The Go Object Lifecycle

Ben Johnson

June 20, 2018

Despite such a simple language, Go developers have found a surprising number of ways to create and use objects. In this post we'll look at a 3-step approach to object management—*instantiation, initialization, & initiation*. We'll also contrast this with other methodologies for creating and using objects and we'll review the pros and cons of each.

1 Our Goals

It may seem like a dumb question but what are our goals for creating and using objects in Go? In keeping with Go style, I prioritize the following:

1. Simplicity
2. Flexibility
3. Documentation friendly

In addition, we should also state what are not our goals. We should assume a basic level of competency of the end user so we don't need to provide excessive guardrails. Users of our code can RTFM [1] (assuming we provide a quality "FM"). We should also assume that users of our code are not adversarial—e.g. we don't need to protect our object fields because we think developers will otherwise use them maliciously.

2 The Process

2.1 Instantiation

First, we need to allocate memory for our object. The general recommendation in the Go community is to make the zero value [2] useful. I find this to be good advice for primitive constructs like `sync.Mutex` or `bytes.Buffer` where their API is limited. This is show in Listing 1.

However, for most application & library developers, constructors can provide efficiency and prevent future bugs.

Listing 1: Mutex usage

```
var mu sync.Mutex
mu.Lock()
// do things...
mu.Unlock()
```

Listing 2: Constructor example

```
// DefaultClientTimeout is the default Client.Timeout.
const DefaultClientTimeout = 30 * time.Seconds

// Client represents a client to our server.
type Client struct {
        Host    string
        Timeout time.Duration
}

// NewClient returns a new instance of Client with default settings.
func NewClient(host string) *Client {
        return &Client{
                Host:    host,
                Timeout: DefaultClientTimeout,
        }
}
```

2.1.1 Using Constructors

Constructors in Go typically take the form of `New` followed by the type name. We can see an example of this for our `Client` type in Listing 2.

By using a constructor, we get several benefits. First, we don't need to check the zero value of `Timeout` every time we use it to see if we should use the default value. It is always set to the correct value.

Second, we provide a seamless upgrade experience if we ever need to initialize fields in the future. Assume we add a `map` of cached values that needs to be initialized on creation as shown in Listing 3.

If we add a constructor in future versions of our library to initialize `cache` then all existing clients using the zero value would be broken. By including the constructor from the beginning and documenting its usage we avoid needing to

Listing 3: Client with unexported cache

```
type Client struct {
        cache map[string]interface{}

        Host    string
        Timeout time.Duration
}
```

Listing 4: Type with exported configuration fields

```
type Client struct {
        // Host and port of remote server.
        // Must be set before Open().
        Host string

        // Time until connection is cancelled.
        // Must be set before Open().
        Timeout time.Duration
}
```

break future versions.

2.1.2 Use natural naming

Another benefit that we get from using our constructor is that our configuration field names no longer need to conform because of zero values. That is, if we have an object that should be "editable" by default, we don't need to make a boolean field called `NotEditable` to make the default zero value (`false`) fit. We can simply use the natural name, `Editable`, and our constructor can set it to `true`.

2.2 Initialization

Once your memory is allocated and default values are assigned, you need to configure your object to your specific use case. This is the area I find most Go developers will overcomplicate but it's quite simple in practice.

2.2.1 Please just use fields

In general, you should just use exported fields for your settings. In our `Client` example above we provided configuration via the `Host` and `Timeout` fields.

To avoid race conditions with other goroutines, this configuration fields should be set once and then left alone since other functions such as an `Open()` or `Start()` may kick off additional goroutines. We can document this restriction on our struct as shown in Listing 4.

One exception to this rule is if you have fields that are updated after you start using the object and need to be mutated concurrently. In this case, we provide getter & setter functions as shown in Listing 5.

However, I find that changing configuration settings during usage to typically be a code smell and it should generally be avoided. It's usually cleaner to simply stop your object and restart with a fresh instance.

Listing 5: Type with mutable configuration

```
type Client struct {
        mu      sync.Mutex
        timeout time.Duration

        // Host and port of remote server.
        // Must be set before Open().
        Host string
}

// Timeout returns the duration until connection is cancelled.
func (c *Client) Timeout() time.Duration {
        c.mu.RLock()
        defer c.mu.RUnlock()
        return c.timeout
}

// SetTimeout sets the duration until connection is cancelled.
func (c *Client) SetTimeout(d time.Duration) {
        c.mu.Lock()
        defer c.mu.Unlock()
        c.timeout = d
}
```

2.3 Initiation

Now that we have memory allocated and our object is configured—let's do something useful. Simple objects are typically ready to go at this point but more complex objects like a server need to be kicked off. They may need to connect to resources or start background goroutines to monitor resources like a `net.Listener`.

In Go we typically see this in the form of an `Open()` or `Start()` function. I tend to prefer `Open()` because the naming pairs well with the `Close()` method in `io.Closer`.

In our client example, we might use the `Open()` to create a network connection and the `Close()` to shut it down as shown in Listing 6.

There are two important facts to notice in this simple example. First, our `Host` is used once in `Open()` and then not touched again. This avoids any race conditions with setting the host after opening the object. Second, we don't try to reset our object state in order to reuse our object. These one-time objects avoid a large class of bugs that occur when trying to reuse objects.

2.3.1 One-time use objects

In practice, it's hard to properly clean up complex objects and reuse them again. In our example, we don't try to set the `conn` to `nil` on `Close()`. This is because the `Client` may have a background goroutine trying to monitor the connection and mutating the `conn` value would require us to add a mutex to protect the field.

Listing 6: Client initiation

```
type Client struct {
        conn net.Conn

        // Host and port of remote server.
        // Must be set before Open().
        Host string
}

// Open opens the connection to the remote server.
func (c *Client) Open() error {
        conn, err := net.Dial("tcp", c.Host)
        if err != nil {
                return err
        }
        c.conn = conn

        return nil
}

// Close disconnects the underlying connection to the server.
func (c *Client) Close() error {
        if c.conn != nil {
                return c.conn.Close()
        }
        return nil
}
```

Listing 7: Prevent double open

```
// Open opens the connection to the remote server.
func (c *Client) Open() error {
        if c.conn != nil {
                return errors.New("myapp.Client: cannot reopen client")
        }
        ...
}
```

We can also use the field to protect against a double open as shown in Listing 7. However, we should assume basic competency of the end user and generally avoid these excessive guardrails.

3 Alternative Methods

Now that we've looked at the instantiation-initialization-initiation method, let's evaluate some other common approaches in the Go community.

3.1 Alternative #1: Functional Options

Dave Cheney describes a pattern called functional options in his post, *Functional Options for Friendly APIs* [4]. The idea is that we can declare a functional argument type to update our unexported fields. We can then initiate our object in the same call since it is already initialized.

To use our `Client` example above, it would look something like Listing 8. Our usage can then fit on a single line as shown in Listing 9.

While this approach hides configuration fields, it does so at the cost of complexity and readability. The godoc API also becomes large and unusable as the number of options grow which makes it difficult to determine which options fit with which types at first glance.

Ultimately, though, we don't need to hide our configuration fields. We should document their usage and trust developers to use them correctly. Leaving these fields exported will group all related configuration fields together within the type as you can see with the `net.Request` type godoc [5].

3.2 Alternative #2: Config Instantiation

Another common approach is to provide "config" objects for your types. These attempt to separate your configuration fields from your type itself. Many times, developers will either copy fields from the config object to the type or embed the config directly into the type.

Using our `Client` example above, it would look like Listing 10. Again, this hides the configuration field on your `Client` type but provides no other benefits. Instead we should simply expose our `Client.Host` field and let our

Listing 8: Function Options example

```
type Client struct {
        host string
}

// OpenClient returns a new, opened client.
func OpenClient(opts ...ClientOption) (*Client, error) {
        c := &Client{}
        for _, opt := range opts {
                if err := opt(c); err != nil {
                        return err
                }
        }
        // open client...
        return c, nil
}

// ClientOption represents an option to initialize the Client.
type ClientOption func(*Client) error

// Host sets the host field of the client.
func Host(host string) ClientOption {
        return func(c *Client) error {
                c.host = host
                return nil
        }
}
```

Listing 9: Function Options usage

```
client, err := OpenClient(Host("google.com"))
```

Listing 10: Config instantiation

```
type Client struct {
        host string
}

type ClientConfig struct {
        Host string
}

func NewClient(config ClientConfig) *Client {
        return &Client{
                host: config.Host,
        }
}
```

users manage it directly. This reduces the complexity of our API and provides cleaner documentation.

3.2.1 When to use configuration objects

Configuration objects are useful but not as an interface between an API caller and the API author. Configuration objects should exist when interfacing between the end user and your software.

For example, a configuration object can provide an interface via a YAML [3] file and your code. These configuration objects should generally live in your `main` package since your binary serves as the translation layer between end users and your code. This is shown in Listing 11.

4 Conclusion

We've looked at a method for managing Go object lifecycle that combines simplicity, flexibility, and is documentation-friendly. First, we *instantiate* our object to allocate memory and set defaults. Next, we *initialize* our object by customizing exported fields. Finally, we *initiate* our object which may start background goroutines or connections.

These simple 3 steps help build code that can be easily used by developers today and can be maintained by developers in the future.

References

[1] RTFM, `https://www.urbandictionary.com/define.php?term=RTFM`

Listing 11: Using configuration types appropriately

```
package main

func main() {
        config := NewConfig()
        if err := readConfig(path); err != nil {
                fmt.Fprintln(os.Stderr, "cannot read config file:", err
                    )
                os.Exit(1)
        }

        client := NewClient()
        client.Host = config.Host
        if err := client.Open(); err != nil {
                fmt.Fprintln(os.Stderr, "cannot open client:", err)
                os.Exit(1)
        }

        // do stuff...
}

type Config struct {
        Host string `yaml:"host"`
}

func NewConfig() Config {
        return \&Config{
                Host: "localhost:1234"
        }
}
```

[2] The Go Programming Language Specification: The Zero Value, https://golang.org/ref/spec#The_zero_value

[3] YAML, http://yaml.org/

[4] Functional Options for Friendly APIs, https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis

[5] net/http.Request, https://golang.org/pkg/net/http/#Request