Exploring structs and interfaces in Go

May 14, 2020 · 13 min read

Introduction

Go is a type-safe, statically typed, compiled programming language. The type system, with types denoted by type names and type declarations, is designed to prevent occurrences of unchecked runtime type errors.

In Go, there are several built-in types for identifiers, also known as predeclared types. They include Boolean, string, numeric (float, int, complex), and other types. Additionally, there are composite types, which are composed from predeclared types.

Composite types are mainly specified using type literals. They include arrays, interfaces, structs, functions, map types, etc. In this article, we are going to focus on the struct and interface types in Go.

Prerequisites

To easily follow along with this tutorial, it is important to have a basic understanding of Go. It is also advisable to install the binaries on our machines. Instruction to do so are available here. However, for the sake of simplicity and for the purpose of this article, we will be making use of the Go Playground for our examples.

A primer on Go

Go is a modern, fast, compiled language (i.e., it generates machine code from source code) with a lot of awesome features. With support for concurrency out of the box, it is also applicable in areas relating to low-level computer networking and systems programming.

To explore its features, we need to set it up for development. To do so, we can go ahead and install the Go binaries. We then navigate into the Go workspace folder, where we will find the `bin` , `pkg` , and `src` directories. In earlier Go versions (pre-version 1.13), source code had to written inside the `src` directory, which contains Go source files.

> *This is because Go needs a way to find, install and build source files.*

This requires us to set the `$GOPATH` environment variable on our development machines, which Go uses to identify the path to the root folder of our workspace. Therefore, in order to create a new directory inside our workspace, we had to specify the full path like this:

```
$ mkdir -p $GOPATH/src/github.com/firebase007
```

`$GOPATH` can be any path on our machine, usually `$HOME/go` , except the path to the Go installation on our machine. Inside the specified path above, we can then have package directories and, subsequently, `.go` files in that directory.

The `bin` directory contains executable Go binaries. The `go` toolchain, with its sets of commands, builds and installs binaries into this directory. The tool offers a standard way of fetching, building, and installing Go packages. The documentation for the `go` toolchain can be found here.

> ***Note**: The `pkg` directory is where Go stores a cache of pre-compiled files for subsequent compilation. More detailed information on how to write Go code with `$GOPATH` can be found here.*

Packages

Programs are grouped as packages for encapsulation, dependency management, and reusability. Packages are source files stored in the same directory and compiled together. They are stored inside a module, where a module is a group of related Go packages that performs specific operations.

> ***Note**: A Go repository typically contains only one module, which is located at the root of the repository. However, a repository can also contain more than one module.*

Nowadays, with the introduction of Go Modules in versions 1.13 and above, we would run and compile a simple Go module or program like this:

```
retina@alex Desktop % mkdir examplePackage // create a directory on our machine outside $GOPATH/src
retina@alex Desktop % cd examplePackage  // navigate into that directory
retina@alex examplePackage % go mod init github.com/firebase007/test
// choose a module path and create a go.mod file that declares that path
go: creating new go.mod: module github.com/firebase007/test
retina@Terra-011 examplePackage % ls
go.mod
```

Assuming `test` is the name of our module above, we can go ahead and create a package directory, and then create new files inside same directory. Let's look at a simple example.

```
retina@alex examplePackage % mkdir test
retina@alex examplePackage % ls
go.mod  test
retina@alex examplePackage % cd test
retina@alex test % ls
retina@alex test % touch test.go
retina@alex test % ls
test.go
retina@alex test % go run test.go
Hello, Go
retina@alex test %
```

Sample code inside the `test.go` file is shown below:

```
package main  // specifies the package name

import "fmt"

func main() {
  fmt.Println("Hello, Go")
}
```

__Note__: The `go.mod` file declares the path to a module, which also includes the import path prefix for all packages within the module. This corresponds to its location inside a workspace or in a remote repository. More details about organizing Go code with Modules can be found here.

Type system in Go

Just like the type system in other languages, Go's type system specifies a set of rules that assign a type property to variables, functions and identifiers.

Types in Go can be grouped into the following categories below:

- **String types**: Represent a set of string values, which is a slice of byte in Go. They are immutable or read-only once created. Strings are defined

types because they have methods attached to them

- **Boolean types**: Denoted by the predeclared constants `true` and `false`
- **Numeric types**: Represent sets of integer or floating-point values. They include `uint8` (or `byte`), `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` (or `rune`), `int64`, `float32`, `float64`, `complex64`, and `complex128`. These types are further categorized into signed integers, unsigned integers, and real and complex numbers. They are available in different sizes and are mostly platform-specific. More details about numeric types can be found here.
- **Array type**: Numbered collection of elements of the same type. Basically, they are building blocks for slices. Arrays are values in Go, which means that when they are assigned to a variable or passed as an argument to a function, their original values are copied, not their memory addresses
- **Slice type**: A slice is just a segment of an underlying array, or, basically, references to an underlying array. `[]T` is a slice with elements of type `T`.
- **Pointer type**: Reference type that denotes the set of all pointers to variables of a given type. Generally, pointer types hold a memory address of another variable. The zero value of a pointer is `nil`

More details about other types, like maps, functions, channels, etc., can be found on the types section of the language spec. As mentioned earlier, we are going to be focusing on the interface and struct types in this article.

Introduction to interfaces and structs

Structs

Go has struct types that contain fields of the same or different types. Structs are basically a collection of named fields that has a logical meaning or construct, wherein each field has a specific type.

Generally, struct types are combinations of user-defined types. They are specialized types because they allow us to define custom data types in such cases where the built-in types are not sufficient. Let's use an example to better understand this.

Let's say we have a blog post that we intend to publish. Using a struct type to represent the data fields would look like this:

```
type blogPost struct {
  author  string    // field
  title   string    // field
  postId  int       // field
}
// Note that we can create instances of a struct types
```

In the above struct definition, we have added different field values. Now, to instantiate or initialize the struct using a literal, we can do the following:

```
}

func main() {
        var b blogPost // initialize the struct type

        fmt.Println(b) // print the zero value

        b = blogPost{ //
        author: "Alex",
        title: "Understand struct and interface types",
        postId: 12345,
        }

        fmt.Println(b)

}

//output

{  0}  // zero values of the struct type is shown
{Alex Understand struct and interface types 12345}
```

Here is a link to the playground to run the above code.

We can also use the dot `.` operator to access individual fields in the struct type after initializing them. Let's see how we would do that with an example:

```
import "fmt"

type blogPost struct {
  author  string
  title   string
  postId  int
}

func main() {
        var b blogPost // b is a type Alias for the BlogPost
        b.author= "Alex"
        b.title="understand structs and interface types"
        b.postId=12345

        fmt.Println(b)

        b.author = "Chinedu"  // since everything is pass by value by default in Go, we can update this field after initializing – see pointer types later
```

Again, here is a link to run the code snippet above in the playground. Further, we can use the short literal notation to instantiate a struct type without using field names, as shown below:

```
package main

import "fmt"

type blogPost struct {
  author  string
  title   string
  postId  int
}

func main() {
        b := blogPost{"Alex", "understand struct and interface type",
12345}
        fmt.Println(b)

}
```

Note that with the approach above, we must always pass the field values in the same order in which they are declared in the struct type. Also, all the fields must be initialized.

Finally, if we have a struct type to be used only once inside a function, we can define them inline, as shown below:

```
  author  string
  title   string
  postId  int
}

func main() {

        // inline struct init
        b := struct {
          author  string
          title   string
          postId  int
         }{
          author: "Alex",
          title:"understand struct and interface type",
          postId: 12345,
        }

        fmt.Println(b)
}
```

__Note__: We can also initialize struct types with the `new` keyword. In that case, we can do:

```
b := new(blogPost)
```

Then, we can use the dot `.` operator to set and get the values of the fields, as we have seen earlier. Let's see an example:

```
  title   string
  postId  int
}

func main() {
        b := new(blogPost)

        fmt.Println(b) // zero value

        b.author= "Alex"
        b.title= "understand interface and struct type in Go"
        b.postId= 12345

        fmt.Println(*b)   // dereference the pointer

}

//output
&{  0}
{Alex understand interface and struct type in Go 12345}
```

__Note__: As we can see from the output, by using the `new` keyword, we allocate storage for the variable `b` , which then initializes the zero values of our struct fields — in this case `(author="", title="", postId=0)` . This then returns a pointer type `*b` , containing the address of the above variables in memory.

Here is a link to the playground to run the code. More details about the behavior of the `new` keyword can be found here.

Pointer to a struct

In our earlier examples, we have used Go's default behavior, wherein everything is passed by value. With pointers, this is not the case. Let's see with an example:

```
import "fmt"

type blogPost struct {
  author  string
  title   string
  postId  int
}

func main() {
        b := &blogPost{
                author:"Alex",
                title: "understand structs and interface types",
                postId: 12345,
                }

        fmt.Println(*b)   // dereference the pointer value

       fmt.Println("Author's name", b.author) // in this case Go would
handle the dereferencing on our behalf
```

Here is a link to the playground to run the code.

We will begin to understand the benefits of this approach as we proceed with the section on methods and interfaces.

Nested or embedded struct fields

Earlier we mentioned that struct types are composite types. Therefore, we can also have structs that are nested inside other structs. For example, suppose we have a `blogPost` and an `Author` struct, defined below:

```
type blogPost struct {
  title      string
  postId     int
  published  bool
}


type Author struct {
  firstName, lastName, Biography string
  photoId    int
}
```

Then, we can nest the `Author` struct in the `blogPost` struct like this:

```
package main

import "fmt"

type blogPost struct {
  author  Author // nested struct field
  title   string
  postId  int
  published  bool
}

type Author struct {
  firstName, lastName, Biography string
  photoId    int
}

func main() {
        b := new(blogPost)

        fmt.Println(b)
```

Here is the link to run the code in the playground.

In Go, there is a concept of promoted fields for nested struct types. In this case, we can directly access struct types defined in an embedded struct without having to go some levels deep, e.g., doing `b.author.firstName`. Let's see how we can achieve this:

```
package main

import "fmt"

type Author struct {
  firstName, lastName, Biography string
  photoId    int
}

type BlogPost struct {
  Author  // directly passing the Author struct as a field - also
called an anonymous field orembedded type
  title   string
  postId  int
  published  bool
}

func main() {
        b := BlogPost{
        Author: Author{"Alex", "Nnakwue", "I am a lazy engineer",
```

Here is a link to the playground to run the code.

> ***Note**: Go does not support inheritance, but rather composition. In the coming sections, we will learn how these concepts can be applied to struct and interface types and how we can add behavior to them with methods.*

Methods and functions in Go

Method sets

The method set of a type `T` consists of all methods declared with receiver types `T`. Note that the receiver is specified via an extra parameter preceding the method name. More details about receiver types can be found here.

> *Methods in Go are special kinds of functions with a receiver.*

In Go, we can create a type with a behavior by defining a method on that type. In essence, a method set is a list of methods that a type must have in order to implement an interface. Let's look at an example:

```
// BlogPost struct with fields defined
type BlogPost struct {
  author  string
  title   string
  postId  int
}

// Create a BlogPost type called (under) Technology
type Technology BlogPost
```

> ***Note**: We are using a struct type here because we are focusing on structs in this article. Methods can also be defined on other named types.*

```
// write a method that publishes a blogPost - accepts the Technology
type as a pointer receiver
func (t *Technology) Publish() {
    fmt.Printf("The title on %s has been published by %s, with postId
%d\n" , t.title, t.author, t.postId)
}


// Create an instance of the type
t := Technology{"Alex","understand structs and interface types",12345}


// Publish the BlogPost -- This method can only be called on the
Technology type
t.Publish()

// output
The title on understand structs and interface types has been published
by Alex, with postId 12345
```

Here is a link to the playground to run the code.

> ***Note**: Methods with pointer receivers will work on both pointers or values. However, it is not so the other way around. More details on method sets can be found here in the language spec.*

Interfaces

In Go, interfaces serve a major purpose of encapsulation and allows us write a more cleaner and robust code. By doing this we are only exposing methods and behavior in our program. As we mentioned in the last section, method sets add behavior to one or more types.

Interface types define one or more method sets. A type, therefore, is said to implement an interface by implementing its methods. In that light, interfaces enable us compose custom types that have a common behavior.

In Go, interfaces are implicit. This means that if every method belonging to the method set of an interface type is implemented by a type, then that type is said to implement the interface. To declare an interface:

```
type Publisher interface {
    publish()  error
}
```

In the `publish()` interface method we set above, if a type (e.g., a struct) implements the method, then we can say that the type implements the interface. Let's define a method that accepts a struct type `blogpost` below:

```
func (b blogPost) publish() error {
   fmt.Println("The title has been published by ", b.author)
   return nil
}
```

Now to implement the interface:

```
package main

import "fmt"

// interface definition
type Publisher interface {
     Publish()  error
}

type blogPost struct {
  author  string
  title   string
  postId  int
}

// method with a value receiver
func (b blogPost) Publish() error {
   fmt. Printf("The title on %s has been published by %s, with postId
%d\n" , b.title, b.author, b.postId)
   return nil
```

Here is a link to the playground to run the code.

We can also alias interface types like this:

```
type publishPost Publisher  // alias to the interface defined above -
only suited for third-party interfaces
```

> ***Note**: If more than one type implements the same method, the method set can form an interface type. This allows us to pass that interface type as an argument to a function that intends to implement that interface's behavior. This way, polymorphism can be achieved.*

Unlike functions, methods can only be called from an instance of the type they were defined on. The benefit is that instead of specifying a particular data type we want to accept as an argument to functions, it would be nice if we could

specify the behavior of the objects that need to be passed to that function as arguments.

Let's look at how we can use interface types as arguments to functions. To begin, let's add a method to our struct type:

```
package main

import "fmt"


type Publisher interface {
     Publish()  error
}

type blogPost struct {
  author  string
  title   string
  postId  int
}


func (b blogPost) Publish() error {
   fmt.Printf("The title on %s has been published by %s\n" , b.title,
b.author)
   return nil
```

Here is the link to run the code on the playground.

As we have previously mentioned, we can pass a method receiver either by value or by pointer type. When we pass by value, we store a copy of the value we are passing. This means that when we call the method, we are not making a change to that underlying value. However, when we pass by pointer, we are directly sharing the underlying memory address and, thus, the location of the variable declared in the underlying type.

Note: As a reminder, a type is said to implement an interface when it defines method sets available on the interface type.

Again, types are not required to nominate that they implement an interface; instead, any type implements an interface, provided it has methods whose signature matches the interface declaration.

Finally, we are going to look at the signature for embedding interface types in Go. Let's use a dummy example:

```
//embedding interfaces
type interface1 interface {
    Method1()
}

type interface2 interface {
    Method2()
}

type embeddedinterface interface {
    interface1
    interface2
}

func (s structName)  method1 (){

}

func (s structName)  method2 (){
```

Note: As a rule of thumb, when we start to have multiple types in our package implement the same method signatures, we can then begin to refactor our code and use an interface type. Doing so avoids early abstractions.

Things to note about struct types

- Field names may be specified either implicitly with a variable or as embedded types without field names. In this case, the field must be specified as a type name `T` or as a pointer to a non-interface type name `*T`
- Field names must be unique inside a struct type
- A field or a method of an embedded type can be promoted
- Promoted fields cannot be used as field names in the struct
- A field declaration may be followed by an optional string literal tag
- An exported struct field must begin with a capital letter
- Apart from basic types, we can also have function types and interface types as struct fields

More details about the struct type can be found here in the language specification.

Things to note about interface types

- The zero value of an interface is `nil`
- An empty interface contains zero methods. Note that all types implement the empty interface. This means that if you write a function that takes an empty `interface{}` value as a parameter, you can supply that function with any value
- Interfaces generally belong in the package that uses values of the interface type and not the package that implements those values

More details about the interface type can be found here in the language specification.

Conclusion

As we have learned, interface types can store the copy of a value, or a value can be shared with the interface by storing a pointer to the value's address. One important thing to note about interface types is that it is advisable not to focus on optimizing too early, as we do not want to define interfaces before they are used.

The rules for determining interface adherence or usage are based on method receivers and how the interface calls are being made. Read more about this in the Go code review and comments section here.

A quite confusing rule about pointers and values for method receivers is that while value methods can be invoked on both pointers and values, pointer methods can only be invoked on pointers. For receiver types, if a method needs to mutate the receiver, the receiver must be a pointer.

Extra details about interface types can be found in effective Go. Specifically, you can take a look at interfaces and methods, interface checks, and interface conversions and type assertions. Type assertions are more like operations applied to an underlying value of an interface type. Essentially, it is a process for extracting the values of an interface type. They are represented as `x.(T)`, where the value `x` is an interface type.

Again, thanks for reading and please feel free to add questions or comments in the comment section below, or reach out on Twitter. Go forth and keep learning 🙂

Share this:

Twitter Reddit LinkedIn Facebook

Alexander Nnakwue Follow

Software engineer. React, Node.js, Python, and other developer tools and libraries.

#go

Leave a Reply

Enter your comment here...