原创 职责链模式（Chain of Responsibility）的Java实现

2010-07-01 13:48:08

标签：Chain Responsibility 休闲 职责链 职场

职责链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

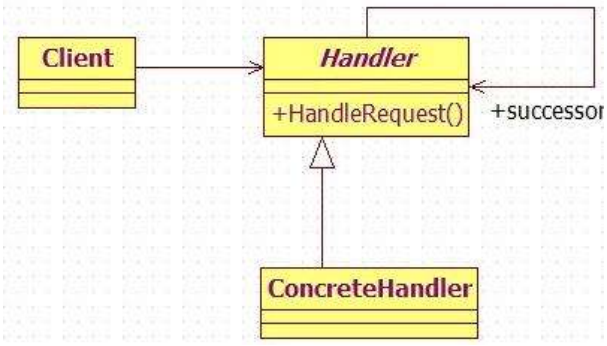
适用场景：

1、有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定；

2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；

3、处理一个请求的对象集合应被动态指定。

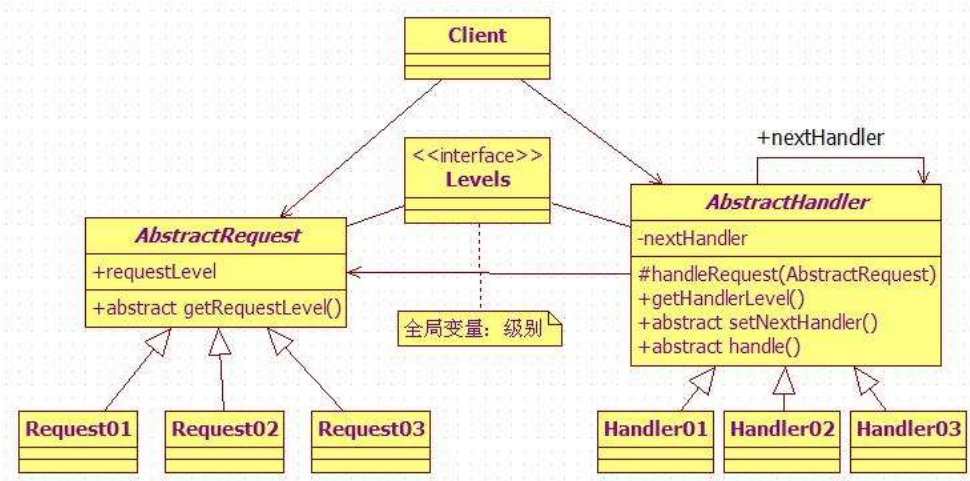
通用类图：



在大学里面当班干部，时常要向上级申请各方面的东西。譬如申请全班外出秋游，普通同学将申请表交给班长，班长签字之后交给辅导员，辅导员批准之后上交到主任办公室…就是这样，一个请求（这里是一份申请表）有时候需要经过好几个级别的处理者（这里是辅导员、主任）的审查才能够最终被确定可行与否。

在这里表现出来的是一个职责链，即不同的处理者对同一个请求可能担负着不同的处理方式、权限，但是我们希望这个请求必须到达最终拍板的处理者（否则秋游就没戏了）。这种关系就很适合使用职责链模式了。

类图结构如下：



代码实现如下：

```
// 全局变量，接口类型
/**
 * 使用Java中的interface定义全局变量，可根据具体需要在
 * 具体的包中使用静态导入相关的全局变量，语法如下：
 *  import static package01.package02.*;
 */
interface Levels {
    public static final int LEVEL_01 = 1;
    public static final int LEVEL_02 = 2;
    public static final int LEVEL_03 = 3;
}
```

```
// 抽象请求类
abstract class AbstractRequest {
    private String content = null;

    public AbstractRequest(String content) {
        this.content = content;
    }

    public String getContent() {
        return this.content;
    }

    // 获得请求的级别
    public abstract int getRequestLevel();
}
```

```
// 具体请求类01
class Request01 extends AbstractRequest {
    public Request01(String content) {
        super(content);
    }

    @Override
    public int getRequestLevel() {
        return Levels.LEVEL_01;
    }
}

// 具体请求类02
class Request02 extends AbstractRequest {
    public Request02(String content) {
        super(content);
    }

    @Override
    public int getRequestLevel() {
        return Levels.LEVEL_02;
    }
}

// 具体请求类03
class Request03 extends AbstractRequest {
    public Request03(String content) {
        super(content);
    }

    @Override
    public int getRequestLevel() {
        return Levels.LEVEL_03;
    }
}
```

```java
// 抽象处理者类，
abstract class AbstractHandler {
    // 责任链的下一个节点，即处理者
    private AbstractHandler nextHandler = null;

    // 捕获具体请求并进行处理，或是将请求传递到责任链的下一级别
    public final void handleRequest(AbstractRequest request) {

        // 若该请求与当前处理者的级别层次相对应，则由自己进行处理
        if (this.getHandlerLevel() == request.getRequestLevel()) {
            this.handle(request);
        } else {
            // 当前处理者不能胜任，则传递至职责链的下一节点
            if (this.nextHandler != null) {
                System.out.println("当前 处理者-0" + this.getHandlerLevel()
                        + " 不足以处理 请求-0" + request.getRequestLevel());

                // 这里使用了递归调用
                this.nextHandler.handleRequest(request);
            } else {
                System.out.println("职责链上的所有处理者都不能胜任该请求...");
            }
        }
    }

    // 设置责任链中的下一个处理者
    public void setNextHandler(AbstractHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    // 获取当前处理者的级别
    protected abstract int getHandlerLevel();

    // 定义链中每个处理者具体的处理方式
    protected abstract void handle(AbstractRequest request);
}
```

```java
// 具体处理者-01
class Handler01 extends AbstractHandler {
    @Override
    protected int getHandlerLevel() {
        return Levels.LEVEL_01;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("处理者-01 处理 " + request.getContent() + "\n");
    }
}

// 具体处理者-02
class Handler02 extends AbstractHandler {
    @Override
    protected int getHandlerLevel() {
        return Levels.LEVEL_02;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("处理者-02 处理 " + request.getContent()+ "\n");
    }
}

// 具体处理者-03
class Handler03 extends AbstractHandler {
    @Override
    protected int getHandlerLevel() {
        return Levels.LEVEL_03;
    }

    @Override
    protected void handle(AbstractRequest request) {
        System.out.println("处理者-03 处理 " + request.getContent()+ "\n");
    }
}
```

```java
// 测试类
public class Client {
    public static void main(String[] args) {
        // 创建指责链的所有节点
        AbstractHandler handler01 = new Handler01();
        AbstractHandler handler02 = new Handler02();
        AbstractHandler handler03 = new Handler03();

        // 进行链的组装，即头尾相连，一层套一层
        handler01.setNextHandler(handler02);
```

```
        handler02.setNextHandler(handler03);

        // 创建请求并提交到指责链中进行处理
        AbstractRequest request01 = new Request01("请求-01");
        AbstractRequest request02 = new Request02("请求-02");
        AbstractRequest request03 = new Request03("请求-03");

        // 每次提交都是从链头开始遍历
        handler01.handleRequest(request01);
        handler01.handleRequest(request02);
        handler01.handleRequest(request03);
    }
}
```

测试结果：

```
处理者-01 处理 请求-01

当前 处理者-01 不足以处理 请求-02
处理者-02 处理 请求-02

当前 处理者-01 不足以处理 请求-03
当前 处理者-02 不足以处理 请求-03
处理者-03 处理 请求-03
```

在上面抽象处理者 AbstractHandler 类的 handleRequest() 方法中，被 protected 修饰，并且该方法中调用了两个必须被子类覆盖实现的抽象方法，这里是使用了模板方法模式（Template Mehtod）。其实在这里，抽象父类的 handleRequest() 具备了请求传递的功能，即对某些请求不能处理时，马上提交到下一结点（处理者）中，而每个具体的处理者仅仅完成了具体的处理逻辑，其他的都不用理。

记得第一次看到职责链模式的时候，我很惊讶于它能够把我们平时在代码中的 if..else.. 的语句块变成这样灵活、适应变化。例如：如果现在辅导员请长假了，但我们的秋游还是要争取申请成功呀，那么我们在 Client 类中可以不要创建 handler02，即不要将该处理者组装到职责链中。这样子处理比 if..else..好多了。或者说，突然来了个爱管闲事的领导，那么我照样可以将其组装到职责链中。

关于上面使用场景中提到的3个点：

1、处理者在运行时动态确定其实是我们在 Client 中组装的链所引起的，因为具体的职责逻辑就在链中一一对应起来；

2、因为不确定请求的具体处理者是谁，所以我们把所有可能的处理者组装成一条链，在遍历的过程中就相当于向每个处理者都提交了这个请求，等待其审查。并且在审查过程中，即使不是最终处理者，也可以进行一些请求的“包装”操作（这种功能类似于装饰者模式），例如上面例子中的签名批准；

3、处理者集合的动态指定跟上面的第1、2点类似，即在 Client 类中创建了所有可能的处理者。

不足之处：

1、对于每一个请求都需要遍历职责链，性能是个问题；

2、抽象处理者 AbstractHandler 类中的 handleRequest() 方法中使用了递归，栈空间的大小也是个问题。

个人看法：

职责链模式对于请求的处理是不知道最终处理者是谁，所以是运行动态寻找并指定；而命令模式中对于命令的处理时在创建命令是已经显式或隐式绑定了接收者。

相关文章：

命令模式（Command）的两种不同实现（http://haolloyin.blog.51cto.com/1177454/339076）

（Template Method）模板方法模式的Java实现（http://haolloyin.blog.51cto.com/1177454/333063）

装饰模式（Decorator）与动态代理的强强联合（http://haolloyin.blog.51cto.com/1177454/338671）

本文出自 “蚂蚁” 博客，请务必保留此出处http://haolloyin.blog.51cto.com/1177454/342166

附件下载：
源代码

相关文章

- Java与模式：适配器模式
- 设计模式之java接口和java抽象类
- 单子模式
- Swing中的并发-使用SwingWorker线程模式
- 传智播客学习总结——Java Web——tomcat开..
- 追MM与Java的23种设计模式
- java代理模式

文章评论

[1楼] [匿名]51CTO游客 回复

2010-08-02 17:42:01

博主这儿有很多好东西。

[2楼]楼主 haolloyin 回复

2010-08-02 22:14:16

回复 :[1楼]
呵呵...谢谢！刚刚起步学习呢，有空欢迎多来坐坐！

[3楼] huayurei 回复

2011-02-24 10:40:33

性能问题啥的在应用程序开发方面其实问题不是很大，毕竟大部分情况下可以升级硬件。只是觉得职责链需要在写代码的时候自己维护职责链，感觉比较麻烦比如：
handler01.setNextHandler(handler02);
handler02.setNextHandler(handler03);
如果调用者这里的职责链顺序写错了咋办，毕竟这个人工维护还是很麻烦的，所以感觉还有优化的空间。
刚学习设计模式，小小的疑问。

[4楼] 蓝枫居士 回复

2014-05-17 20:30:48

楼主的文章还是不错的
我这有两点建议 楼主可以考虑一下
第一职责链模式如果与数据结构中的知识点相对应的话 那就是链表了 但是 职责链毕竟不是链表 区别在什么地方？在于不管是有头节点的链表还是没有头节点的链表 对它的访问都是从第一个节点开始的 但职责链不是这样,我可以从中间开始访问 例如 职责连中有班长 辅导员 主任 院长 如果班长临时出去了 学生也可以把假条直接递给辅导员呀 这个是符合客观事实的 当然这样一来 职责连的每个阶段都还得加上前驱节点 各种利弊 楼主可以考虑一下

[5楼] 蓝枫居士 回复

2014-05-17 20:34:41

其二
我觉得还可以再对最最单纯的职责链模式做以改进 在原始的模式中 每个处理者的下一位必定是另一个"级别"的处理者 我认为这一点也可以改动一下,可以递交给同级的处理者 当然这两个同级的处理者权限会有一点不一样 总之 一句话 处理对象应该可以在同级处理者之间传递

[6楼] 蓝枫居士 回复

2014-05-17 20:42:28

另外 调用后继者的处理方法 应该不算是递归吧 楼主考虑考虑

发表评论 51CTO学院2周年庆，分享故事赢大礼包

昵 称： 登录 快速注册

验证码： 请点击后输入验证码 博客过2级，无需填写验证码

内 容：