第23章

策略（Strategy）模式

策略模式是一个计划或者方式，根据给定的输入条件达成一个目标。策略和算法很相似，算法是一段程序，它可以对一组输入进行处理，获得一个输出。通常情况下，策略提供的范围比算法要广泛。这就意味着策略通常会提供一组或者一族可互换的方法。

当一个计算机程序存在多种策略时，代码就会变得复杂。要从一组策略中选择合适的策略，选择逻辑自身可能会很复杂。当多种策略的选择与执行导致复杂度增加时，就可以使用策略模式来简化它。

策略操作定义了策略的输入与输出，实现则由各个独立的类完成。这些类的实现虽然不同，但是由于接口是一致的，因此可以使用相同的接口给用户提供不同的策略进行互换。策略模式可以让一组策略共存，代码互不干扰。它还将选择策略的逻辑从策略本身中分离了出来。

策略模式的意图是将可互换的方法封装在各自独立的类中，并且让每个方法都实现一个公共的操作。

策略建模

策略模式通过将问题的不同解决方法封装在不同的类，以帮助我们组织和简化代码。为了理解策略模式是如何工作的，首先来了解一下不使用它的情况下，如何对策略进行建模。在下

一小节，我们再重构这块代码，使用策略模式来提高代码质量。

考虑 Oozinoz 公司焰火的广告策略，当用户访问该公司的网站或者给呼叫中心打电话时，都会建议客户购买该焰火产品。Oozinoz 公司使用两款现有的商业推荐引擎，帮助选择正确的广告，然后推荐给客户。Customer 类负责选择与使用其中的一款引擎，来决定给客户推荐哪一种焰火。

Rel8 是其中的一款推荐引擎，它基于用户之间的相似度进行推荐。为使该引擎能正常工作，需要让已注册的用户填写焰火弹及其他娱乐设施的相关预设信息。

如果用户没有注册，Oozinoz 公司使用另一款供应商提供的 LikeMyStuff 引擎向用户推荐，它根据用户最近购买的产品进行决策。如果系统收集的数据太少，不足以提供推荐，该软件将会随机选择一个焰火广告进行投放。然而，Oozinoz 公司出于销售方面的考虑，可能需要推出某项促销业务，而该促销业务的规则将会覆盖所有引擎的业务规则，并推出指定的焰火广告。图 23.1 展示了各个类之间的关系，用于向客户提供焰火广告。

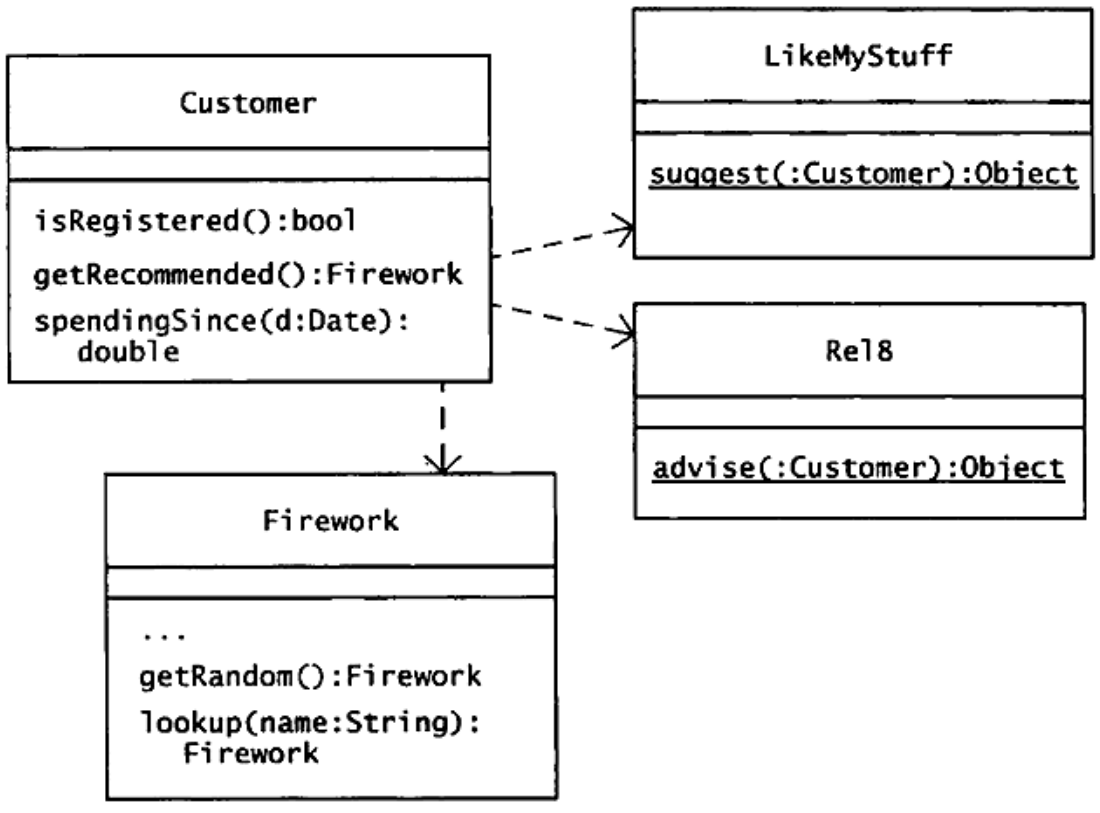

图 23.1　Customer 类依赖于其他类来获取推荐的焰火广告，包括两个现成的推荐引擎

LikeMyStuff 引擎和 Rel8 引擎都接收一个 Customer 对象，向客户推荐广告。Oozinoz 公司的这两款引擎的配置均用于提供焰火弹广告，但 LikeMyStuff 引擎依赖于数据库，而 Rel8 则依赖于对象模型。Customer 类的 getRecommended()方法体现了 Oozinoz 公司的广告策略，代码如下：

```
public Firework getRecommended() {
    // 如果对指定的焰火进行了促销，则返回
    try {
        Properties p = new Properties();
        p.load(ClassLoader.getSystemResourceAsStream(
            "config/strategy.dat"));
        String promotedName = p.getProperty("promote");

        if (promotedName != null) {
            Firework f = Firework.lookup(promotedName);
            if (f != null) return f;
        }
    } catch (Exception ignored) {
        // 如果资源丢失或加载失败，就进入下一种方式
    }

    // 如果注册了，则与其他客户进行比较
    if (isRegistered()) {
        return (Firework) Rel8.advise(this);
    }
    // 检查上一年的花费
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.YEAR, -1);
    if (spendingSince(cal.getTime()) > 1000)
        return (Firework) LikeMyStuff.suggest(this);

    // 好的，不错！
    return Firework.getRandom();
}
```

这段代码放在 Oozinoz 代码库的 com.oozinoz.recommendation 包中，可以从 www.oozinoz.com 处获取。如果对某次焰火弹进行过促销，getRecommended()方法期望将焰火弹的名字保存在 config 目录的 strategy.dat 文件中。如下所示：

```
promote=JSquirrel
```

如果该文件不存在，getRecommended()的代码在用户已经注册的前提下，将会使用 Rel8 引擎。如果没有促销策略，客户也没有注册，但用户在过去一年内通过本系统购买过商品，代码将会使用 LikeMyStuff 引擎。如果所有的推荐条件都不具备，系统将会随机选择推荐一种焰

火弹。getRecommended()方法可以工作，也许你认为它并不糟糕，但是我们希望精益求精。

重构到策略模式

getRecommended()方法有很多问题。首先，方法太长——长到需要通过注释来解释各个部分。短方法更容易被理解，几乎不需要注释，完全优于长方法。此外，getRecommended()同时完成了选择策略和执行策略两件事情，这是两件不同的事情，属于独立的功能。可以使用策略模式来简化代码。为此，需要执行以下几个步骤：

- 创建一个接口来定义策略操作。
- 分别用不同的类实现该策略接口。
- 重构代码，选择使用正确的策略类。

假设你创建了一个 Advisor 接口，如图 23.2 所示。

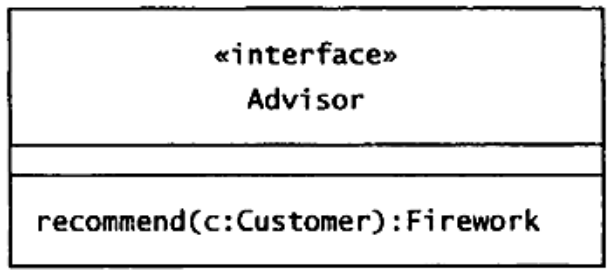

图 23.2　Advisor 接口定义了一个操作，多个类都可以使用不同的策略来实现它

Advisor 接口要求实现该接口的类接收一个客户对象，并返回一种推荐的焰火弹。在下一步重构 Customer 类的 getRecommended()代码时，需要创建几个表示推荐策略的类。每个类都会分别实现 Advisor 接口的 recommend()方法。

> **挑战 23.1**
>
> 请填充图 23.3，该图展示了推荐逻辑被重构成了一组策略类。
>
> 答案参见第 350 页

创建好策略类后，下一步就是将代码从 Customer 类的 getRecommended()方法中移到这些新类中。GroupAdvisor 和 ItemAdvisor 是两个最简单的类。它们只是简单地包装了一下对

两个现有推荐引擎的调用。接口只能定义实例方法，因此，GroupAdvisor 和 ItemAdvisor 必须被实例化，以支持 Advisor 接口。由于，我们始终需要这样一个对象，因此可以让 Customer 类持有每个实现类唯一的静态实例。图 23.4 展示了该类的一个设计。

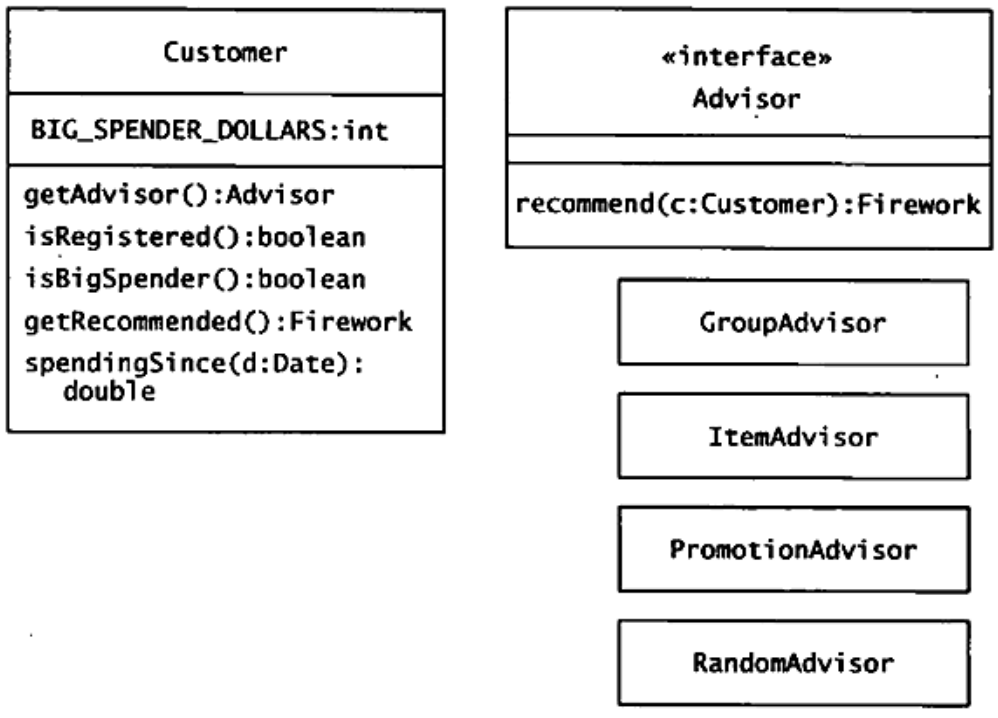

图 23.3　通过完成这个图可以展示重构推荐软件的过程。图中不同的策略都实现了同一个接口

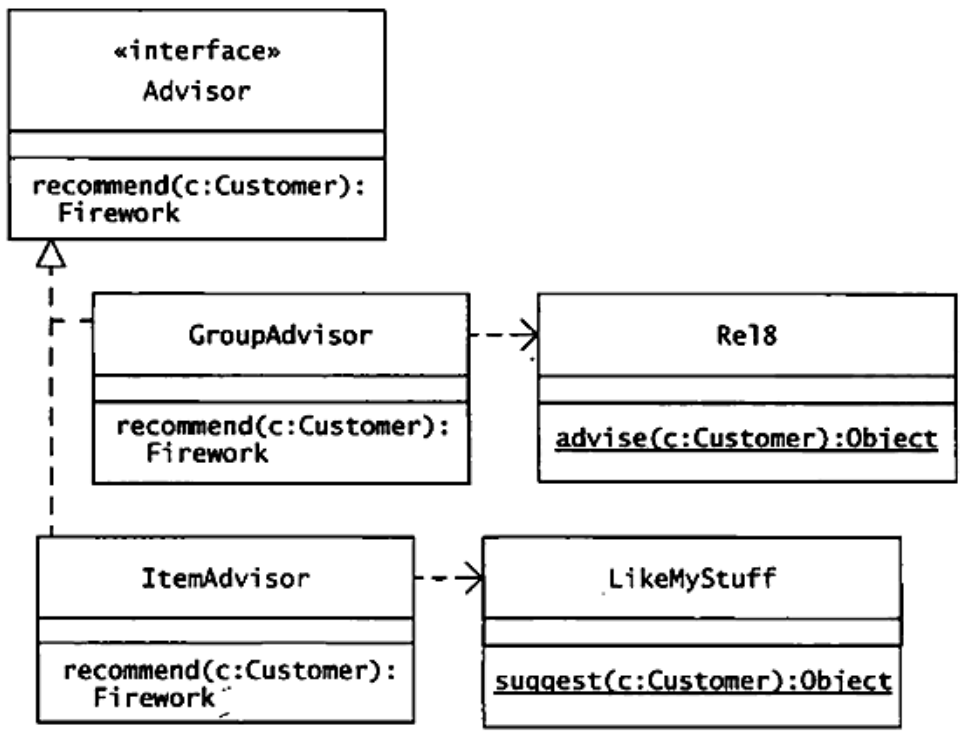

图 23.4　Advisor 接口的两个实现类提供了 recommend()方法的不同策略，它们都依赖于现有的推荐引擎

这些 advisor 类将对 recommend()方法的调用传递给了背后所需的引擎类。例如，GroupAdvisor 类将对 recommend()的调用传递给 Rel8 引擎需要的类。

```
public Firework recommend(Customer c) {
    return (Firework) Rel8.advise(c);
}
```

挑战 23.2

除了策略模式外，GroupAdvisor 和 ItemAdvisor 类还使用了哪种模式？

答案参见第 350 页

GroupAdvisor 类和 ItemAdvisor 类的工作将对 recommend()方法的调用传递给某个推荐引擎。我们还需要创建 PromotionAdvisor 类和 RandomAdvisor 类，这可以通过对 Customer 类的 getRecommended()方法进行重构来完成。与 GroupAdvisor 类和 ItemAdvisor 类一样，这两个类都会实现 recommend()操作。

PromotionAdvisor 类的构造函数应该判断是否有促销活动。你可能需要为它提供一个 hasItem()方法，用以展示是否存在促销的产品：

```
public class PromotionAdvisor implements Advisor {
    private Firework promoted;

    public PromotionAdvisor() {
        try {
            Properties p = new Properties();
            p.load(ClassLoader.getSystemResourceAsStream(
                "config/strategy.dat"));
            String promotedFireworkName = p.getProperty("promote");
            if (promotedFireworkName != null)
                promoted = Firework.lookup(promotedFireworkName);
        } catch (Exception ignored) {
            // 资源未找到或加载失败
            promoted = null;
        }
    }
```

```
    public boolean hasItem() {
        return promoted != null;
    }

    public Firework recommend(Customer c) {
        return promoted;
    }
}
```

RandomAdvisor 类很简单：

```
public class RandomAdvisor implements Advisor {
    public Firework recommend(Customer c) {
        return Firework.getRandom();
    }
}
```

通过运用策略模式对 Customer 类进行重构，将对策略的选择和执行分离开。Customer 对象的 advisor 属性保存了当前选择的策略。重构后的 Customer2 类延迟初始化了该类，它的逻辑反映了 Oozinoz 公司的广告策略：

```
private Advisor getAdvisor() {
    if (advisor == null) {
        if (promotionAdvisor.hasItem())
            advisor = promotionAdvisor;
        else if (isRegistered())
            advisor = groupAdvisor;
        else if (isBigSpender())
            advisor = itemAdvisor;
        else
            advisor = randomAdvisor;
    }
    return advisor;
}
```

挑战 23.3

请写出 Customer.getRecommended()方法的最新代码。

答案参见第 351 页

比较策略模式与状态模式

重构后的代码几乎都是简单的类和方法。这就是重构的好处，并且，重构后的代码可以方便添加新的策略。这次重构的主要目的是将某个操作分散到一组相关的类中。就这点而言，策略模式和状态模式是一致的。事实上，一些开发人员甚至认为这两种模式并无区别。

一方面，状态模式和策略模式的模型区别微乎其微。显然，多态使得状态模式和策略模式的结构看起来几乎完全一致。

另一方面，在现实生活中，策略和状态是两种完全不同的思想。在对状态和策略进行建模时，这种现实的差异就会带来不同的问题。例如，在对状态进行建模时，状态的迁移至关重要；而对策略的建模却无须考虑这一问题。另一个区别是，策略模式允许客户选择或者提供某个策略，而状态模式却很少这样设计。

状态模式和策略模式的意图迥然不同，因此我们仍将认为它们是不同的模式。但是你必须意识到这并非所有人的共识。

比较策略模式和模板方法模式

第 21 章的模板方法模式，使用了排序作为例子来描述。可以使用 `sort()`算法对任意的 `Arrays` 或者 `Collection` 类进行排序，只要你能够比较集合中的两个对象。但也可以声称：改变两个对象的比较算法，其实就是在改变策略。例如，在销售火箭时，按照价格排序和按照推动力排序就是两种不同的策略[译注1]。

译注1：从结构上讲，策略模式的抽象策略通常定义为接口，而模板方法模式则定义为抽象类。虽然二者都可以看做是对算法或策略的抽象，但模板方法模式往往规定好整个策略的框架，子类实现的常常是策略的一部分；而策略模式则是对整个算法进行抽象。对于第 21 章给出的例子而言，如果针对排序，这自然是采用模板方法模式，但如果将比较看做是我们要抽象的算法，并将比较逻辑分离为单独的接口，则可以认为在模板方法模式中针对比较算法又运用了策略模式。

挑战 23.4

请问 Arrays.sort()方法属于模板方法模式，还是策略模式？

答案参见第 351 页

小结

不同策略模型的逻辑可能会放到同一个类中，并常常写在一个方法里。这样的方法可能过于复杂，并且混合了策略的选择逻辑和执行逻辑[译注2]。

为了简化这样的代码，需要创建一组类，每个类代表一种策略。定义一个接口方法，并让这些类都实现它。这就使得每个类都封装了一种策略，代码也会变得简单。同时，还需要为策略提供选择逻辑。该逻辑在重构后仍然可能比较复杂，但是可以简化这些代码，使其等价于问题域中描述策略的伪代码。

典型情况下，客户会使用一个上下文变量来维护对策略的选择。通过将策略操作的调用转发给上下文，并使用多态执行正确的策略，可以使得策略的执行变得更简单。策略模式将可相互替换的策略封装到不同的类，并让这些类实现一个公共接口，这样就可以帮助我们创建简洁的代码，为解决问题的各种方式建立统一的模型。

译注2：这事实上也违背了单一职责原则。