuPercMap()

{

return cpuPercMap;

}

public void setCpuPercMap(Map<String, Object> cpuPercMap)

{

this.cpuPercMap = cpuPercMap;

}

public Map<String, Object> getMemoryMap()

{

return memoryMap;

}

public void setMemoryMap(Map<String, Object> memoryMap)

{

this.memoryMap = memoryMap;

}

}

**5 Netty高阶**

作为一个学Java的，如果没有研究过Netty，那么你对Java语言的使用和理解仅仅停留在表面水平，会点SSH，写几个MVC，访问数据库和缓存，这些只是初等Java程序员干的事。如果你要进阶，想了解Java服务器的深层高阶知识，Netty绝对是一个必须要过的门槛。有了Netty，你可以实现自己的HTTP服务器，FTP服务器，UDP服务器，RPC服务器，WebSocket服务器，Redis的Proxy服务器，MySQL的Proxy服务器等等。如果你想

知道Nginx是怎么写出来的，如果你想知道Tomcat和Jetty是如何实现的，如果你也想实现一个简单的Redis服务器，那都应该好好理解一下Netty，它们高性能的原理都是类似的。

我们回顾一下传统的HTTP服务器的原理创建一个ServerSocket，监听并绑定一个端口一系列客户端来请求这个端口服务器使用Accept，获得一个来自客户端的Socket连接对象启动一个新线程处理连接读Socket，得到字节流解码协议，得到Http请求对象处理Http请求，得到一个结果，封装成一个HttpResponse对象编码协议，将结果序列化字节流写Socket，将字节流发给客户端继续循环步骤3HTTP服务器之所以称为HTTP服务器，是因为编码解码协议是HTTP协议，如果协议是Redis协议，那它就成了Redis服务器，如果协议是WebSocket，那它就成了WebSocket服务器，等等。使用Netty你就可以定制编解码协议，实现自己的特定协议的服务器。上面我们说的是一个传统的多线程服务器，这个也是Apache处理请求的模式。在高并发环境下，线程数量可能会创建太多，操作系统的任务调度压力大，系统负载也会比较高。那怎么办呢？

于是NIO诞生了，NIO并不是Java独有的概念，NIO代表的一个词汇叫着IO多路复用。它是由操作系统提供的系统调用，早期这个操作系统调用的名字是select，但是性能低下，后来渐渐演化成了Linux下的epoll和Mac里的kqueue。我们一般就说是epoll，因为没有人拿苹果电脑作为服务器使用对外提供服务。而Netty就是基于Java NIO技术封装的一套框架。为什么要封装，因为原生的JavaNIO使用起来没那么方便，而且还有臭名昭著的bug，Netty把它封装之后，提供了一个易于操作的使用模式和接口，用户使用起来也就便捷多了。那NIO究竟是什么东西呢？

NIO的全称是NoneBlocking IO，非阻塞IO，区别与BIO，BIO的全称是BlockingIO，阻塞IO。那这个阻塞是什么意思呢？Accept是阻塞的，只有新连接来了，Accept才会返回，主线程才能继Read是阻塞的，只有请求消息来了，Read才能返回，子线程才能继续处理Write是阻塞的，只有客户端把消息收了，Write才能返回，子线程才能继续读取下一个请求所以传统的多线程服务器是BlockingIO模式的，从头到尾所有的线程都是阻塞的。这些线程就干等在哪里，占用了操作系统的调度资源，什么事也不干，是浪费。那么NIO是怎么做到非阻塞的呢。它用的是事件机制。它可以用一个线程把Accept，读写操作，请求处理的逻辑全干了。如果什么事都没得做，它也不会死循环，它会将线程休眠起来，直到下一个事件来了再继续干活，这样的一个线程称之为NIO线程。

while true {

events = takeEvents(fds) // 获取事件，如果没有事件，线程就休眠

for event in events {

if event.isAcceptable {

doAccept() // 新链接来了

} elif event.isReadable {

request = doRead() // 读消息

if request.isComplete() {

doProcess()

}

} elif event.isWriteable {

doWrite() // 写消息

}

}

}

NIO的流程大致就是上面的伪代码描述的过程，跟实际真实的代码有较多差异，不过对于初学者，这样理解也是足够了。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并发模型。Netty提供了内置的常用编解码器，包括行编解码器［一行一个请求］，前缀长度编解码器［前N个字节定义请求的字节长度］，可重放解码器［记录半包消息的状态］，HTTP编解码器，WebSocket消息编解码器等等Netty提供了一些列生命周期回调接口，当一个完整的请求到达时，当一个连接关闭时，当一个连接建立时，用户都会收到回调事件，然后进行逻辑处理。Netty可以同时管理多个端口，可以使用NIO客户端模型，这些对于RPC服务是很有必要的。Netty除了可以处理TCP Socket之外，还可以处理UDP Socket。在消息读写过程中，需要大量使用ByteBuffer，Netty对ByteBuffer在性能和使用的便捷性上都进行了优化和抽象。总之，Netty是Java程序员进阶的必备神奇。如果你知其然，还想知其所以然，一定要好好研究下Netty。如果你觉得Java枯燥无谓，Netty则是重新开启你对Java兴趣大门的钥匙。