

## 37 | 编程范式游记 ( 8 ) - Go 语言的委托模式

2018-02-06 陈皓

左耳听风


[进入课程 >](#)



我们再来看 Go 语言这个模式，Go 语言的这个模式挺好玩儿的。声明一个 struct，跟 C 很一样，然后直接把这个 struct 类型放到另一个 struct 里。

### 委托的简单示例

我们来看几个示例：

 复制代码

```
1 type Widget struct {
2     X, Y int
3 }
4
5 type Label struct {
6     Widget      // Embedding (delegation)
7     Text string // Aggregation
8     X int       // Override
9 }
10
11 func (label Label) Paint() {
```

```
15 }
```

上面，

我们声明了一个 `Widget`，其有 `X,Y`；

然后用它来声明一个 `Label`，直接把 `Widget` 委托进去；

然后再给 `Label` 声明并实现了一个 `Paint()` 方法。

于是，我们就可以这样编程了：

[📄 复制代码](#)

```
1 label := Label{Widget{10, 10}, "State", 100}
2
3 // X=100, Y=10, Text=State, Widget.X=10
4 fmt.Printf("X=%d, Y=%d, Text=%s Widget.X=%d\n",
5     label.X, label.Y, label.Text,
6     label.Widget.X)
7 fmt.Println()
8 // {Widget:{X:10 Y:10} Text:State X:100}
9 // {{10 10} State 100}
10 fmt.Printf("%+v\n%v\n", label, label)
11
12 label.Paint()
```

我们可以看到，如果有成员变量重名，则需要手动地解决冲突。

我们继续扩展代码。


先来一个 `Button`：

[📄 复制代码](#)

```
1 type Button struct {
2     Label // Embedding (delegation)
3 }
```


```
7 }
8 func (button Button) Paint() { // Override
9     fmt.Printf("[%p] - Button.Paint(%q)\n",
10         &button, button.Text)
11 }
12 func (button Button) Click() {
13     fmt.Printf("[%p] - Button.Click()\n", &button)
14 }
```

再来一个 ListBox：


 复制代码

```
1 type ListBox struct {
2     Widget          // Embedding (delegation)
3     Texts []string // Aggregation
4     Index int       // Aggregation
5 }
6 func (listBox ListBox) Paint() {
7     fmt.Printf("[%p] - ListBox.Paint(%q)\n",
8         &listBox, listBox.Texts)
9 }
10 func (listBox ListBox) Click() {
11     fmt.Printf("[%p] - ListBox.Click()\n", &listBox)
12 }
```

然后，声明两个接口用于多态：

 复制代码

```
1 type Painter interface {
2     Paint()
3 }
4
5 type Clicker interface {
6     Click()
7 }
```


 复制代码

```
1 button1 := Button{Label{Widget{10, 70}, "OK", 10}}
2 button2 := NewButton(50, 70, "Cancel")
3 listBox := ListBox{Widget{10, 40},
4     []string{"AL", "AK", "AZ", "AR"}, 0}
5
6 fmt.Println()
7 //[0xc4200142d0] - Label.Paint("State")
8 //[0xc420014300] - ListBox.Paint(["AL" "AK" "AZ" "AR"])
9 //[0xc420014330] - Button.Paint("OK")
10 //[0xc420014360] - Button.Paint("Cancel")
11 for _, painter := range []Painter{label, listBox, button1, button2} {
12     painter.Paint()
13 }
14
15 fmt.Println()
16 //[0xc420014450] - ListBox.Click()
17 //[0xc420014480] - Button.Click()
18 //[0xc4200144b0] - Button.Click()
19 for _, widget := range []interface{}{label, listBox, button1, button2} {
20     if clicker, ok := widget.(Clicker); ok {
21         clicker.Click()
22     }
23 }
```

## 一个 Undo 的委托重构

上面这个是 Go 语中的委托和接口多态的编程方式，其实是面向对象和原型编程综合的玩法。这个玩法可不可以玩得更有意思呢？这是可以的。


首先，我们先声明一个数据容器，其中有 `Add()`、`Delete()` 和 `Contains()` 方法。还有一个转字符串的方法。

 复制代码

```
1 type IntSet struct {
2     data map[int]bool
3 }
4
5 func NewIntSet() IntSet {
6     return IntSet{make(map[int]bool)}
7 }
```

```
11 }
12
13 func (set *IntSet) Delete(x int) {
14     delete(set.data, x)
15 }
16
17 func (set *IntSet) Contains(x int) bool {
18     return set.data[x]
19 }
20
21 func (set *IntSet) String() string { // Satisfies fmt.Stringer interface
22     if len(set.data) == 0 {
23         return "{}"
24     }
25     ints := make([]int, 0, len(set.data))
26     for i := range set.data {
27         ints = append(ints, i)
28     }
29     sort.Ints(ints)
30     parts := make([]string, 0, len(ints))
31     for _, i := range ints {
32         parts = append(parts, fmt.Sprint(i))
33     }
34     return "{" + strings.Join(parts, ",") + "}"
35 }
```

我们如下使用这个数据容器：

 复制代码

```
1 ints := NewIntSet()
2 for _, i := range []int{1, 3, 5, 7} {
3     ints.Add(i)
4     fmt.Println(ints)
5 }
6 for _, i := range []int{1, 2, 3, 4, 5, 6, 7} {
7     fmt.Print(i, ints.Contains(i), " ")
8     ints.Delete(i)
9     fmt.Println(ints)
10 }
```

这个数据容器平淡无奇，我们想给它加一个 Undo 的功能。我们可以这样来：

```

2  IntSet // Embedding (delegation)
3  functions []func()
4  }
5
6  func NewUndoableIntSet() UndoableIntSet {
7      return UndoableIntSet{NewIntSet(), nil}
8  }
9
10 func (set *UndoableIntSet) Add(x int) { // Override
11     if !set.Contains(x) {
12         set.data[x] = true
13         set.functions = append(set.functions, func() { set.Delete(x) })
14     } else {
15         set.functions = append(set.functions, nil)
16     }
17 }
18
19 func (set *UndoableIntSet) Delete(x int) { // Override
20     if set.Contains(x) {
21         delete(set.data, x)
22         set.functions = append(set.functions, func() { set.Add(x) })
23     } else {
24         set.functions = append(set.functions, nil)
25     }
26 }
27
28 func (set *UndoableIntSet) Undo() error {
29     if len(set.functions) == 0 {
30         return errors.New("No functions to undo")
31     }
32     index := len(set.functions) - 1
33     if function := set.functions[index]; function != nil {
34         function()
35         set.functions[index] = nil // Free closure for garbage collection
36     }
37     set.functions = set.functions[:index]
38     return nil
39 }

```

于是就可以这样使用了：

复制代码

```

1 ints := NewUndoableIntSet()
2 for _, i := range []int{1, 3, 5, 7} {
3     ints.Add(i)

```

```
7     fmt.Println(i, ints.Contains(i), " ")
8     ints.Delete(i)
9     fmt.Println(ints)
10 }
11 fmt.Println()
12 for {
13     if err := ints.Undo(); err != nil {
14         break
15     }
16     fmt.Println(ints)
17 }
```

但是，需要注意的是，我们用了一个新的 `UndoableIntSet` 几乎重写了所有的 `IntSet` 和 “写” 相关的方法，这样就可以把操作记录下来，然后 **Undo** 了。

但是，可能别的类也需要 `Undo` 的功能，我是不是要重写所有的需要这个功能的类啊？这样的代码类似，就是因为数据容器不一样，我就要去重写它们，这太二了。


我们能不能利用前面学到的泛型编程、函数式编程、IoC 等范式来把这个事干得好一些呢？当然是可以的。

如下所示：

我们先声明一个 `Undo []` 的函数数组（其实是一个栈）。

并实现一个通用 `Add()`。其需要一个函数指针，并把这个函数指针存放到 `Undo []` 函数数组中。

在 `Undo()` 的函数中，我们会遍历 `Undo []` 函数数组，并执行之，执行完后就弹栈。

 复制代码

```
1 type Undo []func()
2
3 func (undo *Undo) Add(function func()) {
4     *undo = append(*undo, function)
5 }
6
7 func (undo *Undo) Undo() error {
```

```
11     }
12     index := len(functions) - 1
13     if function := functions[index]; function != nil {
14         function()
15         functions[index] = nil // Free closure for garbage collection
16     }
17     *undo = functions[:index]
18     return nil
19 }
20
```

那么我们的 `IntSet` 就可以改写成如下的形式：

[复制代码](#)

```
1 type IntSet struct {
2     data map[int]bool
3     undo Undo
4 }
5
6 func NewIntSet() IntSet {
7     return IntSet{data: make(map[int]bool)}
8 }
9
```

然后在其中的 `Add` 和 `Delete` 中实现 `Undo` 操作。

`Add` 操作时加入 `Delete` 操作的 `Undo`。

`Delete` 操作时加入 `Add` 操作的 `Undo`。

[复制代码](#)

```
1
2 func (set *IntSet) Add(x int) {
3     if !set.Contains(x) {
4         set.data[x] = true
5         set.undo.Add(func() { set.Delete(x) })
6     } else {
7         set.undo.Add(nil)
8     }
9 }
```



```
12     if set.Contains(x) {
13         delete(set.data, x)
14         set.undo.Add(func() { set.Add(x) })
15     } else {
16         set.undo.Add(nil)
17     }
18 }
19
20 func (set *IntSet) Undo() error {
21     return set.undo.Undo()
22 }
23
24 func (set *IntSet) Contains(x int) bool {
25     return set.data[x]
26 }
27
```

我们再次看到，Go 语言的 Undo 接口把 Undo 的流程给抽象出来，而要怎么 Undo 的事交给了业务代码来维护（通过注册一个 Undo 的方法）。这样在 Undo 的时候，就可以回调这个方法来做与业务相关的 Undo 操作了。

## 小结

这是不是和最一开始的 C++ 的泛型编程很像？也和 map、reduce、filter 这样的只关心控制流程，不关心业务逻辑的做法很像？而且，一开始用一个 UndoableIntSet 来包装 IntSet 类，到反过来在 IntSet 里依赖 Undo 类，这就是控制反转 IoC。

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。**这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。**

[编程范式游记（1）- 起源](#)

[编程范式游记（2）- 泛型编程](#)

[编程范式游记（3）- 类型系统和泛型的本质](#)

[编程范式游记（4）- 函数式编程](#)


[编程范式游记（5）- 修饰器模式](#)

[编程范式游记 \( 8 \) - Go 语言的委托模式](#)

[编程范式游记 \( 9 \) - 编程的本质](#)

[编程范式游记 \( 10 \) - 逻辑编程范式](#)

[编程范式游记 \( 11 \) - 程序世界里的编程范式](#)

 极客时间

# 左耳朵耗子

全年独家专栏《左耳听风》

20000 名程序员的练级攻略

陈皓

资深技术专家  
骨灰级程序员



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 36 | [编程范式游记 \( 7 \) - 基于原型的编程范式](#)

下一篇 38 | [编程范式游记 \( 9 \) - 编程的本质](#)

## 精选留言 (8)

 写留言



Yayu

2018-07-13

 4

文章内容很棒，对于一位 golang 新手很有指导意义。但是，文章有大篇幅代码，只能在

**milley**

2018-02-06

👍 3

这样的代码和思维只能说赏心悦目！

展开 ▾

**Z3**

2018-02-06

👍 1

```
sort.Ints(ints) parts := make([]string, 0, len(ints)) for _, i := range ints {
```

这块要sort吗？能否直接for ( i=0 ; i<len ) print ints[i]

展开 ▾

**小破**

2018-02-06

👍 1

几个月前听到代码时间做节目，陈老师讲的内容让我感觉很实在，今天终于跟过来了😊

**亢 (知行合...**

2019-02-26

👍

依赖的东西要可靠、稳定，也就是接口。

业务与控制分离，控制就可以复用。

把变化频率不同的事物分开。

展开 ▾

**寻找的人cs**

2019-02-06

👍

web端功能多一点就好了，比如显示文章列表的时候感觉不如app端那么清爽

**xiao豪**

2018-02-08

👍



下载APP



**Andylee**

2018-02-06



这样写的undo在第一次插入过后，可以无限撤销了吧

展开 ∨