

JVM内存管理深度剖析

- 1.快速掌握JVM内存管理
- 2.如何通过栈帧分析字节码的运行过程
- 3.内存可视化工具之HSDB大揭秘
- 4.架构师角度认识内存溢出

King老师

前58同城、招商银行工程师

6月7日 20:00

VIP课程



课程安排



享学课堂移动互联网开发——JVM课程表	
课次序号	章节名称
1	从底层深入理解运行时数据区
2	深入理解对象与垃圾回收机制
注意：为了保证学员的学习效果以及内容的深度，上课进度会根据实际情况有所变动	

上课说明：

- 1、首次出现的知识如需要进行编码，一般会进行手写，以后再出现则可能会事先准备好或者进行拷贝。
- 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Java语言高级特性的章节安排，不代表上课次数，如果一章内容在一次课内未讲完，则会顺延到下次课继续讲解。
- 4、一般会遵循 基础入门→初步应用→高级学习路径
- 5、课程预备知识，请提前学习或者复习好：Java语言的基础语法。

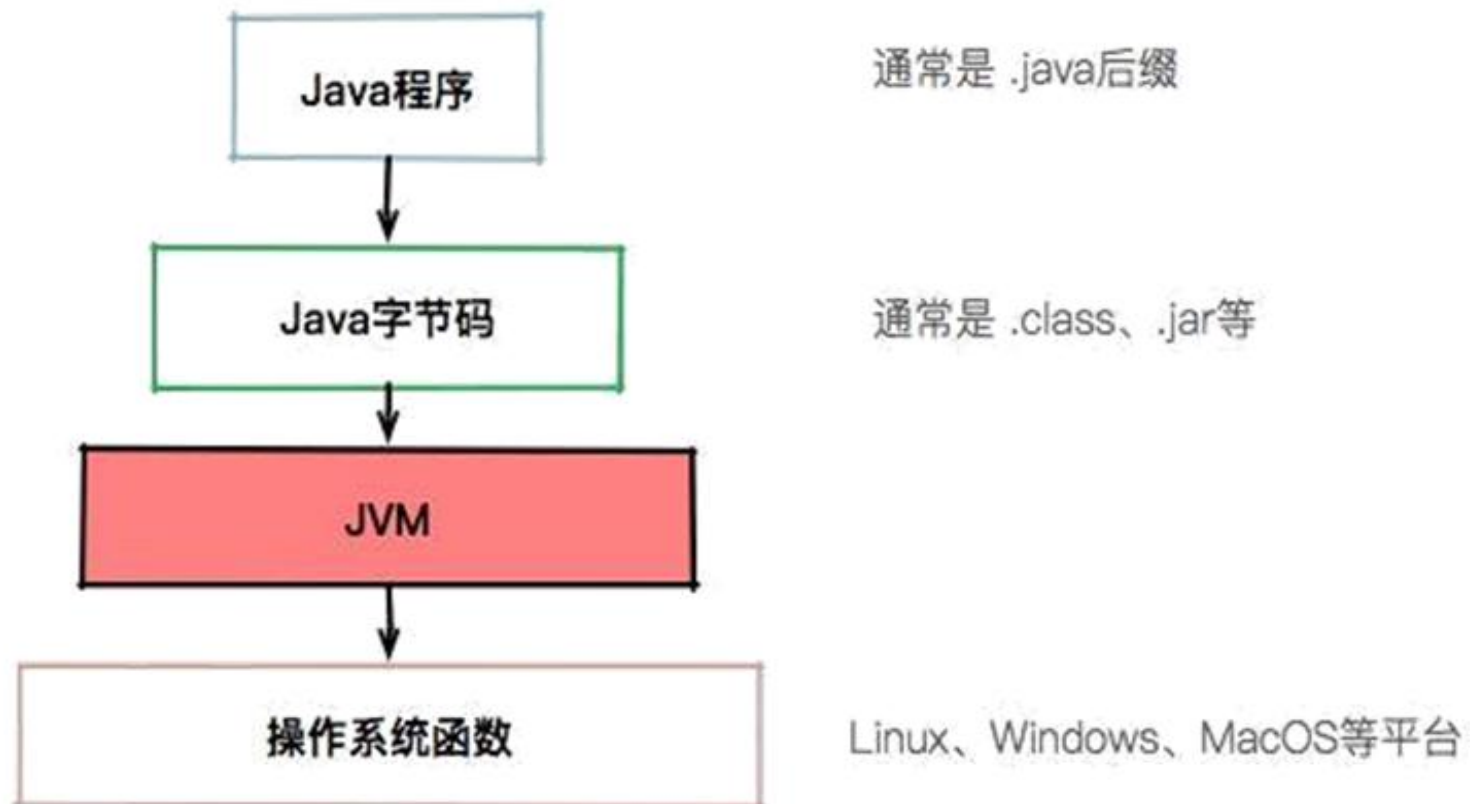
JVM与操作系统的关系



■ Java Virtual Machine

■ 翻译

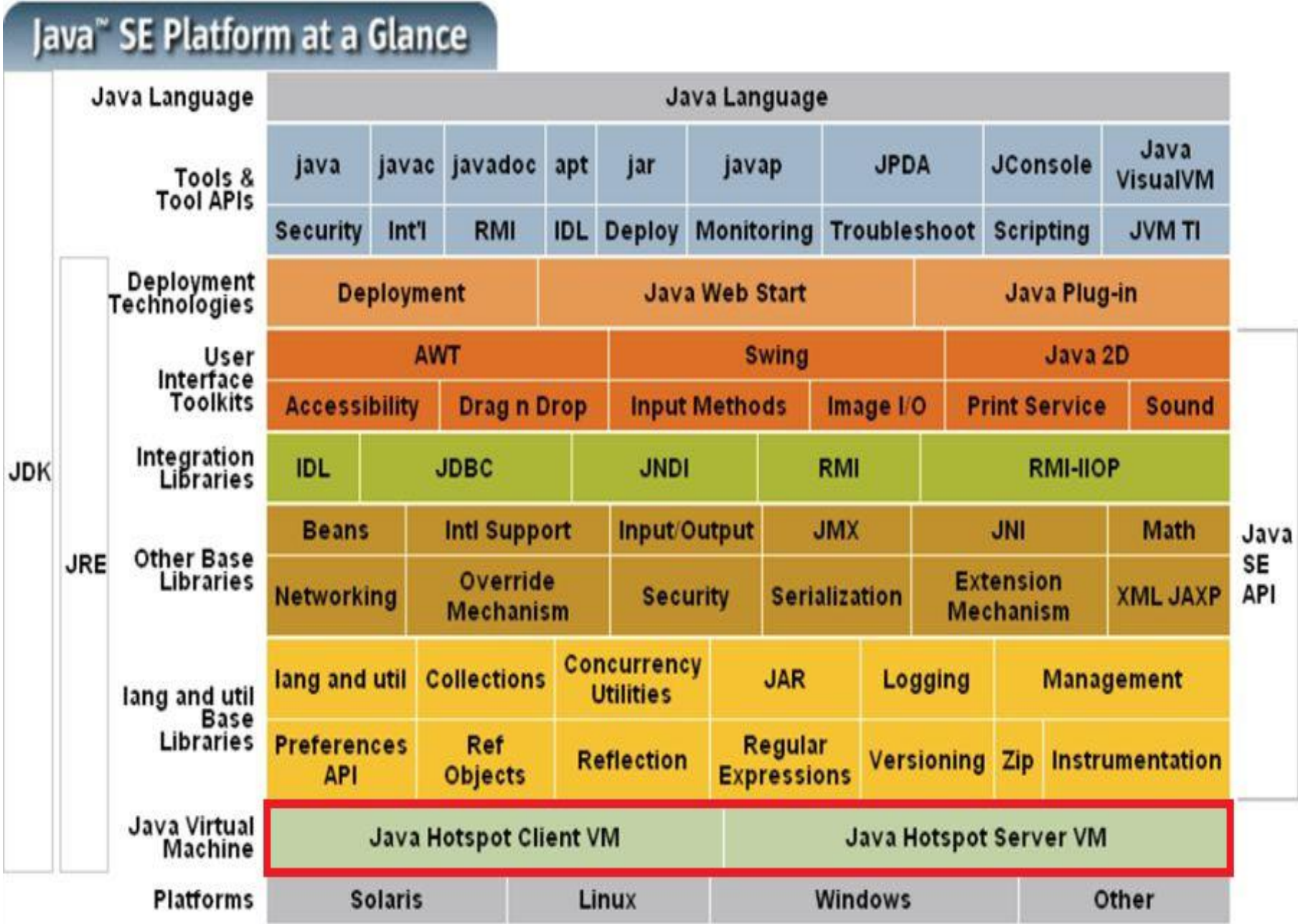
■ 从跨平台到跨语言





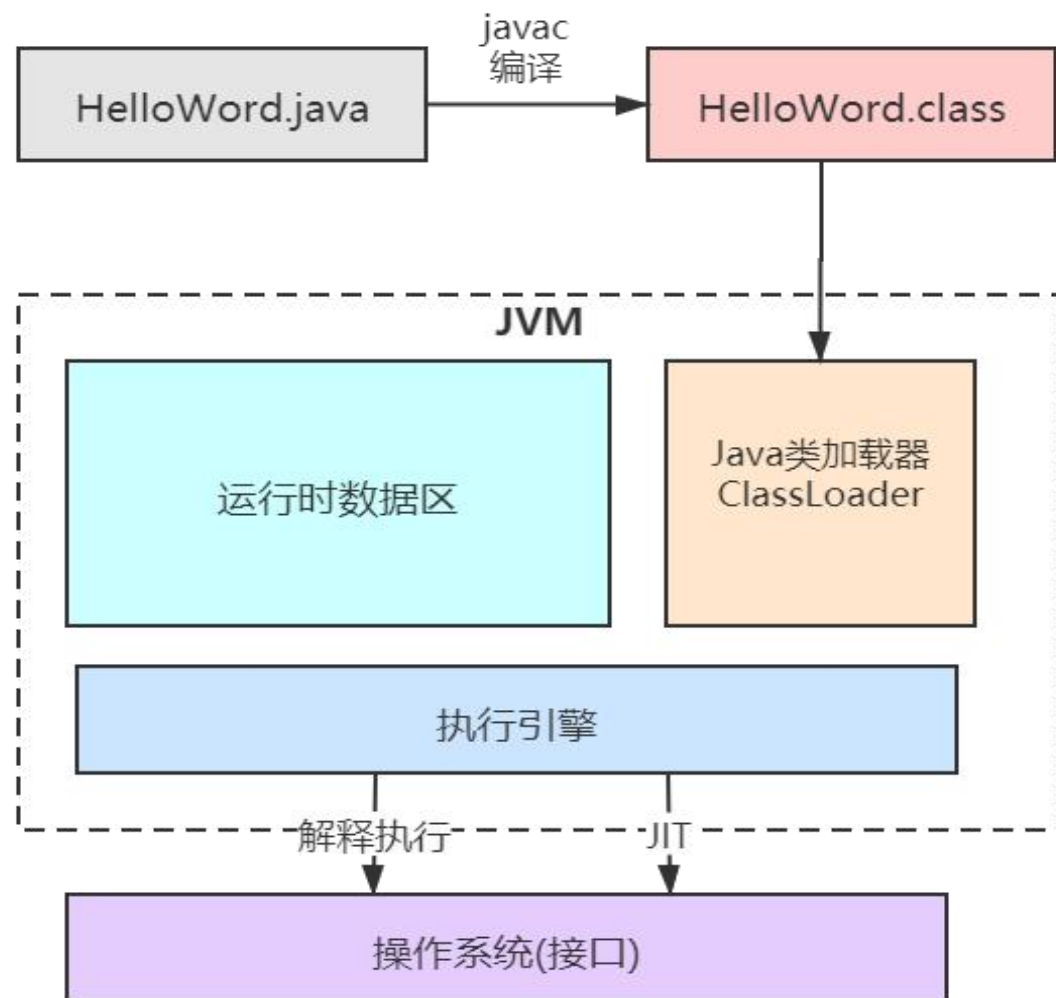
Java SE体系架构

- JVM只是一个翻译
- JRE提供了基础类库
- JDK提供了工具



■ JVM的运行过程

■ 本次课程的重点



运行时数据区域

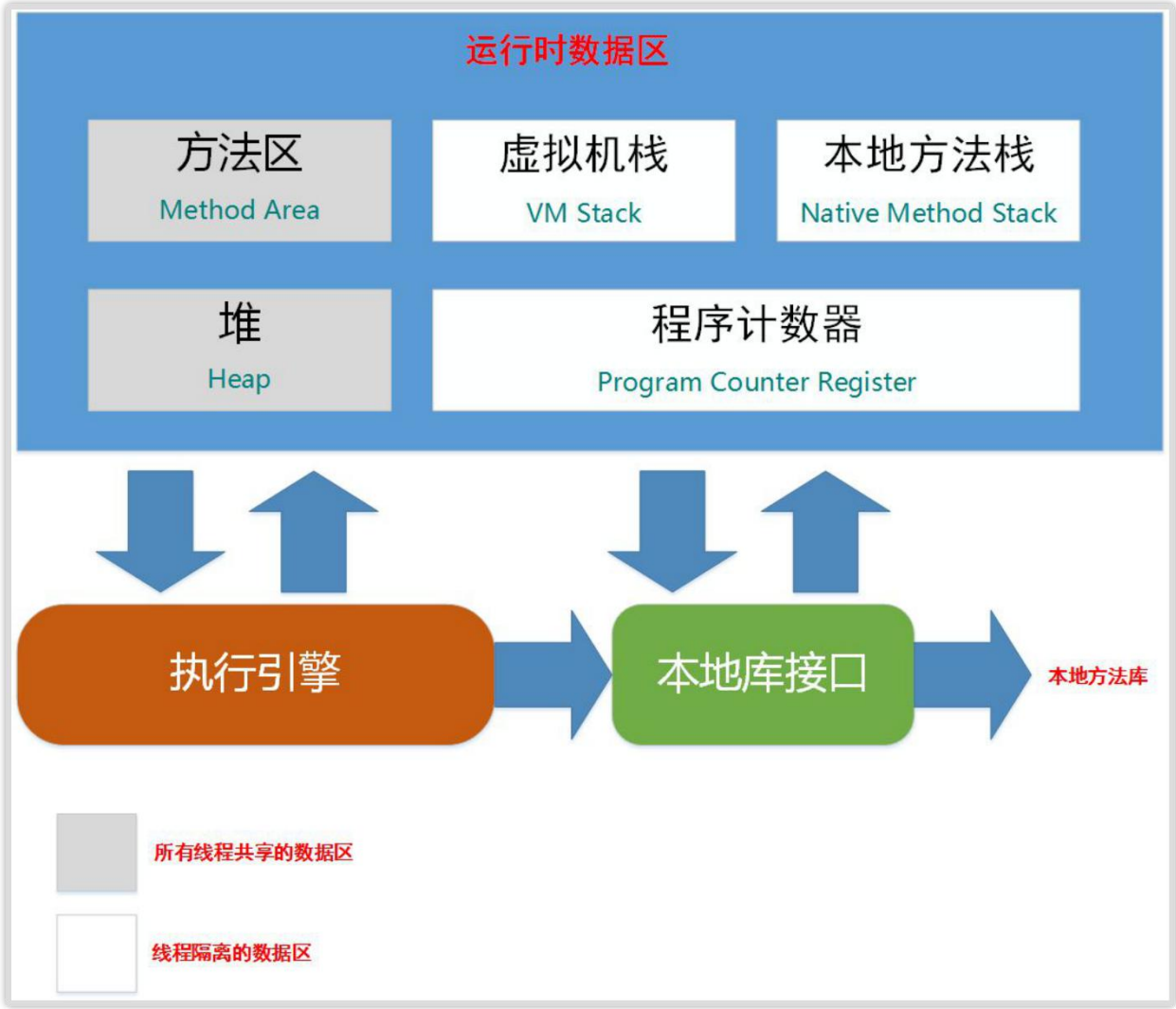


定义

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域

类型

程序计数器、虚拟机栈、本地方法栈、Java堆、方法区（运行时常量池）、直接内存



JAVA方法运行的内存区域



■ 程序计数器

指向当前线程正在执行的字节码指令的地址

■ 虚拟机栈

存储当前线程运行方法所需的数据，指令、返回地址

◆ 栈帧

- 局部变量表
- 操作数栈
- 动态连接
- 完成出口

◆ 大小限制 -Xss





```
/**
 * @author King老师
 */
public class Person {

    public int work(){
        int x =1;
        int y =2;
        int z =(x+y)*10;
        return z;
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.work();
    }
}
```

work()的字节码

操作码 (助记符)

操作描述

iconst_1	将 <u>int</u> 型 1 入操作数栈
istore_1	将操作数栈中栈顶 <u>int</u> 型数值，存入局部变量表（下标为 1 的位置）
iconst_2	将 <u>int</u> 型 2 入操作数栈
istore_2	将操作数栈中栈顶 <u>int</u> 型数值，存入局部变量表（下标为 2 的位置）
iload_1	将局部变量表中下标为 1 的 <u>int</u> 型数据入栈
iload_2	将局部变量表中下标为 2 的 <u>int</u> 型数据入栈
iadd	1)将栈顶两 <u>int</u> 型数值出栈 2)相加 3)并将结果压入操作数栈
bipush 10	10 的值扩展成 <u>int</u> 值入操作数栈
imul	1)将栈顶两 <u>int</u> 型数值出栈 2)相加相乘 3)并将结果压入操作数栈
istore_3	将操作数栈中栈顶 <u>int</u> 型数值，存入局部变量表（下标为 3 的位置）
iload_3	将局部变量表中下标为 3 的 <u>int</u> 型数据入栈

本地(native)方法运行的内存区域



■ 本地方法栈

本地方法栈保存的是native方法的信息

- 当一个JVM创建的线程调用native方法后，JVM不再为其在虚拟机栈中创建栈帧，JVM只是简单地动态链接并直接调用native方法

虚拟机规范无强制规定，各版本虚拟机自由实现

HotSpot直接把本地方法栈和虚拟机栈合二为一



■ 方法区

- ✓ 类信息
- ✓ 常量
- ✓ 静态变量
- ✓ 即时编译期编译后的代码

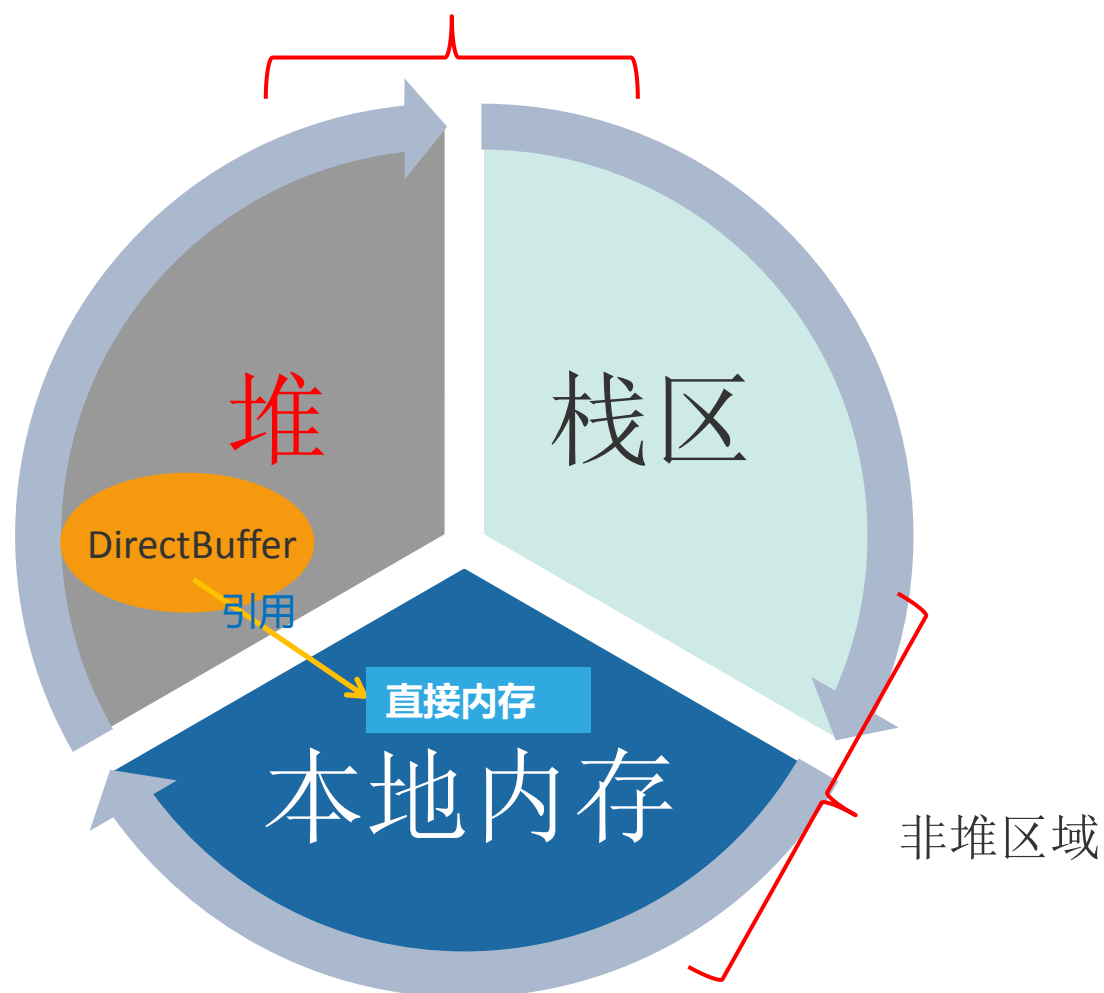
■ Java堆

- ✓ 对象实例（几乎所有）
- ✓ 数组

■ Java堆的大小参数设置

- Xmx 堆区内存可被分配的最大上限
- Xms 堆区内存初始内存分配的大小

JVM内存区域模型

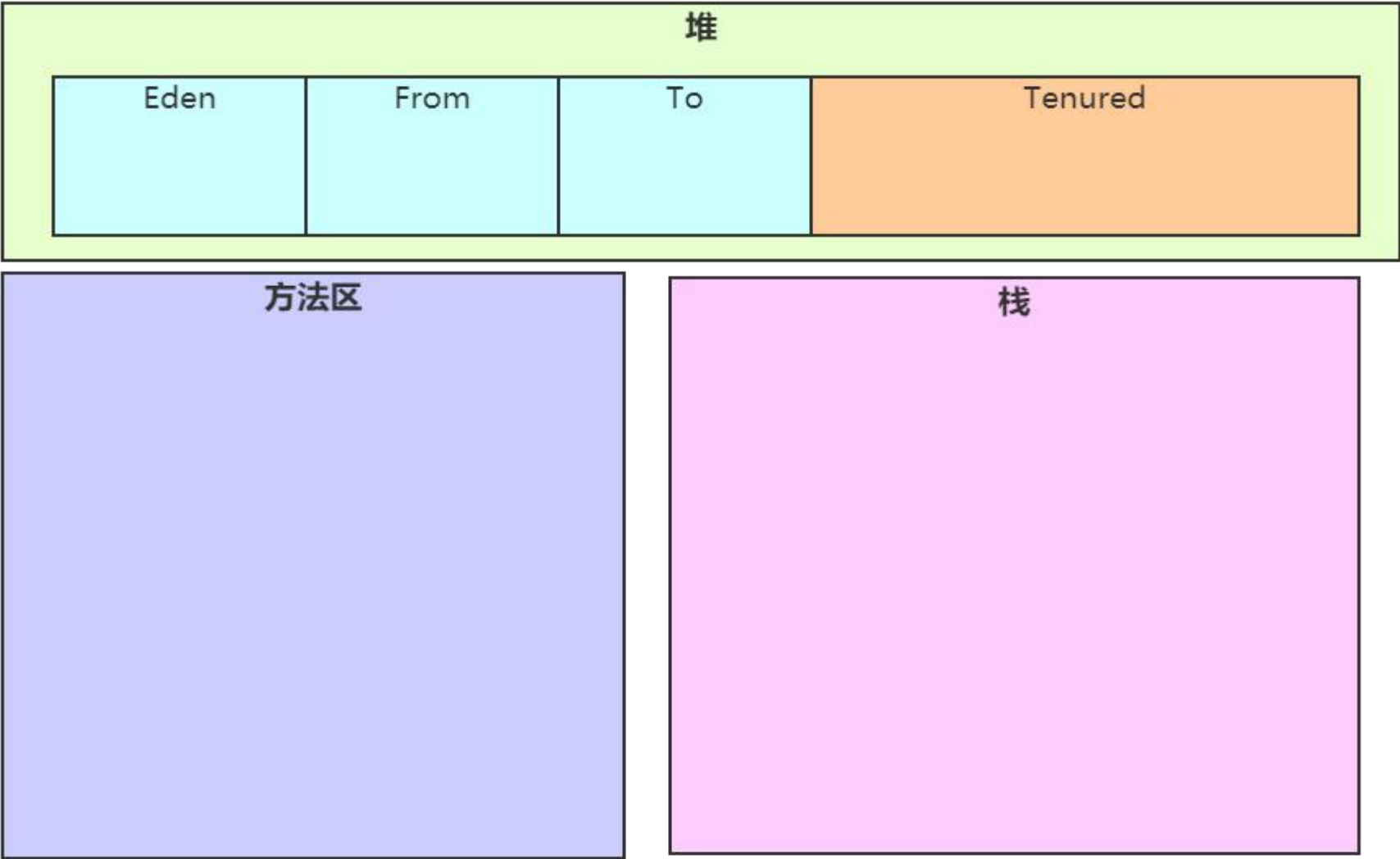
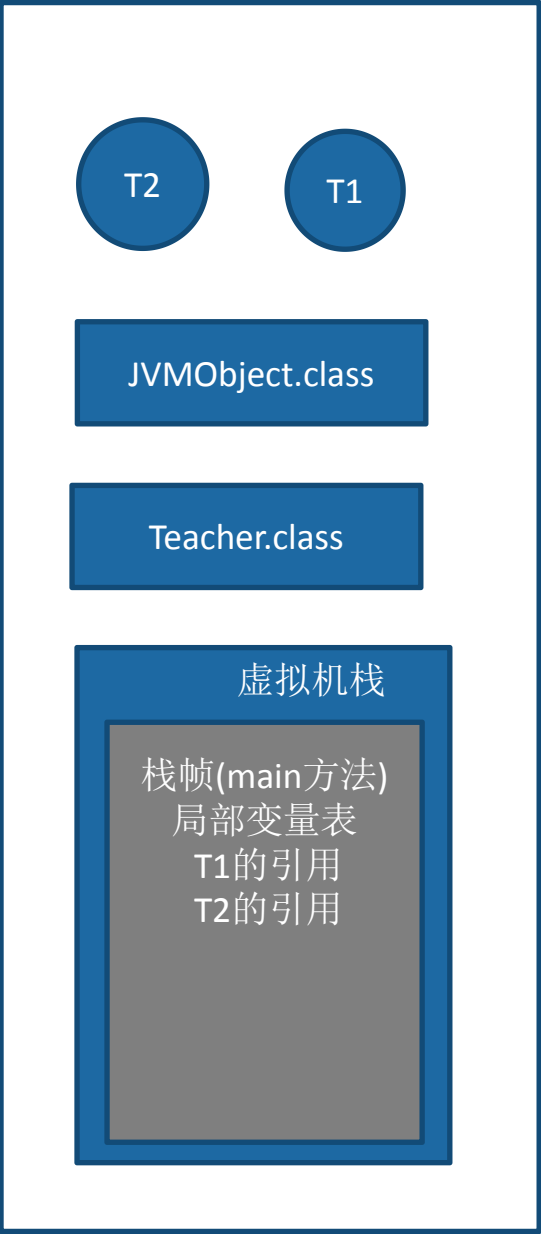


直接内存

不是虚拟机运行时数据区的一部分，也不是java虚拟机规范中定义的内存区域；

- ✓ 如果使用了NIO,这块区域会被频繁使用，在java堆内可以用directByteBuffer对象直接引用并操作；
- ✓ 这块内存不受java堆大小限制，但受本机总内存的限制，可以通过MaxDirectMemorySize来设置（默认与堆内存最大值一样），所以也会出现OOM异常；

从底层深入理解运行时数据区





■ 功能

- 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char等）以及对象的引用变量，其内存分配在栈上，变量出了作用域就会自动释放；
- 而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；

■ 线程独享还是共享

- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
- 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

■ 空间大小

- 栈的内存要远远小于堆内存，栈的深度是有限制的，可能发生StackOverflowError问题。

内存溢出



■ 内存溢出

- ✓ 栈溢出
- ✓ 堆溢出
- ✓ 方法区溢出
- ✓ 本机直接内存溢出

■ 解决方案

无法分配内存！ 可用内存不足，无法完成内存分配！	
无视	重启

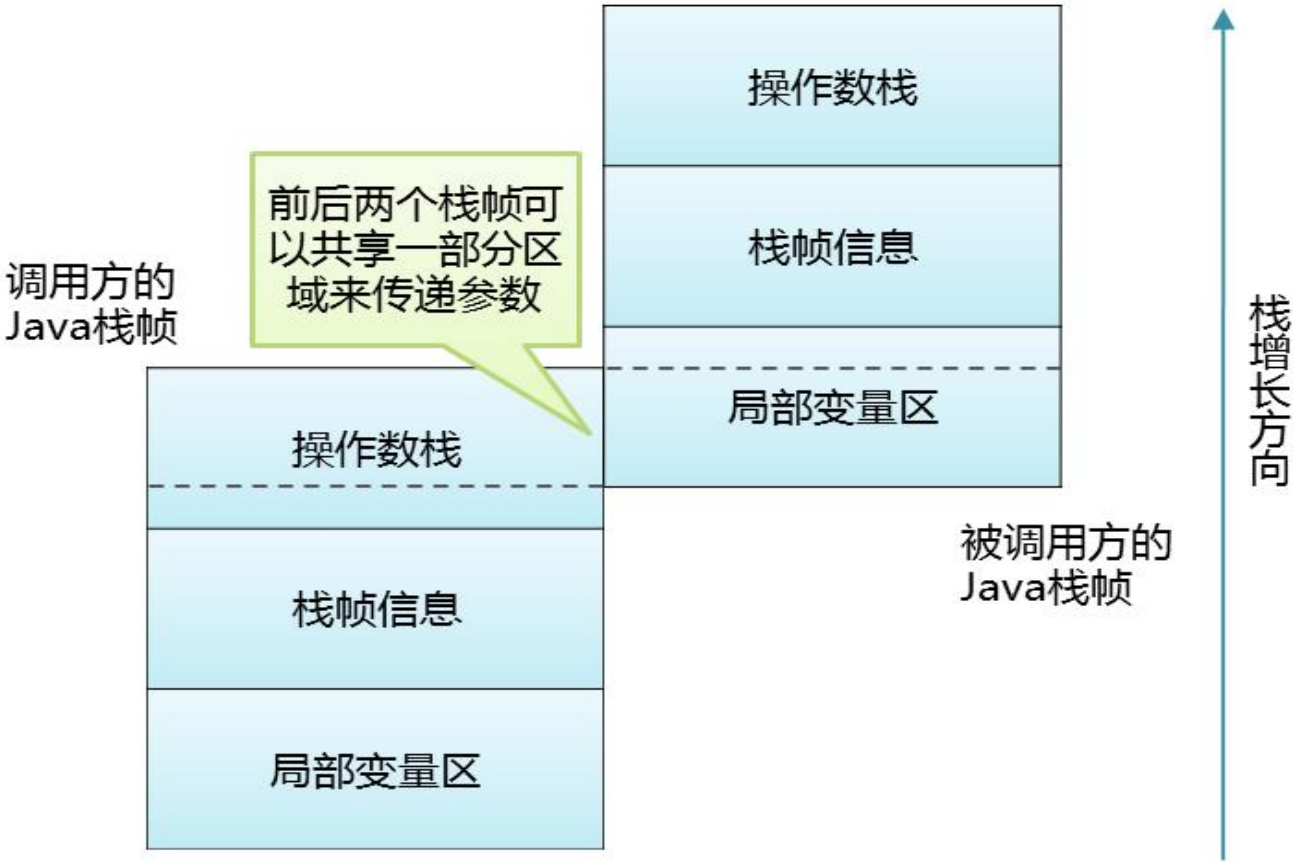


■编译优化技术

➤方法内联

■栈的优化技术

➤栈帧之间数据共享



深入理解

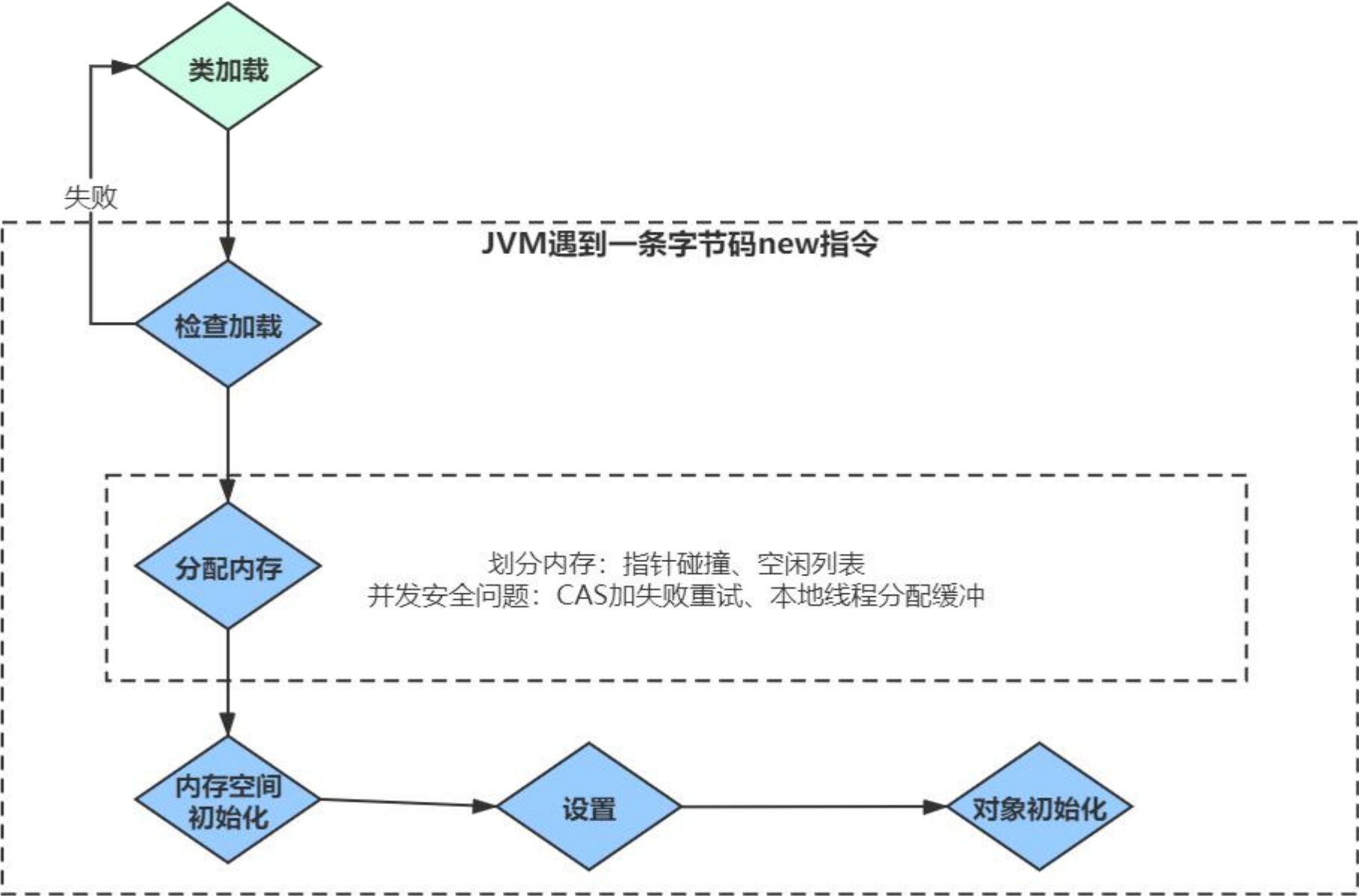
6月9日 20:00

对象与垃圾回收机制

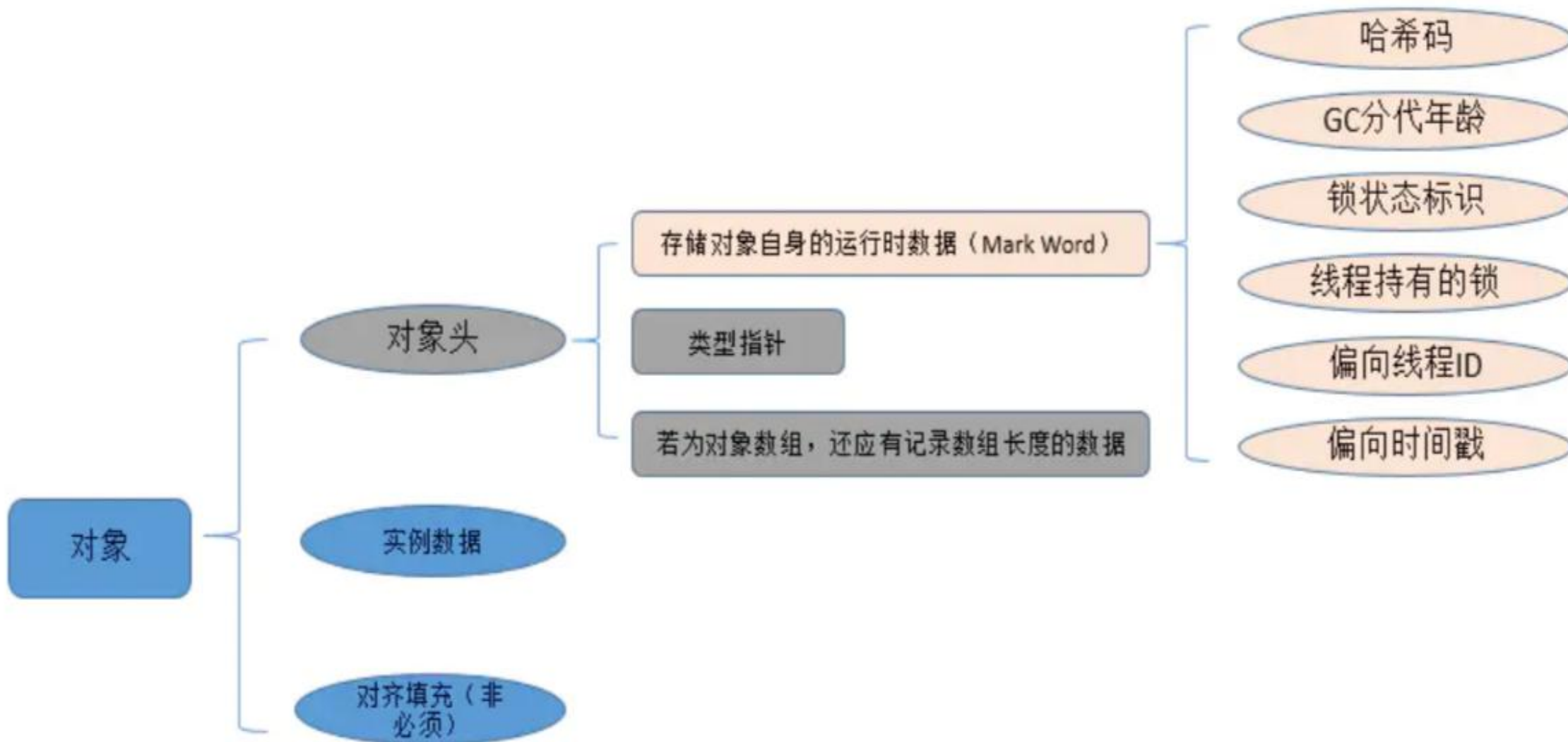
- 1.虚拟机中对象的创建过程
- 2.对象的存活以及各种引用
- 3.垃圾回收算法与垃圾收集器



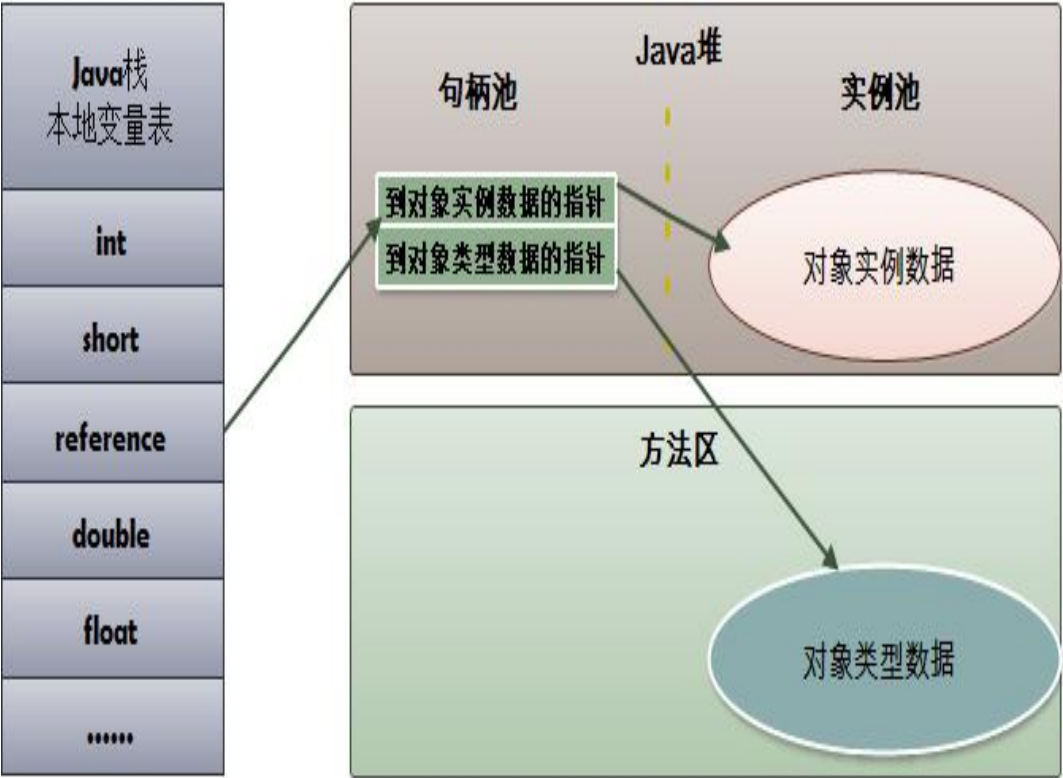
虚拟机中对象的创建过程



对象的内存布局

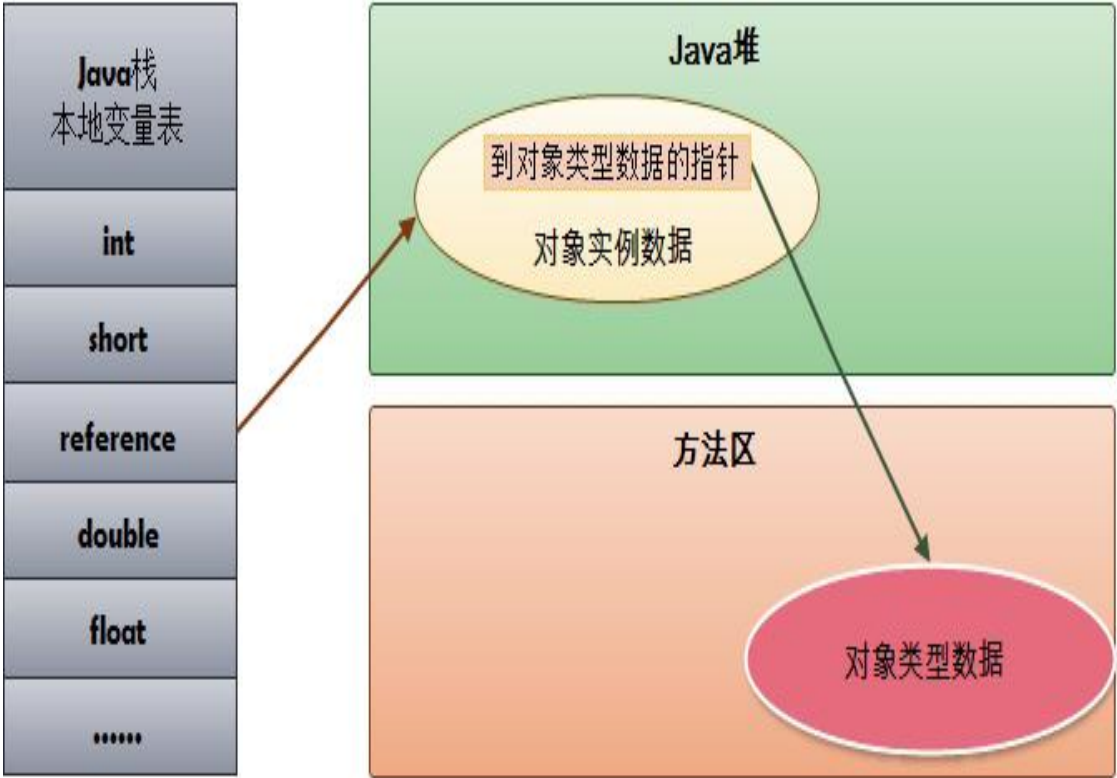


对象的访问定位



句柄方式访问对象

使用句柄



直接指针方式访问对象

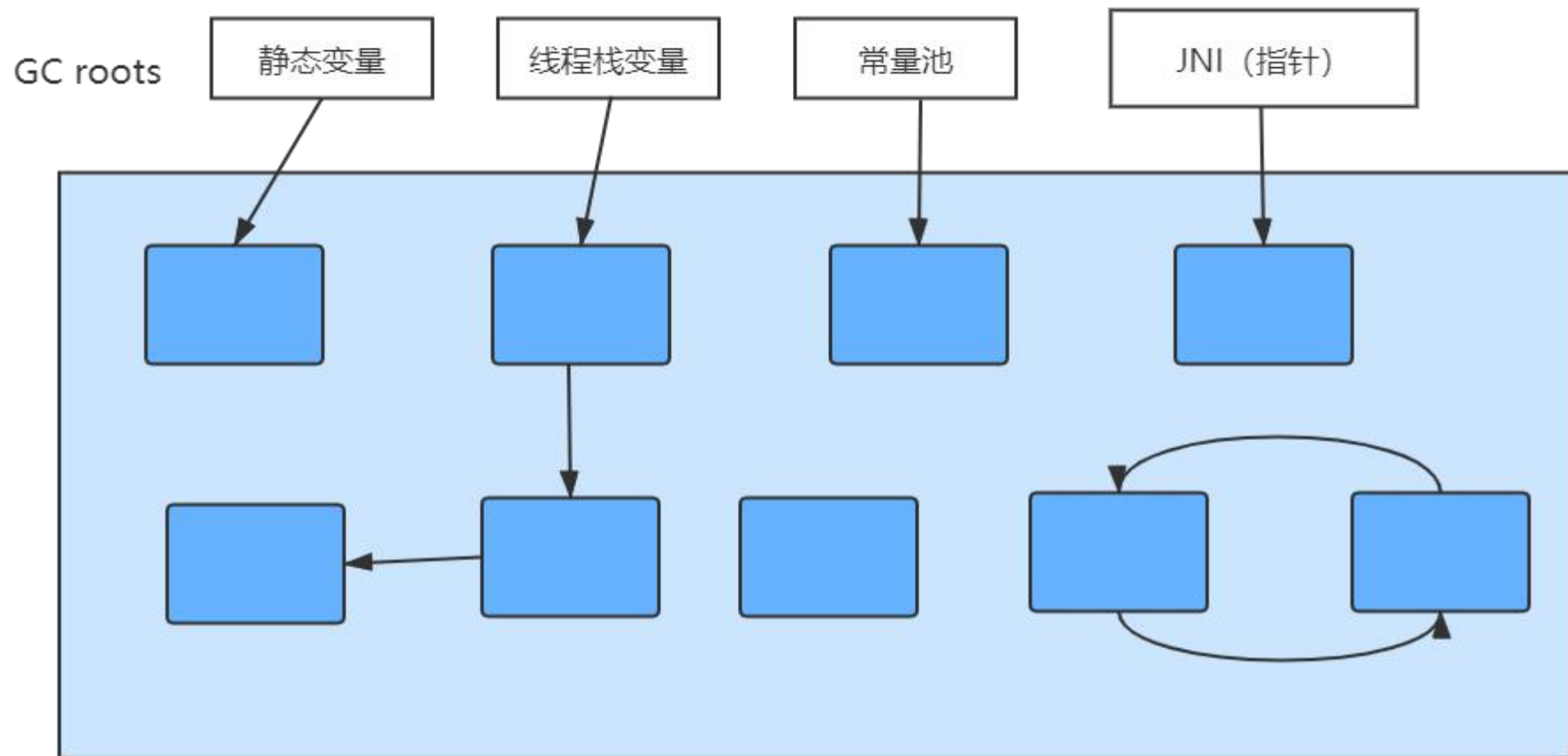
直接指针



判断对象的存活

- 引用计数算法
- 可达性分析(根可达)

■ finalize



各种引用



- 强引用 =
- 软引用 SoftReference
- 弱引用 WeakReference
- 虚引用 PhantomReference



■ Java与C++之间的区别

■ GC

Garbage Collection

■ GC的“自动化”时代

■ 谁需要GC？

■ GC要做的事

1、Where/Which？

2、When？

3、How？

■ 为什么我们要去了解GC和内存分配？

对象的分配策略



■ 对象的分配原则

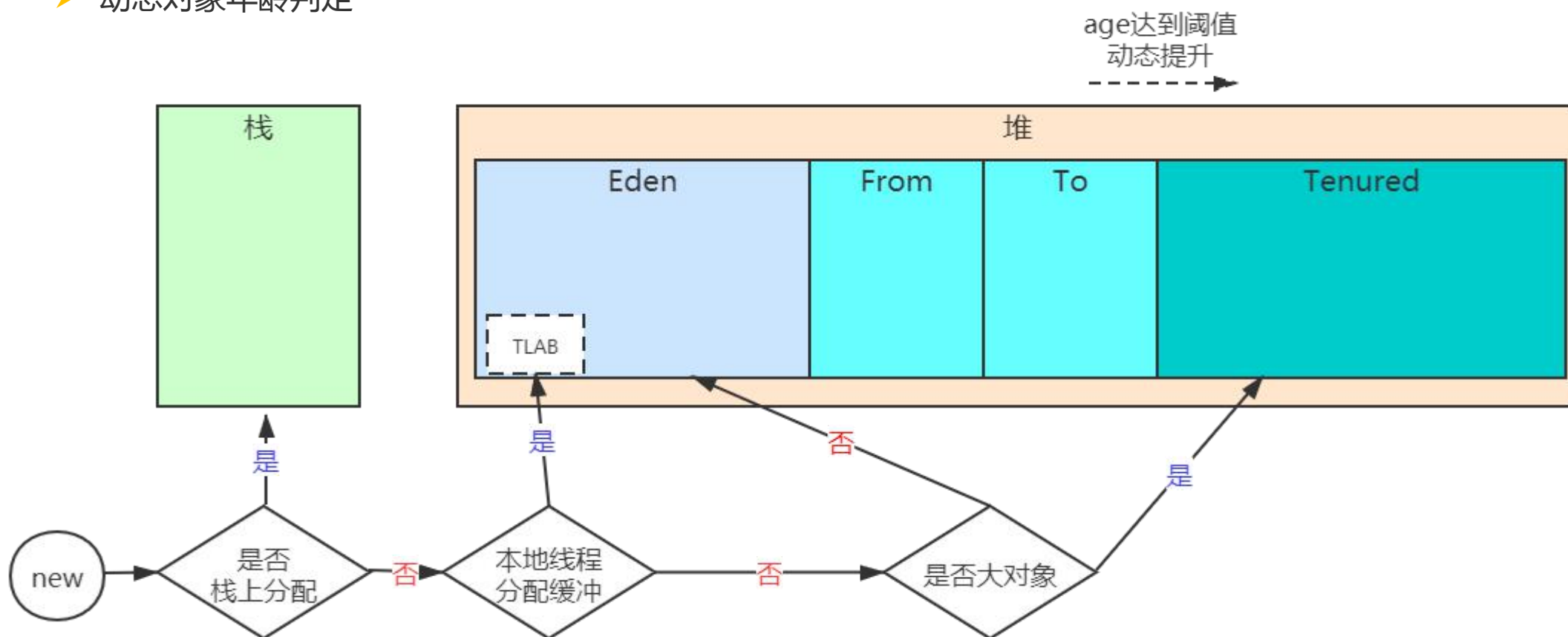
- 对象优先在Eden分配
- 空间分配担保
- 大对象直接进入老年代
- 长期存活的对象进入老年代
- 动态对象年龄判定

■ 栈中分配对象

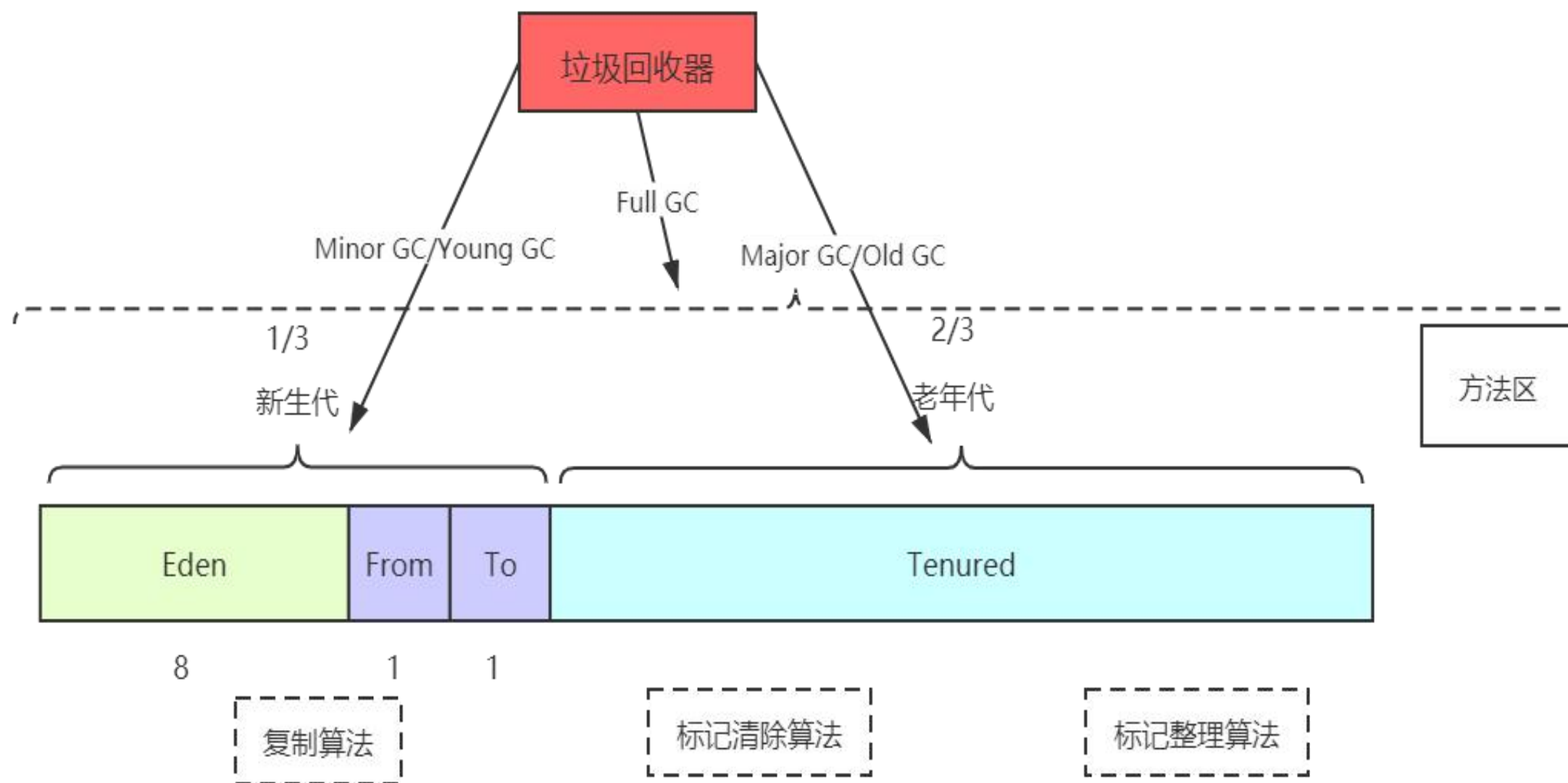
- 逃逸分析

■ 堆中的优化技术

- 本地线程分配缓冲(TLAB)



分代收集理论



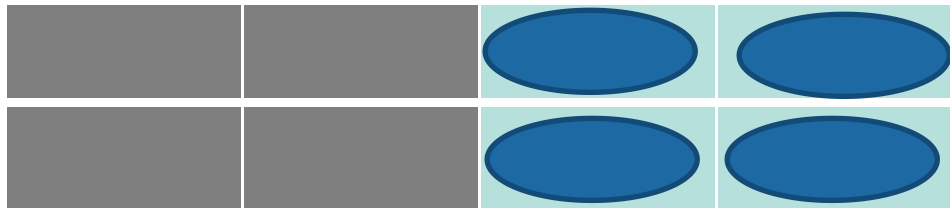


复制算法 (Copying)



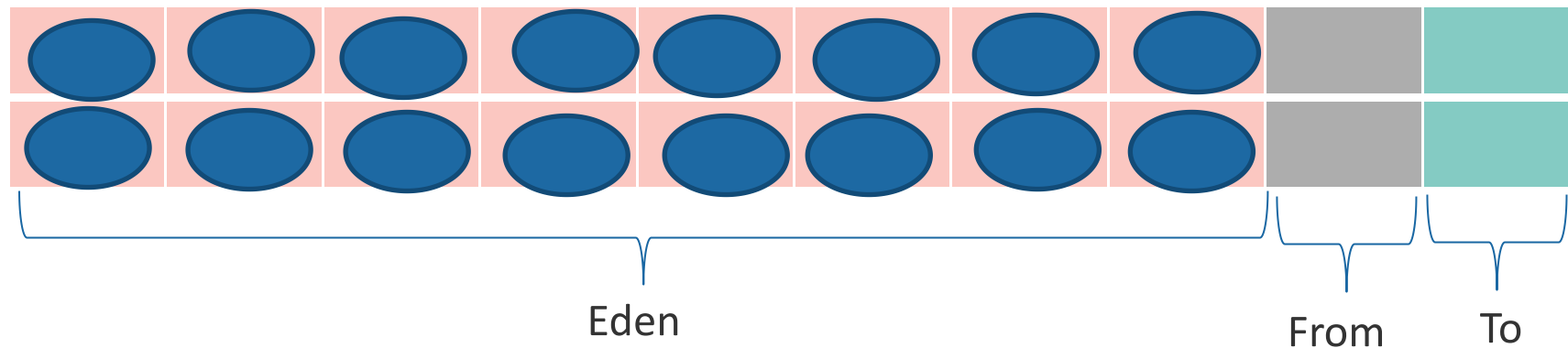
■ 特点

- 实现简单、运行高效
- 内存复制、没有内存碎片
- 利用率只有一半

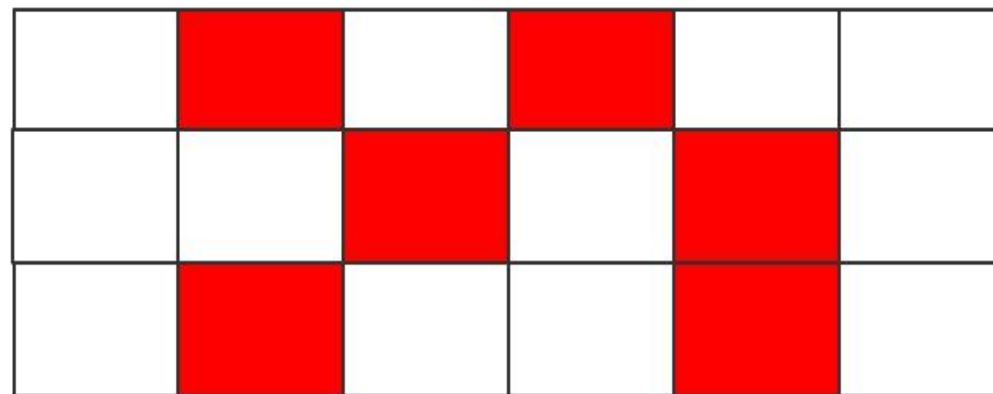


■ Eden区的来源

- Appel式回收
- 提高空间利用率和空间分配担保



➤ 内存碎片导致提前GC

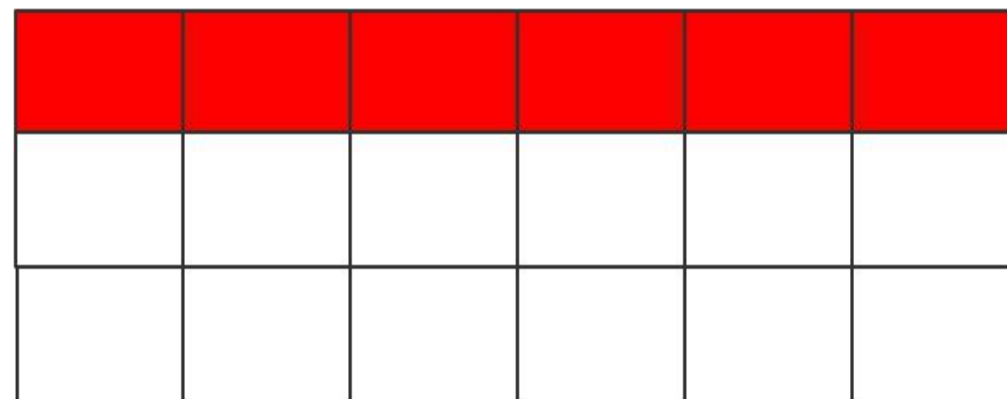
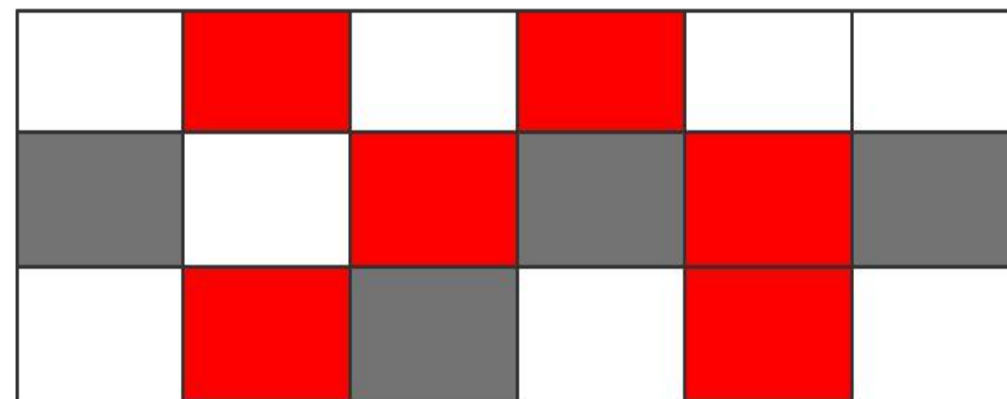


对象移动

引用更新

➤ 用户线程暂停

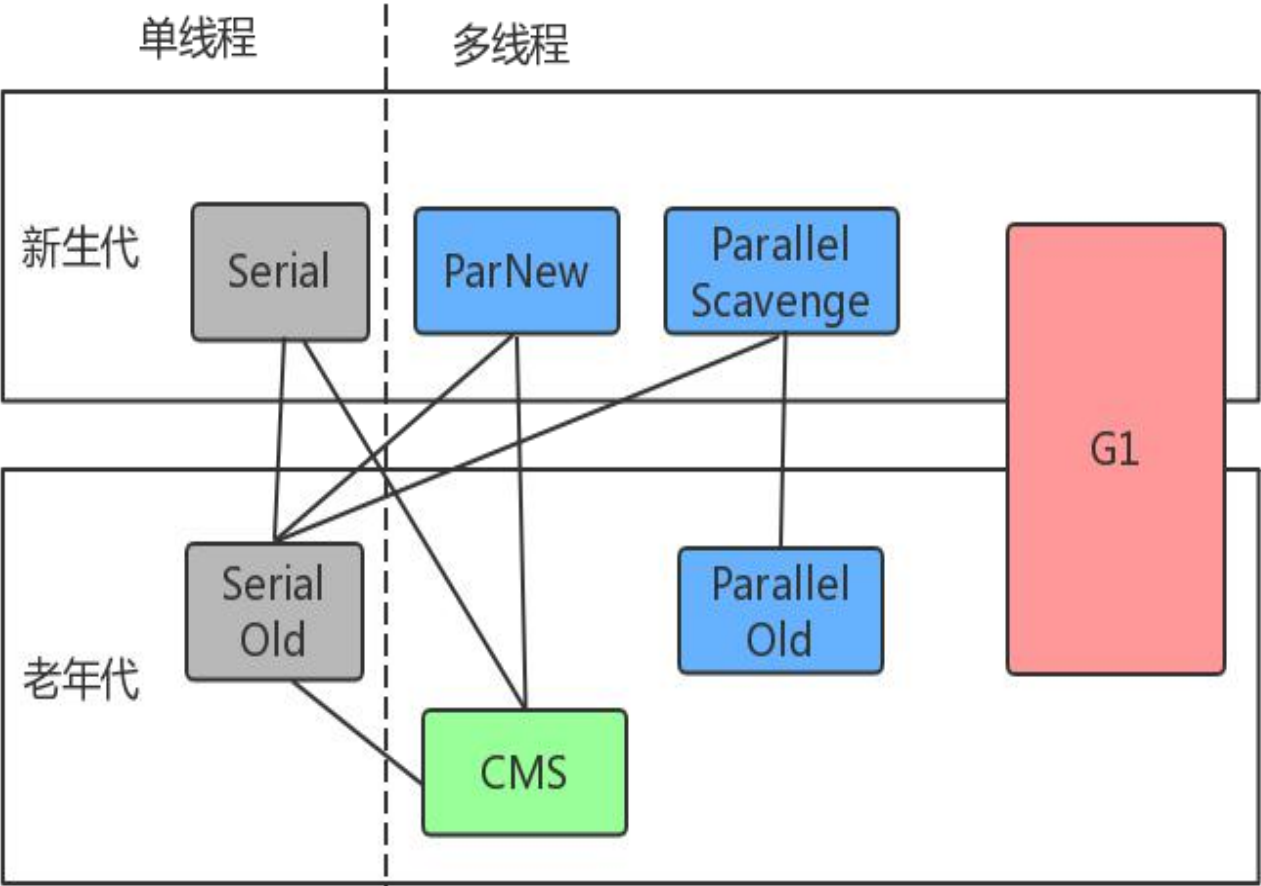
没有内存碎片



JVM中常见的垃圾收集器



- 单线程垃圾收集器
- 多线程并行垃圾收集器
- 多线程并发垃圾收集器



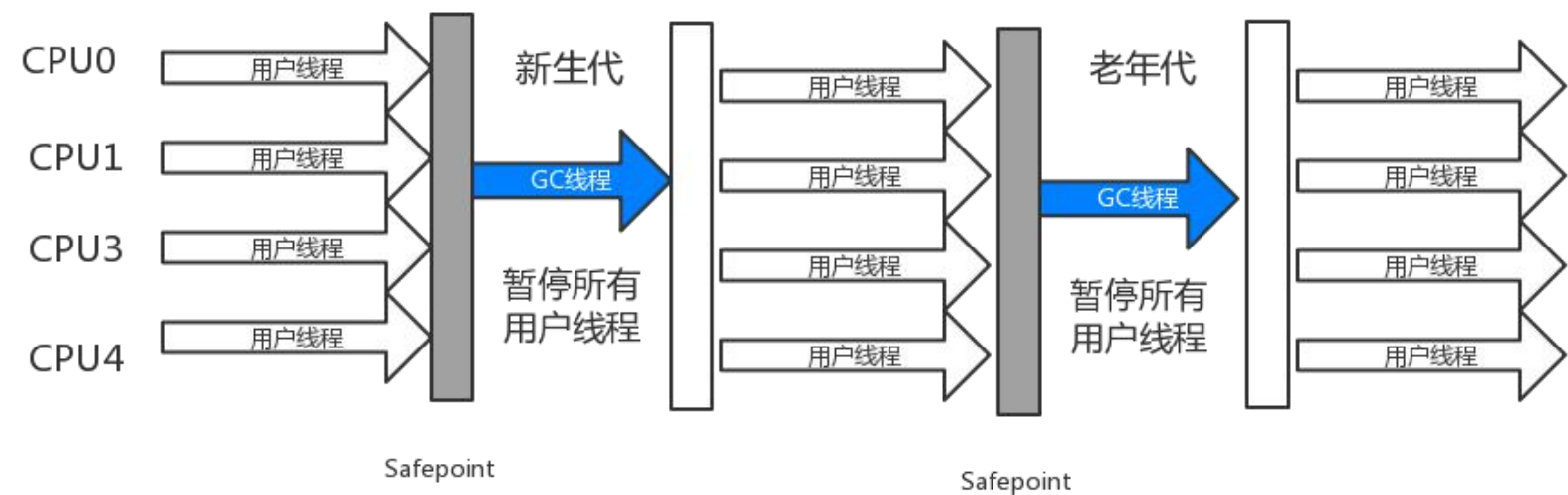
收集器	收集对象和算法	收集器类型
Serial	新生代，复制算法	单线程
ParNew	新生代，复制算法	并行的多线程收集器
Parallel Scavenge	新生代，复制算法	并行的多线程收集器

收集器	收集对象和算法	收集器类型
Serial Old	老年代，标记整理算法	单线程
Parallel Old	老年代，标记整理算法	并行的多线程收集器
CMS	老年代， 标记清除 算法	并行与 并发 收集器
G1	跨新生代和老年代； 标记整理 + 化整为零	并行与 并发 收集器

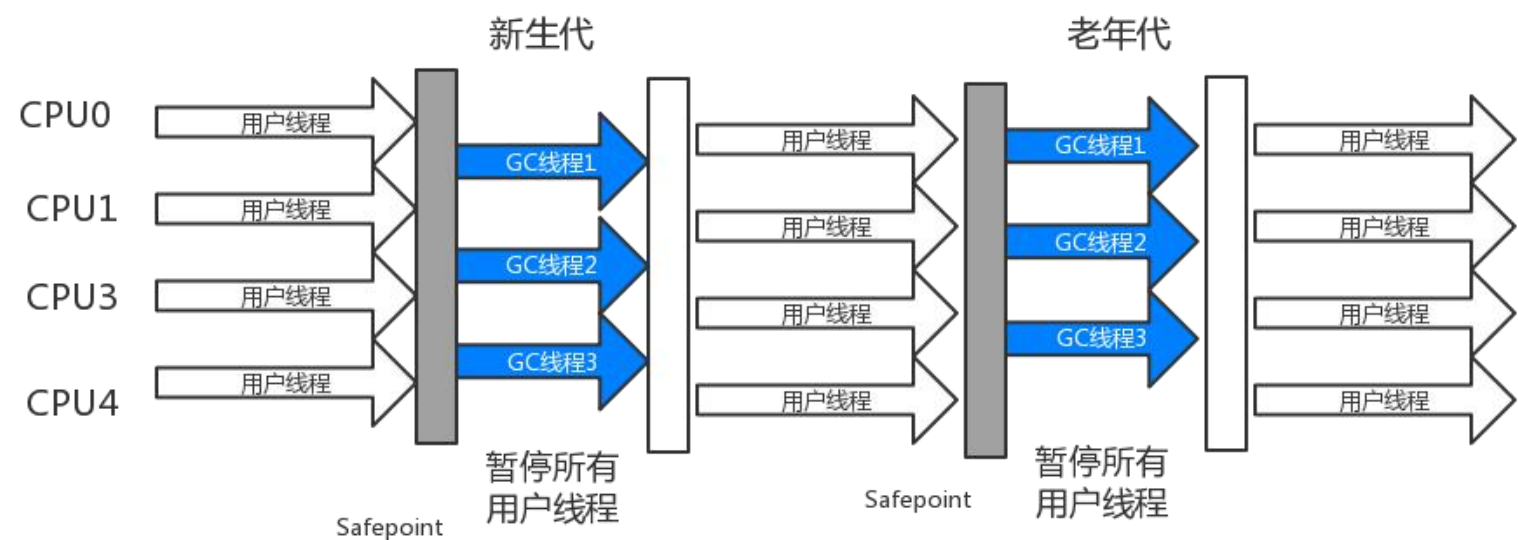
简单的垃圾回收器工作示意图



■ 单线程收集



■ 多线程收集 并行收集





CMS垃圾回收器工作示意图



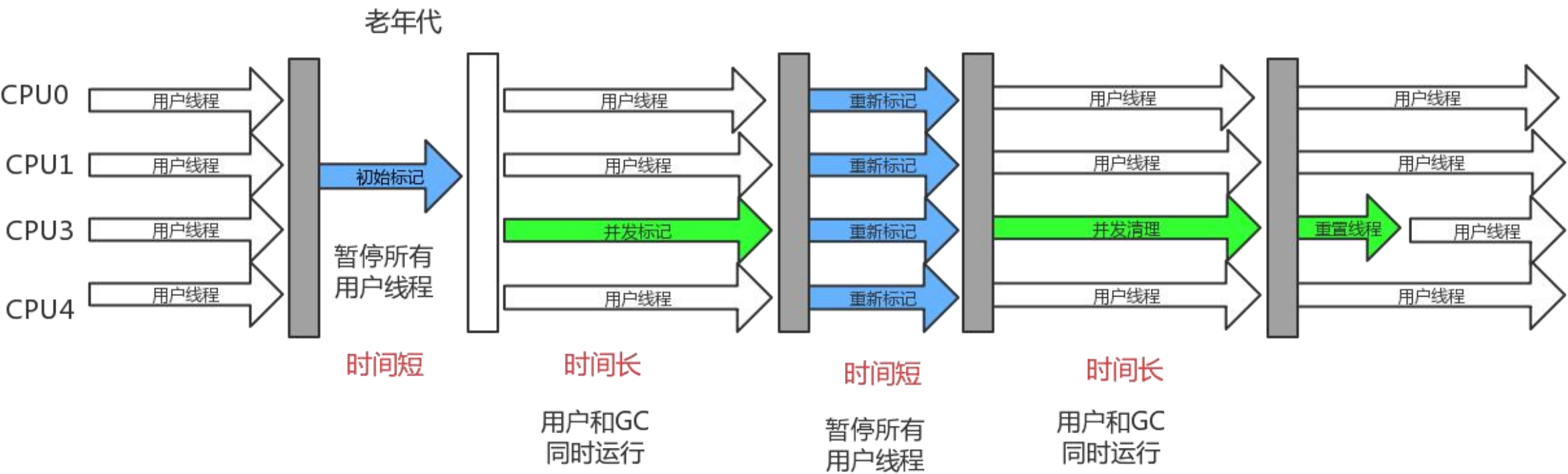
Concurrent Mark Sweep

标记清除算法

- 初始标记 -- 暂停
- 并发标记 -- 同时进行
- 重新标记 -- 暂停
- 并发清除 -- 同时进行

CMS的缺点

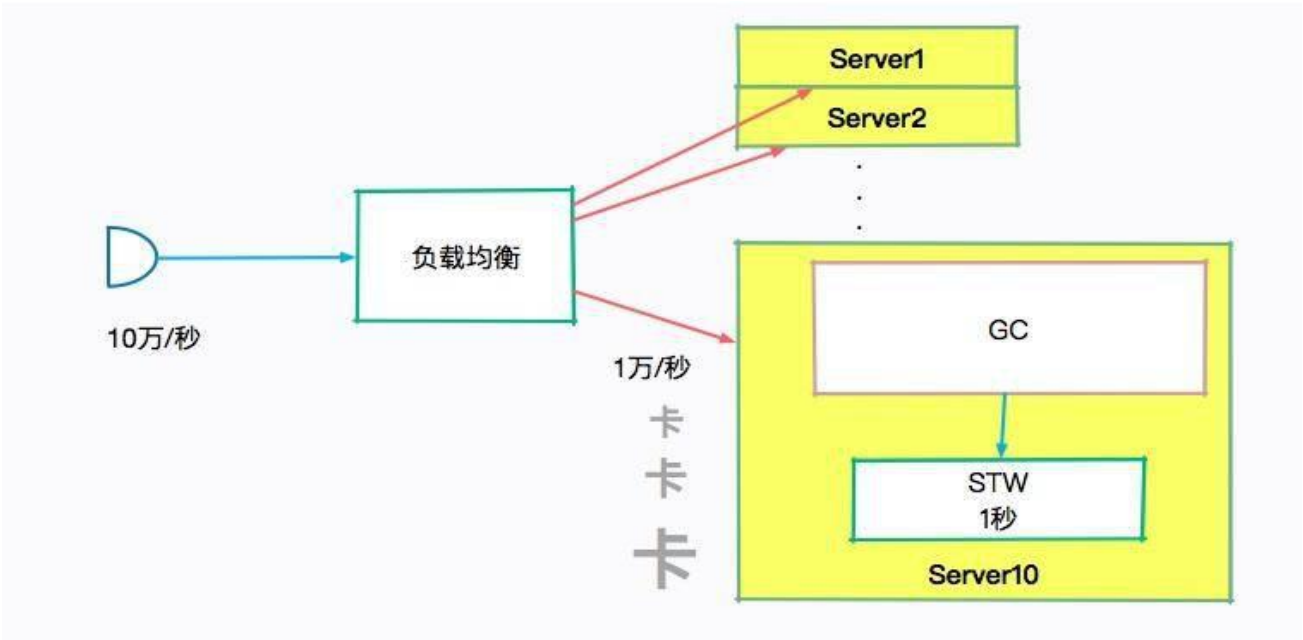
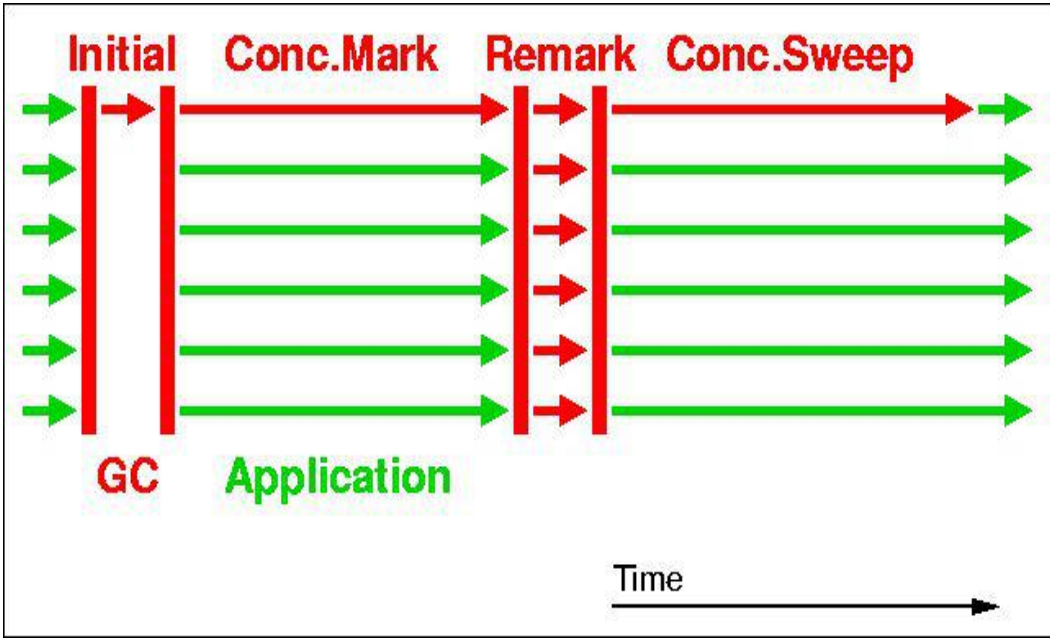
- CPU敏感
- 浮动垃圾
- 内存碎片



Stop The World现象



- 什么是STW
- 为什么要STW
- STW的危害

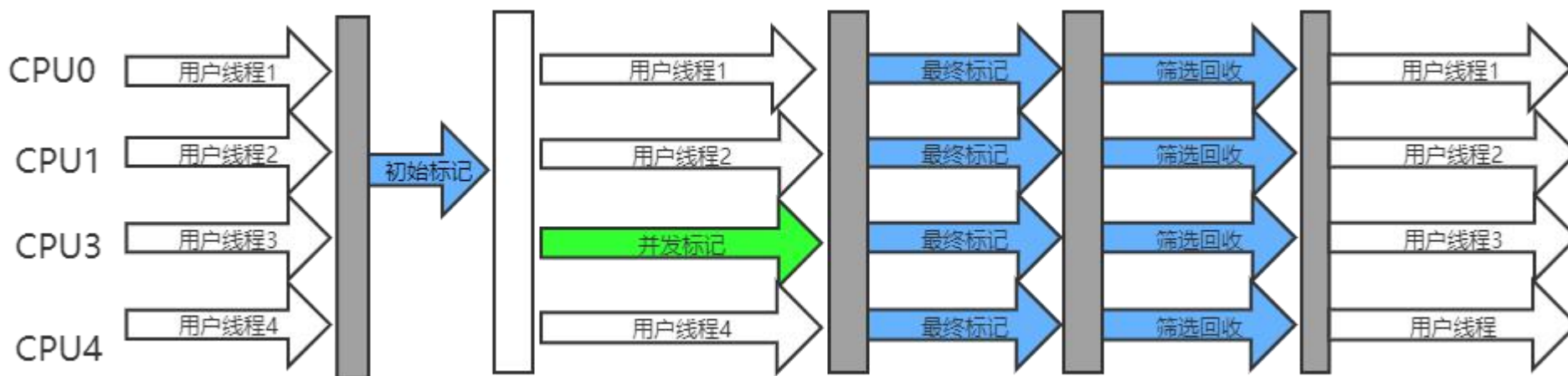
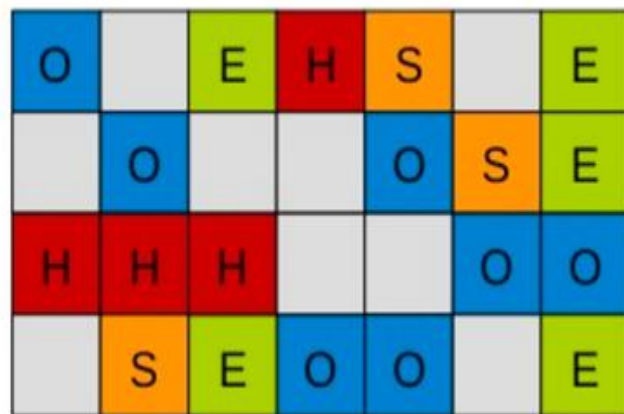


G1图示



■ Garbage First

- 追求停顿时间
- Region区
- 筛选回收
- 可预测停顿
- 算法——复制和标记整理





常量池与String

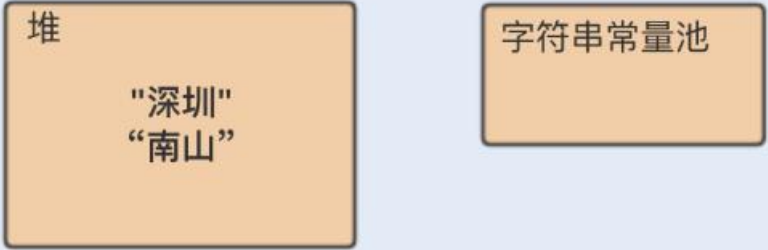
■ 常量池

- ✓ 静态常量池
- ✓ 运行时常量池

■ String 的创建分配内存地址

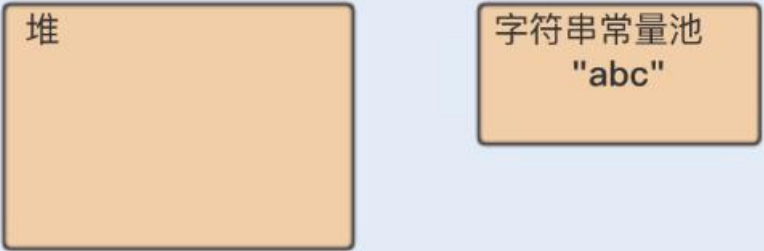
```
public class Location {  
    private String city;  
    private String region;  
}
```

在运行时，创建的String对象会在堆中直接创建，不会再常量池中创建



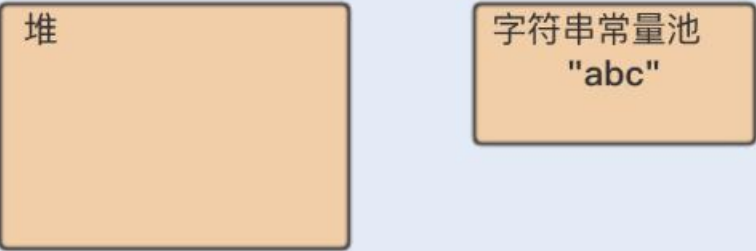
String str="abc"

代码编译加载时，会在常量池中创建常量“abc”，运行时，返回常量池中的字符串引用

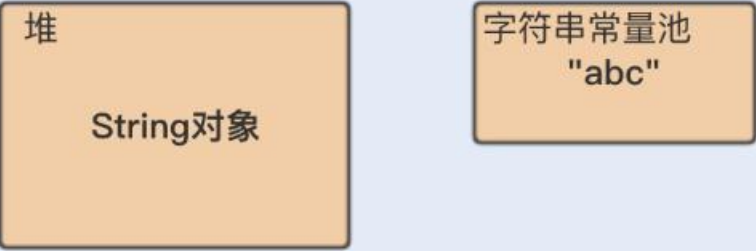


String str = new String("abc")

1) 代码编译加载时，会在常量池中创建常量“abc”



2) 在调用new时，会在堆中创建String对象，并引用常量池中的字符串对象char[]数组。并返回String对象引用。



面试常见问题剖析



- JVM内存结构说一下！
- 什么情况下内存栈溢出？
- 描述new一个对象的流程！
- Java对象会不会分配在栈中？
- 如果判断一个对象是否被回收，有哪些算法，实际虚拟机使用得最多的是什么？
- GC收集算法有哪些？他们的特点是什么？
- JVM中一次完整的GC流程是怎样的？对象如何晋级到老年代？
- Java中的几种引用关系，他们的区别是什么？
- final、finally、finalize的区别？
- `String s = new String("xxx");`创建了几个对象？