# JVM

线程私有的:

* 程序计数器
* 虚拟机栈
* 本地方法栈

线程共享的:

* 堆
* 方法区
* 直接内存(非运行时数据区的一部分)

## 程序计数器

程序计数器是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, **程序计数器是唯一一个不会出现 OutOfMemoryError 的内存区域**, 它的生命周期随着线程创建和结束.

程序计数器主要有两个作用:

1. 字节码解释器通过改变程序计数器来依次读取指令, 从而**实现代码的流程控制**, 如:顺序执行, 选择, 循环, 异常处理.
2. 在多线程的情况下, 程序计数器用于记录当前线程执行的位置, 从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了.

## Java 虚拟机栈

与程序计数器一样, Java 虚拟机栈也是线程私有的, 它的生命周期和线程相同, 描述的是 Java 方法执行的内存模型, 每次方法调用的数据都是通过栈传递的, Java 虚拟机栈是由一个个栈帧组成, 而每个栈帧中都拥有: 局部变量表, 操作数栈, 动态链接, 方法出口信息.

局部变量表主要存放了编译器可知的各种数据类型 (boolean, byte, char, short, int, float, long, double) , 对象引用 (reference 类型, 它不同于对象本身, 可能是一个指向对象起始地址的引用指针, 也可能是指向一个代表对象的句柄或其他与此对象相关的位置) .

Java 虚拟机栈会出现两种异常:StackOverFlowError 和 OutOfMemoryError.

1. StackOverFlowError: 若 Java 虚拟机栈的内存大小不允许动态扩展, 那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候, 就抛出 StackOverFlowError 异常.
2. OutOfMemoryError: 若 Java 虚拟机栈的内存大小允许动态扩展, 且当线程请求栈时内存用完了, 无法再动态扩展了, 此时抛出 OutOfMemoryError 异常.

Java 栈可用类比数据结构中栈, Java 栈中保存的主要内容是栈帧, 每一次函数调用都会有一个对应的栈帧被压入 Java 栈, 每一个函数调用结束后 (return 语句, 抛出异常), 都会有一个栈帧被弹出. 每次报错的就是相应的方法栈.

### 本地方法栈

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 HotSpot 虚拟机中和 Java 虚拟机栈合二为一.

本地方法被执行的时候, 在本地方法栈也会创建一个栈帧, 用于存放该本地方法的**局部变量表, 操作数栈, 动态链接, 出口信息.**

方法执行完毕后相应的栈帧也会出栈并释放内存空间, 也会出现 StackOverFlowError 和 OutOfMemoryError 两种异常.

## 堆

Java 虚拟机所管理的内存中最大的一块, 唯一目的就是存放对象实例, 几乎所有的对象实例以及数组都在这里分配内存.

Java 堆是垃圾收集器管理的主要区域, 因此也被称作 GC 堆 (Garbage Collected Heap) .从垃圾回收的角度, 由于现在收集器基本都采用分代垃圾收集算法, 所以 Java 堆还可以细分为：新生代和老年代：再细致一点有：Eden 空间, From Survivor, To Survivor 空间, tentired 区老年代等. 进一步划分的目的是更好地回收内存, 或者更快地分配内存.

大部分情况, 对象都会首先在 Eden 区域分配, 在一次新生代垃圾回收后, 如果对象还存活, 则会进入 s0 或者 s1, 并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1), 当它的年龄增加到一定程度 (默认为 15 岁) , 就会被晋升到老年代中. 对象晋升到老年代的年龄阈值, 可以通过参数 -XX:MaxTenuringThreshold 来设置.

## **方法区**

方法区与 Java 堆一样, 是各个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.

永久代是 HotSpot 虚拟机对虚拟机规范中方法区的一种实现方式.方法区是 Java 虚拟机规范中的定义, 是一种规范, 而永久代是一种实现, 一个是标准一个是实现.

#### **常用参数**

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小

|  |  |
| --- | --- |
| 1 2 | -XX:PermSize=N //方法区(永久代)初始大小 -XX:MaxPermSize=N //方法区(永久代)最大大小,超过这个值将会抛出OutOfMemoryError异常: java.lang.OutOfMemoryError: PermGen |

JDK 1.8, 元空间使用的是直接内存, 与永久代很大的不同就是, 如果不指定大小的话, 随着更多类的创建, 虚拟机会耗尽所有可用的系统内存.

|  |  |
| --- | --- |
| 1 2 | -XX:MetaspaceSize=N //设置Metaspace的初始 (和最小大小) -XX:MaxMetaspaceSize=N //设置Metaspace的最大大小 |

整个永久代有一个 JVM 本身设置固定大小上线, 无法进行调整, 而元空间使用的是直接内存, 受本机可用内存的限制, 并且永远不会得到 java.lang.OutOfMemoryError

## **运行时常量池**

运行时常量池是方法区的一部分. Class 文件中除了有类的版本, 字段, 方法, 接口等描述信息外, 还有常量池信息 (用于存放编译期生成的各种字面量和符号引用) .运行时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. JDK1.7 及之后版本的 JVM 已经将运行时常量池从方法区中移了出来, 在 Java 堆 (Heap) 中开辟了一块区域存放运行时常量池.

常量池包含的内容有:

* 字面量
  + 文本字符串
  + 被声明为 final 的常量值
  + 基本数据类型的值
  + 其他
* 符号引用
  + 类和结构的完全限定名
  + 字段名称和描述符
  + 方法名称和描述符

## **直接内存**

直接内存并不是虚拟机运行时数据区的一部分, 也不是虚拟机规范中定义的内存区域, 但是这部分内存也被频繁地使用. 而且也可能导致 OutOfMemoryError 异常出现.

JDK1.4 中新加入的 NIO(New Input/Output) 类, 引入了一种基于通道 (Channel) 与缓存区 (Buffer) 的 I/O 方式, 它可以直接使用 Native 函数库直接分配堆外内存, 然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作. 这样就能在一些场景中显著提高性能, 因为避免了在 Java 堆和 Native 堆之间来回复制数据.

本机直接内存的分配不会收到 Java 堆的限制, 但是, 既然是内存就会受到本机总内存大小以及处理器寻址空间的限制.