# 详解逃逸分析

Go是一门带有垃圾回收的现代语言，它抛弃了传统C/C++的开发者需要手动管理内存的方式，实现了内存的主动申请和释放的管理。Go的垃圾回收，让堆和栈的概念对程序员保持透明，它增加的逃逸分析与GC，使得程序员的双手真正地得到了解放，给了开发者更多的精力去关注软件设计本身。

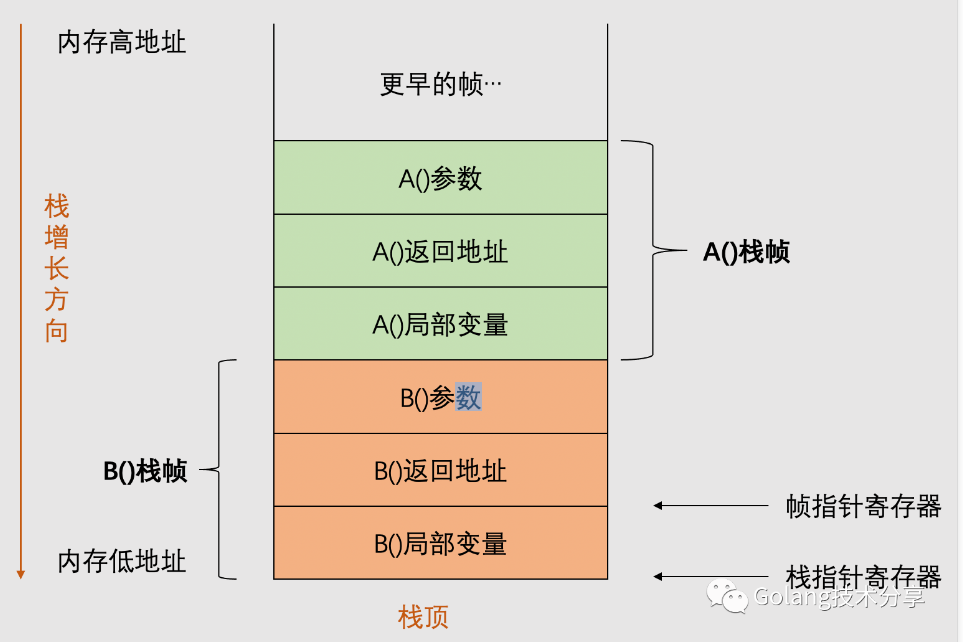
就像[《CPU缓存体系对Go程序的影响》](http://mp.weixin.qq.com/s?__biz=MzIwNTA4MDAwMQ==&mid=2247484486&idx=1&sn=83ebd97be71caad3537502e459182f88&chksm=973710e8a04099fe6f113c38a9e9940c0ff590877ad7ba0a0f4c383356e91619574f052a6064&scene=21#wechat_redirect)文章中说过的一样，“你不一定需要成为一名硬件工程师，但是你确实需要了解硬件的工作原理”。Go虽然帮我们实现了内存的自动管理，我们仍然需要知道其内在原理。内存管理主要包括两个动作：分配与释放。逃逸分析就是服务于内存分配，为了更好理解逃逸分析，我们先谈一下堆栈。

## 堆和栈

应用程序的内存载体，我们可以简单地将其分为堆和栈。

在Go中，栈的内存是由编译器自动进行分配和释放，栈区往往存储着函数参数、局部变量和调用函数帧，它们随着函数的创建而分配，函数的退出而销毁。一个goroutine对应一个栈，栈是调用栈（call stack）的简称。一个栈通常又包含了许多栈帧（stack frame），它描述的是函数之间的调用关系，每一帧对应一次尚未返回的函数调用，它本身也是以栈形式存放数据。

举例：在一个goroutine里，函数A()正在调用函数B()，那么这个调用栈的内存布局示意图如下。



与栈不同的是，应用程序在运行时只会存在一个堆。狭隘地说，内存管理只是针对堆内存而言的。程序在运行期间可以主动从堆上申请内存，这些内存通过Go的内存分配器分配，并由垃圾收集器回收。

栈是每个goroutine独有的，这就意味着栈上的内存操作是不需要加锁的。而堆上的内存，有时需要加锁防止多线程冲突（为什么要说有时呢，因为Go的内存分配策略学习了TCMalloc的线程缓存思想，他为每个处理器P分配了一个mcache，从mcache分配内存也是无锁的）。

而且，对于程序堆上的内存回收，还需要通过标记清除阶段，例如Go采用的三色标记法。但是，在栈上的内存而言，它的分配与释放非常廉价。简单地说，它只需要两个CPU指令：一个是分配入栈，另外一个是栈内释放。而这，只需要借助于栈相关寄存器即可完成。

另外还有一点，栈内存能更好地利用CPU的缓存策略。因为它们相较于堆而言是更连续的。

我们如何知道一个对象是应该放在堆内存，还是栈内存之上呢？可以官网的FAQ（地址：https://golang.org/doc/faq）中找到答案。

如果可以，Go编译器会尽可能将变量分配到到栈上。但是，当编译器无法证明函数返回后，该变量没有被引用，那么编译器就必须在堆上分配该变量，以此避免悬挂指针（dangling pointer）。另外，如果局部变量非常大，也会将其分配在堆上。

那么，Go是如何确定的呢？答案就是：逃逸分析。编译器通过逃逸分析技术去选择堆或者栈，逃逸分析的基本思想如下：检查变量的生命周期是否是完全可知的，如果通过检查，则可以在栈上分配。否则，就是所谓的逃逸，必须在堆上进行分配。

Go语言虽然没有明确说明逃逸分析规则，但是有以下几点准则，是可以参考的。

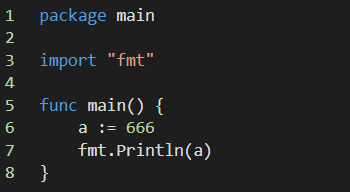
1、逃逸分析是在编译器完成的，这是不同于jvm的运行时逃逸分析;

2、如果变量在函数外部没有引用，则优先放到栈中；

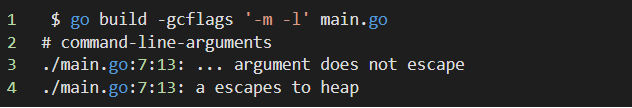
3、如果变量在函数外部存在引用，则必定放在堆中；

我们可通过go build -gcflags '-m -l'命令来查看逃逸分析结果，其中-m 打印逃逸分析信息，-l禁止内联优化。下面，我们通过一些案例，来熟悉一些常见的逃逸情况。

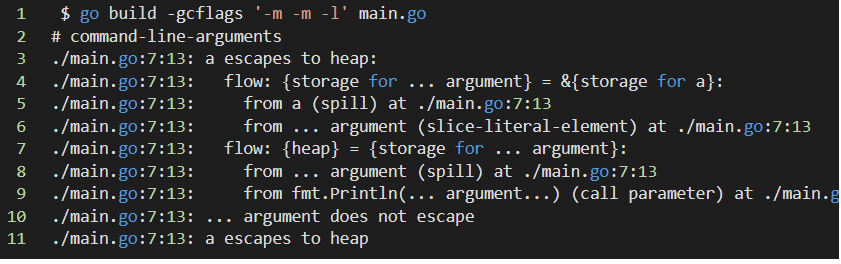
### 情况一：变量类型不确定



逃逸分析结果如下



分析结果告诉我们变量a逃逸到了堆上。但是我们并没有外部引用，为什么也会有逃逸呢？为了看到更多细节，可以在语句中再添加一个-m参数。得到信息如下

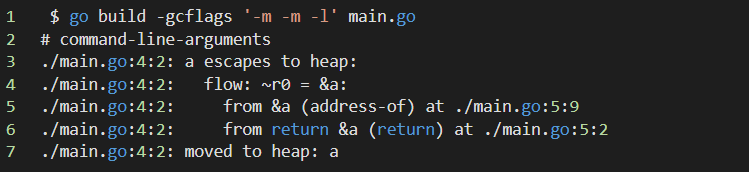


a逃逸是因为它被传入了fmt.Println的参数中，这个方法参数自己发生了逃逸。

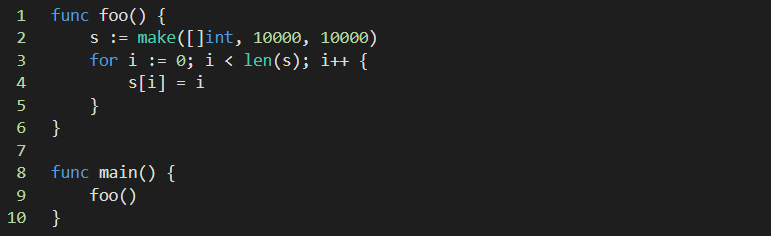
因为fmt.Println的函数参数为**interface类型**，编译期不能确定其**参数的具体类型**，所以将其分配于堆上。

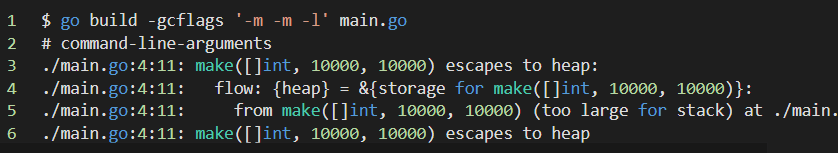
### 情况二：暴露给外部指针

逃逸分析如下，变量a发生了逃逸。

这种情况直接满足我们上述中的原则：变量在函数外部存在引用。这个很好理解，因为当函数执行完毕，对应的**栈帧**就被**销毁**，但是引用已经被返回到函数之外。如果这时外部从引用地址取值，虽然**地址还在**，但是这块**内存**已经被**释放回收**了，这就是**非法内存**，问题可就大了。所以，很明显，这种情况必须分配到堆上。

### 情况三：变量所占内存较大

逃逸分析结果



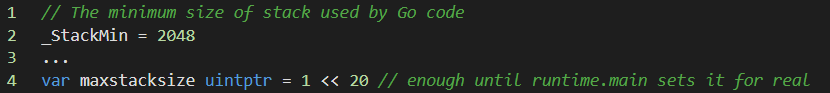
当我们创建了一个容量为10000的int类型的底层数组对象时，由于对象过大，它也会被分配到堆上。这里我们不禁要想一个问题，为啥大对象需要分配到堆上。

这里需要注意，在上文中没有说明的是：在Go中，执行用户代码的**goroutine**是一种**用户态线程**，其调用栈内存被称为**用户栈**，它其实也是从**堆区**分配的，但是我们仍然可以将其看作和系统栈一样的内存空间，它的分配和释放是通过编译器完成的。与其相对应的是**系统栈**，它的分配和释放是操作系统完成的。在GMP模型中，一个M对应一个**系统栈**（也称为M的g0栈），M上的多个**goroutine**会**共享**该**系统栈**。

不同平台上的系统栈最大限制不同。



以x86\_64架构为例，它的系统栈大小最大可为8Mb。我们常说的goroutine初始大小为2kb，其实说的是用户栈，它的最小和最大可以在runtime/stack.go中找到，分别是2KB和1GB。

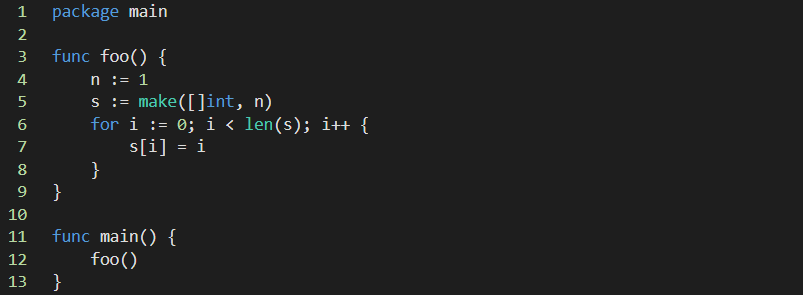


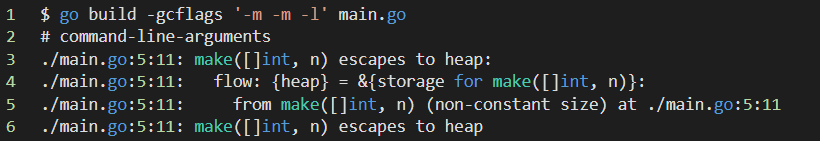
而堆则会大很多，从1.11之后，Go采用了稀疏的内存布局，在Linux的x86-64架构上运行时，整个堆区最大可以管理到256TB的内存。所以，**为了不造成栈溢出和频繁的扩缩容，大的对象分配在堆上更加合理**。那么，多大的对象会被分配到堆上呢。

通过测试，小菜刀发现该大小为64KB（这在Go内存分配中是属于大对象的范围：>32kb），即s :=make([]int, n, n)中，一旦n达到8192，就一定会逃逸。注意，网上有人通过fmt.Println(unsafe.Sizeof(s))得到s的大小为24字节，就误以为只需分配24个字节的内存，这是错误的，因为实际还有底层数组的内存需要分配。

### ****情况四：变量大小不确定****

我们将情况三种的示例，简单更改一下。





这次，我们在make方法中，没有直接指定大小，而是填入了变量n，这时Go逃逸分析也会将其分配到堆区去。可见，**为了保证内存的绝对安全**，Go的编译器可能会将**一些变量不合时宜地分配到堆上**，但是因为这些对象最终也会被垃圾收集器处理，所以也能接受。

## **总结**

本文只列举了逃逸分析的部分例子，实际的情况还有很多，理解思想最重要。这里就不过多列举了。

既然Go的堆栈分配对于开发者来说是透明的，编译器已经通过逃逸分析为对象选择好了分配方式。那么我们还可以从中获益什么？

答案是肯定的，理解逃逸分析一定能帮助我们写出更好的程序。知道变量分配在栈堆之上的差别，那么我们就要尽量写出分配在栈上的代码，堆上的变量变少了，可以减轻内存分配的开销，减小gc的压力，提高程序的运行速度。

所以，你会发现有些Go上线项目，它们在函数传参的时候，并没有传递结构体指针，而是直接传递的结构体。这个做法，虽然它需要值拷贝，但是这是在栈上完成的操作，开销远比变量逃逸后动态地在堆上分配内存少的多。当然该做法不是绝对的，如果结构体较大，传递指针将更合适。

因此，从GC的角度来看，指针传递是个双刃剑，需要谨慎使用，否则线上调优解决GC延时可能会让你崩溃。