原文  <http://blog.pandocloud.com/?p=616>

 http://www.open-open.com/lib/view/open1435846881544.html

在实际使用go语言的过程中，碰到了一些看似奇怪的内存占用现象，于是决定对go语言的垃圾回收模型进行一些研究。本文对研究的结果进行一下总结。

什么是垃圾回收？

曾几何时，内存管理是程序员开发应用的一大难题。传统的系统级编程语言（主要指C/C++）中，程序员必须对内存小心的进行管理操作，控制内存的申请及释放。稍有不慎，就可能产生 [内存泄露](https://zh.wikipedia.org/wiki/%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F) 问题，



这种问题不易发现并且难以定位，一直成为困扰开发者的噩梦。如何解决这个头疼的问题呢？过去一般采用两种办法：

1. 内存泄露检测工具。这种工具的原理一般是静态代码扫描，通过扫描程序检测可能出现内存泄露的代码段。然而检测工具难免有疏漏和不足，只能起到辅助作用。
2. 智能指针。这是c++中引入的自动内存管理方法，通过拥有自动内存管理功能的指针对象来引用对象，是程序员不用太关注内存的释放，而达到内存 自动释放的目的。这种方法是采用最广泛的做法，但是对程序员有一定的学习成本（并非语言层面的原生支持），而且一旦有忘记使用的场景依然无法避免内存泄 露。

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行为就被称为 **垃圾回收** 。

常见的垃圾回收方法

* 引用计数（reference counting）

这是最简单的一种垃圾回收算法，和之前提到的智能指针异曲同工。对每个对象维护一个 **引用计数** ，当引用该对象的对象被销毁或更新时被引用对象的引用计数自动减一，当被引用对象被创建或被赋值给其他对象时引用计数自动加一。当引用计数为0时则立即回收对象。

这种方法的优点是实现简单，并且内存的回收很及时。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。简单引用计数算法也有明显的缺点：

1. 频繁更新引用计数降低了性能。一种简单的解决方法就是编译器将相邻的引用计数更新操作合并到一次更新；还有一种方法是针对频繁发生的临时变量引用不进行计数，而是在引用达到0时通过扫描堆栈确认是否还有临时对象引用而决定是否释放。等等还有很多其他方法，具体可以参考 [**这里**](https://en.wikipedia.org/wiki/Reference_counting#Dealing_with_inefficiency_of_updates)。
2. 循环引用问题。当对象间发生循环引用时引用链中的对象都无法得到释放。最明显的解决办法是避免产生循环引用，如cocoa引入了strong指针和weak指针两种指针类型。或者系统检测循环引用并主动打破循环链。当然这也增加了垃圾回收的复杂度。

* 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**](https://en.wikipedia.org/wiki/Tracing_garbage_collection#Tri-color_marking)）优化了这个问题。

* 分代收集（generation）

经过大量实际观察得知，在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为 **代（generation）** 的空间。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），随着垃圾回收的重复执行，生命周期较长的对象会被 **提升（promotion）** 到老年代中。因此，新生代垃圾回收和老年代垃圾回收两种不同的垃圾回收方式应运而生，分别用于对各自空间中的对象执行垃圾回收。新生代垃圾回收的速度非常 快，比老年代快几个数量级，即使新生代垃圾回收的频率更高，执行效率也仍然比老年代垃圾回收强，这是因为大多数对象的生命周期都很短，根本无需提升到老年 代。

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。

* 1.3版本以前，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。
* 1.3版本开始go team开始对gc性能进行持续的改进和优化，每个新版本的go发布时gc改进都成为大家备受关注的要点。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。
* 1.4版本（当前最新稳定版本）对gc的性能改动并不多。1.4版本中runtime很多代码取代了原生c语言实现而采用了go语言实现，对gc带来的一大改变是可以是实现精确的gc。c语言实现在gc时无法获取到内存的对象信息，因此无法准确区分普通变量和指针，只能将普通变量当做指针，如 果碰巧这个普通变量指向的空间有其他对象，那这个对象就不会被回收。而go语言实现是完全知道对象的类型信息，在标记时只会遍历指针指向的对象，这样就避 免了C实现时的堆内存浪费（解决约10-30%）。
* 1.5版本go team对gc又进行了比较大的改进（1.4中已经埋下伏笔如write barrier的引入）,官方的主要目标是减少延迟。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”。分代算法上文已经提及，是一种比较好的垃圾回收管理策略，然 1.5版本中并未考虑实现；我猜测的原因是步子不能迈太大，得逐步改进，go官方也表示会在1.6版本的gc优化中考虑。同时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~）。

实践经验

团队在实践go语言时同样碰到最多和最棘手的问题也是内存问题（其中gc为主），这里把遇到的问题和经验总结下，欢迎大家一起交流探讨。

1. **go程序内存占用大的问题** 。这个问题在我们对后台服务进行压力测试时发现，我们模拟大量的用户请求访问后台服务，这时各服务模块能观察到明显的内存占用上升。但是当停止压测时，内存占用并未发生明显的下降。花了很长时间定位问题，使用gprof等各种方法，依然没有发现原因。最后发现原来这时正常的…主要的原因有两个，一是go的垃圾回收有个触发阈值，这个阈值会随着每次内存使用变大而逐渐增大（如初始阈值是10MB则下一次就是20MB，再下一次就成为了40MB…），如果长时 间没有触发gc go会主动触发一次（2min）。高峰时内存使用量上去后，除非持续申请内存，靠阈值触发gc已经基本不可能，而是要等最多2min主动gc开始才能触发 gc。第二个原因是go语言在向系统交还内存时只是告诉系统这些内存不需要使用了，可以回收；同时操作系统会采取“拖延症”策略，并不是立即回收，而是等到系统内存紧张时才会开始回收这样该程序又重新申请内存时就可以获得极快的分配速度。
2. **gc时间长的问题。** 对于对用户响应事件有要求的后端程序，golang gc时的stop the world兼职是噩梦。根据上文的介绍，1.5版本的go在完成上述改进后应该gc性能会提升不少，但是所有的垃圾回收型语言都难免在gc时面临性能下 降，对此我们对于应该尽量避免频繁创建临时堆对象（如&abc{}, new, make等）以减少垃圾收集时的扫描时间，对于需要频繁使用的临时对象考虑直接通过数组缓存进行重用；很多人采用cgo的方法自己管理内存而绕开垃圾收集，这种方法除非迫不得已个人是不推荐的（容易造成不可预知的问题），当然迫不得已的情况下还是可以考虑的，这招带来的效果还是很明显的~
3. **goroutine泄露的问题。** 我们的一个服务需要处理很多长连接请求，实现时，对于每个长连接请求各开了一个读取和写入协程，全部采用endless **for** loop不停地处理收发数据。当连接被远端关闭后，如果不对这两个协程做处理，他们依然会一直运行，并且占用的channel也不会被释放…这里就必须十分注意，在不使用协程后一定要把他依赖的channel close并通过再协程中判断channel是否关闭以保证其退出。

[代际垃圾回收器的一种快速Write Barrier算法](http://blog.csdn.net/jollyjumper/article/details/41824729)

http://blog.csdn.net/jollyjumper/article/details/41824729