设计模式-观察者模式

参考

- Go设计模式(18)-观察者模式*程序员麻辣烫的博客-CSDN博客*go 观察者模式
- https://www.runoob.com/design-pattern/observer-pattern.html

---

当对象间存在一对多关系时，则使用观察者模式。比如，当一个对象被修改时，则会自动通知依赖它的对象。观察者模式属于行为型模式。

使用场景

- 一个对象必须通知其他对象，而并不知道这些对象是谁
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体由多少个对象将发生变化，可以减低对象之间的耦合度。

Demo分析

商家购买服务的订单，有多个状态，如交易成功、交易取消、开始履行等。如果不使用观察者模式，可以用if-else解决这个问题。但各个状态的处理逻辑比较复杂，而且状态有可能存在乱序的情况，导致要处理的case更多了。这个时候使用观察者模式就比较合适，订单状变更我iSubject，具体处理逻辑为Observer，状态变更时，通知所有处理逻辑，谁适合处理由谁处理。

Go实现

```
package main

import "fmt"

type PurchaseOperFunc func(status string, data string) (res bool, err error)

var PurchaseOperFuncArr = []PurchaseOperFunc{
    create,
    isDeleted,
    apply,
}

/**
* 用于创建的观察者
 */
func create(status string, data string) (res bool, err error) {
```

```
    if status == "create" {
        fmt.Printf("create %s\n", data)
        return true, nil
    }
    return false, nil
}

/**
* 用于删除的观察者
 */
func isDeleted(status string, data string) (res bool, err error) {
    if status == "delete" {
        fmt.Printf("delete %s\n", data)
        return true, nil
    }
    return false, nil
}

/**
* 履行的观察者
 */
func apply(status string, data string) (res bool, err error) {
    if status == "apply" {
        fmt.Printf("apply %s\n", data)
        return true, nil
    }
    return false, nil
}

func test() {
    status := "create"
    data := "test"
    for _, v := range PurchaseOperFuncArr {
        res, err := v(status, data)
        if err != nil {
            fmt.Printf("err: %s\n", err)
        }
        if res {
            fmt.Printf("success\n")
        }
    }
}

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

输出

```
create test
success
```

Python实现

```python
class Observer(object):
  def __init__(self, task):
    task.add_observer(self)
  def update(self):
    pass

class Create(Observer):
  """
    ## 订单创建观察者
  """
  def __init__(self,task):
    super().__init__(task)
  def update(self, state):
    if state == 'create':
      print('create')

class isDelete(Observer):
  """
    ## 订单删除观察者
  """
  def __init__(self,task):
    super().__init__(task)
  def update(self, state):
    if state == 'delete':
      print('delete')

class apply(Observer):
  """
    ## 订单履行观察者
  """
  def __init__(self,task):
    super().__init__(task)
  def update(self, state):
    if state == 'apply':
      print('apply')

class Task(object):
  """
    ## 订单任务
  """
  def __init__(self, state):
    self.state = state
    self.observers = []
  def setState(self, state):
    self.state = state
    self.notify()
  def add_observer(self, observer):
    self.observers.append(observer)
  def notify(self):
    for observer in self.observers:
      observer.update(self.state)

def test():
  task = Task("create")
  Create(task)
```

```
  isDelete(task)
  apply(task)
  task.setState("apply")
  task.setState("delete")

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

输出

```
apply
delete
```

小结

观察者模式跟订阅发布类似，但它将不同的观察者都独立出去，实现了对观察者和被观察者的解耦。