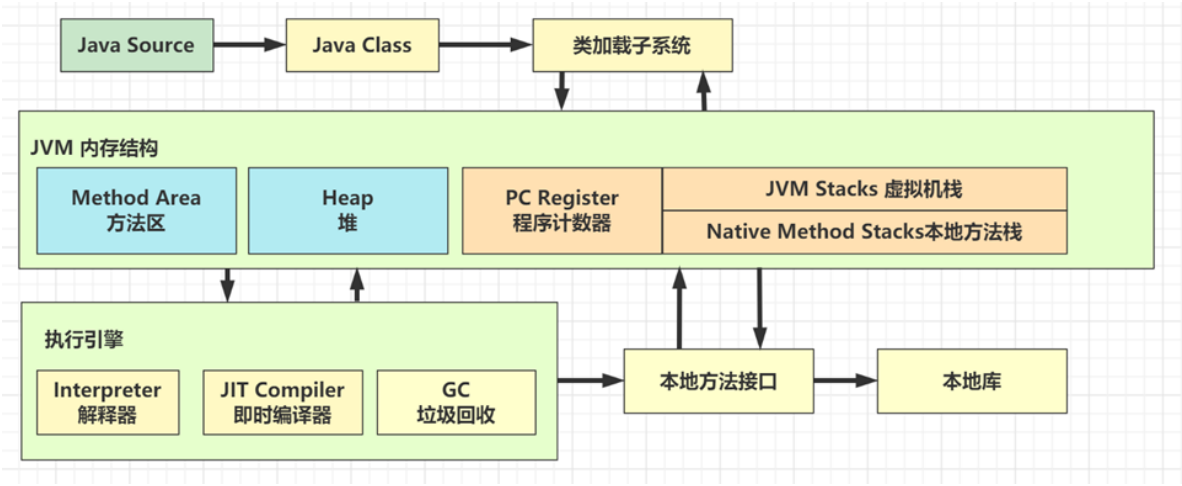
虚拟机篇

1. JVM 内存结构

要求

- 掌握 JVM 内存结构划分
- 尤其要知道方法区、永久代、元空间的关系

结合一段 java 代码的执行理解内存划分



- 执行 javac 命令编译源代码为字节码
- 执行 java 命令
  1. 创建 JVM，调用类加载子系统加载 class，将类的信息存入**方法区**
  2. 创建 main 线程，使用的内存区域是 **JVM 虚拟机栈**，开始执行 main 方法代码
  3. 如果遇到了未见过的类，会继续触发类加载过程，同样会存入**方法区**
  4. 需要创建对象，会使用**堆**内存来存储对象
  5. 不再使用的对象，会由**垃圾回收器**在内存不足时回收其内存
  6. 调用方法时，方法内的局部变量、方法参数所使用的是 **JVM 虚拟机栈**中的栈帧内存
  7. 调用方法时，先要到**方法区**获得到该方法的字节码指令，由**解释器**将字节码指令解释为机器码执行
  8. 调用方法时，会将要执行的指令行号读到**程序计数器**，这样当发生了线程切换，恢复时就可以从中断的位置继续
  9. 对于非 java 实现的方法调用，使用内存称为**本地方法栈**（见说明）
  10. 对于热点方法调用，或者频繁的循环代码，由 **JIT 即时编译器**将这些代码编译成机器码缓存，提高执行性能

说明

- 加粗字体代表了 JVM 虚拟机组件
- 对于 Oracle 的 Hotspot 虚拟机实现，不区分虚拟机栈和本地方法栈

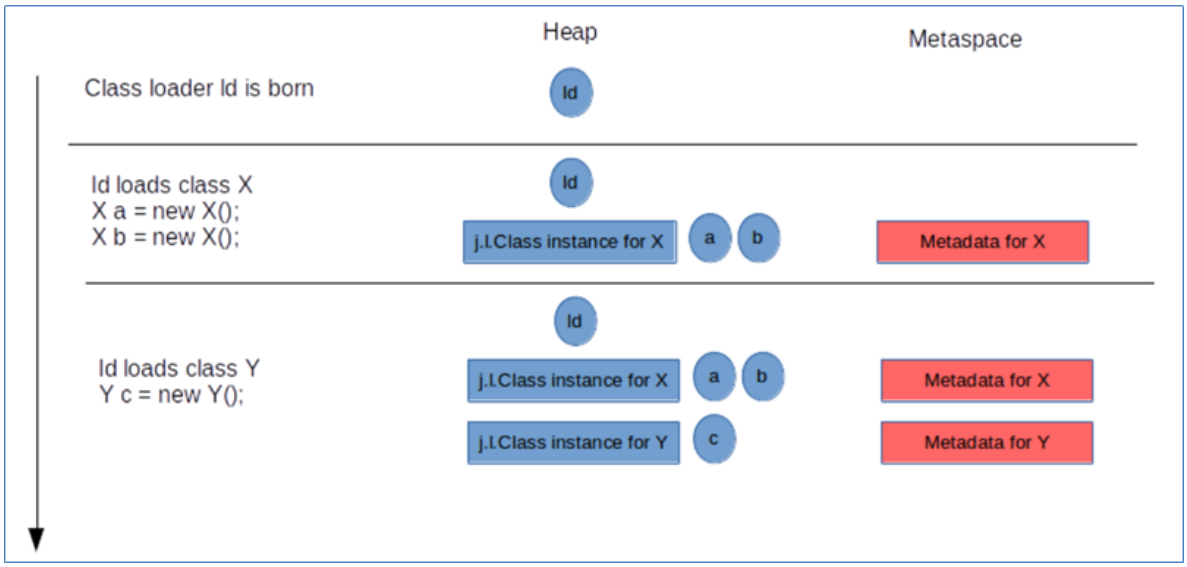
会发生内存溢出的区域

- 不会出现内存溢出的区域 – 程序计数器
- 出现 OutOfMemoryError 的情况
  - 堆内存耗尽 – 对象越来越多，又一直在使用，不能被垃圾回收
  - 方法区内存耗尽 – 加载的类越来越多，很多框架都会在运行期间动态产生新的类

    - 虚拟机栈累积 – 每个线程最多会占用 1 M 内存，线程个数越来越多，而又长时间运行不销毁时
- 出现 StackOverflowError 的区域
    - JVM 虚拟机栈，原因有方法递归调用未正确结束、反序列化 json 时循环引用

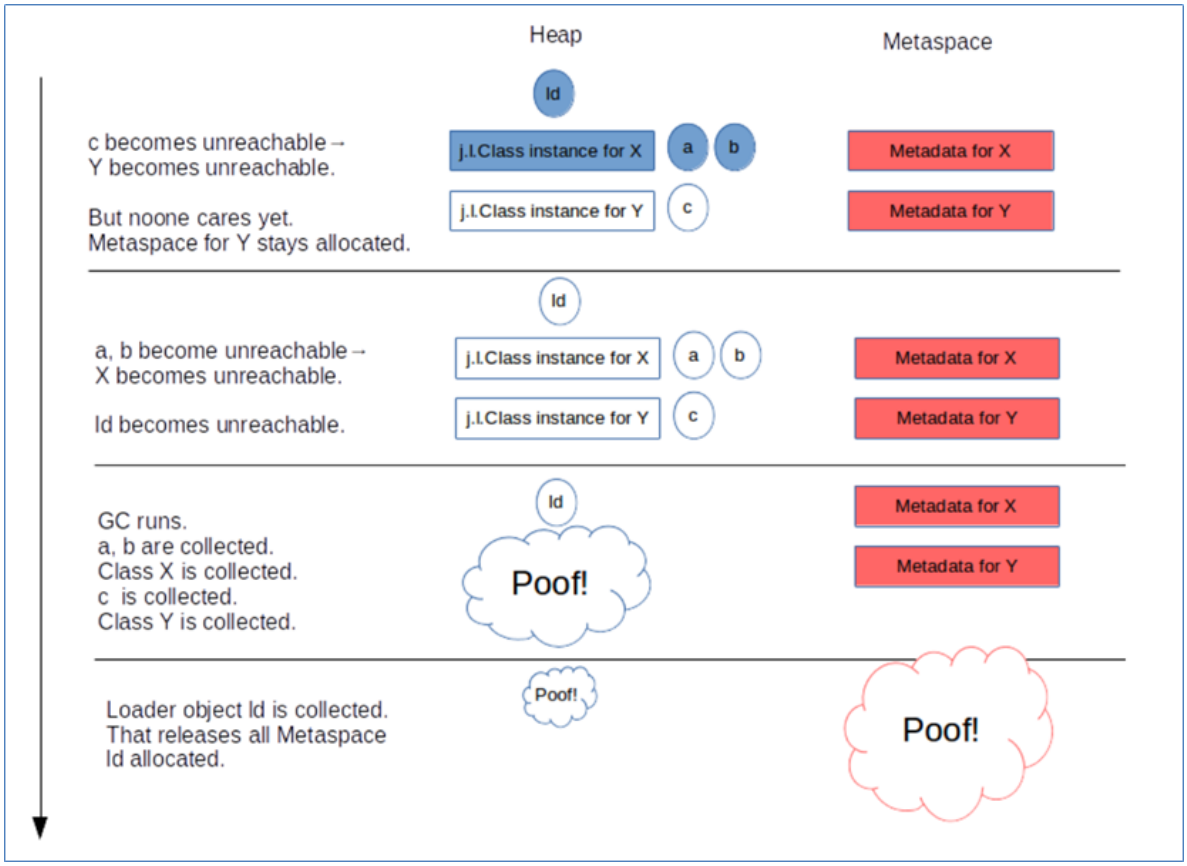
方法区、永久代、元空间

- **方法区**是 JVM 规范中定义的一块内存区域，用来存储类元数据、方法字节码、即时编译器需要的信息等
- **永久代**是 Hotspot 虚拟机对 JVM 规范的实现（1.8 之前）
- **元空间**是 Hotspot 虚拟机对 JVM 规范的另一种实现（1.8 以后），使用本地内存作为这些信息的存储空间



从这张图学到三点

- 当第一次用到某个类是，由类加载器将 class 文件的类元信息读入，并存储于元空间
- X，Y 的类元信息是存储于元空间中，无法直接访问
- 可以用 X.class，Y.class 间接访问类元信息，它们俩属于 java 对象，我们的代码中可以使用



从这张图可以学到

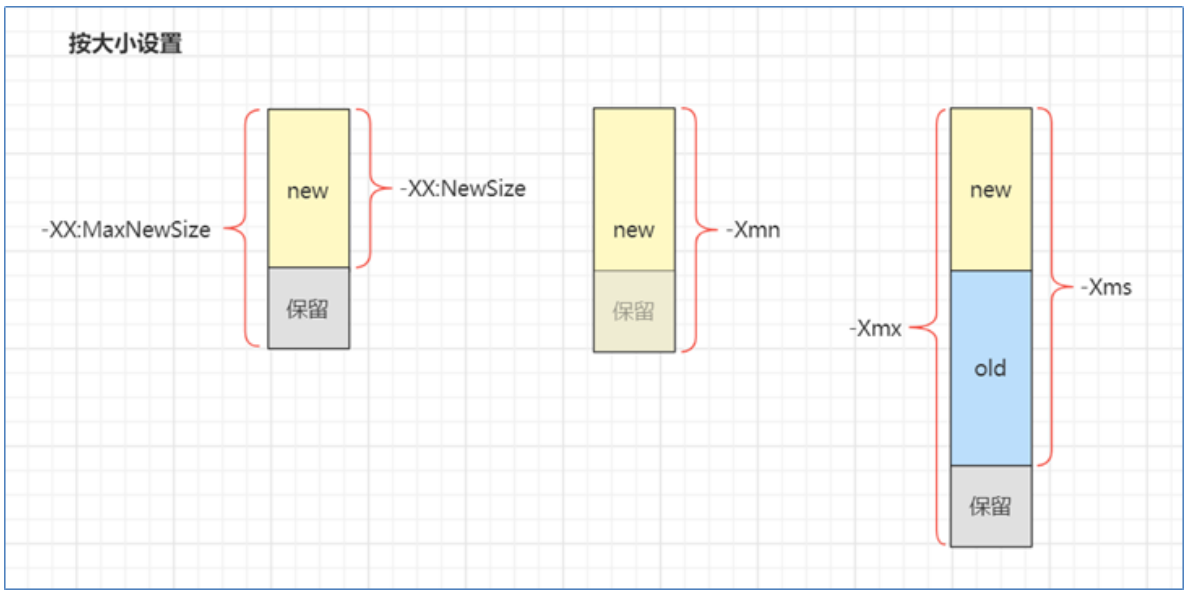
- 堆内存中：当一个**类加载器对象**，这个类加载器对象加载的所有**类对象**，这些类对象对应的所有**实例对象**都没人引用时，GC 时就会对它们占用的对内存进行释放
- 元空间中：内存释放**以类加载器为单位**，当堆中类加载器内存释放时，对应的元空间中的类元信息也会释放

2. JVM 内存参数

要求

- 熟悉常见的 JVM 参数，尤其和大小相关的

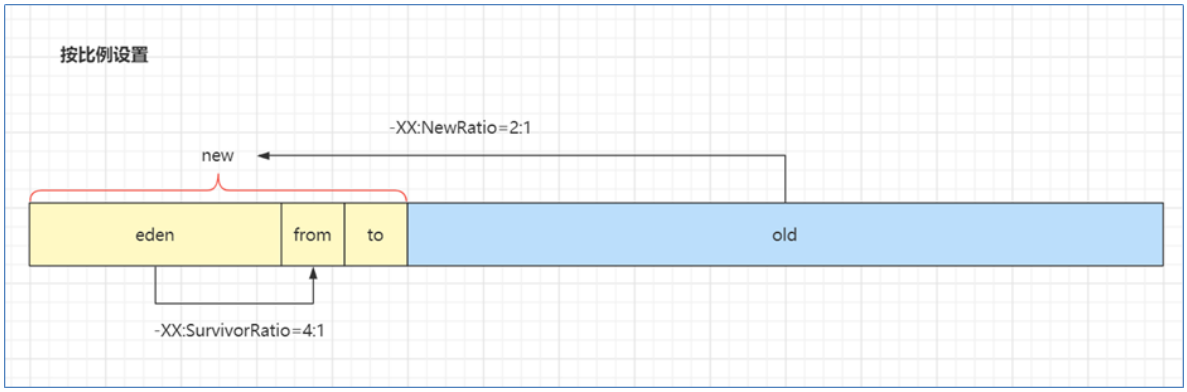
堆内存，按大小设置



解释：

- -Xms 最小堆内存（包括新生代和老年代）
- -Xmx 最大对内存（包括新生代和老年代）
- 通常建议将 -Xms 与 -Xmx 设置为大小相等，即不需要保留内存，不需要从小到大增长，这样性能较好
- -XX:NewSize 与 -XX:MaxNewSize 设置新生代的最小与最大值，但一般不建议设置，由 JVM 自己控制
- -Xmn 设置新生代大小，相当于同时设置了 -XX:NewSize 与 -XX:MaxNewSize 并且取值相等
- 保留是指，一开始不会占用那么多内存，随着使用内存越来越多，会逐步使用这部分保留内存。下同

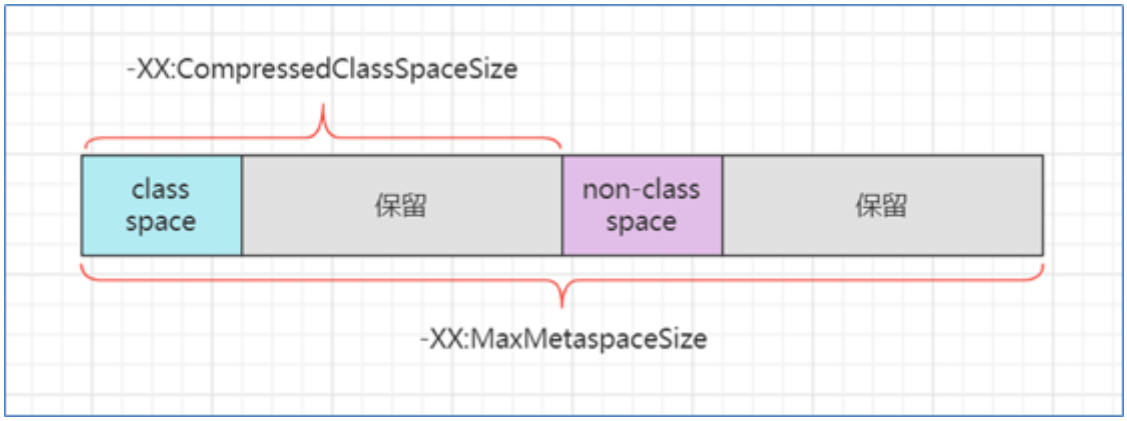
堆内存，按比例设置



解释：

- -XX:NewRatio=2:1 表示老年代占两份，新生代占一份
- -XX:SurvivorRatio=4:1 表示新生代分成六份，伊甸园占四份，from 和 to 各占一份

元空间内存设置



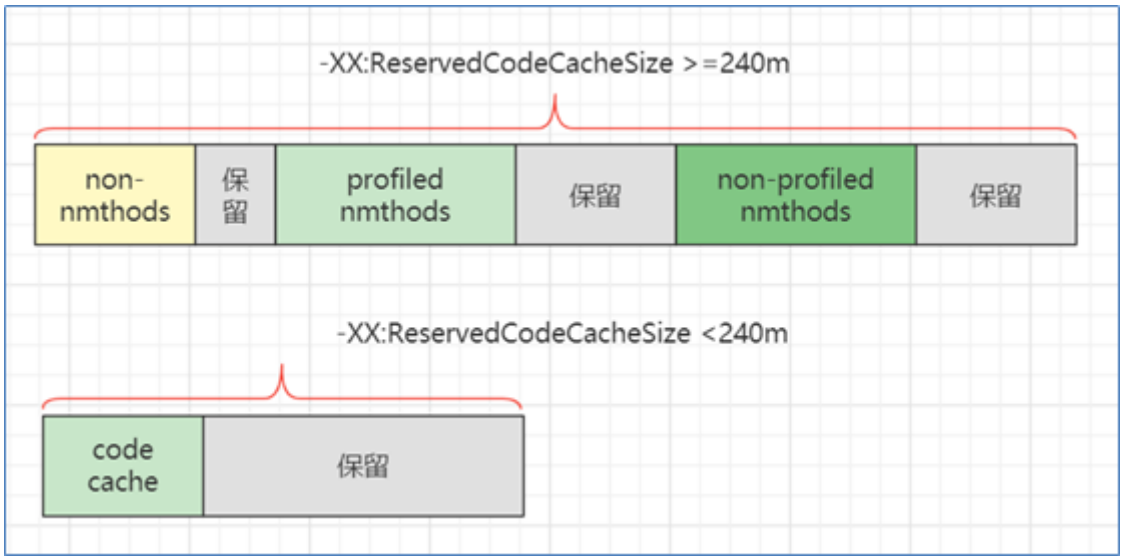
解释：

- class space 存储类的基本信息，最大值受 -XX:CompressedClassSpaceSize 控制
- non-class space 存储除类的基本信息以外的其它信息（如方法字节码、注解等）
- class space 和 non-class space 总大小受 -XX:MaxMetaspaceSize 控制

注意：

- 这里 -XX:CompressedClassSpaceSize 这段空间还与是否开启了指针压缩有关，这里暂不深入展开，可以简单认为指针压缩默认开启

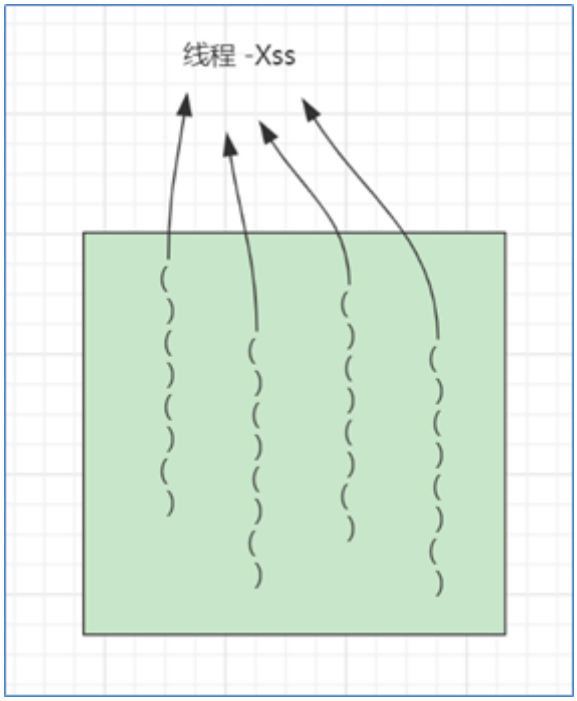
代码缓存内存设置



解释：

- 如果 -XX:ReservedCodeCacheSize < 240m，所有优化机器代码不加区分存在一起
- 否则，分成三个区域（图中笔误 mthod 拼写错误，少一个 e）
  - non-nmethods - JVM 自己用的代码
  - profiled nmethods - 部分优化的机器码
  - non-profiled nmethods - 完全优化的机器码

线程内存设置



> *官方参考文档*
>
> - https://docs.oracle.com/en/java/javase/11/tools/java.html#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE

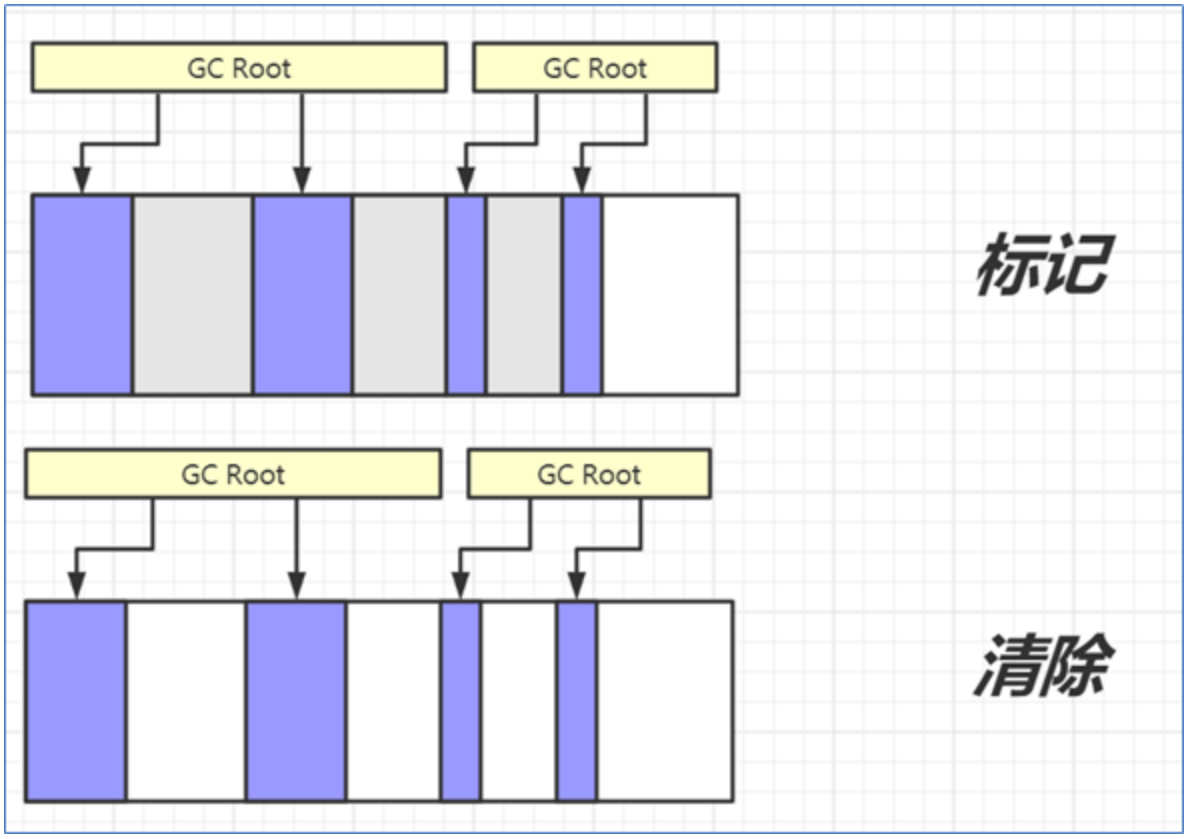
3. JVM 垃圾回收

要求

- 掌握垃圾回收算法
- 掌握分代回收思想
- 理解三色标记及漏标处理
- 了解常见垃圾回收器

三种垃圾回收算法

标记清除法



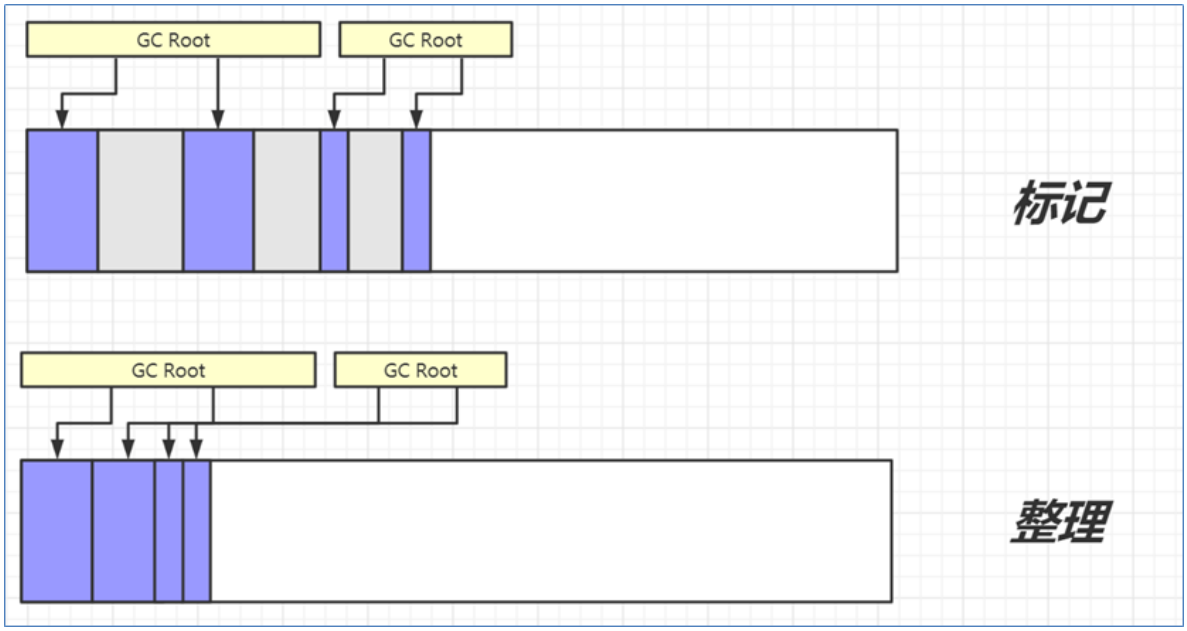
解释：

1. 找到 GC Root 对象，即那些一定不会被回收的对象，如正执行方法内局部变量引用的对象、静态变量引用的对象
2. 标记阶段：沿着 GC Root 对象的引用链找，直接或间接引用到的对象加上标记
3. 清除阶段：释放未加标记的对象占用的内存

要点：

- 标记速度与存活对象线性关系
- 清除速度与内存大小线性关系
- 缺点是会产生内存碎片

标记整理法



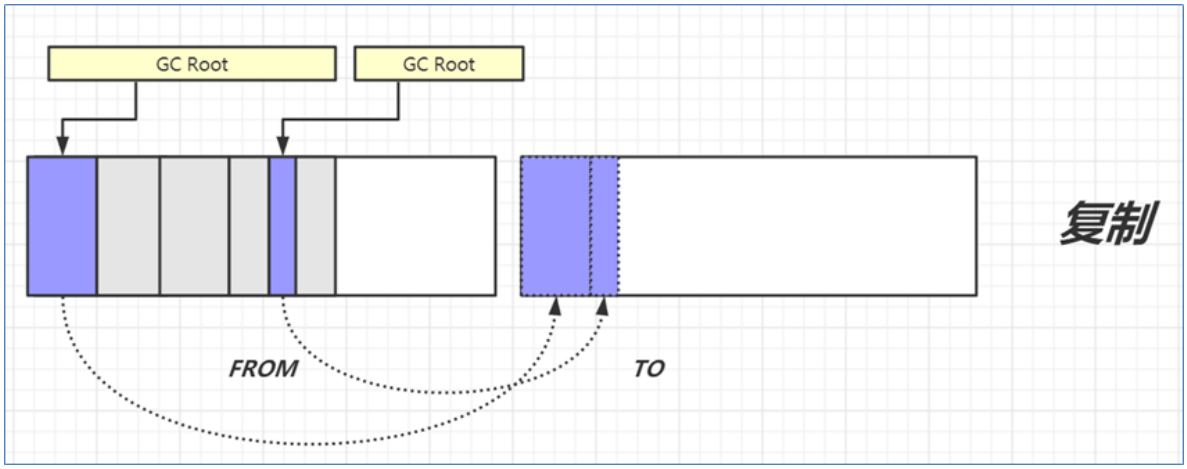
解释：

1. 前面的标记阶段、清理阶段与标记清除法类似
2. 多了一步整理的动作，将存活对象向一端移动，可以避免内存碎片产生

特点：

- 标记速度与存活对象线性关系
- 清除与整理速度与内存大小成线性关系
- 缺点是性能上较慢

标记复制法



解释：

1. 将整个内存分成两个大小相等的区域，from 和 to，其中 to 总是处于空闲，from 存储新创建的对象
2. 标记阶段与前面的算法类似
3. 在找出存活对象后，会将它们从 from 复制到 to 区域，复制的过程中自然完成了碎片整理
4. 复制完成后，交换 from 和 to 的位置即可

特点：

- 标记与复制速度与存活对象成线性关系
- 缺点是会占用成倍的空间

GC 与分代回收算法

GC 的目的在于实现无用对象内存自动释放，减少内存碎片、加快分配速度

GC 要点：

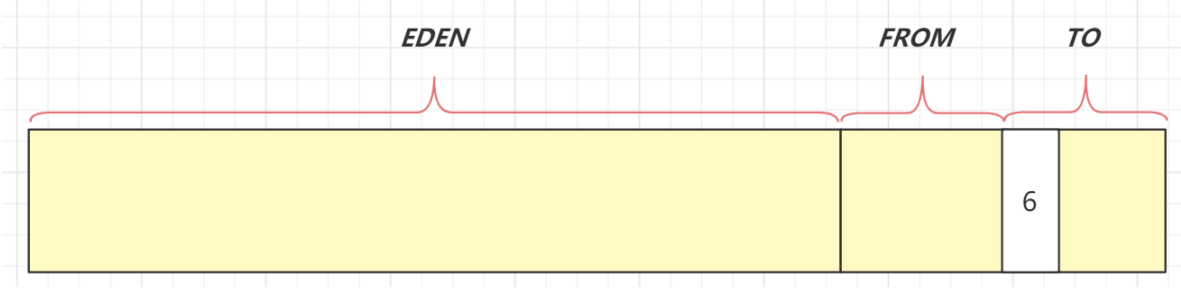
- 回收区域是**堆内存**，不包括虚拟机栈
- 判断无用对象，使用**可达性分析算法**，**三色标记法**标记存活对象，回收未标记对象
- GC 具体的实现称为**垃圾回收器**
- GC 大都采用了**分代回收思想**
  - 理论依据是大部分对象朝生夕灭，用完立刻就可以回收，另有少部分对象会长时间存活，每次很难回收
  - 根据这两类对象的特性将回收区域分为**新生代**和**老年代**，新生代采用标记复制法、老年代一般采用标记整理法
- 根据 GC 的规模可以分成 **Minor GC**，**Mixed GC**，**Full GC**

分代回收

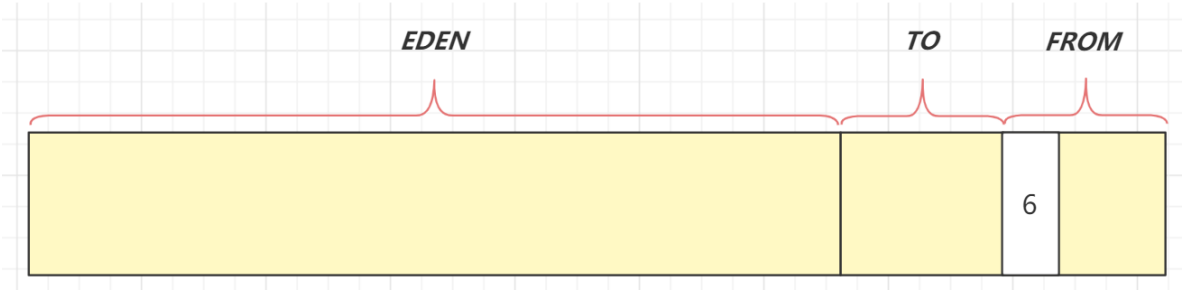
1. 伊甸园 eden，最初对象都分配到这里，与幸存区 survivor（分成 from 和 to）合称新生代，

2. 当伊甸园内存不足，标记伊甸园与 from（现阶段没有）的存活对象

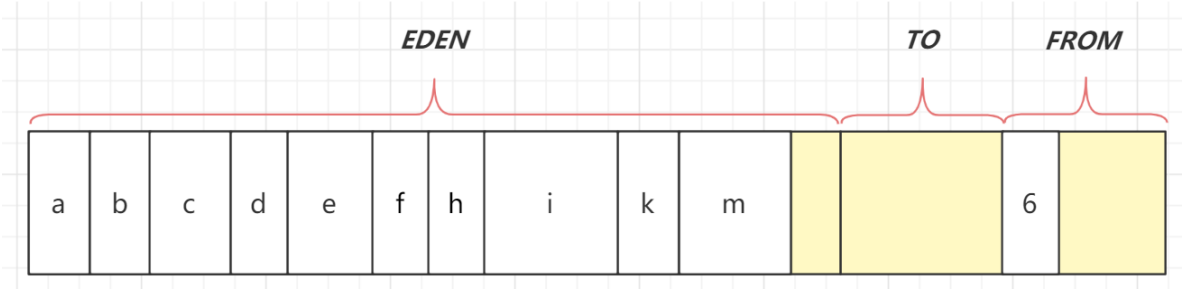
3. 将存活对象采用复制算法复制到 to 中，复制完毕后，伊甸园和 from 内存都得到释放



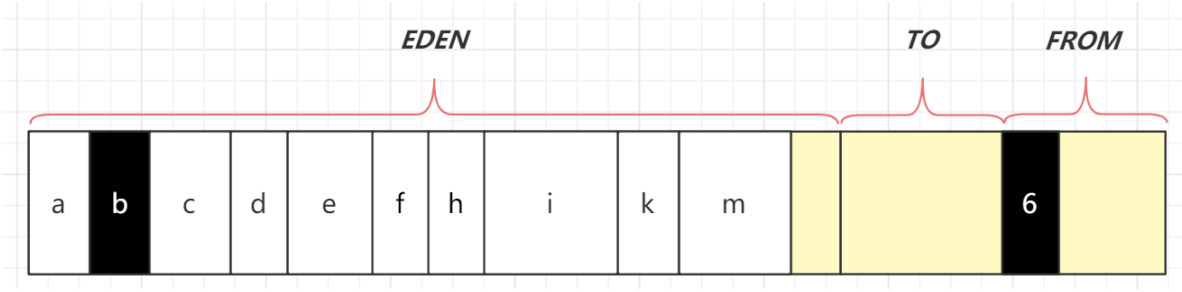
4. 将 from 和 to 交换位置



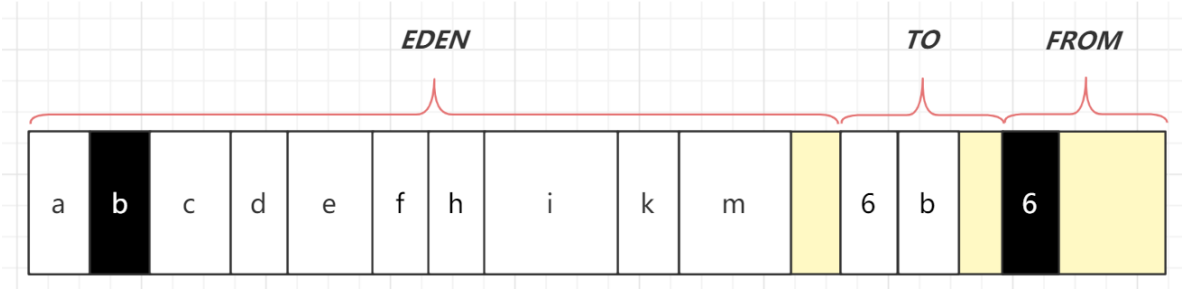
5. 经过一段时间后伊甸园的内存又出现不足



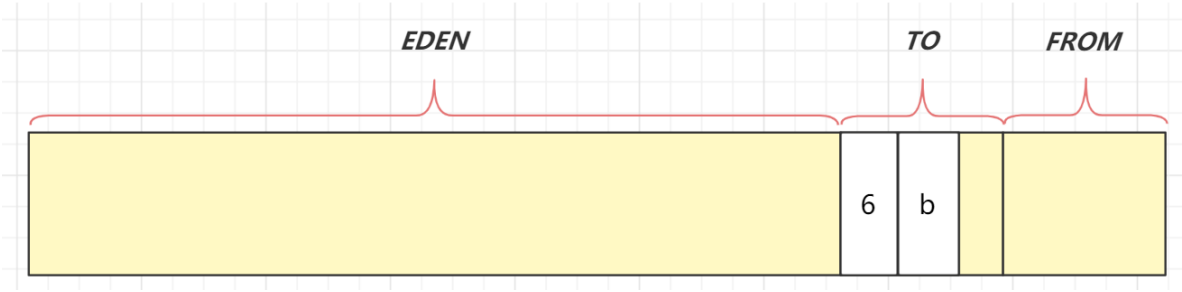
6. 标记伊甸园与 from（现阶段没有）的存活对象



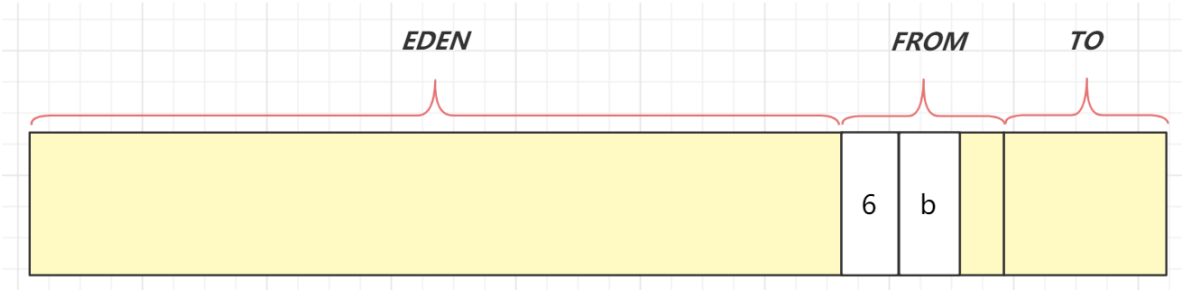
7. 将存活对象采用复制算法复制到 to 中



8. 复制完毕后，伊甸园和 from 内存都得到释放



9. 将 from 和 to 交换位置



10. 老年代 old，当幸存区对象熬过几次回收（最多15次），晋升到老年代（幸存区内存不足或大对象会导致提前晋升）

GC 规模

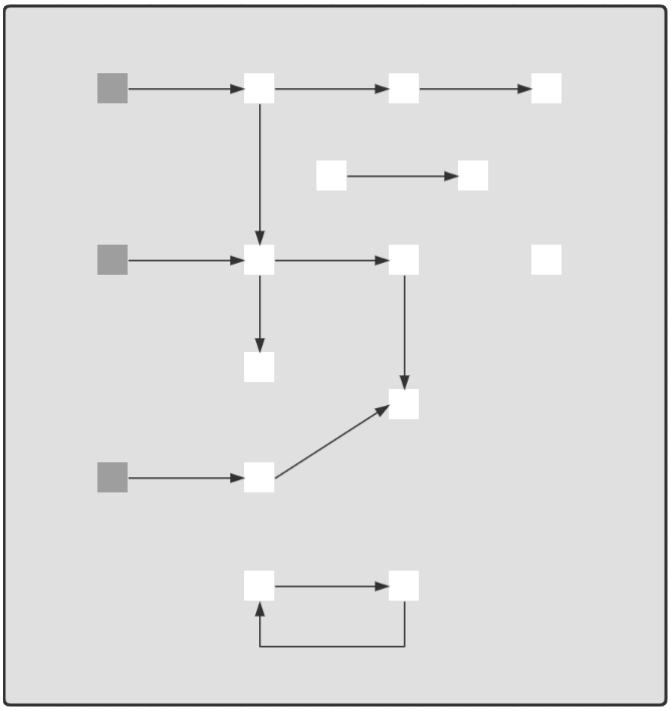
- Minor GC 发生在新生代的垃圾回收，暂停时间短
- Mixed GC 新生代 + 老年代部分区域的垃圾回收，G1 收集器特有
- Full GC 新生代 + 老年代完整垃圾回收，暂停时间长，**应尽力避免**

三色标记

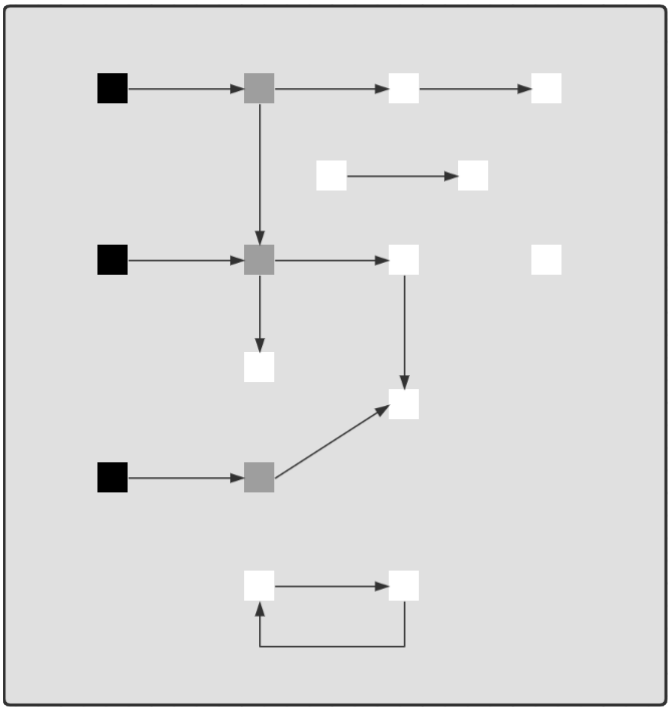
即用三种颜色记录对象的标记状态

- 黑色 – 已标记
- 灰色 – 标记中
- 白色 – 还未标记

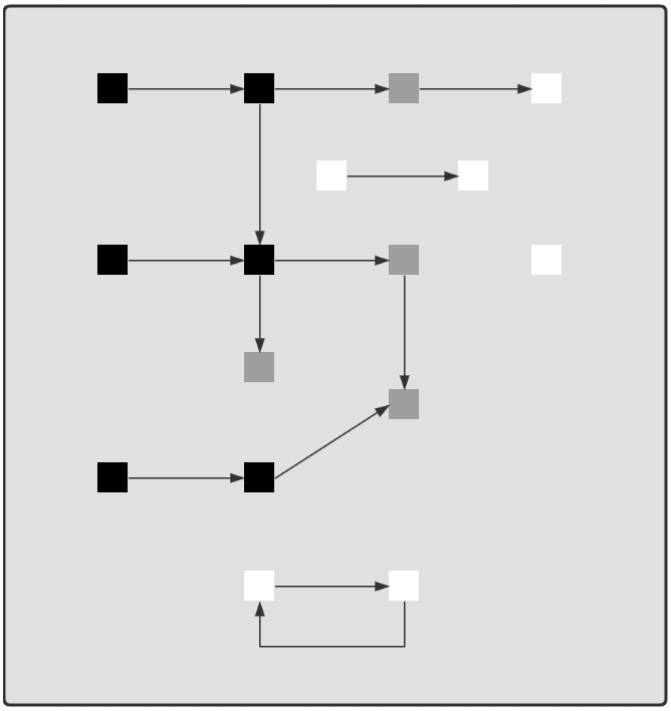
1. 起始的三个对象还未处理完成，用灰色表示



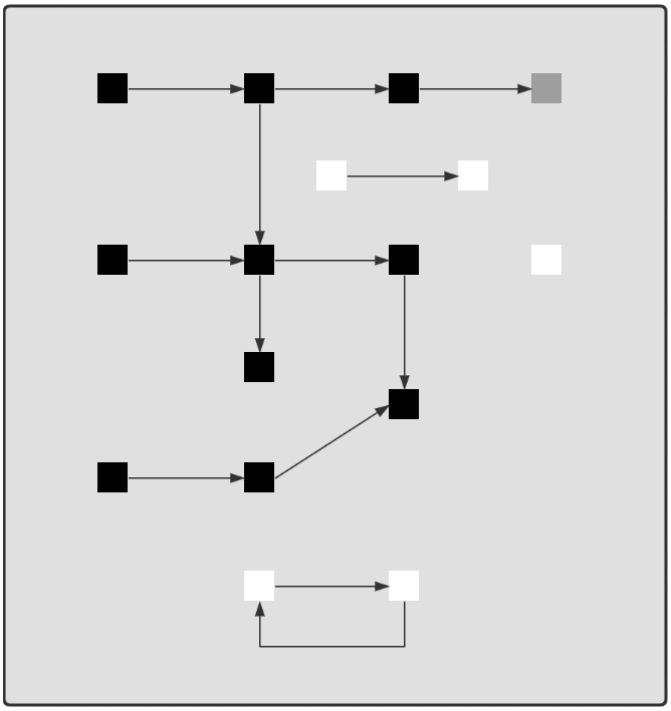
2. 该对象的引用已经处理完成，用黑色表示，黑色引用的对象变为灰色



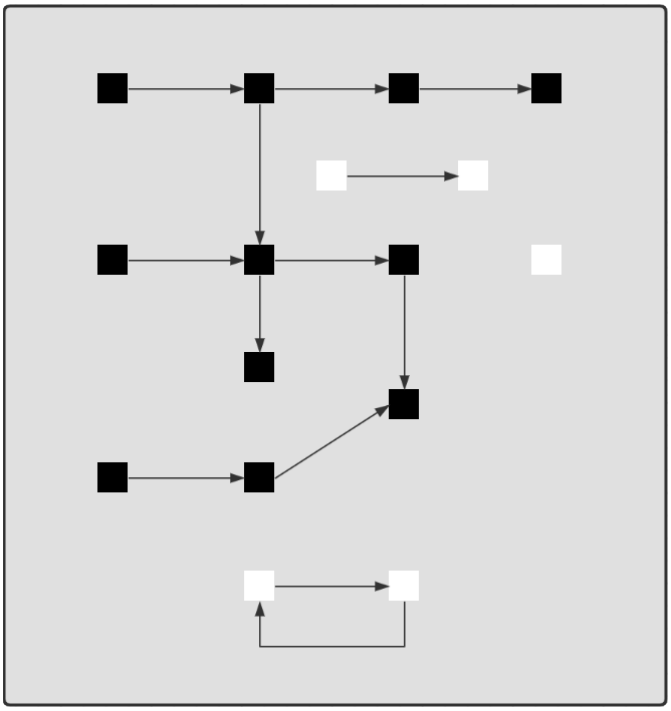
3. 依次类推



4. 沿着引用链都标记了一遍



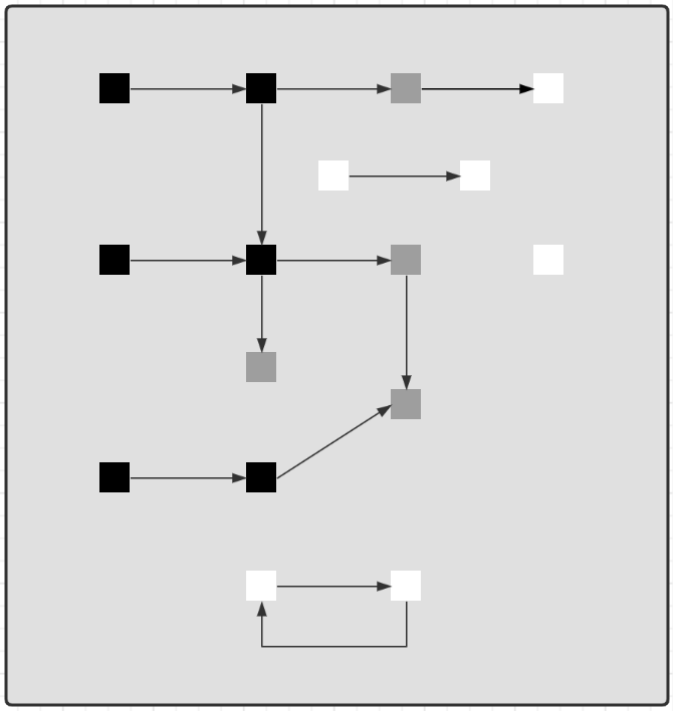
5. 最后为标记的白色对象，即为垃圾



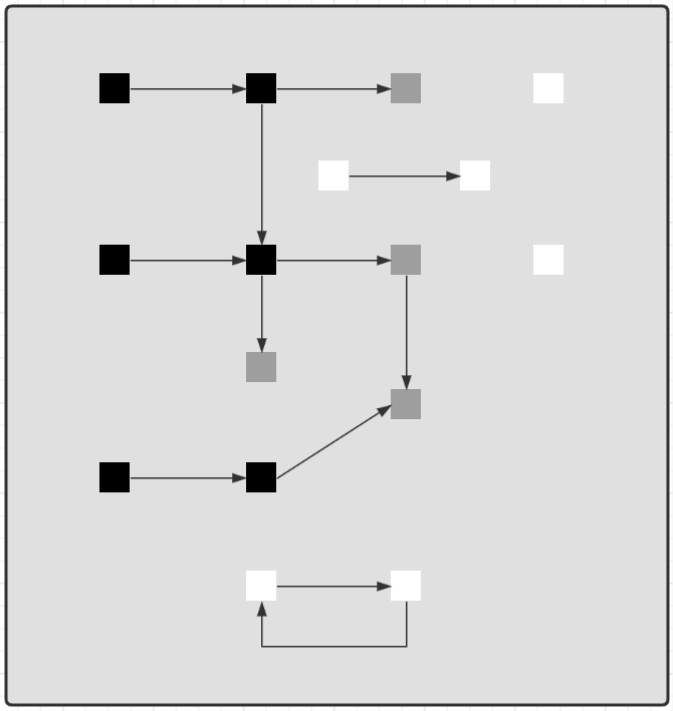
并发漏标问题

比较先进的垃圾回收器都支持**并发标记**，即在标记过程中，用户线程仍然能工作。但这样带来一个新的问题，如果用户线程修改了对象引用，那么就存在漏标问题。例如：

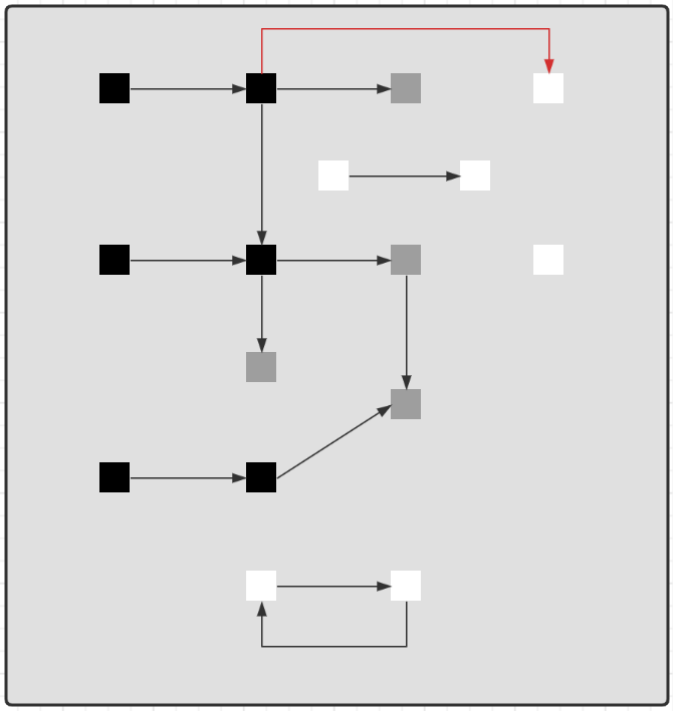
1. 如图所示标记工作尚未完成



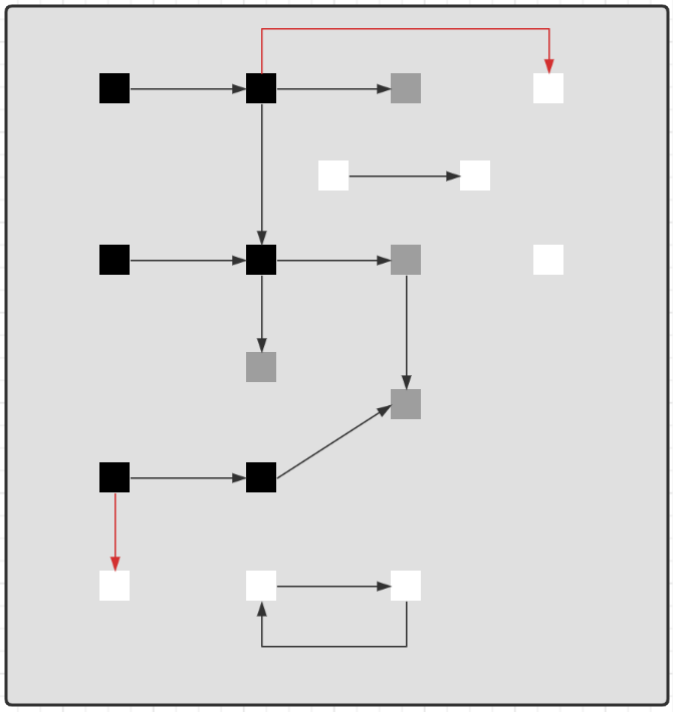
2. 用户线程同时在工作，断开了第一层 3、4 两个对象之间的引用，这时对于正在处理 3 号对象的垃圾回收线程来讲，它会将 4 号对象当做是白色垃圾



3. 但如果其他用户线程又建立了 2、4 两个对象的引用，这时因为 2 号对象是黑色已处理对象了，因此垃圾回收线程不会察觉到这个引用关系的变化，从而产生了漏标



4. 如果用户线程让黑色对象引用了一个新增对象，一样会存在漏标问题



因此对于**并发标记**而言，必须解决漏标问题，也就是要记录标记过程中的变化。有两种解决方法：

1. Incremental Update 增量更新法，CMS 垃圾回收器采用
    - 思路是拦截每次赋值动作，只要赋值发生，被赋值的对象就会被记录下来，在重新标记阶段再确认一遍
2. Snapshot At The Beginning，SATB 原始快照法，G1 垃圾回收器采用
    - 思路也是拦截每次赋值动作，不过记录的对象不同，也需要在重新标记阶段对这些对象二次处理
    - 新加对象会被记录
    - 被删除引用关系的对象也被记录

垃圾回收器 - Parallel GC

- eden 内存不足发生 Minor GC，采用标记复制算法，需要暂停用户线程
- old 内存不足发生 Full GC，采用标记整理算法，需要暂停用户线程
- **注重吞吐量**

垃圾回收器 - ConcurrentMarkSweep GC

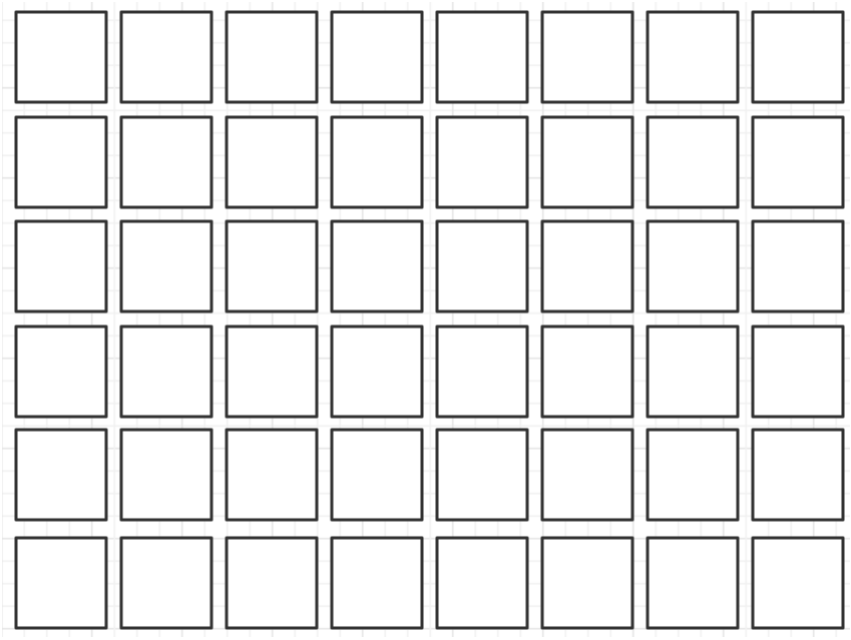
- 它是工作在 old 老年代，支持**并发标记**的一款回收器，采用**并发清除**算法
  - 并发标记时不需暂停用户线程
  - 重新标记时仍需暂停用户线程
- 如果并发失败（即回收速度赶不上创建新对象速度），会触发 Full GC
- **注重响应时间**

垃圾回收器 - G1 GC

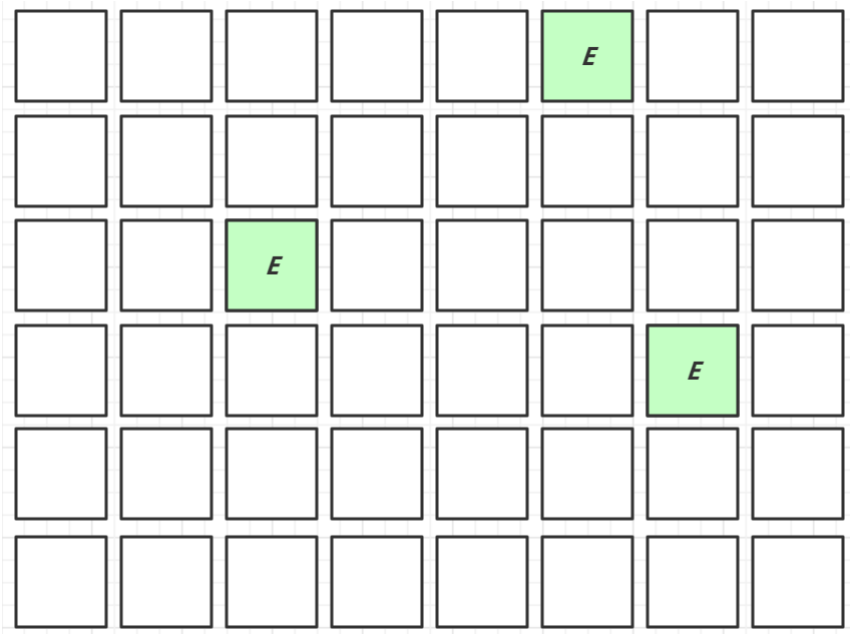
- **响应时间与吞吐量兼顾**
- 划分成多个区域，每个区域都可以充当 eden，survivor，old， humongous，其中 humongous 专为大对象准备
- 分成三个阶段：新生代回收、并发标记、混合收集
- 如果并发失败（即回收速度赶不上创建新对象速度），会触发 Full GC

G1 回收阶段 - 新生代回收

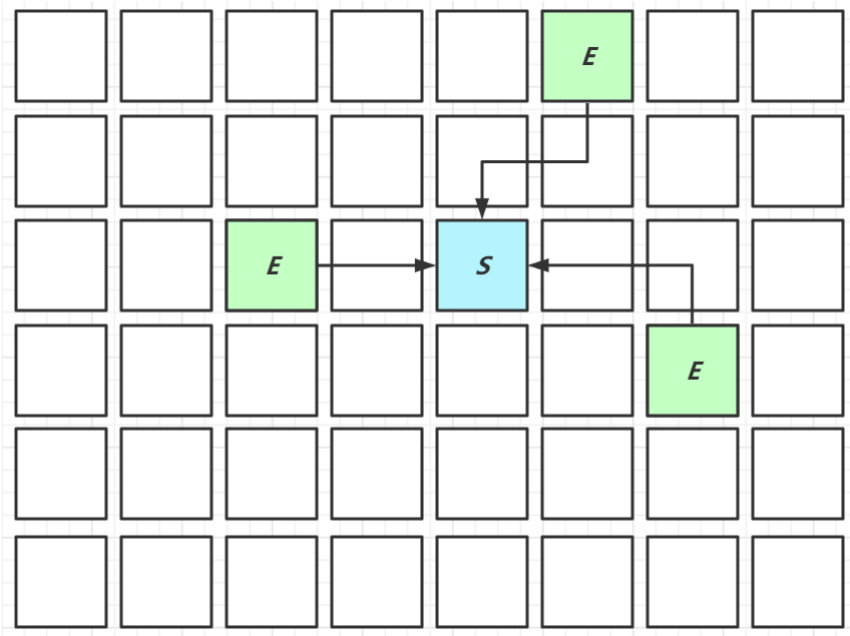
1. 初始时，所有区域都处于空闲状态



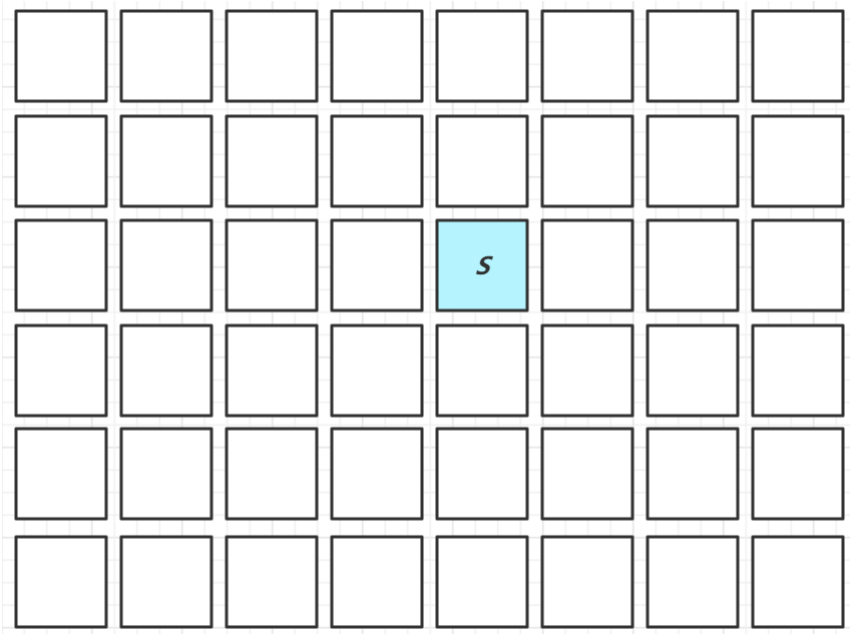
2. 创建了一些对象，挑出一些空闲区域作为伊甸园区存储这些对象



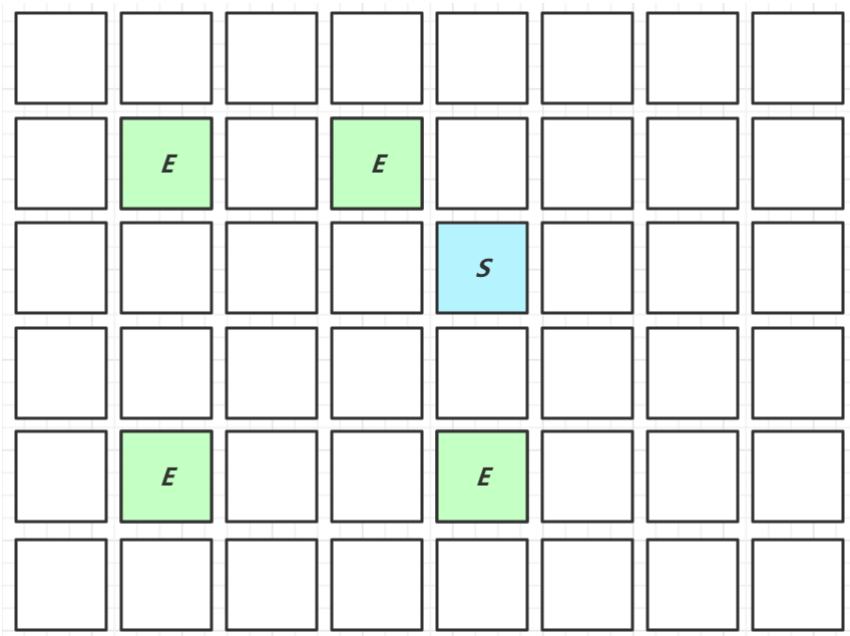
3. 当伊甸园需要垃圾回收时，挑出一个空闲区域作为幸存区，用复制算法复制存活对象，需要暂停用户线程



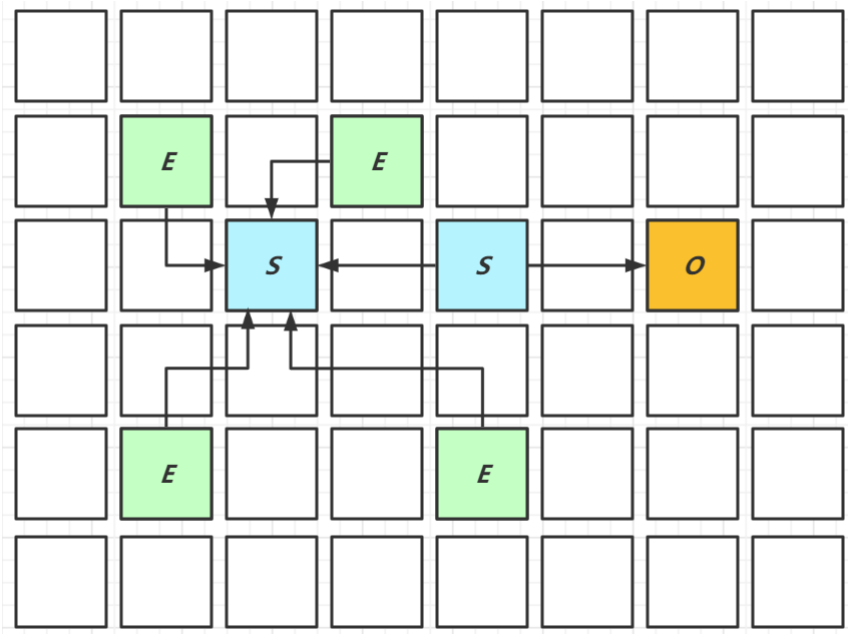
4. 复制完成，将之前的伊甸园内存释放



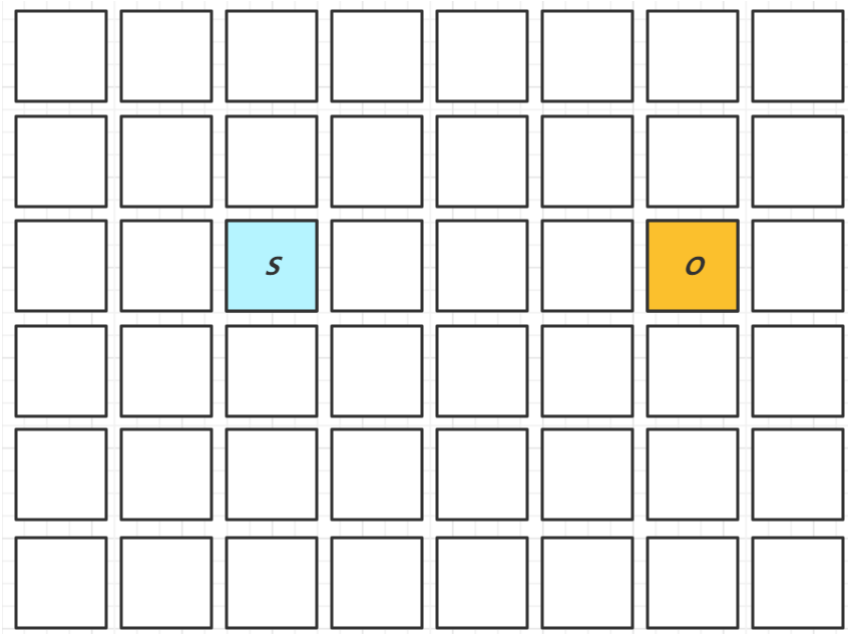
5. 随着时间流逝，伊甸园的内存又有不足



6. 将伊甸园以及之前幸存区中的存活对象，采用复制算法，复制到新的幸存区，其中较老对象晋升至老年代

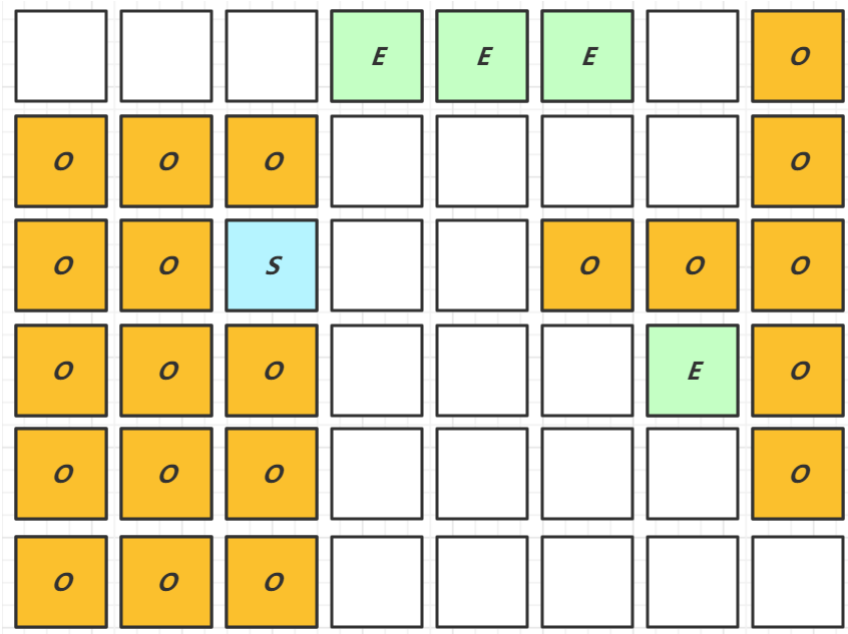


7. 释放伊甸园以及之前幸存区的内存

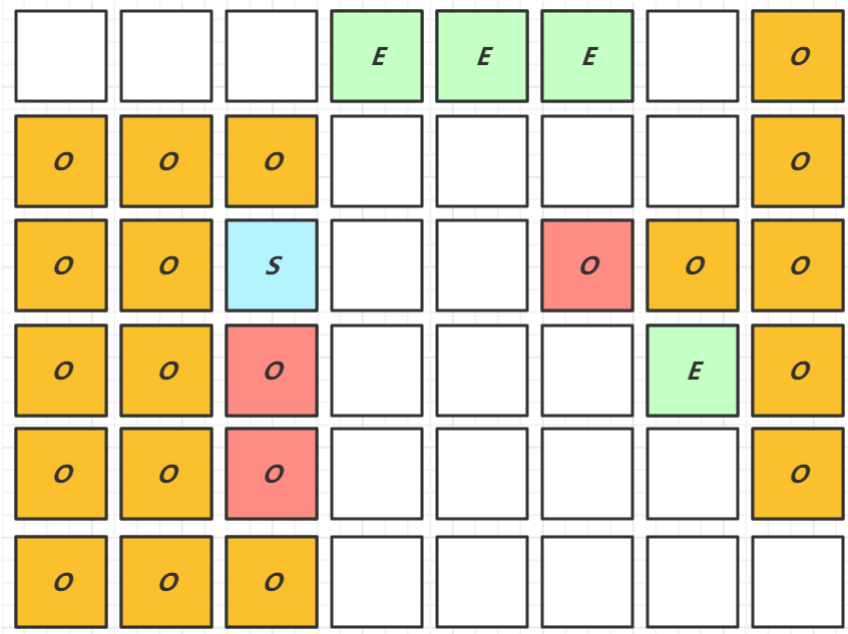


G1 回收阶段 - 并发标记与混合收集

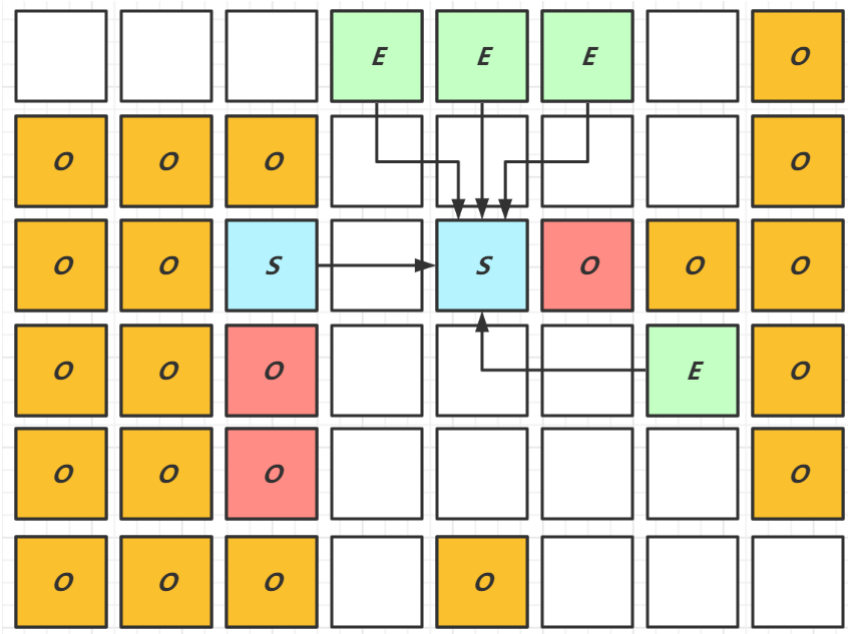
1. 当老年代占用内存超过阈值后，触发并发标记，这时无需暂停用户线程



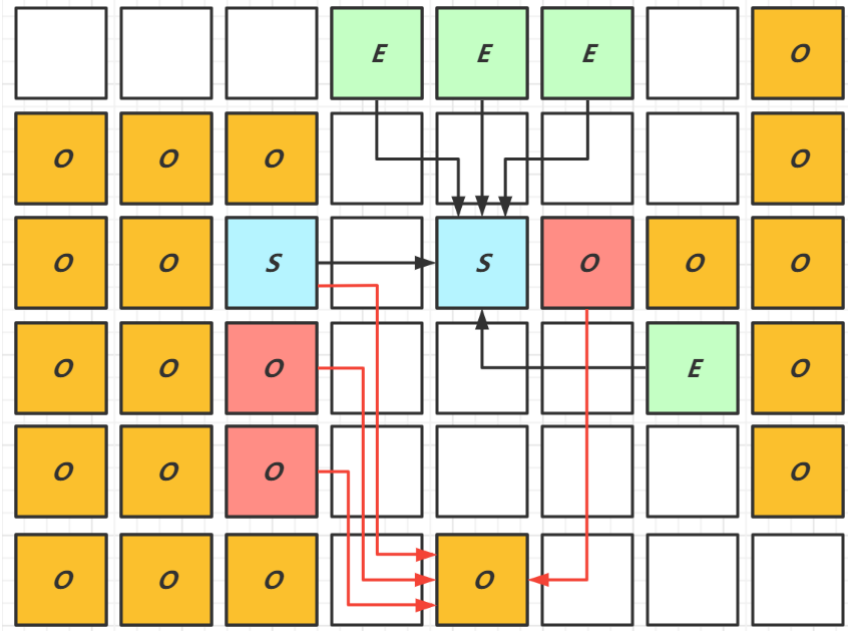
2. 并发标记之后，会有重新标记阶段解决漏标问题，此时需要暂停用户线程。这些都完成后就知道了老年代有哪些存活对象，随后进入混合收集阶段。此时不会对所有老年代区域进行回收，而是根据**暂停时间目标**优先回收价值高（存活对象少）的区域（这也是 Gabage First 名称的由来）。



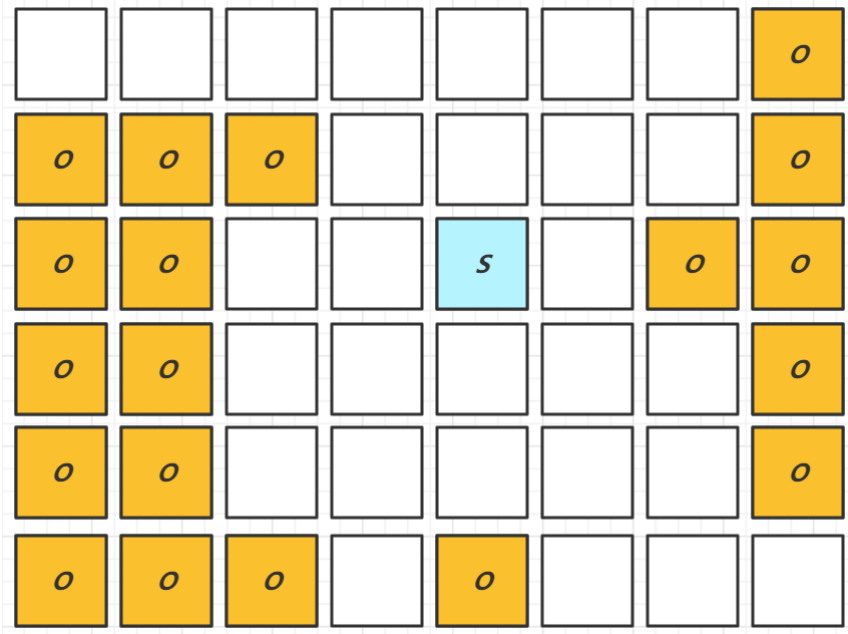
3. 混合收集阶段中，参与复制的有 eden、survivor、old，下图显示了伊甸园和幸存区的存活对象复制



4. 下图显示了老年代和幸存区晋升的存活对象的复制



5. 复制完成，内存得到释放。进入下一轮的新生代回收、并发标记、混合收集



4. 内存溢出

要求

- 能够说出几种典型的导致内存溢出的情况

典型情况

- 误用线程池导致的内存溢出
  - 参考 day03.TestOomThreadPool
- 查询数据量太大导致的内存溢出
  - 参考 day03.TestOomTooManyObject
- 动态生成类导致的内存溢出
  - 参考 day03.TestOomTooManyClass

5. 类加载

要求

- 掌握类加载阶段
- 掌握类加载器
- 理解双亲委派机制

类加载过程的三个阶段

1. 加载
   1. 将类的字节码载入方法区，并创建类.class 对象
   2. 如果此类的父类没有加载，先加载父类
   3. 加载是懒惰执行
2. 链接
   1. 验证 – 验证类是否符合 Class 规范，合法性、安全性检查
   2. 准备 – 为 static 变量分配空间，设置默认值
   3. 解析 – 将常量池的符号引用解析为直接引用
3. 初始化
   1. 静态代码块、static 修饰的变量赋值、static final 修饰的引用类型变量赋值，会被合并成一个 `<cinit>` 方法，在初始化时被调用
   2. static final 修饰的基本类型变量赋值，在链接阶段就已完成
   3. 初始化是懒惰执行

> ***验证手段***
>
> - 使用 jps 查看进程号
> - 使用 jhsdb 调试，执行命令 `jhsdb.exe hsdb` 打开它的图形界面
>   - Class Browser 可以查看当前 jvm 中加载了哪些类
>   - 控制台的 universe 命令查看堆内存范围
>   - 控制台的 g1regiondetails 命令查看 region 详情
>   - `scanoops 起始地址 结束地址 对象类型` 可以根据类型查找某个区间内的对象地址
>   - 控制台的 `inspect 地址` 指令能够查看这个地址对应的对象详情

> - 使用 javap 命令可以查看 class 字节码

> ***代码说明***
>
> - day03.loader.TestLazy - 验证类的加载是懒惰的，用到时才触发类加载
> - day03.loader.TestFinal - 验证使用 final 修饰的变量不会触发类加载

jdk 8 的类加载器

名称	加载哪的类	说明
Bootstrap ClassLoader	JAVA_HOME/jre/lib	无法直接访问
Extension ClassLoader	JAVA_HOME/jre/lib/ext	上级为 Bootstrap，显示为 null
Application ClassLoader	classpath	上级为 Extension
自定义类加载器	自定义	上级为 Application

双亲委派机制

所谓的双亲委派，就是指优先委派上级类加载器进行加载，如果上级类加载器

- 能找到这个类，由上级加载，加载后该类也对下级加载器可见
- 找不到这个类，则下级类加载器才有资格执行加载

双亲委派的目的有两点

1. 让上级类加载器中的类对下级共享（反之不行），即能让你的类能依赖到 jdk 提供的核心类
2. 让类的加载有优先次序，保证核心类优先加载

对双亲委派的误解

下面面试题的回答是错误的

能不能自己写个类叫java.lang.System?

> 答案：通常不可以，但可以采取另类方法达到这个需求。
> 解释：为了不让我们写System类，类加载采用委托机制，这样可以保证爸爸们优先，爸爸们能找到的类，儿子就没有机会加载。而System类是Bootstrap加载器加载的，就算自己重写，也总是使用Java系统提供的System，自己写的System类根本没有机会得到加载。
> 但是，我们可以自己定义一个类加载器来达到这个目的，为了避免双亲委托机制，这个类加载器也必须是特殊的。由于系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载。

错在哪了?

- 自己编写类加载器就能加载一个假冒的 java.lang.System 吗? 答案是不行。
- 假设你自己的类加载器用双亲委派，那么优先由启动类加载器加载真正的 java.lang.System，自然不会加载假冒的
- 假设你自己的类加载器不用双亲委派，那么你的类加载器加载假冒的 java.lang.System 时，它需要先加载父类 java.lang.Object，而你没有用委派，找不到 java.lang.Object 所以加载会失败
- **以上也仅仅是假设。**事实上操作你就会发现，自定义类加载器加载以 java. 打头的类时，会抛安全异常，在 jdk9 以上版本这些特殊包名都与模块进行了绑定，更连编译都过不了

> *代码说明*
>
> - day03.loader.TestJdk9ClassLoader - 演示类加载器与模块的绑定关系

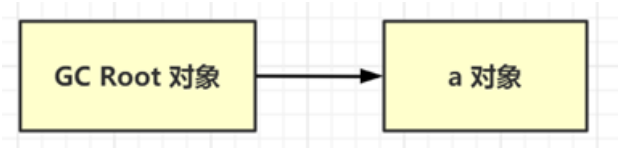
6. 四种引用

要求

- 掌握四种引用

强引用

1. 普通变量赋值即为强引用，如 A a = new A();
2. 通过 GC Root 的引用链，如果强引用不到该对象，该对象才能被回收

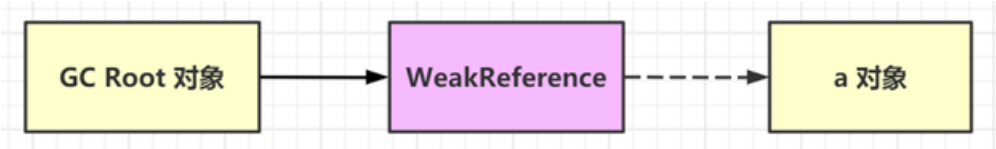


软引用（SoftReference）

1. 例如：SoftReference a = new SoftReference(new A());
2. 如果仅有软引用该对象时，首次垃圾回收不会回收该对象，如果内存仍不足，再次回收时才会释放对象
3. 软引用自身需要配合引用队列来释放
4. 典型例子是反射数据

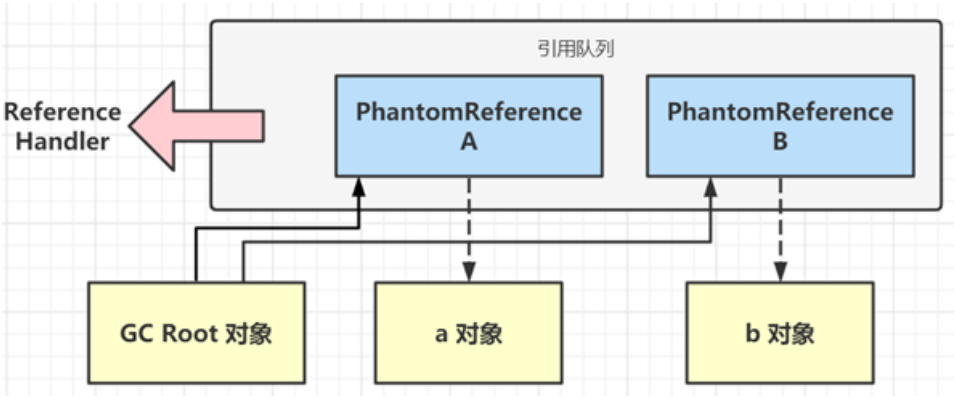
弱引用（WeakReference）

1. 例如：WeakReference a = new WeakReference(new A());
2. 如果仅有弱引用引用该对象时，只要发生垃圾回收，就会释放该对象
3. 弱引用自身需要配合引用队列来释放
4. 典型例子是 ThreadLocalMap 中的 Entry 对象



虚引用（PhantomReference）

1. 例如： PhantomReference a = new PhantomReference(new A(), referenceQueue);
2. 必须配合引用队列一起使用，当虚引用所引用的对象被回收时，由 Reference Handler 线程将虚引用对象入队，这样就可以知道哪些对象被回收，从而对它们关联的资源做进一步处理
3. 典型例子是 Cleaner 释放 DirectByteBuffer 关联的直接内存



> *代码说明*
>
> - day03.reference.TestPhantomReference - 演示虚引用的基本用法
> - day03.reference.TestWeakReference - 模拟 ThreadLocalMap, 采用引用队列释放 entry 内存

7. finalize

要求

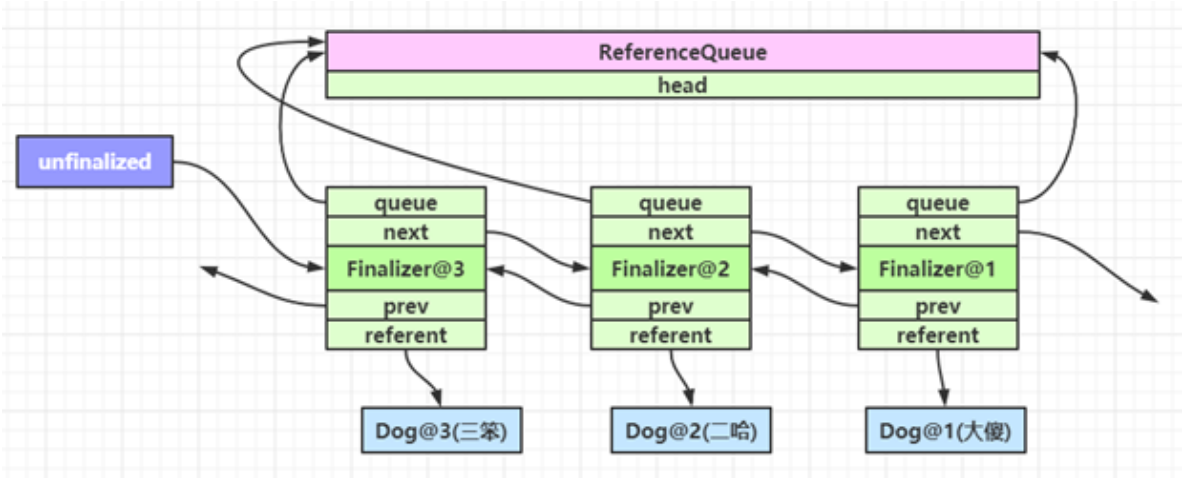
- 掌握 finalize 的工作原理与缺点

finalize

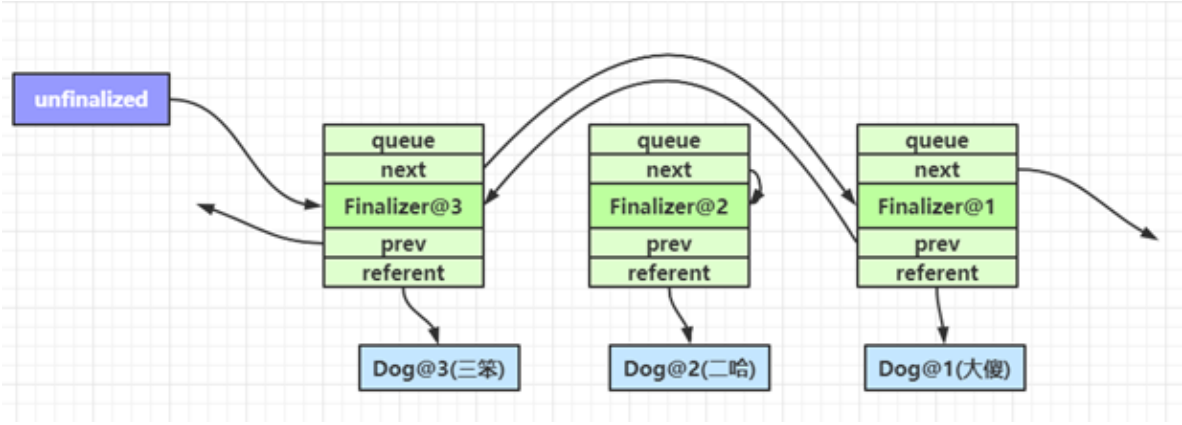
- 它是 Object 中的一个方法，如果子类重写它，垃圾回收时此方法会被调用，可以在其中进行资源释放和清理工作
- 将资源释放和清理放在 finalize 方法中非常不好，非常影响性能，严重时甚至会引起 OOM，从 Java9 开始就被标注为 @Deprecated，不建议被使用了

finalize 原理

1. 对 finalize 方法进行处理的核心逻辑位于 java.lang.ref.Finalizer 类中，它包含了名为 unfinalized 的静态变量（双向链表结构），Finalizer 也可被视为另一种引用对象（地位与软、弱、虚相当，只是不对外，无法直接使用）
2. 当重写了 finalize 方法的对象，在构造方法调用之时，JVM 都会将其包装成一个 Finalizer 对象，并加入 unfinalized 链表中



3. Finalizer 类中还有另一个重要的静态变量，即 ReferenceQueue 引用队列，刚开始它是空的。当狗对象可以被当作垃圾回收时，就会把这些狗对象对应的 Finalizer 对象加入此引用队列
4. 但此时 Dog 对象还没法被立刻回收，因为 unfinalized -> Finalizer 这一引用链还在引用它嘛，为的是【先别着急回收啊，等我调完 finalize 方法，再回收】
5. FinalizerThread 线程会从 ReferenceQueue 中逐一取出每个 Finalizer 对象，把它们从链表断开并真正调用 finallize 方法



6. 由于整个 Finalizer 对象已经从 unfinalized 链表中断开，这样没谁能引用到它和狗对象，所以下次 gc 时就被回收了

finalize 缺点

- 无法保证资源释放：FinalizerThread 是守护线程，代码很有可能没来得及执行完，线程就结束了
- 无法判断是否发生错误：执行 finalize 方法时，会吞掉任意异常（Throwable）
- 内存释放不及时：重写了 finalize 方法的对象在第一次被 gc 时，并不能及时释放它占用的内存，因为要等着 FinalizerThread 调用完 finalize，把它从 unfinalized 队列移除后，第二次 gc 时才能真正释放内存
- 有的文章提到【Finalizer 线程会和我们的主线程进行竞争，不过由于它的优先级较低，获取到的 CPU时间较少，因此它永远也赶不上主线程的步伐】这个显然是错误的，FinalizerThread 的优先级较普通线程更高，原因应该是 finalize 串行执行慢等原因综合导致

> ***代码说明***
>
> - day03.reference.TestFinalize - finalize 的测试代码