## 高并发量催生NIO

一个使用传统阻塞I/O的系统，如果还是使用传统的一个请求对应一个线程这种模式，一旦有高并发的大量请求，就会有如下问题：

* 线程不够用，就算使用了线程池复用线程也无济于事;
* 阻塞I/O模式下，会有大量的线程被阻塞，一直在等待数据，这个时候的线程被挂起，只能干等，CPU利用率很低，换句话说，系统的吞吐量差；
* 如果网络I/O堵塞或者有网络抖动或者网络故障等，线程的阻塞时间可能很长。整个系统也变得不可靠。

HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。

## NIO

java.nio全称java non-blocking IO（实际上是new io），是指JDK 1.4及以上版本里提供的新API（New IO），为所有的原始类型（boolean类型除外）提供缓存支持的数据容器，使用它可以提供非阻塞式的高伸缩性网络。

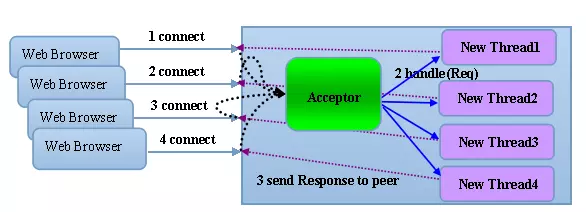
## IO vs NIO

原有的IO是面向流的、阻塞的

java 1.4以前的IO模型，一个连接对应一个线程。处理IO请求的线程面向的对象是IO流（IO流机制本身不存在缓存的概念），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，被读取的数据无法在IO流中被前后移动，除非先将IO流中的数据缓存到一个缓冲区。

另外，Java IO的各种流是阻塞的，这意味着当一个线程调用read或write方法时，该线程被阻塞，直到有一些数据被读取、或数据完全写入，该线程在阻塞等待期间不能再干任何事情了。

NIO是面向块的、非阻塞的



阻塞I/O模型

NIO是面向缓冲区的。数据被读取之前先放入一个缓冲区中，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。

Java NIO的非阻塞读模式，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。

通俗理解：NIO是可以做到用一个线程来处理多个操作的。假设有10000个请求过来，根据实际情况，可以分配50或者100个线程来处理。不像之前的阻塞IO那样，非得分配10000个。

### NIO核心API：Channel Buffer Selector

在标准I/O API中，你可以操作字节流和字符流，但在NIO中，你可以操作通道和缓冲，数据总是从通道被读取到缓冲中或者从缓冲写入到通道中。

#### 通道 Channel

NIO的通道类似于流，但有些区别：

* 通道可以同时进行读写，而流只能读或者只能写；
* 通道可以实现异步读写数据；
* 通道可以从缓冲读数据，也可以写数据到缓冲；

#### 缓存 Buffer

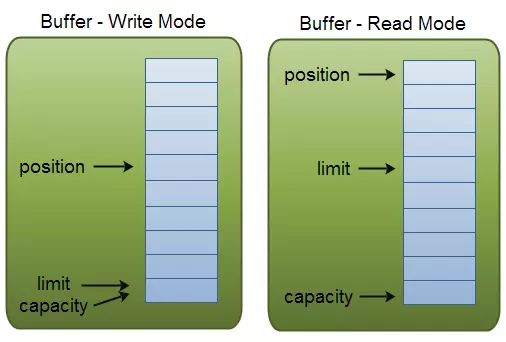
缓冲区本质上是一个可以写入数据的内存块，然后可以再次读取，该对象提供了一组方法，可以更轻松地使用内存块，使用缓冲区读取和写入数据通常遵循以下四个步骤：

1. 写数据到缓冲区；
2. 调用buffer.flip()方法；
3. 从缓冲区中读取数据；
4. 调用buffer.clear()或buffer.compat()方法；

当向buffer写入数据时，buffer会记录下写了多少数据，一旦要读取数据，需要通过flip()方法将buffer从写模式切换到读模式，在读模式下可以读取之前写入到buffer的所有数据，一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。

buffer中的一些属性

|  |  |  |
| --- | --- | --- |
| capacity | buffer的大小/容量 | 作为一个内存块，buffer有一个固定的大小值，用参数capacity表示。 |
| position | 当前读/写的位置 | 当写数据到缓冲时，position表示当前待写入的位置，position最大可为“capacity-1”；当从缓冲读取数据是，position表示从当前位置读取。 |
| limit | 信息末尾的位置 | 在写模式下，缓冲区的limit表示你最多能往buffer里写多少数据；写模式下，limit等于buffer的capacity，意味着你还能从缓冲区获取多少数据。 |



缓冲区常用的操作

* 向缓冲区写数据：

1. 从Channel写到buffer;
2. 通过buffer的put方法写到buffer中

* 从缓冲区读取数据：

1. 从buffer中读取数据到Channel
2. 通过buffer的get方法从buffer中读取数据

* flip方法：

将buffer从写模式切换到读模式，将position值重置为0，limit的值设置为之前position的值；

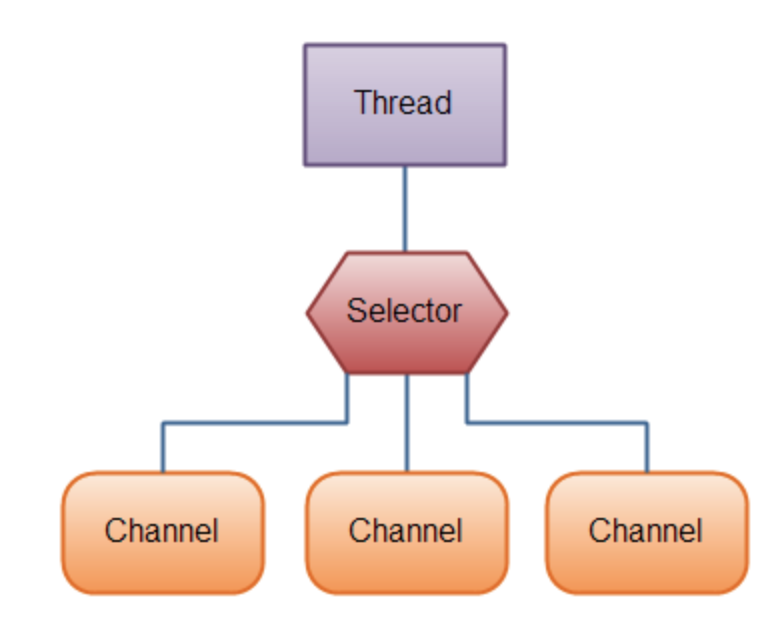
* clear方法 vs compact方法：

clear方法清空缓冲区；compact方法只会清空已读取的数据，而还未读取的数据继续保存在buffer中；

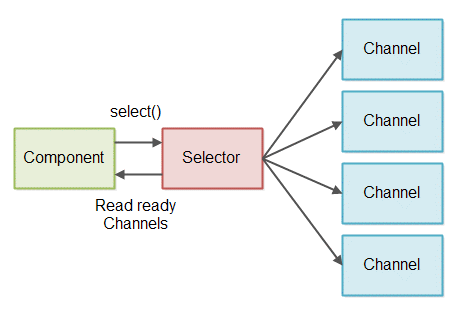
Selector

Selector组件可以检测多个NIO channel，看看读或者写事件是否就绪。

多个Channel以事件的方式可以注册到同一个Selector，从而达到用一个线程处理多个请求成为可能。



一个thread对应多个channel，一个channel处理一个请求。



当你调用Selector的select()或者selectNow()方法，它只会返回有数据读取的SelectableChannel的实例