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

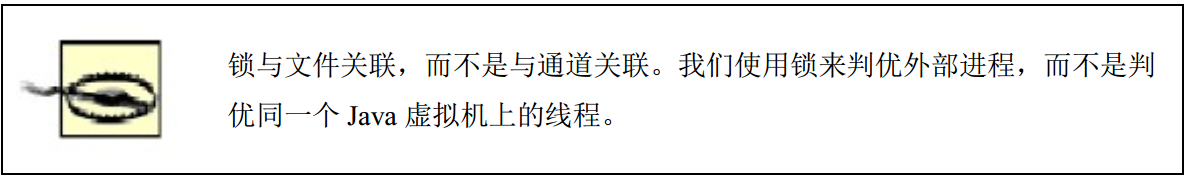
CharBuffer charBuffer=Charset.forName("字符编码").decode(byteBuffer)进行转换。

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。

# 虚拟内存

所有现代操作系统都使用虚拟内存。（使用虚拟地址取代物理内存地址）

实现两大特性：

1.一个以上的虚拟地址可指向同一个物理内存地址。

2.虚拟内存空间可大于实际可用的硬件内存。

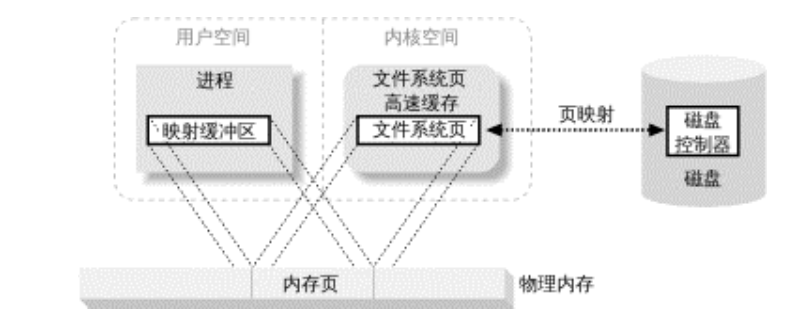
# 内存页面调度

为了支持虚拟内存的第二特性，就必须进行虚拟内存分页。虚拟内存空间的页面能够继续存在于外部磁盘存储，这样就为物理内存中的其他虚拟页面腾出了空间。

# 内存映射文件

内存映射文件没有直接通过用户进程发布read()和write()系统调用传输数据，文件数据并没有一开始就在缓冲区。传统的文件IO需要内核空间先读取文件进内核缓冲区，之后再传输到用户缓冲区，进行了不必要的拷贝操作。

内存映射文件是使用文件系统建立从用户空间直到可用文件系统页的虚拟内存映射。（实际可能还没有读进缓存）



好处：

1.用户进程把文件数据当做内存，所以无需发布read()或write()系统调用

2.当用户进程碰触到映射内存空间，页错误会自动发生，从而将文件数据从磁盘读进内存。如果用户修改了映射内存空间，相关页会自动标记为脏，然后刷新到磁盘。

3.操作系统的虚拟内存子系统会对页进行智能高速缓存，自动根据系统负载进行内存管理。

4.数据总是按页对齐的，无需进行缓冲区拷贝。

5.大型文件使用映射，无需耗费大量内存，即可进行数据拷贝。

缺点：

1.通过测试发现文件内存映射不能超过2G大小。

# 直接缓冲区

java使用内存的两种机制：

1.分配方式产生的内存开销是在JVM中

2.开销在JVM之外，系统级的内存分配。

当java程序接收到外部传来的数据是，首先是被系统内存获取，然后在由系统复制到JVM内存中供Java程序使用。

非直接字节缓冲区传递给通道分析：

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直接缓冲中。

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。

直接缓冲区不会再创建一个临时对象，不会再多一次拷贝，但是创建成本高，只用一次不建议使用。

Q：如上面所说，JNI调用的内存是不能进行GC操作的，那该如何解决了？  
A：①堆内内存与堆外内存之间数据拷贝的方式(并且在将堆内内存拷贝到堆外内存的过程JVM会保证不会进行GC操作)：比如我们要完成一个从文件中读数据到堆内内存的操作，即FileChannelImpl.read(HeapByteBuffer)。这里实际上File I/O会将数据读到堆外内存中，然后堆外内存再讲数据拷贝到堆内内存，这样我们就读到了文件中的内存。

而写操作则反之，我们会将堆内内存的数据线写到对堆外内存中，然后操作系统会将堆外内存的数据写入到文件中。  
② 直接使用堆外内存，如DirectByteBuffer：这种方式是直接在堆外分配一个内存(即，native memory)来存储数据，程序通过JNI直接将数据读/写到堆外内存中。因为数据直接写入到了堆外内存中，所以这种方式就不会再在JVM管控的堆内再分配内存来存储数据了，也就不存在堆内内存和堆外内存数据拷贝的操作了。这样在进行I/O操作时，只需要将这个堆外内存地址传给JNI的I/O的函数就好了。

## 通过配置参数的方式来回收堆外内存

同时我们可以通过-XX:MaxDirectMemorySize来指定最大的堆外内存大小，当使用达到了阈值的时候将调用System.gc()来做一次full gc，以此来回收掉没有被使用的堆外内存。

## 堆外内存那些事

#### 使用堆外内存的原因

* 对垃圾回收停顿的改善  
  因为full gc 意味着彻底回收，彻底回收时，垃圾收集器会对所有分配的堆内内存进行完整的扫描，这意味着一个重要的事实——这样一次垃圾收集对Java应用造成的影响，跟堆的大小是成正比的。过大的堆会影响Java应用的性能。如果使用堆外内存的话，堆外内存是直接受操作系统管理( 而不是虚拟机 )。这样做的结果就是能保持一个较小的堆内内存，以减少垃圾收集对应用的影响。
* 在某些场景下可以提升程序I/O操纵的性能。少去了将数据从堆内内存拷贝到堆外内存的步骤。

#### 什么情况下使用堆外内存

* 堆外内存适用于生命周期中等或较长的对象。( 如果是生命周期较短的对象，在YGC的时候就被回收了，就不存在大内存且生命周期较长的对象在FGC对应用造成的性能影响 )。
* 直接的文件拷贝操作，或者I/O操作。直接使用堆外内存就能少去内存从用户内存拷贝到系统内存的操作，因为I/O操作是系统内核内存和设备间的通信，而不是通过程序直接和外设通信的。
* 同时，还可以使用 池+堆外内存 的组合方式，来对生命周期较短，但涉及到I/O操作的对象进行堆外内存的再使用。( Netty中就使用了该方式 )

#### 堆外内存 VS 内存池

* 内存池：主要用于两类对象：①生命周期较短，且结构简单的对象，在内存池中重复利用这些对象能增加CPU缓存的命中率，从而提高性能；②加载含有大量重复对象的大片数据，此时使用内存池能减少垃圾回收的时间。
* 堆外内存：它和内存池一样，也能缩短垃圾回收时间，但是它适用的对象和内存池完全相反。内存池往往适用于生命期较短的可变对象，而生命期中等或较长的对象，正是堆外内存要解决的。

#### 堆外内存的特点

* 对于大内存有良好的伸缩性
* 对垃圾回收停顿的改善可以明显感觉到
* 在进程间可以共享，减少虚拟机间的复制

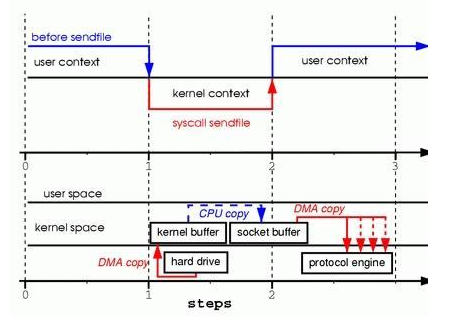
#### 堆外内存的一些问题

* 堆外内存回收问题，以及堆外内存的泄漏问题。这个在上面的源码解析已经提到了
* 堆外内存的数据结构问题：堆外内存最大的问题就是你的数据结构变得不那么直观，如果数据结构比较复杂，就要对它进行串行化（serialization），而串行化本身也会影响性能。另一个问题是由于你可以使用更大的内存，你可能开始担心虚拟内存（即硬盘）的速度对你的影响了。

# 零拷贝

概念：CPU不执行拷贝数据从一个存储区域到另一个存储区域的任务，这通常用于通过网络传输一个文件以减少CPU周期和内存带宽。

## ****通过sendfile实现的零拷贝I/O****



1. 发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard driver ——> kernel buffer)。

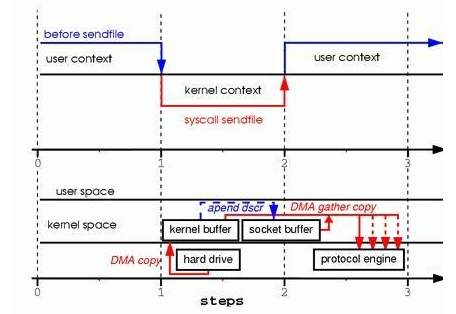
2. 然后再将数据从内核空间缓冲区拷贝到内核中与socket相关的缓冲区中(第二次拷贝: kernel buffer ——> socket buffer)。

3. sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。通过DMA引擎将内核空间socket缓冲区中的数据传递到协议引擎(第三次拷贝: socket buffer ——> protocol engine)。

通过sendfile实现的零拷贝I/O只使用了2次用户空间与内核空间的上下文切换，以及3次数据的拷贝。

你可能会说操作系统仍然需要在内核内存空间中复制数据（kernel buffer —>socket buffer）。 是的，但从操作系统的角度来看，这已经是零拷贝，因为没有数据从内核空间复制到用户空间。 内核需要复制的原因是因为通用硬件DMA访问需要连续的内存空间（因此需要缓冲区）。 但是，如果硬件支持scatter-and-gather，这是可以避免的。

## ****带有DMA收集拷贝功能的sendfile实现的I/O****



1. 发出sendfile系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard drive ——> kernel buffer)。

2. 没有数据拷贝到socket缓冲区。取而代之的是只有相应的描述符信息会被拷贝到相应的socket缓冲区当中。该描述符包含了两方面的信息：a)kernel buffer的内存地址；b)kernel buffer的偏移量。

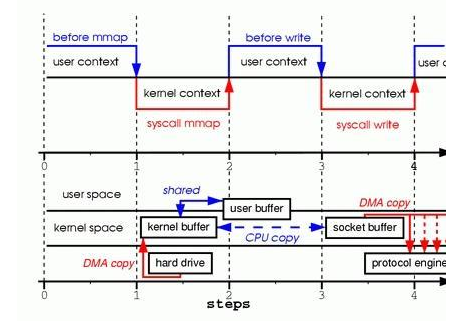
3. sendfile系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。DMA gather copy根据socket缓冲区中描述符提供的位置和偏移量信息直接将内核空间缓冲区中的数据拷贝到协议引擎上(第二次拷贝: kernel buffer ——> protocol engine)，这样就避免了最后一次CPU数据拷贝。

带有DMA收集拷贝功能的sendfile实现的I/O只使用了2次用户空间与内核空间的上下文切换，以及2次数据的拷贝，而且这2次的数据拷贝都是非CPU拷贝。这样一来我们就实现了最理想的零拷贝I/O传输了，不需要任何一次的CPU拷贝，以及最少的上下文切换。

传统I/O用户空间缓冲区中存有数据，因此应用程序能够对此数据进行修改等操作；而sendfile零拷贝消除了所有内核空间缓冲区与用户空间缓冲区之间的数据拷贝过程，因此sendfile零拷贝I/O的实现是完成在内核空间中完成的，这对于应用程序来说就无法对数据进行操作了。为了解决这个问题，Linux提供了mmap零拷贝来实现我们的需求。

## ****通过mmap实现的零拷贝I/O****

mmap(内存映射)是一个比sendfile昂贵但优于传统I/O的方法。



1. 发出mmap系统调用，导致用户空间到内核空间的上下文切换(第一次上下文切换)。通过DMA引擎将磁盘文件中的内容拷贝到内核空间缓冲区中(第一次拷贝: hard drive ——> kernel buffer)。

2. mmap系统调用返回，导致内核空间到用户空间的上下文切换(第二次上下文切换)。接着用户空间和内核空间共享这个缓冲区，而不需要将数据从内核空间拷贝到用户空间。因为用户空间和内核空间共享了这个缓冲区数据，所以用户空间就可以像在操作自己缓冲区中数据一般操作这个由内核空间共享的缓冲区数据。

3. 发出write系统调用，导致用户空间到内核空间的上下文切换(第三次上下文切换)。将数据从内核空间缓冲区拷贝到内核空间socket相关联的缓冲区(第二次拷贝: kernel buffer ——> socket buffer)。

4. write系统调用返回，导致内核空间到用户空间的上下文切换(第四次上下文切换)。通过DMA引擎将内核空间socket缓冲区中的数据传递到协议引擎(第三次拷贝: socket buffer ——> protocol engine)

通过mmap实现的零拷贝I/O进行了4次用户空间与内核空间的上下文切换，以及3次数据拷贝。其中3次数据拷贝中包括了2次DMA拷贝和1次CPU拷贝。明显，它与传统I/O相比仅仅少了1次内核空间缓冲区和用户空间缓冲区之间的CPU拷贝。这样的好处是，我们可以将整个文件或者整个文件的一部分映射到内存当中，用户直接对内存中对文件进行操作，然后是由操作系统来进行相关的页面请求并将内存的修改写入到文件当中。我们的应用程序只需要处理内存的数据，这样可以实现非常迅速的I/O操作。

说了这么多，那么Java NIO中对零拷贝的使用有哪些呢？

NIO DirectByteBuffer

Java NIO引入了用于通道的缓冲区的ByteBuffer。 ByteBuffer有三个主要的实现：

HeapByteBuffer

在调用ByteBuffer.allocate（）时使用。 它被称为堆，因为它保存在JVM的堆空间中，因此您可以获得所有优势，如GC支持和缓存优化。 但是，它不是页面对齐的，这意味着如果您需要通过JNI与本地代码交谈，JVM将不得不复制到对齐的缓冲区空间。

DirectByteBuffer

在调用ByteBuffer.allocateDirect（）时使用。 JVM将使用malloc（）在堆空间之外分配内存空间。 因为它不是由JVM管理的，所以你的内存空间是页面对齐的，不受GC影响，这使得它成为处理本地代码的完美选择。 然而，你要C程序员一样，自己管理这个内存，必须自己分配和释放内存来防止内存泄漏。

MappedByteBuffer

在调用FileChannel.map（）时使用。 与DirectByteBuffer类似，这也是JVM堆外部的情况。 它基本上作为OS mmap（）系统调用的包装函数，以便代码直接操作映射的物理内存数据。

总结

零拷贝是操作系统底层的一种实现，我们在网络编程中，利用操作系统这一特性，可以大大提高数据传输的效率。这也是目前网络编程框架中都会采用的方式。