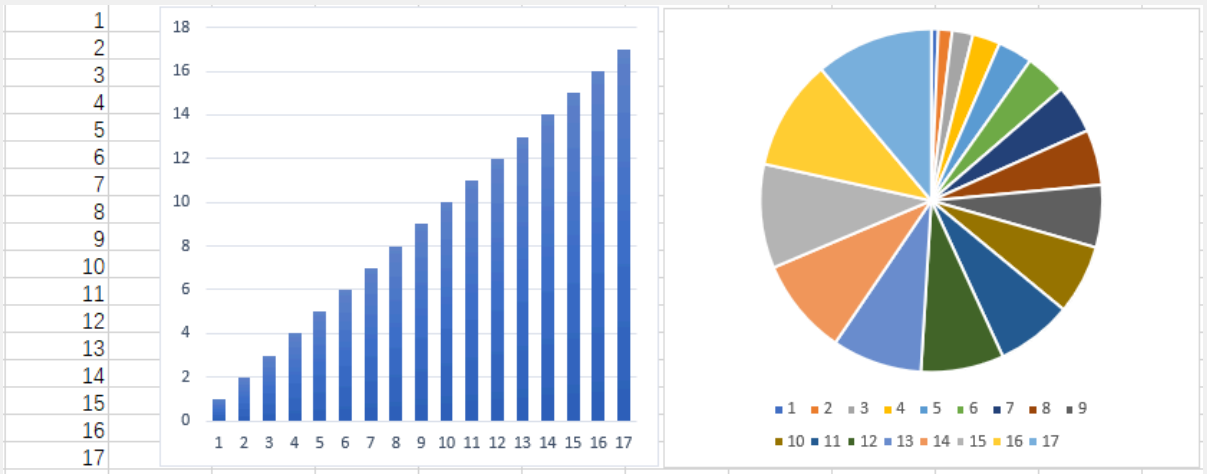
第4章 观察者模式

提出问题

> 设计一套图表工具（例如饼图和柱状图）用于展示选中的数据，当选中的数据状态发生变化时，自动更新图表显示。



模式名称

观察者模式（Observer）或从属者模式（Dependents）或订阅模式（Publish-Subscribe）

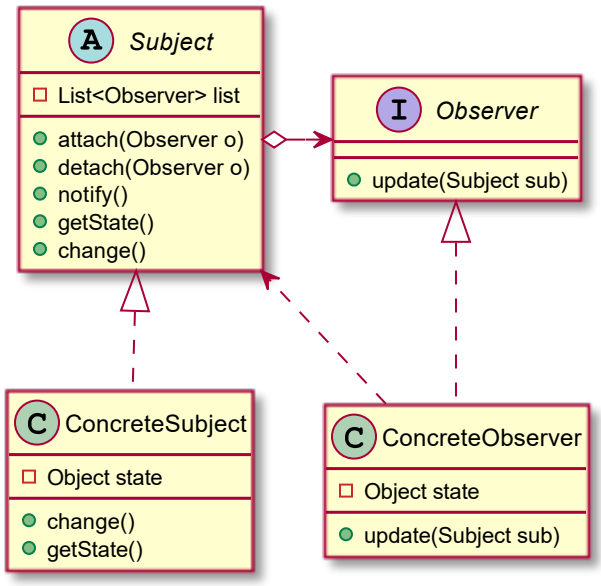
设计意图

观察者模式定义对象间的一对多依赖关系，当一个对象状态改变时，所有依赖者将被通知并自动更新。

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

设计结构

类图



参与者

- 抽象主题（Subject）：被观察对象，定义了增加、删除、通知观察者的方法。
- 具体主题（ConcreteSubject）：具体被观察者，当其内部状态变化时，会通知已注册的观察者。
- 抽象观察者（Observer）：定义响应通知的更新方法。
- 具体观察者（ConcreteObserver）：当得到状态更新的通知时，会自动做出响应。

实现

```java
// 定义观察者接口，通过主题对象获取更新状态
public interface Observer {
    public void update(Subject sub);
}
// 定义抽象主题类，设计观察者管理、状态获取、状态变化、通知等方法
public abstract class Subject {
    private List<Observer> list = new ArrayList<Observer>();
    public void attach(Observer o) {
        list.add(o);
    }
    public void detach(Observer o) {
        if(list.contains(o))
            list.remove(o);
    }
    public List<Observer> getObservers() {
        return list;
    }
    public abstract void notifyObservers();
    public abstract String getState();
    public abstract void change();
}
// 定义具体主题类，方法实现
public class ConcreteSubject extends Subject {
    private String state;
    @Override
    public void notifyObservers() {
        for(Observer o:getObservers()) {
            o.update(this);
        }
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
    @Override
    public void change() {
        this.state = new Date().toString();
        notifyObservers();
    }
}
// 定义具体观察者，状态获取与更新
public class ConcreteObserver implements Observer {
    private String state;
```

```
    @Override
    public void update(Subject sub) {
        setState(sub.getState());
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    public String toString() {
        return this.hashCode() + "当前状态: " + state;
    }
}
// 客户端类测试
public class Client {
    public static void main(String[] args) throws InterruptedException {
        Observer o1 = new ConcreteObserver();
        Observer o2 = new ConcreteObserver();
        Subject sub = new ConcreteSubject();
        sub.attach(o1);
        sub.attach(o2);
        while(true){
            Thread.sleep(1000);
            sub.change();
            System.out.println(o1.toString());
            System.out.println(o2.toString());
        }
    }
}
```

运行结果:

```
787604730当前状态: Sat May 15 23:12:01 CST 2021
812265671当前状态: Sat May 15 23:12:01 CST 2021
787604730当前状态: Sat May 15 23:12:02 CST 2021
812265671当前状态: Sat May 15 23:12:02 CST 2021
787604730当前状态: Sat May 15 23:12:03 CST 2021
812265671当前状态: Sat May 15 23:12:03 CST 2021
787604730当前状态: Sat May 15 23:12:04 CST 2021
812265671当前状态: Sat May 15 23:12:04 CST 2021
```

解决问题

设计一个主题类`DataEditor`提供数据源并支持数据编辑功能：

```
public class DataEditor extends VBox{
    // 数据列表
    private List<Number> list = new ArrayList<Number>();
    // 图表对象列表，作为观察者
    private List<DataViewer> viewers = new ArrayList<DataViewer>();
    public DataEditor(int n) {
        for(int i = 0; i < n; i++) {
            TextField tf = new TextField();
            tf.setText("0.0");
            list.add(0.0);
            this.getChildren().add(tf);
            // 增加数据变化侦听，当数据变化时通知观察者
            tf.textProperty().addListener(new ChangeListener<String>() {
                @Override
                public void changed(ObservableValue<? extends String> arg0, String arg1, String ar
                    notifyViewers();
                }
            });
        }
    }
    // 图表的绑定和脱离
    public void attach(DataViewer dv) {
        viewers.add(dv);
    }
    public void detach(DataViewer dv) {
        viewers.remove(dv);
    }
    // 通知图表更新
    public void notifyViewers(){
        for(int i = 0; i < list.size(); i++) {
            TextField tf = (TextField) this.getChildren().get(i);
            list.set(i, Double.parseDouble(tf.getText()));
        }
        for(DataViewer dv:viewers) {
            dv.update(this);
        }
    }
    // 获取数据
    public List<Number> getData() {
        return list;
    }
}
```

设计观察者：

```
// 观察者接口，定义更新方法
public interface DataViewer {
    public void update(DataEditor de);
}
// 具体的观察者柱状图，对更新方法进行实现，绘制柱状图
public class BarChart extends Canvas implements DataViewer{
    public BarChart(double w, double h) {
        this.setWidth(w);
        this.setHeight(h);
    }
    @Override
    public void update(DataEditor de) {
        List<Number> numbers = de.getData();
        double sum = 0;
        for(Number n:numbers) {
            sum += n.doubleValue();
        }
        GraphicsContext gc = this.getGraphicsContext2D();
        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, this.getWidth(), this.getHeight());

        double width = 20;
        for(int i = 0; i < numbers.size(); i++) {
            gc.setFill(Color.RED);
            double len = (numbers.get(i).doubleValue() / sum) * this.getHeight();
            gc.fillRect((width + 5) * i, 0, width, len);;
        }
    }
}
// 具体的观察者饼图，对更新方法进行实现，绘制饼图
public class PieChart extends Canvas implements DataViewer{
    public PieChart(double w, double h) {
        this.setWidth(w);
        this.setHeight(h);
    }
    @Override
    public void update(DataEditor de) {
        List<Number> numbers = de.getData();
        double sum = 0;
        for(Number n:numbers) {
            sum += n.doubleValue();
        }
        GraphicsContext gc = this.getGraphicsContext2D();
        gc.setFill(Color.WHITE);
        gc.fillRect(0, 0, this.getWidth(), this.getHeight());
```

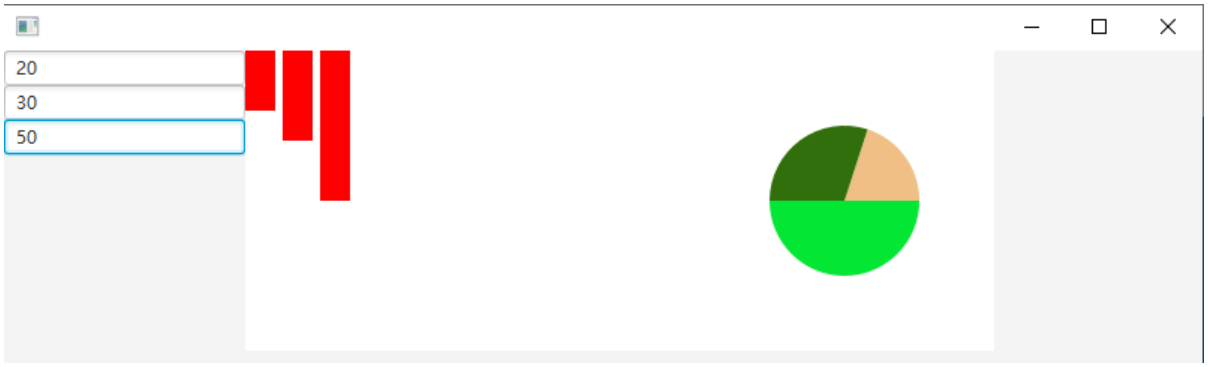
```
        double start = 0;
        for(Number n:numbers) {
            int r = (int)(255 * Math.random());
            int g = (int)(255 * Math.random());
            int b = (int)(255 * Math.random());
            gc.setFill(Color.rgb(r, g, b));
            double extent = (n.doubleValue() / sum) * 360;
            gc.fillArc(50, 50, 100, 100, start, extent, ArcType.ROUND);
            start += extent;
        }
    }
}
```

客户端实现：

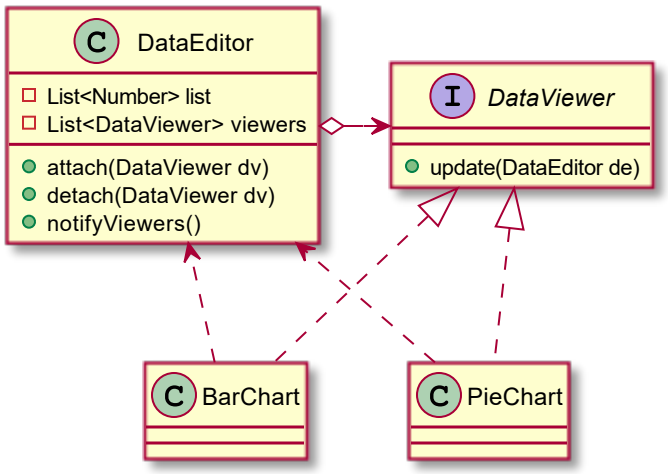
```
public class MyApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        DataEditor de = new DataEditor(3);
        BarChart bc = new BarChart(300, 200);
        PieChart pc = new PieChart(200, 200);
        de.attach(pc);
        de.attach(bc);
        FlowPane pane = new FlowPane();
        pane.getChildren().addAll(de, bc, pc);

        Scene scene = new Scene(pane, 800, 600);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

运行效果：



类图：



效果与适用性

优点

- 实现主题对象与观察者对象松耦合
- 支持通知广播
- 观察者自己决定是否订阅通知

缺点

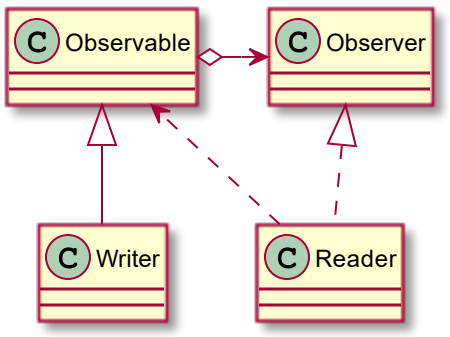
- 松耦合导致代码关系不明显，较难理解
- 广播通知可能带来效率问题

扩展案例

读者与作者

> 设计一个小说订阅程序，当读者更新内容时会通知所有订阅该作者的读者。

采用`java.util`包中的`Observable`类和`Observer`接口来实现。



定义作者类，`Observable`类已经实现了观察者的管理、通知等功能，具体作者类只需要实现业务相关代码：

```
public class Writer extends Observable{
    private String name;
    public void publish(String content) {
        this.setChanged(); //改变状态后才会通知
        this.notifyObservers(content);
    }
}
```

定义读者类，实现`update()`方法：

```
public class Reader implements Observer{
    private String name;
    @Override
    public void update(Observable o, Object arg) {
        String content = (String) arg;
        System.out.println("--------------------");
        System.out.printf("%s, 你好:\n", name);
        Writer w = (Writer) o;
        System.out.printf("作者%s更新了章节《%s》\n", w.getName(), content);
    }
}
```

编写测试程序：

```
public class Client {
    public static void main(String[] args) {
        Writer w1 = new Writer("小花");
        Writer w2 = new Writer("戏诸侯");
        Observer r1 = new Reader("张三");
        Observer r2 = new Reader("李四");
        Observer r3 = new Reader("王五");
        w1.addObserver(r1);
        w1.addObserver(r2);
        w2.addObserver(r1);
        w2.addObserver(r2);
        w2.addObserver(r3);
        w1.publish("起源");
        w2.publish("校园");
    }
}
```

运行效果：

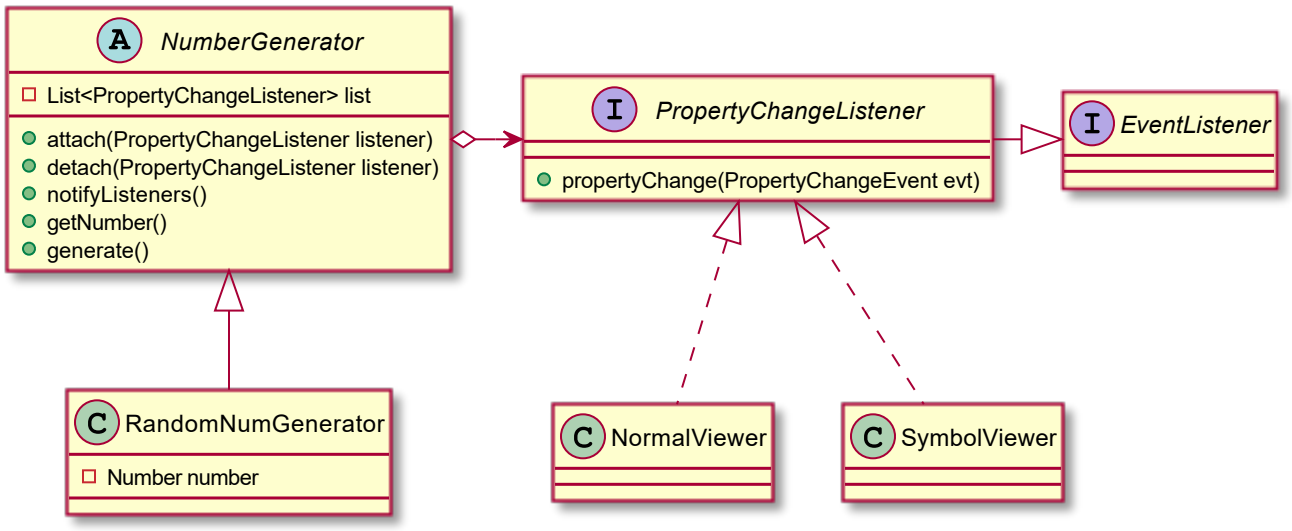
```
------------------
李四，你好：
作者小花更新了章节《起源》
------------------
张三，你好：
作者小花更新了章节《起源》
------------------
王五，你好：
作者戏诸侯更新了章节《校园》
------------------
李四，你好：
作者戏诸侯更新了章节《校园》
------------------
张三，你好：
作者戏诸侯更新了章节《校园》
```

数字生成与显示

> 设计一个数字生成器生成0~9的整数，另设计两个数字显示程序，按不同的方式显示数字，例如一种按普通数字显示，另一种按某种符号显示。显示方式如下：

```
正常显示:4
符号显示:****
正常显示:5
符号显示:*****
正常显示:4
符号显示:****
正常显示:5
符号显示:*****
正常显示:4
符号显示:****
```

由于原有`java.util`包中的`Observable`类和`Observer`接口已经被Java9之后的版本弃用，这里采用`PropertyChangeListener`接口和`PropertyChangeEvent`来实现观察者模式。



`PropertyChangeListener`接口用于定义观察者针对状态变更的处理方法：

```
public interface PropertyChangeListener extends java.util.EventListener {
    // PropertyChangeEvent对象包含事件源、变更属性名称、属性旧值以及新值信息
    void propertyChange(PropertyChangeEvent evt);
}
```

定义抽象主题类：

```
public abstract class NumberGenerator{
    private List<PropertyChangeListener> listeners = new ArrayList<PropertyChangeListener>();

    // 观察者绑定与解绑
    public void attach(PropertyChangeListener listener) {
        listeners.add(listener);
    }
    public void detach(PropertyChangeListener listener) {
        listeners.remove(listener);
    }
    // 通知方法，将主题对象和新值封装在事件对象中传递给观察者
    public void notifyListeners() {
        PropertyChangeEvent pce = new PropertyChangeEvent(this, "number", null, getNumber());
        for(PropertyChangeListener listener:listeners) {
            listener.propertyChange(pce);
        }
    }
    // 状态变更以及获取方法
    public abstract Number getNumber();
    public abstract void generate();
}
```

定义具体主题类：

```
public class RandomNumGenerator extends NumberGenerator{
    private Integer number = 0;
    public Number getNumber() {
        return number;
    }
    // 随机生成整数，改变状态，通知观察者
    public void generate() {
        Integer newNum = (int)(Math.random() * 10);
        number = newNum;
        notifyListeners();
    }
}
```

定义具体的观察者类：

```
// 数字正常显示观察者类
public class NormalViewer implements PropertyChangeListener{
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        Number number = (Number)evt.getNewValue();
        System.out.println("正常显示:" + number.intValue());
    }
}
// 采用符号的方式显示的观察者类
public class SymbolViewer implements PropertyChangeListener{
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        Number number = (Number)evt.getNewValue();
        StringBuilder builder = new StringBuilder();
        for(int i = 0; i < number.intValue(); i++) {
            builder.append("*");
        }
        System.out.println("符号显示:" + builder.toString());
    }
}
```

客户端测试：

```
public class Client {
    public static void main(String[] args) {
        NumberGenerator generator = new RandomNumGenerator();
        generator.attach(new NormalViewer());
        generator.attach(new SymbolViewer());
        for(int i = 0; i < 5; i++)
            generator.generate();
    }
}
```

思考题

> 模拟RSS（Really Simple Syndication）新闻订阅和获取的过程，主要步骤为：用户使用RSS阅读器订阅有价值的RSS信息源，其次接收和获取定制的RSS信息。