观察者模式

背景

在一个系统中，难免会有一些类之间存在相互协作的关系，如果我们单独地去维护这些对象之间的一致性，各个类之间就可能会紧密耦合，为了避免这样的问题产生而降低这些类的可重用性，就需要实现观察者模式。

简介

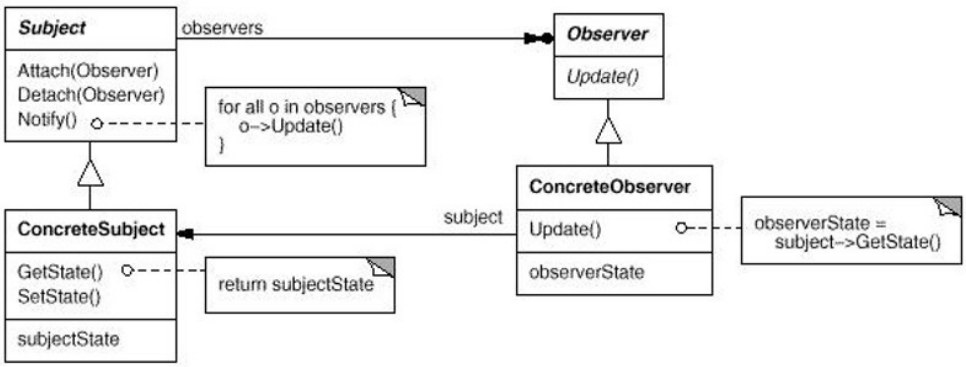
观察者模式指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式。

意图

- 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新

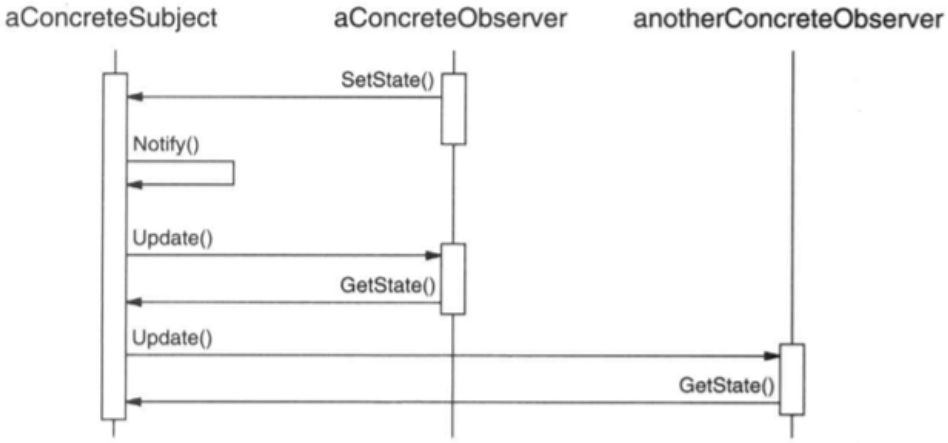
结构

- 抽象目标(**Subject**)：
  - 它提供了一个用于保存观察者对象的聚集类和注册、删除观察者对象的方法，以及通知所有观察者的抽象方法
  - 知道它的观察者，同一目标的观察者可以是任意多的
- 抽象观察者(**Observer**)：
  - 它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用
- 具体目标(**Concrete Subject**)：
  - 存储有关状态
  - 它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象
- 具体观察者(**Concrete Observer**)：
  - 维护一个指向ConcreteSubject对象的引用。
  - 存储有关状态，与具体目标中的状态一致。
  - 实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态



协作过程

- 当具体目标对象发生任何可能导致其观察者与本身状态不一致的变化时，观察者将被通知
- 在得到一个具体目标的改变通知后,具体目标对象可向目标对象查询信息。具体目标对象使用这些信息以使它的状态与目标对象的状态一致。



- 注意发出改变请求的Observer对象并不立即更新,而是将其推迟到它从目标得到一个通知之后。

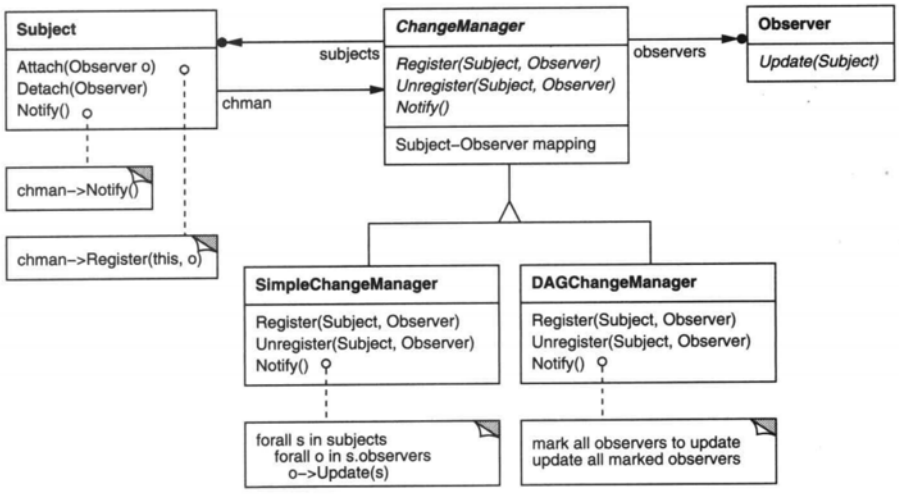
实际使用场景举例

- 微信公众号，只有订阅的用户才会收到内容更新推送
- 线上拍卖竞价，最高出价的更新会被每一个竞价者收到。
- 气象数据更新，手机app中的天气数据会随着相应的气象站数据更新而更新。

实现

- 观察多个对象
  - 在某些情况下,一个观察者依赖于多个目标可能是有意义的。例如,一个表格对象可能依赖于多个数据源。
  - 在这种情况下,必须扩展Update接口以使观察者知道是哪一个目标送来的通知。
- 创建目标到其观察者之间的映射
  - 一个目标对象跟踪它应通知的观察者的最简单的方法是显式地在目标中保存对它们的引用。然而,当目标很多而观察者较少时,这样存储可能代价太高。
  - 一个解决办法是用时间换空间,用一个关联查找机制(例如一个hash表)来维护目标到观察者的映射。这样增加了访问观察者的开销。
- 谁来触发更新
  - 由目标对象的状态设定操作在改变目标对象的状态后自动调用Notify。
  - 让客户负责在适当的时候调用Notify。
    - 优点是避免不必要的中间更新。
    - 缺点是给客户增加了触发更新的责任。客户可能会忘记调用Notify。
- 已删除目标的悬挂引用
  - 删除一个目标时应注意不要在其观察者中遗留对该目标的悬挂引用。

    - 一种避免悬挂引用的方法是,当一个目标被删除时，让它通知它的观察者将对该目标的引用复位。
- 避免特定于观察者的更新协议
    - 不同的观察者可能需要不同的更新信息，信息的量可能很小，也可能很大。
        - 推模型: 目标向观察者发送关于改变的详细信息,而不管它们需要与否。
            - 假定目标知道一些观察者的需要的信息
            - 可能使得观察者相对难以复用，因为目标对观察者的假定可能并不总是正确的。
        - 拉模型: 目标除最小通知外什么也不送出,而在此之后由观察者显式地向目标询问细节。
            - 目标不知道它的观察者。
            - 可能效率较差,因为观察者对象需在没有目标对象帮助的情况下确定什么改变了。
    - 显式地指定感兴趣的改变
        - 可以扩展目标的注册接口,让各观察者注册为仅对特定事件感兴趣，以提高更新的效率。
    - 封装复杂的更新语义
        - 目标和观察者间的依赖关系可能特别复杂。
            - 当一个操作改变了多个互相独立的目标时，需要保证对应的观察者在所有目标都修改完后再收到通知，以防多次通知观察者。
            - 可能需要一个维护这些关系的对象。
                - ChangeManager
                    - 它将一个目标映射到它的观察者并提供一个接口来维护这个映射。这就不需要由目标来维护对其观察者的引用,反之亦然。
                    - 它定义一个特定的更新策略
                    - 根据一个目标的请求,它更新所有依赖于这个目标的观察者

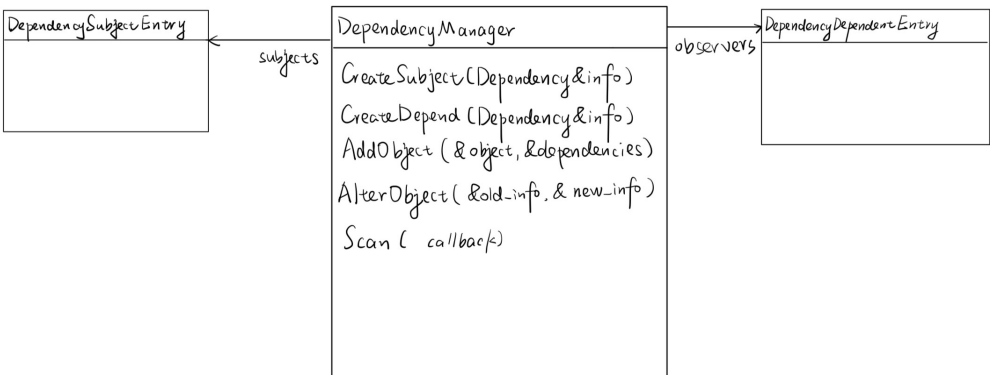


结合开源项目分析

- Subject：**Subject**
- Observer：**Dependent**

- ChangeManager: **DependencyManager**

```
// The dependents of the dependency (target)
DependencyCatalogSet dependents(Dependents(), subject.entry);
// The subjects of the dependencies of the dependent
DependencyCatalogSet subjects(Subjects(), dependent.entry);
```



优缺点分析

优点

- 观察者和被观察者是抽象耦合的
  - 目标仅知道它有一系列观察者,每个都符合抽象的Observer类的简单接口。目标不知道任何一个观察者属于哪一个具体的类。这样目标和观察者之间的耦合是抽象的和最小的。
  - 因为目标和观察者不是紧密耦合的,它们可以属于一个系统中的不同抽象层次。较低层次的目标对象可与较高层次的观察者通信并通知它,保持了系统层次的完整。
- 支持广播通信
  - 不像通常的请求,目标发送的通知不需指定它的接收者。通知被自动广播给所有已向该目标对象登记的有关对象。
  - 目标对象并不关心到底有多少对象对自己感兴趣;它唯一的责任就是通知它的各观察者。
  - 任何时刻都可以增加和删除观察者。处理还是忽略一个通知取决于观察者。

缺点

- 意外之外的更新
  - 因为观察者之间并不知道彼此的存在，所以可能对改变目标的代价一无所知
  - 在目标上一个看似无害的的操作可能会引起一系列对观察者以及依赖于这些观察者的那些对象的更新。
  - 此外,如果依赖准则的定义或维护不当，常常会引起错误的更新,这种错误通常很难捕捉。
- 对于一个对象和多个观察者的模型，如果顺序通知观察者，一个观察者卡住就会导致整个流程卡住，这就是同步阻塞。实际开发中一般使用异步。

参考资料

- 《面向对象程序设计*设计模式*研讨参考资料》——中国科学院大学2024秋季面向对象程序设计课程
- 《Head First设计模式》