Java 内存划分

我们常说的 Java 内存管理就是指这块区域的内存分配和回收，那么，这块儿区域具体是怎么划分的呢？

根据《Java 虚拟机规范》的规定，运行时数据区通常包括这几个部分：

程序计数器(ProgramCounter Register)

Java 栈(VM Stack)

本地方法栈(Native MethodStack)

方法区(Method Area)

堆(Heap)

Java 堆和方法区是所有线程共享（所有执行引擎可访问）；

【Java 堆】用于存储 **Java 对象**，每个 Java 对象都是这个对象类的副本，会复制包含继承自它父类的所有非静态属性。

【方法区】用于存储**类结构**信息，class 文件加载进 JVM 时会被解析成 JVM 识别的几个部分分别存储在不同的数据结构中：常量池、域、方法数据、方法体、构造函数，包括类中的方法、实例初始化、接口初始化等。

方法区被 JVM 的 GC 回收器管理，但是比较稳定，并没有那么频繁的被 GC 回收。

java 栈和 PC 寄存器是线程私有，每个执行引擎启动时都会创建自己的 java 栈和 PC 寄存器；

【Java 栈】和线程关联，每个线程创建的时候，JVM 都会为他分配一个对应的 Java 栈，这个栈含有多个栈帧；栈帧则是个方法关联，每个方法的运行都会创建一个自己的栈帧，含有内存变量，操作栈、方法返回值。

（用于存储方法参数、局部变量、方法返回值和运算中间结果）

【PC 寄存器】则用于记录下一条要执行的字节码指令地址和被中断。如果方法是 native 的，程序计数器寄存器的值不会被定义为空。

【本地方法栈】是为 JVM 运行 Native 方法准备的空间，类似于 Java 栈。

【运行时常量池】关于这个东西要明白三个概念：

- **常量池**（Constant Pool）：常量池数据编译期被确定，是 Class 文件中的一部分。存储了类、方法、接口等中的常量，当然也包括字符串常量。
- **字符串池/字符串常量池**（String Pool/String Constant Pool）：是常量池中的一部分，存储编译期类中产生的字符串类型数据。
- **运行时常量池**（Runtime Constant Pool）：方法区的一部分，所有线程共享。虚拟机加载 Class 后把常量池中的数据放入到运行时常量池。

具体堆里的内存结构，不再详细展开描述。另外虚拟机版本不同可能有细微差别。

常见的 OOM 异常分析

一．java.lang.OutOfMemoryError:Java heap space

由于 Heap 是用来存放实例的，堆溢出，也就说明了当前的实例对象过多，而且这些对象一直处于存活状态（JVM 判断对象是否存活，是通过判断 GC Roots 和对象之间的是否存在可达路径）。出现这种问题，一般要考虑下列两种情况：

- 内存泄露：

一般出现这种情形，需要判断是否是内存泄露，即一些无用对象一直被引用，导致 GC 无法有效回收它，这时可以通过一些工具，查看 Heap dump，看看 GC roots 到对象之间的引用链，定位到泄露的对象。

- 内存溢出：

另一种情形就是内存溢出，也就是这些对象的确是需要存活的，因此也就不存在 GC 回收异常。一般这种情况，可能是对象过大，或者对象的生命周期过长，需要从业务层面，减少这些对象在运行期的内存消耗。
另外一种情形，也有可能是我们的堆分配的内存过小，可以通过配置堆的参数（-Xmx 最大内存与-Xms 最小内存）来设置

二．java.lang.StackOverflowError

栈用来存储线程的局部变量表、操作数栈、动态链接、方法出口等信息。如果请求栈的深度不足时抛出的错误会包含类似下面的信息：

java.lang.StackOverflowError

解决办法，首先排查代码有问题例如递归没有出来，或者是方法调用层级太深等。
如果确认代码没有问题，可以通过 jvm -XSS 参数调大。
-Xss：每个线程的堆栈大小,JDK5.0 以后每个线程堆栈大小为 1M。

三．java.lang.OutofMemoryError: unable to create new native thread

默认情况下，jvm 每个线程栈占用 1M 内存，如果建立的线程过多，导致系统内存不够，就会报这个 OOM 异常。

常见的排查及解决办法：

1. 排查应用是否创建了过多的线程

通过 jstack 确定应用创建了多少线程？超量创建的线程的堆栈信息是怎样的？谁创建了这些线程？一旦明确了这些问题，便很容易解决。

2. 调整操作系统线程数阈值

操作系统会限制进程允许创建的线程数，使用 ulimit -u 命令查看限制。某些服务器上此阈值设置的过小，比如 1024。一旦应用创建超过 1024 个线程，就会遇到 java.lang.OutOfMemoryError: unable to create new native thread 问题。如果是这种情况，可以调大操作系统线程数阈值。

3. 增加机器内存

如果上述两项未能排除问题，可能是正常增长的业务确实需要更多内存来创建更多线程。如果是这种情况，增加机器内存。

4. 减小堆内存

一个老司机也经常忽略的非常重要的知识点：线程不在堆内存上创建，线程在堆内存之外的内存上创建。所以如果分配了堆内存之后只剩下很少的可用内存，依然可能遇到 java.lang.OutOfMemoryError: unable to create new native thread。考虑如下场景：系统总内存 6G，堆内存分配了 5G，永久代 512M。在这种情况下，JVM 占用了 5.5G 内存，系统进程、其他用户进程和线程将共用剩下的 0.5G 内存，很有可能没有足够的可用内存创建新的线程。如果是这种情况，考虑减小堆内存。

5. 减少进程数

这和减小堆内存原理相似。考虑如下场景：系统总内存 32G，java 进程数 5 个，每个进程的堆内存 6G。在这种情况下，java 进程总共占用 30G 内存，仅剩下 2G 内存用于系统进程、其他用户进程和线程，很有可能没有足够的可用内存创建新的线程。如果是这种情况，考虑减少每台机器上的进程数。

6. 减小线程栈大小

线程会占用内存，如果每个线程都占用更多内存，整体上将消耗更多的内存。每个线程默认占用内存大小取决于 JVM 实现。可以利用-Xss 参数限制线程内存大小，降低总内存消耗。例如，JVM 默认每个线程占用 1M 内存，应用有 500 个线程，那么将消耗 500M 内存空间。如果实际上 256K 内存足够线程正常运行，配置-Xss256k，那么 500 个线程将只需要消耗 125M 内存。

四．java.lang.OutofMemoryError: PermGen space

方法区主要存储被虚拟机加载的类信息，如类名、访问修饰符、常量池、字段描述、方法描述等。理论上在 JVM 启动后该区域大小应该比较稳定，但是目前很多框架，比如 Spring 和 Hibernate 等在运行过程中都会动态生成类，因此也存在 OOM 的风险。
如果该区域 OOM，错误结果会包含类似下面的信息：

java.lang.OutofMemoryError: PermGen space

JDK1.6 及以前版本，常量池在方法区内，存放的主要是编译器生成的各种字面量和符号引用，但是运行期间也可能将新的常量放入池中，比如 String 类的 intern 方法。
如果该区域 OOM，错误结果会包含类似下面的信息：
java.lang.OutofMemoryError: PermGen space

五． Java.lang.OutOfMemeoryError:GC overhead limit exceeded

如上异常，即程序在垃圾回收上花费了 98%的时间，却收集不回 2%的空间，通常这样的异常伴随着 CPU 的冲高