策略模式

亦称： Strategy

意图

策略模式是一种行为设计模式，它能让你定义一系列算法，并将每种算法分别放入独立的类中，以使算法的对象能够相互替换。

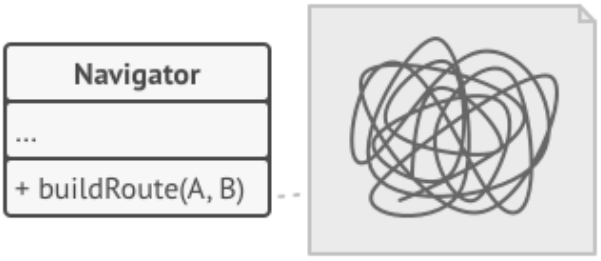
问题

一天，你打算为游客们创建一款导游程序。该程序的核心功能是提供美观的地图，以帮助用户在任何城市中快速定位。

用户期待的程序新功能是自动路线规划：他们希望输入地址后就能在地图上看到前往目的地的最快路线。

程序的首个版本只能规划公路路线。驾车旅行的人们对此非常满意。但很显然，并非所有人都会在度假时开车。因此你在下次更新时添加了规划步行路线的功能。此后，你又添加了规划公共交通路线的功能。

而这只是个开始。不久后，你又要为骑行者规划路线。又过了一段时间，你又要为游览城市中的所有景点规划路线。



导游代码将变得非常臃肿。

尽管从商业角度来看，这款应用非常成功，但其技术部分却让你非常头疼：每次添加新的路线规划算法后，导游应用中主要类的体积就会增加一倍。终于在某个时候，你觉得自己没法继续维护这堆代码了。

无论是修复简单缺陷还是微调街道权重，对某个算法进行任何修改都会影响整个类，从而增加在已有正常运行代码中引入错误的风险。

此外，团队合作将变得低效。如果你在应用成功发布后招募了团队成员，他们会抱怨在合并冲突的工作上花费了太多时间。在实现新功能的过程中，你的团队需要修改同一个巨大的类，这样他们所编写的代码相互之间就可能会出现冲突。

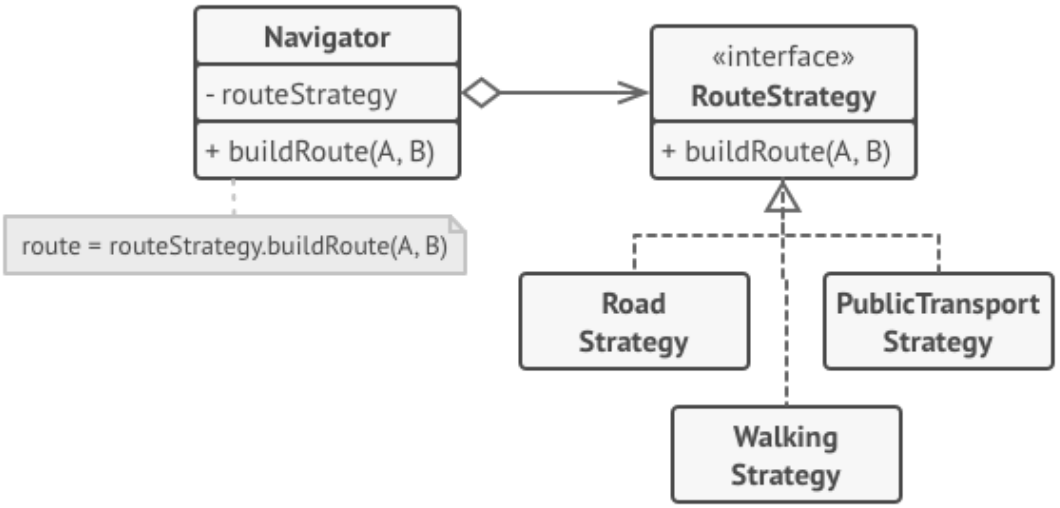
😊 解决方案

策略模式建议找出负责用许多不同方式完成特定任务的类，然后将其中的算法抽取到一组被称为**策略**的独立类中。

名为**上下文**的原始类必须包含一个成员变量来存储对于每种策略的引用。上下文并不执行任务，而是将工作委派给已连接的策略对象。

上下文不负责选择符合任务需要的算法——客户端会将所需策略传递给上下文。实际上，上下文并不十分了解策略，它会通过同样的通用接口与所有策略进行交互，而该接口只需暴露一个方法来触发所选策略中封装的算法即可。

因此，上下文可独立于具体策略。这样你就可在不修改上下文代码或其他策略的情况下添加新算法或修改已有算法了。

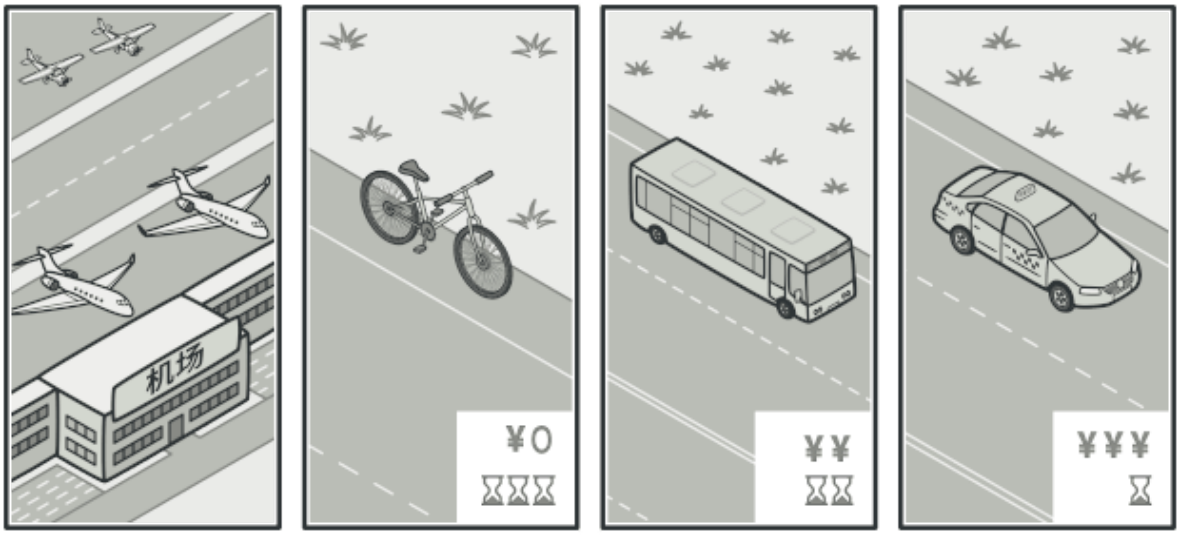


路线规划策略。

在导游应用中，每个路线规划算法都可被抽取到只有一个 `buildRoute` 生成路线 方法的独立类中。该方法接收起点和终点作为参数，并返回路线中途点的集合。

即使传递给每个路径规划类的参数一模一样，其所创建的路线也可能完全不同。主要导游类的主要工作是在地图上渲染一系列中途点，不会在意如何选择算法。该类中还有一个用于切换当前路径规划策略的方法，因此客户端（例如用户界面中的按钮）可用其他策略替换当前选择的路径规划行为。

真实世界类比



各种前往机场的出行策略

假如你需要前往机场。你可以选择乘坐公共汽车、预约出租车或骑自行车。这些就是你的出行策略。你可以根据预算或时间等因素来选择其中一种策略。

策略模式结构

1. **上下文**（Context）维护指向具体策略的引用，且仅通过策略接口与该对象进行交流。
2. **策略**（Strategy）接口是所有具体策略的通用接口，它声明了一个上下文用于执行策略的方法。
3. **具体策略**（Concrete Strategies）实现了上下文所用算法的各种不同变体。
4. 当上下文需要运行算法时，它会在其已连接的策略对象上调用执行方法。上下文不清楚其所涉及的策略类型与算法的执行方式。
5. **客户端**（Client）会创建一个特定策略对象并将其传递给上下文。上下文则会提供一个设置器以便客户端在运行时替换相关联的策略。

伪代码

在本例中，上下文使用了多个**策略**来执行不同的计算操作。

```
// 策略接口声明了某个算法各个不同版本间所共有的操作。上下文会使用该接口来
// 调用有具体策略定义的算法。
interface Strategy is
    method execute(a, b)

// 具体策略会在遵循策略基础接口的情况下实现算法。该接口实现了它们在上下文
// 中的互换性。
class ConcreteStrategyAdd implements Strategy is
    method execute(a, b) is
        return a + b

class ConcreteStrategySubtract implements Strategy is
    method execute(a, b) is
        return a - b

class ConcreteStrategyMultiply implements Strategy is
    method execute(a, b) is
        return a * b

// 上下文定义了客户端关注的接口。
class Context is
    // 上下文会维护指向某个策略对象的引用。上下文不知晓策略的具体类。上下
    // 文必须通过策略接口来与所有策略进行交互。
    private strategy: Strategy

    // 上下文通常会通过构造函数来接收策略对象，同时还提供设置器以便在运行
    // 时切换策略。
    method setStrategy(Strategy strategy) is
        this.strategy = strategy

    // 上下文会将一些工作委派给策略对象，而不是自行实现不同版本的算法。
    method executeStrategy(int a, int b) is
        return strategy.execute(a, b)


// 客户端代码会选择具体策略并将其传递给上下文。客户端必须知晓策略之间的差
// 异，才能做出正确的选择。
class ExampleApplication is
    method main() is

        创建上下文对象。

        读取第一个数。
        读取最后一个数。
        从用户输入中读取期望进行的行为。

        if (action == addition) then
            context.setStrategy(new ConcreteStrategyAdd())

        if (action == subtraction) then
            context.setStrategy(new ConcreteStrategySubtract())

        if (action == multiplication) then
            context.setStrategy(new ConcreteStrategyMultiply())
```

```
result = context.executeStrategy(First number, Second number)

打印结果。
```

策略模式适合应用场景

当你想使用对象中各种不同的算法变体，并希望能在运行时切换算法时，可使用策略模式。

策略模式让你能够将对象关联至可以不同方式执行特定子任务的不同子对象，从而以间接方式在运行时更改对象行为。

当你有许多仅在执行某些行为时略有不同的相似类时，可使用策略模式。

策略模式让你能将不同行为抽取到一个独立类层次结构中，并将原始类组合成同一个，从而减少重复代码。

如果算法在上下文的逻辑中不是特别重要，使用该模式能将类的业务逻辑与其算法实现细节隔离开来。

策略模式让你能将各种算法的代码、内部数据和依赖关系与其他代码隔离开来。不同客户端可通过一个简单接口执行算法，并能在运行时进行切换。

当类中使用了复杂条件运算符以在同一算法的不同变体中切换时，可使用该模式。

策略模式将所有继承自同样接口的算法抽取到独立类中，因此不再需要条件语句。原始对象并不实现所有算法的变体，而是将执行工作委派给其中的一个独立算法对象。

实现方式

1. 从上下文类中找出修改频率较高的算法（也可能是用于在运行时选择某个算法变体的复杂条件运算符）。

2. 声明该算法所有变体的通用策略接口。

3. 将算法逐一抽取到各自的类中，它们都必须实现策略接口。

4. 在上下文类中添加一个成员变量用于保存对于策略对象的引用。然后提供设置器以修改该成员变量。上下文仅可通过策略接口同策略对象进行交互，如有需要还可定义一个接口来让策略访问其数据。

5. 客户端必须将上下文类与相应策略进行关联，使上下文可以预期的方式完成其主要工作。

⚖ 策略模式优缺点

- ✓ 你可以在运行时切换对象内的算法。
- ✓ 你可以将算法的实现和使用算法的代码隔离开来。
- ✓ 你可以使用组合来代替继承。
- ✓ 开闭原则。你无需对上下文进行修改就能够引入新的策略。
- ✗ 如果你的算法极少发生改变，那么没有任何理由引入新的类和接口。使用该模式只会让程序过于复杂。
- ✗ 客户端必须知晓策略间的不同——它需要选择合适的策略。
- ✗ 许多现代编程语言支持函数类型功能，允许你在一组匿名函数中实现不同版本的算法。这样，你使用这些函数的方式就和使用策略对象时完全相同，无需借助额外的类和接口来保持代码简洁。

⇄ 与其他模式的关系

- **桥接模式**、**状态模式**和**策略模式**（在某种程度上包括**适配器模式**）模式的接口非常相似。实际上，它们都基于**组合模式**——即将工作委派给其他对象，不过也各自解决了不同的问题。模式并不只是以特定方式组织代码的配方，你还可以使用它们来和其他开发者讨论模式所解决的问题。

- **命令模式**和**策略**看上去很像，因为两者都能通过某些行为来参数化对象。但是，它们的意图有非常大的不同。

    - 你可以使用命令来将任何操作转换为对象。操作的参数将成为对象的成员变量。你可以通过转换来延迟操作的执行、将操作放入队列、保存历史命令或者向远程服务发送命令等。

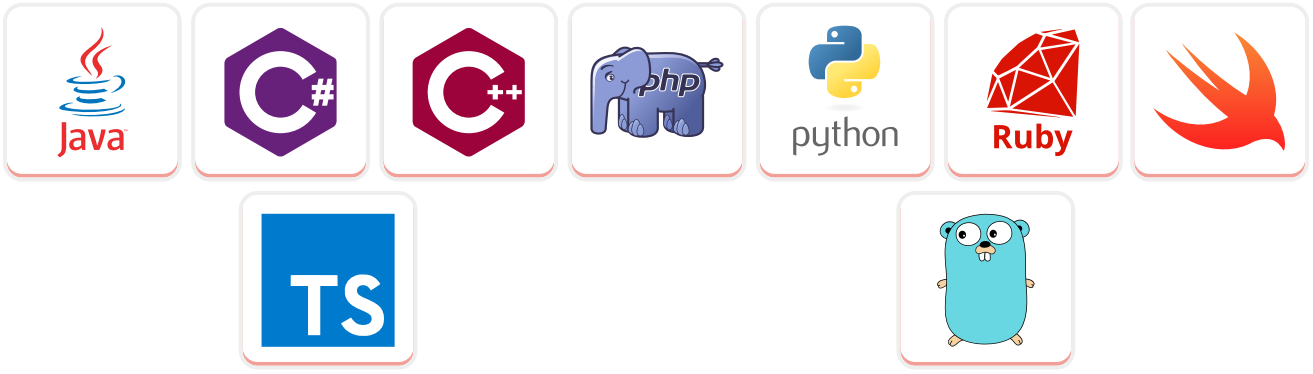
  - 另一方面，策略通常可用于描述完成某件事的不同方式，让你能够在同一个上下文类中切换算法。

- **装饰模式**可让你更改对象的外表，**策略**则让你能够改变其本质。

- **模板方法模式**基于继承机制：它允许你通过扩展子类中的部分内容来改变部分算法。**策略**基于组合机制：你可以通过对相应行为提供不同的策略来改变对象的部分行为。模板方法在类层次上运作，因此它是静态的。策略在对象层次上运作，因此允许在运行时切换行为。

- **状态**可被视为**策略**的扩展。两者都基于组合机制：它们都通过将部分工作委派给“帮手” 对象来改变其在不同情景下的行为。策略使得这些对象相互之间完全独立，它们不知道其他对象的存在。但状态模式没有限制具体状态之间的依赖，且允许它们自行改变在不同情景下的状态。

</> 代码示例



主页

重构

设计模式

会员专属内容

论坛

联系我们

Khmelnitske shosse 19 / 27, Kamianets-Podilskyi, 乌克兰, 32305

Email: support@refactoring.guru

图片作者： Dmitry Zhart

条款与政策

隐私政策

内容使用政策