Go变量逃逸分析

目录

写过C/C++的同学都知道，调用著名的malloc和new函数可以在堆上分配一块内存，这块内存的使用和销毁的责任都在程序员。一不小心，就会发生内存泄露，搞得胆战心惊。

切换到Golang后，基本不会担心内存泄露了。虽然也有new函数，但是使用new函数得到的内存不一定就在堆上。堆和栈的区别对程序员"模糊化"了，当然这一切都是Go编译器在背后帮我们完成的。

一个变量是在堆上分配，还是在栈上分配，是经过编译器的 逃逸分析 之后得出的结论。

这篇文章，就将带领大家一起去探索 逃逸分析 ——变量到底去哪儿，堆还是栈？

什么是逃逸分析

以前写C/C++代码时，为了提高效率，常常将 `pass-by-value` （传值）"升级"成 `pass-by-reference` ，企图避免构造函数的运行，并且直接返回一个指针。

你一定还记得，这里隐藏了一个很大的坑：在函数内部定义了一个局部变量，然后返回这个局部变量的地址（指针）。这些局部变量是在栈上分配的（静态内存分配），一旦函数执行完毕，变量占据的内存会被销毁，任何对这个返回值作的动作（如解引用），都将扰乱程序的运行，甚至导致程序直接崩溃。比如下面的这段代码：

```
int *foo ( void )
{
    int t = 3;
    return &t;
}
```

有些同学可能知道上面这个坑，用了个更聪明的做法：在函数内部使用new函数构造一个变量（动态内存分配），然后返回此变量的地址。因为变量是在堆上创建的，所以函数退出时不会被销毁。但是，这样就行了吗？new出来的对象该在何时何地delete呢？调用者可能会忘记delete或者直接拿返回值传给其他函数，之后就再也不能delete它了，也就是发生了内存泄露。关于这个坑，大家可以去看看《Effective C++》条款21，讲得非常好！

C++是公认的语法最复杂的语言，据说没有人可以完全掌握C++的语法。而这一切在Go语言中就大不相同了。像上面示例的C++代码放到Go里，没有任何问题。

你表面的光鲜，一定是背后有很多人为你撑起的！Go语言里就是编译器的 逃逸分析 。它是编译器执行静态代码分析后，对内存管理进行的优化和简化。

在编译原理中，分析指针动态范围的方法称之为 逃逸分析 。通俗来讲，当一个对象的指针被多个方法或线程引用时，我们称这个指针发生了逃逸。

更简单来说， 逃逸分析 决定一个变量是分配在堆上还是分配在栈上。

为什么要逃逸分析

前面讲的C/C++中出现的问题，在Go中作为一个语言特性被大力推崇。真是C/C++之砒霜Go之蜜糖！

C/C++中动态分配的内存需要我们手动释放，导致猿们平时在写程序时，如履薄冰。这样做有他的好处：程序员可以完全掌控内存。但是缺点也是很多的：经常出现忘记释放内存，导致内存泄露。所以，很多现代语言都加上了垃圾回收机制。

Go的垃圾回收，让堆和栈对程序员保持透明。真正解放了程序员的双手，让他们可以专注于业务，“高效”地完成代码编写。把那些内存管理的复杂机制交给编译器，而程序员可以去享受生活。

逃逸分析 这种“骚操作”把变量合理地分配到它该去的地方，“找准自己的位置”。即使你是用new申请到的内存，如果我发现你竟然在退出函数后没有用了，那么就把你丢到栈上，毕竟栈上的内存分配比堆上快很多；反之，即使你表面上只是一个普通的变量，但是经过逃逸分析后发现在退出函数之后还有其他地方在引用，那我就把你分配到堆上。真正地做到“按需分配”，提前实现共产主义！

如果变量都分配到堆上，堆不像栈可以自动清理。它会引起Go频繁地进行垃圾回收，而垃圾回收会占用比较大的系统开销（占用CPU容量的25%）。

堆和栈相比，堆适合不可预知大小的内存分配。但是为此付出的代价是分配速度较慢，而且会形成内存碎片。栈内存分配则会非常快。栈分配内存只需要两个CPU指令：“PUSH”和“RELEASE”，分配和释放；而堆分配内存首先需要去找到一块大小合适的内存块，之后要通过垃圾回收才能释放。

通过逃逸分析，可以尽量把那些不需要分配到堆上的变量直接分配到栈上，堆上的变量少了，会减轻分配堆内存的开销，同时也会减少gc的压力，提高程序的运行速度。

逃逸分析是怎么完成的

Go逃逸分析最基本的原则是：如果一个函数返回对一个变量的引用，那么它就会发生逃逸。

简单来说，编译器会分析代码的特征和代码生命周期，Go中的变量只有在编译器可以证明在函数返回后不会再被引用的，才分配到栈上，其他情况下都是分配到堆上。

Go语言里没有一个关键字或者函数可以直接让变量被编译器分配到堆上，相反，编译器通过分析代码来决定将变量分配到何处。

对一个变量取地址，可能会被分配到堆上。但是编译器进行逃逸分析后，如果考察到在函数返回后，此变量不会被引用，那么还是会被分配到栈上。套个取址符，就想骗补助？Too young！

简单来说，编译器会根据变量是否被外部引用来决定是否逃逸：

1. 如果函数外部没有引用，则优先放到栈中；
2. 如果函数外部存在引用，则必定放到堆中；

针对第一条，可能放到堆上的情形：定义了一个很大的数组，需要申请的内存过大，超过了栈的存储能力。

逃逸分析实例

Go提供了相关的命令，可以查看变量是否发生逃逸。

还是用上面我们提到的例子：

```
package main

import "fmt"

func foo() *int {
    t := 3
    return &t;
}

func main() {
    x := foo()
    fmt.Println(*x)
}
```

foo函数返回一个局部变量的指针，main函数里变量x接收它。执行如下命令：

```
go build -gcflags '-m -l' main.go
```

加 `-l` 是为了不让foo函数被内联。得到如下输出：

```
# command-line-arguments
src/main.go:7:9: &t escapes to heap
src/main.go:6:7: moved to heap: t
src/main.go:12:14: *x escapes to heap
src/main.go:12:13: main ... argument does not escape
```

foo函数里的变量 `t` 逃逸了，和我们预想的一致。让我们不解的是为什么main函数里的 `x` 也逃逸了？这是因为有些函数参数为interface类型，比如fmt.Println(a ...interface{})，编译期间很难确定其参数的具体类型，也会发生逃逸。

使用反汇编命令也可以看出变量是否发生逃逸。

2019年7月(12)
2019年6月(4)
2019年5月(15)
2019年4月(9)
2019年3月(18)
2019年2月(8)
2019年1月(11)
2018年12月(11)
2018年11月(43)
2018年10月(31)

最新评论

1. Re:Go并发编程

不错，好文章

--jeffhas

2. Re:PHP调用百度地图API

学习了…… 这个API好像更新了？

--钱隆诸葛智能分析软件

3. Re:Installation request for topthink/think-captcha ^3.0 -> satisfiable by topthink/think-captcha[v3.0.0].

完美解决了我的问题！

--pandaQQQ

4. Re:JavaScript原型链

讲的很好， javascript对象原型的理念类似 面向对象设计模式中的 原型模式。
但是又有些区别， javascript对象原型 是动态的，可以动态增加属性和方法。

--雨林

5. Re:Go并发编程

@ hansoluo好的，我现在加上...

--itbsl

阅读排行榜

1. 终端(命令行)连接MySQL(23265)
2. linux创建用户并设置密码(13613)
3. 在HTML中使用JavaScript(12610)
4. MySQL备份与恢复(7131)
5. 10分钟看懂Docker和K8S(5300)

评论排行榜

1. PHP编码风格规范(8)
2. PHPStorm设置等号对齐(5)
3. Go并发编程(3)
4. 10分钟看懂Docker和K8S(3)
5. PHPWord导出word文档(2)

推荐排行榜

1. 10分钟看懂Docker和K8S(12)
2. linux创建用户并设置密码(2)
3. Go语言反射reflect(2)
4. LNMP环境搭建(PHP7.2.25)(2)
5. Go并发编程(2)

```
go tool compile -S main.go
```

截取部分结果，图中标记出来的说明 `t` 是在堆上分配内存，发生了逃逸。

```
"".foo STEXT size=79 args=0x8 locals=0x18
        0x0000 00000 (./src/main.go:5)  TEXT    "".foo(SB), $24-8
        0x0000 00000 (./src/main.go:5)  MOVQ    (TLS), CX
        0x0009 00009 (./src/main.go:5)  CMPQ    SP, 16(CX)
        0x000d 00013 (./src/main.go:5)  JLS     72
        0x000f 00015 (./src/main.go:5)  SUBQ    $24, SP
        0x0013 00019 (./src/main.go:5)  MOVQ    BP, 16(SP)
        0x0018 00024 (./src/main.go:5)  LEAQ    16(SP), BP
        0x001d 00029 (./src/main.go:5)  FUNCDATA        $0, gclocals·2a5305abe05176240e61b8620e19a815(SB)
        0x001d 00029 (./src/main.go:5)  FUNCDATA        $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x001d 00029 (./src/main.go:5)  LEAQ    type.int(SB), AX
        0x0024 00036 (./src/main.go:6)  MOVQ    AX, (SP)
        0x0028 00040 (./src/main.go:6)  PCDATA  $0, $0
        0x0028 00040 (./src/main.go:6)  CALL    runtime.newobject(SB)
        0x002d 00045 (./src/main.go:6)  MOVQ    8(SP), AX
        0x0032 00050 (./src/main.go:6)  MOVQ    $3, (AX)
        0x0039 00057 (./src/main.go:7)  MOVQ    AX, "".~r0+32(SP)
        0x003e 00062 (./src/main.go:7)  MOVQ    16(SP), BP
        0x0043 00067 (./src/main.go:7)  ADDQ    $24, SP
        0x0047 00071 (./src/main.go:7)  RET
        0x0048 00072 (./src/main.go:7)  NOP
        0x0048 00072 (./src/main.go:5)  PCDATA  $0, $-1
        0x0048 00072 (./src/main.go:5)  CALL    runtime.morestack_noctxt(SB)
        0x004d 00077 (./src/main.go:5)  JMP     0
```


总结

堆上动态分配内存比栈上静态分配内存，开销大很多。

变量分配在栈上需要能在编译期确定它的作用域，否则会分配到堆上。

Go编译器会在编译期对考察变量的作用域，并作一系列检查，如果它的作用域在运行期间对编译器一直是可知的，那么就会分配到栈上。

简单来说，编译器会根据变量是否被外部引用来决定是否逃逸。对于Go程序员来说，编译器的这些逃逸分析规则不需要掌握，我们只需通过 `go build -gcflags '-m'` 命令来观察变量逃逸情况就行了。

不要盲目使用变量的指针作为函数参数，虽然它会减少复制操作。但其实当参数为变量自身的时候，复制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。

最后，尽量写出少一些逃逸的代码，提升程序的运行效率。

转载自:
Golang之变量去哪儿?

分类: Go

标签: Go, 逃逸分析

itbsl
关注 - 21
粉丝 - 44
+加关注

1 0

« 上一篇： Git使用详细教程(7)：.gitignore使用详解
» 下一篇： Go接口interface

posted @ 2019-03-05 14:40 itbsl 阅读(1312) 评论(0) 编辑 收藏