Dave Cheney

The acme of foolishness

cgo is not Go

To steal a quote from JWZ,

> *Some people, when confronted with a problem, think "I know, I'll use cgo."*
> *Now they have two problems.*

Recently the use of cgo came up on the Gophers' slack channel and I voiced my concerns that using cgo, especially on a project that is intended to showcase Go inside an organisation was a bad idea. I've said this a number of times, and people are probably sick of hearing my spiel, so I figured that I'd write it down and be done with it.

cgo is an amazing technology which allows Go programs to interoperate with C libraries. It's a tremendously useful feature without which Go would not be in the position it is today. cgo is key to ability to run Go programs on Android and iOS.

However, and to be clear these are my opinions, I am not speaking for anyone else, I think cgo is overused in Go projects. I believe that when faced with reimplementing a large piece of C code in Go, programmers choose instead to use cgo to wrap the library, believing that it is a more tractable problem. I believe this is a false economy.

Obviously, there are some cases where cgo is unavoidable, most notably where you have to interoperate with a graphics driver or windowing system that is only available as a binary blob. But those cases where cgo's use justifies its trade-offs are fewer and further between than many are prepared to admit.

Here is an incomplete list of trade-offs you make, possibly without realising them, when you base your Go project on a cgo library.

Slower build times

When you `import "C"` in your Go package, `go build` has to do a lot more work to build your code. Building your package is no longer simply passing a list of all the .go files in scope to a single invocation of `go tool` compile, instead:

- The cgo tool needs to be invoked to generate the C to Go and Go to C thunks and stubs.
- Your system C compiler has to be invoked for every C file in the package.
- The individual compilation units are combined together into a single .o file.
- The resulting .o file take a trip through the system linker for fix-ups against shared objects they reference.

All this work happens every time you compile or test your package, which is constantly, if you're actively working in that package. The Go tool parallelises some of this work where possible, but your packages' compile time just grew to include a full rebuild of all that C code.

It's possible to work around this by pushing the cgo shims out into their own package, avoiding the compile time hit, but now you've had to restructure your application to work around a problem that you didn't have before you started to use cgo.

Oh, and you have to debug C compilation failures on the various platforms your package supports.

Complicated builds

One of the goals of Go was to produce a language who's build process was self describing; the source of your program contains enough information for a tool to build the project. This is not to say that using a Makefile to automate your build workflow is bad, but before cgo was introduced into a project, you may not have needed anything but the go tool to build and test. Afterwards, to set all the environment variables, keep track of shared objects and header files that may be installed in weird places, now you do.

Keep in mind that Go supports platforms that don't ship with make out of the box, so you'll have to dedicate some time to coming up with a solution for your Windows users.

Oh, and now your users have to have a C compiler installed, not just a Go compiler. They also have to install the C libraries your project depends on, so you'll be taking on that support cost as well.

Cross compilation goes out the window

Go's support for cross compilation is best in class. As of Go 1.5 you can cross compile from any supported platform to any other platform with the official installer available on the Go project website.

By default cgo is disabled when cross compiling. Normally this isn't a problem if your project is pure Go. When you mix in dependencies on C libraries, you either have to give up the option to cross compile your product, or you have to invest time in finding and maintaining cross compilation C toolchains for all your targets.

Maybe if you work on a product that only communicates with clients over TCP sockets and you intend to run it in a SaaS model it's reasonable to say that you don't care about cross compilation. However, if you're making a product which others will use, possibly integrated into their products, maybe it's a monitoring solution, maybe it's a client for your SaaS service, then you've locked them out of being able to easily cross compile.

The number of platforms that Go supports continues to grow. Go 1.5 added support for 64 bit ARM and PowerPC. Go 1.6 adds support for 64 bit MIPS, and IBM's s390 architecture is touted for Go 1.7. RISC-V is in the pipeline. If your product relies on a C library, not only do you have the all problems of cross compilation described above, you also have to make sure the C code you depend on works reliably on the new platforms Go is supporting — and you have to do that with the limited debuggability a C/Go hybrid affords you. Which brings me to my next point.

You lose access to all your tools

Go has great tools; we have the race detector, pprof for profiling code, coverage, fuzz testing, and source code analysis tools. None of those work across the cgo blood/brain barrier.

Conversely excellent tools like valgrind don't understand Go's calling conventions or stack layout. On that point, Ian Lance Taylor's work to integrate clang's memory sanitiser to debug dangling pointers on the C side will be of benefit for cgo users in Go 1.6.

Combing Go code and C code results in the intersection of both worlds, not the union; the memory safety of C, and the debuggability of a Go program.

Performance will always be an issue

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 inconsequential, or substantial.

C doesn't know anything about Go's calling convention or growable stacks, so a call down to C code must record all the details of the goroutine stack, switch to the C stack, and run C code which has no knowledge of how it was invoked, or the larger Go runtime in charge of the program.

To be fair, Go doesn't know anything about C's world either. This is why the rules for passing data between the two have become more onerous over time as the compiler becomes better at spotting stack data that is no longer considered live, and the garbage collector becomes better at doing the same for the heap.

If there is a fault while in the C universe, the Go code has to recover enough state to at least print a stack trace and exit the program cleanly, rather than barfing up a core file.

Managing this transition across call stacks, especially where signals, threads and callbacks are involved is non trivial, and again Ian Lance Taylor has done a huge amount of work in Go 1.6 to improve the interoperability of signal handling with C.

The take away is that the transition between the C and Go world is non trivial, and it will never be free from overhead.

C calls the shots, not your code

It doesn't matter which language you're writing bindings or wrapping C code with; Python, Java with JNI, some language using libFFI, or Go via cgo; it's C's world, you're just living in it.

Go code and C code have to agree on how resources like address space, signal handlers, and thread TLS slots are to be shared — and when I say agree, I actually mean Go has to work around the C code's assumption. C code that can assume it always runs on one thread, or blithely be unprepared to work in a multi threaded environment at all.

You're not writing a Go program that uses some logic from a C library, instead you're writing a Go program that has to coexist with a belligerent piece of C code that is hard to replace, has the upper hand negotiations, and doesn't care about your problems.

Deployment gets more complicated

Any presentation on Go to a general audience will contain at least one slide with these words:

> *Single, static binary*

This is Go's ace in the hole that has lead it to become a poster child of the movement away from virtual machines and managed runtimes. Using cgo, you give that up.

Depending on your environment, it's probably possible to build your Go project into a deb or rpm, and assuming your other dependencies are also packaged, add them as an install dependency and push the problem off the operating system's package manager. But that's several significant changes to a build and deploy process that was previously as straight forward as go build && scp.

It *is* possible to compile a Go program entirely statically, but it is by no means simple and shows that the ramifications of including cgo in your project will ripple through your entire build and deploy life cycle.

Choose wisely

To be clear, I am not saying that you should not use cgo. But before you make that Faustian bargain, please consider carefully the qualities of Go that you'll be giving up in return.

Related Posts:

1. **Cross compilation with Go 1.5**
2. **An introduction to cross compilation with Go**
3. **Cross compilation just got a whole lot better in Go 1.5**
4. **An introduction to cross compilation with Go 1.1**

This entry was posted in Go, Programming and tagged cgo on January 18, 2016 [https://dave.cheney.net/2016/01/18/cgo-is-not-go] .