

22个场景 · Java实战代码

重学 Java 设计模式

小傅哥 · 著



单一职责 / 开闭原则 / 里氏替换 / 接口隔离 / 依赖反转

一线互联网业务开发中，交易、营销、秒杀、中间件、源码等22个真实案例场景。

前言

Hello, world of design ! 你好，设计模式的世界！

欢迎来到这里，很高兴你能拿到这本书，如果你能坚持看完并按照书中的例子进行实践，那么在编程开发的世界里，就又多了一个可以写出良好代码的人，同时也为架构师培养储备了一个人才。

可能在此之前你也多少了解过设计模式，但在实际的业务开发中使用却不多，多数时候都是大面积堆积 `if else` 组装业务流程，对于一次次的需求迭代和逻辑补充，只能东拼西凑 `Ctrl+C`、`Ctrl+V`。

所以为了能让更多的程序员  更好的接受设计思想和架构思维，并能运用到实际的业务场景。本书的作者 **小傅哥**，投入50天时间，从互联网实际业务开发中抽离出，交易、营销、秒杀、中间件、源码等22个真实场景，来学习设计模式实践使用的应用可上手技能。

谁发明了设计模式？

设计模式的概念最早是由 **克里斯托佛·亚历山大** 在其著作 **《建筑模式语言》** 中首次提出的。本书介绍了城市设计的“语言”，提供了253个描述城镇、邻里、住宅、花园、房间及西部构造的模式，而此类“语言”的基本单元就是模式。后来，**埃里希·伽玛**、**约翰·弗利赛德斯**、**拉尔夫·约翰逊** 和 **理查德·赫尔姆** 这四位作者接受了模式的概念。1994年，他们出版了 **《设计模式：可复用面向对象软件的基础》** 一书，将设计模式的概念应用到程序开发领域中。

其实有一部分人并没有仔细阅读过设计模式的相关书籍和资料，但依旧可以编写出优秀的代码。这主要是由于在经过众多项目的锤炼和对程序设计的不断追求，从而在多年编程历程上提炼出来的心得体会。而这份经验最终会与设计模式提到的内容几乎一致，同样会要求高内聚、低耦合、可扩展、可复用。你可能也遇到类似的经历，在学习一些框架的源码时，发现它里的某些设计和你在做开发时一样。

我怎么学不会设计模式？

钱也花了，书也买了。代码还是一坨一坨的！设计模式是由多年的经验提炼出来开发指导思想。就像我告诉你自行车怎么骑、汽车怎么开，但只要你没跑过几千公里，你能记住的只是理论，想上道依旧很慌！

所以，本设计模式专题系列开始，会带着你使用设计模式的思想去优化代码。从而学习设计模式的心得并融入给自己。当然这里还需要多加练习，一定是人车合一，才能站在设计模式的基础上构建出更加合理的代码。

阅读建议

本书属于实战型而不是理论介绍类书籍，每一章节都有对应的完整代码，学习的过程需要参考书中的章节与代码一起学习，同时在学习的过程中需要了解并运行代码。学习完成后进行知识点的总结，以及思考  这样的设计模式在自己的业务场景中需要如何使用。

参考资料

本书在编写的过程中参考了非常优秀的理论资料，读者在学习的过程中也可以互相参考借鉴：

1. [REFACTORING.GURU - https://refactoringguru.cn](https://refactoringguru.cn) - 这是一本图文设计模式资料，里面的图稿非常有助于理解设计模式，作者在编写此书的同时也有相应的图片引用。
2. [菜鸟设计模式 - RUNOOB.COM](http://runoob.com/design-pattern/) - 菜鸟设计模式属于比较简单的资料内容，比较适合还没有接触过设计模式研发人员。作者在编写此书的时候会进行比对，避免内容编写的不可观。

作者

作者 小傅哥，13年毕业于软件工程专业。一线互联网码农，主导过中大型项目建设、参与过大促备战、开发过中间件、写过技术专利，热衷于编码狂热于对技术的探索，内心世界丰富，时而犯二。

从19年开始萌生编写技术资料想法，以沉淀、分享、成长为核心，让自己和他人都能有所收获的想法，截止到当前已编写的内容包括：《用Java实现JVM》、《Netty4.x专题》、《中间件开发》、《领域驱动设计》、《全链路监控》、《字节码编程》等9个专题共计150篇左右原创内容。

本书《重学 Java 设计模式》于5月20日启动，到7月9日正式编写完成，整理成PDF供大家参考学习，也感谢一路来小伙伴们对我的支持。也希望读者能把这本资料分享给更多需要的人，再次感谢！

我的技术站

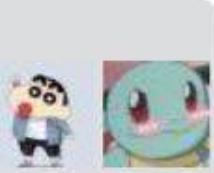
1. 公众号：bugstack虫洞栈 - 日常原创技术推文
2. 博客：<http://bugstack.cn/> - 原创技术文章汇总，适合电脑(PC)端阅读
3. Github：<https://github.com/fuzhengwei/CodeGuide/wiki> - 所有文章涉及的源码汇总以及各类资料
4. 技术圈子：微信圈子里搜索，**虫洞技术栈** - 适合日常技术发帖；资料、发问、编程、解答、插件等

与我联系

如果你在学习和成长的过程中遇到什么问题，也可以添加我的微信(fustack)进行交流，十分期待与同好交流技术，互相学习。

设计模式专栏群

本群的宗旨是给大家提供一个良好的专项的技术学习交流平台，杜绝一切广告！如果微信群人满100之后无法加入或者二维码过期，请添加作者“小傅哥”微信(fustack)，备注：设计模式加群。



《设计模式 • 小傅哥》专栏 学习群



该二维码7天内(7月18日前)有效，重新进入将更新

小傅哥微信

每个人都有自己擅长的技能或者方向，欢迎同好与我成为好友，互相补充技术技能，共同成长。
对我的文章内容或者相关案例，也可以给我提提意见。



公众号(bugstack虫洞栈)

沉淀、分享、成长，专注于原创专题案例，以最易学习编程的方式分享知识，让自己和他人都能有所收获。目前已完成的专题有：Netty4.x实战专题案例、用Java实现JVM、基于JavaAgent的全链路监控、手写RPC框架、DDD专题案例、源码分析等。



友情打赏

感谢对作者辛苦码文的支持，自愿赞赏 

源码

《重学 Java 设计模式》是以互联网真实场景实践开发为基础，每一章节的学习都会涉及到1-3个对应的案例工程，这在每一章节中都有所提到，在学习的过程中可以参考对照即可。

获取源码

1. 添加作者微信获取: `fustack`, 备注获取 `设计模式源码`
2. 微信搜索公众号: `bugstack虫洞栈`, 关注后回复 `源码下载`, 你会获得一个连接, `ID: 18` 的即使对应设计模式代码

源码截图

The screenshot shows a Java code editor in an IDE. The project navigation bar at the top indicates the current path: itstack-demo-design > itstack-demo-design-22-00. The left sidebar displays a tree view of the project structure under 'itstack-demo-design'. The main editor area shows the code for `MQAdapter.java`. The code defines a static method `filter` that takes a JSON string and a map, and returns a filtered map. It uses `JSON.parseObject` to convert the JSON string into a map, then iterates over the key set to apply a filter based on the class name. The code is annotated with several comments and annotations.

```
1 package org.itstack.demo.design;
2
3 import ...
4
5 public class MQAdapter {
6
7     @ ...
8         public static RebateInfo filter(String strJson, Map<String, Object> link) {
9             return filter(JSON.parseObject(strJson, Map.class), link);
10    }
11
12    @ ...
13        public static RebateInfo filter(Map<String, Object> obj, Map<String, Object> link) {
14            RebateInfo rebateInfo = new RebateInfo();
15            for (String key : link.keySet()) {
16                Object val = obj.get(link.get(key));
17                if (val != null) {
18                    try {
19                        rebateInfo.getClass().getMethod("set" + key, val.getClass()).invoke(rebateInfo, val);
20                    } catch (Exception e) {
21                        e.printStackTrace();
22                    }
23                }
24            }
25            return rebateInfo;
26        }
27
28    }
29}
```

目录

设计模式遵循六大原则：单一职责(一个类和方法只做一件事)、里氏替换(多态，子类可扩展父类)、依赖倒置(细节依赖抽象，下层依赖上层)、接口隔离(建立单一接口)、迪米特原则(最少知道，降低耦合)、开闭原则(抽象架构，扩展实现)，会在具体的设计模式章节中，进行体现。

1. 创建型模式

这类模式提供创建对象的机制，能够提升已有代码的灵活性和可复用性。

序号	类型	图稿	业务场景	实现要点
1	工厂方法		多种类型商品不同接口，统一发奖服务搭建场景	定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。
2	抽象工厂		替换Redis双集群升级，代理类抽象场景	提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
3	建造者		各项装修物料组合套餐选配场景	将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。
4	原型		上机考试多套试，每人题目和答案乱序排列场景	用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
5	单例		7种单例模式案例，Effective Java作者推荐枚举单例模式	保证一个类仅有一个实例，并提供一个访问它的全局访问点。

2. 结构型模式

这类模式介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。

序号	类型	图稿	业务场景	实现要点
1	适配器		从多个MQ消息体中，抽取指定字段值场景	将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
2	桥接		多支付渠道(微信、支付宝)与多支付模式(刷脸、指纹)场景	将抽象部分与实现部分分离，使它们都可以独立的变化。
3	组合		营销差异化人群发券，决策树引擎搭建场景	将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。
4	装饰		SSO单点登录功能扩展，增加拦截用户访问方法范围场景	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。
5	外观		基于SpringBoot开发门面模式中间件，统一控制接口白名单场景	为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
6	享元		基于Redis秒杀，提供活动与库存信息查询场景	运用共享技术有效地支持大量细粒度的对象。
7	代理		模拟mybatis-spring中定义DAO接口，使用代理类方式操作数据库原理实现场景	为其他对象提供一种代理以控制对这个对象的访问。

3. 行为模式

这类模式负责对象间的高效沟通和职责委派。

序号	类型	图稿	业务场景	实现要点
1	责任链		模拟618电商大促期间，项目上线流程多级负责人审批场景	避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。
	命令		模拟高档餐厅八大	将一个请求封装成一个对象，从而使您可以

2	命令		菜系，小二点单厨师烹饪场景	用不同的请求对客户进行参数化。
3	迭代器		模拟公司组织架构树结构关系，深度迭代遍历人员信息输出场景	提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。
4	中介者		按照Mybatis原理手写ORM框架, 给JDBC方式操作数据库增加中介者场景	用一个中介对象来封装一系列的对象交互, 中介者使各对象不需要显式地相互引用, 从而使其耦合松散, 而且可以独立地改变它们之间的交互。
5	备忘录		模拟互联网系统上线过程中, 配置文件回滚场景	在不破坏封装性的前提下, 捕获一个对象的内部状态, 并在该对象之外保存这个状态。
6	观察者		模拟类似小客车指标摇号过程, 监听消息通知用户中签场景	定义对象间的一种一对多的依赖关系, 当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。
7	状态		模拟系统营销活动, 状态流程审核发布上线场景	允许对象在内部状态发生改变时改变它的行为, 对象看起来好像修改了它的类。
8	策略		模拟多种营销类型优惠券, 折扣金额计算策略场景	定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。
9	模板方法		模拟爬虫各类电商商品, 生成营销推广海报场景	定义一个操作中的算法的骨架, 而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
10	访问者		模拟家长与校长, 对学生和老师的不同的视角信息的访问场景	主要将数据结构与数据操作分离。

以上图稿和部分描述参考<https://refactoringguru.cn>、<https://www.runoob.com/design-pattern/visitor-pattern.html>

创建者模式(5节)

这类模式提供创建对象的机制，能够提升已有代码的灵活性和可复用性。

创建者模式包括：工厂方法、抽象工厂、生成器、原型、单例，这5类。

第1节：工厂方法模式

好看的代码千篇一律，恶心的程序升职加薪。

该说不说几乎是程序员就都知道或者了解设计模式，但大部分小伙伴写代码总是习惯于一把梭。无论多少业务逻辑就一个类几千行，这样的开发也可以归纳为三步：定义属性、创建方法、调用展示，Done！只不过开发一时爽，重构火葬场。

好的代码不只为了完成现有功能，也会考虑后续扩展。在结构设计上松耦合易读易扩展，在领域实现上高内聚对外暴漏实现细节不被外部干扰。而这就有点像家里三居(MVC)室、四居(DDD)室的装修，你不会允许几十万的房子把走线水管裸漏在外面，也不会允许把马桶放到厨房，炉灶安装到卫生间。

谁发明了设计模式？ 设计模式的概念最早是由 [克里斯托佛·亚历山大](#) 在其著作 [《建筑模式语言》](#) 中首次提出的。本书介绍了城市设计的“语言”，提供了253个描述城镇、邻里、住宅、花园、房间及西部构造的模式，而此类“语言”的基本单元就是模式。后来，[埃里希·伽玛](#)、[约翰·弗利赛德斯](#)、[拉尔夫·约翰逊](#) 和 [理查德·赫尔姆](#) 这四位作者接受了模式的概念。1994年，他们出版了 [《设计模式：可复用面向对象软件的基础》](#) 一书，将设计模式的概念应用到程序开发领域中。

其实有一部分人并没有仔细阅读过设计模式的相关书籍和资料，但依旧可以编写出优秀的代码。这主要是由于在经过众多项目的锤炼和对程序设计的不断追求，从而在多年编程历程上提炼出来的心得体会。而这份经验最终会与设计模式提到的内容几乎一致，同样会要求高内聚、低耦合、可扩展、可复用。你可能也遇到类似的经历，在学习一些框架的源码时，发现它里的某些设计和你在做开发时一样。

我怎么学不会设计模式？ 钱也花了，书也买了。代码还是一坨一坨的！设计模式是由多年的经验提炼出来开发指导思想。就像我告诉你自行车怎么骑、汽车怎么开，但只要你没跑过几千公里，你能记住的只是理论，想上道依旧很慌！

所以，本设计模式专题系列开始，会带着你使用设计模式的思想去优化代码。从而学习设计模式的心得并融入给自己。当然这里还需要多加练习，一定是人车合一，才能站在设计模式的基础上构建出更加合理的代码。

一、开发环境

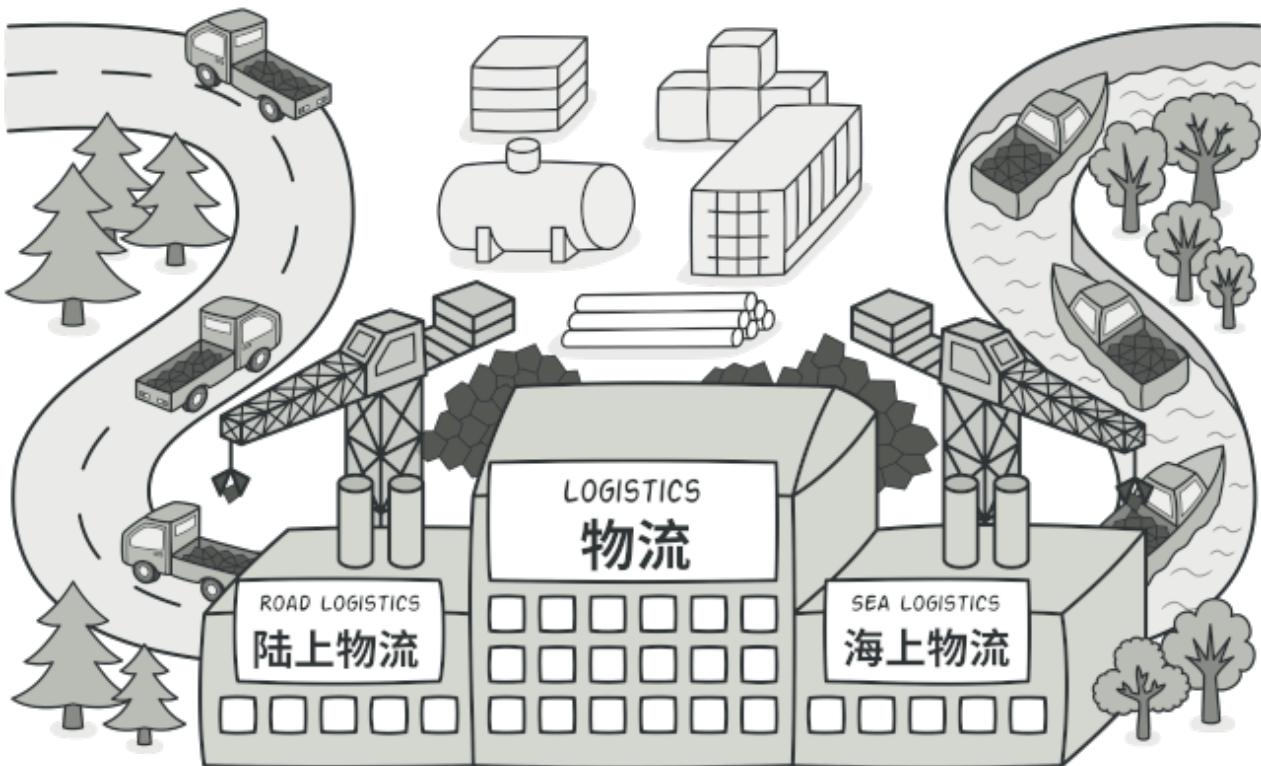
1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取。你会获得一个连接
打开后的列表中编号 [18](#)：[itstack-demo-design](#)

工程	描述
itstack-demo-design-1-00	场景模拟工程，用于提供三组不同奖品的发放接口
itstack-demo-design-1-01	使用一坨代码实现业务需求，也是对ifelse的使用
itstack-demo-design-1-02	通过设计模式优化改造代码，产生对比性从而学习

- 1-00，1 代表着第一个设计模式，工厂方法模式
- 1-00，00 代表模拟的场景

- 1-01, 01 代表第一种实现方案, 后续 02 03 以此类推

二、工厂方法模式介绍



- 工厂方法模式, 图片来自 refactoringguru.cn

工厂模式又称工厂方法模式, 是一种创建型设计模式, 其在父类中提供一个创建对象的方法, 允许子类决定实例化对象的类型。

这种设计模式也是 Java 开发中最常见的一种模式, 它的主要意图是定义一个创建对象的接口, 让其子类自己决定实例化哪一个工厂类, 工厂模式使其创建过程延迟到子类进行。

简单说就是为了提供代码结构的扩展性, 屏蔽每一个功能类中的具体实现逻辑。让外部可以更加简单的只是知道调用即可, 同时, 这也是去掉众多 `if else` 的方式。当然这可能也有一些缺点, 比如需要实现的类非常多, 如何去维护, 怎样减低开发成本。但这些问题都可以在后续的设计模式结合使用中, 逐步降低。

三、模拟发奖多种商品



为了可以让整个学习的案例更加贴近实际开发，这里模拟互联网中在营销场景下的业务。由于营销场景的复杂、多变、临时的特性，它所需要的设计需要更加深入，否则会经常面临各种紧急CRUD操作，从而让代码结构混乱不堪，难以维护。

在营销场景中经常会有某个用户做了一些操作：打卡、分享、留言、邀请注册等等，进行返利积分，最后通过积分在兑换商品，从而促活和拉新。

那么在这里我们模拟积分兑换中的发放多种类型商品，假如现在我们有如下三种类型的商品接口；

序号	类型	接口
1	优惠券	<code>CouponResult sendCoupon(String uId, String couponNumber, String uuid)</code>
2	实物商品	<code>Boolean deliverGoods(DeliverReq req)</code>
3	第三方爱奇艺 兑换卡	<code>void grantToken(String bindMobileNumber, String cardId)</code>

从以上接口来看有如下信息：

- 三个接口返回类型不同，有对象类型、布尔类型、还有一个空类型。
- 入参不同，发放优惠券需要权重、兑换卡需要卡ID、实物商品需要发货位置(对象中含有)。
- 另外可能会随着后续的业务的发展，会新增其他种商品类型。因为你所有的开发需求都是随着业务对市场的拓展而带来的。

四、用一坨坨代码实现

如果不考虑任何扩展性，只为了尽快满足需求，那么对这么几种奖励发放只需使用ifelse语句判断，调用不同的接口即可满足需求。可能这也是一些刚入门编程的小伙伴，常用的方式。接下来我们就先按照这样的方式来实现业务的需求。

1. 工程结构

```
1 itstack-demo-design-1-01
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── AwardReq.java
7     │           ├── AwardRes.java
8     │           └── PrizeController.java
9     └── test
10    └── java
11        └── org.itstack.demo.design.test
12            └── ApiTest.java
```

- 工程结构上非常简单，一个入参对象 `AwardReq`、一个出参对象 `AwardRes`，以及一个接口类 `PrizeController`

2. ifelse实现需求

```
1 public class PrizeController {
2
3     private Logger logger =
4         LoggerFactory.getLogger(PrizeController.class);
5
6     public AwardRes awardToUser(AwardReq req) {
7         String reqJson = JSON.toJSONString(req);
8         AwardRes awardRes = null;
9         try {
10             logger.info("奖品发放开始{}。req:{}",
11                         req.getId(), reqJson);
12             // 按照不同类型方法商品[1优惠券、2实物商品、3第三方兑换卡(爱奇艺)]
13             if (req.getAwardType() == 1) {
14                 CouponService couponService = new CouponService();
15                 CouponResult couponResult =
16                     couponService.sendCoupon(req.getId(),
17                     req.getAwardNumber(),
18                     req.getBizId());
19                 if ("0000".equals(couponResult.getCode())) {
20                     awardRes = new AwardRes("0000", "发放成功");
21                 } else {
22                     awardRes = new AwardRes("0001",
23                     couponResult.getInfo());
24                 }
25             } else if (req.getAwardType() == 2) {
26                 GoodsService goodsService = new GoodsService();
27                 DeliverReq deliverReq = new DeliverReq();
28                 deliverReq.setUserName(queryUserName(req.getId()));
29
30                 deliverReq.setUserPhone(queryUserPhoneNumber(req.getId()));
31             }
32         }
33     }
34 }
```

```

24         deliverReq.setSku(req.getAwardNumber());
25         deliverReq.setOrderId(req.getBizId());
26
27     deliverReq.setConsigneeUserName(req.getExtMap().get("consigneeUserName"));
28
29     deliverReq.setConsigneeUserPhone(req.getExtMap().get("consigneeUserPhone"));
30
31     deliverReq.setConsigneeUserAddress(req.getExtMap().get("consigneeUserAddress"));
32
33     Boolean isSuccess = goodsService.deliverGoods(deliverReq);
34
35     if (isSuccess) {
36         awardRes = new AwardRes("0000", "发放成功");
37     } else {
38         awardRes = new AwardRes("0001", "发放失败");
39     }
40
41     } else if (req.getAwardType() == 3) {
42         String bindMobileNumber =
43             queryUserPhoneNumber(req.getuId());
44
45         IQiYiCardService iQiYiCardService = new
46             IQiYiCardService();
47
48         iQiYiCardService.grantToken(bindMobileNumber,
49             req.getAwardNumber());
50
51         awardRes = new AwardRes("0000", "发放成功");
52     }
53
54     logger.info("奖品发放完成{}。", req.getuId());
55
56     } catch (Exception e) {
57         logger.error("奖品发放失败{}。req:{}",
58             req.getuId(), reqJson, e);
59
60         awardRes = new AwardRes("0001", e.getMessage());
61     }
62
63
64     return awardRes;
65
66 }
67
68
69     private String queryUserName(String uId) {
70         return "花花";
71     }
72
73
74     private String queryUserPhoneNumber(String uId) {
75         return "15200101232";
76     }
77
78 }

```

- 如上就是使用 `ifelse` 非常直接的实现出来业务需求的一坨代码，如果仅从业务角度看，研发如期甚至提前实现了功能。
- 那这样的代码目前来看并不会有什么问题，但如果在经过几次的迭代和拓展，接手这段代码的研发

将十分痛苦。重构成本高需要理清之前每一个接口的使用，测试回归验证时间长，需要全部验证一次。这也就是很多人并不愿意接手别人的代码，如果接手了又被压榨开发时间。那么可想而知这样的 `if else` 还会继续增加。

3. 测试验证

写一个单元测试来验证上面编写的接口方式，养成单元测试的好习惯会为你增强代码质量。

编写测试类：

```
1  @Test
2  public void test_awardToUser() {
3      PrizeController prizeController = new PrizeController();
4      System.out.println("\r\n模拟发放优惠券测试\r\n");
5      // 模拟发放优惠券测试
6      AwardReq req01 = new AwardReq();
7      req01.setUserId("10001");
8      req01.setAwardType(1);
9      req01.setAwardNumber("EGM1023938910232121323432");
10     req01.setBizId("791098764902132");
11     AwardRes awardRes01 = prizeController.awardToUser(req01);
12     logger.info("请求参数: {}", JSON.toJSONString(req01));
13     logger.info("测试结果: {}", JSON.toJSONString(awardRes01));
14     System.out.println("\r\n模拟方法实物商品\r\n");
15     // 模拟方法实物商品
16     AwardReq req02 = new AwardReq();
17     req02.setUserId("10001");
18     req02.setAwardType(2);
19     req02.setAwardNumber("9820198721311");
20     req02.setBizId("1023000020112221113");
21     Map<String, String> extMap = new HashMap<String, String>();
22     extMap.put("consigneeUserName", "谢飞机");
23     extMap.put("consigneeUserPhone", "15200292123");
24     extMap.put("consigneeUserAddress", "吉林省.长春市.双阳区.xx街道.檀溪苑小区.#18-2109");
25     req02.setExtMap(extMap);
26
27     commodityService_2.sendCommodity("10001", "9820198721311", "1023000020112221113", extMap);
28
29     AwardRes awardRes02 = prizeController.awardToUser(req02);
30     logger.info("请求参数: {}", JSON.toJSONString(req02));
31     logger.info("测试结果: {}", JSON.toJSONString(awardRes02));
32     System.out.println("\r\n第三方兑换卡(爱奇艺)\r\n");
33     AwardReq req03 = new AwardReq();
34     req03.setUserId("10001");
35     req03.setAwardType(3);
36     req03.setAwardNumber("AQY1xjkUodl8L0975GdfrYUio");
37     AwardRes awardRes03 = prizeController.awardToUser(req03);
```

```
38     logger.info("请求参数: {}", JSON.toJSONString(req03));
39     logger.info("测试结果: {}", JSON.toJSONString(awardRes03));
40 }
```

结果:

```
1 模拟发放优惠券测试
2
3 22:17:55.668 [main] INFO o.i.demo.design.PrizeController - 奖品发放开始
10001。req:
{"awardNumber": "EGM1023938910232121323432", "awardType": 1, "bizId": "791098764902132",
 "uId": "10001"}
4 模拟发放优惠券一张: 10001,EGM1023938910232121323432,791098764902132
5 22:17:55.671 [main] INFO o.i.demo.design.PrizeController - 奖品发放完成
10001。
6 22:17:55.673 [main] INFO org.itstack.demo.test.ApiTest - 请求参
数: {"uId": "10001", "bizId": "791098764902132", "awardNumber": "EGM102393891023
2121323432", "awardType": 1}
7 22:17:55.674 [main] INFO org.itstack.demo.test.ApiTest - 测试结
果: {"code": "0000", "info": "发放成功"}
8
9 模拟方法实物商品
10
11 22:17:55.675 [main] INFO o.i.demo.design.PrizeController - 奖品发放开始
10001。req:
{"awardNumber": "9820198721311", "awardType": 2, "bizId": "1023000020112221113",
 "extMap": {"consigneeUserName": "谢飞
机", "consigneeUserPhone": "15200292123", "consigneeUserAddress": "吉林省.长春
市.双阳区.xx街道.檀溪苑小区.#18-2109"}, "uId": "10001"}
12 模拟发货实物商品一个: {"consigneeUserAddress": "吉林省.长春市.双阳区.xx街道.檀溪苑小
区.#18-2109", "consigneeUserName": "谢飞
机", "consigneeUserPhone": "15200292123", "orderId": "1023000020112221113", "sk
u": "9820198721311", "userName": "花花", "userPhone": "15200101232"}
13 22:17:55.677 [main] INFO o.i.demo.design.PrizeController - 奖品发放完成
10001。
14 22:17:55.677 [main] INFO org.itstack.demo.test.ApiTest - 请求参
数: {"extMap": {"consigneeUserName": "谢飞机", "consigneeUserAddress": "吉林省.长
春市.双阳区.xx街道.檀溪苑小区.#18-
2109", "consigneeUserPhone": "15200292123"}, "uId": "10001", "bizId": "102300002
0112221113", "awardNumber": "9820198721311", "awardType": 2}
15 22:17:55.677 [main] INFO org.itstack.demo.test.ApiTest - 测试结
果: {"code": "0000", "info": "发放成功"}
16
17 第三方兑换卡(爱奇艺)
18
19 22:17:55.678 [main] INFO o.i.demo.design.PrizeController - 奖品发放开始
10001。req:
{"awardNumber": "AQY1xjkUodl8L0975GdfrYUio", "awardType": 3, "uId": "10001"}
20 模拟发放爱奇艺会员卡一张: 15200101232, AQY1xjkUodl8L0975GdfrYUio
```

```
21 22:17:55.678 [main] INFO o.i.demo.design.PrizeController - 奖品发放完成  
10001。  
22 22:17:55.678 [main] INFO org.itstack.demo.test.ApiTest - 请求参  
数: {"uId": "10001", "awardNumber": "AQY1xjkUodl8LO975GdfrYUio", "awardType": 3}  
23 22:17:55.678 [main] INFO org.itstack.demo.test.ApiTest - 测试结  
果: {"code": "0000", "info": "发放成功"}  
24  
25 Process finished with exit code 0
```

- 运行结果正常，满足当前所有业务产品需求，写的还很快。但！实在难以为维护！

五、工厂模式优化代码

接下来使用工厂方法模式来进行代码优化，也算是一次很小的重构。整理重构会你会发现代码结构清晰了、也具备了下次新增业务需求的扩展性。但在实际使用中还会对此进行完善，目前的只是抽离出最核心的部分体现到你面前，方便学习。

1. 工程结构

```
1 itstack-demo-design-1-02  
2 └── src  
3     ├── main  
4     │   └── java  
5     │       └── org.itstack.demo.design  
6     │           ├── store  
7     │           │   └── impl  
8     │           │       ├── CardCommodityService.java  
9     │           │       ├── CouponCommodityService.java  
10    │           │       └── GoodsCommodityService.java  
11    │           └── ICommodity.java  
12    └── StoreFactory.java  
13 └── test  
14     └── java  
15         └── org.itstack.demo.design.test  
16             └── ApiTest.java
```

- 首先，从上面的工程结构中你是否一些感觉，比如；它看上去清晰了、这样分层可以更好扩展了、似乎可以想象到每一个类做了什么。
- 如果还不能理解为什么这样修改，也没有关系。因为你是在通过这样的文章，来学习设计模式的魅力。并且再获取源码后，进行实际操作几次也就慢慢掌握了工厂模式的技巧。

2. 代码实现

2.1 定义发奖接口

```

1 public interface ICommodity {
2
3     void sendCommodity(String uId, String commodityId, String bizId,
4                         Map<String, String> extMap) throws Exception;
5 }

```

- 所有的奖品无论是实物、虚拟还是第三方，都需要通过我们的程序实现此接口进行处理，以保证最终入参出参的统一性。
- 接口的入参包括：用户ID、奖品ID、业务ID以及扩展字段用于处理发放实物商品时的收获地址。

2.2 实现奖品发放接口

优惠券

```

1 public class CouponCommodityService implements ICommodity {
2
3     private Logger logger =
4         LoggerFactory.getLogger(CouponCommodityService.class);
5
6     private CouponService couponService = new CouponService();
7
8     public void sendCommodity(String uId, String commodityId, String
9                               bizId, Map<String, String> extMap) throws Exception {
10        CouponResult couponResult = couponService.sendCoupon(uId,
11                                                               commodityId, bizId);
12        logger.info("请求参数[优惠券] => uId: {} commodityId: {} bizId: {}"
13                  + "extMap: {}", uId, commodityId, bizId, JSON.toJSONString(extMap));
14        logger.info("测试结果[优惠券]: {}", JSON.toJSONString(couponResult));
15        if (!"0000".equals(couponResult.getCode())) throw new
16            RuntimeException(couponResult.getInfo());
17    }
18 }

```

实物商品

```

1 public class GoodsCommodityService implements ICommodity {
2
3     private Logger logger =
4         LoggerFactory.getLogger(GoodsCommodityService.class);
5
6     private GoodsService goodsService = new GoodsService();
7
8     public void sendCommodity(String uId, String commodityId, String
9                               bizId, Map<String, String> extMap) throws Exception {
10        DeliverReq deliverReq = new DeliverReq();
11
12        // 构造商品发放请求
13        deliverReq.setCommodityId(commodityId);
14        deliverReq.setBizId(bizId);
15        deliverReq.setUser(uId);
16
17        // 将扩展字段映射到请求中
18        for (Map.Entry<String, String> entry : extMap.entrySet()) {
19            deliverReq.setExtField(entry.getKey(), entry.getValue());
20        }
21
22        // 调用GoodsService的send方法
23        goodsService.send(deliverReq);
24
25        logger.info("商品发放结果: {}", deliverReq.getResult());
26    }
27 }

```

```

9     deliverReq.setUserName(queryUserName(uId));
10    deliverReq.setUserPhone(queryUserPhoneNumber(uId));
11    deliverReq.setSku(commodityId);
12    deliverReq.setOrderId(bizId);
13    deliverReq.setConsigneeUserName(extMap.get("consigneeUserName"));
14
15    deliverReq.setConsigneeUserPhone(extMap.get("consigneeUserPhone"));
16
17    Boolean isSuccess = goodsService.deliverGoods(deliverReq);
18
19    logger.info("请求参数[优惠券] => uId: {} commodityId: {} bizId: {} extMap: {}", uId, commodityId, bizId, JSON.toJSONString(extMap));
20    logger.info("测试结果[优惠券]: {}", isSuccess);
21
22    if (!isSuccess) throw new RuntimeException("实物商品发放失败");
23 }
24
25 private String queryUserName(String uId) {
26     return "花花";
27 }
28
29 private String queryUserPhoneNumber(String uId) {
30     return "15200101232";
31 }
32
33 }

```

第三方兑换卡

```

1 public class CardCommodityService implements ICommodity {
2
3     private Logger logger =
4         LoggerFactory.getLogger(CardCommodityService.class);
5
6     // 模拟注入
7     private IQiYiCardService iQiYiCardService = new IQiYiCardService();
8
9     public void sendCommodity(String uId, String commodityId, String
10        bizId, Map<String, String> extMap) throws Exception {
11         String mobile = queryUserMobile(uId);
12         iQiYiCardService.grantToken(mobile, bizId);
13         logger.info("请求参数[爱奇艺兑换卡] => uId: {} commodityId: {} bizId: {} extMap: {}", uId, commodityId, bizId, JSON.toJSONString(extMap));
14         logger.info("测试结果[爱奇艺兑换卡]: success");
15     }
16
17     private String queryUserMobile(String uId) {

```

```
16         return "15200101232";
17     }
18
19 }
```

- 从上面可以看到每一种奖品的实现都包括在自己的类中，新增、修改或者删除都不会影响其他奖品功能的测试，降低回归测试的可能。
- 后续在新增的奖品只需要按照此结构进行填充即可，非常易于维护和扩展。
- 在统一了入参以及出参后，调用方不在需要关心奖品发放的内部逻辑，按照统一的方式即可处理。

2.3 创建商店工厂

```
1 public class StoreFactory {
2
3     public ICommodity getCommodityService(Integer commodityType) {
4         if (null == commodityType) return null;
5         if (1 == commodityType) return new CouponCommodityService();
6         if (2 == commodityType) return new GoodsCommodityService();
7         if (3 == commodityType) return new CardCommodityService();
8         throw new RuntimeException("不存在的商品服务类型");
9     }
10
11 }
```

- 这里我们定义了一个商店的工厂类，在里面按照类型实现各种商品的服务。可以非常干净整洁的处理你的代码，后续新增的商品在这里扩展即可。如果你不喜欢 `if` 判断，也可以使用 `switch` 或者 `map` 配置结构，会让代码更加干净。
- 另外很多代码检查软件和编码要求，不喜欢 `if` 语句后面不写扩展，这里是为了更加干净的向你体现逻辑。在实际的业务编码中可以添加括号。

3. 测试验证

编写测试类：

```
1 @Test
2 public void test_commodity() throws Exception {
3     StoreFactory storeFactory = new StoreFactory();
4     // 1. 优惠券
5     ICommodity commodityService_1 = storeFactory.getCommodityService(1);
6     commodityService_1.sendCommodity("10001", "EGM1023938910232121323432",
7                                     "791098764902132", null);
8     // 2. 实物商品
9     ICommodity commodityService_2 = storeFactory.getCommodityService(2);
10
11     Map<String, String> extMap = new HashMap<String, String>();
12     extMap.put("consigneeUserName", "谢飞机");
13     extMap.put("consigneeUserPhone", "15200292123");
14     extMap.put("consigneeUserAddress", "吉林省.长春市.双阳区.xx街道.檀溪苑小区.#18-2109");
```

```

14
15
    commodityService_2.sendCommodity("10001", "9820198721311", "102300002011222
1113", extMap);
16 // 3. 第三方兑换卡(爱奇艺)
17 ICommodity commodityService_3 = storeFactory.getCommodityService(3);
18
commodityService_3.sendCommodity("10001", "AQY1xjkUodl8LO975GdfrYUio", null
, null);
19 }

```

结果：

```

1 模拟发放优惠券一张: 10001,EGM1023938910232121323432,791098764902132
2 22:48:10.922 [main] INFO o.i.d.d.s.i.CouponCommodityService - 请求参数[优惠
券] => uId: 10001 commodityId: EGM1023938910232121323432 bizId:
791098764902132 extMap: null
3 22:48:10.957 [main] INFO o.i.d.d.s.i.CouponCommodityService - 测试结果[优惠
券]: {"code": "0000", "info": "发放成功"}
4 模拟发货实物商品一个: {"consigneeUserAddress": "吉林省.长春市.双阳区.xx街道.檀溪苑小
区.#18-2109", "consigneeUserName": "谢飞
机", "consigneeUserPhone": "15200292123", "orderId": "1023000020112221113", "sk
u": "9820198721311", "userName": "花花", "userPhone": "15200101232"}
5 22:48:10.962 [main] INFO o.i.d.d.s.impl.GoodsCommodityService - 请求参数[优
惠券] => uId: 10001 commodityId: 9820198721311 bizId: 1023000020112221113
extMap: {"consigneeUserName": "谢飞机", "consigneeUserAddress": "吉林省.长春市.双
阳区.xx街道.檀溪苑小区.#18-2109", "consigneeUserPhone": "15200292123"}
6 22:48:10.962 [main] INFO o.i.d.d.s.impl.GoodsCommodityService - 测试结果[优
惠券]: true
7 模拟发放爱奇艺会员卡一张: 15200101232, null
8 22:48:10.963 [main] INFO o.i.d.d.s.impl.CardCommodityService - 请求参数[爱
奇艺兑换卡] => uId: 10001 commodityId: AQY1xjkUodl8LO975GdfrYUio bizId: null
extMap: null
9 22:48:10.963 [main] INFO o.i.d.d.s.impl.CardCommodityService - 测试结果[爱
奇艺兑换卡]: success
10
11 Process finished with exit code 0

```

- 运行结果正常，既满足了业务产品需求，也满足了自己对代码的追求。这样的代码部署上线运行，内心不会恐慌，不会觉得半夜会有电话。
- 另外从运行测试结果上也可以看出来，在进行封装后可以非常清晰的看到一整套发放奖品服务的完整性，统一了入参、统一了结果。

六、总结

- 从上到下的优化来看，工厂方法模式并不复杂，甚至这样的开发结构在你有所理解后，会发现更加简单了。
- 那么这样的开发的好处知道后，也可以总结出来它的优点；避免创建者与具体的产品逻辑耦合、满

足单一职责，每一个业务逻辑实现都在所属自己的类中完成、满足开闭原则，无需更改使用调用方就可以在程序中引入新的产品类型。但这样也会带来一些问题，比如有非常多的奖品类型，那么实现的子类会极速扩张。因此也需要使用其他的模式进行优化，这些在后续的设计模式中会逐步涉及到。

- 从案例入手看设计模式往往要比看理论学的更加容易，因为案例是缩短理论到上手的最佳方式，如果你已经有所收获，一定要去尝试实操。

第 2 节：抽象工厂模式

代码一把梭，兄弟来背锅。

大部分做开发的小伙伴初心都希望把代码写好，除了把编程当作工作以外他们还是具备工匠精神的从业者。但很多时候又很难让你把初心坚持下去，就像；接了个烂手的项目、产品功能要的急、个人能力不足，等等原因导致工程代码臃肿不堪，线上频出事故，最终离职走人。

看了很多书、学了很多知识，多线程能玩出花，可最后我还是写不好代码！

这就有点像家里装修完了买物件，我几十万的实木沙发，怎么放这里就不好看。同样代码写的不好并不一定是基础技术不足，也不一定是产品要得急 怎么实现我不管明天上线。而很多时候是我们对编码的经验的不足和对架构的把控能力不到位，我相信产品的第一个需求往往都不复杂，甚至所见所得。但如果你不考虑后续的是否会拓展，将来会在哪些模块继续添加功能，那么后续的代码就会随着你种下的第一颗恶性的种子开始蔓延。

学习设计模式的心得有哪些，怎么学才会用！

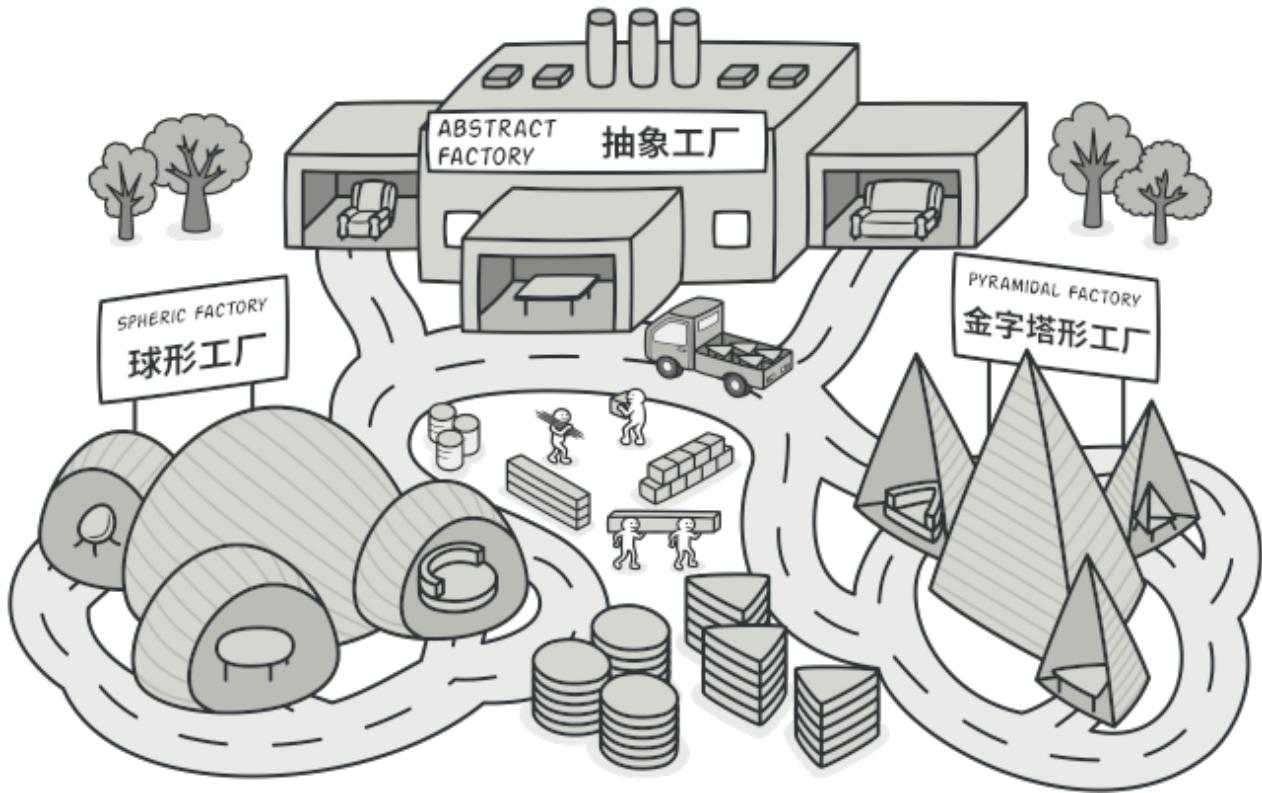
设计模式书籍，有点像考驾驶证的科一、家里装修时的手册、或者单身狗的恋爱宝典。但！你只要不实操，一定能搞的乱 码 七糟。因为这些指导思想都是从实际经验中提炼的，没有经过提炼的小白，很难驾驭这样的知识。所以在学习的过程中首先要有案例，之后再结合案例与自己实际的业务，尝试重构改造，慢慢体会其中的感受，从而也就学会了如果搭建出优秀的代码。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取

工程	描述
itstack-demo-design-2-00	场景模拟工程，模拟出使用Redis升级为集群时类改造
itstack-demo-design-2-01	使用一坨代码实现业务需求，也是对ifelse的使用
itstack-demo-design-2-02	通过设计模式优化改造代码，产生对比性从而学习

二、抽象工厂模式介绍



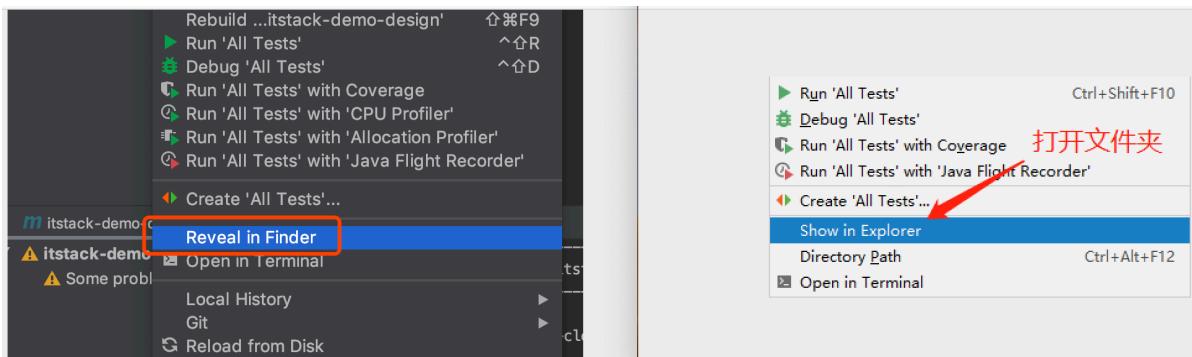
抽象工厂模式与工厂方法模式虽然主要意图都是为了解决，接口选择问题。但在实现上，抽象工厂是一个中心工厂，创建其他工厂的模式。

可能在平常的业务开发中很少关注这样的设计模式或者类似的代码结构，但是这种场景确一直在我们身边，例如：

1. 不同系统内的回车换行

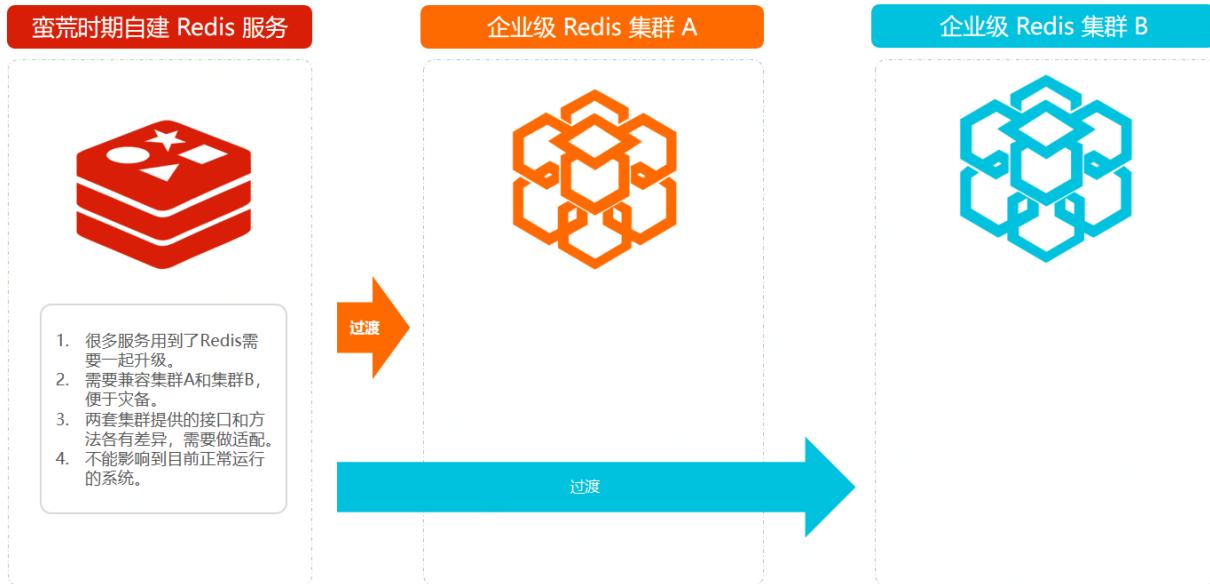
1. Unix系统里，每行结尾只有 <换行>，即 `\n`；
2. Windows系统里面，每行结尾是 <换行><回车>，即 `\n\r`；
3. Mac系统里，每行结尾是 <回车>

2. IDEA 开发工具的差异展示(Win\Mac)



除了这样显而易见的例子外，我们的业务开发中时常也会遇到类似的问题，需要兼容做处理。但大部分经验不足的开发人员，常常直接通过添加 `if else` 方式进行处理了。

三、案例场景模拟



很多时候初期业务的蛮荒发展，也会牵动着研发对系统的建设。

预估 QPS 较低、系统压力较小、并发访问不大、近一年没有大动作 等等，在考虑时间投入成本的前提下，并不会投入特别多的人力去构建非常完善的系统。就像对 Redis 的使用，往往可能只要是单机的就可以满足现状。

不吹牛的讲百度首页我上学时候一天就能写完，等毕业工作了就算给我一年都完成不了！

但随着业务超过预期的快速发展，系统的负载能力也要随着跟上。原有的单机 Redis 已经满足不了系统需求。这时候就需要更换为更为健壮的 Redis 集群服务，虽然需要修改但是不能影响目前系统的运行，还要平滑过渡过去。

随着这次的升级，可以预见的问题会有：

- 很多服务用到了Redis需要一起升级到集群。
- 需要兼容集群A和集群B，便于后续的灾备。
- 两套集群提供的接口和方法各有差异，需要做适配。
- 不能影响到目前正常运行的系统。

1. 场景模拟工程

```

1 itstack-demo-design-2-00
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 └── matter
7                     ├── EGM.java
8                     └── IIR.java
9             └── RedisUtils.java

```

工程中的所有代码可以通过关注公众号：bugstack虫洞栈，回复 源码下载 进行获取。

2. 场景简述

2.1 模拟单机服务 RedisUtils

```
public class RedisUtils {  
  
    private Logger logger = LoggerFactory.getLogger(RedisUtils.class);  
  
    private Map<String, String> dataMap = new ConcurrentHashMap<>();  
  
    public String get(String key) {  
        logger.info("Redis获取数据 key: {}", key);  
        return dataMap.get(key);  
    }  
  
    public void set(String key, String value) {  
        logger.info("Redis写入数据 key: {} val: {}", key, value);  
        dataMap.put(key, value);  
    }  
  
    public void set(String key, String value, long timeout, TimeUnit timeUnit) {  
        logger.info("Redis写入数据 key: {} val: {} timeout: {} timeUnit: {}", key, value, timeout, timeUnit.toString());  
        dataMap.put(key, value);  
    }  
  
    public void del(String key) {  
        logger.info("Redis删除数据 key: {}", key);  
        dataMap.remove(key);  
    }  
}
```

- 模拟Redis功能，也就是假定目前所有的系统都在使用的服务
- 类和方法名次都固定写死到各个业务系统中，改动略微麻烦

2.2 模拟集群 EGM

```
public class EGM {  
  
    private Logger logger = LoggerFactory.getLogger(EGM.class);  
  
    private Map<String, String> dataMap = new ConcurrentHashMap<>();  
  
    public String gain(String key) {  
        logger.info("EGM获取数据 key: {}", key);  
        return dataMap.get(key);  
    }  
  
    public void set(String key, String value) {  
        logger.info("EGM写入数据 key: {} val: {}", key, value);  
        dataMap.put(key, value);  
    }  
  
    public void setEx(String key, String value, long timeout, TimeUnit timeUnit) {  
        logger.info("EGM写入数据 key: {} val: {} timeout: {} timeUnit: {}", key, value, timeout, timeUnit.toString());  
        dataMap.put(key, value);  
    }  
  
    public void delete(String key) {  
        logger.info("EGM删除数据 key: {}", key);  
        dataMap.remove(key);  
    }  
}
```



方法名不同

- 模拟一个集群服务，但是方法名与各业务系统中使用的方法名不同。有点像你mac，我用win。做一样的事，但有不同的操作。

2.3 模拟集群 IIR

```

public class IIR {

    private Logger logger = LoggerFactory.getLogger(IIR.class);

    private Map<String, String> dataMap = new ConcurrentHashMap<>();

    public String get(String key) {
        logger.info("IIR获取数据 key: {}", key);
        return dataMap.get(key);
    }

    public void set(String key, String value) {
        logger.info("IIR写入数据 key: {} val: {}", key, value);
        dataMap.put(key, value);
    }

    public void setExpire(String key, String value, long timeout, TimeUnit timeUnit) {
        logger.info("IIR写入数据 key: {} val: {} timeout: {} timeUnit: {}", key, value, timeout, timeUnit.toString());
        dataMap.put(key, value);
    }

    public void del(String key) {
        logger.info("IIR删除数据 key: {}", key);
        dataMap.remove(key);
    }
}

```

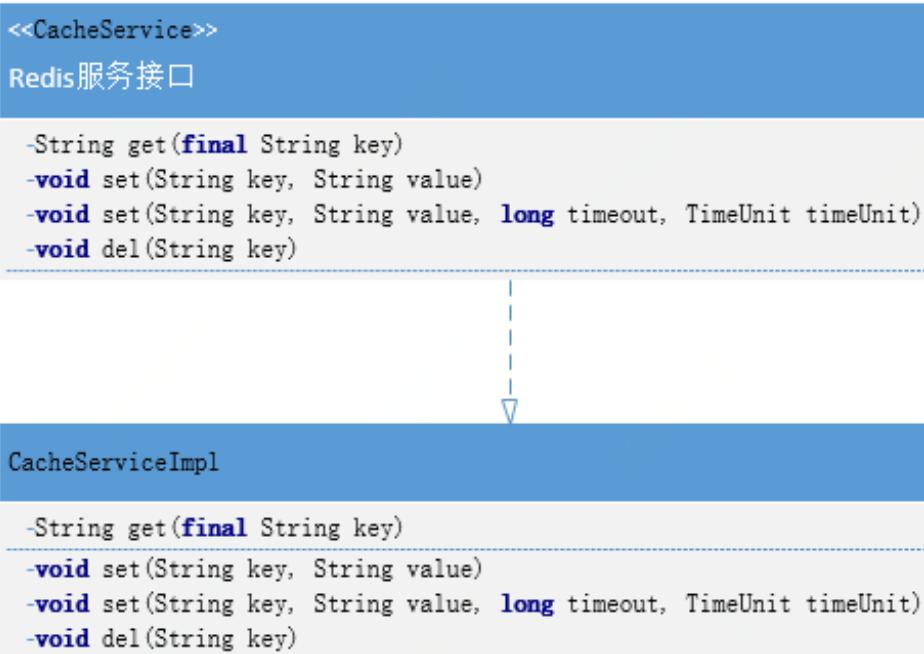


- 这是另外一套集群服务，有时候在企业开发中就很有可能出现两套服务，这里我们也是为了做模拟案例，所以添加两套实现同样功能的不同服务，来学习抽象工厂模式。

综上可以看到，我们目前的系统中已经在大量的使用redis服务，但是因为系统不能满足业务的快速发展，因此需要迁移到集群服务中。而这时有两套集群服务需要兼容使用，又要满足所有的业务系统改造的同时不影响线上使用。

3. 单集群代码使用

以下是案例模拟中原有的单集群Redis使用方式，后续会通过对这里的代码进行改造。



3.1 定义使用接口

```

1 public interface CacheService {
2
3     String get(final String key);
4
5     void set(String key, String value);
6
7     void set(String key, String value, long timeout, TimeUnit timeUnit);
8
9     void del(String key);
10
11 }

```

3.2 实现调用代码

```

1 public class CacheServiceImpl implements CacheService {
2
3     private RedisUtils redisUtils = new RedisUtils();
4
5     public String get(String key) {
6         return redisUtils.get(key);
7     }
8
9     public void set(String key, String value) {
10        redisUtils.set(key, value);
11    }
12

```

```

13     public void set(String key, String value, long timeout, TimeUnit
timeUnit) {
14         redisUtils.set(key, value, timeout, timeUnit);
15     }
16
17     public void del(String key) {
18         redisUtils.del(key);
19     }
20
21 }

```

- 目前的代码对于当前场景下的使用没有什么问题，也比较简单。但是所有的业务系统都在使用同时，需要改造就不那么容易了。这里可以思考下，看如何改造才是合理的。

四、用一坨坨代码实现

讲道理没有ifelse解决不了的逻辑，不行就在加一行！

此时的实现方式并不会修改类结构图，也就是与上面给出的类层级关系一致。通过在接口中添加类型字段区分当前使用的是哪个集群，来作为使用的判断。可以说目前的方式非常难用，其他使用方改动颇多，这里只是做为例子。

1. 工程结构

```

1 itstack-demo-design-2-01
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── impl
7                 │   └── CacheServiceImpl.java
8                 └── CacheService.java

```

- 此时的只有两个类，类结构非常简单。而我们需要的补充扩展功能也只是在 `CacheServiceImpl` 中实现。

2. ifelse实现需求

```

1 public class CacheServiceImpl implements CacheService {
2
3     private RedisUtils redisUtils = new RedisUtils();
4
5     private EGM egm = new EGM();
6
7     private IIR iir = new IIR();
8
9     public String get(String key, int redisType) {
10
11         if (1 == redisType) {

```

```

12         return egm.gain(key);
13     }
14
15     if (2 == redisType) {
16         return iir.get(key);
17     }
18
19     return redisUtils.get(key);
20 }
21
22 public void set(String key, String value, int redisType) {
23
24     if (1 == redisType) {
25         egm.set(key, value);
26         return;
27     }
28
29     if (2 == redisType) {
30         iir.set(key, value);
31         return;
32     }
33
34     redisUtils.set(key, value);
35 }
36
37 //... 同类不做太多展示，可以下载源码进行参考
38
39 }

```

- 这里的实现过程非常简单，主要根据类型判断是哪个Redis集群。
- 虽然实现是简单了，但是对使用者来说就麻烦了，并且也很难应对后期的拓展和不停的维护。

3. 测试验证

接下来我们通过junit单元测试的方式验证接口服务，强调日常编写好单测可以更好的提高系统的健壮度。

编写测试类：

```

1 @Test
2 public void test_CacheService() {
3     CacheService cacheService = new CacheServiceImpl();
4     cacheService.set("user_name_01", "小傅哥", 1);
5     String val01 = cacheService.get("user_name_01", 1);
6     System.out.println(val01);
7 }

```

结果：

```
1 22:26:24.591 [main] INFO org.itstack.demo.design.matter.EGM - EGM写入数据  
key: user_name_01 val: 小傅哥  
2 22:26:24.593 [main] INFO org.itstack.demo.design.matter.EGM - EGM获取数据  
key: user_name_01  
3 测试结果: 小傅哥  
4  
5 Process finished with exit code 0
```

- 从结果上看运行正常，并没有什么问题。但这样的代码只要到生成运行起来以后，想再改就真的难了！

五、抽象工厂模式重构代码

接下来使用抽象工厂模式来进行代码优化，也算是一次很小的重构。

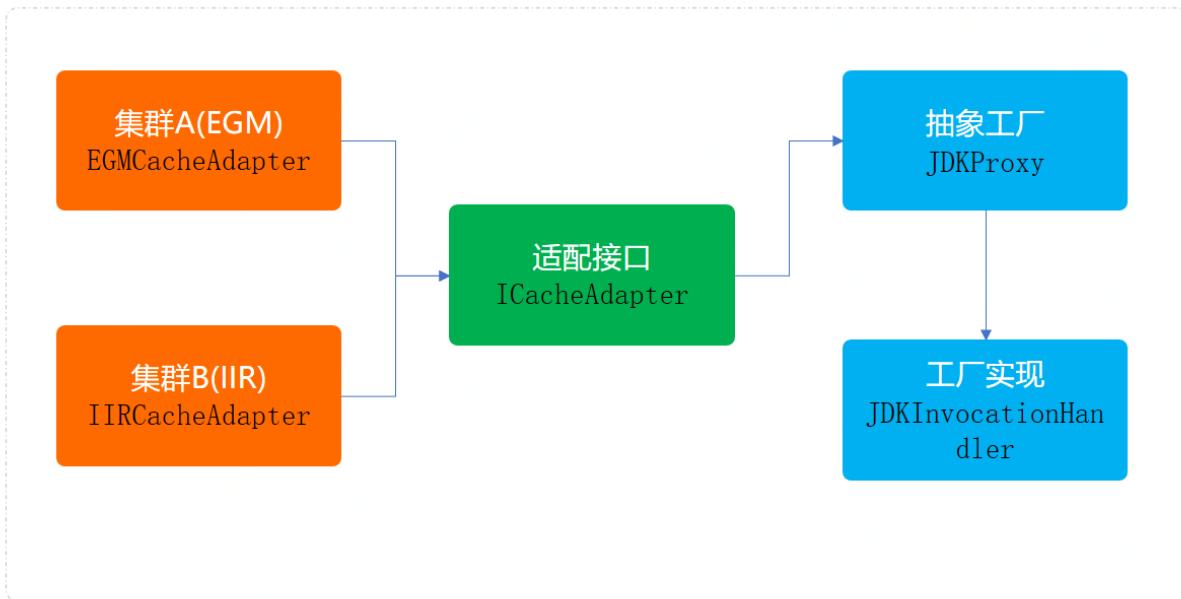
这里的抽象工厂的创建和获取方式，会采用代理类的方式进行实现。所被代理的类就是目前的Redis操作方法类，让这个类在不需要任何修改下，就可以实现调用集群A和集群B的数据服务。

并且这里还有一点非常重要，由于集群A和集群B在部分方法提供上是不同的，因此需要做一个接口适配，而这个适配类就相当于工厂中的工厂，用于创建把不同的服务抽象为统一的接口做相同的业务。这一块与我们上一章节中的 [工厂方法模型](#) 类型，可以翻阅参考。

1. 工程结构

```
1 itstack-demo-design-2-02  
2 └── src  
3     ├── main  
4     │   └── java  
5     │       └── org.itstack.demo.design  
6     │           ├── factory  
7     │           │   └── impl  
8     │           │       └── EGMCacheAdapter.java  
9     │           │       └── IIRCacheAdapter.java  
10    │           └── ICacheAdapter.java  
11    │           └── JDKInvocationHandler.java  
12    │           └── JDKProxy.java  
13    └── impl  
14        └── CacheServiceImpl.java  
15    └── CacheService.java  
16    └── test  
17        └── java  
18            └── org.itstack.demo.design.test  
19                └── ApiTest.java
```

抽象工厂模型结构



```

CacheService proxy_EGM = JDKProxy.getProxy(CacheServiceImpl.class,
                                            new EGMCacheAdapter());
  
```

- 工程中涉及的部分核心功能代码，如下；

- `ICacheAdapter`，定义了适配接口，分别包装两个集群中差异化的接口名称。`EGMCacheAdapter`、`IIRCacheAdapter`
- `JDKProxy`、`JDKInvocationHandler`，是代理类的定义和实现，这部分也就是抽象工厂的另外一种实现方式。通过这样的方式可以很好的把原有操作Redis的方法进行代理操作，通过控制不同的入参对象，控制缓存的使用。

好，那么接下来会分别讲解几个类的具体实现。

2. 代码实现

2.1 定义适配接口

```

1 public interface ICacheAdapter {
2
3     String get(String key);
4
5     void set(String key, String value);
6
7     void set(String key, String value, long timeout, TimeUnit timeUnit);
8
9     void del(String key);
10
11 }
  
```

- 这个类的主要作用是让所有集群的提供方，能在统一的方法名称下进行操作。也方便后续的拓展。

2.2 实现集群使用服务

EGMCacheAdapter

```
1 public class EGMCacheAdapter implements ICacheAdapter {  
2  
3     private EGM egm = new EGM();  
4  
5     public String get(String key) {  
6         return egm.gain(key);  
7     }  
8  
9     public void set(String key, String value) {  
10        egm.set(key, value);  
11    }  
12  
13    public void set(String key, String value, long timeout, TimeUnit  
timeUnit) {  
14        egm.setEx(key, value, timeout, timeUnit);  
15    }  
16  
17    public void del(String key) {  
18        egm.delete(key);  
19    }  
20}
```

IIRCacheAdapter

```
1 public class IIRCacheAdapter implements ICacheAdapter {  
2  
3     private IIR iir = new IIR();  
4  
5     public String get(String key) {  
6         return iir.get(key);  
7     }  
8  
9     public void set(String key, String value) {  
10        iir.set(key, value);  
11    }  
12  
13    public void set(String key, String value, long timeout, TimeUnit  
timeUnit) {  
14        iir.setExpire(key, value, timeout, timeUnit);  
15    }  
16  
17    public void del(String key) {  
18        iir.del(key);  
19    }  
20}
```

- 以上两个实现都非常容易，在统一方法名下进行包装。

2.3 定义抽象工程代理类和实现

JDKProxy

```

1 public static <T> T getProxy(Class<T> interfaceClass, ICacheAdapter
2 cacheAdapter) throws Exception {
3     InvocationHandler handler = new JDKInvocationHandler(cacheAdapter);
4     ClassLoader classLoader =
5         Thread.currentThread().getContextClassLoader();
6     Class<?>[] classes = interfaceClass.getInterfaces();
7     return (T) Proxy.newProxyInstance(classLoader, new Class[] {classes[0]},
8         handler);
9 }
```

- 这里主要的作用就是完成代理类，同时对于使用哪个集群有外部通过入参进行传递。

JDKInvocationHandler

```

1 public class JDKInvocationHandler implements InvocationHandler {
2
3     private ICacheAdapter cacheAdapter;
4
5     public JDKInvocationHandler(ICacheAdapter cacheAdapter) {
6         this.cacheAdapter = cacheAdapter;
7     }
8
9     public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
10        return ICacheAdapter.class.getMethod(method.getName(),
11            ClassLoaderUtils.getClazzByArgs(args)).invoke(cacheAdapter, args);
12    }
13}
```

- 在代理类的实现中其实也非常简单，通过穿透进来的集群服务进行方法操作。
- 另外在 `invoke` 中通过使用获取方法名称反射方式，调用对应的方法功能，也就简化了整体的使用。
- 到这我们就已经将整体的功能实现完成了，关于抽象工厂这部分也可以使用非代理的方式进行实现。

3. 测试验证

编写测试类：

```

1  @Test
2  public void test_CacheService() throws Exception {
3      CacheService proxy_EGM = JDKProxy.getProxy(CacheServiceImpl.class, new
4          EGMCacheAdapter());
5      proxy_EGM.set("user_name_01", "小傅哥");
6      String val01 = proxy_EGM.get("user_name_01");
7      System.out.println(val01);
8
9      CacheService proxy_IIR = JDKProxy.getProxy(CacheServiceImpl.class, new
10         IIRCacheAdapter());
11     proxy_IIR.set("user_name_01", "小傅哥");
12     String val02 = proxy_IIR.get("user_name_01");
13     System.out.println(val02);
14 }
```

- 在测试的代码中通过传入不同的集群类型，就可以调用不同的集群下的方法。`JDKProxy.getProxy(CacheServiceImpl.class, new EGMCacheAdapter());`
- 如果后续有扩展的需求，也可以按照这样的类型方式进行补充，同时对于改造上来说并没有改动原来的方法，降低了修改成本。

结果：

```

1  23:07:06.953 [main] INFO org.itstack.demo.design.matter.EGM - EGM写入数据
key: user_name_01 val: 小傅哥
2  23:07:06.956 [main] INFO org.itstack.demo.design.matter.EGM - EGM获取数据
key: user_name_01
3  测试结果：小傅哥
4  23:07:06.957 [main] INFO org.itstack.demo.design.matter.IIR - IIR写入数据
key: user_name_01 val: 小傅哥
5  23:07:06.957 [main] INFO org.itstack.demo.design.matter.IIR - IIR获取数据
key: user_name_01
6  测试结果：小傅哥
7
8  Process finished with exit code 0
```

- 运行结果正常，这样的代码满足了这次拓展的需求，同时你的技术能力也给老板留下了深刻的印象。
- 研发自我能力的提升远不是外接的压力就是编写一坨坨代码的接口，如果你已经熟练了很多技能，那么可以在即使紧急的情况下，也能做出完善的方案。

六、总结

- 抽象工厂模式，所要解决的问题就是在一个产品族，存在多个不同类型的产品(Redis集群、操作系统)情况下，接口选择的问题。而这种场景在业务开发中也是非常多见的，只不过可能有时候没有将它们抽象化出来。
- 你的代码只是被`if else`埋上了！当你知道什么场景下何时可以被抽象工程优化代码，那么你的代码层级结构以及满足业务需求上，都可以得到很好的完成功能实现并提升扩展性和优雅度。
- 那么这个设计模式满足了；单一职责、开闭原则、解耦等优点，但如果随着业务的不断拓展，可

能会造成类实现上的复杂度。但也可以说算不上缺点，因为可以随着其他设计方式的引入和代理类以及自动生成加载的方式降低此项缺点。

第3节：建造者模式

乱码七糟 [luàn qī bā zāo]，我时常怀疑这个成语是来形容程序猿的！

无论承接什么样的需求，是不是身边总有那么几个人代码写的烂，但是却时常有测试小姐姐过来聊天(求改bug)、有产品小伙伴送吃的(求写需求)、有业务小妹妹陪着改代码(求上线)，直至领导都认为他的工作很重要，而在旁边的你只能蹭点吃的。

那你说，CRUD的代码还想让我怎么样？

这样的小伙伴，可能把代码写的很直接，`ifelse`多用一点，满足于先临时支持一下，想着这也没什么的。而且这样的业务需求要的急又都是增删改查的内容，实在不想做设计。而如果有人提到说好好设计下，可能也会被反对不要过渡设计。

贴膏药似的修修补补，一次比一次恐怖！

第一次完成产品需求实在是很快，但互联网的代码不比传统企业。在传统行业可能一套代码能用十年，但在互联网高速的迭代下你的工程，一年就要变动几十次。如果从一开始就想只要完成功能就可以，那么随之而来的是后续的需求难以承接，每次看着成片成片的代码，实在不知如何下手。

在研发流程规范下执行，才能写出好程序！

一个项目的上线往往要经历业务需求、产品设计、研发实现、测试验证、上线部署到正式开量，而这其中对研发非常重要的一环就是研发实现的过程，又可以包括为：架构选型、功能设计、设计评审、代码实现、代码评审、单测覆盖率检查、编写文档、提交测试。所以在一些流程规范下，其实很难让你随意开发代码。

开发代码的过程不是炫技，就像盖房子如果不按照图纸来修建，回首就在山墙上搭一个厨房卫浴！可能在现实场景中这很荒唐，但在功能开发中却总有这样的代码。

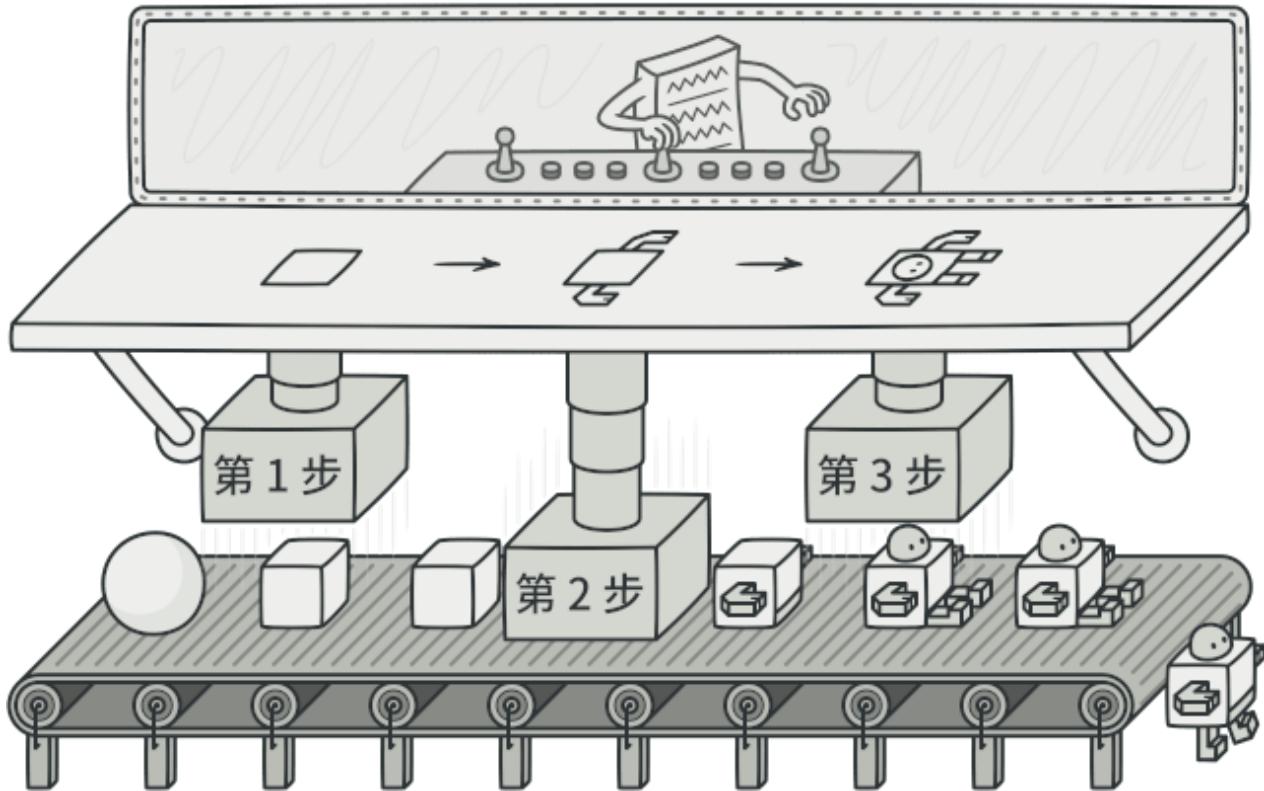
所以我们也需要一些设计模式的标准思想，去建设代码结构，提升全局把控能力。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复源码下载获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-3-00	场景模拟工程，模拟装修过程中的套餐选择(豪华、田园、简约)
itstack-demo-design-3-01	使用一坨代码实现业务需求，也是对ifelse的使用
itstack-demo-design-3-02	通过设计模式优化改造代码，产生对比性从而学习

二、建造者模式介绍



建造者模式所完成的内容就是通过将多个简单对象通过一步步的组装构建出一个复杂对象的过程。

那么，哪里有这样的场景呢？

例如你玩王者荣耀的时的初始化界面；有三条路、有树木、有野怪、有守卫塔等等，甚至依赖于你的网络情况会控制清晰度。而当你换一个场景进行其他不同模式的选择时，同样会建设道路、树木、野怪等等，但是他们的摆放和大小都有不同。这里就可以用到建造者模式来初始化游戏元素。

而这样的根据相同的 物料，不同的组装所产生出的具体的内容，就是建造者模式的最终意图，也就是；将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

三、案例场景模拟



这里我们模拟装修公司对于设计出一些套餐装修服务的场景。

很多装修公司都会给出自家的套餐服务，一般有：欧式豪华、轻奢田园、现代简约等等，而这些套餐的后面是不同的商品的组合。例如：一级&二级吊顶、多乐士涂料、圣象地板、马可波罗地砖等等，按照不同的套餐的价格选取不同的品牌组合，最终再按照装修面积给出一个整体的报价。

这里我们就模拟装修公司想推出一些套餐装修服务，按照不同的价格设定品牌选择组合，以达到使用建造者模式的过程。

1. 场景模拟工程

```
1 itstack-demo-design-3-00
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── ceiling
7                 |   ├── LevelOneCeiling.java
8                 |   └── LevelTwoCeiling.java
9                 ├── coat
10                |   ├── DuluxCoat.java
11                |   └── LiBangCoat.java
12                |   └── LevelTwoCeiling.java
13                ├── floor
14                |   ├── DerFloor.java
15                |   └── ShengXiangFloor.java
16                ├── tile
17                |   ├── DongPengTile.java
18                |   └── MarcoPoloTile.java
19                └── Matter.java
```

在模拟工程中提供了装修中所需要的物料：`ceiling(吊顶)`、`coat(涂料)`、`floor(地板)`、`tile(地砖)`，这么四项内容。（实际的装修物料要比这个多的多）

2. 场景简述

2.1 物料接口

```
1 public interface Matter {
2
3     String scene();          // 场景：地板、地砖、涂料、吊顶
4
5     String brand();          // 品牌
6
7     String model();          // 型号
8
9     BigDecimal price();      // 价格
10
11    String desc();           // 描述
12
13 }
```

- 物料接口提供了基本的信息，以保证所有的装修材料都可以按照统一标准进行获取。

2.2 吊顶(ceiling)

一级顶

```
1 public class LevelOneCeiling implements Matter {  
2  
3     public String scene() {  
4         return "吊顶";  
5     }  
6  
7     public String brand() {  
8         return "装修公司自带";  
9     }  
10  
11    public String model() {  
12        return "一级顶";  
13    }  
14  
15    public BigDecimal price() {  
16        return new BigDecimal(260);  
17    }  
18  
19    public String desc() {  
20        return "造型只做低一级，只有一个层次的吊顶，一般离顶120-150mm";  
21    }  
22  
23}
```

二级顶

```
1 public class LevelTwoCeiling implements Matter {  
2  
3     public String scene() {  
4         return "吊顶";  
5     }  
6  
7     public String brand() {  
8         return "装修公司自带";  
9     }  
10  
11    public String model() {  
12        return "二级顶";  
13    }  
14  
15    public BigDecimal price() {  
16        return new BigDecimal(850);  
17    }
```

```
18
19     public String desc() {
20         return "两个层次的吊顶，二级吊顶高度一般就往下吊20cm，要是层高很高，也可增加
21         每级的厚度";
22     }
23 }
```

2.3 涂料(coat)

多乐士

```
1 public class DuluxCoat implements Matter {
2
3     public String scene() {
4         return "涂料";
5     }
6
7     public String brand() {
8         return "多乐士(Dulux)";
9     }
10
11    public String model() {
12        return "第二代";
13    }
14
15    public BigDecimal price() {
16        return new BigDecimal(719);
17    }
18
19    public String desc() {
20        return "多乐士是阿克苏诺贝尔旗下的著名建筑装饰油漆品牌，产品畅销于全球100个国家，每年全球有5000万户家庭使用多乐士油漆。";
21    }
22
23 }
```

立邦

```
1 public class LiBangCoat implements Matter {
2
3     public String scene() {
4         return "涂料";
5     }
6
7     public String brand() {
8         return "立邦";
9     }
10
```

```
11     public String model() {
12         return "默认级别";
13     }
14
15     public BigDecimal price() {
16         return new BigDecimal(650);
17     }
18
19     public String desc() {
20         return "立邦始终以开发绿色产品、注重高科技、高品质为目标，以技术力量不断推进科
研和开发，满足消费者需求。";
21     }
22
23 }
```

2.4 地板(floor)

德尔

```
1  public class DerFloor implements Matter {
2
3      public String scene() {
4          return "地板";
5      }
6
7      public String brand() {
8          return "德尔(Der)";
9      }
10
11     public String model() {
12         return "A+";
13     }
14
15     public BigDecimal price() {
16         return new BigDecimal(119);
17     }
18
19     public String desc() {
20         return "DER德尔集团是全球领先的专业木地板制造商，北京2008年奥运会家装和公装
地板供应商";
21     }
22
23 }
```

圣象

```
1  public class ShengXiangFloor implements Matter {
2
3      public String scene() {
```

```
4         return "地板";
5     }
6
7     public String brand() {
8         return "圣象";
9     }
10
11    public String model() {
12        return "一级";
13    }
14
15    public BigDecimal price() {
16        return new BigDecimal(318);
17    }
18
19    public String desc() {
20        return "圣象地板是中国地板行业著名品牌。圣象地板拥有中国驰名商标、中国名牌、国
家免检、中国环境标志认证等多项荣誉。";
21    }
22
23 }
```

2.5 地砖(tile)

东鹏

```
1 public class DongPengTile implements Matter {
2
3     public String scene() {
4         return "地砖";
5     }
6
7     public String brand() {
8         return "东鹏瓷砖";
9     }
10
11    public String model() {
12        return "10001";
13    }
14
15    public BigDecimal price() {
16        return new BigDecimal(102);
17    }
18
19    public String desc() {
20        return "东鹏瓷砖以品质铸就品牌，科技推动品牌，口碑传播品牌为宗旨，2014年品牌
价值132.35亿元，位列建陶行业榜首。";
21    }
22 }
```

马可波罗

```

1  public class MarcoPoloTile implements Matter {
2
3      public String scene() {
4          return "地砖";
5      }
6
7      public String brand() {
8          return "马可波罗(MARCO POLO)";
9      }
10
11     public String model() {
12         return "缺省";
13     }
14
15     public BigDecimal price() {
16         return new BigDecimal(140);
17     }
18
19     public String desc() {
20         return "“马可波罗”品牌诞生于1996年，作为国内最早品牌化的建陶品牌，以“文化陶
21 瓷”占领市场，享有“仿古砖至尊”的美誉。";
22     }
23 }

```

- 以上就是本次装修公司所提供的装修配置单，接下我们会通过案例去使用不同的物料组合出不同的套餐服务。

四、用一坨坨代码实现

讲道理没有if else解决不了的逻辑，不行就在加一行！

每一个章节中我们都会使用这样很直白的方式去把功能实现出来，在通过设计模式去优化完善。这样的代码结构也都是非常简单的，没有复杂的类关系结构，都是直来直去的代码。除了我们经常强调的这样的代码不能很好的扩展外，做一些例子demo工程还是可以的。

1. 工程结构

```

1  itstack-demo-design-3-01
2  └─ src
3    └─ main
4      └─ java
5        └─ org.itstack.demo.design
6          └─ DecorationPackageController.java

```

一个类几千行的代码你是否见过，嚯？那今天就让你见识一下有这样潜质的类！

2. ifelse实现需求

```
1 public class DecorationPackageController {  
2  
3     public String getMatterList(BigDecimal area, Integer level) {  
4  
5         List<Matter> list = new ArrayList<Matter>(); // 装修清单  
6         BigDecimal price = BigDecimal.ZERO; // 装修价格  
7  
8         // 豪华欧式  
9         if (1 == level) {  
10             LevelTwoCeiling levelTwoCeiling = new LevelTwoCeiling(); // 吊  
顶, 二级顶  
11             DuluxCoat duluxCoat = new DuluxCoat(); // 涂  
料, 多乐士  
12             ShengXiangFloor shengXiangFloor = new ShengXiangFloor(); // 地  
板, 圣象  
13  
14             list.add(levelTwoCeiling);  
15             list.add(duluxCoat);  
16             list.add(shengXiangFloor);  
17  
18             price = price.add(area.multiply(new  
19             BigDecimal("0.2").multiply(levelTwoCeiling.price())));  
20             price = price.add(area.multiply(new  
21             BigDecimal("1.4").multiply(duluxCoat.price())));  
22             price = price.add(area.multiply(shengXiangFloor.price()));  
23  
24         }  
25  
26         // 轻奢田园  
27         if (2 == level) {  
28             LevelTwoCeiling levelTwoCeiling = new LevelTwoCeiling(); // 吊  
顶, 二级顶  
29             LiBangCoat liBangCoat = new LiBangCoat(); // 涂  
料, 立邦  
30             MarcoPoloTile marcoPoloTile = new MarcoPoloTile(); // 地  
砖, 马可波罗  
31  
32             list.add(levelTwoCeiling);  
33             list.add(liBangCoat);  
34             list.add(marcoPoloTile);  
35  
36             price = price.add(area.multiply(new  
37             BigDecimal("0.2").multiply(levelTwoCeiling.price())));
```

```
37         price = price.add(area.multiply(new
38             BigDecimal("1.4")).multiply(liBangCoat.price())));
39         price = price.add(area.multiply(marcoPoloTile.price())));
40     }
41
42     // 现代简约
43     if (3 == level) {
44
45         LevelOneCeiling levelOneCeiling = new LevelOneCeiling(); // 吊顶, 二级顶
46         LiBangCoat liBangCoat = new LiBangCoat(); // 涂料, 立邦
47         DongPengTile dongPengTile = new DongPengTile(); // 地砖, 东鹏
48
49         list.add(levelOneCeiling);
50         list.add(liBangCoat);
51         list.add(dongPengTile);
52
53         price = price.add(area.multiply(new
54             BigDecimal("0.2")).multiply(levelOneCeiling.price())));
55         price = price.add(area.multiply(new
56             BigDecimal("1.4")).multiply(liBangCoat.price()));
57         price = price.add(area.multiply(dongPengTile.price()));
58     }
59
60     StringBuilder detail = new StringBuilder("\r\n-----\r\n-----\r\n" +
61         "装修清单" + "\r\n" +
62         "套餐等级: " + level + "\r\n" +
63         "套餐价格: " + price.setScale(2, BigDecimal.ROUND_HALF_UP) +
64         " 元\r\n" +
65         "房屋面积: " + area.doubleValue() + " 平米\r\n" +
66         "材料清单: \r\n");
67
68     for (Matter matter: list) {
69
70         detail.append(matter.scene()).append(": ").append(matter.brand()).append(
71             "、").append(matter.model()).append("、平米价
格: ").append(matter.price()).append(" 元。 \r\n");
72     }
73
74
75     return detail.toString();
76
77 }
78 }
```

- 首先这段代码所要解决的问题就是接收入参；装修面积(area)、装修等级(level)，根据不同类型的装修等级选择不同的材料。
- 其次在实现过程中可以看到每一段 if 块里，都包含着不通的材料(吊顶，二级顶、涂料，立邦、地砖，马可波罗)，最终生成装修清单和装修成本。
- 最后提供获取装修详细信息的方法，返回给调用方，用于知道装修清单。

3. 测试验证

接下来我们通过junit单元测试的方式验证接口服务，强调日常编写好单测可以更好的提高系统的健壮度。

编写测试类：

```

1  @Test
2  public void test_DecorationPackageController(){
3      DecorationPackageController decoration = new
4      DecorationPackageController();
5      // 豪华欧式
6      System.out.println(decoration.getMatterList(new
7          BigDecimal("132.52"),1));
8      // 轻奢田园
9      System.out.println(decoration.getMatterList(new
10     BigDecimal("98.25"),2));
11     // 现代简约
12     System.out.println(decoration.getMatterList(new
13     BigDecimal("85.43"),3));
14 }
```

结果：

```

1 -----
2 装修清单
3 套餐等级: 1
4 套餐价格: 198064.39 元
5 房屋面积: 132.52 平米
6 材料清单:
7 吊顶: 装修公司自带、二级顶、平米价格: 850 元。
8 涂料: 多乐士(Dulux)、第二代、平米价格: 719 元。
9 地板: 圣象、一级、平米价格: 318 元。
10
11
12 -----
13 装修清单
14 套餐等级: 2
15 套餐价格: 119865.00 元
16 房屋面积: 98.25 平米
17 材料清单:
18 吊顶: 装修公司自带、二级顶、平米价格: 850 元。
19 涂料: 立邦、默认级别、平米价格: 650 元。
```

```
20 地砖：马可波罗 (MARCO POLO)、缺省、平米价格：140 元。
21
22
23 -----
24 装修清单
25 套餐等级：3
26 套餐价格：90897.52 元
27 房屋面积：85.43 平米
28 材料清单：
29 吊顶：装修公司自带、一级顶、平米价格：260 元。
30 涂料：立邦、默认级别、平米价格：650 元。
31 地砖：东鹏瓷砖、10001、平米价格：102 元。
32
33
34 Process finished with exit code 0
```

- 看到输出的这个结果，已经很有装修公司提供报价单的感觉了。以上这段使用 `ifelse` 方式实现的代码，目前已经满足的我们的也许功能。但随着老板对业务的快速发展要求，会提供很多的套餐针对不同的户型。那么这段实现代码将迅速扩增到几千行，甚至在修改改中，已经像膏药一样难以维护。

五、建造者模式重构代码

接下来使用建造者模式来进行代码优化，也算是一次很小的重构。

建造者模式主要解决的问题是在软件系统中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的过程构成；由于需求的变化，这个复杂对象的各个部分经常面临着重大的变化，但是将它们组合在一起的过程却相对稳定。

这里我们会把构建的过程交给 `创建者` 类，而创建者通过使用我们的 `构建工具包`，去构建出不同的 `装修套餐`。

1. 工程结构

```
1 itstack-demo-design-3-02
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── Builder.java
7     │           ├── DecorationPackageMenu.java
8     │           └── IMenu.java
9     └── test
10    └── java
11        └── org.itstack.demo.design.test
12            └── ApiTest.java
```

建造者模型结构



工程中有三个核心类和一个测试类，核心类是建造者模式的具体实现。与 `if else` 实现方式相比，多出来了两个二外的类。具体功能如下；

- `Builder`，建造者类具体的各种组装由此类实现。
- `DecorationPackageMenu`，是 `IMenu` 接口的实现类，主要是承载建造过程中的填充器。相当于这是一套承载物料和创建者中间衔接的内容。

好，那么接下来会分别讲解几个类的具体实现。

2. 代码实现

2.1 定义装修包接口

```

1 public interface IMenu {
2
3     IMenu appendCeiling(Matter matter); // 吊顶
4
5     IMenu appendCoat(Matter matter); // 涂料
6
7     IMenu appendFloor(Matter matter); // 地板
8
9     IMenu appendTile(Matter matter); // 地砖
10
11    String getDetail(); // 明细
12
13 }
```

- 接口类中定义了填充各项物料的方法；吊顶、涂料、地板、地砖，以及最终提供获取全部明细的方法。

2.2 装修包实现

```

1 public class DecorationPackageMenu implements IMenu {
2
3     private List<Matter> list = new ArrayList<Matter>(); // 装修清单
4     private BigDecimal price = BigDecimal.ZERO; // 装修价格
5 }
```

```
6     private BigDecimal area; // 面积
7     private String grade; // 装修等级：豪华欧式、轻奢田园、现代简约
8
9     private DecorationPackageMenu() {
10    }
11
12    public DecorationPackageMenu(Double area, String grade) {
13        this.area = new BigDecimal(area);
14        this.grade = grade;
15    }
16
17    public IMenu appendCeiling(Matter matter) {
18        list.add(matter);
19        price = price.add(area.multiply(new
20            BigDecimal("0.2")).multiply(matter.price())));
21        return this;
22    }
23
24    public IMenu appendCoat(Matter matter) {
25        list.add(matter);
26        price = price.add(area.multiply(new
27            BigDecimal("1.4")).multiply(matter.price())));
28        return this;
29    }
30
31    public IMenu appendFloor(Matter matter) {
32        list.add(matter);
33        price = price.add(area.multiply(matter.price()));
34        return this;
35    }
36
37    public IMenu appendTile(Matter matter) {
38        list.add(matter);
39        price = price.add(area.multiply(matter.price()));
40        return this;
41    }
42
43    public String getDetail() {
44        StringBuilder detail = new StringBuilder("\r\n-----\r\n-----\r\n" +
45                "装修清单" + "\r\n" +
46                "套餐等级：" + grade + "\r\n" +
47                "套餐价格：" + price.setScale(2, BigDecimal.ROUND_HALF_UP) +
48                " 元\r\n" +
49                "房屋面积：" + area.doubleValue() + " 平米\r\n" +
50                "材料清单：\r\n");
51
52    }
53
54    for (Matter matter: list) {
```

```

51     detail.append(matter.scene()).append(": ").append(matter.brand()).append(
52         "、").append(matter.model()).append("、平米价
53         格: ").append(matter.price()).append(" 元。 \n");
54     }
55 }
56
57 }
```

- 装修包的实现中每一个方法都会了 `this`，也就可以非常方便的用于连续填充各项物料。
- 同时在填充时也会根据物料计算平米数下的报价，吊顶和涂料按照平米数适量乘以常熟计算。
- 最后同样提供了统一的获取装修清单的明细方法。

2.3 建造者方法

```

1 public class Builder {
2
3     public IMenu levelOne(Double area) {
4         return new DecorationPackageMenu(area, "豪华欧式")
5             .appendCeiling(new LevelTwoCeiling())           // 吊顶, 二级顶
6             .appendCoat(new DuluxCoat())                  // 涂料, 多乐士
7             .appendFloor(new ShengXiangFloor());          // 地板, 圣象
8     }
9
10    public IMenu levelTwo(Double area){
11        return new DecorationPackageMenu(area, "轻奢田园")
12            .appendCeiling(new LevelTwoCeiling())           // 吊顶, 二级顶
13            .appendCoat(new LiBangCoat())                  // 涂料, 立邦
14            .appendTile(new MarcoPoloTile());            // 地砖, 马可波罗
15    }
16
17    public IMenu levelThree(Double area){
18        return new DecorationPackageMenu(area, "现代简约")
19            .appendCeiling(new LevelOneCeiling())          // 吊顶, 二级顶
20            .appendCoat(new LiBangCoat())                  // 涂料, 立邦
21            .appendTile(new DongPengTile());              // 地砖, 东鹏
22    }
23
24 }
```

- 建造者的使用中就已经非常容易了，统一的建造方式，通过不同物料填充出不同的装修风格；`豪华欧式`、`轻奢田园`、`现代简约`，如果将来业务扩展也可以将这部分内容配置到数据库自动生成。但整体的思想还可以使用创建者模式进行搭建。

3. 测试验证

编写测试类：

```
1  @Test
2  public void test_Builder(){
3      Builder builder = new Builder();
4      // 豪华欧式
5      System.out.println(builder.levelOne(132.52D).getDetail());
6      // 轻奢田园
7      System.out.println(builder.levelTwo(98.25D).getDetail());
8      // 现代简约
9      System.out.println(builder.levelThree(85.43D).getDetail());
10 }
```

结果：

```
1 -----
2 装修清单
3 套餐等级：豪华欧式
4 套餐价格：198064.39 元
5 房屋面积：132.52 平米
6 材料清单：
7 吊顶：装修公司自带、二级顶、平米价格：850 元。
8 涂料：多乐士(Dulux)、第二代、平米价格：719 元。
9 地板：圣象、一级、平米价格：318 元。
10
11 -----
12 装修清单
13 套餐等级：轻奢田园
14 套餐价格：119865.00 元
15 房屋面积：98.25 平米
16 材料清单：
17 吊顶：装修公司自带、二级顶、平米价格：850 元。
18 涂料：立邦、默认级别、平米价格：650 元。
19 地砖：马可波罗(MARCO POLO)、缺省、平米价格：140 元。
20
21
22 -----
23 装修清单
24 套餐等级：现代简约
25 套餐价格：90897.52 元
26 房屋面积：85.43 平米
27 材料清单：
28 吊顶：装修公司自带、一级顶、平米价格：260 元。
29 涂料：立邦、默认级别、平米价格：650 元。
30 地砖：东鹏瓷砖、10001、平米价格：102 元。
31
32
33
34 Process finished with exit code 0
```

- 测试结果是一样的，调用方式也基本类似。但是目前的代码结构却可以让你很方便的很有调理的进行扩展业务开发。而不是以往一样把所有代码都写到 `if else` 里面。

六、总结

- 通过上面对建造者模式的使用，已经可以摸索出一点心得。那就是什么时候会选择这样的设计模式，当：一些基本物料不会变，而其组合经常变化的时候，就可以选择这样的设计模式来构建代码。
- 此设计模式满足了单一职责原则以及可复用的技术、建造者独立、易扩展、便于控制细节风险。但同时当出现特别多的物料以及很多的组合后，类的不断扩展也会造成难以维护的问题。但这种设计结构模型可以把重复的内容抽象到数据库中，按照需要配置。这样就可以减少代码中大量的重复。
- 设计模式能带给你的是一些思想，但在平时的开发中怎么样清晰的提炼出符合此思路的建造模块，是比较难的。需要经过一些锻炼和不断承接更多的项目，从而获得这部分经验。有的时候你的代码写的好，往往是倒逼的，复杂的业务频繁的变化，不断的挑战！

第4节：原型模式

老板你加钱我的代码能飞

程序员这份工作里有两种人；一类是热爱喜欢的、一类是仅当成工作的。而喜欢代码编程的这部分人会极其主动学习去丰富自己的羽翼，也非常喜欢对技术探索力求将学到的知识赋能到平时的业务需求开发中。对于这部分小伙伴来说上班写代码还能赚钱真的是幸福！

怎么成为喜欢编码都那部分人

无论做哪行那业你都喜欢，往往来自从中持续不断都获取成就感。就开发编程而言因为你的一行代码影响到了千千万万的人、因为你的一行代码整个系统更加稳定、因为你的一行代码扛过了所有秒杀等等，这样一行行的代码都是你日积月累学习到的经验。那如果你也想成为这样有成就感的程序员就需要不断的学习，不断的用更多的技能知识把自己编写的代码运用到更核心的系统。

方向不对努力白费

平常你也付出了很多的时间，但就是没有得到多少收益。就像有时候很多小伙伴问我，我是该怎么学一个我没接触过的内容。我的个人经验非常建议，先不要学太多理论性的内容，而是尝试实际操作下，把要学的内容做一些Demo案例出来。这有点像你买了个自行车是先拆了学学怎么个原理，还是先骑几圈呢？哪怕摔了跟头，但那都是必须经历后留下的经验。

同样我也知道很多人看了设计模式收获不大，这主要新人对没有案例或者案例不贴近实际场景没有学习方向导致。太空、太虚、太玄，让人没有抓手！

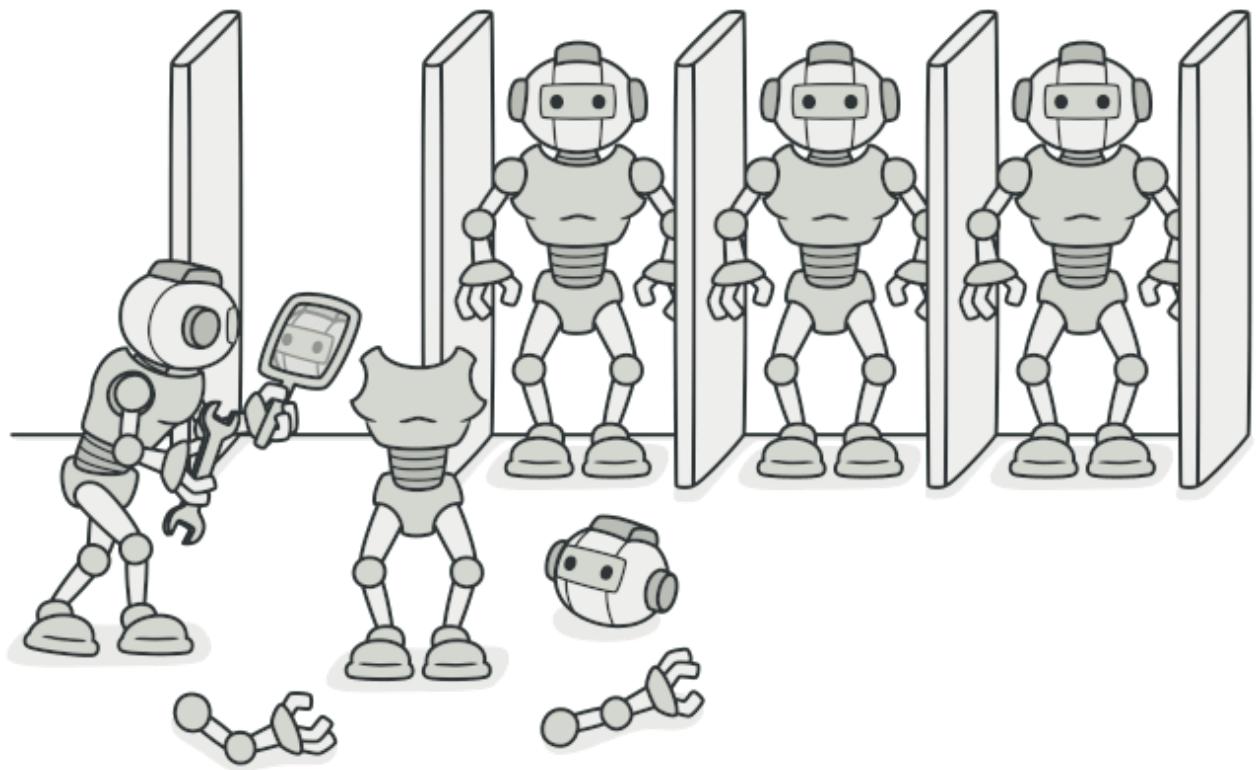
所以我开始编写以实际案例为着手的方式，讲解设计模式的文章，帮助大家成长的同时也让我自己有所沉淀！

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-4-00	场景模拟工程，模拟在线考试题库抽提打乱顺序
itstack-demo-design-4-01	使用一坨代码实现业务需求，也是对ifelse的使用
itstack-demo-design-4-02	通过设计模式优化改造代码，产生对比性从而学习

二、原型模式介绍



原型模式主要解决的问题就是创建重复对象，而这部分对象内容本身比较复杂，生成过程可能从库或者RPC接口中获取数据的耗时较长，因此采用克隆的方式节省时间。

其实这种场景经常出现在我们的身边，只不过很少用到自己的开发中，就像：

1. 你经常 `ctrl+C`、`Ctrl+V`，复制粘贴代码。
2. Java多数类中提供的API方法：`Object clone()`。
3. 细胞的有丝分裂。

类似以上的场景并不少，但如果让你去思考平时的代码开发中，有用到这样的设计模式吗？确实不容易找到，甚至有时候是忽略了这个设计模式的方式。在没有阅读下文之前，也可以思考下哪些场景可以应用到。

三、案例场景模拟



每个人都经历过考试，从纸制版到上机答题，大大小小也有几百场。而以前坐在教室里答题身边的人都是一套试卷，考试的时候还能偷摸或者别人给发信息抄一抄答案。

但从一部分可以上机考试的内容开始，在保证大家的公平性一样的题目下，开始出现试题混排更有做的好的答案选项也混排。这样大大的增加了抄的成本，也更好的做到了考试的公平性。

但如果这个公平性的考试需求交给你来完成，你会怎么做？

因为需要实现一个上机考试抽题的服务，因此在这里建造一个题库题目的场景类信息，用于创建：`选择题`、`问答题`。

1. 场景模拟工程

```
1 | itstack-demo-design-4-00
2 | └─ src
3 |   └─ main
4 |     └─ java
5 |       └─ org.itstack.demo.design
6 |         └─ ChoiceQuestion.java
7 |         └─ AnswerQuestion.java
```

- 在这里模拟了两个试卷题目的类：`ChoiceQuestion(选择题)`、`AnswerQuestion(问答题)`。如果是实际的业务场景开发中，会有更多的题目类型，可以回忆一下你的高考试卷。

2. 场景简述

2.1 选择题

```
1 public class ChoiceQuestion {  
2  
3     private String name; // 题目  
4     private Map<String, String> option; // 选项; A、B、C、D  
5     private String key; // 答案; B  
6  
7     public ChoiceQuestion() {  
8     }  
9  
10    public ChoiceQuestion(String name, Map<String, String> option, String  
key) {  
11        this.name = name;  
12        this.option = option;  
13        this.key = key;  
14    }  
15  
16    // ...get/set  
17 }
```

2.2 问答题

```
1 public class AnswerQuestion {  
2  
3     private String name; // 问题  
4     private String key; // 答案  
5  
6     public AnswerQuestion() {  
7     }  
8  
9     public AnswerQuestion(String name, String key) {  
10        this.name = name;  
11        this.key = key;  
12    }  
13  
14    // ...get/set  
15 }
```

- 以上两个类就是我们场景中需要的物料内容，相对来说比较简单。如果你在测试的时候想扩充学习，可以继续添加一些其他物料(题目类型)。

四、用一坨坨代码实现

今天的实现方式没有if else了，但是没有一个类解决不了的业务，只要你胆大！

在以下的例子中我们会按照每一个用户创建试卷的题目，并返回给调用方。

1. 工程结构

```
1 itstack-demo-design-4-01
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 └── QuestionBankController.java
```

- 一个类几千行的代码你是否见过，嚯？那今天就再让你见识一下有这样潜质的类！

2. 一把梭实现需求

```
1 public class QuestionBankController {
2
3     public String createPaper(String candidate, String number) {
4
5         List<ChoiceQuestion> choiceQuestionList = new
6             ArrayList<ChoiceQuestion>();
7         List<AnswerQuestion> answerQuestionList = new
8             ArrayList<AnswerQuestion>();
9
10        Map<String, String> map01 = new HashMap<String, String>();
11        map01.put("A", "JAVA2 EE");
12        map01.put("B", "JAVA2 Card");
13        map01.put("C", "JAVA2 ME");
14        map01.put("D", "JAVA2 HE");
15        map01.put("E", "JAVA2 SE");
16
17        Map<String, String> map02 = new HashMap<String, String>();
18        map02.put("A", "JAVA程序的main方法必须写在类里面");
19        map02.put("B", "JAVA程序中可以有多个main方法");
20        map02.put("C", "JAVA程序中类名必须与文件名一样");
21        map02.put("D", "JAVA程序的main方法中如果只有一条语句, 可以不用{}(大括号)括
起来");
22
23        Map<String, String> map03 = new HashMap<String, String>();
24        map03.put("A", "变量由字母、下划线、数字、$符号随意组成; ");
25        map03.put("B", "变量不能以数字作为开头; ");
26        map03.put("C", "A和a在java中是同一个变量; ");
27        map03.put("D", "不同类型的变量, 可以起相同的名字; ");
28
29        Map<String, String> map04 = new HashMap<String, String>();
30        map04.put("A", "STRING");
31        map04.put("B", "x3x;");
32        map04.put("C", "void");
33        map04.put("D", "de$f");
34
35        Map<String, String> map05 = new HashMap<String, String>();
36        map05.put("A", "31");
```

```
35     map05.put("B", "0");
36     map05.put("C", "1");
37     map05.put("D", "2");
38
39     choiceQuestionList.add(new ChoiceQuestion("JAVA所定义的版本中不包括",
map01, "D"));
40     choiceQuestionList.add(new ChoiceQuestion("下列说法正确的是", map02,
"A"));
41     choiceQuestionList.add(new ChoiceQuestion("变量命名规范说法正确的是",
map03, "B"));
42     choiceQuestionList.add(new ChoiceQuestion("以下()不是合法的标识符",
map04, "C"));
43     choiceQuestionList.add(new ChoiceQuestion("表达式(11+3*8)/4%3的值
是", map05, "D"));
44     answerQuestionList.add(new AnswerQuestion("小红马和小黑马生的小马几条
腿", "4条腿"));
45     answerQuestionList.add(new AnswerQuestion("铁棒打头疼还是木棒打头疼",
"头最疼"));
46     answerQuestionList.add(new AnswerQuestion("什么床不能睡觉", "牙床"));
47     answerQuestionList.add(new AnswerQuestion("为什么好马不吃回头草", "后
面的草没了"));
48
49     // 输出结果
50     StringBuilder detail = new StringBuilder("考生: " + candidate +
"\r\n" +
51             "考号: " + number + "\r\n" +
52             "-----\r\n" +
53             "一、选择题" + "\r\n\r\n");
54
55     for (int idx = 0; idx < choiceQuestionList.size(); idx++) {
56         detail.append("第").append(idx + 1).append("题: ").append(choiceQuestionList.get(idx).getName()).append("\r\n");
57         Map<String, String> option =
choiceQuestionList.get(idx).getOption();
58         for (String key : option.keySet()) {
59
60             detail.append(key).append(": ").append(option.get(key)).append("\r\n");
61             ;
62         }
63         detail.append("答
案: ").append(choiceQuestionList.get(idx).getKey()).append("\r\n\r\n");
64     }
65
66     detail.append("二、问答题" + "\r\n\r\n");
67
68     for (int idx = 0; idx < answerQuestionList.size(); idx++) {
```

```

68         detail.append("第").append(idx +
69             "题: ").append(answerQuestionList.get(idx).getName()).append("\r\n");
70             detail.append("答
71             案: ").append(answerQuestionList.get(idx).getKey()).append("\r\n\r\n");
72     }
73 }
74
75 }

```

- 这样的代码往往都非常易于理解，要什么程序就给什么代码，不面向对象，只面向过程。不考虑扩展性，能用就行。
- 以上的代码主要就三部分内容；首先创建选择题和问答题到集合中、定义详情字符串包装结果、返回结果内容。
- 但以上的代码有一个没有实现的地方就是不能乱序，所有人的试卷顺序都是一样的。如果需要加乱序也是可以的，但复杂度又会增加。这里不展示具体过多实现，只为后文对比重构。

3. 测试验证

接下来我们通过junit单元测试的方式验证接口服务，强调日常编写好单测可以更好的提高系统的健壮度。

编写测试类：

```

1 @Test
2 public void test_QuestionBankController() {
3     QuestionBankController questionBankController = new
4     QuestionBankController();
5     System.out.println(questionBankController.createPaper("花花",
6 "1000001921032"));
7     System.out.println(questionBankController.createPaper("豆豆",
8 "1000001921051"));
9     System.out.println(questionBankController.createPaper("大宝",
10 "1000001921987"));
11 }

```

结果：

```

1 考生：花花
2 考号：1000001921032
3 -----
4 一、选择题
5
6 第1题：JAVA所定义的版本中不包括
7 A: JAVA2 EE
8 B: JAVA2 Card
9 C: JAVA2 ME

```

10 D: JAVA2 HE
11 E: JAVA2 SE
12 答案: D
13
14 第2题: 下列说法正确的是
15 A: JAVA程序的main方法必须写在类里面
16 B: JAVA程序中可以有多个main方法
17 C: JAVA程序中类名必须与文件名一样
18 D: JAVA程序的main方法中如果只有一条语句, 可以不用{}(大括号)括起来
19 答案: A
20
21 第3题: 变量命名规范说法正确的是
22 A: 变量由字母、下划线、数字、\$符号随意组成;
23 B: 变量不能以数字作为开头;
24 C: A和a在java中是同一个变量;
25 D: 不同类型的变量, 可以起相同的名字;
26 答案: B
27
28 第4题: 以下()不是合法的标识符
29 A: STRING
30 B: x3x;
31 C: void
32 D: de\$f
33 答案: C
34
35 第5题: 表达式 $(11+3*8)/4 \% 3$ 的值是
36 A: 31
37 B: 0
38 C: 1
39 D: 2
40 答案: D
41
42 二、问答题
43
44 第1题: 小红马和小黑马生的小马几条腿
45 答案: 4条腿
46
47 第2题: 铁棒打头疼还是木棒打头疼
48 答案: 头最疼
49
50 第3题: 什么床不能睡觉
51 答案: 牙床
52
53 第4题: 为什么好马不吃回头草
54 答案: 后面的草没了
55
56
57 考生: 豆豆
58 考号: 1000001921051

59 -----
60 一、选择题
61
62 第1题：JAVA所定义的版本中不包括
63 A: JAVA2 EE
64 B: JAVA2 Card
65 C: JAVA2 ME
66 D: JAVA2 HE
67 E: JAVA2 SE
68 答案：D
69
70 第2题：下列说法正确的是
71 A: JAVA程序的main方法必须写在类里面
72 B: JAVA程序中可以有多个main方法
73 C: JAVA程序中类名必须与文件名一样
74 D: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来
75 答案：A
76
77 第3题：变量命名规范说法正确的是
78 A: 变量由字母、下划线、数字、\$符号随意组成；
79 B: 变量不能以数字作为开头；
80 C: A和a在java中是同一个变量；
81 D: 不同类型的变量，可以起相同的名字；
82 答案：B
83
84 第4题：以下()不是合法的标识符
85 A: STRING
86 B: x3x;
87 C: void
88 D: de\$f
89 答案：C
90
91 第5题：表达式($11+3*8)/4 \% 3$ 的值是
92 A: 31
93 B: 0
94 C: 1
95 D: 2
96 答案：D
97
98 二、问答题
99
100 第1题：小红马和小黑马生的小马几条腿
101 答案：4条腿
102
103 第2题：铁棒打头疼还是木棒打头疼
104 答案：头最疼
105
106 第3题：什么床不能睡觉
107 答案：牙床

108
109 第4题：为什么好马不吃回头草
110 答案：后面的草没了
111
112
113 考生：大宝
114 考号：1000001921987

115
116 一、选择题
117
118 第1题：JAVA所定义的版本中不包括
119 A: JAVA2 EE
120 B: JAVA2 Card
121 C: JAVA2 ME
122 D: JAVA2 HE
123 E: JAVA2 SE
124 答案：D
125
126 第2题：下列说法正确的是
127 A: JAVA程序的main方法必须写在类里面
128 B: JAVA程序中可以有多个main方法
129 C: JAVA程序中类名必须与文件名一样
130 D: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来
131 答案：A
132
133 第3题：变量命名规范说法正确的是
134 A: 变量由字母、下划线、数字、\$符号随意组成；
135 B: 变量不能以数字作为开头；
136 C: A和a在java中是同一个变量；
137 D: 不同类型的变量，可以起相同的名字；
138 答案：B
139
140 第4题：以下()不是合法的标识符
141 A: STRING
142 B: x3x;
143 C: void
144 D: de\$f
145 答案：C
146
147 第5题：表达式 $(11+3*8)/4 \% 3$ 的值是
148 A: 31
149 B: 0
150 C: 1
151 D: 2
152 答案：D
153
154 二、问答题
155
156 第1题：小红马和小黑马生的小马几条腿

```
157 答案：4条腿
158
159 第2题：铁棒打头疼还是木棒打头疼
160 答案：头最疼
161
162 第3题：什么床不能睡觉
163 答案：牙床
164
165 第4题：为什么好马不吃回头草
166 答案：后面的草没了
167
168 Process finished with exit code 0
```

- 以上呢就是三位考试的试卷；`花花`、`豆豆`、`大宝`，每个人的试卷内容是一样的这没问题，但是三个人的题目以及选项顺序都是一样，就没有达到我们说希望的乱序要求。
- 而且以上这样的代码非常难扩展，随着题目的不断的增加以及乱序功能的补充，都会让这段代码变得越来越混乱。

五、原型模式重构代码

接下来使用原型模式来进行代码优化，也算是一次很小的重构。

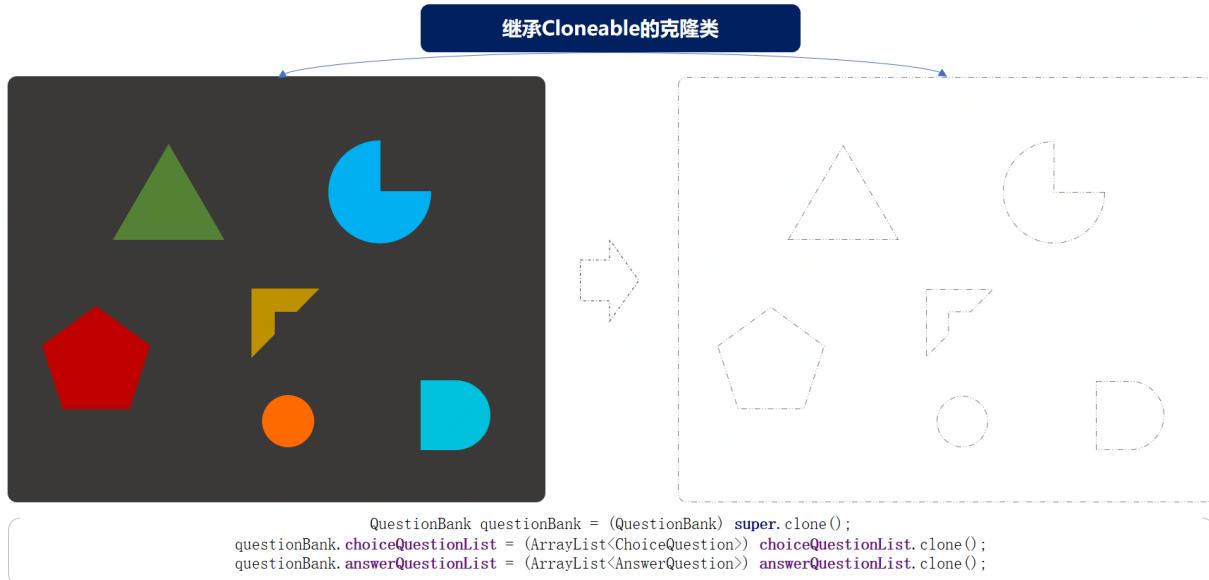
原型模式主要解决的问题就是创建大量重复的类，而我们模拟的场景就需要给不同的用户都创建相同的试卷，但这些试卷的题目不便于每次都从库中获取，甚至有时候需要从远程的RPC中获取。这样都是非常耗时的，而且随着创建对象的增多将严重影响效率。

在原型模式中所需要的非常重要的手段就是克隆，在需要用到克隆的类中都需要实现 `implements Cloneable` 接口。

1. 工程结构

```
1 itstack-demo-design-4-02
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── util
7     │           │   └── Topic.java
8     │           │   └── TopicRandomUtil.java
9     │           └── QuestionBank.java
10    │           └── QuestionBankController.java
11    └── test
12        └── java
13            └── org.itstack.demo.design.test
14                └── ApiTest.java
```

原型模式模型结构



- 工程中包括了核心的题库类 `QuestionBank`，题库中主要负责将各个的题目进行组装最终输出试卷。
- 针对每一个试卷都会使用克隆的方式进行复制，复制完成后将试卷中题目以及每个题目的答案进行乱序处理。这里提供了工具包；`TopicRandomUtil`

2. 代码实现

2.1 题目选项乱序操作工具包

```

1  /**
2   * 乱序Map元素，记录对应答案key
3   * @param option 题目
4   * @param key     答案
5   * @return Topic 乱序后 {A=c., B=d., C=a., D=b.}
6   */
7  static public Topic random(Map<String, String> option, String key) {
8      Set<String> keySet = option.keySet();
9      ArrayList<String> keyList = new ArrayList<String>(keySet);
10     Collections.shuffle(keyList);
11     HashMap<String, String> optionNew = new HashMap<String, String>();
12     int idx = 0;
13     String keyNew = "";
14     for (String next : keySet) {
15         String randomKey = keyList.get(idx++);
16         if (key.equals(next)) {
17             keyNew = randomKey;
18         }
19         optionNew.put(randomKey, option.get(next));
20     }
21     return new Topic(optionNew, keyNew);
22 }

```

- 可能你还记得上文里我们提供了Map存储题目选项，同时key的属性存放答案。如果忘记可以往上

翻翻

- 这个这个工具类的操作就是将原有Map中的选型乱序操作，也就是A的选项内容给B，B的可能给C，同时记录正确答案在处理后的位置信息。

2.2 克隆对象处理类

```
1 public class QuestionBank implements Cloneable {
2
3     private String candidate; // 考生
4     private String number;    // 考号
5
6     private ArrayList<ChoiceQuestion> choiceQuestionList = new
7     ArrayList<ChoiceQuestion>();
8     private ArrayList<AnswerQuestion> answerQuestionList = new
9     ArrayList<AnswerQuestion>();
10
11    public QuestionBank append(ChoiceQuestion choiceQuestion) {
12        choiceQuestionList.add(choiceQuestion);
13        return this;
14    }
15
16    public QuestionBank append(AnswerQuestion answerQuestion) {
17        answerQuestionList.add(answerQuestion);
18        return this;
19    }
20
21    @Override
22    public Object clone() throws CloneNotSupportedException {
23        QuestionBank questionBank = (QuestionBank) super.clone();
24        questionBank.choiceQuestionList = (ArrayList<ChoiceQuestion>)
25        choiceQuestionList.clone();
26        questionBank.answerQuestionList = (ArrayList<AnswerQuestion>)
27        answerQuestionList.clone();
28
29        // 题目乱序
30        Collections.shuffle(questionBank.choiceQuestionList);
31        Collections.shuffle(questionBank.answerQuestionList);
32        // 答案乱序
33        ArrayList<ChoiceQuestion> choiceQuestionList =
34        questionBank.choiceQuestionList;
35        for (ChoiceQuestion question : choiceQuestionList) {
36            Topic random = TopicRandomUtil.random(question.getOption(),
37            question.getKey());
38            question.setOption(random.getOption());
39            question.setKey(random.getKey());
40        }
41        return questionBank;
42    }
43}
```

```

38     public void setCandidate(String candidate) {
39         this.candidate = candidate;
40     }
41
42     public void setNumber(String number) {
43         this.number = number;
44     }
45
46     @Override
47     public String toString() {
48
49         StringBuilder detail = new StringBuilder("考生: " + candidate +
50             "\r\n" +
51             "考号: " + number + "\r\n" +
52             "-----\r\n" +
53             "一、选择题" + "\r\n\r\n");
54
55         for (int idx = 0; idx < choiceQuestionList.size(); idx++) {
56             detail.append("第").append(idx +
57                 1).append("题: ").append(choiceQuestionList.get(idx).getName()).append("\r\n");
58             Map<String, String> option =
59             choiceQuestionList.get(idx).getOption();
60             for (String key : option.keySet()) {
61
62                 detail.append(key).append(": ").append(option.get(key)).append("\r\n");
63             }
64             detail.append("答
案: ").append(choiceQuestionList.get(idx).getKey()).append("\r\n\r\n");
65         }
66
67         detail.append("二、问答题" + "\r\n\r\n");
68
69         for (int idx = 0; idx < answerQuestionList.size(); idx++) {
70             detail.append("第").append(idx +
71                 1).append("题: ").append(answerQuestionList.get(idx).getName()).append("\r\n");
72             detail.append("答
案: ").append(answerQuestionList.get(idx).getKey()).append("\r\n\r\n");
73         }
74     }
75
76     return detail.toString();
77 }
78 }
```

这里的主要操作内容有三个，分别是：

- 两个 `append()`，对各项题目的添加，有点像我们在建造者模式中使用的方式，添加装修物料。

- `clone()`，这里的核心操作就是对对象的复制，这里的复制不只是包括了本身，同时对两个集合也做了复制。只有这样的拷贝才能确保在操作克隆对象的时候不影响原对象。
- 乱序操作，在`list`集合中有一个方法，`Collections.shuffle`，可以将原有集合的顺序打乱，输出一个新的顺序。在这里我们使用此方法对题目进行乱序操作。

2.4 初始化试卷数据

```

1  public class QuestionBankController {
2
3      private QuestionBank questionBank = new QuestionBank();
4
5      public QuestionBankController() {
6
7          Map<String, String> map01 = new HashMap<String, String>();
8          map01.put("A", "JAVA2 EE");
9          map01.put("B", "JAVA2 Card");
10         map01.put("C", "JAVA2 ME");
11         map01.put("D", "JAVA2 HE");
12         map01.put("E", "JAVA2 SE");
13
14         Map<String, String> map02 = new HashMap<String, String>();
15         map02.put("A", "JAVA程序的main方法必须写在类里面");
16         map02.put("B", "JAVA程序中可以有多个main方法");
17         map02.put("C", "JAVA程序中类名必须与文件名一样");
18         map02.put("D", "JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来");
19
20         Map<String, String> map03 = new HashMap<String, String>();
21         map03.put("A", "变量由字母、下划线、数字、$符号随意组成；");
22         map03.put("B", "变量不能以数字作为开头；");
23         map03.put("C", "A和a在java中是同一个变量；");
24         map03.put("D", "不同类型的变量，可以起相同的名字；");
25
26         Map<String, String> map04 = new HashMap<String, String>();
27         map04.put("A", "STRING");
28         map04.put("B", "x3x;");
29         map04.put("C", "void");
30         map04.put("D", "de$f");
31
32         Map<String, String> map05 = new HashMap<String, String>();
33         map05.put("A", "31");
34         map05.put("B", "0");
35         map05.put("C", "1");
36         map05.put("D", "2");
37
38         questionBank.append(new ChoiceQuestion("JAVA所定义的版本中不包括",
39             map01, "D"))
40                 .append(new ChoiceQuestion("下列说法正确的是", map02, "A"))

```

```

40             .append(new ChoiceQuestion("变量命名规范说法正确的是", map03,
41                         "B"))
42             .append(new ChoiceQuestion("以下()不是合法的标识符", map04,
43                         "C"))
44             .append(new ChoiceQuestion("表达式(11+3*8)/4%3的值是",
45                         map05, "D"))
46             .append(new AnswerQuestion("小红马和小黑马生的小马几条腿", "4条
47                         腿"))
48             .append(new AnswerQuestion("铁棒打头疼还是木棒打头疼", "头最
49                         疼"))
50             .append(new AnswerQuestion("什么床不能睡觉", "牙床"))
51             .append(new AnswerQuestion("为什么好马不吃回头草", "后面的草没
52                         了"));
53         }
54     }
55 }
56 }
```

- 这个类的内容就比较简单了，主要提供对试卷内容的模式初始化操作(所有考生试卷一样，题目顺序不一致)。
- 以及对外部提供创建试卷的方法，在创建的过程中使用的是克隆的方式；`(QuestionBank) questionBank.clone();`，并最终返回试卷信息。

3. 测试验证

编写测试类：

```

1  @Test
2  public void test_QuestionBank() throws CloneNotSupportedException {
3      QuestionBankController questionBankController = new
4      QuestionBankController();
5      System.out.println(questionBankController.createPaper("花花",
6          "1000001921032"));
7      System.out.println(questionBankController.createPaper("豆豆",
8          "1000001921051"));
9      System.out.println(questionBankController.createPaper("大宝",
10         "1000001921987"));
11 }
```

结果：

1 考生：花花
2 考号：1000001921032

3
4 一、选择题
5
6 第1题：JAVA所定义的版本中不包括
7 A: JAVA2 Card
8 B: JAVA2 HE
9 C: JAVA2 EE
10 D: JAVA2 ME
11 E: JAVA2 SE
12 答案：B
13
14 第2题：表达式 $(11+3*8)/4 \% 3$ 的值是
15 A: 1
16 B: 0
17 C: 31
18 D: 2
19 答案：D
20
21 第3题：以下()不是合法的标识符
22 A: void
23 B: de\$f
24 C: STRING
25 D: x3x;
26 答案：A
27
28 第4题：下列说法正确的是
29 A: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来
30 B: JAVA程序中可以有多个main方法
31 C: JAVA程序的main方法必须写在类里面
32 D: JAVA程序中类名必须与文件名一样
33 答案：C
34
35 第5题：变量命名规范说法正确的是
36 A: 变量由字母、下划线、数字、\$符号随意组成；
37 B: A和a在java中是同一个变量；
38 C: 不同类型的变量，可以起相同的名字；
39 D: 变量不能以数字作为开头；
40 答案：D
41
42 二、问答题
43
44 第1题：小红马和小黑马生的小马几条腿
45 答案：4条腿
46
47 第2题：什么床不能睡觉
48 答案：牙床
49

50 第3题：铁棒打头疼还是木棒打头疼
51 答案：头最疼
52
53 第4题：为什么好马不吃回头草
54 答案：后面的草没了
55
56
57 考生：豆豆
58 考号：1000001921051

59
60 一、选择题
61
62 第1题：下列说法正确的是
63 A: JAVA程序中可以有多个main方法
64 B: JAVA程序的main方法必须写在类里面
65 C: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来
66 D: JAVA程序中类名必须与文件名一样
67 答案：B
68
69 第2题：表达式 $(11+3*8)/4 \% 3$ 的值是
70 A: 2
71 B: 1
72 C: 31
73 D: 0
74 答案：A
75
76 第3题：以下()不是合法的标识符
77 A: void
78 B: de\$f
79 C: x3x;
80 D: STRING
81 答案：A
82
83 第4题：JAVA所定义的版本中不包括
84 A: JAVA2 Card
85 B: JAVA2 HE
86 C: JAVA2 ME
87 D: JAVA2 EE
88 E: JAVA2 SE
89 答案：B
90
91 第5题：变量命名规范说法正确的是
92 A: 变量不能以数字作为开头；
93 B: A和a在java中是同一个变量；
94 C: 不同类型的变量，可以起相同的名字；
95 D: 变量由字母、下划线、数字、\$符号随意组成；
96 答案：A
97
98 二、问答题

99
100 第1题：什么床不能睡觉
101 答案：牙床
102
103 第2题：铁棒打头疼还是木棒打头疼
104 答案：头最疼
105
106 第3题：为什么好马不吃回头草
107 答案：后面的草没了
108
109 第4题：小红马和小黑马生的小马几条腿
110 答案：4条腿
111
112
113 考生：大宝
114 考号：1000001921987
115 -----
116 一、选择题
117
118 第1题：以下()不是合法的标识符
119 A: x3x;
120 B: de\$f
121 C: void
122 D: STRING
123 答案：C
124
125 第2题：表达式 $(11+3*8)/4 \% 3$ 的值是
126 A: 31
127 B: 0
128 C: 2
129 D: 1
130 答案：C
131
132 第3题：变量命名规范说法正确的是
133 A: 不同类型的变量，可以起相同的名字；
134 B: 变量由字母、下划线、数字、\$符号随意组成；
135 C: 变量不能以数字作为开头；
136 D: A和a在java中是同一个变量；
137 答案：C
138
139 第4题：下列说法正确的是
140 A: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来
141 B: JAVA程序的main方法必须写在类里面
142 C: JAVA程序中类名必须与文件名一样
143 D: JAVA程序中可以有多个main方法
144 答案：B
145
146 第5题：JAVA所定义的版本中不包括
147 A: JAVA2 EE

```

148 B: JAVA2 Card
149 C: JAVA2 HE
150 D: JAVA2 SE
151 E: JAVA2 ME
152 答案: C
153
154 二、问答题
155
156 第1题: 为什么好马不吃回头草
157 答案: 后面的草没了
158
159 第2题: 小红马和小黑马生的小马几条腿
160 答案: 4条腿
161
162 第3题: 什么床不能睡觉
163 答案: 牙床
164
165 第4题: 铁棒打头疼还是木棒打头疼
166 答案: 头最疼
167
168 Process finished with exit code 0

```

从以上的输出结果可以看到，每个人的题目和答案都是差异化的乱序的，如下图比对结果； - 花花、豆豆、大宝，每个人的试卷都存在着题目和选项的混乱排序

<p>考生: 花花 考号: 1000001921032</p> <p>一、选择题</p> <p>第1题: JAVA所定义的版本中不包括 A: JAVA2 Card B: JAVA2 HE C: JAVA2 BE D: JAVA2 ME E: JAVA2 SE 答案: B</p> <p>第2题: 表达式$(11+3*8)/4%3$的值是 A: 1 B: 0 C: 31 D: 2 答案: D</p> <p>第3题: 以下()不是合法的标识符 A: void B: de\$f C: STRING D: x3s 答案: A</p> <p>第4题: 下列说法正确的是 A: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来 B: JAVA程序中可以有多个main方法 C: JAVA程序的main方法必须写在类里面 D: JAVA程序中类名必须与文件名一样 答案: C</p>	<p>考生: 豆豆 考号: 1000001921051</p> <p>一、选择题</p> <p>第1题: 下列说法正确的是 A: JAVA程序中可以有多个main方法 B: JAVA程序的main方法必须写在类里面 C: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来 D: JAVA程序中类名必须与文件名一样 答案: B</p> <p>第2题: 表达式$(11+3*8)/4%3$的值是 A: 2 B: 1 C: 31 D: 0 答案: A</p> <p>第3题: 以下()不是合法的标识符 A: void B: de\$f C: x3s D: STRING 答案: A</p> <p>第4题: JAVA所定义的版本中不包括 A: JAVA2 Card B: JAVA2 HE C: JAVA2 ME D: JAVA2 BE E: JAVA2 SE 答案: B</p>	<p>考生: 大宝 考号: 1000001921987</p> <p>一、选择题</p> <p>第1题: 以下()不是合法的标识符 A: x3s; B: de\$f C: void D: STRING 答案: C</p> <p>第2题: 表达式$(11+3*8)/4%3$的值是 A: 1 B: 0 C: 2 D: 1 答案: C</p> <p>第3题: 变量命名规范说法正确的是 A: 不同类型的变量，可以起相同的名字； B: 变量由字母、下划线、数字、\$符号随意组成； C: 变量不能以数字作为开头； D: A和a在java中是同一个变量； 答案: C</p> <p>第4题: 下列说法正确的是 A: JAVA程序的main方法中如果只有一条语句，可以不用{}(大括号)括起来 B: JAVA程序的main方法必须写在类里面 C: JAVA程序中类名必须与文件名一样 D: JAVA程序中可以有多个main方法 答案: B</p>
--	--	---



憨憨找题
1. 题目顺序已乱序
2. 考试答案已乱序
憨憨，不能作弊了

六、总结

- 以上的实际场景模拟了原型模式在开发中重构的作用，但是原型模式的使用频率确实不是很高。如果有有一些特殊场景需要使用到，也可以按照此设计模式进行优化。
- 另外原型设计模式的优点包括：便于通过克隆方式创建复杂对象、也可以避免重复做初始化操作、不需要与类中所属的其他类耦合等。但也有一些缺点如果对象中包括了循环引用的克隆，以及类中深度使用对象的克隆，都会使此模式变得异常麻烦。
- 终究设计模式是一整套的思想，在不同的场景合理的运用可以提升整体的架构的质量。永远不要想

着去硬凑设计模式，否则将会引起过渡设计，以及在承接业务反复变化的需求时造成浪费的开发和维护成本。

- 初期是代码的优化，中期是设计模式的使用，后期是把控全局服务的搭建。不断的加强自己对全局能力的把控，也加深自己对细节的处理。可上可下才是一个程序员最佳处理方式，选取做合适的是最好的选择。

第 5 节：单例模式

5个创建型模式的最后一个

在设计模式中按照不同的处理方式共包含三大类；**创建型模式**、**结构型模式**和**行为模式**，其中**创建型模式**目前已经介绍了其中的四个；**工厂方法模式**、**抽象工厂模式**、**生成器模式**和**原型模式**，除此之外还有最后一个**单例模式**。

掌握了的知识才是自己的

在本次编写的**重学 Java 设计模式**的编写中尽可能多的用各种场景案例还介绍设计的使用，包括我们已经使用过的场景；**各种类型奖品发放**、**多套Redis缓存集群升级**、**装修公司报价清单**和**百份考卷题目与答案乱序**，通过这些场景案例的实践感受设计模式的思想。但这些场景都是作者通过经验分离出来的，还并不是读者的知识，所以你如果希望可以融会贯通的掌握那么一定要亲力亲为的操作，事必躬亲的完成。

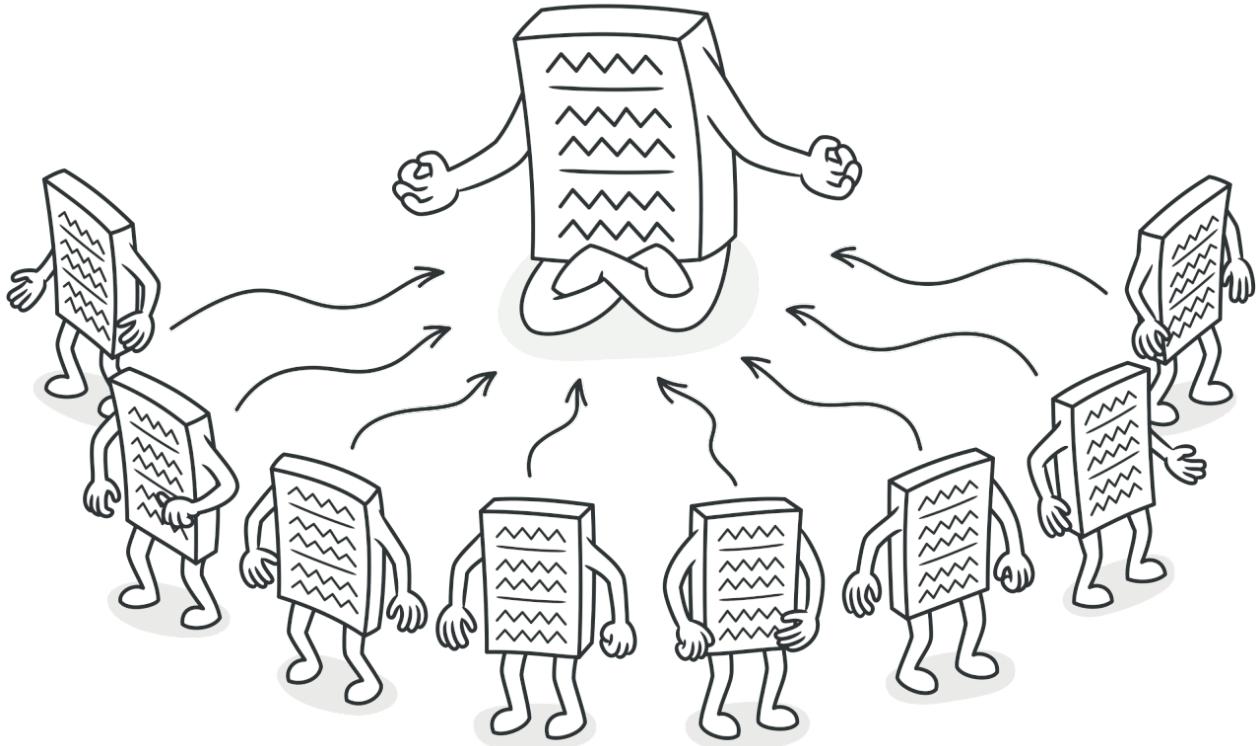
书不是看的是用的

在这里还是想强调一下学习方法，总有很多小伙伴对学习知识有疑惑，明明看了、看的时候也懂了，但到了实际使用的时候却用不上。或者有时候在想是不要是有更加生动的漫画或者什么对比会好些，当然这些方式可能会加快一个新人对知识的理解速度。但只要你把学习视频当电影看、学习书籍当故事看，就很难掌握这项技术栈。只有你把它用起来，逐字逐句的深挖，一点点的探求，把各项遇到的盲点全部扫清，才能让你真的掌握这项技能。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程1个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

二、单例模式介绍



单例模式可以说是整个设计中最简单的模式之一，而且这种方式即使在没有看设计模式相关资料也会常用在编码开发中。

因为在编程开发中经常会遇到这样一种场景，那就是需要保证一个类只有一个实例哪怕多线程同时访问，并需要提供一个全局访问此实例的点。

综上以及我们平常的开发中，可以总结一条经验，单例模式主要解决的是，一个全局使用的类频繁的创建和消费，从而提升提升整体的代码的性能。

三、案例场景

本章节的技术所出现的场景非常简单也是我们日常开发所能见到的，例如：

1. 数据库的连接池不会反复创建
2. spring中一个单例模式bean的生成和使用
3. 在我们平常的代码中需要设置全局的一些属性保存

在我们的日常开发中大致上会出现如上这些场景中使用到单例模式，虽然单例模式并不复杂但是使用面却比较广。

四、7种单例模式实现

单例模式的实现方式比较多，主要在实现上是否支持懒汉模式、是否线程安全中运用各项技巧。当然也有一些场景不需要考虑懒加载也就是懒汉模式的情况，会直接使用 `static` 静态类或属性和方法的方式进行处理，供外部调用。

那么接下来我们就通过实现不同方式的实现进行讲解单例模式。

0. 静态类使用

```
1 public class Singleton_00 {  
2  
3     public static Map<String, String> cache = new ConcurrentHashMap<String,  
4         String>();  
5 }
```

- 以上这种方式在我们平常的业务开发中非常常见，这样静态类的方式可以在第一次运行的时候直接初始化Map类，同时这里我们也不需要到延迟加载在使用。
- 在不需要维持任何状态下，仅仅用于全局访问，这个使用使用静态类的方式更加方便。
- 但如果需要被继承以及需要维持一些特定状态的情况下，就适合使用单例模式。

1. 懒汉模式(线程不安全)

```
1 public class Singleton_01 {  
2  
3     private static Singleton_01 instance;  
4  
5     private Singleton_01() {  
6     }  
7  
8     public static Singleton_01 getInstance(){  
9         if (null != instance) return instance;  
10        instance = new Singleton_01();  
11        return instance;  
12    }  
13  
14 }
```

- 单例模式有一个特点就是不允许外部直接创建，也就是 `new Singleton_01()`，因此这里在默认的构造函数上添加了私有属性 `private`。
- 目前此种方式的单例确实满足了懒加载，但是如果多个访问者同时去获取对象实例你可以想象成一堆人在抢厕所，就会造成多个同样的实例并存，从而没有达到单例的要求。

2. 懒汉模式(线程安全)

```
1 public class Singleton_02 {  
2  
3     private static Singleton_02 instance;  
4  
5     private Singleton_02() {  
6     }  
7  
8     public static synchronized Singleton_02 getInstance(){  
9         if (null != instance) return instance;  
10        instance = new Singleton_02();  
11        return instance;  
12    }
```

```
13  
14 }
```

- 此种模式虽然是安全的，但由于把锁加到方法上后，所有的访问都因需要锁占用导致资源的浪费。如果不是特殊情况下，不建议此种方式实现单例模式。

3. 饿汉模式(线程安全)

```
1 public class Singleton_03 {  
2  
3     private static Singleton_03 instance = new Singleton_03();  
4  
5     private Singleton_03() {  
6     }  
7  
8     public static Singleton_03 getInstance() {  
9         return instance;  
10    }  
11  
12 }
```

- 此种方式与我们开头的第一个实例化 Map 基本一致，在程序启动的时候直接运行加载，后续有外部需要使用的时候获取即可。
- 但此种方式并不是懒加载，也就是说无论你程序中是否用到这样的类都会在程序启动之初进行创建。
- 那么这种方式导致的问题就像你下载个游戏软件，可能你游戏地图还没有打开呢，但是程序已经将这些地图全部实例化。到你手机上最明显体验就一开游戏内存满了，手机卡了，需要换了。

4. 使用类的内部类(线程安全)

```
1 public class Singleton_04 {  
2  
3     private static class SingletonHolder {  
4         private static Singleton_04 instance = new Singleton_04();  
5     }  
6  
7     private Singleton_04() {  
8     }  
9  
10    public static Singleton_04 getInstance() {  
11        return SingletonHolder.instance;  
12    }  
13  
14 }
```

- 使用类的静态内部类实现的单例模式，既保证了线程安全有保证了懒加载，同时不会因为加锁的方式耗费性能。
- 这主要是因为JVM虚拟机可以保证多线程并发访问的正确性，也就是一个类的构造方法在多线程环

境下可以被正确的加载。

- 此种方式也是非常推荐使用的一种单例模式

5. 双重锁校验(线程安全)

```
1 public class Singleton_05 {  
2  
3     private static Singleton_05 instance;  
4  
5     private Singleton_05() {  
6     }  
7  
8     public static Singleton_05 getInstance(){  
9         if(null != instance) return instance;  
10        synchronized (Singleton_05.class){  
11            if (null == instance){  
12                instance = new Singleton_05();  
13            }  
14        }  
15        return instance;  
16    }  
17  
18 }
```

- 双重锁的方式是方法级锁的优化，减少了部分获取实例的耗时。
- 同时这种方式也满足了懒加载。

6. CAS 「AtomicReference」 (线程安全)

```
1 public class Singleton_06 {  
2  
3     private static final AtomicReference<Singleton_06> INSTANCE = new  
4         AtomicReference<Singleton_06>();  
5  
6     private static Singleton_06 instance;  
7  
8     private Singleton_06() {  
9     }  
10  
11    public static final Singleton_06 getInstance() {  
12        for (; ; ) {  
13            Singleton_06 instance = INSTANCE.get();  
14            if (null != instance) return instance;  
15            INSTANCE.compareAndSet(null, new Singleton_06());  
16            return INSTANCE.get();  
17        }  
18    }  
19  
20    public static void main(String[] args) {
```

```
20     System.out.println(Singleton_06.getInstance()); //  
21     org.itstack.demo.design.Singleton_06@2b193f2d  
22     System.out.println(Singleton_06.getInstance()); //  
23     org.itstack.demo.design.Singleton_06@2b193f2d  
24 }
```

- java并发库提供了很多原子类来支持并发访问的数据安全性；`AtomicInteger`、`AtomicBoolean`、`AtomicLong`、`AtomicReference`。
- `AtomicReference` 可以封装引用一个V实例，支持并发访问如上的单例方式就是使用了这样的一个特点。
- 使用CAS的好处就是不需要使用传统的加锁方式保证线程安全，而是依赖于CAS的忙等算法，依赖于底层硬件的实现，来保证线程安全。相对于其他锁的实现没有线程的切换和阻塞也就没有了额外的开销，并且可以支持较大的并发性。
- 当然CAS也有一个缺点就是忙等，如果一直没有获取到将会处于死循环中。

7. Effective Java作者推荐的枚举单例(线程安全)

```
1 public enum Singleton_07 {  
2  
3     INSTANCE;  
4     public void test(){  
5         System.out.println("hi~");  
6     }  
7 }  
8 }
```

约书亚·布洛克（英语：Joshua J. Bloch，1961年8月28日—），美国著名程序员。他为Java平台设计并实作了许多的功能，曾担任Google的首席Java架构师（Chief Java Architect）。

- Effective Java 作者推荐使用枚举的方式解决单例模式，此种方式可能是平时最少用到的。
- 这种方式解决了最主要的；线程安全、自由串行化、单一实例。

调用方式

```
1 @Test  
2 public void test() {  
3     Singleton_07.INSTANCE.test();  
4 }
```

这种写法在功能上与共有域方法相近，但是它更简洁，无偿地提供了串行化机制，绝对防止对此实例化，即使是在面对复杂的串行化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现Singleton的最佳方法。

但也要知道此种方式在存在继承场景下是不可用的。

五、总结

- 虽然只是一个很平常的单例模式，但在各种的实现上真的可以看到java的基本功的体现，这里包括

- 了；懒汉、饿汉、线程是否安全、静态类、内部类、加锁、串行化等等。
- 在平时的开发中如果可以确保此类是全局可用不需要做懒加载，那么直接创建并给外部调用即可。但如果是很多的类，有些需要在用户触发一定的条件后(游戏关卡)才显示，那么一定要用懒加载。线程的安全上可以按需选择。
 - 建议在学习的过程中一定要加以实践，否则很难完完整整的掌握一整套的知识体系。例如案例中的出现的 `Effective Java` 一书也非常建议大家阅读。另外推荐下这位大神的 Github: <https://github.com/jbloch>

结构型模式(7节)

这类模式介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。

结构型模式包括：适配器、桥接、组合、装饰器、外观、享元、代理，这7类。

第1节：适配器模式

擦屁屁纸80%的面积都是保护手的！

工作到3年左右很大一部分程序员都想提升自己的技术栈，开始尝试去阅读一些源码，例如 Spring、Mybatis、Dubbo 等，但读着读着发现越来越难懂，一会从这过来一会跑到那去。甚至怀疑自己技术太差，慢慢也就不愿意再触碰这部分知识。

而这主要的原因是一个框架随着时间的发展，它的复杂程度是越来越高的，从最开始只有一个非常核心的点到最后开枝散叶。这就像你自己开发的业务代码或者某个组件一样，最开始的那部分核心代码也许只能占到20%，而其他大部分代码都是为了保证核心流程能正常运行的。所以这也是你读源码费劲的一部分原因。

框架中用到了设计模式吗？

框架中不仅用到设计模式还用了很多，而且有些时候根本不是一个模式的单独使用，而是多种设计模式的综合运用。与大部分小伙伴平时开发的CRUD可就不一样了，如果都是if语句从上到下，也就算得不上什么框架了。就像你到Spring的源码中搜关键字 Adapter，就会出现很多实现类，例如：UserCredentialsDataSourceAdapter。而这种设计模式就是我们本文要介绍的适配器模式。

适配器在生活里随处可见

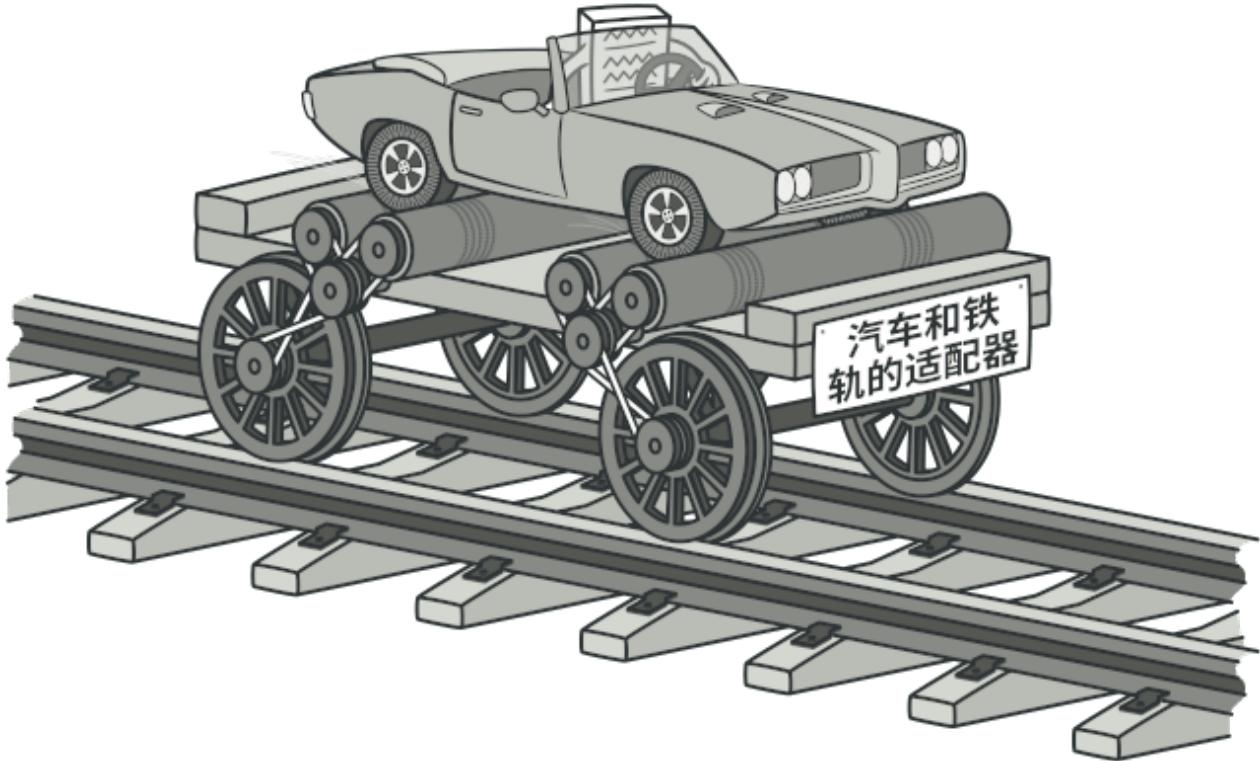
如果提到在日常生活中就很多适配器的存在你会想到什么？在没有看后文之前可以先思考下。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-6-00	场景模拟工程；模拟多个MQ消息体
itstack-demo-design-6-01	使用一坨代码实现业务需求
itstack-demo-design-6-02	通过设计模式优化改造代码，产生对比性从而学习

二、适配器模式介绍



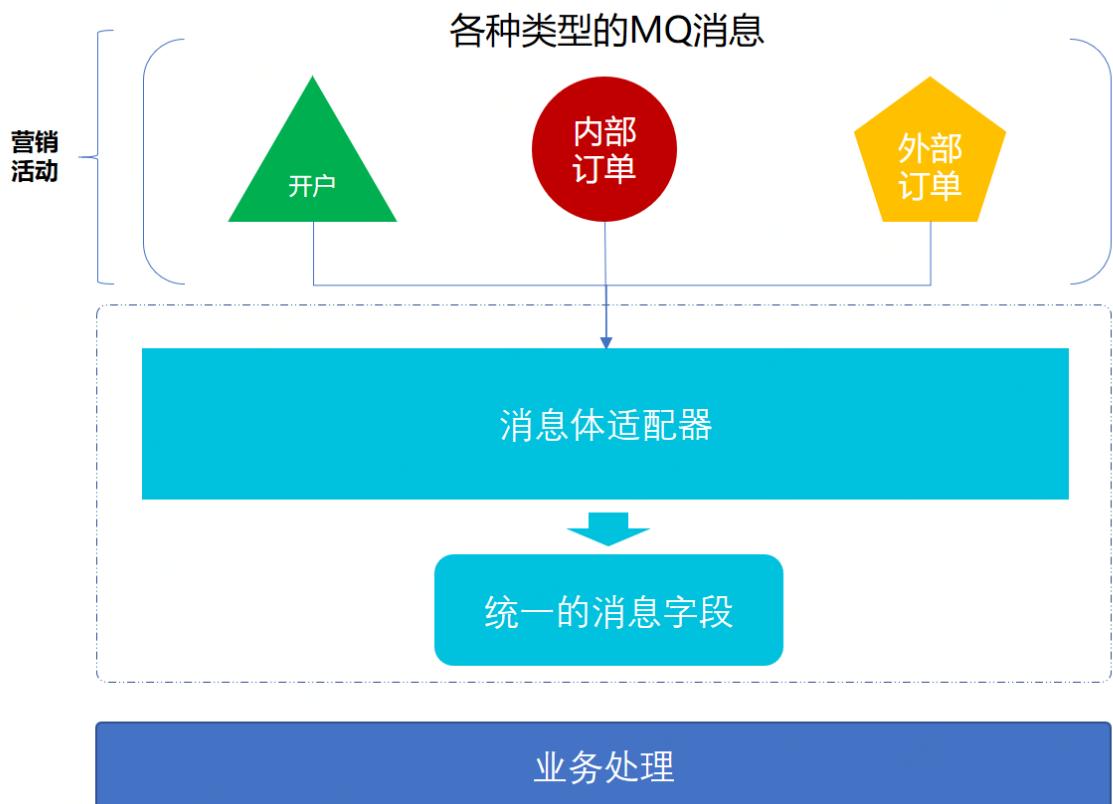
适配器模式的主要作用就是把原本不兼容的接口，通过适配修改做到统一。使得用户方便使用，就像我们提到的万能充、数据线、MAC笔记本的转换头、出国旅游买个插座等等，他们都是为了适配各种不同的口，做的兼容。。



除了我们生活中出现的各种适配的场景，那么在业务开发中呢？

在业务开发中我们会经常的需要做不同接口的兼容，尤其是中台服务，中台需要把各个业务线的各种类型服务做统一包装，再对外提供接口进行使用。而这在我们平常的开发中也是非常常见的。

三、案例场景模拟



随着公司的业务的不断发展，当基础的系统逐步成型以后。业务运营就需要开始做用户的拉新和促活，从而保障 DUA 的增速以及最终 ROI 转换。

而这时候就会需要做一些营销系统，大部分常见的都是裂变、拉客，例如：你邀请一个用户开户、或者邀请一个用户下单，那么平台就会给你返利，多邀多得。同时随着拉新的量越来越多开始设置每月下单都会给首单奖励，等等，各种营销场景。

那么这个时候做这样一个系统就会接收各种各样的MQ消息或者接口，如果一个个的去开发，就会耗费很大的成本，同时对于后期的拓展也有一定的难度。此时就会希望有一个系统可以配置一下就把外部的MQ接入进行，这些MQ就像上面提到的可能是一些注册开户消息、商品下单消息等等。

而适配器的思想方式也恰恰可以运用到这里，并且我想强调一下，适配器不只是可以适配接口往往还可以适配一些属性信息。

1. 场景模拟工程

```

1  itstack-demo-design-6-00
2  └── src
3      └── main
4          └── java
5              └── org.itstack.demo.design
6                  └── mq
7                      ├── create_account.java
8                      ├── OrderMq.java
9                      └── POPOrderDelivered.java
10             └── service
11                 ├── OrderService.java
12                 └── POPOrderService.java

```

- 这里模拟了三个不同类型的MQ消息，而在消息体中都有一些必要的字段，比如：用户ID、时间、业务ID，但是每个MQ的字段属性并不一样。就像用户ID在不同的MQ里也有不同的字段：uId、userId等。
- 同时还提供了两个不同类型的接口，一个用于查询内部订单订单下单数量，一个用于查询第三方是否首单。
- 后面会把这些不同类型的MQ和接口做适配兼容。

2. 场景简述

1.1 注册开户MQ

```

1 public class CreateAccount {
2
3     private String number;          // 开户编号
4     private String address;         // 开户地
5     private Date accountDate;       // 开户时间
6     private String desc;            // 开户描述
7
8     // ... get/set
9 }
```

1.2 内部订单MQ

```

1 public class OrderMq {
2
3     private String uid;             // 用户ID
4     private String sku;              // 商品
5     private String orderId;          // 订单ID
6     private Date createOrderTime;    // 下单时间
7
8     // ... get/set
9 }
```

1.3 第三方订单MQ

```

1 public class POPOrderDelivered {
2
3     private String uId;             // 用户ID
4     private String orderId;          // 订单号
5     private Date orderTime;         // 下单时间
6     private Date sku;               // 商品
7     private Date skuName;           // 商品名称
8     private BigDecimal decimal;      // 金额
9
10    // ... get/set
11 }
```

1.4 查询用户内部下单数量接口

```
1 public class OrderService {  
2  
3     private Logger logger =  
4         LoggerFactory.getLogger(POPOrderService.class);  
5  
6     public long queryUserOrderCount(String userId){  
7         logger.info("自营商家, 查询用户的订单是否为首单: {}", userId);  
8         return 10L;  
9     }  
10 }
```

1.5 查询用户第三方下单首单接口

```
1 public class POPOrderService {  
2  
3     private Logger logger =  
4         LoggerFactory.getLogger(POPOrderService.class);  
5  
6     public boolean isFirstOrder(String uId) {  
7         logger.info("POP商家, 查询用户的订单是否为首单: {}", uId);  
8         return true;  
9     }  
10 }
```

- 以上这几项就是不同的MQ以及不同的接口的一个体现，后面我们将使用这样的MQ消息和接口，给它们做相应的适配。

四、用一坨坨代码实现

其实大部分时候接MQ消息都是创建一个类用于消费，通过转换他的MQ消息属性给自己的方法。

我们接下来也是先体现一下这种方式的实现模拟，但是这样的实现有一个很大的问题就是，当MQ消息越来越多后，甚至几十几百以后，你作为中台要怎么优化呢？

1. 工程结构

```
1 itstack-demo-design-6-01  
2 └── src  
3     └── main  
4         └── java  
5             └── org.itstack.demo.design  
6                 └── create_accountMqService.java  
7                 └── OrderMqService.java  
8                 └── POPOrderDeliveredService.java
```

- 目前需要接收三个MQ消息，所有就有了三个对应的类，和我们平时的代码几乎一样。如果你的

MQ量不多，这样的写法也没什么问题，但是随着数量的增加，就需要考虑用一些设计模式来解决。

2. Mq接收消息实现

```
1 public class create_accountMqService {
2
3     public void onMessage(String message) {
4
5         create_account mq = JSON.parseObject(message,
6         create_account.class);
7
8         mq.getNumber();
9         mq.getAccountDate();
10
11        // ... 处理自己的业务
12    }
13 }
```

- 三组MQ的消息都是一样模拟使用，就不一一展示了。可以获取源码后学习。

五、适配器模式重构代码

接下来使用适配器模式来进行代码优化，也算是一次很小的重构。

适配器模式要解决的主要问题就是多种差异化类型的接口做统一输出，这在我们学习工厂方法模式中也有所提到不同种类的奖品处理，其实那也是适配器的应用。

在本文中我们还会再另外体现出一个多种MQ接收，使用MQ的场景。来把不同类型的消息做统一的处理，便于减少后续对MQ接收。

在这里如果你之前没要开发过接收MQ消息，可能听上去会有些不理解这样的场景。对此，我个人建议先了解下MQ。另外就算不了解也没关系，不会影响对思路的体会。

再者，本文所展示的MQ兼容的核心部分，也就是处理适配不同的类型字段。而如果我们接收MQ后，在配置不同的消费类时，如果不希望一个个开发类，那么可以使用代理类的方式进行处理。

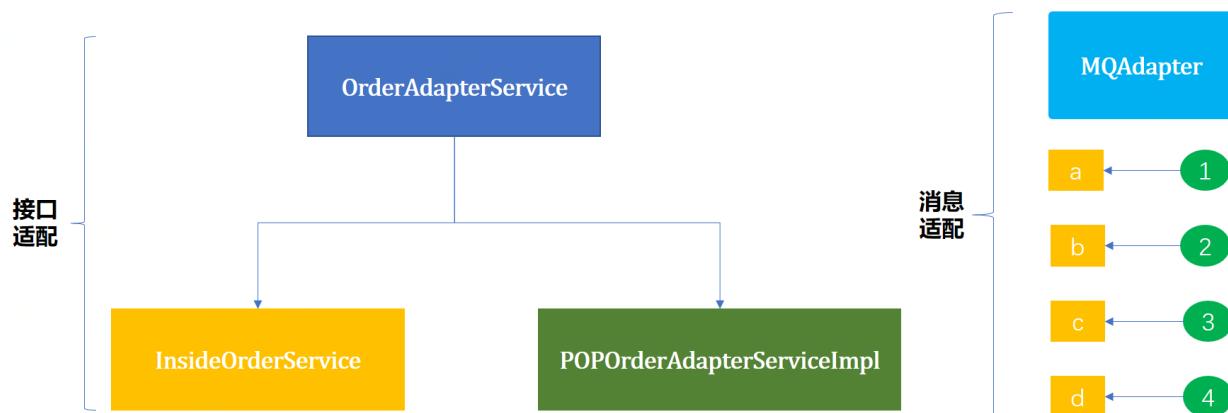
1. 工程结构

```

1 itstack-demo-design-6-02
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── impl
7                 |   └── InsideOrderService.java
8                 |   └── POPOrderAdapterServiceImpl.java
9                 └── MQAdapter.java
10                └── OrderAdapterService.java
11                └── RebateInfo.java

```

适配器模型结构



- 这里包括了两个类型的适配；接口适配、MQ适配。之所以不只是模拟接口适配，因为很多时候大家都很常见了，所以把适配的思想换一下到MQ消息体上，增加大家对设计模式的认知。
- 先是做MQ适配**，接收各种各样的MQ消息。当业务发展的很快，需要对下单用户首单才给奖励，在这样的场景下再增加对接口的适配操作。

2. 代码实现(MQ消息适配)

2.1 统一的MQ消息体

```

1 public class RebateInfo {
2
3     private String userId; // 用户ID
4     private String bizId; // 业务ID
5     private Date bizTime; // 业务时间
6     private String desc; // 业务描述
7
8     // ... get/set
9 }

```

- MQ消息中会有多种多样的类型属性，虽然他们都有同样的值提供给使用方，但是如果都这样接入那么当MQ消息特别多时候就会很麻烦。
- 所以在一个案例中我们定义了通用的MQ消息体，后续把所有接入进来的消息进行统一的处理。

2.2 MQ消息体适配类

```
1 public class MQAdapter {  
2  
3     public static RebateInfo filter(String strJson, Map<String, String>  
4         link) throws NoSuchMethodException, InvocationTargetException,  
5         IllegalAccessException {  
6         return filter(JSON.parseObject(strJson, Map.class), link);  
7     }  
8  
9     public static RebateInfo filter(Map obj, Map<String, String> link)  
10    throws NoSuchMethodException, InvocationTargetException,  
11        IllegalAccessException {  
12        RebateInfo rebateInfo = new RebateInfo();  
13        for (String key : link.keySet()) {  
14            Object val = obj.get(link.get(key));  
15            RebateInfo.class.getMethod("set" + key.substring(0,  
16                1).toUpperCase() + key.substring(1), String.class).invoke(rebateInfo,  
17                val.toString());  
18        }  
19        return rebateInfo;  
20    }  
21  
22}
```

- 这个类里的方法非常重要，主要用于把不同类型MQ中的各种属性，映射成我们需要的属性并返回。就像一个属性中有 `userID;uId`，映射到我们需要的；`userId`，做统一处理。
- 而在这个处理过程中需要把映射管理传递给 `Map<String, String> link`，也就是准确的描述了，当前MQ中某个属性名称，映射为我们的某个属性名称。
- 最终因为我们接收到的 `mq` 消息基本都是 `json` 格式，可以转换为MAP结构。最后使用反射调用的方式给我们的类型赋值。

2.3 测试适配类

2.3.1 编写单元测试类

```
1 @Test  
2 public void test_MQAdapter() throws NoSuchMethodException,  
3     IllegalAccessException, InvocationTargetException {  
4     create_account create_account = new create_account();  
5     create_account.setNumber("100001");  
6     create_account.setAddress("河北省.廊坊市.广阳区.大学里职业技术学院");  
7     create_account.setAccountDate(new Date());  
8     create_account.setDesc("在校开户");  
9  
10    HashMap<String, String> link01 = new HashMap<String, String>();  
11    link01.put("userId", "number");  
12    link01.put("bizId", "number");  
13    link01.put("bizTime", "accountDate");
```

```

13     link01.put("desc", "desc");
14     RebateInfo rebateInfo01 = MQAdapter.filter(create_account.toString(),
15         link01);
16     System.out.println("mq.create_account(适配前) " +
17         create_account.toString());
18     System.out.println("mq.create_account(适配后) " +
19         JSON.toJSONString(rebateInfo01));
20
21     OrderMq orderMq = new OrderMq();
22     orderMq.setUid("100001");
23     orderMq.setSku("10928092093111123");
24     orderMq.setOrderId("100000890193847111");
25     orderMq.setCreateOrderTime(new Date());
26
27     HashMap<String, String> link02 = new HashMap<String, String>();
28     link02.put("userId", "uid");
29     link02.put("bizId", "orderId");
30     link02.put("bizTime", "createOrderTime");
31     RebateInfo rebateInfo02 = MQAdapter.filter(orderMq.toString(),
32         link02);
33
34     System.out.println("mq.orderMq(适配前) " + orderMq.toString());
35     System.out.println("mq.orderMq(适配后) " +
36         JSON.toJSONString(rebateInfo02));
37 }
```

- 在这里我们分别模拟传入了两个不同的MQ消息，并设置字段的映射关系。
- 等真的业务场景开发中，就可以配这种映射配置关系交给配置文件或者数据库后台配置，减少编码。

2.3.2 测试结果

```

1 mq.create_account(适配前){"accountDate":1591024816000,"address":"河北省·廊坊
2 市·广阳区·大学里职业技术学院","desc":"在校开户","number":"100001"}
3 mq.create_account(适配后){"bizId":"100001","bizTime":1591077840669,"desc":"在
4 校开户","userId":"100001"}
5 mq.orderMq(适配前)
6 {"createOrderTime":1591024816000,"orderId":"100000890193847111","sku":"1092
7 8092093111123","uid":"100001"}
8 mq.orderMq(适配后)
9 {"bizId":"100000890193847111","bizTime":1591077840669,"userId":"100001"}}
10
11 Process finished with exit code 0
```

- 从上面可以看到，同样的字段值在做了适配前后分别有统一的字段属性，进行处理。这样业务开发中也就非常简单了。

- 另外有一个非常重要的地方，在实际业务开发中，除了反射的使用外，还可以加入代理类把映射的配置交给它。这样就可以不需要每一个mq都手动创建类了。

3. 代码实现(接口使用适配)

就像我们前面提到随着业务的发展，营销活动本身要修改，不能只是接了MQ就发奖励。因为此时已经拉新的越来越多了，需要做一些限制。

因为增加了只有首单用户才给奖励，也就是你一年或者新人或者一个月的第一单才给你奖励，而不是你之前每一次下单都给奖励。

那么就需要对此种方式进行限制，而此时MQ中并没有判断首单的属性。只能通过接口进行查询，而拿到的接口如下；

接口	描述
org.itstack.demo.design.service.OrderService.queryUserOrderCount(String userId)	出参long，查询订单数量
org.itstack.demo.design.service.OrderService.isFirstOrder(String uid)	出参boolean，判断是否首单

- 两个接口的判断逻辑和使用方式都不同，不同的接口提供方，也有不同的出参。一个是直接判断是否首单，另外一个需要根据订单数量判断。
- 因此这里需要使用到适配器的模式来实现，当然如果你去编写if语句也是可以实现的，但是我们经常会提到这样的代码很难维护。

3.1 定义统一适配接口

```

1 public interface OrderAdapterService {
2
3     boolean isFirst(String uid);
4
5 }
```

- 后面的实现类都需要完成此接口，并把具体的逻辑包装到指定的类中，满足单一职责。

3.2 分别实现两个不同的接口

内部商品接口

```

1 public class InsideOrderService implements OrderAdapterService {
2
3     private OrderService orderService = new OrderService();
4
5     public boolean isFirst(String uid) {
6         return orderService.queryUserOrderCount(uid) <= 1;
7     }
8
9 }
```

第三方商品接口

```
1 public class POPOrderAdapterServiceImpl implements OrderAdapterService {  
2  
3     private POPOrderService popOrderService = new POPOrderService();  
4  
5     public boolean isFirst(String uId) {  
6         return popOrderService.isFirstOrder(uId);  
7     }  
8  
9 }
```

- 在这两个接口中都实现了各自的判断方式，尤其像是提供订单数量的接口，需要自己判断当前接到mq时订单数量是否 `<= 1`，以此判断是否为首单。

3.3 测试适配类

3.3.1 编写单元测试类

```
1 @Test  
2 public void test_itfAdapter() {  
3     OrderAdapterService popOrderAdapterService = new  
4         POPOrderAdapterServiceImpl();  
5     System.out.println("判断首单, 接口适配(POP): " +  
6         popOrderAdapterService.isFirst("100001"));  
7  
8     OrderAdapterService insideOrderService = new InsideOrderService();  
9     System.out.println("判断首单, 接口适配(自营): " +  
10        insideOrderService.isFirst("100001"));  
11 }
```

3.3.2 测试结果

```
1 23:25:47.076 [main] INFO o.i.d.design.service.POPOrderService - POP商家, 查  
2 询用户的订单是否为首单: 100001  
3 判断首单, 接口适配(POP): true  
4 23:25:47.079 [main] INFO o.i.d.design.service.POPOrderService - 自营商家, 查  
5 询用户的订单是否为首单: 100001  
6 判断首单, 接口适配(自营): false  
7  
8 Process finished with exit code 0
```

- 从测试结果上来看，此时已经的接口已经做了统一的包装，外部使用时候就不需要关心内部的具体逻辑了。而且在调用的时候只需要传入统一的参数即可，这样就满足了适配的作用。

六、总结

- 从上文可以看到不使用适配器模式这些功能同样可以实现，但是使用了适配器模式就可以让代码：干净整洁易于维护、减少大量重复的判断和使用、让代码更加易于维护和拓展。
- 尤其是我们对MQ这样的多种消息体中不同属性同类的值，进行适配再加上代理类，就可以使用简

单的配置方式接入对方提供的MQ消息，而不需要大量重复的开发。非常利于拓展。

- 设计模式的学习学习过程可能会在一些章节中涉及到其他设计模式的体现，只不过不会重点讲解，避免喧宾夺主。但在实际的使用中，往往很多设计模式是综合使用的，并不会单一出现。

第 2 节：桥接模式

为什么你的代码那么多 `if else`

同类的业务、同样的功能，怎么就你能写出来那么多 `if else`。很多时候一些刚刚从校园进入企业的萌新，或者一部分从小公司跳槽到大企业的程序员，初次承接业务需求的时候，往往编码还不成熟，经常一杆到底的写需求。初次实现确实很快，但是后期维护和扩展就十分痛苦。因为一段代码的可读性阅读他后期的维护成本也就越高。

设计模式是可以帮助你改善代码

很多时候你写出来的 `if else` 都是没有考虑使用设计模式优化，就像；同类服务的不同接口适配包装、同类物料不同组合的建造、多种奖品组合的营销工厂等等。它们都可以让你代码中原本使用 `if` 判断的地方，变成一组组类和面向对象的实现过程。

怎么把设计模式和实际开发结合起来

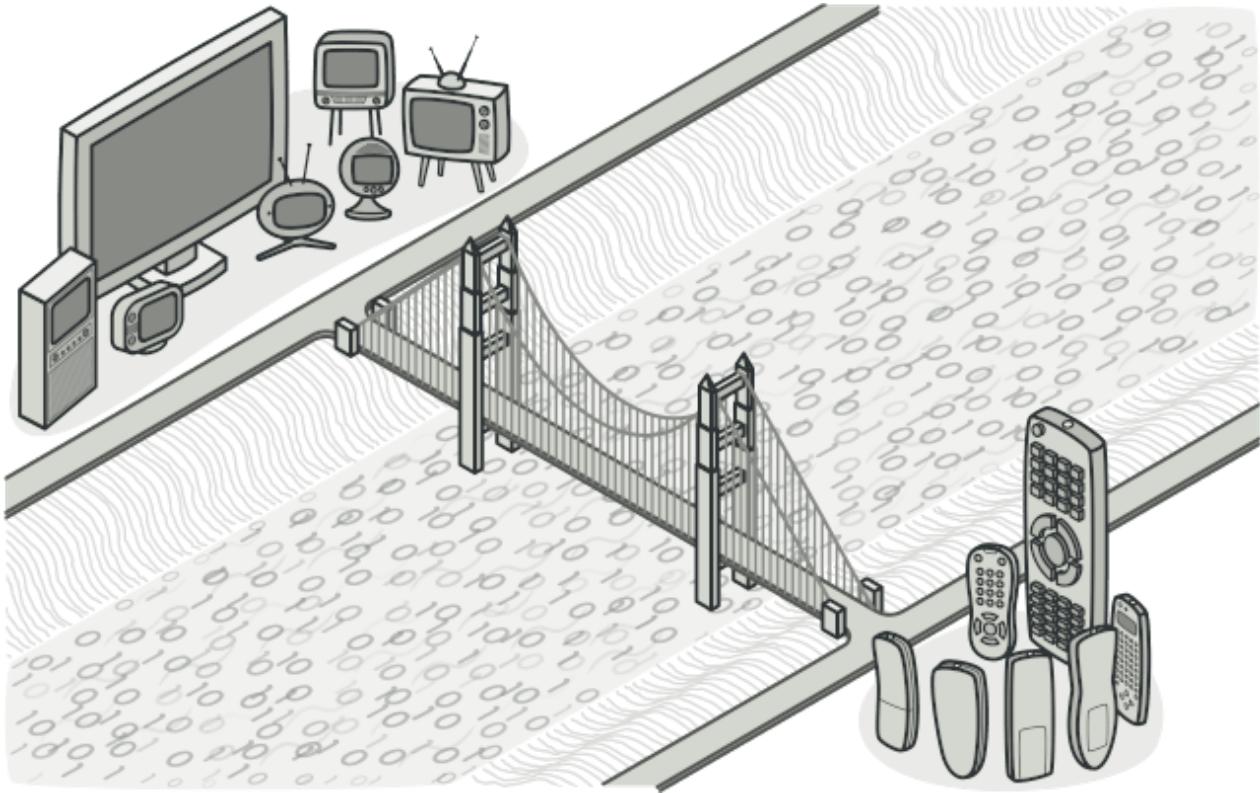
多从实际场景思考，只找到代码优化的最佳点，不要可以想着设计模式的使用。就像你最开始看设计模式适合，因为没有真实的场景模拟案例，都是一些画圆形、方形，对新人或者理解能力还不到的伙伴来说很不友好。所以即使学了半天，但实际使用还是摸不着头脑。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-7-01	使用一坨代码实现业务需求
itstack-demo-design-7-02	通过设计模式优化改造代码，产生对比性从而学习

二、桥接模式介绍

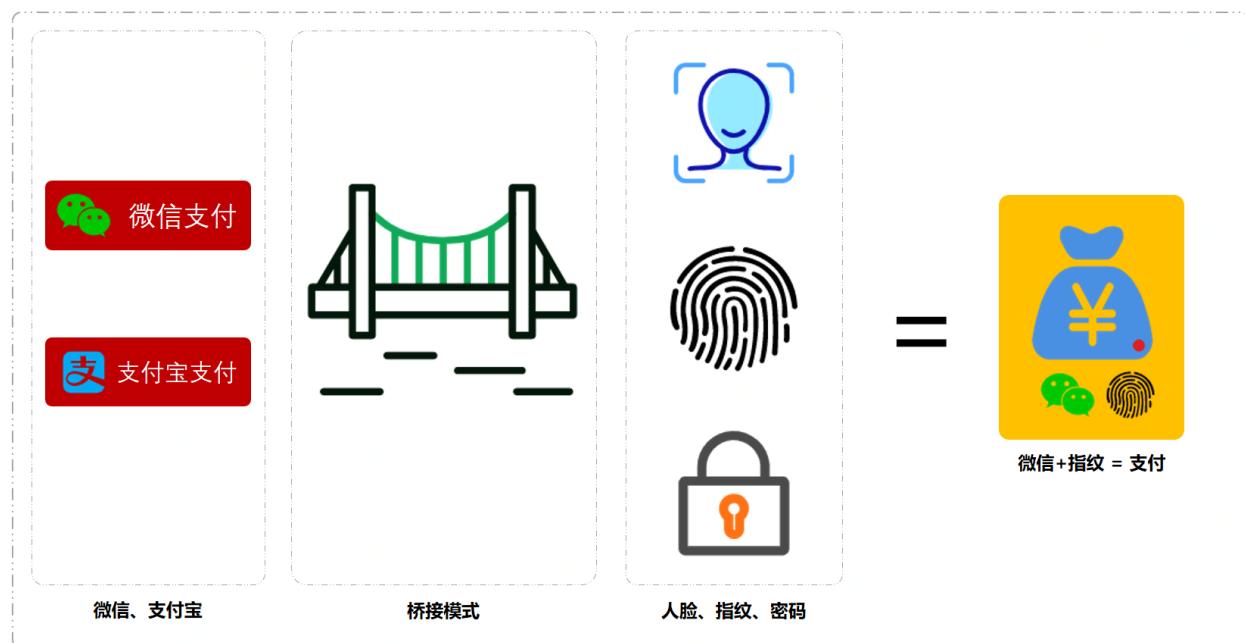


桥接模式的主要作用就是通过将抽象部分与实现部分分离，把多种可匹配的使用进行组合。说白了核心实现也就是在A类中含有B类接口，通过构造函数传递B类的实现，这个B类就是设计的桥。

那么这样的桥接模式，在我们平常的开发中有哪些场景

JDBC多种驱动程序的实现、同品牌类型的台式机和笔记本平板、业务实现中的多类接口同组过滤服务等。这些场景都比较适合使用桥接模式进行实现，因为在一些组合中如果有如果每一个类都实现不同的服务可能会出现笛卡尔积，而使用桥接模式就可以非常简单。

三、案例场景模拟



随着市场的竞争在支付服务行业出现了微信和支付宝还包括一些其他支付服务，但是对于商家来说并不希望改变用户习惯。就像如果我的地摊只能使用微信或者只能使用支付宝付款，那么就会让我顾客伤心，鸡蛋灌饼也卖不动了。

在这个时候就出现了第三方平台，把市面上综合占据市场90%以上的支付服务都集中到自己平台中，再把这样的平台提供给店铺、超市、地摊使用，同时支持人脸、扫描、密码多种方式。

我们这个案例就模拟一个这样的第三方平台来承接各个支付能力，同时使用自家的人脸让用户支付起来更加容易。那么这里就出现了多支付与多模式的融合使用，如果给每一个支付都实现一次不同的模式，即使是继承类也需要开发好多。而且随着后面接入了更多的支付服务或者支付方式，就会呈爆炸似的扩展。

所以你现在可以思考一下这样的场景该如何实现？

四、用一坨坨代码实现

产品经理说老板要的需求，要尽快上，kpi你看着弄！

既然你逼我那就别怪我无情，还没有我一个类写不完的需求！反正写完就完了，拿完绩效也要走了天天逼着写需求，代码越来越乱心疼后面的兄弟3秒。

1. 工程结构

```
1 itstack-demo-design-7-01
2   └── src
3     └── main
4       └── java
5         └── org.itstack.demo.design
6           └── PayController.java
```

- 只有一个类里面都是 if else，这个类实现了支付和模式的全部功能。

2. 代码实现

```
1 public class PayController {
2
3     private Logger logger = LoggerFactory.getLogger(PayController.class);
4
5     public boolean doPay(String uId, String tradeId, BigDecimal amount,
6             int channelType, int modeType) {
7         // 微信支付
8         if (1 == channelType) {
9             logger.info("模拟微信渠道支付划账开始。uId: {} tradeId: {} amount: {}",
10                uId, tradeId, amount);
11             if (1 == modeType) {
12                 logger.info("密码支付，风控校验环境安全");
13             } else if (2 == modeType) {
14                 logger.info("人脸支付，风控校验脸部识别");
15             } else if (3 == modeType) {
16                 logger.info("指纹支付，风控校验指纹信息");
17             }
18         }
19     }
20 }
```

```

15     }
16 }
17 // 支付宝支付
18 else if (2 == channelType) {
19     logger.info("模拟支付宝渠道支付划账开始。uId: {} tradeId: {}"
20     amount: {}", uId, tradeId, amount);
21     if (1 == modeType) {
22         logger.info("密码支付，风控校验环境安全");
23     } else if (2 == modeType) {
24         logger.info("人脸支付，风控校验脸部识别");
25     } else if (3 == modeType) {
26         logger.info("指纹支付，风控校验指纹信息");
27     }
28     return true;
29 }
30
31 }

```

- 上面的类提供了一个支付服务功能，通过提供的必要字段；`用户ID`、`交易ID`、`金额`、`渠道`、`模式`，来控制支付方式。
- 以上的`ifelse`应该是最差的一种写法，即使写`ifelse`也是可以优化的方式去写的。

3. 测试验证

3.1 编写测试类

```

1 @Test
2 public void test_pay() {
3     PayController pay = new PayController();
4     System.out.println("\r\n模拟测试场景：微信支付、人脸方式。");
5     pay.doPay("weixin_1092033111", "100000109893", new BigDecimal(100), 1,
6     2);
7     System.out.println("\r\n模拟测试场景：支付宝支付、指纹方式。");
8     pay.doPay("jlu19dlxo111", "100000109894", new BigDecimal(100), 2, 3);
9 }

```

- 以上分别测试了两种不同的支付类型和支付模式；微信人脸支付、支付宝指纹支付

3.2 测试结果

```

1 模拟测试场景：微信支付、人脸方式。
2 23:05:59.152 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   划账开始。uId: weixin_1092033111 tradeId: 100000109893 amount: 100
3 23:05:59.155 [main] INFO o.i.demo.design.pay.mode.PayCypher - 人脸支付，风
   控校验脸部识别
4 23:05:59.155 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   风控校验。uId: weixin_1092033111 tradeId: 100000109893 security: true
5 23:05:59.155 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   划账成功。uId: weixin_1092033111 tradeId: 100000109893 amount: 100
6
7 模拟测试场景：支付宝支付、指纹方式。
8 23:05:59.156 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
   付划账开始。uId: jlu19dlxo111 tradeId: 100000109894 amount: 100
9 23:05:59.156 [main] INFO o.i.demo.design.pay.mode.PayCypher - 指纹支付，风
   控校验指纹信息
10 23:05:59.156 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
    付风控校验。uId: jlu19dlxo111 tradeId: 100000109894 security: true
11 23:05:59.156 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
    付划账成功。uId: jlu19dlxo111 tradeId: 100000109894 amount: 100
12
13 Process finished with exit code 0

```

- 从测试结果看已经满足了我们的不同支付类型和支付模式的组合，但是这样的代码在后面的维护以及扩展都会变得非常复杂。

五、桥接模式重构代码

接下来使用桥接模式来进行代码优化，也算是一次很小的重构。

从上面的 `if else` 方式实现来看，这是两种不同类型的相互组合。那么就可以把支付方式和支付模式进行分离通过抽象类依赖实现类的方式进行桥接，通过这样的拆分后支付与模式其实是可以单独使用的，当需要组合时候只需要把模式传递给支付即可。

桥接模式的关键是选择的桥接点拆分，是否可以找到这样类似的相互组合，如果没有就不必要非得使用桥接模式。

1. 工程结构

```

1 itstack-demo-design-7-02
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design.pay
6                 └── channel
7                     └── Pay.java
8                     └── WxPay.java
9                     └── ZfbPay.java
10                └── mode
11                    └── IPayMode.java

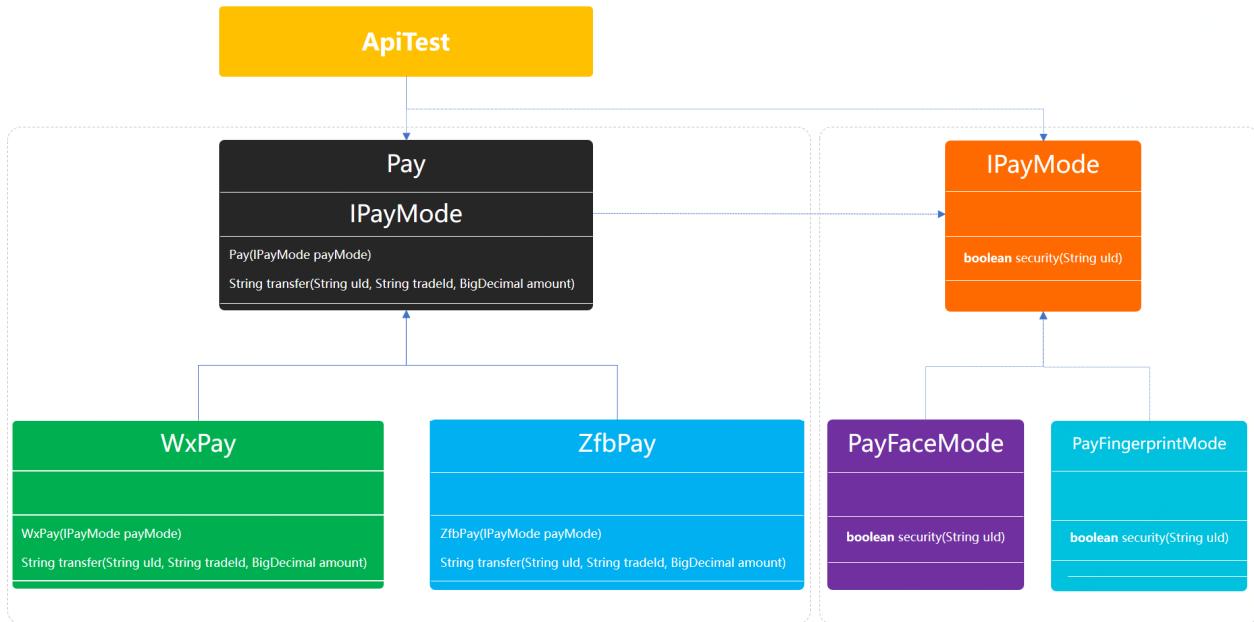
```

```

12 |           └── PayCypher.java
13 |           └── PayFaceMode.java
14 |           └── PayFingerprintMode.java
15 └── test
16   └── java
17     └── org.itstack.demo.design.test
18       └── ApiTest.java

```

桥接模式模型结构



- 左侧 **Pay** 是一个抽象类，往下是它的两个支付类型实现；微信支付、支付宝支付。
- 右侧 **IPayMode** 是一个接口，往下是它的两个支付模型；刷脸支付、指纹支付。
- 那么，**支付类型** × **支付模型** = 就可以得到相应的组合。
- 注意，每种支付方式的不同，刷脸和指纹校验逻辑也有差异，可以使用适配器模式进行处理，这里不是本文重点不做介绍，可以看适配器模式章节。

2. 代码实现

2.1 支付类型桥接抽象类

```

1 public abstract class Pay {
2
3     protected Logger logger = LoggerFactory.getLogger(Pay.class);
4
5     protected IPayMode payMode;
6
7     public Pay(IPayMode payMode) {
8         this.payMode = payMode;
9     }
10
11    public abstract String transfer(String uId, String tradeId, BigDecimal
amount);
12
13 }

```

- 在这个类中定义了支付方式的需要实现的划账接口： `transfer`，以及桥接接口； `IPayMode`，并在构造函数中用户方自行选择支付方式。
- 如果没有接触过此类实现，可以重点关注 `IPayMode payMode`，这部分是桥接的核心。

2.2 两个支付类型的实现

微信支付

```

1 public class WxPay extends Pay {
2
3     public WxPay(IPayMode payMode) {
4         super(payMode);
5     }
6
7     public String transfer(String uId, String tradeId, BigDecimal amount)
{
8         logger.info("模拟微信渠道支付划账开始。uId: {} tradeId: {} amount: {}",
uId, tradeId, amount);
9         boolean security = payMode.security(uId);
10        logger.info("模拟微信渠道支付风控校验。uId: {} tradeId: {} security:
{}", uId, tradeId, security);
11        if (!security) {
12            logger.info("模拟微信渠道支付划账拦截。uId: {} tradeId: {} amount:
{}", uId, tradeId, amount);
13            return "0001";
14        }
15        logger.info("模拟微信渠道支付划账成功。uId: {} tradeId: {} amount: {}",
uId, tradeId, amount);
16        return "0000";
17    }
18}
19}

```

支付宝支付

```

1 public class ZfbPay extends Pay {
2
3     public ZfbPay(IPayMode payMode) {
4         super(payMode);
5     }
6
7     public String transfer(String uId, String tradeId, BigDecimal amount)
8     {
9         logger.info("模拟支付宝渠道支付划账开始。uId: {} tradeId: {} amount: {}",
10            uId, tradeId, amount);
11        boolean security = payMode.security(uId);
12        logger.info("模拟支付宝渠道支付风控校验。uId: {} tradeId: {} security: {}",
13            uId, tradeId, security);
14        if (!security) {
15            logger.info("模拟支付宝渠道支付划账拦截。uId: {} tradeId: {} amount: {}",
16            uId, tradeId, amount);
17            return "0001";
18        }
19        logger.info("模拟支付宝渠道支付划账成功。uId: {} tradeId: {} amount: {}",
20            uId, tradeId, amount);
21        return "0000";
22    }
23
24 }
```

- 这里分别模拟了调用第三方的两个支付渠道；微信、支付宝，当然作为支付综合平台可能不只是接了这两个渠道，还会有其很多渠道。
- 另外可以看到在支付的时候分别都调用了风控的接口进行验证，也就是不同模式的支付(刷脸、指纹)，都需要过指定的风控，才能保证支付安全。

2.3 定义支付模式接口

```

1 public interface IPayMode {
2
3     boolean security(String uId);
4
5 }
```

- 任何一个支付模式；刷脸、指纹、密码，都会过不同程度的安全风控，这里定义一个安全校验接口。

2.4 三种支付模式风控(刷脸、指纹、密码)

刷脸

```
1 public class PayFaceMode implements IPayMode{  
2  
3     protected Logger logger = LoggerFactory.getLogger(PayCypher.class);  
4  
5     public boolean security(String uId) {  
6         logger.info("人脸支付, 风控校验脸部识别");  
7         return true;  
8     }  
9  
10 }
```

指纹

```
1 public class PayFingerprintMode implements IPayMode{  
2  
3     protected Logger logger = LoggerFactory.getLogger(PayCypher.class);  
4  
5     public boolean security(String uId) {  
6         logger.info("指纹支付, 风控校验指纹信息");  
7         return true;  
8     }  
9  
10 }
```

密码

```
1 public class PayCypher implements IPayMode{  
2  
3     protected Logger logger = LoggerFactory.getLogger(PayCypher.class);  
4  
5     public boolean security(String uId) {  
6         logger.info("密码支付, 风控校验环境安全");  
7         return true;  
8     }  
9  
10 }
```

- 在这里实现了三种支付模式(刷脸、指纹、密码)的风控校验，在用户选择不同支付类型的时候，则会进行相应的风控拦截以此保障支付安全。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test_pay() {
3      System.out.println("\r\n模拟测试场景：微信支付、人脸方式。");
4      Pay wxPay = new WxPay(new PayFaceMode());
5      wxPay.transfer("weixin_1092033111", "100000109893", new
6      BigDecimal(100));
7
8      System.out.println("\r\n模拟测试场景：支付宝支付、指纹方式。");
9      Pay zfbPay = new ZfbPay(new PayFingerprintMode());
10     zfbPay.transfer("jlu19dlxo111", "100000109894", new BigDecimal(100));
11 }

```

- 与上面的ifelse实现方式相比，这里的调用方式变得整洁、干净、易使用；`new WxPay(new PayFaceMode())`、`new ZfbPay(new PayFingerprintMode())`
- 外部的使用接口的用户不需要关心具体的实现，只按需选择使用即可。
- 目前以上优化主要针对桥接模式的使用进行重构`if`逻辑部分，关于调用部分可以使用抽象工厂或策略模式配合map结构，将服务配置化。因为这里主要展示桥接模式，所以就不在额外多加代码，避免喧宾夺主。

3.2 测试结果

```

1 模拟测试场景：微信支付、人脸方式。
2 23:14:40.911 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   划账开始。uId: weixin_1092033111 tradeId: 100000109893 amount: 100
3 23:14:40.914 [main] INFO o.i.demo.design.pay.mode.PayCypher - 人脸支付，风
   控校验脸部识别
4 23:14:40.914 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   风控校验。uId: weixin_1092033111 tradeId: 100000109893 security: true
5 23:14:40.915 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟微信渠道支付
   划账成功。uId: weixin_1092033111 tradeId: 100000109893 amount: 100
6
7 模拟测试场景：支付宝支付、指纹方式。
8 23:14:40.915 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
   付划账开始。uId: jlu19dlxo111 tradeId: 100000109894 amount: 100
9 23:14:40.915 [main] INFO o.i.demo.design.pay.mode.PayCypher - 指纹支付，风
   控校验指纹信息
10 23:14:40.915 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
    付风控校验。uId: jlu19dlxo111 tradeId: 100000109894 security: true
11 23:14:40.915 [main] INFO o.i.demo.design.pay.channel.Pay - 模拟支付宝渠道支
    付划账成功。uId: jlu19dlxo111 tradeId: 100000109894 amount: 100
12
13 Process finished with exit code 0

```

- 从测试结果看内容是一样的，但是整体的实现方式有了很大的变化。所以有时候不能只看结果，也要看看过程

六、总结

- 通过模拟微信与支付宝两个支付渠道在不同的支付模式下，`刷脸`、`指纹`、`密码`，的组合从而体现了桥接模式的在这类场景中的合理运用。简化了代码的开发，给后续的需求迭代增加了很好的扩展性。
- 从桥接模式的实现形式来看满足了单一职责和开闭原则，让每一部分内容都很清晰易于维护和拓展，但如果我们是实现的高内聚的代码，那么就会很复杂。所以在选择重构代码的时候，需要考虑好整体的设计，否则选不到合理的设计模式，将会让代码变得难以开发。
- 任何一种设计模式的选择和使用都应该遵顼符合场景为主，不要刻意使用。而且统一场景因为业务的复杂从而可能需要使用到多种设计模式的组合，才能将代码设计的更加合理。但这种经验需要从实际的项目中学习经验，并提不断的运用。

第3节：组合模式

`小朋友才做选择题，成年人我都要`

头几年只要群里一问我该学哪个开发语言，哪个语言最好。群里肯定聊的特别火热，有人支持PHP、有人喊号Java、也有C++和C#。但这几年开始好像大家并不会真的刀枪棍棒、斧钺钩叉般讨论了，大多数时候都是开玩笑的闹一闹。于此同时在整体的互联网开发中很多时候是一些开发语言公用的，共同打造整体的生态圈。而大家选择的方式也是更偏向于不同领域下选择适合的架构，而不是一味地追求某个语言。这可以给很多初学编程的新人一些提议，不要刻意的觉得某个语言好，某个语言不好，只是在适合的场景下选择最需要的。而你要选择的那个语言可以参考招聘网站的需求量和薪资水平决定。

编程开发不是炫技

总会有人喜欢在整体的项目开发中用上点新特性，把自己新学的知识实践试试。不能说这样就是不好，甚至可以说这是一部分很热爱学习的人，喜欢创新，喜欢实践。但编程除了用上新特性外，还需要考虑整体的扩展性、可读性、可维护、易扩展等方面的考虑。就像你家里雇佣了一伙装修师傅，有那么一个小工喜欢炫技搞花活，在家的淋浴下安装了马桶。

即使是写CRUD也应该有设计模式

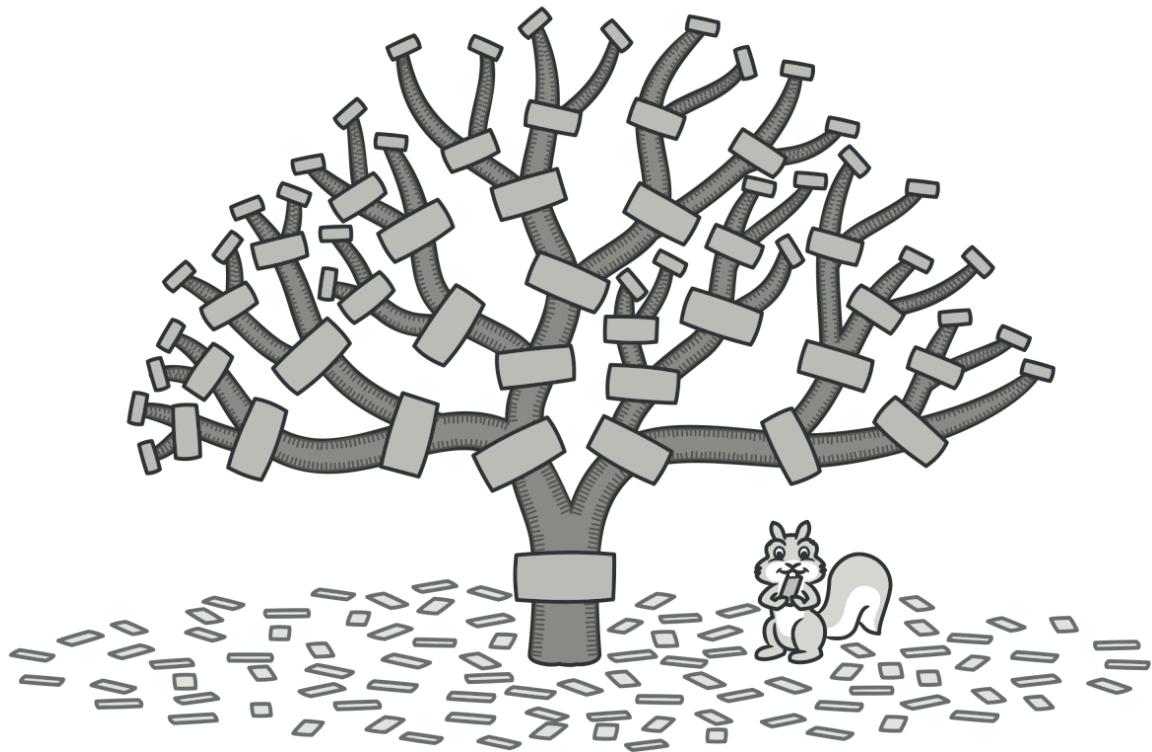
往往很多大需求都是通过增删改查堆出来的，今天要一个需求 if一下，明天加个内容 else 扩展一下。日积月累需求也就越来越大，扩展和维护的成本也就越来越高。往往大部分研发是不具备产品思维和整体业务需求导向的，总以为写好代码完成功能即可。但这样的不考虑扩展性的实现，很难让后续的需求都快速迭代，久而久之就会被陷入恶性循环，每天都有bug要改。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-8-01	使用一坨代码实现业务需求
itstack-demo-design-8-02	通过设计模式优化改造代码，产生对比性从而学习

二、组合模式介绍

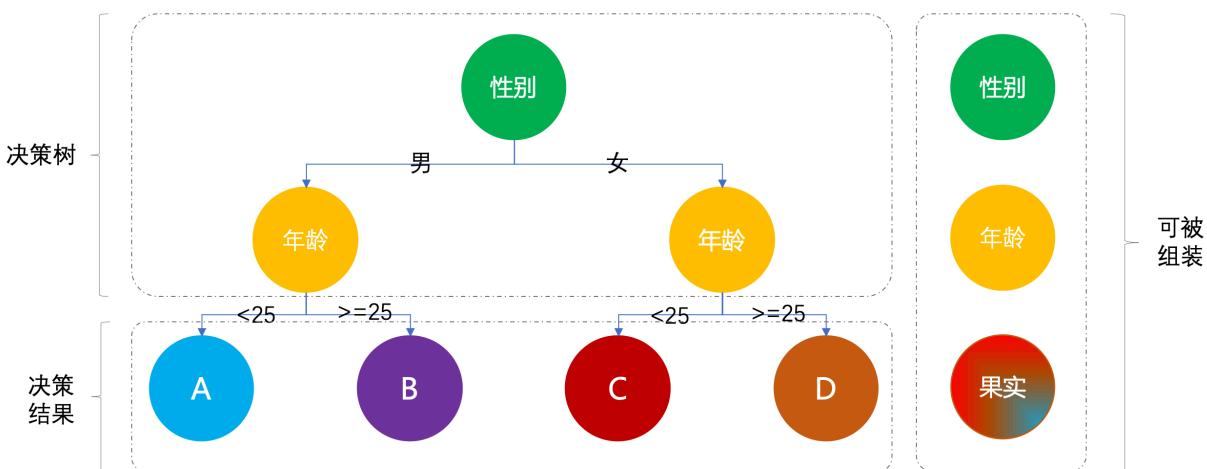


从上图可以看到这有点像螺丝和螺母，通过一堆的链接组织出一棵结构树。而这种通过把相似对象(也可以称作是方法)组合成一组可被调用的设计思路叫做组合模式。

这种方式可以让你的服务组节点进行自由组合对外提供服务，例如你有三个原子校验功能(A: 身份证、B: 银行卡、C: 手机号)服务并对外提供调用使用。有些调用方需要使用AB组合，有些调用方需要使用到CBA组合，还有一些可能只使用三者中的一个。那么这个时候你就可以使用组合模式进行构建服务，对于不同类型的调用方配置不同的组织关系树，而这个树结构你可以配置到数据库中也可以不断的通过图形界面来控制树结构。

所以不同的设计模式用在恰当好处的场景可以让代码逻辑非常清晰并易于扩展，同时也可以减少团队新增人员对项目的学习成本。

三、案例场景模拟



以上是一个非常简化版的营销规则决策树，根据性别、年龄来发放不同类型的优惠券，来刺激消费起到精准用户促活的目的。

虽然一部分小伙伴可能并没有开发过营销场景，但你可能时时刻刻的被营销着。比如你去经常浏览男性喜欢的机械键盘、笔记本电脑、汽车装饰等等，那么久给你推荐此类的优惠券刺激你消费。那么如果你购物不多，或者钱不在自己手里。那么你是否打过车，有一段时间经常有小伙伴喊，为什么同样的距离他就10元，我就15元呢？其实这些都是被营销的案例，一般对于不常使用软件的小伙伴，经常会进行稍微大力度的促活，增加用户粘性。

那么在这里我们就模拟一个类似的决策场景，体现出组合模式在其中起到的重要性。另外，组合模式不只是可以运用于规则决策树，还可以做服务包装将不同的接口进行组合配置，对外提供服务能力，减少开发成本。

四、用一坨坨代码实现

这里我们举一个关于 `if else` 诞生的例子，介绍小姐姐与程序员之间的故事导致的事故。

日期	需求	紧急程度	程序员(话外音)
星期一 早上	猿哥哥，老板说要搞一下营销拉拉量，给男生女生发不同的优惠券，促活消费。	很紧急，下班就要	行吧，也不难，加下判断就上线
星期二 下午	小哥哥，咱们上线后非常好。要让咱们按照年轻、中年、成年，不同年龄加下判断，准确刺激消费。	超紧急，明天就要	也不难，加就加吧
星期三 晚上	喂，小哥哥！睡了吗！老板说咱们这次活动很成功，可以不可以在细分下，把单身、结婚、有娃的都加上不同判断。这样更能刺激用户消费。	贼紧急，最快上线。	已经意识到 <code>if else</code> 越来越多了
星期四 凌晨	哇！小哥哥你们太棒了，上的真快。嘻嘻！有个小请求，需要调整下年龄段，因为现在学生处对象的都比较早，有对象的更容易买某某某东西。要改下值！辛苦辛苦！	老板，在等着呢！	一大片的值要修改，哎！这么多 <code>if else</code> 了
星期五 半夜	歪歪喂！巴巴，坏了，怎么发的优惠券不对了，有客诉了，很多女生都来投诉。你快看看。老板，他...	(一头汗)，哎，值粘错位置了！	终究还是一个人扛下了所有

1. 工程结构

```
1 itstack-demo-design-8-01
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 └── EngineController.java
```

- 公司里要都是这样的程序员绝对省下不少成本，根本不要搭建微服务，一个工程搞定所有业务！
- 但千万不要这么干！ 酒肉穿肠过，佛祖心中留。世人若学我，如同进魔道。

2. 代码实现

```
1 public class EngineController {
2
3     private Logger logger =
4         LoggerFactory.getLogger(EngineController.class);
5
6     public String process(final String userId, final String userSex, final
7     int userAge) {
8
9         logger.info("ifelse实现方式判断用户结果。userId: {} userSex: {}"
10            + "userAge: {}", userId, userSex, userAge);
11
12         if ("man".equals(userSex)) {
13             if (userAge < 25) {
14                 return "果实A";
15             }
16
17         if (userAge >= 25) {
18             return "果实B";
19         }
20
21         if ("woman".equals(userSex)) {
22             if (userAge < 25) {
23                 return "果实C";
24             }
25
26             if (userAge >= 25) {
27                 return "果实D";
28             }
29
30         }
31     }
32
33 }
```

- 除了我们说的扩展性和每次的维护以外，这样的代码实现起来是最快的。而且从样子来看也很适合新人理解。
- 但是 `我劝你别写`，写这样代码不是被扣绩效就是被开除。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test_EngineController() {
3      EngineController engineController = new EngineController();
4      String process = engineController.process("Oli09pLkdjh", "man", 29);
5      logger.info("测试结果: {}", process);
6  }

```

- 这里我们模拟了一个用户ID，并传输性别：man、年龄：29，我们的预期结果是：果实B。实际对应业务就是给 `头秃的程序员发一张枸杞优惠券`。

3.2 测试结果

```

1  22:10:12.891 [main] INFO  o.i.demo.design.EngineController - ifelse实现方式判
2  断用户结果。userId: Oli09pLkdjh userSex: man userAge: 29
3  22:10:12.898 [main] INFO  org.itstack.demo.design.test.ApiTest - 测试结果: 果
4  実B
5
6  Process finished with exit code 0

```

- 从测试结果上看我们的程序运行正常并且符合预期，只不过实现上并不是我们推荐的。接下来我们会采用 `组合模式` 来优化这部分代码。

五、组合模式重构代码

`接下来使用组合模式来进行代码优化，也算是一次很小的重构。`

接下来的重构部分代码改动量相对来说会比较大一些，为了让我们可以把不同类型的决策节点和最终的果实组装成一棵可被运行的决策树，需要做适配设计和工厂方法调用，具体会体现在定义接口以及抽象类和初始化配置决策节点(`性别`、`年龄`)上。建议这部分代码多阅读几次，最好实践下。

1. 工程结构

```

1  itstack-demo-design-8-02
2  └── src
3      ├── main
4      │   └── java
5      │       └── org.itstack.demo.design.domain
6      │           ├── model
7      │           │   └── aggregates
8      │           │       └── TreeRich.java
9      │           └── vo

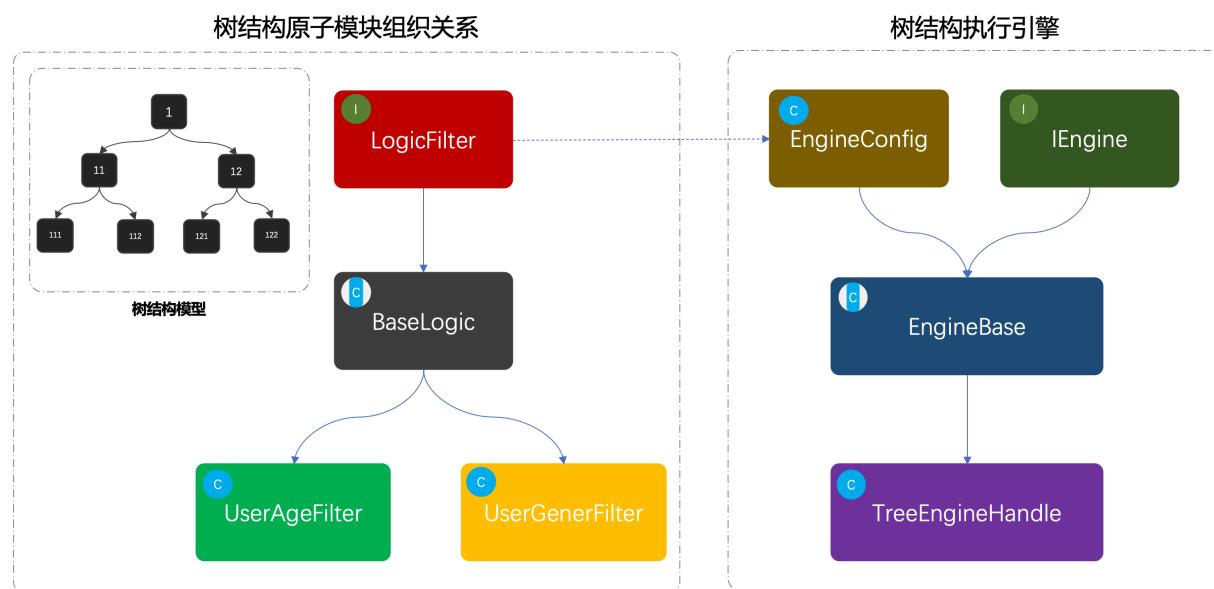
```

```

10 |           └── EngineResult.java
11 |           └── TreeNode.java
12 |           └── TreeNodeLink.java
13 |           └── TreeRoot.java
14 |       └── service
15 |           └── engine
16 |               └── impl
17 |                   └── TreeEngineHandle.java
18 |               └── EngineBase.java
19 |               └── EngineConfig.java
20 |               └── IEngine.java
21 |           └── logic
22 |               └── impl
23 |                   └── LogicFilter.java
24 |                   └── LogicFilter.java
25 |                   └── LogicFilter.java
26 └── test
27     └── java
28         └── org.itstack.demo.design.test
29             └── ApiTest.java

```

组合模式模型结构



- 首先可以看下黑色框框的模拟指导树结构；1、11、12、111、112、121、122，这是一组树结构的ID，并由节点串联组合出一棵关系树。
- 接下来是类图部分，左侧是从 `LogicFilter` 开始定义适配的决策过滤器，`BaseLogic` 是对接口的实现，提供最基本的通用方法。`UserAgeFilter`、`UserGenderFilter`，是两个具体的实现类用于判断年龄和性别。
- 最后则是对这颗可以被组织出来的决策树，进行执行的引擎。同样定义了引擎接口和基础的配置，在配置里面设定了需要的模式决策节点。

```

1 static {
2     logicFilterMap = new ConcurrentHashMap<>();
3     logicFilterMap.put("userAge", new UserAgeFilter());
4     logicFilterMap.put("userGender", new UserGenderFilter());
5 }

```

- 接下来会对每一个类进行细致的讲解，如果感觉没有读懂一定是我作者的表述不够清晰，可以添加我的微信(fustack)与我交流。

2. 代码实现

2.1 基础对象

包路径	类	介绍
model.aggregates	TreeRich	聚合对象，包含组织树信息
model.vo	EngineResult	决策返回对象信息
model.vo	TreeNode	树节点；子叶节点、果实节点
model.vo	TreeNodeLink	树节点链接链路
model.vo	TreeRoot	树根信息

- 以上这部分简单介绍，不包含逻辑只是各项必要属性的 `get/set`，整个源代码可以通过关注微信公众号：`bugstack虫洞栈`，回复源码下载打开链接获取。

2.2 树节点逻辑过滤器接口

```

1 public interface LogicFilter {
2
3     /**
4      * 逻辑决策器
5      *
6      * @param matterValue          决策值
7      * @param treeNodeLineInfoList 决策节点
8      * @return 下一个节点Id
9      */
10     Long filter(String matterValue, List<TreeNodeLink>
11                  treeNodeLineInfoList);
12
13     /**
14      * 获取决策值
15      *
16      * @param decisionMatter 决策物料
17      * @return 决策值
18      */
19     String matterValue(Long treeId, String userId, Map<String, String>
20                       decisionMatter);

```

```
19  
20 }
```

- 这一部分定义了适配的通用接口，逻辑决策器、获取决策值，让每一个提供决策能力的节点都必须实现此接口，保证统一性。

2.3 决策抽象类提供基础服务

```
1 public abstract class BaseLogic implements LogicFilter {  
2  
3     @Override  
4     public Long filter(String matterValue, List<TreeNodeLink>  
treeNodeLinkList) {  
5         for (TreeNodeLink nodeLine : treeNodeLinkList) {  
6             if (decisionLogic(matterValue, nodeLine)) return  
nodeLine.getNodeIdTo();  
7         }  
8         return 0L;  
9     }  
10  
11     @Override  
12     public abstract String matterValue(Long treeId, String userId,  
Map<String, String> decisionMatter);  
13  
14     private boolean decisionLogic(String matterValue, TreeNodeLink  
nodeLink) {  
15         switch (nodeLink.getRuleLimitType()) {  
16             case 1:  
17                 return matterValue.equals(nodeLink.getRuleLimitValue());  
18             case 2:  
19                 return Double.parseDouble(matterValue) >  
Double.parseDouble(nodeLink.getRuleLimitValue());  
20             case 3:  
21                 return Double.parseDouble(matterValue) <  
Double.parseDouble(nodeLink.getRuleLimitValue());  
22             case 4:  
23                 return Double.parseDouble(matterValue) <=  
Double.parseDouble(nodeLink.getRuleLimitValue());  
24             case 5:  
25                 return Double.parseDouble(matterValue) >=  
Double.parseDouble(nodeLink.getRuleLimitValue());  
26             default:  
27                 return false;  
28         }  
29     }  
30 }  
31 }
```

- 在抽象方法中实现了接口方法，同时定义了基本的决策方法；1、2、3、4、5，等于、小于、大

于、小于等于、大于等于 的判断逻辑。

- 同时定义了抽象方法，让每一个实现接口的类都必须按照规则提供 决策值，这个决策值用于做逻辑比对。

2.4 树节点逻辑实现类

年龄节点

```
1 public class UserAgeFilter extends BaseLogic {  
2  
3     @Override  
4     public String matterValue(Long treeId, String userId, Map<String,  
String> decisionMatter) {  
5         return decisionMatter.get("age");  
6     }  
7 }  
8 }
```

性别节点

```
1 public class UserGenderFilter extends BaseLogic {  
2  
3     @Override  
4     public String matterValue(Long treeId, String userId, Map<String,  
String> decisionMatter) {  
5         return decisionMatter.get("gender");  
6     }  
7 }  
8 }
```

- 以上两个决策逻辑的节点获取值的方式都非常简单，只是获取用户的入参即可。实际的业务开发可以从数据库、RPC接口、缓存运算等各种方式获取。

2.5 决策引擎接口定义

```
1 public interface IEngine {  
2  
3     EngineResult process(final Long treeId, final String userId, TreeRich  
treeRich, final Map<String, String> decisionMatter);  
4  
5 }
```

- 对于使用方来说也同样需要定义统一的接口操作，这样的好处非常方便后续拓展出不同类型的决策引擎，也就是可以建造不同的决策工厂。

2.6 决策节点配置

```
1 public class EngineConfig {  
2 }
```

```

3     static Map<String, LogicFilter> logicFilterMap;
4
5     static {
6         logicFilterMap = new ConcurrentHashMap<>();
7         logicFilterMap.put("userAge", new UserAgeFilter());
8         logicFilterMap.put("userGender", new UserGenderFilter());
9     }
10
11    public Map<String, LogicFilter> getLogicFilterMap() {
12        return logicFilterMap;
13    }
14
15    public void setLogicFilterMap(Map<String, LogicFilter> logicFilterMap)
16    {
17        this.logicFilterMap = logicFilterMap;
18    }
19}

```

- 在这里将可提供服务的决策节点配置到 map 结构中，对于这样的 map 结构可以抽取到数据库中，那么就可以非常方便的管理。

2.7 基础决策引擎功能

```

1  public abstract class EngineBase extends EngineConfig implements IEngine {
2
3     private Logger logger = LoggerFactory.getLogger(EngineBase.class);
4
5     @Override
6     public abstract EngineResult process(Long treeId, String userId,
7                                         TreeRich treeRich, Map<String, String> decisionMatter);
8
9     protected TreeNode engineDecisionMaker(TreeRich treeRich, Long treeId,
10                                            String userId, Map<String, String> decisionMatter) {
11         TreeRoot treeRoot = treeRich.getTreeRoot();
12         Map<Long, TreeNode> treeNodeMap = treeRich.getTreeNodeMap();
13         // 规则树根ID
14         Long rootNodeId = treeRoot.getTreeRootNodeId();
15         TreeNode treeNodeInfo = treeNodeMap.get(rootNodeId);
16         //节点类型[NodeType]; 1子叶、2果实
17         while (treeNodeInfo.getNodeType().equals(1)) {
18             String ruleKey = treeNodeInfo.getRuleKey();
19             LogicFilter logicFilter = logicFilterMap.get(ruleKey);
20             String matterValue = logicFilter.matterValue(treeId, userId,
21                 decisionMatter);
22             Long nextNode = logicFilter.filter(matterValue,
23                 treeNodeInfo.getTreeNodeLinkList());
24             treeNodeInfo = treeNodeMap.get(nextNode);
25         }
26     }
27 }

```

```

21         logger.info("决策树引擎=>{} userId: {} treeId: {} treeNode: {}"
22             ruleKey: {} matterValue: {}", treeRoot.getTreeName(), userId, treeId,
23             treeNodeInfo.getTreeNodeId(), ruleKey, matterValue);
24     }
25
26 }

```

- 这里主要提供决策树流程的处理过程，有点像通过链路的关系(性别、年龄)在二叉树中寻找果实节点的过程。
- 同时提供一个抽象方法，执行决策流程的方法供外部去做具体的实现。

2.8 决策引擎的实现

```

1 public class TreeEngineHandle extends EngineBase {
2
3     @Override
4     public EngineResult process(Long treeId, String userId, TreeRich
5         treeRich, Map<String, String> decisionMatter) {
6         // 决策流程
7         TreeNode treeNode = engineDecisionMaker(treeRich, treeId, userId,
8             decisionMatter);
9         // 决策结果
10        return new EngineResult(userId, treeId, treeNode.getTreeNodeId(),
11            treeNode.getNodeValue());
12    }
13 }

```

- 这里对于决策引擎的实现就非常简单了，通过传递进来的必要信息：决策树信息、决策物料值，来做具体的树形结构决策。

3. 测试验证

3.1 组装树关系

```

1 @Before
2 public void init() {
3     // 节点: 1
4     TreeNode treeNode_01 = new TreeNode();
5     treeNode_01.setTreeId(10001L);
6     treeNode_01.setTreeNodeId(1L);
7     treeNode_01.setNodeType(1);
8     treeNode_01.setNodeValue(null);
9     treeNode_01.setRuleKey("userGender");
10    treeNode_01.setRuleDesc("用户性别[男/女]");
11    // 链接: 1->11
12    TreeNodeLink treeNodeLink_11 = new TreeNodeLink();

```

```
13     treeNodeLink_11.setNodeIdFrom(1L);
14     treeNodeLink_11.setNodeIdTo(11L);
15     treeNodeLink_11.setRuleLimitType(1);
16     treeNodeLink_11.setRuleLimitValue("man");
17     // 链接: 1->12
18     TreeNodeLink treeNodeLink_12 = new TreeNodeLink();
19     treeNodeLink_12.setNodeIdTo(1L);
20     treeNodeLink_12.setNodeIdTo(12L);
21     treeNodeLink_12.setRuleLimitType(1);
22     treeNodeLink_12.setRuleLimitValue("woman");
23     List<TreeNodeLink> treeNodeLinkList_1 = new ArrayList<>();
24     treeNodeLinkList_1.add(treeNodeLink_11);
25     treeNodeLinkList_1.add(treeNodeLink_12);
26     treeNode_01.setTreeNodeLinkList(treeNodeLinkList_1);
27     // 节点: 11
28     TreeNode treeNode_11 = new TreeNode();
29     treeNode_11.setTreeId(10001L);
30     treeNode_11.setTreeNodeId(11L);
31     treeNode_11.setNodeType(1);
32     treeNode_11.setNodeValue(null);
33     treeNode_11.setRuleKey("userAge");
34     treeNode_11.setRuleDesc("用户年龄");
35     // 链接: 11->111
36     TreeNodeLink treeNodeLink_111 = new TreeNodeLink();
37     treeNodeLink_111.setNodeIdFrom(11L);
38     treeNodeLink_111.setNodeIdTo(111L);
39     treeNodeLink_111.setRuleLimitType(3);
40     treeNodeLink_111.setRuleLimitValue("25");
41     // 链接: 11->112
42     TreeNodeLink treeNodeLink_112 = new TreeNodeLink();
43     treeNodeLink_112.setNodeIdFrom(11L);
44     treeNodeLink_112.setNodeIdTo(112L);
45     treeNodeLink_112.setRuleLimitType(5);
46     treeNodeLink_112.setRuleLimitValue("25");
47     List<TreeNodeLink> treeNodeLinkList_11 = new ArrayList<>();
48     treeNodeLinkList_11.add(treeNodeLink_111);
49     treeNodeLinkList_11.add(treeNodeLink_112);
50     treeNode_11.setTreeNodeLinkList(treeNodeLinkList_11);
51     // 节点: 12
52     TreeNode treeNode_12 = new TreeNode();
53     treeNode_12.setTreeId(10001L);
54     treeNode_12.setTreeNodeId(12L);
55     treeNode_12.setNodeType(1);
56     treeNode_12.setNodeValue(null);
57     treeNode_12.setRuleKey("userAge");
58     treeNode_12.setRuleDesc("用户年龄");
59     // 链接: 12->121
60     TreeNodeLink treeNodeLink_121 = new TreeNodeLink();
61     treeNodeLink_121.setNodeIdFrom(12L);
```

```
62     treeNodeLink_121.setNodeIdTo(121L);
63     treeNodeLink_121.setRuleLimitType(3);
64     treeNodeLink_121.setRuleLimitValue("25");
65     // 链接: 12->122
66     TreeNodeLink treeNodeLink_122 = new TreeNodeLink();
67     treeNodeLink_122.setNodeIdFrom(12L);
68     treeNodeLink_122.setNodeIdTo(122L);
69     treeNodeLink_122.setRuleLimitType(5);
70     treeNodeLink_122.setRuleLimitValue("25");
71     List<TreeNodeLink> treeNodeLinkList_12 = new ArrayList<>();
72     treeNodeLinkList_12.add(treeNodeLink_121);
73     treeNodeLinkList_12.add(treeNodeLink_122);
74     treeNode_12.setTreeNodeLinkList(treeNodeLinkList_12);
75     // 节点: 111
76     TreeNode treeNode_111 = new TreeNode();
77     treeNode_111.setTreeId(10001L);
78     treeNode_111.setTreeNodeId(111L);
79     treeNode_111.setNodeType(2);
80     treeNode_111.setNodeValue("果实A");
81     // 节点: 112
82     TreeNode treeNode_112 = new TreeNode();
83     treeNode_112.setTreeId(10001L);
84     treeNode_112.setTreeNodeId(112L);
85     treeNode_112.setNodeType(2);
86     treeNode_112.setNodeValue("果实B");
87     // 节点: 121
88     TreeNode treeNode_121 = new TreeNode();
89     treeNode_121.setTreeId(10001L);
90     treeNode_121.setTreeNodeId(121L);
91     treeNode_121.setNodeType(2);
92     treeNode_121.setNodeValue("果实C");
93     // 节点: 122
94     TreeNode treeNode_122 = new TreeNode();
95     treeNode_122.setTreeId(10001L);
96     treeNode_122.setTreeNodeId(122L);
97     treeNode_122.setNodeType(2);
98     treeNode_122.setNodeValue("果实D");
99     // 树根
100    TreeRoot treeRoot = new TreeRoot();
101    treeRoot.setTreeId(10001L);
102    treeRoot.setTreeRootNodeId(1L);
103    treeRoot.setTreeName("规则决策树");
104    Map<Long, TreeNode> treeNodeMap = new HashMap<>();
105    treeNodeMap.put(1L, treeNode_01);
106    treeNodeMap.put(11L, treeNode_11);
107    treeNodeMap.put(12L, treeNode_12);
108    treeNodeMap.put(111L, treeNode_111);
109    treeNodeMap.put(112L, treeNode_112);
110    treeNodeMap.put(121L, treeNode_121);
```

```
111     treeNodeMap.put(122L, treeNode_122);
112     treeRich = new TreeRich(treeRoot, treeNodeMap);
113 }
```

```
1:{  
    "nodeType":1,  
    "ruleDesc":"用户性别[男/女]",  
    "ruleKey":"userGender",  
    "treeId":10001,  
    "treeNodeId":1,  
    "treeNodeLinkList": [  
        {  
            "nodeIdFrom":1,  
            "nodeIdTo":11,  
            "ruleLimitType":1,  
            "ruleLimitValue":"man"  
        },  
        {  
            "nodeIdTo":12,  
            "ruleLimitType":1,  
            "ruleLimitValue":"woman"  
        },  
    ],  
    "ruleType":1  
},  
11:{  
    "nodeType":1,  
    "ruleDesc":"用户年龄",  
    "ruleKey":"userGender",  
    "treeId":10001,  
    "treeNodeId":11,  
    "treeNodeLinkList": [  
        {  
            "nodeIdFrom":11,  
            "nodeIdTo":111,  
            "ruleLimitType":3,  
            "ruleLimitValue":"25"  
        }  
    ]  
}
```

对树形结构组织关系的部分截取

1. nodeFrom: 从哪开始的节点
2. nodeTo: 节点要指向到哪
3. ruleLimitType: 节点的比对方式
4. ruleLimitValue: 节点的比对值

- **重要**, 这一部分是组合模式非常重要的使用, 在我们已经建造好的决策树关系下, 可以创建出树的各个节点, 以及对节点间使用链路进行串联。
- 及时后续你需要做任何业务的扩展都可以在里面添加相应的节点, 并做动态化的配置。
- 关于这部分手动组合的方式可以提取到数据库中, 那么也就可以扩展到图形界面的进行配置操作。

3.2 编写测试类

```

1  @Test
2  public void test_tree() {
3      logger.info("决策树组合结构信息: \r\n" + JSON.toJSONString(treeRich));
4
5      IEngine treeEngineHandle = new TreeEngineHandle();
6      Map<String, String> decisionMatter = new HashMap<>();
7      decisionMatter.put("gender", "man");
8      decisionMatter.put("age", "29");
9
10     EngineResult result = treeEngineHandle.process(10001L, "Oli09pLkdjh",
11             treeRich, decisionMatter);
12
13     logger.info("测试结果: {}", JSON.toJSONString(result));
14 }

```

- 在这里提供了调用的通过组织模式创建出来的流程决策树，调用的时候传入了决策树的ID，那么如果是业务开发中就可以方便的解耦决策树与业务的绑定关系，按需传入决策树ID即可。
- 此外入参我们还提供了需要处理；男 (man)、年龄 (29岁)，的参数信息。

3.3 测试结果

```

1  23:35:05.711 [main] INFO o.i.d.d.d.service.engine.EngineBase - 决策树引擎=>
规则决策树 userId: Oli09pLkdjh treeId: 10001 treeNode: 11 ruleKey: userGender
matterValue: man
2  23:35:05.712 [main] INFO o.i.d.d.d.service.engine.EngineBase - 决策树引擎=>
规则决策树 userId: Oli09pLkdjh treeId: 10001 treeNode: 112 ruleKey: userAge
matterValue: 29
3  23:35:05.715 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结
果: {"nodeId":112,"nodeValue":"果实
B","success":true,"treeId":10001,"userId":"Oli09pLkdjh"}
4
5  Process finished with exit code 0

```

- 从测试结果上看这与我们使用 `if else` 是一样的，但是目前这与的组合模式设计下，就非常方便后续的拓展和修改。
- 整体的组织关系框架以及调用决策流程已经搭建完成，如果阅读到此没有完全理解，可以下载代码观察结构并运行调试。

六、总结

- 从以上的决策树场景来看，组合模式的主要解决的是一系列简单逻辑节点或者扩展的复杂逻辑节点在不同结构的组织下，对于外部的调用是仍然可以非常简单的。
- 这部分设计模式保证了开闭原则，无需更改模型结构你就可以提供新的逻辑节点的使用并配合组织出新的关系树。但如果是一些功能差异化非常大的接口进行包装就会变得比较困难，但也不是不能很好的处理，只不过需要做一些适配和特定化的开发。
- 很多时候因为你的极致追求和稍有倔强的工匠精神，即使在面对同样的业务需求，你能完成出最好的代码结构和最易于扩展的技术架构。不要被远不能给你指导提升能力的影响到放弃自己的追求！

第4节：装饰器模式

对于代码你有编程感觉吗

很多人写代码往往是没有编程感觉的，也就是除了可以把功能按照固定的流程编写出流水式的代码外，很难去思考整套功能服务的扩展性和可维护性。尤其是在一些较大型的功能搭建上，比较缺失一些驾驭能力，从而导致最终的代码相对来说不能做到尽善尽美。

江洋大盗与江洋大偷

两个本想描述一样的意思的词，只因一字只差就让人觉得一个是好牛，一个好搞笑。往往我们去开发编程写代码时也经常将一些不恰当的用法用于业务需求实现中，但却不能意识到。一方面是由于编码不多缺少较大型项目的实践，另一方面是不思进取的总在以完成需求为目标缺少精益求精的工匠精神。

书从来不是看的而是用的

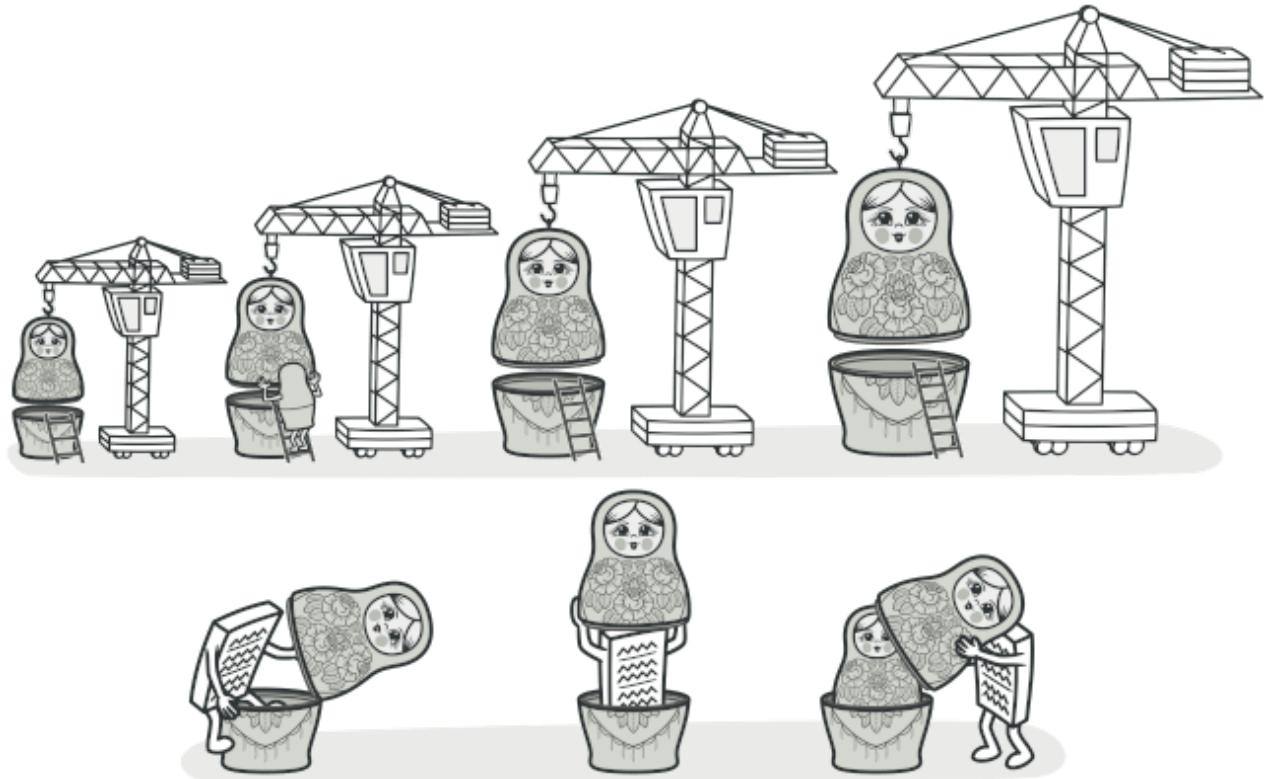
在这个学习资料几乎爆炸的时代，甚至你可以轻易就获取几个T的视频，小手轻轻一点就收藏一堆文章，但却很少去看。学习的过程从不只是简单的看一遍就可以，对于一些实操性的技术书籍，如果真的希望学到知识，那么一定是把这本书用起来而绝对不是看起来。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-9-00	场景模拟工程；模拟单点登录类
itstack-demo-design-9-01	使用一坨代码实现业务需求
itstack-demo-design-9-02	通过设计模式优化改造代码，产生对比性从而学习

二、装饰器模式介绍



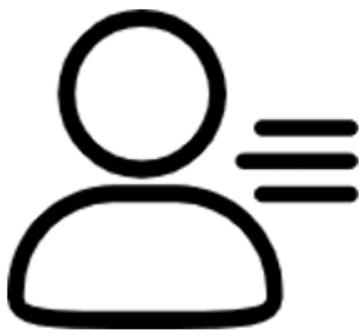
初看上图感觉装饰器模式有点像俄罗斯套娃、某众汽车，而装饰器的核心就是再不改原有类的基础上给类新增功能。**不改变原有类**，可能有的小伙伴会想到继承、AOP切面，当然这些方式都可以实现，但是使用装饰器模式会是另外一种思路更为灵活，可以避免继承导致的子类过多，也可以避免AOP带来的复杂性。

你熟悉的场景很多用到装饰器模式

```
new BufferedReader(new FileReader(""));
```

这段代码你是否熟悉，相信学习java开发到字节流、字符流、文件流的内容时都见到了这样的代码，一层嵌套一层，一层嵌套一层，字节流转字符流等等，而这样方式的使用就是装饰器模式的一种体现。

三、案例场景模拟



SSO

新增方法拦截

查询用户信息

查询商品列表

创建营销活动

修改商品金额

修改活动状态

在本案例中我们模拟一个单点登录功能扩充的场景

一般在业务开发的初期，往往内部的ERP使用只需要判断账户验证即可，验证通过后即可访问ERP的所有资源。但随着业务的不断发展，团队里开始出现专门的运营人员、营销人员、数据人员，每个人员对于ERP的使用需求不同，有些需要创建活动，有些只是查看数据。同时为了保证数据的安全性，不会让每个用户都有最高的权限。

那么以往使用的 `sso` 是一个组件化通用的服务，不能在里面添加需要的用户访问验证功能。这个时候我们就可以使用装饰器模式，扩充原有的单点登录服务。但同时也保证原有功能不受破坏，可以继续使用。

1. 场景模拟工程

```
1 itstack-demo-design-9-00
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── HandlerInterceptor.java
7                 └── SsoInterceptor.java
```

- 这里模拟的是spring中的类：`HandlerInterceptor`，实现接口功能`SsoInterceptor`模拟的单点登录拦截服务。
- 为了避免引入太多spring的内容影响对设计模式的阅读，这里使用了同名的类和方法，尽可能减少外部的依赖。

2. 场景简述

2.1 模拟Spring的HandlerInterceptor

```
1 public interface HandlerInterceptor {
2
3     boolean preHandle(String request, String response, Object handler);
4
5 }
```

- 实际的单点登录开发会基于：`org.springframework.web.servlet.HandlerInterceptor` 实现。

2.2 模拟单点登录功能

```
1 public class SsoInterceptor implements HandlerInterceptor{
2
3     public boolean preHandle(String request, String response, Object
4 handler) {
5         // 模拟获取cookie
6         String ticket = request.substring(1, 8);
7         // 模拟校验
8         return ticket.equals("success");
9     }
10 }
```

- 这里的模拟实现非常简单只是截取字符串，实际使用需要从`HttpServletRequest request`对象中获取`cookie`信息，解析`ticket`值做校验。
- 在返回的里面也非常简单，只要获取到了`success`就认为是允许登录。

四、用一坨坨代码实现

此场景大多数实现的方式都会采用继承类

继承类的实现方式也是一个比较通用的方式，通过继承后重写方法，并发将自己的逻辑覆盖进去。如果是一些简单的场景且不需要不断维护和扩展的，此类实现并不会有什么，也不会导致子类过多。

1. 工程结构

```
1 itstack-demo-design-9-01
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ LoginSsoDecorator.java
```

- 以上工程结构非常简单，只是通过 `LoginSsoDecorator` 继承 `SsoInterceptor`，重写方法功能。

2. 代码实现

```
1 public class LoginSsoDecorator extends SsoInterceptor {
2
3     private static Map<String, String> authMap = new
4     ConcurrentHashMap<String, String>();
5
6     static {
7         authMap.put("huahua", "queryUserInfo");
8         authMap.put("doudou", "queryUserInfo");
9     }
10
11    @Override
12    public boolean preHandle(String request, String response, Object
13 handler) {
14        // 模拟获取cookie
15        String ticket = request.substring(1, 8);
16        // 模拟校验
17        boolean success = ticket.equals("success");
18
19        if (!success) return false;
20
21        String userId = request.substring(9);
22        String method = authMap.get(userId);
23
24        // 模拟方法校验
25        return "queryUserInfo".equals(method);
26    }
27}
```

- 以上这部分通过继承重写方法，将个人可访问哪些方法的功能添加到方法中。
- 以上看着代码还算比较清晰，但如果是比较复杂的业务流程代码，就会很混乱。

3. 测试验证

3.1 编写测试类

```
1  @Test
2  public void test_LoginSsoDecorator() {
3      LoginSsoDecorator ssoDecorator = new LoginSsoDecorator();
4      String request = "1successhuahua";
5      boolean success = ssoDecorator.preHandle(request, "ewcdqwt40liuiu",
6          "t");
7      System.out.println("登录校验: " + request + (success ? " 放行" : " 拦
截"));
8  }
```

- 这里模拟的相当于登录过程中的校验操作，判断用户是否可登录以及是否可访问方法。

3.2 测试结果

```
1  登录校验: 1successhuahua 拦截
2
3  Process finished with exit code 0
```

- 从测试结果来看满足我们的预期，已经做了拦截。如果你在学习的过程中，可以尝试模拟单点登录并继承扩展功能。

五、装饰器模式重构代码

接下来使用装饰器模式来进行代码优化，也算是一次很小的重构。

装饰器主要解决的是直接继承下因功能的不断横向扩展导致子类膨胀的问题，而是用装饰器模式后就会比直接继承显得更加灵活同时这样也就不再需要考虑子类的维护。

在装饰器模式中有四个比较重要点抽象出来的点；

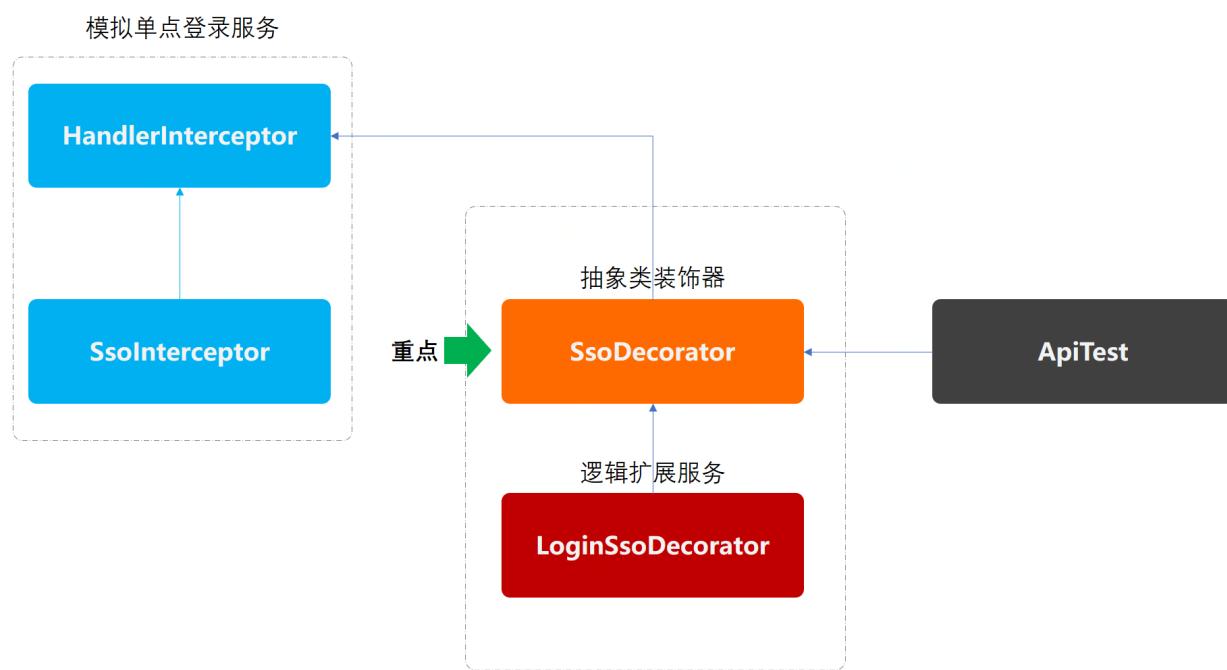
- 抽象构件角色(Component) - 定义抽象接口
- 具体构件角色(ConcreteComponent) - 实现抽象接口，可以是一组
- 装饰角色(Decorator) - 定义抽象类并继承接口中的方法，保证一致性
- 具体装饰角色(ConcreteDecorator) - 扩展装饰具体的实现逻辑

通过以上这四项来实现装饰器模式，主要核心内容会体现在抽象类的定义和实现上。

1. 工程结构

```
1  itstack-demo-design-9-02
2  └── src
3      └── main
4          └── java
5              └── org.itstack.demo.design
6                  ├── LoginSsoDecorator.java
7                  └── SsoDecorator.java
```

装饰器模式模型结构



- 以上是一个装饰器实现的类图结构，重点的类是 `SsoDecorator`，这个类是一个抽象类主要完成了对接口 `HandlerInterceptor` 继承。
- 当装饰角色继承接口后会提供构造函数，入参就是继承的接口实现类即可，这样就可以很方便的扩展出不同功能组件。

2. 代码实现

2.1 抽象类装饰角色

```
1 public abstract class SsoDecorator implements HandlerInterceptor {
2
3     private HandlerInterceptor handlerInterceptor;
4
5     private SsoDecorator(){}
6
7     public SsoDecorator(HandlerInterceptor handlerInterceptor) {
8         this.handlerInterceptor = handlerInterceptor;
9     }
10
11     public boolean preHandle(String request, String response, Object
12         handler) {
13         return handlerInterceptor.preHandle(request, response, handler);
14     }
15 }
```

- 在装饰类中有两个重点的地方是；1)继承了处理接口、2)提供了构造函数、3)覆盖了方法 `preHandle`。
- 以上三个点是装饰器模式的核心处理部分，这样可以踢掉对子类继承的方式实现逻辑功能扩展。

2.2 装饰角色逻辑实现

```
1 public class LoginSsoDecorator extends SsoDecorator {
2
3     private Logger logger =
4         LoggerFactory.getLogger(LoginSsoDecorator.class);
5
6     private static Map<String, String> authMap = new
7         ConcurrentHashMap<String, String>();
8
9     static {
10        authMap.put("huahua", "queryUserInfo");
11        authMap.put("doudou", "queryUserInfo");
12    }
13
14    public LoginSsoDecorator(HandlerInterceptor handlerInterceptor) {
15        super(handlerInterceptor);
16    }
17
18    @Override
19    public boolean preHandle(String request, String response, Object
20        handler) {
21        boolean success = super.preHandle(request, response, handler);
22        if (!success) return false;
23        String userId = request.substring(8);
24        String method = authMap.get(userId);
25        logger.info("模拟单点登录方法访问拦截校验: {} {}", userId, method);
26        // 模拟方法校验
27        return "queryUserInfo".equals(method);
28    }
29}
```

- 在具体的装饰类实现中，继承了装饰类 `SsoDecorator`，那么现在就可以扩展方法：`preHandle`
- 在 `preHandle` 的实现中可以看到，这里只关心扩展部分的功能，同时不会影响原有类的核心服务，也不会因为使用继承方式而导致的多余子类，增加了整体的灵活性。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test_LoginSsoDecorator() {
3      LoginSsoDecorator ssoDecorator = new LoginSsoDecorator(new
4          SsoInterceptor());
5      String request = "1successhuahua";
6      boolean success = ssoDecorator.preHandle(request, "ewcdqwt40liuiu",
7          "t");
8      System.out.println("登录校验: " + request + (success ? " 放行" : " 拦
9      截"));
10 }

```

- 这里测试了对装饰器模式的使用，通过透传原有单点登录类 `new SsoInterceptor()`，传递给装饰器，让装饰器可以执行扩充的功能。
- 同时对于传递者和装饰器都可以是多组的，在一些实际的业务开发中，往往也是由于太多类型的子类实现而导致不易于维护，从而使用装饰器模式替代。

3.2 测试结果

```

1  23:50:50.796 [main] INFO  o.i.demo.design.LoginSsoDecorator - 模拟单点登录方法
2  访问拦截校验: huahua queryUserInfo
3  登录校验: 1successhuahua 放行
4  Process finished with exit code 0

```

- 结果符合预期，扩展了对方法拦截的校验性。
- 如果你在学习的过程中有用到过单点登陆，那么可以适当在里面进行扩展装饰器模式进行学习使用。
- 另外，还有一种场景也可以使用装饰器。例如；你之前使用某个实现某个接口接收单个消息，但由于外部的升级变为发送 `list` 集合消息，但你又不希望所有的代码类都去修改这部分逻辑。那么可以使用装饰器模式进行适配 `list` 集合，给使用者依然是 `for` 循环后的单个消息。

六、总结

- 使用装饰器模式满足单一职责原则，你可以在自己的装饰类中完成功能逻辑的扩展，而不影响主类，同时可以按需在运行时添加和删除这部分逻辑。另外装饰器模式与继承父类重写方法，在某些时候需要按需选择，并不一定某一个就是最好。
- 装饰器实现的重点是对抽象类继承接口方式的使用，同时设定被继承的接口可以通过构造函数传递其实现类，由此增加扩展性并重写方法里可以实现此部分父类实现的功能。
- 就像夏天热你穿短裤，冬天冷你穿棉裤，雨天挨浇你穿雨衣一样，你的根本本身没有被改变，而你的需求却被不同的装饰而实现。生活中往往比比皆是设计，当你可以融合这部分活灵活现的例子到代码实现中，往往会创造出更加优雅的实现方式。

第5节：外观模式

你感受到的容易，一定有人为你承担不容易

这句话更像是描述生活的，许许多多的磕磕绊绊总有人为你提供躲雨的屋檐和避风的港湾。其实编程开发的团队中也一样有人只负责CRUD中的简单调用，去使用团队中高级程序员开发出来的核心服务和接口。这样的编程开发对于初期刚进入程序员行业的小伙伴来说锻炼锻炼还是不错的，但随着开发的日子越来越久一直做这样的事情就很难得到成长，也想努力的去做一些更有难度的承担，以此来增强个人的技术能力。

没有最好的编程语言，语言只是工具

刀枪棍棒、斧钺钩叉、包子油条、盒子麻花，是语言。五郎八卦棍、十二路弹腿、洪家铁线拳，是设计。记得叶问里有一句台词是：金山找：今天我北方拳术，输给你南方拳术了。叶问：你错了，不是南北拳的问题，是你的问题。所以当你编程开发写久了，就不会再特别在意用的语言，而是为目标服务，用最好的设计能力也就是编程的智慧做出做最完美的服务。这也就是编程人员的价值所在！

设计与反设计以及过渡设计

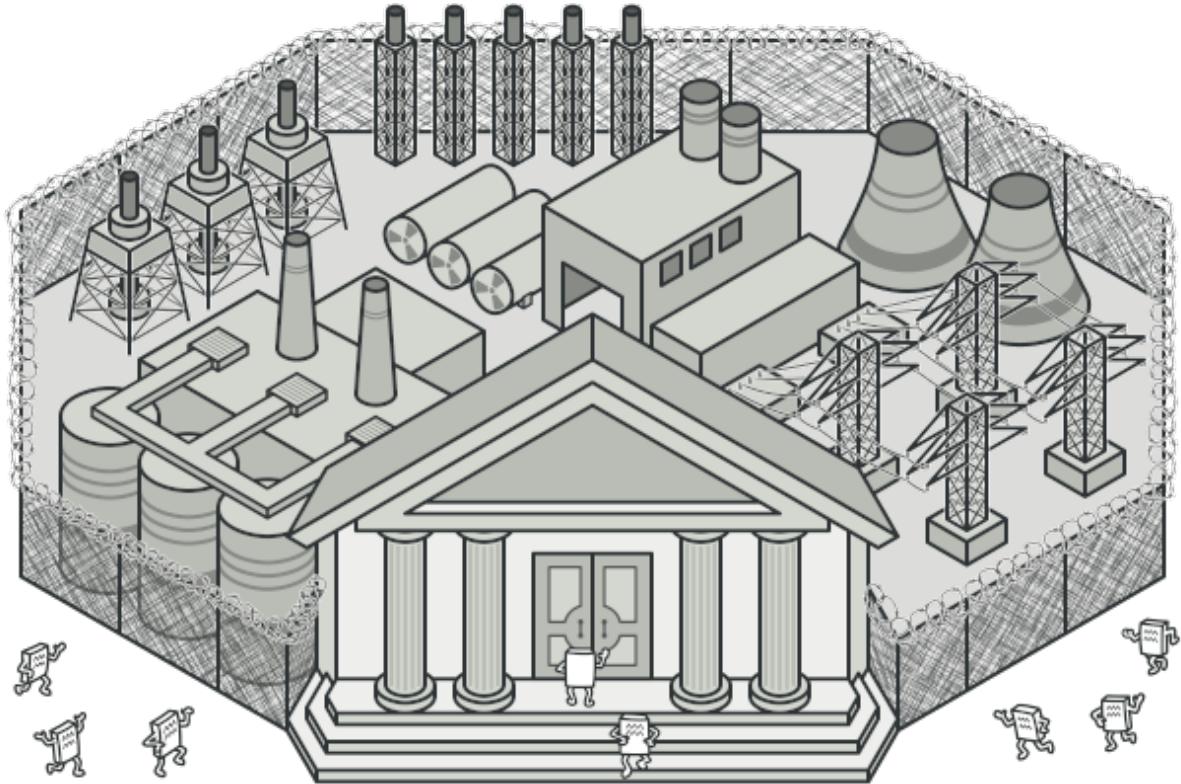
设计模式是解决程序中不合理、不易于扩展、不易于维护的问题，也是干掉大部分 `if else` 的利器，在我们常用的框架中基本都会用到大量的设计模式来构建组件，这样也能方便框架的升级和功能的扩展。但！如果不能合理的设计以及乱用设计模式，会导致整个编程变得更加复杂难维护，也就是我们常说的；反设计、过渡设计。而这部分设计能力也是从实践的项目中获取的经验，不断的改造优化摸索出的最合理的方式，应对当前的服务体量。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. SpringBoot 2.1.2.RELEASE
4. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-10-00	场景模拟工程；模拟一个提供接口服务的SpringBoot工程
itstack-demo-design-10-01	使用一坨代码实现业务需求
itstack-demo-design-10-02	通过设计模式开发为中间件，包装通用型核心逻辑

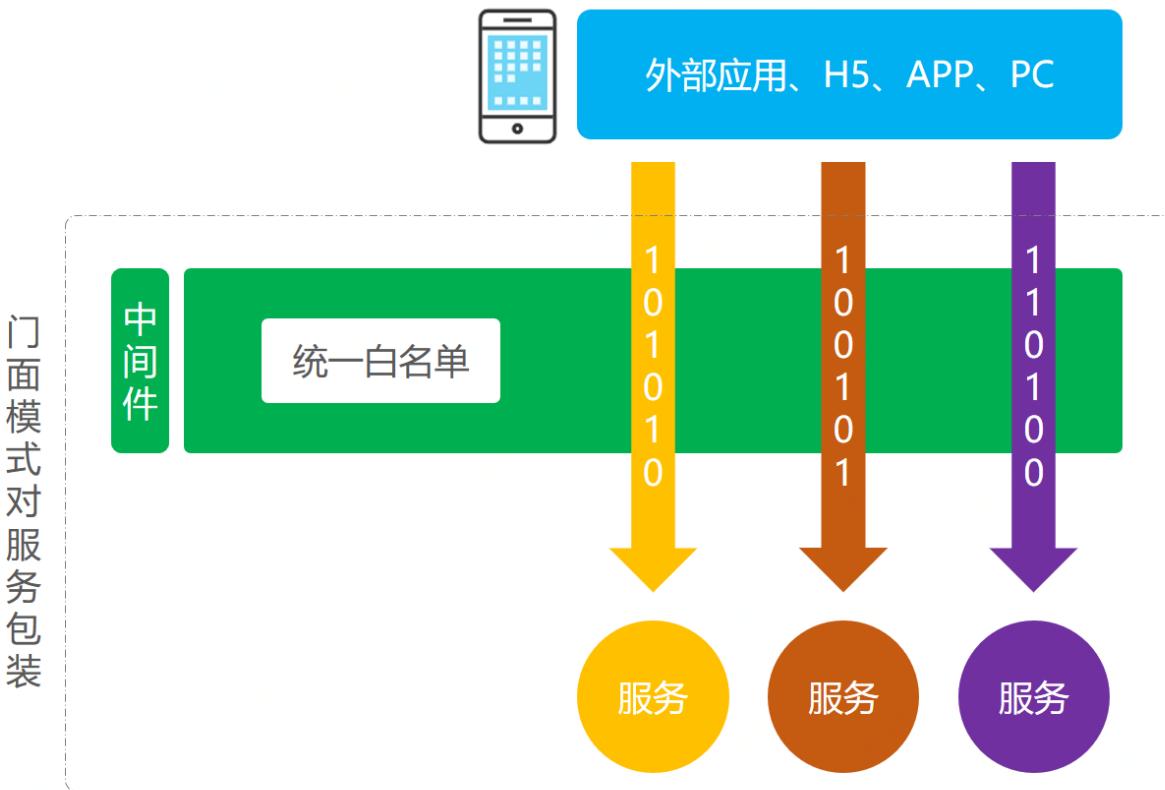
二、外观模式介绍



外观模式也叫门面模式，主要解决的是降低调用方的使用接口的复杂逻辑组合。这样调用方与实际的接口提供方提供了一个中间层，用于包装逻辑提供API接口。有些时候外观模式也被用在中间件层，对服务中的通用性复杂逻辑进行中间件层包装，让使用方可以只关心业务开发。

那么这样的模式在我们的所见产品功能中也经常遇到，就像几年前我们注册一个网站时候往往要添加很多信息，包括：姓名、昵称、手机号、QQ、邮箱、住址、单身等等，但现在注册成为一个网站的用户只需要一步即可，无论是手机号还是微信也都提供了这样的登录服务。而对于服务端应用开发来说以前是提供了一个整套的接口，现在注册的时候并没有这些信息，那么服务端就需要进行接口包装，在前端调用注册的时候服务端获取相应的用户信息(从各个渠道)，如果获取不到会让用户后续进行补全(营销补全信息给奖励)，以此来拉动用户的注册量和活跃度。

三、案例场景模拟



在本案例中我们模拟一个将所有服务接口添加白名单的场景

在项目不断壮大发展的路上，每一次发版上线都需要进行测试，而这部分测试验证一般会进行白名单开量或者切量的方式进行验证。那么如果在每一个接口中都添加这样的逻辑，就会非常麻烦且不易维护。另外这是一类具备通用逻辑的共性需求，非常适合开发成组件，以此来治理服务，让研发人员更多的关心业务功能开发。

一般情况下对于外观模式的使用通常是用在复杂或多个接口进行包装统一对外提供服务上，此种使用方式也相对简单在我们平常的业务开发中也是最常用的。你可能经常听到把这两个接口包装一下，但在本例子中我们把这种设计思路放到中间件层，让服务变得可以统一控制。

1. 场景模拟工程

```

1  itstack-demo-design-10-00
2  └── src
3      ├── main
4          ├── java
5          │   └── org.itstack.demo.design
6          │       ├── domain
7          │       │   └── UserInfo.java
8          │       └── web
9          │           └── HelloWorldController.java
10         └── HelloWorldApplication.java
11     └── resources
12         └── application.yml
13 └── test
14     └── java
15         └── org.itstack.demo.test

```

- 这是一个 SpringBoot 的 HelloWorld 工程，在工程中提供了查询用户信息的接口 `HelloWorldController.queryUserInfo`，为后续扩展此接口的白名单过滤做准备。

2. 场景简述

2.1 定义基础查询接口

```

1  @RestController
2  public class HelloWorldController {
3
4      @Value("${server.port}")
5      private int port;
6
7      /**
8       * key: 需要从入参取值的属性字段, 如果是对象则从对象中取值, 如果是单个值则直接使用
9       * returnJson: 预设拦截时返回值, 是返回对象的Json
10      *
11      * http://localhost:8080/api/queryUserInfo?userId=1001
12      * http://localhost:8080/api/queryUserInfo?userId=小团团
13      */
14      @RequestMapping(path = "/api/queryUserInfo", method =
RequestMethod.GET)
15      public UserInfo queryUserInfo(@RequestParam String userId) {
16          return new UserInfo("虫虫:" + userId, 19, "天津市南开区旮旯胡同100
号");
17      }
18
19  }
```

- 这里提供了一个基本的查询服务，通过入参 `userId`，查询用户信息。后续就需要在这里扩展白名单，只有指定用户才可以查询，其他用户不能查询。

2.2 设置Application启动类

```

1  @SpringBootApplication
2  @Configuration
3  public class HelloWorldApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(HelloWorldApplication.class, args);
7      }
8
9  }
```

- 这里是通用的 SpringBoot 启动类。需要添加的是一个配置注解 `@Configuration`，为了后续可以读取白名单配置。

四、用一坨坨代码实现

一般对于此种场景最简单的做法就是直接修改代码

累加 `if` 块几乎是实现需求最快也是最慢的方式，快是修改当前内容很快，慢是如果同类的内容几百个也都需要如此修改扩展和维护会越来越慢。

1. 工程结构

```
1 | itstack-demo-design-10-01
2 | └─ src
3 |   └─ main
4 |     └─ java
5 |       └─ org.itstack.demo.design
6 |         └─ HelloWorldController.java
```

- 以上的实现是模拟一个Api接口类，在里面添加白名单功能，但类似此类的接口会有很多都需要修改，所以这也是不推荐使用此种方式的重要原因。

2. 代码实现

```
1 | public class HelloWorldController {
2 |
3 |     public UserInfo queryUserInfo(@RequestParam String userId) {
4 |
5 |         // 做白名单拦截
6 |         List<String> userList = new ArrayList<String>();
7 |         userList.add("1001");
8 |         userList.add("aaaa");
9 |         userList.add("ccc");
10 |        if (!userList.contains(userId)) {
11 |            return new UserInfo("1111", "非白名单可访问用户拦截!");
12 |        }
13 |
14 |        return new UserInfo("虫虫:" + userId, 19, "天津市南开区旮旯胡同100号");
15 |    }
16 |
17 |}
```

- 在这里白名单的代码占据了一大块，但它又不是业务中的逻辑，而是因为我们上线过程中需要做的开量前测试验证。
- 如果你日常对待此类需求经常是这样开发，那么可以按照此设计模式进行优化你的处理方式，让后续的扩展和摘除更加容易。

五、外观模式重构代码

接下来使用外观器模式来进行代码优化，也算是一次很小的重构。

这次重构的核心是使用外观模式也可以说门面模式，结合 SpringBoot 中的自定义 starter 中间件开发的方式，统一处理所有需要白名单的地方。

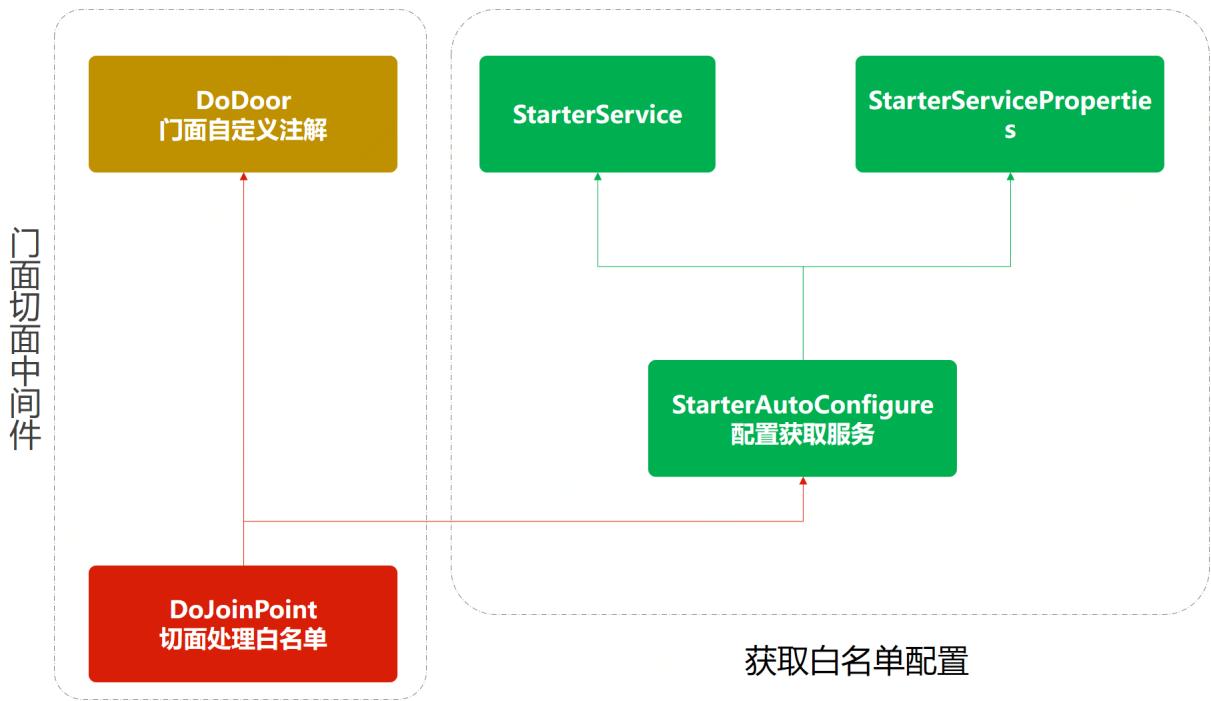
后续接下来的实现中，会涉及的知识：

1. SpringBoot的starter中间件开发方式。
2. 面向切面编程和自定义注解的使用。
3. 外部自定义配置信息的透传，SpringBoot与Spring不同，对于此类方式获取白名单配置存在差异。

1. 工程结构

```
1 itstack-demo-design-10-02
2 └── src
3     ├── main
4         ├── java
5             └── org.itstack.demo.design.door
6                 ├── annotation
7                     └── DoDoor.java
8                 ├── config
9                     ├── StarterAutoConfigure.java
10                    ├── StarterService.java
11                    └── StarterServiceProperties.java
12                     └── DoJoinPoint.java
13                 └── resources
14                     └── META-INF
15                         └── spring.factories
16             └── test
17                 └── java
18                     └── org.itstack.demo.test
19                         └── ApiTest.java
```

门面模式模型结构



- 以上是外观模式的中间件实现思路，右侧是为了获取配置文件，左侧是对于切面的处理。
- 门面模式可以是对接口的包装提供出接口服务，也可以是对逻辑的包装通过自定义注解对接口提供服务能力。

2. 代码实现

2.1 配置服务类

```

1 public class StarterService {
2
3     private String userStr;
4
5     public StarterService(String userStr) {
6         this.userStr = userStr;
7     }
8
9     public String[] split(String separatorChar) {
10        return StringUtils.split(this.userStr, separatorChar);
11    }
12
13 }
```

- 以上类的内容较简单只是为了获取配置信息。

2.2 配置类注解定义

```

1 @ConfigurationProperties("itstack.door")
2 public class StarterServiceProperties {
3
4     private String userStr;
5 }
```

```

6     public String getUserStr() {
7         return userStr;
8     }
9
10    public void setUserStr(String userStr) {
11        this.userStr = userStr;
12    }
13
14 }
```

- 用于定义好后续在 `application.yml` 中添加 `itstack.door` 的配置信息。

2.3 自定义配置类信息获取

```

1 @Configuration
2 @ConditionalOnClass(StarterService.class)
3 @EnableConfigurationProperties(StarterServiceProperties.class)
4 public class StarterAutoConfigure {
5
6     @Autowired
7     private StarterServiceProperties properties;
8
9     @Bean
10    @ConditionalOnMissingBean
11    @ConditionalOnProperty(prefix = "itstack.door", value = "enabled",
12        havingValue = "true")
13    StarterService starterService() {
14        return new StarterService(properties.getUserStr());
15    }
16 }
```

- 以上代码是对配置的获取操作，主要是对注解的定义；`@Configuration`、`@ConditionalOnClass`、`@EnableConfigurationProperties`，这一部分主要是与SpringBoot的结合使用。

2.4 切面注解定义

```

1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.METHOD)
3 public @interface DoDoor {
4
5     String key() default "";
6
7     String returnJson() default "";
8
9 }
```

- 定义了外观模式门面注解，后续就是此注解添加到需要扩展白名单的方法上。

- 这里提供了两个入参，**key**: 获取某个字段例如用户ID、**returnJson**: 确定白名单拦截后返回的具体内容。

2.5 白名单切面逻辑

```

1  @Aspect
2  @Component
3  public class DoJoinPoint {
4
5      private Logger logger = LoggerFactory.getLogger(DoJoinPoint.class);
6
7      @Autowired
8      private StarterService starterService;
9
10
11     @Pointcut("@annotation(org.itstack.demo.design.door.annotation.DoDoor)")
12     public void aopPoint() {
13
14         @Around("aopPoint()")
15         public Object doRouter(ProceedingJoinPoint jp) throws Throwable {
16             //获取内容
17             Method method = getMethod(jp);
18             DoDoor door = method.getAnnotation(DoDoor.class);
19             //获取字段值
20             String keyValue = getFiledValue(door.key(), jp.getArgs());
21             logger.info("itstack door handler method: {} value: {}",
22                         method.getName(), keyValue);
23             if (null == keyValue || "".equals(keyValue)) return jp.proceed();
24             //配置内容
25             String[] split = starterService.split(",");
26             //白名单过滤
27             for (String str : split) {
28                 if (keyValue.equals(str)) {
29                     return jp.proceed();
30                 }
31                 //拦截
32                 return returnObject(door, method);
33             }
34
35             private Method getMethod(JoinPoint jp) throws NoSuchMethodException {
36                 Signature sig = jp.getSignature();
37                 MethodSignature methodSignature = (MethodSignature) sig;
38                 return getClass(jp).getMethod(methodSignature.getName(),
39                                 methodSignature.getParameterTypes());
40             }

```

```

41     private Class<? extends Object> getClass(JoinPoint jp) throws
42         NoSuchMethodException {
43         return jp.getTarget().getClass();
44     }
45
46     //返回对象
47     private Object returnObject(DoDoor doGate, Method method) throws
48         IllegalAccessException, InstantiationException {
49         Class<?> returnType = method.getReturnType();
50         String returnJson = doGate.returnJson();
51         if ("".equals(returnJson)) {
52             return returnType.newInstance();
53         }
54         return JSON.parseObject(returnJson, returnType);
55     }
56
57     //获取属性值
58     private String getFiledValue(String filed, Object[] args) {
59         String filedValue = null;
60         for (Object arg : args) {
61             try {
62                 if (null == filedValue || "".equals(filedValue)) {
63                     filedValue = BeanUtils.getProperty(arg, filed);
64                 } else {
65                     break;
66                 }
67             } catch (Exception e) {
68                 if (args.length == 1) {
69                     return args[0].toString();
70                 }
71             }
72         }
73         return filedValue;
74     }

```

- 这里包括的内容较多，核心逻辑主要是；`Object doRouter(ProceedingJoinPoint jp)`，接下来我们分别介绍。

`@Pointcut("@annotation(org.itstack.demo.design.door.annotation.DoDoor)")`

定义切面，这里采用的是注解路径，也就是所有的加入这个注解的方法都会被切面进行管理。

getFiledValue

获取指定key也就是获取入参中的某个属性，这里主要是获取用户ID，通过ID进行拦截校验。

returnObject

返回拦截后的转换对象，也就是说当非白名单用户访问时则返回一些提示信息。

doRouter

切面核心逻辑，这一部分主要是判断当前访问的用户ID是否白名单用户，如果是则执行 `jp.proceed();`，否则返回自定义的拦截提示信息。

3. 测试验证

这里的测试我们会在工程：`itstack-demo-design-10-00` 中进行操作，通过引入jar包，配置注解的方式进行验证。

3.1 引入中间件POM配置

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>itstack-demo-design-10-02</artifactId>
4 </dependency>
```

- 打包中间件工程，给外部提供jar包服务

3.2 配置application.yml

```
1 # 自定义中间件配置
2 itstack:
3   door:
4     enabled: true
5     userStr: 1001,aaaa,ccc #白名单用户ID, 多个逗号隔开
```

- 这里主要是加入了白名单的开关和白名单的用户ID，逗号隔开。

3.3 在Controller中添加自定义注解

```
1 /**
2  * http://localhost:8080/api/queryUserInfo?userId=1001
3  * http://localhost:8080/api/queryUserInfo?userId=小团团
4  */
5 @DoDoor(key = "userId", returnJson = "{\"code\": \"1111\", \"info\": \"非白名单可访问用户拦截! \"}")
6 @RequestMapping(path = "/api/queryUserInfo", method = RequestMethod.GET)
7 public UserInfo queryUserInfo(@RequestParam String userId) {
8     return new UserInfo("虫虫:" + userId, 19, "天津市南开区旮旯胡同100号");
9 }
```

- 这里核心的内容主要是自定义的注解的添加 `@DoDoor`，也就是我们的外观模式中间件化实现。
- key：需要从入参取值的属性字段，如果是对象则从对象中取值，如果是单个值则直接使用。
- returnJson：预设拦截时返回值，是返回对象的Json。

3.4 启动SpringBoot

```

1   .
2   / \ \ / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
3   ( ( ) \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
4   \ \ / \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
5   ' \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
6   ======|_|=====|_|/_/=//_/_/_/ 
7   :: Spring Boot ::           (v2.1.2.RELEASE)
8
9  2020-06-11 23:56:55.451  WARN 65228 --- [           main]
10 ion$DefaultTemplateResolverConfiguration : Cannot find template location:
11 classpath:/templates/ (please add some templates or check your Thymeleaf
12 configuration)
13 2020-06-11 23:56:55.531  INFO 65228 --- [           main]
14 o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080
15 (http) with context path ''
16 2020-06-11 23:56:55.533  INFO 65228 --- [           main]
17 o.i.demo.design.HelloWorldApplication : Started HelloWorldApplication
18 in 1.688 seconds (JVM running for 2.934)

```

- 启动正常，SpringBoot已经启动可以对外提供服务。

3.5 访问接口测试

白名单用户访问

<http://localhost:8080/api/queryUserInfo?userId=1001>

```

1 { "code": "0000", "info": "success", "name": "虫虫:1001", "age": 19, "address": "天津市
2 南开区旮旯胡同100号" }

```

- 此时的测试结果正常，可以拿到接口数据。

非白名单用户访问

<http://localhost:8080/api/queryUserInfo?userId=小团团>

```

1 { "code": "1111", "info": "非白名单可访问用户拦截!", "name": null, "age": null, "address": null }

```

- 这次我们把 userId 换成 小团团，此时返回的信息已经是被拦截的信息。而这个拦截信息正式我们自定义注解中的信息： `@DoDoor(key = "userId", returnJson = "{\"code\": \"1111\", \"info\": \"非白名单可访问用户拦截!\"}")`

六、总结

- 以上我们通过中间件的方式实现外观模式，这样的设计可以很好的增强代码的隔离性，以及复用性，不仅使用上非常灵活也降低了每一个系统都开发这样的服务带来的风险。
- 可能目前你看这只是非常简单的白名单控制，是否需要这样的处理。但往往一个小小的开始会影响着后续无限的扩展，实际的业务开发往往也要复杂的很多，不可能如此简单。因而使用设计模式来

让代码结构更加干净整洁。

- 很多时候不是设计模式没有用，而是自己编程开发经验不足导致即使学了设计模式也很难驾驭。毕竟这些知识都是经过一些实际操作提炼出来的精华，但如果你可以按照本系列文章中的案例方式进行学习实操，还是可以增强这部分设计能力的。

第 6 节：享元模式

程序员的上下文是什么？

很多时候一大部分编程开发的人员都只是关注于功能的实现，只要自己把这部分需求写完就可以了，有点像被动的交作业。这样的问题一方面是由于很多新人还不了解程序员的职业发展，还有一部分是对于编程开发只是工作并非兴趣。但在程序员的发展来看，如果不能很好的处理上文(产品)，下文(测试)，在这样不能很好的了解业务和产品发展，也不能编写出很有体系结构的代码，日久天长，1到3年、3到5年，就很难跨越一个个技术成长的分水岭。

拥有接受和学习新知识的能力

你是否有感受过小时候在什么都还不会的时候接受知识的能力很强，但随着我们开始长大后，慢慢学习能力、处事方式、性格品行，往往固定。一方面是形成了各自的性格特征，一方面是圈子已经固定。但也正因为这样的故步，而很少愿意听取别人的意见，就像即使看到了一整片内容，在视觉盲区下也会过滤掉到80%，就在眼前也看不见，也因此导致了能力不再有较大的提升。

编程能力怎样会成长的最快

工作内容往往有些像在工厂拧螺丝，大部分内容是重复的，也可以想象过去的一年你有过多少创新和学习了新的技能。那么这时候一般为了多学些内容会买一些技术书籍，但！技术类书籍和其他书籍不同，只要不去用看了也就只是轻描淡写，很难接纳和理解。就像设计模式，虽然可能看了几遍，但是在实际编码中仍然很少会用，大部分原因还是没有认认真真的跟着实操。事必躬亲才是学习编程的最好方式。

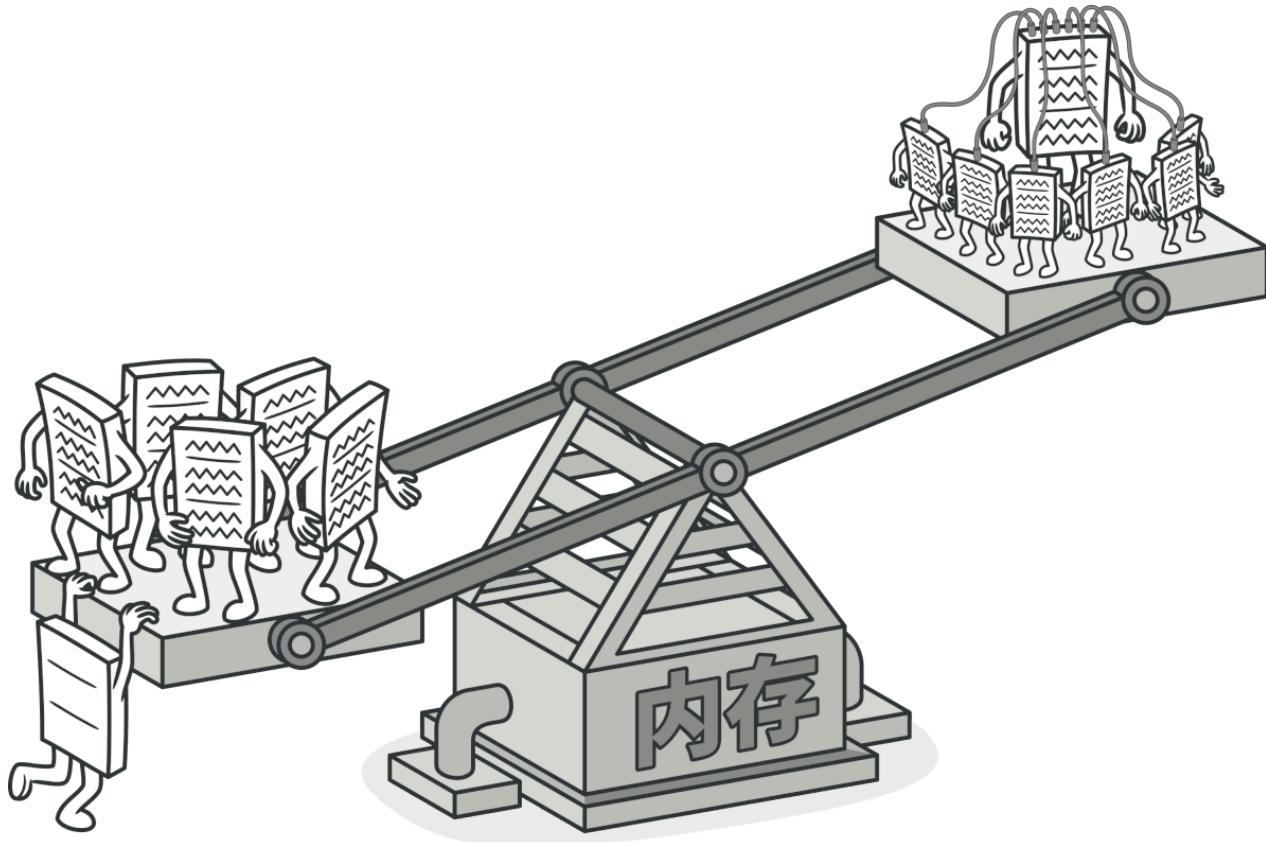
一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，

找到序号18)

工程	描述
itstack-demo-design-11-01	使用一坨代码实现业务需求
itstack-demo-design-11-02	通过设计模式优化代码结构，减少内存使用和查询耗时

二、享元模式介绍

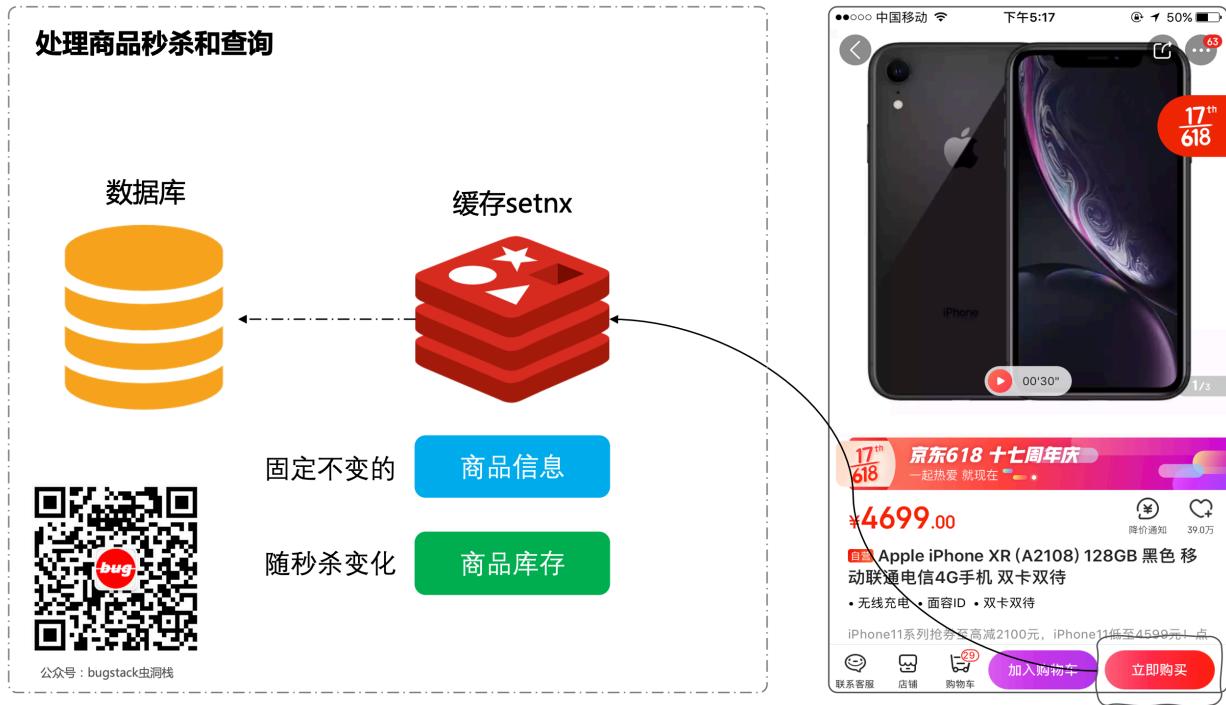


享元模式，主要在于共享通用对象，减少内存的使用，提升系统的访问效率。而这部分共享对象通常比较耗费内存或者需要查询大量接口或者使用数据库资源，因此统一抽离作为共享对象使用。

另外享元模式可以分为在服务端和客户端，一般互联网H5和Web场景下大部分数据都需要服务端进行处理，比如数据库连接池的使用、多线程线程池的使用，除了这些功能外，还有些需要服务端进行包装后的处理下发给客户端，因为服务端需要做享元处理。但在一些游戏场景下，很多都是客户端需要进行渲染地图效果，比如：树木、花草、鱼虫，通过设置不同元素描述使用享元公用对象，减少内存的占用，让客户端的游戏更加流畅。

在享元模型的实现中需要使用到享元工厂来进行管理这部分独立的对象和共享的对象，避免出现线程安全的问题。

三、案例场景模拟



在这个案例中我们模拟在商品秒杀场景下使用享元模式查询优化

你是否经历过一个商品下单的项目从最初的日均十几单到一个月后每个时段秒杀量破十万的项目。一般在最初如果没有经验的情况下可能会使用数据库行级锁的方式下保证商品库存的扣减操作，但是随着业务的快速发展秒杀的用户越来越多，这个时候数据库已经扛不住了，一般都会使用redis的分布式锁来控制商品库存。

同时在查询的时候也不需要每一次对不同的活动查询都从库中获取，因为这里除了库存以外其他的活动商品信息都是固定不变的，以此这里一般大家会缓存到内存中。

这里我们模拟使用享元模式工厂结构，提供活动商品的查询。活动商品相当于不变的信息，而库存部分属于变化的信息。

四、用一坨坨代码实现

逻辑很简单，就怕你写乱。一片片的固定内容和变化内容的查询组合，cv的哪里都是！

其实这部分逻辑的查询在一般情况很多程序员都是先查询固定信息，在使用过滤的或者添加if判断的方式补充变化的信息，也就是库存。这样写最开始并不会看出来有什么问题，但随着方法逻辑的增加，后面就越来越多重复的代码。

1. 工程结构

```

1 | itstack-demo-design-11-01
2 |   └─ src
3 |     └─ main
4 |       └─ java
5 |         └─ org.itstack.demo.design
6 |           └─ ActivityController.java

```

- 以上工程结构比较简单，之后一个控制类用于查询活动信息。

2. 代码实现

```
1  /**
2  * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3  * 公众号: bugstack虫洞栈
4  * Create by 小傅哥(fustack) @2020
5  */
6  public class ActivityController {
7
8      public Activity queryActivityInfo(Long id) {
9          // 模拟从实际业务应用从接口中获取活动信息
10         Activity activity = new Activity();
11         activity.setId(10001L);
12         activity.setName("图书嗨乐");
13         activity.setDesc("图书优惠券分享激励分享活动第二期");
14         activity.setStartTime(new Date());
15         activity.setStopTime(new Date());
16         activity.setStock(new Stock(1000, 1));
17         return activity;
18     }
19
20 }
```

- 这里模拟的是从接口中查询活动信息，基本也就是从数据库中获取所有的商品信息和库存。有点像最开始写商品销售系统，数据库就可以抗住购物量。
- 当后续因为业务的发展需要扩展代码将库存部分交给redis处理，那么就需要从redis中获取活动的库存，而不是从库中，否则将造成数据不统一的问题。

五、享元模式重构代码

接下来使用享元模式来进行代码优化，也算是一次很小的重构。

享元模式一般情况下使用此结构在平时的开发中并不太多，除了一些线程池、数据库连接池外，再就是游戏场景下的场景渲染。另外这个设计的模式思想是减少内存的使用提升效率，与我们之前使用的原型模式通过克隆对象的方式生成复杂对象，减少rpc的调用，都是此类思想。

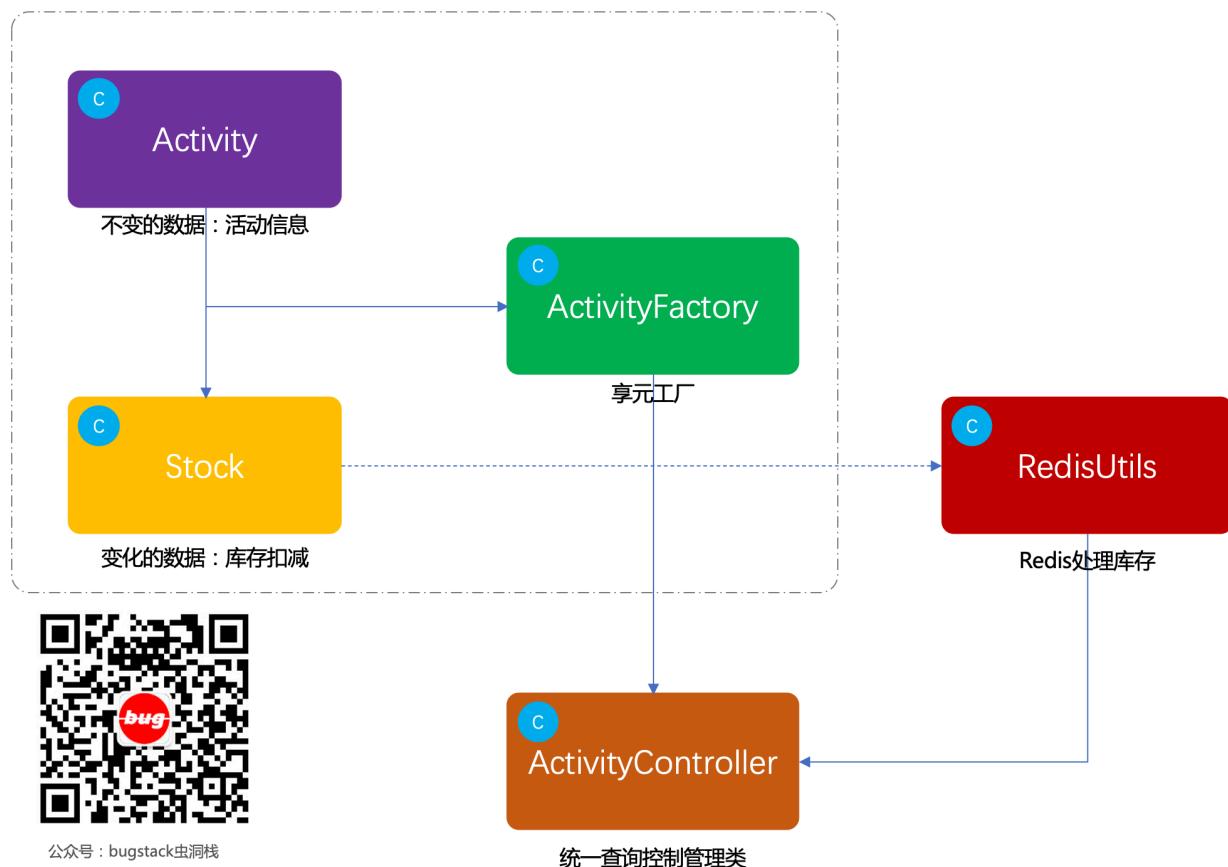
1. 工程结构

```
1  itstack-demo-design-11-02
2  └── src
3      ├── main
4      │   └── java
5      │       └── org.itstack.demo.design
6      │           ├── util
7      │           │   └── RedisUtils.java
8      │           ├── Activity.java
9      │           ├── ActivityController.java
10     │           ├── ActivityFactory.java
11     │           └── Stock.java
```

```
12 └── test
13     └── java
14         └── org.itstack.demo.test
15             └── ApiTest.java
```

享元模式模型结构

统一包装数据服务，将变化的与不变化的结合



- 以上是我们模拟查询活动场景的类图结构，左侧构建的是享元工厂，提供固定活动数据的查询，右侧是Redis存放的库存数据。
- 最终交给活动控制类来处理查询操作，并提供活动的所有信息和库存。因为库存是变化的，所以我们模拟的 `RedisUtils` 中设置了定时任务使用库存。

2. 代码实现

2.1 活动信息

```
1 /**
2  * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获！
3  * 公众号: bugstack虫洞栈
4  * Create by 小傅哥(fustack) @2020
5  */
6 public class Activity {
7
8     private Long id;          // 活动ID
9     private String name;       // 活动名称
10    private String desc;        // 活动描述
```

```
11     private Date startTime; // 开始时间
12     private Date stopTime; // 结束时间
13     private Stock stock; // 活动库存
14
15     // ...get/set
16 }
```

- 这里的对象类比较简单，只是一个活动的基础信息；id、名称、描述、时间和库存。

2.2 库存信息

```
1 public class Stock {
2
3     private int total; // 库存总量
4     private int used; // 库存已用
5
6     // ...get/set
7 }
```

- 这里是库存数据我们单独提供了一个类进行保存数据。

2.3 享元工厂

```
1 /**
2  * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3  * 公众号: bugstack虫洞栈
4  * Create by 小傅哥(fustack) @2020
5 */
6 public class ActivityFactory {
7
8     static Map<Long, Activity> activityMap = new HashMap<Long, Activity>();
9
10    public static Activity getActivity(Long id) {
11        Activity activity = activityMap.get(id);
12        if (null == activity) {
13            // 模拟从实际业务应用从接口中获取活动信息
14            activity = new Activity();
15            activity.setId(10001L);
16            activity.setName("图书嗨乐");
17            activity.setDesc("图书优惠券分享激励分享活动第二期");
18            activity.setStartTime(new Date());
19            activity.setStopTime(new Date());
20            activityMap.put(id, activity);
21        }
22        return activity;
23    }
24
25 }
```

- 这里提供的是一个享元工厂, 通过map结构存放已经从库表或者接口中查询到的数据，存放到内存中，用于下次可以直接获取。
- 这样的结构一般在我们的编程开发中还是比较常见的，当然也有些时候为了分布式的获取，会把数据存放到redis中，可以按需选择。

2.4 模拟Redis类

```

1  /**
2   * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3   * 公众号: bugstack虫洞栈
4   * Create by 小傅哥(fustack) @2020
5   */
6  public class RedisUtils {
7
8      private ScheduledExecutorService scheduledExecutorService =
9          Executors.newScheduledThreadPool(1);
10
11     private AtomicInteger stock = new AtomicInteger(0);
12
13     public RedisUtils() {
14         scheduledExecutorService.scheduleAtFixedRate(() -> {
15             // 模拟库存消耗
16             stock.addAndGet(1);
17         }, 0, 100000, TimeUnit.MICROSECONDS);
18     }
19
20     public int getStockUsed() {
21         return stock.get();
22     }
23
24 }
```

- 这里处理模拟redis的操作工具类外，还提供了一个定时任务用于模拟库存的使用，这样方面我们在测试的时候可以观察到库存的变化。

2.4 活动控制类

```

1  /**
2   * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3   * 公众号: bugstack虫洞栈
4   * Create by 小傅哥(fustack) @2020
5   */
6  public class ActivityController {
7
8      private RedisUtils redisUtils = new RedisUtils();
9
10     public Activity queryActivityInfo(Long id) {
11         Activity activity = ActivityFactory.getActivity(id);
```

```
12     // 模拟从Redis中获取库存变化信息
13     Stock stock = new Stock(1000, redisUtils.getStockUsed());
14     activity.setStock(stock);
15     return activity;
16 }
17
18 }
```

- 在活动控制类中使用了享元工厂获取活动信息，查询后将库存信息在补充上。因为库存信息是变化的，而活动信息是固定不变的。
- 最终通过统一的控制类就可以把完整包装后的活动信息返回给调用方。

3. 测试验证

3.1 编写测试类

```
1 public class ApiTest {
2
3     private Logger logger = LoggerFactory.getLogger(ApiTest.class);
4
5     private ActivityController activityController = new
ActivityController();
6
7     @Test
8     public void test_queryActivityInfo() throws InterruptedException {
9         for (int idx = 0; idx < 10; idx++) {
10             Long req = 10001L;
11             Activity activity = activityController.queryActivityInfo(req);
12             logger.info("测试结果: {} {}", req,
JSON.toJSONString(activity));
13             Thread.sleep(1200);
14         }
15     }
16
17 }
```

- 这里我们通过活动查询控制类，在`for`循环的操作下查询了十次活动信息，同时为了保证库存定时任务的变化，加了睡眠操作，实际的开发中不会有这样的睡眠。

3.2 测试结果

```
1 22:35:20.285 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":1},"stopTime":1592130919931}
2 22:35:21.634 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":18},"stopTime":1592130919931}
3 22:35:22.838 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":30},"stopTime":1592130919931}
4 22:35:24.042 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":42},"stopTime":1592130919931}
5 22:35:25.246 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":54},"stopTime":1592130919931}
6 22:35:26.452 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":66},"stopTime":1592130919931}
7 22:35:27.655 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":78},"stopTime":1592130919931}
8 22:35:28.859 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":90},"stopTime":1592130919931}
9 22:35:30.063 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":102},"stopTime":1592130919931}
10 22:35:31.268 [main] INFO org.i..t.ApiTest - 测试结果: 10001 {"desc":"图书优惠券分享激励分享活动第二期","id":10001,"name":"图书嗨乐","startTime":1592130919931,"stock": {"total":1000,"used":114},"stopTime":1592130919931}
11
12 Process finished with exit code 0
```

- 可以仔细看下 stock 部分的库存是一直在变化的，其他部分是活动信息，是固定的，所以我们使用享元模式来将这样的结构进行拆分。

六、总结

- 关于享元模式的设计可以着重学习享元工厂的设计，在一些有大量重复对象可复用的场景下，使用

此场景在服务端减少接口的调用，在客户端减少内存的占用。是这个设计模式的主要应用方式。

- 另外通过 map 结构的使用方式也可以看到，使用一个固定id来存放和获取对象，是非常关键的点。而且不只是在享元模式中使用，一些其他工厂模式、适配器模式、组合模式中都可以通过map结构存放服务供外部获取，减少ifelse的判断使用。
- 当然除了这种设计的减少内存的优点外，也有它带来的缺点，在一些复杂的业务处理场景，很不容易区分出内部和外部状态，就像我们活动信息部分与库存变化部分。如果不能很好的拆分，就会把享元工厂设计的非常混乱，难以维护。

第 7 节：代理模式

难以跨越的瓶颈期，把你拿捏滴死死的！

编程开发学习过程中遇到的瓶颈期，往往是由于看不到前进的方向。这个时候你特别希望能有人告诉你，你还欠缺些什么朝着哪个方向努力。而导致这一问题的主要原因是由于日常的业务开发太过于复制过去，日复一日的重复。没有太多的挑战，也没参与过较大体量的业务场景，除了这些开发场景因素外，还有缺少组内的技术氛围和技术分享，没有人做传播和布道者，也缺少自己对各项技术学习的热情，从而导致一直游荡在瓶颈之下，难以提升。

小公司与大公司，选择哪个？

刨除掉薪资以外你会选择什么，是不有人建议小公司，因为可以接触到各个环境，也有人建议大公司，因为正规体量大可以学习到更多。有些时候你的技术成长缓慢也是因为你的不同选择而导致的，小公司确实要接触各个环境，但往往如果你所做的业务体量不高，那么你会用到的技术栈就会相对较少，同时也会技术栈研究的深度也会较浅。大公司中确实有时候你不需要去关心一个集群的部署和维护、一个中间件的开发、全套服务监控等等，但如果你愿意了解这些技术在内部都是公开的，你会拥有无限的技术营养可以补充。而这最主要的是提升视野和事业。

除了业务中的CRUD开发，有些技术你真的很难接触到！

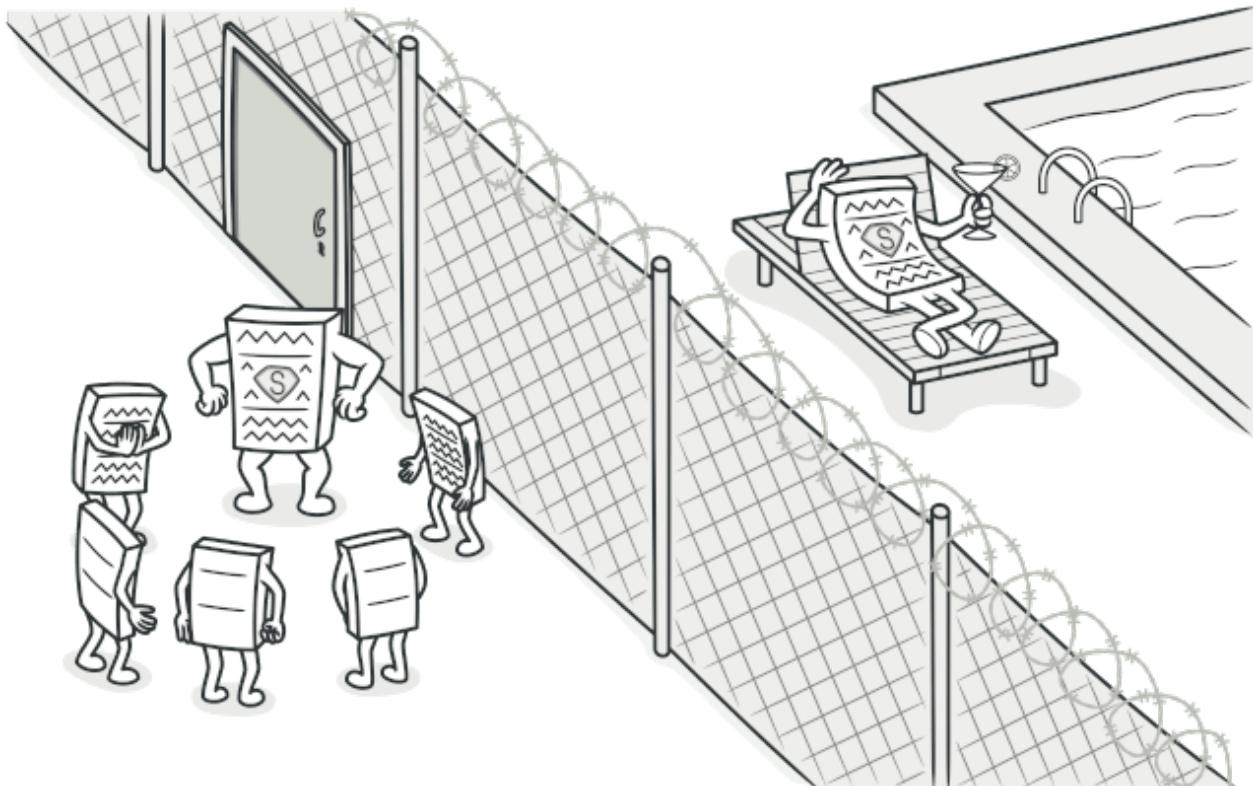
可能很多小伙伴认为技术开发就是承接下产品需求，写写CRUD，不会的百度一下，就完事了，总觉得别人问的东西像再造火箭一样。但在高体量、高并发的业务场景下，每一次的压测优化，性能提升，都像在研究一道数学题一样，反复的锤炼，压榨性能。不断的深究，找到最合适的设计。除了这些优化提升外，还有那么广阔的技术体系栈，都可能因为你只是注重CRUD而被忽略；字节码编程、领域驱动设计架构、代理模式中间件开发、JVM虚拟机实现原理等等。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. Spring 4.3.24.RELEASE
4. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-12-00	模拟MyBatis开发中间件代理类部分

二、代理模式介绍

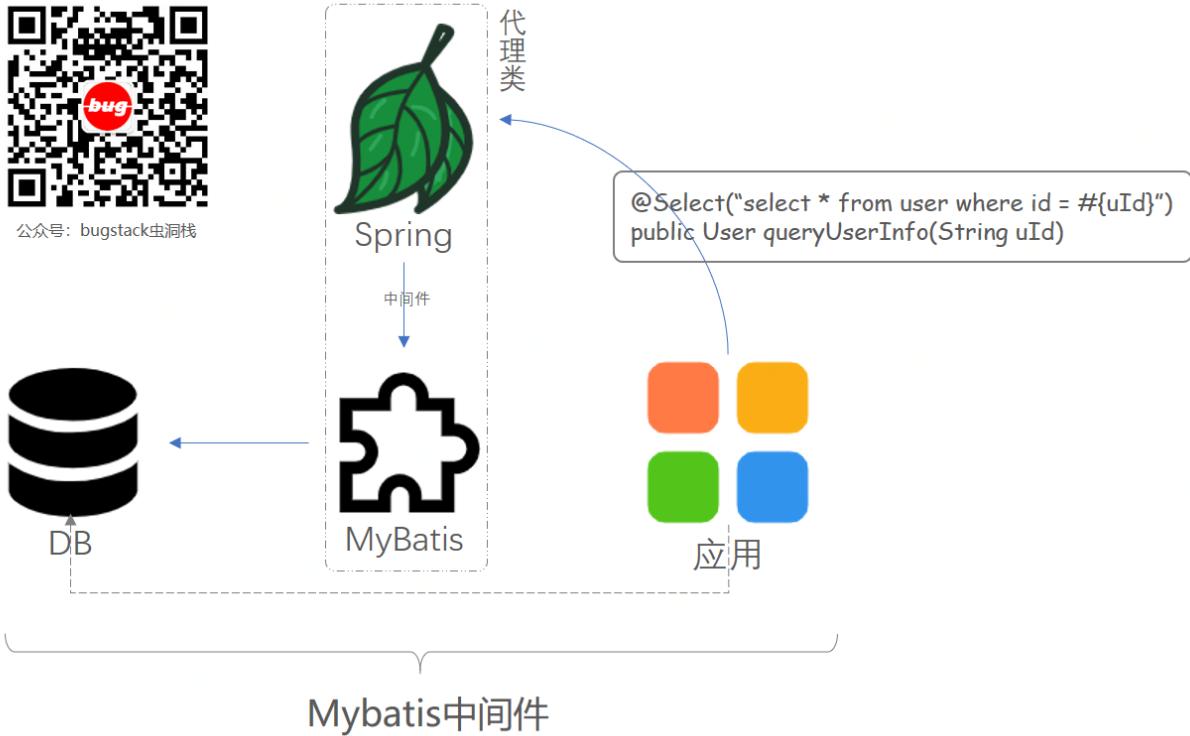


代理模式有点像老大和小弟，也有点像分销商。主要解决的是问题是为某些资源的访问、对象的类的易用操作上提供方便使用的代理服务。而这种设计思想的模式经常会在我们的系统中，或者你用到过的组件中，它们都提供给你一种非常简单易用的方式控制原本你需要编写很多代码的进行使用的服务类。

类似这样的场景可以想到：

1. 你的数据库访问层面经常会提供一个较为基础的应用，以此来减少应用服务扩容时不至于数据库连接数暴增。
2. 使用过的一些中间件例如：RPC框架，在拿到jar包对接口的描述后，中间件会在服务启动的时候生成对应的代理类，当调用接口的时候，实际是通过代理类发出的socket信息进行通过。
3. 另外像我们常用的 MyBatis，基本是定义接口但是不需要写实现类，就可以对 `xml` 或者自定义注解里的 `sql` 语句进行增删改查操作。

三、案例场景模拟



在本案例中我们模拟实现mybatis-spring中代理类生成部分

对于Mybatis的使用中只需要定义接口不需要写实现类就可以完成增删改查操作，有疑问的小伙伴，在本章节中就可以学习到这部分知识。解析下来我们会通过实现一个这样的代理类交给spring管理的核心过程，来讲述代理类模式。

这样的案例场景在实际的业务开发中其实不多，因为这是将这种思想运用在中间件开发上，而很多小伙伴经常是做业务开发，所以对Spring的bean定义以及注册和对代理以及反射调用的知识了解的相对较少。但可以通过本章节作为一个入门学习，逐步了解。

四、代理类模式实现过程

接下来会使用代理类模式来模拟实现一个Mybatis中对类的代理过程，也就是只需要定义接口，就可以关联到方法注解中的 `sql` 语句完成对数据库的操作。

这里需要注意一些知识点：

1. `BeanDefinitionRegistryPostProcessor`，spring的接口类用于处理对bean的定义注册。
2. `GenericBeanDefinition`，定义bean的信息，在mybatis-spring中使用到的是；`ScannedGenericBeanDefinition` 略有不同。
3. `FactoryBean`，用于处理bean工厂的类，这个类非常见。

1. 工程结构

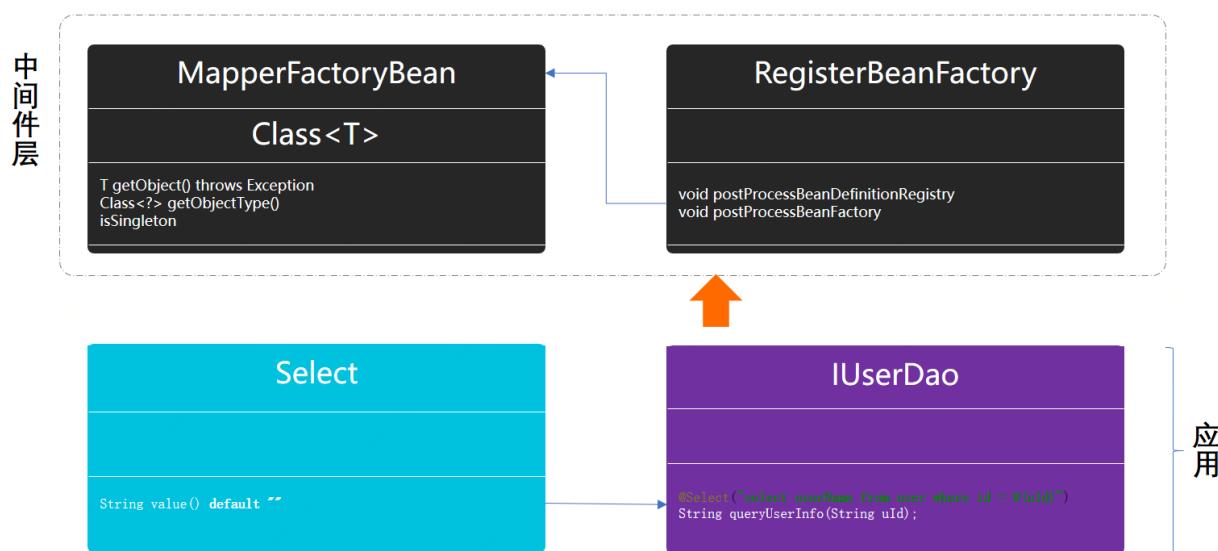
```

1  itstack-demo-design-12-00
2  └── src
3      └── main
4          └── java
5              └── org.itstack.demo.design
6                  └── agent

```

```
7 |     |     | └── MapperFactoryBean.java  
8 |     |     | └── RegisterBeanFactory.java  
9 |     |     | └── Select.java  
10 |     |     └── IUserDao.java  
11 |     └── resources  
12 |         └── spring-config.xml  
13 └── test  
    └── java  
        └── org.itstack.demo.test  
            └── ApiTest.java
```

代理模式中间件模型结构



- 此模型中涉及的类并不多，但都是抽离出来的核心处理类。主要的事情就是对类的代理和注册到spring中。
- 上图中最上面是关于中间件的实现部分，下面对应的是功能的使用。

2. 代码实现

2.1 自定义注解

```
1  @Documented  
2  @Retention(RetentionPolicy.RUNTIME)  
3  @Target({ElementType.METHOD})  
4  public @interface Select {  
5  
6      String value() default ""; // sql语句  
7  
8 }
```

- 这里我们定义了一个模拟mybatis-spring中的自定义注解，用于使用在方法层面。

2.2 Dao层接口

```

1  public interface IUserDao {
2
3      @Select("select userName from user where id = #{uId}")
4      String queryUserInfo(String uId);
5
6  }

```

- 这里定义一个Dao层接口，并把自定义注解添加上。这与你使用的mybatis组件是一样的。
- 2.1和2.2是我们的准备工作，后面开始实现中间件功能部分。

2.3 代理类定义

```

1  public class MapperFactoryBean<T> implements FactoryBean<T> {
2
3      private Logger logger =
4          LoggerFactory.getLogger(MapperFactoryBean.class);
5
6      private Class<T> mapperInterface;
7
8      public MapperFactoryBean(Class<T> mapperInterface) {
9          this.mapperInterface = mapperInterface;
10     }
11
12     @Override
13     public T getObject() throws Exception {
14         InvocationHandler handler = (proxy, method, args) -> {
15             Select select = method.getAnnotation(Select.class);
16             logger.info("SQL: {}", select.value().replace("#{uId}",
17                     args[0].toString()));
18             return args[0] + ",小傅哥,bugstack.cn - 沉淀、分享、成长，让自己和他
19             人都能有所收获!";
20         };
21         return (T)
22             Proxy.newProxyInstance(this.getClass().getClassLoader(), new Class[]
23             {mapperInterface}, handler);
24     }
25
26     @Override
27     public Class<?> getObjectType() {
28         return mapperInterface;
29     }
30
31 }

```

- 如果你有阅读过mybatis源码，是可以看到这样的一个类：`MapperFactoryBean`，这里我们也模拟一个这样的类，在里面实现我们对代理类的定义。
- 通过继承`FactoryBean`，提供bean对象，也就是方法：`T getObject()`。
- 在方法`getObject()`中提供类的代理以及模拟对sql语句的处理，这里包含了用户调用dao层方法时候的处理逻辑。
- 还有最上面我们提供构造函数来透传需要被代理类，`Class<T> mapperInterface`，在mybatis中也是使用这样的方式进行透传。
- 另外`getObjectType()`提供对象类型反馈，以及`isSingleton()`返回类是单例的。

2.4 将Bean定义注册到Spring容器

```

1  public class RegisterBeanFactory implements
2      BeanDefinitionRegistryPostProcessor {
3
4      @Override
5      public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
6          registry) throws BeansException {
7
8          GenericBeanDefinition beanDefinition = new
9              GenericBeanDefinition();
10         beanDefinition.setBeanClass(MapperFactoryBean.class);
11         beanDefinition.setScope("singleton");
12
13         beanDefinition.getConstructorArgumentValues().addGenericArgumentValue(IUser
14             Dao.class);
15
16         BeanDefinitionHolder definitionHolder = new
17             BeanDefinitionHolder(beanDefinition, "userDao");
18         BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
19             registry);
20     }
21
22     @Override
23     public void postProcessBeanFactory(ConfigurableListableBeanFactory
24         configurableListableBeanFactory) throws BeansException {
25         // left intentionally blank
26     }
27
28 }

```

- 这里我们将代理的bean交给spring容器管理，也就可以非常方便让我们可以获取到代理的bean。这部分是spring中关于一个bean注册过程的源码。
- `GenericBeanDefinition`，用于定义一个bean的基本信息`setBeanClass(MapperFactoryBean.class)`，也包括可以透传给构造函数信息`addGenericArgumentValue(IUserDao.class)`；
- 最后使用`BeanDefinitionReaderUtils.registerBeanDefinition`，进行bean的注册，也就是注册到`DefaultListableBeanFactory`中。

2.5 配置文件spring-config

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans
5   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
6   default-autowire="byName">
7
8   <bean id="userDao"
9     class="org.itstack.demo.design.agent.RegisterBeanFactory" />
9 </beans>
```

- 接下来在配置文件中添加我们的bean配置，在mybatis的使用中一般会配置扫描的dao层包，这样就可以减少这部分的配置。

3. 测试验证

3.1 编写测试类

```
1 @Test
2 public void test_IUserDao() {
3     BeanFactory beanFactory = new ClassPathXmlApplicationContext("spring-
4     config.xml");
5     IUserDao userDao = beanFactory.getBean("userDao", IUserDao.class);
6     String res = userDao.queryUserInfo("100001");
7     logger.info("测试结果: {}", res);
7 }
```

- 测试的过程比较简单，通过加载Bean工厂获取我们的代理类的实例对象，之后调用方法返回结果。
- 那么这个过程你可以看到我们是没有对接口先一个实现类的，而是使用代理的方式给接口生成一个实现类，并交给spring管理。

3.2 测试结果

```
1 23:21:57.551 [main] DEBUG o.s.core.env.StandardEnvironment - Adding
PropertySource 'systemProperties' with lowest search precedence
2 ...
3 23:21:57.858 [main] DEBUG o.s.c.s.ClassPathXmlApplicationContext - Unable
to locate LifecycleProcessor with name 'lifecycleProcessor': using default
[org.springframework.context.support.DefaultLifecycleProcessor@7bc1a03d]
4 23:21:57.859 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning
cached instance of singleton bean 'lifecycleProcessor'
5 23:21:57.860 [main] DEBUG o.s.c.e.PropertySourcesPropertyResolver - Could
not find key 'spring.liveBeansView.mbeanDomain' in any property source
6 23:21:57.861 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning
cached instance of singleton bean 'userDao'
7 23:21:57.915 [main] INFO o.i.d.design.agent.MapperFactoryBean - SQL:
select userName from user where id = 100001
8 23:21:57.915 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果:
100001,小傅哥,bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
9
10 Process finished with exit code 0
```

- 从测试结果可以看到，我们打印了SQL语句，这部分语句是从自定义注解中获取的；`select
userName from user where id = 100001`，我们做了简单的适配。在mybatis框架中会交给`SqlSession`的实现类进行逻辑处理返回操作数据库数据
- 而这里我们的测试结果是一个固定的，如果你愿意更加深入的研究可以尝试与数据库操作层进行关联，让这个框架可以更加完善。

五、总结

- 关于这部分代理模式的讲解我们采用了开发一个关于`mybatis-spring`中间件中部分核心功能来体现代理模式的强大之处，所以涉及到了一些关于代理类的创建以及spring中bean的注册这些知识点，可能在平常的业务开发中都是很少用到的，但是在中间件开发中确实非常常见的操作。
- 代理模式除了开发中间件外还可以是对服务的包装，物联网组件等等，让复杂的各项服务变为轻量级调用、缓存使用。你可以理解为你家里的电灯开关，我们不能操作220v电线的人肉连接，但是可以使用开关，避免触电。
- 代理模式的设计方式可以让代码更加整洁、干净易于维护，虽然在这部分开发中额外增加了很多类也包括了自己处理bean的注册等，但是这样的中间件复用性极高也更加智能，可以非常方便的扩展到各个服务应用中。

行为模式(10节)

这类模式负责对象间的高效沟通和职责委派。

行为模式包括：责任链、命令、迭代器、中介者、备忘录、观察者、状态、策略、模板、访问者，这10类。

第1节：责任链模式

场地和场景的重要性

射击需要去靶场学习、滑雪需要去雪场体验、开车需要能上路实践，而编程开发除了能完成产品的功能流程，还需要保证系统的可靠性能。就像你能听到的一些系统监控指标；`QPS`、`TPS`、`TP99`、`TP999`、`可用率`、`响应时长`等等，而这些指标的总和评估就是一个系统的健康度。但如果你几乎没有听到这样的技术术语，也没接触过类似高并发场景，那么就很像驾驶证的科目1考了100分，但不能上路。没有这样的技术场景给你训练，让你不断的体会系统的脾气秉性，即便你有再多的想法都没法实现。所以，如果真的想学习一定要去一个有实操的场景，下水试试才能学会狗刨。

你的视觉盲区有多大

同样一本书、同样一条路、同样一座城，你真的以为生活有选择吗？有时候很多选项都是摆设，给你多少次机会你都选的一模一样。这不是你选不选而是你的认知范围决定了你下一秒做的事情，另外的一个下一秒又决定了再下一个下一秒。就像管中窥豹一样，20%的面积在你视觉里都是黑色的，甚至就总是忽略看不到，而这看不到的20%就是生命中的时运！但，人可以学习，可以成长，可以脱胎换骨，可以努力付出，通过一次次的蜕变而看到剩下的20%！

没有设计图纸你敢盖楼吗

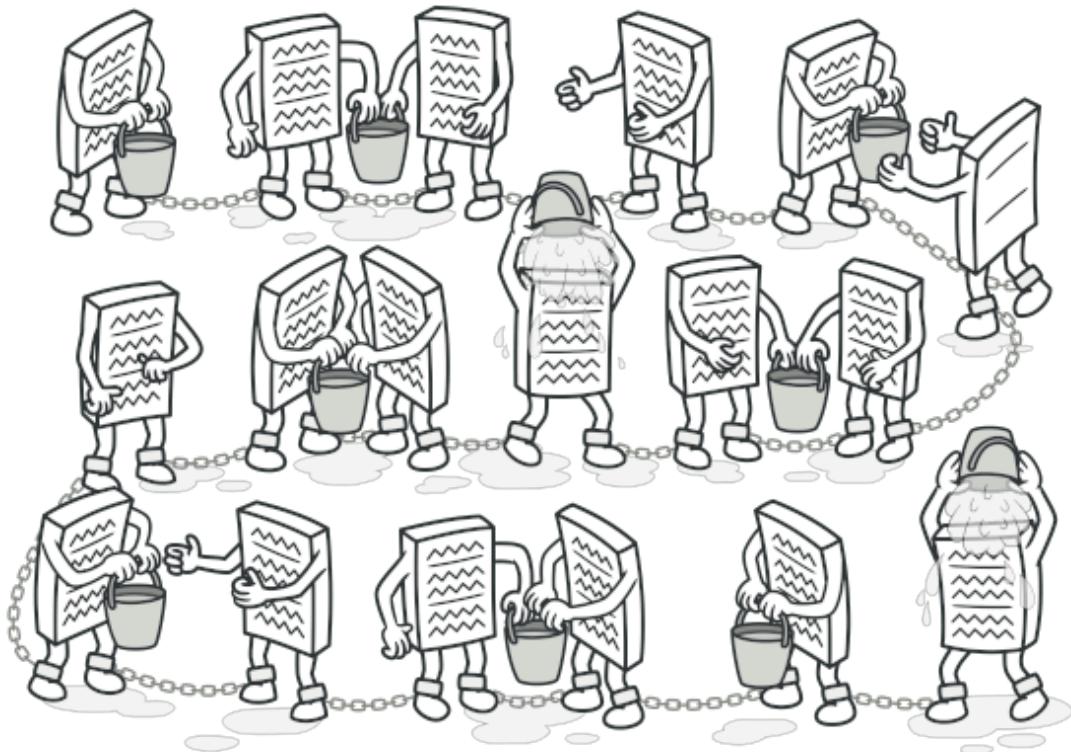
编程开发中最好的什么，是设计。运用架构思维、经验心得、`才华灵感`，构建出最佳的系统。真正的研发会把自己写的代码当做作品来欣赏，你说这是一份工作，但在这样的人眼里这可不是一份工作，而是一份工匠精神。就像可能时而你也会为自己因为一个`niubility`的设计而豪迈万丈，为能上线一个扛得住每秒200万访问量的系统会精神焕发。这样的自豪感就是一次次垒砖一样垫高脚底，不断的把你的视野提高，让你能看到上层设计也能知晓根基建设。可以把控全局，也可以治理细节。这一份份知识的沉淀，来帮助你绘制出一张系统架构蓝图。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-13-00	场景模拟工程；模拟一个上线流程审批的接口。
itstack-demo-design-13-01	使用一坨代码实现业务需求
itstack-demo-design-13-02	通过设计模式优化改造代码，产生对比性从而学习

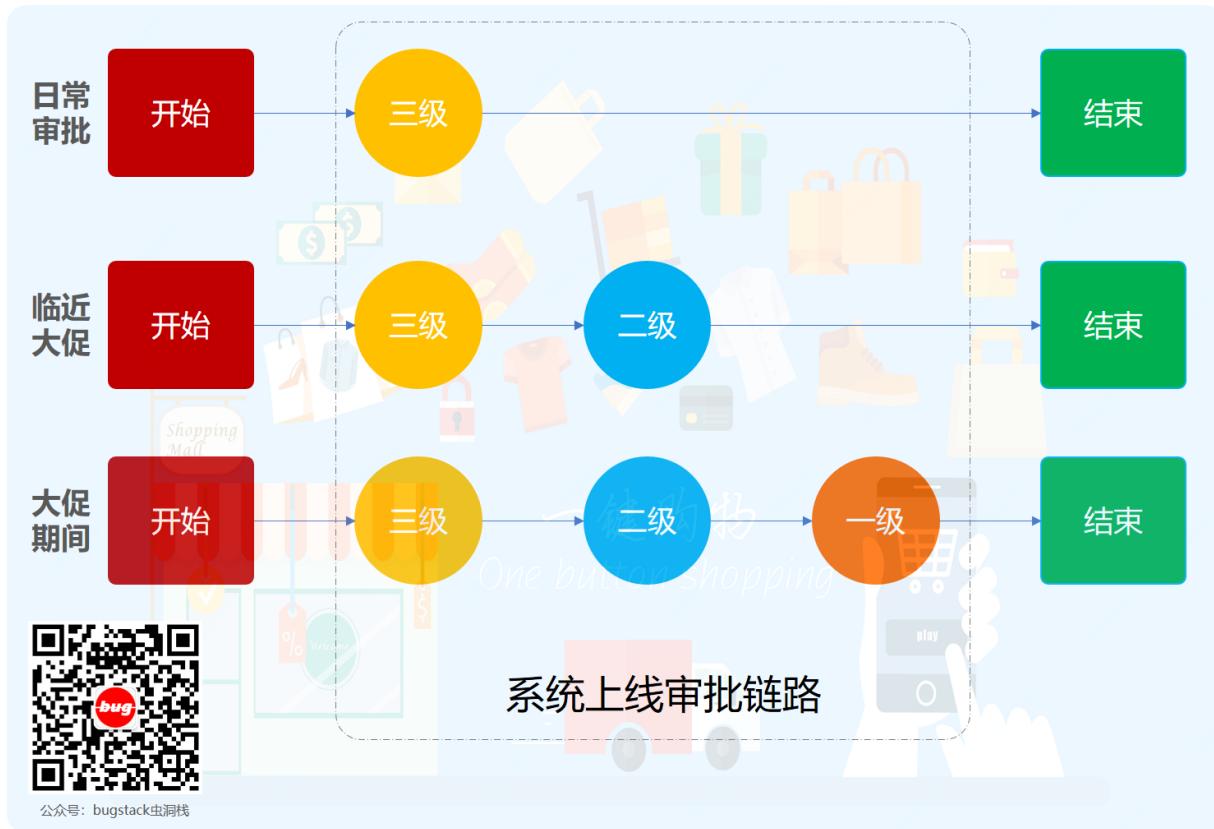
二、责任链模式介绍



击鼓传雷，看上图你是否想起周星驰有一个电影，大家坐在海边围成一个圈，拿着一个点燃的炸弹，互相传递。

责任链模式的核心是解决一组服务中的先后执行处理关系，就有点像你没钱花了，需要家庭财务支出审批，10块钱以下找闺女审批，100块钱先闺女审批在媳妇审批。你可以理解想象成当你要跳槽的时候被安排的明明白白的被各个领导签字放行。

三、案例场景模拟



在本案例中我们模拟在618大促销期间的业务系统上线审批流程场景

像是这些一线电商类的互联网公司，阿里、京东、拼多多等，在618期间都会做一些运营活动场景以及提供的扩容备战，就像过年期间百度的红包一样。但是所有开发的这些系统都需要陆续的上线，因为临近618有时候也有一些紧急的调整的需要上线，但为了保障线上系统的稳定性是尽可能的减少上线的，也会相应的增强审批力度。就像一级响应、二级响应一样。

而这审批的过程在随着特定时间点会增加不同级别的负责人加入，每个人就像责任链模式中的每一个核心点。对于研发小伙伴并不需要关心具体的审批流程处理细节，只需要知道这个上线更严格，级别也更高，但对于研发人员来说同样是点击相同的提审按钮，等待审核。

接下来我们就模拟这样一个业务诉求场景，使用责任链的设计模式来实现此功能。

1. 场景模拟工程

```

1 itstack-demo-design-13-00
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ AuthService.java

```

- 这里的代码结构比较简单，只有一个模拟审核和查询审核结果的服务类。相当于你可以调用这个类去审核工程和获取审核结构，这部分结果信息是模拟的写到缓存实现。

2. 场景简述

2.1 模拟审核服务

```

1 public class AuthService {
2
3     private static Map<String, Date> authMap = new
4     ConcurrentHashMap<String, Date>();
5
6     public static Date queryAuthInfo(String uId, String orderId) {
7         return authMap.get(uId.concat(orderId));
8     }
9
10    public static void auth(String uId, String orderId) {
11        authMap.put(uId.concat(orderId), new Date());
12    }
13}

```

- 这里面提供了两个接口一个是查询审核结果(queryAuthInfo)、另外一个是处理审核(auth)。
- 这部分是把由谁审核的和审核的单子ID作为唯一key值记录到内存Map结构中。

四、用一坨坨代码实现

这里我们先使用最直接的方式来实现功能

按照我们的需求审批流程，平常系统上线只需要三级负责人审批就可以，但是到了618大促时间点，就需要由二级负责以及一级负责人一起加入审批系统上线流程。在这里我们使用非常直接的if判断方式来实现这样的需求。

1. 工程结构

```

1 itstack-demo-design-13-01
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 └── AuthController.java

```

- 这部分非常简单的只包含了一个审核的控制类，就像有些伙伴开始写代码一样，一个类写所有需求。

2. 代码实现

```

1 public class AuthController {
2
3     private SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd
4 HH:mm:ss"); // 时间格式化
5
6     public AuthInfo doAuth(String uId, String orderId, Date authDate)
7         throws ParseException {
8
9 }

```

```

7     // 三级审批
8         Date date = AuthService.queryAuthInfo("1000013", orderId);
9         if (null == date) return new AuthInfo("0001", "单号: ", orderId, "状态: 待三级审批负责人 ", "王工");
10
11     // 二级审批
12     if (authDate.after(f.parse("2020-06-01 00:00:00")) &&
authDate.before(f.parse("2020-06-25 23:59:59"))) {
13         date = AuthService.queryAuthInfo("1000012", orderId);
14         if (null == date) return new AuthInfo("0001", "单号: ",
orderId, "状态: 待二级审批负责人 ", "张经理");
15     }
16
17     // 一级审批
18     if (authDate.after(f.parse("2020-06-11 00:00:00")) &&
authDate.before(f.parse("2020-06-20 23:59:59"))) {
19         date = AuthService.queryAuthInfo("1000011", orderId);
20         if (null == date) return new AuthInfo("0001", "单号: ",
orderId, "状态: 待一级审批负责人 ", "段总");
21     }
22
23     return new AuthInfo("0001", "单号: ", orderId, "状态: 审批完成");
24 }
25
26 }
```

- 这里从上到下分别判断了在指定时间范围内由不同的人员进行审批，就像618上线的时候需要三个负责人都审批才能让系统进行上线。
- 像是这样的功能看起来很简单的，但是实际的业务中会有很多部门，但如果这样实现就很难进行扩展，并且在改动扩展调整也非常麻烦。

3. 测试验证

3.1 编写测试类

```

1 @Test
2 public void test_AuthController() throws ParseException {
3     AuthController authController = new AuthController();
4
5     // 模拟三级负责人审批
6     logger.info("测试结果: {}", JSON.toJSONString(authController.doAuth("小
傅哥", "1000998004813441", new Date())));
7     logger.info("测试结果: {}", "模拟三级负责人审批, 王工");
8     AuthService.auth("1000013", "1000998004813441");
9
10    // 模拟二级负责人审批
11    logger.info("测试结果: {}", JSON.toJSONString(authController.doAuth("小
傅哥", "1000998004813441", new Date())));
12    logger.info("测试结果: {}", "模拟二级负责人审批, 张经理");
```

```

13     AuthService.auth("1000012", "1000998004813441");
14
15     // 模拟一级负责人审批
16     logger.info("测试结果: {}", JSON.toJSONString(authController.doAuth("小
17     傅哥", "1000998004813441", new Date())));
18     logger.info("测试结果: {}", "模拟一级负责人审批, 段总");
19     AuthService.auth("1000011", "1000998004813441");
20
21     logger.info("测试结果: {}", "审批完成");
22 }

```

- 这里模拟每次查询是否审批完成，随着审批的不同节点，之后继续由不同的负责人进行审批操作。
- `authController.doAuth`，是查看审批的流程节点、`AuthService.auth`，是审批方法用于操作节点流程状态。

3.2 测试结果

```

1  23:25:00.363 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结
2  果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待三级审批负责人 王工"}
3  23:25:00.366 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模
4  拟三级负责人审批, 王工
5  23:25:00.367 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结
6  果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待二级审批负责人 张经理"}
7  23:25:00.367 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模
8  拟二级负责人审批, 张经理
9  23:25:00.368 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结
10 果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待一级审批负责人 段总"}
11 23:25:00.368 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模
12 拟一级负责人审批, 段总
13 23:25:00.368 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 审
14 批完成
15
16 Process finished with exit code 0

```

- 从测试结果上可以看到一层层的由不同的人员进行审批，审批完成后到下一个人进行处理。单看结果是满足我们的诉求，只不过很难扩展和调整流程，相当于代码写的死死的。

五、责任链模式重构代码

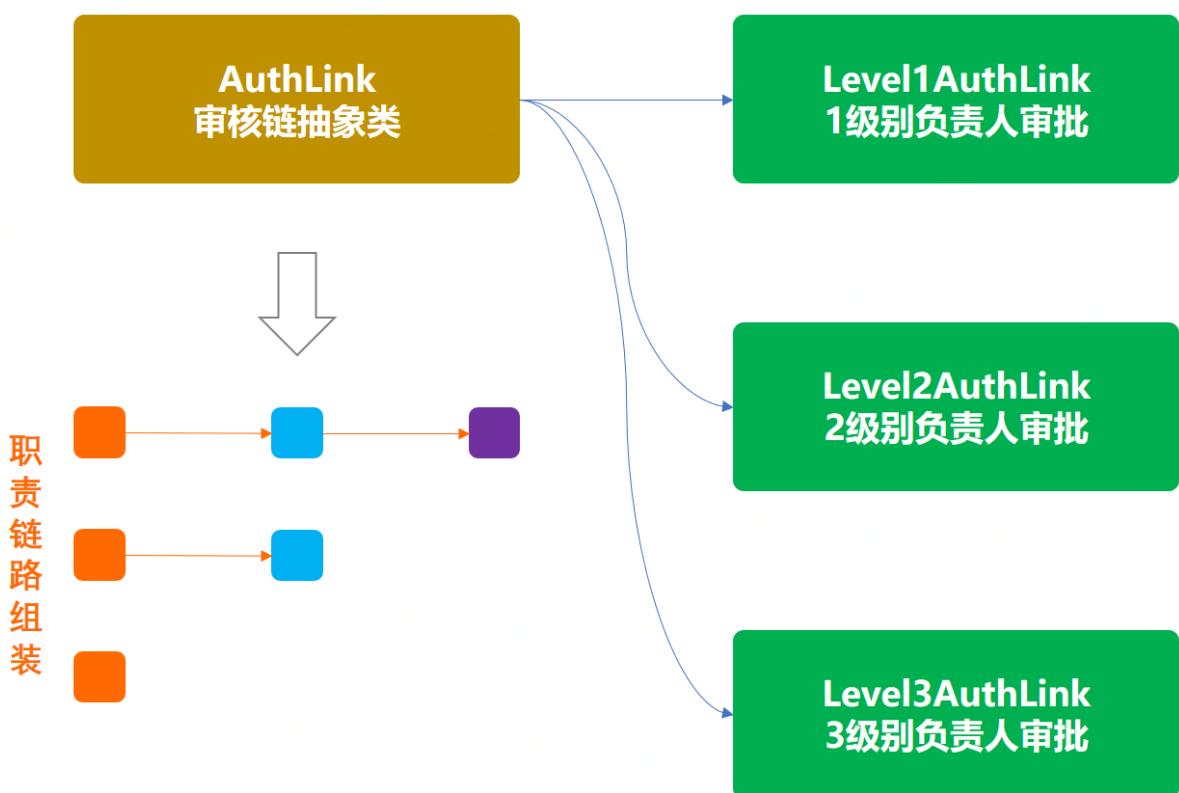
接下来使用装饰器模式来进行代码优化，也算是一次很小的重构。

责任链模式可以让各个服务模块更加清晰，而每一个模块间可以通过 `next` 的方式进行获取。而每一个 `next` 是由继承的统一抽象类实现的。最终所有类的职责可以动态的进行编排使用，编排的过程可以做成可配置化。

1. 工程结构

```
1 itstack-demo-design-13-02
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ impl
7           └─ Level1AuthLink.java
8           └─ Level2AuthLink.java
9           └─ Level3AuthLink.java
10      └─ AuthInfo.java
11      └─ AuthLink.java
```

责任链模式模型结构



- 上图是这个业务模型中责任链结构的核心部分，通过三个实现了统一抽象类 `AuthLink` 的不同规则，再进行责任编排模拟出一条链路。这个链路就是业务中的责任链。
- 一般在使用责任链时候如果是场景比较固定，可以通过写死到代码中进行初始化。但如果业务场景经常变化可以做成xml配置的方式进行处理，也可以落到库里进行初始化操作。

2. 代码实现

2.1 责任链中返回对象定义

```
1 public class AuthInfo {
2
3   private String code;
4   private String info = "";
5 }
```

```

6     public AuthInfo(String code, String ...infos) {
7         this.code = code;
8         for (String str:infos){
9             this.info = this.info.concat(str);
10        }
11    }
12
13    // ...get/set
14 }
```

- 这个类的是包装了责任链处理过程中返回结果的类，方面处理每个责任链的返回信息。

2.2 链路抽象类定义

```

1  public abstract class AuthLink {
2
3      protected Logger logger = LoggerFactory.getLogger(AuthLink.class);
4
5      protected SimpleDateFormat f = new SimpleDateFormat("yyyy-MM-dd
6 HH:mm:ss"); // 时间格式化
7      protected String levelUserId; // 级别人员ID
8      protected String levelUserName; // 级别人员姓名
9      private AuthLink next; // 责任链
10
11     public AuthLink(String levelUserId, String levelUserName) {
12         this.levelUserId = levelUserId;
13         this.levelUserName = levelUserName;
14     }
15
16     public AuthLink next() {
17         return next;
18     }
19
20     public AuthLink appendNext(AuthLink next) {
21         this.next = next;
22         return this;
23     }
24
25     public abstract AuthInfo doAuth(String uId, String orderId, Date
26 authDate);
27 }
```

- 这部分是责任链，链接起来的核心部分。`AuthLink next`，重点在于可以通过`next`方式获取下一个链路需要处理的节点。
- `levelUserId`、`levelUserName`，是责任链中的公用信息，标记每一个审核节点的人员信息。
- 抽象类中定义了一个抽象方法，`abstract AuthInfo doAuth`，这是每一个实现者必须实现的类，不同的审核级别处理不同的业务。

2.3 三个审核实现类

Level1AuthLink

```
1 public class Level1AuthLink extends AuthLink {  
2  
3     public Level1AuthLink(String levelUserId, String levelUserName) {  
4         super(levelUserId, levelUserName);  
5     }  
6  
7     public AuthInfo doAuth(String uId, String orderId, Date authDate) {  
8         Date date = AuthService.queryAuthInfo(levelUserId, orderId);  
9         if (null == date) {  
10             return new AuthInfo("0001", "单号: ", orderId, " 状态: 待一级审批  
负责人 ", levelUserName);  
11         }  
12         AuthLink next = super.next();  
13         if (null == next) {  
14             return new AuthInfo("0000", "单号: ", orderId, " 状态: 一级审批完  
成负责人 ", " 时间: ", f.format(date), " 审批人: ", levelUserName);  
15         }  
16  
17         return next.doAuth(uId, orderId, authDate);  
18     }  
19  
20 }
```

Level2AuthLink

```
1 public class Level2AuthLink extends AuthLink {  
2  
3     private Date beginDate = f.parse("2020-06-11 00:00:00");  
4     private Date endDate = f.parse("2020-06-20 23:59:59");  
5  
6     public Level2AuthLink(String levelUserId, String levelUserName) throws  
ParseException {  
7         super(levelUserId, levelUserName);  
8     }  
9  
10    public AuthInfo doAuth(String uId, String orderId, Date authDate) {  
11        Date date = AuthService.queryAuthInfo(levelUserId, orderId);  
12        if (null == date) {  
13            return new AuthInfo("0001", "单号: ", orderId, " 状态: 待二级审批  
负责人 ", levelUserName);  
14        }  
15        AuthLink next = super.next();  
16        if (null == next) {  
17            return new AuthInfo("0000", "单号: ", orderId, " 状态: 二级审批完  
成负责人 ", " 时间: ", f.format(date), " 审批人: ", levelUserName);  
18        }  
19    }  
20 }
```

```

18     }
19
20     if (authDate.before(beginDate) || authDate.after(endDate)) {
21         return new AuthInfo("0000", "单号: ", orderId, " 状态: 二级审批完
成负责人", " 时间: ", f.format(date), " 审批人: ", levelUserName);
22     }
23
24     return next.doAuth(uId, orderId, authDate);
25 }
26
27 }

```

Level3AuthLink

```

1 public class Level3AuthLink extends AuthLink {
2
3     private Date beginDate = f.parse("2020-06-01 00:00:00");
4     private Date endDate = f.parse("2020-06-25 23:59:59");
5
6     public Level3AuthLink(String levelUserId, String levelUserName) throws
ParseException {
7         super(levelUserId, levelUserName);
8     }
9
10    public AuthInfo doAuth(String uId, String orderId, Date authDate) {
11        Date date = AuthService.queryAuthInfo(levelUserId, orderId);
12        if (null == date) {
13            return new AuthInfo("0001", "单号: ", orderId, " 状态: 待三级审批
负责人 ", levelUserName);
14        }
15        AuthLink next = super.next();
16        if (null == next) {
17            return new AuthInfo("0000", "单号: ", orderId, " 状态: 三级审批负
责人完成", " 时间: ", f.format(date), " 审批人: ", levelUserName);
18        }
19
20        if (authDate.before(beginDate) || authDate.after(endDate)) {
21            return new AuthInfo("0000", "单号: ", orderId, " 状态: 三级审批负
责人完成", " 时间: ", f.format(date), " 审批人: ", levelUserName);
22        }
23
24        return next.doAuth(uId, orderId, authDate);
25    }
26
27 }

```

- 如上三个类; `Level1AuthLink`、`Level2AuthLink`、`Level3AuthLink`，实现了不同的审核级别处理的简单逻辑。

- 例如第一个审核类中会先判断是否审核通过，如果没有审核通过则返回结果给调用方，引导去审核。（这里简单模拟审核后有时间信息不为空，作为判断条件）
- 判断完成后获取下一个审核节点；`super.next();`，如果不存在下一个节点，则直接返回结果。
- 之后是根据不同的业务时间段进行判断是否需要，二级和一级的审核。
- 最后返回下一个审核结果；`next.doAuth(uid, orderId, authDate);`，有点像递归调用。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test_AuthLink() throws ParseException {
3      AuthLink authLink = new Level3AuthLink("1000013", "王工")
4          .appendNext(new Level2AuthLink("1000012", "张经理")
5              .appendNext(new Level1AuthLink("1000011", "段总")));
6
7      logger.info("测试结果: {}", JSON.toJSONString(authLink.doAuth("小傅哥",
8          "1000998004813441", new Date())));
9
10     // 模拟三级负责人审批
11     AuthService.auth("1000013", "1000998004813441");
12     logger.info("测试结果: {}", "模拟三级负责人审批, 王工");
13     logger.info("测试结果: {}", JSON.toJSONString(authLink.doAuth("小傅哥",
14         "1000998004813441", new Date())));
15
16     // 模拟二级负责人审批
17     AuthService.auth("1000012", "1000998004813441");
18     logger.info("测试结果: {}", "模拟二级负责人审批, 张经理");
19     logger.info("测试结果: {}", JSON.toJSONString(authLink.doAuth("小傅哥",
20         "1000998004813441", new Date())));
21
22     // 模拟一级负责人审批
23     AuthService.auth("1000011", "1000998004813441");
24     logger.info("测试结果: {}", "模拟一级负责人审批, 段总");
25     logger.info("测试结果: {}", JSON.toJSONString(authLink.doAuth("小傅哥",
26         "1000998004813441", new Date())));
27 }
```

- 这里包括最核心的责任链创建，实际的业务中会包装到控制层；`AuthLink authLink = new Level3AuthLink("1000013", "王工") .appendNext(new Level2AuthLink("1000012", "张经理") .appendNext(new Level1AuthLink("1000011", "段总")));` 通过把不同的责任节点进行组装，构成一条完整业务的责任链。
- 接下里不断的执行查看审核链路`authLink.doAuth(...)`，通过返回结果对数据进行3、2、1级负责人审核，直至最后审核全部完成。

3.2 测试结果

```
1 23:49:46.585 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结  
果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待三级审批负责人 王工"}  
2 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模  
拟三级负责人审批, 王工  
3 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结  
果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待二级审批负责人 张经理"}  
4 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模  
拟二级负责人审批, 张经理  
5 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结  
果: {"code": "0001", "info": "单号: 1000998004813441 状态: 待一级审批负责人 段总"}  
6 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: 模  
拟一级负责人审批, 段总  
7 23:49:46.590 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结  
果: {"code": "0000", "info": "单号: 1000998004813441 状态: 一级审批完成负责人 时间:  
2020-06-18 23:49:46 审批人: 段总"}  
8  
9 Process finished with exit code 0
```

- 从上述的结果可以看到我们的责任链已经生效，按照责任链的结构一层层审批，直至最后输出审批结束到一级完成的结果。
- 这样责任链的设计方式可以方便的进行扩展和维护，也把if语句干掉了。

六、总结

- 从上面代码从if语句重构到使用责任链模式开发可以看到，我们的代码结构变得清晰干净了，也解决了大量if语句的使用。并不是if语句不好，只不过if语句并不适合做系统流程设计，但是在做判断和行为逻辑处理中还是非常可以使用的。
- 在我们前面学习结构性模式中讲到过组合模式，它像是一颗组合树一样，我们搭建出一个流程决策树。其实这样的模式也是可以和责任链模型进行组合扩展使用，而这部分的重点在于如何关联链路的关联，最终的执行都是在执行在中间的关系链。
- 责任链模式很好的处理单一职责和开闭原则，简单了耦合也使对象关系更加清晰，而且外部的调用方并不需要关心责任链是如何进行处理的(以上程序中可以把责任链的组合进行包装，在提供给外部使用)。但除了这些优点外也需要是适当的场景才进行使用，避免造成性能以及编排混乱调试测试疏漏问题。

第2节：命令模式

持之以恒的重要性

初学编程往往都很懵，几乎在学习的过程中会遇到各种各样的问题，哪怕别人那运行好好的代码，但你照着写完就报错。但好在你坚持住了，否则你可能看不到这篇文章。时间和成长就是相互关联着，你在哪条路上坚持走的久，就能看见那条的终点有多美，但如果你浪费了一次又一次努力的机会，那么你也会同样错过很多机遇，因为你的路换了。坚持学习、努力成长，持之以恒的付出一定会有收获。

学习方法的重要性

不会学习往往会造成很多时间的浪费，又没有可观的收成。但不会学习有时候是因为懒造成的，尤其是学习视频、书籍资料、技术文档等，如果只是看了却不是实际操作验证，那么真的很难把别人的知识让自己吸收，即使是当时感觉会了也很快就会忘记。时而也经常会有人找到你说：“这个我不知道，你先告诉我，过后我就学。”但过后你学了吗？

你愿意为一个知识盲区付出多长时间

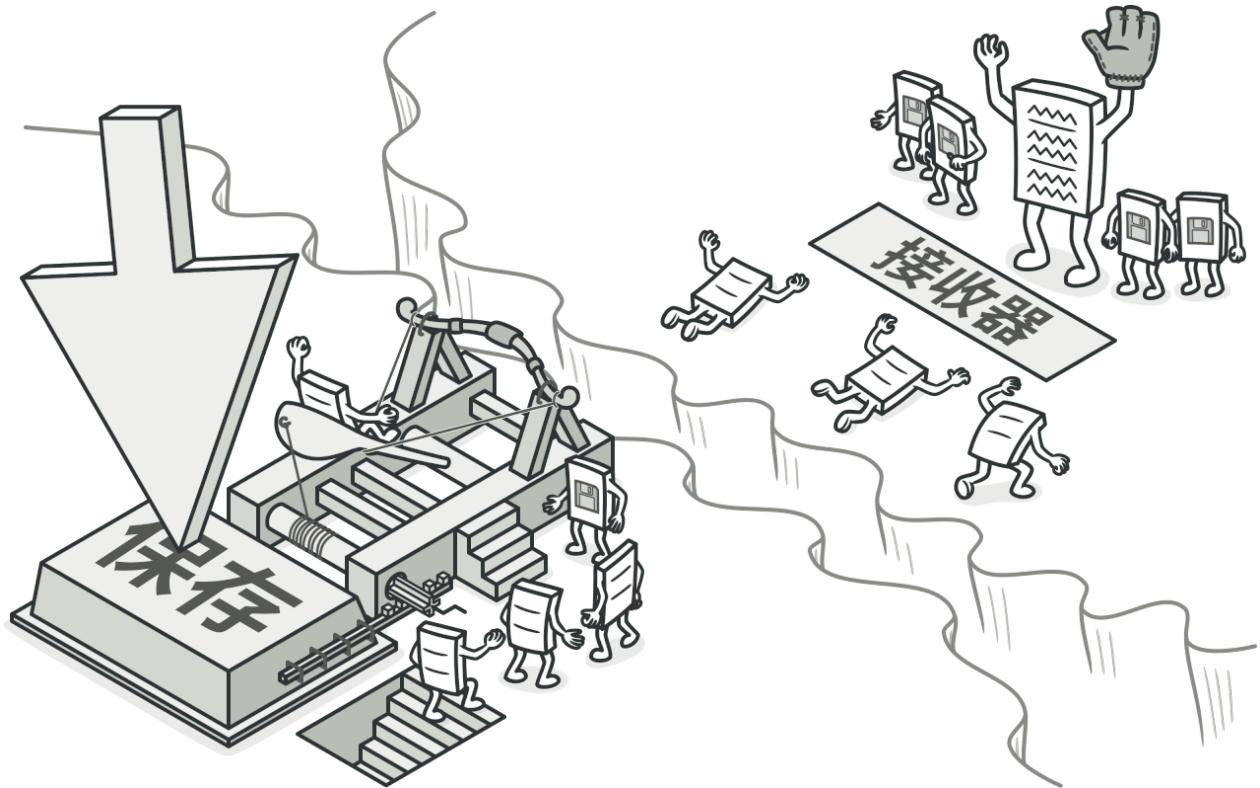
你心里时而会蹦出这样的词吗；`太难了我不会`、`找个人帮一下吧`、`放弃了放弃了`，其实谁都可能遇到很不好解决的问题，也是可以去问去咨询的。但，如果在这之前你没有在自己的大脑中反复的寻找答案，那么你的大脑中就不会形成一个凸点的知识树，缺少了这个学习过程也就缺少了查阅各种资料给自己大脑填充知识的机会，哪怕是问到了答案最终也会因时间流逝而忘记。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-14-01	使用一坨代码实现业务需求
itstack-demo-design-14-02	通过设计模式优化代码结构，增加扩展性和维护性

二、命令模式介绍



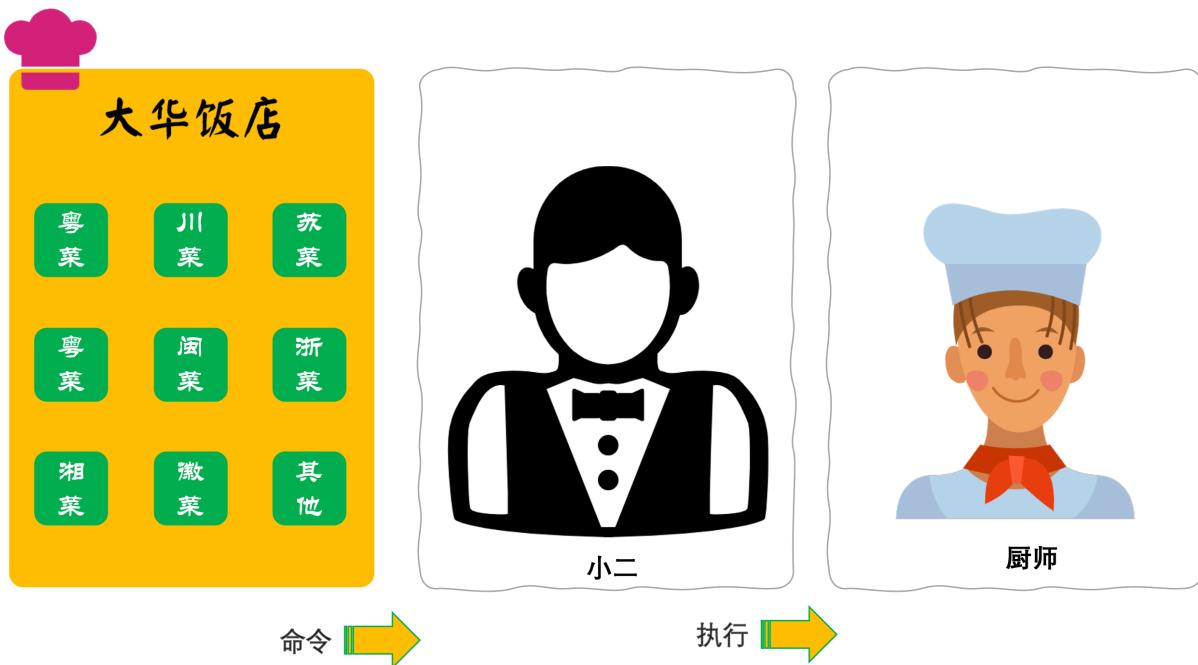
命令模式在我们通常的互联网开发中相对来说用的比较少，但这样的模式在我们的日常中却经常使用到，那就是 `Ctrl+C`、`Ctrl+V`。当然如果你开发过一些桌面应用，也会感受到这样设计模式的应用场景。从这样的模式感受上，可以想到这是把逻辑实现与操作请求进行分离，降低耦合方便扩展。

命令模式是行为模式中的一种，以数据驱动的方式将 `命令对象`，可以使用构造函数的方式传递给调用者。调用者再提供相应的实现为命令执行提供操作方法。可能会感觉这部分有一些绕，可以通过对代码的实现进行理解，在通过实操来熟练。

在这个设计模式的实现过程中有如下几个比较重要的点；

1. 抽象命令类；声明执行命令的接口和方法
2. 具体的命令实现类；接口类的具体实现，可以是一组相似的行为逻辑
3. 实现者；也就是为命令做实现的具体实现类
4. 调用者；处理命令、实现的具体操作者，负责对外提供命令服务

三、案例场景模拟



在这个案例中我们模拟在餐厅中点餐交给厨师烹饪的场景

命令场景的核心的逻辑是调用方与不需要去关心具体的逻辑实现，在这个场景中也就是点餐人员只需要把需要点的各种菜系交个 小二 就可以，小二再把各项菜品交给各个厨师进行烹饪。也就是点餐人员不需要跟各个厨师交流，只需要在统一的环境里下达命令就可以。

在这个场景中可以看到有不同的菜品；山东（鲁菜）、四川（川菜）、江苏（苏菜）、广东（粤菜）、福建（闽菜）、浙江（浙菜）、湖南（湘菜），每种菜品都会有不同的厨师 进行烹饪。而客户并不会去关心具体是谁烹饪，厨师也不会去关心谁点的餐。客户只关心早点上菜，厨师只关心还有多少个菜要做。而这中间的衔接的过程，由小二完成。

那么在这样的一个模拟场景下，可以先思考哪部分是命令模式的拆解，哪部分是命令的调用者以及命令的实现逻辑。

四、用一坨坨代码实现

不考虑设计模式的情况下，在做这样一个点单系统，有一个类就够了

像是这样一个复杂的场景，如果不知道设计模式直接开发，也是可以达到目的的。但对于后续的各项的菜品扩展、厨师实现以及如何调用上会变得非常耦合难以扩展。

1. 工程结构

```

1 itstack-demo-design-14-01
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ XiaoEr.java

```

- 这里只有一个饭店小二的类，通过这样的一个类实现整个不同菜品的点单逻辑。

2. 代码实现

```

1  public class XiaoEr {
2
3      private Logger logger = LoggerFactory.getLogger(XiaoEr.class);
4
5      private Map<Integer, String> cuisineMap = new
6      ConcurrentHashMap<Integer, String>();
7
8      public void order(int cuisine) {
9          // 广东 (粤菜)
10         if (1 == cuisine) {
11             cuisineMap.put(1, "广东厨师, 烹饪鲁菜, 宫廷最大菜系, 以孔府风味为龙
12             头");
13         }
14
15         // 江苏 (苏菜)
16         if (2 == cuisine) {
17             cuisineMap.put(2, "江苏厨师, 烹饪苏菜, 宫廷第二大菜系, 古今国宴上最受人
18             欢迎的菜系。");
19         }
20
21         // 山东 (鲁菜)
22         if (3 == cuisine) {
23             cuisineMap.put(3, "山东厨师, 烹饪鲁菜, 宫廷最大菜系, 以孔府风味为龙
24             头。");
25         }
26
27     }
28
29
30     public void placeOrder() {
31         logger.info("菜单: {}", JSON.toJSONString(cuisineMap));
32     }
33
34 }
```

- 在这个类的实现中提供了两个方法，一个方法用于点单添加菜品 `order()`，另外一个方法展示菜品的信息 `placeOrder()`。
- 从上面可以看到有比较多的if语句判断类型进行添加菜品，那么对于这样的代码后续就需要大量的经历进行维护，同时可能实际的逻辑要比这复杂的多。都写在这样一个类里会变得耦合的非常严重。

五、命令模式重构代码

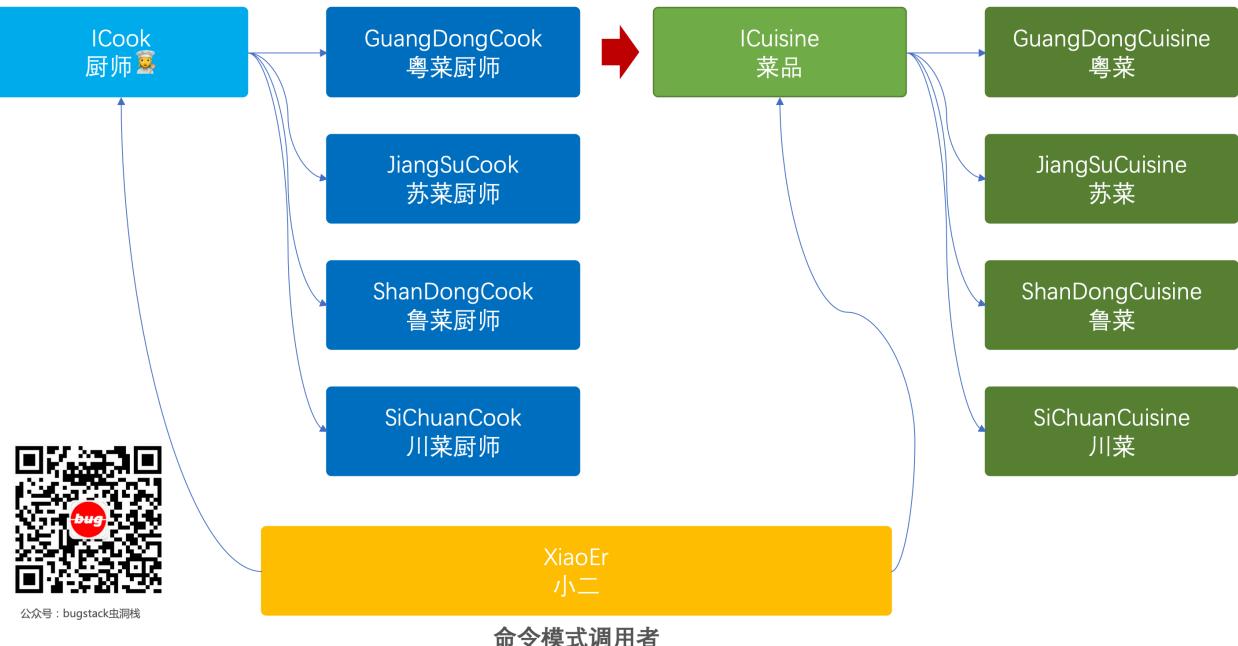
接下来使用命令模式来进行代码优化，也算是一次很小的重构。

命令模式可以将上述的模式拆解三层大块，命令、命令实现者、命令的调用者，当有新的菜品或者厨师扩充时候就可以在指定的类结构下进行实现添加即可，外部的调用也会非常的容易扩展。

1. 工程结构

```
1 itstack-demo-design-14-02
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── cook
7     │           │   └── impl
8     │           │       ├── GuangDongCook.java
9     │           │       ├── JiangSuCook.java
10    │           │       ├── ShanDongCook.java
11    │           │       └── SiChuanCook.java
12    │           └── ICook.java
13    ├── cuisine
14    │   └── impl
15    │       ├── GuangDoneCuisine.java
16    │       ├── JiangSuCuisine.java
17    │       ├── ShanDongCuisine.java
18    │       └── SiChuanCuisine.java
19    └── ICuisine.java
20
21 └── test
22     └── java
23         └── org.itstack.demo.test
24             └── ApiTest.java
```

命令模式模型结构



- 从上图可以看到整体分为三大块；命令实现(菜品)、逻辑实现(厨师)、调用者(小二)，以上这三面的实现就是命令模式的核心内容。
- 经过这样的拆解就可以非常方便的扩展菜品、厨师，对于调用者来说这部分都是松耦合的，在整体的框架下可以非常容易加入实现逻辑。

2. 代码实现

2.1 抽象命令定义(菜品接口)

```

1  /**
2   * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3   * 公众号: bugstack虫洞栈
4   * Create by 小傅哥(fustack) @2020
5   *
6   * 菜系
7   * 01、山东（鲁菜）—宫廷最大菜系，以孔府风味为龙头。
8   * 02、四川（川菜）—中国最有特色的菜系，也是民间最大菜系。
9   * 03、江苏（苏菜）—宫廷第二大菜系，古今国宴上最受人欢迎的菜系。
10  * 04、广东（粤菜）—国内民间第二大菜系，国外最有影响力的中国菜系，可以代表中国。
11  * 05、福建（闽菜）—客家菜的代表菜系。
12  * 06、浙江（浙菜）—中国最古老的菜系之一，宫廷第三大菜系。
13  * 07、湖南（湘菜）—民间第三大菜系。
14  * 08、安徽（徽菜）—徽州文化的典型代表。
15  */
16 public interface ICuisine {
17
18     void cook(); // 烹调、制作
19
20 }

```

- 这是命令接口类的定义，并提供了一个烹饪方法。后面会选四种菜品进行实现。

2.2 具体命令实现(四种菜品)

广东 (粤菜)

```
1 public class GuangDoneCuisine implements ICuisine {  
2  
3     private ICook cook;  
4  
5     public GuangDoneCuisine(ICook cook) {  
6         this.cook = cook;  
7     }  
8  
9     public void cook() {  
10        cook.doCooking();  
11    }  
12  
13 }
```

江苏 (苏菜)

```
1 public class JiangSuCuisine implements ICuisine {  
2  
3     private ICook cook;  
4  
5     public JiangSuCuisine(ICook cook) {  
6         this.cook = cook;  
7     }  
8  
9     public void cook() {  
10        cook.doCooking();  
11    }  
12  
13 }
```

山东 (鲁菜)

```
1 public class ShanDongCuisine implements ICuisine {  
2  
3     private ICook cook;  
4  
5     public ShanDongCuisine(ICook cook) {  
6         this.cook = cook;  
7     }  
8  
9     public void cook() {  
10        cook.doCooking();  
11    }  
12  
13 }
```

四川 (川菜)

```
1 public class SichuanCuisine implements ICuisine {  
2  
3     private ICook cook;  
4  
5     public SichuanCuisine(ICook cook) {  
6         this.cook = cook;  
7     }  
8  
9     public void cook() {  
10        cook.doCooking();  
11    }  
12}  
13}
```

- 以上是四种菜品的实现，在实现的类中都有添加了一个厨师类(`ICook`)，并通过这个类提供的方法进行操作命令(烹饪菜品)`cook.doCooking()`。
- 命令的实现过程可以是按照逻辑进行添加补充，目前这里抽象的比较简单，只是模拟一个烹饪的过程，相当于同时厨师进行菜品烹饪。

2.3 抽象实现者定义(厨师接口)

```
1 public interface ICook {  
2  
3     void doCooking();  
4  
5 }
```

- 这里定义的是具体的为命令的实现者，这里也就是菜品对应的厨师烹饪的指令实现。

2.4 实现者具体实现(四类厨师)

粤菜，厨师

```
1 public class GuangDongCook implements ICook {  
2  
3     private Logger logger = LoggerFactory.getLogger(ICook.class);  
4  
5     public void doCooking() {  
6         logger.info("广东厨师，烹饪鲁菜，宫廷最大菜系，以孔府风味为龙头");  
7     }  
8  
9 }
```

苏菜，厨师

```
1 public class JiangSuCook implements ICook {  
2  
3     private Logger logger = LoggerFactory.getLogger(ICook.class);  
4  
5     public void doCooking() {  
6         logger.info("江苏厨师, 烹饪苏菜, 宫廷第二大菜系, 古今国宴上最受人欢迎的菜  
系。");  
7     }  
8  
9 }
```

鲁菜，厨师

```
1 public class ShanDongCook implements ICook {  
2  
3     private Logger logger = LoggerFactory.getLogger(ICook.class);  
4  
5     public void doCooking() {  
6         logger.info("山东厨师, 烹饪鲁菜, 宫廷最大菜系, 以孔府风味为龙头");  
7     }  
8  
9 }
```

苏菜，厨师

```
1 public class SiChuanCook implements ICook {  
2  
3     private Logger logger = LoggerFactory.getLogger(ICook.class);  
4  
5     public void doCooking() {  
6         logger.info("四川厨师, 烹饪川菜, 中国最有特色的菜系, 也是民间最大菜系。");  
7     }  
8  
9 }
```

- 这里是四类不同菜品的厨师 ，在这个实现的过程是模拟打了日志，相当于通知了厨房里具体的厨师进行菜品烹饪。
- 从以上可以看到，当我们需要进行扩从的时候是可以非常方便的进行添加的，每一个类都具备了单一职责原则。

2.5 调用者(小二)

```
1 public class XiaoEr {  
2  
3     private Logger logger = LoggerFactory.getLogger(XiaoEr.class);  
4  
5     private List<ICuisine> cuisineList = new ArrayList<ICuisine>();  
6 }
```

```

7     public void order(ICuisine cuisine) {
8         cuisineList.add(cuisine);
9     }
10
11    public synchronized void placeOrder() {
12        for (ICuisine cuisine : cuisineList) {
13            cuisine.cook();
14        }
15        cuisineList.clear();
16    }
17
18 }
```

- 在调用者的具体实现中，提供了菜品的添加和菜单执行烹饪。这个过程是命令模式的具体调用，通过外部将菜品和厨师传递进来而进行具体的调用。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test(){
3
4      // 菜系 + 厨师；广东（粤菜）、江苏（苏菜）、山东（鲁菜）、四川（川菜）
5      ICuisine guangDoneCuisine = new GuangDoneCuisine(new GuangDongCook());
6      JiangSuCuisine jiangSuCuisine = new JiangSuCuisine(new JiangSuCook());
7      ShanDongCuisine shanDongCuisine = new ShanDongCuisine(new
8          ShanDongCook());
9      SiChuanCuisine siChuanCuisine = new SiChuanCuisine(new SiChuanCook());
10
11     // 点单
12     XiaoEr xiaoEr = new XiaoEr();
13     xiaoEr.order(guangDoneCuisine);
14     xiaoEr.order(jiangSuCuisine);
15     xiaoEr.order(shanDongCuisine);
16     xiaoEr.order(siChuanCuisine);
17
18     // 下单
19     xiaoEr.placeOrder();
}
```

- 这里可以主要观察 `菜品` 与 `厨师` 的组合；`new GuangDoneCuisine(new GuangDongCook());`，每一个具体的命令都拥有一个对应的实现类，可以进行组合。
- 当菜品和具体的实现定义完成后，由小二进行操作点单，`xiaoEr.order(guangDoneCuisine);`，这里分别添加了四种菜品，给小二。
- 最后是下单，这个是具体命令实现的操作，相当于把小二手里的菜单传递给厨师。当然这里也可以提供删除和撤销，也就是客户取消了自己的某个菜品。

3.2 测试结果

```
1 22:12:13.056 [main] INFO org.itstack.demo.design.cook.ICook - 广东厨师, 烹饪  
鲁菜, 宫廷最大菜系, 以孔府风味为龙头  
2 22:12:13.059 [main] INFO org.itstack.demo.design.cook.ICook - 江苏厨师, 烹饪  
苏菜, 宫廷第二大菜系, 古今国宴上最受人欢迎的菜系。  
3 22:12:13.059 [main] INFO org.itstack.demo.design.cook.ICook - 山东厨师, 烹饪  
鲁菜, 宫廷最大菜系, 以孔府风味为龙头  
4 22:12:13.059 [main] INFO org.itstack.demo.design.cook.ICook - 四川厨师, 烹饪  
川菜, 中国最有特色的菜系, 也是民间最大菜系。  
5  
6 Process finished with exit code 0
```

- 从上面的测试结果可以看到, 我们已经交给调用者(小二)的点单, 由不同的厨师具体实现(烹饪)。
- 此外当我们需要不同的菜品时候或者修改时候都可以非常方便的添加和修改, 在具备单一职责的类下, 都可以非常方便的扩展。

六、总结

- 从以上的内容和例子可以感受到, 命令模式的使用场景需要分为三个比较大的块; 命令、实现、调用者, 而这三块内容的拆分也是选择适合场景的关键因素, 经过这样的拆分可以让逻辑具备单一职责的性质, 便于扩展。
- 通过这样的实现方式与if语句相比, 降低了耦合性也方便其他的命令和实现的扩展。但同时这样的设计模式也带来了一点问题, 就是在各种命令与实现的组合下, 会扩展出很多的实现类, 需要进行管理。
- 设计模式的学习一定要勤加练习, 哪怕最开始是模仿实现也是可以的, 多次的练习后再去找到一些可以优化的场景, 并逐步运用到自己的开发中。提升自己对代码的设计感觉, 让代码结构更加清晰易扩展。

第3节：迭代器模式

相信相信的力量!

从懵懂的少年, 到拿起键盘, 可以写一个HelloWorld。多数人在这并不会感觉有多难, 也不会认为做不出来。因为这样的例子, 有老师的指导、有书本的例子、有前人的经验。但随着你的开发时间越来越长, 要解决更复杂的问题或者技术创新, 因此在网上搜了几天几夜都没有答案, 这个时候是否想过放弃, 还是一直坚持不断的尝试一点点完成自己心里要的结果。往往这种没有前车之鉴需要自己解决问题的时候, 可能真的会折磨到要崩溃, 但你要愿意执着、愿意倔强, 愿意选择相信相信的力量, 就一定能解决。哪怕解决不了, 也可以在这条路上摸索出其他更多的收获, 为后续前进的道路填充好垫脚石。

时间紧是写垃圾代码的理由？

拧螺丝？Ctrl+C、Ctrl+V？贴膏药一样写代码？没有办法，没有时间，往往真的是借口，胸中没用笔墨，才只能凑合。难道一定是好好写代码就浪费时间，拼凑CRUD就快吗，根本不可能的。因为不会，没用实操过，很少架构出全场景的设计，才很难写出优良的代码。多增强自身的编码(武术)修为，在各种编码场景中让自己变得老练，才好应对紧急情况下的需求开发和人员安排。就像韩信一样有谋有略，才能执掌百万雄兵。

不要只是做个工具人！

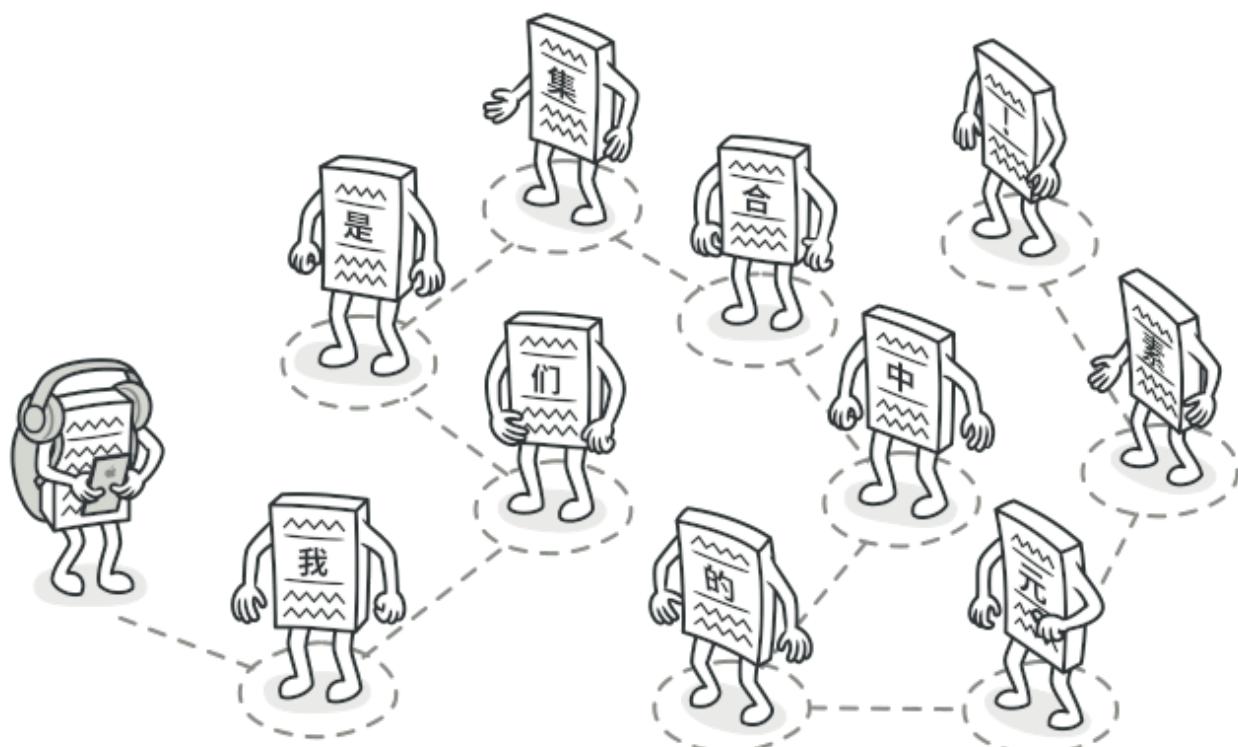
因为日常的编写简单业务需求，导致自己像个工具人一样，日久天长的也就很少去深入学习更多技术栈。看见有工具、有组件、有框架，拿来就用用，反正没什么体量也不会出什么问题。但如果你想要更多的收入，哪怕是重复的造轮子，你也要去尝试造一个，就算不用到生产，自己玩玩总可以吧。有些事情只有自己经历过，才能有最深的感触，参与过实践过，才好总结点评学习。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程一个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-15-00	开发树形组织架构关系迭代器

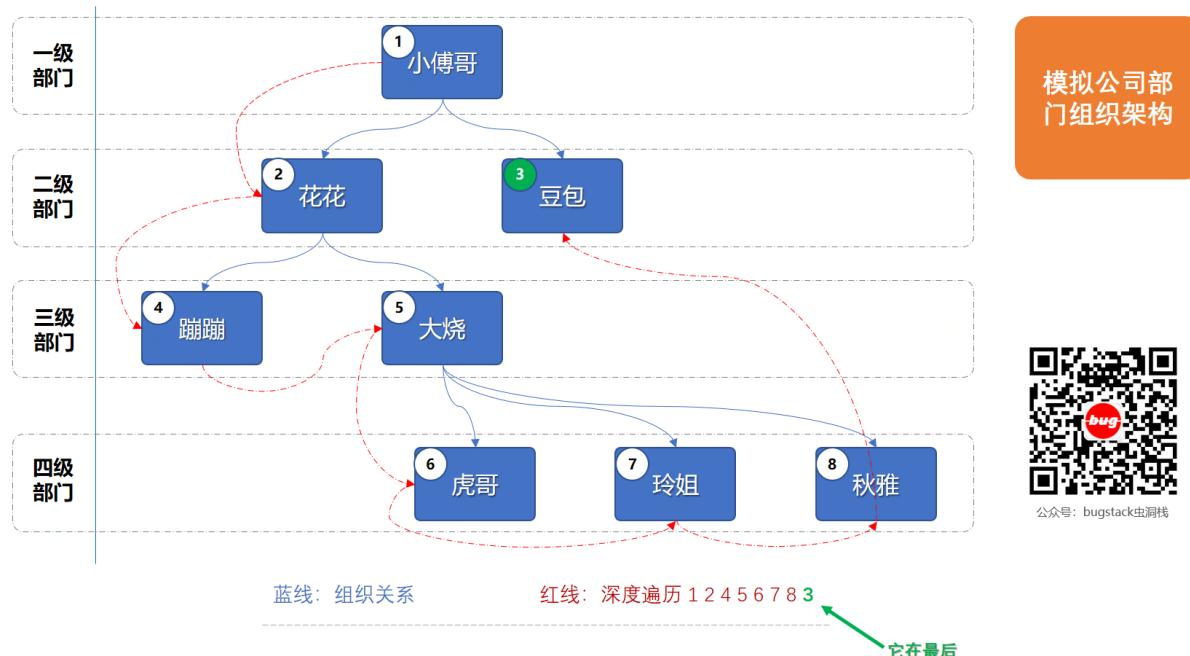
二、迭代器模式介绍



迭代器模式，常见的就是我们日常使用的 `Iterator` 遍历。虽然这个设计模式在我们的实际业务开发中的场景并不多，但却几乎每天都要使用 `jdk` 为我们提供的 `List` 集合遍历。另外增强的 `for` 循环虽然是循环输出数据，但是他不是迭代器模式。迭代器模式的特点是实现 `Iterable` 接口，通过 `next` 的方式获取集合元素，同时具备对元素的删除等操作。而增强的 `for` 循环是不可以的。

这种设计模式的优点是可以让我们以相同的方式，遍历不同的数据结构元素，这些数据结构包括：数组、链表、树等，而用户在使用遍历的时候并不需要去关心每一种数据结构的遍历处理逻辑，从而让使用变得统一易用。

三、案例场景模拟



在本案例中我们模拟迭代遍历输出公司中树形结构的组织架构关系中雇员列表

大部分公司的组织架构都是金字塔结构，也就这种树形结构，分为一级、二级、三级等部门，每个组织部门由雇员填充，最终体现出一个整体的树形组织架构关系。

一般我们常用的遍历就是 `jdk` 默认提供的方法，对 `List` 集合遍历。但是对于这样的偏业务特性较大的树形结构，如果需要使用到遍历，那么就可以自己来实现。接下来我们会把这个组织层次关系通过树形数据结构来实现，并完成迭代器功能。

四、迭代器模式遍历组织结构

在实现迭代器模式之前可以先阅读下 `java` 中 `List` 方法关于 `Iterator` 的实现部分，几乎所有的迭代器开发都会按照这个模式来实现，这个模式主要分为以下几块：

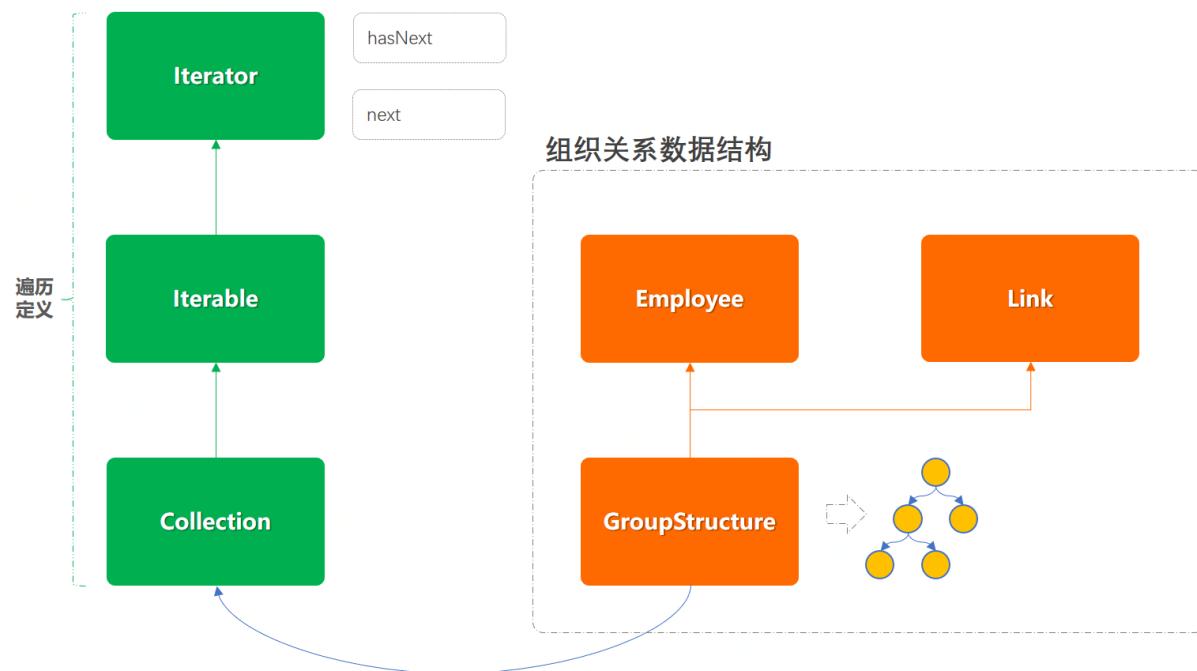
1. `Collection`，集合方法部分用于对自定义的数据结构添加通用方法；`add`、`remove`、`iterator` 等核心方法。
2. `Iterable`，提供获取迭代器，这个接口类会被 `Collection` 继承。
3. `Iterator`，提供了两个方法的定义；`hasNext`、`next`，会在具体的数据结构中写实现方式。

除了这样通用的迭代器实现方式外，我们的组织关系结构树，是由节点和节点间的关系链构成，所以会比上述的内容多一些入参。

1. 工程结构

```
1 itstack-demo-design-15-00
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── group
7     │           │   └── Employee.java
8     │           │   └── GroupStructure.java
9     │           └── lang
10    │               └── Collection.java
11    │               └── Iterable.java
12    │               └── Iterator.java
13    └── test
14        └── java
15            └── org.itstack.demo.design.test
16                └── ApiTest.java
```

迭代器模式模型结构



- 以上是我们工程类图的模型结构，左侧是对迭代器的定义，右侧是在数据结构中实现迭代器功能。
- 关于左侧部分的实现与jdk中的方式是一样的，所以在学习的过程中可以互相参考，也可以自己扩展学习。
- 另外这个遍历方式一个树形结构的深度遍历，为了可以更加让学习的小伙伴容易理解，这里我实现了一种比较简单的树形结构深度遍历方式。后续读者也可以把遍历扩展为横向遍历也就是宽度遍历。

2. 代码实现

2.1 雇员实体类

```
1  /**
2  * 雇员
3  */
4  public class Employee {
5
6      private String uid;    // ID
7      private String name;  // 姓名
8      private String desc;  // 备注
9
10     // ...get/set
11 }
```

- 这是一个简单的雇员类，也就是公司员工的信息类，包括必要的信息；id、姓名、备注。

2.2 树节点链路

```
1  /**
2  * 树节点链路
3  */
4  public class Link {
5
6      private String fromId; // 雇员ID
7      private String toId;   // 雇员ID
8
9      // ...get/set
10 }
```

- 这个类用于描述结构树中的各个节点之间的关系链，也就是A to B、B to C、B to D，以此描述出一套完整的树组织结构。

2.3 迭代器定义

```
1  public interface Iterator<E> {
2
3      boolean hasNext();
4
5      E next();
6
7  }
```

- 这里的这个类和java的jdk中提供的是一样的，这样也方便后续读者可以对照list的Iterator进行源码学习。
- 方法描述：hasNext，判断是否有下一个元素、next，获取下一个元素。这个在list的遍历中是经常用到的。

2.4 可迭代接口定义

```
1 public interface Iterable<E> {
2
3     Iterator<E> iterator();
4
5 }
```

- 这个接口中提供了上面迭代器的实现 `Iterator` 的获取，也就是后续在自己的数据结构中需要实现迭代器的功能并交给 `Iterable`，由此让外部调用方进行获取使用。

2.5 集合功能接口定义

```
1 public interface Collection<E, L> extends Iterable<E> {
2
3     boolean add(E e);
4
5     boolean remove(E e);
6
7     boolean addLink(String key, L l);
8
9     boolean removeLink(String key);
10
11    Iterator<E> iterator();
12
13 }
```

- 这里我们定义集合操作接口；`Collection`，同时继承了另外一个接口 `Iterable` 的方法 `iterator()`。这样后续谁来实现这个接口，就需要实现上述定义的一些基本功能；添加元素、删除元素、遍历。
- 同时你可能注意到这里定义了两个泛型 `<E, L>`，因为我们的数据结构一个是用于添加元素，另外一个是用于添加树节点的链路关系。

2.6 (核心)迭代器功能实现

```
1 public class GroupStructure implements Collection<Employee, Link> {
2
3     private String groupId;
4         // 组织ID，也是一个组织链的头部ID
5     private String groupName;
6         // 组织名称
7     private Map<String, Employee> employeeMap = new
8     ConcurrentHashMap<String, Employee>(); // 雇员列表
9     private Map<String, List<Link>> linkMap = new
10    ConcurrentHashMap<String, List<Link>>(); // 组织架构关系；id->list
11    private Map<String, String> invertedMap = new
12    ConcurrentHashMap<String, String>(); // 反向关系链
13
14
15    public GroupStructure(String groupId, String groupName) {
16        this.groupId = groupId;
```

```
11     this.groupName = groupName;
12 }
13
14 public boolean add(Employee employee) {
15     return null != employeeMap.put(employee.getId(), employee);
16 }
17
18 public boolean remove(Employee o) {
19     return null != employeeMap.remove(o.getId());
20 }
21
22 public boolean addLink(String key, Link link) {
23     invertedMap.put(link.getToId(), link.getFromId());
24     if (linkMap.containsKey(key)) {
25         return linkMap.get(key).add(link);
26     } else {
27         List<Link> links = new LinkedList<Link>();
28         links.add(link);
29         linkMap.put(key, links);
30         return true;
31     }
32 }
33
34 public boolean removeLink(String key) {
35     return null != linkMap.remove(key);
36 }
37
38 public Iterator<Employee> iterator() {
39
40     return new Iterator<Employee>() {
41
42         HashMap<String, Integer> keyMap = new HashMap<String, Integer>();
43
44         int totalIdx = 0;
45         private String fromId = groupId; // 雇员ID, From
46         private String toId = groupId; // 雇员ID, To
47
48         public boolean hasNext() {
49             return totalIdx < employeeMap.size();
50         }
51
52         public Employee next() {
53             List<Link> links = linkMap.get(toId);
54             int cursorIdx = getCursorIdx(toId);
55
56             // 同级节点扫描
57             if (null == links) {
58                 cursorIdx = getCursorIdx(fromId);
```

```

59             links = linkMap.get(fromId);
60         }
61
62         // 上级节点扫描
63         while (cursorIdx > links.size() - 1) {
64             fromId = invertedMap.get(fromId);
65             cursorIdx = getCursorIdx(fromId);
66             links = linkMap.get(fromId);
67         }
68
69         // 获取节点
70         Link link = links.get(cursorIdx);
71         toId = link.getToId();
72         fromId = link.getFromId();
73         totalIdx++;
74
75         // 返回结果
76         return employeeMap.get(link.getToId());
77     }
78
79     // 给每个层级定义宽度遍历进度
80     public int getCursorIdx(String key) {
81         int idx = 0;
82         if (keyMap.containsKey(key)) {
83             idx = keyMap.get(key);
84             keyMap.put(key, ++idx);
85         } else {
86             keyMap.put(key, idx);
87         }
88         return idx;
89     }
90 };
91 }
92 }
93 }
```

- 以上的这部分代码稍微有点长，主要包括了对元素的添加和删除。另外最重要的是对遍历的实现 `new Iterator<Employee>`。
- 添加和删除元素相对来说比较简单，使用了两个map数组结构进行定义；`雇员列表`、`组织架构关系`；`id->list`。当元素添加元素的时候，会分别在不同的方法中向 `map` 结构中进行填充**指向关系(A->B)**，也就构建出了我们的树形组织关系。

迭代器实现思路

- 这里的树形结构我们需要做的是深度遍历，也就是左侧的一直遍历到最深节点。
- 当遍历到最深节点后，开始遍历最深节点的横向节点。
- 当横向节点遍历完成后则向上寻找横向节点，直至树结构全部遍历完成。

3. 测试验证

3.1 编写测试类

```
1  @Test
2  public void test_iterator() {
3      // 数据填充
4      GroupStructure groupStructure = new GroupStructure("1", "小傅哥");
5
6      // 雇员信息
7      groupStructure.add(new Employee("2", "花花", "二级部门"));
8      groupStructure.add(new Employee("3", "豆包", "二级部门"));
9      groupStructure.add(new Employee("4", "蹦蹦", "三级部门"));
10     groupStructure.add(new Employee("5", "大烧", "三级部门"));
11     groupStructure.add(new Employee("6", "虎哥", "四级部门"));
12     groupStructure.add(new Employee("7", "玲姐", "四级部门"));
13     groupStructure.add(new Employee("8", "秋雅", "四级部门"));
14
15     // 节点关系 1->(1,2) 2->(4,5)
16     groupStructure.addLink("1", new Link("1", "2"));
17     groupStructure.addLink("1", new Link("1", "3"));
18     groupStructure.addLink("2", new Link("2", "4"));
19     groupStructure.addLink("2", new Link("2", "5"));
20     groupStructure.addLink("5", new Link("5", "6"));
21     groupStructure.addLink("5", new Link("5", "7"));
22     groupStructure.addLink("5", new Link("5", "8"));
23
24     Iterator<Employee> iterator = groupStructure.iterator();
25     while (iterator.hasNext()) {
26         Employee employee = iterator.next();
27         logger.info("{}，雇员 Id: {} Name: {}", employee.getDesc(),
28                     employee.getId(), employee.getName());
29     }
}
```

3.2 测试结果

```
1 22:23:37.166 [main] INFO org.itstack.demo.design.test.ApiTest - 二级部门, 雇  
员 Id: 2 Name: 花花  
2 22:23:37.168 [main] INFO org.itstack.demo.design.test.ApiTest - 三级部门, 雇  
员 Id: 4 Name: 蹦蹦  
3 22:23:37.169 [main] INFO org.itstack.demo.design.test.ApiTest - 三级部门, 雇  
员 Id: 5 Name: 大烧  
4 22:23:37.169 [main] INFO org.itstack.demo.design.test.ApiTest - 四级部门, 雇  
员 Id: 6 Name: 虎哥  
5 22:23:37.169 [main] INFO org.itstack.demo.design.test.ApiTest - 四级部门, 雇  
员 Id: 7 Name: 玲姐  
6 22:23:37.169 [main] INFO org.itstack.demo.design.test.ApiTest - 四级部门, 雇  
员 Id: 8 Name: 秋雅  
7 22:23:37.169 [main] INFO org.itstack.demo.design.test.ApiTest - 二级部门, 雇  
员 Id: 3 Name: 豆包  
8  
9 Process finished with exit code 0
```

- 从遍历的结果可以看到，我们是顺着树形结构的深度开始遍历，一直到右侧的节点3；雇员 Id: 2、雇员 Id: 4...雇员 Id: 3

五、总结

- 迭代器的设计模式从以上的功能实现可以看到，满足了单一职责和开闭原则，外界的调用方也不需要知道任何一个不同的数据结构在使用上的遍历差异。可以非常方便的扩展，也让整个遍历变得更加干净整洁。
- 但从结构的实现上可以看到，迭代器模式的实现过程相对来说是比较负责的，类的实现上也扩增了需要外部定义的类，使得遍历与原数据结构分开。虽然这是比较麻烦的，但可以看到在使用java的jdk时候，迭代器的模式还是很好用的，可以非常方便扩展和升级。
- 以上的设计模式场景实现过程可能对新人有一些不好理解点，包括：迭代器三个和接口的定义、树形结构的数据关系、树结构深度遍历思路。这些都需要反复实现练习才能深入的理解，事必躬亲，亲历亲为，才能让自己掌握这些知识。

第4节：中介者模式

同龄人的差距是从什么时候拉开的

同样的幼儿园、同样的小学、一样的书本、一样的课堂，有人学习好、有人学习差。不只是上学，几乎人生处处都是赛道，发令枪响起的时刻，也就把人生的差距拉开。编程开发这条路也是很长很宽，有人跑得快有人跑得慢。那么你是否想起过，这一点点的差距到遥不可及的距离，是从哪一天开始的。摸摸肚子的肉，看看远处的路，别人讲的是故事，你想起的都是事故。

思想没有产品高才写出一片的if else

当你承接一个需求的时候，比如：交易、订单、营销、保险等各类场景。如果你不熟悉这个场景下的业务模式，以及将来的拓展方向，那么很难设计出良好可扩展的系统。再加上产品功能初建，说老板要的急，尽快上线。作为程序员的你更没有时间思考，整体一看现在的需求也不难，直接上手开干(一个方法两个if语句)，这样确实满足了当前需求。但老板的想法多呀，产品也跟着变化快，到你这就是改改改，加加加。当然你也不客气，回首掏就是1024个if语句！

日积月累的技术沉淀是为了厚积薄发

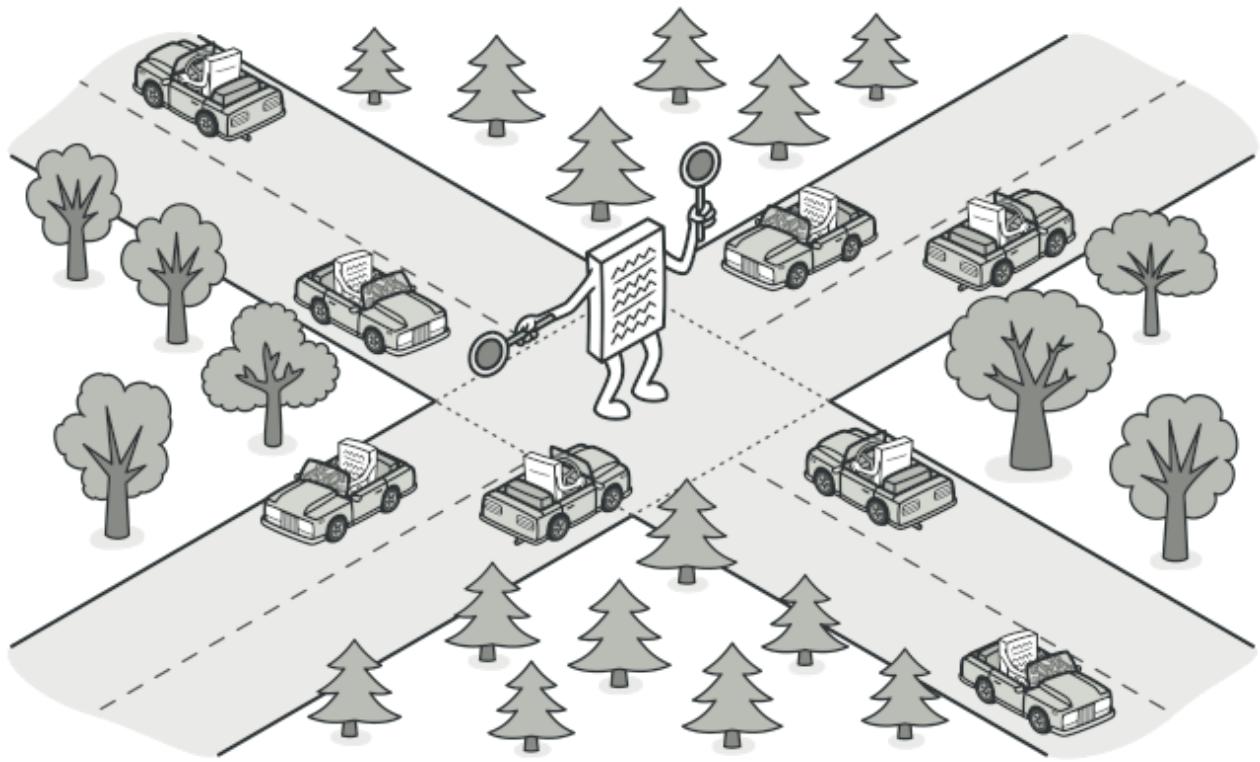
粗略的估算过，如果从上大学开始每天写 200 行，一个月是 6000 行，一年算10个月话，就是6万行，第三年出去实习的是时候就有 20 万行的代码量。如果你能做到这一点，找工作难？有时候很多事情就是靠时间积累出来的，想走捷径有时候真的没有。你的技术水平、你的业务能力、你身上的肉，都是一点点积累下来的，不要浪费看似很短的时间，一年年坚持下来，留下印刻青春的痕迹，多给自己武装上一些能力。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. mysql 5.1.20
4. 涉及工程一个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-16-01	使用JDBC方式连接数据库
itstack-demo-design-16-02	手写ORM框架操作数据库

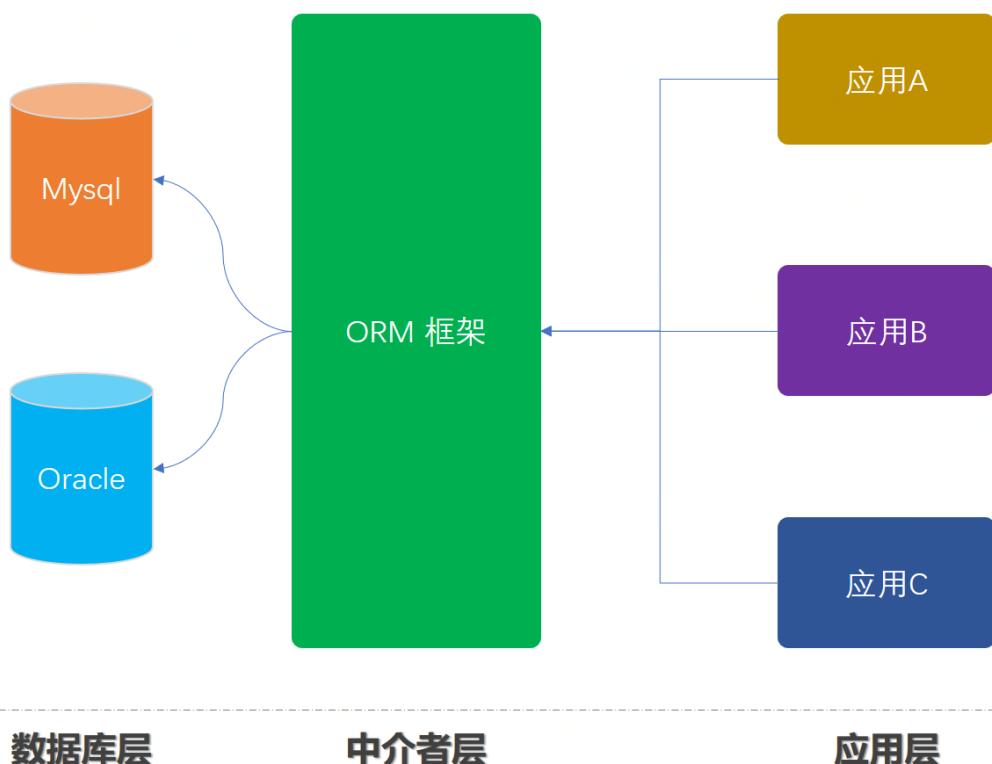
二、中介者模式介绍



中介者模式要解决的就是复杂功能应用之间的重复调用，在这中间添加一层中介者包装服务，对外提供简单、通用、易扩展的服务能力。

这样的设计模式几乎在我们日常生活和实际业务开发中都会见到，例如：飞机降落有小姐姐在塔台喊话、无论哪个方向来的候车都从站台上下、公司的系统中有一个中台专门为你包装所有接口和提供统一的服务等等，这些都运用了中介者模式。除此之外，你用到的一些中间件，他们包装了底层多种数据库的差异化，提供非常简单的方式进行使用。

三、案例场景模拟



在本案例中我们通过模仿Mybatis手写ORM框架，通过这样操作数据库学习中介者运用场景

除了这样的中间件层使用场景外，对于一些外部接口，例如N种奖品服务，可以由中台系统进行统一包装对外提供服务能力。也是中介者模式的一种思想体现。

在本案例中我们会把jdbc层进行包装，让用户在使用数据库服务的时候，可以和使用mybatis一样简单方便，通过这样的源码方式学习中介者模式，也方便对源码知识的拓展学习，增强知识栈。

四、用一坨坨代码实现

这是一种关于数据库操作最初的方式

基本上每一个学习开发的人都学习过直接使用jdbc方式连接数据库，进行CRUD操作。以下的例子可以当做回忆。

1. 工程结构

```
1 itstack-demo-design-16-01
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 └── JDBCUtil.java
```

- 这里的类比较简单只包括了一个数据库操作类。

2. 代码实现

```
1 public class JDBCUtil {
2
3     private static Logger logger =
4         LoggerFactory.getLogger(JDBCUtil.class);
5
6     public static final String URL = "jdbc:mysql://127.0.0.1:3306/itstack-
7         demo-design";
8     public static final String USER = "root";
9     public static final String PASSWORD = "123456";
10
11    public static void main(String[] args) throws Exception {
12        //1. 加载驱动程序
13        Class.forName("com.mysql.jdbc.Driver");
14        //2. 获得数据库连接
15        Connection conn = DriverManager.getConnection(URL, USER,
16            PASSWORD);
17        //3. 操作数据库
18        Statement stmt = conn.createStatement();
19        ResultSet resultSet = stmt.executeQuery("SELECT id, name, age,
20            createTime, updateTime FROM user");
21        //4. 如果有数据 resultSet.next() 返回true
22        while (resultSet.next()) {
```

```
19         logger.info("测试结果 姓名: {} 年龄: {}",  
20             resultSet.getString("name"), resultSet.getInt("age"));  
21     }  
22 }  
23 }
```

- 以上是使用JDBC的方式进行直接操作数据库，几乎大家都使用过这样的方式。

3. 测试结果

```
1 15:38:10.919 [main] INFO org.itstack.demo.design.JDBCUtil - 测试结果 姓名: 水  
水 年龄: 18  
2 15:38:10.922 [main] INFO org.itstack.demo.design.JDBCUtil - 测试结果 姓名: 豆  
豆 年龄: 18  
3 15:38:10.922 [main] INFO org.itstack.demo.design.JDBCUtil - 测试结果 姓名: 花  
花 年龄: 19  
4  
5 Process finished with exit code 0
```

- 从测试结果可以看到这里已经查询到了数据库中的数据。只不过如果在全部的业务开发中都这样实现，会非常的麻烦。

五、中介模式开发ORM框架

接下来就使用中介模式的思想完成模仿Mybatis的ORM框架开发~

1. 工程结构

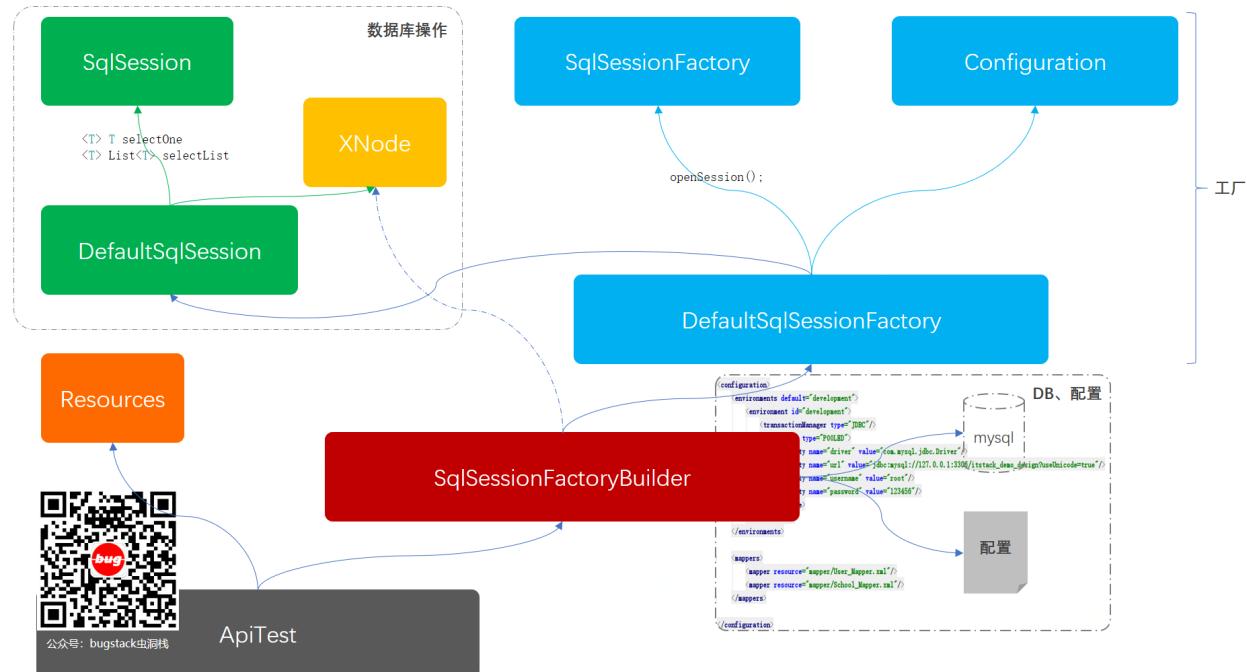
```
1 itstack-demo-design-16-02  
2 └── src  
3     ├── main  
4     │   └── java  
5     │       └── org.itstack.demo.design  
6     │           ├── dao  
7     │           │   └── ISchool.java  
8     │           │   └── IUserDao.java  
9     │           ├── mediator  
10    │           │   └── Configuration.java  
11    │           │   └── DefaultSqlSession.java  
12    │           │   └── DefaultSqlSessionFactory.java  
13    │           │   └── Resources.java  
14    │           │   └── SqlSession.java  
15    │           │   └── SqlSessionFactory.java  
16    │           │   └── SqlSessionFactoryBuilder.java  
17    │           │   └── SqlSessionFactoryBuilder.java  
18    │           └── po  
19        └── School.java  
20        └── User.java
```

```

21 |     └── resources
22 |         ├── mapper
23 |         |   ├── School_Mapper.xml
24 |         |   └── User_Mapper.xml
25 |         └── mybatis-config-datasource.xml
26 |
27 └── test
28     └── java
29         └── org.itstack.demo.design.test
            └── ApiTest.java

```

中介者模式模型结构



- 以上是对ORM框架实现的核心类，包括了；加载配置文件、对xml解析、获取数据库session、操作数据库以及结果返回。
- 左上是对数据库的定义和处理，基本包括我们常用的方法；`<T> T selectOne`、`<T> List<T> selectList`等。
- 右侧蓝色部分是对数据库配置的开启session的工厂处理类，这里的工厂会操作 `DefaultSqlSession`
- 之后是红色地方的 `SqlSessionFactoryBuilder`，这个类是对数据库操作的核心类；处理工厂、解析文件、拿到session等。

接下来我们就分别介绍各个类的功能实现过程。

2. 代码实现

2.1 定义SqlSession接口

```

1  public interface SqlSession {
2
3      <T> T selectOne(String statement);
4
5      <T> T selectOne(String statement, Object parameter);
6
7      <T> List<T> selectList(String statement);
8
9      <T> List<T> selectList(String statement, Object parameter);
10
11     void close();
12 }

```

- 这里定义了对数据库操作的查询接口，分为查询一个结果和查询多个结果，同时包括有参数和没有参数的方法。

2.2 SqlSession具体实现类

```

1  public class DefaultSqlSession implements SqlSession {
2
3      private Connection connection;
4      private Map<String, XNode> mapperElement;
5
6      public DefaultSqlSession(Connection connection, Map<String, XNode>
mapperElement) {
7          this.connection = connection;
8          this.mapperElement = mapperElement;
9      }
10
11     @Override
12     public <T> T selectOne(String statement) {
13         try {
14             XNode xNode = mapperElement.get(statement);
15             PreparedStatement preparedStatement =
connection.prepareStatement(xNode.getSql());
16             ResultSet resultSet = preparedStatement.executeQuery();
17             List<T> objects = resultSet2Obj(resultSet,
Class.forName(xNode.getResultType()));
18             return objects.get(0);
19         } catch (Exception e) {
20             e.printStackTrace();
21         }
22         return null;
23     }
24
25     @Override
26     public <T> List<T> selectList(String statement) {
27         XNode xNode = mapperElement.get(statement);
28         try {

```

```
29         PreparedStatement preparedStatement =
30             connection.prepareStatement(xNode.getSql());
31         ResultSet resultSet = preparedStatement.executeQuery();
32         return resultSet2Obj(resultSet,
33             Class.forName(xNode.getResultType()));
34     } catch (Exception e) {
35         e.printStackTrace();
36     }
37
38 // ...
39
40     private <T> List<T> resultSet2Obj(ResultSet resultSet, Class<?> clazz)
41     {
42         List<T> list = new ArrayList<>();
43         try {
44             ResultSetMetaData metaData = resultSet.getMetaData();
45             int columnCount = metaData.getColumnCount();
46             // 每次遍历行值
47             while (resultSet.next()) {
48                 T obj = (T) clazz.newInstance();
49                 for (int i = 1; i <= columnCount; i++) {
50                     Object value = resultSet.getObject(i);
51                     String columnName = metaData.getColumnName(i);
52                     String setMethod = "set" + columnName.substring(0,
53                         1).toUpperCase() + columnName.substring(1);
54                     Method method;
55                     if (value instanceof Timestamp) {
56                         method = clazz.getMethod(setMethod, Date.class);
57                     } else {
58                         method = clazz.getMethod(setMethod,
59                             value.getClass());
60                     }
61                     method.invoke(obj, value);
62                 }
63                 list.add(obj);
64             }
65         } catch (Exception e) {
66             e.printStackTrace();
67         }
68     }
69     @Override
70     public void close() {
71         if (null == connection) return;
72         try {
73             connection.close();
```

```
73     } catch (SQLException e) {
74         e.printStackTrace();
75     }
76 }
77 }
```

- 这里包括了接口定义的方法实现，也就是包装了jdbc层。
- 通过这样的包装可以让对数据库的jdbc操作隐藏起来，外部调用的时候对入参、出参都有内部进行处理。

2.3 定义SqlSessionFactory接口

```
1 public interface SqlSessionFactory {
2
3     SqlSession openSession();
4
5 }
```

- 开启一个`SqlSession`，这几乎是大家在平时的使用中都需要进行操作的内容。虽然你看不见，但是当你有数据库操作的时候都会获取每一次执行的`sqlSession`。

2.4 SqlSessionFactory具体实现类

```
1 public class DefaultSqlSessionFactory implements SqlSessionFactory {
2
3     private final Configuration configuration;
4
5     public DefaultSqlSessionFactory(Configuration configuration) {
6         this.configuration = configuration;
7     }
8
9     @Override
10    public SqlSession openSession() {
11        return new DefaultSqlSession(configuration.connection,
12            configuration.mapperElement);
13    }
14 }
```

- `DefaultSqlSessionFactory`，是使用mybatis最常用的类，这里我们简单的实现了一个版本。
- 虽然是简单的版本，但是包括了最基本的核心思路。当开启`SqlSession`时会进行返回一个`DefaultSqlSession`
- 这个构造函数中向下传递了`Configuration`配置文件，在这个配置文件中包括：`Connection connection`、`Map<String, String> dataSource`、`Map<String, XNode> mapperElement`。如果有你阅读过Mybatis源码，对这个就不会陌生。

2.5 SqlSessionFactoryBuilder实现

```
1 public class SqlSessionFactoryBuilder {
2
3     public DefaultSqlSessionFactory build(Reader reader) {
4         SAXReader saxReader = new SAXReader();
5         try {
6             saxReader.setEntityResolver(new XMLMapperEntityResolver());
7             Document document = saxReader.read(new InputSource(reader));
8             Configuration configuration =
9                 parseConfiguration(document.getRootElement());
10            return new DefaultSqlSessionFactory(configuration);
11        } catch (DocumentException e) {
12            e.printStackTrace();
13        }
14        return null;
15    }
16
17    private Configuration parseConfiguration(Element root) {
18        Configuration configuration = new Configuration();
19
20        configuration.setDataSource(dataSource(root.selectNodes("//dataSource")));
21
22        configuration.setConnection(connection(configuration.dataSource));
23
24        configuration.setMapperElement(mapperElement(root.selectNodes("mappers")));
25
26        return configuration;
27    }
28
29    // 获取数据源配置信息
30    private Map<String, String> dataSource(List<Element> list) {
31        Map<String, String> dataSource = new HashMap<>(4);
32        Element element = list.get(0);
33        List content = element.content();
34        for (Object o : content) {
35            Element e = (Element) o;
36            String name = e.attributeValue("name");
37            String value = e.attributeValue("value");
38            dataSource.put(name, value);
39        }
40        return dataSource;
41    }
42
43    private Connection connection(Map<String, String> dataSource) {
44        try {
45            Class.forName(dataSource.get("driver"));
46            return DriverManager.getConnection(dataSource.get("url"),
47                dataSource.get("username"), dataSource.get("password"));
48        } catch (ClassNotFoundException | SQLException e) {
49        }
50    }
51
52    private void printStackTrace() {
53        System.out.println("-----");
54        StackTraceElement[] stackTraceElements = Thread.currentThread().getStackTrace();
55        for (StackTraceElement stackTraceElement : stackTraceElements) {
56            System.out.println(stackTraceElement);
57        }
58    }
59}
```



```

89         xNode.setResultType(resultType);
90         xNode.setSql(sql);
91         xNode.setParameter(parameter);
92
93         map.put(namespace + "." + id, xNode);
94     }
95 } catch (Exception ex) {
96     ex.printStackTrace();
97 }
98
99 }
100 return map;
101 }
102
103 }

```

- 在这个类中包括的核心方法有； `build(构建实例化元素)`、`parseConfiguration(解析配置)`、`dataSource(获取数据库配置)`、`connection(Map<String, String> dataSource)`（链接数据库）、`mapperElement (解析sql语句)`
- 接下来我们分别介绍这样的几个核心方法。

build(构建实例化元素)

这个类主要用于创建解析xml文件的类，以及初始化SqlSession工厂

类 `DefaultSqlSessionFactory`。另外需要注意这段代码 `saxReader.setEntityResolver(new XMLMapperEntityResolver());`，是为了保证在不联网的时候一样可以解析xml，否则会需要从互联网获取dtd文件。

parseConfiguration(解析配置)

是对xml中的元素进行获取，这里主要获取了； `dataSource`、`mappers`，而这两个配置一个是我们数据库的链接信息，另外一个是数据库操作语句的解析。

connection(Map<String, String> dataSource) (链接数据库)

链接数据库的地方和我们常见的方式是一样的； `Class.forName(dataSource.get("driver"));`，但是这样包装以后外部是不需要知道具体的操作。同时当我们需要链接多套数据库的时候，也是可以在这里扩展。

mapperElement (解析sql语句)

这部分代码块内容相对来说比较长，但是核心的点就是为了解析xml中的sql语句配置。在我们平常的使用中基本都会配置一些sql语句，也有一些入参的占位符。在这里我们使用正则表达式的方式进行解析操作。

解析完成的sql语句就有了一个名称和sql的映射关系，当我们进行数据库操作的时候，这个组件就可以通过映射关系获取到对应sql语句进行操作。

3. 测试验证

在测试之前需要导入sql语句到数据库中；

- 库名: itstack-demo-design

- 表名: user、school

```
1 CREATE TABLE school ( id bigint NOT NULL AUTO_INCREMENT, name varchar(64),  
2   address varchar(256), createTime datetime, updateTime datetime, PRIMARY KEY  
3   (id) ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
4 insert into school (id, name, address, createTime, updateTime) values (1,  
5   '北京大学', '北京市海淀区颐和园路5号', '2019-10-18 13:35:57', '2019-10-18  
6   13:35:57');  
7 insert into school (id, name, address, createTime, updateTime) values (2,  
8   '南开大学', '中国天津市南开区卫津路94号', '2019-10-18 13:35:57', '2019-10-18  
9   13:35:57');  
10 insert into school (id, name, address, createTime, updateTime) values (3,  
11   '同济大学', '上海市彭武路1号同济大厦A楼7楼7区', '2019-10-18 13:35:57', '2019-10-  
12   18 13:35:57');  
13 CREATE TABLE user ( id bigint(11) NOT NULL AUTO_INCREMENT, name  
14   varchar(32), age int(4), address varchar(128), entryTime datetime, remark  
15   varchar(64), createTime datetime, updateTime datetime, status int(4)  
16   DEFAULT '0', dateTIme varchar(64), PRIMARY KEY (id), INDEX idx_name (name)  
17   ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  
18 insert into user (id, name, age, address, entryTime, remark, createTime,  
19   updateTime, status, dateTIme) values (1, '水水', 18, '吉林省榆树市黑林镇尹家村5  
20   组', '2019-12-22 00:00:00', '无', '2019-12-22 00:00:00', '2019-12-22  
21   00:00:00', 0, '20200309');  
22 insert into user (id, name, age, address, entryTime, remark, createTime,  
23   updateTime, status, dateTIme) values (2, '豆豆', 18, '辽宁省大连市清河湾司马道  
24   407路', '2019-12-22 00:00:00', '无', '2019-12-22 00:00:00', '2019-12-22  
25   00:00:00', 1, null);  
26 insert into user (id, name, age, address, entryTime, remark, createTime,  
27   updateTime, status, dateTIme) values (3, '花花', 19, '辽宁省大连市清河湾司马道  
28   407路', '2019-12-22 00:00:00', '无', '2019-12-22 00:00:00', '2019-12-22  
29   00:00:00', 0, '20200310');
```

3.1 创建数据库对象类

用户类

```
1 public class User {  
2  
3     private Long id;  
4     private String name;  
5     private Integer age;  
6     private Date createTime;  
7     private Date updateTime;  
8  
9     // ... get/set  
10 }
```

学校类

```
1 public class School {  
2  
3     private Long id;  
4     private String name;  
5     private String address;  
6     private Date createTime;  
7     private Date updateTime;  
8  
9     // ... get/set  
10 }
```

- 这两个类都非常简单，就是基本的数据库信息。

3.2 创建DAO包

用户Dao

```
1 public interface IUserDao {  
2  
3     User queryUserInfoById(Long id);  
4  
5 }
```

学校Dao

```
1 public interface ISchoolDao {  
2  
3     School querySchoolInfoById(Long treeId);  
4  
5 }
```

3.3 ORM配置文件

链接配置

```
1 <configuration>  
2     <environments default="development">  
3         <environment id="development">  
4             <transactionManager type="JDBC"/>  
5             <dataSource type="POOLED">  
6                 <property name="driver" value="com.mysql.jdbc.Driver"/>  
7                 <property name="url"  
8                     value="jdbc:mysql://127.0.0.1:3306/itstack_demo_design?useUnicode=true"/>  
9                 <property name="username" value="root"/>  
10                <property name="password" value="123456"/>  
11            </dataSource>  
12        </environment>
```

```
12     </environments>
13
14     <mappers>
15         <mapper resource="mapper/User_Mapper.xml" />
16         <mapper resource="mapper/School_Mapper.xml" />
17     </mappers>
18
19 </configuration>
```

- 这个配置与我们平常使用的mybatis基本是一样的，包括了数据库的连接池信息以及需要引入的mapper映射文件。

操作配置(用户)

```
1 <mapper namespace="org.itstack.demo.design.dao.IUserDao">
2
3     <select id="queryUserInfoById" parameterType="java.lang.Long"
4         resultType="org.itstack.demo.design.po.User">
5         SELECT id, name, age, createTime, updateTime
6         FROM user
7         where id = #{id}
8     </select>
9
10    <select id="queryUserList"
11        parameterType="org.itstack.demo.design.po.User"
12        resultType="org.itstack.demo.design.po.User">
13        SELECT id, name, age, createTime, updateTime
14        FROM user
15        where age = #{age}
16    </select>
17
18 </mapper>
```

操作配置(学校)

```
1 <mapper namespace="org.itstack.demo.design.dao.ISchoolDao">
2
3     <select id="querySchoolInfoById"
4         resultType="org.itstack.demo.design.po.School">
5         SELECT id, name, address, createTime, updateTime
6         FROM school
7         where id = #{id}
8     </select>
9 </mapper>
```

3.4 单个结果查询测试

```

1  @Test
2  public void test_queryUserInfoById() {
3      String resource = "mybatis-config-datasource.xml";
4      Reader reader;
5      try {
6          reader = Resources.getResourceAsReader(resource);
7          SqlSessionFactory sqlMapper = new
8          SqlSessionFactoryBuilder().build(reader);
9          SqlSession session = sqlMapper.openSession();
10         try {
11             User user =
12             session.selectOne("org.itstack.demo.design.dao.IUserDao.queryUserInfoById"
13             , 1L);
14             logger.info("测试结果: {}", JSON.toJSONString(user));
15         } finally {
16             session.close();
17             reader.close();
18         }
19     }

```

- 这里的使用方式和 Mybatis 是一样的，都包括了：资源加载和解析、`SqlSession` 工厂构建、开启 `SqlSession` 以及最后执行查询操作 `selectOne`

测试结果

```

1  16:56:51.831 [main] INFO  org.itstack.demo.design.demo.ApiTest - 测试结
果: {"age":18,"createTime":1576944000000,"id":1,"name":"水
2  水","updateTime":1576944000000}
3  Process finished with exit code 0

```

- 从结果上看已经满足了我们的查询需求。

3.5 集合结果查询测试

```

1  @Test
2  public void test_queryUserList() {
3      String resource = "mybatis-config-datasource.xml";
4      Reader reader;
5      try {
6          reader = Resources.getResourceAsReader(resource);
7          SqlSessionFactory sqlMapper = new
8          SqlSessionFactoryBuilder().build(reader);
9          SqlSession session = sqlMapper.openSession();
10         try {

```

```

10     User req = new User();
11     req.setAge(18);
12     List<User> userList =
13         session.selectList("org.itstack.demo.design.dao.IUserDao.queryUserList",
14             req);
15         logger.info("测试结果: {}", JSON.toJSONString(userList));
16     } finally {
17         session.close();
18         reader.close();
19     }
20 } catch (Exception e) {
21     e.printStackTrace();
}

```

- 这个测试内容与以上只是查询方法有所不同；`session.selectList`，是查询一个集合结果。

测试结果

```

1 16:58:13.963 [main] INFO org.itstack.demo.design.demo.ApiTest - 测试结
果: [{"age":18,"createTime":1576944000000,"id":1,"name":"水
2 水","updateTime":1576944000000},
3 {"age":18,"createTime":1576944000000,"id":2,"name":"豆
4 豆","updateTime":1576944000000}]
5
6 Process finished with exit code 0

```

- 测试验证集合的结果也是正常的，目前位置测试全部通过。

六、总结

- 以上通过中介者模式的设计思想我们手写了一个ORM框架，隐去了对数据库操作的复杂度，让外部的调用方可以非常简单的进行操作数据库。这也是我们平常使用的Mybatis的原型，在我们日常的开发使用中，只需要按照配置即可非常简单的操作数据库。
- 除了以上这种组件模式的开发外，还有服务接口的包装也可以使用中介者模式来实现。比如你们公司有很多的奖品接口需要在营销活动中对接，那么可以把这些奖品接口统一收到中台开发一个奖品中心，对外提供服务。这样就不需要每一个需要对接奖品的接口，都去找具体的提供者，而是找中台服务即可。
- 在上述的实现和测试使用中可以看到，这种模式的设计满足了；单一职责 和 开闭原则，也就符合了迪米特原则，即越少人知道越好。外部的人只需要按照需求进行调用，不需要知道具体的是如何实现的，复杂的一面已经有组件合作服务平台处理。

第5节：备忘录模式

实现不了是研发的借口？

实现不了，有时候是功能复杂度较高难以实现，有时候是工期较短实现不完。而编码的行为又是一个不太好量化的过程，同样一个功能每个人的实现方式不一样，遇到开发问题解决问题的速度也不一样。除此之外还很不好给产品解释具体为什么要这个工期时间，这就像盖楼的图纸最终要多少水泥砂浆一样。那么这时研发会尽可能的去通过一些经验，制定流程规范、设计、开发、评审等，确定一个可以完成的时间范围，又避免风险的时间点后。再被压缩，往往会出现一些矛盾点，能压缩要解释为什么之前要那么多时间，不能压缩又有各方不断施加的压力。因此有时候不一定是借口，是要考虑如何让整个团队健康的发展。

鼓励有时比压力要重要！

在学习的过程中，很多时候我们听到的都是，你要怎样，怎样，你瞧瞧谁谁谁，哪怕今天听不到这样的声音了，但因为曾经反复听到过而导致内心抗拒。虽然也知道自己要去学，但是很难坚持，学着学着就没有了方向，看到还有那么多不会的就更慌了，以至于最后心态崩了，更不愿意学。其实程序员的压力并不小，想成长几乎是需要一直的学习，就像似乎再也不敢说精通java了一样，知识量实在是随着学习的深入，越来越深，越来越广。所以需要，开心学习，快乐成长！

临阵的你好像一直很着急！

经常的听到；老师明天就要了你帮我弄弄吧、你给我写一下完事我就学这次着急、现在这不是没时间学吗快给我看看。其实看到的类似的还有很多，很纳闷你的着急怎么来的，不太可能，人在家中坐，祸从天上来。老师怎么就那个时间找你了，老板怎么就今天管你要了，还不是日积月累你没有学习，临时抱佛脚乱着急！即使后来真的有人帮你了，但最好不要放松，要尽快学会，躲得过初一还有初二呢！

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程一个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

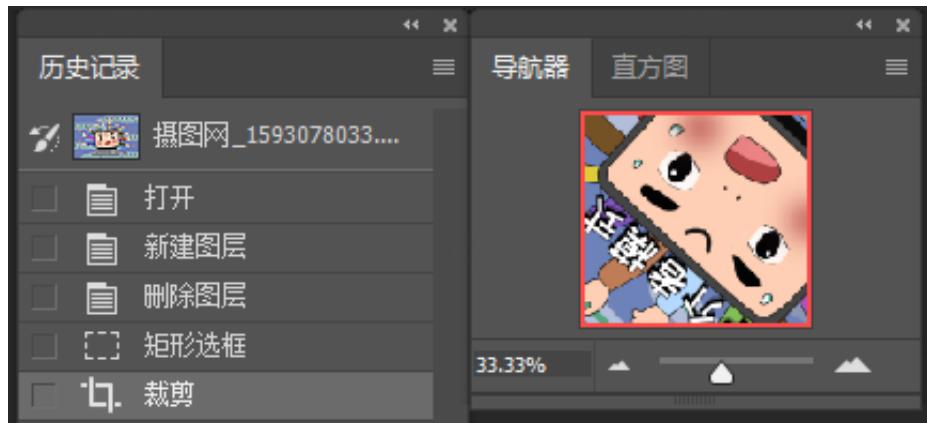
工程	描述
itstack-demo-design-17-00	开发配置文件备忘录

二、备忘录模式介绍



备忘录模式是以可以恢复或者说回滚，配置、版本、悔棋为核心功能的设计模式，而这种设计模式属于行为模式。在功能实现上是以不破坏原对象为基础增加备忘录操作类，记录原对象的行为从而实现备忘录模式。

这个设计在我们平常的生活或者开发中也是比较常见的，比如：后悔药、孟婆汤(一下回滚到0)，IDEA编辑和撤销、小霸王游戏机存档。当然还有我们非常常见的Photoshop，如下；



三、案例场景模拟



在本案例中我们模拟系统在发布上线的过程中记录线上配置文件用于紧急回滚

在大型互联网公司系统的发布上线一定是易用、安全、可处理紧急状况的，同时为了可以隔离线上和本地环境，一般会把配置文件抽取出放到线上，避免有人误操作导致本地的配置内容发布出去。同时线上的配置文件也会在每次变更的时候进行记录，包括：版本号、时间、MD5、内容信息和操作人。

在后续上线时如果发现紧急问题，系统就会需要回滚操作，如果执行回滚那么也可以设置配置文件是否回滚。因为每一个版本的系统可能会随着带着一些配置文件的信息，这个时候就可以很方便的让系统与配置文件一起回滚操作。

我们接下来就使用备忘录模式，模拟如何记录配置文件信息。实际的使用过程中还会将信息存放到库中进行保存，这里暂时只是使用内存记录。

四、备忘录模式记录配置文件版本信息

备忘录的设计模式实现方式，重点在于不更改原有类的基础上，增加备忘录类存放记录。可能平时虽然不一定非得按照这个设计模式的代码结构来实现自己的需求，但是对于功能上可能也完成过类似的功能，记录系统的信息。

除了现在的这个案例外，还可以是运营人员在后台erp创建活动对信息的记录，方便运营人员可以上下修改自己的版本，而不至于因为误操作而丢失信息。

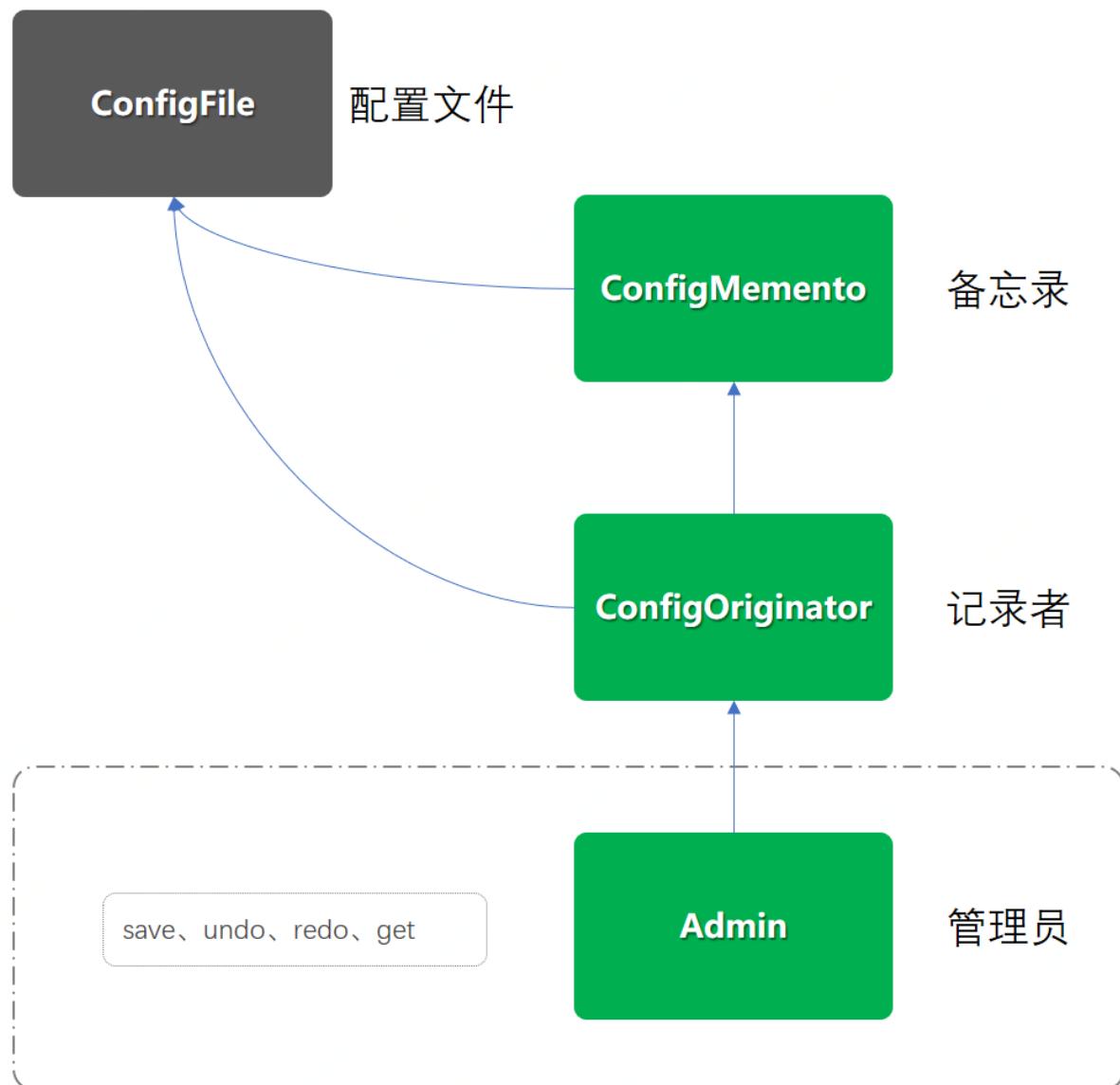
1. 工程结构

```

1  itstack-demo-design-17-00
2  └── src
3      ├── main
4      │   └── java
5      │       └── org.itstack.demo.design
6      │           ├── Admin.java
7      │           ├── ConfigFile.java
8      │           ├── ConfigMemento.java
9      │           └── ConfigOriginator.java
10     └── test
11         └── java
12             └── org.itstack.demo.design.test
13                 └── ApiTest.java

```

备忘录模式模型结构



- 以上是工程结构的一个类图，其实相对来说并不复杂，除了原有的配置类(`ConfigFile`)以外，只新增加了三个类。
- `ConfigMemento`：备忘录类，相当于是对原有配置类的扩展
- `ConfigOriginator`：记录者类，获取和返回备忘录类对象信息
- `Admin`：管理员类，用于操作记录备忘信息，比如你一些列的顺序执行了什么或者某个版本下的内容信息

2. 代码实现

2.1 配置信息类

```
1 public class ConfigFile {  
2  
3     private String versionNo; // 版本号  
4     private String content; // 内容  
5     private Date dateTime; // 时间  
6     private String operator; // 操作人  
7  
8     // ...get/set  
9 }
```

- 配置类可以是任何形式的，这里只是简单的描述了一个基本的配置内容信息。

2.2 备忘录类

```
1 public class ConfigMemento {  
2  
3     private ConfigFile configFile;  
4  
5     public ConfigMemento(ConfigFile configFile) {  
6         this.configFile = configFile;  
7     }  
8  
9     public ConfigFile getConfigFile() {  
10        return configFile;  
11    }  
12  
13    public void setConfigFile(ConfigFile configFile) {  
14        this.configFile = configFile;  
15    }  
16  
17 }
```

- 备忘录是对原有配置类的扩展，可以设置和获取配置信息。

2.3 记录者类

```
1 public class ConfigOriginator {  
2  
3     private ConfigFile configFile;  
4  
5     public ConfigFile getConfigFile() {  
6         return configFile;  
7     }  
8  
9     public void setConfigFile(ConfigFile configFile) {  
10        this.configFile = configFile;  
11    }  
12  
13     public ConfigMemento saveMemento(){
```

```

14         return new ConfigMemento(configFile);
15     }
16
17     public void getMemento(ConfigMemento memento){
18         this.configFile = memento.getConfigFile();
19     }
20
21 }

```

- 记录者类除了对 configFile 配置类增加了获取和设置方法外，还增加了保存 saveMemento()、
获取 getMemento(ConfigMemento memento)。
- saveMemento：保存备忘录的时候会创建一个备忘录信息，并返回回去，交给管理者处理。
- getMemento：获取的之后并不是直接返回，而是把备忘录的信息交给现在的配置文件
this.configFile，这部分需要注意。

2.4 管理员类

```

1  public class Admin {
2
3     private int cursorIdx = 0;
4     private List<ConfigMemento> mementoList = new ArrayList<ConfigMemento>()
5     ();
6     private Map<String, ConfigMemento> mementoMap = new
7     ConcurrentHashMap<String, ConfigMemento>();
8
9     public void append(ConfigMemento memento) {
10        mementoList.add(memento);
11        mementoMap.put(memento.getConfigFile().getVersionNo(), memento);
12        cursorIdx++;
13    }
14
15    public ConfigMemento undo() {
16        if (--cursorIdx <= 0) return mementoList.get(0);
17        return mementoList.get(cursorIdx);
18    }
19
20    public ConfigMemento redo() {
21        if (++cursorIdx > mementoList.size()) return
22        mementoList.get(mementoList.size() - 1);
23        return mementoList.get(cursorIdx);
24    }
25
26    public ConfigMemento get(String versionNo){
27        return mementoMap.get(versionNo);
28    }
29
30 }

```

- 在这个类中主要实现的核心功能就是记录配置文件信息，也就是备忘录的效果，之后提供可以回滚和获取的方法，拿到备忘录的具体内容。
- 同时这里设置了两个数据结构来存放备忘录，实际使用中可以按需设置。`List<ConfigMemento>`、`Map<String, ConfigMemento>`。
- 最后是提供的备忘录操作方法；存放(`append`)、回滚(`undo`)、返回(`redo`)、定向获取(`get`)，这样四个操作方法。

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test() {
3      Admin admin = new Admin();
4      ConfigOriginator configOriginator = new ConfigOriginator();
5      configOriginator.setConfigFile(new ConfigFile("1000001", "配置内容A=哈
6      哈", new Date(), "小傅哥"));
7      admin.append(configOriginator.saveMemento()); // 保存配置
8      configOriginator.setConfigFile(new ConfigFile("1000002", "配置内容A=嘻
9      嘻", new Date(), "小傅哥"));
10     admin.append(configOriginator.saveMemento()); // 保存配置
11     configOriginator.setConfigFile(new ConfigFile("1000003", "配置内容A=么
12     么", new Date(), "小傅哥"));
13     admin.append(configOriginator.saveMemento()); // 保存配置
14     configOriginator.setConfigFile(new ConfigFile("1000004", "配置内容A=嘿
15     嘿", new Date(), "小傅哥"));
16     admin.append(configOriginator.saveMemento()); // 保存配置
17
18     // 历史配置(回滚)
19     configOriginator.getMemento(admin.undo());
20     logger.info("历史配置(回滚)undo: {}",
21     JSON.toJSONString(configOriginator.getConfigFile()));
22
23     // 历史配置(回滚)
24     configOriginator.getMemento(admin.undo());
25     logger.info("历史配置(回滚)undo: {}",
26     JSON.toJSONString(configOriginator.getConfigFile()));
27
28     // 历史配置(前进)
29     configOriginator.getMemento(admin.redo());
30     logger.info("历史配置(前进)redo: {}",
31     JSON.toJSONString(configOriginator.getConfigFile()));
32
33     // 历史配置(获取)
34     configOriginator.getMemento(admin.get("1000002"));
35     logger.info("历史配置(获取)get: {}",
36     JSON.toJSONString(configOriginator.getConfigFile()));
37 }

```

- 这个设计模式的学习有一部分重点是体现在了单元测试类上，这里包括了四次的信息存储和备忘录历史配置操作。
- 通过上面添加了四次配置后，下面分别进行操作是；回滚1次、再回滚1次，之后向前进1次，最后是获取指定的版本配置。具体的效果可以参考测试结果。

3.2 测试结果

```

1 23:12:09.512 [main] INFO org.itstack.demo.design.test.ApiTest - 历史配置(回
滚)undo: {"content": "配置内容A=嘿嘿", "dateTime": 159209829432, "operator": "小傅
哥", "versionNo": "1000004"}
2 23:12:09.514 [main] INFO org.itstack.demo.design.test.ApiTest - 历史配置(回
滚)undo: {"content": "配置内容A=么么", "dateTime": 159209829432, "operator": "小傅
哥", "versionNo": "1000003"}
3 23:12:09.514 [main] INFO org.itstack.demo.design.test.ApiTest - 历史配置(前
进)redo: {"content": "配置内容A=嘿嘿", "dateTime": 159209829432, "operator": "小傅
哥", "versionNo": "1000004"}
4 23:12:09.514 [main] INFO org.itstack.demo.design.test.ApiTest - 历史配置(获
取)get: {"content": "配置内容A=嘻嘻", "dateTime": 159320989432, "operator": "小傅
哥", "versionNo": "1000002"}
5
6 Process finished with exit code 0

```

- 从测试效果上可以看到，历史配置按照我们的指令进行了回滚和前进，以及最终通过指定的版本进行获取，符合预期结果。

五、总结

- 此种设计模式的方式可以满足在不破坏原有属性类的基础上，扩充了备忘录的功能。虽然和我们平时使用的思路是一样的，但在具体实现上还可以细细品味，这样的方式在一些源码中也有所体现。
- 在以上的实现中我们是将配置模拟存放到内存中，如果关机了会导致配置信息丢失，因为在一些真实的场景里还是需要存放到数据库中。那么此种存放到内存中进行回复的场景也不是没有，比如；Photoshop、运营人员操作ERP配置活动，那么也就是即时性的一般不需要存放到库中进行恢复。另外如果是使用内存方式存放备忘录，需要考虑存储问题，避免造成内存大量消耗。
- 设计模式的学习都是为了更好的写出可扩展、可管理、易维护的代码，而这个学习的过程需要自己不断的尝试实际操作，理论的知识与实际结合还有很长一段距离。切记多多上手！

第6节：观察者模式

知道的越多不知道的就越多

编程开发这条路上的知识是无穷无尽的，就像以前你敢说精通Java，到后来学到越来越多只想写了解Java，过了几年现在可能想说懂一点点Java。当视野和格局的扩大，会让我们越来越发现原来的看法是多么浅显，这就像站在地球看地球和站在宇宙看地球一样。但正因为胸怀和眼界的提升让我们有了更多的认识，也逐渐学会了更多的技能。虽然不知道的越来越多，但也因此给自己填充了更多的技术栈，让自己越来越强大。

拒绝学习的惰性很可怕

现在与以前不一样，资料多、途径广，在这中间夹杂的广告也非常多。这就让很多初学者很难找到自己要的知识，最后看到有人推荐相关学习资料立刻屏蔽、删除，但同时技术优秀的资料也不能让需要的人看见了。久而久之把更多的时间精力都放在游戏、娱乐、影音上，适当的放松是可以的，但往往沉迷以后就很难出来，因此需要做好一些可以让自己成长的计划，稍有克制。

平衡好软件设计和实现成本的度。

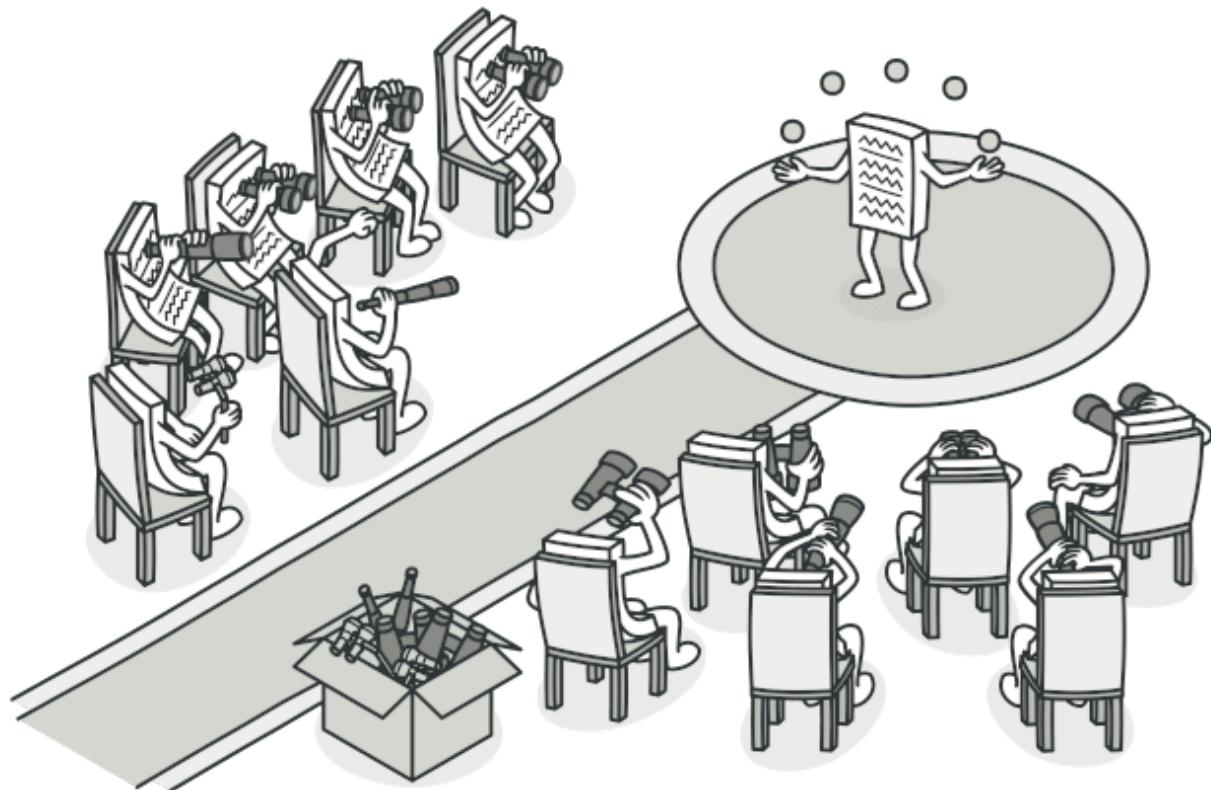
有时候一个软件的架构设计需要符合当前条件下的各项因素，往往不能因为心中想当然的有某个蓝图，就开始执行。也许虽然你的设计是非常优秀的，但是放在当前环境下很难满足业务的时间要求，当一个业务的基本诉求不能满足后，就很难拉动市场。没有产品的DAU支撑，最后整个研发的项目也会因此停滞。但研发又不能一团乱麻的写代码，因此需要找好一个适合的度，比如可以搭建良好的地基，实现上可扩展。但在具体的功能上可以先简化实现，随着活下来了再继续完善迭代。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-18-00	场景模拟工程；模拟一个小客车摇号接口
itstack-demo-design-18-01	使用一坨代码实现业务需求
itstack-demo-design-18-02	通过设计模式优化改造代码，产生对比性从而学习

二、观察者模式介绍



简单来讲观察者模式，就是当一个行为发生时传递信息给另外一个用户接收做出相应的处理，两者之间没有直接的耦合关联。例如；狙击手、李云龙。



除了生活中的场景外，在我们编程开发中也会常用到一些观察者的模式或者组件，例如我们经常使用的MQ服务，虽然MQ服务是有一个通知中心并不是每一个类服务进行通知，但整体上也可以算作是观察者模式的思路设计。再比如可能有做过的一些类似事件监听总线，让主线服务与其他辅线业务服务分离，为了使系统降低耦合和增强扩展性，也会使用观察者模式进行处理。

三、案例场景模拟



在本案例中我们模拟每次小客车指标摇号事件通知场景(真实的不会由官网给你发消息)

可能大部分人看到这个案例一定会想到自己每次摇号都不中的场景，收到一个遗憾的短信通知。当然目前的摇号系统并不会给你发短信，而是由百度或者一些其他插件发的短信。那么假如这个类似的摇号功能如果由你来开发，并且需要对外部的用户做一些事件通知以及需要在主流程外再添加一些额外的辅助流程时该如何处理呢？

基本很多人对于这样的通知事件类的实现往往比较粗犷，直接在类里面就添加了。1是考虑🤔这可能不会怎么扩展，2是压根就没考虑😊过。但如果你有仔细思考过你的核心类功能会发现，这里面有一些核心主链路，还有一部分是辅助功能。比如完成了某个行为后需要触发MQ给外部，以及做一些消息PUSH给用户等，这些都不算做是核心流程链路，是可以通过事件通知的方式进行处理。

那么接下来我们就使用这样的设计模式来优化重构此场景下的代码。

1. 场景模拟工程

```

1 | itstack-demo-design-18-00
2 | └─ src
3 |   └─ main
4 |     └─ java
5 |       └─ org.itstack.demo.design
6 |         └─ MinibusTargetService.java

```

- 这里提供的是一个模拟小客车摇号的服务接口。

2. 场景简述

2.1 摆号服务接口

```

1 public class MinibusTargetService {
2
3     /**
4      * 模拟摇号，但不是摇号算法
5      *
6      * @param uId 用户编号
7      * @return 结果
8      */
9     public String lottery(String uId) {
10         return Math.abs(uId.hashCode()) % 2 == 0 ? "恭喜你，编
11             码".concat(uId).concat("在本次摇号中签") : "很遗憾，编
12             码".concat(uId).concat("在本次摇号未中签或摇号资格已过期");
13     }

```

- 非常简单的一个模拟摇号接口，与真实公平的摇号是有差别的。

四、用一坨坨代码实现

这里我们先使用最粗暴的方式来实现功能

按照需求需要在原有的摇号接口中添加MQ消息发送以及短消息通知功能，如果是最直接的方式那么可以直接在方法中补充功能即可。

1. 工程结构

```

1 itstack-demo-design-18-01
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         ├─ LotteryResult.java
7         ├─ LotteryService.java
8         └─ LotteryServiceImpl.java

```

- 这段代码接口中包括了三部分内容；返回对象(`LotteryResult`)、定义接口(`LotteryService`)、具体实现(`LotteryServiceImpl`)。

2. 代码实现

```

1 public class LotteryServiceImpl implements LotteryService {
2
3     private Logger logger =
4         LoggerFactory.getLogger(LotteryServiceImpl.class);
5
6     private MinibusTargetService minibusTargetService = new
7         MinibusTargetService();

```

```

6
7     public LotteryResult doDraw(String uId) {
8         // 摆号
9         String lottery = minibusTargetService.lottery(uId);
10        // 发短信
11        logger.info("给用户 {} 发送短信通知(短信): {}", uId, lottery);
12        // 发MQ消息
13        logger.info("记录用户 {} 摆号结果(MQ): {}", uId, lottery);
14        // 结果
15        return new LotteryResult(uId, lottery, new Date());
16    }
17
18 }

```

- 从以上的方法实现中可以看到，整体过程包括三部分：摇号、发短信、发MQ消息，而这部分都是顺序调用的。
- 除了摇号接口调用外，后面的两部分都是非核心主链路功能，而且会随着后续的业务需求发展而不断的调整和扩充，在这样的开发方式下就非常不利于维护。

3. 测试验证

3.1 编写测试类

```

1 @Test
2 public void test() {
3     LotteryService lotteryService = new LotteryServiceImpl();
4     LotteryResult result = lotteryService.doDraw("2765789109876");
5     logger.info("测试结果: {}", JSON.toJSONString(result));
6 }

```

- 测试过程中提供对摇号服务接口的调用。

3.2 测试结果

```

1 22:02:24.520 [main] INFO o.i.demo.design.LotteryServiceImpl - 给用户
2 2765789109876 发送短信通知(短信): 很遗憾，编码2765789109876在本次摇号未中签或摇号资格
3 已过期
2 22:02:24.523 [main] INFO o.i.demo.design.LotteryServiceImpl - 记录用户
2 2765789109876 摆号结果(MQ): 很遗憾，编码2765789109876在本次摇号未中签或摇号资格已过期
3 22:02:24.606 [main] INFO org.itstack.demo.design.ApiTest - 测试结
3 果: {"dateTime":1598764144524,"msg":"很遗憾，编码2765789109876在本次摇号未中签或
3 摆号资格已过期","uId":"2765789109876"}
4
5 Process finished with exit code 0

```

- 从测试结果上是符合预期的，也是平常开发代码的方式，还是非常简单的。

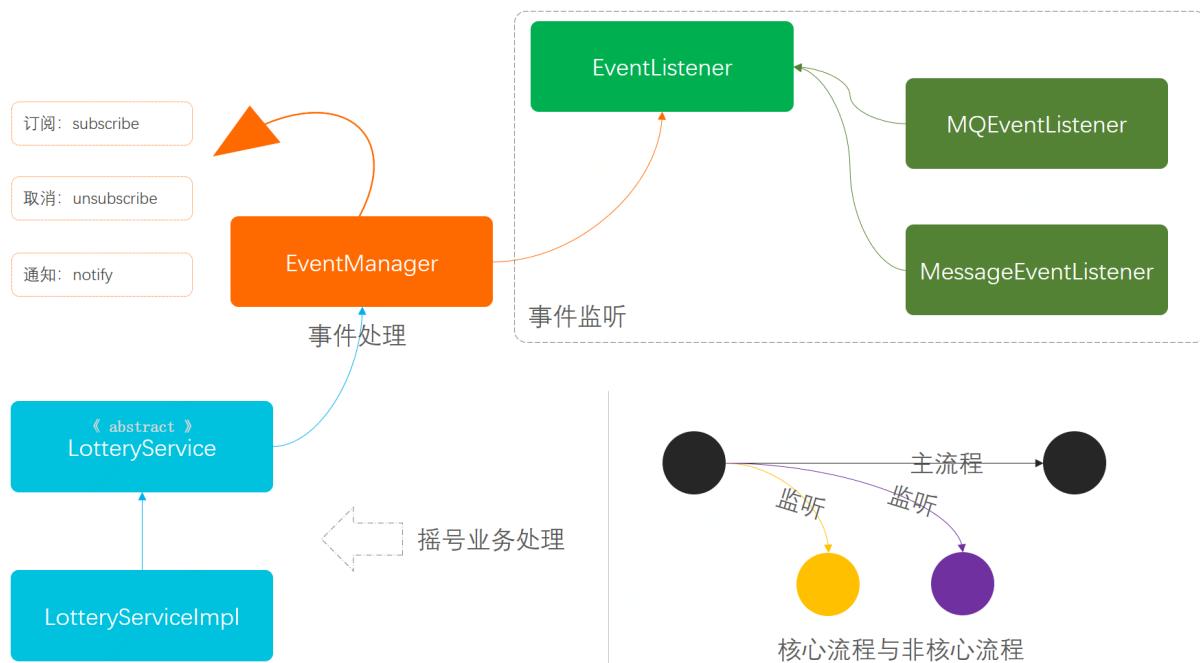
五、观察者模式重构代码

接下来使用观察者模式来进行代码优化，也算是一次很小的重构。

1. 工程结构

```
1 itstack-demo-design-18-02
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── event
7                 │   └── listener
8                 │       ├── EventListener.java
9                 │       ├── MessageEventListener.java
10                │       └── MQEventListener.java
11                └── EventManager.java
12
13
14 └── LotteryResult.java
    └── LotteryService.java
    └── LotteryServiceImpl.java
```

观察者模式模型结构



- 从上图可以分为三大块看；事件监听、事件处理、具体的业务流程，另外在业务流程中 `LotteryService` 定义的是抽象类，因为这样可以通过抽象类将事件功能屏蔽，外部业务流程开发者不需要知道具体的通知操作。
- 右下角圆圈图表示的是核心流程与非核心流程的结构，一般在开发中会把主线流程开发完成后，再使用通知的方式处理辅助流程。他们可以是异步的，在MQ以及定时任务的处理下，保证最终一致性。

2. 代码实现

2.1 事件监听接口定义

```
1 public interface EventListener {  
2  
3     void doEvent(LotteryResult result);  
4  
5 }
```

- 接口中定义了基本的事件类，这里如果方法的入参信息类型是变化的可以使用泛型 <T>

2.2 两个监听事件的实现

短消息事件

```
1 public class MessageEventListener implements EventListener {  
2  
3     private Logger logger =  
4         LoggerFactory.getLogger(MessageEventListener.class);  
5  
6     @Override  
7     public void doEvent(LotteryResult result) {  
8         logger.info("给用户 {} 发送短信通知(短信): {}", result.getuid(),  
9                     result.getMsg());  
10    }  
11 }
```

MQ发送事件

```
1 public class MQEventListener implements EventListener {  
2  
3     private Logger logger =  
4         LoggerFactory.getLogger(MQEventListener.class);  
5  
6     @Override  
7     public void doEvent(LotteryResult result) {  
8         logger.info("记录用户 {} 摆号结果(MQ): {}", result.getuid(),  
9                     result.getMsg());  
10    }  
11 }
```

- 以上是两个事件的具体实现，相对来说都比较简单。如果是实际的业务开发那么会需要调用外部接口以及控制异常的处理。
- 同时我们上面提到事件接口添加泛型，如果有需要那么在事件的实现中就可以按照不同的类型进行包装事件内容。

2.3 事件处理类

```
1 public class EventManager {
```

```
2
3     Map<Enum<EventType>, List<EventListener>> listeners = new HashMap<>();
4
5     public EventManager(Enum<EventType>... operations) {
6         for (Enum<EventType> operation : operations) {
7             this.listeners.put(operation, new ArrayList<>());
8         }
9     }
10
11    public enum EventType {
12        MQ, Message
13    }
14
15    /**
16     * 订阅
17     * @param eventType 事件类型
18     * @param listener 监听
19     */
20    public void subscribe(Enum<EventType> eventType, Listener
listener) {
21        List<EventListener> users = listeners.get(eventType);
22        users.add(listener);
23    }
24
25    /**
26     * 取消订阅
27     * @param eventType 事件类型
28     * @param listener 监听
29     */
30    public void unsubscribe(Enum<EventType> eventType, Listener
listener) {
31        List<EventListener> users = listeners.get(eventType);
32        users.remove(listener);
33    }
34
35    /**
36     * 通知
37     * @param eventType 事件类型
38     * @param result 结果
39     */
40    public void notify(EventType eventType, LotteryResult result) {
41        List<EventListener> users = listeners.get(eventType);
42        for (EventListener listener : users) {
43            listener.doEvent(result);
44        }
45    }
46
47 }
```

- 整个处理的实现上提供了三个主要方法：订阅(`subscribe`)、取消订阅(`unsubscribe`)、通知(`notify`)。这三个方法分别用于对监听时间的添加和使用。
- 另外因为事件有不同的类型，这里使用了枚举的方式进行处理，也方便让外部在规定下使用事件，而不至于乱传信息(`EventManager.EventType.MQ`、`EventManager.EventType.Message`)。

2.4 业务抽象类接口

```

1  public abstract class LotteryService {
2
3      private EventManager eventManager;
4
5      public LotteryService() {
6          eventManager = new EventManager(EventManager.EventType.MQ,
7              EventManager.EventType.Message);
8          eventManager.subscribe(EventManager.EventType.MQ, new
9              MQEventListener());
10         eventManager.subscribe(EventManager.EventType.Message, new
11             MessageEventListener());
12     }
13
14     public LotteryResult draw(String uId) {
15         LotteryResult lotteryResult = doDraw(uId);
16         // 需要什么通知就给调用什么方法
17         eventManager.notify(EventManager.EventType.MQ, lotteryResult);
18         eventManager.notify(EventManager.EventType.Message,
19             lotteryResult);
20         return lotteryResult;
21     }
22
23     protected abstract LotteryResult doDraw(String uId);
24
25 }
```

- 这种使用抽象类的方式定义实现方法，可以在方法中扩展需要的额外调用。并提供抽象类`abstract LotteryResult doDraw(String uId)`，让类的继承者实现。
- 同时方法的定义使用的是`protected`，也就是保证将来外部的调用方不会调用到此方法，只有调用到`draw(String uId)`，才能让我们完成事件通知。
- 此种方式的实现就是在抽象类中写好一个基本的方法，在方法中完成新增逻辑的同时，再增加抽象类的使用。而这个抽象类的定义会有继承者实现。
- 另外在构造函数中提供了对事件的定义：`eventManager.subscribe(EventManager.EventType.MQ, new MQEventListener())`。
- 在使用的时候也是使用枚举的方式进行通知使用，传了什么类型`EventManager.EventType.MQ`，就会执行什么事件通知，按需添加。

2.5 业务接口实现类

```
1 public class LotteryServiceImpl extends LotteryService {  
2  
3     private MinibusTargetService minibusTargetService = new  
MinibusTargetService();  
4  
5     @Override  
6     protected LotteryResult doDraw(String uId) {  
7         // 摆号  
8         String lottery = minibusTargetService.lottery(uId);  
9         // 结果  
10        return new LotteryResult(uId, lottery, new Date());  
11    }  
12  
13 }
```

- 现在再看业务流程的实现中可以看到已经非常简单了，没有额外的辅助流程，只有核心流程的处理。

3. 测试验证

3.1 编写测试类

```
1 @Test  
2 public void test() {  
3     LotteryService lotteryService = new LotteryServiceImpl();  
4     LotteryResult result = lotteryService.draw("2765789109876");  
5     logger.info("测试结果: {}", JSON.toJSONString(result));  
6 }
```

- 从调用上来看几乎没有区别，但是这样的实现方式就可以非常方便的维护代码以及扩展新的需求。

3.2 测试结果

```
1 23:56:07.597 [main] INFO o.i.d.d.e.listener.MQEventListener - 记录用户  
2765789109876 摆号结果(MQ): 很遗憾, 编码2765789109876在本次摇号未中签或摇号资格已过期  
2 23:56:07.600 [main] INFO o.i.d.d.e.l.MessageEventListener - 给用户  
2765789109876 发送短信通知(短信): 很遗憾, 编码2765789109876在本次摇号未中签或摇号资格已过期  
3 23:56:07.698 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: {"dateT  
4 ume":1599737367591,"msg":"很遗憾, 编码2765789109876在本次摇号未中签或  
5 摆号资格已过期","uId":"2765789109876"}  
Process finished with exit code 0
```

- 从测试结果上看满足😊我们的预期，虽然结果是一样的，但只有我们知道了设计模式的魅力所在。

六、总结

- 从我们最基本的过程式开发以及后来使用观察者模式面向对象开发，可以看到设计模式改造后，拆分出了核心流程与辅助流程的代码。一般代码中的核心流程不会经常变化。但辅助流程会随着业务的各种变化而变化，包括：`营销`、`裂变`、`促活`等等，因此使用设计模式架设代码就显得非常有必要。
- 此种设计模式从结构上是满足开闭原则的，当你需要新增其他的监听事件或者修改监听逻辑，是不需要改动事件处理类的。但是可能你不能控制调用顺序以及需要做一些事件结果的返回继续操作，所以使用的过程时需要考虑场景的合理性。
- 任何一种设计模式有时候都不是单独使用的，需要结合其他模式共同建设。另外设计模式的使用是为了让代码更加易于扩展和维护，不能因为添加设计模式而把结构处理更加复杂以及难以维护。这样的合理使用的经验需要大量的实际操作练习而来。

第 7 节：状态模式

`写好代码三个关键点`

如果把写代码想象成家里的软装，你肯定会想到家里需要有一个非常不错格局最好是南北通透的，买回来的家具最好是品牌保证质量的，之后呢是大小合适，不能摆放完了看着别扭。那么把这一过程抽象成写代码就是需要三个核心的关键点：`架构`(房间的格局)、`命名`(品牌和质量)、`注释`(尺寸大小说明书)，只有这三个点都做好才能完成出一套赏心悦目的家。

`平原走码`  `易放难收`

上学期间你写了多少代码？上班一年你能写多少代码？回家自己学习写了多少代码？个人素养的技术栈地基都是一块一块砖码出来的，写的越广越深，根基就越牢固。当根基牢固了以后在再上层建设就变得迎刃而解了，也更容易建设了。往往最难的就是一层一层阶段的突破，突破就像破壳一样，也像夯实地基，短时间看不到成绩，也看不出高度。但以后谁能走的稳，就靠着默默的沉淀。

技术传承的重要性

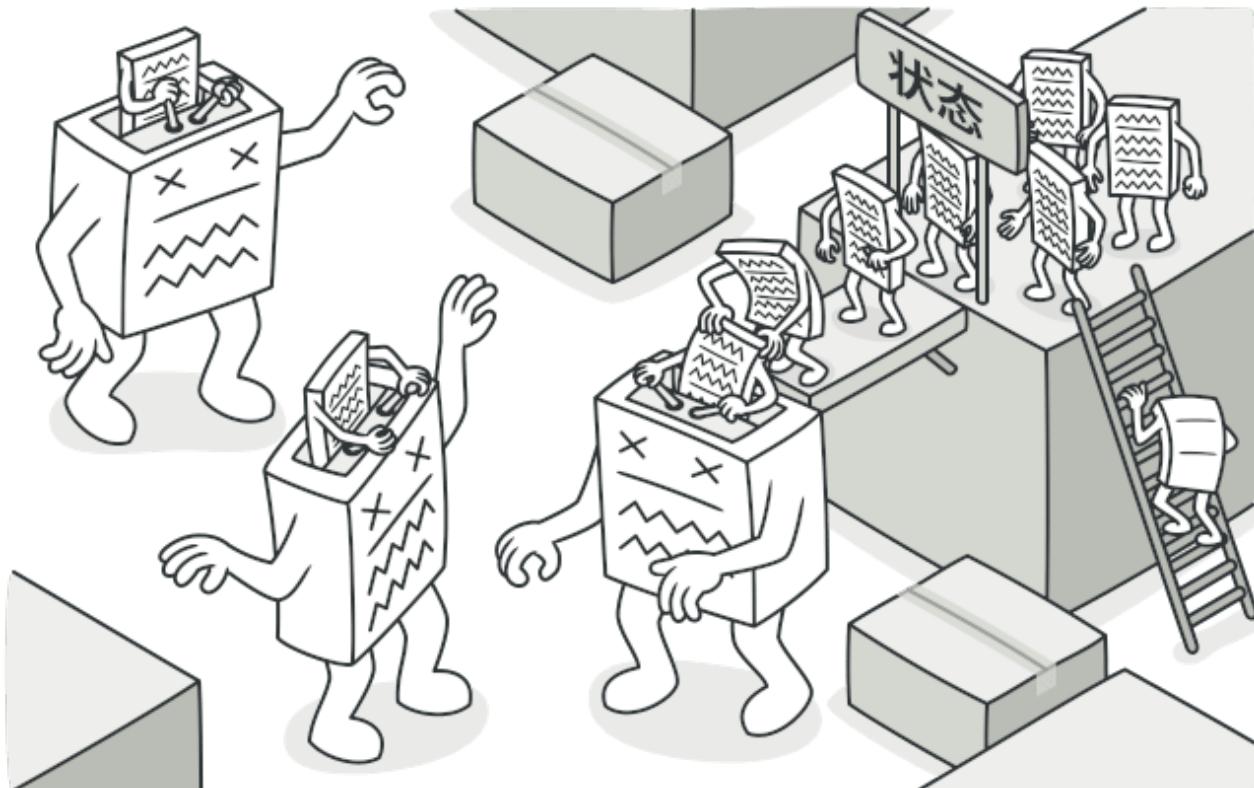
可能是现在时间节奏太快，一个需求下来恨不得当天就上线(这个需求很简单，怎么实现我不管，明天上线！)，导致团队的人都很慌、很急、很累、很崩溃，最终反反复复的人员更替，项目在这个过程中也交接了N次，文档不全、代码混乱、错综复杂，谁在后面接手也都只能修修补补，就像烂尾楼。这个没有传承、没有沉淀的项目，很难跟随业务的发展。最终！根基不牢，一地鸡毛。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-19-00	场景模拟工程；模拟营销活动操作服务(查询、审核)
itstack-demo-design-19-01	使用一坨代码实现业务需求
itstack-demo-design-19-02	通过设计模式优化改造代码，产生对比性从而学习

二、状态模式介绍

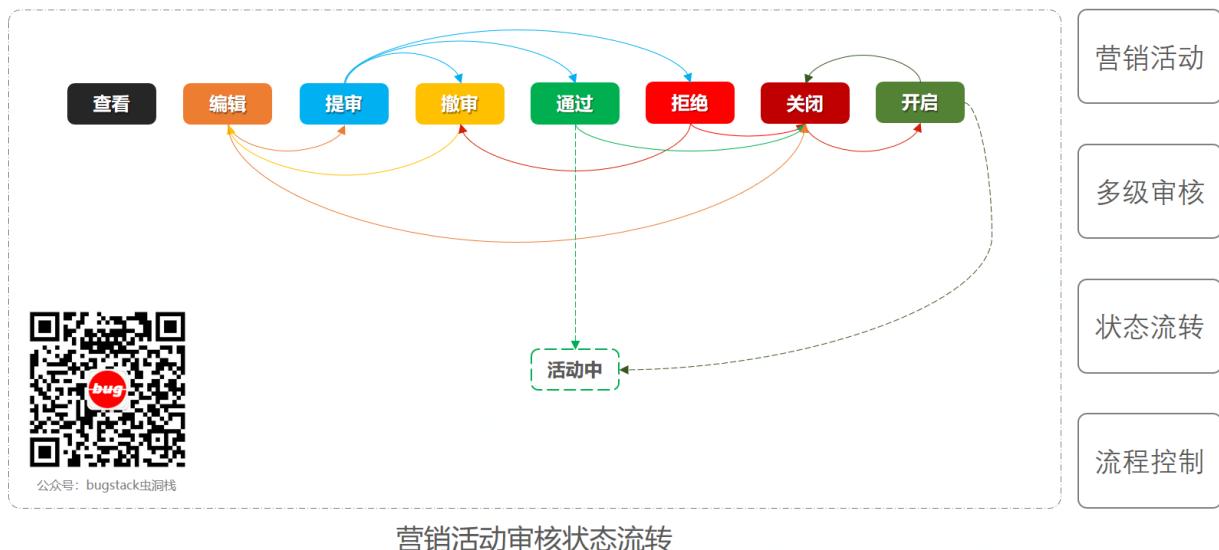


状态模式描述的是一个行为下的多种状态变更，比如我们最常见的一个网站的页面，在你登录与不登录下展示的内容是略有差异的(不登录不能展示个人信息)，而这种 登录 与 不登录 就是我们通过改变状态，而让整个行为发生了变化。



至少80后、90后的小伙伴基本都用过这种磁带放音机(可能没有这个好看),它的上面是一排按钮,当放入磁带后,通过上面的按钮就可以让放音机播放磁带上的内容(listen to 英语听力考试),而且有些按钮是互斥的,当在某个状态下才可以按另外的按钮(这在设计模式里也是一个关键的点)。

三、案例场景模拟



在本案例中我们模拟营销活动审核状态流转场景(一个活动的上线是多个层级审核上线的)

在上图中也可以看到我们的流程节点中包括了各个状态到下一个状态扭转的关联条件,比如:审核通过才能到活动中,而不能从编辑中直接到活动中,而这些状态的转变就是我们要完成的场景处理。

大部分程序员基本都开发过类似的业务场景，需要对活动或者一些配置需要审核后才能对外发布，而这个审核的过程往往会随着系统的重要程度而设立多级控制，来保证一个活动可以安全上线，避免造成资源。

当然有时候会用到一些审批流的过程配置，也是非常方便开发类似的流程的，也可以在配置中设定某个节点的审批人员。但这不是我们主要体现的点，在本案例中我们主要是模拟学习对一个活动的多个状态节点的审核控制。

1. 场景模拟工程

```
1 | itstack-demo-design-19-00
2 | └─ src
3 |   └─ main
4 |     └─ java
5 |       └─ org.itstack.demo.design
6 |         ├─ ActivityInfo.java
7 |         ├─ Status.java
8 |         └─ ActivityService.java
```

- 在这个模拟工程里我们提供了三个类，包括：状态枚举(`Status`)、活动对象(`ActivityInfo`)、活动服务(`ActivityService`)，三个服务类。
- 接下来我们就分别介绍三个类包括的内容。

2. 代码实现

2.1 基本活动信息

```
1 | public class ActivityInfo {
2 |
3 |     private String activityId;      // 活动ID
4 |     private String activityName;    // 活动名称
5 |     private Enum<Status> status;    // 活动状态
6 |     private Date beginTime;        // 开始时间
7 |     private Date endTime;          // 结束时间
8 |
9 |     // ...get/set
10| }
```

- 一些基本的活动信息；活动ID、活动名称、活动状态、开始时间、结束时间。

2.2 活动枚举状态

```
1 | public enum Status {
2 |
3 |     // 1创建编辑、2待审核、3审核通过(任务扫描成活动中)、4审核拒绝(可以撤审到编辑状态)、
4 |     // 5活动中、6活动关闭、7活动开启(任务扫描成活动中)
5 |     Editing, Check, Pass, Refuse, Doing, Close, Open
6 | }
```

- 活动的枚举；1创建编辑、2待审核、3审核通过(任务扫描成活动中)、4审核拒绝(可以撤审到编辑状态)、5活动中、6活动关闭、7活动开启(任务扫描成活动中)

2.3 活动服务接口

```

1  public class ActivityService {
2
3      private static Map<String, Enum<Status>> statusMap = new
4          ConcurrentHashMap<String, Enum<Status>>();
5
6      public static void init(String activityId, Enum<Status> status) {
7          // 模拟查询活动信息
8          ActivityInfo activityInfo = new ActivityInfo();
9          activityInfo.setActivityId(activityId);
10         activityInfo.setActivityName("早起学习打卡领奖活动");
11         activityInfo.setStatus(status);
12         activityInfo.setBeginTime(new Date());
13         activityInfo.setEndTime(new Date());
14         statusMap.put(activityId, status);
15     }
16
17     /**
18      * 查询活动信息
19      *
20      * @param activityId 活动ID
21      * @return 查询结果
22      */
23     public static ActivityInfo queryActivityInfo(String activityId) {
24         // 模拟查询活动信息
25         ActivityInfo activityInfo = new ActivityInfo();
26         activityInfo.setActivityId(activityId);
27         activityInfo.setActivityName("早起学习打卡领奖活动");
28         activityInfo.setStatus(statusMap.get(activityId));
29         activityInfo.setBeginTime(new Date());
30         activityInfo.setEndTime(new Date());
31         return activityInfo;
32     }
33
34     /**
35      * 查询活动状态
36      *
37      * @param activityId 活动ID
38      * @return 查询结果
39      */
40     public static Enum<Status> queryActivityStatus(String activityId) {
41         return statusMap.get(activityId);
42     }
43     /**

```

```

44     * 执行状态变更
45     *
46     * @param activityId 活动ID
47     * @param beforeStatus 变更前状态
48     * @param afterStatus 变更后状态 b
49     */
50     public static synchronized void execStatus(String activityId,
51         Enum<Status> beforeStatus, Enum<Status> afterStatus) {
52         if (!beforeStatus.equals(statusMap.get(activityId))) return;
53         statusMap.put(activityId, afterStatus);
54     }
55 }
```

- 在这个静态类中提供了活动的查询和状态变更接口；`queryActivityInfo`、`queryActivityStatus`、`execStatus`。
- 同时使用Map的结构来记录活动ID和状态变化信息，另外还有init方法来初始化活动数据。实际的开发中这类信息基本都是从数据库或者Redis中获取。

四、用一坨坨代码实现

这里我们先使用最粗暴的方式来实现功能

对于这样各种状态的变更，最让我们直接想到的就是使用`if`和`else`进行判断处理。每一个状态可以流转到下一个什么状态，都可以使用嵌套的`if`实现。

1. 工程结构

```

1 itstack-demo-design-19-01
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ ActivityExecStatusController.java
7         └─ Result.java
```

- 整个实现的工程结构比较简单，只包括了两个类：`ActivityExecStatusController`、`Result`，一个是处理流程状态，另外一个是返回的对象。

2. 代码实现

```

1 public class ActivityExecStatusController {
2
3     /**
4      * 活动状态变更
5      * 1. 编辑中 -> 提审、关闭
6      * 2. 审核通过 -> 拒绝、关闭、活动中
```

```
7     * 3. 审核拒绝 -> 撤审、关闭
8     * 4. 活动中 -> 关闭
9     * 5. 活动关闭 -> 开启
10    * 6. 活动开启 -> 关闭
11    *
12    * @param activityId 活动ID
13    * @param beforeStatus 变更前状态
14    * @param afterStatus 变更后状态
15    * @return 返回结果
16    */
17    public Result execStatus(String activityId, Enum<Status> beforeStatus,
18    Enum<Status> afterStatus) {
19
20        // 1. 编辑中 -> 提审、关闭
21        if (Status.Editing.equals(beforeStatus)) {
22            if (Status.Check.equals(afterStatus) ||
23                Status.Close.equals(afterStatus)) {
24                ActivityService.execStatus(activityId, beforeStatus,
25                afterStatus);
26                return new Result("0000", "变更状态成功");
27            } else {
28                return new Result("0001", "变更状态拒绝");
29            }
30        }
31
32        // 2. 审核通过 -> 拒绝、关闭、活动中
33        if (Status.Pass.equals(beforeStatus)) {
34            if (Status.Refuse.equals(afterStatus) ||
35                Status.Doing.equals(afterStatus) || Status.Close.equals(afterStatus)) {
36                ActivityService.execStatus(activityId, beforeStatus,
37                afterStatus);
38                return new Result("0000", "变更状态成功");
39            } else {
40                return new Result("0001", "变更状态拒绝");
41            }
42        }
43
44        // 3. 审核拒绝 -> 撤审、关闭
45        if (Status.Refuse.equals(beforeStatus)) {
46            if (Status.Editing.equals(afterStatus) ||
47                Status.Close.equals(afterStatus)) {
48                ActivityService.execStatus(activityId, beforeStatus,
49                afterStatus);
50                return new Result("0000", "变更状态成功");
51            } else {
52                return new Result("0001", "变更状态拒绝");
53            }
54        }
55    }
```

```

49     // 4. 活动中 -> 关闭
50     if (Status.Doing.equals(beforeStatus)) {
51         if (Status.Close.equals(afterStatus)) {
52             ActivityService.execStatus(activityId, beforeStatus,
53                                         afterStatus);
54             return new Result("0000", "变更状态成功");
55         } else {
56             return new Result("0001", "变更状态拒绝");
57         }
58     }
59
60     // 5. 活动关闭 -> 开启
61     if (Status.Close.equals(beforeStatus)) {
62         if (Status.Open.equals(afterStatus)) {
63             ActivityService.execStatus(activityId, beforeStatus,
64                                         afterStatus);
65             return new Result("0000", "变更状态成功");
66         } else {
67             return new Result("0001", "变更状态拒绝");
68         }
69     }
70
71     // 6. 活动开启 -> 关闭
72     if (Status.Open.equals(beforeStatus)) {
73         if (Status.Close.equals(afterStatus)) {
74             ActivityService.execStatus(activityId, beforeStatus,
75                                         afterStatus);
76             return new Result("0000", "变更状态成功");
77         } else {
78             return new Result("0001", "变更状态拒绝");
79         }
80     }
81 }
82
83 }
```

- 这里我们只需要看一下代码实现的结构即可。从上到下是一整篇的 `ifelse`，基本这也是大部分初级程序员的开发方式。
- 这样的面向过程式开发方式，对于不需要改动代码，也不需要二次迭代的，还是可以使用的（但基本不可能不迭代）。而且随着状态和需求变化，会越来越难以维护，后面的人也不好看懂并且很容易填充其他的流程进去。`越来越乱就是从点滴开始的`

3. 测试验证

3.1 编写测试类

```

1  @Test
2  public void test() {
3      // 初始化数据
4      String activityId = "100001";
5      ActivityService.init(activityId, Status.Editing);
6
7      ActivityExecStatusController activityExecStatusController = new
ActivityExecStatusController();
8      Result resultRefuse =
activityExecStatusController.execStatus(activityId, Status.Editing,
Status.Refuse);
9      logger.info("测试结果(编辑中到审核拒绝): {}", JSON.toJSONString(resultRefuse));
10
11     Result resultCheck =
activityExecStatusController.execStatus(activityId, Status.Editing,
Status.Check);
12     logger.info("测试结果(编辑中到提交审核): {}", JSON.toJSONString(resultCheck));
13 }

```

- 我们的测试代码包括了两个功能的验证，一个是从 编辑中 到 审核拒绝，另外一个是编辑中到 提交审核。
- 因为从我们的场景流程中可以看到，编辑中的活动是不能直接到 审核拒绝 的，这中间还需要 提审。

3.2 测试结果

```

1  23:24:30.774 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(编
辑中到审核拒绝): {"code": "0001", "info": "变更状态拒绝"}
2  23:24:30.778 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(编
辑中到提交审核): {"code": "0000", "info": "变更状态成功"}
3
4  Process finished with exit code 0

```

- 从测试结果和我们的状态流程的流转中可以看到，是符合测试结果预期的。除了不好维护外，这样的开发过程还是蛮快的，但不建议这么搞！

五、状态模式重构代码

接下来使用状态模式来进行代码优化，也算是一次很小的重构。

重构的重点往往是处理掉 `if else`，而想处理掉 `if else` 基本离不开接口与抽象类，另外还需要重新改造代码结构。

1. 工程结构

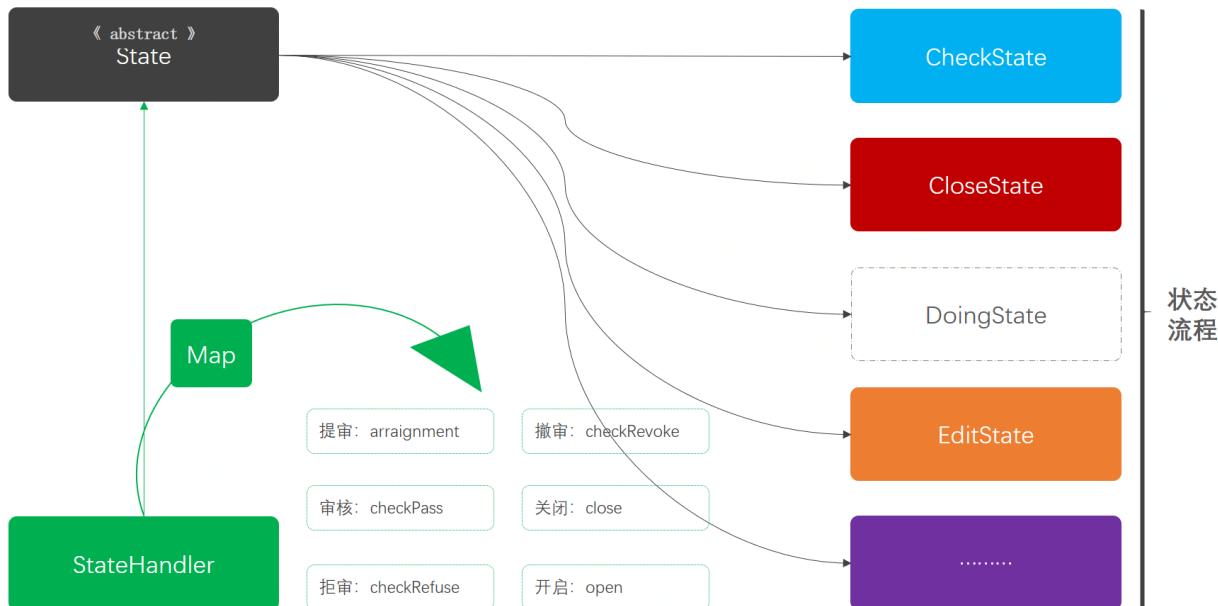
```
1  itstack-demo-design-19-02
```

```

2   └── src
3       └── main
4           └── java
5               └── org.itstack.demo.design
6                   └── event
7                       ├── CheckState.java
8                       ├── CloseState.java
9                       ├── DoingState.java
10                      ├── EditingState.java
11                      ├── OpenState.java
12                      ├── PassState.java
13                      └── RefuseState.java
14
15                  └── Result.java
16
17                  └── State.java
18
19                  └── StateHandler.java

```

状态模式模型结构



- 以上是状态模式的整个工程结构模型，`State`是一个抽象类，定义了各种操作接口(提审、审核、拒审等)。
- 右侧的不同颜色状态与我们场景模拟中的颜色保持一致，是各种状态流程流转的实现操作。这里的实现有一个关键点就是每一种状态到下一个状态，都分配到各个实现方法中控制，也就不需要 `if` 语言进行判断了。
- 最后是 `stateHandler` 对状态流程的统一处理，里面提供 `Map` 结构的各项服务接口调用，也就避免了使用 `if` 判断各项状态转变的流程。

2. 代码实现

2.1 定义状态抽象类

```

1  public abstract class State {
2
3      /**

```

```
4     * 活动提审
5     *
6     * @param activityId    活动ID
7     * @param currentStatus 当前状态
8     * @return 执行结果
9     */
10    public abstract Result arraignment(String activityId, Enum<Status>
currentStatus);
11
12    /**
13     * 审核通过
14     *
15     * @param activityId    活动ID
16     * @param currentStatus 当前状态
17     * @return 执行结果
18     */
19    public abstract Result checkPass(String activityId, Enum<Status>
currentStatus);
20
21    /**
22     * 审核拒绝
23     *
24     * @param activityId    活动ID
25     * @param currentStatus 当前状态
26     * @return 执行结果
27     */
28    public abstract Result checkRefuse(String activityId, Enum<Status>
currentStatus);
29
30    /**
31     * 撤审撤销
32     *
33     * @param activityId    活动ID
34     * @param currentStatus 当前状态
35     * @return 执行结果
36     */
37    public abstract Result checkRevoke(String activityId, Enum<Status>
currentStatus);
38
39    /**
40     * 活动关闭
41     *
42     * @param activityId    活动ID
43     * @param currentStatus 当前状态
44     * @return 执行结果
45     */
46    public abstract Result close(String activityId, Enum<Status>
currentStatus);
47
```

```

48     /**
49      * 活动开启
50      *
51      * @param activityId    活动ID
52      * @param currentStatus 当前状态
53      * @return 执行结果
54      */
55     public abstract Result open(String activityId, Enum<Status>
currentStatus);
56
57     /**
58      * 活动执行
59      *
60      * @param activityId    活动ID
61      * @param currentStatus 当前状态
62      * @return 执行结果
63      */
64     public abstract Result doing(String activityId, Enum<Status>
currentStatus);
65
66 }

```

- 在整个接口中提供了各项状态流转服务的接口，例如；活动提审、审核通过、审核拒绝、撤审撤销等7个方法。
- 在这些方法中所有的入参都是一样的，activityId(活动ID)、currentStatus(当前状态)，只有他们的具体实现是不同的。

2.2 部分状态流转实现

编辑

```

1  public class EditingState extends State {
2
3      public Result arraignment(String activityId, Enum<Status>
currentStatus) {
4          ActivityService.execStatus(activityId, currentStatus,
Status.Check);
5          return new Result("0000", "活动提审成功");
6      }
7
8      public Result checkPass(String activityId, Enum<Status> currentStatus)
{
9          return new Result("0001", "编辑中不可审核通过");
10 }
11
12     public Result checkRefuse(String activityId, Enum<Status>
currentStatus) {
13         return new Result("0001", "编辑中不可审核拒绝");
14     }

```

```
15
16     @Override
17     public Result checkRevoke(String activityId, Enum<Status>
18     currentStatus) {
18         return new Result("0001", "编辑中不可撤销审核");
19     }
20
21     public Result close(String activityId, Enum<Status> currentStatus) {
22         ActivityService.execStatus(activityId, currentStatus,
23         Status.Close);
23         return new Result("0000", "活动关闭成功");
24     }
25
26     public Result open(String activityId, Enum<Status> currentStatus) {
27         return new Result("0001", "非关闭活动不可开启");
28     }
29
30     public Result doing(String activityId, Enum<Status> currentStatus) {
31         return new Result("0001", "编辑中活动不可执行活动中变更");
32     }
33
34 }
```

提审

```
1 public class CheckState extends State {
2
3     public Result arraignment(String activityId, Enum<Status>
4     currentStatus) {
4         return new Result("0001", "待审核状态不可重复提审");
5     }
6
7     public Result checkPass(String activityId, Enum<Status> currentStatus)
8     {
8         ActivityService.execStatus(activityId, currentStatus,
9         Status.Pass);
9         return new Result("0000", "活动审核通过完成");
10    }
11
12    public Result checkRefuse(String activityId, Enum<Status>
13    currentStatus) {
13        ActivityService.execStatus(activityId, currentStatus,
14        Status.Refuse);
14        return new Result("0000", "活动审核拒绝完成");
15    }
16
17    @Override
18    public Result checkRevoke(String activityId, Enum<Status>
19    currentStatus) {
```

```

19         ActivityService.execStatus(activityId, currentStatus,
20         Status.Editing);
21         return new Result("0000", "活动审核撤销回到编辑中");
22     }
23
24     public Result close(String activityId, Enum<Status> currentStatus) {
25         ActivityService.execStatus(activityId, currentStatus,
26         Status.Close);
27         return new Result("0000", "活动审核关闭完成");
28     }
29
30     public Result open(String activityId, Enum<Status> currentStatus) {
31         return new Result("0001", "非关闭活动不可开启");
32     }
33
34     public Result doing(String activityId, Enum<Status> currentStatus) {
35         return new Result("0001", "待审核活动不可执行活动中变更");
36     }

```

- 这里提供了两个具体实现类的内容，编辑状态和提审状态。
- 例如在这两个实现类中，`checkRefuse`这个方法对于不同的类中有不同的实现，也就是不同状态下能做的下一步流转操作已经可以在每一个方法中具体控制了。
- 其他5个类的操作是类似的具体就不在这里演示了，大部分都是重复代码。可以通过源码进行学习理解。

2.3 状态处理服务

```

1  public class StateHandler {
2
3      private Map<Enum<Status>, State> stateMap = new
4          ConcurrentHashMap<Enum<Status>, State>();
5
6      public StateHandler() {
7          stateMap.put(Status.Check, new CheckState());           // 待审核
8          stateMap.put(Status.Close, new CloseState());          // 已关闭
9          stateMap.put(Status.Doing, new DoingState());         // 活动中
10         stateMap.put(Status.Editing, new EditingState());       // 编辑中
11         stateMap.put(Status.Open, new OpenState());            // 已开启
12         stateMap.put(Status.Pass, new PassState());           // 审核通过
13         stateMap.put(Status.Refuse, new RefuseState());        // 审核拒绝
14     }
15
16     public Result arraignment(String activityId, Enum<Status>
17         currentStatus) {
18         return stateMap.get(currentStatus).arraignment(activityId,
19             currentStatus);
20     }

```

```

18     public Result checkPass(String activityId, Enum<Status> currentStatus)
19     {
20         return stateMap.get(currentStatus).checkPass(activityId,
21               currentStatus);
22     }
23
24     public Result checkRefuse(String activityId, Enum<Status>
25               currentStatus) {
26         return stateMap.get(currentStatus).checkRefuse(activityId,
27               currentStatus);
28     }
29
30
31     public Result checkRevoke(String activityId, Enum<Status>
32               currentStatus) {
33         return stateMap.get(currentStatus).checkRevoke(activityId,
34               currentStatus);
35     }
36
37     public Result close(String activityId, Enum<Status> currentStatus) {
38         return stateMap.get(currentStatus).close(activityId,
39               currentStatus);
40     }
41
42     public Result open(String activityId, Enum<Status> currentStatus) {
43         return stateMap.get(currentStatus).open(activityId,
44               currentStatus);
45     }
46
47     public Result doing(String activityId, Enum<Status> currentStatus) {
48         return stateMap.get(currentStatus).doing(activityId,
49               currentStatus);
50     }
51
52 }

```

- 这是对状态服务的统一控制中心，可以看到在构造函数中提供了所有状态和实现的具体关联，放到 Map 数据结构中。
- 同时提供了不同名称的接口操作类，让外部调用方可以更加容易的使用此项功能接口，而不需要像在 `itstack-demo-design-19-01` 例子中还得传两个状态来判断。

3. 测试验证

3.1 编写测试类(**Editing2Arraignment**)

```
1  @Test
2  public void test_Editing2Arraignment() {
3      String activityId = "100001";
4      ActivityService.init(activityId, Status.Editing);
5      StateHandler stateHandler = new StateHandler();
6      Result result = stateHandler.arraignment(activityId, Status.Editing);
7      logger.info("测试结果(编辑中To提审活动): {}", JSON.toJSONString(result));
8      logger.info("活动信息: {} 状态: {}",
9          JSON.toJSONString(ActivityService.queryActivityInfo(activityId)),
10         JSON.toJSONString(ActivityService.queryActivityInfo(activityId).getStatus()
11     );
12 }
13 }
```

测试结果

```
1  23:59:20.883 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(编
2  辑中To提审活动): {"code":"0000","info":"活动提审成功"}
3  23:59:20.907 [main] INFO org.itstack.demo.design.test.ApiTest - 活动信
4  息: {"activityId":"100001","activityName":"早起学习打卡领奖活
5  动","beginTime":1593694760892,"endTime":1593694760892,"status":"Check"} 状
6  态: "Check"
7
8  Process finished with exit code 0
```

- 测试编辑中To提审活动， 的状态流转。

3.2 编写测试类(Editing2Open)

```
1  @Test
2  public void test_Editing2Open() {
3      String activityId = "100001";
4      ActivityService.init(activityId, Status.Editing);
5      StateHandler stateHandler = new StateHandler();
6      Result result = stateHandler.open(activityId, Status.Editing);
7      logger.info("测试结果(编辑中To开启活动): {}", JSON.toJSONString(result));
8      logger.info("活动信息: {} 状态: {}",
9          JSON.toJSONString(ActivityService.queryActivityInfo(activityId)),
10         JSON.toJSONString(ActivityService.queryActivityInfo(activityId).getStatus()
11     );
12 }
13 }
```

测试结果

```
1 23:59:36.904 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(编  
辑中To开启活动): {"code":"0001","info":"非关闭活动不可开启"}  
2 23:59:36.914 [main] INFO org.itstack.demo.design.test.ApiTest - 活动信  
息: {"activityId":"100001","activityName":"早起学习打卡领奖活  
动","beginTime":1593694776907,"endTime":1593694776907,"status":"Editing"} 状  
态: "Editing"  
3  
4 Process finished with exit code 0
```

- 测试编辑中To开启活动， 的状态流转。

3.3 编写测试类(Refuse2Doing)

```
1 @Test  
2 public void test_Refuse2Doing() {  
3     String activityId = "100001";  
4     ActivityService.init(activityId, Status.Refuse);  
5     StateHandler stateHandler = new StateHandler();  
6     Result result = stateHandler.doing(activityId, Status.Refuse);  
7     logger.info("测试结果(拒绝To活动中): {}", JSON.toJSONString(result));  
8     logger.info("活动信息: {} 状态: {}",  
9                 JSON.toJSONString(ActivityService.queryActivityInfo(activityId)),  
10                JSON.toJSONString(ActivityService.queryActivityInfo(activityId).getStatus()  
11            ));  
12 }
```

测试结果

```
1 23:59:46.339 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(拒  
绝To活动中): {"code":"0001","info":"审核拒绝不可执行活动为进行中"}  
2 23:59:46.352 [main] INFO org.itstack.demo.design.test.ApiTest - 活动信  
息: {"activityId":"100001","activityName":"早起学习打卡领奖活  
动","beginTime":1593694786342,"endTime":1593694786342,"status":"Refuse"} 状  
态: "Refuse"  
3  
4 Process finished with exit code 0
```

- 测试拒绝To活动中， 的状态流转。

3.4 编写测试类(Refuse2Revoke)

```

1  @Test
2  public void test_Refuse2Revoke() {
3      String activityId = "100001";
4      ActivityService.init(activityId, Status.Refuse);
5      StateHandler stateHandler = new StateHandler();
6      Result result = stateHandler.checkRevoke(activityId, Status.Refuse);
7      logger.info("测试结果(拒绝To撤审): {}", JSON.toJSONString(result));
8      logger.info("活动信息: {} 状态: {}",
9      JSON.toJSONString(ActivityService.queryActivityInfo(activityId)),
10     JSON.toJSONString(ActivityService.queryActivityInfo(activityId).getStatus()
11 );
12 }

```

测试结果

```

1  23:59:50.197 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果(拒
绝To撤审): {"code":"0000","info":"撤销审核完成"}
2  23:59:50.208 [main] INFO org.itstack.demo.design.test.ApiTest - 活动信
息: {"activityId":"100001","activityName":"早起学习打卡领奖活
动","beginTime":1593694810201,"endTime":1593694810201,"status":"Editing"} 状
态: "Editing"
3
4  Process finished with exit code 0

```

- 测试测试结果(拒绝To撤审), 的状态流转。
- 综上以上四个测试类分别模拟了不同状态之间的 有效流转 和 拒绝流转 , 不同的状态服务处理不同的服务内容。

六、总结

- 从以上的两种方式对一个需求的实现中可以看到, 在第二种使用设计模式处理后已经没有了 `if else`, 代码的结构也更加清晰易于扩展。这就是设计模式的好处, 可以非常强大的改变原有代码的结构, 让以后的扩展和维护都变得容易些。
- 在实现结构的编码方式上可以看到这不再是面向过程的编程, 而是面向对象的结构。并且这样的设计模式满足了 单一职责 和 开闭原则 , 当你只有满足这样的结构下才会发现代码的扩展是容易的, 也就是增加和修改功能不会影响整体的变化。
- 但如果状态和各项流转较多像本文的案例中, 就会产生较多的实现类。因此可能也会让代码的实现上带来了时间成本, 因为如果遇到这样的场景可以按需评估投入回报率。主要点在于看是否经常修改、是否可以做成组件化、抽离业务与非业务功能。

第8节：策略模式

文无第一，武无第二

不同方向但同样努力的人，都有自身的价值和亮点，也都是可以互相学习的。不要太过于用自己手里的矛去攻击别人的盾，哪怕一时争辩过了也多半可能是你被安放的角色不同。取别人之强补自己之弱，矛与盾的结合可能就是坦克。

能把复杂的知识讲的简单很重要

在学习的过程中我们看过很多资料、视频、文档等，因为现在资料视频都较多所以往往一个知识点会有多种多样的视频形式讲解。除了推广营销以外，确实有很多人的视频讲解非常优秀，例如李永乐老师的短视频课，可以在一个黑板上把那么复杂的知识，讲解的那么容易理解，那么透彻。而我们学习编程的人也是，不只是要学会把知识点讲明白，也要写明白。

提升自己的眼界交往更多同好

有时候圈子很重要，就像上学期间大家都会发现班里有这样一类学生不怎么听课，但是就是学习好。那假如让他回家呆着，不能在课堂里呢？类似的圈子还有；图书馆、网吧、车友群、技术群等等，都可以给你带来同类爱好的人所分享出来的技能或者大家一起烘托出的氛围帮你成长。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-20-01	使用一坨代码实现业务需求
itstack-demo-design-20-02	通过设计模式优化改造代码，产生对比性从而学习

二、策略模式介绍



策略模式是一种行为模式，也是替代大量 `if else` 的利器。它所能帮你解决的是场景，一般是具有同类可替代的行为逻辑算法场景。比如：不同类型的交易方式(信用卡、支付宝、微信)、生成唯一ID策略 (UUID、DB自增、DB+Redis、雪花算法、Leaf算法)等，都可以使用策略模式进行行为包装，供给外部使用。



策略模式也有点像三国演义中诸葛亮给刘关张的锦囊；

- 第一个锦囊：见乔国老，并把刘备娶亲的事情du搞得东吴人尽皆知。
- 第二个锦囊：用谎言（曹操打荆州）骗泡在温柔乡里的刘备回去。
- 第三个锦囊：让孙夫人摆平东吴的追兵，她是孙权妹妹，东吴将领惧她三分。

三、案例场景模拟



在本案例中我们模拟在购买商品时候使用的各种类型优惠券(满减、直减、折扣、n元购)

这个场景几乎也是大家的一个日常购物省钱渠道，购买商品的时候都希望找一些优惠券，让购买的商品更加实惠。而且到了大促的时候就会有更多的优惠券需要计算那些商品一起购买更加优惠！！！

这样的场景有时候用户用起来还是蛮爽的，但是最初这样功能的设定以及产品的不断迭代，对于程序员*💻*开发还是不太容易的。因为这里包括了很多的规则和优惠逻辑，所以我们模拟其中的一个计算优惠的方式，使用策略模式来实现。

四、用一坨坨代码实现

这里我们先使用最粗暴的方式来实现功能

对于优惠券的设计最初可能非常简单，就是一个金额的折扣，也没有现在这么多种类型。所以如果没有这样场景的经验话，往往设计上也是非常简单的。但随着产品功能的不断迭代，如果程序最初设计的不具备很好的扩展性，那么往后就会越来越混乱。

1. 工程结构

```

1 itstack-demo-design-20-01
2 └─ src
3   └─ main
4     └─ java
5       └─ org.itstack.demo.design
6         └─ CouponDiscountService.java

```

- 一坨坨 工程的结构很简单，也是最直接的面向过程开发方式。

2. 代码实现

```

1  /**
2  * 博客: https://bugstack.cn - 沉淀、分享、成长，让自己和他人都能有所收获!
3  * 公众号: bugstack虫洞栈
4  * Create by 小傅哥(fustack) @2020
5  * 优惠券折扣计算接口
6  * <p>
7  * 优惠券类型;
8  * 1. 直减券
9  * 2. 满减券
10 * 3. 折扣券
11 * 4. n元购
12 */
13 public class CouponDiscountService {
14
15     public double discountAmount(int type, double typeContent, double
16 skuPrice, double typeExt) {
17         // 1. 直减券
18         if (1 == type) {
19             return skuPrice - typeContent;
20         }
21         // 2. 满减券
22         if (2 == type) {
23             if (skuPrice < typeExt) return skuPrice;
24             return skuPrice - typeContent;
25         }
26         // 3. 折扣券
27         if (3 == type) {
28             return skuPrice * typeContent;
29         }
30         // 4. n元购
31         if (4 == type) {
32             return typeContent;
33         }
34     }
35
36 }
```

- 以上是不同类型的优惠券计算折扣后的实际金额。
- 入参包括：优惠券类型、优惠券金额、商品金额，因为有些优惠券是满多少减少多少，所以增加了 `typeExt` 类型。这也是方法的不好扩展性问题。
- 最后是整个的方法体中对优惠券抵扣金额的实现，最开始可能是一个最简单的优惠券，后面随着产品功能的增加，不断的扩展 `if` 语句。实际的代码可能要比这个多很多。

五、策略模式重构代码

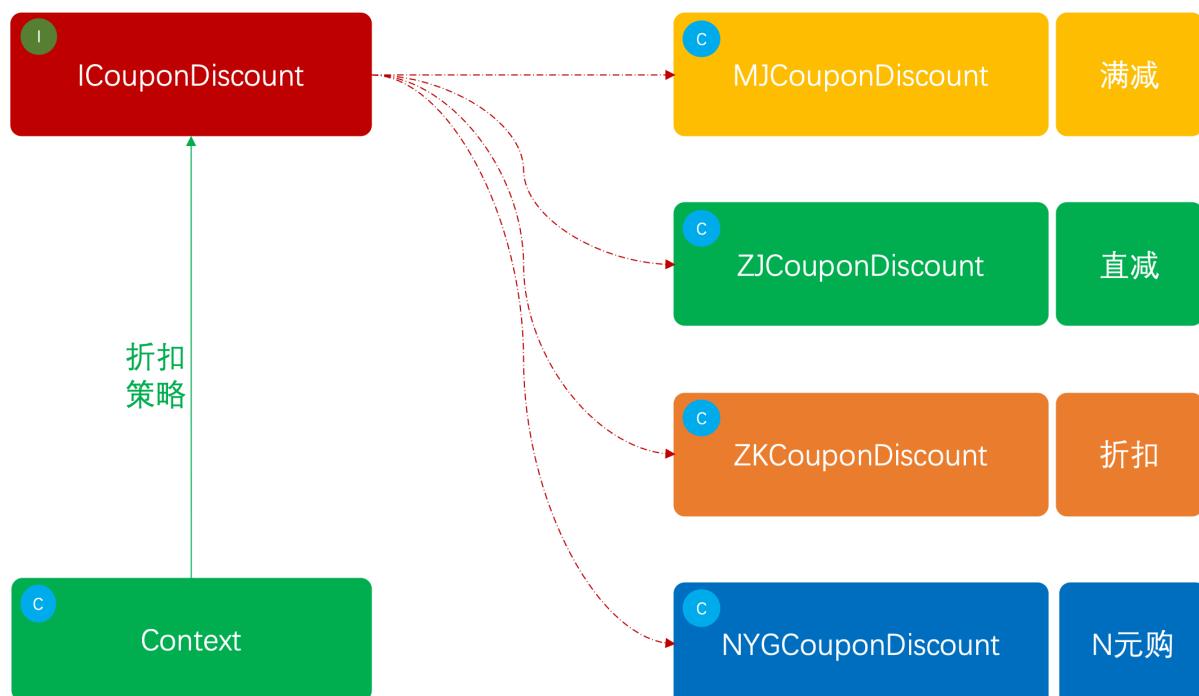
接下来使用策略模式来进行代码优化，也算是一次很小的重构。

与上面面向流程式的开发这里会使用设计模式，优惠代码结构，增强整体的扩展性。

1. 工程结构

```
1 itstack-demo-design-20-02
2 └── src
3     └── main
4         └── java
5             └── org.itstack.demo.design
6                 ├── event
7                 │   └── MJCouponDiscount.java
8                 │   └── NYGCouponDiscount.java
9                 │   └── ZJCouponDiscount.java
10                └── ZKCouponDiscount.java
11                └── Context.java
12                └── ICouponDiscount.java
```

策略模式模型结构



- 整体的结构模式并不复杂，主要体现的不同类型的优惠券在计算优惠券方式的不同计算策略。
- 这里包括一个借口类(`ICouponDiscount`)以及四种优惠券类型的实现方式。
- 最后提供了策略模式的上下控制类处理，整体的策略服务。

2. 代码实现

2.1 优惠券接口

```

1 public interface ICouponDiscount<T> {
2
3     /**
4      * 优惠券金额计算
5      * @param couponInfo 券折扣信息；直减、满减、折扣、N元购
6      * @param skuPrice   sku金额
7      * @return           优惠后金额
8      */
9     BigDecimal discountAmount(T couponInfo, BigDecimal skuPrice);
10
11 }
12

```

- 定义了优惠券折扣接口，也增加了泛型用于不同类型的接口可以传递不同的类型参数。
- 接口中包括商品金额以及出参返回最终折扣后的金额，这里在实际开发中会比现在的接口参数多一些，但核心逻辑是这些。

2.2 优惠券接口实现

满减

```

1 public class MJCouponDiscount implements
2     ICouponDiscount<Map<String, String>> {
3
4     /**
5      * 满减计算
6      * 1. 判断满足x元后-n元，否则不减
7      * 2. 最低支付金额1元
8      */
9     public BigDecimal discountAmount(Map<String, String> couponInfo,
10        BigDecimal skuPrice) {
11         String x = couponInfo.get("x");
12         String o = couponInfo.get("n");
13
14         // 小于商品金额条件的，直接返回商品原价
15         if (skuPrice.compareTo(new BigDecimal(x)) < 0) return skuPrice;
16         // 减去优惠金额判断
17         BigDecimal discountAmount = skuPrice.subtract(new BigDecimal(o));
18         if (discountAmount.compareTo(BigDecimal.ZERO) < 1) return
19             BigDecimal.ONE;
20
21         return discountAmount;
22     }
23 }

```

直减

```

1 public class ZJCouponDiscount implements ICouponDiscount<Double> {

```

```
2
3     /**
4      * 直减计算
5      * 1. 使用商品价格减去优惠价格
6      * 2. 最低支付金额1元
7      */
8     public BigDecimal discountAmount(Double couponInfo, BigDecimal
9     skuPrice) {
10        BigDecimal discountAmount = skuPrice.subtract(new
11        BigDecimal(couponInfo));
12        if (discountAmount.compareTo(BigDecimal.ZERO) < 1) return
13        BigDecimal.ONE;
14        return discountAmount;
15    }
16}
```

折扣

```
1 public class ZKCouponDiscount implements ICouponDiscount<Double> {
2
3
4     /**
5      * 折扣计算
6      * 1. 使用商品价格乘以折扣比例，为最后支付金额
7      * 2. 保留两位小数
8      * 3. 最低支付金额1元
9      */
10    public BigDecimal discountAmount(Double couponInfo, BigDecimal
11    skuPrice) {
12        BigDecimal discountAmount = skuPrice.multiply(new
13        BigDecimal(couponInfo)).setScale(2, BigDecimal.ROUND_HALF_UP);
14        if (discountAmount.compareTo(BigDecimal.ZERO) < 1) return
15        BigDecimal.ONE;
16        return discountAmount;
17    }
18}
```

N元购

```

1 public class NYGCouponDiscount implements ICouponDiscount<Double> {
2
3     /**
4      * n元购购买
5      * 1. 无论原价多少钱都固定金额购买
6      */
7     public BigDecimal discountAmount(Double couponInfo, BigDecimal
8     skuPrice) {
9         return new BigDecimal(couponInfo);
10    }
11}

```

- 以上是四种不同类型的优惠券计算折扣金额的策略方式，可以从代码中看到每一种优惠方式的优惠金额。

2.3 策略控制类

```

1 public class Context<T> {
2
3     private ICouponDiscount<T> couponDiscount;
4
5     public Context(ICouponDiscount<T> couponDiscount) {
6         this.couponDiscount = couponDiscount;
7     }
8
9     public BigDecimal discountAmount(T couponInfo, BigDecimal skuPrice) {
10        return couponDiscount.discountAmount(couponInfo, skuPrice);
11    }
12
13}

```

- 策略模式的控制类主要是外部可以传递不同的策略实现，在通过统一的方法执行优惠策略计算。
- 另外这里也可以包装成map结构，让外部只需要对应的泛型类型即可使用相应的服务。

3. 测试验证

3.1 编写测试类(直减优惠)

```

1 @Test
2 public void test_zj() {
3     // 直减：100-10，商品100元
4     Context<Double> context = new Context<Double>(new ZJCouponDiscount());
5     BigDecimal discountAmount = context.discountAmount(10D, new
6     BigDecimal(100));
7     logger.info("测试结果：直减优惠后金额 {}", discountAmount);
8 }

```

测试结果

```
1 15:43:22.035 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果：直  
2 减优惠后金额 90  
3 Process finished with exit code 0
```

3.2 编写测试类(满减优惠)

```
1 @Test  
2 public void test_mj() {  
3     // 满100减10，商品100元  
4     Context<Map<String, String>> context = new Context<Map<String, String>>  
(new MJCouponDiscount());  
5     Map<String, String> mapReq = new HashMap<String, String>();  
6     mapReq.put("x", "100");  
7     mapReq.put("n", "10");  
8     BigDecimal discountAmount = context.discountAmount(mapReq, new  
9     BigDecimal(100));  
10    logger.info("测试结果：满减优惠后金额 {}", discountAmount);  
11 }
```

测试结果

```
1 15:43:42.695 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果：满  
2 减优惠后金额 90  
3 Process finished with exit code 0
```

3.3 编写测试类(折扣优惠)

```
1 @Test  
2 public void test_zk() {  
3     // 折扣9折，商品100元  
4     Context<Double> context = new Context<Double>(new ZKCouponDiscount());  
5     BigDecimal discountAmount = context.discountAmount(0.9D, new  
6     BigDecimal(100));  
7     logger.info("测试结果：折扣9折后金额 {}", discountAmount);  
8 }
```

测试结果

```
1 15:44:05.602 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果：折  
2 扣9折后金额 90.00  
3 Process finished with exit code 0
```

3.4 编写测试类(n元购优惠)

```
1  @Test
2  public void test_nyg() {
3      // n元购; 100-10, 商品100元
4      Context<Double> context = new Context<Double>(new NYGCouponDiscount());
5      BigDecimal discountAmount = context.discountAmount(90D, new
6      BigDecimal(100));
7      logger.info("测试结果: n元购优惠后金额 {}", discountAmount);
```

测试结果

```
1  15:44:24.700 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果: n
2  元购优惠后金额 90
3  Process finished with exit code 0
```

- 以上四组测试分别验证了不同类型优惠券的优惠策略，测试结果是满足我们的预期。
- 这里四种优惠券最终都是在原价 100元 上折扣 10元，最终支付 90元。

六、总结

- 以上的策略模式案例相对来说并不复杂，主要的逻辑都是体现在关于不同种类优惠券的计算折扣策略上。结构相对来说也比较简单，在实际的开发中这样的设计模式也是非常常用的。另外这样的设计与命令模式、适配器模式结构相似，但是思路是有差异的。
- 通过策略设计模式的使用可以把我们方法中的if语句优化掉，大量的if语句使用会让代码难以扩展，也不好维护，同时在后期遇到各种问题也很难维护。在使用这样的设计模式后可以很好的满足隔离性与和扩展性，对于不断新增的需求也非常方便承接。
- 策略模式、适配器模式、组合模式等，在一些结构上是比较相似的，但是每一个模式有自己的逻辑特点，在使用的过程中最佳的方式是经过较多的实践来吸取经验，为后续的研发设计提供更好的技术输出。

第9节：模板模式

黎明前的坚守，的住吗？

有人举过这样一个例子，先给你张北大的录取通知书，但要求你每天5点起床，12点睡觉😴，刻苦学习，勤奋上进。只要你坚持三年，这张通知书就有效。如果是你，你能坚持吗？其实对于这个例子很难在我们的人生中出现，因为它目标明确，有准确的行军路线。就像你是土豪家庭，家里给你安排的明明白白一样，只要你按照这个方式走就不会有问题。可大多数时候我们并没有这样的路线，甚至不知道多久到达自己的黎明。但！谁又不渴望见到黎明呢，坚持吧！

不要轻易被洗脑

键盘侠💻、网络喷壶，几乎当你努力坚持一件事的时候，在这条路上会遇到形形色色的人和事。有时候接收建议完善自己是有必要的，但不能放弃自己的初心和底线，有时候只坚持自己也是难能可贵的。**子路之勇，子贡之辩，冉有之智，此三子者，皆天下之所谓难能而可贵者也。**阳光和努力是这个世界最温暖的东西，加油坚持好自己的选的路。

有时还好坚持了

当你为自己的一个决定而感到万分开心😊时，是不是也非常感谢自己还好坚持了。坚持、努力、终身学习，似乎在程序员这个行业是离不开的，当你愿意于把这当做一份可以努力的爱好时，你就会愿意为此而努力。而我们很难说只在机会要来时准备，而是一直努力等待机会。也就是很多人说的别人抓住机会是因为一直在准备着。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-21-00	场景模拟工程：模拟爬虫商品生成海报场景

二、模版模式介绍

模板模式的核心设计思路是通过在，抽象类中定义抽象方法的执行顺序，并将抽象方法设定为只有子类实现，但不设计独立访问的方法。简单说也就是把你安排的明明白白的。

就像西游记的99八十一难，基本每一关都是；师傅被掳走、打妖怪、妖怪被收走，具体什么妖怪你自己定义，怎么打你想办法，最后收走还是弄死看你本事，我只定义执行顺序和基本策略，具体的每一难由观音来安排。

三、案例场景模拟

在本案例中我们模拟爬虫各类电商商品，生成营销推广海报场景

关于模版模式的核心点在于由抽象类定义抽象方法执行策略，也就是说父类规定了好一系列的执行标准，这些标准的串联成一整套业务流程。

在这个场景中我们模拟爬虫爬取各类商家的商品信息，生成推广海报(海报中含带个人的邀请码)赚取商品返利。声明，这里是模拟爬取，并没有真的爬取

而整个的爬取过程分为；模拟登录、爬取信息、生成海报，这三个步骤，另外；

1. 因为有些商品只有登录后才可以爬取，并且登录可以看到一些特定的价格这与未登录用户看到的价格不同。
2. 不同的电商网站爬取方式不同，解析方式也不同，因此可以作为每一个实现类中的特定实现。
3. 生成海报的步骤基本一样，但会有特定的商品来源标识。所以这样三个步骤可以使用模版模式来设定，并有具体的场景做子类实现。

四、模版模式搭建工程

模版模式的业务场景可能在平时的开发中并不是很多，主要因为这个设计模式会在抽象类中定义逻辑行为的执行顺序。一般情况下，我们用的抽象类定义的逻辑行为都比较轻量级或者没有，只是提供一些基本方法公共调用和实现。

但如果遇到适合的场景使用这样的设计模式也是非常方便的，因为他可以控制整套逻辑的执行顺序和统一的输入、输出，而对于实现方只需要关心好自己的业务逻辑即可。

而在我们这个场景中，只需要记住这三步的实现即可； 模拟登录、爬取信息、生成海报

1. 工程结构

```
1 itstack-demo-design-21-00
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── group
7     │           │   ├── DangDangNetMall.java
8     │           │   ├── JDNetMall.java
9     │           │   └── TaoBaoNetMall.java
10    │           ├── HttpClient.java
11    │           └── NetMall.java
12    └── test
13        └── java
14            └── org.itstack.demo.design.test
15                └── ApiTest.java
```

模版模式模型结构

- 以上的代码结构还是比较简单的，一个定义了抽象方法执行顺序的核心抽象类，以及三个模拟具体

的实现(京东、淘宝、当当)的电商服务。

2. 代码实现

2.1 定义执行顺序的抽象类

```
1  /**
2  * 基础电商推广服务
3  * 1. 生成最优价商品海报
4  * 2. 海报含带推广邀请码
5  */
6  public abstract class NetMall {
7
8      protected Logger logger = LoggerFactory.getLogger(NetMall.class);
9
10     String uId;    // 用户ID
11     String uPwd;   // 用户密码
12
13     public NetMall(String uId, String uPwd) {
14         this.uId = uId;
15         this.uPwd = uPwd;
16     }
17
18     /**
19      * 生成商品推广海报
20      *
21      * @param skuUrl 商品地址(京东、淘宝、当当)
22      * @return 海报图片base64位信息
23      */
24     public String generateGoodsPoster(String skuUrl) {
25         if (!login(uId, uPwd)) return null;           // 1. 验证登录
26         Map<String, String> reptile = reptile(skuUrl); // 2. 爬虫商品
27         return createBase64(reptile);                 // 3. 组装海报
28     }
29
30     // 模拟登录
31     protected abstract Boolean login(String uId, String uPwd);
32
33     // 爬虫提取商品信息(登录后的优惠价格)
34     protected abstract Map<String, String> reptile(String skuUrl);
35
36     // 生成商品海报信息
37     protected abstract String createBase64(Map<String, String> goodsInfo);
38
39 }
```

- 这个类是此设计模式的灵魂
- 定义可被外部访问的方法 `generateGoodsPoster`，用于生成商品推广海报
- `generateGoodsPoster` 在方法中定义抽象方法的执行顺序 1 2 3 步

- 提供三个具体的抽象方法，让外部继承方实现；模拟登录(`login`)、模拟爬取(`reptile`)、生成海报(`createBase64`)

2.2 模拟爬虫京东

```

1  public class JDNetMall extends NetMall {
2
3      public JDNetMall(String uId, String uPwd) {
4          super(uId, uPwd);
5      }
6
7      public Boolean login(String uId, String uPwd) {
8          logger.info("模拟京东用户登录 uId: {} uPwd: {}", uId, uPwd);
9          return true;
10     }
11
12     public Map<String, String> reptile(String skuUrl) {
13         String str = HttpClient.doGet(skuUrl);
14         Pattern p9 = Pattern.compile("(?=<title\\>).*?(?=)");
15         Matcher m9 = p9.matcher(str);
16         Map<String, String> map = new ConcurrentHashMap<String, String>();
17         if (m9.find()) {
18             map.put("name", m9.group());
19         }
20         map.put("price", "5999.00");
21         logger.info("模拟京东商品爬虫解析: {} | {} 元 {}", map.get("name"),
22             map.get("price"), skuUrl);
23         return map;
24     }
25
26     public String createBase64(Map<String, String> goodsInfo) {
27         BASE64Encoder encoder = new BASE64Encoder();
28         logger.info("模拟生成京东商品base64海报");
29         return encoder.encode(JSON.toJSONString(goodsInfo).getBytes());
30     }
31 }
```

- 模拟登录
- 爬取信息，这里只是把`title`的信息爬取后的结果截取出来。
- 模拟创建`base64`图片的方法

2.3 模拟爬虫淘宝

```

1  public class TaoBaoNetMall extends NetMall {
2
3      public TaoBaoNetMall(String uId, String uPwd) {
4          super(uId, uPwd);
5      }

```

```

6
7     @Override
8     public Boolean login(String uId, String uPwd) {
9         logger.info("模拟淘宝用户登录 uId: {} uPwd: {}", uId, uPwd);
10        return true;
11    }
12
13    @Override
14    public Map<String, String> reptile(String skuUrl) {
15        String str = HttpClient.doGet(skuUrl);
16        Pattern p9 = Pattern.compile("(?=<title\\>).*?(?=)");
17        Matcher m9 = p9.matcher(str);
18        Map<String, String> map = new ConcurrentHashMap<String, String>();
19        if (m9.find()) {
20            map.put("name", m9.group());
21        }
22        map.put("price", "4799.00");
23        logger.info("模拟淘宝商品爬虫解析: {} | {} 元 {}", map.get("name"),
24        map.get("price"), skuUrl);
25        return map;
26    }
27
28    @Override
29    public String createBase64(Map<String, String> goodsInfo) {
30        BASE64Encoder encoder = new BASE64Encoder();
31        logger.info("模拟生成淘宝商品base64海报");
32        return encoder.encode(JSONObject.toJSONString(goodsInfo).getBytes());
33    }
34}

```

- 同上，模拟登录和爬取以及创建图片的 base64

2.4 模拟爬虫当当

```

1 public class DangDangNetMall extends NetMall {
2
3     public DangDangNetMall(String uId, String uPwd) {
4         super(uId, uPwd);
5     }
6
7     @Override
8     public Boolean login(String uId, String uPwd) {
9         logger.info("模拟当当用户登录 uId: {} uPwd: {}", uId, uPwd);
10        return true;
11    }
12
13    @Override
14    public Map<String, String> reptile(String skuUrl) {

```

```

15     String str = HttpClient.doGet(skuUrl);
16     Pattern p9 = Pattern.compile("(?=<title\\>).*?(?=</title>)"); 
17     Matcher m9 = p9.matcher(str);
18     Map<String, String> map = new ConcurrentHashMap<String, String>();
19     if (m9.find()) {
20         map.put("name", m9.group());
21     }
22     map.put("price", "4548.00");
23     logger.info("模拟当当商品爬虫解析: {} | {} 元 {}", map.get("name"),
24     map.get("price"), skuUrl);
25     return map;
26 }
27 @Override
28 public String createBase64(Map<String, String> goodsInfo) {
29     BASE64Encoder encoder = new BASE64Encoder();
30     logger.info("模拟生成当当商品base64海报");
31     return encoder.encode(JSONObject.toJSONString(goodsInfo).getBytes());
32 }
33 }
34 }
```

- 同上，模拟登录和爬取以及创建图片的 base64

3. 测试验证

3.1 编写测试类

```

1 /**
2  * 测试链接
3  * 京东; https://item.jd.com/100008348542.html
4  * 淘宝; https://detail.tmall.com/item.htm
5  * 当当; http://product.dangdang.com/1509704171.html
6  */
7 @Test
8 public void test_NetMall() {
9     NetMall netMall = new JDNetMall("1000001", "*****");
10    String base64 =
11        netMall.generateGoodsPoster("https://item.jd.com/100008348542.html");
12    logger.info("测试结果: {}", base64);
13 }
```

- 测试类提供了三个商品链接，也可以是其他商品的链接
- 爬取的过程模拟爬取京东商品，可以替换为其他商品服务 new JDNetMall、new TaoBaoNetMall、new DangDangNetMall

3.2 测试结果

```
1 23:33:13.616 [main] INFO org.itstack.demo.design.NetMall - 模拟京东用户登录
  uId: 1000001 uPwd: *****
2 23:33:15.038 [main] INFO org.itstack.demo.design.NetMall - 模拟京东商品爬虫解析: 【AppleiPhone 11】Apple iPhone 11 (A2223) 128GB 黑色 移动联通电信4G手机 双卡
  双待【行情 报价 价格 评测】-京东 | 5999.00 元
  https://item.jd.com/100008348542.html
3 23:33:15.038 [main] INFO org.itstack.demo.design.NetMall - 模拟生成京东商品
  base64海报
4 23:33:15.086 [main] INFO org.itstack.demo.design.test.ApiTest - 测试结果:
  eyJwcmljZSI6IjU5OTkuMDAiLCJuYWlIjoi44CQQXBwbGVpUGhvbmUgMTHjgJFBcHBsZSBpUGH
  v
5  bmUgMTEgKEEyMjIzKSAxMjhHQiDpu5HoibIg56e75Yqo6IGU6YCa55S15L+hNEfmiYvmnLog5Y+
  M
6  5Y2h5Y+M5b6F44CQ6KGM5oOFIOaKpeS7tyDku7fmoLwg6K+E5rWL44CRLeS6rOS4nCJ9
7
8  Process finished with exit code 0
```

五、总结

- 通过上面的实现可以看到模版模式在定义统一结构也就是执行标准上非常方便，也就很好的控制了后续的实现者不用关心调用逻辑，按照统一方式执行。那么类的继承者只需要关心具体的业务逻辑实现即可。
- 另外模版模式也是为了解决子类通用方法，放到父类中设计的优化。让每一个子类只做子类需要完成的内容，而不需要关心其他逻辑。这样提取公用代码，行为由父类管理，扩展可变部分，也就非常有利于开发拓展和迭代。
- 但每一种设计模式都有自己的特定场景，如果超过场景外的建设就需要额外考虑🤔其他模式的运用。而不是非要生搬硬套，否则自己不清楚为什么这么做，也很难让后续者继续维护代码。而想要活学活用就需要多加练习，有实践的经历。

第 10 节：访问者模式

能力，是你前行的最大保障

年龄会不断的增长，但是什么才能让你不慌张。一定是能力，即使是在一个看似还很安稳的工作中也是一样，只有拥有能留下的本事和跳出去的能力，你才会是安稳的。而能力的提升是不断突破自己的未知也就是拓展宽度，以及在专业领域建设个人影响力也就是深度。如果日复日365天，天天搬砖，一切都沒有变化的重复只能让手上增长点老茧，岁月又叹人生苦短。

站得高看的远吗？

站得高确实能看得远，也能给自己更多的追求。但，站的高了，原本看的清的东西就变得看不清了。视角和重点的不同，会让我们有很多不同的选择，而脚踏实地是给自己奠定能攀升起来的基石，当真的可以四平八稳的走向山头的时候，才是适合看到更远的时候。

数学好才能学编码吗

往往很多时候学编程的初学者都会问数学不好能学会吗？其实可以想想那为什么数学不好呢？在这条没学好的路上，你为它们付出了多少时间呢？如果一件事情你敢做到和写自己名字一样熟悉，还真的有难的东西吗。从大学到毕业能写出40万行代码的，还能愁找不到工作吗，日积月累，每一天并没有多难。难的你想用最后一个月的时间学完人家四年努力的成绩的。学习，要趁早。

一、开发环境

1. JDK 1.8
2. Idea + Maven
3. 涉及工程三个，可以通过关注公众号：[bugstack虫洞栈](#)，回复 源码下载 获取(打开获取的链接，找到序号18)

工程	描述
itstack-demo-design-22-00	场景模拟工程；模拟学生和老师信息不同视角访问

二、访问者模式介绍

访问者要解决的核心事项是，在一个稳定的数据结构下，例如用户信息、雇员信息等，增加易变的业务访问逻辑。为了增强扩展性，将这两部分的业务解耦的一种设计模式。

说白了访问者模式的核心在于同一个事物不同视角下的访问信息不同，比如一个美女手里拿个冰激凌。小朋友会注意冰激凌，大朋友会找自己喜欢的地方观测敌情。

三、案例场景模拟

在本案例中我们模拟校园中的学生和老师对于不同用户的访问视角

这个案例场景我们模拟校园中有学生和老师两种身份的用户，那么对于家长和校长关心的角度来看，他们的视角是不同的。家长更关心孩子的成绩和老师的能力，校长更关心老师所在班级学生的人数和升学率{此处模拟的}。

那么这样 学生 和 老师 就是一个固定信息的内容，而想让不同视角的用户获取关心的信息，就比较适合使用观察者模式来实现，从而让实体与业务解耦，增强扩展性。但观察者模式的整体类结构相对复杂，需要梳理清楚再开发

四、访问者模式搭建工程

访问者模式的类结构相对其他设计模式来说比较复杂，但这样的设计模式在我看来更加 烧气有魅力，它能开阔你对代码结构的新认知，用这样思维不断的建设出更好的代码架构。

关于这个案例的核心逻辑实现，有以下几点：

1. 建立用户抽象类和抽象访问方法，再由不同的用户实现；老师和学生。
2. 建立访问者接口，用于不同人员的访问操作；校长和家长。
3. 最终是对数据的看板建设，用于实现不同视角的访问结果输出。

1. 工程结构

```
1 itstack-demo-design-22-00
2 └── src
3     ├── main
4     │   └── java
5     │       └── org.itstack.demo.design
6     │           ├── user
7     │           │   └── impl
8     │           │       ├── Student.java
9     │           │       ├── Teacher.java
10    │           │       └── User.java
11    │           └── visitor
12    │               └── impl
13    │                   ├── Parent.java
14    │                   ├── Principal.java
15    │                   └── Visitor.java
16    └── DataView.java
17
18 └── test
19     └── java
20         └── org.itstack.demo.design.test
21             └── ApiTest.java
```

访问者模式模型结构

以上是视图展示了代码的核心结构，主要包括不同视角下的不同用户访问模型。

在这里有一个关键的点非常重要，也就是整套设计模式的核心组成部分：`visitor.visit(this)`，这个方法在每一个用户实现类里，包括：`Student`、`Teacher`。在以下的实现中可以重点关注。

2. 代码实现

2.1 定义用户抽象类

```
1 // 基础用户信息
2 public abstract class User {
3
4     public String name;          // 姓名
5     public String identity;    // 身份：重点班、普通班 | 特级教师、普通教师、实习教
6         师
7     public String clazz;        // 班级
8
9     public User(String name, String identity, String clazz) {
10         this.name = name;
11         this.identity = identity;
12         this.clazz = clazz;
13     }
14
15     // 核心访问方法
16     public abstract void accept(Visitor visitor);
17 }
```

- 基础信息包括：姓名、身份、班级，也可以是一个业务用户属性类。
- 定义抽象核心方法，`abstract void accept(Visitor visitor)`，这个方法是为了让后续的用户具体实现者都能提供出一个访问方法，共外部使用。

2.2 实现用户信息(老师和学生)

老师类

```
1 public class Teacher extends User {
2
3     public Teacher(String name, String identity, String clazz) {
4         super(name, identity, clazz);
5     }
6
7     public void accept(Visitor visitor) {
8         visitor.visit(this);
9     }
10
11     // 升本率
12     public double entranceRatio() {
13         return BigDecimal.valueOf(Math.random() * 100).setScale(2,
14             BigDecimal.ROUND_HALF_UP).doubleValue();
15     }
16 }
```

```
15  
16 }
```

学生类

```
1 public class Student extends User {  
2  
3     public Student(String name, String identity, String clazz) {  
4         super(name, identity, clazz);  
5     }  
6  
7     public void accept(Visitor visitor) {  
8         visitor.visit(this);  
9     }  
10  
11    public int ranking() {  
12        return (int) (Math.random() * 100);  
13    }  
14  
15 }
```

- 这里实现了老师和学生类，都提供了父类的构造函数。
- 在 `accept` 方法中，提供了本地对象的访问；`visitor.visit(this)`，这块需要加深理解。
- 老师和学生类又都单独提供了各自的特性方法；升本率(`entranceRatio`)、排名(`ranking`)，类似这样的方法可以按照业务需求进行扩展。

2.3 定义访问数据接口

```
1 public interface Visitor {  
2  
3     // 访问学生信息  
4     void visit(Student student);  
5  
6     // 访问老师信息  
7     void visit(Teacher teacher);  
8  
9 }
```

- 访问的接口比较简单，相同的方法名称，不同的入参用户类型。
- 让具体的访问者类，在实现时可以关注每一种用户类型的具体访问数据对象，例如；升学率和排名。

2.4 实现访问类型(校长和家长)

访问者；校长

```
1 public class Principal implements Visitor {  
2  
3     private Logger logger = LoggerFactory.getLogger(Principal.class);  
4  
5     public void visit(Student student) {  
6         logger.info("学生信息 姓名: {} 班级: {}", student.name,  
7             student.clazz);  
8     }  
9  
10    public void visit(Teacher teacher) {  
11        logger.info("学生信息 姓名: {} 班级: {} 升学率: {}", teacher.name,  
12            teacher.clazz, teacher.entranceRatio());  
13    }  
14}
```

访问者；家长

```
1 public class Parent implements Visitor {  
2  
3     private Logger logger = LoggerFactory.getLogger(Parent.class);  
4  
5     public void visit(Student student) {  
6         logger.info("学生信息 姓名: {} 班级: {} 排名: {}", student.name,  
7             student.clazz, student.ranking());  
8     }  
9  
10    public void visit(Teacher teacher) {  
11        logger.info("老师信息 姓名: {} 班级: {} 级别: {}", teacher.name,  
12            teacher.clazz, teacher.identity());  
13    }  
14}
```

- 以上是两个具体的访问者实现类，他们都有自己的视角需求。
- 校长关注：学生的名称和班级，老师对这个班级的升学率
- 家长关注：自己家孩子的排名，老师的班级和教学水平

2.5 数据看版

```
1 public class DataView {  
2  
3     List<User> userList = new ArrayList<User>();  
4  
5     public DataView() {  
6         userList.add(new Student("谢飞机", "重点班", "一年一班"));  
7         userList.add(new Student("windy", "重点班", "一年一班"));  
8         userList.add(new Student("大毛", "普通班", "二年三班"));  
9     }  
10}
```

```

9     userList.add(new Student("Shing", "普通班", "三年四班"));
10    userList.add(new Teacher("BK", "特级教师", "一年一班"));
11    userList.add(new Teacher("娜娜Goddess", "特级教师", "一年一班"));
12    userList.add(new Teacher("dangdang", "普通教师", "二年三班"));
13    userList.add(new Teacher("泽东", "实习教师", "三年四班"));
14 }
15
16 // 展示
17 public void show(Visitor visitor) {
18     for (User user : userList) {
19         user.accept(visitor);
20     }
21 }
22
23 }

```

- 首先在这个类中初始化了基本的数据，学生和老师的信息。
- 并提供了一个展示类，通过传入不同的观察者(校长、家长)而差异化的打印信息。

3. 测试验证

3.1 编写测试类

```

1 @Test
2 public void test(){
3     DataView dataView = new DataView();
4
5     logger.info("\r\n家长视角访问: ");
6     dataView.show(new Parent());      // 家长
7
8     logger.info("\r\n校长视角访问: ");
9     dataView.show(new Principal());  // 校长
10 }

```

- 从测试类可以看到，家长和校长分别是不同的访问视角。

3.2 测试结果

```

1 23:00:39.726 [main] INFO org.itstack.demo.design.test.ApiTest -
2 家长视角访问:
3 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 学生信息 姓名: 谢飞机 班级: 一年一班 排名: 62
4 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 学生信息 姓名: windy 班级: 一年一班 排名: 51
5 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 学生信息 姓名: 大毛 班级: 二年三班 排名: 16
6 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 学生信息 姓名: Shing 班级: 三年四班 排名: 98

```

```

7 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 老师信息 姓名: BK 班级: 一年一班 级别: 特级教师
8 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 老师信息 姓名: 娜娜Goddess 班级: 一年一班 级别: 特级教师
9 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 老师信息 姓名: dangdang 班级: 二年三班 级别: 普通教师
10 23:00:39.730 [main] INFO o.i.demo.design.visitor.impl.Parent - 老师信息 姓名: 泽东 班级: 三年四班 级别: 实习教师
11 23:00:39.730 [main] INFO org.itstack.demo.design.test.ApiTest -
12 校长视角访问:
13 23:00:39.731 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: 谢飞机 班级: 一年一班
14 23:00:39.731 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: windy 班级: 一年一班
15 23:00:39.731 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: 大毛 班级: 二年三班
16 23:00:39.731 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: Shing 班级: 三年四班
17 23:00:39.733 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: BK 班级: 一年一班 升学率: 70.62
18 23:00:39.733 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: 娜娜Goddess 班级: 一年一班 升学率: 23.15
19 23:00:39.734 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: dangdang 班级: 二年三班 升学率: 70.98
20 23:00:39.734 [main] INFO o.i.d.design.visitor.impl.Principal - 学生信息 姓名: 泽东 班级: 三年四班 升学率: 90.14
21
22 Process finished with exit code 0

```

- 通过测试结果可以看到，家长和校长的访问视角同步，数据也是差异化的。
- 家长视角看到学生的排名；`排名: 62`、`排名: 51`、`排名: 16`、`排名: 98`。
- 校长视角看到班级升学率；`升学率: 70.62`、`升学率: 23.15`、`升学率: 70.98`、`升学率: 90.14`。
- 通过这样的测试结果，可以看到访问者模式的初心和结果，在适合的场景运用合适的模式，非常有利于程序开发。

五、总结

- 从以上的业务场景中可以看到，在嵌入访问者模式后，可以让整个工程结构变得容易添加和修改。也就做到了系统服务之间的解耦，不至于为了不同类型信息的访问而增加很多多余的`if`判断或者类的强制转换。也就是通过这样的设计模式而让代码结构更加清晰。
- 另外在实现的过程可能你可能也发现了，定义抽象类的时候还需要等待访问者接口的定义，这样的设计首先从实现上会让代码的组织变得有些难度。另外从设计模式原则的角度来看，违背了迪米特原则，也就是最少知道原则。因此在使用上一定要符合场景的运用，以及提取这部分设计思想的精髓。
- 好的学习方式才好更容易接受知识，学习编程的更需要的不单单是看，而是操作。二十多种设计模式每一种都有自己的设计技巧，也可以说是巧妙之处，这些巧妙的地方往往是解决复杂难题的最佳视角。亲力亲为，才能为所欲为，为了自己的欲望而努力！

结尾

截止到此设计模式内容就全部讲完了，可能书中会因作者水平有限，有一些描述不准确或者错字内容。欢迎提交给我，也欢迎和我讨论相关的技术内容，作者小傅哥，非常愿意与同好进行交流，互相提升技术。