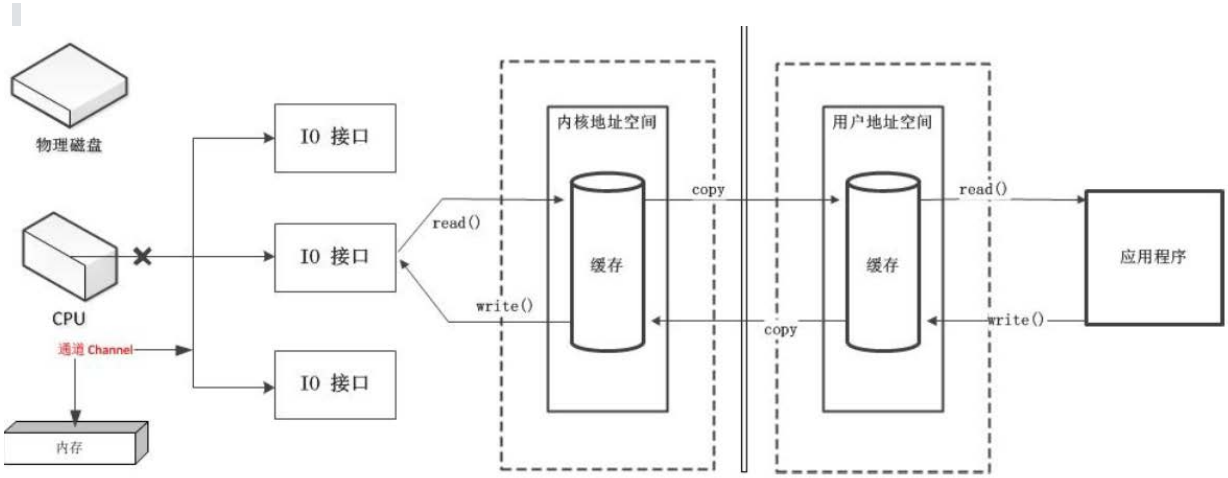
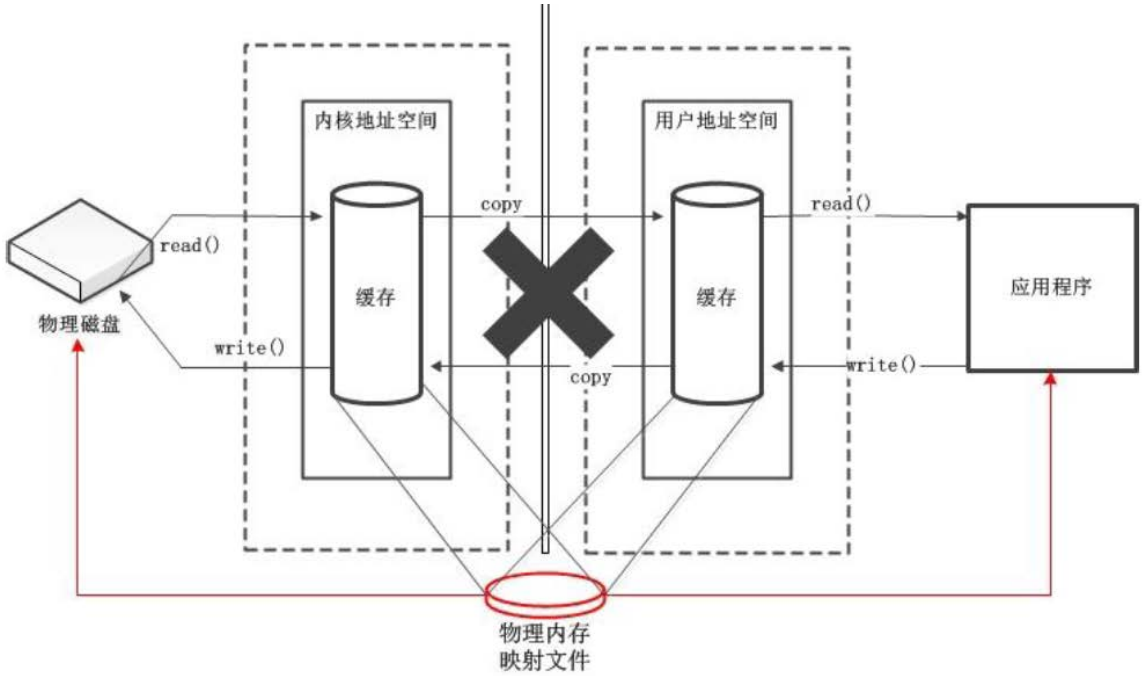
- mmap基础概念
- mmap内存映射原理
- mmap和常规文件操作的区别
- mmap优点总结
- mmap使用细节
  - 堆外内存
  - 在讲解DirectByteBuffer之前，需要先简单了解两个知识点
    - java引用类型，因为DirectByteBuffer是通过虚引用(Phantom Reference)来实现堆外内存的释放的。
    - 关于linux的内核态和用户态
  - DirectByteBuffer ———— 直接缓冲
  - DirectByteBuffer堆外内存的创建和回收的源码解读
    - 堆外内存分配
    - Bits.reserveMemory(size, cap) 方法
    - 堆外内存回收
    - 通过配置参数的方式来回收堆外内存
  - 堆外内存那些事
    - 使用堆外内存的原因

- 什么情况下使用堆外内存
- 堆外内存 VS 内存池
- 堆外内存的特点
- 堆外内存的一些问题

- 参考文章

阅读目录





mmap基础概念

对文件的操作直接反映到内存上对内存的操作也直接反映到文件上

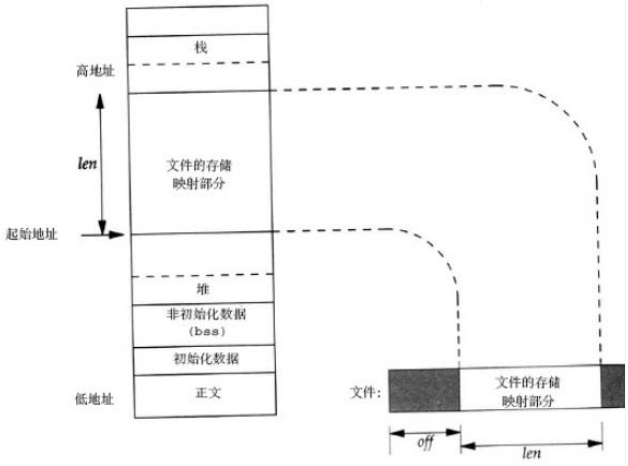
mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。如下图所示：

一般情况：应用程序需要读取文件数据发起read调用，后内核对文件发起read系统调用，数据读取到内核空间，之后从内核空间拷贝到用用户进程空间。因此用户空间内存和物理磁盘数据交换需要通过内核进行数据拷贝。有了mmap之后，进程空间内存可以“通过映射”直接读取物理磁盘中数据，不经过内核。内核也可以写数据到物理磁盘，导致用户空间内存发生改变。内核空间对物理磁盘的修改直接反应到了用户空间的内存中。

由上图可以看出，进程的虚拟地址空间，由多个虚拟内存区域构成。虚拟内存区域是进程的虚拟地址空间中的一个同质区间，即具有同样特性的连续地址范围。上图中所示的text数据段（代码段）、初始数据段、BSS数据段、堆、栈和内存映射，都是一个独立的虚拟内存区域。而为内存映射服务的地址空间处在堆栈之间的空余部分。

linux内核使用vm_area_struct结构来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个vm_area_struct结构来分别表示不同类型的虚拟内存区域。各个vm_area_struct结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：vm_area_struct结构中包含区域起始和终止地址以及其他相关信息，同时也包含一个vm_ops指针，其内部可引出所有针对这个区域可以使用的系统调用函数。这样，进程对某一虚拟内存区域的任何操作需要用要的信息，都可以从vm_area_struct中获得。mmap函数就是要创建一个新的vm_area_struct结构，并将其与文件的物理磁盘地址相连。具体步骤请看下一节。

mmap内存映射原理



mmap内存映射的实现过程，总的来说可以分为三个阶段：

（一）进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域

1、进程在用户空间调用库函数mmap，原型：
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

2、在当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续的虚拟地址

3、为此虚拟区分配一个vm_area_struct结构，接着对这个结构的各个域进行了初始化

4、将新建的虚拟区结构（vm_area_struct）插入进程的虚拟地址区域链表或树中

（二）调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系

5、为映射分配了新的虚拟地址区域后，通过待映射的文件指针，在文件描述符表中找到对应的文件描述符，通过文件描述符，链接到内核“已打开文件集”中该文件的文件结构体（struct file），每个文件结构体维护着和这个已打开文件相关各项信息。

6、通过该文件的文件结构体，链接到file_operations模块，调用内核函数mmap，其原型为：int mmap(struct file *filp, struct vm_area_struct *vma)，不同于用户空间库函数。

7、内核mmap函数通过虚拟文件系统inode模块定位到文件磁盘物理地址。

8、通过remap_pfn_range函数建立页表，即实现了文件地址和虚拟地址区域的映射关系。此时，这片虚拟地址并没有任何数据关联到主存中。

（三）进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝

注：前两个阶段仅在于创建虚拟区间并完成地址映射，但是并没有将任何文件数据的拷贝至主存。真正的文件读取是当进程发起读或写操作时。

9、进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上。因为目前只建立了地址映射，真正的硬盘数据还没有拷贝到内存中，因此引发缺页异常。

10、缺页异常进行一系列判断，确定无非法操作后，内核发起请求调页过程。

11、调页过程先在交换缓存空间（swap cache）中寻找需要访问的内存页，如果没有则调用nopage函数把所缺的页从磁盘装入到主存中。

12、之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。

注：修改过的脏页面并不会立即更新回文件中，而是有一段时间的延迟，可以调用msync()来强制同步, 这样所写的内容就能立即保存到文件里了。

回到顶部

mmap和常规文件操作的区别

对linux文件系统不了解的朋友，请参阅我之前写的博文《从内核文件系统看文件读写过程》，我们首先简单的回顾一下常规文件系统操作（调用read/fread等类函数）中，函数的调用过程：

1、进程发起读文件请求。

2、内核通过查找进程文件符表，定位到内核已打开文件集上的文件信息，从而找到此文件的inode。

3、inode在address_space上查找要请求的文件页是否已经缓存在页缓存中。如果存在，则直接返回这片文件页的内容。

4、如果不存在，则通过inode定位到文件磁盘地址，将数据从磁盘复制到页缓存。之后再次发起读页面过程，进而将页缓存中的数据发给用户进程。

页缓冲位于操作系统内核空间

总结来说，常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。写操作也是一样，待写入的buffer在内核空间不能直接访问，必须要先拷贝至内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

而使用mmap操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。

总而言之，常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。 FileChannel的读取效率比mmap效率低的原因

回到顶部

mmap优点总结

由上文讨论可知，mmap优点共有一下几点：

适合数据拷贝

1、对文件的读取操作跨过了页缓存，减少了数据的拷贝次数，用内存读写取代I/O读写，提高了文件读取效率。 内核空间对物理文件的write写操作，立刻映射到用户空间的物理内存
用户空间的进程对内存的写操作，立刻映射到物理文件。

2、实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。

多个进程通过mmap映射到同一个文件，达到共享内存

3、提供进程间共享内存及相互通信的方式。不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

适合大量数据拷贝

4、可用于实现高效的大规模数据传输。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。

mmap使用细节

1、使用mmap需要注意的一个关键点是，mmap映射区域大小必须是物理页大小(page_size)的整倍数（32位系统中通常是4k字节）。原因是，内存的最小粒度是页，而进程虚拟地址空间和内存的映射也是以页为单位。为了匹配内存的操作，mmap从磁盘到虚拟地址空间的映射也必须是页。

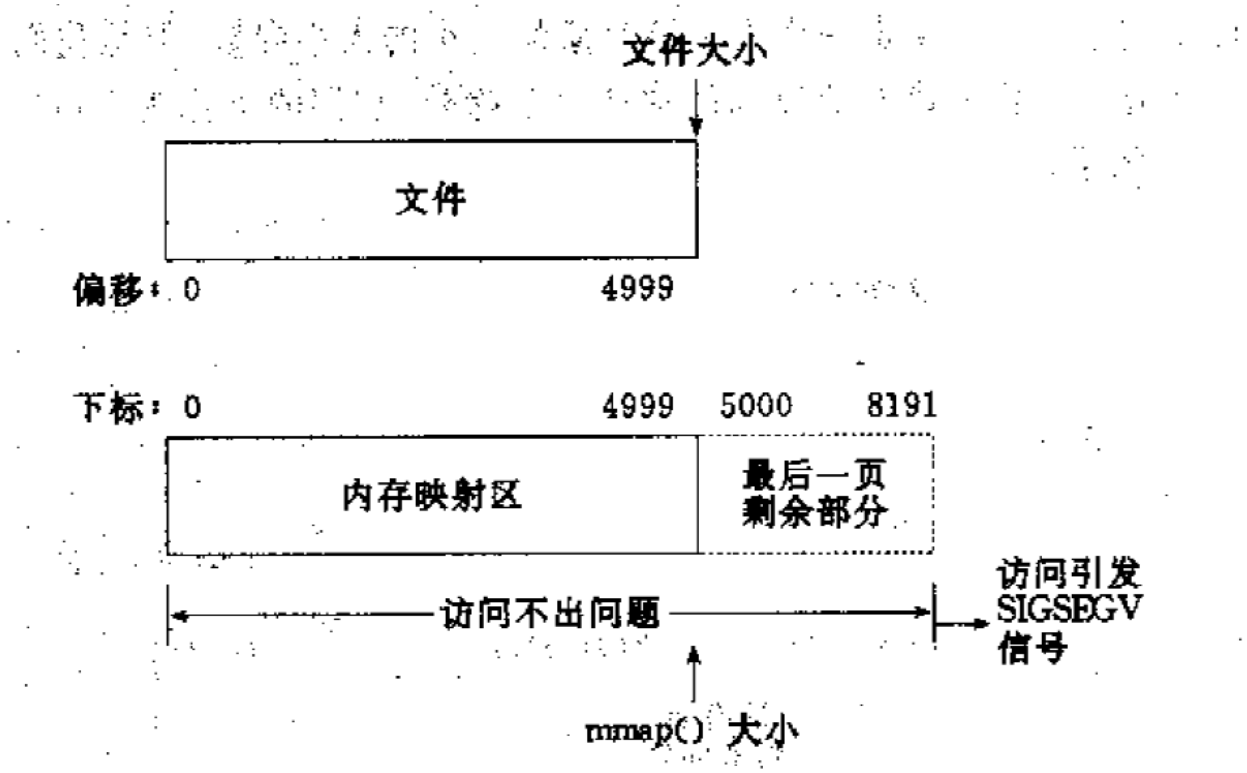
2、内核可以跟踪被内存映射的底层对象（文件）的大小，进程可以合法的访问在当前文件大小以内又在内存映射区以内的那些字节。也就是说，如果文件的大小一直在扩张，只要在映射区域范围内的数据，进程都可以合法得到，这和映射建立时文件的大小无关。具体情形参见“情形三”。

3、映射建立之后，即使文件关闭，映射依然存在。因为映射的是磁盘的地址，不是文件本身，和文件句柄无关。同时可用于进程间通信的有效地址空间不完全受限于被映射文件的大小，因为是按页映射。

在上面的知识前提下，我们下面看看如果大小不是页的整倍数的具体情况：

情形一：一个文件的大小是5000字节，mmap函数从一个文件的起始位置开始，映射5000字节到虚拟内存中。

分析：因为单位物理页面的大小是4096字节，虽然被映射的文件只有5000字节，但是对应到进程虚拟地址区域的大小需要满足整页大小，因此mmap函数执行后，实际映射到虚拟内存区域8192个 字节，5000~8191的字节部分用零填充。映射后的对应关系如下图所示：



此时：

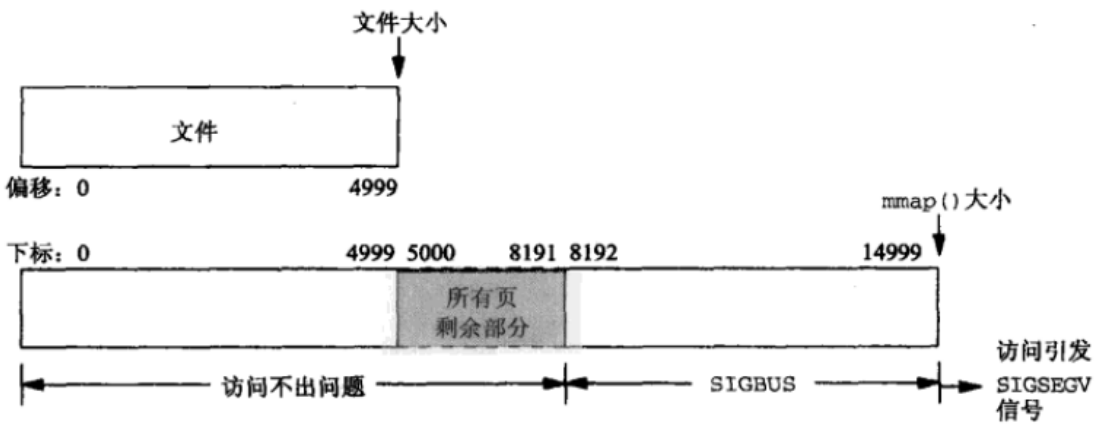
（1）读/写前5000个字节（0~4999），会返回操作文件内容。

（2）读字节5000~~8191时，结果全为0。写5000~~8191时，进程不会报错，但是所写的内容不会写入原文件中 。

（3）读/写8192以外的磁盘部分，会返回一个SIGSECV错误。

情形二：一个文件的大小是5000字节，mmap函数从一个文件的起始位置开始，映射15000字节到虚拟内存中，即映射大小超过了原始文件的大小。

分析：由于文件的大小是5000字节，和情形一一样，其对应的两个物理页。那么这两个物理页都是合法可以读写的，只是超出5000的部分不会体现在原文件中。由于程序要求映射15000字节，而文件只占两个物理页，因此8192字节~15000字节都不能读写，操作时会返回异常。如下图所示：



此时：

（1）进程可以正常读/写被映射的前5000字节(0~4999)，写操作的改动会在一定时间后反映在原文件中。

（2）对于5000~8191字节，进程可以进行读写过程，不会报错。但是内容在写入前均为0，另外，写入后不会反映在文件中。

（3）对于8192~14999字节，进程不能对其进行读写，会报SIGBUS错误。

（4）对于15000以外的字节，进程不能对其读写，会引发SIGSEGV错误。

情形三：一个文件初始大小为0，使用mmap操作映射了1000*4K的大小，即1000个物理页大约4M字节空间，mmap返回指针ptr。

分析：如果在映射建立之初，就对文件进行读写操作，由于文件大小为0，并没有合法的物理页对应，如同情形二一样，会返回SIGBUS错误。

但是如果，每次操作ptr读写前，先增加文件的大小，那么ptr在文件大小内部的操作就是合法的。例如，文件扩充4096字节，ptr就能操作ptr ~ [(char)ptr + 4095]的空间。只要文件扩充的范围在1000个物理页（映射范围）内，ptr都可以对应操作相同的大小。

这样，方便随时扩充文件空间，随时写入文件，不造成空间浪费。

本文转自：https://www.jianshu.com/p/007052ee3773

堆外内存

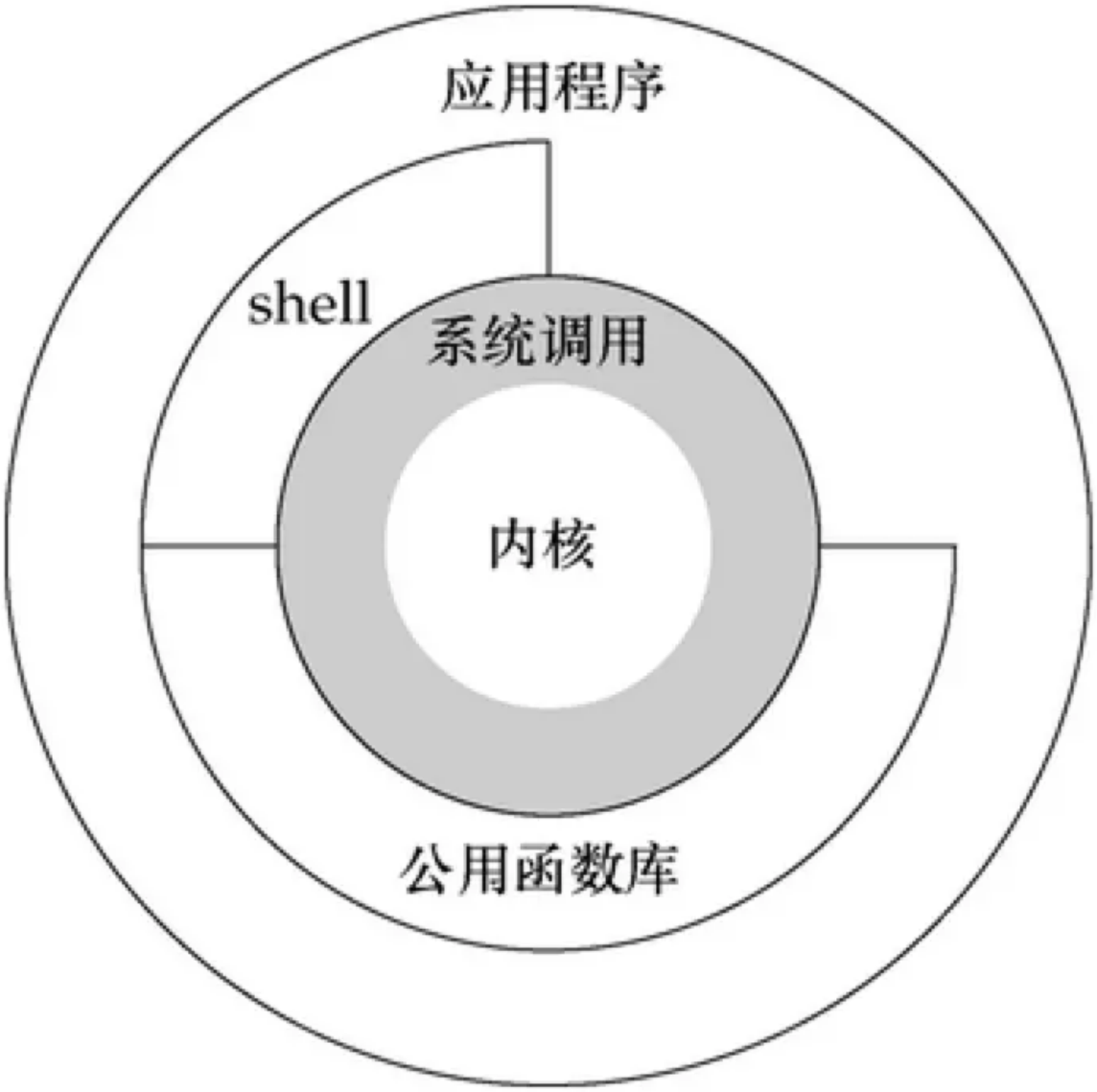
堆外内存是相对于堆内内存的一个概念。堆内内存是由JVM所管控的Java进程内存，我们平时在Java中创建的对象都处于堆内内存中，并且它们遵循JVM的内存管理机制，JVM会采用垃圾回收机制统一管理它们的内存。那么堆外内存就是存在于JVM管控之外的一块内存区域，因此它是不受JVM的管控。

在讲解DirectByteBuffer之前，需要先简单了解两个知识点

java引用类型，因为DirectByteBuffer是通过虚引用(Phantom Reference)来实现堆外内存的释放的。

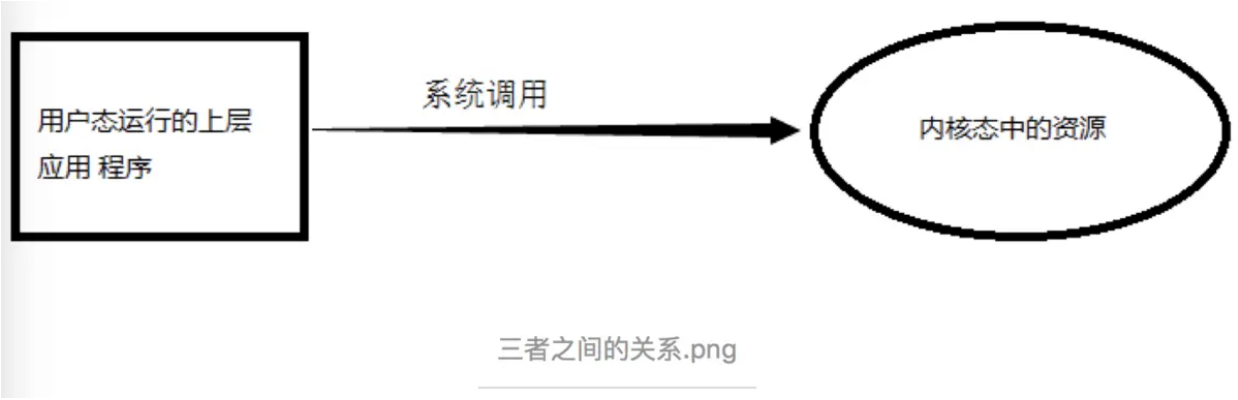
PhantomReference 是所有“弱引用”中最弱的引用类型。不同于软引用和弱引用，虚引用无法通过 get() 方法来取得目标对象的强引用从而使用目标对象，观察源码可以发现 get() 被重写为永远返回 null。 那虚引用到底有什么作用？其实虚引用主要被用来 跟踪对象被垃圾回收的状态，通过查看引用队列中是否包含对象所对应的虚引用来判断它是否 即将被垃圾回收，从而采取行动。它并不被期待用来取得目标对象的引用，而目标对象被回收前，它的引用会被放入一个 ReferenceQueue 对象中，从而达到跟踪对象垃圾回收的作用。 关于java引用类型的实现和原理可以阅读之前的文章Reference 、ReferenceQueue 详解 和Java 引用类型简述

关于linux的内核态和用户态



unix和linux的体系架构.png

- 内核态：控制计算机的硬件资源，并提供上层应用程序运行的环境。比如socket I/0操作或者文件的读写操作等
- 用户态：上层应用程序的活动空间，应用程序的执行必须依托于内核提供的资源。
- 系统调用：为了使上层应用能够访问到这些资源，内核为上层应用提供访问的接口。



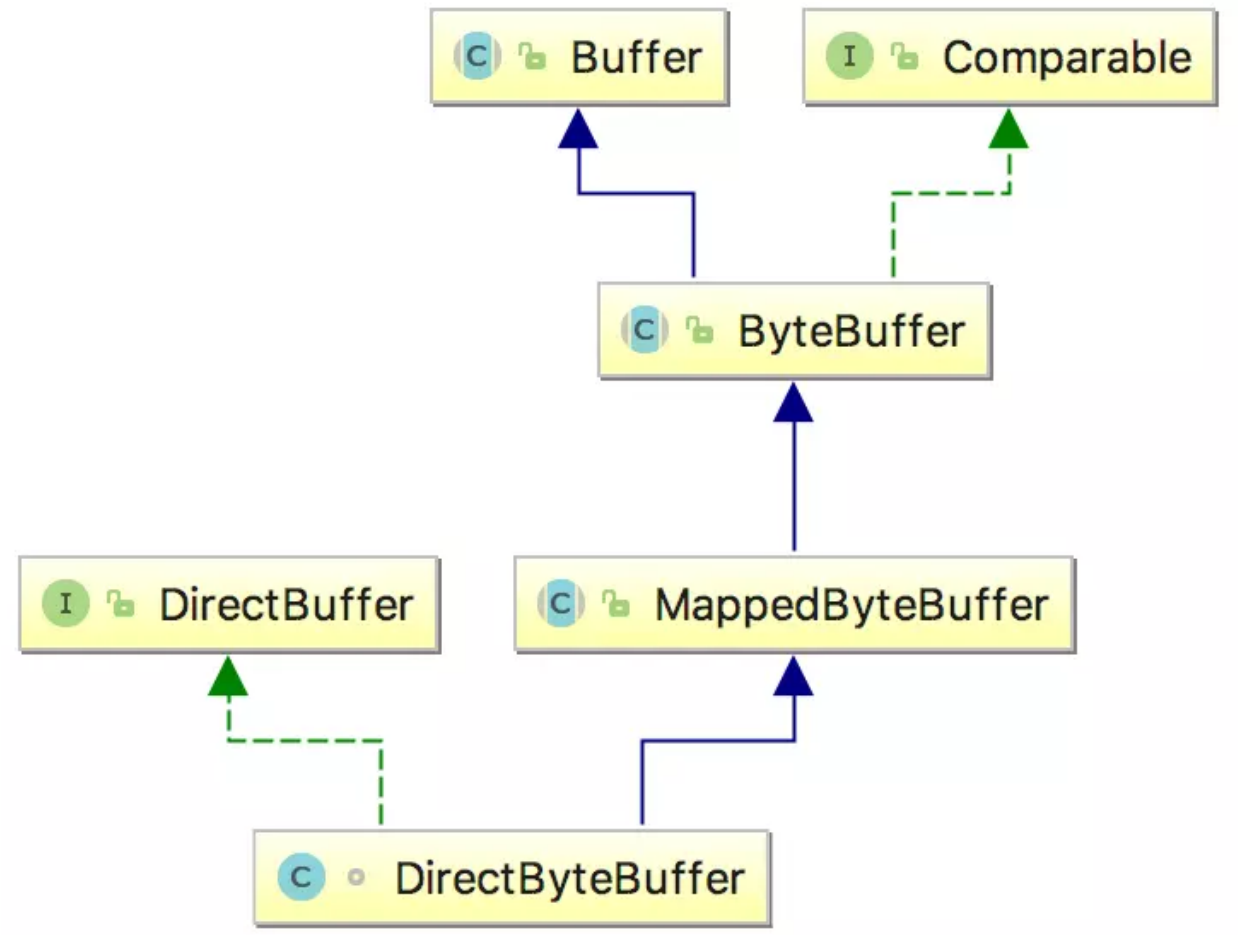
三者之间的关系.png

因此我们可以得知当我们通过JNI调用的native方法实际上就是从用户态切换到了内核态的一种方式。并且通过该系统调用使用操作系统所提供的功能。

Q：为什么需要用户进程(位于用户态中)要通过系统调用(Java中即使JNI)来调用内核态中的资源，或者说调用操作系统的服务了？ A：intel cpu提供Ring0-Ring3四种级别的运行模式，Ring0级别最高，Ring3最低。Linux使用了Ring3级别运行用户态，Ring0作为内核态。Ring3状态不能访问Ring0的地址空间，包括代码和数据。因此用户态是没有权限去操作内核态的资源的，它只能通过系统调用外完成用户态到内核态的切换，然后在完成相关操作后再有内核态切换回用户态。

DirectByteBuffer ———— 直接缓冲

DirectByteBuffer是Java用于实现堆外内存的一个重要类，我们可以通过该类实现堆外内存的创建、使用和销毁。



DirectByteBuffer该类本身还是位于Java内存模型的堆中。堆内内存是JVM可以直接管控、操纵。 而DirectByteBuffer中的unsafe.allocateMemory(size);是个一个native方法，这个方法分配的是堆外内存，通过C的malloc来进行分配的。分配的内存是系统本地的内存，并不在Java的内存中，也不属于JVM管控范围，所以在DirectByteBuffer一定会存在某种方式来操纵堆外内存。 在DirectByteBuffer的父类Buffer中有个address属性：

```
// Used only by direct buffers
// NOTE: hoisted here for speed in JNI GetDirectBufferAddress
long address;
```

address只会被直接缓存给使用到。之所以将address属性升级放在Buffer中，是为了在JNI调用GetDirectBufferAddress时提升它调用的速率。 address表示分配的堆外内存的地址。

DirectByteBuffer › DirectByteBuffer()

```
DirectByteBuffer(int cap) {                   // package-private

    super(-1, 0, cap, cap);
    boolean pa = VM.isDirectMemoryPageAligned();
    int ps = Bits.pageSize();
    long size = Math.max(1L, (long)cap + (pa ? ps : 0));
    Bits.reserveMemory(size, cap);

    long base = 0;
    try {
        base = unsafe.allocateMemory(size);
    } catch (OutOfMemoryError x) {
        Bits.unreserveMemory(size, cap);
        throw x;
    }
    unsafe.setMemory(base, size, (byte) 0);
    if (pa && (base % ps != 0)) {
        // Round up to page boundary
        address = base + ps - (base & (ps - 1));
    } else {
        address = base;
    }
    cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
    att = null;


}
```

unsafe.allocateMemory(size);分配完堆外内存后就会返回分配的堆外内存基地址，并将这个地址赋值给了address属性。这样我们后面通过JNI对这个堆外内存操作时都是通过这个address来实现的了。

在前面我们说过，在linux中内核态的权限是最高的，那么在内核态的场景下，操作系统是可以访问任何一个内存区域的，所以操作系统是可以访问到Java堆的这个内存区域的。 Q：那为什么操作系统不直接访问Java堆内的内存区域了？ A：这是因为JNI方法访问的内存区域是一个已经确定了的内存区域地质，那么该内存地址指向的是Java堆内内存的话，那么如果在操作系统正在访问这个内存地址的时候，Java在这个时候进行了GC操作，而GC操作会涉及到数据的移动操作[GC经常会进行先标志在压缩的操作。即，将可回收的空间做标志，然后清空标志位置的内存，然后会进行一个压缩，压缩就会涉及到对象的移动，移动的目的是为了腾出一块更加完整、连续的内存空间，以容纳更大的新对象]，数据的移动会使JNI调用的数据错乱。所以JNI调用的内存是不能进行GC操作的。

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。

FileChannelImpl › read()

```
public int read(ByteBuffer dst) throws IOException {
    ensureOpen();
    if (!readable)
        throw new NonReadableChannelException();
    synchronized (positionLock) {
        int n = 0;
        int ti = -1;
        try {
            begin();
            ti = threads.add();
            if (!isOpen())
                return 0;
            do {
                n = IOUtil.read(fd, dst, -1, nd);
            } while ((n == IOStatus.INTERRUPTED) && isOpen());
            return IOStatus.normalize(n);
        } finally {
            threads.remove(ti);
            end(n > 0);
            assert IOStatus.check(n);
        }
    }
}
```

```
    static int read(FileDescriptor var0, ByteBuffer var1, long var2,
NativeDispatcher var4) throws IOException {
        if (var1.isReadOnly()) {
            throw new IllegalArgumentException("Read-only buffer");
        } else if (var1 instanceof DirectBuffer) {
            return readIntoNativeBuffer(var0, var1, var2, var4);
        } else {
            // 分配临时的堆外内存
```

```
            ByteBuffer var5 = Util.getTemporaryDirectBuffer(var1.remaining());

            int var7;
            try {
                // File I/O 操作会将数据读入到堆外内存中
                int var6 = readIntoNativeBuffer(var0, var5, var2, var4);
                var5.flip();
                if (var6 > 0) {
                    // 将堆外内存的数据拷贝到堆内存中
                   var1.put(var5);
                }

                var7 = var6;
            } finally {
                // 里面会调用DirectBuffer.cleaner().clean()来释放临时的堆外内存
                Util.offerFirstTemporaryDirectBuffer(var5);
            }

            return var7;
        }
    }
```

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。

DirectByteBuffer堆外内存的创建和回收的源码解读

堆外内存分配

```
    DirectByteBuffer(int cap) {                   // package-private
        super(-1, 0, cap, cap);
        boolean pa = VM.isDirectMemoryPageAligned();
        int ps = Bits.pageSize();
        long size = Math.max(1L, (long)cap + (pa ? ps : 0));
        // 保留总分配内存(按页分配)的大小和实际内存的大小
        Bits.reserveMemory(size, cap);

        long base = 0;
        try {
            // 通过unsafe.allocateMemory分配堆外内存，并返回堆外内存的基地址
            base = unsafe.allocateMemory(size);
        } catch (OutOfMemoryError x) {
            Bits.unreserveMemory(size, cap);
            throw x;
        }
        unsafe.setMemory(base, size, (byte) 0);
```

```
        if (pa && (base % ps != 0)) {
            // Round up to page boundary
            address = base + ps - (base & (ps - 1));
        } else {
            address = base;
        }
        // 构建Cleaner对象用于跟踪DirectByteBuffer对象的垃圾回收，以实现当
DirectByteBuffer被垃圾回收时，堆外内存也会被释放
        cleaner = Cleaner.create(this, new Deallocator(base, size, cap));
        att = null;
    }
```

Bits.reserveMemory(size, cap) 方法

```
    static void reserveMemory(long size, int cap) {

        if (!memoryLimitSet && VM.isBooted()) {
            maxMemory = VM.maxDirectMemory();
            memoryLimitSet = true;
        }

        // optimist!
        if (tryReserveMemory(size, cap)) {
            return;
        }

        final JavaLangRefAccess jlra = SharedSecrets.getJavaLangRefAccess();

        // retry while helping enqueue pending Reference objects
        // which includes executing pending Cleaner(s) which includes
        // Cleaner(s) that free direct buffer memory
        while (jlra.tryHandlePendingReference()) {
            if (tryReserveMemory(size, cap)) {
                return;
            }
        }

        // trigger VM's Reference processing
        System.gc();

        // a retry loop with exponential back-off delays
        // (this gives VM some time to do it's job)
        boolean interrupted = false;
        try {
            long sleepTime = 1;
            int sleeps = 0;
            while (true) {
                if (tryReserveMemory(size, cap)) {
                    return;
                }
                if (sleeps >= MAX_SLEEPS) {
                    break;
```

```
                }
                if (!jlra.tryHandlePendingReference()) {
                    try {
                        Thread.sleep(sleepTime);
                        sleepTime <<= 1;
                        sleeps++;
                    } catch (InterruptedException e) {
                        interrupted = true;
                    }
                }
            }

            // no luck
            throw new OutOfMemoryError("Direct buffer memory");

        } finally {
            if (interrupted) {
                // don't swallow interrupts
                Thread.currentThread().interrupt();
            }
        }
    }
```

该方法用于在系统中保存总分配内存(按页分配)的大小和实际内存的大小。

其中，如果系统中内存(即，堆外内存)不够的话：

```
        final JavaLangRefAccess jlra = SharedSecrets.getJavaLangRefAccess();

        // retry while helping enqueue pending Reference objects
        // which includes executing pending Cleaner(s) which includes
        // Cleaner(s) that free direct buffer memory
        while (jlra.tryHandlePendingReference()) {
            if (tryReserveMemory(size, cap)) {
                return;
            }
        }
```

jlra.tryHandlePendingReference()会触发一次非堵塞的Reference#tryHandlePending(false)。该方法会将已经被JVM垃圾回收的DirectBuffer对象的堆外内存释放。 因为在Reference的静态代码块中定义了：

```
        SharedSecrets.setJavaLangRefAccess(new JavaLangRefAccess() {
            @Override
            public boolean tryHandlePendingReference() {
                return tryHandlePending(false);
            }
        });
```

如果在进行一次堆外内存资源回收后，还不够进行本次堆外内存分配的话，则

```
// trigger VM's Reference processing
System.gc();
```

System.gc()会触发一个full gc，当然前提是你没有显示的设置-XX:+DisableExplicitGC来禁用显式GC。并且你需要知道，调用System.gc()并不能够保证full gc马上就能被执行。所以在后面打代码中，会进行最多9次尝试，看是否有足够的可用堆外内存来分配堆外内存。并且每次尝试之前，都对延迟等待时间，已给JVM足够的时间去完成full gc操作。如果9次尝试后依旧没有足够的可用堆外内存来分配本次堆外内存，则抛出OutOfMemoryError("Direct buffer memory")异常。

```
// a retry loop with exponential back-off delays
// (this gives VM some time to do it's job)
boolean interrupted = false;
try {
    long sleepTime = 1;
    int sleeps = 0;
    while (true) {
        if (tryReserveMemory(size, cap)) {
            return;
        }
        if (sleeps >= MAX_SLEEPS) {
            break;
        }
        if (!jlra.tryHandlePendingReference()) {
            try {
                Thread.sleep(sleepTime);
                sleepTime <<= 1;
                sleeps++;
            } catch (InterruptedException e) {
                interrupted = true;
            }
        }
    }

    // no luck
    throw new OutOfMemoryError("Direct buffer memory");
```

9

注意，这里之所以用使用full gc的很重要的一个原因是：System.gc()会对新生代的老生代都会进行内存回收，这样会比较彻底地回收DirectByteBuffer对象以及他们关联的堆外内存. DirectByteBuffer对象本身其实是很小的，但是它后面可能关联了一个非常大的堆外内存，因此我们通常称之为冰山对象. 我们做ygc的时候会将新生代里的不可达的DirectByteBuffer对象及其堆外内存回收了，但是无法对old里的DirectByteBuffer对象及其堆外内存进行回收，这也是我们通常碰到的最大的问题。（并且堆外内存多用于生命期中等或较长的对象）如果有大量的DirectByteBuffer对象移到了old，但是又一直没有做cms gc或者full gc，而只进行ygc，那么我们的物理内存可能被慢慢耗光，但是我们还不知道发生了什么，因为heap明明剩余的内存还很多(前提是我们禁用了System.gc – JVM参数DisableExplicitGC)。

总的来说，Bits.reserveMemory(size, cap)方法在可用堆外内存不足以分配给当前要创建的堆外内存大小时，会实现以下的步骤来尝试完成本次堆外内存的创建： ① 触发一次非堵塞的Reference#tryHandlePending(false)。该方法会将已经被JVM垃圾回收的DirectBuffer对象的堆外内存释放。 ② 如果进行一次堆外内存资源回收后，还不够进行本次堆外内存分配的话，则进行 System.gc()。System.gc()会触发一个full gc，但你需要知道，调用System.gc()并不能够保证full gc马上就能被执行。所以在后面打代码中，会进行最多9次尝试，看是否有足够的可用堆外内存来分配堆外内存。并且每次尝试之前，都对延迟等待时间，已给JVM足够的时间去完成full gc操作。 注意，如果你设置了-XX:+DisableExplicitGC，将会禁用显示GC，这会使System.gc()调用无效。 ③ 如果9次尝试后依旧没有足够的可用堆外内存来分配本次堆外内存，则抛出OutOfMemoryError("Direct buffer memory")异常。

那么可用堆外内存到底是多少了？，即默认堆外存内存有多大： ① 如果我们没有通过-XX:MaxDirectMemorySize来指定最大的堆外内存。则👇 ② 如果我们没通过-Dsun.nio.MaxDirectMemorySize指定了这个属性，且它不等于-1。则👇 ③ 那么最大堆外内存的值来自于directMemory = Runtime.getRuntime().maxMemory()，这是一个native方法

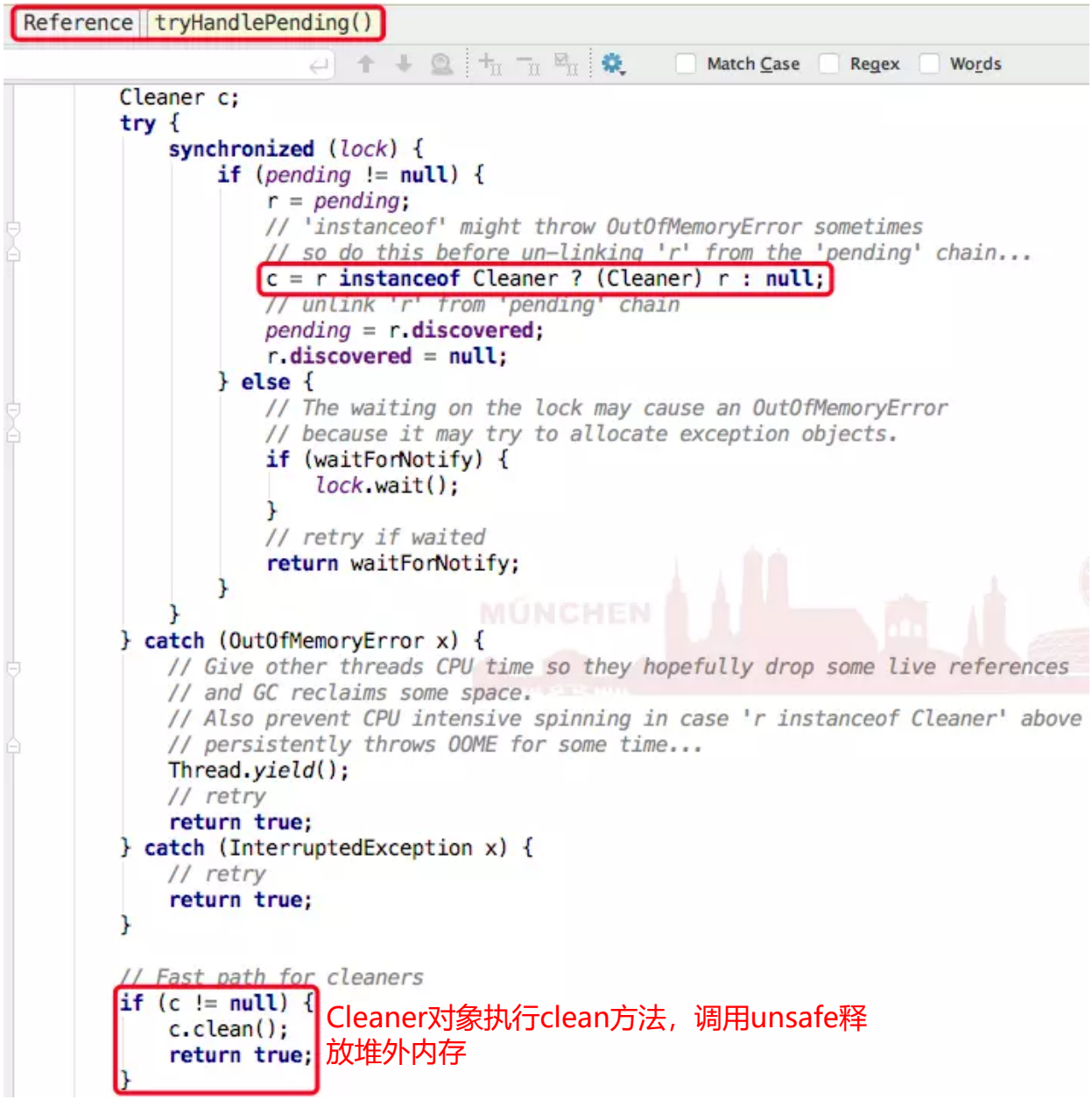
```
JNIEXPORT jlong JNICALL
Java_java_lang_Runtime_maxMemory(JNIEnv *env, jobject this)
{
    return JVM_MaxMemory();
}

JVM_ENTRY_NO_ENV(jlong, JVM_MaxMemory(void))
  JVMWrapper("JVM_MaxMemory");
  size_t n = Universe::heap()->max_capacity();
  return convert_size_t_to_jlong(n);
JVM_END
```

其中在我们使用CMS GC的情况下也就是我们设置的-Xmx的值里除去一个survivor的大小就是默认的堆外内存的大小了。

堆外内存回收

Cleaner是PhantomReference的子类，并通过自身的next和prev字段维护的一个双向链表。PhantomReference的作用在于跟踪垃圾回收过程，并不会对对象的垃圾回收过程造成任何的影响。 所以cleaner = Cleaner.create(this, new Deallocator(base, size, cap)); 用于对当前构造的DirectByteBuffer对象的垃圾回收过程进行跟踪。 当DirectByteBuffer对象从pending状态 ——> enqueue状态时，会触发Cleaner的clean()，而Cleaner的clean()的方法会实现通过unsafe对堆外内存的释放。

Cleaner对象跟踪堆内存DirectByteBuffer垃圾回收状态




```
public void clean() {
    if(remove(this)) {
        try {
            this.thunk.run();
        } catch (final Throwable var2) {
            AccessController.doPrivileged(run() -> {
                    if(System.err != null) {
                        (new Error("Cleaner terminated abnormally", var2)).printStackTrace();
                    }

                    System.exit(1);
                    return null;
            });
        }

    }
}
```

虽然Cleaner不会调用到Reference.clear()，但Cleaner的clean()方法调用了remove(this)，即将当前Cleaner从Cleaner链表中移除，这样当clean()执行完后，Cleaner就是一个无引用指向的对象了，也就是可被GC回收的对象。

thunk方法：

```
private static class Deallocator
    implements Runnable
{

    private static Unsafe unsafe = Unsafe.getUnsafe();

    private long address;
    private long size;
    private int capacity;

    private Deallocator(long address, long size, int capacity) {
        assert (address != 0);
        this.address = address;
        this.size = size;
        this.capacity = capacity;
    }

    public void run() {
        if (address == 0) {
            // Paranoia
            return;
        }
        unsafe.freeMemory(address);
        address = 0;
        Bits.unreserveMemory(size, capacity);
    }

}
```

Clearner对象指向为空，可以被GC回收。并且调用了unsafe对堆外内存进行释放

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。

参考文章

http://lovestblog.cn/blog/2015/05/12/direct-buffer/
http://www.infoq.com/cn/news/2014/12/external-memory-heap-memory
http://www.jianshu.com/p/85e931636f27 圣思园《并发与Netty》课程