马士兵教育

Garbage Collector
GC tuning

马士兵

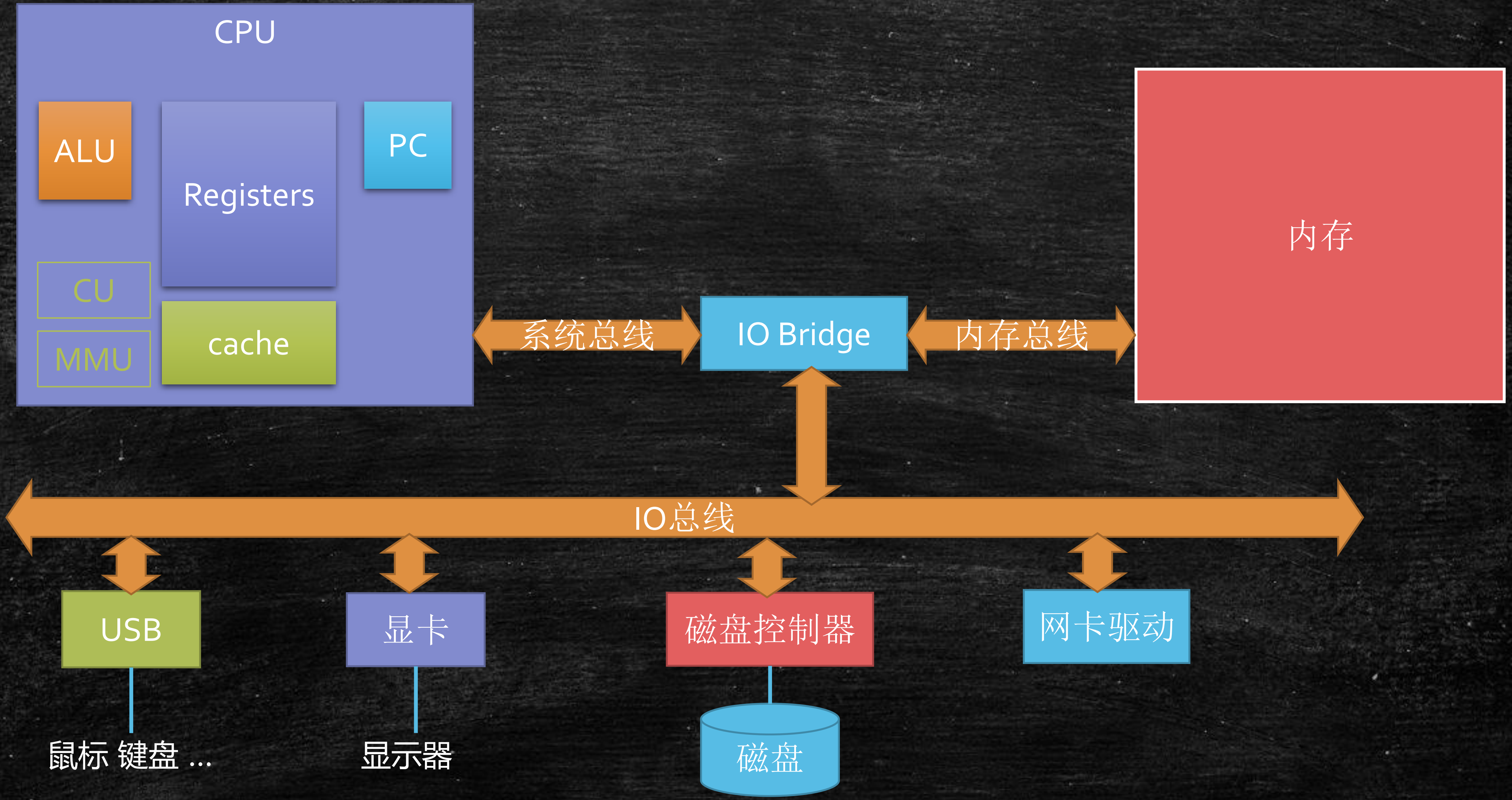
http://mashibing.com

熟悉GC常用算法，熟悉常见垃圾收集器，具有实际JVM调优实战经验

contents

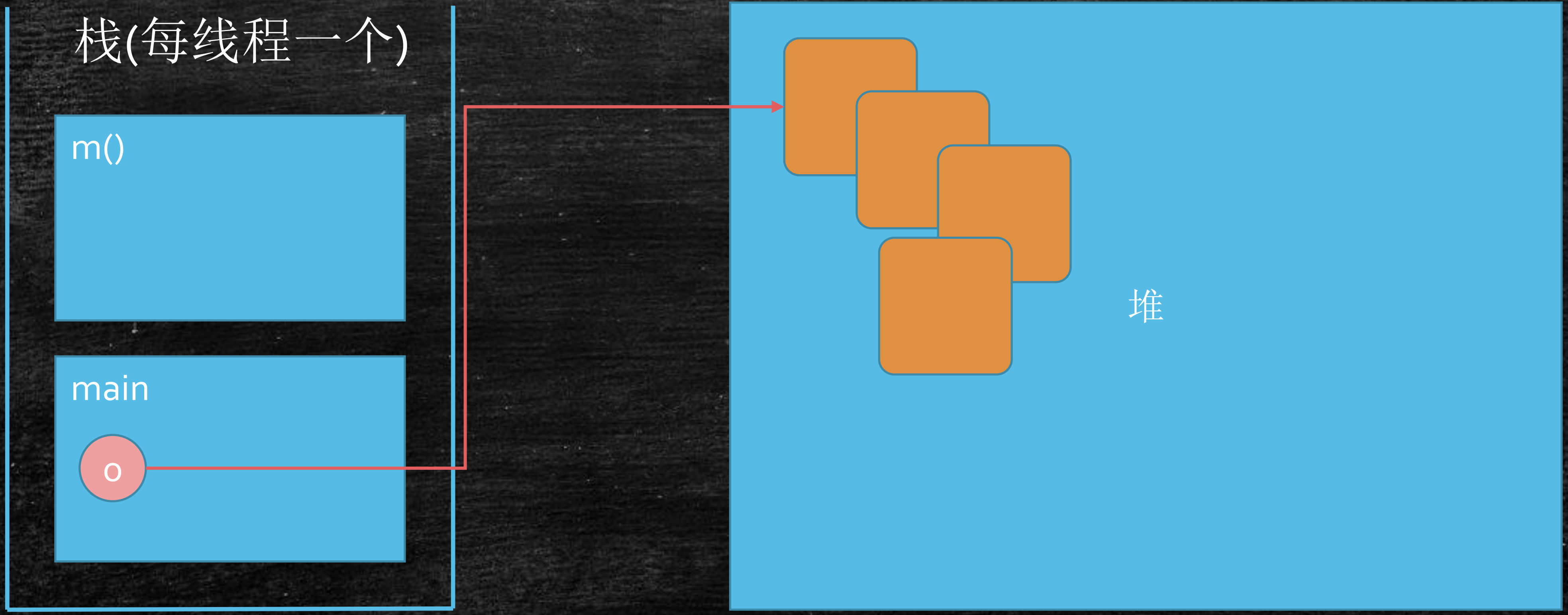
- what is garbage
- how to find it
- GC algorithms
- available collectors
- tuning

从本质谈起


CPU
ALU
Registers
PC
CU
MMU
cache
系统总线
IO Bridge
内存总线
内存
IO总线
USB
显卡
磁盘控制器
网卡驱动
鼠标 键盘 ...
显示器
磁盘

程序的栈（栈帧 stack frame）和堆

```
main {
    Object o = new Object();
    m();
}
```



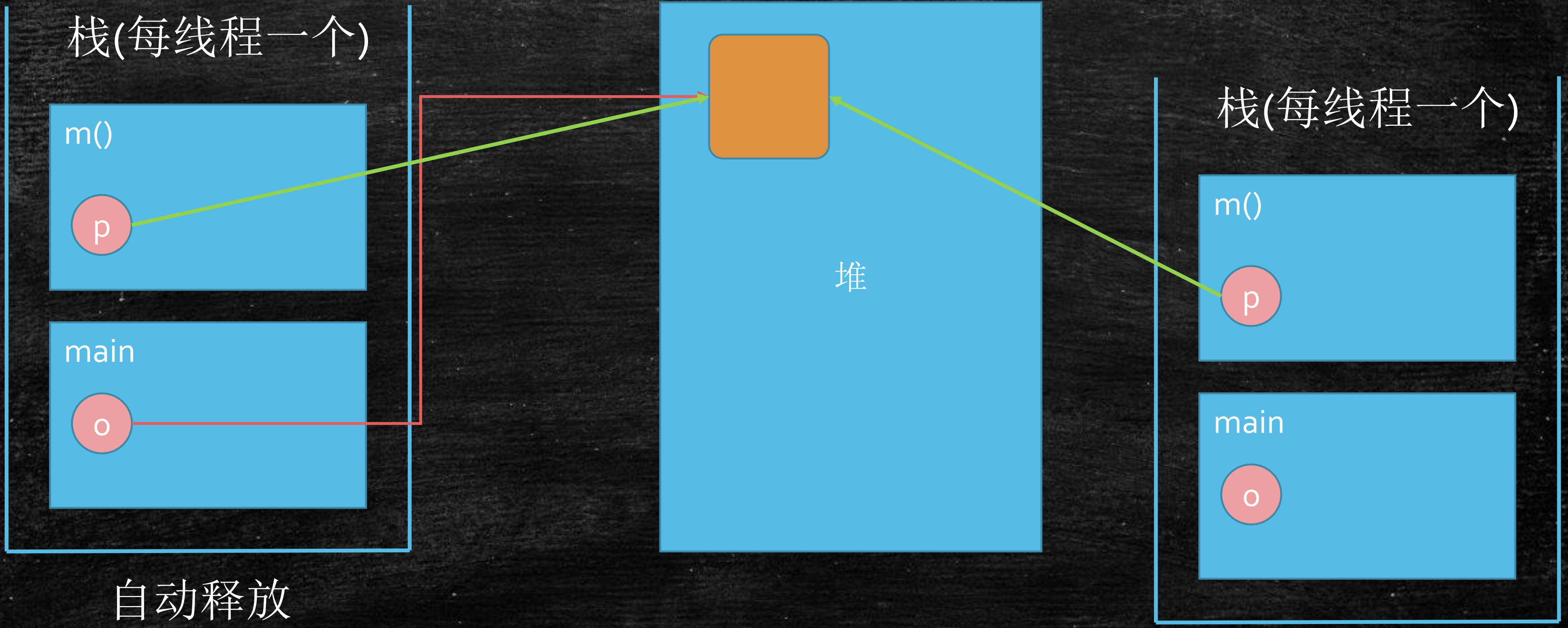
自动释放

最难调试的bug

- 野指针
  - 同一个对象，两个指针，一个释放了，另外一个不知道还拿来用
  - 同一个指针，不同位置，
  - 不再指向任何对象的指针
  - NullPointerExcetion
- 并发问题
  - 多线程访问同一块儿内存空间

程序的栈（栈帧 stack frame）和堆

```
main {
   Object o = new Object();
   m();
}
```



http://mashibing.com

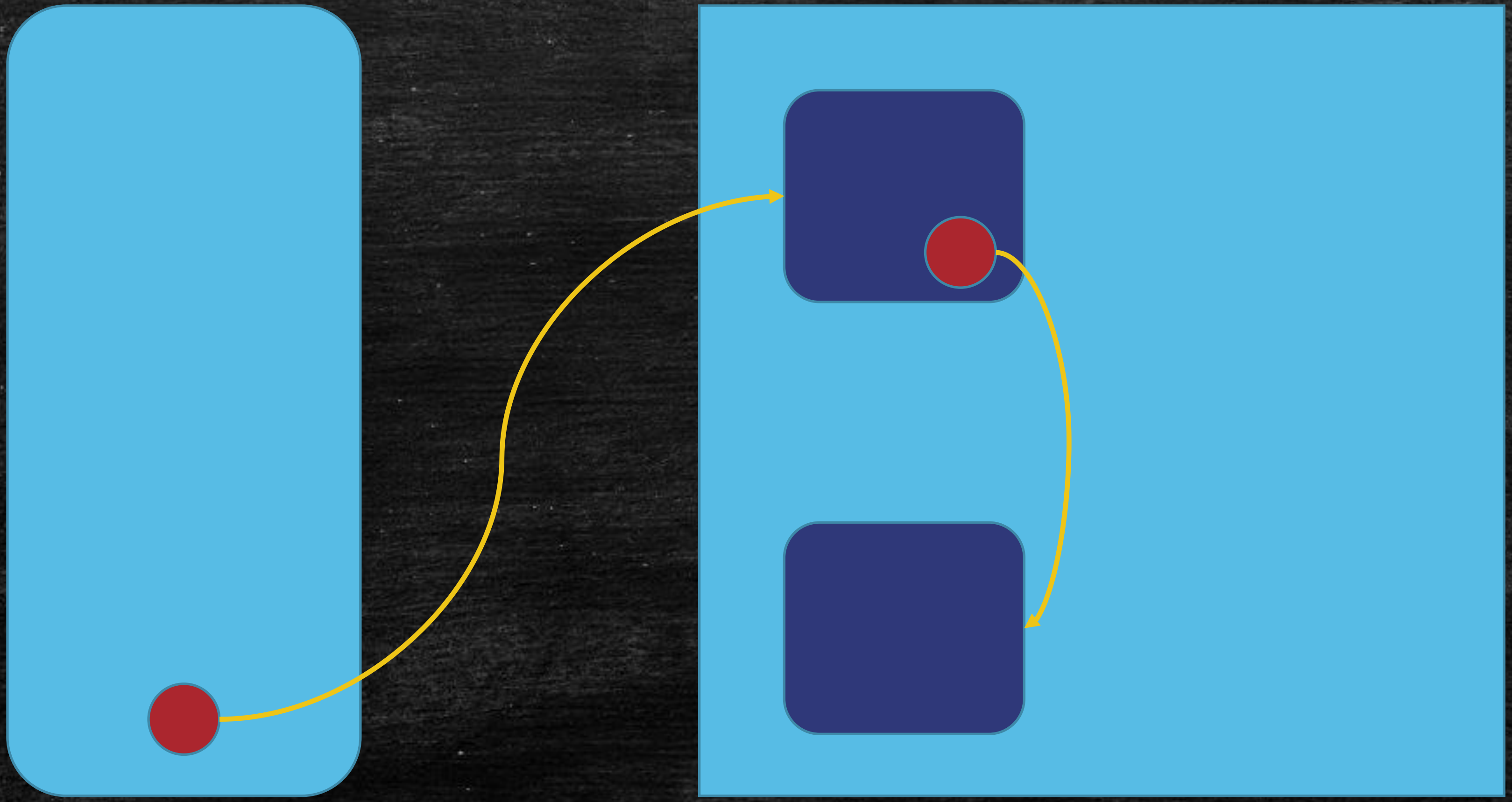
语言的发展历史

- c / c++
  - 手工管理 malloc free / new delete
  - 忘记释放 – memory leak – out of memory
  - 释放多次 产生极其难易调试的bug，一个线程空间莫名其妙被另外一个释放了
  - 开发效率很低
- java python go js kotlin scala
  - 方便内存管理的语言
  - GC – Garbage Collector – 应用线程只管分配，垃圾回收器负责回收
  - 大大减低程序员门槛
- rust
  - 运行效率超高（ asm c c++)
  - 不用手工管理内存（没有GC）
  - 学习曲线巨高 （ownership）
  - 你只要程序语法通过，就不会有bug

聊聊JVM的GC历史

- 三种算法都有毛病，三种的综合运用，诞生了各种各样的垃圾回收器

garbage?



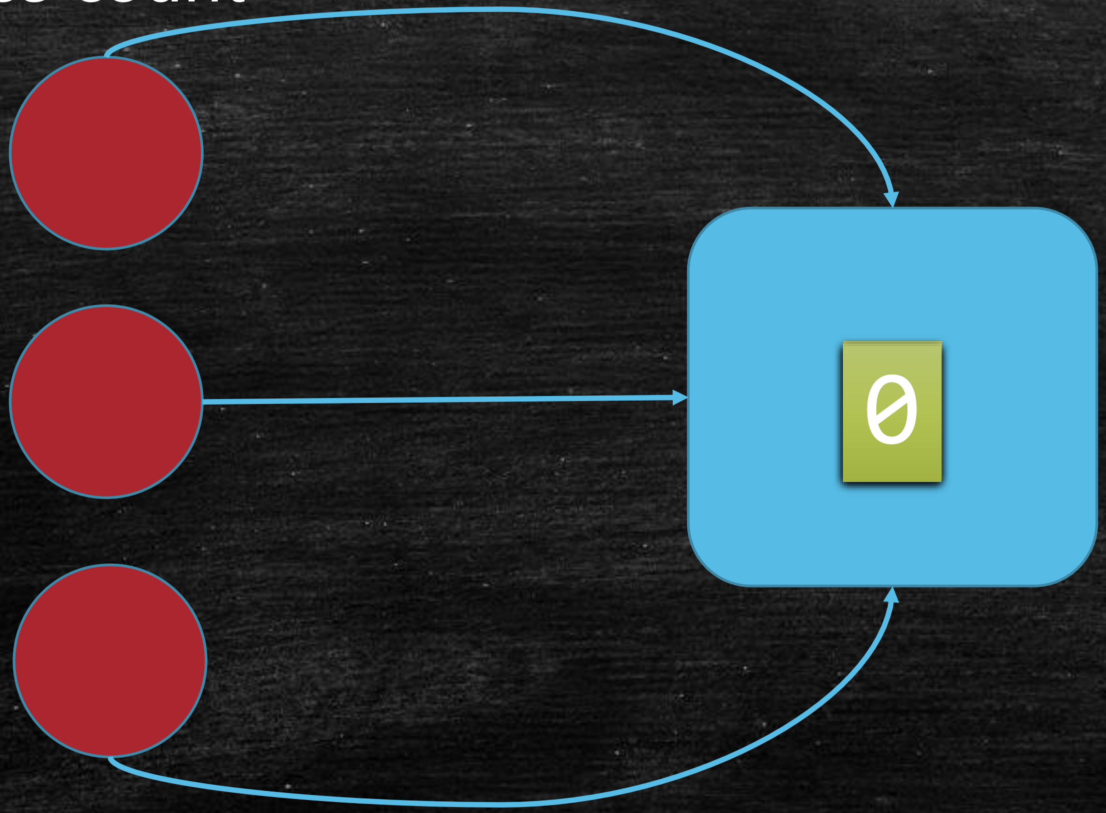
http://mashibing.com

java vs c++

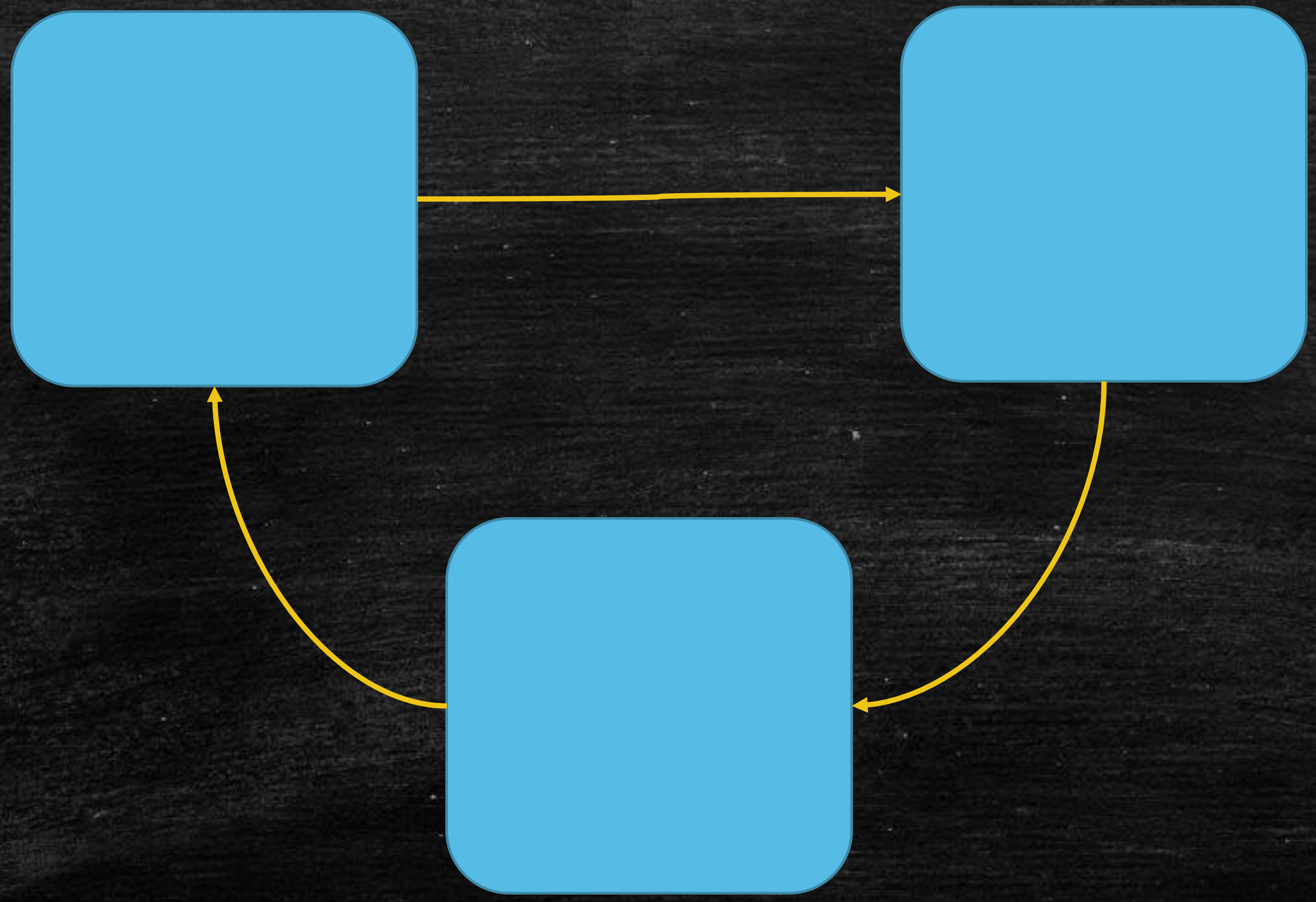
- java
  - GC处理垃圾
  - 开发效率高，执行效率低
- C++
  - 手工处理垃圾
  - 忘记回收
    - 内存泄漏
  - 回收多次
    - 非法访问
  - 开发效率低，执行效率高

how to find a garbage?

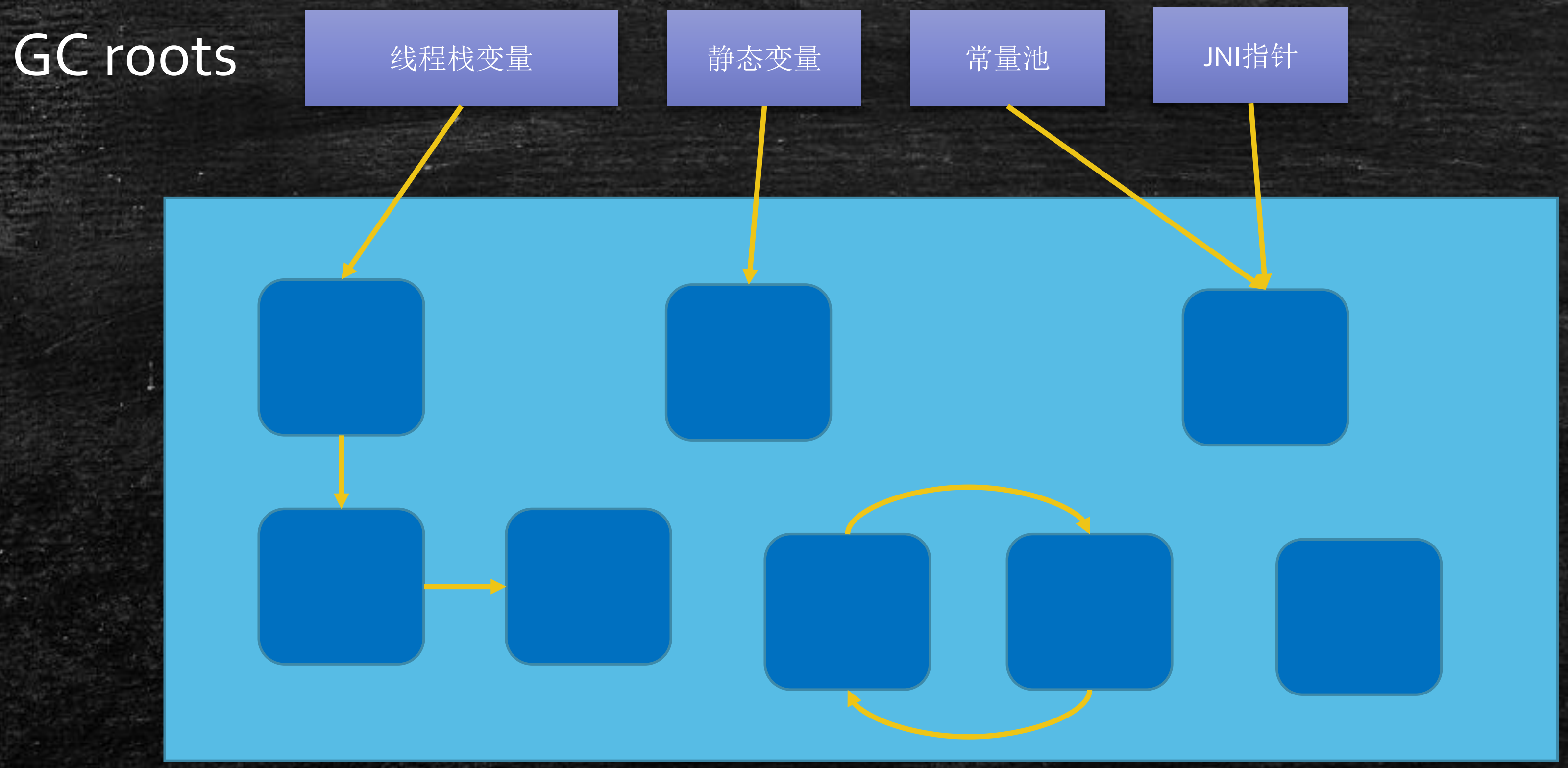
- reference count



RC can't resolve:



Root Searching

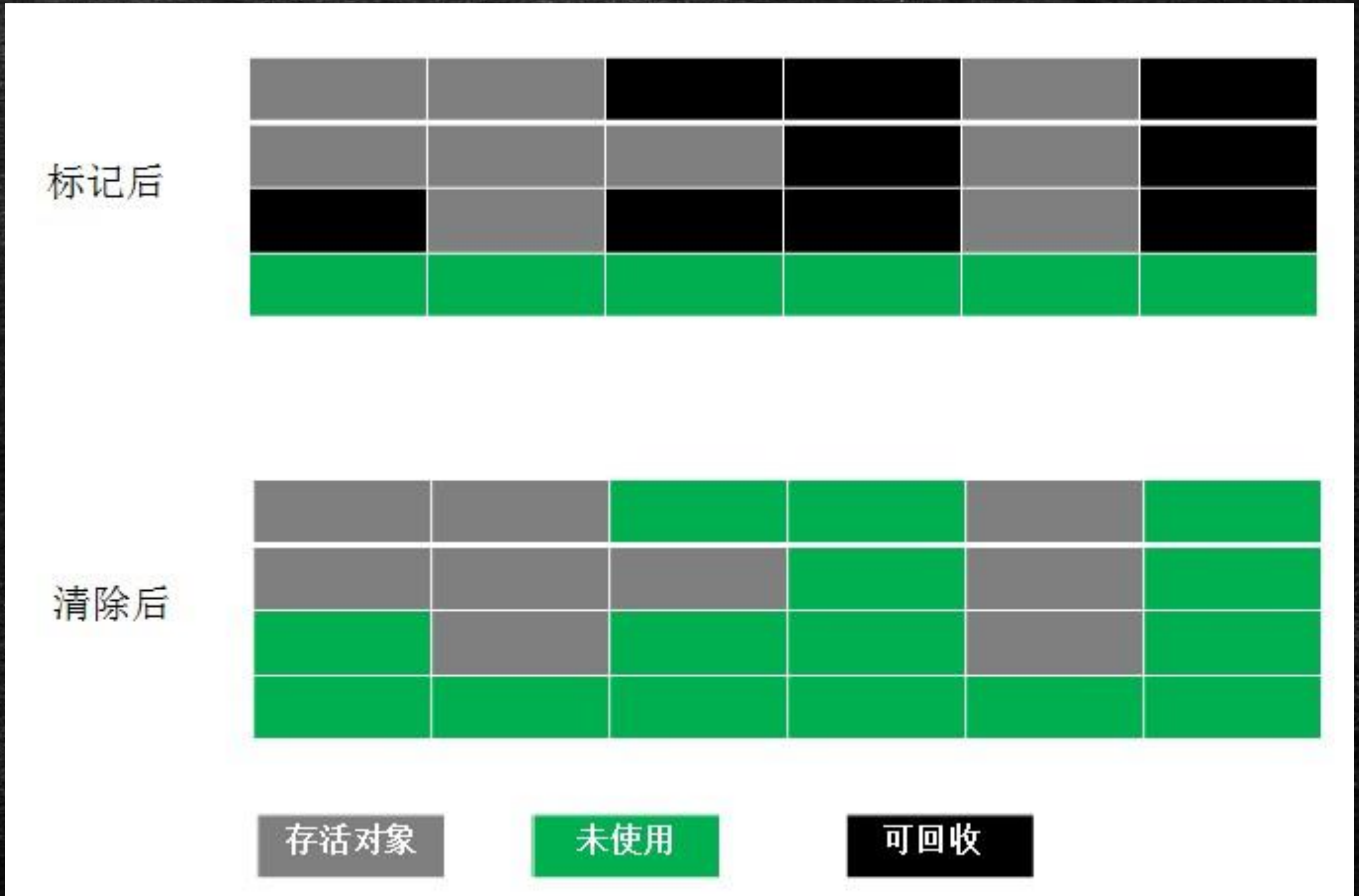


which instances are roots?
JVM stack, native method stack, run-time constant pool, static references in method area, Clazz

GC Algorithms

- Mark-Sweep（标记清除）
- Copying（拷贝）
- Mark-Compact（标记压缩）

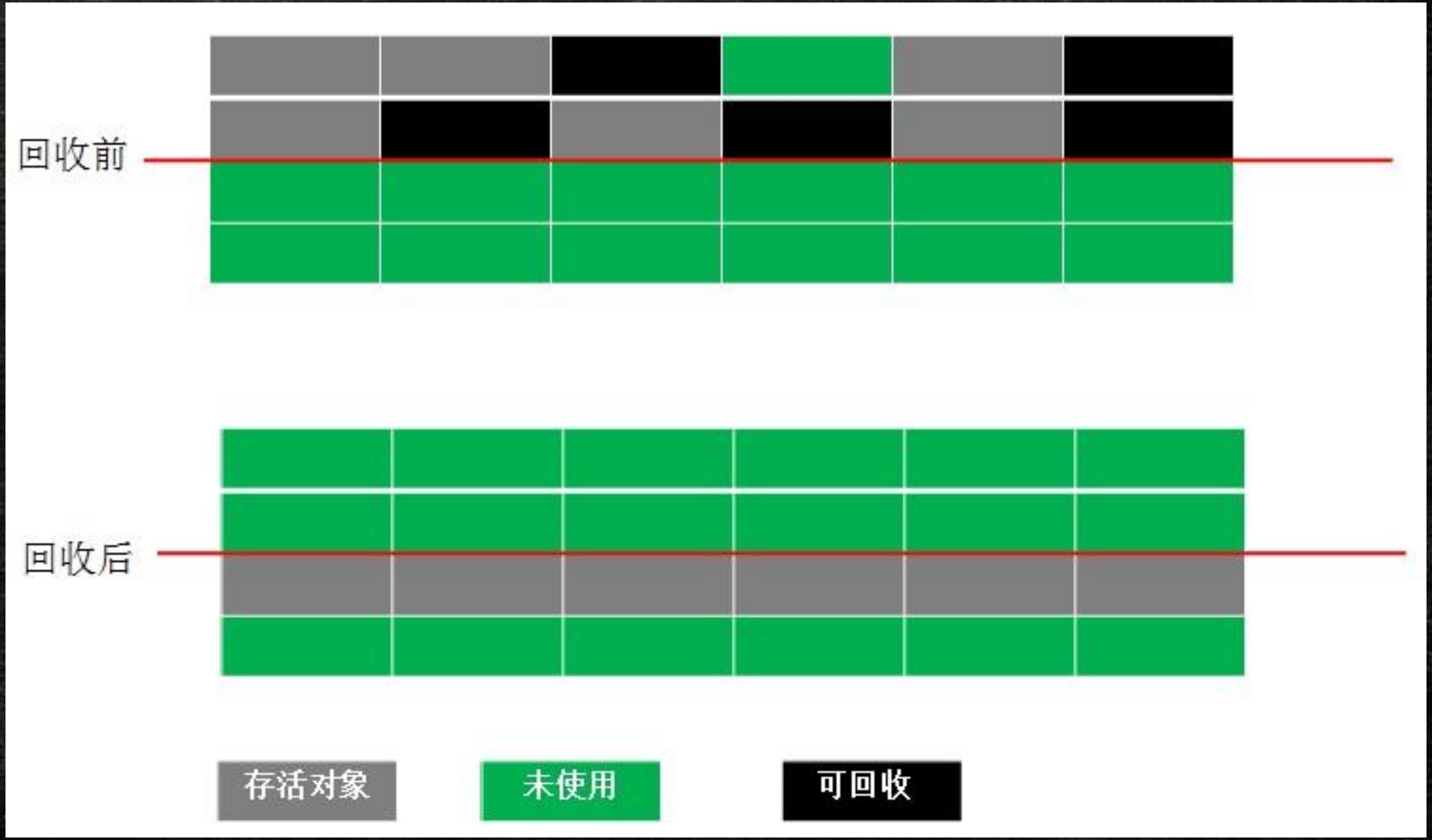
Mark-Sweep



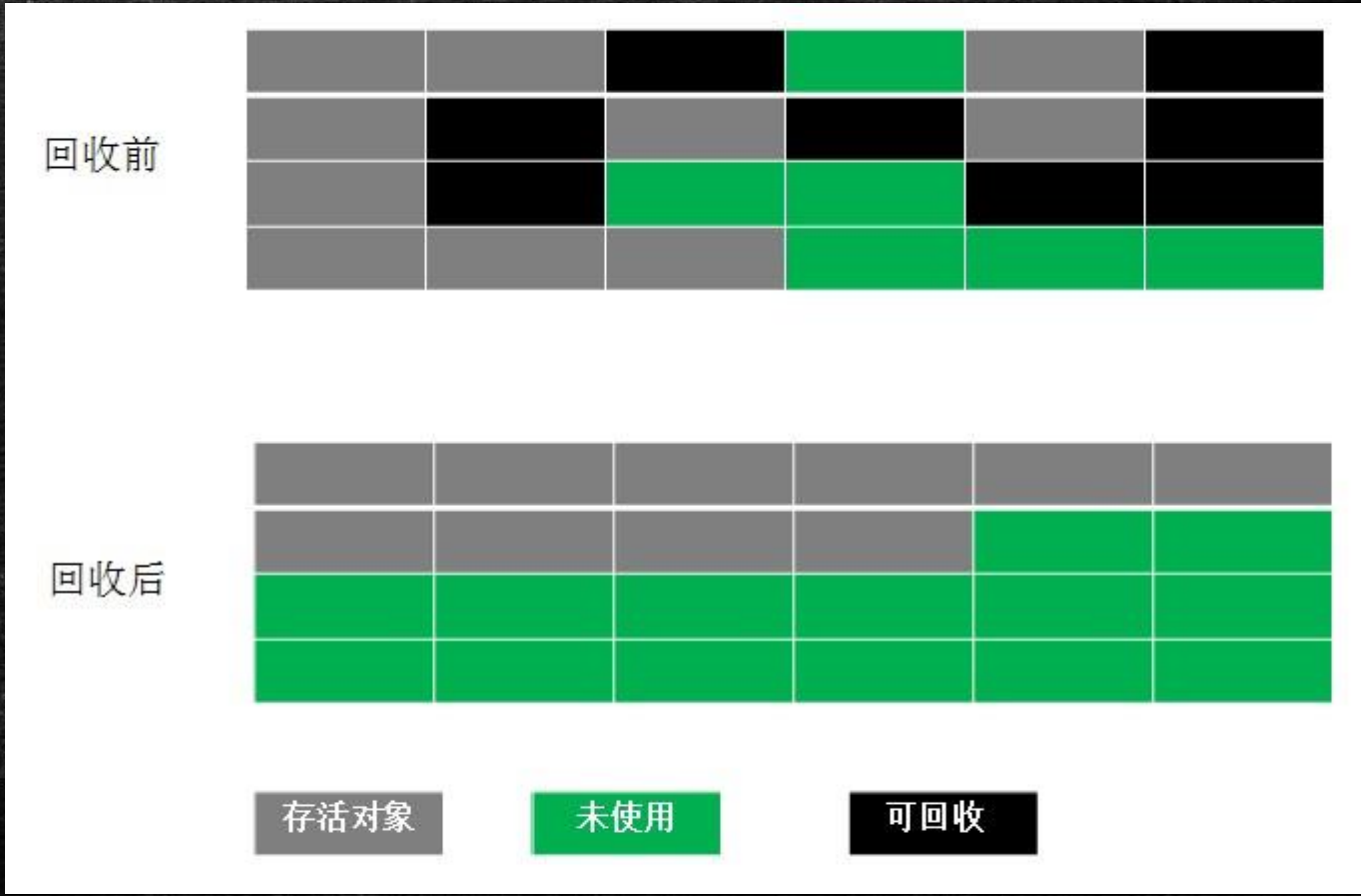
碎片化

Copying

内存浪费

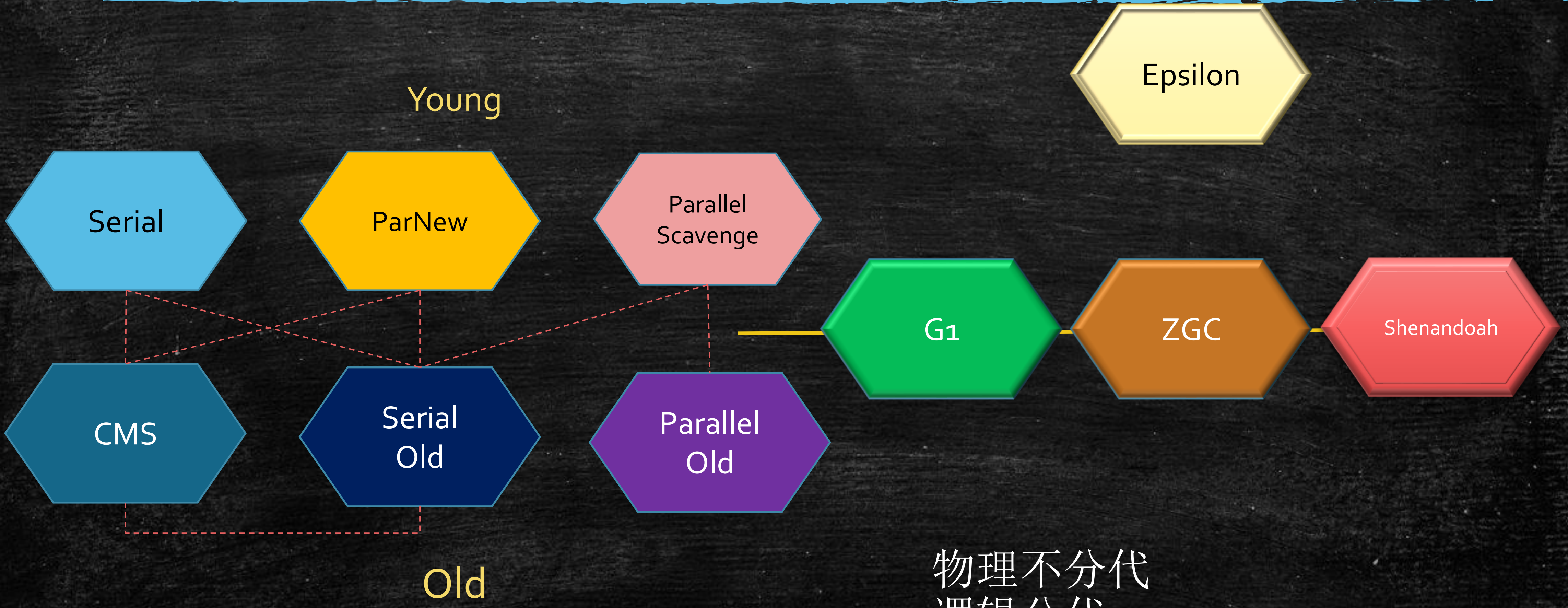


Mark-Compact



效率比copy略低

Garbage Collectors



物理不分代
逻辑分代

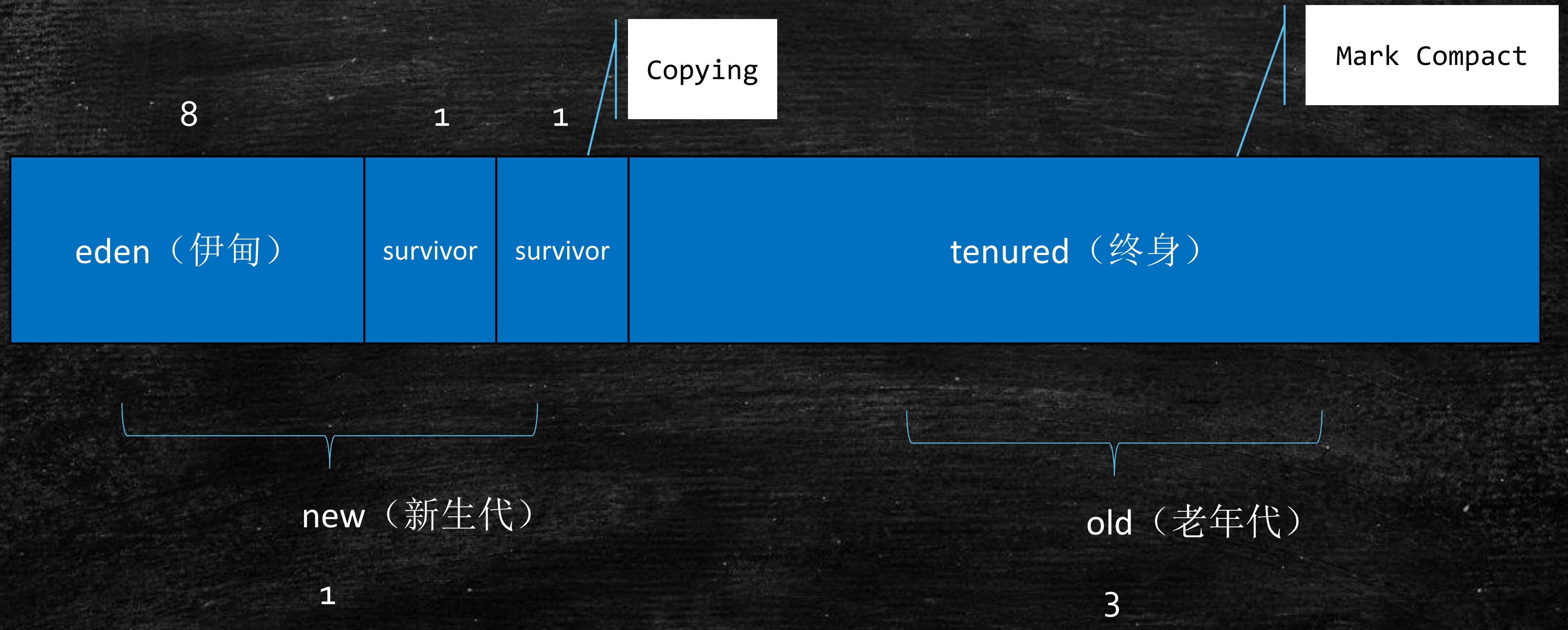
GC的演化

- 随着内存大小的不断增长而演进
- 几兆 – 几十兆
  - Serial 单线程STW垃圾回收 年青代 老年代
- 几十兆 – 上百兆1G
  - parallel 并行多线程
- - 几十G
  - Concurrent GC

JVM分代算法

- new – young
  - 存活对象少
  - 使用copy算法，效率高
- old
  - 垃圾少
  - 一般使用mark compact
  - g1使用copy

堆内存逻辑分区



Copying
Mark Compact
8
1
1
eden（伊甸）
survivor
survivor
tenured（终身）
new（新生代）
1
old（老年代）
3

一个对象从出生到消亡

不分代行不行?

详解

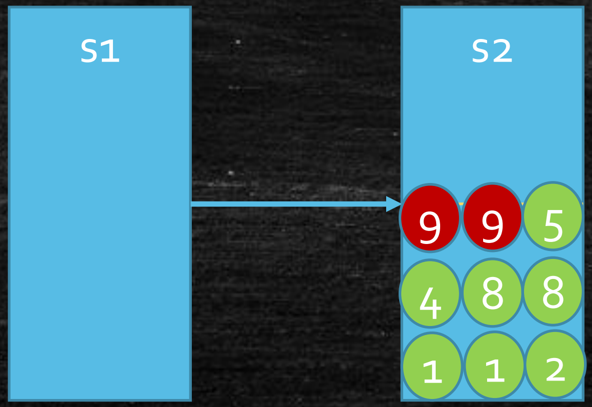
- 栈上分配
  - 线程私有小对象
  - 无逃逸
  - 支持标量替换
  - 无需调整
- 线程本地分配TLAB （Thread Local Allocation Buffer）
  - 占用eden，默认1%
  - 多线程的时候不用竞争eden就可以申请空间，提高效率
  - 小对象
  - 无需调整
- 老年代
  - 大对象
- eden

YGC FGC

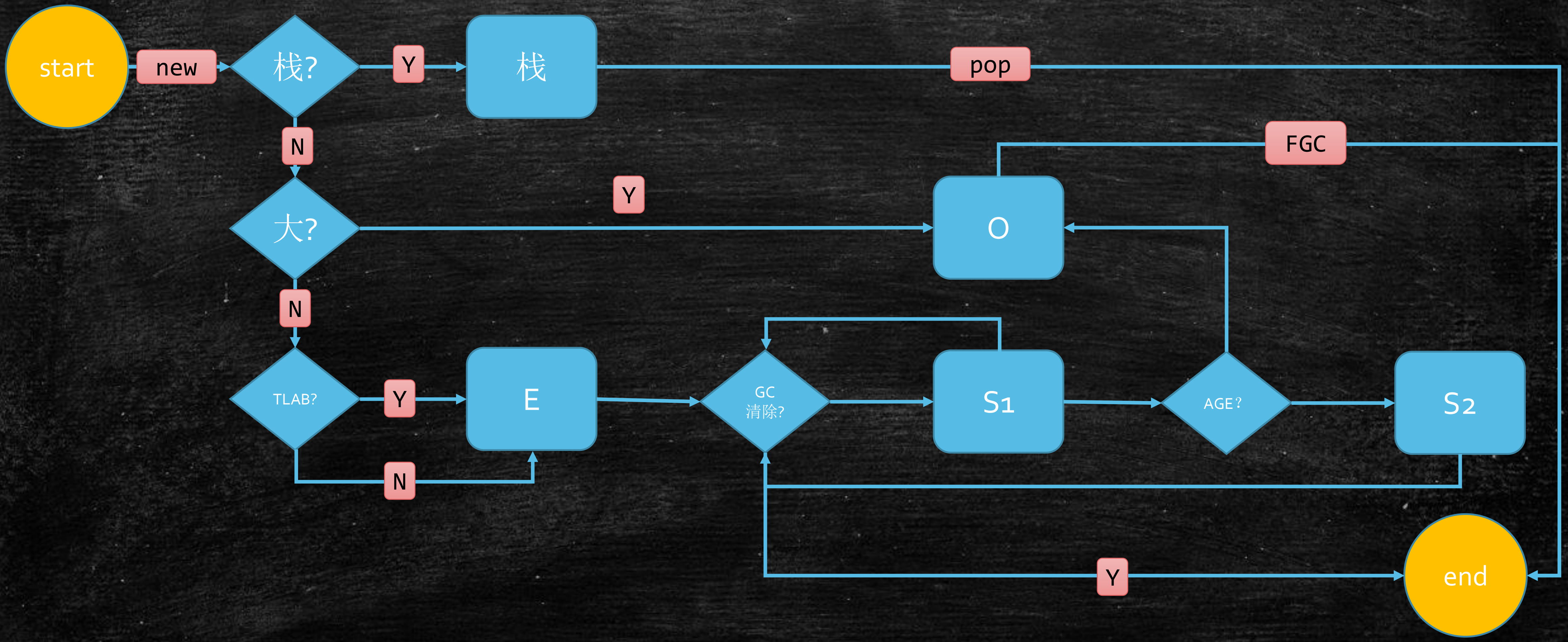
- YGC
  - Young GC Minor GC
  - Eden区不足
- FGC
  - Full GC Major GC
  - Old空间不足
  - System.gc()
  - …

对象何时进入老年代

- 超过 XX:MaxTenuringThreshold 指定次数（YGC）
  - Parallel Scavenge 15
  - CMS 6
  - G1 15
- 动态年龄
  - s1 - > s2超过50%
  - 把年龄最大的放入O

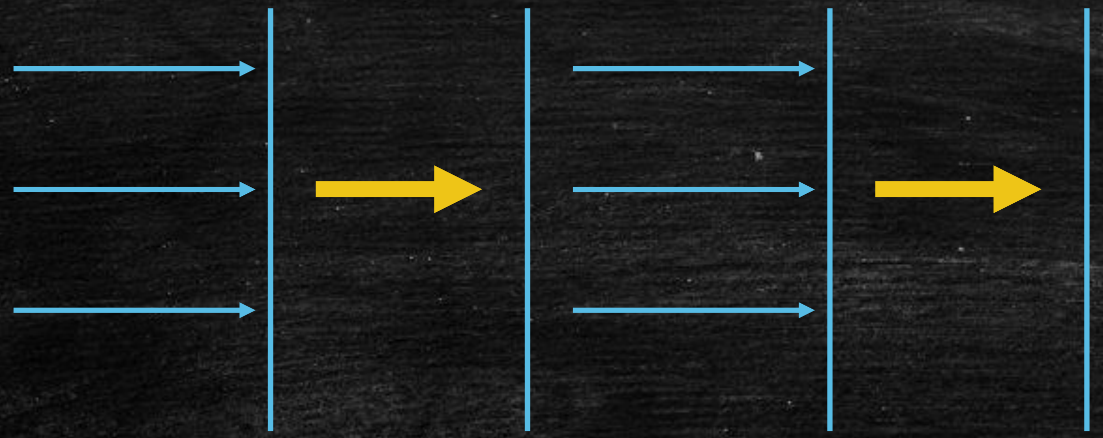


总结



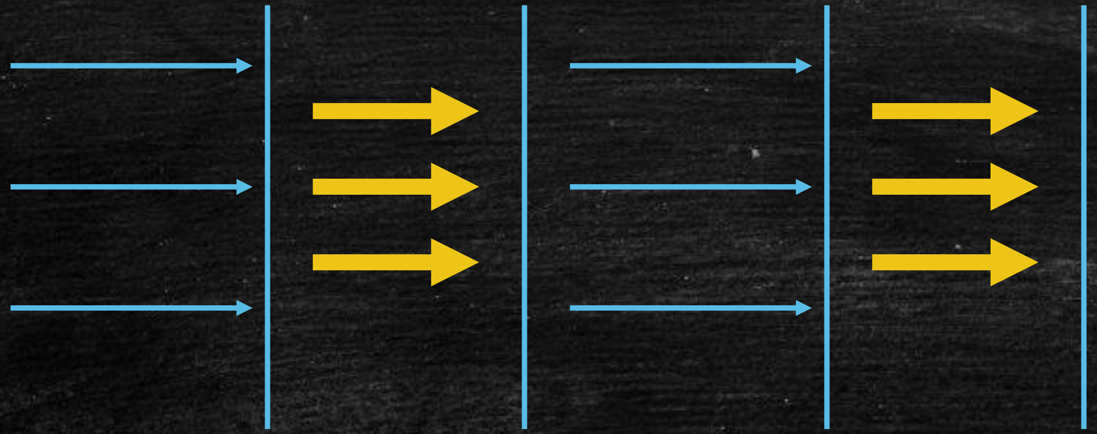
Serial

- a stop-the-world, copying collector which uses a single GC thread



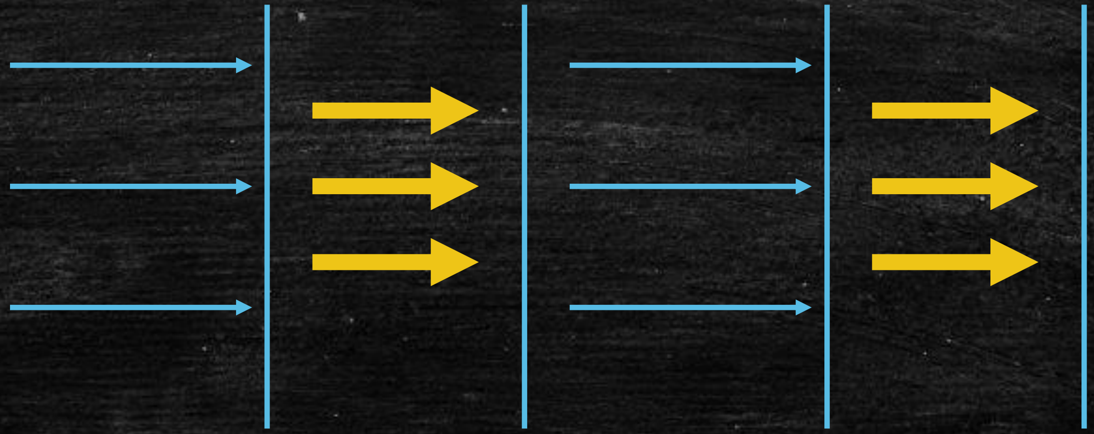
Parallel Scavenge

- a stop-the-world, copying collector which uses multiple GC threads



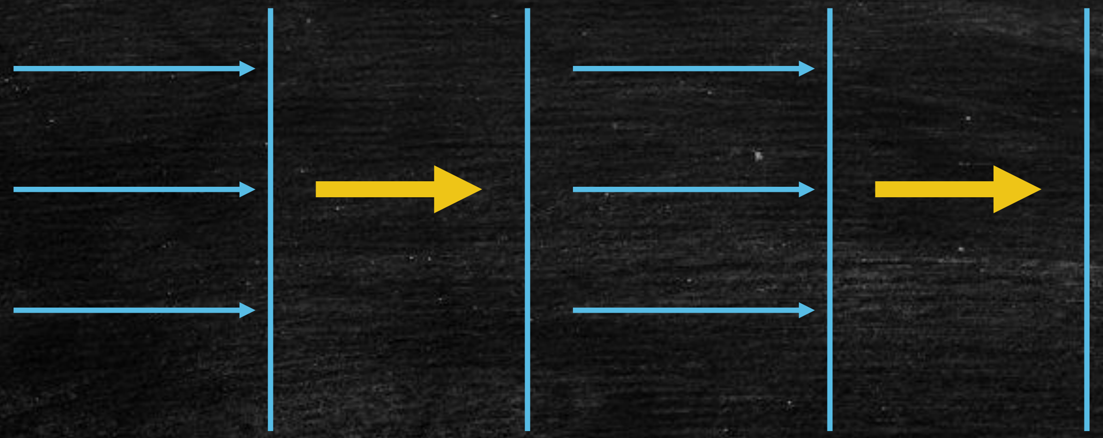
ParNew

- a stop-the-world, copying collector which uses multiple GC threads
- It differs
  from "Parallel Scavenge" in that it has enhancements that make it usable with
  CMS
- For example, "ParNew" does the
  synchronization needed so that it can run during the
  concurrent phases of CMS.



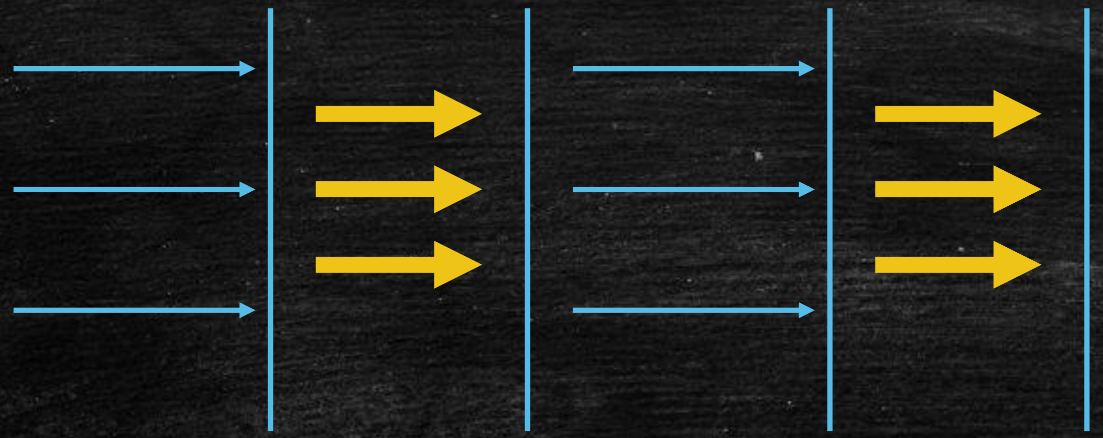
Serial Old

- a stop-the-world,
  mark-sweep-compact collector that uses a single GC thread.



parallel old

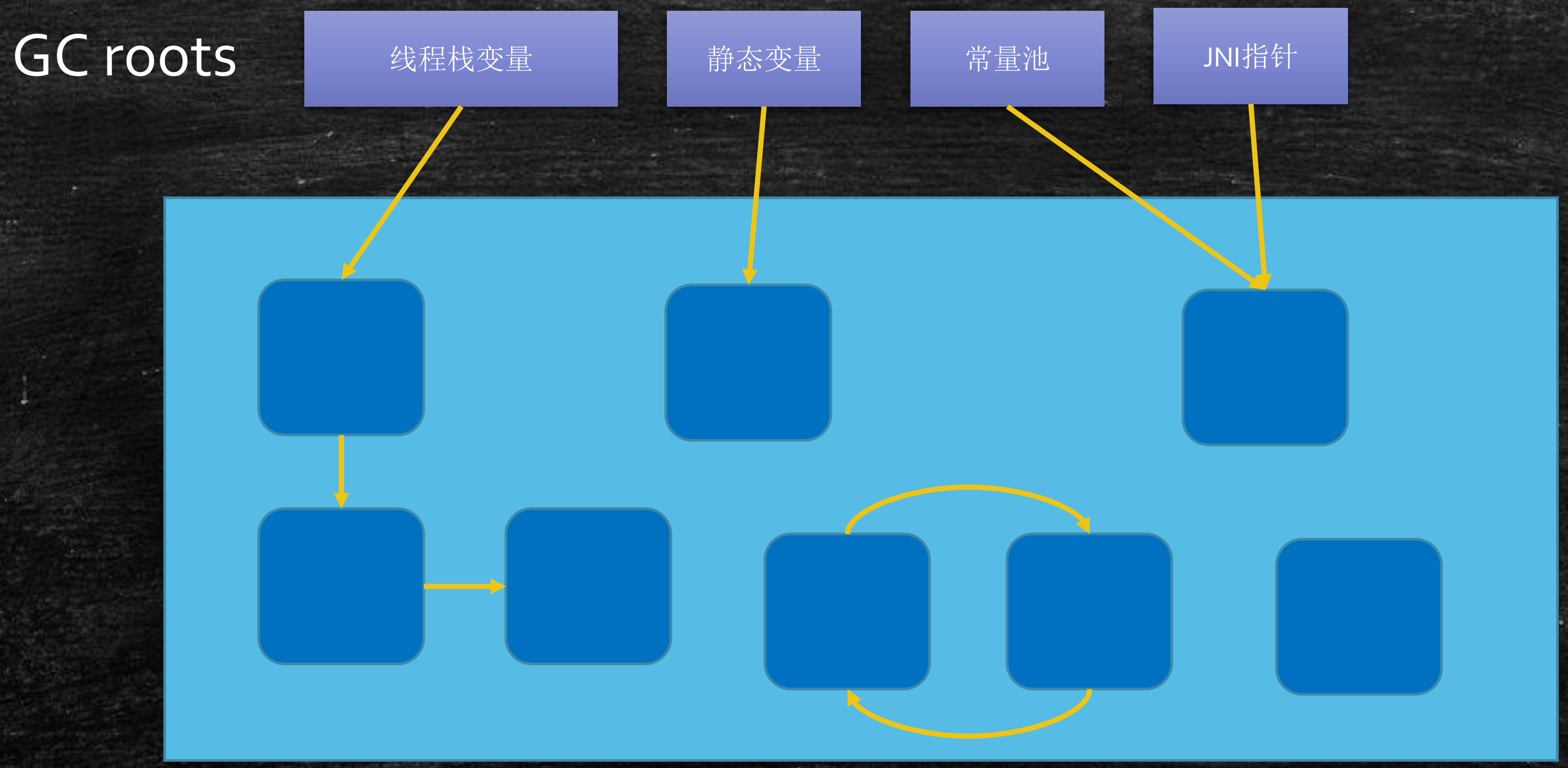
- a compacting collector that uses multiple GC threads.



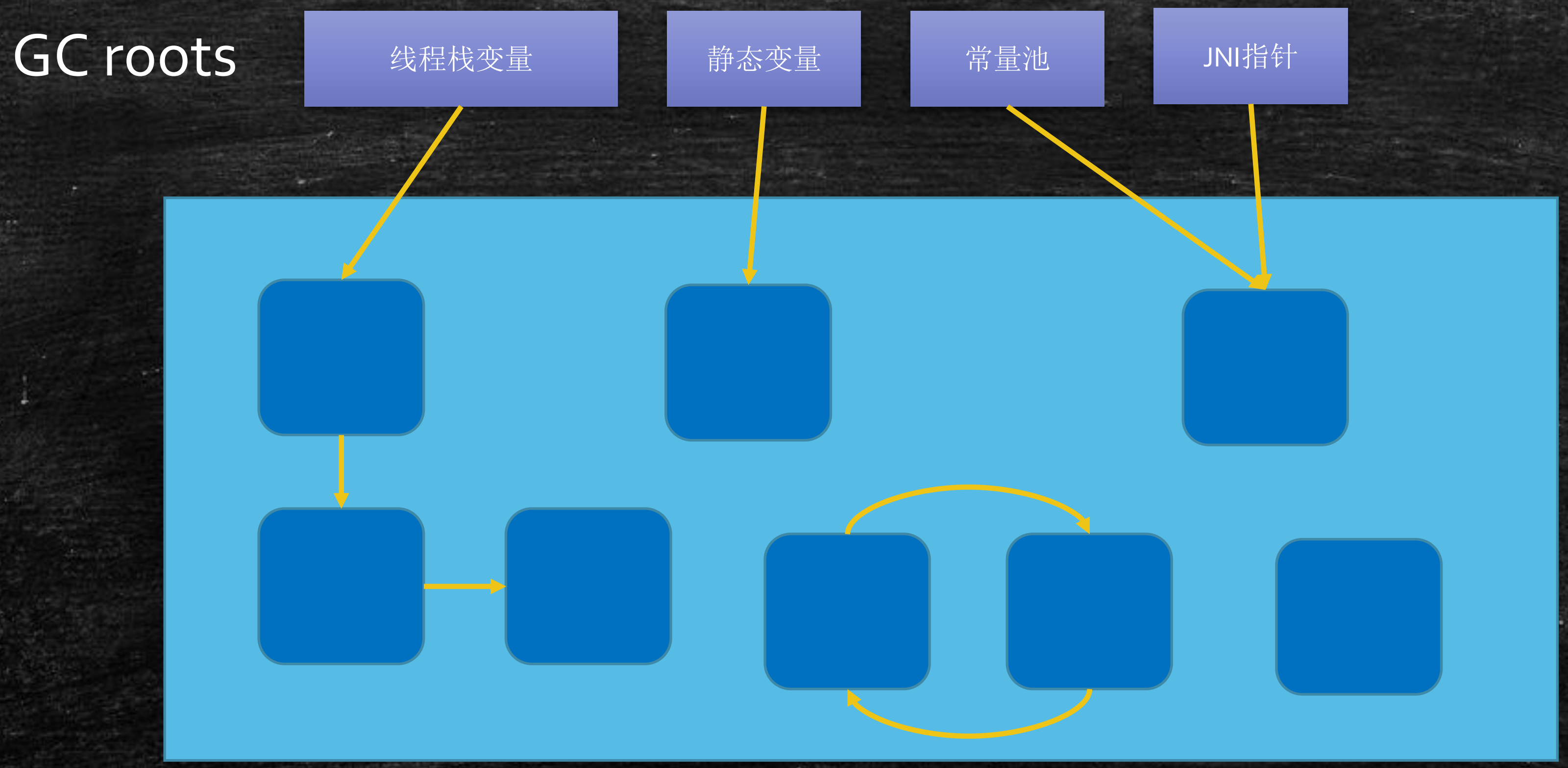
CMS

- concurrent mark sweep
- a mostly concurrent, low-pause collector.
- 4 phases
  1. initial mark
  2. concurrent mark
  3. remark
  4. concurrent sweep

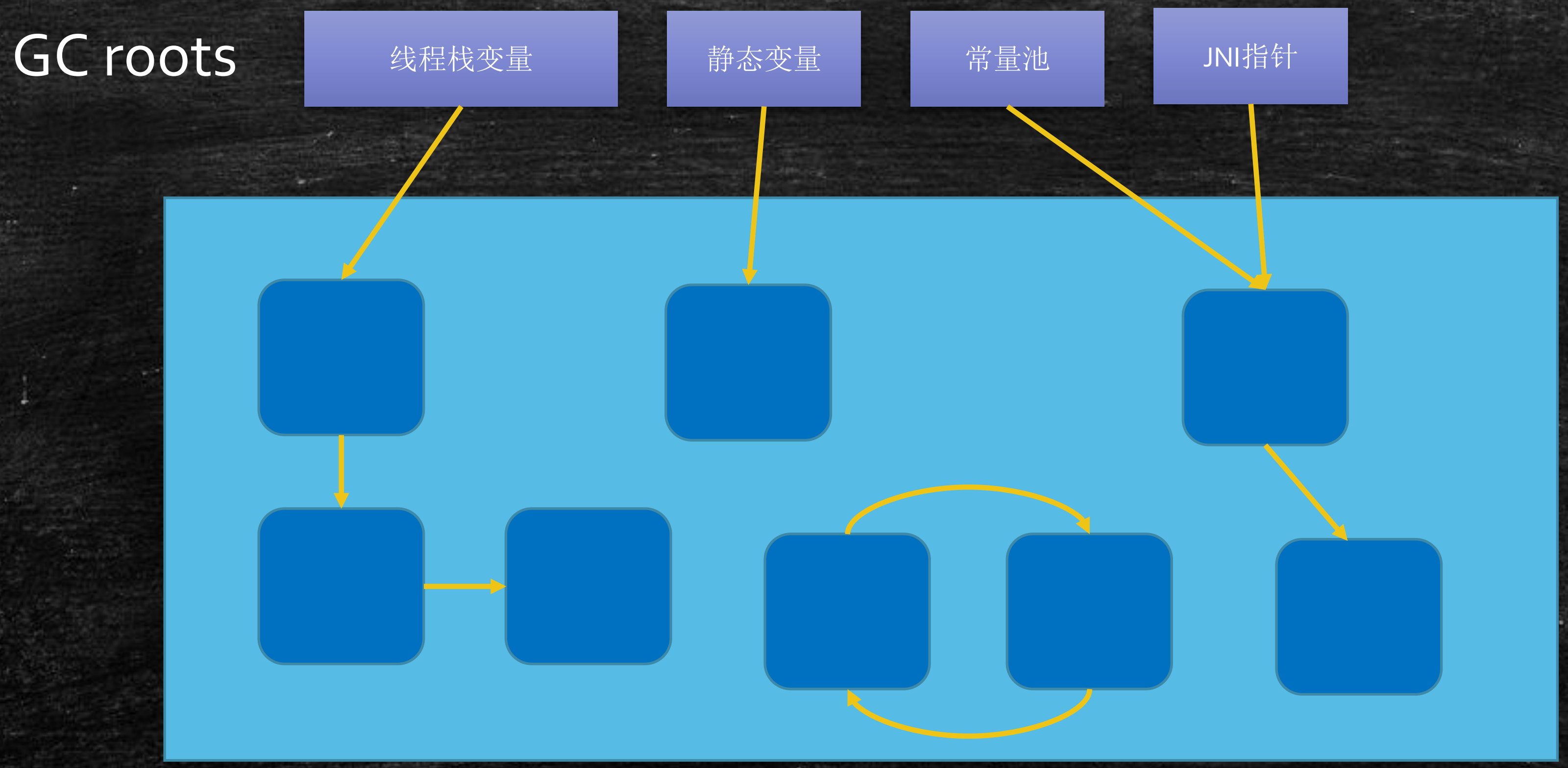
CMS initial mark



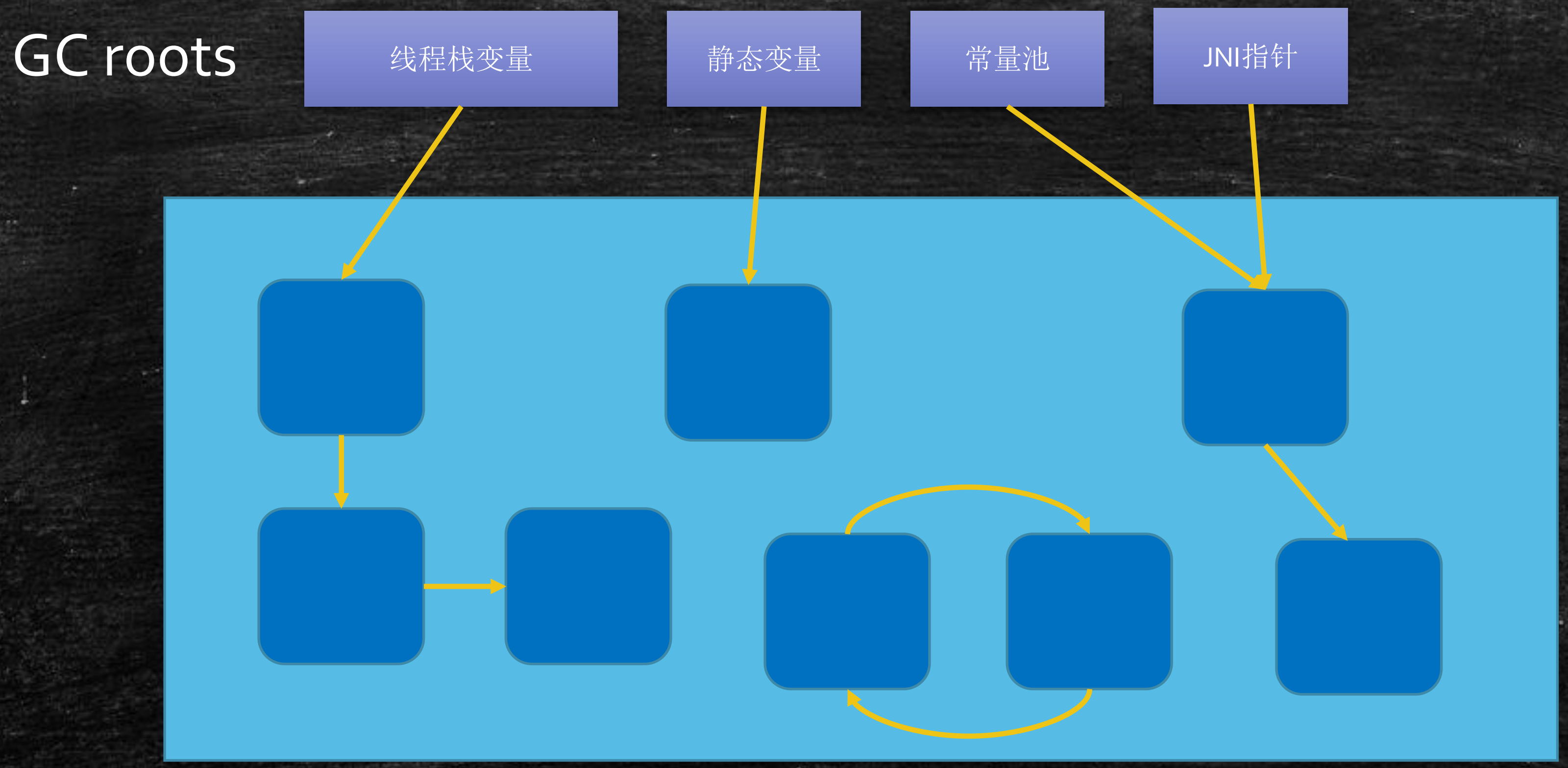
CMS concurrent mark



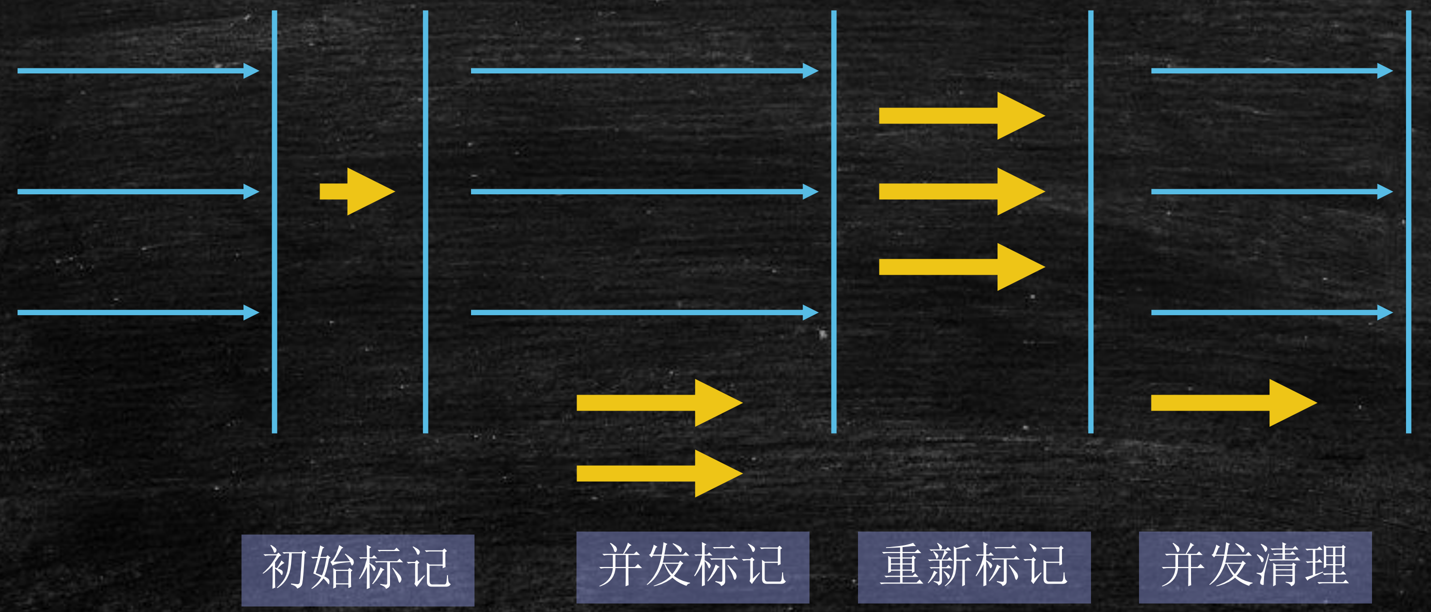
CMS remark



CMS remark

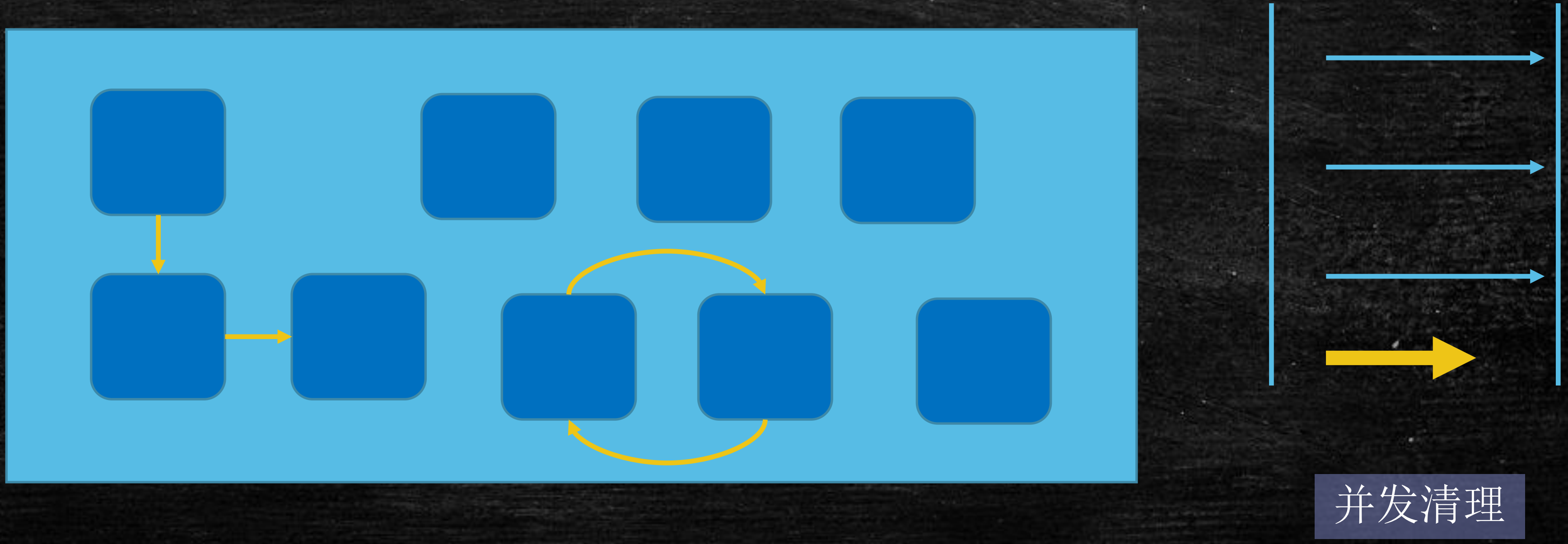


从线程角度



CMS缺点

- memory fragmentation
  - -XX:CMSFullGCsBeforeCompaction
- floating garbage
  - Concurrent Mode Failure –XX:CMSInitiatingOccupancyFraction 92%
  - SerialOld



组合参数

- UseSerialGC = Serial + Serial Old
- UseParNewGC = ParNew + Serial Old
- UseConcurrentMarkSweepGC = ParNew + CMS + Serial Old
  - "CMS" is used most of the time to collect the tenured generation. "Serial Old" is used when a concurrent mode failure occurs
- UseParallelGC = Parallel Scavenge + Serial Old
- UseParallelOldGC = Parallel Scavenge + Parallel Old

questions

- ParNew 和 PS哪一个更快?
- 采用何种类型的GC
  - 如何确定系统适用吞吐量优先的GC还是反应时间优先的GC
- 如果采用ParNew + CMS
  - 怎么做才能够让系统基本不产生FGC

Answers:

1. 频繁客户访问响应 VS 长时间计算
2. 答案：
   1. 加大JVM内存
   2. 加大Young的比例
   3. 提高Y-O的年龄
   4. 提高S区比例
   5. 避免代码内存泄漏

JVM参数类型

- -
  - 标准参数，所有JVM都应该支持
- -X
  - 非标，每个JVM实现不同
- -XX
  - 不稳定参数，下个版本可能取消

java hotspot vm options

- 参考
  - https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html
- 常用：
  - -XX:+PrintFlagsFinal
    - 设置值（最终生效值）
  - -XX:+PrintFlagsInitial
    - 默认值
  - -XX:+PrintCommandLineFlags
    - 命令行参数

总结

- 什么是垃圾
- 如何确定垃圾对象
- 常用垃圾回收算法
- 常用垃圾收集器

参考资料

https://blogs.oracle.com/jonthecollector/our-collectors

GC Tuning实战

GC Tuning实战包括哪些内容?

- 系统上线前，
  - 预估预优化JVM的各种垃圾回收选择
- 系统上线后，
  - 优化运行JVM的运行环境，解决JVM运行中出现的问题

JVM GC 第七次课程内容

- 复习基础知识
- 拾遗：MethodArea的演进
  - <1.8
  - >=1.8
- 实战理解GC 调优
  - PS + PO

面试：CPU突然飙高如何解决？

top –Hp 1122

```
Tasks:  61 total,   1 running,  60 sleeping,   0 stopped,   0 zombie
Cpu(s): 43.8%us,  1.8%sy,  0.0%ni, 54.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1004412k total,   396844k used,   607568k free,     8496k buffers
Swap:  2047992k total,        0k used,  2047992k free,    61408k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1124 root      20   0 2185m 233m  11m S 20.6 23.8   0:03.65 java
 1172 root      20   0 2185m 233m  11m S  1.0 23.8   0:03.67 java
 1133 root      20   0 2185m 233m  11m R  0.7 23.8   0:03.52 java
 1135 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.65 java
 1136 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.63 java
 1137 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.59 java
 1139 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.65 java
 1142 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.58 java
 1143 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.57 java
 1144 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.57 java
 1147 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.62 java
 1148 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.61 java
 1150 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.56 java
 1153 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.57 java
 1156 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.67 java
 1157 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.68 java
 1158 root      20   0 2185m 233m  11m S  0.7 23.8   0:03.63 java
```

```
top - 15:46:53 up 19 min,  2 users,  load average: 0.82, 0.40, 0.16
Tasks:  61 total,   1 running,  60 sleeping,   0 stopped,   0 zombie
Cpu(s):100.0%us,  0.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   1004412k total,   397340k used,   607072k free,     8496k buffers
Swap:  2047992k total,        0k used,  2047992k free,    61468k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1124 root      20   0 2185m 234m  11m R 95.2 23.9   1:01.69 java
 1138 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.07 java
 1151 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.14 java
 1155 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.11 java
 1156 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.25 java
 1158 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.20 java
 1162 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.23 java
 1164 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.16 java
 1172 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.23 java
 1173 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.27 java
 1178 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.16 java
 1181 root      20   0 2185m 234m  11m S  0.3 23.9   0:04.22 java
 1122 root      20   0 2185m 234m  11m S  0.0 23.9   0:00.04 java
 1123 root      20   0 2185m 234m  11m S  0.0 23.9   0:01.56 java
 1125 root      20   0 2185m 234m  11m S  0.0 23.9   0:00.00 java
 1126 root      20   0 2185m 234m  11m S  0.0 23.9   0:00.00 java
 1127 root      20   0 2185m 234m  11m S  0.0 23.9   0:00.00 java
```

jstack 1122

```
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000f8ad4378> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:144)
        - locked <0x00000000f8ad4378> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:165)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:216)

"Reference Handler" #2 daemon prio=10 os_prio=0 tid=0x00007faae4075800 nid=0x465 in Object.wait() [0x00007faae9284000]
   java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000f8ad4530> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
        - locked <0x00000000f8ad4530> (a java.lang.ref.Reference$Lock)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"VM Thread" os_prio=0 tid=0x00007faae406d800 nid=0x464 runnable

"VM Periodic Task Thread" os_prio=0 tid=0x00007faae40cb000 nid=0x46b waiting on condition

JNI global references: 183
```

十进制1122

吞吐量 = 用户代码执行时间 / (用户代码执行时间 + 垃圾收集执行时间)
响应时间快 = 用户线程停顿的时间短

确定调优之前，应该确定到底是哪个优先，是计算型任务还是响应型任务

GC Tuning实战包括哪些内容?

- 系统上线前，
  - 预估预优化JVM的各种垃圾回收选择
- 系统上线后，
  - 优化运行JVM的运行环境，解决JVM运行中出现的问题（OOM）

服务器配置预估

- 场景：垂直电商，最高每日百万订单，处理订单系统需要什么样的服务器配置？
- 伪命题：从小-大
- 峰值：一个小时36万订单，100单/秒，
- 处理一张订单多长时间，0.05秒，1秒钟20单，
- 5台机器，
- 内存设置：20个订单多少内存 （取决于能容忍的YGC FGC的频率）

GC日志详解

YGC

GC
原因

回收前年
轻代空间

回收后年
轻代空间

年代

[GC
(Allocation Failure)
[DefNew: 4544K->259K(6144K), 0.0873295 secs]
4544K->4356K(19840K), 0.0873710 secs]
[Times: user=0.00 sys=0.09, real=0.09 secs]

年轻代总
大小

linux
time ls

回收前堆
占用空间

回收后堆
占用空间

总堆空间

```
Heap
 def new generation   total 6144K, used 5504K [0x00000000fec00000, 0x00000000ff2a0000,
0x00000000ff2a0000)
  eden space 5504K, 100% used [0x00000000fec00000, 0x00000000ff160000, 0x00000000ff160000)
  from space 640K,   0% used [0x00000000ff160000, 0x00000000ff160000, 0x00000000ff200000)
  to   space 640K,   0% used [0x00000000ff200000, 0x00000000ff200000, 0x00000000ff2a0000)
 tenured generation   total 13696K, used 13312K [0x00000000ff2a0000, 0x0000000100000000,
0x0000000100000000)
   the space 13696K,  97% used [0x00000000ff2a0000, 0x00000000fffa0148, 0x00000000fffa0200,
0x0000000100000000
) Metaspace       used 2538K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 275K, capacity 386K, committed 512K, reserved 1048576K
```

已经使用

总容量

虚拟内存
占用

虚拟内存
保留

案例
有一个50万PV的资料类网站（从磁盘提取文档到内存）原服务器32位，1.5G的堆，用户反馈网站比较缓慢，因此公司决定升级，新的服务器为64位，16G的堆内存，结果用户反馈卡顿十分严重，反而比以前效率更低了

为什么？
如何优化？

案例：
开源软件Xfire缺陷
https://blog.csdn.net/qq_15037231/article/details/80689905