The Case for `interface{}`

Sam Whited

February 02, 2018

Abstract

The empty interface (`interface{}`) is one of the most interesting, powerful, and easily abused features of Go, but there are use cases for which it is uniquely and excellently suited. In this talk for the Go devroom, I will propose a set of rules which can be used to see if your use of `interface{}` will be the elegant and simple API that `interface{}` promised, or merely lead to a maintainability nightmare down the road.

1 Introduction

In this crowd I suspect I don't have to start with an explanation of `interface{}` or its dangers, but let me briefly lay out a few assumptions that will be essential going forward:

- In Go, interfaces should describe behavior, not data
- `interface{}` is easy to abuse (and thus, is abused; widely and often)
- `interface{}` is code for "dynamic typing"

That is to say, whenever I say "`interface{}`", you should hear "dynamic typing". Let us also assume that it's widely acknowledged that if you can describe your behavior with a more specific type, you should. Finally, let us take it as a given that heavy use of reflection leads to difficult to maintain code. If you don't agree, please attempt to make changes to one of the `encoding/*` packages and then get back to me.

2 Library Maintainability

Let's start by talking about maintainability by taking an example from the standard library, in fact, from `encoding/xml`:

```
// Marshal returns the XML encoding of v.
func Marshal(v interface{}) ([]byte, error)
```

We can say that this function "consumes" a dynamically typed value. Because, as Rob Pike famously said, "empty interface says nothing", it must rely on a mix of type switching and reflection to determind how the value should be marshaled into XML. For primitive types and types defined in the `encoding/xml` package this works well enough, we type switch and the package can perform some pre-defined behavior. For example, if the type is an `xml.Marshaler` it can call its `MarshalXML` method, or if it is a primitive type the logic for marshaling the type is already contained in the package. However, for types that are defined outside of the `encoding/xml` package things are more complicated. Type erasure means we have no information about a user defined struct; no methods to call, no known fields, etc. Since no information is known about the type we either must limit the usability of the package by not allowing user defined types or gather the information we need using reflection, leading to maintainability problems. As I said before, if you've ever tried to modify `encoding/xml` or `encoding/json`, you'll know what I'm talking about: part of your code base is heavily reflection based, and part of your code is normal, idiomatic Go. It becomes difficult to reconcile the two distinct code bases.

The `MarshalXML` API is simple, even elegant, from the users perspective. The user doesn't know or care about the type of data: they put arbitrary data in and get some serialized representation of that data out. This contract is reflected with appropriate types in the function signature. However, from the library developers perspective things became very hard to maintain because the library *does* need to care about the type of the data being put in. When the *producer* of some data does not care about the type, but the *consumer* does, the library becomes difficult to maintain.

Rephrased, this becomes our first rule for using `interface{}`: "The producer of the `interface{}`must also be the consumer of the `interface{}`."

3 Application Maintainability

Let's talk about the `context` package.

```go
package context

// WithValue returns a copy of parent in which the value
// associated with key is val.
func WithValue(
	parent Context, key, val interface{},
) Context

type Context interface {
	// Value returns the value associated with this
	// context for key, or nil if no value is
	// associated with key.
	Value(key interface{}) interface{}
}
```

`context` does not have the same problems as `encoding/xml`. Context is commonly used to write HTTP middleware, and when doing so, a library using context generally defines the key and value, for instance in a "WithSessionID" middleware, and generally also provides a way to consume that value ("GetSessionID"). This API means that the context package does not need to make heavy use of reflection, and the package is a study in simplicity. If you're learning to write idiomatic Go and haven't looked at the source of `context`, I highly recommend it.

However, `context` has a different problem. The documentation for the `context` package says: "Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes." Let's stop for a moment to think about "request scoped values" means. I suspect the intent was for that short list to include things like:

1. Request ID
2. Session ID
3. Trace ID

And that's about it really: anything that should be accessable from anywhere that can reference the request and ends with "ID".

Unfortunately, I suspect we've all seen the horror that is a developer who's used to certain Java web frameworks discovering Go's "dependency injection package":

```go
func AddErrorLogger(
	ctx context.Context,
	logger log.Logger,
) context.Context {
	/* … */
}
```

```go
func AddDebugLogger(
	ctx context.Context,
	logger log.Logger,
) context.Context {
	/* … */
}
```

This is an understandable mistake, and soon corrected. Side channel APIs that are, effectively, stringly-typed quickly become a maintainability problem when scaling your team to a size greater than one. So why do developers abuse context as a side-channel API for dependency injection so often?

Certainly it can be partially attributed to new Go users coming from other languages where this is an accepted pattern, but there is also a simpler answer: "because they can." As a rule if something can be abused, it will. And in this case the abuse is made possible by context's use of the empty interface. Take, for example the logging middleware mentioned earlier:

```go
// LogKey is a context key that can be used for
// getting a log.Logger from a request.
// Don't do this.
type LogKey struct{}

// AddLogger adds a log.Logger to a request.
// No really, Don't do this.
func AddLogger(next Handler, l *log.Logger) HandlerFunc {
	return func(w ResponseWriter, r *Request) {
		ctx := r.Context()
		ctx = context.WithValue(
                        ctx, LogKey{}, logger)
                r = r.WithContext(ctx)
                h.ServeHTTP(w, r)
        }
}
```

This is broken *because* of our first rule: the producer of the value in the `interface{}` is also the consumer, and therefore they can use the `interface{}` to create a side channel API, passing arbitrary values between `ServeHTTP` calls. We've traded unmaintainability in our library for unmaintainability in our user's library or application by allowing them to produce and consume `interface{}` typed values. To fix this, we need another rule: "`interface{}` should not cross package boundaries." If you must use `interface{}` in your library, ensure that it is an implementation detail and not leaked in a usable way by any exported function or method.

Finally, let's look at an example that follows these rules:

```
// Mechanism represents an auth mechanism
// (eg. plain, scram, or oauth2).
type Mechanism struct {
        Next func(data interface{}) (cache interface{})
}

// Negotiator is a state machine that handles
// requests and responses in the auth flow.
type Negotiator struct{
        cache interface{}
}

// Step advances the state machine.
func (c *Negotiator) Step(challenge []byte) (resp []byte)
```

The Simple Authentication and Security Layer (SASL) is an authentication framework defined by RFC 4422. You probably use it every time you access your email, auth to Freenode, etc. The specifcs don't matter, except that SASL supports many authentication mechanisms; these are things like plain text, SCRAM, OAUTH2, etc. and that, because these mechanisms may require multiple round trips to the server there is generally a state machine, the "`Negotiator`" in this API, that controls the auth flow, ensures that steps cannot be repeated, and generally enforces security constraints.

As you can see, the `Mechanism` makes use of `interface{}`. In each step of the state machine the selected mechanism's '`Next`' function is called. `Mechanism`, for reasons that don't matter here, must remain stateless. Unfortunately, mechanisms also may need data from previous steps, and this state must be stored somewhere. Since mechanisms are used by a state machine, which is (obviously) already stateful, it makes sense to store that state as part of the `Negotiator`. SASL is also a very flexible framework, so we don't necessarily know what that data will be: sounds like a job for `interface{}`. Each time the '`Next`' function is called it optionally returns a value that will be needed in a future step. The negotiator then stores this state, and the next time it calls `Next` it passes in the stored value. In practice, a stateless `Mechanism`'s `Next` functions end up looking something like this:

```go
func Next(step int, data interface{}) interface{} {
	// State machine will always advance "step"
	switch step {
	case 0:
		// Do stuff
		// Return a "random" integer ID:
		return 4
	case 1:
		// We know it's an int!
		id := data.(int)
		// Do more stuff
		return nil
	}
	panic("the state machine is broken!")
}
```

The mechanism is not only the producer of the value in the interface (the "cache" return parmeter), it is also the consumer of the value (the "data" argument), so it meets our first requirement. This `interface{}` is also an internal implementation detail: it never leaks outside of the package. Together, this means that `Mechanism`'s remain simple: reflection is not necessary, and the `interface{}` is not a part of a public API, so it can't be used to expose a side-channel for smuggling data into a system.