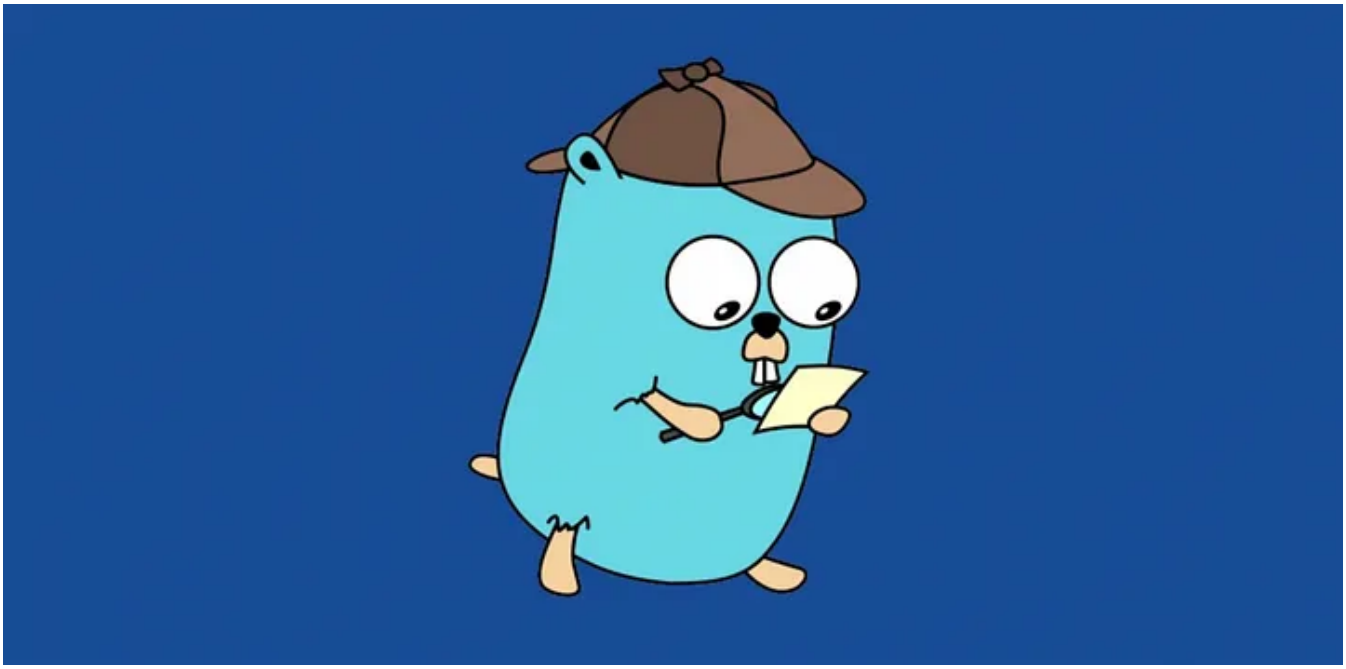
Aiden (@func25) Follow

Jan 6 · 4 min read · · Listen

Save

Golang Technique: How to Add Default Values to Function Parameters?



Have you ever been frustrated by the fact that Go doesn't have default parameter values? Well, you're not alone! This annoying limitation can make your code more tedious to write and harder to read.

All the time, you have to write extra code to check if the parameter was provided and use a default value if it wasn't.

Don't worry! There are ways to get around this limitation and add default values to your Go functions. Of course it's not as convenient as having a built-in way to do it,

but at least you won't have to carry that metaphorical umbrella around all the time.

Golang Technique Series:

- Golang Technique: How to Add Default Values to Function Parameters? (You're here)
- Golang Technique: Custom Struct Tag Like `json:"name"`

Simple wrapper

One way to do this is to define a wrapper function that calls the original function with the default values for the parameters.

If the client does not specify a name, the default name is "Aiden." Here is an example using wrapper:

```
func greet(name string) string {
 return "Hello, " + name
}

func greetWithDefaultAiden(name string) string {
 if name == "" {
  name = "Aiden"
 }
 return greet(name)
}

// you can have more than 1 default set
func greetWithDefaultJohn(name string) string {
 if name == "" {
  name = "John"
 }
 return greet(name)
}
```

By doing it this way, you can set the default value for greet without modifying any code within the greet function.

> *"But this is too much for a simple function?"*

This can be a disadvantage because it requires you to write extra code and it can make your code more difficult to read.

Hide your argument

We can place the arguments for our function in a non-exported struct, allowing the client to initialize the arguments as desired:

```
type greetingArguments struct {
 Name string
 Age  int
}

func GreetingArguments() greetingArguments {
 return greetingArguments{
  Name: "Aiden",
  Age:  30,
 }
}
```

Now let's define our Greet function:

```
func Greet(options greetingArguments) string {
  return "Hello, my name is " + options.Name + " and I am " + strconv.Itoa(opt
}
```

Every time the client wants to use the Greet function, they must use the `GreetingArguments()` function to create a `greetingArguments` struct.

This method only works when calling the function from outside the package, not from within the package.

Another option is to use a functional options pattern, which allows you to pass a variable number of options to a function as arguments. This can be more flexible and easier to read, but it can also make your code more complex.

Functional options pattern

This popular pattern is used in many libraries. In this section, I'll walk you through how to use it step by step:

1. Create a struct to hold our arguments with two fields: Name and Age.

```
type GreetingOptions struct {
  Name string
  Age  int
}
```

2. Now let's define the Greet function, taking our new struct as an argument:

```
func Greet(options GreetingOptions) string {
  return "Hello, my name is " + options.Name + " and I am " + strconv.Itoa(opt
}
```

3. This is the interesting part where we define functional options for the fields in the struct:

```
type GreetingOption func(*GreetingOptions)

func WithName(name string) GreetingOption {
  return func(o *GreetingOptions) {
    o.Name = name
  }
}

func WithAge(age int) GreetingOption {
  return func(o *GreetingOptions) {
    o.Age = age
  }
}
```

4. Create a wrapper with our new type **GreetingOption**:

```
func GreetWithDefaultOptions(options ...GreetingOption) string {
  opts := GreetingOptions{
    Name: "Aiden",
    Age:  30,
  }
  for _, o := range options {
```

```
        o(&opts)
    }
    return Greet(opts)
}
```

The `GreetWithDefaultOptions` function sets default values for the `Name (= “Aiden”)` and `Age (= 30)` fields of the `GreetingOptions` struct, and then applies the options passed as arguments to the struct.

Finally, it calls the `Greet` function with the modified struct as a parameter.

To use this code, you can call the `GreetWithDefaultOptions` function with the options you want to customize:

```
greeting := GreetWithDefaultOptions(WithName("Alice"), WithAge(20))

// "Hello, my name is Alice and I am 20 years old."
```

Many libraries utilize the functional options pattern, including mongodb, aws-sdk-go, gorm, cli, and many others.

The Next Chapter Awaits

In summary, adding default values to function arguments can be a useful way to provide flexibility and convenience in your code. Incorporating default values into your functions can be a valuable technique to have in your toolkit.

If you’re interested in staying up to date with the latest happenings in the Golang world, give me a follow. I’ll be sure to keep you in the loop!

Also, every 👋 helps to spread the word about my writing and I would really appreciate it.

Just remember to always keep learning and have fun with it, **happy coding**!

Go Golang Golang Tutorial Golang Development