# 观察者模式简介

在软件系统中对象并不是孤立存在的，一个对象行为的改变可能会导致一个或多个其他与之存在依赖关系的对象行为发生改变。观察者模式用于描述对象之间的依赖关系，为实现多个对象之间的联动提供了一种解决方案，它是一种使用频率非常高的设计模式。

## 概述

1. “红灯停，绿灯行”，在日常生活中交通信号灯管理着城市，指挥着日益拥挤城市的交通。当红灯亮起，来往的汽车将停止；而绿灯亮起，汽车可以继续前行。在这个过程中交通信号灯是汽车（更准确地说应该是汽车驾驶员）的观擦目标，而汽车是观察者。随着交通信号灯的变化，汽车的行为也随之发送变化，一盏交通信号灯可以指挥多辆汽车。在软件系统中有些对象之间也存在类似交通信号灯和汽车之间的关系，一个对象的状态或行为的变化将导致其他对象的状态或行为也发送改变，它们之间将产生联动，正所谓“触一而牵百发”。为了更好地描述对象之间的这种一对多（包括一对一）的联动，观察者模式应运而生，它定义了对象之间的一种一

对多的依赖关系，让一个对象的改变能够影响其他对象。

1. 观察者模式是使用频率较高的设计模式之一，它用于建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应作出反应。在观察者模式中发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以有多个观察者，而且这些观察者之间可以没有任何相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展。
2. 观察者模式（Observer Pattern）的定义如下：定义对象之间的一种一对多的依赖关系，使得每当一个对象状态发生改变时其相关依赖对象皆得到通知并被自动更新。
3. 观察者模式是一种对象行为型模式。观察者模式的别名有发布-订阅（Publish-Subscribe）模式、模型-视图（Model-View）模式、源-监听器（Source-Listener）模式、从属者（Dependents）模式。

## 结构

观察者模式结构中通常把偶偶观察目标和观察者两个继承层次结构，其结构图如下所示：



由图可知，观察者模式包含以下4个角色。

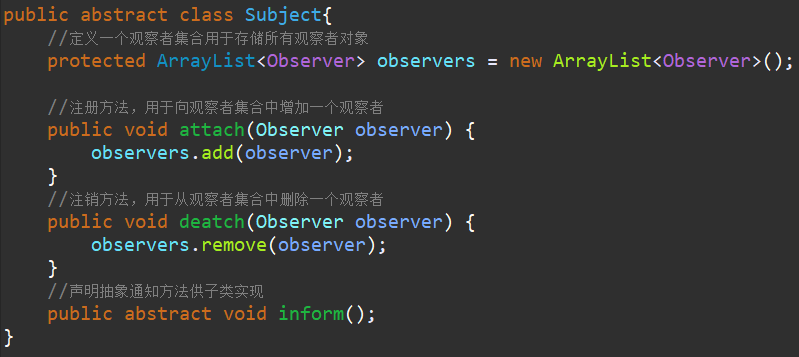
1. Subject（目标）：目标又称为主题，它是指被观察的对象。在目标中定义了一个观察者集合，一个观察目标可以接收任意数量的观察者来观察，它提供了一系列方法来增加和删除观察者对象，同时它定义了通知方法notify( )。目标类可以是接口、抽象类或具体类。
2. ConcreteSubject（具体目标）：具体目标是目标类的子类，它通常包含有经常发生改变的数据，当它的状态发生改变时将向它的各个观察者发出通知；同时它还实现了在目标类中定义的抽象业务逻辑方法（如果有）。如果无须扩展目标了，则具体目标类可以省略。
3. Observer（观察者）：观察者将对观察目标的改变作出反应，观察者一般定义为接口，该接口声明了更新数据的方法update( )，因此又称为抽象观察者。
4. ConcreteObserver（具体观察者）：在具体观察者中维护一个指向具体目标对象的引用，它存储具体观察者的有关状态，这些状态需要和具体目标的状态一致；它实现了在抽象观察者Observer中定义的update( )方法。通常在实现时可以调用具体目标类attach( )方法将自己添加到目标类的集合中或通过detach( )方法将自己从目标类的集合中删除。

# 实现

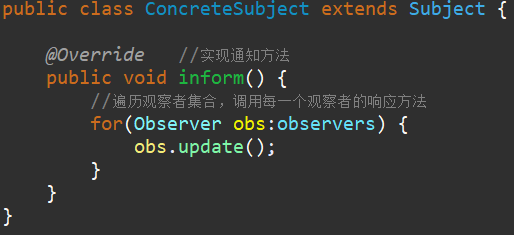
## 实现原理

观察者模式描述了如何建立对象之间的依赖关系，以及如何构造满足这种需求的系统。观察者模式包含观察目标和观察者两类对象，一个目标可以有任意数量的观察者，一旦观察目标的状态发生改变，所有的观察者都将得到通知。作为对这个通知的相应，每个观察者都将监视观察目标的状态，以使其状态与目标状态同步，这种交互也称为发布-订阅（Publis-Subscribe）。观察目标是通知的发布者，它发出的通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅并接受通过。

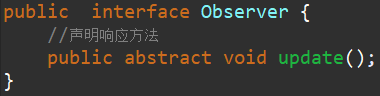
1. 定义抽象目标类Subject，典型代码如下：



1. 具体目标类ConcreteSubject是实现了抽象目标类Subject的一个具体子类，其典型代码如下：



1. 抽象观察者角色一般定义为一个接口，通常只声明一个update( )方法，为不同观察者的更新（响应）行为定义相同的接口，这个方法在其子类中实现，不同的观察者具有不同的响应方法。典型代码如下：



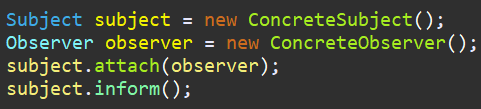
1. 在具体观察者ConcreteObserver中实现了update( )方法，其典型代码如下：



在有些更加复杂的情况下，具体观察者ConcreteObserver的update( )方法在执行时需要使用到具体目标类ConcreteSubject中的状态（属性），因此在ConcreteObserver与ConcreteSubject之间有时候还存在关联或依赖关系，在ConcreteObserver中定义一个ConcreteSubject实例，通过该实例获取存储在ConcreteSubject中的状态。如果ConcreteObserver的update( )方法不需要使用到ConcreteSubject中的状态属性，则可以对观察者模式的标准结构进行简化，在具体观察者对象ConcreteObserver和具体目标ConcreteSubject之间不须维持对象引用。

如果在具体层之间具有关联关系（观察者与观察者之间），系统的扩展性将受到一定的影响，增加新的具体目标类有时候需要修改原有观察者的代码，在一定程度上违反了开闭原则，但是如果原有观察者类无须关联新增的具体目标，则系统扩展性不受影响。

1. 在客户端代码中首先创建具体目标对象以及具体观察者对象，然后调用具体目标对象的attach( )方法，将这个观察者对象在目标对象中登记，也就是将它加入到目标对象的观察者集合中，代码片段如下：



客户端在调用目标对象的inform( )方法时将调用在其观察者集合中注册的观察 者对象的update( )方法。

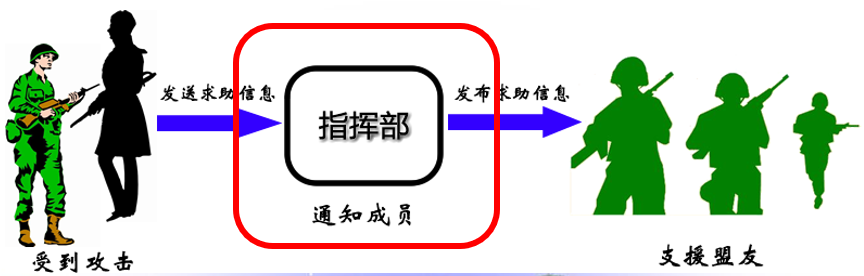
## 实例

实例说明：在某多人联机游戏中，多个玩家可以加入同一战队组成联盟，当战队中的某一成员受到敌人攻击时将给所有其他盟友发送通知，盟友收到通知后将作出响应。现使用观察者模式设计并实现该过程，以实现战队成员之间的联动。

通过分析不难发现在该系统中战队成员之间的联动过程可以简单描述如下：

**联盟成员受到攻击 --> 发送通知给盟友 --> 盟友作出回应**

如果按照上述思路来设计系统，一个战队联盟成员在受到攻击时需要通知它的每一位盟友，每个联盟成员都需要持有其他所有盟友的信息，这将导致系统开销较大，因此可以引入一个新角色，即指挥部（战队控制中心）来负责维护和管理每个战队中所有成员的信息，每一个联盟成员受到攻击时将向对应的指挥部发送求助信息，指挥部逐一通知每一位盟友，盟友再作出相应。如下图所示：

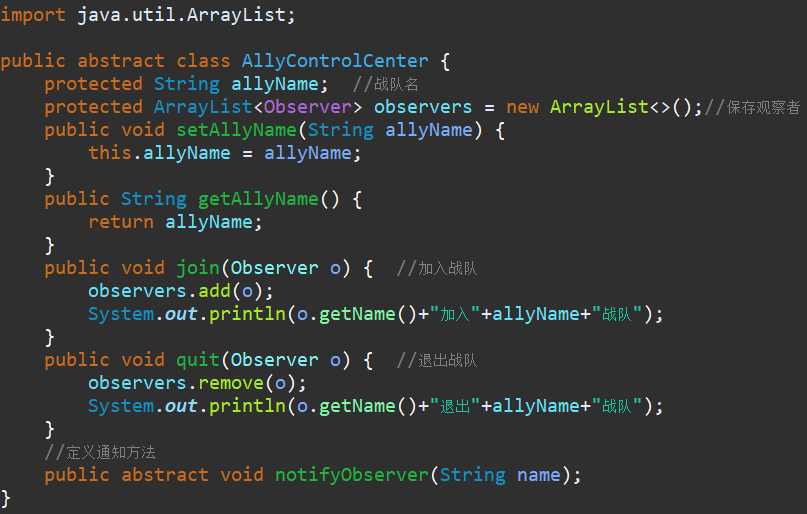


图中，受到攻击的联盟成员将与指挥部产生联动，指挥部还将与其他盟友产生联动。通过分析，本实例的结构图如下所示：

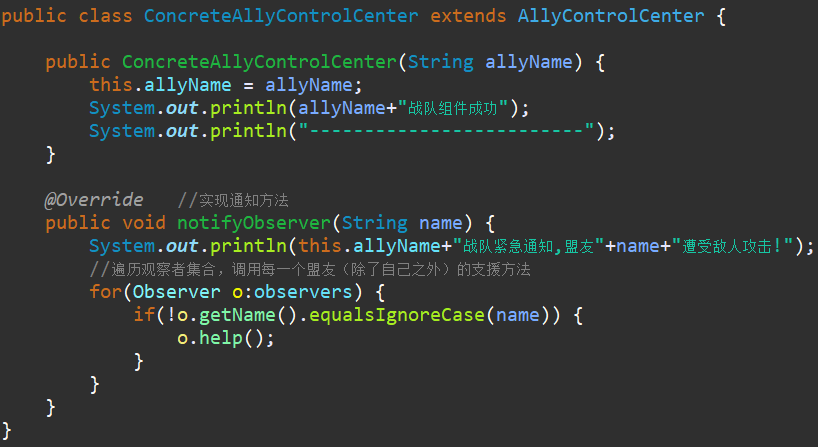


结构图中，AllyControlCenter充当抽象目标类，ConcreteAllyControlCenter充当具体目标类，Observer充当抽象观察者，Player充当具体观察者。

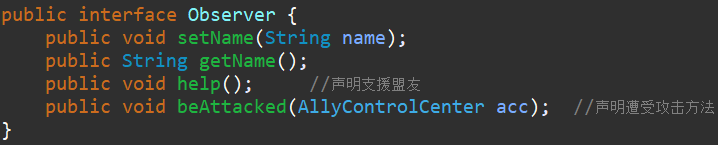
1. AllyControlCenter：指挥部（战队控制中心），充当抽象目标类。



1. ConcreteAllyControlCenter：具体指挥部类，充当具体目标类



1. Observer：抽象观察者类



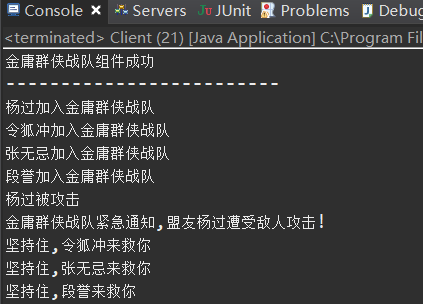
1. Player：战队成员类，充当具体观察者类



1. Client：客户端测试类



编译并运行程序，输出结果如下：



在本实例中实现了对象之间的联动，当一个游戏玩家Player对象的beAttacked( )方法被调用时将调用指挥部AllyControlCenter的nofityObserver( )方法进行处理，而在notifyObserver( )方法中又将调用其他Player对象的help( )方法。Player的beAttacked( )方法，AllyControlCenter的notifyObserver( )方法以及Player的help( )方法构成了一个联动触发链，执行顺序如下：

**Player.beAttacked( ) --> AllyControlCenter.notifyObserver( ) --> Player.help( )**

## JDK对观察者模式的支持

观察者模式在Java语言中占据非常重要的地位，在JDK的java.util包中提供了Observable类以及Observer接口，它们构成了JDK对观察者模式的支持。如下图所示：



1. **Observer接口**

在java.util.Observer接口中只声明了一个方法，它充当抽象观察者，其方法声明如下：

void update ( Observer o , Object arg ) ;

当观察目标的状态发生变化时该方法将会被调用，在Observer的子类中将实现update( )方法，即具体观察者可以根据需要具有不同的更新行为。当调用观察目标类Observable的notifyObservers( )方法时将执行观察者类中的update( )方法。

1. **Observable类**

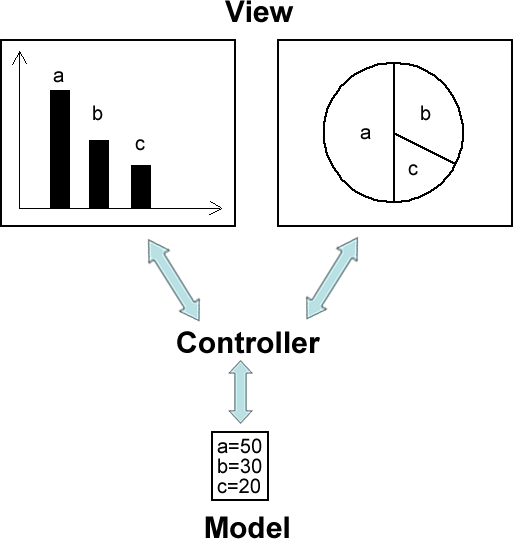
java.util.Observable类充当观察目标类，在Observable类中定义了一个Vector来存储观察者对象，它所包含的方法及说明如下：

|  |  |
| --- | --- |
| **方法名** | **方法描述** |
| Observable( ) | 构造方法，会实例化Vector向量 |
| addObserver(Observer o ) | 用于注册新的观察者对象到向量中 |
| deleteObserver(Observer o) | 用于删除向量中的某一个观察者对象 |
| notifyObservers( )和  notifyObservers(Object arg) | 通知方法，用于在方法内部循环调用向量中的每一个观察者的update( )方法 |
| deleteObservers( ) | 用于清空向量，即删除向量中所有的观察者对象 |
| setChanged( ) | 该方法被调用后会设置一个boolean类型的内部标记变量changed的值为true，表示观察目标对象的状态发生了改变 |
| clearChanged( ) | 用于将changed变量的值设为false，表示对象状态不再发生改变或者已经通知了所有的观察者对象，调用了它们的update( )方法 |
| hasChanged( ) | 用于判断对象状态是否改变 |
| countObservers( ) | 用于返回向量中观察者的数量 |

用户可以直接使用Observer接口和Observerable类作为观察者模式的抽象层，再自定义具体观察者类和具体目标类。通过使用JDK中的Observer接口和Observable类可以更加方便地在Java语言中应用观察者模式。

## 观察者模式与MVC

在当前流行的MVC（Model-View-Controller）架构中也应用了观察者模式，MVC是一种架构模式，它包含3个角色即模型（Model）、视图（View）和控制器（Controller）。其中，模型可对应观察者中的观察目标，而视图对应于观察者，控制器可充当两者之间的中介者。当模型层的数据发生改变时视图层（刷新后）将自动改变其显示内容。MVC结构示意图如下：



图中，模型层提供的数据时视图层所观察的对象，在视图层中包含了两个用于显示数据的图表对象，一个是柱状图，一个是饼状图，形同的数据拥有不同的图表显示方式，如果模型层的数据发生改变，两个图表对象将随之发送变化，这意味着图表对象依赖模型层提供的数据对象，因此数据对象的任何状态改变都应立即通知它们。同时，这两个图标之间相互独立，不存在任何联系，而且图表对象的个数没有任何限制，用户可以根据需要再增加新的图表对象，例如折线图。在增加新的图表对象时无须修改原有类库，满足开闭原则。

## 观察者模式与Java事件处理

Java事件处理模型采用了基于观察者模式的委派事件模型（Delegation Event Model，DEM），即一个Java组件所引发的事件并不由引发事件的对象自己来负责，而是委派给独立的事件处理对象负责。

# 优缺点及适用环境

观察者模式是一种使用频率非常高的设计模式，无论是移动应用、Web应用或者桌面应用，观察者模式几乎无处不在，它为实现对象之间的联动提供了一套完整的解决方案，凡是涉及一对一或者一对多的对象交互场景都可以使用观察者模式。观察者模式广泛应用于各种编程语言的GUI事件处理的实现，在基于事件的XML解析技术（例如SAX2）以及Web事件处理中也都使用了观察者模式。

## 优点

观察者模式的优点主要如下：

1. 可以实现表示层和数据逻辑层的分离，定义了稳定的消息更新传递机制，并抽象了更新接口，使得可以由各种各样不同的表示层充当具体观察者角色。
2. 在观察目标和观察者之间建立一个抽象的耦合。观察目标只需要维持一个抽象观察者的集合，无须了解其具体观察者。由于观察目标和观察者没有紧密地耦合在一起，因此他们可以属于不同的抽象画层次。
3. 支持广播通信，观察目标会向所有已注册的观察者对象发送通知，简化了一对多系统设计的难度。
4. 符合开闭原则，增加新的具体观察类无须修改原有系统代码，在具体观察者与观察目标之间不存在关联关系的情况下增加新的观察目标也很方便。

## 缺点

观察者模式的缺点主要如下：

1. 如果一个观察目标对象有很多直接和间接观察者，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存在循环依赖，观察目标会触发它们之间进行循环调用，可能会导致系统崩溃。
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送了变化。

## 适用环境

在以下情况下可以考虑适用观察者模式：

1. 一个抽象模型由两个方面，其中一个方面依赖于另一个方面，将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用。
2. 一个对象的改变将导致一个或多个其他对象也发生改变，而并不知道具体有多少对象将发生改变，也不知道这些对象是谁。
3. 需要在系统中创建一个触发链，A对象的行为将影响到B对象，B对象的行为将影响到C对象……，可以使用观察者模式创建一种链式触发机制。