2 观察者（Observer）模式

让你的对象知悉现况

有趣的事情发生时，可千万别错过了！ 有一个模式可以帮你的对象知悉现况，不会错过该对象感兴趣的事。对象甚至在运行时可决定是否要继续被通知。观察者模式是JDK中使用最多的模式之一，非常有用。我们也会一并介绍一对多关系，以及松耦合（对，没错，我们说耦合）。有了观察者，你将会消息灵通。

恭喜你！

你的团队刚刚赢得一纸合约，负责建立Weather-O-Rama公司的下一代气象站——Internet气象观测站。

Weather-O-Rama气象站
100 Main Street
Tornado Alley, OK 45021

工作合约

恭喜贵公司获选为敝公司建立下一代Internet气象观测站！该气象站必须建立在我们专利申请中的WeatherData对象上，由WeatherData对象负责追踪目前的天气状况（温度、湿度、气压）。我们希望贵公司能建立一个应用，有三种布告板，分别显示目前的状况、气象统计及简单的预报。当WeatherObject对象获得最新的测量数据时，三种布告板必须实时更新。

而且，这是一个可以扩展的气象站，Weather-O-Rama气象站希望公布一组API，好让其他开发人员可以写出自己的气象布告板，并插入此应用中。我们希望贵公司能提供这样的API。

Weather-O-Rama气象站有很好的商业营运模式：一旦客户上钩，他们使用每个布告板都要付钱。最好的部分就是，为了感谢贵公司建立此系统，我们将以公司的认股权支付你。

我们期待看到你的设计和应用的alpha版本。

真挚的

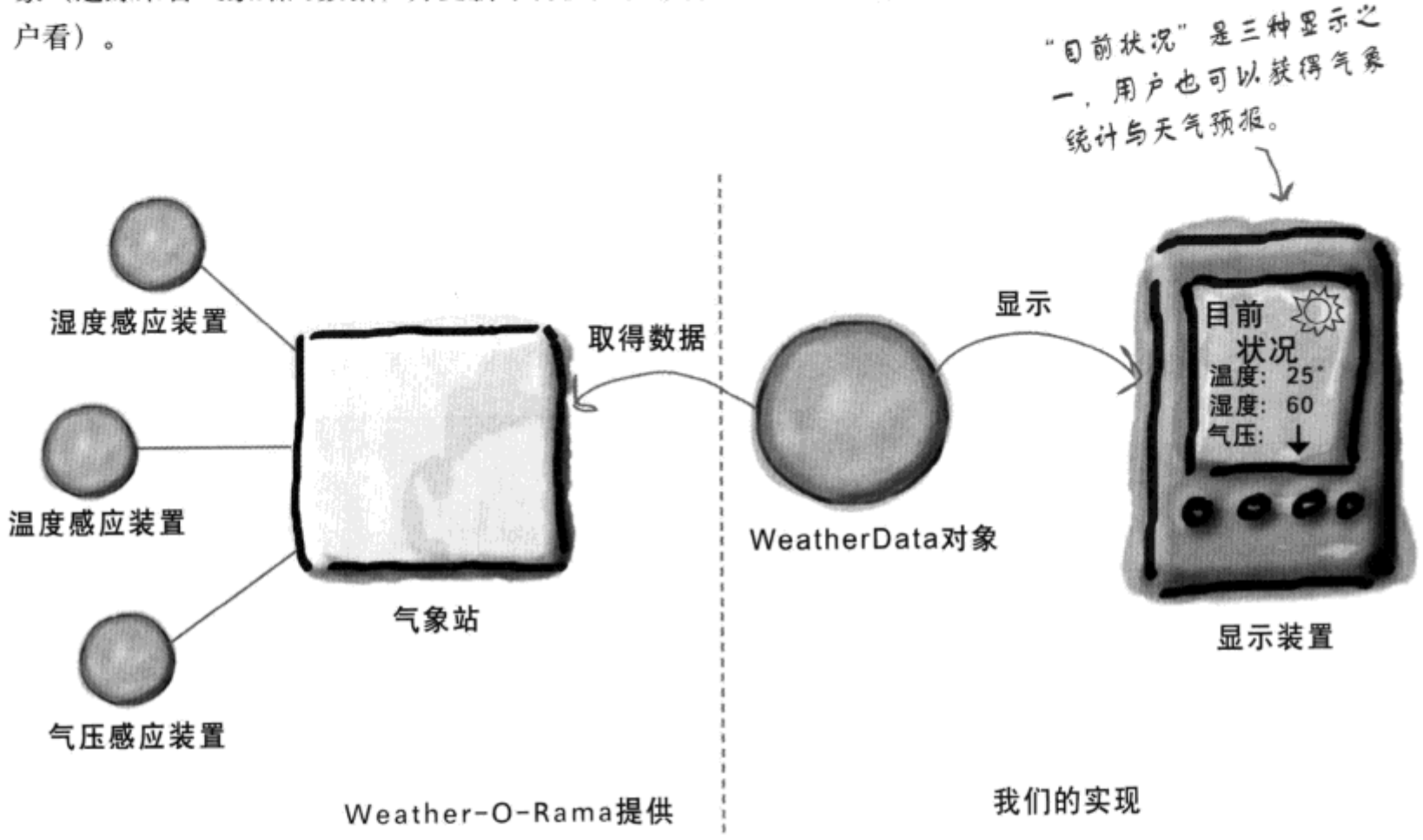
Johnny Hurricane

Johnny Hurricane——Weather-O-Rama气象站执行长

附注：我们正通宵整理WeatherData源文件给你们。

气象监测应用的概况

此系统中的三个部分是气象站（获取实际气象数据的物理装置）、WeatherData对象（追踪来自气象站的数据，并更新布告板）和布告板（显示目前天气状况给用户看）。

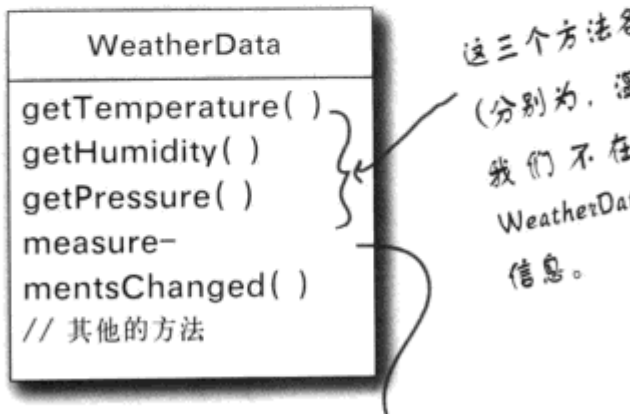


WeatherData对象知道如何跟物理气象站联系，以取得更新的数据。WeatherData对象会随即更新三个布告板的显示：目前状况（温度、湿度、气压）、气象统计和天气预报。

如果我们选择接受这个项目，我们的工作就是建立一个应用，利用WeatherData对象取得数据，并更新三个布告板：目前状况、气象统计和天气预报。

瞧一瞧刚送到的WeatherData类

如同他们所承诺的，隔天早上收到了WeatherData源文件，看了一下代码，一切都很直接：



这三个方法各自返回最近的气象测量数据（分别为，温度、湿度、气压）。我们不在乎这些变量“如何”被设置，WeatherData对象自己知道如何从气象站获取更新信息。

```
/*
 * 一旦气象测量更新，此方法会被调用
 */
public void measurementsChanged() {
  // 你的代码加在这里
}
```

WeatherData.java

WeatherObject的开发人员留了一个线索，好让我们知道该加些什么……

再次提醒，这只是三个显示布告板中的一个。

显示装置

我们的工作是实现measurementsChanged()，好让它更新目前状况、气象统计、天气预报的显示布告板。

我们目前知道些什么？

Weather-O-Rama气象站的要求说明并不是很清楚，我们必须搞懂该做些什么。那么，我们目前知道些什么呢？

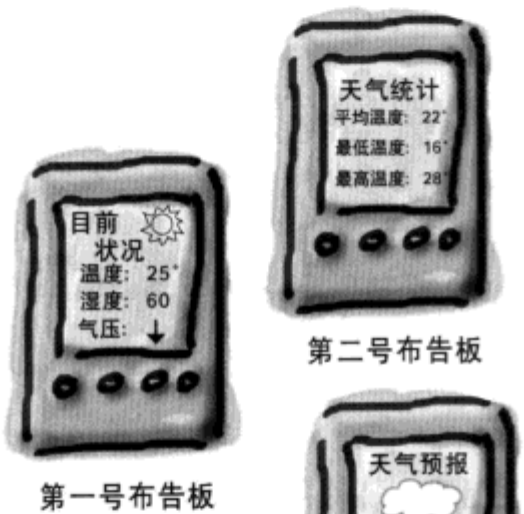
- WeatherData类具有getter方法，可以取得三个测量值：温度、湿度与气压。

```
getTemperature()
getHumidity()
getPressure()
```

- 当新的测量数据备妥时，measurementsChanged()方法就会被调用（我们不在乎此方法是如何被调用的，我们只在乎它被调用了）。

```
measurementsChanged()
```

- 我们需要实现三个使用天气数据的布告板：“目前状况”布告、“气象统计”布告、“天气预报”布告。一旦WeatherData有新的测量，这些布告必须马上更新。



第一号布告板

第二号布告板

第三号布告板

- 此系统必须可扩展，让其他开发人员建立定制的布告板，用户可以随心所欲地添加或删除任何布告板。目前初始的布告板有三类：“目前状况”布告、“气象统计”布告、“天气预报”布告。

将来的布告板

先看一个错误示范

这是第一个可能的实现：我们依照Weather-O-Rama气象站开发人员的暗示，在measurementsChanged()方法中添加我们的代码：

```
public class WeatherData {

    // 实例变量声明

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // 这里是其他WeatherData方法
}
```

调用 WeatherData 的三个getXxx()方法，以取得最近的测量值。这些getXxx()方法已经实现好了。

现在，更新布告板……

调用每个布告板更新显示，传入最新的测量。

Sharpen your pencil

在我们的第一个实现中，下列哪种说法正确？（多选）

- ❑ A. 我们是针对具体实现编程，而非针对接口。
- ❑ B. 对于每个新的布告板，我们都得修改代码。
- ❑ C. 我们无法在运行时动态地增加（或删除）布告板。
- ❑ D. 布告板没有实现一个共同的接口。
- ❑ E. 我们尚未封装改变的部分。
- ❑ F. 我们侵犯了WeatherData类的封装。

SWAG的定义:Scientific Wild A** Guess

我们的实现有什么不对？

回想第 1 章的概念和原则……

```
public void measurementsChanged( ) {

    float temp = getTemperature( );
    float humidity = getHumidity( );
    float pressure = getPressure( );

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

改变的地方，需要封装起来。

针对具体实现编程，会导致我们以后在增加或删除布告板时必须修改程序。

至少，这里看起来像是一个统一的接口，布告板的方法名称都是*update()*，参数都是温度、湿度、气压。

我们现在就来看观察者模式，然后再回来看看如何将此模式应用到气象观测站。

认识观察者模式

我们看看报纸和杂志的订阅是怎么回事：

1. 报社的业务就是出版报纸。

2. 向某家报社订阅报纸，只要他们有新报纸出版，就会给你送来。只要你是他们的订户，你就会一直收到新报纸。

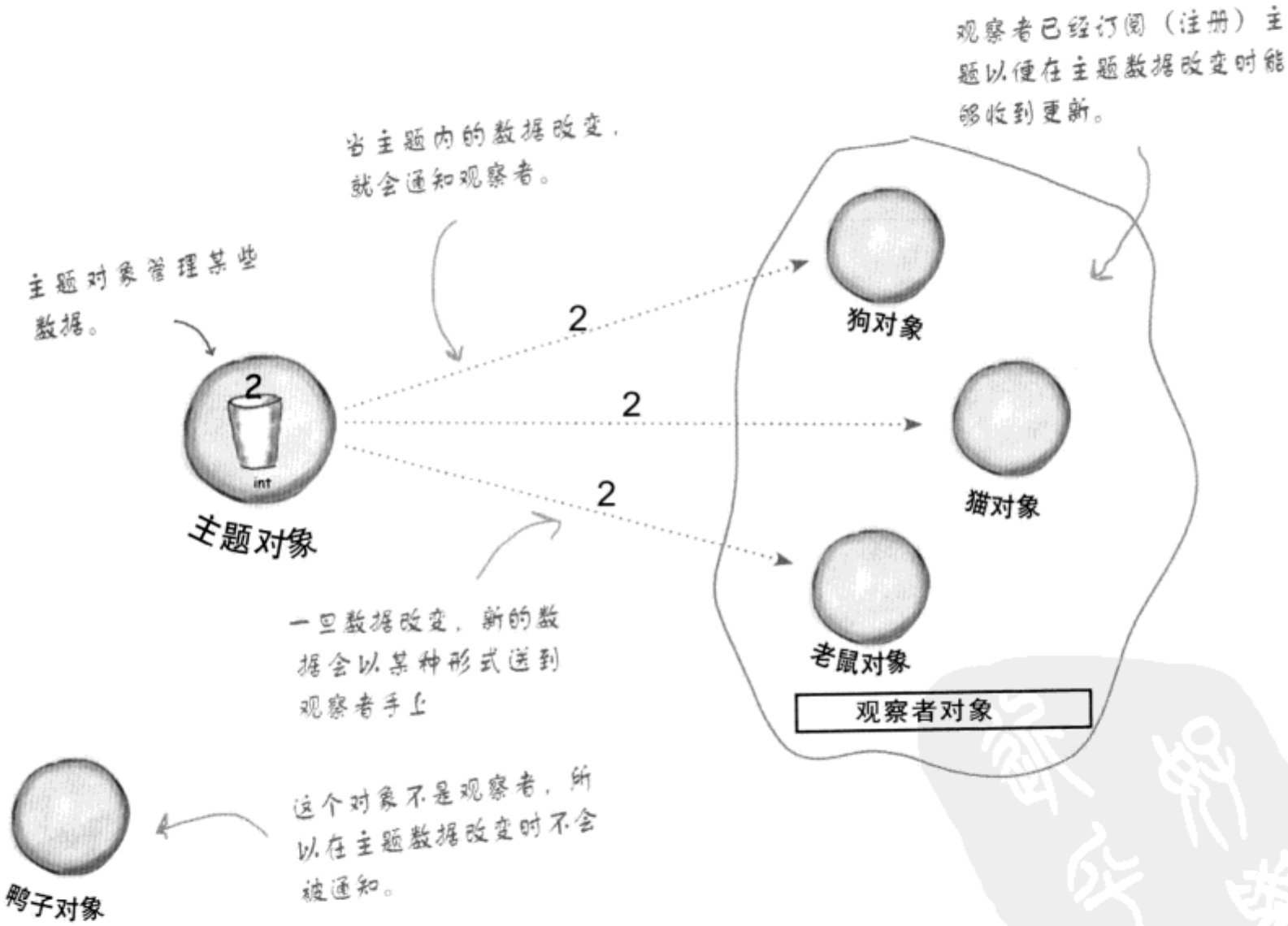
3. 当你不想再看报纸的时候，取消订阅，他们就不会再送新报纸来。

4. 只要报社还在运营，就会一直有人（或单位）向他们订阅报纸或取消订阅报纸。

出版者+订阅者=观察者模式

如果你了解报纸的订阅是怎么回事，其实就知道观察者模式是怎么回事，只是名称不太一样：出版者改称为“主题”（Subject），订阅者改称为“观察者”（Observer）。

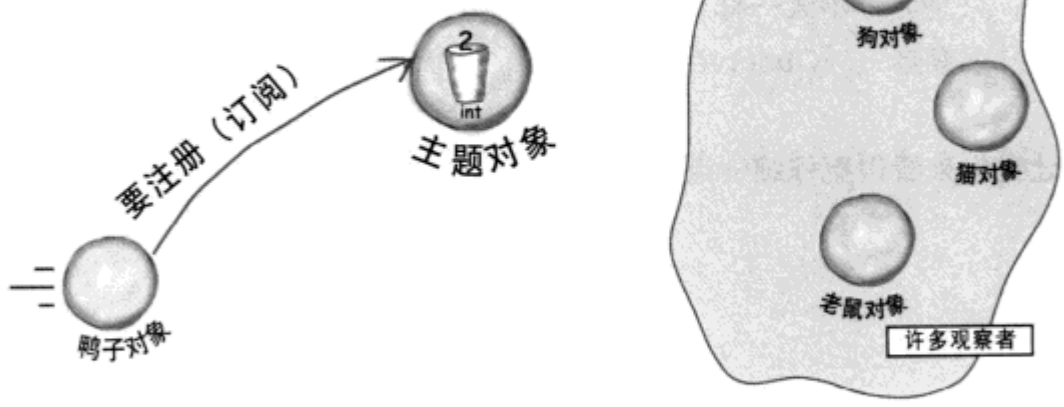
让我们来看得更仔细一点：



观察者模式的一天

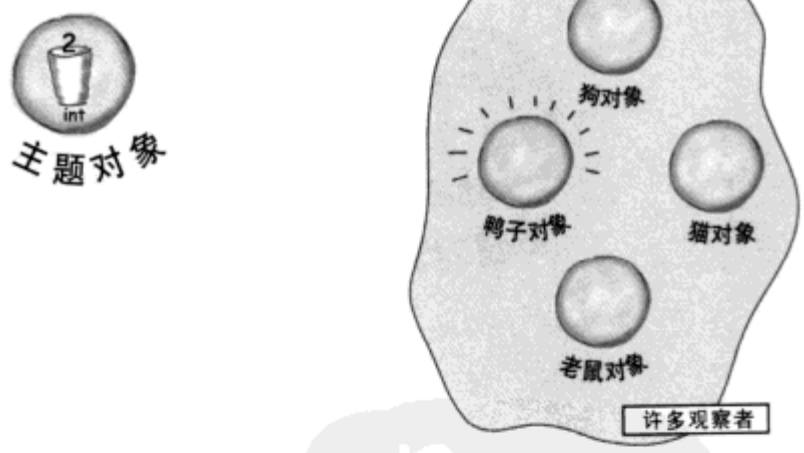
鸭子对象过来告诉主题，它想当一个观察者。

鸭子其实想说的是：我对你的数据改变感兴趣，一有变化请通知我。



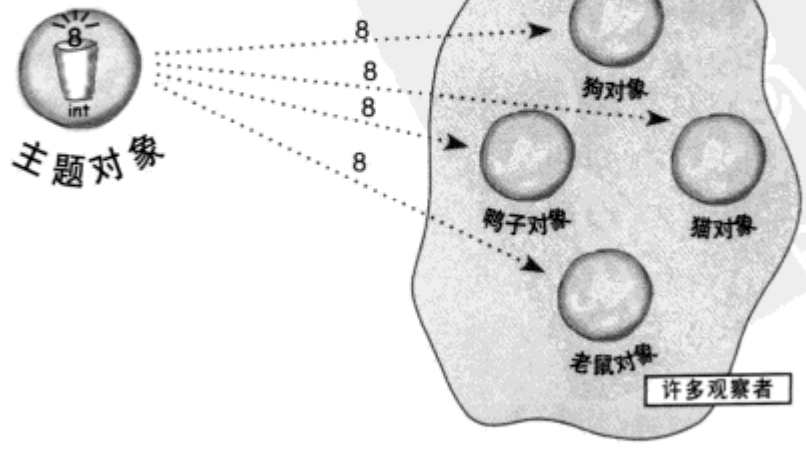
鸭子对象现在已经是正式的观察者了。

鸭子静候通知，等待参与这项伟大的事情。一旦接获通知，就会得到一个整数。



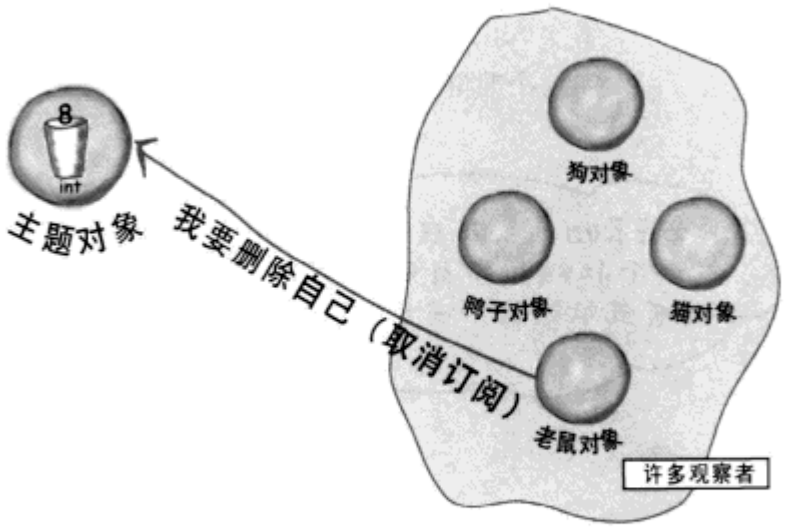
主题有了新的数据值！

现在鸭子和其他所有观察者都会收到通知：主题已经改变了。



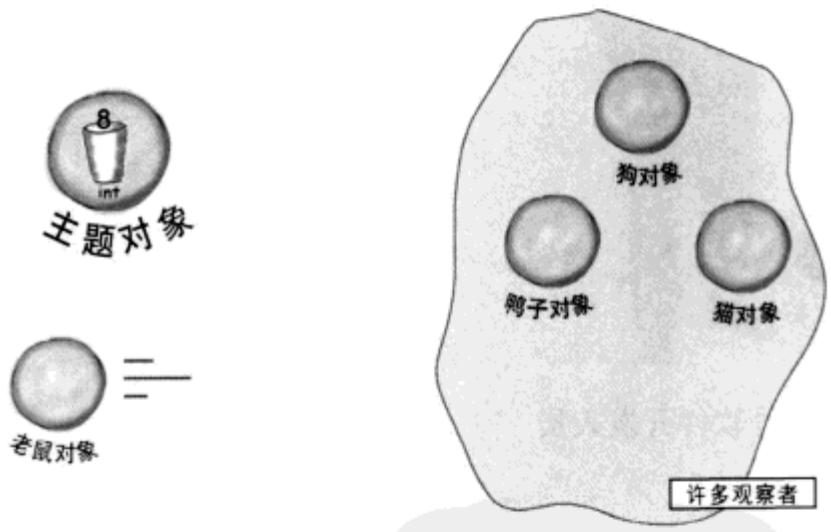
老鼠对象要求从观察者中把自己除名。

老鼠已经观察此主题太久，厌倦了，所以决定不再当个观察者。



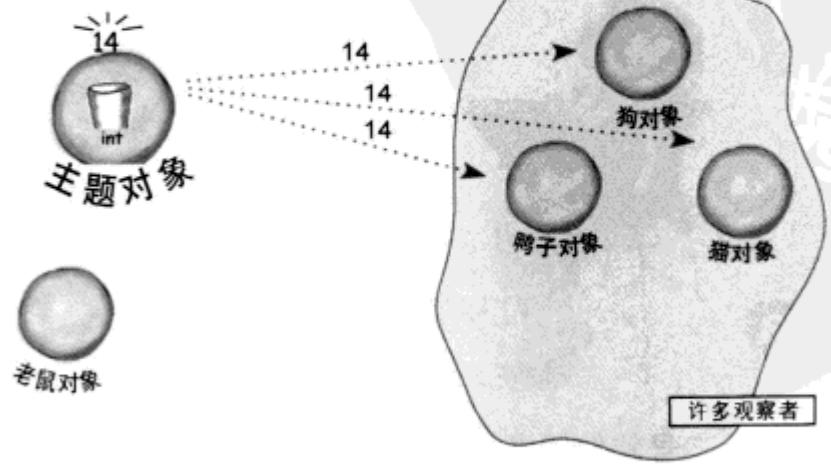
老鼠离开了!

主题知道老鼠的请求之后，把它从观察者中除名。



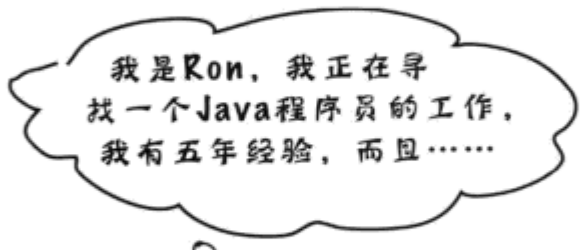
主题有一个新的整数。

除了老鼠之外,每个观察者都会收到通知，因为它已经被除名了。嘘！不要告诉别人，老鼠其实心中暗暗地怀念这些整数，或许哪天又会再次注册，回来继续当观察者呢!

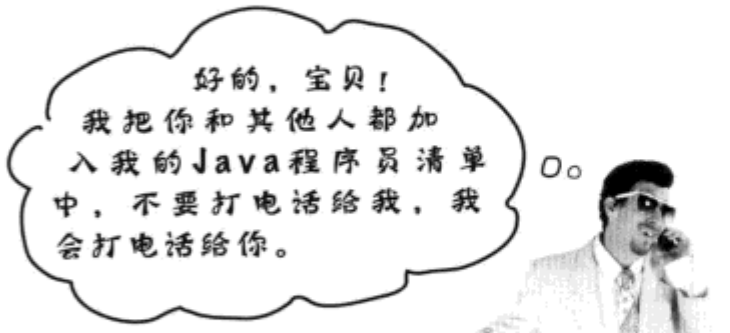


五分钟短剧：观察的主题

在今天的讽刺短剧中，有两个后泡沫时期的软件工程师，遇到一个真正的猎头……



一号软件开发人员



猎头/主题

二号软件开发人员

主题

❺ 其间，Ron和Jill继续过自己的日子，如果Java工作来了，他们会接到通知，毕竟，他们是观察者嘛！

❻ 主题

❼ 观察者

观察者

Jill自己找到工作了！

❽ 观察者

❾ 主题

两周后……

Jill热爱她的现状，她不再是观察者了。她还因为签约获得了一笔奖金，因为公司不用拨出一大笔钱给猎头。

但是，我们亲爱的Ron，又如何了？我们听说他设局把原来的猎头搞得毫无招架之力。他不只是一个观察者，也有了自己的求职者清单，只要付一笔钱给猎头，就可从其他求职者赚取更多钱。Ron既是一个主题，也是一个观察者，集两种角色于一身。

定义观察者模式

当你试图勾勒观察者模式时，可以利用报纸订阅服务，以及出版者和订阅者比拟这一切。

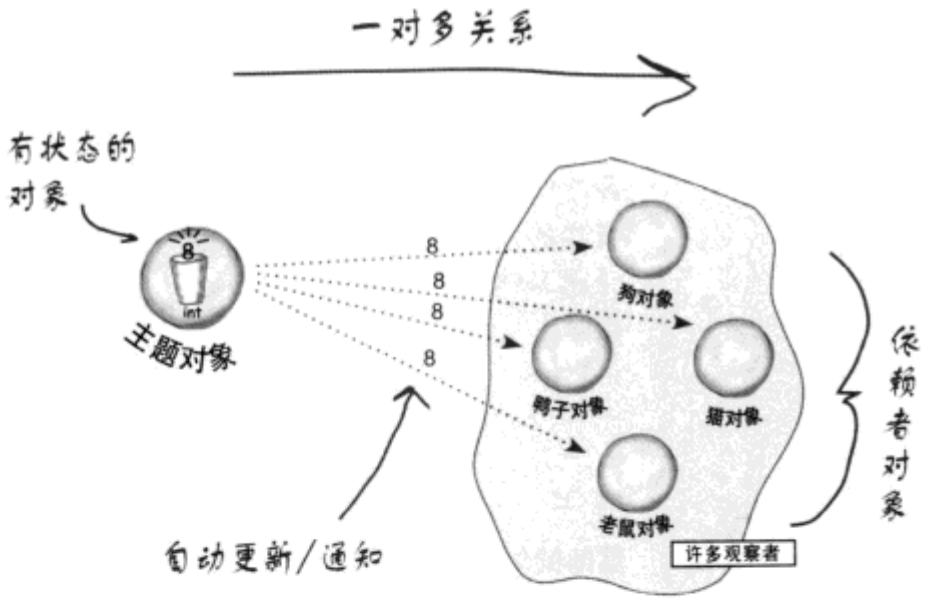
在真实的世界中，你通常会看到观察者模式被定义成：

> **观察者模式**定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

观察者模式定义了一系列对象之间的一对多关系。

当一个对象改变状态，其他依赖者都会收到通知。

让我们看看这个定义，并和之前的例子做个对照：

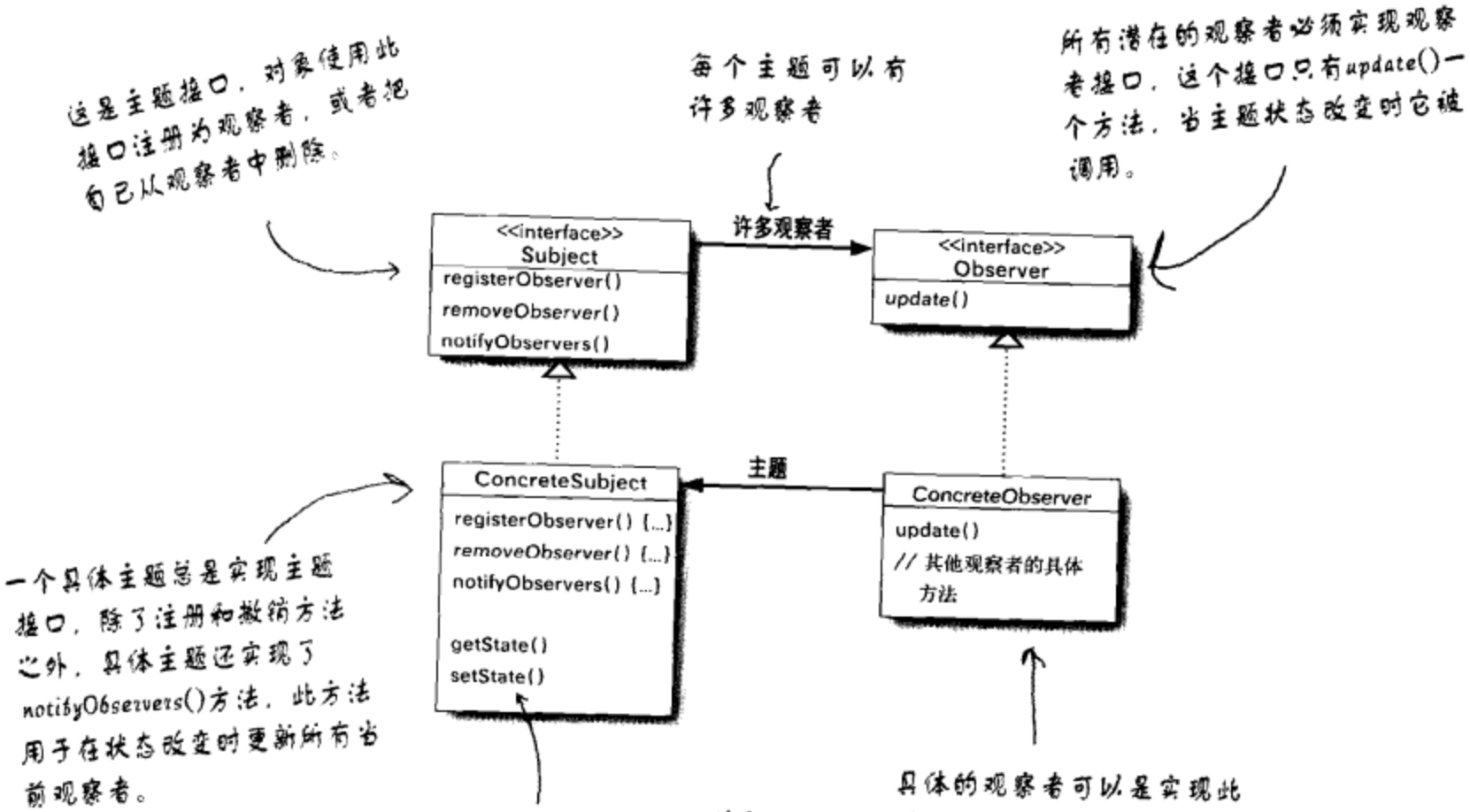


主题和观察者定义了一对多的关系。观察者依赖于此主题，只要主题状态一有变化，观察者就会被通知。根据通知的风格，观察者可能因此新值而更新。

稍后你会看到，实现观察者模式的方法不只一种，但是以包含Subject与Observer接口的类设计的做法最常见。

让我们快来看看吧……

定义观察者模式：类图



there are no Dumb Questions

问： 这和一对多的关系有何关联？

答： 利用观察者模式，主题是具有状态的对象，并且可以控制这些状态。也就是说，有“一个”具有状态的主题。另一方面，观察者使用这些状态，虽然这些状态并不属于他们。有许多的观察者，依赖主题来告诉他们状态何时改变了。这就产生一个关系：“一个”主题对“多个”观察者的关系。

问： 其间的依赖是如何产生的？

答： 因为主题是真正拥有数据的人，观察者是主题的依赖者，在数据变化时更新，这样比起让许多对象控制同一份数据来，可以得到更干净的的OO设计。

松耦合的威力

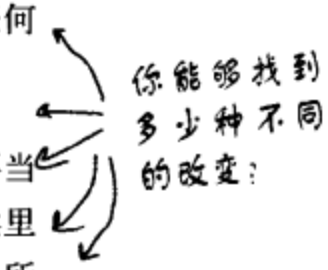
当两个对象之间松耦合，它们依然可以交互，但是不太清楚彼此的细节。

观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

为什么呢？

关于观察者的一切，主题只知道观察者实现了某个接口（也就是Observer接口）。主题不需要知道观察者的具体类是谁、做了些什么或其他任何细节。

任何时候我们都可以增加新的观察者。因为主题唯一依赖的东西是一个实现Observer接口的对象列表，所以我们可以随时增加观察者。事实上，在运行时我们可以用新的观察者取代现有的观察者，主题不会受到任何影响。同样的，也可以在任何时候删除某些观察者。



有新类型的观察者出现时，主题的代码不需要修改。假如我们有个新的具体类需要当观察者，我们不需要为了兼容新类型而修改主题的代码，所有要做的就是在新的类里实现此观察者接口，然后注册为观察者即可。主题不在乎别的，它只会发送通知给所有实现了观察者接口的对象。

我们可以独立地复用主题或观察者。如果我们在其他地方需要使用主题或观察者，可以轻易地复用，因为二者并非紧耦合。

改变主题或观察者其中一方，并不会影响另一方。因为两者是松耦合的，所以只要他们之间的接口仍被遵守，我们就可以自由地改变他们。

设计原则

为了交互对象之间的松耦合设计而努力。

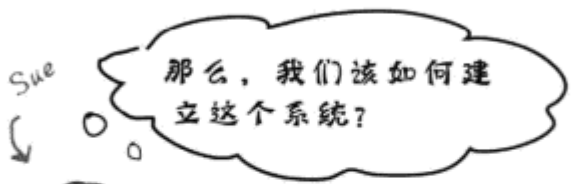
松耦合的设计之所以能让我们建立有弹性的OO系统，能够应对变化，是因为对象之间的互相依赖降到了最低。

在继续后面的内容之前，请试着画出实现气象站所需要的类，其中包括WeatherData类及布告板组件。确定你的图能够显示出各个部分如何结合起来，以及别的开发人员如何能够实现他自己的布告板组件。

如果你需要一点小帮助，请阅读下一页，你的队友正在讨论如何设计气象站。

办公室隔间对话

回到气象站项目，你的队友们已经开始全面思考这个问题了……



Mary：这个嘛！使用观察者模式啰！

Sue：是的……但是如何应用？

Mary：唔，我们再看一下定义好了：

观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

Mary：当你思考这个定义时，你会发现很有道理。我们的WeatherData类正是此处所说的“一”，而我们的“多”正是使用天气观测的各种布告板。

Sue：没错。WeatherData对象的确是有状态，包括了温度、湿度、气压，而这些值都会改变。

Mary：对呀！而且，当这些观测值改变时，必须通知所有的布告板，好让它们各自做出处理。

Sue：好棒！我现在知道如何将观察者模式应用在气象站问题上了。

Mary：还有一些问题有待理清，我现在还不太了解它们的解决方法。

Sue：什么问题？

Mary：其中一个问题是，我们如何将气象观测值放到布告板上。

Sue：回头去看看观察者模式的图，如果我们把WeatherData对象当作主题，把布告板当作观察者，布告板为了取得信息，就必须先向WeatherData对象注册。对不对？

Mary：是的……一旦WeatherData知道有某个布告板的存在，就会适时地调用布告板的某个方法来告诉布告板观测值是多少。

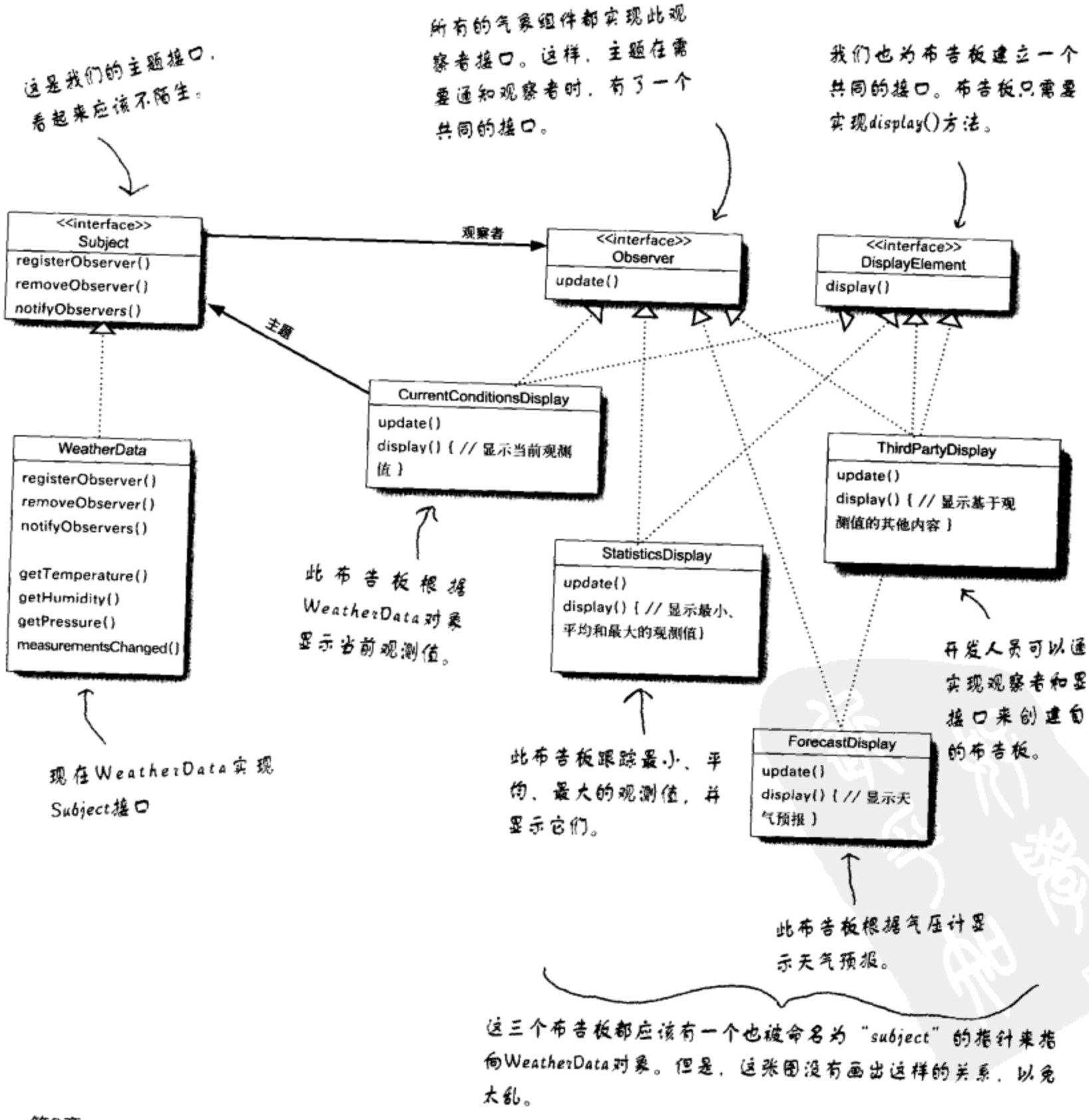
Sue：我们必须记得，每个布告板都有差异，这也就是为什么我们需要一个共同的接口的原因。尽管布告板的类都不一样，但是它们都应该实现相同的接口，好让WeatherData对象能够知道如何把观测值送给它们。

Mary：我懂你的意思。所以每个布告板都应该有一个大概名为update()的方法，以供WeatherData对象调用。

Sue：而这个update()方法应该在所有布告板都实现的共同接口里定义。

设计气象站

看看这个设计图，和你的设计图有何异同？



实现气象站

依照两页前Mary和Sue的讨论，以及上一页的类图，我们要开始实现这个系统了。稍后，你将会在本章看到Java为观察者模式提供了内置的支持，但是，我们暂时不用它，而是先自己动手。虽然，某些时候可以利用Java内置的支持，但是有许多时候，自己建立这一切会更具弹性（况且建立这一切并不是很麻烦）。所以，让我们从建立接口开始吧：

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

这两个方法都需要一个观察者作为变量，该观察者是用来注册或被删除的。

当主题状态改变时，这个方法会被调用，以通知所有的观察者。

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

当气象观测值改变时，主题会把这些状态值当作方法的参数，传送给观察者。

所有的观察者都必须实现update()方法，以实现观察者接口。在这里，我们按照Mary和Sue的想法把观测值传入观察者中。

```
public interface DisplayElement {
    public void display();
}
```

DisplayElement接口只包含了一个方法，也就是display()。当布告板需要显示时，调用此方法。

Mary和Sue认为：把观测值直接传入观察者中是更新状态的最直接的方法。你认为这样的做法明智吗？暗示：这些观测值的种类和个数在未来有可能改变吗？如果以后会改变，这些变化是否被很好地封装？或者是需要修改许多代码才能办到？

关于将更新的状态传送给观察者，你能否想到更好的方法解决此问题？

别担心，在我们完成第一次实现后，我们会再回来探讨这个设计决策。

在WeatherData中实现主题接口

还记得我们在本章一开始的地方就试图实现WeatherData类吗？你可以去回顾一下。现在，我们要用观察者模式实现……

提醒你：为了节省篇幅，我们在代码中没有列出import和package语句。你可以到wickedlysmart网站找到完整的源代码，URL在本书的第xxxv页。

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

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // WeatherData的其他方法
}
```

这部分是Subject接口的实现。

WeatherData现在实现了Subject接口。

我们加上一个ArrayList来纪录观察者，此ArrayList是在构造器中建立的。

当注册观察者时，我们只要把它加到ArrayList的后面即可。

同样地，当观察者想取消注册，我们把它从ArrayList中删除即可。

有趣的地方来了！在这里，我们把状态告诉每一个观察者。因为观察者都实现了update()，所以我们知道如何通知它们。

当从气象站得到更新观测值时，我们通知观察者。

我们想要每本书随书赠送一个小型气象站，但是出版社不肯。所以，和从装置中读取实际的气象数据相比，我们宁愿利用这个方法来测试布告板。或者，为了好玩，你也可以写代码从网站上抓取观测值。

现在，我们来建立布告板吧！

我们已经把WeatherData类写出来了，现在轮到布告板了。Weather-O-Rama气象站订购了三个布告板：目前状况布告板、统计布告板和预测布告板。我们先看看目前状况布告板。一旦你熟悉此布告板之后，可以在本书的代码目录中，找到另外两个布告板的源代码，你会觉得这些布告板都很类似。

此布告板实现了Observer接口，所以可以从WeatherData对象中获得改变。

它也实现了DisplayElement接口，因为我们的API规定所有的布告板都必须实现此接口。

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

构造器需要weatherData对象（也就是主题）作为注册之用。

当update()被调用时，我们把温度和湿度保存起来，然后调用display()。

display()方法就只是把最近的温度和湿度显示出来。

there are no Dumb Questions

问：update()是最适合调用display()的地方吗？

答：在这个简单的例子中，当值变化的时候调用display()，是很合理的。然而，你是对的，的确是有很多更好的方法来设计显示数据的方式。当我们谈到MVC（Model-View-Controller）模式时会再作说明。

问：为什么要保存对Subject的引用呢？构造完后似乎用不着了呀？

答：的确如此，但是以后我们可能想要取消注册，如果已经有了对Subject的引用会比较方便。

启动气象站

❶ 先建立一个测试程序

气象站已经完成得差不多了，我们还需要一些代码将这一切连接起来。这是我们的第一次尝试，本书中稍后我们会再回来确定每个组件都能通过配置文件来达到容易“插拔”。现在开始测试吧：

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

首先，建立一个WeatherData对象。

如果你还不想下载完整的代码，可以将这两行注释掉，就能顺利执行了。

建立三个布告板，并把WeatherData对象传给它们。

模拟新的气象测量。

❷ 运行程序，让观察者模式表演魔术。

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Sharpen your pencil

Johnny Hurricane（Weather-O-Rama气象站的CEO）刚刚来电告知，他们还需要酷热指数（HeatIndex）布告板，这是不可或缺的。细节如下：

酷热指数是一个结合温度和湿度的指数，用来显示人的温度感受。可以利用温度T和相对湿度RH套用下面的公式来计算酷热指数：

$$
\begin{aligned}
heatindex = {} & 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T \\
& * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} \\
& * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - \\
& 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH \\
& + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * \\
& 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3
\end{aligned}
$$

开始练习打字吧！

开玩笑的啦！别担心，你不需要亲自输入此公式，只要建立你自己的HeatIndexDisplay.java文件并把公式从heatindex.txt文件中拷贝进来就可以了。

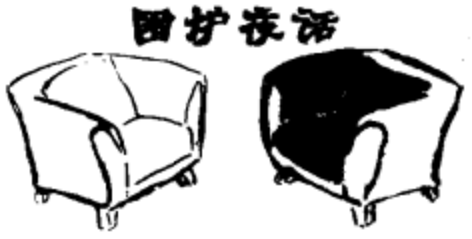
heatindex.txt文件可从wickedlysmart.com取得

这个公式是怎么回事？你可以参考《Head First气象学》，或者问问国家气象局的员工（或用Google搜索）。

当你完成后，输出结果应如下所示：

输出结果有改变的地方在这里。

```
File Edit Window Help OverdaRainbow
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```



今夜话题：主题和观察者就使观察者获得状态信息的正确方法发生了争吵。

主题

我很高兴，我们终于有机会面对面聊天了。

唉呀！我把该做的事都做到了，不是吗？我总是会通知你们发生什么事了……我虽然不知道你们是谁，但这不意味着我不在乎你们。况且，我知道关于你们的一件重要的事：你们实现了Observer接口。

是吗？说来听听！

拜托，我必须主动送出我的状态和通知给大家，好让你们这些懒惰的观察者知道发生什么事了。

嗯……这样或许也行，只是我必须因此门户大开，让你们全都可以进来取得你们需要的状态，这样太危险了。我不能让你们进来里面大肆挖掘我的各种数据。

观察者

是这样吗？我以为你根本不在乎我们这群观察者呢。

是呀，但这只是关于我的一小部分罢了！无论如何，我对你更了解……

嗯！你总是将你的状态传给我们，所以我们可以知道你内部的情况。有时候，这很烦人的……

咳！等等。我说主题先生，首先，我们并不懒，在你那些“很重要”通知的空档中，我们还有别的事要做。另外，为何由你主动送数据过来，而不是让我们主动去向你索取数据？

主题

是的，我可以让你们“拉”走我的状态，但是你不觉得这样对你们反而不方便吗？如果每次想要数据时都来找我，你可能要调用很多次才能收集齐全你所要的状态。这就是为什么我更喜欢“推”的原因，你们可以在一次通知中一口气得到所有东西。

是的。两种做法都有各自的优点。我注意到Java内置的Observer 模式两种做法都支持。

太好了，或许我会看到一个“拉”的好例子，因而改变我的想法。

观察者

你何不提供一些公开的getter方法，让我们“拉”走我们需要的状态？

死鸭子嘴硬！观察者种类这么多，你不可能事先料到我们每个人的需求，还是让我们直接去取得我们需要的状态比较恰当，这样一来，如果我们有人只需要一点点数据，就不会被强迫收到一堆数据。这么做同时也可以在以后比较容易修改。比方说，哪一天你决定扩展功能，新增更多的状态，如果采用我建议的方式，你就不用修改和更新对每位观察者的调用，只需改变自己来允许更多的getter方法来取得新增的状态。

真的吗？我们得去瞧瞧……

什么？我们会有意见相同的一天？不会吧！

使用Java内置的观察者模式

到目前为止，我们已经从无到有地完成了观察者模式，但是，Java API有内置的观察者模式。java.util包（package）内包含最基本的Observer接口与Observable类，这和我们的Subject接口与Observer接口很相似。Observer接口与Observable类使用上更方便，因为许多功能都已经事先准备好了。你甚至可以使用推（push）或拉（pull）的方式传送数据，稍后就会看到这样的例子。

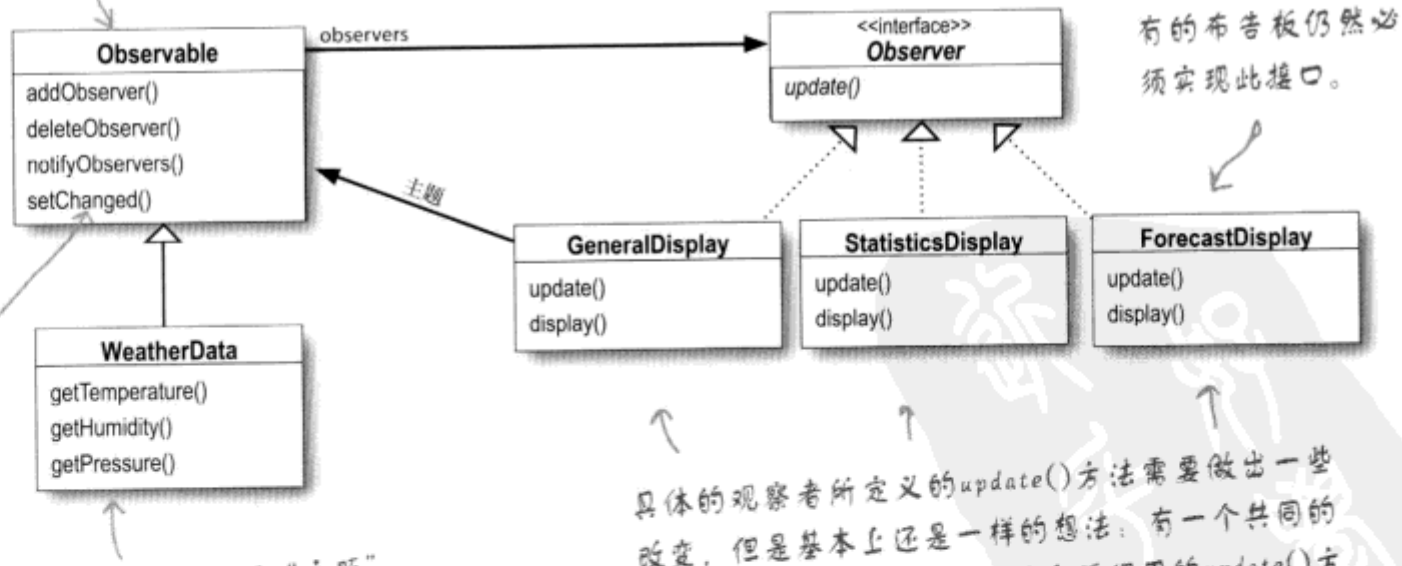
为了更了解java.uitl.Observer和java.util.Observable，看看下面的图，这是修改后的气象站OO设计。

这个看起来应该很熟悉，其实，它和之前的类图完全一样。

为了精简起见，我们在此没有把DisplayElement接口绘制出来，但是所有的布告板仍然必须实现此接口。

Observable类追踪所有的观察者，并通知他们。

Observable是一个“类”，而不是一个接口，所以WeatherData扩展了Observable主题。



这个方法你可能会觉得陌生，我们等一下就会说明……

这就是我们以前所称的“主题”（Subject），从今以后也可以改称为“可观察者”（Observable）。我们不需要在此提供register()、remove()和notifyObservers()方法，因为我们已经从超类继承了这些行为。

具体的观察者所定义的update()方法需要做出一些改变，但是基本上还是一样的想法：有一个共同的Observer接口，提供了一个被主题调用的update()方法。

Java内置的观察者模式如何运作

Java内置的观察者模式运作方式，和我们在气象站中的实现类似，但有一些小差异。最明显的差异是WeatherData（也就是我们的主题）现在扩展自Observable类，并继承到一些增加、删除、通知观察者的方法（以及其他的方法）。Java版本的用法如下：

如何把对象变成观察者……

如同以前一样，实现观察者接口（java.uitl.Observer），然后调用任何Observable对象的addObserver()方法。不想再当观察者时，调用deleteObserver()方法就可以了。

可观察者要如何送出通知……

首先，你需要利用扩展java.util.Observable接口产生“可观察者”类，然后，需要两个步骤：

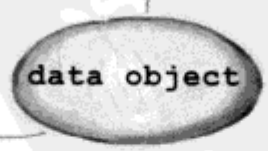
1. 先调用setChanged()方法，标记状态已经改变的事实。
2. 然后调用两种notifyObservers()方法中的一个：

```
notifyObservers()   或   notifyObservers(Object arg)
```

当通知时，此版本可以传送任何的数据对象给每一个观察者。

观察者如何接收通知……

同以前一样，观察者实现了更新的方法，但是方法的签名不太一样：




```
update(Observable o, Object arg)
```

主题本身当作第一个变量，好让观察者知道是哪个主题通知它的。

这正是传入*notifyObservers()*的数据对象。如果没有说明则为空。

如果你想“推”（push）数据给观察者，你可以把数据当作数据对象传送给notifyObservers(arg)方法。否则，观察者就必须从可观察者对象中“拉”（pull）数据。如何拉数据？我们再做一遍气象站，你很快就会看到。

Observable类的伪代码

setChanged()方法用来标记状态已经改变的事实，好让notifyObservers()知道当它被调用时应该更新观察者。如果调用notifyObservers()之前没有先调用setChanged()，观察者就“不会”被通知。让我们看看Observable 内部，以了解这一切：


```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
         call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

setChanged()方法把changed标志设为true。

notifyObservers() 只会在changed标为“true”时通知观察者。

在通知观察者之后，把changed标志设回false。

这样做有其必要性。setChanged()方法可以让你在更新观察者时，有更多的弹性，你可以更适当地通知观察者。比方说，如果没有setChanged()方法，我们的气象站测量是如此敏锐，以致于温度计读数每十分之一度就会更新，这会造成WeatherData对象持续不断地通知观察者，我们并不希望看到这样的事情发生。如果我们希望半度以上才更新，就可以在温度差距到达半度时，调用setChanged()，进行有效的更新。

你也许不会经常用到此功能，但是把这样的功能准备好，当需要时马上就可以使用。总之，你需要调用setChanged()，以便通知开始运转。如果此功能在某些地方对你有帮助，你可能也需要clearChanged()方法，将changed状态设置回false。另外也有一个hasChanged()方法，告诉你changed标志的当前状态。

利用内置的支持重做气象站

首先，把WeatherData改成使用java.util.Observable

❶ 记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在继承Observable。

❸ 我们不再需要追踪观察者了，也不需要管理注册与删除（让超类代劳即可）。所以我们把注册、添加、通知的相关代码删除。

```
import java.util.Observable;
import java.util.Observer;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); ★
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

❹ 我们的构造器不再需要为了记住观察者们而建立数据结构了。

★ 注意：我们没有调用notifyObservers()传送数据对象，这表示我们采用的做法是“拉”。

❺ 在调用notifyObservers()之前，要先调用setChanged()来指示状态已经改变。

❻ 这些并不是新方法，只是因为我们要使用“拉”的做法，所以才提醒你有这些方法。察者会利用这些方法取得WeatherData对象的状态。

现在，让我们重做CurrentConditionsDisplay

❶ 再说一遍，记得要导入（import）正确的Observer/Observable。

❷ 我们现在正在实现java.util.Observer接口。

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

❸ 现在构造器需要一Observable当参数，并将CurrentCondi— tionsDisplay对象登记成为观察者。

❹ 改变update()方法，增加Observable和数据对象作为参数。

❺ 在 update()中，先确定可观察者属于WeatherData类型，然后利用 getter方法获取温度和湿度测量值，最后调用display()。

练习

代码帖

ForecastDisplay类的代码小纸片在冰箱上被弄乱了。你能够重新排列它们，好恢复原来的样子吗？有些大括号掉到地上了，因为太小捡起来不易，所以如果你觉得需要大括号时，可以自行加上。

```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // 这里显示代码
}
```

```
lastPressure = currentPressure;
currentPressure = weatherData.getPressure();
```

```
private float currentPressure = 29.92f;
private float lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

运行新的代码

让我们运行新的代码，以确定它是对的……

```
File Edit Window Help TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```

嗯！你注意到差别了吗？再看一次……

你会看到相同的计算结果，但是奇怪的地方在于，文字输出的次序不一样。怎么会这样呢？在继续之前，请花一分钟的时间思考……

不要依赖于观察者被通知的次序

java.uitl.Observable实现了它的notifyObservers()方法，这导致了通知观察者的次序不同于我们先前的次序。谁也没有错，只是双方选择不同的方式实现罢了。

但是可以确定的是，如果我们的代码依赖这样的次序，就是错的。为什么呢？因为一旦观察者/可观察者的实现有所改变，通知次序就会改变，很可能就会产生错误的结果。这绝对不是我们所认为的松耦合。

java.util.Observable的黑暗面

是的，你注意到了！如同你所发现的，可观察者是一个“类”而不是一个“接口”，更糟的是，它甚至没有实现一个接口。不幸的是，java.util.Observable的实现有许多问题，限制了它的使用和复用。这并不是说它没有提供有用的功能，我们只是想提醒大家注意一些事实。

Observable是一个类

你已经从我们的原则中得知这不是一件好事，但是，这到底会造成什么问题呢？

首先，因为Observable是一个“类”，你必须设计一个类继承它。如果某类想同时具有Observable类和另一个超类的行为，就会陷入两难，毕竟Java不支持多重继承。这限制了Observable的复用潜力（而增加复用潜力不正是我们使用模式最原始的动机吗？）。

再者，因为没有Observable接口，所以你无法建立自己的实现，和Java内置的Observer API搭配使用，也无法将java.util的实现换成另一套做法的实现（比方说，

Observable将关键的方法保护起来

如果你看看Observable API，你会发现setChanged()方法被保护起来了（被定义成protected）。那又怎么样呢？这意味着：除非你继承自Observable，否则你无法创建Observable实例并组合到你自己的对象中来。这个设计违反了第二个设计原则：“多用组合，少用继承”。

做什么呢？

如果你能够扩展java.util.Observable，那么Observable“可能”可以符合你的需求。否则，你可能需要像本章开头的做法那样自己实现这一整套观察者模式。不管用哪一种方法，反正你都已经熟悉观察者模式了，应该都能善用它们。

在JDK中，还有哪些地方可以找到观察者模式

在JDK中，并非只有在java.util中才能找到观察者模式，其实在JavaBeans和Swing中，也都实现了观察者模式。现在，你已经具备足够的能力来自行探索这些API，但是我们还是在此稍微提一个简单的Swing例子，让你感受一下其中的乐趣。

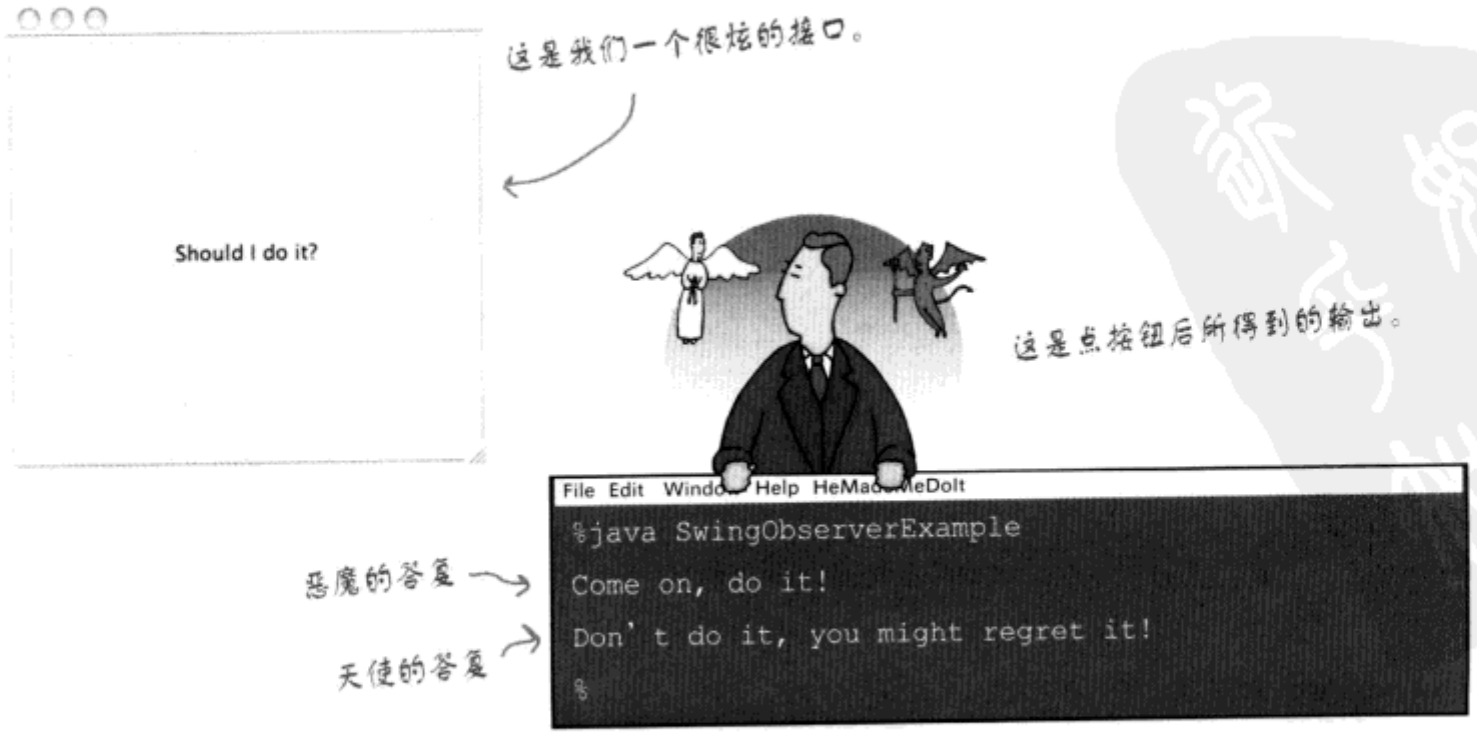
如果你对JavaBeans里的观察者模式感到好奇，可以查一下PropertyChangeListener接口。

背景介绍……

让我们看看一个简单的Swing API：JButton。如果你观察一下JButton的超类AbstractButton，会看到许多增加与删除倾听者（listener）的方法，这些方法可以让观察者感应到Swing组件的不同类型事件。比方说：ActionListener让你“倾听”可能发生在按钮上的动作，例如按下按钮。你可以在Swing API中找到许多不同类型的倾听者。

一个小的、改变生活的程序

我们的程序很简单，你有一个按钮，上面写着“Should I do it?”（我该做吗？）。当你按下按钮，倾听者（观察者）必须回答此问题。我们实现了两个倾听者，一个是天使（AngelListener），一个是恶魔（DevilListener）。程序的行为如下：



代码是这样的……

这个改变生活的程序需要的代码很短。我们只需要建立一个JButton对象，把它加到JFrame，然后设置好倾听者就行了。我们打算用内部类（inner class）作为倾听者类（这样的技巧在Swing中很常见）。如果你对内部类或Swing不熟悉，可以读一读《Head First Java》中的并于“获得GUI”的章节。

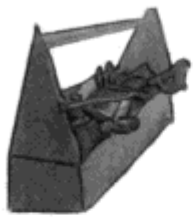
简单的Swing应用：建立一个JFrame，然后放上一个按钮。

```
public class SwingObserverExample {
    JFrame frame;

    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // 在这里设置frame属性
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

制造出两个倾听者（观察者），一个天使，一个恶魔。

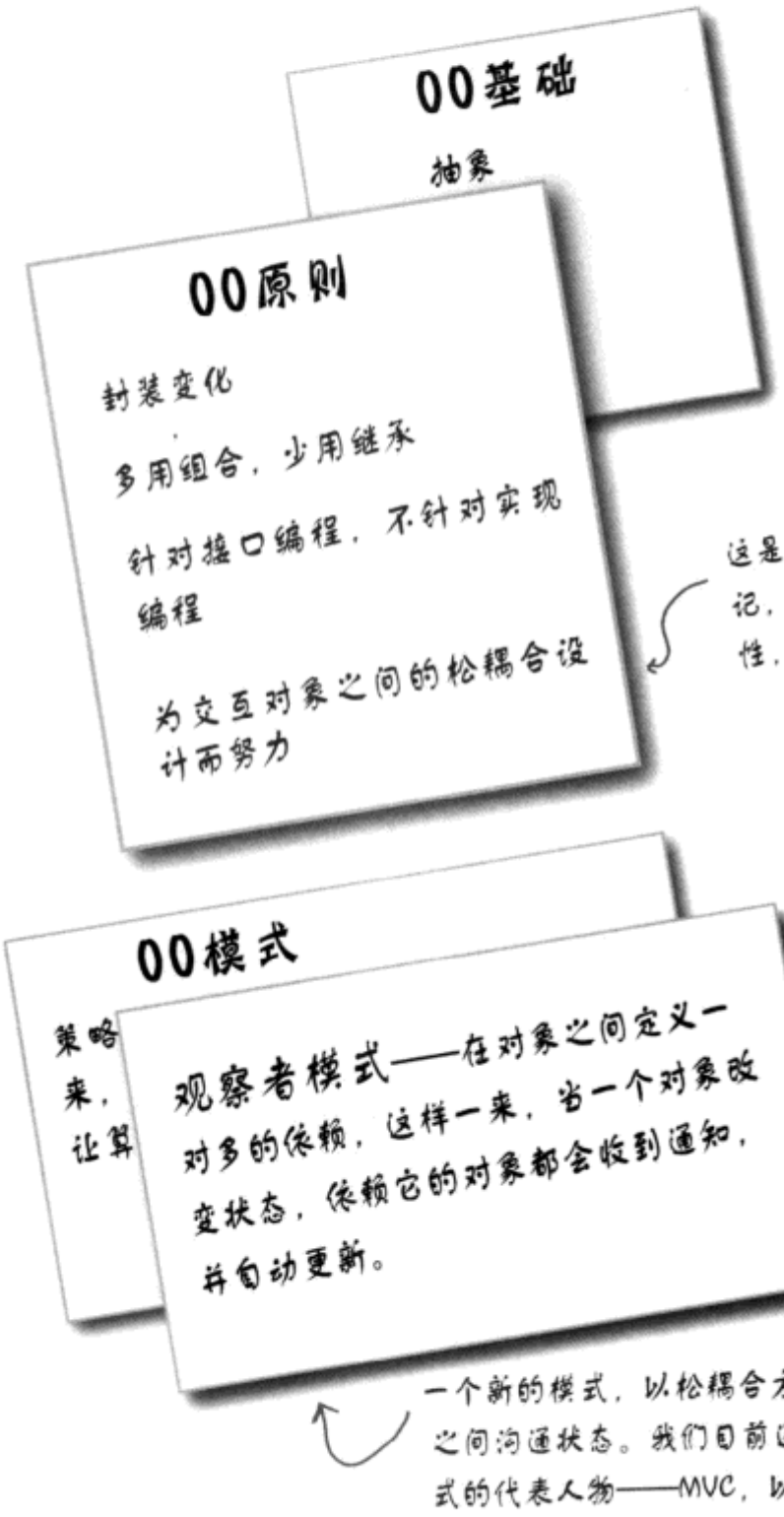
这是观察者的类定义，定义成内部类（你也可以不这么做）。

当主题（JButton）的状态改变时，在本例中，不是调用update()，而是调用actionPerformed()。



设计箱内的工具

欢迎来到第2章的结尾，你的对象工具箱内又多了一些东西……



要点

- 观察者模式定义了对象之间一对多的关系。
- 主题（也就是可观察者）用一个共同的接口来更新观察者
- 观察者和可观察者之间用松耦合方式结合（loosecoupling），可观察者不知道观察者的细节，只知道观察者实现了观察者接口。
- 使用此模式时，你可从被观察者处推（push）或拉（pull）数据（然而，推的方式被认为更“正确”）。
- 有多个观察者时，不可以依赖特定的通知次序。
- Java有多种观察者模式的实现，包括了通用的java.util.Observable。
- 要注意java.util.Observable实现上所带来的一些问题。
- 如果有必要的话，可以实现自己的Observable，这并不难，不要害怕。
- Swing大量使用观察者模式，许多GUI框架也是如此。
- 此模式也被应用在许多地方，例如：JavaBeans、RMI。