guyan0319 891

GO GC 垃圾回收机制

golang gc 垃圾回收 发布于 2019-02-15 • 约 10 分钟

垃圾回收(Garbage Collection，简称GC)是编程语言中提供的内存管理功能。

在传统的系统级编程语言（主要指C/C++）中，程序员定义了一个变量，就是在内存中开辟了一段相应的空间来存值。由于内存是有限的，所以当程序不再需要使用某个变量的时候，就需要销毁该对象并释放其所占用的内存资源，好重新利用这段空间。在C/C++中，释放无用变量内存空间的事情需要由程序员自己来处理。就是说当程序员认为变量没用了，就手动地释放其占用的内存。但是这样显然非常繁琐，如果有所遗漏，就可能造成资源浪费甚至内存泄露。当软件系统比较复杂，变量多的时候程序员往往就忘记释放内存或者在不该释放的时候释放内存了。这对于程序开发人员是一个比较头痛的问题。

为了解决这个问题，后来开发出来的几乎所有新语言（java，python，php等等）都引入了语言层面的自动内存管理 – 也就是语言的使用者只用关注内存的申请而不必关心内存的释放，内存释放由虚拟机（virtual machine）或运行时（runtime）来自动进行管理。而这种对不再使用的内存资源进行自动回收的功能就被称为垃圾回收。

垃圾回收常见的方法

引用计数（reference counting）

引用计数通过在对象上增加自己被引用的次数，被其他对象引用时加1，引用自己的对象被回收时减1，引用数为0的对象即为可以被回收的对象。这种算法在内存比较紧张和实时性比较高的系统中使用的比较广泛，如ios cocoa框架，php，python等。

优点：

1、方式简单，回收速度快。

缺点：

1、需要额外的空间存放计数。

2、无法处理循环引用（如a.b=b;b.a=a这种情况）。

3、频繁更新引用计数降低了性能。

标记-清除（mark and sweep）

该方法分为两步，标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为"被引用"；标记完成后进行清除操作，对没有标记过的内存进行回收（回收同时可能伴有碎片整理操作）。这种方法解决了引用计数的不足，但是也有比较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执行，回收是系统响应能力大大降低！当然后续也出现了很多mark&sweep算法的变种（如三色标记法）优化了这个问题。

复制收集

复制收集的方式只需要对对象进行一次扫描。准备一个「新的空间」，从根开始，对对象进行扫，如果存在对这个对象的引用，就把它复制到「新空间中」。一次扫描结束之后，所有存在于「新空间」的对象就是所有的非垃圾对象。

这两种方式各有千秋，标记清除的方式节省内存但是两次扫描需要更多的时间，对于垃圾比例较小的情况占优势。复制收集更快速但是需要额外开辟一块用来复制的内存，对垃圾比例较大的情况占优势。特别的，复制收集有「局部性」的优点。

在复制收集的过程中，会按照对象被引用的顺序将对象复制到新空间中。于是，关系较近的对象被放在距离较近的内存空间的可能性会提高，这叫做局部性。局部性高的情况下，内存缓存会更有效地运作，程序的性能会提高。

对于标记清除，有一种标记-压缩算法的衍生算法：

对于压缩阶段，它的工作就是移动所有的可达对象到堆内存的同一个区域中，使他们紧凑的排列在一起，从而将所有非可达对象释放出来的空闲内存都集中在一起，通过这样的方式来达到减少内存碎片的目的。

分代收集（generation）

这种收集方式用了程序的一种特性：大部分对象会从产生开始在很短的时间内变成垃圾，而存在的很长时间的对象往往都有较长的生命周期。

根据对象的存活周期不同将内存划分为新生代和老年代，存活周期短的为新生代，存活周期长的为老年代。这样就可以根据每块内存的特点采用最适当的收集算法。

新创建的对象存放在称为 新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多）。高频对新生成的对象进行回收，称为「小回收」，低频对所有对象回收，称为「大回收」。每一次「小回收」过后，就把存活下来的对象归为老年代，「小回收」的时候，遇到老年代直接跳过。大多数分代回收算法都采用的「复制收集」方法，因为小回收中垃圾的比例较大。

这种方式存在一个问题：如果在某个新生代的对象中，存在「老生代」的对象对它的引用，它就不是垃圾了，那么怎么制止「小回收」对其回收呢？这里用到了一中叫做写屏障的方式。

程序对所有涉及修改对象内容的地方进行保护，被称为「写屏障」（Write Barrier）。写屏障不仅用于分代收集，也用于其他GC算法中。

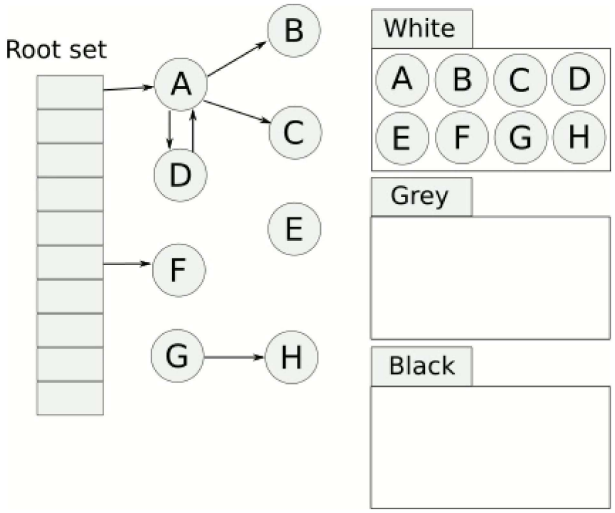
在此算法的表现是，用一个记录集来记录从新生代到老生代的引用。如果有两个对象A和B，当对A的对象内容进行修改并加入B的引用时，如果①A是「老生代」②B是「新生代」。则将这个引用加入到记录集中。「小回收」的时候，因为记录集中有对B的引用，所以B不再是垃圾。

■ 三色标记算法

三色标记算法是对标记阶段的改进，原理如下：

1. 起初所有对象都是白色。
2. 从根出发扫描所有可达对象，标记为灰色，放入待处理队列。
3. 从队列取出灰色对象，将其引用对象标记为灰色放入队列，自身标记为黑色。
4. 重复 3，直到灰色对象队列为空。此时白色对象即为垃圾，进行回收。

可视化如下。



三色标记的一个明显好处是能够让用户程序和 mark 并发的进行，具体可以参考论文：《On-the-fly garbage collection: an exercise in cooperation.》。Golang 的 GC 实现也是基于这篇论文，后面再具体说明。

GO的垃圾回收器

go语言垃圾回收总体采用的是经典的mark and sweep算法。

- v1.3以前版本 STW（Stop The World）
  golang的垃圾回收算法都非常简陋，然后其性能也广被诟病:go runtime在一定条件下（内存超过阈值或定期如2min），暂停所有任务的执行，进行mark&sweep操作，操作完成后启动所有任务的执行。在内存使用较多的场景下，go程序在进行垃圾回收时会发生非常明显的卡顿现象（Stop The World）。在对响应速度要求较高的后台服务进程中，这种延迟简直是不能忍受的！
  这个时期国内外很多在生产环境实践go语言的团队都或多或少踩过gc的坑。当时解决这个问题比较常用的方法是尽快控制自动分配内存的内存数量以减少gc负荷，同时采用手动管理内存的方法处理需要大量及高频分配内存的场景。

- v1.3 Mark STW, Sweep 并行
  1.3版本中，go runtime分离了mark和sweep操作，和以前一样，也是先暂停所有任务执行并启动mark，mark完成后马上就重新启动被暂停的任务了，而是让sweep任务和普通协程任务一样并行的和其他任务一起执行。如果运行在多核处理器上，go会试图将gc任务放到单独的核心上运行而尽量不影响业务代码的执行。go team自己的说法是减少了50%-70%的暂停时间。
- v1.5 三色标记法
  go 1.5正在实现的垃圾回收器是"非分代的、非移动的、并发的、三色的标记清除垃圾收集器"。引入了上文介绍的三色标记法，这种方法的mark操作是可以渐进执行的而不需每次都扫描整个内存空间，可以减少stop the world的时间。 由此可以看到，一路走来直到1.5版本，go的垃圾回收性能也是一直在提升，但是相对成熟的垃圾回收系统（如java jvm和javascript v8），go需要优化的路径还很长（但是相信未来一定是美好的~）。
- v1.8 混合写屏障（hybrid write barrier）
  这个版本的 GC 代码相比之前改动还是挺大的，采用一种混合的 write barrier 方式 （Yuasa-style deletion write barrier [Yuasa '90] 和 Dijkstra-style insertion write barrier [Dijkstra '78]）来避免 堆栈重新扫描。
  混合屏障的优势在于它允许堆栈扫描永久地使堆栈变黑（没有STW并且没有写入堆栈的障碍），这完全消除了堆栈重新扫描的需要，从而消除了对堆栈屏障的需求。重新扫描列表。特别是堆栈障碍在整个运行时引入了显着的复杂性，并且干扰了来自外部工具（如GDB和基于内核的分析器）的堆栈遍历。
  此外，与Dijkstra风格的写屏障一样，混合屏障不需要读屏障，因此指针读取是常规的内存读取; 它确保了进步，因为物体单调地从白色到灰色再到黑色。
  混合屏障的缺点很小。它可能会导致更多的浮动垃圾，因为它会在标记阶段的任何时刻保留从根（堆栈除外）可到达的所有内容。然而，在实践中，当前的Dijkstra障碍可能几乎保留不变。混合屏障还禁止某些优化：特别是，如果Go编译器可以静态地显示指针是nil，则Go编译器当前省略写屏障，但是在这种情况下混合屏障需要写屏障。这可能会略微增加二进制大小。

小结:

通过go team多年对gc的不断改进和忧化，GC的卡顿问题在1.8 版本基本上可以做到 1 毫秒以下的 GC 级别。 实际上，gc低延迟是有代价的，其中最大的是吞吐量的下降。由于需要实现并行处理，线程间同步和多余的数据生成复制都会占用实际逻辑业务代码运行的时间。GHC的全局停止GC对于实现高吞吐量来说是十分合适的，而Go则更擅长与低延迟。
并行GC的第二个代价是不可预测的堆空间扩大。程序在GC的运行期间仍能不断分配任意大小的堆空间，因此我们需要在到达最大的堆空间之前实行一次GC，但是过早实行GC会造成不必要的GC扫描，这也是需要衡量利弊的。因此在使用Go时，需要自行保证程序有足够的内存空间。

垃圾收集是一个难题，没有所谓十全十美的方案，通常是为了适应应用场景做出的一种取舍。

相信GO未来会更好。

参考:

https://github.com/golang/pro...

http://legendtkl.com/2017/04/...

https://blog.twitch.tv/gos-ma...

https://blog.plan99.net/moder...

links

- 目录

guyan0319

891

关注作者

4 条评论

得票 •时间

撰写评论 ...

提交评论

l1nkkk： 写的好。

• 回复 • 2019-09-07

穷得和蚂蚁抢食吃： 现在都1.13了 代码完全不同了 引用的文章 还是1.7的 醉了

• 回复 • 2019-11-17

百里： 三色标记法, 什么时候变成黑色呢?

• 回复 • 2月17日

guyan0319： gc扫描内存对象的时候，会把该内存对象变为黑色

• 回复 • 2月17日

推荐阅读

JVM垃圾回收机制

垃圾回收需要考虑的三件事：为什么要学习GC呢？当需要排查各种内存溢出。内存泄漏的问题时，当垃圾收集成为系统达到更高...

scu酱油仔 • 阅读 8

V8垃圾回收机制

原文地址本文是深入浅出nodejs的部分学习笔记在node中javascript能使用的内存是有限制的，64位系统下约为1.4GB，32位系统下...

easyhappy • 阅读 12

简述JavaScript的垃圾回收机制

不管是高级语言，还是低级语言。内存的管理都是：前两步，大家都没有太大异议。关键是释放内存这一步，各种语言都有自己的...

谷雨 • 阅读 99

php7 垃圾回收机制详解

在php中的变量占用的空间，是不需要我们手动回收的。内核帮我们处理了这一部分的工作。相比C，这大大方便了我们的操作。...

我想做超人 • 阅读 560

GC垃圾回收机制： 浅析与理解

对垃圾回收进行分析前，我们先来了解一些基本概念内存管理：内存管理对于编程语言至关重要。汇编允许你操作所有东西，或者...

言己 • 阅读 21

golang垃圾回收

设计垃圾回收算法时，折中无处不在。较大的吞吐量和较短的最大暂停时间往往不可兼得。golang采用三色法作为GC的计算方式,...

pengj • 阅读 371

【GC】垃圾回收算法

程序里,对于堆(Heap)的内存空间管理,是需要额外分配以及释放。在一些语言中,会有对应的算法,计时器进行自动分配及释放。核心:...

wayneli • 阅读 11

PHP垃圾回收机制

PHP是一种弱类型的脚本语言，弱类型不表示PHP变量没有类型的区别，PHP变量有8种原始类型：四种标量类型：两种复合类型：...

阿杰 • 阅读 88

golang开发笔记

用户专栏

微不足道的进步也是进步，学习是一个不断积累的过程。

405 人关注　**33** 篇文章

关注专栏　专栏主页

/