http://blog.csdn.net/anxpp/article/details/51512200

## 阻塞 非阻塞 同步 异步

1 阻塞和非阻塞是进程在访问数据时，数据是否准备就绪的一种处理方式。当数据没有准备好的时候，

阻塞：往往需要等待缓冲区中的数据准备好之后才能处理其他事情，否则一直等待在那里。

非阻塞：当进程访问数据缓冲区时，数据没有准备好的时候，直接返回，不需要等待，数据有的时候，也直接返回。

2 同步和异步

同步和异步是基于应用程序和操作系统处理IO事件所采用的方式，比如同步：应用程序要直接参与IO读写的操作 异步：所有的IO读写交给操作系统去处理。同步的方式在处理IO事件的时候，必须阻塞在某个方法上等待IO事件完成（阻塞IO事件或者通过轮询IO事件）

，对于异步来说，所有的IO读写都交给操作系统，我们可以去做其他事情，并不需要完成真正的IO操作，当操作系统完成Io后，给应用程序发一个通知或者回调即可。

同步：

1阻塞到IO事件 阻塞read/write，这个时候不能做其他事情。让读写方法加入到线程里面，然后阻塞线程来实现，这样对线程开销比较大。

2 IO事件轮询——多路复用技术(select模式)

将读写事件交给一个单独的线程来处理，这个线程完成IO事件的注册功能，还有就是不断去轮询我们的读写缓冲区，看是否有数据准备好。准备好了，就可以通知相应的读写线程，这样的话，以前的读写线程就可以做其他事情，这个时候阻塞不是阻塞IO，而是阻塞select这个线程。

## Java io模型

BIO：jdk1.4之前使用的都是BIO 阻塞IO 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，服务器端就启动一个线程进行处理，如果这个连接不做任何事情，就会造成不必要的线程开销，可以使用线程池改善。

阻塞读写方法，可以阻塞到线程来提高性能，但是对于线程开销，就是性能的浪费。

NIO：jdk1.4 linux多路复用技术(select模式) 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会注册到多路复用器上，多路复用器轮询到有连接IO请求时才启动一个线程进行处理。实现IO事件的轮询

同步非阻塞方式，目前主流的网络通信方式 如Mina Netty

AIO：jdk1.7（NIO2），异步IO，利用linux的epoll模型实现。服务器实现模式为一个有效请求一个线程。客户端IO请求都是由操作系统先完成了再通知服务器应用去启动线程进行处

理。

相对于NIO来说，AIO是在数据读取或者写入调用已经完成的时候再去通知调用者，也就是数据已经从内核空间拷贝到用户空间了再去通知调用者。而多路复用IO是在有数据就绪时，可以读写的时候就会通知调用者，读写仍然是由调用者执行并且是阻塞的（这意味着如果要同时进行其他操作，要控制读写操作不能阻塞太长时间或者放到单独的线程中执行）

适应场景：

BIO适应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，并发局限在应用中

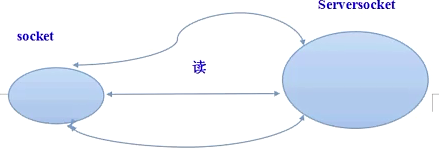
NIO适应于连接数目较多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限在应用中

AIO适应于连接数目多且连接比较长（重操作）架构，比如相册服务器，充分调用操作系统参与并发操作

## NIO AIO原理

对于网络通信而言，NIO AIO并没有改变通信的基本步骤，只是在原来的基础上（serversocket socket）做了改进。

Socket和ServerSocket三次握手连接，开销比较大。解决方案：从思想上来说比较容易，减少连接次数，对读写通信管道进行一个抽象。



对于读和写，采用抽象的管道概念。

Channel是在一个TCP连接之间的抽象，一个TCP连接可以对应多个管道，而不是以前的方式只有一个通信信道，减少了TCP连接的次数。

UDP采用相同的方式，也抽象成 管道。

### NIO原理

通过selector选择器，管理所有的IO事件。

Connection accept 客户端和服务器的读写——IO事件

Selector如何管理IO事件？

当IO事件注册到选择器时，选择器会给它们分配一个key（相当于一个事件的标签），当IO事件完成后，会通过key值找到相应的管道，然后通过管道发送数据和接收数据等操作。

数据缓冲区：通过ByteBuffer实现。 put() get()

服务端：ServerSocketChannel

客户端：SocketChannel

选择器：Selector selector = Selector.open() 打开选择器

SelectionKey：可以通过它来判断IO事件是否已经就绪

Key.isAcceptable 是否可以接收客户端连接

Key.isConnectionable 是否可以连接服务端

Key.isReadable 缓冲区是否可读

Key.isWriteable 缓冲区是否可写

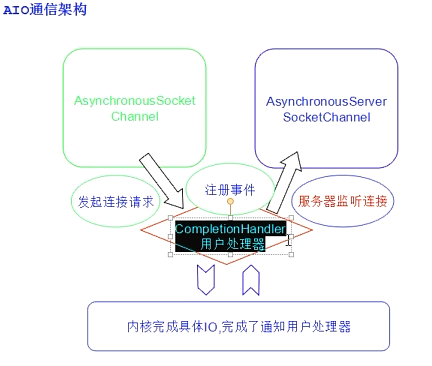
如何获得事件keys

keys = Selector.selectedKeys();

如何注册

Channel.regist(Selector,SelectionKey.OP\_Write | ….);

### AIO原理



服务端：AsynchronousServerSocketChannel

客户端：AsynchronousSocketChannel

用户处理器：CompletionHandler接口，这个接口实现应用程序向操作系统发送IO请求，当完成后处理具体逻辑，否则做自己该做的事情。这是一个回调接口，在socket进行accept/connect/read/write等操作时，可以传入一个CompletionHandler的实现，操作执行完毕后，会调用注册的CompletionHandler

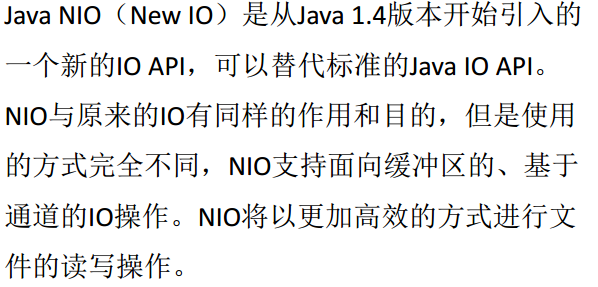
除了CompletionHandler这种回调，还支持Future对象，使用Future来设定回调操作

方法 void completed(V result,A attachment) 当操作完成后，被调用

void failed(Throwable exc,A attachment) 当操作失败后，被调用

http://blog.csdn.net/anxpp/article/details/51512200

## NIO简介



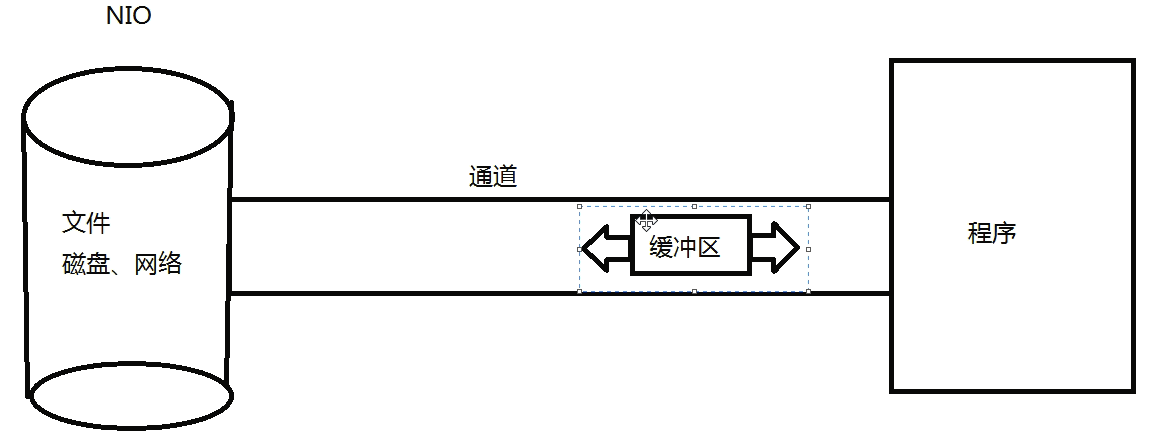
## NIO和IO区别



理解：

1 传统IO是面向流的，也就是说从流中一次可以传输一个或者多个字节，而且只能顺序从流中读取数据，如果需要跳过一些字节或者再读取已经读过的字节，必须将流中读取的数据先缓存起来。传统IO是单向的，分为输入流和输出流。

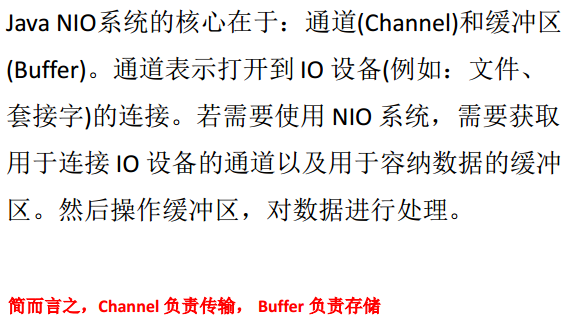
NIO是面向块的，数据会先被读/写到缓冲区buffer中，可以根据需要控制读取什么位置上的数据。



传统IO类似于水流，需要有个水管，流直接在水管中传输。而NIO是铁道，本身不具备传输功能，只是起到连接作用，需要将数据放到缓冲区中进行传输。

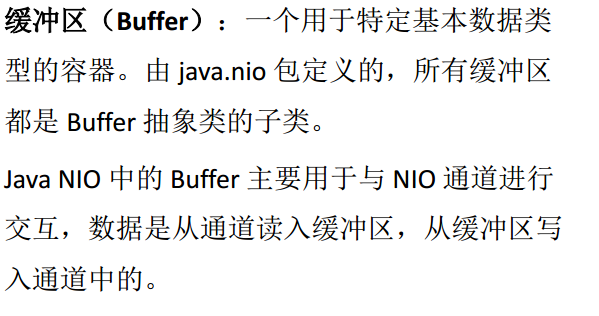
2 阻塞和非阻塞，选择器是针对网络IO区别而言的。

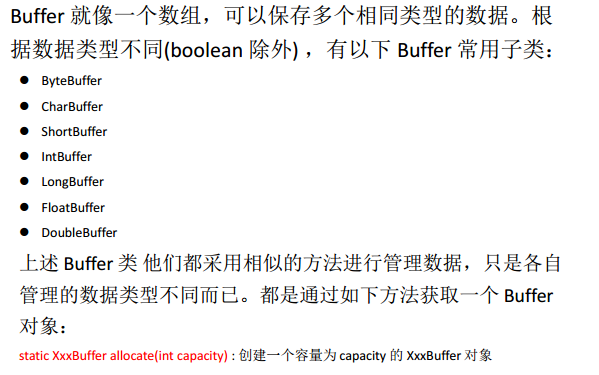
## 通道和缓冲区



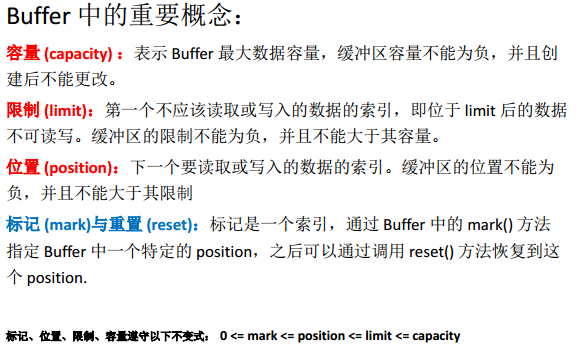
## 缓冲区

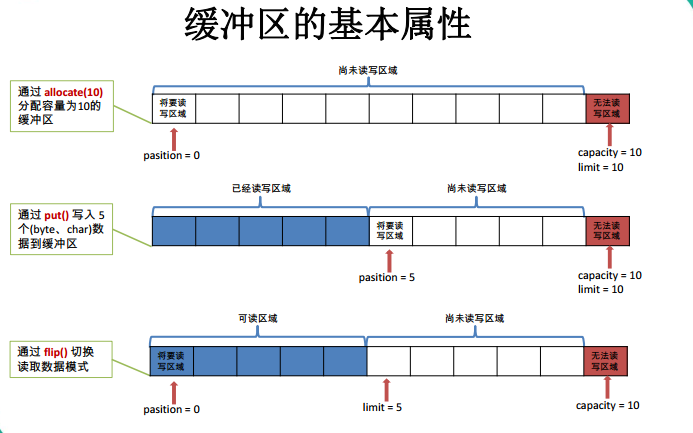
### 概念





### 重要属性



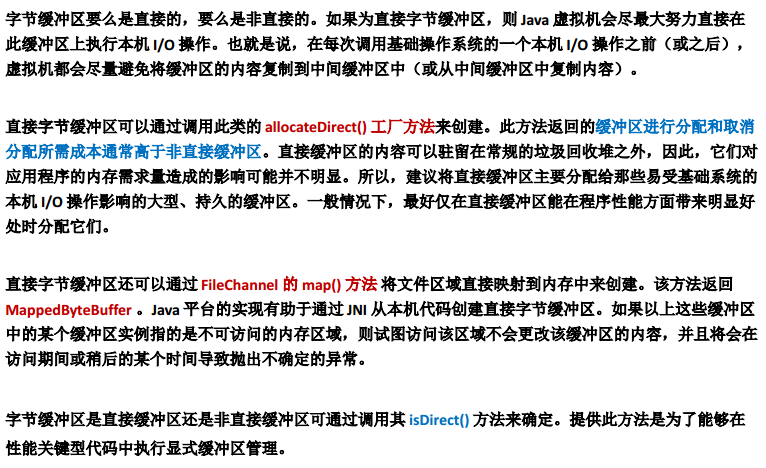


### buffer中常用的方法

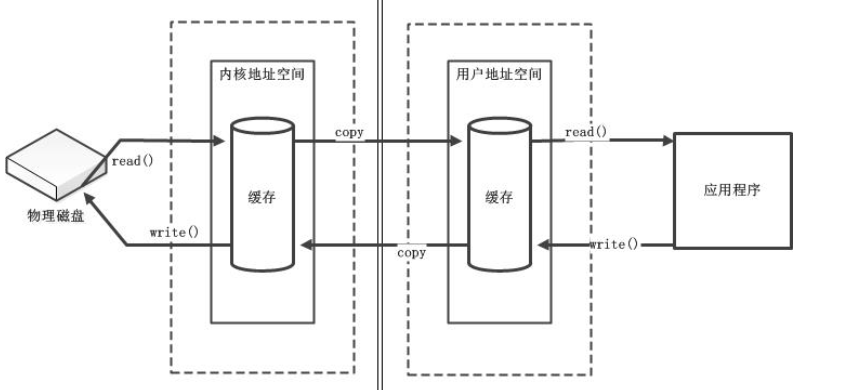


### 直接缓冲区和非直接缓冲区

#### 概念

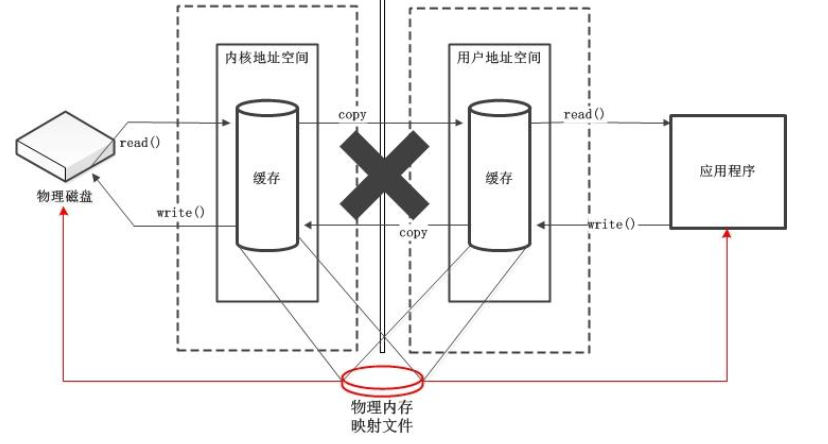


#### 非直接缓冲区



读取时，先将数据读取到内核地址空间中，然后将内核地址空间的数据copy到用户地址空间中（JVM中），然后给应用程序使用。写入数据时，过程相反。

#### 直接缓冲区

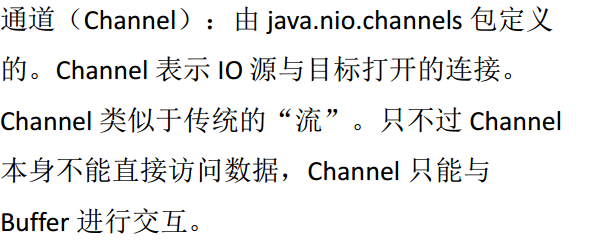


在物理内存中直接开辟了缓冲区。这样省去了copy的过程。

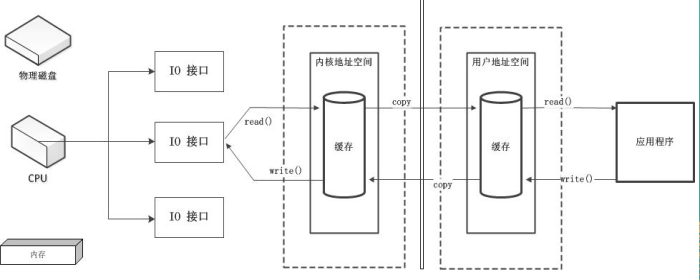
缺点：不安全，分配和销毁时，消耗的资源大。只有当垃圾回收时，也就是full gc时，才会回收这块区域。只有当有一些数据，长时间需要存放时，可以将数据放在直接内存中。

## 通道

### 概念

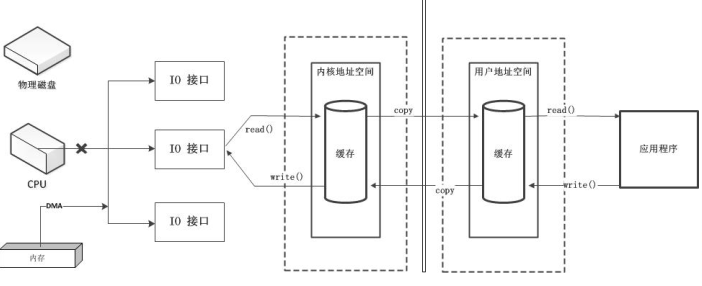


刚开始的操作系统，是直接由CPU负责IO连接，这样当有大量IO请求时，就会占用CPU。这样并没有充分利用CPU利用率。

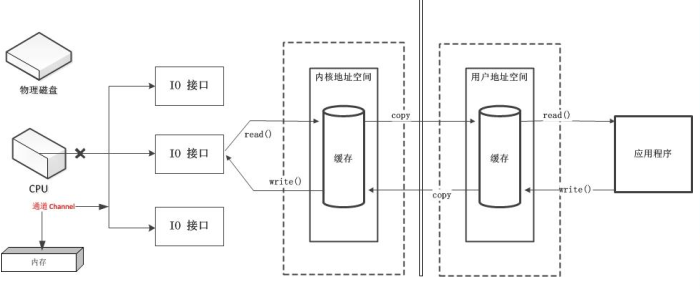


后来，操作系统在内存和IO之间引入了DMA（直接存储器存储）。当发送IO请求时，DMA会先向CPU申请权限，然后之后的读取操作完全由DMA负责，CPU不需要干预。

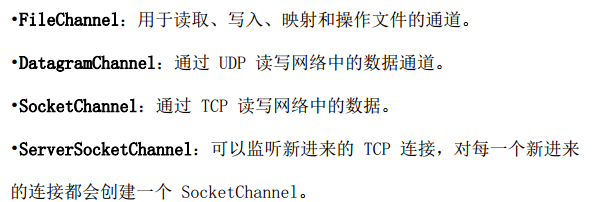
但是当有大量请求时，会建立大量的DMA总线，会导致DMA总线冲突，从而性能下降。



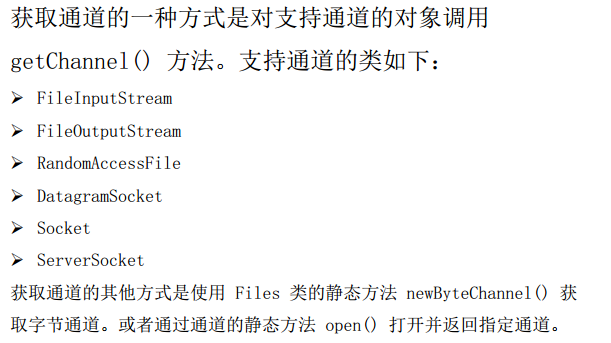
后来，引入了通道。通道是一个完全独立的处理器。专门用于IO操作。有自己的命令。这样当来了大量CPU请求，也不需要向CPU申请权限了。



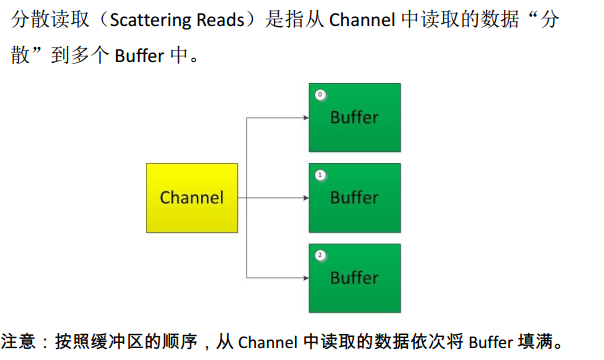
### 主要实现类

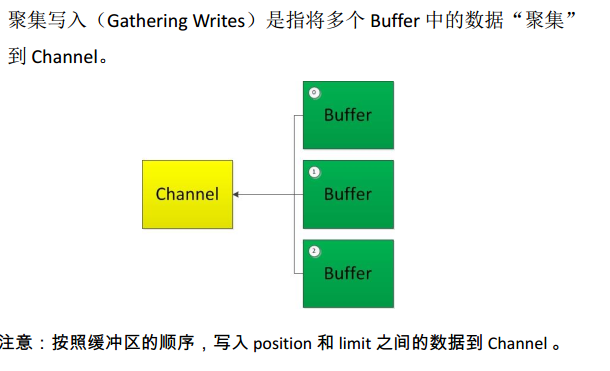


### 获取通道的方法



### 分散与聚集





### TranserFrom



### TransferTo



### 常用方法



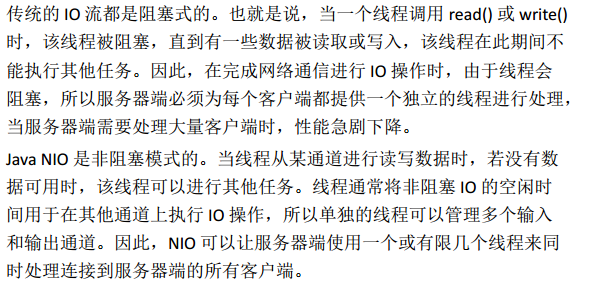
## NIO非阻塞式通信

### 简介

1 client向server端建立连接后，server端的工作线程会一直阻塞在read/write上，直至有数据进行操作时，才不会阻塞。

解决方案：在server端开启多个线程，一个线程对应一个连接。但是在一个线程阻塞期间，仍然不可以做其他操作，并没有充分利用cpu。而且当系统负载增大，并发请求数增多时，在server端需要的线程数就会增加，这会成为系统扩展的瓶颈。（一个机器所能开启的线程数是有一个合适的范围的，如果超过了，可能会导致性能下降）

2 NIO是非阻塞。多个IO通道会注册在selector选择器上，selector选择器负责监控这些通道的状态(轮询)，只有当通道中的数据已经准备好了之后，才会将这个通道的任务分配给服务器端一个或多个线程上。在此之前，服务器端的线程可以处理自己的事情。

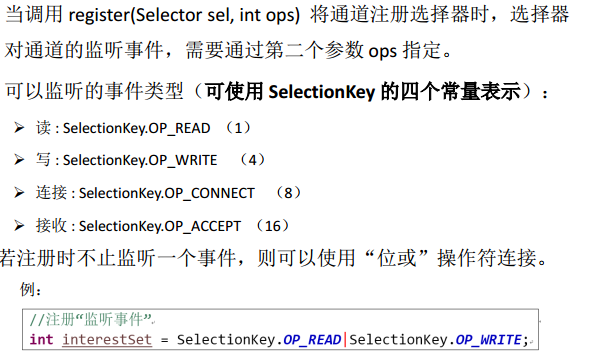


这里的非阻塞指的是网络IO，对于FileChannle是无法切换到非阻塞模式。

非阻塞指的是是IO事件本身不会阻塞，但是在获取IO事件的select方法是阻塞的。IO是阻塞在IO操作上，NIO是阻塞在事件获取上，没有事件就没有IO。NIO本质上是延迟IO操作到真正发生IO的时候，而不是只要IO流打开了就会等待IO操作。

### 选择器selector





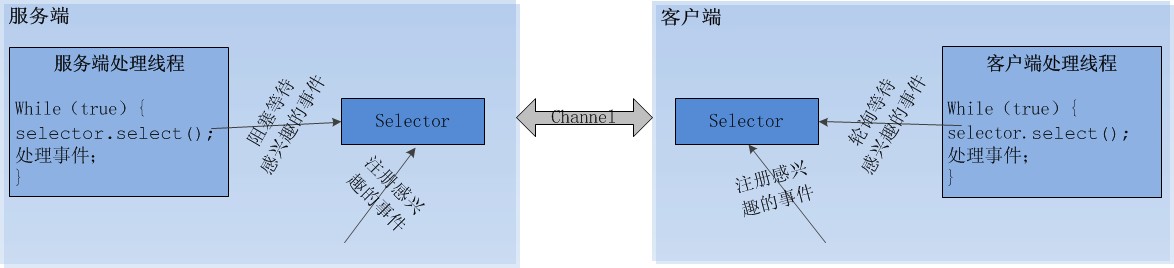
一个selector对象包含3种类型的SelectionKey集合：

All-keys集合：当前所有向selector注册的SelectionKey的集合，selector.key()

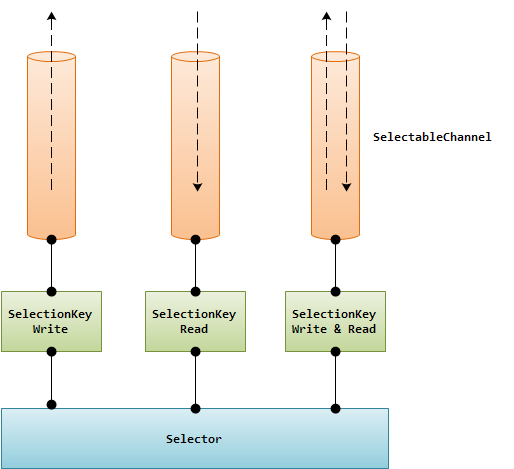
Selected-keys集合：相关事件已经被selector捕获的SelectionKey的集合，selector.selectedKeys()

Cancelled-keys集合：已经被取消的SelectionKey的集合

服务端和客户端各自维护一个管理通道的对象，我们称之为selector，该对象能检测一个或多个通道 (channel) 上的事件。我们以服务端为例，如果服务端的selector上注册了读事件，某时刻客户端给服务端发送了一些数据，阻塞I/O这时会调用read()方法 阻塞地读取数据，而NIO的服务端会在selector中添加一个读事件。服务端的处理线程会轮询地访问selector，如果访问selector时发 现有感兴趣的事件到达，则处理这些事件，如果没有感兴趣的事件到达，则处理线程会一直阻塞直到感兴趣的事件到达为止。



当有读或写等任何注册的事件发生时，可以从Selector中获得相应的SelectionKey，同时从 SelectionKey中可以找到发生的事件和该事件所发生的具体的SelectableChannel，以获得客户端发送过来的数据。 使用NIO中非阻塞I/O编写服务器处理程序，大体上可以分为下面三个步骤：   
1. 向Selector对象注册感兴趣的事件   
2. 从Selector中获取感兴趣的事件  
3. 根据不同的事件进行相应的处理



### SelectionKey



在selectionkey对象的有效期间，selector会一直监控与SelectionKey对象相关的事件，如果事件发生，就会把SelectionKey对象加入到selected-keys集合中。

在以下情况下，SelectionKey对象会失效，也就是说selector不会再监控与它相关的事件：

1 程序调用SelectionKey的cancel()方法

2 关闭与SelectionKey关联的Channel

3 与SelectionKey关联的Selector被关闭

SelectionKey中定义的四种事件：

1 SelectionKey.OP\_ACCEPT 接收连接继续事件，表示服务器监听到了客户端连接，服务器可以接收这个连接了

2 SelectionKey.OP\_CONNECT 连接就绪事件，表示客户端和服务器的连接已经建立成功

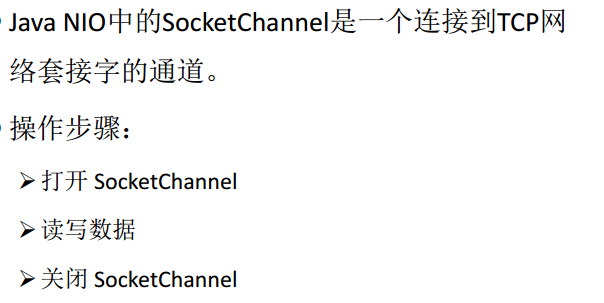
3 SelectionKey.OP\_READ 读就绪事件，表示通道中已经有了可读的数据，可以执行读操作

4 SelectionKey.OP\_WRITE 写就绪事件，表示已经可以向通道中写数据了

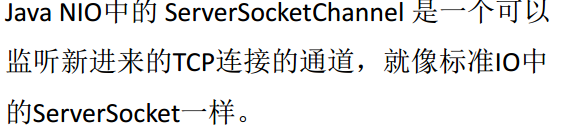
### Selector常用方法



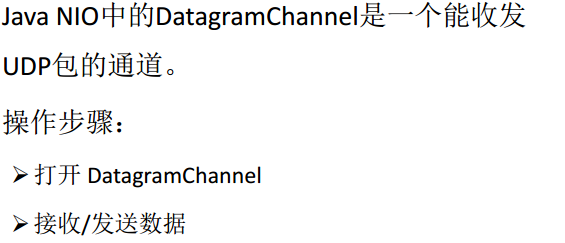
### SocketChannel



### ServerSocketChannel



### DatagramChannel



### DatagramChannel



### 管道PIPE

