设计模式-策略模式

参考

- Go设计模式(20)-策略模式*程序员麻辣烫的博客-CSDN博客*go 策略模式
- https://www.runoob.com/design-pattern/strategy-pattern.html

---

在策略模式中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的context对象。策略对象改变context对象的执行算法。

使用场景

- 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地在许多行为汇总选择一种行为
- 一个系统需要动态地在几种算法中选择一种

Demo分析

> 分析：
>
> 面对一个计算跨境商品税费的需求，我们需要针对是否含税以及不同类型商品税率计算不同，来设计一个税费计算的接口。

Go实现

```
package main

import "fmt"

const (
    Common = "Common"
    Win    = "Win"
)

/**
* 获取商品类型
 */
func getProductType(hscode string) string {
    if hscode == "11" {
```

```
        return Common
    } else {
        return Win
    }
}

/**
* 税费计算函数
 */
type TaxComputeFunc func(price int64, qty int64) (taxPrice int64)

var TaxComputeFunMap = map[int]map[string]TaxComputeFunc{
    0: map[string]TaxComputeFunc{
        Common: common,
        Win:    win,
    },
    1: map[string]TaxComputeFunc{
        Common: common,
        Win:    win,
    },
}

/**
* 计算普通商品税费
 */
func common(price int64, qty int64) (taxPrice int64) {
    radio := 0.1
    fmt.Printf("计算普通商品税费")
    taxPrice = int64(float64(price*qty) * radio)
    return taxPrice
}

/**
* 计算窗口商品税费
 */
func win(price int64, qty int64) (taxPrice int64) {
    radio := 0.2
    fmt.Printf("计算窗口商品税费")
    taxPrice = int64(float64(price*qty) * radio)
    return taxPrice
}

/**
*计算税费
*@param price 商品价格
 */
func ComputeTaxPrice(withTax int, productType string, price int64, qty int64) {
    if taxFunc, ok := TaxComputeFunMap[withTax][productType]; ok {
        taxPrice := taxFunc(price, qty)
        fmt.Printf("税费为:%d", taxPrice)
    } else {
        fmt.Printf("没有找到税费计算函数")
    }
}
```

```
func test() {
    withTax := 0
    var price, qty int64 = 1000, 3
    productType := getProductType("11")
    ComputeTaxPrice(withTax, productType, price, qty)
}

func main() {
    test()
}
```

输出

```
计算窗口商品税费税费为:600
```

Python实现

```
class ProductType:
  Win = "Win"
  Common = "Common"


def getProductType(hscode):
  if hscode == "11":
    return ProductType.Common
  else:
    return ProductType.Win

class TaxCompute(object):
  """
    ## 包含各种税费的计算方法
  """
  @classmethod
  def common(self, price, qty):
    radio = 0.2
    return price * qty * radio
  @classmethod
  def win(self, price, qty):
    radio = 0.05
    return price * qty * radio

# 计算税费的查表
TaxComputeFuncMap = {
  0: {
    ProductType.Common: TaxCompute.common,
    ProductType.Win: TaxCompute.win
  },
  1: {
    ProductType.Common: TaxCompute.common,
    ProductType.Win: TaxCompute.win
  }
}

def ComputeTaxPrice(withTax, productType, price, qty):
```

```
    """
      ## 计算税费
    """
    taxFunc = TaxComputeFuncMap.get(withTax, {}).get(productType, None)
    if taxFunc is None:
      raise Exception("Invalid tax type")
    return taxFunc(price, qty)

def test():
    withTax = 0
    price, qty = 2000, 2
    productType = getProductType("111")
    print(ComputeTaxPrice(withTax, productType, price, qty))

if __name__ == '__main__':
    test()
```

输出

```
200.0
```

小结

策略模式其实采用了一种查表的方式，针对多分支的目的场景。采用策略模式帮助我们避免大量使用if-else来实现分支判断，更容易让我们理清代码的结构。