

如何解决代码中 if...else 过多的问题

芋道源码 昨天

点击上方“[芋道源码](#)”，选择“设为星标”

做积极的人，而不是积极废人！



微信扫一扫
关注该公众号

源码精品专栏

- 原创 | Java 2020 超神之路，很肝~
- 中文详细注释的开源项目
- RPC 框架 Dubbo 源码解析
- 网络应用框架 Netty 源码解析
- 消息中间件 RocketMQ 源码解析
- 数据库中间件 Sharding-DB 和 MyCAT 源码解析
- 作业调度中间件 Elastic-Job 源码解析
- 分布式事务中间件 TCC-Transaction 源码解析
- Eureka 和 Hystrix 源码解析
- Java 开发源码

来源：艾瑞克·邵

cnblogs.com/eric-shao/p/10115577.html

- 前言
- 问题一：if...else 过多
 - 问题表现
 - 如何解决
 - 小结
- 问题二：if...else 嵌套过深
 - 问题表现
 - 如何解决
- 问题三：if...else 表达式过于复杂
 - 问题表现
 - 如何解决
- 总结



前言

if...else 是所有高级编程语言都有的必备功能。但现实中的代码往往存在着过多的 if...else。虽然 if...else 是必须的，但滥用 if...else 会对代码的可读性、可维护性造成很大伤害，进而危害到整个软件系统。现在软件开发领域出现了很多新技术、新概念，但 if...else 这种基本的程序形式并没有发生太大变化。使用好 if...else 不仅对于现在，而且对于将来，都是十分有意义的。今天我们就来看看如何“干掉”代码中的 if...else，还代码以清爽。



问题一：if...else 过多



问题表现

if...else 过多的代码可以抽象为下面这段代码。其中只列出5个逻辑分支，但实际工作中，能见到一个方法包含10个、20个甚至更多的逻辑分支的情况。另外，if...else 过多通常会伴随着另两个问题：逻辑表达式复杂和 if...else 嵌套过深。对于后两个问题，本文将在下面两节介绍。本节先来讨论 if...else 过多的情况。

```
●●●  
1if (condition1) {  
2
```

```
3 } elseif (condition2) {  
4  
5 } elseif (condition3) {  
6  
7 } elseif (condition4) {  
8  
9 } else {  
10  
11 }
```

通常，if...else 过多的方法，通常可读性和可扩展性都不好。从软件设计角度讲，代码中存在过多的 if...else 往往意味着这段代码违反了违反单一职责原则和开闭原则。因为在实际的项目中，需求往往是不断变化的，新需求也层出不穷。所以，软件系统的扩展性是非常重要的。而解决 if...else 过多问题的最大意义，往往就在于提高代码的可扩展性。



接下来我们来看如何解决 if...else 过多的问题。下面我列出了一些解决方法。

1. 表驱动
2. 职责链模式
3. 注解驱动
4. 事件驱动
5. 有限状态机
6. Optional
7. Assert
8. 多态

④ 方法一：表驱动

介绍

对于逻辑表达模式固定的 if...else 代码，可以通过某种映射关系，将逻辑表达式用表格的方式表示；再使用表格查找的方式，找到某个输入所对应的处理函数，使用这个处理函数进行运算。

适用场景

逻辑表达模式固定的 if...else

实现与示例

```
● ● ●  
1if (param.equals(value1)) {  
2    doAction1(someParams);  
3 } elseif (param.equals(value2)) {  
4    doAction2(someParams);  
5 } elseif (param.equals(value3)) {  
6    doAction3(someParams);  
7 }  
8// ...
```

可重构为

```
● ● ●  
1 Map<?, Function<?>> actionMappings = new HashMap<>(); // 这里泛型 ? 是为方便演示，实  
2
```

```
3// When init
4 actionMappings.put(value1, (someParams) -> { doAction1(someParams)});
5 actionMappings.put(value2, (someParams) -> { doAction2(someParams)});
6 actionMappings.put(value3, (someParams) -> { doAction3(someParams)});
7
8// 省略 null 判断
9 actionMappings.get(param).apply(someParams);
```

上面的示例使用了Java 8 的Lambda 和Functional Interface，这里不做讲解。

表的映射关系，可以采用集中的方式，也可以采用分散的方式，即每个处理类自行注册。也可以通过配置文件的方式表达。总之，形式有很多。

还有一些问题，其中的条件表达式并不像上例中的那样简单，但稍加变换，同样可以应用表驱动。下面借用《编程珠玑》中的一个税金计算的例子：

```
●●●

1if income <= 2200
2   tax = 0
3elseif income <= 2700
4   tax = 0.14 * (income - 2200)
5elseif income <= 3200
6   tax = 70 + 0.15 * (income - 2700)
7elseif income <= 3700
8   tax = 145 + 0.16 * (income - 3200)
9 .....
10else
11   tax = 53090 + 0.7 * (income - 102200)
```

对于上面的代码，其实只需将税金的计算公式提取出来，将每一档的标准提取到一个表格，在加上一个循环即可。具体重构之后的代码不给出，大家自己思考。

方法二：职责链模式

介绍

当 if..else 中的条件表达式灵活多变，无法将条件中的数据抽象为表格并用统一的方式进行判断时，这时应将对条件的判断权交给每个功能组件。并用链的形式将这些组件串联起来，形成完整的功能。

适用场景

条件表达式灵活多变，没有统一的形式。

实现与示例

职责链的模式在开源框架的 Filter、Interceptor 功能的实现中可以见到很多。下面看一下通用的使用模式：

重构前：

```
●●●

1public void handle(request) {
2  if (handlerA.canHandle(request)) {
3    handlerA.handleRequest(request);
4  } elseif (handlerB.canHandle(request)) {
5    handlerB.handleRequest(request);
6  } elseif (handlerC.canHandle(request)) {
7    handlerC.handleRequest(request);
8  }
```

```
9 }
```

重构后：

```
●●●  
1public void handle(Request request) {  
2    handlerA.handleRequest(request);  
3}  
4  
5public abstract class Handler {  
6    protected Handler next;  
7    public abstract void handleRequest(Request request);  
8    public void setNext(Handler next) { this.next = next; }  
9}  
10  
11public class HandlerA extends Handler {  
12    public void handleRequest(Request request) {  
13        if (canHandle(request)) doHandle(request);  
14        else if (next != null) next.handleRequest(request);  
15    }  
16}
```

当然，示例中的重构前的代码为了表达清楚，做了一些类和方法的抽取重构。现实中，更多的是平铺式的代码实现。

注：职责链的控制模式

职责链模式在具体实现过程中，会有一些不同的形式。从链的调用控制角度看，可分为外部控制和内部控制两种。

外部控制不灵活，但是减少了实现难度。职责链上某一环上的具体实现不用考虑对下一环的调用，因为外部统一控制了。但是一般的外部控制也不能实现嵌套调用。如果有嵌套调用，并且希望由外部控制职责链的调用，实现起来会稍微复杂。具体可以参考 Spring Web Interceptor 机制的实现方法。

内部控制就比较灵活，可以由具体的实现来决定是否需要调用链上的下一环。但如果调用控制模式是固定的，那这样的实现对于使用者来说是不便的。

设计模式在具体使用中会有很多变种，大家需要灵活掌握

方法三：注解驱动

介绍

通过 Java 注解（或其它语言的类似机制）定义执行某个方法的条件。在程序执行时，通过对比入参与注解中定义的条件是否匹配，再决定是否调用此方法。具体实现时，可以采用表驱动或职责链的方式实现。

适用场景

适合条件分支很多，对程序扩展性和易用性均有较高要求的场景。通常是某个系统中经常遇到新需求的核心功能。

实现与示例

很多框架中都能看到这种模式的使用，比如常见的 Spring MVC。因为这些框架很常用，demo 随处可见，所以这里不再上具体的演示代码了。

这个模式的重难点在于实现。现有的框架都具备于实现某一特定领域的功能，例如 MVC 框架

业务系统如采用此模式需自行实现相关核心功能。主要会涉及反射、职责链等技术。具体的实现这里就不做演示了。

④ 方法四：事件驱动

介绍

通过关联不同的事件类型和对应的处理机制，来实现复杂的逻辑，同时达到解耦的目的。

适用场景

从理论角度讲，事件驱动可以看做是表驱动的一种，但从实践角度讲，事件驱动和前面提到的表驱动有多处不同。具体来说：

1. 表驱动通常是一对一的关系；事件驱动通常是一对多；
2. 表驱动中，触发和执行通常是强依赖；事件驱动中，触发和执行是弱依赖

正是上述两者不同，导致了两者适用场景的不同。具体来说，事件驱动可用于如订单支付完成触发库存、物流、积分等功能。

实现与示例

实现方式上，单机的实践驱动可以使用 Guava、Spring 等框架实现。分布式的则一般通过各种消息队列方式实现。但是因为这里主要讨论的是消除 if...else，所以主要是面向单机问题域。因为涉及具体技术，所以此模式代码不做演示。

⑤ 方法五：有限状态机

介绍

有限状态机通常被称为状态机（无限状态机这个概念可以忽略）。先引用维基百科上的定义：

有限状态机（英语：finite-state machine，缩写：FSM），简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。

其实，状态机也可以看做是表驱动的一种，其实就是当前状态和事件两者组合与处理函数的一种对应关系。当然，处理成功之后还会有一个状态转移处理。

适用场景

虽然现在互联网后端服务都在强调无状态，但这并不意味着不能使用状态机这种设计。其实，在很多场景中，如协议栈、订单处理等功能中，状态机有这其天然的优势。因为这些场景中天然存在着状态和状态的流转。

实现与示例

实现状态机设计首先需要有相应的框架，这个框架需要实现至少一种状态机定义功能，以及对于的调用路由功能。状态机定义可以使用 DSL 或者注解的方式。原理不复杂，掌握了注解、反射等功能的同学应该可以很容易实现。

参考技术：

- Apache Mina State Machine Apache Mina 框架，虽然在 IO 框架领域不及 Netty，但它却提供了一个状态机的功能。<https://mina.apache.org/mina-project/userguide/ch14-state-machine/ch14-state-machine.html>。有自己实现状态机功能的同学可以参考其源码。

- Spring State Machine Spring 子项目众多，其中有个不显山不露水的状态机框架——Spring State Machine <https://projects.spring.io/spring-statemachine/>。可以通过 DSL 和注解两种方式定义。

上述框架只是起到一个参考的作用，如果涉及到具体项目，需要根据业务特点自行实现状态机的核心功能。

方法六：Optional

介绍

Java 代码中的一部分 if...else 是由非空检查导致的。因此，降低这部分带来的 if...else 也就能够降低整体的 if...else 的个数。

Java 从 8 开始引入了 Optional 类，用于表示可能为空的对象。这个类提供了很多方法，用于相关操作，可以用于消除 if...else。开源框架 Guava 和 Scala 语言也提供了类似的功能。

使用场景

有较多用于非空判断的 if...else。

实现与示例

传统写法：

```
● ● ●  
1 string str = "Hello World!";  
2 if (str != null) {  
3     System.out.println(str);  
4 } else {  
5     System.out.println("Null");  
6 }
```

使用 Optional 之后：

```
● ● ●  
1 Optional<String> strOptional = Optional.of("Hello World!");  
2 strOptional.ifPresentOrElse(System.out::println, () -> System.out.println("Null"));
```

Optional 还有很多方法，这里不一一介绍了。但请注意，不要使用 `get()` 和 `isPresent()` 方法，否则和传统的 if...else 无异。

扩展：Kotlin Null Safety

Kotlin 带有一个被称为 Null Safety 的特性：

```
● ● ●  
bob?.department?.head?.name
```

对于一个链式调用，在 Kotlin 语言中可以通过 `?.` 避免空指针异常。如果某一环为 null，那整个链式表达式的值便为 null。

方法七：Assert 模式

介绍

上一个方法适用于解决非空检查场景所导致的 if...else，类似的场景还有各种参数验证，比如还有字符串不为空等等。很多框架类库，例如 Spring、Apache Commons 都提供了工具里，用于实现这种通用的功能。这样大家就不必自行编写 if...else 了。

- Apache Commons Lang 中的 Validate 类：
<https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/Validate.html>
- Spring 的 Assert 类：<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html>

使用场景

通常用于各种参数校验

扩展：Bean Validation

类似上一个方法，介绍 Assert 模式顺便介绍一个有类似作用的技术——Bean Validation。Bean Validation 是 Java EE 规范中的一个。Bean Validation 通过在 Java Bean 上用注解的方式定义验证标准，然后通过框架统一进行验证。也可以起到了减少 if...else 的作用。

④ 方法八：多态

介绍

使用面向对象的多态，也可以起到消除 if...else 的作用。在代码重构这本书中，对此也有介绍：<https://refactoring.com/catalog/replaceConditionalWithPolymorphism.html>

使用场景

链接中给出的示例比较简单，无法体现适合使用多态消除 if...else 的具体场景。一般来说，当一个类中的多个方法都有类似于示例中的 if...else 判断，且条件相同，那就可以考虑使用多态的方式消除 if...else。

同时，使用多态也不是彻底消除 if...else。而是将 if...else 合并转移到了对象的创建阶段。在创建阶段的 if..，我们可以使用前面介绍的方法处理。



小结

上面这节介绍了 if...else 过多所带来的问题，以及相应的解决方法。除了本节介绍的方法，还有一些其它的方法。比如，在《重构与模式》一书中就介绍了“用 Strategy 替换条件逻辑”、“用 State 替换状态改变条件语句”和“用 Command 替换条件调度程序”这三个方法。其中的“Command 模式”，其思想同本文的“表驱动”方法大体一致。另两种方法，因为在《重构与模式》一书中已做详细讲解，这里就不再重复。

何时使用何种方法，取决于面对的问题的类型。上面介绍的一些适用场景，只是一些建议，更多的需要开发人员自己的思考。



问题二：if...else 嵌套过深



问题表现

if...else 嵌套过深会严重影响代码的可读性。同时 if...else 嵌套很深，也很复杂，导致代码可读性很差，自然也就难以维护。

```
1if (condition1) {  
2    action1();  
3if (condition2) {  
4    action2();  
5if (condition3) {  
6    action3();  
7if (condition4) {  
8    action4();  
9    }  
10   }  
11 }  
12 }
```

if...else 嵌套过深会严重地影响代码的可读性。当然，也会有上一节提到的两个问题。



上一节介绍的方法也可用用来解决本节的问题，所以对于上面的方法，此节不做重复介绍。这一节重点一些方法，这些方法并不会降低 if...else 的个数，但是会提高代码的可读性：

1. 抽取方法
2. 卫语句

④ 方法一：抽取方法

介绍

抽取方法是代码重构的一种手段。定义很容易理解，就是将一段代码抽取出来，放入另一个单独定义的方法。借用 <https://refactoring.com/catalog/extractMethod.html> 中的定义：

适用场景

if...else 嵌套严重的代码，通常可读性很差。故在进行大型重构前，需先进行小幅调整，提高其代码可读性。抽取方法便是最常用的一种调整手段。

实现与示例

重构前：

```
1public void add(Object element) {  
2if (!readOnly) {  
3int newSize = size + 1;  
4if (newSize > elements.length) {  
5    Object[] newElements = new Object[elements.length + 10];  
6for (int i = 0; i < size; i++) {  
7    newElements[i] = elements[i];  
8    }  
9    elements = newElements  
10   }  
11   elements[size++] = element;  
12 }
```

重构后：

```
●●●
1public void add(Object element) {
2if (readOnly) {
3return;
4}
5
6if (overCapacity()) {
7    grow();
8}
9
10 addElement(element);
11}
```

方法二：卫语句

介绍

在代码重构中，有一个方法被称为“使用卫语句替代嵌套条件语句”

<https://refactoring.com/catalog/replaceNestedConditionalWithGuardClauses.html>。直接看代码：

```
●●●
1double getPayAmount() {
2double result;
3if (_isDead) result = deadAmount();
4else {
5if (_isSeparated) result = separatedAmount();
6else {
7if (_isRetired) result = retiredAmount();
8else result = normalPayAmount();
9}
10}
11return result;
12}
```

重构之后

```
●●●
1double getPayAmount() {
2if (_isDead) return deadAmount();
3if (_isSeparated) return separatedAmount();
4if (_isRetired) return retiredAmount();
5return normalPayAmount();
6}
```

使用场景

当看到一个方法中，某一层代码块都被一个 if...else 完整控制时，通常可以采用卫语句。



if...else 所导致的第三个问题来自过于复杂的条件表达式。下面给个简单的例子，当 condition 1、2、3、4 分别为 true、false，请大家排列组合一下下面表达式的结果。

```
●●●
1 if ((condition1 && condition2) || ((condition2 || condition3) && condition4)) {
2
3 }
```

我想没人愿意干上面的事情。关键是，这一大坨表达式的含义是什么？关键便在于，当不知道表达式的含义时，没人愿意推断它的结果。

所以，表达式复杂，并不一定是错。但是表达式难以让人理解就不好了。

如何解决

对于 if...else 表达式复杂的问题，主要用代码重构中的抽取方法、移动方法等手段解决。因为这些方法在《代码重构》一书中都有介绍，所以这里不再重复。

总结

本文一个介绍了10种（算上扩展有12种）用于消除、简化 if...else 的方法。还有一些方法，如通过策略模式、状态模式等手段消除 if...else 在《重构与模式》一书中也有介绍。

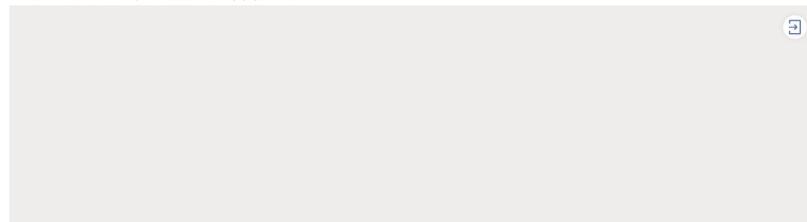
正如前言所说，if...else 是代码中的重要组成部分，但是过度、不必要地使用 if...else，会对代码的可读性、可扩展性造成负面影响，进而影响到整个软件系统。

"干掉"if...else 的能力高低反映的是程序员对软件重构、设计模式、面向对象设计、架构模式、数据结构等多方面技术的综合运用能力，反映的是程序员的内功。要合理使用 if...else，不能没有设计，也不能过度设计。这些对技术的综合、合理地运用都需要程序员在工作中不断的摸索总结。

欢迎加入我的知识星球，一起探讨架构，交流源码。加入方式，[长按下方二维码噢](#)：



已在知识星球更新源码解析如下：



《Dubbo 源码解析》	《Spring Cloud 源码解析》
01. 调试环境搭建 02. 项目结构一览 03. 配置 Configuration 04. 核心流程一览 05. 拓展机制 SPI 06. 线程池 ThreadPool 07. 服务暴露 Export 08. 服务引用 Refer 09. 注册中心 Registry 10. 动态编译 Compile 11. 动态代理 Proxy 12. 服务调用 Invoke 13. 调用特性 14. 过滤器 Filter 15. NIO 服务器 16. P2P 服务器 17. HTTP 服务器 18. 序列化 Serialization 19. 集群容错 Cluster 20. 优雅停机 Shutdown 21. 日志适配 Logging 22. 状态检查 Status 23. 监控中心 Monitor 24. 管理中心 Admin 25. 远维命令 QOS 26. 链路追踪 Tracing 27. Spring Boot 集成 28. Spring Cloud 集成 ... 一共 73+ 篇	01. 网关 Spring Cloud Gateway 25 篇 02. 注册中心 Eureka 23 篇 03. 熔断器 Hystrix 9 篇 04. 配置中心 Apollo 32 篇 05. 链路追踪 SkyWalking 38 篇 06. 调度中心 Elastic Job 24 篇
	《Netty 源码解析》
	01. 调试环境搭建 02. NIO 基础 03. Netty 简介 04. 启动 Bootstrap 05. 事件轮询 EventLoop 06. 通道管道 ChannelPipeline 07. 通道 Channel 08. 字节缓冲区 ByteBuf 09. 通道处理器 ChannelHandler 10. 编解码 Codec 11. 工具类 Util ... 一共 61+ 篇
	《MyBatis 源码解析》
	01. 调试环境搭建 02. 项目结构一览 03. MyBatis 初始化 04. SQL 初始化 05. SQL 执行 06. 插件体系 07. Spring 集成 ... 一共 34+ 篇

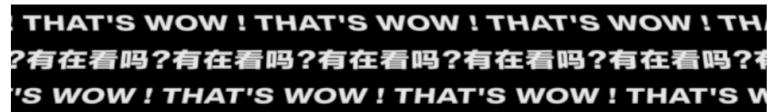
07. AOP 源码导读 08. Transaction 源码导读 ... 一共 46+ 篇	07. HandlerExceptionResolver 组件 08. RequestToViewNameTranslator 组件 09. LocaleResolver 组件 10. ThemeResolver 组件 11. ViewResolver 组件 12. MultipartResolver 组件 13. FlashMapManager 组件 ... 一共 24+ 篇
	《数据库实体设计》



最近更新《芋道 SpringBoot 2.X 入门》系列，已经 20 余篇，覆盖了 MyBatis、Redis、MongoDB、ES、分库分表、读写分离、SpringMVC、Webflux、权限、WebSocket、Dubbo、RabbitMQ、RocketMQ、Kafka、性能测试等等内容。

提供近 3W 行代码的 SpringBoot 示例，以及超 4W 行代码的电商微服务项目。

获取方式：点“**在看**”，关注公众号并回复**666**领取，更多内容陆续奉上。



[阅读原文](#)