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。

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，从而使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。

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。

两脚插转三角插

手机充电接口

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；

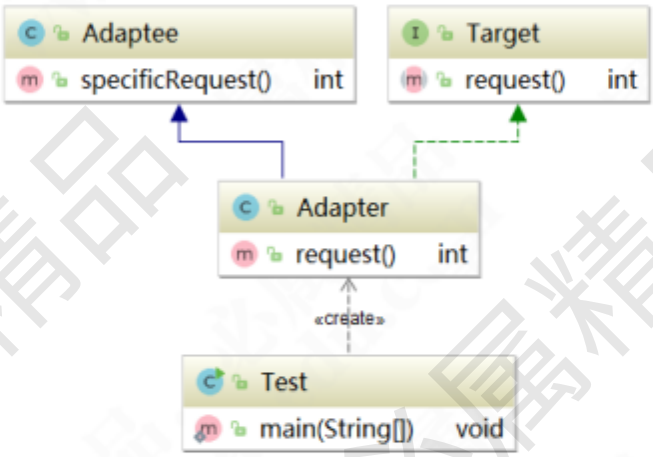
适配器模式各角色之间的关系如下：

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需求，而 Adaptee 实例符合需求；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。

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。下面我们有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：

```
public class AC220 {
    public int outputAC220V(){
        int output = 220;
        System.out.println("输出电压" + output + "V");
        return output;
    }
}
```

创建 Target 角色 DC5 接口，表示 5V 直流电的标准：

```
public interface DC5 {
    int outoupDC5V();
}
```

创建 Adapter 角色电源适配器 PowerAdapter 类：

```
public class PowerAdapter extends AC220 implements DC5 {
    public int output5V() {
```

```
        int adapterInput = super.outputAC220V();
        int adapterOutput = adapterInput / 44;
        System.out.println("使用 Adapter 输入 AC" + adapterInput + "V,输出 DC" + adapterOutput + "V");
        return adapterOutput;
    }
}
```

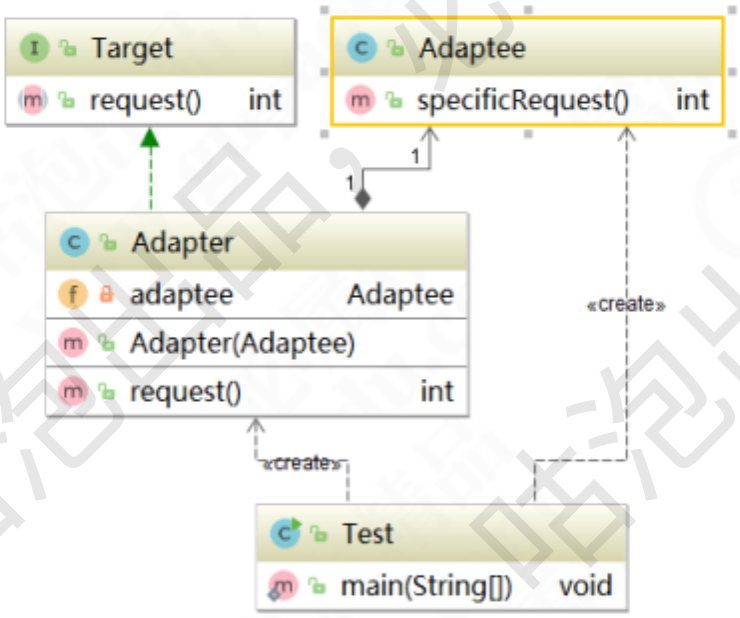
客户端测试代码：

```
    public static void main(String[] args) {
        DC5 adapter = new PowerAdapter();
        adapter.output5V();
    }
```

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

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。



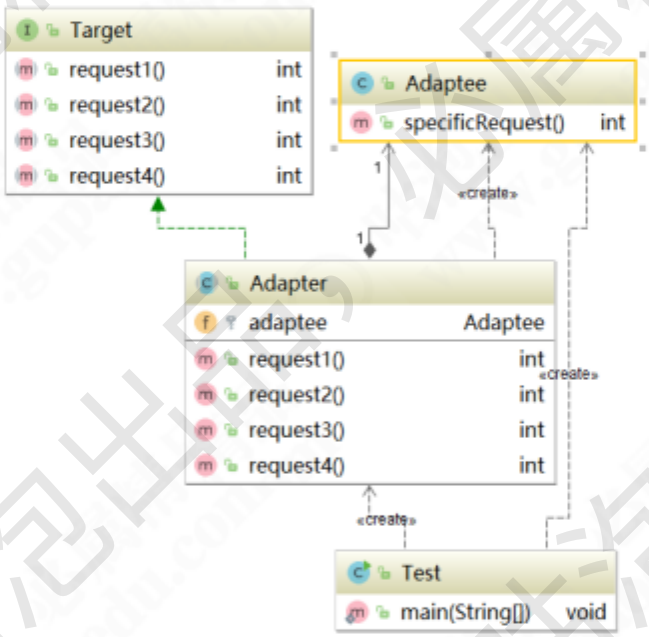
代码只需更改适配器（Adapter）实现，其他与类适配器一致：

```
public class PowerAdapter implements DC5 {
    private AC220 ac220;

    public PowerAdapter(AC220 ac220) {
        this.ac220 = ac220;
    }
```

```
    public int output5V() {
        int adapterInput = ac220.outputAC220V();
        int adapterOutput = adapterInput / 44;
        System.out.println("使用 Adapter 输入 AC" + adapterInput + "V,输出 DC" + adapterOutput + "V");
        return adapterOutput;
    }
}
```

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原理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：

```
public interface DC {
    int output5V();
    int output12V();
    int output24V();
    int output36V();
}
```

创建 Adaptee 角色 AC220 类：

```
public class AC220 {
    public int outputAC220V(){
```

```
        int output = 220;
        System.out.println("输出电压" + 220 + "V");
        return output;
    }
}
```

创建 Adapter 角色 PowerAdapter 类：

```
public class PowerAdapter implements DC {
    private AC220 ac220;

    public PowerAdapter(AC220 ac220) {
        this.ac220 = ac220;
    }

    public int output5V() {
        int adapterInput = ac220.outputAC220V();
        int adapterOutput = adapterInput / 44;
        System.out.println("使用 Adapter 输入 AC" + adapterInput + "V,输出 DC" + adapterOutput + "V");
        return adapterOutput;
    }

    public int output12V() {
        return 0;
    }

    public int output24V() {
        return 0;
    }

    public int output36V() {
        return 0;
    }
}
```

客户端代码：

```
public class Test {
    public static void main(String[] args) {
        DC adapter = new PowerAdapter(new AC220());
        adapter.output5V();
    }
}
```

重构第三登录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：

```
/**
 * Created by Tom.
 */
public class ResultMsg {

    private int code;
    private String msg;
    private Object data;

    public ResultMsg(int code, String msg, Object data) {
        this.code = code;
        this.msg = msg;
        this.data = data;
    }

    public int getCode() {
        return code;
    }

    public void setCode(int code) {
        this.code = code;
    }

    public String getMsg() {
        return msg;
    }

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }
}
```

假设老系统的登录逻辑 PassportService：

```
public class PassportService {

    /**
     * 注册方法
     * @param username
     * @param password
     * @return
     */
    public ResultMsg regist(String username,String password){
        return   new ResultMsg(200,"注册成功",new Member());
    }


    /**
     * 登录的方法
     * @param username
```

```
     * @param password
     * @return
     */
    public ResultMsg login(String username,String password){
        return null;
    }

}
```

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：

```
/**
 * Created by Tom.
 */
public class Member {

    private String username;
    private String password;
    private String mid;
    private String info;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getMid() {
        return mid;
    }

    public void setMid(String mid) {
        this.mid = mid;
    }

    public String getInfo() {
        return info;
    }

    public void setInfo(String info) {
        this.info = info;
    }
}
```

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：

```
public interface IPassportForThird {
```

```
    ResultMsg loginForQQ(String openId);

    ResultMsg loginForWechat(String openId);

    ResultMsg loginForToken(String token);

    ResultMsg loginForTelphone(String phone,String code);
}
```

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：

```
public class PassportForThirdAdapter extends PassportService implements IPassportForThird {

    public ResultMsg loginForQQ(String openId) {
        return loginForRegist(openId,null);
    }

    public ResultMsg loginForWechat(String openId) {
        return loginForRegist(openId,null);
    }

    public ResultMsg loginForToken(String token) {
        return loginForRegist(token,null);
    }

    public ResultMsg loginForTelphone(String phone, String code) {
        return loginForRegist(phone,null);
    }

    private ResultMsg loginForRegist(String username,String password){
        if(null == password){
            password = "THIRD_EMPTY";
        }
        super.regist(username,password);
        return super.login(username,password);
    }
}
```

客户端测试代码：

```
    public static void main(String[] args) {
        PassportForThirdAdapter adapter = new PassportForThirdAdapter();
        adapter.login("tom","123456");
        adapter.loginForQQ("sjooguwoersdfjhasjfsa");
        adapter.loginForWechat("slfsjoljsdo8234ssdfs");
    }
```

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 LoginAdapter 接口：

```
public interface ILoginAdapter {
    boolean support(Object object);
    ResultMsg login(String id,Object adapter);
}
```

然后，创建一个抽象类 AbstraceAdapter 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：

```
public class LoginForQQAdapter extends AbstraceAdapter{
    public boolean support(Object adapter) {
        return adapter instanceof LoginForQQAdapter;
    }

    public ResultMsg login(String id, Object adapter) {
        if(!support(adapter)){return null;}
        //accesseToken
        //time
        return super.loginForRegist(id,null);

    }

}
```

手机号登录 LoginForTelAdapter：

```
public class LoginForTelAdapter extends AbstraceAdapter{
    public boolean support(Object adapter) {
        return adapter instanceof LoginForTelAdapter;
    }

    public ResultMsg login(String id, Object adapter) {
        return super.loginForRegist(id,null);
    }
}
```

Token 自动登录 LoginForTokenAdapter:

```
public class LoginForTokenAdapter extends AbstraceAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForTokenAdapter;
    }

    public ResultMsg login(String id, Object adapter) {
        return super.loginForRegist(id,null);
    }
}
```

微信登录 LoginForWechatAdapter：

```
public class LoginForWechatAdapter extends AbstraceAdapter{
    public boolean support(Object adapter) {
```

```
        return adapter instanceof LoginForWechatAdapter;
    }

    public ResultMsg login(String id, Object adapter) {
        return super.loginForRegist(id,null);
    }
}
```

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

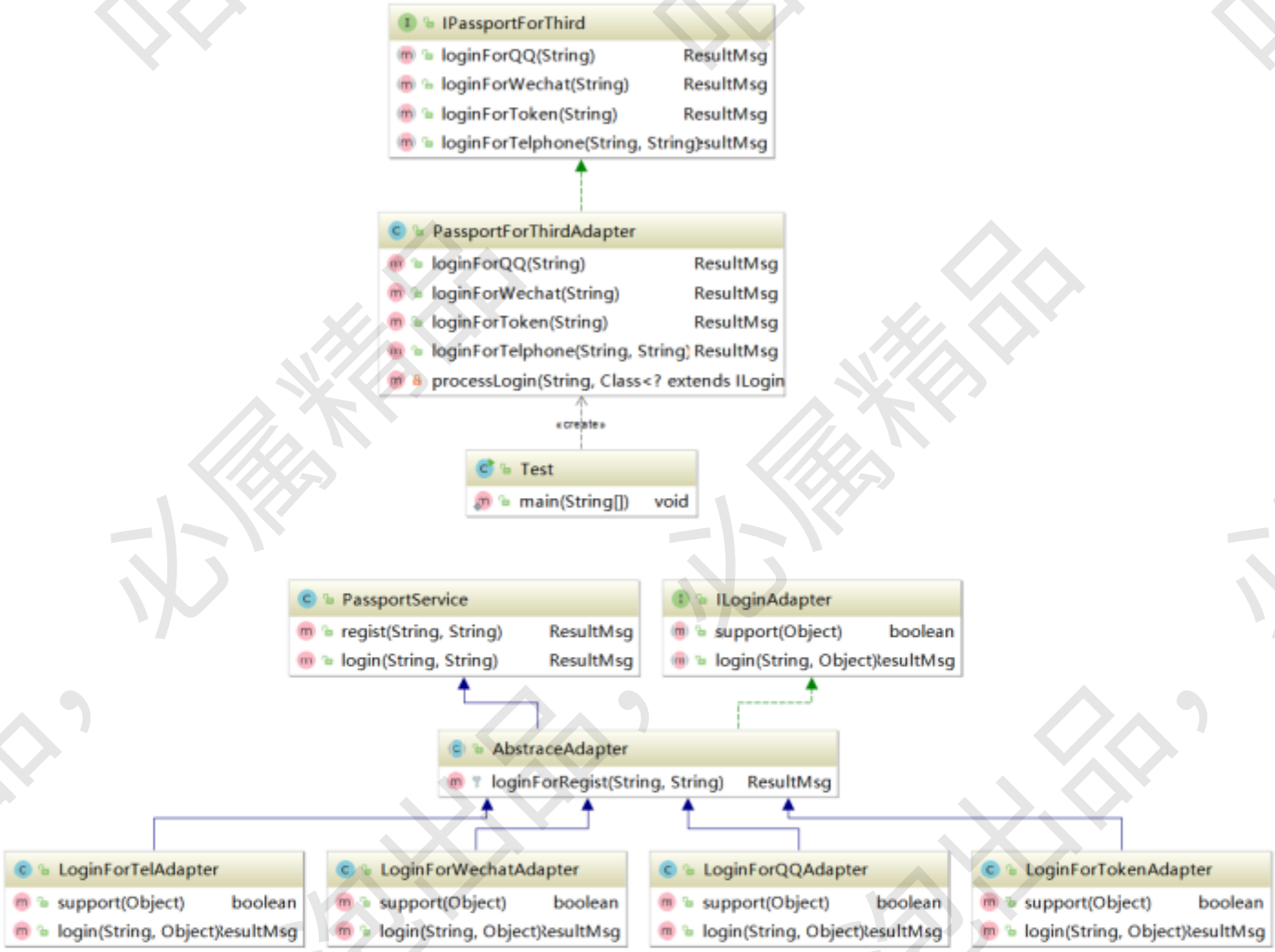
```
public class PassportForThirdAdapter implements IPassportForThird {

    public ResultMsg loginForQQ(String openId) {
        return processLogin(openId, LoginForQQAdapter.class);
    }

    public ResultMsg loginForWechat(String openId) {

        return processLogin(openId, LoginForWechatAdapter.class);

    }

    public ResultMsg loginForToken(String token) {

        return processLogin(token, LoginForTokenAdapter.class);
    }

    public ResultMsg loginForTelphone(String phone, String code) {
        return processLogin(phone, LoginForTelAdapter.class);
    }


    private ResultMsg processLogin(String id,Class<? extends ILoginAdapter> clazz){
        try {
            ILoginAdapter adapter = clazz.newInstance();
            if (adapter.support(adapter)){
                return adapter.login(id,adapter);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

}
```

客户端测试代码：

```
    public static void main(String[] args) {
```

```
        IPassportForThird adapter = new PassportForThirdAdapter();
        adapter.loginForQQ("sdfasdfasfasfas");
    }
```

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，我目前的这个设计也并不完美，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。

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：

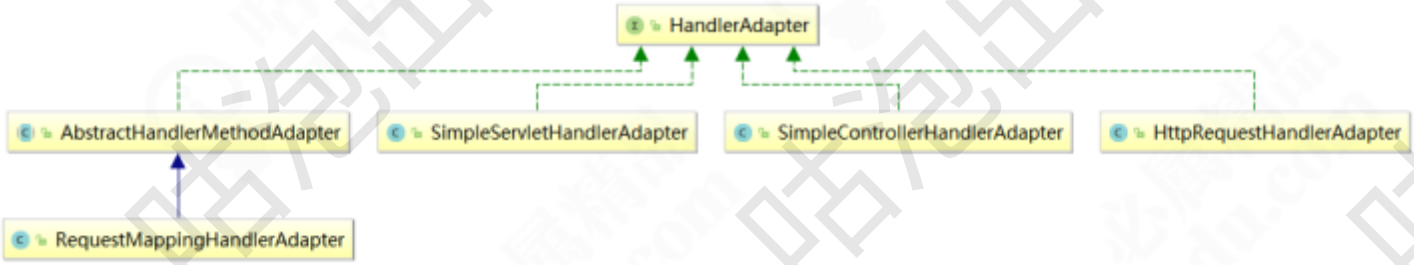
```
package org.springframework.aop.framework.adapter;

import org.aopalliance.aop.Advice;
import org.aopalliance.intercept.MethodInterceptor;
import org.springframework.aop.Advisor;

public interface AdvisorAdapter {
    boolean supportsAdvice(Advice var1);
    MethodInterceptor getInterceptor(Advisor var1);
}
```

再看 MethodBeforeAdviceAdapter 类：

```
package org.springframework.aop.framework.adapter;

import java.io.Serializable;
import org.aopalliance.aop.Advice;
import org.aopalliance.intercept.MethodInterceptor;
import org.springframework.aop.Advisor;
import org.springframework.aop.MethodBeforeAdvice;

class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {
    MethodBeforeAdviceAdapter() {
    }

    public boolean supportsAdvice(Advice advice) {
        return advice instanceof MethodBeforeAdvice;
    }
    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice)advisor.getAdvice();
        return new MethodBeforeAdviceInterceptor(advice);
    }
}
```

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：

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;
    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        try {
            ModelAndView mv = null;
            Object dispatchException = null;

            try {
                processedRequest = this.checkMultipart(request);
                multipartRequestParsed = processedRequest != request;
                mappedHandler = this.getHandler(processedRequest);
                if(mappedHandler == null) {
                    this.noHandlerFound(processedRequest, response);
                    return;
                }

                HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());
                String method = request.getMethod();
                boolean isGet = "GET".equals(method);
                if(isGet || "HEAD".equals(method)) {
                    long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                    if(this.logger.isDebugEnabled()) {
                        this.logger.debug("Last-Modified value for [" + getRequestUri(request) + "] is: " + lastModified);
                    }

                    if((new ServletWebRequest(request, response)).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }

                if(!mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }

                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
                if(asyncManager.isConcurrentHandlingStarted()) {
                    return;
                }

                this.applyDefaultViewName(processedRequest, mv);
                mappedHandler.applyPostHandle(processedRequest, response, mv);
            } catch (Exception var20) {
                dispatchException = var20;
            } catch (Throwable var21) {
                dispatchException = new NestedServletException("Handler dispatch failed", var21);
            }
```

```
            this.processDispatchResult(processedRequest, response, mappedHandler, mv, (Exception)dispatchException);
        } catch (Exception var22) {
            this.triggerAfterCompletion(processedRequest, response, mappedHandler, var22);
        } catch (Throwable var23) {
            this.triggerAfterCompletion(processedRequest, response, mappedHandler, new NestedServletException("Handler
processing failed", var23));
        }

    } finally {
        if(asyncManager.isConcurrentHandlingStarted()) {
            if(mappedHandler != null) {
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        } else if(multipartRequestParsed) {
            this.cleanupMultipart(processedRequest);
        }
    }
}
```

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

```
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    if(this.handlerAdapters != null) {
        Iterator var2 = this.handlerAdapters.iterator();

        while(var2.hasNext()) {
            HandlerAdapter ha = (HandlerAdapter)var2.next();
            if(this.logger.isTraceEnabled()) {
                this.logger.trace("Testing handler adapter [" + ha + "]");
            }

            if(ha.supports(handler)) {
                return ha;
            }
        }
    }

    throw new ServletException("No adapter for handler [" + handler + "]: The DispatcherServlet configuration needs to
include a HandlerAdapter that supports this handler");
}
```

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。

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

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。

桥接模式

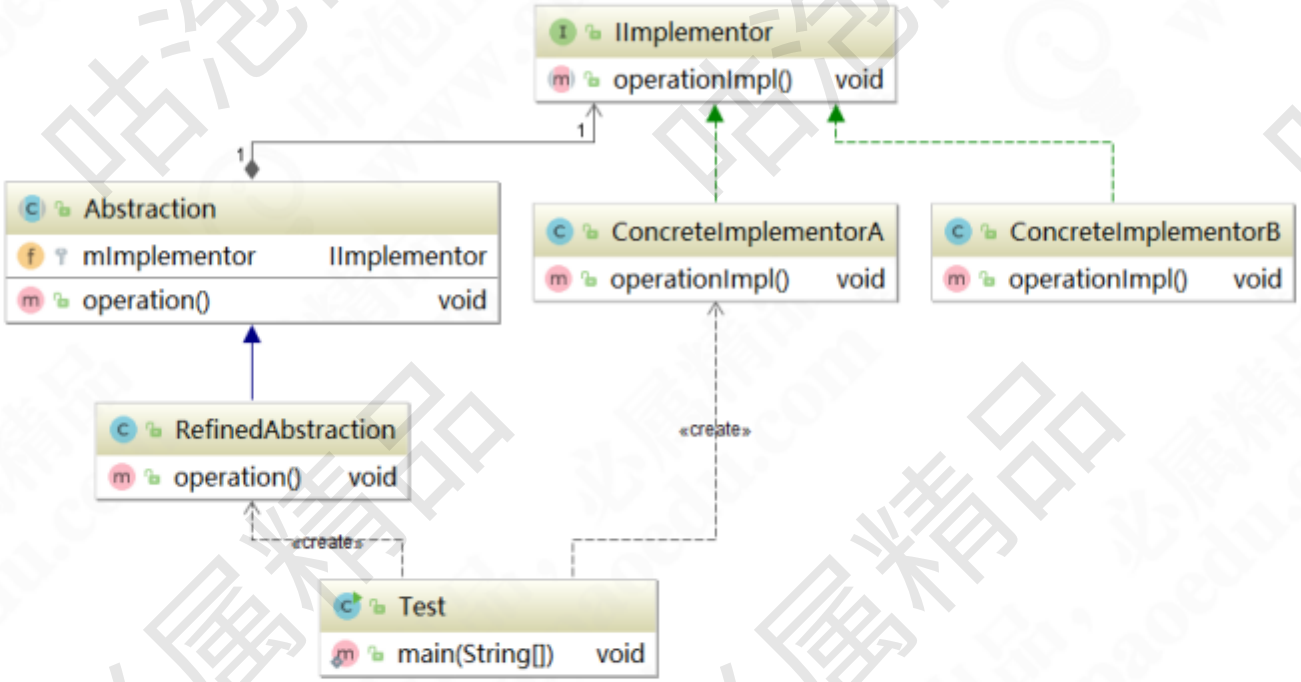
桥接模式（Bridge Pattern）也称为桥梁模式、接口(Interfce)模式或柄体（Handle and Body）模式，是将抽象部分与它的具体实现部分分离，使它们都可以独立地变化，属于结构型模式。

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得单一职责原则，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（ConcreteImplementor）：Implementor 的具体实现。

桥接模式的通用写法

创建抽象角色 Abstraction 类：

```
public abstract class Abstraction {
    protected IImplementor mImplementor;

    public Abstraction(IImplementor implementor) {
        this.mImplementor = implementor;
    }

    public void operation() {
```

```
        this.mImplementor.operationImpl();
    }
}
```

创建修正抽象角色 RefinedAbstraction 类：

```
public class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(IImplementor implementor) {
        super(implementor);
    }

    @Override
    public void operation() {
        super.operation();
        System.out.println("refined operation");
    }
}
```

创建实现角色 Implementor 类：

```
public interface IImplementor {
    void operationImpl();
}
```

创建具体实现 ConcreteImplementorA 类：

```
public class ConcreteImplementorA implements IImplementor {
    public void operationImpl() {
        System.out.println("I'm ConcreteImplementor A");
    }
}
```

客户端测试代码：

```
    public static void main(String[] args) {
        // 来一个实现化角色
        IImplementor imp = new ConcreteImplementorA();
        // 来一个抽象化角色，聚合实现
        Abstraction abs = new RefinedAbstraction(imp);
        // 执行操作
        abs.operation();
    }
```

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，单你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。

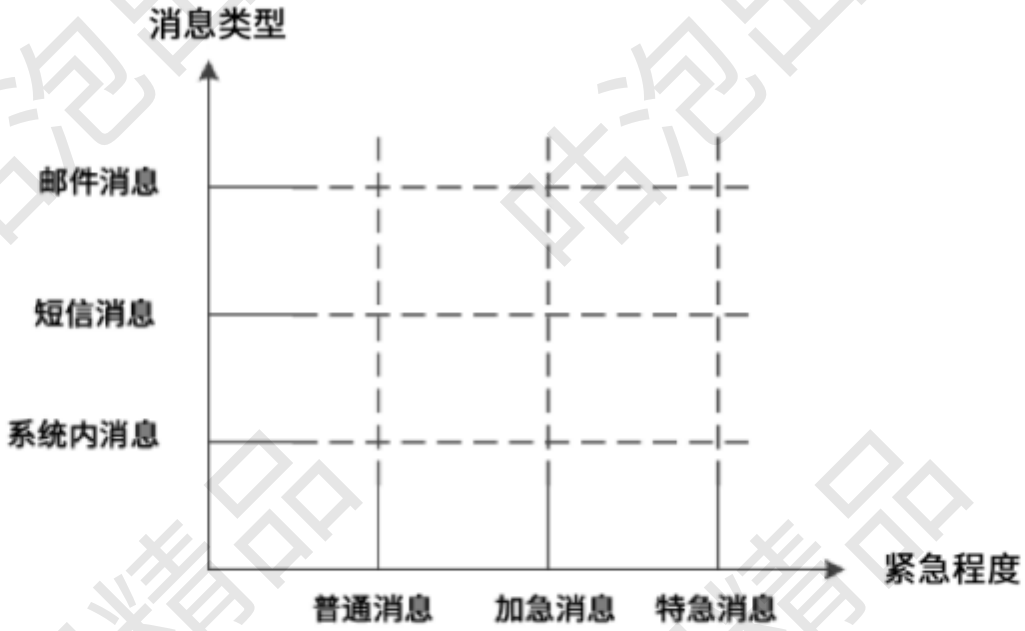
在生活场景中的桥接模式也随处可见，比如连接起两个空间维度的桥，比如链接虚拟网络与真实网络的连接。

连接起两个空间维度的桥

虚拟网络与真实网络的桥接

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，整个消息体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：

```
/**
 * 实现消息发送的统一接口
 */
public interface IMessage {
    //要发送的消息的内容和接收人
    void send(String message, String toUser);
}
```

创建邮件消息实现 EmailMessage 类：

```
/**
 * 邮件短消息的实现类
 */
public class EmailMessage implements IMessage {

    public void send(String message, String toUser) {
        System.out.println(String.format("使用邮件短消息的方法，发送消息 %s 给 %s", message, toUser));
    }
}
```

创建手机短信实现 SmsMessage 类：

```
/**
 * 系统内短消息的实现类
 * SMS(Short IMessage Service)短信息服务
 */
public class SmsMessage implements IMessage {

    public void send(String message, String toUser) {
        System.out.println(String.format("使用系统内部短消息的方法，发送消息 %s 给 %s", message, toUser));
    }
}
```

然后，再创建桥接抽象角色 AbstractMessage 类：

```
/**
 * 抽象消息类
 */
public abstract class AbstractMessage {
    //持有一个实现部分的对象
    IMessage message;

    //构造方法，传入实现部分的对象
    public AbstractMessage(IMessage message) {
        this.message = message;
    }

    //发送消息，委派给实现部分的方法
    public void sendMessage(String message, String toUser) {
        this.message.send(message, toUser);
    }
}
```

创建具体实现普通消息 NomalMessage 类：

```
/**
 * 普通消息类
 */
public class NomalMessage extends AbstractMessage {

    //构造方法，传入实现部分的对象
    public NomalMessage(IMessage message) {
        super(message);
    }

    @Override
    public void sendMessage(String message, String toUser) {
        //对于普通消息，直接调用父类方法，发送消息即可
        super.sendMessage(message, toUser);
    }
}
```

创建具体实现紧急消息 UrgencyMessage 类：

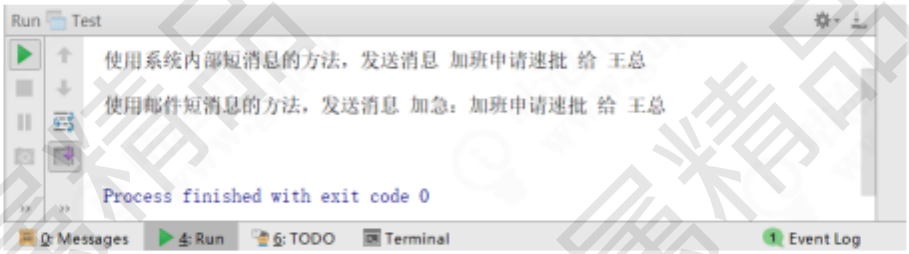
```
/**
 * 加急消息类
 */
public class UrgencyMessage extends AbstractMessage {

    //构造方法
    public UrgencyMessage(IMessage message) {
        super(message);
    }

    @Override
    public void sendMessage(String message, String toUser) {
        message = "加急：" + message;
        super.sendMessage(message, toUser);
    }

    //扩展它自己的功能，监控某个消息的处理状态
    public Object watch(String messageId) {
        //根据给出的 消息编码(messageId) 查询消息的处理状态，
        // 组织成监控的处理状态，然后返回。
        return null;
```

```
    }
}
```

编写客户端测试代码：

```
public static void main(String[] args) {
        IMessage message = new SmsMessage();
        AbstractMessage abstractMessage = new NomalMessage(message);
        abstractMessage.sendMessage("加班申请速批", "王总");

        message = new EmailMessage();
        abstractMessage = new UrgencyMessage(message);
        abstractMessage.sendMessage("加班申请速批", "王总");
    }
```

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。

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：

```
public interface Driver {

  Connection connect(String url, java.util.Properties info) throws SQLException;

  boolean acceptsURL(String url) throws SQLException;
```

```
    DriverPropertyInfo[] getPropertyInfo(String url, java.util.Properties info)  throws SQLException;

    int getMajorVersion();

    int getMinorVersion();

    boolean jdbcCompliant();

    public Logger getParentLogger() throws SQLFeatureNotSupportedException;

}
```

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。

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    public Driver() throws SQLException {
    }

    static {
        try {
            DriverManager.registerDriver(new Driver());
        } catch (SQLException var1) {
            throw new RuntimeException("Can't register driver!");
        }
    }
}
```

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：

```
public class DriverManager {
    // List of registered JDBC drivers
    private final static CopyOnWriteArrayList<DriverInfo> registeredDrivers = new CopyOnWriteArrayList<>();
    ...
    private DriverManager(){}
    static {
        loadInitialDrivers();
        println("JDBC DriverManager initialized");
    }

   ...
public static synchronized void registerDriver(java.sql.Driver driver)
        throws SQLException {

        registerDriver(driver, null);
    }

    public static synchronized void registerDriver(java.sql.Driver driver,
            DriverAction da)
        throws SQLException {
        if(driver != null) {
            registeredDrivers.addIfAbsent(new DriverInfo(driver, da));
        } else {
            throw new NullPointerException();
        }
```

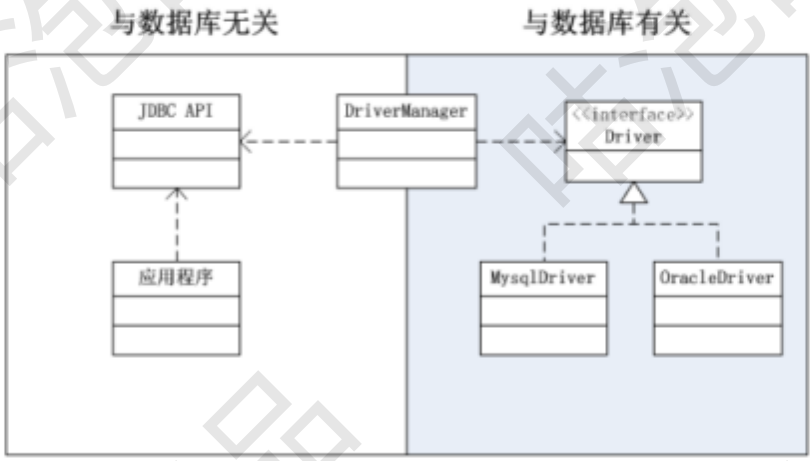
```
        println("registerDriver: " + driver);

    }
    ...
    }
```

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：

```
public class DriverManager {

    ...

    public static Connection getConnection(String url,
        java.util.Properties info) throws SQLException {

        return (getConnection(url, info, Reflection.getCallerClass()));
    }


    public static Connection getConnection(String url,
        String user, String password) throws SQLException {
        java.util.Properties info = new java.util.Properties();

        if (user != null) {
            info.put("user", user);
        }
        if (password != null) {
            info.put("password", password);
        }

        return (getConnection(url, info, Reflection.getCallerClass()));
    }


    public static Connection getConnection(String url)
        throws SQLException {

        java.util.Properties info = new java.util.Properties();
        return (getConnection(url, info, Reflection.getCallerClass()));
    }

    ...


    private static Connection getConnection(
                            String url, java.util.Properties info, Class<?> caller) throws SQLException {

        ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;
        synchronized(DriverManager.class) {
            if (callerCL == null) {
                callerCL = Thread.currentThread().getContextClassLoader();
            }
        }

        if(url == null) {
            throw new SQLException("The url cannot be null", "08001");
        }

        println("DriverManager.getConnection(\"" + url + "\")");
```

```
        SQLException reason = null;

        for(DriverInfo aDriver : registeredDrivers) {
            if(isDriverAllowed(aDriver.driver, callerCL)) {
                try {
                    println("    trying " + aDriver.driver.getClass().getName());
                    Connection con = aDriver.driver.connect(url, info);
                    if (con != null) {
                        println("getConnection returning " + aDriver.driver.getClass().getName());
                        return (con);
                    }
                } catch (SQLException ex) {
                    if (reason == null) {
                        reason = ex;
                    }
                }

            } else {
                println("    skipping: " + aDriver.getClass().getName());
            }

        }
        if (reason != null)    {
            println("getConnection failed: " + reason);
            throw reason;
        }

        println("getConnection: no suitable driver found for "+ url);
        throw new SQLException("No suitable driver found for "+ url, "08001");
    }
    ...
}
```

在 getConnection()中就又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：



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