如果是边读边写，就会很慢，也伤硬盘。缓冲区就是内存里的一块区域，把数据先存内存里，然后一次性写入，类似数据库的批量操作，这样效率比较高。  
  
调用I\O操作的时候，实际上还是一个一个的读或者写，关键就在，CPU只有一个，不论是几个核心。CPU在系统调用时，会不会还要参与主要操作？参与多次就会花更多的时间。   
  
系统调用时，若不用缓冲，CPU会酌情考虑使用 中断。此时CPU是主动地，每个周期中都要花去一部分去询问I\O设备是否读完数据，这段时间CPU不能做任何其他的事情（至少负责执行这段模块的核不能）。所以，调用一次读了一个字，通报一次，CPU腾出时间处理一次。   
  
而设置缓冲，CPU通常会使用 DMA 方式去执行 I\O 操作。CPU 将这个工作交给[DMA控制器](https://www.baidu.com/s?wd=DMA%E6%8E%A7%E5%88%B6%E5%99%A8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)来做，自己腾出时间做其他的事，当DMA完成工作时，DMA会主动告诉CPU“操作完成”。这时，CPU接管后续工作。在此，CPU 是被动的。DMA是专门 做 I＼O 与 内存 数据交换的，不仅自身效率高，也节约了[CPU时间](https://www.baidu.com/s?wd=CPU%E6%97%B6%E9%97%B4&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，CPU在DMA[开始和结束](https://www.baidu.com/s?wd=%E5%BC%80%E5%A7%8B%E5%92%8C%E7%BB%93%E6%9D%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)时做了一些设置罢了。   
所以，调用一次，不必通报CPU，等缓冲区满了，DMA 会对C PU 说 “嘿，伙计！快过来看看，把他们都搬走吧”。   
  
综上，设置缓冲，就建立了数据块，使得DMA执行更方便，CPU也有空闲，而不是呆呆地候着I\O数据读来。从微观角度来说，设置缓冲效率要高很多。尽管，不能从这个程序上看出来。 几万字的读写\就能看到差距

# 直接存取

JVM 中，字节数  
组可能不会在内存中连续存储，或者无用存储单元收集可能随时对其进行移动。在 Java 中，  
数组是对象，而数据存储在对象中的方式在不同的 JVM 实现中都各有不同。

出于这一原因，引入了直接缓冲区的概念。直接缓冲区被用于与通道和固有 I/O 例程交  
互。它们通过使用固有代码来告知操作系统直接释放或填充内存区域，对用于通道直接或原始  
存取的内存区域中的字节元素的存储尽了最大的努力。

直接字节缓冲区通常是 I/O 操作最好的选择。在设计方面，它们支持 JVM 可用的最高效  
I/O 机制。非直接字节缓冲区可以被传递给通道，但是这样可能导致性能损耗。通常非直接缓  
冲不可能成为一个本地 I/O 操作的目标。如果您向一个通道中传递一个非直接 ByteBuffer  
对象用于写入，通道可能会在每次调用中隐含地进行下面的操作：  
1.创建一个临时的直接 ByteBuffer 对象。  
2.将非直接缓冲区的内容复制到临时缓冲中。  
3.使用临时缓冲区执行低层次 I/O 操作。  
4.临时缓冲区对象离开作用域，并最终成为被回收的无用数据。  
这可能导致缓冲区在每个 I/O 上复制并产生大量对象，而这种事都是我们极力避免的。

# 文件通道总是阻塞式的

文件通道总是阻塞式的，因此不能被置于非阻塞模式。

面向流的 I/O 的非阻塞范例对于面向文件的操作并无多大意义，这是由文件 I/O 本质上的不同  
性质造成的。对于文件 I/O，最强大之处在于异步 I/O（asynchronous I/O），它允许一个进程可以  
从操作系统请求一个或多个 I/O 操作而不必等待这些操作的完成。发起请求的进程之后会收到它请  
求的 I/O 操作已完成的通知。异步 I/O 是一种高级性能，当前的很多操作系统都还不具备。以后的  
NIO 增强也会把异步 I/O 纳入考虑范围。

*FileChannel* 对象是线程安全（thread-safe）的。多个进程可以在同一个实例上并发调用方法而  
不会引起任何问题，不过并非所有的操作都是多线程的（multithreaded）。影响通道位置或者影响  
文件大小的操作都是单线程的（single-threaded）。如果有一个线程已经在执行会影响通道位置或文  
件大小的操作，那么其他尝试进行此类操作之一的线程必须等待。

//改变文件大小或position改变的操作是线程不安全的  
channel.position(position);  
channel.write(buffer);

FileChannel read 是线程安全的因为没有改变position

channel.read(buffer);

# 读取文件编码

所以对于ByteBuffer中存放UTF-16BE编码的字节时，可以直接使用asCharBuffer()直接转换成字符缓冲。

那么对于不是存放UTF-16BE编码的ByteBuffer，需要使用

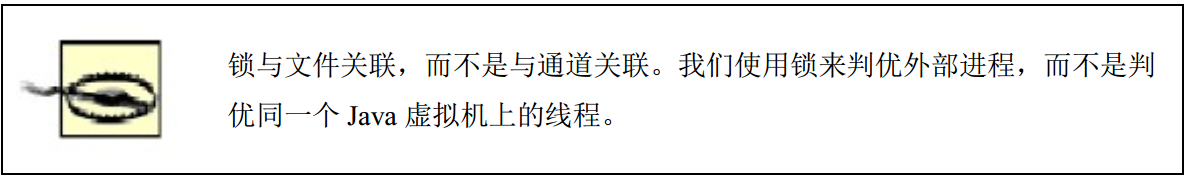
CharBuffer charBuffer=Charset.forName("字符编码").decode(byteBuffer)进行转换。

public class FilePointer {  
  
 private static final String *DEMOGRAPHIC* = "C:\\Users\\Administrator\\Desktop\\websocketBim\\1441795\\tree.json";  
  
  
 public static void main(String[] argv)  
 throws IOException {  
 RandomAccessFile randomAccessFile = new RandomAccessFile(*DEMOGRAPHIC*, "r");  
  
 randomAccessFile.seek(1000);  
  
 FileChannel fileChannel = randomAccessFile.getChannel();  
  
 // This will print "1000"  
 System.*out*.println("file pos: " + fileChannel.position());  
  
 randomAccessFile.seek(500);  
  
 // This will print "500"  
 System.*out*.println("file pos: " + fileChannel.position());  
  
 fileChannel.position(0);  
 ByteBuffer byteBuffer = ByteBuffer.*allocate*(1000);  
 fileChannel.read(byteBuffer);  
 byteBuffer.position(202);  
 //CharBuffer charBuffer = byteBuffer.asCharBuffer();  
 CharBuffer charBuffer=Charset.*forName*("UTF-8").decode(byteBuffer);  
  
 // This will print "200"  
 System.*out*.println("file pos: " + randomAccessFile.getFilePointer());  
 }  
}

# 文件锁

有关 *FileChannel* 实现的文件锁定模型的一个重要注意项是：锁的对象是文件而不是通道或线  
程，这意味着文件锁不适用于判优同一台 Java 虚拟机上的多个线程发起的访问。

如果一个线程在某个文件上获得了一个独占锁，然后第二个线程利用一个单独打开的通道来请  
求该文件的独占锁，那么第二个线程的请求会被批准。但如果这两个线程运行在不同的 Java 虚拟  
机上，那么第二个线程会阻塞，因为锁最终是由操作系统或文件系统来判优的并且几乎总是在进程  
级而非线程级上判优。锁都是与一个文件关联的，而不是与单个的文件句柄或通道关联。



文件锁是和文件相关的，和通关无关，

经过测试发现，在同一进程内开启两个线程，请求共享，或者独占锁，都会抛异常，

测试:开启两个进程，访问同一个文件，如果一个是独占锁，另一个进程会阻塞，注意文件锁可以精确到文件的具体position，size

# 文件映射

三种模式

MappedByteBuffer ro = channel.map (  
 FileChannel.MapMode.*READ\_ONLY*, 0, channel.size());  
MappedByteBuffer rw = channel.map (  
 FileChannel.MapMode.*READ\_WRITE*, 0, channel.size());  
MappedByteBuffer cow = channel.map (  
 FileChannel.MapMode.*PRIVATE*, 0, channel.size());

private是写拷贝，修改某个不会影响源文件，其他进程，线程看不到修改，写拷贝可以考到写模式修改其他位置后的变化。

READ\_WRITE，写模式修改了文件后，其他线程，进程都能看到文件的修改，并且在没有使用force()方法之前就能看到，force()是将修改同步到源文件，测试发现即使不使用force也会将修改同步。

文件映射用占用的内存空间位于 Java 虚拟机内存堆之外（并且可能不会算作 Java 虚拟机的内存占用，不过这取决于操作系统的虚拟内存模型）。

# SocketChannel

Socket 通道是线程安全的。并发访问时无需特别措施来保护发起访问的多个线程，不过任何时  
候都只有一个读操作和一个写操作在进行中。请记住， sockets 是面向流的而非包导向的。它们可  
以保证发送的字节会按照顺序到达但无法承诺维持字节分组。某个发送器可能给一个 socket 写入了  
20 个字节而接收器调用 *read( )*方法时却只收到了其中的 3 个字节。剩下的 17 个字节还是传输中。  
由于这个原因，让多个不配合的线程共享某个流 socket 的同一侧绝非一个好的设计选择。

*connect( )*和 *finishConnect( )*方法是互相同步的，并且只要其中一个操作正在进行，任何读或写  
的方法调用都会阻塞，即使是在非阻塞模式下。如果此情形下您有疑问或不能承受一个读或写操作  
在某个通道上阻塞，请用 *isConnected( )*方法测试一下连接状态。

使用方法，先open,在connect,在轮询 sc.finishConnect()，直到返回true

InetSocketAddress addr = new InetSocketAddress (host, port);  
//socketChannel虽然已经打开，但是还未连接，直接进行IO操作会抛异常  
SocketChannel sc = SocketChannel.*open*();  
  
sc.configureBlocking (false);  
  
System.*out*.println ("initiating connection");  
  
System.*out*.println(sc.connect (addr));  
//调用 finishConnect( )方法来完成连接过程，该方法任何时候都可以安全地进行调用  
while ( ! sc.finishConnect()) {  
 *doSomethingUseful*();  
}  
ByteBuffer byteBuffer = ByteBuffer.*allocate*(100);  
sc.read(byteBuffer);

# 管道 pipe

这里的管道和unix系统中的管道不一样，unix系统管道是用来连接一个进程的输出和另一个进程的输入。

*Pipe* 类实现一个管道范例，不过它所创建的管道是进程内（在 Java 虚拟机进程内部）而非进程间使用的。

# 可选择通道

只有SocketChannel，DatagramChannel继承了SelectableChannel，因此只有该类可以使用选择器。

fileChannel不能使用选择器。

# 选择器

选择器才是提供管理功能的对象，而不是可选择通道对象。选择器对象对注  
册到它之上的通道执行就绪选择，并管理选择键。

选择键的就绪状态只有在选择器对象在选择操作过程中才会修改。

维护着三种类型SelectionKey集合：

已注册的键的集合：keys()

注册后不能直接修改

已经选择的键的集合：selectedKeys()方法返回。这个集合的每个成员都是相关的通道被选择器(在前一个选择操作中)判断为已经准备好的，并且包含于键的interest集合中的操作。

已取消的键的集合(Cancelled key set)：已注册的键的集合的子集，这个集合包含了cancel()方法被调用过的键(这个键已经被无效化)，但它们还没有被注销。这个集合是选择器对象的私有成员，因而无法直接访问。

三种停止选择过程的方法

selector.wakeup(); 立即唤醒第一个还没有返回的选择器操作

第一个没有被选择的选择线程被唤醒。

//selector.close();//选择器相关通道将注销，键取消，直接抛异常

thread1.interrupt();

Selector 对象将捕捉 InterruptedException 异常并调用 *wakeup( )*方法

管理选择键

选择键一旦被选择就不会主动清理，会保留，需要手工清理。

并发性

选择器对象是线程安全的，但它们包含的键集合不是。通过 *keys( )*和 *selectKeys( )*返回的键的  
集合是 Selector 对象内部的私有的 Set 对象集合的直接引用。这些集合可能在任意时间被改变。已  
注 册 的 键 的 集 合 是 只 读 的 。

在多线程的场景中，如果您需要对任何一个键的集合进行更改，不管是直接更改还是其他操作  
带来的副作用，您都需要首先以相同的顺序，在同一对象上进行同步。锁的过程是非常重要的。如  
果竞争的线程没有以相同的顺序请求锁，就将会有死锁的潜在隐患。如果您可以确保否其他线程不  
会同时访问选择器，那么就不必要进行同步了。

不要使用两个线程同时对一个selector对象进行select()函数

# 选择键

键对象表示了一种特定的注册关系。当应该终结这种关系的时候，可以调用 SelectionKey对象的 *cancel( )*方法。

如果选择键的存续时间很长，但您附加的对象不应该存在那么长时  
间，请记得在完成后清理附件。否则，您附加的对象将不能被垃圾回  
收，您将会面临内存泄漏问题。

关于 SelectionKey 的最后一件需要注意的事情是并发性。总体上说， SelectionKey 对象是线程安全的，但知道修改 interest 集合的操作是通过 Selector 对象进行同步的是很重要的。这可能会导致 *interestOps( )*方法的调用会阻塞不确定长的一段时间

当一个通道关闭时，它相关的键也就都被取消了。。