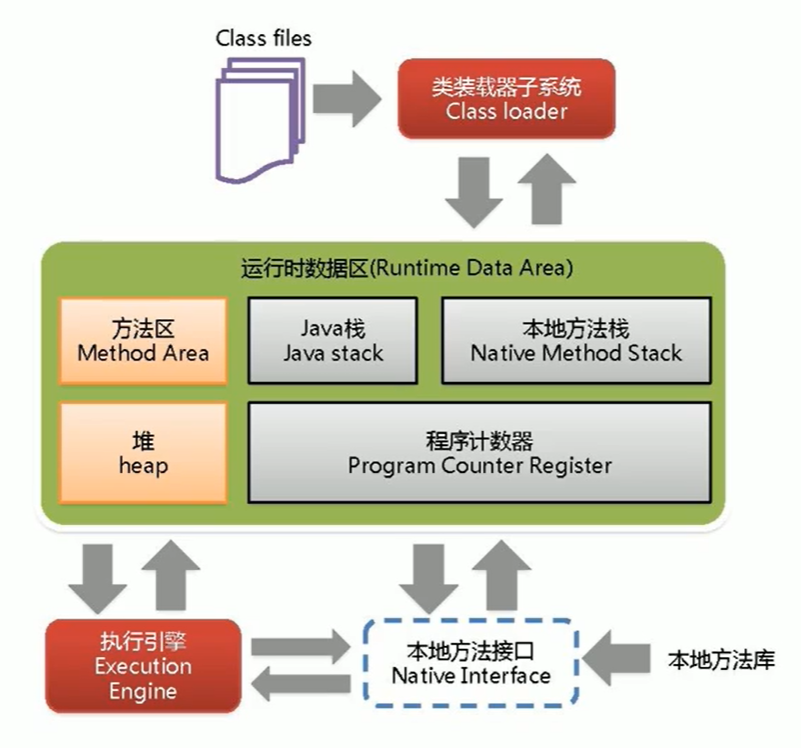
**JVM笔记**

# JVM内存区域与内存溢出异常



## 运行时数据区

上面图片

灰色区域为线程私有（Java栈、本地方法栈、程序计数器）

亮色部分为公有（方法区、堆）

### 程序计数器

*Program Counter Register*

这是一块较小的内存空间，可以看作是当前线程所执行字节码的行号指示器（指向下一条即将被执行的字节码）。

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，例如分支、循环、跳转异常处理、线程恢复等基础功能都靠它完成。

如果执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址。

如果执行的是本地（native）方法，这个计数器的值应该为空（Undefined）。

注：这块区域是唯一一个不会出现OutOfMemoryError的地方。

### Java虚拟机栈

*Java Virtual Machine Stack*

虚拟机栈描述的是Java方法执行的线程内存模型：

每个方法被执行的时候，Java虚拟机都会同步创建一个栈帧（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。

每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

栈帧：是方法运行期很重要的数据结构

我们关注的点一般就是栈中存放的局部变量表，存放了

1. 8中基本数据类型
2. 对象引用
3. returnAddress类型（指向了一条字节码指令的地址，即实例方法）

这块区域可能出现俩种异常：StackOverflowError和OutOfMemoryError

### 本地方法栈

Native Method Stack

与虚拟机栈非常相似，区别是：

虚拟机栈为Java方法（也就是字节码）服务

本地方法栈为本地方法服务

可能出现的异常也是：StackOverflowError和OutOfMemoryError

### Java堆

Java Heap

是虚拟机管理内存中最大的一块，所有线程共享，虚拟机启动时创建。

此区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

可扩展（-Xms和-Xmx）

可分为新生代、老年代、元空间（java7是永久代）

S1（1/10）

S0（1/10）

Eden（8/10）

新生代（1/3） 老年代（2/3）

### 方法区

Method Area

线程共享，用于存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

虽然此区域划被描述为堆的一个逻辑部分，但他却有着一个别名叫“非堆”（Non-Heap），目的是与堆区分开。

这里不得不提“永久代(JDK8以前)“，当时的HotSpot将分代收集设计扩展至方法区，或者说用永久代实现的方法区，这样使得HotSpot的垃圾收集器能够就像管理堆一样管理这部分内存，省去了专门为方法区编写内存管理代码的工作。

Java8将永久代替换为在本地内存中实现的元空间（metaspace）

《Java虚拟机规范》对方法区的规定十分宽松，除了和Java堆一样不需要连续的内存、可扩展（-XX：Metaspace）之外，甚至可以选择不实现垃圾回收。

相对而言，垃圾收集行为在这个区域确实很少出现。方法区的垃圾回收主要针对常量池的回收和对类型的卸载，条件相当苛刻。

可能出现的异常：OOM

## 补充

### 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表（Constant Pool Table），用于存放编译期生成的各种字面量与符号引用，这部分内容将在加载后存放到方法区的运行时常量池中。

Java虚拟机对于Class文件的每一个部分（自然也包括常量池）的格式有着严格的要求，例如每一个字节用于存储哪种数据都必须符合规范上的要求才会被JVM认可、加载和执行，但对于运行时常量池，并没有要求。一般来说，除了保存Class文件中描述的符号引用外，还会把由符号引用翻译过来的直接引用也存储在运行时常量池中。

运行时常量池相对于Class文件常量池的另外一个特征是具备动态性，Java语言并不要求常量一定只有编译期才能产生，也就是说，并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可以将新的常量放入常量池中，这种特性被开发人员利用的比较多的便是String类的intern（）方法。

可能的异常：OOM

### 直接内存

*Direct Memory*

并不是JVM的一部分。

但是这部分内存也被频繁使用，也肯能报OOM异常。

JDK1.4加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O方式，它可以利用Native函数库直接分配堆外内存，然后通过一个存储在Java堆里面的DirectByteBuffer对象作为这块内存的引用进行操作。这样能在一些场景中显著地提高性能，因为避免了在Java堆和native堆中来回复制数据。

可能地异常：OOM