# **NIO是怎么工作的**

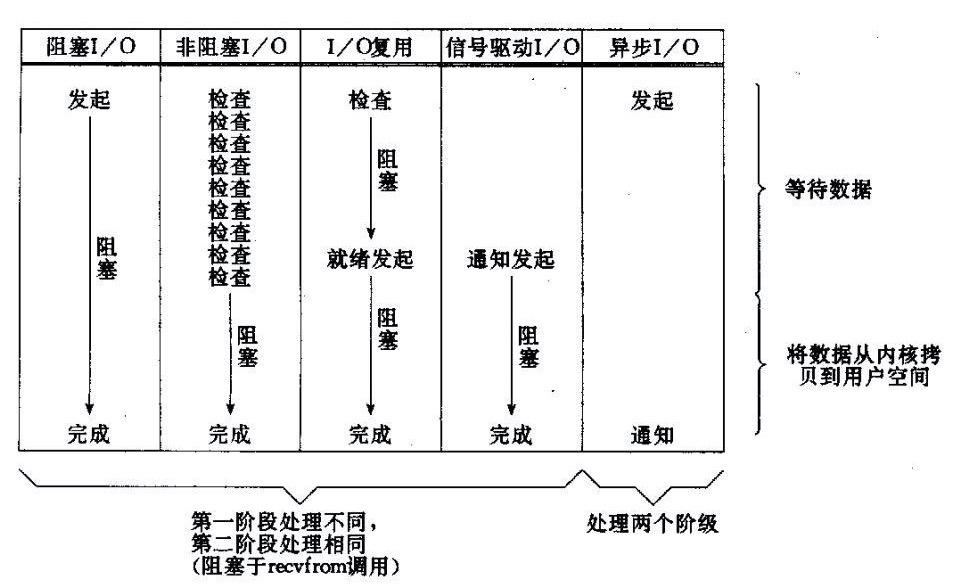
很多刚接触NIO的人，第一眼看到的就是Java相对晦涩的API，比如：Channel，Selector，Socket什么的；然后就是一坨上百行的代码来演示NIO的服务端Demo……瞬间头大有没有？

我们不管这些，抛开现象看本质，先分析下NIO是怎么工作的。

## **常见I/O模型对比**

所有的系统I/O都分为两个阶段：等待就绪和操作。举例来说，读函数，分为等待系统可读和真正的读；同理，写函数分为等待网卡可以写和真正的写。

需要说明的是等待就绪的阻塞是不使用CPU的，是在“空等”；而真正的读写操作的阻塞是使用CPU的，真正在"干活"，而且这个过程非常快，属于memory copy，带宽通常在1GB/s级别以上，可以理解为基本不耗时。

下图是几种常见I/O模型的对比：  


以socket.read()为例子：

传统的BIO里面socket.read()，如果TCP RecvBuffer里没有数据，函数会一直阻塞，直到收到数据，返回读到的数据。

对于NIO，如果TCP RecvBuffer有数据，就把数据从网卡读到内存，并且返回给用户；反之则直接返回0，永远不会阻塞。

最新的AIO(Async I/O)里面会更进一步：不但等待就绪是非阻塞的，就连数据从网卡到内存的过程也是异步的。

**换句话说，BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"，在AIO模型里用户更需要关注的是“读完了”。**

**NIO一个重要的特点是：socket主要的读、写、注册和接收函数，在等待就绪阶段都是非阻塞的，真正的I/O操作是同步阻塞的（消耗CPU但性能非常高）。**

**如何结合事件模型使用NIO同步非阻塞特性**

回忆BIO模型，之所以需要多线程，是因为在进行I/O操作的时候，一是没有办法知道到底能不能写、能不能读，只能"傻等"，即使通过各种估算，算出来操作系统没有能力进行读写，也没法在socket.read()和socket.write()函数中返回，这两个函数无法进行有效的中断。所以除了多开线程另起炉灶，没有好的办法利用CPU。

NIO的读写函数可以立刻返回，这就给了我们不开线程利用CPU的最好机会：如果一个连接不能读写（socket.read()返回0或者socket.write()返回0），我们可以把这件事记下来，记录的方式通常是在Selector上注册标记位，然后切换到其它就绪的连接（channel）继续进行读写。

下面具体看下如何利用事件模型单线程处理所有I/O请求：

NIO的主要事件有几个：读就绪、写就绪、有新连接到来。

我们首先需要注册当这几个事件到来的时候所对应的处理器。然后在合适的时机告诉事件选择器：我对这个事件感兴趣。对于写操作，就是写不出去的时候对写事件感兴趣；对于读操作，就是完成连接和系统没有办法承载新读入的数据的时；对于accept，一般是服务器刚启动的时候；而对于connect，一般是connect失败需要重连或者直接异步调用connect的时候。

其次，用一个死循环选择就绪的事件，会执行系统调用（Linux 2.6之前是select、poll，2.6之后是epoll，Windows是IOCP），还会阻塞的等待新事件的到来。新事件到来的时候，会在selector上注册标记位，标示可读、可写或者有连接到来。

注意，select是阻塞的，无论是通过操作系统的通知（epoll）还是不停的轮询(select，poll)，这个函数是阻塞的。所以你可以放心大胆地在一个while(true)里面调用这个函数而不用担心CPU空转。

所以我们的程序大概的模样是：

interface ChannelHandler{

void channelReadable(Channel channel);

void channelWritable(Channel channel);

}

class Channel{

Socket socket;

Event event;//读，写或者连接

}

//IO线程主循环:

class IoThread extends Thread{

public void run(){

Channel channel;

while(channel=Selector.select()){//选择就绪的事件和对应的连接

if(channel.event==accept){

registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器

}

if(channel.event==write){

getChannelHandler(channel).channelWritable(channel);//如果可以写，则执行写事件

}

if(channel.event==read){

getChannelHandler(channel).channelReadable(channel);//如果可以读，则执行读事件

}

}

}

Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器

}

**这个程序很简短，也是最简单的Reactor模式：注册所有感兴趣的事件处理器，单线程轮询选择就绪事件，执行事件处理器。**

## **优化线程模型**

由上面的示例我们大概可以总结出NIO是怎么解决掉线程的瓶颈并处理海量连接的：

NIO由原来的阻塞读写（占用线程）变成了单线程轮询事件，找到可以进行读写的网络描述符进行读写。除了事件的轮询是阻塞的（没有可干的事情必须要阻塞），剩余的I/O操作都是纯CPU操作，没有必要开启多线程。

并且由于线程的节约，连接数大的时候因为线程切换带来的问题也随之解决，进而为处理海量连接提供了可能。

单线程处理I/O的效率确实非常高，没有线程切换，只是拼命的读、写、选择事件。但现在的服务器，一般都是多核处理器，如果能够利用多核心进行I/O，无疑对效率会有更大的提高。

仔细分析一下我们需要的线程，其实主要包括以下几种：

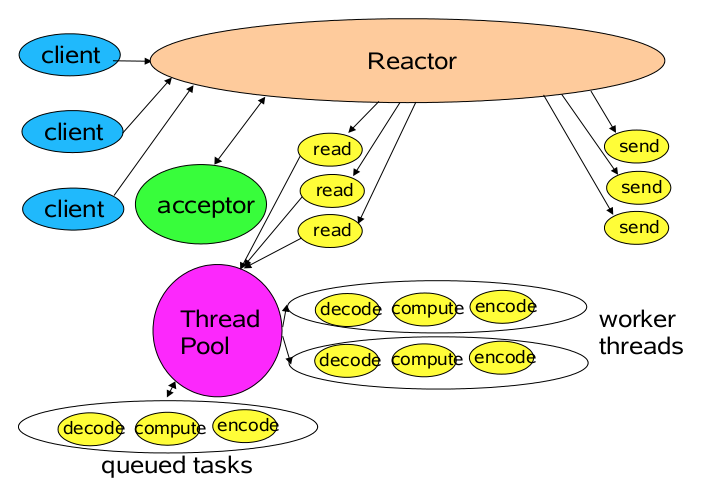
事件分发器，单线程选择就绪的事件。

I/O处理器，包括connect、read、write等，这种纯CPU操作，一般开启CPU核心个线程就可以。

业务线程，在处理完I/O后，业务一般还会有自己的业务逻辑，有的还会有其他的阻塞I/O，如DB操作，RPC等。只要有阻塞，就需要单独的线程。

**Java的Selector对于Linux系统来说，有一个致命限制：同一个channel的select不能被并发的调用。因此，如果有多个I/O线程，必须保证：一个socket只能属于一个IoThread，而一个IoThread可以管理多个socket。**

另外连接的处理和读写的处理通常可以选择分开，这样对于海量连接的注册和读写就可以分发。虽然read()和write()是比较高效无阻塞的函数，但毕竟会占用CPU，如果面对更高的并发则无能为力。



**NIO在客户端的魔力**

通过上面的分析，可以看出NIO在服务端对于解放线程，优化I/O和处理海量连接方面，确实有自己的用武之地。那么在客户端上，NIO又有什么使用场景呢?

常见的客户端BIO+连接池模型，可以建立n个连接，然后当某一个连接被I/O占用的时候，可以使用其他连接来提高性能。

但多线程的模型面临和服务端相同的问题：如果指望增加连接数来提高性能，则连接数又受制于线程数、线程很贵、无法建立很多线程，则性能遇到瓶颈。

**每连接顺序请求的Redis**

对于Redis来说，由于服务端是全局串行的，能够保证同一连接的所有请求与返回顺序一致。这样可以使用单线程＋队列，把请求数据缓冲。然后pipeline发送，返回future，然后channel可读时，直接在队列中把future取回来，done()就可以了。

伪代码如下：

class RedisClient Implements ChannelHandler{

private BlockingQueue CmdQueue;

private EventLoop eventLoop;

private Channel channel;

class Cmd{

String cmd;

Future result;

}

public Future get(String key){

Cmd cmd= new Cmd(key);

queue.offer(cmd);

eventLoop.submit(new Runnable(){

List list = new ArrayList();

queue.drainTo(list);

if(channel.isWritable()){

channel.writeAndFlush(list);

}

});

}

public void ChannelReadFinish(Channel channel，Buffer Buffer){

List result = handleBuffer();//处理数据

//从cmdQueue取出future，并设值，future.done();

}

public void ChannelWritable(Channel channel){

channel.flush();

}

}

这样做，能够充分的利用pipeline来提高I/O能力，同时获取异步处理能力。

**NIO高级主题**

**Proactor与Reactor**

一般情况下，I/O 复用机制需要事件分发器（event dispatcher）。 事件分发器的作用，即将那些读写事件源分发给各读写事件的处理者，就像送快递的在楼下喊: 谁谁谁的快递到了， 快来拿吧！开发人员在开始的时候需要在分发器那里注册感兴趣的事件，并提供相应的处理者（event handler)，或者是回调函数；事件分发器在适当的时候，会将请求的事件分发给这些handler或者回调函数。

涉及到事件分发器的两种模式称为：Reactor和Proactor。 Reactor模式是基于同步I/O的，而Proactor模式是和异步I/O相关的。

在Reactor模式中，事件分发器等待某个事件或者可应用某个操作的状态发生（比如文件描述符可读写，或者是socket可读写），事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。

而在Proactor模式中，**事件处理者（或者代由事件分发器发起）直接发起一个异步读写操作（相当于请求），而实际的工作是由操作系统来完成的。**发起时，需要提供的参数包括用于存放读到数据的缓存区、读的数据大小或用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。**事件分发器得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。**举例来说，在Windows上事件处理者投递了一个异步IO操作（称为overlapped技术），事件分发器等IO Complete事件完成。这种异步模式的典型实现是基于操作系统底层异步API的，所以我们可称之为“系统级别”的或者“真正意义上”的异步，因为具体的读写是由操作系统代劳的。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（写操作类似）。

**在Reactor中实现读**

注册读就绪事件和相应的事件处理器。

事件分发器等待事件。

事件到来，激活分发器，分发器调用事件对应的处理器。

事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

**在Proactor中实现读：**

处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。

事件分发器等待操作完成事件。

在分发器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分发器读操作完成。

事件分发器呼唤处理器。

事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分发器。

可以看出，两个模式的相同点，都是对某个I/O事件的事件通知（即告诉某个模块，这个I/O操作可以进行或已经完成)。在结构上，两者也有相同点：事件分发器负责提交IO操作（异步)、查询设备是否可操作（同步)，然后当条件满足时，就回调handler；不同点在于，异步情况下（Proactor)，当回调handler时，表示I/O操作已经完成；同步情况下（Reactor)，回调handler时，表示I/O设备可以进行某个操作（can read 或 can write)。

下面，我们将尝试应对为Proactor和Reactor模式建立可移植框架的挑战。在改进方案中，我们将Reactor原来位于事件处理器内的Read/Write操作移至分发器（不妨将这个思路称为“模拟异步”），以此寻求将Reactor多路同步I/O转化为模拟异步I/O。以读操作为例子，改进过程如下：

注册读就绪事件和相应的事件处理器。并为分发器提供数据缓冲区地址，需要读取数据量等信息。

分发器等待事件（如在select()上等待）。

事件到来，激活分发器。分发器执行一个非阻塞读操作（它有完成这个操作所需的全部信息），最后调用对应处理器。

事件处理器处理用户自定义缓冲区的数据，注册新的事件（当然同样要给出数据缓冲区地址，需要读取的数据量等信息），最后将控制权返还分发器。

如我们所见，通过对多路I/O模式功能结构的改造，可将Reactor转化为Proactor模式。改造前后，模型实际完成的工作量没有增加，只不过参与者间对工作职责稍加调换。没有工作量的改变，自然不会造成性能的削弱。对如下各步骤的比较，可以证明工作量的恒定：

**标准/典型的Reactor：**

步骤1：等待事件到来（Reactor负责）。

步骤2：将读就绪事件分发给用户定义的处理器（Reactor负责）。

步骤3：读数据（用户处理器负责）。

步骤4：处理数据（用户处理器负责）。

改进实现的模拟Proactor：

步骤1：等待事件到来（Proactor负责）。

步骤2：得到读就绪事件，执行读数据（现在由Proactor负责）。

步骤3：将读完成事件分发给用户处理器（Proactor负责）。

步骤4：处理数据（用户处理器负责）。

对于不提供异步I/O API的操作系统来说，这种办法可以隐藏Socket API的交互细节，从而对外暴露一个完整的异步接口。借此，我们就可以进一步构建完全可移植的，平台无关的，有通用对外接口的解决方案。

代码示例如下：

interface ChannelHandler{

void channelReadComplate(Channel channel，byte[] data);

void channelWritable(Channel channel);

}

class Channel{

Socket socket;

Event event;//读，写或者连接

}

//IO线程主循环：

class IoThread extends Thread{

public void run(){

Channel channel;

while(channel=Selector.select()){//选择就绪的事件和对应的连接

if(channel.event==accept){

registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器

Selector.interested(read);

}

if(channel.event==write){

getChannelHandler(channel).channelWritable(channel);//如果可以写，则执行写事件

}

if(channel.event==read){

byte[] data = channel.read();

if(channel.read()==0)//没有读到数据，表示本次数据读完了

{

getChannelHandler(channel).channelReadComplate(channel，data;//处理读完成事件

}

if(过载保护){

Selector.interested(read);

}

}

}

}

Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器

}

## **Selector.wakeup()**

### **主要作用**

解除阻塞在Selector.select()/select(long)上的线程，立即返回。

两次成功的select之间多次调用wakeup等价于一次调用。

如果当前没有阻塞在select上，则本次wakeup调用将作用于下一次select——“记忆”作用。

为什么要唤醒？

注册了新的channel或者事件。

channel关闭，取消注册。

优先级更高的事件触发（如定时器事件），希望及时处理。

### **原理**

Linux上利用pipe调用创建一个管道，Windows上则是一个loopback的tcp连接。这是因为win32的管道无法加入select的fd set，将管道或者TCP连接加入select fd set。

wakeup往管道或者连接写入一个字节，阻塞的select因为有I/O事件就绪，立即返回。可见，wakeup的调用开销不可忽视。

## **Buffer的选择**

通常情况下，操作系统的一次写操作分为两步：

1. 将数据从用户空间拷贝到系统空间。
2. 从系统空间往网卡写。同理，读操作也分为两步：  
   ① 将数据从网卡拷贝到系统空间；  
   ② 将数据从系统空间拷贝到用户空间。

对于NIO来说，缓存的使用可以使用DirectByteBuffer和HeapByteBuffer。如果使用了DirectByteBuffer，一般来说可以减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，更不宜维护，通常会用内存池来提高性能。

如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer；反之可以用directBuffer。

# **NIO存在的问题**

使用NIO != 高性能，当连接数<1000，并发程度不高或者局域网环境下NIO并没有显著的性能优势。

NIO并没有完全屏蔽平台差异，它仍然是基于各个操作系统的I/O系统实现的，差异仍然存在。使用NIO做网络编程构建事件驱动模型并不容易，陷阱重重。

推荐大家使用成熟的NIO框架，如Netty，MINA等。解决了很多NIO的陷阱，并屏蔽了操作系统的差异，有较好的性能和编程模型。

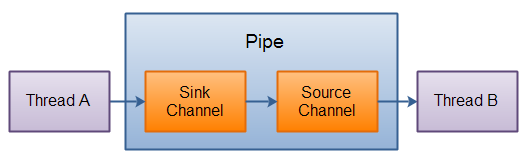
# **总结**

最后总结一下到底NIO给我们带来了些什么：

* **事件驱动模型**
* **避免多线程**
* **单线程处理多任务**
* **非阻塞I/O，I/O读写不再阻塞，而是返回0**
* **基于block的传输，通常比基于流的传输更高效**
* **更高级的IO函数，zero-copy**
* **IO多路复用大大提高了Java网络应用的可伸缩性和实用性**

Java NIO 管道是2个线程之间的单向数据连接。Pipe有一个source通道和一个sink通道。数据会被写到sink通道，从source通道读取。

这里是Pipe原理的图示：



I/O多路复用模式  
I/O多路复用有两种经典模式：基于同步I/O的reactor和基于异步I/O的proactor。  
• Reactor   
o 某个事件处理者宣称它对某个socket上的读事件很感兴趣;   
o 事件分离者等着这个事件的发生;   
o 当事件发生了，事件分离器被唤醒，这负责通知先前那个事件处理者;   
o 事件处理者收到消息，于是去那个socket上读数据了. 如果需要，它再次宣称对这个socket上的读事件感兴趣，一直重复上面的步骤;   
• Proactor

与Reactor模式不同，Proactor模式将所有I/O操作都交给主线程和内核来处理，工作线程仅仅负责业务逻辑。  
o 事件处理者直接投递发一个写操作(当然，操作系统必须支持这个异步操作). 这个时候，事件处理者根本不关心读事件，它只管发这么个请求，它魂牵梦萦的是这个写操作的完成事件。这个处理者很拽，发个命令就不管具体的事情了，只等着别人（系统）帮他搞定的时候给他回个话。   
o 事件分离者等着这个读事件的完成(比较下与Reactor的不同);   
o 当事件分离者默默等待完成事情到来的同时，操作系统已经在一边开始干活了，它从目标读取数据，放入用户提供的缓存区中，最后通知事件分离者，这个事情我搞完了;   
o 事件分享者通知之前的事件处理者: 你吩咐的事情搞定了;   
o 事件处理者这时会发现想要读的数据已经乖乖地放在他提供的缓存区中，想怎么处理都行了。如果有需要，事件处理者还像之前一样发起另外一个写操作，和上面的几个步骤一样。   
异步的proactor固然不错，但它局限于操作系统（要支持异步操作），为了开发真正独立平台的通用接口，我们可以通过reactor模拟来实现proactor。  
• Proactor（模拟）   
o 等待事件 (Proactor 的工作)   
o 读数据(看，这里变成成了让 Proactor 做这个事情)   
o 把数据已经准备好的消息给用户处理函数，即事件处理者(Proactor 要做的)   
o 处理数据 (用户代码要做的)

我应该何时使用IO，何时使用NIO呢？

在本文中，我会尽量清晰地解析Java NIO和IO的差异、它们的使用场景，以及它们如何影响您的代码设计。

Java NIO和IO的主要区别

**下表总结了Java NIO和IO之间的主要差别**

我会更详细地描述表中每部分的差异。

|  |  |
| --- | --- |
| IO                           NIO 面向流                     面向缓冲 阻塞IO                    非阻塞IO 无                           选择器 |  |

# 面向流与面向缓冲

**Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。**Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。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。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

# 阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，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。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

# 选择器（Selectors）

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

# NIO和IO如何影响应用程序的设计

无论您选择IO或NIO工具箱，可能会影响您应用程序设计的以下几个方面：

1. 对NIO或IO类的API调用。
2. 数据处理。
3. 用来处理数据的线程数。

## API调用

当然，使用NIO的API调用时看起来与使用IO时有所不同，但这并不意外，因为并不是仅从一个InputStream逐字节读取，而是数据必须先读入缓冲区再处理。

## 数据处理

使用纯粹的NIO设计相较IO设计，数据处理也受到影响。

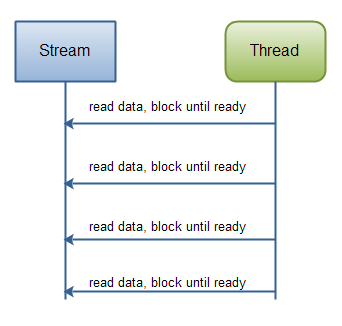
在IO设计中，我们从InputStream或 Reader逐字节读取数据。假设你正在处理一基于行的文本数据流，例如：

|  |  |
| --- | --- |
| 1  2  3  4 | Name: Anna  Age: 25  Email: anna@mailserver.com  Phone: 1234567890 |

该文本行的流可以这样处理：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | InputStream input = ... ; // get the InputStream from the client socket    BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(input));    String nameLine   = reader.readLine();  String ageLine    = reader.readLine();  String emailLine  = reader.readLine();  String phoneLine  = reader.readLine(); |

请注意处理状态由程序执行多久决定。换句话说，一旦reader.readLine()方法返回，你就知道肯定文本行就已读完， readline()阻塞直到整行读完，这就是原因。你也知道此行包含名称；同样，第二个readline()调用返回的时候，你知道这行包含年龄等。 正如你可以看到，该处理程序仅在有新数据读入时运行，并知道每步的数据是什么。一旦正在运行的线程已处理过读入的某些数据，该线程不会再回退数据（大多如此）。下图也说明了这条原则：



而一个NIO的实现会有所不同，下面是一个简单的例子：

|  |  |
| --- | --- |
| 1  2 | ByteBuffer buffer = ByteBuffer.allocate(48);  **int** bytesRead = inChannel.read(buffer); |

注意第二行，从通道读取字节到ByteBuffer。当这个方法调用返回时，你不知道你所需的所有数据是否在缓冲区内。你所知道的是，该缓冲区包含一些字节，这使得处理有点困难。  
假设第一次 read(buffer)调用后，读入缓冲区的数据只有半行，例如，“Name:An”，你能处理数据吗？显然不能，需要等待，直到整行数据读入缓存，在此之前，对数据的任何处理毫无意义。

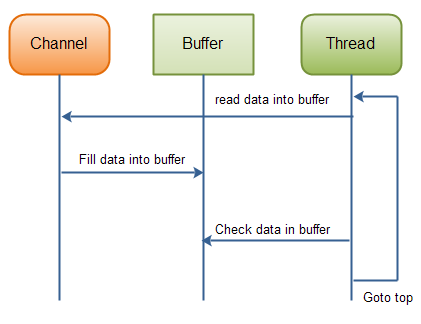
所以，你怎么知道是否该缓冲区包含足够的数据可以处理呢？好了，你不知道。发现的方法只能查看缓冲区中的数据。其结果是，在你知道所有数据都在缓冲区里之前，你必须检查几次缓冲区的数据。这不仅效率低下，而且可以使程序设计方案杂乱不堪。例如：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ByteBuffer buffer = ByteBuffer.allocate(48);    **int** bytesRead = inChannel.read(buffer);    **while**(! bufferFull(bytesRead) ) {    bytesRead = inChannel.read(buffer);    } |

bufferFull()方法必须跟踪有多少数据读入缓冲区，并返回真或假，这取决于缓冲区是否已满。换句话说，如果缓冲区准备好被处理，那么表示缓冲区满了。

bufferFull()方法扫描缓冲区，但必须保持在bufferFull（）方法被调用之前状态相同。如果没有，下一个读入缓冲区的数据可能无法读到正确的位置。这是不可能的，但却是需要注意的又一问题。

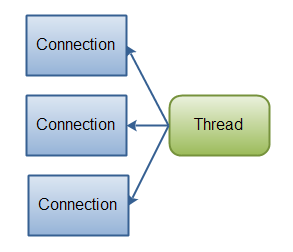
如果缓冲区已满，它可以被处理。如果它不满，并且在你的实际案例中有意义，你或许能处理其中的部分数据。但是许多情况下并非如此。下图展示了“缓冲区数据循环就绪”：



# 总结

NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

如果需要管理同时打开的成千上万个连接，这些连接每次只是发送少量的数据，例如聊天服务器，实现NIO的服务器可能是一个优势。同样，如果你需要维持许多打开的连接到其他计算机上，如P2P网络中，使用一个单独的线程来管理你所有出站连接，可能是一个优势。一个线程多个连接的设计方案如下图所示：



如果你有少量的连接使用非常高的带宽，一次发送大量的数据，也许典型的IO服务器实现可能非常契合。下图说明了一个典型的IO服务器设计：

