面试专题（一）

jvm虚拟机

lison老师：3325521094

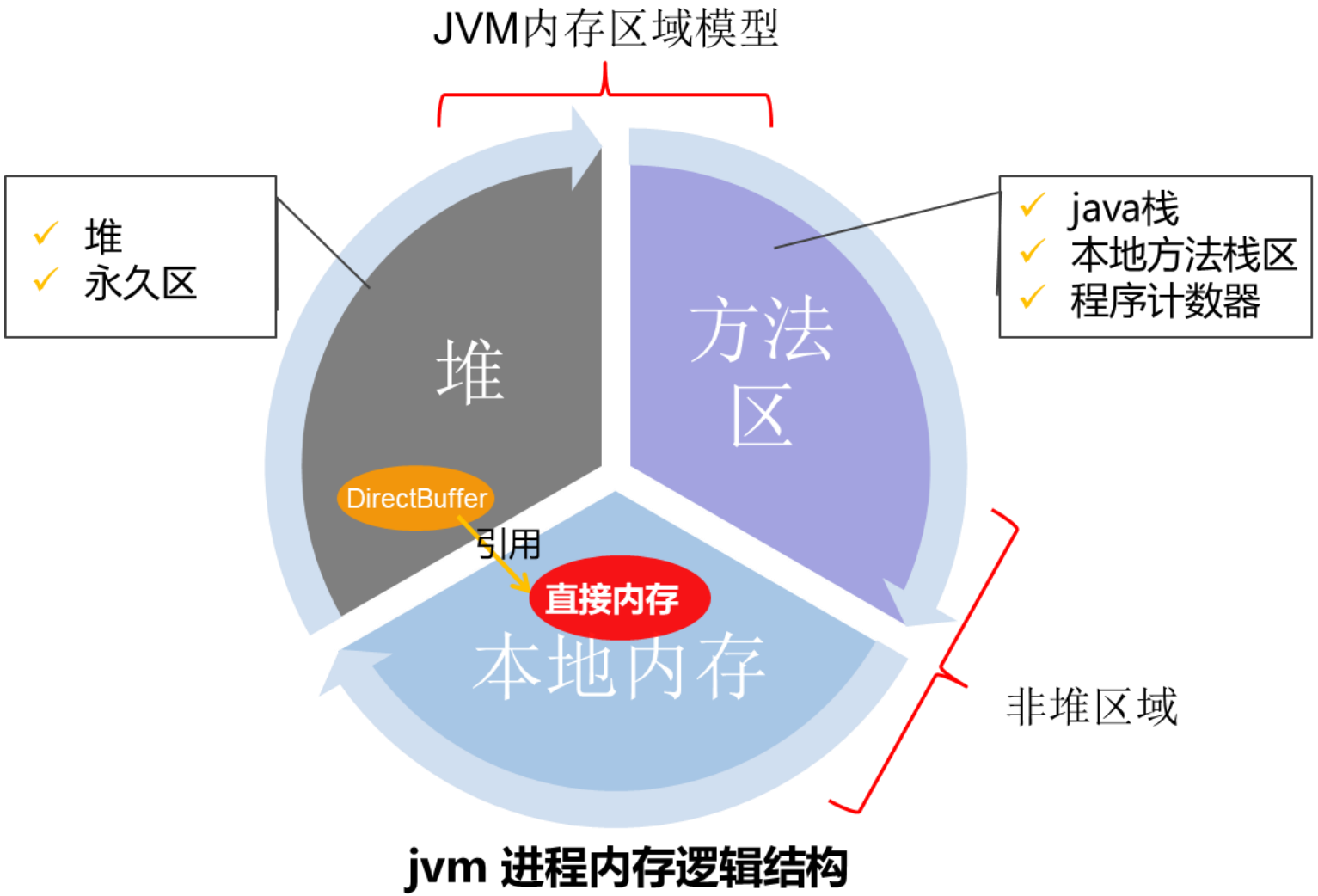
目录 CONTENTS

1 java内存模型问题

2 常见内存溢出异常问题

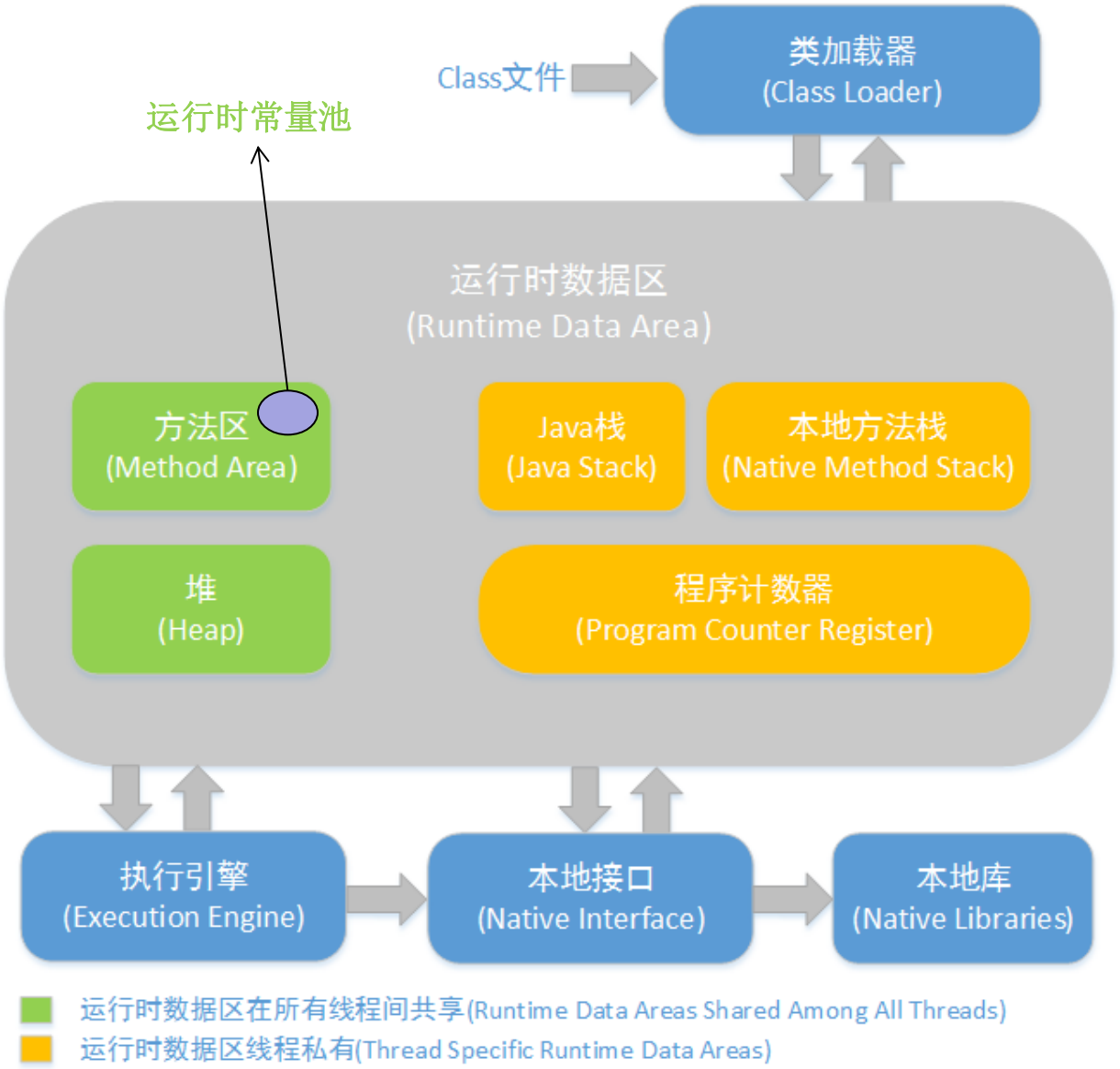
3 垃圾回收面试问题

01 jvm 进程内存逻辑结构



jvm 进程内存逻辑结构

02 你对jvm内存结构了解吗？

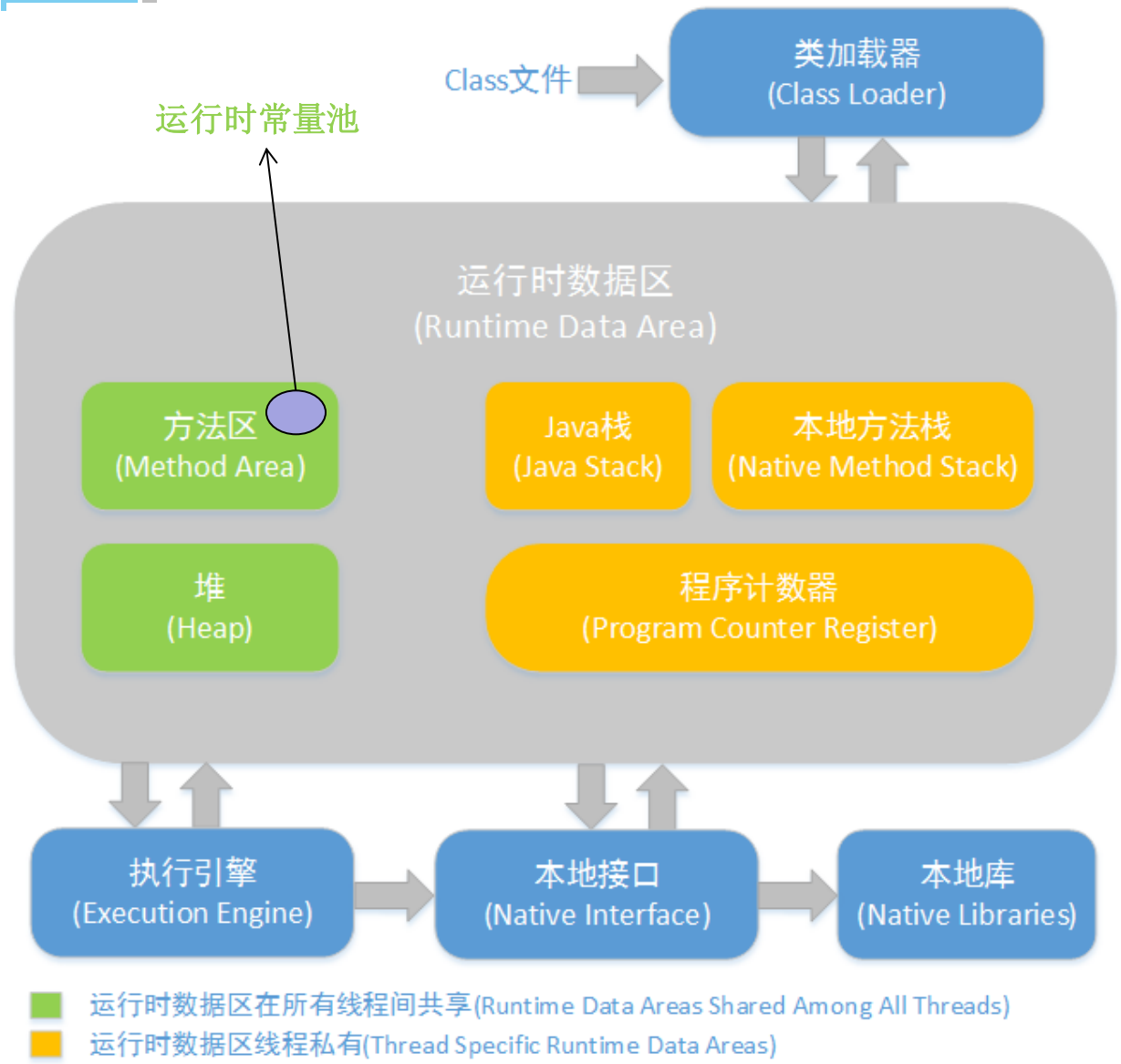


程序计数器：较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响；

java栈：线程私有，生命周期和线程，每个方法在执行的同时都会创建一个**栈帧**用于存储局部变量表，操作数栈，动态链接，方法出口等信息。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程；栈里面存放着各种基本数据类型和对象的引用；

本地方法栈：本地方法栈保存的是native方法的信息，当一个JVM创建的线程调用native方法后，JVM不再为其在虚拟机栈中创建栈帧，JVM只是简单地动态链接并直接调用native方法；

03 你对jvm内存结构了解吗？

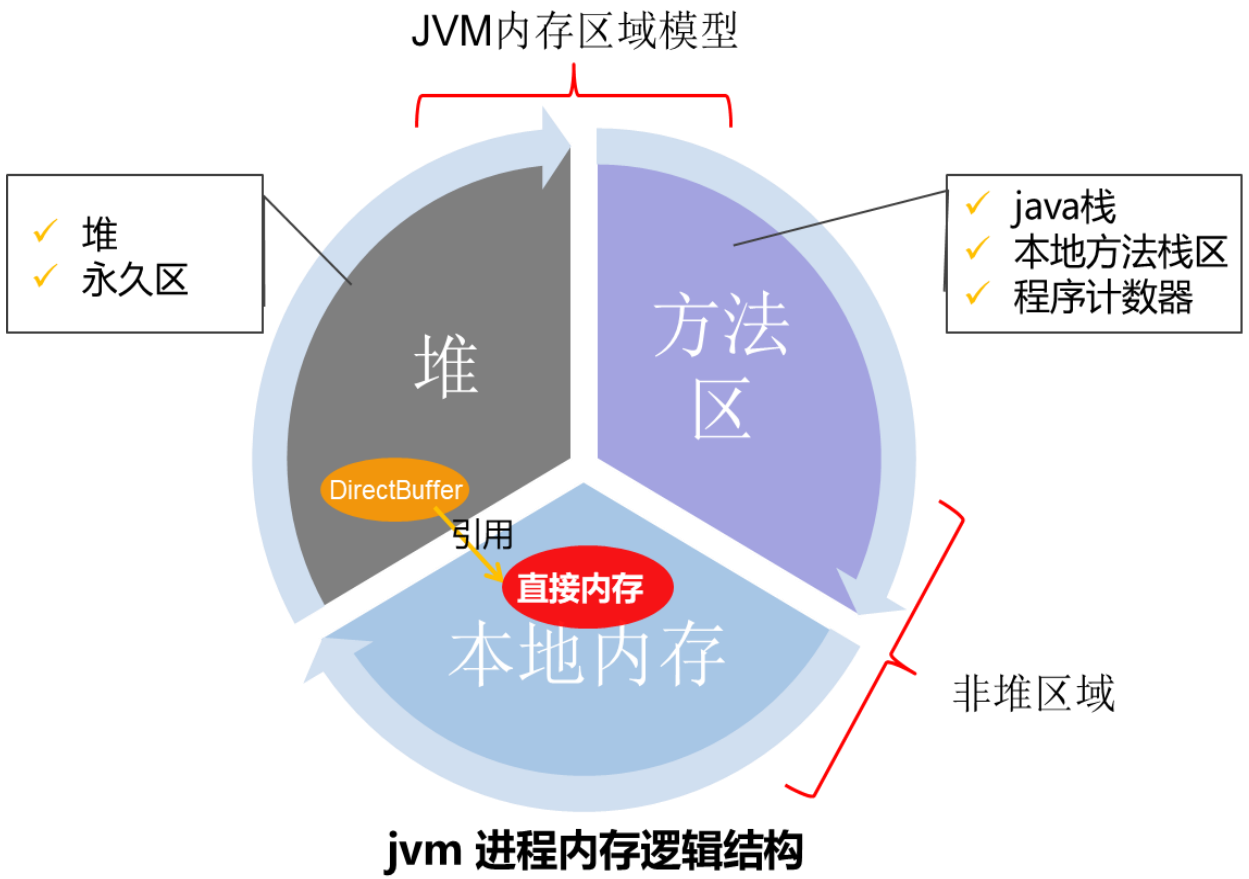


堆：Java堆是Javaer需要重点关注的一块区域，因为涉及到内存的分配(new关键字，反射等)与回收(回收算法，收集器等)；

方法区：也叫永久区，用于存储已经被虚拟机加载的类信息，常量("zdy","123"等)，静态变量(static变量)等数据。

运行时常量池：运行时常量池是方法区的一部分，用于存放编译期生成的各种字面量("zdy","123"等)和符号引用。

04 jvm 进程内存逻辑结构

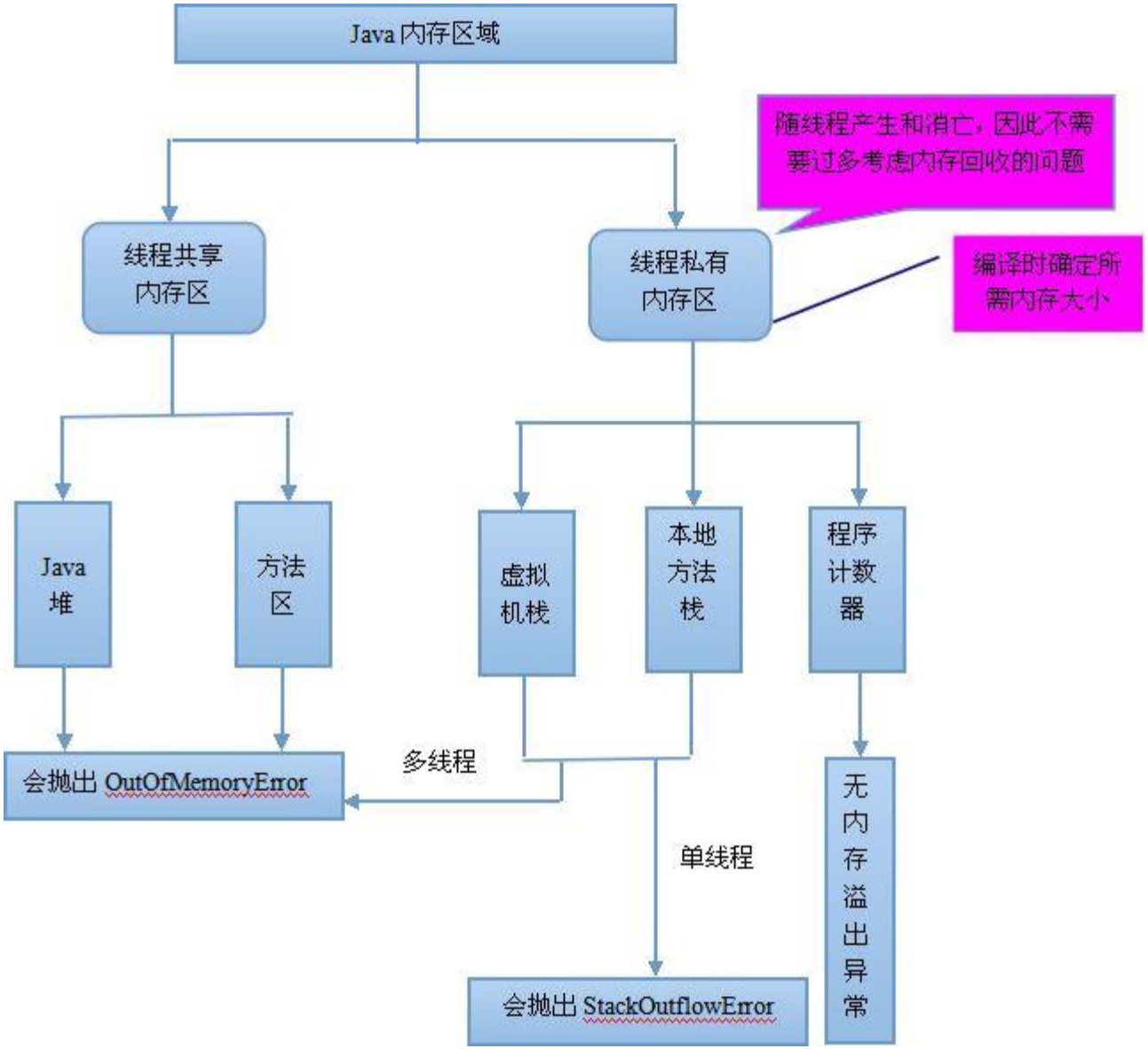


jvm 进程内存逻辑结构

直接内存：不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域；

- ✓ 如果使用了NIO,这块区域会被频繁使用，在java堆内可以用directByteBuffer对象直接引用并操作；
- ✓ 这块内存不受java堆大小限制，但受本机总内存的限制，可以通过MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常；

05 你对jvm内存结构了解吗？线程共享与线程私有



06 堆和栈的区别是什么？

功能

- 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char等）以及对象的引用变量，其内存分配在栈上，变量出了作用域就会自动释放；
- 而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；

线程独享还是共享

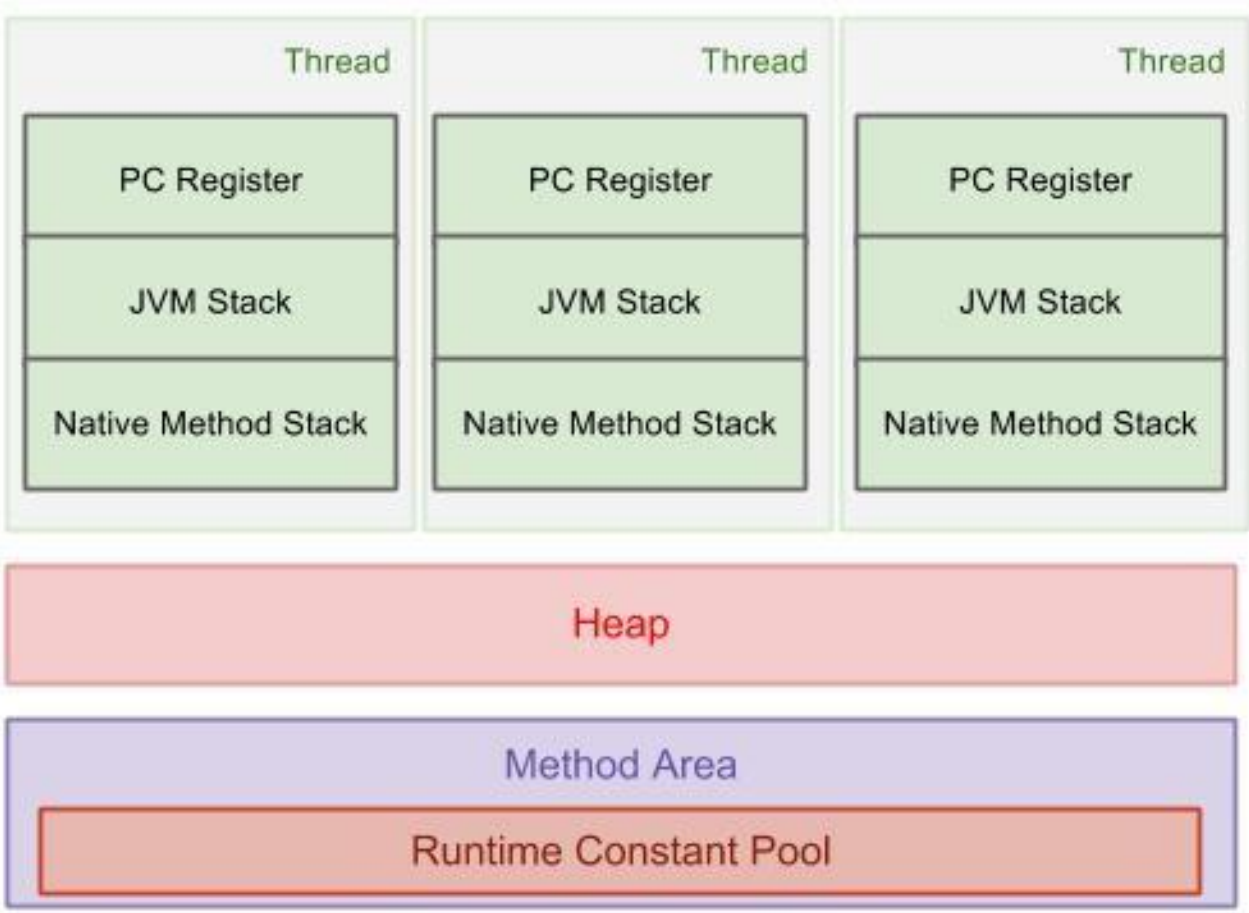
- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
- 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

07 堆和栈的区别是什么？

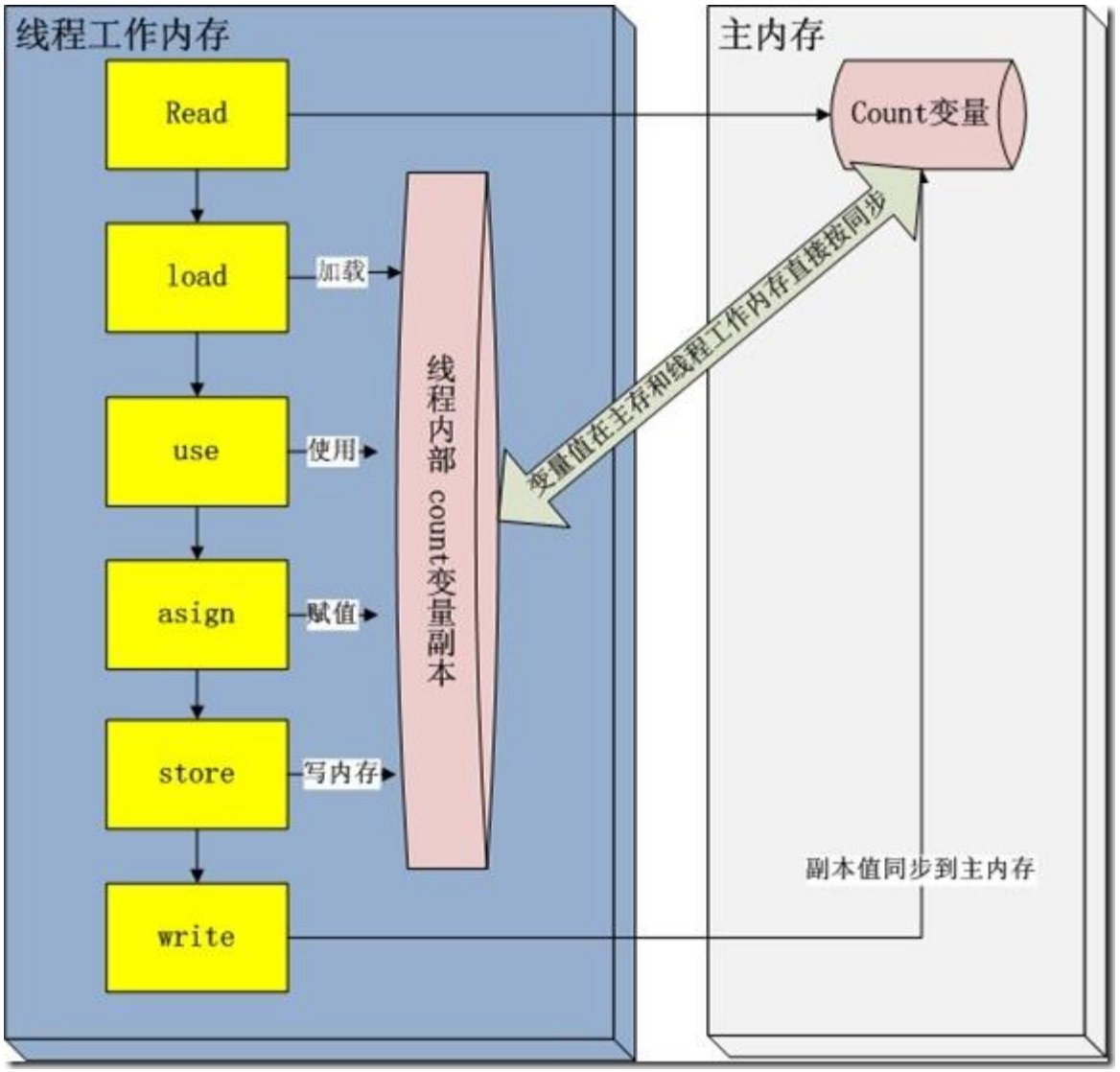
空间大小

- 栈的内存要远远小于堆内存，栈的深度是有限制的，如果递归没有及时跳出，很可能发生StackOverFlowError问题。
- 你可以通过-Xss选项设置栈内存的大小。-Xms选项可以设置堆的开始时的大小，-Xmx选项可以设置堆的最大值

08 你对jvm内存结构了解吗？堆和栈

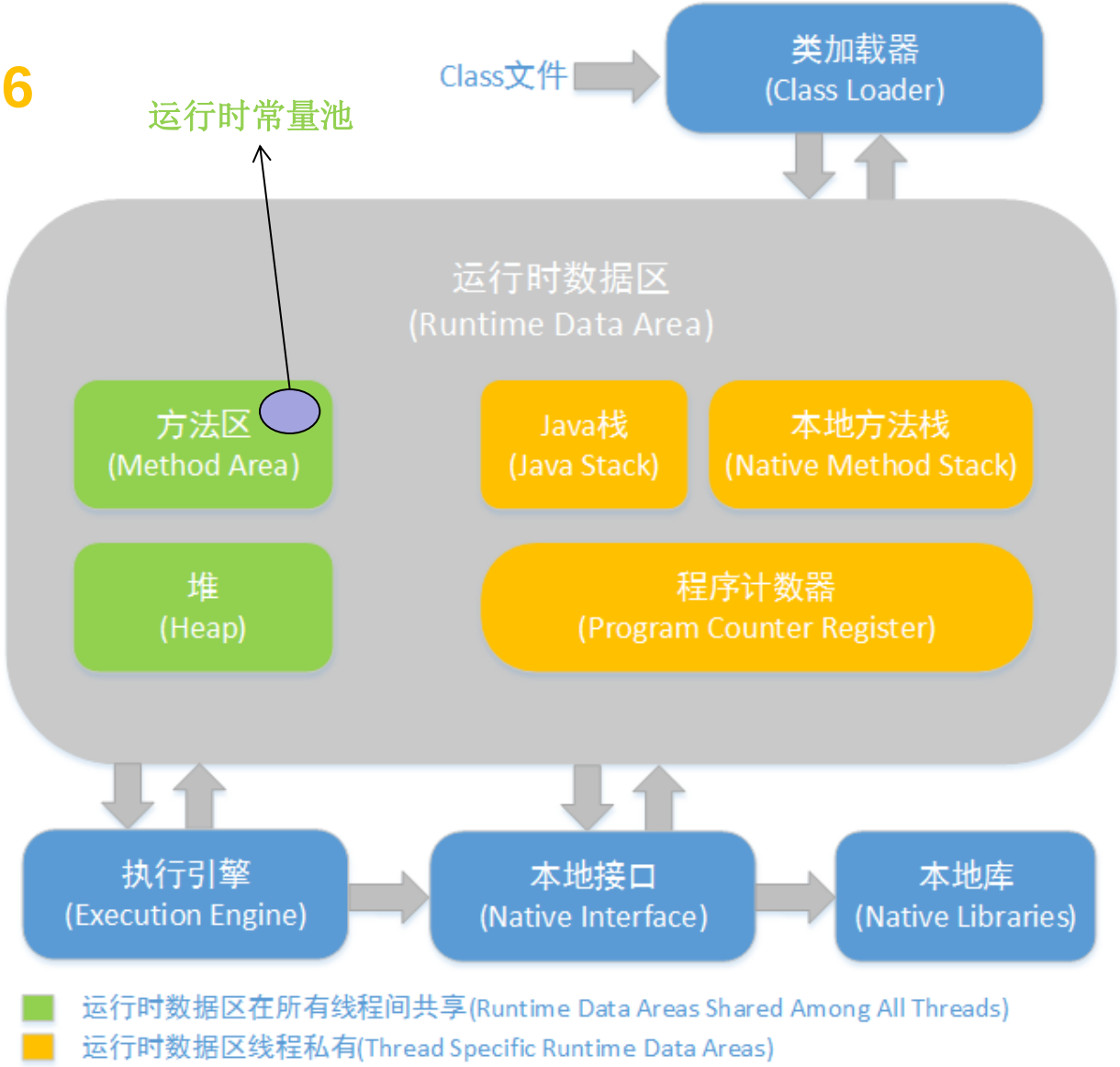


09 你对jvm内存结构了解吗？线程安全的本质



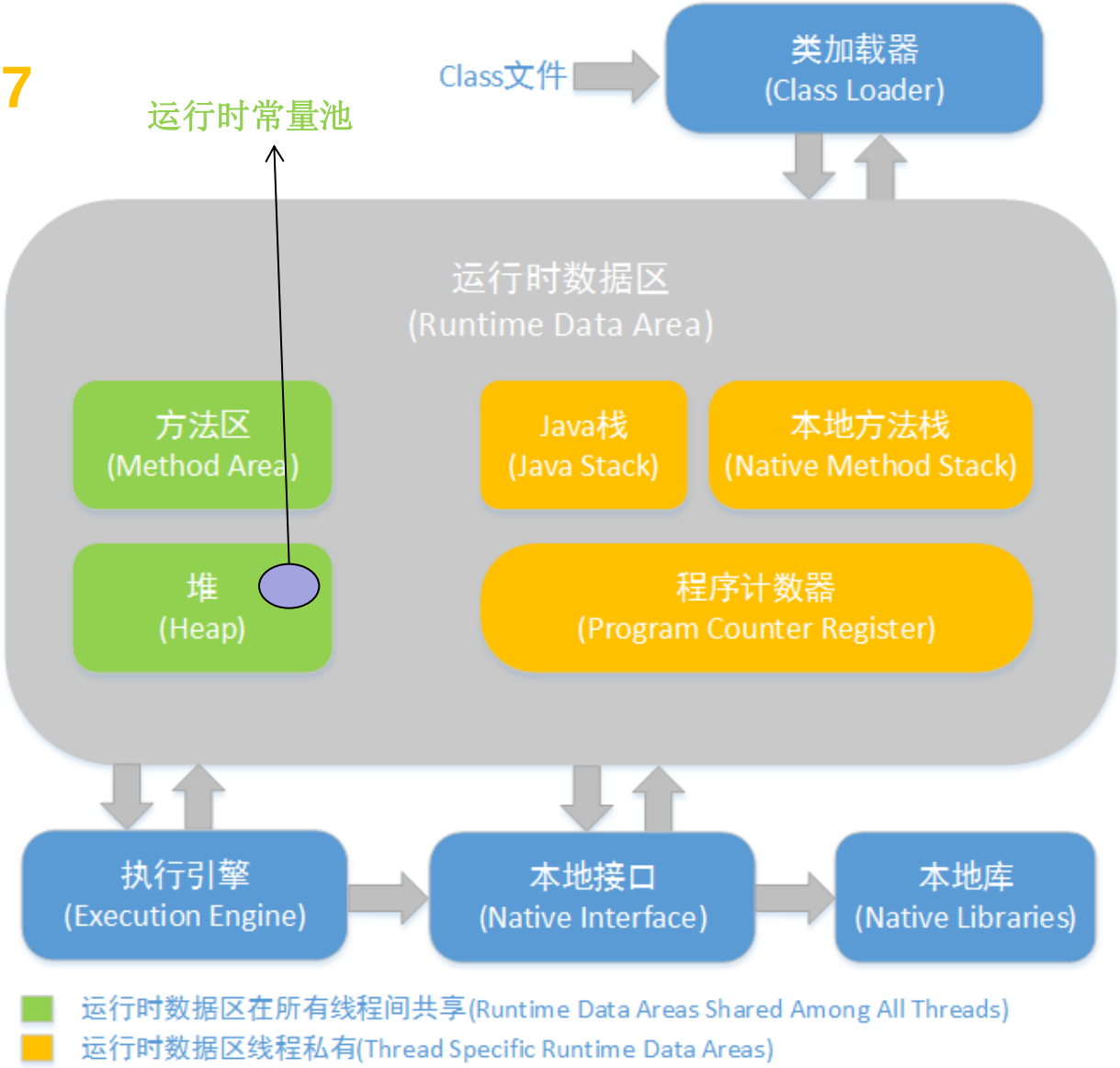
10 jdk1.6、jdk1.7和jdk1.8内存结构区别

jdk1.6



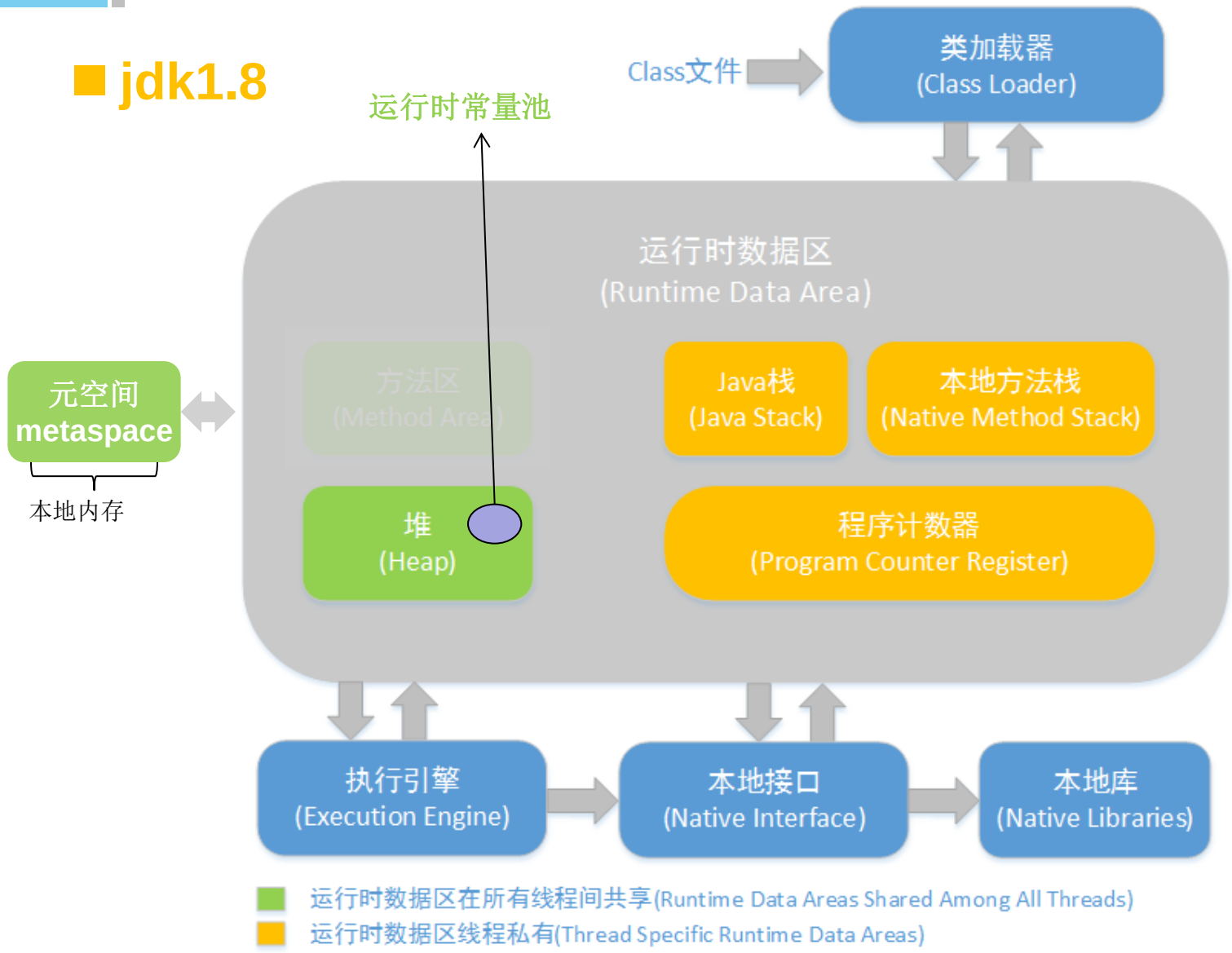
11 jdk1.6、jdk1.7和jdk1.8内存结构区别

jdk1.7

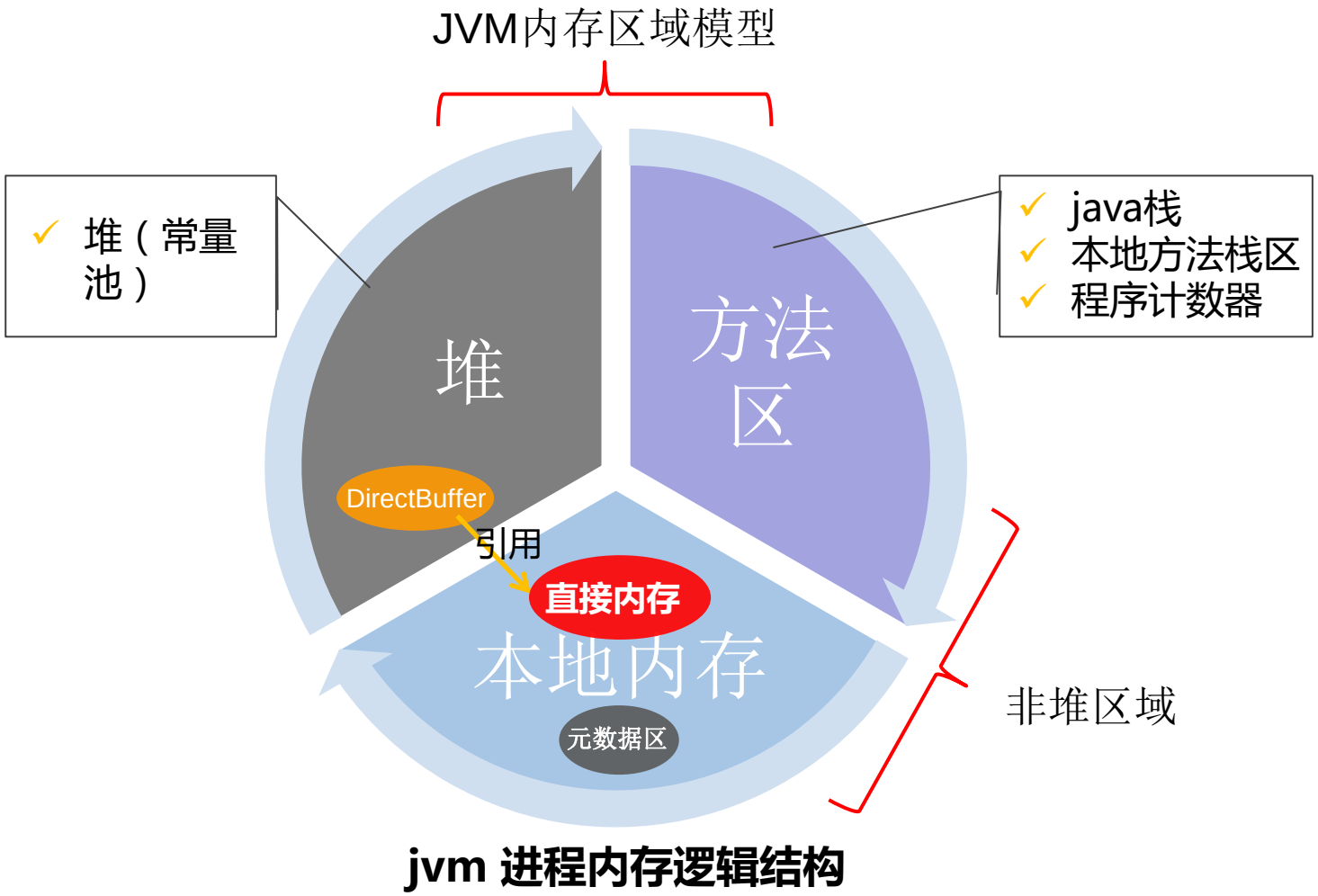


12 jdk1.6、jdk1.7和jdk1.8内存结构区别

jdk1.8



13 jdk1.8的jvm 进程内存逻辑结构

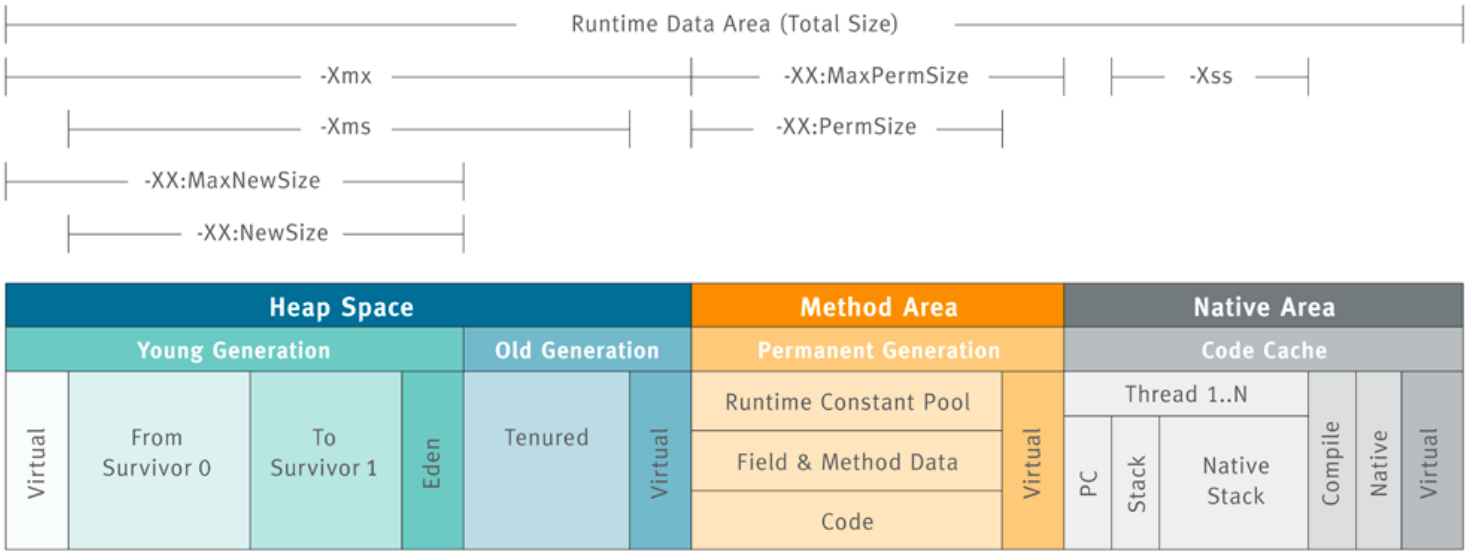


jvm 进程内存逻辑结构

14 为什么去除方法区

- 永久代来存储类信息、常量、静态变量等数据不是个好主意, 很容易遇到内存溢出的问题.JDK8的实现中将类的元数据放入 native memory, 将字符串池和类的静态变量放入java堆中. 可以使用MaxMetaspaceSize对元数据区大小进行调整；
- 对永久代进行调优是很困难的,同时将元空间与堆的垃圾回收进行了隔离，避免永久代引发的Full GC和OOM等问题；

15 jvm常用内存参数设置



C目录 ONTENTS

1 java内存模型问题

2 常见内存溢出异常问题

3 垃圾回收面试问题

01 有哪些java内存溢出异常？

java内存溢出异常主要有两个：

- ✓ OutOfMemeoryError：当堆、栈（多线程情况）、方法区、元数据区、直接内存中数据达到最大容量时产生；
- ✓ StackOverFlowError：如果线程请求的栈深度大于虚拟机锁允许的最大深度，将抛出StackOverFlowError，其本质还是数据达到最大容量；

02 什么情况下出现堆溢出？怎么解决？

- **产生原因**

堆用于存储实例对象，只要不断创建对象，并且保证GC Roots到对象之间有引用的可达，避免垃圾收集器回收实例对象，就会在对象数量达到堆最大容量时产生OutOfMemoryError异常。

java.lang.OutOfMemoryError: Java heap space

- **解决办法**

使用-XX:+HeapDumpOnOutOfMemoryError可以让java虚拟机在出现内存溢出时产生当前堆内存快照以便进行异常分析，主要分析那些对象占用了内存；也可使用jmap将内存快照导出；一般检查哪些对象占用空间比较大，由此判断代码问题，没有问题的考虑调整堆参数；

03 什么情况下出现栈溢出？怎么解决？

- **产生原因**

✓ 如果线程请求的栈深度大于虚拟机锁允许的最大深度，将抛出StackOverFlowError;

✓ 如果虚拟机在扩展栈时无法申请到足够的内存空间，抛出OutOfMemeoryError;

- **解决办法**

✓ StackOverFlowError 一般是函数调用层级过多导致，比如死递归、死循环；

✓ OutOfMemeoryError一般是在多线程环境才会产生，一般用“减少内存的方法”，既减少最大堆和减少栈容量来换取更多的线程支持；

04 什么情况下出现方法区或元数据区溢出？怎么解决？

- **产生原因**

- jdk 1.6以前，运行时常量池还是方法区一部分，当常量池满了以后（主要是字符串变量），会抛出OOM异常；
- 方法区和元数据区还会用于存放class的相关信息，如：类名、访问修饰符、常量池、方法、静态变量等；当工程中类比较多，而方法区或者元数据区太小，在启动的时候，也容易抛出OOM异常；

- **解决办法**

- ✓ jdk 1.7之前，通过-XX:PermSize,-XX:MaxPerSize，调整方法区的大小；
- ✓ jdk 1.8以后，通过-XX:MetaspaceSize ,-XX:MaxMetaspaceSize，调整元数据区的大小；

05 什么情况下出现本机直接内存溢出？怎么解决？

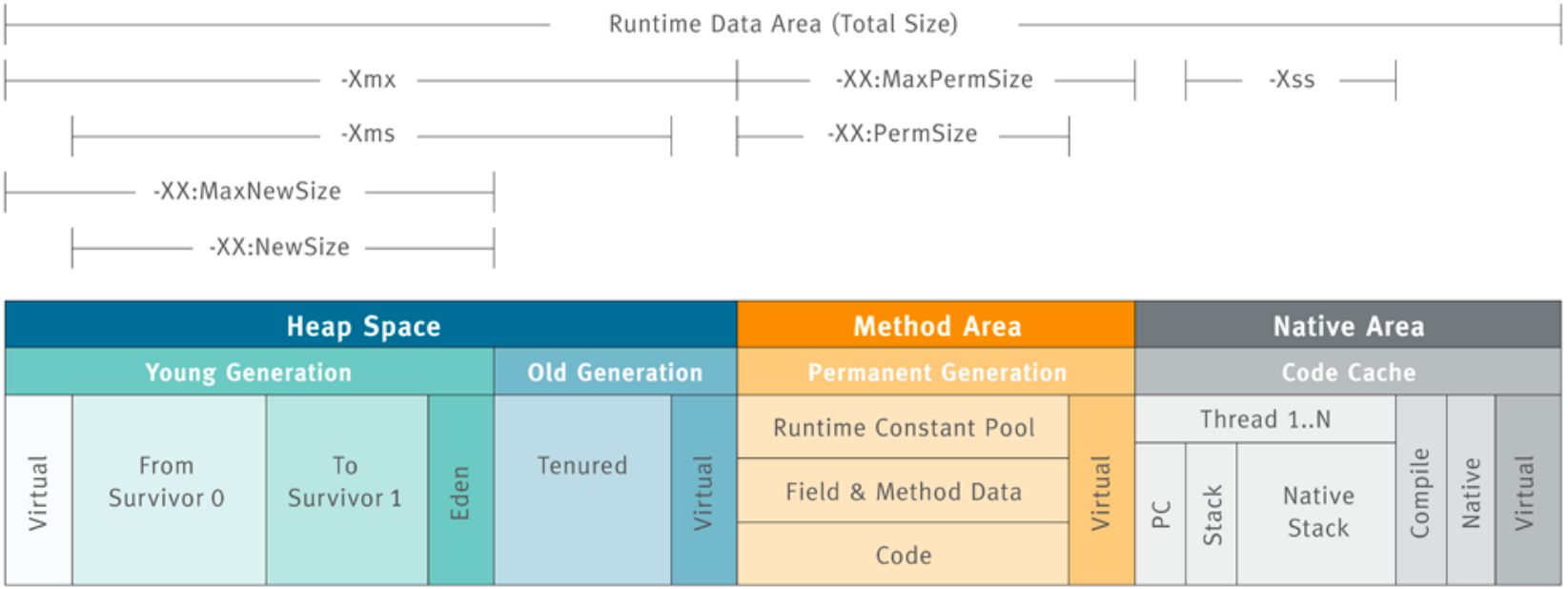
- 产生原因

jdk本身很少操作直接内存，而直接内存（DirectMemory）导致溢出最大的特征是，Heap Dump文件不会看到明显异常，而程序中直接或者间接的用到了NIO；

- 解决办法

直接内存不受java堆大小限制，但受本机总内存的限制，可以通过MaxDirectMemorySize来设置（默认与堆内存最大值一样）

06 jvm常用内存参数设置



C目录 ONTENTS

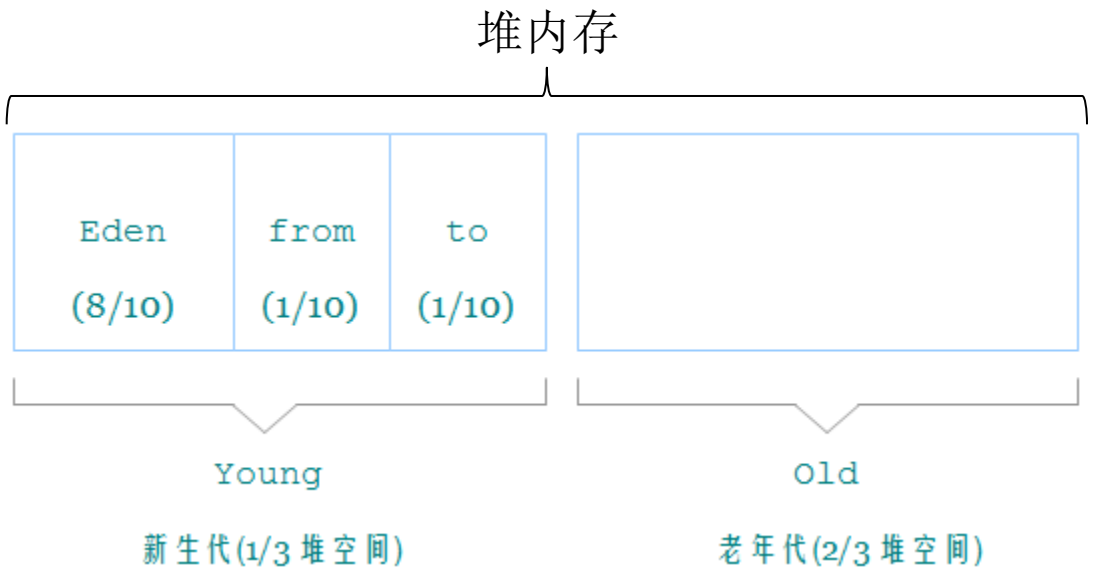
1 java内存模型问题

2 常见内存溢出异常问题

3 **垃圾回收面试问题**

01 关于垃圾回收我们必须要了解的知识

- 垃圾回收主要回收的是堆内存，基于分代的思想：



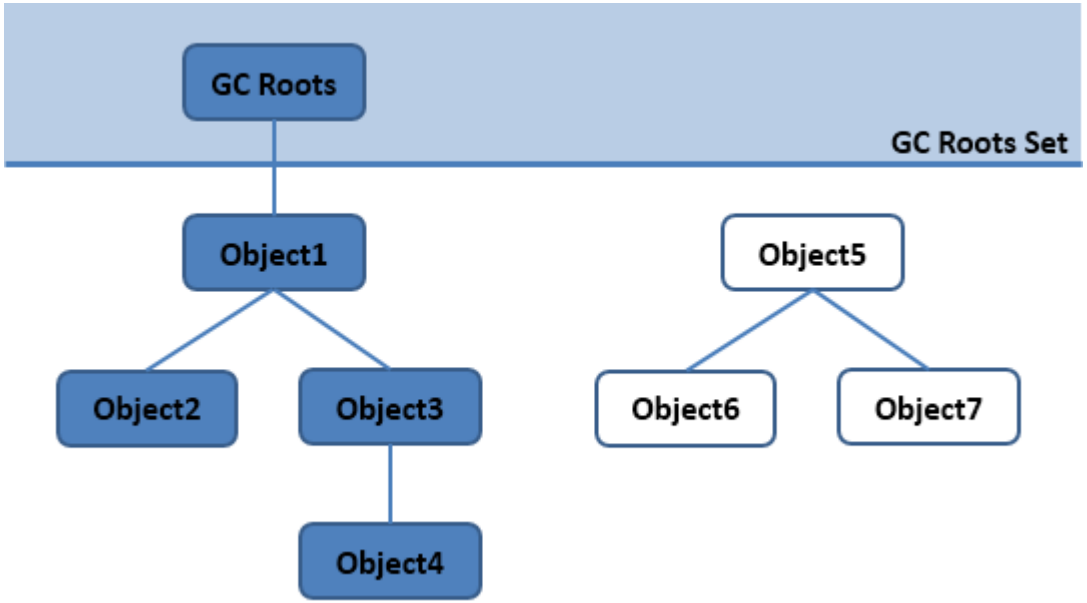
01 内存怎么样分配

- **对象分配**
  - ✓ 优先在Eden区分配。当Eden区没有足够空间分配时, VM发起一次Minor GC, 将Eden区和其中一块Survivor区内尚存活的对象放入另一块Survivor区域。如Minor GC时survivor空间不够，对象提前进入老年代，老年代空间不够时进行Full GC；
  - ✓ 大对象直接进入老年代，避免在Eden区和Survivor区之间产生大量的内存复制, 此外大对象容易导致还有不少空闲内存就提前触发GC以获取足够的连续空间.
- **对象晋级**
  - ✓ 年龄阈值：VM为每个对象定义了一个对象年龄(Age)计数器, 经第一次Minor GC后仍然存活, 被移动到Survivor空间中, 并将年龄设为1. 以后对象在Survivor区中每熬过一次Minor GC年龄就+1. 当增加到一定程度(-XX:MaxTenuringThreshold, 默认15), 将会晋升到老年代.
  - ✓ 提前晋升: 动态年龄判定；如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半, 年龄大于或等于该年龄的对象就可以直接进入老年代, 而无须等到晋升年龄.

01 那些要收回？对象生死判定

- 可达性分析算法

通过一系列的称为 GC Roots 的对象作为起点, 然后向下搜索; 搜索所走过的路径称为引用链/Reference Chain, 当一个对象到 GC Roots 没有任何引用链相连时, 即该对象不可达, 也就说明此对象是不可用的;



在Java, 可作为**GC Roots**的对象包括:

1.方法区: 类静态属性引用的对象;

2.方法区: 常量引用的对象;

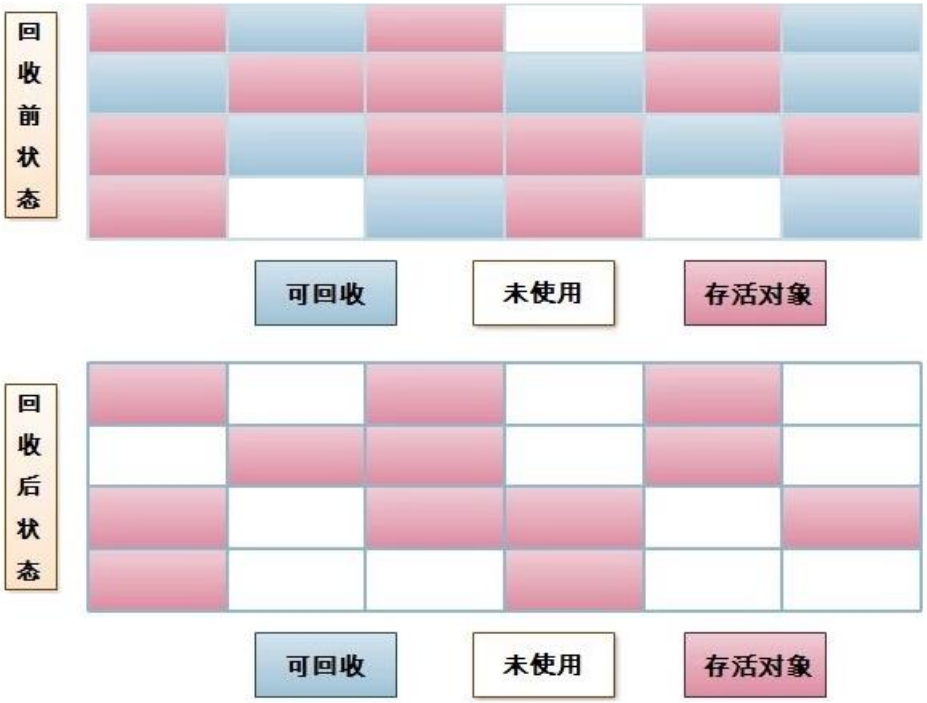
3.虚拟机栈(本地变量表)中引用的对象

4.本地方法栈JNI(Native方法)中引用的对象。

01 怎么回收？方法论？分代收集

新生代-标记清除法

该算法分为“标记”和“清除”两个阶段: 首先标记出所有需要回收的对象(可达性分析), 在标记完成后统一清理掉所有被标记的对象.



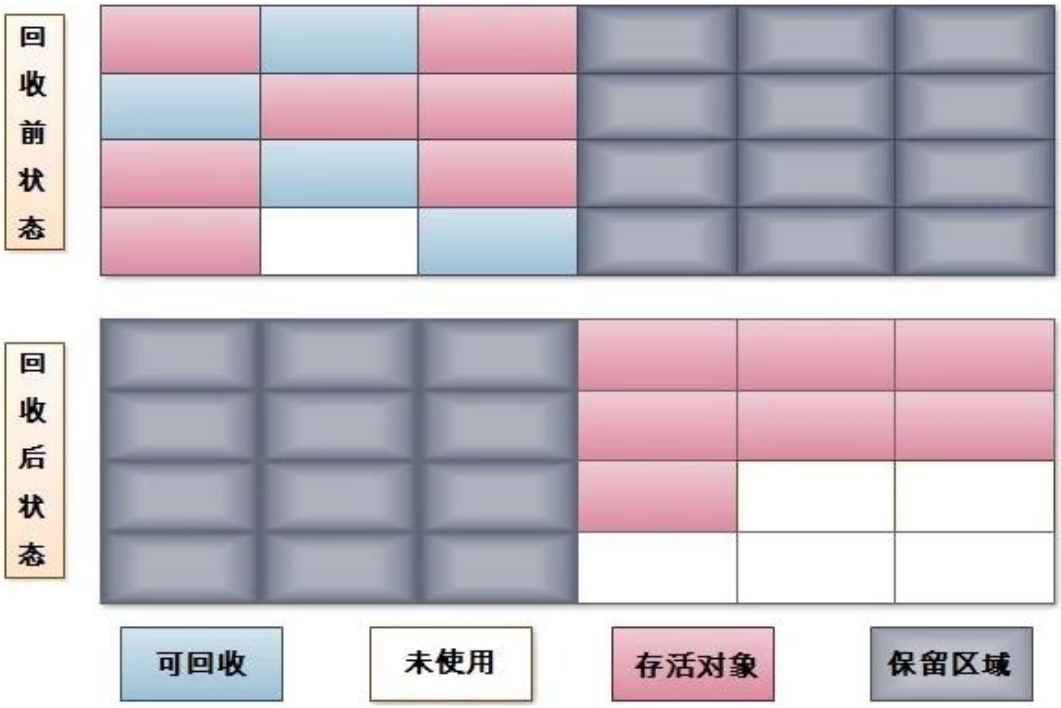
缺点

- ✓ 效率问题: 标记和清除过程的效率都不高;
- ✓ 空间问题: 标记清除后会产生大量不连续的内存碎片, 空间碎片太多可能会导致在运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集.

01 怎么回收？方法论？分代收集

新生代-复制算法

该算法的核心是将可用内存按容量划分为大小相等的两块, 每次只用其中一块, 当这一块的内存用完, 就将还存活的对象复制到另外一块上面, 然后把已使用过的内存空间一次清理掉.



优点

- ✓ 由于是每次都对整个半区进行内存回收，内存分配时不必考虑内存碎片问题。
- ✓ 垃圾回收后空间连续，只要移动堆顶指针，按顺序分配内存即可；
- ✓ 特别适合java朝生夕死的对象特点；

缺点

- ✓ 内存减少为原来的一半，太浪费了；
- ✓ 对象存活率较高的时候就要执行较多的复制操作，效率变低；、
- ✓ 如果不使用50%的对分策略，老年代需要考虑的空间担保策略

01 怎么回收？方法论？分代收集

老年代-标记整理算法

该算法分为“标记”和“清除”两个阶段: *首先标记出所有需要回收的对象(可达性分析), 在标记完成后让所有存活的对象都向一端移动,然后清理掉端边界以外的内存;*

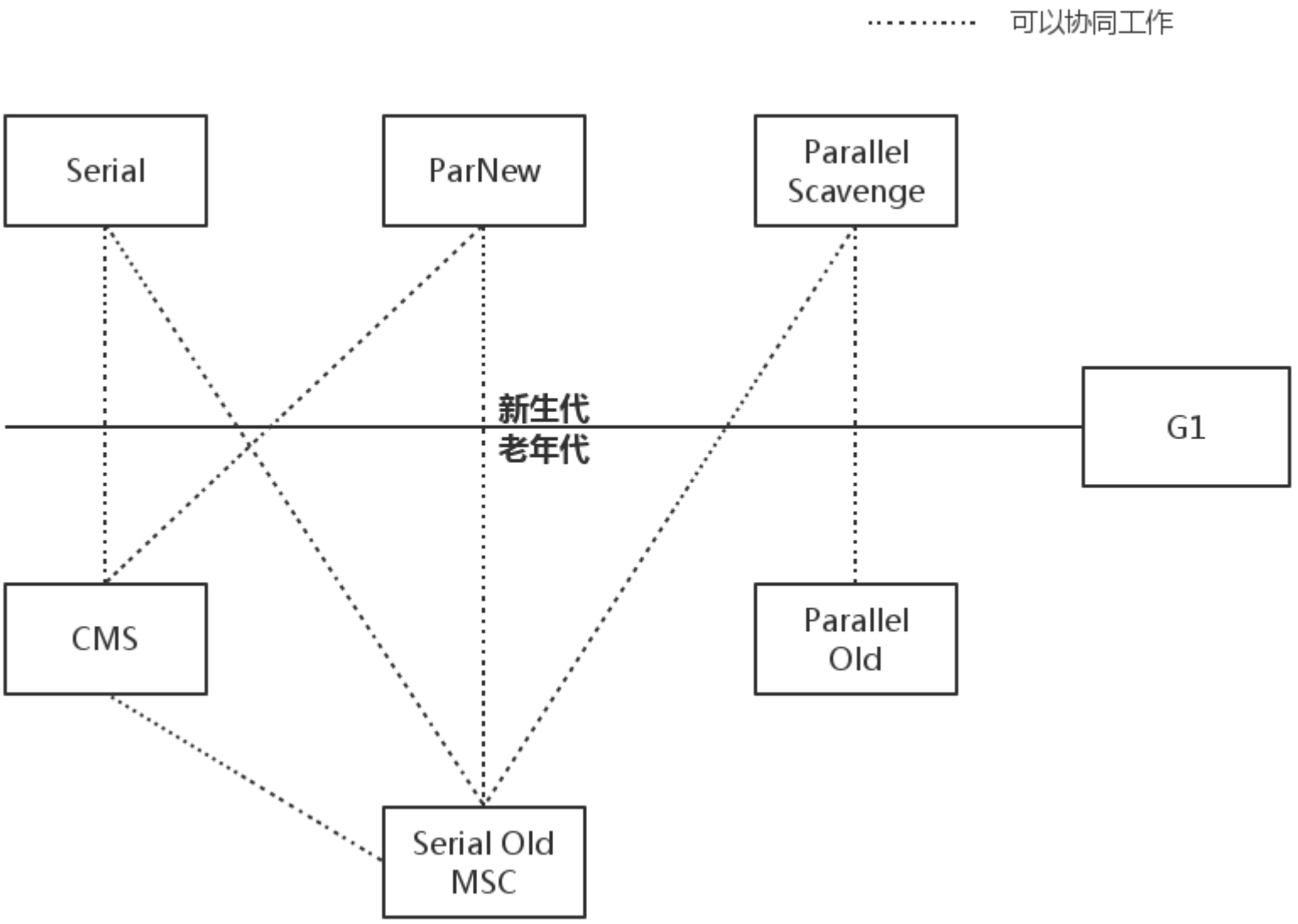
优点

- ✓ 不会损失50%的空间；
- ✓ 垃圾回收后空间连续，只要移动堆顶指针，按顺序分配内存即可；
- ✓ 比较适合有大量存活对象的垃圾回收；

缺点

- ✓ 标记/整理算法唯一的缺点就是效率也不高，不仅要标记所有存活对象，还要整理所有存活对象的引用地址。从效率上来说，标记/整理算法要低于复制算法。

01 实现回收？谁来做？垃圾回收器



01 实现回收？谁来做？垃圾回收器

收集器	收集对象和算法	收集器类型	说明	适用场景
Serial	新生代，复制算法	单线程	进行垃圾收集时，必须暂停所有工作线程，直到完成；(stop the world)	简单高效； 适合内存不大的情况；
ParNew	新生代，复制算法	并行的多线程收集器	ParNew垃圾收集器是Serial收集器的多线程版本	运行在server模式下的虚拟机中首选
Parallel Scavenge 吞吐量优先收集器	新生代，复制算法	并行的多线程收集器	类似ParNew，更加关注吞吐量，达到一个可控制的吞吐量；jdk默认的新生代垃圾回收器	本身是Server级别多CPU机器上的默认GC方式，良好的响应速度能够提升用户的体验，如web后台系统

01 实现回收？谁来做？垃圾回收器

收集器	收集对象和算法	收集器类型	说明	适用场景
Serial Old	老年代，标记整理算法	单线程		Client模式下虚拟机使用
Parallel Old	老年代，标记整理算法	并行的多线程收集器	Parallel Scavenge收集器的老年代版本，为了配合Parallel Scavenge的面向吞吐量的特性而开发的对应组合；jdk默认的老生代垃圾回收器	在注重吞吐量以及CPU资源敏感的场合采用
CMS	老年代，标记清除算法	并发收集器	尽可能的缩短垃圾收集时用户线程停止时间； 缺点在于： 1.内存碎片 2.需要更多cpu资源 3.需要更大的堆空间	重视服务的响应速度、系统停顿时间和用户体验的互联网网站或者B/S系统。互联网后端目前cms是主流的垃圾回收器；
G1	跨新生代和老年代	并行与并发收集器	JDK1.7才正式引入，采用分区回收的思维，基本不牺牲吞吐量的前提下完成低停顿的内存回收	面向服务端应用的垃圾回收器

01 JVM垃圾回收面试常见面试题

- **JVM垃圾回收面试常见面试题**

✓ 垃圾回收常用的算法有哪些？特点是什么？（见垃圾回收算法）

✓ 哪几种垃圾收集器，各自的优缺点，重点讲下cms（见垃圾回收器）

✓ jvm中一次完整的GC流程（从ygc到fgc）是怎样的，重点讲讲对象如何晋升到老年代等（见内存怎么样分配）

✓ JVM垃圾回收机制，何时触发MinorGC或FullGC等操作

答：从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC，对老年代GC称为Major GC,而Full GC是对整个堆来说；

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：
- ✓ System.gc()
- ✓ 老年代空间不足
- ✓ 永生区空间不足
- ✓ 统计得到的Minor GC晋升到旧生代的平均大小大于老年代的剩余空间
- ✓ 堆中分配很大的对象