GC 调整

高吞吐量 * 低延时 * 空间利用率

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

C vs. JAVA

老弟！ 好钢用到刀刃上，总感觉
GC 忙个不停一直占着资源在清理
“垃圾”

C
Language

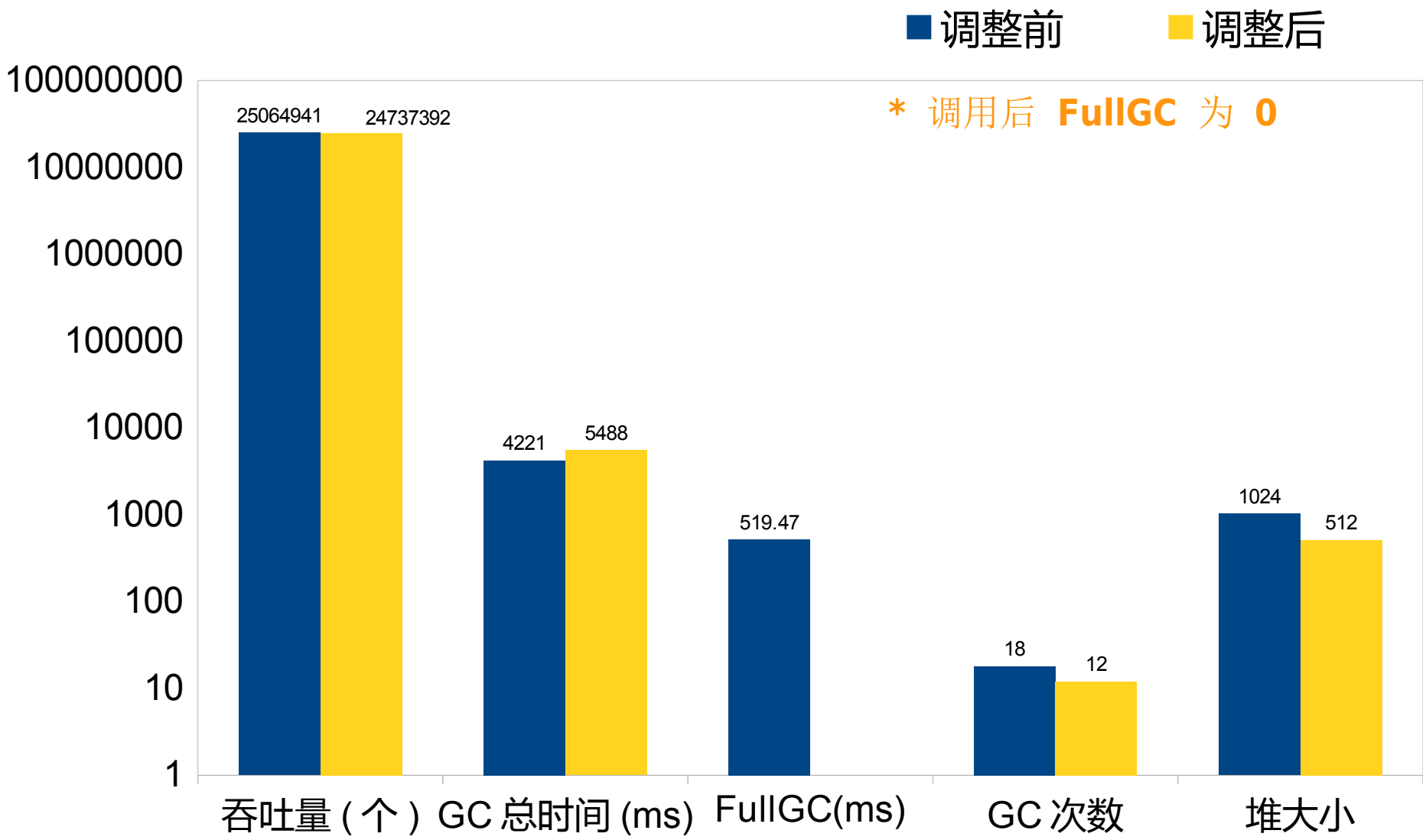
老兄！ malloc / free
就象一门艺术

各异的业务需求

- 默认选项不可能适应所有需求
- 有时需要严格控制停顿时间（SLA)
- 有时需要避免 FULL GC
- 有时需要最大化吞吐量
- 有时想完全避免 GC
- 有时需要控制堆大小
- 有时想尝试最新技术
- 有时 ...

DEMO

- 每 **1000~10000** 对象 sleep **1~10**ms
- Short term/Long term objects = **3/1**
- 测试时长 **90**s


调整前
调整后
* 调用后 FullGC 为 0
100000000
10000000
1000000
100000
10000
1000
100
10
1
25064941
24737392
4221
5488
519.47
18
12
1024
512
吞吐量 (个)
GC 总时间 (ms)
FullGC(ms)
GC 次数
堆大小

结果

- 避免了 Full GC
  - 牺牲小量的 throughput
  - 稍微增加了总 GC 时间
  - 堆空间减小
  - GC 次数减少

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

GC 概念	说明
Ergonomic 工效学	• JAVA 5 引入 • 基于平台 / 操作系统信息 • 包办 GC/HEAP/VM 配置 • 减少人工调整
Generation 分代式	• 堆空间分代： young, old, permanent • 集中精力“关怀”年轻代 • 延缓“衰老” • 两个假说 • i) 95% 新分配对象都“短命” • ii) 很少有对象 年长代引用年轻代

GC 概念	说明
Compact 压实式	• 堆空间内移动 Live 对象 • 连续的空闲空间 • Bump-the-pointer • 避免内存碎片
Non-compact 无压实式	• 回收但不移动所有的 live 对象 • 内存“千疮百孔”，内存碎片 • Free-lists 空闲空间 • 分配时间复杂度升高
Copy 拷贝式	• 两个拷贝空间 from, to • 总有一块空闲 • 拷贝时间开销大，当堆增大时

GC 概念	说明
Minor GC	• 也叫年轻代 GC • 只在年轻代空间中执行
Major GC	• 也叫 Full GC • 年长代或永久代满时 • 操作整个堆包括年轻代
Soft Real Time	• Pause Time 比较高可预测性 • 但还不是 Hard Real Time • 因为在资源有限的情况下维持吞吐量
Promptness 敏捷性	• 垃圾清理速度 • 即内存回收速度 • 想过 Distributed Garbage GC 吗？

GC 名称	特性
Serial 顺序式	• stop the world 挂起应用线程 • 一次性进行 • 仅利用 1 个 CPU
Parallel 并行式	• 当今主流硬件：廉价内存和多核 CPU • 任务分割，多个 CPU 执行 • 只有 minor GC 并行 • 内存碎片 (fragmentation)
Parallel Old 年长代并行式	• 又叫做 Parallel Compacting GC • major GC 并行 • pause time 降低 • Compacting 避免内存碎片

GC 名称	特性
CMS 并发标记清扫	• 与应用线程并发进行，不必挂起（理论上） • 更大的开销以及更大的堆空间 • Mark 追踪对象引用图谱：寻找 live 对象 • Sweep 扫除垃圾：消亡对象
G1 垃圾优先	• JAVA 7 new star • 实验性引入 Java SE 6 Update 14 • 年轻代与年长代 regions 无严格边界 • 首先选择垃圾最多的区域 （名字来历） • 长期目标代替 Concurrent Mark-Sweep GC

Ergonomic (工效学)

- **Server-class machine**
  - 2 or 2+ CPU (Core)
  - 2 or 2+ G RAM
  - Non-32-bit Windows
  - Java -server (default)
  - Java -client 覆盖 default
  - parallel GC
  - Xms = [1/64, 1G] RAM
  - Xmx = [¼, 1G] RAM

Ergonomic (工效学)

- **Client-class machine**
  - Java -client
  - Java -server 不起作用
  - Serial GC
  - Xms = 4MB
  - Xmx = 64MB

Ergonomic (工效学)

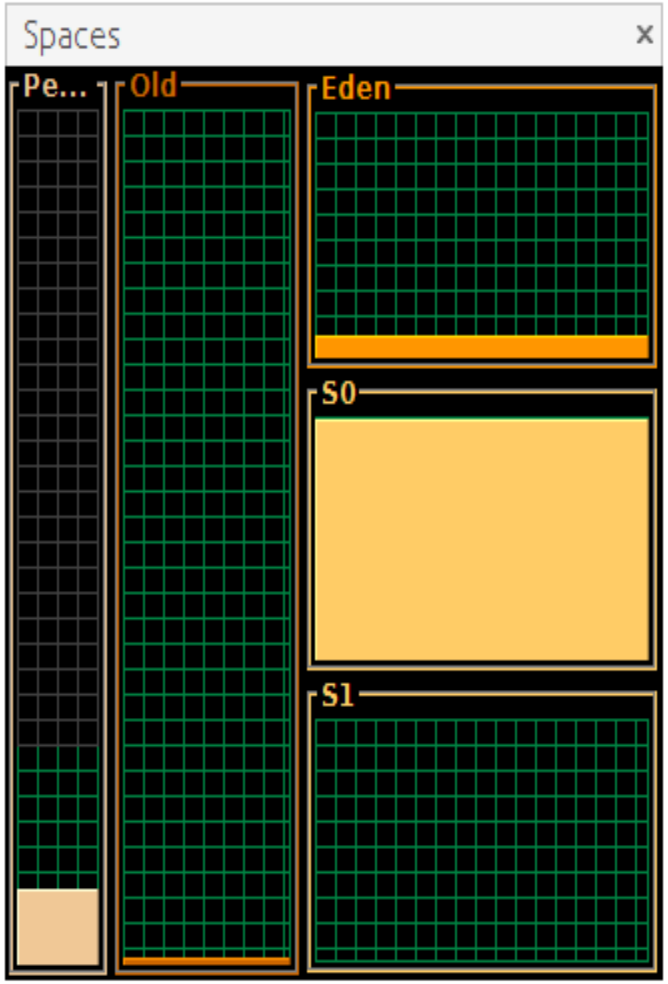
- **Java 6 u18**
  - Java -client
    - **if** RAM < 1 G Xmx = [½, 192MB], Xms=8M
    - **Else** Xms=1/64 RAM, Xmx = ¼ RAM
    - 年轻代 = 1/3 total heap

分代式 GC

- **Permanent Generation** 永久代
  - 老不死？其实不是了
- **Old Generation** 年老代
  - 有些文档称 tenured generation
- **Young Generation** 年轻代
  - Eden 伊甸园 新生对象的乐土
  - S0 幸存者空间 0 (Survivor Spaces 0)
  - S1 幸存者空间 1 (Survivor Spaces 1)

分代式 GC 部局

- **Eden**
  - 大多数对象在 Eden 中分配，age=0
  - 超大对象 年长代 中分配
- **S0/S1**
  - 其一总为空的
  - 另一个存有 Eden 中存活下来的对象
  - 当 Eden 充满，minor GC
  - 拷贝 GC 的来历
  - age++
- **Old**
  - Survivor Space 对象提升
- **Permanent**
  - JVM 的元数据 (metadata) 和自身的“看家”对象
  - 直接从 JVM 分配，而不是 new Object()
  - 类加载器 (class loader) （如，应用服务器）
  - 动态生成类



GC 分代“传说”

- 不同的代空间 不同年龄的对象
- 当某个代空间填满时， GC 在该代中进行
- 所有对象在 Eden 中出生 (分配)
- 不幸的是，大多数对象早早在哪里夭折
- 当伊甸园“尸首遍地”时，不得不进行一次小范围的清理 (minor GC) ；
- 幸存者被送到 Survivor space 。
- minor GC 只在年轻代中进行

GC 分代“传说” Cont.

- 一次 minor GC 过后
  - 足够成熟的对象被送往 年长代
  - 不够成熟的继续拷贝到另一 Survivor Space
  - Eden 又重新开始新的轮回
- 渐渐地，幸存者越来越多
  - 他们在足够老时都送往 年长代
  - 但那里的空间也是有限的
  - 当 年长代 充满时，一场大规模清理 (major GC) 不可避免
  - 整个堆空间进行：永久代 / 年长代 / 永久代
  - 持续的时间比 minor GC 长得多

Java 7 New

- **Escape Analysis**
  - JIT Compiler
  - 用 Stack 分配代替
  - 同步锁排除
- **Compressed Oops**
  - Oop: ordinary object pointer
  - 用 32bit 指针模拟 64bit 指针，减少引用大小 (8B to 4B)
- **NUMA**
  - Non Uniform Memory Access
  - 近水楼台原则
  - Latest Solaris & Linux only

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

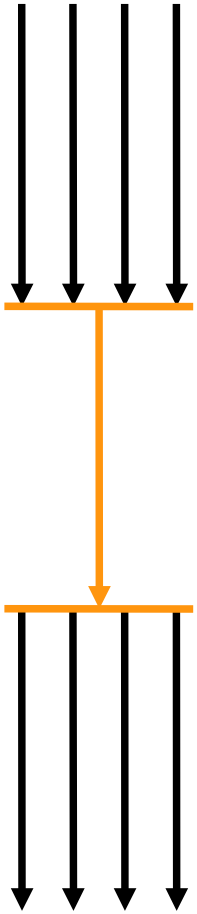
Serial GC

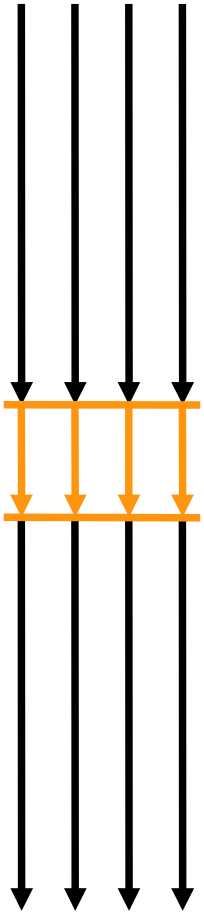
- **minor GC**
  - Eden 中的 live 对象被拷贝到空闲的 Survivor Space
  - 太大没法在 survivor space 存下的直接送往年长代
  - live 对象 , 分两成两组
    - 相对还是比较 年轻 的，送往空闲的 Survivor Space
    - 已经足够老的送往 年长代
  - 当 minor GC 结束后
    - Eden 清空
    - 先前非空的 Survivor Space 被清空
    - 先前空的 Survivor Space 存放幸存的对象
- **major GC**
  - 年长 和 永久代代都是通过顺序式 mark-sweep-compact 算法进行
  - mark 阶段标识哪些对象仍然 live
  - sweep 阶段清除没有被标识的对象（即垃圾）
  - compact 将 live 对象移到各个堆的开头，剩下的一边是连续的空闲空间

Parallel GC

- **顺应**当今主流硬件：**廉价**内存和多核 **CPU**
- **minor GC**
  - 算法同 Serial GC 的 minor GC 一样
  - 分而治之，充分利用所有的 CPU
  - 仍然 stop the world
  - 只不过是采用并行的版本，时间大大缩短
  - 吞吐量提升
- **major GC**
  - 算法同 Serial GC 的 major GC 一样：顺序式 mark-sweep-compact

Serial vs. Parallel





Stop the world

Parallel Old GC

- **Java5.0 update 6** 引入
- **Minor GC** 同 parallel GC
- **Major GC**
  - stop the world, 化分固定的区域
  - **mark** 阶段
    - 多个GC 处理线程
    - 并行
    - live对象大小和位置信息存放在其所在的区域
  - **summary** 阶段
    - 操作目标是区域(region)，而不是对象
    - compacting 致使堆的前部分应该是 live 对象密集区
    - 寻找平衡 : dense prefix 左边不处理右边压实
  - **compact** 阶段
    - 并行
    - 右边空出连续的空闲堆空间

Concurrent Mark-Sweep GC

- 产生背景
  - Serial / Parallel GC 都是 stop the world
  - Parallel 采用多 CPU 并行来缩短时间
  - 堆越来越大
  - SLA (Service Level Agreement) 很难保证
  - 继续寻找低延时 (low-latency) GC
  - 以牺牲吞吐量 (throughput) 为代价

Concurrent Mark-Sweep GC

- **Minor GC** 同 parallel GC
- **Major GC**
  - 大部分时间不需要挂起应用 (即 concurrent 的来历)
  - **initial mark 阶段**
    - 短暂的 pause
    - 单线程标识所有从应用代码可直达的 初始 live 对象
  - **concurrent mark 阶段**
    - 并发地标识可以从上述初始 live 对象到达的 live 对象
  - **concurrent preclean 阶段**
    - 查找在 concurrent mark 更新的对象
      - 从 年轻代 promotion 过来的
      - 新分配的
      - 所有其他更新的
    - 以降低后续的 stop-the-world 'remark' 阶段中断时间

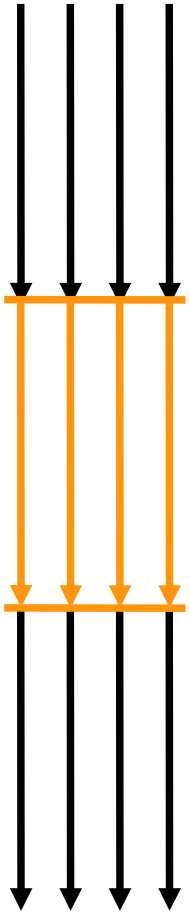
Concurrent Mark-Sweep GC

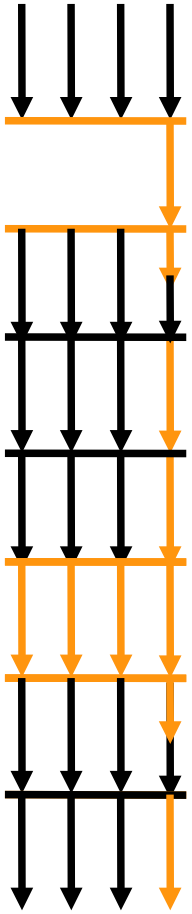
- **Major GC (CONT.)**
  - **concurrent abortable preclean 阶段**
    - concurrent preclean 之后，如果 Eden 的占有大小达到 -XX:CMSScheduleRemarkEdenSizeThreshold=<n> 时
      - 触发，直到 Eden 占用率到百分比降至 -XX:CMSScheduleRemarkEdenPenetration=<n>
      - 本阶段可随时中断， abortable 来历
    - 否则，跳过本阶段，直接进入到 multi-thread remark 阶段。
    - 引入本阶段目的
      - 延迟进入 remark 阶段的时间，降低频率，以减轻 remark 的压力
      - survivor 提升之后，堆上存在很多“灰”对象（GC 算法为对象涂色： white/gray/black ）
      - 灰对象需要重新扫描

Concurrent Mark-Sweep GC

- **Major GC (CONT.)**
  - **并发 remark 阶段**
    - concurrent mark 阶段找到的并不是所有的 live 对象
    - 应用状态不停地改变
    - 短暂的 pause 阶段，重新标识确保找全所有 live 对象
  - **concurrent sweep 阶段**
    - 清除所有未标识的对象
  - **concurrent reset 阶段**
    - 重新计算空闲的堆数据结构

SMS vs. CMS





Initial mark

Concurrent mark

Concurrent preClean

Concurrent abortable preClean

Remark

Concurrent sweep

Concurrent reset

Stop the world

Concurrent Mark-Sweep GC

- **Major GC (CONT.)**
  - 无压实 (compacting) 阶段，吃惊吗？
  - 碎片 (fragmentation) 是肯定的，空闲的堆空间不再连续
  - Free-list, not bump-the-pointer
    - 需空间存储 list 结构，空间分配 new Object() 效率大大降低
    - 影响 minor GC 的效率：大多数年长代 对象是从 年轻代 提升过来的 (分配对象总开销大大增加)
  - Footprint 更大：在收集的同时应用可以继续运行 (分配对象)
  - 期待在年长代填满之前就开始执行
  - 严重时衰退成 **stop the world** mark-sweep-compact

即 serial / parallel GC 使用算法

Concurrent Mark-Sweep GC

- **Major GC (CONT.)**
  - 为避免衰退成 stop-the-world
  - 动态统计年长代 变满的趋势：增加了 GC 的开销
  - –XX:CMSInitiatingOccupancyFraction=<n> 触发百分比
  - –XX:+CMSIncrementalMode 增量式并发标记清扫 (iCMS)
  - 并发阶段 (concurrent mark/sweep ）增量 (Increment) 进行
    - 周期地打断
    - 将总体并发处理时间分成小块 (icms_dc (**duty cycles**))
    - 在每次 minor GC 之间进行
    - -XX:MaxGCPauseMillis=<n> 只是一个 hint

Concurrent Mark-Sweep GC

// It is so complicated ...

Garbage First (G1)

```
/**
 * So, G1 is coming.
 * <p>
 * But it still need time to mature.
 * Hope I'll share with you in the future...
 */
```

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

梦幻 GC

- 低开销 (low overhead)
  - 高吞吐量 (high throughput)
- 停顿时间短 (low pause time)
- 空间利用率高 (space efficient)
- 实际往往只能 **3** 选 **2**

开销低

- 开销：花在 GC 上的时间
- 低开销意味着高吞量
- 若 GC 频繁，难怪上述 C 老兄会嘲笑
  - 尽量减少 Full GC 次数
  - 当堆很大时，停顿时间成比例升高

停顿时间短

- 完全并发 GC 只存在理论上的可能
- 停顿还是秒级的，除非非常小心
- GC 算法的使命
- 或提高 GC 频率，以降低吞吐量为代价

空间利用率高

- 低 footprint
- 完成 GC 所需的附加堆空间
- 硬件资源永远稀缺
  - 寄存器
  - CPU cache line: L1/2/3
  - 内存
  - Mmap
  - GC 占去越多，真正用来干活的就越少

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

通用策略

- 来个超大号的堆总没错！
  - 最大值 <= ½ 系统内存
  - 除非你跑着玩
  - 另一半留给操作系统看家进程和 mmap
  - +20~40% 生命周期内最大内存占有有量
  - 但是，有时增加内存并不管用

通用策略

- 大小平衡 空间大 = GC 不频繁
  - 足够的时间给对象成为“垃圾”，否则
  - 年轻代：
    - 过早地搬到 Survivor Space
    - Survivor Space -> 年长代
    - 与延缓衰老矛盾
  - 年长代：
    - 年长代 GC 时间长与低延时目标背离
    - 用不了多久又有一大批对象消亡
    - 与一次尽量多清除些垃圾矛盾

通用策略

- 大小平衡　空间小＝ **GC** 时间短
  - 但 GC 总时间也许不成立
  - 当空间严重缺乏时
    - Finalize
    - Full GC
    - Swap if enabled
    - OOME(Out Of Memory Error)

通用策略

- 年轻代
  - 调整原则： minor GC 算法目标是 pause time 短
  - 整体空间大小（三选一）
    - -XX:NewSize=<n>：年轻代初始大小
      -XX:MaxNewSize=<n>：年轻代最大值
    - -Xmn n: 等同于 -XX:NewSize=<n> 和
      -XX:MaxNewSize=<n>（更可取）
    - XX:NewRatio=<ratio>: 年轻代与年长代的比率
      如 ratio =3 表示 年轻代/年长代 = 1/3
  - 增加 CPU 核心，增加年轻代大小以充分利用并行
  - 堆大小固定了，增加年轻代意味着减少年长代

通用策略

- 年轻代 **Eden** 空间大小
  - 关系到 minor GC 的频率
    - 越小越容易填满，频率越高
    - 不利于对象成为垃圾
  - 关系到在伊甸园中夭折对象的比例
    - 在 Major GC 中耗费力气
  - 增加 Eden 大小并不总是减小 minor GC 的时间
    - 拷贝 Eden 中的 age=0 的对象到 survivor space TO
    - 处理 survivor space FROM 中的 age>0 的对象

通用策略

- 年轻代 **Survivor** 空间大小
  - 存在两个 Survivor: from, to
  - -XX:SurvivorRatio=<ratio>
    - 单个 Survivor 与 Eden 的比例
    - ratio=6 表示 每个 Survivor / Eden = 1/6，即每个 Survivor 是整个年轻代的 1/8，因为有两个 Survivor Space:

```
2s + e = y
s/e = 1/6
2s + 6s = y
S = y/8
```

通用策略

- 年轻代 **Survivor** 空间大小
  - -XX:TargetSurvivorRatio=<percent>： 在每个 Survivor 占用率到达这一比率时，将被提升到年长代中
  - 自适应 (adaptive) GC 调整
    - -XX:InitialTenuringThreshold=<threshold>： 在被提升到年长代之前，对象可在年轻代中存活的 GC 次数初始值
    - -XX:MaxTenuringThreshold=<threshold>: 在被提升到年长代之前，对象可在年轻代中存活的 GC 次数最大值
  - -XX:+AlwaysTenure 从不将对象保留在 Survivor Space
    - 每次 minor GC 直接将对象送往 年长代
    - 适用于 mid/long time live 对象多，以避免在 Survivor Space 中来回拷贝的开销

通用策略

- 年轻代　空间大小权衡
  - mid/long live time 对象比例小
    - 尽可能长地呆在 survivor space
    - 在年轻代被回收，不必提升到年长代，避免频繁 major GC
  - mid/long live time 对象的比例大
    - 避免在 survivor space 中来回拷贝
    - 最终要进到年长代
  - 很难预言对象的生命周期
  - 在 survivor space 中拷贝比无谓地提升到年长代要好点
  - -XX:+PrintTenuringDistribution：监视 survivor size 的行为分布，统计出合适的 survivor 空间大小

通用策略

- 年长代大小
  - major GC 算法目标是空间利用率，而不是快速
  - 尽可能一次 GC 回收更多空间
  - 尽可能减少 major GC 频率
  - 容纳"稳态"活对象总大小 + 20~40%
  - -Xms == -Xmx
    - -Xmx<n> 最大堆大小（年轻代 + 年长代）
    - -Xms<n> 初始堆大小（年轻代 + 年长代）
    - -Xms != -Xmx (unusual)
    - 堆扩张或收缩触发 Full GC

通用策略

- 年长代大小
  - 如果可能，让空间大到容纳应用生命周期内所有对象，以避免 major GC
  - 如果可能，将 major GC 调度到低峰时段执行
  - 说起来容易，做起来难
    - 容易地预测 WEB 服务器的高峰与空闲状态负载的回归线么？
    - 容易地预测分布式存储系统的 read write 比么？
    - 使用 NetBeans Profile (free) / YourKit Profiler

通用策略

- 年长代大小
  - **-Xms != -Xmx** 适用场景
    - 应用"稳态"的活对象总大小 <= -Xms
    - 极小情况峰值负载 (如 WEB 服务器) 或数据集 (存储系统) 将达到 -Xmx 值
    - 以堆空间扩张的 Full GC 开销为代价

通用策略

- 永久代大小
  - -XX:PermSize != -XX:MaxPermSize
    - 永久代堆扩张 / 收缩需要 Full GC
  - -XX:PermSize=<n> 永久代 初始大小
  - -XX:MaxPermSize=<n> 永久代 最大值
  - 永久代 对 GC 影响不大
    - 在 major GC 时，永久代空间一同被清理
  - 动态加载类的场景请确保空间足够大
    - 避免 OutOfMemoryError: PermGen space ...
    - -XX:MaxPermSize=<N> 不幸地很难拿捏这个值
      - java -verbose:class / Profile it

通用策略

- 固定大小策略意味着以下选项不需要
  - -XX:YoungGenerationSizeIncrement=<m>
  - -XX:TenuredGenerationSizeIncrement=<n>
  - -XX:AdaptiveSizeDecrementScaleFactor=<r>

通用策略

- Footprint 不应该超过系统可用的物理内存
  - **Swap on**
    - 对于 JAVA 服务器应用，并不是好主意
    - swap in/out 或许可能缓解一会儿内存短缺
    - 程序将遭受 内存与 I/O 访问的时间的差异
    - JAVA 应用卡住的根源 (Elipse, NetBeans, etc.)
  - **Swap off**
    - 哈哈！ 嘣 , OOME
    - 个别 GC, 即使 swap 开启，如果大部分时间花在 swap in/out 蚂蚁搬家的话，也一样 OOME
  - 利用操作系统都有 **mmap I/O**
    - 不要贪心地将所有系统物理内存让 JVM 独吞
    - 不要失去来自操作系统的免费而超优化的缓存功能

Parallel GC / Parallel Old GC

- 为当今主流硬件开发
- 采用 ergonomics
- 通常它自动调整过了，并且足够好
- 首先按照前述调整 年轻代
- 通常，它认为承载它的系统只有它一个 JVM

Parallel GC / Parallel Old GC

- XX:ParallelGCThreads=<n> 指定并行的线程数
  - 系统是否有多个 JVM
  - 系统的处理器核心个数
  - 处理器是否支持一个核心多个硬件线程 比如：
    - SUN/ORACLE UltraSPARC T1/T2
    - INTEL HT
  - 尽量降低 major GC 的频率
  - 如用于 low-pause 环境
    - 最大化 heap 空间
    - 避免或最小化 Survivor promotion 来避免 Full GC

CMS GC

- 按照前述调整年轻代
- 别错误地认为 CMS 并发能力发生在年长代而忽略年轻代的调整
- 更加小心避免过早的 Survivor Space promotion
- CMS 采用 free lists, promotion 开销很大
- Promotion 越频繁越有可能造成堆碎片
- 重点调整 minor GC
  - 应用负载超过以往的最大值时，CMS 作为最后一道防线
  - 将 Full GC 安排在非关键时间段以减少堆碎片

CMS GC

- 无法避免堆碎片 **(fragmentation)**
  - 因为 non-compact, 内存“千疮百孔”
  - 找不到足够大的空间来完成分配请求，即使剩余空间总计大小大于分配请求大小
  - 分配器为了效率，采用近似策略，这近似值最终浪费不少空闲空间
  - 不同大小的“大对象”是元凶

CMS GC

- -XX:ParallelCMSThreads=<n>　并行的 CMS 线程数
- –XX:+CMSIncrementalMode 增量模式
- –XX:+CMSIncrementalPacing 增量模式下每次工作量
- 卸载 永久代中的类
  - 默认 CMS 不卸载永久代 中的类
  - -XX:+CMSClassUnloadingEnabled
  - -XX: +PermGenSweepingEnabled

CMS GC

- **CMS** 启动时机
  - 启动太早，GC 频繁及高并发的开销
  - 启动太晚，衰退成 Serial Mark-Sweep-Compact 模式
    - Full GC，没有利用到 CMS 的优势
  - -XX:+UseCMSInitiatingOccupancyOnly
    - 默认情况下，CMS 会自动统计来找到最佳启动时机，即 ergonomic

CMS GC

- **mid/long live time** 对象比例
- 比例小，可以晚些启动
  - 很少 Survivor Space promotion 发生
  - 年长代 增长缓慢
  - 低频率地 major GC (即 CMS GC)
- 比例大，相对早些启动
  - 很多 Survivor Space promotion 发生
  - 年长代 增长很快
  - 高频率地 major GC (即 CMS GC)

CMS GC

- **-XX:+UseCMSInitiatingOccupancyOnly**
  - ergonomic 不可能覆盖所有场景
- **-XX:CMSInitiatingOccupancyFraction=<percent>**
  - 当年长代占用率达到这一百分比时，启动 CMS GC
- **-XX:CMSInitiatingOccupancyFraction=<percent>**
  - Live 对象占有率百分
  - 此值应该比应用“稳态”下的 所有 live 对象大小大得多
  - 否则，CMS 将不停地发生
- **-XX:CMSInitiatingPermOccupancyFraction=<percent>**
  - 永久代占用率达到这一百分比时触发 GC
  - 前提是 -XX:+CMSClassUnloadingEnabled 激活了

CMS GC

- **System.gc()**
  - 也需进行一次 Full GC
  - -XX:+DisableExplicitGC 忽略 System.gc() 调用
  - 指定 System.gc() 采用 CMS 算法
    - -XX:+ExplicitGCInvokesConcurrent
    - -XX:+ExplicitGCInvokesConcurrentAndUnloadClasses
    - 有关 Soft/Weak/Phantom references / finalizer

CMS GC

- It is complicated
- It is hard to tune
- It is easy to make mistake
- So far, we don't have the workaround
- **Let's meet G1 ...**

G1 GC

- 仍然处于实验阶段
- 在最“坏”的时候并不比 iCMS 好
- 一些开关选项，开了又关，关了又开
- Java SE 7u2 选项开关
  - **-XX:+UnlockExperimentalVMOptions** 打开实验性选项
  - **-XX:+UseG1GC** 使用 G1GC
  - **-XX:MaxGCPauseMillis**=30 最大停顿时间
  - **-XX:GCPauseIntervalMillis**=200 GC 执行最短间隔时间

Serial GC

- 默认在 client-style 机器上
- -XX:+UseSerialGC
- Tumblr architecture
  - ID generator
  - RAM: 500m
  - SLA : < 1ms
  - Load: 10,000 requests/s

GC LOG ?

- 也许再需要 2 小时才能讲清楚
  - 乏味且邪恶
  - 但是往往成为最后的救命稻草
  - Please refer [interpret-gc-log.doc]

Any Else ...

- 使用最新的 JDK
- Read every release note
- Read JavaOne sessions
- Subscribe NewsLetters
- 站在巨人的肩膀上
- ...
- 熟能生巧 (no pain no gain)

为何调整

GC 概念

GC 算法

调整目标

调整策略

演示用例

DEMO 准备

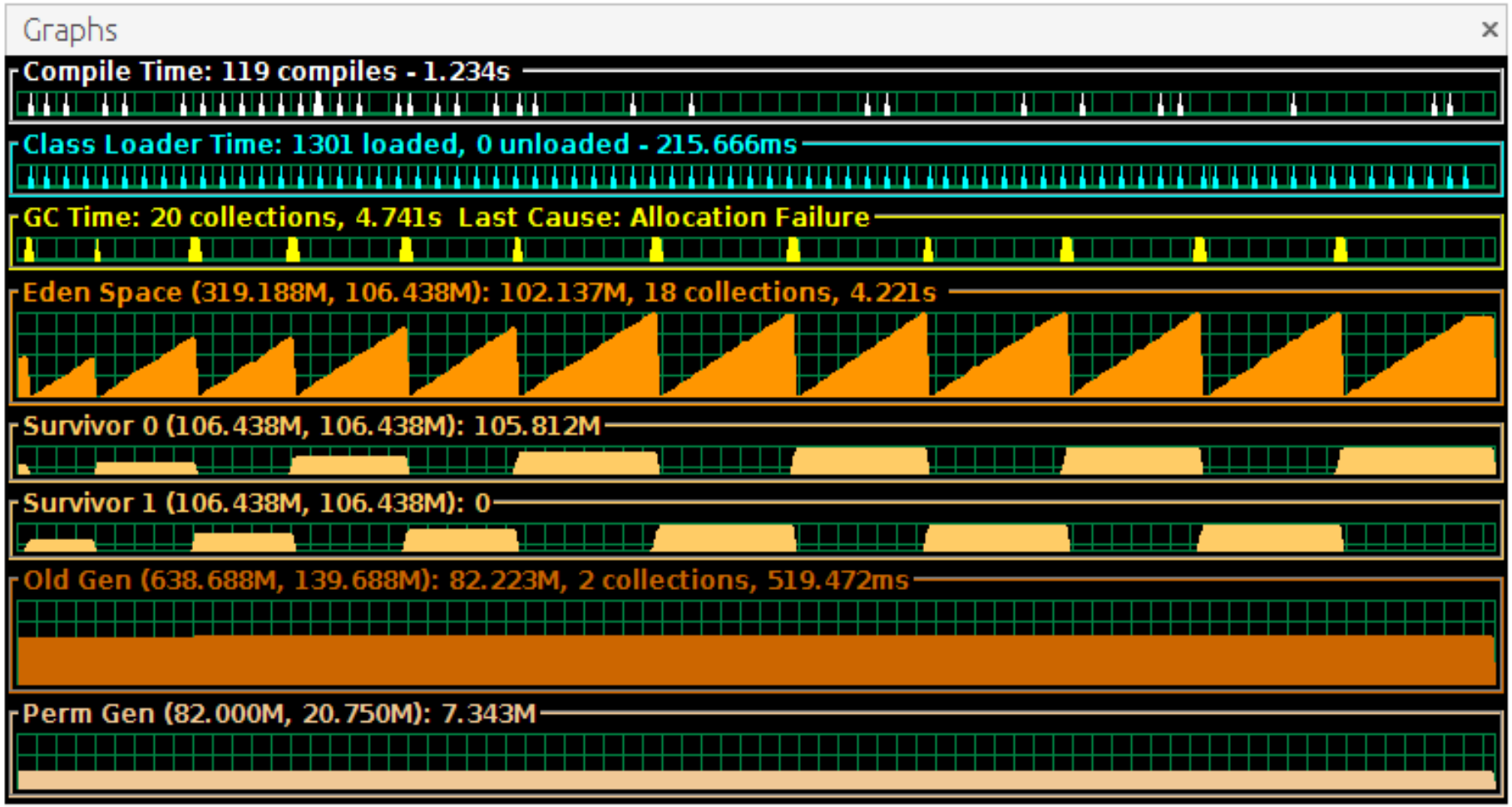
-  下载 VisualVM with VisualGC plugin
- github SOCIAL CODING 下载 demo source code
- Java 当然，装上 JAVA 6 / JAVA 7

DEMO 1 ： high load

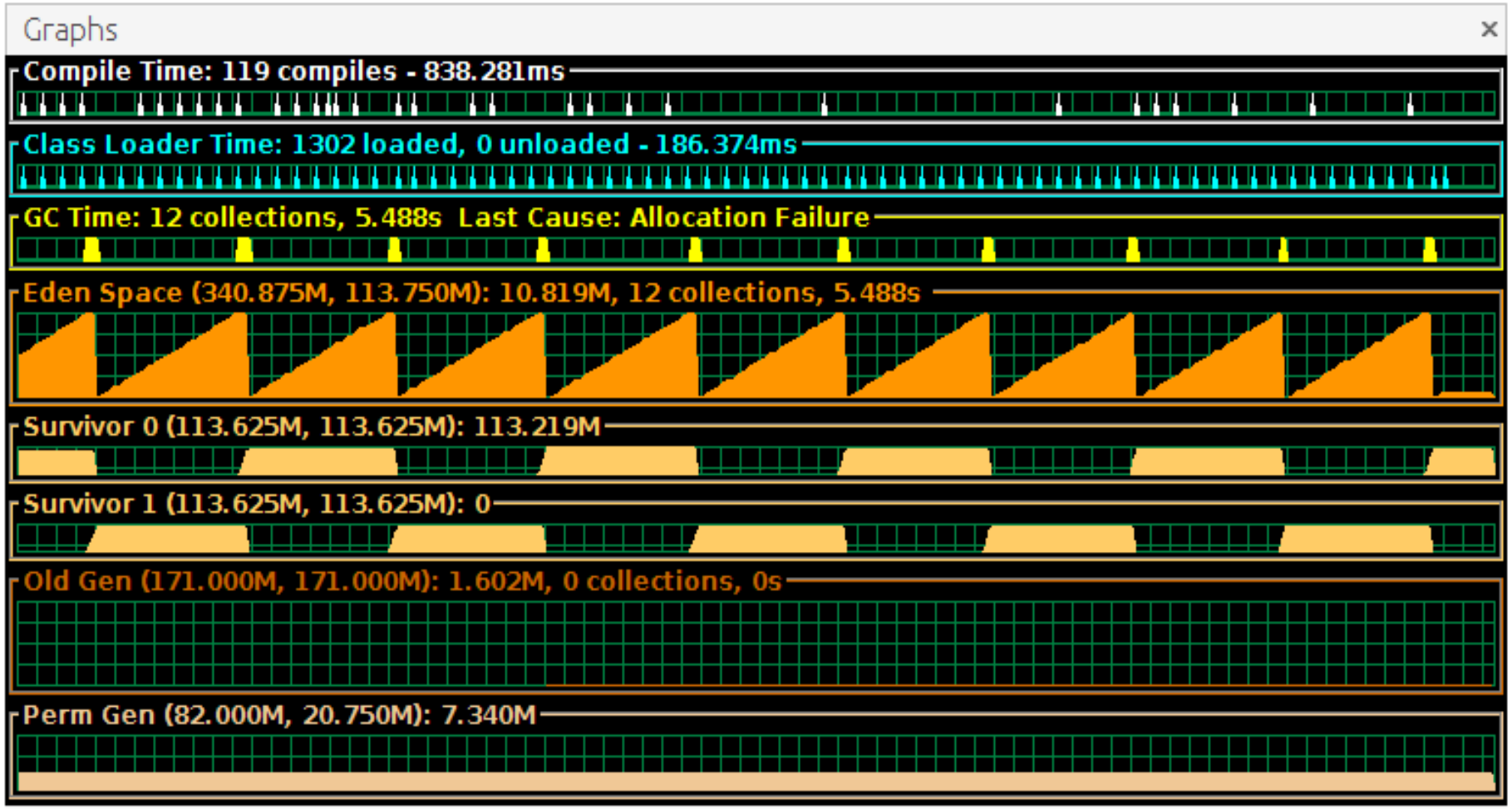
- 每 **1000~10000** 对象 sleep **1~10** ms
- Short term/Long term objects = **3/1**
- 测试时长 **90** s

```
1-1$ java -cp gc-demo.jar GcDemo
1-2$ java -XX:+UseNUMA -XX:
  +HeapDumpOnOutOfMemoryError -Xmn341m
  -Xms512m -Xmx512m -XX:SurvivorRatio=1 -cp
  gc-demo.jar GcDemo
```

DEMO 1: 1-1


Graphs
Compile Time: 119 compiles - 1.234s
Class Loader Time: 1301 loaded, 0 unloaded - 215.666ms
GC Time: 20 collections, 4.741s Last Cause: Allocation Failure
Eden Space (319.188M, 106.438M): 102.137M, 18 collections, 4.221s
Survivor 0 (106.438M, 106.438M): 105.812M
Survivor 1 (106.438M, 106.438M): 0
Old Gen (638.688M, 139.688M): 82.223M, 2 collections, 519.472ms
Perm Gen (82.000M, 20.750M): 7.343M

DEMO 1: 1-2

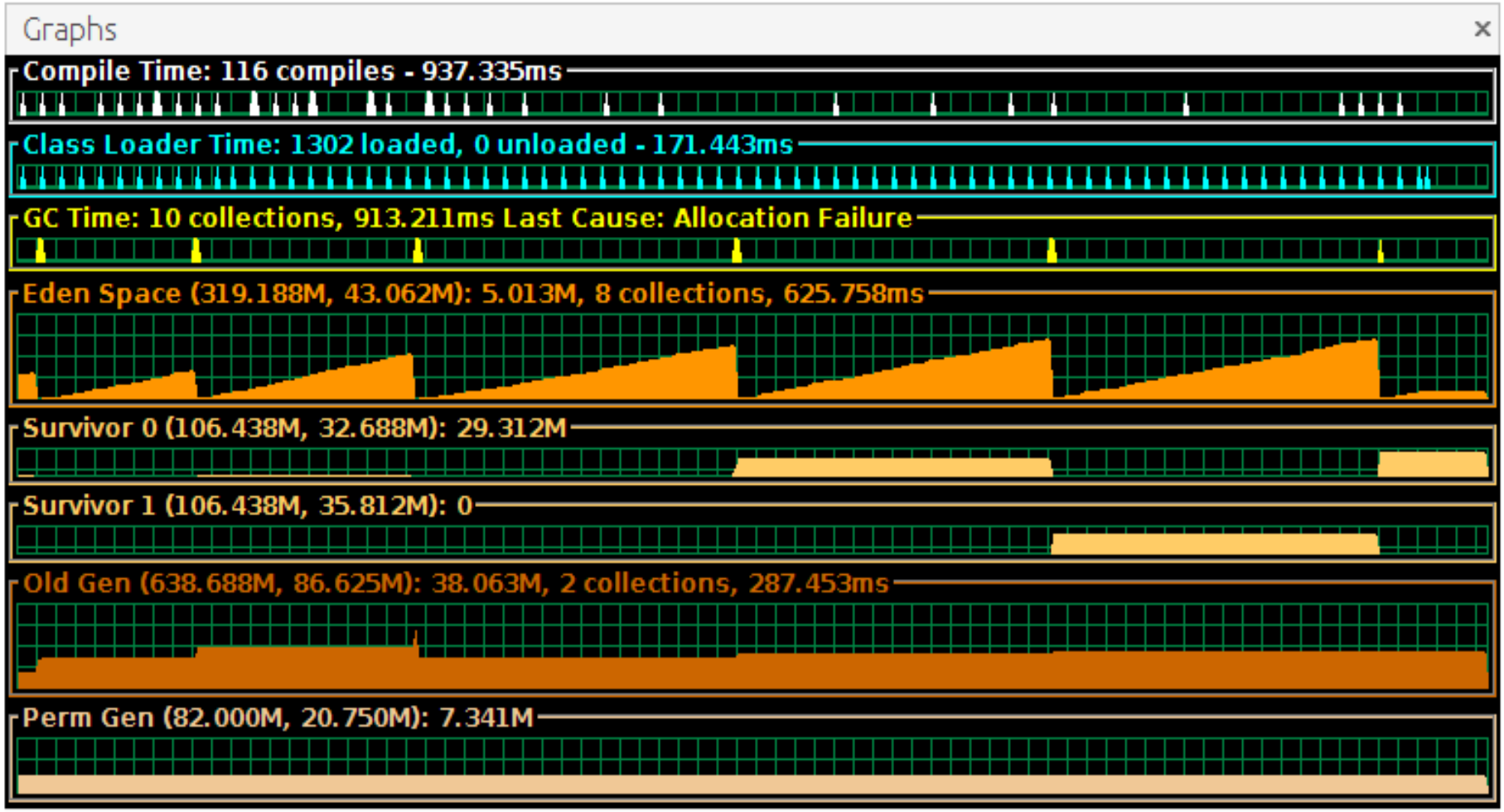

Graphs
Compile Time: 119 compiles - 838.281ms
Class Loader Time: 1302 loaded, 0 unloaded - 186.374ms
GC Time: 12 collections, 5.488s Last Cause: Allocation Failure
Eden Space (340.875M, 113.750M): 10.819M, 12 collections, 5.488s
Survivor 0 (113.625M, 113.625M): 113.219M
Survivor 1 (113.625M, 113.625M): 0
Old Gen (171.000M, 171.000M): 1.602M, 0 collections, 0s
Perm Gen (82.000M, 20.750M): 7.340M

DEMO 2: low load

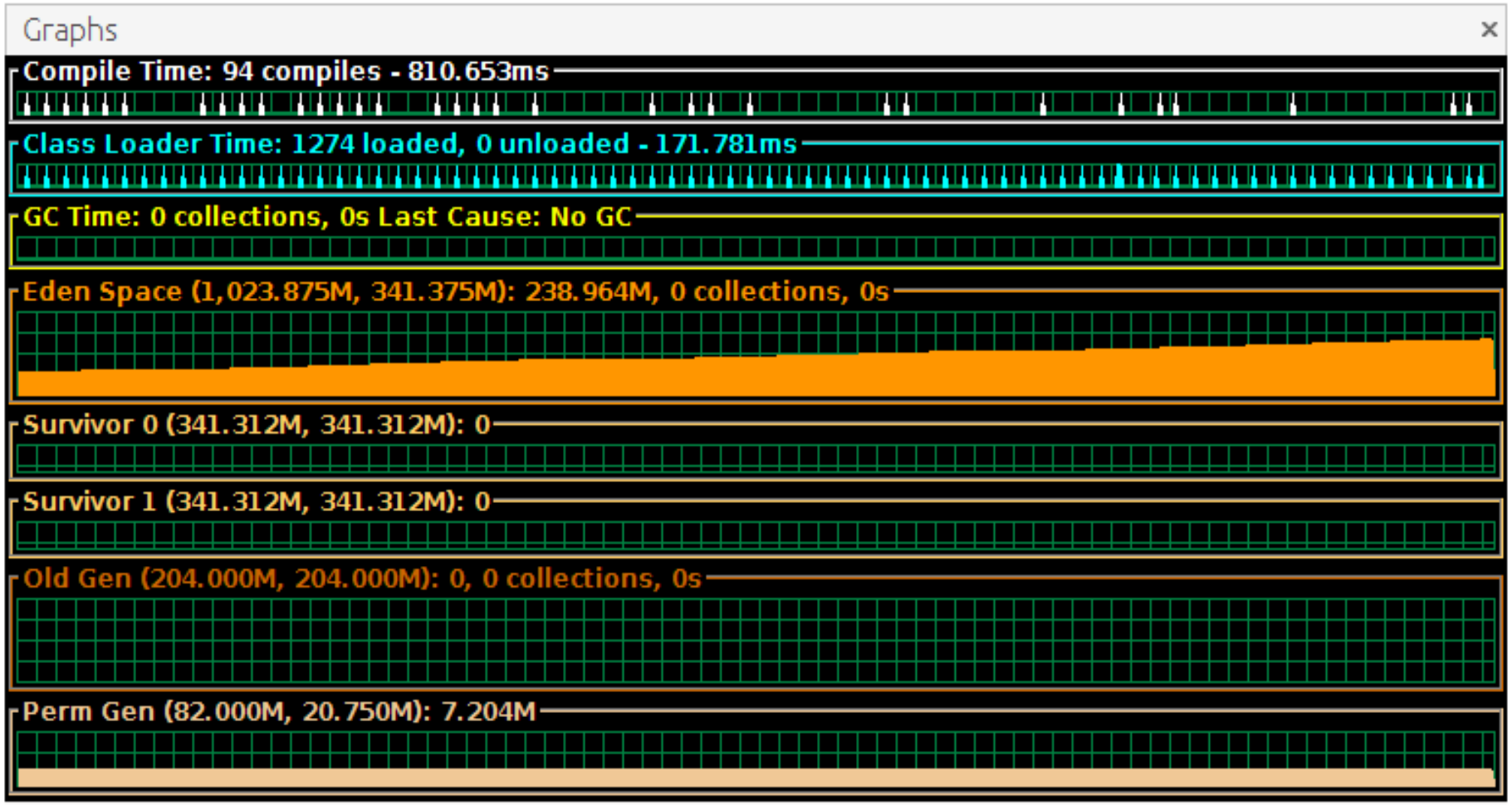
- 每 **1000~10000** 对象 sleep **1~100** ms
- Short term/Long term objects = **3/1**
- 测试时长 **90** s

```
2-1$ java -cp gc-demo.jar GcDemoLowLoad
2-2$ java -XX:+UseNUMA -XX:
  +HeapDumpOnOutOfMemoryError -Xmn1024m
  -Xms1228m -Xmx1228m -XX:SurvivorRatio=1
  -cp gc-demo.jar GcDemoLowLoad
```

DEMO 2: 2-1


Graphs
Compile Time: 116 compiles - 937.335ms
Class Loader Time: 1302 loaded, 0 unloaded - 171.443ms
GC Time: 10 collections, 913.211ms Last Cause: Allocation Failure
Eden Space (319.188M, 43.062M): 5.013M, 8 collections, 625.758ms
Survivor 0 (106.438M, 32.688M): 29.312M
Survivor 1 (106.438M, 35.812M): 0
Old Gen (638.688M, 86.625M): 38.063M, 2 collections, 287.453ms
Perm Gen (82.000M, 20.750M): 7.341M

DEMO 2: 2-2


Graphs
Compile Time: 94 compiles - 810.653ms
Class Loader Time: 1274 loaded, 0 unloaded - 171.781ms
GC Time: 0 collections, 0s Last Cause: No GC
Eden Space (1,023.875M, 341.375M): 238.964M, 0 collections, 0s
Survivor 0 (341.312M, 341.312M): 0
Survivor 1 (341.312M, 341.312M): 0
Old Gen (204.000M, 204.000M): 0, 0 collections, 0s
Perm Gen (82.000M, 20.750M): 7.204M

真的要调整么？

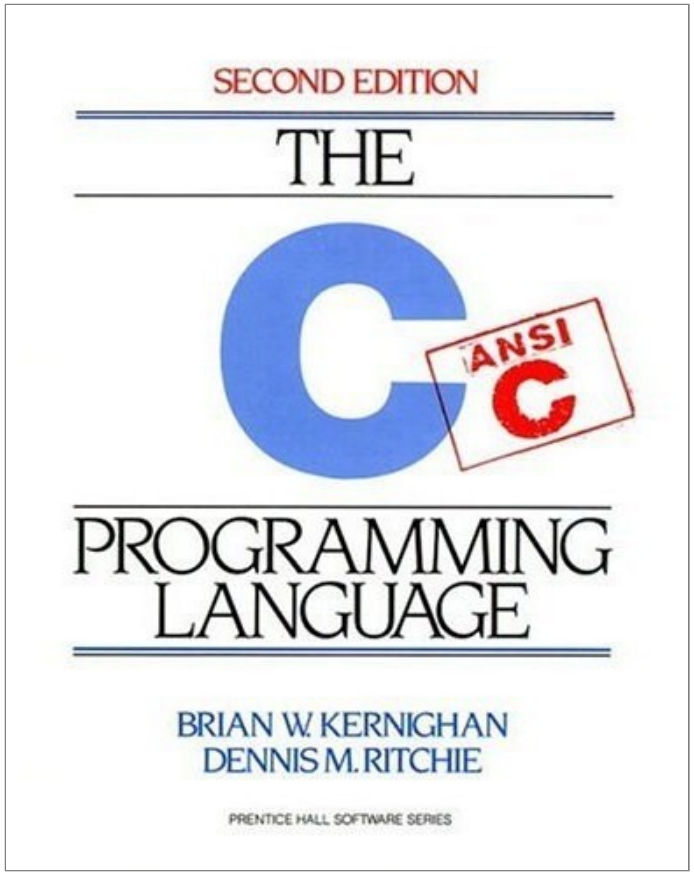
- 如果有足够多的预算：64bit-OS/CPU/RAM
- 如果数据量不够大
- 如果从来没有 OOME
- 如果没有严格的 SLA
- 如果没人抱怨
- 如果 ...
- 那么，忘掉它，睡懒觉去！

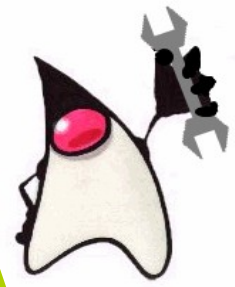
真的要调整么？

- 如果无论怎么调都达不到任何理想三选二目标
- 如果 GC 调整的努力 > 捕捉 C ”野指针”的努力
- 如果有一个出色的 C 团队
- 如果能在 BUG 到来前找出 C 的所有“死亡圣器”
- 如果 ...
- 那么， code in C please

Java GC 调整就象一门艺术

C Language





向 K&R 致敬

Reference

- Java SE HotSpot at a Glance
- Ergonomics in Java Virtual Machine
- VisualVM
- VisualGC
- Java HotSpot Garbage Collection G1
- Inside the Java Virtual Machine
- Garbage Collection : Algorithms for Automatic Dynamic Memory Management
- Oracle's guide with 80+ options
- Stas's guide with 800+ options

Q&A

@

QQ: 1756420205

THANK YOU

quest.run@gmail.com
http://hi.baidu.com/quest2run