## 第24章 购买手机选项多，如何选购是难题——策略模式

## 24.1 如何挑选一部手机

兔小白：熊小猫，你对手机有研究吗？我的手机屏幕摔坏了，准备换一部新手机。

熊小猫：换个屏幕就可以了，为什么要换手机？我猜你是早就想换手机了吧？

兔小白：哈哈，被你看穿了，我确实早就想换手机，已经研究了一段时间。但是手机种类太多了，我已经挑花了眼。你给我点建议？

熊小猫：这得看你的需求是什么，我再看给你推荐的策略是什么。你是有钱任性，只选最贵，还是追求极致性能呢？

//你选最贵的准没错，不求最好，但求最贵！我还是更看重性价比

兔小白：我还是更倾向于选择性价比最高的手机。不过你也可以按照不同的侧重点都给我推荐一部，我自己再综合考虑。

熊小猫：你这是把推荐的工作翻了3倍呀！我得按照3种不同的策略帮你选手机。

兔小白：好人做到底嘛，我自己真的是选不出来，太纠结了。

熊小猫：这样吧，咱们的工作既然是软件开发，那么就用程序来解决问题。我可以指导你开发程序实现手机推荐。

兔小白：一个小问题，不至于开发程序吧？

熊小猫：问题虽然不大，但是程序设计却值得思考。

兔小白：听你这样说，我倒是很有兴趣挑战一下！

## 24.2

熊小猫：咱们就根据你挑选手机的需求来开发程序。有3种不同的手机推荐策略，不同的推荐策略推荐算法不同，推荐的手机也可能不一样。具体的推荐策略如下。

1. 选择价格最高的手机。
2. 选择性能最好的手机
3. 选择性价比最高的手机。性价比的计算规则是手机性能得分除以价格。

//图：一堆手机，三种策略各选出一种。

兔小白：这个需求的场景似曾相识，我先想想如何设计程序。

20分钟后，兔小白写完了程序。

兔小白：怪不得我觉得这个场景很熟悉，原来是简单工厂的应用模式。在程序中，先定义推荐策略接口，再开发3个推荐策略实现类。

RecommendStrategy为策略接口，定义推荐手机的方法。

//代码

HighestPriceStrategy为最高价格推荐策略。

//代码

HighestPerformanceStrategy为最优性能推荐策略。

//代码

CostPerformanceStrategy为最优性价比推荐策略。

//代码

RecommendStrategyFactory是推荐策略工厂，负责创建推荐策略实例。

//代码

客户端代码中，使用工厂对象创建的不同推荐策略实例进行推荐。

//代码

从输出可以看到，每种策略都按照各自的逻辑推荐了一部手机。

//代码

简单工厂是你最早给我讲的设计模式，我掌握的还算熟练吧？

熊小猫：使用简单工厂确实也能解决这个问题。但我的目的肯定不是让你复习简单工厂。

## 24.3

熊小猫：我们的程序设计需要贴合真实场景，这样做出来的设计才符合直觉，无论开发还是理解都容易的多。我们回归到真实场景，给你推荐手机的是推荐策略吗？

兔小白：给我推荐手机的是你呀！

熊小猫：没错，是我给你推荐手机。我通过使用不同的策略来给你推荐手机。

兔小白：你的意思是我应该定义推荐人对象，客户端和推荐人对象打交道，不应该直接和推荐策略打交道。

熊小猫：这样做不但符合真实场景，而且符合迪米特法则。客户端并不需要和那么多的推荐策略“对话”。推荐人需要留有切换策略的接口，可以按需要改变自己的推荐策略。可以考虑使用代理的思想。

兔小白：代理的思想真是无处不在，我再想想怎么优化。

15分钟后，兔小白修改完了程序。

兔小白：策略接口和三个策略实现类我没有做任何修改。我增加了推荐人Recommender类。我们先看看程序的结构图吧！

//结构图

Recommender维护一个策略类型对象，并使用该对象的推荐策略实现进行推荐。

//代码

客户端代码中，声明一个Recommender对象，然后通过他的setRecommendStrategy方法切换不同策略进行推荐。

//代码

可以看到程序输出没有任何变化。

//代码

熊小猫：优化的不错，正是我想看到的代码。现在客户端代码只需要和Recommender对象“对话”，Recommender按照设置的Strategy进行推荐。而原来的代码中，客户端需要和三个Strategy对象直接“对话”。

兔小白：但是Recommender需要和三个Strategy对象“对话”，依赖关系只是被转移了，并没有消除。不知道我这么理解对不对。

熊小猫：依赖关系确实被转移，没有被消除，但转移是有意义的。Recommender和Strategy对象构成了“推荐组件”，因此Recommender和Strategy的依赖关系是组件内依赖。客户端和Strategy的依赖是组件间的依赖。当出现多个客户端调用推荐组件时，优劣便显而易见。

//图多个人问熊小猫手机推荐，画出调用者，熊小猫、策略间的关系。

兔小白：确实将依赖关系转移到组件内部后，整个程序的依赖关系会简单很多。

熊小猫：类是一种封装，组件也是一种封装。类和类之间的依赖要尽量减少，组件和组件间的依赖更要尽量减少。这就像在公司里，部门内部的问题都好解决，一旦问题跨部门，就会变的复杂。

兔小白：这让我想起了外观模式。Recommender就像是推荐组件的外观。

熊小猫：设计思想是有相同之处，但外观模式适合更为复杂的组件场景。策略模式有其独特的适用场景。下面我们来详细讲讲策略模式。

## 24.4

熊小猫：我们先来看看策略模式的结构图。

//图

Strategy是策略接口，定义算法的公共接口。ConcreteStrategy是具体策略类，根据自己的策略实现算法。Context是上下文类，它作为算法组件的窗口，对外提供算法调用。它自己维护了一个Strategy对象的引用，它对算法的实现，实际是调用了Strategy对象。

策略模式的侧重点在于算法的封装。封装算法的策略类可扩展，并且Context可以灵活切换算法。我们看看策略模式适用场景的特点。

1. 同样的接口定义，但存在多种算法。比如例子中的商品推荐策略；时间最优和空间最优算法；路线推荐的时间优先和费用优先。
2. 需要灵活切换。既然有多种算法，一般都需要灵活切换，但也取决于业务需要。

## 24.5 策略模式和简单工厂模式

有的时候，开发人员会把策略模式的场景，错误的使用了简单工厂模式。因为每个策略类可以独立提供服务，那么客户端通过简单工厂拿到想要的策略实例也可以达到目的。

兔小白：你直接报我名字吧......不过你说的确实是事实。

熊小猫：其实两种模式的区别很大。我们需要从客户端的角度来看待两种模式。简单工厂模式属于创建型设计模式，重点在于创建对象。而策略模式属于行为型模式，重点在于封装不同的算法，对外提提供统一调用入口。当你发现简单工厂的产品家族，行为很少，不同产品实现只是算法略有区别，那么可以考虑使用策略模式。

其实两者可以结合一下。我之前说策略模式符合迪米特法则其实不是特别的准确。

兔小白：我也发现了这个问题。客户端虽然不用和策略对象直接“对话”，但在使用Context对象时，还是需要先创建策略对象，并为Context设置策略对象。

熊小猫：Context需要设置策略对象，那么策略对象从何而来呢？创建对象自然要交给创建型的设计模式。这里可以结合简单工厂来优化。你要不要来挑战一下？

兔小白：好啊，两种模式都学完了，我看看怎么结合起来！

10分钟后，兔小白改好了程序。

兔小白：我结合简单工厂创建产品的方式，改造了Recommender设置Strategy对象的方法。直接看代码吧！

//代码

这样的话，客户端代码不用去创建策略对象，只需告诉Recommender对象，自己想要使用的算法名字即可。

//代码。

熊小猫：这样改造后，客户端代码不用知道任何策略类，和策略类彻底说拜拜了！推荐组件的内部实现被完全隐藏，依赖关系更为简单，程序符合迪米特法则。