

桥接模式

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

我们通过下面的实例来演示桥接模式（Bridge Pattern）的用法。其中，可以使用相同的抽象类方法但是不同的桥接实现类，来画出不同颜色的圆。

介绍

意图：将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

何时使用：实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决：把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

关键代码：抽象类依赖实现类。

应用实例：1、猪八戒从天蓬元帅转世投胎到猪，转世投胎的机制将尘世划分为两个等级，即：灵魂和肉体，前者相当于抽象化，后者相当于实现化。生灵通过功能的委派，调用肉体对象的功能，使得生灵可以动态地选择。2、墙上的开关，可以看到的开关是抽象的，不用管里面具体怎么实现的。

优点：1、抽象和实现的分离。2、优秀的扩展能力。3、实现细节对客户透明。

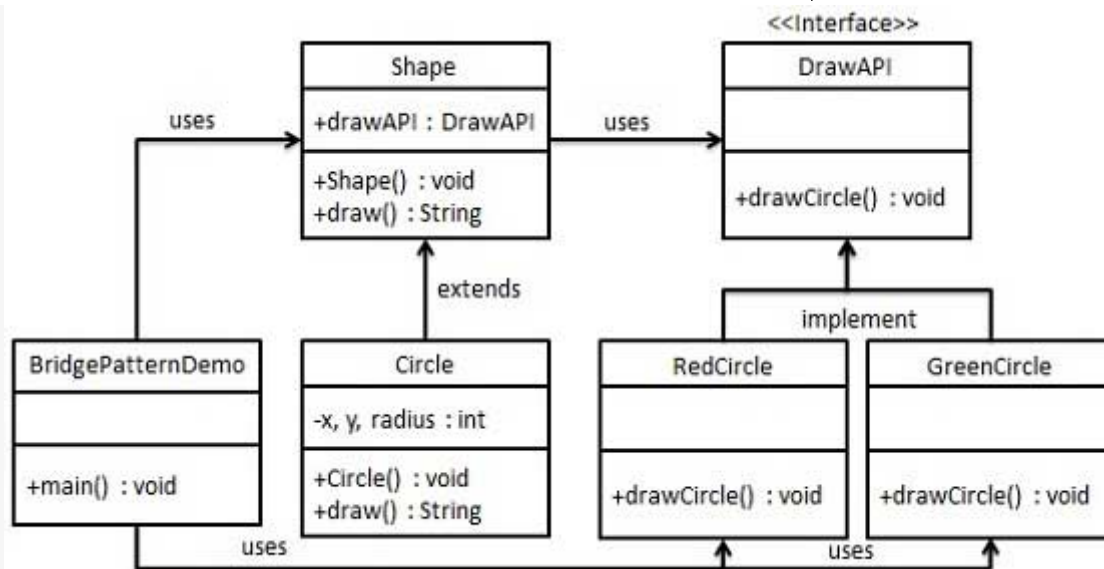
缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

使用场景：1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

注意事项：对于两个独立变化的维度，使用桥接模式再适合不过了。

实现

我们有一个作为桥接实现的 *DrawAPI* 接口和实现了 *DrawAPI* 接口的实体类 *RedCircle*、*GreenCircle*。*Shape* 是一个抽象类，将使用 *DrawAPI* 的对象。*BridgePatternDemo*，我们的演示类使用 *Shape* 类来画出不同颜色的圆。



步骤 1

创建桥接实现接口。

DrawAPI.java

```

public interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
  
```

步骤 2

创建实现了 *DrawAPI* 接口的实体桥接实现类。

RedCircle.java

```

public class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: "
            + radius + ", x: " + x + ", " + y + " ]");
    }
}
  
```

GreenCircle.java

```

public class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: "
            + radius + ", x: " + x + ", " + y + " ]");
    }
}
  
```

步骤 3

使用 *DrawAPI* 接口创建抽象类 *Shape*。

Shape.java

```
public abstract class Shape {  
    protected DrawAPI drawAPI;  
    protected Shape(DrawAPI drawAPI){  
        this.drawAPI = drawAPI;  
    }  
    public abstract void draw();  
}
```

步骤 4

创建实现了 *Shape* 接口的实体类。

Circle.java

```
public class Circle extends Shape {  
    private int x, y, radius;  
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {  
        super(drawAPI);  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
    public void draw() {  
        drawAPI.drawCircle(radius,x,y);  
    }  
}
```

步骤 5

使用 *Shape* 和 *DrawAPI* 类画出不同颜色的圆。

BridgePatternDemo.java

```
public class BridgePatternDemo {  
    public static void main(String[] args) {  
        Shape redCircle = new Circle(100,100, 10, new RedCircle());  
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());  
        redCircle.draw();  
        greenCircle.draw();  
    }  
}
```

步骤 6

执行程序，输出结果：

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]  
Drawing Circle[ color: green, radius: 10, x: 100, 100]
```

相关文章推荐

桥接模式[← 适配器模式](#)[过滤器模式 →](#)

2 篇笔记

[写笔记](#)

桥接模式：Bridge Pattern

将抽象和实现放在两个不同的类层次中，使它们可以独立地变化。——《Head First 设计模式》

将类的功能层次结构和实现层次结构相分离，使二者能够独立地变化，并在两者之间搭建桥梁，实现桥接。——《图解设计模式》

类的功能层次结构：父类具有基本功能，在子类中增加新的功能；

类的实现层次结构：父类通过声明抽象方法来定义接口，子类通过实现具体方法来实现接口；

桥接模式中有四个角色：

抽象化角色：使用实现者角色提供的接口来定义基本功能接口。

持有实现者角色，并在功能接口中委托给它，起到搭建桥梁的作用；

注意，抽象化角色并不是指它就是一个抽象类，而是指抽象了实现。

改善后的抽象化角色：作为抽象化角色的子类，增加新的功能，也就是增加新的接口（方法）；与其构成类的功能层次结构；

实现者角色：提供了用于抽象化角色的接口；它是一个抽象类或者接口。

具体的实现者角色：作为实现者角色的子类，通过实现具体方法来实现接口；与其构成类的实现层次结构。

如果抽象和实现两者做不到独立地变化，就不算桥接模式。

jade 6个月前 (09-22)



桥接模式中，具体的实现类放在作为“桥梁”的接口中实现，而“桥梁”接口中只有实现功能的抽象方法；具体实现类是继承“桥梁”，而不直接继承实现类的抽象类，抽象类与具体的实现类在结构上是相互独立的，两者的相互变化并不会影响到彼此，只要“桥梁”没变，两者的变化并不会影响到彼此。

就以上教程的实例程序来说，具体的实现是RedCircle、GreenCircle，它们的抽象类为Shape，按一般逻辑来说，我们是直接继承Shape来创建不同的具体对象，但桥接模式中是通过“桥梁”DrawAPI建立抽象与具体实现之间的联系，调用DrawAPI中的方法来具体实现。

yjb 4个月前 (11-22)