JVM GC 介绍

BOSS直聘 研发五组 陈良柱

/01 简介

- Overview of famous JVMs and Garbage Collection;
- Summary for Garbage Collectors in Hotspot JVM.

GC – Garbage Collection

A form of Automatic Memory Management

The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to simplify manual memory management in Lisp.

01

The First GC

- 1959 年，MIT 的 Lisp 语言
- John McCarthy（Lisp之父 & 人工智能之父）
- 标记-清除 (Mark-Sweep) 算法。

02

Advantages

- 不用重复造轮子
- 不容易出现内存泄漏Bug
- 出了Bug可以甩锅

03

Why We Learn?

- 工作需要：解释现象，解决问题，避免掉坑
- 职业发展：一门不会过时的技术，一部技术演进和系统设计优化的发展史
- 人生追求： # // /* £ € ¥ ฿ ₩ $ ₤ ৺ ……

那些自带GC的 LANG/VM

1 2 3 4 5

Lisp

- Lisp1 (Mark-Sweep)
- Lisp2 (Mark-Compact)

JavaScript V8

- Cheney (Copying), Mark-Sweep, Mark-Compact,
- Generational, Incremental Marking, Lazy Sweeping

Sun Java VM

- Copying, Mark-Sweep, Mark-Compact
- Generational, Incremental, Parallel, Concurrent

Golang

- Mark-Sweep (<1.3)
- Mark & Concurrent Sweep (1.3~1.4)
- Concurrent Mark-Sweep (≥1.5)

Python

- Reference Counting, Mark-Sweep
- Generational

Sun JVM 发展史

Sun Classic VM

- 1996年，JDK 1.0发布，提供了纯解释执行的 Java 虚拟机实现：Sun Classic VM
- 1997年，JDK 1.1发布，虚拟机没有做变更，依然使用 Sun Classic VM 作为默认的虚拟机

JDK 1.0 ~ 1.2

Sun Exact VM

1998年，JDK 1.2发布，提供了运行在Solaris平台的Exact VM，但默认还是Sun Classic VM

JDK 1.2 Solaris

Sun HotSpot VM

- 2000年，JDK 1.3发布，默认虚拟机改为Sun HotSopt VM，而Sun Classic VM则作为备用虚拟机
- 2002年，JDK 1.4发布，Classic VM退出商用舞台，Sun HotSpot VM作为默认虚拟机，直到现在

JDK 1.3 ~ now

那些传说中的 JVM

01

通用平台 - 大型商用JVM

- BEA JRockit VM (Mission Control)
- IBM J9 VM

02

特定硬件及平台 - 高性能JVM

- Azul Zing VM (Vega系统)
- BEA Liquid VM (Hypervisor系统)

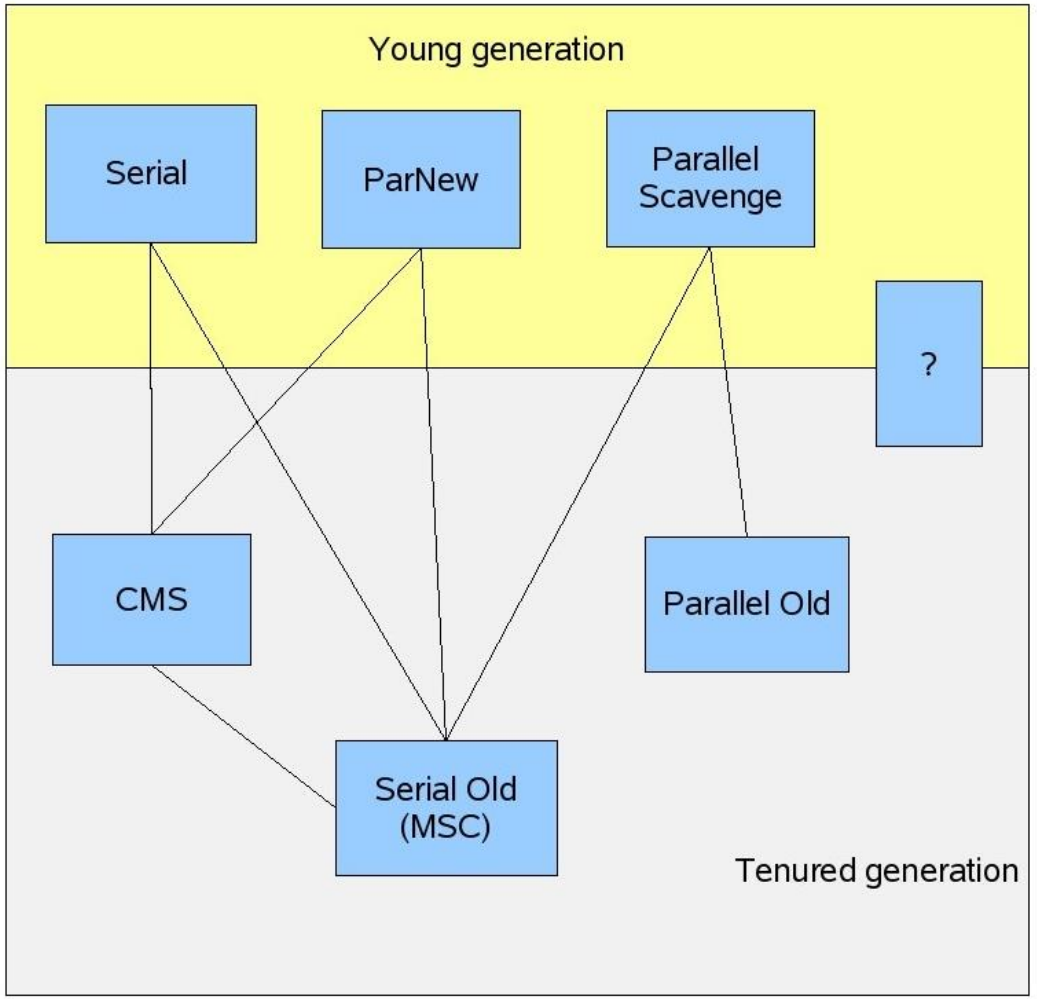
03

其他“准”JVM

- Apache Harmony (开源项目)
- Google Dalvik VM
- Microsoft JVM (J++)

Sun JVM 垃圾收集器 (Garbage Collector)

	Young	Tenured
Serial	✓ Serial	✓ Serial Old
Parallel	✓ Parallel Scavenge ✓ ParNew	✓ Parallel Old
Concurrent		✓ CMS
Partial / Incremental	✓ Generational GC ✓ Train GC	
Concurrent & Partial	✓ CMS Incremental Mode ✓ G1（初期无分代） ✓ Balanced GC ✓ Shenandoah（暂无分代） ✓ ZGC（暂无分代）	



GC 命名的艺术

- [GC[**DefNew**: 3324K- > 152K(3712K), 0.0025925secs]3324K->152K(11904K), 0.0031680 secs]
- [FullGC[**Tenured**: 0K- > 210K(10240K), 0.0149142secs]4603K->210K(19456K), [**Perm** : 2999K- > 2999K(21248K)], 0.0150007 secs][Times: user=0.01 sys=0.00, real=0.02 secs]

- [GC [**PSYoungGen**: 4423K->320K(9216K)] 4423K->320K(58880K), 0.0011900 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
- [Full GC (System) [PSYoungGen: 320K->0K(9216K)] [**ParOldGen**: 0K->222K(49664K)] 320K->222K(58880K) [**PSPermGen**: 2458K->2456K(21248K)], 0.0073610 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

- [GC 38887.270: [**ParNew**: 1791076K->170624K(1877376K), 0.2324440 secs] 2988366K->1413629K(4023680K), 0.2326470 secs] [Times: user=0.80 sys=0.00, real=0.23 secs]
- [GC 7.092 : [ParNew: 471872K->471872K(471872K), 0.0000420 secs] 7.092 : [**CMS**: 366666K->524287K(524288K), 27.0023450 secs] 838538K->829914K(996160K), [**CMS Perm** : 3196K->3195K(131072K)], 27.0025170 secs]
- [Full GC (System) 50.568: [CMS: 943772K->220K(2596864K), 2.3424070 secs] 1477000K->220K(4061184K), [CMS Perm : 3361K->3361K(98304K)], 2.3425410 secs] [Times: user=2.33 sys=0.01, real=2.34 secs]

垃圾收集的性能指标

《JDK官方文档》

- **Throughput**
  - the percentage of total time not spent in garbage collection considered over long periods of time.
- **Pauses**
  - the times when an application appears unresponsive because garbage collection is occurring.
- Footprint
  - the working set of a process, measured in pages and cache lines.
- Promptness
  - the time between when an object becomes dead and when the memory becomes available.

《垃圾回收的算法与实现》

I. **吞吐量**

II. **最大暂停时间**

III. 堆使用效率

IV. 访问的局部性

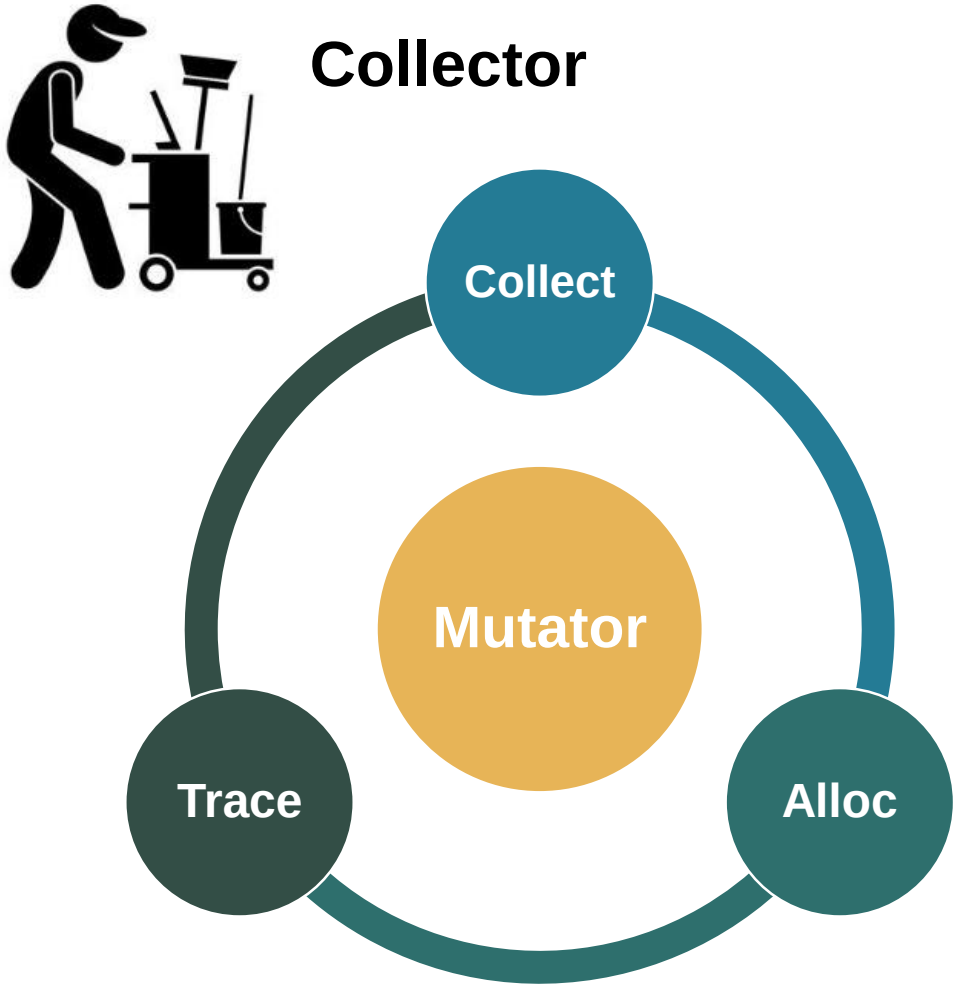
主要垃圾收集器对比

	Multiple GC Threads	STW (Young)	STW (Tenured)	Heap Compaction	Primary Goal
Serial	No	Yes	Yes	Yes	-
Parallel	Yes	Yes	Yes	Yes	Throughput
CMS	Yes	Yes	Short	No	Latency
G1	Yes	Yes	Very short	Partially	Latency
ZGC	Yes	-	No	Yes	Latency

/02 原理

- Understand how GC works;
- Dive deep into the classic algorithms.

GC 基本工作原理



- 对象标记算法
  - 引用计数法 Reference Counting
  - 根搜索算法 Roots Tracing（可达性分析、引用链）
- 垃圾对象回收算法
  - 标记 - 清除 Mark-Sweep
  - 标记 - 压缩 Mark-Compact
  - 复制 Copying
- 优化/改进策略
  - 增量/部分/分代收集 Incremental/Partial/Generational
  - 多线程并行收集 Parallel
  - 与应用并发执行 Concurrent

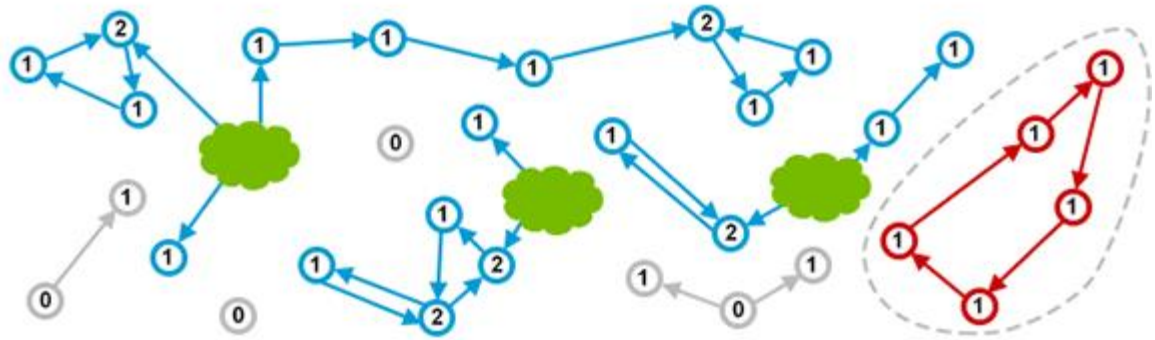
GC 对象标记算法比较

- **引用计数法 Reference Counting**

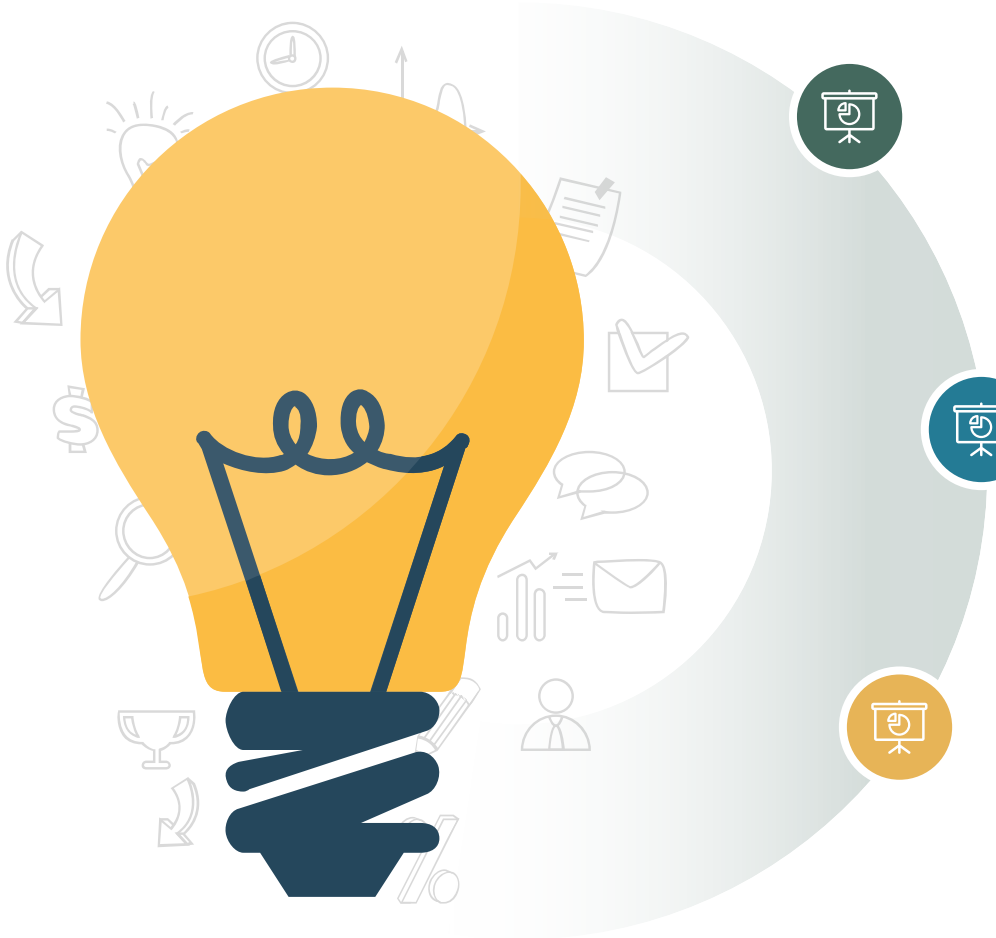
  + 实时回收
  + 暂停时间短
  + 算法简单，实现简单（可能更加复杂）
  - 循环引用问题 Cycles
  - 额外的空间 Space overhead
  - 计数器更新频繁 Speed overhead, Atomicity

- **根搜索算法 Roots Tracing**

  - 对象延迟释放
  - 暂停时间难以控制



GC 对象回收算法



标记 - 清除 Mark-Sweep

类似于操作系统内存分配与回收管理的处置方式，存在内存碎片问题

著名的内存分配器：dlmalloc, jemalloc, tcmalloc

标记 - 压缩 Mark-Compact

也叫“标记 - 整理”算法，需要移动内存数据，更新相关引用

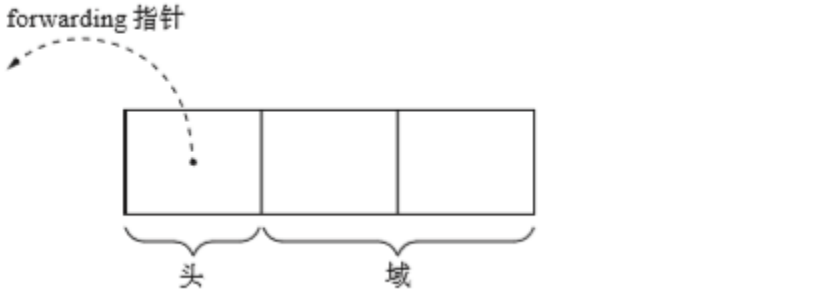
内存利用率高，需要多次遍历数据，时间换空间

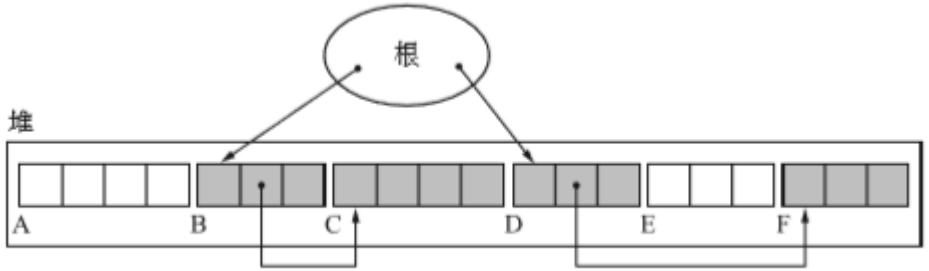
复制 Copying

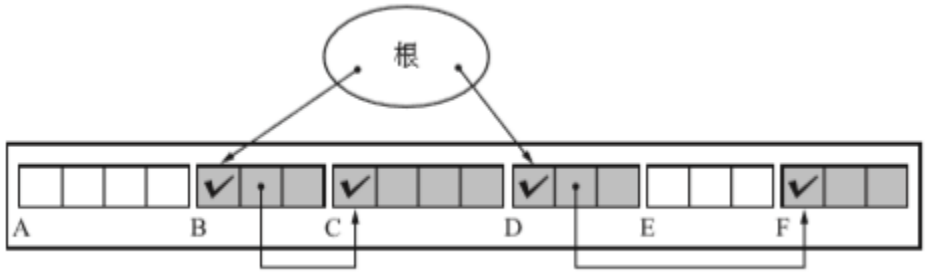
Semi-Space，内存一分为二，交替使用，同一时刻只用其一

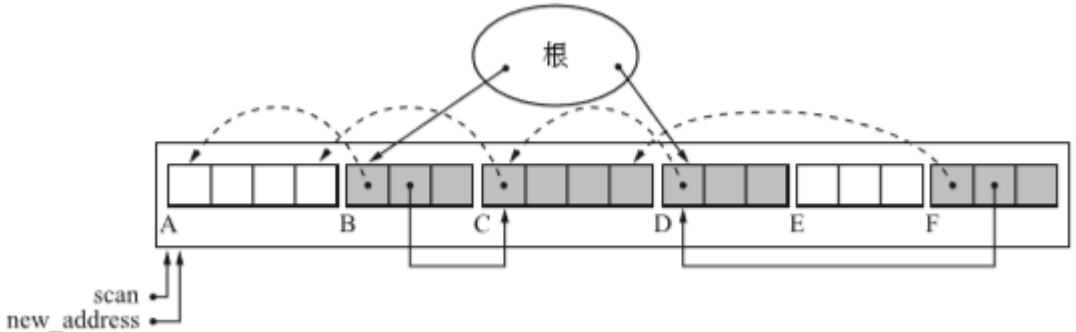
一次遍历完成内存回收，空间换时间

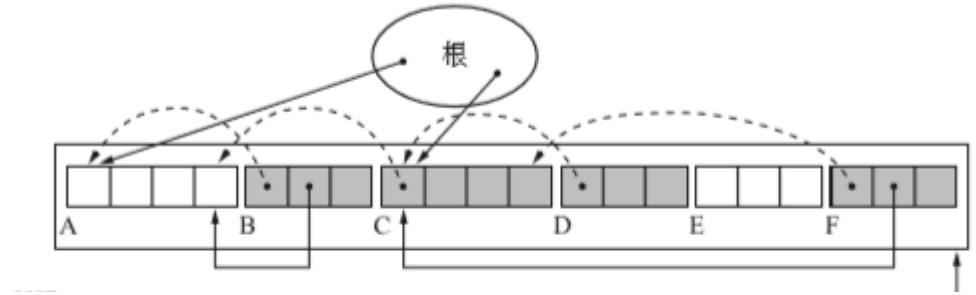
GC 算法选讲 —— Lisp2（Mark-Compact）

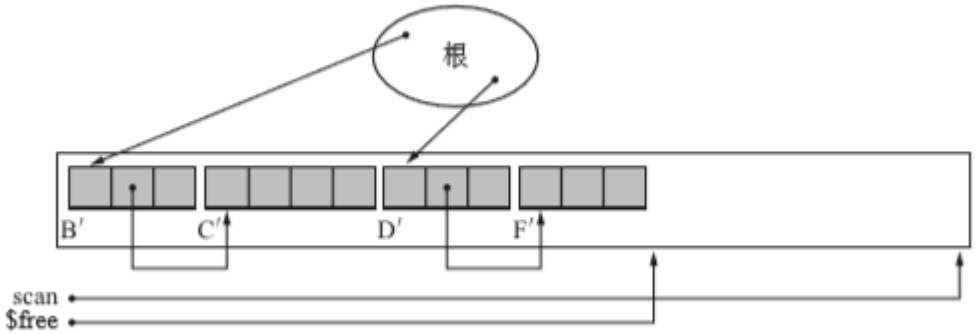












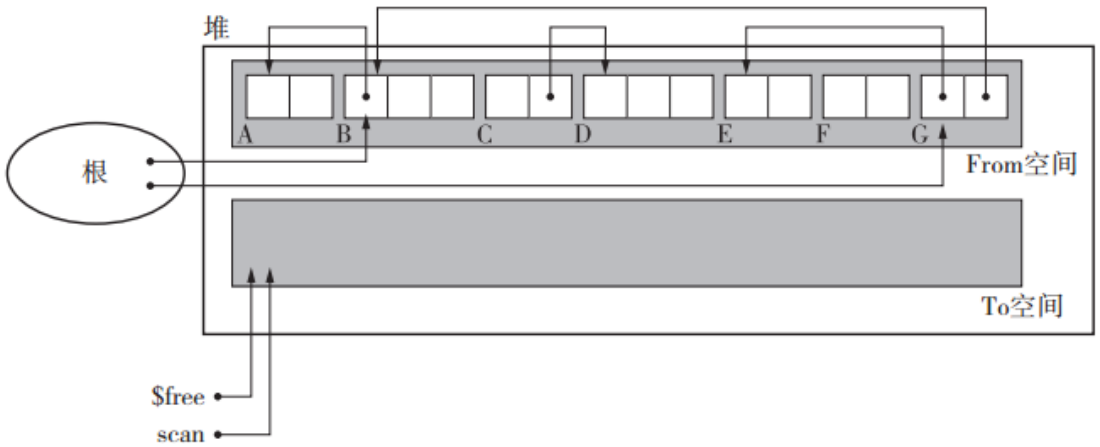
GC 算法选讲 —— Lisp2（Mark-Compact）

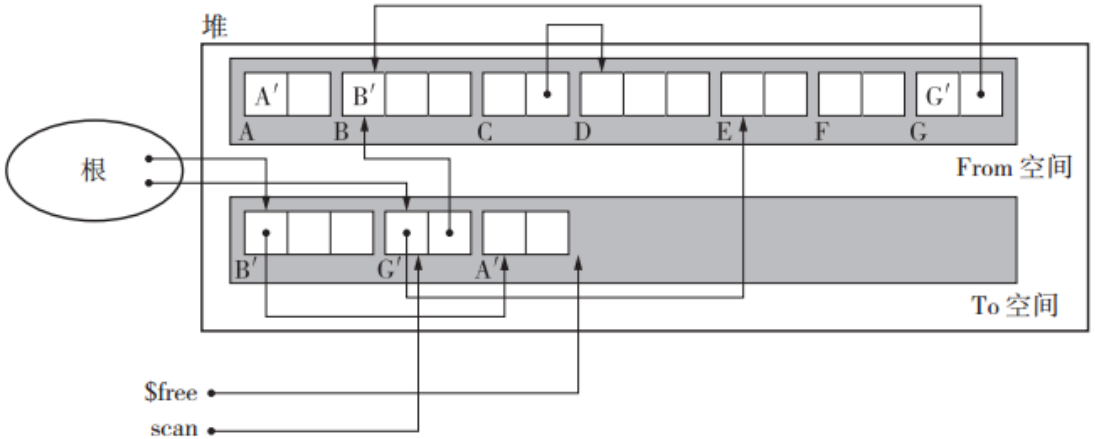
```
compaction_phase() {
    set_forwarding_ptr()   // 设置forwarding指针
    adjust_ptr()           // 更新指针
    move_obj()             // 移动对象
}
```

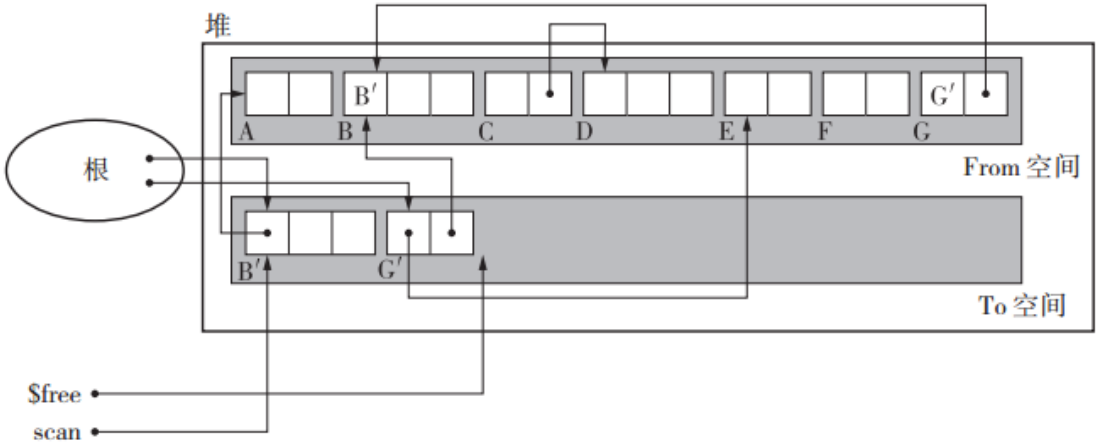
- 优点：空间利用率高
- 缺点：时间成本高
  - 必须对整个堆进行3次搜索

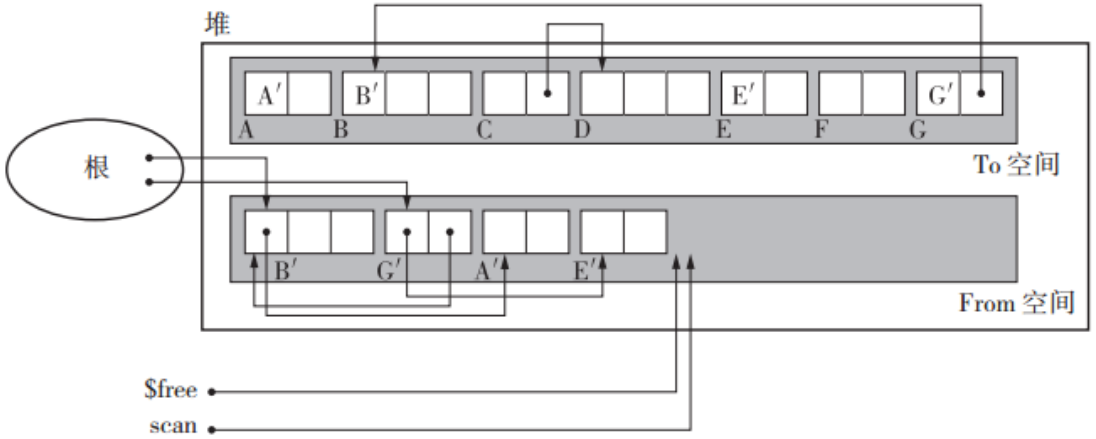
```
set_fowarding_ptr() {
    scan = new_address = $heap_start
    while (scan < $heap_end)
        if (scan.mark = TRUE)
            scan.forwarding = new_address
            new_address += scan.size
        scan += scan.size
}
```

GC 算法选讲 —— Cheney（Copying）









GC 算法选讲 —— Cheney（Copying）

```
copying() {
    scan = $free = $to_start
    for (r : $roots)
        *r = copy(*r)
    while (scan != $free) {
        for (child : children(scan))
            *child = copy(*child)
        scan += scan.size
    }
}
```

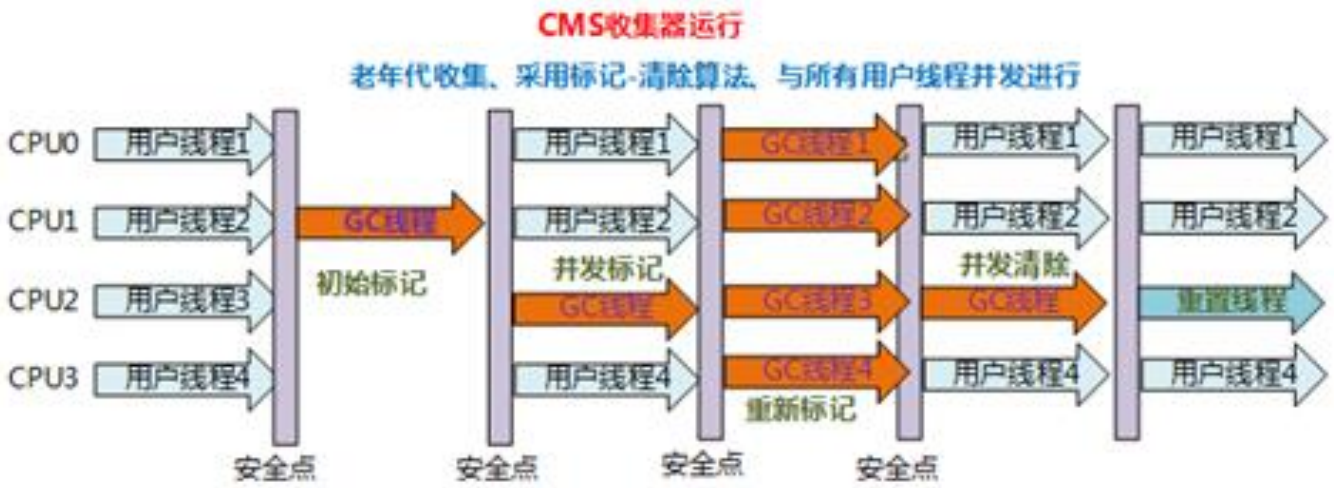
```
copy(obj) {
    if (is_not_pointer_to_heap(obj.forwarding, $to_start))
        copy_date($free, obj, obj.size)
        obj.forwarding = $free
        $free += obj.size
    return obj.forwarding
}
```

- 优点：非递归实现，堆直接作为队列使用
- 缺点：缓存利用率差

GC 算法选讲 —— CMS并发GC原理

- Tri-Color Marking
- Remembered Set & Write Barrier
- 主要分为四个阶段：
  - ① Initial Mark（初始标记） - STW
  - ② Concurrent Mark（并发标记）
  - ③ Remark （重新标记） - STW
  - ④ Concurrent Sweep（并发清除）
- 主要特点：
  - ① 停顿时间短
  - ② 并发执行阶段占用CPU资源
  - ③ 会产生浮动垃圾 Floating Garbage

Mostly-concurrent Garbage Collector



CMS发并GC线程数默认值： (CPU个数 + 3) / 4

公式的含义：保证GC线程占用至少 **25%** 的CPU资源

GC 算法选讲 —— CMS并发GC原理

- **Tri-Color Marking**
- **Remembered Set** & **Write Barrier**
- 主要分为四个阶段：
  - ① Initial Mark（初始标记） - STW
  - ② Concurrent Mark（并发标记）
  - ③ Remark （重新标记） - STW
  - ④ Concurrent Sweep（并发清除）
- 主要特点：
  - ① 停顿时间短
  - ② 并发执行阶段占用CPU资源
  - ③ 会产生浮动垃圾 Floating Garbage

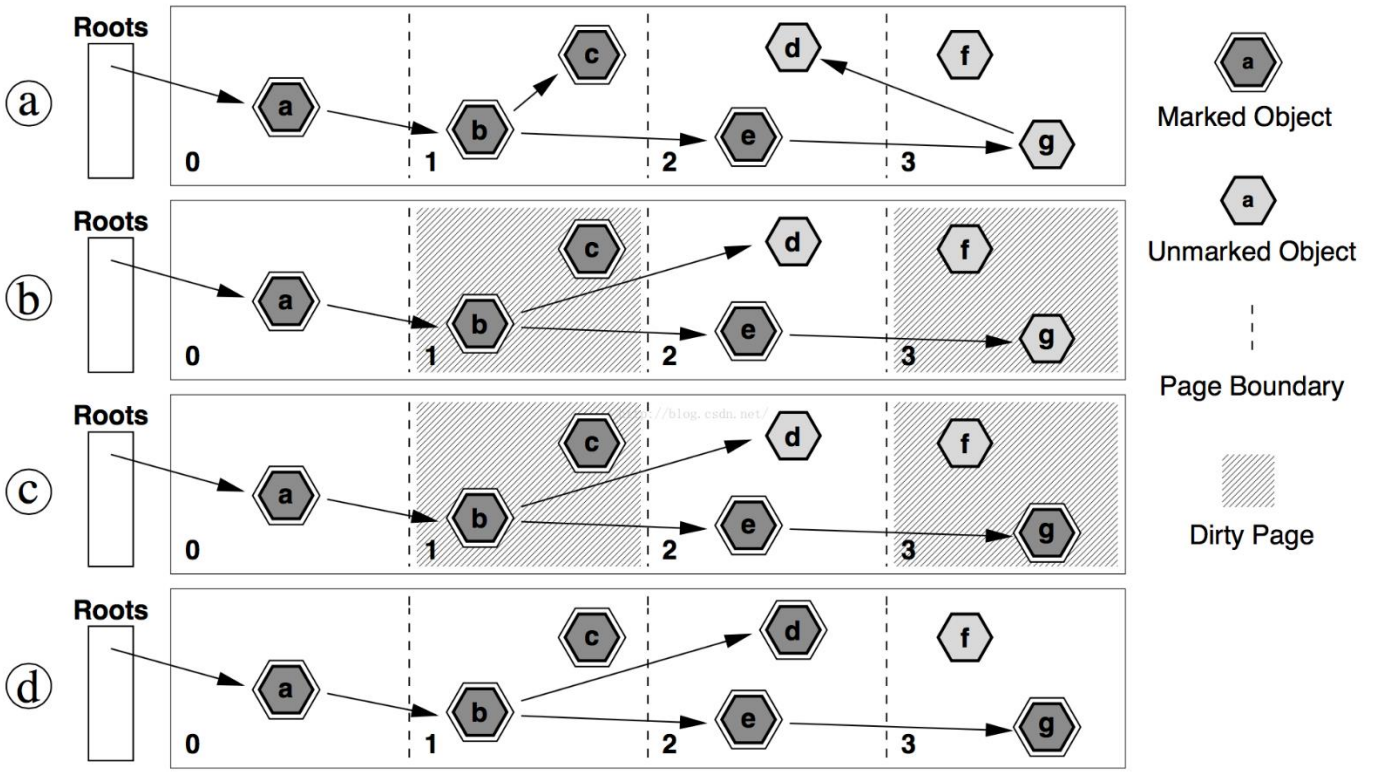
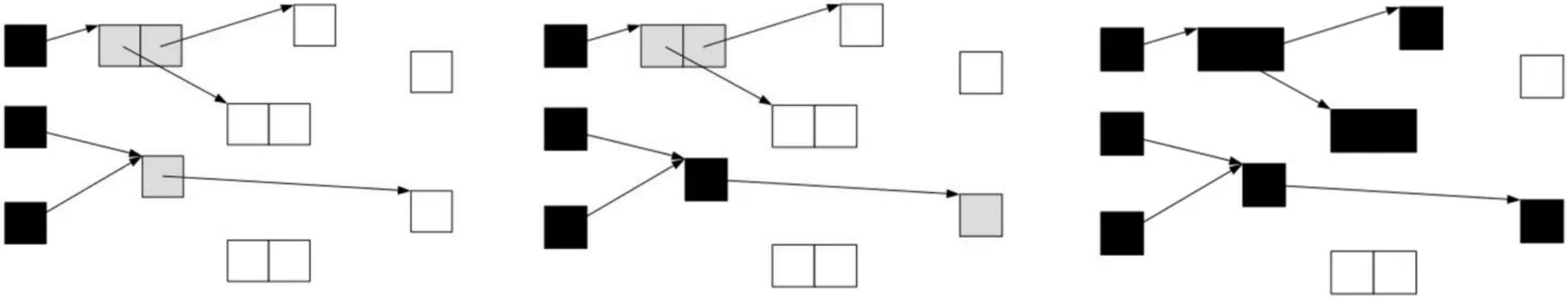


Figure 1: Concrete example of the operation of the original mostly-concurrent garbage collector.

GC 算法选讲 —— Tri-Color Marking（三色标记法）

- **黑色**：根对象，或者该对象与它的子对象都被扫描过；
- **灰色**：对象本身被扫描，但还没扫描完该对象中的子对象；
- 白色：未被扫描对象，扫描完成所有对象之后，最终为白色的为不可达对象，既垃圾对象。



GC 算法选讲 —— 并发问题

1. 错标

   - 把垃圾对象标记为可达对象，不影响程序的正确性，但会产生浮动垃圾；

2. 漏标：

   - 可达对象没有做标记，会被当作垃圾回收掉，影响程序的正确性。

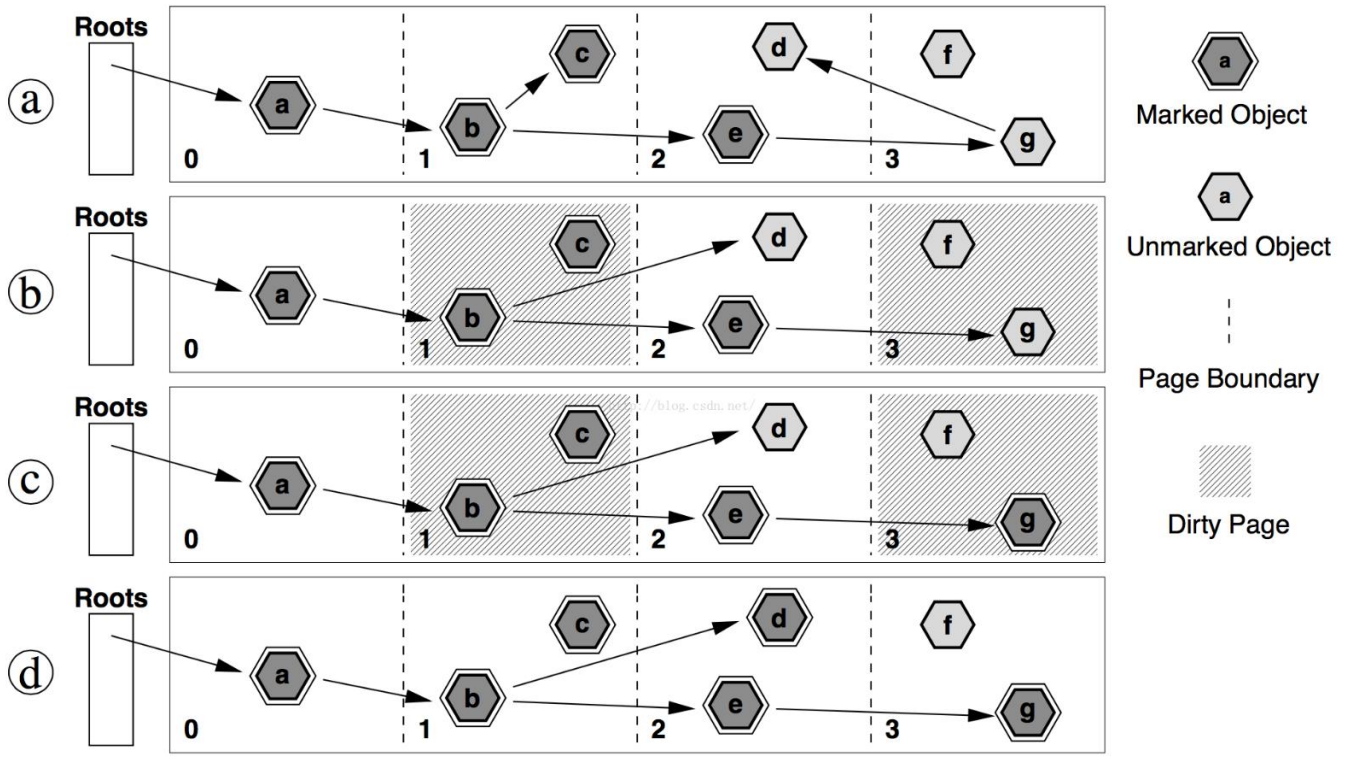


Figure 1: Concrete example of the operation of the original mostly-concurrent garbage collector.

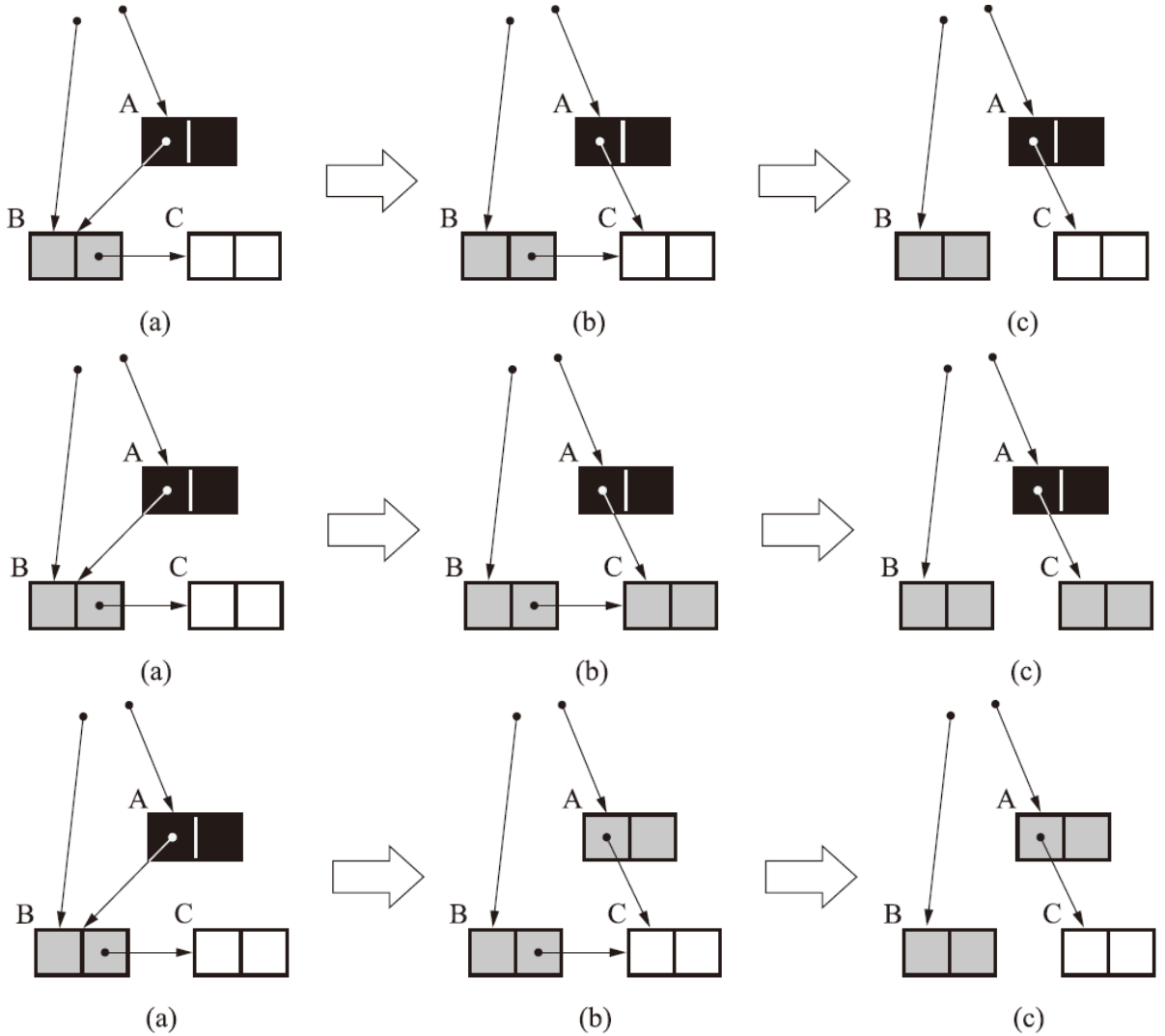
GC 算法选讲 —— Write Barrier（写屏障）

并发问题

- 错标：把垃圾对象标记为可达对象，可忍；
- 漏标：可达对象没有做标记，不可忍。

解决方法

- Write Barrier（写屏障）
  - 对象引用变更时插入检测和标记
- Remembered Set（标记结果集）
  - Card Table, Mod Union Table
- SATB
  - Snapshot At The Beginning



/03 实践

- Glance at the key options in common usage of GC;
- Discuss the details of CMS GC and how to adjust it.

GC 收集器使用偏好

	Collector	Memory	Focus
Client	Serial	-	High Throughput & Low CPU Usage
Server	Parallel	-	High Throughput
Server	ParNew + CMS	≤ 4G	Low Latency
Server	ParNew + CMS	> 4G	Low Latency
Server	G1	Large	Low Latency & No Fragmentation
Server	ZGC	ALAP	Very Low Latency (≤10ms)

低延迟方案应用趋势：CMS -> G1 -> ZGC

理解 GC 并发收集的思路

- 并发GC根本上要跟应用玩追赶游戏：应用一边在分配，GC一边在收集，如果GC收集的速度能跟得上应用分配的速度，那就一切都很完美；一旦GC开始跟不上了，垃圾就会渐渐堆积起来，最终到可用空间彻底耗尽的时候，应用的分配请求就只能暂时等一等了，等GC追赶上来。

—— *@RednaxelaFX*

- 并发GC在并发收集过程中，应用不可避免地会产生新的对象，要避免错标或者漏标这些对象以及与其他对象之间的关系，必须在对象引用的改变过程中插入特殊的处理；这种特殊代码如果插入在引用赋值，则称为写屏障（write barrier），如 CMS 和 G1 的实现方式；如果插入在引用读取过程，则称为读屏障（read barrier / load barrier），如 ZGC 的实现方式。

CMS 日志体验

- 64.425: [GC (CMS Initial Mark) [1 **CMS-initial-mark**: 10812086K(11901376K)] 10887844K(12514816K), 0.0001997 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
- 64.425: [CMS-concurrent-mark-start]
- 64.460: [**CMS-concurrent-mark**: 0.035/0.035 secs] [Times: user=0.07 sys=0.00, real=0.03 secs]
- 64.460: [CMS-concurrent-preclean-start]
- 64.476: [**CMS-concurrent-preclean**: 0.016/0.016 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
- 64.476: [CMS-concurrent-abortable-preclean-start]
- 65.550: [**CMS-concurrent-abortable-preclean**: 0.167/1.074 secs] [Times: user=0.20 sys=0.00, real=1.07 secs]
- 65.550: [GC (CMS Final Remark) [YG occupancy: 387920 K (613440 K)]65.550: [Rescan (parallel) , 0.0085125 secs]65.559: [weak refs processing, 0.0000243 secs]65.559: [class unloading, 0.0013120 secs]65.560: [scrub symbol table, 0.0008345 secs]65.561: [scrub string table, 0.0001759 secs][1 **CMS-remark**: 10812086K(11901376K)] 11200006K(12514816K), 0.0110730 secs] [Times: user=0.06 sys=0.00, real=0.01 secs]
- 65.561: [CMS-concurrent-sweep-start]
- 65.588: [**CMS-concurrent-sweep**: 0.027/0.027 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
- 65.589: [CMS-concurrent-reset-start]
- 65.601: [**CMS-concurrent-reset**: 0.012/0.012 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

GC 配置实践

要点	参数
获取参数初始值：	*$JAVA_CMD -XX:+PrintFlagsInitial*
确认参数最终值：	*$JAVA_CMD -XX:+PrintFlagsFinal*
输出GC原因（JDK7）：	*-XX:+PrintGCCause*
控制堆空间分配：	*-Xms -Xmx (避免堆大小动态调整)* *-Xmn -XX:NewSize -XX:MaxNewSize -XX:NewRatio=N*
控制新生代空间比例：	*-XX:SurvivorRatio=N*
控制大对象直接进入老年代：	*-XX:PretenureSizeThreshold=?*
慎用 FullGC 屏蔽和替换：	*-XX:+DisableExplicitGC* *-XX:+ExplicitGCInvokesConcurrent*

GC 配置实践 —— CMS重要参数

要点	参数
控制GC并发线程数：	*-XX:ParallelGCThreads=N (Young)* *-XX:ConcGCThreads=N OR -XX:ParallelCMSThreads=N (CMS)*
干预CMS启动时间：	*-XX:+UseCMSInitiatingOccupancyOnly* *-XX:CMSInitiatingOccupancyFraction=N*
协调YGC和CMS：	*-XX:+CMSScavengeBeforeRemark* *-XX:-CMSPrecleaningEnabled* *-XX:CMSMaxAbortablePrecleanTime=N (ms)*
启用并行缩短STW时间：	*-XX:+CMSParallelInitialMarkEnabled* *-XX:+CMSParallelRemarkEnabled*
压缩处理的普遍误解：	*-XX:+UseCMSCompactAtFullCollection* *-XX:CMSFullGCsBeforeCompaction=N* *Background CMS vs Foreground CMS vs Full GC*

CMS ≠ Full GC

- Minor GC, Major GC, Full GC
- Background CMS, Foreground CMS
- 什么时候会发生 Full GC？
  - Tenured 空间不足：java.lang.OutOfMemoryError：Java heap space
  - Perm 空间不足：java.lang.OutOfMemoryError：PermGen space
  - CMS GC时出现 promotion failed 和 concurrent mode failure
  - Tenured 剩余空间小于 Minor GC 晋升的平均大小（悲观策略）
  - 主动触发Full GC：System.gc(), jmap -histo:live [pid]

参考资料推荐

- *中村成洋, 相川光. 垃圾回收的算法与实现. 人民邮电出版社, 2016*
- *Tony Printezis, David Detlefs. A Generational Mostly-concurrent Garbage Collector. In Proceedings of the 2nd international Symposium on Memory Management, 2000*
- *David Detlefs, Christine Flood, Steve Heller, Tony Printezis. Garbage-first garbage collection. In Proceedings of the 4th International Symposium on MemoryManagement, 2004*
- *Cliff Click, Gil Tene, Michael Wolf. The Pauseless GC Algorithm. In Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005*
- *Christine Flood, Roman Kennke, Andrew Dinn, Andrew Haley, Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. 2016*
- *@RednaxelaFX*

THANKS

Thanks.
Question or Discussion?

BOSS直聘 研发五组 陈良柱
www.zhipin.com