观察者模式

引言

概念

观察者模式实际上是一种发布-订阅模式，当一个对象的状态发生改变时，所有依赖的对象都会收到通知。例如当用户注册成功后，下发注册成功短信，以及发送优惠券等

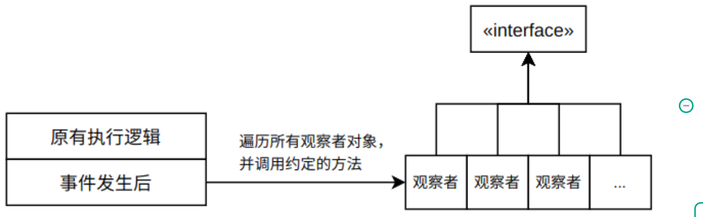
从基本概念中就可以看出，观察者模式可以说是在系统设计中使用最为广泛的设计模式之一

- Linux层面
  - 监控文件事件: inotify API
  - 父进程中的wait()与waitpid()
  - POSIX线程中的通知状态的改变: 条件变量
- Web框架层面
  - Django Model Signals
- 网络层面
  - Redis发布-订阅
  - RabbitMQ的消息发送

观察者模式的实现多种多样，在不同的层面上会有完全不同的实现方法。可以是同步阻塞的，也可以是异步非阻塞的; 可以是进程内的实现，也可以是跨进程的实现

基本实现

实现一个最小原理的观察者模式比较的简单，本质上就是在被观察对象发生某些行为时，调用相关的方法，通知所有的观察者



强类型语言(Java、Go)在实现时，需保证所有的观察者对象为同一个类型(均实现了某一个接口)

弱类型语言(Python)则无此限制，并且由于存在*args和**kwargs等潘多拉魔盒的存在，使得可以随意地注册方法

Go最简实现

```go
type Observer interface {
    Notify(message string)
}

type Subject interface {
    RegisterObservers(observer Observer) // 注册
    RemoveObservers(observer Observer)   // 移除
    NotifyAll(message string)            // 通知所有的观察者
}

type ConcreteSubject struct {
    observers []Observer
}

func (c *ConcreteSubject) RegisterObservers(observer Observer) {
    c.observers = append(c.observers, observer)
}

func (c *ConcreteSubject) RemoveObservers(observer Observer) {/*...*/}

func (c *ConcreteSubject) NotifyAll(message string) {
    for _, observer := range c.observers {
        observer.Notify(message)
    }
}
```

这种模式是一种很模板化的观察者模式，在实际的业务应用中，通常会据此进行相应的调整

但本质不会发生变化: 事件发生时调用相应的方法，但是利用抽象和解耦的方式将业务逻辑和相关调用分离

观察者模式的实际应用

分类

- 进程内的异步非阻塞实现
  - 当事件发生时所需要通知的观察者对象过多，或者是某一个观察者执行自身逻辑需要花费大量时间时，可以考虑使用异步非阻塞的方式实现
  - 实现方式也比较简单，利用线程池或者是协程池将任务提交至队列，原有业务逻辑直接返回
- 跨进程的观察者模式实现
  - Linux内核尽管提供了进程间的通信工具，如管道、信号量以及消息队列，但是使用较为复杂，且无法跨节点通信
  - 这时候消息队列(RabbitMQ、Kafka)会是个更好的选择，但由于存在网络数据的发送与接收，此时向消息队列推送数据必须以异步的方式进行

通知事件的时机

以个人经验来看，采用何种消息发送的形式(同步阻塞调用、异步非阻塞调用)通常不是观察者模式的复杂点

通常来说，消息队列，线程池，或者基于Redis、RabbitMQ的异步任务框架等方式的实现，在项目初期就已经实现完毕，可以直接使用

使用观察者模式必然会侵入到原有业务代码的执行流程，可能是某个Model的成功入库，或者是用户成功注册，或是用户订单支付成功，等等，事件通知的时机错综复杂

微服务下的观察者模式

上面有提到通知事件的时机因业务需求不同可能散布在项目的各个层次之中，而在微服务系统设计下，这一问题棘手程度会部分的降低

在微服务架构下，一个服务只负责某一个具体的领域，例如用户管理，订单管理，商品管理等服务

事件发生的种类比较单一，那么开发者就可以为某一个常用事件(如订单已支付)添加一个消息总线，在全局模式下解决