# 基本概念

定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。

# 定义

## Context封装角色

叫做上下文角色， 起承上启下封装作用， 屏蔽高层模块对策略、 算法的直接访问，封装可能存在的变化。

## Strategy 抽象策略角色

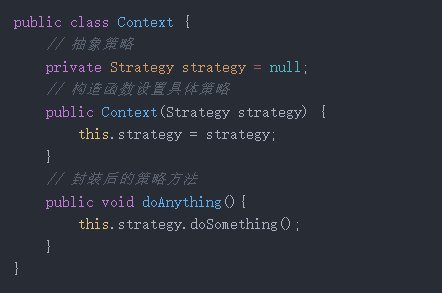
策略、 算法家族的抽象， 通常为接口， 定义每个策略或算法必须具有的方法和属性。

## ConcreteStrategy 具体策略角色

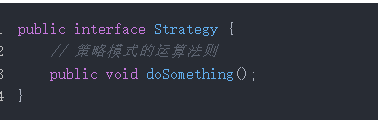
实现抽象策略中的操作， 该类含有具体的算法。

# 通用代码

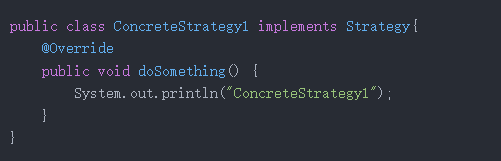
## 1、创建Context类

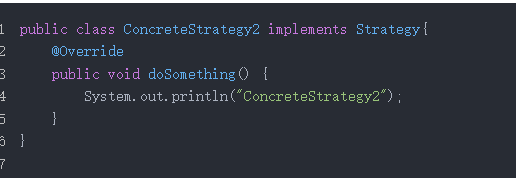


## 2、创建Strategy接口

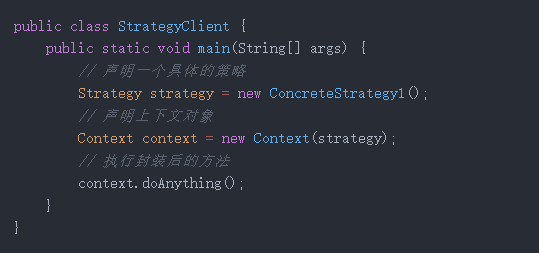


## 3、创建具体的策略实现类





## 4、测试



# 用策略模式改写if-else

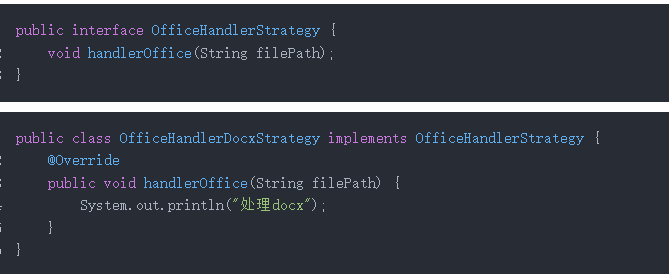
## 背景

假设我们要处理一个office文件，分为三种类型 docx、xlsx、pptx，分别表示Word文件、Excel文件、PPT文件，根据文件后缀分别解析。

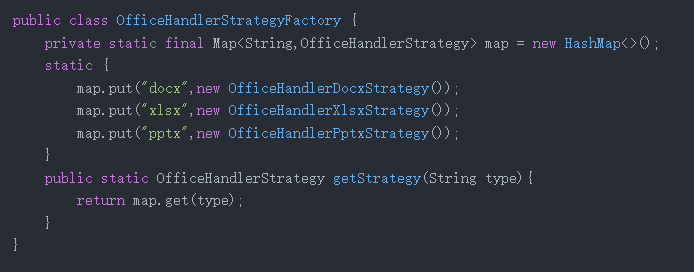


处理逻辑全部放在一个类中，会导致整个类特别庞大，假设我们要新增一种类型处理，比如对于2007版之前的office文件，后缀分别是 doc/xls/ppt，那我们得增加 else if 逻辑，违反了开闭原则，如何解决这种问题呢，答案就是通过策略模式。

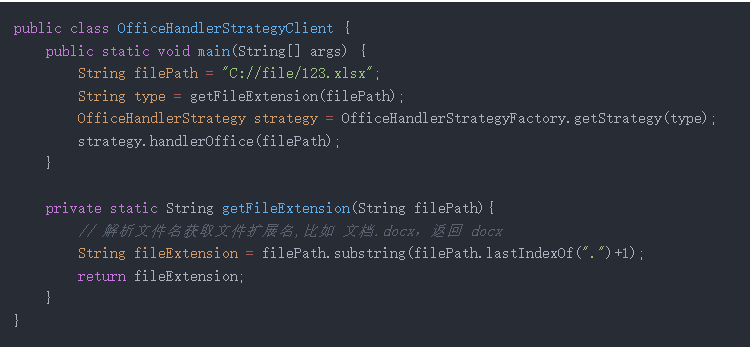
## 策略模式改写



// 省略 OfficeHandlerXlsxStrategy/OfficeHandlerPptxStrategy 类



## 测试



# 策略模式优点

## 1、算法可以自由切换

这是策略模式本身定义的， 只要实现抽象策略， 它就成为策略家族的一个成员， 通过封装角色对其进行封装， 保证对外提供“可自由切换”的策略。

## 2、避免使用多重条件判断

简化多重if-else，或多个switch-case分支。

## 3、扩展性良好

增加一个策略，只需要实现一个接口即可。

# 应用场景

1、多个类只有在算法或行为上稍有不同的场景。

2、算法需要自由切换的场景。

3、需要屏蔽算法规则的场景。