初探 golang 編譯器優化

...

Peter Lai

Golang Taipei Gathering #63 Webinar

Outline

- History
- Escape Analysis
- Inlining
- Dead Code Eliminiation
- Bounds Check Eliminiation
- Conclusion

History

- Website: https://golang.design/history/#compiler
- Original forker from Plan9 compiler toolchain written in C
- Rewrite toolchain into Go from 1.5
- Transformed to SSA backend in Go 1.7
  - Static single assignment (靜態單賦值形式)
  - Code: https://tinyurl.com/5bj4n26t
  - **GOSSAFUNC=Foo{+} go build**
  - Enable some optimizations
    - Dead code elimination
    - Removal of unused branches

```
b1:
  v1 (?) = InitMem <mem>
  v2 (?) = SP <uintptr>
  v3 (?) = SB <uintptr>
  v4 (?) = GetClosurePtr <*uint8>
  v5 (?) = Const64 <int> [100] (a[int])
  v6 (?) = Const64 <int> [20] (b[int])
  v7 (?) = Const64 <int> [0] (~R0[int])
  v8 (+16) = InlMark <void> [0] v1
  v9 (8) = Less64 <bool> v6 v5
  v14 (?) = Addr <*uint8> {type.int} v3
  v15 (?) = Addr <*int> {""..stmp_0} v3
If v9 → b3 b2 (8)
```

History

- Parsing
  - Lexical analysis
  - Syntax analysis
  - Syntax tree
- Type-checking and AST transformations
  - Type-checking
  - Convert the syntax package's syntax tree to the compiler's AST representation
- Generic SSA
- Generating machine code

Escape Analysis

- Stack allocations are significantly cheaper for both memory allocator and garbage collector
- In Go, the compiler automatically moves a value to the heap if it lives beyond the lifetime of the function call. It is said that the value escapes to the heap
- Value was referenced by other might be escaped

Escape Analysis

- Return value will escape to heap if it's pointer
- go build -gcflags="-m" foo.go

```go
3   type Foo struct {
4       a, b, c, d, e int
5   }
6
7   func NewFoo() *Foo {
8       return &Foo{1, 2, 3, 4, 5}
9   }
10
11  func main() {
12      NewFoo()
13  }
```

Escape Analysis

- Slice won’t esape if it’s constant (number won’t escape)
- go build -gcflags="-m" sum.go

```go
6   func Sum() int {
7       const count = 100
8       numbers := make([]int, count)
9       for i := range numbers {
10          numbers[i] = i + 1
11      }
12
13      var sum int
14      for _, i := range numbers {
15          sum += i
16      }
17      return sum
18  }
```

Escape Analysis

- Slice will esape if it’s not constant (number willescape)
- go build -gcflags="-m" sum.go

```go
6   func Sum(count int) int {
7       numbers := make([]int, count)
8       for i := range numbers {
9           numbers[i] = i + 1
10      }
11
12      var sum int
13      for _, i := range numbers {
14          sum += i
15      }
16      return sum
17  }
```

Escape Analysis

- Value will esape if it’s referenced by map
- go build -gcflags="-m" fetch.go

```go
7  func Fetch(k int) *int {
8      v := make(map[*int]*int)
9      ke := 0
10     va := 0
11     v[&ke] = &va
12     return v[&k]
13 }
```

Escape Analysis

- Jalex Chang - https://tinyurl.com/2p8eaha5
- Pointer will escape
- Slice will esape if it’s not constant
- Value will esape if it’s referenced by map

Inlining

- Function calls have overhead (stack and preemption checks)
- Enables other optimizations, like Loop-Invariant Code Motion
- Short and simple functions are inlined (https://tinyurl.com/yckztves):
  - Function should be simple enough, the number of AST nodes must less than the budget (80)
  - Function doesn't contain complex things like closures, defer, recover, select, etc
  - Function isn't prefixed by go:noinline / go:norace / go:nocheckptr / go:cgo_unsafe_args / go:uintptrescapes || use –race / -d
  - Function has body

```
for x := range xs {
    s += Sqrt(x*x + C*C)
}
```

```
Z := C*C
for x := range xs {
    s += Sqrt(x*x + Z)
}
```

Inlining

- go build -gcflags="-m" max.go
- # command-line-arguments
- ./max.go:3:6: can inline Max
- ./max.go:10:6: can inline F
- ./max.go:12:8: inlining call to Max
- ./max.go:17:6: can inline main
- ./max.go:18:3: inlining call to F
- ./max.go:18:3: inlining call to Max

```go
7   func Max(a, b int) int {
8       if a > b {
9           return a
10      }
11      return b
12  }
13
14  func F() {
15      const a, b = 100, 20
16      if Max(a, b) == b {
17          panic(b)
18      }
19  }
```

Inlining

- go build -gcflags=-S ./max.go 2>&1 | grep -A5 '""".F STEXT'

```
""".F STEXT nosplit size=1 args=0x0 locals=0x0 funcid=0x0
        0x0000 00000 (/max.go:10)  TEXT        """.F(SB), NOSPLIT|ABIInternal, $0-0
        0x0000 00000 (/max.go:10)  FUNCDATA      $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x0000 00000 (/max.go:10)  FUNCDATA      $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
        0x0000 00000 (/max.go:15)  RET
        0x0000 c3
```

Inlining

- -gcflags=-l, inlining disabled.
- -gcflags='-l=1' or nothing, regular inlining (default option).
- *-gcflags='-l -l' inlining level 2, more aggressive, might be faster, may make bigger binaries.*
- *-gcflags='-l -l -l' inlining level 3, more aggressive again, binaries definitely bigger, maybe faster again, but might also be buggy.*
- -gcflags=-l=4 (four `-l`s) in Go 1.11 start enable the experimental mid-stack inlining optimisation. (non-leaf function)

Inlining

- https://github.com/golang/go/blob/80a7504a13a5dccb60757d1fc66d71bcba359799/src/cmd/compile/internal/inline/inl.go#L10
- // The Debug.l flag controls the aggressiveness. Note that main() swaps level 0 and 1,
- // making 1 the default and -l disable. Additional levels (beyond -l) may be buggy and
- // are not supported.
- // 0: disabled
- // 1: 80-nodes leaf functions, oneliners, panic, lazy typechecking (default)
- // 2: (unassigned)
- // 3: (unassigned)
- // 4: allow non-leaf functions

Dead Code Elimination

- Remove non recheable Blocks in SSA
- Because a, b is constant, compiler can do optimization
- https://tinyurl.com/yc86hw9u

```go
func Max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

func F() {
    const a, b = 100, 20
    if Max(a, b) == b {
        panic(b)
    }
}
```

Dead Code Elimination

```go
func F() {
    const a, b = 100, 20
    var result int
    if a > b {
        result = a
    } else {
        result = b
    }
    if result == b {
        panic(b)
    }
}
```

```go
func F() {
    const a, b = 100, 20
    var result int
    if true {
        result = a
    } else {
        result = b
    }
    if result == b {
        panic(b)
    }
}
```

Dead Code Elimination

```go
func F() {
    const a, b = 100, 20
    const result = a
    if result == b {
        panic(b)
    }
}
```

```go
func F() {
}
```

Bounds Check Elimination

- In go, array and slice subscript operations are checked to ensure they are within the bounds of the respective types
- Normally a go program will panic when a slice or a string is accessed outside of its bounds
- For arrays, this can be done at compile time, for slices, this must be done at runtime
- go build -gcflags="-d=ssa/check_bce/debug=1" ./t.test.go (Go 1.7+)
- Disable bounds checking: -gcflags=-B

Bounds Check Elimination

- go build -gcflags="-d=ssa/check_bce/debug=1" ./bce.go
- ./bce.go:11:8: Found IsInBounds
- ./bce.go:12:8: Found IsInBounds
- ./bce.go:13:8: Found IsInBounds
- ./bce.go:14:8: Found IsInBounds
- ...

```go
5   var v = make([]int, 9)
6
7   var A, B, C, D, E, F, G, H, I int
8
9   func main() {
10      for n := 0; n < 1; n++ {
11          A = v[0]
12          B = v[1]
13          C = v[2]
14          D = v[3]
15          E = v[4]
16          F = v[5]
17          G = v[6]
18          H = v[7]
19          I = v[8]
20      }
21  }
```

Bounds Check Elimination

- go build -gcflags="-d=ssa/check_bce/debug=1" ./bce.go
- No bounds check

```go
var v [9]int

var A, B, C, D, E, F, G, H, I int

func main() {
    for n := 0; n < 1; n++ {
        A = v[0]
        B = v[1]
        C = v[2]
        D = v[3]
        E = v[4]
        F = v[5]
        G = v[6]
        H = v[7]
        I = v[8]
    }
}
```

Bounds Check Elimination

- go build -gcflags="-d=ssa/check_bce/debug=1" ./bce.go
- No bounds check

```
31    for i := 0; i < len(v); i++ {
32        I = v[i]
33    }
```

```
35    for _, va := range v {
36        I = va
37    }
```

Bounds Check Elimination

- Duplicate checks
- Constant index and constant size
- Decreasing constant indexes

Conclusion

- Be careful when use pointers
- Enable inline optimization and write small function

Q & A

Reference

- Original post: https://tinyurl.com/4dwu77d2
- Translation: https://tinyurl.com/mwpfx56t
- Compiler history of Golang: https://golang.design/history/#compiler
- Github wiki: https://tinyurl.com/2p9hfu82
- Escape Analysis Presentation: https://tinyurl.com/2p8eaha5
- Mid-stack inlining in Golang Presentation: https://tinyurl.com/b8s43te7
- Compile package: https://tinyurl.com/nrybyacj
- Utilize Go 1.7 SSA Compiler: https://tinyurl.com/4fd5sk6f