# 重构的那些事儿 - Jun.M - 博客园

http://www.cnblogs.com/jun-ma/p/4967839.html

几天前的一次上线，脑残手抖不小心写了bug，虽然组里的老大没有说什么，但心里面很是难过。同事说我之所以写虫子是因为我讨厌if/else，这个习惯不好。的确，if/else可以帮助我们很方便的写出流程控制代码，简洁明了，这个条件做什么，那个条件做什么，说得很清楚。说真的，我从来不反对if/else，从经验上看，越复杂的业务场景下，代码写的越简单单一，通常越不容易出错。以结果为导向的现代项目管理方式，这是一种很有效实践经验。

同事说的没错，我的确很讨厌if/else。这个习惯很大程度是受Thoughtworks一位咨询师朋友影响，他经常在我耳边唠叨，写代码要干净，要简洁，要灵活多变，不要固守城规，不要动不动就if/else，switch/case。初入it领域，我一直把这句话奉为经典。在以后的学习工作中也时刻提醒自己要让自己的代码尽可能的看起来简洁，不失灵活。不喜欢if/else并不意味着拒绝它，该使用的时候必要使用，比如函数接口入参check，处理异常分支逻辑流程等。通常能不用分支语句，我尽量不会使用，因为我觉得if/else很丑，每每看到if/else代码，总会以挑剔的眼光看待它，想想能不能重构的更好。大多数时候，关于什么好的代码，大家的意见往往分歧很大，每个人都有各自的想法，审查你代码的人可能会选择另一种实现方式，这并不能说明谁对谁错。

OO设计遵循SOLID(单一功能、开闭原则、里氏替换、接口隔离以及依赖反转)原则，使用这个原则去审视if/else，可能会发现很多问题，比如不符合单一原则，它本身就像一团浆糊，融合了各种作料，黏糊糊的很不干净；比如不符合开闭原则，每新增一种场景，就需要修改源文件增加一条分支语句，业务逻辑复杂些若有1000种场景就得有1000个分支流，这种情况下代码不仅仅恶心问题了，效率上也存在很大问题。由此可见，if/else虽然简单方便，但不恰当的使用会给编码代码带来非常痛苦的体验。针对这种恶心的if/else分支，我们当然首先想到的去重构它--在不改变代码外部功能特征的前提下对代码内部逻辑进行调整和优化，但，如何做呢？前段时间在项目中正好遇到一个恶心的if/else例子，想在这篇博客里和大家分享一下去除if/else重构的历程。



## if/else的恶瘤

有句话说的好--好文章是改出来，同样，好的代码也肯定是重构出来的，因为没有哪个软件工程师能够拍着胸脯保证在项目之初代码设计这块，就考虑到了所有需求变化可能性的扩展。随着项目的不断成长，业务逻辑变的越来越复杂，代码也开始变的越来越多，原有的设计可能不再满足需求，那么此时必须要重构。就系统整体架构而言，重构可能需要很大的改动，可能在架构流程上需要评审；就功能内代码层次而言，这种重构在我们编码过程中随时可以进行，类似于if/else，swicth/case这种代码的重构也属于这种类型。今天我们要重构的if/else源码如下所示，针对不同的status code，CountRecoder对象会执行不同的set方法，为不同内部属性赋值。

public CountRecoder getCountRecoder(List<CountEntry> countEntries) {

CountRecoder countRecoder = new CountRecoder();

for (CountEntry countEntry : countEntries) {

if (1 == countEntry.getCode()) {

countRecoder.setCountOfFirstStage(countEntry.getCount());

} else if (2 == countEntry.getCode()) {

countRecoder.setCountOfSecondStage(countEntry.getCount());

} else if (3 == countEntry.getCode()) {

countRecoder.setCountOfThirdtage(countEntry.getCount());

} else if (4 == countEntry.getCode()) {

countRecoder.setCountOfForthtage(countEntry.getCount());

} else if (5 == countEntry.getCode()) {

countRecoder.setCountOfFirthStage(countEntry.getCount());

} else if (6 == countEntry.getCode()) {

countRecoder.setCountOfSixthStage(countEntry.getCount());

}

}

return countRecoder;

}

CountRecoder对象是一个简单的Java Bean，用于保存一天之中六种状态分别对应的数据条目，提供了get和set方法。CountEntry是对应数据库中每种状态的数据条目记录，包含状态code和以及count两个字段, 我们可以使用mybatis实现数据库记录和java对象之间的转换。上面getCountRecoder的方法实现了将list转换为CountRecoder的功能。

看到这段代码，想必已经有很多人要呵呵了，像一坨啥啥啥，长得这么丑，真不知道它"爸妈"怎么想的，怎么敢"生"出来。啥都不说了，直接回炉重构吧。重构是门艺术，Martin flow曾写过一本书《重构改变代码之道》，里面详细的记录了重构的方法论，感兴趣的朋友可以阅读一下。说到重构，通常我们在重构中会遇到一个问题，那就是如何能够保证重构的代码不改变原有的外部功能特征 ？经过TDD训练的朋友应该知道答案，那就是单元测试，重构之前要写单元测试，准确的来说应该是补单元测试，毕竟TDD的核心理念是测试驱动开发。对于今天博客中分享的例子，因为代码逻辑比较简单，所以偷了懒，省却了单元测试的历程。

## 重构初体验--反射

要重构上面的代码，对设计模式精通的人可以立马可以看出来这是使用策略模式/状态模式的绝佳场景，将策略模式稍微变换，工厂模式应该也是ok的，当然也有些人会选择使用反射。对于这些方法，这里不一一列出，主要想讲一下使用反射和工厂模式如何解决消除if/else问题，那先说反射吧，代码如下所示：

private static Map<Integer, String> methodsMap = new HashMap<>();

static {

methodsMap.put(1, "setCountOfFirstStage");

methodsMap.put(2, "setCountOfSecondStage");

methodsMap.put(3, "setCountOfThirdtage");

methodsMap.put(4, "setCountOfForthtage");

methodsMap.put(5, "setCountOfFirthStage");

methodsMap.put(6, "setCountOfSixthStage");

}

public CountRecoder getCountRecoderByReflect(List<CountEntry> countEntries) {

CountRecoder countRecoder = new CountRecoder();

countEntries.stream().forEach(countEntry -> fillCount(countRecoder, countEntry));

return countRecoder;

}

private void fillCount(CountRecoder shippingOrderCountDto, CountEntry countEntry) {

String name = methodsMap.get(countEntry.getCode());

try {

Method declaredMethod = CountRecoder.class.getMethod(name, Integer.class);

declaredMethod.invoke(shippingOrderCountDto, countEntry.getCount());

} catch (Exception e) {

System.out.println(e);

}

}

## 重构初体验--所谓模式

使用反射去掉if/else的原理很简单，使用HashMap建立状态码和需要调用的方法的方法名之间的映射关系，对于每个CountEntry，首先取出状态码，然后根据状态码获得相应的要调用方法的方法名，然后使用java的反射机制就可以实现对应方法的调用了。本例中使用反射的确可以帮助我们完美的去掉if/else的身影，但是，众所周知，反射效率很低，在高并发的条件下，反射绝对不是一个良好的选择。除去反射这种方法，能想到的就剩下使用策略模式或者与其类似的状态模式，以及工厂模式了，我们以工厂模式为例，经典的架构UML架构图通常由三个组成要素：

1. 抽象产品角色：通常是一个抽象类或者接口，里面定义了抽象方法
2. 具体产品角色：具体产品的实现类，继承或是实现抽象策略类，通常由一个或多个组成类组成。
3. 工厂角色：持有抽象产品类的引用，负责动态运行时产品的选择和构建

策略模式的架构图和工厂模式非常类似，不过在策略模式里执行的对象不叫产品，叫策略。在本例中，这里的产品是虚拟产品，它是服务类性质的接口或者实现。Ok，按照工厂模式的思路重构我们的代码，我们首先定义一个抽象产品接口FillCountService，里面定义产品的行为方法fillCount，代码如下所示：

public interface FillCountService {

void fillCount(CountRecoder countRecoder, int count);

}

接着我们需要分别实现这六种服务类型的产品，在每种产品中封装不同的服务算法，具体的代码如下所示：

class FirstStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfFirstStage(count);

}

}

class SecondStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfSecondStage(count);

}

}

class ThirdStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfThirdtage(count);

}

}

class ForthStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfForthtage(count);

}

}

class FirthStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfFirthStage(count);

}

}

class SixthStageService implements FillCountService {

@Override

public void fillCount(CountRecoder countRecoder, int count) {

countRecoder.setCountOfSixthStage(count);

}

}

紧接着，我们需要是实现工厂角色，在工厂内需要实现产品的动态选择算法，使用HashMap维护状态code和具体产品的对象之间的映射关系，  
就可以非常容易的实现这一点，具体代码如下所示：

public class FillCountServieFactory {

private static Map<Integer, FillCountService> fillCountServiceMap = new HashMap<>();

static {

fillCountServiceMap.put(1, new FirstStageService());

fillCountServiceMap.put(2, new SecondStageService());

fillCountServiceMap.put(3, new ThirdStageService());

fillCountServiceMap.put(4, new ForthStageService());

fillCountServiceMap.put(5, new FirthStageService());

fillCountServiceMap.put(6, new SixthStageService());

}

public static FillCountService getFillCountStrategy(int statusCode) {

return fillCountServiceMap.get(statusCode);

}

}

客户端在具体使用的时候就变的很简单，那getCountRecoder方法就可以用下面的代码实现:

public CountRecoder getCountRecoder(List<CountEntry> countEntries) {

CountRecoder countRecoder = new CountRecoder();

countEntries.stream().forEach(countEntry ->

FillCountServieFactory.getFillCountStrategy(countEntry.getCode())

.fillCount(countRecoder, countEntry.getCount()));

return countRecoder;

}

## 重构初体验--Java8对模式设计的精简

和反射一样使用设计模式也同样完美的去除了if/else，但是不得不引入大量的具体服务实现类，同时程序中出现大量的模板代码，使得我们程序看起来很不干净，幸好Java 8之后引入了Functional Interface，我们可以使用lambda表达式来去除这些模板代码。将一个接口变为Functional interface，可以通过在接口上添加FunctionalInterface注解实现，代码如下所示：

@FunctionalInterface

public interface FillCountService {

void fillCount(CountRecoder countRecoder, int count);

}

那么具体的服务实现类就可以使用一个简单的lambda表达式代替，原先的FirstStageService类对象就可以使用下面的表达式代替：

(countRecoder, count) -> countRecoder.setCountOfFirstStage(count)

那么工厂类中的代码就可以变为：

public class FillCountServieFactory {

private static Map<Integer, FillCountService> fillCountServiceMap = new HashMap<>();

static {

fillCountServiceMap.put(1, (countRecoder, count) -> countRecoder.setCountOfFirstStage(count));

fillCountServiceMap.put(2, (countRecoder, count) -> countRecoder.setCountOfSecondStage(count));

fillCountServiceMap.put(3, (countRecoder, count) -> countRecoder.setCountOfThirdtage(count));

fillCountServiceMap.put(4, (countRecoder, count) -> countRecoder.setCountOfForthtage(count));

fillCountServiceMap.put(5, (countRecoder, count) -> countRecoder.setCountOfFirthStage(count));

fillCountServiceMap.put(6, (countRecoder, count) -> countRecoder.setCountOfSixthStage(count));

}

public static FillCountService getFillCountStrategy(int statusCode) {

return fillCountServiceMap.get(statusCode);

}

}

这样我们的代码就重构完毕了，当然了还是有些不完美，程序中的魔法数字不利于阅读理解，可以使用易读的常量标识它们，在这里就不做过多说明了。

## 总结

Craig Larman曾经说过软件开发最重要的设计工具不是什么技术，而是一颗在设计原则方面训练有素的头脑。重构的最终结果不一定会让代码变少，相反还有可能增加程序的复杂度和抽象性，就本例中的if/else而言，确实如此。我非常赞同我的一位朋友说的话，做技术要有追求，没错if/else可以在代码中工作的挺好，也可以很容易的被接替者所理解，但是我们可以有更好的选择，因为简单的代码也可以变得很精彩。多勤多思，也许有一天真的就可以达到Craig所说的在设计原则方面拥有训练有素的头脑，谁说不是这样呢？加油吧。

欢迎转载

# 你还在用if else吗？ - 解道Jdon

http://www.jdon.com/artichect/ifelse.htm

你还在用if else吗？

[**板桥里人**](http://www.jdon.com/aboutme.htm) http://www.jdon.com 2006/1/11（转载请保留）

　　面向过程设计和面向对象设计的主要区别是：是否在业务逻辑层使用冗长的if else判断。如果你还在大量使用if else，当然，界面表现层除外，即使你使用Java/C#这样完全面向对象的语言，也只能说明你的思维停留在传统的面向过程语言上。

**传统思维习惯分析**

　　为什么会业务逻辑层使用if else，其实使用者的目的也是为了重用，但是这是面向过程编程的重用，程序员只看到代码重用，因为他看到if else几种情况下大部分代码都是重复的，只有个别不同，因此使用if else可以避免重复代码，并且认为这是模板Template模式。

　　他范的错误是：程序员只从代码运行顺序这个方向来看待它的代码，这种思维类似水管或串行电路，水沿着水管流动（代码运行次序），当遇到几个分管（子管），就分到这几个分管子在流动，这里就相当于碰到代码的if else处了。

　　而使用OO，则首先打破这个代码由上向下顺序等同于运行时的先后循序这个规律，代码结构不由执行循序决定，由什么决定呢？由OO设计；设计模式会取代这些if else，但是最后总是由一个Service等总类按照运行顺序组装这些OO模块，只有一处，这处可包含事务，一般就是Service，EJB中是Session bean。

　　一旦需求变化，我们更多的可能是Service中各个OO模块，甚至是只改动Service中的OO模块执行顺序就能符合需求。

　　这里我们也看到OO分离的思路，将以前过程语言的一个Main函数彻底分解，将运行顺序与代码其他逻辑分离开来，而不是象面向过程那样混乱在一起。所以有人感慨，OO也是要顺序的，这是肯定的，关键是运行顺序要单独分离出来。

　　是否有if else可以看出你有没有将运行顺序分离到家。

**设计模式的切入口**

　　经常有人反映，设计模式是不错，但是我很难用到，其实如果你使用if else来写代码时（除显示控制以外），就是在写业务逻辑，只不过使用简单的判断语句来作为现实情况的替代者。

　　 还是以大家熟悉的论坛帖子为例子，如ForumMessage是一个模型，但是实际中帖子分两种性质：主题贴（第一个根贴）和回帖（回以前帖子的帖子），这里有一个朴素的解决方案：  
建立一个ForumMessage，然后在ForumMessage加入isTopic这样判断语句，注意，你这里一个简单属性的判断引入，可能导致你的程序其他地方到处存在if else 的判断。

　　如果我们改用另外一种分析实现思路，以对象化概念看待，实际中有主题贴和回帖，就是两种对象，但是这两种对象大部分是一致的，因此，我将ForumMessage设为表达主题贴；然后创建一个继承ForumMessage的子类ForumMessageReply作为回帖，这样，我在程序地方，如Service中，我已经确定这个Model是回帖了，我就直接下溯为ForumMessageReply即可，这个有点类似向Collection放入对象和取出时的强制类型转换。通过这个手段我消灭了以后程序中if else的判断语句出现可能。

　　从这里体现了，如果分析方向错误，也会导致误用模式。

　　讨论设计模式举例，不能没有业务上下文场景的案例，否则无法决定是否该用模式，下面举两个对比的例子：

　　第一.[**这个帖子**](http://www.jdon.com/24681)中举例的第一个代码案例是没有上下文的，文中只说明有一段代码：

|  |  |
| --- | --- |
| main() {  if（case A）{  //do with strategy A  }else(case B){  //do with strategy B  }else(case C){  //do with strategy C  }  } |  |

　　这段代码只是纯粹的代码，没有业务功能，所以，在这种情况下，我们就很难确定使用什么模式，就是一定用策略模式等，也逃不过还是使用if else的命运，设计模式不是魔法，不能将一段毫无意义的代码变得简单了，只能将其体现的业务功能更加容易可拓展了。

　　第二.在[**这个帖子**](http://www.jdon.com/24333&message=14484871#14484871)中，作者举了一个PacketParser业务案例，这段代码是体现业务功能的，是一个数据包的分析，作者也比较了各种模式使用的不同，所以我们还是使用动态代理模式或Command模式来消灭那些可能存在的if else

　　由以上两个案例表明：业务逻辑是我们使用设计模式的切入点，而在分解业务逻辑时，我们习惯则可能使用if else来实现，当你有这种企图或者已经实现代码了，那么就应该考虑是否需要重构Refactoring了。

**if else替代者**

　　那么实战中，哪些设计模式可以替代if else呢？其实GoF设计模式都可以用来替代if else，我们分别描述如下：

 状态模式　  
　　当数据对象存在各种可能性的状态，而且这种状态将会影响到不同业务结果时，那么我们就应该考虑是否使用状态模式，当然，使用状态模式之前，你必须首先有内存状态这个概念，而不是数据库概念，因为在传统的面向过程的/面向数据库的系统中，你很难发现状态的，从数据库中读取某个值，然后根据这个值进行代码运行分流，这是很多初学者常干的事情。参考文章:[**状态对象：数据库的替代者**](http://www.jdon.com/artichect/state.htm)  
　　使用传统语言思维的情况还有：使用一个类整数变量标识状态：

|  |
| --- |
| public class Order{  private int status;  //说明：  //status=1 表示订货但为查看 ；  //status=2 表示已经查看未处理；  //status=3 表示已经处理未付款  //status=4 表示已经付款未发货  //status=5 表示已经发货  } |

　　上述类设计，无疑是将类作为传统语言的函数来使用，这样导致程序代码中存在大量的if else。

 策略模式　  
　　当你面临几种算法或者公式选择时，可以考虑策略模式，传统过程语言情况是：从数据库中读取算法数值，数值1表示策略1，例如保存到数据库；数值为2表示策略2，例如保存到XMl文件中。这里使用if else作为策略选择的开关。 

 command模式　  
　　传统过程的思维情况是：如果客户端发出代号是1或"A"，那么我调用A.java这个对象来处理；如果代号是2或"B"，我就调用B.java来处理，通过if else来判断客户端发送过来的代码，然后按事先约定的对应表，调用相应的类来处理。

 MVC模式　  
　　MVC模式的传统语言误用和Command模式类似，在一个Action类中，使用if else进行前后台调度，如果客户端传送什么命令；我就调用后台什么结果；如果后台处理什么结构，再决定推什么页面，不过，现在我们使用Struts/JSF这样MVC模式的框架实现者就不必范这种低级错误。

 职责链模式　  
　　职责链模式和Command模式是可选的，如果你实在不知道客户端会发出什么代号；也没有一个事先定义好的对照表，那么你只能编写一个个类去碰运气一样打开这个包看一下就可以。与Command是不同在[**AOP vs Decorator**](http://www.jdon.com/AOPdesign/decorator.htm)一文中有分析。

 代理或动态代理模式　  
　　代理对象可以是符合某种条件的代表者，比如，权限检验，传统面向过程思维是：当一个用户登陆后，访问某资源时，使用if else进行判断，只有某种条件符合时，才能允许访问，这样权限判断和业务数据逻辑混乱在一起，使用代理模式可以清晰分离，如果嫌不太好，使用动态代理，或者下面AOP等方式。

 AOP或Decorator模式  
　　  
　　其实使用filter过滤器也可以替代我们业务中的if else，过滤器起到一种过滤和筛选作用，将符合本过滤器条件的对象拦截下来做某件事情，这就是一个过滤器的功能，多个过滤器组合在一起实际就是if else的组合。  
　　所以，如果你实在想不出什么办法，可以使用过滤器，将过滤器看成防火墙就比较好理解，当客户端有一个请求时，经过不同性质的防火墙，这个防火墙是拦截端口的；那个防火墙是安全检查拦截等等。过滤器也如同红蓝白各种光滤镜；红色滤镜只能将通过光线中的红色拦截了；蓝色滤镜将光线中的蓝色拦截下来，这实际上是对光线使用if else进行分解。  
  
  
　　如图，通过一个个条件过滤器我们立体地实现了对信号的分离，如果你使用if else，说明你是将图中的条件1/2/3/4合并在一起，在同一个地方实现条件判断。  
　　需要深入了解过滤器的实现细节和微小区别，请参考文章：[**AOP vs Decorator**](http://www.jdon.com/AOPdesign/decorator.htm)

**OO设计的总结**

　　还有一种伪模式，虽然使用了状态等模式，但是在模式内部实质还是使用if else或switch进行状态切换或重要条件判断，那么无疑说明还需要进一步努力。更重要的是，不能以模式自居，而且出书示人。

　　真正掌握面向对象这些思想是一件困难的事情，目前有各种属于揪着自己头发向上拔的解说，都是误人子弟的，所以我觉得初学者读Thinking in Java（Java编程思想）是没有用，它试图从语言层次来讲OO编程思想，非常失败，作为语言参考书可以，但是作为Java体现的OO思想的学习资料，就错了。

　　OO编程思想是一种方法论，方法论如果没有应用比较，是无法体会这个方法论的特点的，禅是古代一个方法论，悟禅是靠挑水砍柴这些应用才能体会。

　　那么OO思想靠什么应用能够体会到了？是GoF设计模式，GoF设计模式是等于软件人员的挑水砍柴等基本活，所以，如果一个程序员连基本活都不会，他何以自居OO程序员？从事OO专业设计编程这个工作，如果不掌握设计模式基本功，就象一个做和尚的人不愿意挑水砍柴，他何以立足这个行业？早就被师傅赶下山。

　　最后总结：将if else用在小地方还可以，如简单的数值判断；但是如果按照你的传统习惯思维，在实现业务功能时也使用if else，那么说明你的思维可能需要重塑，你的编程经验越丰富，传统过程思维模式就容易根深蒂固，想靠自己改变很困难；建议接受专业头脑风暴培训。

　　用一句话总结：如果你做了不少系统，很久没有使用if else了，那么说明你可能真正进入OO设计的境地了。（这是本人自己发明的实战性的衡量考核标准）。

相关资料：

[**为什么需要状态？**](http://www.jdon.com/45632)

[**再谈IF ELSE**](http://www.jdon.com/39881)

[**面向对象与领域建模**](http://www.jdon.com/mda/modeling.html)

[**GoF设计模式**](http://www.jdon.com/designpatterns/index.htm)

[**软件最大的追求是什么？**](http://www.jdon.com/artichect/coupling.htm)

[**数据库时代的终结**](http://www.jdon.com/artichect/dbover.htm)

[**发表讨论**](http://www.jdon.com/24647)