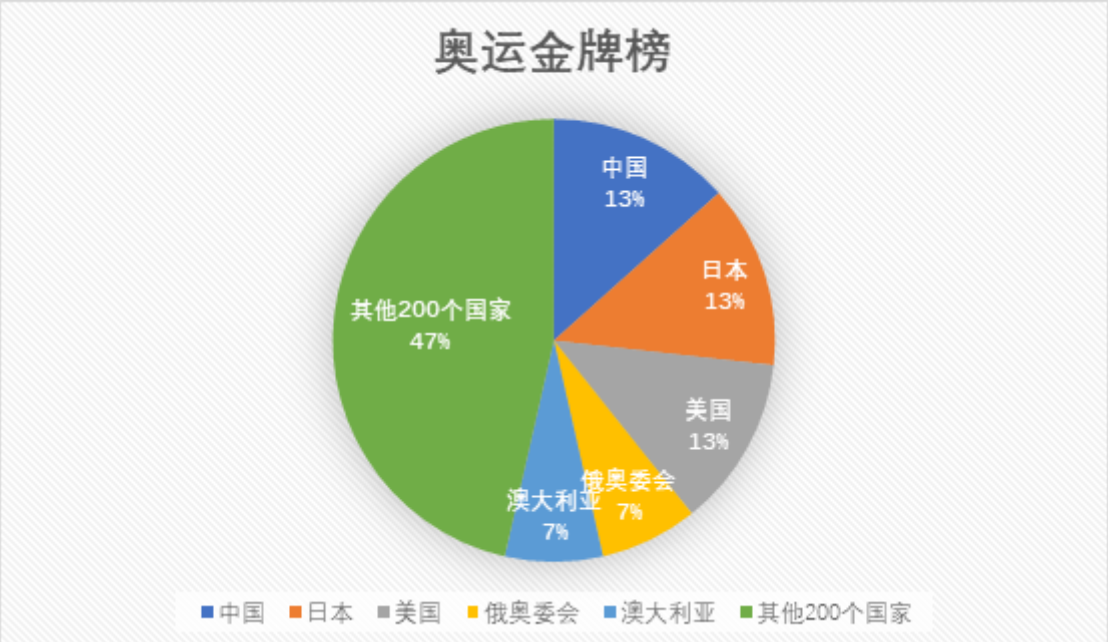
文章已收录我的仓库：Java学习笔记与免费书籍分享

设计意图

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。

在实际设计开发中，我们通常会降低类与类之间的耦合度，这样可能会产生一个副作用：由于类与类被分割，我们难以维护类之间的一致性。

举一个常见的例子，我们对用户显示数学饼状图是需要数据支撑的，例如下面这张东京奥运会金牌榜:



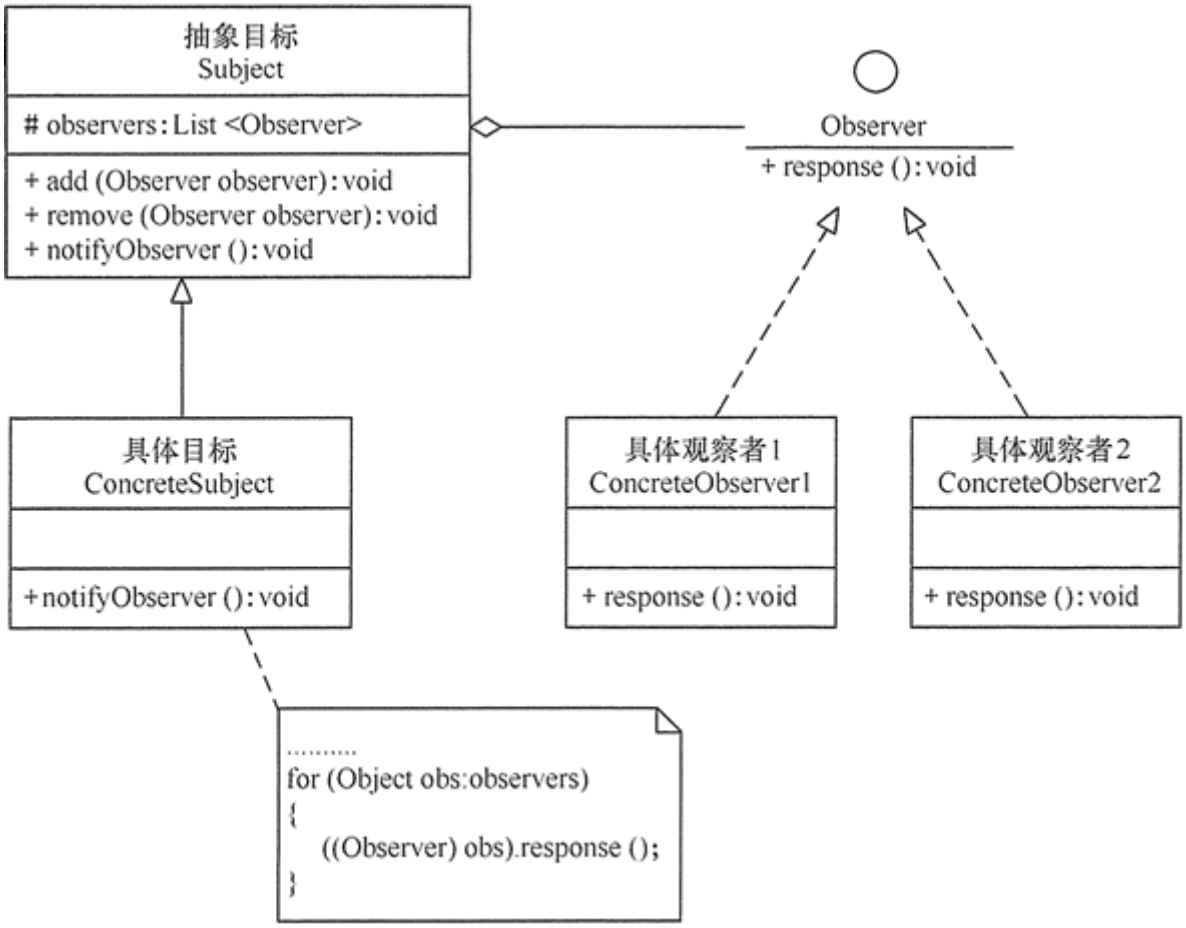
在开发中，这张图表分为两个部分，一个是视图部分，也就是以饼状图呈现出的样子，一个是数据部分，即各国的金牌数量，由于我们将数据与视图抽离，因此一旦数据部分更新，视图部分得不到最新的数据，难以维持一致性，这个时候我们需要一个时刻关注数据变化的观察者，一旦观察者感知到数据变化则立即更新视图，我们可以让视图本身作为一个观察者，但这样设计是不好的，视图类应当做好设计视图的事而无需插手其他工作，更

好的办法是单独分离出一个观察者类以维护两个类之间的一致性，这就是观察者模式的设计意图。

在实际例子中，这种模式应用非常广泛，例如一旦小说更新将会自动订阅，一旦会员过期将会自动续费，MVC三层模式中的控制器就会观察视图并实时更新模型部分...观察者模式是应用最广泛的模式之一。

设计

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。



观察者模式的主要角色如下。

1. 抽象主题（Subject）角色：也叫抽象目标类或目标接口类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。
2. 具体主题（Concrete Subject）（被观察目标）角色：也叫具体目标类，它是被观察的目标，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。
3. 观察者接口（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

4. 具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

设计接口(抽象)是常规的设计思想：定义一个接口由子类实现。这样做利于后续扩展，但如果确定只有一个被观察对象，则没有必要设计接口(抽象)类。

常见的设计是：观察者到被观察目标中注册登记，告诉它有一个观察者正在观察他，如有变化请通知，随后观察目标发生变化，则通知所有注册登录过的观察者并告诉自己的身份(观察者可能观察多个目标，某些时候它必须知道具体是那个目标发生了变化)，随后观察者更新相应数据。

代码示例

我们考虑上述数据与视图之间的例子，这里假设我们的视图接收谷歌数据源与百度数据源：

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

//视图类
class View {
    //通过复杂转换将数据可视化，这里简单的打印
    public void show(Object data) {
        System.out.println(data);
    }
}

//定义抽象类 数据源类
abstract class DataSource {
    //相关的源数据
    protected String data = "";

    //存储已经注册过的观察者
    protected List<Observer> observers = new ArrayList<>();

    //获取该数据
    public String getData() {
        return data;
    }

    //观察者到这里注册，被观察者保存观察者信息
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    //移除更改观察者就不写了

    //接口方法，更新数据，由目标类通知观察者
    abstract protected void updateData(String newData);
```

```
    //接口方法，通知观察者，由子类采用不同的方法实现
    abstract public void notifyObserver();
}

//数据源类的具体实现之一，百度数据源类
class BaiduDataSource extends DataSource {
    @Override
    protected void updateData(String newData) {
        //如果数据发生变化，则更新数据并通知观察者
        if (!newData.equals(data)) {
            //这一步是必须的，在通知观察者前一定要完成变化
            //这就好比你明天才出发可你却告诉你的好朋友今天走，你的好朋友来接你没看
到你，友情破碎
            //必须要保持状态的一致性
            data = newData;
            notifyObserver();
        }
    }

    @Override
    public void notifyObserver() {
        //广播消息，并告知观察者自己是谁
        for (var observer : observers) {
            observer.update(this, data);
        }
    }
}

//数据源类的具体实现之一，谷歌数据源类
class GoogleDataSource extends DataSource {
    @Override
    protected void updateData(String newData) {
        //如果数据发生变化，则更新数据并通知观察者
        if (!newData.equals(data)) {
            //必须要保持状态的一致性
            data = newData;
            notifyObserver();
        }
    }

    @Override
    public void notifyObserver() {
        //广播消息，并告知观察者自己是谁
        for (var observer : observers) {
            observer.update(this, data);
        }
    }
}

//观察者接口
interface Observer {
    /**
     * 更新操作
     * @param ds    观察的具体数据源
     * @param data  更新的数据
```

```
     */
    void update(DataSource ds, String data);
}

//观察者A
class ObserverA implements Observer {
    //由view示例委托观察数据源
    private View view;

    public ObserverA(View view) {
        this.view = view;
    }

    @Override
    public void update(DataSource ds, String data) {
        System.out.println("观察到" + ds.getClass().getSimpleName() + "发
生变化，更新视图");
        //更新视图View
        view.show(data);
    }
}


//测试类
public class Test {
    public static void main(String[] args) {
        //定义视图类
        View view = new View();
        view.show("初始状态");

        System.out.println();

        //定义与view相关数据源
        DataSource bds = new BaiduDataSource();//百度数据源
        DataSource gds = new GoogleDataSource();//谷歌数据源

        //为view添加观察数据源的观察者
        Observer observer = new ObserverA(view);

        //观察者需要到到数据源类中注册
        bds.addObserver(observer);
        gds.addObserver(observer);

        //手动更新数据
        bds.updateData("这是百度新数据--" + new Date());
        System.out.println();
        gds.updateData("这是谷歌新数据--" + new Date());
    }
}
//输出
/*
初始状态

观察到BaiduDataSource发生变化，更新视图
这是百度新数据--Fri Jul 30 10:43:55 CST 2021
```

```
观察到GoogleDataSource发生变化，更新视图
这是谷歌新数据--Fri Jul 30 10:43:55 CST 2021
*/
```

讨论与优化

我们围绕上面的代码示例来讨论。

- 在发送给通知给观察者前，维护自身状态一致性是很重要的，在上面的代码中我们必须要先更新数据在发送通知，就像例子说的，你明明要等到明天才出发，可你却通知你的好朋友马上就走走，这样总会引起一些不好的结果。
- 上述代码只设置了一个观察者，实际中可能有多个观察者，可是观察者之间却又互相不知道彼此的存在，这就可能会造成重复更新的甚至更严重的问题，我们必须要好好设置观察者，以保证它们在功能上不具有重复性。事实上，当观察者越来越多时，代码会变得更加难以扩展维护。
- 上述代码中我们让观察者保存了View的实例，实际的更新还是由该实例自己来完成，这是符合观察者模式的定义的。但实际上，常常会由观察者自身来更新相关数据。
- 观察者可能观察多个目标，因此当目标通知观察者时应该告知观察者它自己是谁，以便观察者做出相应操作，实现的办法就是目标将自身传入观察者方法的参数中。这样是符合常理的——观察者正在观察5岁、6岁、7岁的人比赛跑步，一旦出现达到终点则观察者颁发奖状，不同年龄的人评奖原则也是不同的，所以观察者必须知道到底是谁完成比赛。
- 上述代码中一旦有变化则通知所有的观察者——尽管有些观察者对这些消息并不感兴趣，当观察者较多时，效率是很低的，我们应该只通知那些对该变化感兴趣的观察者们，我们可以定义一个Aspect类表示该变化的特点，可以采用哈希表保存观察者：

  ```
  Map<Aspect, List<Observer>> map = new HashMap<>();
  ```

  观察者注册时，必须表面自己对那些方面的变化感兴趣：

  ```
  public void addObserver(Aspect aspect, Observer observer) {
          map.put(aspect, observer);
  }
  ```

其他

在 Java 中，通过 java.util.Observable 类和 java.util.Observer 接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。我们来分析主要的类与它们的功

能：

1. Observable类

Observable 类是抽象目标类，它有一个 Vector 向量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法。

1. void addObserver(Observer o) 方法：用于将新的观察者对象添加到向量中。
2. void notifyObservers(Object arg) 方法：调用向量中的所有观察者对象的 update() 方法，通知它们数据发生改变。通常越晚加入向量的观察者越先得到通知。
3. void setChange() 方法：用来设置一个 boolean 类型的内部标志位，注明目标对象发生了变化。当它为真时，notifyObservers() 才会通知观察者。

2. Observer 接口

Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用 void update(Observable o,Object arg) 方法，进行相应的工作。

事实上这一套类已经太老了，效率比较低，不建议使用。

总结

观察者模式是一种对象行为型模式，其主要优点如下。

1. 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。符合依赖倒置原则。
2. 目标与观察者之间建立了一套触发机制。

它的主要缺点如下。

1. 目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。
2. 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率，并且可能会导致意外的更新。