# 1 IO和NIO的区别

原文链接：<https://www.cnblogs.com/xiaoxi/p/6576588.html>

1. IO面向流，IO读取的字节没有被缓存到任何地方；NIO面向缓冲，NIO将读取到的字节缓存到缓冲区中，数据可以在缓冲区中前后移动。
2. IO的各种流是阻塞的，执行read()或write()的线程会一直阻塞至有数据读取或将数据完全写入，执行accept()的线程会一直阻塞到有新的连接进来。NIO有阻塞模式和非阻塞模式，如果没有数据读取，该线程不会阻塞，可以做其他事情，数据写入时先写到通道，然后该线程不会阻塞，可以做其他事情，不需要等待数据完全写入。
3. 就算用一般的多线程优化IO，IO中一个线程只能处理一个客户端连接（一个服务员对一个客人）；但NIO中一个线程可以通过Selector来处理多个客户端连接（一个服务员对多个客人）。

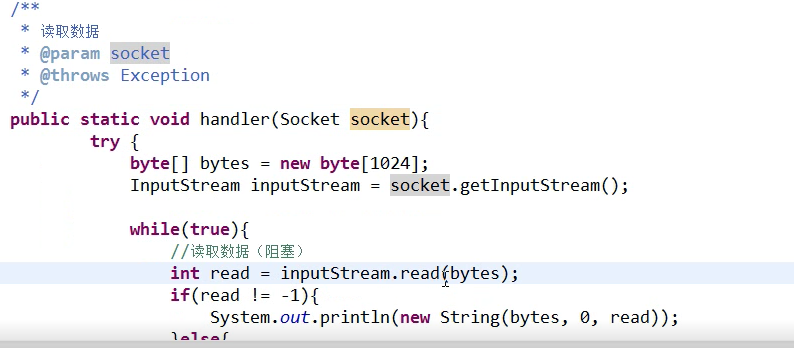
## 传统IO的缺点

1. IO的各种流是阻塞的，执行read()或write()的线程会一直阻塞至有数据读取或将数据完全写入，执行accept()的线程会一直阻塞到有新的连接进来。

图1



图2

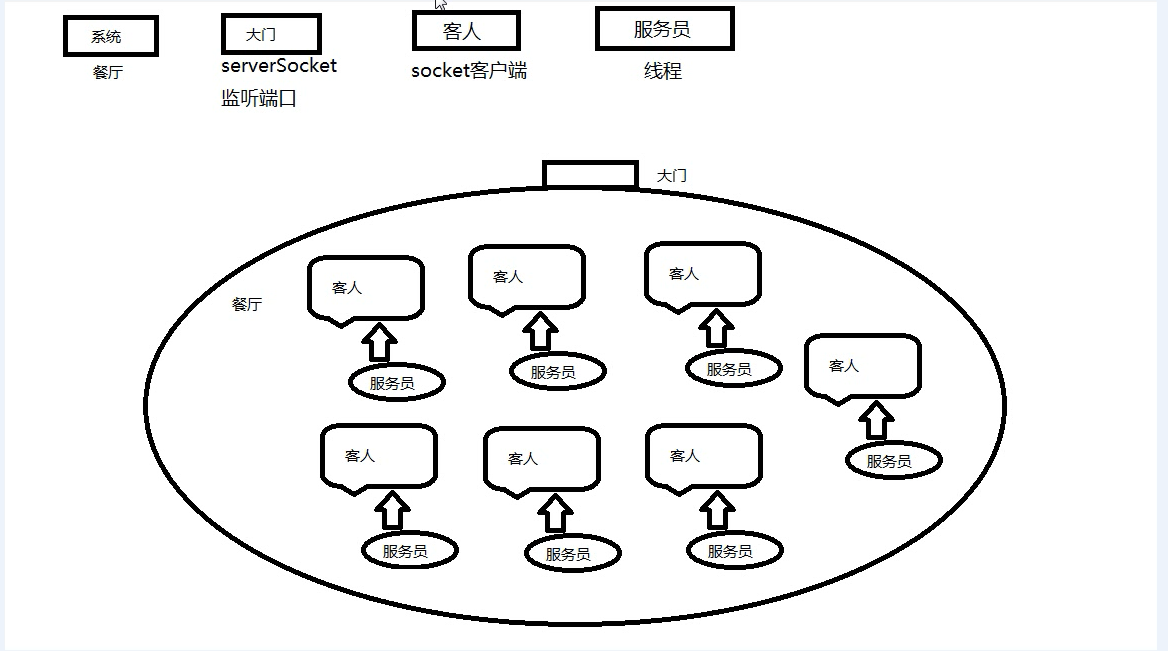


解决：（1）建立线程池，一个线程处理一个客户端Socket，这样就不会阻塞后面的客户端Socket（缺点：相当于餐厅里面一个服务生同时只能服务一个客人，这样线程的消耗太多；不适合长连接，而短连接时线程释放快这种方法可行）。

图3



图4

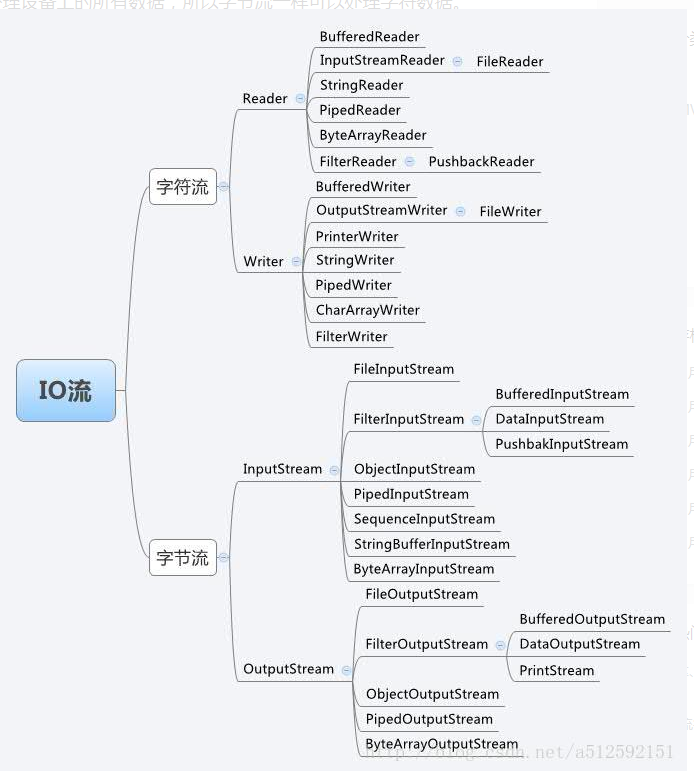


# 2 IO流分类

图1



图2



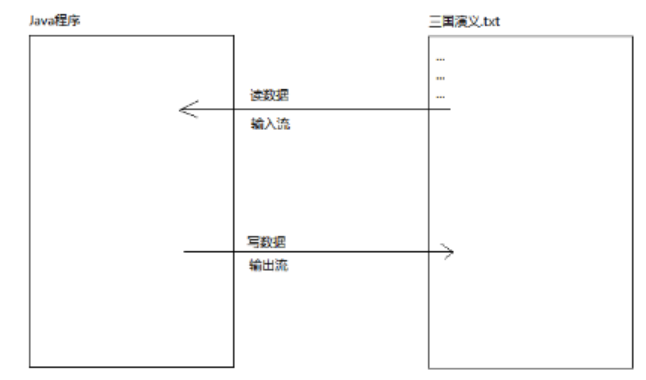
## 2.1 输入流和输出流

根据数据的流向分为：输入流和输出流。

输入流：xxx------->程序

输出流：程序------->xxx

图1



## 2.1 节点流和处理流

根据读/写的程序另一端类型分为：节点流和处理流。

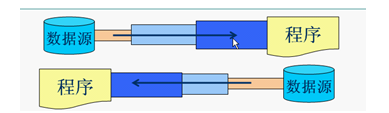
节点流：程序另一端是数据源（文件，网络等）。

图1



处理流（过滤流）：程序另一端是节点流。

图2



## 2.2 字节流和字符流

根据读取数据单位分为：字节流和字符流

字节流：每次读取一个或若干个字节。

字符流：每次读取字符单位的字节（每个字符的字节数都不一样）。

# 3 同步和异步，阻塞和非阻塞的区别和关系

## 内核操作和应用程序操作

内核操作：

将文件数据读到内核空间，或将内核空间的数据写到文件中。

（注意：内核操作是文件到内核空间的数据交换）

（注意：内核操作会占用CPU的其中一个核）

（注意：内核操作过程由内核完成，应用程序的线程不需要参与，可以选择阻塞或不阻塞）

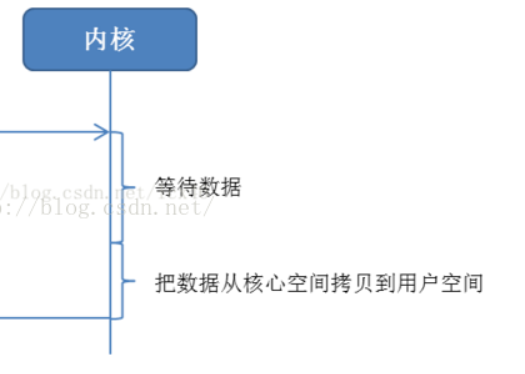
应用程序操作：

将内核空间的数据读到用户空间，或将用户空间的数据写到内核空间。

（注意：应用程序操作是用户空间到内核空间的数据交换）

（注意：应用程序操作由应用程序的线程完成，这个阶段线程一定会阻塞）

图1



## 3.2 同步，异步

原文链接：<https://www.cnblogs.com/George1994/p/6702084.html>

同步，异步判断的依据是：对于一次操作，是否等待应用程序操作的结果的返回。

同步：A程序发起操作后，必须等待应用程序操作的返回，才得到返回值。

异步：A程序发起操作后，立刻得到返回值；当应用程序操作完成后，发通知给A程序。

（注意：可以假设应用程序操作的过程很长，如果A是等待一段时间后返回，那么是同步，如果是立刻返回，则是异步）

## 3.3 阻塞，非阻塞

原文链接：<https://www.cnblogs.com/George1994/p/6702084.html>

阻塞和非阻塞判断的依据是：对于一次操作，在内核操作期间，执行操作的线程是否挂起。

阻塞：内核操作期间，该线程挂起，不能干别的事情。

非阻塞：内核操作期间，该线程不会挂起，可以去干别的事情。

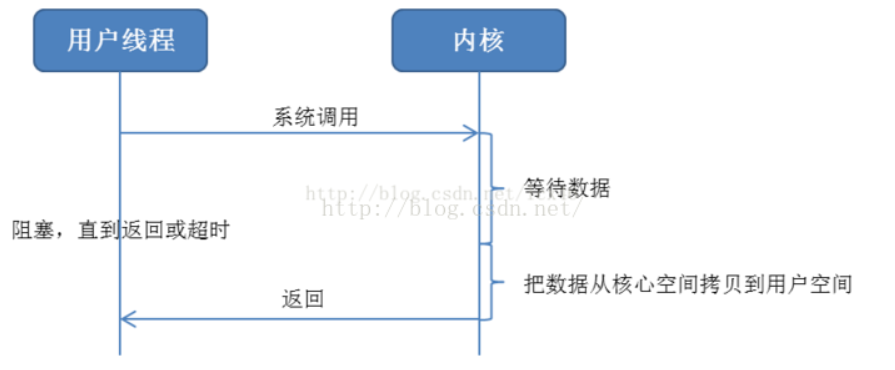
（注意：阻塞和不阻塞是主要针对内核操作期间，因为应用程序操作期间一定会阻塞，这个阶段是应用程序要将内核空间的数据读取到用户空间中）

## 3.4 同步阻塞IO（BIO），同步非阻塞IO（NIO）

原文链接：<https://www.cnblogs.com/songjy2116/p/7711368.html>

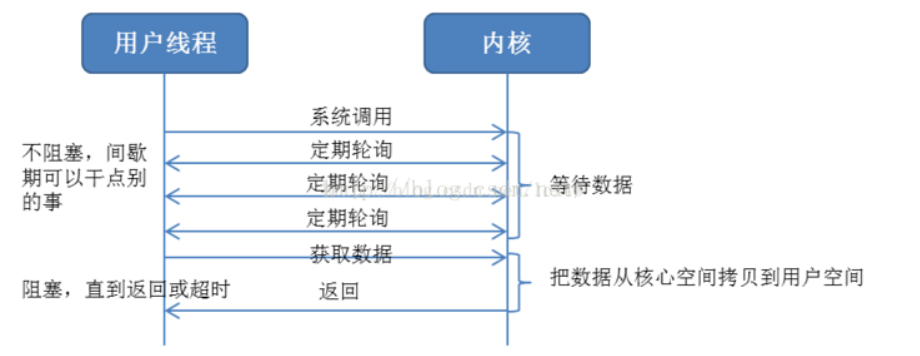
同步阻塞IO（BIO）：同步即应用程序操作期间，在等待结果返回；阻塞即内核操作期间，线程挂起，不能干别的事情。

图1



同步非阻塞IO（NIO）：同步即应用程序操作期间，在等待结果返回；非阻塞即内核操作期间，线程不挂起，可以干别的事情。

图2



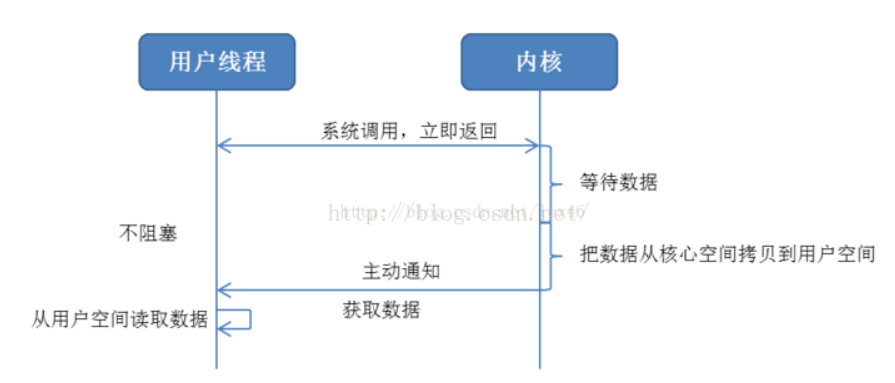
## 3.5 异步阻塞IO，异步非阻塞IO（AIO）

异步阻塞IO：感觉无法实现，因为异步就是发起操作后立刻返回，而阻塞需要在内存操作期间挂起。

异步非阻塞IO（AIO）：异步即发起操作后，立刻返回；非阻塞即内核操作期间，线程不挂起，可以干别的事情。

（注意：图1有误，就算是异步非阻塞IO，在应用程序操作期间（即“把数据从内核空间拷贝到用户空间”）这个阶段是应用程序的线程区做的，一定会阻塞；实际上这个图的主动通知应该画在内核操作之后，应用程序操作之前）

图1



# 4 NIO

推荐文章链接：<https://segmentfault.com/a/1190000006824155>

当服务器端接收到一个客户端连接时，接收到连接的那个线程Thread会使用Selector选择一个就绪的Channel。

然后这个就绪Channel不管是和客户端Socket还是服务器端Socket交换数据，都通过Buffer。

图1

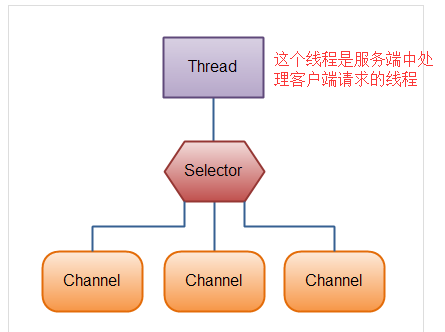
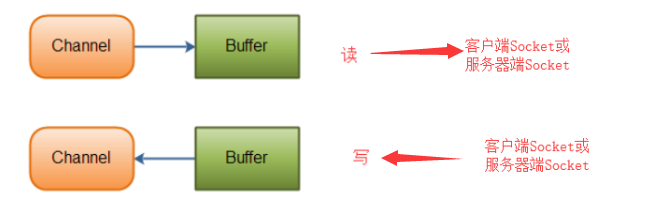


图2



## 4.1 Channel

Channel就是通道：一端连接缓冲区Buffer（Buffer再过去就是线程Thread），另一端连接实体（文件或套接字）。

### 4.1.1 Channel和Stream的区别

1.Channel是通道，是双向的，既可以读又可以写；Stream是流，是单向的，规定从文件流向程序，或从程序流向文件。

### 4.1.2 Channel的两端

Channel的一端是缓冲区Buffer，另一端是实体（实体一般是文件或套接字）

### 4.1.4 Channel的实现类

Channel是一个接口，有不同的实现类读取不同的实体。

图1

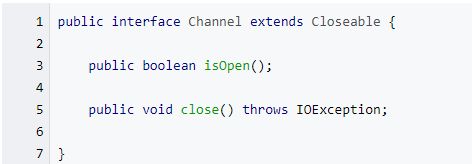
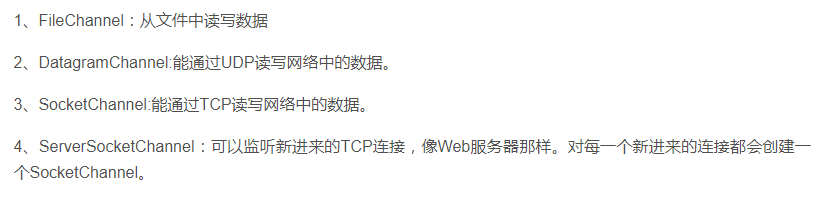


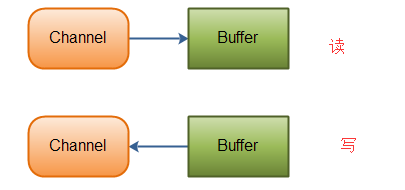
图2



### 4.1.5 Channel和Buffer的区别

1.用户不会直接读写Channel，而是直接读写Buffer。

图1



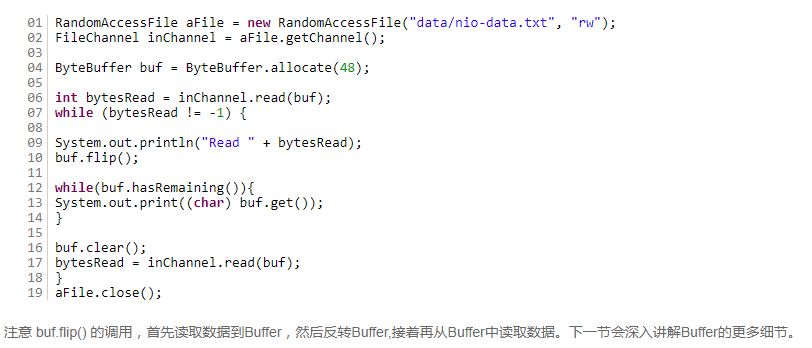
### 4.1.6 FileChannel

FileChannel的特点：

1. FileChannel是线程安全的，因为影响通道位置或文件大小的操作都加了锁，在这些方法上是单线程的；其他方法上可以并发执行。
2. FileChannel是阻塞的，没有非阻塞的选择。

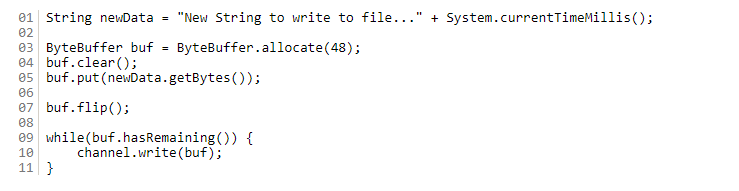
read()从FileChannel读取数据：

图1



write()向FileChannel写数据：

图2



close()关闭FileChannel，用完FileChannel要关闭：

图3



position()从FileChannel的特定位置读取数据：

(注意：FileChannel的position位置其实就是文件中的position位置，和Buffer中的position不是一个东西)

图4



size()获取FileChannel关联的文件大小：

图5



truncate()截取文件大小，指定长度后面的部分会被删掉：

图6



force()将通道里未写入磁盘的数据强制写入磁盘中，这是因为出于性能的原因，写到通道的数据不一定会即时写入磁盘中。true表示写入文件元数据。

图7



### 4.1.7 SocketChannel

SocketChannel就是一端是Buffer，另一端是TCP网络客户端套接字。

open()创建SocketChannel：

图1



close()关闭SocketChannel，用完SocketChannel要关闭：

图2



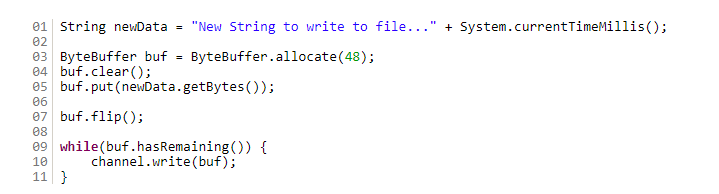
read()从SocketChannel读取数据：

图3



write()向SocketChannel写入数据：

图4



### 4.1.8 ServerSocketChannel

ServerSocketChannel就是一端是Buffer，另一端是TCP网络服务器端套接字。

open()创建ServerSocketChannel：

图1



close()关闭ServerSocketChannel：

图2



accept()监听SocketChannel的连接：

（注意：默认是阻塞模式，accdept()会一直阻塞直到有新的连接，如图3；可以配置成非阻塞模式，accept()会立刻返回，如果没有新的连接则返回null，如图4）

图3

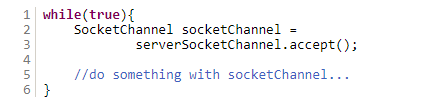
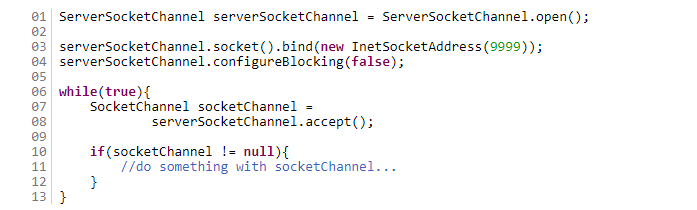


图4



### 4.1.9 DatagramChannel

DatagramChannel就是一端是Buffer，另一端是UDP网络套接字。

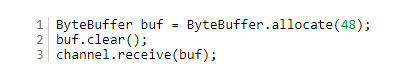
open()创建DatagramChannel：

图1



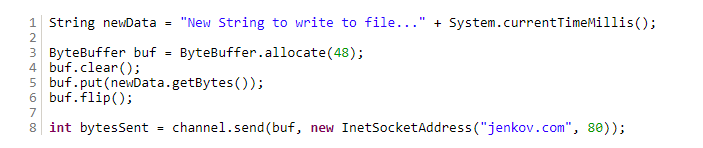
receive()从DatagramChannel读取（接收）数据，会从DatagramChannel读取数据到Buffer中：

图2



send()向DatagramChannel写入（发送）数据：

图3

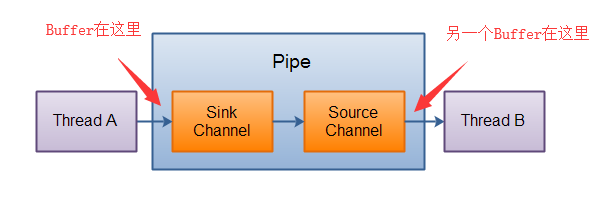


### 4.1.10 Pipe

原文链接：<http://ifeve.com/pipe/>

Pipe是用于2个线程之间的单向数据传输。

图1



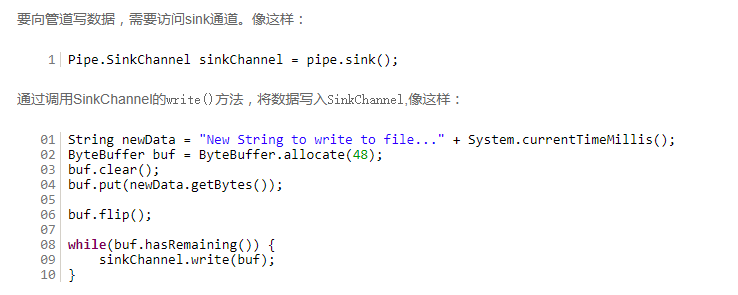
open()创建管道：

图2



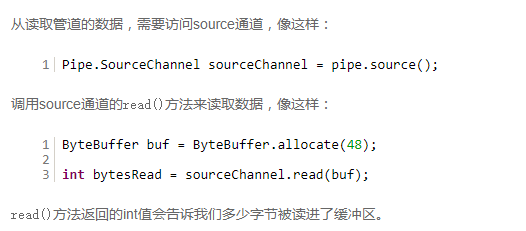
write()向sink通道写入数据：

图3



从source()通道读取数据：

图4



## 4.2 Buffer

原文链接：<http://ifeve.com/buffers/>

Buffer就是缓冲区，可以向其写入数据，也可以从中读取数据。

（注意：如果刚写完数据进缓冲区，然后要读取缓冲区中的内容，需要切换到读模式）

### 4.2.1 Buffer的基本用法

原文链接：<http://ifeve.com/buffers/>

1.将数据写入到Buffer中。

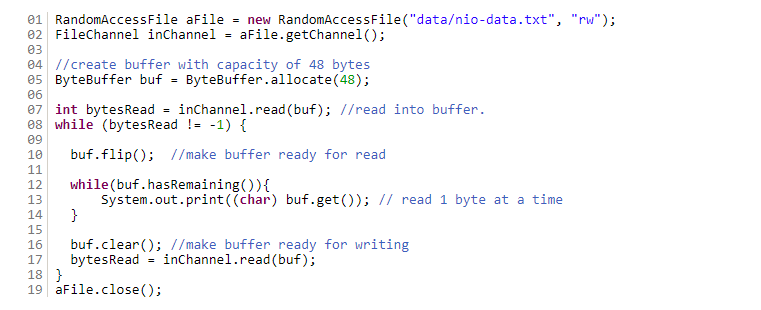
2.调用Buffer的flip()方法将Buffer切换到读模式。

3.从Buffer中读取刚刚写入的数据。

4.调用Buffer的clear()或compact()方法删除Buffer中的内容，并将Buffer切换到写模式。

（注意：为什么需要先删除Buffer中的内容，再写入，是因为Buffer是固定大小的，如果里面的内容不删除，新的内容就写不进去了）

图1



### 4.2.2 clear()和compact()的区别

1.clear()会清空整个Buffer缓冲区中的内容；compact()只会清空Buffer缓冲区中已经读过的内容。

### 4.2.3 Buffer的原理

Buffer是堆中的一块可以读写的内存块（长得有点像数组，但不确定是数组，数组中的类型根据不同的实现类决定，比如ByteBuffer的类型就是Byte）。

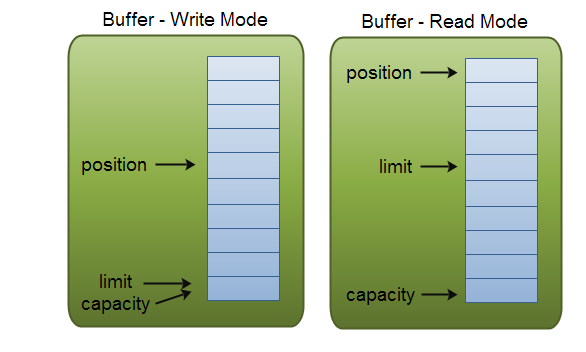
Buffer的几个属性：

1. capacity：Buffer的固定大小，只能存capacity个byte，long，char等类型的值。

（注意：如果Buffer满了，需要删除数据才能继续写入新的数据）

1. position：表示读/写操作的当前位置，当读/写了一个元素时，会自动移动到下一个位置；如果从写模式切换到读模式，position会重置为0，最大为capacity-1。
2. limit：表示最多能读/写的数据量。写模式下，当Buffer是空时，能写入最多数据量，所以limit=capacity；如果从写模式切换到读模式后，此时的读模式最多能读写的数据量就是刚刚写模式下的position，limit会被设置成刚刚写模式下的position。

图1



### 4.2.4 分散（scatter）和聚集（gather）

分散（scatter）：将一个Channel的数据读取到多个Buffer中。

（注意：是读满前面一个Buffer，剩下的继续读取到下一个Buffer，并不是每一个Buffer的内容都相同）

图1

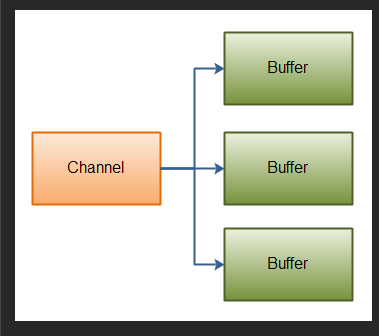
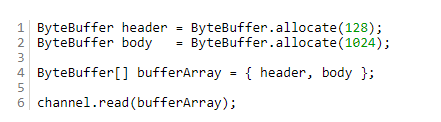


图2



聚集（gather）：将多个Buffer的数据写入到一个Channel中。

图3

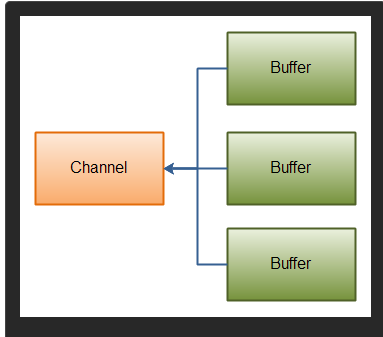
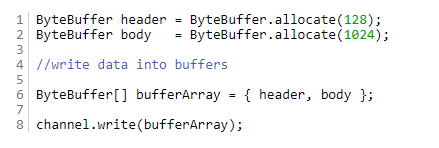


图4



### 4.2.5 Channel之间的数据传输

如果2个Channel之间有一个是FileChannel，可以将数据从一个Channel传输到另一个Channel。

图1

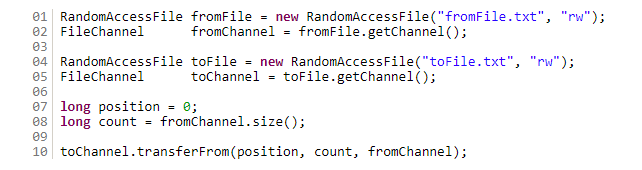
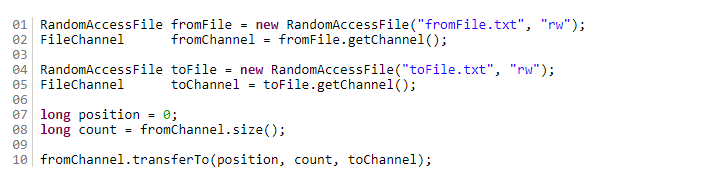


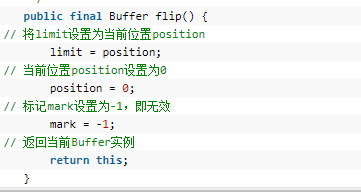
图2



### 4.2.6 Buffer读写模式切换

flip()方法将Buffer从写模式切换到读模式：最多可读数据量limit会被设置成刚刚写模式下的position，当前位置position会重置为0。

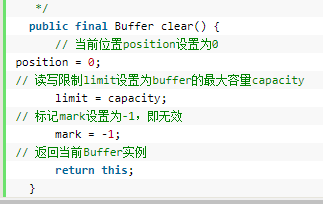
图1



clear()或compact()方法将Buffer从读模式切换到写模式：最大可写数量limit会被设置成Buffer的固定大小capacity，当前位置position会重置为0。

（注意：clear()方法实际不删除Buffer中的数据）

图2



### 4.2.7 rewind() 重读数据

rewind()将当前位置position重置为0，但最多可读数量limit不变，实现重读。

## 4.3 Selector

原文链接：<http://ifeve.com/selectors/>

因为如果一个单个线程处理单个通道会建立太多线程，Selector就是为了用单个线程处理多个通道。

创建一个Seletor：

图1

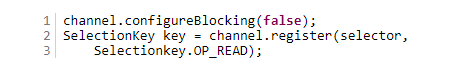


### 4.3.1 Channel和Selector的关系

通道（Channel）需要注册到某个Selector上。

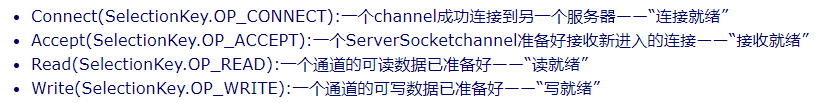
（注意：注册到Selector上的Channel必须是非阻塞的，所以FileChannel不行）

图1



第二个参数是指Selector对这个Channel的哪些事件感兴趣。

图2



### 4.3.2 SelectionKey

通道向Selector注册时，会返回一个SelectionKey。

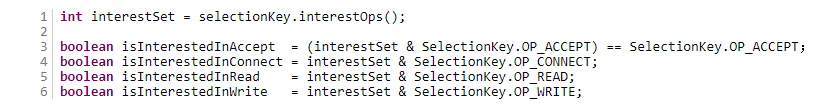
（注意：多个通道可能到注册到同一个Selector中，而SelectionKey是保存一个Channel的信息，不是保存一个Selector的信息）

以下是SelectionKey保存的关于Channel的一些信息：

interest集合：Channel注册到Selector时的事件集合。

（注意：这个事件是从Channel这个通道的角度来看的，比如这个通道可读/可写，而不是对于客户端和服务器端Socket来说的）

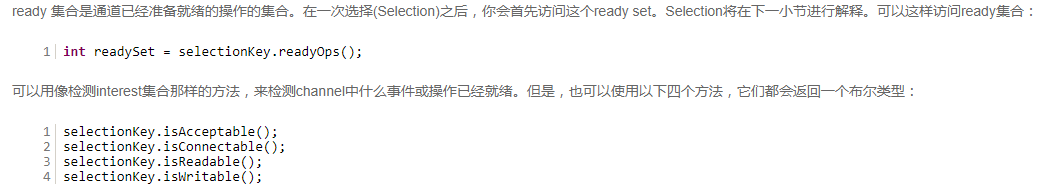
图1



ready集合：已经就绪的事件集合。可以用readyOpts()获取就绪的事件集合，也可以用每个事件的就绪方法判断单个事件是否就绪。

（注意：interest集合是该Channel感兴趣的所有事件集合，而ready集合是该Channel感兴趣的且就绪的事件集合）

图2



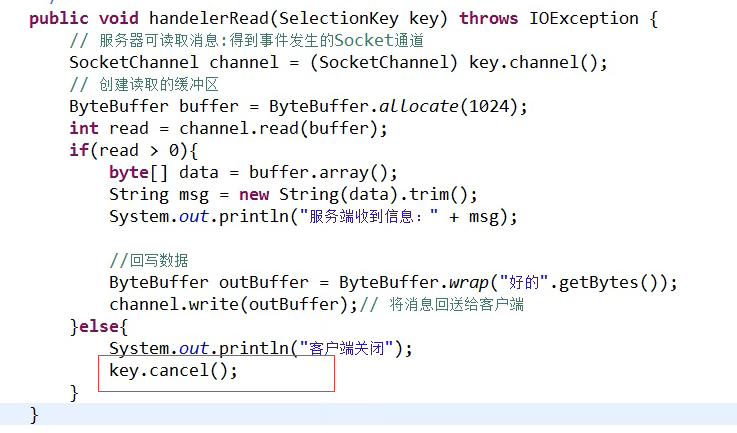
用SelectionKey访问Channel和Selector：

图3



cancel()方法会关闭这个SelectionKey绑定的通道Channel：

图4

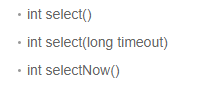


### 4.3.3 选择通道

可以通过重载的select方法获取感兴趣事件的就绪通道个数。

（注意：它只会返回那些在感兴趣的事件上就绪的通道）

图1



select()：会一直阻塞，直到至少一个通道在感兴趣的事件上就绪了。

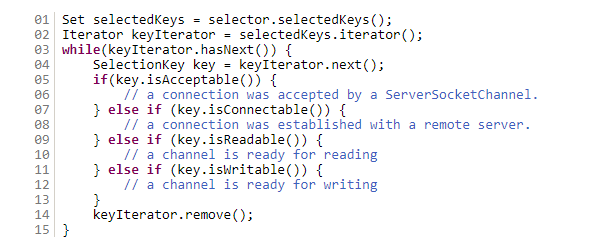
select(long timeout)：和select()一样会阻塞，但最长阻塞timeout毫秒。

selectNow()：不会阻塞，会立刻返回，如果没有通道在感兴趣的事件上就绪则返回0。

由于select只是获取就绪通道个数，具体获取通道要用selectKeys()方法。

（注意：图2的if判断是因为每个通道感兴趣的事件都不相同）

图2

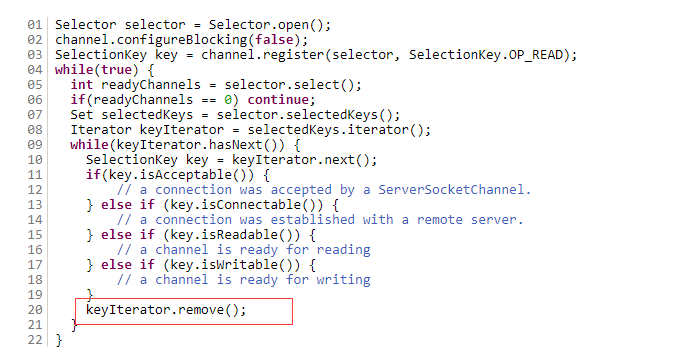


### 4.3.4 Iterator remove() 安全删除对象

当处理了某个就绪通道后，将这个就绪通道从Selector中移除，否则下次Selector.select()又会把本次的SelectionKey算进去，造成死循环。但是如果不是调用Iterator的remove()方法，会导致抛出CocurrentModifyException，但Iterator的remove()方法可以保证安全的删除对象。

（注意：如图1移除后，只是从selector的就绪通道集合里面删除了这个就绪通道，并不是删除了这个通道，因为keyIterator只是selecedKeys就绪通道集合的迭代器。当这个通道下次就绪时，又可以从就绪通道集合中获取就绪通道）

图1



### 4.3.5 close()关闭Selector

selector.close()关闭Selector本身，并使所有的SelectionKey失效，但不会关闭Channel。

## 4.4 NIO为什么是同步非阻塞IO

NIO在发起了读操作之后，也是要等待操作结果的返回结果的，不是立刻返回然后被通知的模式，所以是同步；因为Seletor获取到的Channel都是就绪的，也就是内核操作已经完成，这个阶段线程不会阻塞，所以是非阻塞。

## 4.5 线程，Seletor，Channel的关系

注意：这个线程是指服务端中处理客户端连接的线程。如图2中一个线程执行一个listen()方法，该线程会调用Selector去选择一个就绪的Channel。

所以一个线程对应一个Selector，然后一个Selector对应多个Channel。

（注意：一个IdnetAdress只需要设置一个Channel，所以这里的一个Selector的Channel的IdnetAdress都是不相同的）

图2

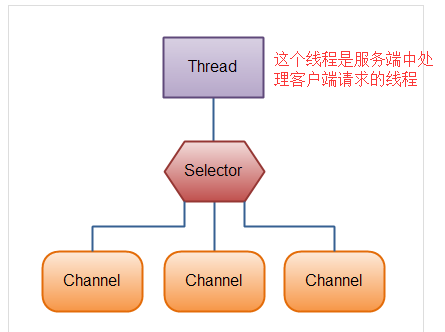
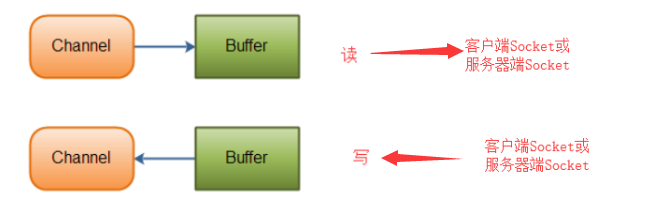


图3



当线程通过Selector选择了一个Channel后，客户端和这个Channel之间通过Buffer交互，Buffer可读可写。

图4



# 6 TCP连接

原文链接：<https://blog.csdn.net/qzcsu/article/details/72861891>

<https://blog.csdn.net/hacker00011000/article/details/52319111>

## 6.1 TCP报文首部

1.序号seq：序号，占用4个字节，因为TCP连接中传输的是字节流，字节流中的每个字节都按顺序编号。

（注意：序号和数据部分是不同的，序号是报文首部的一部分，序号4个字节，数据部分若干个字节）

2.确认号ack：确认号是期望收到对方下一个报文的序号。

3.同步SYN：用来建立连接时使用，使用时SYN=1。

（注意：SYN=1仅在三次握手的前2次使用，最后一次握手SYN=0）

（注意：SYN=1的报文不能携带数据，但需要消耗一个序号）

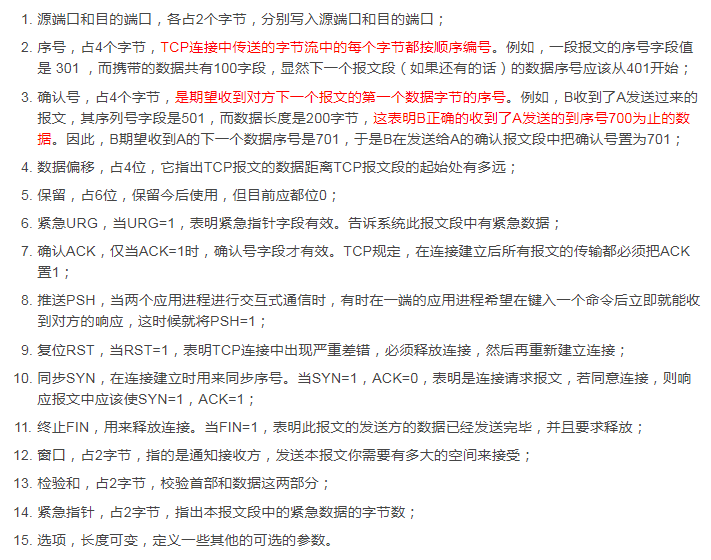
4.确认ACK：仅当ACK=1时，确认号ack才有效。

（注意：TCP规定，连接建立后的所有报文的ACK都必须置为1）

1. 释放FIN：用来释放连接时使用，使用时FIN=1。

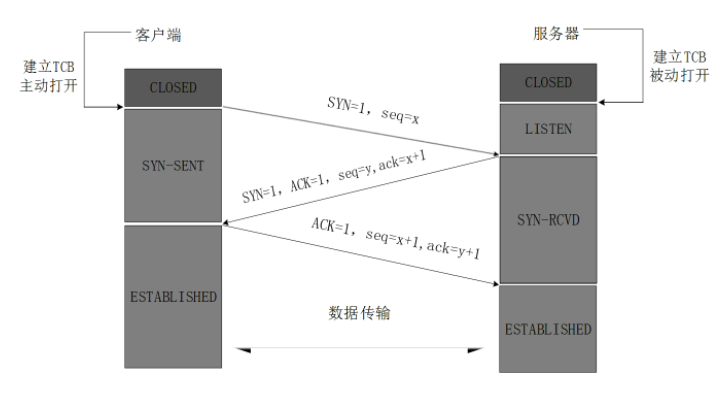
（注意：FIN=1的报文可以携带数据，但不管携不携带数据，都需要消耗一个序号）

图1



## 6.2 三次握手

图1



1. 服务器创建传输控制块TCB，并处于LISTEN状态，等待客户端的连接。

（注意：创建TCB块时还没有建立连接）

1. 客户端创建传输控制块TCB。
2. 客户端发起连接，客户端进入SYN-SENT（同步已发送）状态，这是TCP的第一次握手。
3. 服务器收到连接后，如果同意连接，则发送确认报文。然后让服务器进入SYN-RCVD（同步已收到）状态，这是TCP的第二次握手。
4. 客户端收到服务器的确认后，还要向服务器端发送确认报文。然后客户端进入ESTABLISHED状态，服务器端收到后也进入ESTABLISHED状态。

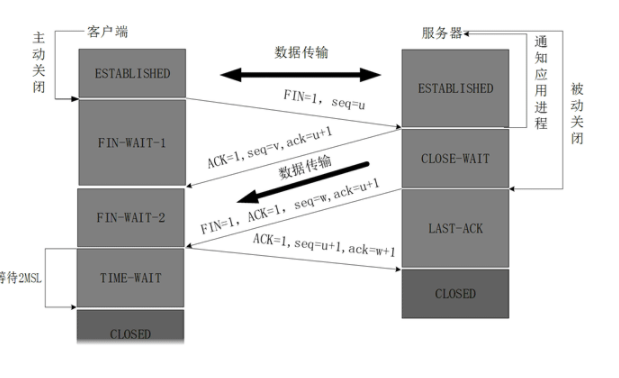
### 6.2.1 为什么需要第三次握手

如果没有第三次握手，如果只有2次握手，假设第一个连接报文在网络中滞留了很长时间，然后客户端因为超时又发了第二个连接报文。此时服务器端收到第二个连接报文，然后确认，若干时间后，收到第一个连接报文，然后也确认。此时服务器会等待第一个连接的数据，但客户端知道这个连接是失效的，不会在这个连接上发送数据，而服务器端一直在等第一个连接的数据，然后超时关闭连接。

如果有第三次握手，因为客户端知道这个连接是失效的，没有给服务器端发送确认数据，则这个失效的连接不会建立。

## 6.3 四次挥手

图1



1. 客户端发送连接释放报文，然后客户端进入FIN-WAIT-1（终止等待1）状态。
2. 服务器端收到连接释放报文后，发出确认报文，服务器进入CLOSE-WAIT（关闭等待）状态。

（注意：这时候也是半关闭状态，此时客户端已经没有要发送的数据，但服务器端还可以发送数据）

1. 客户端收到服务器的确认报文后，客户端进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文。
2. 服务器发送完最后的数据后，向客户端发送连接释放报文，然后服务器进入LAST-ACK（最后确认）状态。
3. 客户端收到服务器的连接释放报文后，发送确认报文，然后客户端进入TIME-WAIT（时间等待）状态。

（注意：此时客户端还没有进入CLOSED状态，客户端必须经过2\*MSL（最长报文段寿命）的时间后，然后客户端撤销相应的TCB后，才进入CLOSED状态）

1. 服务端收到客户端的确认报文后，撤销撤销相应的TCB后，然后进入CLOSED状态。

## 6.4 TCP和UDP的区别

## 6.5 TCP长连接和TCP短连接的区别

原文链接：<https://www.cnblogs.com/onlysun/p/4520553.html>

1.TCP长连接和短连接在连接和释放都要进行三次握手和4次挥手，但TCP长连接在没有数据通信时，会定时发送心跳包，以保持连接状态；而TCP短连接在发送完数据后，就马上关闭连接。

2.TCP长连接适合连接数不多（并发不多）的场景，因为每一条连接都一直存在很长时间；TCP短连接适合连接数多的场景。

3.TCP长连接适合操作频繁的场景（比如读写操作），因为每次建立连接会降低效率。