moxie: Mocking with Proxy Methods

Chris Lesiw
7 August 2024

Mocking in Go

Most tests should be fast. The faster the tests are, the shorter the feedback cycle is, and the cheaper experimentation is.

Most tests should test components in isolation.

- Without isolation, it's hard to match a test failure to its program code.
- Without isolation, it's not possible to write enough tests to validate every combination of code paths.

One of the most useful tools to make fast, isolated tests, is mocking: intercepting code just before it crosses an interesting boundary and validating that the code is behaving as it should.

What are the dominant strategies for doing this in Go?

Strategy 1: Don't

Much of the Go philosophy is about radical simplicity. Given that, it's reasonable to ask whether we need mocks at all.

The Go team appears to discourage them. In a 2014 talk (https://go.dev/talks/2014/testing.slide) about testing techniques, one of the Go maintainers demonstrates a method for faking remote servers by creating real loopback servers instead.

The approach is creative, but not universally applicable. The behavior being faked in that talk - HTTP status codes - is trivially reproducible. It's less feasible to develop a local server for every third-party service a program might need to talk to.

It's also testing at the wrong boundary. Many third-party services provide their own SDKs. What actually needs to be faked is the SDK, not the service behind it.

"Don't mock" is a simple strategy, but not always a practical one.

Strategy 2: Thick interfaces

A thick interface describes the same methods as a concrete type so that another implementation can be created or generated for use during tests.

```
type S3Client interface {
    CreateBucket(context.Context, *s3.CreateBucketInput, ...func(*s3.Options)) (*s3.CreateBucketOutput
    ListObjectsV2(context.Context, *s3.ListObjectsV2Input, ...func(*s3.Options)) (*s3.ListObjectsV2Out
    GetObject(context.Context, *s3.GetObjectInput, ...func(*s3.Options)) (*s3.GetObjectOutput, error)
    PutObject(context.Context, *s3.PutObjectInput, ...func(*s3.Options)) (*s3.PutObjectOutput, error)
    // ...many more methods...
}
```

Philosophically, this is un-Go-like. Interfaces should be small and describe a broadly-applicable capability, not a concrete type. Contrast the above to an `io.Reader`.

Practically, this is inefficient. Every time you need a new method, first you need to copy-paste its signature into the interface. You also lose information about the concrete type in tools like IntelliSense when writing code with the interface.

Strategy 3: Interface spam

This is an attempt to reconcile testing needs with small interfaces.

```
type CreateBucketer interface {
    CreateBucket(context.Context, *s3.CreateBucketInput, ...func(*s3.Options)) (*s3.CreateBucketOutput
}
type PutObjecter interface {
    PutObject(context.Context, *s3.PutObjectInput, ...func(*s3.Options)) (*s3.PutObjectOutput, error)
}
```

An `io.Writer` is powerful because it unifies the act of writing bytes in many different contexts: files, network connections, the serial console on an Arduino.

In contrast, a `PutObjecter` (or `ObjectPutter`, if you prefer) describes something very specific, as evidenced by the lengthy method signature. In this way, these interfaces are only describing one real type, albeit in an indirect, distributed manner.

(In fact, the right interface abstraction for an `ObjectPutter` could be an `io.Writer`.)

Beyond interfaces

Mocking is a crucial part of testing.

Interfaces can help here, if the abstraction is right.

But interfaces are not the only Go language feature that can help.

Type embedding

By embedding one type inside another, all selectors, including methods, transparently pass through to the inner type.

```
type S3Client struct{
    *s3.Client
}
func main() {
    client := &S3Client{s3.New()}
    client.ListBuckets(context.Background(), nil) // Just works.
}
```

This transparent passthrough also happens in the language server.

However, if methods with the same name are defined on the outer type, Go will access those instead of the methods on the embedded type.

This is not a quirk or a side-effect of some other feature, but a well-defined part of the language specification (https://go.dev/ref/spec#Selectors).

Proxy methods

Knowing this, it's possible to generate proxy methods for a type to mock its behavior.

```
func (o *Outer) DoSomething() error {
    if o.DoSomethingMock != nil {
        return o.DoSomethingMock()
    }
    return o.Inner.DoSomething()
}
```

However, this is a lot of code that does nothing to end up in the final program. It also invites the possibility of mock code running in production by mistake.

If only it were possible to define such methods conditionally.

Test exclusivity

Turns out it is: `_test.go` files are only built during `go test`. They are ignored in all other circumstances.

Therefore, it is possible to generate methods on the outer type that are only defined at test time.

I made a package to do this: `lesiw.io/moxie`

Demo

Summary

`lesiw.io/moxie` generates proxy methods for mocking that are only present during test time.

Wrapped types work completely normally during regular program execution.

While under test, `moxie` functions give you fine-grained control over which methods will use real or fake implementations, and provides call information regardless of whether a function is mocked or not.

This is especially helpful for interacting with third party API client packages.

Thank you

Chris Lesiw
7 August 2024
chris@chrislesiw.com (mailto:chris@chrislesiw.com)