Java HotSpot™ 虚拟机中无用存储单元的收集（GC）

John Coomes
Tony Printezis
Sun Microsystems 公司

演示目的

帮助你理解无用存储单元的收集，以及它如何与 Java™ 程序进行互操作，这样当你在设计、开发和配置应用程序时可以在充分了解信息的基础上作出选择

演讲人情况

- **John Coomes** 是 Sun Microsystems 公司的资深工程师
- 目前研究 HotSpot™ 虚拟机中的无用存储单元收集
- 已在 Java™ 2 平台的不同部分，以及它的标准版本方面工作了 5 年；在 HotSpot 方面工作三年
- 喜欢骑山地车

演讲人情况

- **Tony Printezis** 是 Sun Microsystems 实验室 Java™ 技术研究小组的成员
- 原来在苏格兰格拉斯哥大学计算机科学系教书
- 已从事 GC 方面的工作 5 年；
  编写了近乎并发的收集器的第一个版本
- 不喜欢骑山地车

我们正在努力出售的东西…

GC 是你的朋友

终结不是

议程

- 什么是自动内存管理以及为什么进行自动内存管理
- GC 的特点
- HotSpot 中的 GC
- 有利于 GC 程序设计
- 问与答

什么是自动内存管理？

- 对象分配
  - 新操作
- 无用存储单元的收集 (GC)
  - 回收未用的内存
  - 类的卸载
  - 弱引用处理及终结
  - 对象堆的布局

为什么进行自动内存管理？

- 简化程序
  - 消除对重新分配的需求
  - 防止内存泄漏
  - 简化数据类型接口
  - 启用正确的封装
- 保证编程语言的安全
  - 防止虚悬指针

议程

- 什么是自动内存管理以及为什么进行自动内存管理
- **GC 的特点**
- HotSpot 中的 GC
- 有利于 GC 的程序设计
- 问与答

GC 的设计目的

- 快速、快速、快速的分派
- 未用存储单元的及时回收
- 正在运行的应用程序的最短中断
  - 微小的、可预见的停顿
- 少量的系统开销 （空间、时间）
- 可升级到大型对象堆

GC 的‘真相’

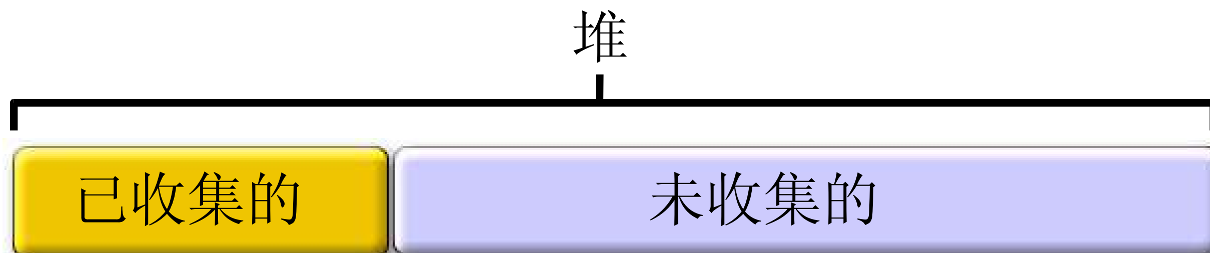
- 算法与策略的平衡
  - 堆的大小 vs. 收集时间开销
  - 停顿时间和应用程序的吞吐量
- 既然一种方法无法适合所有的……
  - 提供选择
  - 利用可调整的、可以自行调整的算法

GC 的特点

- 部分收集 vs. 完全收集
- 全部停止 vs. 并发
- 串行的 vs. 并行的

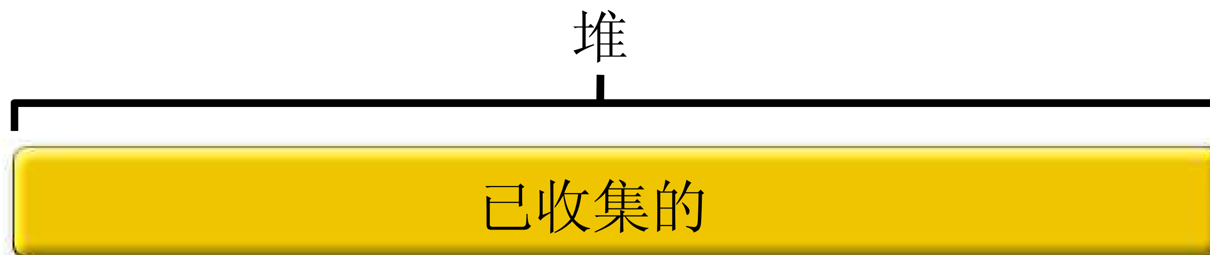
部分收集 vs. 完全收集

- 部分收集
  单独收集堆的子区域
  - + 利用对象生存期和无用存储单元密度
    - 每单位的 GC 工作回收更多的存储单元
  - – 必须追踪进入收集区的引用
    - 写屏蔽
      - 对执行时间的微弱影响
    - JIT 编译器必须予以配合



部分收集 vs. 完全收集

- 完全收集
  每次收集整个堆
  - 停顿时间变长
    - 与堆大小成正比
  + 无写屏障，执行更容易



全部停止 vs. 并发

- 全部停止条件下的 GC
  应用程序停止时完成全部 GC 工作
  - + GC 时对象图冻结
  - – 较长的停顿时间
- 并发式 GC
  大多数 GC 在应用程序运行的情况下完成
  - – GC 时对象图变化
    - 要求同步，更加复杂
  - – 占用空间更大（“浮动的无用单元”）
  - + 停顿较短

串行的 vs. 并行的

- 串行的
  - + 算法更简单
    - – 多余的 CPU （ > 1 ）在 GC 期间空闲
  - – GC 占用固定时钟时间较长
- 并行的
    - – 要求同步，更为复杂
  - + GC 使用多个 CPU
  - + GC 占用固定时钟时间较短

议程

- 什么是自动内存管理以及为什么进行自动内存管理
- GC 的特点
- HotSpot 中的 GC
- 有利于 GC 的程序设计
- 问与答

HotSpot 中的 GC

- 快速的存储单元分配
- 部分收集
  - 分代
- 全部停止，或者近乎并发[1]
- 串行的或者并行的[1]

[1]可用，但默认情况下不启用

快速分配

- 新对象被分派到“托管单元中”
  - 托管单元通常是本地线程
  - 大的对象可能被分配到别的地方
- 存储单元分配：修改单个指针
  - 通常通过编译器内嵌存在
- **new java.lang.Object()**
  大约是 10 个本地指令
- GC 激活的存储单元快速分配

HotSpot 中全部停止式的 GC

- VM 跟踪线程状态
  - 运行编译过的字节代码
  - 解释程序字节代码
  - 运行本地代码
  - 锁定等待
- 只有执行字节代码的线程停止
  - 本地代码形式的线程继续运行
- 一些收集工作可以并发

HotSpot 中的部分 GC

堆划分为“代”

- 弱分代假设 ：
  - 大多数对象存活时间短
  - 旧对象很少引用新对象
- 新生代：托管单元
  - 经常性的收集
  - 持续时间较短
- 旧代：对象在一次或多次 GC 中存活
  - 不经常收集
  - 持续时间较长

HotSpot 中的新生代

- 大多数对象存活期是很短的
  - 在新生代中诞生、生长、死亡
- 结点复制收集器
  - 当无用存储单元比例较高时效率最高
- 串行的和并行的收集策略
  - 默认的是串行的 GC

HotSpot 中并行 GC

- 并行的新生代收集
  - 利用多个 CPU
  - 提高吞吐量，停顿时间
  - 与堆的大小成比例
  - 通过工作挪用达到负载平衡
  - 顾客引语：
    “近几年来我所看过的最好的 VM 改进”
- 使用 -XX:+Use 启用并行 GC
- 旧代收集工作是串行完成的

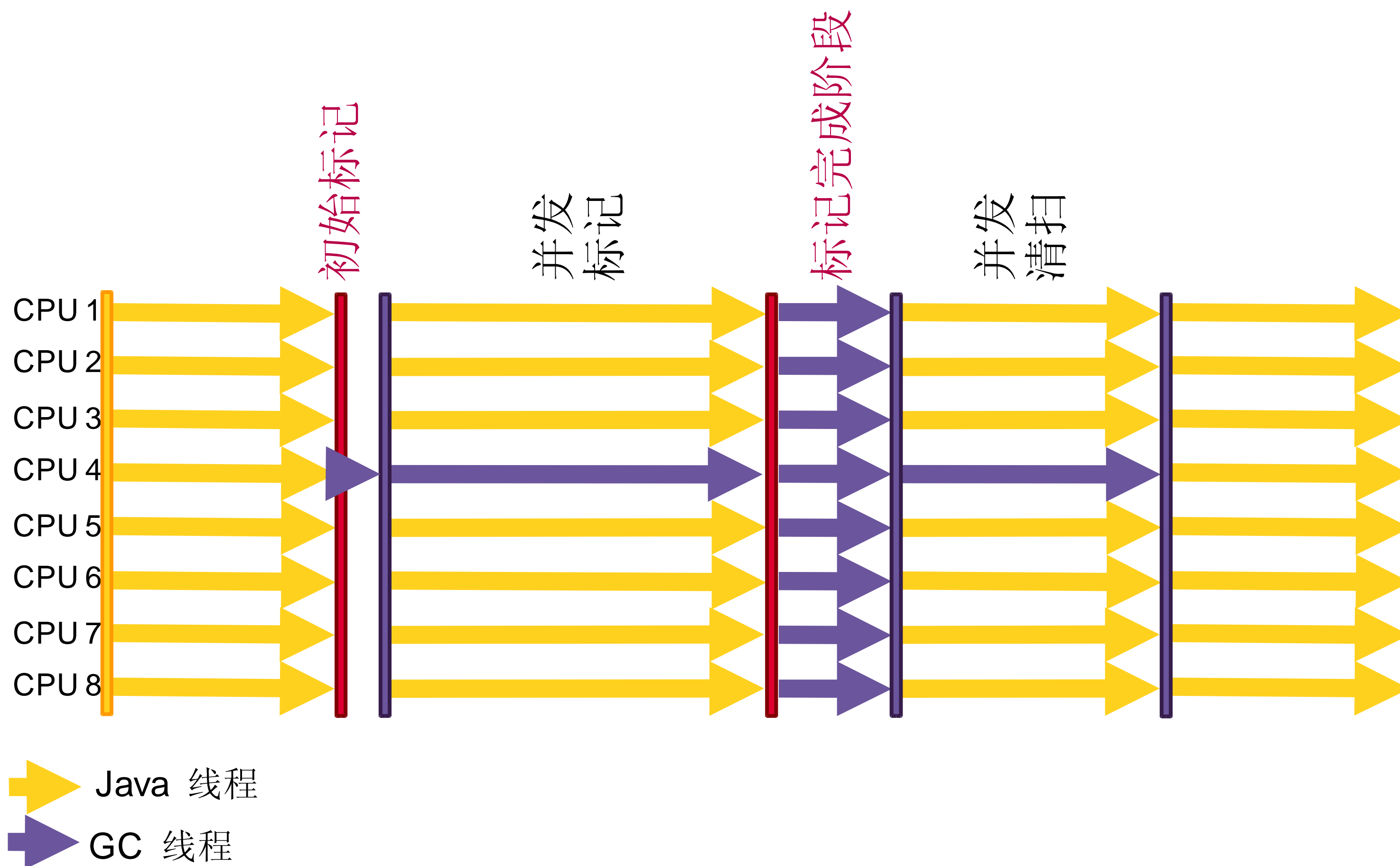
演示

HotSpot 中的并发收集

- 旧生代收集器大都为并发型
  - 并发标记 / 清扫收集器，或者 CMS
  - 大量 GC 工作在应用程序运行状态下完成
    - 较短的停顿时间
  - 旧生代对象并不移动
    - 分配而不是进行指针缓冲
- 使用 -XX:+UseConc 启用标记 / 清扫 GC
- 新生代 GC
  - 如果 CPU 可用，默认将采用并行方式

HotSpot 中的并发收集

并发标记清扫阶段



演示

将来的改进

- 为了更好性能而必需的 GC 调整
  – 转向可自我调整的 VM
  – 可用的调整指南
    - http://java.sun.com/docs/hotspot/
- 更好的可观察性
  – JSR 174 进行监视及管理
- 并发收集器
  – 更多并行，更多并发
  – 更短的停顿

议程

- 什么是自动内存管理及为什么进行自动内存管理
- Gc 的特点
- HotSpot 中的 GC
- 有利于 GC 的程序设计
- 问与答

有利于 GC 的程序设计

- 终结
- 对象池
- 其它要考虑的事情

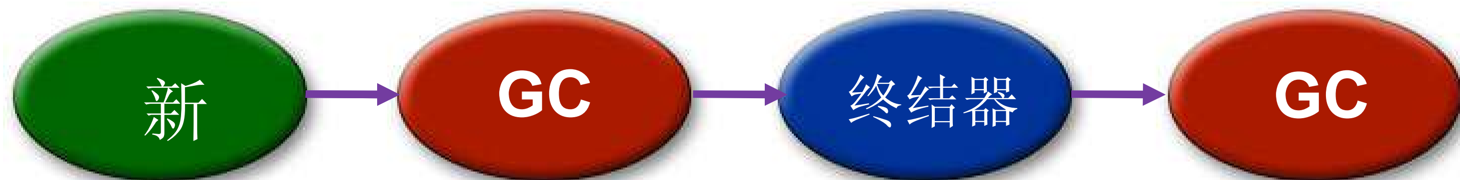
什么是终结？

- 外部资源清理分支指令
  - 文件描述符
  - 本地 GUI 状态模型
- 类过盈
  - **protected void finalize （） { ... }**
  - 在对象不可达后的某个未指定时间
  - **finalize** （）也许会被调用

终结工作如何进行？

每个实例

1. 分配时进行注册
2. 不可达时被排队
3. **finalize （）**方法被调用了吗
4. 再次不可达（是否在步骤 3 中恢复）
5. 存储器被回收了吗



演示

终结操作的影响

- 执行速度
  - 分配比较慢
  - 终结器的线程影响调度
- 堆大小
  - 内存保留时间更长
- 收集停顿
  - 发现与排队

有利于 GC 的终结

- 用来清理外部资源
- 建议
  - 限制可终结对象的数目
  - 重新组织类，这样可终结对象不包含多余数据
  - 在标准库中扩展的可终结对象要小心
  - GUI 组件，无缓冲
- 备选方案
  - 使用 `java.lang.ref.WeakReference` 没有终结器

对象池

- 人工内存管理
  - 分配串行化
  - 当前收集器支持并行分配
- 数据人工保持活力
  - 增加了无用存储单元收集器的工作压力
- 分解抽象数据类型
  - 谁对实例负责？
- 只在分配或者初始化开销比较大时使用

对象池范例

- ```
  class Node {
    private static Node head = null;
    private Node next;

    public static synchronized Node allocate（）  {
      if  （head == null）  return new Node（）  ;
      Node result = head;
      head = head.next;
      return result;
    }
    public static synchronized void free（Node n）
  {
      n.next = head;
      head = n;
    }
    ...
  }
  ```
  ```

实际问题

- 对象池从不削减
- 峰值有效数据 ~300MB
- 平均有效数据 ~100MB
- 问题
  - 由库生成的其它无用存储单元
  - GC 不太频繁，但需要处理 300MB 的数据
- 解决方案
  - 移除对象池；应用程序运行得更快

其它要考虑的问题

- 确定堆的适当大小
  - 最大的堆应该比工作区大……但是比可用的物理内存小
  - 为该系统留出可调整的空间
- 避免使用 **java.lang.System.gc** （）
- 尝试不同的收集算法
  - -XX:+ UseParallelGC
  - -XX:+ UseConcMarkSweepGC

总结

- GC 简化了 Java™ 程序
- 几种不同类型的 GC
  - 串行的，并行的，并发……
  - 每一种 GC 适合一个应用程序的子集
  - HotSpot 提供多种选择
- 避免终结，对象池
  - 作为最后的手段使用

我们希望你购买……

GC 是你的朋友

- 或者至少它可以是

我们希望你购买……

GC 是你的朋友

— 或者至少那是可能的

终结不是

—或者至少不可能是（你只是不知道而已）

问与答

Java™

JavaOne℠

Sun's 2003 Worldwide Java Developer Conference

Java™

java.sun.com/javaone/sf