我们应该知道的官方标准 Go编译器中做出的一些优化

Go 夜读 SIG 小组

2020-08-19

官方标准Go编译器

- 简称gc（Go compiler，非Garbage Collection）
- 另一款官方Go编译器为gccgo，主要做为一个参考编译器，帮助发现gc中的bugs和改善官方文档。目前从编译速度和编译出的代码质量（正确性和执行速度）都大大落后于gc。
- Go设计中推荐使用编译器优化来弥补一些语言中缺失的小功能。

（本分享将Go runtime看做是编译器的一部分。另外限于时间，具体哪个Toolchain版本引入的优化不能一一确认。）

字符串和字节切片之间的转换

一般情况下，这样的转化需要复制一份底层的字节序列。但是gc编译器做了一些优化，使得下面这些情况下的这样的转化无需复制底层的字节序列：

1. 紧跟range关键字的从字符串到字节切片的转换；
2. 映射元素读取索引语法中被用做键值的从字节切片到字符串的转换；
3. 字符串比较表达式中被用做比较值的从字节切片到字符串的转换；
4. （至少有一个被衔接的字符串值为非空字符串常量的）字符串衔接表达式中的从字节切片到字符串的转换。

优化1：紧跟range关键字的从字符串到字节切片的转换

```
var gogogo = strings.Repeat("Go", 1024)

func f() { for range []byte(gogogo) {} }
func g() { bs := []byte(gogogo);   for range bs {} }

func main() {
    fmt.Println(testing.AllocsPerRun(1, f)) // 0
    fmt.Println(testing.AllocsPerRun(1, g)) // 1
}
```

优化2：映射元素读取索引语法中被用做键值的从字节切片到字符串的转换

```
var name = bytes.Repeat([]byte{'x'}, 33)
var m, s = make(map[string]string, 10), ""

func f() { s = m[string(name)] } // 有效
func g() { key := string(name); s = m[key] } // 无效
func h() { m[string(name)] = "Golang" } // 无效

    fmt.Println(testing.AllocsPerRun(1, f)) // 0
    fmt.Println(testing.AllocsPerRun(1, g)) // 1
    fmt.Println(testing.AllocsPerRun(1, h)) // 1
```

优化3：一个字符串比较表达式中被用做比较值的从字节切片到字符串的转换

```
var x = []byte{1023: 'x'}
var y = []byte{1023: 'y'}
var b bool

func f() { b = string(x) != string(y) }
func g() { sx, sy := string(x), string(y); b = sx == sy }

    fmt.Println(testing.AllocsPerRun(1, f)) // 0
    fmt.Println(testing.AllocsPerRun(1, g)) // 2
```

优化4：含非空字符串常量的字符串衔接表达式中的从字节切片到字符串的转换

```
var x = []byte{1023: 'x'}
var y = []byte{1023: 'y'}
var s string

func f() { s = ("-" + string(x) + string(y))[1:] }
func g() { s = string(x) + string(y) }

    fmt.Println(testing.AllocsPerRun(1, f)) // 1
    fmt.Println(testing.AllocsPerRun(1, g)) // 3
```

优化5：[]rune(aString)转换的时间和空间复杂度都是O(n)，但是len([]rune(aString))中的此转换不需开辟内存

```
var GoGoGo = strings.Repeat("Go", 100)

func f() { _ = len([]rune(GoGoGo)) }
func g() { _ = len([]byte(GoGoGo)) } // 未对len([]byte(aString))做优化

func main() {
    fmt.Println(testing.AllocsPerRun(1, f)) // 0
    fmt.Println(testing.AllocsPerRun(1, g)) // 1
}
```

优化6：字符串衔接表达式只需开辟一次内存，无论需要衔接多少个字符串

```
var x, y, z, w = "Hello ", "World! ", "Let's ", "Go!"
var s string

func f() { s = x + y + z + w }
func g() { s = x + y; s += z; s += w }

func main() {
    fmt.Println(testing.AllocsPerRun(1, f)) // 1
    fmt.Println(testing.AllocsPerRun(1, g)) // 3
}
```

对于在编译时刻衔接的字符串的数量已知的情况下，此种方法衔接字符串的效率最高。

优化7：for i := range anArrayOrSlice {anArrayOrSlice[i] = zeroElement}形式将被优化为一个内部的memclr操作

```
const N = 1024 * 100

var a [N]int


func clearArray() { for i := range a { a[i] = 0 } }

func clearSlice() { s := a[:]; for i := range s { s[i] = 0 } }

func clearArrayPtr() { for i := range &a { a[i] = 0 } } // 无效
```

```
Benchmark_clearArray-4          77971          14698 ns/op
Benchmark_clearSlice-4          76803          14771 ns/op
Benchmark_clearArrayPtr-4       30687          39002 ns/op
```

优化8：for k = range m {delete(m, k)}形式将被优化为一个内部的map清空操作

这个优化貌似对于平时编程并没有太大的意义，因为这是清空map条目的唯一方法。但是其实有些Go程序员可能会通过用make来新开出来一个map的途径来变相清空map条目。

其实两种方法各有所长。目前官方标准编译器的实现中，一个map的底层哈希表数组的长度是永不收缩的。所以这个优化并不释放为底层哈希表数组开辟的内存。它只是比一个一个删除操作要快得多。

相比用make来新开出来一个map，此优化将减少一些GC（垃圾回收）压力。所以，具体应该使用哪种方法，视具体情况而定。

优化9：尺寸不大于4个原生字（即int）并且字段数不超过4个的结构体值被视为是小尺寸值

```
type S1 struct{a int}
type S2 struct{a, b int}
type S3 struct{a, b, c int}
type S4 struct{a, b, c, d int}
type S5 struct{a, b, c, d, e int}
type S6 struct{a, b, c, d, e, f int}
var ss1, ss2, ss3, ss4, ss5, ss6 =
    make([]S1, 1000), make([]S2, 1000),
    make([]S3, 1000), make([]S4, 1000),
    make([]S5, 1000), make([]S6, 1000)
var x1, x2, x3, x4, x5, x6 int
```

```
Benchmark_Range1-4   3057718  389 ns/op
Benchmark_Range2-4   3187244  375 ns/op
Benchmark_Range3-4   3146187  377 ns/op
Benchmark_Range4-4   3117408  383 ns/op
Benchmark_Range5-4    483148 2462 ns/op
Benchmark_Range6-4    289340 3571 ns/op
```

```
func Benchmark_Range1(b *testing.B) {
    for i := 0; i < b.N; i++ {
                for _, v := range ss1 { x1 = v.a }
    }
}
```

优化10：接口值包裹指针值比包裹其它类型的值要快

```
var p, p2 = new([100]int), new([100]int)
var ip interface{}

func Benchmark_BoxPointer(b *testing.B) {
    for i := 0; i < b.N; i++ { ip = p }
}
```

```
func Benchmark_PointerAssert(b *testing.B) {
    for i := 0; i < b.N; i++ { p = ip.(*[100]int) }
}

func Benchmark_PointerAssign(b *testing.B) {
    for i := 0; i < b.N; i++ { p = p2 }
}
```

```
Benchmark_BoxPointer-4        1000000000    0.818 ns/op
Benchmark_PointerAssert-4 1000000000        0.813 ns/op
Benchmark_PointerAssign-4 1000000000        0.547 ns/op
```

优化10：接口值包裹指针值比包裹其它类型的值要快，因为少开辟一次内存

```
var p, x, y, z = new([]int), 256, "Go", []int{0}
var ip, ix, iy, iz interface{}

func Benchmark_BoxPointer(b *testing.B) {
    for i := 0; i < b.N; i++ { ip = p } // pointer
}
```

```
func Benchmark_BoxInt(b *testing.B) {
    for i := 0; i < b.N; i++ { ix = x } // int
}
func Benchmark_BoxString(b *testing.B) {
    for i := 0; i < b.N; i++ { iy = y } // string
}
func Benchmark_BoxSlice(b *testing.B) {
    for i := 0; i < b.N; i++ { iz = z } // slice
}
```

```
Benchmark_BoxPointer-4      0.818 ns/op    0 allocs/op
Benchmark_BoxInt-4          17.6 ns/op     1 allocs/op
Benchmark_BoxString-4       30.9 ns/op     1 allocs/op
Benchmark_BoxSlice-4        41.9 ns/op     1 allocs/op
```

优化11：接口值包裹0-255的整数也很快

```
var x, y = 255, 256
var iy, ix interface{}

func Benchmark_x(b *testing.B) {
    for i := 0; i < b.N; i++ { ix = x }
}

func Benchmark_y(b *testing.B) {
    for i := 0; i < b.N; i++ { iy = y }
}
```

从Go工具链1.15开始，个Go运行时内部维护着一个0到255的小数组。但包裹0-255的整数时，将直接包裹此数组的相应元素的指针而少开辟一次内存。

但是比直接包裹指针还是慢一点。

```
Benchmark_x-4          2.98 ns/op     0 allocs/op
Benchmark_y-4          17.0 ns/op     1 allocs/op
```

BCE（Bounds Check Elimination）优化

Go是一门内存安全的语言。检查下标越界是保证内存安全的重要举措之一。但另一方面检查下标越界也耗费一些CPU计算。事实上绝大部分的下标越界检查都不会发现有问题的。这就是维护内存安全的代价。

在某些情形下，编译器在代码编译阶段可以确定某些下标越界检查是不必要的从而可以避免这些检查，这样将提升程序运行效率。

编译器并不总是足够得聪明，有时需要人为干预引导编译器来消除一些下标越界检查。

优化12：Bounds Check Elimination

```
func f1(s []int) {
    _ = s[0]
    _ = s[1]
    _ = s[2]
}

func f2(s []int) {
    _ = s[2]
    _ = s[1]
    _ = s[0]
}
```

```
func f9(s []int) {
    if len(s) > 2 {
        _, _, _ = s[0], s[1], s[2]
    }
}

func f4(a [5]int) {
    _ = a[4]
}
```

```
func f8(s []int, index int) {
    if index >= 0 && index < len(s) {
        _ = s[index]
        _ = s[index:len(s)]
    }
}

func f3(s []int, index int) {
    _ = s[index]
    _ = s[index]
}
```

```
go build -gcflags="-d=ssa/check_bce/debug=1" [more...]
```

优化12：Bounds Check Elimination

```
func f5(s []int) {
    for i := range s {
        _ = s[i]
        _ = s[i:len(s)]
        _ = s[:i+1]
    }
}
```

```
func f6(s []int) {
    for i := 0; i < len(s); i++ {
        _ = s[i]
        _ = s[i:len(s)]
        _ = s[:i+1]
    }
}
```

```
func f7(s []int) {
    for i := len(s) - 1; i >= 0; i-- {
        _ = s[i]
        _ = s[i:len(s)]
    }
}
```

优化12：Bounds Check Elimination （手动干预，Go官方编译器1.15）

```
func fd(is []int, bs []byte) {
    if len(is) >= 256 {
        for _, n := range bs {
            _ = is[n]
        }
    }
}
```

```
func fd2(is []int, bs []byte) {
    if len(is) >= 256 {
        is = is[:256] // 给编译器一个暗示
        for _, n := range bs {
            _ = is[n]
        }
    }
}
```

优化12：Bounds Check Elimination （手动干预，Go官方编译器1.15）

```
func fe(isa []int, isb []int) {
    if len(isa) > 0xFFF {
        for _, n := range isb {
            _ = isa[n & 0xFFF]
        }
    }
}
```

```
func fe2(isa []int, isb []int) {
    if len(isa) > 0xFFF {
        isa = isa[:0xFFF+1] // 给编译器一个暗示
        for _, n := range isb {
            _ = isa[n & 0xFFF]
        }
    }
}
```

优化12：Bounds Check Elimination （手动干预，Go官方编译器1.15）

```
func fp(x, y string) int {
    if len(x) > len(y) {
        x, y = y, x
    }
    for i := 0; i < len(x); i++ {
        if x[i] != y[i] {
            return i
        }
    }
    return len(x)
}
```

```
BenchmarkFP-4   287 ns/op
BenchmarkFQ-4   160 ns/op
```

```
func fq(x, y string) int {
    if len(x) > len(y) {
        x, y = y, x
    }
    if len(x) <= len(y) { // 给编译器一个暗示
        for i := 0; i < len(x); i++ {
            if x[i] != y[i] {
                return i
            }
        }
    }
    return len(x)
}
```

参考资料

1. Go 101项目：https://github.com/golang101/golang101
2. Go 101官网：https://gfw.go101.org
3. Go 101公众号