逃逸分析

前言

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。
函数中申请一个新的对象

- 如果分配在栈中，则函数执行结束可自动将内存回收；
- 如果分配在堆中，则函数执行结束可交给GC（垃圾回收）处理；

有了逃逸分析，返回函数局部变量将变得可能，除此之外，逃逸分析还跟闭包息息相关，了解哪些场景下对象会逃逸至关重要。

逃逸策略

每当函数中申请新的对象，编译器会根据该对象是否被函数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；

注意，对于函数外部没有引用的对象，也有可能放到堆中，比如内存过大超过栈的存储能力。

逃逸场景

指针逃逸

我们知道Go可以返回局部变量指针，这其实是一个典型的变量逃逸案例，示例代码如下：

```
package main

type Student struct {
    Name string
    Age  int
}

func StudentRegister(name string, age int) *Student {
    s := new(Student) //局部变量s逃逸到堆

    s.Name = name
    s.Age = age

    return s
}

func main() {
    StudentRegister("Jim", 18)
}
```

函数StudentRegister()内部s为局部变量，其值通过函数返回值返回，s本身为一指针，其指向的内存地址不会是栈而是堆，这就是典型的逃逸案例。

通过编译参数-gcflag=-m可以查看编译过程中的逃逸分析：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:8: can inline StudentRegister
.\main.go:17: can inline main
.\main.go:18: inlining call to StudentRegister
.\main.go:8: leaking param: name
.\main.go:9: new(Student) escapes to heap
.\main.go:18: main new(Student) does not escape
```

可见在StudentRegister()函数中，也即代码第9行显示”escapes to heap”，代表该行内存分配发生了逃逸现象。

栈空间不足逃逸

看下面的代码，是否会产生逃逸呢?

```
package main

func Slice() {
    s := make([]int, 1000, 1000)

    for index, _ := range s {
        s[index] = index
    }
}

func main() {
    Slice()
}
```

上面代码Slice()函数中分配了一个1000个长度的切片，是否逃逸取决于栈空间是否足够大。
直接查看编译提示，如下：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:4: Slice make([]int, 1000, 1000) does not escape
```

我们发现此处并没有发生逃逸。那么把切片长度扩大10倍即10000会如何呢?

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:4: make([]int, 10000, 10000) escapes to heap
```

我们发现当切片长度扩大到10000时就会逃逸。

实际上当栈空间不足以存放当前对象时或无法判断当前切片长度时会将对象分配到堆中。

动态类型逃逸

很多函数参数为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，也会产生逃逸。
如下代码所示：

```
package main

import "fmt"

func main() {
    s := "Escape"
    fmt.Println(s)
}
```

上述代码s变量只是一个string类型变量，调用fmt.Println()时会产生逃逸：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:7: s escapes to heap
.\main.go:7: main ... argument does not escape
```

闭包引用对象逃逸

某著名的开源框架实现了某个返回Fibonacci数列的函数：

```
func Fibonacci() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}
```

该函数返回一个闭包，闭包引用了函数的局部变量a和b，使用时通过该函数获取该闭包，然后每次执行闭包都会依次输出Fibonacci数列。
完整的示例程序如下所示：

```
package main

import "fmt"

func Fibonacci() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}

func main() {
    f := Fibonacci()

    for i := 0; i < 10; i++ {
        fmt.Printf("Fibonacci: %d\n", f())
```

```
    }
}
```

上述代码通过Fibonacci()获取一个闭包，每次执行闭包就会打印一个Fibonacci数值。输出如下所示：

```
D:\SourceCode\GoExpert\src>src.exe
Fibonacci: 1
Fibonacci: 1
Fibonacci: 2
Fibonacci: 3
Fibonacci: 5
Fibonacci: 8
Fibonacci: 13
Fibonacci: 21
Fibonacci: 34
Fibonacci: 55
```

Fibonacci()函数中原本属于局部变量的a和b由于闭包的引用，不得不将二者放到堆上，以致产生逃逸：

```
D:\SourceCode\GoExpert\src>go build -gcflags=-m
# _/D_/SourceCode/GoExpert/src
.\main.go:7: can inline Fibonacci.func1
.\main.go:7: func literal escapes to heap
.\main.go:7: func literal escapes to heap
.\main.go:8: &a escapes to heap
.\main.go:6: moved to heap: a
.\main.go:8: &b escapes to heap
.\main.go:6: moved to heap: b
.\main.go:17: f() escapes to heap
.\main.go:17: main ... argument does not escape
```

逃逸总结

- 栈上分配内存比在堆中分配内存有更高的效率
- 栈上分配的内存不需要GC处理
- 堆上分配的内存使用完毕会交给GC处理
- 逃逸分析目的是决定内分配地址是栈还是堆
- 逃逸分析在编译阶段完成