

6 原型模式（深克隆和浅克隆）

实验介绍

本文会介绍 GoF 23 种设计模式的第 6 种设计模式：原型模式。提到原型模式，就不得不提深克隆和浅克隆的区别，所以本文也会向大家分别介绍利用深克隆和浅克隆两种方式来实现原型模式。

知识点

- 原型模式定义
- 深克隆和浅克隆
- 原型模式示例
- 原型模式适用场景
- 原型模式的优缺点
- 原型模式能解决什么问题

什么是原型模式

原型模式（Prototype Pattern）一般指的是我们通过一个原型实例，然后创建出和原型实例一样的重复对象，主要就是用来实现对象的克隆。

深克隆和浅克隆

原型模式的实质就是克隆对象，而克隆又可以分为浅克隆和深克隆。

浅克隆是指拷贝对象时仅仅拷贝对象本身和对象中的基本变量，但是不拷贝对象包含的引用类型。假如对象中含有一个引用属性，那么拷贝的时候只会把引用属性的地址拷贝过来，这样的缺点就是一旦原型实例对象的引用属性发生了修改，那么克隆过来的对象也会一起变动。

深克隆不仅拷贝对象本身，而且会将引用对象也一起实现拷贝，这样一旦原型实例中的引用属性发生变化，不会影响到克隆后的对象。

如果大家对这两个概念还是有点模糊，也没有关系，继续往下面看示例，示例中我会采用浅克隆和深克隆来分别实现原型模式，通过代码实现来对比两种克隆方式，大家就会更容易明白了。

示例

这里我们需要新建一个 `prototype` 目录，相关类创建在 `prototype` 目录下。

- 新建一个原型接口 `IPrototype.java`。

```
package prototype;

public interface IPrototype {
    IPrototype clone(); // 克隆方法
}
```

这个抽象接口中，我们只定义了一个克隆方法，用来克隆对象。

- 接下来定义一个原型实例类 `ShallowPrototype.java` 来实现 `IPrototype` 接口。

```
package prototype;

import java.util.List;

public class ShallowPrototype implements IPrototype {
    private String name;

    private int age;

    private List<String> phoneList;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public List<String> getPhoneList() {
        return phoneList;
    }
}
```

```

    public void setPhoneList(List<String> phoneList) {
        this.phoneList = phoneList;
    }

    @Override
    public IPrototype clone() {
        ShallowPrototype shallowPrototype = new ShallowPrototype();
        shallowPrototype.setAge(this.age);
        shallowPrototype.setName(this.name);
        shallowPrototype.setPhoneList(this.phoneList);
        return shallowPrototype;
    }
}

```

这个类里面定义了 3 个属性，其中 1 个是属于引用类型。

- 最后我们新建一个测试类 `TestShallowPrototype.java`。

```

package prototype;

import java.util.ArrayList;
import java.util.List;

public class TestShallowPrototype {
    public static void main(String[] args){
        //初始化一个原型实例对象ShallowPrototype
        ShallowPrototype shallowPrototype = new ShallowPrototype();
        shallowPrototype.setAge(18);
        shallowPrototype.setName("张三");
        List<String> phoneList = new ArrayList<>();
        phoneList.add("131XXXXXXX");
        shallowPrototype.setPhoneList(phoneList);

        ShallowPrototype cloneShallowPrototype = (ShallowPrototype)
        shallowPrototype.clone();//克隆原型对象
        System.out.println(shallowPrototype.getPhoneList());
        System.out.println(cloneShallowPrototype.getPhoneList());
        System.out.println(shallowPrototype.getPhoneList() ==
        cloneShallowPrototype.getPhoneList());//true
    }
}

```

现在我们需要验证一下结果，先执行 `javac prototype/*.java` 命令进行编译。然后再执行 `java prototype.TestShallowPrototype` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```
9      ShallowPrototype shallowPrototype = new ShallowPrototype();
10     shallowPrototype.setAge(18);
11     shallowPrototype.setName("张三");
12     List<String> phoneList = new ArrayList<>();
13     phoneList.add("131XXXXXXX");
14     shallowPrototype.setPhoneList(phoneList);
15
16     ShallowPrototype cloneShallowPrototype = (ShallowPrototype) shallowPrototype.clone();
17     System.out.println(shallowPrototype.getPhoneList());
18     System.out.println(cloneShallowPrototype.getPhoneList());
19     System.out.println(shallowPrototype.getPhoneList() == cloneShallowPrototype.getPhoneList());
20 }
21
shiyanolou@9039f515cfa7: /home/project x
shiyanolou:project/ $ javac prototype/*.java
shiyanolou:project/ $ java prototype.TestShallowPrototype
[131XXXXXXX]
[131XXXXXXX]
true
shiyanolou:project/ $
```

可以看到，最后两个对象中的 `phoneList` 属性是相等的，说明这两个属性实际上指向的是同一个内存地址，所以一旦一个对象修改了这个属性，那么另一个对象也会随之改变。

- 接下来我们改造一下测试类 `TestShallowPrototype.java`，来验证一下是不是只要改变了 `list`，两个对象都会同时发生改变。

```
package prototype;

import java.util.ArrayList;
import java.util.List;

public class TestShallowPrototype {
    public static void main(String[] args){
        //初始化一个原型实例对象ShallowPrototype
        ShallowPrototype shallowPrototype = new ShallowPrototype();
        shallowPrototype.setAge(18);
        shallowPrototype.setName("张三");
        List<String> phoneList = new ArrayList<>();
        phoneList.add("131XXXXXXX");
        shallowPrototype.setPhoneList(phoneList);

        ShallowPrototype cloneShallowPrototype = (ShallowPrototype)
shallowPrototype.clone(); //克隆原型对象
        System.out.println(shallowPrototype.getPhoneList());
        System.out.println(cloneShallowPrototype.getPhoneList());
        System.out.println(shallowPrototype.getPhoneList() ==
cloneShallowPrototype.getPhoneList()); //true

        //修改原对象中的属性phoneList
        List<String> list = shallowPrototype.getPhoneList(); //获取到
phoneList
        list.add("132XXXXXXX"); //再添加一个值
        System.out.println(shallowPrototype.getPhoneList()); //输出:
[131XXXXXXX, 132XXXXXXX]
        System.out.println(cloneShallowPrototype.getPhoneList()); //输出:
[131XXXXXXX, 132XXXXXXX]
```

```
}
}
```

重新执行 `javac prototype/*.java` 命令进行编译。然后再执行 `java prototype.TestShallowPrototype` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

15
16 ShallowPrototype cloneShallowPrototype = (ShallowPrototype) shallowPrototype.clone();
17 System.out.println(shallowPrototype.getPhoneList());
18 System.out.println(cloneShallowPrototype.getPhoneList());
19 System.out.println(shallowPrototype.getPhoneList() == cloneShallowPrototype.getPhoneList());
20
21 //修改原对象中的属性phoneList
22 List<String> list = shallowPrototype.getPhoneList(); //获取到phoneList
23 list.add("132XXXXXXX"); //再添加一个值
24 System.out.println(shallowPrototype.getPhoneList()); //输出: [131XXXXXXX, 132XXXXXXX]
25 System.out.println(cloneShallowPrototype.getPhoneList()); //输出: [131XXXXXXX, 132XXXXXXX]
26
27 }

> shiyanlou@9039f515cfa7: /home/project
shiyanlou:project/ $ javac prototype/*.java
shiyanlou:project/ $ java prototype.TestShallowPrototype
[131XXXXXXX]
[131XXXXXXX]
true
[131XXXXXXX, 132XXXXXXX]
[131XXXXXXX, 132XXXXXXX]
shiyanlou:project/ $

```

可以看到，当我们把 `phoneList` 修改之后，两个对象都一同被修改了。

- 我们再看另一种操作方式，还是修改一下上面的测试类 `TestShallowPrototype.java`。

```

package prototype;

import java.util.ArrayList;
import java.util.List;

public class TestShallowPrototype {
    public static void main(String[] args){
        //初始化一个原型实例对象ShallowPrototype
        ShallowPrototype shallowPrototype = new ShallowPrototype();
        shallowPrototype.setAge(18);
        shallowPrototype.setName("张三");
        List<String> phoneList = new ArrayList<>();
        phoneList.add("131XXXXXXX");
        shallowPrototype.setPhoneList(phoneList);

        ShallowPrototype cloneShallowPrototype = (ShallowPrototype)
        shallowPrototype.clone(); //克隆原型对象
        System.out.println(shallowPrototype.getPhoneList());
        System.out.println(cloneShallowPrototype.getPhoneList());
        System.out.println(shallowPrototype.getPhoneList() ==
        cloneShallowPrototype.getPhoneList()); //true

        //直接修改prototypeA对象的phoneList指向
        shallowPrototype.setPhoneList(new ArrayList<>());
    }
}

```

不知道有没有人有这种误解，以为上面说的修改属性，是直接修改的 `phoneList`，但是这种修改方式并没有修改到其所指向的地址，只是把 `phoneList` 属性修改了一个指向，而原 `list` 对象还在内存中，并且被 `cloneShallowPrototype` 对象所持有的，所以这种修改是不会影响到克隆对象的，只有修改了 `phoneList` 属性所指向的对象本身才有效。

上面就是一个浅克隆的示例，浅克隆存在的问题使它有些时候并不适合我们的业务需求，所以接下来让我们再看一个深克隆的示例。

- 新建一个深克隆原型实例对象 `DeepPrototype.java` 实现 `Cloneable` 和 `Serializable` 两个接口。

```
package prototype;

import java.io.*;
import java.util.List;

public class DeepPrototype implements Cloneable, Serializable {
    private String name;

    private int age;

    private List<String> phoneList;

    public String getName() {
        return name;
    }

    public void setName(String name){
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public List<String> getPhoneList() {
        return phoneList;
    }

    public void setPhoneList(List<String> phoneList) {
        this.phoneList = phoneList;
    }
}
```

```

    public Object clone(){
        return this.deepClone();
    }

    public DeepPrototype deepClone(){
        try {
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(bos);
            oos.writeObject(this);

            ByteArrayInputStream bis = new
ByteArrayInputStream(bos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bis);

            DeepPrototype clone = (DeepPrototype)ois.readObject();

            return clone;
        }catch (Exception e){
            System.out.println("深克隆异常");
            e.printStackTrace();
        }
        return null;
    }
}

```

可以看到，深克隆类相比较于浅克隆，多定义了一个 `deepClone` 方法，而 `deepClone` 内部是通过序列化来克隆一个对象。

- 新建一个测试类 `TestDeepPrototype.java` 来测试一下。

```

package prototype;

import java.util.ArrayList;
import java.util.List;

public class TestDeepPrototype {
    public static void main(String[] args) throws
CloneNotSupportedException {
        DeepPrototype deepPrototype = new DeepPrototype();
        deepPrototype.setAge(18);
        deepPrototype.setName("张三");
        List<String> phoneList = new ArrayList<>();
        phoneList.add("131XXXXXXX");
        deepPrototype.setPhoneList(phoneList);
    }
}

```

```

        DeepPrototype cloneDeepProtoType =
        (DeepPrototype)deepPrototype.clone();
        System.out.println(deepPrototype.getPhoneList()); //输出:
        [131XXXXXXXX]
        System.out.println(cloneDeepProtoType.getPhoneList()); //输出:
        [131XXXXXXXX]
        System.out.println(deepPrototype.getPhoneList() ==
        cloneDeepProtoType.getPhoneList()); //false
    }
}

```

重新执行 `javac prototype/*.java` 命令进行编译。然后再执行 `java prototype.TestDeepPrototype` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

DeepPrototype.class      9
DeepPrototype.java      10
IPrototype.class        11
IPrototype.java          12
ShallowPrototype.class  13
ShallowPrototype.java   14
TestDeepPrototype.cl... 15
TestDeepPrototype.java  16
TestShallowPrototype... 17
TestShallowPrototype... 18
TestShallowPrototype... 19
strategy
template
> shiyanlou@9039f515cfa7: /home/project x
true
[131XXXXXXXX, 132XXXXXXXX]
[131XXXXXXXX, 132XXXXXXXX]
shiyanlou:project/ $ javac prototype/*.java
shiyanlou:project/ $ java prototype.TestDeepPrototype
[131XXXXXXXX]
[131XXXXXXXX]
false
shiyanlou:project/ $

```

可以看到，这里虽然输出的值是相同的，但是最后的结果却是 `false` 了，这就说明原对象和克隆对象之间是完全独立，`phoneList` 属性没有共用同一个内存地址。

- 同样的，我们修改一下 `phoneList` 来测试一下是不是真的实现了深克隆，修改 `TestDeepPrototype.java` 文件。

```

package prototype;

import java.util.ArrayList;
import java.util.List;

public class TestDeepPrototype {
    public static void main(String[] args) throws
    CloneNotSupportedException {
        DeepPrototype deepPrototype = new DeepPrototype();
        deepPrototype.setAge(18);
        deepPrototype.setName("张三");
        List<String> phoneList = new ArrayList<>();
        phoneList.add("131XXXXXXXX");
        deepPrototype.setPhoneList(phoneList);
    }
}

```



```

        DeepPrototype cloneDeepProtoType =
        (DeepPrototype)deepPrototype.clone();
        System.out.println(deepPrototype.getPhoneList()); //输出:
[131XXXXXXXX]
        System.out.println(cloneDeepProtoType.getPhoneList()); //输出:
[131XXXXXXXX]
        System.out.println(deepPrototype.getPhoneList() ==
cloneDeepProtoType.getPhoneList()); //false

        //修改原对象中的属性phoneList
        List<String> list = deepPrototype.getPhoneList(); //获取到phoneList
        list.add("132XXXXXXXX"); //再添加一个值
        System.out.println(deepPrototype.getPhoneList()); //输出:
[131XXXXXXXX, 132XXXXXXXX]
        System.out.println(cloneDeepProtoType.getPhoneList()); //输出:
[131XXXXXXXX]
    }
}

```

重新执行 `javac prototype/*.java` 命令进行编译。然后再执行 `java prototype.TestDeepPrototype` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

14
15
16
17
18
19
20
21
22
23
24
25
DeepPrototype cloneDeepProtoType = (DeepPrototype)deepPrototype.clone();
System.out.println(deepPrototype.getPhoneList()); //输出: [131XXXXXXXX]
System.out.println(cloneDeepProtoType.getPhoneList()); //输出: [131XXXXXXXX]
System.out.println(deepPrototype.getPhoneList() == cloneDeepProtoType.getPhoneList()); //fa

//修改原对象中的属性phoneList
List<String> list = deepPrototype.getPhoneList(); //获取到phoneList
list.add("132XXXXXXXX"); //再添加一个值
System.out.println(deepPrototype.getPhoneList()); //输出: [131XXXXXXXX, 132XXXXXXXX]
System.out.println(cloneDeepProtoType.getPhoneList()); //输出: [131XXXXXXXX]
}

shiyaniou@9039f515ca7: /home/project x
false
shiyaniou:project/ $ javac prototype/*.java
shiyaniou:project/ $ java prototype.TestDeepPrototype
[131XXXXXXXX]
[131XXXXXXXX]
false
[131XXXXXXXX, 132XXXXXXXX]
[131XXXXXXXX]
shiyaniou:project/ $ H

```

结果一目了然，修改了原对象的引用属性之后，并没有影响到克隆对象。

关于深克隆不得不说的

上面我们的深克隆实现了两个接口：`Cloneable` 和 `Serializable`。为什么要实现这两个接口呢？

`Serializable` 接口大家应该比较好理解，因为我们的 `deepClone` 方法是通过序列化来实现对象克隆的，所以必须要实现序列化接口，否则会抛出异常 `NotSerializableException`。

对于 `Cloneable` 接口，因为 `clone` 方法我们知道其实是 `Object` 对象的，而 Java 中所有的对象默认都是 `Object` 对象的子类，所以我们的类中也同样的具有 `clone` 方法，但是默认的 `clone` 方法实现的是浅克隆，为了使得我们的深克隆对象中不会同时具备深克隆和浅克隆两个功能，我这里选择了重写 `clone` 方法，而 Java 中的规范约定，如果我们需要实现 `clone` 功能，那么必须要实现 `Cloneable` 接口，否则就会抛出 `CloneNotSupportedException` 异常。

原型模式适用场景

原型模式的作用就是克隆对象，而如果一个对象本身的创建就非常简单，那么没必要使用克隆模式，所以原型模式一般适用于类初始化消耗资源较多时或者就是创建一个对象非常复杂的场景。

原型模式优点

1. 当我们创建一个对象比较复杂时，使用原型对象通常效率会更高也更方便快捷。

原型模式缺点

1. 每个对象都需要单独实现克隆的方法。
2. 深克隆和浅克隆需要灵活应用，否则可能会导致业务出错。

实验总结

本文主要介绍了原型模式的两种写法：浅克隆和深克隆写法。同时也介绍了浅克隆和深克隆的区别，在实际业务中，并不是说浅克隆一定不好，而深克隆就一定很好，我们一定要视实际需求来灵活运用浅克隆和深克隆。

7 观察者模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 7 种设计模式：观察者模式。观察者顾名思义就是有监听的意思，可以用来实现对象的监听。观察者模式主要有两种写法：一种是监听到消息之后主动获取，另一种就是对方主动推送消息到观察者。

知识点

- 观察者模式定义
- 观察者模式示例
- JDK 自带观察者模式局限性
- 观察者模式适用场景
- 观察者模式的优缺点
- 观察者模式能解决什么问题

什么是观察者模式

观察者模式（Observer Pattern）也叫做发布订阅模式，其定义了对象之间的一对多依赖，让多个观察者对象同时监听一个主体对象，当主体对象发生变化时，它的所有依赖者（观察者）都会收到通知并更新，观察者模式属于行为型模式。

push 模式示例

下面我们就以获取天气预报中的气温举例进行说明，看看观察者模式应该如何实现（这里我们需要新建一个 `observer` 目录，相关类创建在 `observer` 目录下）。

- 新建一个观察者接口 `Observer.java`。

```
package observer;

public interface Observer {
    void update(float temperature); //更新天气信息
}
```

- 接下来新建一个 `Subject.java` 类，用来管理观察者，主要定义了三个方法。

```
package observer;

public interface Subject {
    void registerObserver(Observer o); //注册观察对象
    void removeObserver(Observer o); //移除观察对象
    void notifyObservers(); //通知观察对象
}
```

- 接下来再建立一个 `WeatherData.java` 类，实现 `Subject` 接口。

```
package observer;

import java.util.ArrayList;
import java.util.List;

public class WeatherData implements Subject {
    private List<Observer> observers; //观察者不止一个，所以用list进行维护

    private float temperature; //温度
```

```

    public void setMeasurements(float temperature){
        this.temperature = temperature;

        notifyObservers();//气温信息发生变化时，通知所有观察者
    }

    public WeatherData() { //初始化list
        this.observers = new ArrayList<>();
    }

    /**
     * 注册观察者
     * @param o
     */
    @Override
    public void registerObserver(Observer o) {
        observers.add(o);
    }

    /**
     * 移除观察者
     * @param o
     */
    @Override
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if(i ≥ 0){
            observers.remove(i);
        }
    }

    /**
     * 通知所有观察者
     */
    @Override
    public void notifyObservers() {
        for (Observer observer : observers){ //遍历所有观察者
            observer.update(temperature); //通知观察者更新数据信息
        }
    }
}

```

WeatherData 类就相当于是一个被观察者，所以接下来我们还需要一个观察者。

- 新建一个观察者类 `WeatherDisplay.java`，需要实现 Observer 接口。

```

package observer;

public class WeatherDisplay implements Observer {
    private Subject subject; //维护观察者
    private float temperature; //温度

    public WeatherDisplay(Subject subject) { //注册监听对象
        this.subject = subject;
        subject.registerObserver(this);
    }

    @Override
    public void update(float temperature) { //当被观察者气温发生变化会调用这个方法，也就等于更新了观察者对象的数据
        this.temperature = temperature;
    }

    public void display(){
        System.out.println("当前最新的温度为: " + temperature);
    }
}

```

- 最后我们新建一个测试类 `TestWeather.java` 来测试一下。

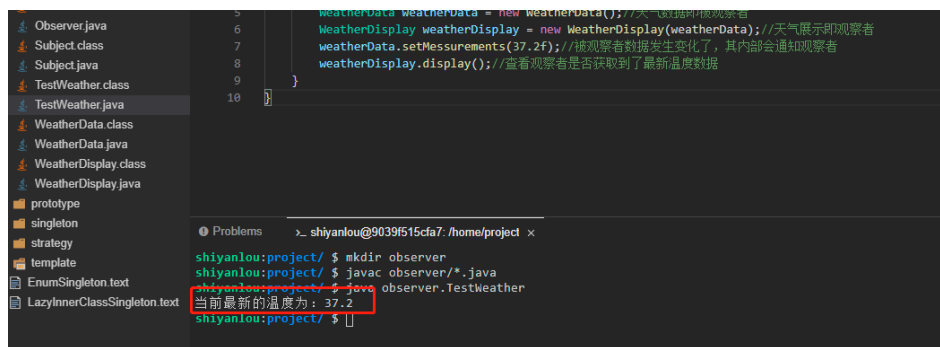
```

package observer;

public class TestWeather {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData(); //天气数据即被观察者
        WeatherDisplay weatherDisplay = new WeatherDisplay(weatherData); //天气展示即观察者
        weatherData.setMeasurements(37.2f); //被观察者数据发生了变化了，其内部会通知观察者
        weatherDisplay.display(); //查看观察者是否获取到了最新温度数据
    }
}

```

现在我们需要验证一下结果，先执行 `javac observer/*.java` 命令进行编译。然后再执行 `java observer.TestWeather` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



可以看到，我们只是设置了 WeatherData 对象中的温度，但是 WeatherDisplay 中也实时改变了温度，这就是观察者模式。但是我们这里是观察者主动推的数据给观察者，也就是 push 模式，这样不论观察者是不是需要数据，都会被推送，那么假如数据信息很多，但是我只要其中一种呢？能不能只是观察者自己去拿数据，这是可以的，这就是接下来我们要介绍的观察者模式的另一种写法，也就是 pull 写法。

pull 模式示例

在 JDK 中自带了观察者模式的写法，下面我们就利用 JDK 自带的观察者模式来实现一个 pull 类型的观察者模式。

比如我们以空间好友自行去获取好友发表在空间的动态来举例说明，看看到底该怎么写 pull 类型的观察者模式。

- 新建一个空间动态类 Trends.java 来记录动态信息。

```
package observer;

public class Trends {
    private String nickName; //发表动态的用户昵称

    private String content; //发表的动态内容

    public String getNickName() {
        return nickName;
    }

    public void setNickName(String nickName) {
        this.nickName = nickName;
    }

    public String getContent() {
        return content;
    }
}
```

```

    }

    public void setContent(String content) {
        this.content = content;
    }
}

```

- 新建一个空间 `Zone.java` 类（被观察者），继承 JDK 自带的被观察者对象：`Observable`。

```

package observer;

import java.util.Observable;

public class Zone extends Observable {
    //发表动态
    public void publishTrends(Trends trends){
        System.out.println(trends.getNickName() + "发表了一个动态【" +
trends.getContent() + "】");
        setChanged(); //占位,只是设置一个标记说明数据改变了
        notifyObservers(trends); //通知所有观察者
    }
}

```

- 新建一个好友 `Friends.java` 类（观察者），实现 JDK 自带的观察者接口：`Observer`。

```

package observer;

import java.util.Observable;
import java.util.Observer;

public class Friends implements Observer {
    private String name; //看动态的人名

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void update(Observable o, Object arg) { //获取（空间）被观察者数据
        Trends trends = new Trends();
        if(null != arg && arg instanceof Trends){

```

```

        trends = (Trends)arg;
    }

    System.out.println(this.getName() + ", 您好! 您收到了来自 " +
trends.getNickName() +
        "的一条动态【" + trends.getContent() + "】" + "快去点赞
吧!");
    }
}

```

- 最后我们新建一个测试类 `TestObserver.java` 来测试一下。

```

package observer;

public class TestObserver {
    public static void main(String[] args) {
        Zone zone = new Zone();
        Friends friends = new Friends(); // 观察者，即查看好友动态的人
        friends.setName("张三丰");

        Trends trends = new Trends(); // 被观察者，即发送动态的人
        trends.setNickName("张无忌");
        trends.setContent("祝太师傅长命百岁!");
        zone.addObserver(friends); // 注册观察者
        zone.publishTrends(trends); // 发布动态
    }
}

```

重新执行 `javac observer/*.java` 命令进行编译。然后再执行 `java observer.TestObserver` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

shiyanolou@9039f515cfa7: ~/home/project
shiyanolou:project/ $ mkdir observer
shiyanolou:project/ $ javac observer/*.java
shiyanolou:project/ $ java observer.TestWeather
当前最新的温度为: 37.2
shiyanolou:project/ $ javac observer/*.java
shiyanolou:project/ $ java observer.TestObserver
张无忌发表了一个动态【祝太师傅长命百岁!】
张三丰, 您好! 您收到了来自张无忌的一条动态【祝太师傅长命百岁!】快去点赞吧!
shiyanolou:project/ $

```

可以看到，当动态发布之后，好友立即就收到了动态，而这里我们是主动去获取数据的，即 `Friends` 类的 `update` 方法。我们可以根据自己的需求想要什么数据就拿什么数据，其它不关心的数据一律不要。

JDK 自带的观察者模式的局限性

JDK 自带的观察者模式虽然可以主动去 pull 自己需要的数据，但也同时存在以下两个问题：

1. Observable 是一个类而不是一个接口，所以复用性就不是很好，如果某类同时想具有 Observable 类和另一个超类的行为，就会有问题，毕竟 Java 不支持多继承。
2. Observable 将关键的方法保护起来了，比如 setChanged 方法，也就是我们只能继承 Observable 类才能实现想要的功能，这个违反了合成复用原则。

观察者模式适用场景

观察者模式一般适用于需要实时监听数据的场景，比如事件监听器等。

观察者模式的优点

观察者和被观察者之间建立了一个抽象的耦合，要扩展观察者只需要新建观察者并注册进去就可以，扩展性好。

观察者模式的缺点

观察者之间有过多的细节依赖、提高时间消耗及程序的复杂度，此外，当被观察者对象过多时，会使得系统非常复杂难以维护。

实验总结

本文主要介绍了观察者模式的两种写法：pull 和 push，两种写法通过不同的模式实现，最后还介绍了 JDK 自带的观察者模式的局限性，在实际开发中我们可以综合多方便进行考虑，灵活选择写法进行实现。

9 装饰者模式

什么是装饰者模式

装饰者模式（Decorator Pattern）是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案来扩展原有对象的功能，装饰者模式属于结构型模式。

示例

下面我们以生活中的蛋糕举例说明，看看如何实现装饰者模式（这里我们需要新建一个 `decorator` 目录，相关类创建在 `decorator` 目录下）。

- 新建一个蛋糕的抽象类 `Cake.java`，这个类中定义了两个方法，一个获取蛋糕的基本信息，另一个获取蛋糕的价格。

```
package decorator;

import java.math.BigDecimal;

public abstract class Cake {
    public abstract String getCakeMsg(); // 获取蛋糕的基本信息

    public abstract BigDecimal getPrice(); // 获取蛋糕的价格
}
```

- 接下来定义一个原生蛋糕（没有加其它比如水果之类的） `BaseCake.java`，并实现抽象蛋糕类 `Cake`。

```
package decorator;

import java.math.BigDecimal;

public class BaseCake extends Cake {
    @Override
    public String getCakeMsg() { // 获取蛋糕信息
        return "我是一个8英寸的普通蛋糕";
    }

    @Override
    public BigDecimal getPrice() { // 获取蛋糕价格
        return new BigDecimal("68");
    }
}
```

- 现在我们还缺少一个装饰器，新建一个抽象装饰器 `CakeDecorator.java`，注意装饰器也需要实现抽象蛋糕类 `Cake`。

```
package decorator;

import java.math.BigDecimal;

public abstract class CakeDecorator extends Cake {
    private Cake cake;

    public CakeDecorator(Cake cake) {
```

```

        this.cake = cake;
    }

    @Override
    public String getCakeMsg() { // 获取蛋糕的信息
        return this.cake.getCakeMsg(); // 调用被装饰的对象原生方法
    }

    @Override
    public BigDecimal getPrice() { // 获取蛋糕价格
        return this.cake.getPrice(); // 调用被装饰的对象原生方法
    }
}

```

可以看到，装饰器当中持有了蛋糕类 `Cake`，这个看起来有点像静态代理模式，但是请别着急，继续往后面看，这两种模式还是有本质区别的。

- 现在假如我们需要给蛋糕加点葡萄，那么就需要新建一个葡萄装饰器类 `CakeAddGrapeDecorator.java`，同时继承抽象装饰器类 `CakeDecorator`。

```

package decorator;

import java.math.BigDecimal;

public class CakeAddGrapeDecorator extends CakeDecorator {

    public CakeAddGrapeDecorator(Cake cake) {
        super(cake);
    }

    @Override
    public String getCakeMsg() { // 获取蛋糕信息
        return super.getCakeMsg() + "+1个葡萄"; // 调用父类装饰器方法，再加上自定义的装饰(加1个葡萄)
    }

    @Override
    public BigDecimal getPrice() { // 获取价格
        return super.getPrice().add(new BigDecimal("5")); // 调用父类装饰器方法，再加上自定义的装饰(加5块钱)
    }
}

```

- 假如又需要加芒果，那么就再新建一个芒果装饰器类 `CakeAddMangoDecorator.java`，同时也继承抽象装饰器类 `CakeDecorator`。

```

package decorator;

```

```
import java.math.BigDecimal;

public class CakeAddMangoDecorator extends CakeDecorator {

    public CakeAddMangoDecorator(Cake cake) {
        super(cake);
    }

    @Override
    public String getCakeMsg() { // 获取蛋糕信息
        return super.getCakeMsg() + "+1个芒果"; // 调用父类装饰器方法，再加上自定义的装饰(加1个芒果)
    }

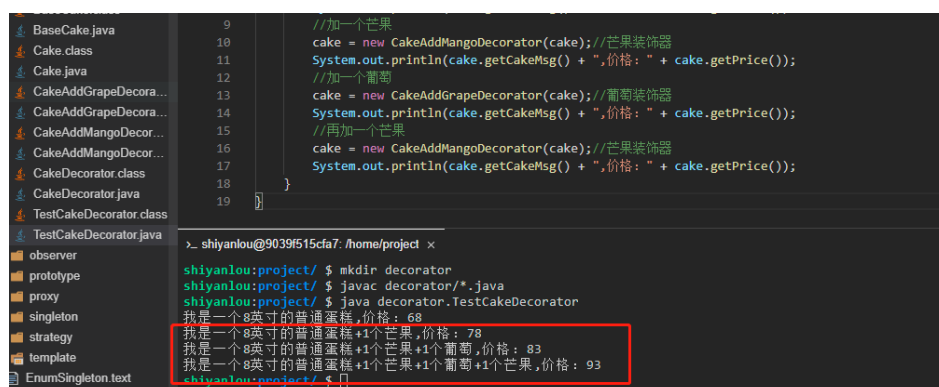
    @Override
    public BigDecimal getPrice() { // 获取价格
        return super.getPrice().add(new BigDecimal("10")); // 调用父类装饰器方法，再加上自定义的装饰(加10块钱)
    }
}
```

- 最后我们新建一个测试类 `TestCakeDecorator.java`。

```
package decorator;

public class TestCakeDecorator {
    public static void main(String[] args) {
        Cake cake = null;
        // 普通蛋糕
        cake = new BaseCake();
        System.out.println(cake.getCakeMsg() + ", 价格: " +
            cake.getPrice());
        // 加一个芒果
        cake = new CakeAddMangoDecorator(cake); // 芒果装饰器
        System.out.println(cake.getCakeMsg() + ", 价格: " +
            cake.getPrice());
        // 加一个葡萄
        cake = new CakeAddGrapeDecorator(cake); // 葡萄装饰器
        System.out.println(cake.getCakeMsg() + ", 价格: " +
            cake.getPrice());
        // 再加一个芒果
        cake = new CakeAddMangoDecorator(cake); // 芒果装饰器
        System.out.println(cake.getCakeMsg() + ", 价格: " +
            cake.getPrice());
    }
}
```

执行 `javac decorator/*.java` 命令进行编译，然后再执行 `java decorator.TestCakeDecorator` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



The screenshot shows an IDE with a project named 'decorator'. The left sidebar lists files: BaseCake.java, Cake.class, Cake.java, CakeAddGrapeDecora..., CakeAddGrapeDecora..., CakeAddMangoDecor..., CakeAddMangoDecor..., CakeDecorator.class, CakeDecorator.java, TestCakeDecorator.class, and TestCakeDecorator.java. The main editor displays the code for 'TestCakeDecorator.java'. The code defines a 'Cake' interface with 'getCakeMsg()' and 'getPrice()' methods. It then defines several decorators: 'CakeAddMangoDecorator', 'CakeAddGrapeDecorator', and 'CakeDecorator' (which implements 'Cake'). The 'TestCakeDecorator' class uses these decorators to create cakes with different toppings (mango, grape, mango, grape, mango). The output of the program is shown in the bottom console, listing the cake descriptions and their prices: '我是一个8英寸的普通蛋糕, 价格: 68', '我是一个8英寸的普通蛋糕+1个芒果, 价格: 78', '我是一个8英寸的普通蛋糕+1个芒果+1个葡萄, 价格: 83', and '我是一个8英寸的普通蛋糕+1个芒果+1个葡萄+1个芒果, 价格: 93'.

可以看到，在使用装饰者模式之后，扩展之前的功能变得极为方便，可以根据现有的装饰器进行任意组合，而如果现有装饰器无法满足也可以新建装饰器来完成功能的扩展，依然十分方便。

装饰者模式适用场景

1. 装饰者模式能将代理对象与真实被调用的目标对象分离，降低了系统的耦合度，所以扩展性比较好。
2. 动态的给一个对象添加功能时非常方便，而且还支持随时撤销这些添加的功能。

装饰器模式优点

1. 装饰者比继承更加灵活，可以在不改变原有对象的情况下动态地给一个对象扩展功能，即插即用。
2. 通过使用不同装饰类以及这些装饰类的排列组合，可以实现不同效果。
3. 新增装饰者模式时，只需要新增对应的装饰者类，无需修改源码，符合开闭原则。

装饰器模式缺点

1. 当装饰者非常多的时候，会引起类膨胀，使得系统更加复杂难以维护。

实验总结

本文主要介绍了装饰者模式的原理及其使用，装饰者模式和代理模式都增强了原对象的功能，但是代理模式是通过重新生成代理类来实现功能增加，同一时间一般是一对一的关系，而装饰者模式是一对多关系，同一时间可以使用多个装饰器同时增强原有对象功能。

10 适配器模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 10 种设计模式：适配器模式。适配器模式是一种在最开始设计程序的时候不会考虑的设计模式，而是在后期需要新功能兼容旧功能时来做一种适配，适配器模式是一种相对比较简单的设计模式。

知识点

- 适配器模式的定义
- 适配器模式示例
- 适配器模式适用场景
- 适配器模式的优缺点
- 适配器模式能解决什么问题

什么是适配器模式

适配器模式（Adapter Pattern）是指将一个类的接口转换成客户期望的另一个接口，使原本的接口不兼容的类可以一起工作，适配器模式属于结构型设计模式。

登录是一个非常常用的功能，在最原始的 web 网站一般都是通过账号密码登录，但是随着通讯软件的发展，现在的登录都需要支持手机登录，或者是 qq、微信、微博等第三方快捷登录，但是不管选择什么登录，登录之后的处理逻辑肯定是一样的，所以为了遵循软件的开闭原则，我们不能直接改变原有的登录逻辑，这时候可以考虑让适配器模式闪亮登场了。

示例

下面我们通过一个登录例子进行说明，来看看适配器模式是怎么实现的（这里我们需要新建一个 `adapter` 目录，相关类创建在 `adapter` 目录下）。

- 这是一个最原始的登录逻辑处理类 `LoginService.java`。

```
package adapter;

public class LoginService {
    //为了便于理解，我们忽略账号注册功能，只考虑登录功能
    public void login(String userName,String password){
        System.out.println("登录成功，欢迎您" + userName);
    }
}
```

- 然后现在需要支持手机登录，新建一个类 `LoginByTelephone.java`，并继承原有的登录类 `LoginService`。

```
package adapter;

public class LoginByTelephone extends LoginService {
```

```

@Override
public void login(String userName, String password) {
    super.login(userName, password);
}

public void loginByTelephone(String telephone){
    //处理业务逻辑，如注册，存储手机号校验验证码等
    //为了兼容之前的账号密码登录，可以初始化与一个固定一个账号，并设置密
    码

    System.out.println("-----手机登录-----");
    this.login(telephone, "设置好的密码");
}
}

```

- 然后过了一段时间，又需要支持微信登录了，这时候可以再新建一个类 `LoginByWechat.java`，并继承原有的登录类 `LoginService`。

```

package adapter;

public class LoginByWechat extends LoginService {
    @Override
    public void login(String userName, String password) {
        super.login(userName, password);
    }

    public void loginByWechat(String openid){
        //处理业务逻辑，如注册，存储微信账号信息等
        //为了兼容之前的账号密码登录，可以初始化与一个固定一个账号，并设置密
        码

        System.out.println("-----微信登录-----");
        this.login(openid, "设置好的密码");
    }
}

```

- 没多久，又需要支持 QQ 登录了，这时候可以再新建一个类： `LoginByQQ.java`，并继承原有的登录类 `LoginService`。

```

package adapter;

public class LoginByQQ extends LoginService {
    @Override
    public void login(String userName, String password) {
        super.login(userName, password);
    }

    public void loginByQQ(String qqNum){
        //处理业务逻辑，如注册，存储qq账号信息等
    }
}

```

```
//为了兼容之前的账号密码登录，可以初始化与一个固定一个账号，并设置密码
码

System.out.println("-----QQ登录-----");
this.login(qqNum,"设置好的密码");
}
}
```

- 最后我们新建一个测试类 `TestLoginAdapter.java` 来测试一下。

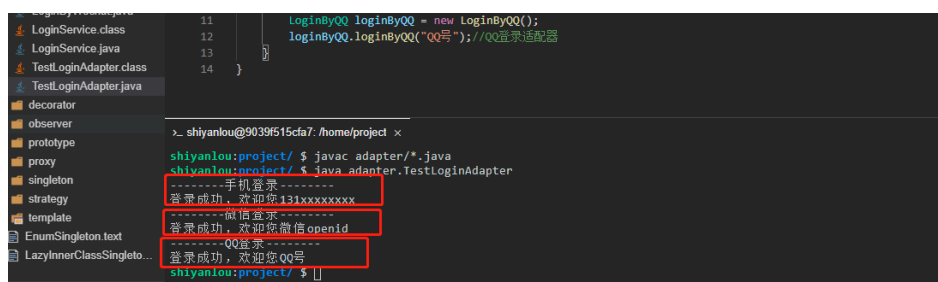
```
package adapter;

public class TestLoginAdapter {
    public static void main(String[] args) {
        LoginByTelephone loginByTelephone = new LoginByTelephone(); //手机
        登录适配器
        loginByTelephone.loginByTelephone("131xxxxxxx");

        LoginByWechat loginByWechat = new LoginByWechat();
        loginByWechat.loginByWechat("微信openid"); //微信登录适配器

        LoginByQQ loginByQQ = new LoginByQQ();
        loginByQQ.loginByQQ("QQ号"); //QQ登录适配器
    }
}
```

执行 `javac adapter/*.java` 命令进行编译，然后再执行 `java adapter.TestLoginAdapter` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



```

11 LoginByQQ loginByQQ = new LoginByQQ();
12 loginByQQ.loginByQQ("QQ号"); //QQ登录适配器
13 }
14 }

shiyanylou@9039f515ca7: /home/project x
shiyanylou:project/ $ javac adapter/*.java
shiyanylou:project/ $ java adapter.TestLoginAdapter
-----手机登录-----
登录成功，欢迎您131xxxxxxx
-----微信登录-----
登录成功，欢迎您微信openid
-----QQ登录-----
登录成功，欢迎您QQ号
shiyanylou:project/ $
```

适配器模式的实现特别简单，就是通过继承来实现的。

适配器模式适用场景

1. 针对已经存在的类，它的方法和需求不匹配的情况可以通过适配器来进行转换兼容。
2. 适配器模式不是软件设计阶段考虑的设计模式，是随着软件维护，产生了许多功能类似而接口不相同情况下的一种解决方案。

适配器模式优点

1. 能提高类的透明性和复用，现有的类复用且不需要改变。
2. 目标类和适配器类的解耦提高程序的扩展性。
3. 扩展功能时通过新建类来实现，不需改动源码，符合开闭原则。

适配器模式缺点

1. 适配器过多时会降低代码阅读性，使得代码变得比较凌乱，难以阅读。
2. 适配器模式是通过继承来实现的，违背了合成复用原则。

实验总结

本文主要介绍了适配器模式的使用，适配器模式一般都是在系统后期维护阶段为了兼容性而考虑使用的一种设计模式，虽然使用是比较简单，但是在有些场景确实也是一种比较好的解决方法，比如 MyBatis 的源码中就是使用适配器模式来兼容了市面上几乎所有的日志框架。

11 建造者模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 11 种设计模式：建造者模式。当我们创建一个复杂对象时，可能大家的第一反应就是使用工厂模式，但是如果构建一个对象非常复杂，而且有些比如说属性之类的是可选的，且需要支持随意的动态搭配，那么这时候如果要用工厂设计模式就不太好实现了，使用建造者模式来实现就会比较方便。

知识点

- 建造者模式的定义
- 建造者模式示例
- 建造者模式适用场景
- 建造者模式的优缺点
- 建造者模式能解决什么问题

什么是建造者模式

建造者模式（Builder Pattern）是将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式属于创建型模式，对使用者而言，只需要指定需要建造的类型就可以获得对象，建造过程和细节不需要了解。

建造者模式的概念听上去有点抽象，但是实际上可以这么说，基本上只要做 Web 开发的基本都用过，只是可能自己不知道这就是建造者模式而已。

示例

我们以家庭作业为例，假设老师会根据每个不同基础的同学布置不同难度的题目来实现一个简单的建造者模式（这里我们需要新建一个 `builder` 目录，相关类创建在 `builder` 目录下）。

- 首先建造一个家庭作业类 `Homework.java`，这个类里面有四个属性，分别代表四种不同难度的家庭作业。

```
package builder;

public class Homework {
    private String easyQc; // 简答题目

    private String normalQc; // 正常题目

    private String MediumQc; // 中等难度题目

    private String HardQc; // 困难题目

    public String getEasyQc() {
        return easyQc;
    }

    public void setEasyQc(String easyQc) {
        this.easyQc = easyQc;
    }

    public String getNormalQc() {
        return normalQc;
    }

    public void setNormalQc(String normalQc) {
        this.normalQc = normalQc;
    }

    public String getMediumQc() {
        return MediumQc;
    }

    public void setMediumQc(String mediumQc) {
        MediumQc = mediumQc;
    }

    public String getHardQc() {
        return HardQc;
    }
}
```

```

        public void setHardQc(String hardQc) {
            HardQc = hardQc;
        }
    }
}

```

- 然后新建一个建造者类 `SimpleBuilder.java`，建造者类就是用来创建对象的。

```

package builder;

public class SimpleBuilder {
    private Homework homework;

    public SimpleBuilder(Homework homework) {
        this.homework = homework;
    }

    public void buildEasyQc(String easyQc){
        homework.setEasyQc(easyQc);
    }

    public void buildNormalQc(String normalQc){
        homework.setNormalQc(normalQc);
    }

    public void buildMediumQc(String mediumQc){
        homework.setMediumQc(mediumQc);
    }

    public void buildHardQc(String hardQc){
        homework.setHardQc(hardQc);
    }

    public Homework build(){
        return homework;
    }
}

```

这个建造者类里面持有了家庭作业对象，并且为其每个属性都提供了一个方法进行赋值。

- 最后我们新建一个测试类 `TestSimpleBuilder.java` 来测试一下。

```

package builder;

public class TestSimpleBuilder {
    public static void main(String[] args) {

```

```

SimpleBuilder simpleBuilder = new SimpleBuilder(new Homework());
simpleBuilder.buildEasyQc("简单题目");//1
simpleBuilder.buildNormalQc("标准难度题目");//2
simpleBuilder.buildMediumQc("中等难度题目");//3
simpleBuilder.buildHardQc("高难度题目");//4
Homework homework = simpleBuilder.build();

StringBuilder sb = new StringBuilder();
sb.append(null == homework.getEasyQc() ? "" : homework.getEasyQc()
+ ",");
sb.append(null == homework.getNormalQc() ? "" :
homework.getNormalQc() + ",");
sb.append(null == homework.getMediumQc() ? "" :
homework.getMediumQc() + ",");
sb.append(null == homework.getHardQc() ? "" :
homework.getHardQc());

System.out.println("我的家庭作业有: " + sb.toString());
}
}

```

执行 `javac builder/*.java` 命令进行编译，然后再执行 `java builder.TestSimpleBuilder` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

shiyanolou:project/ $
shiyanolou:project/ $ mkdir builder
shiyanolou:project/ $ javac builder/*.java
shiyanolou:project/ $ java builder.TestSimpleBuilder
我的家庭作业有: 简单题目,标准难度题目,中等难度题目,高难度题目
shiyanolou:project/ $

```

这就是一个简单的建造者模式，测试代码中的 1 2 3 4 四行代码可以随意搭配或者取消，这就是建造者模式的灵活性，但是我们可以发现，这里面是直接面向了具体的建造者来编程，在以后的扩展会比较困难，而我们一般都是选择面向抽象编程，所以接下来需要再看一下面向抽象的标准建造者模式写法。

- 建立一个抽象的建造者类 `HomeworkBuilder.java`。

```

package builder;

public abstract class HomeworkBuilder {
    public abstract HomeworkBuilder buildEasyQc(String easyQc);

    public abstract HomeworkBuilder buildNormalQc(String normalQc);

    public abstract HomeworkBuilder buildMediumQc(String mediumQc);

    public abstract HomeworkBuilder buildHardQc(String hardQc);

    public abstract Homework build();
}

```

这个类和之前一样都是分别为四个属性提供了一个方法，区别是这些方法都返回了建造者本身，这是为了实现后面的链式写法。

- 新建一个具体的建造者类 `ConcreateBuilder.java` 继承抽象建造者类 `HomeworkBuilder`。

```

package builder;

public class ConcreateBuilder extends HomeworkBuilder {
    private Homework homework;

    public ConcreateBuilder(Homework homework) {
        this.homework = homework;
    }

    @Override
    public HomeworkBuilder buildEasyQc(String easyQc) {
        homework.setEasyQc(easyQc);
        return this;
    }

    @Override
    public HomeworkBuilder buildNormalQc(String normalQc) {
        homework.setNormalQc(normalQc);
        return this;
    }

    @Override
    public HomeworkBuilder buildMediumQc(String mediumQc) {
        homework.setMediumQc(mediumQc);
        return this;
    }
}

```

```

@Override
public HomeworkBuilder buildHardQc(String hardQc) {
    homework.setHardQc(hardQc);
    return this;
}

@Override
public Homework build() {
    return homework;
}
}

```

- 最后新建一个测试类 `TestBuilder.java` 来测试一下。

```

package builder;

public class TestBuilder {
    public static void main(String[] args) {
        Homework homework = new Homework();
        HomeworkBuilder homeworkBuilder = new ConcreateBuilder(homework);
        homeworkBuilder.buildEasyQc("我是一道简单题目")
            .buildNormalQc("我是一道标准难度题目")
            .buildMediumQc("我是一道中等难度题目")
            .buildHardQc("我是一道高难度题目");
        homework = homeworkBuilder.build();

        StringBuilder sb = new StringBuilder();
        sb.append(null = homework.getEasyQc() ? "" : homework.getEasyQc()
+ ",");
        sb.append(null = homework.getNormalQc() ? "" :
homework.getNormalQc() + ",");
        sb.append(null = homework.getMediumQc() ? "" :
homework.getMediumQc() + ",");
        sb.append(null = homework.getHardQc() ? "" :
homework.getHardQc());

        System.out.println("我的家庭作业有: " + sb.toString());
    }
}

```

再次执行 `javac builder/*.java` 命令进行编译，然后再执行 `java builder.TestBuilder` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

The screenshot shows an IDE with a project named 'builder'. The left sidebar lists files: ConcreateBuilder.class, ConcreateBuilder.java, Homework.class, Homework.java, HomeworkBuilder.class, HomeworkBuilder.java, SimpleBuilder.class, SimpleBuilder.java, TestBuilder.class, TestBuilder.java, TestSimpleBuilder.class, TestSimpleBuilder.java, decorator, observer, prototype, and proxy. The main editor displays the following Java code:

```
9      .buildNormalQc("我是一道标准难度题目");
10     .buildMediumQc("我是一道中等难度题目");
11     .buildHardQc("我是一道高难度题目");
12
13     homework = homeworkBuilder.build();
14
15     StringBuilder sb = new StringBuilder();
16     sb.append(null == homework.getEasyQc() ? "" : homework.getEasyQc() + ",");
17     sb.append(null == homework.getNormalQc() ? "" : homework.getNormalQc() + ",");
18     sb.append(null == homework.getMediumQc() ? "" : homework.getMediumQc() + ",");
19     sb.append(null == homework.getHardQc() ? "" : homework.getHardQc());
20
21     System.out.println("我的家庭作业有: " + sb.toString());
```

The bottom panel shows the command prompt output:

```
shiyanolou:project/ $ javac builder/*.java
shiyanolou:project/ $ java builder.TestBuilder
我的家庭作业有: 我是一道简单题目,我是一道标准难度题目,我是一道中等难度题目,我是一道高难度题目
shiyanolou:project/ $
```

建造者模式适用场景

建造者模式适用于一个具有较多属性的复杂产品创建过程，而且产品的各个属性还需要支持动态变化，但属性的种类却总体稳定的场景。所以建造者和工厂模式的区别也很明显，工厂模式是返回一个完整的产品，不支持自由组合属性，而建造者更加灵活，完全可以由使用者自由搭配从而使同一个建造者可以建造出不同的产品。

建造者模式优点

1. 封装性好，创建和使用分离。
2. 扩展性好，各个建造类之间相互独立，实现了解耦。

建造者模式缺点

1. 创建产品过程中会产生多余的 Builder 对象。
2. 产品内部发生变化时，建造者都需要修改，成本较大，所以建造者模式的前提需要产品总体稳定，细节自由搭配的场景。

实验总结

本实验主要介绍了建造者模式，并分别示范了一个简单的建造者模式和标准建造者模式的区别，学习完建造者模式，大家心里应该会有熟悉的感觉，比如示例中的 SimpleBuilder 其实就是建造者模式的体现，还有 MyBatis 源码中都有其体现。

13 享元模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 13 种设计模式：享元模式。我们知道，数据库的连接非常消耗性能，所以需要连接池来减少连接操作的性能消耗，而线程池的出现也是为了减少创建线程带来的消耗。本实验介绍的享元模式正是基于这种思想的设计模式。

知识点

- 享元模式的定义
- 享元模式示例
- 内部状态和外部状态
- 享元模式能解决什么问题

什么是享元模式

享元模式（Flyweight Pattern），又称之为蝇量模式，是对象池的一种实现。主要用于减少创建对象的数量，以减少内存占用和提高性能。类似于我们的数据库连接池和线程池。

享元模式的宗旨就是共享细粒度对象，将多个对同一对象的访问集中起来，不必为每个访问者创建一个单独的对象，以此来降低内存的消耗，享元模式属于结构型模式。

示例

下面就让我们以购买火车票为例，来看看享元模式是怎么实现的（这里我们需要新建一个 `flyweight` 目录，相关类创建在 `flyweight` 目录下）。

- 首先新建一个车票接口 `ITicket.java`，只定义了一个查询车票信息的方法。

```
package flyweight;

public interface ITicket {
    void info();
}
```

- 再定义一个火车票类 `TrainTicket.java` 来实现车票接口 `ITicket`。

```
package flyweight;

public class TrainTicket implements ITicket{
    private String from;
    private String to;

    public TrainTicket(String from, String to) {
        this.from = from;
        this.to = to;
    }

    @Override
    public void info() {
        System.out.println(from + "→" + to + ":硬座：100元，硬卧：200元");
    }
}
```



```
}  
}
```

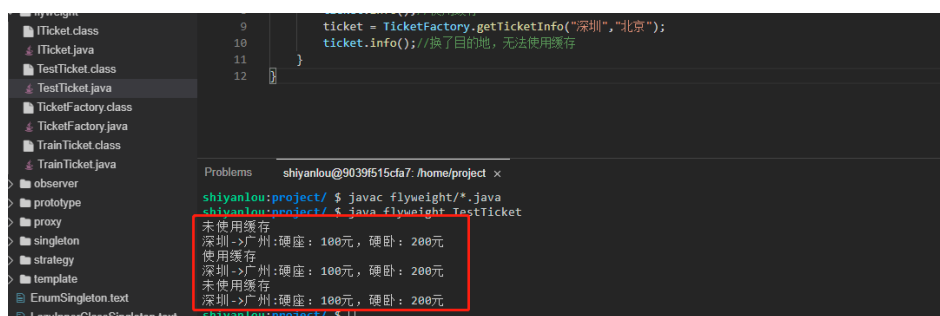
- 定义一个工厂类 `TicketFactory.java`，用来管理享元对象，其内部使用了一个 `Map` 来存储对象，把火车票的出发地和目的地作为 `key` 值，如果存在则直接从 `Map` 取，否则就新建一个对象，并且加入到 `Map` 中。

```
package flyweight;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class TicketFactory {  
    private static Map<String, ITicket> CACHE_POOL = new HashMap<>(); // 缓存对象  
  
    public static ITicket getTicketInfo(String from, String to) {  
        String key = from + "→" + to;  
        if (TicketFactory.CACHE_POOL.containsKey(key)) { // 对象存在缓存  
            System.out.println("使用缓存");  
            return TicketFactory.CACHE_POOL.get(key);  
        }  
        // 对象不存在缓存则创建一个对象，并加入缓存  
        System.out.println("未使用缓存");  
        ITicket ticket = new TrainTicket("深圳", "广州");  
        CACHE_POOL.put(key, ticket);  
        return ticket;  
    }  
}
```

- 最后来创建一个测试类 `TestTicket.java` 进行测试。

```
package flyweight;  
  
public class TestTicket {  
    public static void main(String[] args) {  
        ITicket ticket = TicketFactory.getTicketInfo("深圳", "广州");  
        ticket.info(); // 首次创建对象  
        ticket = TicketFactory.getTicketInfo("深圳", "广州");  
        ticket.info(); // 使用缓存  
        ticket = TicketFactory.getTicketInfo("深圳", "北京");  
        ticket.info(); // 换了目的地，无法使用缓存  
    }  
}
```

执行 `javac flyweight/*.java` 命令进行编译，然后再执行 `java flyweight.TestTicket` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



上面就是一个简单的享元模式的示例，可能大家会觉得这不就是一个简单的缓存的使用吗？确实如此，享元模式核心思想就是使用缓存，享元模式看起来和单例有点相似，但是单例关注的是整个对象只能有一个实例；而享元模式关注的是状态，也就是说同一个类，只有状态一致，比如上面示例中深圳到广州的火车票，这就属于状态一致，所以使用缓存，而深圳到北京又是另一个状态，所以就无法使用缓存。

那么享元模式的状态究竟是什么呢？

享元模式的状态

享元模式的状态分为两种：

- 内部状态：指对象共享出来的信息，存储在享元对象内部并且不会随环境的改变而改变。
- 外部状态：指对象需要依赖的一个标记，是随环境改变而改变的、不可共享的状态。

前面的例子中，我们可以把实例对象划分一下，比如上面车票的对象，可以把 from 和 to 两个属性作为可共享状态，不可改变。然后再新增一个属性用来对应座位，这个就属于外部状态。

享元模式状态示例

下面我们利用内部状态和外部状态将前面的示例进行一次改造。

- 首先还是新建一个车票接口 `IShareTicket.java`，比上面的接口新增了一个设置车票座位的方法。

```
package flyweight;

public interface IShareTicket {
    void info(); // 获取车票信息

    void setSeat(String seatType); // 设置车票座位
}
```

- 依然是定义一个火车票类 `TrainShareTicket.java` 来实现车票接口 `IShareTicket`。

```
package flyweight;

import java.math.BigDecimal;

public class TrainShareTicket implements IShareTicket {
    private String from; // 内部状态
    private String to; // 内部状态

    private String seatType = "站票"; // 外部状态

    public TrainShareTicket(String from, String to) {
        this.from = from;
        this.to = to;
    }

    @Override
    public void setSeat(String seatType) { // 设置座位，即设置外部状态
        this.seatType = seatType;
    }

    @Override
    public void info() {
        System.out.println(from + "→" + to + ":" + seatType +
            this.getPrice(seatType));
    }

    private BigDecimal getPrice(String seatType) { // 获取不同座位的价格
        BigDecimal value = null;
        switch (seatType) {
            case "硬座":
                value = new BigDecimal("100");
                break;
            case "硬卧":
                value = new BigDecimal("200");
                break;
            default:
                value = new BigDecimal("50");
        }
    }
}
```

```

    }
    return value;
}
}

```

- 定义一个工厂类 `TicketShareFactory.java`，用来管理享元对象，其内部使用了一个 Map 来存储对象，把火车票的出发地和目的地作为 key 值，如果存在了则直接从 Map 取，否则就新创建一个对象，并且加入到 Map 中。

```

package flyweight;

import java.util.HashMap;
import java.util.Map;

public class TicketShareFactory {
    private static Map<String, IShareTicket> CACHE_POOL = new HashMap<>
(); // 存储对象

    public static IShareTicket getTicketInfo(String from, String to){
        String key = from + "→" + to;
        if (TicketShareFactory.CACHE_POOL.containsKey(key)){//缓存中存在
            System.out.println("使用缓存");
            return TicketShareFactory.CACHE_POOL.get(key);
        }
        //对象不存在缓存则创建一个对象，并加入缓存
        System.out.println("未使用缓存");
        IShareTicket ticket = new TrainShareTicket(from, to);
        CACHE_POOL.put(key, ticket);
        return ticket;
    }
}

```

- 最后我们新建一个测试类 `TestShareTicket.java` 来测试一下。

```

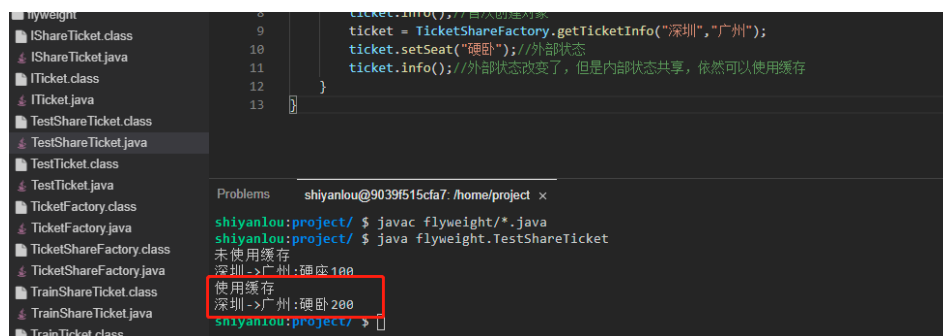
package flyweight;

public class TestShareTicket {

    public static void main(String[] args) {
        IShareTicket ticket = TicketShareFactory.getTicketInfo("深圳","广州");
        ticket.setSeat("硬座");//外部状态
        ticket.info();//首次创建对象
        ticket = TicketShareFactory.getTicketInfo("深圳","广州");
        ticket.setSeat("硬卧");//外部状态
        ticket.info();//外部状态改变了，但是内部状态共享，依然可以使用缓存
    }
}

```

执行 `javac flyweight/*.java` 命令进行编译，然后再执行 `java flyweight.TestShareTicket` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



可以看到，将状态分离之后，外部状态的修改并不影响内部状态，也就是对象依然可以被缓存。

享元模式适用场景

1. 当系统中多处需要用到一些公共信息时，可以把这些信息封装到一个对象实现享元模式，避免重复创建对象带来系统的开销。
2. 享元模式主要用于系统中存在大量相似对象，且需要缓冲池的场景，一般情况下享元模式用于底层开发较多，以便提升系统性能。

享元模式优点

减少对象的创建，降低了系统中对象的数量，故而可以降低系统的使用内存，提高效率。

享元模式缺点

1. 提高了系统的复杂度，需要注意分离出外部状态和内部状态。
2. 享元模式默认是线程不安全的，所以如果是并发场景需要考虑线程安全性问题。

实验总结

本实验主要介绍了享元模式的基本用法，随之介绍了将对象分离成内部状态和外部状态之后享元模式又是如何实现的，享元模式一般都是和工厂模式同时出现，因为我们的享元对象需要用工厂进行管理，享元模式的思想说到底就是缓存思想，所以还是非常实用的一种设计模式。

14 组合模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 14 种设计模式：组合模式。在编码原则中，有一条是：多用组合，少用继承。当然这里的组合和我们今天要讲的组合模式并不等价，这里的组合其实就是一种聚合，组合和聚合有本质的区别。

知识点

- 组合和聚合
- 组合模式的定义
- 组合模式透明写法和安全写法
- 组合模式适用场景
- 组合模式的优缺点
- 组合模式能解决什么问题

组合和聚合

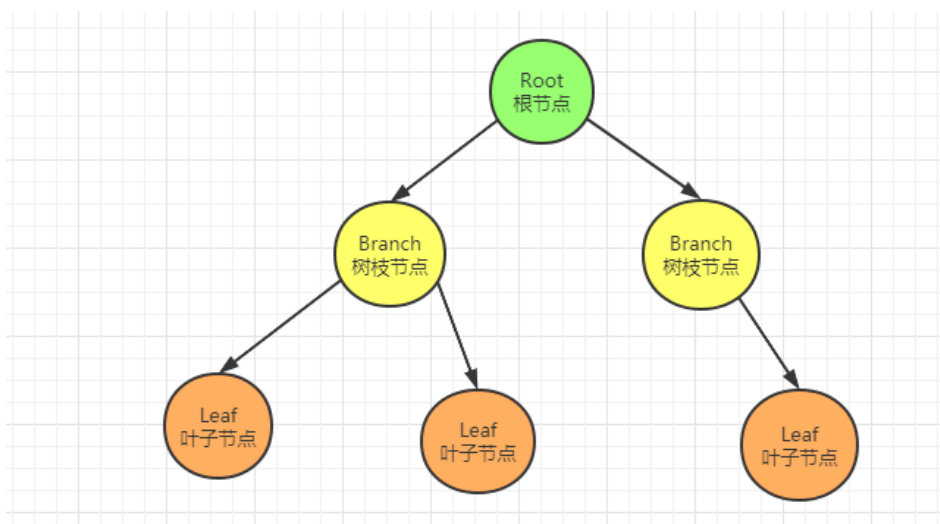
“人在一起叫团伙，心在一起叫团队”。用这句话来诠释组合与聚合的区别是比较恰当的。聚合就是说各个对象聚合在一起工作，但是我没有你也行，我照样可以正常运行。但是组合呢，关系就比较密切，组合中的各个对象之间组成了一个整体，缺少了某一个对象就不能正常运行或者说功能会有很大缺陷。

换言之：聚合对象不具备相同生命周期，而组合的对象具有相同的生命周期。

举个例子：比如电脑和 U 盘就是聚合（离开之后彼此都能正常使用），而电脑显示器和主机就是组合（没了主机或者没了显示器电脑都是不完整的）。

组合模式（Composite Pattern）也称之为整体-部分（Part-Whole）模式。组合模式的核心是通过将单个对象（叶子节点）和组合对象（树枝节点）用相同的接口进行表示，使得单个对象和组合对象的使用具有一致性，组合模式属于结构型模式。

组合模式一般用来描述整体与部分的关系，它将对象组织到树形结构中，最顶层的节点称为根节点，根节点下面可以包含树枝节点和叶子节点，树枝节点下面又可以包含树枝节点和叶子节点如下图所示：



组合模式有两种写法，分别是**透明模式**和**安全模式**。下面我们会分别示范这两种写法，并对其进行对比分析。

透明模式示例

下面我们以高考的科目为例，来看看组合模式中的透明模式该如何实现（这里我们需要新建 `composite/opacity` 目录，相关类创建在 `composite/opacity` 目录下）。

- 首先新建一个高考科目的抽象类 `GkAbstractCourse.java`，里面提供了三个方法。

```
package composite.opacity;

public abstract class GkAbstractCourse {
    public void addChild(GkAbstractCourse course){//添加子节点
        System.out.println("不支持添加操作");
    }

    public String getName() throws Exception {//获取当前节点名称
        throw new Exception("不支持获取名称");
    }
}
```

```

        public void info() throws Exception{//获取高考科目信息
            throw new Exception("不支持查询信息操作");
        }
    }
}

```

这个类是抽象类，但是在这里并没有将这些方法定义为抽象方法，而是默认都抛出异常。这么做的原因是假如定义为抽象方法，那么所有的子类都必须重写父类方法。但是这种通过抛异常的方式，如果子类需要用到的功能就重写覆盖父类方法即可，不需要用到的方法直接无需重写。

- 新增一个叶子节点，普通高考科目类 `LeafCourse.java` 来继承 `GkAbstractCourse` 类，这个类就是最底层，无法包含其它科目，比如语文，正因为其无法包含其它课程所以不需要重写 `addChild` 方法。

```

package composite.transparency;

public class LeafCourse extends GkAbstractCourse {
    private String name;//课程名称
    private String score;//课程分数

    public LeafCourse(String name, String score) {
        this.name = name;
        this.score = score;
    }

    @Override
    public String getName(){
        return this.name;
    }

    @Override
    public void info() {
        System.out.println("课程:" + this.name + ",分数:" + score);
    }
}

```

- 接下来新建一个树枝类 `BranchCourse.java`，当前类可以包含其它科目的类，比如理综可以包括物理、化学和生物，这个类将父类的 3 个方法全部进行了重写。

```

package composite.transparency;

import java.util.ArrayList;
import java.util.List;

public class BranchCourse extends GkAbstractCourse{
    private List<GkAbstractCourse> courseList = new ArrayList<>();
    private String name;//科目名称
}

```



```

        private int level; // 层级

        public BranchCourse(String name, int level) {
            this.name = name;
            this.level = level;
        }

        @Override
        public void addChild(GkAbstractCourse course) { // 添加子课程
            courseList.add(course);
        }

        @Override
        public String getName() { // 获取课程名称
            return this.name;
        }

        @Override
        public void info() throws Exception { // 打印课程信息
            System.out.println("课程:" + this.name);
            for (GkAbstractCourse course : courseList) {
                for (int i = 0; i < level; i++) { // 根据层级关系打印空格，这样打印结果可以明显看出层级关系
                    System.out.print("  ");
                }
                System.out.print(">");
                course.info();
            }
        }
    }
}

```

- 最后我们新建一个测试类 `TestTransparency.java` 来测试一下。

```

package composite.transparency;

public class TestTransparency {
    public static void main(String[] args) throws Exception {
        GkAbstractCourse ywCourse = new LeafCourse("语文", "150");
        GkAbstractCourse sxCourse = new LeafCourse("数学", "150");
        GkAbstractCourse yyCourse = new LeafCourse("英语", "150");

        GkAbstractCourse wlCourse = new LeafCourse("物理", "110");
        GkAbstractCourse hxCourse = new LeafCourse("化学", "100");
        GkAbstractCourse swCourse = new LeafCourse("生物", "90");

        GkAbstractCourse lzCourse = new BranchCourse("理综", 2);
        lzCourse.addChild(wlCourse);
    }
}

```

```

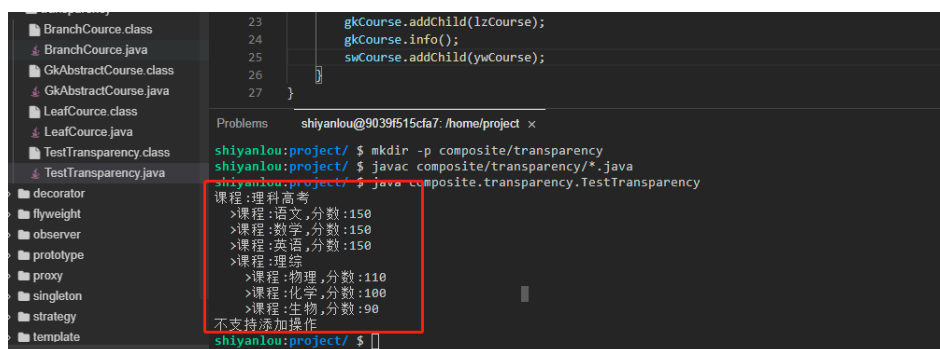
        lzCourse.addChild(hxCourse);
        lzCourse.addChild(swCourse);

        GkAbstractCourse gkCourse = new BranchCourse("理科高考",1);
        gkCourse.addChild(ywCourse);
        gkCourse.addChild(sxCourse);
        gkCourse.addChild(yyCourse);

        gkCourse.addChild(lzCourse);
        gkCourse.info();
        swCourse.addChild(ywCourse); //这里会报错，因为生物已经是叶子节点，无法继续添加子节点
    }
}

```

执行 `javac composite/transparency/*.java` 命令进行编译，然后再执行 `java composite.transparency.TestTransparency` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



```

shiyanolou@9039f515cfa7: /home/project $ mkdir -p composite/transparency
shiyanolou@9039f515cfa7: /home/project $ javac composite/transparency/*.java
shiyanolou@9039f515cfa7: /home/project $ java composite.transparency.TestTransparency
课程 :理科高考
>课程 :语文,分数 :150
>课程 :数学,分数 :150
>课程 :英语,分数 :150
>课程 :理综
>课程 :物理,分数 :110
>课程 :化学,分数 :100
>课程 :生物,分数 :90
不支持添加操作
shiyanolou@9039f515cfa7: /home/project $

```

上面就是一个透明模式的写法，之所以称之为透明模式就是因为父类将所有方法都定义好了，对各个子类完全透明。这样做的好处是客户端无需分辨当前对象是属于树枝节点还是叶子节点，因为它们具备了完全一致的接口，不过缺点就是叶子节点得到了一些不属于它的方法，比如上面的 `addChild` 方法，这个很明显违背了接口隔离性原则。

而组合模式只是规定了系统各个层次的最基础的一致性行为，而把组合（树节点）本身的方法（如树枝节点管理子类的 `addChild` 等方法）放到自身当中。接下来我们再看看安全模式应该如何实现。

安全模式示例

这里我们新建一个 `composite/safe` 目录，相关类创建在 `composite/safe` 目录下。

- 首先还是新建一个顶层的抽象课程类 `GkAbstractCourse.java`，不过这次只定义了一个通用的查询课程信息的方法，而且是抽象方法，同时又多具备了课程名字和课程分数两个属性。

```
package composite.safe;

public abstract class GkAbstractCourse {
    protected String name; // 课程名称
    protected String score; // 课程分数

    public GkAbstractCourse(String name, String score) {
        this.name = name;
        this.score = score;
    }

    public abstract void info(); // 获取课程信息
}
```

- 新增一个叶子节点，普通高考科目类 `LeafCourse.java` 来继承 `GkAbstractCourse` 类，这个类就是最底层，无法包含其它科目，比如语文，所以它没什么其它特殊的自定义方法。

```
package composite.safe;

public class LeafCourse extends GkAbstractCourse {
    public LeafCourse(String name, String score) {
        super(name, score);
    }

    @Override
    public void info() { // 获取课程信息
        System.out.println("课程:" + this.name + ", 分数:" + this.score);
    }
}
```

- 接下来新建一个树枝类 `BranchCourse.java`，当前类可以包含其它科目的类，比如理综可以包括物理、化学和生物，所以这个类额外自定义了一个层级属性和一个添加子课程 `addChild` 方法。

```
package composite.safe;

import java.util.ArrayList;
import java.util.List;

public class BranchCourse extends GkAbstractCourse {
    private List<GkAbstractCourse> courseList = new ArrayList<>(); // 子课程
    private int level; // 层级

    public BranchCourse(String name, String score, int level) {
```

```

        super(name,score);
        this.level = level;
    }

    public void addChild(GkAbstractCourse course) { //添加子课程
        courseList.add(course);
    }

    @Override
    public void info() { //打印课程信息
        System.out.println("课程:" + this.name + ",分数: " + this.score);
        for (GkAbstractCourse course : courseList){
            for (int i=0;i<level;i++){ //根据层级关系打印空格，这样打印结果可以明显看出层级关系
                System.out.print("  ");
            }
            System.out.print(">");
            course.info();
        }
    }
}

```

- 最后我们新建一个测试类 `TestSafe.java` 来测试一下。

```

package composite.safe;

public class TestSafe {
    public static void main(String[] args) throws Exception {
        LeafCourse ywCourse = new LeafCourse("语文", "150");
        LeafCourse sxCourse = new LeafCourse("数学", "150");
        LeafCourse yyCourse = new LeafCourse("英语", "150");

        LeafCourse wlCourse = new LeafCourse("物理", "110");
        LeafCourse hxCourse = new LeafCourse("化学", "100");
        LeafCourse swCourse = new LeafCourse("生物", "90");

        BranchCourse lzCourse = new BranchCourse("理综", "300", 2);
        lzCourse.addChild(wlCourse);
        lzCourse.addChild(hxCourse);
        lzCourse.addChild(swCourse);

        BranchCourse gkCourse = new BranchCourse("理科高考", "750", 1);
        gkCourse.addChild(ywCourse);
        gkCourse.addChild(sxCourse);
        gkCourse.addChild(yyCourse);

        gkCourse.addChild(lzCourse);
    }
}

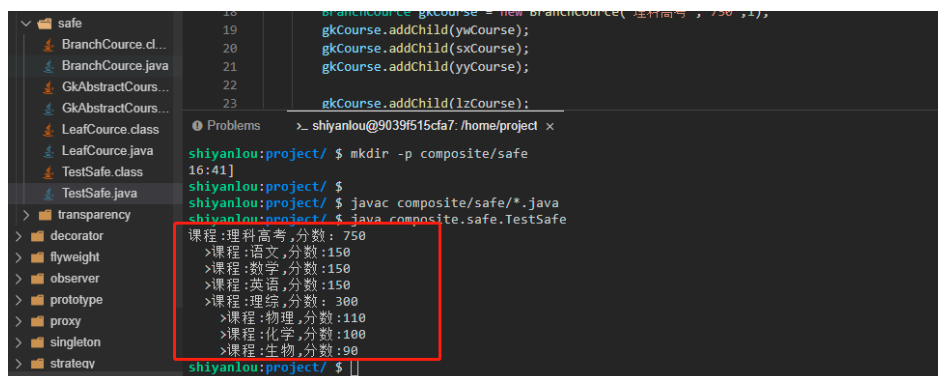
```

```

        gkCourse.info();
    }
}

```

执行 `javac composite/safe/*.java` 命令进行编译，然后再执行 `java composite.safe.TestSafe` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



```

shiyanolou:project/ $ mkdir -p composite/safe
shiyanolou:project/ $ javac composite/safe/*.java
shiyanolou:project/ $ java composite.safe.TestSafe
课程:理科高考,分数: 750
>课程:语文,分数:150
>课程:数学,分数:150
>课程:英语,分数:150
>课程:理综,分数: 300
>课程:物理,分数:110
>课程:化学,分数:100
>课程:生物,分数:90
shiyanolou:project/ $ 

```

这里和透明方式不一样，叶子节点不具备 `addChild` 功能，所以无法调用，而上面的示例中时可以被调用，但是调用之后显示不支持，这就是这两种写法最大的区别。

组合模式适用场景

组合模式一般应用在有层级关系的场景，最经典的就是树形菜单、文件和文件夹的管理等。

组合模式优点

组合模式清楚的定义了分层次的复杂对象，让客户端可以忽略层次的差异，方便对整个层次进行动态控制。

组合模式缺点

组合模式的叶子和树枝的声明是实现类而不是接口，违反了依赖倒置原则，而且组合模式会使设计更加抽象不好理解。

实验总结

本实验主要介绍了组合模式，并且讲述了组合和聚合的区别。最后介绍了组合模式的透明模式和安全模式两种写法，并分别分析了两种写法本质上的区别。

15 门面（外观）模式

什么是门面模式

门面模式（Facade Pattern），又被称之为外观模式。门面模式提供了一个统一的接口，这个接口可以用来访问相同子系统或者不同子系统中的一群接口。门面模式使得系统更加容易调用，属于结构型模式。

示例

下面就让我们以多个系统互相调用为例，看看如何实现门面模式（这里我们需要新建一个 `facade` 目录，相关类创建在 `facade` 目录下）。

- 首先新建一个子系统 `SubSystemA.java`，这里面只定义一个方法。

```
package facade;

public class SubSystemA {
    public void doSomething(){
        System.out.println("I'm A");
    }
}
```

- 再新建一个子系统 `SubSystemB.java`，同样也只定义一个方法。

```
package facade;

public class SubSystemB {
    public void doSomething(){
        System.out.println("I'm B");
    }
}
```

- 现在假设这两个系统的方法我们不想对所有人都开放，所以我们就新建一个门面类 `SimpleFacade.java` 来集成这两个子系统，并对外开放统一接口。

```
package facade;

public class SimpleFacade {
    private SubSystemA systemA = new SubSystemA();
    private SubSystemB systemB = new SubSystemB();

    public void doA(){
        this.systemA.doSomething();
    }
    public void doB(){
        this.systemB.doSomething();
    }
}
```

- 最后我们新建一个测试类 `TestSimpleFacade.java` 来测试一下。

```
package facade;

public class TestSimpleFacade {
    public static void main(String[] args) {
        SimpleFacade simpleFacade = new SimpleFacade();
        simpleFacade.doA();
        simpleFacade.doB();
    }
}
```

执行 `javac facade/*.java` 命令进行编译，然后再执行 `java facade.TestSimpleFacade` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。



可以看到，测试类中不会直接调用子系统，而是通过门面类来进行统一调用。通过这个例子，大家可能会有一种这也是设计模式的感慨，正如我们在实验一提到的，其实设计模式并不是新的高深的知识，只是一种经验总结而已。

上面的示例中门面类中提供的方法和子系统方法是一对一关系，现在假如我们一个流程里面需要很多个子系统的方法一起调用才能完成，那么这时候我们对上面的门面类进行稍加改造就可以了。

- 重新新建一个门面类 `Facade.java`。

```
package facade;

public class Facade {
    private SubSystemA systemA = new SubSystemA();
    private SubSystemB systemB = new SubSystemB();

    public void doAB(){
        this.systemA.doSomething();
        this.systemB.doSomething();
    }
}
```

- 新建测试类 `TestFacade.java` 进行测试。

```
package facade;

public class TestFacade {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.doAB();
    }
}
```

再次执行 `javac facade/*.java` 命令进行编译，然后再执行 `java facade.TestSimpleFacade` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```
shiyanolou:project/ $ mkdir facade
shiyanolou:project/ $ javac facade/*.java
shiyanolou:project/ $ java facade.TestSimpleFacade
I'm A
I'm B
shiyanolou:project/ $ javac facade/*.java
shiyanolou:project/ $ java facade.TestSimpleFacade
I'm A
I'm B
shiyanolou:project/ $ java facade.TestFacade
I'm A
I'm B
shiyanolou:project/ $
```

门面模式适用场景

门面模式一般可以适用于以下两种场景：

1. 当各个子系统越来越复杂时，可以提供门面接口来统一调用。
2. 构建多层系统结构时，利用门面对象来作为每层的入口，这样可以简化分层间的接口调用。

门面模式优点

1. 简化了调用过程，调用者无需深入了解子系统，以防给子系统带来风险。
2. 减少了系统间的依赖，松散了耦合度。
3. 将系统划分了层次，提高了安全性。
4. 通过门面类隐藏了各个子系统，使得子系统只对门面类可知，所以遵循了迪米特法则。

门面模式缺点

1. 当增加子系统或者扩展子系统功能时，可能容易带来未知风险。
2. 当我们需要新增或者修改调用逻辑时，需要修改门面类，不符合开闭原则。

实验总结

本次实验主要分析了门面模式的定义并通过示例展示了如何完成一个门面模式，希望通过本次实验可以让大家体会到设计模式并没有想象中的那么高深，学习设计模式主要是体会其中的思想，然后在适当的时候可以运用到我们代码中。

16 桥接模式

实验介绍

本实验会介绍 GoF 23 种设计模式的第 16 种设计模式：桥接模式。桥接模式其实和我们前面介绍的组合模式有点类似，桥接模式也是通过组合的思想来实现的，但是桥接模式和组合模式的侧重点不一样。

知识点

- 桥接模式的定义
- 桥接模式示例
- 桥接模式适用场景
- 桥接模式的优缺点
- 桥接模式能解决什么问题

什么是桥接模式

桥接模式（Bridge Pattern）也称之为桥梁模式，接口（Interface）模式或者柄体（Handle and Body）模式，其核心思想是将抽象部分与它的实现部分分离，使抽象部分和实现部分两个维度都可以独立的变化。

桥接模式属于结构型模式，主要目的是通过组合的方式建立两个类之间的关系，而不是通过继承来实现，从而达到解耦抽象和实现的目的。

示例

在 Web 开发中，我们一般都有三种常见的发送消息的方式：站内信、短信（SMS）、邮件。而假如系统中的消息又需要按照紧急程度来划分普通消息和加急消息，那么这就有了两个维度：

1. 抽象部分维度：发送消息的类型。
2. 实现部分维度：普通消息和紧急消息。

下面就让我们以发送消息的这两个维度来实现一个桥接模式（这里我们需要新建一个 `bridge` 目录，相关类创建在 `bridge` 目录下）。

- 首先新建一个子系统 `IMessage.java`，这里面只定义一个发送消息的方法。

```
package bridge;

public interface IMessage {
    void send(String content, String toUser); // 发送消息
}
```

- 接下来需要建立三种消息类型之短信消息类 `SmsMessage.java` 并实现 `IMessage` 接口。

```
package bridge;

public class SmsMessage implements IMessage {
    @Override
    public void send(String content, String toUser) {
        System.out.println(String.format("SMS消息 → %s: %s", toUser, content));
    }
}
```

- 继续建立三种消息类型之邮件消息类 `EmailMessage.java` 并实现 `IMessage` 接口。

```
package bridge;

public class EmailMessage implements IMessage {
    @Override
    public void send(String content, String toUser) {
        System.out.println(String.format("邮件消息 → %s: %s", toUser, content));
    }
}
```

- 继续建立三种消息类型之站内消息类 `WebMessage.java` 并实现 `IMessage` 接口。

```
package bridge;

public class WebMessage implements IMessage {
    @Override
    public void send(String content, String toUser) {
        System.out.println(String.format("站内消息 →%s: %s", toUser, content));
    }
}
```

- 现在我们需要建立一个中介人即桥接者类 `AbstractBridgeMessage.java`，桥接者类一般设置为抽象类，其内部需要持有抽象维护（消息类型）的引用，并且定义和抽象维度类相同功能的方法。

```
package bridge;

public abstract class AbstractBridgeMessage {
    private IMessage iMessage; // 持有抽象维度的引用

    public AbstractBridgeMessage(IMessage iMessage) {
        this.iMessage = iMessage;
    }

    public void sendMessage(String content, String toUser) { // 定义一个和抽象维度类中具有相同功能的方法：发送消息
        this.iMessage.send(content, toUser); // 调用抽象维度内方法：发送消息
    }
}
```

- 接下来需要建立两个用于表示不同紧急程度的对象之普通消息 `CommonMsg.java`，并继承桥接者类 `AbstractBridgeMessage`，这就是桥接模式的核心之处，因为普通消息这个具体维度不去继承抽象维度，而是继承桥接者，这就实现了抽象和实现的解耦，桥接者在中间起到了桥梁的作用。

```
package bridge;

public class CommonMsg extends AbstractBridgeMessage {
    public CommonMsg(IMessage iMessage) {
        super(iMessage);
    }

    @Override
    public void sendMessage(String content, String toUser) {
        this.doSomething(); // 做一点普通消息该做的事
        super.sendMessage(content, toUser); // 调用桥接者中的方法：发送消息
    }

    private void doSomething() {
```

```

        System.out.println("我只是一个普通消息，什么都不用做");
    }
}

```

- 最后再建立一个紧急消息类 `UrgentMessage.java`。

```

package bridge;

public class UrgentMessage extends AbstractBridgeMessage {
    public UrgentMessage( IMessage iMessage ) {
        super(iMessage);
    }

    @Override
    public void sendMessage(String content, String toUser) {
        doSomething(); // 做点紧急消息需要做的事情
        super.sendMessage(content, toUser); // 调用桥接者中的方法：发送消息
    }

    private void doSomething() {
        System.out.println("这是紧急消息，请优先发送");
    }
}

```

- 现在可以建立一个测试类 `TestBridge.java` 来测试一下了。

```

package bridge;

import java.io.IOException;

public class TestBridge {
    public static void main(String[] args) throws IOException {
        IMessage iMessage = new EmailMessage();
        AbstractBridgeMessage abstractBridgeMessage = new
        UrgentMessage(iMessage); // 紧急邮件消息
        abstractBridgeMessage.sendMessage("您好", "双子孤狼1号");
        // 再来一个普通短信消息
        System.out.println("-----分割线-----");
        iMessage = new SmsMessage();
        abstractBridgeMessage = new CommonMsg(iMessage);
        abstractBridgeMessage.sendMessage("您好", "双子孤狼2号");

        // 最后再来一个紧急的站内信
        System.out.println("-----分割线-----");
        iMessage = new WebMessage();
        abstractBridgeMessage = new UrgentMessage(iMessage);
        abstractBridgeMessage.sendMessage("您好", "实验楼的小伙伴");
    }
}

```

```
}
}
```

执行 `javac bridge/*.java` 命令进行编译，然后再执行 `java bridge.TestBridge` 命令运行测试类（大家一定要自己动手运行哦，只有自己实际去运行了才会更能体会其中的思想）。

```

11      System.out.println("-----分割线-----");
12      iMessage = new SmsMessage();
13      abstractBridgeMessage = new CommonMsg(iMessage);
14      abstractBridgeMessage.sendMessage("您好","双子孤狼2号");
15
16      //最后再来一个紧急的站内信
17      System.out.println("-----分割线-----");
18      iMessage = new WebMessage();
19      abstractBridgeMessage = new UrgentMessage(iMessage);
20
21
Problems shiyanlou@9039f515cf7: /home/project x
shiyanlou:project/ $ mkdir bridge
:21]
shiyanlou:project/ $ javac bridge/*.java
shiyanlou:project/ $ java bridge.TestBridge
这是紧急消息，请优先发送
邮件消息->双子孤狼1号：您好
-----分割线-----
我只是一个普通消息，什么都不用做
SMS消息->双子孤狼2号：您好
-----分割线-----
这是紧急消息，请优先发送
站内消息->空验楼的小伙伴：您好
shiyanlou:project/ $

```

这就是一个桥接模式，后面这两个维度无论哪个维度需要扩展，都只需要新建一个类就好了，扩展起来非常的方便。

桥接模式适用场景

1. 在抽象和具体之间需要增加更多灵活性的场景。
2. 一个类存在 2 个或者以上独立变化的维度，而这些维度又需要独立进行扩展时，但是需要注意的是桥接模式一般用于抽象和实现解耦，多个维度一般是指的一个抽象维度和多个具体维度。
3. 不希望使用继承，或因为多层继承导致类的个数剧增时可以考虑使用桥接模式。

桥接模式优点

1. 分离了抽象部分及其实现部分两个维度，实现了代码的解耦，提高了系统的扩展性。
2. 扩展功能时只需要新增类，无需修改源代码，符合开闭原则。
3. 通过组合而不是继承来实现耦合，符合合成复用原则。

桥接模式缺点

1. 增加了系统的理解难度和设计难度（和类的膨胀缺点一样，这也算是大部分设计模式的共性）。
2. 需要正确识别系统中各个独立变化的维度。

实验总结

本次实验主要介绍桥接模式的原理，在开发中大家都知道要解耦，解耦的实质就是减少对象之间的关联，而继承是一种强关联，因为一旦通过继承，那么子类就会拥有父类所有公开的方法和属性，有些可能并不是子类需要的。而组合就不一样，组合是一种弱关联，我只是持有一个对象，但是我持有对象所拥有

的功能并不是我的，和我并没有很强烈的关系。所以实质上在很多场景我们都可以通过组合来解耦继承对象之间的强关联关系。