Chicago-based software developer, @jordanorelli on twitter. • Ask me anything

JORDAN ORELLI

Search

How to use interfaces in Go

Follow jordanorelli

Before I started programming Go, I was doing most of my work with Python. As a Python programmer, I found that learning to use interfaces in Go was extremely difficult. That is, the basics were easy, and I knew how to use the interfaces in the standard library, but it took some practice before I knew how to design my own interfaces. In this post, I'll discuss Go's type system in an effort to explain how to use interfaces effectively.

Introduction to interfaces

So what *is* an interface? An interface is two things: it is a set of methods, but it is also a type. Let's focus on the method set aspect of interfaces first.

Typically, we're introduced to interfaces with some contrived example. Let's go with the contrived example of writing some application where you're defining Animal datatypes, because that's a totally realistic situation that happens all the time. The `Animal` type will be an interface, and we'll define an `Animal` as being *anything that can speak*. This is a core concept in Go's type system; instead of designing our abstractions in terms of what kind of data our types can hold, we design our abstractions in terms of what actions our types can execute.

We start by defining our `Animal` interface:

```
type Animal interface {
    Speak() string
}
```

pretty simple: we define an `Animal` as being any type that has a method named `Speak`. The `Speak` method takes no arguments and returns a string. Any type that defines this method is said to *satisfy* the `Animal` interface. There is no `implements` keyword in Go; whether or not a type satisfies an interface is determined automatically. Let's create a couple of types that satisfy this interface:

```
type Dog struct {
}

func (d Dog) Speak() string {
    return "Woof!"
}

type Cat struct {
}

func (c Cat) Speak() string {
    return "Meow!"
}

type Llama struct {
}

func (l Llama) Speak() string {
    return "?????"
}

type JavaProgrammer struct {
}

func (j JavaProgrammer) Speak() string {
    return "Design patterns!"
}
```

We now have four different types of animals: A dog, a cat, a llama, and a Java programmer. In our `main()` function, we Animals, and put one of each type into that slice, and see what each animal says. Let’s do that now:

```
func main() {
    animals := []Animal{Dog{}, Cat{}, Llama{}, JavaProgrammer{}}
    for _, animal := range animals {
        fmt.Println(animal.Speak())
    }
}
```

You can view and run this example here:http://play.golang.org/p/yGTd4MtgD5

Great, now you know how to use interfaces, and I don’t need to talk about them any more, right? Well, no, not really. Let’s look at a few things that aren’t very obvious to the budding gopher.

The `interface{}` type

The `interface{}` type, *the empty interface*, is the source of much confusion. The `interface{}` type is the interface that has no methods. Since there is no `implements` keyword, all types implement at least zero methods, and satisfying an interface is done automatically, *all types satisfy the empty interface*. That means that if you write a function that takes an `interface{}` value as a parameter, you can supply that function with any value. So, this function:

```
func DoSomething(v interface{}) {
   // ...
}
```

will accept any parameter whatsoever.

Here’s where it gets confusing: inside of the `DoSomething` function, what is `v`’s type? Beginner gophers are led to believe that “`v` is of any type”, but that is wrong. `v` is not of *any* type; it is of `interface{}` type. Wait, what? When passing a value into the `DoSomething` function, the Go runtime will perform a type conversion (if necessary), and convert the value to an `interface{}` value. All values have exactly one type at runtime, and `v`’s one static type is `interface{}`.

This should leave you wondering: ok, so if a conversion is taking place, what is *actually* being passed into a function that takes an `interface{}` value (or, what is actually stored in an `[]Animal` slice)? An interface value is constructed of two words of data; one word is used to point to a method table for the value’s underlying type, and the other word is used to point to the actual data being held by that value. I don’t want to bleat on about this endlessly. If you understand that an interface value is two words wide and it contains a pointer to the underlying data, that’s typically enough to avoid common pitfalls. If you are curious to learn more about the implementation of interfaces, I think Russ Cox’s description of interfaces is very, very helpful.

In our previous example, when we constructed a slice of `Animal` values, we did not have to say something onerous like `Animal(Dog{})` to put a value of type `Dog` into the slice of `Animal` values, because the conversion was handled for us automatically. Within the `animals` slice, each element is of `Animal` type, but our different values have different underlying types.

So… why does this matter? Well, understanding how interfaces are represented in memory makes some potentially Follow jordanorelli y obvious. For example, the question “can I convert a []T to an []interface{}” is easy to answer once you understand how interfaces are represented in memory. Here’s an example of some broken code that is representative of a common misunderstanding of the `interface{}` type:

```
package main

import (
    "fmt"
)

func PrintAll(vals []interface{}) {
    for _, val := range vals {
        fmt.Println(val)
    }
}

func main() {
    names := []string{"stanley", "david", "oscar"}
    PrintAll(names)
}
```

Run it here:http://play.golang.org/p/4DuBoi2hJU

By running this, you can see that we encounter the following error: `cannot use names (type []string) as type []interface {} in function argument`. If we want to actually make that work, we would have to convert the `[]string` to an `[]interface{}`:

```
package main

import (
    "fmt"
)

func PrintAll(vals []interface{}) {
    for _, val := range vals {
        fmt.Println(val)
    }
}

func main() {
    names := []string{"stanley", "david", "oscar"}
    vals := make([]interface{}, len(names))
    for i, v := range names {
        vals[i] = v
    }
    PrintAll(vals)
}
```

Run it here:http://play.golang.org/p/Dhg1YS6BJS

That’s pretty ugly, but *c'est la vie*. Not everything is perfect. (in reality, this doesn’t come up very often, because `[]interface{}` turns out to be less useful than you would initially expect)

Pointers and interfaces

Another subtlety of interfaces is that an interface definition does not prescribe whether an implementor should implement the interface using a pointer receiver or a value receiver. When you are given an interface value, there’s no guarantee whether the underlying type is or isn’t a pointer. In our previous example, we defined all of our methods on value receivers, and we put the associated values into the `Animal` slice. Let's change this and make the `Cat`'s `Speak()` method take a pointer receiver:

```
func (c *Cat) Speak() string {
    return "Meow!"
}
```

If you change that one signature, and you try to run the same program exactly as-is (http://play.golang.org/p/TvR758rfre), you will see the following error:

```
prog.go:40: cannot use Cat literal (type Cat) as type Animal in array element:
    Cat does not implement Animal (Speak method requires pointer receiver)
```

This error message is a bit confusing at first, to be honest. What it's saying is not that the interface `Animal` demands that you define your method as a pointer receiver, but that you have tried to convert a `Cat` struct into an `Animal` interface value, but only `*Cat` satisfies that interface. You can fix this bug by passing in a `*Cat` pointer to the `Animal` slice instead of a `Cat` value, by using `new(Cat)` instead of `Cat{}` (you could also say `&Cat{}`, I simply prefer the look of `new(Cat)`):

```
animals := []Animal{Dog{}, new(Cat), Llama{}, JavaProgrammer{}}
```

now our program works again:http://play.golang.org/p/x5VwyExxBM

Let's go in the opposite direction: let's pass in a `*Dog` pointer instead of a `Dog` value, but this time we *won't* change the definition of the `Dog` type's `Speak` method:

```
animals := []Animal{new(Dog), new(Cat), Llama{}, JavaProgrammer{}}
```

This also works (http://play.golang.org/p/UZ618qbPkj), but recognize a subtle difference: we didn't need to change the type of the receiver of the `Speak` method. This works because a pointer type can access the methods of its associated value type, but not vice versa. That is, a `*Dog` value can utilize the `Speak` method defined on `Dog`, but as we saw earlier, a `Cat` value cannot access the `Speak` method defined on `*Cat`.

That may sound cryptic, but it makes sense when you remember the following: everything in Go is passed by value. Every time you call a function, the data you're passing into it is copied. In the case of a method with a value receiver, the value is copied when calling the method. This is slightly more obvious when you understand that a method of the following signature:

```
func (t T)MyMethod(s string) {
    // ...
}
```

is a function of type `func(T, string)`; method receivers are passed into the function by value just like any other parameter.

Any changes to the receiver made inside of a method defined on a value type (e.g., `func (d Dog) Speak() { ... }`) will not be seen by the caller because the caller is scoping a completely separate `Dog` value. Since everything is passed by value, it should be obvious why a `*Cat` method is not usable by a `Cat` value; any one `Cat` value may have any number of `*Cat` pointers that point to it. If we try to call a `*Cat` method by using a `Cat` value, we never had a `*Cat` pointer to begin with. Conversely, if we have a method on the `Dog` type, and we have a `*Dog` pointer, we know exactly which `Dog` value to use when calling this method, because the `*Dog` pointer points to exactly one `Dog` value; the Go runtime will dereference the pointer to its associated `Dog` value any time it is necessary. That is, given a *Dog value d and a method Speak on the Dog type, we can just say d.Speak(); we don’t ng like d->Speak() as we might do in other languages.

Follow jordanorelli

The real world: getting a proper timestamp out of the Twitter API

The Twitter API represents timestamps using a string of the following format:

```
"Thu May 31 00:00:01 +0000 2012"
```

of course, timestamps can be represented in any number of ways in a JSON document, because timestamps aren’t a part of the JSON spec. For the sake of brevity, I won’t put in the entire JSON representation of a tweet, but let’s take a look at how the created_at field would be handled by encoding/json:

```
package main

import (
    "encoding/json"
    "fmt"
    "reflect"
)

// start with a string representation of our JSON data
var input = `
{
    "created_at": "Thu May 31 00:00:01 +0000 2012"
}
`

func main() {
    // our target will be of type map[string]interface{}, which is a
    // pretty generic type that will give us a hashtable whose keys
    // are strings, and whose values are of type interface{}
    var val map[string]interface{}

    if err := json.Unmarshal([]byte(input), &val); err != nil {
        panic(err)
    }

    fmt.Println(val)
    for k, v := range val {
        fmt.Println(k, reflect.TypeOf(v))
    }
}
```

run it here:http://play.golang.org/p/VJAyqO3hTF

Running this application, we see that we arrive at the following output:

```
map[created_at:Thu May 31 00:00:01 +0000 2012]
created_at string
```

We can see that we’ve retrieved the key properly, but having a timestamp in a string format like that isn’t very useful. If we want to compare timestamps to see which is earlier, or see how much time has passed since the given value and the current time, using a plain string won’t be very helpful.

Let’s naively try to unmarshal this to a time.Time value, which is the standard library representation of time, and see what kind of error we get. Make the following change:

```
   var val map[string]time.Time

    if err := json.Unmarshal([]byte(input), &val); err != nil {
        panic(err)
    }
```

running this, we will encounter the following error:

```
parsing time ""Thu May 31 00:00:01 +0000 2012"" as ""2006-01-02T15:04:05Z07:00"":

    cannot parse "Thu May 31 00:00:01 +0000 2012"" as "2006"
```

that somewhat confusing error message comes from the way that Go handles the conversion of `time.Time` values to and from strings. In a nutshell, what it means is that the string representation we gave it does not match the standard time formatting (because Twitter's API was originally written in Ruby, and the default format for Ruby is not the same as the default format for Go). We'll need to define our own type in order to unmarshal this value correctly. The `encoding/json` package looks to see if values passed to `json.Unmarshal` satisfy the `json.Unmarshaler` interface, which looks like this:

```
type Unmarshaler interface {
    UnmarshalJSON([]byte) error
}
```

this is referenced in the documentation here:http://golang.org/pkg/encoding/json/#Unmarshaler

So what we need is a `time.Time` value with an `UnmarshalJSON([]byte) error` method:

```
type Timestamp time.Time

func (t *Timestamp) UnmarshalJSON(b []byte) error {
    // ...
}
```

By implementing this method we satisfy the `json.Unmarshaler` interface, causing `json.Unmarshal` to call our custom unmarshalling code when seeing a `Timestamp` value. For this case, we use a pointer method, because we want the caller to the see the changes made to the receiver. In order to set the value that a pointer points to, we dereference the pointer manually using the * operator. Inside of the `UnmarshalJSON` method, `t` represents a pointer to a `Timestamp` value. By saying `*t`, we dereference the pointer `t` and we are able to access the value that `t` points to. Remember: everything is pass-by-value in Go. That means that inside of the `UnmarshalJSON` method, the pointer `t` is not the same pointer as the pointer in its calling context; it is a copy. If you were to assign `t` to another value directly, you would just be reassigning a function-local pointer; the change would not be seen by the caller. However, the pointer inside of the method call points to the same data as the pointer in its calling scope; by dereferencing the pointer, we make our change visible to the caller.

We can make use of the `time.Parse` method, which has the signature `func(layout, value string) (Time, error)`. That is, it takes two strings: the first string is a layout string that describes how we are formatting our timestamps, and the second is the value we wish to parse. It returns a `time.Time` value, as well as an error (in case we failed to parse the timestamp for some reason). You can read more about the semantics of the layout strings in the time package documentation, but in this example we won't need to figure out the layout string manually because this layout string already exists in the standard library as the value `time.RubyDate`. So in effect, we can resolve the string "Thu May 31 00:00:01 +0000 2012" to a `time.Time` value by invoking the function `time.Parse(time.RubyDate, "Thu May 31 00:00:01 +0000 2012")`. The value we will receive is of type `time.Time`. In our example, we're interested in values of the type `Timestamp`. We can convert the `Timestamp` value by saying `Timestamp(v)`, where `v` is our `time.Time` value. Ultimately, our `UnmarshalJSON` function winds up looking like this:

```
func (t *Timestamp) UnmarshalJSON(b []byte) error {
    v, err := time.Parse(time.RubyDate, string(b[1:len(b)-1]))
    if err != nil {
        return err
    }
    *t = Timestamp(v)
    return nil
}
```

we take a subslice of the incoming byte slice because the incoming byte slice is the raw data of the JSON element and contains the quotation marks surrounding the string value; we want to chop those off before passing the string value into `time.Parse`.

Source for the entire timestamp example can be seen (and executed) here:http://play.golang.org/p/QpiFsJi-nZ

Real-world interfaces: getting an object out of an http request

Let's wrap up by seeing how we might design an interfaces to solve a common web development problem: we wish to parse the body of an HTTP request into some object data. At first, this is not a very obvious interface to define. We might try to say that we're going to get a resource from an HTTP request like this:

```
GetEntity(*http.Request) (interface{}, error)
```

because an `interface{}` can have any underlying type, so we can just parse our request and return whatever we want. This turns out to be a pretty bad strategy, the reason being that we wind up sticking too much logic into the `GetEntity` function, the `GetEntity` function now needs to be modified for every new type, and we'll need to use a type assertion to do anything useful with that returned `interface{}` value. In practice, functions that return `interface{}` values tend to be quite annoying, and as a rule of thumb you can just remember that it's typically better to take in an `interface{}` value as a parameter than it is to return an `interface{}` value. (Postel's Law, applied to interfaces)

We might also be tempted to write some type-specific function like this:

```
GetUser(*http.Request) (User, error)
```

This also turns out to be pretty inflexible, because now we have different functions for every type, but no sane way to generalize them. Instead, what we really want to do is something more like this:

```
type Entity interface {
    UnmarshalHTTP(*http.Request) error
}

func GetEntity(r *http.Request, v Entity) error {
    return v.UnmarshalHTTP(r)
}
```

Where the `GetEntity` function takes an interface value that is guaranteed to have an `UnmarshalHTTP` method. To make use of this, we would define on our `User` object some method that allows the `User` to describe how it would get itself out of an HTTP request:

```
func (u *User) UnmarshalHTTP(r *http.Request) error {
   // ...
}
```

in your application code, you would declare a var of `User` type, and then pass a pointer to this function into `GetEntity`:

```
var u User
if err := GetEntity(req, &u); err != nil {
    // ...
}
```

That's very similar to how you would unpack JSON data. This type of thing works consistently and safely because the statement `var u User` will automatically zero the `User` struct. Go is not like some other languages in that declaration and initialization occur separately, and that by declaring a value without initializing it you can create a subtle pitfall wherein you might access a section of junk data; when declaring the value, the runtime will zero the appropriate memory space to hold that value. Even if our UnmarshalHTTP method fails to utilize some fields, those fields will contain valid zero data instead of junk.

That should seem weird to you if you're a Python programmer, because it's basically inside-out of what we typically do in Python. The reason that this form becomes so handy is that now we can define an arbitrary number of types, each of which is responsible for its own unpacking from an http request. It is now up to the entity definitions to decide how they may be represented. Then, we can build around the `Entity` type to create things like generic HTTP handlers.

Wrapping up

I hope, after reading this, that you feel more comfortable using interfaces in Go. Remember the following:

- create abstractions by considering the functionality that is common between datatypes, instead of the fields that are common between datatypes
- an `interface{}` value is not of any type; it is of `interface{}` type
- interfaces are two words wide; schematically they look like `(type, value)`
- it is better to accept an `interface{}` value than it is to return an `interface{}` value
- a pointer type may call the methods of its associated value type, but not vice versa
- everything is pass by value, even the receiver of a method
- an interface value isn't strictly a pointer or not a pointer, it's just an interface
- if you need to completely overwrite a value inside of a method, use the `*` operator to manually dereference a pointer

Ok, I think that sums up everything about interfaces that I personally found confusing. Happy coding :)

8:07 am • 1 October 2012 • 119 notes

1. washingtonwellness said: bewellwashington.net/bw...
2. nouveangel liked this
3. atomicityblr liked this
4. swordfish9919 liked this
5. rayincloud-blog liked this

Follow jordanorelli

6. cute--japan--peggy--jones liked this
7. kosovo0275 liked this
8. shilinlee reblogged this from jordanorelli and added:

   值得学习，go的接口

9. bluewolfchung reblogged this from jordanorelli
10. bluewolfchung liked this
11. myhgew liked this
12. maximeha liked this
13. sinhtobo liked this
14. ournewoverlords liked this
15. danyparc liked this
16. faintedtowers liked this
17. pawka liked this
18. steamycurtains liked this
19. fobo66 liked this
20. gaocegege liked this
21. hiro-prog liked this
22. alanzabalegui liked this
23. titpetric liked this
24. cheezbot liked this
25. yourgan liked this
26. darkspotcorrectors liked this
27. bestdarkspotcorrectors liked this
28. yamidev liked this
29. vantischen liked this
30. kenng-simply reblogged this from jordanorelli
31. kenng-simply liked this
32. darkfoxh4ck3r liked this
33. aneaglenamedsmallgovernment liked this
34. arganoilhome liked this
35. michaelymg liked this
36. campbellsouped liked this
37. larevoluciondeldia liked this
38. alanzafame liked this
39. webhat liked this
40. pipefail liked this
41. sevvvil liked this
42. footloosemoose liked this
43. v2rred liked this
44. lumengxi liked this
45. jordanorelli posted this
46. Show more notes

Back • Next

Just Plain Theme by Peter Vidani
Powered by Tumblr