代码多版改造，应用责任链设计模式

架构精进之路 2023-02-08 08:31 发表于北京

hello，大家好，我是张张，「架构精进之路」公号作者。

1、背景

责任链模式（又称职责链模式，The Chain of Responsibility Pattern），作为开发设计中常用的代码设计模式之一，属于行为模式中的一种，历来被我们开发所熟悉。

近期，团队内一位成员在开发时使用了责任链模式，代码堆地非常多，bug 也多，显然没有达到我们预期的效果。

实际上，针对这项功能，我认为模版方法更合适！为此，隔壁团队也拿出我们的案例，进行了集体 code review。

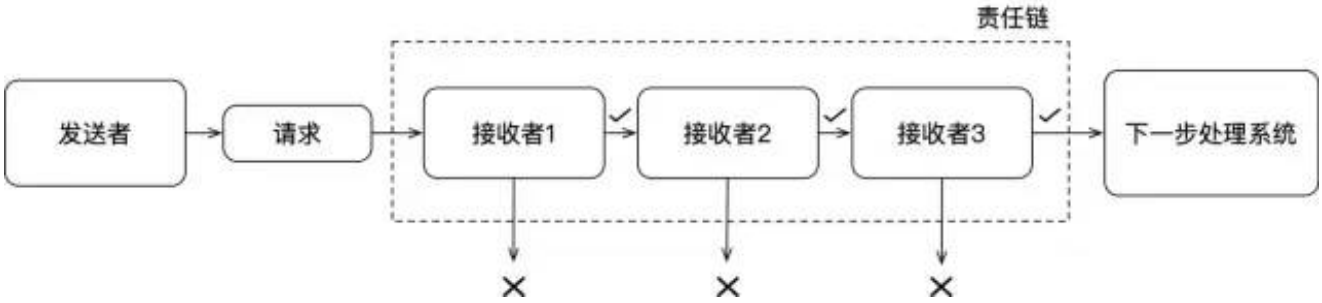
学好设计模式，且不要为了练习，强行使用！本篇旨在重新梳理一下责任链设计模式，并诉诸文字，温故而知新。

2、什么是责任链模式

责任链模式是一种理解上比较简单的行为设计模式，它允许开发者通过处理者链进行顺序发送，每个链节点在收到请求后，具备两种能力：

1. 对其进行处理（消费）
2. 将其传递给链上的下个处理者

当你想要让一个以上的对象有机会能处理某个请求时，就可以使用责任链模式。通过责任链模式，为某个请求创建一个对象链，每个对象链依序检查此请求，并对其进行处理，或者将它传给链中的下一个对象。



责任链到底解决了什么问题?

1. 前置检查，减少不必要的后置代码逻辑
2. 发送者（sender）和接收者（receiver(s)）的逻辑解耦，提高代码灵活性，这一点是最重要的
3. 通过链路顺序传递请求，也使每一个链节点职责明确，符合单一职责原则
4. 通过改变链内的成员或调动它们的次序，允许你动态地新增或删除，也提高了代码的灵活性

3、应用案例

假设现在有一个闯关游戏，进入下一关的条件是上一关的分数要高于 xx：

- 游戏一共 3 个关卡
- 进入第二关需要第一关的游戏得分大于等于 80
- 进入第三关需要第二关的游戏得分大于等于 90

3.1 最初方案

那么代码可以这样写：

```
//第一关
public class FirstPassHandler {
    public int handler(){
        System.out.println("第一关-->FirstPassHandler");
        return 80;
    }
}

//第二关
public class SecondPassHandler {
    public int handler(){
        System.out.println("第二关-->SecondPassHandler");
        return 90;
    }
}

```

```java
//第三关
public class ThirdPassHandler {
    public int handler(){
        System.out.println("第三关-->ThirdPassHandler，这是最后一关啦");
        return 95;
    }
}

//客户端
public class HandlerClient {
    public static void main(String[] args) {
        FirstPassHandler firstPassHandler = new FirstPassHandler();//第一
        SecondPassHandler secondPassHandler = new SecondPassHandler();//
        ThirdPassHandler thirdPassHandler = new ThirdPassHandler();//第三

        int firstScore = firstPassHandler.handler();
        //第一关的分数大于等于80则进入第二关
        if(firstScore >= 80){
            int secondScore = secondPassHandler.handler();
            //第二关的分数大于等于90则进入第二关
            if(secondScore >= 90){
                thirdPassHandler.handler();
            }
        }
    }
}
```

那么如果这个游戏有 100 关，我们的代码很可能就会写成这个样子：

```java
if(第1关通过){
  // 第2关 游戏
  if(第2关通过){
    // 第3关 游戏
    if(第3关通过){
      // 第4关 游戏
      if(第4关通过){
        // 第5关 游戏
        if(第5关通过){
          // 第6关 游戏
```

```
            if(第6关通过){
              //...
            }
          }
        }
      }
    }
```

这种代码不仅冗余，并且当我们要将某两关进行调整时会对代码非常大的改动，这种操作的风险是很高的，因此，该写法非常糟糕。

3.2 初步改造

如何解决这个问题，我们可以通过链表将每一关连接起来，形成责任链的方式，第一关通过后是第二关，第二关通过后是第三关，这样客户端就不需要进行多重 if 的判断了。

```
public class FirstPassHandler {
    /**
     * 第一关的下一关是 第二关
     */
    private SecondPassHandler secondPassHandler;

    public void setSecondPassHandler(SecondPassHandler secondPassHandler)
      this.secondPassHandler = secondPassHandler;
    }

    //本关卡游戏得分
    private int play(){
        return 80;
    }

    public int handler(){
        System.out.println("第一关-->FirstPassHandler");
        if(play() >= 80){
            //分数>=80 并且存在下一关才进入下一关
            if(this.secondPassHandler != null){
                return this.secondPassHandler.handler();
            }
        }
        return 80;
```

```
    }
}

public class SecondPassHandler {

    /**
     * 第二关的下一关是 第三关
     */
    private ThirdPassHandler thirdPassHandler;

    public void setThirdPassHandler(ThirdPassHandler thirdPassHandler) {
        this.thirdPassHandler = thirdPassHandler;
    }

    //本关卡游戏得分
    private int play(){
        return 90;
    }

    public int handler(){
        System.out.println("第二关-->SecondPassHandler");

        if(play() >= 90){
            //分数>=90 并且存在下一关才进入下一关
            if(this.thirdPassHandler != null){
                return this.thirdPassHandler.handler();
            }
        }
        return 90;
    }
}

public class ThirdPassHandler {

    //本关卡游戏得分
    private int play(){
        return 95;
    }
    /**
     * 这是最后一关，因此没有下一关
     */
```

```
66      public int handler(){
67          System.out.println("第三关-->ThirdPassHandler，这是最后一关啦");
68          return play();
69      }
70  }
71
72  public class HandlerClient {
73
74      public static void main(String[] args) {
75          FirstPassHandler firstPassHandler = new FirstPassHandler();//第一
76          SecondPassHandler secondPassHandler = new SecondPassHandler();//
77          ThirdPassHandler thirdPassHandler = new ThirdPassHandler();//第三
78
79          firstPassHandler.setSecondPassHandler(secondPassHandler);//第一关
80          secondPassHandler.setThirdPassHandler(thirdPassHandler);//第二关的
81
82          //说明：因为第三关是最后一关，因此没有下一关
83          //开始调用第一关 每一个关卡是否进入下一关卡 在每个关卡中判断
84          firstPassHandler.handler();
85      }
86
87  }
```

初步改造方案的缺点：

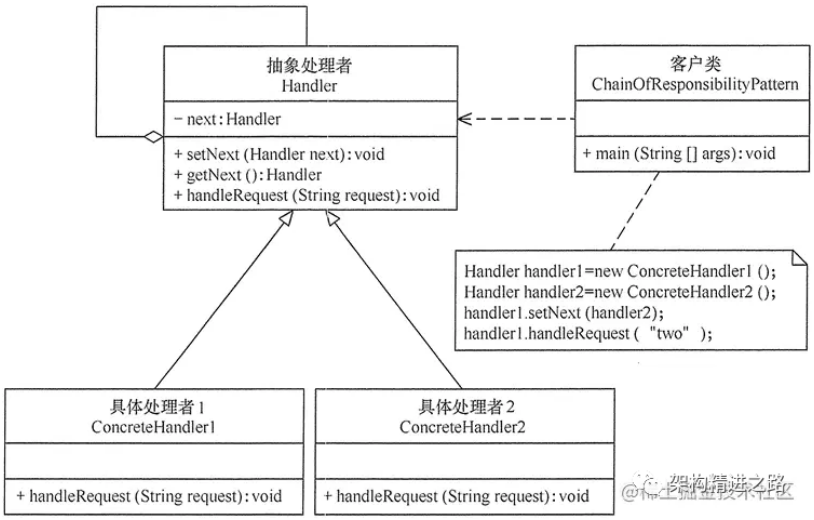
- 每个关卡中都有下一关的成员变量并且是不一样的，形成链很不方便；
- 代码的扩展性非常不好。

3.3 责任链优化

既然每个关卡中都有下一关的成员变量并且是不一样的，那么我们可以在关卡上抽象出一个父类或者接口，然后每个具体的关卡去继承或者实现。

有了思路，我们先来简单介绍一下责任链设计模式的基本组成：

- **抽象处理者（Handler）角色**：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
- **具体处理者（Concrete Handler）角色**：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
- **客户类（Client）角色**：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。




```java
public abstract class AbstractHandler {

    /**
     * 下一关用当前抽象类来接收
     */
    protected AbstractHandler next;

    public void setNext(AbstractHandler next) {
        this.next = next;
    }

    public abstract int handler();
}

public class FirstPassHandler extends AbstractHandler{

    private int play(){
        return 80;
    }

    @Override
    public int handler(){
        System.out.println("第一关-->FirstPassHandler");
        int score = play();
        if(score >= 80){
            //分数>=80 并且存在下一关才进入下一关
            if(this.next != null){
```

```
                return this.next.handler();
            }
        }
        return score;
    }
}

public class SecondPassHandler extends AbstractHandler{

    private int play(){
        return 90;
    }

    public int handler(){
        System.out.println("第二关-->SecondPassHandler");

        int score = play();
        if(score >= 90){
            //分数>=90 并且存在下一关才进入下一关
            if(this.next != null){
                return this.next.handler();
            }
        }

        return score;
    }

}

public class ThirdPassHandler extends AbstractHandler{

    private int play(){
        return 95;
    }

    public int handler(){
        System.out.println("第三关-->ThirdPassHandler");
        int score = play();
        if(score >= 95){
            //分数>=95 并且存在下一关才进入下一关
            if(this.next != null){
```

```
                return this.next.handler();
            }
        }
        return score;
    }

}

public class HandlerClient {

    public static void main(String[] args) {
        FirstPassHandler firstPassHandler = new FirstPassHandler();//第一
        SecondPassHandler secondPassHandler = new SecondPassHandler();//
        ThirdPassHandler thirdPassHandler = new ThirdPassHandler();//第三

        // 和上面没有更改的客户端代码相比，只有这里的set方法发生变化，其他都是一样的
        firstPassHandler.setNext(secondPassHandler);//第一关的下一关是第二关
        secondPassHandler.setNext(thirdPassHandler);//第二关的下一关是第三关
        //因为第三关是最后一关，因此没有下一关

        //从第一个关卡开始
        firstPassHandler.handler();
    }

}
```

3.4 责任链工厂优化

对于上面的请求链，我们也可以把这个关系维护到配置文件中或者一个枚举中。我将使用枚举来教会大家怎么动态的配置请求链并且将每个请求者形成一条调用链。

```
public enum GatewayEnum {
    // handlerId, 拦截者名称，全限定类名，preHandlerId，nextHandlerId
    API_HANDLER(new GatewayEntity(1, "api接口限流", "cn.dgut.design.chain_
    BLACKLIST_HANDLER(new GatewayEntity(2, "黑名单拦截", "cn.dgut.design.c
    SESSION_HANDLER(new GatewayEntity(3, "用户会话拦截", "cn.dgut.design.cl

    GatewayEntity gatewayEntity;

```

```
    public GatewayEntity getGatewayEntity() {
        return gatewayEntity;
    }

    GatewayEnum(GatewayEntity gatewayEntity) {
        this.gatewayEntity = gatewayEntity;
    }
}

public class GatewayEntity {

    private String name;
    private String conference;
    private Integer handlerId;
    private Integer preHandlerId;
    private Integer nextHandlerId;
}


public interface GatewayDao {

    /**
     * 根据 handlerId 获取配置项
     * @param handlerId
     * @return
     */
    GatewayEntity getGatewayEntity(Integer handlerId);

    /**
     * 获取第一个处理者
     * @return
     */
    GatewayEntity getFirstGatewayEntity();
}

public class GatewayImpl implements GatewayDao {

    /**
     * 初始化，将枚举中配置的handler初始化到map中，方便获取
     */
    private static Map<Integer, GatewayEntity> gatewayEntityMap = new Ha
```

```
50
51      static {
52          GatewayEnum[] values = GatewayEnum.values();
53          for (GatewayEnum value : values) {
54              GatewayEntity gatewayEntity = value.getGatewayEntity();
55              gatewayEntityMap.put(gatewayEntity.getHandlerId(), gatewayEn
56          }
57      }
58
59      @Override
60      public GatewayEntity getGatewayEntity(Integer handlerId) {
61          return gatewayEntityMap.get(handlerId);
62      }
63
64      @Override
65      public GatewayEntity getFirstGatewayEntity() {
66          for (Map.Entry<Integer, GatewayEntity> entry : gatewayEntityMap.
67              GatewayEntity value = entry.getValue();
68              //  没有上一个handler的就是第一个
69              if (value.getPreHandlerId() == null) {
70                  return value;
71              }
72          }
73          return null;
74      }
75  }
76
77  public class GatewayHandlerEnumFactory {
78
79      private static GatewayDao gatewayDao = new GatewayImpl();
80
81      // 提供静态方法，获取第一个handler
82      public static GatewayHandler getFirstGatewayHandler() {
83
84          GatewayEntity firstGatewayEntity = gatewayDao.getFirstGatewayEnt
85          GatewayHandler firstGatewayHandler = newGatewayHandler(firstGate
86          if (firstGatewayHandler == null) {
87              return null;
88          }
89
90          GatewayEntity tempGatewayEntity = firstGatewayEntity;
```

```
        Integer nextHandlerId = null;
        GatewayHandler tempGatewayHandler = firstGatewayHandler;
        // 迭代遍历所有handler，以及将它们链接起来
        while ((nextHandlerId = tempGatewayEntity.getNextHandlerId()) !=
            GatewayEntity gatewayEntity = gatewayDao.getGatewayEntity(ne
            GatewayHandler gatewayHandler = newGatewayHandler(gatewayEnt
            tempGatewayHandler.setNext(gatewayHandler);
            tempGatewayHandler = gatewayHandler;
            tempGatewayEntity = gatewayEntity;
        }
        // 返回第一个handler
        return firstGatewayHandler;
    }

    /**
     * 反射实体化具体的处理者
     * @param firstGatewayEntity
     * @return
     */
    private static GatewayHandler newGatewayHandler(GatewayEntity firstG
        // 获取全限定类名
        String className = firstGatewayEntity.getConference();
        try {
            // 根据全限定类名，加载并初始化该类，即会初始化该类的静态段
            Class<?> clazz = Class.forName(className);
            return (GatewayHandler) clazz.newInstance();
        } catch (ClassNotFoundException | IllegalAccessException | Insta
            e.printStackTrace();
        }
        return null;
    }

}

public class GetewayClient {
    public static void main(String[] args) {
        GetewayHandler firstGetewayHandler = GetewayHandlerEnumFactory.g
        firstGetewayHandler.service();
    }

```

```
    }
```

结语

最后借用《Head First Design Patterns》一书中对设计模式如何使用的表述，做一个收尾，深以为然。

1. 为实际需要的扩展使用模式，不要只是为了假想的需要而使用模式
2. 简单才是王道，如果不用模式就能设计出更简单的方案，那就去干吧
3. 模式是工具而不是规则，需要被适当地调整以符合实际的需求

参考

- 《Dive-into Design Patter》 Alexander Shvets
- 《Head First Design Patterns》 Elisabeth Freeman and Kathy Sierra

往期热文推荐：

- 面对一堆烂代码，重构，还是重新开发？
- 电商并发减库存设计，如何做到不超卖
- 身处职场，一定要落落大方，不要畏手畏脚

收录于合集 #系统架构应用汇总 40

〈上一篇
太强了，全面解析缓存应用经典问题

下一篇〉
PHP 中数组是如何灵活支持多数据类型的?

喜欢此内容的人还喜欢

04 | 重学前端之HTML-约束验证-上
重学前端MDN

PHP 任意文件读取漏洞
安全街

详解 23 种设计模式（多图 + 代码）
BUG弄潮儿