**观察者模式**

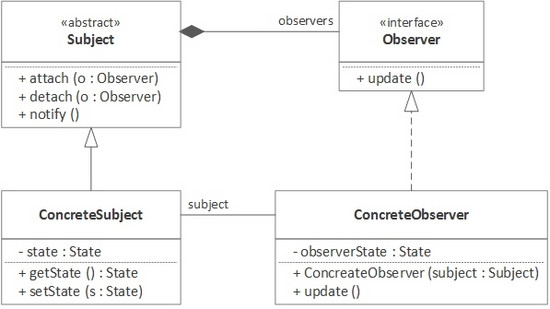
根据 GoF 提出的经典定义，观察者模式的主旨是：

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。

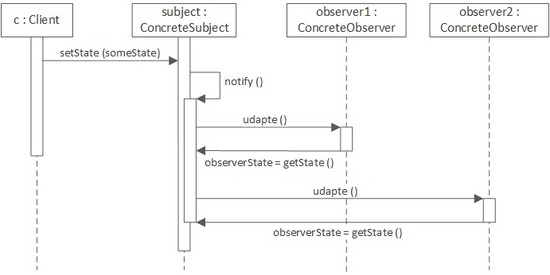
什么意思呢？很多软件应用中，对象之间的状态都是互相依赖的。例如，如果一个应用专注于数值数据加工，这个数据也许会通过图形用户界面（GUI）的表格或图表来展现或者两者同时使用，也就是说，当底层数据更新时，相应的 GUI 组件也要更新。问题的关键在于如何做到底层数据更新时 GUI 组件也随之更新，同时尽量减小 GUI 组件和底层数据的耦合度。

一种简单且不可扩展的解决方案是给管理这些底层数据的对象该表格和图像 GUI 组件的引用，使得对象可以在底层数据变化时能够通知 GUI 组件。显然，对于处理有更多 GUI 组件的复杂应用，这个简单的解决方案很快显示出其不足。例如，有20个 GUI 组件都依赖于底层数据，那么管理底层数据的对象就需要维护指向这20个组件的引用。随着依赖于相关数据的对象数量的增加，数据管理和对象之间的耦合度也变得难以控制。

另一个更好的解决方案是允许对象注册获取感兴趣数据更新的权限，当数据变化时，数据管理器就会通知这些对象。通俗地说就是，让感兴趣的数据对象告诉管理器：“当数据变化时请通知我”。此外，这些对象不仅可以注册获取更新通知，也可以取消注册，保证数据管理器在数据变化时不再通知该对象。在 GoF 的原始定义中，注册获取更新的对象叫作“观察者”（observer），对应的数据管理器叫作“目标”(Subject)，观察者感兴趣的数据叫作“目标状态”，注册过程叫“添加”（attach），撤销观察的过程叫“移除”（detach）。前文已经提到观察者模式又叫发布－订阅模式，可以理解为客户订阅关于目标的观察者，当目标状态更新时，目标把这些更新发布给订阅者（这种设计模式扩展为通用架构，称为发布——订阅架构）。这些概念可以用下面的类图表示：



具体观察者（ConcereteObserver）用来接收更新的状态变化，同时将指向具体主题（ConcereteSubject）的引用传递给它的构造函数。这为具体观察者提供了指向具体主题的引用，在状态变化时可由此获得更新。简单来说，具体观察者会被告知主题更新，同时用其构造函数中的引用来获取具体主题的状态，最后将这些检索状态对象存储在具体观察者的观察状态（observerState）属性下。这一过程如下面的序列图所示：



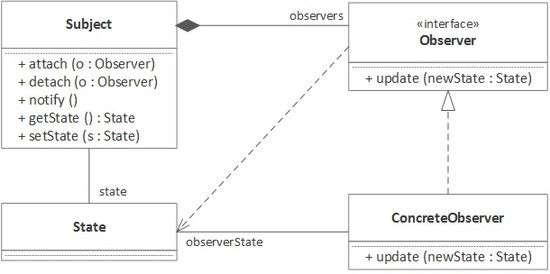
**经典模式的专业化**

尽管观察者模式是通用的，但也有很多专业化的模式，最常见是以下两种：

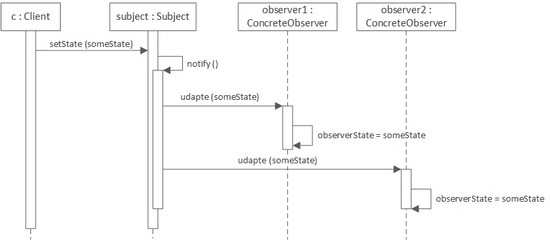
为State对象提供一个参数，传给观察者调用的Update方法。在经典模式下，当观察者被通知Subject状态发生变化后，会直接从Subject获得其更新后状态。这要求观察者保存指向获取状态的对象引用。这样就形成了一个循环引用，ConcreteSubject的引用指向其观察者列表，ConcreteObserver的引用指向能获得主题状态的ConcreteSubject。除了获得更新的状态，观察者和其注册监听的Subject间并没有联系，观察者关心的是State对象，而非Subject本身。也就是说，很多情况下都将ConcreteObserver和ConcreteSubject强行联系一起，相反，当ConcreteSubject调用Update函数时，将State对象传递给ConcreteObserver，二者就无需关联。ConcreteObserver和State对象之间关联减小了观察者和State之间的依赖程度（关联和依赖的更多区别请参见Martin Fowler's的文章）。

将Subject抽象类和ConcreteSubject合并到一个 singleSubject类中。多数情况下，Subject使用抽象类并不会提升程序的灵活性和可扩展性，因此，将这一抽象类和具体类合并简化了设计。

这两个专业化的模式组合后，其简化类图如下：



在这些专业化的模式中，静态类结构大大简化，类之间的相互作用也得以简化。此时的序列图如下：



专业化模式另一特点是删除了 ConcreteObserver 的成员变量 observerState。有时候具体观察者并不需要保存Subject的最新状态，而只需要监测状态更新时 Subject 的状态。例如，如果观察者将成员变量的值更新到标准输出上，就可以删除 observerState，这样一来就删除了ConcreteObserver和State类之间的关联。

更常见的命名规则

经典模式甚至是前文提到的专业化模式都用的是attach，detach和observer等术语，而Java实现中很多都是用的不同的词典，包括register，unregister，listener等。值得一提的是State是listener需要监测变化的所有对象的统称，状态对象的具体名称需要看观察者模式用到的场景。例如，在listener监听事件发生场景下的观察者模式，已注册的listener将会在事件发生时收到通知，此时的状态对象就是event，也就是事件是否发生

**就此程序分析：**

创建一个系统zoo，允许用户监听和撤销监听添加新对象animal的状态，另外再创建一个具体监听器，负责输出新增动物的name。

根据前面对观察者模式的学习知道实现这样的应用需要创建4个类，具体是：

Zoo类：即模式中的主题，负责存储动物园中的所有动物，并在新动物加入时通知所有已注册的监听器。

Animal类：代表动物对象。

AnimalAddedListener类：即观察者接口。

PrintNameAnimalAddedListener：具体的观察者类，负责输出新增动物的name。

首先我们创建一个Animal类，它是一个包含name成员变量、构造函数、getter和setter方法的简单Java对象，代码如下：

public class Animal {

private String name;

public Animal (String name) {

this.name = name;

}

public String getName () {

return this.name;

}

public void setName (String name) {

this.name = name;

}

}

用这个类代表动物对象，接下来就可以创建AnimalAddedListener接口了：

public interface AnimalAddedListener {

public void onAnimalAdded (Animal animal);

}

前面两个类很简单，就不再详细介绍，接下来创建Zoo类：

public class Zoo {

private List<Animal> animals = new ArrayList<>();

private List<AnimalAddedListener> listeners = new ArrayList<>();

public void addAnimal (Animal animal) {

// Add the animal to the list of animals

this.animals.add(animal);

// Notify the list of registered listeners

this.notifyAnimalAddedListeners(animal);

}

public void registerAnimalAddedListener (AnimalAddedListener listener) {

// Add the listener to the list of registered listeners

this.listeners.add(listener);

}

public void unregisterAnimalAddedListener (AnimalAddedListener listener) {

// Remove the listener from the list of the registered listeners

this.listeners.remove(listener);

}

protected void notifyAnimalAddedListeners (Animal animal) {

// Notify each of the listeners in the list of registered listeners

this.listeners.forEach(listener -> listener.updateAnimalAdded(animal));

}

}

这个类比前面两个都复杂，其包含两个list，一个用来存储动物园中所有动物，另一个用来存储所有的监听器，鉴于animals和listener集合存储的对象都很简单，本文选择了ArrayList来存储。存储监听器的具体数据结构要视问题而定，比如对于这里的动物园问题，如果监听器有优先级，那就应该选择其他的数据结构，或者重写监听器的register算法。

注册和移除的实现都是简单的委托方式：各个监听器作为参数从监听者的监听列表增加或者移除。notify函数的实现与观察者模式的标准格式稍微偏离，它包括输入参数：新增加的animal，这样一来notify函数就可以把新增加的animal引用传递给监听器了。用streams API的forEach函数遍历监听器，对每个监听器执行theonAnimalAdded函数。

在addAnimal函数中，新增的animal对象和监听器各自添加到对应list。如果不考虑通知过程的复杂性，这一逻辑应包含在方便调用的方法中，只需要传入指向新增animal对象的引用即可，这就是通知监听器的逻辑实现封装在notifyAnimalAddedListeners函数中的原因，这一点在addAnimal的实现中也提到过。

除了notify函数的逻辑问题，需要强调一下对notify函数可见性的争议问题。在经典的观察者模型中，如GoF在设计模式一书中第301页所说，notify函数是public型的，然而尽管在经典模式中用到，这并不意味着必须是public的。选择可见性应该基于应用，例如本文的动物园的例子，notify函数是protected类型，并不要求每个对象都可以发起一个注册观察者的通知，只需保证对象能从父类继承该功能即可。当然，也并非完全如此，需要弄清楚哪些类可以激活notify函数，然后再由此确定函数的可见性。

接下来需要实现PrintNameAnimalAddedListener类，这个类用System.out.println方法将新增动物的name输出，具体代码如下：

public class PrintNameAnimalAddedListener implements AnimalAddedListener {

@Override

public void updateAnimalAdded (Animal animal) {

// Print the name of the newly added animal

System.out.println("Added a new animal with name '" + animal.getName() + "'");

}

}

最后要实现驱动应用的主函数：

public class Main {

public static void main (String[] args) {

// Create the zoo to store animals

Zoo zoo = new Zoo();

// Register a listener to be notified when an animal is added

zoo.registerAnimalAddedListener(new PrintNameAnimalAddedListener());

// Add an animal notify the registered listeners

zoo.addAnimal(new Animal("Tiger"));

}

}

主函数只是简单的创建了一个zoo对象，注册了一个输出动物name的监听器，并新建了一个animal对象以触发已注册的监听器，最后的输出为：

Added a new animal with name 'Tiger'