观察者模式

1. GitHub地址

<https://github.com/Wang-Jun-Chao/Java-Design-Pattern/tree/master/14-%E8%A7%82%E5%AF%9F%E8%80%85%E6%A8%A1%E5%BC%8F>

1. 代码介绍

作者学习设计模式时做的各个设计模式的Demo，这个只是其中的观察者模式。在GitHub上查询时，看到作者的一个仓库，顺便看了作者的各个仓库，各个知识板块的Demo都很多，总结的也很好。

1. 类图



1. 观察者模式的结构

本观察者模式中主要包含三个角色：

* 被观察者：从类图中可以看到，类中有一个用来存放观察者对象的List容器，这个List容器是被观察者类的核心，另外还有四个方法：attach方法是向这个容器中添加观察者对象；setState是设置观察者对象；getState是返回观察者对象;notify方法是依次调用观察者对象的对应方法。这个角色可以是接口，也可以是抽象类或者具体的类，因为很多情况下会与其他的模式混用，所以使用抽象类的情况比较多。
* 观察者：观察者角色一般是一个接口，它只有一个update方法，在被观察者状态发生变化时，这个方法就会被触发调用。
* 具体的观察者：观察者接口的具体实现，在这个角色中，将定义被观察者对象状态发生变化时所要处理的逻辑。

1. 主要代码
2. 观察者

**package** Qiyanhui;

//观察者

**public** **abstract** **class** Observer {

**protected** Subject subject;

**public** **abstract** **void** update();

}

1. 被观察者

**package** Qiyanhui;

**import** java.util.ArrayList;

**import** java.util.List;

//被观察者

**public** **class** Subject {

**private** List<Observer> observers = **new** ArrayList<Observer>();

**private** **int** state;

**public** **int** getState() {

**return** state;

}

**public** **void** setState(**int** state) {

**this**.state = state;

notifyAllObservers();

}

**public** **void** attach(Observer observer){

observers.add(observer);

}

**public** **void** notifyAllObservers(){

**for** (Observer observer : observers) {

observer.update();

}

}

}

1. 具体观察者

\*yObserver.java

**package** Qiyanhui;

//具体的观察者

**public** **class** BinaryObserver **extends** Observer {

**public** BinaryObserver(Subject subject){

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println( "Binary String: " + Integer.*toBinaryString*( subject.getState() ) );

}

}

\*exaObserver.java

**package** Qiyanhui;

//具体的观察者

**public** **class** HexaObserver **extends** Observer {

**public** HexaObserver(Subject subject) {

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println("Hex String: " + Integer.*toHexString*(subject.getState()).toUpperCase());

}

}

\*ctalObserver.java

**package** Qiyanhui;

//具体的观察者

**public** **class** OctalObserver **extends** Observer {

**public** OctalObserver(Subject subject){

**this**.subject = subject;

**this**.subject.attach(**this**);

}

@Override

**public** **void** update() {

System.***out***.println( "Octal String: " + Integer.*toOctalString*( subject.getState() ) );

}

}

4)主函数

**package** Qiyanhui;

**import** Qiyanhui.BinaryObserver;

**import** Qiyanhui.HexaObserver;

**import** Qiyanhui.OctalObserver;

**import** Qiyanhui.Subject;

//主函数

**public** **class** ObserverPatternDemo {

**public** **static** **void** main(String[] args) {

Subject subject = **new** Subject();

**new** HexaObserver(subject);

**new** OctalObserver(subject);

**new** BinaryObserver(subject);

System.***out***.println("First state change: 15");

subject.setState(15);

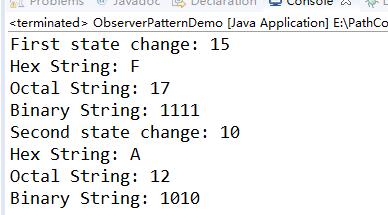
System.***out***.println("Second state change: 10");

subject.setState(10);

}

}

5）结果



通过运行结果可以看到，我们只调用了Subject的方法，但同时两个观察者的相关方法都被同时调用了。

1. 总结

观察式的优缺点

观察者与被观察者之间是属于轻度的关联关系，并且是抽象耦合的，这样，对于两者来说都比较容易进行扩展。

观察者模式是一种常用的触发机制，它形成一条触发链，依次对各个观察者的方法进行处理。但同时，这也算是观察者模式一个缺点，由于是链式触发，当观察者比较多的时候，性能问题是比较令人担忧的。并且，在链式结构中，比较容易出现循环引用的错误，造成系统假死。