Functional Options Pattern

Adam Wolfe Gordon
Edmonton Go Meetup
2018-01-22

Going to talk today about a pattern that's fairly common in Go called the functional options pattern.
You might have seen code that uses this pattern, but unless you've implemented it you may not have thought about how it works.

I don't want to jump right into the pattern itself, because it seems like a lot of overhead on its own. Need some motivation first.

Motivation

- Suppose we're writing a logging library.
- Want to configure a bunch of things:
  - Log destinations (file, screen, etc.)
  - Log level
  - Log message prefix
  - Date format
- We should provide nice defaults.
- How do we expose these config options?

```
type Logger interface {
    Debug(string)
    Info(string)
    Warn(string)
    Error(string)
    Fatal(string)
}
```

So, a motivating example.
Suppose we're building a logging library.
We'll have a logger interface that lets us print messages. I'll leave implementation to your imagination.
We want to have a bunch of config options: where to log to, how much to log, a prefix for each message, date/time format, etc.
We should provide nice defaults, but also let a user override any/all of them.
How should we expose all these options to consumers of the library? We'll go through some options.

Option 1: Multiple Implementations

- Different structs that implement the interface.
- A wrapper that lets you log to multiple loggers.

- Only covers **one** of the config dimensions. Really awkward to do multiple dimensions.
- Pretty clunky/verbose for users.

```
type StdoutLogger struct {}
type FileLogger struct {
    FileName string
}
type MultiLogger struct {
    Loggers []Logger
}

func main() {
    l1 := &log.StdoutLogger{}
    l2 := &log.FileLogger{
        FileName: “log”
    }
    l3 := &log.MultiLogger{
        Loggers: []log.Logger{l1, l2}
    }
}
```

One option is to have multiple structs that implement the interface. E.g…
But this is pretty awkward for callers.
AND it only covers one config dimension. Then does each logger need its own set of other config options?

Option 2: Struct With Configuration

- One struct for implementation.
- Struct contains config options.

- The zero value isn't useful.
- Implementation is very exposed.

```
type Logger struct {
    Sinks      []io.Writer
    Level      int
    Prefix     string
    PrintDate  bool
    DateFormat string
}

func main() {
    l := &log.Logger{
        Sinks: []io.Writer{
            os.Stdout,
        }
        Level:  log.DEBUG,
        Prefix: "my_app:",
    }
}
```

So, the next option is to have a single implementation struct, with a bunch of config options.
This gives the user all the options and makes them easy to configure.
But, the zero value isn't useful. The user *must* configure some options or else they'll get no logging at all!
I.e., no good mechanism for defaults.
Also, it would be easy to accidentally modify the logger after instantiation, which probably isn't desirable.
Similar option with slightly better defaults: a config struct.

Option 3: Constructor With Options

- Private implementation type with config.
- Constructor that provides all the config options.

- Better, but still no great way to do defaults.
- Constructor signature will have to change if we think of more options.

```
type logger struct {
    sinks      []io.Writer
    level      int
    prefix     string
    printDate  bool
    dateFormat string
}

func NewLogger(sinks []io.Writer,
    level int, prefix string,
    printDate bool, dateFormat string) Logger
```

This is essentially the same as the previous option, but with the config a bit more hidden.

Option 4: Multiple Constructors

- Same implementation struct as previous.
- A constructor for each set of config options.

- Now we have defaults.
- But there's an **explosion** of constructors.

```
func NewLogger() Logger
func NewLoggerWithSinks(sinks []io.Writer)
    Logger
func NewLoggerWithPrefix(prefix string) Logger
func NewLoggerWithDate() Logger
func NewLoggerWithDateFormat(fmt string)
    Logger
func NewLoggerWithSinksAndPrefix(
    sinks []io.Writer, prefix string) Logger
func NewLoggerWithSinksAndDate(
    sinks []io.Writer) Logger
… … …
```

This is getting better: now we can provide defaults for all the options that aren't provided.
But, we're going to have a gigantic number of constructors. Super wordy.

Functional Options Pattern: Consumer

```
func main() {
    // Logger with defaults
    l := log.NewLogger()
    // Defaults, but print to stderr as well.
    l = log.NewLogger(log.WithAdditionalSinks(os.Stderr))
    // Defaults, but only print to stderr.
    l = log.NewLogger(log.WithSinks(os.Stderr))
    // Defaults, but with a custom prefix and no date.
    l = log.NewLogger(log.WithPrefix("my_app:"), log.WithoutDate())
}
```

OK, finally, we're at the meat of this presentation: the functional options pattern.
First, here's what it looks like in the caller. Once we understand this we'll see how it's implemented.
The constructor takes an arbitrary number of arguments.
Each argument controls one config option.
Any config option not set by an argument will use the default.

Functional Options Pattern: Provider (1)

```
type logger struct { … }

func NewLogger(opts ...LoggerOpt) Logger { … }

type LoggerOpt func(*logger)

func WithPrefix(string prefix) LoggerOpt {
    return func(l *logger) {
        l.prefix = prefix
    }
}
```

First, let's look at the implementation of the options themselves.
Note the signature of the constructor - it takes an arbitrary number of arguments of the LoggerOpt. (Note syntax.)
You can see now where the name of the pattern comes from: the options are functions!
Each option is a function that operates on the logger type. What's especially neat (evil?) is that a caller can't actually define their own options since the argument to the options is a private type.
E.g., the WithPrefix option returns a closure that sets the logger's prefix.

Functional Options Pattern: Provider (2)

```
type logger struct { … }

func NewLogger(opts ...LoggerOpt) Logger {
    // Set up logger with nice defaults.
    l := &logger{
        sinks:      []io.Writer{ os.Stdout },
        printDate:  true,
        dateFormat: time.RFC3339Nano,
    }
    // Apply options.
    for _, opt := range opts {
        opt(l)
    }

    return l
}
```

On to the implementation of the constructor.
It sets up a logger with the defaults we've chosen.
Then it applies all the options provided by the user.

Pretty simple, right?

Not Just For Constructors!

```
type options struct {
    maxTries       int
    baseSleep      time.Duration
    sleepFactor    int64
    maxSleep       time.Duration
}

type Opt func(*options)

func WithMaxTries(tries int) Opt { … }
func WithBaseSleep(sleep time.Duration) Opt { … }
func WithSleepFactor(factor int64) Opt { … }
func WithMaxSleep(sleep time.Duration) Opt { … }

var defaultOpts = options{
    maxTries:    10,
    baseSleep:   10*time.Millisecond,
    sleepFactor: 2,
    maxSleep:    10*time.Second,
}
```

```
func Retry(fn func() error, opts ...Opt) error {
    o := defaultOpts
    for _, opt := range opts {
        opt(&o)
    }

    var err error
    sleep := o.baseSleep
    for n := 0; n < o.maxTries; n++ {
        err = fn()
        if err == nil {
            break
        }
        time.Sleep(sleep)
        sleep = time.Duration(
            sleep.Nanoseconds() * o.sleepFactor)
        if sleep > o.maxSleep {
            sleep = o.maxSleep
        }
    }
    return err
}
```

Note that this technique isn't only useful for constructors.
Here we have a plain function, Retry, that can be used to retry a call until success, with exponential backoff.
There are some options for the retry, but it has good defaults.
We use the functional options pattern to allow overriding the defaults. Nice tidy interface for doing retry.
This is based on a real internal package we have at DO.

Real-World Examples

- gRPC uses functional options all over the place.
  - `func Dial(target string, opts ...DialOption) (*ClientConn, error)`
  - `func NewServer(opt ...ServerOption) *Server`
- Google Cloud, too.
  - `func NewClient(ctx context.Context, opts ...option.ClientOption) (*Client, error)`
- AWS uses it a bit.
  - `func NewSigner(credentials *credentials.Credentials, options ...func(*Signer)) *Signer`
- The etcd client has a nice example.
  - `func OpDelete(key string, opts ...OpOption) Op`
  - `func OpGet(key string, opts ...OpOption) Op`
  - `func OpPut(key, val string, opts ...OpOption) Op`

This pattern is pretty common in the go world. A few examples of prominent projects that use it.

Questions?
Comments?

Adam Wolfe Gordon
@maybeawg
awg@xvx.ca

Shameless Plug:
DigitalOcean is always hiring Go developers.
We are very remote-friendly!
http://l.xvx.ca/do-jobs

See Also

- https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis