一、JVM内存模型

Java虚拟机的内存根据用途被划分为五部分：

程序计数器

Java栈

本地方法栈

Java堆

方法区

如下图



这些区域都有什么用途呢

1、程序计数器

程序计数器的内存区域很小，用来记录当前线程正在执行的字节码指令地址，是线程私有的，每个线程都有自己的程序计数器。

注意：字节码指令就是jvm跨语言跨平台的最终奥秘，不管什么语言，只要编译器生成class文件结构的编译结果就能在jvm上运行，编写的代码字节码可以使用javap来查看。

字节码的长度是一个字节，所以字节码不能超过256个，现在已有200多个字节码指令。

程序计数器对于线程有什么用呢，线程在时间片耗尽以后会被挂起，再次运行的时候，需要通过自己的程序计数器找到执行到哪里了。

程序计数器只记录了一个值，所以这个区域不需要太大空间，程序计数器也是唯一没有内存溢出的区域。

注意：在执行native本地方法的时候，程序计数器的值为空。

2、java栈

Java栈主要服务于Java方法，每个方法执行时都会创建一个栈帧，方法的调用就是栈帧入栈出栈的过程，栈帧中存储局部变量表、操作数栈、动态链接、方法出口，Java栈是线程私有的，生命周期和线程一致。

1、局部变量表是存储变量的位置，存储了编译器可知的基本类型数据、对象引用和returnAddress类型。

经常说的基本类型和对象类型的引用存储在栈上，就是说的局部变量表

不过，基本类型是存储在Java栈中的这种说法不全对，这里的基本类型指的是局部变量中的基本类型（在方法中的临时变量），堆中对象的基本类型属性存储在堆中，final和static修饰的基本类型变量存储在方法区。可以考虑下面代码中no1、no2、no3的存储情况。

public class Demo1 {  
 static int *no1*=0;  
 int no2=0;  
 public void demo()  
 {  
 int no3=0;  
 }  
}

2、操作数栈，可以理解为jvm虚拟机计算过程中的一个临时存储区。存储的数据与局部变量表一样，都是基本类型数据、对象引用和returnAddress类型。

方法开始执行时，操作数栈是空的，随着字节码的执行，数据值不断的向操作数栈写入和提取，既然是栈，那就写入和提取就分别对应着入栈和出栈。

操作数栈是一个临时存储区域，经常会有人质疑它是否真实存在，不过我个人理解是操作数栈存在的。

3、动态链接，每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接(Dynamic Linking)。

个人理解，class文件包含大量的符号引用，符号引用转换为直接引用的过程包括：动态链接和静态链接

静态链接，加载的解析阶段会将符号引用转换为直接引用，即引用指向真实的的物理存储地址。

动态链接，在运行期间将符号引用转换为直接引用的过程。

3、本地方法栈

本地方法栈和java栈作用差不多，不过服务的对象是native方法。

4、java堆

这是最大的一块内存区域，通常说的java内存主要就是堆。

堆中存储了所有的对象类型和数组，不过这个在编译期间，逃逸优化以后不是绝对的了。

Java线程共享堆内存，堆可以分为两部分：年轻代和年老代，年轻代又可以分为三部分：Eden、From Survivor、To Survivor。

年轻代用来存放新创建的对象，通常认为年轻代中对象的死亡率很高，所以年轻代的三个区域使用复制算法，相互拷贝存活的对象，每一次拷贝，对象的年龄加1，达到一定年龄（默认15 可以通过参数-XX:MaxTenuringThreshold设置）后会被放到老年代。

老年代存储的是存活了很久的对象，这个区域对象的垃圾收集算法的是标记整理算法。

注意：年轻代中对象移到年老代还有以下情况

1、存活对象太多，Eden存储空间不足

2、大对象会直接进入老年代（-XX：PretenureSizeThreshold设置大小）

3、相同年龄的对象超过survivor空间的一半，大于等于这个年龄的会进入老年代

5、方法区

方法区也被成为永久代（垃圾回收条件很苛刻），方法区在线程中也是共享的。

方法区中存储了：

1、class的类信息

2、常量

3、静态变量

4、普通方法在编译器编译后生成的字节码

运行常量池是在方法区的一部分。

二、内存溢出

除了方法计数器不会有内存溢出，其他的内存区域都会有内存溢出的情况

1、堆溢出

异常：

java.lang.OutOfMemoryError:Java heap space

java.lang.OutOfMemoryError: GC overhead limit exceeded

原因：

1、堆内存溢出，即创建对象所需要的内存空间已经超过堆的最大值

2、内存泄漏，已经申请的内存空间无法释放，就会造成内存泄漏，例如线程死循环、资源不关闭等，大量的内存泄漏堆积就会造成内存不够，从而发生内存溢出

设置参数-Xms20m -Xmx20m运行代码

public static void main(String[] args) {  
 for(int i=0;i<100;i++)  
 {  
 new Thread(()->{  
 List<String> list = new ArrayList();  
 while(true)  
 {  
 list.add("mg" + System.*currentTimeMillis*());  
 }  
 }).start();  
 }  
}

解决方案：

内存不足的情况下，修改-Xmx和-Xms调节内存大小就能解决

内存泄露的情况需要通过jmap和jstack或者jvisualvm生成内存dump快照来具体分析哪里堆积了大量内存（具体的分析过程暂不详述）

2、方法区溢出

异常 ： java.lang.OutOfMemoryError: PermGen space

原因：方法区加载了类信息、常量和静态变量，加载占用内存太大，空间不足

解决方案：

通过**-XX:PermSize和-XX:MaxPermSize**调节方法区大小

3、栈溢出

异常：java.lang.StackOverflowError

原因：需要栈大小 > 栈的最大允许大小,一般是代码中出现无限递归的情况，如下代码

public class StackOverFlow {  
 public static void main(String[] args) {  
 StackOverFlow overFlow = new StackOverFlow();  
 overFlow.each();  
 }  
 public void each()  
 {  
 each();  
 }  
}

解决方案：谨慎使用递归，控制递归的深度

4、直接内存溢出

异常：java.lang.OutOfMemoryError: Direct buffer memory

原因：Direct Memory区域分配内存失败，这个内存不在堆上，这个区域的回收也是通过GC，不过回收的前提是Direct buffer在堆上对象的引用被回收。试着执行如下代码

public static void main(String[] args) {  
 List<ByteBuffer> list = new ArrayList<ByteBuffer>();  
 while(true)  
 {  
 list.add(ByteBuffer.*allocateDirect*(1000\*5));  
 }  
}

解决方案：通过-XX:MaxDirectMemorySize修改直接内存大小。