## 第21章 物业通知很重要，一个住户不能漏——观察者模式

## 21.1 没有收到停水通知

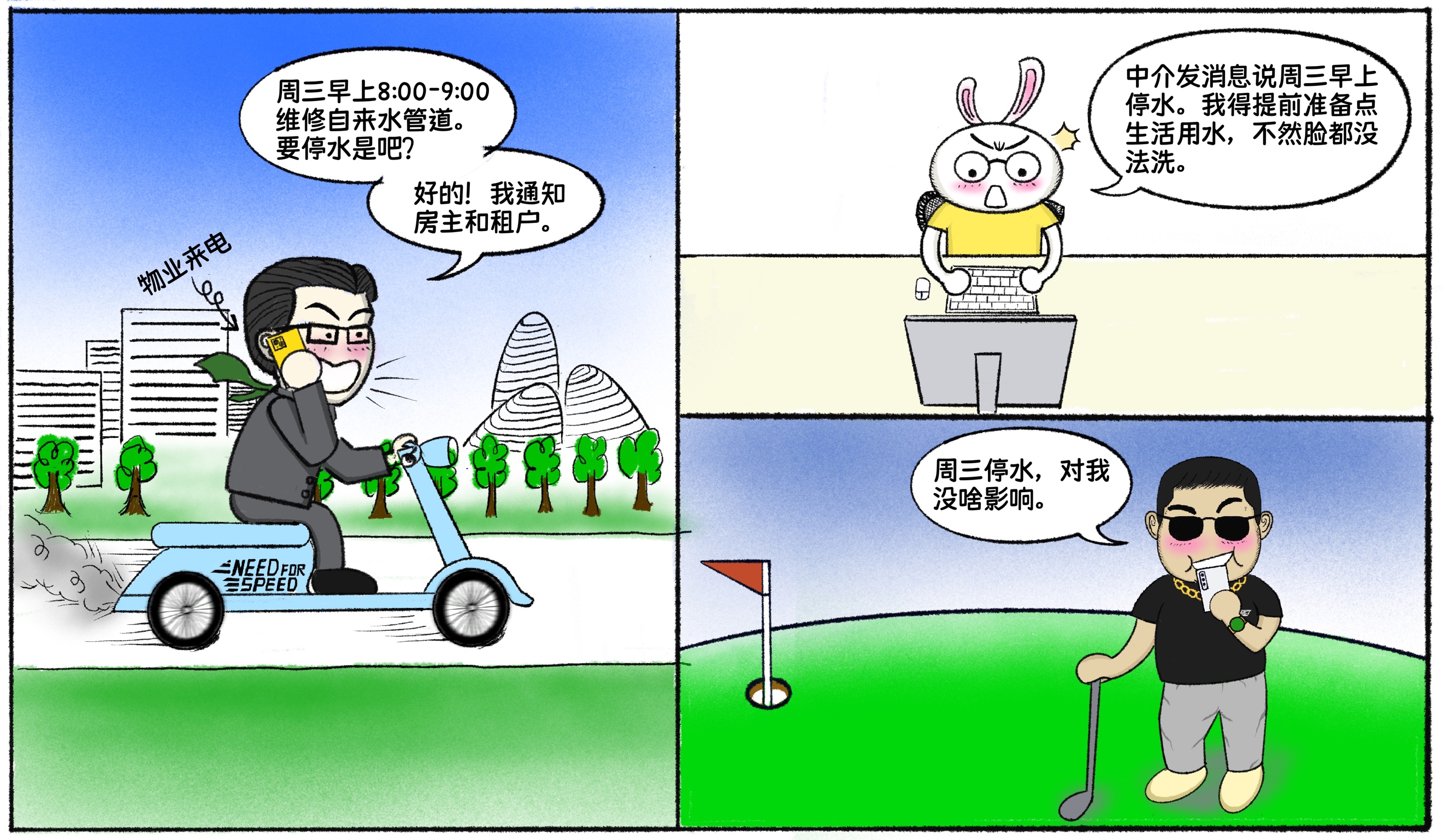
兔小白：哎......今天物业修理自来水管道，早上停水了，我的脸都没洗。

熊小猫：物业没有提前通知吗？你应该提前准备点生活用水呀！

兔小白：物业应该是给房主发了信息，但没有通知到我。我也没有留意物业贴出来的通知。

熊小猫：物业不知道你的存在，当然也不会单独给你发信息。但是，如果你租的是公寓式房子，中介会将物业通知转发给租房人和房主。咱们之前学习中介者模式的练习中，就有中介转发物业通知的场景。

兔小白：我记得这个场景。中介收到物业通知后，会调用租房人和房主的接收通知接口，将通知转发过去。



熊小猫：之前的练习只有一名房主和一名租房人。但在真实情况中，中介提供服务的房主和租房人肯定不会只有一名。当中介要通知多名房主和租房人时，如何处理消息转发就变得复杂起来了。

兔小白：你说的这个场景符合实际情况，程序确实需要优化。但是中介者模式中好像没有提到这种场景呀！

熊小猫：设计模式往往需要多个结合起来使用。比如这个场景，可以结合使用观察者模式。咱们把之前的程序找出来，看看如何使用观察者模式进行改造。

## 21.2 有必要区分房主还是租房人吗？

熊小猫：咱们先把中介者模式练习的需求做减法，只保留中介接收通知，转发给房主和租房人的需求。租房人收到停水消息后，需要提前准备生活用水。房主收到消息后不需要做任何处理。然后再对需求做加法，中介可以维护多位房主和租房人，并且确保将通知送达他所维护的每一位房主和租房人。

兔小白：听起来不难，中介类修改成维护房主和租房人的列表，其它变化不大。

10分钟后，兔小白写完了代码。

兔小白：中介类EstateMediator 维护租房人和房主两个列表。acceptNotification方法迭代租房人和房主列表，发送通知。

public class EstateMediator {  
    List<HouseOwner> houseOwners = new ArrayList<>();  
    List<Tenant> tenants = new ArrayList<>();  
​  
    public void attachHouseOwner(HouseOwner houseOwner) {  
        houseOwners.add(houseOwner);  
   }  
​  
    public void attachTenant(Tenant tenant) {  
        tenants.add(tenant);  
   }  
​  
    public void acceptNotification(String notification) {  
        for (HouseOwner houseOwner : houseOwners) {  
            houseOwner.acceptNotification(notification);  
       }  
​  
        for (Tenant tenant : tenants) {  
            tenant.acceptNotification(notification);  
       }  
   }  
}

房主类HouseOwner在收到通知后，不需要做任何处理。

public class HouseOwner {  
    public void acceptNotification(String notification) {  
        System.out.println("房主：收到维修通知-" + notification);  
   }  
}

租房人类Tenant收到通知后，需要准备生活用水。

public class Tenant {  
    public void acceptNotification(String notification) {  
        System.out.println("租房人：收到维修通知-" + notification);  
        System.out.println("租房人：我会提前准备生活用水");  
   }  
}

这版代码只是将中介类维护单一对象改成了维护列表，变化并不大。

熊小猫：功能实现没有问题，但优化空间还很大。最明显的问题是两个列表的迭代。中介发通知的行为完全不需要区分房主和租房人。房主和租房人接收通知的方法一模一样，为什么不抽象成接口呢？现在的代码中，中介类依赖具体的租房人和房主类，并不符合依赖倒置原则。

另外，Tenant和HouseOwner接收消息的acceptNotification方法不具备扩展性，只能处理中介的消息。假如房主还要接收居委会的消息，现在的程序很难处理。

兔小白：我也觉得迭代两个列表写的不够优雅。听你说完，我知道问题出在哪里了。我再想想如何解决发送者的扩展问题。

## 21.3 房主和租房人一视同仁

30分钟后，兔小白对熊小猫提出的两个问题进行了优化。

兔小白：我把接收并处理消息的行为抽象成了Observer接口。

public interface Observer {  
    void update(Subject subject);  
}

HouseOwner和Tenant类实现Observer接口，实现方式一样。以HouseOwner为例，update方法接收Subject类型参数，通过判断Subject的具体类型，选择相应的逻辑处理。

public class HouseOwner implements Observer {  
    public void update(Subject subject) {  
        if (subject instanceof EstateMediator) {  
            System.out.println("房主：收到维修通知-" +  
                   ((EstateMediator) subject).getNotification());  
       }  
   }  
}

Tenant类代码如下。

public class Tenant implements Observer {  
    public void update(Subject subject) {  
        if (subject instanceof EstateMediator) {  
            System.out.println("租房人：收到维修通知-" +  
                   ((EstateMediator) subject).getNotification());  
            System.out.println("租房人：我会提前准备生活用水");  
       }  
   }  
}

Subject封装了向Observer发送消息的行为。它维护一个Observer对象列表，提供了注册和移除Observer对象的方法。notifyObservers方法会迭代Observer对象列表，调用Observer对象的update方法，让其处理自己的业务。

public class Subject {  
    private final List<Observer> observers = new ArrayList<>();  
​  
    public void attach(Observer observer) {  
        observers.add(observer);  
   }  
​  
    public void detach(Observer observer) {  
        observers.remove(observer);  
   }  
​  
    public void notifyObservers() {  
        for (Observer observer : observers) {  
            observer.update(this);  
       }  
   }  
}

EstateMediator类继承Subject类，因此它拥有维护Observer对象列表以及通知每一个Observer对象的能力。除此之外，EstateMediator提供接口来保存外部传递给它的通知。

public class EstateMediator extends Subject {  
​  
    private String notification;  
​  
    public String getNotification() {  
        return notification;  
   }  
​  
    public void setNotification(String notification) {  
        this.notification = notification;  
   }  
​  
    public void acceptNotification(String notification) {  
        setNotification(notification);  
        notifyObservers();  
   }  
}

客户端代码分别生成2个房主和租房人对象，通过attach方法保存到中介维护的Observer列表中，然后向中介发出通知。中介会将通知发送给每一个房主和租房人。房主和租房人会各自处理该此消息。

EstateMediator estateMediator = new EstateMediator();  
​  
HouseOwner houseOwner1 = new HouseOwner();  
HouseOwner houseOwner2 = new HouseOwner();  
Tenant tenant1 = new Tenant();  
Tenant tenant2 = new Tenant();  
​  
estateMediator.attach(houseOwner1);  
estateMediator.attach(houseOwner2);  
estateMediator.attach(tenant1);  
estateMediator.attach(tenant2);  
​  
estateMediator.acceptNotification("[金色家园]11月5日修理自来水管道，8：00-9：00停水");

输出结果和之前一样，符合预期。

房主：收到维修通知-[金色家园]11月5日修理自来水管道，8：00-9：00停水  
房主：收到维修通知-[金色家园]11月5日修理自来水管道，8：00-9：00停水  
租房人：收到维修通知-[金色家园]11月5日修理自来水管道，8：00-9：00停水  
租房人：我会提前准备生活用水  
租房人：收到维修通知-[金色家园]11月5日修理自来水管道，8：00-9：00停水  
租房人：我会提前准备生活用水

熊小猫：这版代码改造得怎么样，我修改需求一试便知。我在需求中加入新的通知发送者——居委会。居委会发布活动通知，只有房主会关注居委会的消息，收到通知后报名参加活动。你看看如何实现。

兔小白：这个简单，我增加居委会类Committee，同样继承自Subject类。因此，它也拥有了维护Observer对象列表、发送通知的能力。此外，Committee还可以创建活动，然后发出通知。

public class Committee extends Subject {  
    private String activityRegistration;  
​  
    public String getActivityRegistration() {  
        return activityRegistration;  
   }  
​  
    public void setActivityRegistration(String activityRegistration) {  
        this.activityRegistration = activityRegistration;  
   }  
​  
    public void holdActivity(String activity) {  
        setActivityRegistration(activity);  
        notifyObservers();  
   }  
}

HouseOwner类的update方法需要增加Committee类型的消息处理。

public class HouseOwner implements Observer {   
    public void update(Subject subject) {  
        if (subject instanceof EstateMediator) {  
            System.out.println("房主：收到维修通知-" +  
                   ((EstateMediator) subject).getNotification());  
       }  
​  
        if (subject instanceof Committee) {  
            System.out.println("房主：收到维修通知-" +  
                   ((Committee) subject).getActivityRegistration());  
            System.out.println("房主：报名参加");  
       }  
   }  
}

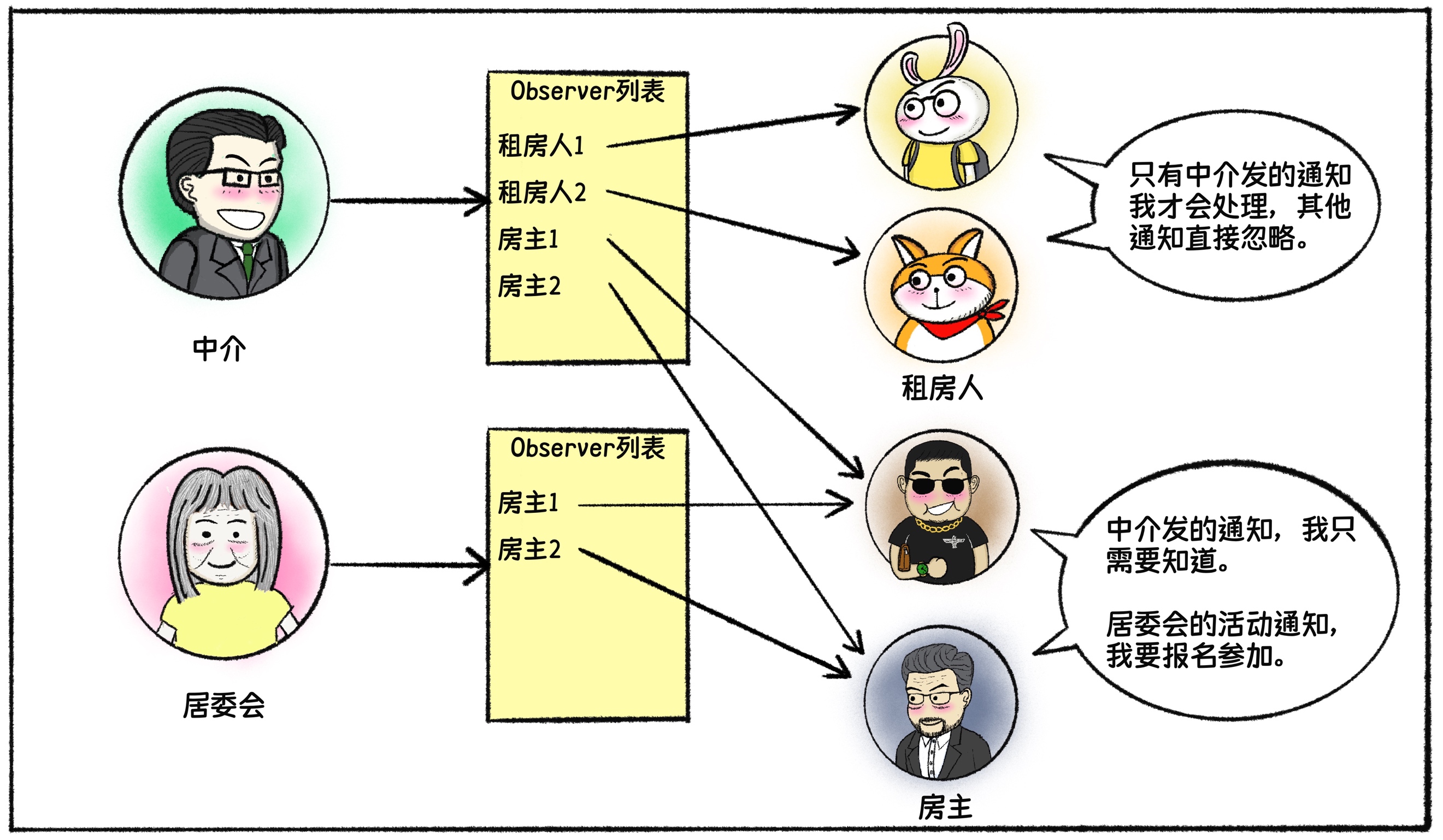
客户端代码中，创建两个HouseOwner对象，注册到Committee的Observer列表中。调用Committee对象的举办活动方法holdActivity，触发发送活动通知。

HouseOwner houseOwner1 = new HouseOwner();  
HouseOwner houseOwner2 = new HouseOwner();  
Committee committee = new Committee();  
committee.attach(houseOwner1);  
committee.attach(houseOwner2);  
committee.holdActivity("[南湖街道居委会]将于11月8日14：00-15：00举办亲子阅读活动,欢迎咨询报名");

HouseOwner对象在收到通知后，将执行报名逻辑。程序输出如下。

房主：收到维修通知-[南湖街道居委会]将于11月8日14：00-15：00举办亲子阅读活动,欢迎咨询报名  
房主：报名参加  
房主：收到维修通知-[南湖街道居委会]将于11月8日14：00-15：00举办亲子阅读活动,欢迎咨询报名  
房主：报名参加

中介、居委会、房主、租房人的关系如下图所示。

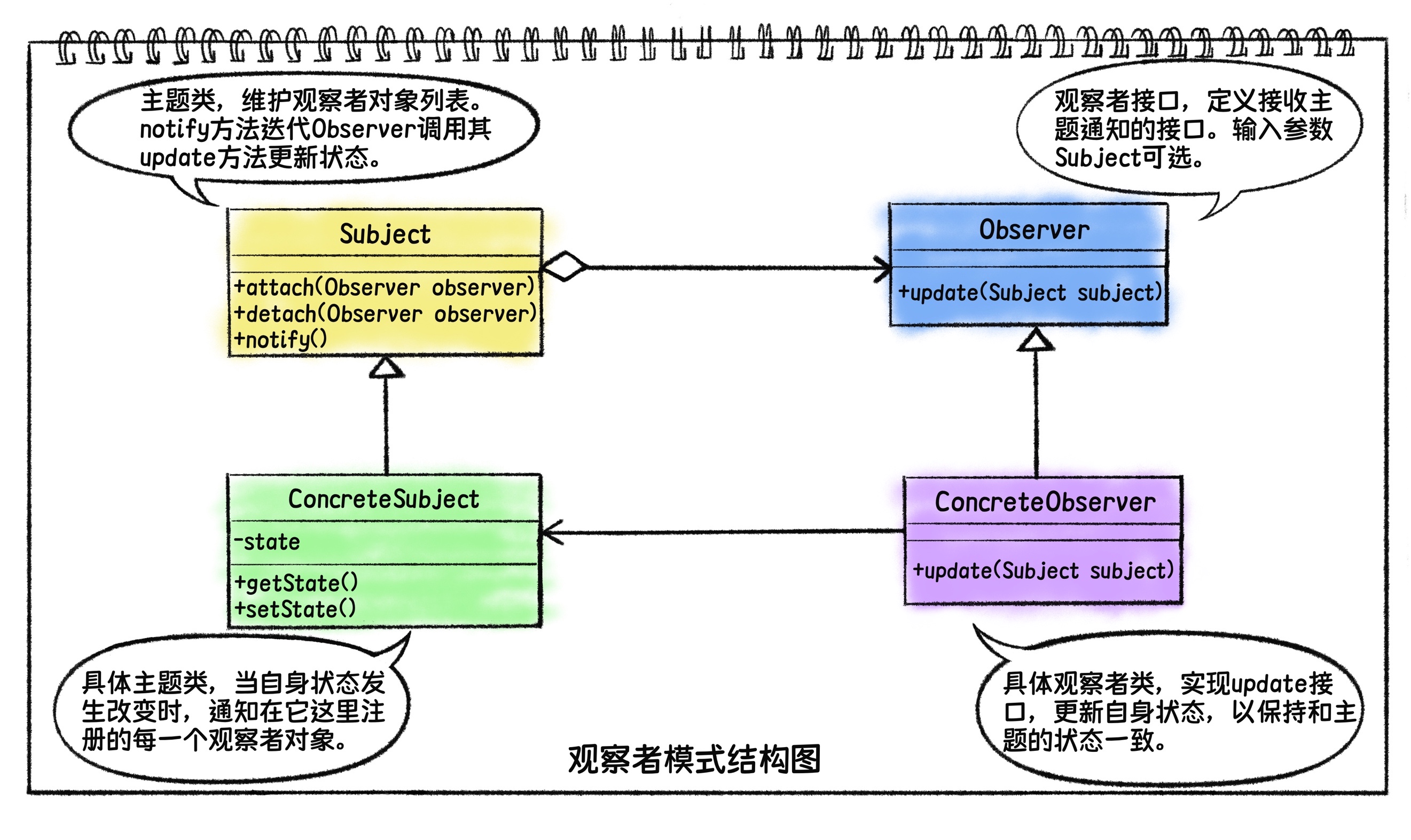


熊小猫：这就是面向接口编程的好处，不会把两个具体的类彻底绑定。类之间的依赖只认“类型”，扩展就来自于此。

这版代码基本实现了观察者模式。下面我再讲解观察者模式，你一定很容易就能理解。

## 20.4 观察者模式的优缺点及适用场景

熊小猫：我们先来看看观察者模式的结构图。



观察者模式也被称为发布/订阅模式。它的结构很简单，分为Subject和Observer两个继承体系。Subject意为主题，是被订阅的主体。Observer是观察者，可以关注Subject。一个Subject会有多个Observer对象关注它，当Subject状态变化时，会通过notify方法通知每一个观察它的Observer对象。

一个Observer对象也可以关注多个Subject。当Observer对象的update方法被触发时，它需要判断是哪一个主题触发，然后进行相应的处理。

练习中的EstateMediator和Committee属于Subject。HouseOwner和Tenant则属于Observer。

观察者模式定义了一种一对多的对象依赖关系。当Subject对象的状态发生改变时，所有依赖它的对象都将得到通知。

观察者模式让主题和观察者只依赖对方的接口，它拥有如下优点。

1. 观察者易于编写。某个对象只要属于Observer接口的实现类，就可以成为主题的观察者。Observer的接口非常简单，只含有一个高度抽象的update方法。这使得已经存在的类可以被方便地改造为观察者。
2. 主题和观察者可以独立发展。二者接口依赖， 在符合接口定义的前提下，既可以发展出一系列丰富多彩的主题，又可以扩展出功能各异的观察者。

兔小白：观察者之所以这么容易被扩展，关键在于高度抽象的update方法。

熊小猫：确实如此。在实际开发中，update可以不接收任何参数，只作为一个更新触发点。它也可以接受Subject类型参数，将Subject的自身状态传递给Observer。如果Observer需要更多的数据，也可以扩展update方法的参数列表，将与Subject状态无关，但与Observer的update行为有关的信息分离出来，单独作为参数传递给Observer。

兔小白：我觉得观察者模式有一个不太好的地方。由于update方法的抽象程度过高，当Subject传递了一些不恰当的数据给Observer时，接口可以正常调用，但在执行过程中可能会出现问题。这种问题在编译期无法被暴露出来。

熊小猫：接口抽象程度的提高，可以带来更好的通用性。但接口定义过于宽泛，会延迟问题暴露。建议不要使用update接口传递复杂的数据结构。在实际开发中，最好让Observer根据Subject的状态更新，自己去获取update操作所需要的数据。

接下来我们看看观察者模式的适用场景。

1. 一对多的对象关系中，一方对象的状态改变会联动多方响应。例如数据可视化，同一份数据可能绑定多种展示图形，数据的变化会触发所有依赖它的图形发生变化。
2. 广播消息的场景。观察者模式也称为发布/订阅。消息的发布者将消息广播给每一位订阅者，这是观察者模式的核心应用场景。例如在窗体界面中存在多个窗口，某个窗口状态的改变，会引起其窗口的改变。此时可以使用中介者+观察者模式，变化的窗口将自己状态的变更发布给中介者，中介者再使用观察者模式，将窗口状态变更的消息广播给所有感兴趣的窗体控件。
3. Subject不关心Observer的处理结果。如果Subject需要使用Observer的处理结果执行后续操作，那么并不适合使用观察者模式。原因在于，不同类型的Observer的update方法执行结果，可能存在较大差异，难以返回类型一致的结果。

观察者模式除了应用在对象解耦，还被广泛应用于系统间的解耦。如果两个系统通过API直接调用，会造成高度耦合。我们可以借助消息中间件，如Kafka，系统间以收发消息的形式进行通信。

兔小白：看来学习好设计模式，对系统架构也非常有帮助！

熊小猫：学习需要把底层原理理解清楚，触类旁通，否则是学不过来的。这也是为什么Java内置了观察者模式的实现，我还要让你自己思考，写了一遍代码。

兔小白：啊......我又在重复造轮子？

## 20.5 Java内置的观察者模式实现

熊小猫：这个轮子造的有意义呀！自己动手才能深入理解。Java从JDK 1.0开始，内置了Observable类，以及Observer接口。Observable类对应观察者模式中的Subject类。下面我将你的代码改造为使用Java内置接口实现观察者模式，非常简单。

你定义的Subject类和Observer接口不再需要，可以直接移除。

让EstateMediator类继承Observable类。在acceptNotification中，需要先调用Observable的setChanged方法设置changed状态为true，这样在调用notifyObservers方法时，才会发布消息给Observer。

public class EstateMediator extends Observable {  
​  
    private String notification;  
​  
    public String getNotification() {  
        return notification;  
   }  
​  
    public void setNotification(String notification) {  
        this.notification = notification;  
   }  
​  
    public void acceptNotification(String notification) {  
        setNotification(notification);  
        setChanged();  
        notifyObservers();  
   }  
}

HouserOwner和Tenant的修改类似，实现Java内置的Observer接口，update方法的入参增加了一个Object类型的对象，用来传递不包含在Observable中的必要数据。下面的代码以Tenant为例。

public class Tenant implements Observer {  
    public void update(Observable observable, Object arg) {  
        if (observable instanceof EstateMediator) {  
            System.out.println("租房人：收到维修通知-" +  
                   ((EstateMediator) observable).getNotification());  
            System.out.println("租房人：我会提前准备生活用水");  
       }  
   }  
}

客户端代码几乎没有改动，只是将添加Observer的方法改为addObserver。

EstateMediator estateMediator = new EstateMediator();  
​  
HouseOwner houseOwner1 = new HouseOwner();  
HouseOwner houseOwner2 = new HouseOwner();  
Tenant tenant1 = new Tenant();  
Tenant tenant2 = new Tenant();  
​  
estateMediator.addObserver(houseOwner1);  
estateMediator.addObserver(houseOwner2);  
estateMediator.addObserver(tenant1);  
estateMediator.addObserver(tenant2);  
​  
estateMediator.acceptNotification("[金色家园]11月5日修理自来水管道，8：00-9：00停水");

整个改造变化并不大。我们使用Java提供的观察者模式接口可以满足大多数场景。但是由于Java不支持多继承，如果作为Subject的类存在父类，那么需要让Subject的父类继承Observable类。

兔小白：我的体会是，尽量使用Java的内置实现，不到万不得已，不要自己去造轮子。否则可能费力不讨好呀！