http://wiki.jikexueyuan.com/project/java-nio-zh/java-nio-non-blocking-server.html

现在我们已经知道了Java NIO里面那些非阻塞特性是怎么工作的，但是要设计一个非阻塞的服务仍旧比较困难。非阻塞IO相对传统的阻塞IO给开发者带来了更多的挑战。在本节非阻塞服务的讲解中，我们一起来讨论这些会面临的主要挑战，同时也会给出一些潜在的解决方案。

虽然本文介绍的一些点子是为Java NIO设计的，但是我相信这些思路同样适用于其他编程语言，只要他们也存在和Selector类似结构，概念。就目前我的了解来说，这些结构底层OS提供的，所以基本上你可以运用到其他编程语言中去。

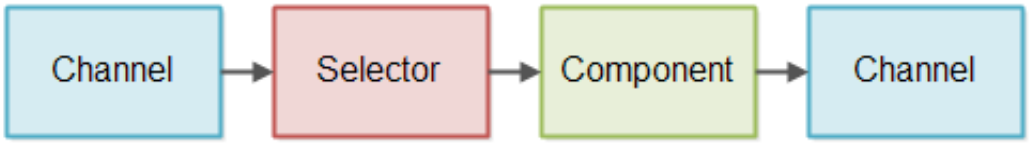
非阻塞服务-GitHub源码仓（Non-blocking Server - GitHub Repository）

为了演示本文探讨的一些技术，笔者已经在GitHub上面建立了相应的源码仓，地址如下：

https://github.com/jjenkov/java-nio-server

非阻塞IO通道（Non-blocking IO Pipelines）

非阻塞的IO管道（Non-blocking IO Pipelines）可以看做是整个非阻塞IO处理过程的链条。包括在以非阻塞形式进行的读与写操作。下面有一张插图，简单的描述了一个基础的非阻塞的IO管道（Non-blocking IO Pipelines）：



我们的组件（Component）通过Selector检查当前Channel是否有数据需要写入。此时component读入数据，并且根据输入的数据input对外提供数据输出output。这个对外的数据输出output被写到了另一个Channel中。

一个非阻塞的IO管道不必同时需要读和写数据，通常来说有些管道只需要读数据，而另一些管道则只需写数据。

上面的这幅流程图仅仅展示了一个组件。实际上一个管道可能存在多个component在处理输入数据。管道的长度取决于管道具体要做的事情。

当然一个非阻塞的IO管道他也可以同时从多个Channel中读取数据，例如同时从多个

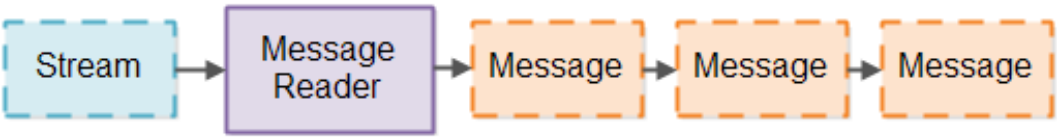
SocketChannel中读取数据；

上面的流程图实际上被简化了，图中的Component实际上负责初始化Selector，从Channel中读取数据，而不是由Channel往Selector压如数据（push），这是简化的上图容易给人带来的误解。

非阻塞和阻塞通道比较（Non-blocking vs. Blocking IO Pipelines）

非阻塞IO管道和阻塞IO管道之间最大的区别是他们各自如何从Channel（套接字socket或文件file）读写数据。

IO管道通常直接从流中（来自于socket或file的流）读取数据，然后把数据分割为连续的消息。这个处理与我们读取流信息，用tokenizer进行解析非常相似。不同的是我们在这里会把数据流分割为更大一些的消息块。我把这个过程叫做Message Reader.下面是一张说明的插图：



一个阻塞IO管道的使用可以和输入流一样调用，每次从Channel中读取一个字节的数据，阻塞自身直到有数据可读。这个流程就是一个阻塞的Messsage Reader实现。

使用阻塞IO大大简化了Message Reader的实现成本。阻塞的Message Reader无需关注没有数据返回的情形，无需关注返回部分数据或者数据解析需要被复用的问题。

相似的，一个阻塞的Message Writer也不需要关注写入部分数据，和数据复用的问题。

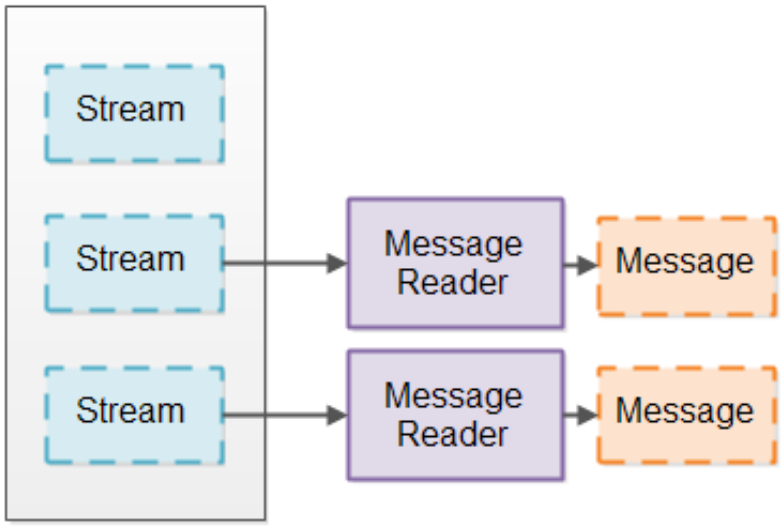
阻塞IO通道的缺点（Blocking IO Pipeline Drawbacks）

上面提到了阻塞的Message Reader易于实现，但是阻塞也给他带了不可避免的缺点，必须为每个数据数量都分配一个单独线程。原因就在于IO接口在读取数据时在有数据返回前会一直被阻塞住。这直接导致我们无法用单线程来处理一个流没有数据返回时去读取其他的流。每当一个线程尝试去读取一个流的数据，这个线程就会被阻塞直到有数据真正返回。

如果这样的IO管道运用到服务器去处理高并发的链接请求，服务器将不得不为每一个到来的链接分配一个单独的线程。如果并发数不高比如每一时刻只有几百并发，也行不会有太大问题。一旦服务器的并发数上升到百万级别，这种设计就缺乏伸缩性。每

个线程需要为堆栈分配320KB（32位JVM）到1024KB(64位JVM)的内存空间。这就是说如果有1,000,000个线程，需要1TB的内存。而这些在还没开始真正处理接收到的消息前就需要（消息处理中还需要为对象开拍内存）。

为了减少线程数，很多服务器都设计了线程池，把所有接收到的请求放到队列内，每次读取一条连接进行处理。这种设计可以用下图表示：



但是这种设计要求缓冲的连接进程发送有意义的数据。如果这些连接长时间处于非活跃的状态，那么大量非活跃的连接会阻塞线程池中的所有线程。这会导致服务器的响应速度特别慢甚至无响应。

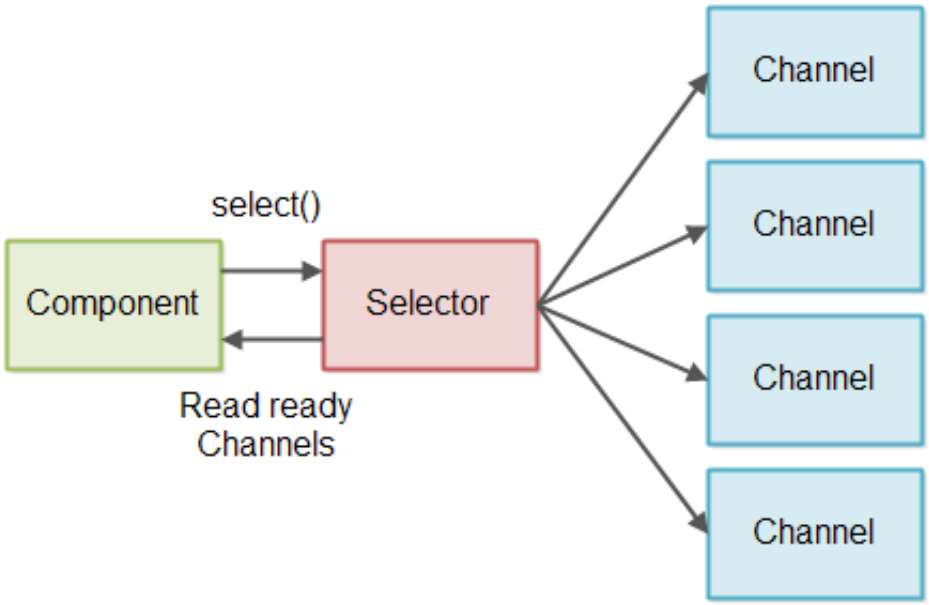
有些服务器为了减轻这个问题，采取的操作是适当增加线程池的弹性。例如，当线程池所有线程都处于饱和时，线程池可能会自动扩容，启动更多的线程来处理事务。这个解决方案会使得服务器维护大量不活跃的链接。但是需要谨记服务器所能开辟的线程数是有限制的。所有当有1,000,000个低速的链接时，服务器还是不具备伸缩性。

基础的非阻塞通道设计（Basic Non-blocking IO Pipeline Design）

一个非阻塞的IO通道可以用单线程读取多个数据流。这个前提是相关的流可以切换为非阻塞模式（并不是所有流都可以以非阻塞形式操作）。在非阻塞模式下，读取一个流可能返回0个或多个字节。如果流还没有可供读取的数据那么就会返回0，其他大于1的返回都表明这是实际读取到的数据；

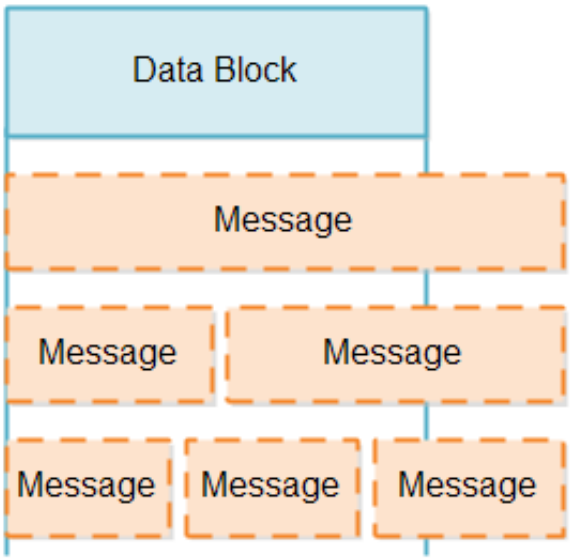
为了避开没有数据可读的流，我们结合Java NIO中的Selector。一个Selector可以注册多个SelectableChannel实例。当我们调用select()或selectorNow()方法时

Selector会返回一个有数据可读的SelectableChannel实例。这个设计可以如下插图：



读取部分信息(Reading Partial Messages)

当我们从SelectableChannel中读取一段数据后，我们并不知道这段数据是否是完整的一个message。因为一个数据段可能包含部分message，也就是说即可能少于一个message，也可能多一个message，正如下面这张插图所示意的那样：

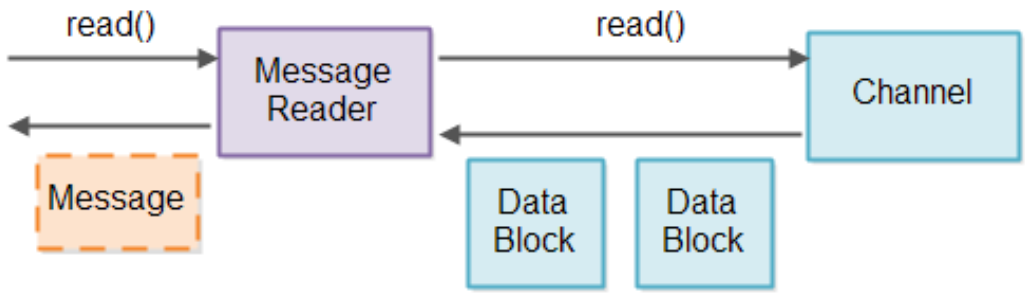


要处理这种截断的message，我们会遇到两个问题：

1. 检测数据段中是否包含一个完整的message
2. 在message剩余部分获取到之前，我们如何处理不完整的message

检测完整message要求Message Reader查看数据段中的数据是否至少包含一个完整的message。如果包含一个或多个完整message，这些message可以被下发到通道中处理。查找完整message的过程是个大量重复的操作，所以这个操作必须是越快越好的。

当数据段中有一个不完整的message时，无论不完整消息是整个数据段还是说在完整message前后，这个不完整的message数据都需要在剩余部分获得前存储起来。检查message完整性和存储不完整message都是Message Reader的职责。为了避免混淆来自不同Channel的数据，我们为每一个Channel分配一个Message Reader。整个设计大概是这样的：



当我们通过Selector获取到一个有数据可以读取的Channel之后，改Channel关联的Message Reader会读取数据，并且把数据打断为Message块。得到完整的message后就可以通过通道下发到其他组件进行处理。

一个Message Reader自然是协议相关的。他需要知道message的格式以便读取。如果我们的服务器是跨协议复用的，那他必须实现Message Reader的协议-大致类似于接收一个Message Reader工厂作为配置参数。

存储不完整的Message（Storing Partial Messages）

现在我们已经明确了由Message Reader负责不完整消息的存储直到接收到完整的消息。我们还需要知道这个存储过程需要如何来实现。

在设计的时候我们需要考虑两个关键因素：

1. 我们希望在拷贝消息数据的时候数据量能尽可能的小，拷贝量越大则性能相对越低；
2. 我们希望完整的消息是以顺序的字节存储，这样方便进行数据的解析；

为每个Message Reader分配Buffer（A Buffer Per Message

Reader）

显然不完整的消息数据需要存储在某种buffer中。比较直接的办法是我们为每个Message Reader都分配一个内部的buffer成员。但是，多大的buffer才合适呢？这个buffer必须能存储下一个message最大的大小。如果一个message最大是1MB，那每个Message Reader内部的buffer就至少有1MB大小。

在百万级别的并发链接数下，1MB的buffer基本没法正常工作。举例来说，1,000,000 x 1MB就是1TB的内存大小！如果消息的最大数据量是16MB又需要多少内存呢？128MB呢？

可伸缩Buffer（Resizable Buffers）

另一个方案是在每个Message Reader内部维护一个容量可变的buffer。一个可变的buffer在初始化时占用较少控件，在消息变得很大超出容量时自动扩容。这样每个链接就不需要都占用比如1MB的空间。每个链接只使用承载下一个消息所必须的内存大小。

要实现一个可伸缩的buffer有几种不同的办法。每一种都有它的优缺点，下面几个小结我会逐一讨论它们。

拷贝扩容（Resize by Copy）

第一种实现可伸缩buffer的办法是初始化buffer的时候只申请较少的空间，比如4KB。如果消息超出了4KB的大小那么开赔一个更大的空间，比如8KB，然后把4KB中的数据拷贝纸8KB的内存块中。

以拷贝方式扩容的优点是一个消息的全部数据都被保存在了一个连续的字节数组中。这使得数据解析变得更加容易。

同时它的缺点是会增加大量的数据拷贝操作。

为了减少数据的拷贝操作，你可以分析整个消息流中的消息大小，一次来找到最适合当前机器的可以减少拷贝操作的buffer大小。例如，你可能会注意到觉大多数的消息都是小于4KB的，因为他们仅仅包含了一个非常请求和响应。这意味着消息的处所荣校应该设置为4KB。

同时，你可能会发现如果一个消息大于4KB，很可能是因为他包含了一个文件。你会可能注意到 大多数通过系统的数据都是小于128KB的。所以我们可以在第一次扩容设置为128KB。

最后你可能会发现当一个消息大于128KB后，没有什么规律可循来确定下次分配的空间大小，这意味着最后的buffer容量应该设置为消息最大的可能数据量。

结合这三次扩容时的大小设置，可以一定程度上减少数据拷贝。4KB以下的数据无需拷贝。在1百万的连接下需要的空间例如1,000,000x4KB=4GB，目前（2015）大多数服务器都扛得住。4KB到128KB会仅需拷贝一次，即拷贝4KB数据到128KB的里面。消息大小介于128KB和最大容量的时需要拷贝两次。首先4KB数据被拷贝第二次是拷贝128KB的数据，所以总共需要拷贝132KB数据。假设没有很多的消息会超过128KB，那么这个方案还是可以接受的。

当一个消息被完整的处理完毕后，它占用的内容应当即刻被释放。这样下一个来自东一个链接通道的消息可以从最小的buffer大小重新开始。这个操作是必须的如果我们需要尽可能高效地复用不同链接之间的内存。大多数情况下并不是所有的链接都会在同一时刻需要大容量的buffer。

笔者写了一个完整的教程阐述了如何实现一个内存buffer使其支持扩容：Resizable Arrays 。这个教程也附带了一个指向GitHub上的源码仓地址，里面有实现方案的具体代码。

追加扩容（Resize by Append）

另一种实现buffer扩容的方案是让buffer包含几个数组。当需要扩容的时候只需要在开辟一个新的字节数组，然后把内容写到里面去。

这种扩容也有两个具体的办法。一中是开辟单独的字节数组，然后用一个列表把这些独立数组关联起来。另一种是开辟一些更大的，相互共享的字节数组切片，然后用列表把这些切片和buffer关联起来。个人而言，笔者认为第二种切片方案更好一点点，但是它们之前的差异比较小。（译者话：关于这两个办法，我个人觉得概念介绍有点难懂，建议读者也参考一下原文😏）

这种追加扩容的方案不管是用独立数组还是切片都有一个优点，那就是写数据的时候不需要二外的拷贝操作。所有的数据可以直接从socket（Channel）中拷贝至数组活切片当中。

这种方案的缺点也很明显，就是数据不是存储在一个连续的数组中。这会使得数据的解析变得更加复杂，因为解析器不得不同时查找每一个独立数组的结尾和所有数组的结尾。正因为我们需要在写数据时查找消息的结尾，这个模型在设计实现时会相对不那么容易。

TLV编码消息(TLV Encoded Messages)

有些协议的消息消失采用的是一种TLV格式（Type, Length, Value）。这意味着当消息到达时，消息的完整大小存储在了消息的开始部分。我们可以立刻判断为消息开

辟多少内存空间。

TLV编码是的内存管理变得更加简单。我们可以立刻知道为消息分配多少内存。即便是不完整的消息，buffer结尾后面也不会有浪费的内存。

TLV编码的一个缺点是我们需要在消息的全部数据接收到之前就开辟好需要用的所有内存。因此少量链接慢，但发送了大块数据的链接会占用较多内存，导致服务器无响应。

解决上诉问题的一个变通办法是使用一种内部包含多个TLV的消息格式。这样我们为每个TLV段分配内存而不是为整个的消息分配，并且只在消息的片段到达时才分配内存。但是消息片段很大时，任然会出现一样的问题。

另一个办法是为消息设置超时，如果长时间未接收到的消息（比如10-15秒）。这可以让服务器从偶发的并发处理大块消息恢复过来，不过还是会让服务器有一段时间无响应。另外恶意的DoS攻击会导致服务器开辟大量内存。

TLV编码有不同的变种。有多少字节使用这样确切的类型和字段长度取决于每个独立的TLV编码。有的TLV编码吧字段长度放在前面，接着放类型，最后放值。尽管字段的顺序不同，但他任然是一个TLV的类型。

TLV编码使得内存管理更加简单，这也是HTTP1.1协议让人觉得是一个不太优良的的协议的原因。正因如此，HTTP2.0协议在设计中也利用TLV编码来传输数据帧。也是因为这个原因我们设计了自己的利用TLV编码的网络协议VStack.co。

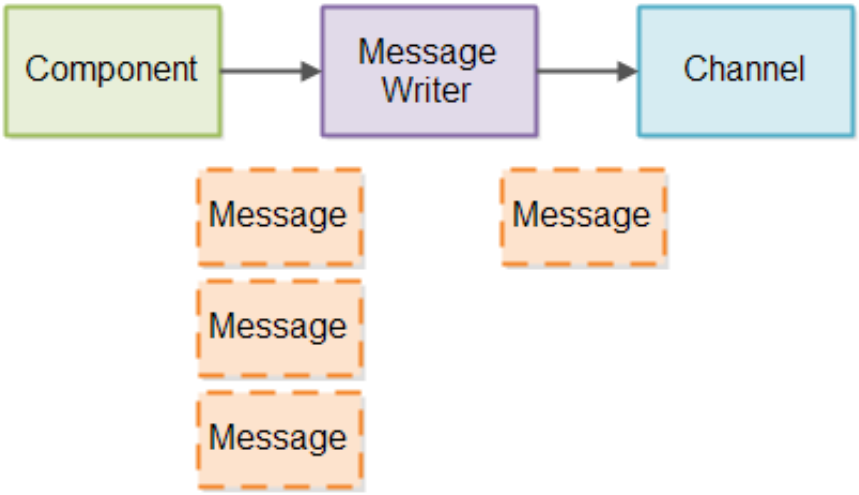
写不完整的消息（Writing Partial Messages）

在非阻塞IO管道中，写数据也是一个不小的挑战。当你调用一个非阻塞模式Channel的write()方法时，无法保证有多少机字节被写入了ByteBuffer中。write方法返回了实际写入的字节数，所以跟踪记录已被写入的字节数也是可行的。这就是我们遇到的问题：持续记录被写入的不完整的小树知道一个消息中所有的数据都发送完毕。

为了管理不完整消息的写操作，我们需要创建一个Message Writer。正如前面的Message Reader，我们也需要每个Channel配备一个Message Writer来写数据。在每个Message Writer中我们记录准确的已经写入的字节数。

为了避免多个消息传递到Message Writer超出他所能处理到Channel的量，我们需要让到达的消息进入队列。Message Writer则尽可能快的将数据写到Channel里。

下面是一个流程图，展示的是不完整消息被写入的过程：



为了使Message Writer能够持续发送刚才已经发送了一部分的消息，Message Writer需要被移植调用，这样他就可以发送更多数据。

如果你有大量的链接，你会持有大量的Message Writer实例。检查比如1百万的Message Writer实例是来确定他们是否处于可写状态是很慢的操作。首先，许多Message Writer可能根本就没有数据需要发送。我们不想检查这些实例。其次，不是所有的Channel都处于可写状态。我们不想浪费时间在这些非写入状态的Channel。

为了检查一个Channel是否可写，可以把它注册到Selector上。但是我们不希望把所有的Channel实例都注册到Selector。试想一下，如果你有1百万的链接，这里面大部分是空闲的，把1百万链接都祖册到Selector上。然后调用select方法的时候就会有很多的Channel处于可写状态。你需要检查所有这些链接中的Message Writer以确认是否有数据可写。

为了避免检查所有的这些Message Writer，以及那些根本没有消息需要发送给他们的Channel实例，我么可以采用入校两步策略：

1. 当有消息写入到Message Writer忠厚，把它关联的Channel注册到Selector上（如果还未注册的话）。
2. 当服务器有空的时候，可以检查Selector看看注册在上面的Channel实例是否处于可写状态。每个可写的channel，使其Message Writer向Channel中写入数据。如果Message Writer已经把所有的消息都写入Channel，把Channel从Selector上解绑。

这两个小步骤确保只有有数据要写的Channel才会被注册到Selector。

集成（Putting it All Together）

正如你所知到的，一个被阻塞的服务器需要时刻检查当前是否有显得完整消息抵达。

在一个消息被完整的收到前，服务器可能需要检查多次。检查一次是不够的。

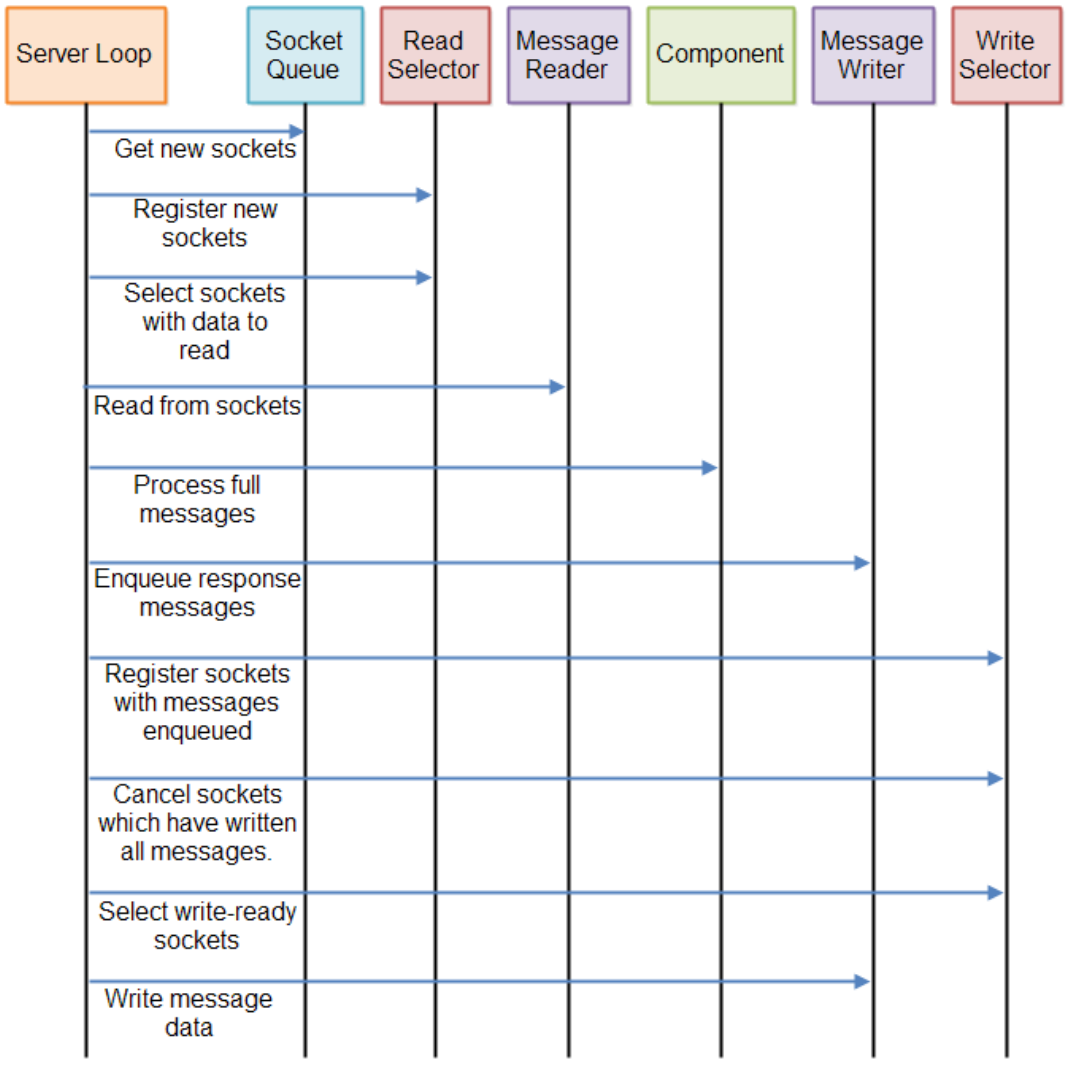
类似的，服务器也需要时刻检查当前是否有任何可写的数据。如果有的话，服务器需要检查相应的链接看他们是否处于可写状态。仅仅在消息第一次进入队列时检查是不够的，因为一个消息可能被部分写入。

总而言之，一个非阻塞的服务器要三个管道，并且经常执行：

- 读数据管道，用来检查打开的链接是否有新的数据到达；
- 处理数据管道，负责处理接收到的完整消息；
- 写数据管道，用于检查是否有数据可以写入打开的连接中；

这三个管道在循环中重复执行。你可以尝试优化它的执行。比如，如果没有消息在队列中等候，那么可以跳过写数据管道。或者，如果没有收到新的完整消息，你甚至可以跳过处理数据管道。

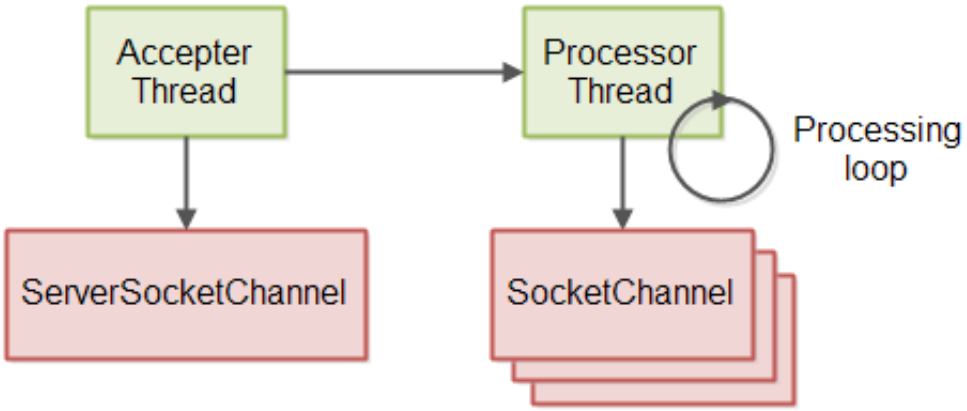
下面这张流程图阐述了这整个服务器循环过程：



假如你还是柑橘这比较复杂难懂，可以去clone我们的源码仓：https://github.com/jjenkov/java-nio-server 也许亲眼看到了代码会帮助你理解这一块是如何实现的。

服务器线程模型（Server Thread Model）

我们在GitHub上的源码中实现的非阻塞IO服务使用了一个包含两条线程的线程模型。第一个线程负责从ServerSocketChannel接收到达的链接。另一个线程负责处理这些链接，包括读消息，处理消息，把响应写回到链接。这个双线程模型如下：



前一节中已经介绍过的服务器的循环处理在处理线程中执行。