【软件构造】第八章第一节 软件构造性能的度量原理

第八章第一节 软件构造性能的度量原理

本章是课程覆盖的第5个质量指标：时空性能

这是大家最熟悉的指标，虽然很重要，但并非软件构造中最重要的指标，当其他指标得以优化之后，再去考虑性能问题。

Outline

- 性能度量指标
- 存储性能
- 内存管理：
  - 对象管理模型：静态、堆、栈
  - 内存管理模型：
- Java垃圾回收机制
  - 基本概念
  - GC的四种基本算法
- JVM中的GC
- JVM GC性能调优

Notes

性能度量指标

- 时间性能
  - 每条指令、每个控制 结构、整个程序的执行时间
  - 不同语句或控制结构执行时间的分布情况
  - 时间瓶颈在哪里
- 空间性能
  - 每个变量、每个复杂结构、整个程序的内存消耗
  - 不同变量/数据结构的相对消耗
  - 空间瓶颈在哪里
  - 随时间的变化情况

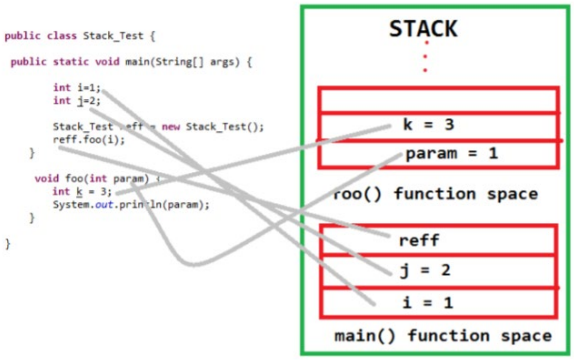
内存管理

【对象管理模型】

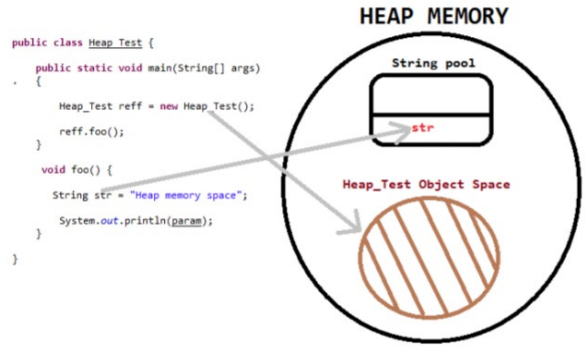
- 三者的差异在于：如何与何时在程序对象与内存对象之间建立联系
- 静态
  - 定义：静态内存是指在程序开始运行时由编译器分配的内存，它的分配是在程序开始编译时完成的，不占用CPU资源。

- 程序中的各种变量，在编译时系统已经为其分配了所需的内存空间，当该变量在作用域内使用完毕时，系统会自动释放所占用的内存空间；
- 不支持递归，不支持动态创建可变长的复杂数据类型；
- 在程序执行期内实体至多关联一个运行时对象
- eg: 基本类型，数组

- 动态-基于栈
  - 栈定义：方法调用和局部变量的存储位置，保存基本类型
    - 如果一个方法被调用，它的栈帧被放到调用栈的顶部
    - 栈帧保存方法的状态，包括执行哪行代码以及所有局部变量的值
    - 栈顶始终是当前运行方法
  - 一个实体可以在运行时连续地连接到多个对象，并且运行时机制以堆栈中的后进先出顺序分配和释放这些对象
  - 栈无法支持复杂数据结构

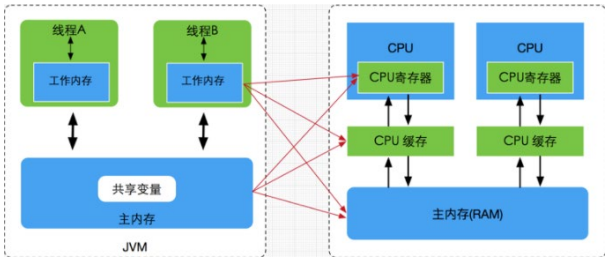


- 动态-基于堆
  - 堆定义：在一块内存里分为多个小块，每块包含 一个对象，或者未被占用
  - 自由模式的内存管理，动态分配，可管理复杂的动态数据结构
  - 代码中的一个变量可以在不同时间被关联到不同的内存对象上，无法在编译阶段确定。内存对象也可以进一步指向其他对象
  -

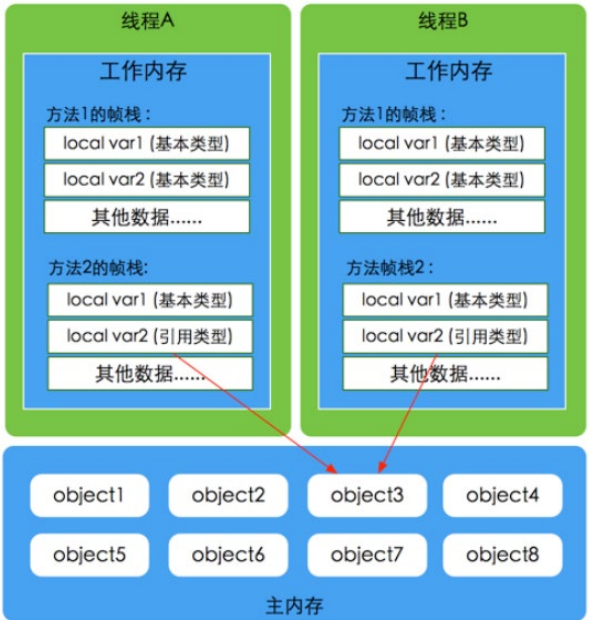


【Java内存管理模型（Java Memory Model）】

- Java内存模型(简称JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。即回答：JVM如何管理内存
  - 如何在堆上创建新对象
  - 当某个对象不再有reference指向他，如何阐述对象、释放内存

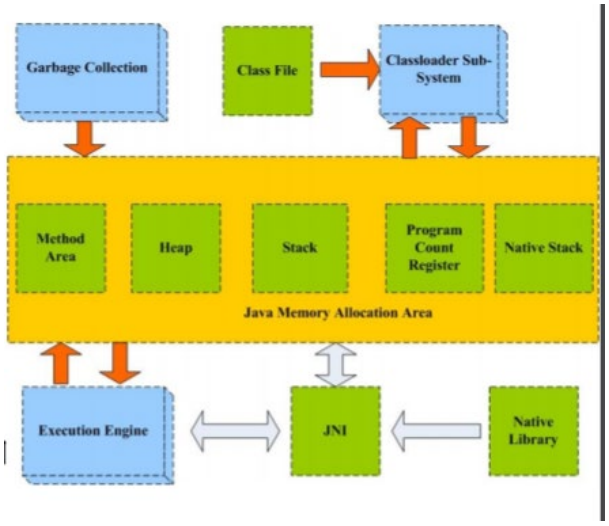


- 上图右侧是经过简化CPU与内存操作的硬件处理器简易图；多线程的执行最终都会映射到硬件处理器上进行执行，其框架结构与内存硬件框架关系如全图所示。
- 线程栈：每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，
  - 每个线程有自己的栈，管理其局部数据，各栈之间彼此不可见
  - 所有局部的基本数据类型都在栈上创建
  - 多线程之间传递数据，是通过复制而非引用
- 堆：所有对象（即使是局部变量的object）都是在堆上创建的
  - 主内存可被多线程共享



- 对上创建的对象可被所有线程共享引用；
- 可访问对象，就可以访问对象内的成员变量；
- 如果两个线程调用同一个对象上的某个方法，它们分别保留该方法的局部变量的拷贝；
- JMM例子：
  - 一个基本类型的局部变量，一直被保存在 线程栈 中
  - 局部变量引用的对象，保存在 线程栈 中，对象本身存在 堆 中
  - 对象可能包含方法，这些方法可能包含局部变量。 这些局部变量存储在线程栈上，并且该方法所属的对象存储在 堆
  - 对象的原始成员变量存储在堆上。 如果一个成员变量是一个对象的引用，它将被存储在堆
  - 静态类变量保存在堆上

更多内容参考 **zejiang**的博客



除了堆栈外，我们还需要JVM使用的本地方法栈、PC代码行号治时期 和 用于存储被VM加载的类信息、常量、静态变量等的Method Area

Java垃圾回收机制

【内存回收的三种方式】

①静态模式下的内存回收：在静态内存分配模式下，无需进行内存回收：所有都是已确定的。

②在栈模式下的内存回收：按block（某个方法）整体进行

③在堆模式下的内存回收：在heap上进行内存空间回收，最复杂——无法提前预知某个object是否已经变得无用。

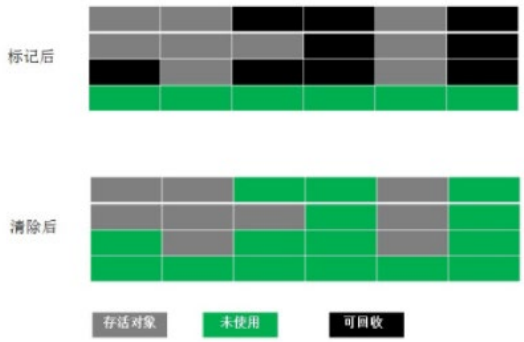
【动态垃圾回收 相关概念】（转自 长安蒹葭的博客）

- GC（Garbage Collection）：识别垃圾并释放其占用的内存
  - 垃圾回收器根据对象的“活性”(从root的可达性)来决定是否回收该对象的内存，”死“的对象是需要回收的垃圾
- Root
  - 根集合由root对象和局部对象构成
  - root对象：Class（不能被回收）、Thread、Java方法/接口的本地变量或参数、全局接口引用等
- 可达/不可达对象（Reachable/Unreachable）：free模式
  - 从根可以直接或间接到达的对象为可达的，否则为不可达的
  - 从根开始，不断将指向的对象加入活动集，剩下的是垃圾
- 活动/死亡对象（Live/dead）：
  - 在stack和free的结合模式下，对象的引用被视为有向图，可以从根访问的对象为活动对象，否则为死亡对象。

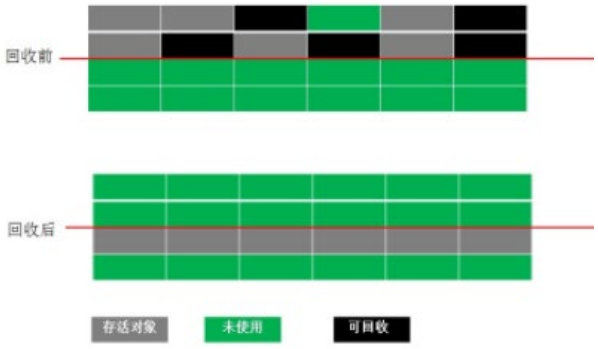
【**GC**的四种算法】

- 引用计数
  - 基本思想：为每个object存储一个计数RC，当有其他 reference指向它时，RC++；当其他reference与其断开时，RC--；如 果RC==0，则回收它。
  - 优点：简单、计算代价分散，“幽灵时间”短 为0
  - 缺点：不全面（容易漏掉循环引用的对象）、并发支 持较弱、占用额外内存空间、等
- Mark-Sweep（标记-清除）算法
  - 基本思想：为每个object设定状态位(live/dead)并记录，即mark阶段；将标记为dead的对象进行清理，即sweep可阶段。
  - 优点：可以处理循环调用，指针操作无开销，对象不变

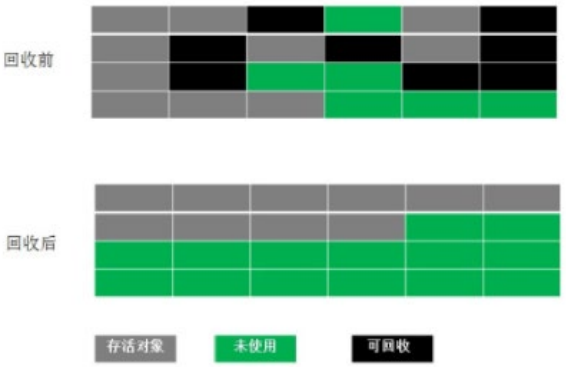
- 缺点：复杂度为O(heap),高　堆的占用比高时影响性能，容易造成碎片，需要找到root



- Copying（复制）算法
  - 基本思想：为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。
  - 优势：运行高效、不易产生内存碎片
  - 缺点：复制花费大量的时间，牺牲内存空间



- Mark-Compact（标记-整理）算法
  - 基本思想：为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。



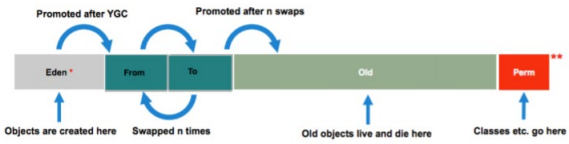
JVM中的GC

关于该部分的内容 请参考 **Java 内存区域和GC机制**

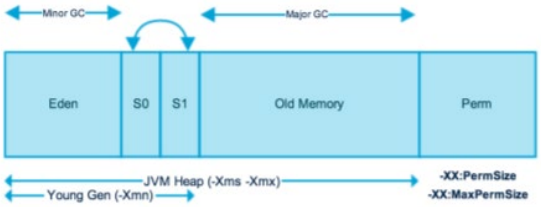
Java GC将堆分为不同的区域，各区域采用不同的GC策略，以提高GC的效率

（使用“-verbose:gc”在控制台或日志文件中输出JVM进行GC的全过程）

- Java内存分配和回收的机制概括的说，就是：分代分配，分代回收。
- 对象将根据存活的时间被分为：年轻代（Young Generation）、年老代（Old Generation）、永久代（Permanent Generation，也就是方法区）



- 年轻代：
  - 对象被创建时，内存的分配首先发生在年轻代（大对象可以直接 被创建在年老代）
  - 大部分的对象在创建后很快就不再使用，因此很快变得不可达，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消 亡的）
  - 为减少GC代价，使用**copying**算法
  - 具体过程
    - 绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；
    - 当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；
    - 此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；
    - 当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。
    - 当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。
- 年老代：
  - 对象如果在年轻代存活了足够长的时间而没有被清理掉，则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。
  - 使用Mark-Sweep或Mark-Compact算法；
  - Minor GC和full GC独立进行，减小代价；
  - 当perm generation满了之后，无法存储更多的元数据，也启动full GC。

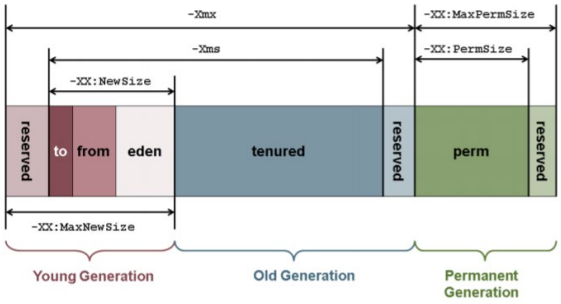


GVM GC性能调优

- 尽可能减少GC时间，一般不超过程序执行时间的5%
- 一旦初始分配给程序的内存满了，就抛出内存溢出异常，

```
"Exception in thread java.lang.OutOfMemoryError:
                Java heap space".
```

- 在启动程序时，可为其配置内存分配的具体大小



- 堆的大小决定着VM将会以何种频度进行GC、每次GC的时间多长。
  - 这两个指标具体取值多少为“优”，需要针对特定应用进行分析。
  - 较大的heap会导致较少发生GC，但每次GC时间很长
  - 如果根据程序需要来设置heap大小，则需要频繁GC，但每次GC的时间较短
- 设定堆的大小的具体方法
  - Xmx/-Xms：指定年轻代和老年代空间的初始值和最大值；Xms小于Xmx时，年轻代和老年代所消耗的空间量可以根据应用程序的需求增长或收缩；Java堆的增长不会比Xms大，也不会比Xmx小
  - XX: NewSize=<n>[g|m|k]：年轻代空间的初始和最小尺寸，<n>是大小，[g | m | k]指示大小是否应解释为千兆字节，兆字节或千字节
  - XX: MaxNewSize=<n>[g|m|k]：年轻代空间的最大值
  - Xmn<n>[g|m|k]：将年轻代的初始值、最小值、最大值设为同一值
- GC模式选择
  - 增长或收缩年轻代或老年代的空间时需要Full GC
  - Full GC可能会降低吞吐量并导致超出期望的延迟
  - 串行收集器（-XX:+UseSerialGC）：使用单个线程执行所有垃圾收集工作
  - 并行收集器（-XX:+UseParallelGC）：并行执行Minor GC，显著减少垃圾收集开销
  - 并发低暂停收集器（-XX:+UseConcMarkSweepGC）：收集持久代，与执行应用程序同时执行大部分收集，在收集期间会暂停一小段时间
  - 增量低暂停收集器（-XX:+UseTrainGC）：收集每个Minor的部分老年代，并尽量减少Major的大停顿
  - -verbose:gc：打印GC信息