Programming in Go

Matt Holiday
Christmas 2020

Benchmarking

Go benchmarks

Go has standard tools and conventions for running benchmarks

Benchmarks live in test files ending with `_test.go`

You run benchmarks with `go test -bench`

Go only runs the `BenchmarkXXX` functions

Simple example function

```
func Fib(n int, recursive bool) int {
    switch n {
    case 0:
        return 0
    case 1:
        return 1
    default:
        if recursive {
            return Fib(n-1, r) + Fib(n-2, r)
        }
        a, b := 0, 1
        for i := 1; i < n; i++ {
            a, b = b, a+b
        }
        return b
    }
}
```

Simple benchmark tests

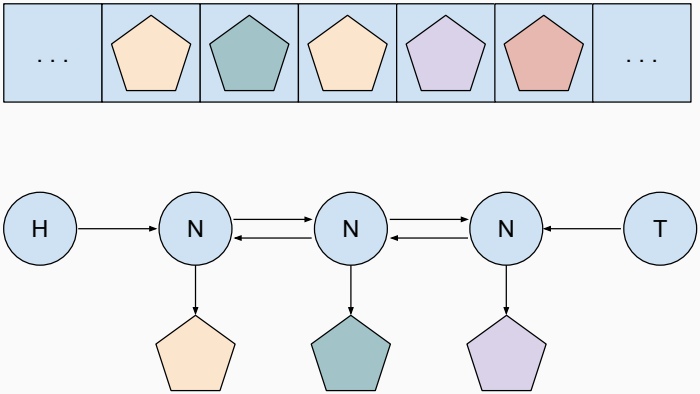
```go
func BenchmarkFib20T(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(20, true)  // run the Fib function b.N times
    }
}

func BenchmarkFib20F(b *testing.B) {
    for n := 0; n < b.N; n++ {
        Fib(20, false) // run the Fib function b.N times
    }
}
```

```
$ go test -bench=. ./fib_test.go
goos: darwin
goarch: amd64
BenchmarkFib20T-16         20851         59634 ns/op
BenchmarkFib20F-16      94855990            12.6 ns/op
```

List vs Slice example

A slice of objects beats a list with pointers



List vs Slice example

```
package main
import "testing"

type node struct {
    v int           // value in the list node
    t *node
}

func insert(i int, h *node) *node {
    t := &node{i, nil}

    if h != nil {
        h.t = t
    }

    return t
}
```

List vs Slice example

```
func mkList(n int) *node {
    var h, t *node

    h = insert(0, h)
    t = insert(1, h)
    for i := 2; i < n; i ++ {
        t = insert(i, t)
    }
    return h
}

func sumList(n *node) (i int) {
    for h := n; h != nil; h = h.t {
        i += h.v
    }
    return
}
```

List vs Slice example

```
func mkSlice(n int) []int {
    r := make([]int, n)

    for i := 0; i < n; i++ {
        r[i] = i
    }

    return r
}

func sumSlice(l []int) (i int) {
    for _, v := range l {
        i += v
    }

    return
}
```

List vs Slice example

```
func BenchmarkList(b *testing.B) {
    for n := 0; n < b.N; n++ {
        l := mkList(1200)
        sumList(l)
    }
}

func BenchmarkSlice(b *testing.B) {
    for n := 0; n < b.N; n++ {
        l := mkSlice(1200)
        sumSlice(l)
    }
}
```

```
$ go test -bench=. ./list_test.go
BenchmarkList-16           35452          33904 ns/op
BenchmarkSlice-16         769028           1555 ns/op
```

List vs Slice example

In this example, we'll separate out the cost of making the list

```
func BenchmarkList(b *testing.B) {
    l := mkList(1200); b.ResetTimer()
    for n := 0; n < b.N; n++ {
        sumList(l)
    }
}

func BenchmarkSlice(b *testing.B) {
    l := mkSlice(1200); b.ResetTimer()
    for n := 0; n < b.N; n++ {
        sumSlice(l)
    }
}
```

```
BenchmarkList-16          885607          1243 ns/op
BenchmarkSlice-16        2910057           414 ns/op
```

List vs Slice example

In this version, we keep the value separate from the list node

```
type node struct {
    v *int
    t *node
}

func insert(i int, n *node) *node {
    t := &node{&i, nil}

    if n != nil {
        n.t = t
    }

    return t
}
```

List vs Slice example

In this version, we keep the value separate from the list node

```
func sumList(n *node) (i int) {
    for h := n; h != nil; h = h.t {
        i += *h.v
    }
    return
}
```

```
BenchmarkList-16          707344          1596 ns/op
BenchmarkSlice-16        2883547           417 ns/op
```

What if we go back and include the cost of building the list?

```
BenchmarkList-16           24858         48690 ns/op
BenchmarkSlice-16         662131          1518 ns/op
```

List vs Slice example

We can extend the benchmark to include memory allocations using the `-benchmem` flag

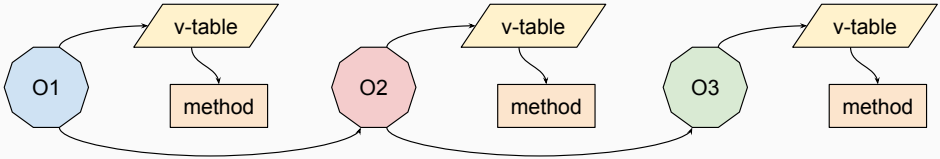
This version is with the list building included (since that's where the allocation takes place)

We don't just allocate more, we do it in smaller chunks

```
$ go test -bench=. -benchmem ./list_test.go
goos: darwin
goarch: amd64
BenchmarkList-16    23977  49766 ns/op  28800 B/op  2400 allocs/op
BenchmarkSlice-16  741272   1515 ns/op   9728 B/op     1 allocs/op
```

Forwarding example

Calling lots short methods via dynamic dispatch is very expensive



The cost of calling a function should be proportional to the work it does
(short inline functions vs longer methods with late binding)

Forwarding example

```
package forward
import ( "math/rand"; "testing" )

const defaultChars = "01234567 . . . mnopqrstuvwxyz"

func randString(length int, charset string) string {
    b := make([]byte, length)
    for i := range b {
        b[i] = charset[rand.Intn(len(charset))]
    }
    return string(b)
}

type forwarder interface {
    forward(string) int
}
```

Forwarding example

```
type thing1 struct {
    t forwarder
}

func (t1 *thing1) forward(s string) int {
    return t1.t.forward(s)
}

type thing2 struct {
    t forwarder
}

func (t2 *thing2) forward(s string) int {
    return t2.t.forward(s)
}

type thing3 struct {}
```

Forwarding example

```
func (t3 *thing3) forward(s string) int {
    return len(s)
}

func length(s string) int {
    return len(s)
}

func BenchmarkDirect(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h[i] = length(r)
    }
}
```

Forwarding example

```
func BenchmarkForward(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)

    var t3 forwarder = &thing3{}
    var t2 forwarder = &thing2{t3}
    var t1 forwarder = &thing1{t2}

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        h[i] = t1.forward(r)
    }
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8      1000000000          0.627 ns/op
BenchmarkForward-8     189176289           6.35 ns/op
```

Forwarding example

Let's make one small change

```
//go:noinline
func length(s string) int {
    return len(s)
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8     624469180          1.87 ns/op
BenchmarkForward-8    189169784          6.35 ns/op
```

Presumably that makes both function calls happen out of line

Forwarding example

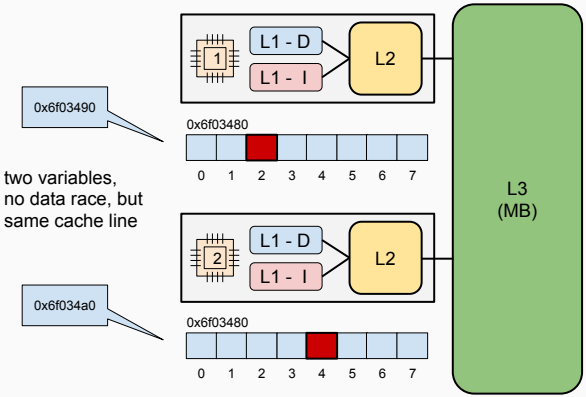
Let's make one other change, not so small

```
func BenchmarkForward(b *testing.B) {
    r := randString(rand.Intn(24), defaultChars)
    h := make([]int, b.N)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        var t3 forwarder = &thing3{}
        var t2 forwarder = &thing2{t3}
        var t1 forwarder = &thing1{t2}
        h[i] = t1.forward(r)
    }
}
```

```
$ go test -bench=. ./forward_test.go
BenchmarkDirect-8      623482398          1.88 ns/op
BenchmarkForward-8     23797018          49.2 ns/op
```

False sharing example

False sharing: cores fight over a cache line for *different* variables



False sharing example

We're going to let CPUs clobber each other's cache

```
import (
    "sync"
    "testing"
)

const (
    nworker = 8
    buffer  = 1024
)

var wg sync.WaitGroup
```

False sharing example

```
func run() (total int) {
    cnt := make([]uint64, nworker)   // one cache line
    in := make([]chan int, nworker)
    for i := 0; i < nworker; i++ {
        in[i] = make(chan int, buffer)
        go fill(10000, in[i])
    }
    for i := 0; i < nworker; i++ {
        wg.Add(1)
        go count(&cnt[i], in[i])
    }
    wg.Wait()
    for _, v := range cnt {
        total += int(v)
    }
    return
}
```

False sharing example

```
func count(cnt *uint64, in <-chan int) {
    // false sharing

    for i := range in {
        *cnt += uint64(i)
    }

    wg.Done()
}

func fill(n int, in chan<- int) {
    for i := 0; i < n; i++ {
        in <- i
    }

    close(in)
}
```

False sharing example

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

We're going to run with different numbers of cores

```
$ go test -bench=. -benchtime=10s -cpu=2,4,8 ./share_test.go
goos: darwin
goarch: amd64
BenchmarkShare-2            3630        3280637 ns/op
BenchmarkShare-4            4291        2832584 ns/op
BenchmarkShare-8            4910        2629852 ns/op
```

And watch it blow up due to false sharing

False sharing example

Now we're going to write back a local total instead

```
func count(cnt *uint64, in <-chan int) {
    var total int

    for i := range in {
        total += i
    }
    *cnt = uint64(total)
    wg.Done()
}
```

We actually see real improvement with more cores

```
BenchmarkShare-2            5526        2206546 ns/op
BenchmarkShare-4            9950        1241392 ns/op
BenchmarkShare-8           13222         998438 ns/op
```

A few things to consider

Here are some concerns about CPU/memory benchmarking:

- is the data / code available in cache?
- did you hit a garbage collection?
- did virtual memory have to page in/out?
- did branch prediction work the same way?
- did the compiler remove code via optimization?
  (are there side effects in the code?)
- are you running in parallel? how many cores?
- are those cores physical or virtual?
- are you sharing a core with anything else?
- what other processes are sharing the machine?