# Go语言机制

本系列文章总共包括 4 篇，主要帮助大家理解 Go 语言中一些语言机制和其背后的设计原则，包括指针、栈、堆、逃逸分析和值传递/地址传递。这一篇是本系列的第一篇，主要介绍栈和指针。

以下是本系列文章的索引：

1. **[Go 语言机制之栈与指针](https://studygolang.com/articles/12443)**
2. **[Go 语言机制之逃逸分析](https://studygolang.com/articles/12444)**
3. **[Go 语言机制之内存剖析](https://studygolang.com/articles/12445)**
4. **[Go 语言机制之数据和语法的设计哲学](https://studygolang.com/articles/12487)**

## 栈和指针

### 简介

我不打算说指针的好话，它确实很难理解。如果应用不当，会产生恼人的 bug，甚至会导致性能问题。当写并发和多线程程序时更是如此。所以许多语言试着用其它方法让编程人员避免指针的使用。但如果你是在用 Go 语言的话，你就不得不使用它们。如果不能很好的理解指针，是很难写出干净、简单并且高效的代码的。

### 帧边界（Frame Boundaries）

帧边界为每个函数提供了它自己独有的内存空间，函数就是在这个内存空间内执行的。帧边界除了可以让函数在自己的上下文环境中运行外还提供一些流程控制功能。函数可以通过帧边界指针直接访问自己帧边界中的内存，但如果想要访问自己帧边界外的内存，就需要用间接访问来实现了。要实现间接访问，被访问的内存必须和函数共享，要想弄清楚共享是怎么实现的，我们就得先了解一下由这些帧边界建立起来的内存结构以及其中的一些限制。

当一个函数被调用时，会在两个相关的帧边界间进行上下文切换。从调用函数切换到被调用函数，如果函数调用时需要传递参数，那么这些参数值也要传递到被调用函数的帧边界中。**Go 语言中帧边界间的数据传递是按值传递的。**

按值传递的好处是可读性好，拷贝并被函数接收到的值就是在函数调用时传入的值 。这就是为什么我把按值传递叫做 WYSIWYG（what you see is what you get 的缩写）。如果发生上下文环境转换时参数是按值传递的，我们就可以很清楚的知道这个函数调用会怎样影响程序的执行

让我们看一下下面这个小程序，主程序用按值传递的方式调用了一个函数：

package main

func main() {

// Declare variable of type int with a value of 10.

count := 10

// Display the "value of" and "address of" count.

println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

// Pass the "value of" the count.

increment(count)

println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")}

//go:noinline

func increment(inc int) {

// Increment the "value of" inc.

inc++

println("inc:\tValue Of[", inc, "]\tAddr Of[", &inc, "]")

}

程序启动后，语言运行环境会创建 main goroutine 来执行包含在函数 main 内的所有初始化代码。goroutine 是被放置在操作系统线程上的可执行序列，在 Go 语言的1.8版本中，为每一个 goroutine 分配了 2048 byte 的连续内存作为它的栈空间。这个初始化的内存大小几年来一直在变化，而且未来很有可能继续变化。

栈在 Go 语言中是非常重要的，因为它为分配给每个函数的帧边界提供了物理内存空间。main goroutine 在执行表 1 中的代码时，goroutine 的栈看起来像下面这个样子（在一个比较高的语言层次）

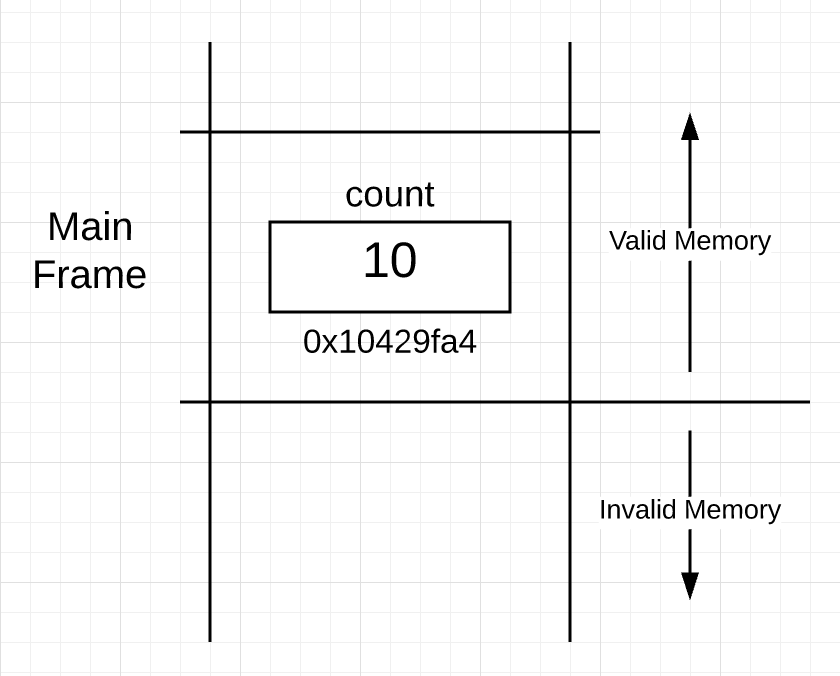


图 1

在图 1 中可以看到，一部分栈空间被框了起来，作为函数 main 的可用空间，这块栈区域叫做**「栈帧」**,正是它界定了函数 main 在栈上的边界。这块栈空间是在函数被调用后，随着一些初始化代码的执行一并被创建的。可以看到变量 count 被放置到了函数 main 的栈帧中地址为 0x10429fa4 的地方。

在图 1 中也可以发现另外一点，就是在活动栈帧之下的栈空间是不可用的，只在活动栈帧以及它之上的栈空间是可用的。这个可用栈空间与不可用栈空间的边界我们需要明确一下。

### 地址

变量名是为了标识一块内存，使代码更具可读性而存在的。一个好的变量名可以让编程人员清楚的知道它代表了什么。如果你已经有了一个变量，那在内存中就有一个值与它对应；反之，如果在内存中有一个值，就必须有一个与之对应的变量，通过这个变量来访问这个内存值。在第 9 行，主函数调用了内置函数 println 来显示变量 count 的值和地址。

09 println("count:\tValue Of[", count, "]\tAddr Of[", &count, "]")

用 & 操作符来获取变量的地址并不新鲜，许多其它语言也同样用这个操作符来获取变量地址。如果你在 32 位机器上运行这段代码（例如 playgournd ），第 9 行的输出应该像下面这样。

count: Value Of[ 10 ] Addr Of[ 0x10429fa4 ]

### 函数调用

接下来第 12 行，函数 main 调用了函数 increment：

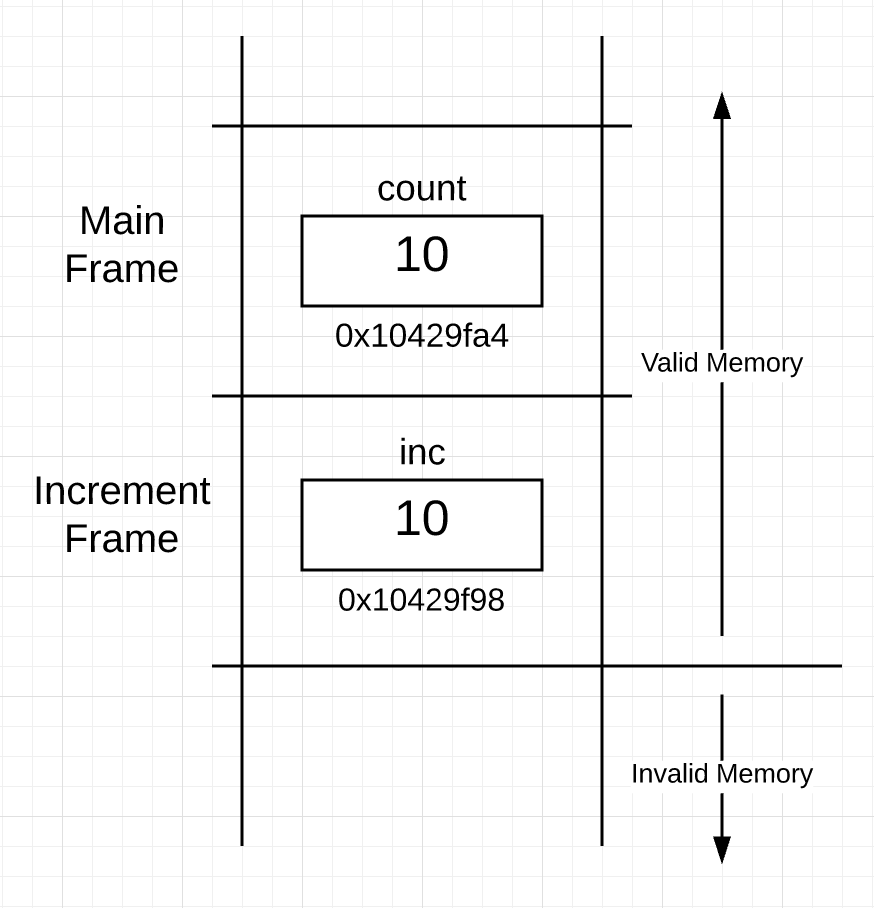
12 increment(count)

函数调用意味着 goroutine 需要在栈空间中创建一个新的栈帧。然而，这里并没有这么简单。要成功的调用一个函数，需要将数据在上下文转换过程中跨栈帧边界传递到新建的栈帧中。特别的，对于 integer 值，在调用过程中需要拷贝并传递过去，在第 18 行对函数 increment 的声明语句中可以看到这一点：

18 func increment(inc int) {

如果再看一下第 12 行对函数 increment 的调用，可以看到传递的正是变量 count 的值。这个值经过拷贝、传递并最终放置到了函数 increment 的栈帧中。因为函数 increment 只能直接访问自己栈帧里的内存，所以它用变量 inc 来接收、存储和访问从变量 count 传递过来的值。

在函数 increment 刚刚要开始执行的时候，goroutine 的栈结构看起来像下面这个样子（从一个比较高的语言层次）。



可以看到，现在在栈里有两个栈帧, 一个是函数 main 的，它下面的是函数 increment 的。在函数 increment 栈帧里，有一个变量 inc，它的值是当函数调用时从外面拷贝并传递过来的 10，它的地址是 0x10429f98，因为栈帧是从上往下使用栈空间的，所以它的地址比上面的小，不过这只是一个实现细节，并不保证所有实现都这样。重要的是 goroutine 把函数 main 的栈帧中的变量 count 的值拷贝并传递给了函数 increment 的栈帧中的变量 inc。

函数 increment 剩下的代码显示了变量 inc 的值和地址：

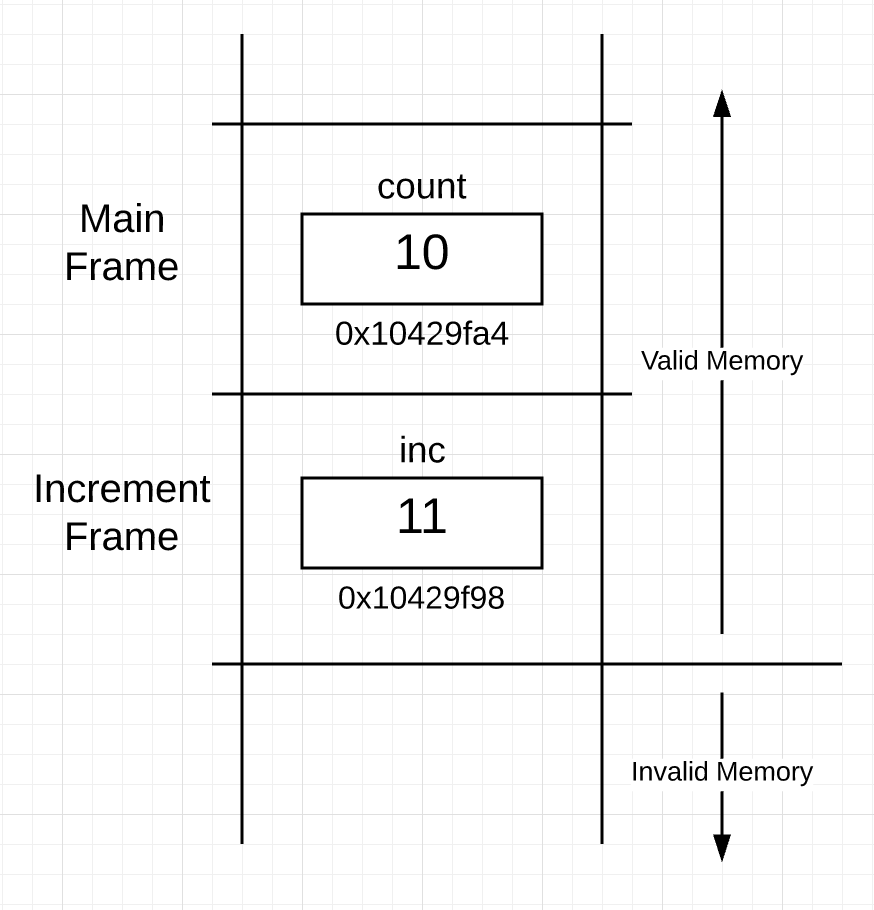
21 inc++

22 println("inc:\tValue Of[", inc, "]\tAddr Of[", &inc, "]")

在 playground 平台上，第 22 行的输出看起来像这样：

inc: Value Of[ 11 ] Addr Of[ 0x10429f98 ]

当执行完了这些代码以后，栈结构变成下面这个样子



执行完第 21 行和第 22 行以后，函数 increment 返回，控制权重新回到了函数 main 中，然后函数 main 再一次显示了变量 count 的值和地址：

14 println("count:\tValue Of[",count, "]\tAddr Of[", &count, "]")

在 playgournd 平台上，程序全部的输出如下：

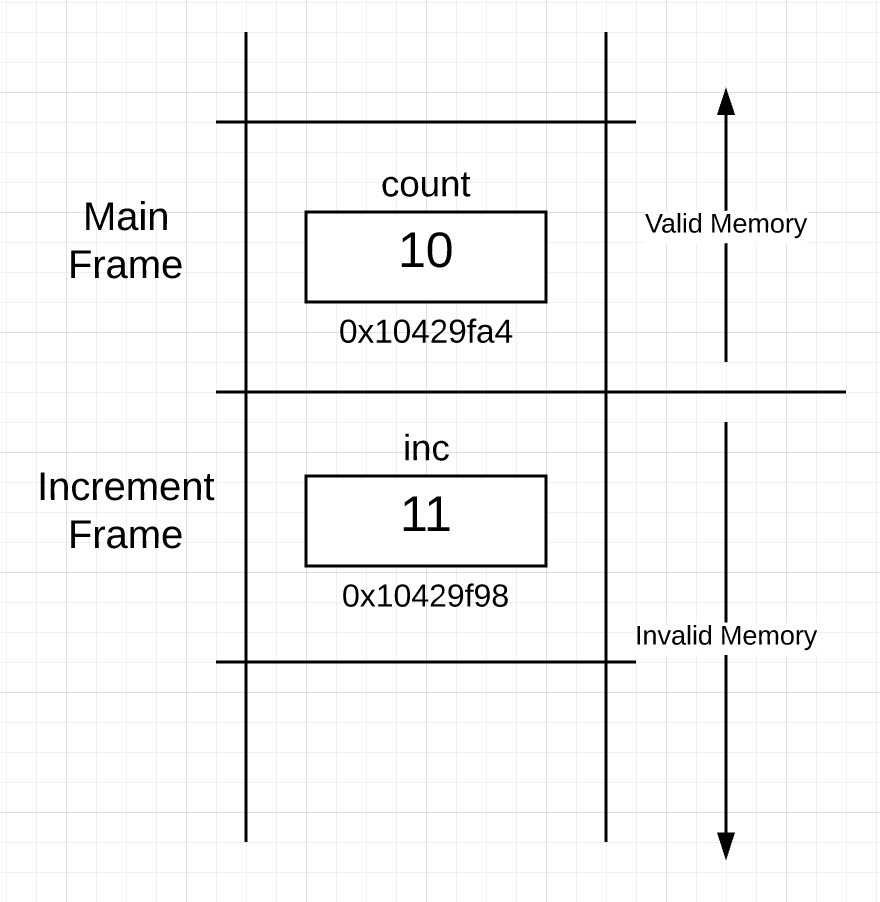
count: Value Of[ 10 ] Addr Of[ 0x10429fa4 ]

inc: Value Of[ 11 ] Addr Of[ 0x10429f98 ]

count: Value Of[ 10 ] Addr Of[ 0x10429fa4 ]

### 函数返回

当函数返回，控制权回到调用函数后，栈结构发生了什么变化呢？答案是什么也没有。下面就是当函数 increment 返回后，栈结构的样子：



除了为函数 increment 创建的栈帧现在变为不可用外，其他和图 3 一模一样。这是因为函数 main 的栈帧变成了活动栈帧。对函数 incrment 的栈帧没有做任何处理。

函数调用完成后，没有必要立即清理被调用函数的栈帧空间，这样做只会浪费时间，因为你不知道那块内存之后是否会被再次用到。所以相应内存就原封不动的留在那里。只有当发生了函数调用，这块内存被再次用到时，才会对它进行清理。清理过程是通过拷贝过来的值在这个栈帧中的初始化完成的，因为所有的变量至少会被初始化为相应类型的零值，这就保证了发生函数调用时，栈空间一定会被合理的清理。

### 值的共享

但是如果我们想在函数 increment 中直接操作存在于函数 main 的栈帧中的变量 count，应该怎么办呢？这时候我们就要用到指针了。指针存在在目的就是为了和一个函数共享变量，从而让这个函数可以对这个共享变量进行读写，即使这个变量没有直接放置在这个函数的栈帧中。

如果当你用指针时，一下子想到的不是「共享」，那就得看看是不是真的有必要使用指针了。当我们学习指针的内容时，有一点很重要，就是要用一个明确的单词而不是操作符或者语法来对待指针。所以请记住，用指针是为了共享，在阅读代码的时候也应该把 & 操作符当做共享来看。

#### 指针类型

对每个已经声明的类型，不管是语言自己定义的还是用户定义的，都有一个与之对应的指针类型，用它来进行数据共享。比如 Go 语言中有一个内置的 int 类型，所以一定有一个与 int 类型对应的叫做 \*int 的指针类型。如果你定义了一个叫做 User 的类型，那么语言会自动为你生成一个与它对应的叫做 \*User 的指针类型。

所有的指针类型有两个共同点。一、它们以 \* 开头。二、它们占用相同的内存大小（4 个字节或者 8 个字节）并且表示的是一个地址。在 32 位的系统上（比如 playground )，一个指针占用 4 个字节，在 64 位的系统上（比如你自己的电脑）占用 8 个字节。

规范一点说，指针类型被认为是一个字面类型（type literals)，也就是说它是通过对已有类型进行组合而成的。

#### 间接内存访问

看下面这段程序，它同样调用了一个函数，不过这次传递的是变量的地址。这样被调用的函数 increment 就可以和函数 main 共享变量 count 了：

package main

func main() {

// Declare variable of type int with a value of 10.

count := 10

// Display the "value of" and "address of" count.

println("count:\tValue Of[", count, "]\t\tAddr Of[", &count, "]")

// Pass the "address of" count.

increment(&count)

println("count:\tValue Of[", count, "]\t\tAddr Of[", &count, "]")}

//go:noinlinefunc increment(inc \*int) {

// Increment the "value of" count that the "pointer points to". (dereferencing)

\*inc++

println("inc:\tValue Of[", inc, "]\tAddr Of[", &inc, "]\tValue Points To[", \*inc, "]")}

同原来的程序比起来，新的程序存在 3 点不同

12 increment(&count)

在程序的第 12 行，并没有像之前一样传递变量 count 的值，而是传递的变量 count 的地址。现在我们可以说，我将要和函数 increment 共享变量 count 了，这就是 & 操作符想要表达的。

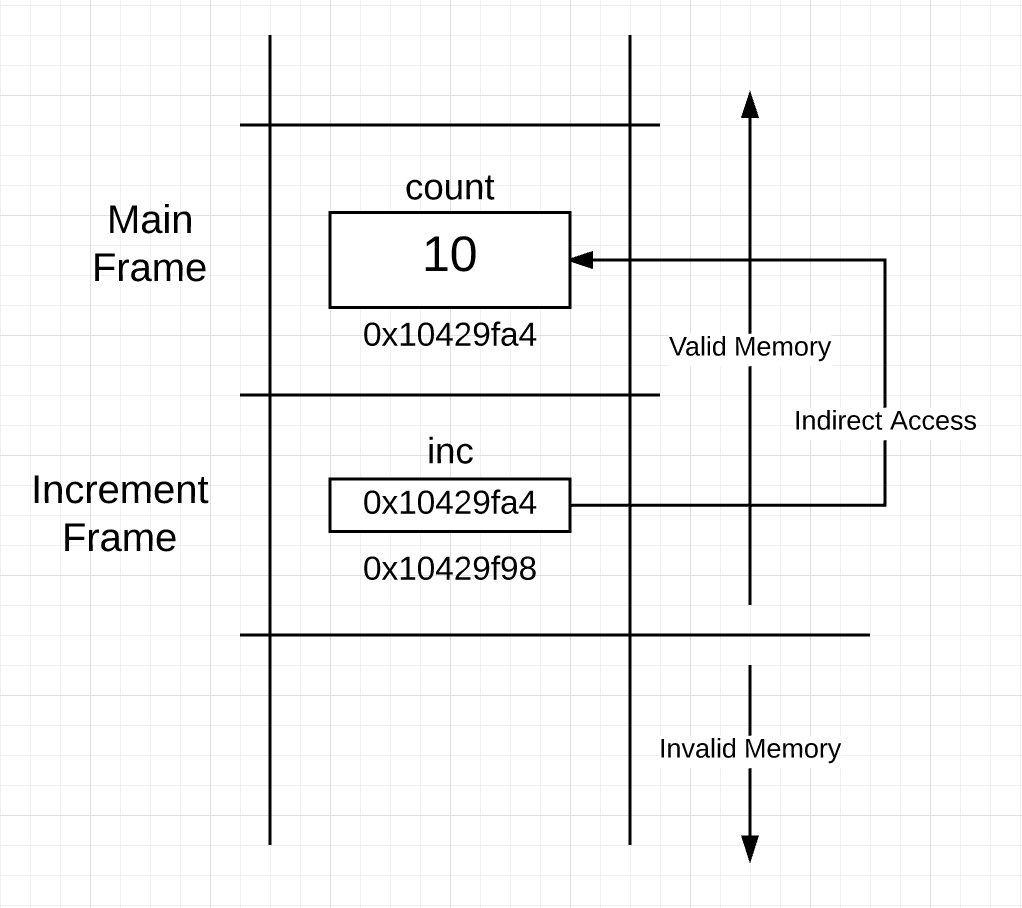
变量仍然是按值传递的，唯一不同的是，这次传递的是一个 integer 的地址。地址同样是一个值；这就是在函数调用时跨越两个帧边界被拷贝和传递的东西。

鉴于有一个值正在被拷贝和传递，在函数 inrement 中我们就需要一个变量来接收并存储这个基于地址的 integer 值，所以我们在程序的第 18 行把参数声明为了 \*int 类型。

18 func increment(inc \*int) {

如果你传递的是 User 类型的地址值，这里声明的类型就应该换成 \*User，尽管所有的指针存储的都是地址值，但是传递和接收的必须是同一个类型才可以，这个是关键。我们之所以要共享一个变量，是因为在函数内我们要对那个变量进行读写操作，而我们只有知道了这个类型的具体信息后才可以这样做。编译器会保证传递的是同一个指针类型的值。

下面是调用了函数 increment 后，栈结构的样子。



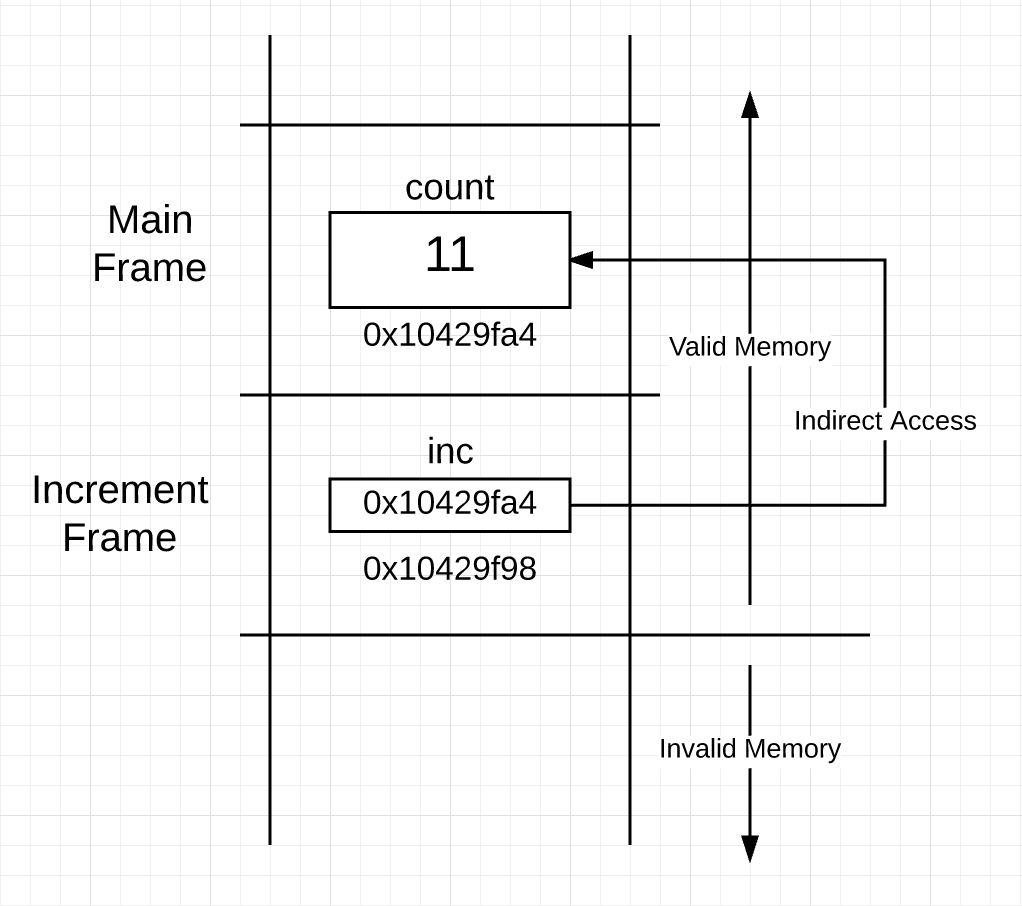
在图 5 中我们可以看到，当把一个地址按值进行传递后，栈结构会变成什么样子。函数 increment 的栈帧中的指针变量 inc 指向了存在于函数 main 的栈帧中的变量 count。

通过这个指针变量，函数就可以以间接方式读写存在于函数 main 的栈帧中的变量 count 了。

21 \*inc++

这个时候，\* 被用作一个操作符和指针变量一起使用，把 \* 用作操作符，意思是说要得到指针变量所指向的内容，在这里也就是函数 main 中的 count 变量。指针变量允许在使用它的栈帧中间接访问此栈帧之外的内存空间。有时候我们把这种间接访问叫做**指针的解引用**。在函数 increment 中仍然需要一个可以直接访问的本地指针变量来执行间接访问，在这里就是变量 inc。

当执行了第 21 行后，栈结构的变成下面这个样子。



下面是程序的全部输出：

count: Value Of[ 10 ] Addr Of[ 0x10429fa4 ]

inc: Value Of[ 0x10429fa4 ] Addr Of[ 0x10429f98 ] Value Points To[ 11 ]

count: Value Of[ 11 ] Addr Of[ 0x10429fa4 ]

可以看到，变量 inc 的值正是变量 count 的地址，就是这一个联系才使得访问本栈帧外的内存成为可能。一旦函数 increment 通过指针变量执行了写操作，当控制返回到函数 main 后，修改就会反应到对应的共享变量中。

### 指针型变量并不特别

指针类型和其它类型一样，一点也不特殊。它们有一块分配的内存并存放了一个值，抛开它指向的类型，指针类型总是占用同样的大小并且有相同的表示。唯一可能让我们感到困惑的是字符 \*，在函数 increment 内部，它被用作操作符，在函数声明时用来声明指针变量。如果你可以分清指针声明时和指针的解引用操作时的区别，应该就没那么困惑了。

### 总结

这篇文章讨论了设计指针背后的目的，以及在 Go 语言中栈和指针是怎样工作的。这是理解 Go 语言的语言机制、设计哲学的第一步，也对写出一致的、可读性好的代码有一定的指导作用。

下面来总结一下我们学到了什么：

1. 帧边界为每个函数提供了独立的内存空间，函数就是在自己的帧边界内执行的
2. 当调用函数时，上下文环境会在两个帧边界间切换
3. 按值传递的优点是可读性好
4. 栈是非常重要的，因为它为分配给每个函数的帧边界提供了可访问的物理内存空间
5. 在活动栈帧以下的栈空间是不可用的，只有活动栈帧和它之上的栈空间是可用的
6. 函数调用意味着 goroutine 需要在栈上为函数创建一个新的栈帧
7. 只有当发生了函数调用 ，栈区块被分配的栈帧占用后，相应栈空间才会被初始化
8. 使用指针是为了和被调用函数共享变量，使被调用函数可以用间接方式访问自己栈帧之外的变量
9. 每一个类型，不管是语言内置的还是用户定义的，都有一个与之对应的指针类型
10. 使用指针变量的函数，可以通过它间接访问函数栈帧之外的内存
11. 指针变量和其它变量一样，并不特殊，同样是有一块内存，在其中存放值而已

## 逃逸分析

### 介绍（Introduction）

在四部分系列的第一部分，我用一个将值共享给 goroutine 栈的例子介绍了指针结构的基础。而我没有说的是值存在栈之上的情况。为了理解这个，你需要学习值存储的另外一个位置：**堆**。有这个基础，就可以开始学习逃逸分析。

**逃逸分析**是编译器用来决定你的程序中值的位置的过程。特别地，编译器执行静态代码分析，以确定一个构造体的实例化值是否会逃逸到堆。在 Go 语言中，你没有可用的关键字或者函数，能够直接让编译器做这个决定。只能够通过你写代码的方式来作出这个决定。

### 堆（Heaps）

堆是内存的第二区域，除了栈之外，用来存储值的地方。堆无法像栈一样能自清理，所以使用这部分内存会造成很大的开销（相比于使用栈）。重要的是，开销跟 GC（垃圾收集），即被牵扯进来保证这部分区域干净的程序，有很大的关系。当垃圾收集程序运行时，它会占用你的可用 CPU 容量的 25%。更有甚者，它会造成微秒级的 “stop the world” 的延时。拥有 GC 的好处是你可以不再关注堆内存的管理，这部分很复杂，是历史上容易出错的地方。

在 Go 中，会将一部分值分配到堆上。这些分配给 GC 带来了压力，因为堆上没有被指针索引的值都需要被删除。越多需要被检查和删除的值，会给每次运行 GC 时带来越多的工作。所以，分配算法不断地工作，以平衡堆的大小和它运行的速度。

### 共享栈（Sharing Stacks）

在 Go 语言中，不允许 goroutine 中的指针指向另外一个 goroutine 的栈。这是因为当栈增长或者收缩时，goroutine 中的栈内存会被一块新的内存替换。如果运行时需要追踪指针指向其他的 goroutine 的栈，就会造成非常多需要管理的内存，以至于更新指向那些栈的指针将使 “stop the world” 问题更严重。

这里有一个栈被替换好几次的例子。看输出的第 2 和第 6 行。你会看到 main 函数中的栈的字符串地址值改变了两次。**<https://play.golang.org/p/pxn5u4EBSI>**

### 逃逸机制（Escape Mechanics）

任何时候，一个值被分享到函数栈帧范围之外，它都会在堆上被重新分配。这是逃逸分析算法发现这些情况和管控这一层的工作。（内存的）完整性在于确保对任何值的访问始终是准确、一致和高效的。

通过查看这个语言机制了解逃逸分析。**<https://play.golang.org/p/Y_VZxYteKO>**

package main

type user struct {

name string

email string

}

func main() {

u1 := createUserV1()

u2 := createUserV2()

println("u1", &u1, "u2", &u2)

}

//go:noinline

func createUserV1() user {

u := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

println("V1", &u)

return u

}

//go:noinline

func createUserV2() \*user {

u := user{

name: "Bill",

email: "bill@ardanlabs.com",

}

println("V2", &u)

return &u

}

我使用 go:noinline 指令，阻止在 main 函数中，编译器使用内联代码替代函数调用。内联（优化）会使函数调用消失，并使例子复杂化。我将在下一篇博文介绍内联造成的副作用。

**问：什么是内联？**

答：栈分配内存会比堆分配高效地多，那么，我们就会希望对象能尽可能被分配在栈上。在Go中，一个goroutine会有一个单独的栈，栈又会包含多个栈帧，栈帧是函数调用时在栈上为函数所分配的区域。但其实，函数调用是存在一些固定开销的，例如维护帧指针寄存器BP、栈溢出检测等。因此，对于一些代码行比较少的函数，编译器倾向于将它们在编译期展开从而消除函数调用，这种行为就是内联。

Go程序编译时，默认将进行内联优化。我们可通过-gcflags="-l"选项全局禁用内联，与一个-l禁用内联相反，如果传递两个或两个以上的-l则会打开内联，并启用更激进的内联策略。如果不想全局范围内禁止优化，则可以在函数定义时添加 //go:noinline 编译指令来阻止编译器内联函数。

在表 1 中，你可以看到创建 user 值，并返回给调用者的两个不同的函数。在函数版本 1 中，返回值。

16 func createUserV1() user {

17 u := user{

18 name: "Bill",

19 email: "bill@ardanlabs.com",

20 }

21

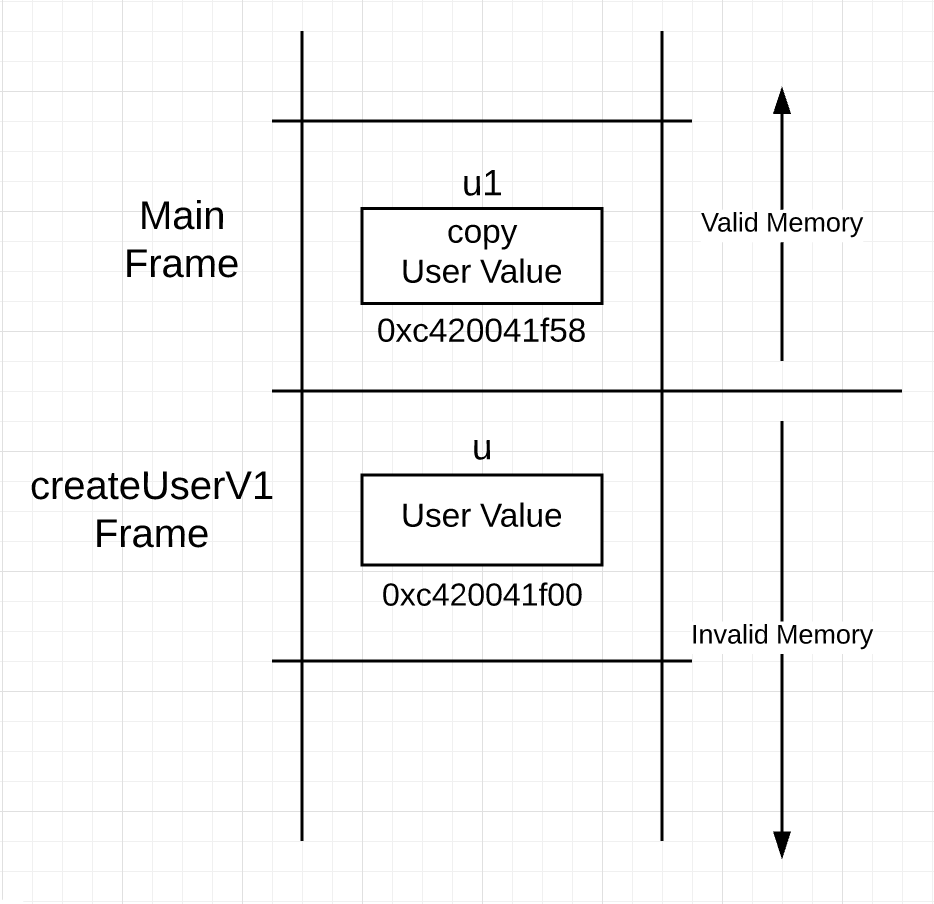
22 println("V1", &u)

23 return u

24 }

我说这个函数返回的是值是因为这个被函数创建的 user 值被拷贝并传递到调用栈上。这意味着调用函数接收到的是这个值的拷贝。

你可以看下第 17 行到 20 行 user 值被构造的过程。然后在第 23 行，user 值的副本被传递到调用栈并返回给调用者。函数返回后，栈看起来如下所示。



你可以看到图 1 中，当调用完 createUserV1 ，一个 user 值同时存在（两个函数的）栈帧中。

在函数版本 2 中，返回指针。

27 func createUserV2() \*user {

28 u := user{

29 name: "Bill",

30 email: "bill@ardanlabs.com",

31 }

32

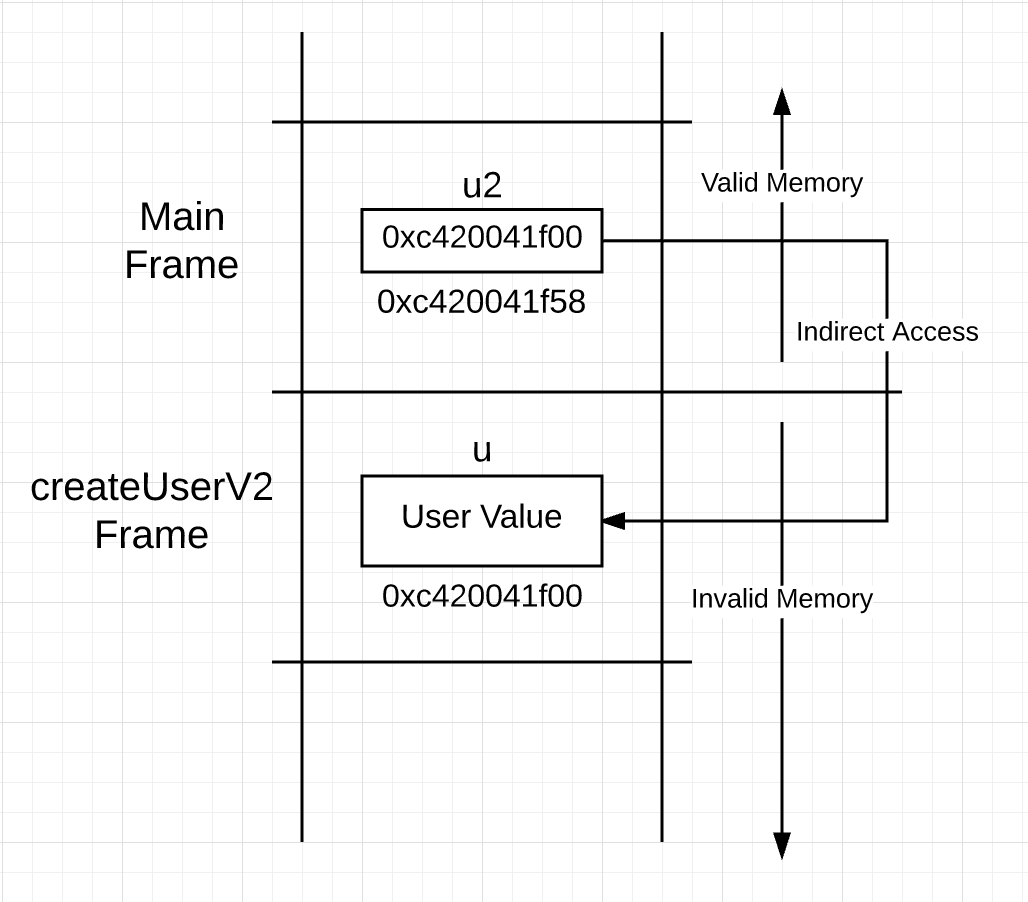
33 println("V2", &u)

34 return &u

35 }

我说这个函数返回的是指针是因为这个被函数创建的 user 值通过调用栈被共享了。这意味着调用函数接收到一个值的地址拷贝。

你可以看到在第 28 行到 31 行使用相同的字段值来构造 user 值，但在第 34 行返回时却是不同的。不是将 user 值的副本传递到调用栈，而是将 user 值的地址传递到调用栈。基于此，你也许会认为栈在调用之后是这个样子。



如果看到的图 2 真的发生的话，你将遇到一个问题。指针指向了栈下的无效地址空间。当 main 函数调用下一个函数，指向的内存将重新映射并将被重新初始化。

**这就是逃逸分析将开始保持完整性的地方。在这种情况下，编译器将检查到，在 createUserV2 的（函数）栈中构造 user 值是不安全的，因此，替代地，会在堆中构造（相应的）值。**这（个分析并处理的过程）将在第 28 行构造时立即发生。

### 可读性（Readability）

在上一篇博文中，我们知道一个函数只能直接访问它的（函数栈）空间，或者通过（函数栈空间内的）指针，通过跳转访问（函数栈空间外的）外部内存。这意味着访问逃逸到堆上的值也需要通过指针跳转。

记住 createUserV2 的代码的样子：

27 func createUserV2() \*user {

28 u := user{

29 name: "Bill",

30 email: "bill@ardanlabs.com",

31 }

32

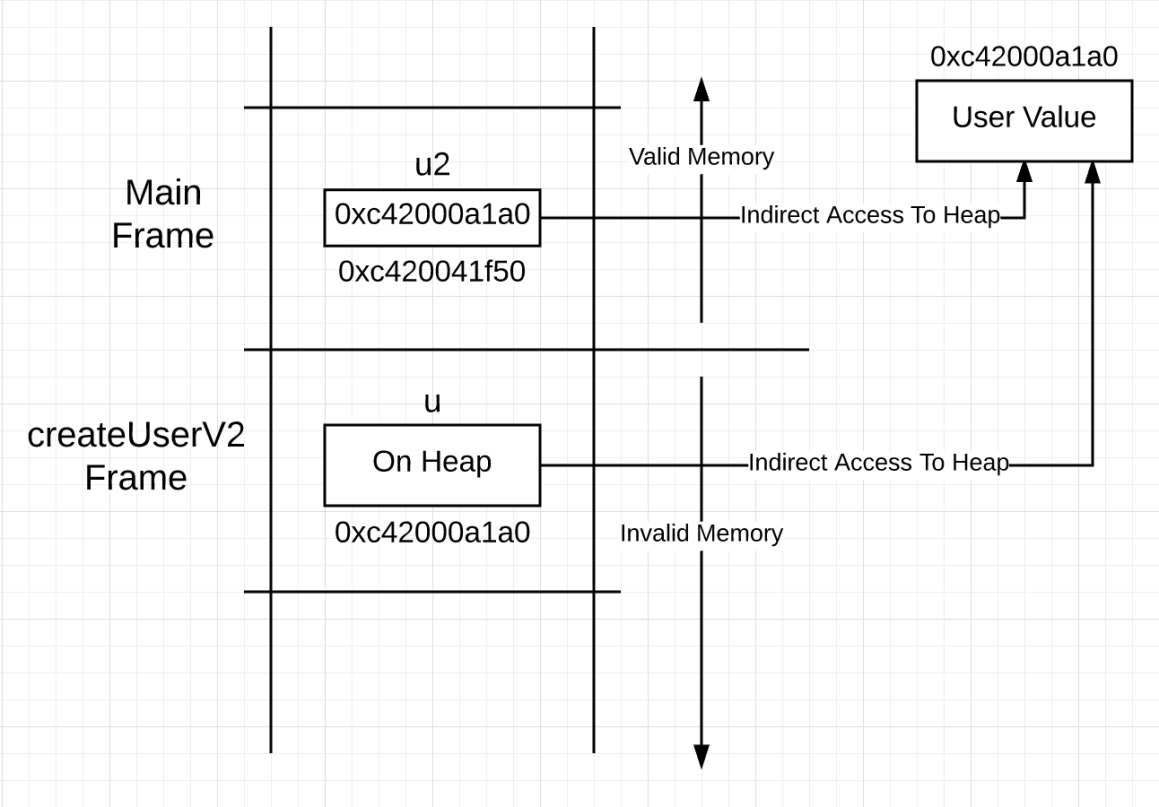
33 println("V2", &u)

34 return &u

35 }

语法隐藏了代码中真正发生的事情。第 28 行声明的变量 u 代表一个 user 类型的值。Go 代码中的类型构造不会告诉你值在内存中的位置。所以直到第 34 行返回类型时，你才知道值需要逃逸（处理）。这意味着，虽然 u 代表类型 user 的一个值，但对该值的访问必须通过指针进行。

你可以在函数调用之后，看到堆栈就像（图 3）这样。



在 createUserV2 函数栈中，变量 u 代表的值存在于堆中，而不是栈。这意味着用 u 访问值时，使用指针访问而不是直接访问。你可能想，

为什么不让 u 成为指针，毕竟访问它代表的值需要使用指针？

27 func createUserV2() \*user {

28 u := &user{

29 name: "Bill",

30 email: "bill@ardanlabs.com",

31 }

32

33 println("V2", u)

34 return u

35 }

如果你这样做，将使你的代码缺乏重要的可读性。（让我们）离开整个函数一秒，只关注 return。

34 return u

35 }

这个 return 告诉你什么了呢？它说明了返回 u 值的副本给调用栈。

然而，当你使用 & 操作符，return 又告诉你什么了呢？

34 return &u

35 }

多亏了 & 操作符，return 告诉你 u 被分享给调用者，因此，已经逃逸到堆中。记住，当你读代码的时候，指针是为了共享，& 操作符对应单词 "sharing"。这在提高可读性的时候非常有用，这（也）是你不想失去的部分。

01 var u \*user

02 err := json.Unmarshal([]byte(r), &u)

03 return u, err

为了让其可以工作，你一定要通过共享指针变量（的方式）给（函数） json.Unmarshal。json.Unmarshal 调用时会创建 user 值并将其地址赋值给指针变量。**<https://play.golang.org/p/koI8EjpeIx>**

代码解释：

01：创建一个类型为 user，值为空的指针。  
02：跟函数 json.Unmarshal 函数共享指针。  
03：返回 u 的副本给调用者。

这里并不是很好理解，user值被 json.Unmarshal 函数创建，并被共享给调用者。

如何在构造过程中使用语法语义来改变可读性？

01 var u user

02 err := json.Unmarshal([]byte(r), &u)

03 return &u, err

代码解释：

01：创建一个类型为 user，值为空的变量。  
02：跟函数 json.Unmarshal 函数共享 u。  
03：跟调用者共享 u。

这里非常好理解。第 02 行共享 user 值到调用栈中的 json.Unmarshal，在第 03 行 user 值共享给调用者。这个共享过程将会导致 user 值逃逸。

在构建一个值时，使用值语义，并利用 & 操作符的可读性来明确值是如何被共享的。

### 编译器报告（Compiler Reporting）

想查看编译器（关于逃逸分析）的决定，你可以让编译器提供一份报告。**你只需要在调用 go build 的时候，打开 -gcflags 开关，并带上 -m 选项**。

实际上总共可以使用 4 个 -m，（但）超过 2 个级别的信息就已经太多了。我将使用 2 个 -m 的级别。

$ go build -gcflags "-m -m"

./main.go:16: cannot inline createUserV1: marked go:noinline

./main.go:27: cannot inline createUserV2: marked go:noinline

./main.go:8: cannot inline main: non-leaf function

./main.go:22: createUserV1 &u does not escape

./main.go:34: &u escapes to heap

./main.go:34: from ~r0 (return) at

./main.go:34

./main.go:31: moved to heap: u

./main.go:33: createUserV2 &u does not escape

./main.go:12: main &u1 does not escape

./main.go:12: main &u2 does not escape

你可以看到编译器报告是否需要逃逸处理的决定。编译器都说了什么呢？请再看一下引用的 createUserV1 和 createUserV2 函数。

16 func createUserV1() user {

17 u := user{

18 name: "Bill",

19 email: "bill@ardanlabs.com",

20 }

21

22 println("V1", &u)

23 return u

24 }

27 func createUserV2() \*user {

28 u := user{

29 name: "Bill",

30 email: "bill@ardanlabs.com",

31 }

32

33 println("V2", &u)

34 return &u

35 }

从报告中的这一行开始。

./main.go:22: createUserV1 &u does not escape

这是说在函数 createUserV1 调用 println 不会造成 user 值逃逸到堆。这是必须检查的，因为它将会跟函数 println 共享（u）。

接下来看报告中的这几行。

./main.go:34: &u escapes to heap

./main.go:34: from ~r0 (return) at

./main.go:34

./main.go:31: moved to heap: u

./main.go:33: createUserV2 &u does not escape

这几行是说，类型为 user，并在第 31 行被赋值的 u 的值，因为第 34 行的 return 逃逸。最后一行是说，跟之前一样，在 33 行调用 println 不会造成 user 值逃逸。

阅读这些报告可能让人感到困惑，（编译器）会根据所讨论的变量的类型是基于值类型还是指针类型而略有变化。

将 u 改为指针类型的 \*user，而不是之前的命名类型 user。

27 func createUserV2() \*user {

28 u := &user{

29 name: "Bill",

30 email: "bill@ardanlabs.com",

31 }

32

33 println("V2", u)

34 return u

35 }

再次生成报告。

./main.go:30: &user literal escapes to heap

./main.go:30: from u (assigned) at

./main.go:28

./main.go:30: from ~r0 (return) at

./main.go:34

现在报告说在 28 行赋值的指针类型 \*user，u 引用的 user 值，因为 34 行的 return 逃逸。

### 结论

值在构建时并不能决定它将存在于哪里。只有当一个值被共享，编译器才能决定如何处理这个值。当你在调用时，共享了栈上的一个值时，它就会逃逸。在下一篇中你将探索一个值逃逸的其他原因。

这些文章试图引导你选择给定类型的值或指针的指导原则。每种方式都有（对应的）好处和（额外的）开销。保持在栈上的值，减少了 GC 的压力。但是需要存储，跟踪和维护不同的副本。将值放在堆上的指针，会增加 GC 的压力。然而，也有它的好处，只有一个值需要存储，跟踪和维护。（其实，）最关键的是如何保持正确地、一致地以及均衡（开销）地使用。

## 内存剖析

### 介绍（Introduction）

在前面的博文中，通过一个共享在 goroutine 的栈上的值的例子讲解了逃逸分析的基础。还有其他没有介绍的造成值逃逸的场景。为了帮助大家理解，我将调试一个分配内存的程序，并使用非常有趣的方法。

### 程序（The Program）

我想了解 io 包，所以我创建了一个简单的项目。给定一个字符序列，写一个函数，可以找到字符串 elvis 并用大写开头的 Elvis 替换它。我们正在讨论国王（Elvis 即猫王，摇滚明星），他的名字总是大写的。

这是一个解决方案的链接：**<https://play.golang.org/p/n_SzF4Cer4>**

这是一个压力测试的链接：**<https://play.golang.org/p/TnXrxJVfLV>**

代码列表里面有两个不同的函数可以解决这个问题。这篇博文将会关注（其中的）algOne 函数，因为它使用到了 io 库。你可以自己用下 algTwo，体验一下内存，CPU 消耗的差异。

Input:

abcelvisaElvisabcelviseelvisaelvisaabeeeelvise l v i saa bb e l v i saa elvi

selvielviselvielvielviselvi1elvielviselvis

Output:

abcElvisaElvisabcElviseElvisaElvisaabeeeElvise l v i saa bb e l v i saa elvi

selviElviselvielviElviselvi1elviElvisElvis

这是完整的 algOne 函数。

func algOne(data []byte, find []byte, repl []byte, output \*bytes.Buffer) {

// Use a bytes Buffer to provide a stream to process.

input := bytes.NewBuffer(data)

// The number of bytes we are looking for.

size := len(find)

// Declare the buffers we need to process the stream.

buf := make([]byte, size)

end := size - 1

// Read in an initial number of bytes we need to get started.

if n, err := io.ReadFull(input, buf[:end]); err != nil {

output.Write(buf[:n])

return

}

for {

// Read in one byte from the input stream.

if \_, err := io.ReadFull(input, buf[end:]); err != nil {

// Flush the reset of the bytes we have.

output.Write(buf[:end])

return

}

// If we have a match, replace the bytes.

if bytes.Compare(buf, find) == 0 {

output.Write(repl)

// Read a new initial number of bytes.

if n, err := io.ReadFull(input, buf[:end]); err != nil {

output.Write(buf[:n])

return

}

continue

}

// Write the front byte since it has been compared.

output.WriteByte(buf[0])

// Slice that front byte out.

copy(buf, buf[1:])

}}

我想知道的是这个函数的性能表现得怎么样，以及它在堆上分配带来什么样的压力。为了这个目的，我们将进行压力测试。

### 压力测试（Benchmarking）

这个是我写的压力测试函数，它在内部调用 algOne 函数去处理数据流。

func BenchmarkAlgorithmOne(b \*testing.B) {

var output bytes.Buffer

in := assembleInputStream()

find := []byte("elvis")

repl := []byte("Elvis")

b.ResetTimer()

for i := 0; i < b.N; i++ {

output.Reset()

algOne(in, find, repl, &output)

}}

有这个压力测试函数，我们就可以运行 go test 并使用 -bench，-benchtime 和 -benchmem 选项。

$ go test -run none -bench AlgorithmOne -benchtime 3s -benchmem

BenchmarkAlgorithmOne-8 2000000 2522 ns/op 117 B/op 2 allocs/op

运行完压力测试后，我们可以看到 algOne 函数分配了两次值，每次分配了 117 个字节。这真的很棒，但我们还需要知道哪行代码造成了分配。为了这个目的，我们需要生成压力测试的分析数据。

### 性能分析（Profiling）

为了生成分析数据，我们将再次运行压力测试，但这次为了生成内存检测数据，我们打开 -memprofile 开关。

$ go test -run none -bench AlgorithmOne -benchtime 3s -benchmem -memprofile mem.out

BenchmarkAlgorithmOne-8 2000000 2570 ns/op 117 B/op 2 allocs/op

一旦压力测试完成，测试工具就会生成两个新的文件。

~/code/go/src/.../memcpu

$ ls -l

total 9248

-rw-r--r-- 1 bill staff 209 May 22 18:11 mem.out (NEW)

-rwxr-xr-x 1 bill staff 2847600 May 22 18:10 memcpu.test (NEW)

-rw-r--r-- 1 bill staff 4761 May 22 18:01 stream.go

-rw-r--r-- 1 bill staff 880 May 22 14:49 stream\_test.go

源码在 memcpu 目录中，algOne 函数在 stream.go 文件中，压力测试函数在 stream\_test.go 文件中。新生成的文件为 mem.out 和 memcpu.test。mem.out 包含分析数据和 memcpu.test 文件，以及包含我们查看分析数据时需要访问符号的二进制文件。

有了分析数据和二进制测试文件，我们就可以运行 pprof 工具学习数据分析。

$ go tool pprof -alloc\_space memcpu.test mem.out

Entering interactive mode (type "help" for commands)

(pprof) \_

当分析内存数据时，为了轻而易举地得到我们要的信息，你会想用 -alloc\_space 选项替代默认的 -inuse\_space 选项。这将会向你展示每一次分配发生在哪里，不管你分析数据时它是不是还在内存中。

在 （pprof） 提示下，我们使用 list 命令检查 algOne 函数。这个命令可以使用正则表达式作为参数找到你要的函数。

(pprof) list algOne

Total: 335.03MB

ROUTINE ======================== .../memcpu.algOne in code/go/src/.../memcpu/stream.go

335.03MB 335.03MB (flat, cum) 100% of Total

. . 78:

. . 79:// algOne is one way to solve the problem.

. . 80:func algOne(data []byte, find []byte, repl []byte, output \*bytes.Buffer) {

. . 81:

. . 82: // Use a bytes Buffer to provide a stream to process.

318.53MB 318.53MB 83: input := bytes.NewBuffer(data)

. . 84:

. . 85: // The number of bytes we are looking for.

. . 86: size := len(find)

. . 87:

. . 88: // Declare the buffers we need to process the stream.

16.50MB 16.50MB 89: buf := make([]byte, size)

. . 90: end := size - 1

. . 91:

. . 92: // Read in an initial number of bytes we need to get started.

. . 93: if n, err := io.ReadFull(input, buf[:end]); err != nil || n < end {

. . 94: output.Write(buf[:n])

(pprof) \_

基于这次的数据分析，我们现在知道了 input，buf 数组在堆中分配。因为 input 是指针变量，分析数据表明 input 指针变量指定的 bytes.Buffer 值分配了。我们先关注 input 内存分配以及弄清楚为啥会被分配。

我们可以假定它被分配是因为调用 bytes.NewBuffer 函数时在栈上共享了 bytes.Buffer 值。然而，存在于 flat 列（pprof 输出的第一列）的值告诉我们值被分配是因为 algOne 函数共享造成了它的逃逸。

我知道 flat 列代表在函数中的分配是因为 list 命令显示 Benchmark 函数中调用了 aglOne。

(pprof) list Benchmark

Total: 335.03MB

ROUTINE ======================== .../memcpu.BenchmarkAlgorithmOne in code/go/src/.../memcpu/stream\_test.go

0 335.03MB (flat, cum) 100% of Total

. . 18: find := []byte("elvis")

. . 19: repl := []byte("Elvis")

. . 20:

. . 21: b.ResetTimer()

. . 22:

. 335.03MB 23: for i := 0; i < b.N; i++ {

. . 24: output.Reset()

. . 25: algOne(in, find, repl, &output)

. . 26: }

. . 27:}

. . 28:

(pprof) \_

因为在 cum 列（第二列）只有一个值，这告诉我 Benchmark 没有直接分配。所有的内存分配都发生在函数调用的循环里。你可以看到这两个 list 调用的分配次数是匹配的。

我们还是不知道为什么 bytes.Buffer 值被分配。这时在 go build 的时候打开 -gcflags "-m -m" 就派上用场了。分析数据只能告诉你哪些值逃逸，但编译命令可以告诉你为啥。

### 编译器报告（Compiler Reporting）

让我们看一下编译器关于代码中逃逸分析的判决。

go build -gcflags "-m -m"

这个命令产生了一大堆的输出。我们只需要搜索输出中包含 stream.go:83，因为 stream.go 是包含这段代码的文件名并且第 83 行包含 bytes.Buffer 的值。搜索后我们找到 6 行。

./stream.go:83: inlining call to bytes.NewBuffer func([]byte) \*bytes.Buffer { return &bytes.Buffer literal }

./stream.go:83: &bytes.Buffer literal escapes to heap

./stream.go:83: from ~r0 (assign-pair) at ./stream.go:83

./stream.go:83: from input (assigned) at ./stream.go:83

./stream.go:83: from input (interface-converted) at ./stream.go:93

./stream.go:83: from input (passed to call[argument escapes]) at ./stream.go:93

我们搜索 stream.go:83 找到的第一行很有趣。

./stream.go:83: inlining call to bytes.NewBuffer func([]byte) \*bytes.Buffer { return &bytes.Buffer literal }

可以肯定 bytes.Buffer 值没有逃逸，因为它传递给了调用栈。这是因为没有调用 bytes.NewBuffer，函数内联处理了。

所以这是我写的代码片段：

83 input := bytes.NewBuffer(data)

因为编译器选择内联 bytes.NewBuffer 函数调用，我写的代码被转成：

input := &bytes.Buffer{buf: data}

这意味着 algOne 函数直接构造 bytes.Buffer 值。

那么，现在的问题是什么造成了值从 algOne 栈帧中逃逸？答案在我们搜索结果中的另外 5 行。

./stream.go:83: &bytes.Buffer literal escapes to heap

./stream.go:83: from ~r0 (assign-pair) at ./stream.go:83

./stream.go:83: from input (assigned) at ./stream.go:83

./stream.go:83: from input (interface-converted) at ./stream.go:93

./stream.go:83: from input (passed to call[argument escapes]) at ./stream.go:93

这几行告诉我们代码中的第 93 行造成了逃逸。input 变量被赋值给一个接口变量。

### 接口（Interfaces）

我完全不记得在代码中将值赋给了接口变量。然而，如果你看到 93 行，就可以非常清楚地看到发生了什么。

93 if n, err := io.ReadFull(input, buf[:end]); err != nil {

94 output.Write(buf[:n])

95 return

96 }

io.ReadFull 调用造成了接口赋值。如果你看了 io.ReadFull 函数的定义，你可以看到一个接口类型是如何接收 input 值。

type Reader interface {

Read(p []byte) (n int, err error)

}

func ReadFull(r Reader, buf []byte) (n int, err error) {

return ReadAtLeast(r, buf, len(buf))

}

传递 bytes.Buffer 地址到调用栈，在 Reader 接口变量中存储会造成一次逃逸。现在我们知道**使用接口变量是需要开销的：分配和重定向**。所以，如果没有很明显的使用接口的原因，你可能不想使用接口。下面是我选择在我的代码中是否使用接口的原则。

**使用接口的情况**：

* 用户 API 需要提供实现细节的时候。
* API 的内部需要维护多种实现。
* 可以改变的 API 部分已经被识别并需要解耦。

**不使用接口的情况**：

* 为了使用接口而使用接口。
* 推广算法。
* 当用户可以定义自己的接口时。

现在我们可以问自己，这个算法真的需要 io.ReadFull 函数吗？答案是否定的，因为bytes.Buffer` 类型有一个方法可以供我们使用。使用方法而不是调用一个函数可以防止重新分配内存。

让我们修改代码，删除 io 包，并直接使用 Read 函数而不是 input 变量。

修改后的代码删除了 io 包的调用，为了保留相同的行号，我使用空标志符替代 io 包的引用。这会允许（没有使用的）库导入的行待在列表中。

import (

"bytes"

"fmt"

\_ "io"

)

func algOne(data []byte, find []byte, repl []byte, output \*bytes.Buffer) {

// Use a bytes Buffer to provide a stream to process.

input := bytes.NewBuffer(data)

// The number of bytes we are looking for.

size := len(find)

// Declare the buffers we need to process the stream.

buf := make([]byte, size)

end := size - 1

// Read in an initial number of bytes we need to get started.

if n, err := input.Read(buf[:end]); err != nil || n < end {

output.Write(buf[:n])

return

}

for {

// Read in one byte from the input stream.

if \_, err := input.Read(buf[end:]); err != nil {

// Flush the reset of the bytes we have.

output.Write(buf[:end])

return

}

// If we have a match, replace the bytes.

if bytes.Compare(buf, find) == 0 {

output.Write(repl)

// Read a new initial number of bytes.

if n, err := input.Read(buf[:end]); err != nil || n < end {

output.Write(buf[:n])

return

}

continue

}

// Write the front byte since it has been compared.

output.WriteByte(buf[0])

// Slice that front byte out.

copy(buf, buf[1:])

}}

修改后我们执行压力测试，可以看到 bytes.Buffer 的分配消失了。

$ go test -run none -bench AlgorithmOne -benchtime 3s -benchmem -memprofile mem.out

BenchmarkAlgorithmOne-8 2000000 1814 ns/op 5 B/op 1 allocs/op

我们可以看到大约 29% 的性能提升。代码从 2570 ns/op 降到 1814 ns/op。解决了这个问题，我们现在可以关注 buf 切片数组。如果再次使用测试代码生成分析数据，我们应该能够识别到造成剩下的分配的原因。

$ go tool pprof -alloc\_space memcpu.test mem.out

Entering interactive mode (type "help" for commands)

(pprof) list algOne

Total: 7.50MB

ROUTINE ======================== .../memcpu.BenchmarkAlgorithmOne in code/go/src/.../memcpu/stream\_test.go

11MB 11MB (flat, cum) 100% of Total

. . 84:

. . 85: // The number of bytes we are looking for.

. . 86: size := len(find)

. . 87:

. . 88: // Declare the buffers we need to process the stream.

11MB 11MB 89: buf := make([]byte, size)

. . 90: end := size - 1

. . 91:

. . 92: // Read in an initial number of bytes we need to get started.

. . 93: if n, err := input.Read(buf[:end]); err != nil || n < end {

. . 94: output.Write(buf[:n])

只剩下 89 行所示，对数组切片的分配。

### 栈帧

想知道造成 buf 数组切片的分配的原因？让我们再次运行 go build，并使用 -gcflags "-m -m" 选项并搜索 stream.go:89。

$ go build -gcflags "-m -m"

./stream.go:89: make([]byte, size) escapes to heap

./stream.go:89: from make([]byte, size) (too large for stack) at ./stream.go:89

报告显示，对于栈来说，数组太大了。这个信息误导了我们。并不是说底层的数组太大，而是编译器在编译时并不知道数组的大小。

值只有在编译器编译时知道其大小才会将它分配到栈中。这是因为每个函数的栈帧大小是在编译时计算的。如果编译器不知道其大小，就只会在堆中分配。

为了验证（我们的想法），我们将值硬编码为 5，然后再次运行压力测试。

89 buf := make([]byte, 5)

这一次我们运行压力测试，分配消失了。

$ go test -run none -bench AlgorithmOne -benchtime 3s -benchmem

BenchmarkAlgorithmOne-8 3000000 1720 ns/op 0 B/op 0 allocs/op

如果你再看一下编译器报告，你会发现没有需要逃逸处理的。

$ go build -gcflags "-m -m"

./stream.go:83: algOne &bytes.Buffer literal does not escape

./stream.go:89: algOne make([]byte, 5) does not escape

很明显我们无法确定切片的大小，所以我们在算法中需要一次分配。

### 分配和性能（Allocation and Performance）

比较一下我们在重构过程中，每次提升的性能。

Before any optimization

BenchmarkAlgorithmOne-8 2000000 2570 ns/op 117 B/op 2 allocs/op

Removing the bytes.Buffer allocation

BenchmarkAlgorithmOne-8 2000000 1814 ns/op 5 B/op 1 allocs/op

Removing the backing array allocation

BenchmarkAlgorithmOne-8 3000000 1720 ns/op 0 B/op 0 allocs/op

删除掉 bytes.Buffer 里面的（重新）内存分配，我们获得了大约 29% 的性能提升，删除掉所有的分配，我们能获得大约 33% 的性能提升。内存分配是应用程序性能影响因素之一。

### 结论（Conclusion）

Go 拥有一些神奇的工具使你能了解编译器作出的跟逃逸分析相关的一些决定。基于这些信息，你可以通过重构代码使得值存在于栈中而不需要在（被重新分配到）堆中。你不是想去掉所有软件中所有的内存（再）分配，而是想最小化这些分配。

这就是说，写程序时永远不要把性能作为第一优先级，因为你并不想（在写程序时）一直猜测性能。写正确的代码才是你第一优先级。这意味着，我们首先要关注的是完整性、可读性和简单性。一旦有了可以运行的程序，才需要确定程序是否足够快。假如程序不够快，那么使用语言提供的工具来查找和解决性能问题。

## 数据和语法设计哲学

### 设计哲学（Design Philosophies）

"在栈上保存值，这减少了垃圾收集器（GC）的压力。然而，却要求存储、跟踪和维护给定值的多个副本。将值放在堆上，这会给 GC 增加压力。但是它也是有用的，因为只需要针对一个值进行存储、跟踪和维护。" - Bill Kennedy

对于给定类型的数据，想在整个软件中保持完整性和可读性，使用值或者指针要保持一致。为什么？因为，如果你在函数间传递数据时修改数据语义，将很难维护一个清晰一致的心智模型。代码库和团队越大，越多的 bug、对数据的竞争和其他副作用就会悄悄地潜入到代码库中。

我想从一组设计哲学开始讨论，它将指导（我们如何）选择一种语义而不是另外一种语义的方法。

### 心智模型（Mental Models）

（译者注：心智模型是经由经验及学习，脑海中对某些事物发展的过程，所写下的剧本。可以当成对代码整体的把控）

"让我们想象有这样一个项目，它包含一百万行以上的代码量。这些项目当前在美国能成功的可能性很低，远低于 50%。或许有人不同意这个说法。" - Tom Love (inventor of Objective C)

Tom 还说一盒复印纸可以容纳 10 万行代码。稍微想一下。你能掌控这个盒子中的代码的百分之多少呢？

我相信要一个开发人员维护一张纸上的代码的心智模型（大约 1 万行代码）已经是个问题。但是，我们还是假设每个开发人员开发 1 万行代码，那么需要由 100 位开发人员组成的团队来维护一个包含 100 万行代码的代码库。也就是说 100 人需要协调，分组，跟踪和不断沟通。现在，再看看你们 1 到 10 名开发人员组成的团队。你们在这个小得多的规模做得如何？假设每人 1 万行代码，（你们）团队规模与代码库的大小是否相符？

### 调试（Debugging）

"最大的问题是你的心智模型是错误的，所以你根本找不到问题所在。" - Brian Kernighan

我不相信，你能在没有心智模型的基础上，使用调试器解决问题，你只不过是在浪费时间精力尝试理解问题。

如果你在生产环境中遇到问题，你能问谁？没错，**日志**。如果日志在你开发过程中对你没有用，那么当生产环境上出问题，它也一定对你没有用。日志应该基于代码的心智模型，这样才能通过阅读代码找到问题所在。

### 可读性（Readability）

C 语言是我见过的在性能和表达性上平衡得最好的。你可以通过简单的编程实现任何你想要做的事情，并且你会对机器即将要发生的事情拥有一个非常好的心智模型。你可以非常合理地预测它的速度，你知道即将要发生什么..." - Brian Kernighan

我相信 Brian 这句话也适用于 Go。保持这种 "心智模型" 就是一切。它驱动完整性，可读性和简单性。这些是精心编写的软件的基石，使得它可以保持正常并持续运行下去。编写保证给定类型数据的值或者指针语义一致的代码是实现这一点的重要方法。

### 面向数据设计（Data Oriented Design）

"如果你不了解这些数据，你就不明白这个问题。因为所有的问题都是独特的，并且与你所使用的数据关系紧密。当数据发生变化时，你的问题也会跟着变化。但问题发生变化时，你的算法（数据转换）也需要跟着变化。" - Bill Kennedy

想一想。你解决问题的方法实际上是解决数据转换的问题。你写的每个函数，运行的每个程序，（只不过）都是获取一些输入数据，产生一些输出数据。从这个角度看，你的软件的心智模型就是对这些数据转换的理解（例如，如何在代码中组织和使用它们）。"少即是多" 的原则对于解决问题时实现较少的层数，代码量，迭代次数，以及降低复杂性和减少工作量非常重要。

### 类型（就是生命）（Type (Is Life)）

"完整性意味着每次分配内存，读取内存和写入内存都是准确，一致和高效的。类型系统对于我们具有这种微观完整性至关重要。" - William Kennedy

如果数据驱动你所做的一切，那么代表数据的类型就十分地重要。在我的观点里面 "类型就是生命"，因为类型为编译器提供了确保数据完整性的能力。类型也驱动并指示语义规则，程序必须遵循其所操作的数据的语义。这是正确地使用值或者指针语义的开始：使用类型。

### 数据（的能力）

"当数据是实际和合理的，方法才是有效的。" - William Kennedy

值或者指针语义的思想不会直接影响 Go 开发人员，除非他们需要决定方法接收值还是指针。这是我遇到的一个问题：我应该使用值作为参数还是指针？一听到这个问题，我就知道这个开发人员没有理解好这些（类型的）语义。

方法的目的是使这些数据具有某种能力。想象一下，数据有能力做某些事情。我总是希望把重点放在数据上，因为它驱动程序的功能。数据驱动你写的算法，封装和能达到的性能。

### 多态（Polymorphism）

"多态意味着你写了一个特定的程序，但它的行为有所不同，具体取决于它所操作的数据。" - Tom Kurtz (inventor of BASIC)

我很喜欢 Tom 上面说的话。函数的行为可以根据操作的数据的不同而不同。这个数据的行为是将函数从它们可以接受和使用的具体数据类型中分离出来的，这是数据可以具有某种能力的原因。这个观点是使得架构和设计可以适应变化的系统的基石。

### 原型的第一种方法（Prototype First Approach）

"除非开发人员对软件会被如何使用有一个很好的了解，否则软件很可能会出问题。如果开发人员不是很了解或者对软件不是很理解，那么获得尽可能多的用户输入和用户级测试就相当的重要。" - Brian Kernighan

我希望你始终专注于理解具体的数据和为了解决问题所需要的数据转换的算法。采用这种原型的第一种方法，编写也可以在生产环境中部署的具体实现（如果这样做是合理和实际的话）。一旦一个具体的实现已经能够工作，一旦你已经知道哪些工作起作用，哪些不起作用，就应该关注于重构，将实现与具体数据分离，将之赋予数据以能力（译者注：我的理解，简单地说，就是抽象为数据类型的一个方法）。

### 语义原则（Semantic Guidelines）

你在声明类型时，必须决定特定数据类型将使用哪种语义，值或者指针。接收或返回该类型数据的 API 必须遵循为该类型选择的语义。API 不允许（用户）指定或改变语义。他们必须知道数据使用什么语义，并符合这一点。这是实现大型代码库一致性的起码要求。

以下是基本指导原则：

* 当你声明一个类型时，你必须决定所使用的语义
* 函数和方法必须遵循给定类型所选择的语义
* 避免让方法接收与给定类型相对应的不同语义
* 避免函数接收或者返回与给定类型相对应的不同语义
* 避免改变给定类型的语义

这些指导原则有一些例外的情况，最大的是 unmarshaling。Unmarshaling 总是需要使用指针语义。Marshaling 和 unmarshaling 似乎总是例外的规则。

你如何选择一种给定类型的一种语义而不是另外一种？这些指导方针将回答这个问题。以下我们将在具体的情况下使用指导原则：

### 内置类型

Go 语言中内置类型包括数字，文本和布尔类型。这些类型应该使用值语义进行处理。除非你有非常好的理由，否则不要使用指针来共享这些类型的值。

作为一个例子，从 strings 包中查看这些函数的声明。

func Replace(s, old, new string, n int) string

func LastIndex(s, sep string) int

func ContainsRune(s string, r rune) bool

所有这些函数在 API 设置中都使用值语义。

### 引用类型

Go 语言中引用类型包括切片，map，接口，函数和 channel。这些类型建议使用值语义，因为它们被设计成待在栈中以最小化堆的压力。它们允许每个函数都有自己的值副本，而不是每个函数都会造成潜在的分配。这是可能的，因为这些值包含一个在调用之间共享底层数据结构的指针。

除非你有很好的理由，否则不要用指针共享这些类型的值。将调用栈中的 map 或 slice 共享给 Unmarshal 函数可能是一个例外。作为一个例子，看看 net 库上声明的这两种类型。

type IP []byte

type IPMask []byte

IP 和 IPMask 都是字节切片。这意味着它们是引用类型，并且它们应该要符合值语义。下面是一个名叫 Mask 的方法，它被声明为接收一个 IPMask 值的 IP 类型。

func (ip IP) Mask(mask IPMask) IP {

if len(mask) == IPv6len && len(ip) == IPv4len && allFF(mask[:12]) {

mask = mask[12:]

}

if len(mask) == IPv4len && len(ip) == IPv6len && bytesEqual(ip[:12], v4InV6Prefix) {

ip = ip[12:]

}

n := len(ip)

if n != len(mask) {

return nil

}

out := make(IP, n)

for i := 0; i < n; i++ {

out[i] = ip[i] & mask[i]

}

return out

}

请注意，此方法是一种转变操作，并使用值语义的 API 样式。它使用 IP 值作为接收方，并根据传入的 IPMask 值创建一个新的 IP 值并将其返回给调用方。该方法遵循对引用类型使用值语义（的基本指导原则）。

这跟系统默认的 append 函数有点相似。

var data []string

data = append(data, "string")

append 函数的转变操作使用值语义。将切片值传递给 append，并在变化之后返回一个新切片值。

总是除了 unmarshaling，它需要使用指针语义。

func (ip \*IP) UnmarshalText(text []byte) error {

if len(text) == 0 {

\*ip = nil

return nil

}

s := string(text)

x := ParseIP(s)

if x == nil {

return &ParseError{Type: "IP address", Text: s}

}

\*ip = x

return nil

}

UnmarshalText 实现 encoding.TextUnmarshaler 接口。如果没有使用指针语义，根本无法实现。但这是可以的，因为共享值通常是安全的。除了 unmarshaling 之外，如果为一个引用类型使用指针语义，你应该三思。

#### 用户定义类型（User Defined Types）

这是你最多需要作出决定的地方。你必须在你声明类型的时候决定使用什么语义。

如果我要求你给 time 包编写 API 接口，给你这种类型。

type Time struct {

sec int64

nsec int32

loc \*Location

}

你会使用什么语义？

在 Time 包中查看此类型的实现以及工厂函数 Now。

func Now() Time {

sec, nsec := now()

return Time{sec + unixToInternal, nsec, Local}

}

工厂函数对于类型来说是一种非常重要的函数，因为它告诉你（这种类型）所选择的语义。Now 函数就很清晰地（向我们）表明使用了值语义。该函数创建一个类型为 Time 的值并将该值的副本返回给调用者。 共享 Time 值不是必要的，（因为）他们的生命周期内不需要一直存在于堆上。

再看一下 Add 方法，它也是一个转变操作。

func (t Time) Add(d Duration) Time {

t.sec += int64(d / 1e9)

nsec := t.nsec + int32(d%1e9)

if nsec >= 1e9 {

t.sec++

nsec -= 1e9

} else if nsec < 0 {

t.sec--

nsec += 1e9

}

t.nsec = nsec

return t

}

你可以再次看到 Add 方法遵循类型所选择的语义。Add 方法使用一个值接收器来操作它自己的 Time 值副本。其中，Time 值副本在调用中使用。它将修改自己的副本，并将 Time 值的新副本返回给调用者。

以下是一个接受 Time 值的函数：

func div(t Time, d Duration) (qmod2 int, r Duration) {

再一次，接受 Time 类型的值使用值语义。唯一使用指针语义的 Time API 接口，是这些 Unmarshal 相关的函数：

func (t \*Time) UnmarshalBinary(data []byte) error {

func (t \*Time) GobDecode(data []byte) error {

func (t \*Time) UnmarshalJSON(data []byte) error {

func (t \*Time) UnmarshalText(data []byte) error {

大多数情况下，使用值语义的能力是有限的。将值从一个函数传递到另一个函数，（通常）使用值拷贝的方法是不正确或者不合理的。修改数据需要将其隔离成单个值再进行共享。这时，应该使用指针语义。如果你没办法 100% 确定拷贝值是正确并且合理的，那就使用指针语义吧。

查看 os 包中的 File 类型的生产函数。

func Open(name string) (file \*File, err error) {

return OpenFile(name, O\_RDONLY, 0)

}

Open 函数返回一个 File 类型的指针。这意味着，对于 File 类型值，你应该使用指针语义来共享 File 的值。将指针语义修改为值语义，可能会对你的程序造成破坏性影响。当你与一个函数共享值时，最好假定你不允许拷贝值的指针并使用这个指针。否则，不知道将会出现什么样的异常情况。

查看更多的 API， 你将会看到更多使用指针语义的例子。

func (f \*File) Chdir() error {

if f == nil {

return ErrInvalid

}

if e := syscall.Fchdir(f.fd); e != nil {

return &PathError{"chdir", f.name, e}

}

return nil

}

虽然 File 值永远不会被修改，但是 Chdir 方法还是使用指针语义。该方法必须遵循该类型的语义约定。

func epipecheck(file \*File, e error) {

if e == syscall.EPIPE {

if atomic.AddInt32(&file.nepipe, 1) >= 10 {

sigpipe()

}

} else {

atomic.StoreInt32(&file.nepipe, 0)

}

}

这是一个名为 epipecheck 的函数，它使用指针来接收 File 值。再次注意一下，对于 File 值，一致使用指针语义。

### 结论

我在做代码 review 时，会寻找值或者指针语义是否使用一致。它可以帮助你保证代码的一致性和可预测性。它还使每个人能保持清晰和一致的心智模型。随着代码库和团队变得越来越大，值或者指针语义的一致性使用将会越来越重要。

Go 语言令人不解的地方在于指针和值语义之间的选择早已超出了接收器和函数参数的声明范围。接口的机制，函数值和切片都在语言的工作范围内。在将来的文章中，我将在这些语言的不同部分中展示值或者指针语义。

# Go 中的调度器

这篇文章是三部曲系列文章中的第一篇，这个系列的文章将会对 Go 中调度器背后的机制和语义做深入的了解。本文主要关注操作系统调度器的部分。

Go 调度器系列文章：

* **[Go 中的调度器：第一部分 - 操作系统调度器](https://studygolang.com/articles/14264)**
* **[Go 中的调度器：第二部分 - Go 调度器](https://studygolang.com/articles/15316)**
* **[Go 中的调度器：第三部分 - 并发](https://studygolang.com/articles/17014)**

## 第一部分-操作系统调度器

### 简介

首先，Golang 调度器的设计和实现让我们的 Go 程序在多线程执行时效率更高，性能更好。这要归功于 Go 调度器与操作系统（OS）调度器的协同合作。不过在本篇文章中，多线程 Go 程序在设计和实现上是否与调度器的工作原理完全契合不是重点。重要的是对系统调度器和 Go 调度器，它们是如何正确地设计多线程程序，有一个全面且深入的理解。

本章多数内容将侧重于讨论调度器的高级机制和语义。我将展示一些细节，让你可以通过图像来理解它们是如何工作的，可以让你在写代码时做出更好的决策。因为原理和语义是必备的基础知识中的关键。

### 系统调度

操作系统调度器是一个复杂的程序。它们要考虑到运行时的硬件设计和设置，其中包括但不限于多处理器核心、CPU 缓存和 NUMA，只有考虑全面，调度器才能做到尽可能地高效。值得高兴的是，你不需要深入研究这些问题，就可以大致上了解操作系统调度器是如何工作的。

你的代码会被翻译成一系列机器指令，然后依次执行。为了实现这一点，操作系统使用线程（Thread）的概念。线程负责顺序执行分配给它的指令。一直执行没有指令为止。这就是我将线程称为“执行通路”的原因。

你运行的每个程序都会创建一个进程，每个进程都有一个初始线程。而后线程可以创建更多的线程。每个线程互相独立地运行着，调度是在线程级别而不是在进程级别做出的。****线程可以并发运行(每个线程在单个内核上轮流运行)，也可以并行运行(每个线程在不同的内核上同时运行)。****线程还维护自己的状态，以便安全、本地和独立地执行它们的指令。

如果有线程可以执行，操作系统调度器就会调度它到空闲的 CPU 核心上去执行，保证 CPU 不闲着。它还必须模拟一个假象，即所有可以执行的线程都在同时地执行着。在这个过程中，调度器还会根据优先级不同选择线程执行的先后顺序，高优先级的先执行，低优先级的后执行。当然，低优先级的线程也不会被饿着。调度器还需要通过快速而明智的决策尽可能减少调度延迟。

为了实现这一目标，算法在其中做了很多工作，且幸运的是，这个领域已经积累了几十年经验。为了我们能更好地理解这一切，接下来我们来看几个重要的概念。

### 执行指令

程序计数器(PC)，有时称为指令指针(IP)，线程利用它来跟踪下一个要执行的指令。在大多数处理器中，PC指向的是下一条指令，而不是当前指令。

如果你之前看过 Go 程序的堆栈跟踪，那么你可能已经注意到了每行末尾的这些十六进制数字。如下：

goroutine 1 [running]:

main.example(0xc000042748, 0x2, 0x4, 0x106abae, 0x5, 0xa)

stack\_trace/example1/example1.go:13 +0x39 <- LOOK HERE

main.main()

stack\_trace/example1/example1.go:8 +0x72 <- LOOK HERE

这些数字表示 PC 值与相应函数顶部的偏移量。+0x39PC 偏移量表示在程序没中断的情况下，线程即将执行的下一条指令。如果控制权回到主函数中，则主函数中的下一条指令是0+x72PC 偏移量。更重要的是，指针前面的指令是当前正在执行的指令。

下面是对应的代码

https://github.com/ardanlabs/gotraining/blob/master/topics/go/profiling/stack\_trace/example1/example1.go

07 func main() {

08 example(make([]string, 2, 4), "hello", 10)

09 }

12 func example(slice []string, str string, i int) {

13 panic("Want stack trace")

14 }

十六进制数+0x39表示示例函数内的一条指令的 PC 偏移量，该指令位于函数的起始指令后面第57条(10进制)。接下来，我们用 objdump 来看一下汇编指令。找到第57条指令，注意，runtime.gopanic那一行。

$ go tool objdump -S -s "main.example" ./example1

TEXT main.example(SB) stack\_trace/example1/example1.go

func example(slice []string, str string, i int) {

0x104dfa0 65488b0c2530000000 MOVQ GS:0x30, CX

0x104dfa9 483b6110 CMPQ 0x10(CX), SP

0x104dfad 762c JBE 0x104dfdb

0x104dfaf 4883ec18 SUBQ $0x18, SP

0x104dfb3 48896c2410 MOVQ BP, 0x10(SP)

0x104dfb8 488d6c2410 LEAQ 0x10(SP), BP

panic("Want stack trace")

0x104dfbd 488d059ca20000 LEAQ runtime.types+41504(SB), AX

0x104dfc4 48890424 MOVQ AX, 0(SP)

0x104dfc8 488d05a1870200 LEAQ main.statictmp\_0(SB), AX

0x104dfcf 4889442408 MOVQ AX, 0x8(SP)

0x104dfd4 e8c735fdff CALL runtime.gopanic(SB)

0x104dfd9 0f0b UD2 <--- 这里是 PC(+0x39)

****记住: PC 是下一个指令，而不是当前指令****。上面是基于 amd64 的汇编指令的一个很好的例子，该 Go 程序的线程负责顺序执行。

### 线程状态

另一个重要的概念是线程状态，它描述了调度器在线程中的角色。  
线程可以处于三种状态之一: 等待中(Waiting)、待执行(Runnable)或执行中(Executing)。

等待中(Waiting):这意味着线程停止并等待某件事情以继续。这可能是因为等待硬件(磁盘、网络)、操作系统(系统调用)或同步调用(原子、互斥)等原因。这些类型的延迟是性能下降的根本原因。

待执行(Runnable):这意味着线程需要内核上的时间，以便执行它指定的机器指令。如果有很多线程都需要时间，那么线程需要等待更长的时间才能获得执行。此外，由于更多的线程在竞争，每个线程获得的单个执行时间都会缩短。这种类型的调度延迟也可能导致性能下降。

执行中(Executing):这意味着线程已经被放置在一个核心上，并且正在执行它的机器指令。与应用程序相关的工作正在完成。这是每个人都想要的。

### 工作类型

线程可以做两种类型的工作。第一个称为 ****CPU-Bound****，第二个称为 ****IO-Bound****。

****CPU-Bound****：这种工作类型永远也不会让线程处在等待状态，因为这是一项不断进行计算的工作。比如计算 π 的第 n 位，就是一个 CPU-Bound 线程。

****IO-Bound****：这是导致线程进入等待状态的工作类型。比如通过网络请求对资源的访问或对操作系统进行系统调用。

### 上下文切换

诸如 Linux、Mac、 Windows 是一个具有抢占式调度器的操作系统。这意味着一些重要的事情。首先，这意味着调度程序在什么时候选择运行哪些线程是不可预测的。线程优先级和事件混在一起(比如在网络上接收数据)使得无法确定调度程序将选择做什么以及什么时候做。

其次，这意味着你永远不能基于一些你曾经历过但不能保证每次都发生的行为来编写代码。如果应用程序中需要确定性，则必须控制线程的同步和协调管理。

在核心上交换线程的物理行为称为上下文切换。当调度器将一个正在执行的线程从内核中取出并将其更改状态为一个可运行的线程时，就会发生上下文切换。

上下文切换的代价是高昂的，因为在核心上交换线程会话费很多时间。上下文切换的延迟取决于不同的因素，大概在在 50 到 100 纳秒之间。考虑到硬件应该能够合理地(平均)在每个核心上每纳秒执行 12 条指令，那么一次上下文切换可能会花费 600 到 1200 条指令的延迟时间。实际上，上下文切换占用了大量程序执行指令的时间。

如果你在执行一个 IO-Bound 程序，那么上下文切换将是一个优势。一旦一个线程更改到等待状态，另一个处于可运行状态的线程就会取而代之。这使得 CPU 总是在工作。这是调度器最重要的之一，最好不要让 CPU 闲下来。

而如果你在执行一个 CPU-Bound 程序，那么上下文切换将成为性能瓶颈的噩梦。由于线程总是有工作要做，所以上下文切换阻碍了工作的进展。这种情况与 IO-Bound 类型的工作形成了鲜明对比。

### 少即是多

在早期处理器只有一个核心的时代，调度相对简单。因为只有一个核心，所以物理上在任何时候都只有一个线程可以执行。其思想是定义一个调度程序周期，并尝试在这段时间内执行所有可运行线程。****算法很简单：用调度周期除以需要执行的线程数。****

例如，如果你将调度器周期定义为 10ms(毫秒)，并且你有 2 个线程，那么每个线程将分别获得 5ms。如果你有 5 个线程，每个线程得到 2ms。但是，如果有 100 个线程，会发生什么情况呢？给每个线程一个时间片 10μs (微秒)？错了，这么干是愚蠢的，因为你会话费大量的时间在上下文切换上，而真正的工作却做不成。

你需要限制时间片的长度。在最后一个场景中，如果最小时间片是 2ms，并且有 100 个线程，那么调度器周期需要增加到 2s(秒)。如果有 1000 个线程，那么调度器周期就是 20s。在这个简单的例子中，如果每个线程使用它的全时间片，那么所有线程运行一次需要花费 20s。

要知道，这是一个非常简单的场景。在真正进行调度决策时，调度程序需要考虑和处理比这更多的事情。你可以控制应用程序中使用的线程数量。当有更多的线程要考虑，并且发生 IO-Bound 工作时，就会出现一些混乱和不确定的行为。任务需要更长的时间来调度和执行。

这就是为什么游戏规则是“少即是多”。处于可运行状态的线程越少，意味着调度开销越少，每个线程执行的时间越长。完成的工作会越多。如此，效率就越高。

### 寻找一个平衡

你需要在 ****CPU 核心数****和为应用程序获得最佳吞吐量所需的****线程数****之间找到****平衡****。当涉及到管理这种平衡时，线程池是一个很好的解决方案。将在第二部分中为你解析，Go 不是这样做的。

### CPU 缓存

从主存访问数据有很高的延迟成本(大约 100 到 300 个时钟周期)，因此处理器核心使用本地高速缓存来将数据保存在需要的硬件线程附近。从缓存访问数据的成本要低得多(大约 3 到 40 个时钟周期)，这取决于所访问的缓存。如今，提高性能的一个方面是关于如何有效地将数据放入处理器以减少这些数据访问延迟。编写多线程应用程序也需要考虑 CPU 缓存的机制。

数据通过cache lines在处理器和主存储器之间交换。cache line是在主存和高速缓存系统之间交换的 64 字节内存块。每个内核都有自己所需的cache line的副本，这意味着硬件使用值语义。这就是为什么多线程应用程序中内存的变化会造成性能噩梦。

当并行运行的多个线程正在访问相同的数据值，甚至是相邻的数据值时，它们将访问同一cache line上的数据。在任何核心上运行的任何线程都将获得同一cache line的副本。

如果某个核心上的一个线程对其cache line的副本进行了更改，那么同一cache line的所有其他副本都必须标记为dirty的。当线程尝试对dirty cache line进行读写访问时，需要向主存访问(大约 100 到 300 个时钟周期)来获得cache line的新副本。

也许在一个 2 核处理器上这不是什么大问题，但是如果一个 32 核处理器在同一cache line上同时运行 32 个线程来访问和改变数据，那会发生什么？如果一个系统有两个物理处理器，每个处理器有16个核心，那又该怎么办呢？这将变得更糟，因为处理器到处理器的通信延迟更大。应用程序将会在主存中周转，性能将会大幅下降。

这被称为**缓存一致性问题**，还引入了错误共享等问题。在编写可能会改变共享状态的多线程应用程序时，必须考虑缓存系统。

### 调度决策场景

假设我要求你基于我给你的信息编写操作系统调度器。考虑一下这个你必须考虑的情况。记住，这是调度程序在做出调度决策时必须考虑的许多有趣的事情之一。

启动应用程序，创建主线程并在核心1上执行。当线程开始执行其指令时，由于需要数据，正在检索cache line。现在，线程决定为一些并发处理创建一个新线程。下面是问题：

1. 进行上下文切换，切出核心1的主线程，切入新线程？这样做有助于提高性能，因为这个新线程需要的相同部分的数据很可能已经被缓存。但主线程没有得到它的全部时间片。
2. 新线程等待核心1在主线程完成之前变为可用？线程没有运行，但一旦启动，获取数据的延迟将被消除。
3. 线程等待下一个可用的核心？这意味着所选核心的cache line将被刷新、检索和复制，从而导致延迟。然而，线程将启动得更快，主线程可以完成它的时间片。

有意思吗？这些是系统调度器在做出调度决策时需要考虑的有趣问题。幸运的是，不是我做的。我能告诉你的就是，如果有一个空闲核心，它将被使用。你希望线程在可以运行时运行。

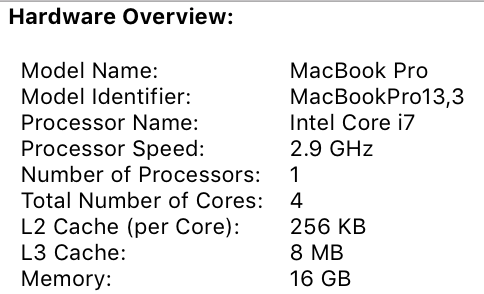
### 结论

本文的第一部分深入介绍了在编写多线程应用程序时需要考虑的关于线程和系统调度器的问题。这些是 Go 调度器也要考虑的事情。在下一篇文章中，我将解析 Go 调度器的语义以及它们如何与这些信息相关联，并通过一些示例程序来展示。

## 第二部分-Go调度器

### 开始

****当 Go 程序启动时，它会为主机上标识的每个虚拟核心提供一个逻辑处理器（P）****。如果处理器每个物理核心可以提供多个硬件线程（超线程），那么每个硬件线程都将作为虚拟核心呈现给 Go 程序。为了更好地理解这一点，下面实验都基于如下配置的 MacBook Pro 的系统。



可以看到它是一个 4 核 8 线程的处理器。这将告诉 Go 程序有 8 个虚拟核心可用于并行执行系统线程。

用下面的程序来验证一下:

package main

import (

"fmt"

"runtime"

)

func main() {

// NumCPU 返回当前可用的逻辑处理核心的数量

fmt.Println(runtime.NumCPU())

}

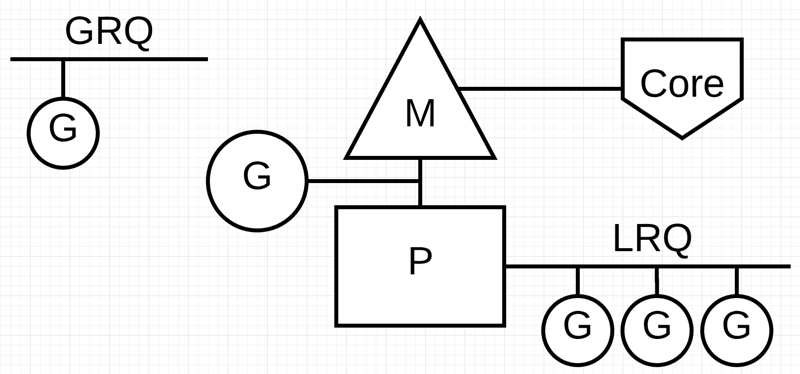
当我运行该程序时，NumCPU() 函数调用的结果将是 8 。意味着在我的机器上运行的任何 Go 程序都将被赋予 8 个 **P**。

****每个**P**都被分配一个系统线程**M**。M 代表机器（machine），它仍然是由操作系统管理的，操作系统负责将线程放在一个核心上执行。这意味着当在我的机器上运行 Go 程序时，有 8 个线程可以执行我的工作，每个线程单独连接到一个 P。

****每个 Go 程序都有一个初始**G**。G 代表 Go 协程（Goroutine），它是 Go 程序的执行路径。Goroutine 本质上是一个 **[Coroutine](https://en.wikipedia.org/wiki/Coroutine)**，但因为是 Go 语言，所以把字母 “C” 换成了 “G”，我们得到了这个词。你可以将 Goroutines 看作是应用程序级别的线程，它在许多方面与系统线程都相似。正如系统线程在物理核心上进行上下文切换一样，Goroutines 在 **M** 上进行上下文切换。

最后一个重点是运行队列。Go 调度器中有两个不同的运行队列：**全局运行队列(GRQ)**和**本地运行队列(LRQ)**。****每个**P**都有一个LRQ****，用于管理分配给在**P**的上下文中执行的 Goroutines，这些 Goroutine 轮流被**和P绑定的M**进行上下文切换。GRQ 适用于尚未分配给**P**的 Goroutines。其中有一个过程是将 Goroutines 从 GRQ 转移到 LRQ，我们将在稍后讨论。

下面图示展示了它们之间的关系：



### 协作式调度器

正如我们在第一篇文章中所讨论的，OS 调度器是一个抢占式调度器。从本质上看，这意味着你无法预测调度程序在任何给定时间将执行的操作。由内核做决定，一切都是不确定的。在操作系统之上运行的应用程序无法通过调度控制内核内部发生的事情，除非它们利用像 **[atomic](https://en.wikipedia.org/wiki/Linearizability)** 指令 和 **[mutex](https://en.wikipedia.org/wiki/Lock_(computer_science))** 调用之类的同步原语。

Go 调度器是 Go 运行时的一部分，Go 运行时内置在应用程序中。这意味着 Go 调度器在内核之上的用户空间中运行。Go 调度器的当前实现不是抢占式调度器，而是协作式调度器。作为一个协作的调度器，意味着调度器需要明确定义用户空间事件，这些事件发生在代码中的安全点，以做出调度决策。

Go 协作式调度器的优点在于它看起来和感觉上都是抢占式的。你无法预测 Go 调度器将会执行的操作。这是因为这个协作调度器的决策不掌握在开发人员手中，而是在 Go 运行时。将 Go 调度器视为抢占式调度器是非常重要的，并且由于调度程序是非确定性的，因此这并不是一件容易的事。

### Goroutine 状态

就像线程一样，Goroutines 有相同的三个高级状态。它们标识了 Go 调度器在任何给定的 Goroutine 中所起的作用。Goroutine 可以处于三种状态之一：**Waiting**（等待状态）****、**Runnable**（可运行状态）****或**Executing**（运行中状态）****。

**Waiting**：****这意味着 Goroutine 已停止并等待一些事情以继续。这可能是因为等待操作系统（系统调用）或同步调用（原子和互斥操作）等原因。这些类型的延迟是性能下降的根本原因。

**Runnable **：****这意味着 Goroutine 需要**M**上的时间片，来执行它的指令。如果同一时间有很多 Goroutines 在竞争时间片，它们都必须等待更长时间才能得到时间片，而且每个 Goroutine 获得的时间片都缩短了。这种类型的调度延迟也可能导致性能下降。

**Executing **：****这意味着 Goroutine 已经被放置在**M**上并且正在执行它的指令。与应用程序相关的工作正在完成。这是每个人都想要的。

### 上下文切换

Go 调度器需要有明确定义的用户空间事件，这些事件发生在要切换上下文的代码中的安全点上。这些事件和安全点在函数调用中表现出来。函数调用对于 Go 调度器的运行状况是至关重要的。现在（使用 Go 1.11或更低版本），如果你运行任何未进行函数调用的**[紧凑循环](https://en.wiktionary.org/wiki/tight_loop)**，你会导致调度器和垃圾回收有延迟。让函数调用在合理的时间范围内发生是至关重要的。

注意：在 Go 1.12 版本中有一个提议被接受了，它可以使 Go 调度器使用非协作抢占技术，以允许抢占紧密循环。

在 Go 程序中有四类事件，它们允许调度器做出调度决策：

* 使用关键字 go
* 垃圾回收
* 系统调用
* 同步和**[编配](https://zh.wikipedia.org/zh-hans/%E7%BC%96%E9%85%8D_(%E8%AE%A1%E7%AE%97%E6%9C%BA))**

#### **使用关键字** go

关键字 go 是用来创建 Goroutines 的。一旦创建了新的 Goroutine，它就为调度器做出调度决策提供了机会。

#### 垃圾回收

由于 GC 使用自己的 Goroutine 运行，所以这些 Goroutine 需要在 M 上运行的时间片。这会导致 GC 产生大量的调度混乱。但是，调度程序非常聪明地了解 Goroutine 正在做什么，它将智能地做出一些决策。

#### 系统调用

如果 Goroutine 进行系统调用，那么会导致这个 Goroutine 阻塞当前**M**，有时调度器能够将 Goroutine 从**M**换出并将新的 Goroutine 换入。然而，有时需要新的**M**继续执行在**P**中排队的 Goroutines。这是如何工作的将在下一节中更详细地解释。

#### 同步和编配

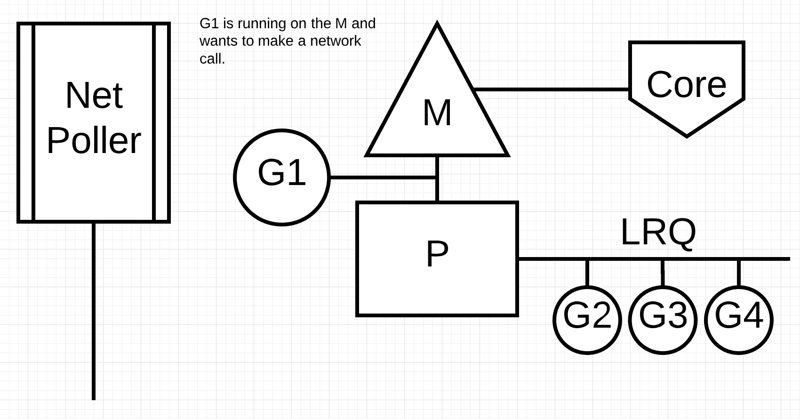
如果**原子、互斥量或通道操作调用**将导致 Goroutine 阻塞，调度器可以将之切换到一个新的 Goroutine 去运行。一旦 Goroutine 可以再次运行，它就可以重新排队，并最终在**M**上切换回来。

### 异步系统调用

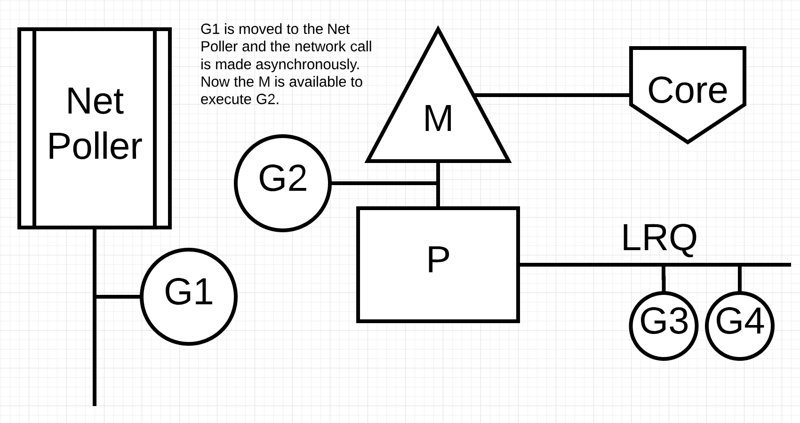
当你的操作系统能够异步处理系统调用时，可以使用称为网络轮询器的东西来更有效地处理系统调用。这是通过在这些操作系统中使用 kqueue（MacOS），epoll（Linux）或 iocp（Windows）来实现的。

基于网络的系统调用可以由我们今天使用的许多操作系统异步处理。这就是为什么我管它叫网络轮询器，因为它的主要用途是处理网络操作。通过使用网络轮询器进行网络系统调用，调度器可以防止 Goroutine 在进行这些系统调用时阻塞**M**。这可以让**M**执行**P**的 LRQ 中其他的 Goroutines，而不需要创建新的**M**。有助于减少操作系统上的调度负载。

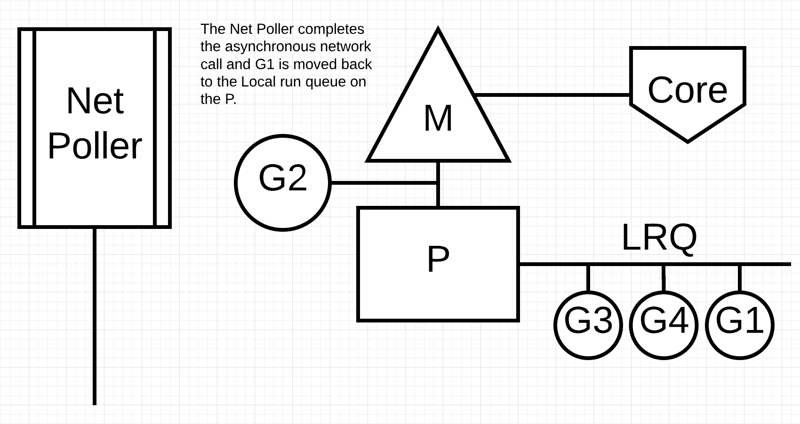
下图展示它的工作原理：**G1**正在**M**上执行，还有 3 个 Goroutine 在 LRQ 上等待执行。网络轮询器空闲着，什么都没干。



接下来，情况发生了变化：**G1**想要进行网络系统调用，因此它被移动到网络轮询器并且处理异步网络系统调用。然后，**M**可以从 LRQ 执行另外的 Goroutine。此时，**G2**就被上下文切换到**M**上了。



最后：异步网络系统调用由网络轮询器完成，**G1**被移回到**P**的 LRQ 中。一旦**G1**可以在**M**上进行上下文切换，它负责的 Go 相关代码就可以再次执行。这里的最大优势是，执行网络系统调用不需要额外的**M**。网络轮询器使用系统线程，它时刻处理一个有效的事件循环。

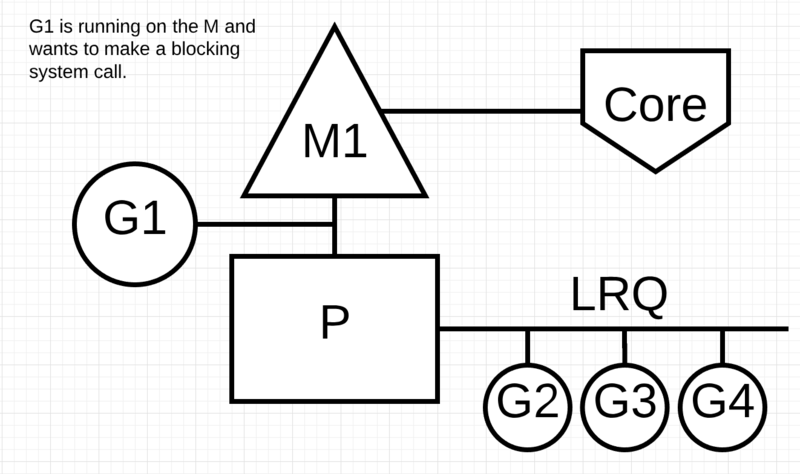


### 同步系统调用

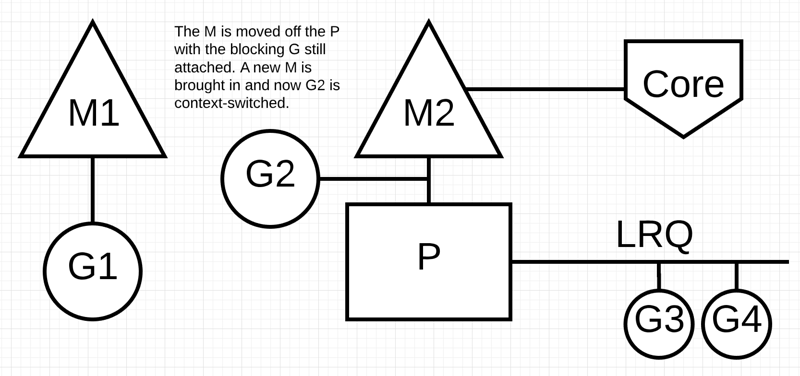
如果 Goroutine 要执行同步的系统调用，会发生什么？在这种情况下，网络轮询器无法使用，而进行系统调用的 Goroutine 将阻塞当前**M**。这是不幸的，但是没有办法防止这种情况发生。需要同步进行的系统调用的一个例子是基于文件的系统调用。如果你正在使用 CGO，则可能还有其他情况，调用 C 函数也会阻塞**M**。

注意：Windows 操作系统确实能够异步进行基于文件的系统调用。从技术上讲，在 Windows 上运行时，可以使用网络轮询器。

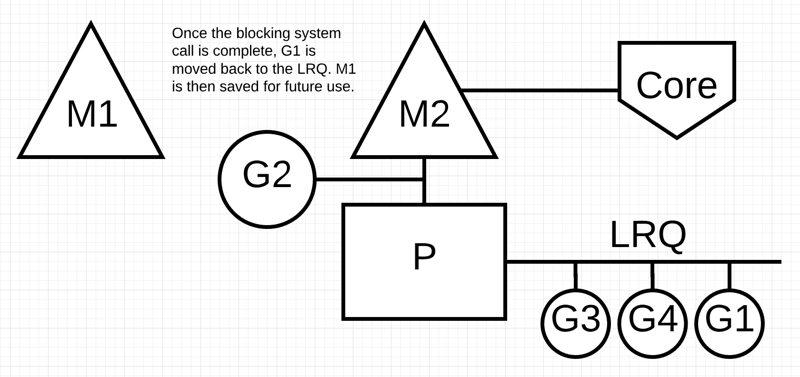
让我们来看看同步系统调用（如文件I/O）会导致**M**阻塞的情况：**G1**将进行同步系统调用以阻塞**M1**。



调度器介入后：识别出**G1**已导致**M1**阻塞，此时，调度器将**M1**与**P**分离，同时也将**G1**带走。然后调度器引入新的**M2**来服务**P**。此时，可以从 LRQ 中选择**G2**并在**M2**上进行上下文切换。



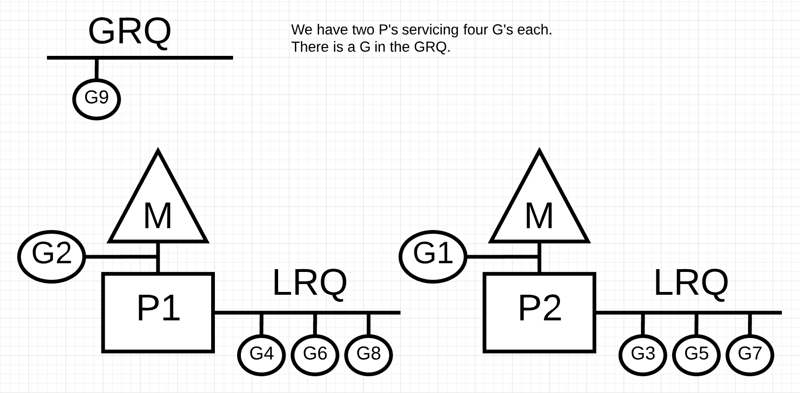
阻塞的系统调用完成后：**G1**可以移回 LRQ 并再次由**P**执行。如果这种情况需要再次发生，M1将被放在旁边以备将来使用。



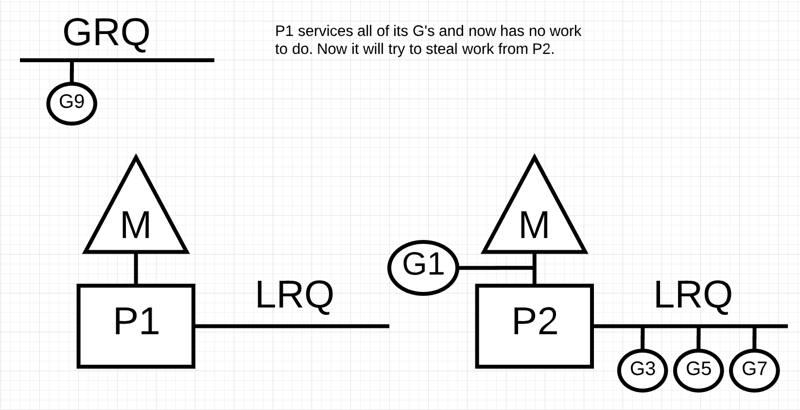
### 任务窃取（负载均衡思想）

调度器的另一个方面是它是一个任务窃取的调度器。这有助于在一些领域保持高效率的调度。首先，你最不希望的事情是**M**进入等待状态，因为一旦发生这种情况，操作系统就会将**M**从内核切换出去。这意味着**P**无法完成任何工作，即使有 Goroutine 处于可运行状态也不行，直到一个**M**被上下文切换回核心。任务窃取还有助于平衡所有**P**的 Goroutines 数量，这样工作就能更好地分配和更有效地完成。

看下面的一个例子：这是一个多线程的 Go 程序，其中有两个**P**，每个**P**都服务着四个 Goroutine，另在 GRQ 中还有一个单独的 Goroutine。如果其中一个**P**的所有 Goroutines 很快就执行完了会发生什么？



如你所见：**P1**的 Goroutines 都执行完了。但是还有 Goroutines 处于可运行状态，在 GRQ 中有，在**P2**的 LRQ 中也有。  
这时**P1**就需要窃取任务。



窃取的规则在这里定义了：**[https://golang.org/src/runtim...](https://golang.org/src/runtime/proc.go)**

if gp == nil {

// 1/61的概率检查一下全局可运行队列，以确保公平。否则，两个 goroutine 就可以通过不断地相互替换来完全占据本地运行队列。

if \_g\_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {

lock(&sched.lock)

gp = globrunqget(\_g\_.m.p.ptr(), 1)

unlock(&sched.lock)

}

}

if gp == nil {

gp, inheritTime = runqget(\_g\_.m.p.ptr())

if gp != nil && \_g\_.m.spinning {

throw("schedule: spinning with local work")

}

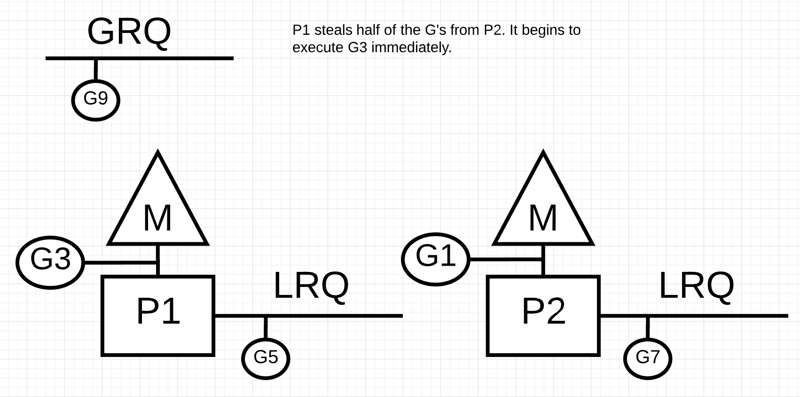
}

if gp == nil {

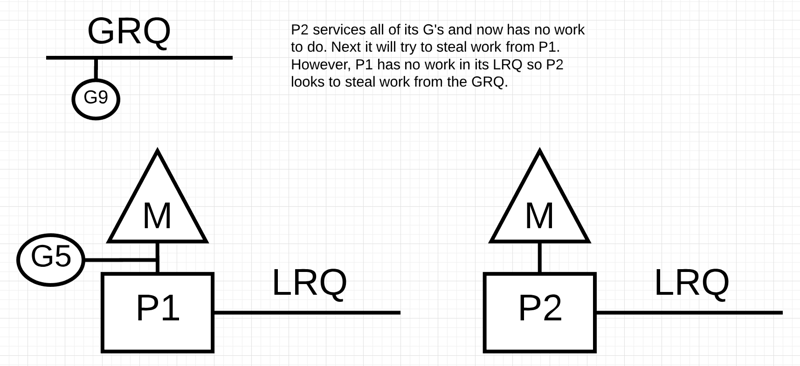
gp, inheritTime = findrunnable()

}

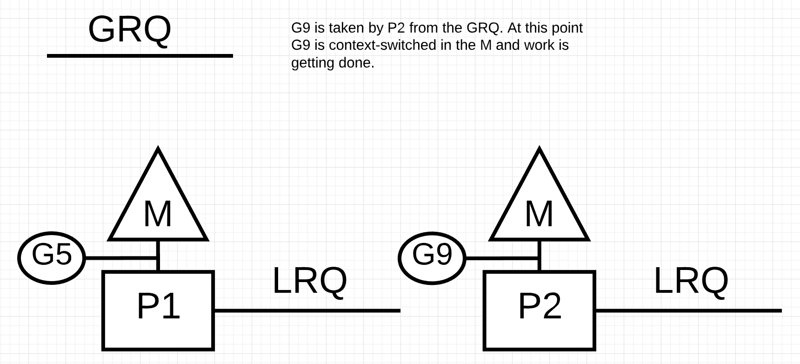
根据规则，**P1**将窃取**P2**中一半的 Goroutines，窃取完成后的样子如下：



我们再来看一种情况，如果**P2**完成了对所有 Goroutine 的服务，而**P1**的 LRQ 也什么都没有，会发生什么?



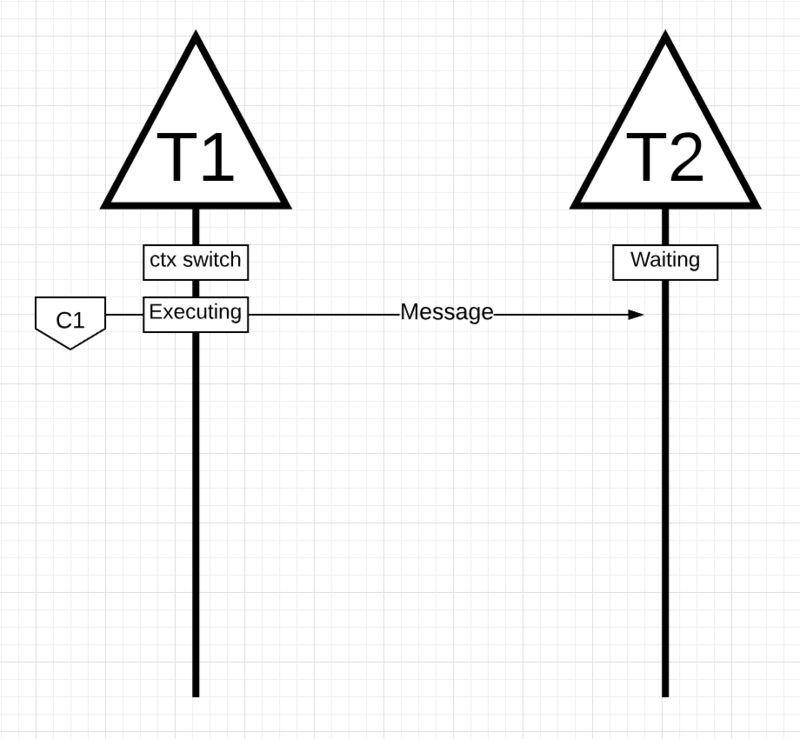
**P2**完成了所有任务，现在需要窃取一些。首先，它将查看**P1**的 LRQ，但找不到任何 Goroutines。接下来，它将查看 GRQ。  
在那里它会找到**G9**，**P2**从 GRQ 手中抢走了**G9**并开始执行。以上任务窃取的好处在于它使**M**不会闲着。在窃取任务时，**M**是自旋的。这种自旋还有其他的好处，可以参考 **[work-stealing](https://rakyll.org/scheduler/)** 。



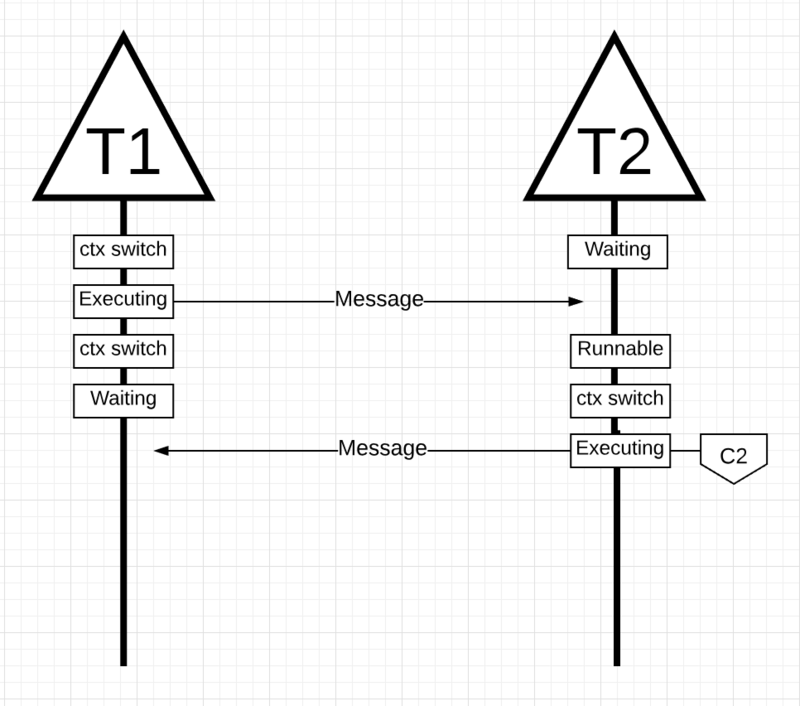
### 实例

有了相应的机制和语义，我将向你展示如何将所有这些结合在一起，以便 Go 调度程序能够执行更多的工作。设想一个用 C 编写的多线程应用程序，其中程序管理两个操作系统线程，这两个线程相互传递消息。

下面有两个线程，线程 T1 在内核 C1 上进行上下文切换，并且正在运行中，这允许 T1 将其消息发送到 T2。

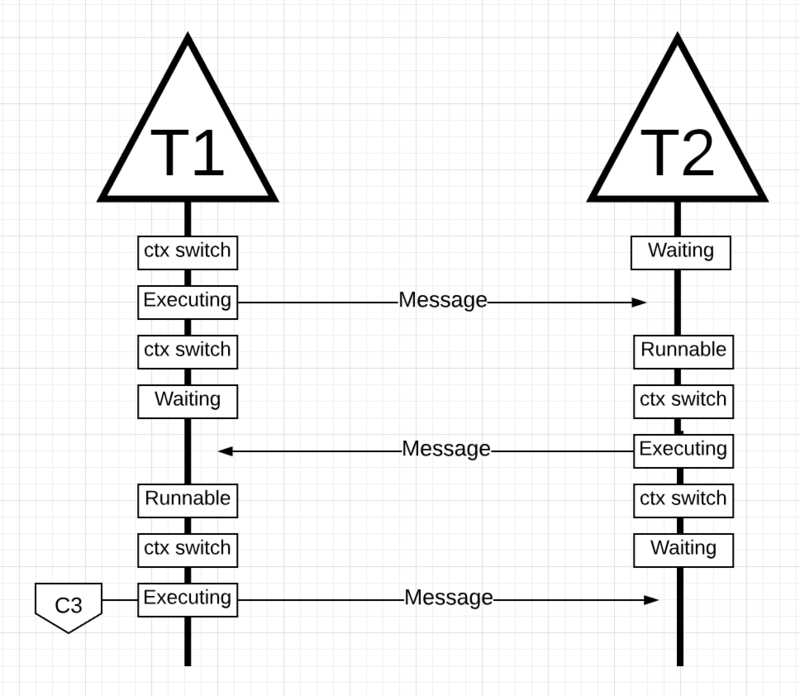


当 T1 发送完消息，它需要等待响应。这将导致 T1 从 C1 上下文换出并进入等待状态。  
当 T2 收到有关该消息的通知，它就会进入可运行状态。  
现在操作系统可以执行上下文切换并让 T2 在一个核心上执行，而这个核心恰好是 C2。接下来，T2 处理消息并将新消息发送回 T1。

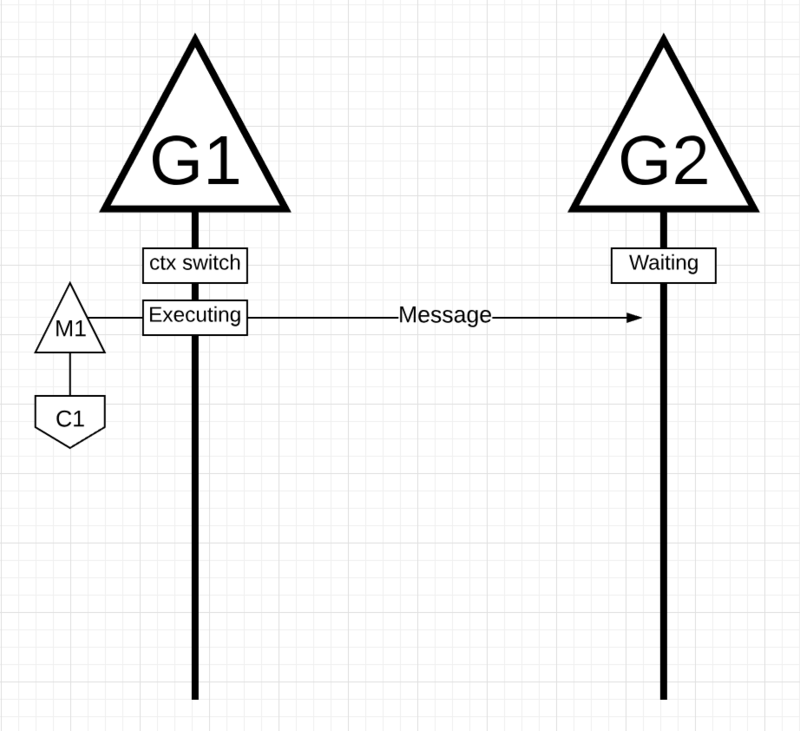


然后，T2 的消息被 T1 接收，线程上下文切换再次发生。现在，T2 从运行中状态切换到等待状态，T1 从等待状态切换到可运行状态，再被执行变为运行中状态，这允许它处理并发回新消息。

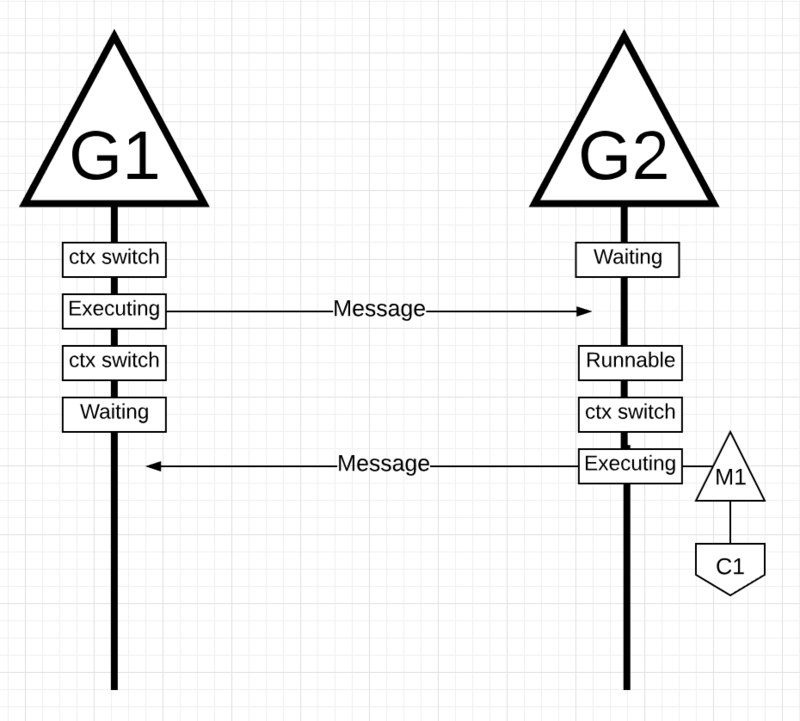
所有这些上下文切换和状态更改都需要时间来执行，这限制了工作的完成速度。  
由于每个上下文切换可能会产生 50 纳秒的延迟，并且理想情况下硬件每纳秒执行 12 条指令，因此你会看到有差不多 600 条指令，在上下文切换期间被停滞掉了。并且由于这些线程也在不同的内核之间跳跃，因 **[cache-line](https://en.wikipedia.org/wiki/CPU_cache)** 未命中引起额外延迟的可能性也很高。



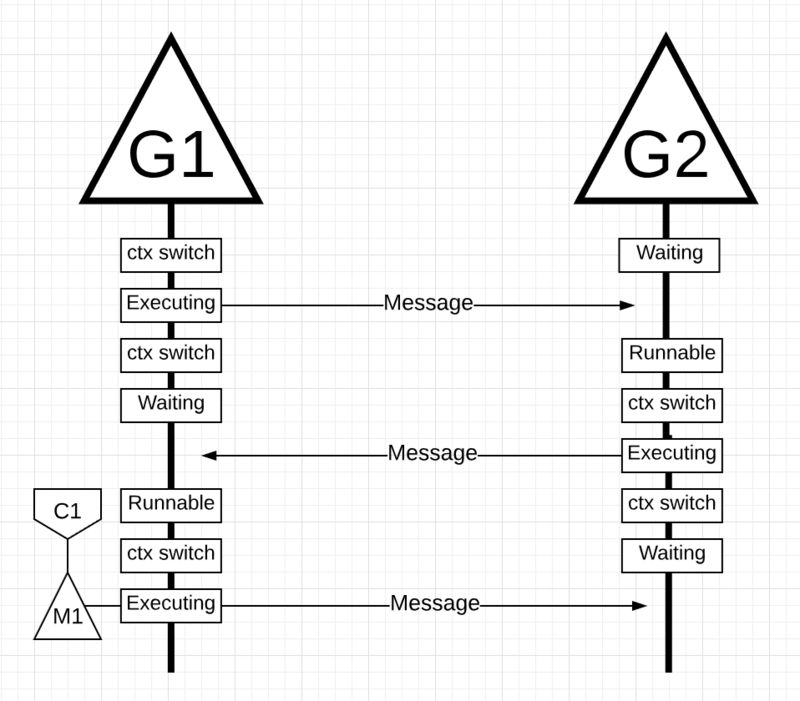
下面我们还用这个例子，来看看 Goroutine 和 Go 调度器是怎么工作的：  
有两个goroutine，它们彼此协调，来回传递消息。**G1**在**M1**上进行上下文切换，而**M1**恰好运行在**C1**上，这允许**G1**执行它的工作。即向**G2**发送消息。



**G1**发送完消息后，需要等待响应。**M1**就会把**G1**换出并使之进入等待状态。一旦**G2**得到消息，它就进入可运行状态。现在 Go 调度器可以执行上下文切换，让**G2**在**M1**上执行，**M1**仍然在**C1**上运行。接下来，**G2**处理消息并将新消息发送回**G1**。



当**G2**发送的消息被**G1**接收时，上下文切换再次发生。现在**G2**从运行中状态切换到等待状态，**G1**从等待状态切换到可运行状态，最后返回到执行状态，这允许它处理和发送一个新的消息。



表面上看起来没有什么不同。无论使用线程还是 Goroutine，都会发生相同的上下文切换和状态变更。然而，使用线程和 Goroutine 之间有一个主要区别：  
****在使用 Goroutine 的情况下，会复用同一个系统线程和核心。这意味着，从操作系统的角度来看，操作系统线程永远不会进入等待状态。因此，在使用系统线程时的开销在使用 Goroutine 时就不存在了。****

基本上，Go 已经在操作系统级别将 IO-Bound 类型的工作转换为 CPU-Bound 类型。由于所有的上下文切换都是在应用程序级别进行的，所以在使用线程时，每个上下文切换(平均)不至于迟滞 600 条指令。该调度程序还有助于提高 cache-line 效率和 NUMA。在 Go 中，随着时间的推移，可以完成更多的工作，因为 Go 调度器尝试使用更少的线程，在每个线程上做更多的工作，这有助于减少操作系统和硬件的负载。

### 结论

Go 调度器在设计中考虑到复杂的操作系统和硬件的工作方式，真是令人惊叹。在操作系统级别将 IO-Bound 类型的工作转换为 CPU-Bound 类型的能力是我们在利用更多 CPU 的过程中获得巨大成功的地方。这就是为什么不需要比虚拟核心更多的操作系统线程的原因。你可以合理地期望每个虚拟内核只有一个系统线程来完成所有工作(CPU和IO)。对于网络应用程序和其他不会阻塞操作系统线程的系统调用的应用程序来说，这样做是可能的。

作为一个开发人员，你当然需要知道程序在运行中做了什么。你不可能创建无限数量的 Goroutine ，并期待惊人的性能。越少越好，但是通过了解这些 Go 调度器的语义，您可以做出更好的工程决策。

## 第三部分-并发

### 介绍

每当我解决一个问题时，尤其是一个新问题时，我刚开始并不会考虑并发是否合适的问题。我会先寻找一些解决方案来确保它正常工作。然后保证代码的可读性，在技术审阅之后，我才会开始提出并发性是否合理及实用的问题。对于并发性，有时候它是一个好东西，有时候却未必是。

在本系列文章的**[第一部分](https://studygolang.com/articles/14264)**，我解释了操作系统调度器的机制和语义，如果你有计划编写多线程代码，我认为这方面知识很重要。在本系列文章的**[第二部分](https://studygolang.com/articles/15316)**，我解释了 Go 中的调度器背后的机制和语义，我相信这对于理解如何在 Go 中编写并发代码是至关重要的。在本文中，我会将操作系统和 Go 调度器的机制和语义结合在一起，以便更深入地理解什么是并发，什么不是并发。

本文的目标是：

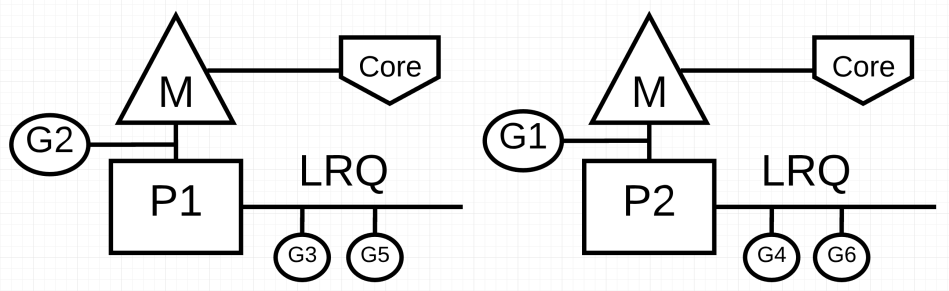
* 提供语义上的指导，来考虑当前的工作负载是否适合使用并发特性。
* 向你展示如何改变不同工作负载下的语义，来更改你想要做出的工程上的决策。

### 什么是并发

并发意味着“无序”执行。拿到一组本来会有序执行的指令，然后使用无序的执行方法，最终却能得到相同的结果。所以摆在你面前的问题非常明显：无序执行会增加一些“价值”，当我在说“价值”时，我实际的意思是复杂度成本的增加，能够带来足够的性能上的收益。这取决于你具体的问题，有些时候无序执行可能无法实现或者根本没有价值。

再者，理解**[并发并不等同于并行](https://blog.golang.org/concurrency-is-not-parallelism)** 至关重要。并行意味着同时执行两个或多个指令。并行的概念和并发是完全不同的。只有当你拥有至少两个操作系统（OS) 或硬件线程，并且至少需要两个 Goroutines 时才能实现真正的并行，每个 Goroutine 在各自的操作系统 / 硬件线程上独立地执行指令。

配图 1：并发和并行



在配图 1 中，你会看到包含两个逻辑处理器（P）的关系图，每个逻辑处理器的独立操作系统线程（M) 附着到机器上的独立硬件线程（核心）上。同时，你也可以看到有两个 Goroutines（G1 和 G2) 在并行地运行，同一时间在它们各自的操作系统 / 硬件线程上执行它们的指令。在每个逻辑处理器内部，有三个 Goroutines 轮流共享各自的操作系统线程。所有这些 Goroutines 都是并行运行的，它们的指令执行时是无序的，且会在操作系统线程上共享时间片。

这就是问题所在，有时候在没有并行的基础上进行并发可能会导致应用程序的吞吐量下降。同样有趣的是，有时候我们在并行的基础上同时使用并发也未必会带来性能上的提升。

### 工作负载

如何才能知道”无序执行“可能会产生收益？了解你所处理的工作负载是一个好的开始。在考虑并发时，有两种重要类型的工作负载需要注意：

* **CPU 密集型**：这种工作负载永远不会导致 Goroutines 进入等待的状态，这是一种持续不断的计算工作。例如，计算圆周率π的第 n 位就是一个 CPU 密集型的工作。
* **IO 密集型**：这种工作负载会导致 Goroutines 自然地进入等待的状态。这类的工作负载一般包括通过网络去访问某些资源，或者向操作系统进行系统调用，或者等待某个事件的发生。如果一个 Goroutine 需要读取一个文件，那么这就是 IO 密集型的工作。我个人倾向于将同步事件（互斥锁，原子操作）归为此类，因为这些操作也会导致 Goroutines 进入等待状态（阻塞）。

如果是 CPU 密集型的工作，你可能需要并行来提高并发性能。单线程处理多个 Goroutines 不够高效，因为 Goroutines 作为工作负载的一部分并不会切换至等待状态。如果 Goroutines 的数量多于操作系统线程的数量，也可能会减缓代码的执行速度，因为在操作系统线程上切换 Goroutines 需要考虑延迟成本（切换所花费的时间）。上下文切换会导致一个 "STW" 事件，因为在切换期间，代码并没有被真正的执行到。

如果是 IO 密集型的工作，你可能就不需要并行来提高并发性能。单个操作系统线程可以非常高效地处理多个 Goroutines，因为 Goroutines 会因为其工作性质而自然地进行等待 - 执行这种状态的切换。Goroutines 的数量多于操作系统线程数时，会增加相关作业的执行速度，因为上下文切换导致的延迟开销并不会产生” STW “事件。你的工作会自然地终止，这允许不同的 Goroutine 高效地利用同一个操作系统线程，从而不会让操作系统线程处于空闲状态。

那么该如何计算每个操作系统线程上运行多少个 Goroutines 才能达到最佳吞吐量？如果 Goroutines 太少，会导致更多的线程空闲时间。Goroutines 太多，又会因为上下文切换而产生更高的延迟成本。但是这个问题已经超出了本文的范畴，实际上，这个问题应该由你自己去思考。

现在来看，最重要的事情是通过检查一些代码来巩固你评估工作负载并适当地使用并发性的能力。

### 数字累加

我们不会使用复杂的用例来解释和理解这些语义。首先，我们看下面代码片段中的 add 函数，它执行的是两个整型数字的求和操作。

36 func add(numbers []int) int {

37 var v int

38 for \_, n := range numbers {

39 v += n

40 }

41 return v

42 }

在清单 1 的第 36 行，声明了一个名为 add 的函数用来求一个元素为 int 切片的所有元素之和。在 37 行，它声明了一个名为 v 的变量来表示求和的结果 sum。然后在 38 行，该函数开始迭代切片，并依次累加各个元素。最后在 41 行，该函数返回求和结果。

问题：add 函数是一个适合无序执行的工作负载吗？我相信答案是肯定的。整型的切片集合可以被划分为多个小集合并且可以被并发地处理。一旦所有的集合都求和完成，就可以将各个集合的结果累加至 sum，这样和顺序求和的结果是一致的。

但是，这又会引入一个其他的问题。到底需要划分多少个小集合进行并发处理才能达到最佳的系统吞吐量？想要回答这个问题， 前提是你必须知道在运行的工作负载 add 具体是什么类型。事实上，add 函数是一个 CPU 密集型的作业类型，因为它运行的是纯数学运算，并且不会有其他的问题导致其 Goroutine 会进入等待状态。这也就意味着为每个操作系统线程分配一个 Goroutine 即可以达到最佳的吞吐量。

下面清单 2 是我编写的并发版本的 add

注意：你可以使用多种方式来编写并发版本的 add 函数。不用被我写的特定的实现方式所束缚。如果你有更加好的实现方式，并且愿意分享，那真是极好的。

func addConcurrent(goroutines int, numbers []int) int {

45 var v int64

46 totalNumbers := len(numbers)

47 lastGoroutine := Goroutines - 1

48 stride := totalNumbers / Goroutines

49

50 var wg sync.WaitGroup

51 wg.Add(goroutines)

52

53 for g := 0; g < Goroutines; g++ {

54 Go func(g int) {

55 start := g \* stride

56 end := start + stride

57 if g == lastGoroutine {

58 end = totalNumbers

59 }

60

61 var lv int

62 for \_, n := range numbers[start:end] {

63 lv += n

64 }

65

66 atomic.AddInt64(&v, int64(lv))

67 wg.Done()

68 }(g)

69 }

70

71 wg.Wait()

72

73 return int(v)

74 }

在清单 2 中出现的 addConcurrent 函数是之前 add 函数的并发版本。并发版本使用了 26 行代码，而之前简单的版本只使用了 5 行代码。多了很多代码，所以我只关注需要重点理解的代码。

**48 行**：每个 Goroutine 只需要计算一个唯一的，且更加小的列表。小列表的大小是根据总列表元素个数除以 Goroutines 数量来计算的。

**53 行**：创建了一堆 Goroutines 用来执行累加工作。

**57-59 行**：最后一个 Goroutine 将累加包含了剩余数字的列表，这些数字可能比其它 Goroutines 中的数字更大。

**66 行**：每个小列表的求和运算结果存到最终的 sum 中。

很明显，并发的版本比顺序处理的版本复杂了很多，但这种复杂性有价值吗？回答此问题最好的方式就是创建一个基准测试。在基准测试里，我使用了 1000 万个数字集合，并且关闭了垃圾回收器。同时，该基准测试中也包含了一个顺序求值的版本。

func BenchmarkSequential(b \*testing.B) {

for i := 0; i < b.N; i++ {

add(numbers)

}}

func BenchmarkConcurrent(b \*testing.B) {

for i := 0; i < b.N; i++ {

addConcurrent(runtime.NumCPU(), numbers)

}}

清单 3 展示了基准测试的内容。下面是所有 Goroutines 共用单个操作系统线程的结果。顺序求值版本使用了一个 Goroutine，而并发版本使用了和 CPU 核心数 runtime.NumCPU 一样多的 Goroutines（8 个）。在这个案例中，并发版本并没有利用到并行性。

10 Million Numbers using 8 Goroutines with 1 core2.9 GHz Intel 4 Core i7

Concurrency WITHOUT Parallelism-----------------------------------------------------------------------------

$ GOGC=off Go test -cpu 1 -run none -bench . -benchtime 3s

goos: darwin

goarch: amd64

pkg: Github.com/ardanlabs/gotraining/topics/go/testing/benchmarks/cpu-bound

BenchmarkSequential 1000 5720764 ns/op : ~10% Faster

BenchmarkConcurrent 1000 6387344 ns/op

BenchmarkSequentialAgain 1000 5614666 ns/op : ~13% Faster

BenchmarkConcurrentAgain 1000 6482612 ns/op

注意：在你本地机器上运行基准测试是很复杂的。有很多意外的因素会导致你的测试结果不精确。你需要确保你的机器尽可能地处于空闲状态且多运行几次基准测试。你希望在测试结果中看到一致性，通过测试工具运行两次基准测试，将会使得测试结果达到最一致的状态。

清单 4 中的基准测试显示了顺序求值的版本比并发版本在使用单个操作系统线程时每次操作快了约 10%-13%。这正是我所期望的，因为并发版本在单个操作系统线程上因为上下文切换和 Goroutines 管理而造成了一些额外的开销。

下面的清单是当每个 Goroutine 都有一个独立的操作系统线程为之运行时的结果。顺序求值版本使用了单个 Goroutine，并发版本使用了 8 个（和 runtime.NumCPU 一致）Goroutines。在这种情况下，并发版本开始利用并行特性。

10 Million Numbers using 8 Goroutines with 8 cores2.9 GHz Intel 4 Core i7

Concurrency WITH Parallelism-----------------------------------------------------------------------------

$ GOGC=off Go test -cpu 8 -run none -bench . -benchtime 3s

goos: darwin

goarch: amd64

pkg: Github.com/ardanlabs/gotraining/topics/go/testing/benchmarks/cpu-bound

BenchmarkSequential-8 1000 5910799 ns/op

BenchmarkConcurrent-8 2000 3362643 ns/op : ~43% Faster

BenchmarkSequentialAgain-8 1000 5933444 ns/op

BenchmarkConcurrentAgain-8 2000 3477253 ns/op : ~41% Faster

清单 5 中的基准测试显示了并发版本在每个 Goroutine 使用独立的操作系统线程时，每次操作的耗时比顺序求值版本快了约 %41-%43。这也正是我所期望的，因为所有的 Goroutines 都在并行运行，8 个 Goroutines 同时进行它们各自的工作。

### 排序

理解并非所有的 CPU 密集型的工作都适合并发是非常重要的。当分离工作或者合并结果的操作都非常昂贵时，这基本上是正确的。冒泡排序是解释这类问题的一个比较好的例子。下面的代码是用 Go 实现的冒泡排序算法。

01 package main

02

03 import "fmt"

04

05 func bubbleSort(numbers []int) {

06 n := len(numbers)

07 for i := 0; i < n; i++ {

08 if !sweep(numbers, i) {

09 return

10 }

11 }

12 }

13

14 func sweep(numbers []int, currentPass int) bool {

15 var idx int

16 idxNext := idx + 1

17 n := len(numbers)

18 var swap bool

19

20 for idxNext < (n - currentPass) {

21 a := numbers[idx]

22 b := numbers[idxNext]

23 if a > b {

24 numbers[idx] = b

25 numbers[idxNext] = a

26 swap = true

27 }28 idx++

29 idxNext = idx + 1

30 }31 return swap

32 }3334 func main() {

35 org := []int{1, 3, 2, 4, 8, 6, 7, 2, 3, 0}

36 fmt.Println(org)

37

38 bubbleSort(org)

39 fmt.Println(org)

40 }

清单 6 是一个用 Go 编写的冒泡排序算法。这种排序算法会扫描每次迭代时需要交换值的整数集合。依赖于列表的顺序，在对所有的元素进行排序之前，可能会多次遍历集合。

问题：bubbleSort 函数是适合无序执行的工作负载吗？我相信答案肯定是否定的。整数集合确实可以被划分为多个小列表然后并发地进行排序。但是，当并发排序完成后，并没有很高效的方式将这些小列表整合到一起。

下面的例子是一个并发版本的冒泡排序算法的实现。

01 func bubbleSortConcurrent(goroutines int, numbers []int) {

02 totalNumbers := len(numbers)

03 lastGoroutine := Goroutines - 1

04 stride := totalNumbers / Goroutines

05

06 var wg sync.WaitGroup

07 wg.Add(goroutines)

08

09 for g := 0; g < Goroutines; g++ {

10 Go func(g int) {

11 start := g \* stride

12 end := start + stride

13 if g == lastGoroutine {

14 end = totalNumbers

15 }

16

17 bubbleSort(numbers[start:end])

18 wg.Done()

19 }(g)

20 }

21

22 wg.Wait()

23

24 // Ugh, we have to sort the entire list again.

25 bubbleSort(numbers)

26 }

清单 7 中，函数 bubbleSortConcurrent 是冒泡排序算法的并发版本的实现。它使用了多个 Goroutines 同时对列表的某一段进行排序。但是，你最后得到了一个按块排序后的值列表。如果我们给定一个 36 个整型数字的列表，并将其划分为 12 组，每组两个数字，如果在 25 行没有再次进行排序，将会得到如下所示的结果。

Before:

25 51 15 57 87 10 10 85 90 32 98 53

91 82 84 97 67 37 71 94 26 2 81 79

66 70 93 86 19 81 52 75 85 10 87 49

After:

10 10 15 25 32 51 53 57 85 87 90 98

2 26 37 67 71 79 81 82 84 91 94 97

10 19 49 52 66 70 75 81 85 86 87 93

由于冒泡排序的本质是扫描整个列表，我们在 25 行对 bubbuleSort 的调用已经否定了使用并发特性会产生任何收益的可能性。

### 文件读取

上面我们已经解释了两种 CPU 密集型的工作负载，那么如果是一个 IO 密集型的工作负载呢？如果 Goroutines 能够自然地进出等待状态时，语义是否会有不同？下面让我们看看一个 IO 密集型的工作负载：文件读取，并运行一个文本搜索操作。

第一个版本是顺序版本，该函数名为 find：

42 func find(topic string, docs []string) int {

43 var found int

44 for \_, doc := range docs {

45 items, err := read(doc)

46 if err != nil {

47 continue

48 }

49 for \_, item := range items {

50 if strings.Contains(item.Description, topic) {

51 found++

52 }

53 }

54 }

55 return found

56 }

在清单 9 中，你可以看到一个名为 find 函数，这是一个顺序执行的版本。在 43 行中，我们声明了一个名为 found 变量，该变量用来维护在给定的文档中找到的指定的 topic 的次数。然后在 44 行，我们开始迭代所有文档，并使用 read 函数在第 45 行读取单个文档。最终在第 49-53 行，使用 strings 包中的 Contains 函数来检查指定的 topic 字符串是否被包含在文档中。如果 topic 存在，found 变量会自增计数。

下面是 find 函数中出现的 read 函数的实现。

33 func read(doc string) ([]item, error) {

34 time.Sleep(time.Millisecond) // Simulate blocking disk read.

35 var d document

36 if err := xml.Unmarshal([]byte(file), &d); err != nil {

37 return nil, err

38 }

39 return d.Channel.Items, nil

40 }

清单 10 中的 read 函数刚开始就执行了 time.Sleep(time.MilliSecond) 调用来模拟从磁盘上读取文件而执行的系统调用所产生的延迟。这种延迟的一致性对于准确测量 find 函数的顺序版本和并发版本的差异至关重要。然后我们在第 35-39 行，存储在全局变量中的 mock 的 xml 文档被编码进一个结构体的值以便后续进行处理。最后，在第 39 行返回被处理后的项目的集合。

在完成了顺序版本之后，下面是并发版本。

注意：你可以使用多种方式来编写并发版本的 find 函数。不用被我写的特定的实现方式所束缚。如果你有更加好的实现方式，并且愿意分享，那真是极好的。

58 func findConcurrent(goroutines int, topic string, docs []string) int {

59 var found int64

60

61 ch := make(chan string, len(docs))

62 for \_, doc := range docs {

63 ch <- doc

64 }

65 close(ch)

66

67 var wg sync.WaitGroup

68 wg.Add(goroutines)

69

70 for g := 0; g < Goroutines; g++ {

71 Go func() {

72 var lFound int64

73 for doc := range ch {

74 items, err := read(doc)

75 if err != nil {

76 continue

77 }

78 for \_, item := range items {

79 if strings.Contains(item.Description, topic) {

80 lFound++

81 }

82 }

83 }

84 atomic.AddInt64(&found, lFound)

85 wg.Done()

86 }()

87 }

88

89 wg.Wait()

90

91 return int(found)

92 }

清单 11 中，findConcurrent 函数是之前 find 函数的并发版本。并发的版本使用了 30 多行代码，而顺序版本的代码只有 13 行。我实现并发版本的目的在于控制用来处理未知数量的文档的 Goroutines 的数量。一个基于通道的 Goroutines 池的模式是我的主要实现方式。

上面的清单中有很多的代码，这里我只会关注需要重点理解的代码。

**61-64 行**：创建了一个通道，并填充相关的文档以便后续处理。

**65 行**： 当所有的文档被处理完成后，我们需要关闭通道来通知相关的 Goroutines 终止执行。

**70 行**： 创建了一堆（一池）的 Goroutines。

**73-83 行**： 该池中的每个 Goroutine 都会从通道中接收到一个文档，并将文档读取到内存中，之后检查文档中是否包含指定的 topic。如果匹配成功，那么本地的 found 变量会自增计数。

**84 行**： 将各个 Goroutines 的计数求和，作为最终的计数。

很明显，并发的版本比顺序处理的版本复杂了很多，但这种复杂性有价值吗？回答此问题最好的方式还是创建一个基准测试。在这个基准测试里，我关闭了垃圾回收器并使用了一个拥有 1000 个文档的集合。分别来测试顺序版本的 find 函数和并发版本的 findConcurrent 函数。

func BenchmarkSequential(b \*testing.B) {

for i := 0; i < b.N; i++ {

find("test", docs)

}}

func BenchmarkConcurrent(b \*testing.B) {

for i := 0; i < b.N; i++ {

findConcurrent(runtime.NumCPU(), "test", docs)

}}

清单 12 展示了基准测试函数。下面是当所有 Goroutines 只有单个操作系统线程可用时的结果。顺序版本使用了一个 Goroutine，而并发版本使用了 8 个 Goroutines（这和 runtime.NumCPU 数量一致）。本例中，并发版本并没有利用到并行特性。

10 Thousand Documents using 8 Goroutines with 1 core2.9 GHz Intel 4 Core i7

Concurrency WITHOUT Parallelism-----------------------------------------------------------------------------

$ GOGC=off Go test -cpu 1 -run none -bench . -benchtime 3s

goos: darwin

goarch: amd64

pkg: Github.com/ardanlabs/gotraining/topics/go/testing/benchmarks/io-bound

BenchmarkSequential 3 1483458120 ns/op

BenchmarkConcurrent 20 188941855 ns/op : ~87% Faster

BenchmarkSequentialAgain 2 1502682536 ns/op

BenchmarkConcurrentAgain 20 184037843 ns/op : ~88% Faster

清单 13 中的结果展示了当所有的 Goroutines 只有单个操作系统线程可用时，并发版本比顺序版本每次操作快了约 87%-88%。这也正是我所期望的，因为所有的 Goroutines 都在高效地共享单个操作系统线程。对于在进行 read 操作中的每个 Goroutine，自然而然的上下文切换允许在单个操作系统线程上完成更多的工作。

下面是利用了并行特性的并发版本的基准测试结果。

10 Thousand Documents using 8 Goroutines with 1 core2.9 GHz Intel 4 Core i7

Concurrency WITH Parallelism-----------------------------------------------------------------------------

$ GOGC=off Go test -run none -bench . -benchtime 3s

goos: darwin

goarch: amd64

pkg: Github.com/ardanlabs/gotraining/topics/go/testing/benchmarks/io-bound

BenchmarkSequential-8 3 1490947198 ns/op

BenchmarkConcurrent-8 20 187382200 ns/op : ~88% Faster

BenchmarkSequentialAgain-8 3 1416126029 ns/op

BenchmarkConcurrentAgain-8 20 185965460 ns/op : ~87% Faster

清单 14 中的基准测试表明引入了额外的操作系统线程并不能提供更好的性能。

### 结论

本文的目的是，在需要确定工作负载是否适合并发特性时，提供相关的必须要考虑的语义方面的指导。我尝试提供了不同类型的算法和工作负载的示例，以便于你能看到语义上的差异以及需要考虑的不同的工作决策。

你可以很清楚地看到，IO 密集型的工作负载并不需要并行性来获得性能上的巨大提升。相反，在 CPU 密集型的工作负载中，诸如像冒泡排序这样的算法时，并发特性的使用不仅会增加复杂性，而且没有任何实际的性能优势。确定你的工作负载是否适合并发，然后使用正确的语义来定义你的工作负载是至关重要的。

# Go 中的垃圾回收

这是三篇系列文章的第一篇博文，系列文章提供了 Go 中垃圾回收背后的机制和概念的理解。这篇博文主要介绍回收器的基础概念。

三篇系列文章的索引：

1) **[Go 中的垃圾回收：第一部分 - 概念](https://studygolang.com/articles/21569)**

2) **[Go 中的垃圾回收：第二部分 -GC 追踪](https://studygolang.com/articles/21570)**

3) **[Go 中的垃圾回收：第三部分 - GC 步调](https://studygolang.com/articles/24562)**

## 第一部分-概念

### 简介

垃圾回收器负责跟踪堆内存分配，释放无用的分配内存以及维护在用分配内存。语言如何设计去实现这些行为是很复杂的，但不应该要求应用开发者为了构建软件而去理解细节。而且，对于语言不同版本的 VM 和运行时（runtime），这些细节的实现一直都在发展变化。对于应用开发者而言，重要的是保持一个良好的工作模型，了解垃圾回收器对其语言的行为以及如何在不关心其实现的情况下，对这种行为表示友好。

在 1.12 版本，Go 语言使用了**无分代同步三色标记清除回收器（non-generational concurrent tri-color mark and sweep collector）**。如果想形象化地了解标记清除回收器如何工作，Ken Fox 写了这篇提供了动画的**[好文章](https://spin.atomicobject.com/2014/09/03/visualizing-garbage-collection-algorithms)**。Go 回收器的实现随着发行版而发展变化，所以一旦下一版本发行，任意讲述实现细节的博文将不再准确。

虽然这样，我将在本文中做的分析不会关注实际的实现细节，而是关注你经历到的行为以及你希望在未来几年看到的行为。在这篇文章中，我将和你分享回收器的行为，并解释如何对该行为表示友好，无论当前实现或未来如何变化。这些都会让你成为更好的 Go 开发者。

注意：这里你可以对有关**[垃圾回收器](https://github.com/ardanlabs/gotraining/tree/master/reading" \l "garbage-collection)** 以及 Go 实际的回收器进行扩展阅读

### 堆不是一个容器

我永远不会把堆叫做用来存储或释放值的容器。重要的是，要理解没有线性限制的内存都定义为“堆”。应该认为应用程序进程空间中的保留的任何内存都可用于堆内存分配。无论任何给定的堆内存分配属于虚拟或物理存储都与我们的模型无关。这种理解将帮助您更好地了解垃圾回收器的工作原理。

### 回收器行为

当某次回收开始，回收器经历三个阶段的工作。其中两个阶段引起 Stop The World ( STW ) 的延迟，另外的阶段会产生降低程序吞吐量的延迟。这三个阶段为：

* 标记开始 - STW
* 标记中 - 并发
* 标记结束 - STW

以下为每一个阶段的细分

#### 标记开始 - STW

当回收开始，首先执行的动作是打开写屏障。写屏障的目的是允许回收器在收集过程保持堆上的数据完整性，因为回收器和应用程序的 Goroutine 会并发执行。

为了打开写屏障，必须停止应用运行的所有 Goroutine 。这个动作通常非常快，平均在 10~30 微秒之间。这是指，如果应用程序的 Goroutine 表现正常情况下。

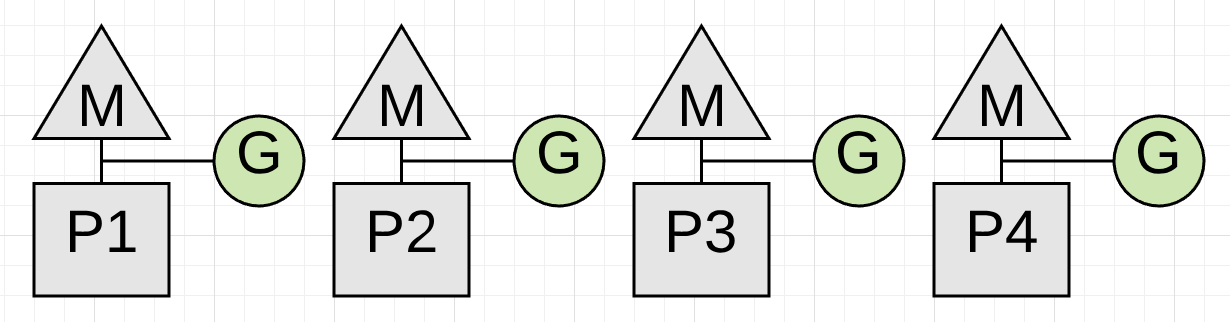
注意：为了更好理解这些调度图，请务必阅读**[Go Scheduler](https://studygolang.com/articles/14264)** 上的系列文章。

图 1 展示了回收之前应用有 4 个 Goroutine 在运行。这 4 个 Goroutine 都应该被停掉，唯一方法就是观察和等待每个 Goroutine 进行函数调用。函数调用保证了 Goroutine 在一个安全的点上被停掉。如果其中一个 Goroutine 没有进行函数调用，但其他的却做了函数调用，这会发生什么呢？

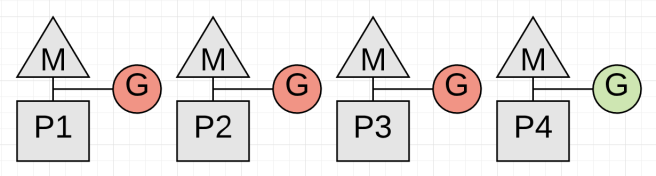


图 2 展示了一个真正的问题。在 P4 上运行的 Goroutine 停下来之前都不会开始回收。然而 P4 的 Goroutine 是不会停止的，因为它正在**[紧密循环](https://github.com/golang/go/issues/10958)** 地进行某些数学运算。

func add(numbers []int) int {

var v int

for \_, n := range numbers {

v += n

}

return v

}

清单 1 展示了运行在 P4 上的 Goroutine 正在执行的代码。goroutine 可能以不合理的大量时间运行从而无法停止，这取决于切片的大小。这种代码会拖延回收的启动，更糟糕的是当回收器在等待时，其他的 P 都不能为任何其他的 Goroutine 服务。gorotine 在一个合理的时间范围内进行函数调用显得极其重要。

注意 : 这是语言团队想要在 1.14 通过加入**[preemotive](https://github.com/golang/go/issues/24543)** 技术到调度中去修复的问题

#### 标记中 - 并发

一旦开启了写障碍，回收器进入标记阶段。回收器做的第一件事是占用 CPU 可用处理能力的 25%。回收器使用 Gorouitne 去做回收工作，也同样需要应用程序的 Goroutine 使用的 P 和 M（译者注：从此处开始作者将 G 划分了两类，一类是应用程序用于自身工作的 Gourinte ，下文称应用 Goroutine，一类是用于 GC 的 Goroutine，这样会更好理解）。

这意味着对于我们四个线程的 Go 程序，有一个完整的 P 会专门用来进行回收工作。

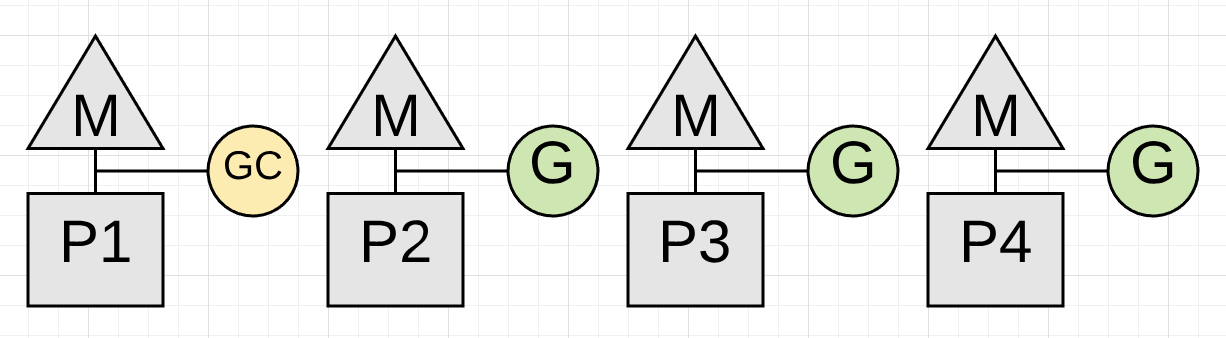


图 3 展示了回收器在回收过程中如何为自身占有 P1。现在回收器开始标记阶段了，标记阶段标记堆内存中仍在使用的值。这个工作先检查栈内所有存活的 Gorouitne，去寻找堆内存的根指针。然后回收器必须从那些根指针遍历堆内存图。当标记工作发生在 P1 上，应用程序可以继续在 P2, P3 和 P4 上同步工作。这意味着回收器的影响被最小化到当前 CPU 处理能力的 25%。

我希望这个事就这样完了然而并没有。如果在收集过程中，确认在 P1 上专用于 GC 的 Goroutine 在堆内存达到上限之前无法完成标记工作，该怎么办？如果 3 个 Goroutine 中，其中一个所进行的应用工作导致回收器无法及时完成 ( 标记工作 ) 又怎么办？ ( 译者注：此处的意思为内存分配过快 )。在这种情况下，新的分配必须放慢速度，特别是从那个 ( 导致标记无法完成的 ) Goroutine。

如果回收器确定它需要减慢分配，它会招募应用 Goroutine 以协助标记工作，这叫做**辅助标记**。任何应用 Goroutine 花费在辅助标记的时间长度与它添加到堆内存中的数据量成正比。辅助标记的一个好处是它有助于更快地完成回收。

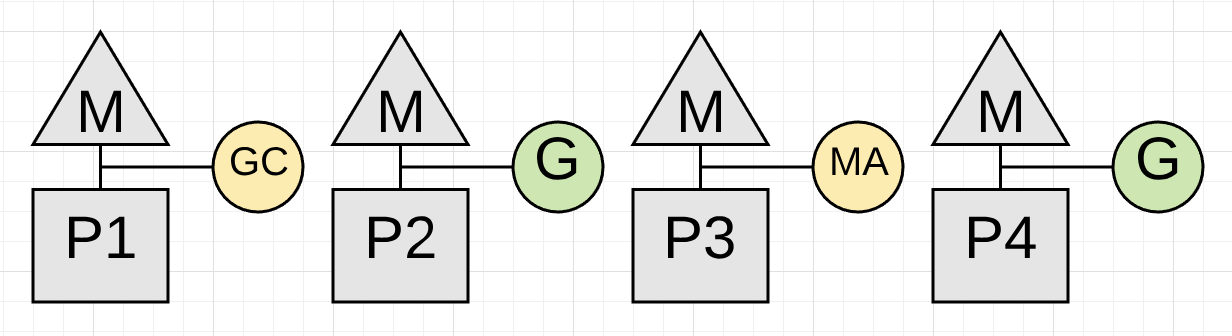


图 4 展示了运行在 P3 上的应用 Goroutine 如何进行辅助标记来帮助回收工作的。希望其他的应用 Goroutine 不用参与进来。分配动作较多的应用可以看到，大部分运行中的 Goroutine 在回收过程中都进行了少量的辅助标记。

回收器的一个目标是消除对辅助标记的需求。如果每次回收都需要大量的辅助标记才能结束，那么回收器很快就会开始下一次的垃圾回收。为了不那么快进行下一次的回收，努力去减少辅助标记的数量是必要的。

#### 标记结束 - STW

一旦标记工作完成，下阶段就是标记结束了。到这个阶段，写屏障会被停止，各样的清洁工作会被执行，然后计算好下一次的回收目标。在标记阶段，发现自身处于紧密循环的 Goroutine 也会延长这个阶段的时长。

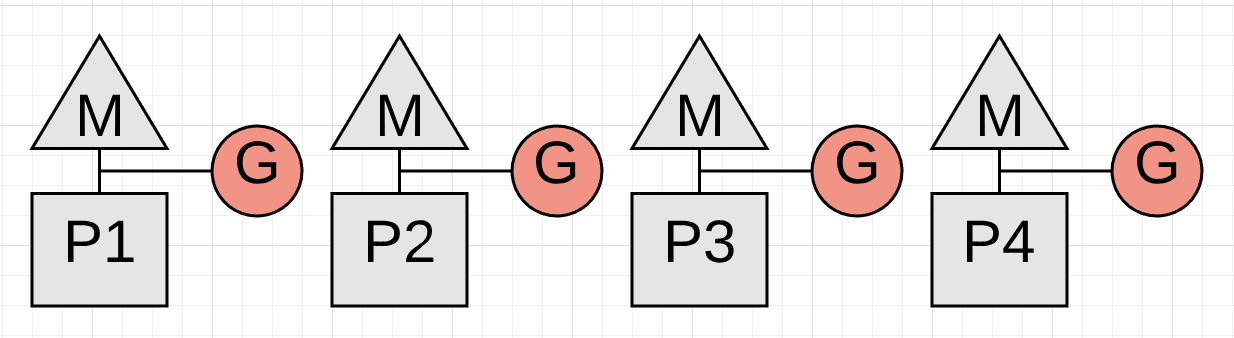


图 5 展示了在标记结束阶段完成时，所有的 Goroutine 如何停止的。这个动作通常平均在 60 到 90 微秒之间。这个阶段可以不需要 STW，但通过使用 STW，代码会更简单，小小的收益抵不上增加的复杂度。

一旦回收完成，每个 P 都能服务于应用 Goroutine，然后应用回到全力运行状态。

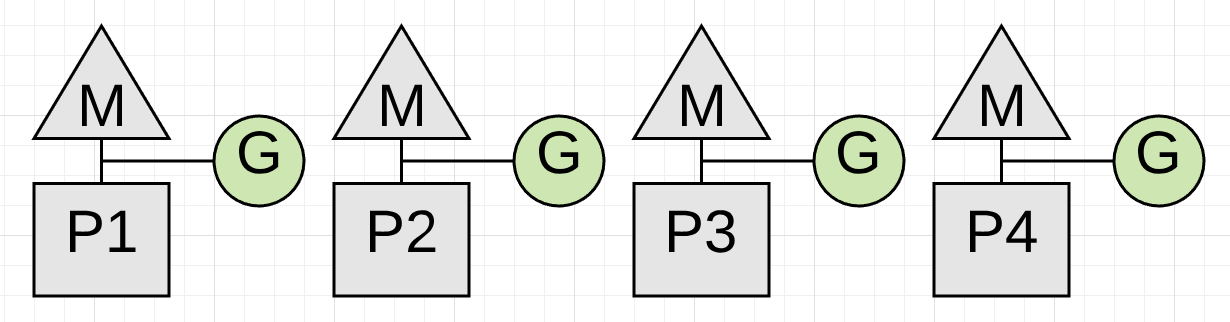


图 6 展示了一旦回收完成，所有可选的 P 如何再次处理应用的工作，应用回到回收开始前的全力运行状态。

### 并发清除

在回收完成之后有另一个叫清除的动作发生。清除是指回收堆内存中，未标记为使用中的值所关联的内存。该动作会在应用程序 Goroutine 尝试分配新值到堆内存时发生。清除的延迟被算到在堆内存中执行分配的成本中，与垃圾回收相关的任何延迟无关。

下面是我机器上的追踪样本，有 12 条硬件线程可用于执行 Gorouitne。

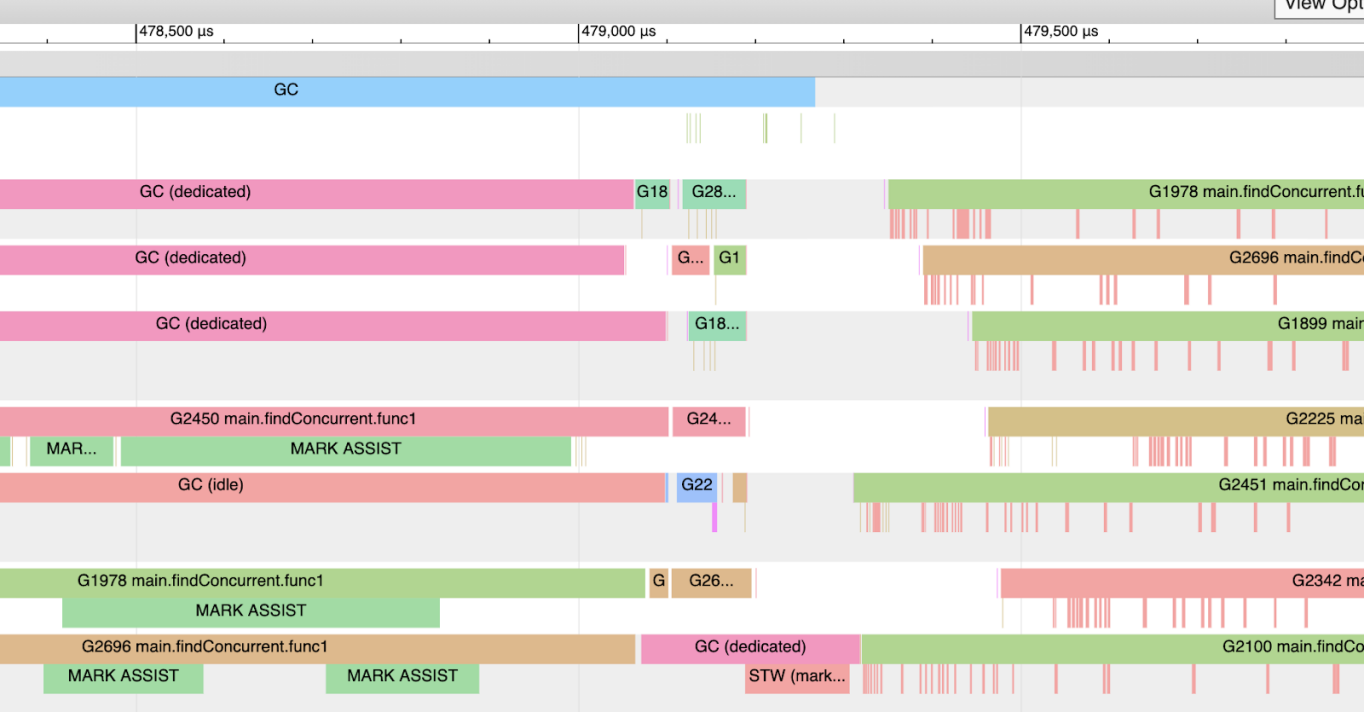


图 7 展示了追踪的部分快照。你可以看到在回收过程中 ( 盯着顶部的蓝色 GC 线 )，12 个 P 中的其中 3 个如何专门用于 GC。你可以看到 Goroutine2450，1978 和 2696 在这段时间进行了数次辅助标记，而不是执行应用的工作。在回收的最后，只有一个 P 用于 GC 并最终执行 STW ( 标记结束 ) 的工作。

在回收完成后，应用程序回到全力运行状态。此外你看到在 Goroutine 下面有一些玫瑰色的线条。

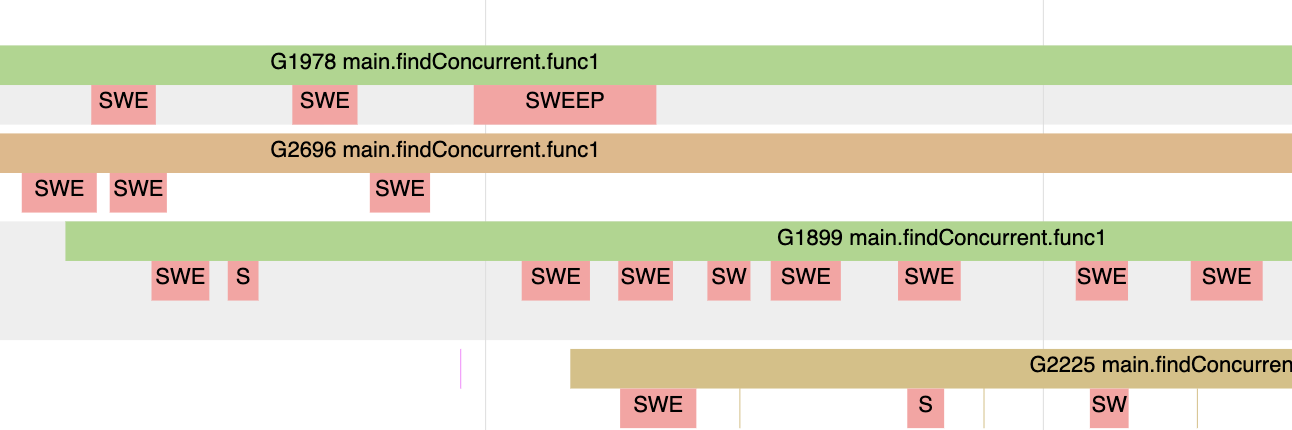


图 8 展示了那些玫瑰色的线条如何代表 Goroutine 不执行应用的工作而进行清除工作的时刻。这些都是 Goroutine 尝试分配新值到堆内存的时刻。

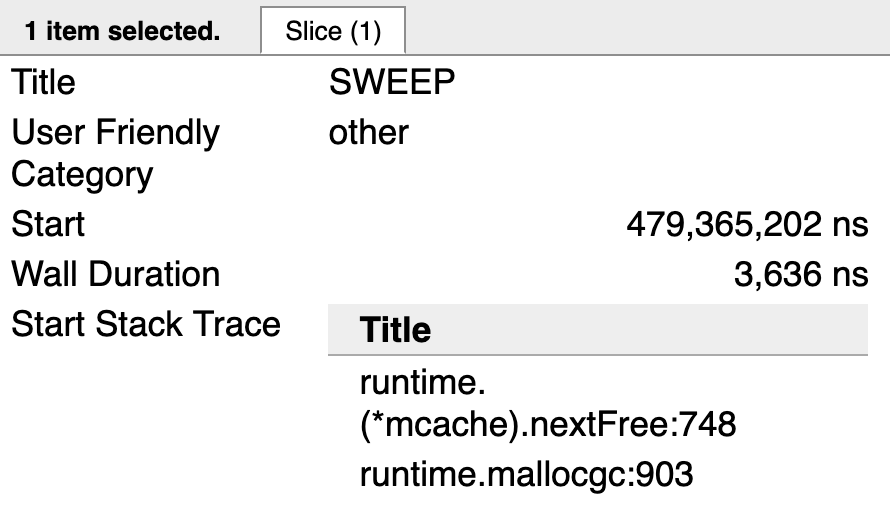


图 9 展示了其中一个进行清除动作的 Gorouitne 最后的栈跟踪情况。runtime.mallocgc 的调用会导致在堆内存分配新值。runtime.(\*mcache).nextFree 的调用引起清除动作。一旦堆内存上没有更多的已分配内存需要回收，就不会再看到 nextFree 的调用。

刚刚描述的回收动作仅仅在回收过程开始和进行中才会发生。配置项 GC 百分比在决定何时启动垃圾回收任务中扮演重要角色。

### GC 百分比

运行过程中有一个配置项叫 GC 百分比，默认值设置为 100。这个值代表了在下次回收开始前能分配多大的堆内存。设置 GC 百分比为 100 意味着，根据回收完成后标记为存活的堆内存量，下一次回收必须在堆内存上添加 100 ％ 以上的新分配 ( 内存 ) 才启动。

举个例子，想象某次回收完成后堆内存有 2MB 存活。( 译者注：后半句话应该是分配 2MB 后 GC 才会开始，作者省了。。。)

注意 : 在这篇博文中堆内存的图不代表使用 Go 的时候的真实情况。Go 中的堆内存通常是碎片化和混乱的，而且没有图像所描绘的那么清晰。这些图在更为易于理解的方式上提供可视化堆内存的方法，这种方式对于你将体验的行为是准确的。

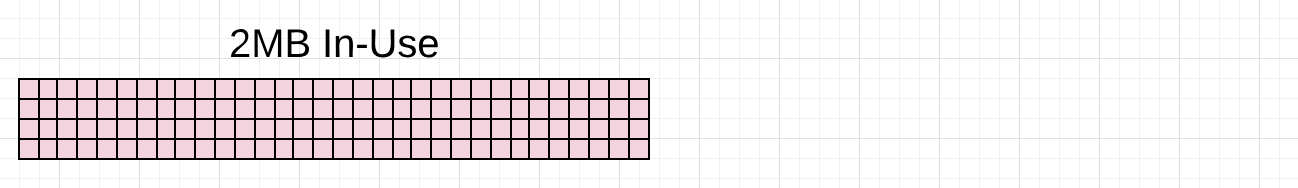


图 10 展示了最后的回收完成后，使用中的 2 MB 堆内存。因为 GC 百分比设置为 100%，下一次回收需要在额外分配 2 MB 的堆内存时才开始，或者在超过 2 MB 之前开始。

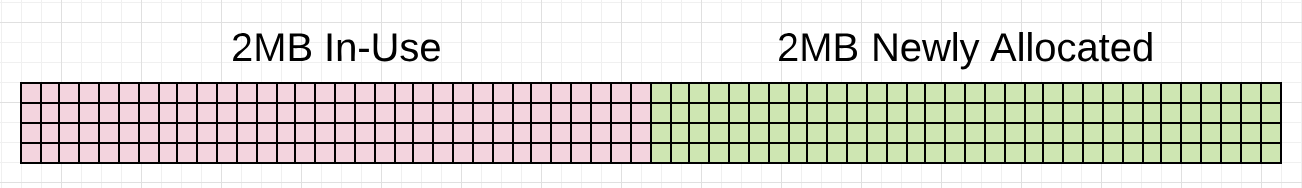


图 11 展示了超过 2MB 堆内存正在使用，这会触发回收。查看该动作所有 ( 细节 ) 的方法是，为每次回收生成 GC 追踪。

### GC 追踪

GC 追踪可以通过在运行任意 Go 应用时包含环境变量 GODEBUG 并指定 gctracec=1 来生成。每次回收发生，运行时会将 GC 追踪信息写到 stderr 中。

GODEBUG=gctrace=1 ./app

gc 1405 @6.068s 11%: 0.058+1.2+0.083 ms clock, 0.70+2.5/1.5/0+0.99 ms CPU, 7->11->6 MB, 10 MB Goal, 12 P

gc 1406 @6.070s 11%: 0.051+1.8+0.076 ms clock, 0.61+2.0/2.5/0+0.91 ms CPU, 8->11->6 MB, 13 MB Goal, 12 P

gc 1407 @6.073s 11%: 0.052+1.8+0.20 ms clock, 0.62+1.5/2.2/0+2.4 ms CPU, 8->14->8 MB, 13 MB Goal, 12 P

清单 2 展示了如何使用 GODEBUG 变量来生成 GC 追踪。清单也展示了运行 Go 应用生成的 3 份追踪信息。

以下是通过查看清单中的第一个 GC 追踪线来拆解 GC 追踪中每个值的含义。

gc 1405 @6.068s 11%: 0.058+1.2+0.083 ms clock, 0.70+2.5/1.5/0+0.99 ms CPU, 7->11->6 MB, 10 MB Goal, 12 P

// General

gc 1404 : 自程序启动以来，1404 的 GC 运行 ( 译者注：此处应当是笔误，联系上文其实是 1405)

@6.068s : 自程序启动至此总共 6s

11% : 到目前为止，可用 CPU 的 11% 被用于 GC

// Wall-Clock

0.058ms : STW : 标记开始，开启写障碍

1.2ms : 并发 : 标记中

0.083ms : STW : 标记结束 - 关闭写障碍并清除

// CPU Time

0.70ms : STW : 标记开始

2.5ms : 并发 : 辅助标记时间 (GC 按照分配执行 )

1.5ms : 并发 : 标记 - 后台 GC 时间

0ms : 并发 : 标记 - 空闲 GC 时间

0.99ms : STW : 标记结束

// Memory

7MB : 标记开始前使用中的堆内存

11MB : 标记完成后使用中的堆内存

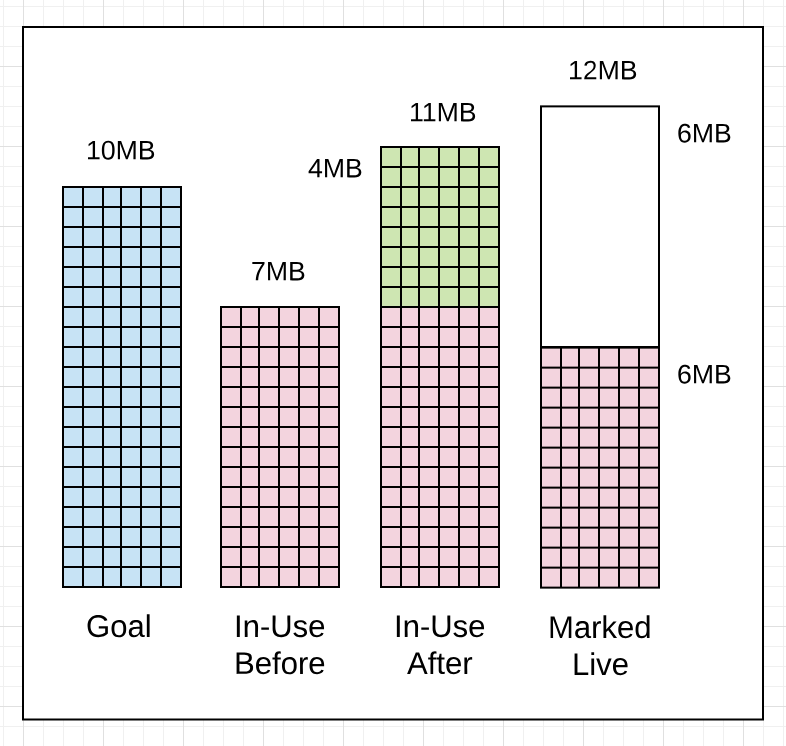
6MB : 标记完成后被标记为存活的堆内存

10MB : 标记完成后使用中的堆内存收集目标

// Threads

12P : 用于运行 Gorouitne 的物理调度器或线程的数量

清单 3 展示了第一条 GC 追踪线的实际数字所代表的含义，按行进行拆解。我后面会谈及这些值中的大部分，但现在只要关注 1405 的 GC 追踪的内存部分。



// Memory

7MB : 标记开始前使用中的堆内存

11MB : 标记完成后使用中的堆内存

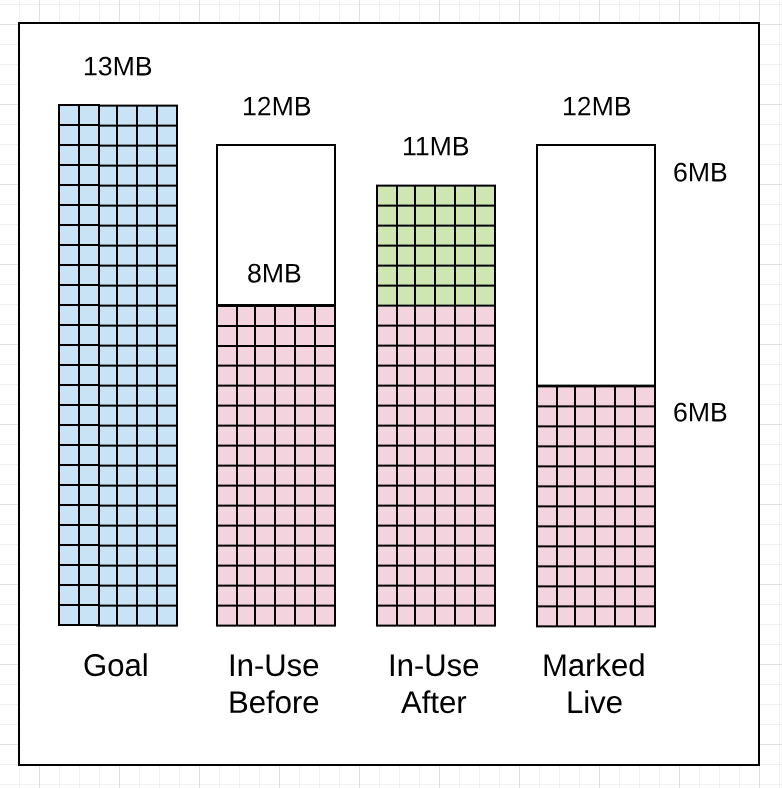
6MB : 标记完成后被标记为存活的堆内存

10MB : 标记完成后使用中的堆内存收集目标

清单 4 中的 GC 追踪线想告诉你的是，在标记工作开始前使用中的堆内存大小为 7 MB。当标记工作完成时，使用中的堆内存大小达到了 11 MB。这意味着在回收过程中出现了额外的 4 MB 内存分配。在标记工作完成后被标记为存活的堆内存大小为 6 MB。这意味着在下次回收开始前应用可以增加使用的堆内存到 12 MB ( 存活堆大小 6 MB 的 100%)。

你可以看到回收器与其目标有 1 MB 的偏差，标记工作完成后正在使用的堆内存量为 11 MB 而不是 10 MB。这没关系，因为目标是根据当前正在使用的堆内存量、标记为存活的堆内存量以及在回收运行时将会发生的其他分配的时间计算情况来计算的。在这种情况下，应用程序做了一些需要在标记之后使用更多的堆内存的事情，而不是像预期那样。

如果查看下一个 GC 跟踪线（1406），你会看到事情在 2 ms 内发生了变化。



gc 1406 @6.070s 11%: 0.051+1.8+0.076 ms clock, 0.61+2.0/2.5/0+0.91 ms CPU, 8->11->6 MB, 13 MB Goal, 12 P

// Memory

8MB : 标记开始前使用中的堆内存

11MB : 标记完成后使用中的堆内存

6MB : 标记开完成后被标记为存活的堆内存

13MB : 标记完成后的使用堆内存收集目标

清单 5 展示了这次回收如何在前一次回收 2 ms 之后开始了，即便使用中的堆内存仅仅达到了 8 MB，而所允许的是 12 MB。这需要特别注意，如果回收器认为早点开始回收会好一点，那么就会提前开始。在这种情况下，它提前开始大概是因为应用在进行大量的分配工作，然后回收器想要降低这次回收的辅助标记的延时。

还有两件事要注意。回收器这次在他的目标之内。标记完成后使用中堆内存的大小是 11 MB 而不是 13 MB，少了 2 MB。标记完成后标记为存活的堆内存大小一样为 6 MB。

附注一点，你可以通过增加 gcpacertrace=1 标志从 GC 追踪获取更多细节，这会让回收器打印更多有关并发步调器的内部状态。

$ export GODEBUG=gctrace=1,gcpacertrace=1 ./app

样本输出：

gc 5 @0.071s 0%: 0.018+0.46+0.071 ms clock, 0.14+0/0.38/0.14+0.56 ms CPU, 29->29->29 MB, 30 MB Goal, 8 P

pacer: sweep done at heap size 29MB; allocated 0MB of spans; swept 3752 pages at +6.183550e-004 pages/byte

pacer: assist ratio=+1.232155e+000 (scan 1 MB in 70->71 MB) workers=2+0

pacer: H\_m\_prev=30488736 h\_t=+2.334071e-001 H\_T=37605024 h\_a=+1.409842e+000 H\_a=73473040 h\_g=+1.000000e+000 H\_g=60977472 u\_a=+2.500000e-001 u\_g=+2.500000e-001 W\_a=308200 Goal Δ =+7.665929e-001 actual Δ =+1.176435e+000 u\_a/u\_g=+1.000000e+000

运行 GC 追踪可以告诉你很多关于程序健康状态以及回收器步调的事情。回收器运行的步调在回收过程中起了重要作用。

### 步调

回收器具有确定何时开始收集的步调算法。算法依赖于回收器用于收集有关正在运行的应用的信息以及应用在堆上分配的压力的反馈循环。压力可以被定义为在指定时间范围内应用分配堆内存的速度。正是压力决定了回收器需要运行的速度。

在回收器开始回收之前，它会计算完成回收所需的时间。然后一旦回收运行，将会对正在运行的应用程序上造成延迟，这将让应用程序的工作变慢。每次回收都会增加应用程序的整体延迟。

一种误解是认为降低回收器步调是改善性能的一种方法。这个想法是，如果你能延缓下次回收的开始，那么你也能延缓它所造成的延时。对回收器友好并不是要降慢其步调。

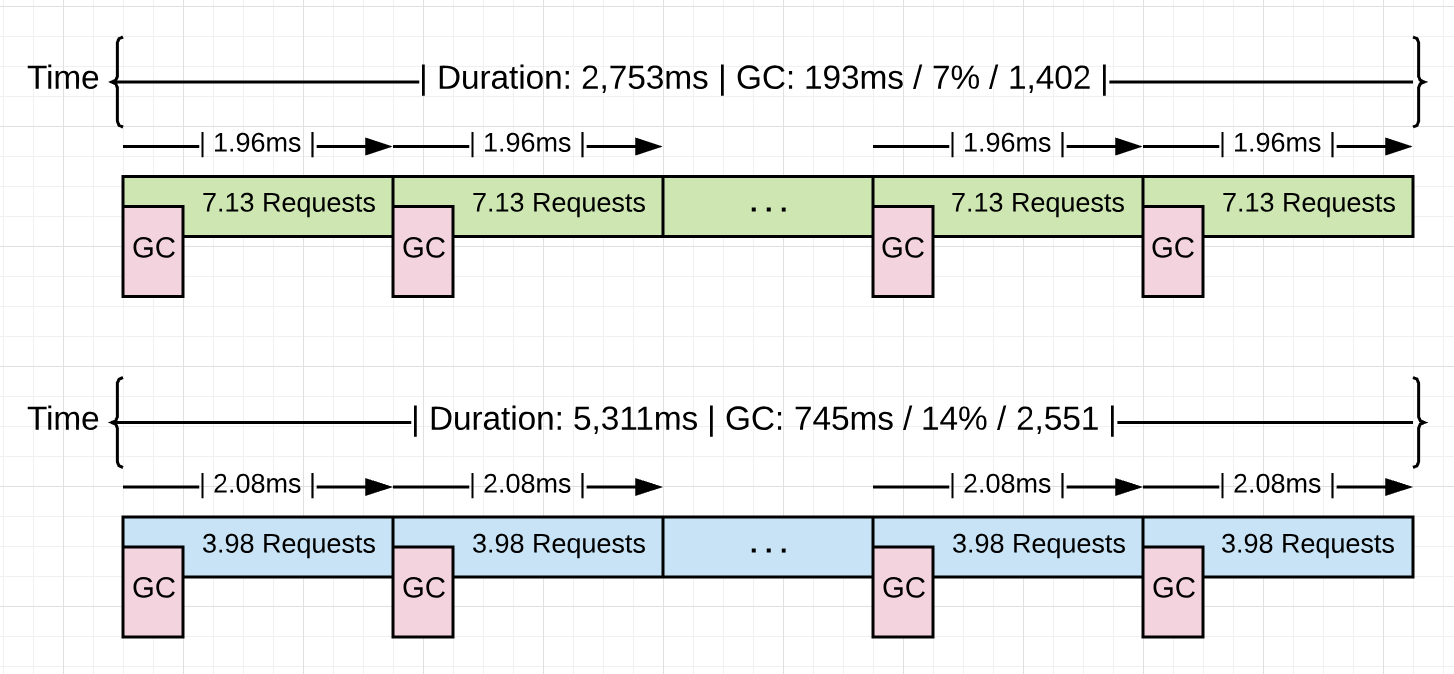
你可以决定改变 GC 百分比的值使其超过 100。这会在下次回收开始前增加分配的堆内存的大小，从而导致回收的步调降低，不要考虑做这种事。



图 14 展示了改变 GC 百分比会如何改变下次回收开始前允许分配的堆内存大小。你可以想象回收器如何因为等待更多的堆内存被使用而变慢。

尝试直接影响回收器的步调对友好对待回收器并无帮助。如果确实希望在每次回收之间或回收期间完成更多的工作，可以减少任意工作添加到堆内存的分配数量或次数。

注意：这个想法也是为了用尽可能小的堆来实现所需的吞吐量。请记住，在云环境中运行时，最小化堆内存等资源的使用非常重要。



清单 15 显示了将在本系列 ( 博文 ) 的下一篇文章中所使用的 Go 应用程序运行的一些统计信息。蓝色版本显示没经过任何优化的应用程序在处理 10 K 请求时的统计信息。绿色版本显示了发现并去掉应用程序 4.48 GB 的非生产性的内存分配后，处理相同的 10 k 请求的统计信息。

看这两个版本的平均收集速度（2.08 ms vs 1.96 ms），它们几乎相同，约为 2.0 ms。这两个版本之间的根本差异是每次回收之间的工作量，从每次回收处理 3.98 增加到 7.13 个请求，以同样的速度完成的工作量增加了 79.1 ％。正如你所看到的，回收并没有随着这些分配的减少而减慢，而是保持不变，（绿色版本的）胜出之处是因为每次回收之间完成了更多工作。

调整回收的步调以延缓其延迟花费并不是你提高应用程序性能的方式。减少回收器运行所需的时间，这反过来就会减少造成的延迟成本。虽然已经对回收器造成的延迟花费进行了解释，但为了清楚起见，让我再总结一下。

### 回收器延时消耗

运行应用中每次回收有两种类型的延时。第一种是窃取（stealing） CPU 的处理能力。窃取 CPU 处理能力的影响是你的应用在回收过程中不能以全力状态运行。因为应用的 Goruinte 正在和回收器的 Goroutine 共享 P，或者正在帮助回收 ( 辅助标记 )。

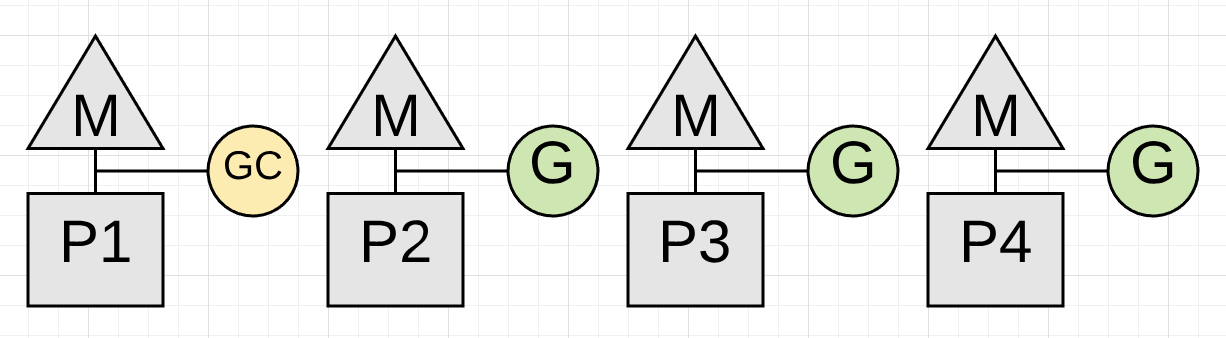


图 16 展示了应用如何仅仅使用 CPU 处理能力的 75% 去工作。这是因为回收器为了回收占用了 P1。

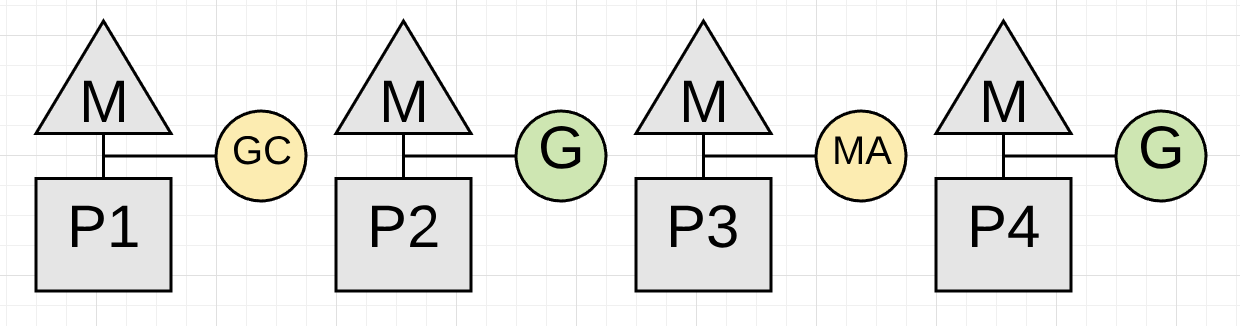


图 17 展示了在这个时刻（通常只有几微秒）应用如何只使用一半的 CPU 处理能力为应用工作。这是因为在 P3 上的 Goroutine 正在进行辅助标记，并且回收器为自己设置了专用的 P1。

注意：标记通常需要 4 个 CPU- 毫秒（CPU-millseconds）处理每 MB 存活的堆 (e.g. 为了评估标记需要运行多少毫秒，用存活的堆大小 MB 然后除以 0.25 乘上 CPU 个数 )。标记实际以 1 MB/ms 运行，但是因为只用了 1/4 的 CPU（译者注：所以是 4 ms 处理 1 MB，也就是开头的 4 个 CPU- 毫秒每 MB）

第二个延时取决于在回收过程中出现的 STW 延迟出现的次数。STW 时间是没有应用程序 Goroutine 执行任意应用程序工作的时间。该应用程序基本上停止了。

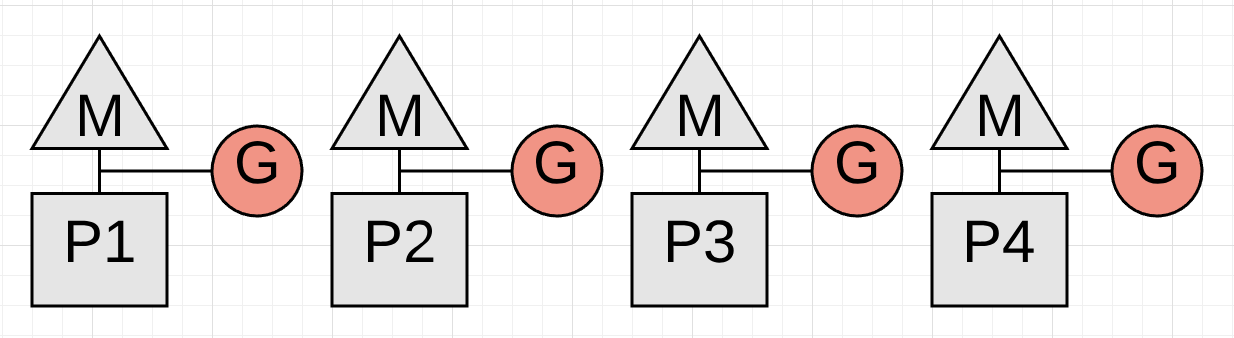


图 18 展示了 STW 延时，这个时候所有 Goroutine 都会停止，这会在每次回收发生两次。如果你的应用健康，回收器可以保持大部分回收过程的总 STW 时间在 100 微秒之内。

你现在已经知道回收器的不同时期，内存如何分配，步调器如何工作，以及回收器在你运行应用中主要出现的不同延时。通过这些知识，你如何对回收器友好的问题终于能解决了。

### 对回收器友好

对回收器表示友好就是降低内存压力。请记住，压力定义为应用在指定时间内分配内存的速度。当压力降低时，因回收器主要引发的延迟就会降低。而 GC 延迟会拖慢你的应用。

能够降低 GC 延迟的方式是，从应用中辨别和去掉不需要的内存分配。可以通过以下几种方式帮助回收器。

帮助回收器：

* 尽可能维护最小化的堆
* 找到最佳的一致步调
* 每次回收保持在目标之内
* 最小化每次回收，STW 以及辅助标记的持续时长

以上所列都能帮助降低在你运行中的程序，主要因回收器造成的延迟大小。这会改善应用的吞吐量表现。我们不需要回收器的步调做任何处理，下面是你可以做的其他事情，以帮助做出更好的工程决策，减少堆上的压力。

### 了解应用程序执行的工作负载的性质

了解工作负载意味着确保使用合理数量的 Goroutine 来完成你的工作。CPU 密集型与 IO 密集型的工作负载不同，需要不同的工程决策。

**<https://studygolang.com/articles/17014>**

### 了解已定义的数据及其在应用程序中的传递方式

了解数据意味着了解你尝试解决的问题。数据语义一致性是维护数据完整性的关键部分，并且允许你在堆栈上选择堆分配时（通过读取代码）知道这件事。

**<https://studygolang.com/articles/12487>**

### 结论

作为 Go 开发者，如果你花时间专注于减少分配，你正在对垃圾回收器表示友好。你不能编写零分配的应用程序，因此重要的是要认识到有效的分配（对应用有助）和无生产力的分配（对应用有害）之间的差异。然后信任垃圾回收器，相信它能保持堆处于健康状态，并使你的应用程序始终如一地运行。

拥有垃圾回收器是一笔很划算的交易，我花费垃圾回收的成本，因而没有内存管理的负担。Go 允许你作为开发人员提高工作效率的同时还可以编写足够快的应用程序。垃圾回收器对实现这一目标起了重要作用。在下一篇文章中，我将向你展示一个示例 Web 应用程序以及如何动手使用工具查看所有这些信息。

## 第二部分-GC追踪

### 介绍

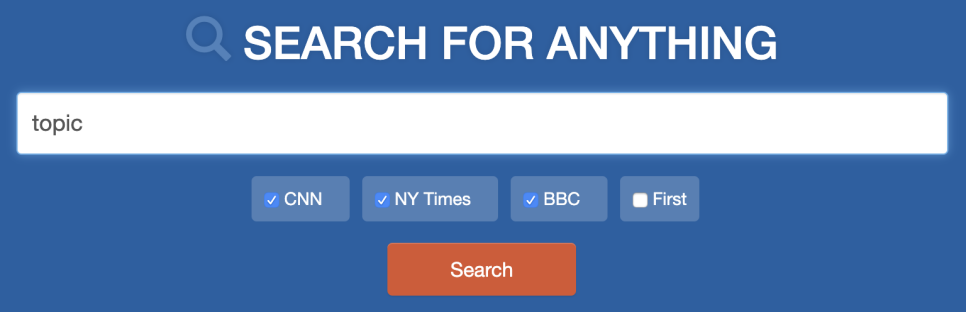
在第一篇文章中，我花了一些时间描述了垃圾回收器的行为并且展示了它对正在运行的应用造成的延迟。我还分享了如何去生成并且解释 GC 追踪，展示了堆中的内存是如何变化的，并且解释了 GC 的不同阶段以及它们是如何影响延迟成本的。

那篇文章得出的最后结论是，降低堆的压力，就会降低延迟成本从而增加应用的性能。我也提出了另外一个观点，通过找到增加任意两次垃圾回收间隔时间的方法，来降低回收器开始的步调不是一个好的策略。一个稳定的步调，即使比较快，也会有利于保持应用的高性能运行

在本篇文章中，我将会带领你运行一个真实的 Web 应用并且向你展示如何生成 GC 追踪和应用的性能概要（profile）文件。然后我会向你展示如何解释这些工具的输出内容，从而找到一个提升应用性能的方法。

### 运行应用

看一下我在 Go 练习中用到的这个 Web 应用



**<https://github.com/ardanlabs/gotraining/tree/master/topics/go/profiling/project>**

图 1 展示了这个 Web 应用的界面，这个应用从不同的新闻提供者那里下载三组 rss 源并且允许用户进行搜索。编译过后，执行该应用

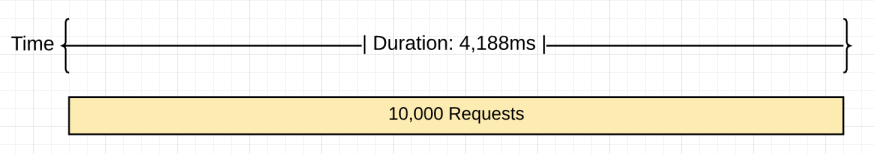
$ go build

$ GOGC=off ./project > /dev/null

清单 1 向我们展示了如何启动应用，并通过设置 GOGC 环境变量为 off 来关闭垃圾回收。日志文件被重定向到了 /dev/null 设备。应用运行的过程中，我们可以发送请求到服务器。

$ hey -m POST -c 100 -n 10000 "http://localhost:5000/search?term=topic&cnn=on&bbc=on&nyt=on"

清单 2 向我们展示了如何利用 hey 工具通过 100 个连接向服务端发送 10k 的请求。当所有的请求都成功发送到服务器时，就会发生接下来的结果



图片 2 形象的展示了在垃圾回收器关闭的情况下处理 10k 数据的过程。处理 10k 的请求共花费了 4188 ms，这样算下来服务器每秒只能处理 2387 个请求。

### 打开垃圾回收器

如果启动这个应用的时候把垃圾回收打开会发生什么呢？

$ GODEBUG=gctrace=1 ./project > /dev/null

清单 3 展示了如何启动应用并打开 GC 追踪。这里移除了 GOGC 变量并且替换为 GODEBUG。GODEBUG 这样设置了之后，运行时就会在每次垃圾回收发生的时候产生一条 GC 追踪。现在又可以向服务器发送同样的 10k 请求了。一旦所有的请求都发送到服务器后，就会产生一些可以用来分析的 GC 追踪信息和 hey 工具产生的信息。

$ GODEBUG=gctrace=1 ./project > /dev/null

gc 3 @3.182s 0%: 0.015+0.59+0.096 ms clock, 0.19+0.10/1.3/3.0+1.1 ms CPU, 4->4->2 MB, 5 MB Goal, 12 P...

gc 2553 @8.452s 14%: 0.004+0.33+0.051 ms clock, 0.056+0.12/0.56/0.94+0.61 ms CPU, 4->4->2 MB, 5 MB Goal, 12 P

清单 4 展示了运行过程中的第三条和最后一条 GC 追踪信息。我没有展示前两个垃圾回收信息是因为负载发送到服务器的时候这两次垃圾回收已经发生了。最后一次垃圾回收显示了它在处理 10k 请求过程中一共发生了 2551 次垃圾回收（减去了前两次，它们没有计算在内）。

下边是追踪信息的具体内容

gc 2553 @8.452s 14%: 0.004+0.33+0.051 ms clock, 0.056+0.12/0.56/0.94+0.61 ms CPU, 4->4->2 MB, 5 MB Goal, 12 P

gc 2553 : The 2553 GC runs since the program started

@8.452s : Eight seconds since the program started14% : Fourteen percent of the available CPU so far has been spent in GC

// wall-clock0.004ms : STW : Write-Barrier - Wait for all Ps to reach a GC safe-point.0.33ms : Concurrent : Marking0.051ms : STW : Mark Term - Write Barrier off and clean up.

// CPU time0.056ms : STW : Write-Barrier0.12ms : Concurrent : Mark - Assist Time (GC performed in line with allocation)0.56ms : Concurrent : Mark - Background GC time0.94ms : Concurrent : Mark - Idle GC time0.61ms : STW : Mark Term

4MB : Heap memory in-use before the Marking started

4MB : Heap memory in-use after the Marking finished

2MB : Heap memory marked as live after the Marking finished

5MB : Collection Goal for heap memory in-use after Marking finished

// Threads

12P : Number of logical processors or threads used to run Goroutines.

清单 5 展示了最后一次垃圾回收的实际数据。幸亏有了 hey 工具，这些是运行的性能结果。

Requests : 10,000------------------------------------------------------

Requests/sec : 1,882 r/s - Hey

Total Duration : 5,311ms - Hey

Percent Time in GC : 14% - GC Trace

Total Collections : 2,551 - GC Trace------------------------------------------------------

Total GC Duration : 744.54ms - (5,311ms \* .14)

Average Pace of GC : ~2.08ms - (5,311ms / 2,551)

Requests/Collection : ~3.98 r/gc - (10,000 / 2,511)

清单 6 展示了执行结果。下边则更形象的向我们展示了发生了什么

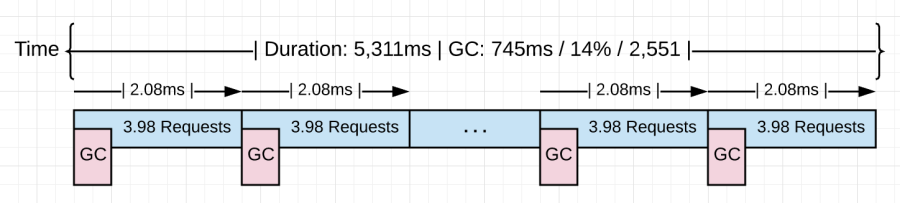


图 3 形象的展示了发生了什么。当垃圾回收打开的时候它必须执行 2.5k 次来处理同样的 10k 请求。平均每 2 ms 开始一次垃圾回收，执行所有的垃圾回收需要增加额外的 1.1 秒的延迟

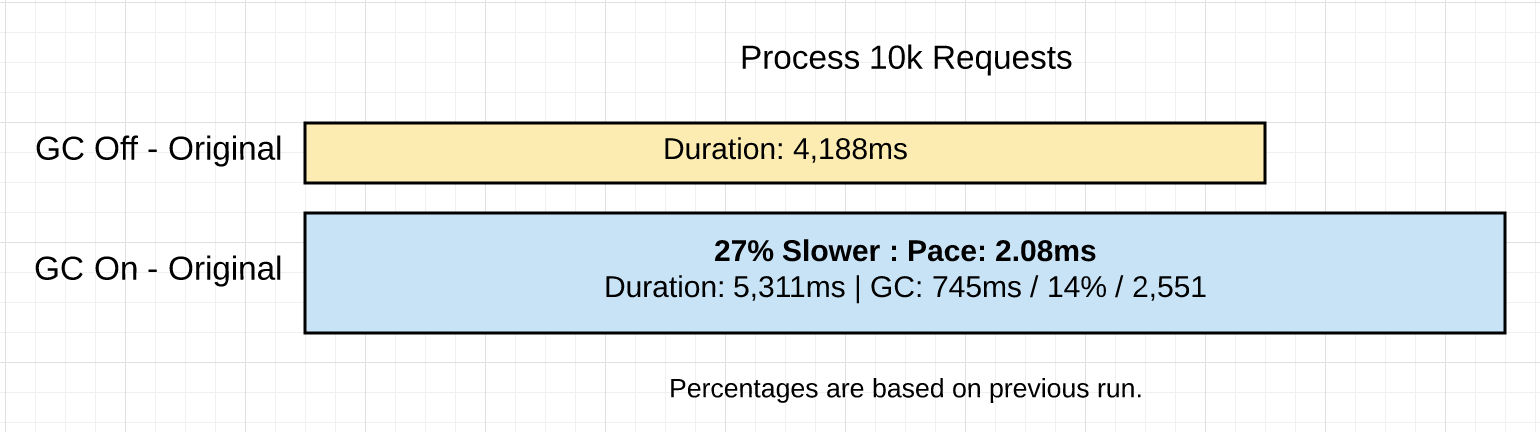


图 4 展示了到现在为止应用两次执行的对比

### 减少分配

获取堆的 profile 并看下有没有可移除的非生产性质的分配将会很有用

go tool pprof http://localhost:5000/debug/pprof/allocs

清单 7 展示了使用 pprof 工具调用 /debug/pprof/allocs 路径（endpoint）来从运行的应用中拉取内存的性能概要（profile）。之所以用那个路径是因为在源程序中添加了以下代码

import \_ "net/http/pprof"

go func() {

http.ListenAndServe("localhost:5000" , http.DefaultServeMux)

}()

清单 8 展示了如何绑定 /debug/pprof/allocs 路径到任何应用中。增加 net/http/pprof 包的导入可以绑定路径到默认的服务器路由。然后调用 http.ListenAndServer 方法并传入 http.DefaultServerMux 常量使该路径可用。

profiler 启动后，就可以用 top 命令查看正在分配内存的前 6 个方法

(pprof) top 6 -cum

Showing nodes accounting for 0.56GB, 5.84% of 9.56GB total

Dropped 80 nodes (cum <= 0.05GB)

Showing top 6 nodes out of 51

flat flat% sum% cum cum%

0 0% 0% 4.96GB 51.90% net/http.(\*conn).serve

0.49GB 5.11% 5.11% 4.93GB 51.55% project/service.handler

0 0% 5.11% 4.93GB 51.55% net/http.(\*ServeMux).ServeHTTP

0 0% 5.11% 4.93GB 51.55% net/http.HandlerFunc.ServeHTTP

0 0% 5.11% 4.93GB 51.55% net/http.serverHandler.ServeHTTP

0.07GB 0.73% 5.84% 4.55GB 47.63% project/search.rssSearch

清单 9 在清单的底部展示了 rssSearch 方法的表现。这个方法到现在共分配了 5.96 GB 中的 4.55 GB。现在是时候使用 list 命令来分析 rssSearch 方法的细节了

(pprof) list rssSearch

Total: 9.56GB

ROUTINE ======================== project/search.rssSearch in project/search/rss.go

71.53MB 4.55GB (flat, cum) 47.63% of Total

. . 117: // Capture the data we need for our results if we find ...

. . 118: for \_, item := range d.Channel.Items {

. 4.48GB 119: if strings.Contains(strings.ToLower(item.Description), strings.ToLower(term)) {

48.53MB 48.53MB 120: results = append(results, Result{

. . 121: Engine: engine,

. . 122: Title: item.Title,

. . 123: Link: item.Link,

. . 124: Content: item.Description,

. . 125: })

清单 10 展示了 list 命令的执行结果。指出了 119 行代码分配了大量的内存

4.48GB 119: if strings.Contains(strings.ToLower(item.Description), strings.ToLower(term)) {

清单 11 展示了出现问题的那行代码。那个方法到现在一共分配了 4.55 GB，仅改行代码就占了 4.48 GB。接下来，是时候审核一下这行代码来看看有什么可以做的。

117 // Capture the data we need for our results if we find the search term.

118 for \_, item := range d.Channel.Items {

119 if strings.Contains(strings.ToLower(item.Description), strings.ToLower(term)) {

120 results = append(results, Result{

121 Engine: engine,

122 Title: item.Title,

123 Link: item.Link,

124 Content: item.Description,

125 })

126 }

127 }

清单 12 展示了那行代码在一个紧密的循环中。对 strings.ToLower 的调用将会产生分配因为创建新的 strings 需要在堆上分配内存。这些 strings.ToLower 的调用是没有必要的，因为这些调用可以在循环外完成。

修改一下 119 行可以移除掉所有的这些内存分配

// Before the code change.if strings.Contains(strings.ToLower(item.Description), strings.ToLower(term)) {

// After the code change.if strings.Contains(item.Description, term) {

注意：你没看到的其他的改动的代码功能是将源放到缓存里之前让描述变为小写。新闻源每 15 分钟缓存一次。使 term 变为小写的调用是在循环外部

清单 13 展示了如何移除了对 strings.ToLower 的调用。用新的改动过的代码重新编译该项目，重新对服务器发起 10 请求。

$ go build

$ GODEBUG=gctrace=1 ./project > /dev/null

gc 3 @6.156s 0%: 0.011+0.72+0.068 ms clock, 0.13+0.21/1.5/3.2+0.82 ms CPU, 4->4->2 MB, 5 MB Goal, 12 P...

gc 1404 @8.808s 7%: 0.005+0.54+0.059 ms clock, 0.060+0.47/0.79/0.25+0.71 ms CPU, 4->5->2 MB, 5 MB Goal, 12 P

清单 14 展示了在代码修改之后处理同样的 10k 请求现在是如何使用了 1402 次垃圾回收的。这些是两次执行的所有结果。

With Extra Allocations Without Extra Allocations======================================================================

Requests : 10,000 Requests : 10,000----------------------------------------------------------------------

Requests/sec : 1,882 r/s Requests/sec : 3,631 r/s

Total Duration : 5,311ms Total Duration : 2,753 ms

Percent Time in GC : 14% Percent Time in GC : 7%

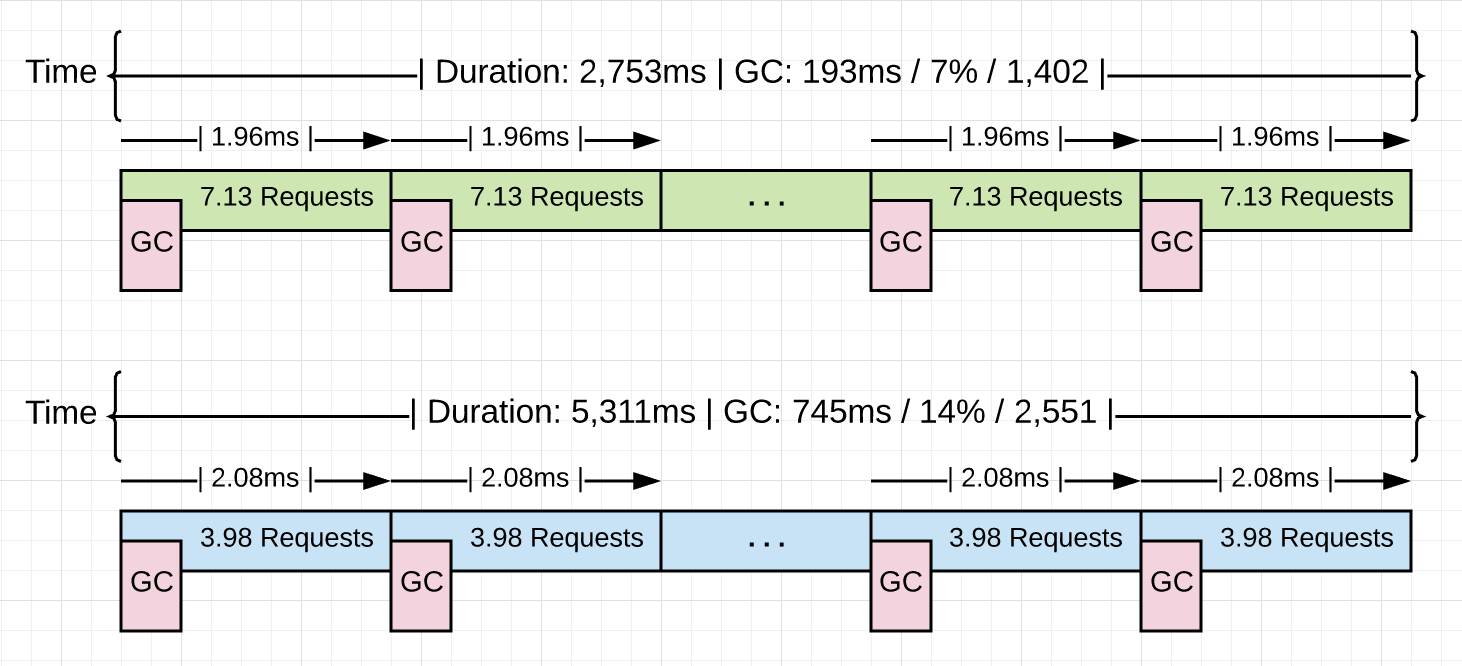
Total Collections : 2,551 Total Collections : 1,402----------------------------------------------------------------------

Total GC Duration : 744.54ms Total GC Duration : 192.71 ms

Average Pace of GC : ~2.08ms Average Pace of GC : ~1.96ms

Requests/Collection : ~3.98 r/gc Requests/Collection : 7.13 r/gc

清单 15 展示了跟最后一次的对比结果。下边是更形象的表达发生了什么



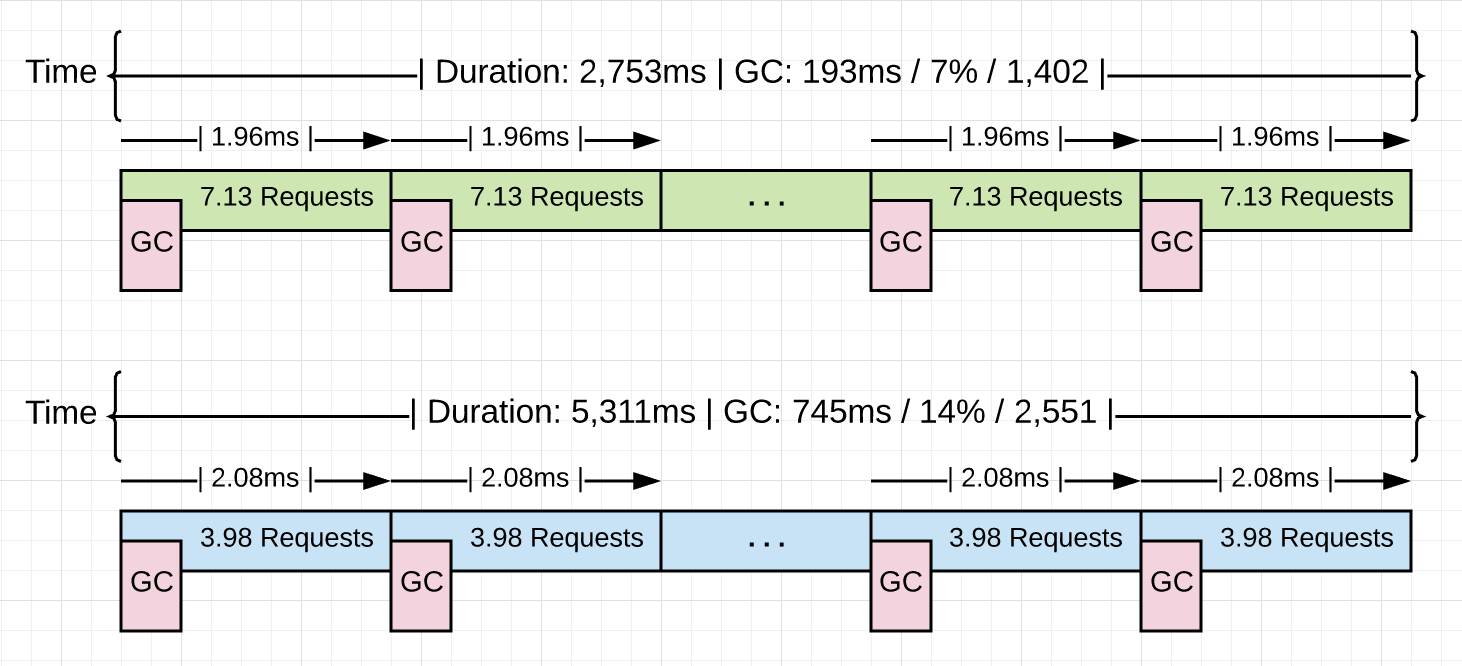


图 5 形象的展示了发生了什么。这次处理相同的 10k 请求回收器少执行了 1149 (1420 vs 2551) 次。这使得整个的 GC 时间百分比从 14% 降到 7%。使应用的运行速度提升了 48%，垃圾回收时间降低了 74%。

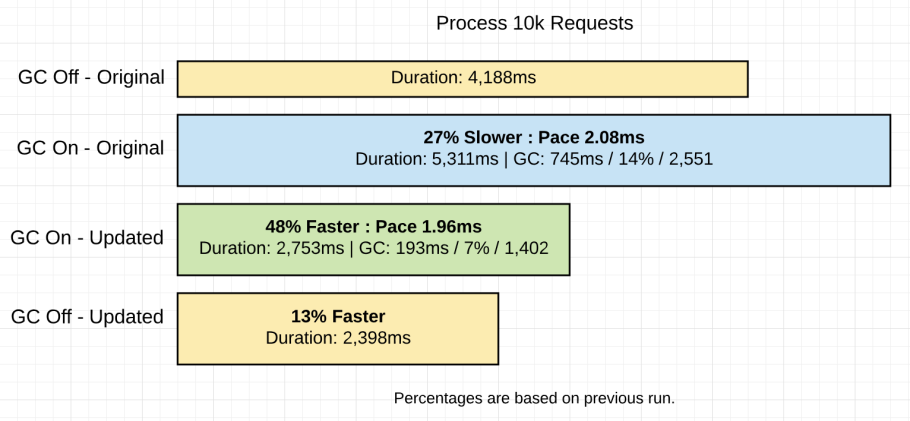


图 6 展示了应用所有不同运行情况的一个对比。为了完整性，我将优化后的代码并关闭垃圾回收的运行情况也包含在内

### 我们学到了什么

正如我在上一篇文章中说的，对垃圾回收器友好就是降低堆上的压力。记住，压力可以定义为应用在确定时间内在堆上分配所有可用内存的速度。当压力降低时，由垃圾回收器所造成的延迟也会降低。拖慢你应用的就是这个延迟。

对回收器友好跟放慢垃圾回收的步调无关，而是跟在垃圾回收的间隔或期间让更多的工作做完有关。你可以通过降低任何一个工作在堆上分配内存的数量和次数来达到目的。

With Extra Allocations Without Extra Allocations======================================================================

Requests : 10,000 Requests : 10,000----------------------------------------------------------------------

Requests/sec : 1,882 r/s Requests/sec : 3,631 r/s

Total Duration : 5,311ms Total Duration : 2,753 ms

Percent Time in GC : 14% Percent Time in GC : 7%

Total Collections : 2,551 Total Collections : 1,402----------------------------------------------------------------------

Total GC Duration : 744.54ms Total GC Duration : 192.71 ms

Average Pace of GC : ~2.08ms Average Pace of GC : ~1.96ms

Requests/Collection : ~3.98 r/gc Requests/Collection : 7.13 r/gc

清单 16 展示了两个版本的应用在垃圾回收打开的情况下运行的结果。很明显，移除了 4.48 G 的内存分配使得应用运行的更快。有趣的是，每一次的垃圾回收的平均时间几乎一样，差不多 2.0 ms。这两个版本之间最根本的改变是每次垃圾回收间隔期间工作完成的数量。应用从 3.98 r/gc 到 7.13 r/gc。工作完成数量增加了 79.1%。

在任何两次垃圾回收间隔之间让更多的工作做完帮助将所需要的垃圾回收次数从 2551 降到 1402，降低了 45%。应用在整个 GC 时间上从 745 ms 降到了 193 ms，74% 的降低，对于各自的版本在垃圾回收时间上也有一个 14% 到 7% 的降低。当以关闭垃圾回收的方式运行代码优化后的版本时，应用消耗时间从 2753 降到了 2398，性能差距仅有 13%。

### 结论

作为一个 Go 开发者，如果你花时间专注于降低内存分配，你正在尽你所能的对垃圾回收器友好。你不能写一个 0 内存分配的应用，所以认清生产性 ( 有助于应用 ) 和非生产性 ( 对应用有害 ) 的内存分配之间的区别很重要。信任垃圾回收器，并保证堆的合理使用，你的应用就会一直运行完好

有一个垃圾回收器是一个很好的权衡。我会为垃圾回收的成本买单，所以我没有内存管理的负担。Go 是想让你作为一个开发者能够快速写出一个性能足够好的应用。垃圾回收器为实现这一目标起到了很大作用。在下一篇文章中，我将会分享另外一个项目，它将会展示垃圾回收器是如何分析你的 Go 应用并找到最佳回收路径的。

## 第三部分-GC步调

### 简介

在第二篇文章里，我向你展示了垃圾回收器的行为以及如何使用工具查看回收器给你的运行程序带来的延迟。我带着你运行了一个真实的 web 应用并向你展示了如何生成 GC 追踪和应用性能分析。接着我还向你展示了如何解读这些工具的输出，以便你找到提高应用程序性能的方法。

第二篇文章的结论与第一篇相同：如果你减少了堆上的压力，就可以降低延迟成本，从而提高应用程序的性能。对回收器友好的最佳策略就是减少每个任务所需内存分配的次数或大小。本篇中，我将向你展示步调算法是如何持续找出给定工作压力下的最优步调的。

### 并发示例代码

我将使用位于该链接处的代码：**<https://github.com/ardanlabs/gotraining/tree/master/topics/go/profiling/trace>**

这个程序将找出某个特定主题在一组 RSS 新闻摘要文档中的出现频率。程序中包含了不同版本的查找算法以测试不同的并发模式。我将主要关注 freq，freqConcurrent 和 freqNumCPU 这几个版本。

注：我是在一台 Macbook Pro 上使用 go1.12.7 运行的代码，该机配备了一个具有 12 个硬件线程的 Intel i9 处理器。在不同的体系架构、操作系统和 Go 版本下，你会看到不同的结果，但本篇的核心结论应该保持不变。

我首先从 freq 版本开始。它表示这个程序的非并发串行版本，并将为下文的并发版本提供一个基准。

01 func freq(topic string, docs []string) int {

02 var found int

03

04 for \_, doc := range docs {

05 file := fmt.Sprintf("%s.xml", doc[:8])

06 f, err := os.OpenFile(file, os.O\_RDONLY, 0)

07 if err != nil {

08 log.Printf("Opening Document [%s] : ERROR : %v", doc, err)

09 return 0

10 }

11 defer f.Close()

12

13 data, err := ioutil.ReadAll(f)

14 if err != nil {

15 log.Printf("Reading Document [%s] : ERROR : %v", doc, err)

16 return 0

17 }

18

19 var d document

20 if err := xml.Unmarshal(data, &d); err != nil {

21 log.Printf("Decoding Document [%s] : ERROR : %v", doc, err)

22 return 0

23 }

24

25 for \_, item := range d.Channel.Items {

26 if strings.Contains(item.Title, topic) {

27 found++

28 continue

29 }

30

31 if strings.Contains(item.Description, topic) {

32 found++33 }

34 }

35 }

36

37 return found

38 }

清单 1 展示的是 freq 函数。这个串行版本遍历一个文件名集合并执行 4 种操作：打开文件、读取文件、解码和检索。它对每个文件执行上述操作，一次一个。

在我的机器上运行这个版本的 freq 时，我得到了如下的结果：

$ time ./trace2019/07/02 13:40:49 Searching 4000 files, found president 28000 times../trace 2.54s user 0.12s system 105% cpu 2.512 total

从 time 的输出中可以看到，程序处理 4000 个文件用了大约 2.5 秒。能看到垃圾回收在其中所占的比例当然是很好的，你可以通过查看程序的追踪结果做到这点。由于这是一个启动并完成的程序，你可以使用 trace 包生成一个追踪。

03 import "runtime/trace"

04

05 func main() {

06 trace.Start(os.Stdout)

07 defer trace.Stop()

清单 3 展示的是从程序生成追踪所需的代码。从标准库的 runtime 文件夹里导入 trace 包后，调用 trace.Start 和 trace.Stop。将追踪输出定向到 os.Stdout 只是简化了代码。

有了这段代码，现在你可以重新编译和运行这个程序了。不要忘了把标准输出重定向到一个文件。

$ go build

$ time ./trace > t.out

Searching 4000 files, found president 28000 times../trace > t.out 2.67s user 0.13s system 106% cpu 2.626 total

运行时间增加了 100 毫秒多一点，但这是意料之中的。追踪捕捉了每次函数调用，进入和退出，直到微秒精度。重要的是现在有了一个名为 t.out 的文件，里面有追踪数据。

为了查看追踪结果，需要使用追踪工具来运行追踪数据。

$ go tool trace t.out

执行以上命令会启动 Chrome 浏览器并显示以下内容：

注：追踪工具使用了 Chrome 浏览器内置的工具，所以它只能在 Chrome 里工作。

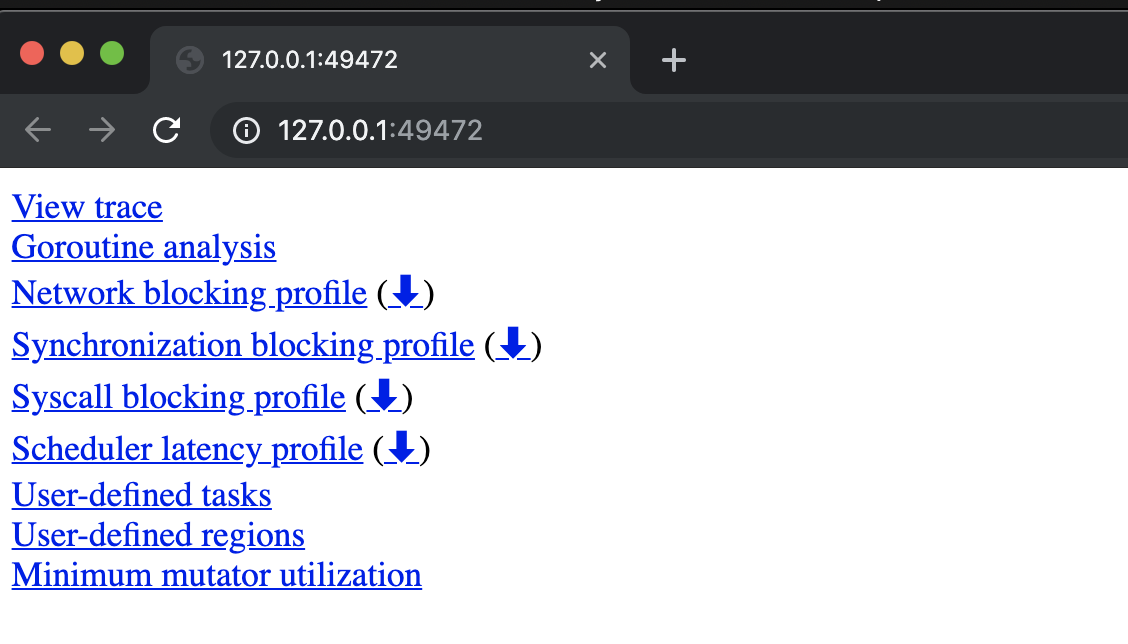


图 1 展示的是追踪工具启动时显示的 9 个链接。当下最重要的是第一个标着 View trace 的链接。点击之后，你就会看到与下图类似的画面：



图 2 展示的是在我的机器上运行程序时的完整追踪窗口。本篇中，我会重点关注与垃圾回收器相关的部分，即标着 Heap 的第二部分和标着 GC 的第四部分。

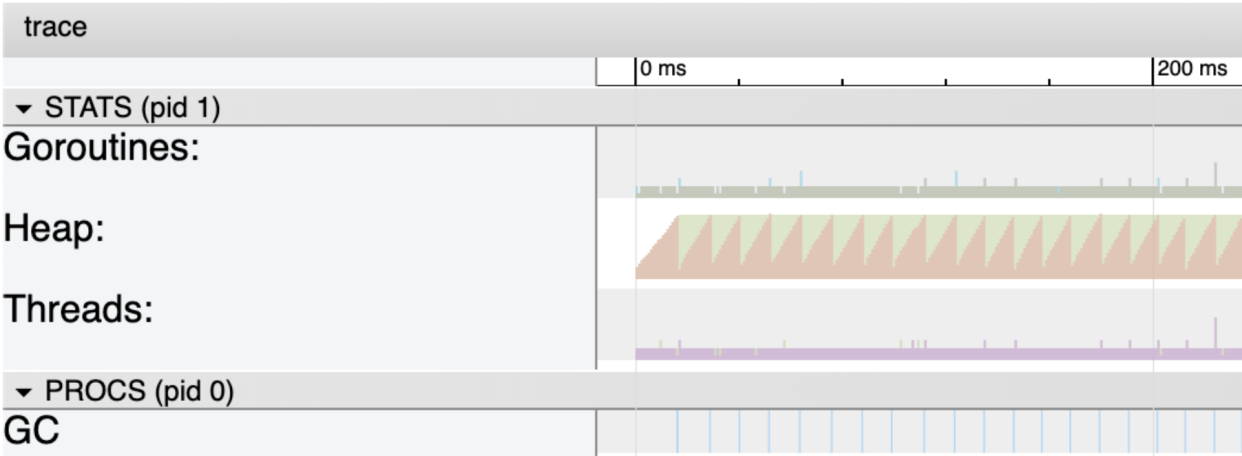


图 3 更详细地展示了追踪的前 200 毫秒。注意观察 Heap（绿色和橙色区域）和 GC（底部的蓝线）。Heap 部分向你展示了两个信息：橙色区域代表在每个微秒时刻对应的堆上正在使用的空间，绿色代表触发下次回收的堆使用量。这就是为什么每当橙色区域到达绿色区域的顶端就会发生一次垃圾回收的原因。蓝线表示一次垃圾回收。

在这个版本的程序整个运行过程中，堆上内存使用量一直保持在大约 4 MB。要想查看每次垃圾回收时的统计数据，你可以使用选择工具在所有蓝线周围绘制一个框。

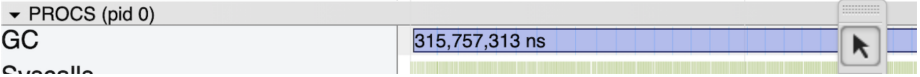


图 4 展示的是如何使用箭头工具在蓝线周围绘制蓝框。你应该想要框住每一条线。框中的数字表示选中的项目消耗的总时间。上图中，有接近 316 毫秒（ms，μs，ns）的区域被选中。当所有的蓝色线条被选中时，就得到了如下的统计结果：

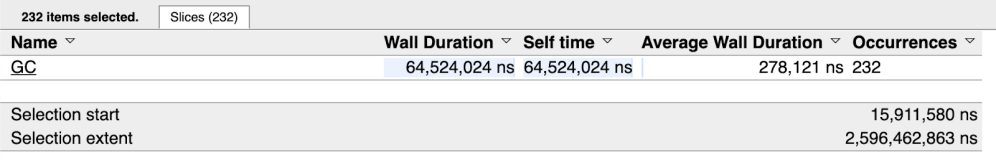


图 5 显示所有的蓝线都在 15.911 毫秒标记到 2.596 秒标记之间。一共有 232 次垃圾回收，共消耗 64.524 毫秒，平均每次消耗 287.121 微秒。而程序运行需要 2.626 秒，这就意味着垃圾回收只占总运行时间的 2%。基本上，垃圾回收是运行该程序的一个微不足道的成本。

有了这个基准，可以使用并发算法完成相同的工作，以期加快程序的速度。

01 func freqConcurrent(topic string, docs []string) int {

02 var found int32

03

04 g := len(docs)

05 var wg sync.WaitGroup

06 wg.Add(g)

07

08 for \_, doc := range docs {

09 go func(doc string) {

10 var lFound int32

11 defer func() {

12 atomic.AddInt32(&found, lFound)

13 wg.Done()

14 }()

15

16 file := fmt.Sprintf("%s.xml", doc[:8])

17 f, err := os.OpenFile(file, os.O\_RDONLY, 0)

18 if err != nil {

19 log.Printf("Opening Document [%s] : ERROR : %v", doc, err)

20 return

21 }

22 defer f.Close()

23

24 data, err := ioutil.ReadAll(f)

25 if err != nil {

26 log.Printf("Reading Document [%s] : ERROR : %v", doc, err)

27 return

28 }

29

30 var d document

31 if err := xml.Unmarshal(data, &d); err != nil {

32 log.Printf("Decoding Document [%s] : ERROR : %v", doc, err)

33 return

34 }

35

36 for \_, item := range d.Channel.Items {

37 if strings.Contains(item.Title, topic) {

38 lFound++

39 continue

40 }

41

42 if strings.Contains(item.Description, topic) {

43 lFound++

44 }

45 }

46 }(doc)

47 }

48

49 wg.Wait()

50 return int(found)

51 }

清单 6 展示的是 freq 的一个可能的并发版本。这个版本的核心设计模式是使用扇出模式。对于 docs 集合中的每个文件，都会创建一个 goroutine 来处理。如果有 4000 个文档要处理，就要使用 4000 个 goroutine。这个算法的优点就是它是利用并发性的最简单方法。每个 goroutine 处理一个且仅处理一个文件。可以使用 WaitGroup 执行等待处理每个文档的编排，并且原子指令可以使计数器保持同步。

这个算法的缺点在于它不能很好地适应文档或 CPU 核心的数量。所有的 goroutine 在程序启动的时候就开始运行，这意味着很快就会消耗大量内存。由于在第 12 行使用了 found 变量，它还导致了缓存一致性的问题。由于每个核心共享这个变量的高速缓存行，这将导致内存抖动。随着文件或核心数量的增加，这会变得更糟。

有了代码，现在你可以重新编译和运行这个程序了。

$ go build

$ time ./trace > t.out

Searching 4000 files, found president 28000 times../trace > t.out 6.49s user 2.46s system 941% cpu 0.951 total

从清单 7 中的输出可以看到，现在程序花了 951 毫秒来处理相同的 4000 个文件。这是大约 64% 的性能提升。看一下追踪结果。



图 6 展示了这个版本的程序在我的机器上运行时是怎样占用了比之前多得多的 CPU。图中的开始部分有很多密集的线条，这是因为所有的 goroutine 都被创建之后，它们运行并且开始尝试从堆上分配内存。很快的，前 4 MB 内存被分配了，紧接着就有一个 GC 启动了。在这次 GC 当中，每个 goroutine 都有时间运行，大部分由于在堆上申请内存而被置于等待状态。至少有 9 个 goroutine 继续运行并在 GC 结束时将堆增长到大约 26 MB。

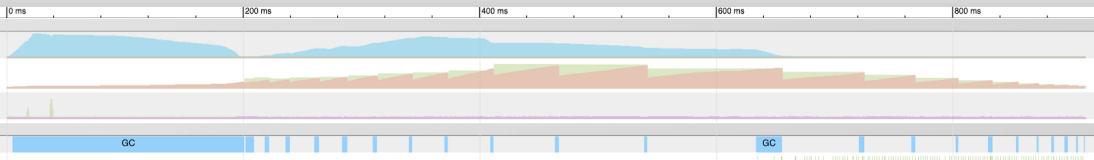


图 7 展示了在首次 GC 的大部分时间里有很多 goroutine 处于已就绪（Runnable）和运行中（Running）的状态以及这一幕是如何再次快速发生的。请注意，堆性能概要（profile）看起来不规则并且垃圾回收也不像之前那样有规律。如果你仔细看，就会发现第二次 GC 几乎是在第一次 GC 结束之后就立即开始了。

如果选中了所有的垃圾回收，你就能看到下面的一幕：

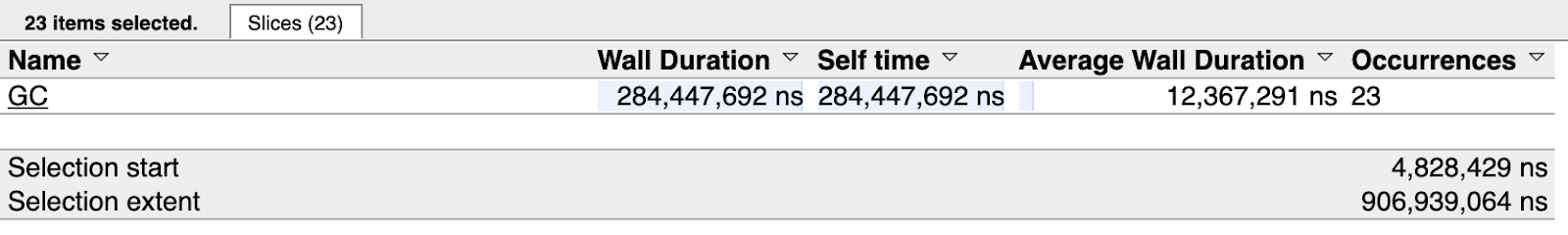


图 8 显示图中所有的蓝线都在 4.828 毫秒标记到 906.939 毫秒标记之间。一共有 23 次垃圾回收，共占用 284.447 毫秒，平均每次占用 12.367 毫秒。知道程序运行需要 951 毫秒，垃圾回收在整个运行时间里占用了大约 34%。

这与串行版本在性能和 GC 时间上有显著差异。并行运行更多的 goroutine 让工作时间缩短了大约 64%。代价就是大幅增加了机器上各种资源的占用。不幸的是，堆上内存的占用最高达到了大约 200 MB。

有了这个并发的基准，下一个并发算法试图更有效率的使用资源。

01 func freqNumCPU(topic string, docs []string) int {

02 var found int32

03

04 g := runtime.NumCPU()

05 var wg sync.WaitGroup

06 wg.Add(g)

07

08 ch := make(chan string, g)

09

10 for i := 0; i < g; i++ {

11 go func() {

12 var lFound int32

13 defer func() {

14 atomic.AddInt32(&found, lFound)

15 wg.Done()

16 }()

17

18 for doc := range ch {

19 file := fmt.Sprintf("%s.xml", doc[:8])

20 f, err := os.OpenFile(file, os.O\_RDONLY, 0)

21 if err != nil {

22 log.Printf("Opening Document [%s] : ERROR : %v", doc, err)

23 return

24 }

25

26 data, err := ioutil.ReadAll(f)

27 if err != nil {

28 f.Close()

29 log.Printf("Reading Document [%s] : ERROR : %v", doc, err)

23 return

24 }

25 f.Close()

26

27 var d document

28 if err := xml.Unmarshal(data, &d); err != nil {

29 log.Printf("Decoding Document [%s] : ERROR : %v", doc, err)

30 return

31 }

32

33 for \_, item := range d.Channel.Items {

34 if strings.Contains(item.Title, topic) {

35 lFound++

36 continue

37 }

38

39 if strings.Contains(item.Description, topic) {

40 lFound++

41 }

42 }

43 }

44 }()

45 }

46

47 for \_, doc := range docs {

48 ch <- doc

49 }

50 close(ch)

51

52 wg.Wait()

53 return int(found)

54 }

清单 8 展示的是 freqNumCPU 版本的程序。该版本的核心设计模式是池模式。由一个基于逻辑处理器数目的 goroutine 池来处理所有的文件。如果有 12 个逻辑处理器可以使用，就使用 12 个 goroutine。这个算法的优点是它从头到尾保持了资源占用的一致性。由于使用了固定数量的 goroutine，所以只需要分配那 12 个 goroutine 运行所需的内存。这也解决了高速缓存一致性问题导致的内存抖动。这是因为第 14 行调用的原子指令只会发生一个很小的固定次数。

该算法的缺点就是它更复杂了。它额外使用了一个 channel 来给 goroutine 池分发工作。要想在每次使用池模式时都为池子指定一个“正确”数量的 goroutine 是非常复杂的。作为一个通常的做法，我为池子启动了与逻辑处理器相同数量的 goroutine。然后通过压力测试或者使用生产指标，可以计算出一个最终的池子大小。

有了代码，现在你可以重新编译和运行这个程序了。

$ go build

$ time ./trace > t.out

Searching 4000 files, found president 28000 times../trace > t.out 6.22s user 0.64s system 909% cpu 0.754 total

从清单 9 中的输出可以看到，现在程序花了 754 毫秒来处理相同的 4000 个文件。程序快了大约 200 毫秒，对于这样一个小的工作量而言，这是一个很大的提升。看一下追踪结果。



图 9 显示了这个版本的程序运行时也同样使用了我机器上所有的 CPU。仔细看的话，你就会发现一个固定的节奏在这个程序里又出现了，跟串行版本很像。

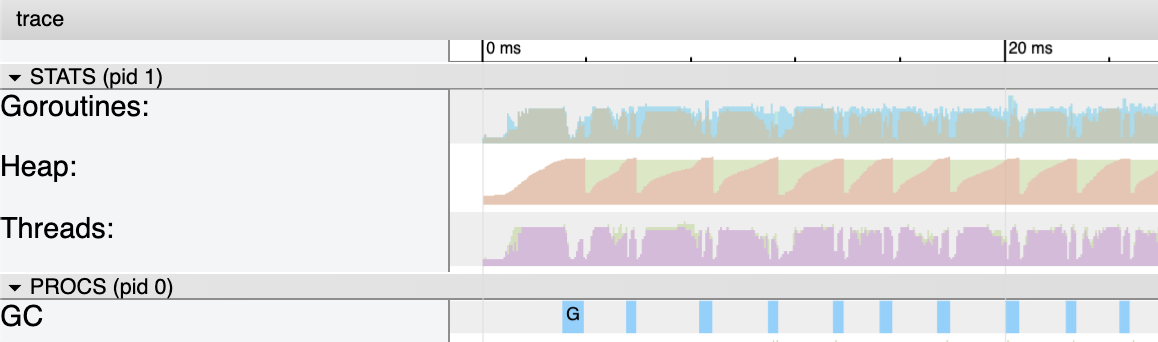


图 10 展示的是程序运行前 20 毫秒的核心指标的一个特写。与串行版本相比，垃圾回收肯定占用了更长的时间，但是有 12 个 goroutine 同时在运行。使用的堆内存在整个程序的运行当中一直保持在 4 MB 左右，这又与串行版本一样了。

如果选中了所有的垃圾回收，你就能看到下面的一幕：

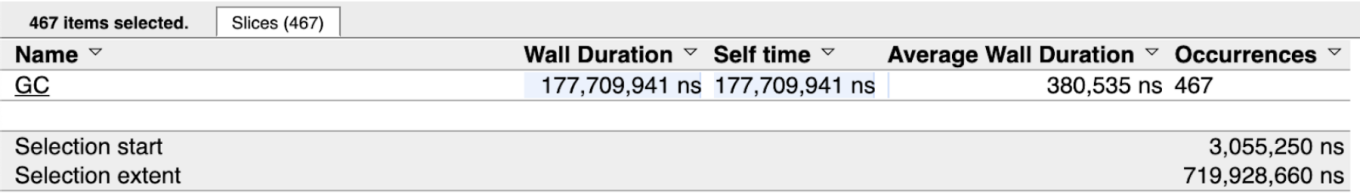


图 11 显示图中所有的蓝线都位于 3.055 毫秒标记到 719.928 毫秒标记之间。共有 467 次垃圾回收，耗时 177.709 毫秒，平均每次花费 380.535 微秒。知道程序运行需要 754 毫秒，这意味着垃圾回收在总的运行时间里占了大约 25%，相比另一个并发版本降低了 9%。

这个版本的并发算法看起来可以适应更多的文件和核心。我认为它增加的复杂性成本是值得的。可以使用列表切片来代替 channel 将每个 goroutine 的任务放到一个桶里。这必然会增加复杂度，尽管它可以减少由 channel 带来的延迟成本。虽然随着文件和核心的增多，这个收益可能会不容忽视，但是它带来的复杂度成本还是需要衡量的。这是你可以自己尝试的东西。

### 结语

我喜欢比较算法的三个版本就是要看看 GC 如何处理每种情况。处理文件所需的内存总量并不随程序版本而变化，不同的是程序如何分配内存。

当只有一个 goroutine 时，只需要最起码的 4 MB 堆内存。当程序一次性分派了所有的工作，GC 采取的策略是使堆增长，减少收集的次数，但是每次运行更长的时间。当程序限制了并行处理的文件数量时，GC 又采取了保持一个小堆的策略，增加回收的次数，但是每次运行更少的时间。基本上，GC 采取的每个策略都是为了使 GC 对程序运行的影响最小。

| Algorithm | Program | GC Time | % Of GC | # of GC’s | Avg GC | Max Heap ||------------|---------|----------|---------|-----------|----------|----------|| freq | 2626 ms | 64.5 ms | ~2% | 232 | 278 μs | 4 meg || concurrent | 951 ms | 284.4 ms | ~34% | 23 | 12.3 ms | 200 meg || numCPU | 754 ms | 177.7 ms | ~25% | 467 | 380.5 μs | 4 meg |

就两个并发版本而言，freqNumCPU 版本带来的额外好处是更好的处理高速缓存一致性的问题，这很有帮助。然而，每个程序在 GC 上花费的总时间差别不大，大约 284.4 毫秒对大约 177.7 毫秒。有时在我的机器上运行这些程序，这些数字会更加接近。使用 go 1.13.beta1 版本做了一些实验，我甚至看到这两个算法运行了相同的时间。这可能意味着新版的 go 有一些改进能够让 GC 更好的预测如何运行。

所有这些都让我有信心在程序运行时抛出大量任务。一个例子就是一个使用 50000 goroutine 的 web 服务，这就是一个本质上与第一个并发算法类似的扇出模式。GC 会研究工作量并为服务找到最优的步调来避开它。至少对我而言，使用 GC 是值得的，因为不用去考虑所有这些。