课程目标

1、掌握门面模式和装饰器模式的特征和应用场景

2、理解装饰器模式和代理模式的根本区别。

3、了解门面模式的优、缺点。

4、了解装饰器模式的优、缺点。

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、深刻了解门面模式和装饰器模式的应用场景。

门面模式

门面模式（Facade Pattern）又叫外观模式，提供了一个统一的接口，用来访问子系统中的一群接口。其主要特征是定义了一个高层接口，让子系统更容易使用，属于结构性模式。

> **原文:**Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
>
> **解释：**要求一个子系统的外部与其内部的通信必须通过一个同一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

其实，在我们日常的编码工作中，我们都在有意无意地大量使用门面模式，但凡只要高层模块需要调度多个子系统（2 个以上类对象），我们都会自觉地创建一个新类封装这些子系统，提供精简接口，让高层模块可以更加容易间接调用这些子系统的功能。尤其是现阶段各种第三方 SDK，各种开源类库，很大概率都会使用门面模式。尤其是你觉得调用越方便的，门面模式使用的一般更多。

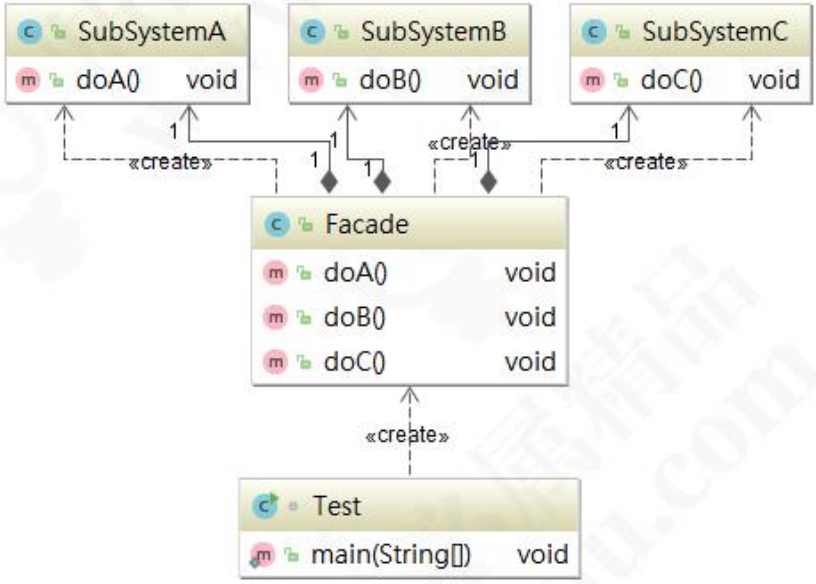
门面模式的应用场景

1、子系统越来越复杂，增加门面模式提供简单接口

2、构建多层系统结构，利用门面对象作为每层的入口，简化层间调用。

门面模式的通用写法

首先来看门面模式的 UML 类图：



门面模式主要包含 2 种角色：

外观角色（Facade）：也称 门面角色，系统对外的统一接口；

子系统角色（SubSystem）：可以同时有一个或多个 SubSystem。每个 SubSytem 都不是一个单独的类，而是一个类的集合。SubSystem 并不知道 Facade 的存在，对于 SubSystem 而言，Facade 只是另一个客户端而已（即 Facade 对 SubSystem 透明）。

下面是门面模式的通用代码，首先分别创建 3 个子系统的业务逻辑 SubSystemA、SubSystemB、SubSystemC，代码很简单：

```
public class SubSystemA {
    public void doA() {
        System.out.println("doing A stuff");
    }
}
public class SubSystemB {
```

```
    public void doB() {
        System.out.println("doing B stuff");
    }
}
public class SubSystemC {
    public void doC() {
        System.out.println("doing C stuff");
    }
}
```

然后，创建外观角色 Facade 类：

```
public class Facade {
    private SubSystemA a = new SubSystemA();
    private SubSystemB b = new SubSystemB();
    private SubSystemC c = new SubSystemC();

    // 对外接口
    public void doA() {
        this.a.doA();
    }

    // 对外接口
    public void doB() {
        this.b.doB();
    }

    // 对外接口
    public void doC() {
        this.c.doC();
    }
}
```

来看客户端代码：

```
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.doA();
        facade.doB();
        facade.doC();
    }
```

门面模式业务场景实例

Gper 社区上线了一个积分兑换礼品的商城，这礼品商城中的大部分功能并不是全部重新开发的，而是要去对接已有的各个子系统（如下图所示）：

这些子系统可能涉及到积分系统、支付系统、物流系统的接口调用。如果所有的接口调用全部由前端发送网络请求去调用现有接口的话，一则会增加前端开发人员的难度，二则会增加一些网络请求影响页面性能。这个时候就可以发挥门面模式的优势了。将所有现成的接口全部整合到一个类中，由后端提供统一的接口给前端调用，这样前端开发人员就不需要关心各接口的业务关系，只需要把精力集中在页面交互上。下面我们用代码来模拟一下这个场景。

首先，创建礼品的实体类 GiftInfo：

```
public class GiftInfo {
    private String name;

    public GiftInfo(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

然后，编写各个子系统的业务逻辑代码，分别创建积分系统 QualifyService 类：

```
public class QualifyService {
    public boolean isAvailable(GiftInfo giftInfo){
        System.out.println("校验" + giftInfo.getName() + " 积分资格通过,库存通过");
        return true;
    }
}
```

支付系统 PaymentService 类：

```
public class PaymentService {
```

```
    public boolean pay(GiftInfo pointsGift){
        //扣减积分
        System.out.println("支付" + pointsGift.getName() + " 积分成功");
        return true;
    }
}
```

物流系统 ShippingService 类：

```
public class ShippingService {

    //发货
    public String delivery(GiftInfo giftInfo){
        //物流系统的对接逻辑
        System.out.println(giftInfo.getName() + "进入物流系统");
        String shippingOrderNo = "666";
        return shippingOrderNo;
    }
}
```

然后创建外观角色 GiftFacdeService 类，对外只开放一个兑换礼物的 exchange()方法，在 exchange()方法内部整合 3 个子系统的所有功能。

```
public class GiftFacadeService {
    private QualifyService qualifyService = new QualifyService();
    private PaymentService pointsPaymentService = new PaymentService();
    private ShippingService shippingService = new ShippingService();

    //兑换
    public void exchange(GiftInfo giftInfo){
        if(qualifyService.isAvailable(giftInfo)){
            //资格校验通过
            if(pointsPaymentService.pay(giftInfo)){
                //如果支付积分成功
                String shippingOrderNo = shippingService.delivery(giftInfo);
                System.out.println("物流系统下单成功,订单号是:"+shippingOrderNo);
            }
        }
    }
}
```

最后，来看客户端代码：

```
    public static void main(String[] args) {
        GiftInfo giftInfo = new GiftInfo("《Spring 5 核心原理》");
        GiftFacadeService giftFacadeService = new GiftFacadeService();
        giftFacadeService.exchange(giftInfo);
    }
```

运行结果如下：

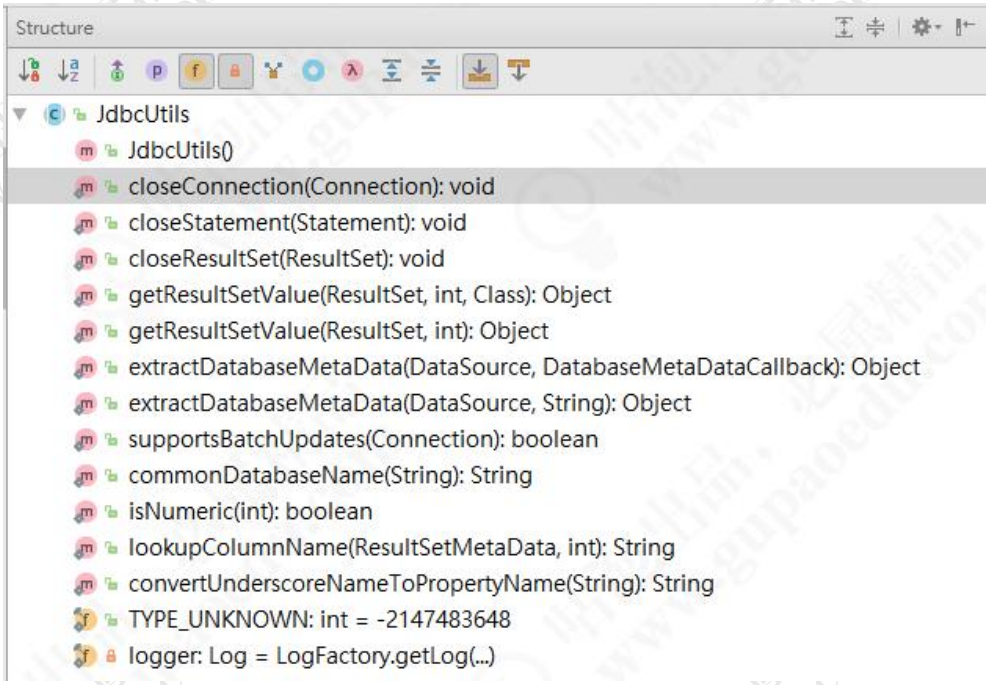
通过这样一个案例对比之后，相信大家对门面模式的印象非常深刻了。

门面模式在源码中的应用

下面我们来门面模式在源码中的应用，先来看 Spring JDBC 模块下的 JdbcUtils 类，它封装了和 JDBC 相关的所有操作，它一个代码片段：

```
public abstract class JdbcUtils {
    public static final int TYPE_UNKNOWN = -2147483648;
    private static final Log logger = LogFactory.getLog(JdbcUtils.class);

    public JdbcUtils() {
    }

    public static void closeConnection(Connection con) {
        if(con != null) {
            try {
                con.close();
            } catch (SQLException var2) {
                logger.debug("Could not close JDBC Connection", var2);
            } catch (Throwable var3) {
                logger.debug("Unexpected exception on closing JDBC Connection", var3);
            }
        }

    }

    public static void closeStatement(Statement stmt) {
        if(stmt != null) {
            try {
                stmt.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC Statement", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC Statement", var3);
            }
        }

    }

    public static void closeResultSet(ResultSet rs) {
        if(rs != null) {
            try {
                rs.close();
            } catch (SQLException var2) {
                logger.trace("Could not close JDBC ResultSet", var2);
            } catch (Throwable var3) {
                logger.trace("Unexpected exception on closing JDBC ResultSet", var3);
            }
        }

    }
     ...
}
```

其他更多的操作，看它的结构就非常清楚了：



再来看一个 MyBatis 中的 Configuration 类。它其中有很多 new 开头的方法，来看一下源代码：

```
public MetaObject newMetaObject(Object object) {
        return MetaObject.forObject(object, this.objectFactory, this.objectWrapperFactory, this.reflectorFactory);
    }

    public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql
boundSql) {
        ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement,
parameterObject, boundSql);
        parameterHandler = (ParameterHandler)this.interceptorChain.pluginAll(parameterHandler);
        return parameterHandler;
    }

    public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds,
ParameterHandler parameterHandler, ResultHandler resultHandler, BoundSql boundSql) {
        ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler,
resultHandler, boundSql, rowBounds);
        ResultSetHandler resultSetHandler = (ResultSetHandler)this.interceptorChain.pluginAll(resultSetHandler);
        return resultSetHandler;
    }

    public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object
parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
        StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject,
rowBounds, resultHandler, boundSql);
        StatementHandler statementHandler = (StatementHandler)this.interceptorChain.pluginAll(statementHandler);
        return statementHandler;
    }

    public Executor newExecutor(Transaction transaction) {
        return this.newExecutor(transaction, this.defaultExecutorType);
    }
```

上面的这些方法都是对 JDBC 中关键组件操作的封装。另外地在 Tomcat 的源码中也有体现，也非常的有意思。举个例子 RequestFacade 类，来看源码：

```
public class RequestFacade implements HttpServletRequest {
...
@Override
    public String getContentType() {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        return request.getContentType();
    }


    @Override
    public ServletInputStream getInputStream() throws IOException {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        return request.getInputStream();
    }


    @Override
    public String getParameter(String name) {

        if (request == null) {
            throw new IllegalStateException(
                            sm.getString("requestFacade.nullRequest"));
        }

        if (Globals.IS_SECURITY_ENABLED){
            return AccessController.doPrivileged(
                new GetParameterPrivilegedAction(name));
        } else {
            return request.getParameter(name);
        }
    }
...
}
```

我们看名字就知道它用了门面模式。它封装了非常多的 request 的操作，也整合了很多 servlet-api 以外的一些内容，给用户使用提供了很大便捷。同样，Tomcat 对 Response 和 Session 当也封装了 ResponseFacade 和 StandardSessionFacade 类，感兴趣的小伙伴可以去深入了解一下。

门面模式的优缺点

优点：

1、简化了调用过程，无需深入了解子系统，以防给子系统带来风险。

2、减少系统依赖、松散耦合

3、更好地划分访问层次，提高了安全性

4、遵循迪米特法则，即最少知道原则。

缺点：

1、当增加子系统和扩展子系统行为时，可能容易带来未知风险

2、不符合开闭原则

3、某些情况下可能违背单一职责原则。

装饰器模式

装饰器模式（Decorator Pattern）,也称为包装模式（Wrapper Pattern）是指在不改变原有对象的基础之上，将功能附加到对象上，提供了比继承更有弹性的替代方案（扩展原有对象的功能），属于结构型模式。

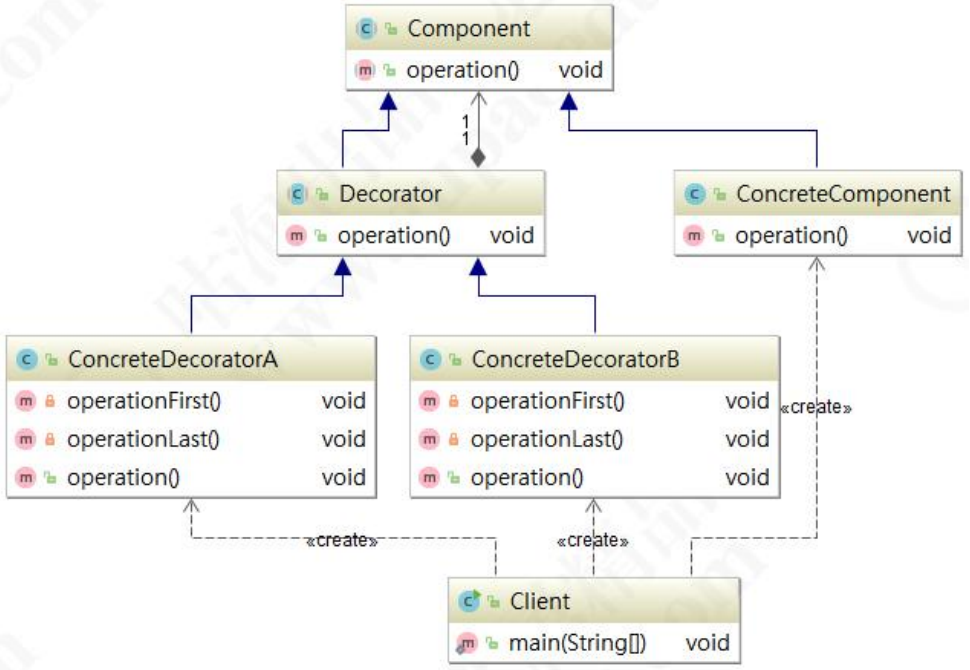
> **原文：** Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
>
> **解释：** 动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

装饰器模式的核心是功能扩展。使用装饰器模式可以透明且动态地扩展类的功能。

装饰器模式主要用于透明且动态地扩展类的功能。其实现原理为：让装饰器实现被包装类（Concrete Component）相同的接口（Component）（使得装饰器与被扩展类类型一致），并在构造函数中传入该接口（Component）对象，然后就可以在接口需要实现的方法中在被包装类对象的现有功能上添加新功能了。而且由于装饰器与被包装类属于同一类型（均为 Component），且构造函数的参数为其实现接口类（Component），因此装饰器模式具备嵌套扩展功能，这样我们就能使用装饰器模式一层一层的对最底层被包装类进行功能扩展了。

首先看下装饰器模式的通用 UML 类图：

从 UML 类图中，我们可以看到，装饰器模式 主要包含四种角色：



抽象组件（Component）：可以是一个接口或者抽象类，其充当被装饰类的原始对象，规定了被装饰对象的行为；

具体组件（ConcreteComponent）：实现/继承 Component 的一个具体对象，也即 被装饰对象；

抽象装饰器（Decorator）：通用的装饰 ConcreteComponent 的装饰器，其内部必然有一个属性指向 Component 抽象组件；其实现一般是一个抽象类，主要是为了让其子类按照其构造形式传入一个 Component 抽象组件，这是强制的通用行为（当然，如果系统中装饰逻辑单一，并不需要实现许多装饰器，那么我们可以直接省略该类，而直接实现一个具体装饰器（ConcreteDecorator）即可）；

具体装饰器 (ConcreteDecorator)：Decorator 的具体实现类，理论上，每个 ConcreteDecorator 都扩展了 Component 对象的一种功能；

总结：装饰器模式角色分配符合设计模式里氏替换原则，依赖倒置原则，从而使得其具备很强的扩展性，最终满足开闭原则。

装饰器模式的应用场景

装饰器模式在我们生活中应用也比较多如给煎饼加鸡蛋；给蛋糕加上一些水果；给房子装修等，为

对象扩展一些额外的职责。装饰器在代码程序中适用于以下场景：

1、用于扩展一个类的功能或给一个类添加附加职责。

2、动态的给一个对象添加功能，这些功能可以再动态的撤销。

3、需要为一批的兄弟类进行改装或加装功能。

来看一个这样的场景，上班族白领其实大多有睡懒觉的习惯，每天早上上班都是踩点，于是很多小伙伴为了多赖一会儿床都不吃早餐。那么，也有些小伙伴可能在上班路上碰到卖煎饼的路边摊，都会顺带一个到公司茶水间吃早餐。卖煎饼的大姐可以给你的煎饼加鸡蛋，也可以加香肠（如下图，PS：我买煎饼一般都要求不加生菜）。

煎饼侠（大鹏）

水果蛋糕

下面我们用代码还原一下码农的生活。首先创建一个煎饼 Battercake 类:

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

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

    protected String getMsg(){
        return "煎饼";
    }

    public int getPrice(){
        return 5;
    }
}
```

创建一个加鸡蛋的煎饼 BattercakeWithEgg 类:

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
```

```
 * Created by Tom.
 */
public class BattercakeWithEgg extends Battercake{
    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 个鸡蛋";
    }

    @Override
    //加一个鸡蛋加 1 块钱
    public int getPrice() {
        return super.getPrice() + 1;
    }
}
```

再创建一个既加鸡蛋又加香肠的 BattercakeWithEggAndSausage 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
 * Created by Tom.
 */
public class BattercakeWithEggAndSausage extends BattercakeWithEgg{
    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 根香肠";
    }

    @Override
    //加一个香肠加 2 块钱
    public int getPrice() {
        return super.getPrice() + 2;
    }
}
```

编写客户端测试代码：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v1;

/**
 * Created by Tom.
 */
public class BattercakeTest {
    public static void main(String[] args) {

        Battercake battercake = new Battercake();
        System.out.println(battercake.getMsg() + ",总价格：" + battercake.getPrice());

        Battercake battercakeWithEgg = new BattercakeWithEgg();
        System.out.println(battercakeWithEgg.getMsg() + ",总价格：" + battercakeWithEgg.getPrice());

        Battercake battercakeWithEggAndSausage = new BattercakeWithEggAndSausage();
        System.out.println(battercakeWithEggAndSausage.getMsg() + ",总价格：" + battercakeWithEggAndSausage.getPrice());

    }
}
```

运行结果：

运行结果没有问题。但是，如果用户需要一个加 2 个鸡蛋加 1 根香肠的煎饼，那么用我们现在的类结构是创建不出来的，也无法自动计算出价格，除非再创建一个类做定制。如果需求再变，一直加定制显然是不科学的。那么下面我们就用装饰器模式来解决上面的问题。首先创建一个建煎饼的抽象 Battercake 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public abstract class Battercake {
    protected abstract String getMsg();
    protected abstract int getPrice();
}
```

创建一个基本的煎饼（或者叫基础套餐）BaseBattercake：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class BaseBattercake extends Battercake {
    protected String getMsg(){
        return "煎饼";
    }

    public int getPrice(){ return 5;  }
}
```

然后，再创建一个扩展套餐的抽象装饰器 BattercakeDecotator 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public abstract class BattercakeDecorator extends Battercake {
    //静态代理，委派
    private Battercake battercake;

    public BattercakeDecorator(Battercake battercake) {
        this.battercake = battercake;
    }
    protected abstract void doSomething();

    @Override
    protected String getMsg() {
        return this.battercake.getMsg();
    }
```

```
    @Override
    protected int getPrice() {
        return this.battercake.getPrice();
    }
}
```

然后，创建鸡蛋装饰器 EggDecorator 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class EggDecorator extends BattercakeDecorator {
    public EggDecorator(Battercake battercake) {
        super(battercake);
    }

    protected void doSomething() {}

    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 个鸡蛋";
    }

    @Override
    protected int getPrice() {
        return super.getPrice() + 1;
    }
}
```

创建香肠装饰器 SausageDecorator 类：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class SausageDecorator extends BattercakeDecorator {
    public SausageDecorator(Battercake battercake) {
        super(battercake);
    }

    protected void doSomething() {}

    @Override
    protected String getMsg() {
        return super.getMsg() + "+1 根香肠";
    }
    @Override
    protected int getPrice() {
        return super.getPrice() + 2;
    }
}
```

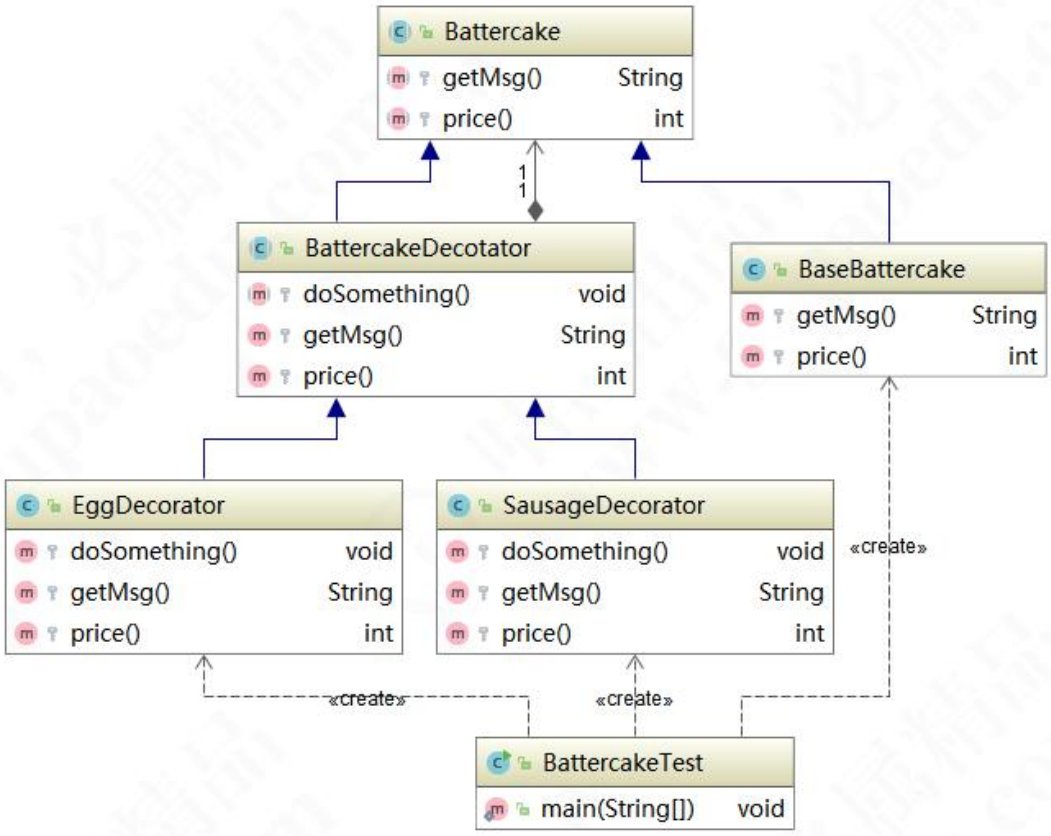
编写客户端测试代码：

```
package com.gupaoedu.vip.pattern.decorator.battercake.v2;
/**
 * Created by Tom.
 */
public class BattercakeTest {
    public static void main(String[] args) {
        Battercake battercake;
        //路边摊买一个煎饼
```

```
        battercake = new BaseBattercake();
        //煎饼有点小，想再加一个鸡蛋
        battercake = new EggDecorator(battercake);
        //再加一个鸡蛋
        battercake = new EggDecorator(battercake);
        //很饿，再加根香肠
        battercake = new SausageDecorator(battercake);

        //跟静态代理最大区别就是职责不同
        //静态代理不一定要满足 is-a 的关系
        //静态代理会做功能增强，同一个职责变得不一样

        //装饰器更多考虑是扩展
        System.out.println(battercake.getMsg() + ",总价：" + battercake.getPrice());
    }
}
```

运行结果：

来看一下类图：



为了加深印象，我们再来看一个应用场景。需求大致是这样，系统采用的是 sls 服务监控项目日志，以 Json 的格式解析，所以需要将项目中的日志封装成 json 格式再打印。现有的日志体系采用了 log4j + slf4j 框架搭建而成。客户端调用是这样的：

```
    private static final Logger logger = LoggerFactory.getLogger(Component.class);
    logger.error(string);
```

这样打印出来的是毫无规则的一行行字符串。在考虑将其转换成 json 格式时，我采用了装饰器模式。目前有的是统一接口 Logger 和其具体实现类，我要加的就是一个装饰类和真正封装成 Json 格式的装饰产品类。创建装饰器类 DecoratorLogger ：

```
public class DecoratorLogger implements Logger {

    public Logger logger;

    public DecoratorLogger(Logger logger) {

        this.logger = logger;
    }

    public void error(String str) {}

    public void error(String s, Object o) {

    }
    //省略其他默认实现
}
```

创建具体组件 JasonLogger 类实现代码如下：

```
public class JsonLogger extends DecoratorLogger {
    public JsonLogger(Logger logger) {
        super(logger);
    }

    @Override
    public void info(String msg) {

        JSONObject result = composeBasicJsonResult();
        result.put("MESSAGE", msg);
        logger.info(result.toString());
    }

    @Override
    public void error(String msg) {

        JSONObject result = composeBasicJsonResult();
        result.put("MESSAGE", msg);
        logger.error(result.toString());
    }

    public void error(Exception e) {

        JSONObject result = composeBasicJsonResult();
        result.put("EXCEPTION", e.getClass().getName());
```

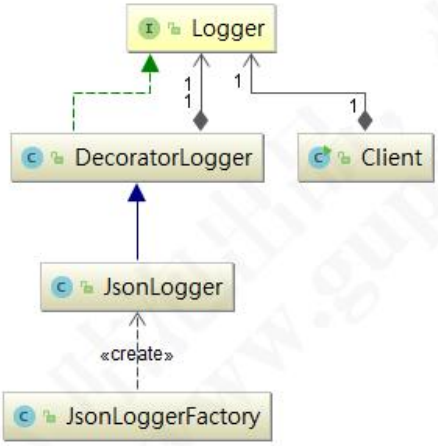
```
        String exceptionStackTrace = Arrays.toString(e.getStackTrace());
        result.put("STACKTRACE", exceptionStackTrace);
        logger.error(result.toString());
    }



    private JSONObject composeBasicJsonResult() {
        //拼装了一些运行时信息
        return new JSONObject();
    }
}
```

可以看到，在 JsonLogger 中，对于 Logger 的各种接口，我都用 JsonObject 对象进行一层封装。在打印的时候，最终还是调用原生接口 logger.error(string)，只是这个 string 参数已经被我们装饰过了。如果有额外的需求，我们也可以再写一个函数去实现。比如 error(Exception e)，只传入一个异常对象，这样在调用时就非常方便了。

另外，为了在新老交替的过程中尽量不改变太多的代码和使用方式。我又在 JsonLogger 中加入了一个内部的工厂类 JsonLoggerFactory（这个类转移到 DecoratorLogger 中可能更好一些），他包含一个静态方法，用于提供对应的 JsonLogger 实例。最终在新的日志体系中，使用方式如下：

```
    private static final Logger logger = JsonLoggerFactory.getLogger(Client.class);
    public static void main(String[] args) {

        logger.error("错误信息");
    }
```

对于客户端而言，唯一与原先不同的地方就是将 LoggerFactory 改为 JsonLoggerFactory 即可，这样的实现，也会被更快更方便的被其他开发者接受和习惯。最后看一下类图：

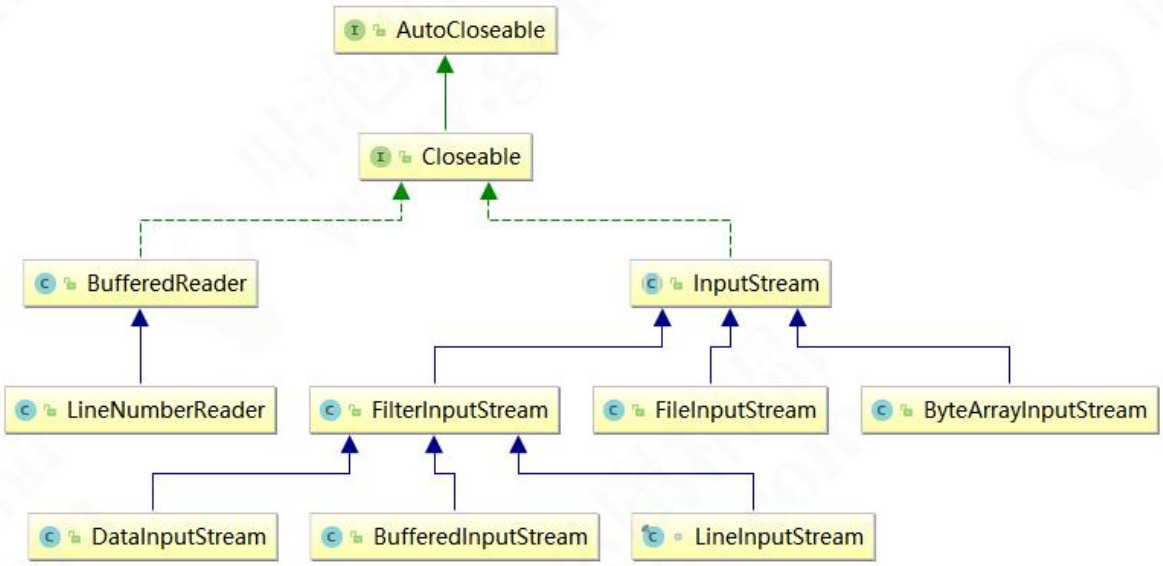


装饰器模式最本质的特征是讲原有类的附加功能抽离出来，简化原有类的逻辑。通过这样两个案例，

我们可以总结出来，其实抽象的装饰器是可有可无的，具体可以根据业务模型来选择。

装饰器模式在源码中的应用

装饰器模式在源码中也应用得非常多，在 JDK 中体现最明显的类就是 IO 相关的类，如 BufferedReader、InputStream、OutputStream，看一下常用的 InputStream 的类结构图：



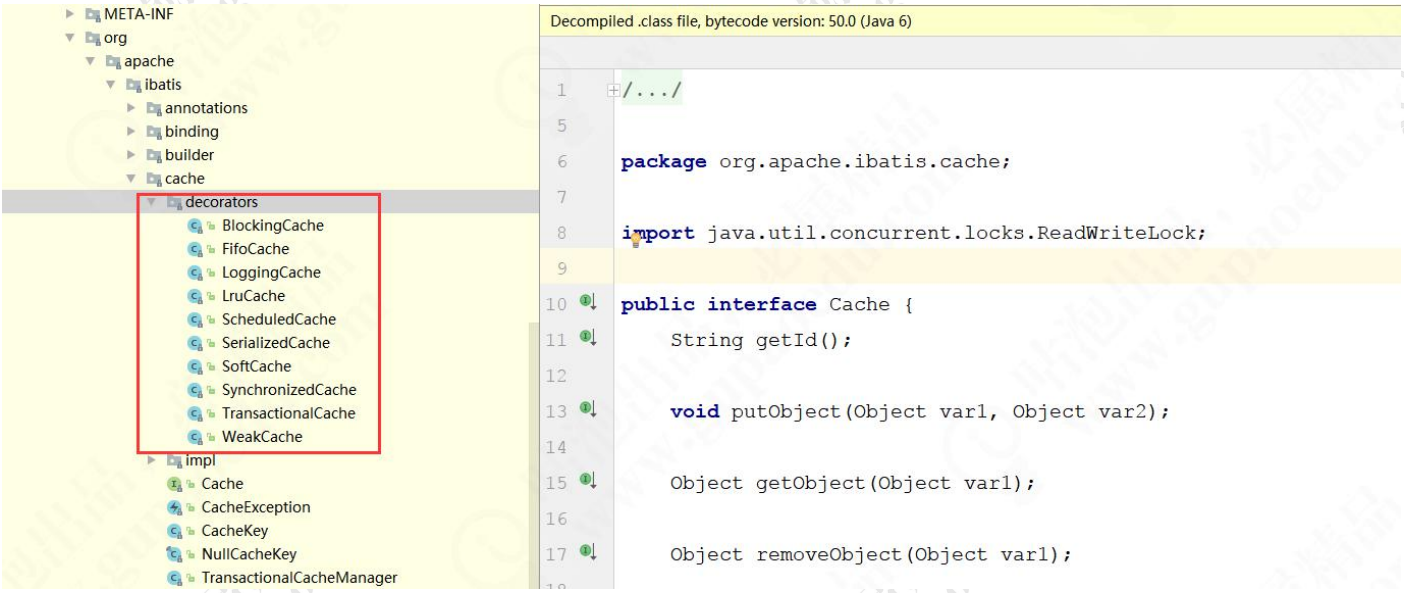
在 Spring 中的 TransactionAwareCacheDecorator 类我们也可以来尝试理解一下，这个类主要是用来处理事务缓存的，来看一下代码：

```
public class TransactionAwareCacheDecorator implements Cache {
    private final Cache targetCache;
    public TransactionAwareCacheDecorator(Cache targetCache) {
        Assert.notNull(targetCache, "Target Cache must not be null");
        this.targetCache = targetCache;
    }
    public Cache getTargetCache() {
        return this.targetCache;
    }
...
}
```

TransactionAwareCacheDecorator 就是对 Cache 的一个包装。再来看一个 MVC 中的装饰器模式 HttpHeadResponseDecorator 类：

```
public class HttpHeadResponseDecorator extends ServerHttpResponseDecorator {

   public HttpHeadResponseDecorator(ServerHttpResponse delegate) {
      super(delegate);
   }
...
}
```

最后，看看 MyBatis 中的一段处理缓存的设计 org.apache.ibatis.cache.Cache 类，找到它的包定位：



从名字上来看其实更容易理解了。比如 FifoCache 先入先出算法的缓存；LruCache 最近最少使用的缓存；TransactionlCache 事务相关的缓存，都是采用装饰器模式。MyBatis 源码在我们后续的课程也会深入讲解，感兴趣的小伙伴可以详细看看这块的源码，也可以好好学习一下 MyBatis 的命名方式，今天我们还是把重点放到设计模式上。

装饰器模式和代理模式对比

从代理模式的 UML 类图和通用代码实现上看，代理模式与装饰器模式几乎一模一样。代理模式的 Subject 对应装饰器模式的 Component，代理模式的 RealSubject 对应装饰器模式的 ConcreteComponent，代理模式的 Proxy 对应装饰器模式的 Decorator。确实，从代码实现上看，代理模式的确与装饰器模式是一样的（其实装饰器模式就是代理模式的一个特殊应用），但是这两种设计模式所面向的功能扩展面是不一样的：

装饰器模式强调自身功能的扩展。Decorator 所做的就是增强 ConcreteComponent 的功能（也有可能减弱功能），主体对象为 ConcreteComponent，着重类功能的变化；

代理模式强调对代理过程的控制。Proxy 完全掌握对 RealSubject 的访问控制，因此，Proxy 可以

决定对 RealSubject 进行功能扩展，功能缩减甚至功能散失（不调用 RealSubject 方法），主体对象为 Proxy；

简单来讲，假设现在小明想租房，那么势必会有一些事务发生：房源搜索，联系房东谈价格…

假设我们按照代理模式进行思考，那么小明只需找到一个房产中介，让他去干房源搜索，联系房东谈价格这些事情，小明只需等待通知然后付点中介费就行了；

而如果采用装饰器模式进行思考，因为装饰器模式强调的是自身功能扩展，也就是说，如果要找房子，小明自身就要增加房源搜索能力扩展，联系房东谈价格能力扩展，通过相应的装饰器，提升自身能力，一个人做满所有的事情。

装饰器模式的优缺点

优点：

1、装饰器是继承的有力补充，比继承灵活，不改变原有对象的情况下动态地给一个对象扩展功能，即插即用。

2、通过使用不同装饰类以及这些装饰类的排列组合，可以实现不同效果。

3、装饰器完全遵守开闭原则。

缺点：

1、会出现更多的代码，更多的类，增加程序复杂性。

2、动态装饰时，多层装饰时会更复杂。

那么装饰器模式我们就讲解到这里，希望小伙伴们认真体会，加深理解。