虚拟机的前世今生
深入理解JVM内存区域

T H A N K Y O U F O R W A T C H I N G

享学课堂_主讲老师：King

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
3	

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。

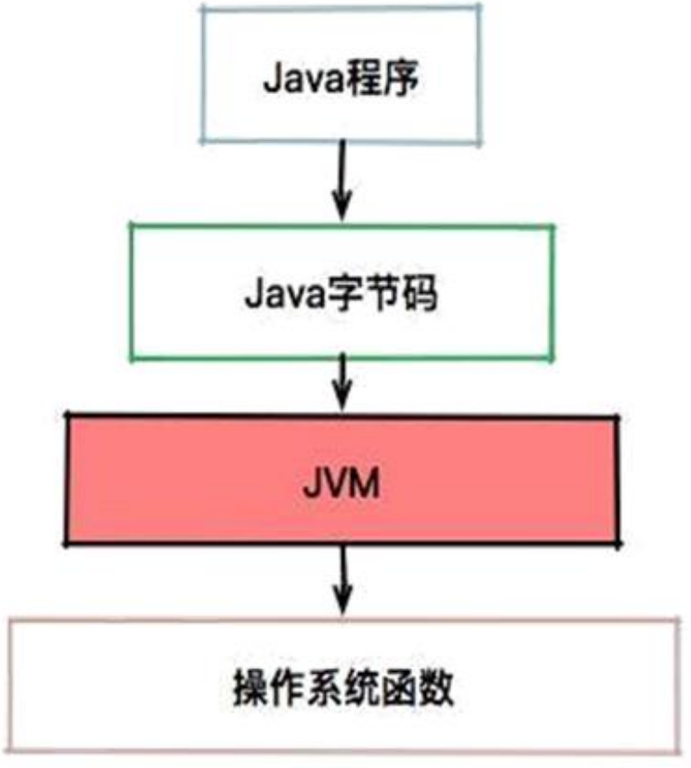
3、以上为**Java语言高级特性**的章节安排，不代表上课次数，如果一章内容在一次课内未讲完，则会顺延到下次课继续讲解。

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。**

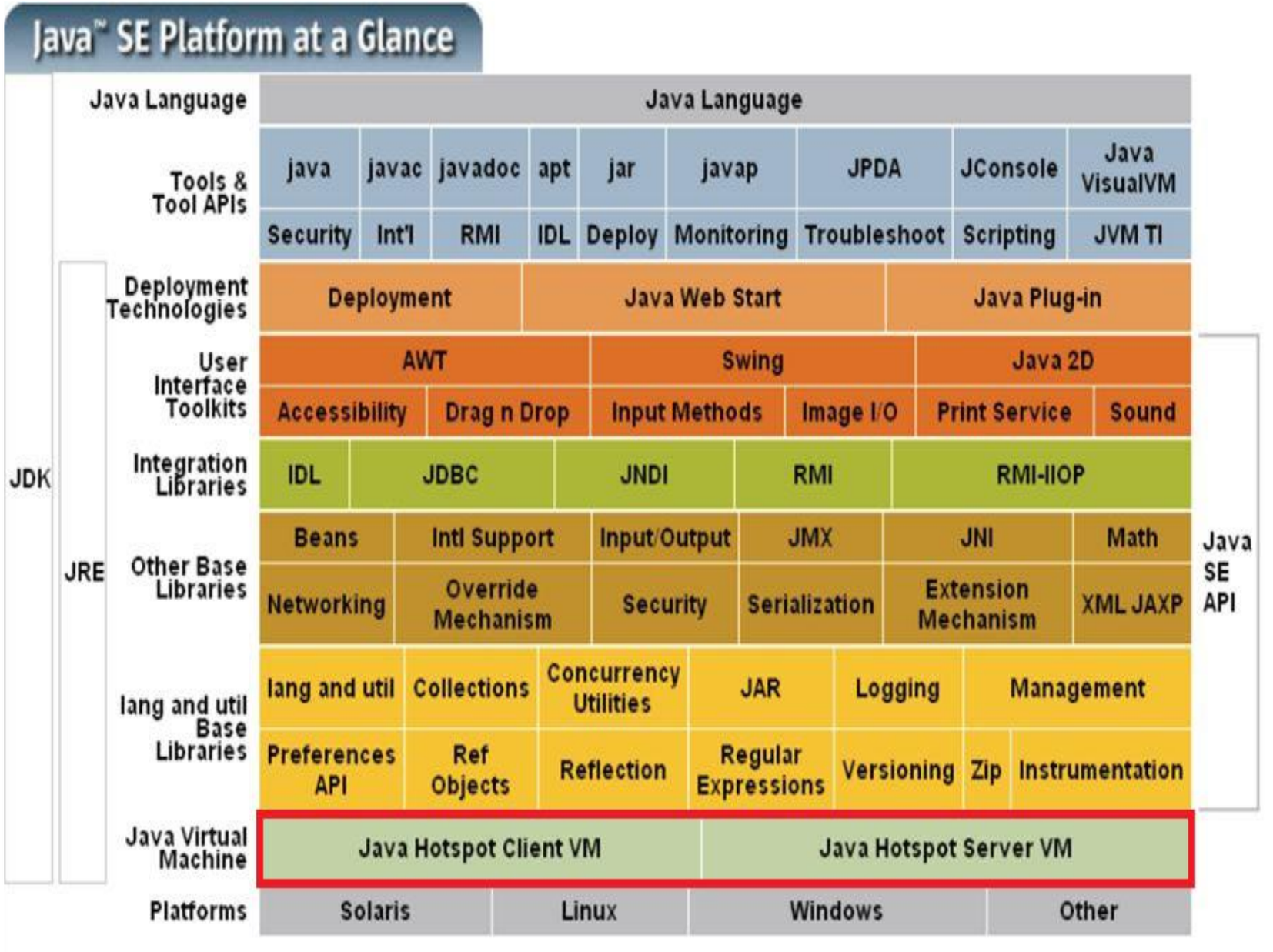
JVM与操作系统的关系

- Java Virtual Machine
- 翻译
- 从跨平台到跨语言



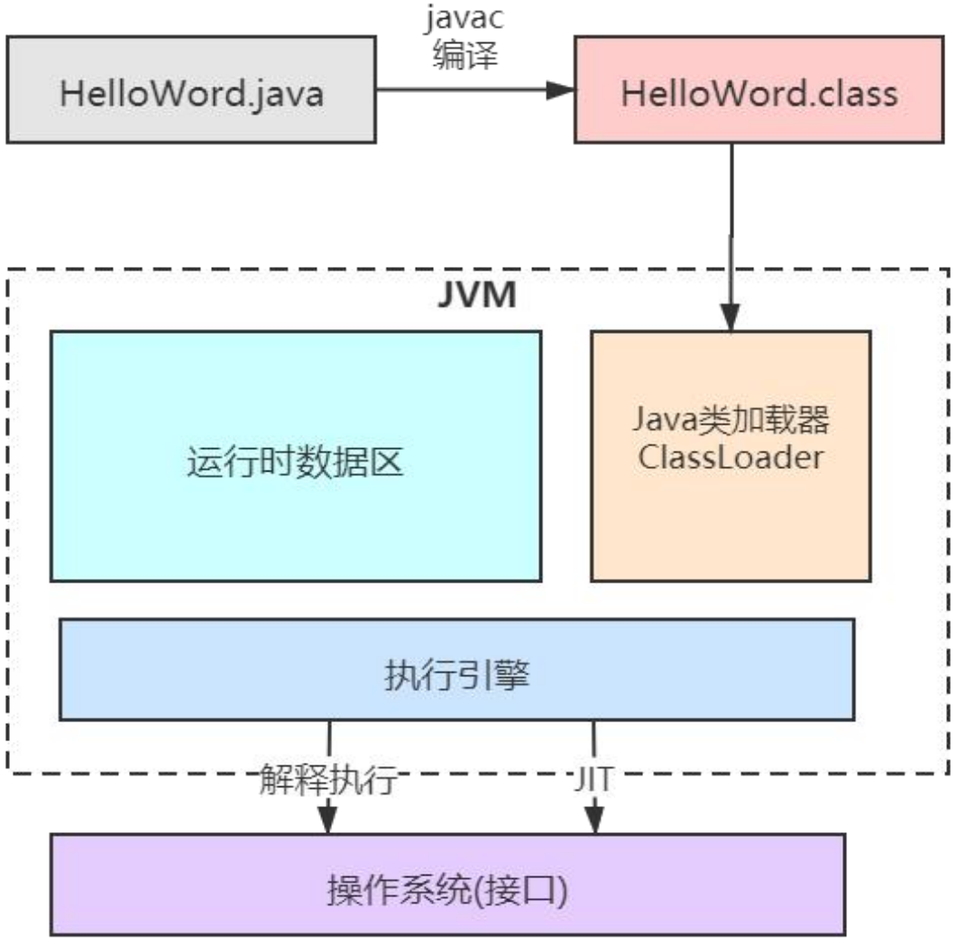
Java SE体系架构

- JVM只是一个翻译
- JRE提供了基础类库
- JDK提供了工具



JVM整体

- JVM的运行过程
- 本次课程的重点



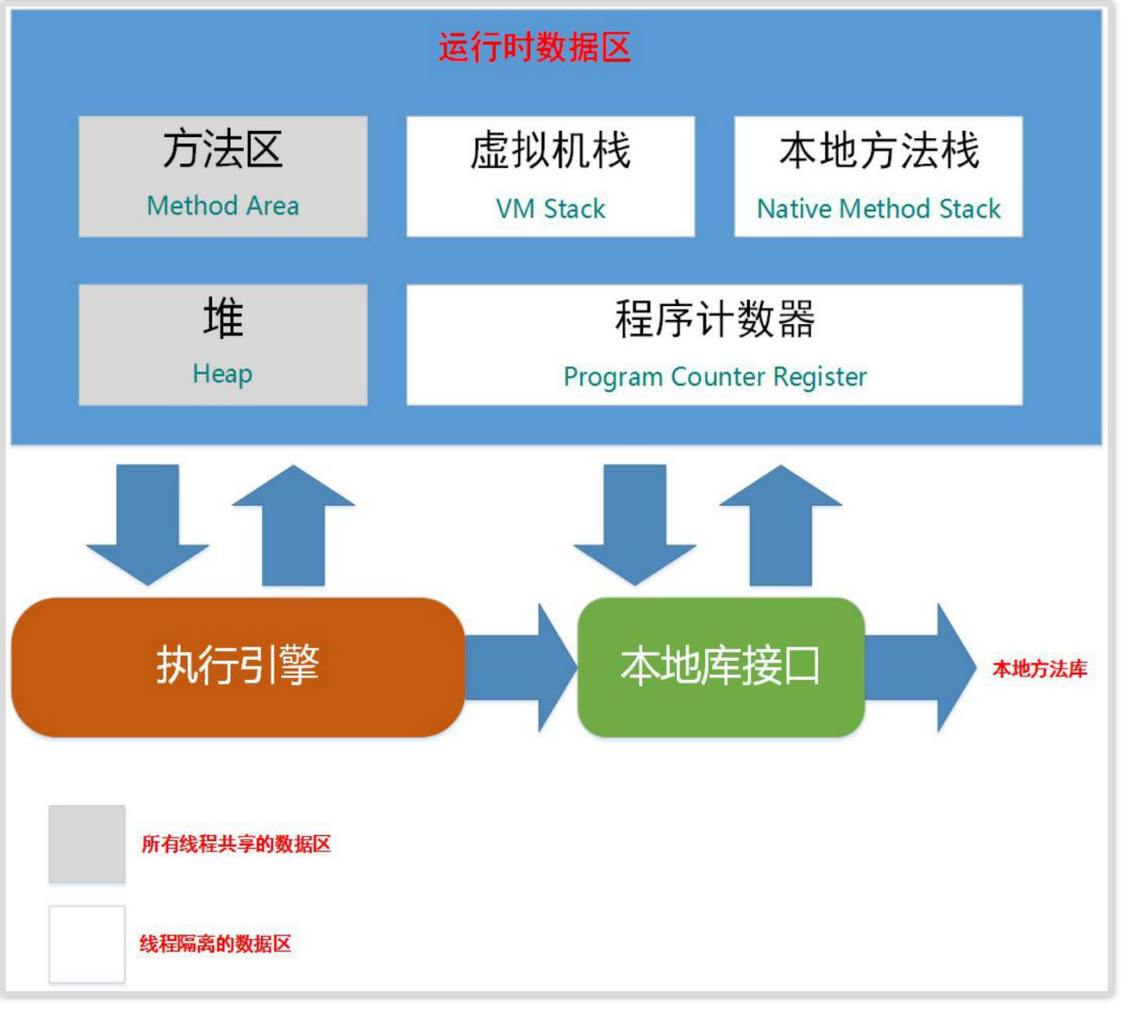
运行时数据区域

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**

栈帧执行对内存区域的影响

```java
/**
 * @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 int 型 1 入操作数栈
istore_1	将操作数栈中栈顶 int 型数值，存入局部变量表（下标为 1 的位置）
iconst_2	将 int 型 2 入操作数栈
istore_2	将操作数栈中栈顶 int 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 int 型数据入栈
iload_2	将局部变量表中下标为 2 的 int 型数据入栈
iadd	1)将栈顶两 int 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 int 值入操作数栈
imul	1)将栈顶两 int 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 int 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 int 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

线程共享的区域

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

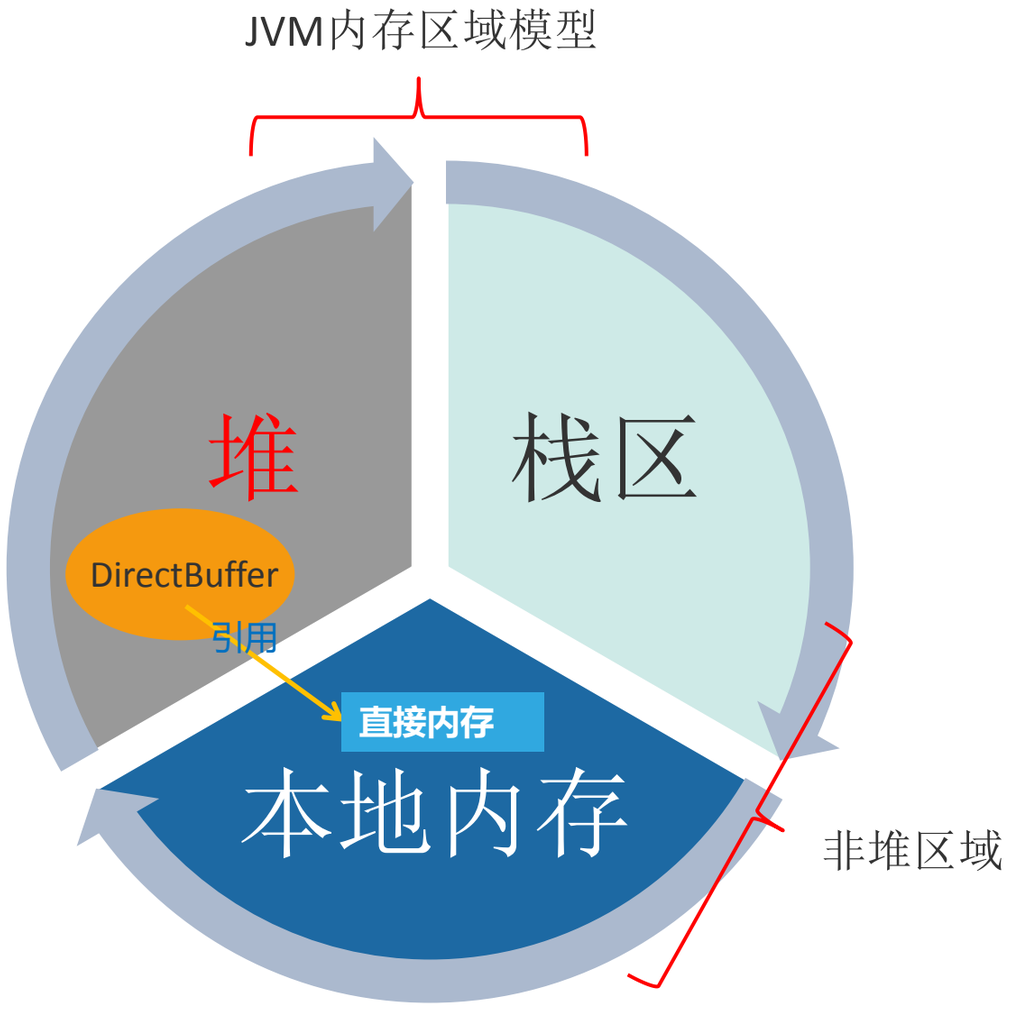
- ✓ 对象实例（几乎所有）
- ✓ 数组

- 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

直接内存

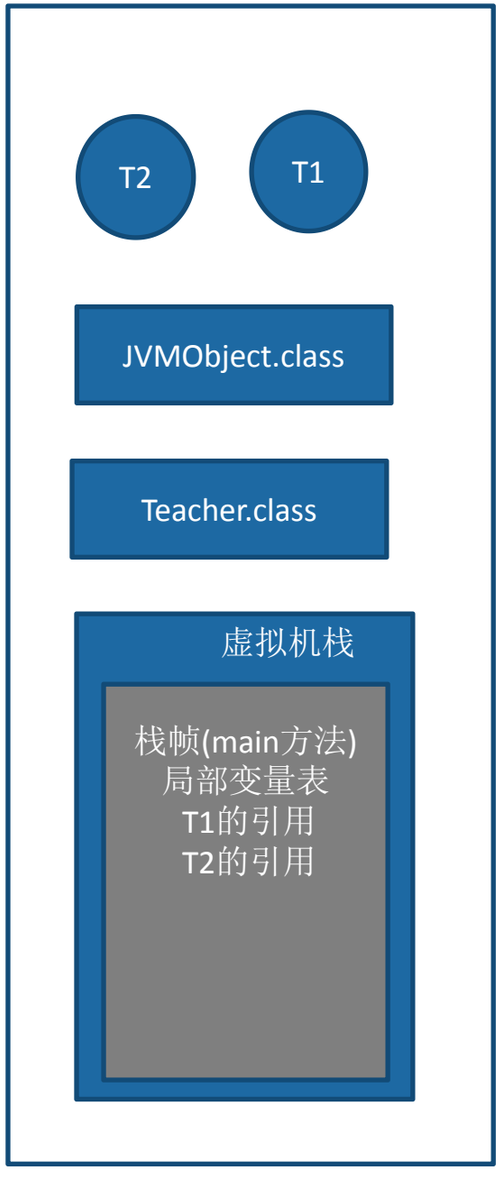


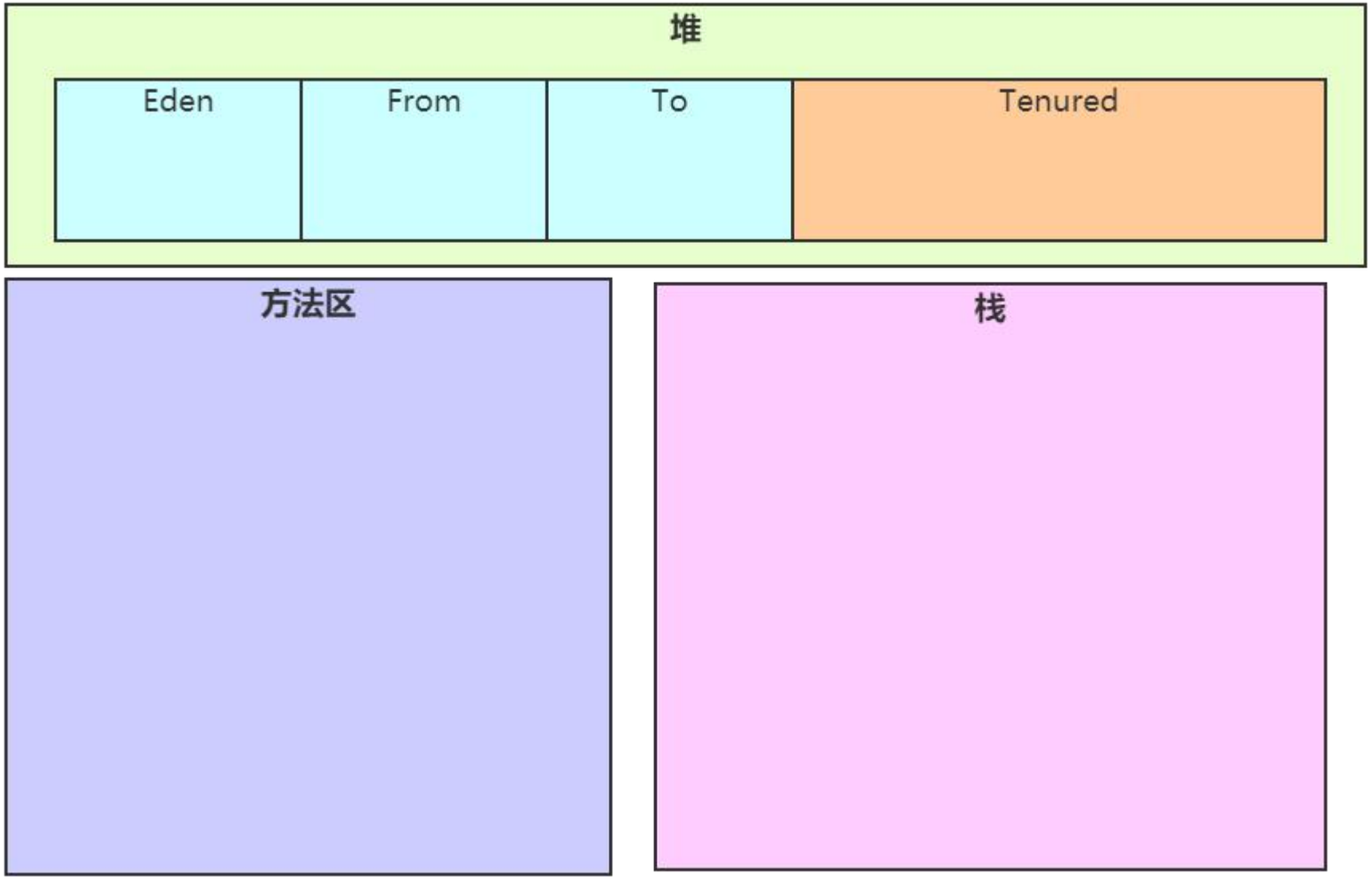
直接内存

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





深入辨析堆和栈

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。

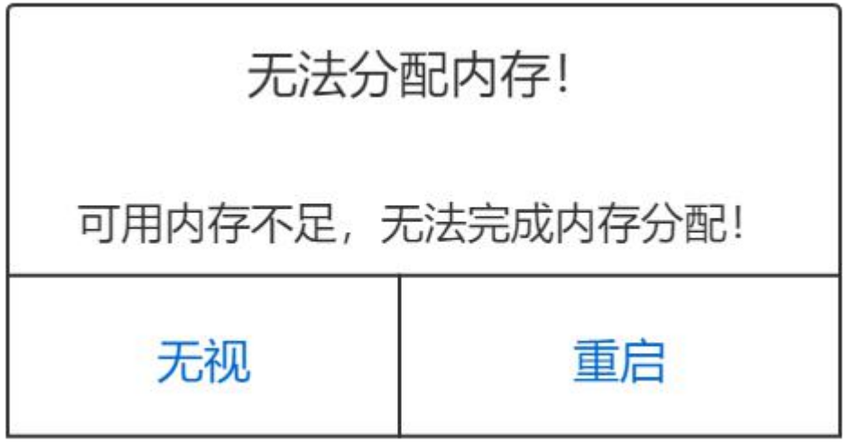
空间大小

- 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

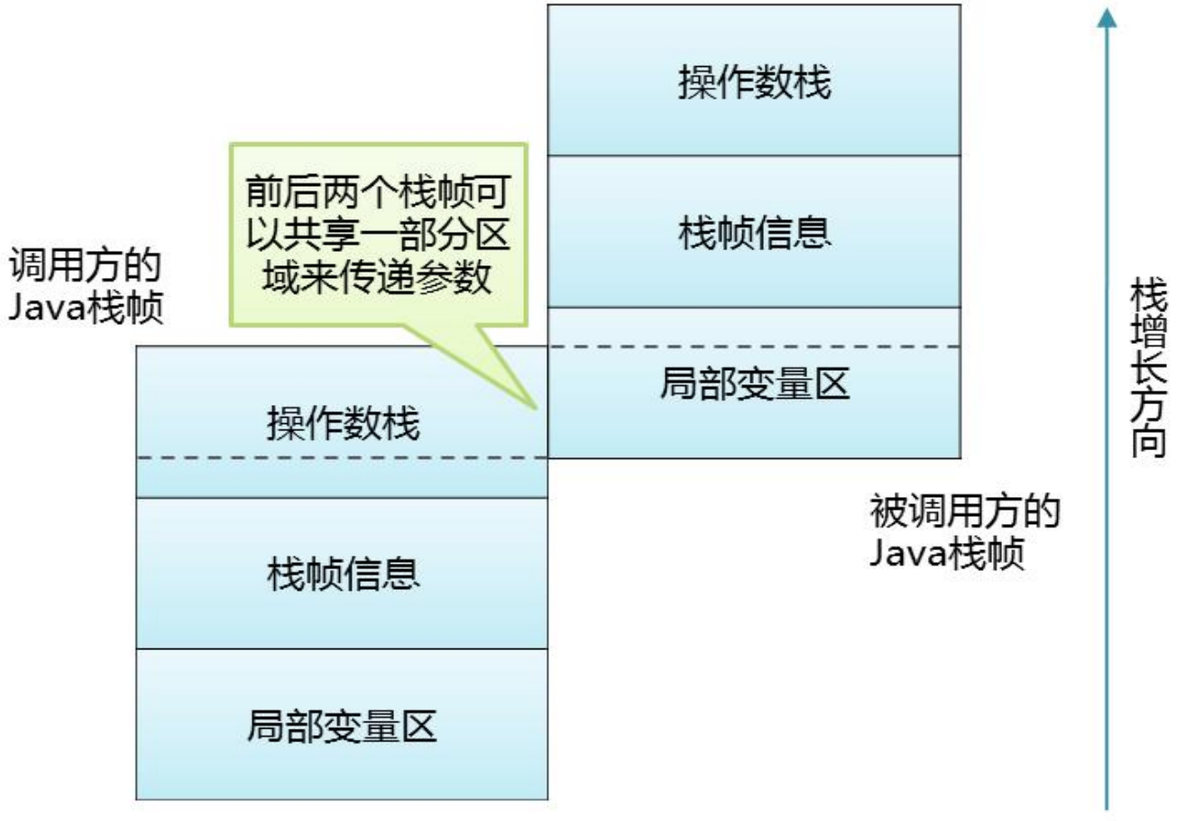
虚拟机优化技术

编译优化技术

- 方法内联

栈的优化技术

- 栈帧之间数据共享

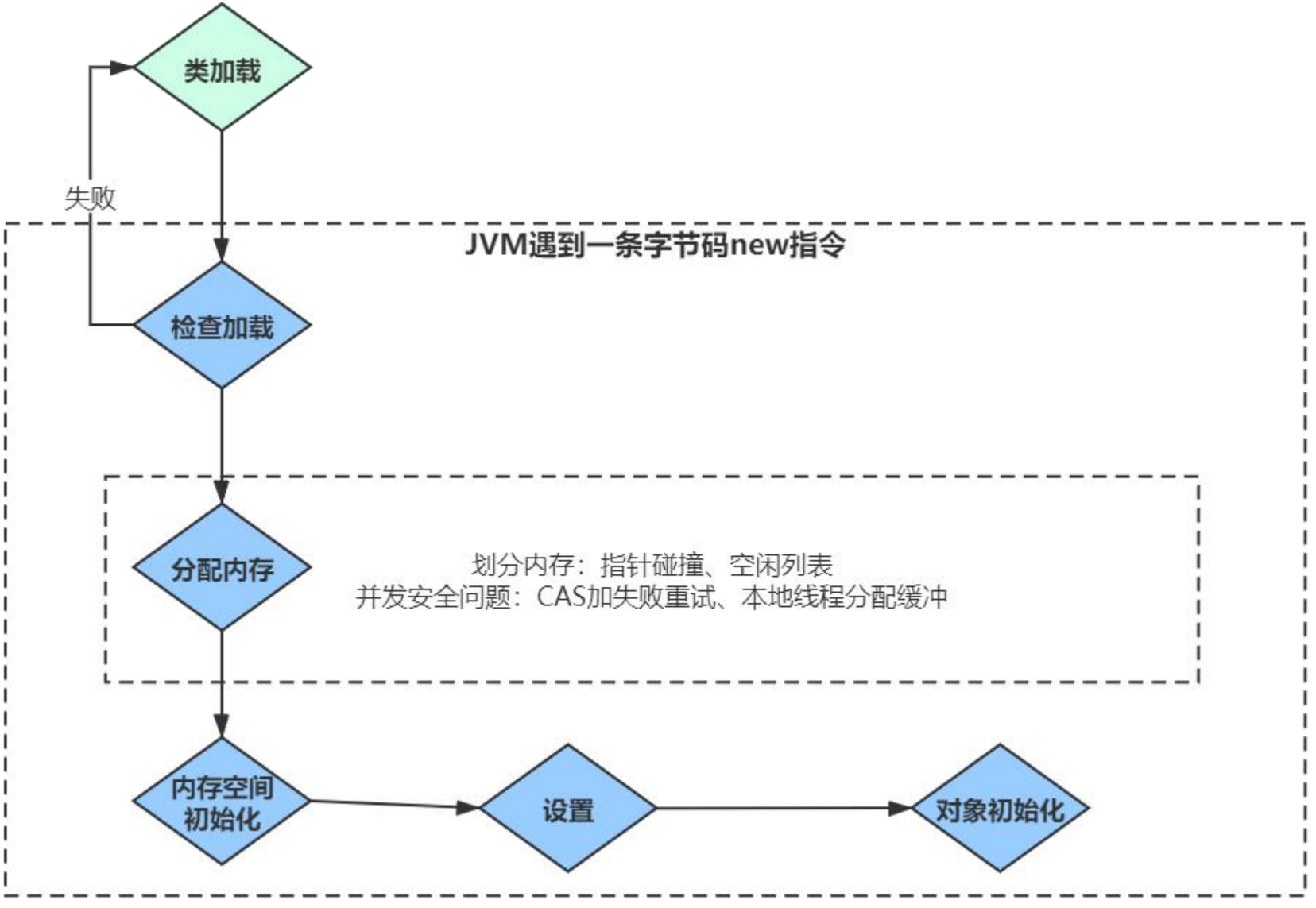


2、深入理解对象与垃圾回收机制

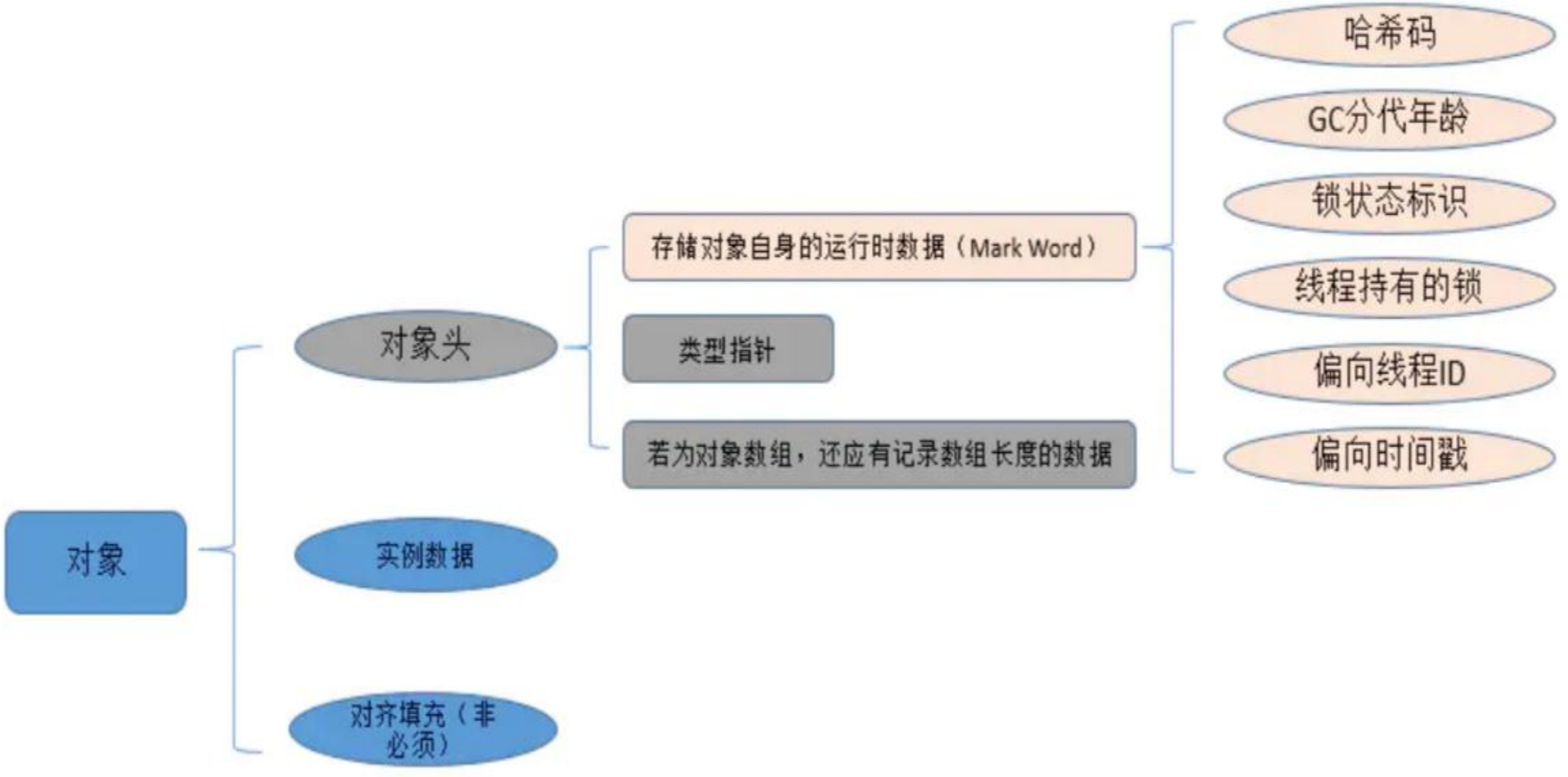
THANK YOU FOR WATCHING

主讲老师King：2962938812

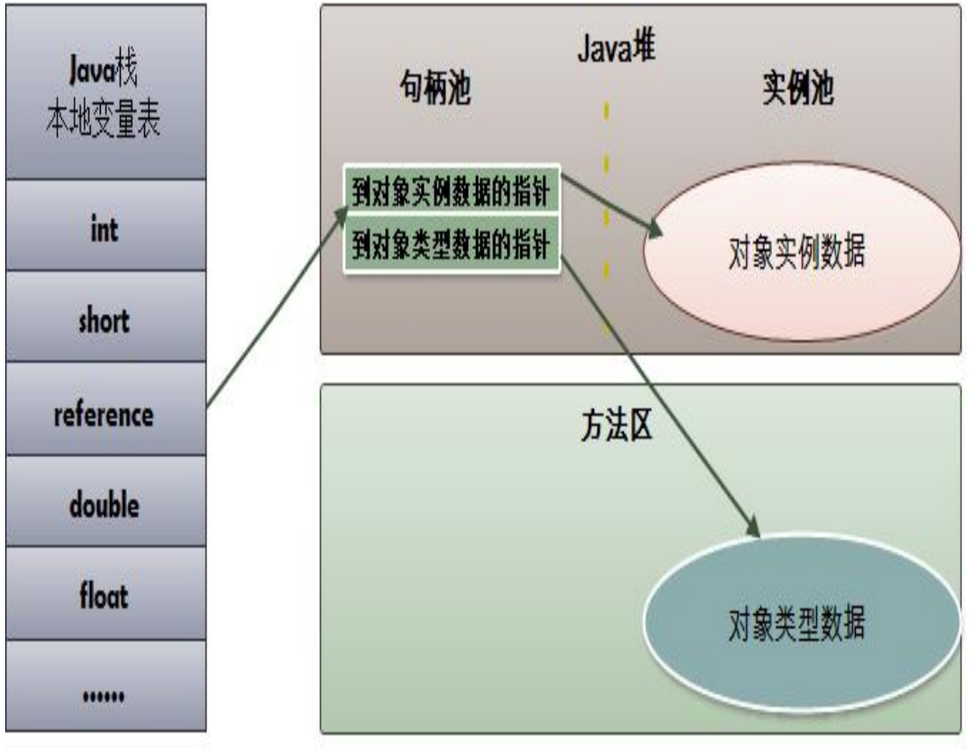
虚拟机中对象的创建过程



对象的内存布局

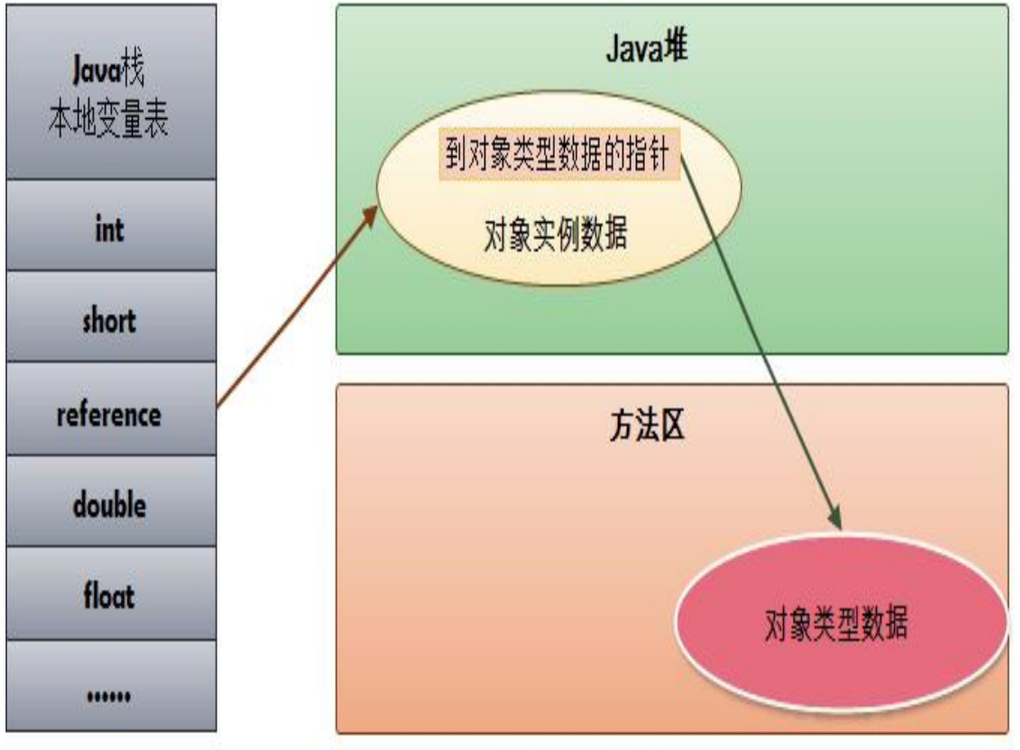

对象
对象头
存储对象自身的运行时数据（Mark Word）
哈希码
GC分代年龄
锁状态标识
线程持有的锁
偏向线程ID
偏向时间戳
类型指针
若为对象数组，还应有记录数组长度的数据
实例数据
对齐填充（非必须）

对象的访问定位



句柄方式访问对象

使用句柄

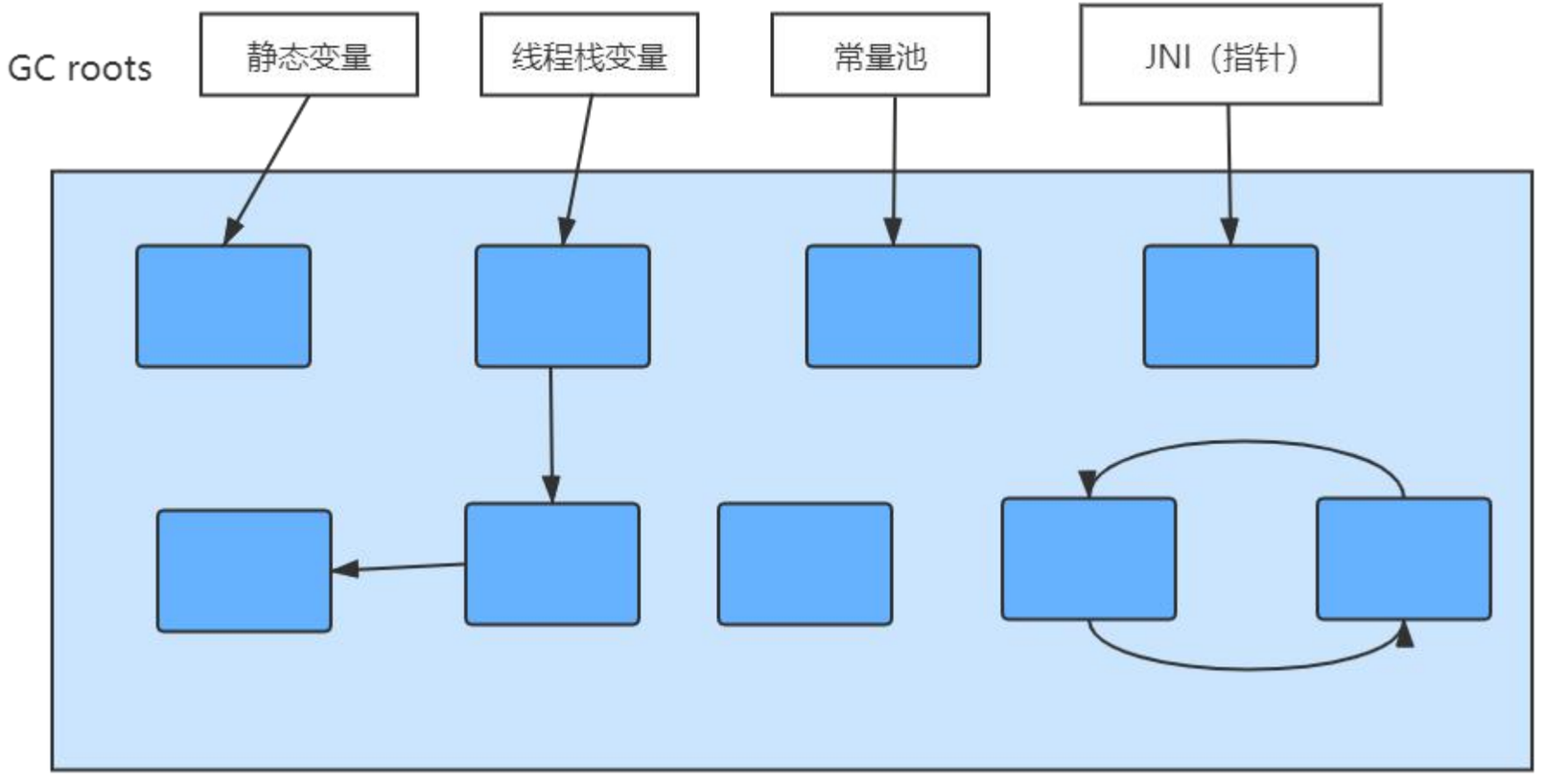


直接指针方式访问对象

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

学习垃圾回收的意义

- 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

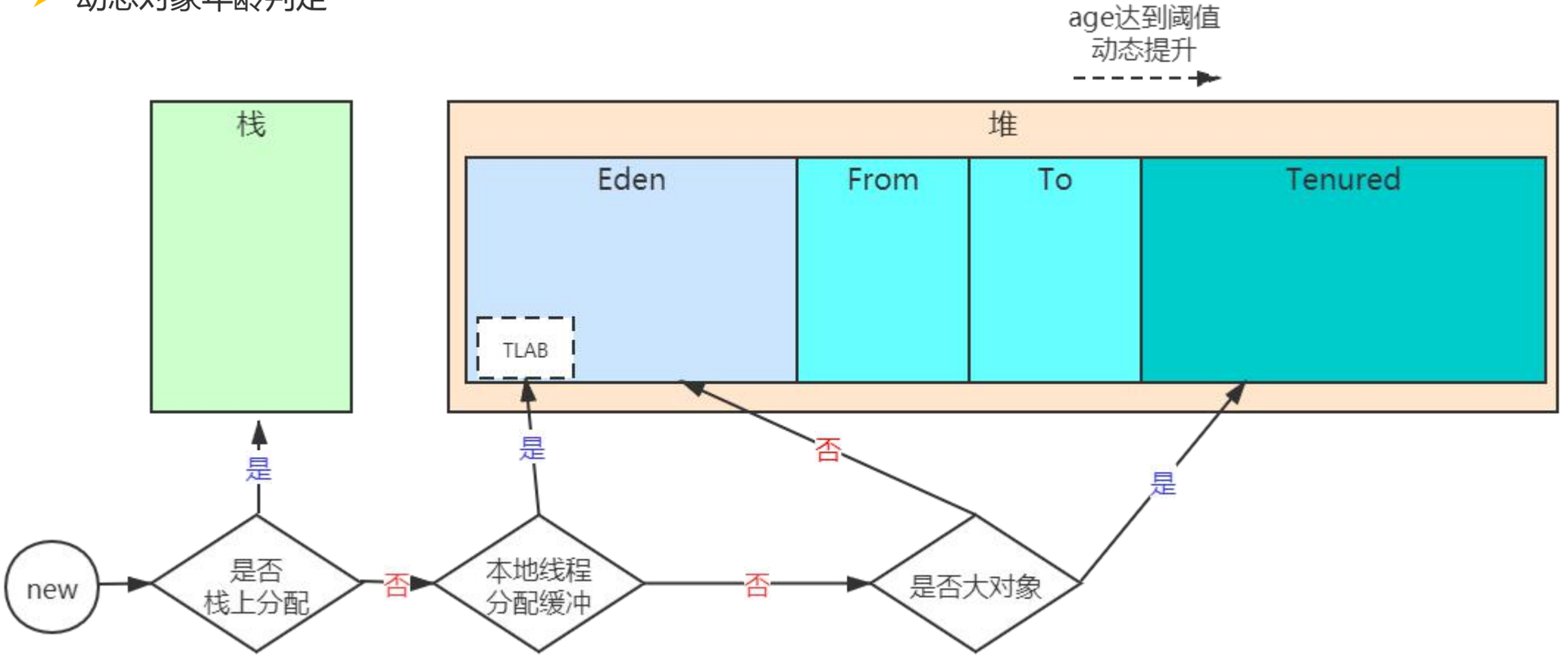
- 对象优先在Eden分配
- 空间分配担保
- 大对象直接进入老年代
- 长期存活的对象进入老年代
- 动态对象年龄判定

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)

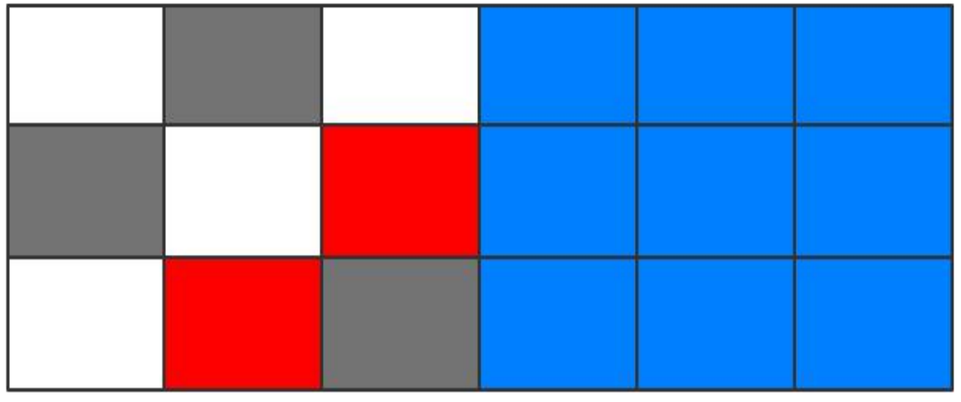


复制算法（Copying）

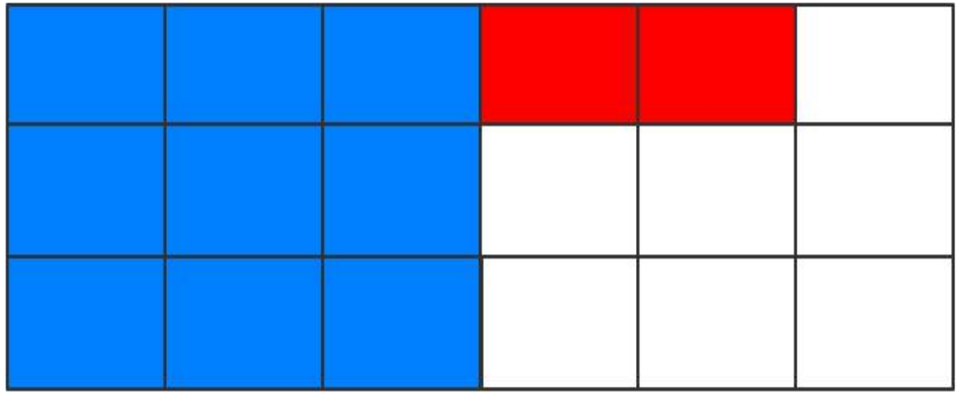
特点

- 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

回收前



回收后

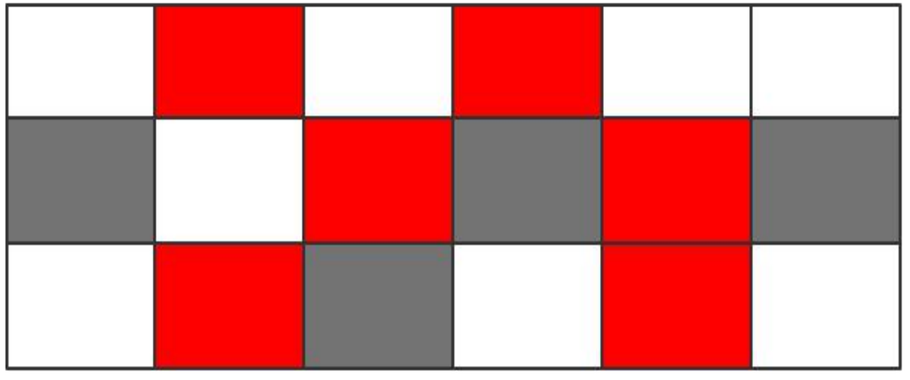


标记-清除算法（Mark-Sweep）

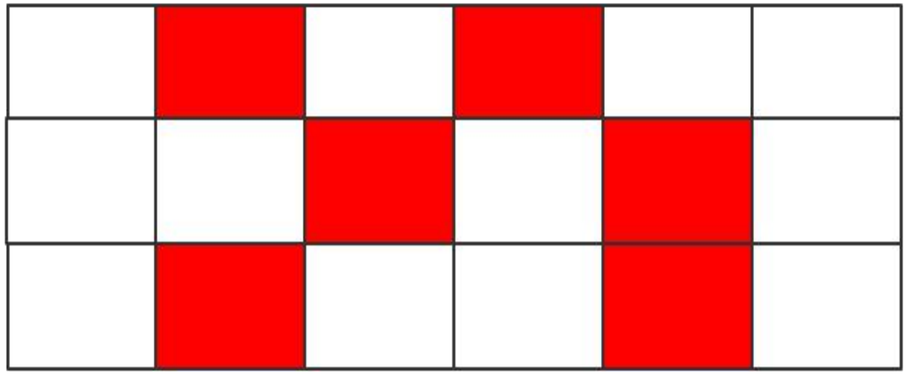
特点

- 利用率百分之百
- 不需要内存复制
- 有内存碎片

回收前



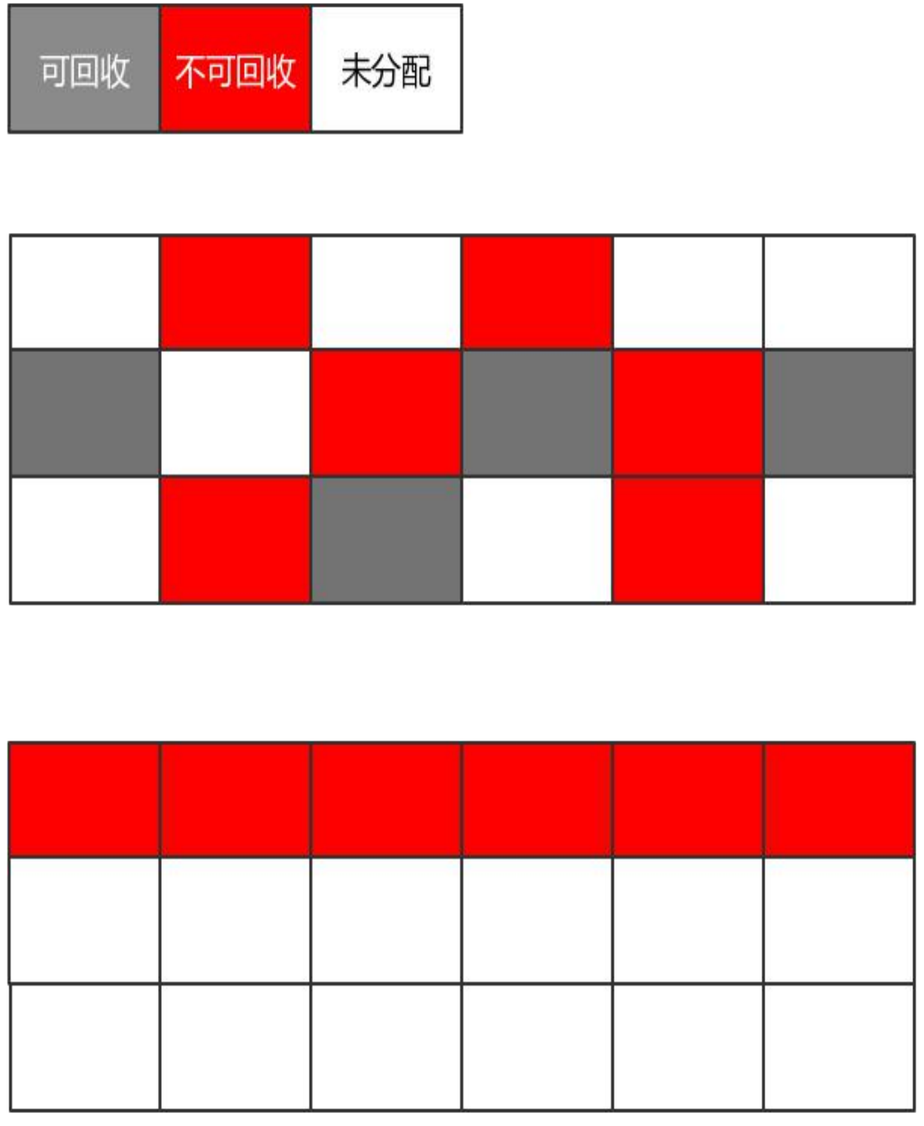
回收后



标记-整理算法（Mark-Compact）

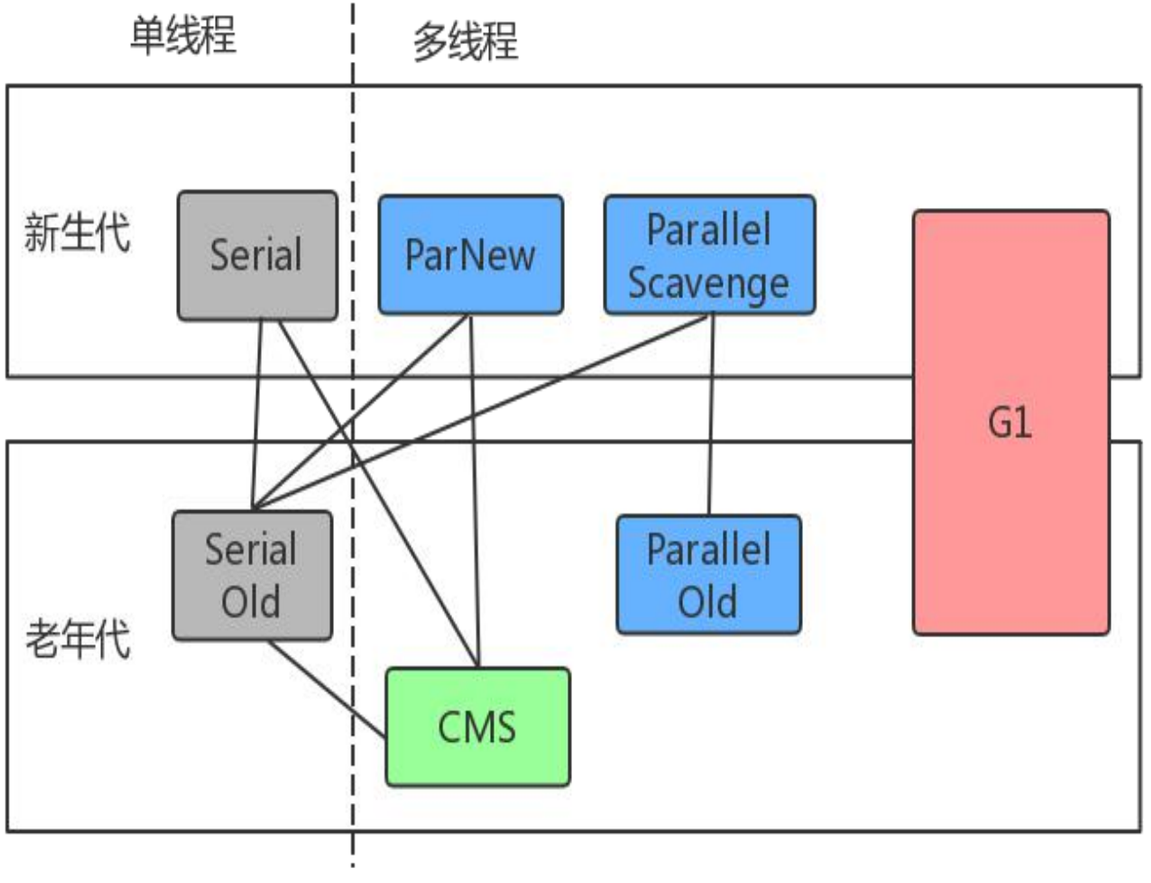
特点

- 利用率百分之百
- 没有内存碎片
- 需要内存复制
- 效率一般般



JVM中常见的垃圾收集器

- 单线程与多线程
- 并发收集器

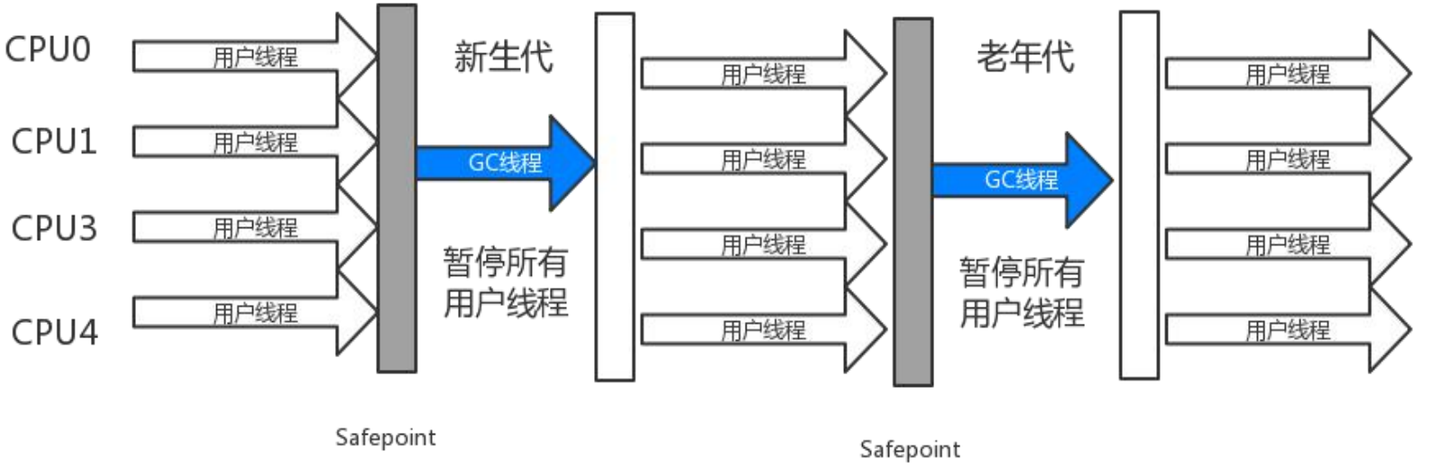


收集器	收集对象和算法	收集器类型
Serial	新生代，复制算法	单线程
ParNew	新生代，复制算法	并行的多线程收集器
Parallel Scavenge	新生代，复制算法	并行的多线程收集器

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

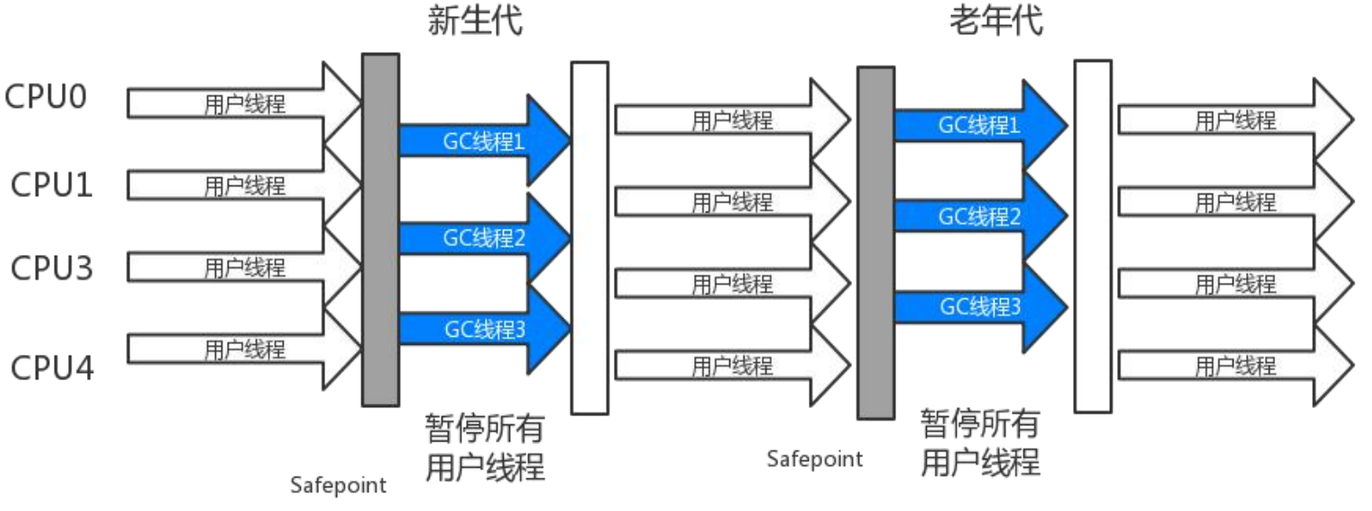
简单的垃圾回收器工作示意图

- 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

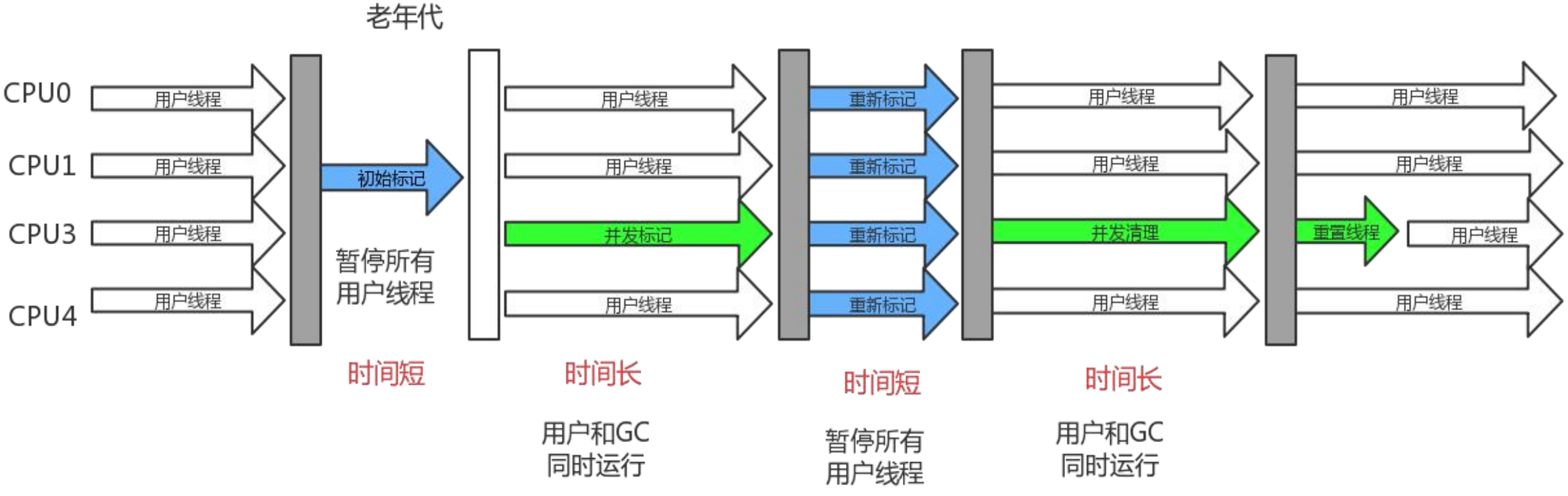
- 并行收集与并发收集

CMS收集器

标记清除算法 ---内存有碎片

- 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

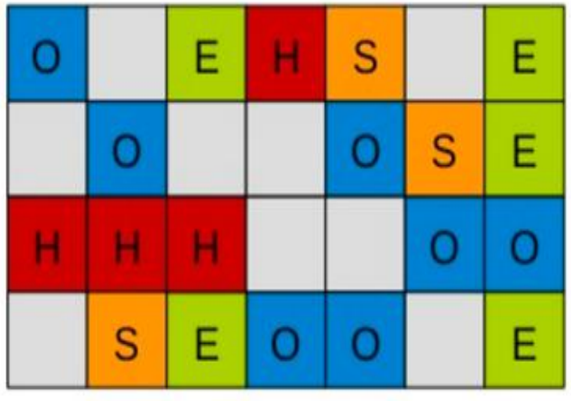
- 浮动垃圾

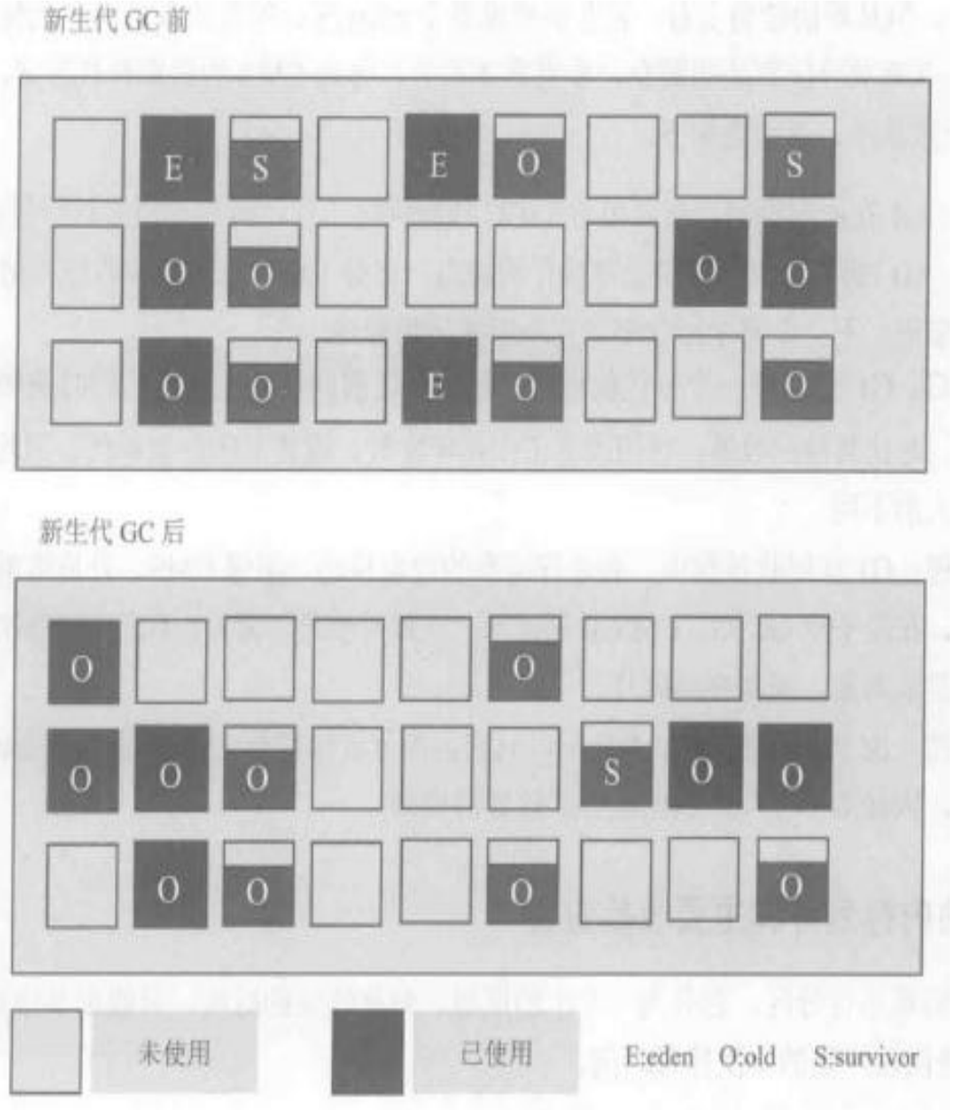


G1图示

G1收集的几个阶段

- 新生代GC
- 并发标记周期
- 混合收集
- 可能的FullGC





G1图示

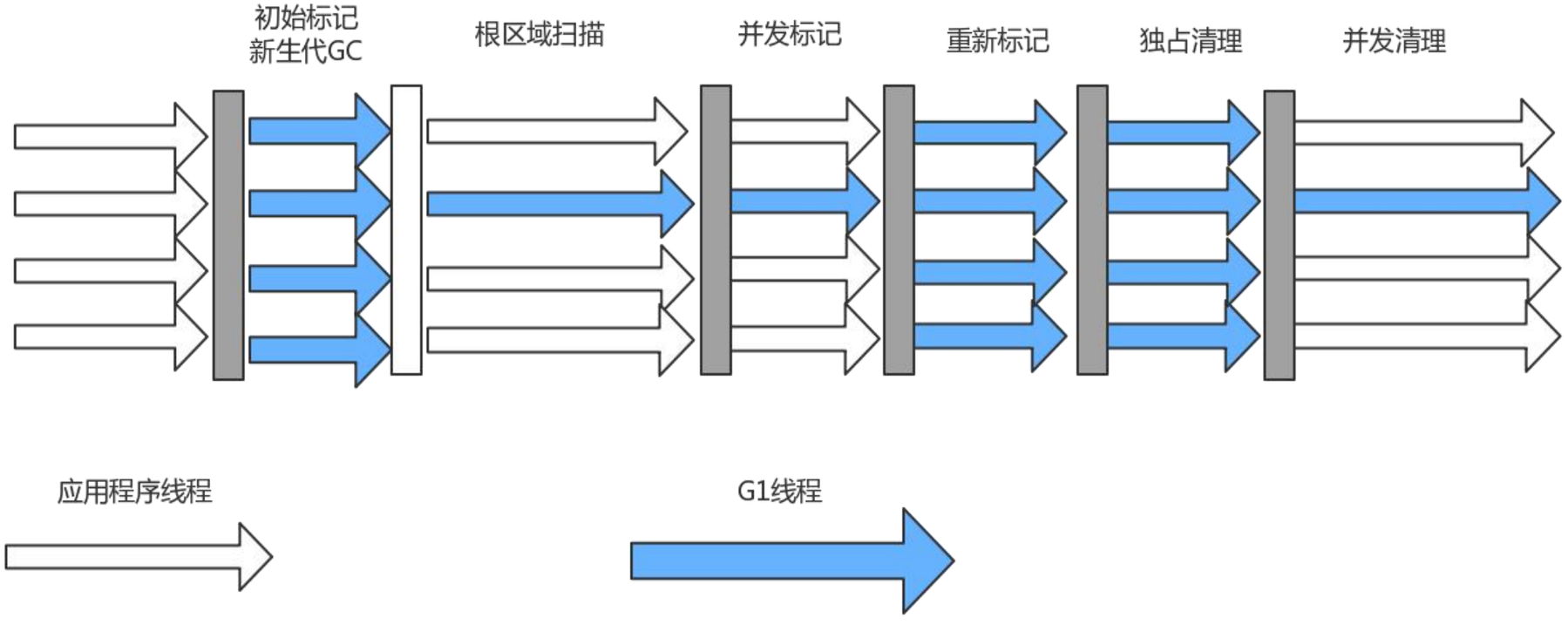
G1收集器并发标记运行示意图

JDK1.7才正式引入，采用分区回收的思维，

基本不牺牲吞吐量的前提下完成低停顿的内存回收；可预测的停顿是其最大的优势

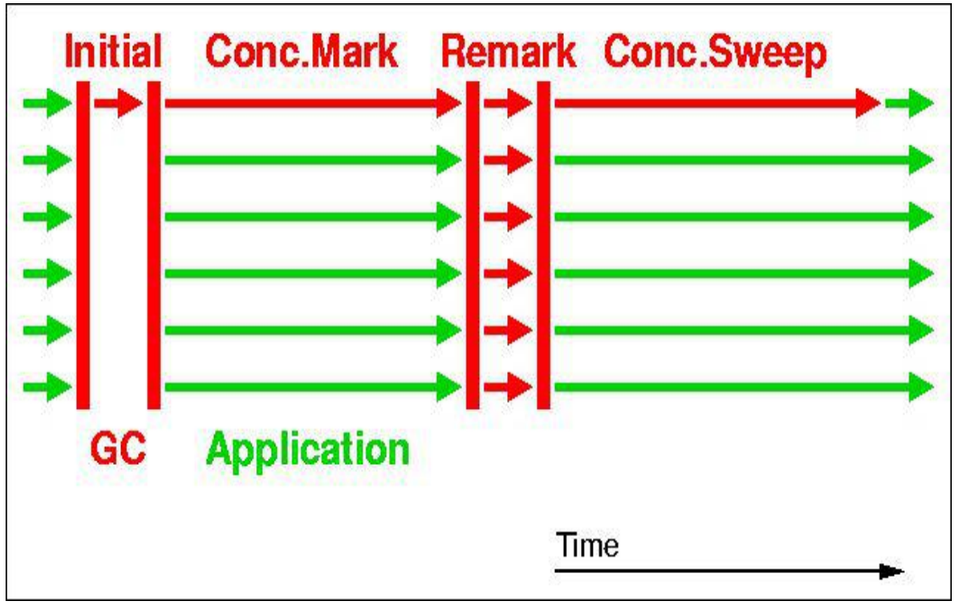
吞吐量= CPU的时间100% GC 10% 业务线程90% = 业务线程/总时间

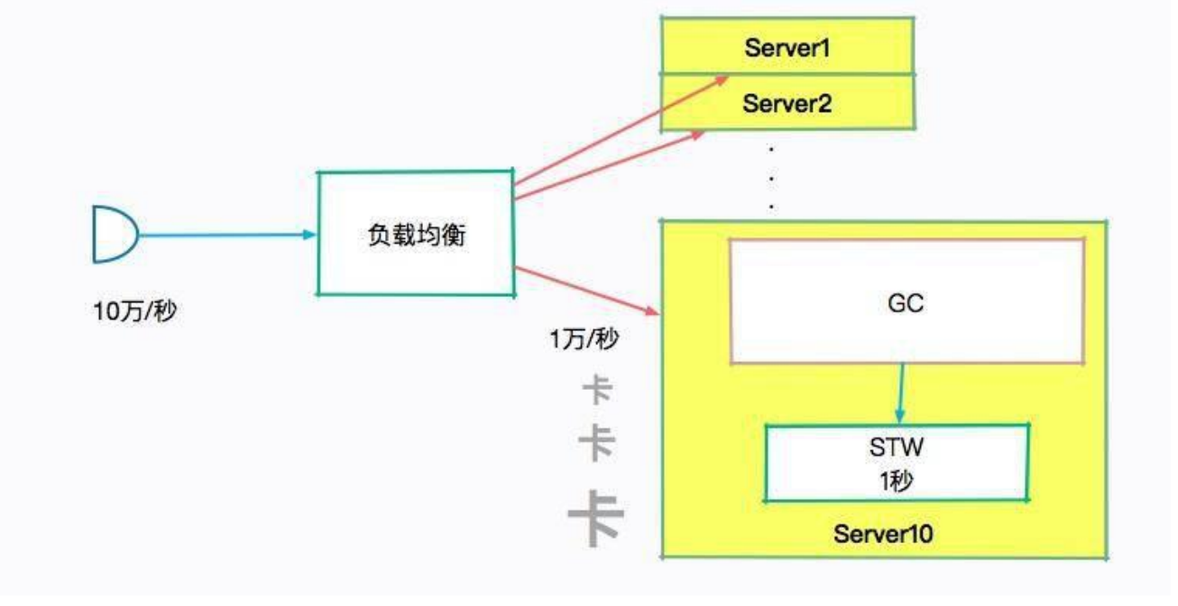
GC 98% 业务线程2%



Stop The World现象

- 什么是STW
- 为什么要STW
- STW的危害





内存泄漏和内存溢出辨析

- 相同与不同

- 如何避免内存泄漏

- MAT（GC ROOT）