

原

JAVA NIO之浅谈内存映射文件原理与DirectMemory

2013年03月08日 01:41:31

fcbayermunchen

阅读数：15400

12

14

JAVA类库中的NIO包相对于IO 包来说有一个新功能是内存映射文件，日常编程中并不是经常用到，但是在处理大文件时是比较理想的提高效率的手段。

在传统的文件IO操作中，我们都是调用操作系统提供的底层标准IO系统调用函数 read()、write()，此时调用此函数时进程（在JAVA中即java进程）由当前态切换到内核态，然后OS的内核代码负责将相应的文件数据读取到内核的IO缓冲区，然后再把数据从内核IO缓冲区拷贝到用户态进程的私有地址空间中去，这样一次IO操作。至于为什么要多此一举搞一个内核IO缓冲区把原本只需一次拷贝数据的事情搞成需要2次数据拷贝呢？这是因为操作系统或者计算机系统结构道，这么做是为了减少磁盘的IO操作，为了提高性能而考虑的，因为我们的程序访问一般都带有局部性，也就是局部性原理，在这里主要是指的空间局部性，即我们访问了文件的某一段数据，那么接下去很可能还会访问接下去的一段数据，由于磁盘IO操作的速度比直接访问内存慢了好几个数量级，所以OS空间局部性原理会在一次 read()系统调用过程中预读更多的文件数据缓存在内核IO缓冲区中，当继续访问的文件数据在缓冲区中时便直接拷贝数据到进程私有空间，从而避免低效率的磁盘IO操作。在JAVA中当我们采用IO包下的文件操作流，如：

```
FileInputStream in = new FileInputStream("D:\\java.txt");

in.read();
```

JAVA虚拟机内部便会调用OS底层的 read()系统调用完成操作，如上所述，在第二次调用 in.read()的时候可能就是从内核缓冲区直接返回数据了（可能还会在用户态堆做一次中转，因为这些函数都被声明为 native，即本地平台相关，所以可能在C代码中有做一次中转，如 win32中是通过 C代码从OS读取数据，然后拷贝到用户态内存）。既然如此，JAVA的IO包中为啥还要提供一个 BufferedInputStream 类来作为缓冲区呢。关键在于四个字，"系统调用"！当读取OS内核缓冲区数据时，便发起了一次系统调用操作（通过native的C函数调用），而系统调用的代价相对来说是比较高的，涉及到进程用户态和内核态的上下文切换等一系列操作，所以我们经常采用如下的包装：

```
FileInputStream in = new FileInputStream("D:\\java.txt");

BufferedInputStream buf_in = new BufferedInputStream(in);

buf_in.read();
```

这样一来，我们每一次 buf_in.read() 时候，BufferedInputStream 会根据情况自动为我们预读更多的字节数据到它自己维护的一个内部字节数组缓冲区中，从而可以减少系统调用次数，从而达到其缓冲区的目的。所以要明确的一点是 BufferedInputStream 的作用不是减少 磁盘IO操作次数（这个OS已经帮我们做了），而是通过减少系统调用次数来提高性能的。同理 BufferedOuputStream , BufferedReader/Writer 也是一样的。在 C语言的函数库中也有类似的实现，如 fread 函数就是 C语言中的缓冲IO，作用与BufferedInputStream()相同。

这里简单的引用下JDK6 中 BufferedInputStream 的源码验证下：

```
1 public
2 class BufferedInputStream extends FilterInputStream {
3
4     private static int defaultBufferSize = 8192;
5
6     /**
7      * The internal buffer array where the data is stored. When necessary,
8      * it may be replaced by another array of
9      * a different size.
10     */
11     protected volatile byte buf[];
12     /**
13      * The index one greater than the index of the last valid byte in
14      * the buffer.
15      * This value is always
16      * in the range <code>0</code> through <code>buf.length</code>;
17      * elements <code>buf[0]</code> through <code>buf[count-1</code>
18      * </code>contain buffered input data obtained
19      * from the underlying input stream.
20     */
21     protected int count;
22
23     /**
24      * The current position in the buffer. This is the index of the next
25      * character to be read from the <code>buf</code> array.
26      * <p>
27      * This value is always in the range <code>0</code>
28      * through <code>count</code>. If it is less
29      * than <code>count</code>, then <code>buf[pos]</code>
30      * is the next byte to be supplied as input;
31      * if it is equal to <code>count</code>, then
32      * the next <code>read</code> or <code>skip</code>
```

33 | * operation will require more bytes to be read from the contained input stream.

34 | * read from the contained input stream.

35 | *

36 | * @see java.io.BufferedInputStream#buf

37 | */

38 | protected int pos;

39 |

40 | /* 这里省略去 N 多代码 -----> */

41 |

42 | /**

43 | * See

44 | * the general contract of the <code>read</code>

45 | * method of <code>InputStream</code>.

46 | *

47 | * @return the next byte of data, or <code>-1</code> if the end of the

48 | * stream is reached.

49 | * @exception IOException if this input stream has been closed by

50 | * invoking its {@link #close()} method,

51 | * or an I/O error occurs.

52 | * @see java.io.FilterInputStream#in

53 | */

54 | public synchronized int read() throws IOException {

55 | if (pos >= count) {

56 | fill();

57 | if (pos >= count)

58 | return -1;

59 | }

60 | return getBufIfOpen()[pos++] & 0xff;

61 | }

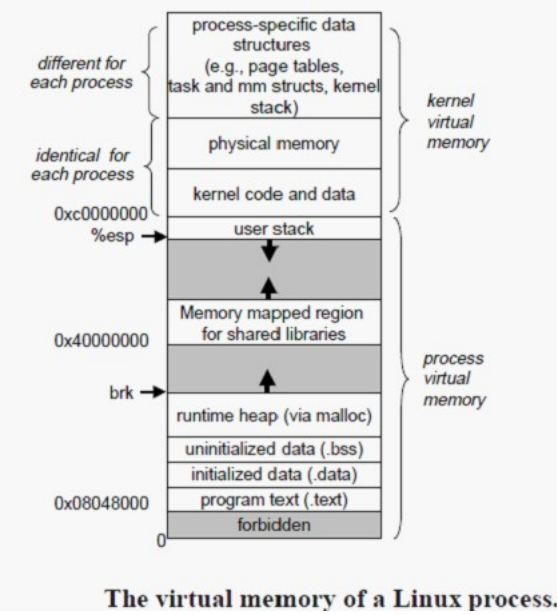
12

14

我们可以看到，BufferedInputStream 内部维护着一个 字节数组 byte[] buf 来实现缓冲区的功能，我们调用的 buf_in.read() 方法在返回数据之前有做一个如果 buf 数组的当前索引不在有效的索引范围之内，即 if 条件成立， buf 字段维护的缓冲区已经不够了，这时候会调用 内部的 fill() 方法进行填充，而fill()会的数据到 buf 数组缓冲区中去，然后再返回当前字节数据，如果 if 条件不成立便直接从 buf缓冲区数组返回数据了。其中getBufIfOpen()返回的就是 buf字段顺便说下，源码中的 buf 字段声明为 protected volatile byte buf[]; 主要是为了通过 volatile 关键字保证 buf数组在多线程并发环境中的内存可见性.

和 JAVA NIO 的内存映射无关的部分说了这么多篇幅，主要是为了做个铺垫，这样才能建立起一个知识体系，以便更好的理解内存映射文件的优点。

内存映射文件和之前说的 标准IO操作最大的不同之处就在于它虽然最终也是要从磁盘读取数据，但是它并不需要将数据读取到OS内核缓冲区，而是直接用户私有地址空间中的一部分区域与文件对象建立起映射关系，就好像直接从内存中读、写文件一样，速度当然快了。为了说清楚这个，我们以 Linux操作子，看下图：



此图为 Linux 2.X 中的进程虚拟存储器，即进程的虚拟地址空间，如果你的机子是 32 位，那么就有 $2^{32} = 4G$ 的虚拟地址空间，我们可以看到图中有一“Memory mapped region for shared libraries”，这段区域就是在内存映射文件的时候将某一段的虚拟地址和文件对象的某一部分建立起映射关系，此时并数据到内存中去，而是当进程代码第一次引用这段代码内的虚拟地址时，触发了缺页异常，这时候OS根据映射关系直接将文件的相关部分数据拷贝到进程有空间中去，当有操作第N页数据的时候重复这样的OS页面调度程序操作。注意啦，原来内存映射文件的效率比标准IO高的重要原因就是因为少了把数据拷内核缓冲区这一步（可能还少了native堆中转这一步）。

java中提供了3种内存映射模式，即：只读(readonly)、读写(read_write)、专用(private)，对于 只读模式来说，如果程序试图进行写操作，则会抛出ReaderException异常；第二种的读写模式表明了通过内存映射文件的方式写或修改文件内容的话是会立刻反映到磁盘文件中去的，别的进程如果共享了同文件，那么也会立即看到变化！而不是像标准IO那样每个进程有各自的内核缓冲区，比如JAVA代码中，没有执行 IO输出流的 flush() 或者 close() 操作，那的修改不会更新到磁盘去，除非进程运行结束；最后一种专用模式采用的是OS的“写时拷贝”原则，即在没有发生写操作的情况下，多个进程之间都是共享一块物理内存（进程各自的虚拟地址指向同一片物理地址），一旦某个进程进行写操作，那么将会把受影响的文件数据单独拷贝一份到进程的私有缓冲区中映到物理文件中去。

在JAVA NIO中可以很容易的创建一块内存映射区域，代码如下：

```
1 | File file = new File("E:\\download\\office2007pro.chs.ISO"); 2 | FileInputStream in = new FileInputStream(file);
3 | FileChannel channel = in.getChannel();
4 | MappedByteBuffer buff = channel.map(FileChannel.MapMode.READ_ONLY, 0,channel.size());
```

这里创建了一个只读模式的内存映射文件区域，接下来我就来测试下与普通NIO中的通道操作相比性能上的优势，先看下代码：

```
1 | public class IOTest {
2 |     static final int BUFFER_SIZE = 1024;
3 |
4 |     public static void main(String[] args) throws Exception {
5 |
6 |         File file = new File("F:\\aa.pdf");
7 |         FileInputStream in = new FileInputStream(file);
8 |         FileChannel channel = in.getChannel();
9 |         MappedByteBuffer buff = channel.map(FileChannel.MapMode.READ_ONLY, 0,
10 |             channel.size());
11 |
12 |         byte[] b = new byte[1024];
13 |         int len = (int) file.length();
14 |
15 |         long begin = System.currentTimeMillis();
16 |
17 |         for (int offset = 0; offset < len; offset += 1024) {
18 |
19 |             if (len - offset > BUFFER_SIZE) {
20 |                 buff.get(b);
21 |             } else {
22 |                 buff.get(new byte[len - offset]);
23 |             }
24 |         }
25 |
26 |         long end = System.currentTimeMillis();
27 |         System.out.println("time is:" + (end - begin));
28 |
29 |     }
30 | }
```

输出为 63，即通过内存映射文件的方式读取 86M多的文件只需要78毫秒，我现在改为普通NIO的通道操作看下：

```
1 | File file = new File("F:\\liq.pdf");
2 | FileInputStream in = new FileInputStream(file);
3 | FileChannel channel = in.getChannel();
4 | ByteBuffer buff = ByteBuffer.allocate(1024);
5 |
6 | long begin = System.currentTimeMillis();
7 | while (channel.read(buff) != -1) {
8 |     buff.flip();
9 |     buff.clear();
10 | }
11 | long end = System.currentTimeMillis();
12 | System.out.println("time is:" + (end - begin));
```

输出为 468毫秒，几乎是 6 倍的差距，文件越大，差距便越大。所以内存映射文件特别适合于对大文件的操作，JAVA中的限制是最大不得超过 Integer.MAX_VALUE，即2G左右，不过我们可以通过分次映射文件(channel.map)的不同部分来达到操作整个文件的目的。

按照jdk文档的官方说法，内存映射文件属于JVM中的直接缓冲区，还可以通过 ByteBuffer.allocateDirect()，即DirectMemory的方式来创建直接缓冲区。基础的 IO操作来说就是少了中间缓冲区的数据拷贝开销。同时他们属于JVM堆外内存，不受JVM堆内存大小的限制。

其中 DirectMemory 默认的大小是等同于JVM最大堆，理论上说受限于 进程的虚拟地址空间大小，比如 32位的windows上，每个进程有4G的虚拟空间除去S内核保留外，再减去 JVM堆的最大值，剩余的才是DirectMemory大小。通过 设置 JVM参数 -Xmx64M，即JVM最大堆为64M，然后执行以下程序可以证明DirectMemory不受JVM堆大小控制：

```
1 | public static void main(String[] args) {
2 |     ByteBuffer.allocateDirect(1024*1024*100); // 100MB
3 | }
```

我们设置了JVM堆 64M限制，然后在 直接内存上分配了 100MB空间，程序执行后直接报错：Exception in thread "main" java.lang.OutOfMemoryError: Direct memory。接着我设置 -Xmx200M，程序正常结束。然后我修改配置：-Xmx64M -XX:MaxDirectMemorySize=200M，程序正常结束。因此得出结论：DirectMemory的大小默认为 -Xmx 的JVM堆的最大值，但是并不受其限制，而是由JVM参数 MaxDirectMemorySize单独控制。接下来我们来证明直接内存存在JVM堆中。我们先执行以下程序，并设置 JVM参数 -XX:+PrintGC，

| | | |
|---|--|----|
| 1 | public static void main(String[] args) { | |
| 2 | for(int i=0;i<20000;i++) { | |
| 3 | ByteBuffer.allocateDirect(1024*100); //100K | |
| 4 | } | 12 |
| 5 | } | |
| | | 14 |
| | | |
| | | |
| | | |
| | | |

输出结果如下：

[GC 1371K->1328K(61312K), 0.0070033 secs]
[Full GC 1328K->1297K(61312K), 0.0329592 secs]
[GC 3029K->2481K(61312K), 0.0037401 secs]
[Full GC 2481K->2435K(61312K), 0.0102255 secs]

我们看到这里执行 GC的次数较少，但是触发了 两次 Full GC，原因在于直接内存不受 GC(新生代的Minor GC)影响，只有当执行老年代的 Full GC时候才收直接内存！而直接内存是通过存储在JVM堆中的DirectByteBuffer对象来引用的，所以当众多的DirectByteBuffer对象从新生代被送入老年代后才触发了 ful

再看直接在JVM堆上分配内存区域的情况：

```
1 public static void main(String[] args) {
2     for(int i=0;i<10000;i++) {
3         ByteBuffer.allocate(1024*100);  //100K
4     }
5 }
```

ByteBuffer.allocate 意味着直接在 JVM堆上分配内存，所以受 新生代的 Minor GC影响，输出如下：

[GC 16023K->224K(61312K), 0.0012432 secs]
[GC 16211K->192K(77376K), 0.0006917 secs]
[GC 32242K->176K(77376K), 0.0010613 secs]
[GC 32225K->224K(109504K), 0.0005539 secs]
[GC 64423K->192K(109504K), 0.0006151 secs]
[GC 64376K->192K(171392K), 0.0004968 secs]
[GC 128646K->204K(171392K), 0.0007423 secs]
[GC 128646K->204K(299968K), 0.0002067 secs]
[GC 257190K->204K(299968K), 0.0003862 secs]
[GC 257193K->204K(287680K), 0.0001718 secs]
[GC 245103K->204K(276480K), 0.0001994 secs]
[GC 233662K->204K(265344K), 0.0001828 secs]
[GC 222782K->172K(255232K), 0.0001998 secs]
[GC 212374K->172K(245120K), 0.0002217 secs]

可以看到，由于直接在 JVM堆上分配内存，所以触发了多次GC，且不会触及 Full GC，因为对象根本没机会进入老年代。

我想提个疑问，NIO中的DirectMemory和内存文件映射同属于直接缓冲区，但是前者和 -Xmx和-XX:MaxDirectMemorySize有关，而后者完全没有JVM参与和控制，这让人不禁怀疑两者的直接缓冲区是否相同，前者指的是 JAVA进程中的 native堆，即涉及底层平台如 win32的dll 部分，因为 C语言中的 malloc内存就属于 native堆，不属于 JVM堆，这也是DirectMemory能在一些场景中显著提高性能的原因，因为它避免了在 native堆和jvm堆之间数据的来回复制；是没有经过 native堆，是由 JAVA进程直接建立起 某一段虚拟地址空间和文件对象的关联映射关系，参见 Linux虚拟存储器图中的“Memory mapped region d libraries” 区域，所以内存映射文件的区域并不在JVM GC的回收范围内，因为它本身就不属于堆区，卸载这部分区域只能通过系统调用 unmap()来实现，而 JAVA API 只提供了 FileChannel.map 的形式创建内存映射区域，却没有提供对应的 unmap()，让人十分费解，导致要卸载这部分区域比较麻烦。

最后再试试通过 DirectMemory来操作前面 内存映射和基本通道操作的例子，来看看直接内存操作的话，程序的性能如何：

```
1 File file = new File("F:\\liq.pdf");
2 FileInputStream in = new FileInputStream(file);
3 FileChannel channel = in.getChannel();
4 ByteBuffer buff = ByteBuffer.allocateDirect(1024);
5
6 long begin = System.currentTimeMillis();
7 while (channel.read(buff) != -1) {
8     buff.flip();
9     buff.clear();
10 }
11 long end = System.currentTimeMillis();
12 System.out.println("time is:" + (end - begin));
```

程序输出为 312毫秒，看来比普通的NIO通道操作（468毫秒）来的快，但是比 mmap 内存映射的 63秒差距太多了，我想应该不至于吧，通过修改ByteB = ByteBuffer.allocateDirect(1024); 为 ByteBuffer buff = ByteBuffer.allocateDirect(((int)file.length())), 即一次性分配整个文件长度大小的堆外内存，最终输出为 312毫秒，由此可以得出两个结论：1.堆外内存的分配耗时比较大。 2.还是比mmap内存映射来得慢，都不要说通过mmap 内存映射数据的时候还涉及缺页异常、页面调度调用了，看来内存映射文件确实NB啊，这还只是 86M的文件，如果上 G 的大小呢？

12

最后一点为 DirectMemory的内存只有在 JVM执行 full gc 的时候才会被回收，那么如果在其上分配过大的内存空间，那么也会出现 OutofMemoryError，且堆中的很多内存处于空闲状态。

14


本来只想写点内存映射部分，但是写着写着涉及进来的知识多了点，边界不好把控啊。。。

尼玛，都是3月8号凌晨快2点了，不过想想总比以前玩 拳皇游戏 熬夜来的好吧，写完收工，赶紧睡觉去。。。

我想补充下额外的一个知识点，关于 JVM堆大小的设置是不受限于物理内存，而是受限于虚拟内存空间大小，理论上来说是进程的虚拟地址空间大小，上我们的虚拟内存空间是有限制的，一般windows上默认在C盘，大小为物理内存的2倍左右。我做了个实验：我机子是 64位的win7，那么理论上说进程虚拟内存几乎无限大，物理内存为4G，而我设置 -Xms5000M，即在启动JAVA程序的时候一次性申请到超过物理内存大小的5000M内存，程序正常启动，而当我添加 -Xmx5000M的时候就报OOM错误了，然后我修改增加 win7的虚拟内存，程序又正常启动了，说明 -Xms 受限于虚拟内存的大小。我设置-Xms5000M，即超过了4G物理内存，并在一个死循环中不断创建对象，并保证不会被GC回收。程序运行一会后整个电脑几乎死机状态，即卡住了，反映很慢很慢，推测是发生了系统颠簸的页面调度置换导致，说明 -Xms -Xmx不是局限于物理内存的大小，而是综合虚拟内存了，JVM会根据电脑虚拟内存的设置来控制。

想对作者说点什么

 **alabowa**: 老铁666 牛逼牛逼牛逼 (05-17 16:04 #11楼)

 **KevanLiu**: 不错不错，赞一个 (05-07 22:49 #10楼)

 **tomas家的小拨浪鼓**: 最近有看了些关于mmap相关的资料。个人和博主的观念有很大的出路。原来内存映射文件的效率比标准IO高的重要原因是少了内核内存到用户内存拷贝不走，而不是少了把数据拷贝到OS内核缓冲区这一步。推荐看下这篇文章：<http://xcorpion.tech/2016/09/10/it-s-all-about-buffers-zero-copy-mmap-and-java-nio/> (09-04 04:56 #9楼) [查看回复\(1\)](#)

[查看 14 条热评](#)

内存映射文件原理探索

 5744

首先说说这篇文章要解决什么问题？1.虚拟内存与内存映射文件的区别与联系. 2.内存映射文件的原理. 3.内存映射文件的效率. 4.传统IO...

内存文件映射原理和简单应用

 2104

参考博客：<http://blog.csdn.net/haiross/article/details/46875211> 参考博客：[http://blog.csdn.net/mg0832058/arti...](http://blog.csdn.net/mg0832058/article/details/46875211)

java nio 之MappedByteBuffer，高效文件/内存映射

 9057

MappedByteBuffer是java nio引入的文件内存映射方案，读写性能极高。NIO最主要的就是实现了对异步操作的支持。其中一种通过把...

内存映射文件（专门读写大文件）

 4486

引言 文件操作是应用程序最为基本的功能之一，Win32 API和MFC均提供有支持文件处理的函数和类，常用的有Win32 API的Crea...

内存映射文件原理

 8215

一直都对内存映射文件这个概念很模糊，不知道它和虚拟内存有什么区别，而且映射这个词也很让人迷茫，今天终于搞清楚了。。。下...

从DirectMemory谈谈Java NIO

 4197

本机内存DirectMemory，属于C Heap，可以通过参数-XX:MaxDirectMemorySize指定。如果不指定，该参数的默认值为Xmx的值减去...

内存映射机制

 996

现代意义上的操作系统都处于32位保护模式下。每个进程一般都能寻址4G的内存空间。但是我们的物理内存常常没有这么大，进程怎...

JVM的DirectMemory设置

 5395

几台服务器的JVM占用内存总是持续增长，大大超过-Xmx设定的值，服务器物理内存几乎被耗尽。使用jmap查看JVM的内存使用，发...

MemMap内存映射

一、内存映射文件是由一个文件到一块内存的映射。Win32提供了允许应用程序把文件映射到一个进程的函数 (CreateFileMapping)。

运行时数据区域——直接内存（Direct Memory）。

博文中的内容来源《深入理解Java虚拟机_JVM高级特性与最佳实践》这一本书，感激不尽。...

相关热词

javall 与java java的~ java java和--



2854

12



14

个人资料



fcbayernmunchen

关注

原创

10

粉丝

83

喜欢

5

评论

157

等级：

博客



访问：6万+

积分：801

排名：7万+

最新文章

Docker问答录系列——Docker引擎相关问题(五)

linux系统的用户磁盘配额quota

设计产品，除了用户体验，别轻视了运营

星巴克内容营销案例体会

轻松扩展LinkedHashMap类实现LRU算法

个人分类

docker

1篇

归档

2017年2月

1篇

2016年10月

1篇

2016年7月

1篇

2016年6月

1篇

2013年7月

1篇

展开

热门文章

面向对象分析过程案例实战

阅读量：17262

JAVA NIO之浅谈内存映射文件原理与Direct Memory

阅读量：15394

struts2源码分析-IOC容器的实现机制（上篇）

阅读量：10173

本人对于“用例”的一些理解和总结

阅读量：6793

浅谈Struts2拦截器Interceptor的设计原理

阅读量：3858

最新评论

JAVA NIO之浅谈内存映射文件...

alabowa：老铁666 牛逼牛逼牛逼

JAVA NIO之浅谈内存映射文件...

u011207553：不错不错，赞一个

JAVA NIO之浅谈内存映射文件...

u013096088：[reply]tonyllz[/reply] 我发现你了，小伙伴

JAVA NIO之浅谈内存映射文件...

tonyllz：最近有看了些关于mmap相关的资料。个人和博主的观念有很大的出路。原来内存映射文件的效率比标准IO...

JAVA NIO之浅谈内存映射文件...

tonyllz: 呀! 惊喜的发现原来同是拜仁球迷~ 哈哈
哈 文章很好, 收获很大~ 嗯, 补充下。FileChanne...

联系我们



请扫描二维码联系客服

✉ webmaster@csdn.net

☎ 400-660-0108

💬 QQ客服

💬 客服论坛

[关于](#) [招聘](#) [广告服务](#) [网站地图](#)

©2018 CSDN版权所有 京ICP证09002463号

🐶 百度提供搜索支持



CSDN APP

经营性网站备案信息

网络110报警服务

中国互联网举报中心

北京互联网违法和不良信息举报中心

| |
|----|
| 12 |
| 14 |
| |
| |
| |
| |