JVM内存模型看这个就够了

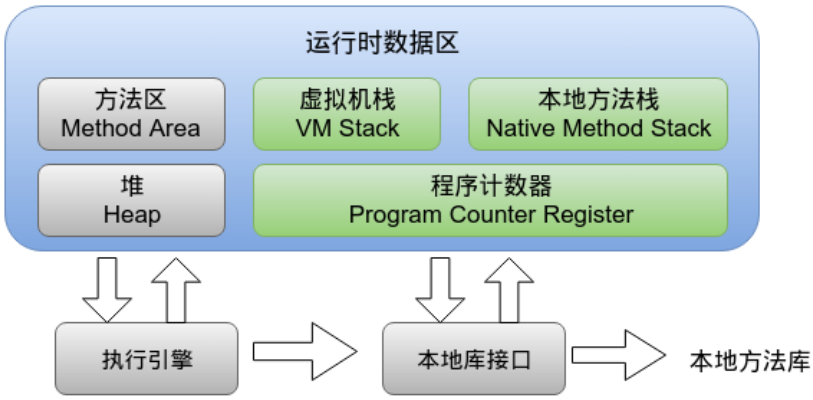
牛客7380095号
编辑于 2019-01-11 11:17:49

0 联系我

1.Q群【Java开发技术交流】：https://jq.qq.com/?_wv=1027&k=5UB4P1T
2.简书博客:www.shishusheng.com
3.知乎:http://www.zhihu.com/people/shi-shu-sheng-
4.微信公众号:JavaEdge
5.Github:https://github.com/Wasabi1234

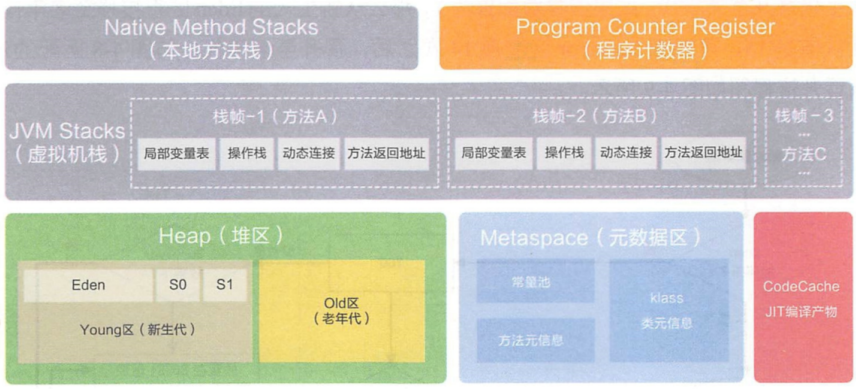
正文

内存是非常重要的系统资源，是硬盘和CPU的中间仓库及桥梁，承载着操作系统和应用程序的实时运行
JVM内存布局规定了Java在运行过程中内存申请、分配、管理的策略，保证了JVM的高效稳定运行
不同的JVM对于内存的划分方式和管理机制存在着部分差异
结合JVM虚拟机规范，来探讨经典的JVM内存布局




由所有线程共享的数据区


线程隔离的数据区



1 Program Counter Register (程序计数寄存器)

示器等，线程执行或恢复都要依赖程序计数器。程序计数器在各个线程之间互不影响，此区域也不会发生内存溢出异常。

1.1. 定义

程序计数器是一块较小的内存空间，可看作当前线程正在执行的字节码的行号指示器
如果当前线程正在执行的是

- Java方法
  计数器记录的就是当前线程正在执行的字节码指令的地址
- 本地方法
  那么程序计数器值为undefined

1.2. 作用

程序计数器有两个作用

- 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
- 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。

1.3. 特点

一块较小的内存空间
线程私有。每条线程都有一个独立的程序计数器。
是唯一一个不会出现OOM的内存区域。
生命周期随着线程的创建而创建，随着线程的结束而死亡。

2. Java虚拟机栈(JVM Stack)

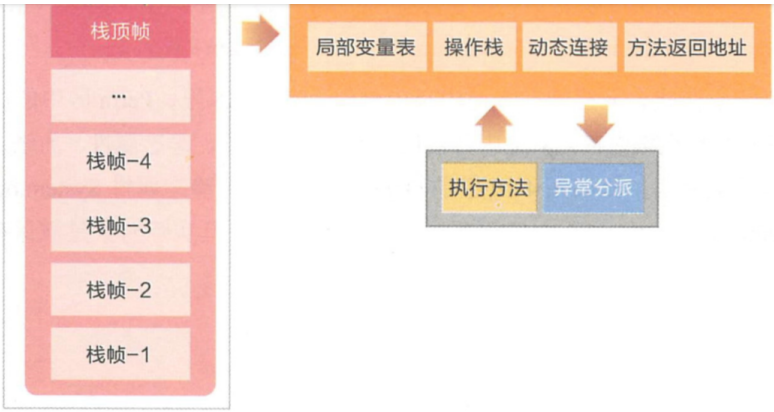
2.1. 定义

相对于基于寄存器的运行环境来说，JVM是基于栈结构的运行环境
栈结构移植性更好，可控性更强
JVM中的虚拟机栈是描述Java方法执行的内存区域，它是线程私有的

栈中的元素用于支持虚拟机进行方法调用，每个方法从开始调用到执行完成的过程，就是栈帧从入栈到出栈的过程
在活动线程中，只有位于栈顶的帧才是有效的，称为当前栈帧
正在执行的方法称为当前方法
栈帧是方法运行的基本结构

在执行引擎运行时，所有指令都只能针对当前栈帧进行操作
StackOverflowError表示请求的栈溢出，导致内存耗尽，通常出现在递归方法中
JVM能够横扫千军，虚拟机栈就是它的心腹大将，当前方法的栈帧，都是正在战斗的战场，其中的操作栈

JVM内存模型看这个就够了



虚拟机栈通过压/出栈的方式，对每个方法对应的活动栈帧进行运算处理，方法正常执行结束，肯定会跳转到另一个栈帧上
在执行的过程中，如果出现异常，会进行异常回溯，返回地址通过异常处理表确定
栈帧在整个JVM体系中的地位颇高,包括局部变量表、操作栈、动态连接、方法返回地址等

- 局部变量表
  存放方法参数和局部变量
  相对于类属性变量的准备阶段和初始化阶段来说，局部变量没有准备阶段，必须显式初始化
  如果是非静态方法，则在index[0]位置上存储的是方法所属对象的实例引用，随后存储的是参数和局部变量
  字节码指令中的STORE指令就是将操作栈中计算完成的局部变量写回局部变量表的存储空间内
- 操作栈
  操作栈是一个初始状态为空的桶式结构栈
  在方法执行过程中，会有各种指令往栈中写入和提取信息
  JVM的执行引擎是基于栈的执行引擎，其中的栈指的就是操作栈
  字节码指令集的定义都是基于栈类型的,栈的深度在方法元信息的stack属性中

下面用一段简单的代码说明操作栈与局部变量表的交互

```
13      public int simpleMethod() {
14          int x = 13;
15          int y = 14;
16          int z = x + y;
17          return z;
18      }
```

- 详细的字节码操作顺序如下:

```
public int simpleMethod();
  descriptor: ()I
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=4, args_size=1 // 最大栈深度为2,局部变量个数为4
       0: bipush        13         // 常量13压入操作栈
       2: istore_1                 // 并保存到局部变量表的slot_1中 (第1处)

       3: bipush        14         // 常量14压入操作栈
       5: istore_2                 // 并保存到局部变量表的slot_2中

       6: iload_1                  // 把局部变量表的slot_ 1元素(int x)压入操作栈
       7: iload_2                  // 把局部变量表的slot_2元素(inty)压入操作栈
       8: iadd                     // 把上方的两个数都取出来，在CPU里加一下，并压回操作栈的栈顶
       9: istore_3                 // 把栈顶的结果存储到局部变量表的slot 3中
      10: iload_3
      11: ireturn                  // 返回栈顶元素值
```

第1处说明:局部变量表就像个中药柜，里面有很多抽屉,依次编号为0, 1, 2,3，.,. n
字节码指令istore_ 1就是打开1号抽屉，把栈顶中的数13存进去
栈是一个很深的竖桶，任何时候只能对桶口元素进行操作，所以数据只能在栈顶进行存取

0: iload_1 1: iinc 1, 1 4: istore_2	0: iinc 1, 1 3: iload_1 4: istore_2

- iload_ 1从局部变量表的第1号抽屉里取出一个数,压入栈顶，下一步直接在抽屉里实现+1的操作，而这个操作对栈顶元素的值没有影响
  所以istore_ 2只是把栈顶元素赋值给a
- 表格右列，先在第1号抽屉里执行+1操作，然后通过iload_ 1 把第1号抽屉里的数压入栈顶，所以istore_ 2存入的是+1之后的值

这里延伸一个信息，i++并非原子操作。即使通过volatile关键字进行修饰，多个线程同时写的话，也会产生数据互相覆盖的问题.

- 动态连接
  每个栈帧中包含一个在常量池中对当前方法的引用，目的是支持方法调用过程的动态连接
- 方法返回地址
  方法执行时有两种退出情况
  - 正常退出
    正常执行到任何方法的返回字节码指令，如RETURN、IRETURN、ARETURN等
  - 异常退出

无论何种退出情况，都将返回至方法当前被调用的位置。方法退出的过程相当于弹出当前栈帧

退出可能有三种方式:

- 返回值压入，上层调用栈帧
- 异常信息抛给能够处理的栈帧
- PC计数器指向方法调用后的下一条指令

Java虚拟机栈是描述Java方法运行过程的内存模型

Java虚拟机栈会为每一个即将运行的Java方法创建“栈帧”
用于存储该方法在运行过程中所需要的一些信息

- 局部变量表
  存放基本数据类型变量、引用类型的变量、returnAddress类型的变量
- 操作数栈
- 动态链接
- 当前方法的常量池指针
- 当前方法的返回地址
- 方法出口等信息

每一个方法从被调用到执行完成的过程,都对应着一个个栈帧在JVM栈中的入栈和出栈过程

> 注意：人们常说，Java的内存空间分为“栈”和“堆”，栈中存放局部变量，堆中存放对象。
> 这句话不完全正确！这里的“堆”可以这么理解，但这里的“栈”就是现在讲的虚拟机栈,或者说Java虚拟机栈中的局部变量表部分.
> 真正的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.

2.2. 特点

局部变量表的创建是在方法被执行的时候,随着栈帧的创建而创建.
而且表的大小在编译期就确定,在创建的时候只需分配事先规定好的大小即可.
在方法运行过程中,表的大小不会改变

- **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虚拟机栈也是线程私有的,每个线程都有各自的Java虚拟机栈,而且随着线程的创建而创建,随着线程的死亡而死亡.

3. 本地方法栈(Native Method Stack)

本地方法栈和Java虚拟机栈实现的功能与抛出异常几乎相同
只不过虚拟机栈是为虚拟机执行Java方法(也就是字节码)服务,本地方法区则为虚拟机使用到的Native方法服务.

在JVM内存布局中，也是线程对象私有的,但是虚拟机栈“主内”，而本地方法栈“主外”
这个“内外”是针对JVM来说的，本地方法栈为Native方法服务
线程开始调用本地方法时，会进入一个不再受JVM约束的世界
本地方法可以通过JNI(Java Native Interface)来访问虚拟机运行时的数据区，甚至可以调用寄存器,具有和JVM相同的能力和权限
当大量本地方法出现时,势必会削弱JVM对系统的控制力,因为它的出错信息都比较黑盒.
对于内存不足的情况，本地方法栈还是会抛出native heap OutOfMemory

最著名的本地方法应该是System.currentTimeMillis()，JNI 使Java深度使用OS的特性功能，复用非Java代码
但是在项目过程中，如果大量使用其他语言来实现JNI,就会丧失跨平台特性，威胁到程序运行的稳定性
假如需要与本地代码交互，就可以用中间标准框架进行解耦，这样即使本地方法崩溃也不至于影响到JVM的稳定
当然，如果要求极高的执行效率、偏底层的跨进程操作等，可以考虑设计为JNI调用方式

4 Java堆(Java Heap)

Heap是OOM故障最主要的发源地，它存储着几乎所有的实例对象，堆由垃圾收集器自动回收，堆区由各子线程共享使用
通常情况下，它占用的空间是所有内存区域中最大的，但如果无节制地创建大量对象，也容易消耗完所有的空间
堆的内存空间既可以固定大小，也可运行时动态地调整，通过如下参数设定初始值和最大值，比如

```
-Xms256M. -Xmx1024M
```

其中-X表示它是JVM运行参数

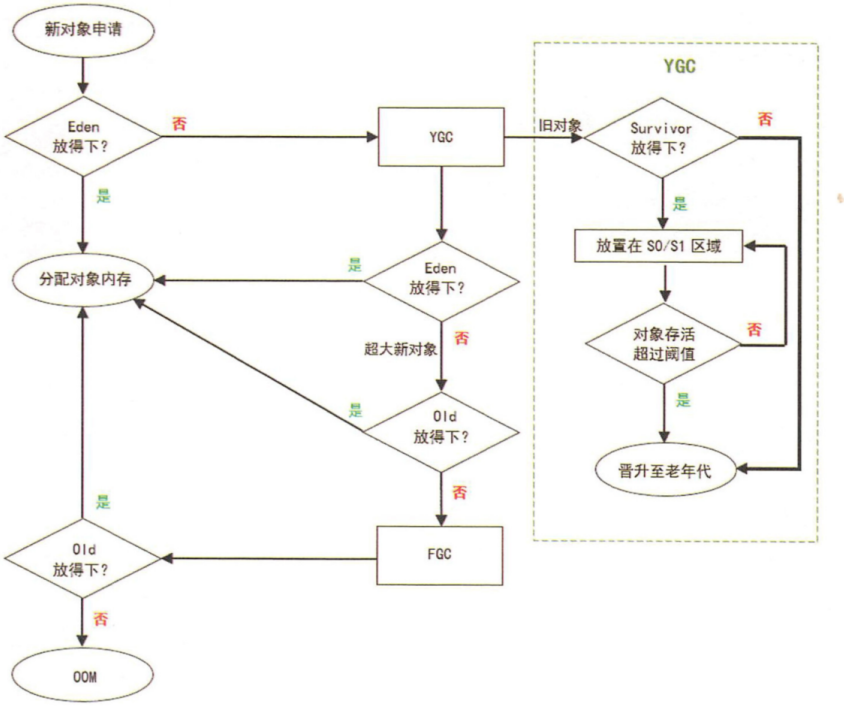
- ms是memorystart的简称 最小堆容量
- mx是memory max的简称 最大堆容量

但是在通常情况下，服务器在运行过程中，堆空间不断地扩容与回缩，势必形成不必要的系统压力，所以在线上生产环境中，JVM的Xms和Xmx设置成一样大小，避免在GC后调整堆大小时带来的额外压力

堆分成两大块:新生代和老年代
对象产生之初在新生代，步入暮年时进入老年代，但是老年代也接纳在新生代无法容纳的超大对象

新生代= 1个Eden区+ 2个Survivor区
绝大部分对象在Eden区生成，当Eden区装填满的时候，会触发Young GC。垃圾回收的时候，在Eden区实现清除策略，没有被引用的对象则直接回收。依然存活的对象会被移送到Survivor区，这个区真是名副其实的存在
Survivor 区分为S0和S1两块内存空间，送到哪块空间呢?每次Young GC的时候，将存活的对象复制到未使用的那块空间，然后将当前正在使用的空间完全清除，交换两块空间的使用状态
如果YGC要移送的对象大于Survivor区容量上限，则直接移交给老年代

生代的Eden区直接移至老年代。默认值是15，可以在Survivor 区交换14次之后，晋升至老年代



若Survivor区无法放下，或者超大对象的阈值超过上限，则尝试在老年代中进行分配;
如果老年代也无法放下，则会触发Full Garbage Collection(Full GC);
如果依然无法放下，则抛OOM.

堆出现OOM的概率是所有内存耗尽异常中最高的
出错时的堆内信息对解决问题非常有帮助，所以给JVM设置运行参数-

```
XX:+HeapDumpOnOutOfMemoryError
```

让JVM遇到OOM异常时能输出堆内信息

> 在不同的JVM实现及不同的回收机制中，堆内存的划分方式是不一样的

存放所有的类实例及数组对象
除了实例数据，还保存了对象的其他信息，如Mark Word（存储对象哈希码，GC标志，GC年龄，同步锁等信息），Klass Pointy(指向存储类型元数据的指针）及一些字节对齐补白的填充数据（若实例数据刚好满足8字节对齐，则可不存在补白）

特点

Java虚拟机所需要管理的内存中最大的一块.

堆内存物理上不一定要连续,只需要逻辑上连续即可,就像磁盘空间一样.
堆是垃圾回收的主要区域,所以也被称为GC堆.

堆的大小既可以固定也可以扩展,但主流的虚拟机堆的大小是可扩展的(通过-Xmx和-Xms控制),因此当线程请求分配内存,但堆已满,且内存已满无法再扩展时,就抛出OutOfMemoryError.

线程共享
整个Java虚拟机只有一个堆,所有的线程都访问同一个堆.
它是被所有线程共享的一块内存区域,在虚拟机启动时创建.
而程序计数器、Java虚拟机栈、本地方法栈都是一个线程对应一个

5 方法区

5.2 特点

- 线程共享
  方法区是堆的一个逻辑部分,因此和堆一样,都是线程共享的.整个虚拟机中只有一个方法区.
- 永久代
  方法区中的信息一般需要长期存在，而且它又是堆的逻辑分区，因此用堆的划分方法，我们把方法区称为永久代.
- 内存回收效率低
  Java虚拟机规范对方法区的要求比较宽松,可以不实现垃圾收集.
  方法区中的信息一般需要长期存在,回收一遍内存之后可能只有少量信息无效.
  对方法区的内存回收的主要目标是:**对常量池的回收和对类型的卸载**

和堆一样，允许固定大小，也允许可扩展的大小，还允许不实现垃圾回收。

当方法区内存空间无法满足内存分配需求时,将抛出OutOfMemoryError异常.

5.3 运行时常量池(Runtime Constant Pool)

5.3.1 定义

运行时常量池是方法区的一部分.
方法区中存放三种数据：类信息、常量、静态变量、即时编译器编译后的代码.其中常量存储在运行时常量池中.

我们知道，.java文件被编译之后生成的.class文件中除了包含：类的版本、字段、方法、接口等信息外，还有一项就是常量池
常量池中存放编译时期产生的各种字面量和符号引用，.class文件中的常量池中的所有的内容在类被加载后存放到方法区的运行时常量池中。
PS：int age = 21;//age是一个变量，可以被赋值；21就是一个字面值常量，不能被赋值；
int final pai = 3.14;//pai就是一个符号常量，一旦被赋值之后就不能被修改。

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(Constant pool table)，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入运行时常量池中存放。运行时常量池相对于class文件常量池的另外一个特性是具备动态性，java语言并不要求常量一定只有编译器才产生，也就是并非预置入class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中。

在近三个JDK版本（6、7、8）中， 运行时常量池的所处区域一直在不断的变化，
在JDK6时它是方法区的一部分
7又把他放到了堆内存中
8之后出现了元空间，它又回到了方法区。
其实，这也说明了官方对“永久代”的优化从7就已经开始了

5.3.2 特性

class文件中的常量池具有动态性.
Java并不要求常量只能在编译时候产生,Java允许在运行期间将新的常量放入方法区的运行时常量池中.
String类中的intern()方法就是采用了运行时常量池的动态性.当调用 intern 方法时,如果池已经包含一个等于此 String 对象的字符串,则返回池中的字符串.否则,将此 String 对象添加到池中,并返回此 String 对象的引用.

5.3.3 可能抛出的异常

运行时常量池是方法区的一部分,所以会受到方法区内存的限制,因此当常量池无法再申请到内存时就会抛出OutOfMemoryError异常.

量。

当运行时常量池中的某些常量没有被对象引用，同时也没有被变量引用，那么就需要垃圾收集器回收。

6 直接内存(Direct Memory)

直接内存不是虚拟机运行时数据区的一部分,也不是JVM规范中定义的内存区域,但在JVM的实际运行过程中会频繁地使用这块区域.而且也会抛OOM

在JDK 1.4中加入了NIO(New Input / Output)类,引入了一种基于管道和缓冲区的IO方式,它可以使用Native函数库直接分配堆外内存,然后通过一个存储在堆里的DirectByteBuffer对象作为这块内存的引用来操作堆外内存中的数据.
这样能在一些场景中显著提升性能,因为避免了在Java堆和Native堆中来回复制数据.

> 综上看来
> 程序计数器、Java虚拟机栈、本地方法栈是线程私有的，即每个线程都拥有各自的程序计数器、Java虚拟机栈、本地方法区。并且他们的生命周期和所属的线程一样。
> 而堆、方法区是线程共享的，在Java虚拟机中只有一个堆、一个方法栈。并在JVM启动的时候就创建，JVM停止才销毁。

7 Metaspace (元空间)

在JDK8，元空间的前身Perm区已经被淘汰,在JDK7及之前的版本中，只有Hotspot才有Perm区(永久代),它在启动时固定大小,很难进行调优,并且Full GC时会移动类元信息

在某些场景下,如果动态加载类过多,容易产生Perm区的OOM.
比如某个实际Web工程中，因为功能点比较多，在运行过程中，要不断动态加载很多的类，经常出现致命错误:

```
Exception in thread ‘dubbo client x.x connector' java.lang.OutOfMemoryError: PermG
```

为解决该问题，需要设定运行参数

```
-XX:MaxPermSize= l280m
```

如果部署到新机器上，往往会因为JVM参数没有修改导致故障再现。不熟悉此应用的人排查问题时往往苦不堪言，除此之外，永久代在GC过程中还存在诸多问题

所以，JDK8使用元空间替换永久代.区别于永久代，元空间在本地内存中分配.
也就是说，只要本地内存足够，它不会出现像永久代中java.lang.OutOfMemoryError: PermGen space

同样的，对永久代的设置参数PermSize和MaxPermSize也会失效
在JDK8及以上版本中，设定MaxPermSize参数，JVM在启动时并不会报错，但是会提示:

```
Java HotSpot 64Bit Server VM warning:ignoring option MaxPermSize=2560m; support wa
```

默认情况下，“元空间”的大小可以动态调整，或者使用新参数MaxMetaspaceSize来限制本地内存分配给类元数据的大小.

在JDK8里，Perm 区所有内容中

- 字符串常量移至堆内存

JVM内存模型看这个就够了


```
#5 = Integer            10000000
#6 = Class              #34            // java/lang/Object
```

比如上图中的Object类元信息、静态属性System.out、整型常量000000等
图中显示在常量池中的String，其实际对象是被保存在堆内存中的。

元空间特色

- 充分利用了Java语言规范：类及相关的元数据的生命周期与类加载器的一致
- 每个类加载器都有它的内存区域-元空间
- 只进行线性分配
- 不会单独回收某个类（除了重定义类 RedefineClasses 或类加载失败）
- 没有GC扫描或压缩
- 元空间里的对象不会被转移
- 如果GC发现某个类加载器不再存活，会对整个元空间进行集体回收

GC

- Full GC时，指向元数据指针都不用再扫描，减少了Full GC的时间
- 很多复杂的元数据扫描的代码（尤其是CMS里面的那些）都删除了
- 元空间只有少量的指针指向Java堆
  这包括：类的元数据中指向java.lang.Class实例的指针;数组类的元数据中，指向java.lang.Class集合的指针。
- 没有元数据压缩的开销
- 减少了GC Root的扫描（不再扫描虚拟机里面的已加载类的目录和其它的内部哈希表）
- G1回收器中，并发标记阶段完成后就可以进行类的卸载

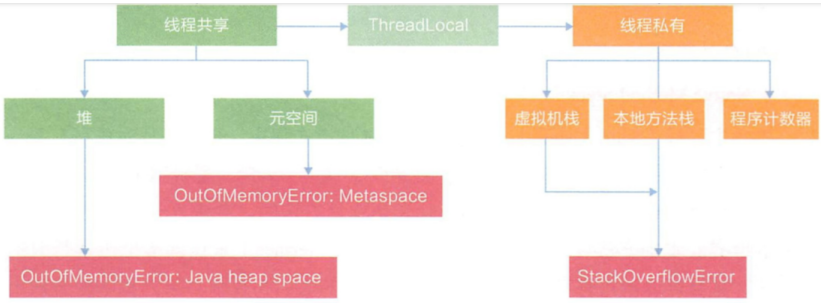
元空间内存分配模型

- 绝大多数的类元数据的空间都在本地内存中分配
- 用来描述类元数据的对象也被移除
- 为元数据分配了多个映射的虚拟内存空间
- 为每个类加载器分配一个内存块列表
  - 块的大小取决于类加载器的类型
  - Java反射的字节码存取器（sun.reflect.DelegatingClassLoader ）占用内存更小
- 空闲块内存返还给块内存列表
- 当元空间为空，虚拟内存空间会被回收
- 减少了内存碎片

最后,从线程共享的角度来看

- 堆和元空间是所有线程共享的
- 虚拟机栈、本地方法栈、程序计数器是线程内部私有的

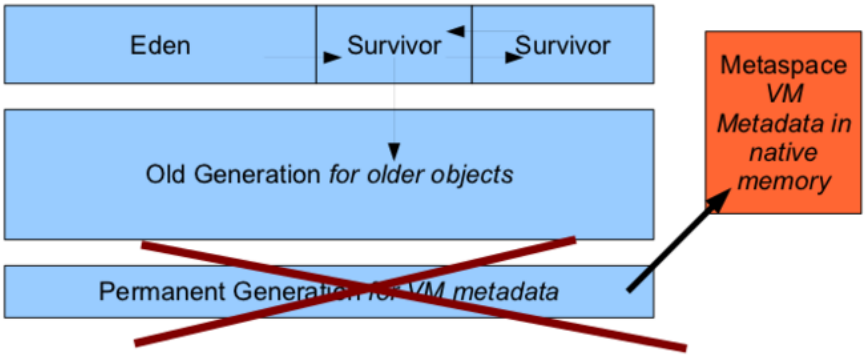
JVM内存模型看这个就够了



8 从GC角度看Java堆

> 堆和方法区都是线程共享的区域，主要用来存放对象的相关信息。我们知道，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序运行期间才能知道会创建哪些对象，因此， 这部分的内存和回收都是动态的，垃圾收集器所关注的就是这部分内存（本节后续所说的“内存”分配与回收也仅指这部分内存）。而在JDK1.7和1.8对这部分内存的分配也有所不同，下面我们来详细看一下

Java8中堆内存分配如下图：



9 JVM关闭

- 正常关闭：当最后一个非守护线程结束或调用了System.exit或通过其他特定于平台的方式,比如ctrl+c。
- 强制关闭：调用Runtime.halt方法，或在操作系统中直接kill（发送single信号）掉JVM进程。
- 异常关闭：运行中遇到RuntimeException 异常等

在某些情况下，我们需要在JVM关闭时做一些扫尾的工作，比如删除临时文件、停止日志服务。为此JVM提供了关闭钩子（shutdown hocks）来做这些事件。

Runtime类封装java应用运行时的环境，每个java应用程序都有一个Runtime类实例，使用程序能与其运行环境相连。

关闭钩子本质上是一个线程（也称为hock线程），可以通过Runtime的addshutdownhock （Thread hock）向主jvm注册一个关闭钩子。hock线程在jvm正常关闭时执行，强制关闭不执行。

对于在jvm中注册的多个关闭钩子，他们会并发执行，jvm并不能保证他们的执行顺序。