**深入理解JVM虚拟机**

# 自动内存管理机制

## Java虚拟机原理

**所谓虚拟机，就是一台虚拟的机器。他是一款软件，用来执行一系列虚拟计算指令，大体上虚拟机可以分为**

**系统虚拟机和程序虚拟机， 大名鼎鼎的Visual Box、Vmare就属于系统虚拟机，他们完全是对物理计算的仿真，**

**提供了一个可以运行完整操作系统的软件平台。**

**程序虚拟机典型代码就是Java虚拟机，它专门为执行单个计算程序而计算，在Java虚拟机中执行的指令我们成为Java**

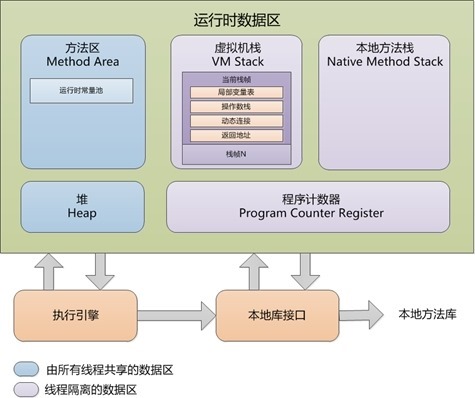
**自己码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。**

**Java发展至今，出现过很多虚拟机，做初Sun使用的一款叫ClassIc的Java虚拟机，到现在引用最广泛的是HotSpot虚拟**

**机，除了Sum意外，还有BEA的Jrockit，目前Jrockit和HostSopt都被oralce收入旗下，大有整合的趋势。**

## Java内存结构





1. **类加载子系统:负责从文件系统或者网络加载Class信息，加载的信息存放在一块称之方法区的内存空间。**
2. **方法区:就是存放类的信息、常量信息、常量池信息、包括字符串字面量和数字常量等。**
3. **Java堆：在Java虚拟机启动的时候建立Java堆，它是Java程序最主要的内存工作区域，几乎所有的对象实例都存放到**

**Java堆中，堆空间是所有线程共享。**

1. **直接内存：JavaNio库允许Java程序直接内存，从而提高性能，通常直接内存速度会优于Java堆。读写频繁的场合可能会考虑使用。**
2. **每个虚拟机线程都有一个私有栈，一个线程的Java栈在线程创建的时候被创建，Java栈保存着局部变量、方法参数、同事Java的方法调用、**

**返回值等。**

1. **本地方法栈，最大不同为本地方法栈用于本地方法调用。Java虚拟机允许Java直接调用本地方法（通过使用C语言写）**
2. **垃圾收集系统是Java的核心，也是不可少的，Java有一套自己进行垃圾清理的机制，开发人员无需手工清理，下一节课详细讲。**
3. **PC（Program Couneter）寄存器也是每个线程私有的空间， Java虚拟机会为每个线程创建PC寄存器，在任意时刻，**

**一个Java线程总是在执行一个方法，这个方法称为当前方法，如果当前方法不是本地方法，PC寄存器总会执行当前正在被执行的指令，**

**如果是本地方法，则PC寄存器值为Underfined，寄存器存放如果当前执行环境指针、程序技术器、操作栈指针、计算的变量指针等信息。**

1. **虚拟机核心的组件就是执行引擎，它负责执行虚拟机的字节码，一般户先进行编译成机器码后执行。**

## 堆、栈、方法区概念区别

### Java堆

**堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。**

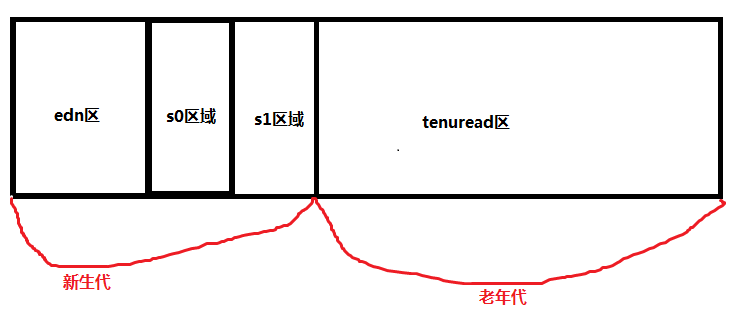
**根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为**

**新生代和老年代。其中新声带存放新生的对象或者年龄不大的对象，老年代则存放老年对象。**

**新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色的空间。**

**绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次**

**新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，则进入老年代。**



### Java栈

**Java栈是一块线程私有的空间，一个栈，一般由三部分组成:局部变量表、操作数据栈和帧数据区**

**局部变量表：用于报错函数的参数及局部变量**

**操作数栈：主要保存计算过程的中间结果，同时作为计算过程中的变量临时的存储空间。**

**帧数据区:除了局部变量表和操作数据栈以外，栈还需要一些数据来支持常量池的解析，这里帧数据区保存着**

**访问常量池的指针，方便计程序访问常量池，另外当函数返回或出现异常时卖虚拟机子必须有一个异常处理表，方便发送异常**

**的时候找到异常的代码，因此异常处理表也是帧数据区的一部分。**

****

### Java方法区

**Java方法区和堆一样，方法区是一块所有线程共享的内存区域，他保存系统的类信息。**

**比如类的字段、方法、常量池等。方法区的大小决定系统可以保存多少个类。如果系统**

**定义太多的类，导致方法区溢出。虚拟机同样会抛出内存溢出的错误。方法区可以理解**

**为永久区。**

## 虚拟机参数配置

### 什么是虚拟机参数配置

**在虚拟机运行的过程中，如果可以跟踪系统的运行状态，那么对于问题的故障**

**排查会有一定的帮助，为此，在虚拟机提供了一些跟踪系统状态的参数，使用**

**给定的参数执行Java虚拟机，就可以在系统运行时打印相关日志，用于分析实际**

**问题。我们进行虚拟机参数配置，其实就是围绕着堆、栈、方法区、进行配置。**

**你说下 你熟悉那些jvm参数调优**

### 堆的参数配置

**-XX:+PrintGC    每次触发GC的时候打印相关日志**

**-XX:+UseSerialGC 串行回收**

**-XX:+PrintGCDetails  更详细的GC日志**

**-Xms    堆初始值**

**-Xmx    堆最大可用值**

**-Xmn    新生代堆最大可用值**

**-XX:SurvivorRatio 用来设置新生代中eden空间和from/to空间的比例.**

**含以-XX:SurvivorRatio=eden/from=den/to**

**总结:在实际工作中，我们可以直接将初始的堆大小与最大堆大小相等，**

**这样的好处是可以减少程序运行时垃圾回收次数，从而提高效率。**

**-XX:SurvivorRatio 用来设置新生代中eden空间和from/to空间的比例.**

#### 设置最大堆内存

**参数: -Xms5m -Xmx20m -XX:+PrintGCDetails -XX:+UseSerialGC -XX:+PrintCommandLineFlags**

|  |
| --- |
| **/\*\***  **\* jvm参数设置**  **\***  **\* @author Administrator**  **\***  **\*/**  **public class JvmDemo01 {**  **public static void main(String[] args) throws InterruptedException {**  **byte[] b1 = new byte[1 \* 1024 \* 1024];**  **System.*out*.println("分配了1m");**  ***jvmInfo*();**  **Thread.*sleep*(3000);**  **byte[] b2 = new byte[4 \* 1024 \* 1024];**  **System.*out*.println("分配了4m");**  **Thread.*sleep*(3000);**  ***jvmInfo*();**  **}**  **/\*\***  **\* 转换为m**  **\***  **\* @param maxMemory**  **\* @return**  **\*/**  **static private String toM(long maxMemory) {**  **float num = (float) maxMemory / (1024 \* 1024);**  **DecimalFormat df = new DecimalFormat("0.00");// 格式化小数**  **String s = df.format(num);// 返回的是String类型**  **return s;**  **}**  **static private void jvmInfo() {**  **// 最大内存**  **long maxMemory = Runtime.*getRuntime*().maxMemory();**  **System.*out*.println("maxMemory:" + maxMemory + ",转换为M:" + *toM*(maxMemory));**  **// 当前空闲内存**  **long freeMemory = Runtime.*getRuntime*().freeMemory();**  **System.*out*.println("freeMemory:" +freeMemory+",转换为M:"+*toM*(freeMemory));**  **// 已经使用内存**  **long totalMemory = Runtime.*getRuntime*().totalMemory();**  **System.*out*.println("totalMemory:" +totalMemory+",转换为M"+*toM*(totalMemory));**  **}**  **}** |

#### 设置新生代与老年代优化参数

**-Xmn    新生代大小，一般设为整个堆的1/3到1/4左右**

**-XX:SurvivorRatio    设置新生代中eden区和from/to空间的比例关系n/1**

##### 设置新生代比例参数

**参数: -Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC**

|  |
| --- |
| **public class JvmDemo02 {**  **public static void main(String[] args) {**  **//-Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC**  **byte [] b = null;**  **for (int i = 0; i < 10; i++) {**  **b =new byte[1\*1024\*1024];**  **}**    **}**    **}** |

##### 设置新生与老年代代参数

-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC

-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC

**-XX:NewRatio=2**

**总结:不同的堆分布情况，对系统执行会产生一定的影响，在实际工作中，**

**应该根据系统的特点做出合理的配置，基本策略：尽可能将对象预留在新生代，**

**减少老年代的GC次数。**

**除了可以设置新生代的绝对大小(-Xmn),可以使用(-XX:NewRatio)设置新生代和老年**

**代的比例:-XX:NewRatio=老年代/新生代**

#### 内存溢出解决办法

##### 设置堆内存大小

错误原因: java.lang.OutOfMemoryError: Java heap space

解决办法:设置堆内存大小 -Xms1m -Xmx70m -XX:+HeapDumpOnOutOfMemoryError

|  |
| --- |
| **public static void main(String[] args) throws InterruptedException {**  **List<Object> list = new ArrayList<>();**  **Thread.*sleep*(3000);**  ***jvmInfo*();**  **for (int i = 0; i < 10; i++) {**  **System.*out*.println("i:"+i);**  **Byte [] bytes= new Byte[1\*1024\*1024];**  **list.add(bytes);**  ***jvmInfo*();**  **}**  **System.*out*.println("添加成功...");**  **}** |

##### 设置栈内存大小

错误原因: java.lang.StackOverflowError

栈溢出 产生于递归调用，循环遍历是不会的，但是循环方法里面产生递归调用， 也会发生栈溢出。

解决办法:设置线程最大调用深度

-Xss5m 设置最大调用深度

|  |
| --- |
| **public class JvmDemo04 {**  **private static int *count*;**  **public static void count(){**  **try {**  ***count*++;**  ***count*();**  **} catch (Throwable e) {**  **System.*out*.println("最大深度:"+*count*);**  **e.printStackTrace();**  **}**  **}**  **public static void main(String[] args) {**  ***count*();**  **}**  **}** |

#### Tomcat内存溢出在catalina.sh 修改JVM堆内存大小 JAVA\_OPTS="-server -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:MaxNewSize=512m"

### JVM参数调优总结

    在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

* GC的时间足够的小
* GC的次数足够的少
* 发生Full GC的周期足够的长

  前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

   （1）针对JVM堆的设置，一般可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值  
   （2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

   （3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

* 更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC
* 更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率
* 如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间