

面向对象

面向对象三要素：

- 封装：将属性（数据）和方法（操作）封装，提供访问控制，隐藏实现细节，暴露该暴露的
- 继承：子类可以从父类直接获得属性和方法，减少重复定义。子类中如果与父类不同，可以自己定义新的属性和方法，也可以覆盖同名的属性和方法
- 多态：前提是继承和覆盖，使得子类中虽然使用同一个方法，但是不同子类表现不同，就是不同的态

实现了以上特征的语言，才能成为面向对象编程范式语言。

严格意义来说，Go语言就是不想实现面向对象编程范式。但是面向对象又有一些不错的特性，Go语言通过组合的方式实现了类似的功能。

只能说，Go语言实现了一种非常有自我特征的面向对象。

封装

通过结构体，可以把数据字段封装在内，还可以为结构体提供方法。

访问控制：

- 属性、方法标识符首字母大写，实现了包外可见的访问控制
- 属性、方法标识符首字母小写，包内可见
- 这些一定程度上实现了public、private的访问控制

构造函数

Go没有提供类似C++、Java一样的构造函数、析构函数。在Go中，用构造结构体实例的函数，这个函数没有特别的要求，只要返回结构体实例或其指针即可（建议返回指针，不然返回值会拷贝）。习惯上，构造函数命名是New或new开头。如果有多个构造函数，可以使用不同命名函数，因为Go也没有函数重载。

```
1  type Animal struct {
2      name string
3      age  int
4  }
5
6  func NewDefaultAnimal() *Animal {
7      return &Animal{"nobody", 1}
8  }
9
10 func NewAnimal(name string, age int) *Animal {
11     return &Animal{name, age}
12 }
```

通过不同的函数名来模拟构造函数重载

继承

Go语言没有提供继承的语法，实际上需要通过匿名结构体嵌入（组合）来实现类似效果。

```
1 package main
2
3 import "fmt"
4
5 type Animal struct {
6     name string
7     age  int
8 }
9
10 func (*Animal) run() {
11     fmt.Println("Animal run~~~")
12 }
13
14 type Cat struct {
15     Animal // 匿名结构体嵌入
16     color string
17 }
18
19 func main() {
20     cat := new(Cat)
21     cat.run()
22     cat.Animal.run()
23 }
```

通过匿名结构体嵌入，子结构体就拥有了父结构体的属性name、age，和run方法。

覆盖

覆盖override，也称重写。注意不是重载overload。

```
1 func (*Cat) run() {
2     fmt.Println("Cat run+++")
3 }
```

为Cat增加一个run方法，这就是覆盖。特别注意 `cat.run()` 和 `cat.Animal.run()` 的区别。

上例增加run方法是完全覆盖，就是不依赖父结构体方法，重写功能。

如果是依赖父结构体方法，那就要在子结构体方法中调用它。

```
1 func (c *Cat) run() {
2     c.run()           // 可以吗？
3     c.Animal.run()   // 可以吗？
4     fmt.Println("Cat run+++")
5 }
```

`cat.run()` 这是无限递归调用，小心！

`c.Animal.run()` 这是调用父结构体方法。

多态

Go语言不能像Java语言一样使用多态。可以通过引入interface接口来解决。

```
1 package main
2
3 import "fmt"
4
5 type Runner interface {
6     run()
7 }
8
9 type Animal struct {
10     name string
11     age  int
12 }
13
14 func (*Animal) run() {
15     fmt.Println("Animal run~~~")
16 }
17
18 type Cat struct {
19     Animal // 匿名结构体嵌入
20     color  string
21 }
22
23 func (c *Cat) run() {
24     c.Animal.run()
25     fmt.Println("Cat run+++")
26 }
27
28 type Dog struct {
29     Animal // 匿名结构体嵌入
30     color  string
31 }
32
33 func (d *Dog) run() {
34     d.Animal.run()
35     fmt.Println("Dog run+++")
36 }
37
38 func test(a Runner) { // 多态
39     a.run()
40 }
41
42 func main() {
43     d := new(Dog)
44     d.name = "snoopy"
45     test(d)
46     c := new(Cat)
47     c.name = "Garfield"
48     test(c)
49 }
```

test使用同一个类型的同一个接口却表现不同，这就是多态。

结构体排序

我们在第二章讲过对sort包的基本用法。如果要对结构体实例排序，该怎么办呢？

```
1 // An implementation of Interface can be sorted by the routines in this
  package.
2 // The methods refer to elements of the underlying collection by integer
  index.
3 type Interface interface {
4     // Len is the number of elements in the collection.
5     Len() int
6
7     // Less reports whether the element with index i
8     // must sort before the element with index j.
9     //
10    // If both Less(i, j) and Less(j, i) are false,
11    // then the elements at index i and j are considered equal.
12    // Sort may place equal elements in any order in the final result,
13    // while Stable preserves the original input order of equal elements.
14    //
15    // Less must describe a transitive ordering:
16    // - if both Less(i, j) and Less(j, k) are true, then Less(i, k) must
  be true as well.
17    // - if both Less(i, j) and Less(j, k) are false, then Less(i, k) must
  be false as well.
18    //
19    // Note that floating-point comparison (the < operator on float32 or
  float64 values)
20    // is not a transitive ordering when not-a-number (NaN) values are
  involved.
21    // See Float64Slice.Less for a correct implementation for floating-point
  values.
22    Less(i, j int) bool
23
24    // Swap swaps the elements with indexes i and j.
25    Swap(i, j int)
26 }
```

从接口定义来看，要实现某类型的排序

- 要知道有多少个元素
- 2个指定索引的元素怎么比较大小，索引i的元素小于索引j的值返回true，反之返回false
- 如何交换指定索引上的元素

那么自定义类型，要想排序，就要实现该接口。

假设有N个学生，学生有姓名和年龄，按照**年龄**排序结构体实例。

学生使用结构体Student，多个学生就使用切片[]Student。

参照 `sort.Ints()` 的实现。

安装了Vscode插件后，可以键入sort后按tab，直接出现接口实现代码模板。

```

1 func Ints(x []int) { Sort(IntSlice(x)) } 观察这个方法，它依赖下面的定义
2
3 // IntSlice attaches the methods of Interface to []int, sorting in increasing
  order.
4 type IntSlice []int
5
6 func (x IntSlice) Len() int           { return len(x) }
7 func (x IntSlice) Less(i, j int) bool { return x[i] < x[j] }
8 func (x IntSlice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }

```

就是要在[]Student上实现Interface接口的Len、Less、Swap方法。为了方便可以定义一个新类型，好实现方法。

```

1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sort"
7     "time"
8
9     "strconv"
10 )
11
12 type Student struct {
13     Name string
14     Age  int
15 }
16
17 type Studentslice []Student
18
19 func (x Studentslice) Len() int           { return len(x) }
20 func (x Studentslice) Less(i, j int) bool { return x[i].Age < x[j].Age }
21 func (x Studentslice) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
22
23 func main() {
24     // 随机生成学生数据
25     r := rand.New(rand.NewSource(time.Now().UnixNano()))
26     students := make([]Student, 0, 5)
27     for i := 0; i < 5; i++ {
28         name := "Tom" + strconv.Itoa(i)
29         age := r.Intn(30) + 20
30         students = append(students, Student{name, age})
31     }
32     fmt.Printf("%+v, %[1]T\n", students)
33     fmt.Println("~~~~~")
34     sort.Sort(Studentslice(students))
35     // 强制类型转化为Studentslice后就可以应用接口方法排序了
36     fmt.Printf("%+v, %[1]T\n", students)
37 }

```

切片排序简化

上例中，对于切片来说，Len、Swap实现其实都这么写，切片中元素排序，就是某种类型的元素之间如何比较大小不知道，能否只提出这一部分的逻辑单独提供？从而简化切片的排序。这就要靠

`sort.Slice(待排序切片, less函数)`了。

```
1 package main
2
3 import (
4     "fmt"
5     "math/rand"
6     "sort"
7     "time"
8
9     "strconv"
10 )
11
12 type Student struct {
13     Name string
14     Age  int
15 }
16
17 func main() {
18     // 随机生成学生数据
19     r := rand.New(rand.NewSource(time.Now().UnixNano()))
20     students := make([]Student, 0, 5)
21     for i := 0; i < 5; i++ {
22         name := "Tom" + strconv.Itoa(i)
23         age := r.Intn(30) + 20
24         students = append(students, Student{name, age})
25     }
26     fmt.Printf("%+v, %[1]T\n", students)
27     fmt.Println("~~~~~")
28
29     sort.Slice(students, func(i, j int) bool {
30         return students[i].Age < students[j].Age
31     })
32     fmt.Printf("%+v, %[1]T\n", students)
33 }
```

map的排序

map是键值对的集合，是无序的hash表。但是排序输出是序列，也就是排序所需的键或值要存入序列中，然后才能排序。

key排序

思路：提取key为序列，排序后，用有序序列中的key映射value输出

```
1 package main
2
3 import (
4     "fmt"
5     "sort"
```

```

6  )
7
8  func main() {
9      // To create a map as input
10     m := make(map[int]string)
11     m[1] = "a"
12     m[2] = "c"
13     m[0] = "b"
14
15     // To store the keys in slice in sorted order
16     var keys []int
17     for k := range m {
18         keys = append(keys, k)
19     }
20     sort.Ints(keys)
21
22     // To perform the operation you want
23     for _, k := range keys {
24         fmt.Println("key:", k, "value:", m[k])
25     }
26 }

```

value排序

不能使用key排序思路，想象每一个键值对就是一个{key:xxx, value:yyy}的结构体实例，就转换成了结构体序列排序了。

```

1  package main
2
3  import (
4      "fmt"
5      "sort"
6  )
7
8  type Entry struct {
9      Key   int
10     Value string
11 }
12
13 func main() {
14     m := make(map[int]string)
15     m[1] = "a"
16     m[2] = "c"
17     m[0] = "b"
18
19     p := make([]Entry, len(m))
20     i := 0
21     for k, v := range m {
22         p[i] = Entry{k, v}
23         i++
24     }
25     fmt.Println(p)
26 }

```

```
27     sort.Slice(p, func(i, j int) bool {
28         return p[i].value < p[j].value
29     })
30     fmt.Println(p)
31 }
```

泛型

Go 1.18, 加入泛型。泛即通用的。

泛型函数

没有泛型，同一种类型需要用重载overload完成，虽然Go没有重载，但是我们可以定义出来如下形式

```
1 func addInt(a, b int) int {
2     return a + b
3 }
4
5 func addFloat(a, b float64) float64 {
6     return a + b
7 }
8
9 func addString(a, b string) string {
10    return a + b
11 }
```

可以看出，代码形式上大量重复。如果形参的类型也能替代就好了？

由此，我们希望得到下面的形式

```
1 // 伪代码如下
2 func add(a, b T) T {
3     return a + b
4 }
```

T被称为 类型形参（type parameter），也就是说参数的类型也可以变，`a, b T`说明a、b要类型一致。

T最终一定会确定是某种类型，例如是int。

Go语言中，使用泛型的类型形参就可以解决上面的问题


```

1 package main
2
3 import "fmt"
4
5 func add[T int | float64 | string](a, b T) T { return a + b }
6
7 func main() {
8     fmt.Println(add(4, 5), add[int](4, 5))
9     fmt.Println(add(4.1, 5.2))
10    fmt.Println(add("abc", "xyz"))
11 }

```

可以看到，大量冗余代码被简化，代码可读性也提高了。

- **T类型形参** (type parameter)，只是一个类型的占位符
- `int | float64 | string`称为**类型约束** (type constraint)，`|` 表示或
- `T int | float64 | string`称为**类型参数列表**，目前只有一个类型形参T
 - `[T int | string, P any]`，多个类型参数使用逗号分隔
- `add[T]`就是新定义的**泛型函数**
- `add[int]`中`int`就是**类型实参**，传入`int`给泛型函数的过程称为**实例化**
 - `add[int](4, 5)`可以写作`add(4, 5)`，因为可以根据函数的实参推断出来
- 可以看到上面是在函数名后面跟着类型参数列表，所以，**匿名函数不可以定义成泛型函数**，但可以使用定义好的类型形参T

类型约束

类型约束是一个接口。为了支持泛型，Go 1.18对接口语法进行了扩展。

用在泛型中，接口含义是符合这些特征的类型的集合。

Go内置了2个约束

- `any` 表示任意类型
- `comparable` 表示类型的值应该可以使用`==`和`!=`比较

```

1 [T int] 等价于 [T interface{int}]，表示T只能是int类型
2
3 type Constraint1 interface {
4     int|string
5 }
6 [T int|string]、[T interface{int|string}]、[T Constraint1]三者等价，表示类型只能是int或string类型

```

泛型类型

```

1 package main
2
3 import "fmt"
4
5 type Runner interface {
6     run() // 注意，这里不应该看做普通接口，而应该看做约束，要求这一类都要实现这个方法
7 }

```

```
8
9 // 表示该map的key被约束为int或string类型, value被约束为实现Runner接口的类型
10 type MyMap[K string | int, V Runner] map[K]V
11
12 type MyString string
13
14 func (ms MyString) run() {
15     fmt.Println("run", ms)
16 }
17
18 func main() {
19     // var d MyMap[int, MyString] = make(MyMap[int, MyString])
20     var d = make(MyMap[int, MyString]) // 相当于于map[int]MyString{}
21     fmt.Printf("%T, %v\n", d, d)
22     d[100] = "abc"
23     fmt.Println(d)
24     fmt.Println("~~~~~")
25     d[100].run()
26 }
```

