

# 桥接模式

## 简介

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。简单来说是将抽象部分与它的实现部分分离，使它们都可以独立地变化

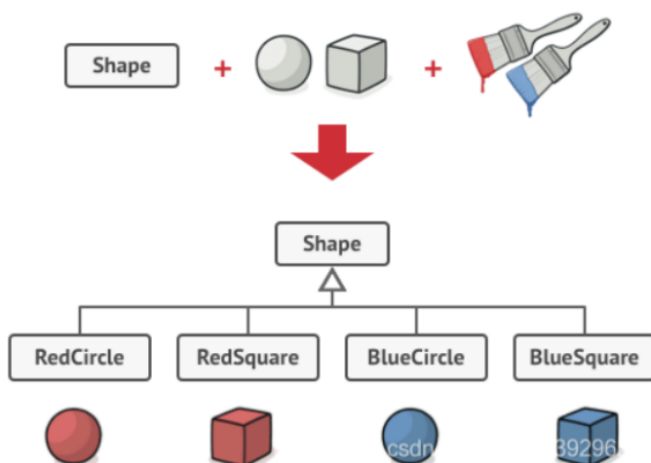
**桥**我们大家都熟悉，顾名思义就是用来将河的两岸联系起来的。而此处的桥是用来将两个独立的结构联系起来，而这两个被联系起来的结构可以独立的变化，所有其他的理解只要建立在这个层面上就会比较容易。

**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。

继承类爆炸问题：

先来看一个例子：

假如你有一个几何形状Shape类，从它能扩展出两个子类：圆形Circle和 方形Square 。你希望对这样的类层次结构进行扩展以使其包含颜色，所以你打算创建名为红色Red和蓝色Blue的形状子类。但是，由于你已有两个子类， 所以总共需要创建四个类才能覆盖所有组合， 例如 蓝色圆形BlueCircle和 红色方形RedSquare 。



**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

**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。

**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。

## 步骤 1

我们就以上述形状与颜色这两个独立的维度来实现给不同的形状刷上不同颜色的例子来讲解：

ColorAPI：用于画各种颜色的接口

```
public interface ColorAPI {  
    public void paint();  
}
```

## 步骤 2

BlueColorAPI：画蓝色的实现类和RedColorAPI：画红色的实现类

```
public class BlueColorAPI implements ColorAPI {  
    @Override  
    public void paint() {  
        System.out.println("画上蓝色");  
    }  
}
```

```
public class RedColorAPI implements ColorAPI  
{  
    @Override  
    public void paint() {  
        System.out.println("画上红色");  
    }  
}
```

## 步骤 3

Shape：抽象形状类

```

public abstract class Shape {
    protected ColorAPI colorAPI;    //添加一个颜色的成员变量以调用ColorAPI 的方法来实现
    给不同的形状上色

    public void setDrawAPI(ColorAPI colorAPI) {    //注入颜色成员变量
        this.colorAPI= colorAPI;
    }

    public abstract void draw();
}

```

## 步骤 4

Circle：圆形类

```

public class Circle extends Shape {
    @Override
    public void draw() {
        System.out.print("我是圆形");
        colorAPI.paint();
    }
}

```

## 步骤 5

Rectangle：长方形类

```

public class Rectangle extends Shape {
    @Override
    public void draw() {
        System.out.print("我是长方形");
        colorAPI.paint();
    }
}

```

## 步骤 6

Client：客户端

```

public class Client {

    public static void main(String[] args) {

```

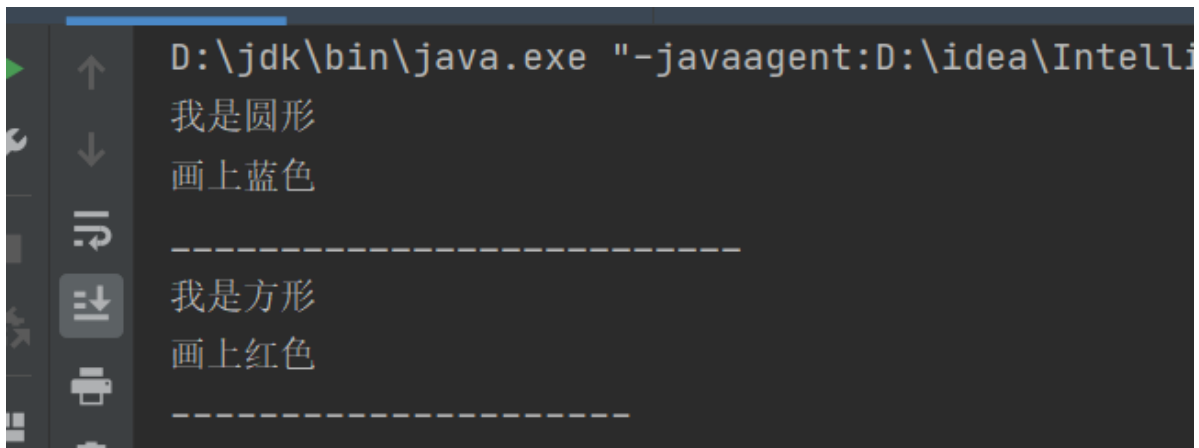
```

        //创建一个圆形
        Shape shape = new Circle();
        //给圆形蓝色的颜料
        shape.setDrawAPI(new BlueColorAPI());
        //上色
        shape.draw();

        //创建一个长方形
        Shape shape1 = new Rectangle();
        //给长方形红色的颜料
        shape1.setDrawAPI(new RedColorAPI());
        //上色
        shape1.draw();
    }
}

```

执行程序，输出结果：



```

D:\jdk\bin\java.exe "-javaagent:D:\idea\Intelli
我是圆形
画上蓝色
-----
我是方形
画上红色
-----

```

假如现在客户让我们增了一个三角形，我们只需要新增一个三角形类就可以了，而无需把每一种颜色都增加一个，我们在客户端调用时只需按照需求来挑选即可：

```

public class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("我是三角形");
        colorAPI.paint();
    }
}

```

增加颜色也是一样，我们只需要增加一个新的颜色并实现ColorAPI的接口即可，而无需更改类的层次，例如增加一个绿色：

```
public class GreenColorAPI implements ColorAPI {  
    @Override  
    public void paint() {  
        System.out.println("画上绿色");  
    }  
}
```

```
D:\jdk\bin\java.exe "-javaagent:D:\idea\IntelliJ IDEA 2020.3.1\lib\idea_rt.jar=58290:D:\id  
我是圆形  
画上蓝色  
-----  
我是方形  
画上红色  
-----  
三角形  
画上红色  
  
Process finished with exit code 0
```

## 总结： 将抽象和实现解耦，抽象类依赖与实现类

**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。