## Buffer

### 缓冲区基础

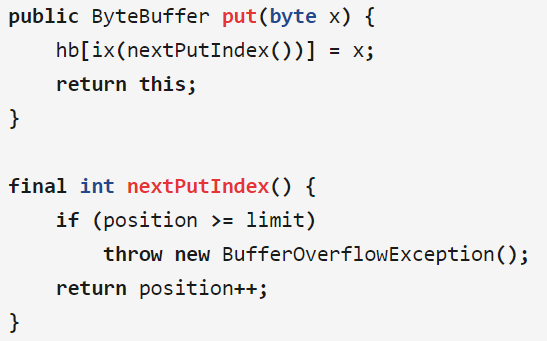
本质上，缓冲区是就是一个数组。所有的缓冲区都具有四个属性来提供关于其所包含的数组的信息。

1. 容量（Capacity） 缓冲区能够容纳的数据元素的最大数量。容量在缓冲区创建时被设定，并且永远不能被改变。
2. 上界（Limit） 缓冲区里的数据的总数，代表了当前缓冲区中一共有多少数据。
3. 位置（Position） 下一个要被读或写的元素的位置。Position会自动由相应的 get( )和 put( )函数更新。
4. 标记（Mark） 一个备忘位置。用于记录上一次读写的位置。一会儿，我会通过reset方法来说明这个属性的含义。

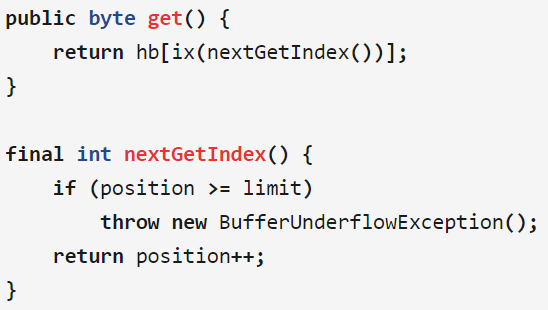
以上的语句可以创建一个大小为256字节的ByteBuffer，此时，mark = -1, pos = 0, limit = 256, capacity = 256。capacity在初始化的时候确定了，运行时就不会再变化了，而另外三个变量是随着程序的执行而不断变化的。

### 缓冲区的存取

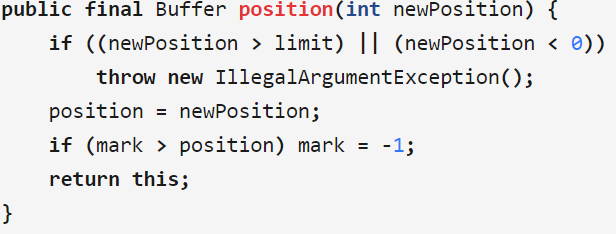
put

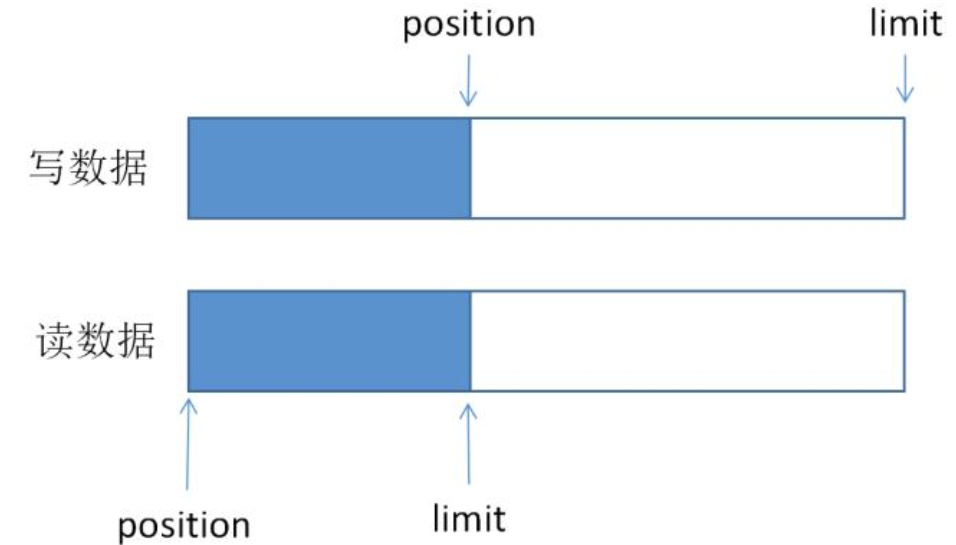


Get

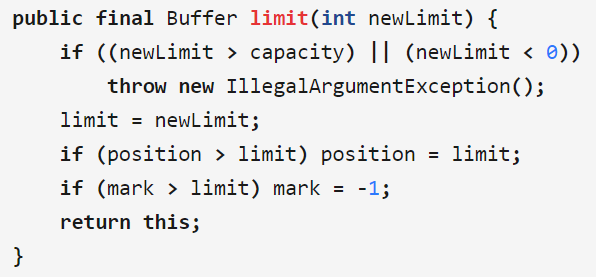


要是想在一个Buffer中放入了数据，然后想从中读取的话，就要把position调到我想读的那个位置才行。为此，ByteBuffer上定义了一个方法：

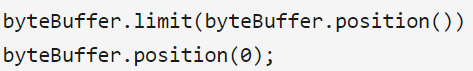




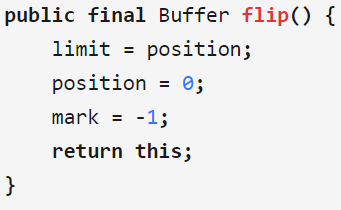
为了达到从写数据的情况变成读数据的情况，还需要修改limit，这就要用到limit方法



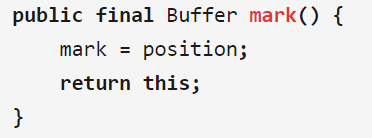
把byteBuffer从读变成写



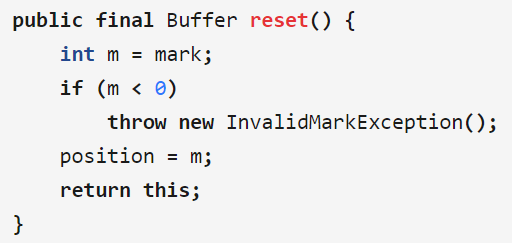
由于这个操作非常频繁，jdk就为我们封装了一个这样的方法



在理解了position的作用以后，mark就很容易理解了，它就是记住当前的位置用的：



我们在调用过mark以后，再进行缓冲区的读写操作，position就会发生变化，为了再回到当初的位置，我们可以调用reset方法恢复position的值



## channel

NIO中通过channel封装了对数据源的操作。在Java IO中，基本上可以分为文件类和Stream类两大类。Channel 也相应地分为了FileChannel 和 Socket Channel，其中 socket channel 又分为三大类，一个是用于监听端口的ServerSocketChannel，第二类是用于TCP通信的SocketChannel，第三类是用于UDP通信的DatagramChannel。

### Channel的作用





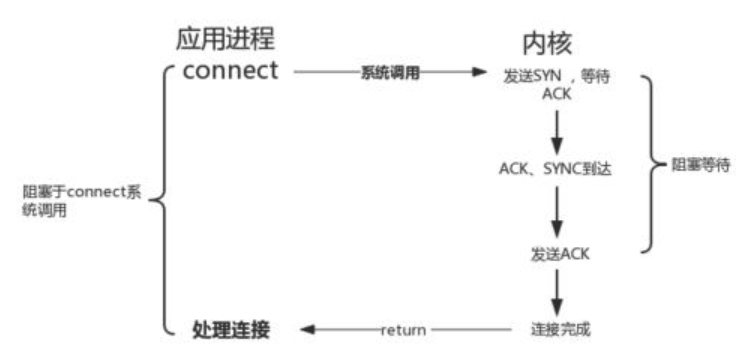
### Scatter / Gather

Channel 提供了一种被称为 Scatter/Gather 的新功能，也称为本地矢量 I/O。Scatter/Gather 是指在多个缓冲区上实现一个简单的 I/O 操作。对于一个 write 操作而言，数据是从几个缓冲区（通常就是一个缓冲区数组）按顺序抽取（称为 gather）并使用 channel 发送出去。缓冲区本身并不需要具备这种 gather 的能力。gather 过程等效于全部缓冲区的内容被连结起来，并在发送数据前存放到一个大的缓冲区中。对于 read 操作而言，从 通道读取的数据会按顺序被散布（称为 scatter）到多个缓冲区，将每个缓冲区填满直至通道中的数据或者缓冲区的最大空间被消耗完。



**IO模型**

**完全阻塞模型**



就是说，如果我客户端发起了connect请求，那么当前线程就会休眠，等待服务端响应完毕，返回消息，才会继续走下去。

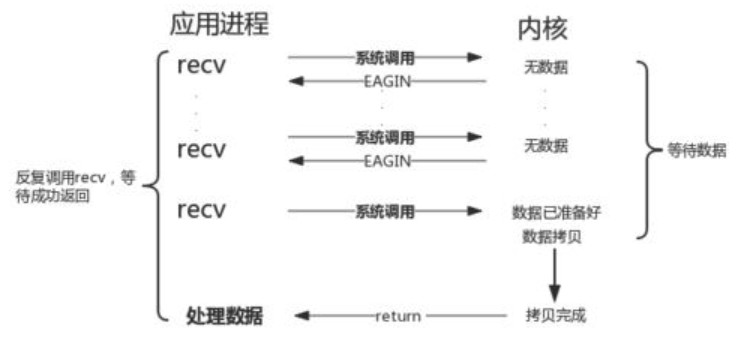
就好比，我叫个外卖，然后我就去大门口傻等着，外卖不送到，我也什么都不做，就坐在门口打盹，直到外卖小哥过来，把我叫醒，我才拿着外卖回家去吃。这样做显然效率不高，显得脑子有水。

传统的编程方式：



**非阻塞式IO**

把非阻塞的文件描述符称为非阻塞I/O。可以通过设置SOCK\_NONBLOCK标记创建非阻塞的socket fd，或者使用fcntl将fd设置为非阻塞。对非阻塞fd调用系统接口时，不需要等待事件发生而立即返回，事件没有发生，接口返回-1，此时需要通过errno的值来区分是否出错。



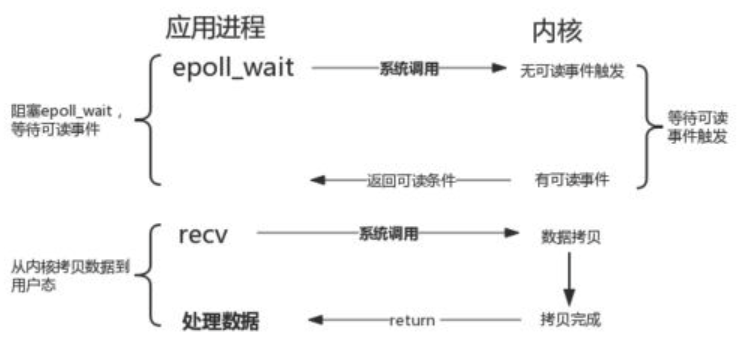
就是说，客户端程序会不停地去尝试读取数据，但是不会阻塞在那个读方法里，如果读的时候，没有读到内容，也会立即返回。这就允许我们在客户端里，读到不数据的时候可以搞点其他的事情了。

仍然以外卖举例，就相当于，我一边扫地，一边等外卖。我不再像原来一样，在门口傻等了，而是扫两下，就跑到门口看看外卖到了没有。一直这样循环，直到我取到外卖，才从这个循环中跳出来，进入吃的流程。

**IO多路复用**

NIO的核心与关键，也是高性能服务器的第一要诀。

最常用的I/O事件通知机制就是I/O复用(I/O multiplexing)。Linux 环境中使用select/poll/epoll\_wait 实现I/O复用，I/O复用接口本身是阻塞的，在应用程序中通过I/O复用接口向内核注册fd所关注的事件，当关注事件触发时，通过I/O复用接口的返回值通知到应用程序。I/O复用接口可以同时监听多个I/O事件以提高事件处理效率。



如果我们整栋楼的人，很多人叫了外卖，都有下楼来看外卖到没到的需求，于是物业就出了个招，让门卫小哥帮大家看着，整栋楼上的，不管是谁的外卖到了，先放到门卫小哥那里，然后门卫小哥再通知你下来拿自己的外卖。这样一来，我们就把本来多个人要跑去看自己的外卖到了这件事交给门卫小哥去做了。而我们解放出来，就可以继续看电视，打扫卫生，刷知乎了。由于我们可以继续 做自己的事情，外卖小哥和门卫小哥在同时也在工作，互不干扰，所以这种工作方式就被称为**异步模型**