设计模式-建造者模式

参考

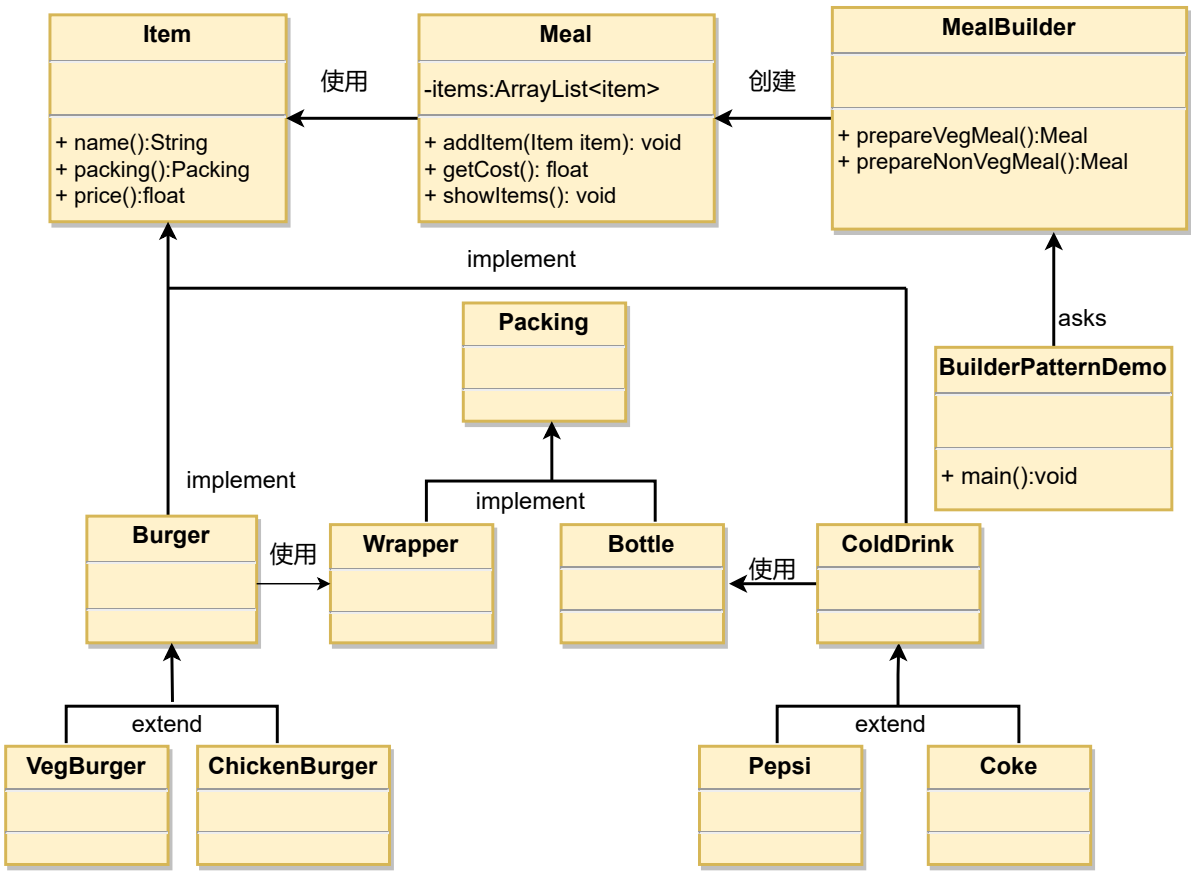
- ()[]

建造者模式使用多个简单的对象一步一步构建成一个复杂对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

使用场景

- 需要生成的对象具有复杂的内部结构
- 需要生成的对象内部属性本身相互依赖

Demo分析



分析：

> 上述案例我们需要完成一个快餐店的商业案例，其中我们完成两种套餐 `VegMeal` 和 `NonVegMeal` 的组成信息。而每个套餐中包含一个汉堡和一杯冷饮，且汉堡和冷饮都有两种类型。此时我们可以分析不同汉堡间有共性，不同的冷饮之间有共性同时每一种成品之间也有相同的特性。因此，我们按照上述的UML图进行抽象。

Go实现

food.go

> 对各种食物进行抽象

```
package main

type Burger struct {
}

func (c *Burger) Pack() string {
    packing := Wrapper{}
    return packing.Pack()
}

type VegBurger struct {
    Burger
}

func (v *VegBurger) Price() float64 {
    return float64(3.8)
}
func (v *VegBurger) Name() string {
    return "VegBurger"
}

type ChickenBurger struct {
    Burger
}

func (c *ChickenBurger) Price() float64 {
    return float64(2.8)
}

func (c *ChickenBurger) Name() string {
    return "ChickenBurger"
}

type ColdDrink struct {
}

func (c *ColdDrink) Pack() string {
    packing := Bottle{}
    return packing.Pack()
}

type Pepsi struct {
    ColdDrink
}
```

```
func (c *Pepsi) Price() float64 {
    return float64(1.8)
}

func (c *Pepsi) Name() string {
    return "Pepsi"
}

type Coke struct {
    ColdDrink
}

func (c *Coke) Price() float64 {
    return float64(0.8)
}

func (c *Coke) Name() string {
    return "Coke"
}
```

Item.go

> 对商品和套餐进行抽象

```
package main

import "fmt"

type Item interface {
    Price() float64
    Name() string
    Pack() string
}

type Meal struct {
    Items []Item
}

func (m *Meal) addItem(item Item) {
    m.Items = append(m.Items, item)
}

func (m *Meal) getCost() (sum float64) {
    for _, item := range m.Items {
        sum += item.Price()
    }
    return sum
}

func (m *Meal) showItems() {
    fmt.Println("Items:")
    for _, item := range m.Items {
        fmt.Printf("Item: %s, Price: %.2f, Pack:%s\n", item.Name(),
item.Price(), item.Pack())
```

```
    }
}
```

pack.go

> 对两种打包类型进行抽象

```
package main

type Packing interface {
    Pack() string
}

type Wrapper struct {
}

func (w *Wrapper) Pack() string {
    return "Wrapper Pck"
}

type Bottle struct {
}

func (w *Bottle) Pack() string {
    return "Bottle Pck"
}
```

main.go

```
package main

import "fmt"

/**
* 建造者
 */
type MealBuilder struct{}

func (builder *MealBuilder) PrepareVegMeal() Meal {
    meal := Meal{}
    meal.addItem(&VegBurger{})
    meal.addItem(&Coke{})
    return meal
}

func (builder *MealBuilder) prepareNonVegMeal() Meal {
    meal := Meal{}
    meal.addItem(&ChickenBurger{})
    meal.addItem(&Pepsi{})
    return meal
}

func test() {
    builder := MealBuilder{}
    vegMeal := builder.PrepareVegMeal()
```

```
    NoeVegMeal := builder.prepareNonVegMeal()
    vegMeal.showItems()
    fmt.Printf("Cost of the veg meal is: %.2f\n", vegMeal.getCost())
    NoeVegMeal.showItems()
    fmt.Printf("Cost of the Non-Veg meal is: %.2f\n", NoeVegMeal.getCost())
}

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

输出

```
Items:
Item: VegBurger, Price: 3.80, Pack:Wrapper Pck
Item: Coke, Price: 0.80, Pack:Bottle Pck
Cost of the veg meal is: 4.60
Items:
Item: ChickenBurger, Price: 2.80, Pack:Wrapper Pck
Item: Pepsi, Price: 1.80, Pack:Bottle Pck
Cost of the Non-Veg meal is: 4.60
```

Python实现

food.py

```
from pack import Wrapper, Bottle
from Item import Item

class Burger(Item):
  def Pack(self):
    return Wrapper().Pack()

class VegBurger(Burger):
  def __init__(self):
    self.name = "Veg Burger"
    self.price = 1.5

class ChickenBurger(Burger):
  def __init__(self):
    self.name = "Chicken Burger"
    self.price = 2.5

class ClodDrink(Item):
  def Pack(self):
    return Bottle().Pack()

class Coke(ClodDrink):
  def __init__(self):
    self.name = "Coke"
    self.price = 0.9

class Pepsi(ClodDrink):
  def __init__(self):
```

```
    self.name = "Pepsi"
    self.price = 0.4
```

Item.py

```
class Item(object):
  def Price(self):
    return self.price
  def Name(self):
    return self.name
  def Pack(self):
    self.Pack()

class Meal():
  def __init__(self):
    self.items = []
  def addItem(self, item):
    self.items.append(item)
  def getCost(self):
    cost = 0.0
    for item in self.items:
      cost += item.Price()
    return cost
  def showItems(self):
    print(f"Items:")
    for item in self.items:
      print(f"Name::{item.Name()}, Price::{item.Price()}")
```

pack.py

```
class Packing(object):
  def Pack(self):
    pass

class Wrapper(Packing):
  def Pack(self):
    return "Pack :: Wrapper"

class Bottle(Packing):
  def Pack(self):
    return "Pack :: Bottle"
```

main.py

```
from Item import Meal
from food import VegBurger, Coke, ChickenBurger, Pepsi

class MealBuilder():
  def prepareVegMeal(self):
    meal = Meal()
    meal.addItem(VegBurger())
    meal.addItem(Coke())
    return meal
  def prepareNonVegMeal(self):
```

```
    meal = Meal()
    meal.addItem(ChickenBurger())
    meal.addItem(Pepsi())
    return meal

def test():
  mealBuilder = MealBuilder()
  vegMeal = mealBuilder.prepareVegMeal()
  nonVegMeal = mealBuilder.prepareNonVegMeal()
  vegMeal.showItems()
  print(f"Cost of vegMeal is {vegMeal.getCost()}")
  nonVegMeal.showItems()
  print(f"Cost of nonVegMeal is {nonVegMeal.getCost()}")

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

输出

```
Items:
Name::Veg Burger, Price::1.5
Name::Coke, Price::0.9
Cost of vegMeal is 2.4
Items:
Name::Chicken Burger, Price::2.5
Name::Pepsi, Price::0.4
Cost of nonVegMeal is 2.9
```

小结

> 对比上述的两种不同语言的写法，Go语言的写法相较Python这种类式的写法更加灵活避免了深层的继承，更加灵活但是看起来更加复杂。