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：

```
type Widget struct {
    X, Y int
}

type Label struct {
    Widget        // Embedding (delegation)
    Text   string // Aggregation
    X int         // Override
}

func (label Label) Paint() {
    // [0xc4200141e0] - Label.Paint("State")
    fmt.Printf("[%p] - Label.Paint(%q)\n",
        &label, label.Text)
}
```

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：

```
label := Label{Widget{10, 10}, "State", 100}

// X=100, Y=10, Text=State, Widget.X=10
fmt.Printf("X=%d, Y=%d, Text=%s Widget.X=%d\n",
    label.X, label.Y, label.Text,
    label.Widget.X)
fmt.Println()
// {Widget:{X:10 Y:10} Text:State X:100}
// {{10 10} State 100}
fmt.Printf("%+v\n%v\n", label, label)

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`：

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

func NewButton(x, y int, text string) Button {
    return Button{Label{Widget{x, y}, text, x}}
}
func (button Button) Paint() { // Override
    fmt.Printf("[%p] - Button.Paint(%q)\n",
        &button, button.Text)
}
func (button Button) Click() {
    fmt.Printf("[%p] - Button.Click()\n", &button)
}
```

再来一个 `ListBox`：

```
type ListBox struct {
    Widget          // Embedding (delegation)
    Texts  []string // Aggregation
    Index  int      // Aggregation
}
func (listBox ListBox) Paint() {
    fmt.Printf("[%p] - ListBox.Paint(%q)\n",
        &listBox, listBox.Texts)
}
func (listBox ListBox) Click() {
    fmt.Printf("[%p] - ListBox.Click()\n", &listBox)
}
```

然后，声明两个接口用于多态：

```
type Painter interface {
    Paint()
}

type Clicker interface {
    Click()
}
```

于是我们就可以这样泛型地使用（注意其中的两个for循环）：

```
button1 := Button{Label{Widget{10, 70}, "OK", 10}}
button2 := NewButton(50, 70, "Cancel")
listBox := ListBox{Widget{10, 40},
    []string{"AL", "AK", "AZ", "AR"}, 0}

fmt.Println()
//[0xc4200142d0] - Label.Paint("State")
//[0xc420014300] - ListBox.Paint(["AL" "AK" "AZ" "AR"])
//[0xc420014330] - Button.Paint("OK")
//[0xc420014360] - Button.Paint("Cancel")
for _, painter := range []Painter{label, listBox, button1, button2} {
    painter.Paint()
}

fmt.Println()
//[0xc420014450] - ListBox.Click()
//[0xc420014480] - Button.Click()
//[0xc4200144b0] - Button.Click()
for _, widget := range []interface{}{label, listBox, button1, button2} {
    if clicker, ok := widget.(Clicker); ok {
        clicker.Click()
    }
}
```

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。

```
type IntSet struct {
    data map[int]bool
}

func NewIntSet() IntSet {
    return IntSet{make(map[int]bool)}
}

func (set *IntSet) Add(x int) {
    set.data[x] = true
}

func (set *IntSet) Delete(x int) {
    delete(set.data, x)
}

func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}

func (set *IntSet) String() string { // Satisfies fmt.Stringer interface
    if len(set.data) == 0 {
        return "{}"
    }
    ints := make([]int, 0, len(set.data))
    for i := range set.data {
        ints = append(ints, i)
    }
    sort.Ints(ints)
    parts := make([]string, 0, len(ints))
    for _, i := range ints {
        parts = append(parts, fmt.Sprint(i))
    }
    return "{" + strings.Join(parts, ",") + "}"
}
```

我们如下使用这个数据容器：

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

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：

```
type UndoableIntSet struct { // Poor style
    IntSet    // Embedding (delegation)
    functions []func()
}

func NewUndoableIntSet() UndoableIntSet {
    return UndoableIntSet{NewIntSet(), nil}
}

func (set *UndoableIntSet) Add(x int) { // Override
    if !set.Contains(x) {
        set.data[x] = true
        set.functions = append(set.functions, func() { set.Delete(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}

func (set *UndoableIntSet) Delete(x int) { // Override
    if set.Contains(x) {
        delete(set.data, x)
        set.functions = append(set.functions, func() { set.Add(x) })
    } else {
        set.functions = append(set.functions, nil)
    }
}

func (set *UndoableIntSet) Undo() error {
    if len(set.functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(set.functions) - 1
    if function := set.functions[index]; function != nil {
        function()
        set.functions[index] = nil // Free closure for garbage collection
    }
    set.functions = set.functions[:index]
    return nil
}
```

于是就可以这样使用了：

```
ints := NewUndoableIntSet()
for _, i := range []int{1, 3, 5, 7} {
    ints.Add(i)
    fmt.Println(ints)
}
for _, i := range []int{1, 2, 3, 4, 5, 6, 7} {
    fmt.Println(i, ints.Contains(i), " ")
    ints.Delete(i)
    fmt.Println(ints)
}
fmt.Println()
for {
    if err := ints.Undo(); err != nil {
        break
    }
    fmt.Println(ints)
}
```

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。

```
type Undo []func()

func (undo *Undo) Add(function func()) {
    *undo = append(*undo, function)
}

func (undo *Undo) Undo() error {
    functions := *undo
    if len(functions) == 0 {
        return errors.New("No functions to undo")
    }
    index := len(functions) - 1
    if function := functions[index]; function != nil {
        function()
        functions[index] = nil // Free closure for garbage collection
    }
    *undo = functions[:index]
    return nil
}
```

那么我们的 `IntSet` 就可以改写成如下的形式：

```
type IntSet struct {
    data map[int]bool
    undo Undo
}

func NewIntSet() IntSet {
    return IntSet{data: make(map[int]bool)}
}
```

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。

```
func (set *IntSet) Add(x int) {
    if !set.Contains(x) {
        set.data[x] = true
        set.undo.Add(func() { set.Delete(x) })
    } else {
        set.undo.Add(nil)
    }
}

func (set *IntSet) Delete(x int) {
    if set.Contains(x) {
        delete(set.data, x)
        set.undo.Add(func() { set.Add(x) })
    } else {
        set.undo.Add(nil)
    }
}

func (set *IntSet) Undo() error {
    return set.undo.Undo()
}

func (set *IntSet) Contains(x int) bool {
    return set.data[x]
}
```

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（6）- 面向对象编程
- 编程范式游记（7）- 基于原型的编程范式
- 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