Writing High Performance Go

GopherChina
17 April 2016

Dave Cheney

Welcome

您好!

Thank you for coming to my talk.

Thank you also to the GopherChina organisers for inviting me to speak.

My name is David. I'm a Go programmer from Sydney, Australia.

I'm a contributor to the Go project and I run the Sydney Go Users' group.

Agenda

This talk is aimed at development teams who are building production Go applications intended for high scale deployment.

This presentation will be available after the talk. It contains lots of examples and links to other material.

Today I am going to cover four areas:

- Performance measurement and profiling
- Benchmarking
- Memory management and GC tuning
- Concurrency

I'm going to be here for the entire conference, please come and ask me your questions after the talk.

Performance measurement and profiling

Performance measurement and profiling

There is a old Australian proverb.

"Measure twice, cut once"

Before you can begin to tune your application, you need to know if your changes are making things better, or worse.

You must establish a reliable baseline to measure the impact of your change.

In other words, *"Don't guess, measure"*

Profiling basics

Before you profile, you must have a stable environment to get repeatable results.

- The machine must be idle—don't profile on shared hardware, don't browse the web while waiting for a long benchmark to run.
- Watch out for power saving and thermal scaling.
- Avoid virtual machines and shared cloud hosting; they are too noisy for consistent measurements.
- There is a kernel bug on OS X versions less than El Capitan; upgrade or avoid profiling on OS X.

If you can afford it, buy dedicated performance test hardware. Rack them, disable all the power management and thermal scaling and never update the software on those machines.

For everyone else, have a before and after sample and run them multiple times to get consistent results.

pprof

The primary tool we're going to be talking about today is *pprof*.

pprof descends from the Google Performance Tools suite.

pprof profiling is built into the Go runtime.

We're going to discuss CPU and Memory profiling today.

Further reading: runtime/pprof (https://golang.org/pkg/runtime/pprof)

Further reading: Google Perf Tools (https://github.com/gperftools/gperftools)

CPU profiling

CPU profiling is the most common type of profile.

When CPU profiling is enabled, the runtime will interrupt itself every 10ms and record the stack trace of the currently running goroutines.

Once the profile is saved to disk, we can analyse it to determine the hottest code paths.

The more times a function appears in the profile, the more time that code path is taking as a percentage of the total runtime.

Memory profiling

Memory profiling records the stack trace when a *heap* allocation is made.

Memory profiling, like CPU profiling is sample based. By default memory profiling samples 1 in every 1000 allocations. This rate can be changed.

Stack allocations are assumed to be free and are *not tracked* in the memory profile.

Because of memory profiling is sample based and because it tracks *allocations* not *use*, using memory profiling to determine your application's overall memory usage is difficult.

We'll talk more about measuring memory usage later.

One profile at at time

Profiling is not free.

Profiling has a moderate, but measurable impact on program performance—especially if you increase the memory profile sample rate.

Most tools will not stop you from enabling multiple profiles at once.

If you enable multiple profiles at the same time, they will observe their own interactions and skew your results.

Do not enable more than one kind of profile at a time.

Using pprof

Now that I've talked about what pprof can measure, I will talk about how to use pprof to analyse a profile.

pprof should always be invoked with *two* arguments.

```
go tool pprof /path/to/your/binary /path/to/your/profile
```

The `binary` argument **must** be the binary that produced this profile.

The `profile` argument **must** be the profile generated by this binary.

Warning: Because pprof also supports an online mode where it can fetch profiles from a running application over http, the pprof tool can be invoked without the name of your binary (issue 10863 (https://github.com/golang/go/issues/10863)):

```
go tool pprof /tmp/c.pprof
```

Do not do this or pprof will report your profile is empty.

Using pprof (cont.)

This is a sample cpu profile:

```
% go tool pprof $BINARY /tmp/c.p
Entering interactive mode (type "help" for commands)
(pprof) top
Showing top 15 nodes out of 63 (cum >= 4.85s)
      flat  flat%   sum%        cum   cum%
    21.89s  9.84%  9.84%    128.32s 57.71%  net.(*netFD).Read
    17.58s  7.91% 17.75%     40.28s 18.11%  runtime.exitsyscall
    15.79s  7.10% 24.85%     15.79s  7.10%  runtime.newdefer
    12.96s  5.83% 30.68%    151.41s 68.09%  test_frame/connection.(*ServerConn).readBytes
    11.27s  5.07% 35.75%     23.35s 10.50%  runtime.reentersyscall
    10.45s  4.70% 40.45%     82.77s 37.22%  syscall.Syscall
     9.38s  4.22% 44.67%      9.38s  4.22%  runtime.deferproc_m
     9.17s  4.12% 48.79%     12.73s  5.72%  exitsyscallfast
     8.03s  3.61% 52.40%     11.86s  5.33%  runtime.casgstatus
     7.66s  3.44% 55.85%      7.66s  3.44%  runtime.cas
     7.59s  3.41% 59.26%      7.59s  3.41%  runtime.onM
     6.42s  2.89% 62.15%    134.74s 60.60%  net.(*conn).Read
     6.31s  2.84% 64.98%      6.31s  2.84%  runtime.writebarrierptr
     6.26s  2.82% 67.80%     32.09s 14.43%  runtime.entersyscall
```

Often this output is hard to understand.

Using pprof (cont.)

A better way to understand your profile is to visualise it.

```
% go tool pprof $BINARY /tmp/c.p
Entering interactive mode (type "help" for commands)
(pprof) web
```

Opens a web page with a graphical display of the profile.

I find this method to be superior to the text mode, I strongly recommend you try it.

writing-high-performance-go/profile.svg (writing-high-performance-go/profile.svg)

`pprof` supports a non interactive form with flags like `-svg`, `-pdf`, etc. See `go tool pprof -help` for more details.

Further reading: Profiling Go programs (http://blog.golang.org/profiling-go-programs)

Further reading: Debugging performance issues in Go programs (https://software.intel.com/en-us/blogs/2014/05/10/debugging-performance-issues-in-go-programs)

Using pprof (cont.)

We can visualise memory profiles in the same way.

```
% go build -gcflags='-memprofile=/tmp/m.p'
% go tool pprof --alloc_objects -svg $(go tool -n compile) /tmp/m.p > alloc_objects.svg
% go tool pprof --inuse_objects -svg $(go tool -n compile) /tmp/m.p > alloc_objects.svg
```

The allocation profile reports the location of where every allocation was made.

writing-high-performance-go/alloc_objects.svg (writing-high-performance-go/alloc_objects.svg)

In use profile reports the location of an allocation that are live at the end of the profile.

writing-high-performance-go/inuse_objects.svg (writing-high-performance-go/inuse_objects.svg)

Benchmarking

Benchmarking

Now that we discussed what profiling is, and how to use `pprof`, we're going to look at writing benchmarks and interpreting their results.

The Go runtime's profiling interface, `runtime/pprof`, is a very low level tool. For historic reasons the interfaces to the different kinds of profile are not uniform.

To make it easier to profile your code you should use a higher level interface.

- For profiling `testing` package benchmarks, the `go test` command has integrated support for profiling the code under test:
- For profiling an application, I recommend the github.com/pkg/profile (https://github.com/pkg/profile) package.

This section focuses on how to construct useful benchmarks using the Go testing framework, and gives practical tips for avoiding the pitfalls.

Using the testing package for benchmarking

fib.go:

```
// Fib computes the n'th number in the Fibonacci series.
func Fib(n int) int {
    if n < 2 {
        return n
    }
    return Fib(n-1) + Fib(n-2)
}
```

fib_test.go:

```
import "testing"

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

DEMO: `go test -bench=. ./fib`

Comparing benchmarks

For repeatable results, you should run benchmarks multiple times.

You can do this manually, or use the `-count=` flag.

Determining the performance delta between two sets of benchmarks can be tedious and error prone.

Tools like rsc.io/benchstat (https://godoc.org/rsc.io/benchstat) are useful for comparing results.

```
% go test -bench=. -count=5 > old.txt
```

DEMO: Improve `Fib`

```
% go test -bench=. -count=5 > new.txt
% benchstat old.txt new.txt
```

DEMO: `benchstat {old,new}.txt`

Watch out for compiler optimisations

How fast will this benchmark run ?

```
const m1 = 0x5555555555555555
const m2 = 0x3333333333333333
const m4 = 0x0f0f0f0f0f0f0f0f
const h01 = 0x0101010101010101

func popcnt(x uint64) uint64 {
    x -= (x >> 1) & m1
    x = (x & m2) + ((x >> 2) & m2)
    x = (x + (x >> 4)) & m4
    return (x * h01) >> 56
}

func BenchmarkPopcnt(b *testing.B) {
    for i := 0; i < b.N; i++ {
        x := i
        x -= (x >> 1) & m1
        x = (x & m2) + ((x >> 2) & m2)
        x = (x + (x >> 4)) & m4
        _ = (x * h01) >> 56
    }
}
```

DEMO: `go test -bench=. ./popcnt`

What happened?

`popcnt` is a leaf function, so the compiler can inline it.

Because the function is inlined, the compiler can see it has no side effects, so the call is eliminated. This is what the compiler sees:

```
func BenchmarkPopcnt(b *testing.B) {
    for i := 0; i < b.N; i++ {
        // optimised away
    }
}
```

This is *not* a bug.

The same optimisations that make code fast, by removing unnecessary computation, are the same ones that remove benchmarks that have no observable side effects.

DEMO: show how to fix popcnt

Further reading: How to write benchmarks in Go (http://dave.cheney.net/2013/06/30/how-to-write-benchmarks-in-go)

Profiling benchmarks

The `testing` package has built in support for generating CPU, memory, and block profiles.

- `-cpuprofile=$FILE` writes a CPU profile to `$FILE`.
- `-memprofile=$FILE`, writes a memory profile to `$FILE`, `-memprofilerate=N` adjusts the profile rate to `1/N`.

Using any of these flags also preserves the binary.

```
% go test -run=XXX -bench=IndexByte -cpuprofile=/tmp/c.p bytes
% go tool pprof bytes.test /tmp/c.p
```

Note: use `-run=XXX` to disable tests, you only want to profile benchmarks.

Profiling applications

Profiling `testing` benchmarks is useful for *microbenchmarks*, but what if you want to profile a complete application?

To profile an application, you could use the `runtime/pprof` package, but that is fiddly and low level.

A few years ago I wrote a small package, github.com/pkg/profile (https://godoc.org/github.com/pkg/profile), to make it easier to profile an application.

```
import "github.com/pkg/profile"

func main() {
      defer profile.Start().Stop()
      ...
}
```

DEMO: Show profiling `cmd/godoc` with `pkg/profile`

Memory management and GC tuning

Memory management and GC tuning

Go is a garbage collected language. This is a design principal, it will not change.

The Go GC favors *lower latency* over *maximum throughput*; it moves some of the allocation cost to the mutator to reduce the cost of cleanup later.

As a garbage collected language, the performance of Go programs is often determined by their interaction with the garbage collector.

Next to your choice of algorithms, memory consumption is the most important factor that determines the performance and scalability of your application.

In this section we will discuss the operation of the garbage collector, how to measure the memory usage of your program and strategies for lowering memory usage if garbage collector performance is a bottleneck.

Garbage collector monitoring

A simple way to obtain a general idea of how hard the garbage collector is working is to enable the output of GC logging.

These stats are always collected, but normally supressed, you can enable their display by setting the `GODEBUG` environment variable.

```
% env GODEBUG=gctrace=1 godoc -http=:8080
gc 1 @0.017s 8%: 0.021+3.2+0.10+0.15+0.86 ms clock, 0.043+3.2+0+2.2/0.002/0.009+1.7 ms cpu, 5->6->
gc 2 @0.026s 12%: 0.11+4.9+0.12+1.6+0.54 ms clock, 0.23+4.9+0+3.0/0.50/0+1.0 ms cpu, 4->6->3 MB, 6
gc 3 @0.035s 14%: 0.031+3.3+0.76+0.17+0.28 ms clock, 0.093+3.3+0+2.7/0.012/0+0.84 ms cpu, 4->5->3
gc 4 @0.042s 17%: 0.067+5.1+0.15+0.29+0.95 ms clock, 0.20+5.1+0+3.0/0/0.070+2.8 ms cpu, 4->5->4 MB
gc 5 @0.051s 21%: 0.029+5.6+0.33+0.62+1.5 ms clock, 0.11+5.6+0+3.3/0.006/0.002+6.0 ms cpu, 5->6->4
gc 6 @0.061s 23%: 0.080+7.6+0.17+0.22+0.45 ms clock, 0.32+7.6+0+5.4/0.001/0.11+1.8 ms cpu, 6->6->5
gc 7 @0.071s 25%: 0.59+5.9+0.017+0.15+0.96 ms clock, 2.3+5.9+0+3.8/0.004/0.042+3.8 ms cpu, 6->8->6
```

The trace output gives a general measure of GC activity.

DEMO: Show godoc with `GODEBUG=gctrace=1` enabled

Garbage collector monitoring (cont.)

Using `GODEBUG=gctrace=1` is good when you *know* there is a problem, but for general telemetry I recommend the `net/http/pprof` interface.

```
import _ "net/http/pprof"
```

Importing the `net/http/pprof` package will register a handler at `/debug/pprof` with various runtime metrics, including:

- A list of all the running goroutines, `/debug/pprof/heap?debug=1`.
- A report on the memory allocation statistics, `/debug/pprof/heap?debug=1`.

Warning: `net/http/pprof` will register itself with your default `http.ServeMux`.

Be careful as this will be visible if you use `http.ListenAndServe(address, nil)`.

DEMO: `godoc -http=:8080`, show `/debug/pprof`.

Garbage collector tuning

The Go runtime provides one environment variable to tune the GC, `GOGC`.

The formula for `GOGC` is as follows.

```
goal = reachable * (1 + GOGC/100)
```

For example, if we currently have a 256mb heap, and `GOGC=100` (the default), when the heap fills up it will grow to

```
512mb = 256mb * (1 + 100/100)
```

- Values of `GOGC` greater than 100 causes the heap to grow faster, reducing the pressure on the GC.
- Values of `GOGC` less than 100 cause the heap to grow slowly, increasing the pressure on the GC.

The default value of 100 is only a guide, you should choose your own value *after profiling your application with production loads*.

Reduce allocations

Make sure your APIs allow the caller to reduce the amount of garbage generated.

Consider these two Read methods

```
func (r *Reader) Read() ([]byte, error)
func (r *Reader) Read(buf []byte) (int, error)
```

The first Read method takes no arguments and returns some data as a `[]byte`. The second takes a `[]byte` buffer and returns the amount of bytes read.

The first Read method will *always* allocate a buffer, putting pressure on the GC. The second fills the buffer it was given, allowing the caller to reuse the buffer.

strings and []bytes

Most programs prefer to work with `string`, but most IO is done with `[]byte`.

In Go `string` values are immutable, `[]byte` are mutable. Converting between the two generates garbage.

Avoid `[]byte` to string conversions wherever possible, this normally means picking one representation, either a `string` or a `[]byte` for a value. Often this will be `[]byte` if you read the data from the network or disk.

The `bytes` (https://golang.org/pkg/bytes/) package contains many of the same operations— `Split`, `Compare`, `HasPrefix`, `Trim`, etc—as the `strings` (https://golang.org/pkg/strings/) package.

Under the hood `strings` uses same assembly primitives as the `bytes` package.

Using []byte as a map key

It is very common to use a `string` as a map key, but often you have a `[]byte`.

The compiler implements a specific optimisation for this case

```
var m map[string]string
v, ok := m[string(bytes)]
```

This will avoid the conversion of the byte slice to a string for the map lookup. This is very specific, it won't work if you do something like

```
key := string(bytes)
val, ok := m[key]
```

Avoid string concatenation

Go strings are immutable. Concatenating two strings generates a third.

Avoid string concatenation by appending into a `[]byte` buffer.

Before:

```
s := request.ID
s += " " + client.Address().String()
s += " " + time.Now().String()
return s
```

After:

```
b := make(b, 0, 40) // guess
b = append(b, request.ID...)
b = append(b, ' ')
b = append(b, addr.String()...)
b = append(b, ' ')
b = time.Now().AppendFormat(b, "2006-01-02 15:04:05.999999999 -0700 MST")
return string(b)
```

DEMO: `go test -bench=. ./concat`

Preallocate slices if the length is known

Append is convenient, but wasteful.

What is the capacity of b after we append one more item to it?

```
func main() {
    b := make([]int, 1024)
    b = append(b, 99)
    fmt.Println("len:", len(b), "cap:", cap(b))
}
```

Run

Slices grow by doubling up to 1024 elements, then by approximately 25% after that. What is the capacity of b after we append one more item to it?

If you use the append pattern you could be copying a lot of data and creating a lot of garbage.

If know know the length of the slice beforehand, then pre-allocate the target to avoid copying and to make sure the target is exactly the right size.

Concurrency

Goroutines

Go's signature feature is its lightweight concurrency model.

Goroutines are so cheap to create, and so easy to use, you could think of them as *almost* free.

The Go runtime has been written for programs with tens of thousands of goroutines as the norm, hundreds of thousands are not unexpected.

While cheap, these features are not free, and overuse often leads to unexpected performance problems.

This final section concludes with a set of do's and don't's for efficient use of Go's concurrency primitives.

Know when to stop a goroutine

Goroutines are cheap to start and cheap to run, but they do have a finite cost in terms of memory footprint; you cannot create an infinite number of them.

Each goroutine consumes a minimum amount of memory for the goroutine's stack, currently at least 2k.

2048 * 1,000,000 goroutines == 2Gb of memory per 1,000,000 goroutines.

Every time you use the go keyword in your program to launch a goroutine, you must know how and when that goroutine will exit.

If you don't know the answer, that's a potential memory leak.

In your design, some goroutines may run until the program exits. These goroutines are rare enough to not become an exception to the rule.

Never start a goroutine without knowing how it will stop.

Use streaming IO interfaces

Where-ever possible avoid reading data into a `[]byte` and passing it around.

Depending on the request you may end up reading megabytes (or more!) of data into memory. This places huge pressure on the GC, which will increase the average latency of your application.

Instead use `io.Reader` and `io.Writer` to construct processing pipelines to cap the amount of memory in use per request.

For efficiency, consider implementing `io.ReaderFrom` / `io.WriterTo` if you use a lot of `io.Copy`. These interface are more efficient and avoid copying memory into a temporary buffer.

io.Reader and io.Writer are not buffered

`io.Reader` and `io.Writer` implementations are not buffered.

This includes `net.Conn`, `*os.File`, and `os.Stdout`.

Use `bufio.NewReader(r)` and `bufio.NewWriter(w)` to get a buffered reader and writer.

Don't forget to `Flush` or `Close` your `bufio.Writer` to flush its buffer to the underlying `Writer`.

Timeouts, timeouts, timeouts

Never start an IO operating without knowing the maximum time it will take.

You need to set a timeout on every network request you make with `SetDeadline`, `SetReadDeadline`, `SetWriteDeadline`.

```
// sendfile sends the contents of path to the client c.
func sendfile(c net.Conn, path string) error {
    r, err := os.Open(path)
    if err != nil {
        return err
    }
    defer r.Close()

    // Set the deadline to one minute from now.
    c.SetWriteDeadline(time.Now().Add(60 * time.Second))

    // Copy will send as much of r to the client as it can in 60 seconds.
    _, err = io.Copy(c, r)
    return err
}
```

Go uses efficient polling, sometimes

The Go runtime handles network IO using an operating system polling mechanism. Many waiting goroutines will be serviced by a single operating system thread.

However, for local file IO, Go does not implement any IO polling. Each operation on a `*os.File` consumes one operating system thread while in progress.

Heavy use of local file IO can cause your program to spawn hundreds or thousands of threads; possibly more than your operating system allows.

Your disk subsystem does not expect to be able to handle hundreds or thousands of concurrent IO requests.

You need to limit the amount of blocking IO you issue. Use a pool of worker goroutines, or a buffered channel as a semaphore.

Watch out for IO multipliers in your application

Most server programs take a request, do some processing, then return a result.

This sounds simple, but depending on the result it can let the client consume a large (possibly unbounded) amount of resources on your server.

How many IO events does a single client request generate? Is it fixed, N+1, or linear (reading the whole table to generate the last page of results).

If memory is slow, relatively speaking, then IO is so slow that you should avoid doing it at all costs.

Most importantly avoid doing IO in the context of a request—don't make the user wait for your disk subsystem to write to disk.

Minimise CGO

cgo allows Go programs to call into C libraries.

C code and Go code live in two different universes, cgo traverses the boundary between them.

This transition is not free and depending on where it exists in your code, the cost could be substantial.

Do not call out to C code in the middle of a tight loop.

cgo calls are similar to blocking IO, they consume a thread during operation.

For best performance I recommend avoiding cgo in your applications.

> Further reading: cgo is not Go. (http://dave.cheney.net/2016/01/18/cgo-is-not-go)

Conclusion

Always write the simplest code you can

Start with the simplest possible code.

Measure.

If performance is good, *stop*. You don't need to optimise everything, only the hottest parts of your code.

As you application grows, or your traffic pattern evolves, the performance hot spots will change.

Don't leave complex code that is not performance critical, rewrite it with simpler operations if the bottleneck moves elsewhere.

Always use the latest released version of Go

Old versions of Go will never get better. They will never get bug fixes or optimisations.

I'm sorry about Go 1.5 and Go 1.6 compile speed, believe me, nobody is happy with the situation and we are working on improving it.

Always use the latest version of Go and you will get the best possible performance.

Further reading: Go 1.7 toolchain improvements (http://dave.cheney.net/2016/04/02/go-1-7-toolchain-improvements)

In conclusion

Profile your code to identify the bottlenecks, *do not guess*.

Always write the simplest code you can, the compiler is optimised for *normal* code.

Shorter code is faster code; Go is not C++, do not expect the compiler to unravel complicated abstractions.

Shorter code is *smaller* code; which is important for the CPU's cache.

Pay very close attention to allocations, avoid unnecessary allocation where possible.

Don't trade performance for reliability; I see little value in having a very fast server that panics, deadlocks or OOMs on a regular basis.

Thank you

Dave Cheney
dave@cheney.net (mailto:dave@cheney.net)
http://dave.cheney.net/ (http://dave.cheney.net/)
@davecheney (http://twitter.com/davecheney)