深入理解Java虚拟机

Chapter2 Java内存区域于内存溢出异常

在java虚拟机自动内存管理机制的帮助下，我们不需要为每一个new操作去写配对的delete/free代码，不容易出现内存泄漏和溢出的问题。

虚拟机运行Java时数据区域

每个区域的用途，创建和销毁的时间不同。

- Method Area
- VM Stack
- Native Method Stack
- Heap
- Program Counter Register

其中Method Area和Heap是所有线程共享的，其他三个是线程独有的

1. Program Counter

一块较小的内存区域，字节码的行号指示器

Java虚拟机的多线程：线程轮流切换并分配处理器执行时间

一个时刻一个处理器只能执行一条线程，每个线程保留一个单独的PC（线程私有内存）

当该处理器在执行Java Method，PC指定的是虚拟机字节码指令的地址

当该处理器执行的是Native Method（java调用非java代码的接口），PC的值为空（Undefined）

该内存区域是JAVA虚拟机中唯一没有规定OutOfMemoryError的区域

2. Java VM Stack

线程私有的栈，描述Java方法执行的内存模型

Stack Frame: 存储着局部变量表（存放着编译期克制的各种基本数据类型，对象引用，返回地址），操作数栈，动态链接，方法出口

局部变量表的大小在编译的时候已经被确定下来

VM stack存储着这些帧，在每个线程中调用方法就是在VM stack中压入或弹出Stack Frame的过程

两种异常：

- StackOverflowError

When a function call is invoked by a Java application, a stack frame is allocated on the call stack. The stack frame contains the parameters of the invoked method, its local parameters, and the return address of the method. If there is no space for a new stack frame then, the `StackOverflowError` is thrown by the Java Virtual Machine (JVM).

- OutOfMemoryError

  当虚拟机栈可以倍动态扩展的时候，当扩展时不能申请到足够的内存，就会抛出OutOfMemoryError异常

3. Native Method Stack

类似VM Stack，为虚拟机使用到的Native方法服务

不同的虚拟机中VM Stack的实现方法不同

也会抛出StackOverflowError和OutOfMemoryError

在Sun Hotspot虚拟机中把虚拟机栈和本地方法栈合为一体

4. Java堆

通常是Java虚拟机锁管理的内存中最大的一块，被所有线程共享的内存区域

在虚拟机启动的时候创建Java堆，用来存放对象实例，几乎所有对象实例都在这里分配内存

但随着JIT编译器和逃逸分析技术的出现，栈上分配、标量替换优化技术使得所有对象在堆上被分配也渐渐不那么绝对了

垃圾收集器管理的主要区域，“GC堆”

现在基本使用分代收集算法，Java堆分为新生代和老年代

Java堆可以存放在物理上不连续的内存空间中，只要逻辑上是连续就可以

目前都是可扩展的，当堆中没有内存完成实例分配，并且对无法扩展时，抛出OutOfMemoryError

小知识补充时间：一定要在堆上分配对象么？

a. JIT编译器

在主流商用JVM（HotSpot、J9）中，Java程序一开始是通过解释器（Interpreter）进行解释执行的。当JVM发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“热点代码（Hot Spot Code）”，然后JVM会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为：即时编译器（Just In Time Compiler，JIT）

JIT编译器是“动态编译器”的一种，相对的“静态编译器”则是指的比如：C/C++的编译器

b. 逃逸分析技术

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，称为方法逃逸。甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为线程逃逸

只发生在JIT的即时分析中

c. 栈上分配

栈上分配就是把方法中的变量和对象分配到栈上，方法执行完后自动销毁，而不需要垃圾回收的介入，从而提高系统性能（如果对象不发生逃逸，则将堆分配转换成栈分配，从而降低垃圾收集器运行的频率）

d.标量替换

标量和聚合量

标量即不可被进一步分解的量，而JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量。

替换过程

通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间

5. 方法区

Method Area：各线程共享的内存区域，被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

Hotspot使用永久代来实现方法区

可以不实现垃圾分类，该区域的内存回收目标主要是针对常量池的回收和类型的卸载

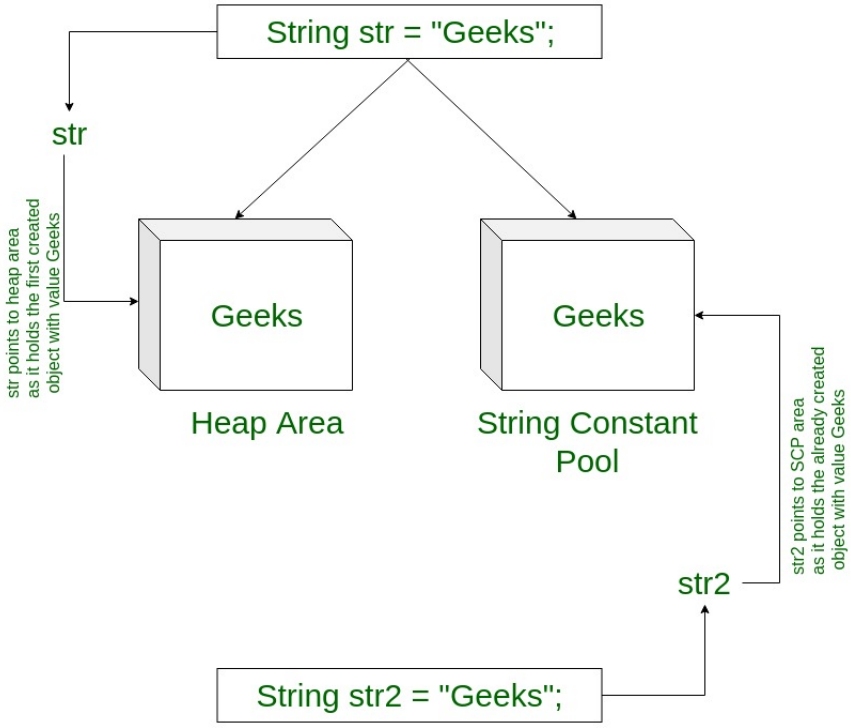
当方法区 无法看足内存分配需求--> OutofMemoryError

Runtime Constant Pool,方法区的一部分，Class文件中定义了一项信息：Constant Pool Table，在类加载后进入方法区在Runtime Constant Pool中存放

运行时常量池比Class文件常量池中另外一个重要特征是具备动态性，Java并不要求常量只能在编译期产生，这意味着并非预置入Class文件中常量池的内容才能进入方法区Runtime Constant Pool，运行时也可以将新的变量放入池中，用的比较多的是String类的intern()方法

Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.

By applying String.intern() on a couple of strings will ensure that all strings having the same contents share the same memory



6. 直接内存

Java应用程序执行时会启动一个Java进程，这个进程的用户地址空间可以被分成两份：JVM数据区 + direct memory

通俗的说，JVM数据区就是Java代码可以直接操作的那部分内存，由heap/stack/pc/method area等组成，GC也工作在这一片区域里

direct memory则是额外划分出来的一段内存区域，无法用Java代码直接操作，GC无法直接控制direct memory，全靠手工维护

direct memory更加适合IO操作

heap memory不可能直接用于系统IO，数据只能先读到direct memory里去，然后再复制到heap memory

HotSpot虚拟机在Java堆中的对象分配

这里讨论的是创建普通java对象，不包括数组和Class对象

1. 检查new指令的参数（类名）是否能在常量池中定位到符号引用，并检查这个符号引用代表的类是否被夹在、解析和初始化过，如果没有就执行对应的类加载代码

常量池主要用于存放两大类常量：**字面量**(Literal)和**符号引用量**(Symbolic References)，字面量相当于Java语言层面常量的概念，如文本字符串，声明为final的常量值等，符号引用则属于编译原理方面的概念，包括了如下三种类型的常量：类和接口的全限定名、字段名称和描述符和方法名称和描述符

2. 为新生对象分配内存

   从Java堆中分配内存，内存大小在类加载以后就可被完全确定

用过的内存放在一边，空闲的内存放在另一边，中间用指针（指针碰撞的内存分配策略）

内存不规整，已使用的内存和未使用的内存交错，使用列表记录哪些内存块可以被使用（Free List）

内存块是否规整取决于垃圾收集器是否带有压缩整理功能：有-->指针碰撞 没有--> Free List

3. 考虑内存分配并发冲突

   正在给对象A分配内存，还没修改指针，对象B使用了原来的指针（CAS和失败重试；TLAB给每个线程分配本地线程分配缓存）

4. 内存空间初始化为0
5. 设置Object Header（这个对象是哪个类的实例，怎样找到类的元数据，对象的Hash Code，对象的GC分代年龄）
6. 实现类的init方法（取决于字节码中是否包含invoke special）

Hotspot中对象在Java堆中的内存存储布局

对象在内存中存储的布局可分为3块区域：Header，Instance Data和Padding

1. **Header（与对象自身定义的数据无关的额外存储成本）**
   - 存储对象自身的运行时数据（Hash Code），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，Mark Word，在32位虚拟机和64位虚拟机中分别为32位和64位
   - 类型指针（对象指向他的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例），不是每个虚拟机都通过在对象数据上保留类型指针来查找对象的元数据信息的
   - 如果对象是数组的话，还需要在对象头部记录数组长度，因为数据的元数据中虚拟机无法确定数据的大小
2. **Instance Data**

   对象真实存储的有效信息，程序代码中定义的各种类型的字段内容（父类和子类），存储数据顺序收到虚拟机分配策略参数影响，Hotspot把相同宽度的字段分配到一起

3. Padding

当对象实例数据部分没有对齐的时候，需要通过对齐填充来补全

Hotspot中对象在Java堆中的访问定位

使用对象时，我们需要通过stack伤的reference数据来操作堆上的具体对象。如何使用reference来定位、访问堆中的对象由虚拟机来决定，目前主要有使用句柄和直接指针两种

- 直接句柄

  Java堆中划分一块内存作为句柄池，reference里面存着对象的句柄地址，而句柄包含了对象实例数据（java堆中的实例池）和类型数据（方法区中的对象类型数据）各自的地址信息

  优点：在对象被移动的时候（垃圾收集时移动对象）不需要改变reference，只需要改变句柄的实例数据指针

- 直接指针

reference中存放的事Java堆中的对象实例数据的地址，在对象实例数据中存放着到方法区中对应的的对象类型数据的地址

优点：速度更快，减少了一次指针定位的时间开销。在Java中对象访问非常频繁，这类开销积少成多也是一种很可观的执行成本（Sun Hotspot采用这一种）