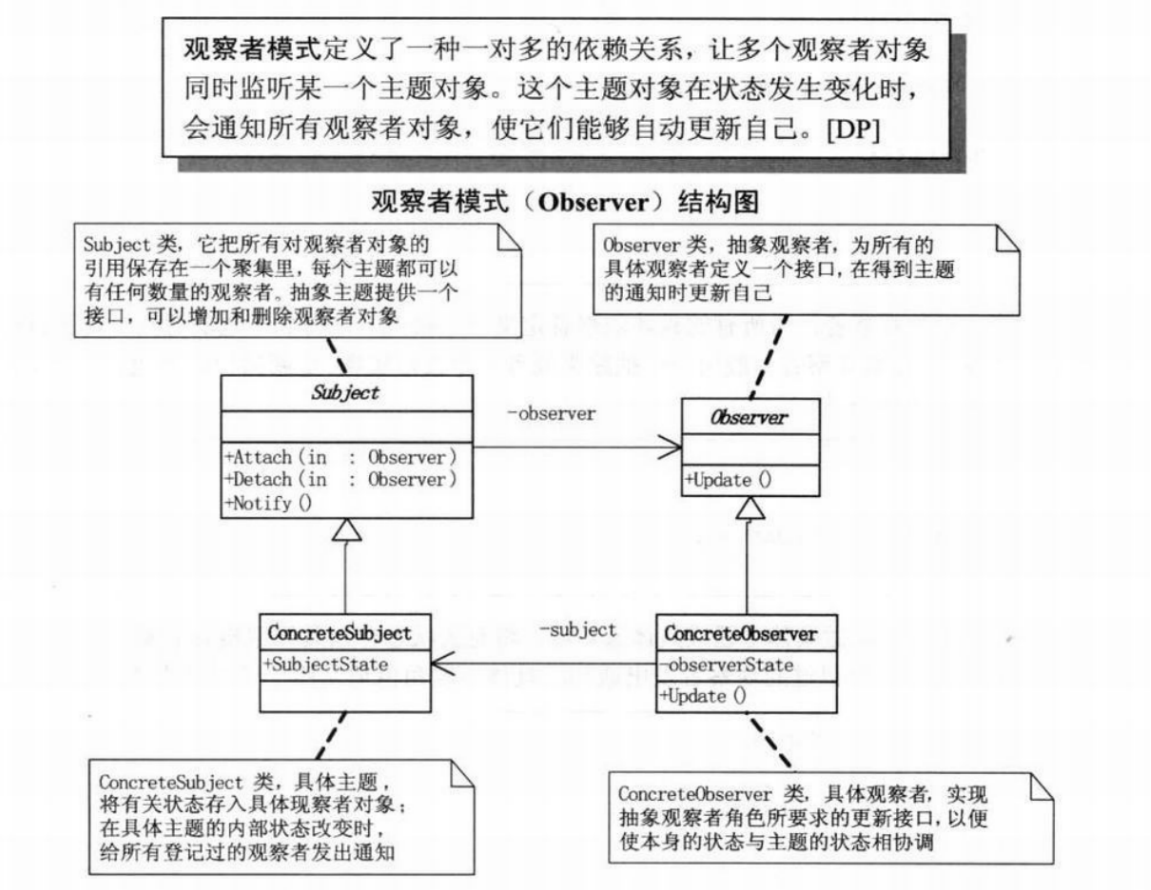
一、什么是观察者模式

在许多设计中，经常涉及多个对象都对一个特殊对象中的数据变化感兴趣，而且这多个对象都希望跟踪那个特殊对象中的数据变化，也就是说当对象间存在一对多关系时，在这样的情况下就可以使用观察者模式。当一个对象被修改时，则会自动通知它的依赖对象。

观察者模式是关于多个对象想知道一个对象中数据变化情况的一种成熟的模式。观察者模式中有一个称作“主题”的对象和若干个称作“观察者”的对象，“主题”和“观察者”间是一种一对多的依赖关系，当“主题”的状态发生变化时，所有“观察者”都得到通知。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

二、观察者模式的结构



观察者模式的结构中包含四种角色：

（1）主题（Subject）：主题是一个接口，该接口规定了具体主题需要实现的方法，比如，添加、删除观察者以及通知观察者更新数据的方法。

（2）观察者（Observer）：观察者是一个接口，该接口规定了具体观察者用来更新数据的方法。

（3）**具体主题（ConcreteSubject）**：具体主题是实现主题接口类的一个实例，该实例包含有可以经常发生变化的数据。具体主题需使用一个集合，比如ArrayList，存放观察者的引用，以便数据变化时通知具体观察者。

（4）**具体观察者（ConcreteObserver）**：具体观察者是实现观察者接口类的一个实例。具体观察者包含有可以存放具体主题引用的主题接口变量，以便具体观察者让具体主题将自己的引用添加到具体主题的集合中，使自己成为它的观察者，或让这个具体主题将自己从具体主题的集合中删除，使自己不再是它的观察者。

三、观察者模式的使用场景

（1）当一个对象的数据更新时需要通知其他对象，但这个对象又不希望和被通知的那些对象形成紧耦合。

（2）当一个对象的数据更新时，这个对象需要让其他对象也各自更新自己的数据，但这个对象不知道具体有多少对象需要更新数据。

观察者模式在实际项目的应用中非常常见，比如你到 ATM 机器上取钱，多次输错密码，卡就会被 ATM吞掉，吞卡动作发生的时候，会触发哪些事件呢？第一摄像头连续快拍，第二，通知监控系统，吞卡发生；第三，初始化 ATM 机屏幕，返回最初状态，你不能因为就吞了一张卡，整个 ATM 都不能用了吧，一般前两个动作都是通过观察者模式来完成的。观察者可以实现消息的广播，**一个消息可以触发多个事件，这是观察者模式非常重要的功能。**

使用观察者模式也有两个重点问题要解决：

广播链的问题：
如果你做过数据库的触发器，你就应该知道有一个触发器链的问题，比如表 A 上写了一个触发器，内容是一个字段更新后更新表 B 的一条数据，而表 B 上也有个触发器，要更新表 C，表 C 也有触发器…,完蛋了，这个数据库基本上就毁掉了！我们的观察者模式也是一样的问题，一个观察者可以有双重身份，即使观察者，也是被观察者，这没什么问题呀，但是链一旦建立，这个逻辑就比较复杂，可维护性非常差，根据经验建议，**在一个观察者模式中最多出现一个对象**既是观察者也是被观察者，也就是说消息最多转发一次（传递两次），这还是比较好控制的；

异步处理问题：

被观察者发生动作了，观察者要做出回应，如果观察者比较多，而且处理时间比较长怎么办？那就用异步呗，异步处理就要考虑线程安全和队列的问题，这个大家有时间看看Message Queue，就会有更深的了解。

四、观察者模式的优缺点

优点：

1、具体主题和具体观察者是松耦合关系。由于主题接口仅仅依赖于观察者接口，因此具体主题只是知道它的观察者是实现观察者接口的某个类的实例，但不需要知道具体是哪个类。同样，由于观察者仅仅依赖于主题接口，因此具体观察者只是知道它依赖的主题是实现主题接口的某个类的实例，但不需要知道具体是哪个类。

2、观察者模式满足“开-闭原则”。主题接口仅仅依赖于观察者接口，这样，就可以让创建具体主题的类也仅仅是依赖于观察者接口，因此，如果增加新的实现观察者接口的类，不必修改创建具体主题的类的代码。。同样，创建具体观察者的类仅仅依赖于主题接口，如果增加新的实现主题接口的类，也不必修改创建具体观察者类的代码。

缺点：

1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

五、观察者模式的实现

Subject类---抽象主题或者抽象通知者，一般用一个抽象类或者一个接口实现。
它把所有对观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者。

```
// 主题的接口
public interface Subject {
    public void registerObserver(Observer observer);
```

```
    public void removeObserver(Observer observer);
    public void notifyObserver();
}
```

Observer类---抽象观察者，为所有具体观察者定义一个接口，在得到主题通知时更新自己。

这个接口叫做更新接口，抽象观察者一般用一个抽象类或者一个接口实现。更新接口通常包括一个Update方法，这个方法叫做更新方法。

```
// 观察者接口
public interface Observer {
    // 观察者需要有一个更新方法，供主题调用
    public void update(float temperature, float pressure, float humidity);
}
```

ConcreteSubject类---具体主题或者具体通知者，将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发送通知。

具体主题角色通常用一个具体类实现。

```
import java.util.ArrayList;

public class WeatherData implements Subject{
    private float temperature;
    private float pressure;
    private float humidity;

    private ArrayList<Observer> observers;

    public WeatherData(){
        observers = new ArrayList<Observer>();
    }

    public void dataChange() {
        notifyObserver();
    }

    public void setData(float temperature, float pressure, float humidity){
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        dataChange(); // 更新了数据及时通知观察者
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
```

```
    }

    @Override
    public void removeObserver(Observer observer) {
        if (observers.contains(observer)) {
            observers.remove(observer);
        }
    }

    @Override
    public void notifyObserver() {
        for (int i = 0; i < observers.size(); i++) {
            observers.get(i).update(this.temperature, this.pressure, this.humidity);  // 调
用每个观察者类的update方法更新他们的数据
        }
    }

    public float getTemperature() {
        return temperature;
    }

    public float getPressure() {
        return pressure;
    }

    public float getHumidity() {
        return humidity;
    }
}
```

ConcreteObserver类---具体观察者，实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。

具体观察者角色可以保存一个指向具体主题对象的引用。具体观察者角色通常用一个具体类实现。

具体观察者1

```
public class CurrentCondition implements Observer{

    // 温度，气压，湿度
    private float temperature;
    private float pressure;
    private float humidity;

    // 更新天气状况，由WeatherData来调用
    public void update(float temperature, float pressure, float humidity) {
        this.temperature = temperature;
```

```
        this.pressure = pressure;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("*** Today Temperature：" + temperature + " ***");
        System.out.println("*** Today Pressure：" + pressure + " ***");
        System.out.println("*** Today Humidity：" + humidity + " ***");
    }
}
```

具体观察者2

```
public class BaiduSite implements Observer{

    // 温度，气压，湿度
    private float temperature;
    private float pressure;
    private float humidity;

    // 更新天气状况，由WeatherData来调用
    public void update(float temperature, float pressure, float humidity) {
        this.temperature = temperature;
        this.pressure = pressure;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("*** 百度 Temperature：" + temperature + " ***");
        System.out.println("*** 百度 Pressure：" + pressure + " ***");
        System.out.println("*** 百度 Humidity：" + humidity + " ***");
    }
}
```

客户端代码

```
public class Client {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentCondition currentCondition = new CurrentCondition();
        weatherData.registerObserver(currentCondition);

        // 测试
        System.out.println("通知各个注册的观察者，更新信息");
```

```
        BaiduSite baiduSite = new BaiduSite();
        weatherData.registerObserver(baiduSite);  // 新增的观察者只需要注册到主题实现类即可，不需要改变 WeatherData 已有的代码

        weatherData.setData(10f, 100f, 30.2f);  // 更新主题数据，所有的观察者的数据都会同步更新
        // weatherData.removeObserver(currentCondition);  // 移除的观察者不会再接收数据更新
    }
}
```

运行结果

```
通知各个注册的观察者，更新信息
*** Today Temperature：10.0 ***
*** Today Pressure：100.0 ***
*** Today Humidity：30.2 ***
*** 百度 Temperature：10.0 ***
*** 百度 Pressure：100.0 ***
*** 百度 Humidity：30.2 ***
```

六、总结

实现观察者模式的时候要注意，观察者和被观察对象之间的互动关系不能体现成类之间的直接调用，否则就将使观察者和被观察对象之间紧密的耦合起来，从根本上违反面向对象的设计的原则。无论是观察者“观察”观察对象，还是被观察者将自己的改变“通知”观察者，都不应该直接调用。

另外redis里的pub/sub也可以实现观察者模式。