How do I copy a struct

Asked 6 years, 4 months ago Modified 2 years ago Viewed 170k times

83

I want to copy an object so that I have two identical objects with two different memory addresses. My first attempt at this has failed:

```
aa := a
assert.NotEqual(t, &a, &aa, "Copied items should not be the same object.") //
Test fails
```

Can I fix this so that it really does a copy of the struct? There's nothing special about this structure.

go

Share
Improve this question
Follow

edited Nov 24, 2022 at 15:14
Jonathan Hall
79.3k ● 18 ● 157 ● 201

asked Aug 1, 2018 at 14:22
Salim Fadhley
8,085 ● 18 ● 54 ● 89

1 Please post more code to regenerate the issue you are facing. And you can use `reflect.DeepEqual` to check if two structs are equal. – Himanshu Aug 1, 2018 at 14:23

3 If you're using testify, the docs say: `Pointer variable equality is determined based on the equality of the referenced values (as opposed to the memory addresses).` So this comparison probably isn't doing what you think it is in the first place. – JimB Aug 1, 2018 at 14:27

3 If `a` and `aa` are both value types (not pointers), then you're already creating a copy. If they're not, dereference in the assignment to create a copy: `aa := *a` . – Adrian Aug 1, 2018 at 14:28

1 The addresses are different, but assert compares the values pointed to. – Peter Aug 1, 2018 at 14:29

2 You may want the new `assert.NotSame()` – Jeff Widman Feb 20, 2020 at 1:11

6 Answers

Sorted by: Highest score (default)

177

In go, primitive types, and structs containing only primitive types, are copied by value, so you can copy them by simply assigning to a new variable (or returning from a function). For example:

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

alice1 := Person{"Alice", 30}
alice2 := alice1
fmt.Println(alice1 == alice2)   // => true, they have the same field values
fmt.Println(&alice1 == &alice2) // => false, they have different addresses

alice2.Age += 10
fmt.Println(alice1 == alice2)   // => false, now they have different field
values
```

Note that, as mentioned by commenters, the confusion in your example is likely due to the semantics of the test library you are using.

If your struct happens to include arrays, slices, or pointers, then you'll need to perform a deep copy of the referenced objects unless you want to retain references between copies. Golang provides no builtin deep copy functionality so you'll have to implement your own or use one of the many freely available libraries that provide it.

Share

Improve this answer

Follow

edited Aug 1, 2018 at 16:48

answered Aug 1, 2018 at 16:27

maerics

156k ● 47 ● 276 ● 299

15 seeing is believing play.golang.org/p/dZD8xoW_kvS – Alok Kumar Singh Sep 9, 2020 at 13:54

Arrays, specifically, are passed as values, though, as opposed to slices. Correct me if I'm wrong. go.dev/play/p/GzRAocbcBN5 – m_ocean Apr 1, 2023 at 15:09

12

DeepCopy is a very heavy operation and hence should be avoided if possible. For complex structures such as the following, we can optimise the code.

```
type Address struct {
    city  string
    state string
}

type Person struct {
    age     int
    name    string
    address []Address
}

p := Person{
    age:  20,
    name: "Jack",
    address: []Address{
        {
            city:  "city1",
            state: "state1",
        }, {
            city:  "city2",
            state: "state2",
        },
    },
}

var q Person
q.age = p.age
q.name = p.name
q.address = append(q.address, p.address...)
q.address[0].city = "cityx"
```

Result:

```
p object:
{20 Jack [{city1 state1} {city2 state2}]}

q object:
{20 Jack [{cityx state1} {city2 state2}]}
```

Inference:
As can be seen from the above example, the p object did not change when q was changed. This approach can be used in nested array of structs.

Share

Improve this answer

Follow

edited Dec 24, 2019 at 10:57
Attilio
1,712 ● 1 ● 21 ● 29

answered Dec 17, 2019 at 17:58
Sezal Agrawal
151 ● 1 ● 6

10

Be careful, if your source `struct` is actually a pointer, then a normal assign won't work:

```
package main
import "net/http"

func main() {
   a, err := http.NewRequest("GET", "https://stackoverflow.com", nil)
   if err != nil {
      panic(err)
   }
   b := a.URL
   b.Host = "superuser.com"
   println(a.URL.Host == "superuser.com")
}
```

Instead, you need to dereference the pointer:

```
package main
import "net/http"

func main() {
   a, err := http.NewRequest("GET", "https://stackoverflow.com", nil)
   if err != nil {
      panic(err)
   }
   b := *a.URL
   b.Host = "superuser.com"
   println(a.URL.Host == "stackoverflow.com")
}
```

Share

Improve this answer

Follow

edited Oct 29, 2021 at 16:02

Zoe - Save the data dump ♦

28.1k ● 22 ● 127 ● 158

answered Apr 17, 2021 at 19:20

Zombo

1

3

You can use a function with pass by value and return the argument untouched or changed depending on your needs.

Using the structs from above:

```
func main() {
    copyOf := func(y Person) Person {
        y.name = "Polonius"
        y.address = append(y.address, Address{
            city:  "other city",
            state: "other state",
        })
        return y
    }

    p := Person{
        age:  20,
        name: "Jack",
        address: []Address{
            {
                city:  "city1",
                state: "state1",
            }, {
                city:  "city2",
                state: "state2",
            },
        },
    }

    q := copyOf(p)

    fmt.Printf("Orig %v, \naddrss: %p \n\n", p, &p)
    fmt.Printf("Copy %v, \naddress: %p\n\n", q, &q)
}
```

Share Improve this answer Follow

answered Mar 11, 2020 at 20:27
Bogdan Costea
69 ● 1 ● 3

3

You could try `gob.Encode` then `gob.Decode`, like we do in javascript: first `JSON.stringify` then `JSON.parse`.

In golang you should use `gob` instead of `json.Marshal/json.Unmarshal` because you don't need to add json tags on fields, and you can handle interface by using gob. Every interface used in you struct field should be called by `gob.Register`

```
import (
    "bytes"
    "encoding/gob"
)

func DeepCopy(src, dist interface{}) (err error){
    buf := bytes.Buffer{}
    if err = gob.NewEncoder(&buf).Encode(src); err != nil {
        return
    }
    return gob.NewDecoder(&buf).Decode(dist)
}
```

`dist` should always be a pointer

Share Improve this answer Follow

answered Feb 26, 2022 at 13:44
Plasmatium
624 ● 6 ● 7

2 It's easy to use, but about 100 times slower than recursively directly copying the fields/members. – Gergely Máté Sep 20, 2022 at 11:50

It's slower than manual copy fields from struct, but saving programer's time. Also we can try Rust way. Rust using macro to generate fields copy code, in golang we have go generate. We can analyze the struct fields and generate copy code automatically, by go generate
– Plasmatium Oct 11 at 2:14

0

The easiest way of doing this is using a library (I'd recommend https://github.com/jinzhu/copier) combined with some reflection magic:

```
var cloneObj MyCustomStruct =
reflect.New(reflect.ValueOf(sourceObj).Elem().Type()).Interface()
copier.Copy(cloneObj, sourceObj)
```

This gives you a deep copy of the source struct correctly typed (which is important). If you don't use an interface, you may need to customize the reflection use a bit (remove the `.Interface()`).

Share Improve this answer Follow

answered Dec 12, 2022 at 8:37

Michael Kleinhenz
473 ● 6 ● 11