

为Java开疆拓土的 ZGC深度剖析

King老师

- 十余年Java行业经验
- 先后在58、招行等工作
- 主导互联网金融项目的核心架构开发

课程内容

- 1、ZGC中JVM内存布局及设计
- 2、ZGC如何做到10ms的暂停
- 3、ZGC基于指针着色的并发标记算法
- 4、ZGC基于指针着色的并发转移算法
- 5、ZGC整体流程分析及性能对比实战
- 6、大厂项目ZGC的技术运用与挑战
- 7、并发转移算法关键技术之读屏障
- 8、ZGC的触发时机与性能优化



腾讯课堂-图灵课堂

01月26日 晚上20:00

课程安排

Turing College图灵课堂

致敬大师
致敬未来的你

JVM最新技术	
章节	章节名称
1	为Java开疆拓土的ZGC深度剖析
2	让Java性能提升的JIT深度剖析
3	GraalVM:云原生时代的Java虚拟机

上课说明：

课程前置知识：JVM性能调优专题

- 1、课程中简单代码不会手写，一般只会手写核心代码（提高课程效率）
- 2、一个知识点如果大部分同学明白，不会重复讲解，未明白的同学请看视频、笔记、请教同学或加老师QQ
- 3、以上为本次的章节安排，不是课时安排，如果一章内容在一次课内未讲完，则会进行顺延



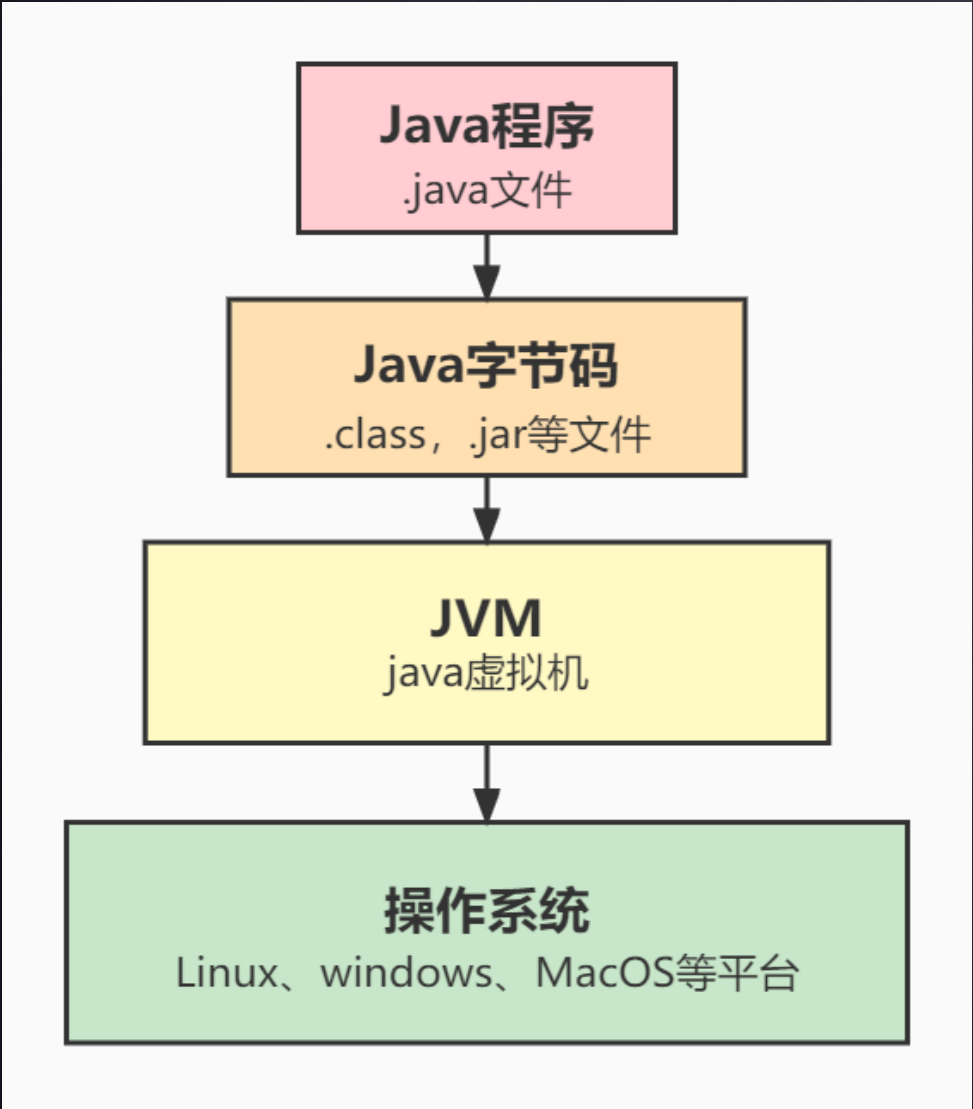
King老师

十三年Java行业经验。

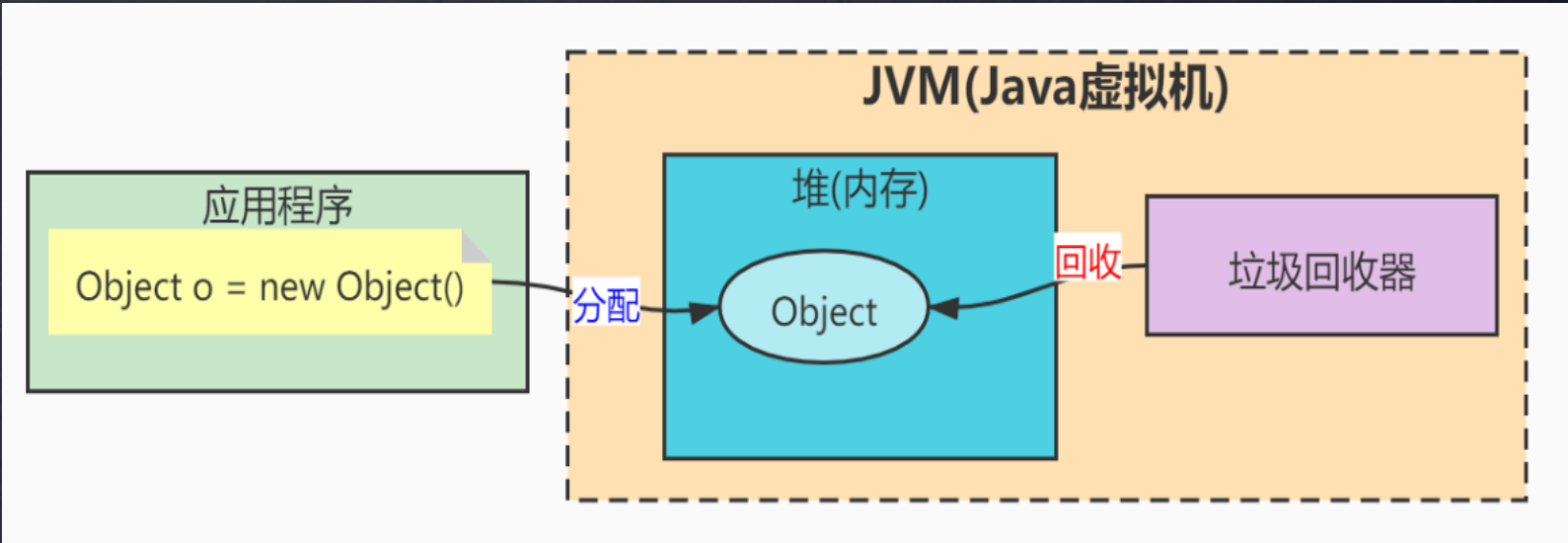
- 资深架构师
- 资深讲师

此处本应有100行介绍，(地方小装不下)

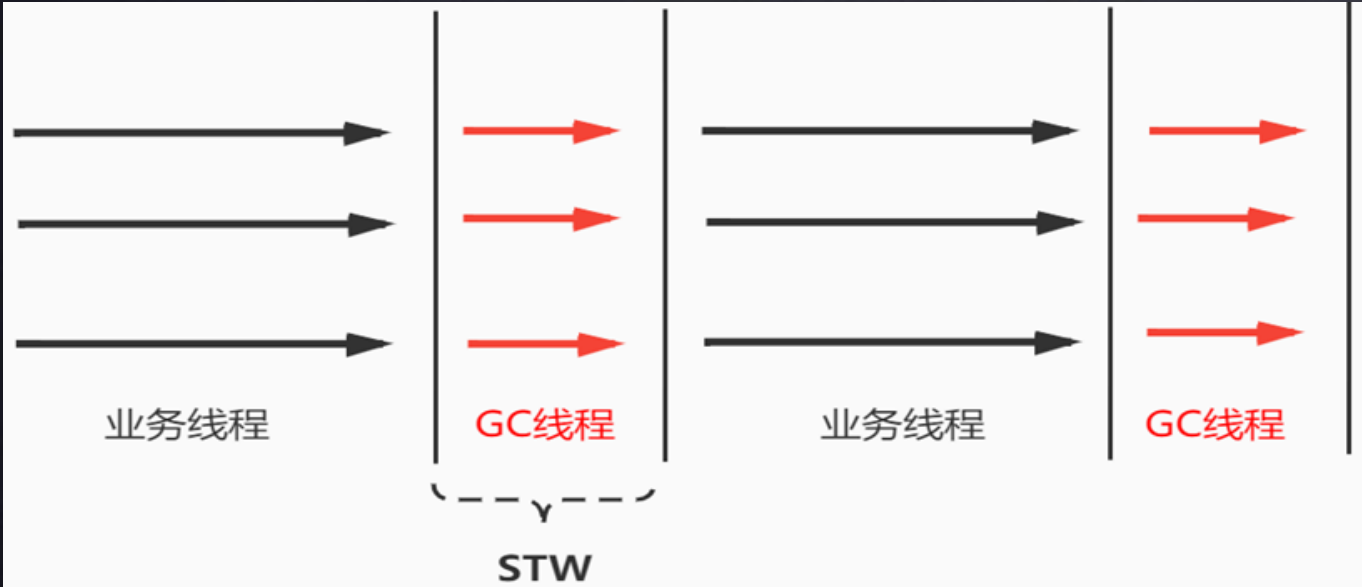
■ JVM: Java虚拟机



■ JVM核心功能：自动化的垃圾回收机制



■ STW: Stop The World



■ Java生态圈已经足够强大，但还不够！

- 手机系统(Android) 显示卡顿（google主导）
- 证券交易系统实时性要求（抢C++市场）
- 大数据平台(HBase集群性能)（58、腾讯、阿里等公司）

■ 单线程

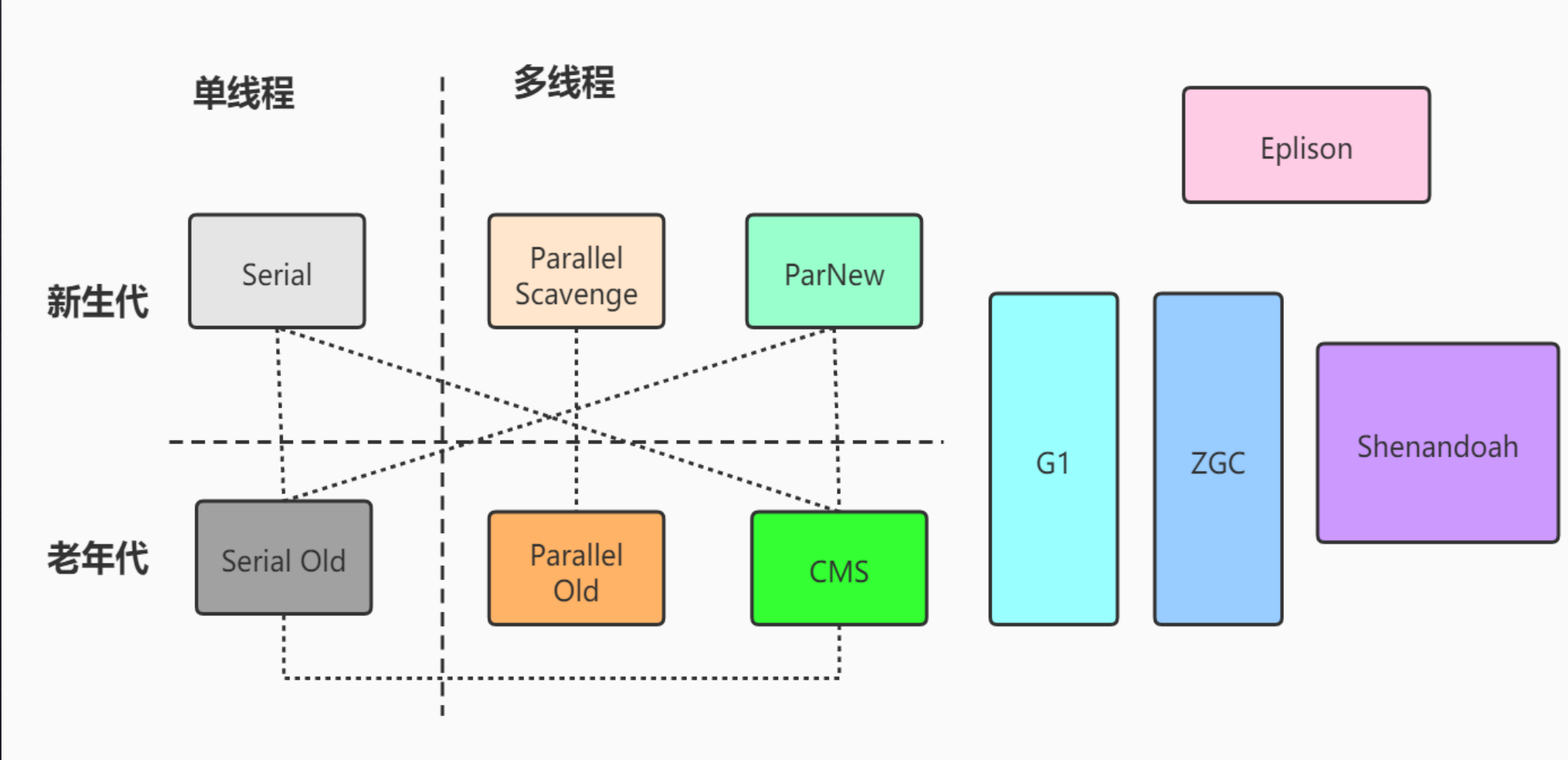
- Serial
- SerialOld

■ 多线程

- ParallelScavenge
- ParallelOld

■ 多线程+并发

- CMS
- G1
- Shenadndoah
- ZGC(STW控制在1ms)



- JDK11中推出的一款低延迟垃圾回收器
- 支持16TB级别的堆
- 停顿时间(STW)不超过1ms，且不会随着堆的大小增加而增加

ZGC

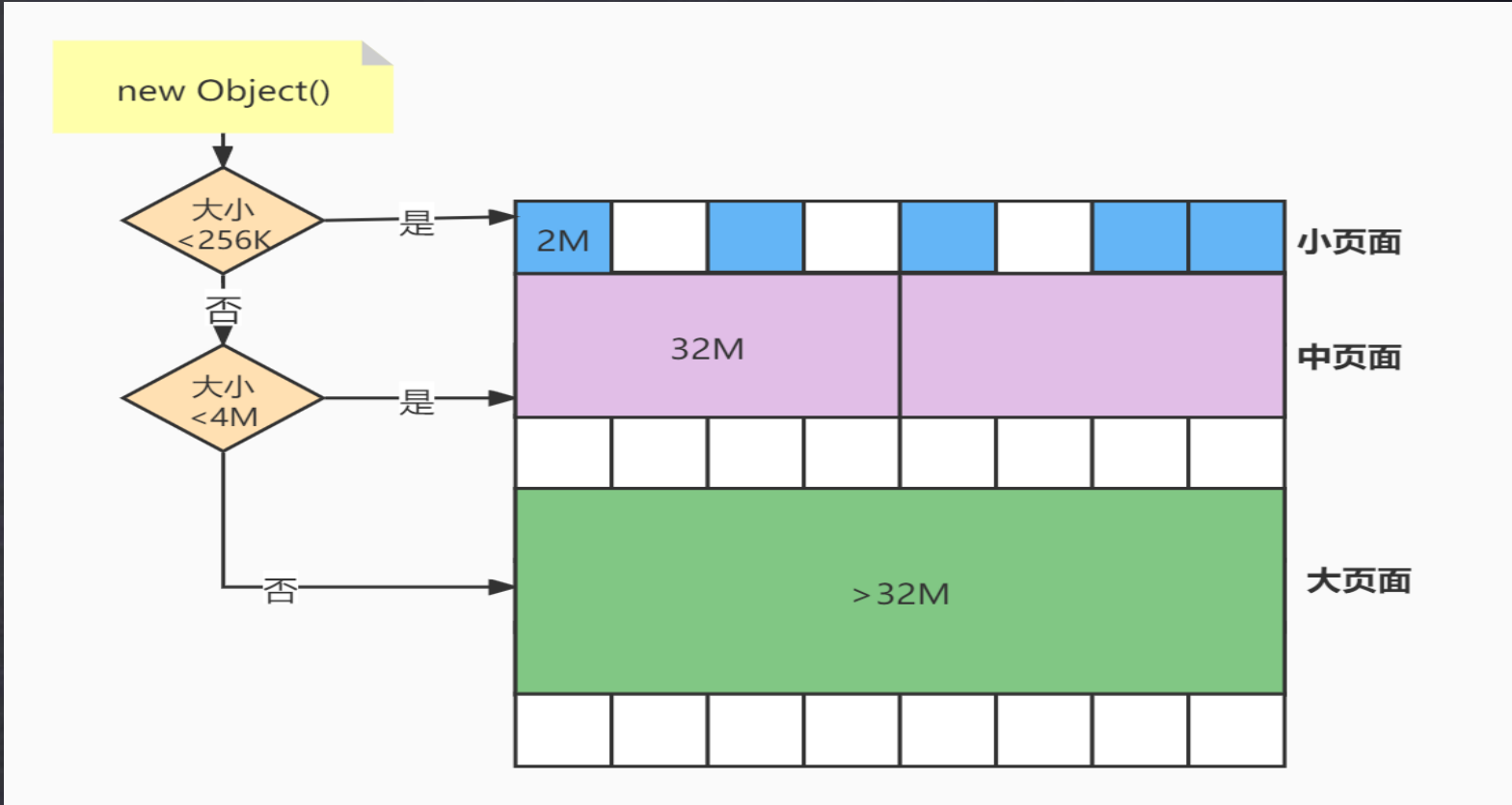
The Z Garbage Collector



- ZGC出现的目标：卷其他语言！

■ 堆空间分页模型（无分代）

- 小页面
- 中页面
- 大页面



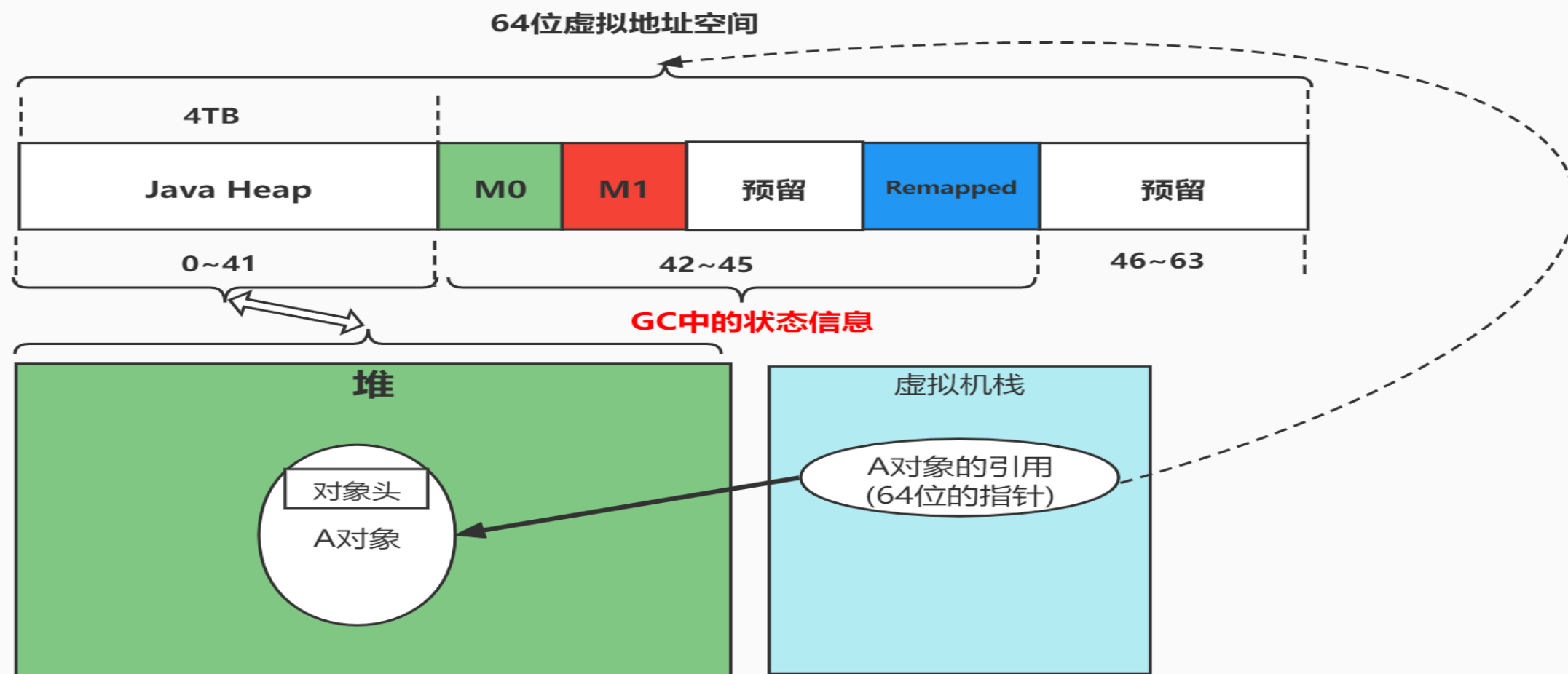
■ 为什么这么设计？

Linux Kernel 2.6引入的标准大页（huge page）

- ZGC支持NUMA(了解即可)
Non-Uniform Memory Access(非统一内存访问)

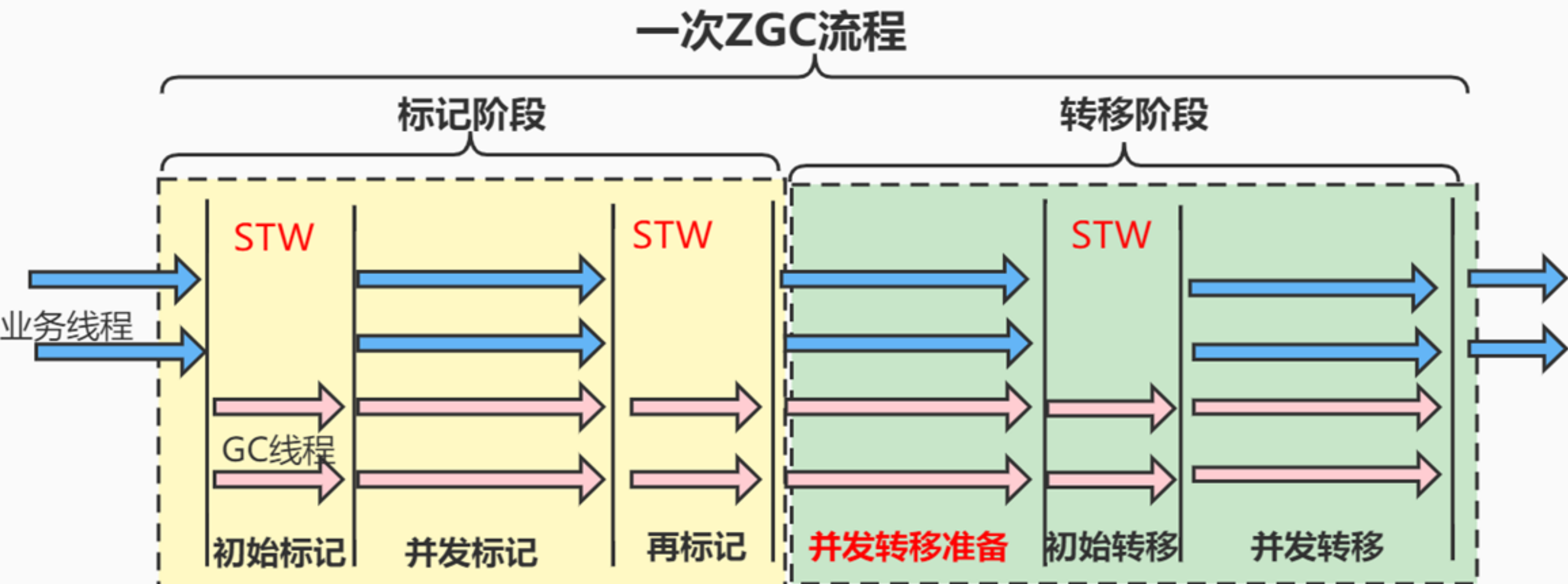
■ 指针着色技术（Color Pointers）

- ZGC只支持64位系统(使用64位指针)
- ZGC中低42位表示使用中的堆空间
- ZGC借几位高位来做GC相关的事情(快速实现垃圾回收中的并发标记、转移和重定位等)



■ 一次ZGC流程

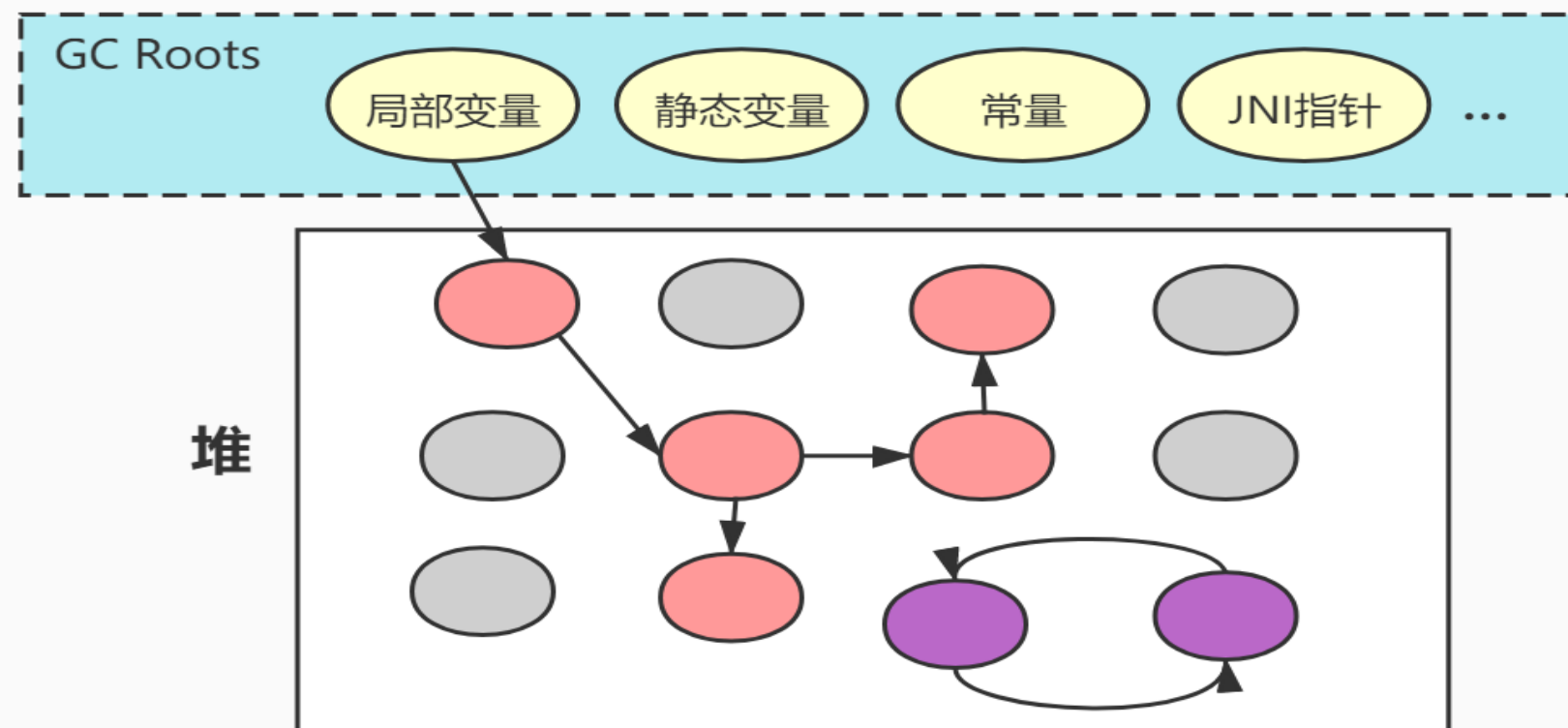
- 标记阶段(标识垃圾)
- 转移阶段(对象复制或移动)



■ 根可达算法

■ 作为GC Roots的对象主要包括下面4种

- 虚拟机栈（栈帧中的本地变量表）
- 方法区中类静态变量
- 方法区中常量
- 本地方法栈中JNI指针



垃圾回收算法

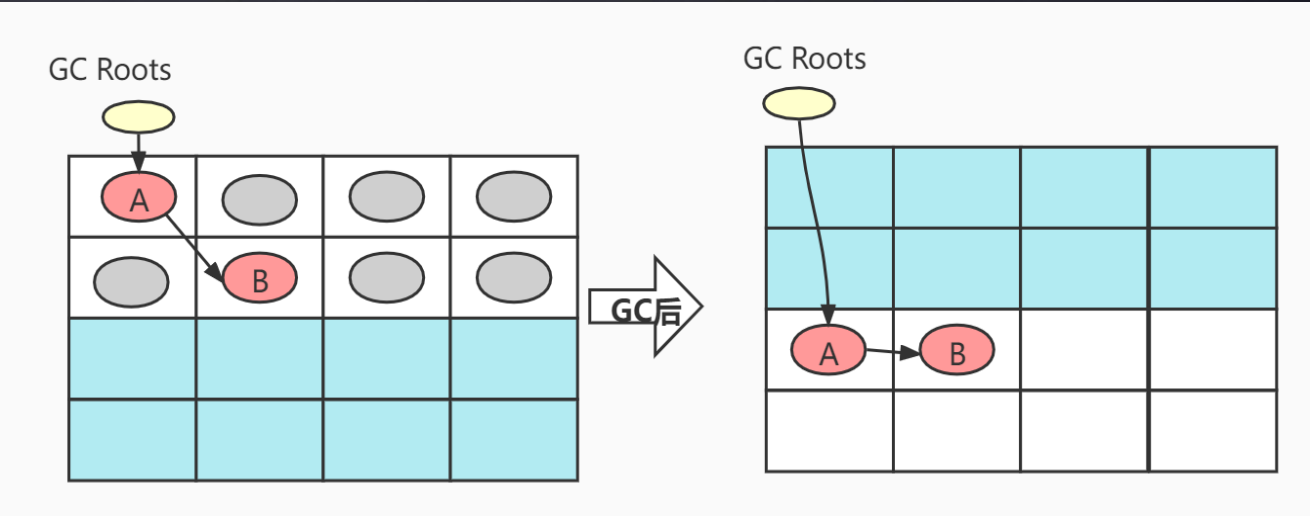
■ ZGC中的 “转移”

- 如果是同一个页面(等同标记整理)
- 如果是不同页面(等同复制算法)

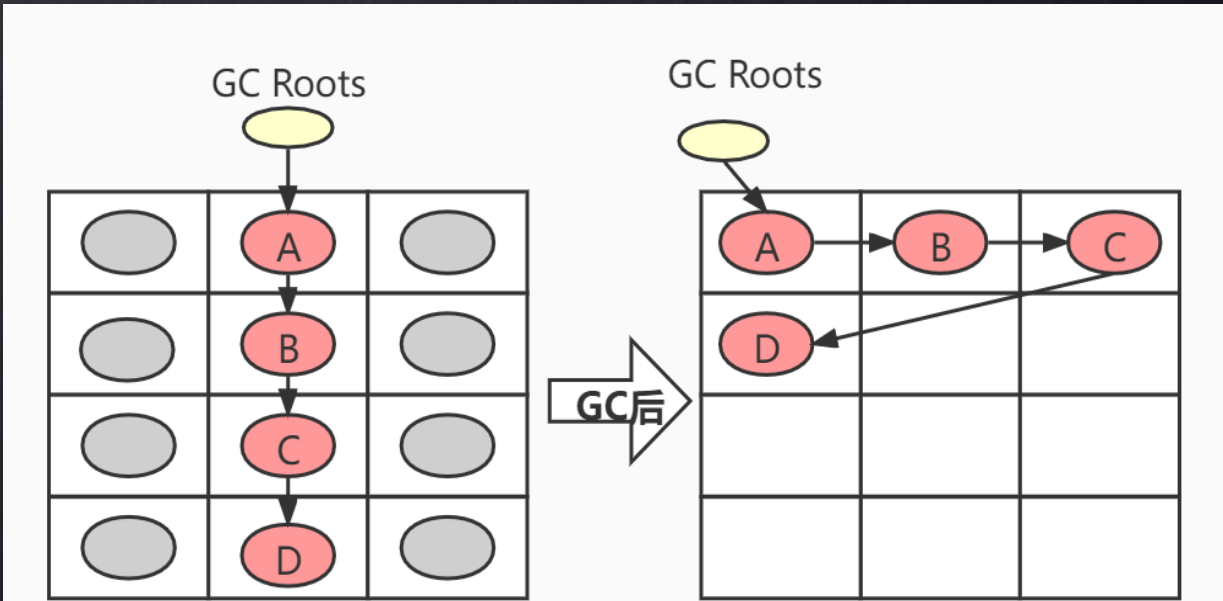
■ ZGC中垃圾回收算法流程

- 标记(mark):从根集合出发，标记活跃对象；此时内存中存在活跃对象和已死亡对象。
- 转移(relocate): 把活跃对象转移（复制）到新的内存上，原来的内存空间可以回收。
- 重定位(remap): 因为对象的内存地址发生了变化，所以所有指向对象老地址的指针都要调整到对象新的地址上。

■ 复制算法



■ 标记整理



ZGC基于指针着色的并发标记算法

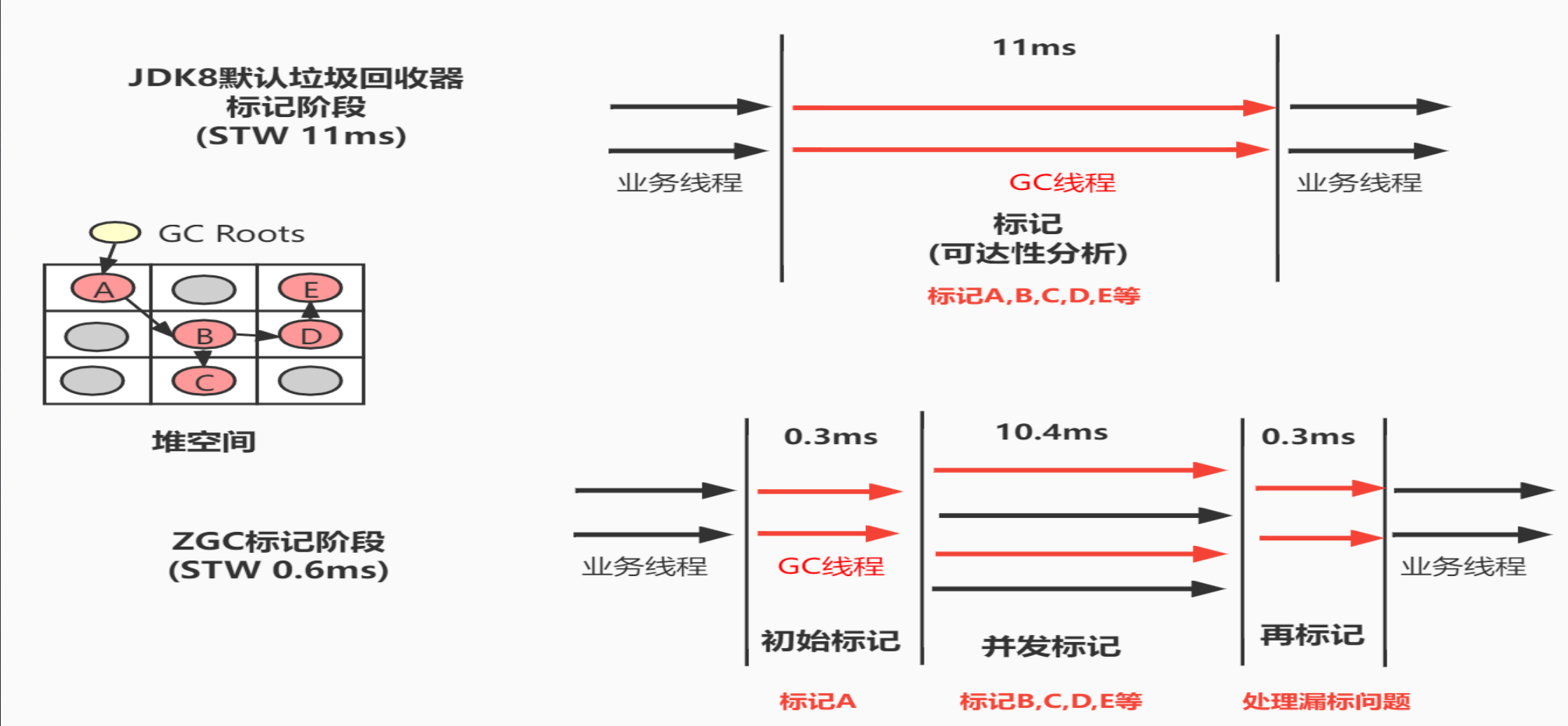
■ ZGC中标记阶段

- 初始标记：从根集合出发，找出根集合直接引用的活跃对象（图中A对象）
- 并发标记：根据初始标记找到的根对象，使用深度优先遍历对象的成员变量进行标记（图中B,C,D,E对象）
- 再标记：并发标记需要解决标记过程中引用关系变化导致的漏标记问题

■ 指针着色技术

区分相邻两次GC中的标记

- M0(mark-0)
- M1(mark-1)

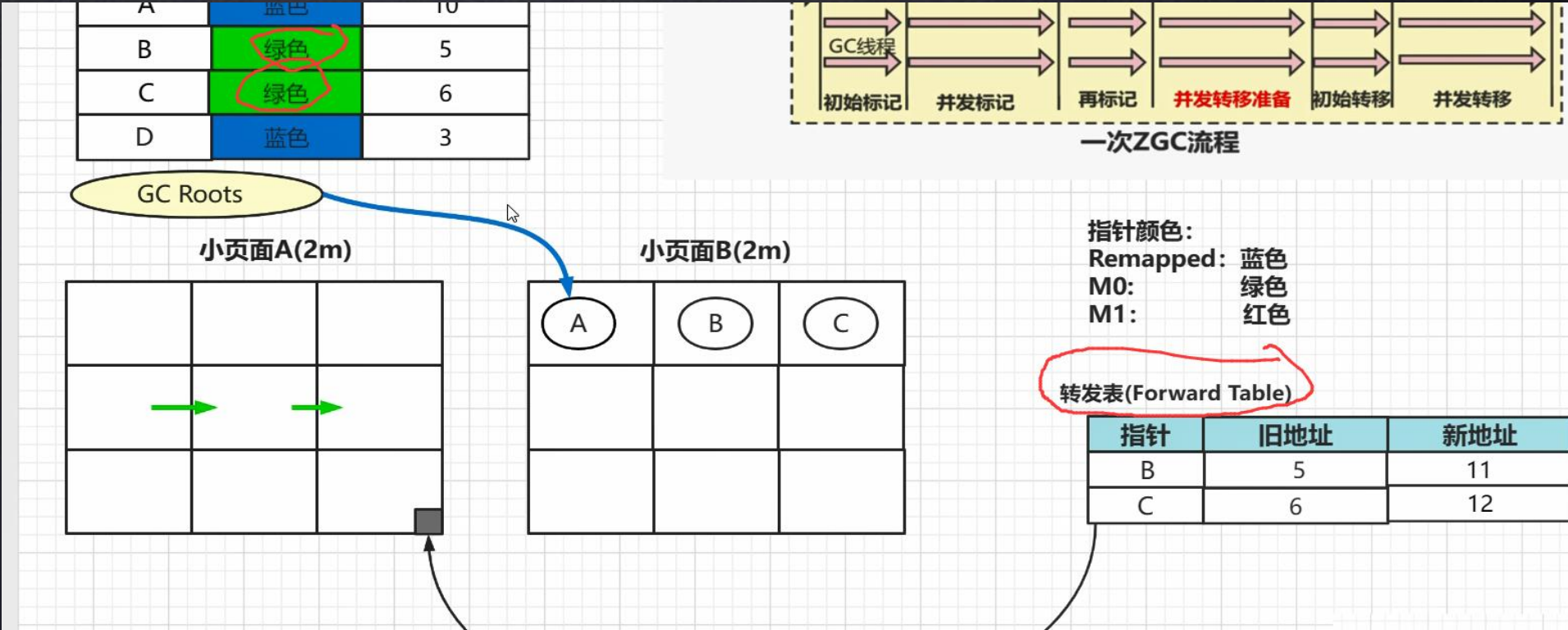


ZGC基于指针着色的并发转移算法

- ZGC的转移阶段
 - 并发转移准备(分析最有价值GC分页<无STW >)
 - 初始转移（转移初始标记的存活对象同时做对象重定位<有STW> ）
 - 并发转移（对转移并发标记的存活对象做转移<无STW>）

- 如何做到并发转移？

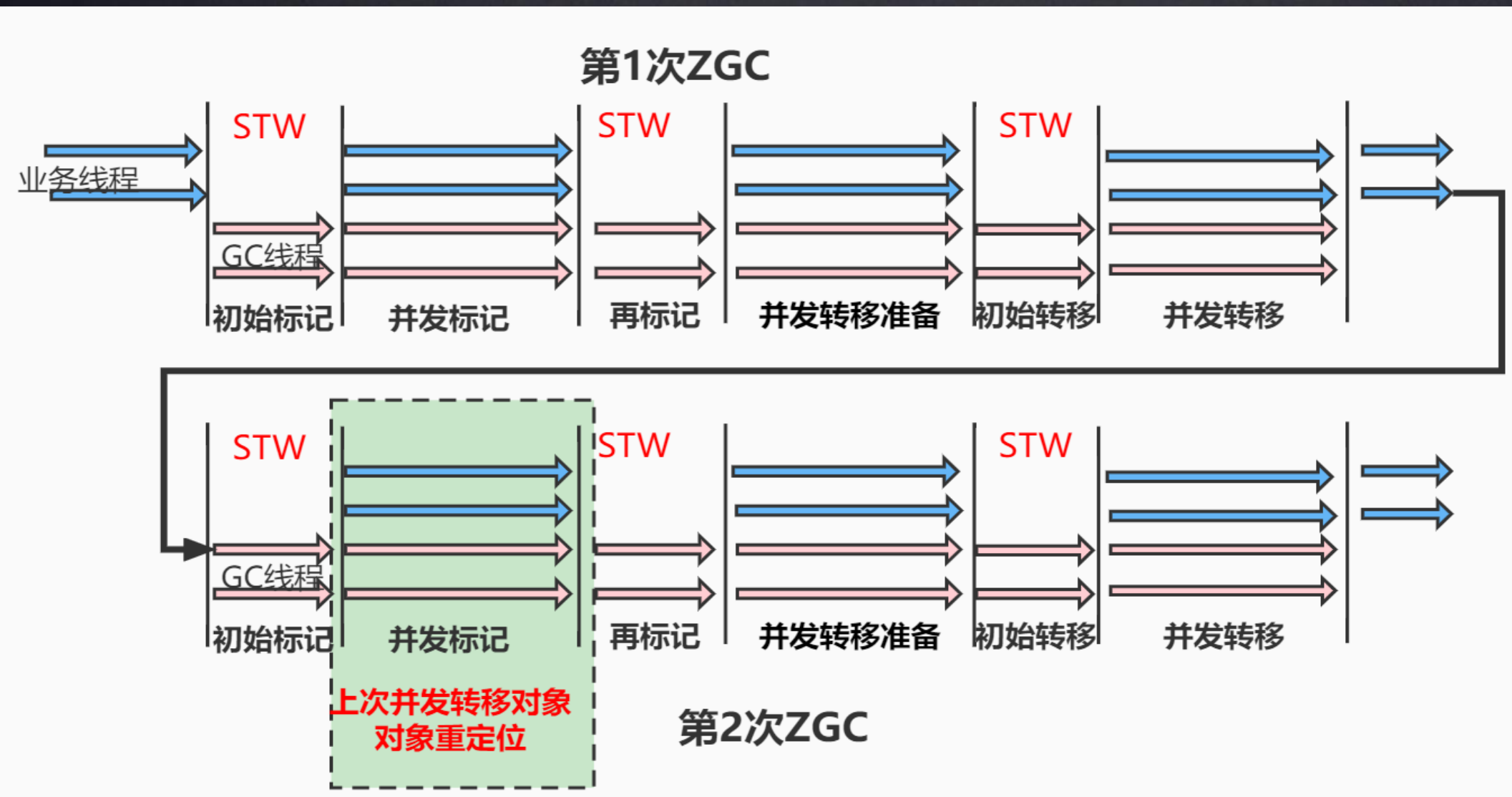
- 转发表(类似于HashMap)
- 对象转移和插转发表做原子操作



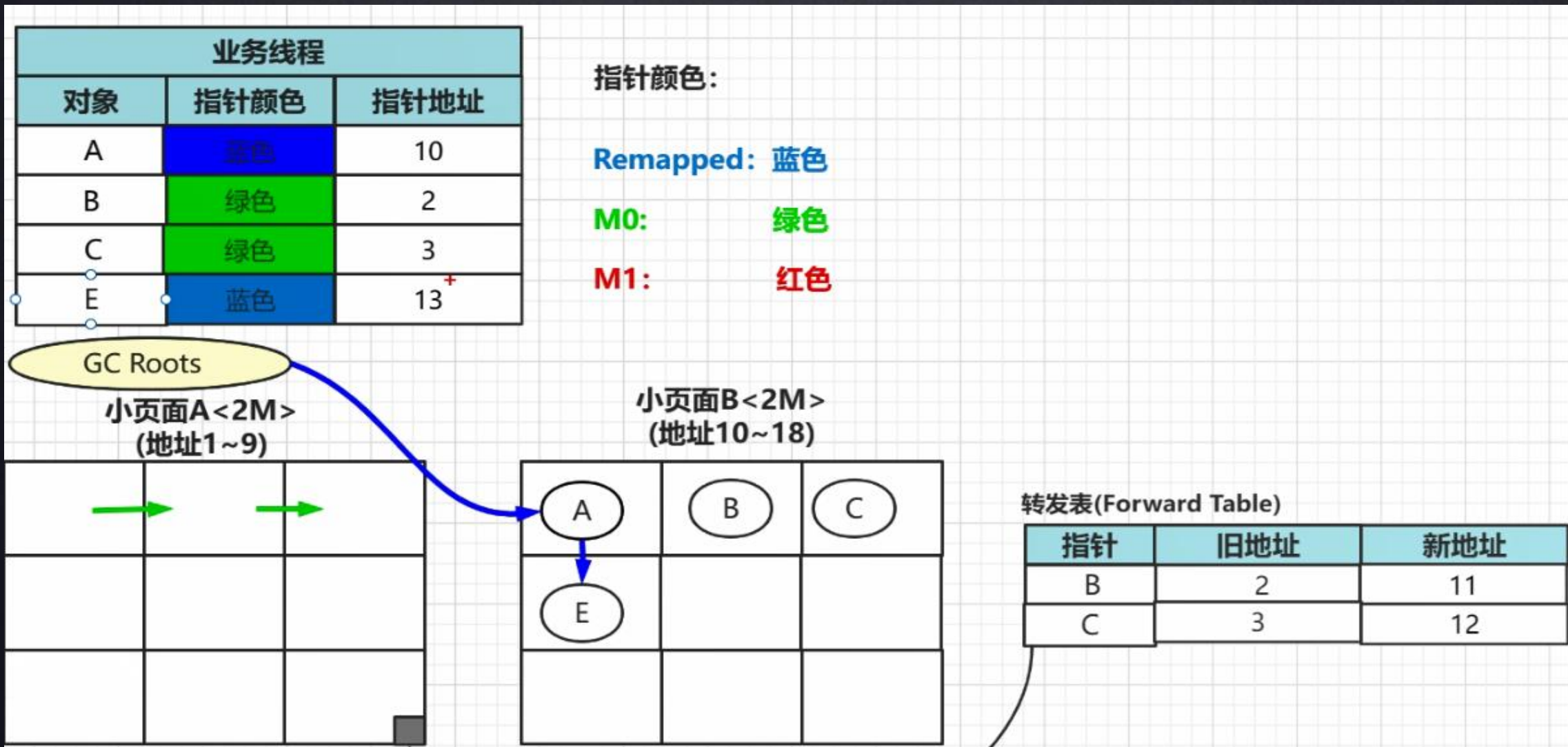
ZGC基于指针着色的重定位算法

- 并发标记对象的重定位
- 下次GC中的并发标记（同时做上次并发标记对象的重定位）
- 技术上：指针着色中M0和M1区分

在两次GC中间业务线程
如何访问没有做完重定位的对象？



- 涉及对象：并发转移但还没做对象重定位的对象 (B、C)
- 触发时机：在两次GC之间业务线程访问这样的对象 (B、C)
- 读屏障 (Load Barrier)：对象重定位+删除转发表记录（两个一起做原子操作）
- 为什么要做读屏障：对象 (B、C) 已经被移动但指针未修正，应用程序访问到旧地址出错



读屏障是JVM向应用代码插入一小段代码的技术

1. 当从堆中读引用时，就需要加入一个Load Barrier(读屏障)
2. 判断当前指针是否Bad Color(不是本次GC的Mark颜色)
3. 修正指针：对象重定位+删除转发表记录

```
public class ZObject {  
    ZObject instance = null;  
    int i=13;  
    public void ReadBarrier() {  
        ZObject A = new ZObject();  
        A.instance= new ZObject();//A->B  
        System.gc();//todo 这里发生一次ZGC  
        ZObject D = A.instance; //这里需不需要加读屏障？（1需要，2不需要）  
        ZObject E = D;//这里需不需要加读屏障？（1需要，2不需要）  
        D.hashCode();//这里需不需要加读屏障？（1需要，2不需要）  
        int j = D.i;//这里需不需要加读屏障？（1需要，2不需要）  
    }  
}
```

■ 读屏障的影响

- 不是涉及的代码都要触发
- 吞吐量降低（官方测试多4%开销）

■ ZGC常见触发时机

- 基于分配速率的自适应算法(最主要): `ZAllocationSpikeTolerance`控制
- 基于固定时间间隔: `ZCollectionInterval`参数控制
- 主动触发规则: `Zproactive`控制
- 启动预热: 关键词warmup

■ ZGC参数设优化建议

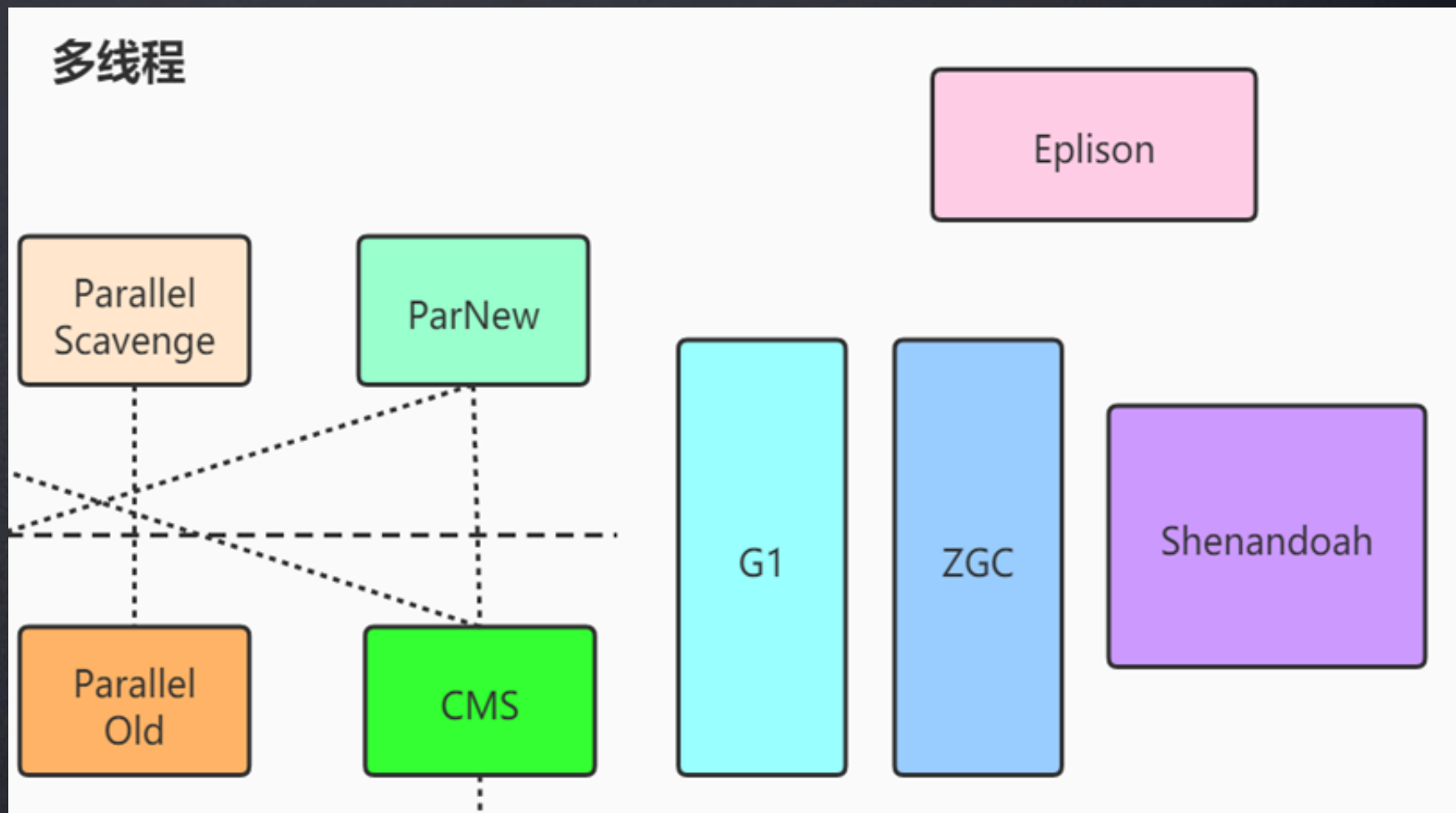
- 堆大小: (GC日志中看到Allocation Stall, 通常可以认为堆空间偏小或者ConcGCThreads数偏小)
- `ParallelGCThreads`: 是设置 STW 任务的 GC 线程数目, 默认为 CPU 个数的 60%
- `ConcGCThreads`: 是并发阶段 GC 线程的数目, 默认为 CPU 个数的 12.5%

■ 案例代码介绍

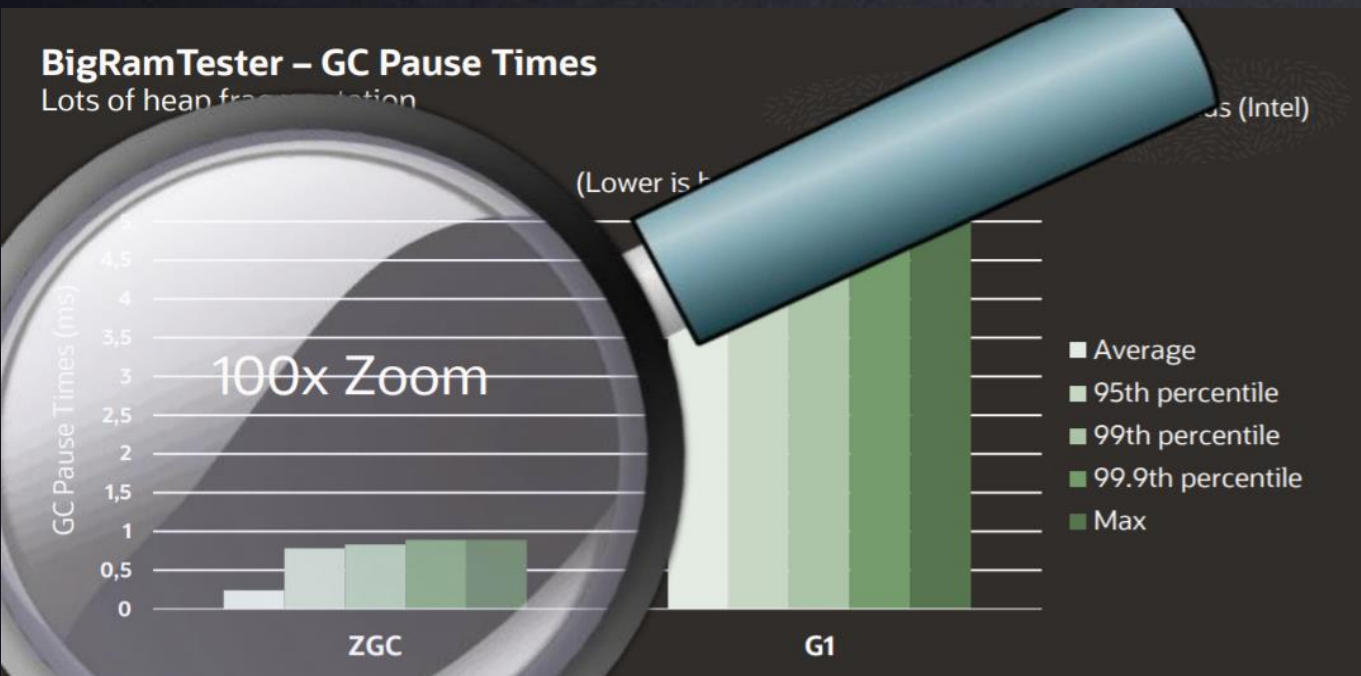
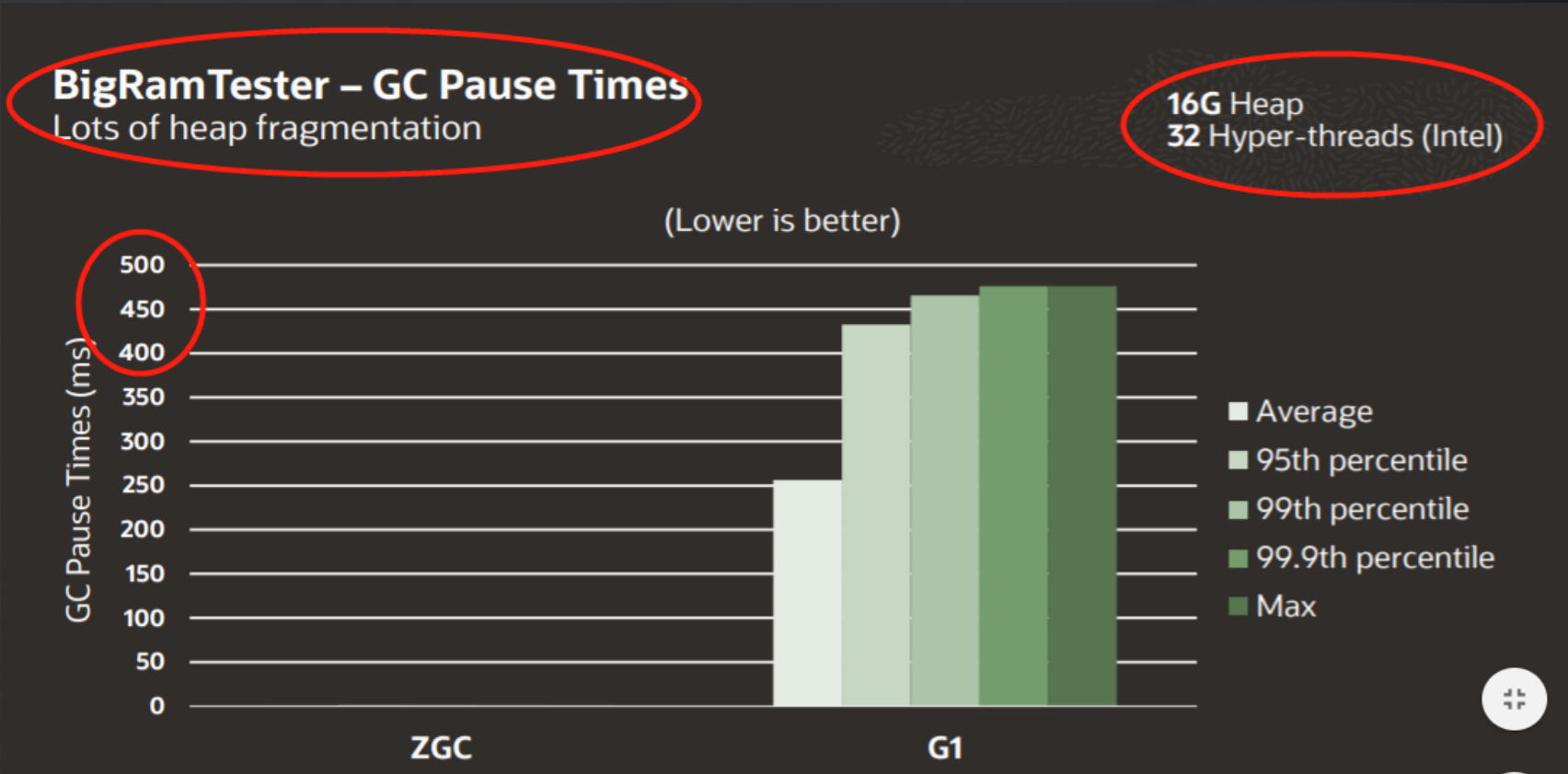
■ 对比PS、G1、ZGC的STW

■ ZGC大厂的运用

- 阿里（借鉴ZGC优化自己JVM）
- 美团（规则平台等）
- 58（Hbase平台）
- 腾讯（在线交互、竞价广告、量化交易等）
- 华为（毕昇JDK、大数据项目）



ZGC与G1测试对比



让Java性能提升的 JIT深度剖析

King老师

- 十余年Java行业经验
- 先后在58、招行等工作
- 主导互联网金融项目的核心架构开发

课程内容

- 1、Java的解释执行与JIT(即时编译器)
- 2、JIT历史发展与分层编译
- 3、JIT优化技术之方法内联
- 4、JIT优化技术之锁消除
- 5、JIT优化技术之逃逸分析
- 6、JIT优化技术之标量替换与栈上分配



腾讯课堂-图灵课堂

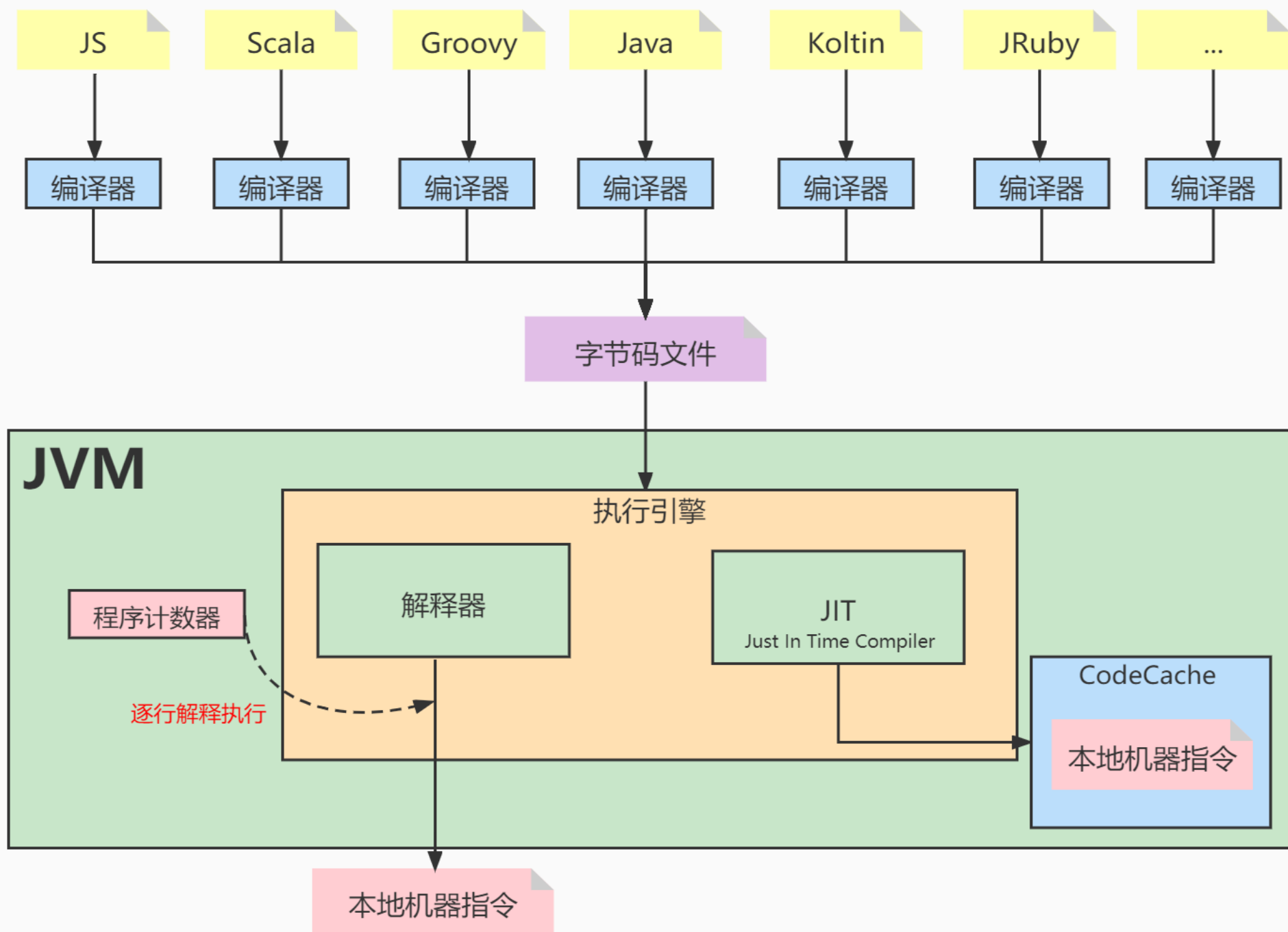
02月09日 晚上20:00

■ JVM的语言无关性

■ 解释器与JIT(即时编译器)

■ 为什么会出现JIT

➤ Java被吐槽“慢”！



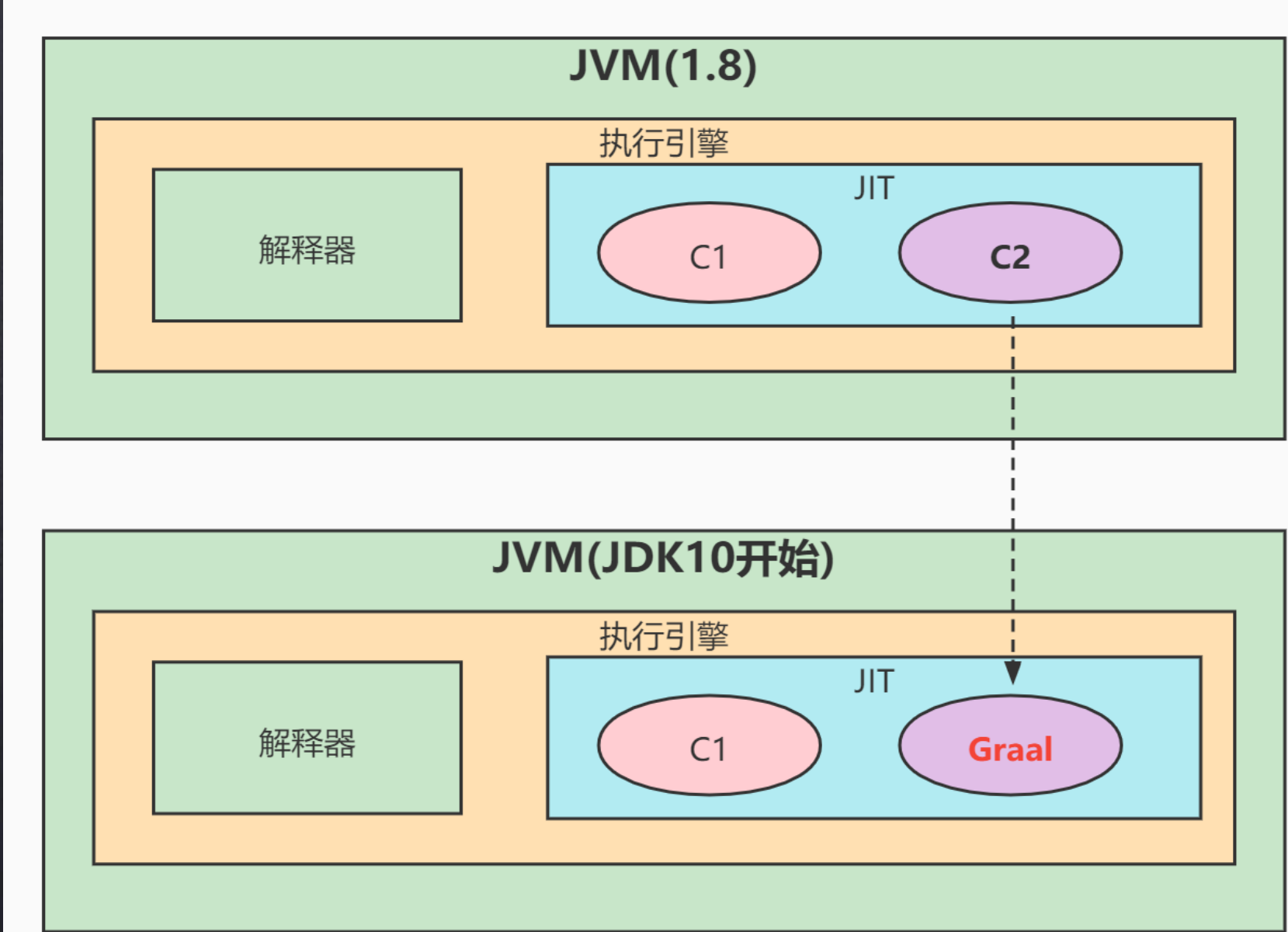
■ JDK10之前

- C1（只做编译，几乎不做优化）
- C2（为长期运行的应用做性能调优）
- < C2代码已超级复杂，无人能维护！ >

■ JDK10开始

- C1
- Graal（Java编写，下章节详细讲）

■ 分层编译(了解即可!)

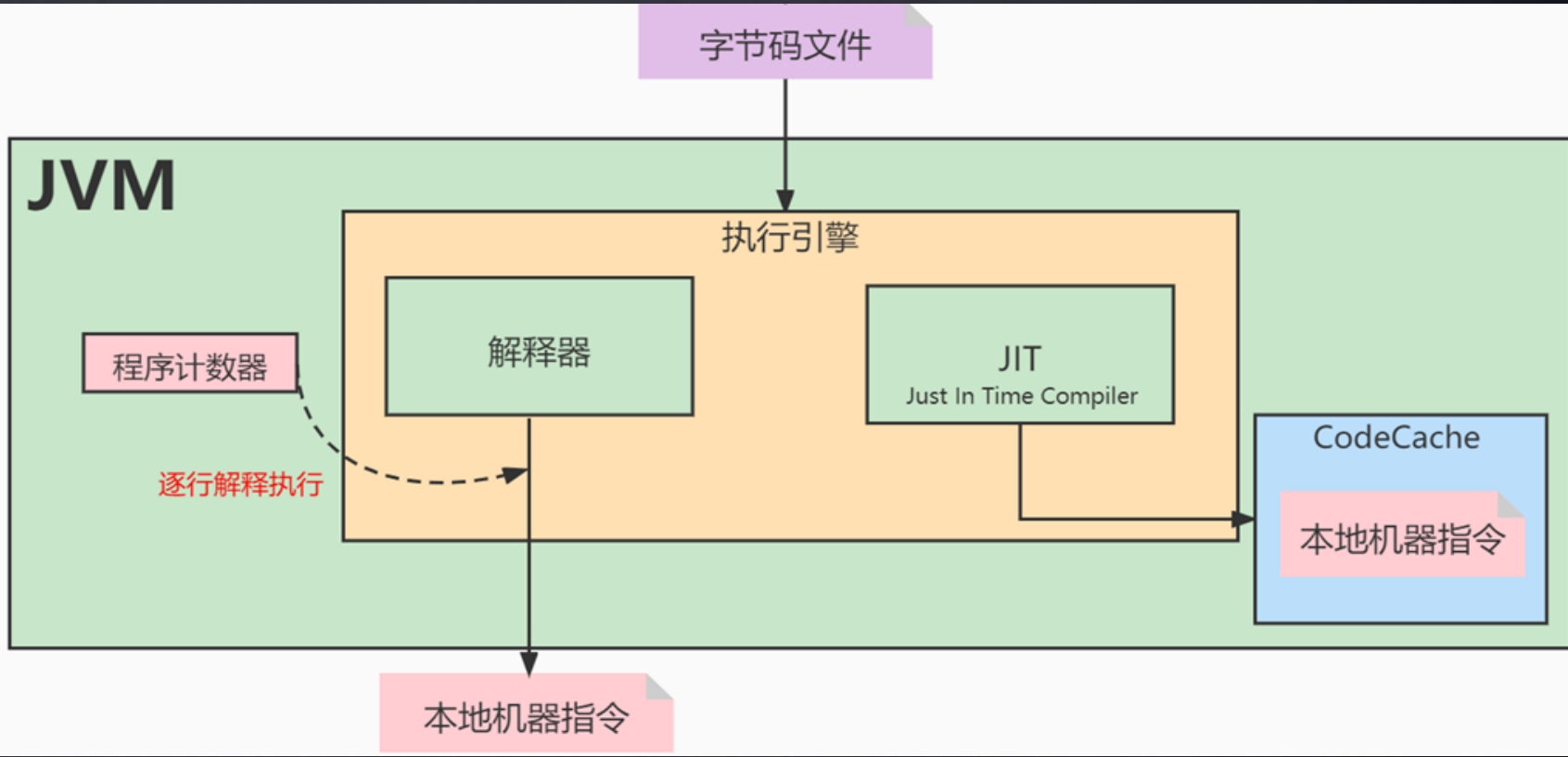


Hotspot

- 热点探测技术
- CodeCache (-XX:ReservedCodeCacheSize)

热点探测技术(以下条件满足一个即可)

- 方法调用计数器
服务端模式默认10000
- 回边计数器
服务端模式默认10700



■ 方法内联

方法内联的优化行为就是把目标方法的代码复制到发起调用的方法之中，避免发生真实的方法调用。

- 热点探测技术触发
- 方法体大小受限制
- 使用方法内联提升性能

■ 锁消除

- 逃逸分析技术
- 标量替换
- 栈上分配

