课程目标

1、掌握委派模式，精简程序逻辑，提升代码的可读性。

2、通过学习策略模式来消除程序中大量的 if...else...和 switch 语句。

3、深刻理解策略模式的应用场景，提高算法的保密性和安全性。

内容定位

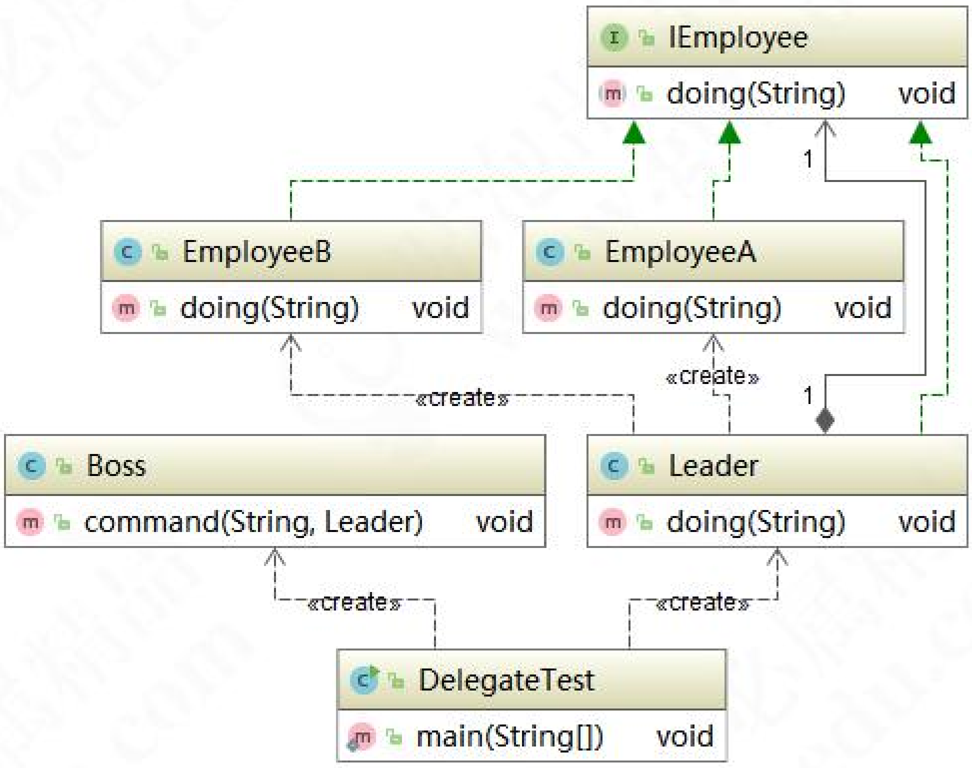
1、希望通过对委派模式的学习，让自己写出更加优雅的代码的人群。

2、希望通过对策略模式的学习，来消除程序中大量的冗余代码和多重条件转移语句的人群。

委派模式

委派模式的定义及应用场景

委派模式不属于 GOF23 种设计模式中。委派模式（Delegate Pattern）的基本作用就是负责任务的调用和分配任务，跟代理模式很像，可以看做是一种特殊情况下的静态代理的全权代理，但是代理模式注重过程，而委派模式注重结果。委派模式在 Spring 中应用非常多，大家常用的 DispatcherServlet 其实就是用到了委派模式。现实生活中也常有委派的场景发生，例如：老板（Boss）给项目经理（Leader）下达任务，项目经理会根据实际情况给每个员工派发工作任务，待员工把工作任务完成之后，再由项目经理汇报工作进度和结果给老板。我们用代码来模拟下这个业务场景，先来看一下类图：



创建 IEmployee 员工接口：

```java
package com.gupaoedu.vip.pattern.delegate.simple;

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

    public void doing(String command);

}
```

创建员工 EmployeeA 类：

```java
package com.gupaoedu.vip.pattern.delegate.simple;

/**
 * Created by Tom.
 */
public class EmployeeA implements IEmployee {
    @Override
    public void doing(String command) {
        System.out.println("我是员工 A，我现在开始干" + command + "工作");
    }
}
```

创建员工 EmployeeB 类:

```
package com.gupaoedu.vip.pattern.delegate.simple;

/**
 * Created by Tom.
 */
public class EmployeeB implements IEmployee {
    @Override
    public void doing(String command) {
        System.out.println("我是员工 B，我现在开始干" + command + "工作");
    }
}
```

创建项目经理 Leader 类:

```
package com.gupaoedu.vip.pattern.delegate.simple;

import java.util.HashMap;
import java.util.Map;

/**
 * Created by Tom.
 */
public class Leader implements IEmployee {

    private Map<String,IEmployee> targets = new HashMap<String,IEmployee>();

    public Leader() {
        targets.put("加密",new EmployeeA());
        targets.put("登录",new EmployeeB());
    }

    //项目经理自己不干活
    public void doing(String command){
        targets.get(command).doing(command);
    }

}
```

创建 Boss 类下达命令:

```
package com.gupaoedu.vip.pattern.delegate.simple;
/**
```

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

    public void command(String command,Leader leader){
        leader.doing(command);
    }
}
```

测试代码：

```
package com.gupaoedu.vip.pattern.delegate.simple;

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

    public static void main(String[] args) {

        //客户请求（Boss）、委派者（Leader）、被被委派者（Target）
        //委派者要持有被委派者的引用
        //代理模式注重的是过程， 委派模式注重的是结果
        //策略模式注重是可扩展（外部扩展），委派模式注重内部的灵活和复用
        //委派的核心：就是分发、调度、派遣

        //委派模式：就是静态代理和策略模式一种特殊的组合

        new Boss().command("登录",new Leader());

    }

}
```

通过上面的代码，生动地还原了项目经理分配工作的业务场景，也是委派模式的生动体现。

委派模式在源码中的体现

下面我们再来还原一下 SpringMVC 的 DispatcherServlet 是如何实现委派模式的。创建业务类 MemberController：

```
package com.gupaoedu.vip.pattern.delegate.mvc.controllers;

```

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

    public void getMemberById(String mid){

    }

}
```

OrderController 类:

```
package com.gupaoedu.vip.pattern.delegate.mvc.controllers;

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

    public void getOrderById(String mid){

    }

}
```

SystemController 类:

```
package com.gupaoedu.vip.pattern.delegate.mvc.controllers;

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

    public void logout(){

    }
}
```

创建 DispatcherServlet 类:

```
package com.gupaoedu.vip.pattern.delegate.mvc;

import com.gupaoedu.vip.pattern.delegate.mvc.controllers.MemberController;
import com.gupaoedu.vip.pattern.delegate.mvc.controllers.OrderController;
import com.gupaoedu.vip.pattern.delegate.mvc.controllers.SystemController;
```

```java
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

/**
 * 相当于是项目经理的角色
 * Created by Tom.
 */
public class DispatcherServlet extends HttpServlet{

    private void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception{

        String uri = request.getRequestURI();
        String mid = request.getParameter("mid");

        if("getMemberById".equals(uri)){
            new MemberController().getMemberById(mid);
        }else if("getOrderById".equals(uri)){
            new OrderController().getOrderById(mid);
        }else if("logout".equals(uri)){
            new SystemController().logout();
        }else {
            response.getWriter().write("404 Not Found!!");
        }
    }

    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try {
            doDispatch(req,resp);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

配置 web.xml 文件:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:javaee="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">
   <display-name>Gupao Web Application</display-name>


   <servlet>
      <servlet-name>delegateServlet</servlet-name>
      <servlet-class>com.gupaoedu.vip.pattern.delegate.mvc.DispatcherServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>delegateServlet</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>


</web-app>
```

一个完整的委派模式就实现出来了。当然，在 Spring 中运用到委派模式不仅于此，还有很多。小伙伴们可以通过命名就可以识别。在 Spring 源码中，只要以 Delegate 结尾的都是实现了委派模式。例如：BeanDefinitionParserDelegate 根据不同类型委派不同的逻辑解析 BeanDefinition。

策略模式

策略模式（Strategy attern）是指定义了算法家族、分别封装起来，让它们之间可以互相替换，此模式让算法的变化不会影响到使用算法的用户。

策略模式的应用场景

1、假如系统中有很多类，而他们的区别仅仅在于他们的行为不同。

2、一个系统需要动态地在几种算法中选择一种。

用策略模式实现选择支付方式的业务场景

大家都知道，我们咕泡学院的架构师课程经常会有优惠活动，优惠策略会有很多种可能如：领取优惠券抵扣、返现促销、拼团优惠。下面我们用代码来模拟，首先我们创建一个促销策略的抽象 PromotionStrategy：

```
package com.gupaoedu.vip.pattern.strategy.promotion;

/**
 * 促销策略抽象
 * Created by Tom
 */
public interface PromotionStrategy {
    void doPromotion();
}
```

然后分别创建优惠券抵扣策略 CouponStrategy 类、返现促销策略 CashbackStrategy 类、拼团优惠策略 GroupbuyStrategy 类和无优惠策略 EmptyStrategy 类：

CouponStrategy 类：

```
package com.gupaoedu.vip.pattern.strategy.promotion;
/**
 * 优惠券
 * Created by Tom
 */
public class CouponStrategy implements PromotionStrategy {
    public void doPromotion() {
        System.out.println("领取优惠券,课程的价格直接减优惠券面值抵扣");
    }
}
```

CashbackStrategy 类：

```
package com.gupaoedu.vip.pattern.strategy.promotion;
/**
 * 返现活动
 * Created by Tom
 */
public class CashbackStrategy implements PromotionStrategy {
    public void doPromotion() {
        System.out.println("返现促销,返回的金额转到支付宝账号");
    }
}
```

GroupbuyStrategy 类：

```
package com.gupaoedu.vip.pattern.strategy.promotion;
/**
 * 拼团优惠
 * Created by Tom
 */
public class GroupbuyStrategy implements PromotionStrategy{
    public void doPromotion() {
        System.out.println("拼团，满 20 人成团，全团享受团购价");
    }
}
```

EmptyStrategy 类：

```
package com.gupaoedu.vip.pattern.strategy.promotion;
/**
 * 无优惠
 * Created by Tom
 */
public class EmptyStrategy implements PromotionStrategy {
    public void doPromotion() {
        System.out.println("无促销活动");
    }
}
```

然后创建促销活动方案 PromotionActivity 类：

```
package com.gupaoedu.vip.pattern.strategy.promotion;

/**
 * 优惠活动
 * Created by Tom
 */
public class PromotionActivity {
    private PromotionStrategy promotionStrategy;

    public PromotionActivity(PromotionStrategy promotionStrategy) {
        this.promotionStrategy = promotionStrategy;
    }
    public void execute(){
        promotionStrategy.doPromotion();
    }
}
```

编写客户端测试类：

```
public static void main(String[] args) {
    PromotionActivity activity618 = new PromotionActivity(new CouponStrategy());
    PromotionActivity activity1111 = new PromotionActivity(new CashbackStrategy());

    activity618.execute();
    activity1111.execute();
}
```

此时，小伙伴们会发现，如果把上面这段测试代码放到实际的业务场景其实并不实用。因为我们做活动时候往往是要根据不同的需求对促销策略进行动态选择的，并不会一次性执行多种优惠。所以，我们的代码通常会这样写：

```
public static void main(String[] args) {
    PromotionActivity promotionActivity = null;

    String promotionKey = "COUPON";

    if(StringUtils.equals(promotionKey,"COUPON")){
        promotionActivity = new PromotionActivity(new CouponStrategy());
    }else if(StringUtils.equals(promotionKey,"CASHBACK")){
        promotionActivity = new PromotionActivity(new CashbackStrategy());
    }//......
    promotionActivity.execute();
}
```

这样改造之后，满足了业务需求，客户可根据自己的需求选择不同的优惠策略了。但是，经过一段时间的业务积累，我们的促销活动会越来越多。于是，我们的程序猿小哥哥就忙不赢了，每次上活动之前都要通宵改代码，而且要做重复测试，判断逻辑可能也变得越来越复杂。这时候，我们是不需要思考代码是不是应该重构了？回顾我们之前学过的设计模式应该如何来优化这段代码呢？其实，我们可以结合单例模式和工厂模式。创建

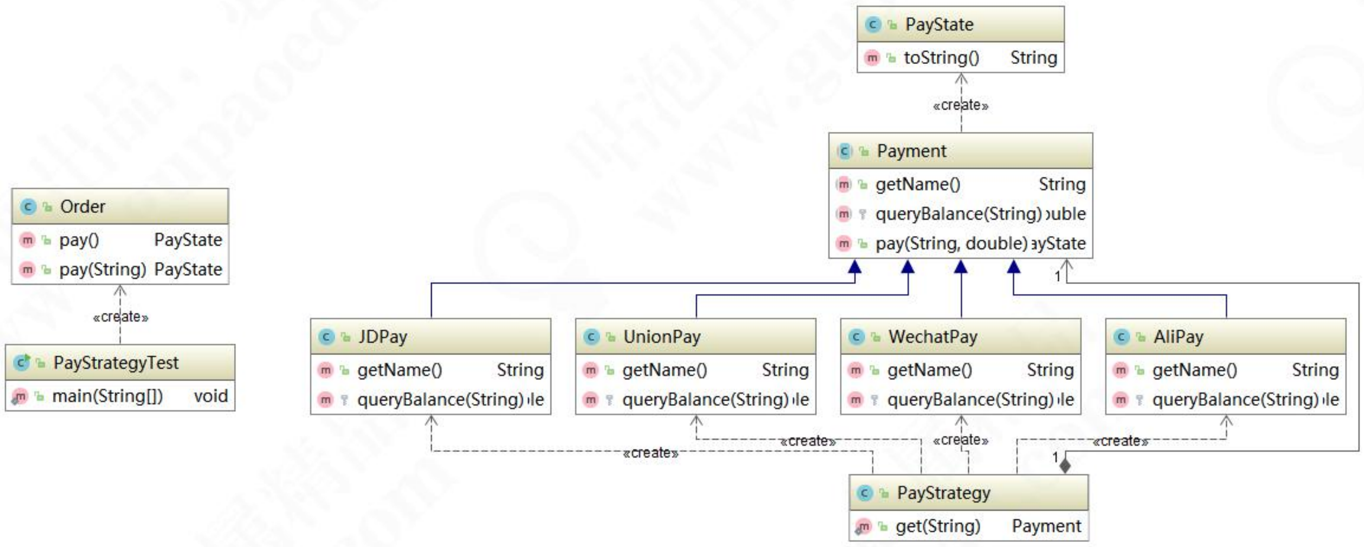
PromotionStrategyFactory 类：

```java
package com.gupaoedu.vip.pattern.strategy.promotion;

import java.util.HashMap;
import java.util.Map;

/**
 * 促销策略工厂
 * Created by Tom
 */
public class PromotionStrategyFactory {
    private static Map<String,PromotionStrategy> PROMOTION_STRATEGY_MAP = new HashMap<String,
PromotionStrategy>();
    static {
        PROMOTION_STRATEGY_MAP.put(PromotionKey.COUPON,new CouponStrategy());
        PROMOTION_STRATEGY_MAP.put(PromotionKey.CASHBACK,new CashbackStrategy());
        PROMOTION_STRATEGY_MAP.put(PromotionKey.GROUPBUY,new GroupbuyStrategy());
    }

    private static final PromotionStrategy NON_PROMOTION = new EmptyStrategy();

    private PromotionStrategyFactory(){}

    public static PromotionStrategy getPromotionStrategy(String promotionKey){
        PromotionStrategy promotionStrategy = PROMOTION_STRATEGY_MAP.get(promotionKey);
        return promotionStrategy == null ? NON_PROMOTION : promotionStrategy;
    }

    private interface PromotionKey{
        String COUPON = "COUPON";
        String CASHBACK = "CASHBACK";
        String GROUPBUY = "GROUPBUY";
    }
}
```

这时候我们客户端代码就应该这样写了：

```java
public static void main(String[] args) {
    String promotionKey = "GROUPBUY";
    PromotionActivity promotionActivity = new
PromotionActivity(PromotionStrategyFactory.getPromotionStrategy(promotionKey));
    promotionActivity.execute();
}
```

代码优化之后，是不是我们程序猿小哥哥的维护工作就轻松了？每次上新活动，不影响原来的代码逻辑。为了加深对策略模式的理解，我们再来举一个案例。相信小伙伴们都用过支付宝、微信支付、银联支付以及京东白条。一个常见的应用场景就是大家在下单支付时会提示选择支付方式，如果用户未选，系统也会默认好推荐的支付方式进行结算。来看一下类图，下面我们用策略模式来模拟此业务场景：



创建 Payment 抽象类，定义支付规范和支付逻辑，代码如下：

```java
package com.gupaoedu.vip.pattern.strategy.pay.payport;

import com.gupaoedu.vip.pattern.strategy.pay.PayState;

/**
 * 支付渠道
 * Created by Tom.
 */
public abstract class Payment {

    //支付类型
    public abstract String getName();

    //查询余额
    protected abstract double queryBalance(String uid);

    //扣款支付
    public PayState pay(String uid,double amount) {
        if(queryBalance(uid) < amount){
            return new PayState(500,"支付失败","余额不足");
        }
        return new PayState(200,"支付成功","支付金额：" + amount);
    }
}
```

分别创建具体的支付方式，支付宝 AliPay 类：

```
package com.gupaoedu.vip.pattern.strategy.pay.payport;

/**
 * Created by Tom.
 */
public class AliPay extends Payment {

    public String getName() {
        return "支付宝";
    }

    protected double queryBalance(String uid) {
        return 900;
    }
}
```

京东白条 JDPay 类：

```
package com.gupaoedu.vip.pattern.strategy.pay.payport;
/**
 * Created by Tom.
 */
public class JDPay extends Payment {

    public String getName() {
        return "京东白条";
    }

    protected double queryBalance(String uid) {
        return 500;
    }
}
```

微信支付 WechatPay 类：

```
package com.gupaoedu.vip.pattern.strategy.pay.payport;
/**
 * Created by Tom.
 */
public class WechatPay extends Payment {

    public String getName() {
        return "微信支付";
    }

    protected double queryBalance(String uid) {
        return 256;
    }
}
```

银联支付 UnionPay 类：

```
package com.gupaoedu.vip.pattern.strategy.pay.payport;
/**
 * Created by Tom.
 */
public class UnionPay extends Payment {
    public String getName() {
        return "银联支付";
    }

    protected double queryBalance(String uid) {
        return 120;
    }
}
```

创建支付状态的包装类 PayState:

```
package com.gupaoedu.vip.pattern.strategy.pay;

/**
 * 支付完成以后的状态
 * Created by Tom.
 */
public class PayState {
    private int code;
    private Object data;
    private String msg;

    public PayState(int code, String msg,Object data) {
        this.code = code;
        this.data = data;
        this.msg = msg;
    }
    public String toString(){
        return ("支付状态：[" + code + "]," + msg + ",交易详情：" + data);
    }
}
```

创建支付策略管理类：

```
package com.gupaoedu.vip.pattern.strategy.pay.payport;

import java.util.HashMap;
import java.util.Map;

/**
 * 支付策略管理
 * Created by Tom.
 */
public class PayStrategy {
    public static final String ALI_PAY = "AliPay";
    public static final String JD_PAY = "JdPay";
    public static final String UNION_PAY = "UnionPay";
    public static final String WECHAT_PAY = "WechatPay";
    public static final String DEFAULT_PAY = ALI_PAY;

    private static Map<String,Payment> payStrategy = new HashMap<String,Payment>();
    static {
        payStrategy.put(ALI_PAY,new AliPay());
        payStrategy.put(WECHAT_PAY,new WechatPay());
        payStrategy.put(UNION_PAY,new UnionPay());
        payStrategy.put(JD_PAY,new JDPay());
    }

    public static Payment get(String payKey){
        if(!payStrategy.containsKey(payKey)){
            return payStrategy.get(DEFAULT_PAY);
        }
        return payStrategy.get(payKey);
    }
}
```

创建订单 Order 类：

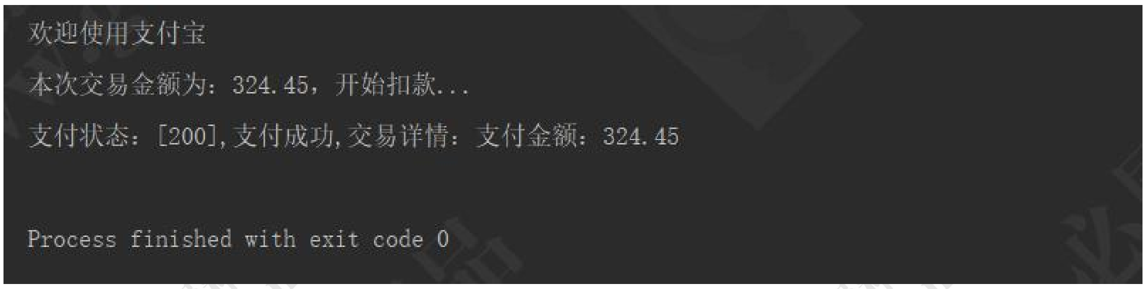
```
package com.gupaoedu.vip.pattern.strategy.pay;

import com.gupaoedu.vip.pattern.strategy.pay.payport.PayStrategy;
import com.gupaoedu.vip.pattern.strategy.pay.payport.Payment;
```

```java
/**
 * Created by Tom.
 */
public class Order {
    private String uid;
    private String orderId;
    private double amount;

    public Order(String uid,String orderId,double amount){
        this.uid = uid;
        this.orderId = orderId;
        this.amount = amount;
    }

    //完美地解决了 switch 的过程，不需要在代码逻辑中写 switch 了
    //更不需要写 if    else if
    public PayState pay(){
        return pay(PayStrategy.DEFAULT_PAY);
    }

    public PayState pay(String payKey){
        Payment payment = PayStrategy.get(payKey);
        System.out.println("欢迎使用" + payment.getName());
        System.out.println("本次交易金额为：" + amount + "，开始扣款...");
        return payment.pay(uid,amount);
    }
}
```

测试代码：

```java
package com.gupaoedu.vip.pattern.strategy.pay;

import com.gupaoedu.vip.pattern.strategy.pay.payport.PayStrategy;

public class PayStrategyTest {

    public static void main(String[] args) {

        //省略把商品添加到购物车，再从购物车下单
        //直接从点单开始
        Order order = new Order("1","20180311001000009",324.45);

        //开始支付，选择微信支付、支付宝、银联卡、京东白条、财付通
        //每个渠道它支付的具体算法是不一样的
        //基本算法固定的

        //这个值是在支付的时候才决定用哪个值
        System.out.println(order.pay(PayStrategy.ALI_PAY));

    }
}
```

运行结果:



```
欢迎使用支付宝
本次交易金额为：324.45，开始扣款...
支付状态：[200],支付成功,交易详情：支付金额：324.45

Process finished with exit code 0
```

希望通过大家耳熟能详的业务场景来举例，让小伙伴们更深刻地理解策略模式。希望小伙伴们在面试和工作体现出自己的优势。

策略模式在 JDK 源码中的体现

首先来看一个比较常用的比较器 Comparator 接口，我们看到的一个大家常用的 compare()方法，就是一个策略抽象实现:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

Comparator 抽象下面有非常多的实现类，我们经常会把 Comparator 作为参数传入作为排序策略，例如 Arrays 类的 parallelSort 方法等:

```
public class Arrays {
    ...
    public static <T> void parallelSort(T[] a, int fromIndex, int toIndex,
                                        Comparator<? super T> cmp) {
    ...
    }
    ...
}
```

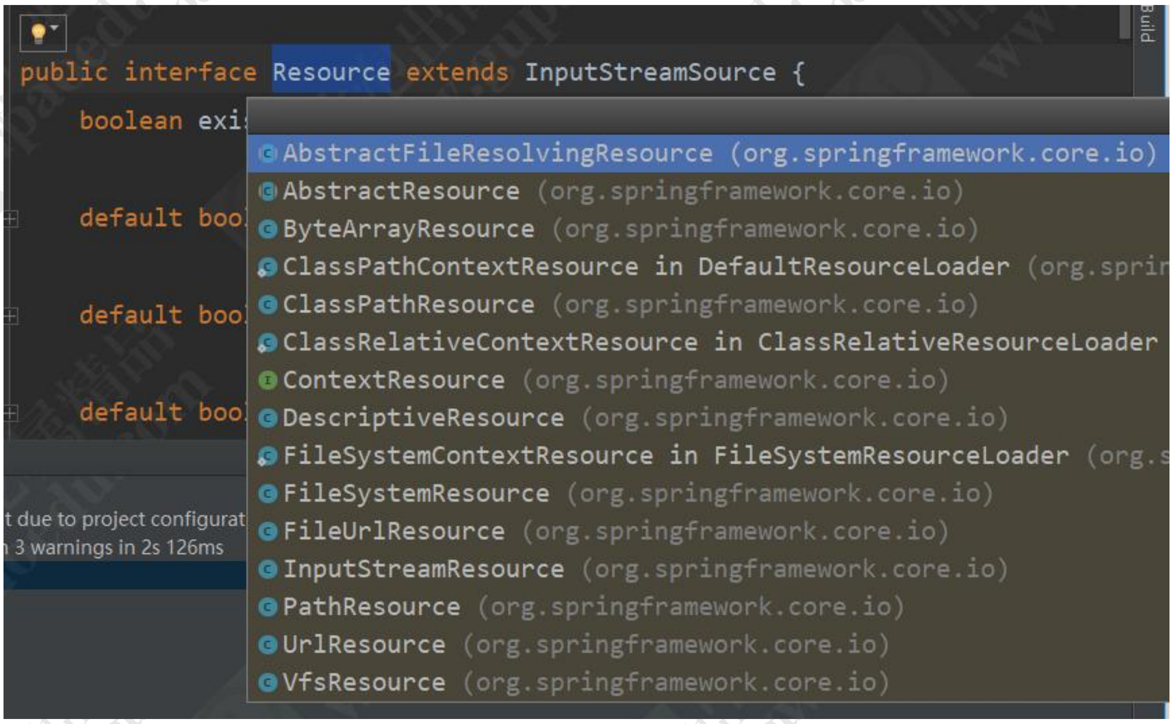
还有 TreeMap 的构造方法:

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
{
   ...
    public TreeMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
    }
    ...
}
```

这就是 Comparator 在 JDK 源码中的应用。那我们来看策略模式在 Spring 源码中的应用，来看 Resource 类：

```
package org.springframework.core.io;

import java.io.File;
import java.io.IOException;
import java.net.URI;
import java.net.URL;
import java.nio.channels.Channels;
import java.nio.channels.ReadableByteChannel;
import org.springframework.lang.Nullable;

public interface Resource extends InputStreamSource {
    boolean exists();

    default boolean isReadable() {
        return true;
    }

    default boolean isOpen() {
        return false;
    }

    default boolean isFile() {
        return false;
    }

    URL getURL() throws IOException;

    URI getURI() throws IOException;

    File getFile() throws IOException;
    default ReadableByteChannel readableChannel() throws IOException {
        return Channels.newChannel(this.getInputStream());
    }

    long contentLength() throws IOException;

    long lastModified() throws IOException;

    Resource createRelative(String var1) throws IOException;

    @Nullable
    String getFilename();

    String getDescription();
}
```

我们虽然没有直接使用 Resource 类，但是我们经常使用它的子类，例如：



```
public interface Resource extends InputStreamSource {
    boolean exi
    default boo
    default boo
    default boo
```

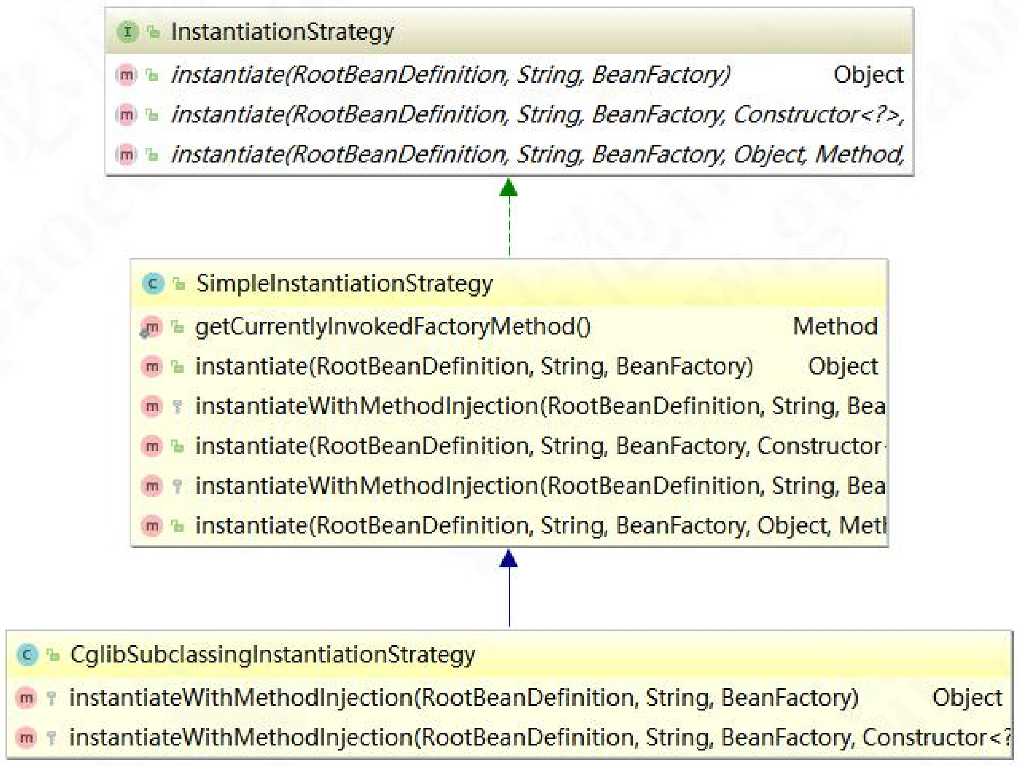
```
AbstractFileResolvingResource (org.springframework.core.io)
AbstractResource (org.springframework.core.io)
ByteArrayResource (org.springframework.core.io)
ClassPathContextResource in DefaultResourceLoader (org.sprin
ClassPathResource (org.springframework.core.io)
ClassRelativeContextResource in ClassRelativeResourceLoader
ContextResource (org.springframework.core.io)
DescriptiveResource (org.springframework.core.io)
FileSystemContextResource in FileSystemResourceLoader (org.s
FileSystemResource (org.springframework.core.io)
FileUrlResource (org.springframework.core.io)
InputStreamResource (org.springframework.core.io)
PathResource (org.springframework.core.io)
UrlResource (org.springframework.core.io)
VfsResource (org.springframework.core.io)
```

t due to project configurat
n 3 warnings in 2s 126ms

还有一个非常典型的场景，Spring 的初始化也采用了策略模式，不同的类型的类采用不同的初始化策略。首先有一个 InstantiationStrategy 接口，我们来看一下源码：

```
package org.springframework.beans.factory.support;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.lang.Nullable;

public interface InstantiationStrategy {
    Object instantiate(RootBeanDefinition var1, @Nullable String var2, BeanFactory var3) throws
BeansException;

    Object instantiate(RootBeanDefinition var1, @Nullable String var2, BeanFactory var3,
Constructor<?> var4, @Nullable Object... var5) throws BeansException;

    Object instantiate(RootBeanDefinition var1, @Nullable String var2, BeanFactory var3, @Nullable
Object var4, Method var5, @Nullable Object... var6) throws BeansException;
}
```

顶层的策略抽象非常简单，但是它下面有两种策略 SimpleInstantiationStrategy 和 CglibSubclassingInstantiationStrategy，我们看一下类图：


InstantiationStrategy
instantiate(RootBeanDefinition, String, BeanFactory) Object
instantiate(RootBeanDefinition, String, BeanFactory, Constructor<?>,
instantiate(RootBeanDefinition, String, BeanFactory, Object, Method,
SimpleInstantiationStrategy
getCurrentlyInvokedFactoryMethod() Method
instantiate(RootBeanDefinition, String, BeanFactory) Object
instantiateWithMethodInjection(RootBeanDefinition, String, Bea
instantiate(RootBeanDefinition, String, BeanFactory, Constructor
instantiateWithMethodInjection(RootBeanDefinition, String, Bea
instantiate(RootBeanDefinition, String, BeanFactory, Object, Meth
CglibSubclassingInstantiationStrategy
instantiateWithMethodInjection(RootBeanDefinition, String, BeanFactory) Object
instantiateWithMethodInjection(RootBeanDefinition, String, BeanFactory, Constructor<?

打开类图我们还发现 CglibSubclassingInstantiationStrategy 策略类还继承了 SimpleInstantiationStrategy 类，说明在实际应用中多种策略之间还可以继承使用。小

伙们可以作为一个参考，在实际业务场景中，可以根据需要来设计。

策略模式的优缺点

优点：

1、策略模式符合开闭原则。

2、避免使用多重条件转移语句，如 if...else...语句、switch 语句

3、使用策略模式可以提高算法的保密性和安全性。

缺点：

1、客户端必须知道所有的策略，并且自行决定使用哪一个策略类。

2、代码中会产生非常多策略类，增加维护难度。

委派模式与策略模式综合应用

在上面的代码中我们列举了非常几个业务场景，相信小伙伴对委派模式和策略模式有了非常深刻的理解了。现在，我们再来回顾一下，DispatcherServlet 的委派逻辑，代码如下：

```
package com.gupaoedu.vip.pattern.delegate.mvc;

...

public class DispatcherServlet extends HttpServlet{

   private void doDispatch(HttpServletRequest request, HttpServletResponse response) throws
Exception{

       String uri = request.getRequestURI();
       String mid = request.getParameter("mid");

       if("getMemberById".equals(uri)){
           new MemberController().getMemberById(mid);
```

```
        }else if("getOrderById".equals(uri)){
            new OrderController().getOrderById(mid);
        }else if("logout".equals(uri)){
            new SystemController().logout();
        }else {
            response.getWriter().write("404 Not Found!!");
        }
    }

  ...

}
```

这样的代码扩展性不太优雅，也不现实，因为我们实际项目中一定不止这几个 Controller，往往是成千上万个 Controller，显然，我们不能写成千上万个 if...else... 。那么我们如何来改造呢？小伙伴们一定首先就想到了策略模式，来看一下我是怎么优化的：

```
package com.gupaoedu.vip.pattern.delegate.mvc;

import com.gupaoedu.vip.pattern.delegate.mvc.controllers.MemberController;
import com.gupaoedu.vip.pattern.delegate.mvc.controllers.OrderController;
import com.gupaoedu.vip.pattern.delegate.mvc.controllers.SystemController;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

/**
 * 相当于是项目经理的角色
 * Created by Tom.
 */
public class DispatcherServlet extends HttpServlet{

    private List<Handler> handlerMapping = new ArrayList<Handler>();

    public void init() throws ServletException {
        try {
            Class<?> memberControllerClass = MemberController.class;
            handlerMapping.add(new Handler()
                    .setController(memberControllerClass.newInstance())
                    .setMethod(memberControllerClass.getMethod("getMemberById", new
Class[]{String.class}))
                    .setUrl("/web/getMemberById.json"));
        }catch(Exception e){

        }
    }

    private void doDispatch(HttpServletRequest request, HttpServletResponse response){

        //1、获取用户请求的 url
        //   如果按照 J2EE 的标准、每个 url 对对应一个 Serlvet，url 由浏览器输入
       String uri = request.getRequestURI();

        //2、Servlet 拿到 url 以后，要做权衡（要做判断，要做选择）
        //   根据用户请求的 URL，去找到这个 url 对应的某一个 java 类的方法

        //3、通过拿到的 URL 去 handlerMapping（我们把它认为是策略常量）
        Handler handle = null;
        for (Handler h: handlerMapping) {
            if(uri.equals(h.getUrl())){
                handle = h;
                break;
            }
        }

        //4、将具体的任务分发给 Method（通过反射去调用其对应的方法）
        Object object = null;
        try {
            object =
handle.getMethod().invoke(handle.getController(),request.getParameter("mid"));
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }

        //5、获取到 Method 执行的结果，通过 Response 返回出去
//          response.getWriter().write();
    }
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        try {
            doDispatch(req,resp);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    class Handler{

        private Object controller;
        private Method method;
        private String url;

        public Object getController() {
            return controller;
        }

        public Handler setController(Object controller) {
            this.controller = controller;
            return this;
        }

        public Method getMethod() {
            return method;
        }

        public Handler setMethod(Method method) {
            this.method = method;
            return this;
        }

        public String getUrl() {
            return url;
        }

        public Handler setUrl(String url) {
            this.url = url;
            return this;
        }
    }
}
```

上面的代码我结合了策略模式、工厂模式、单例模式。当然，我的优化方案不一定是最完美的，仅代表个人观点。感兴趣的小伙伴可以继续思考，如何让这段代码变得更优雅。当然，我们后面在讲 Spring 源码时还会讲到 DispatcherServlet 的相关内容。