# 第2章 Java内存区域与内存溢出异常

在C/C++与Java之间存在着一道由内存动态分配和垃圾收集技术搭建而成的“高墙”，墙外的人想要进去，墙里的人却想出来。

## 2.1概述

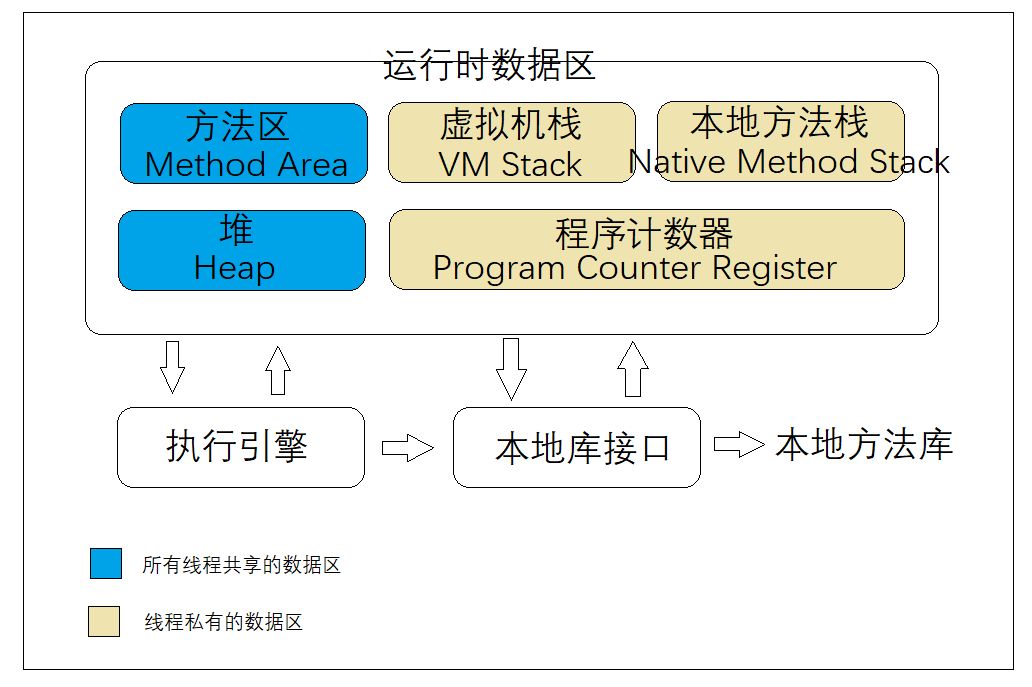
对于从事C、C++程序开发工作的开发人员来说，在内存管理方面，他们既是拥有最高权力的“皇帝”，又是从事最基础工作的“劳动人民”——既拥有每一个对象的“所有权”，又负责着每一个对象生命从开始到终结的维护责任。

对于Java程序员来说，在虚拟机自动内存管理机制的辅助下，不再需要为每一个new操作去写匹配的delete/free代码，不容易出现内存泄漏和内存溢出的问。由虚拟机来负责内存管理看似十分完美，不过也正是由于Java程序员将内存控制的权利交给了Java虚拟机，一旦出现内存泄漏和内存溢出的问题，如果不了解Java虚拟机是怎样使用内存的，那么排除错误将会成为一项非常艰难的工作。

## 2.2运行时数据区域

Java虚拟机在执行java程序的过程中，会将它管理的内存划分为若干个不同的数据区域。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有的区域则依赖用户线程的启动和结束而建立和销毁。

根据《Java虚拟机规范》的规定，Java虚拟机所管理的内存将会包含以下几个运行时数据区域：



### 2.2.1程序计数器

程序计数器（Program Counter Register）是一块较小的**内存空间**，它可以看作是**当前程序所执行的字节码的行号指示器**。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更加高效的方式去实现），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、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于**Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式实现的**，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，**为了线程切换后能恢复到正确的执行位置，每条线程都需要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，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指向的地址；如果正在执行的是Native方法，这个计数器值为空（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。

### 2.2.2虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是**线程私有**的，它的生命周期与线程相同，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（Stack Frame）**用于存储局部变量表、操作数栈、动态链接、方法出口等信息。**每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈和出栈的过程。**

在Java虚拟机规范中，对虚拟机栈规定了两种异常情况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩容（当前大多数Java虚拟机都可以动态扩容），如果在扩容的过程中申请不到足够的内存，就会抛出OutOfMemoryError异常。

### 2.2.3本地方法栈

本地方法栈（Native Method Stack）与虚拟机栈发挥的作用是非常相似的，它们之间的区别不过是**虚拟机栈为虚拟机执行Java方法（也就是字节码）服务**，**而本地方法栈则为虚拟机使用到的Native方法服务**。在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot）直接就将本地方法栈和虚拟机方法栈合二为一。

与虚拟机栈一样，本地方法栈也是线程私有的，本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

### 2.2.4Java堆

对于大多数应用来说，Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块。Java堆是被所有**线程共享**的一块内存区域，在虚拟机启动时创建。此内存区域的唯一作用就是**存放对象实例**，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配（The heap is the runtime data area from which memory for all class instances and arrays is allocated），但是随着JIT编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也逐渐变得不是那么绝对了。

Java堆是垃圾收集器管理的主要区域，因此很多时候Java堆也被称为“GC堆”（Garbage Collected Heap，幸好国内没翻译称“垃圾堆”）。

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是可拓展的，不过目前主流的虚拟机都是按照可拓展来实现的，如果在堆中没有内存完成实例分配，并且堆也无法再扩容时，将会抛出OutOfMemoryError异常。

### 2.2.5方法区

方法区（Method Area）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。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做Non-Heap（非堆），目的应该是与Java堆区分开来。

### 2.2.6运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。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（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池。

运行时常量池相对于Class文件中的常量池的一个重要特征就是具有动态性，Java语言并不要求常量一定只有编译期才能产生，也就是说并非只有预置入Class文件的常量池中的内容才能进入方法区运行时常量池，运行期间可可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String类的intern()方法。

既然运行常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError异常。

### 2.2.7直接内存

## 2.3HotSpot虚拟机对象探秘

本小节将介绍HotSpot虚拟机中对象分配、布局和访问的全过程。

### 2.3.1对象创建

Java是一门面向对象的语言，在Java程序的运行过程中无时无刻都有新的对象被创建出来。从语言的层面看，创建对象通常仅仅是一个new关键字而已，从虚拟机的角度看，创建对象又是怎样的一个过程呢？

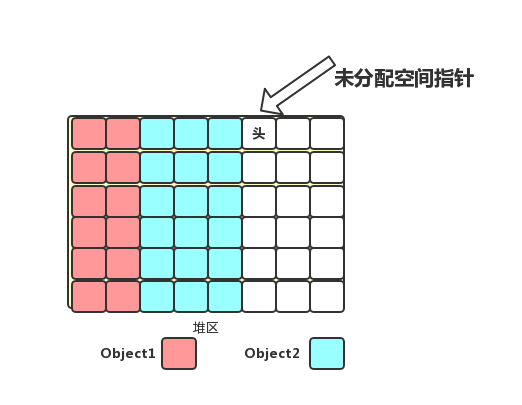
虚拟机遇到一条new指令时，首先将会去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，必须想执行相应的类加载过程。

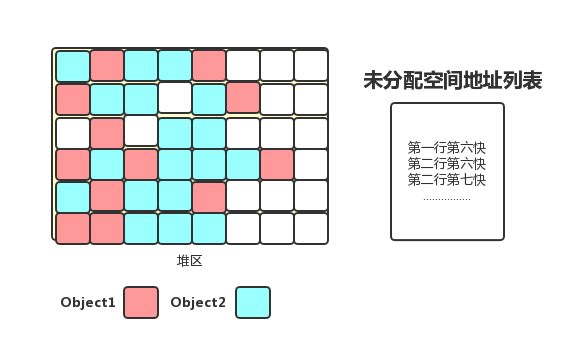
在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定，为对象分配空间其实就是把一块大小确定的内存从java堆中划分出来。

假设Java堆中内存是绝对规整的，所有用过的内存放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那分配内存的工作其实就是将这个指针向空闲空间挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”。

如果Java堆中的内存并不是规整的，已使用的内存和未使用的内存交错存在，那就没有办法进行简单的指针碰撞了，虚拟机必须维护一个列表，记录哪些内存是尚未使用的、可分配的，在分配内存的时候从列表中找到一块足够大的空闲空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”

具体使用哪种方式分配内存由Java堆是否规整决定，而Java堆是否规整又由采用的垃圾收集器是否带有压缩整理功能决定。





除了如何划分可用空间，另外还需要注意空间划分时的线程安全问题。对象的创建在虚拟机中是非常频繁的，以较为简单的“指针碰撞”为例，即使只是简单的修改指针指向的位置，在在并发情况下一样存在线程安全问题，比如说正在给A分配内存，指针还没来得及修改，对象B也请求分配内存，也需要修改指针。

上述问题有两个解决方案：

方案一 对分配内存的操作进行同步处理——实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。

方案二 把分配内存的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）。哪个线程要分配内存，就在哪个线程的TLAB上分配，只有TLAB用完并需要分配新的TLAB的时候，才需要进行同步锁定。虚拟机是否使用TLAB，可以通过-XX:+/-UseTLAB参数来设定。

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头，因为对象头中要保存其他信息），如果使用TLAB，这一工作可以提前至TLAB分配的时候进行。这一操作保证了对象的实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头（Object Header）之中。根据虚拟机当前运行状体的不同，如是否启动偏向锁等，对象头会有不同的设置方式。

上面的工作完成之后，从虚拟机的视角来看，一个新的对象就已经产生了，但从Java程序的视角看，对象的创建才刚刚开始——<init>方法还没有执行，所有的字段还是零值。所以，一般来说（由字节码中是否跟随invoke special指令所决定），执行new指令之后会接着执行<init>方法，把对象按照程序员的意愿进行初始化，这样一个真正的对象才算产生出来了。.

总结对象的创建过程：

**类加载检查**→**对象内存分配**→**各字段赋零值**→**对象头信息设置**→**初始化**

### 2.3.2