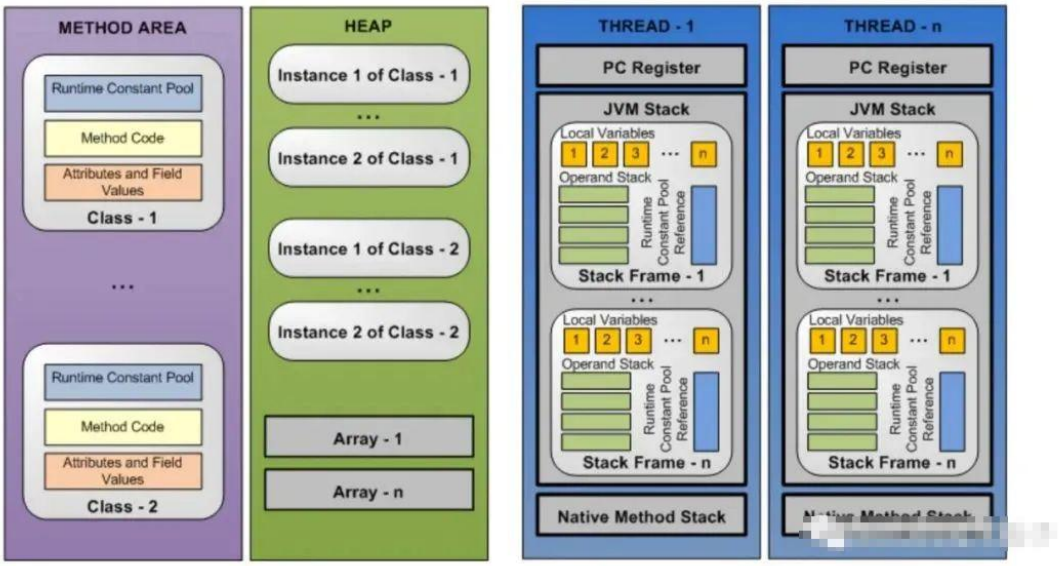
《深入理解 Java 虚拟机》第三版学习笔记

公众号后端面试那些事整理汇总

ch02. Java 内存区域与内存溢出

2.1 运行时数据区域



参考：JVM 规范，Memories of a Java Runtime

「堆」：JVM 启动时按-Xmx, -Xms 大小创建的内存区域，用于分配对象、数组所需内存，由 GC 管理和回收

「方法区」：存储被 JVM 加载的类信息（字段、成员方法的字节码指令等）、运行时常量池（字面量、符号引用等）、JIT 编译后的 Code Cache 等信息；JDK8 前 Hotspot 将方法区存储于永久代堆内存，之后参考 JRockit 废弃了永久代，存储于本地内存的 Metaspace 区

「直接内存」：JDK1.4 引入 NIO 使用 Native/Unsafe 库直接分配系统内存，使用 Buffer,Channel 与其交互，避免在系统内存与 JVM 堆内存之间拷贝的开销

「线程私有内存」：

- 程序计数器：记录当前线程待执行的下一条指令位置，上下文切换后恢复执行，由字节码解释器负责更新
- JVM 栈
  - 描述 Java 方法执行的内存模型：执行新方法时创建栈帧，存储局部变量表、操作数栈等信息
  - 存储单位：变量槽 slot，long, double 占 2 个 slot，其他基本数据类型、引用类型占 1 个，故表的总长度在编译期可知

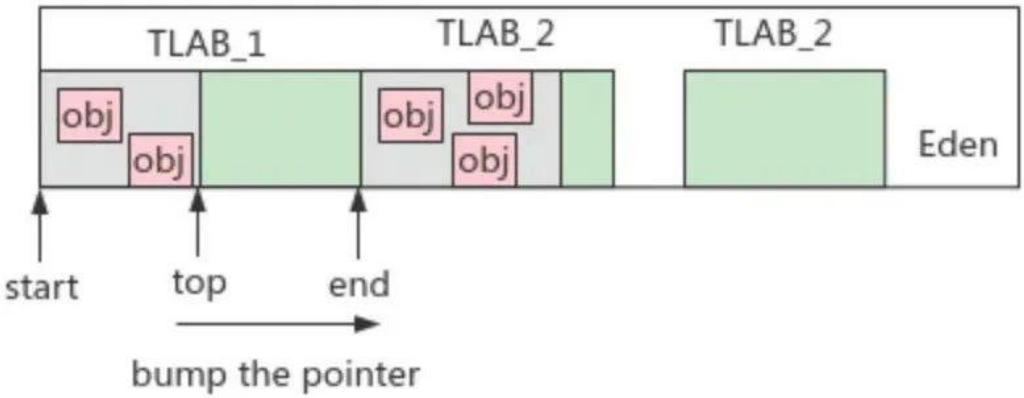
本地方法栈：执行本地 C/C++ 方法。如果您正在学习 Spring Boot，那么推荐一个连载多年还在继续更新的免费教程：
http://blog.didispace.com/spring-boot-learning-2x/

2.2 JVM 对象

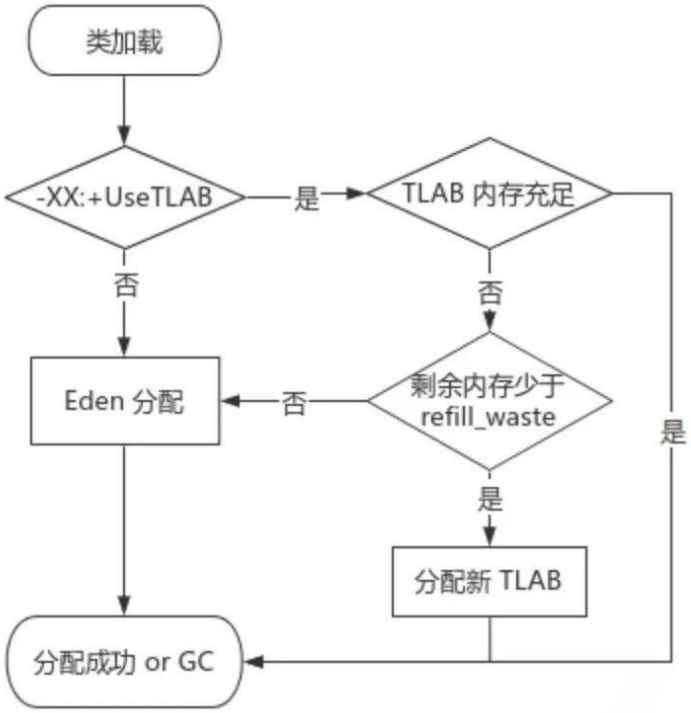
1. 创建对象

分配堆内存：类加载完毕后，其对象所需内存大小是确定的；堆内存由多线程共享，若并发创建对象都通过 CAS 乐观锁争夺内存，则效率低。故线程创建时在堆内存为其分配私有的分配缓冲区（TLAB：Thread Local Allocation Buffer）

- 内存模型



- 分配流程



注：当 TLAB 剩余空间不足以分配新对象，但又小于最大浪费空间阈值时，才会加锁创建新的 TLAB

零值初始化对象的堆内存、设置对象头信息、执行构造函数()V

2. 对象的内存布局

「对象头」

- Mark Word：记录对象的运行时信息，如 hashCode，GC 分代年龄，尾部 2 bit 用于标记锁状态

<table>
<tr><td colspan="5">Mark Word: 1 个字长 (32b or 64b)</td><td>状态</td></tr>
<tr><td colspan="2">hash code: 25b</td><td>age: 4b</td><td>biased_lock: 0</td><td>lock: 01</td><td>未锁定</td></tr>
<tr><td>thread_id: 23b</td><td>epoch:2b</td><td>age: 4b</td><td>biased_lock: 1</td><td>01</td><td>偏向锁</td></tr>
<tr><td colspan="4">pointer to thread lock record: 30b</td><td>00</td><td>轻量级锁</td></tr>
<tr><td colspan="4">pointer to heavyweight monitor: 30b</td><td>10</td><td>重量级锁</td></tr>
<tr><td colspan="4"><empty></td><td>11</td><td></td></tr>
</table>

- Class Pointer：指向所属的类信息
- 数组长度（可选，对象为数组）：4 字节存储其长度

「对象数据」：各种字段的值，按宽度分类紧邻存储

「对齐填充」：内存对齐为 1 个字长整数倍，减少 CPU 总线周期

验证：openjdk/jol 检查对象内存布局

```
1.  public class User {
2.      private int age = -1;
3.      private String name = "unknown";
4.  }
5.
6.  // java -jar ~/Downloads/jol-cli-latest.jar internals -cp . com.jol.User
7.  OFF  SZ               TYPE DESCRIPTION               VALUE
8.    0   8                    (object header: mark)     0x0000000000000001 (non
    -biasable; age: 0)
9.    8   4                    (object header: class)    0xf8021e85 // User.clas
    s 引用地址
```

```
12   4                int User.age                    -1          // 基本类型则直接存储值
16   4   java.lang.String User.name                   (object)    // 引用类型，指向运行时常量池中的 String 对象
20   4                    (object alignment gap)                  // 有 4 字节的内存填充
Instance size: 24 bytes
```

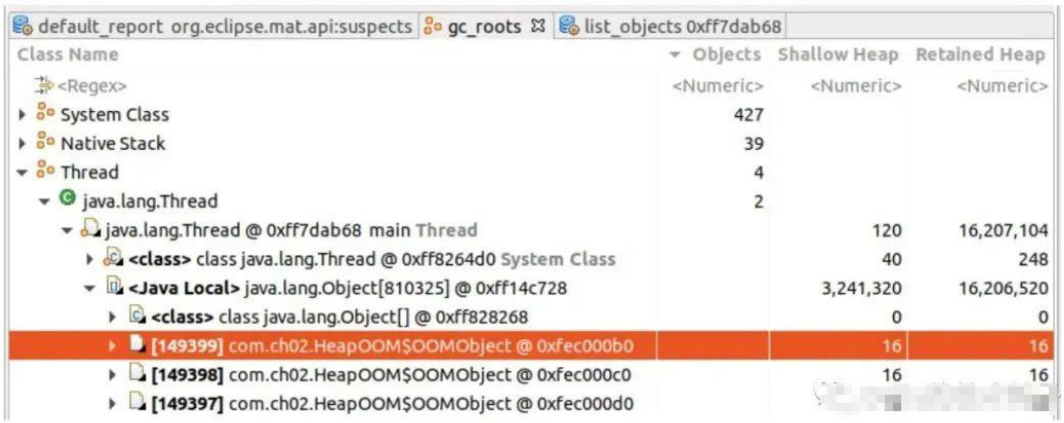
2.3 内存溢出

「堆内存」：-Xms 指定堆初始大小，当大量无法被回收的对象所占内存超出-Xmx 上限时，将发生内存溢出 OutOfMemoryError

- 排查：通过 Eclipse MAT 分析 -XX:+HeapDumpOnOutOfMemory 生成的 *.hprof 堆转储文件，定位无法被回收的大对象，找出其 GC Root 引用路径
- 解决：若为内存泄露，则修改代码用 null 显式赋值、虚引用等方式及时回收大对象；若为内存溢出，大对象都是必须存活的，则调大-Xmx、减少大对象的生命周期、检查数据结构使用是否合理等

```
// -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
public class HeapOOM {
    static class OOMObject {}
    public static void main(String[] args) {
        List<OOMObject> vs = new ArrayList<>();
        while (true)
            vs.add(new OOMObject());
    }
}
```

分析 GC Root 发现 com.ch02.HeapOOM 对象间接引用了大量的 OOMObject 对象，共占用 15.4MB 堆内存，无法回收最终导致 OOM。如果您正在学习 Spring Boot，那么推荐一个连载多年还在继续更新的免费教程：
http://blog.didispace.com/spring-boot-learning-2x/



「栈内存」：-Xss 指定栈大小，当栈深度超阈值（比如未触发终止条件的递归调用）、本地方法变量表过大等，都可能导致内存溢出 StackOverflowError

「方法区」：-XX:MetaspaceSize 指定元空间初始大小，-XX:MaxMetaspaceSize 指定最大大小，默认 -1 无限制，若在运行时动态生成大量的类，则可能触发 OOM

「运行时常量池」：strObj.intern()动态地将首次出现的字符串对象放入字符串常量池并返回，JDK7 前会拷贝到永久代，之后则直接引用堆对象

```
String s1 = "java"; // 类加载时，从字节码常量池中拷贝符号到了运行时常量池，在解析阶段初始化的字符串对象
String s2 = "j";
String s3 = s2 + "ava"; // 堆上动态分配的字符串对象
println(s3 == s1);          // false
println(s3.intern() == s1); // true // 已在字符串常量池中存在
```

「直接内存」：-XX:MaxDirectMemorySize 指定大小，默认与-Xmx 一样大，不被 GC 管理，申请内存超阈值时 OOM

ch03. 垃圾回收与内存分配

GC 可分解为 3 个子问题：which（哪些内存可被回收）、when（什么时候回收）、how（如何回收）

3.1 GC 条件

「1. 引用计数算法（reference counting）」

原理：每个对象都维护一个引用计数器 rc，当通过赋值、传参等方式引用它时 rc++，当引用变量修改指向、离开函数作用域等方式解除引用时 rc--，递减到 0 时说明对象无法再被使用，可回收。伪代码：

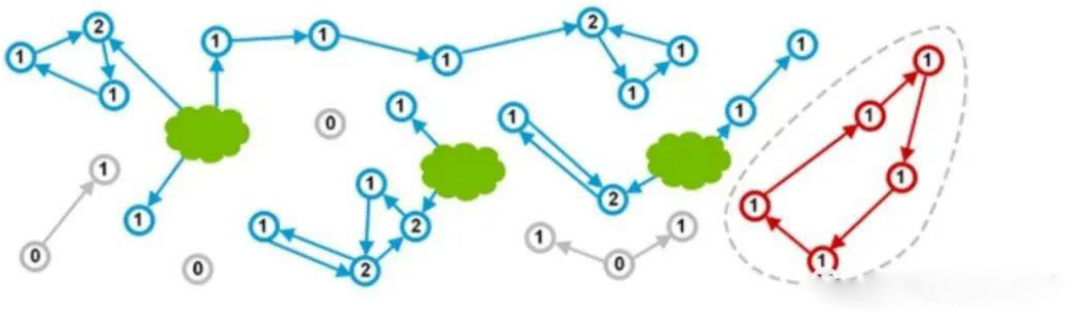
```
assign(var, obj):
    incr_ref(obj) # self = self # 先增再减，避免引用自身导致内存提前释放
    decr_ref(var)
    var = obj

incr(obj):
    obj.rc++

decr(obj):
    obj.rc--
    if obj.rc == 0:
        remove_ref(obj) # 断开 obj 与其他对象的引用关系
        gc(obj)         # 回收 obj 内存
```

优点：思路简单，对象无用即回收，延迟低，适合内存少的场景

缺点：此算法中对象是孤立的，无法在全局视角检查对象的真实有效性，循环引用的双方对象需引入外部机制来检测和回收，如下图红色圈（图源：what-is-garbage-collection）



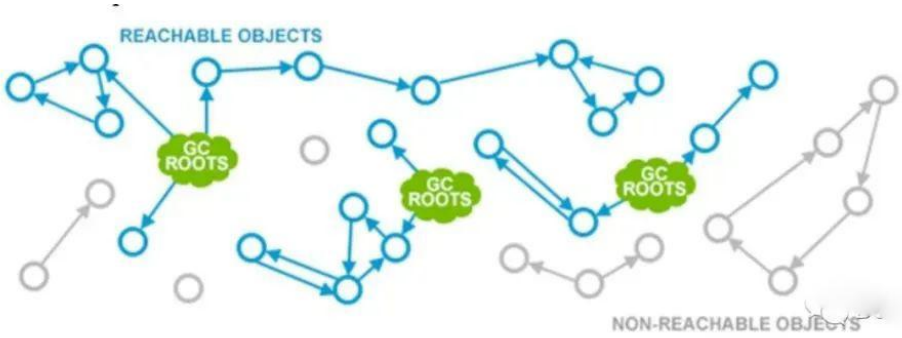
「2. 可达性分析算法（reachability analysis）」

原理：从肯定不会被回收的对象（GC Roots）出发，向外搜索全局对象图，不可达的对象即无法再被使用，可回收；常见可作为 GC Root 的对象有：

- 执行上下文：JVM 栈中参数、局部变量、临时变量等引用的堆对象
- 全局引用：方法区中类的静态引用、常量引用（如 StringTable 中的字符串对象）所指向的对象

优点：无需对象维护 GC 元信息，开销小；单次扫描即可批量识别、回收对象，吞吐高

缺点：多线程环境下对象间的引用关系随时在变化，为保证 GC Root 标记的准确性，需在不变化的 snapshot 中进行，会产生 Stop The World（以下简称 STW） 卡顿现象



「3. 四种引用类型」

引用类型	类	回收时机
强引用	-	只要与 GC Root 存在引用链，则不被回收
软引用	SoftReference	只被软引用所引用的对象，当 GC 后内存依然不足，才被回收
弱引用	WeakReference	只被弱引用所引用的对象，无论内存是否足够，都将被回收
虚引用	PhantomReference	被引用的对象无感知，进行正常 GC，仅在回收时通知虚引用（回调）

示例：限制堆内存 50MB，其中新生代 30MB，老年代 20MB；依次分配 5 次 10MB 的 byte[]对象，仅使用软引用来引用，观察 GC 过程

```
1.  public static void main(String[] args) {
2.      // softRefList --> SoftReference --> 10MB byte[]
3.      List<SoftReference<byte[]>> softRefList = new ArrayList<>();
4.      ReferenceQueue<byte[]> softRefQueue = new ReferenceQueue<>(); // 无效引用队列
5.      for (int i = 0; i < 5; i++) {
6.          SoftReference<byte[]> softRef = new SoftReference<>(new byte[10*1024*1024], softRefQueue);
7.          softRefList.add(softRef);
8.
9.          for (SoftReference<byte[]> ref : softRefList) // dump 所有软引用指向的对象，检查是否已被回收
10.             System.out.print(ref.get() == null ? "gced " : "ok ");
11.         System.out.println();
12.     }
13.     Reference<? extends byte[]> ref = softRefQueue.poll();
14.     while (ref != null) {
15.         softRefList.remove(ref); // 解除对软引用对象本身的引用
16.         ref = softRefQueue.poll();
17.     }
18.     System.out.println("effective soft ref: " + softRefList.size()); // 2
```

```
}

// java -verbose:gc -XX:NewSize=30m -Xms50m -Xmx50m -XX:+PrintGCDetails com.ch02.DemoRef
ok
ok ok
// 分配第三个 []byte 时，Eden GC 无效，触发 Full GC 将一个 []byte 晋升到老年区
// 此时三个 byte[] 都只被软引用所引用，被标记为待二次回收（若为弱引用，此时 Eden 已被回收）
[GC (Allocation Failure) --[PSYoungGen: 21893K->21893K(27136K)] 21893K->32141K(47616K), 0.0046324 secs]
[Full GC (Ergonomics) [PSYoungGen: 21893K->10527K(27136K)] [ParOldGen: 10248K->10240K(20480K)] 32141K->20767K(47616K), [Metaspace: 2784K->2784K(1056768K)], 0.004 secs]
ok ok ok
// 再次 GC，前三个 byte[] 全部被回收
[GC (Allocation Failure) --[PSYoungGen: 20767K->20767K(27136K)] 31007K->31007K(47616K), 0.0007963 secs]
[Full GC (Ergonomics) [PSYoungGen: 20767K->20759K(27136K)] [ParOldGen: 10240K->10240K(20480K)] 31007K->30999K(47616K), [Metaspace: 2784K->2784K(1056768K)], 0.003 secs]
[GC (Allocation Failure) --[PSYoungGen: 20759K->20759K(27136K)] 30999K->30999K(47616K), 0.0007111 secs]
[Full GC (Allocation Failure) [PSYoungGen: 20759K->0K(27136K)] [ParOldGen: 10240K->267K(20480K)] 30999K->267K(47616K), [Metaspace: 2784K->2784K(1056768K)], 0.003 secs]
gced gced gced ok
gced gced gced ok ok
```

「4. finalize」

原理：若对象不可达，被标记为可回收后，会进行 finalize()是否被重写、是否已执行过等条件筛选，若通过则对象会被放入 F-Queue 队列，等待低优先级的后台 Finalizer 线程触发其 finallize() 的执行（不保证执行结束），对象可在 finalize 中建立与 GC Root 对象图上任一节点的引用关系，来逃脱 GC

使用：finalize 机制与 C++ 中的析构函数并不等价，其执行结果并不确定，不推荐使用，可用 try-finally 替代。如果您正在学习 Spring Boot，那么推荐一个连载多年还在继续更新的免费教程：http://blog.didispace.com/spring-boot-learning-2x/

3.2 GC 算法

「分代收集理论」

两个分代假说：符合大多数程序运行的实际情况

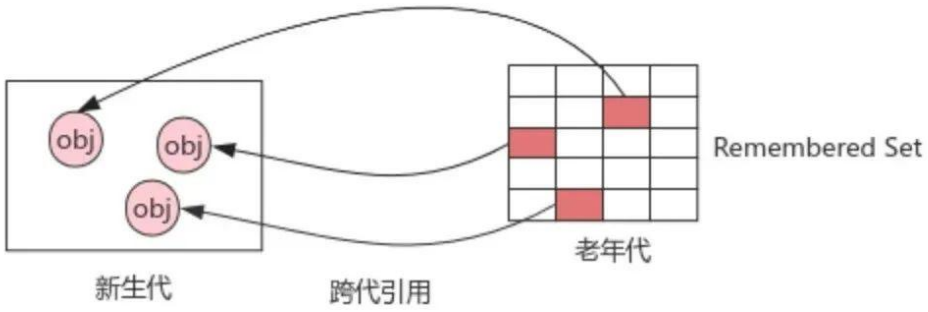
- 弱分代假说：绝大多数对象是朝生夕灭，生存时间极短
- 强分代假说：熬过越多次 GC 的对象，越可能被继续使用，越难以回收

对应地，JVM 堆被划分为 2 个不同区域，将对象按年龄分类，兼顾了 GC 耗时与内存利用率

- 新生代：大量对象将被回收，只关注「仍存活」的对象，逐步晋升
- 老年代：大量对象不被回收，只关注「要被回收的」对象

跨代引用

- 问题：老年代会引用新生代，新生代 GC 时需遍历老年代中大量的存活对象，分析可达性，时间复杂度高
- 背景：相互引用的对象倾向于同时存亡，比如跨代引用关系中的新生代必然会逐步晋升，最终消除跨代关系
- 假说：跨代引用相比同代引用只占极少数，无需全量扫描老年代
- 实现：新生代维护全局数据结构：记忆集（Remembered Set），将老年代分为多个子块，标记存在跨代引用的子块，等待后续扫描；代价：为保证记忆集的正确性，需在跨代引用建立或断开时保持同步

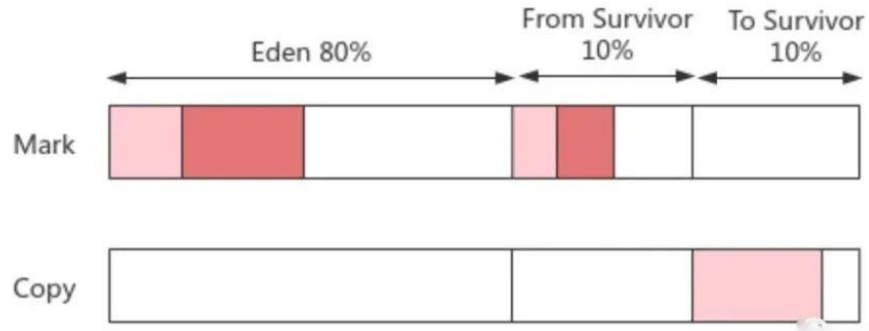


「1. 标记清除：Mark-Sweep」

- 原理：标记不可达对象，统一清理回收，反之亦可
- 缺点：执行效率不稳定，回收耗时取决于活跃对象的数量；内存碎片多，会出现内存充足但无法分配过大的连续内存（数组）

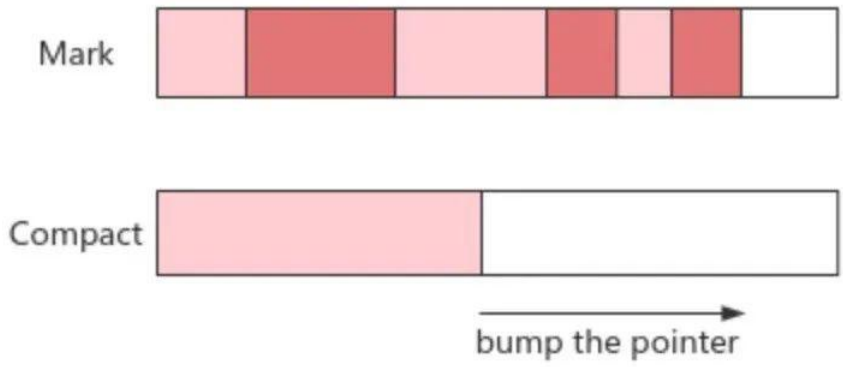
「2. 标记复制：Mark-Copy」

- 理论：将堆内存切为两等份 A, B，每次仅使用 A，用完后标记存活对象复制到 B，清空 A 后执行 swap
- 优点：直接针对半区回收，无内存碎片问题；分配内存只需移动堆顶指针，高效顺序分配
- 缺点：当 A 区有大量存活对象时，复制开销大；B 区长时间闲置，内存浪费严重
- 实践：对于存活对象少的新生代，无需按 1:1 分配，而是按 8:1:1 的内存布局，其中 Eden 和 From 区同时使用，只有 To 区会被闲置（担保机制：若 To 区不够容纳 Minor GC 后的存活对象，则晋升到老年区）



「3. 标记整理：Mark-Compact」

- 原理：标记存活对象后统一移动到内存空间一侧，再回收边界之外的内存
- 优点：内存模型简单，无内存碎片，降低内存分配和访问的时间成本，能提高吞吐
- 缺点：对象移动需 STW 同步更新引用关系，会增加延迟

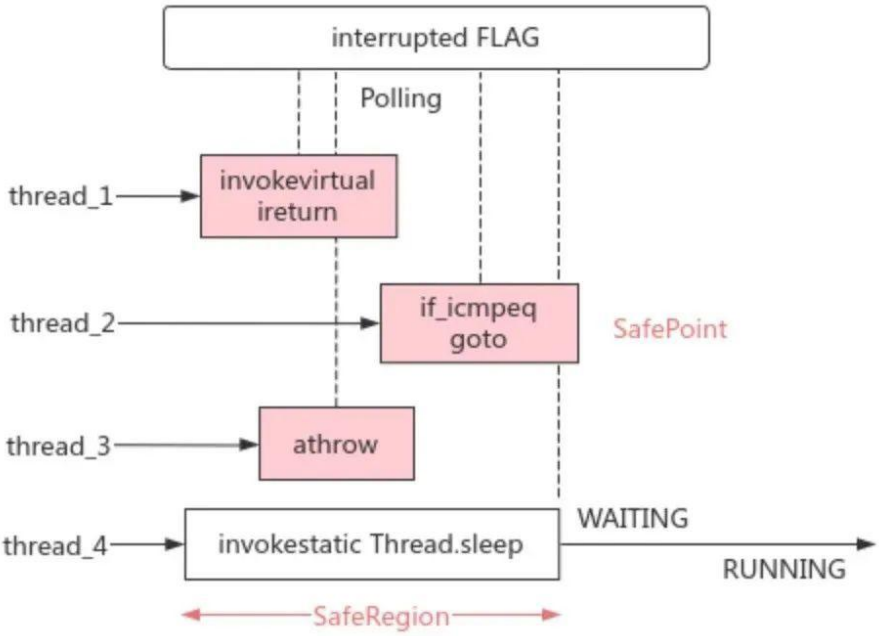


3.3 HotSpot GC 算法细节

「1. 发起 GC：安全点与安全区域」

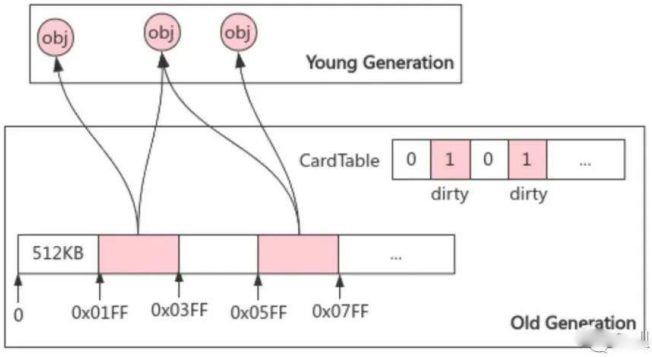
- 问题：为保证可达性分析结果的准确性，需挂起用户线程（STW），再从各线程的执行上下文中收集 GC Root，如何通知线程挂起？
- 安全点：HotSpot 内部有线程中断标记；在各线程的方法调用、循环跳转、异常跳转等会长时间执行的指令处，额外插入检查该标记的 test 高效指令；若轮询发现标记为真，线程会主动在最近的 SafePoint 处挂起，此时其栈上对象的引用关系不再变化，可收集 GC Root 对象
- 安全区域：引用关系不会变化的指令区域，可安全地收集 GC Root；线程离开此区域时，若 GC Root 收集过程还未结束，则需等待

示意图



「2. 加速 GC：CardTable」问题：非收集区域（老年代）会存在到收集区域（新生代）的跨代引用，如何避免对前者的全量扫描？

- 卡表：记忆集的字节数组实现；将老年代内存划分为 Card Page（512KB）大小的子内存块，若新建跨代引用，则将对应的 Card 标记为 dirty，GC 时只需扫描老年代中被标记为 dirty 的子内存块

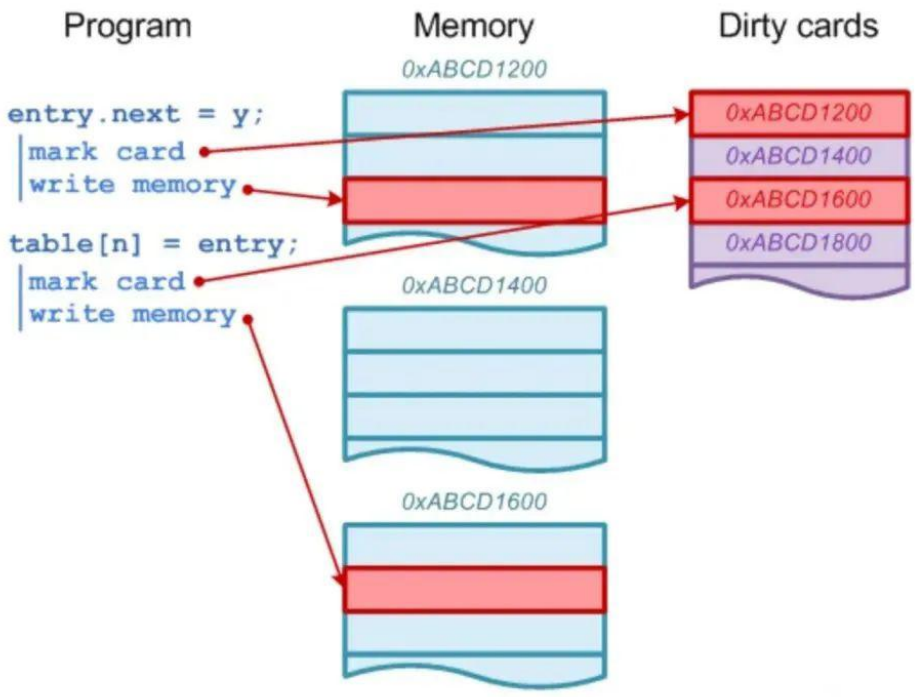


- 写屏障：有别于 volatile 禁用指令重排的内存屏障，GC 中的写屏障是在对象引用更新时执行额外 hook 动作的机制。简单实现：

```
void oop_field_store(oop* field, oop new_val) { // oop: ordinary object pointer
    // pre_write_barrier(field, new_val); // 写前屏障：更新前先执行，使用 oop 旧状态
    *field = new_val;
    post_write_barrier(field, new_val); // 写后屏障：更新完才执行
```

```
5.  }
```

使用写屏障保证 CardTable 的实时更新（图源：The JVM Write Barrier - Card Marking）



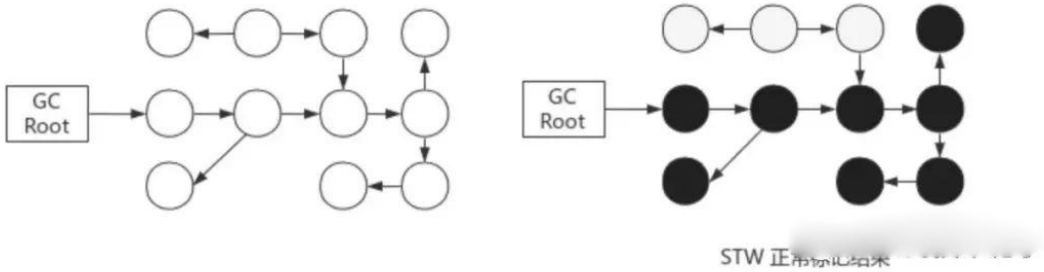
「3. 正确 GC：并发可达性分析」参考演讲：Shenandoah: The Garbage Collector That Could by Aleksey Shipilev

问题：GC Roots 的对象源固定，故枚举时 STW 时间短暂且可控。但后续可达性分析的时间复杂度与堆中对象数量成正相关，即堆中对象越多，对象图越复杂，堆变大后 STW 时间不可接受

解决：并发标记。引出新问题：用户线程动态建立、解除引用，标记过程中图结构发生变化，结果不可靠；证明：用三色法描述对象状态

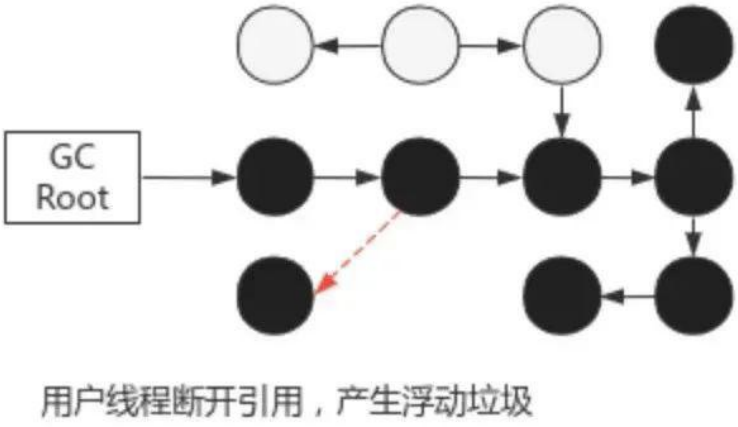
- 白色：未被回收器访问过的对象；分析开始都是白色，分析结束还是白色则不可达
- 灰色：被回收器访问过，但其上至少还有 1 个引用未被扫描（中间态）
- 黑色：被回收器访问过，其上引用全部都已被扫描，存在引用链，为存活对象；若其他对象引用了黑色对象，则不必再扫描，肯定也存活；黑色不可能直接引用白色

STW 无并发的正确标记：顶部 3 个对象将被回收

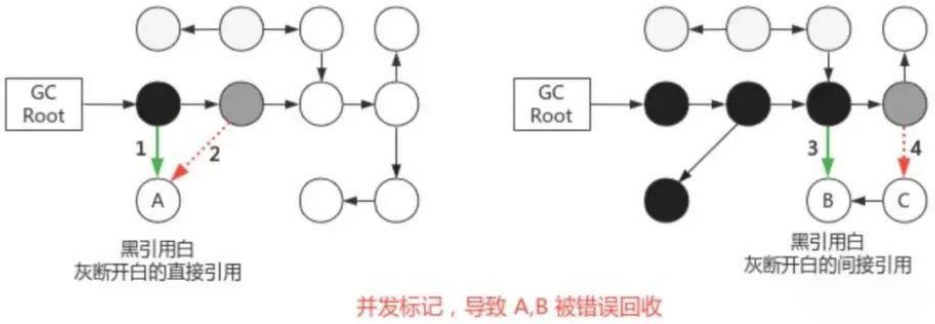


用户线程并发修改引用，会导致标记结果无效，分 2 种情况：

- 少回收，对象标记为存活，但用户解除了引用：产生浮动垃圾，可接受，等待下次 GC



- 误回收，对象标记为可回收，但用户新建了引用：实际存活对象被回收，内存错误



论文《Uniprocessor Garbage Collection Techniques - Paul R. Wilson》§3.2 证明了「实际存活的对象被标记为可回收」必须同时满足两个条件（有时间序）

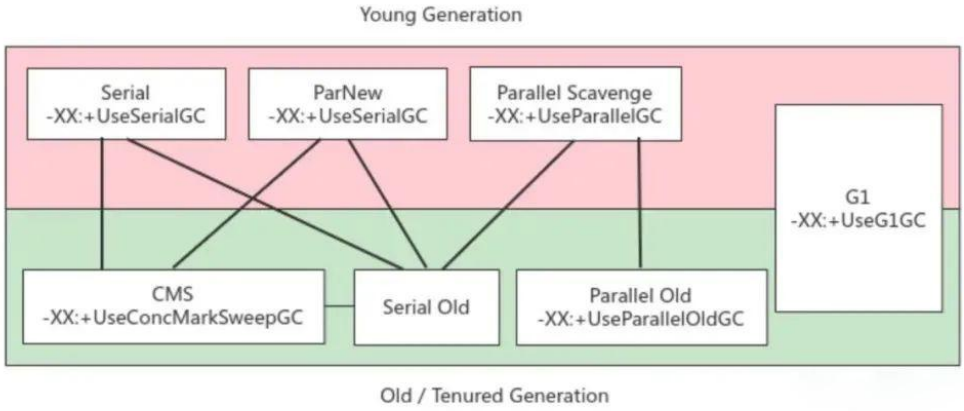
- 插入一条或多条从黑色到白色的新引用
- 删除所有灰色到该白色的直接、间接引用

为正确实现标记，打破其中一个条件即可（类比打破死锁四个条件之一的思想），分别对应两种方案：

- 增量更新 Increment Update：记录黑到白的引用关系，并发标记结束后，以黑为根，重新扫描；A 直接存活
- 原始快照 SATB（Snapshot At The Begining）：记录灰到白的解引用关系，并发标记结束后，以灰为根，重新扫描；B 为灰色，最后变为黑色，存活。需注意，若没有步骤 3，则 B,C 变为浮动垃圾

3.4 经典垃圾回收器

搭配使用示意图：



「1. Serial, SerialOld」

原理：内存不足触发 GC 后会暂停所有用户线程，单线程地在新生代中标记复制，在老年代中标记整理，收集完毕后恢复用户线程

优点：全程 STW 简单高效

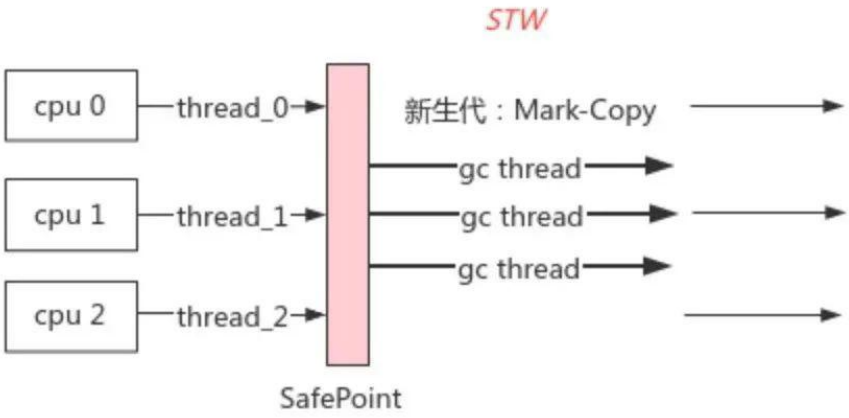
缺点：STW 时长与堆对象数量成正相关，且 GC 线程只能用到 1 core 无法加速

场景：单核 CPU 且可用内存少（如百兆级），JDK1.3 之前的唯一选择

「2. ParNew」

原理：多线程并行版的 Serial 实现，能有效减少 STW 时长；线程数默认与核数相同，可配置

场景：JDK7 之前搭配老年代的 CMS 回收器使用

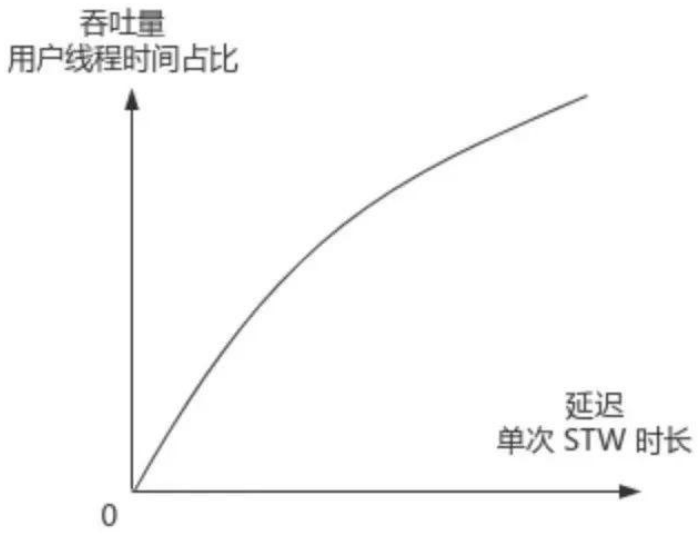


ParNew 工作流程

「3. Parallel, Parallel Old」

垃圾回收有两个通常不可兼得的目标

- 低延迟：STW 时长短，响应快；允许高频、短暂 GC，比如调小新生代空间，加快收集延迟（吞吐下降）
- 高吞吐量：用户线程耗时 /（用户线程耗时 + GC 线程耗时）高，GC 总时间低；允许低频、单次长时间 GC，（延迟增加）



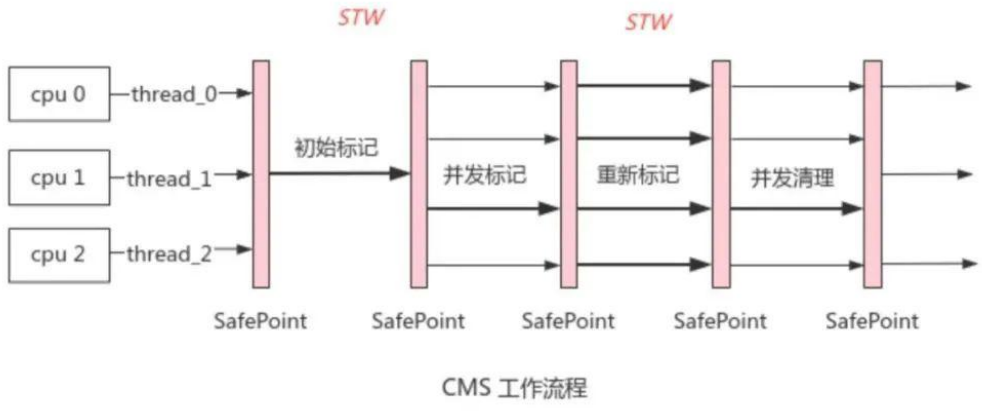
原理：与 ParNew 类似都是并行回收，主要增加了 3 个选项（倾向于提高吞吐量）

- -XX:MaxGCPauseTime：控制最大延迟
- -XX:GCTimeRatio：控制吞吐（默认 99%）
- -XX:+UseAdaptiveSizePolicy ：启用自适应策略，自动调整 Eden 与 2 个 Survivor 区的内存占比-XX:SurvivorRatio，老年代晋升阈值 -XX:PretenureSizeThreshold

「4. CMS」

CMS：Concurrent Mark Sweep，即并发标记清除，主要有 4 个阶段

- 初始标记（initial mark）：STW 快速收集 GC Roots
- 并发标记（concurrent mark）：从 GC Roots 出发检测引用链，标记可回收对象；与用户线程并发执行，通过增量更新来避免误回收
- 重新标记（remark）：STW 重新分析被增量更新所收集的 GC Roots
- 并发清除（concurrent sweep）：并发清除可回收对象



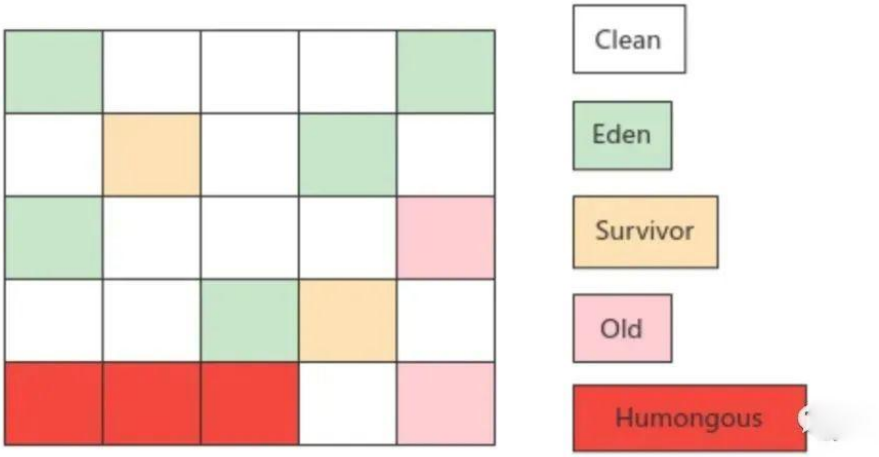
优点：两次 STW 时间相比并发标记耗时要短得多，相比前三种收集器，延迟大幅降低

缺点

- CPU 敏感：若核数较少（< 4core），并发标记将占用大量 CPU 时间，会导致吞吐突降
- 无法处理浮动垃圾：-XX:CMSInitiatingOccupancyFration（默认 92%）指定触发 CMS GC 的阈值；在并发标记、并发清理的同时，用户线程会产生浮动垃圾（引用可回收对象、产生新对象），若浮动垃圾占比超过 -XX:CMSInitiatingOccupancyFration；若 GC 的同时产生过多的浮动垃圾，导致老年代内存不足，会出现 CMS 并发失败，退化为 Serial Old 执行 Full GC，会导致延迟突增
- 无法避免内存碎片：-XX:CMSFullGCsBeforeCompaction（默认 0）指定每次在 Full GC 前，先整理老年代的内存碎片

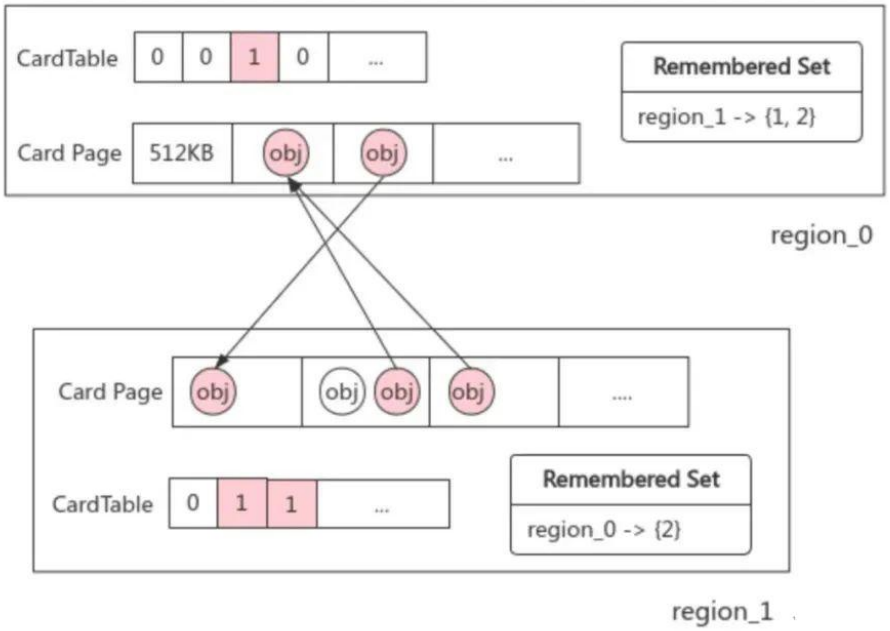
「5. G1」

特点：基于 region 内存布局实现局部回收；GC 延迟目标可配置；无内存碎片问题



	G1	之前回收器
堆内存划分方式	多个等大的 region，各 region 分代角色并不固定，按需在 Eden, Survivor, Old 间切换	固定大小、固定数量的分代区域
回收目标	回收价值高的 region 动态组成的回收集合	新生代、整个堆内存

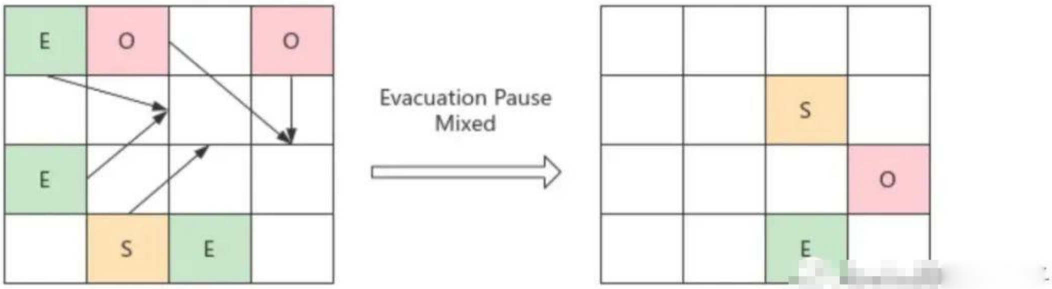
跨代引用：各 region 除了用卡表标记各卡页是否为 dirty 之外，还用哈希表记录了各卡页正在被哪些 region 引用，通过这种“双向指针”机制，能直接找到 Old 区，避免了全量扫描（G1 自身内存开销大头）



G1 GC 有 3 个阶段（参考其 GC 日志）

- 新生代 GC：Eden 区占比超阈值触发；标记存活对象并复制到 Survivor 区，其内可能有对象会晋升到 Old 区

- 老年代 GC：Old 区占比达到阈值后触发，执行标记整理
  - 初始标记：枚举 GC Roots，已在新生代 GC 时顺带完成
  - 并发标记：并发执行可达性分析，使用 SATB 记录引用变更
  - 重新标记：SATB 分析，避免误回收
  - 筛选回收：将 region 按回收价值和时间成本筛选组成回收集，STW 将存活对象拷贝到空 regions 后清理旧 regions，完成回收
- 混合 GC



参数控制（文档：HotSpot GC Tuning Guide）

参数及默认值	描述
-XX:+UseG1GC	JDK9 之前手动启用 G1
-XX:MaxGCPauseMillis=200	预期的最大 GC 停顿时间；不宜过小，避免每次回收内存少而导致频繁 GC
-XX:ParallelGCThreads=N	STW 并行线程数，若可用核数 M < 8 则 N=1，否则默认 N=M*5/8
-XX:ConcGCThreads=N	并发阶段并发线程数，默认是 ParallelGCThreads 的 1/4
-XX:InitiatingHeapOccupancyPercent=45	老年代 region 占比超过 45% 则触发老年代 GC
-XX:G1HeapRegionSize=N	单个 region 大小，1~32MB
-XX:G1NewSizePercent=5, -XX:G1MaxNewSizePercent=60	新生代 region 最小占整堆的 5%，最大 60%，超出则触发新生代 GC
-XX:G1HeapWastePercent=5	允许浪费的堆内存占比，可回收内存低于 5% 则不进行混合回收
-XX:G1MixedGCLiveThresholdPercent=85	老年代存活对象占比超 85%，回收价值低，暂不回收
-XX:G1MixedGCCountTarget=8	单次收集中混合回收次数

3.8 内存分配策略

使用 Serial 收集器 -XX:+UseG1GC 演示

「**1. 对象优先分配在 Eden 区**」

新对象在 Eden 区分配，空间不足则触发 Minor GC，存活对象拷贝到 To Survivor，若还是内存不足则通过分配担保机制转移到老年区，依旧不足才 OOM

```
byte[] buf1 = new byte[6 * MB];
byte[] buf2 = new byte[6 * MB]; // 10MB 的 eden 区剩余 4MB，空间不足，触发 minor GC

```

```
// java -verbose:gc -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+UseSerialGC com.ch03.Allocation
// minor gc 后新生代内存从 6M 降到 0.2M，存活对象移到了老年区，总的堆内存用量依旧是 6MB
[GC (Allocation Failure) [DefNew: 6823K->286K(9216K), 0.002 secs] 6823K->6430K(19456K), 0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 6513K
  eden space 8192K,  76% used // buf2
  from space 1024K,  28% used
  to   space 1024K,   0% used
 tenured generation   total 10240K, used 6144K
   the space 10240K,  60% used // buf1
```

「2. 大对象直接进入老年区」

对于 Serial, ParNew，可配置超过阈值 -XX:PretenureSizeThreshold 的大对象（连续内存），直接在老年代中分配，避免触发 minor gc，导致 Eden 和 Survivor 产生大量的内存复制操作

```
byte[] buf1 = new byte[4 * MB];

// java -verbose:gc -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+UseSerialGC
// -XX:PretenureSizeThreshold=3145728 com.ch03.Allocation // 3145728 即 3MB

Heap
 def new generation   total 9216K, used 843K
  eden space 8192K,  10% used
  from space 1024K,   0% used
  to   space 1024K,   0% used
 tenured generation   total 10240K, used 4096K
   the space 10240K,  40% used // buf1
```

「3. 长期存活的对象进入老年代」

对象头中 4bit 的 age 字段存储了对象当前 GC 分代年龄，当超过阈值 -XX:MaxTenuringThreshold（默认 15，也即 age 字段最大值）后，将晋升到老年代，可搭配-XX:+PrintTenuringDistribution 观察分代分布

```
byte[] buf1 = new byte[MB / 16];
byte[] buf2 = new byte[4 * MB];
byte[] buf3 = new byte[4 * MB]; // 触发 minor gc
buf3 = null;
buf3 = new byte[4 * MB];
```

```
6.
7.  // java -verbose:gc -Xms20m -Xmx20m -Xmn10m -XX:+PrintGCDetails -XX:+UseSerialGC
8.  //  -XX:MaxTenuringThreshold=1  -XX:+PrintTenuringDistribution  com.ch03.Allocation
9.  [GC (Allocation Failure) [DefNew
10. Desired survivor size 524288 bytes, new threshold 1 (max 1)
11. - age   1:     359280 bytes,     359280 total
12. : 4839K->350K(9216K)] 4839K->4446K(19456K), 0.0017247 secs]
13. // 至此，buf1 熬过了第一次收集，age=1
14. [GC (Allocation Failure) [DefNew
15. Desired survivor size 524288 bytes, new threshold 1 (max 1): 4446K->0K(9216K)] 8542K->4438K(19456K)]
16. Heap
17.  def new generation   total 9216K, used 4178K
18.   eden space 8192K,  51% used
19.   from space 1024K,   0% used // buf1 在第二轮收集中被提前晋升
20.   to   space 1024K,   0% used
21.  tenured generation   total 10240K, used 4438K
22.    the space 10240K,  43% used
```

「4. 分代年龄动态判定」

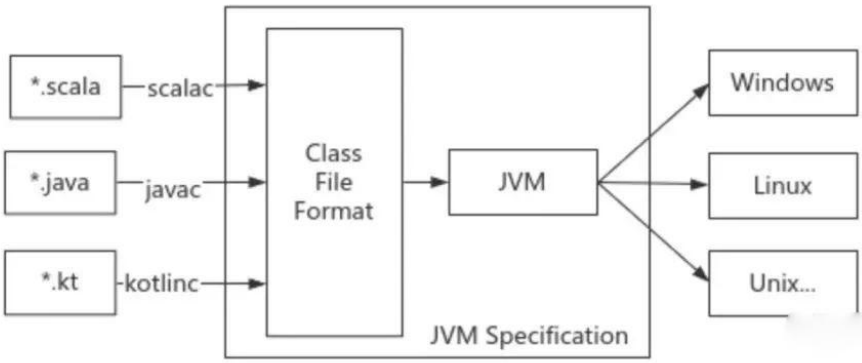
-XX:MaxTenuringThreshold 并非晋升的最低硬性门槛，当 Survivor 中同龄对象超 50% 后，大于等于该年龄的对象会被自动晋升，哪怕还没到阈值

「5. 空间分配担保」

老年代作为 To Survivor 区的担保区域，当 Eden + From Survivor 中存活对象的总大小超出 To Survivor 时，将尝试存入老年代。JDK6 之后，只要老年代的连续空间大于新生代对象的总大小，或之前晋升的平均大小，则只会进行 Minor GC，否则进行 Full GC

ch06. 类文件结构

Class 文件实现语言无关性，JVM 实现平台无关性，参考《Java 虚拟机规范》



一个 Class 文件描述了一个类或接口的明确定义，文件内容是一组以 8 字节为单位的二进制流，各数据项间没有分隔符，超过 8 字节的数据项按 Big-Endian 切分后存储。数据项分两种：

- 无符号数：描述基本类型；用 u1,u2,u4,u8 分别表示 1,2,4,8 字节长度的无符号数；存储数字值、索引序号、UTF-8 编码值等
- 表：由无符号数、其他表嵌套构成的复合类型；约定 _info 后缀；存储字段类型、方法签名等

6.1 结构定义

1. 语法 参考文档：The class File Format

```
ClassFile {
    u4             magic;         // 魔数
    u2             minor_version; // 版本号
    u2             major_version;
    u2             constant_pool_count; // 常量池
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;      // 类访问标记
    u2             this_class;        // 本类全限定名
    u2             super_class;       // 单一父类
    u2             interfaces_count; // 多个接口
    u2             interfaces[interfaces_count];
    u2             fields_count;  // 字段表
    field_info     fields[fields_count];
    u2             methods_count; // 方法表
    method_info    methods[methods_count];
    u2             attributes_count; // 类属性
    attribute_info attributes[attributes_count];
}
```


- magic：魔数，简单识别 *.class 文件，值固定为 0xCAFEBABE

- minor_version, major_version：Class 文件的次、主版本号
- constant_pool_count：常量池大小+1
- constant_pool：常量池，索引从 1 开始，0 值被预留表示不引用任何常量池中的任何常量；常量分两类
  - 字面量：如 UTF8 字符串、int、float、long、double 等数字常量
  - 符号引用：类、接口的全限定名、字段名与描述符、方法类型与描述符等 现有常量共计 17 种，常量间除了都使用 u1 tag 前缀标识常量类型外，结构互不相同，常见的有：
  - CONSTANT_Utf8_info：保存由 UTF8 编码的字符串

```
CONSTANT_Utf8_info {
    u1 tag;           // 值为 1
    u2 length;        // bytes 数组长度，u2 最大值 65535，即单个字符串字面量不超过 64KB
    u1 bytes[length]; // 长度不定的字节数组
}
```

- CONSTANT_Class_info：表示类或接口的符号引用

```
CONSTANT_Class_info {
    u1 tag;        // 值为 7
    u2 name_index; // 指向全限定类名的 Utf8_info // 常量间存在层级组合关系
}
```

- CONSTANT_Fieldref_info, CONSTANT_Methodref_info, CONSTANT_NameAndType_info：成员变量、成员方法及其类型描述符

```
CONSTANT_Fieldref_info {
    u1 tag;                // 值为 9
    u2 class_index;        // 所属类
    u2 name_and_type_index; // 字段的名称、类型描述符
}
CONSTANT_Methodref_info {
    u1 tag;                // 值为 10
    u2 class_index;        // 所属类
    u2 name_and_type_index; // 方法的名称、签名描述符
}
CONSTANT_NameAndType_info {
    u1 tag;              // 值为 12
    u2 name_index;       // 字段或方法的名称
    u2 descriptor_index; // 类型描述符
}
```

如上只列举了其中 5 种常量的结构，可见常量间通过组合的方式，来描述层级关系

- access_flags：类的访问标记，有 16bit，每个标记对应一个位，比如 ACC_PUBLIC 对应 0x0001，表示类被 public 修饰；其他 8 个标记参考 Opcodes.ACC_XXX
- this_class, super_class：指向本类、唯一父类的 Class_info 符号常量
- interface_count, interfaces：描述此类实现的多个接口信息
- fields_count, fields：字段表；描述类字段、成员变量的个数及详细信息

```
field_info {
    u2             access_flags;     // 作用域、static,final,volatile 等访问标记
    u2             name_index;       // 字段名
    u2             descriptor_index; // 类型描述符
    u2             attributes_count; // 字段的属性表
    attribute_info attributes[attributes_count];
}
```

类型描述符简化描述了字段的数据类型、方法的参数列表及返回值，与 Java 中的类型对于关系如下：

- 基本类型：Z:boolean, B:byte, C:char, S:short, I:int, F:float, D:double, J:long
- void 及引用类型：V:void
- 引用类型：L:_，类名中的 . 替换为 /，添加 ; 分隔符，如 Object 类描述为 Ljava/lang/Object;
- 数组类型：每一维用一个前置 [表示 示例：boolean regionMatch(int, String, int, int) 对应描述符为 (ILjava/lang/String;II)Z
- methods_count, methods：方法表；完整描述各成员方法的修饰符、参数列表、返回值等签名信息

```
method_info {
    u2             access_flags;     // 访问标记
    u2             name_index;       // 方法名
    u2             descriptor_index; // 方法描述符
    u2             attributes_count; // 方法属性表
    attribute_info attributes[attributes_count];
}
```

字段表、方法表都可以带多个属性表，如常量字段表、方法字节码指令表、方法异常表等。属性模板：

```
attribute_info {
    u2 attribute_name_index;    // 属性名
    u4 attribute_length;        // 属性数据长度
    u1 info[attribute_length]; // 其他字段，各属性的结构不同
}
```

属性有 20+ 种，此处只记录常见的三种

- Code 属性：存储方法编译后的字节码指令

```
Code_attribute {
    u2 attribute_name_index; // 属性名，指向的 Utf8_info 值固定为 "Code"
    u4 attribute_length;     // 剩下字节长度
    u2 max_stack;  // 操作数栈最大深度，对于此方法的栈帧中操作数栈的深度
    u2 max_locals; // 以 slot 变量槽为单位的局部变量表大小，存储隐藏参数 this，实参列表，catch 参数，局部变量等
    u4 code_length;        // 字节码指令总长度
    u1 code[code_length]; // JVM 指令集大小 200+，单个指令的编号用 u1 描述
     u2 exception_table_length; // 异常表，描述方法内各指令区间产生的异常及其 handler 地址
    {   u2 start_pc;   // catch_type 类型的异常，会在 [start_pc, end_pc) 指令范围内抛出
        u2 end_pc;
        u2 handler_pc; // 若抛出此异常，则 goto 到 handler_pc 处执行
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count; // Code 属性自己的属性
    attribute_info attributes[attributes_count];
}
```

- LineNumberTable 属性：记录 Java 源码行号与字节码行号的对应关系，用于抛异常时显示堆栈对应的行号等信息。可作为 Code 属性的子属性

```
LineNumberTable_attribute {
    u2 attribute_name_index; u4 attribute_length;
    u2 line_number_table_length;
    {   u2 start_pc;     // 字节码指令区间开始位置
        u2 line_number;  // 对应的源码行号
    } line_number_table[line_number_table_length];
}
```

- LocalVariableTable 属性：记录 Java 方法中局部变量的变量名，与栈帧局部变量表中的变量的对应关系，用于保留各方法有意义的变量名称

```
LocalVariableTable_attribute {
    u2 attribute_name_index; u4 attribute_length;
    u2 local_variable_table_length;
    {   u2 start_pc; // 局部变量生命周期开始的字节码偏移量
        u2 length;   // 向后生命周期覆盖的字节码长度
        u2 name_index;       // 变量名
```

```
        u2 descriptor_index; // 类型描述符
        u2 index; // 对应的局部变量表中的 slot 索引
    } local_variable_table[local_variable_table_length];
}
```

其他属性直接参考 JVM 文档

. 示例 源码：com/cls/Structure.java

```
package com.cls;

public class Structure {
    public static void main(String[] args) {
        System.out.println("hello world");
    }
}
```

javac -g:lines com/cls/Structure.java 编译后，参考 javap 反编译得到的正确结果，od -x --endian=big Structure.class 得出 class 文件内容的十六进制表示，解读如下：

```
cafe babe # 1.  u4 魔数，标识 class 文件类型
0000 0034 # 2.  u2,u2 版本号，52 JDK8

# 3. 常量池
---1---
001f # u2 constant_pool_count，31 项（从 1 开始计数，0 预留）
0a      # u1 tag，10，Methoddef_info，成员方法结构
0006        # u2 index，6，所属类的 Class_info 在常量池中的编号  ## java/lang/Object
0011    # u2 index，17，此方法 NameAndType 编号             ## <init>:()V

---2---
09      # 9，Fileddef_info，成员变量结构
0012    # u2 index，18，所属类 Class_info 编号     ## java/lang/System
0013            # u2 index，19，此字段 NameAndType 编号    ## out:Ljava/io/PrintStream

---3---
08      # 8，String_info，字符串
0014    # u2 index，20，字面量编号      ## hello world

---4---
0a
0015    # 21    ## java/io/PrintStream
0016    # 22    ## println:(Ljava/lang/String;)V

---5---
```

```
07      # Class_info，全限定类名
0017    # u2 index，23，字面量编号      ## com/cls/Structure

---6---
07      # 7，Class_info，类引用
0018    # 24    ## java/lang/Object

---7---
01      # Utf8_info，UTF8 编码的字符串
0006    # u2 length，6，字符串长度
3c 69 6e 69 74 3e # 字面量值     ## "<init>"

---8-16---
01 0003 282956                                          ## "()V"
01 0004 436f6465                                        ## "Code"
01 000f 4c696e654e756d6265725461626c65                   ## "LineNumberTable"

01 0004 6d61696e                                        ## "main"
01 0016 285b4c6a6176612f6c616e672f537472696e673b2956     ## "([Ljava/lang/String;)V"
01 0010 4d6574686f64506172616d6574657273                 ## "MethodParameters"
01 0004 61726773                                        ## "args"
01 000a 536f7572636546696c65                            ## "SourceFile"
01 000e 537472756374757265652e6a617661                   ## "Structure.java"

---17---
0c      # 12，NameAndType，名字及类型描述符
0007    # u2 index，7，字段或方法名字面量编号     ## <init>
0008    # u2 index，8，字段或方法结构编号       ## ()V

---18---
07 0019 # 25    ## java/lang/System

---19---
0c
001a 001b   # 26:27    ## out:Ljava/io/PrintStream;

---20---
01 000b 68656c6c6f20776f726c64    ## "hello world"

---21--
07 001c # 28    ## java/io/PrintStream
```

```
66.
67. ---22--
68. 0c
69. 001d 001e   # 29:30                              ## println:(Ljava/lang/Strin
    g;)V
70.
71. ---23-31---
72. 01 0011 636f6d2f636c732f537472756374757265          ## "com/cls/Structure"
73. 01 0010 6a6176612f6c616e672f4f626a656374             ## "java/lang/Object "
74. 01 0010 6a6176 612f 6c61 6e67 2f53 7973 7465 6d     ## "java/lang/System"
75. 01 0003 6f7574                                      ## "out"
76. 01 0015 4c6a6176612f696f2f5072696e7453747265616d3b  ## "Ljava/io/PrintStream;
    "
77. 01 0013 6a6176612f696f2f5072696e7453747265616d       ## "java/io/PrintStream"

78. 01 0007 7072696e746c6e                              ## "println"
79. 01 0015 284c6a6176612f6c616e672f537472696e673b2956   ## "(Ljava/lang/String;)
    V"
80.
81. 0021                #                4.                u2                ,
    access_flags                          ## ACC_PUBLIC | ACC_SUPER
82. 0005 # 5. u2, this_class,5                           ## --5.Class_info--> co
    m/cls/Structure
83. 0006 # 6. u2, super_class, 6                         ## --6.Class_info--> ja
    va/lang/Object
84. 0000 # 7. u2, interface_count, 0
85. 0000 # 8. u2, fields_count, 0
86.
87. 0002 # 9. methods count, 2
88.         # 方法一
89. 0001    # u2, access_flags, ACC_PUBLIC
90. 0007    # u2, name_index, 7                         ## <init>
91. 0008    # u2, descriptor_index, 8                    ## ()V

92. 0001    # u2, attribute_count, 1
93. 0009        # u2, attribute_name_index, 9           ## Code 属性
94. 0000 001d   # u4, attribute_length, 30
95. 0001        # u2, max_stack, 1
96. 0001        # u2, max_locals, 1
97. 0000 0005  # u4, code_array_length, 5
98. 2a              # u1, aload_0                       ## 将第 0 个 slot 中的变
    量 this 入栈
99. b7   0001        # u1, invokespecial                 ## 执行从 Object 继承
    的 <init>
```

```
100. b1                # u1, return                          ## 返回 void
101. 0000          # u2, exception_table_length, 0          ## exception table 为空，无异常
102. 0001          # u2, attributes_count, 1                ## Code 属性本身的子属性
103. 000a             # 10                                  ## LineNumberTable 属性
104. 0000 0006        # 6
105. 0001             # u2, line_number_table_length, 1
106. 0000                 # u2, start_pc, 0
107. 0003                 # u2, line_number, 3
108.         # 方法二
109. 0009     # access_flags                                 ## ACC_PIBLIC | ACC_STATIC
110. 000b     # name_index, 11                              ## main
111. 000c     # descriptor_index, 12                        ## ([Ljava/lang/String;)V
112. 0002     # attribute_count, 2
113. 0009         # attribute_name_index, 9                 ## Code
114. 0000 0025  # attribute_length, 37
115. 0002         # max_stack, 2
116. 0001         # max_locals, 1
117. 0000 0009  # code_array_length, 9
118. b2   0002        # getstatic, 2                         ## Field: java/lang/System.out:Ljava/io/PrintStream; // 加载静态对象变量
119. 12   03          # ldc, 3                               ## String: "hello world"  // 将常量参数入栈
120. b6   0004        # invokevirtual, 4                     ## Method: java/io/PrintStream.println:(Ljava/lang/String;)V // 执行方法
121. b1               # return
122. 0000         # exception_table_length, 0
123. 0001         # attributes_count, 1
124. 000a         # 10                                     ## LineNumberTable
125. 0000 000a    # 10
126. 0002             # line_number_table_length, 2
127. 0000 0005            # 0 -> 5
128. 0008 0006            # 8 -> 6
```

6.2 字节码指令

JVM 面向操作数栈（operand stack）设计了指令集，每个指令由 1 字节的操作码（opcode）表示，其后跟随 0 个或多个操作数（operand），指令集列表参考 Java bytecode instruction listings

- 大部分与数据类型相关的指令，其操作码符号都会带类型前缀，如 i 前缀表示操作 int，剩余对应关系为 b:byte, c:char, s:short, f:float, d:double, l:long, a:reference
- 由于指令集大小有限（256 个），故 boolean, byte, char, short 会被转为 int 运算

字节码可大致分为六类：

- 加载和存储指令：将变量从局部变量表 slot 加载到操作数栈的栈顶，反向则是存储

```
// 将 slot 0,1,2,3,N 加载到栈顶，T 表示类型简记前缀，可取 i,l,f,d,a
Tload_0, Tload_1, Tload_2, Tload_3, Tload n
// 将栈顶数据写回指定的 slot
Tstore_0, Tstore_1, Tstore_2, Tstore_3, Tstore n
// 将不同范围的常量值加载到栈顶，由于 0~5 常量过于常用，有单独对应的指令，ldc 则加载普通常量
bipush, sipush, Tconst_[0,1,2,3,4,5], aconst_null, ldc
```

- 运算指令

```
Tadd, Tsub, Tmul, Tdiv, Trem      // 算术运算：加减乘除，取余
Tneg, Tor, Tand, Txor             // 位运算：取反、或、与、异或
dcmpg, dcmpl, fcmpg, fcmpl, lcmp // 比较运算：后缀 g 即 greater, l 即 less than
iinc                              // 局部自增运算，与 iload 搭配使用
```

- 强制类型转换指令：窄化转换为 T 类型（长度为 N）时，会直接丢弃除了低 N 位外的其他位，可能会导致数据溢出、正负号不确定，浮点数转整型则会丢失精度

```
i2b // int -> byte
i2c, i2s; l2i, f2i, d2i; d2l, f2l; d2f
```

- 对象创建与访问指令：类实例、数组都是对象，存储结构不同，创建和访问指令有所区别

```
new                                    // 创建类实例
newarray, annewarray, multianewarry    // 创建基本类型数组、引用类型数组、多维引用类型数组
getfield, putfield; getstatic, putstatic // 读写类实例字段；读写类静态字段
Taload, Tastore; arraylength           // 读写数组元素；计算数组长度
instanceof; checkcast                  // 校验对象是否为类实例；执行强制转换
```

- 操作数栈管理指令

```
pop, pop2       // 弹出栈顶 1，2 元素
dup, dup2; swap // 复制栈顶 1，2 个元素并重新入栈；交换栈顶两个元素
```

- 控制转移指令：判断条件成立，则跳转到指定的指令行（修改 PC 指向）

```
if_<icmpeq,icmpne;icmplt,icmple;icmpgt,icmpge;acmpe,acmpne> // 整型比较，引用相等性判断
```

```
2. if<eq,lt,le,gt,ge,null,nonnull>                    // 搭配其他类型的比较运算指令使用
```

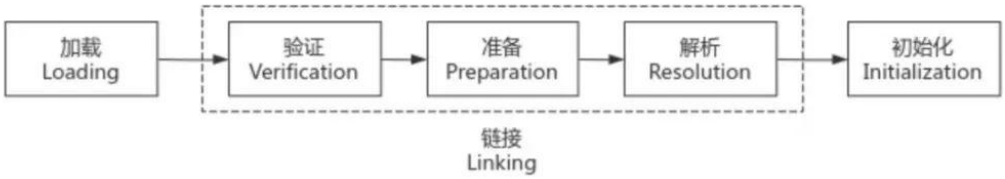
- 方法调用与返回指令

```
1. invokevirtual    // 根据对象的实际类型进行分派，调用对应的方法（比如继承后方法重写）
2. invokespecial    // 调用特殊方法，如 <cint>()V, <init>()V 等初始化方法、私有方法、父类方法
3. invokestatic     // 调用类的静态方法
4. invokeinterface // 调用接口方法（实现接口的类对象，但被声明为接口类型，调用方法）
5. invokedynamic    // TODO
6. Treturn, return // 返回指定类型，返回 void
```

- 异常处理指令：athrow 抛出异常，异常处理则由 exception_table 描述
- 同步指令：synchronized 对象锁由 monitorenter, monitorexit 搭配对象的 monitor 锁共同实现

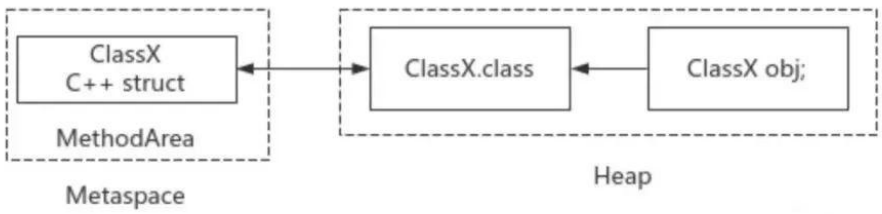
ch07. 类加载

7.1 类加载过程



「1. 加载」

原理：委托 ClassLoader 读取 Class 二进制字节流，载入到方法区内存，并在堆内存中生成对应的 java.lang.Class 对象相互引用



「2. 验证」

校验字节流确保符合 Class 文件格式，执行语义分析确保符合 Java 语法，校验字节码指令合法性

「3. 准备」

在堆中分配类变量（static）内存并初始化为零值，主义还没到执行 putstatic 指令赋值的初始化阶段，但静态常量属性除外：

```
1.  public class ClassX {
2.      final static int n = 2;           // 常量的值在编译期就已知，准备阶段完成赋值，值存储在 ConstantValue
3.      final static String str = "str"; // 字符串静态常量同理
4.  }
5.
6.  static final java.lang.String str;
7.    descriptor: Ljava/lang/String;
8.    flags: ACC_STATIC, ACC_FINAL
9.    ConstantValue: String str
```

「4. 解析」

将常量池中的符号引用（Class_info, Fieldref_info, Methodref_info）替换为直接引用（内存地址）

「5. 初始化」

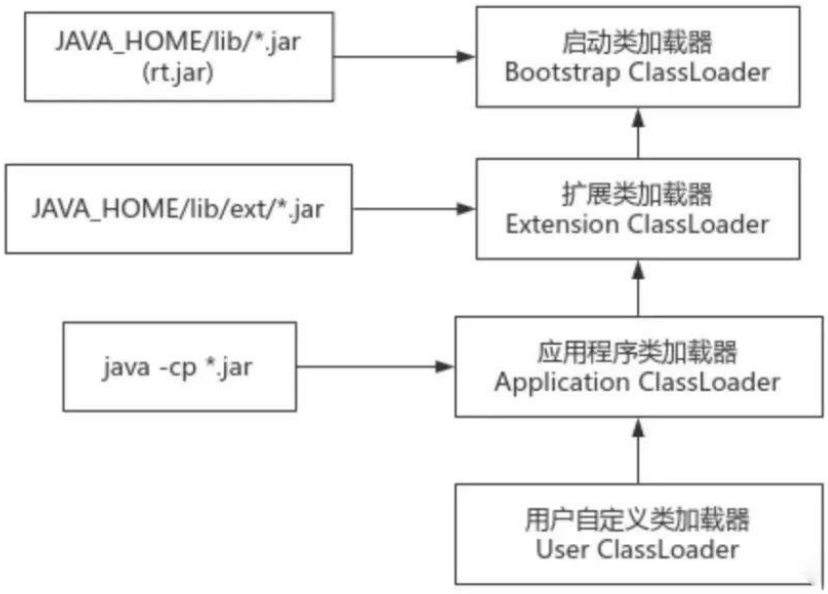
javac 会从上到下合并类中 static 变量赋值、static 语句块，生成类构造器()V，在初始化阶段执行，此方法的执行由 JVM 保证线程安全；注意 JVM 规定有且仅有的，会立即触发对类初始化的六种 case

```
1.  public class ClassX {
2.      static {
3.          println("main class ClassX init"); // 1. main() 所在的主类，总是先被初始化
4.      }
5.
6.      public static void main(String[] args) throws Exception {
7.          // 首次会触发类的初始化
8.                          //   SubX   b   =   new   SubX();      //   new   对象 // 2. new, getsatic, putstatic, invokestatic 指令
9.          // println(SuperX.a);    // 读写类的 static 变量，或调用 static 方法
10.         // println(SubX.c);      // 3. 子类初始化，会触发父类初始化
11.         // println(SubX.a);       //     子类访问父类的静态变量，只会触发父类初始化
```

```
12.
13.         // 不会触发类的初始化
14.         // println(SubX.b);      // 1. 访问类的静态常量(基本类型、字符串字面量)
15.         // println(SubX.class);  // 2. 访问类对象
16.         // println(new SubX[2]); // 3. 创建类的数组
17.     }
18. }
19.
20. class SuperX {
21.     static int a = 0;
22.     static {
23.         println("class SuperX initiated");
24.     }
25. }
26.
27. class SubX extends SuperX {
28.     final static double b = 0.1;
29.     static boolean c = false;
30.     static {
31.         println("class SubX initiated");
32.     }
33. }
```

7.2 类加载器

层级关系



双亲委派机制

- 原理：一个类加载器收到加载某个类的请求时，会先委派上层的父类加载器去加载，逐层向上，当父类加载器逐层向下反馈都无法加载此类后，该类加载器才会尝试自己加载；此模型保证了，诸如 rt.jar 中的 java.lang.Object 类，不论在底层哪种类加载器中都一定是被 Bootstrap 类加载器加载， JVM 中仅此一份，保证了一致性

- 实现

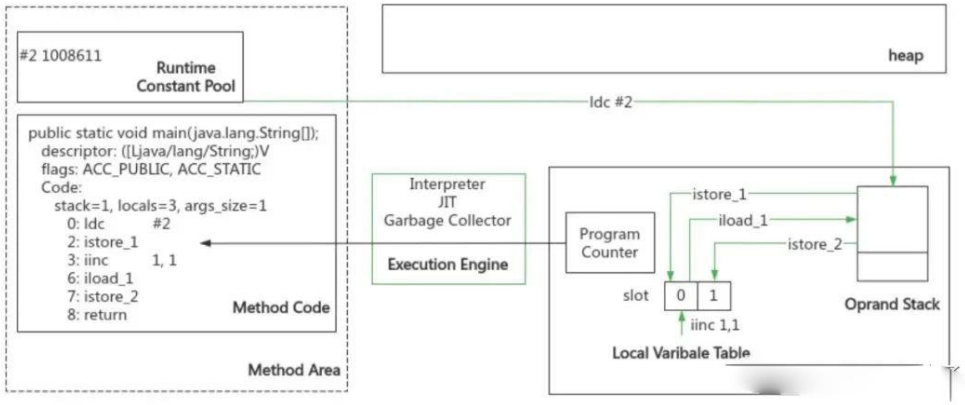
```
1.  // java/lang/ClassLoader
2.  protected Class<?> loadClass(String name, boolean resolve) throws ClassNotFo
    undException {
3.          synchronized (getClassLoadingLock(name)) {
4.              // 1. 先检查自己的加载器是否已加载此类
5.              Class<?> c = findLoadedClass(name);
6.              if (c == null) {
7.                  long t0 = System.nanoTime();
8.                  try {
9.                      if (parent != null) {
10.                         // 2. 还有上层则委派给上层去加载
11.                         c = parent.loadClass(name, false);
12.                     } else {
13.                         // 3. 如果没有上级，则委派给 Bootstrap 加载
14.                         c = findBootstrapClassOrNull(name);
15.                     }
16.                 } catch (ClassNotFoundException e) {
17.                     // 类不存在
18.                 }
19.
20.                 if (c == null) {
21.                     // 4. 到自己的 classpath 中查找类，用户自定义 ClassLoader 自
    定义了查找规则
22.                     long t1 = System.nanoTime();
23.                     c = findClass(name);
24.                 }
25.             }
26.             if (resolve) {
27.                 resolveClass(c);
28.             }
29.             return c;
30.         }
31.     }
```

ch08. 字节码执行引擎

8.1 运行时栈帧结构

```
public static void main(String[] args) {
    int a = 1008611;
    int b = ++a;
}
```

对应运行时栈帧结构：



- 局部变量表：大小在编译期确定，用于存放实参和局部变量，以大小为 32 bit 的变量槽为最小单位
  - long, double 类型被切分为 2 个 slot 同时读写（单线程操作，无线程安全问题）
  - 类对象调用方法时，slot 0 固定为当前对象的引用，即 this 隐式实参
  - 变量槽有重用优化，当 pc 指令超出某个槽中的变量的作用域时，该槽会被其他变量重用

```
1.  public static void main(String[] args) {
2.      {
3.          byte[] buf = new byte[10 * 1024 * 1024];
4.      }
5.      System.gc();  // buf 还在局部变量表的 slot 0 中，作为 GC Root 无法被回收
6.      // int v = 0; // 变量 v 重用 slot 0，gc 生效
7.      // System.gc();
```

- 操作数栈：最大深度在编译期确定，与局部变量表配合入栈、执行指令、出栈来执行字节码指令

- 返回地址：遇到 return 族指令则正常调用完成，发生异常但异常表中没有对应的 handler 则异常调用完成；正常退出到上层方法后，若有返回值则压入栈，并将程序计数器恢复到方法调用指令的下一条指令

8.2 方法调用

「1. 虚方法、非虚方法」非虚方法：编译期可知（程序运行前就唯一确定）、且运行期不可变的方法，在类加载阶段就会将方法的符号引用解析为直接引用。有 5 种：

- 静态方法（与类唯一关联）：invokestatic 调用
- 私有方法（外部不可访问）、构造器方法、父类方法：invokespecial 调用
- final 方法（无法被继承）：由 invokevirtual 调用（历史原因）

```
1.  public class StaticResolution {
2.   public static void doFunc() {
3.    System.out.println("do func...");
4.   }
5.   public static void main(String[] args) {
6.    StaticResolution.doFunc();
7.   }
8.  }
9.
10. stack=0, locals=1, args_size=1     // 静态方法的调用版本，在编译时就以常量的形式，存入字节码的参数
11.     0: invokestatic  #5           // Method doFunc:()V
12.        3: return
```

虚方法：需在运行时动态确定直接引用的方法，由 invokevirtual, invokeinterface 调用

「2. 静态分派、动态分派」背景：方法可被重载（参数类型不同，或数量不同）、可被重写（子类继承后覆盖）

分派：对象可声明为类、父类、实现的接口等类型，当对象作为实参或调用方法时，需根据其静态类型或实际类型，才能确定要调用的方法的版本，进而确定其直接引用。此过程即方法的分派

reference 变量的 2 种类型

- 静态类型：变量被声明的类型，不会改变，编译期可知
- 实际类型：变量指向的对象可被替换，运行时随时可能修改

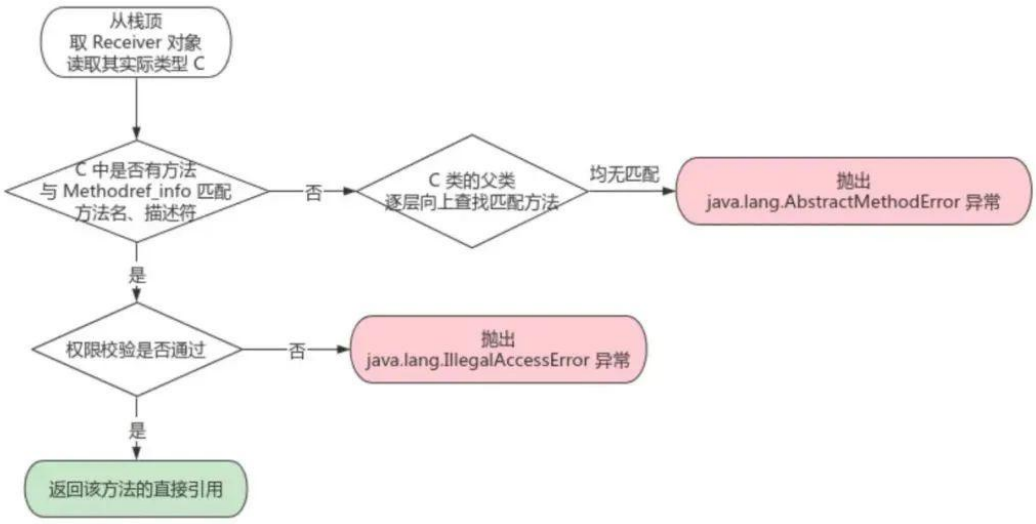
「静态分派」

- 原理：方法重载时，依赖参数的静态类型，来确定要使用哪个重载版本的方法
- 特点：发生在编译阶段，由 javac 确定类型“匹配度最高的”重载版本，来作为 invokevirtual 的参数

```
1.  public class StaticDispatch {
2.   static abstract class Human {}
3.   static class Man extends Human {}
4.   static class Woman extends Human { }
5.
6.   public void f(Human human) {System.out.println("f(Human)");}
7.   public void f(Man man) {System.out.println("f(Man)");}
8.   public void f(Woman woman) {System.out.println("f(Woman)");}
9.
10.  public static void main(String[] args) {
11.   Human man = new Man();     // 静态类型都是 Human
12.   Human woman = new Woman(); // 实际类型分别为 Man, Woman
13.   StaticDispatch sd = new StaticDispatch();
14.   sd.f(man);    // f(Human) // invokevirtual #13 // Method f:(Lcom/ch08/Stati
    cDispatch$Human;)V
15.   sd.f(woman); // f(Human) // 编译期就已确定重载版本，写入字节码中
16.  }
17. }
```

「动态分派」

- 原理：方法重写时，依赖 Receiver 对象的实际类型，来确定要使用哪个类版本的方法
- 特点：发生在运行时，依赖 invokevirtual 指令来确定调用方法的版本，进而实现多态，解析流程为



注：类的方法查找是高频操作，JVM 会在方法区中为类建一张虚方法表 vtable，以实现方法的快速查找

```
1.  public class DynamicDispatch {
2.      static abstract class Human {
3.          protected abstract void f();
4.      }
5.
6.      static class Man extends Human {
7.          @Override
8.          protected void f() {
9.              System.out.println("Man f()");
10.         }
11.     }
12.
13.     static class Woman extends Human {
14.         @Override
15.         protected void f() {
16.             System.out.println("Woman f()");
17.         }
18.     }
19.
20.     public static void main(String[] args) {
21.         Human man = new Man(); // 虽然静态类型都是 Human
22.         Human woman = new Woman();
23.          man.f();    // Man f()    // invokevirtual #6 // Method com/ch08/Dynam
    icDispatch$Human.f:()V
24.         woman.f(); // Woman f() // 虽然字节码指令的参数，都是静态类型方法的符号引
    用
25.         man = new Woman();
26.          man.f(); // Woman f() // 但 invokevirtual 会根据 Receiver 实际类型，在
    运行时解析到实际类的直接引用
27.     }
28. }
```

注意，类的字段读写指令 getfield, putfield 没有 invokevirtual 的动态分派机制，即子类的同名字段会直接覆盖父类的字段。示例：

```
1.  public class FieldHasNoPolymorphic {
2.      static class Father {
3.          public int money = 1;
4.          public Father() {
5.              money = 2;
6.              showMoney();
7.          }
```

```
8.          public void showMoney() { System.out.println("Father, money = " + mo
    ney); }
9.      }
10.
11.     static class Son extends Father {
12.         public int money = 3; // 子类字段在类加载的准备阶段被赋零值
13.         public Son() { // 子类构造器第一行默认隐藏调用 super()
14.             money = 4;
15.             showMoney();
16.         }
17.         public void showMoney() { System.out.println("Son, money = " + money)
    ; }
18.     }
19.
20.     public static void main(String[] args) {
21.         Father guy = new Son();
22.         System.out.println("guy, money = " + guy.money);
23.     }
24. }
25.
26. // Son, money = 0 // Father 类构造器执行，动态分派执行了 Son::showMoney()
27. // Son, money = 4 // Son 类构造器中访问最新的、自己的 money 字段
28. // guy, money = 2 // 字段的读写没有动态分派，静态类型是谁，就访问谁的字段
```

「3. 单分派、多分派」方法的 Receiver，与方法的参数，都是方法的宗量，根据一个宗量来选择目标方法称为单分派，需要多个宗量才能确定方法的叫多分派

- 静态分派机制会让编译器在编译阶段，对重载的多个方法，会选出参数匹配度最高的作为目标方法
- 动态分派机制在运行时，依赖 Receiver 实际类型，配合 invokevirtual 定位唯一的实例方法作为目标方法

综上两点，Java 是静态多分派、动态单分派的语言

注明：第 10,11 章讲 Java 的前后端编译，学习了自动装箱等常见语法糖的字节码实现，其余部分待有空搭配龙书一起学；第 12,13 章内容与《Java Concurrency In Practice》等书重合度较高，此处不再赘述

总结

学习《深入理解 JVM 3ed》，初步掌握了 JVM 内存区域的划分模型、GC 算法理论及常见回收器原理、Class 文件结构中各数据项解释、类加载流程、方法的执行与分派等五大方面的知识，收获颇丰。不过大部分都是理论，若有机会还是要研究下 openjdk 的源码实现 。