链滴

设计模式 |08 适配器模式

作者：qq692310342
原文链接：https://ld246.com/article/1567907357427
来源网站：链滴

开头说几句

博主的博客地址： https://www.jeffcc.top/

博主学习设计模式用的书是Head First的《设计模式》，强烈推荐配套使用！

什么是适配器模式

权威定义：适配器模式将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的可以合作无间。

博主的理解：适配器就类似我们平时生活中的万能插头接口转换器，我们可以通过这个转换器来达到USB接口的线使用上Lighting的接口。

这个模式可以让客户从真正实现中解耦，在代码中客户是不知道适配器的存在的。

设计要点

1. 当我们需要使用一个现有的类而其接口并不符合你的需求时，就可以使用适配器；
2. 适配器模式改变接口以符合客户的期望，但是客户并不知情；
3. 实现一个适配器是否麻烦取决于接口的大小与复杂程度；
4. 适配器模式有两种，对象适配器和类适配器，但是类适配器需要实现多重继承，这一点java做不到；
5. 适配器是通过组合的方式进行增强的，所以无需修改原本的源码，这也是优势体现；

适配器的使用过程

1. 客户通过目标借口调用适配器的方法对适配器提出请求；
2. 适配器再使用被适配器的接口把请求转换成被适配者的一个或多个调用的接口；

3. 客户接收到了调用的结果，但是并不知道是适配器在起作用；

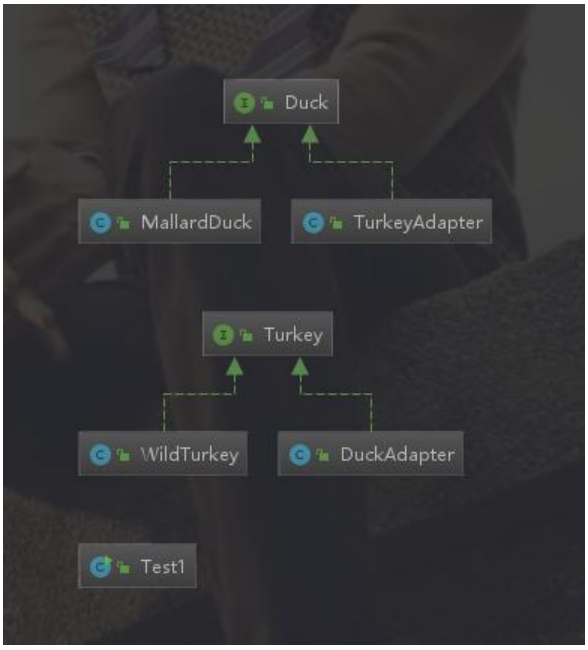
设计实例

火鸡与鸭子的适配

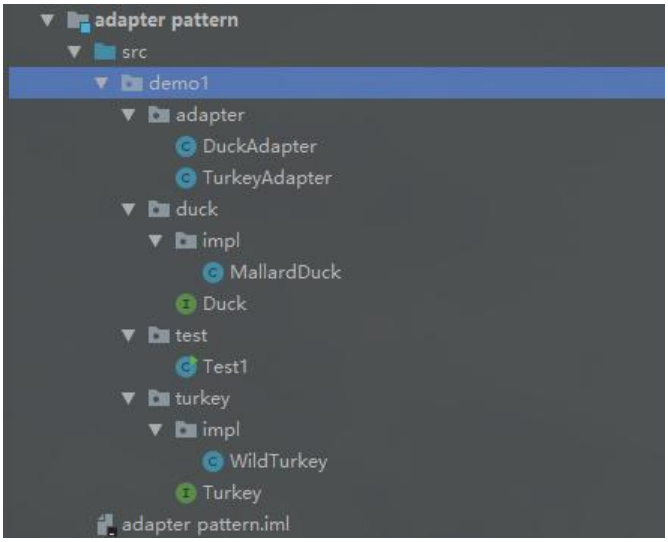
背景

现在缺失鸭子实现类，但是我们想用火鸡对象来适配出一个鸭子接口
此时鸭子接口为被适配者，火鸡为适配器
拓展设计一个鸭子适配器

项目类图



项目结构



鸭子接口

```
package demo1.duck;

/**
 * 这是鸭子的规范接口
 * 规范了叫声和飞行行为
 */
public interface Duck {
    public void quack();
    public void fly();
}
```

绿头鸭实现类

```
package demo1.duck.impl;

import demo1.duck.Duck;

/**
 * 绿头鸭 是鸭子的子类
 */
public class MallardDuck implements Duck {
    @Override
    public void quack() {
        System.out.println("Qucak!!!");
    }

    @Override
    public void fly() {
        System.out.println("I am flying!!!");
    }
}
```

火鸡接口

```
package demo1.turkey;

/**
 * 火鸡的规范接口
 * 规范了叫声以及飞行行为
 */
public interface Turkey {
    public void gobble();
    public void fly();
}
```

火鸡实现类

```
package demo1.turkey.impl;

import demo1.turkey.Turkey;

/**
 * 火鸡的实现类
 * 飞不远
 */
public class WildTurkey implements Turkey {
    @Override
    public void gobble() {
        System.out.println("gobble!!!");
    }

    @Override
    public void fly() {
        System.out.println("I am flying a short distance!!!");
    }
}
```

鸭子适配器

```
package demo1.adapter;

import demo1.duck.Duck;
import demo1.turkey.Turkey;

import java.util.Random;

/**
 * 鸭子适配火鸡
 */
public class DuckAdapter implements Turkey {
    private Duck duck;

    public DuckAdapter(Duck duck) {
        this.duck = duck;
    }

//    变成呱呱叫
    @Override
    public void gobble() {
        duck.quack();
    }

//    平均五秒飞一次
    @Override
    public void fly() {
        Random random = new Random();
        if(random.nextInt(5)==0){
            duck.fly();
        }
    }
```

```
}
```

火鸡适配器

```
package demo1.adapter;

import demo1.duck.Duck;
import demo1.turkey.Turkey;

/**
 * 火鸡适配鸭子
 */
public class TurkeyAdapter implements Duck {
    private Turkey turkey;

    //    构造器获取当前引用
    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
//        直接调用火鸡的叫声
        turkey.gobble();
    }

//    火鸡飞不远 所以调用五次
    @Override
    public void fly() {
        for (int i = 0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

测试

```
package demo1.test;

import demo1.adapter.DuckAdapter;
import demo1.adapter.TurkeyAdapter;
import demo1.duck.Duck;
import demo1.duck.impl.MallardDuck;
import demo1.turkey.Turkey;
import demo1.turkey.impl.WildTurkey;

public class Test1 {
    public static void main(String[] args) {
//        测试火鸡适配器
        System.out.println("-------------- TurkeyAdapter -----------------");
        Turkey turkey = new WildTurkey();
```

```
//        适配成功
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

//        客户只知道是只鸭子
        turkeyAdapter.quack();
        turkeyAdapter.fly();

//        测试鸭子适配器
        System.out.println("-------------- DuckAdapter -----------------");
        Duck duck = new MallardDuck();
        Turkey duckAdapter = new DuckAdapter(duck);

//        客户只知道这是一只火鸡
        duckAdapter.gobble();
        duckAdapter.fly();

    }
}
```

输出结果

```
-------------- TurkeyAdapter -----------------
gobble!!!
I am flying a short distance!!!
I am flying a short distance!!!
I am flying a short distance!!!
I am flying a short distance!!!
I am flying a short distance!!!
-------------- DuckAdapter -----------------
Qucak!!!

Process finished with exit code 0
```

真实世界的适配器

背景

枚举器（Enumeration）和迭代器（Iterator）的相互适配

枚举器适配器

```
package demo2;

import java.util.Enumeration;
import java.util.Iterator;

/**
 * 枚举器适配器
 */
public class EnumerationInterator implements Iterator {
//    创建一个枚举器 通过枚举器来使用
```

```
//    枚举器是一个只读的,所以无法remove,需要抛出异常
    private Enumeration enumeration;

    public EnumerationInterator(Enumeration enumeration) {
        this.enumeration = enumeration;
    }

    @Override
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    @Override
    public Object next() {
        return enumeration.nextElement();
    }

    @Override
    public void remove() {
        throw  new UnsupportedOperationException();
    }
}
```

迭代器适配器

```
package demo2;

import java.util.Enumeration;
import java.util.Iterator;

/**
 * 迭代器适配器
 */
public class IteratorAdapter implements Enumeration {
    private Iterator iterator;

    public IteratorAdapter(Iterator iterator) {
        this.iterator = iterator;
    }

    @Override
    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    @Override
    public Object nextElement() {
        return iterator.next();
    }
}
```

回到定义

适配器模式的实现首先需要的就是适配器要实现想转换成的接口的接口，其次就是通过另一个对象的作实现对接口的适配；完成一个适配器的功能；

我们所说的XX适配器，其实是指适配器内使用的是XX对象来进行适配的；

适配器模式和装饰者模式以及外观模式的异同会在下一篇博客做具体介绍；

END

2019年9月8日09:49:01