链滴

golang 逃逸分析

作者：cuua

原文链接：https://ld246.com/article/1618996032697

来源网站：链滴

golang逃逸分析

垃圾回收是 Go - 自动内存管理的一个便利功能， 使代码更清洁，内存泄漏的可能性更小。但是，GC 还会增加间接性能消耗，因为程序需要定期停止并收集未使用的对象。Go 编译器足够智能，可以自决定是否应在堆上分配变量，之后需要在堆上收集垃圾，或者是否可以将其分配为该变量的函数的栈一部分。栈与堆分配变量不同，堆分配变量不会产生任何 GC 开销，因为它们在栈的其余部分（当功返回时）被销毁。

例如，Go 的逃生分析比HotSpot JVM 更基本。基本规则是，如果从申报的函数返回对变量的引用，会"逃逸" - 函数返回后可以引用该变量，因此必须将其堆分配。这是比较复杂的，因为：

- 调用其他功能的函数
- 分配给结构体成员的引用
- 切片和maps
- cgo将指针指向变量

为了执行逃生分析，Go 在编译时构建一个函数调用图，并跟踪输入参数和返回值的流。函数可能引其中一个参数，但如果该引用未返回，变量不会逃逸。函数也可以返回引用，但在申明变量返回的函之前，该引用可能由栈中的另一个函数取消引用或未返回。为了说明一些简单的案例，我们可以运行译器，这将打印详细的逃生分析信息：-gcflags '-m'

```
package main

type S struct {}

func main() {
  var x S
  _ = identity(x)
}

func identity(x S) S {
  return x
}
```

你必须用 go run -gcflags '-m -l' '-l'标签阻止功能被内联 （这是另一个时间的主题） 来构建这个功。输出是：什么都没有！ Go 使用值传递，因此始终将变量复制到栈中。在没有引用的一般代码中，是很少使用栈分配。没有逃生分析可做。再看下面一个例子：

```
package main

type S struct {}

func main() {
  var x S
  y := &x
  _ = *identity(y)
}

func identity(z *S) *S {
  return z
}
```

输出：

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
.\main.go:11:15: leaking param: z to result ~r1 level=0
```

第一行显示变量"流过"：输入变量返回为输出。但不采取参考，所以变量不会逃逸。不在main返回之没有对x的引用存在，因此x分配在main的堆上。 第三个实验：

```
package main

type S struct {}

func main() {
  var x S
  _ = *ref(x)
}

func ref(z S) *S {
  return &z
}
```

输出：

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
.\main.go:10:10: moved to heap: z
```

现在有一些逃避正在发生。请记住，go是值传递，所以z是main中x变量的副本。 返回z的引用，所以不能是栈的一部分-返回时的参考点在哪里？取而代之的是它逃到堆。尽管 Go 在不取消计算参考值情况下会立即扔掉引用，但 Go 的逃逸分析不够精密，无法找出这一点 - 它只查看输入和返回变量的。值得注意的是，在这种情况下，如果我们不阻止它，编译器就会强调这一点。

如果将引用分配给结构成员，该怎么办？

```
package main

type S struct {
  M *int
}

func main() {
  var i int
  refStruct(i)
}

func refStruct(y int) (z S) {
  z.M = &y
  return z
}
```

输出：

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
.\main.go:13:16: moved to heap: y
```

在这种情况下，Go 仍然可以跟踪引用流，即使引用是结构体的成员。既然refStruct 做了引用并返回，y就必须逃逸。与本案例相比：

```
package main

type S struct {
  M *int
}

func main() {
  var i int
  refStruct(&i)
}

func refStruct(y *int) (z S) {
  z.M = y
  return z
}
```

输出：

```
$ go run -gcflags '-m -l' main.go
# command-line-arguments
.\main.go:13:16: leaking param: y to result z level=0
```

由于main做了引用并传递refStruct，引用永远不会超过申报引用变量的栈。这和前面的程序有稍微同的语义，但如果第二个程序足够的话，它会更有效率：在第一个例子i必须分配在main的栈上，然在堆上重新分配并将其复制为refStruct的参数。在第二个示例中i只分配一次，并传递引用。

一个更深入的例子：

```
package main

type S struct {
  M *int
}

func main() {
  var x S
  var i int
  ref(&i, &x)
}

func ref(y *int, z *S) {
  z.M = y
}
```

输出：

```
$ go run -gcflags '-m -l' main.go
```

```
# command-line-arguments
.\main.go:14:10: leaking param: y
.\main.go:14:18: z does not escape
.\main.go:10:6: moved to heap: i
```

这里的问题是y是分配给输入结构体的成员。Go 无法跟踪该关系 - 输入仅允许流到输出 - 因此逃逸分失败，必须对变量进行堆分配。有许多有据可查的案例（as of Go 1.5），由于go逃逸分析的限制，须堆分配变量 -请参阅此链接 。

最后，maps和切片呢？请记住，maps和切片实际上只是使用指针构建到堆分配的内存：切片结构暴在包中（SliceHeader ）中。map结构是更难找到的，但它存在：hmap 。如果这些结构无法逃逸，们将被栈分配，但备份数组或哈希存储桶中的数据本身将每次都堆分配。避免这种情况的唯一方法是配一个固定大小的数组（如[10000]int）。

如果您已经看过分析程序的堆使用情况，并且需要减少 GC 时间，则可能会从堆中移动频繁分配的变而获得一些收获。这也只是一个引人入胜的话题：要进一步阅读 HotSpot JVM 如何处理逃逸分析，查看这篇文章，其中涉及堆栈分配，以及检测何时可以消除同步。