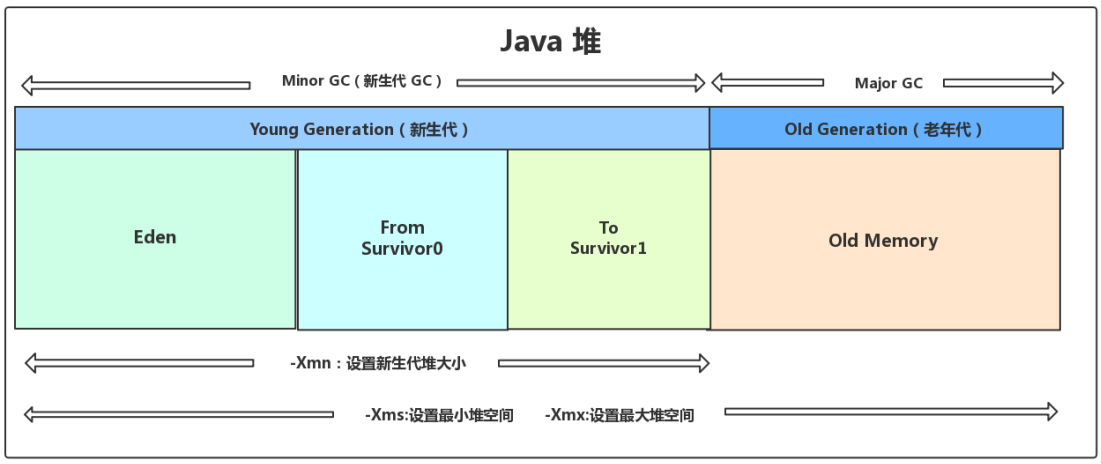
GC篇



堆内分配与回收策略

对象优先在 Eden 区分配

- 大多数情况下，对象在新生代Eden区中分配，当 Eden区没有足够空间进行分配时，虚拟机将发起一次 Minor GC

什么是Minor GC

- 指发生在新生代的垃圾收集，因为 Java对象大多朝生夕灭，所以 Minor GC非常频繁， 一般回收速度也比较快

大对象直接进入老年代

- **-XX:PretenureSizeThreshold** 当创建的对象超过指定大小时，直接把对象分配在老年代
- 大对象就是指需要大量连续内存空间的 Java 对象如字符串、数组，为了避免为大对象分配内存时由于分配担保
  机制带来的复制而降低效率

长期存活的对象进入老年代

- -XX:MaxTenuringThreshold 设定对象在Survivor区最大年龄阈值，超过阈值转移到老年代，默认15
- 对象头的 Age 属性记录年龄，对象在 Eden 出生并经过第一次 Minor GC 后仍然能够存活，并且能被 Survivor
  容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1，对象在 Survivor 中每熬过一次 MinorGC，年龄
  就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中

动态对象年龄判定

- 并不一定要Age到达阈值才晋升到老年代，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

空间分配担保

- -XX:HandlePromotionFailure
- 老年代的连续空间大于新生代对象总大小或者历次晋升到老年代对象的平均大小，就会进行 Minor GC，否则将进行 Full GC

GC 的分类

- 部分收集 (Partial GC)
    1. 新生代收集（Minor GC / Young GC）：只对新生代进行垃圾收集
    2. 老年代收集（Major GC / Old GC）：只对老年代进行垃圾收集。需要注意的是 Major GC 在有的语境中也用于指代整堆收集；目前只有 CMS 收集器会有单独收集老年代的行为
    3. 混合收集（Mixed GC）：对整个新生代和部分老年代进行垃圾收集 目前只有 G1
- 整堆收集 (Full GC) 收集整个 Java 堆 和 方法区

如何判定对象是否死亡

- 引用计数法：有地方引用它，计数器加一，引用失效，计数器减一 简单高效但是不能解决循环引用问题
- 可达性分析算法：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对象不可达，则此对象不会再被使用

CMS 用增量更新，G1用原始快照来进行可达性分析，因为要保证在一致性的快照上进行对象图的遍历

可达性分析算法的优化

- 可达性分析算法中，从GC Roots集合找引用链时，为了避免从所有 GCRoots 候选位置中进行根节点枚举，使用OopMap 数据结构来直接得到什么地方存放着对象引用，并不需要真正一个不漏地从方法区等GC Roots开始查找，缩短了根节点枚举时间

OopMap 存储哪两种对象引用

- 对象内的引用 在类加载完的时候，HotSpot就把对象内什么偏移量上是什么类型的数据计算出来
- 栈、寄存器中引用 在JIT编译过程中，也会在特定的位置记录下栈和寄存器中哪些位置是引用

OopMap 生成条件

- 只会在安全点生成

什么是安全点

- 可达性分析算法必须是在一个确保一致性的内存快照中进行。如果在分析的过程中对象引用关系还在不断变化，分析结果的准确性就不能保证。安全点意味着在这个点时，所有工作线程的状态是确定的，JVM 就可以安全地执行 GC

安全点选取的条件

- 能让程序长时间执行的地方，这些指令代表着我们的代码将长时间不会继续往下执行，避免GC等待太久，就应该在这些地方设立安全点
- 长时间执行 的最明显特征就是指令序列的复用， 例如方法调用、 循环跳转、 异常跳转等都属于指令序列复用

如何将线程停止在安全点

- 抢先式中断 系统中断所有用户线程，没到安全点的，恢复它，让其继续运行到安全点
- 主动式中断 在安全点设置一个标志，不断轮询这个中断标志，标志为真，就停在安全点指令
- 安全点存在的 因线程阻塞或休眠，无法响应虚拟机的中断请求，走到安全的地方去中断挂起自己的问题

什么是安全区

- 安全点的拓展，是指能够确保在某一段代码片段之中，引用关系不会发生变化，这个区域内，任何地方垃圾回收都是安全的，解决了上述问题

可达性分析判定为不可达对象一定被回收吗

- 不一定，如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize() 方法，当对象没有覆盖 finalize() 方法或者已被虚拟机调用过，那么就认为是没必要的，如果该对象有必要执行 finalize() 方法，那么这个对象将会放在一个称为 F - Queue 的队列中，虚拟机会触发一个 Finalizer 线程去执行它们的 finalize() 方法 ，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize() 执行缓慢或者发生了死锁，那么就会造成 F-Queue 队列一直等待，造成了内存回收系统的崩溃。之后 GC 对处于 F-Queue 中的对象进行第二次被标记，这时，如果对象已经在在 finalize() 中成功拯救自己 —— 只要重新与引用链上的任何一个对象建立关联即可，如把自己 (this关键字) 赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移出 "即将回收" 的集合，如果没有被移出集合，就会被回收

可达性分析

通过三色标记算法解释 为什么要在一致性的快照上进行对象图的遍历

- 白色： 对象还未访问，最后还是白色说明对象不可达
- 黑色: 可达对象 没有引用没扫描过
- 灰色： 已经被扫描过了但还有引用没扫描过

用户线程与收集线程并行，导致产生浮动垃圾，或则错误回收存活对象 (致命)

1. 插入黑色对象到白色对象引用
2. 删除所有灰色对象到白色对象的直接或间接引用

增量更新用来解决错误回收存活对象

1. 插入的记下来，黑色变灰色 增量更新
2. 删除的也会按没删除搜索 原始快照

什么是 GC Roots

- GC Roots 特指的是垃圾收集器(Garbage Collector)的对象，GC会收集那些不是 GC Roots 且没有被 GC Roots 引用的对象

GC Roots 的对象包括哪几种

- 虚拟机栈 (栈帧中的局部变量表)中引用的对象、本地方法栈(Native 方法)中引用的对象、方法区中类静态属性引用的对象、方法区中常量引用的对象 、所有被同步锁持有的对象

四种引用类型

- 强引用：程序代码中的引用赋值，会持续存在，不会被回收，就算OOM
- 软引用：软引用是用来描述一些还有用但并非必须的对象，如果内存空间足够，垃圾回收器就不会回收它，在系统将要发生内存溢出之前，将会把这些对象列入回收范围之中进行第二次回收。如果这次回收后还没有足够的内存，才会抛出内存溢出异常，用来实现内存敏感的高速缓存； JDK1.2 SoftReference
- 弱引用：弱引用也是用来描述非必须对象的，他的强度比软引用更弱一些，在垃圾回收时会直接回收； JDK1.2 WeakReference
- 虚引用：一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知，JVM只管理堆内内存，JDK中直接内存的回收就用到虚引用，JAVA 在申请一块直接内存之后，会在堆内存分配一个对象保存这个堆外内存的引用，这个对象被垃圾收集器管理，一旦这个对象被回收，相应的用户线程会收到通知并对直接内存进行清理工作；JDK1.2 PhantomReference

回收方法区

- 回收常量
  - 字符串常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池了
- 回收类型
  - 该类所有实例都回收；
  - 该类的内加载器也被回收了；
  - 该类对应的Class对象没在任何地方被引用，无法在任何地方通过反射访问该类的方法

垃圾收集算法

分代收集理论

- 弱分代假说：大部分对象都是朝生夕死
- 强分代假说：熬过越多次收集也难回收
- 跨代引假说：占少数(可以通过记忆集，将老年代分成若干块，存在跨代才加入GCRoots中)

记忆集与卡表

- 局部收集，可能存在跨代引用，记忆集用来记录从非收集区指向收集区存在跨代引用的数据结构，避免了把整个非收集区（老年代）加入 GCRoots 中，卡表是记忆集的实现

卡表有几种记录精度

- 字长精度：精确到一个机器字长（就是处理器寻址位数，比如32位或64位）
- 对象精度：精确到一个对象
- 卡精度：精确到一块内存区域

在 HotSpot 中一个卡页是多大

- 一般来讲卡页大小是 2 的 N 次幂

卡表的工作方式

- 只要卡页内有一个或多个对象的字段存在着跨代指针，那就将对应卡表的数组元素的值标识为1，称这个元素变脏，在垃圾收集时，只要筛选出卡表中变脏的元素就能得出哪些卡页内存块中包含跨代指针，把他们加入 GC Roots 中一并扫描

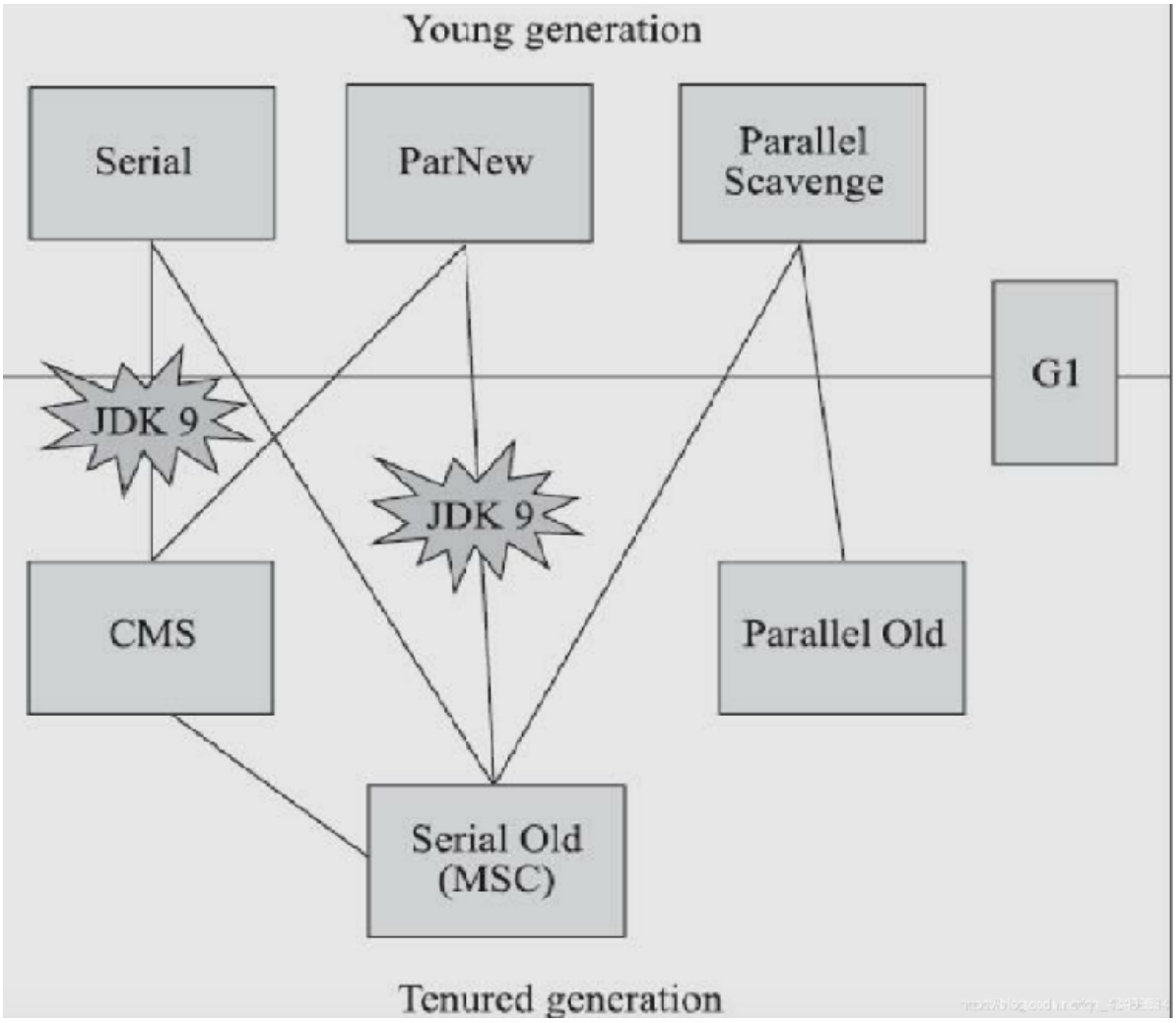
卡表如何更新

- 通过写屏障
- 用来更新卡表，G1之前全是写后屏障， 用于引用类型字段赋值时更新卡表。 G1还需要写前屏障来实现SATB
- ZGC 只需要读屏障（得益于 1.无分代，2.染色指针）

垃圾收集算法

1. 标记 — 清除算法：先标记需清除的对象，之后统一回收。这种方法效率不高，会产生大量不连续的碎片，且对象越多，效率越低；
2. 标记 — 复制算法：最开始将可用内存按容量划分为大小相等的两块 1：1，每次只使用其中一块。当使用的这块空间用完了，就将存活对象复制到另一块，再把已使用过的内存空间一次清理掉。优点是没有空间碎片，简单高效，缺点浪费大量空间；Appel 式 回收将新生代分成 Eden 8：Survivor 1，Eden 和 一个 Survivor 用于分配，另外 一 Survivor 用于复制，并使用老年代进行分配担保 - XX:HandlePromotionFailure -XX:SurvivorRatio 8
3. 标记 — 整理算法：先标记存活对象，然后让所有存活对象向一端移动，之后清理端边界以外的内存；移动回收复杂（更新引用），不移动碎片化分配复杂，总体来说移动的吞吐量最高，CMS(关注低延迟) 先用标记 - 清除，空间碎片过多，使用标记—整理 - XX:CMSFullGCsBeforeCompaction CMS收集器执行过若干次不整理空间的 Full GC 之后， 下一次进入Full GC 前会先进行碎片整理

垃圾收集器



Serial

- 单线程　　客户端模式C1
- 简单而高效，Serial 收集器由于没有线程交互的开销，可以获得很高的单线程收集效率

ParNew

- 多线程并行　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 服务端模式C2

Parallel Scavenge

- 与 ParNew 相似　吞吐量可控，可以自己设定一个期望，选择**自适应调节策略**

Serial　Old

- Serial 收集器的老年代版本，单线程，作为CMS并发收集失败的逃生门 或 JDK5及之前搭配 Parallel Scavenge

Parallel Old

- Parallel Scavenge 收集器的老年代版本，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　**JDK1.8 默认使用该搭配**

收集新生代和老年代都会 Stop the word 新生代标记复制，老年代标记整理

CMS

第一款并发收集器，它第一次实现了让垃圾收集线程与用户线程(基本上) 同时工作

基于**标记-清除**运作过程的四个步骤

- 初始标记 STW 标记 GCRoots 能直接关联到的对象
- 并发标记 从GC Roots的直接关联对象开始遍历整个对象图
- 重新标记 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 **停顿时间比并发标记短，比初始标记长**
- 并发清除 开启用户线程，同时 GC 线程开始对未标记的区域做清扫 **长**

CMS 缺点

- CMS对处理器非常敏感，处理核数量越少，对用户线程影响越大
- CMS无法处理浮动垃圾，CMS和用户线程并发运行期间预留的内存不够新对象分配，导致并发失败 STW 使用 Serial Old 回收
- 它使用的回收算法 "标记-清除" 算法会产生大量空间碎片产生

浮动垃圾是怎么样产生

- CMS 的并发标记和并发清理阶段， 用户线程是还在继续运行的， 程序在运行自然就还会伴随有新的垃圾对象不断产生， 但这一部分垃圾对象是出现在标记过程结束以后，只能下次垃圾回收处理

G1

G1 是 JDK9 后 服务端模式下默认的垃圾收集器，主要针对配备多颗处理器及大容量内存的机器，它是一款能够建立起 "停顿时间模型" 的收集器 —— 当我们指定在一个长度为M毫秒的时间片段的时候，这个垃圾收集器消耗在垃圾收集上的时间大概率不会超过 M 毫秒

怎样建立起可靠的停顿预测模型

Region 和 回收集

- G1仍是遵循分代收集理论， 但不再坚持固定大小以及固定数量的分代区域划分， 而是把堆划分为多个大小相等的区域（Region） 每个 1~32M (总是2的幂次方) ，每一个Region都可以根据需要，扮演不同的角色，收集器能够对扮演不同角色的 Region 采用不同的策略去处理，以达到更好的效果
- 每次回收不针对所有Region，将回收 Region 价值排序，根据停顿的期望值，选择Region加入回收集

四种不同Region标签

- Eden、Survivor、Old、 Humongous；H区可以认为是Old区中一种特列专门用来存储大数据的 当 0.5 Region <= 当对象大小 <= 1 Region 时候将数据存储到 H区 ； 当对象大小 > 1 Region 存储到连续的H区

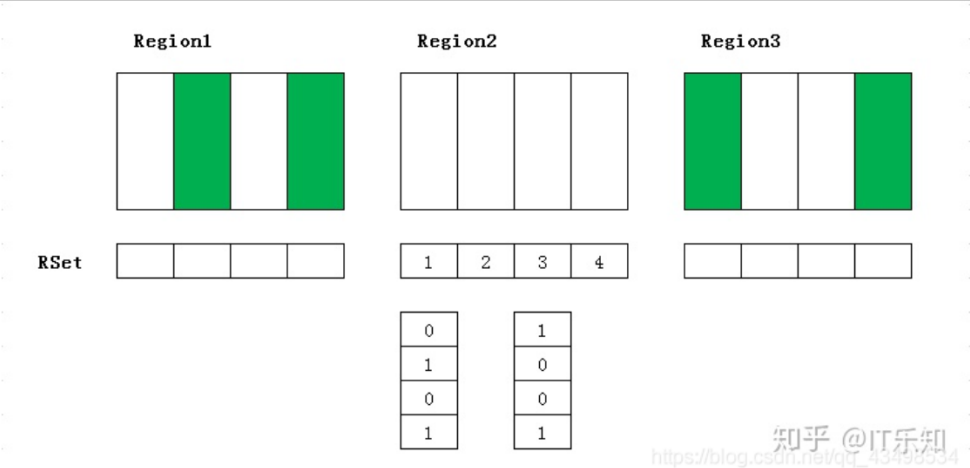
G1特点

1. 空间整合：G1从整体上看是基于标记-整理算法实现的，从局部(两个Region之间)上看是基于复制算法实现的，G1运行期间不会产生内存空间碎片
2. 可预测停顿： G1比CMS牛在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，消耗在垃圾收集上的时间不得超过N毫秒
3. 并行与并发： 能充分利用多CPU、多核环境下的硬件优势；可以并行来缩短 "Stop The World" 停顿时间，也可以 并发让垃圾收集与用户程序同时进行
4. 分代收集： 分代概念在G1中依然得以保留，但G1不再坚持固定大小以及固定数量的分代区域划分，改用角色不同的大小相等的Region来管理，对不同的Region采用不同的策略以达到更好的效果

Region 里面存在的跨 Region 引用对象如何解决

- 每个Region都有一个记忆集的实现 ——双向卡表

卡表记录了其他Region中的对象引用本Region中对象的关系，(谁引用了我的对象），它还记录了我指向谁，双向卡表结构实现比CMS实现复杂，且每个Region都有一份占用更多内存，CMS只有一份



Hashtable <key，int[]> key 是我指向谁 key = Region 1 —— Region 2 指向 Region 1；int[] 中元素为1，代表 Region 1中的第 2块与 第 4 块区域存在对 Region 1 的引用

在并发标记阶段如何保证收集线程与用户线程互不干扰地运行

- 并发标记阶段，为了分配对象和垃圾回收并发执行，每个 Region 设计了两个**TAMS** 的指针 —— PrevTAMS、NextTAMS，在NextTAMS指针之后给新对象分配内存

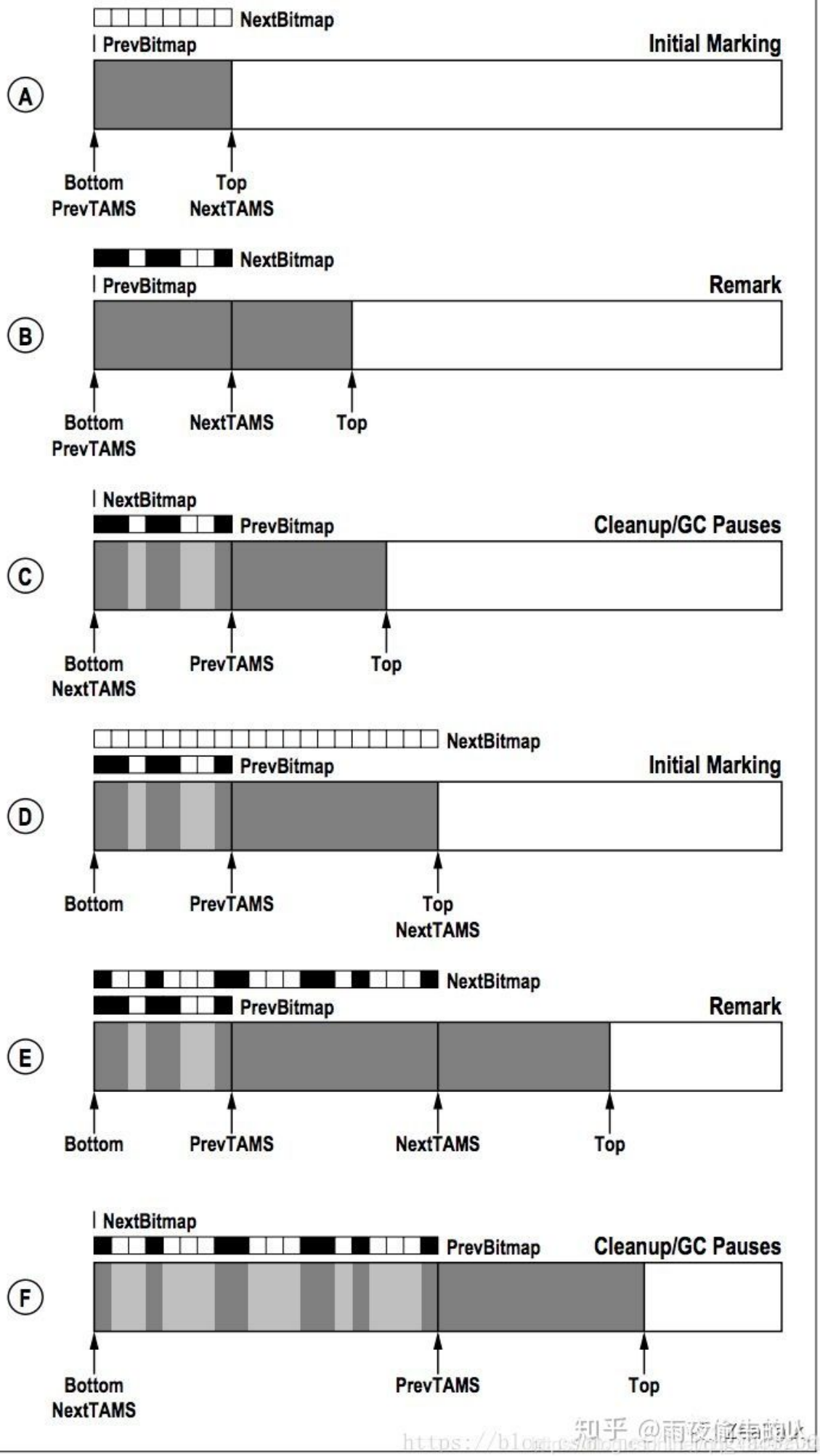
G1 会把存活对象的标记存放在哪 SATB 具体是什么呢

- G1 是借助 bitmap 来存放对象存活标记，每一个 bit 表示每个 Region 中的某个对象起始地址，如果 bit 标记为 1，则表示该对象存活

SATB

- 初始标记开始时，G1 收集器打了一个快照，形成一个所谓的对象图，这个对象图记录在 next marking bitmap 之中
- 并发标记阶段会在这个 bitmap 中记录对象存活标记
- 最终标记阶段，完成对快照对象图所有标记
- NextTAMS 指针之后的内容，在这一次的 GC 周期内并不关注
- 清理阶段，next marking bitmap 与 previous marking bitmap 会发生置换，next marking bitmap 在下一次周期开始前会被清空；那么此时这个 Region 的 previous marking bitmap 可以直接表示出 Region 在 [PrevTAMS、NextTAMS，在指针位置之上给新对象分配内存 ， NextTAMS)

这个区间内存活对象数量，并且可以根据 bitmap 算出存活对象的具体地址，辅助下一步的 Evacuation（选取CSet，拷贝并合并存活对象到新的Region里），回收的同时减少了内存碎片，当然 Evacuation 也是 STW的 至此完成了一次全局并发标记周期


A
NextBitmap
PrevBitmap
Initial Marking
Bottom
PrevTAMS
Top
NextTAMS
B
NextBitmap
PrevBitmap
Remark
Bottom
PrevTAMS
NextTAMS
Top
C
NextBitmap
PrevBitmap
Cleanup/GC Pauses
Bottom
NextTAMS
PrevTAMS
Top
D
NextBitmap
PrevBitmap
Initial Marking
Bottom
PrevTAMS
Top
NextTAMS
E
NextBitmap
PrevBitmap
Remark
Bottom
PrevTAMS
NextTAMS
Top
F
NextBitmap
PrevBitmap
Cleanup/GC Pauses
Bottom
NextTAMS
PrevTAMS
Top

G1 GC 模式是怎样的

YoungGC

- STW 以 RSet 作为根集扫描获取存活对象，将 Eden/Survivor Region 中的存活对象拷贝并合并到一个新的 Region 里 以减少内存碎片

Mixed GC

1. 初始标记： 标记 GC ROOT 能关联到的对象，需要STW
2. 并发标记： 从 GCRoots 的直接关联对象开始遍历整个对象图的过程，扫描完成后还会重新处理SATB记录的在并发标记过程中引用发生变动的对象
3. 最终标记： 短暂暂停用户线程，处理剩下的 SATB 记录，需要STW
4. 筛选回收： 更新 Region 的统计数据，对每个Region的回收价值和成本排序，根据用户设置的停顿时间制定回收计划。再把需要回收的Region中存活对象复制到空的 Region，同时清理旧的 Region，需要STW

G1 与 CMS 的对比

- 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
- 记忆集复杂，每个Region都有一份，内存占用高，某些极端情况下会达到堆空间的 20% 甚至更多
- CMS基于增量更新，G1基于SATB原始快照
- CMS G1都用写后屏障更新卡表，G1还需要写前屏障来实现 SATB

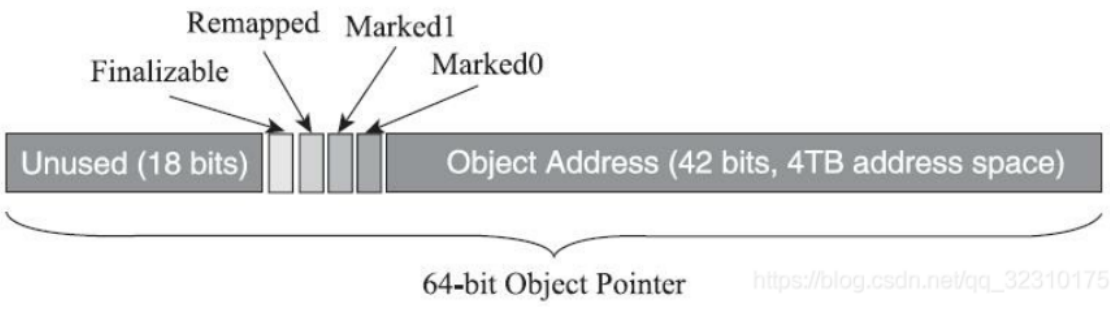
ZGC

ZGC 收集器是一款基于 Region 内存布局的，不设分代的， 使用了读屏障 、 染色指针和内存多重映射等技术来实现可并发的 标记 — 整理算法的， 以低延迟为首要目标的一款垃圾收集器

ZGC的 Region

- 小型 Region：容量固定为2MB，用于放置小于256KB的对象
- 中型Region：容量固定为32MB，用于放置大于等于256KB但小于4MB的对象
- 大型Region：容量不固定，可以动态变化，但必须放置为2MB的整数倍，用于放置4MB或以上的大对象。每个Region只放一个大对象，也就是说他的大小可以小于中型 Region，**大型Region在ZGC实现中不会被重分配，因为复制一个大对象的代价很高昂**

什么是染色指针



染色指针指的是把一些对象的信息标记在引用对象的指针上，这些信息包括了其引用的三色标记状态、是否进入了重分配集，是否只能通过finalize()方法访问到，但是由于这些信息是被标识在寻址指针的高4位上（linux下64位指针，只支持46位）所以，对于ZGC可管理的内存来说，最高只有 2 ^ 42 次方，也就是 4TB

染色指针的优势

- Region上存活对象被移走之后，能立马重用该Region ； 而不必等待整个堆中所有指向该 Region的引用都被修正后才能清理 (自愈)
- 使用染色指针可以只用读屏障， 提升了吞吐量
- 可拓展增强功能，目前还有很多位没用

染色指针如何寻址

- 多重映射 ： 将染色指针的标志位看作地址分段符，将不同分段映射到同一个物理空间(多对一)，经过多重映射转换后就可以寻址了
- 映射就是操作系统的虚拟存储器管理的功能，将虚拟地址空间映射到物理地址空间

ZGC运作过程

四个阶段都能并发

1. 并发标记：遍历对象图做可达性分析 `前后也要经过类似于 G1 的初始标记、 最终标记的短暂停顿，标记阶段会更新染色指针中的Marked 0 、 Marked 1标志`
2. 并发预备重分配：扫描所有Region，选出要收集的Region，组成重分配集 (RSet)
3. 并发重分配：将重分配集中存活的对象复制到新Region中，并为重分配集中每个Region维护一个转发表，当用户线程访问重分配集合中的对象，被读屏障截获，通过转发表指向新对象，并修改引用 —— 也叫指针自愈，当该Region存活的对象复制完毕，该Region就可以重新使用
4. 并发重映射：修改剩下的指向堆中重分配集合的旧引用，释放转发表

类加载篇

一个类型从被加载到虚拟机内存中开始， 到卸载出内存为止， 它的整个生命周期将会经历
加载 、 { 验证 、 准备 、解析} 也称连接 、 初始化 、 使用 和 卸载

类加载器有什么作用

- 实现类的加载
- 同类一起确立在 Java虚拟机中的唯一性，只有相同类加载器加载的类才是相同的

类加载过程

加载

用户可介入，可以自定义类加载器，重写 loadClass / findClass

1. 通过一个类的全限定名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
3. 在堆内存中实例化出代表这个类的 java.lang.Class 对象，通过堆中这个对象作为方法区这个类的各种元数据的访问入口

验证

- 文件格式验证　验证是否符合 Class 文件格式的规范，允许进入方法区
- 元数据验证　　对字节码描述的信息进行语义分析以确保符合 Java 语言规范
- 字节码验证　　保证方法执行的安全性，StackMapTable 将字节码验证的类型推导转变为类型检查，JDK6 将校验辅助措施挪到了 Javac 编译器里进行
- 符号引用验证 验证该类是否缺少或者被禁止访问它依赖的某些外部类、 方法、 字段等资源，以确保确保解析行为能正常执行

准备

为静态变量分配内存并赋初始值 (零值)，final static 在 javac 的时候，字段表集合中会有一个 ConstantValue 的属性，那么在准备阶段就会被赋期望的值，其余的会存放在< clinit >中，在类初始化阶段才会被赋期望值

解析

将符号引用被替换为直接引用并对其可访问性进行检查，需要在 getfield，putfield，getstatic，putstatic 等指令之前执行　解析主要针对 类或接口、 字段、 类方法

符号引用　以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可

直接引用　直接引用是可以直接指向目标的指针、 相对偏移量或者是一个能间接定位到目标的句柄，引用的目标必定已经在虚拟机的内存中存在

类或接口的解析

- 如果被解析的类或接口不是数组类型，那虚拟机会把全限定名传递给当前代码的类加载器去加载，在加载过程中，可能触发其他类或接口的加载动作，一旦有任何异常，就宣告失败
- 如果类或接口是数组类型，并且数组的元素类型为对象，那将会按照第一点的规则加载数组元素类型，接着由虚拟机生成一个代表该数组维度和元素的数组对象
- 若上面没有异常，则进行符号引用验证，确认当前代码对其的访问权限，没权限 java.lang.IllegalAccessError

字段解析

- 会先对字段所属的类或接口解析，成功后才进行字段解析
- 如果类或接口本身包含了简单名称和字段描述符都与目标匹配的字段，返回直接引用
- 如果实现了接口，从下往上递归搜索，看各个接口与父接口是否有简单名称和字段描述符都匹配的字段
- 如果不是 Object，从下往上递归搜索，看父类中否存在
- 否则，抛出 java.lang.NoSuchFieldError 异常
- 查找成功，验证访问权限

类方法解析

- 会先对方法所属的类解析，成功后进行方法解析
- 本来是解析类方法，如果方法是接口方法则会抛出 java.lang.IncompatibleClassChangeError

- 如果类本身就就包含了简单名称和字段描述符都与目标匹配的方法直接返回
- 查找父类、查找接口 、抛出 java.lang.NoSuchMethodError
- 验证访问权限

接口方法解析

- 会先对方法所属的接口解析，成功后进行方法解析
- 本来是解析接口方法，如果方法是类方法则会抛出 java.lang.IncompatibleClassChangeError
- 在接口中查找、查询父接口、抛出java.lang.NoSuchMethodError
- 验证访问权限

初始化

执行类构造器 < clinit > 方法 它是由类变量赋值和 static 语句块构成，执行时父类的< clinit > 一定已经执行完毕，接口实现类初始化时候不会执行父类的 < clinit >

立即进行初始化

也叫**主动引用** 有且只有下面这几种情况

- 遇到 new、 getstatic/putstatic 或 invokestatic 这四条字节码指令时；（被final修饰、 已在编译期把结果放入常量池的静态字段除外）
- 使用 java.lang.reflect 包的方法对类型进行反射调用的时候，如果类型没有进行过初始化，则需要先触发其初化
- 当初始化类的时候， 如果发现其父类还没有进行过初始化， 则需要先触发其父类的初始化，但是对于接口是用到其父接口的时候才会初始化其父接口
- 当虚拟机启动时， 用户需要指定一个要执行的主类（ 包含main()方法的那个类） ， 虚拟机会先初始化这个主类
- 当一个接口中定义了JDK 8 新加入的默认方法 (被default关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化， 那该接口要在其之前被初始化

被动引用

被动引用不会触发初始化

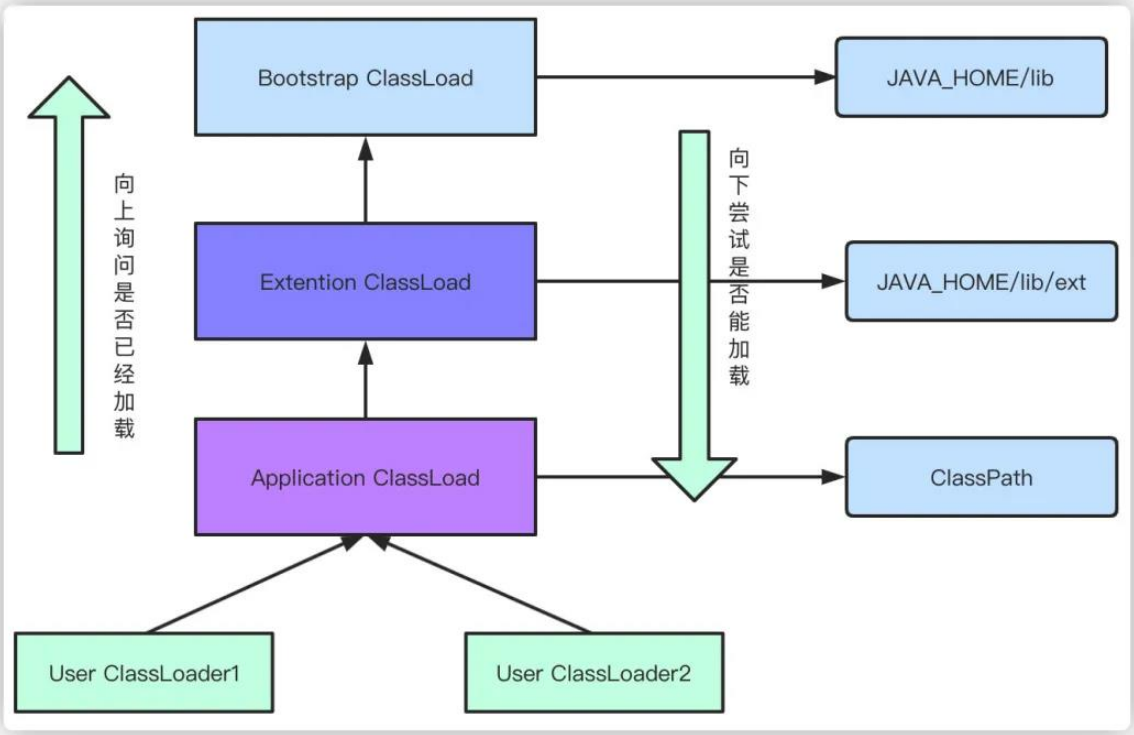
- 父类的静态字段 value，SubClass.value 只会触发父类初始化
- 创建一个对象数组，并不会触发该对象初始化，对象数组的对象类型是由JVM自动生成的一个继承 Object 的子类的类型，创建动作由 newarray 字节码指令触发
- 访问未初始化类中的常量，编译阶段通过常量传播优化，已经存储在访问类的常量池中了，访问也是访问的常量池中的常量

数组如何创建

- 数组的元素类型为对象，会加载数组类型元素，接着由虚拟机生成一个代表该数组维度和元素的数组对象

双亲委派模型

三种类加载器以组合方式形成父子关系，如果一个类加载器收到了类加载的请求， 它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成， 每一个层次的类加载器都是如此， 因此所有的加载请求最终都应该传送到最顶层的启动类加载器中， 只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类） 时， 子加载器才会尝试自己去完成加载



三种类加载器

- 启动类加载器（Bootstrap Class Loader）C++实现 负责加载存放在<JAVA_HOME>\lib目录，或者被-Xbootclasspath参数所指定的路径中存放的jar包和类
- 扩展类加载器（Extension Class Loader） 负责加载<JAVA_HOME>\lib\ext目录中， 或者被java.ext.dirs系统变量所
  指定的路径中jar包和类
- 应用程序类加载器（Application Class Loader） 负责加载用户类路径 ClassPath 上所有的 jar 包和类，我们可以拓展

采用双亲委派模型好处

- Java 中的类随着它的类加载器一起具备了一种带有优先级的层次关系，在程序的各种类加载器环境中都能够保证是同一个类，保证了 Java 程序的稳定运行，避免了类的重复加载，也保证了 Java 的核心 API 不被篡改

如何破坏双亲委派模型

- 继承 ClassLoader，如果我们不想打破双亲委派模型，就重写 ClassLoader 类中的 findClass() 方法即可，无法被父类加载器加载的类最终会通过这个方法被加载。但是，如果想打破双亲委派模型则需要重写 loadClass() 方法

双亲委派模型的三次破坏

- 对于 JDK2 以前的代码做出的妥协，双亲委派模型在 JDK 1.2 之后才被引入，之前已经有代码重写了loadClass方法

- JNDI 会对资源进行查找和集中管理，需要调用 SPI代码，但是 JNDI 通过启动类加载器记载不认识 ClassPath 路径的上的 SPI 代码，故引入了线程上下文类加载器
- 追求程序的动态性，如热部署

什么是 SPI

- 一种服务发现机制，它通过在 ClassPath 路径下的 META-INF/services 文件夹查找文件，自动加载文件里所定义的类

SPI 机制，除线程上下文加载器有无其他方法

- 通过在 META-INF/services 目录下配置，通过 ServiceLoader 加载

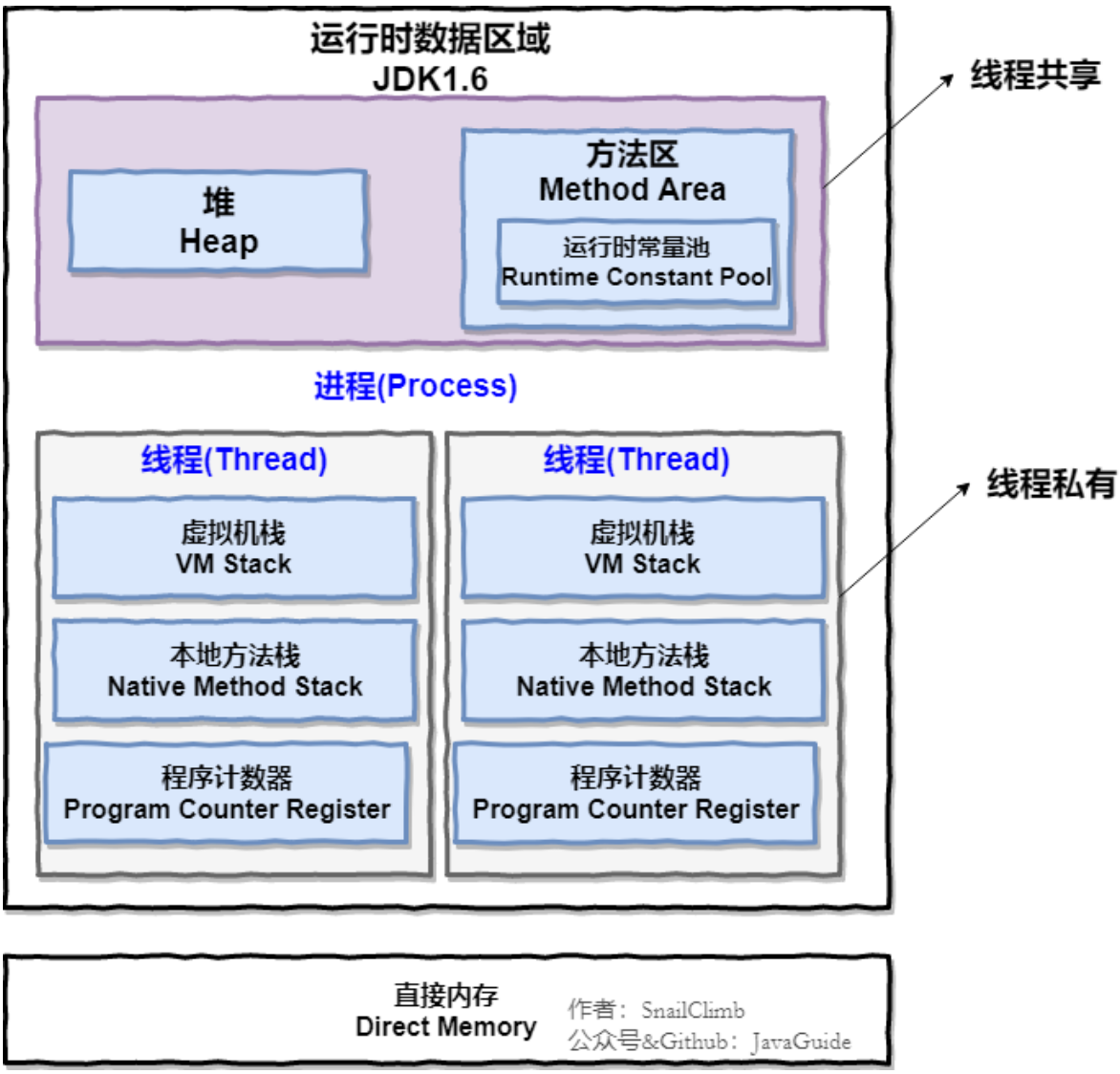
OSGI 如何实现模块化热部署

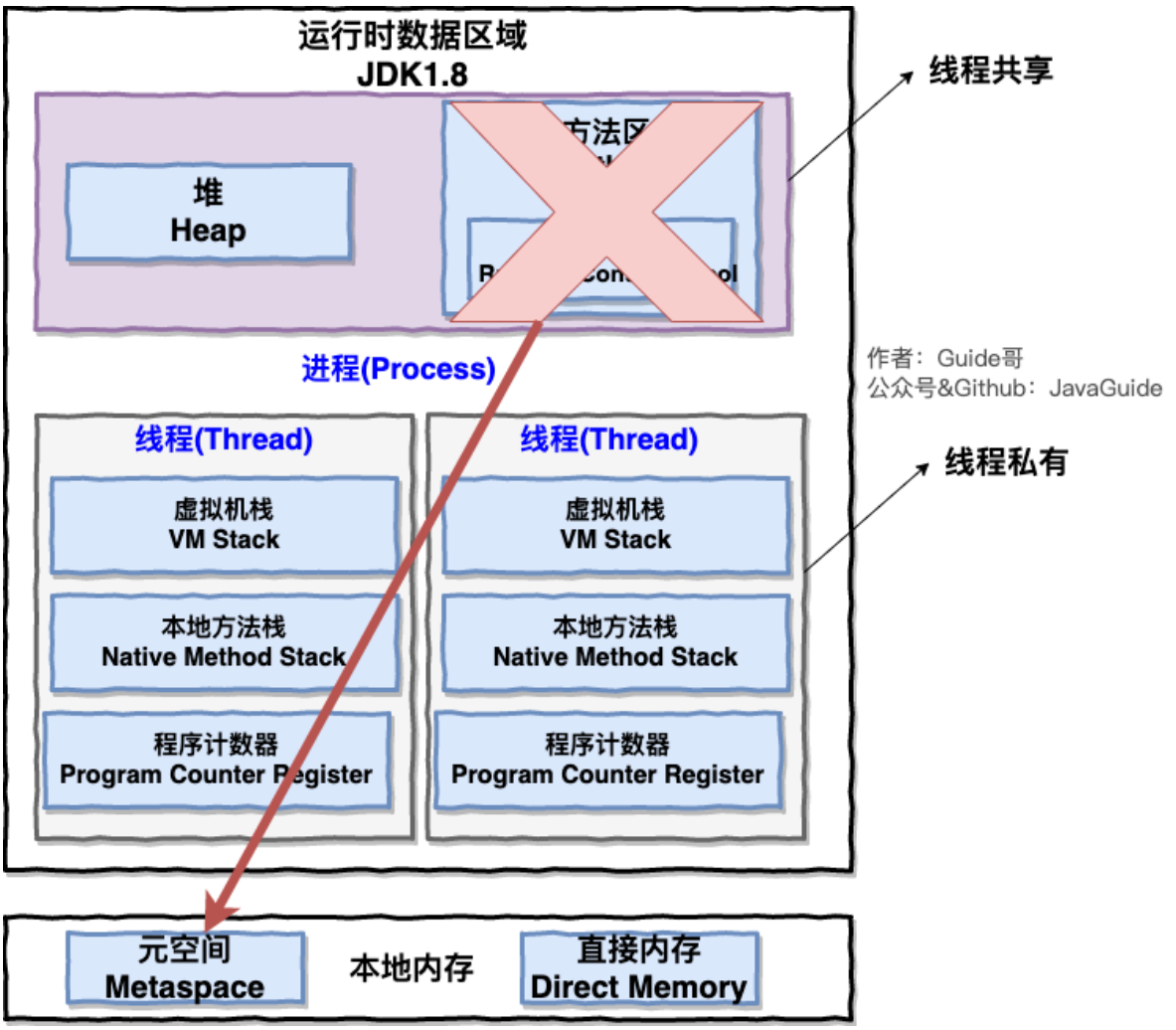
- 每一个程序模块（OSGI 中称为Bundle） 都有一个自己的类加载器，当需要更换一个模块时，就把模块连同类加载器一起换掉以实现代码的热替换

收到类加载请求时，OSGI 搜索类的流程

- 将以 java.* 开头的类，委派给父类加载器加载
- 否则，将委派列表名单中的类，委派给父类加载器
- 否则，将 Import 列表中的类，委派给 Export 这个类的模块的类加载器加载
- 否则，查找当前模块的 ClassPath ，使用自己的类加载器加载
- 否则，判断类是否在 Fragment 模块 中，如果在，委派给Fragment 模块的类加载器加载
- 否则，查找 Dynamic Import 列表的模块，委派给对应模块的类加载器加载
- 否则，类查找失败

内存区域篇


运行时数据区域
JDK1.6
堆
Heap
方法区
Method Area
运行时常量池
Runtime Constant Pool
线程共享
进程(Process)
线程(Thread)
虚拟机栈
VM Stack
本地方法栈
Native Method Stack
程序计数器
Program Counter Register
线程(Thread)
虚拟机栈
VM Stack
本地方法栈
Native Method Stack
程序计数器
Program Counter Register
线程私有
直接内存
Direct Memory
作者：SnailClimb
公众号&Github：JavaGuide



线程私有：随着用户线程启动创建，结束而销毁

共享区域：随着虚拟机进程启动一直存在

线程私有区域

程序计数器

- 当前线程所执行的字节码的行号指示器，程序的分支、 循环、 跳转、 异常处理、 线程恢复都依赖于它完成
- 为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储
- 如果正在执行本地方法，计数器值为空
- 唯 一 一 个不会发生**OOM**

JAVA虚拟机栈

- 描述JAVA方法执行的线程内存模型，一个方法调用与执行完毕对应着一个栈帧在虚拟机栈中的入栈与出栈

  一个栈帧包括 局部变量表，操作数栈，动态连接，方法返回地址等信息

本地方法栈

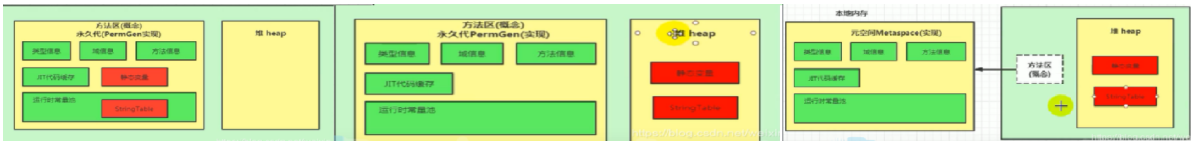
- 本地方法栈为虚拟机使用到的 Native 方法服务，HotSpot 虚拟机中和 Java 虚拟机栈合二为一

线程共享

堆

- 是Java 虚拟机所管理的内存中最大的一块，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就是存放对象实例，**几乎所有**的对象实例以及数组都在这里分配内存

方法区



类型信息、域信息、方法信息、JIT即时编译缓存、静态变量、运行时常量池

- 运行时常量池: Class 文件常量池表中字面量与符号引用在类加载完毕后都会存放在方法区的运行时常量池，是一个动态区域，运行时也可加入常量，例如 String 的 intern () 方法，使用不当会导致 OOM
- 域：成员变量

永久代为什么要被元空间替换

- 不是总能知道永久代应该设置为多大合适
- HotSpot为了获取JRockit中的优秀功能，JRockit 没有永久代

直接内存 (非运行时数据区的一部分)

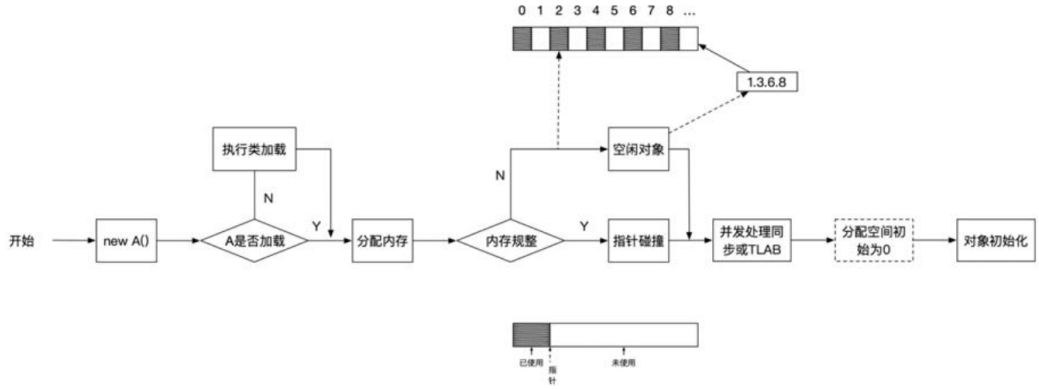
- 一般使用 Native 函数来实现直接分配堆外内存，不是虚拟机运行时数据区的一部分，但这部分内存也被频繁地使用，也可能导致OOM

对象的创建过程

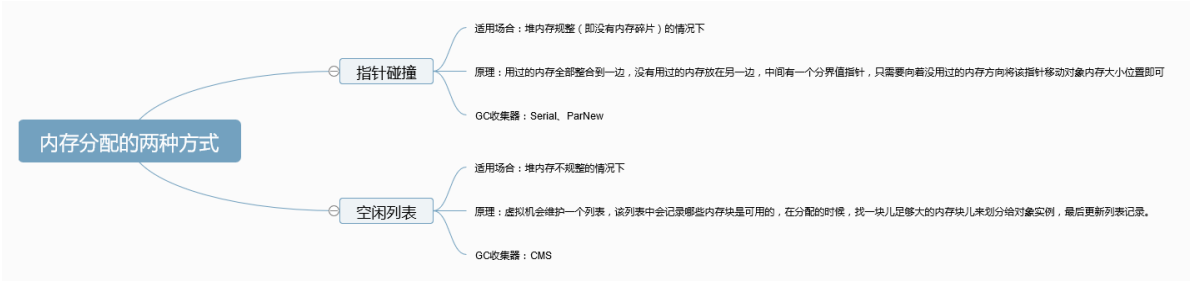
当Java虚拟机遇到一条字节码 new 指令时，首先将去检查这个指令的参数是否能在运行时常量池中定位到一个类的符号引用并且检查这个符号引用代表的类是否已被加载、 解析和初始化过。如果没有，那必须先执行相应的类加载过程

-- 类加载 --

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，分配内存可以通过指针碰撞，或者空闲列表(具体要根据垃圾回收算法)，为线程安全使用CAS+失败重试保证原子性 或 是在TLAB中分配(同时初始化零值)，初始化零值(除对象头外)，对对象头进行必要的设置。执行 <init> 方法 (取决于 new 指令后是否有 invokespecial)，按照程序员的意愿初始化对象，只要是new 对象就有，如果是通过序列化 new 之后就没有 invokespecial



JVM 给对象分配内存的策略



Java 内存分配是如何保证线程安全的

- CAS + 失败重试 冲突失败就重试，直到成功为止
- 为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

对象的内存布局

对象头

- 第一部分用于存储对象自身的运行时数据 哈希码、GC 分代年龄、锁状态标志等
- 一部分是类型指针
- 如果是数组类型，还有数组的长度

普通对象

```
|-------------------------------------------------------------|
|                  Object Header (64 bits)                    |
|-------------------------------------|-----------------------|
|        Mark Word (32 bits)          |    Klass Word (32 bits) |
|-------------------------------------|-----------------------|
```

数组对象

```
|-------------------------------------------------------------------------|
|                           Object Header (96 bits)                       |
|-------------------------------|-----------------------|-----------------|
|       Mark Word(32bits)       |    Klass Word(32bits) |  array length(32bits)  |
|-------------------------------|-----------------------|-----------------|
```

Mark Word (32 bits)	State
hashcode:25 | age:4 | biased_lock:0	01	Normal
thread:23 | epoch:2 | age:4 | biased_lock:1	01	Biased
ptr_to_lock_record:30	00	Lightweight Locked
ptr_to_heavyweight_monitor:30	10	Heavyweight Locked
	11	Marked for GC

Mark Word (64 bits)	State
unused:25 | hashcode:31 | unused:1 | age:4 | biased_lock:0	01	Normal
thread:54 | epoch:2 | unused:1 | age:4 | biased_lock:1	01	Biased
ptr_to_lock_record:62	00	Lightweight Locked
ptr_to_heavyweight_monitor:62	10	Heavyweight Locked
	11	Marked for GC

实例数据

- 类中定义的成员变量

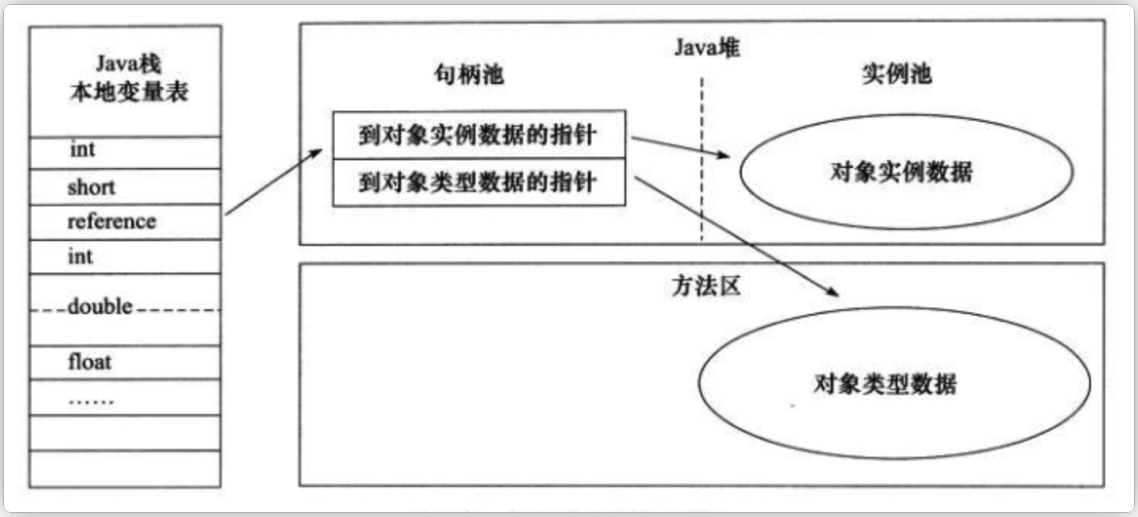
对齐填充

- 对象的大小必须是 8 字节的整数倍

对象的访问方式

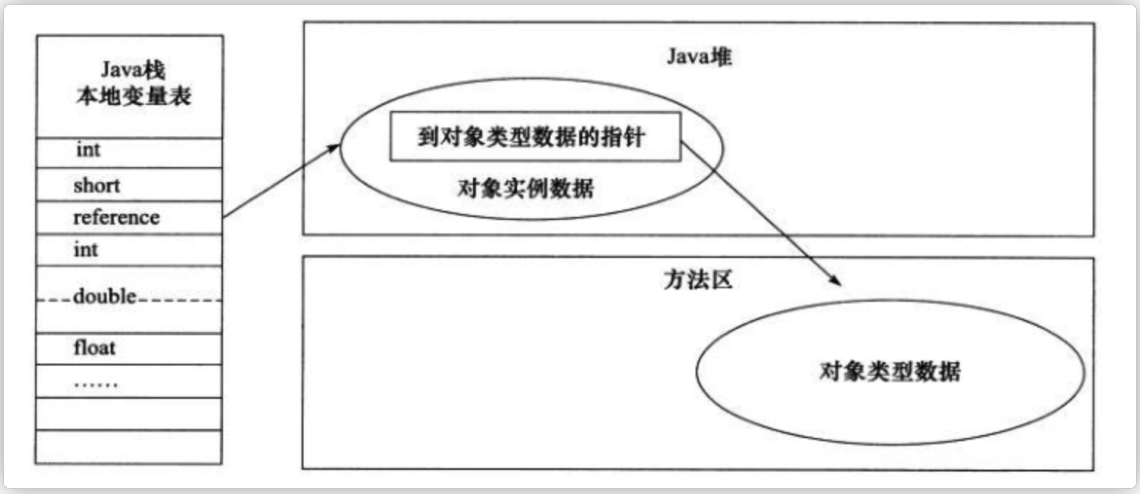
句柄

Java 堆中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实例数据与类型数据各自具体的地址信息



直接指针

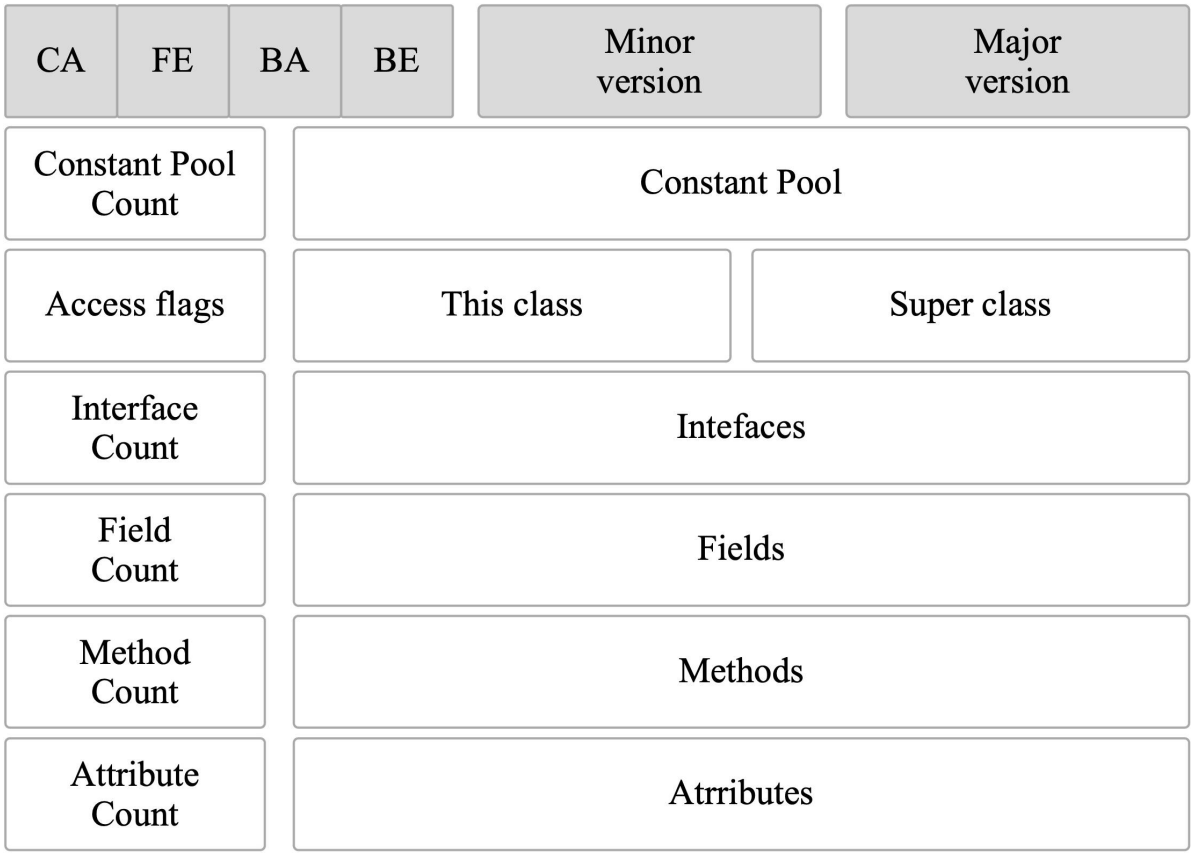
reference 中存储的直接就是对象地址，对象的内存布局就必须考虑如何放置访问类型数据的相关信息



比较句柄与直接指针

- 使用句柄来访问的最大好处是 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。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销

Class文件篇



Class文件常量池主要存放两大常量：字面量和符号引用，字面量比较接近于 Java 语言层面的的常量概念，如文本字符串、声明为 final 的常量值等。而符号引用则属于编译原理方面的概念。包括下面三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

虚拟机字节码执行引擎篇

执行引擎在执行字节码的时候通常有哪两种方式

- 通常会有解释执行 (通过解释器执行) 和 编译执行 (通过即时编译器产生本地代码执行)

什么是解释执行

- 解释执行引擎是基于栈的执行引擎，一个个方法对应一个个栈帧，每个方法的执行和结束对应着栈帧的入栈和出栈

基于栈的执行引擎与基于寄存器的执行引擎比较

- 优点
    1. 基于栈，可移植，不受硬件约束
    2. 编译器实现简单，因为只在栈上操作
- 缺点
    1. 完成相同功能指令条数多
    2. 频繁访内，虽然可以通过栈顶缓存优化，把最常用操作映射到寄存器中避免访内，但不能解决本质问题

栈帧结构

栈帧中包含什么

- 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

局部变量表中存放什么

- 存放方法参数与方法内定义的局部变量

局部变量表的大小

- 在Java程序被编译为 Class文件时，就在方法的Code属性的 max_locals 数据项中确定了该方法所需分配的局部变量表的最大容量

操作数栈的大小

- 编译的时候被写入到 Code属性的 max_stacks 数据项

局部变量表和操作数栈的可以重叠

- 两个不同栈帧相互独立，但是上面栈帧的局部变量表可以与下面栈帧的操作数栈重复，可以节约空间，还可以在进行方法调用时就可以直接共用一部分数据，无须进行额外的参数复制传递

动态连接

- 指向运行时常量池中该栈帧所属方法的(符号)引用，以支持方法调用过程中的动态链接

方法返回地址

- 方法返回分为两种

  （1）正常调用完成 （2) 执行过程中遇到了异常并且在方法内无法处理，无返回值

- 方法返回地址

  （1）方法正常退出 栈帧会保存主调方法 PC 计数器值作为返回地址 （2）方法异常退出时，返回地址是通过异常处理器表来确定

方法调用

- 只是确定调用的版本而不是执行

解析

也叫解析调用

- 解析调用是个静态的过程， 在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用全部转变为明确的直接引用
- 在编译完成就能确定版本的方法（非虚方法），invokestatic 静态方法；invokespecial 调用实例构造器 < init > 方法、私有方法、父类中的方法； invokevirtual (被final修饰的实例方法)，这些方法在类加载解析阶段就会把符号引用转直接引用

JVM支持的方法调用字节码指令

- invokestatic 用于调用静态方法
- invokespecial 用于调用实例构造器 < init > 方法、 私有方法和父类中的方法
- invokevirtual 用于调用所有的虚方法
- invokeinterface 用于调用接口方法， 会在运行时再确定一个实现该接口的对象
- invokedynamic 先在运行时动态解析出调用点限定符所引用的方法， 然后再执行该方法

前面4条调用指令， 分派逻辑都固化在 Java 虚拟机内部， 而 invokedynamic 指令的分派逻辑是由用户设定的引导方法来决定的

静态分派 — 方法重载

方法调用时有静态类型 (外观类型) 和 实际类型 (运行时类型)，编译器对于重载方法的选择，是根据传入参数的静态类型来选择的，在编译器结束就确定了方法的版本，实例方法和静态方法的重载的都是通过静态分派来确定调用版本的

为什么返回值不能作为方法重载依据

- Java 语言要重载一个方法，则必须要与原方法具有相同方法名和不同的**特征签名**，Java 代码层面，返回值并未包含在特征签名之中，但是JVM字节码层面，受查异常表和返回值也包含在特征签名中

动态分派 — 方法重写

- 通过 invokevirtual 指令，invokevirtual 会找到操作数栈栈顶的第一个元素的所指向对象的实际类型，先在实际类型中寻找匹配的方法，后到父类中寻找，找不到抛出**AbstractMethodError**，该指令不仅仅会把符号引用转化为直接引用，还会根据方法接收者的实际类型来选择方法版本

虚表

动态分派是通过虚表实现的，方法有虚方法表，接口方法有虚接口方法表，通过索引来找到各个方法的实际入口地址，父类和子类都有自己虚表，如果子类重写了父类的方法，那么子类的虚方法表中，入口会指向子类实现方法的版本的入口，没有实现则指向父类实现的方法版本入口

单分派与多分派

- 静态分派属于静态多分派，由静态类型与方法参数两个宗量确定；动态分派由实际类型确定，属于单分派
- 宗量 —— 方法的接收者、方法参数 称为宗量

前端编译与优化篇

Javac 流程

插入注解处理器、解析与填充符号表(词法分析、语法分析、填充符号表) 、提前处理注解、解析与填充符号表、语义分析与生成字节码

语义分析与生成字节码

- 标注检查 使用前是否声明 常量折叠优化 int a = 1+2 -> a = 3
- 保证 final 的不变性只在编译器，编译完后，final变量与普通变量无异
- 解语法糖
- 生成字节码　添加 实例构造器({}，实例变量初始化，调用父类的实例构造器) ，类构造器 (static {},类变量初始化，
  调用父类的实例构造器) ，保证先执行父类的实例构造器，然后初始化变量，最后执行语句块

如果用户代码中没有提供任何构造函怎么办

- 在填充符号表阶段将会添加一个没有参数的、可访问性 (public、 protected、 private或 package) 与 当前类型一致的默认构造函数

JAVA 语法糖

泛型

自动装箱、拆箱、循环遍历

- 自动装箱 Interger.valueOf()、拆箱 Interger对象.intvalue()
- 循环遍历 转换成迭代器
- 变长参数转化成数组形式传入

条件编译

- if (true) 语句在编译器就会执行，满足条件才会生成到字节码文件 只支持语句基本块，不支持类级别，不像C++的宏定义

其他语法糖

- Lambda jdk8
- twr jdk7 需要实现 Closeable 或 AutoCloseable

后端编译与优化篇

即时编译器

为什么 HotSpot 虚拟机要使用解释器与即时编译器并存的架构

- 解释执行可以使程序可以迅速启动和执行
- 随着时间的推移， 即使编译器把越来越多的代码编译成本地代码， 这样可以减少解释器的中间损耗，获得更高的执行效率

为何HotSpot虚拟机要实现两个不同的即时编译器

- HotSpot 虚拟机可以根据自身版本与宿主机器的硬件性选择适合的即时编译器，当然用户也可以使用 "-client"或"-server"参数去强制指定虚拟机运行在Client模式或Server模式

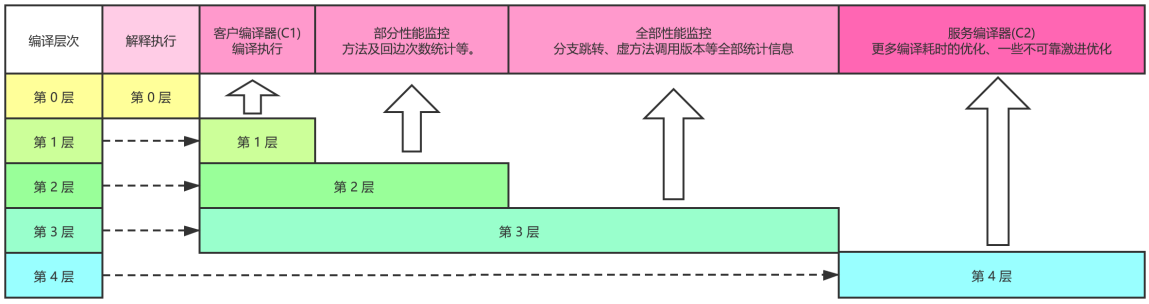
混合模式

- 分层出现之前 解释器 + 用户指定(- Client < C1 > - Server < C2 >) 指定虚拟机运行在服务端还是客户端模式，都叫混合模式

分层编译

分层编译的好处：解释执行不需收集监控信息，客户端编译执行以更高的编译速度给服务端编译提供更多的时间

```
jdk7 服务端模式默认使用默认使用分层编译
level 0 interpreter 解释执行
level 1 C1 编译执行,无profiling (性能监控)
level 2 C1 编译执行,仅方法及循环 back-edge 执行次数的 profiling
level 3 C1 编译执行,开启全部性能监控,收集分支跳转,虚方法调用版本
level 4 C2 编译执行,将字节码编译为本地代码,耗时更长,性能更优秀,根据监控信息激进优化。
```



哪些程序代码会被编译为本地代码

- 热点代码
  - 多次被调用的方法
  - 多次执行的循环体

栈上替换

- 编译发生在方法执行的过程中所以被称为栈上替换 —— 方法的栈帧还在栈上，方法就被替换了

程序何时使用解释器执行，何时使用编译器执行

- 当一个方法被调用或解释器遇到一条回边指令时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行，对方法调用计数器或回边计数器进行相应操作后，如果到达触发即时编译条件，提交即时编译请求，后继续以解释执行，当即时编译完毕后以编译器执行

如何判断是否是热点代码，需要触发即时编译

- HotSpot 采用的是 是基于计数器的热点探测
- 方法调用计数器
- 回边计数器

即时编译过程

- 字节码 -> HIR 高级中间代码表示 -> LIR 低级中间代码表示，中间会进行优化

优化技术

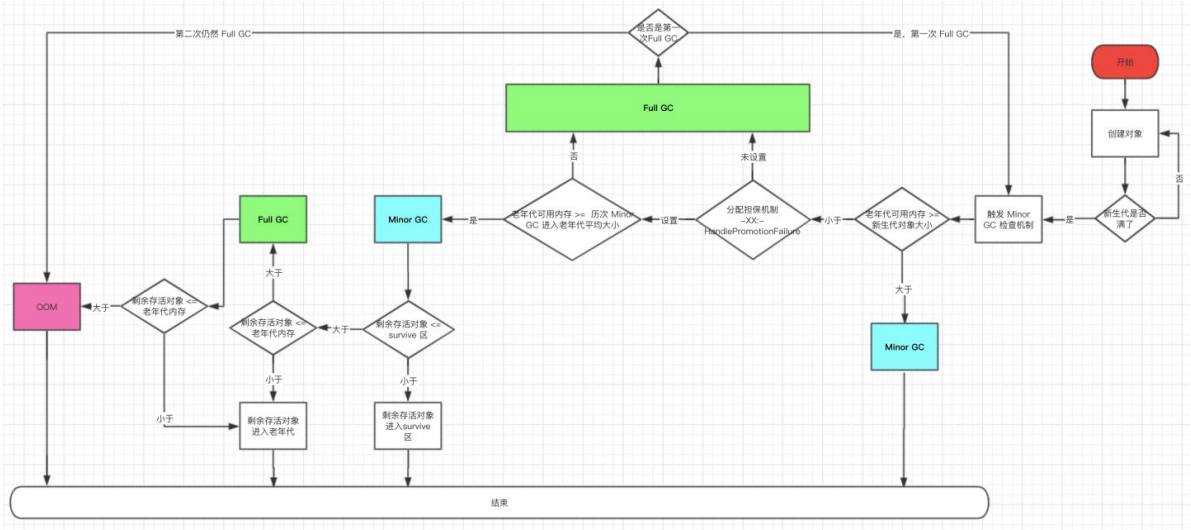
逃逸分析

```
逃逸分析原理：动态分析对象作用域
分为:不会逃逸 方法逃逸(作为参数传递) 线程逃逸

方法逃逸： 作为参数传递到其他方法中
线程逃逸：别的线程能访问到

栈上分配 在栈上分配对象内存,随栈帧出栈而销毁 支持方法逃逸
标量替换 如果一个对象可以被分解 且完全不会逃逸,可以把对象拆分成原始类型栈上分配,为其他优化垫基
同步消除 没有线程逃逸,就没有竞争,就没有同步的必要
```

OOM篇



在正式 Minor GC 前，JVM 会先检查新生代中对象，是比老年代中剩余空间大还是小。假如 Minor GC之后 Survivor 区放不下剩余对象，这些对象就要进入老年代 老年代剩余空间大于新生代中的对象大小，那就直接 Minor GC， GC 完 Survivor 不够放，老年代也绝对够放；老年代剩余空间小于新生代中的对象大小，这个时候就要查看是否启用了老年代空间分配担保规则

触发 Full GC的条件

1. 调用 System.gc() 建议 JVM 进行 Full GC
2. CMS 并发失败，使用 Serial Old 收集
3. 空间分配担保失败
4. 老年代空间不足

Java 堆溢出

- 在 Java 堆中没有内存完成实例分配，并且堆也无法再扩展时， Java 虚拟机将会抛出 OutOfMemoryError 异常

溢出原因

- 应用程序保存了无法被GC回收的对象

排查解决思路

- 查找关键字报错信息 java.lang.OutOfMemoryError: Java heap space
- 使用内存映像分析工具 Jprofiler，对 Dump 出来的堆储存快照进行分析，分析清楚是内存泄漏还是内存溢出
- 如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链，修复应用程序中的内存泄漏
- 如果不存在泄漏，先检查代码是否有死循环，递归等，再考虑用 -Xmx 增加堆大小

栈溢出

如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常； 如果Java虚拟机栈容量可以动态扩展，当栈扩展时无法申请到足够的内存会抛出 OutOfMemoryError 异常

溢出原因

- 在单个线程下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，抛出StackOverflowError 异常
- 不断地建立线程的方式会导致内存溢出

排查解决思路

- 查找关键报错信息，确定是 StackOverflowError 还是 OutOfMemoryError
- 如果是 StackOverflowError，检查代码是否递归调用方法等
- 如果是 OutOfMemoryError，检查是否有死循环创建线程等，通过-Xss 降低的每个线程栈大小的容量

方法区溢出

运行时产生大量的类，填满方法区，或者当常量池无法再申请到内存时会抛出OutOfMemoryError 异常

溢出原因

- 使用CGLib 生成了大量的代理类，导致方法区被撑爆
- 在Java7 之前，频繁的错误使用 String.intern() 方法

排查解决思路

- 检查是否使用 CGLib 生成了大量的代理类
- 检查代码是否频繁错误得使用 String.intern() 方法

直接内存溢出

直接内存也被频繁地使用，也可能导致OOM

溢出原因

- 本机直接内存的分配虽然不会受到 Java 堆大小的限制，但是受到本机总内存大小限制
- NIO程序中，使用 ByteBuffer.allocteDirect(capability) 分配的是直接内存，可能导致直接内存溢出
- 直接内存由 -XX:MaxDirectMemorySize 指定，如果不指定，则默认与Java堆最大值（-Xmx指定）一样

排查解决思路

- 检查代码是否恰当
- 检查 JVM参数 -Xmx，-XX:MaxDirectMemorySize 是否合理

性能监控故障处理

CPU 100%

1. ps -ef | grep 运行的服务名字，直接top命令也可以看到各个进程CPU使用情况
2. top -Hp PID 显示进程PID下所有的线程，定位到消耗CPU最高的线程 top -H -p 特定进程中的线程
3. 将线程ID转换成16进制 printf '%x\n'

4. jstack 导出进程当前时刻的线程快照到文件
5. 最后用 cat 命令结合 grep 命令对十六进制线程 PID 进行过滤，可以定位到出现问题的代码

监控和故障处理工具

- jps 查看所有JAVA进程
- jstat 监视虚拟机各种运行状态信息
- jinfo 实时查看和修改虚拟机参数，不需要重启
- jmap 生成堆转储快照dump
- jhat 分析dump文件
- jstack 生成虚拟机当前时刻的线程快照

可视化工具

- Jconsole
- Visual VM 功能的集合，比Jprofiler强太多

调优篇

GC调优

- GC调优目的：GC时间够少, GC次数够少

minor GC 单次耗时 < 50ms，频率10秒以上。说明年轻代OK

Full GC 单次耗时 < 1秒，频率10分钟以上，说明年老代OK

1. -Xms 5m设置JVM初始堆为5M，-Xmx 5m 设置JVM最大堆为5M。-Xms跟-Xmx值一样时可以避免每次垃圾回收完成后JVM重新分配内存
2. -Xmn 2g:设置年轻代大小为2G，一般默认为整个堆区的1/3 ~ 1/4。**-Xss**每个线程栈空间设置
3. -XX:SurvivorRatio，设置年轻代中Eden区与Survivor区的比值，默认=8，比值为8:1:1
4. -XX:+HeapDumpOnOutOfMemoryError** 当JVM发生OOM时，自动生成DUMP文件
5. -XX:PretenureSizeThreshold 当创建的对象超过指定大小时，直接把对象分配在老年代。
6. -XX:MaxTenuringThreshold 设定对象在Survivor区最大年龄阈值，超过阈值转移到老年代，默认15
7. 开启GC日志对性能影响很小且能帮助我们定位问题
   -XX:+PrintGCTimeStamps - XX:+PrintGCDetails -Xloggc:gc.log 日志位置