

课程目标

内容定位

1. 适配器模式

- 1.1. 适配器模式的应用场景
- 1.2. 类适配器
- 1.3. 对象适配器
- 1.4. 接口适配器
- 1.5. 重构第三方登陆自由适配的业务场景
- 1.6. 适配器模式在源码中的体现
- 1.7. 适配器模式和装饰器模式对比
- 1.8. 适配器模式的优缺点

2. 桥接模式

- 2.1. 桥接模式的通用写法
- 2.2. 桥接模式的应用场景
- 2.3. 桥接模式在业务场景中的应用
- 2.4. 桥接模式在源码中的应用
- 2.5. 桥接模式的优缺点

课程目标

- 1、掌握适配器模式和桥接模式的的应用场景。
- 2、重构第三方登录自由适配的业务场景。
- 3、了解适配器模式和桥接模式在源码中的应用。
- 4、适配器模式和桥接模式的优缺点。

内容定位

已掌握装饰器模式和组合模式，适合有项目开发经验的人群。

1. 适配器模式

适配器模式（Adapter Pattern）又叫做变压器模式，它的功能是将一个类的接口变成客户端所期望的另一种接口，从而便原本因接口不匹配而导致无法在一起工作的两个类能够一起工作，属于结构型设计模式。

原文： Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

解释： 将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

也就是说，当前系统存在两种接口A和B，客户只支持访问A接口，但是当前系统没有A接口对象，但是有B接口对象，但客户无法识别B接口，因此需要通过一个适配器C，将B接口内容转换成A接口，从而使得客户能够从A接口获取得到B接口内容。

在软件开发中，基本上任何问题都可以通过增加一个中间层进行解决。适配器模式其实就是一个中间层。综上，适配器模式其实起着转化/委托的作用，将一种接口转化为另一种符合需求的接口。

1.1. 适配器模式的应用场景

提供一个转换器（适配器），将当前系统存在的一个对象转化为客户端能够访问的接口对象。适配器适用于以下几种业务场景：

- 1、已经存在的类，它的方法和需求不匹配（方法结果相同或相似）的情况。
- 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，由于不同产品、不同厂家造成功能类似而接口不相同情况下的解决方案。有点亡羊补牢的感觉。

生活中也非常的应用场景，例如电源插转换头、手机充电转换头、显示器转接头。



适配器模式一般包含三种角色：

目标角色（Target）：也就是我们期望的接口；

源角色（Adaptee）：存在于系统中，内容满足客户需求（需转换），但接口不匹配的接口实例；

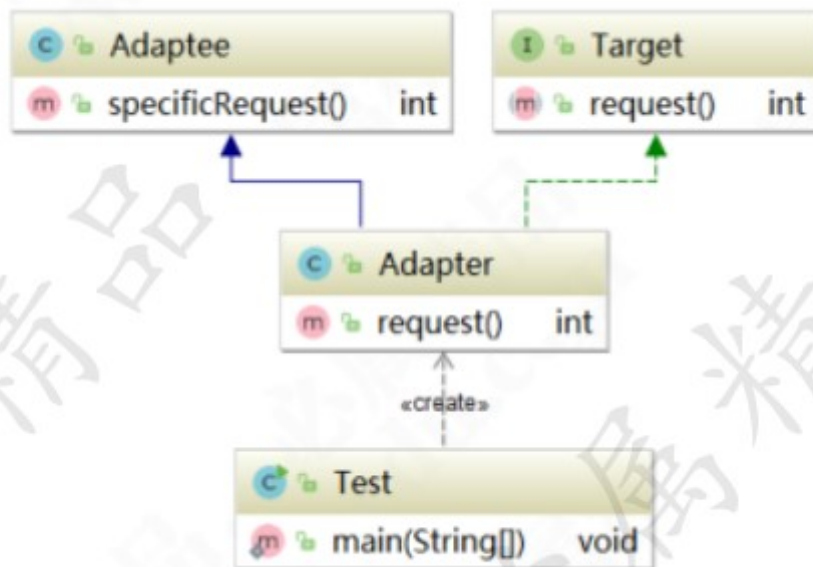
适配器（Adapter）：将源角色（Adaptee）转化为目标角色（Target）的类实例；

适配器模式各角色之间的关系如下：

假设当前系统中，客户端需要访问的是Target接口，但Target接口没有一个实例符合要求，而Adaptee实例符合要求；但是客户端无法直接使用Adaptee（接口不兼容）；因此，我们需要一个适配器（Adapter）来进行中转，让Adaptee能转化为Target接口形式；适配器模式有3种形式：类适配器、对象适配器、接口适配器。

1.2. 类适配器

类适配器的原理就是通过继承来实现适配器功能。具体做法：让Adapter实现Target接口，并且继承Adaptee，这样Adapter就具备Target和Adaptee的特性，就可以将两者进行转化。下面来看UML类图：



下面我们以一个示例进行讲解，来看下该示例分别用类适配器，对象适配器和接口适配器是怎样进行代码实现。在中国民用电都是220V交流电，但我们手机使用的锂电池使用的5V直流电。因此，我们给手机充电时就需要使用电源适配器来进行转换。下面我们由代码来还原这个生活场景，创建 Adaptee角色，需要被转换的对象AC220类，表示220V交流电：

```
1 package cn.sitedev.classadapter;
2
3 public class AC220 {
4     public int outputAC220V() {
5         int output = 220;
6         System.out.println("输出电压:" + output + "V");
7         return output;
8     }
9 }
```

创建Target角色DC5接口，表示5V直流电的标准：

```
1 package cn.sitedev.classadapter;
2
3 public interface DC5 {
4     int outputDC5V();
5 }
```

创建Adapter 角色电源适配器PowerAdapter类：

```

1 package cn.sitedev.classadapter;
2
3 public class PowerAdapter extends AC220 implements DC5 {
4
5     @Override
6     public int outputDC5V() {
7         int adapterInput = super.outputAC220V();
8         int adapterOutput = adapterInput / 44;
9         System.out.println("使用PowerAdapter输入AC" + adapterInput + "V, 输出DC" + adap
10         return adapterOutput;
11     }
12 }

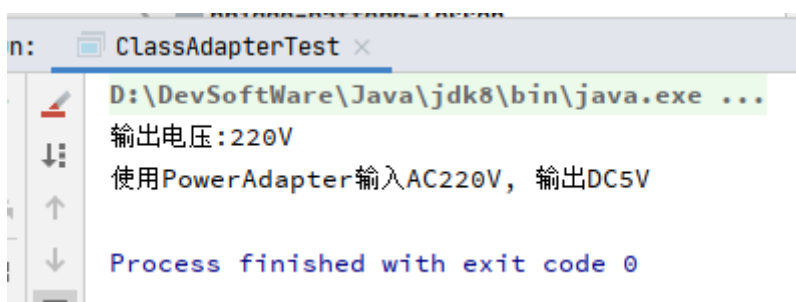
```

客户端测试代码：

```

1 package cn.sitedev.classadapter;
2
3 public class ClassAdapterTest {
4     public static void main(String[] args) {
5         DC5 adapter = new PowerAdapter();
6         adapter.outputDC5V();
7     }
8 }

```



```

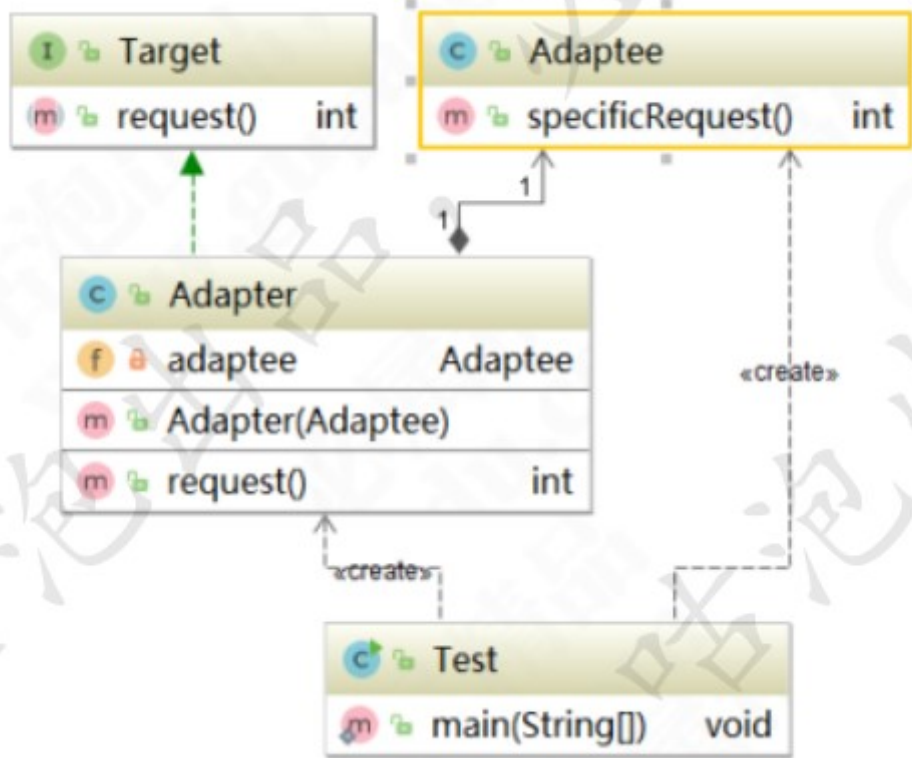
n: ClassAdapterTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
输出电压:220V
使用PowerAdapter输入AC220V, 输出DC5V
Process finished with exit code 0

```

上面的案例中，通过增加PowerAdapter电源适配器，实现了二者的兼容。

1.3. 对象适配器

对象适配器的原理就是通过组合来实现适配器功能。具体做法：让Adapter 实现Target接口，然后内部持有Adaptee实例，然后再Target 接口规定的方法内转换Adaptee。



代码只需更改适配器（Adapter）实现，其他与类适配器一致：

```

1 package cn.sitedev.objectadapter;
2
3 public interface DC5 {
4     int outputDC5V();
5 }
6 ///////////////////////////////////////////////////
7 package cn.sitedev.objectadapter;
8
9 public class AC220 {
10     public int outputAC220V() {
11         int output = 220;
12         System.out.println("输出电压:" + output + "V");
13         return output;
14     }
15 }
16 ///////////////////////////////////////////////////
17 package cn.sitedev.objectadapter;
18
19 public class PowerAdapter implements DC5 {
20
21     private AC220 ac220;
22

```

```

23     public PowerAdapter(AC220 ac220) {
24         this.ac220 = ac220;
25     }
26
27     @Override
28     public int outputDC5V() {
29         int adapterInput = ac220.outputAC220V();
30         int adapterOutput = adapterInput / 44;
31         System.out.println("使用PowerAdapter输入AC" + adapterInput + "V, 输出DC" + adap
32         return adapterOutput;
33     }
34 }

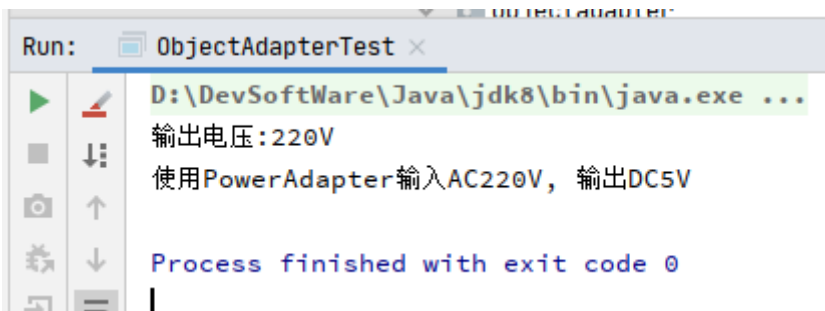
```

客户端测试:

```

1 package cn.sitedev.objectadapter;
2
3 public class ObjectAdapterTest {
4     public static void main(String[] args) {
5         DC5 dc5 = new PowerAdapter(new AC220());
6         dc5.outputDC5V();
7     }
8 }

```



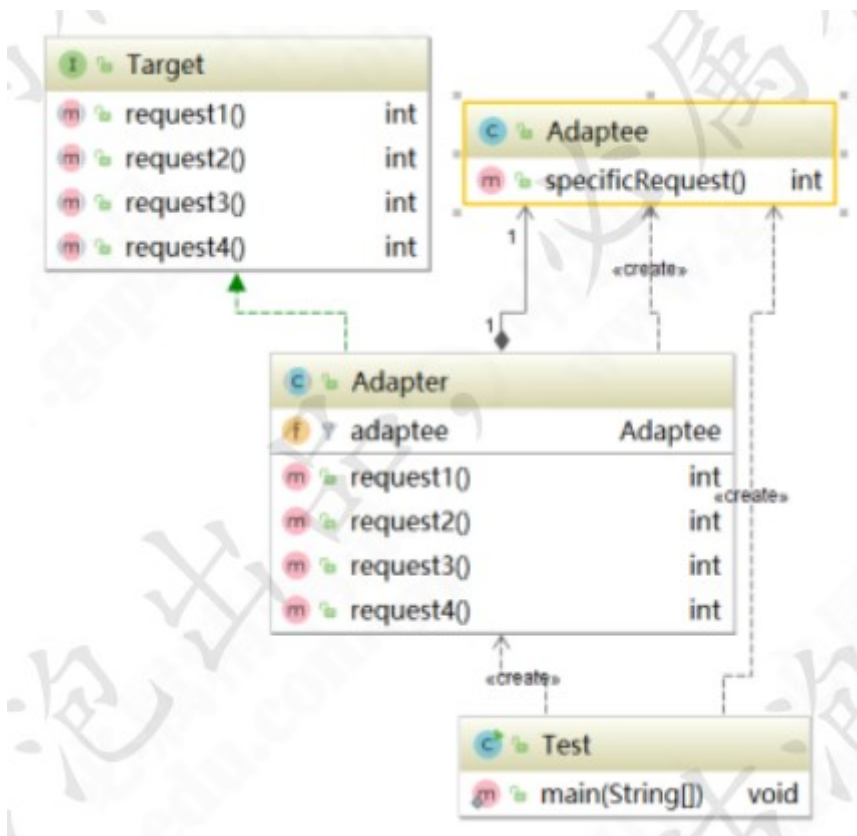
```

Run: ObjectAdapterTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
输出电压:220V
使用PowerAdapter输入AC220V, 输出DC5V
Process finished with exit code 0

```

1.4. 接口适配器

接口适配器的关注点与类适配器和对象适配器的关注点不太一样，类适配器和对象适配器着重于将系统存在的一个角色（Adaptee）转化成目标接口（Target）所需内容，而接口适配器的使用场景是解决接口方法过多，如果直接实现接口，那么类会多出许多空实现的方法，类显得很臃肿。此时，使用接口适配器就能让我们只实现我们需要的接口方法，目标更清晰。



接口适配器的主要原理就是利用抽象类实现接口，并且空实现接口众多方法。下面我们来接口适配器的源码实现，首先创建 Target角色DC类：

```

1 package cn.sitedev.interfaceadapter;
2
3 public interface DC {
4     int output5V();
5
6     int output12V();
7
8     int output24V();
9
10    int output36V();
11 }

```

创建 Adaptee角色AC220类：

```

1 package cn.sitedev.interfaceadapter;
2
3 public class AC220 {
4     public int output220V() {
5         int output = 220;

```

```
6         System.out.println("输出电压:" + output + "V");
7         return output;
8     }
9 }
```

创建 Adapter角色PowerAdapter类：

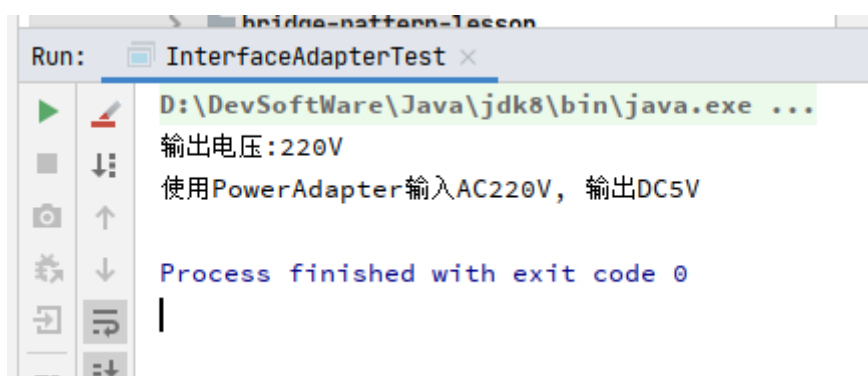
```
1 package cn.sitedev.interfaceadapter;
2
3 public class PowerAdapter implements DC {
4     private AC220 ac220;
5
6     public PowerAdapter(AC220 ac220) {
7         this.ac220 = ac220;
8     }
9
10    @Override
11    public int output5V() {
12        int adapterInput = ac220.output220V();
13        int adapterOutput = adapterInput / 44;
14        System.out.println("使用PowerAdapter输入AC" + adapterInput + "V, 输出DC" + adap
15        return adapterOutput;
16    }
17
18    @Override
19    public int output12V() {
20        return 0;
21    }
22
23    @Override
24    public int output24V() {
25        return 0;
26    }
27
28    @Override
29    public int output36V() {
30        return 0;
31    }
32 }
```

客户端代码：


```

1 package cn.sitedev.interfaceadapter;
2
3 public class InterfaceAdapterTest {
4     public static void main(String[] args) {
5         DC dc = new PowerAdapter(new AC220());
6         dc.output5V();
7     }
8 }

```



1.5. 重构第三方登陆自由适配的业务场景

下面我们来一个实际的业务场景，利用适配模式来解决实际问题。年纪稍微大一点的小伙伴一定经历过这样一个过程。我们很早以前开发的老系统应该都有登录接口，但是随着业务的发展和社会的进步，单纯地依赖用户名密码登录显然不能满足用户需求了。现在，我们大部分系统都已经支持多种登录方式，如QQ登录、微信登录、手机登录、微博登录等等，同时保留用户名密码的登录方式。虽然登录形式丰富了，但是登录后的处理逻辑可以不必改，同样是将登录状态保存到session，遵循开闭原则。首先创建统一的返回结果ResultMsg类：

```

1 package cn.sitedev.login;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class ResultMsg {
11     private int code;
12     private String msg;
13     private Object data;

```

```
14 }
```

假设老系统的登录逻辑 PassportService :

```
1 package cn.sitedev.login;
2
3 public class PassportService {
4     /**
5      * 注册
6      *
7      * @param username
8      * @param password
9      * @return
10    */
11    public ResultMsg regist(String username, String password) {
12        return new ResultMsg(200, "注册成功", new Member());
13    }
14
15    /**
16     * 登陆
17     *
18     * @param username
19     * @param password
20     * @return
21    */
22    public ResultMsg login(String username, String password) {
23        return null;
24    }
25 }
```

为了遵循开闭原则，老系统的代码我们不会去修改。那么下面开启代码重构之路，先创建Member类：

```
1 package cn.sitedev.login;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data
```

```

8 @AllArgsConstructor
9 @NoArgsConstructor
10 public class Member {
11     private String username;
12     private String password;
13     private String mid;
14     private String info;
15 }

```

运行非常稳定的代码我们不去改动，首先创建Target角色IPassportForThird接口：

```

1 package cn.sitedev.login;
2
3 public interface IPassportForThird {
4     ResultMsg login4QQ(String openid);
5
6     ResultMsg login4Wechat(String openid);
7
8     ResultMsg login4Token(String token);
9
10    ResultMsg login4Telephone(String phone, String code);
11 }

```

然后创建适配器Adapter 角色实现兼容，创建一个新的类PassportForThirdAdapter 继承原来的逻辑：

```

1 package cn.sitedev.login;
2
3 public class PassportForThirdAdapter extends PassportService implements IPassportForThird {
4     @Override
5     public ResultMsg login4QQ(String openid) {
6         return login4Regist(openid, null);
7     }
8
9     @Override
10    public ResultMsg login4Wechat(String openid) {
11        return login4Regist(openid, null);
12    }
13
14    @Override

```

```

15     public ResultMsg login4Token(String token) {
16         return login4Regist(token, null);
17     }
18
19     @Override
20     public ResultMsg login4Telephone(String phone, String code) {
21         return login4Regist(phone, null);
22     }
23
24     private ResultMsg login4Regist(String username, String password) {
25         if (password == null) {
26             password = "THIRD_EMPTY";
27         }
28         super.regist(username, password);
29         return super.login(username, password);
30     }
31 }

```

客户端测试代码：

```

1 package cn.sitedev.login;
2
3 public class LoginTest {
4     public static void main(String[] args) {
5         PassportForThirdAdapter adapter = new PassportForThirdAdapter();
6         adapter.login("sitedev", "sitedev");
7         adapter.login4QQ("sitedev");
8         adapter.login4Wechat("sitedev");
9     }
10 }

```

通过这么一个简单的适配，完成了代码兼容。当然，我们代码还可以更加优雅，根据不同的登录方式，创建不同的Adapter。首先，创建ILoginAdapter接口：

```

1 package cn.sitedev.login2;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6

```

```

7  @Data
8  @AllArgsConstructor
9  @NoArgsConstructor
10 public class ResultMsg {
11     private int code;
12     private String msg;
13     private Object data;
14 }
15 ///////////////////////////////////////////////////
16 package cn.sitedev.login2;
17
18 import lombok.AllArgsConstructor;
19 import lombok.Data;
20 import lombok.NoArgsConstructor;
21
22 @Data
23 @AllArgsConstructor
24 @NoArgsConstructor
25 public class Member {
26     private String username;
27     private String password;
28     private String mid;
29     private String info;
30 }
31 ///////////////////////////////////////////////////
32 package cn.sitedev.login2;
33
34 public class PassportService {
35     /**
36      * 注册
37      *
38      * @param username
39      * @param password
40      * @return
41      */
42     public ResultMsg regist(String username, String password) {
43         return new ResultMsg(200, "注册成功", new Member());
44     }
45
46     /**
47      * 登陆
48      *
49      * @param username

```

```

50     * @param password
51     * @return
52     */
53     public ResultMsg login(String username, String password) {
54         return null;
55     }
56 }
57 ///////////////////////////////////////////////////
58 package cn.sitedev.login2;
59
60 public interface IPassportForThird {
61     ResultMsg login4QQ(String openid);
62
63     ResultMsg login4Wechat(String openid);
64
65     ResultMsg login4Token(String token);
66
67     ResultMsg login4Telephone(String phone, String code);
68 }
69 ///////////////////////////////////////////////////
70 package cn.sitedev.login2;
71
72 public interface ILoginAdapter {
73     boolean support(Object object);
74
75     ResultMsg login(String id, Object adapter);
76 }

```

然后，创建一个抽象类AbstractAdapter继承PassportService原有的功能，同时实现ILoginAdapter 接口，然后分别实现不同的登录适配，QQ登录LoginForQQAdapter：

```

1 package cn.sitedev.login2;
2
3 public class LoginForQQAdapter extends AbstractAdapter {
4     @Override
5     public boolean support(Object adapter) {
6         return adapter instanceof LoginForQQAdapter;
7     }
8
9     @Override
10    public ResultMsg login(String id, Object adapter) {

```

```

11         if (!support(adapter)) {
12             return null;
13         }
14         return super.login4Regist(id, null);
15     }
16 }

```

手机号登录 LoginForTelAdapter :

```

1 package cn.sitedev.login2;
2
3 public class LoginForTelAdapter extends AbstractAdapter {
4     @Override
5     public boolean support(Object adapter) {
6         return adapter instanceof LoginForTelAdapter;
7     }
8
9     @Override
10    public ResultMsg login(String id, Object adapter) {
11        return super.login4Regist(id, null);
12    }
13 }

```

Token 自动登录 LoginForTokenAdapter :

```

1 package cn.sitedev.login2;
2
3 public class LoginForTokenAdapter extends AbstractAdapter {
4     @Override
5     public boolean support(Object adapter) {
6         return adapter instanceof LoginForTokenAdapter;
7     }
8
9     @Override
10    public ResultMsg login(String id, Object adapter) {
11        return super.login4Regist(id, null);
12    }
13 }

```

微信登录 LoginForWechatAdapter :

```
1 package cn.sitedev.login2;
2
3 public class LoginForWechatAdapter extends AbstractAdapter {
4     @Override
5     public boolean support(Object adapter) {
6         return adapter instanceof LoginForWechatAdapter;
7     }
8
9     @Override
10    public ResultMsg login(String id, Object adapter) {
11        return login4Regist(id, null);
12    }
13 }
```

然后，创建适配器PasswordForThirdAdapter类，实现目标接口IPassportForThird 完成兼容

```
1 package cn.sitedev.login2;
2
3 public class PasswordForThirdAdapter implements IPassportForThird {
4     @Override
5     public ResultMsg login4QQ(String openid) {
6         return processLogin(openid, LoginForQQAdapter.class);
7     }
8
9     @Override
10    public ResultMsg login4Wechat(String openid) {
11        return processLogin(openid, LoginForWechatAdapter.class);
12    }
13
14    @Override
15    public ResultMsg login4Token(String token) {
16        return processLogin(token, LoginForTokenAdapter.class);
17    }
18
19    @Override
20    public ResultMsg login4Telephone(String phone, String code) {
21        return processLogin(phone, LoginForTelAdapter.class);
22    }
23 }
```

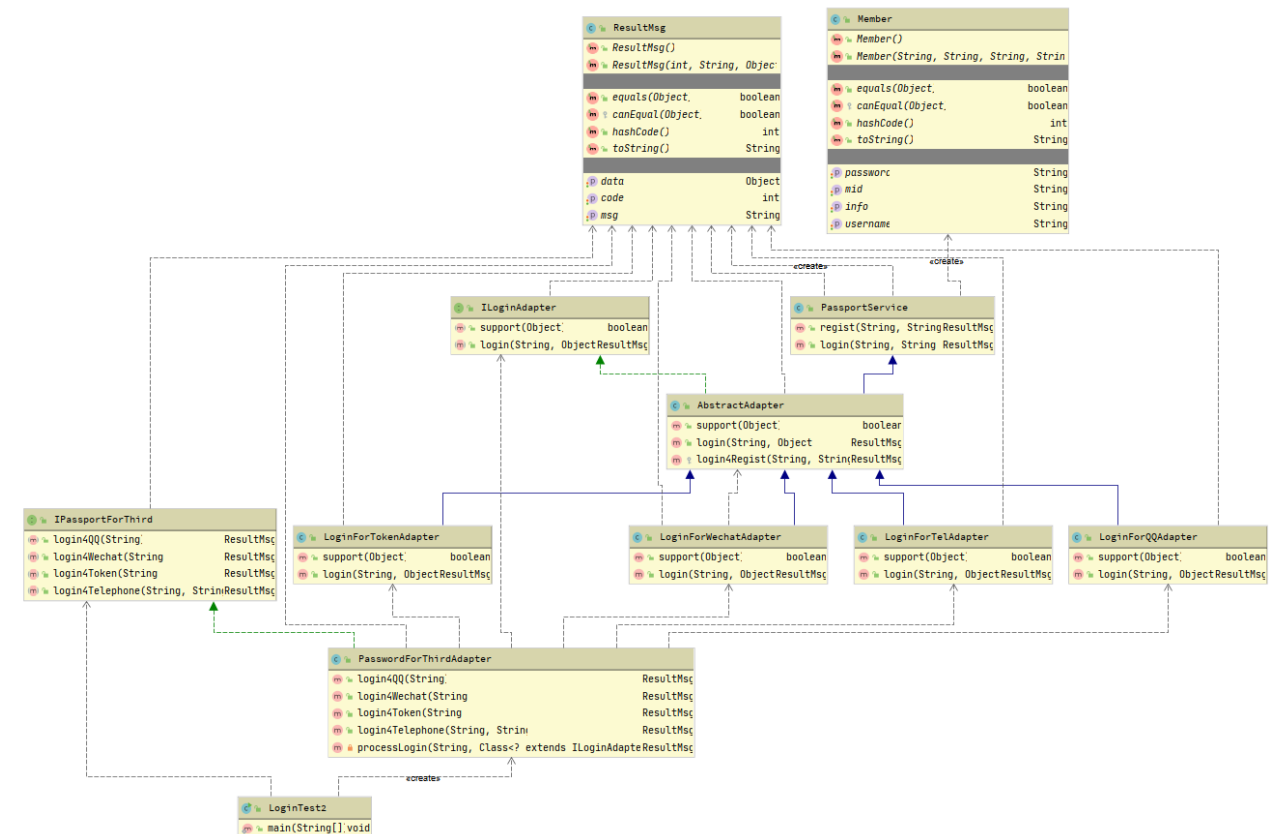
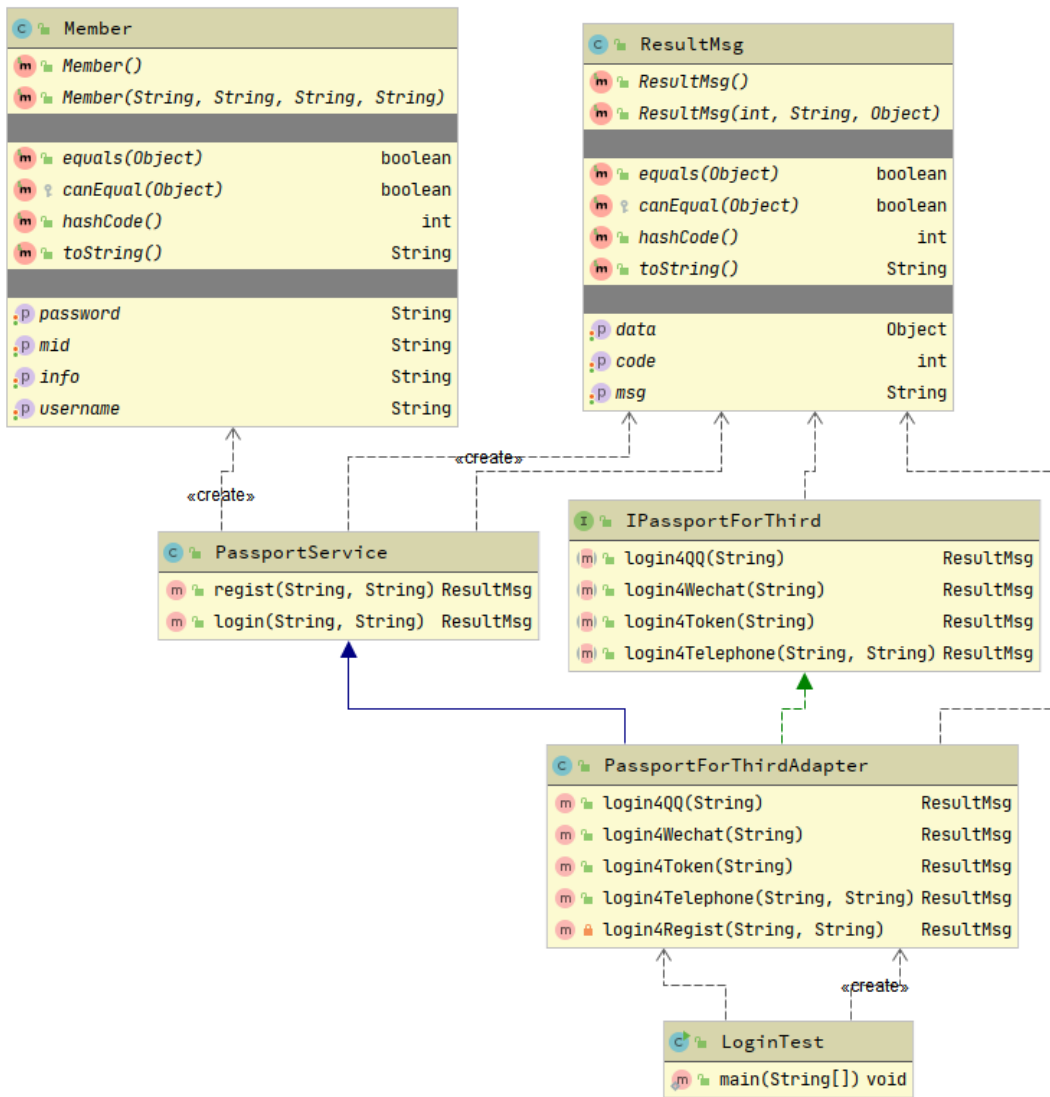


```
23
24     private ResultMsg processLogin(String id, Class<? extends ILoginAdapter> clazz) {
25         try {
26             ILoginAdapter adapter = clazz.newInstance();
27             if (adapter.support(adapter)) {
28                 return adapter.login(id, adapter);
29             }
30         } catch (IllegalAccessException e) {
31             e.printStackTrace();
32         } catch (InstantiationException e) {
33             e.printStackTrace();
34         }
35         return null;
36     }
37 }
```

客户端测试代码：

```
1 package cn.sitedev.login2;
2
3 public class LoginTest2 {
4     public static void main(String[] args) {
5         IPassportForThird adapter = new PasswordForThirdAdapter();
6         adapter.login4QQ("sitedev");
7     }
8 }
```

最后，来看一下类图：



至此，我们在遵循开闭原则的前提下，完整地实现了一个兼容多平台登录的业务场景。当然，我目前的设计也并不完美，仅供参考，感兴趣的小伙伴可以继续完善这段代码。例如适配器中的参数目前是写死为String，改为 `Object[]` 应该更合理。

学习到这里，相信小伙伴会有一个疑问了：适配器模式跟策略模式好像区别不大？在这里我要强调一下，适配器模式主要解决的是功能兼容问题，单场景适配大家可能不会和策略模式有对比。但多场景适配大家产生联想和混淆了。其实，大家有没有发现一个细节，我给每个适配器都加上了一个 `support()` 方法，用来判断是否兼容，`support()` 方法的参数也是 `Object` 的，而 `supoort()` 来自于接口。适配器的实现逻辑并不依赖于接口，我们完全可以将 `ILoginAdapter` 接口去掉。而加上接口，只是为了代码规范。上面的代码可以说是策略模式、简单工厂模式和适配器模式的综合运用。

1.6. 适配器模式在源码中的体现

Spring中适配器模式也应用得非常广泛，例如：SpringAOP中的 `AdvisorAdapter` 类，它有三个实现类 `MethodBeforeAdviceAdapter`、`AfterReturningAdviceAdapter` 和 `ThrowsAdviceAdapter`，先来看顶层接口 `AdvisorAdapter` 的源代码：

```
1 package org.springframework.aop.framework.adapter;
2
3 import org.aopalliance.aop.Advice;
4 import org.aopalliance.intercept.MethodInterceptor;
5
6 import org.springframework.aop.Advisor;
7
8 public interface AdvisorAdapter {
9
10     boolean supportsAdvice(Advice advice);
11
12     MethodInterceptor getInterceptor(Advisor advisor);
13 }
```

再看 `MethodBeforeAdviceAdapter` 类：

```
1 import java.io.Serializable;
2
3 import org.aopalliance.aop.Advice;
4 import org.aopalliance.intercept.MethodInterceptor;
5
6 import org.springframework.aop.Advisor;
7 import org.springframework.aop.MethodBeforeAdvice;
8
```

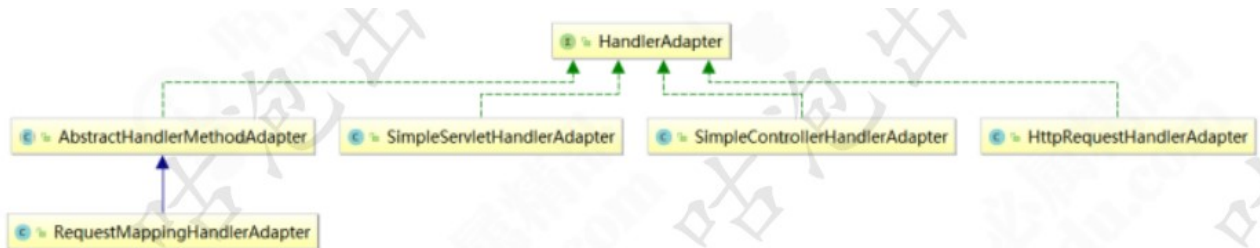
```

9  /**
10 * Adapter to enable {@link org.springframework.aop.MethodBeforeAdvice}
11 * to be used in the Spring AOP framework.
12 *
13 * @author Rod Johnson
14 * @author Juergen Hoeller
15 */
16 @SuppressWarnings("serial")
17 class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {
18
19     @Override
20     public boolean supportsAdvice(Advice advice) {
21         return (advice instanceof MethodBeforeAdvice);
22     }
23
24     @Override
25     public MethodInterceptor getInterceptor(Advisor advisor) {
26         MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
27         return new MethodBeforeAdviceInterceptor(advice);
28     }
29
30 }

```

其它两个类我这里就不把代码贴出来了。Spring会根据不同的AOP配置来确定使用对应的Advice，跟策略模式不同的一个方法可以同时拥有多个Advice。

下面再来看一个SpringMVC中的HandlerAdapter类，它也有多个子类，类图如下：



其适配调用的关键代码还是在DispatcherServlet的doDispatch()方法中，下面我们还是来看源码：

```

1     protected void doDispatch(HttpServletRequest request, HttpServletResponse response)
2         HttpServletRequest processedRequest = request;
3         HandlerExecutionChain mappedHandler = null;
4         boolean multipartRequestParsed = false;
5
6         WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

```

```

7
8     try {
9         ModelAndView mv = null;
10        Exception dispatchException = null;
11
12        try {
13            processedRequest = checkMultipart(request);
14            multipartRequestParsed = (processedRequest != request);
15
16            // Determine handler for the current request.
17            mappedHandler = getHandler(processedRequest);
18            if (mappedHandler == null) {
19                noHandlerFound(processedRequest, response);
20                return;
21            }
22
23            // Determine handler adapter for the current request.
24            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());
25
26            // Process last-modified header, if supported by the handler.
27            String method = request.getMethod();
28            boolean isGet = "GET".equals(method);
29            if (isGet || "HEAD".equals(method)) {
30                long lastModified = ha.getLastModified(request, mappedHandler.getHa
31                if (new ServletWebRequest(request, response).checkNotModified(lastM
32                    return;
33            }
34        }
35
36        if (!mappedHandler.applyPreHandle(processedRequest, response)) {
37            return;
38        }
39
40        // Actually invoke the handler.
41        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
42
43        if (asyncManager.isConcurrentHandlingStarted()) {
44            return;
45        }
46
47        applyDefaultViewName(processedRequest, mv);
48        mappedHandler.applyPostHandle(processedRequest, response, mv);
49    }

```

```

50         catch (Exception ex) {
51             dispatchException = ex;
52         }
53         catch (Throwable err) {
54             // As of 4.3, we're processing Errors thrown from handler methods as we
55             // making them available for @ExceptionHandler methods and other scenar
56             dispatchException = new NestedServletException("Handler dispatch failed
57         }
58         processDispatchResult(processedRequest, response, mappedHandler, mv, dispat
59     }
60     catch (Exception ex) {
61         triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
62     }
63     catch (Throwable err) {
64         triggerAfterCompletion(processedRequest, response, mappedHandler,
65             new NestedServletException("Handler processing failed", err));
66     }
67     finally {
68         if (asyncManager.isConcurrentHandlingStarted()) {
69             // Instead of postHandle and afterCompletion
70             if (mappedHandler != null) {
71                 mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest,
72             }
73         }
74         else {
75             // Clean up any resources used by a multipart request.
76             if (multipartRequestParsed) {
77                 cleanupMultipart(processedRequest);
78             }
79         }
80     }
81 }

```

在doDispatch()方法中调用了getHandlerAdapter()方法，来看代码：

```

1     protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException
2     {
3         if (this.handlerAdapters != null) {
4             for (HandlerAdapter adapter : this.handlerAdapters) {
5                 if (adapter.supports(handler)) {
6                     return adapter;
7                 }
8             }
9         }
10    }

```

```

7         }
8     }
9     throw new ServletException("No adapter for handler [" + handler +
10        "]: The DispatcherServlet configuration needs to include a HandlerAdapt
11    }

```

在getHandlerAdapter()方法中循环调用了supports ()方法判断是否兼容，循环迭代集合中的Adapter又是在初始化时早已赋值。这里我们不再深入，后面的源码专题中还会继续讲解。

1.7. 适配器模式和装饰器模式对比

装饰器和适配器模式都是包装模式（Wrapper Pattern），装饰器也是一种特殊的代理模式。

	装饰器模式	适配器模式
形式	是一种非常特别的适配器模式	没有层级关系，装饰器模式有层级关系
定义	装饰器和被装饰器都实现同一个接口，	适配器和被适配者没有必然的联系，通常是

	主要目的是为了扩展之后依旧保留 OOP 关系	采用继承或代理的形式进行包装
关系	满足 is-a 的关系	满足 has-a 的关系
功能	注重覆盖、扩展	注重兼容、转换
设计	前置考虑	后置考虑

1.8. 适配器模式的优缺点

优点：

- 1、能提高类的透明性和复用，现有的类复用但不需要改变。
- 2、目标类和适配器类解耦，提高程序的扩展性。
- 3、在很多业务场景中符合开闭原则。

缺点：

- 1、适配器编写过程需要全面考虑，可能会增加系统的复杂性。
- 2、增加代码阅读难度，降低代码可读性，过多使用适配器会使系统代码变得凌乱。

2. 桥接模式

桥接模式（Bridge Pattern）也称为桥梁模式、接口（Interface）模式或柄体（Handle and Body）模式，是将抽象部分与它的具体实现部分分离，使它们都可以独立地变化，属于结构型模式。

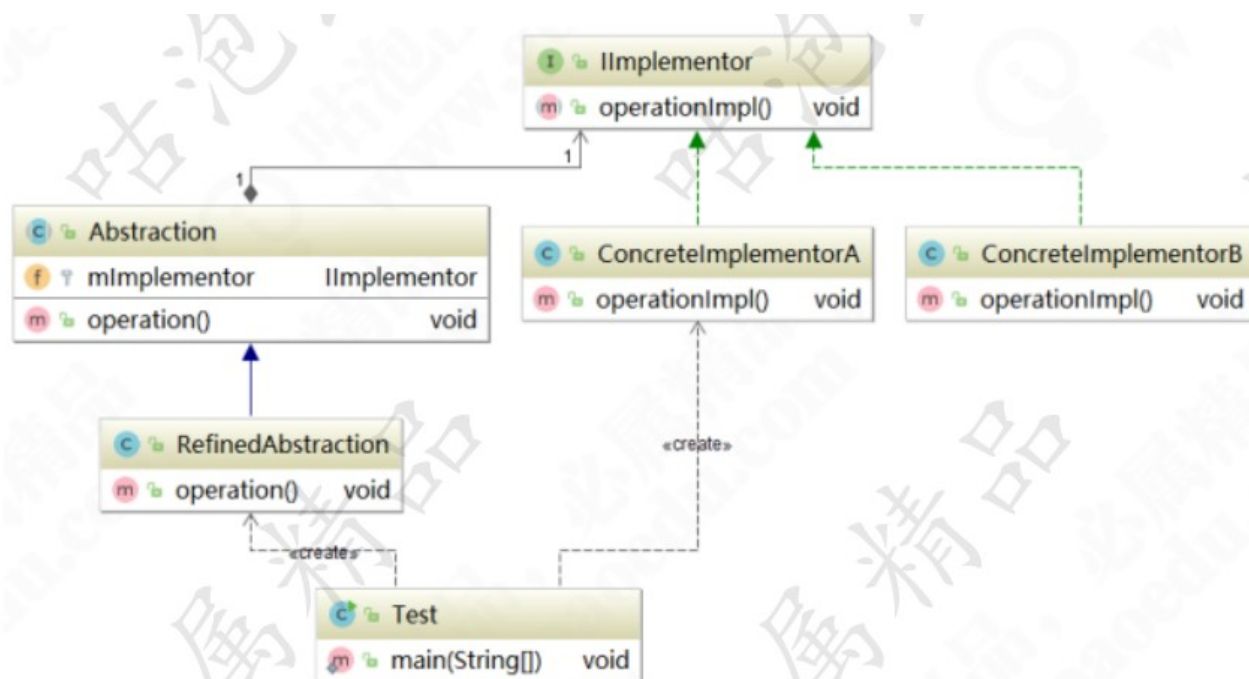
原文 : Decouple an abstraction from its implementation so that the two can vary independently.

解释 : 解耦抽象和实现，使得两者可以独立的变化。

桥接模式主要目的是通过组合的方式建立两个类之间的联系，而不是继承。但又类似于多重继承方案，但是多重继承方案往往违背了类的单一职责原则，其复用性比较差，桥接模式是比多重继承更好的替代方案。桥接模式的核心在于解耦抽象和实现。

注：此处的抽象并不是指抽象类或接口这种高层概念，实现也不是继承或接口实现。抽象与实现其实指的是两种独立变化的维度。其中，抽象包含实现，因此，一个抽象类的变化可能涉及到多种维度的变化导致的。

我们来看下桥接模式的通用UML类图：



从UML类图中，我们可以看到，桥接模式主要包含四种角色：

抽象（Abstraction）：该类持有一个对实现角色的引用，抽象角色中的方法需要实现角色来实现。

抽象角色一般为抽象类（构造函数规定子类要传入一个实现对象）；

修正抽象（RefinedAbstraction）：Abstraction的具体实现，对Abstraction的方法进行完善和扩展；

实现（Implementor）：确定实现维度的基本操作，提供给Abstraction使用。该类一般为接口或抽象类；

具体实现（ConcretImplementor）：Implementor的具体实现。

2.1. 桥接模式的通用写法

创建抽象角色Abstraction类：

```
1 package cn.sitedev.general;
2
```



```

3 public class Abstraction {
4     protected IImplementor iImplementor;
5
6     public Abstraction(IImplementor iImplementor) {
7         this.iImplementor = iImplementor;
8     }
9
10    public void operation() {
11        this.iImplementor.operationImpl();
12    }
13 }

```

创建修正抽象角色RefinedAbstraction类：

```

1 package cn.sitedev.general;
2
3 public class RefinedAbstraction extends Abstraction {
4
5     public RefinedAbstraction(IImplementor iImplementor) {
6         super(iImplementor);
7     }
8
9     @Override
10    public void operation() {
11        super.operation();
12        System.out.println("RefinedAbstraction.operation");
13    }
14 }

```

创建实现角色Implementor类：

```

1 package cn.sitedev.general;
2
3 /**
4  * 抽象实现
5  */
6 public interface IImplementor {
7     void operationImpl();
8 }

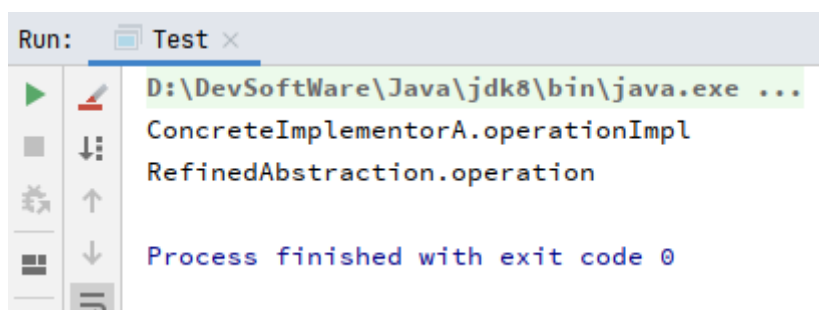
```

创建具体实现ConcreteImplementorA类：

```
1 package cn.sitedev.general;
2
3 public class ConcreteImplementorA implements IImplementor {
4     @Override
5     public void operationImpl() {
6         System.out.println("ConcreteImplementorA.operationImpl");
7     }
8 }
```

客户端测试代码：

```
1 package cn.sitedev.general;
2
3 public class Test {
4     public static void main(String[] args) {
5         // 来一个实现化角色
6         IImplementor iImplementor = new ConcreteImplementorA();
7         // 来一个抽象化角色，聚合实现
8         Abstraction abstraction = new RefinedAbstraction(iImplementor);
9         // 执行操作
10        abstraction.operation();
11    }
12 }
```



2.2. 桥接模式的应用场景

当一个类内部具备两种或多种变化维度时，使用桥接模式可以解耦这些变化的维度，使高层代码架构稳定。桥接模式适用于以下几种业务场景：

- 1、在抽象和具体实现之间需要增加更多的灵活性的场景。

- 2、一个类存在两个（或多个）独立变化的维度，而这两个（或多个）维度都需要独立进行扩展。
- 3、不希望使用继承，或因为多层继承导致系统类的个数剧增。

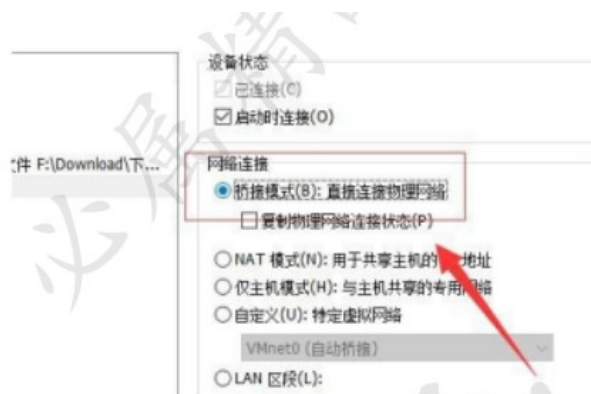
注：桥接模式的一个常用使用场景就是为了替换继承。我们知道，继承拥有很多优点，比如抽象，封装，多态等，父类封装共性，子类实现特性。继承可以很好地帮助我们实现代码复用（封装）的功能，但是同时，这也是继承的一大缺点。因为父类拥有的方法，子类也会继承得到，无论子类需不需要，这说明了继承具备强侵入性（父类代码侵入子类），同时会导致子类臃肿...因此，在设计模式中，有一个原则为：优先使用组合/聚合的方式，而不是继承。

但是，设计模式是死的，人是活的。很多时候，你分不清该使用继承还是组合/聚合或其他方式等，可以从现实语义进行思考，因为软件（代码）最终还是提供给现实生活中的人用的，是服务于人类社会的，软件（代码）是具备现实场景的，你单从纯代码角度无法看清问题时，现实角度可能会给你提供更加开阔的思路。

在生活场景中的桥接模式也随处可见，比如连接起两个空间维度的桥，比如链接虚拟网络与真实网络的连接。



连接起两个空间维度的桥

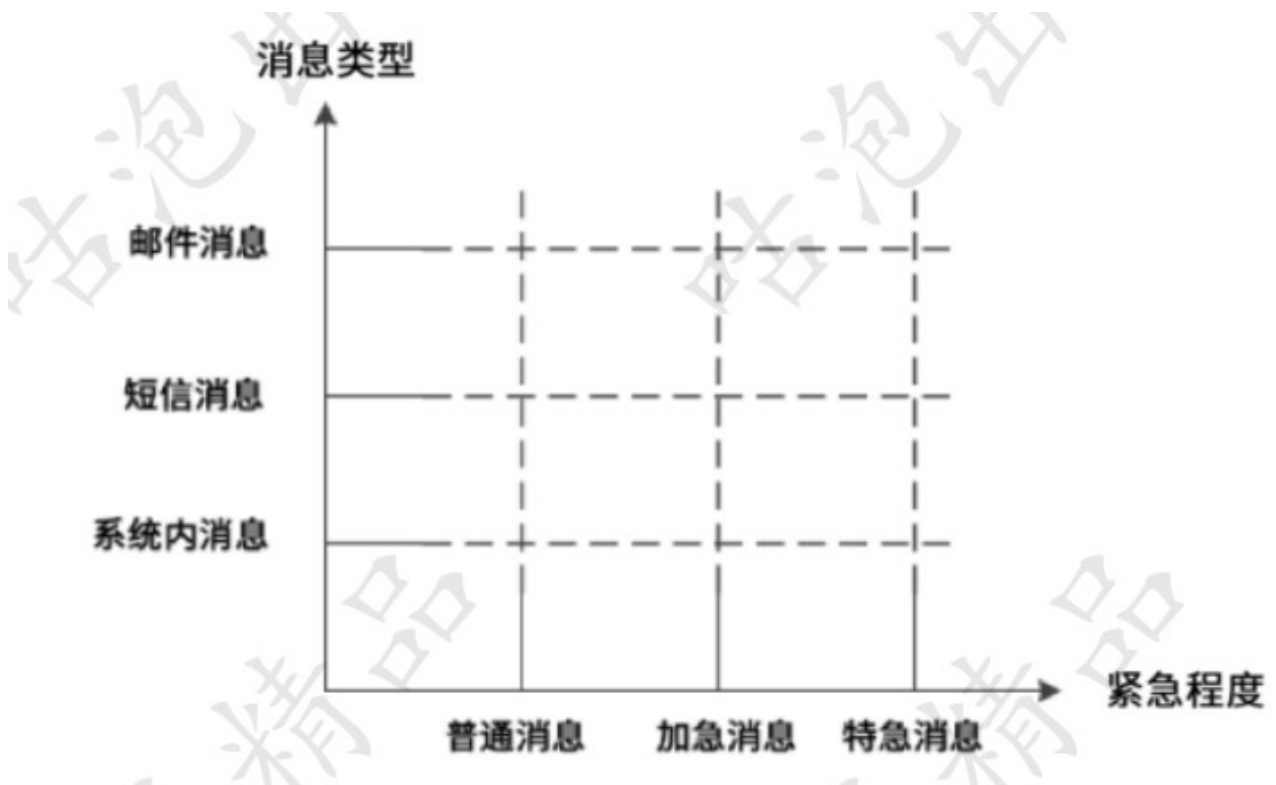


虚拟网络与真实网络的桥接

2.3. 桥接模式在业务场景中的应用

举个例子，我们平时办公的时候经常通过发邮件消息、短信消息或者系统内消息和同事进行沟通。

尤其是在走一些审批流程的时候，我们需要记录这些过程以备查。我们根据消息的类别来划分的话，可以分为邮件消息、短信消息和系统内消息。但是，根据消息的紧急程度来划分的话，可以分为普通消息、紧急消息和特急消息。显然，整个消息系统可以划分为两个维度，如下图：



如果，我们用继承的话情况就复杂了，而且也不利于扩展。邮件信息可以是普通的，也可以是紧急的；短信消息可以是普通的，也可以是紧急的。下面我们用桥接模式来解决这个问题：

首先创建一个IMessage接口担任桥接的角色：

```
1 package cn.sitedev.message;
2
3 /**
4  * 实现消息发送的统一接口
5  */
6 public interface IMessage {
7     /**
8      * 要发送的消息的内容和接收者
9      *
10     * @param message
11     * @param toUser
12     */
13     void send(String message, String toUser);
14
15 }
```

创建邮件消息实现EmailMessage类：

```
1 package cn.sitedev.message;
```

```

2
3 /**
4  * 邮件短消息的实现类
5  */
6 public class EmailMessage implements IMessage {
7     @Override
8     public void send(String message, String toUser) {
9         System.out.printf("使用邮件短消息的方法，发送消息%s 给 %s \n", message, toUser)
10    }
11 }

```

创建手机短信实现SmsMessage类：

```

1 package cn.sitedev.message;
2
3 /**
4  * 系统内短消息的实现类
5  */
6 public class SmsMessage implements IMessage {
7     @Override
8     public void send(String message, String toUser) {
9         System.out.printf("使用系统内短消息的方法，发送消息 %s 给 %s \n", message, toUs
10    }
11 }

```

然后，再创建桥接抽象角色AbstractMessage类：

```

1 package cn.sitedev.message;
2
3 /**
4  * 抽象消息类
5  */
6 public abstract class AbstractMessage {
7     /**
8      * 持有一个实现部分的对象
9      */
10    private IMessage message;
11
12    /**
13      * 构造方法，传入实现部分的对象

```

```

14      *
15      * @param message
16      */
17      public AbstractMessage(IMessage message) {
18          this.message = message;
19      }
20
21      public void sendMessage(String message, String toUser) {
22          this.message.send(message, toUser);
23      }
24  }

```

创建具体实现普通消息NormalMessage类：

```

1  package cn.sitedev.message;
2
3  /**
4   * 普通消息类
5   */
6  public class NormalMessage extends AbstractMessage {
7
8      /**
9       * 构造方法，传入实现部分的对象
10      *
11      * @param message
12      */
13      public NormalMessage(IMessage message) {
14          super(message);
15      }
16
17      @Override
18      public void sendMessage(String message, String toUser) {
19          // 对于普通消息，直接调用父类方法，发送消息即可
20          super.sendMessage(message, toUser);
21      }
22  }

```

创建具体实现紧急消息UrgencyMessage类：

```

1  package cn.sitedev.message;

```

```

2
3 /**
4  * 加急消息类
5  */
6 public class UrgencyMessage extends AbstractMessage {
7
8     /**
9      * 构造方法，传入实现部分的对象
10     *
11     * @param message
12     */
13     public UrgencyMessage(IMessage message) {
14         super(message);
15     }
16
17     @Override
18     public void sendMessage(String message, String toUser) {
19         message = "加急" + message;
20         super.sendMessage(message, toUser);
21     }
22
23     /**
24     * 扩展它自己的功能，监控某个消息的处理状态
25     *
26     * @param messageId
27     * @return
28     */
29     public Object watch(String messageId) {
30         // 根据给出的消息编码(messageId) 查询消息的处理状态
31         // 组织成监控的处理状态，然后返回
32         return null;
33     }
34 }

```

编写客户端测试代码：

```

1 package cn.sitedev.message;
2
3 public class MessageTest {
4     public static void main(String[] args) {
5         IMessage message = new SmsMessage();

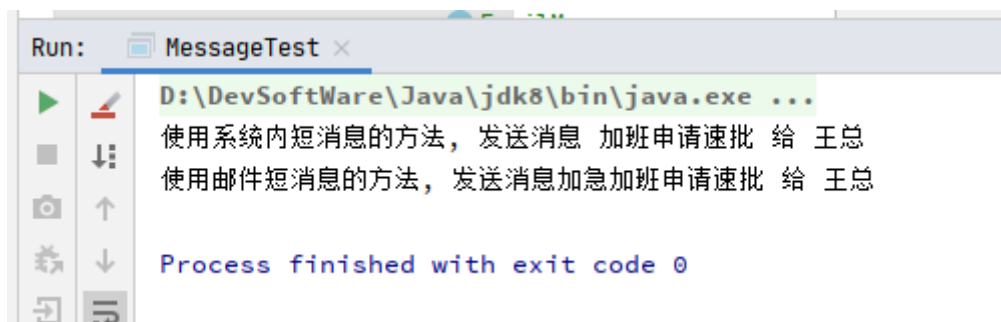
```

```

6      AbstractMessage abstractMessage = new NormalMessage(message);
7      abstractMessage.sendMessage("加班申请速批", "王总");
8
9      message = new EmailMessage();
10     abstractMessage = new UrgencyMessage(message);
11     abstractMessage.sendMessage("加班申请速批", "王总");
12 }
13 }

```

运行结果如下：



上面的案例中，我们采用桥接模式解耦了“消息类型”和“消息紧急程度”这两个独立变化的维度。

后续如果有更多的消息类型，比如微信、钉钉等，那么直接新建一个类继承IMessage即可；如果是紧急程度需要新增，那么同样只需新建一个类实现AbstractMessage类即可。

2.4. 桥接模式在源码中的应用

大家非常熟悉的JDBC API，其中有一个Driver 类就是桥接对象。我们都知道，我们在使用的时候通过Class.forName()方法可以动态加载各个数据库厂商实现的Driver类。具体客户端应用代码如下，以MySQL的实现为例：

```

//1.加载驱动
Class.forName("com.mysql.jdbc.Driver"); //反射机制加载驱动类
// 2.获取连接 Connection
// 主机:端口号/数据库名
Connection conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/test","root","root");
// 3.得到执行 sql 语句的对象 Statement
Statement stmt = conn.createStatement();
// 4.执行 sql 语句，并返回结果
ResultSet rs=stmt.executeQuery("select *from table");

```

首先，我们来看一下Driver 接口的定义：

```

1 package java.sql;
2
3 import java.util.logging.Logger;
4
5 public interface Driver {

```



```

6
7     Connection connect(String url, java.util.Properties info)
8         throws SQLException;
9
10    boolean acceptsURL(String url) throws SQLException;
11
12    DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info)
13        throws SQLException;
14
15    int getMajorVersion();
16
17    int getMinorVersion();
18
19    boolean jdbcCompliant();
20
21    public Logger getParentLogger() throws SQLFeatureNotSupportedException;
22 }

```

Driver在JDBC中并没有做任何实现，具体的功能实现由各厂商完成，我们以MySQL的实现为例。

```

1 public class Driver extends NonRegisteringDriver implements java.sql.Driver {
2     // ~ Static fields/initializers
3     // -----
4
5     //
6     // Register ourselves with the DriverManager
7     //
8     static {
9         try {
10             java.sql.DriverManager.registerDriver(new Driver());
11         } catch (SQLException E) {
12             throw new RuntimeException("Can't register driver!");
13         }
14     }
15     ...

```

当我们执行Class.forName ("com.mysql.jdbc.Driver") 方法的时候，就会执行com.mysql.jdbc.Driver这个类的静态块中的代码。而静态块中的代码只是调用了一下DriverManager的registerDriver()方法，然后将Driver对象注册到DriverManager中。我们可以继续跟进到DriverManager这个类中，来看相关的代码：

```
1 public class DriverManager {
2
3
4     // List of registered JDBC drivers
5     private final static CopyOnWriteArrayList<DriverInfo> registeredDrivers = new CopyOnWriteArrayList<>()
6     ...
7
8     /* Prevent the DriverManager class from being instantiated. */
9     private DriverManager(){}
10
11     ...
12
13     /**
14      * Load the initial JDBC drivers by checking the System property
15      * jdbc.properties and then use the {@code ServiceLoader} mechanism
16      */
17     static {
18         loadInitialDrivers();
19         println("JDBC DriverManager initialized");
20     }
21
22     public static synchronized void registerDriver(java.sql.Driver driver)
23         throws SQLException {
24
25         registerDriver(driver, null);
26     }
27
28     public static synchronized void registerDriver(java.sql.Driver driver,
29         DriverAction da)
30         throws SQLException {
31
32         /* Register the driver if it has not already been added to our list */
33         if(driver != null) {
34             registeredDrivers.addIfAbsent(new DriverInfo(driver, da));
35         } else {
36             // This is for compatibility with the original DriverManager
37             throw new NullPointerException();
38         }
39
40         println("registerDriver: " + driver);
41
42     }
```

在注册之前，将传过来的Driver对象，封装成了一个DriverInfo对象。接下来继续执行客户端代码的第二步，调用DriverManager的 getConnection()方法获取连接对象，我们跟进源码：

```
1 public class DriverManager {
2     ...
3     @CallerSensitive
4     public static Connection getConnection(String url,
5         java.util.Properties info) throws SQLException {
6
7         return (getConnection(url, info, Reflection.getCallerClass()));
8     }
9
10    @CallerSensitive
11    public static Connection getConnection(String url,
12        String user, String password) throws SQLException {
13        java.util.Properties info = new java.util.Properties();
14
15        if (user != null) {
16            info.put("user", user);
17        }
18        if (password != null) {
19            info.put("password", password);
20        }
21
22        return (getConnection(url, info, Reflection.getCallerClass()));
23    }
24
25    @CallerSensitive
26    public static Connection getConnection(String url)
27        throws SQLException {
28
29        java.util.Properties info = new java.util.Properties();
30        return (getConnection(url, info, Reflection.getCallerClass()));
31    }
32    private static Connection getConnection(
33        String url, java.util.Properties info, Class<?> caller) throws SQLException {
34        /*
35         * When callerCl is null, we should check the application's
36         * (which is invoking this class indirectly)
```

```

37      * classloader, so that the JDBC driver class outside rt.jar
38      * can be loaded from here.
39      */
40      ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;
41      synchronized(DriverManager.class) {
42          // synchronize loading of the correct classloader.
43          if (callerCL == null) {
44              callerCL = Thread.currentThread().getContextClassLoader();
45          }
46      }
47
48      if(url == null) {
49          throw new SQLException("The url cannot be null", "08001");
50      }
51
52      println("DriverManager.getConnection(\"" + url + "\")");
53
54      // Walk through the loaded registeredDrivers attempting to make a connection.
55      // Remember the first exception that gets raised so we can reraise it.
56      SQLException reason = null;
57
58      for(DriverInfo aDriver : registeredDrivers) {
59          // If the caller does not have permission to load the driver then
60          // skip it.
61          if(isDriverAllowed(aDriver.driver, callerCL)) {
62              try {
63                  println("    trying " + aDriver.driver.getClass().getName());
64                  Connection con = aDriver.driver.connect(url, info);
65                  if (con != null) {
66                      // Success!
67                      println("getConnection returning " + aDriver.driver.getClass().
68                          return (con);
69                  }
70              } catch (SQLException ex) {
71                  if (reason == null) {
72                      reason = ex;
73                  }
74              }
75
76          } else {
77              println("    skipping: " + aDriver.getClass().getName());
78          }
79

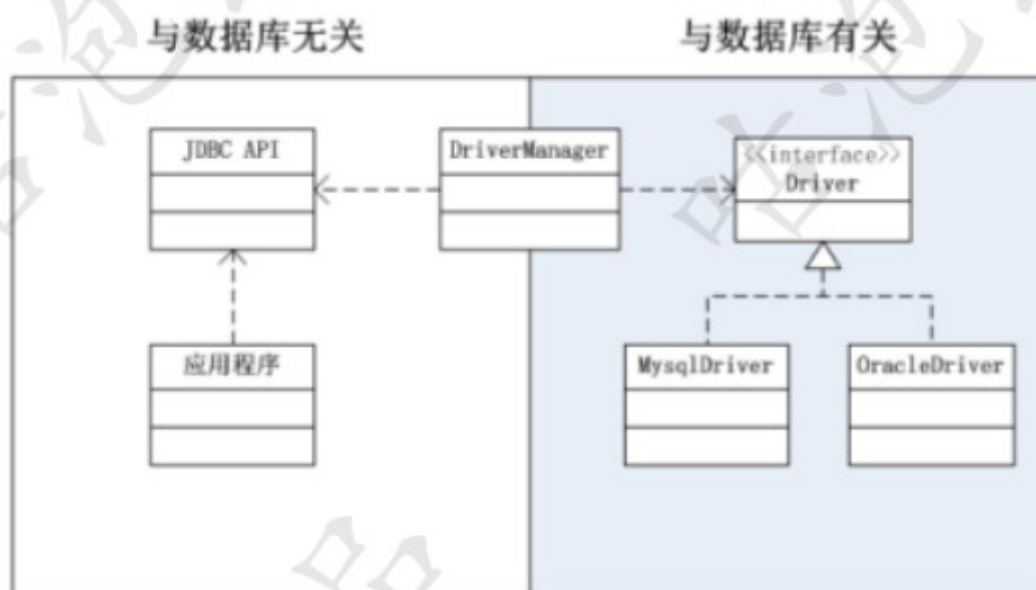
```

```

80     }
81
82     // if we got here nobody could connect.
83     if (reason != null)    {
84         println("getConnection failed: " + reason);
85         throw reason;
86     }
87
88     println("getConnection: no suitable driver found for "+ url);
89     throw new SQLException("No suitable driver found for "+ url, "08001");
90 }
91 ...

```

在getConnection()中就会调用各自厂商实现的Driver的connect()方法获得连接对象。这样的话，就巧妙地避开了使用继承，为不同的数据库提供了相同的接口。JDBC API中DriverManager就是桥，如下图所示：



2.5. 桥接模式的优缺点

通过上面的例子，我们能很好地感知到桥接模式遵循了里氏替换原则和依赖倒置原则，最终实现了开闭原则，对修改关闭，对扩展开放。这里将桥接模式的优缺点总结如下：

优点：

- 1、分离抽象部分及其具体实现部分
- 2、提高了系统的扩展性
- 3、符合开闭原则
- 4、符合合成复用原则

缺点：

- 1、增加了系统的理解与设计难度
- 2、需要正确地识别系统中两个独立变化的维度