

深入浅出MappedByteBuffer



占小狼 (/u/90ab66c248e6) [+ 关注](#)

2016.07.26 17:03* 字数 2633 阅读 10675 评论 5 喜欢 40 赞赏 2

(/u/90ab66c248e6)

简书 占小狼 (http://www.jianshu.com/users/90ab66c248e6/latest_articles)

转载请注明原创出处，谢谢！

前言

java io操作中通常采用BufferedReader，BufferedInputStream等带缓冲的IO类处理大文件，不过java.nio中引入了一种基于MappedByteBuffer操作大文件的方式，其读写性能极高，本文会介绍其性能如此高的内部实现原理。

内存管理

在深入MappedByteBuffer之前，先看看计算机内存管理的几个术语：

- **MMC**：CPU的内存管理单元。
- **物理内存**：即内存条的内存空间。
- **虚拟内存**：计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。
- **页面文件**：操作系统反映构建并使用虚拟内存的硬盘空间大小而创建的文件，在windows下，即pagefile.sys文件，其存在意味着物理内存被占满后，将暂时不用的数据移动到硬盘上。
- **缺页中断**：当程序试图访问已映射在虚拟地址空间中但未被加载至物理内存的一个分页时，由MMC发出的中断。如果操作系统判断此次访问是有效的，则尝试将相关的页从虚拟内存文件中载入物理内存。

为什么会有虚拟内存和物理内存的区别？

如果正在运行的一个进程，它所需的内存是有可能大于内存条容量之和的，如内存条是256M，程序却要创建一个2G的数据区，那么所有数据不可能都加载到内存（物理内存），必然有数据要放到其他介质中（比如硬盘），待进程需要访问那部分数据时，再调度进入物理内存。

什么是虚拟内存地址和物理内存地址？

假设你的计算机是32位，那么它的地址总线是32位的，也就是它可以寻址0~0xFFFFFFFF（4G）的地址空间，但如果你的计算机只有256M的物理内存0x~0x0FFFFFFF（256M），同时你的进程产生了一个不在这256M地址空间中的地址，那么计算机该如何处理呢？回答这个问题前，先说明计算机的内存分页机制。

计算机会对虚拟内存地址空间（32位为4G）进行分页产生页（page），对物理内存地址空间（假设256M）进行分页产生页帧（page frame），页和页帧的大小一样，所以虚拟内存页的个数势必要大于物理内存页帧的个数。在计算机上有一个页表（page table），就是映射虚拟内存页到物理内存页的，更确切的说是页号到页帧号的映射，而且是一对

一的映射。

问题来了，虚拟内存页的个数 > 物理内存页帧的个数，岂不是有些虚拟内存页的地址永远没有对应的物理内存地址空间？不是的，操作系统是这样处理的。操作系统有个页面失效（page fault）功能。操作系统找到一个最少使用的页帧，使之失效，并把它写入磁盘，随后把需要访问的页放到页帧中，并修改页表中的映射，保证了所有的页都会被调度。

现在来看看什么是虚拟内存地址和物理内存地址：

- 虚拟内存地址：由页号（与页表中的页号关联）和偏移量（页的小大，即这个页能存多少数据）组成。

举个例子，有一个虚拟地址它的页号是4，偏移量是20，那么他的寻址过程是这样的：首先到页表中找到页号4对应的页帧号（比如为8），如果页不在内存中，则用失效机制调入页，接着把页帧号和偏移量传给MMC组成一个物理上真正存在的地址，最后就是访问物理内存的数据了。

MappedByteBuffer是什么

从继承结构上看，MappedByteBuffer继承自ByteBuffer，内部维护了一个逻辑地址address。

示例

通过MappedByteBuffer读取文件

```
public class MappedByteBufferTest {
    public static void main(String[] args) {
        File file = new File("D://data.txt");
        long len = file.length();
        byte[] ds = new byte[(int) len];

        try {
            MappedByteBuffer mappedByteBuffer = new RandomAccessFile(file, "r")
                .getChannel()
                .map(FileChannel.MapMode.READ_ONLY, 0, len);
            for (int offset = 0; offset < len; offset++) {
                byte b = mappedByteBuffer.get();
                ds[offset] = b;
            }

            Scanner scan = new Scanner(new ByteArrayInputStream(ds)).useDelimiter("
");
            while (scan.hasNext()) {
                System.out.print(scan.next() + " ");
            }
        } catch (IOException e) {}
    }
}
```

map过程

FileChannel提供了map方法把文件映射到虚拟内存，通常情况可以映射整个文件，如果文件比较大，可以进行分段映射。

- FileChannel中的几个变量：

- **MapMode mode**：内存映像文件访问的方式，共三种：

1. MapMode.READ_ONLY：只读，试图修改得到的缓冲区将导致抛出异常。



2. `MapMode.READ_WRITE`: 读/写, 对得到的缓冲区的更改最终将写入文件; 但该更改对映射到同一文件的其他程序不一定是可见的。
3. `MapMode.PRIVATE`: 私用, 可读可写, 但是修改的内容不会写入文件, 只是buffer自身的改变, 这种能力称之为"copy on write"。

- **position**: 文件映射时的起始位置。
- **allocationGranularity**: Memory allocation size for mapping buffers, 通过native函数initIDs初始化。

接下去通过分析源码, 了解一下map过程的内部实现。

1. 通过`RandomAccessFile`获取`FileChannel`。

```
public final FileChannel getChannel() {
    synchronized (this) {
        if (channel == null) {
            channel = FileChannelImpl.open(fd, path, true, rw, this);
        }
        return channel;
    }
}
```

上述实现可以看出, 由于`synchronized`, 只有一个线程能够初始化`FileChannel`。

2. 通过`FileChannel.map`方法, 把文件映射到虚拟内存, 并返回逻辑地址`address`, 实现如下:

```
**只保留了核心代码**
public MappedByteBuffer map(MapMode mode, long position, long size) throws IOException {
    int pagePosition = (int)(position % allocationGranularity);
    long mapPosition = position - pagePosition;
    long mapSize = size + pagePosition;
    try {
        addr = map0(mode, mapPosition, mapSize);
    } catch (OutOfMemoryError x) {
        System.gc();
        try {
            Thread.sleep(100);
        } catch (InterruptedException y) {
            Thread.currentThread().interrupt();
        }
        try {
            addr = map0(mode, mapPosition, mapSize);
        } catch (OutOfMemoryError y) {
            // After a second OOME, fail
            throw new IOException("Map failed", y);
        }
    }
    int isize = (int)size;
    Unmapper um = new Unmapper(addr, mapSize, isize, mfd);
    if ((!writable) || (mode == MAP_RO)) {
        return Util.newMappedByteBufferR(isize,
                                         addr + pagePosition,
                                         mfd,
                                         um);
    } else {
        return Util.newMappedByteBuffer(isize,
                                         addr + pagePosition,
                                         mfd,
                                         um);
    }
}
```

上述代码可以看出, 最终map通过native函数map0完成文件的映射工作。

1. 如果第一次文件映射导致OOM, 则手动触发垃圾回收, 休眠100ms后再次尝试映射, 如果失败, 则抛出异常。

2. 通过newMappedByteBuffer方法初始化MappedByteBuffer实例，不过其最终返回的是DirectByteBuffer的实例，实现如下：

```
static MappedByteBuffer newMappedByteBuffer(int size, long addr, FileDescriptor fd,
MappedByteBuffer dbb;
if (directByteBufferConstructor == null)
    initDBBConstructor();
dbb = (MappedByteBuffer)directByteBufferConstructor.newInstance(
    new Object[] { new Integer(size),
        new Long(addr),
        fd,
        unmapper }

return dbb;
}
// 访问权限
private static void initDBBConstructor() {
AccessController.doPrivileged(new PrivilegedAction<Void>() {
    public Void run() {
        Class<?> cl = Class.forName("java.nio.DirectByteBuffer");
        Constructor<?> ctor = cl.getDeclaredConstructor(
            new Class<?>[] { int.class,
                long.class,
                FileDescriptor.class,
                Runnable.class });
        ctor.setAccessible(true);
        directByteBufferConstructor = ctor;
    }
});
}
```

由于FileChannelImpl和DirectByteBuffer不在同一个包中，所以有权限访问问题，通过AccessController类获取DirectByteBuffer的构造器进行实例化。

DirectByteBuffer是MappedByteBuffer的一个子类，其实现了对内存的直接操作。

get过程

MappedByteBuffer的get方法最终通过DirectByteBuffer.get方法实现的。

```
public byte get() {
    return ((unsafe.getByte(ix(nextGetIndex()))));
}
public byte get(int i) {
    return ((unsafe.getByte(ix(checkIndex(i)))));
}
private long ix(int i) {
    return address + (i << 0);
}
```

map0()函数返回一个地址address，这样就无需调用read或write方法对文件进行读写，通过address就能够操作文件。底层采用unsafe.getByte方法，通过（address + 偏移量）获取指定内存的数据。

1. 第一次访问address所指向的内存区域，导致缺页中断，中断响应函数会在交换区中查找相对应的页面，如果找不到（也就是该文件从来没有被读入内存的情况），则从硬盘上将文件指定页读取到物理内存中（非jvm堆内存）。
2. 如果在拷贝数据时，发现物理内存不够用，则会通过虚拟内存机制（swap）将暂时不用的物理页面交换到硬盘的虚拟内存中。

性能分析

从代码层面上看，从硬盘上将文件读入内存，都要经过文件系统进行数据拷贝，并且数据拷贝操作是由文件系统和硬件驱动实现的，理论上来说，拷贝数据的效率是一样的。但是通过内存映射的方法访问硬盘上的文件，效率要比read和write系统调用高，这是为



什么？

- read()是系统调用，首先将文件从硬盘拷贝到内核空间的一个缓冲区，再将这些数据拷贝到用户空间，实际上进行了两次数据拷贝；
- map()也是系统调用，但没有进行数据拷贝，当缺页中断发生时，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝。

所以，采用内存映射的读写效率要比传统的read/write性能高。

总结

1. MappedByteBuffer使用虚拟内存，因此分配(map)的内存大小不受JVM的-Xmx参数限制，但是也是有大小限制的。
2. 如果当文件超出1.5G限制时，可以通过position参数重新map文件后面的内容。
3. MappedByteBuffer在处理大文件时的确性能很高，但也存在一些问题，如内存占用、文件关闭不确定，被其打开的文件只有在垃圾回收的才会被关闭，而且这个时间点是未知的。

javadoc中也提到：**A mapped byte buffer and the file mapping that it represents remain* valid until the buffer itself is garbage-collected.**

END。

我是占小狼。

在魔都艰苦奋斗，白天是上班族，晚上是知识服务工作者。

读完我的文章有收获，记得关注和点赞哦，如果非要打赏，我也是不会拒绝的啦！

📖 java进阶干货 (/mb/4893857) 举报文章 © 著作权归作者所有



占小狼 (/u/90ab66c248e6) ♂


写了 140213 字，被 9808 人关注，获得了 5856 个喜欢 (/u/90ab66c248e6)

+ 关注

如果读完觉得有收获的话，欢迎点赞加关注 微信公众号 占小狼的博客 <http://upload-images.jianshu.io/upload-images/4965268/w>

如果觉得我的文章对您有用，请随意赞赏。您的支持将鼓励我继续创作！

赞赏支持



/u/28a408bd3c83)

 喜欢 (/sign_in?utm_source=desktop&utm_medium=not-signed-in-like-button) | 40

 更多分享

(<http://cwb.assets.jianshu.io/notes/images/4965268/w>



登录 (/sign_in?utm_source=desktop&utm_medium=not-signed-in-comment-form)

评论



5条评论 只看作者 按喜欢排序 按时间正序 按时间倒序



周星爷 (/u/dd2f56d130c2)

4楼 · 2017.09.22 19:34

(/u/dd2f56d130c2)
不错不错，收藏了。

推荐下，分库分表中间件 Sharding-JDBC 源码解析 17 篇：<http://t.cn/R0UfGFT>
(<http://t.cn/R0UfGFT>)

怔

👍 6人赞 💬 回复



古岩_2093 (/u/f4d7058fdc01)

2楼 · 2017.06.24 02:02

(/u/f4d7058fdc01)
是mmu吧？

👍 赞 💬 回复



不朽传说_020d (/u/99957c63cad0)

3楼 · 2017.08.11 01:28

(/u/99957c63cad0)
😄 请问如果是同一个文件 A，c++程序用mmap以读写模式打开了它，然后Java用MappedByteBuffer以只读模式打开它，这俩程序在虚拟内存的映射文件是同一个吗？

👍 赞 💬 回复



眼眼眼眼眼_749b (/u/9b8dd6db8b4e)

5楼 · 2017.10.15 23:44

(/u/9b8dd6db8b4e)
map()也是系统调用，但没有进行数据拷贝，当缺页中断发生时，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝。这一句，硬件设备应该没权限将数据拷贝到用户空间吧。这里面实际上是拷贝到内核空间，用虚拟内存读取到的吧？其实数据并不在用户空间吧？

👍 赞 💬 回复



zdjray (/u/61251e8389d2)

6楼 · 2017.10.29 06:24

(/u/61251e8389d2)
写的不错,之前的一段基础也带入的很好

👍 赞 💬 回复

被以下专题收入，发现更多相似内容



首页投稿 (/c/bDHhpK?utm_source=desktop&utm_medium=notes-included-collection)



Java学习笔记 (/c/04cb7410c597?utm_source=desktop&utm_medium=notes-included-collection)




程序员 (/c/NEt52a?utm_source=desktop&utm_medium=notes-included-collection)



技术干货 (/c/38d96caffb2f?utm_source=desktop&utm_medium=notes-)



- included-collection)
-  java进阶干货 (/c/addfce4ca518?utm_source=desktop&utm_medium=notes-included-collection)
-  Java技术文章 (/c/ef7836bf3e22?utm_source=desktop&utm_medium=notes-included-collection)
-  Java core (/c/a05c05518ea6?utm_source=desktop&utm_medium=notes-included-collection)

展开更多 ▾

^

🔗