Abstract

The widely used self-referential function pattern as options, originally proposed by Rob Pike[1], allows us to design a flexible set of APIs to help arbitrary configurations and initialization of a struct. However, when such a pattern is cumbersome when we use one option to support multiple types. This article investigates how the latest Go generics design could empower a refreshed “generic” functional options pattern and show what improvements in the future version of Go could better support such a pattern.

The Functional Options Pattern

In the dotGo 2014, Dave Cheney[2] well explained the motivation and the use of self-referential functional options pattern in addition to the original thoughts from Rob Pike. Let’s recall the key idea briefly.

Assume we have a struct `A` and it internally holds two user-customizable fields `v1`, `v2`:

```go
type A struct {
    v1 int
    v2 int
}
```

Typically, we could make `v1` and `v2` to be public fields, and let users of this struct edit them directly, but this may create difficult compatibility issues to deprecate a field without breaking anything. Another side effect of having public fields is we cannot guarantee the concurrent safety from the user level: there is no way to prevent people from directly editing the public fields.

Instead, we could define a type `Option` to a self referential function `func(*A)`:

```go
type Option func(*A)
```

Then, in order to change the private fields `v1` and `v2`, two functions `V1` and `V2` that returns an `Option` can be written as follows:

---

*Email: research@changkun.de*
func V1(v1 int) Option {
    return func(a *A) {
        a.v1 = v1
    }
}

func V2(v2 int) Option {
    return func(a *A) {
        a.v2 = v2
    }
}

With these functions, the initial settings of an A object could be created by a
NewA function that consumes arbitrary number of options:

func NewA(opts ...Option) *A {
    a := &A{
        for _, opt := range opts {
            opt(a)
        }
    return a
}

For example, the following four different usages both work:

fmt.Printf("%#v\n", NewA()) // &A{v1:0, v2:0}
fmt.Printf("%#v\n", NewA(V1(42)))) // &A{v1:42, v2:0}
fmt.Printf("%#v\n", NewA(V2(42)))) // &A{v1:0, v2:0}
fmt.Printf("%#v\n", NewA(V1(42), V2(42)))) // &A{v1:42, v2:0}

This is also super easy to deprecate an option, because we can simply let an
existing option function not effecting anymore. For instance:

type A struct {
    v1 int
    v2 int
    v3 int
}

type Option func(*A)

func V1(v1 int) Option {
    return func(a *A) {
        a.v1 = v1
    }
}
The golang.design Research

The Problem at Scale

```go
+// Deprecated: Use V3 instead.
func V2(v2 int) Option {
    return func(a *A) {
-        a.v2 = v2
+        // no effects anymore
+        // a.v2 = v2
    }
}

+func V3(v3 int) Option {
+    return func(a *A) {
+        a.v3 = v3
+    }
+}
```

The previous code that uses V2 will have a smooth transition without any breaks.

**The Problem at Scale**

Such a functional option pattern scales very ugly when we have tons of options and multiple types in the same package that need customization.

Let’s explain in more depth with another example. When types A and B sharing similar fields and both need options to customize:

```go
type A struct {
    v1 int
}

type B struct {
    v1 int
    v2 int
}
```

We will have to define two types of options separately for A and B. There is no easy way to write a unified functional option that both works for A and B, and for the same field v1, we need two versions of options V1ForA and V1ForB to manipulate:

```go
type OptionA func(a *A)
type OptionB func(a *B)

func V1ForA(v1 int) OptionA {
    return func(a *A) {
        a.v1 = v1
    }
}
```
```go
func V1ForB(v1 int) OptionB {
    return func(b *B) {
        b.v1 = v1
    }
}

func V2ForB(v2 int) OptionB {
    return func(b *B) {
        b.v2 = v2
    }
}

func NewA(opts ...OptionA) *A {
    a := &A{}
    for _, opt := range opts {
        opt(a)
    }
    return a
}

func NewB(opts ...OptionB) *B {
    b := &B{}
    for _, opt := range opts {
        opt(b)
    }
    return b
}
```

In this way, whenever we need create a new A or B, we could:

```go
fmt.Printf("%#v\n", NewA()) // &A{v1:0}
fmt.Printf("%#v\n", NewA(V1ForA(42))) // &A{v1:42}
fmt.Printf("%#v\n", NewB()) // &B{v1:0, v2:0}
fmt.Printf("%#v\n", NewB(V1ForB(42))) // &B{v1:42, v2:0}
fmt.Printf("%#v\n", NewB(V2ForB(42))) // &B{v1:0, v2:42}
fmt.Printf("%#v\n", NewB(V1ForB(42), V2ForB(42))) // &B{v1:42, v2:42}
```

Although the above workaround is possible, but the actual naming and usage really feels cumbersome, especially when these options are in a separate package where we have to supply the package name when dot import is not used (assume the package name is called pkgname):

```go
fmt.Println(pkgname.NewA())
fmt.Println(pkgname.NewA(pkgname.V1ForA(42)))
fmt.Println(pkgname.NewB())
```
fmt.Println(pkgname.NewB(pkgname.V1ForB(42)))
fmt.Println(pkgname.NewB(pkgname.V2ForB(42)))
fmt.Println(pkgname.NewB(pkgname.V1ForB(42), pkgname.V2ForB(42)))

Can we do something better?

**Using Interfaces**

A quick solution to deal with this is to use an interface where an interface that commonly represents A and B:

```go
type A struct {
    v1 int
}

type B struct {
    v1 int
    v2 int
}

type Common interface {
    /* ... */
}
```

Then we can write options as follows using a `Common` interface, and type switches:

```go
type Option func(c Common)

func V1(v1 int) Option {
    return func(c Common) {
        switch x := c.(type) {
        case *A:
            x.v1 = v1
        case *B:
            x.v1 = v1
        default:
            panic("unexpected use")
        }
    }
}

func V2(v2 int) Option {
    return func(c Common) {
        switch x := c.(type) {
        case *B:
            x.v2 = v2
        default:
            panic("unexpected use")
        }
    }
}
```
The golang.design Research Using Interfaces

```go
func NewA(opts ...Option) *A {
    a := &A{}
    for _, opt := range opts {
        opt(a)
    }
    return a
}

func NewB(opts ...Option) *B {
    b := &B{}
    for _, opt := range opts {
        opt(b)
    }
    return b
}
```

Without further changes, one can use V1 both for A and B, which is a quite
simplification from the previous use already:

```go
fmt.Printf("%#v\n", NewA()) \n// \&A{v1:0}
fmt.Printf("%#v\n", NewA(V1(42))) \n// \&A{v1:42}
fmt.Printf("%#v\n", NewB()) \n// \&B{v1:0, v2:0}
fmt.Printf("%#v\n", NewB(V1(42))) \n// \&B{v1:42, v2:0}
fmt.Printf("%#v\n", NewB(V2(42))) \n// \&B{v1:0, v2:42}
fmt.Printf("%#v\n", NewB(V1(42), V2(42))) \n// \&B{v1:42, v2:42}
```

However, not everything goes as expected. There is a heavy cost for this type
of functional options pattern: safety.

Let’s imagine when we accidentally use V2 in NewA, what will happen?

```go
fmt.Println(NewA(V2(42)))
```

```
panic: unexpected use
goroutine 1 [running]:
main.main.func6({0x104f38a20?, 0x14000122110?})
```

Clearly, code like this will result in a panic at runtime, because there is no
safety mechanism to prevent not using V2 in NewA. Furthermore, from the caller’s
perspective, unless we further look into the implementation of V2, there is no
way we could tell whether we can use V2 in NewA or not.
Using Generics (and Make Call Safer)

With the Go 1.18’s generics, we could consider using a generic version of options to simplify the previously mentioned available options further and guarantee the safety of calls.

Let’s now consider the same types A and B:

```go
type A struct {
    v1 int
}

type B struct {
    v1 int
    v2 int
}
```

Then, instead of defining a direct functional option or using a common interface, we define a generic option `Option[T]` that accepts A or B as its type parameters. In this case, the self-referred function is also a parameterized function `func(*T)`:

```go
type Option[T A | B] func(*T)
```

We can carefully constrain the type parameters of the option functions V1 and V2. Specifically, In the option function V1, is designed to use for either type A or B, therefore constraining its type parameter T also limits the possible return types of V1 to be either `Option[A]` or `Option[B]`: in the option function V2, we only intended to let it is used in type B. Hence we could permit B as its type parameter, and therefore the compiler will only instantiate the version of V2 that returns `Option[B]`.

```go
func V1[T A | B](v1 int) Option[T] {
    return func(a *T) {
        switch x := any(a).(type) {
        case *A:
            x.v1 = v1
        case *B:
            x.v1 = v1
        default:
            panic("unexpected use")
        }
    }
}

func V2[T B](v2 int) Option[T] {
    return func(a *T) {
        switch x := any(a).(type) {
        case *B:
            x.v2 = v2
```
default:
    panic("unexpected use")
}
}
}

Furthermore, in the constructor of A and B. We only permit their dedicated options, such as NewA only permits type A and NewB only allow type B as their type parameters:

```go
func NewA[T A](opts ...Option[T]) *T {
    t := new(T)
    for _, opt := range opts {
        opt(t)
    }
    return t
}

func NewB[T B](opts ...Option[T]) *T {
    t := new(T)
    for _, opt := range opts {
        opt(t)
    }
    return t
}
```

On the call side, we have:

```go
fmt.Printf("%#v\n", NewA())  // &main.A{v1:0}
fmt.Printf("%#v\n", NewA(V1[A](42)))  // &main.A{v1:42}
fmt.Printf("%#v\n", NewB())  // &main.B{v1:0, v2:0}
fmt.Printf("%#v\n", NewB(V1[B](42)))  // &main.B{v1:42, v2:0}
fmt.Printf("%#v\n", NewB(V1[B](42), V2[B](42)))  // &main.B{v1:42, v2:42}
```

With this design, the user of these APIs is safe because it is guaranteed by the compiler at compile-time, to disallow its misuse by the following errors:

```go
// ERROR: B does not implement A
_ = NewA(V2[B](42))
// ERROR: A does not implement B
_ = NewA(V2[A](42))
// ERROR: type Option[B] of V2[B](42) does not match inferred type Option[A] for Option[T]
_ = NewB(V1[A](42), V2[B](42))
// ERROR: type Option[A] of V2[A](42) does not match inferred type Option[B] for Option[T]
_ = NewB(V1[B](42), V2[A](42))
```
Conclusion
This article discussed how generics could empower a future version of functional option pattern to make such a pattern more compact and safer to use. However, there is one thing left that we could not optimize yet, which is the compiler type inference for the readability and simplicity.

In the last generics functional option design, we have calls similar to:

```
NewA(V1[A](42))
NewB(V1[B](42), V2[B](42))
```

This could become a little bit stutter when these functions and options are from a different package, say `pkgname`. In this case, we will have to write:

```
pkgname.NewA(pkgname.V1[pkgname.A](42))
```

One may wonder: can’t we avoid writing the type parameters of `V1` and `V2`?

Indeed, there is only one possibility for `V1` to satisfy the `NewA`'s type constraints because `NewA` only accepts type `A` as type parameters. If `V1` is used as the argument of `NewA`, then `V1` must return `Option[A]`, and therefore the type parameter of `V1` must be `A`; similar to `V2`.

With this observation, we could simplify our code from:

```
pkgname.NewA(pkgname.V1[pkgname.A](42))
```

to

```
pkgname.NewA(pkgname.V1(42))
pkgname.NewB(pkgname.V1(42), pkgname.V2(42))
```

With this simplification, on the caller side, we see a sort of magic function `V1` as an option, which can be used both for `NewA` and `NewB`. Unfortunately, with the current Go 1.18 generics implementation, this type of inference is not yet supported.

We have created an issue[3] for the Go team and see if this type of optimization could be possible without introducing any other flaws. Let’s looking forward to it!

References
