**#成神之路#**

第001期：JVM-JVM内存结构

《深入理解Java虚拟机》

[链接：Java虚拟机的内存组成以及堆内存介绍-HollisChuang's Blog](http://www.hollischuang.com/archives/80)

[链接：Java堆和栈看这篇就够 - Johnny-Zhuang's Technology Blog](https://iamjohnnyzhuang.github.io/java/2016/07/12/Java%E5%A0%86%E5%92%8C%E6%A0%88%E7%9C%8B%E8%BF%99%E7%AF%87%E5%B0%B1%E5%A4%9F.html)

[链接：Java虚拟机的堆、栈、堆栈如何去理解？ - 知乎](https://www.zhihu.com/question/29833675)

[链接：Java 内存之方法区和运行时常量池 - 漠然的博客 | mritd Blog](https://mritd.me/2016/03/22/Java-%E5%86%85%E5%AD%98%E4%B9%8B%E6%96%B9%E6%B3%95%E5%8C%BA%E5%92%8C%E8%BF%90%E8%A1%8C%E6%97%B6%E5%B8%B8%E9%87%8F%E6%B1%A0/)

[链接：从0到1起步-跟我进入堆外内存的奇妙世界 - 简书](https://www.jianshu.com/p/50be08b54bee)

## [Java虚拟机的内存组成以及堆内存介绍](http://www.hollischuang.com/archives/80)

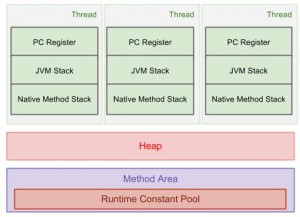
2015-04-14 分类：[Java](http://www.hollischuang.com/archives/category/java) 阅读(9513) 评论(3)   
  
本站采用[[知识共享署名-非商业性使用-相同方式共享 许可协议](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.zh)]进行许可，转载请在正文明显处注明原文地址

什么是Java虚拟机这里就不介绍了，不明白的可以另外一篇博文：[JDK,JRE,JVM区别与联系](http://www.hollischuang.com/archives/78)

**一、java内存组成介绍：堆(Heap)和非堆(Non-heap)内存**

按照官方的说法：“Java 虚拟机具有一个堆，堆是运行时数据区域，所有类实例和数组的内存均从此处分配。堆是在 Java 虚拟机启动时创建的。”“在JVM中堆之外的内存称为非堆内存(Non-heap memory)”。可以看出**JVM主要管理两种类型的内存：堆和非堆**。简单来说堆就是Java代码可及的内存，是留给开发人员使用的；非堆就是JVM留给 自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法 的代码都在非堆内存中。

**二、JVM内存区域模型**

[](http://www.hollischuang.com/wp-content/uploads/2015/04/2354447461.jpg)

**1.方法区** 也称”永久代” 、“非堆”， 它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。默认最小值为16MB，最大值为64MB，可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

运行时常量池：是方法区的一部分，其中的主要内容来自于JVM对Class的加载。

Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种符号引用，这部分内容将在类加载后放到方法区的运行时常量池中。

**2.虚拟机栈**

描述的是java 方法执行的内存模型：每个方法被执行的时候 都会创建一个“栈帧”用于存储局部变量表(包括参数)、操作栈、方法出口等信息。每个方法被调用到执行完的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。声明周期与线程相同，是线程私有的。

局部变量表存放了编译器可知的各种基本数据类型(boolean、byte、char、short、int、float、long、double)、对象引用(引用指针，并非对象本身)，其中64位长度的long和double类型的数据会占用2个局部变量的空间，其余数据类型只占1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在栈帧中分配多大的局部变量是完全确定的，在运行期间栈帧不会改变局部变量表的大小空间。

**3.本地方法栈**

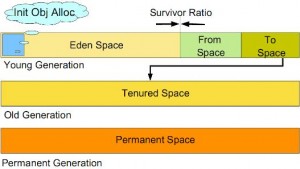
与虚拟机栈基本类似，区别在于虚拟机栈为虚拟机执行的java方法服务，而本地方法栈则是为Native方法服务。

**4.堆**

也叫做java 堆、GC堆是java虚拟机所管理的内存中最大的一块内存区域，也是被各个线程共享的内存区域，在JVM启动时创建。该内存区域存放了对象实例及数组(所有new的对象)。其大小通过-Xms(最小值)和-Xmx(最大值)参数设置，-Xms为JVM启动时申请的最小内存，默认为操作系统物理内存的1/64但小于1G，-Xmx为JVM可申请的最大内存，默认为物理内存的1/4但小于1G，默认当空余堆内存小于40%时，JVM会增大Heap到-Xmx指定的大小，可通过-XX:MinHeapFreeRation=来指定这个比列；当空余堆内存大于70%时，JVM会减小heap的大小到-Xms指定的大小，可通过XX:MaxHeapFreeRation=来指定这个比列，对于运行系统，为避免在运行时频繁调整Heap的大小，通常-Xms与-Xmx的值设成一样。

由于现在收集器都是采用分代收集算法，堆被划分为新生代和老年代。新生代主要存储新创建的对象和尚未进入老年代的对象。老年代存储经过多次新生代GC(Minor GC)任然存活的对象。

新生代： 程序新创建的对象都是从新生代分配内存，新生代由Eden Space和两块相同大小的Survivor Space(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRation来调整Eden Space及Survivor Space的大小。 老年代： 用于存放经过多次新生代GC任然存活的对象，例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：①.大对象，可通过启动参数设置-XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。②.大的数组对象，切数组中无引用外部对象。 老年代所占的内存大小为-Xmx对应的值减去-Xmn对应的值。

[](http://www.hollischuang.com/wp-content/uploads/2015/04/2838681554.jpg)

Young Generation 即图中的Eden + From Space + To Space

Eden 存放新生的对象

Survivor Space 有两个，存放每次垃圾回收后存活的对象

Old Generation Tenured Generation 即图中的Old Space

主要存放应用程序中生命周期长的存活对象

**5.程序计数器**

是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。

**三、直接内存**

直接内存并不是虚拟机内存的一部分，也不是Java虚拟机规范中定义的内存区域。jdk1.4中新加入的NIO，引入了通道与缓冲区的IO方式，它可以调用Native方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

**四、Java堆内存的10个要点**

1. Java堆内存是操作系统分配给JVM的内存的一部分。
2. 当我们创建对象时，它们存储在Java堆内存中。
3. 为了便于垃圾回收，Java堆空间分成三个区域，分别叫作New Generation, Old Generation或叫作Tenured Generation，还有Perm Space。
4. 你可以通过用JVM的命令行选项 -Xms, -Xmx, -Xmn来调整Java堆空间的大小。不要忘了在大小后面加上”M”或者”G”来表示单位。举个例子，你可以用 -Xmx256m来设置堆内存最大的大小为256MB。
5. 你可以用JConsole或者 Runtime.maxMemory(), Runtime.totalMemory(), Runtime.freeMemory()来查看Java中堆内存的大小。
6. 你可以使用命令“jmap”来获得heap dump，用“jhat”来分析heap dump。
7. Java堆空间不同于栈空间，栈空间是用来储存调用栈和局部变量的。
8. Java垃圾回收器是用来将死掉的对象(不再使用的对象)所占用的内存回收回来，再释放到Java堆空间中。
9. 当你遇到java.lang.outOfMemoryError时，不要紧张，有时候仅仅增加堆空间就可以了，但如果经常出现的话，就要看看Java程序中是不是存在内存泄露了。
10. 请使用Profiler和Heap dump分析工具来查看Java堆空间，可以查看给每个对象分配了多少内存。

【公告】[版权声明](http://www.hollischuang.com/%E8%BD%AC%E8%BD%BD%E8%AF%B4%E6%98%8E)

(全文完)

## 一、相关特征

### 1、方法区特征

* 同 Java 堆一样，方法区也是全局共享的一块内存区域
* 方法区的作用是存储 Java 类的结构信息，当我们创建对象实例后，**对象的类型信息存储在方法堆之中，实例数据存放在堆中；实例数据指的是在 Java 中创建的各种实例对象以及它们的值，类型信息指的是定义在 Java 代码中的常量、静态变量、以及在类中声明的各种方法、方法字段等等；同事可能包括即时编译器编译后产生的代码数据。**
* JVMS 不要求该区域实现自动的内存管理，但是商用 JVM 一般都已实现该区域的自动内存管理。
* 方法区分配内存可以不连续，可以动态扩展。
* 该区域并非像 JMM 规范描述的那样数据一旦放进去就属于 “永久代”；**在该区域进行内存回收的主要目的是对常量池的回收和对内存数据的卸载；一般来说这个区域的内存回收效率比起 Java 堆要低得多。**
* 当方法区无法满足内存需求时，将抛出 OutOfMemoryError 异常。

### 2、运行时常量池的特征

* **运行时常量池是方法区的一部分，**所以也是全局共享的。
* **其作用是存储 Java 类文件常量池中的符号信息。**
* **class 文件中存在常量池(非运行时常量池)，其在编译阶段就已经确定；JVM 规范对 class 文件结构有着严格的规范，必须符合此规范的 class 文件才会被 JVM 认可和装载。**
* **运行时常量池** 中保存着一些 class 文件中描述的符号引用，同时还会将这些符号引用所翻译出来的直接引用存储在 **运行时常量池** 中。
* **运行时常量池相对于 class 常量池一大特征就是其具有动态性，Java 规范并不要求常量只能在运行时才产生，也就是说运行时常量池中的内容并不全部来自 class 常量池，class 常量池并非运行时常量池的唯一数据输入口；在运行时可以通过代码生成常量并将其放入运行时常量池中。**
* 同方法区一样，当运行时常量池无法申请到新的内存时，将抛出 OutOfMemoryError 异常。

## 二、HotSpot 方法区变迁

### 1、JDK1.2 ~ JDK6

在 JDK1.2 ~ JDK6 的实现中，HotSpot 使用永久代实现方法区；HotSpot 使用 GC 分代实现方法区带来了很大便利；

### 2、JDK7

由于 GC 分代技术的影响，使之许多优秀的内存调试工具无法在 Oracle HotSpot之上运行，必须单独处理；并且 Oracle 同时收购了 BEA 和 Sun 公司，同时拥有 JRockit 和 HotSpot，在将 JRockit 许多优秀特性移植到 HotSpot 时由于 GC 分代技术遇到了种种困难，**所以从 JDK7 开始 Oracle HotSpot 开始移除永久代。**

**JDK7中符号表被移动到 Native Heap中，字符串常量和类引用被移动到 Java Heap中。**

### 3、JDK8

**在 JDK8 中，永久代已完全被元空间(Meatspace)所取代。**

## 三、永久代变迁产生的影响

#### 1、测试代码1

**public** **class** **Test1** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

String s1 **=** **new** StringBuilder**(**"漠"**).**append**(**"然"**).**toString**();**

System**.**out**.**println**(**s1**.**intern**()** **==** s1**);**

String s2 **=** **new** StringBuilder**(**"漠"**).**append**(**"然"**).**toString**();**

System**.**out**.**println**(**s2**.**intern**()** **==** s2**);**

**}**

**}**

以上代码，在 JDK6 下执行结果为 false、false，在 JDK7 以上执行结果为 true、false。

**首先明确两点：** 1、在 Java 中直接使用双引号展示的字符串将会在常量池中直接创建。 2、String 的 intern 方法首先将尝试在常量池中查找该对象，如果找到则直接返回该对象在常量池中的地址；找不到则将该对象放入常量池后再返回其地址。**JDK6 常量池在方法区，频繁调用该方法可能造成 OutOfMemoryError。**

**产生两种结果的原因：**

在 JDK6 下 s1、s2 指向的是新创建的对象，**该对象将在 Java Heap 中创建，所以 s1、s2 指向的是 Java Heap 中的内存地址；**调用 intern 方法后将尝试在常量池中查找该对象，没找到后将其放入常量池并返回，**所以此时 s1/s2.intern() 指向的是常量池中的地址，JDK6常量池在方法区，与堆隔离，；所以 s1.intern()==s1 返回false。**

#### 2、测试代码2

**public** **class** **Test2** **{**

**public** **static** **void** **main(**String**[]** args**)** **{**

*/\*\**

*\* 首先设置 持久代最大和最小内存占用(限定为10M)*

*\* VM args: -XX:PermSize=10M -XX:MaxPremSize=10M*

*\*/*

List**<**String**>** list **=** **new** ArrayList**<**String**>();**

*// 无限循环 使用 list 对其引用保证 不被GC intern 方法保证其加入到常量池中*

**int** i **=** 0**;**

**while** **(true)** **{**

*// 此处永久执行，最多就是将整个 int 范围转化成字符串并放入常量池*

list**.**add**(**String**.**valueOf**(**i**++).**intern**());**

**}**

**}**

**}**

以上代码在 JDK6 下会出现 Perm 内存溢出，JDK7 or high 则没问题。

**原因分析：**

**JDK6 常量池存在方法区，设置了持久代大小后，不断while循环必将撑满 Perm 导致内存溢出；JDK7 常量池被移动到 Native Heap(Java Heap)，所以即使设置了持久代大小，也不会对常量池产生影响；不断while循环在当前的代码中，所有int的字符串相加还不至于撑满 Heap 区，所以不会出现异常。** 转载请注明出处，本文采用 [CC4.0](http://creativecommons.org/licenses/by-nc-nd/4.0/) 协议授权

## 从0到1起步-跟我进入堆外内存的奇妙世界

[](https://www.jianshu.com/u/a9b2d43bb94e)

[小程故事多](https://www.jianshu.com/u/a9b2d43bb94e) 关注

2016.10.10 19:03\* 字数 2034 阅读 6043评论 5喜欢 34

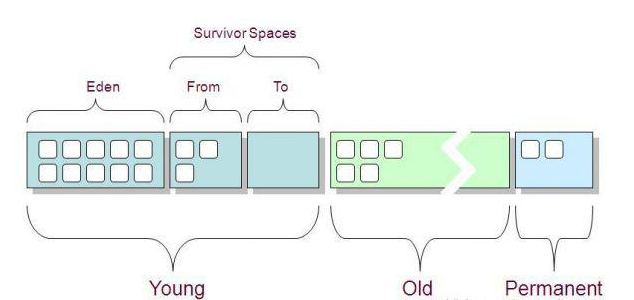
堆外内存一直是Java业务开发人员难以企及的隐藏领域，究竟他是干什么的，以及如何更好的使用呢？那就请跟着我进入这个世界吧。

**一、**什么是堆外内存

**1、堆内内存（on-heap memory）回顾**  
堆外内存和堆内内存是相对的二个概念，其中堆内内存是我们平常工作中接触比较多的，我们在jvm参数中只要使用-Xms，-Xmx等参数就可以设置堆的大小和最大值，理解jvm的堆还需要知道下面这个公式：

堆内内存 = 新生代+老年代+持久代

如下面的图所示：



Paste\_Image.png

在使用堆内内存（on-heap memory）的时候，完全遵守JVM虚拟机的内存管理机制，采用垃圾回收器（GC）统一进行内存管理，GC会在某些特定的时间点进行一次彻底回收，也就是Full GC，GC会对所有分配的堆内内存进行扫描，在这个过程中会对JAVA应用程序的性能造成一定影响，还可能会产生Stop The World。

**常见的垃圾回收算法主要有：**

* 引用计数器法（Reference Counting）
* 标记清除法（Mark-Sweep）
* 复制算法（Coping）
* 标记压缩法（Mark-Compact）
* 分代算法（Generational Collecting）
* 分区算法（Region）  
  **注：**在这里我们不对各个算法进行深入介绍，感兴趣的同学可以关注我的下一篇关于垃圾回收算法的介绍分享。

**2、堆外内存（off-heap memory）介绍**

和堆内内存相对应，堆外内存就是把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机），这样做的结果就是能够在一定程度上减少垃圾回收对应用程序造成的影响。

作为JAVA开发者我们经常用java.nio.DirectByteBuffer对象进行堆外内存的管理和使用，它会在对象创建的时候就分配堆外内存。

DirectByteBuffer类是在Java Heap外分配内存，对堆外内存的申请主要是通过成员变量unsafe来操作，下面介绍构造方法

DirectByteBuffer(int cap) {

super(-1, 0, cap, cap);

*//内存是否按页分配对齐*

boolean pa = VM.isDirectMemoryPageAligned();

*//获取每页内存大小*

int ps = Bits.pageSize();

*//分配内存的大小，如果是按页对齐方式，需要再加一页内存的容量*

long size = Math.max(1L, (long)cap + (pa ? ps : 0));

*//用Bits类保存总分配内存(按页分配)的大小和实际内存的大小*

Bits.reserveMemory(size, cap);

long base = 0;

try {

*//在堆外内存的基地址，指定内存大小*

base = unsafe.allocateMemory(size);

} catch (OutOfMemoryError x) {

Bits.unreserveMemory(size, cap);

throw x;

}

unsafe.setMemory(base, size, (byte) 0);

*//计算堆外内存的基地址*

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;

}

**注：**在Cleaner 内部中通过一个列表，维护了一个针对每一个 directBuffer 的一个回收堆外内存的 线程对象(Runnable)，回收操作是发生在 Cleaner 的 clean() 方法中。

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);

}

}

**二、使用堆外内存的优点**

**1、减少了垃圾回收**  
因为垃圾回收会暂停其他的工作。

**2、加快了复制的速度**  
堆内在flush到远程时，会先复制到直接内存（非堆内存），然后在发送；而堆外内存相当于省略掉了这个工作。

同样任何一个事物使用起来有优点就会有缺点，堆外内存的缺点就是内存难以控制，使用了堆外内存就间接失去了JVM管理内存的可行性，改由自己来管理，当发生内存溢出时排查起来非常困难。

**三、使用DirectByteBuffer的注意事项**

java.nio.DirectByteBuffer对象在创建过程中会先通过Unsafe接口直接通过os::malloc来分配内存，然后将内存的起始地址和大小存到java.nio.DirectByteBuffer对象里，这样就可以直接操作这些内存。这些内存只有在DirectByteBuffer回收掉之后才有机会被回收，因此如果这些对象大部分都移到了old，但是一直没有触发CMS GC或者Full GC，那么悲剧将会发生，因为你的物理内存被他们耗尽了，因此为了避免这种悲剧的发生，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。

**四、DirectByteBuffer使用测试**

我们在写NIO程序经常使用ByteBuffer来读取或者写入数据，那么使用ByteBuffer.allocate(capability)还是使用ByteBuffer.allocteDirect(capability)来分配缓存了？第一种方式是分配JVM堆内存，属于GC管辖范围，由于需要拷贝所以速度相对较慢；第二种方式是分配OS本地内存，不属于GC管辖范围，由于不需要内存拷贝所以速度相对较快。

代码如下：

package com.stevex.app.nio;

import java.nio.ByteBuffer;

import java.util.concurrent.TimeUnit;

public class DirectByteBufferTest {

public static void main(String[] args) throws InterruptedException{

*//分配128MB直接内存*

ByteBuffer bb = ByteBuffer.allocateDirect(1024\*1024\*128);

TimeUnit.SECONDS.sleep(10);

System.out.println("ok");

}

}

测试用例1：设置JVM参数-Xmx100m，运行异常，因为如果没设置-XX:MaxDirectMemorySize，则默认与-Xmx参数值相同，分配128M直接内存超出限制范围。

Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory

at java.nio.Bits.reserveMemory(Bits.java:658)

at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)

at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)

at com.stevex.app.nio.DirectByteBufferTest.main(DirectByteBufferTest.java:8)

测试用例2：设置JVM参数-Xmx256m，运行正常，因为128M小于256M，属于范围内分配。

测试用例3：设置JVM参数-Xmx256m -XX:MaxDirectMemorySize=100M，运行异常，分配的直接内存128M超过限定的100M。

Exception in thread "main" java.lang.OutOfMemoryError: Direct buffer memory

at java.nio.Bits.reserveMemory(Bits.java:658)

at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123)

at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:306)

at com.stevex.app.nio.DirectByteBufferTest.main(DirectByteBufferTest.java:8)

测试用例4：设置JVM参数-Xmx768m，运行程序观察内存使用变化，会发现clean()后内存马上下降，说明使用clean()方法能有效及时回收直接缓存。  
代码如下：

package com.stevex.app.nio;

import java.nio.ByteBuffer;

import java.util.concurrent.TimeUnit;

import sun.nio.ch.DirectBuffer;

public class DirectByteBufferTest {

public static void main(String[] args) throws InterruptedException{

*//分配512MB直接缓存*

ByteBuffer bb = ByteBuffer.allocateDirect(1024\*1024\*512);

TimeUnit.SECONDS.sleep(10);

*//清除直接缓存*

((DirectBuffer)bb).cleaner().clean();

TimeUnit.SECONDS.sleep(10);

System.out.println("ok");

}

}

**五、细说System.gc方法**

**1、JDK里的System.gc的实现**

*/\*\**

*\* Runs the garbage collector.*

*\* <p>*

*\* Calling the <code>gc</code> method suggests that the Java Virtual*

*\* Machine expend effort toward recycling unused objects in order to*

*\* make the memory they currently occupy available for quick reuse.*

*\* When control returns from the method call, the Java Virtual*

*\* Machine has made a best effort to reclaim space from all discarded*

*\* objects.*

*\* <p>*

*\* The call <code>System.gc()</code> is effectively equivalent to the*

*\* call:*

*\* <blockquote><pre>*

*\* Runtime.getRuntime().gc()*

*\* </pre></blockquote>*

*\**

*\* @see java.lang.Runtime#gc()*

*\*/*

public static void gc() {

Runtime.getRuntime().gc();

}

其实发现System.gc方法其实是调用的Runtime.getRuntime.gc()，我们再接着看。

*/\**

*运行垃圾收集器。*

*调用此方法表明，java虚拟机扩展*

*努力回收未使用的对象，以便内存可以快速复用，*

*当控制从方法调用返回的时候，虚拟机尽力回收被丢弃的对象*

*\*/*

public native void gc();

这里看到gc方法是native的，在java层面只能到此结束了，代码只有这么多，要了解更多，可以看方法上面的注释，不过我们需要更深层次地来了解其实现，那还是准备好进入到jvm里去看看。

**2、System.gc的作用有哪些**  
说起堆外内存免不了要提及System.gc方法，下面就是使用了System.gc的作用是什么？

* 做一次full gc
* 执行后会暂停整个进程。
* System.gc我们可以禁掉，使用-XX:+DisableExplicitGC，  
  其实一般在cms gc下我们通过-XX:+ExplicitGCInvokesConcurrent也可以做稍微高效一点的gc，也就是并行gc。
* 最常见的场景是RMI/NIO下的堆外内存分配等

**注：**  
如果我们使用了堆外内存，并且用了DisableExplicitGC设置为true，那么就是禁止使用System.gc，这样堆外内存将无从触发极有可能造成内存溢出错误，在这种情况下可以考虑使用ExplicitGCInvokesConcurrent参数。

说起Full gc我们最先想到的就是**stop thd world**，这里要先提到VMThread，在jvm里有这么一个线程不断轮询它的队列，这个队列里主要是存一些VM\_operation的动作，比如最常见的就是内存分配失败要求做GC操作的请求等，在对gc这些操作执行的时候会先将其他业务线程都进入到安全点，也就是这些线程从此不再执行任何字节码指令，只有当出了安全点的时候才让他们继续执行原来的指令，因此这其实就是我们说的stop the world(STW)，整个进程相当于静止了。

**六、开源堆外缓存框架**

关于堆外缓存的开源实现。查询了一些资料后了解到的主要有：

* Ehcache 3.0：3.0基于其商业公司一个非开源的堆外组件的实现。
* Chronical Map：OpenHFT包括很多类库，使用这些类库很少产生垃圾，并且应用程序使用这些类库后也很少发生Minor GC。类库主要包括：Chronicle Map，Chronicle Queue等等。
* OHC：来源于Cassandra 3.0， Apache v2。
* Ignite: 一个规模宏大的内存计算框架，属于Apache项目。

第002期：JVM-Java内存模型

第003期：JVM-Java对象模型

第004期：Java的垃圾回收机制

第005期：JVM-HotSpot虚拟机-即时编译器、编译优化等相关知识

第006期：Java的类加载机制-双亲委派，破坏双亲委派