# JVM虚拟机笔记

## 第2章 Java内存区域与内存溢出异常

### 运行时数据区：

方法区、程序计数器、Java虚拟机栈、本地方法栈、Java堆、运行时常量池（方法区的一部分）

其中，程序计数器、Java虚拟机栈、本地方法栈随线程而生而灭。

#### 程序计数器：

一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。

每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，这类内存区域为“线程私有”的内存。

如果正在执行Java方法，记录的是正在执行的虚拟机字节码指令的地址；如果是Native方法，计数器值为空(Undefined)，此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。

#### Java虚拟机栈：

和程序计数器一样是线程私有的，生命周期和线程相同。

虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用到执行完成的过程对应栈帧在虚拟机中入栈到出栈的过程。

局部变量表存分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(reference类型)。64位长度的long和double的数据会占2个局部变量空间，其余的占1个。所需内存在编译期分配。

2种异常情况：线程请求的栈深度大于虚拟机所允许的深度-StackOverFlowError；若虚拟机栈可以动态扩展，但扩展时申请不到足够内存-OutOfMemoryError

#### 本地方法栈

相似于虚拟机栈，区别是虚拟机栈为虚拟机执行Java方法服务，本地方法栈为虚拟机使用到的Native方法服务。同样会抛出StackOverFlowError，OutOfMemoryError。

#### Java堆

Java虚拟机管理的内存中最大的一块，唯一目的是存放对象实例。几乎所有的对象实例在这里分配内存。是所有线程共享的一块内存区域。

Java堆是垃圾收集器管理的主要区域，因此往往叫GC堆。

目前内存回收收集器基本采用分代收集算法，所以Java堆还可分为：新生代和老年代-Eden空间、From Survivor空间、To Survivor空间等。 线程共享的Java堆可划分出多个线程私有的分配缓冲区(TLAB)。划分的目的只为更高效地回收或分配内存。

Java堆可用于物理上不连续逻辑上连续的内存空间中。

实现时既可以实现成固定大小的，也可以是可扩展的。主流是可扩展，通过-Xmx和-Xms控制。

堆中没有内存完成实例分配，且无法扩展时，抛出OutOfMemoryError。

#### 方法区

与Java堆一样是所有线程共享的内存区域。它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据，存放Class的相关信息例如类名、访问修饰符、常量池、字段描述、方法描述等。别名为Non-Heap（非堆）、Hotspot：永久代Perm。

无法满足内存分配需求时，抛出OutOfMemoryError。

#### 运行时常量池

方法区的一部分，存放编译器生成的各种字面量和符号引用。无法申请内存时，抛出OutOfMemoryError.

#### 直接内存

### 对象的创建

虚拟机遇到一条new指令

#### ->@一

/\*

\* @一、首先执行类加载检查

\*/

1、检查这个指令的参数是否能在常量池中定位到一个类的符号引用

2、检查这个符号引用代表的类是否已被加载、解析和初始化过

3、if(没有)

执行相应的类加载过程。

#### ->@二 （#### @一end ####）

/\*

\* @二、为新生对象分配内存

\*/

对象所需内存的大小在类加载完成后便可完全确定，分配内存的任务等同于把一块确定大小的内存从java堆中划分出来。

指针碰撞法：假设java堆中内存绝对规整，用过的内存放一边，空闲的内存放另一边，中间放一个指针作为分界点的指示器，那所分配内存仅仅是把指针向空闲空间挪动一段与对象大小相等的距离。

空闲列表法：内存不规整，已使用的和空闲内存相互交错，虚拟机必须维护一张列表，记录哪些内存块可用，分配时从列表中找到一块足够大的空间划分给对象实例，并更新表。

选择哪种由java堆是否规整决定，而java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

/\*

\* 可能出现的问题

\*/

对象创建在虚拟机中很频繁，仅仅是修改一个指针指向的位置，在并发情况下也不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

解决方案：

1、对分配内存空间的动作进行同步处理—虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。

2、把内存分配动作按照线程划分在不同的空间之中进行，即每个线程在java堆中预先分配一小块内存，称为本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。

哪个线程需要分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并分配新的TLAB时，才需要同步锁定。虚拟机是否使用TLAB，可通过-XX:+/-UseTLAB参数来设定。

#### ->@三 #### @二 end ####

/\*

\* @三、将分配到的内存空间都初始化为零值

\*/

如果使用TLAB，则可以提前到TLAB分配时进行。

#### ->@四 #### @三 end ####

/\*

\* @四、对对象进行必要的设置

\*/

例如对象是哪个类的实例，如何找类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。

这些信息存在对象头当中。

===========================DONE=============================

这些步骤完成后从虚拟机视角一个新对象产生了，但从JAVA程序视角看对象创建才刚刚开始。----init方法还没执行，所有字段为0，所以，一般来说执行new指令后会解着执行init方法，才算真正创建对象。

### 对象的内存布局和访问

对象在内存中的布局，分为3个部分：对象头、实例数据、对齐填充。

1. 对象头存储2部分。一部分为对象自身运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等。这部分数据长度为32或64位，被成为Mark Word。另一部分为类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针确定对象是哪个类的实例。若对象是个java数组，则还必须有一块用于记录数组长度的数据。虚拟机可以确定普通对象的大小但不能确定数组的。
2. 实例数据部分是程序代码中所定义的各种类型的字段内容，包括父类和子类中的。存储顺序由虚拟机分配策略参数和字段在java源码中定义顺序决定。
3. 对其填充并不必然存在，对象起始地址必须保持8字节的整数倍，对象大小也是8字节整数倍，不够的就填充。

对象的访问，java程序通过栈上的reference数据来操作堆上的具体对象，访问方式取决于虚拟机实现，主流方式有使用句柄和直接指针两种。

1. 句柄方式，java堆中划分出一块内存作为句柄池，reference中存储对象的句柄地址，句柄中包含了对象实例数据与类型数据各自的地址信息。最大好处是reference中存储的是稳定的句柄地址，对象移动时只会改变句柄中的实例数据指针。
2. 直接指针访问，reference中存储的就是对象地址，最大好处是速度更快，节省了一次指针定位的时间开销。Hotspot用的此方法。

### 实战：OutOfMemoryError（OOM）异常

#### Java堆溢出

1、-Xms参数值与-Xmx设置堆大小，一样则可以避免堆自动扩展。

2、-XX:+HeapDumpOnOutOfMemoryError可以让虚拟机在内存溢出异常时Dump出当前内存堆转储快照。

3、不断创建对象，并保证GC Roots到对象之间有可达路径来避免垃圾回收机制清楚这些对象，可导致内存溢出。

4、简单思路-内存映像分析工具（如Eclipse Memory Analyzer），分析是溢出还是泄露

泄露：通过工具查看泄露对象到GC Roots的引用链，于是就能找到泄露对象是通过怎样的路径与GC Roots相关联并导致垃圾收集器无法自动回收，从而精准定位泄露代码的位置。

不存在泄露：检查虚拟机的堆参数-Xmx,-Xms，看是否可以调大，并检查代码尝试减小一些生命周期和持有状态时间过长的对象。

#### 虚拟机栈和本地方法栈溢出

1、-Xoss设置本地方法栈大小但无效。栈容量只能由-Xss设定。

2、事实上SOF和OOM是同一个异常的2种表述。

3、使用-Xss减小栈容量和定义大量的本地变量增大此方法帧中本地变量表的长度，抛出StackOverflowError（SOF）。

4、不断建立多线程可导致OOM，但这种溢出异常与栈空间是否足够大没关系。

Windows给进程分配的内存有限制，比如2G，虚拟机提供了JAVA堆和方法区这两部分的最大值，剩余内存为2G-Xmx-MaxPermSize，忽略程序计数器和虚拟机，剩下内存归虚拟机栈和本地方法栈，每个线程分配到的栈越大，可以建立的线程数就越少，特别容易耗尽内存，开发多线程时应该注意。

所以不能减少线程数或更换64位虚拟机时可以通过减少最大堆和减小栈容量来换取更多的线程。

5、出现SOF时有错误堆栈可以阅读，比较容易找到问题的所在。

#### 方法区和运行时常量池溢出

1、String.intern()是一个Native方法，作用：如果字符串常量池已包含此String对象的字符串，则返回代表池中这个字符串的String对象；否则，添加到常量池中，并返回此String对象的引用。对于运行时常量池可用无限.intern()并添加到list中以防倍回收。

2、JDK1.6前，常量池在永久代内，可通过-XX:PermSize和-XX:MaxPermSize限制方法区大小，从而间接限制其中常量池容量。JDK1.7中分离了字符串常量池。

3、对于方法区，运行时产生大量的类去填满，直至溢出。借助工具CGLib。

4、方法区溢出是一种常见的溢出异常，类被回收的判定很严格，经常动态生成大量类的应用中要特别注意回收状况。

#### 本机直接内存溢出

1、通过-XX:MaxDirectMemorySize指定DirectMemory，不指定默认为堆的最大值

2、申请分配内存的方法是unsafe.allocateMemory()。

3、直接内存导致的溢出的特征是在Heap Dump文件中不会看到明显的异常。OOM后Dmup文件很小而程序中直接或间接使用了NIO，可能就是这种异常。

=========================第二章DONE=========================

## 第三章 垃圾收集器与内存分配策略

GC要完成的3件事：哪些内存需要回收？什么时候回收？如何回收？

栈中方法结束或线程结束时内存就自然跟着回收了。Java堆和方法区则不一样，一个接口中的多个实现来需要的内存可能不一样，一个方法中的多个分支需要的内存可能不一样，只有程序运行期间才能知道会创建哪些对象，这部分内存的分配和回收是动态的，GC关注的是这部分内存。

### 对象的生死

在java堆中存放着几乎所有对象实例，GC在对堆回收前要先判断哪些还“活着”，哪些已经“死去”（不可能再被任何途径引用的对象）。判断对象存活都与“引用”有关。

#### 判断法-引用计数器法

**原理：**给对象添加一个引用计数器，每当被引用时加1，失效时减1，任何时刻计数器为0的对象是不可能再被引用的。

**好处：**实现简单，判断效率高，被广泛使用。

**缺陷：**主流java虚拟机不使用，主要原因是它很难解决对象之间相互循环引用的问题。如objA.instance=objB,objB.instance=objA，且objAobjB不可能再被访问

#### 判断法-可达性分析算法

**原理：**通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到GC Roots没有任何引用链相连（不可达）时，则此对象不可用。

**可作为GC Roots的对象：**

* 虚拟机栈中引用的对象。
* 方法区中类静态属性引用的对象。
* 方法区中常量引用的对象。
* 本地方法栈中JNI（即Native方法）引用的对象。

#### 引用

JDK1.2之前：如果reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表一个引用。这种定义过于单调，无法描述“食之无味，弃之可惜”的对象。

JDK1.2之后，将引用分为强引用、软引用、弱引用、虚引用，强度依次减弱。

* 强引用：普遍存在，类似Object obj = new Object()，只要强引用还存在，GC就无法回收它。
* 软引用：描述一些还有用但非必须的对象。系统快发生溢出前，会把这些对象列进回收范围之中进行第二次回收。SoftReference类来实现。
* 弱引用：描述非必须对象，此类对象只能生存到下一次垃圾回收发生之前，无论内存是否够，都会回收。WeakReference类实现。
* 虚引用：唯一目的是能在这个对象被收集器回收时收到一个系统通知，不会影响对象生存时间也不会获得对象实例。PhantomReference类实现。

#### 生存还是死亡- finalize()方法

可达性分析算法中不可达的对象也并非“非死不可”。真正宣告一个死亡至少要经历两次标记过程：

->可达性分析后发现没有GC Roots相连的引用链时标记第一次，

->同时进行筛选，条件是此对象是否有必要执行finalize()方法，若没有覆盖或此方法已被虚拟机调用过，则为没有必要执行，则被回收。如果有必要执行，对象将进入F-Queue队列中，稍后由一个虚拟机自动建立、低优先级的Finalizer线程去执行（触发）它。（并不承诺运行完，有可能导致阻塞甚至导致内存回收系统崩溃）。稍后GC对F-Queue队列中的对象进行第二次小规模的标记，finalize()方法是最后一次机会逃离死亡的机会，只需要重新关联引用链上任一对象，譬如把自己this赋值给某个类变量或对象的成员变量，第二次标记时将被移出“即将回收”集合。否则就被回收。

**小结：**1、对象可以在被GC时自救。2、自救机会只有一次，因为finalize()最多只会被系统自动调用一次。

自救的代码实现见代码清单3-2.

不建议自救，运行代价高，用try-finally能更及时更好。

#### 回收方法区

回收性价比低，主要回收2部分内容：废弃常量和无用的类。

**废弃常量**回收和堆中对象回收相似，判断是否是废弃常量也比较简单。

**无用的类**满足3个条件：

* 该类所有实例已被回收
* 加载该类的ClassLoader已被回收
* 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

是否对类进行回收需通过参数设置。-Xnoclassgc控制，-verbose:class和-XX:+TraceClassLoading、-XX:+TraceClassUnLoading查看类加载和卸载信息。

**应用场景：**在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP、OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不溢出。

## 名词总结

虚拟机是什么？

GC堆：Garbage Collected Heap。

TLAB：本地线程分配缓冲

张东东is big pig”s foot