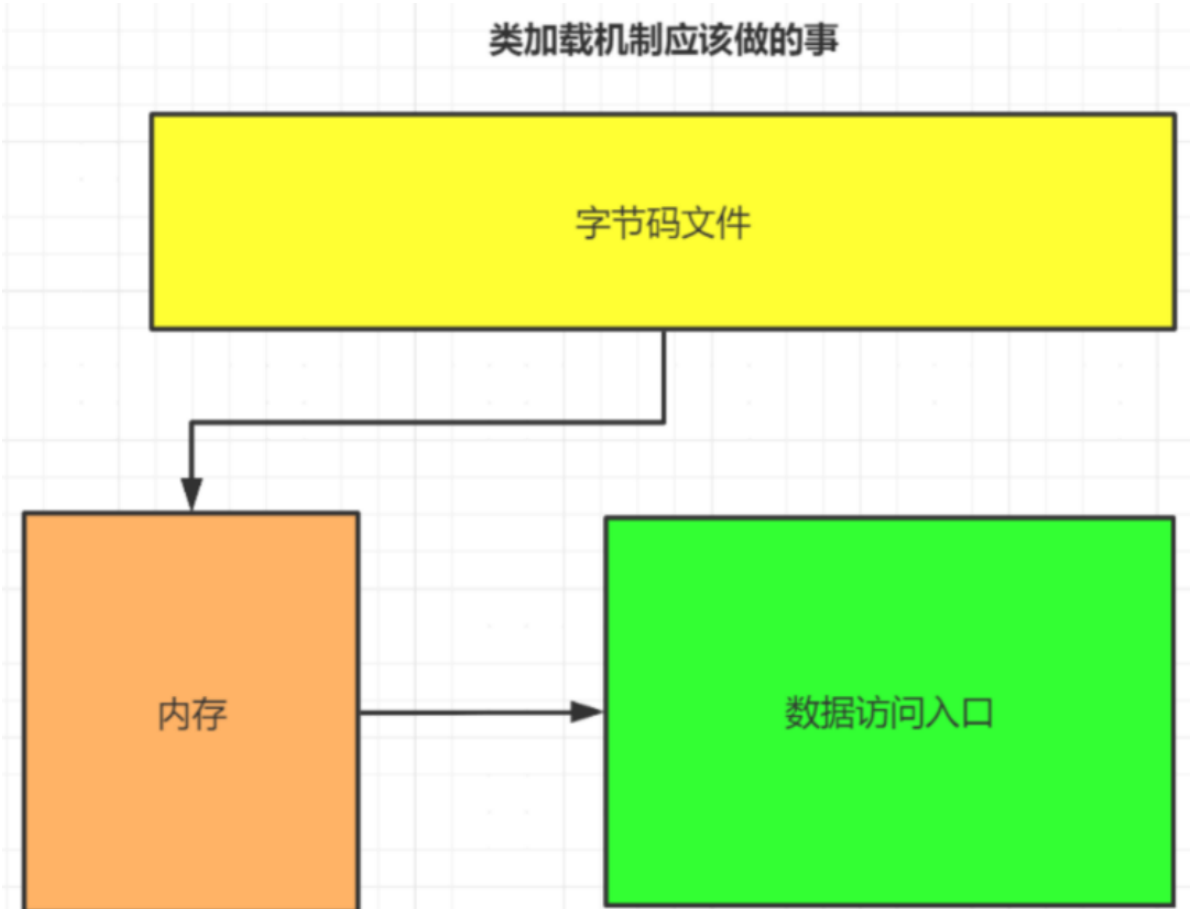
1、JVM 类加载

1、类加载的流程

将 类的字节码 文件读入到 内存，同时生成数据的访问入口。



加载到 什么位置？？

方法区（元空间）：类信息，静态变量，常量

堆（）：指向 方法去的 类引用

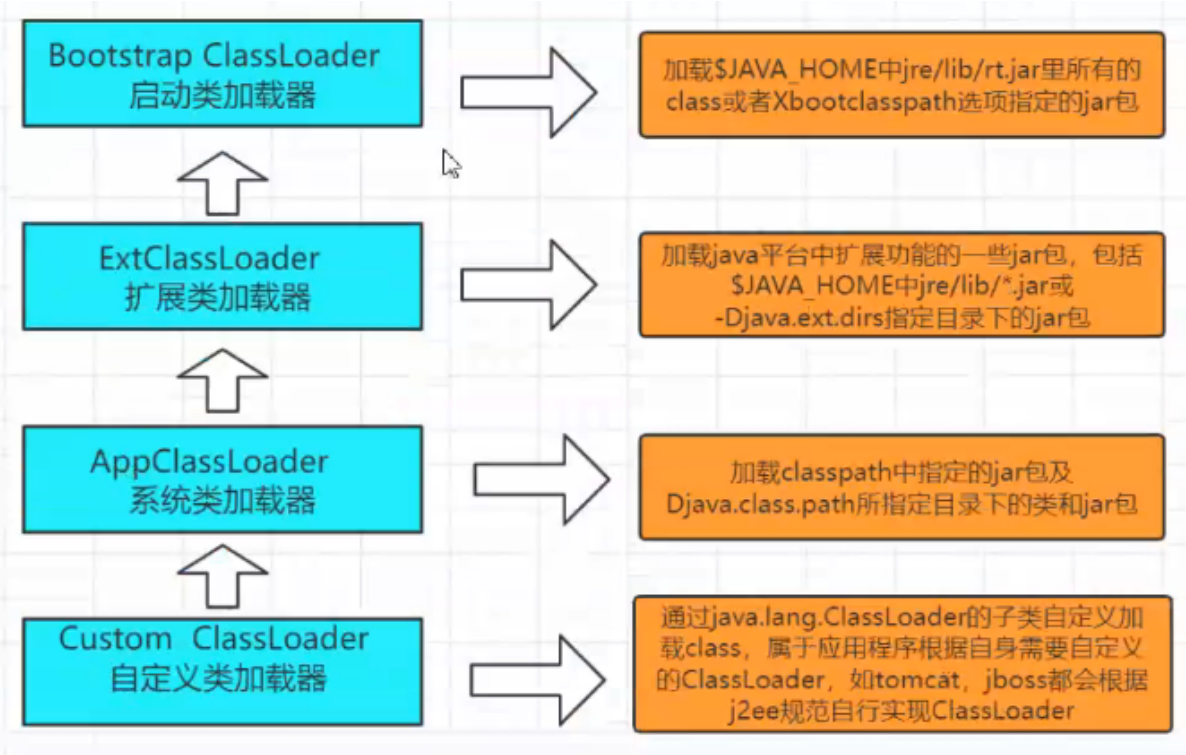
```
1、装载
    a、通过类的 全限定名 获取类的字节码文件。
        类加载器：查找类的 字节码文件的 代码模块。
    b、将 字节码 转为 方法区的运行时数据结构。
    c、在 Java堆 中 生成 类的 Class 对象，作为对方法区中这些数据的访问入口
2、链接
    1、验证（取消验证：-Xverify:none）
        a、jvm   按照 字节码 格式 验证字节码文件是否符合。
        b、符号引用验证：对 常量池中 各种符号引用 的校验
    2、准备
        为类的 静态变量 分配内存，赋默认值?。
        注：
            1.final 在 编译的时候就会分配，此时显式初始化。
            2、ConstantValue 同时final + static 修饰的 基本数据类型 或者 String 生成
字面量的值（常量池）
    3、解析
        将 类 符号引用 转换为 直接引用
```

3、初始化
    执行 class init();
    触发类的初始化的时机：
        1、创建类对象
        2、访问类的静态变量。final + static 的 引用，不会触发
        3、调用静态方法。
        4、反射
        5、初始化类的子类，其父类会被初始化。

2、类加载器

1、三大特性

1、全盘委托
    当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入
2、双亲委派
    指子类加载器如果没有加载过该目标类，就先委托父类加载器加载该目标类，只有在父类加载器找不到字节码文件的情况下才从自己的类路径中查找并装载目标类。
3、缓存机制
    所有加载过的Class都将在内存中缓存
    类变量为什么只会被初始化一次？？？？

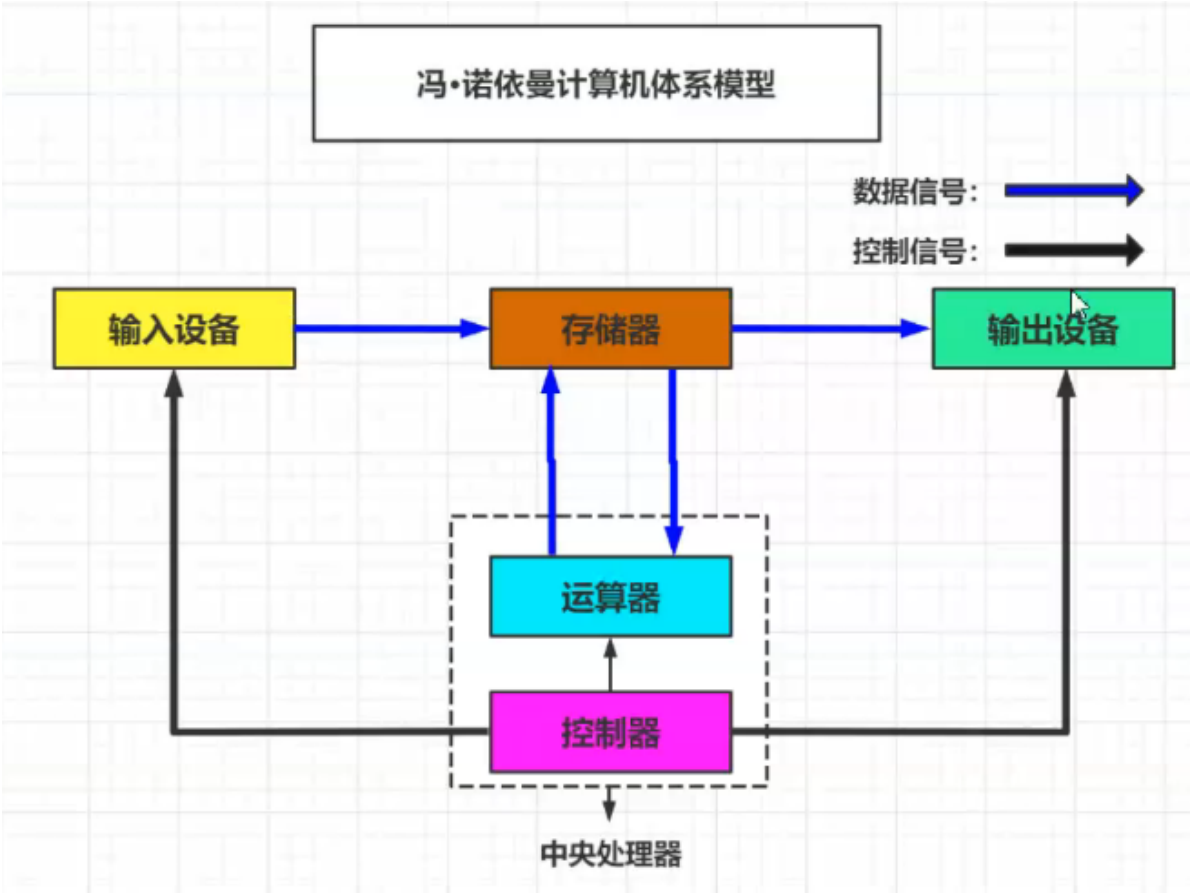


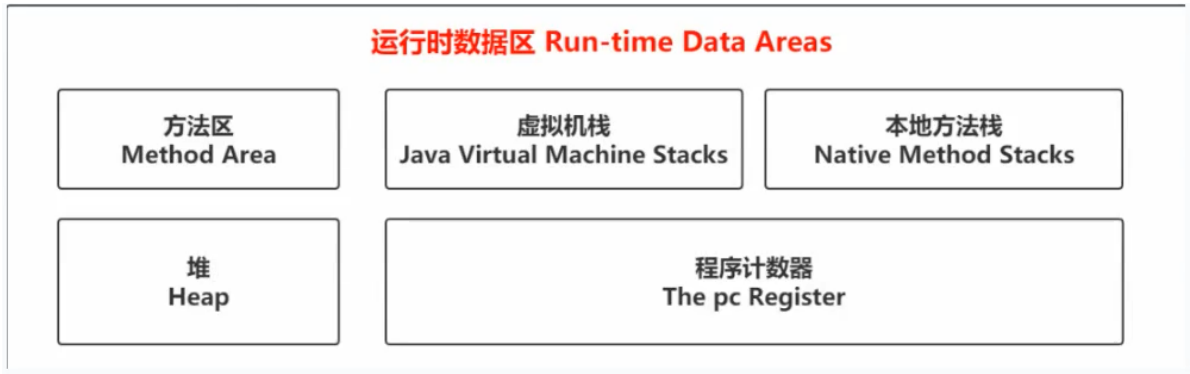
2、双亲委派的打破

1、继承 java.lang.ClassLoader 类。
    a、不打破：重写findClass(name)方法
    b、打破： 重写loadClass(name)方法

2、运行时数据区

1、计算机模型体系

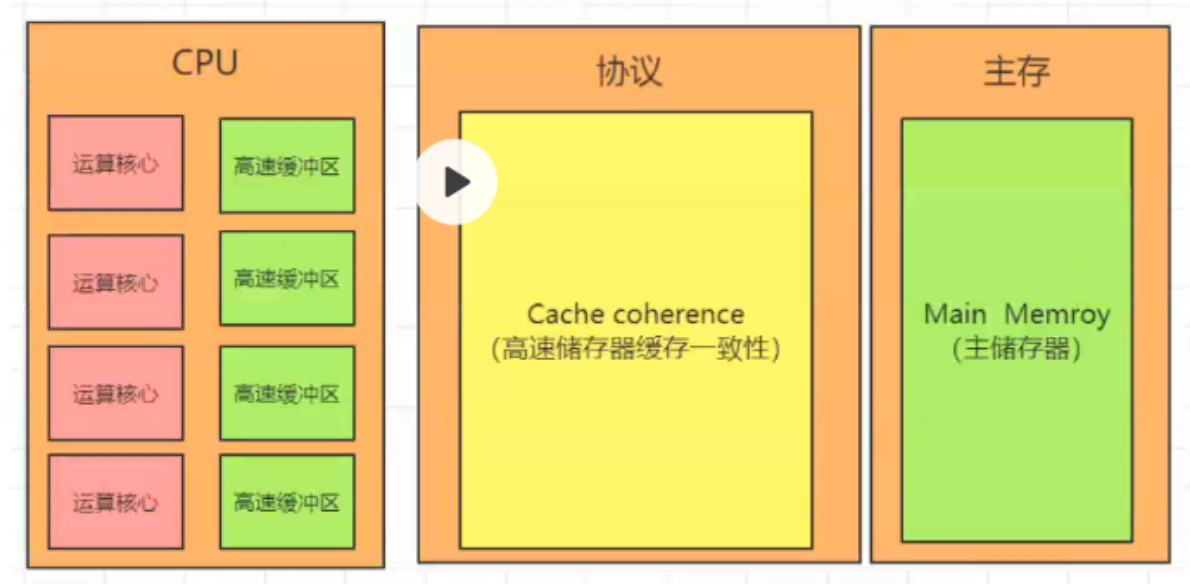





```
1、输出设备：执行引擎
2、输入设备：类加载器
3、存储器：堆，方法区
4、处理器：本地方法栈，java虚拟机栈(栈帧)

栈帧：{
      局部变量表；
      操作数栈；
      动态连接；多态实现的基石
      返回地址；
      附加信息；
}
```

2、cpu内存交互




```
1、缓存一致（MESI）
    modify
    exclude
    share
    invalid
```

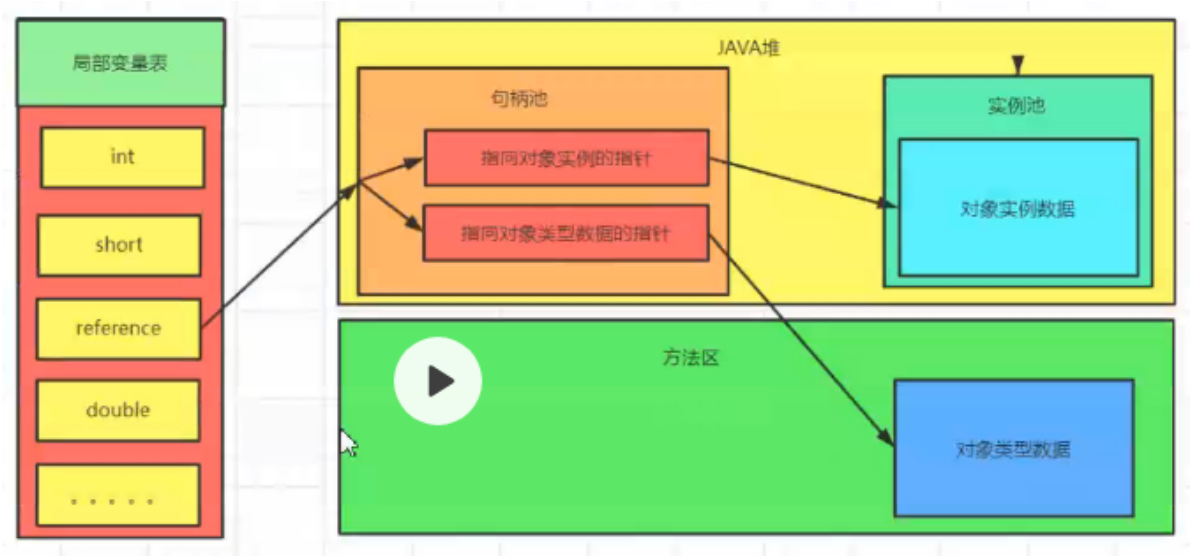
3、对象的内存布局

1、为什么要知道对象的大小

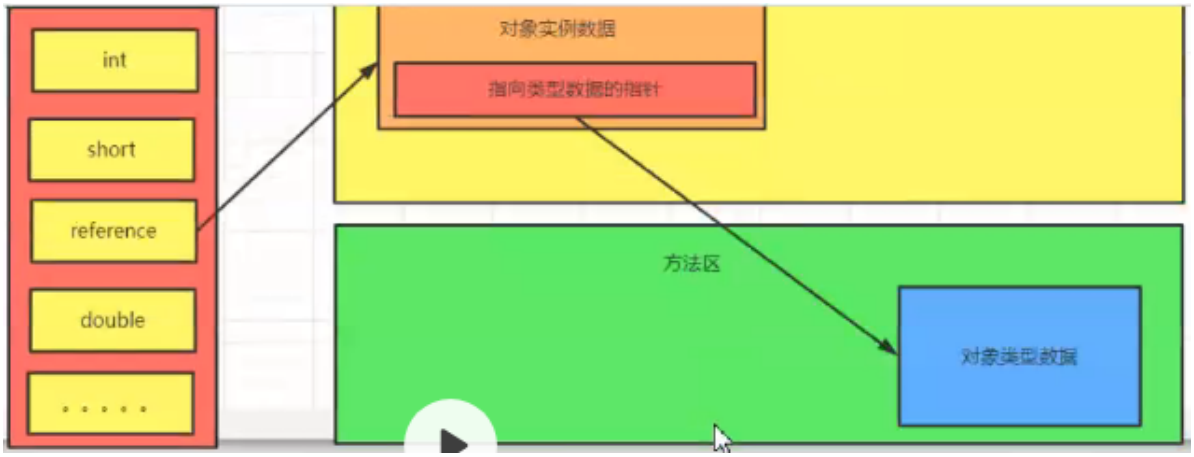
```
1、开启指针压缩：-XX:-UseCompressedOops
2、指针压缩为什么超过 32G 无效
    对象指针在堆中是32位，在寄存器中是35位，2的35次方=32G
```

2、指针模型

1、句柄池访问



2、直接指针访问



3、对象的存储方式

1、大端存储

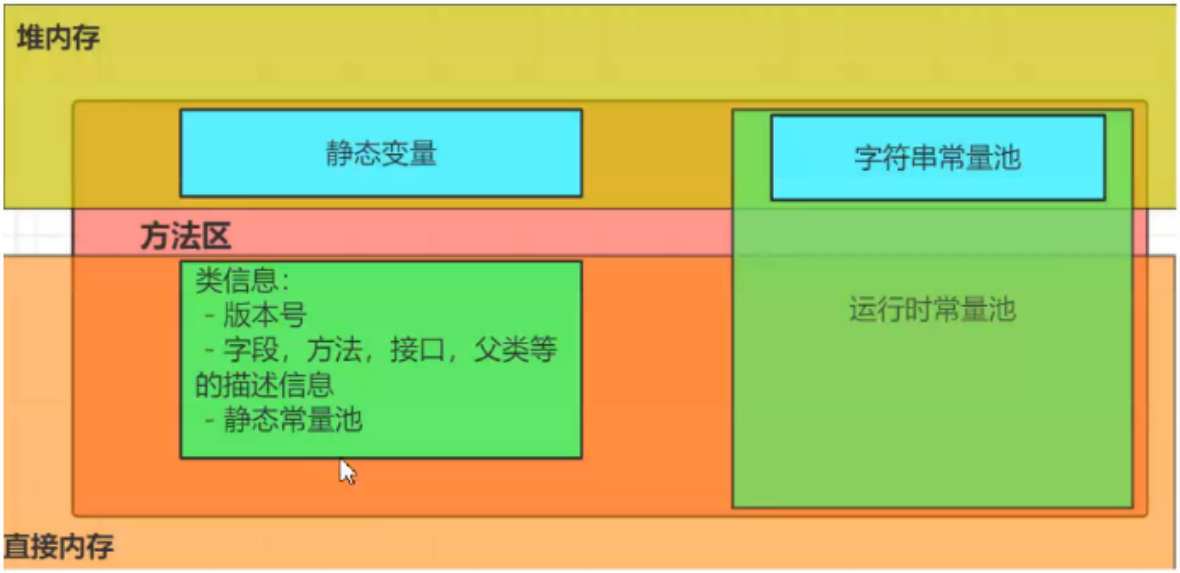
高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。

2、小端存储（java 默认）

低字节排放在内存的低地址端，高位字节排放在内存的高地址端

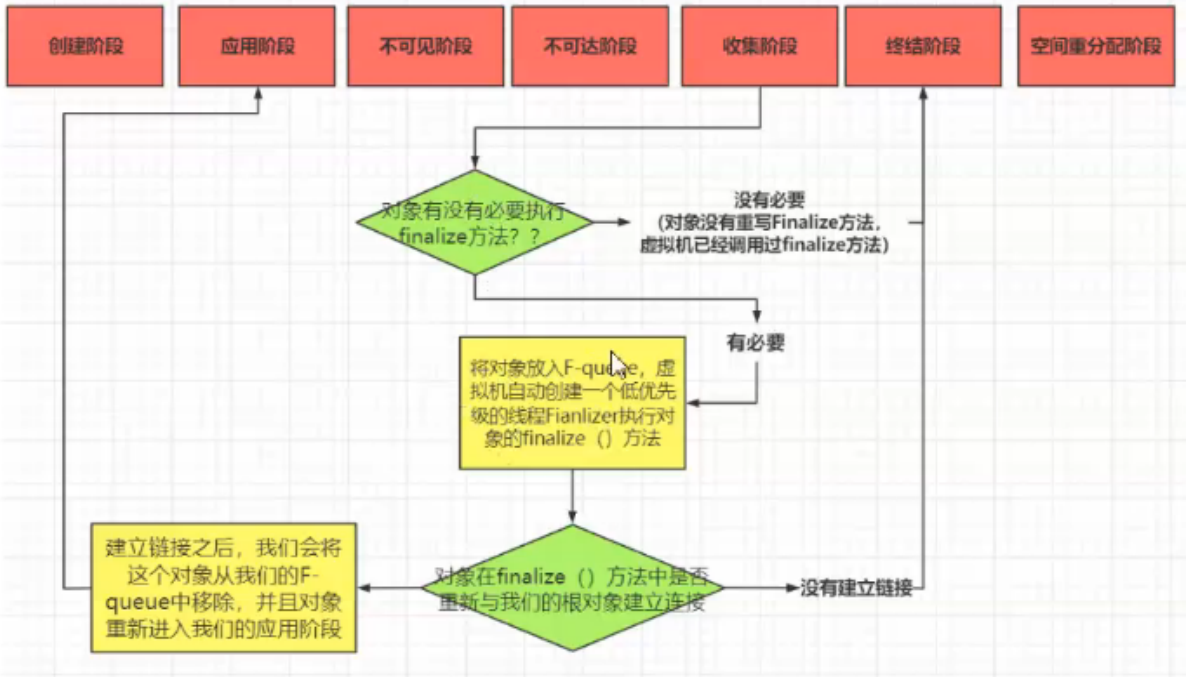
4、常量池在内存中的布局

直接内存：由 OS 管理的内存



4、内存模型与对象已死

1、对象的生命周期



2、对象复活

1、类 重写 `finilize()` [复活币] ，重新建立指向当前对象的连接。

3、引用

1、强引用
2、软引用
3、弱引用
　　图片缓存导致内存泄漏
4、虚引用

5、垃圾回收算法

1、对象的分配策略

1、首次适应算法
2、最佳适应算法
3、最坏适应算法
4、邻近适应算法

2、垃圾回收算法

1、复制算法

将内存划分为两块相等的区域，每次只使用其中一块。
当其中一块内存使用完了，就将还存活的对象复制到另外一块上面，然后把已经使用过的内存空间一次清除掉。

缺点：
a、空间利用率低

2、标记-清除

1、标记阶段：找出内存中需要回收的对象，并且把它们标记出来
2、清除阶段：清除掉被标记需要回收的对象，释放出对应的内存空间

缺点：
a、两个过程都比较耗时，效率不高
b、会产生大量不连续的内存碎片，导致后续大对象分配失败

3、标记-清除-压缩

1、标记阶段：找出内存中需要回收的对象，并且把它们标记出来
2、清除阶段：清除掉被标记需要回收的对象，释放出对应的内存空间
3、整理阶段：将存活的对象整理到一起，减少空间碎片的产生

缺点：
a、耗时，效率不高

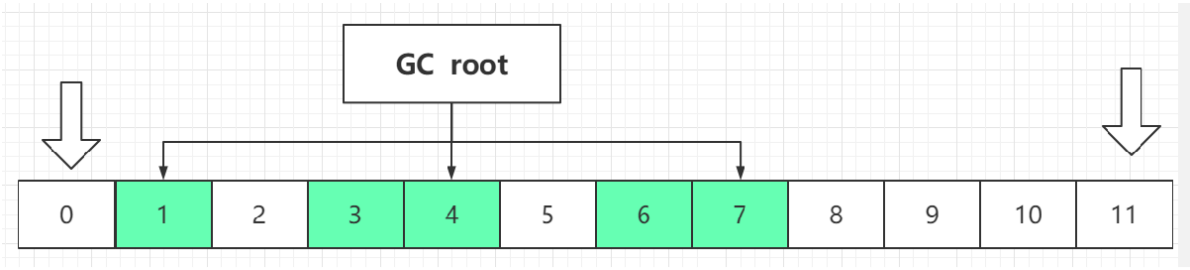
3、整理算法

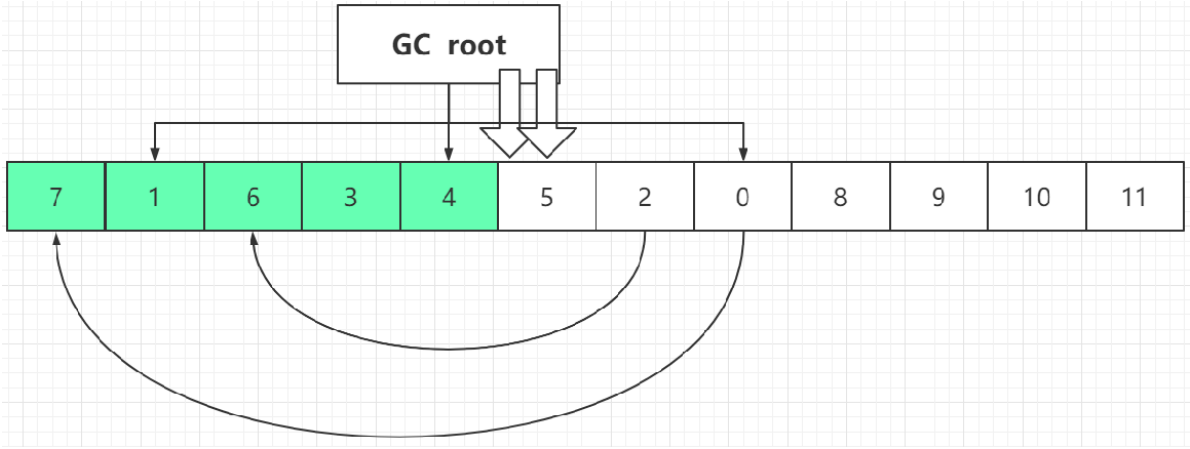
1、随机整理

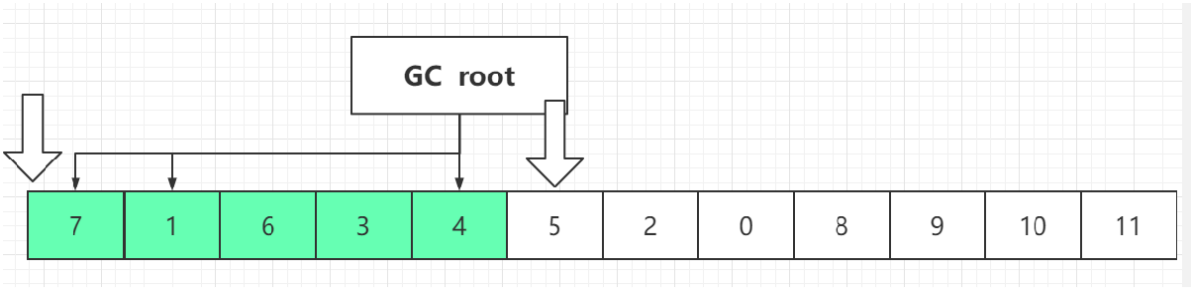
1、双指针回收算法（快速排序）

实现：
a、整理前：两根指针分别位于内存的首尾段
b、第一次遍历：移动位置但是并不更新标记
c、第二次遍历：更新标记

优点：实现简单且速度快
缺点：但会打乱对象的原有布局，破坏局部性原理（空间局部性，时间局部性）







2、线性整理

相关的对象会进行整理，整理成一块块小区域，无法避免内存碎片

3、滑动整理

1、Lisp2算法

实现：

a、整理前：他是一个三指针算法，首（scan）尾(end)指针 + 移动(free)指针
b、第一次遍历：Free指针是为了留位置，而Scan对象是为了找存活对象（占用的空间大小）
c、第二次遍历：更新对象地址
d、第三次遍历：移动对象

2、单次遍历算法

减少了Lisp2 中的 c 步骤，用一个表记录对象需要移动到的位置

4、分代假说

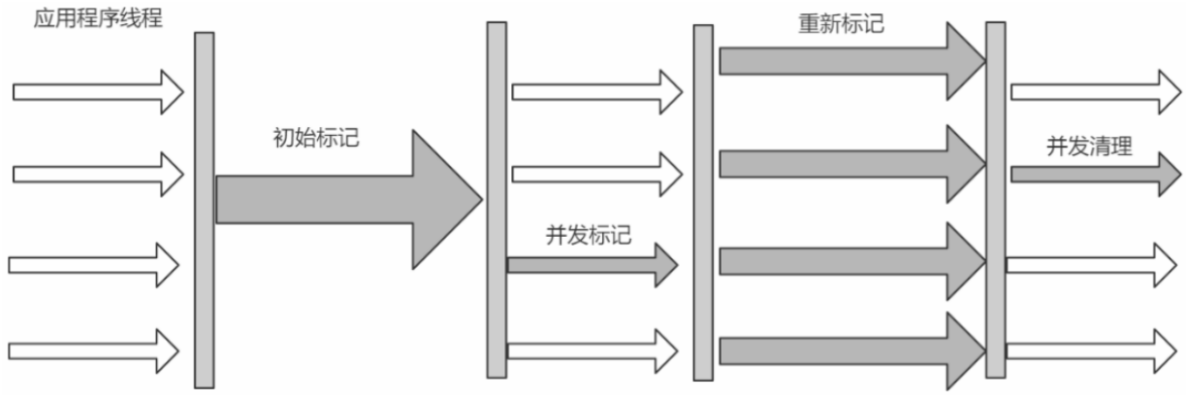
1、弱分代假说：绝大多数对象朝生夕死
2、强分代假说：熬过很多次垃圾回收的对象是越来越难以消亡的
3、跨代引用假说：跨代引用的对象占比很少。

补充：跨代垃圾回收
    1、新生代引用老年代：MajorGC 前伴随着一次 YoungGC
    2、老年代引用新生代：记忆集（卡表）

6、垃圾收集器

https://blog.csdn.net/qq_44734154/article/details/125839186

1、CMS垃圾收集器 （降停顿）

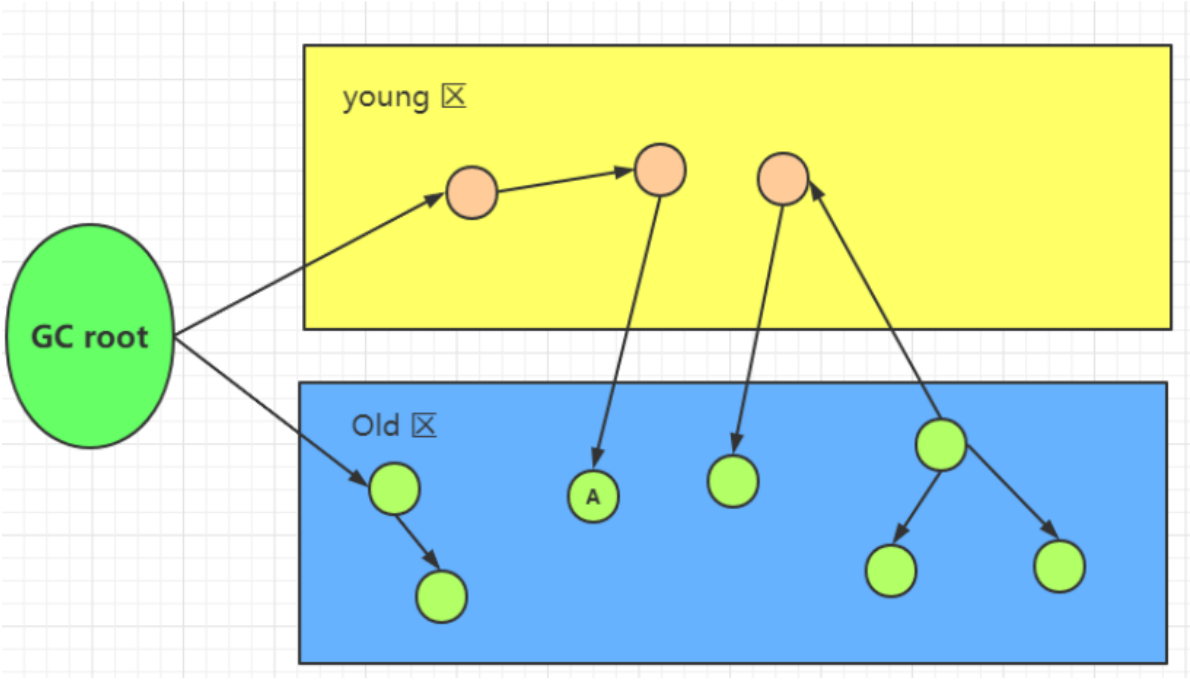


(1)初始标记
(2)并发标记
(3)并发预处理
(4)可中止的预处理
(5)重新标记
(6)并发标记

1、两种模式，一种策略，一种机制

并发预处理：处理新生代

1、正常模式（BackGround）




```
1、如何判断 被年轻代对象引用的 老年代对象可达
    只有在回收老年代的时候，先扫描一边新生代。
2、扫描新生代，就会导致停顿时间增加
    回收老年代之前就提前触发一次 Young GC,(弱分代假说)剩余的新生代对象很少。
```

2、并发失败（ForeGround）

并发标记的时候，内存不够了

```
1、并发搜集器不能在年老代填满之前完成不可达（unreachable）对象的回收
2、年老代中有效的空闲内存空间不能满足某一个内存的分配请求

出现这种情况：CMS 进入 STW，使用 Serial Old 单线程垃圾回收。

解决：尽量避免并发失败的情况
    -XX:+UseCMSInitiatingOccupancyOnly：只是用设定的回收阈值
    -XX:CMSInitiatingOccupancyFraction ：上不指定，则在后续自动调整，老年代占用多少后触发MajorGC
```

3、压缩策略（Mark Sweep Compact）

```
每次 MajorGc 后 需要整理空间，耗时严重。
    -XX:+UseCMSCompactAtFullCollection
    -XX:CMSFullGCsBeforeCompaction=0
```

4、OOM机制

```
-XX:-UseGCOverheadLimit
假如在垃圾搜集的时间超过总时间的 98% ，垃圾清理的时间小于 2%;会抛出 OutOfMemeryError
```

2、可终止的预处理

并发标记 = 并发预处理（处理新生代）+ 可终止预处理

```
提前处理一些 垃圾回收的前置过程
    开始触发 ：CMSScheduleRemarkEdenSizeThreshold    2M  Eden空间使用超过2M的时候启动可中断的并发预清理
    中断时间 ：CMSScheduleRemarkEdenPenetration  50% 到Eden空间使用率到达50%的时候中断（但不是结束），进入Remark（重新标记阶段）。
    结束 ：minorGC (新生代的GC);
    最长执行 ：CMSMaxAbortablePrecleanTime       5S
```

3、三色标记

```
1、黑色：对象已经被垃圾收集器访问过， 且这个对象的所有引用都已经扫描过。
2、灰色：对象已经被垃圾收集器访问过， 但这个对象上至少存在一个引用还没有被扫描过。
3、白色：对象尚未被垃圾收集器访问过。

多标（本应被回收却没被回收）浮动垃圾，在下一次 GC 时回收
漏标（本不该回收却被回收）{灰断开白，且黑引用白} ( 写屏障实现 )
    a、增量更新（黑指白）:黑色对象 插入新的指向 白色对象的引用关系 时， 就将这个新插入的引用记录下来
    b、原始快照（灰断白）（STAB）:灰色对象 删除指向 白色对象的引用关系 时， 就将这个要删除的引用记录下来
```

4、常见参数

2、G1垃圾收集器

https://www.bilibili.com/read/cv16706692

1、G1的意义与特点

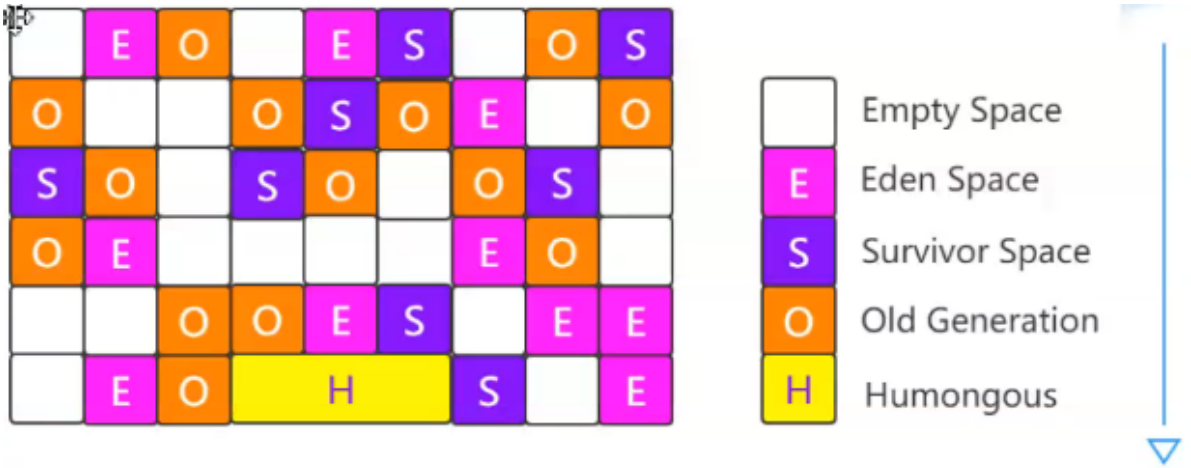
1.意义

```
在延迟可控的情况下，尽可能提高吞吐量。
```

2.特点

```
1、内存空间的重新定义
    逻辑分代，物理不分代
2、更短的停顿时间（要多短有多短）
3、某种程度上解决空间碎片（分页）
```

2、内存的重新定义

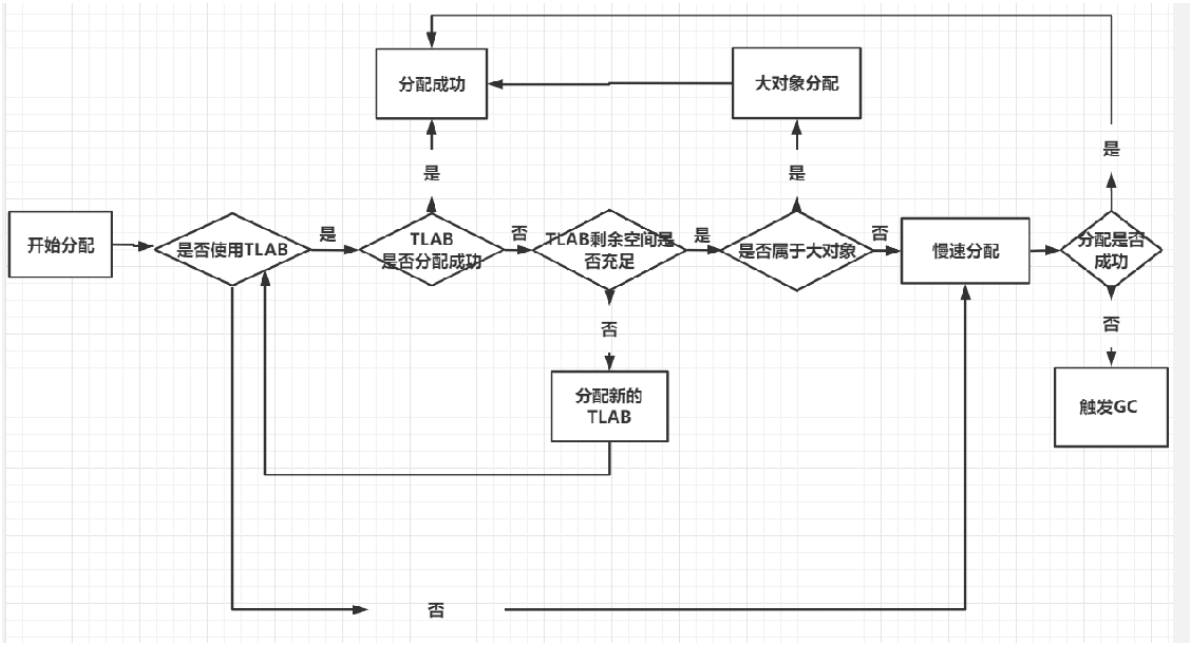



```
1、将对空间 划分为 2048 个 rigion,每个region 的大小范围 1~32M。
2、region 的角色
    a、freeTag
    b、youngHeap = eden + survivor
    c、Old Generation
    d、Hunonmous(大对象分区)
```

3、TLAB

线程本地分配缓冲区

分配空间时，为了减少临界区范围，提高执行效率，避免全局锁。为每一个线程分配一个独占的堆空间（1%）。



4、如何解决跨代垃圾回收

1、卡表 (老年代引用新生代)

-XX:+UseCondCardMark

我引用了谁：记忆集的实现
每个 region 分为 多个 512字节 的卡页。
当卡页中存在任意对象出现跨代引用，该卡页被标记为脏卡。
垃圾收集发生时，筛选出卡表中被标记为边脏的元素，就能轻易得出哪些卡页中包含跨代指针，把它们加入GC Roots中一并扫描。

卡表的维护采用 ： 写屏障。
G1主要在赋值语句中，使用写前屏障(Pre-Write Barrrier)和写后屏障(Post-Write Barrrier)。都是异步实现
    a、写前屏障：左侧对象将修改引用前，记录即将失去引用的对象
    b、写后屏障：右侧对象获取了左侧对象的引用，那么等式右侧对象所在分区的RSet也应该得到更新。
内存伪共享问题。不同线程对 对象引用的更新操作，恰好位于同一个64KB区域内（同一个缓存行），这将导致同时更新卡表的同一个缓存行，从而造成缓存行的写回、无效化或者同步操作，间接影响程序性能。
    不采用无条件的写屏障，而是先检查卡表标记，只有当该卡表项未被标记过才将其标记为dirty。
    这就是JDK 7中引入的解决方法，引入了一个新的JVM参数

2、Remember Set（新生代引用老年代）

谁引用了我:记录其他 Region 中的对象引用 本Region中对象 的 关系

实现：
    a、hash表
        本质是一个 hash 表，记录<region的起始地址，元素卡页索引号数组>
    b、细粒度位图
        一个 BitMap ,通过一个字节标识一个 卡页 的引用信息。当新的 point-in 出现时，将字节对应字段标记为·1
    c、粗粒度位图
        一个 BitMap ,通过一个字节标识一个 region 的引用信息。当新的 point-in 出现时，将字节对应字段标记为·1

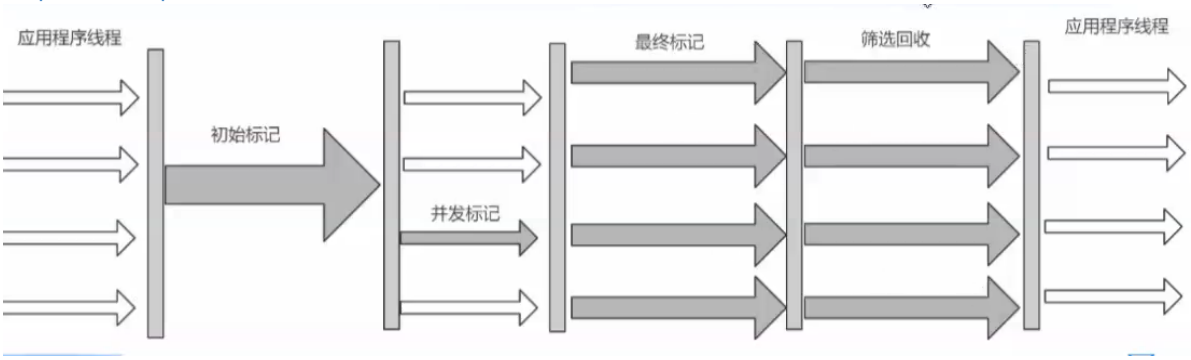
3、Collection Set (回收集)

每次GC暂停时回收的一系列目标分区。
内部存活的对象都会被转移到分配的空闲分区中。

因此无论是年轻代收集，还是混合收集，工作的机制都是一致的。
    年轻代收集CSet只容纳年轻代分区，
    混合收集会通过启发式算法，在老年代候选回收分区中，筛选出回收收益最高的分区添加到CSet中。

5、三大回收流程

https://www.processon.com/view/link/62bc50e47d9c08073522779c



1、YoungGC

-XX:MaxGCPauseMills 最大停顿时间

```
1.大小分配：Eden区的大小范围 = [ -XX:G1NewSizePercent, -XX:G1MaxNewSizePercent ] = [ 整堆5%, 整堆60% ]

2、回收策略：G1会计算下现在Eden区回收大概要多久时间，如果回收时间远远小于参数-XX:MaxGCPauseMills设定的值（默认200ms），那么增加年轻代的region，继续给新对象存放，不会马上做YoungGC。G1计算回收时间接近参数-XX:MaxGCPauseMills设定的值，那么就会触发YoungGC。

3.实现流程
    a、根扫描：
        处理 java 的根 ：已加载类的元数据 所有Java线程当前栈帧的引用和虚拟机内部线程
        处理 jvm 的根：
            JVM内部使用的引用
            JNI句柄
            对象锁的引用
            java.lang.management管理和监控相关类的引用
            JVMTI（JVM Tool Interface）的引用
            AOT静态编译的引用
        处理 String table 的引用：
    b、对象复制
        判断对象是否在CSet(对象的分区)中，如是则判断对象是否已经copy过了
            如果已经copy过，则直接找到新对象
            如果没有copy过，则调用copy_to_survivor_space函数copy对象到survivor区
                1、根据 age 判断当前对象 copy 到（新生代/老年代）
                2、先尝试在PLAB中分配对象（分配逻辑与 TLAB 类似）
                3、age 加1，
        修改老对象的对象头，指向新对象地址，并将老对象锁标志位置为11（GC 标志）（01，00，10）
    c、深度搜索复制
        并行线程处理完当前任务后，可以窃取其他线程没有处理完的对象
```

2、MixedGC

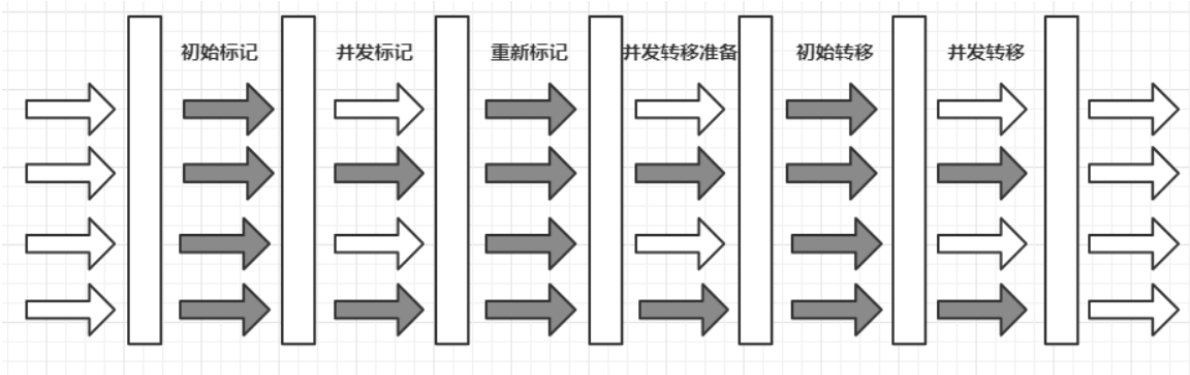
```
本质上不是只针对老年代，也有部分年轻代，所以又叫MixGC。
当old区Heap的对象占总Heap的比例超过 InitiatingHeapOccupancyPercent[45%] 之后，就会启动MixedGC

初始标记
并发标记
重新标记
并发清除
```

3、FullGC（+压缩）

```
Allocation Failure：类似与 CMS 并发失败，降级为 STW 的 fullGC
```

3、ZGC 垃圾收集器



1、三大核心技术

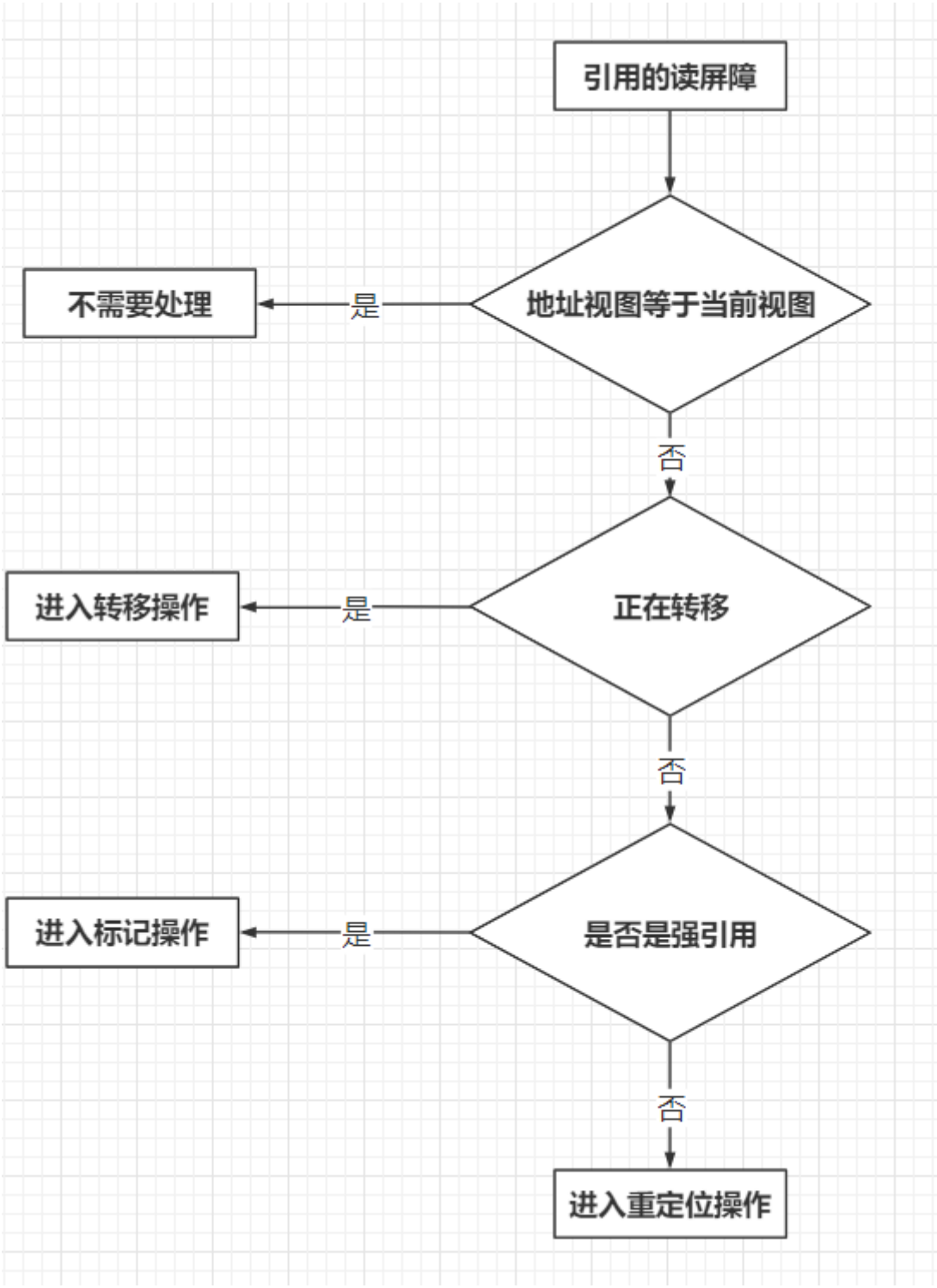
1、多重映射

同一内存地址，在不同的视图下面代表的含义不同，利用虚拟空间换时间

`Remapped，marked0，marked1`

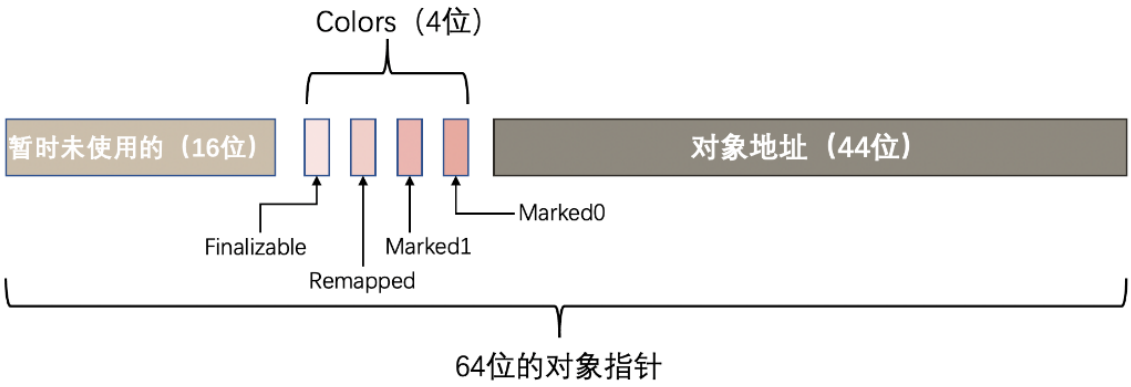
2、读屏障

当应用线程从堆中读取对象引用时，就会执行这段代码。



3、指针染色

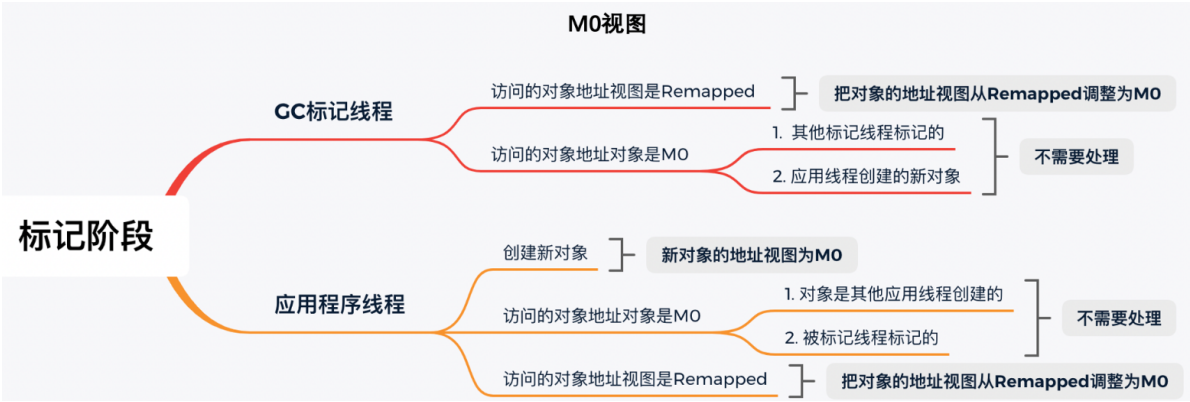
ZGC的染色指针技术继续盯上了这剩下的46位指针宽度，将其高4位提取出来存储四个标志信息。通过这些标志位，虚拟机可以直接从指针中看到其引用对象的三色标记状态、是否进入了重分配集（即被移动过）、是否只能通过finalize()方法才能被访问到。



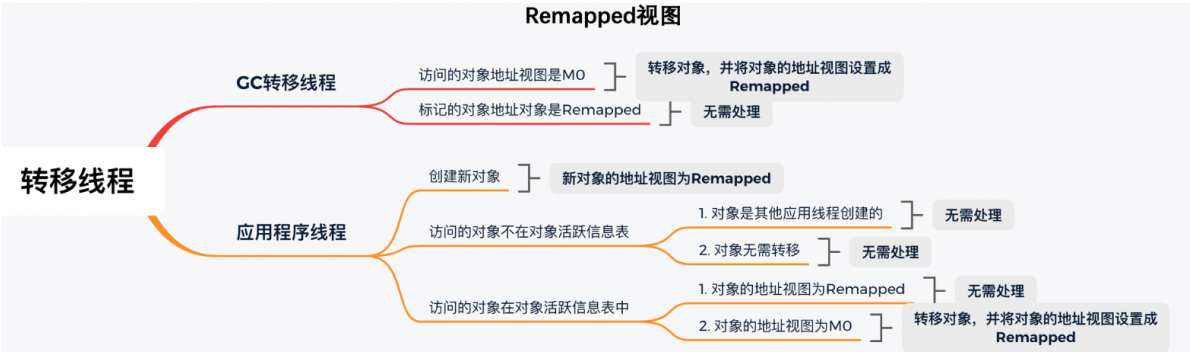
2、垃圾回收触发时机

```
1、初始化阶段：整个内存空间的地址视图被设置为Remapped
2、标记阶段：remapped --> M0/M1
3、转移阶段：M --> remapped
```

标记阶段：



转移阶段：



7、常用命令

8、即时编译器

1、执行引擎

1、解释执行

逐条把字节码翻译成机器码并执行

2、即时编译器

编译器先将字节码编译成对应平台的可执行文件，运行速度快。

把 热点代码 编译成与本地关联的机器码，提高执行效率。

热点代码：
1、被多次调用的方法：编译器会将整个方法作为编译对象
2、被多次执行的循环体：编译器依然会以整个方法（而不是单独的循环体）作为编译对象（OSR 栈上替换）

基于采样的热点探测：虚拟机会周期的对各个线程栈顶进行检查，如果某些方法经常出现在栈顶，这个方法就是“热点方法”。
基于计数的热点探测：
方法调用计数器 ：设定阈值-XX：CompileThreadhold；设定半衰期：-XX：CounterHalfLifeTime
C1 模式下，1500次
C2 模式下。10000次
注：提交的频次是 C1 + C2 超过阈值

回边计数器
C1 模式下,13995
C2 模式下，10700

2、分层编译的5个级别

C1 也称为 Client Compiler，适用于执行时间短或者对启动性能有要求的程序

C2 也称为 Server Compiler，适用于执行时间长或者对峰值性能有要求的程序

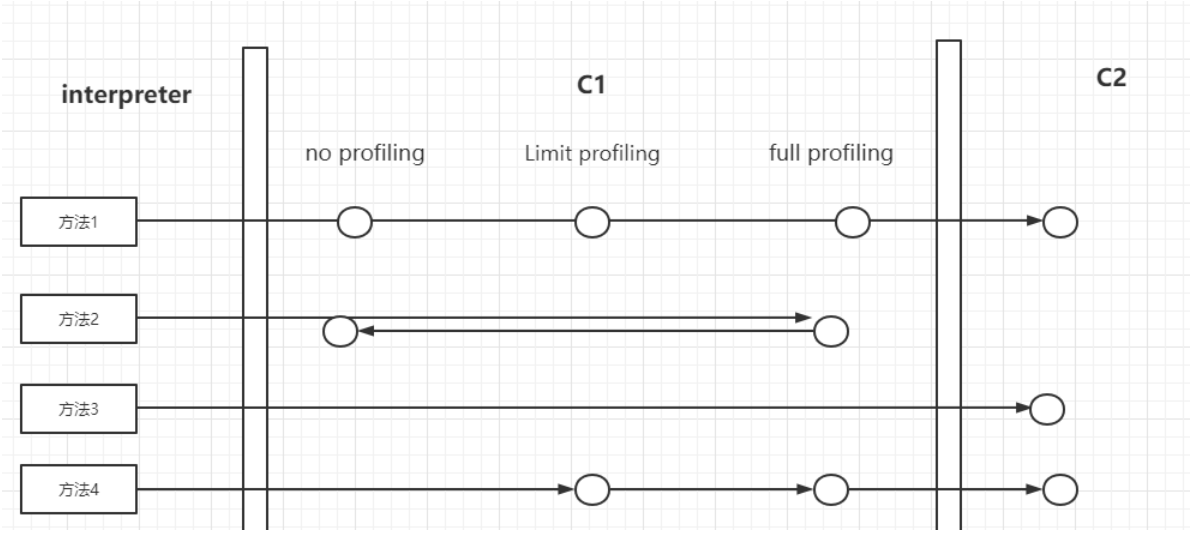
0.解释执行

1.简单的C1编译：仅仅使用我们的C1做一些简单的优化，不会开启Profiling

2.受限的C1编译代码：只会执行我们的方法调用次数以及循环的回边次数（多次执行的循环体）Profiling的C1编译

3.完全C1编译代码：我们Profiling里面所有的代码。也会被C1执行

4.C2编译代码：这个才是优化的级别。
级别越高，我们的应用启动越慢，优化下来开销会越高，同样的，我们的峰值性能也会越高
通常C2 代码的执行效率要比 C1 代码的高出 30% 以上



3、CodeCache

存放 JVM 生成的 native Code 内存空间。
JIT编译（占绝大部分）、JNI等都会编译代码到native code。
    InitialCodeCacheSize –初始代码缓存大小，默认为160K
    ReservedCodeCacheSize –默认最大大小为48MB
    CodeCacheExpansionSize –代码缓存的扩展大小，32KB或64KB

4、方法内联

JVM在运行时将调用次数达到一定阈值的方法调用替换为方法体本身，从而消除调用成本。

方法调用本身是有成本的：栈帧的生成，参数字段的入栈，栈帧的弹出，指令执行。

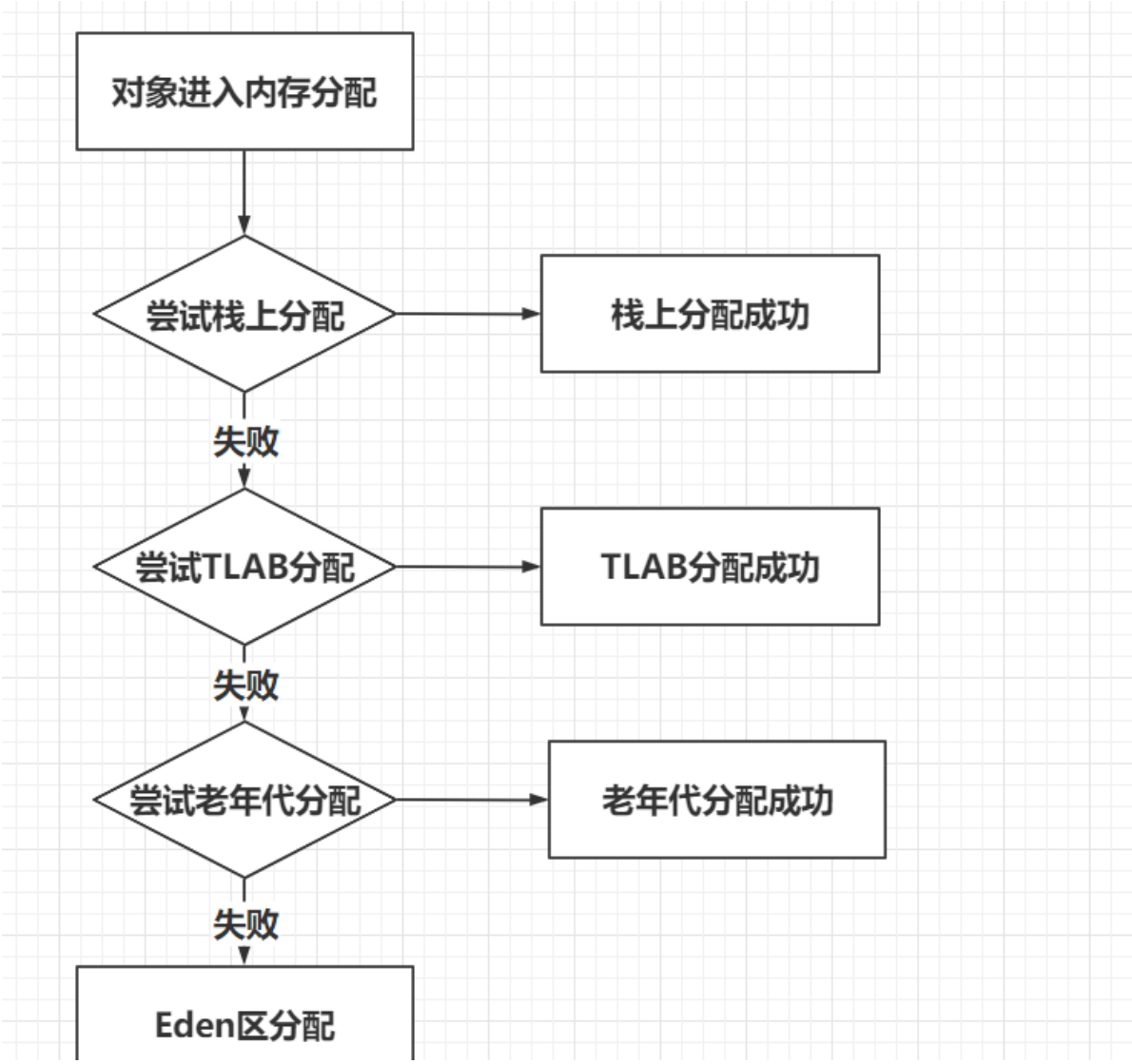
内联条件
    1、热点代码
    2、方法体不能太大
    3、尽量 （static private final ）修饰，直接内联，避免多态

5、逃逸分析

指针逃逸：当变量（或者对象）在方法中分配后，其指针有可能被返回或者被全局引用，这样就会被其他方法或者线程所引用。

通过逃逸分析，检测对象的作用范围，决定对象是否分配到堆内存中。

    1、栈上分配
    2、同步锁消除
    3、标量替换：用不可分割的基础数据类型代替对象标识（局部变量表）



9、工具

1、监控工具

```
1、Jconsole
2、jvisualvm
3、arthas
```

2、内存分析工具

```
1、HeapHero
2、MAT
3、perfma
```

3、日志分析

```
1、GCViewer
2、gceasy
```

10、案列优化

1、高并发的场景