大家好，我是楼仔！

上半年写了8篇设计模式相关的文章，之所以又开始写这个系列，是因为这周看代码时，发现代码中标注的是"XX责任链"，当时怀疑是不是用的责任链模式，但是看了一会又不像。然后之前技术方案评审时，有一位同学也提到过这个设计模式，当时就想再找出来瞬瞬，只是太懒了。

目前也工作了很长时间，实际项目中其实还没有真正去使用过责任链模式，也没有看别人用过，所以感觉这个模式比较小众。

这篇文章没链技术含量，纯粹写了个demo，记录一下这个模式的用法。

定义

我们看一下责任链模式的定义：

> 责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程。所以责任链将请求的发送者和请求的处理者解耦了。

责任链模式

先创建一个抽象类，里面包括责任链等级，以及责任链的下一个元素：

```
public abstract class penguin {
    protected int level; // 责任链等级
    protected penguin nextPenguin; // 责任链中的下一个元素

    public void process(int level, String message) {
        if (level >= this.level) {
            // 如果输入的级别大于对象本身的级别，输出数据
            print(message);
        }
        if (this.nextPenguin != null) {
            // 如果存在下一个元素，就继续往下执行
            nextPenguin.process(level,message);
        }
    }
    public abstract void print(String message);
}
```

下面是实现的子类，为了简单期间，我直接标明每个子类的等级：

```
// 小企鹅子类
public class littlePenguin extends penguin {
    public littlePenguin() {
        this.level = 1;
    }
    @Override
    public void print(String message) {
        System.out.println("Little Penguin:" + message);
    }
}
// 中企鹅子类
public class middlePenguin extends penguin {
    public middlePenguin() {
        this.level = 2;
    }
    @Override
    public void print(String message) {
        System.out.println("Middle Penguin:" + message);
    }
}
// 大企鹅子类
public class bigPenguin extends penguin {
    public bigPenguin() {
        this.level = 3;
    }
    @Override
    public void print(String message) {
        System.out.println("Big Penguin:" + message);
    }
}
```

最后就是测试用例，我们先将三个子类串起来，类似于串链路声，层级依次递进：

```
public class test {
    public static void main(String[] args) {
        littlePenguin penguin1 = new littlePenguin();
        middlePenguin penguin2 = new middlePenguin();
        bigPenguin penguin3 = new bigPenguin();

        penguin3.nextPenguin = penguin2;
        penguin2.nextPenguin = penguin1;

        penguin3.process(3, "print big/middle/small");
        penguin3.process(2, "print middle/small");
        penguin3.process(1, "print small");
    }
}
```

最后的输出结果：

```
Big Penguin:print big/middle/small
Middle Penguin:print big/middle/small
Little Penguin:print big/middle/small
Middle Penguin:print middle/small
Little Penguin:print middle/small
Little Penguin:print small
```

我们可以看到，当level=2时，只输出了Little Penguin和Middle Penguin，因为只有<=2的级别才能输出。当level=2时，就只输出了Little Penguin，满足责任链的场景。

优点 VS 缺点

优点：

- 降低了对象之间的耦合度，该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
- 增强了系统的可扩展性，可以根据需要增加新的请求处理类，满足开闭原则。
- 增强了给对象指派职责的灵活性，当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。
- 责任链简化了对象之间的连接，每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。
- 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

> 感觉这个"责任分担"，给了我一些使用该模式的新思路，比如有2个任务，第一轮只处理第一个任务，第二轮处理第二个任务。

缺点：

- 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
- 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
- 职责链建立的合理性要善客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

应用场景

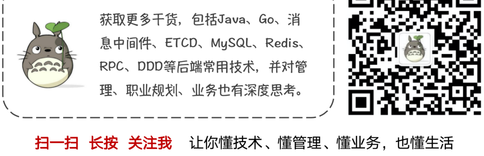
场景举例：一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同，员工必须根据自己要请假的天数去找不同的领导签名，也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。这样的例子还有很多，如找领导出差报销，生活中的"击鼓传花"游戏等。

场景分类：

- 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定。
- 可动态指定一组对象处理请求，或添加新的处理者。
- 在不明确指定请求处理者的情况下，向多个处理者中的一个提交请求。

学习交流

可以扫下面二维码，关注「楼仔」公众号。



长按二维码，回复「加群」，欢迎一起学习交流哈~ 👻👻👻

> 尽信书则不如无书，因个人能力有限，难免有疏漏和错误之处，如发现 bug 或者有更好的建议，欢迎批评指正，不吝感激。