Notes: Debugging Go Programs

Tom Arrell

Wed 12th Feb – Golang Meetup – Berlin

Overview

- Go at SumUp
- Go in Logistics
- Why we're moving to Go
- Primitive
- GDB
- Delve
- Scenarios

Go at SumUp

We use Go here at SumUp. Although we're still relatively early in our Go story.

For some context, Go has been used here in our SysOps team for a couple of years. But it only really reached a production backend service early last year.

We run most of our services inside Kubernetes, on AWS, after a period of migrating away from more primitive deployment methods. Some teams have included in their migration breaking up monolithic Ruby services into smaller Go microservices, we'll touch on why this decision was made in a bit.

In our payments domain, we have a custom environment bootstrapper we call Theseus, which is written in Go. This was built in order to make it easier to bootstrap our payments applications in ephemeral environments with all of their dependencies.

<- slide ->

Go in Logistics

So I work on our Logistics team where we build software to make the delivery process to our merchants more seamless for other teams within SumUp.

When I joined 6 months ago, almost all the Logistics processes within SumUp were running through a single set of Python scripts, running on a single job server which was hooked up to a very large Postgres database. Since then, we've migrated our European operations to a new set of Golang services, which run in Kubernetes, and provide API's for other teams to interact with.

Also, during this time, we took 2 engineers with no background in Golang to being comfortable contributing.

This has brought a massive improvement in productivity, reliability and observability of our processes. As well as a steady platform to expand our logistics operations into new markets.

Go in particular has made this easy to do with its simplicity. We've found that engineers new to Go have had an easy time picking it up and becoming productive.

<- slide -> <- slide -> <- slide -> <- slide ->

Another benefit I consider is its relative lack of abstractions. Logistics is one of those domains with many edge cases. Go intentionally doesn't give you the power to build complicated abstractions that some other languages do.

<- slide ->

I find that heavy use of abstractions shifts the burden onto the *developer* to be sure that they are valid across a variety of scenarios, or in our case, markets. Spoiler alert, I believe that's probably impossible, and that most abstractions will begin to leak given enough time. Go nudges developers towards the path of practicality and solving the problem clearly, rather than introducing too much indirection.

<- slide ->

Why we're moving to Go

SumUp is in the stage of its growth where scaling the engineering practices in the organisation is important. Our engineering is rather scattered across the globe, including in our offices in São Paulo, Sofia, Cologne and two here in Berlin.

Also, at SumUp we really encourage people to be "T" shaped. In other words, having a broad range of skills across many disciplines, while also specializing where needed.

To do this, we support engineers who want to learn new things by helping them change teams, form new teams, and do this with as little friction as possible. Part of what makes this simple is having as much consistency across teams as makes sense. This makes it much easier for someone to get up to speed with the codebase of a new team in relatively little time. Adopting Go is a step towards this possibility, putting aside the yak shaving, leaving more time for decisions that bring value to the business.

We now have teams in all 4 of our locations with engineering who are writing Go.

Previously, most of the company was built in Ruby. Now don't get me wrong, I'm not knocking Ruby as a language, and there are still plenty of people who thoroughly enjoy writing it. However we've found the high level of abstraction that Ruby encourages to ultimately be detrimental to our organisational vision.

Ruby on Rails with things such as ActiveRecord, Sneakers and Sidekiq, which were all used here at SumUp add an extra level of indirection between your code and the underlying mechanisms. We found it's usually more work to onboard people because of this, even if they've had prior experience with the concrete technologies such as Postgres, RabbitMQ, or even parallel ones such as Kubernetes cron jobs.

From a hiring perspective, it's no longer feasible for us to hire developers who must have had experience with technology X. Therefore, we're investing in reducing abstractions where we can, to make our job of onboarding people easier. Go is an important piece in that puzzle for us.

<- slide ->

Go and GDB

Now for the pivot over to debugging.

If you've ever been debugging a Go program, you probably know it can be quite the non-trivial task at times. Especially when your program is highly concurrent.

I also want to add a preface that debuggers themselves should not be a replacement for careful thought. Rob Pike said himself:

> If you dive into the bug, you tend to fix the local issue in the code, but if you think about the bug first, how the bug came to be, you often find and correct a higher-level problem in the code that will improve the design and prevent further bugs.

– Rob Pike

<- slide ->

---

fmt.Println()

Print debugging is something that probably the majority of programmers are familiar with. It's a simple and easy to use tool, especially when you want to inspect very specific parts of your program and are running things locally on your machine.

In fact, I'm going to go out there are say that fmt.Println() is probably the most universal, and all powerful debugger.

<- slide ->

However, if either of those things doesn't hold, then the challenge begins, and only the brave should stick around. What if your bug occurs on the 237th iteration of this *for* loop? Or what if you don't know which iteration it occurs on? Or what if your program is already running, and you can't restart it in fear that you'll have to wait another 3 days for the bug to appear? Hence, sometimes you can save a lot of time by picking up some other tools.

We'll go through a few examples of some situations that would be a bit difficult for fmt.Println(), and see how we can solve them using some fo the tools in the Go ecosystem.

<- slide ->

GDB (GNU Debugger)

Could I get a show of hands of those that have used an interactive debugger of some sort.

And how many of you have used GDB?

Great, well if you've already worked with GDB, a lot of these concepts will already be familiar to you. Unfortunately though, Go programs tend confuse GDB with the way that they handle stack management, threading as well as a few other things.

A quick example is the "defer" statement. You can use the defer statement to change the return value of the function, however this extra execution after a return is non-standard, and can lead to execution of code which is not expected by GDB.

GDB is also not aware of the Go scheduler's context switching. It is possible for a goroutine to be preempted and scheduled on another processor, which can cause the debugger to hang.

Other limitations of GDB include it struggling with types derived from strings, and method qualifications from packages, causing it to treat identifiers including a "." as unstructured literals. This is made even more difficult when you have methods from other packages implementing interfaces defined locally.

GDB is extremely versatile, and the Go team have released extensions to make using it more ergonomic. E.g. pretty printing strings, slices, maps, channels and interfaces. You can even print directly the length or capacity of slices.

These all help to improve the usability of GDB, but we won't be covering GDB today.

<- slide ->

Delve

Enter Delve. Now I'd expect a lot of people here have probably heard, if not used Delve themselves to debug their programs. It's been around for quite a while these days, having been started by Derek Parker back in early 2014.

Delve was purpose built for debugging Go, and deals with some of the shortfalls that are present in GDB.

We'll be making use of Delve a couple of times to help debug during a few of the scenarios that we have.

And just something to note, Delve works best on Linux, a few commands are only available on Linux. If you're running a Mac, you can get most of the benefit running within a Docker container.

<- slide ->

Race Conditions

First up, we'll have a look at a race condition.

Someone a fair bit wiser than myself once said that, "ignoring this prohibition [of data races] introduces a practical risk of future miscompilation of the program."1 In a bit more layman's terms, the dude was essentially saying...

<- slide ->

No race is a safe race.

– Me, just now.

This is something we'd therefore like to avoid in order to prevent potential problems later down the line in our production software.

Let's take a look at a simple program which contains a data race.

<- slide ->

It is a good idea to run your tests with this flag enabled. Be warned though, the race detector is not infallible, and it may possibly miss certains cases. However, it will never report false positives.

<- slide ->

Defer

Defer is a well loved Go feature, allowing you to schedule work to be done before the function exits.

The unique thing about defer which makes it more powerful than simply having the compiler inline statements however is that it can be dynamically set.

You can defer functions from within loops, conditionals, even switches.

The caveat here however is that defers can also have access to the local scope, making it possible to change the values of the return values.

We'll take a look at how we might be able to debug the execution of defer statements.

Core Dumps

Core dumps are essentially a snapshot of a process' memory at the time it crashed. They allow you to analyse a crashed program in more detail, including getting views of the source as the program crashed.

We'll have a look at how we can setup an application to generate a core dump when it crashes, and what sort of analysis we can do to it.

Memory Leaks

Memory leaks are a bit more of a nuanced issue, that may only become a problem over extended periods time.

In order to observe our applications over such periods, it's usually a good idea to have monitoring setup using something like Prometheus.

Using the prometheus client will automatically scrape important metrics such as the heap size, as well as many other useful metrics.

An alternative however is a neat little tool called `pprof`.

You can retrieve a heap dump of the running application and render a tree showing all of the current inuse heap objects, as well as their size. This is particularly useful for identify the specific object causing the leak.

Goroutine Deadlocks

Finally, we'll have a look at how you can identify goroutine deadlocks in your program.

Debugging deadlocks within your program can be one of the most difficult things to debug. Especially in highly concurrent programs.

Deadlocks can occur when two threads hold resources that the other is requesting. It can be made even more obscure when hidden behind a race condition, which was a case we ran into with a bug in a popular AMQP library.

pprof provides a profile for inspecting blocking behaviour in go programs, handily named the "blocking profile". Unfortunately, this only reports the amount of time blocked on things that goroutines are no longer blocked on. If a routine stays blocked, this is not useful.

Instead, you want to take a look at the "full goroutine stack dump" which you can access at the path: `/debug/pprof/goroutine?debug=2`.

Delve Tips

- A couple of things that we didn't cover that are possible with Delve. You can use it to debug your test binaries as well.
- The Go standard library has a handy method `runtime.Breakpoint()` which allows you to trigger a breakpoint trap within your debugger on that line. This is particularly useful when you want to always stop at a particular place.

- You can set checkpoints. These let you restart the program from a specific point. Unfortunately this doesn't work on Mac though.

Conclusion

Let this talk be an example of different ways to debug some problems you may run into when writing a lot of Go. This is by no means and exhaustive list, and I would strongly recommend you use the most suitable tool for the job in each situation.

Sometimes, fmt.Println really is just the best way to go.

Questions

Anyone have any questions?