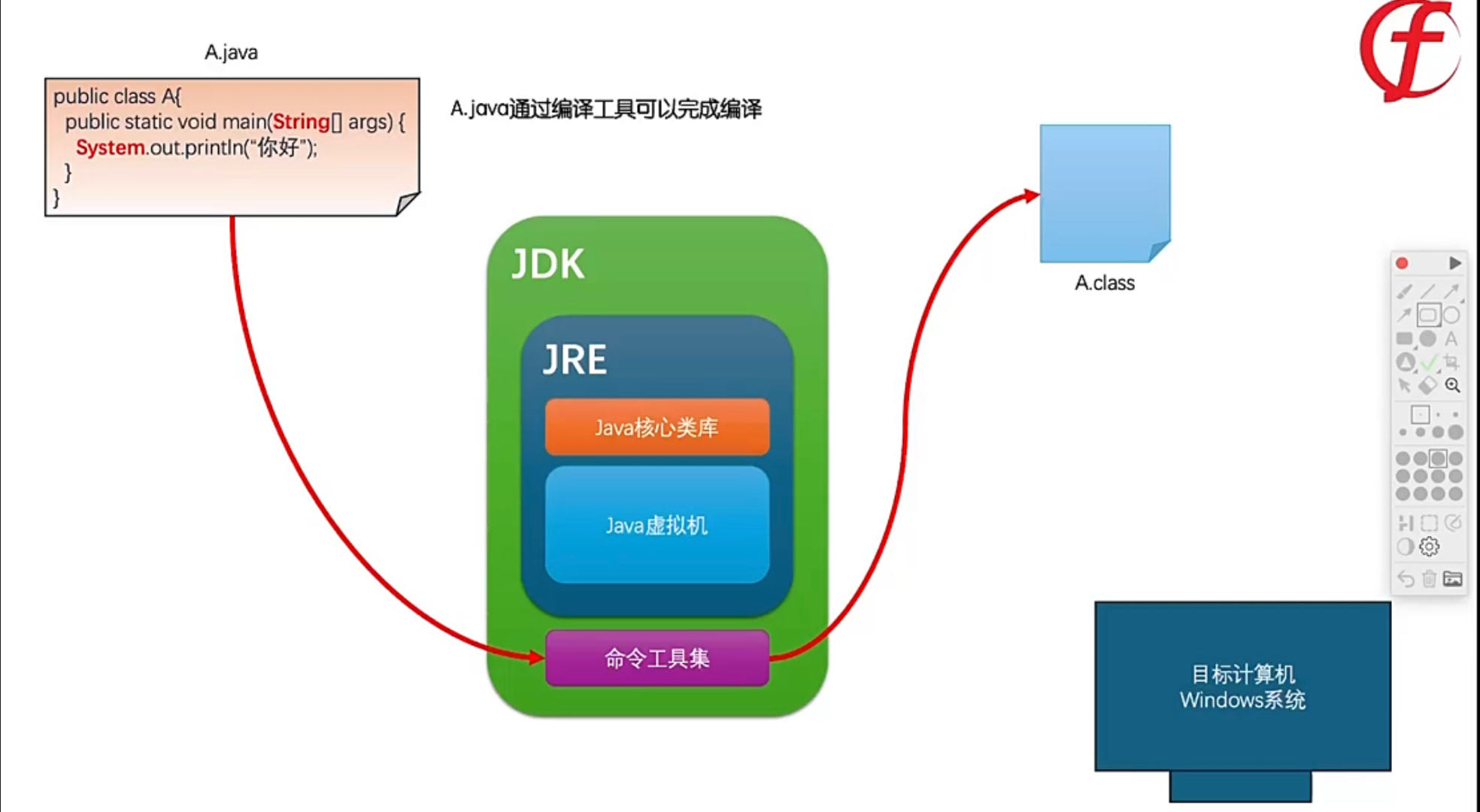
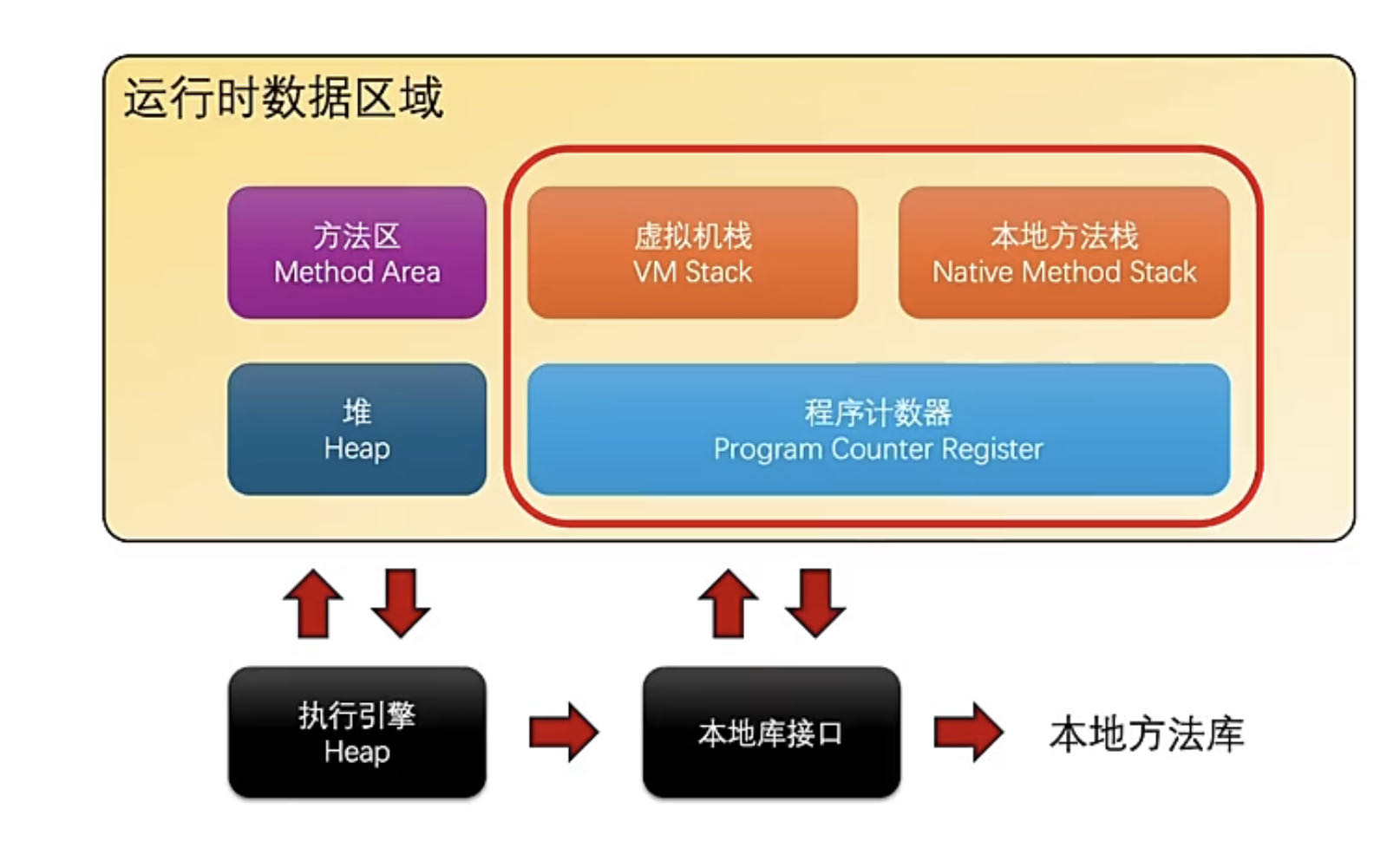
# 总复习-JVM认知

1. JDK,JRE

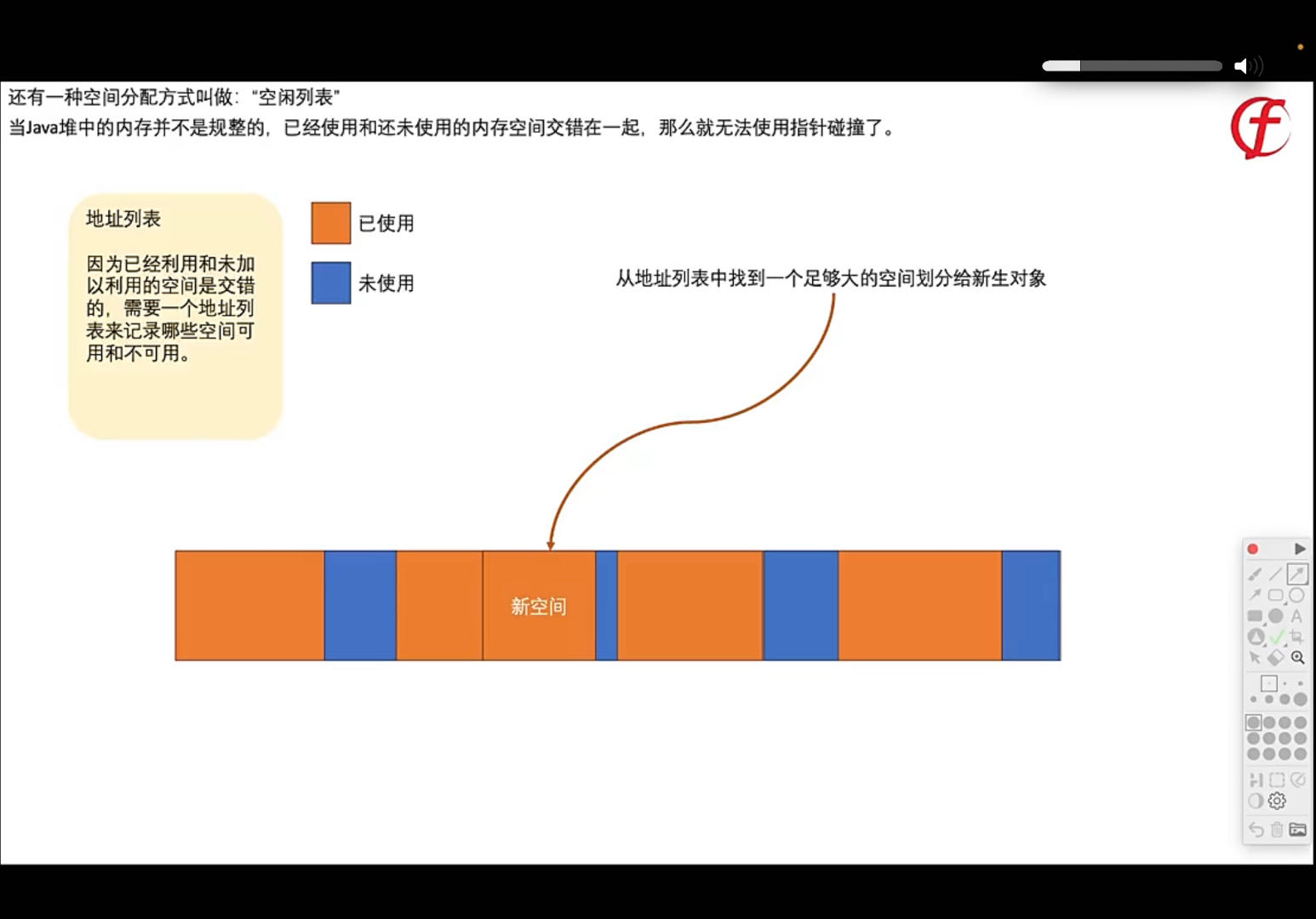


1. 生命周期
   1. 启动，执行，结束
2. 运行时数据区域



* + 1. 虚拟机栈，本地方法栈，程序计数器是线程私有的
    2. 方法区和堆是线程共有的
       1. 堆的空间是不是一定是连续的，java虚拟机官方规范中没有要求堆一定是连续的，而大多数虚拟机为了开发 维护容易写成可连续的
  1. 程序计数器
     1. 一个只读的存储器，用于记录Java虚拟机正在执行的字节码指令的地址
  2. Java虚拟机栈
     1. 线程私有的存储器，每个⽅法在执⾏的同时都会创建⼀个栈帧（StackFrame）⽤于存储局部变量表、操作数栈、动态链接、⽅法出⼝等信息
  3. 本地方法栈
     1. 本地方法栈是一种特殊的栈，他和Java虚拟机栈有着相同的功能，但是它支持本地代码的执行，本地⽅法栈中就是C和C++的代码
  4. 堆（里面包含字符串常量池）
     1. 是存储对象实例，所有的对象实例以及数组都要在堆上分配。
  5. 方法区（永久代）
     1. 用于存储已被加载的类信息，常量，静态变量，即时编译后的代码等数据的内存区域
     2. 运行时常量池：存在于方法区中，用于存储编译生成的信息，主要有字面量和符号引用常量两种
  6. 直接内存和堆内存的区别
     1. 直接内存申请空间耗费很⾼的性能，堆内存申请空间耗费⽐较低
     2. 直接内存的IO读写的性能要优于堆内存，在多次读写操作的情况相差⾮常明显

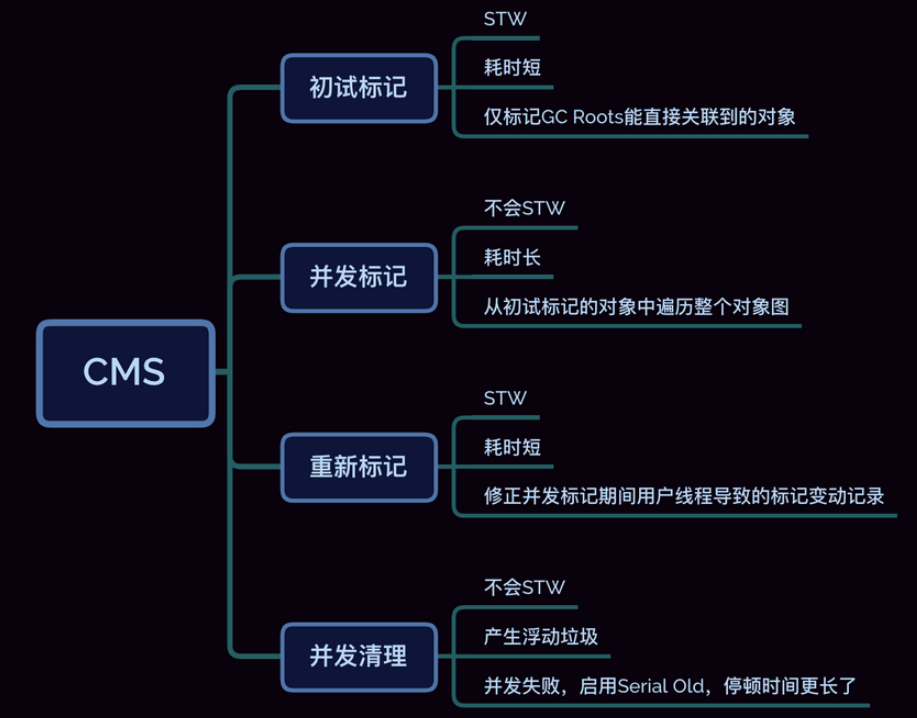
1. 分配对象的两种方式
   1. 指针碰撞
   2. 空闲列表
      1. 存在一个地址列表



1. 垃圾回收机制（GC）
   1. final，finally，finallize（）区别
      1. final 修饰的成员变量，必须在声明的同时赋值，⼀旦创建不可修改（常量）；修饰的⽅法，不能被⼦类重写；修饰的类不能被继承；
      2. finally 是异常处理语句结构 的⼀部分，不管是否有异常，他的语句总是执⾏。 ⼀般⽤来存放⼀些关闭资源的代码。
      3. finalize 是Object类的⼀个⽅法，主要是在收集器执⾏的时候会调⽤被回收对象的此⽅法，进⾏资源回收，例如关闭⽂件等。
   2. 新生代，老年代（1：2）
      1. 新⽣代 ( Young ) 被细分为 Eden 和 两个 Survivor 区域（8：1：1）JVM 每次只会使⽤ Eden 和其中的⼀块 Survivor 区域来为对象服务，所以⽆论什么时候，总是有⼀块 Survivor 区域是空闲着的
   3. Minor GC、Major GC、Full GC
      1. 区别
         1. 1. Minor GC是新⽣代GC，指的是发⽣在新⽣代的垃圾收集动作。当Eden没有⾜够空间的时候就会触发jvm发起⼀次Minor GC。由于java对象⼤都是朝⽣夕灭的，所以Minor GC⾮常频繁，⼀般回收速度也⽐较快
         2. 2. Major GC是⽼年代GC，指的是发⽣在⽼年代的GC，通常执⾏Major GC会连着Minor GC⼀起执⾏。Major GC的速度要⽐Minor GC慢的多。
         3. 3. Full GC是清理整个堆空间，包括年轻代和⽼年代
      2. Minor GC触发条件
         1. 当申请⼀个对象时，发现eden区不够⽤，则触发⼀次MinorGC。
      3. Major GC和Full GC 触发条件
         1. 1. 每次晋升到⽼年代的对象平均⼤⼩>⽼年代剩余空间
         2. 2. MinorGC后存活的对象超过了⽼年代剩余空间
         3. 3.老年代空间不⾜
         4. 4. 执⾏System.gc()（有可能会触发，也有可能是触发Minor GC）
         5. 5. 堆内存分配很⼤的对象
   4. JVM如何判断对象是否存活
      1. •引用计数法：给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。
      2. 可达性分析算法：这个算法的基本思想就是通过一系列的称为，"GC Roots”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。
         1. 优点
            1. 解决了循环依赖问题
            2. 与引用计数法相比无缺点
   5. 垃圾回收算法（GC算法）
      1. 标记复制（新生代垃圾回收算法）
      2. 标记整理（老年代）
      3. 标记清除
   6. 垃圾回收器



* + - 1. Parallel Scavenge(/ˈpærəlel/ /ˈskævɪndʒ/) 收集器（新生代，标记复制）⾼吞吐量为⽬标，（减少垃圾收集时间，让⽤户代码获得更⻓的运⾏时间）吞吐量 = 运⾏⽤户代码时间 / (运⾏⽤户代码时间 + 垃圾收集时间)
      2. Parallel old（老年代，标记整理）
      3. CMS（老年代，标记清除）
         1. 标记的四个阶段



* + - * 1. 在重新标记阶段结束之后，垃圾回收器会执行清除操作，将未被标记为可达对象的对象进行回收，从而释放内存空间。这个过程中，垃圾回收器会将所有未被标记的对象标记为白色
        2. 三色标记法

jvm中一种垃圾标记的算法。解决了循环引用问题和STW时间长

三种颜色的含义

白色：对象没有被标记过

灰色：该对象已经被标记过了，但该对象的引用对象还没标记完

黑色：该对象已经被标记过了，并且他的全部引用对象都标记完了

标记过程

1. 刚开始，所有的对象都是⽩⾊，没有被访问。

2. 将GC Roots直接关联的对象置为灰⾊。

3. 遍历灰⾊对象的所有引⽤，灰⾊对象本身置为⿊⾊，引⽤置为灰⾊。

4. 重复步骤3，直到没有灰⾊对象为⽌。

5. 结束时，⿊⾊对象存活，⽩⾊对象回收

错标（多标）

漏标满足的条件

至少有一个黑色对象在自己被标记之后指向了这个白色对象

所有的灰色对象在自己引用扫描完成之前删除了对白色对象的引用

cms采用增量更新，破坏了第一个条件

增量更新就是实时记录变化，确保每一次变化都会被重新检查，避免漏掉任何可能被错误视为垃圾的活跃对象

G1采用的是原始快照，破环了第二个条件

如果灰色对象在扫描完成前删除了对白色对象的引用，那么我们就在灰色对象取消引用之前，先将灰色对象引用的白色对象记录下来。在后续「重新标记」阶段再以这些白色对象为根，对它的引用进行扫描，从而避免了漏标的问题。通过这种方式，原本漏标的对象就会被重新扫描变成灰色，从而变为存活状态。

漏标

原本不是垃圾，但是GC的过程中，⽤户线程将其引⽤关系修改，导致GC Roots不可达，成为了垃圾。这种情况还好⼀点，⽆⾮就是产⽣了⼀些浮动垃圾，下次GC再清理就好了。

* + - 1. G1(标记整理+复制)
         1. 整体上看是标记整理，局部上看是标记复制
         2. G1采⽤了开创性的局部收集的设计思路和以Region为基本单位的内存布局⽅式，它将Java堆空间划分成多个⼤⼩相等的独⽴区域(Region)，JVM⽬标是总共不超过2048个Region(由JVM源码参数TARGET\_REGION\_NUMBER定义)，虽然可以超过该值，但不推荐。
         3. G1将所有Region分为四种类型：Eden/ˈiːdn/、Survivor/səˈvaɪvə(r)/、Old、Humongous/hjuːˈmʌŋɡəs/。
         4. 默认新⽣代的Region内存占堆空间的5%，分为Eden区和Survivor区，默认⽐例也是8:1:1
         5. G1为⼤对象的内存分配专⻔设计了⼀个Humongous类型的Region，在进⾏Full GC的时候除了要收集新⽣代和⽼年代的Region外，还会将Humongous的Region⼀并进⾏回收
         6. G1在进⾏垃圾收集的时候，会根据每个Region预计垃圾收集所需时间与预计回收内存⼤⼩的占⽐来选择对哪些区域进⾏回收，采⽤⼀种Mixed GC的⽅式，即混合回收的GC⽅式
         7. 垃圾收集步骤

初始标记需要暂定所有线程，即STW，并记录下GC Roots能直接引⽤的对象，速度很快

并发标记可以与应⽤线程⼀起⼯作，进⾏可达性分析

最终标记需要暂定所有线程(STW)，根据三⾊标记算法修复⼀些引⽤的状态，原始快照

筛选回收筛选回收阶段会对各个Region的回收价值和成本进⾏排序，根据⽤户所期望的GC停顿STW时间来制定回收计划

在筛选回收阶段对各个Region进⾏回收价值和成本进⾏排序，这句话怎么理解？⽐如现在有Region1、Region2和Region3三个区域，其中Region1预计可以回收1.5MB内存，预计耗时2MS；Region2预计可以回收1MB内存，预计耗时1MS；Region3预计可以回收0.5MB内存，预计耗时1MS。那么Region1、Region2和Region3各⾃的回收价值与成本⽐值分别是：0.75、1和0.5。⽐值越⾼说明同样的付出，收益越⾼，如果此时只能回收⼀个Region的内存空间，G1就会选择Region2进⾏回收。这种⽅式保证了G1收集器在有限的时间内尽可能地提⾼收集效率。

* + - * 1. 为什么G1使⽤原始快照，⽽CMS使⽤增量更新？

主要原因是增量更新是⼀种深度扫描的算法，CMS中只有⼀块⽼年代，即使进⾏深度更新也没有什么问题。但G1有很多个Region，且对象的引⽤可能分散在多个Region中，如果这种情况还选择深度扫描效率就会很低了。⽽原始快照相对增量更新就会快很多，虽然原始快照可能产⽣浮动垃圾，但产⽣的浮动垃圾下⼀次再回收就好了，并且这部分浮动垃圾也不会很多。所以G1选择原始快照，只是将对象简单标记成⿊⾊，保证本次垃圾收集不进⾏回收就可以了，等下⼀次GC时再做深度更新。

* + - * 1. 由⽤户指定期望的停顿时间是G1收集器很强⼤的⼀个功能，通常把期望停顿时间设置为⼀两百毫秒或者两三百毫秒会是⽐较合理的。
        2. 垃圾收集分类

Young GC

G1与之前垃圾收集器的Young GC有所不同，并不是当新⽣代的Eden区放满了就进⾏垃圾回收，G1会计算当前Eden区回收⼤概需要多久的时间，如果回收时间远⼩于参数-XX:MaxGCPauseMills设定的值，那么G1就会增加年轻代的Region(可以从⽼年代或Humongous区划分Region给新⽣代)，继续给新对象存放；直到下⼀次Eden区放满，G1计算回收时间接近参数-XX:MaxGCPauseMills设定的值，那么就会触发Young GC。

Mixed GC

如果⽼年代的堆空间内存占⽤达到了参数-XX:InitiatingHeapOccupancyPercent设定的值就会触发Mixed GC，回收所有的新⽣代和部分⽼年代(根据⽤户设置的GC停顿时间来确定⽼年代垃圾收集的先后顺序)以及Humongous区。正常情况下G1的垃圾收集是先做Mixed GC，主要使⽤复制算法，需要把各个Region中存活的对象复制到另⼀个空闲的Region，如果在复制过程中发现没有⾜够的空Region放复制的对象，那么就会触发⼀次Full GC。

Full GC

停⽌系统程序，然后采⽤单线程进⾏标记、清理和压缩整理，好空闲出来⼀批Region来供下⼀次MixedGC使⽤，这个过程是⾮常耗时的。G1收集器的Region内存回收时，会涉及到⼤量跨区引⽤的对象，解决⽅式也是通过记忆集(卡表)。每个Region都要维护⼀个其他Region对⾃⼰内部对象的引⽤。CMS中只有新⽣代和⽼年代，即使出现了跨代引⽤，也很好解决。但G1是把堆分成了多个Region，Region中对象的可能被多个Region引⽤，所以G1的卡表实现要⽐CMS复杂很多。

* 1. 空间担保机制

1. Java中的四种引用
   1. 强引用
      1. 强引用：强引用是Java的默认引用形式，使用时不需要显示定义。如果一个对象具有强引用，那垃圾回收器绝不会回收它（可达时）。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。
   2. 弱引用
      1. 如果一个对象只具有弱引用，无论内存充足与否，JavaGC后对象如果只有弱引用将会被自动回收
   3. 软引用
      1. 软引用：软引用不会保证对象一定不会被回收，只能最大可能保证。软引用和弱引用的特性基本一致，主要的区别在于软引用在内存不足时才会被回收。如果一个对象只具有软引用，Java GC在内存充足的时候不会回收它，内存不足时才会被回收。
   4. 虚引用
      1. 虛引用：java.lang.ref.PhantomReference 类中只有一个方法 get0，而且几乎没有实现，只是返回 null.如果一个对象仅有虚引用，那么它就像没有任何引用一样，在任何时候都可能被gc 回收。虚引用主要用来跟踪对象被垃圾回收的活动。
2. 类加载器
   1. 双亲委派机制
      1. 双亲委派机制，其⼯作原理的是，如果⼀个类加载器收到了类加载请求，它并不会⾃⼰先去加载，⽽是把这个请求委托给⽗类的加载器去执⾏，如果⽗类加载器还存在其⽗类加载器，则进⼀步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果⽗类加载器可以完成类加载任务，就成功返回，倘若⽗类加载器⽆法完成此加载任务，⼦加载器才会尝试⾃⼰去加载，这就是双亲委派模式，即每个⼉⼦都很懒，每次有活就丢给⽗亲去⼲，直到⽗亲说这件事我也⼲不了时，⼉⼦⾃⼰才想办法去完成。
      2. 优势：避免重复加载 + 避免核⼼类篡改