一、JVM那些事儿

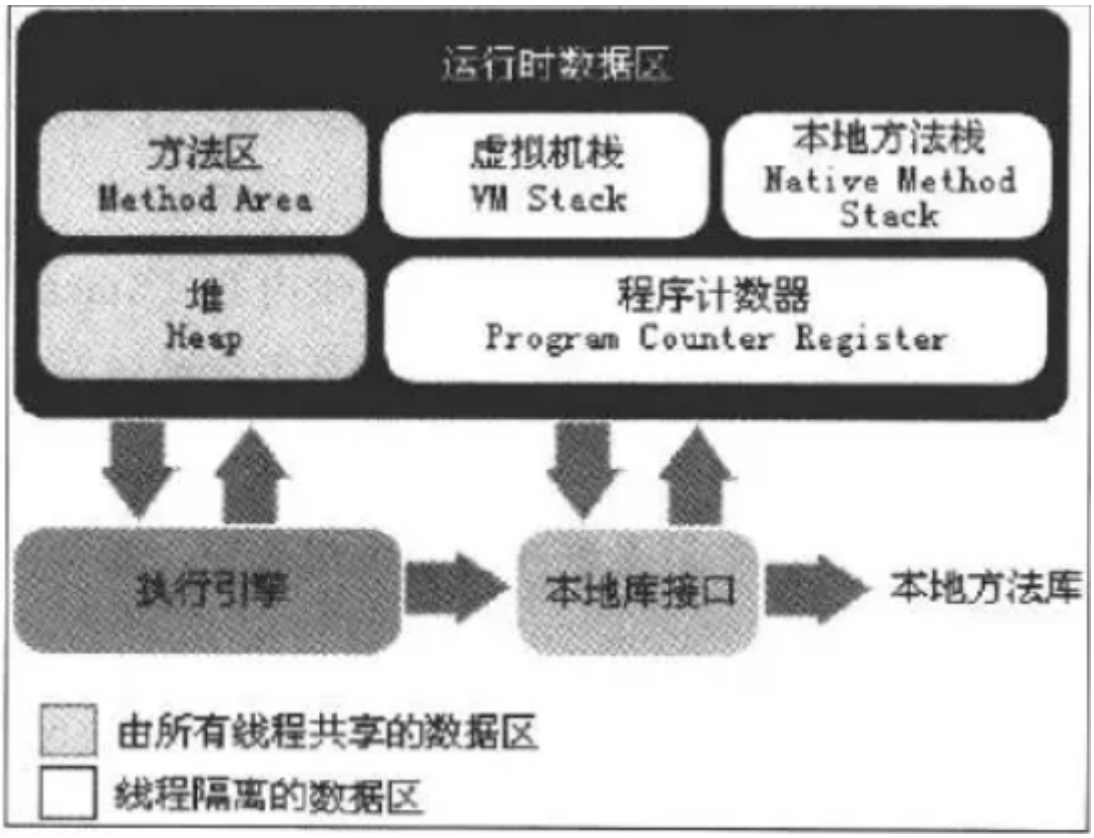
1. Java内存区域

1-1. 1.概述

对于Java程序员，在虚拟机自动内存管理机制下，不再需要像C/C++程序员，为每一个new操作去写对应的delete操作，不容易出现内存泄漏和内存溢出问题

1-2. 2.运行时数据区域

Java虚拟机在执行Java程序的过程中会将它管理的内存分成若干个不同的数据区域



1-2-1. 2.1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。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令。

为了线程切换后能恢复到正确的位置，每条线程都需要有独立的程序计数器，各线程之间计数器互不影响。因此程序计数器应该是"线程私有"的内存

1-2-2. 2.2虚拟机栈

Java虚拟机栈描述的是Java方法执行的内存模型。 Java内存可以粗糙地分为堆内存和栈内存，其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分 (局部变量表主要存放了编译器可知的数据类型和对象引用)。与程序计数器相同，Java虚拟机栈也是线程私有的，生命周期和栈相同。

1-2-3. 2.3本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是： 虚拟机栈为虚拟机执行java方法，而本地方法栈则为虚拟机使用到的Native方法(一般为C++方法)服务

1-2-4. 2.4堆

堆是Java虚拟机所管理的内存中最大的一块，java堆是所有线程共享的一块内存区域，在虚拟机启动时创建。 此内存区域的唯一目的是存放对象实例，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，因此也被叫做GC堆。从垃圾回收的角度看，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：新生代和老年代等。

1-2-5. 2.5方法区

方法区和Java堆一样，是各个线程共享的内存区域， 用于存储已被虚拟机加载的类信息、常量、静态变量、以及编译器编译后的代码等数据

HotSpot虚拟机中方法区也常被称为"永久代",本质上两者并不等价。仅仅是因为HotSpot虚拟机设计团队用永久代来实现方法区而已，这样HotSpot虚拟机的垃圾收集器就可以像管理Java堆一样管理这部分内存了。

1-2-6. 2.6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、还有常量池等信息(用于存放编译期生成的各种字面量和符号引用)

1-2-7. 2.7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。

1-3. 3.HotSpot虚拟机对象探秘

1-3-1. 3.1对象的创建

虚拟机遇到一条new指令，首先先去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已经被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。
在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。**分配方式有"指针碰撞"和“空闲列表”两种，选择哪种分配方式区取决于Java堆是否规整，而Java堆是否规整取决于所采用的垃圾收集器是否带有压缩整理功能**。

接下来，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的GC分代年龄等信息。这些信息存放在对象头中，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会与不同的设置方式。 new指令执行完后，再按照程序员的意愿执行init方法后一个真正可用的对象才诞生。

1-3-2. 3.2对象的内存布局

在Hotspot虚拟机中，对象在内存中的布局可以分为三块区域：**对象头、实例数据和对齐填充**

- 对象头：Hotspot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身运行时数据(哈希码，GC分代年龄，锁状态标志)，另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例
- 实例数据：实例数据是对象真正存储的有效信息，也是在程序中定义的各种类型的字段内容

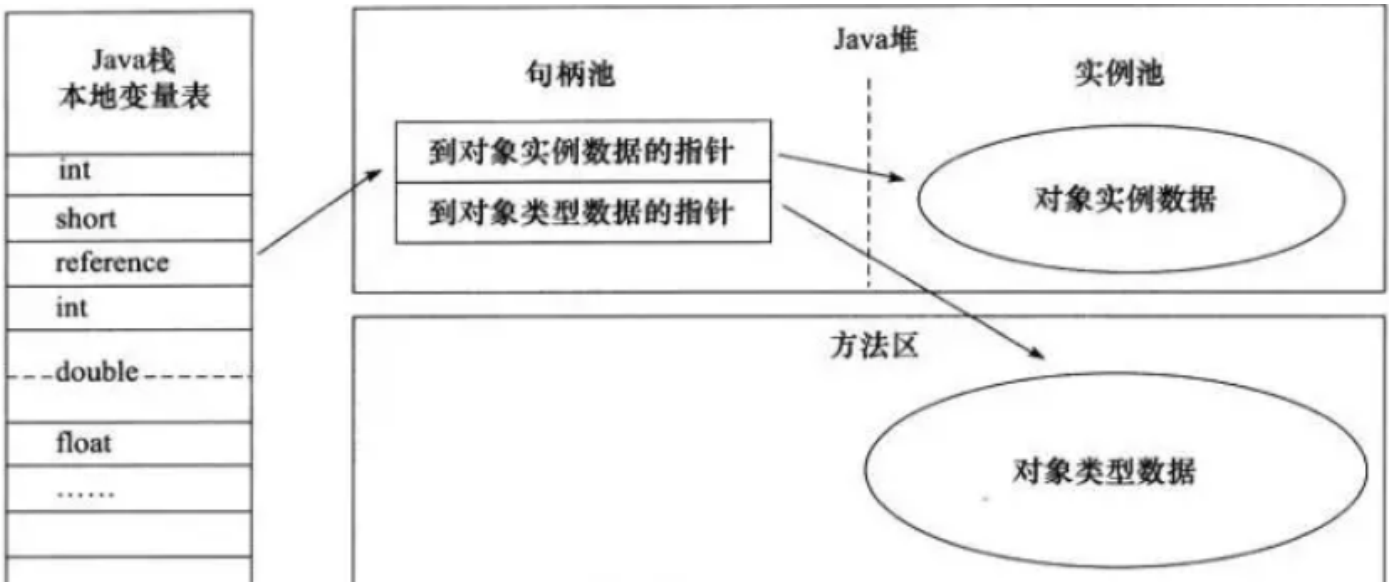
- 对齐填充：对齐填充不是必然存在的，也没有什么特别的含义，仅仅起到占位的作用。因为Hotspot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，换句话说就是对象的大小必须是8字节的整数倍。而对象头部分正好是8字节的倍数（1倍或2倍），因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

1-3-3. 3.3对象的访问定位

建立对象就是为了使用对象，Java程序通过栈上的reference数据来操作堆上的具体对象。对象的访问方式由虚拟机实现而定，主要有"使用句柄"和“直接指针”两种。

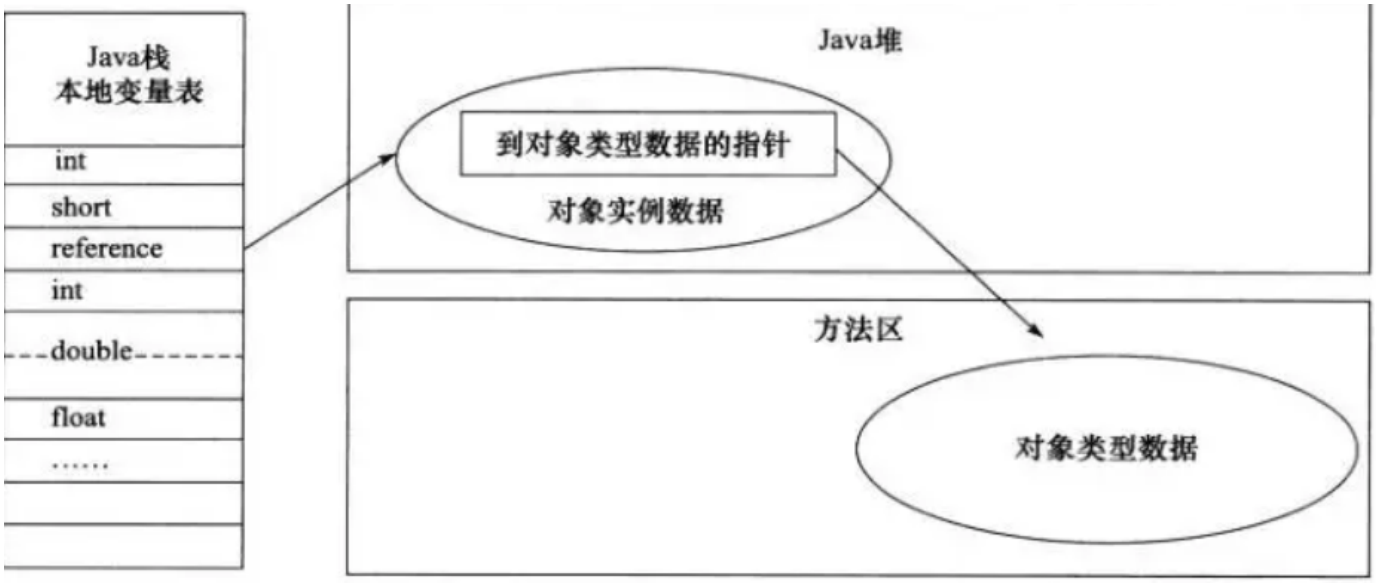
1-3-3-1. 句柄

如果使用句柄的话，Java堆将会划分出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象的实例数据于类型数据各自得具体地址信息



1-3-3-2. 直接指针

如果使用直接指针，栈中的reference存储的就是对象的地址



使用句柄访问的最大好处是reference中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改；使用直接指针访问的最大好处就是速度快，它节省了一次指针定位的时间开销

---

2. 垃圾回收

2-1. 1.如何判断对象已经死亡?

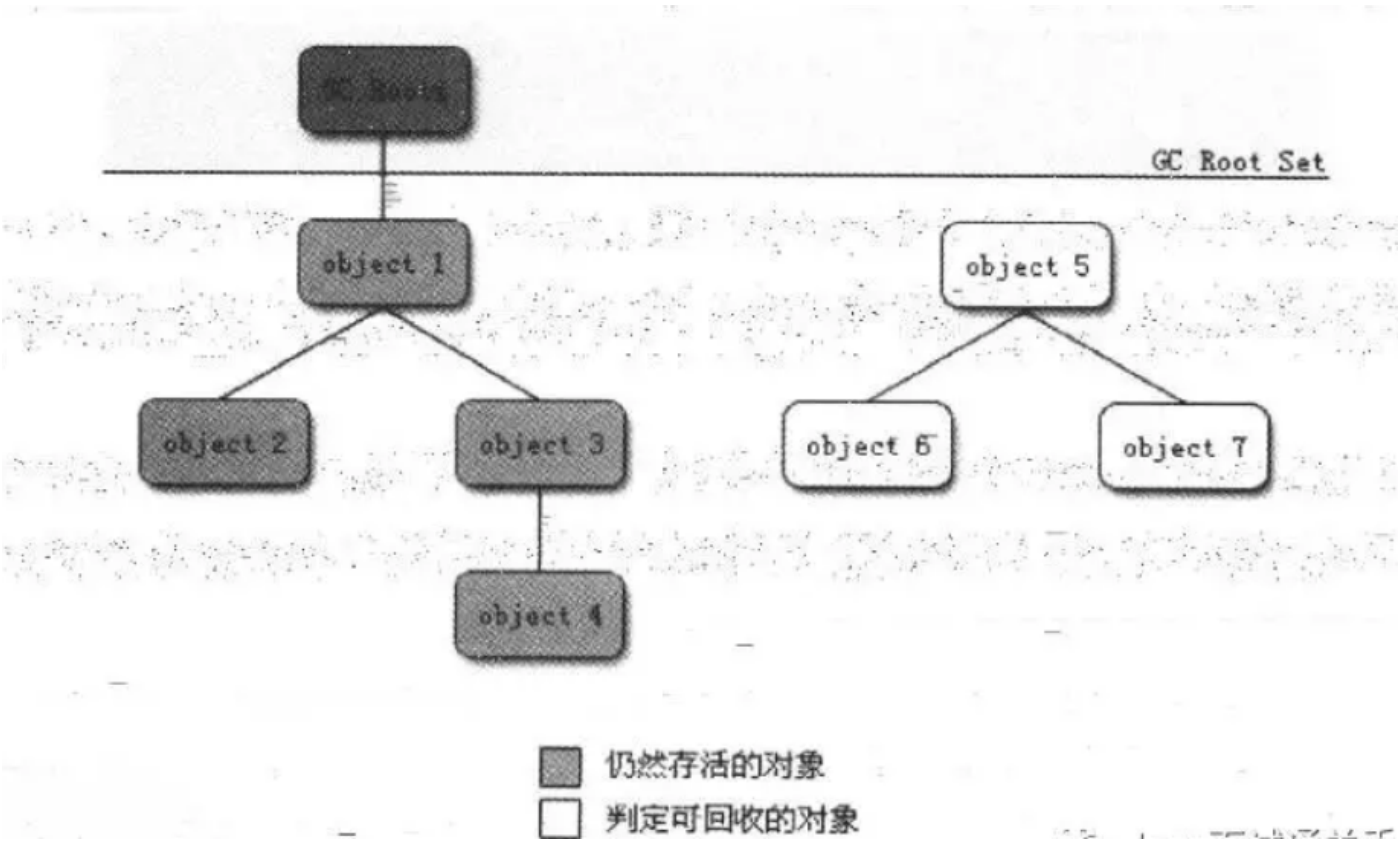
堆中存放着几乎所有的对象实例，对堆的垃圾回收前的第一步就是判断哪些对象已经死亡

2-1-1. 1.1引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。

2-1-2. 1.2可达性分析算法

这个算法的基本思想就是通过一系列的称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连的话，则证明此对象是不可用的



即使在可达性分析法中不可达的对象，也并非是"非死不可"的，这时候它们暂时处于"缓刑阶段"。要真正宣告一个对象死亡，至少要经历两次标记过程。可达性分析法中不可达对象被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要指向finalize方法。当对象没有覆盖finalize方法，或finalize方法已经被虚拟机调用过时，虚拟机将这两种情况视为没有必要执行。被判定为需要执行的对象将会被放在一个队列中进行第二次标记，除非这个对象与引用链上的任何一个对象建立关联，否则就真的被回收。

> 垃圾回收器准备释放内存时，会先调用finalize()，但是此时对象不一定会被回收。
> 垃圾回收和finalize()都是靠不住的，除非JVM内存即将耗尽，否则是不会浪费时间进行垃圾回收的

2-2. 2.谈谈引用

JDK1.2以后，Java对引用的概念进行了补充，将引用分为强引用、软引用、弱引用、虚引用四种

2-2-1. 2.1强引用

如果一个对象具有强引用，那就类似于生活必备品，垃圾回收器是绝不会回收它的。我们使用的大部分引用是强引用，这是使用最普遍的引用。当内存不足时，JVM宁愿抛出OutOfMemoryError错误使程序异常终止，也不会靠随意回收有强引用的对象来解决内存不足的问题。

2-2-2. 2.2软引用

如果一个对象只具有软引用，那就类似于可有可物的生活用品。**如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用**。软引用可用来实现内存敏感的高速缓存。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA虚拟机就会把这个软引用加入到与之关联的引用队列中。

2-2-3. 2.3弱引用

如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，**一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存**。不过，**由于垃圾回收器是一个优先级很低的线程**， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

2-2-4. 2.4虚引用

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

> **虚引用主要用来跟踪对象被垃圾回收的活动**
> 虚引用必须和引用队列（ReferenceQueue）联合使用。当垃 圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是 否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

2-3. 3.回收方法区

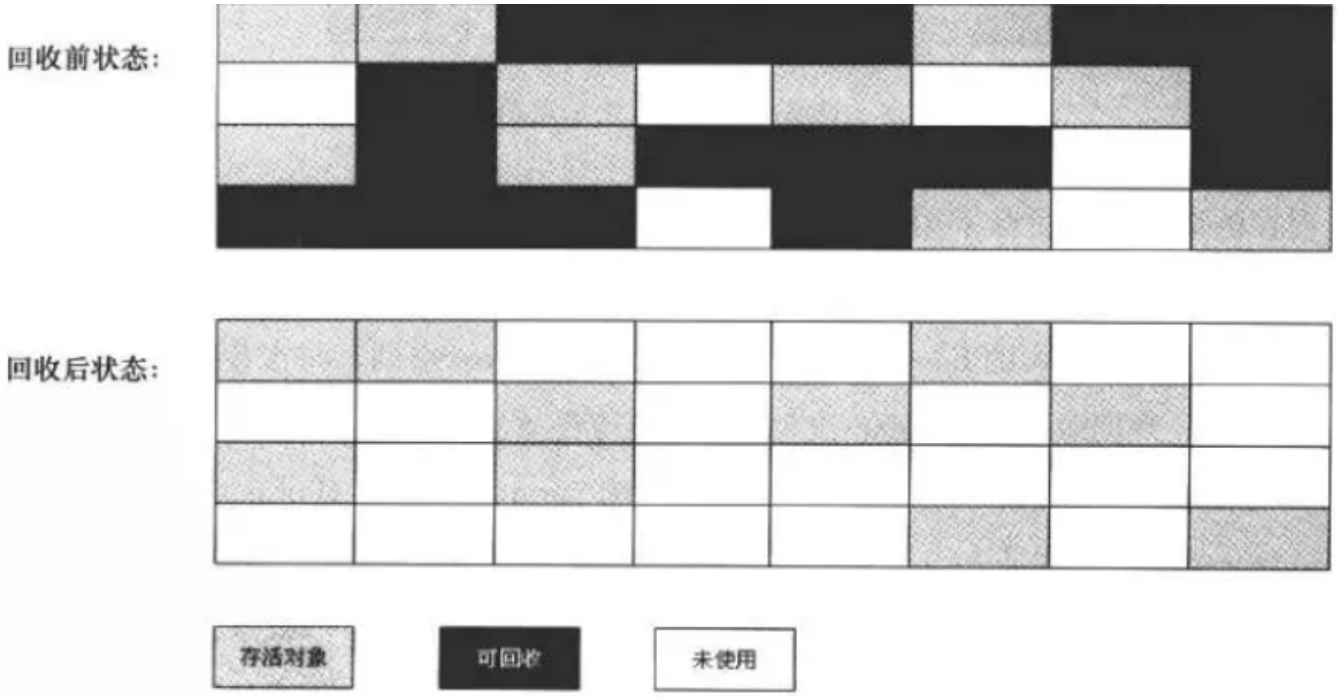
方法区(或Hotspot虚拟机中的永久代)的垃圾收集主要回收两部分内容：废弃常量和无用的类。
判断一个常量是否是“废弃常量”比较简单，而要判定一个类是否是"无用的类"的条件则比较苛刻。
类需要同时满足下面三个条件才能算是"无用的类"：

- 该类的所有实例都已经被回收，Java堆中不存在该类的任何实例
- 加载该类的ClassLoader已经被回收
- 该类对应的java.lang.Class对象已经没有任何地方被引用，无法在任何地方通过反射访问该类的方法

2-4. 4.垃圾收集算法

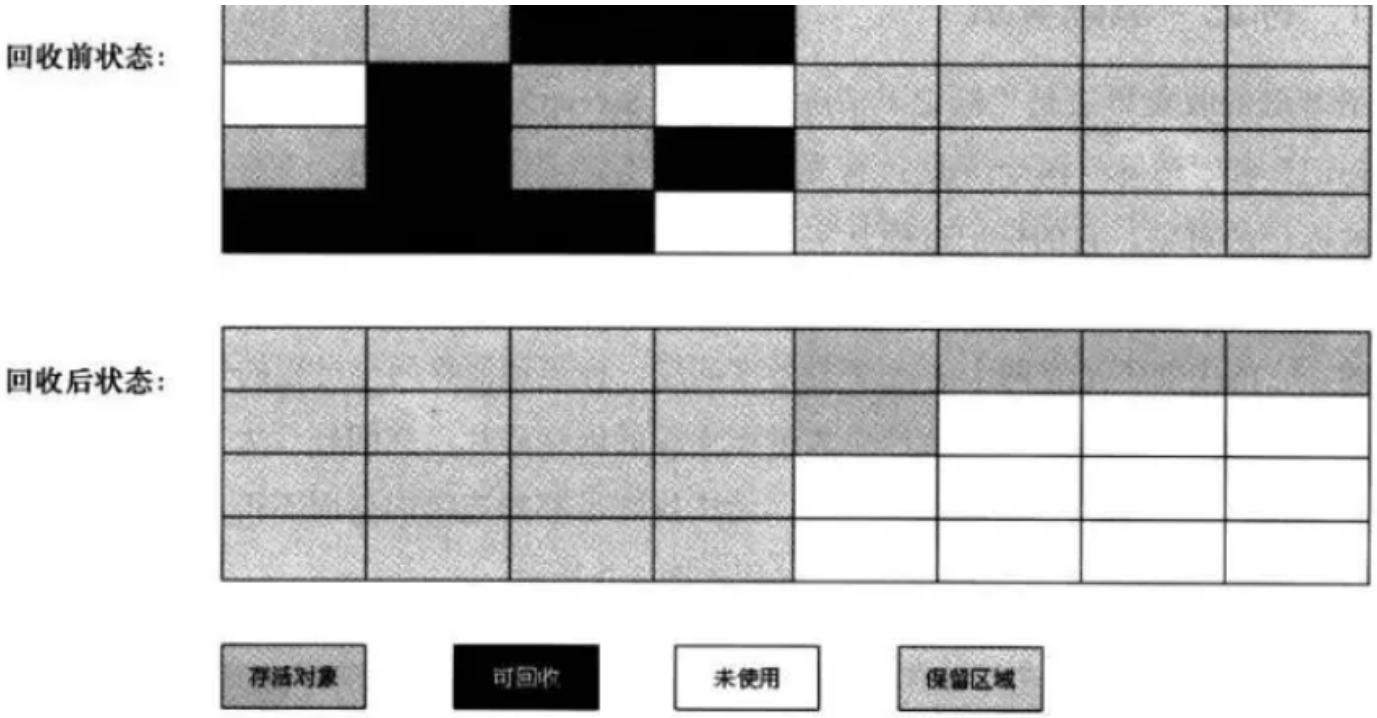
2-4-1. 4.1标记-清除算法

算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，会带来两个明显的问题；1：效率问题和2：空间问题（标记清除后会产生大量不连续的碎片）



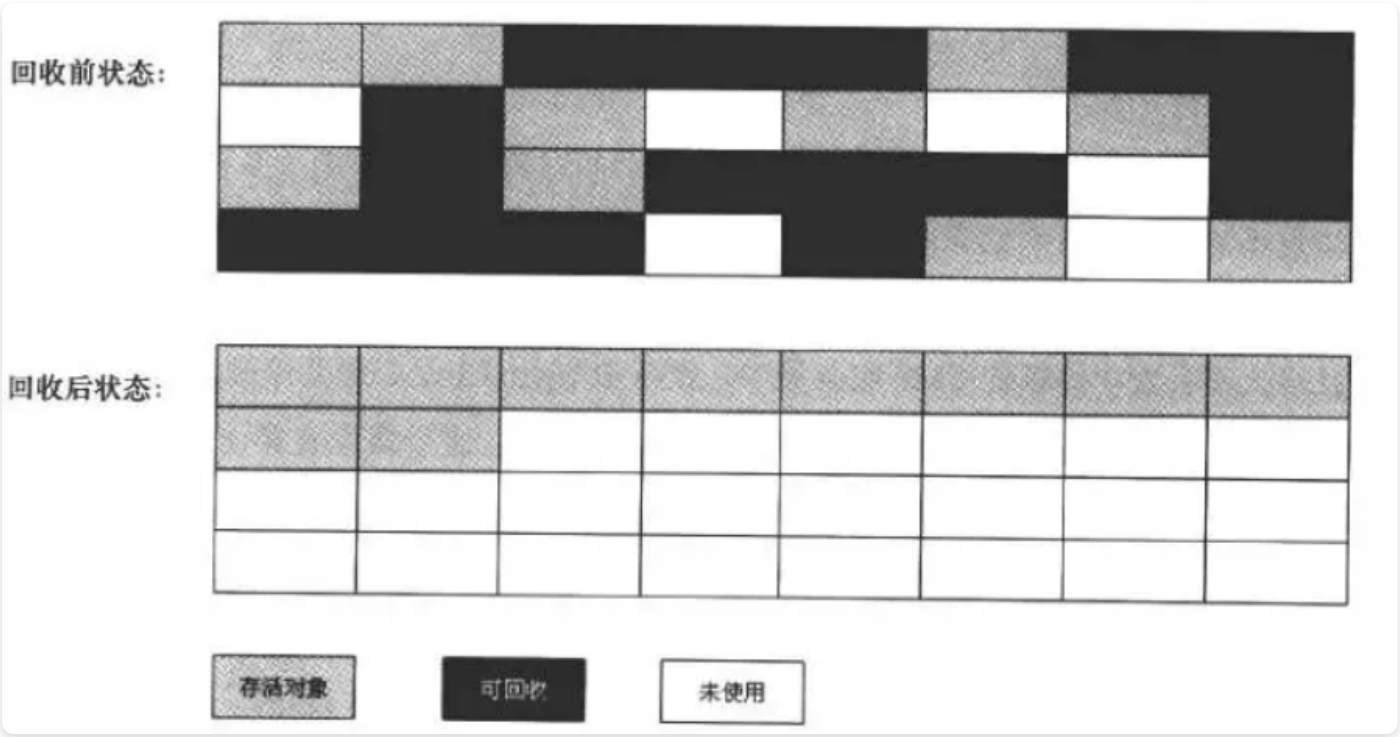
2-4-2. 4.2复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。



2-4-3. 4.3标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一段移动(向一个方向集中)，然后直接清理掉端边界以外的内存



enter description here

2-4-4. 4.4分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。
比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的所以我们可以选择“标记-清理”或“标记-整理”算法进行垃圾收集。

2-4-4-1. 4.4.1新生代和老年代

在Java中，堆被划分为两个不同的区域：新生代和老年代。新生代又划分为三个部分：Edem、From Survivor、To Survivor
堆的内存模型大致如下：

新生代几乎是所有Java对象出生的地方，即 Java 对象申请的内存以及存放都是在这个地方。Java 中的大部分对象通常不需长久存活，具有朝生夕灭的性质。新生代是 GC 收集垃圾的频繁区域。**对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时（ 默认是 15 岁，可以通过参数 -XX:MaxTenuringThreshold 来设定 ），这些对象就会成为老年代**.

> HotSpot为什么要分为新生代和老年代?
> 为了使 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收

2-5. 5.垃圾收集器

如果是垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

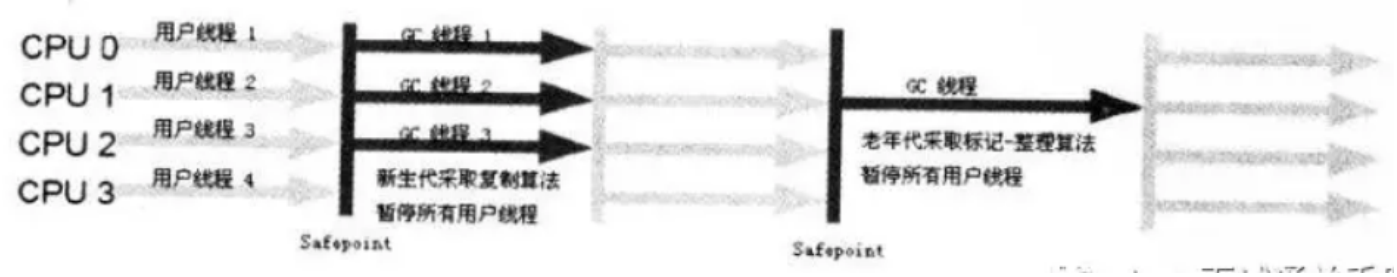
2-5-1. 5.1Serial收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的“单线程”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（ "Stop The World" 了解一下），直到它收集结束。

它简单而高效（与其他收集器的单线程相比）。Serial收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial收集器对于运行在Client模式下的虚拟机来说是个不错的选择。

2-5-2. 5.2ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和Serial收集器完全一样



它是许多运行在Server模式下的虚拟机的首要选择，除了Serial收集器外，只有它能与CMS收集器（真正意义上的并发收集器）配合工作

- 并行(Parallel)：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- 并发(Concurrent)：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个CPU上

2-5-3. 5.3Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的的多线程收集器。

Parallel Scavenge收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的

关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。

2-5-4. 5.4Serial Old收集器

Serial收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用，另一种用途是作为CMS收集器的后备方案

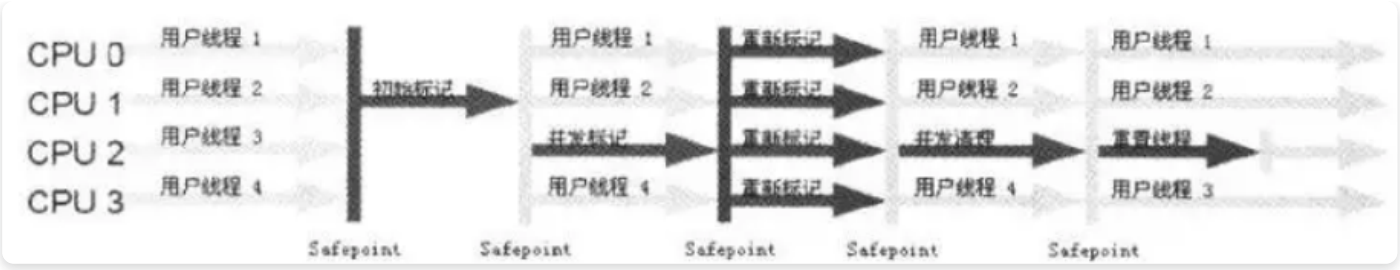
2-5-5. 5.5Parallel Old收集器

Parallel Scavenge收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及CPU资源的场合，都可以优先考虑 Parallel Scavenge收集器和Parallel Old收集器。

2-5-6. 5.6CMS收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。它而非常符合在注重用户体验的应用上使用
从名字中的Mark Sweep这两个词可以看出，CMS收集器是一种 “标记-清除”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- 初始标记： 暂停所有的其他线程，并记录下直接与root相连的对象，速度很快 ；
- 并发标记： 同时开启GC和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以GC线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方
- 重新标记： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- 并发清除： 开启用户线程，同时GC线程开始对为标记的区域做清扫



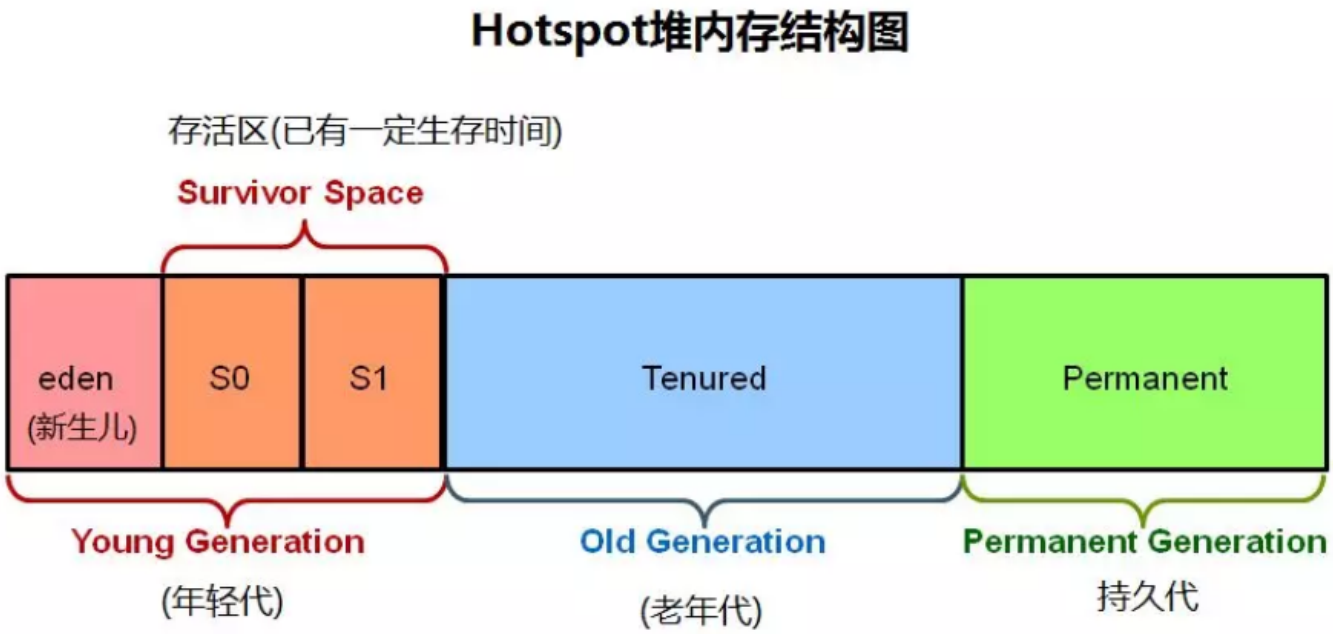
enter description here

从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：并发收集、低停顿。但是它有下面三个明显的缺点：

- 对CPU资源敏感；
- 无法处理浮动垃圾；
- 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生

2-5-7. 5.7G1收集器

上一代的垃圾收集器(串行serial，并行parallel，以及CMS)都把堆内存划分为固定大小的三个部分：年轻代(young generation)，年老代(old generation)，以及持久代(permanent generation)



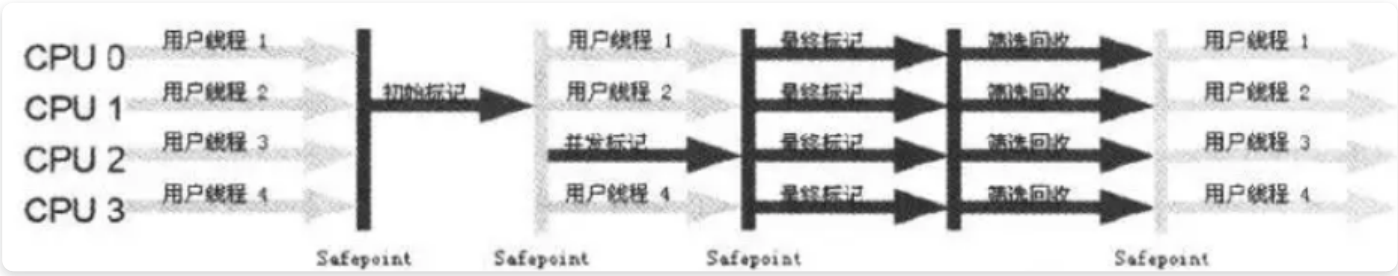
G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

- G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行
- 分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念

- 空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的
- 可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和ＣＭＳ共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内

G1收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region(这也就是它的名字Garbage-First的由来)。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了GF收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）
G1收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收



enter description here

初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS的值，让下一个阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这一阶段需要停顿线程，但是耗时很短，并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段时耗时较长，但可与用户程序并发执行。而最终标记阶段则是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remenbered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这一阶段需要停顿线程，但是可并行执行。最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划

2-6. 6.内存分配与回收策略

大多数情况下，对象在新生代中Eden(新生儿)区分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次GC

2-6-1. 6.1Minor GC 和 Full GC有什么不同?

- 新生代GC(Minor GC):指发生新生代的垃圾收集动作，Minor GC非常频繁，回收速度一般也非常快
- 老年代GC(Major GC/Full GC):指发生在老年代的GC，出现了Major GC经常会伴随至少一次的Minor GC（并非绝对），Major GC的速度一般会比Minor GC的慢10倍以上

2-6-2. 6.2大对象直接进入老年代

大对象就是需要大量连续内存空间的对象（比如：字符串、数组)

2-6-3. 6.3长期存活的对象进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别那些对象应放在新生代，那些对象应放在老年代中。为了做到这一点，虚拟机给每个对象一个对象年龄（Age）计数器

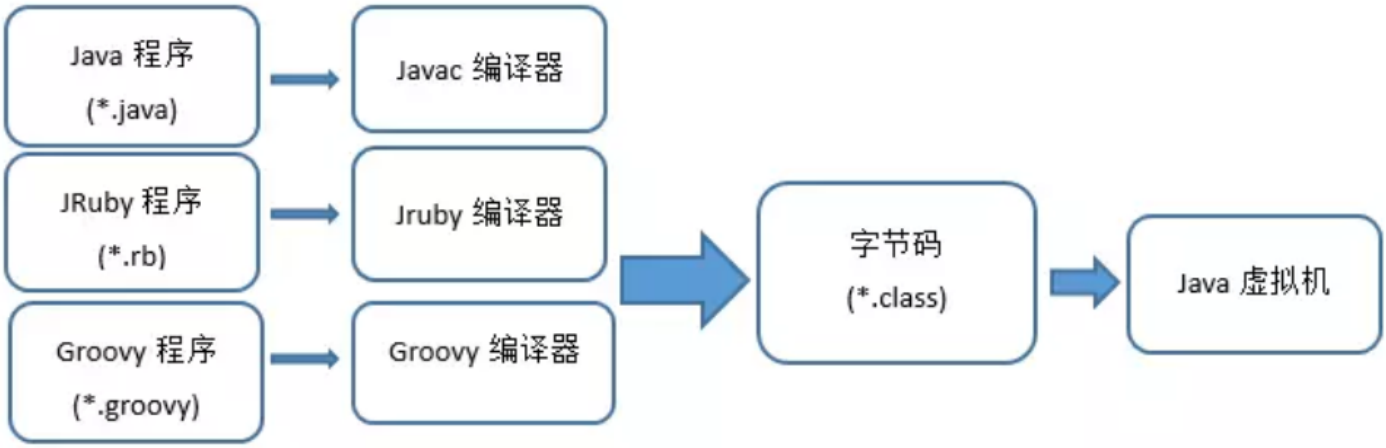
2-6-4. 6.4动态对象年龄判定

为了更好的适应不同程序的内存情况，虚拟机不是永远要求对象年龄必须达到了某个值才能进入老年代，如果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，无需达到要求的年龄

---

3. 3.虚拟机类文件结构

Java虚拟机不和包括Java在内的任何语言绑定，只与class文件这种特定的二进制文件关联，实现了平台无关性。



4. 虚拟机类加载机制

参考自己的博客

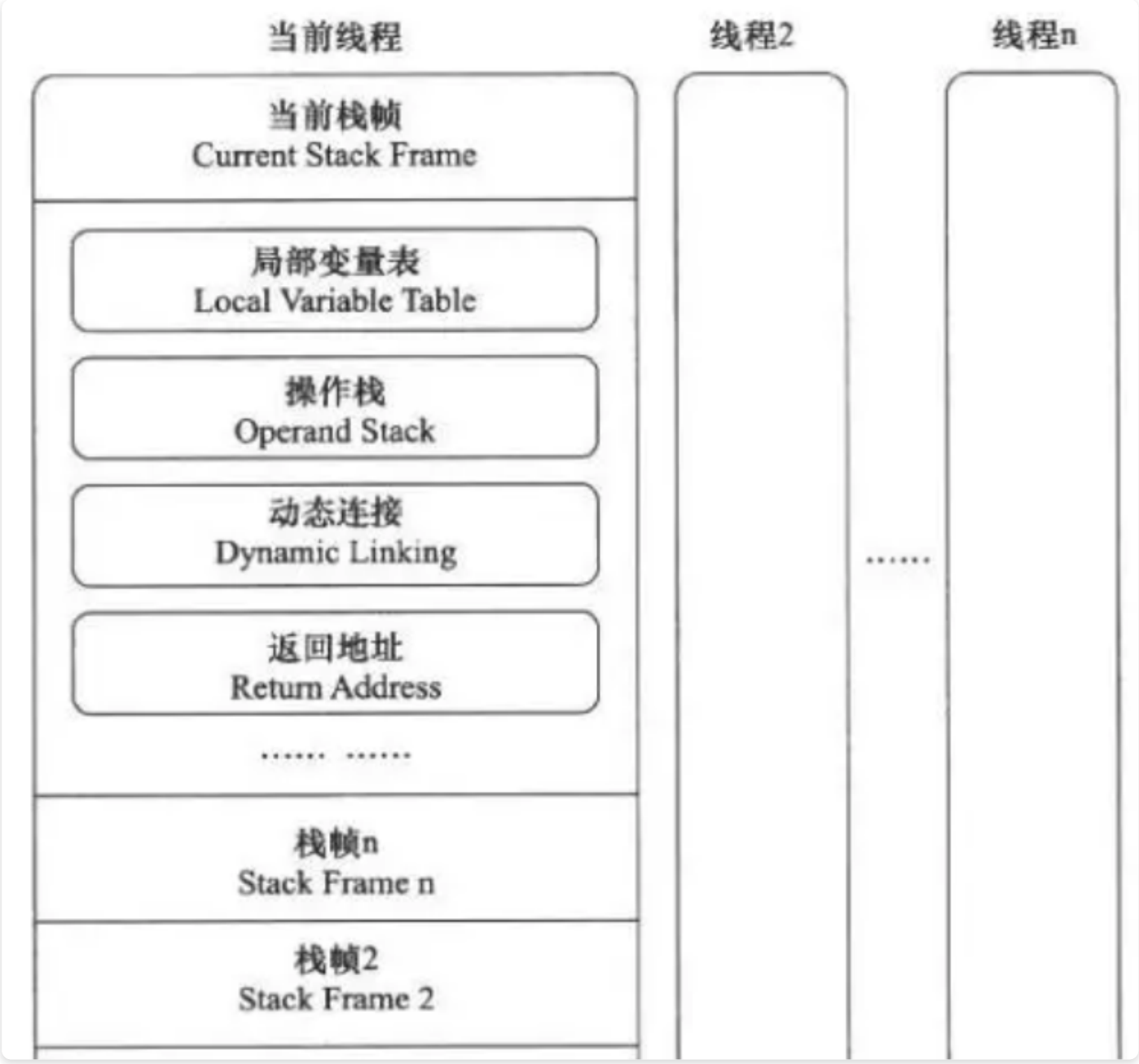
---

5. 4.虚拟机字节码执行引擎

执行引擎是JVM最核心的组成部件之一。所有的Java虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果。

5-1. 4.1运行时栈帧结构

栈帧是用于支持虚拟机方法调用和方法执行的数据结构，它是虚拟机运行时数据区中虚拟机栈的栈元素。
栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。**每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程**。



enter description here

5-2. 4.2局部变量表

局部变量表是一组变量值存储空间，用于存放方法参数和方法内定义的局部变量。
局部变量表的容量以变量槽(Variable Slot)为最小单位。 一个Slot可以存放一个32位以内（boolean、byte、char、short、int、float、reference和returnAddress）的数据类型，reference类型表示一个对象实例的引用。

对于64位的数据类型（Java语言中明确的64位数据类型只有long和double），虚拟机会以高位对齐的方式为其分配两个连续的Slot空间。

虚拟机通过索引定位的方式使用局部变量表，索引值的范围从0开始至局部变量表最大的Slot数量。访问的是32位数据类型的变量，索引n就代表了使用第n个Slot,如果是64位数据类型，就代表会同时使用n和n+1这两个Slot。

5-3. 4.3操纵数栈

操纵数栈也常被称为操作栈，当一个方法开始执行时，这个方法的操作数栈是空的，在方法执行的过程中，会有各种字节码指令往操纵数栈中写入和提取内容，也就是出栈/入栈操作。

在概念模型中，一个活动线程中的两个栈帧是相互独立的。但大多数虚拟机都会做一些优化处理：让下一个栈帧的部分操作数栈与上一个栈帧的部分局部变量表重叠在一起，方便用于共享一部分数据，而无须进行额外的参数复制传递。

5-4. 4.4动态连接

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。
字节码中方法调用指令是以常量池中的指向方法的符号引用为参数的，有一部分符号引用会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为 静态解析，另外一部分在每次的运行期间转化为直接引用，这部分称为动态连接。

5-4-1. 4.4方法返回地址

当一个方法被执行后，有两种方式退出这个方法：

- 第一种是执行引擎遇到任意一个方法返回的字节码指令，这种退出方法的方式称为正常完成出口
- 另外一种是在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，这种退出方式称为异常退出口

无论采用何种退出方式，在方法退出后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的PC计数器的值可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是通过异常处理器表来确定的，栈帧中一般不会保存这部分信息

5-5. 3.方法调用

一切方法调用在Class文件里存储的都只是符号引用，这是需要在类加载期间或者是运行期间，才能确定为方法在实际运行时内存布局中的入口地址

5-5-1. 3.1解析

'编译期可知，运行期不可变"的方法(静态方法和私有方法)，在类加载的解析阶段，会将其符号引用转化为直接引用(入口地址)。这类方法的调用称为”解析“

5-5-2. 3.2分派

分派调用过程中会揭示多态性特征的基本体现如"重载"和"重写"在Java虚拟机是如何实现的

5-5-2-1. 静态分派

静态分派发生在编译阶段，最典型的应用为方法重载。

```
public class StaticDispatch {
    static abstract class Human {
    }
    static class Man extends Human {
    }
    static class Woman extends Human {
    }
    public void sayhello(Human guy) {
        System.out.println("Human guy");
    }
    public void sayhello(Man guy) {
        System.out.println("Man guy");
    }
    public void sayhello(Woman guy) {
        System.out.println("Woman guy");
    }
    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();
        StaticDispatch staticDispatch = new StaticDispatch();
        staticDispatch.sayhello(man);// Human guy
        staticDispatch.sayhello(woman);// Human guy
    }
}
```

运行结果:

Human guy

Human guy

为什么会出现这样的结果呢?

Human man = new Man();其中的Human称为变量的静态类型（Static Type）,Man称为变量的实际类型（Actual Type）。两者的区别是：静态类型在编译器可知，而实际类型到运行期才确定下来。

在重载时通过参数的静态类型而不是实际类型作为判定依据，因此，在编译阶段，Javac编译器会根据参数的静态类型决定使用哪个重载版本。所以选择了sayhello(Human)作为调用目标。

5-5-2-2. 动态分派

在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。最典型的应用就是方法重写

```
public class DynamicDisptch {
    static abstract class Human {
        abstract void sayhello();
    }
    static class Man extends Human {
        @Override
        void sayhello() {
            System.out.println("man");
        }
    }
    static class Woman extends Human {
        @Override
        void sayhello() {
            System.out.println("woman");
        }
    }
    public static void main(String[] args) {
        Human man = new Man();
```

```
            Human woman = new Woman();
            man.sayhello();
            woman.sayhello();
            man = new Woman();
            man.sayhello();
        }
}
```

运行结果:

man

woman

woman