Tips and Tricks for High performance Go

20 June 2019

Erik Dubbelboer
Senior Developer, poki.com
Co-founder/CTO, atomx.com
Maintainer, github.com/valyala/fasthttp

Benchmarking in Go

```
$ go test -bench . -benchmem
```

```
// slice_test.go
func BenchmarkNormal(b *testing.B) {
    for i := 0; i < b.N; i++ {
        // Code you want to benchmark here...
        // This is counted as one "op".
    }
}
```

4

Benchmarking in Go

```
func appendItems(s []int) []int {
    for i := 0; i < 10000; i++ {
        s = append(s, i)
    }
    return s
}

func BenchmarkNormal(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0))
    }
}

func BenchmarkPreallocate(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0, 10000))
    }
}
```

- slices grow exponentially (https://github.com/golang/go/blob/323212b9e6edd55e99d973d00d2132995762c858/src/runtime/slice.go#L96-L114) *ish*

5

Benchmarking in Go

```
func appendItems(s []int) []int {
    for i := 0; i < 10000; i++ {
        s = append(s, i)
    }
    return s
}

func BenchmarkNormal(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0))
    }
}

func BenchmarkPreallocate(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0, 10000))
    }
}
```

```
$ go test -bench . -benchmem
BenchmarkNormal-16          50000    33027 ns/op    386297 B/op
BenchmarkPreallocate-16    200000     8219 ns/op     81920 B/op
```

6

Benchmarking in Go

```
func appendItems(s []int) []int {
    for i := 0; i < 10000; i++ {
        s = append(s, i)
    }
    return s
}

func BenchmarkNormal(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0))
    }
}

func BenchmarkPreallocate(b *testing.B) {
    for i := 0; i < b.N; i++ {
        appendItems(make([]int, 0, 10000))
    }
}
```

```
GOGC=1 go test -bench . -benchmem
BenchmarkNormal-16                 10000        107718 ns/op
BenchmarkPreallocate-16           100000         20040 ns/op
```

7

Unrolling loops

```
func sumFloat64(s []float64) float64 {
    a := float64(0)
    for _, f := range s {
        a += f
    }
    return a
}
```

8

Unrolling loops

```
func sumFloat64Unrolled(s []float64) float64 {
    var a1, a2, a3, a4 float64
    l := len(s)
    lm := l % 4

    for i := lm; i < l; i += 4 {
        a1 += s[i]
        a2 += s[i+1]
        a3 += s[i+2]
        a4 += s[i+3]
    }
    if lm == 1 {
        a1 += s[0]
    } else if lm == 2 {
        a1 += s[0]
        a2 += s[1]
    } else if lm == 3 {
        a1 += s[0]
        a2 += s[1]
        a3 += s[2]
    }

    return a1 + a2 + a3 + a4
}
```

9

Unrolling loops

```
$ go test -bench . -benchmem
BenchmarkSum-16                 5000000        234   ns/op
BenchmarkSumUnrolled-16         20000000        96.4 ns/op
```

10

Removing bounds checks

```
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
a4 += s[i+3]
```

- Bounds check on every line!

11

Removing bounds checks

```
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
a4 += s[i+3]
```

Reverse order?

```
a4 += s[i+3]
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
```

12

Removing bounds checks

```
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
a4 += s[i+3]
```

Move the bounds check up!

```
_ = s[i+3]
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
a4 += s[i+3]
```

Real life example in encoding/binary.LittleEndian.Uint64([]byte) uint64

(https://github.com/golang/go/blob/7a4d02387fa16cd2a88c30357346e5cf0ae282b1/src/encoding/binary/binary.go#L76)13

Removing bounds checks

```
a1 += s[i]
a2 += s[i+1]
a3 += s[i+2]
a4 += s[i+3]
```

Remove bounds checks completely!

```
ss := (*[4]float64)(unsafe.Pointer(&s[i]))
a1 += ss[0]
a2 += ss[1]
a3 += ss[2]
a4 += ss[3]
```

Removing bounds checks

```
$ go test -bench . -benchmem
BenchmarkSum-16                     5000000        234 ns/op
BenchmarkSumUnrolled-16            20000000       96.4 ns/op
BenchmarkSumUnrolledNoBounds-16    20000000       67.8 ns/op
```

15

Escape analysis

```
1  package main
2
3  func doit(x *int) {
4      _ = x
5  }
6
7  func main() {
8      y := 1
9      doit(&y)
10 }
```

```
$ go run -gcflags '-m -m -l' heap.go
./heap.go:3:11: doit x does not escape
./heap.go:9:7: main &y does not escape
```

16

Escape analysis

```
1  package main
2
3  var g *int
4
5  func doit(x *int) {
6      g = x
7  }
8
9  func main() {
10      y := 1
11      doit(&y)
12  }
```

```
$ go run -gcflags '-m -m -l' stack.go
./stack.go:5:11: leaking param: x
./stack.go:5:11:     from g (assigned to top level variable) at .
./stack.go:11:7: &y escapes to heap
./stack.go:11:7:     from &y (passed to call[argument escapes]) a
./stack.go:10:2: moved to heap: y
```

17

Why is garbage bad?

```
var heapSink *int64
var stackSink int64

func BenchmarkHeap(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for j := 0; j < 100; j++ {
            x := int64(i)
            heapSink = &x
        }
    }
}

func BenchmarkStack(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for j := 0; j < 100; j++ {
            x := int64(i)
            stackSink = x
        }
    }
}
```

18

Why is garbage bad?

```
var heapSink *int64
var stackSink int64

func BenchmarkHeap(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for j := 0; j < 100; j++ {
            x := int64(i)
            heapSink = &x
        }
    }
}

func BenchmarkStack(b *testing.B) {
    for i := 0; i < b.N; i++ {
        for j := 0; j < 100; j++ {
            x := int64(i)
            stackSink = x
        }
    }
}
```

```
$ GOGC=1 go test -bench . -benchmem
BenchmarkHeap-16          2000000          6287   ns/op         8
BenchmarkStack-16       500000000            30.7 ns/op
                                                              19
```

sync.Pool

```
package sync

type Pool struct {
    // New optionally specifies a function to generate
    // a value when Get would otherwise return nil.
    // It may not be changed concurrently with calls to Get.
    New func() interface{}
}

func (p *Pool) Get() interface{}

func (p *Pool) Put(x interface{})
```

20

sync.Pool

```
func BenchmarkNoPool(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            bb := &bytes.Buffer{}
            for j := 0; j < 1000; j++ {
                bb.Write([]byte{0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
            }
        }
    })
}
```

21

sync.Pool

```
func BenchmarkPool(b *testing.B) {
    p := sync.Pool{
        New: func() interface{} {
            return &bytes.Buffer{}
        },
    }

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            bb := p.Get().(*bytes.Buffer)
            bb.Reset() // Don't forget to reset the contents!
            for j := 0; j < 1000; j++ {
                bb.Write([]byte{0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
            }
            p.Put(bb)
        }
    })
}
```

22

sync.Pool

```
$ GOGC=1 go test -bench . -benchmem
BenchmarkNoPool-16          200000          7375 ns/op        3851
BenchmarkPool-16           1000000          1047 ns/op
```

- Even faster in 1.13 (no more full clear on GC (https://github.com/golang/go/issues/22950))

string/[]byte conversions

```
type SliceHeader struct {
        Data uintptr
        Len  int
        Cap  int
}
```

```
type StringHeader struct {
        Data uintptr
        Len  int
}
```

24

string/[]byte conversions

```
func BenchmarkSlow(b *testing.B) {
    input := []byte{0x31, 0x33, 0x33, 0x37}

    for i := 0; i < b.N; i++ {
        strconv.Atoi(string(input))
    }
}
```

25

string/[]byte conversions

```
func b2s(b []byte) string {
    return *(*string)(unsafe.Pointer(&b))
}
```

```
func BenchmarkFast(b *testing.B) {
    input := []byte{0x31, 0x33, 0x33, 0x37}

    for i := 0; i < b.N; i++ {
        strconv.Atoi(b2s(input))
    }
}
```

26

string/[]byte conversions

```
func b2s(b []byte) string {
    return *(*string)(unsafe.Pointer(&b))
}
```

```
func BenchmarkFast(b *testing.B) {
    input := []byte{0x31, 0x33, 0x33, 0x37}

    for i := 0; i < b.N; i++ {
        strconv.Atoi(b2s(input))
    }
}
```

```
$ GOGC=1 go test -bench . -benchmem
BenchmarkSlow-16        30000000            40.3  ns/op
BenchmarkFast-16        300000000            4.83 ns/op
```

27

string/[]byte conversions

```
func BenchmarkSlow(b *testing.B) {
    input := "this is a test"

    for i := 0; i < b.N; i++ {
        ioutil.Discard.Write([]byte(input))
    }
}
```

28

string/[]byte conversions

```
func s2b(s string) []byte {
    sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
    bh := reflect.SliceHeader{
        Data: sh.Data,
        Len:  sh.Len,
        Cap:  sh.Len,
    }
    return *(*[]byte)(unsafe.Pointer(&bh))
}
```

```
func BenchmarkFast(b *testing.B) {
    input := "this is a test"

    for i := 0; i < b.N; i++ {
        ioutil.Discard.Write(s2b(input))
    }
}
```

29

string/[]byte conversions

```
func s2b(s string) []byte {
    sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
    bh := reflect.SliceHeader{
        Data: sh.Data,
        Len:  sh.Len,
        Cap:  sh.Len,
    }
    return *(*[]byte)(unsafe.Pointer(&bh))
}
```

```
func BenchmarkFast(b *testing.B) {
    input := "this is a test"

    for i := 0; i < b.N; i++ {
        ioutil.Discard.Write(s2b(input))
    }
}
```

```
$ GOGC=1 go test -bench . -benchmem
BenchmarkSlow-16        20000000           102    ns/op
BenchmarkFast-16        500000000            3.67 ns/op
```

30

Struct packing

```
type TooBig struct {
    a [14]int64 // 112 (14*8) bytes

    b int32   // 4
    c float64 // 8
    d int32   // 4
}
```

31

Struct packing

```
type TooBig struct {
    a [14]int64 // 112 (14*8) bytes

    b int32   // 4
    c float64 // 8
    d int32   // 4
}
```

```
type JustRight struct {
    a [14]int64 // 112 (14*8) bytes

    b int32   // 4
    d int32   // 4
    c float64 // 8
}
```

32

Struct packing

```
func BenchmarkTooBig(b *testing.B) {
    m := make(map[int]TooBig)

    for i := 0; i < b.N; i++ {
        m[i] = TooBig{}
    }
}
```

```
func BenchmarkJustRight(b *testing.B) {
    m := make(map[int]JustRight)

    for i := 0; i < b.N; i++ {
        m[i] = JustRight{}
    }
}
```

33

Struct packing

```
func BenchmarkTooBig(b *testing.B) {
    m := make(map[int]TooBig)

    for i := 0; i < b.N; i++ {
        m[i] = TooBig{}
    }
}
```

```
func BenchmarkJustRight(b *testing.B) {
    m := make(map[int]JustRight)

    for i := 0; i < b.N; i++ {
        m[i] = JustRight{}
    }
}
```

```
$ go test -bench . -benchmem
BenchmarkTooBig-16                 5000000       362 ns/op
BenchmarkJustRight-16              5000000       444 ns/op
```

34

Struct packing

```
func BenchmarkTooBig(b *testing.B) {
    m := make(map[int]TooBig)

    for i := 0; i < b.N; i++ {
        m[i] = TooBig{}
    }
}
```

```
func BenchmarkJustRightPrealloc(b *testing.B) {
    m := make(map[int]JustRight, b.N)

    for i := 0; i < b.N; i++ {
        m[i] = JustRight{}
    }
}
```

```
$ go test -bench . -benchmem
BenchmarkTooBig-16                 5000000       362 ns/op
BenchmarkJustRight-16              5000000       444 ns/op
BenchmarkJustRightPrealloc-16     10000000       340 ns/op
```

35

Thank you

Erik Dubbelboer
Senior Developer, poki.com
Co-founder/CTO, atomx.com
Maintainer, github.com/valyala/fasthttp
erik@dubbelboer.com (mailto:erik@dubbelboer.com)
https://github.com/erikdubbelboer (https://github.com/erikdubbelboer)