课程目标

1、学会用模板模式梳理使用工作中流程标准化的业务场景。

2、通过学习适配模式，优雅地解决代码功能的兼容问题。

3、了解 JDK 源码和 Spring 源码中对模板模式的运用。

内容定位

1、定位高级课程，不太适合接触业务场景比较单一的人群。

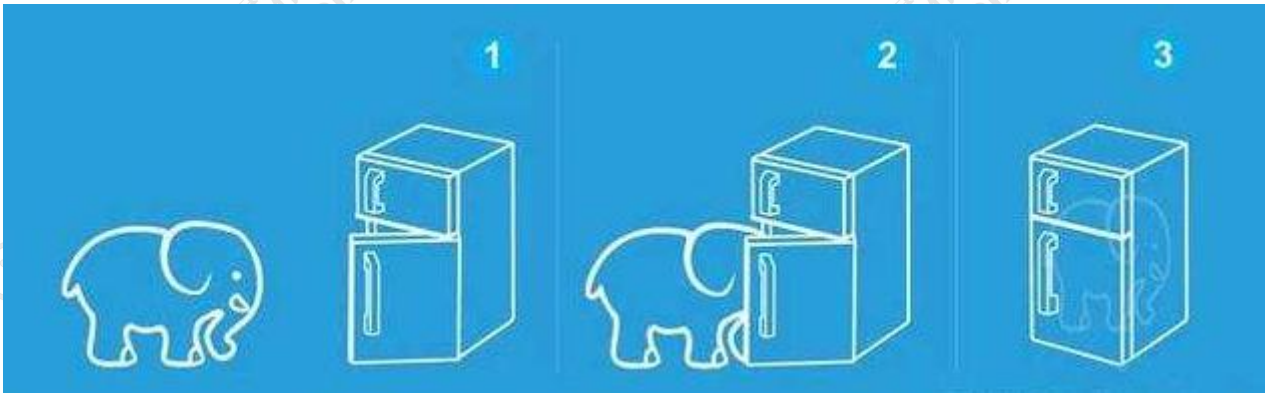
2、有代码重构需求的人群一定要掌握好适配器模式。

3、深刻了解模板模式和适配器模式的应用场景。

模板模式

模板模式的应用场景

我们平时办理入职流程填写入职登记表-->打印简历-->复印学历-->复印身份证-->签订劳动合同-->建立花名册-->办理工牌-->安排工位等；再比如，我平时在家里炒菜：洗锅-->点火-->热锅-->上油-->下原料-->翻炒-->放调料-->出锅；再比如赵本山问宋丹丹:“如何把大象放进冰箱？”宋丹丹回答：“第一步：打开冰箱门，第二步：把大象塞进冰箱，第三步：关闭冰箱门”。赵本山再问：“怎么把长劲鹿放进冰箱？”宋丹丹答：“第一步：打开冰箱门，第二步：把大象拿出来，第三步：把长劲鹿塞进去，第四步：关闭冰箱门”（如下图所示），这些都是模板模式的体现。



模板模式通常又叫模板方法模式（Template Method Pattern）是指定义一个算法的骨架，并允许子类为一个或者多个步骤提供实现。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法的某些步骤，属于行为性设计模式。模板方法适用于以下应用场景：

1、一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

2、各子类中公共的行为被提取出来并集中到一个公共的父类中，从而避免代码重复。

我们还是以咕泡学院的课程创建流程为例：发布预习资料-->制作课件 PPT-->在线直播-->提交课堂笔记-->提交源码-->布置作业-->检查作业。首先我们来创建 NetworkCourse 抽象类：

```
package com.gupaoedu.vip.pattern.template.course;

/**
 * 模板会有一个或者多个未现实方法，
 * 而且这几个未实现方法有固定的执行循序
 * Created by Tom.
 */
public abstract class NetworkCourse {

    protected final void createCourse(){
        //1、发布预习资料
        this.postPreResource();

        //2、制作 PPT 课件
        this.createPPT();
```

```
        //3、在线直播
        this.liveVideo();

        //4、提交课件、课堂笔记
        this.postNote();

        //5、提交源码
        this.postSource();

        //6、布置作业，有些课是没有作业，有些课是有作业的
        //如果有作业的话，检查作业，如果没作业，完成了
        if(needHomework()){
            checkHomework();
        }
    }

    abstract void checkHomework();

    //钩子方法：实现流程的微调
    protected boolean needHomework(){return false;}

    final void postSource(){
        System.out.println("提交源代码");
    }

    final void postNote(){
        System.out.println("提交课件和笔记");
    }

    final void liveVideo(){
        System.out.println("直播授课");
    }

    final void createPPT(){
        System.out.println("创建备课 PPT");
    }

    final void postPreResource(){
        System.out.println("分发预习资料");
    }
}
```

上面的代码中有个钩子方法可能有些小伙伴还不是太理解，在此我稍作解释。设计钩子

方法的主要目的是用来干预执行流程，使得我们控制行为流程更加灵活，更符合实际业务的需求。钩子方法的返回值一般为适合条件分支语句的返回值（如 boolean、int 等）。小伙伴们可以根据自己的业务场景来决定是否需要使用钩子方法。接下来创建 JavaCourse 类：

```
package com.gupaoedu.vip.pattern.template.course;

/**
 * Created by Tom.
 */
public class JavaCourse extends NetworkCourse {
    void checkHomework() {
        System.out.println("检查 Java 的架构课件");
    }
}
```

创建 BigDataCourse 类：

```
package com.gupaoedu.vip.pattern.template.course;

/**
 * Created by Tom on 2019/3/16.
 */
public class BigDataCourse extends NetworkCourse {

    private boolean needHomeworkFlag = false;

    public BigDataCourse(boolean needHomeworkFlag) {
        this.needHomeworkFlag = needHomeworkFlag;
    }

    void checkHomework() {
        System.out.println("检查大数据的课后作业");
    }

    @Override
    protected boolean needHomework() {
        return this.needHomeworkFlag;
    }
}
```

客户端测试代码：

```
package com.gupaoedu.vip.pattern.template.course;

/**
 * Created by Tom on 2019/3/16.
 */
public class NetworkCourseTest {
    public static void main(String[] args) {

        System.out.println("---Java 架构师课程---");
        NetworkCourse javaCourse = new JavaCourse();
        javaCourse.createCourse();

        System.out.println("---大数据课程---");
        NetworkCourse bigDataCourse = new BigDataCourse(true);
        bigDataCourse.createCourse();

    }
}
```

通过这样一个案例，相信下伙伴们对模板模式有了一个基本的印象。为了加深理解，下面我们来结合一个常见的业务场景。

利用模板模式重构 JDBC 操作业务场景

创建一个模板类 JdbcTemplate，封装所有的 JDBC 操作。以查询为例，每次查询的表不同，返回的数据结构也就不一样。我们针对不同的数据，都要封装成不同的实体对象。而每个实体封装的逻辑都是不一样的，但封装前和封装后的处理流程是不变的，因此，我们可以使用模板方法模式来设计这样的业务场景。先创建约束 ORM 逻辑的接口 RowMapper：

```
package com.gupaoedu.vip.pattern.template;

import java.sql.ResultSet;

/**
 * Created by Tom.
 */
public interface RowMapper<T> {

    T mapRow(ResultSet rs, int rowNum) throws Exception;
```

```
}
```

在创建封装了所有处理流程的抽象类 JdbcTemplate：

```
package com.gupaoedu.vip.pattern.template.jdbc;

import com.sun.org.apache.regexp.internal.RE;
import com.sun.org.apache.xerces.internal.xs.datatypes.ObjectList;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

/**
 * Created by Tom on 2019/3/16.
 */
public abstract class JdbcTemplate {
    private DataSource dataSource;

    public JdbcTemplate(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public List<?> executeQuery(String sql, RowMapper<?> rowMapper, Object[] values){
        try {
            //1、获取连接
            Connection conn = this.getConnection();
            //2、创建语句集
            PreparedStatement pstm = this.createPrepareStatement(conn,sql);
            //3、执行语句集
            ResultSet rs = this.executeQuery(pstm,values);
            //4、处理结果集
            List<?> result = this.paresResultSet(rs,rowMapper);
            //5、关闭结果集
            this.closeResultSet(rs);
            //6、关闭语句集
            this.closeStatement(pstm);
            //7、关闭连接
            this.closeConnection(conn);
            return result;
```

```
        }catch (Exception e){
            e.printStackTrace();
        }
        return null;
    }

    protected void closeConnection(Connection conn) throws Exception {
        //数据库连接池，我们不是关闭
        conn.close();
    }

    protected void closeStatement(PreparedStatement pstm) throws Exception {
        pstm.close();
    }

    protected void closeResultSet(ResultSet rs) throws Exception {
        rs.close();
    }

    protected List<?> paresResultSet(ResultSet rs, RowMapper<?> rowMapper) throws Exception {
        List<Object> result = new ArrayList<Object>();
        int rowNum = 1;
        while (rs.next()){
            result.add(rowMapper.mapRow(rs,rowNum ++));
        }
        return result;
    }

    protected ResultSet executeQuery(PreparedStatement pstm, Object[] values) throws Exception {
        for (int i = 0; i < values.length; i++) {
            pstm.setObject(i,values[i]);
        }
        return pstm.executeQuery();
    }

    protected PreparedStatement createPrepareStatement(Connection conn, String sql) throws Exception
{
        return conn.prepareStatement(sql);
    }

    public Connection getConnection() throws Exception {
        return this.dataSource.getConnection();
    }
}
```

创建实体对象 Member 类：

```
package com.gupaoedu.vip.pattern.template.entity;

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

    private String username;
    private String password;
    private String nickName;

    private int age;
    private String addr;

    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 getNickName() {
        return nickName;
    }

    public void setNickName(String nickName) {
        this.nickName = nickName;
    }

    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }

    public String getAddr() {
        return addr;
    }

    public void setAddr(String addr) {
        this.addr = addr;
    }
}
```

创建数据库操作类 MemberDao：

```
package com.gupaoedu.vip.pattern.template.jdbc.dao;

import com.gupaoedu.vip.pattern.template.jdbc.JdbcTemplate;
import com.gupaoedu.vip.pattern.template.jdbc.Member;
import com.gupaoedu.vip.pattern.template.jdbc.RowMapper;

import javax.sql.DataSource;
import java.sql.ResultSet;
import java.util.List;

/**
 * Created by Tom.
 */
public class MemberDao extends JdbcTemplate {
    public MemberDao(DataSource dataSource) {
        super(dataSource);
    }

    public List<?> selectAll(){
        String sql = "select * from t_member";
        return super.executeQuery(sql, new RowMapper<Member>() {
            public Member mapRow(ResultSet rs, int rowNum) throws Exception {
                Member member = new Member();
                //字段过多，原型模式
                member.setUsername(rs.getString("username"));
                member.setPassword(rs.getString("password"));
                member.setAge(rs.getInt("age"));
                member.setAddr(rs.getString("addr"));
                return member;
            }
```

```
        },null);
    }
}
```

客户端测试代码：

```
package com.gupaoedu.vip.pattern.template.jdbc;

import com.gupaoedu.vip.pattern.template.jdbc.dao.MemberDao;
import java.util.List;

/**
 * Created by Tom on 2019/3/16.
 */
public class MemberDaoTest {

    public static void main(String[] args) {
        MemberDao memberDao = new MemberDao(null);
        List<?> result = memberDao.selectAll();
        System.out.println(result);
    }
}
```

希望通过这两个案例的业务场景分析，能够帮助小伙们对模板方法模式有更深的理解。

模板模式在源码中的体现

先来看 JDK 中的 AbstractList，来看代码：

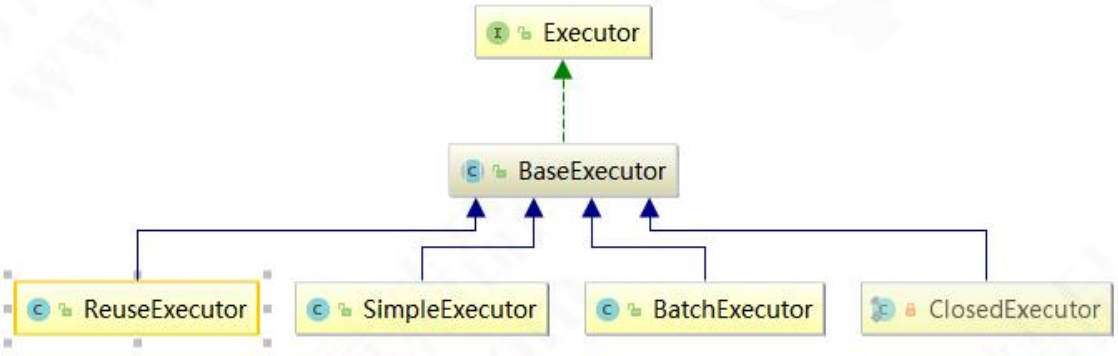
```
package java.util;

public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
    ...
    abstract public E get(int index);
    ...
}
```

我们看到 get()是一个抽象方法，那么它的逻辑就是交给子类来实现，我们大家所熟知的 ArrayList 就是 AbstractList 的子类。同理，有 AbstractList 就有 AbstractSet 和 AbstractMap，有兴趣的小伙伴可以去看看这些的源码实现。还有一个每天都在用的 HttpServlet，有三个方法 service()和 doGet()、doPost()方法，都是模板方法的抽象实现。

在 MyBatis 框架也有一些经典的应用，我们来一下 BaseExecutor 类，它是一个基础的 SQL 执行类，实现了大部分的 SQL 执行逻辑，然后把几个方法交给子类定制化完成，源码如下：

```
...
public abstract class BaseExecutor implements Executor {
    ...
    protected abstract int doUpdate(MappedStatement var1, Object var2) throws SQLException;

    protected abstract List<BatchResult> doFlushStatements(boolean var1) throws SQLException;

    protected abstract <E> List<E> doQuery(MappedStatement var1, Object var2, RowBounds var3,
ResultHandler var4, BoundSql var5) throws SQLException;

    protected abstract <E> Cursor<E> doQueryCursor(MappedStatement var1, Object var2, RowBounds var3,
BoundSql var4) throws SQLException;
      ...
}
```

如 doUpdate、doFlushStatements、doQuery、doQueryCursor 这几个方法就是交由子类来实现，那么 BaseExecutor 有哪些子类呢？我们来看一下它的类图：



我们一起来看一下 SimpleExecutor 的 doUpdate 实现：

```
public int doUpdate(MappedStatement ms, Object parameter) throws SQLException {
    Statement stmt = null;

    int var6;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this, ms, parameter,
RowBounds.DEFAULT, (ResultHandler)null, (BoundSql)null);
```

```
        stmt = this.prepareStatement(handler, ms.getStatementLog());
        var6 = handler.update(stmt);
    } finally {
        this.closeStatement(stmt);
    }

    return var6;
}
```

再来对比一下 BatchExecutor 的 doUpate 实现：

```
public int doUpdate(MappedStatement ms, Object parameterObject) throws SQLException {
    Configuration configuration = ms.getConfiguration();
    StatementHandler handler = configuration.newStatementHandler(this, ms, parameterObject,
RowBounds.DEFAULT, (ResultHandler)null, (BoundSql)null);
    BoundSql boundSql = handler.getBoundSql();
    String sql = boundSql.getSql();
    Statement stmt;
    if(sql.equals(this.currentSql) && ms.equals(this.currentStatement)) {
        int last = this.statementList.size() - 1;
        stmt = (Statement)this.statementList.get(last);
        this.applyTransactionTimeout(stmt);
        handler.parameterize(stmt);
        BatchResult batchResult = (BatchResult)this.batchResultList.get(last);
        batchResult.addParameterObject(parameterObject);
    } else {
        Connection connection = this.getConnection(ms.getStatementLog());
        stmt = handler.prepare(connection, this.transaction.getTimeout());
        handler.parameterize(stmt);
        this.currentSql = sql;
        this.currentStatement = ms;
        this.statementList.add(stmt);
        this.batchResultList.add(new BatchResult(ms, sql, parameterObject));
    }

    handler.batch(stmt);
    return -2147482646;
}
```

细心的小伙伴一定看出来了差异。当然，我们在这里就暂时不对 MyBatis 源码进行深入分析，感兴趣的小伙伴可以继续关注我们后面的课程。

模板模式的优缺点

优点：

1、利用模板方法将相同处理逻辑的代码放到抽象父类中，可以提高代码的复用性。

2、将不同的代码不同的子类中，通过对子类的扩展增加新的行为，提高代码的扩展性。

3、把不变的行为写在父类上，去除子类的重复代码，提供了一个很好的代码复用平台，符合开闭原则。

缺点：

1、类数目的增加，每一个抽象类都需要一个子类来实现，这样导致类的个数增加。

2、类数量的增加，间接地增加了系统实现的复杂度。

3、继承关系自身缺点，如果父类添加新的抽象方法，所有子类都要改一遍。

模板方法模式比较简单，相信小伙伴们肯定能学会，也肯定能理解好！只要勤加练习，多结合业务场景思考问题，就能够把模板方法模式运用好。

适配器模式

适配器模式的应用场景

适配器模式（Adapter Pattern）是指将一个类的接口转换成客户期望的另一个接口，使原本的接口不兼容的类可以一起工作，属于结构型设计模式。

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

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 AC220 类，表示 220V 交流电：

```
package com.gupaoedu.vip.pattern.adapter.objectadapter;

/**
 * Created by Tom
 */
public class AC220 {
    public int outputAC220V(){
        int output = 220;
        System.out.println("输出交流电"+output+"V");
        return output;
    }
}
```

创建 DC5 接口，表示 5V 直流电的标准：

```
package com.gupaoedu.vip.pattern.adapter.objectadapter;

/**
 * Created by Tom
 */
public interface DC5 {
    int outputDC5V();
}
```

创建电源适配器 PowerAdapter 类：

```
package com.gupaoedu.vip.pattern.adapter.objectadapter;

/**
```

```
 * Created by Tom
 */
public class PowerAdapter implements DC5{
    private AC220 ac220;
    public PowerAdapter(AC220 ac220){
        this.ac220 = ac220;
    }
    public int outputDC5V() {
        int adapterInput = ac220.outputAC220V();
        //变压器...
        int adapterOutput = adapterInput/44;
        System.out.println("使用 PowerAdapter 输入 AC:"+adapterInput+"V"+"输出
DC:"+adapterOutput+"V");
        return adapterOutput;
    }
}
```

客户端测试代码：

```
package com.gupaoedu.vip.pattern.adapter.objectadapter;

/**
 * Created by Tom
 */
public class ObjectAdapterTest {
    public static void main(String[] args) {
        DC5 dc5 = new PowerAdapter(new AC220());
        dc5.outputDC5V();
    }
}
```

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

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：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter;

/**
 * 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 SiginService：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v1.service;

import com.gupaoedu.vip.pattern.adapter.loginadapter.Member;
import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class SiginService {

    /**
     * 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：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter;

/**
 * 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;
    }
}
```

创建一个新的类继承原来的逻辑，运行非常稳定的代码我们不去改动：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v1.service;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;
```

```
/**
 *
 * 稳定的方法不去动，直接继承下来
 * Created by Tom.
 */
public class SigninForThirdService extends SiginService {

    public ResultMsg loginForQQ(String openId){
        //1、openId 是全局唯一，我们可以把它当做是一个用户名(加长)
        //2、密码默认为 QQ_EMPTY
        //3、注册（在原有系统里面创建一个用户）

        //4、调用原来的登录方法

        return loginForRegist(openId,null);
    }

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

    public ResultMsg loginForToken(String token){
        //通过 token 拿到用户信息，然后再重新登陆了一次
        return  null;
    }

    public ResultMsg loginForTelphone(String telphone,String code){

        return null;
    }

    public ResultMsg loginForRegist(String username,String password){
        super.regist(username,null);
        return super.login(username,null);
    }
}
```

客户端测试代码：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v1;

import com.gupaoedu.vip.pattern.adapter.loginadapter.v1.service.SigninForThirdService;

/**
 * Created by Tom.
```

```
 */
public class SigninForThirdServiceTest {

    public static void main(String[] args) {

        SigninForThirdService service = new SigninForThirdService();

        //不改变原来的代码，也要能够兼容新的需求
        //还可以再加一层策略模式
        service.loginForQQ("sdfgdgfwresdf9123sdf");

    }
}
```

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：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public interface LoginAdapter {
    boolean support(Object adapter);
    ResultMsg login(String id,Object adapter);
}
```

分别实现不同的登录适配，QQ 登录 LoginForQQAdapter：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class LoginForQQAdapter implements LoginAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForQQAdapter;
    }

    public ResultMsg login(String id, Object adapter) {
```

```
        return null;
    }
}
```

新浪微博登录 LoginForSinaAdapter：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class LoginForSinaAdapter implements LoginAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForSinaAdapter;
    }
    public ResultMsg login(String id, Object adapter) {
        return null;
    }
}
```

手机号登录 LoginForTelAdapter：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class LoginForTelAdapter implements LoginAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForTelAdapter;
    }
    public ResultMsg login(String id, Object adapter) {
        return null;
    }
}
```

Token 自动登录 LoginForTokenAdapter:

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;
```

```
import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class LoginForTokenAdapter implements LoginAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForTokenAdapter;
    }
    public ResultMsg login(String id, Object adapter) {
        return null;
    }
}
```

微信登录 LoginForWechatAdapter：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters;
import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public class LoginForWechatAdapter implements LoginAdapter {
    public boolean support(Object adapter) {
        return adapter instanceof LoginForWechatAdapter;
    }
    public ResultMsg login(String id, Object adapter) {
        return null;
    }
}
```

然后，创建第三方登录兼容接口 IPassportForThird:

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;

/**
 * Created by Tom.
 */
public interface IPassportForThird {

    /**
     * QQ 登录
     * @param id
     * @return
```

```
     */
    ResultMsg loginForQQ(String id);

    /**
     * 微信登录
     * @param id
     * @return
     */
    ResultMsg loginForWechat(String id);

    /**
     * 记住登录状态后自动登录
     * @param token
     * @return
     */
    ResultMsg loginForToken(String token);

    /**
     * 手机号登录
     * @param telphone
     * @param code
     * @return
     */
    ResultMsg loginForTelphone(String telphone, String code);

    /**
     * 注册后自动登录
     * @param username
     * @param passport
     * @return
     */
    ResultMsg loginForRegist(String username, String passport);
}
```

实现兼容 PassportForThirdAdapter：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2;

import com.gupaoedu.vip.pattern.adapter.loginadapter.ResultMsg;
import com.gupaoedu.vip.pattern.adapter.loginadapter.v1.service.SiginService;
import com.gupaoedu.vip.pattern.adapter.loginadapter.v2.adapters.*;

/**
 * 第三方登录自由适配
 * Created by Tom.
```

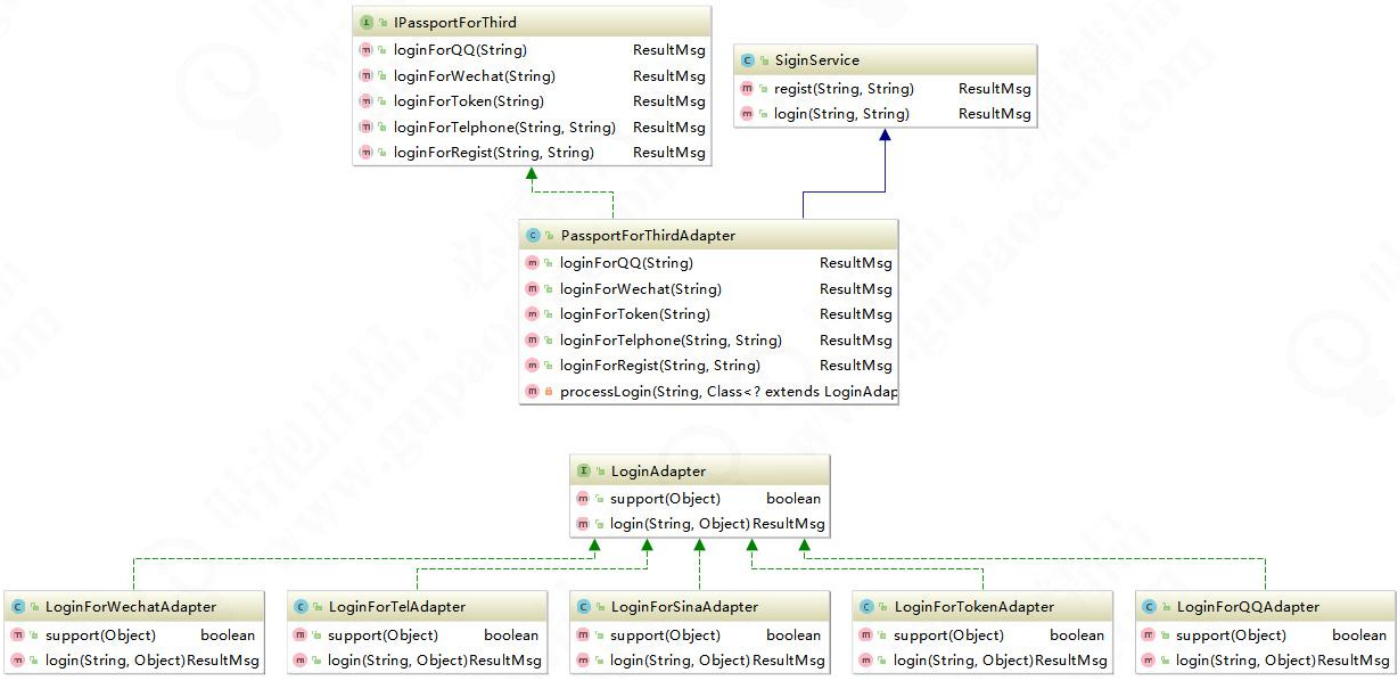
```
 */
public class PassportForThirdAdapter extends SiginService implements IPassportForThird {

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

    public ResultMsg loginForWechat(String id) {
        return processLogin(id,LoginForWechatAdapter.class);
    }

    public ResultMsg loginForToken(String token) {
        return processLogin(token,LoginForTokenAdapter.class);
    }

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

    public ResultMsg loginForRegist(String username, String passport) {
        super.regist(username,null);
        return super.login(username,null);
    }

    //这里用到了简单工厂模式及策略模式
    private ResultMsg processLogin(String key,Class<? extends LoginAdapter> clazz){
        try {
            LoginAdapter adapter = clazz.newInstance();
            if(adapter.support(adapter)) {
                return adapter.login(key, adapter);
            }else {
                return null;
            }
        }catch (Exception e){
            e.printStackTrace();;
        }
        return null;
    }
}
```

客户端测试代码：

```
package com.gupaoedu.vip.pattern.adapter.loginadapter.v2;
/**
 * Created by Tom.
```

```
 */
public class PassportTest {
    public static void main(String[] args) {
        IPassportForThird passportForThird = new PassportForThirdAdapter();
        passportForThird.loginForQQ("");
    }
}
```

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 LoginAdapter 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);
}
```

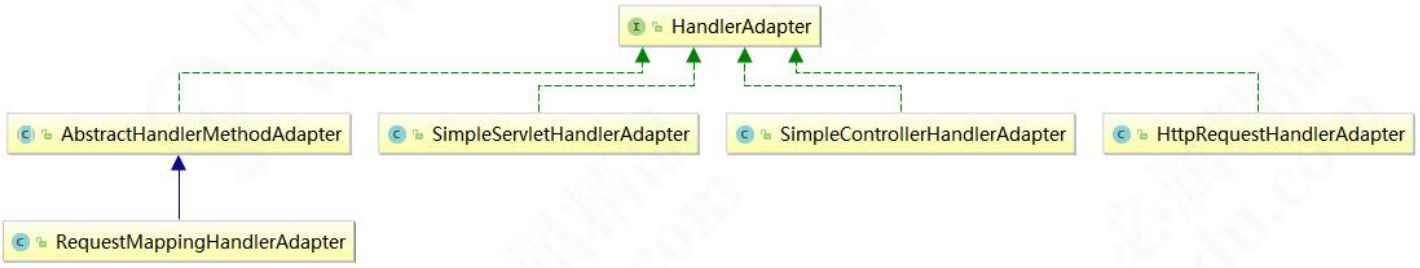
再看 MethodBeforeAdviceAdapter 类：

```
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。