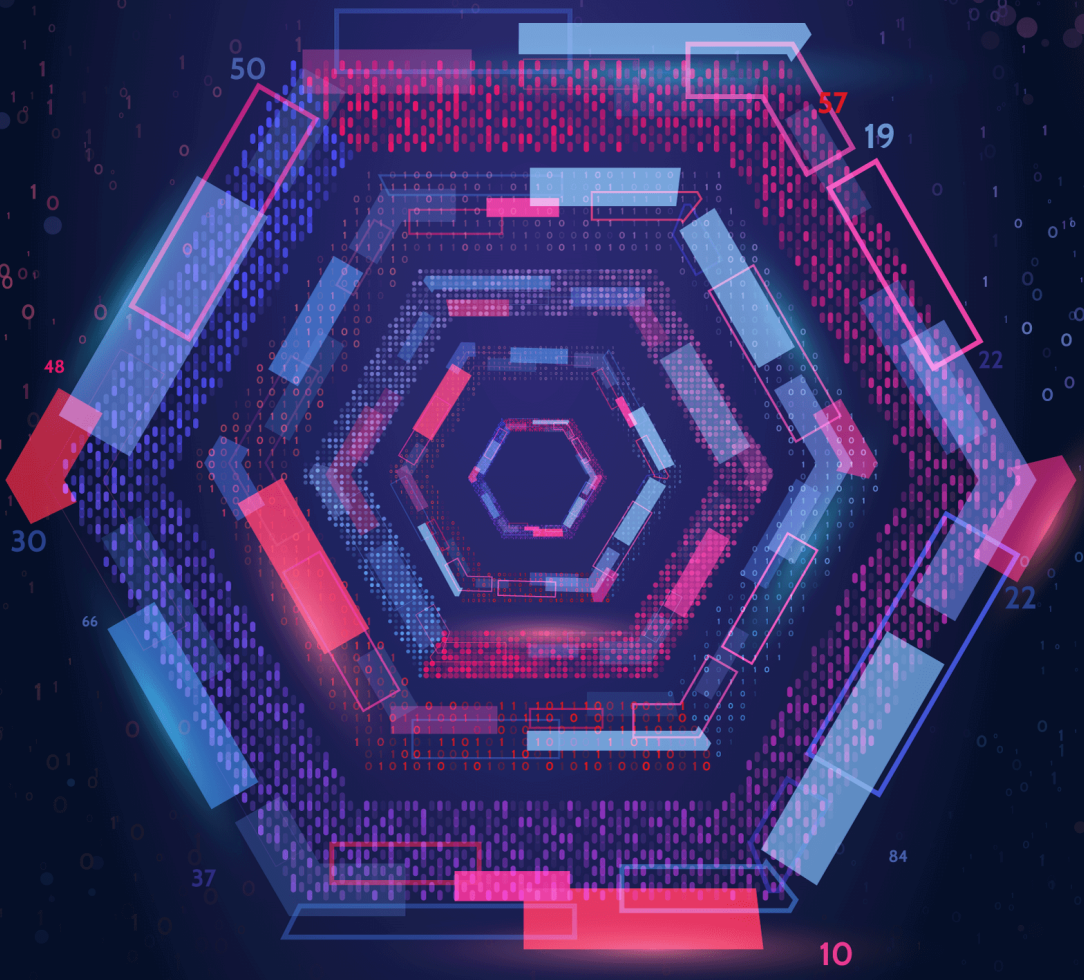
# 深入
# 设计模式




亚历山大·什韦茨 (Alexander Shvets) 著
彭力 译

REFACTORING
·GURU·

# 深入
# 设计模式

v2020-1.20

亚历山大·什韦茨（Alexander Shvets） 著
彭力 译

# 关于版权的简要说明

大家好！ 我叫亚历山大·什韦茨，是《**深入设计模式**》一书以及在线课程《**深入代码重构**》的作者。

本书仅供您个人使用。请不要与家庭成员之外的第三方分享本书。如果您想和朋友或同事分享的话，可以再次购买并赠予他们。您还可以为整个团队或整个公司购买站点许可证。

书籍和课程的销售所得都将用于 **Refactoring.Guru** 网站的开发工作。售出的每件产品都将给该项目以极大的帮助，并且能稍微提前新书发布的时间。



**support@refactoring.guru**

插图：迪米特里·扎特（Dmitry Zhart）
中文版翻译：彭力
编辑：黄佳珍

*谨以此书献给我的夫人玛丽亚。没有她的话，我很可能要到 30 年后才能写完这本书。*

# 目录

**目录** ..... **4**
**如何阅读本书** ..... **6**
**面向对象程序设计简介** ..... **7**
面向对象程序设计基础 ..... 8
面向对象程序设计基础 ..... 13
对象之间的关系 ..... 20
**设计模式简介** ..... **26**
什么是设计模式？ ..... 27
为什么以及如何学习设计模式？ ..... 31
**软件设计原则** ..... **32**
优秀设计的特征 ..... 33
**设计原则** ..... **37**
封装变化的内容 ..... 38
面向接口进行开发，而不是面向实现 ..... 42
组合优于继承 ..... 47
**SOLID 原则** ..... **51**
S: 单一职责原则 ..... 52
O: 开闭原则 ..... 54
L: 里氏替换原则 ..... 57
I: 接口隔离原则 ..... 63
D: 依赖倒置原则 ..... 66

**设计模式目录 ..... 69**

**创建型模式 ..... 70**

工厂方法 / *Factory Method* ..... 72

抽象工厂 / *Abstract Factory* ..... 86

生成器 / *Builder* ..... 101

原型 / *Prototype* ..... 119

单例 / *Singleton* ..... 132

**结构型模式 ..... 140**

适配器 / *Adapter* ..... 143

桥接 / *Bridge* ..... 155

组合 / *Composite* ..... 169

装饰 / *Decorator* ..... 181

外观 / *Facade* ..... 198

享元 / *Flyweight* ..... 207

代理 / *Proxy* ..... 219

**行为模式 ..... 231**

责任链 / *Chain of Responsibility* ..... 235

命令 / *Command* ..... 253

迭代器 / *Iterator* ..... 272

中介者 / *Mediator* ..... 287

备忘录 / *Memento* ..... 300

观察者 / *Observer* ..... 315

状态 / *State* ..... 329

策略 / *Strategy* ..... 345

模板方法 / *Template Method* ..... 357

访问者 / *Visitor* ..... 369

**结语 ..... 384**

# 如何阅读本书

本书对“四人组（GoF）”于 1994 年提出的 22 个经典设计模式进行了详细说明。

每章都会讨论一个特定的模式。因此你可以按照顺序从头到尾进行阅读，也可以挑选自己感兴趣的模式进行阅读。

许多模式之间存在着相互联系，你可以使用大量的链接在主题间跳转。每章末尾会列出与当前模式相关的其他模式的链接列表。如果你看到了一个此前从未见过的模式名称的话，只需接着往下读即可——其内容将会在后续章节中出现。

设计模式是通用的。因此本书中的所有示例代码都以伪代码的形式呈现，而不会出现特定编程语言的内容。

学习模式之前，你可以复习**面向对象程序设计的关键术语**来回忆相关知识。这一章还会介绍 UML 图的基础知识，这些知识非常实用，因为书中会有许多 UML 图。当然，如果你已经知晓了所有这些内容的话，也可以直接开始**学习设计模式**。

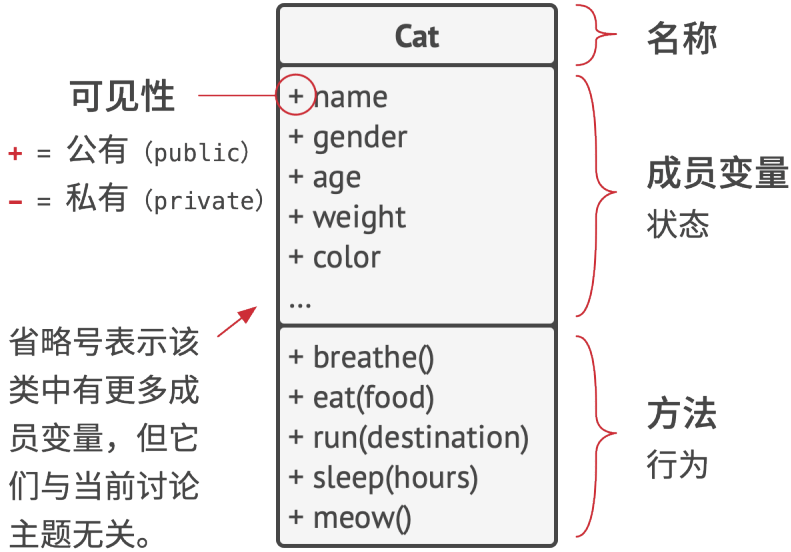
# 面向对象程序设计简介

# 面向对象程序设计基础

**面向对象程序设计**（Object-Oriented Programming，缩写为OOP）是一种范式，其基本理念是将数据块及与数据相关的行为封装成为特殊的、名为**对象**的实体，同时对象实体的生成工作则是基于程序员给出的一系列“蓝图”，这些“蓝图”就是**类**。

## 对象和类

你喜欢猫吗？希望你喜欢，因为我将用和猫有关的各种示例来解释面向对象程序设计的概念。

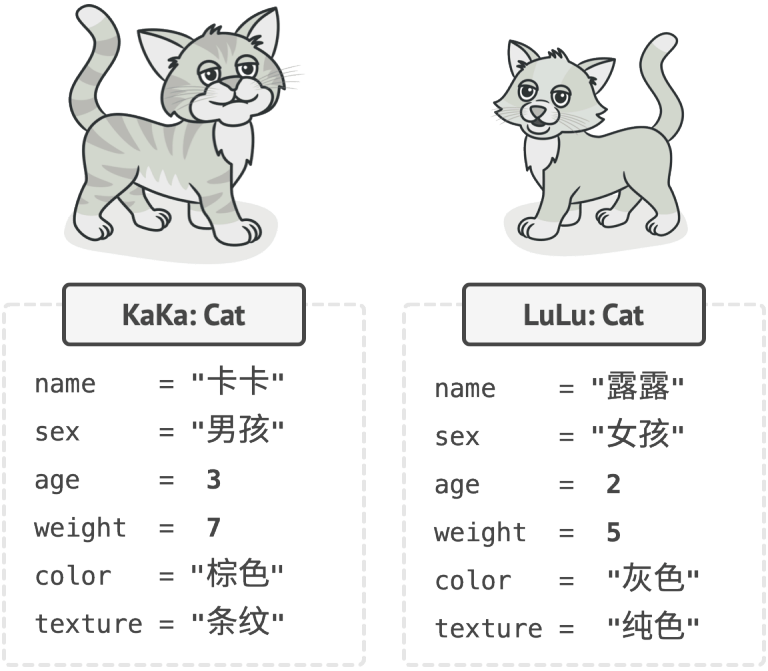



这是一个 UML 类图。你将在本书中看到许多类似的图示。

> 将图表中的类和成员名称保留为英文是一种标准做法，这和在真实代码中一样。但是，注释和备注也可以用中文编写。
>
> 在本书中，我会用中文指代类名，即使它们在图表或代码中以英文的形式出现（就像我处理 `猫` 类那样）。我希望大家在读这本书时，就像和我进行一场朋友间的谈话。我不希望每次要引用某个类时都会让大家碰到生词。

假如你有一只名为卡卡的猫。卡卡是一个对象，也是 `猫 Cat` 这个类的一个实例。每只猫都有许多基本属性：`名字 name`、`性别 sex`、`年龄 age`、`体重 weight`、`毛色 color` 和最爱的食物等。这些都是该类的**成员变量**。

所有猫都有相似的行为：`它们会呼吸 breathe`、`进食 eat`、`奔跑 run`、`睡觉 sleep` 和 `喵喵叫 meow`。这些都是该类的**方法**。成员变量和方法可以统称为类的成员。存储在对象成员变量中的数据通常被称为状态，对象中的所有方法则定义了其行为。




对象是类的实例。

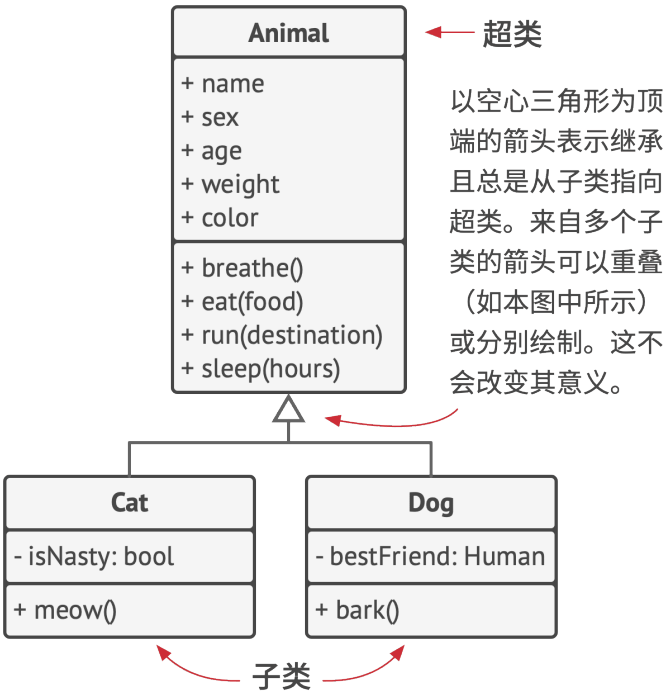
你朋友的猫“露露”也是 `猫` 这个类的一个实例。它拥有与“卡卡”相同的一组属性。不同之处在于这些属性的值：她的性别是“女孩”；她的毛色不同；体重较轻。因此类就像是定义对象结构的蓝图，而对象则是类的具体实例。

## 类层次结构

相信大家都已经了解单独的一个类的结构了，但一个实际的程序显然会包含不止一个类。一些类可能会组织起来形成**类层次结构**。让我们了解一下这是什么意思。

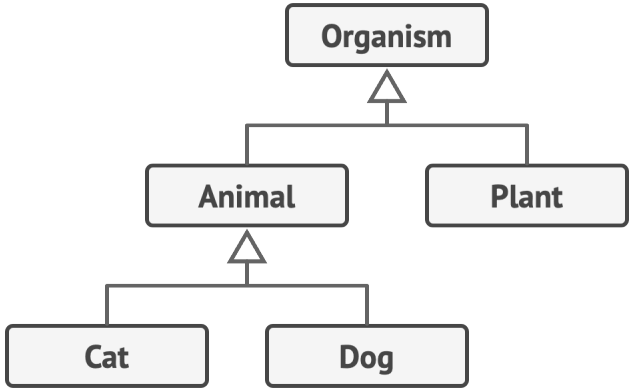
假如你的邻居有一只名为“福福”的狗。其实狗和猫有很多相同的地方：它们都有名字、性别、年龄和毛色等属性。狗和猫一样可以呼吸、睡觉和奔跑。因此似乎我们可定义一个动物 Animal 基类来列出它们所共有的属性和行为。

我们刚刚定义的父类被称为**超类**。继承它的类被称为**子类**。子类会继承其父类的状态和行为，其中只需定义不同于父类的属性或行为。因此，猫类将包含 meow 喵喵叫 方法，而狗 Dog 类则将包含 bark 汪汪叫 方法。




类层次结构的 UML 图。图中所有的类都是 动物 类层次结构中的一部分。

假如我们接到一个相关的业务需求，那就可以继续为所有活的 生物体 Organisms 抽取出一个更通用的类，并将其作为 动物 和 植物 Plants 类的超类。这种由各种类组成的金字塔就是**层次结构**。在这个层次结构中， 猫 类将继承 动物 和 生物体 类的全部内容。

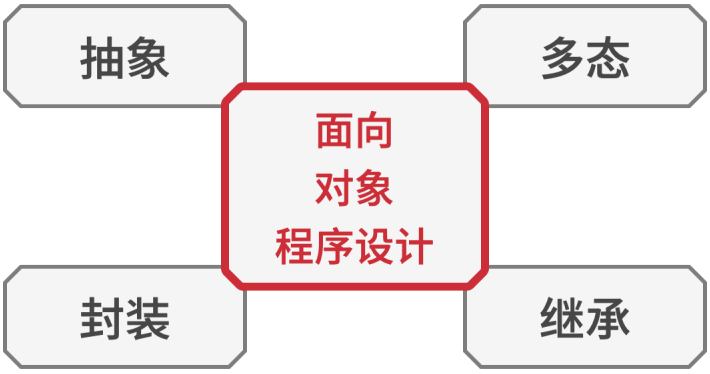



如果展示类之间的关系比展示其内容更重要的话，那可对 UML 图中的类进行简化。

子类可以对从父类中继承而来的方法的行为进行重写。子类可以完全替换默认行为，也可以仅提供额外内容来对其进行加强。

# 面向对象程序设计基础

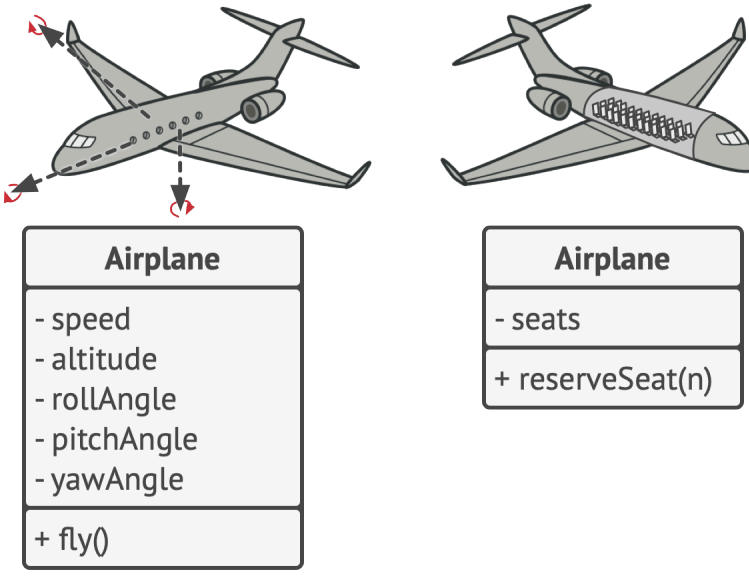
面向对象程序设计的四个基本概念使其区别于其他程序设计范式。




## 抽象

当使用面向对象程序设计的理念开发一款程序时，你会将大部分时间用于根据真实世界对象来设计程序中的对象。但是，程序中的对象并不需要能够百分之百准确地反映其原型（极少情况下才需要做到这一点）。实际上，你的对象只需模拟真实对象的特定属性和行为即可，其他内容可以忽略。

例如，飞行模拟器和航班预订程序中都可能会包含一个飞机 Airplane 类。但是前者需包含与实际飞行相关的详细信息，而后者则只关心座位图和哪些座位可供预订。




同一个真实世界对象的不同模型。

抽象是一种反映真实世界对象或现象中特定内容的模型，它能高精度地反映所有与特定内容相关的详细信息，同时忽略其他内容。

## 封装

如果想要启动一辆车的发动机，你只需转动钥匙或按下按钮即可，无需打开引擎盖手动接线、转动曲轴和气缸并启动发动机的动力循环。这些细节都隐藏在引擎盖下，你只会看到一些简单的接口：启动开关、方向盘和一些踏板。该示例讲述了什么是对象的**接口**——它是对象的公有部分，能够同其他对象进行交互。

**封装**是指一个对象对其他对象隐藏其部分状态和行为，而仅向程序其他部分暴露有限的接口的能力。

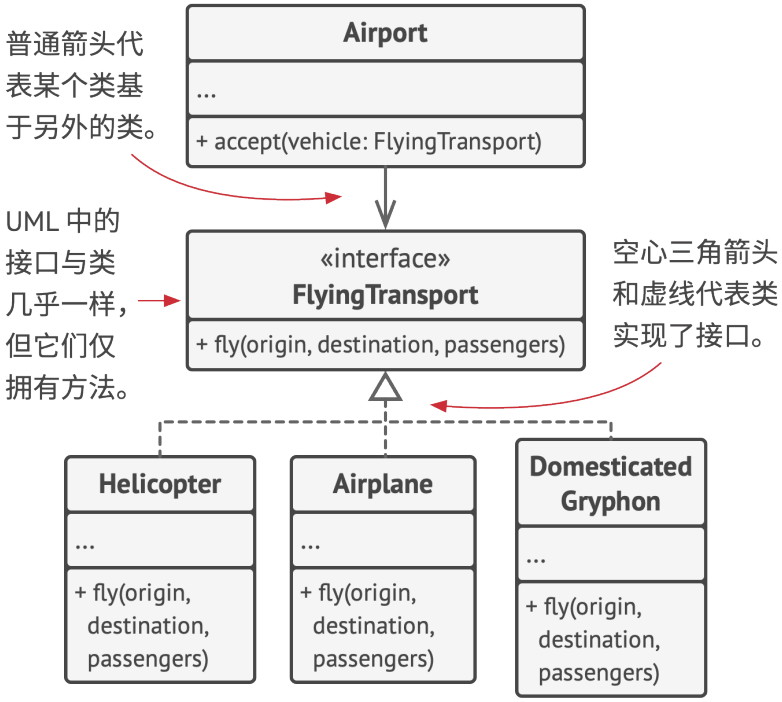
**封装**某个内容意味着使用关键字 private 私有来对其进行修饰，这样仅有其所在类中的方法才能访问这些内容。还有一种限制程度较小的关键字 protected 保护，其所修饰的对象仅允许父类访问其类中的成员。

绝大部分编程语言的接口和抽象类（或方法）都基于抽象和封装的概念。在现代面向对象的编程语言中，接口机制（通常使用 interface 或 protocol 关键字来声明）允许你定义对象之间的交互协议。这也是接口仅关心对象行为，以及你不能在接口中声明成员变量的原因之一。

> 由于**接口**（interface）这个词代表对象的公有部分，而在绝大部分编程语言中又有 interface 类型，因此很容易造成混淆。在这里我将对此进行说明。

假如你的 航空运输 FlyingTransport 接口中有一个 fly(origin, destination, passengers) 方法（即以起点、终点以及乘客为参数的飞行方法）。在设计航空运输模拟器时，你可以对 机场 Airport 类做出限制，使其仅与实现了 航空运输 接口的对象进行交互。此后，你可以确保传递给机场对象的任何对象——无论是 飞机、直升机

Helicopter 还是可怕的家养狮鹫 DomesticatedGryphon——都能到达或离开这种类型的机场。




多个类实现一个接口的 UML 图。

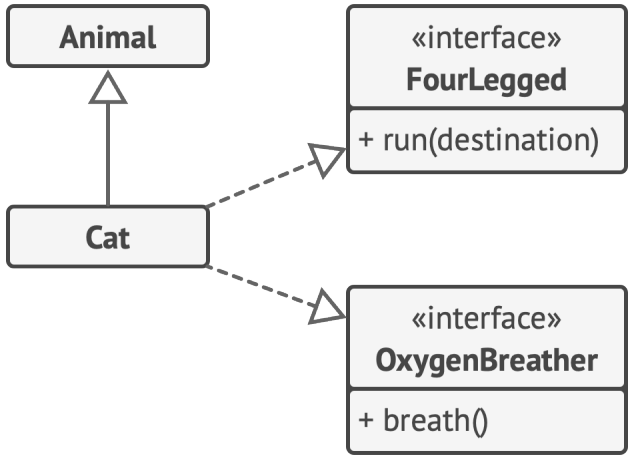
你可以任何方式更改这些类中 fly 方法的实现方式。只要方法签名与接口中的声明保持一致，那么所有机场类的实例都能与飞行对象进行交互。

## 继承

**继承**是指在根据已有类创建新类的能力。继承最主要的好处是代码复用。如果你想要创建的类与已有的类差异不大，那

也没必要重复编写相同的代码。你只需扩展已有的类并将额外功能放入生成的子类（它会继承父类的成员变量和方法）中即可。

使用继承后，子类将拥有与其父类相同的接口。如果父类中声明了某个方法，那么你将无法在子类中隐藏该方法。你还必须实现所有的抽象方法，即使它们对于你的子类而言没有意义。

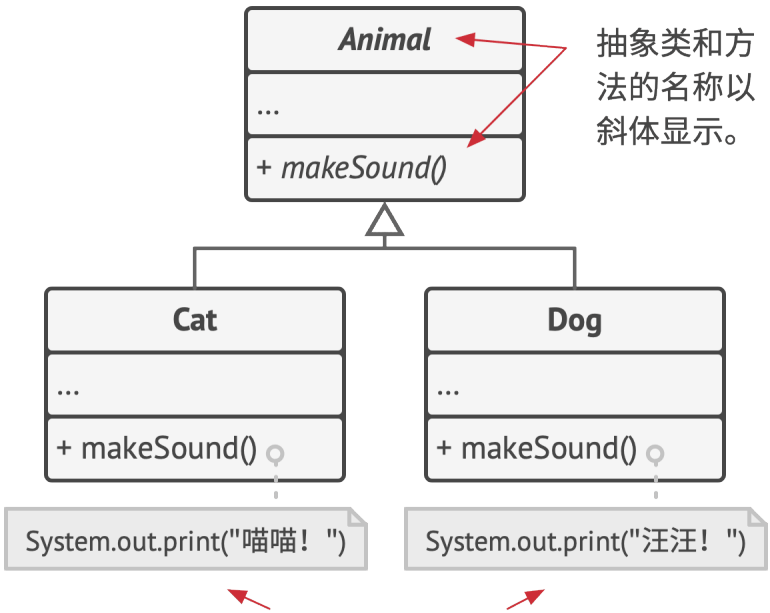



继承单个类和同时实现多个接口的 UML 图。

在绝大多数编程语言中，子类仅能对一个父类进行扩展。另一方面，任何类都可以同时实现多个接口。但是正如我之前提到的那样，如果父类实现了某个接口，那么其所有子类都必须实现该接口。

## 多态

让我们看一些动物的例子。绝大部分 动物 Animals 可以发出声音。我们需要所有子类都重写基类的 makeSound 发出声音 方法，让每个子类都发出正确的声音，因此我们可以马上将其声明为抽象。这让我们得以忽略父类中该方法的所有默认实现，从而强制要求所有子类自行提供该方法的实现。




假如你将几只猫和狗放入一个大袋子中。然后，我们闭上眼睛，将动物一个一个地从袋中取出。我们并不知道自己取出的是何种动物。但如果我们好好地摸摸它们的话，它就会根据自己的具体类发出特殊的欢快叫声。

```
bag = [new Cat(), new Dog()];

foreach (Animal a : bag)
  a.makeSound()

// 喵喵!
// 汪汪!
```

程序并不知道 `a` 变量中所包含的对象的具体类型，但幸亏有被称为**多态**的特殊机制，程序可以追踪对象的子类并调用其方法，从而执行恰当的行为。

**多态**是指程序能够检测对象所属的实际类，并在当前上下文不知道其真实类型的情况下调用其实现的能力。

你还可将多态看作是一个对象“假扮”为其他东西（通常是其扩展的类或实现的接口）的能力。在我们的示例中，袋中的狗和猫就相当于是假扮成了一般的动物。

# 对象之间的关系

除了之前我们已见到的继承和实现之外，对象之间还有其他我们尚未提及的关系。

## 依赖


UML 图中的依赖。教授依赖于课程资料。

依赖是类之间最基础的、也是最微弱的关系类型。如果修改一个类的定义可能会造成另一个类的变化，那么这两个类之间就存在依赖关系。当你在代码中使用具体类的名称时，通常意味着存在依赖关系。例如在指定方法签名类型时，或是通过调用构造函数对对象进行初始化时等。通过让代码依赖接口或抽象类（而不是具体类），你可以降低其依赖程度。

通常情况下，UML 图不会展示所有依赖——它们在真实代码中的数量太多了。为了不让依赖关系破坏 UML 图，你必须对其进行精心选择，仅展示那些对于沟通你的想法来说重要的依赖关系。

## 关联


UML 图中的关联。教授与学生进行交流。

关联是一个对象使用另一对象或与另一对象进行交互的关系。在 UML 图中，关联关系用起始于一个对象并指向其所使用的对象的简单箭头来表示。顺带一提，双向关联也是完全正常的，这种情况就用双向箭头来表示。关联可视为一种特殊类型的依赖，即一个对象总是拥有访问与其交互的对象的权限，而简单的依赖关系并不会在对象间建立永久性的联系。

一般来说，你可以使用关联关系来表示类似于类成员变量的东西。这个关系将一直存在，因此你总能通过“订单”来获取其“顾客”。但是它并非一定是成员变量。如果你根据接口来创建类，它也可以表示为一个可返回“订单”的“顾客”的方法。

为了巩固你对关联和依赖之间区别的理解，下面让我们来看一个两者结合的示例。假设我们有一个名为 `教授`（Professor）的类：

```
class Professor is
  field Student student
  // ...
  method teach(Course c) is
    // ...
    this.student.remember(c.getKnowledge())
```

让我们来看看 teach （教授知识）方法，它将接收一个来自 课程 （Course）类的参数。如果有人修改了 课程 类的 getKnowledge （获取知识）方法（修改方法名或添加一些必须的参数等），代码将会崩溃。这就是依赖关系。

现在，让我们来看看名为 student （学生）的成员变量，以及如何在 teach 方法中使用该变量。我们可以肯定 学生 （Student）类是 教授 类的依赖：如果 remember （记住）方法被修改， 教授 的代码也将崩溃。但由于 教授 的所有方法总能访问 student 成员变量，所以 学生 类就不仅是依赖，而也是关联了。

## 聚合


UML 图中的聚合。院系包含教授。

**聚合**是一种特殊类型的关联，用于表示多个对象之间的“一对多”、“多对多”或“整体对部分”的关系。普通关联仅用于描述两个对象之间的关系。通常在聚合关系中，一个对象“拥有”一组其他对象，并扮演着容器或集合的角色。组件可以独立于容器存在，也可以同时连接多个容器。在 UML 图中，聚合关系使用一端是空心菱形，另一端指向组件的箭头来表示。

尽管我们在此讨论的是对象之间的关系，但请记住 UML 图表示的是**类**之间的关系。这意味着大学对象可能是由多个院系构成的，即便图中的每个实体只用一个“方框”来表示。你可以使用 UML 符号在关系两端标明数量，但如果可从上下文明确数量的话，则可以省略此类标注。

## 组合

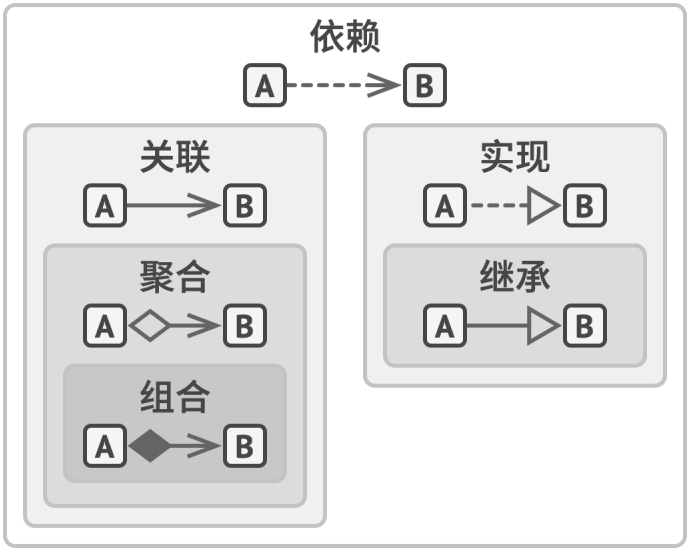

UML 图中的组合。大学由院系构成。

**组合**是一种特殊类型的聚合，其中一个对象由一个或多个其他对象实例构成。组合与其他关系的区别在于组件仅能作为容器的一部分存在。在 UML 图中，组合与聚合关系的符号相同，但箭头起始处的菱形是实心的。

注意，许多人常常在实际想说聚合和组合时使用“组合”这个术语。其中最恶名昭彰的例子是著名的“组合优于继承”原则。这并不是因为人们不清楚它们之间的差别，而是因为“组合（例如‘对象组合’）”说起来更顺口。

## 总结

现在相信你已对对象间所有的关系类型有所了解了，那么就让我们来看看它们之间的联系吧，希望这能解答“聚合与组合有何区别?”或“继承是不是一种依赖关系?”等问题。




对象和类之间的关系：从弱到强。

- **依赖**：对类 B 进行修改会影响到类 A 。
- **关联**：对象 A 知道对象 B。类 A 依赖于类 B。

- **聚合**：对象 A 知道对象 B 且由 B 构成。类 A 依赖于类 B。
- **组合**：对象 A 知道对象 B、由 B 构成而且管理着 B 的生命周期。类 A 依赖于类 B。
- **实现**：类 A 定义的方法由接口 B 声明。对象 A 可被视为对象 B。类 A 依赖于类 B。
- **继承**：类 A 继承类 B 的接口和实现，但是可以对其进行扩展。对象 A 可被视为对象 B。类 A 依赖于类 B。

# 设计模式简介

# 什么是设计模式?

**设计模式**是软件设计中常见问题的典型解决方案。它们就像能根据需求进行调整的预制蓝图，可用于解决代码中反复出现的设计问题。

设计模式与方法或库的使用方式不同，你很难直接在自己的程序中套用某个设计模式。模式并不是一段特定的代码，而是解决特定问题的一般性概念。你可以根据模式来实现符合自己程序实际所需的解决方案。

人们常常会混淆模式和算法，因为两者在概念上都是已知特定问题的典型解决方案。但算法总是明确定义达成特定目标所需的一系列步骤，而模式则是对解决方案的更高层次描述。同一模式在两个不同程序中的实现代码可能会不一样。

算法更像是菜谱：提供达成目标的明确步骤。而模式更像是蓝图：你可以看到最终的结果和模式的功能，但需要自己确定实现步骤。

## 模式包含哪些内容?

大部分模式的描述都会遵循特定的形式，以便在不同情况下使用。模式的描述通常会包括以下部分：

- **意图**部分简要地描述问题和解决方案。
- **动机**部分进一步解释问题并说明模式会如何提供解决方案。
- **结构**部分展示模式的各个部分和它们之间的关系。
- **在不同语言中的实现**提供流行编程语言的代码，让读者更好地理解模式背后的思想。

  部分模式介绍中还列出了其他的一些实用细节，例如模式的适用性、实现步骤以及与其他模式的关系。

## 模式的分类

不同设计模式在其复杂程度、细节层次以及在整个系统中的应用范围等方面各不相同。我喜欢将其比作道路的建造：如果你希望让十字路口更加安全，那么可以安装一些交通信号灯，或者修建有行人地下通道的多层互通式立交桥。

最基础的、底层的模式通常被称为惯用技巧。这类模式一般只能在一种编程语言中使用。

最通用的、高层的模式是架构模式。开发者可以在任何编程语言中使用这类模式。与其他模式不同，它们可用于整个应用程序的架构设计。

此外，所有模式可以根据其意图或目的来分类。本书覆盖了三种主要的模式类别：

- **创建型模式**提供创建对象的机制，增加已有代码的灵活性和可复用性。
- **结构型模式**介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。
- **行为模式**负责对象间的高效沟通和职责委派。

## 谁发明了设计模式?

这是一个很好的问题，但也有点不太准确。设计模式并不是晦涩的、复杂的概念——事实恰恰相反。模式是面向对象设计中常见问题的典型解决方案。同样的解决方案在各种项目中得到了反复使用，所以最终有人给它们起了名字，并对其进行了详细描述。这基本上就是模式被发现的历程了。

模式的概念是由克里斯托佛·亚历山大在其著作《建筑模式语言[1]》中首次提出的。本书介绍了城市设计的“语言”，而该语言的基本单元就是模式。它们可以描述窗户应该在多高、一座建筑应该有多少层以及一片街区应该有多大面积的植被等等。

1. 《建筑模式语言》: **https://refactoringguru.cn/pattern-language-book**

埃里希·伽玛、约翰·弗利赛德斯、拉尔夫·约翰逊和理查德·赫尔姆这四位作者接受了模式的概念。1994 年，他们出版了《设计模式：可复用面向对象软件的基础[1]》一书，将设计模式的概念应用到程序开发领域中。该书提供了 23 个模式来解决面向对象程序设计中的各种问题，很快便成为了畅销书。由于书名太长，人们将其简称为“四人组（Gang of Four，GoF）的书”，并且很快进一步简化为“GoF 的书”。

此后，人们又发现了几十种面向对象的模式。“模式方法”开始在其他程序开发领域中流行起来。如今，人们还在面向对象设计领域之外提出了许多其他的模式。

1. 《设计模式：可复用面向对象软件的基础》：**https://refactoringguru.cn/gof-book**

# 为什么以及如何学习设计模式?

或许你已从事程序开发工作多年，却完全不知道单例模式是什么。很多人都是这样。即便如此，你可能也在不自知的情况下已经使用过一些设计模式了。所以为什么不花些时间来更进一步学习它们呢?

- 设计模式是针对软件设计中常见问题的工具箱，其中的工具就是各种**经过实践验证的解决方案**。即使你从未遇到过这些问题，了解模式仍然非常有用，因为它能指导你如何使用面向对象的设计原则来解决各种问题。

- 设计模式定义了一种让你和团队成员能够更高效沟通的通用语言。你只需说“哦，这里用单例就可以了”，所有人都会理解这条建议背后的想法。只要知晓模式及其名称，你就无需解释什么是单例。

# 软件设计原则

# 优秀设计的特征

在开始学习实际的模式前，让我们来看看软件架构的设计过程，了解一下需要达成目标与需要尽量避免的陷阱。

## ♻ 代码复用

无论是开发何种软件产品，成本和时间都最重要的两个维度。较短的开发时间意味着可比竞争对手更早进入市场；较低的开发成本意味着能够留出更多营销资金，因此能更广泛地覆盖潜在客户。

**代码复用**是减少开发成本时最常用的方式之一。其意图非常明显：与其反复从头开发，不如在新对象中重用已有代码。

这个想法表面看起来很棒，但实际上要让已有代码在全新的上下文中工作，通常还是需要付出额外努力的。组件间紧密的耦合、对具体类而非接口的依赖和硬编码的行为都会降低代码的灵活性，使得复用这些代码变得更加困难。

使用设计模式是增加软件组件灵活性并使其易于复用的方式之一。但是有时，这也会让组件变得更加复杂。设计模式创

始人之一的埃里希·伽玛[1]，在谈到代码复用中设计模式的角色时说：

> 我觉得复用有三个层次。在最底层，你可以复用类：类库、容器，也许还有一些类的“团体（例如容器和迭代器）”。
>
> 框架位于最高层。它们确实能帮助你精简自己的设计，可以用于明确解决问题所需的抽象概念，然后用类来表示这些概念并定义其关系。例如，JUnit 是一个小型框架，也是框架的“Hello, world”，其中定义了 `Test`、`TestCase` 和 `TestSuite` 这几个类及其关系。
>
> 框架通常比单个类的颗粒度要大。你可以通过在某处构建子类来与框架建立联系。这些子类信奉“别给我们打电话，我们会给你打电话的。”这句所谓的好莱坞原则。框架让你可以自定义行为，并会在需要完成工作时告知你。这和 JUnit 一样，对吧？当它希望执行测试时就会告诉你，但其他的一切都仅会在框架中发生。
>
> 还有一个中间层次。这也是我认识中的模式所处位置。设计模式比框架更小且更抽象。它们实际上是对一组类的关系及其互动方式的描述。当你从类转向模式，并最终到达框架的过程中，复用程度会不断增加。

1. 埃里希·伽玛谈灵活性和代码复用：**https://refactoringguru.cn/gamma-interview**

> 中间层次的优点在于模式提供的复用方式要比框架的风险小。创建框架是一项投入重大且风险很高的工作。模式则让你能独立于具体代码来复用设计思想和理念。

# 扩展性

**变化**是程序员生命中唯一不变的事情。

- 你在 Windows 平台上发布了一款游戏，但现在人们想要 macOS 的版本。
- 你创建了一个使用方形按钮的 GUI 框架，但几个月后圆形按钮开始流行起来。
- 你设计了一款优秀的电子商务网站构架，但仅仅几个月后，客户就要求新增接受电话订单的功能。

每位软件开发者都经历过许多相似的故事，导致它们发生的原因也不少。

首先，我们在开始着手解决问题后才能更好地理解问题。通常在完成了第一版的程序后，你就做好了从头开始重写代码的准备，因为现在你已经能在很多方面更好地理解问题了，同时在专业水平上也有所提高，所以之前的代码现在看上去可能会显得很糟糕。

其次可能是在你掌控之外的某些事情发生了变化。这也是导致许多开发团队转变最初想法的原因。每位在网络应用中使用 Flash 的开发者都必须重新开发或移植代码，因为不断地有浏览器停止对 Flash 格式的支持。

第三个原因是需求的改变。你的客户之前对当前版本的程序感到满意，但是现在希望对程序进行 11 个“小小”的改动，使其可完成原始计划阶段中完全没有提到的功能。

> 这也有好的一面：如果有人要求你对程序进行修改，至少说明还有人关心它。

因此在设计程序架构时，所有有经验的开发者会尽量选择支持未来任何可能变更的方式。

# 设计原则

什么是优秀的软件设计？如何对其进行评估？你需要遵循哪些实践方式才能实现这样的方式？如何让你的架构灵活、稳定且易于理解？

这些都是很好的问题。但不幸的是，根据应用类型的不同，这些问题的答案也不尽相同。不过对于你的项目来说，有几个通用的软件设计原则可能会对解决这些问题有所帮助。本书中列出的绝大部分设计模式都是基于这些原则的。

# 封装变化的内容

> 找到程序中的变化内容并将其与不变的内容区分开。

该原则的主要目的是将变更造成的影响最小化。

假设你的程序是一艘船，变更就是徘徊在水下的可怕水雷。如果船撞上水雷就会沉没。

了解到这些情况后，你可将船体分隔为独立的隔间，并对其进行安全的密封，以使得任何损坏都会被限制在隔间范围内。现在，即使船撞上水雷也不会沉没了。

你可用同样的方式将程序的变化部分放入独立的模块中，保护其他代码不受负面影响。最终，你只需花较少时间就能让程序恢复正常工作，或是实现并测试修改的内容。你在修改程序上所花的时间越少，就会有更多时间来实现功能。

## 方法层面的封装

假如你正在开发一个电子商务网站。代码中某处有一个 getOrderTotal 获取订单总额 方法，用于计算订单的总价（包括税金在内）。

我们预计在未来可能会修改与税金相关的代码。税率会根据客户居住的国家/地区、州/省甚至城市而有所不同；而且一段时间后，实际的计算公式可能会由于新的法律或规定而修改。因此，你将需要经常性地修改 `getOrderTotal` 方法。不过仔细看看方法名称，连它都在暗示其不关心税金是**如何**计算出来的。

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // 美国营业税
  else if (order.country == "EU"):
    total += total * 0.20 // 欧洲增值税

  return total
```

**修改前：**税率计算代码和方法的其他代码混杂在一起。

你可以将计算税金的逻辑抽取到一个单独的方法中，并对原始方法隐藏该逻辑。

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity
```

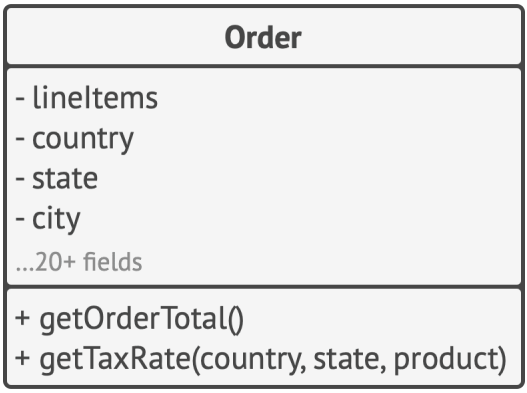
```
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10  method getTaxRate(country) is
11    if (country == "US")
12      return 0.07 // 美国营业税
13    else if (country == "EU")
14      return 0.20 // 欧洲增值税
15    else
16      return 0
```

**修改后：**你可通过调用指定方法获取税率。

这样税率相关的修改就被隔离在单个方法内了。此外，如果税率计算逻辑变得过于复杂，你也能更方便地将其移动到独立的类中。

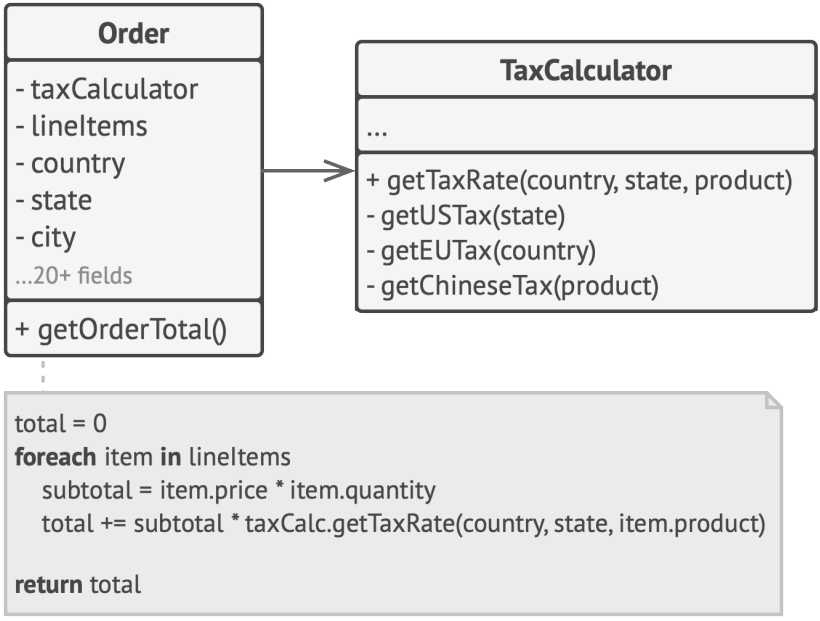
## 类层面的封装

一段时间后，你可能会在一个以前完成简单工作的方法中添加越来越多的职责。新增行为通常还会带来助手成员变量和方法，最终使得包含接纳它们的类的主要职责变得模糊。将所有这些内容抽取到一个新类中会让程序更加清晰和简洁。




**修改前：**在 订单 Order 类中计算税金。

订单 类的对象将所有与税金相关的工作委派给一个专门负责的特殊对象。




**修改后：**对订单类隐藏税金计算。

# 面向接口进行开发，而不是面向实现

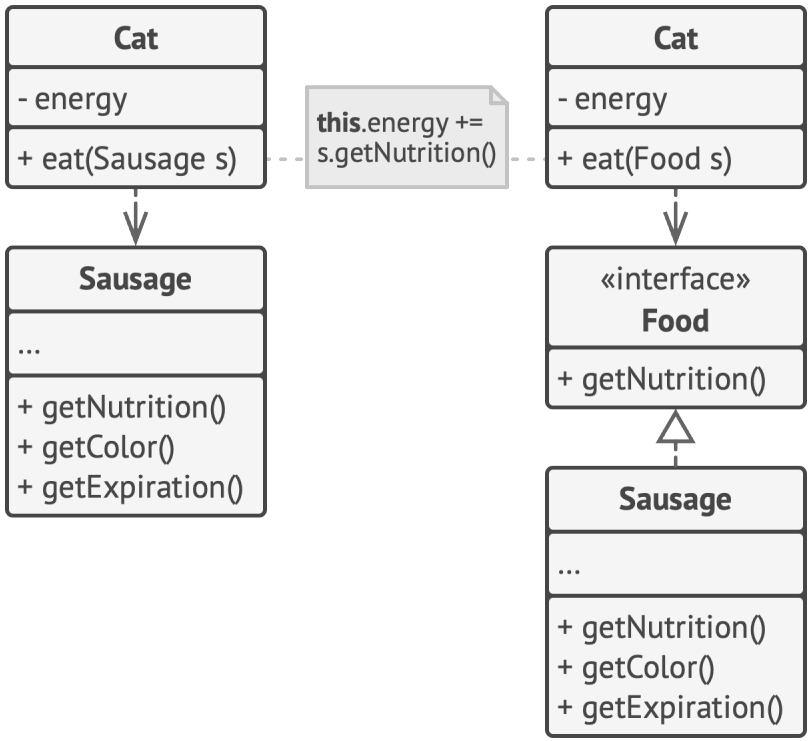
> 面向接口进行开发，而不是面向实现；依赖于抽象类型，而不是具体类。

如果无需修改已有代码就能轻松对类进行扩展，那就可以说这样的设计是灵活的。让我们再来看一个关于猫的例子，看看这个说法是否正确：一只可以吃任何食物的 `猫 Cat` 要比只吃香肠的猫更加灵活。无论如何你都可给第一只猫喂香肠，因为香肠是“任何食物”的一个子集；当然，你也可以喂这只猫任何食物。

当你需要两个类进行合作时，可以让其中一个类依赖于另一个类。实话实说，刚入行时我自己也常常这么做。但是，你可用另外一种更灵活的方式来设置对象之间的合作关系。

1. 确定一个对象对另一对象的确切需求：它需执行哪些方法？
2. 在一个新的接口或抽象类中描述这些方法。
3. 让被依赖的类实现该接口。

4. 现在让有需求的类依赖于这个接口，而不依赖于具体的类。你仍可与原始类中的对象进行互动，但现在其连接将会灵活得多。

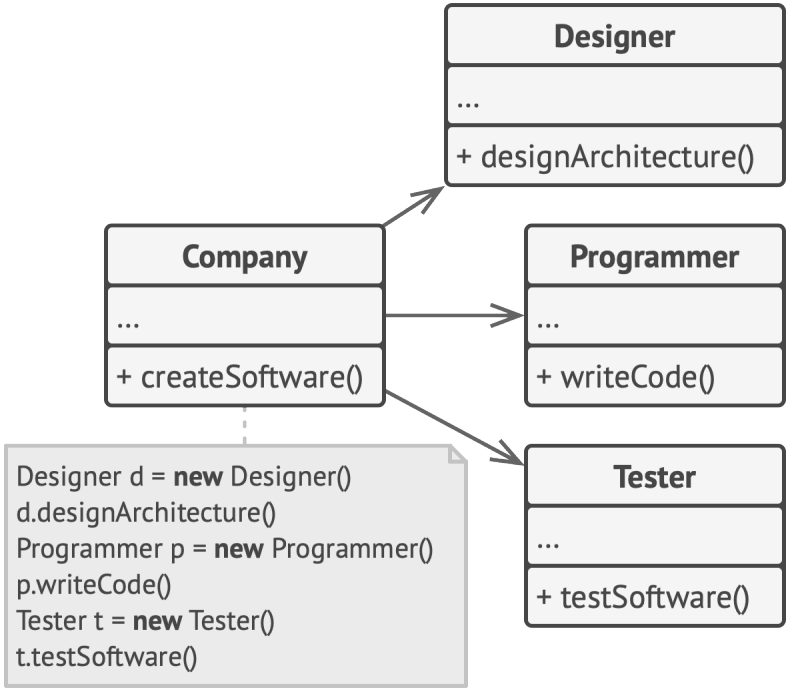



抽取接口前后的对比。右侧的代码要比左侧更加灵活，但也更加复杂。

完成修改后，你很可能没法马上看到任何好处；相反，代码会变得比以前更加复杂。但如果你觉得这里可以是个不错的额外功能扩展点，或者其他使用这些代码的用户希望在此进行扩展的话，那就马上动手去做吧。

## 示例

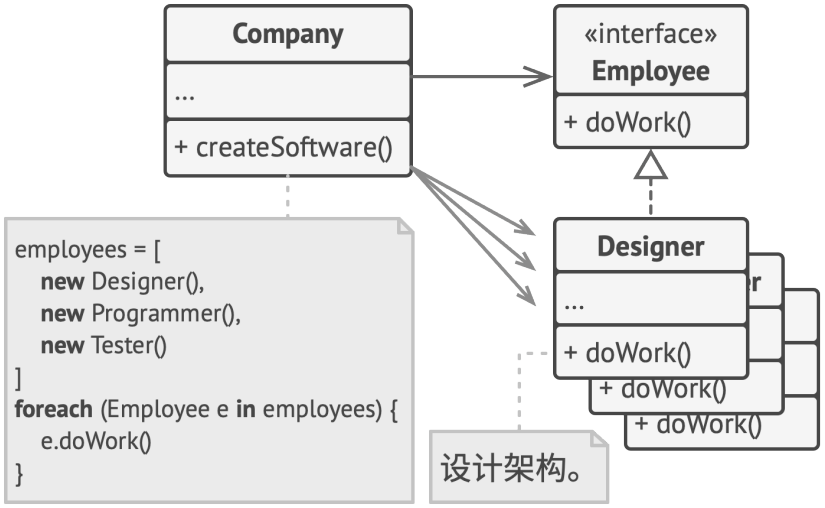
让我们来看另一个例子，它说明了通过接口与对象交互要比依赖于其具体类的好处更多。假设你正在开发一款软件开发公司模拟器，而且使用了不同的类来代表各种类型的雇员。




**修改前：**所有类都紧密耦合。

刚开始时，公司 Company 类与具体雇员类紧密耦合。尽管各个雇员的实现不尽相同，但我们还是可以归纳出几个与工作相关的方法，并且将其抽取为所有雇员的通用接口。

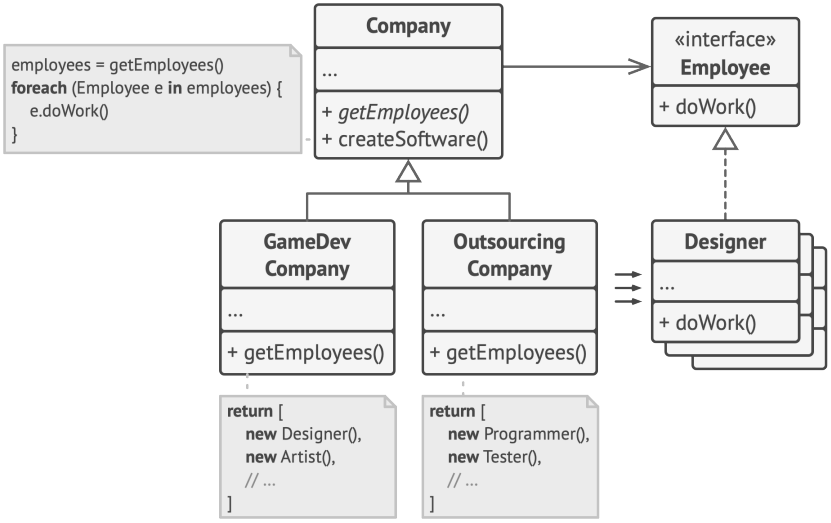
此后，我们可在 公司 类内应用多态机制，通过 雇员 Employee 接口来处理各类雇员对象。




**优化：**多态机制能帮助我们简化代码，但 公司 类的其他部分仍然依赖于具体的雇员类。

公司 类仍与雇员类相耦合，这很糟糕，因为如果引入包含其他类型雇员的公司类型的话，我们就需要重写绝大部分的 公司 类了，不能复用其代码。

为了解决这个问题，我们可以声明一个抽象方法来获取雇员。每个具体公司都将以不同方式实现该方法，从而创建自己所需的雇员。




**修改后：**公司类的主要方法独立于具体的雇员类。雇员对象将在具体公司子类中创建。

修改后的公司类将独立于各种雇员类。现在你可以对该类进行扩展，并在复用部分公司基类的情况下引入新的公司和雇员类型。对公司基类进行扩展时无需修改任何依赖于基类的已有代码。

顺便提一句，你刚刚目睹的就是设计模式的应用！这就是**工厂方法**模式的一个示例。不要担心，稍后我们会对其进行详细讨论。

# 组合优于继承

继承可能是类之间最明显、最简便的代码复用方式。如果你有两个代码相同的类，就可以为它们创建一个通用的基类，然后将相似的代码移动到其中。轻而易举！

不过，继承这件事通常只有在程序中已包含大量类，且修改任何东西都非常困难时才会引起关注。下面就是此类问题的清单。

- **子类不能减少超类的接口**。你必须实现父类中所有的抽象方法，即使它们没什么用。

- **在重写方法时，你需要确保新行为与其基类中的版本兼容**。这一点很重要，因为子类的所有对象都可能被传递给以超类对象为参数的任何代码，相信你不会希望这些代码崩溃的。

- **继承打破了超类的封装**，因为子类拥有访问父类内部详细内容的权限。此外还可能会有相反的情况出现，那就是程序员为了进一步扩展的方便而让超类知晓子类的内部详细内容。

- **子类与超类紧密耦合**。超类中的任何修改都可能会破坏子类的功能。

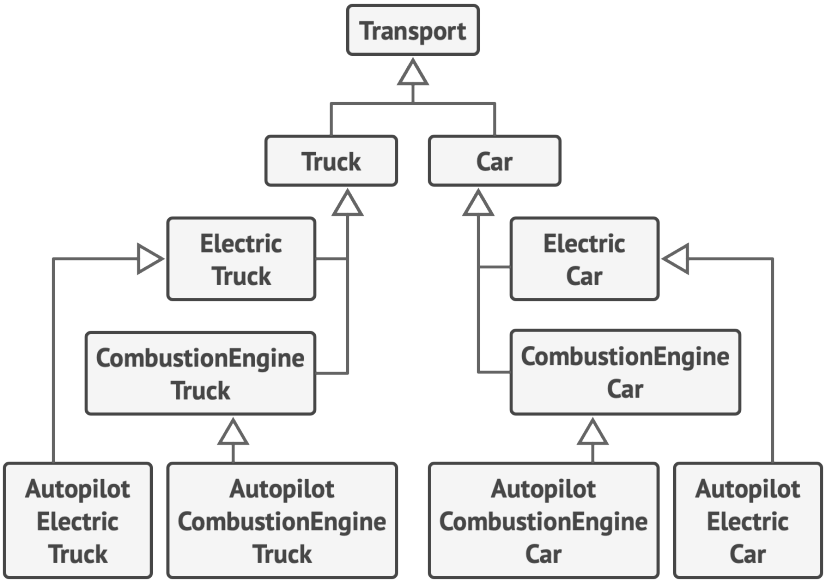
- **通过继承复用代码可能导致平行继承体系的产生**。继承通常仅发生在一个维度中。只要出现了两个以上的维度，你就必须创建数量巨大的类组合，从而使类层次结构膨胀到不可思议的程度。

  组合是代替继承的一种方法。继承代表类之间的“是”关系（汽车**是**交通工具），而组合则代表“有”关系（汽车**有**一个引擎）。

  必须一提的是，这个原则也能应用于聚合（一种更松弛的组合变体，一个对象可引用另一个对象，但并不管理其生命周期）。例如：一辆汽车上有司机，但是司机也可能会使用另一辆汽车，或者选择步行而不使用汽车。

## 示例

假如你需要为汽车制造商创建一个目录程序。该公司同时生产 汽车 Car 和 卡车 Truck，车辆可能是 电动车 Electric 或 汽油车 Combustion；所有车型都配备了手动控制 manual control 或 自动驾驶 Autopilot 功能。

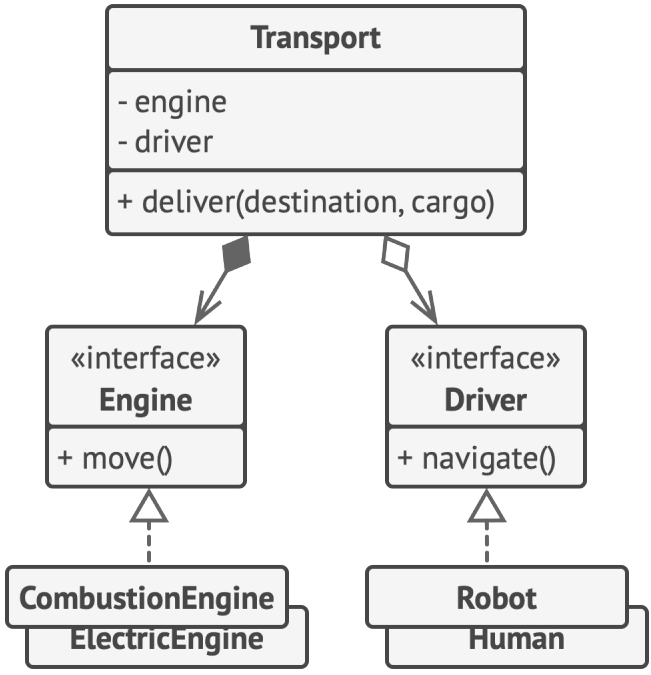



**继承：**在多个维度上扩展一个类（汽车类型 × 引擎类型 × 驾驶类型）可能会导致子类组合的数量爆炸。

正如你所看到的，每个额外参数都将使子类数量倍增。子类中将有大量的重复代码，因为子类不能同时继承两个类。

你可以使用组合来解决这个问题。汽车对象可将行为委派给其他对象，而不是自行实现。

还有一个好处是你可以在运行时对行为进行替换。例如，你可以通过重新为汽车对象分配一个不同的引擎对象来替换已连接至汽车的引擎。




**组合：**将不同“维度”的功能抽取到各自的类层次结构中。

上述类的结构类似于我们稍后将在本书中讨论的策略模式。

# SOLID 原则

现在相信你已经了解了基础的设计模式，那就让我们来看看名为 SOLID 的五条原则吧。这五条原则是在罗伯特·马丁的著作《敏捷软件开发：原则、模式与实践[1]》中首次提出的。

SOLID 是让软件设计更易于理解、更加灵活和更易于维护的五个原则的简称。

与生活中所有事情一样，盲目遵守这些原则可能会弊大于利。在程序架构中应用这些原则可能会使其变得过于复杂。我对于是否真的有能够同时应用所有这五条原则的成功软件产品表示怀疑。有原则是件好事，但是也要时刻从实用的角度来考量，不要把这里的每句话当作放之四海皆准的教条。

1. 《敏捷软件开发：原则、模式与实践》：**https://refactoringguru.cn/principles-book**

# S 单一职责原则 ingle Responsibility Principle

> 修改一个类的原因只能有一个。

尽量让每个类只负责软件中的一个功能，并将该功能完全封装（你也可称之为隐藏）在该类中。

这条原则的主要目的是减少复杂度。你不需要费尽心机地去构思如何仅用 200 行代码来实现复杂设计，实际上完全可以使用十几个清晰的方法。

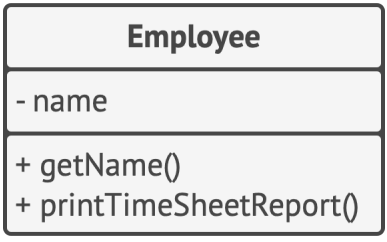
当程序规模不断扩大、变更不断增加后，真实问题才会逐渐显现出来。到了某个时候，类会变得过于庞大，以至于你无法记住其细节。查找代码将变得非常缓慢，你必须浏览整个类，甚至整个程序才能找到需要的东西。程序中实体的数量会让你的大脑堆栈过载，你会感觉自己对代码失去了控制。

还有一点：如果类负责的东西太多，那么当其中任何一件事发生改变时，你都必须对类进行修改。而在进行修改时，你就有可能改动类中自己并不希望改动的部分。

如果你开始感觉在同时关注程序特定方面的内容时有些困难的话，请回忆单一职责原则并考虑现在是否应将某些类分割为几个部分。

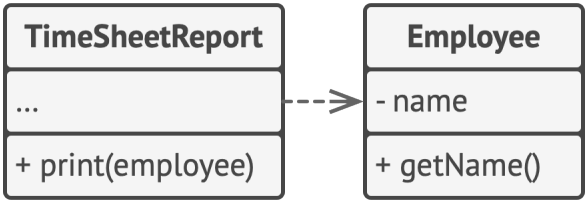
## 示例

我们有几个理由来对 雇员 Employee 类进行修改。第一个理由与该类的主要工作（管理雇员数据）有关。但还有另一个理由：时间表报告的格式可能会随着时间而改变，从而使你需要对类中的代码进行修改。




**修改前：**类中包含多个不同的行为。

解决该问题的方法是将与打印时间表报告相关的行为移动到一个单独的类中。这个改变让你能将其他与报告相关的内容移动到一个新的类中。




**修改后：**额外行为有了它们自己的类。

# O 开闭原则 pen/closed Principle

> 对于扩展，类应该是“开放”的；对于修改，类则应是“封闭”的。

本原则的主要理念是在实现新功能时能保持已有代码不变。

如果你可以对一个类进行扩展，可以创建它的子类并对其做任何事情（如新增方法或成员变量、重写基类行为等），那么它就是开放的。有些编程语言允许你通过特殊关键字（例如 `final`）来限制对于类的进一步扩展，这样类就不再是“开放”的了。如果某个类已做好了充分的准备并可供其他类使用的话（即其接口已明确定义且以后不会修改），那么该类就是封闭（你可以称之为完整）的。

我第一次知道这条原则时曾感到困惑，因为开和闭这两个字听上去是互斥的。但根据这条原则，一个类可以同时是“开放（对于扩展而言）”和“封闭（对于修改而言）”的。

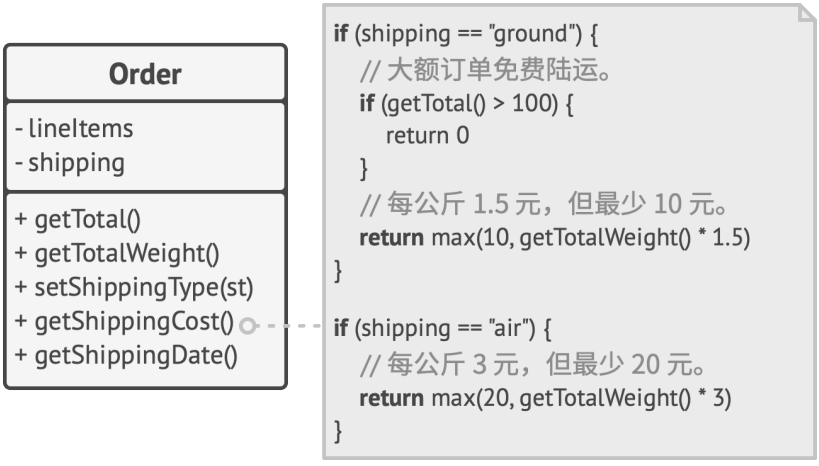
如果一个类已经完成开发、测试和审核工作，而且属于某个框架或者可被其他类的代码直接使用的话，对其代码进行修改就是有风险的。你可以创建一个子类并重写原始类的部分内容以完成不同的行为，而不是直接对原始类的代码进行修

改。这样你既可以达成自己的目标，但同时又无需修改已有的原始类客户端。

这条原则并不能应用于所有对类进行的修改中。如果你发现类中存在缺陷，直接对其进行修复即可，不要为它创建子类。子类不应该对其父类的问题负责。

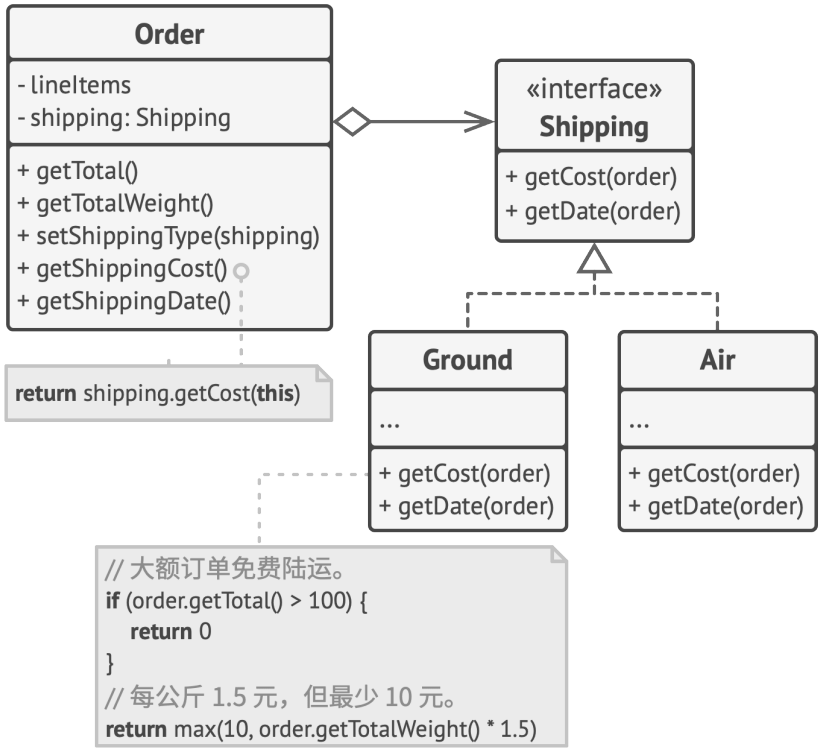
## 示例

你的电子商务程序中包含一个计算运输费用的 订单 Order 类，该类中所有运输方法都以硬编码的方式实现。如果你需要添加一个新的运输方式，那就必须承担对 订单 类造成破坏的可能风险来对其进行修改。




**修改前：**在程序中添加新的运输方式时，你必须对 订单 类进行修改。

你可以通过应用策略模式来解决这个问题。首先将运输方法抽取到拥有同样接口的不同类中。




**修改后：**添加新的运输方式不需要修改已有的类。

现在，当需要实现一个新的运输方式时，你可以通过扩展运输方式 Shipping 接口来新建一个类，无需修改任何订单类的代码。当用户在 UI 中选择这种运输方式时，订单类客户端代码会将订单链接到新类的运输方式对象。

此外，根据单一职责原则，这个解决方案能够让你将运输时间的计算代码移动到与其相关度更高的类中。

# L 里氏替换原则 iskov Substitution Principle[1]

> 当你扩展一个类时，记住你应该要能在不修改客户端代码的情况下将子类的对象作为父类对象进行传递。

这意味着子类必须保持与父类行为的兼容。在重写一个方法时，你要对基类行为进行扩展，而不是将其完全替换。

替换原则是用于预测子类是否与代码兼容，以及是否能与其超类对象协作的一组检查。这一概念在开发程序库和框架时非常重要，因为其中的类将会在他人的代码中使用——你是无法直接访问和修改这些代码的。

与有着多种解释方式的其他设计模式不同，替代原则包含一组对子类（特别是其方法）的形式要求。让我们来仔细看看这些要求。

- **子类方法的参数类型必须与其超类的参数类型相匹配或更加抽象。**听上去让人迷惑？让我们来看一个例子。
  - 假设某个类有个方法用于给猫咪喂食：`feed(Cat c)`。客户端代码总是会将“猫（cat）”对象传递给该方法。

1. 该原则由芭芭拉·里斯科夫命名，她于 1987 年在其著作《数据抽象化与层次结构》中对其进行了定义：**https://refactoringguru.cn/liskov/dah**

- **好的方式：**假如你创建了一个子类并重写了前面的方法，使其能够给任何“动物（animal，即‘猫’的超类）”喂食：`feed(Animal c)`。如果现在你将一个子类对象而非超类对象传递给客户端代码，程序仍将正常工作。该方法可用于给任何动物喂食，因此它仍然可以用于给传递给客户端的任何“猫”喂食。

- **不好的方式：**你创建了另一个子类且限制喂食方法仅接受“孟加拉猫（BengalCat，一个‘猫’的子类）”：`feed(BengalCat c)`。如果你用它来替代链接在某个对象中的原始类，客户端中会发生什么呢？由于该方法只能对特殊种类的猫进行喂食，因此无法为传递给客户端的普通猫提供服务，从而将破坏所有相关的功能。

- **子类方法的返回值类型必须与超类方法的返回值类型或是其子类别相匹配。**正如你所看到的，对于返回值类型的要求与对于参数类型的要求相反。

  - 假如你的一个类中有一个方法`buyCat(): Cat`。客户端代码执行该方法后的预期返回结果是任意类型的“猫”。

  - **好的方式：**子类将该方法重写为：`buyCat(): BengalCat`。客户端将获得一只“孟加拉猫”，自然它也是一只“猫”，因此一切正常。

  - **不好的方式：**子类将该方法重写为：`buyCat(): Animal`。现在客户端代码将会出错，因为它获得的是自己未知的动物种类（短吻鳄？熊?），不适用于为一只“猫”而设计的结构。

  编程语言世界中的另一个反例是动态类型：基础方法返回一个字符串，但重写后的方法则返回一个数字。

- **子类中的方法不应抛出基础方法预期之外的异常类型。**换句话说，异常类型必须与基础方法能抛出的异常或是其子类别相匹配。这条规则源于一个事实：客户端代码的`try-catch`代码块针对的是基础方法可能抛出的异常类型。因此，预期之外的异常可能会穿透客户端的防御代码，从而使整个应用崩溃。

  > 对于绝大部分现代编程语言，特别是静态类型的编程语言（Java 和 C# 等等），这些规则已内置于其中。如果违反了这些规则，你将无法对程序进行编译。

- **子类不应该加强其前置条件。**例如，基类的方法有一个`int`类型的参数。如果子类重写该方法时，要求传递给该方法的参数值必须为正数（如果该值为负则抛出异常），这就是加强了前置条件。客户端代码之前将负数传递给该方法时程序能够正常运行，但现在使用子类的对象时会使程序出错。

- **子类不能削弱其后置条件。**假如你的某个类中有个方法需要使用数据库，该方法应该在接收到返回值后关闭所有活跃的数据库连接。

  你创建了一个子类并对其进行了修改，使得数据库保持连接以便重用。但客户端可能对你的意图一无所知。由于它认为该方法会关闭所有的连接，因此可能会在调用该方法后就马上关闭程序，使得无用的数据库连接对系统造成“污染”。

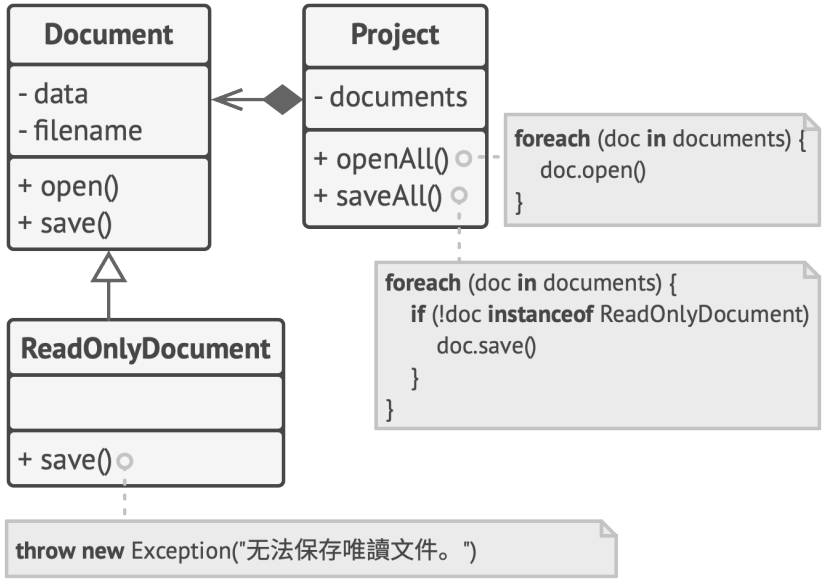
- **超类的不变量必须保留。**这很可能是所有规则中最不“形式”的一条。不变量是让对象有意义的条件。例如，猫的不变量是有四条腿、一条尾巴和能够喵喵叫等。不变量让人疑惑的地方在于它们既可通过接口契约或方法内的一组断言来明确定义，又可暗含在特定的单元测试和客户代码预期中。

  不变量的规则是最容易违反的，因为你可能会误解或没有意识到一个复杂类中的所有不变量。因此，扩展一个类的最安全做法是引入新的成员变量和方法，而不要去招惹超类中已有的成员。当然在实际中，这并非总是可行。

- **子类不能修改超类中私有成员变量的值。**什么？这难道可能吗？原来有些编程语言允许通过反射机制来访问类的私有成员。还有一些语言（Python 和 JavaScript）没有对私有成员进行任何保护。

# 示例

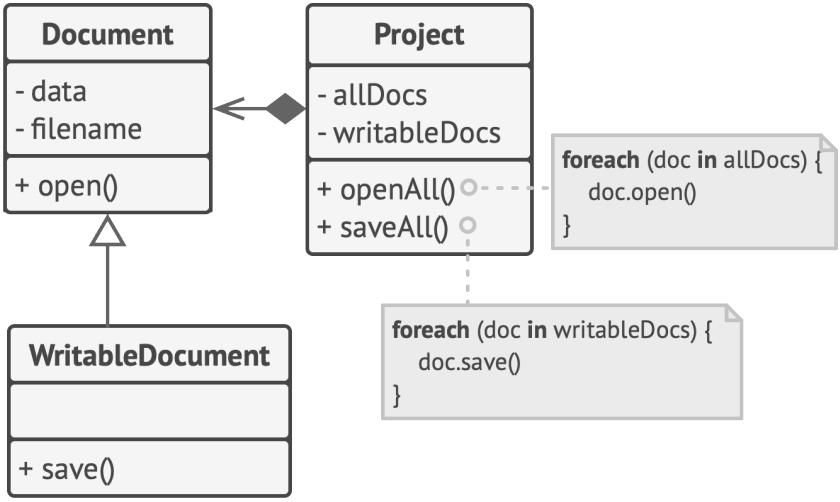
让我们来看看一个违反替换原则的文档类层次结构例子。




**修改前：**只读文件中的保存行为没有任何意义，因此子类试图在重写后的方法中重置基础行为来解决这个问题。

只读文件 ReadOnlyDocuments 子类中的 save 保存 方法会在被调用时抛出一个异常。基础方法则没有这个限制。这意味着如果我们没有在保存前检查文档类型，客户端代码将会出错。

代码也将违反开闭原则，因为客户端代码将依赖于具体的文档类。如果你引入了新的文档子类，则需要修改客户端代码才能对其进行支持。




**修改后：**当把只读文档类作为层次结构中的基类后，这个问题得到了解决。

你可以通过重新设计类层次结构来解决这个问题：一个子类必须扩展其超类的行为，因此只读文档变成了层次结构中的基类。可写文件现在变成了子类，对基类进行扩展并添加了保存行为。

# I 接口隔离原则 nterface Segregation Principle

> 客户端不应被强迫依赖于其不使用的方法。

尽量缩小接口的范围，使得客户端的类不必实现其不需要的行为。

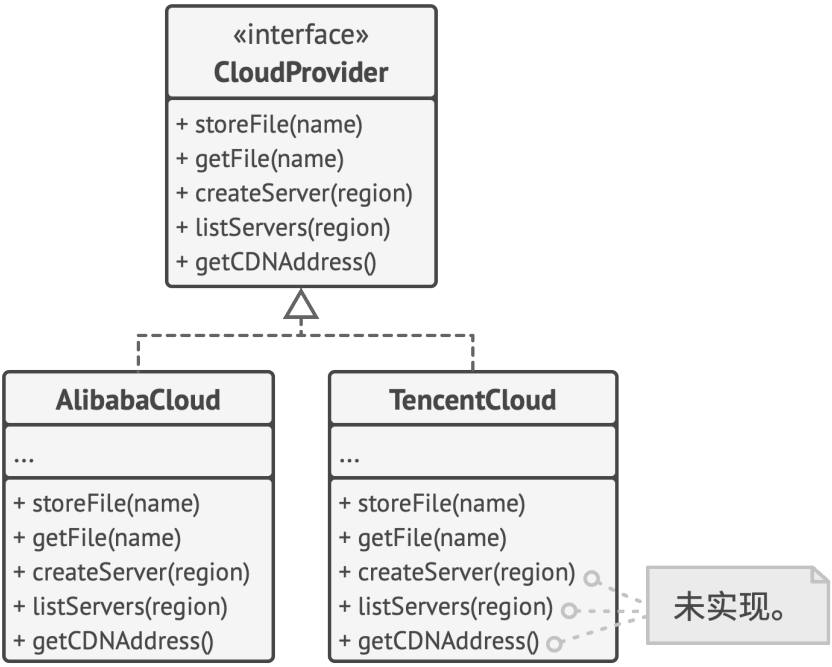
根据接口隔离原则，你必须将“臃肿”的方法拆分为多个颗粒度更小的具体方法。客户端必须仅实现其实际需要的方法。否则，对于“臃肿”接口的修改可能会导致程序出错，即使客户端根本没有使用修改后的方法。

继承只允许类拥有一个超类，但是它并不限制类可同时实现的接口的数量。因此，你不需要将大量无关的类塞进单个接口。你可将其拆分为更精细的接口，如有需要可在单个类中实现所有接口，某些类也可只实现其中的一个接口。

## 示例

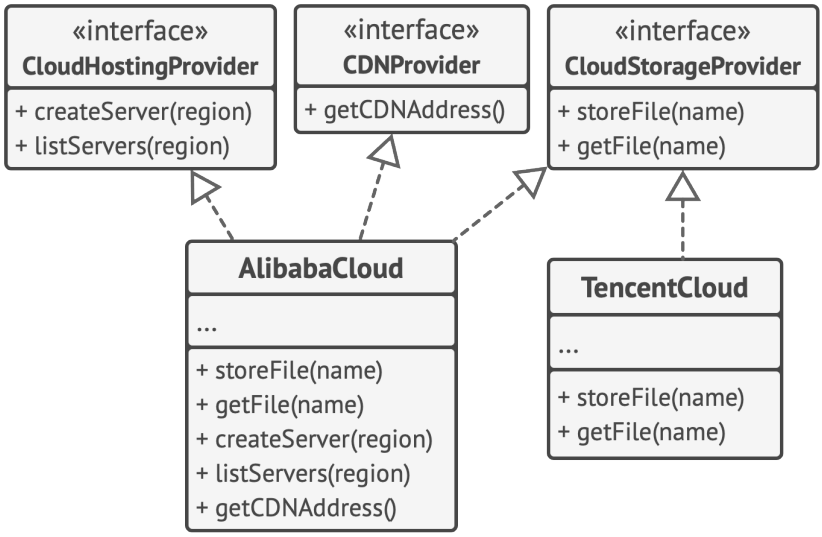
假如你创建了一个程序库，它能让程序方便地与多种云计算供应商进行整合。尽管最初版本仅支持阿里云服务，但它也覆盖了一套完整的云服务和功能。

假设所有云服务供应商都与阿里云一样提供相同种类的功能。但当你着手为其他供应商提供支持时，程序库中绝大部分的接口会显得过于宽泛。其他云服务供应商没有提供部分方法所描述的功能。




**修改前：**不是所有客户端能满足复杂接口的要求。

尽管你仍然可以去实现这些方法并放入一些桩代码，但这绝不是优良的解决方案。更好的方法是将接口拆分为多个部分。能够实现原始接口的类现在只需改为实现多个精细的接口即可。其他类则可仅实现对自己有意义的接口。




**修改后：**一个复杂的接口被拆分为一组颗粒度更小的接口。

与其他原则一样，你可能会过度使用这条原则。不要进一步划分已经非常具体的接口。记住，创建的接口越多，代码就越复杂。因此要保持平衡。

# D 依赖倒置原则 ependency Inversion Principle

> 高层次的类不应该依赖于低层次的类。两者都应该依赖于抽象接口。抽象接口不应依赖于具体实现。具体实现应该依赖于抽象接口。

通常在设计软件时，你可以辨别出不同层次的类。

- **低层次的类**实现基础操作（例如磁盘操作、传输网络数据和连接数据库等）。
- **高层次类**包含复杂业务逻辑以指导低层次类执行特定操作。

有时人们会先设计低层次的类，然后才会开发高层次的类。当你在新系统上开发原型产品时，这种情况很常见。由于低层次的东西还没有实现或不确定，你甚至无法确定高层次类能实现哪些功能。如果采用这种方式，业务逻辑类可能会更依赖于低层原语类。

依赖倒置原则建议改变这种依赖方式。

1. 作为初学者，你最好使用业务术语来对高层次类依赖的低层次操作接口进行描述。例如，业务逻辑应该调用名为 `openReport(file)` 的方法，而不是 `openFile(x)`、

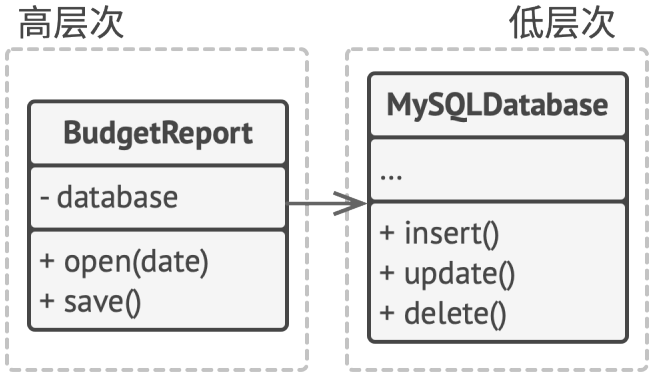
`readBytes(n)` 和 `closeFile(x)` 等一系列方法。这些接口被视为是高层次的。

2. 现在你可基于这些接口创建高层次类，而不是基于低层次的具体类。这要比原始的依赖关系灵活很多。
3. 一旦低层次的类实现了这些接口，它们将依赖于业务逻辑层，从而倒置了原始的依赖关系。

依赖倒置原则通常和开闭原则共同发挥作用：你无需修改已有类就能用不同的业务逻辑类扩展低层次的类。

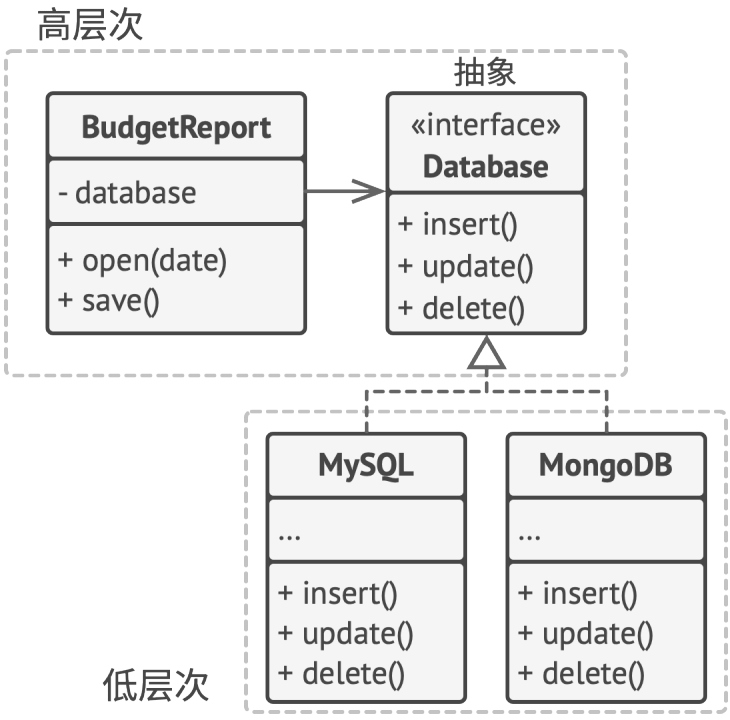
## 示例

在本例中，高层次的预算报告类（BudgetReport）使用低层次的数据库类（MySQLDatabase）来读取和保存其数据。这意味着低层次类中的任何改变（例如当数据库服务器发布新版本时）都可能会影响到高层次的类，但高层次的类不应关注数据存储的细节。




**修改前：**高层次的类依赖于低层次的类。

要解决这个问题，你可以创建一个描述读写操作的高层接口，并让报告类使用该接口代替低层次的类。然后你可以修改或扩展低层次的原始类来实现业务逻辑声明的读写接口。



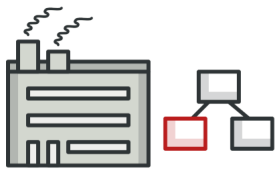

**修改后：**低层次的类依赖于高层次的抽象。

其结果是原始的依赖关系被倒置：现在低层次的类依赖于高层次的抽象。

# 设计模式目录

# 创建型模式

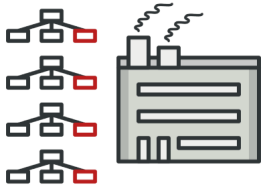
创建型模式提供了创建对象的机制，能够提升已有代码的灵活性和可复用性。



## 工厂方法

Factory Method

在父类中提供一个创建对象的接口以允许子类决定实例化对象的类型。



## 抽象工厂

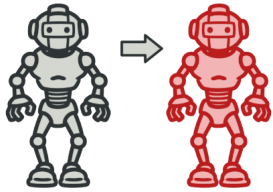
Abstract Factory

让你能创建一系列相关的对象，而无需指定其具体类。

## 生成器

Builder

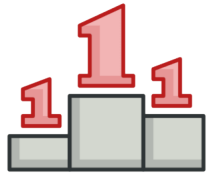
使你能够分步骤创建复杂对象。 该模式允许你使用相同的创建代码生成不同类型和形式的对象。



## 原型

Prototype

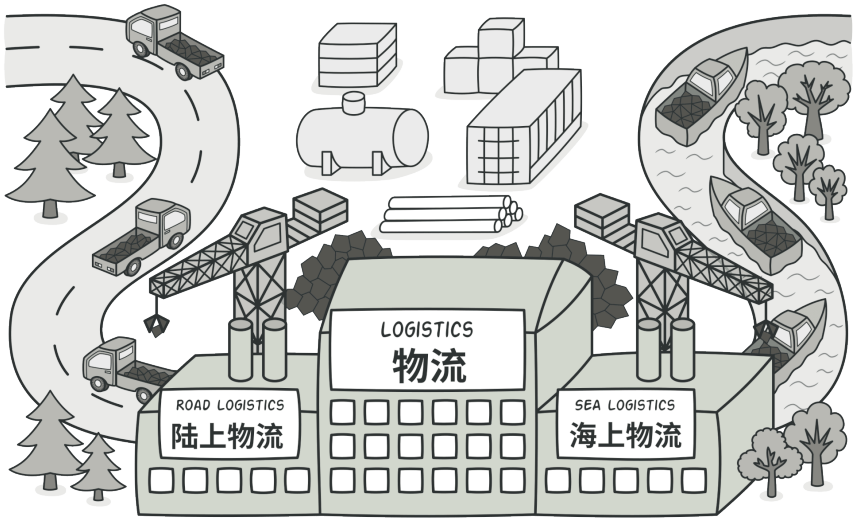
让你能够复制已有对象，而又无需使代码依赖它们所属的类。



## 单例

Singleton

让你能够保证一个类只有一个实例，并提供一个访问该实例的全局节点。




# 工厂方法

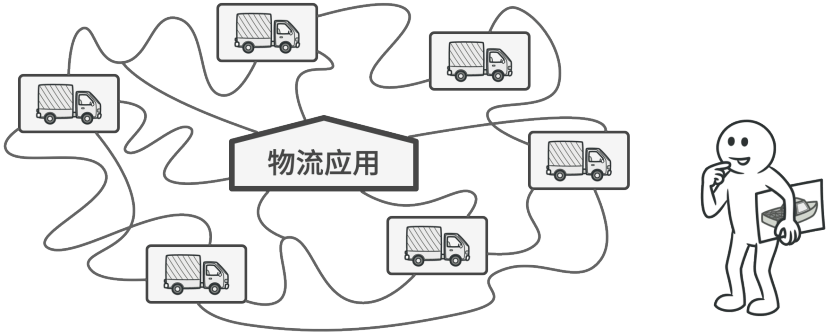
亦称：虚拟构造函数、Virtual Constructor、Factory Method

**工厂方法**是一种创建型设计模式，其在父类中提供一个创建对象的方法，允许子类决定实例化对象的类型。

# 问题

假设你正在开发一款物流管理应用。最初版本只能处理卡车运输，因此大部分代码都在位于名为 `卡车` 的类中。

一段时间后，这款应用变得极受欢迎。你每天都能收到十几次来自海运公司的请求，希望应用能够支持海上物流功能。



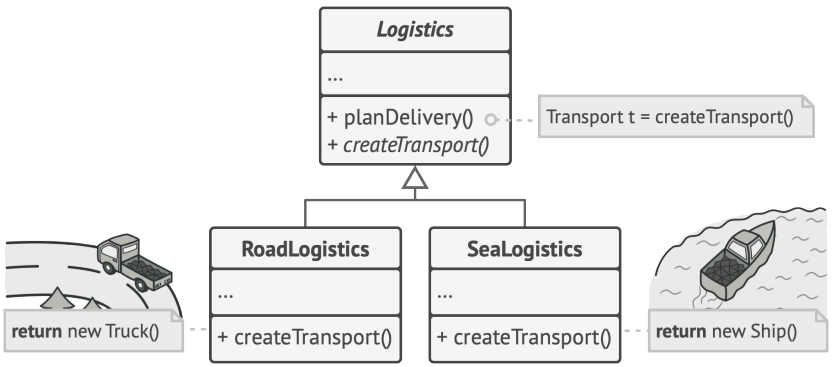

如果代码其余部分与现有类已经存在耦合关系，那么向程序中添加新类其实并没有那么容易。

这可是个好消息。但是代码问题该如何处理呢？目前，大部分代码都与 `卡车` 类相关。在程序中添加 `轮船` 类需要修改全部代码。更糟糕的是，如果你以后需要在程序中支持另外一种运输方式，很可能需要再次对这些代码进行大幅修改。

最后，你将不得不编写繁复的代码，根据不同的运输对象类，在应用中进行不同的处理。

# 😊解决方案

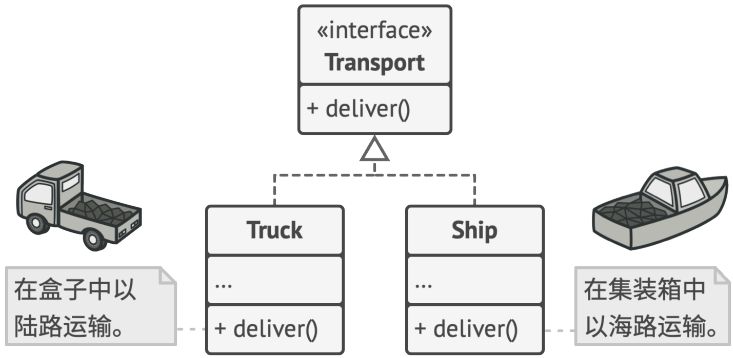
工厂方法模式建议使用特殊的工厂方法代替对于对象构造函数的直接调用（即使用 `new` 运算符）。不用担心，对象仍将通过 `new` 运算符创建，只是该运算符改在工厂方法中调用罢了。工厂方法返回的对象通常被称作“产品”。




子类可以修改工厂方法返回的对象类型。

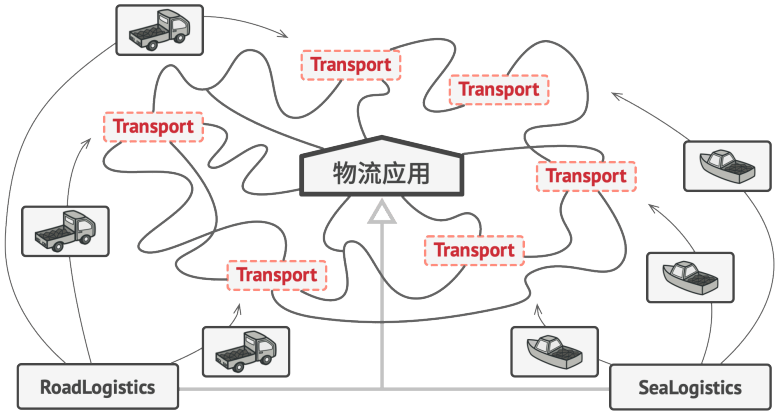
乍看之下，这种更改可能毫无意义：我们只是改变了程序中调用构造函数的位置而已。但是，仔细想一下，现在你可以在子类中重写工厂方法，从而改变其创建产品的类型。

但有一点需要注意:仅当这些产品具有共同的基类或者接口时，子类才能返回不同类型的产品，同时基类中的工厂方法还应将其返回类型声明为这一共有接口。




所有产品都必须使用同一接口。

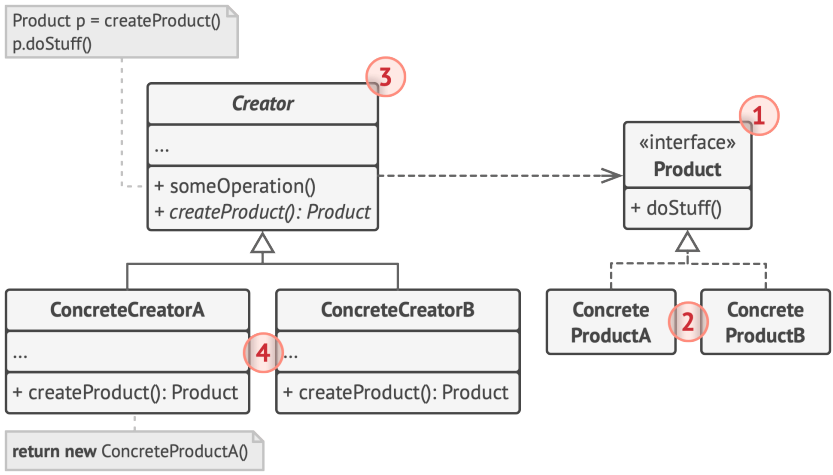
举例来说，卡车 Truck 和轮船 Ship 类都必须实现运输 Transport 接口，该接口声明了一个名为 deliver 交付的方法。每个类都将以不同的方式实现该方法：卡车走陆路交付货物，轮船走海路交付货物。陆路运输 RoadLogistics 类中的工厂方法返回卡车对象，而海路运输 SeaLogistics 类则返回轮船对象。




只要产品类实现一个共同的接口，你就可以将其对象传递给客户代码，而无需提供额外数据。

调用工厂方法的代码（通常被称为客户端代码）无需了解不同子类返回实际对象之间的差别。客户端将所有产品视为抽象的 运输 。客户端知道所有运输对象都提供 交付 方法，但是并不关心其具体实现方式。

## 结构




1. **产品**（Product）将会对接口进行声明。对于所有由创建者及其子类构建的对象，这些接口都是通用的。

2. **具体产品**（Concrete Products）是产品接口的不同实现。

3. **创建者**（Creator）类声明返回产品对象的工厂方法。该方法的返回对象类型必须与产品接口相匹配。

你可以将工厂方法声明为抽象方法，强制要求每个子类以不同方式实现该方法。或者，你也可以在基础工厂方法中返回默认产品类型。

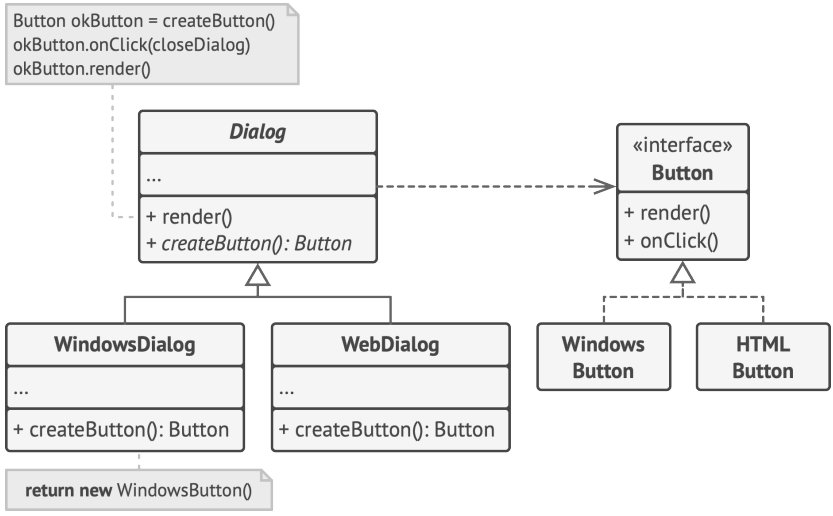
注意，尽管它的名字是创建者，但他最主要的职责并**不是**创建产品。一般来说，创建者类包含一些与产品相关的核心业务逻辑。工厂方法将这些逻辑处理从具体产品类中分离出来。打个比方，大型软件开发公司拥有程序员培训部门。但是，这些公司的主要工作还是编写代码，而非生产程序员。

4. **具体创建者**（Concrete Creators）将会重写基础工厂方法，使其返回不同类型的产品。

   注意，并不一定每次调用工厂方法都会**创建**新的实例。工厂方法也可以返回缓存、对象池或其他来源的已有对象。

# # 伪代码

以下示例演示了如何使用**工厂方法**开发跨平台 UI（用户界面）组件，并同时避免客户代码与具体 UI 类之间的耦合。




跨平台对话框示例。

基础对话框类使用不同的 UI 组件渲染窗口。在不同的操作系统下，这些组件外观或许略有不同，但其功能保持一致。Windows 系统中的按钮在 Linux 系统中仍然是按钮。

如果使用工厂方法，就不需要为每种操作系统重写对话框逻辑。如果我们声明了一个在基本对话框类中生成按钮的工厂方法，那么我们就可以创建一个对话框子类，并使其通过工厂方法返回 Windows 样式按钮。子类将继承对话框基础类的大部分代码，同时在屏幕上根据 Windows 样式渲染按钮。

如需该模式正常工作，基础对话框类必须使用抽象按钮（例如基类或接口），以便将其扩展为具体按钮。这样一来，无论对话框中使用何种类型的按钮，其代码都可以正常工作。

你可以使用此方法开发其他 UI 组件。不过，每向对话框中添加一个新的工厂方法，你就离**抽象工厂**模式更近一步。我们将在稍后谈到这个模式。

```
// 创建者类声明的工厂方法必须返回一个产品类的对象。创建者的子类通常会提供
// 该方法的实现。
class Dialog is
  // 创建者还可提供一些工厂方法的默认实现。
  abstract method createButton():Button

  // 请注意，创建者的主要职责并非是创建产品。其中通常会包含一些核心业务
  // 逻辑，这些逻辑依赖于由工厂方法返回的产品对象。子类可通过重写工厂方
  // 法并使其返回不同类型的产品来间接修改业务逻辑。
  method render() is
    // 调用工厂方法创建一个产品对象。
    Button okButton = createButton()
    // 现在使用产品。
    okButton.onClick(closeDialog)
    okButton.render()


// 具体创建者将重写工厂方法以改变其所返回的产品类型。
class WindowsDialog extends Dialog is
  method createButton():Button is
    return new WindowsButton()

class WebDialog extends Dialog is
  method createButton():Button is
    return new HTMLButton()


```

```
// 产品接口中将声明所有具体产品都必须实现的操作。
interface Button is
  method render()
  method onClick(f)

// 具体产品需提供产品接口的各种实现。
class WindowsButton implements Button is
  method render(a, b) is
    // 根据 Windows 样式渲染按钮。
  method onClick(f) is
    // 绑定本地操作系统点击事件。

class HTMLButton implements Button is
  method render(a, b) is
    // 返回一个按钮的 HTML 表述。
  method onClick(f) is
    // 绑定网络浏览器的点击事件。


class Application is
  field dialog: Dialog

  // 程序根据当前配置或环境设定选择创建者的类型。
  method initialize() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      dialog = new WindowsDialog()
    else if (config.OS == "Web") then
      dialog = new WebDialog()
    else
      throw new Exception("错误! 未知的操作系统。")
```

```
60
61    // 当前客户端代码会与具体创建者的实例进行交互，但是必须通过其基本接口
62    // 进行。只要客户端通过基本接口与创建者进行交互，你就可将任何创建者子
63    // 类传递给客户端。
64    method main() is
65      this.initialize()
66      dialog.render()
```

## 适合应用场景

**当你在编写代码的过程中，如果无法预知对象确切类别及其依赖关系时，可使用工厂方法。**

工厂方法将创建产品的代码与实际使用产品的代码分离，从而能在不影响其他代码的情况下扩展产品创建部分代码。

例如，如果需要向应用中添加一种新产品，你只需要开发新的创建者子类，然后重写其工厂方法即可。

**如果你希望用户能扩展你软件库或框架的内部组件，可使用工厂方法。**

继承可能是扩展软件库或框架默认行为的最简单方法。但是当你使用子类替代标准组件时，框架如何辨识出该子类？

解决方案是将各框架中构造组件的代码集中到单个工厂方法中，并在继承该组件之外允许任何人对该方法进行重写。

让我们看看具体是如何实现的。假设你使用开源 UI 框架编写自己的应用。你希望在应用中使用圆形按钮，但是原框架仅支持矩形按钮。你可以使用 圆形按钮 RoundButton 子类来继承标准的 按钮 Button 类。但是，你需要告诉 UI框架 UIFramework 类使用新的子类按钮代替默认按钮。

为了实现这个功能，你可以根据基础框架类开发子类 圆形按钮 UI UIWithRoundButtons，并且重写其 createButton 创建按钮 方法。基类中的该方法返回 按钮 对象，而你开发的子类返回 圆形按钮 对象。现在，你就可以使用 圆形按钮 UI 类代替 UI框架 类。就是这么简单！

**如果你希望复用现有对象来节省系统资源，而不是每次都重新创建对象，可使用工厂方法。**

在处理大型资源密集型对象（比如数据库连接、文件系统和网络资源）时，你会经常碰到这种资源需求。

让我们思考复用现有对象的方法：

1. 首先，你需要创建存储空间来存放所有已经创建的对象。
2. 当他人请求一个对象时，程序将在对象池中搜索可用对象。
3. …然后将其返回给客户端代码。
4. 如果没有可用对象，程序则创建一个新对象（并将其添加到对象池中）。

这些代码可不少！而且它们必须位于同一处，这样才能确保重复代码不会污染程序。

可能最显而易见，也是最方便的方式，就是将这些代码放置在我们试图重用的对象类的构造函数中。但是从定义上来讲，构造函数始终返回的是**新对象**，其无法返回现有实例。

因此，你需要有一个既能够创建新对象，又可以重用现有对象的普通方法。这听上去和工厂方法非常相像。

## 实现方式

1. 让所有产品都遵循同一接口。该接口必须声明对所有产品都有意义的方法。

2. 在创建类中添加一个空的工厂方法。该方法的返回类型必须遵循通用的产品接口。

3. 在创建者代码中找到对于产品构造函数的所有引用。将它们依次替换为对于工厂方法的调用，同时将创建产品的代码移入工厂方法。你可能需要在工厂方法中添加临时参数来控制返回的产品类型。

   工厂方法的代码看上去可能非常糟糕。其中可能会有复杂的 switch 分支运算符，用于选择各种需要实例化的产品类。但是不要担心，我们很快就会修复这个问题。

4. 现在，为工厂方法中的每种产品编写一个创建者子类，然后在子类中重写工厂方法，并将基本方法中的相关创建代码移动到工厂方法中。

5. 如果应用中的产品类型太多，那么为每个产品创建子类并无太大必要，这时你也可以在子类中复用基类中的控制参数。

   例如，设想你有以下一些层次结构的类。基类 `邮件` 及其子类 `航空邮件` 和 `陆路邮件`；`运输` 及其子类 `飞机`，`卡车` 和 `火车`。`航空邮件` 仅使用 `飞机` 对象，而 `陆路邮件` 则会同时使用 `卡车` 和 `火车` 对象。你可以编写一个新的子类（例如 `火车邮件`）来处理这两种情况，但是还有其他可选的方案。客户端代码可以给 `陆路邮件` 类传递一个参数，用于控制其希望获得的产品。

6. 如果代码经过上述移动后，基础工厂方法中已经没有任何代码，你可以将其转变为抽象类。如果基础工厂方法中还有其他语句，你可以将其设置为该方法的默认行为。

## ⚖ 优缺点

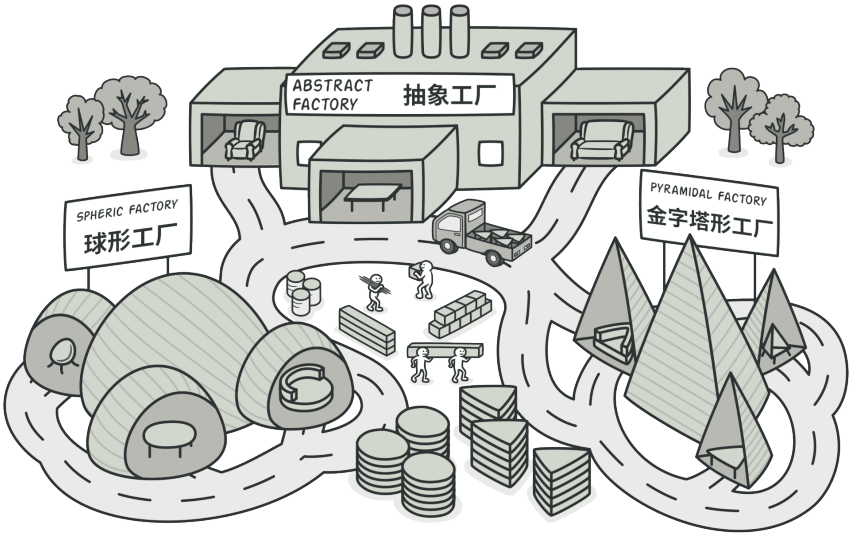
- ✓ 你可以避免创建者和具体产品之间的紧密耦合。
- ✓ 单一职责原则。你可以将产品创建代码放在程序的单一位置，从而使得代码更容易维护。

✓ 开闭原则。无需更改现有客户端代码，你就可以在程序中引入新的产品类型。

✕ 应用工厂方法模式需要引入许多新的子类，代码可能会因此变得更复杂。最好的情况是将该模式引入创建者类的现有层次结构中。

## ⇄ 与其他模式的关系

- 在许多设计工作的初期都会使用**工厂方法**（较为简单，而且可以更方便地通过子类进行定制），随后演化为使用**抽象工厂**、**原型**或**生成器**（更灵活但更加复杂）。
- **抽象工厂**模式通常基于一组**工厂方法**，但你也可以使用**原型**模式来生成这些类的方法。
- 你可以同时使用**工厂方法**和**迭代器**来让子类集合返回不同类型的迭代器，并使得迭代器与集合相匹配。
- **原型**并不基于继承，因此没有继承的缺点。另一方面，原型需要对被复制对象进行复杂的初始化。**工厂方法**基于继承，但是它不需要初始化步骤。
- **工厂方法**是**模板方法**的一种特殊形式。同时，工厂方法可以作为一个大型模板方法中的一个步骤。




# 抽象工厂

亦称：Abstract Factory

**抽象工厂**是一种创建型设计模式，
它能创建一系列相关的对象，而
无需指定其具体类。

# ☹ 问题

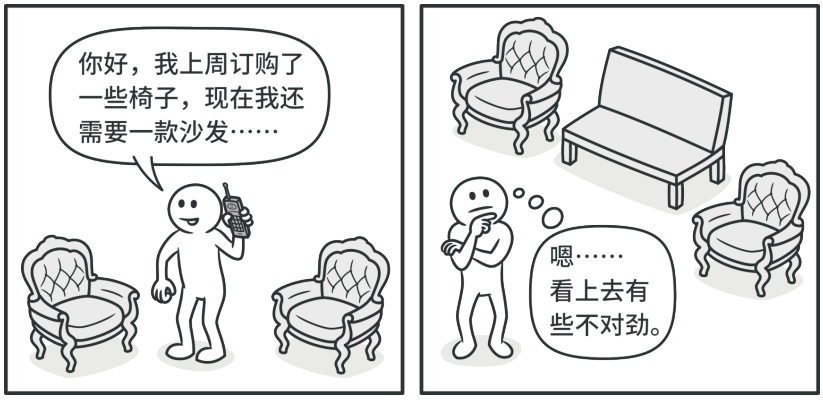
假设你正在开发一款家具商店模拟器。你的代码中包括一些类，用于表示：

1. 一系列相关产品，例如 椅子 Chair、沙发 Sofa 和 咖啡桌 CoffeeTable。

2. 系列产品的不同变体。例如，你可以使用 现代 Modern、维多利亚 Victorian、装饰风艺术 ArtDeco 等风格生成 椅子、沙发 和 咖啡桌。


系列产品及其不同变体。

你需要设法单独生成每件家具对象，这样才能确保其风格一致。如果顾客收到的家具风格不一样，他们可不会开心。

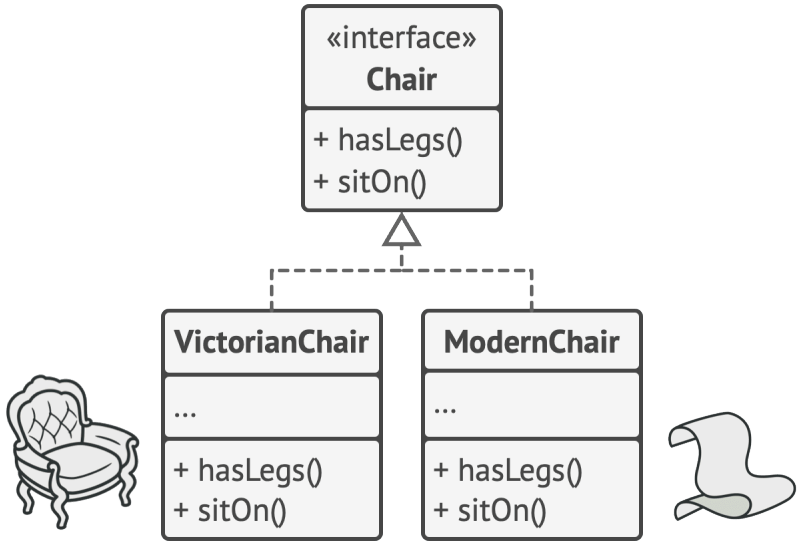



现代风格的沙发和维多利亚风格的椅子不搭。

此外，你也不希望在添加新产品或新风格时修改已有代码。家具供应商对于产品目录的更新非常频繁，你不会想在每次更新时都去修改核心代码的。

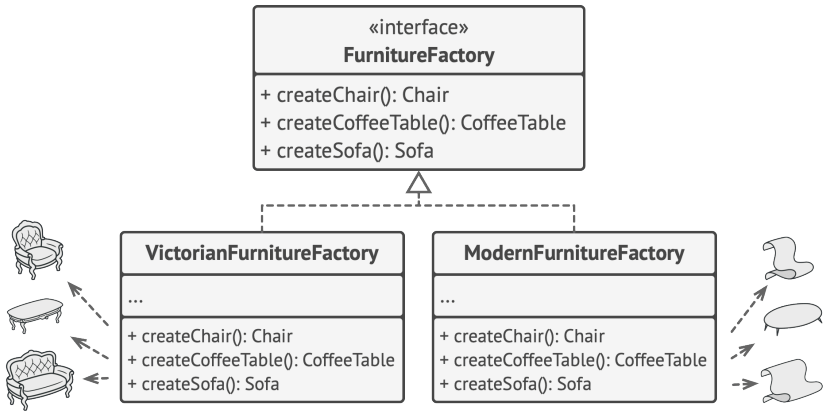
## 😊解决方案

首先，抽象工厂模式建议为系列中的每件产品明确声明接口（例如椅子、沙发或咖啡桌）。然后，确保所有产品变体都继承这些接口。例如，所有风格的椅子都实现`椅子`接口；所有风格的咖啡桌都实现`咖啡桌`接口，以此类推。




同一对象的所有变体都必须放置在同一个类层次结构之中。

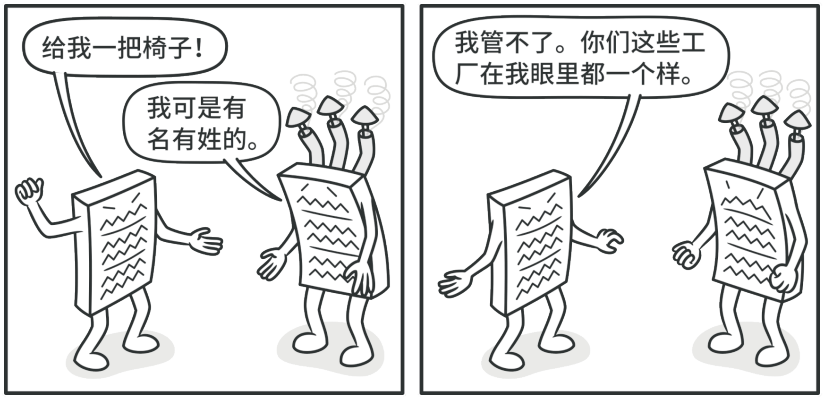
接下来，我们需要声明**抽象工厂**——包含系列中所有产品构造方法的接口。例如 createChair 创建椅子、createSofa 创建沙发和 createCoffeeTable 创建咖啡桌。这些方法必须返回**抽象**产品类型，即我们之前抽取的那些接口：椅子，沙发和咖啡桌等等。




每个具体工厂类都对应一个特定的产品变体。

那么该如何处理产品变体呢？对于系列产品的每个变体，我们都将基于 抽象工厂 接口创建不同的工厂类。每个工厂类都只能返回特定类别的产品，例如，现代家具工厂 ModernFurnitureFactory 只能创建 现代椅子 ModernChair、现代沙发 ModernSofa 和现代咖啡桌 ModernCoffeeTable 对象。

客户端代码可以通过相应的抽象接口调用工厂和产品类。你无需修改实际客户端代码，就能更改传递给客户端的工厂类，也能更改客户端代码接收的产品变体。

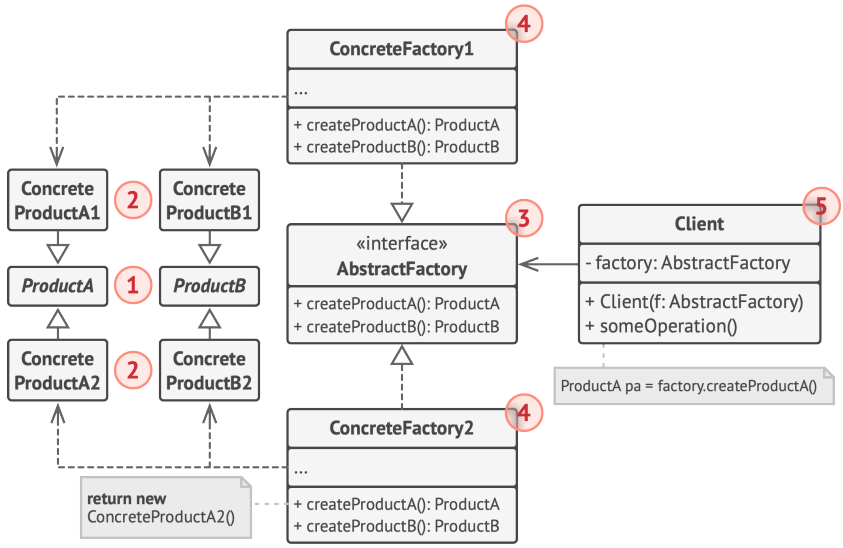



客户端无需了解其所调用工厂的具体类信息。

假设客户端想要工厂创建一把椅子。客户端无需了解工厂类，也不用管工厂类创建出的椅子类型。无论是现代风格，还是维多利亚风格的椅子，对于客户端来说没有分别，它只需调用抽象 `椅子` 接口就可以了。这样一来，客户端只需知道椅子以某种方式实现了 `sitOn 坐下` 方法就足够了。此外，无论工厂返回的是何种椅子变体，它都会和由同一工厂对象创建的沙发或咖啡桌风格一致。

最后一点说明：如果客户端仅接触抽象接口，那么谁来创建实际的工厂对象呢？一般情况下，应用程序会在初始化阶段创建具体工厂对象。而在此之前，应用程序必须根据配置文件或环境设定选择工厂类别。

# 结构




1. **抽象产品**（Abstract Product）为构成系列产品的一组不同但相关的产品声明接口。

2. **具体产品**（Concrete Product）是抽象产品的多种不同类型实现。所有变体（维多利亚/现代）都必须实现相应的抽象产品（椅子/沙发）。

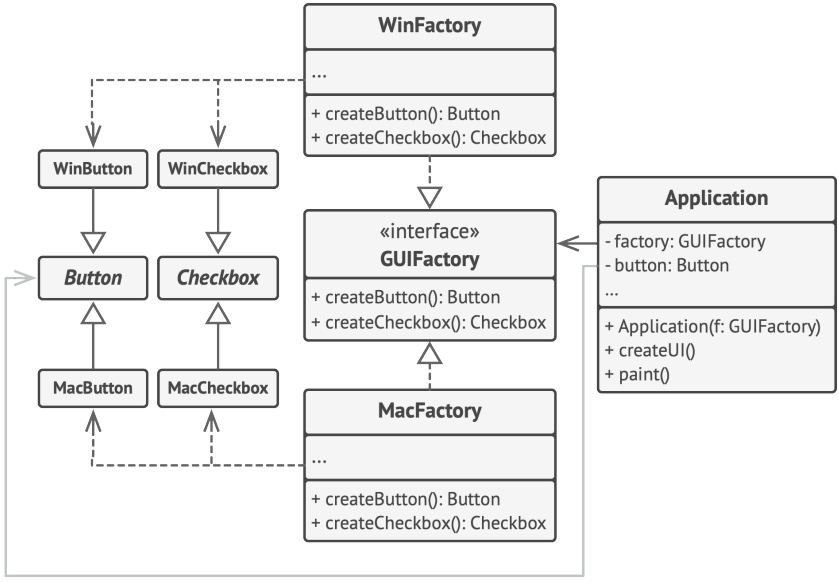
3. **抽象工厂**（Abstract Factory）接口声明了一组创建各种抽象产品的方法。

4. **具体工厂**（Concrete Factory）实现抽象工厂的构建方法。每个具体工厂都对应特定产品变体，且仅创建此种产品变体。

5. 尽管具体工厂会对具体产品进行初始化，其构建方法签名必须返回相应的抽象产品。这样，使用工厂类的客户端代码就不会与工厂创建的特定产品变体耦合。**客户端**（Client）只需通过抽象接口调用工厂和产品对象，就能与任何具体工厂/产品变体交互。

# 伪代码

下面例子通过应用**抽象工厂**模式，使得客户端代码无需与具体 UI 类耦合，就能创建跨平台的 UI 元素，同时确保所创建的元素与指定的操作系统匹配。




跨平台 UI 类示例。

跨平台应用中的相同 UI 元素功能类似，但是在不同操作系统下的外观有一定差异。此外，你需要确保 UI 元素与当前操作系统风格一致。你一定不希望在 Windows 系统下运行的应用程序中显示 macOS 的控件。

抽象工厂接口声明一系列构建方法，客户端代码可调用它们生成不同风格的 UI 元素。每个具体工厂对应特定操作系统，并负责生成符合该操作系统风格的 UI 元素。

其运作方式如下：应用程序启动后检测当前操作系统。根据该信息，应用程序通过与该操作系统对应的类创建工厂对象。其余代码使用该工厂对象创建 UI 元素。这样可以避免生成错误类型的元素。

使用这种方法，客户端代码只需调用抽象接口，而无需了解具体工厂类和 UI 元素。此外，客户端代码还支持未来添加新的工厂或 UI 元素。

这样一来，每次在应用程序中添加新的 UI 元素变体时，你都无需修改客户端代码。你只需创建一个能够生成这些 UI 元素的工厂类，然后稍微修改应用程序的初始代码，使其能够选择合适的工厂类即可。

```
// 抽象工厂接口声明了一组能返回不同抽象产品的方法。这些产品属于同一个系列
// 且在高层主题或概念上具有相关性。同系列的产品通常能相互搭配使用。系列产
// 品可有多个变体，但不同变体的产品不能搭配使用。
interface GUIFactory is
  method createButton():Button
  method createCheckbox():Checkbox


// 具体工厂可生成属于同一变体的系列产品。工厂会确保其创建的产品能相互搭配
// 使用。具体工厂方法签名会返回一个抽象产品，但在方法内部则会对具体产品进
// 行实例化。
class WinFactory implements GUIFactory is
  method createButton():Button is
    return new WinButton()
  method createCheckbox():Checkbox is
    return new WinCheckbox()

// 每个具体工厂中都会包含一个相应的产品变体。
class MacFactory implements GUIFactory is
  method createButton():Button is
    return new MacButton()
  method createCheckbox():Checkbox is
    return new MacCheckbox()


// 系列产品中的特定产品必须有一个基础接口。所有产品变体都必须实现这个接口。
interface Button is
  method paint()

// 具体产品由相应的具体工厂创建。
class WinButton implements Button is
```

```
    method paint() is
      // 根据 Windows 样式渲染按钮。

class MacButton implements Button is
    method paint() is
      // 根据 macOS 样式渲染按钮

// 这是另一个产品的基础接口。所有产品都可以互动，但是只有相同具体变体的产
// 品之间才能够正确地进行交互。
interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
      // 根据 macOS 样式渲染复选框。

class MacCheckbox implements Checkbox is
    method paint() is
      // 根据 macOS 样式渲染复选框。

// 客户端代码仅通过抽象类型（GUIFactory、Button 和 Checkbox）使用工厂
// 和产品。这让你无需修改任何工厂或产品子类就能将其传递给客户端代码。
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
      this.factory = factory
    method createUI() is
      this.button = factory.createButton()
    method paint() is
      button.paint()

```

```
64
65  // 程序会根据当前配置或环境设定选择工厂类型，并在运行时创建工厂（通常在初
66  // 始化阶段）。
67  class ApplicationConfigurator is
68    method main() is
69      config = readApplicationConfigFile()
70
71      if (config.OS == "Windows") then
72        factory = new WinFactory()
73      else if (config.OS == "Mac") then
74        factory = new MacFactory()
75      else
76        throw new Exception("错误！未知的操作系统。")
77
78      Application app = new Application(factory)
```

## 适合应用场景

**如果代码需要与多个不同系列的相关产品交互，但是由于无法提前获取相关信息，或者出于对未来扩展性的考虑，你不希望代码基于产品的具体类进行构建，在这种情况下，你可以使用抽象工厂。**

抽象工厂为你提供了一个接口，可用于创建每个系列产品的对象。只要代码通过该接口创建对象，那么你就不会生成与应用程序已生成的产品类型不一致的产品。

**如果你有一个基于一组抽象方法的类，且其主要功能因此变得不明确，那么在这种情况下可以考虑使用抽象工厂模式。**

在设计良好的程序中，每个类仅负责一件事。如果一个类与多种类型产品交互，就可以考虑将工厂方法抽取到独立的工厂类或具备完整功能的抽象工厂类中。

## 实现方式

1. 以不同的产品类型与产品变体为维度绘制矩阵。

2. 为所有产品声明抽象产品接口。然后让所有具体产品类实现这些接口。

3. 声明抽象工厂接口，并且在接口中为所有抽象产品提供一组构建方法。

4. 为每种产品变体实现一个具体工厂类。

5. 在应用程序中开发初始化代码。该代码根据应用程序配置或当前环境，对特定具体工厂类进行初始化。然后将该工厂对象传递给所有需要创建产品的类。

6. 找出代码中所有对产品构造函数的直接调用，将其替换为对工厂对象中相应构建方法的调用。

# ⚖ 优缺点

- ✓ 你可以确保同一工厂生成的产品相互匹配。
- ✓ 你可以避免客户端和具体产品代码的耦合。
- ✓ 单一职责原则。你可以将产品生成代码抽取到同一位置，使得代码易于维护。
- ✓ 开闭原则。向应用程序中引入新产品变体时，你无需修改客户端代码。
- ✕ 由于采用该模式需要向应用中引入众多接口和类，代码可能会比之前更加复杂。

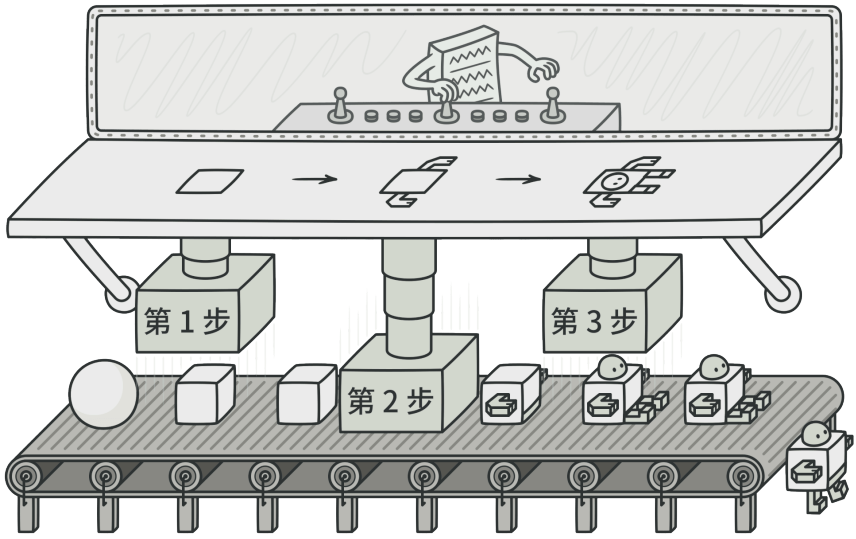
# ⇄ 与其他模式的关系

- 在许多设计工作的初期都会使用**工厂方法**（较为简单，而且可以更方便地通过子类进行定制），随后演化为使用**抽象工厂**、**原型**或**生成器**（更灵活但更加复杂）。
- **生成器**重点关注如何分步生成复杂对象。**抽象工厂**专门用于生产一系列相关对象。抽象工厂会马上返回产品，生成器则允许你在获取产品前执行一些额外构造步骤。
- **抽象工厂**模式通常基于一组**工厂方法**，但你也可以使用**原型**模式来生成这些类的方法。

- 当只需对客户端代码隐藏子系统创建对象的方式时，你可以使用**抽象工厂**来代替**外观**。

- 你可以将**抽象工厂**和**桥接**搭配使用。如果由桥接定义的抽象只能与特定实现合作，这一模式搭配就非常有用。在这种情况下，抽象工厂可以对这些关系进行封装，并且对客户端代码隐藏其复杂性。

- **抽象工厂**、**生成器**和**原型**都可以用**单例**来实现。



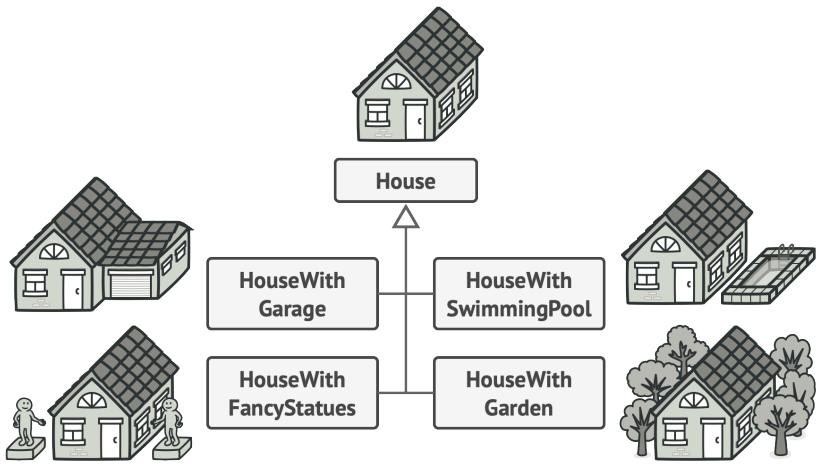

# 生成器

亦称：建造者模式、Builder

**生成器**是一种创建型设计模式，
使你能够分步骤创建复杂对象。
该模式允许你使用相同的创建
代码生成不同类型和形式的对象。

# ☹ 问题

假设有这样一个复杂对象，在对其进行构造时需要对诸多成员变量和嵌套对象进行繁复的初始化工作。这些初始化代码通常深藏于一个包含众多参数且让人基本看不懂的构造函数中；甚至还有更糟糕的情况，那就是这些代码散落在客户端代码的多个位置。

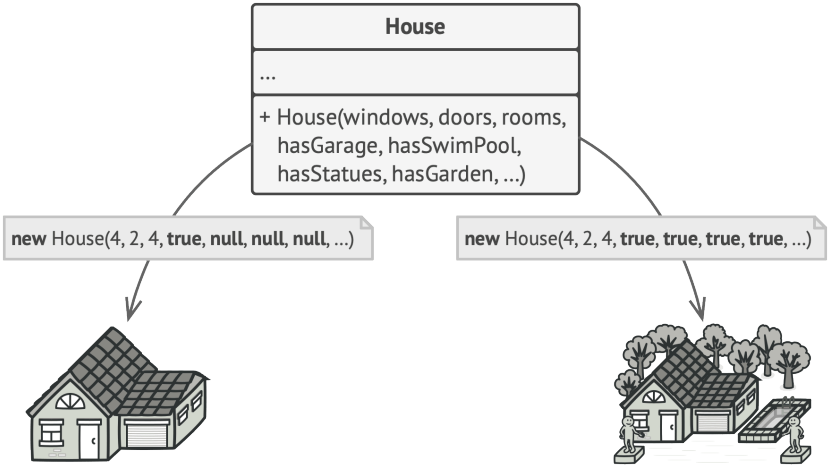



如果为每种可能的对象都创建一个子类，这可能会导致程序变得过于复杂。

例如，我们来思考如何创建一个 房屋 House 对象。建造一栋简单的房屋，首先你需要建造四面墙和地板，安装房门和一套窗户，然后再建造一个屋顶。但是如果你想要一栋更宽敞更明亮的房屋，还要有院子和其他设施（例如暖气、排水和供电设备），那又该怎么办呢？

最简单的方法是扩展 房屋 基类，然后创建一系列涵盖所有参数组合的子类。但最终你将面对相当数量的子类。任何新增的参数（例如门廊类型）都会让这个层次结构更加复杂。

另一种方法则无需生成子类。你可以在 房屋 基类中创建一个包括所有可能参数的超级构造函数，并用它来控制房屋对象。这种方法确实可以避免生成子类，但它却会造成另外一个问题。

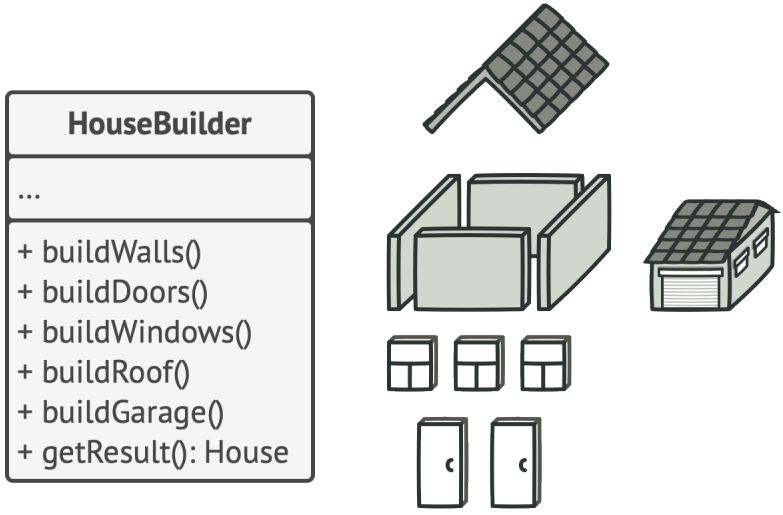



拥有大量输入参数的构造函数也有缺陷：这些参数也不是每次都要全部用上的。

通常情况下，绝大部分的参数都没有使用，这使得**对于构造函数的调用十分不简洁**。例如，只有很少的房子有游泳池，因此与游泳池相关的参数十之八九是毫无用处的。

# 😊解决方案

生成器模式建议将对象构造代码从产品类中抽取出来，并将其放在一个名为**生成器**的独立对象中。

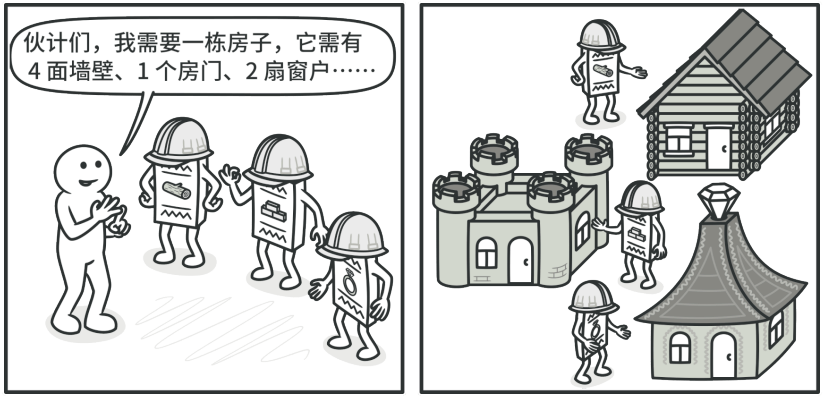



生成器模式让你能够分步骤创建复杂对象。生成器不允许其他对象访问正在创建中的产品。

该模式会将对象构造过程划分为一组步骤，比如 buildWalls 创建墙壁和 buildDoor 创建房门创建房门等。每次创建对象时，你都需要通过生成器对象执行一系列步骤。重点在于你无需调用所有步骤，而只需调用创建特定对象配置所需的那些步骤即可。

当你需要创建不同形式的产品时，其中的一些构造步骤可能需要不同的实现。例如，木屋的房门可能需要使用木头制造，而城堡的房门则必须使用石头制造。

在这种情况下，你可以创建多个不同的生成器，用不同方式实现一组相同的创建步骤。然后你就可以在创建过程中使用这些生成器（例如按顺序调用多个构造步骤）来生成不同类型的对象。

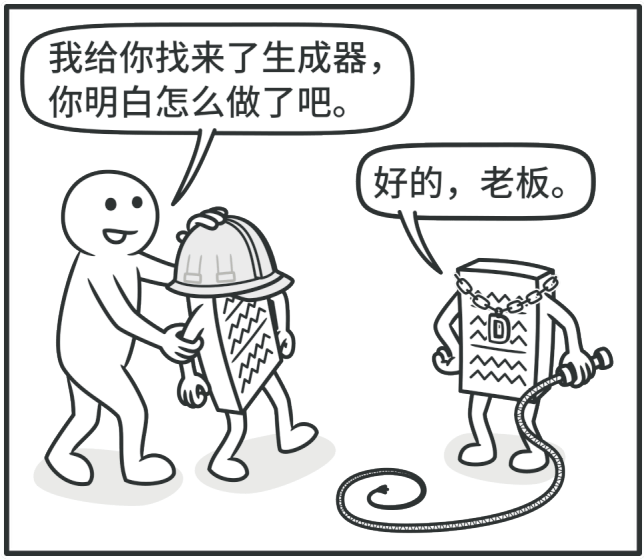



不同生成器以不同方式执行相同的任务。

例如，假设第一个建造者使用木头和玻璃制造房屋，第二个建造者使用石头和钢铁，而第三个建造者使用黄金和钻石。在调用同一组步骤后，第一个建造者会给你一栋普通房屋，第二个会给你一座小城堡，而第三个则会给你一座宫殿。但是，只有在调用构造步骤的客户端代码可以通过通用接口与建造者进行交互时，这样的调用才能返回需要的房屋。

## 主管

你可以进一步将用于创建产品的一系列生成器步骤调用抽取成为单独的主管类。主管类可定义创建步骤的执行顺序，而生成器则提供这些步骤的实现。

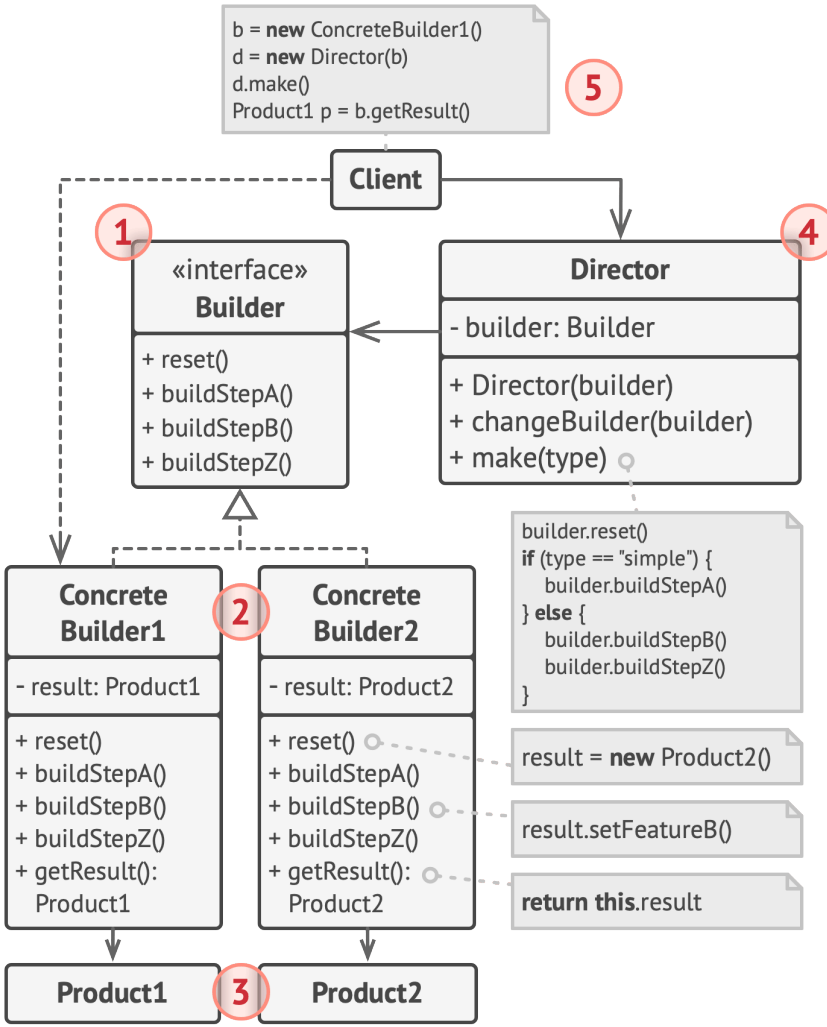



主管知道需要哪些创建步骤才能获得可正常使用的产品。

严格来说，你的程序中并不一定需要主管类。客户端代码可直接以特定顺序调用创建步骤。不过，主管类中非常适合放入各种例行构造流程，以便在程序中反复使用。

此外，对于客户端代码来说，主管类完全隐藏了产品构造细节。客户端只需要将一个生成器与主管类关联，然后使用主管类来构造产品，就能从生成器处获得构造结果了。

# 结构




1. **生成器**（Builder）接口声明在所有类型生成器中通用的产品构造步骤。

2. **具体生成器**（Concrete Builders）提供构造过程的不同实现。具体生成器也可以构造不遵循通用接口的产品。

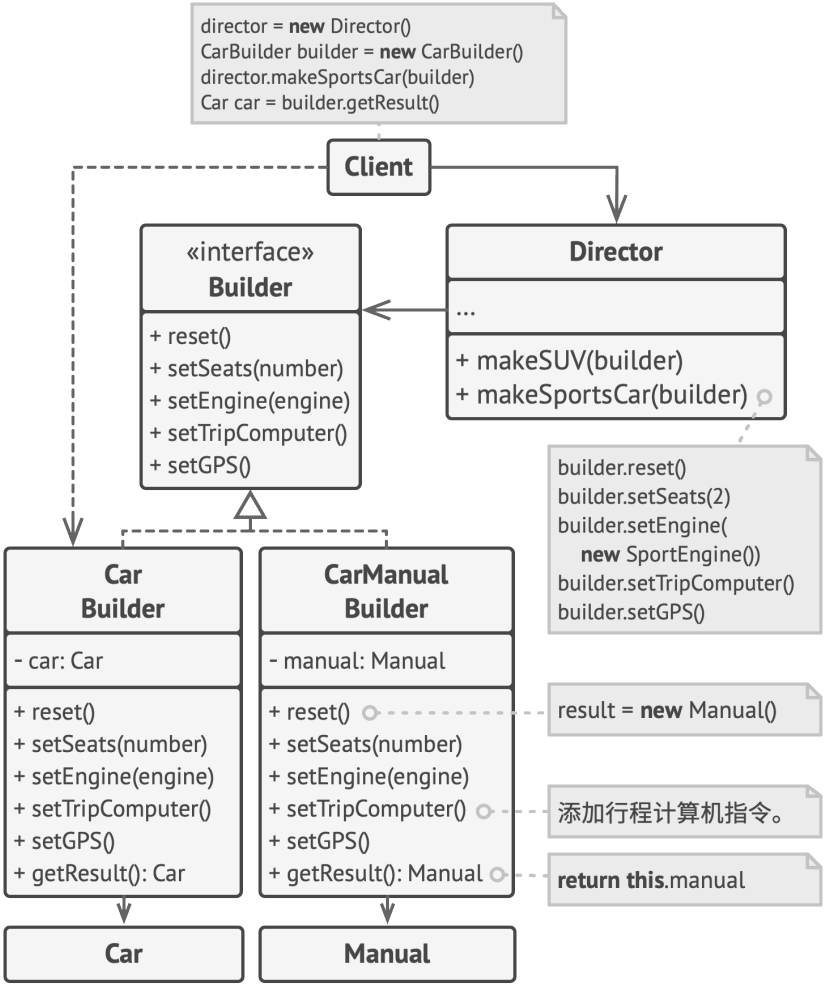
3. **产品**（Products）是最终生成的对象。由不同生成器构造的产品无需属于同一类层次结构或接口。

4. **主管**（Director）类定义调用构造步骤的顺序，这样你就可以创建和复用特定的产品配置。

5. **客户端**（Client）必须将某个生成器对象与主管类关联。一般情况下，你只需通过主管类构造函数的参数进行一次性关联即可。此后主管类就能使用生成器对象完成后续所有的构造任务。但在客户端将生成器对象传递给主管类制造方法时还有另一种方式。在这种情况下，你在使用主管类生产产品时每次都可以使用不同的生成器。

# # 伪代码

下面关于**生成器**模式的例子演示了你可以如何复用相同的对象构造代码来生成不同类型的产品——例如汽车（Car）——及其相应的使用手册（Manual）。




分步骤制造汽车并制作对应型号用户使用手册的示例

汽车是一个复杂对象，有数百种不同的制造方法。我们没有在 `汽车` 类中塞入一个巨型构造函数，而是将汽车组装代码抽取到单独的汽车生成器类中。该类中有一组方法可用来配置汽车的各种部件。

如果客户端代码需要组装一辆与众不同、精心调教的汽车，它可以直接调用生成器。或者，客户端可以将组装工作委托给主管类，因为主管类知道如何使用生成器制造最受欢迎的几种型号汽车。

你或许会感到吃惊，但确实每辆汽车都需要一本使用手册（说真的，谁会去读它们呢?）。使用手册会介绍汽车的每一项功能，因此不同型号的汽车，其使用手册内容也不一样。因此，你可以复用现有流程来制造实际的汽车及其对应的手册。当然，编写手册和制造汽车不是一回事，所以我们需要另外一个生成器对象来专门编写使用手册。该类与其制造汽车的兄弟类都实现了相同的制造方法，但是其功能不是制造汽车部件，而是描述每个部件。将这些生成器传递给相同的主管对象，我们就能够生成一辆汽车或是一本使用手册了。

最后一个部分是获取结果对象。尽管金属汽车和纸质手册存在关联，但它们却是完全不同的东西。我们无法在主管类和具体产品类不发生耦合的情况下，在主管类中提供获取结果对象的方法。因此，我们只能通过负责制造过程的生成器来获取结果对象。

```
// 只有当产品较为复杂且需要详细配置时，使用生成器模式才有意义。下面的两个
// 产品尽管没有同样的接口，但却相互关联。
class Car is
  // 一辆汽车可能配备有 GPS 设备、行车电脑和几个座位。不同型号的汽车（
```

```
  // 运动型轿车、SUV 和敞篷车）可能会安装或启用不同的功能。

class Manual is
  // 用户使用手册应该根据汽车配置进行编制，并介绍汽车的所有功能。


// 生成器接口声明了创建产品对象不同部件的方法。
interface Builder is
  method reset()
  method setSeats(...)
  method setEngine(...)
  method setTripComputer(...)
  method setGPS(...)

// 具体生成器类将遵循生成器接口并提供生成步骤的具体实现。你的程序中可能会
// 有多个以不同方式实现的生成器变体。
class CarBuilder implements Builder is
  private field car:Car

  // 一个新的生成器实例必须包含一个在后续组装过程中使用的空产品对象。
  constructor CarBuilder() is
    this.reset()

  // reset（重置）方法可清除正在生成的对象。
  method reset() is
    this.car = new Car()

  // 所有生成步骤都会与同一个产品实例进行交互。
  method setSeats(...) is
    // 设置汽车座位的数量。

  method setEngine(...) is
```

```
    // 安装指定的引擎。

  method setTripComputer(...) is
    // 安装行车电脑。

  method setGPS(...) is
    // 安装全球定位系统。

  // 具体生成器需要自行提供获取结果的方法。这是因为不同类型的生成器可能
  // 会创建不遵循相同接口的、完全不同的产品。所以也就无法在生成器接口中
  // 声明这些方法（至少在静态类型的编程语言中是这样的）。
  //
  // 通常在生成器实例将结果返回给客户端后，它们应该做好生成另一个产品的
  // 准备。因此生成器实例通常会在 `getProduct（获取产品）`方法主体末尾
  // 调用重置方法。但是该行为并不是必需的，你也可让生成器等待客户端明确
  // 调用重置方法后再去处理之前的结果。
  method getProduct():Car is
    product = this.car
    this.reset()
    return product

// 生成器与其他创建型模式的不同之处在于：它让你能创建不遵循相同接口的产品。
class CarManualBuilder implements Builder is
  private field manual:Manual

  constructor CarManualBuilder() is
    this.reset()

  method reset() is
    this.manual = new Manual()

  method setSeats(...) is
```

```
    // 添加关于汽车座椅功能的文档。

  method setEngine(...) is
    // 添加关于引擎的介绍。

  method setTripComputer(...) is
    // 添加关于行车电脑的介绍。

  method setGPS(...) is
    // 添加关于 GPS 的介绍。

  method getProduct():Manual is
    // 返回使用手册并重置生成器。


// 主管只负责按照特定顺序执行生成步骤。其在根据特定步骤或配置来生成产品时
// 会很有帮助。由于客户端可以直接控制生成器，所以严格意义上来说，主管类并
// 不是必需的。
class Director is
  private field builder:Builder

  // 主管可同由客户端代码传递给自身的任何生成器实例进行交互。客户端可通
  // 过这种方式改变最新组装完毕的产品的最终类型。
  method setBuilder(builder:Builder)
    this.builder = builder

  // 主管可使用同样的生成步骤创建多个产品变体。
  method constructSportsCar(builder: Builder) is
    builder.reset()
    builder.setSeats(2)
    builder.setEngine(new SportEngine())
    builder.setTripComputer(true)
```

```
    builder.setGPS(true)

  method constructSUV(builder: Builder) is
    // ...


// 客户端代码会创建生成器对象并将其传递给主管，然后执行构造过程。最终结果
// 将需要从生成器对象中获取。
class Application is

  method makeCar() is
    director = new Director()

    CarBuilder builder = new CarBuilder()
    director.constructSportsCar(builder)
    Car car = builder.getProduct()

    CarManualBuilder builder = new CarManualBuilder()
    director.constructSportsCar(builder)

    // 最终产品通常需要从生成器对象中获取，因为主管不知晓具体生成器和
    // 产品的存在，也不会对其产生依赖。
    Manual manual = builder.getProduct()
```

## 适合应用场景

使用生成器模式可避免“重叠构造函数（telescopic constructor）”的出现。

假设你的构造函数中有十个可选参数，那么调用该函数会非常不方便；因此，你需要重载这个构造函数，新建几个只有较少参数的简化版。但这些构造函数仍需调用主构造函数，传递一些默认数值来替代省略掉的参数。

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  Pizza(int size, boolean cheese, boolean pepperoni) { ... }
  // ...
```

只有在 C# 或 Java 等支持方法重载的编程语言中才能写出如此复杂的构造函数。

生成器模式让你可以分步骤生成对象，而且允许你仅使用必须的步骤。应用该模式后，你再也不需要将几十个参数塞进构造函数里了。

**当你希望使用代码创建不同形式的产品（例如石头或木头房屋）时，可使用生成器模式。**

如果你需要创建的各种形式的产品，它们的制造过程相似且仅有细节上的差异，此时可使用生成器模式。

基本生成器接口中定义了所有可能的制造步骤，具体生成器将实现这些步骤来制造特定形式的产品。同时，主管类将负责管理制造步骤的顺序。

**使用生成器构造组合树或其他复杂对象。**

生成器模式让你能分步骤构造产品。你可以延迟执行某些步骤而不会影响最终产品。你甚至可以递归调用这些步骤，这在创建对象树时非常方便。

生成器在执行制造步骤时，不能对外发布未完成的产品。这可以避免客户端代码获取到不完整结果对象的情况。

## 实现方法

1. 清晰地定义通用步骤，确保它们可以制造所有形式的产品。否则你将无法进一步实施该模式。

2. 在基本生成器接口中声明这些步骤。

3. 为每个形式的产品创建具体生成器类，并实现其构造步骤。

   不要忘记实现获取构造结果对象的方法。你不能在生成器接口中声明该方法，因为不同生成器构造的产品可能没有公共接口，因此你就不知道该方法返回的对象类型。但是，如果

所有产品都位于单一类层次中，你就可以安全地在基本接口中添加获取生成对象的方法。

4. 考虑创建主管类。它可以使用同一生成器对象来封装多种构造产品的方式。

5. 客户端代码会同时创建生成器和主管对象。构造开始前，客户端必须将生成器对象传递给主管对象。通常情况下，客户端只需调用主管类构造函数一次即可。主管类使用生成器对象完成后续所有制造任务。还有另一种方式，那就是客户端可以将生成器对象直接传递给主管类的制造方法。

6. 只有在所有产品都遵循相同接口的情况下，构造结果可以直接通过主管类获取。否则，客户端应当通过生成器获取构造结果。

## ⚖ 优缺点

✓ 你可以分步创建对象，暂缓创建步骤或递归运行创建步骤。

✓ 生成不同形式的产品时，你可以复用相同的制造代码。

✓ 单一职责原则。你可以将复杂构造代码从产品的业务逻辑中分离出来。

✕ 由于该模式需要新增多个类，因此代码整体复杂程度会有所增加。

# ⇄ 与其他模式的关系

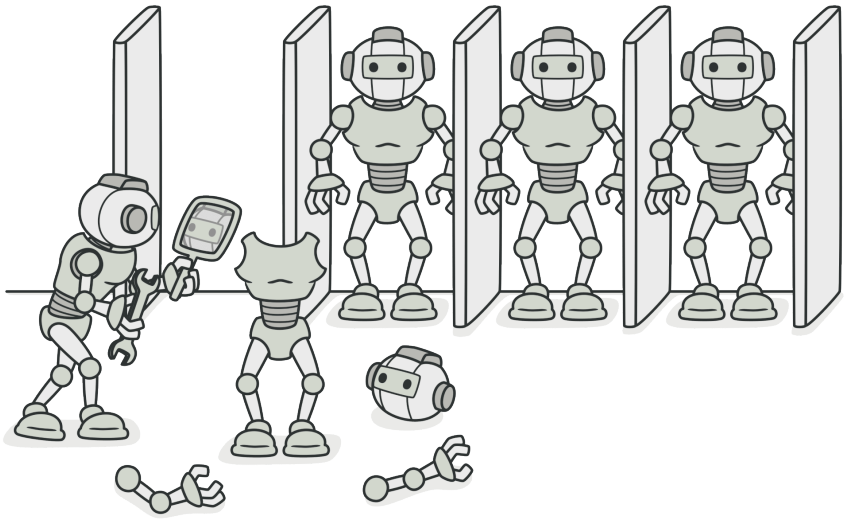
- 在许多设计工作的初期都会使用**工厂方法**（较为简单，而且可以更方便地通过子类进行定制），随后演化为使用**抽象工厂**、**原型**或**生成器**（更灵活但更加复杂）。

- **生成器**重点关注如何分步生成复杂对象。**抽象工厂**专门用于生产一系列相关对象。抽象工厂会马上返回产品，生成器则允许你在获取产品前执行一些额外构造步骤。

- 你可以在创建复杂**组合**树时使用**生成器**，因为这可使其构造步骤以递归的方式运行。

- 你可以结合使用**生成器**和**桥接**模式：主管类负责抽象工作，各种不同的生成器负责实现工作。

- **抽象工厂**、**生成器**和**原型**都可以用**单例**来实现。



# 原型

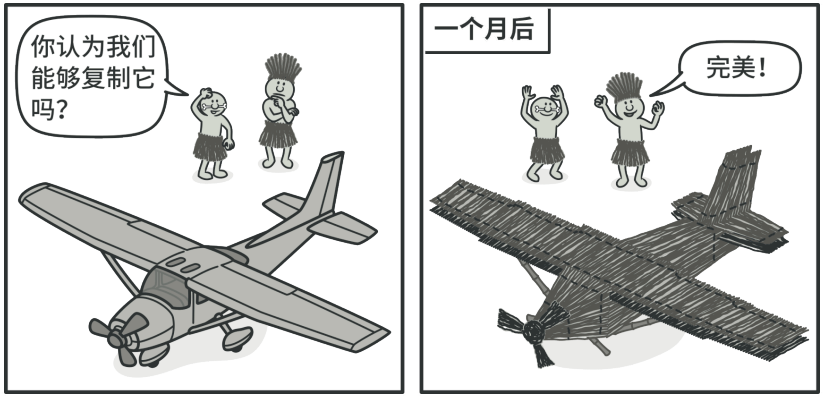
亦称：克隆、Clone、Prototype

**原型**是一种创建型设计模式，使你能够复制已有对象，而又无需使代码依赖它们所属的类。

# ☹ 问题

如果你有一个对象，并希望生成与其完全相同的一个复制品，你该如何实现呢？首先，你必须新建一个属于相同类的对象。然后，你必须遍历原始对象的所有成员变量，并将成员变量值复制到新对象中。

不错！但有个小问题。并非所有对象都能通过这种方式进行复制，因为有些对象可能拥有私有成员变量，它们在对象本身以外是不可见的。




“从外部”复制对象**并非**总是可行。

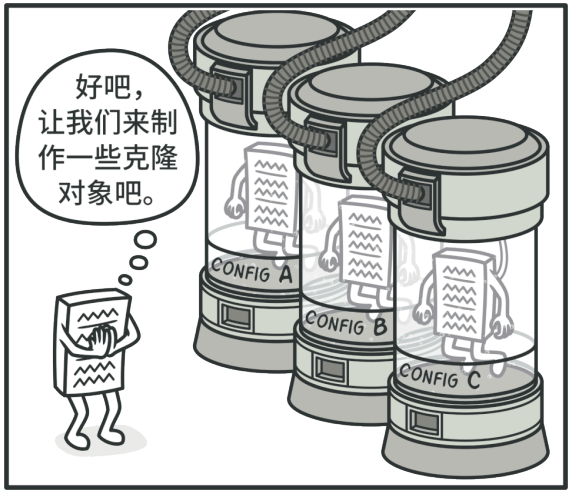
直接复制还有另外一个问题。因为你必须知道对象所属的类才能创建复制品，所以代码必须依赖该类。即使你可以接受额外的依赖性，那还有另外一个问题：有时你只知道对象所实现的接口，而不知道其所属的具体类，比如可向方法的某个参数传入实现了某个接口的任何对象。

# 😊解决方案

原型模式将克隆过程委派给被克隆的实际对象。模式为所有支持克隆的对象声明了一个通用接口，该接口让你能够克隆对象，同时又无需将代码和对象所属类耦合。通常情况下，这样的接口中仅包含一个`克隆`方法。

所有的类对`克隆`方法的实现都非常相似。该方法会创建一个当前类的对象，然后将原始对象所有的成员变量值复制到新建的类中。你甚至可以复制私有成员变量，因为绝大部分编程语言都允许对象访问其同类对象的私有成员变量。

支持克隆的对象即为**原型**。当你的对象有几十个成员变量和几百种类型时，对其进行克隆甚至可以代替子类的构造。

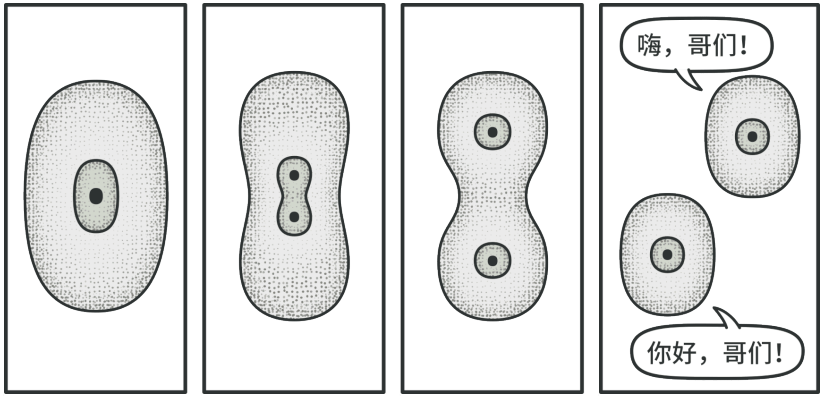



预生成原型可以代替子类的构造。

其运作方式如下：创建一系列不同类型的对象并不同的方式对其进行配置。如果所需对象与预先配置的对象相同，那么你只需克隆原型即可，无需新建一个对象。

## 真实世界类比

现实生活中，产品在得到大规模生产前会使用原型进行各种测试。但在这种情况下，原型只是一种被动的工具，不参与任何真正的生产活动。

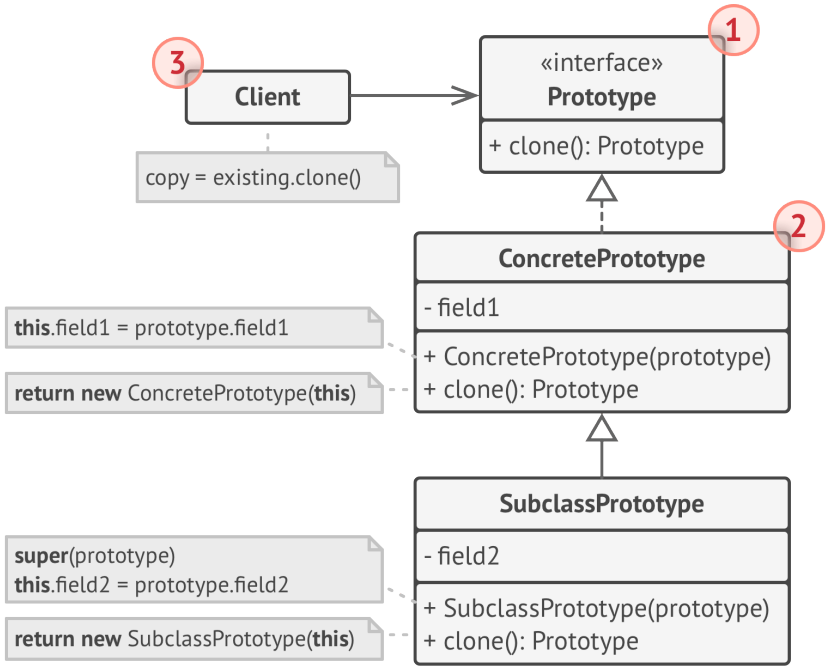



一个细胞的分裂。

由于工业原型并不是真正意义上的自我复制，因此细胞有丝分裂（还记得生物学知识吗?）或许是更恰当的类比。有丝分裂会产生一对完全相同的细胞。原始细胞就是一个原型，它在复制体的生成过程中起到了推动作用。

# 结构

## 基本实现

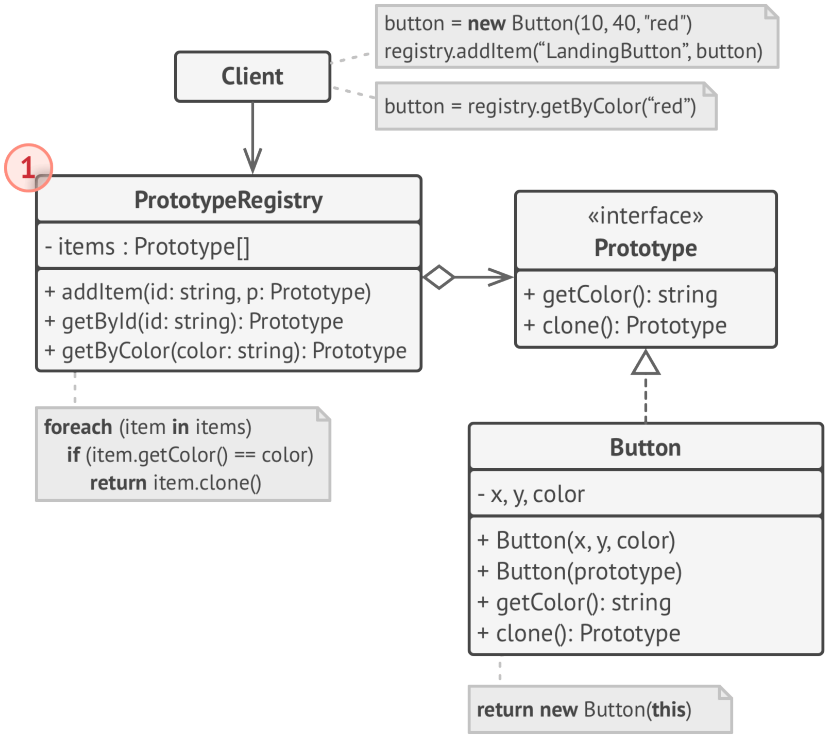



1. **原型**（Prototype）接口将对克隆方法进行声明。在绝大多数情况下，其中只会有一个名为 clone 克隆 的方法。

2. **具体原型**（Concrete Prototype）类将实现克隆方法。除了将原始对象的数据复制到克隆体中之外，该方法有时还需处理克隆过程中的极端情况，例如克隆关联对象和梳理递归依赖等等。

3. **客户端**（Client）可以复制实现了原型接口的任何对象。

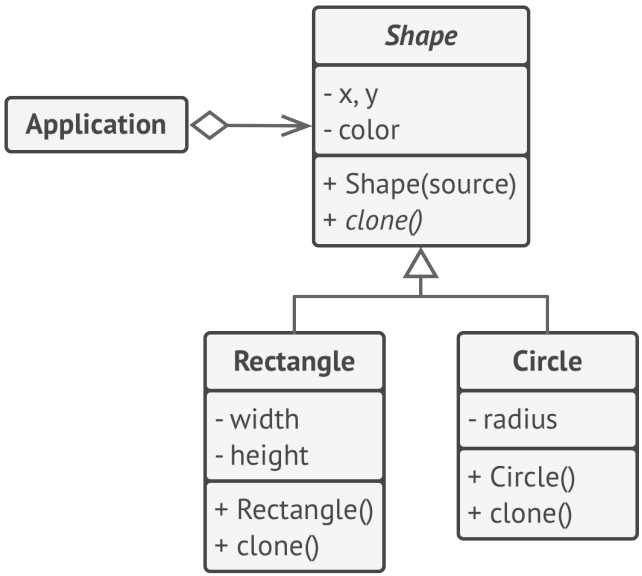
## 原型注册表实现




1. **原型注册表**（Prototype Registry）提供了一种访问常用原型的简单方法，其中存储了一系列可供随时复制的预生成对象。最简单的注册表原型是一个 `名称 → 原型` 的哈希表。但如果需要使用名称以外的条件进行搜索，你可以创建更加完善的注册表版本。

# # 伪代码

在本例中，**原型**模式能让你生成完全相同的几何对象副本，同时无需代码与对象所属类耦合。




克隆一系列位于同一类层次结构中的对象。

所有形状类都遵循同一个提供克隆方法的接口。在复制自身成员变量值到结果对象前，子类可调用其父类的克隆方法。

```
// 基础原型。
abstract class Shape is
  field X: int
  field Y: int
  field color: string
```

```
6
7     // 常规构造函数。
8     constructor Shape() is
9       // ...
10
11    // 原型构造函数。使用已有对象的数值来初始化一个新对象。
12    constructor Shape(source: Shape) is
13      this()
14      this.X = source.X
15      this.Y = source.Y
16      this.color = source.color
17
18    // clone（克隆）操作会返回一个形状子类。
19    abstract method clone():Shape
20
21
22  // 具体原型。克隆方法会创建一个新对象并将其传递给构造函数。直到构造函数运
23  // 行完成前，它都拥有指向新克隆对象的引用。因此，任何人都无法访问未完全生
24  // 成的克隆对象。这可以保持克隆结果的一致。
25  class Rectangle extends Shape is
26    field width: int
27    field height: int
28
29    constructor Rectangle(source: Rectangle) is
30      // 需要调用父构造函数来复制父类中定义的私有成员变量。
31      super(source)
32      this.width = source.width
33      this.height = source.height
34
35    method clone():Shape is
36      return new Rectangle(this)
37
```

```
38
39  class Circle extends Shape is
40    field radius: int
41
42    constructor Circle(source: Circle) is
43      super(source)
44      this.radius = source.radius
45
46    method clone():Shape is
47      return new Circle(this)
48
49
50  // 客户端代码中的某个位置。
51  class Application is
52    field shapes: array of Shape
53
54    constructor Application() is
55      Circle circle = new Circle()
56      circle.X = 10
57      circle.Y = 10
58      circle.radius = 20
59      shapes.add(circle)
60
61      Circle anotherCircle = circle.clone()
62      shapes.add(anotherCircle)
63      // 变量 `anotherCircle（另一个圆）`与 `circle（圆）`对象的内
64      // 容完全一样。
65
66      Rectangle rectangle = new Rectangle()
67      rectangle.width = 10
68      rectangle.height = 20
69      shapes.add(rectangle)
```

```
70
71    method businessLogic() is
72      // 原型是很强大的东西，因为它能在不知晓对象类型的情况下生成一个与
73      // 其完全相同的复制品。
74      Array shapesCopy = new Array of Shapes.
75
76      // 例如，我们不知晓形状数组中元素的具体类型，只知道它们都是形状。
77      // 但在多态机制的帮助下，当我们在某个形状上调用 `clone（克隆）`
78      // 方法时，程序会检查其所属的类并调用其中所定义的克隆方法。这样，
79      // 我们将获得一个正确的复制品，而不是一组简单的形状对象。
80      foreach (s in shapes) do
81        shapesCopy.add(s.clone())
82
83      // `shapesCopy（形状副本）`数组中包含 `shape（形状）`数组所有
84      // 子元素的复制品。
```

## 适合应用场景

**如果你需要复制一些对象，同时又希望代码独立于这些对象所属的具体类，可以使用原型模式。**

这一点考量通常出现在代码需要处理第三方代码通过接口传递过来的对象时。即使不考虑代码耦合的情况，你的代码也不能依赖这些对象所属的具体类，因为你不知道它们的具体信息。

原型模式为客户端代码提供一个通用接口，客户端代码可通过这一接口与所有实现了克隆的对象进行交互，它也使得客户端代码与其所克隆的对象具体类独立开来。

**如果子类的区别仅在于其对象的初始化方式，那么你可以使用该模式来减少子类的数量。别人创建这些子类的目的可能是为了创建特定类型的对象。**

在原型模式中，你可以使用一系列预生成的、各种类型的对象作为原型。

客户端不必根据需求对子类进行实例化，只需找到合适的原型并对其进行克隆即可。

# 实现方式

1. 创建原型接口，并在其中声明 `克隆` 方法。如果你已有类层次结构，则只需在其所有类中添加该方法即可。

2. 原型类必须另行定义一个以该类对象为参数的构造函数。构造函数必须复制参数对象中的所有成员变量值到新建实体中。如果你需要修改子类，则必须调用父类构造函数，让父类复制其私有成员变量值。

如果编程语言不支持方法重载，那么你可能需要定义一个特殊方法来复制对象数据。在构造函数中进行此类处理比较方便，因为它在调用 `new` 运算符后会马上返回结果对象。

3. 克隆方法通常只有一行代码：使用 `new` 运算符调用原型版本的构造函数。注意，每个类都必须显式重写克隆方法并使用自身类名调用 `new` 运算符。否则，克隆方法可能会生成父类的对象。

4. 你还可以创建一个中心化原型注册表，用于存储常用原型。

   你可以新建一个工厂类来实现注册表，或者在原型基类中添加一个获取原型的静态方法。该方法必须能够根据客户端代码设定的条件进行搜索。搜索条件可以是简单的字符串，或者是一组复杂的搜索参数。找到合适的原型后，注册表应对原型进行克隆，并将复制生成的对象返回给客户端。

   最后还要将对子类构造函数的直接调用替换为对原型注册表工厂方法的调用。

## ⚖ 优缺点

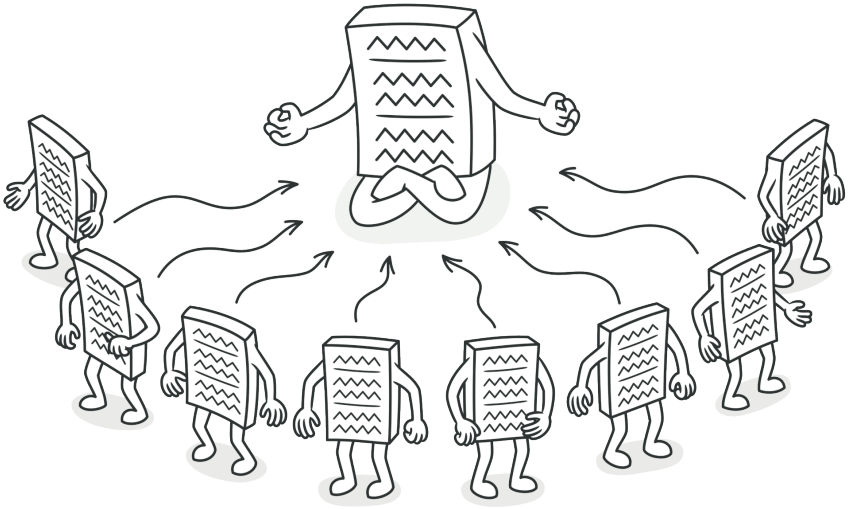
- ✓ 你可以克隆对象，而无需与它们所属的具体类相耦合。
- ✓ 你可以克隆预生成原型，避免反复运行初始化代码。
- ✓ 你可以更方便地生成复杂对象。

✓ 你可以用继承以外的方式来处理复杂对象的不同配置。

✕ 克隆包含循环引用的复杂对象可能会非常麻烦。

## ⇄ 与其他模式的关系

- 在许多设计工作的初期都会使用**工厂方法**（较为简单，而且可以更方便地通过子类进行定制），随后演化为使用**抽象工厂**、**原型**或**生成器**（更灵活但更加复杂）。
- **抽象工厂**模式通常基于一组**工厂方法**，但你也可以使用**原型**模式来生成这些类的方法。
- **原型**可用于保存**命令**的历史记录。
- 大量使用**组合**和**装饰**的设计通常可从对于**原型**的使用中获益。你可以通过该模式来复制复杂结构，而非从零开始重新构造。
- **原型**并不基于继承，因此没有继承的缺点。另一方面，原型需要对被复制对象进行复杂的初始化。**工厂方法**基于继承，但是它不需要初始化步骤。
- 有时候**原型**可以作为**备忘录**的一个简化版本，其条件是你需要在历史记录中存储的对象的状态比较简单，不需要链接其他外部资源，或者链接可以方便地重建。
- **抽象工厂**、**生成器**和**原型**都可以用**单例**来实现。



# 单例

亦称：单件模式、Singleton

**单例**是一种创建型设计模式，让你能够保证一个类只有一个实例，并提供一个访问该实例的全局节点。

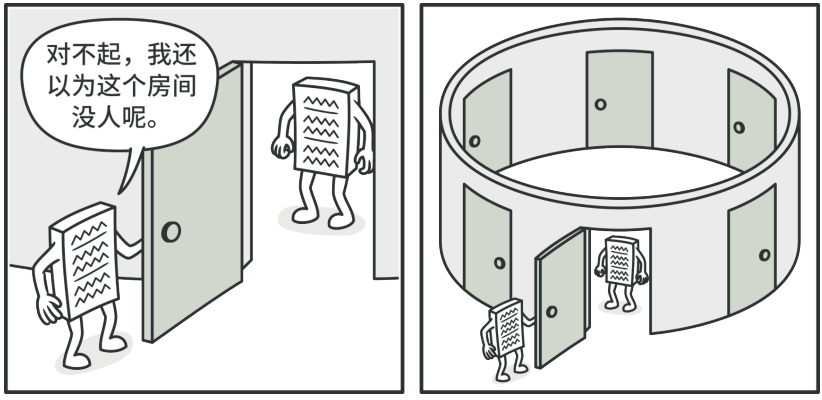
# ☹ 问题

单例模式同时解决了两个问题，所以违反了_单一职责原则_：

1. **保证一个类只有一个实例**。为什么会有人想要控制一个类所拥有的实例数量？最常见的原因是控制某些共享资源（例如数据库或文件）的访问权限。

   它的运作方式是这样的：如果你创建了一个对象，同时过一会儿后你决定再创建一个新对象，此时你会获得之前已创建的对象，而不是一个新对象。

   注意，普通构造函数无法实现上述行为，因为构造函数的设计决定了它**必须**总是返回一个新对象。




客户端甚至可能没有意识到它们一直都在使用同一个对象。

2. **为该实例提供一个全局访问节点**。还记得你（好吧，其实是我自己）用过的那些存储重要对象的全局变量吗？它们在使用上十分方便，但同时也非常不安全，因为任何代码都有可能覆盖掉那些变量的内容，从而引发程序崩溃。

   和全局变量一样，单例模式也允许在程序的任何地方访问特定对象。但是它可以保护该实例不被其他代码覆盖。

   还有一点：你不会希望解决同一个问题的代码分散在程序各处的。因此更好的方式是将其放在同一个类中，特别是当其他代码已经依赖这个类时更应该如此。

   如今，单例模式已经变得非常流行，以至于人们会将只解决上文描述中任意一个问题的东西称为单例。

## 😊解决方案

所有单例的实现都包含以下两个相同的步骤：

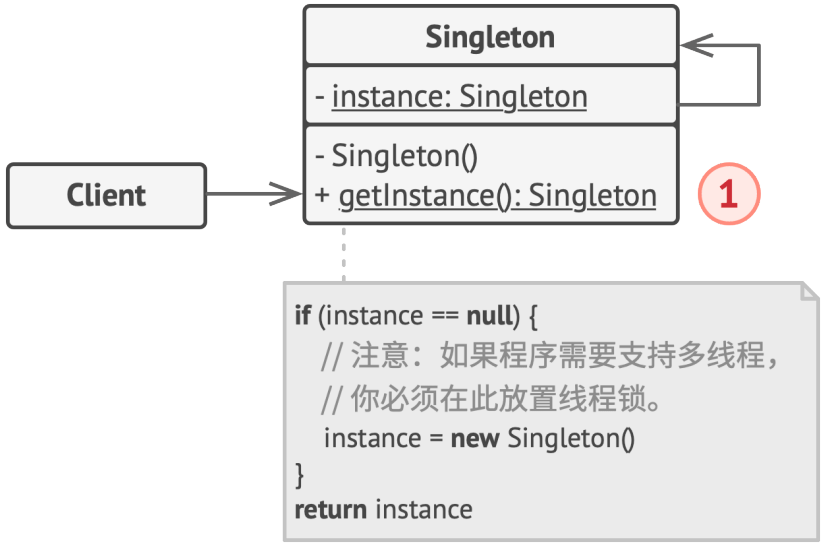
- 将默认构造函数设为私有，防止其他对象使用单例类的 `new` 运算符。
- 新建一个静态构建方法作为构造函数。该函数会“偷偷”调用私有构造函数来创建对象，并将其保存在一个静态成员变量中。此后所有对于该函数的调用都将返回这一缓存对象。

如果你的代码能够访问单例类，那它就能调用单例类的静态方法。无论何时调用该方法，它总是会返回相同的对象。

## 真实世界类比

政府是单例模式的一个很好的示例。一个国家只有一个官方政府。不管组成政府的每个人的身份是什么，“某政府”这一称谓总是鉴别那些掌权者的全局访问节点。

## 结构




1. **单例**（Singleton）类声明了一个名为 getInstance 获取实例 的静态方法来返回其所属类的一个相同实例。

单例的构造函数必须对客户端（Client）代码隐藏。调用`获取实例`方法必须是获取单例对象的唯一方式。

# 伪代码

在本例中，数据库连接类即是一个**单例**。

该类不提供公有构造函数，因此获取该对象的唯一方式是调用`获取实例`方法。该方法将缓存首次生成的对象，并为所有后续调用返回该对象。

```
// 数据库类会对`getInstance（获取实例）`方法进行定义以让客户端在程序各处
// 都能访问相同的数据库连接实例。
class Database is
  // 保存单例实例的成员变量必须被声明为静态类型。
  private static field instance: Database

  // 单例的构造函数必须永远是私有类型，以防止使用`new`运算符直接调用构
  // 造方法。
  private constructor Database() is
    // 部分初始化代码（例如到数据库服务器的实际连接）。
    // ...

  // 用于控制对单例实例的访问权限的静态方法。
  public static method getInstance() is
    if (Database.instance == null) then
      acquireThreadLock() and then
        // 确保在该线程等待解锁时，其他线程没有初始化该实例。
        if (Database.instance == null) then
```

```
            Database.instance = new Database()
        return Database.instance

    // 最后，任何单例都必须定义一些可在其实例上执行的业务逻辑。
    public method query(sql) is
        // 比如应用的所有数据库查询请求都需要通过该方法进行。因此，你可以
        // 在这里添加限流或缓冲逻辑。
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // 变量 `bar` 和 `foo` 中将包含同一个对象。
```

## 适合应用场景

**如果程序中的某个类对于所有客户端只有一个可用的实例，可以使用单例模式。**

单例模式禁止通过除特殊构建方法以外的任何方式来创建自身类的对象。该方法可以创建一个新对象，但如果该对象已经被创建，则返回已有的对象。

**如果你需要更加严格地控制全局变量，可以使用单例模式。**

单例模式与全局变量不同，它保证类只存在一个实例。除了单例类自己以外，无法通过任何方式替换缓存的实例。

请注意，你可以随时调整限制并设定生成单例实例的数量，只需修改 `获取实例` 方法，即 getInstance 中的代码即可实现。

## 实现方式

1. 在类中添加一个私有静态成员变量用于保存单例实例。
2. 声明一个公有静态构建方法用于获取单例实例。
3. 在静态方法中实现"延迟初始化"。该方法会在首次被调用时创建一个新对象，并将其存储在静态成员变量中。此后该方法每次被调用时都返回该实例。
4. 将类的构造函数设为私有。类的静态方法仍能调用构造函数，但是其他对象不能调用。
5. 检查客户端代码，将对单例的构造函数的调用替换为对其静态构建方法的调用。

## 优缺点

- ✓ 你可以保证一个类只有一个实例。
- ✓ 你获得了一个指向该实例的全局访问节点。

✓ 仅在首次请求单例对象时对其进行初始化。

✕ 违反了_单一职责原则_。该模式同时解决了两个问题。

✕ 单例模式可能掩盖不良设计，比如程序各组件之间相互了解过多等。

✕ 该模式在多线程环境下需要进行特殊处理，避免多个线程多次创建单例对象。

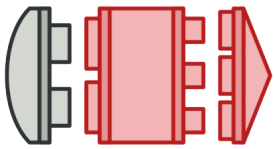
✕ 单例的客户端代码单元测试可能会比较困难，因为许多测试框架以基于继承的方式创建模拟对象。由于单例类的构造函数是私有的，而且绝大部分语言无法重写静态方法，所以你需要想出仔细考虑模拟单例的方法。要么干脆不编写测试代码，或者不使用单例模式。

## ⇄ 与其他模式的关系

- **外观**类通常可以转换为**单例**类，因为在大部分情况下一个外观对象就足够了。
- 如果你能将对象的所有共享状态简化为一个享元对象，那么**享元**就和**单例**类似了。但这两个模式有两个根本性的不同。
  1. 只会有一个单例实体，但是享元类可以有多个实体，各实体的内在状态也可以不同。
  2. 单例对象可以是可变的。享元对象是不可变的。
- **抽象工厂**、**生成器**和**原型**都可以用**单例**来实现。

# 结构型模式

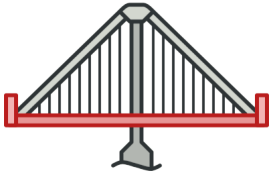
结构型模式介绍如何将对象和类组装成较大的结构，并同时保持结构的灵活和高效。



## 适配器

Adapter

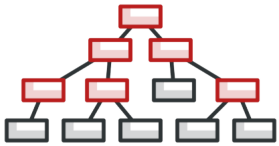
让接口不兼容的对象能够相互合作。



## 桥接

Bridge

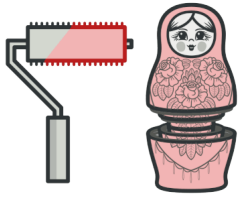
可将一个大类或一系列紧密相关的类拆分为抽象和实现两个独立的层次结构，从而能在开发时分别使用。



## 组合

Composite

你可以使用它将对象组合成树状结构，并且能像使用独立对象一样使用它们。



## 装饰

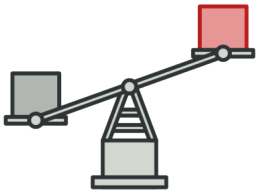
Decorator

允许你通过将对象放入包含行为的特殊封装对象中来为原对象绑定新的行为。

## 外观

Facade

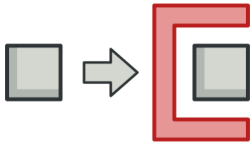
能为程序库、框架或其他复杂类提供一个简单的接口。



## 享元

Flyweight

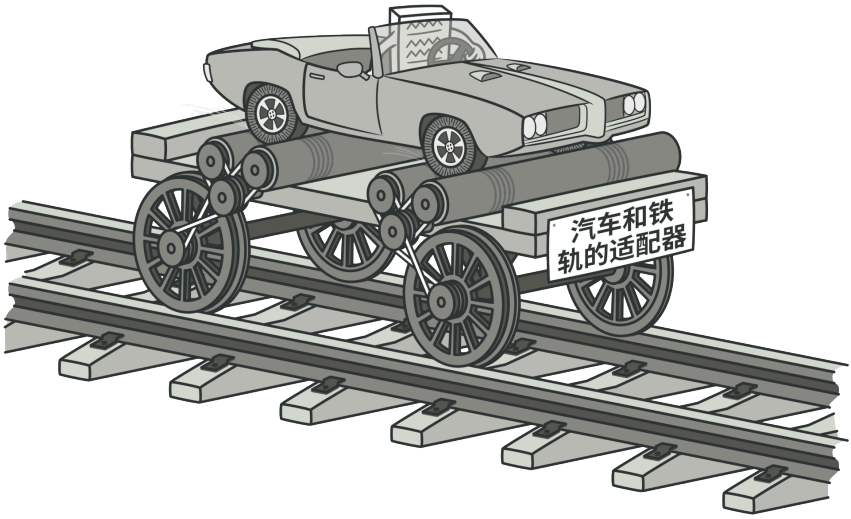
摒弃了在每个对象中保存所有数据的方式，通过共享多个对象所共有的相同状态，让你能在有限的内存容量中载入更多对象。



## 代理

Proxy

让你能够提供对象的替代品或其占位符。代理控制着对于原对象的访问，并允许在将请求提交给对象前后进行一些处理。




# 适配器

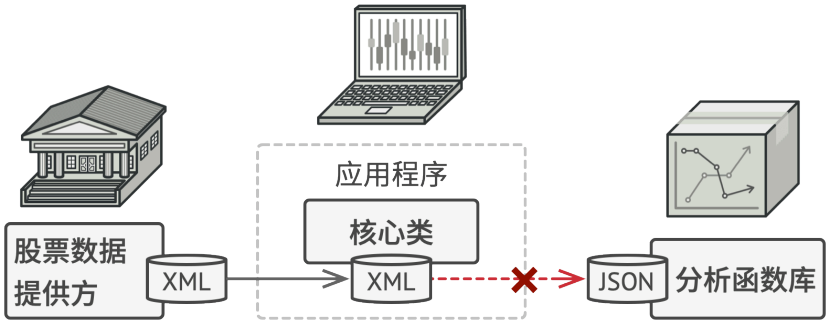
亦称：封装器模式、Wrapper、Adapter

**适配器**是一种结构型设计模式，
它能使接口不兼容的对象能够
相互合作。

# ☹ 问题

假如你正在开发一款股票市场监测程序，它会从不同来源下载 XML 格式的股票数据，然后向用户呈现出美观的图表。

在开发过程中，你决定在程序中整合一个第三方智能分析函数库。但是遇到了一个问题，那就是分析函数库只兼容 JSON 格式的数据。




你无法“直接”使用分析函数库，因为它所需的输入数据格式与你的程序不兼容。

你可以修改程序库来支持 XML。但是，这可能需要修改部分依赖该程序库的现有代码。甚至还有更糟糕的情况，你可能根本没有程序库的源代码，从而无法对其进行修改。

# ☺ 解决方案

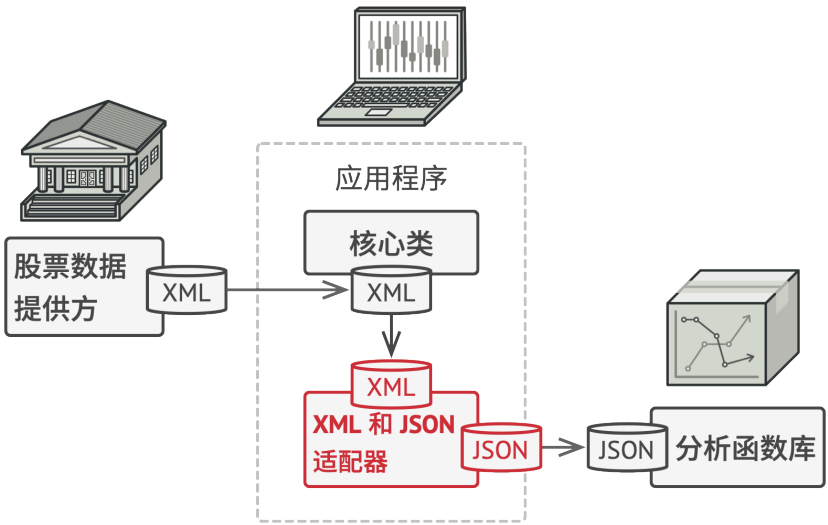
你可以创建一个**适配器**。这是一个特殊的对象，能够转换对象接口，使其能与其他对象进行交互。

适配器模式通过封装对象将复杂的转换过程隐藏于幕后。被封装的对象甚至察觉不到适配器的存在。例如，你可以使用一个将所有数据转换为英制单位（如英尺和英里）的适配器封装运行于米和千米单位制中的对象。

适配器不仅可以转换不同格式的数据，其还有助于采用不同接口的对象之间的合作。它的运作方式如下：

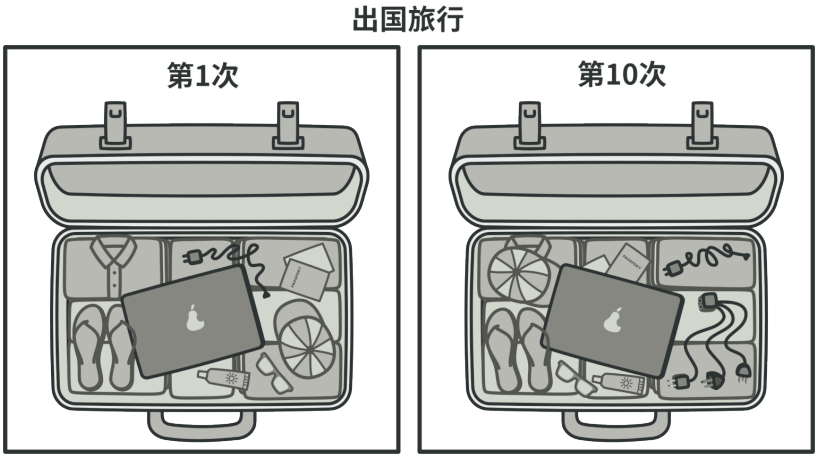
1. 适配器实现与其中一个现有对象兼容的接口。
2. 现有对象可以使用该接口安全地调用适配器方法。
3. 适配器方法被调用后将以另一个对象兼容的格式和顺序将请求传递给该对象。

有时你甚至可以创建一个双向适配器来实现双向转换调用。



让我们回到股票市场程序。为了解决数据格式不兼容的问题，你可以为分析函数库中的每个类创建将 XML 转换为 JSON 格式的适配器，然后让客户端仅通过这些适配器来与函数库进行交流。当某个适配器被调用时，它会将传入的 XML 数据转换为 JSON 结构，并将其传递给被封装分析对象的相应方法。

## 真实世界类比



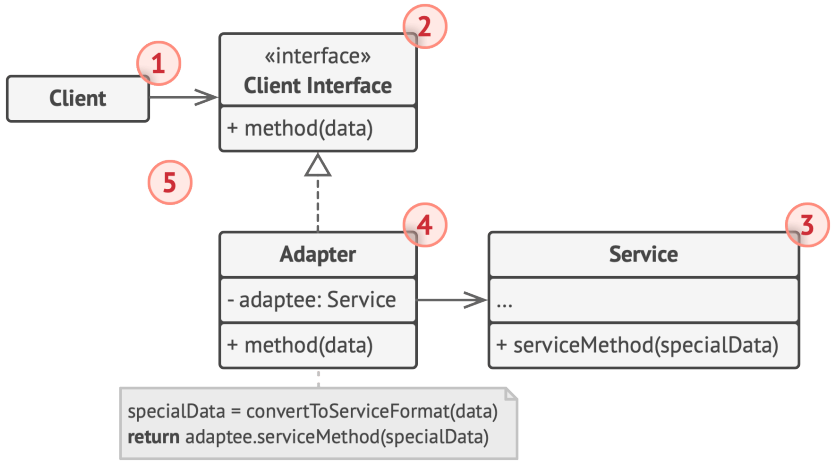

出国旅行前后的旅行箱。

如果你是第一次从美国到欧洲旅行，那么在给笔记本充电时可能会大吃一惊。不同国家的电源插头和插座标准不同。美国插头和德国插座不匹配。同时提供美国标准插座和欧洲标准插头的电源适配器可以解决你的难题。

# 结构

## 对象适配器

实现时使用了构成原则：适配器实现了其中一个对象的接口，并对另一个对象进行封装。所有流行的编程语言都可以实现适配器。




1. **客户端**（Client）是包含当前程序业务逻辑的类。

2. **客户端接口**（Client Interface）描述了其他类与客户端代码合作时必须遵循的协议。

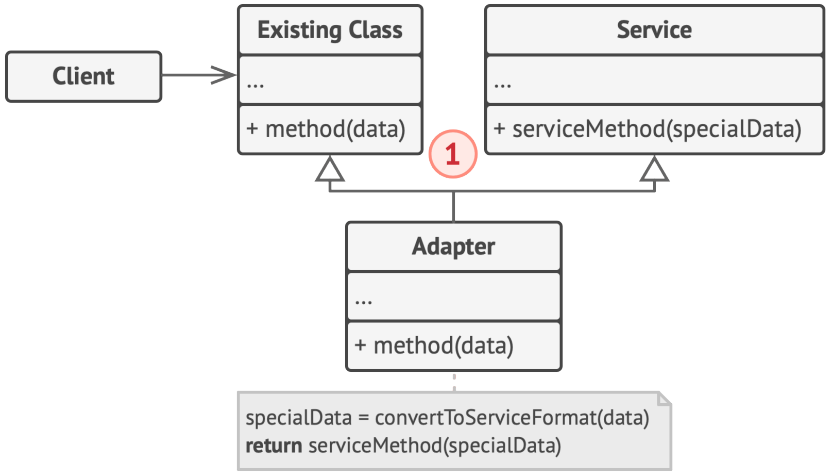
3. **服务**（Service）中有一些功能类（通常来自第三方或遗留系统）。客户端与其接口不兼容，因此无法直接调用其功能。

4. **适配器**（Adapter）是一个可以同时与客户端和服务交互的类：它在实现客户端接口的同时封装了服务对象。适配器接受客户端通过适配器接口发起的调用，并将其转换为适用于被封装服务对象的调用。

5. 客户端代码只需通过接口与适配器交互即可，无需与具体的适配器类耦合。因此，你可以向程序中添加新类型的适配器而无需修改已有代码。这在服务类的接口被更改或替换时很有用：你无需修改客户端代码就可以创建新的适配器类。

### 类适配器

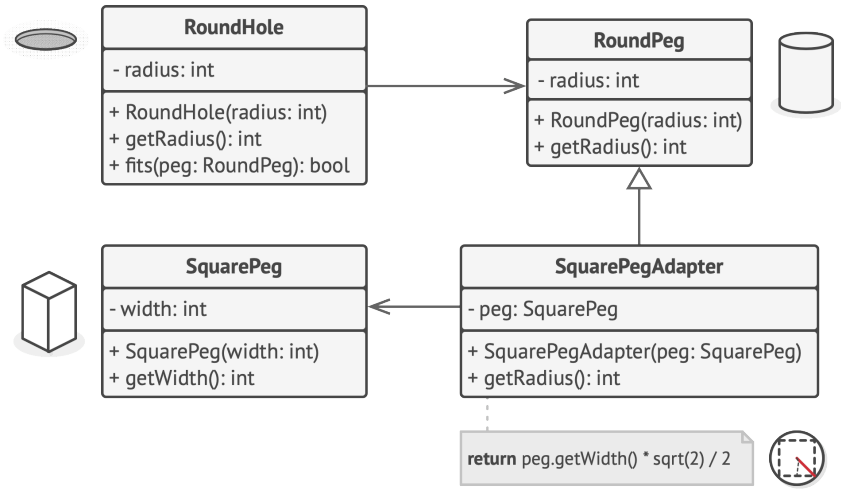
这一实现使用了继承机制：适配器同时继承两个对象的接口。请注意，这种方式仅能在支持多重继承的编程语言中实现，例如 C++。



1. **类适配器**不需要封装任何对象，因为它同时继承了客户端和服务的行为。适配功能在重写的方法中完成。最后生成的适配器可替代已有的客户端类进行使用。

# # 伪代码

下列**适配器**模式演示基于经典的“方钉和圆孔”问题。




让方钉适配圆孔。

适配器假扮成一个圆钉（RoundPeg），其半径等于方钉（SquarePeg）横截面对角线的一半（即能够容纳方钉的最小外接圆的半径）。

```
// 假设你有两个接口相互兼容的类：圆孔（RoundHole）和圆钉（RoundPeg）。
class RoundHole is
  constructor RoundHole(radius) { ... }

  method getRadius() is
    // 返回孔的半径。

  method fits(peg: RoundPeg) is
    return this.getRadius() >= peg.getRadius()

class RoundPeg is
  constructor RoundPeg(radius) { ... }

  method getRadius() is
    // 返回钉子的半径。


// 但还有一个不兼容的类：方钉（SquarePeg）。
class SquarePeg is
  constructor SquarePeg(width) { ... }

  method getWidth() is
    // 返回方钉的宽度。


// 适配器类让你能够将方钉放入圆孔中。它会对 RoundPeg 类进行扩展，以接收适
// 配器对象作为圆钉。
class SquarePegAdapter extends RoundPeg is
  // 在实际情况中，适配器中会包含一个 SquarePeg 类的实例。
  private field peg: SquarePeg

```

```
  constructor SquarePegAdapter(peg: SquarePeg) is
    this.peg = peg

  method getRadius() is
    // 适配器会假扮为一个圆钉，
    // 其半径刚好能与适配器实际封装的方钉搭配起来。
    return peg.getWidth() * Math.sqrt(2) / 2


// 客户端代码中的某个位置。
hole = new RoundHole(5)
rpeg = new RoundPeg(5)
hole.fits(rpeg) // true

small_sqpeg = new SquarePeg(5)
large_sqpeg = new SquarePeg(10)
hole.fits(small_sqpeg) // 此处无法编译（类型不一致）。

small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
hole.fits(small_sqpeg_adapter) // true
hole.fits(large_sqpeg_adapter) // false
```

## 适合应用场景

**当你希望使用某个类，但是其接口与其他代码不兼容时，可以使用适配器类。**

适配器模式允许你创建一个中间层类，其可作为代码与遗留类、第三方类或提供怪异接口的类之间的转换器。

**如果您需要复用这样一些类，他们处于同一个继承体系，并且他们又有了额外的一些共同的方法，但是这些共同的方法不是所有在这一继承体系中的子类所具有的共性。**

你可以扩展每个子类，将缺少的功能添加到新的子类中。但是，你必须在所有新子类中重复添加这些代码，这样会使得代码有**坏味道**。

将缺失功能添加到一个适配器类中是一种优雅得多的解决方案。然后你可以将缺少功能的对象封装在适配器中，从而动态地获取所需功能。如要这一点正常运作，目标类必须要有通用接口，适配器的成员变量应当遵循该通用接口。这种方式同**装饰**模式非常相似。

# 实现方式

1. 确保至少有两个类的接口不兼容：
   - 一个无法修改（通常是第三方、遗留系统或者存在众多已有依赖的类）的功能性服务类。
   - 一个或多个将受益于使用服务类的客户端类。
2. 声明客户端接口，描述客户端如何与服务交互。
3. 创建遵循客户端接口的适配器类。所有方法暂时都为空。

4. 在适配器类中添加一个成员变量用于保存对于服务对象的引用。通常情况下会通过构造函数对该成员变量进行初始化，但有时在调用其方法时将该变量传递给适配器会更方便。

5. 依次实现适配器类客户端接口的所有方法。适配器会将实际工作委派给服务对象，自身只负责接口或数据格式的转换。

6. 客户端必须通过客户端接口使用适配器。这样一来，你就可以在不影响客户端代码的情况下修改或扩展适配器。

## ⚖ 优缺点

✓ _单一职责原则_你可以将接口或数据转换代码从程序主要业务逻辑中分离。

✓ 开闭原则。只要客户端代码通过客户端接口与适配器进行交互，你就能在不修改现有客户端代码的情况下在程序中添加新类型的适配器。

✕ 代码整体复杂度增加，因为你需要新增一系列接口和类。有时直接更改服务类使其与其他代码兼容会更简单。

## ⇄ 与其他模式的关系

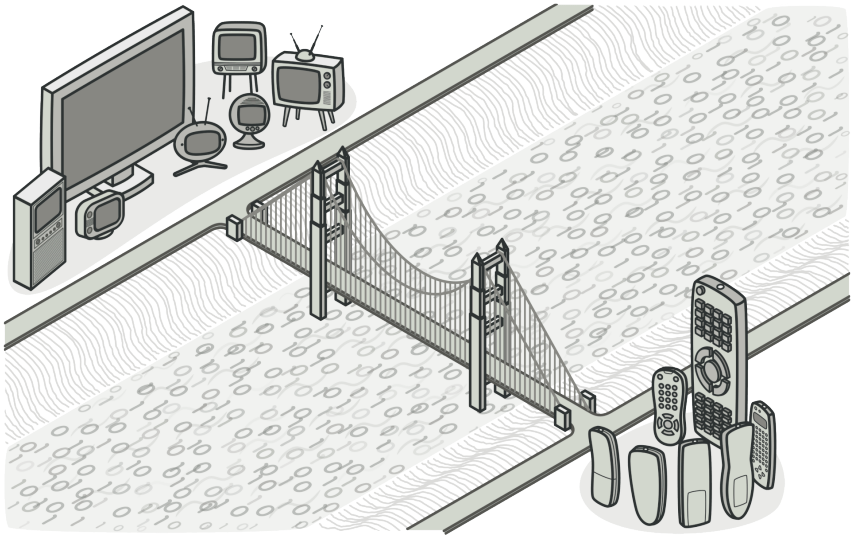
- **桥接**通常会于开发前期进行设计，使你能够将程序的各个部分独立开来以便开发。另一方面，**适配器**通常在已有程序中使用，让相互不兼容的类能很好地合作。

- **适配器**可以对已有对象的接口进行修改，**装饰**则能在不改变对象接口的前提下强化对象功能。此外，装饰还支持递归组合，适配器则无法实现。

- **适配器**能为被封装对象提供不同的接口，**代理**能为对象提供相同的接口，**装饰**则能为对象提供加强的接口。

- **外观**为现有对象定义了一个新接口，**适配器**则会试图运用已有的接口。适配器通常只封装一个对象，外观通常会作用于整个对象子系统上。

- **桥接**、**状态**和**策略**（在某种程度上包括**适配器**）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。



# 桥接

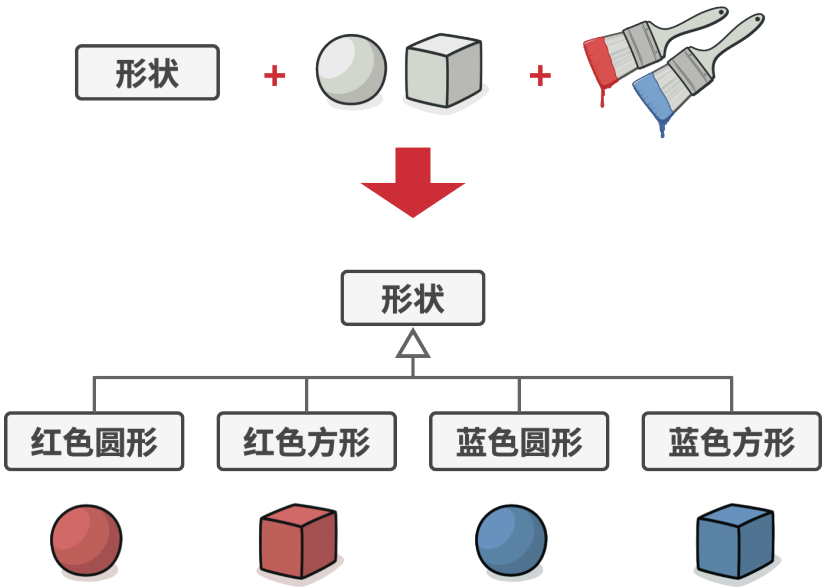
亦称：Bridge

**桥接**是一种结构型设计模式，
可将一个大类或一系列紧密
相关的类拆分为抽象和实现
两个独立的层次结构，从而
能在开发时分别使用。

# ☹ 问题

**抽象？实现？**听上去挺吓人？让我们慢慢来，先考虑一个简单的例子。

假如你有一个几何 形状 Shape 类，从它能扩展出两个子类： 圆形 Circle 和 方形 Square。你希望对这样的类层次结构进行扩展以使其包含颜色，所以你打算创建名为 红色 Red 和 蓝色 Blue 的形状子类。但是，由于你已有两个子类，所以总共需要创建四个类才能覆盖所有组合，例如 蓝色圆形 BlueCircle 和 红色方形 RedSquare。



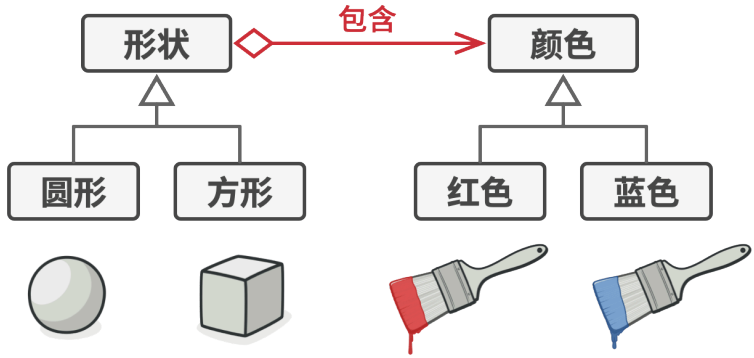

所有组合类的数量将以几何级数增长。

在层次结构中新增形状和颜色将导致代码复杂程度指数增长。例如添加三角形状，你需要新增两个子类，也就是每种颜色一个；此后新增一种新颜色需要新增三个子类，即每种形状一个。如此以往，情况会越来越糟糕。

## 😊解决方案

问题的根本原因是我们试图在两个独立的维度——形状与颜色——上扩展形状类。这在处理类继承时是很常见的问题。

桥接模式通过将继承改为组合的方式来解决这个问题。具体来说，就是抽取其中一个维度并使之成为独立的类层次，这样就可以在初始类中引用这个新层次的对象，从而使得一个类不必拥有所有的状态和行为。




将一个类层次转化为多个相关的类层次，避免单个类层次的失控。

根据该方法，我们可以将颜色相关的代码抽取到拥有 红色 和 蓝色 两个子类的颜色类中，然后在 形状 类中添加一个指向某一颜色对象的引用成员变量。现在，形状类可以将所有与颜色相关的工作委派给连入的颜色对象。这样的引用就成为了 形状 和 颜色 之间的桥梁。此后，新增颜色将不再需要修改形状的类层次，反之亦然。

## 抽象部分和实现部分

设计模式四人组的著作[1] 在桥接定义中提出了抽象部分和实现部分两个术语。我觉得这些术语过于学术了，反而让模式看上去比实际情况更加复杂。在介绍过形状和颜色的简单例子后，我们来看看四人组著作中让人望而生畏的词语的含义。

抽象部分（也被称为接口）是一些实体的高阶控制层。该层自身不完成任何具体的工作，它需要将工作委派给实现部分层（也被称为平台）。

注意，这里提到的内容与编程语言中的接口或抽象类无关。它们并不是一回事。

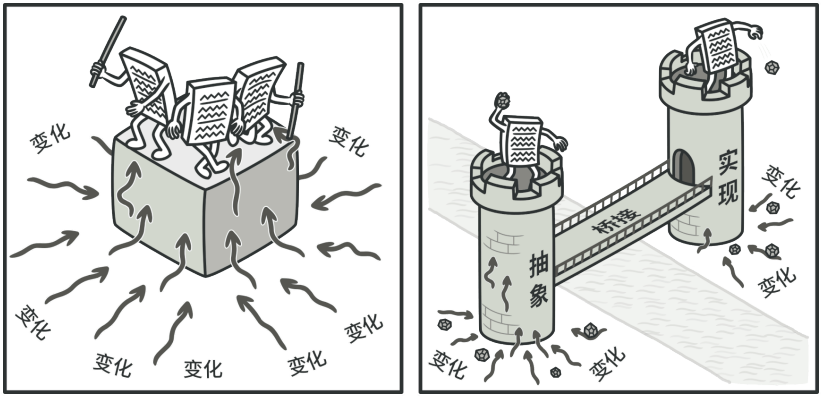
在实际的程序中，抽象部分是图形用户界面（GUI），而实现部分则是底层操作系统代码（API），GUI 层调用 API 层来对用户的各种操作做出响应。

1. “四人组（Gang of Four）” 是设计模式著作：《设计模式：可复用面向对象软件的基础》的四位作者的昵称，可点击连接：**https://refactoringguru.cn/gof-book** 查看全书

一般来说，你可以在两个独立方向上扩展这种应用：

- 开发多个不同的 GUI（例如面向普通用户和管理员进行分别配置）
- 支持多个不同的 API（例如，能够在 Windows、Linux 和 macOS 上运行该程序）。

在最糟糕的情况下，程序可能会是一团乱麻，其中包含数百种条件语句，连接着代码各处不同种类的 GUI 和各种 API。



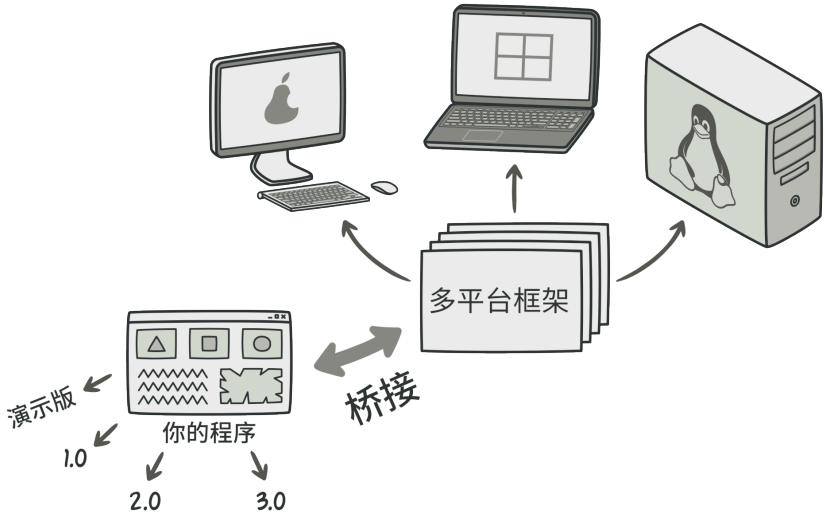

在庞杂的代码中，即使是很小的改动都非常难以完成，因为你必须要在整体上对代码有充分的理解。而在较小且定义明确的模块中，进行修改则要容易得多。

你可以将特定接口-平台的组合代码抽取到独立的类中，以在混乱中建立一些秩序。但是，你很快会发现这种类的数量很

多。类层次将以指数形式增长，因为每次添加一个新的 GUI 或支持一种新的 API 都需要创建更多的类。

让我们试着用桥接模式来解决这个问题。该模式建议将类拆分为两个类层次结构：

- 抽象部分：程序的 GUI 层。
- 实现部分：操作系统的 API。

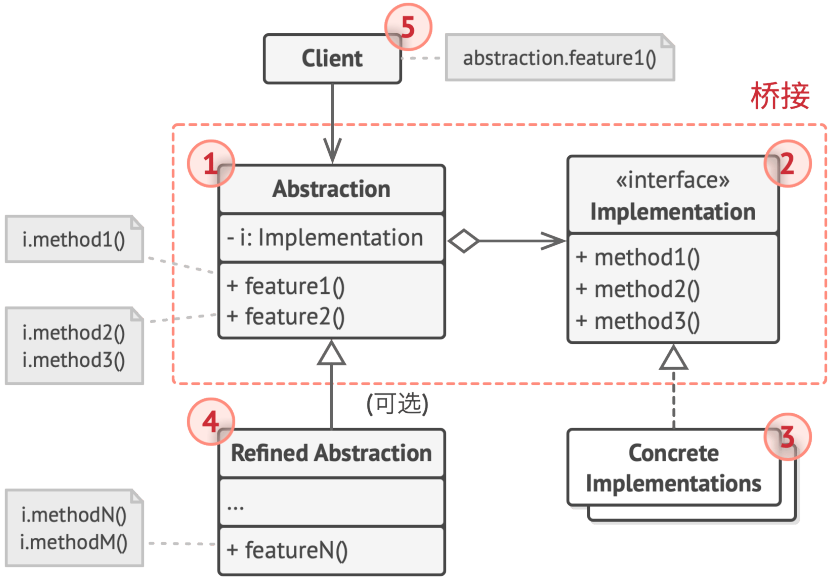



创建跨平台应用程序的一种方法

抽象对象控制程序的外观，并将真实工作委派给连入的实现对象。不同的实现只要遵循相同的接口就可以互换，使同一 GUI 可在 Windows 和 Linux 下运行。

最后的结果是：你无需改动与 API 相关的类就可以修改 GUI 类。此外如果想支持一个新的操作系统，只需在实现部分层次中创建一个子类即可。

## 结构




1. **抽象部分**（Abstraction）提供高层控制逻辑，依赖于完成底层实际工作的实现对象。

2. **实现部分**（Implementation）为所有具体实现声明通用接口。抽象部分仅能通过在这里声明的方法与实现对象交互。

抽象部分可以列出和实现部分一样的方法，但是抽象部分通常声明一些复杂行为，这些行为依赖于多种由实现部分声明的原语操作。

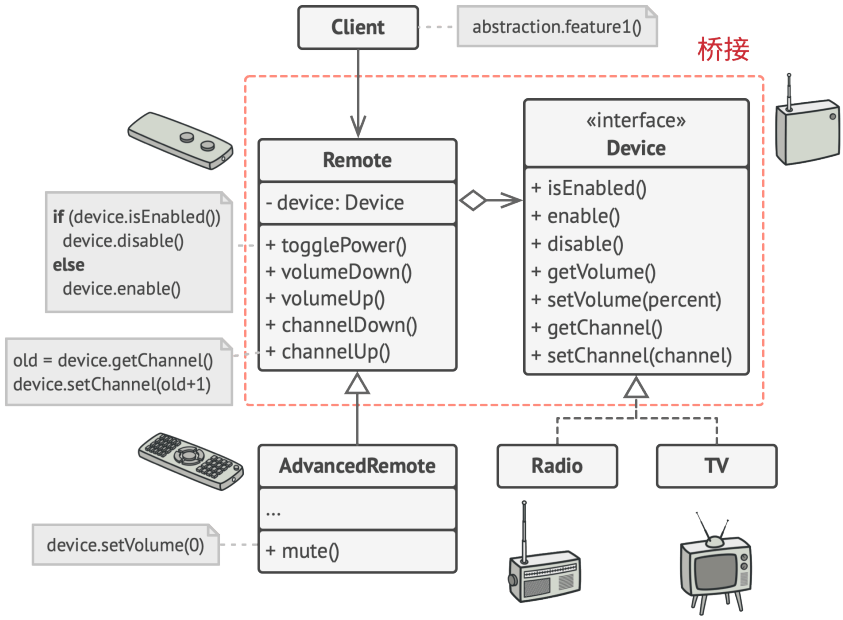
3. **具体实现**（Concrete Implementations）中包括特定于平台的代码。

4. **精确抽象**（Refined Abstraction）提供控制逻辑的变体。与其父类一样，它们通过通用实现接口与不同的实现进行交互。

5. 通常情况下，**客户端**（Client）仅关心如何与抽象部分合作。但是，客户端需要将抽象对象与一个实现对象连接起来。

# # 伪代码

示例演示了**桥接**模式如何拆分程序中同时管理设备及其遥控器的庞杂代码。`设备 Device`类作为实现部分，而`遥控器 Remote`类则作为抽象部分。




最初类层次结构被拆分为两个部分：设备和遥控器。

遥控器基类声明了一个指向设备对象的引用成员变量。所有遥控器通过通用设备接口与设备进行交互，使得同一个遥控器可以支持不同类型的设备。

你可以开发独立于设备类的遥控器类，只需新建一个遥控器子类即可。例如，基础遥控器可能只有两个按钮，但你可在其基础上扩展新功能，比如额外的一节电池或一块触摸屏。

客户端代码通过遥控器构造函数将特定种类的遥控器与设备对象连接起来。

```
// “抽象部分”定义了两个类层次结构中“控制”部分的接口。它管理着一个指向“实
// 现部分”层次结构中对象的引用，并会将所有真实工作委派给该对象。
class RemoteControl is
  protected field device: Device
  constructor RemoteControl(device: Device) is
    this.device = device
  method togglePower() is
    if (device.isEnabled()) then
      device.disable()
    else
      device.enable()
  method volumeDown() is
    device.setVolume(device.getVolume() - 10)
  method volumeUp() is
    device.setVolume(device.getVolume() + 10)
  method channelDown() is
    device.setChannel(device.getChannel() - 1)
  method channelUp() is
    device.setChannel(device.getChannel() + 1)


// 你可以独立于设备类的方式从抽象层中扩展类。
class AdvancedRemoteControl extends RemoteControl is
  method mute() is
    device.setVolume(0)


// “实现部分”接口声明了在所有具体实现类中通用的方法。它不需要与抽象接口相
// 匹配。实际上，这两个接口可以完全不一样。通常实现接口只提供原语操作，而
// 抽象接口则会基于这些操作定义较高层次的操作。
interface Device is
  method isEnabled()
```

```
    method enable()
    method disable()
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)


// 所有设备都遵循相同的接口。
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// 客户端代码中的某个位置。
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()

radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

## 适合应用场景

**如果你想要拆分或重组一个具有多重功能的庞杂类（例如能与多个数据库服务器进行交互的类），可以使用桥接模式。**

类的代码行数越多，弄清其运作方式就越困难，对其进行修改所花费的时间就越长。一个功能上的变化可能需要在整个类范围内进行修改，而且常常会产生错误，甚至还会有一些严重的副作用。

桥接模式可以将庞杂类拆分为几个类层次结构。此后，你可以修改任意一个类层次结构而不会影响到其他类层次结构。这种方法可以简化代码的维护工作，并将修改已有代码的风险降到最低。

**如果你希望在几个独立维度上扩展一个类，可使用该模式。**

桥接建议将每个维度抽取为独立的类层次。初始类将相关工作委派给属于对应类层次的对象，无需自己完成所有工作。

**如果你需要在运行时切换不同实现方法，可使用桥接模式。**

当然并不是说一定要实现这一点，桥接模式可替换抽象部分中的实现对象，具体操作就和给成员变量赋新值一样简单。

顺便提一句，最后一点是很多人混淆桥接模式和**策略**模式的主要原因。记住，设计模式并不仅是一种对类进行组织的方式，它还能用于沟通意图和解决问题。

# 实现方式

1. 明确类中独立的维度。独立的概念可能是：抽象/平台，域/基础设施，前端/后端或接口/实现。
2. 了解客户端的业务需求，并在抽象基类中定义它们。
3. 确定在所有平台上都可执行的业务。并在通用实现接口中声明抽象部分所需的业务。
4. 为你域内的所有平台创建实现类，但需确保它们遵循实现部分的接口。
5. 在抽象类中添加指向实现类型的引用成员变量。抽象部分会将大部分工作委派给该成员变量所指向的实现对象。
6. 如果你的高层逻辑有多个变体，则可通过扩展抽象基类为每个变体创建一个精确抽象。
7. 客户端代码必须将实现对象传递给抽象部分的构造函数才能使其能够相互关联。此后，客户端只需与抽象对象进行交互，无需和实现对象打交道。

# 优缺点

- ✓ 你可以创建与平台无关的类和程序。
- ✓ 客户端代码仅与高层抽象部分进行互动，不会接触到平台的详细信息。

✓ 开闭原则。你可以新增抽象部分和实现部分，且它们之间不会相互影响。

✓ 单一职责原则。抽象部分专注于处理高层逻辑，实现部分处理平台细节。

✕ 对高内聚的类使用该模式可能会让代码更加复杂。

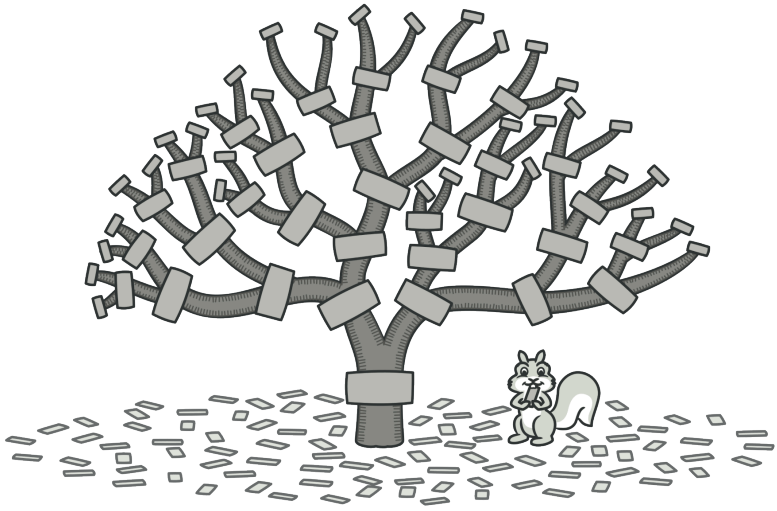
## ⇄ 与其他模式的关系

- **桥接**通常会于开发前期进行设计，使你能够将程序的各个部分独立开来以便开发。另一方面，**适配器**通常在已有程序中使用，让相互不兼容的类能很好地合作。

- **桥接**、**状态**和**策略**（在某种程度上包括**适配器**）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。

- 你可以将**抽象工厂**和**桥接**搭配使用。如果由桥接定义的抽象只能与特定实现合作，这一模式搭配就非常有用。在这种情况下，抽象工厂可以对这些关系进行封装，并且对客户端代码隐藏其复杂性。

- 你可以结合使用**生成器**和**桥接**模式：主管类负责抽象工作，各种不同的生成器负责实现工作。



# 组合

亦称：对象树、Object Tree、Composite

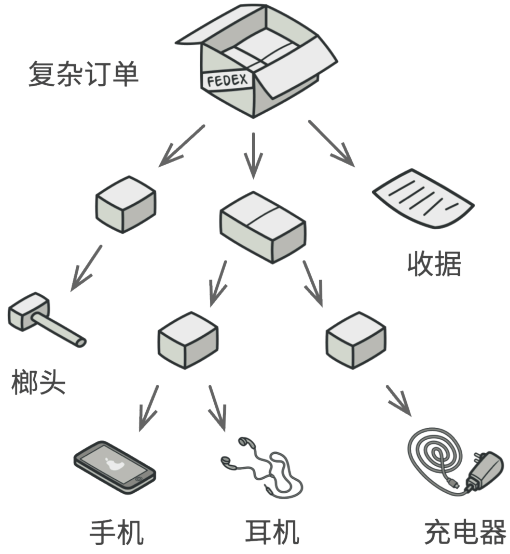
**组合**是一种结构型设计模式，
你可以使用它将对象组合成
树状结构，并且能像使用独
立对象一样使用它们。

# 问题

如果应用的核心模型能用树状结构表示，在应用中使用组合模式才有价值。

例如，你有两类对象：`产品`和`盒子`。一个盒子中可以包含多个`产品`或者几个较小的`盒子`。这些小`盒子`中同样可以包含一些`产品`或更小的`盒子`，以此类推。

假设你希望在这些类的基础上开发一个定购系统。订单中可以包含无包装的简单产品，也可以包含装满产品的盒子……以及其他盒子。此时你会如何计算每张订单的总价格呢？



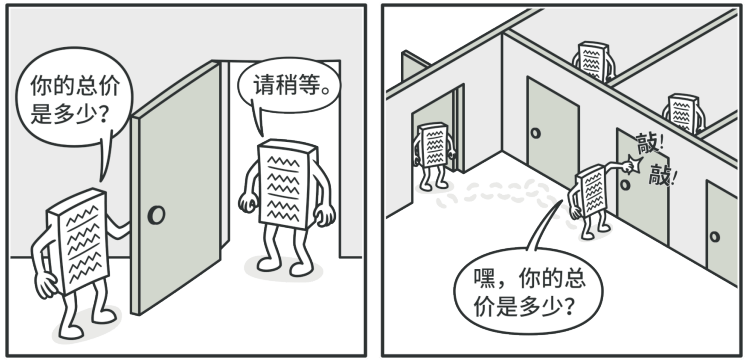

订单中可能包括各种产品，这些产品放置在盒子中，然后又被放入一层又一层更大的盒子中。整个结构看上去像是一棵倒过来的树。

你可以尝试直接计算：打开所有盒子，找到每件产品，然后计算总价。这在真实世界中或许可行，但在程序中，你并不能简单地使用循环语句来完成该工作。你必须事先知道所有产品和盒子的类别，所有盒子的嵌套层数以及其他繁杂的细节信息。因此，直接计算极不方便，甚至完全不可行。

## 😊 解决方案

组合模式建议使用一个通用接口来与产品和盒子进行交互，并且在该接口中声明一个计算总价的方法。

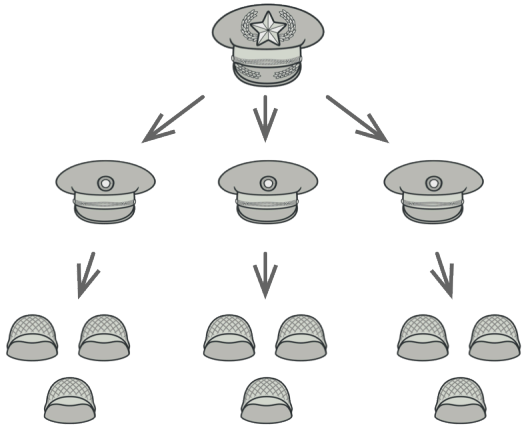
那么方法该如何设计呢？对于一个产品，该方法直接返回其价格；对于一个盒子，该方法遍历盒子中的所有项目，询问每个项目的价格，然后返回该盒子的总价格。如果其中某个项目是小一号的盒子，那么当前盒子也会遍历其中的所有项目，以此类推，直到计算出所有内部组成部分的价格。你甚至可以在盒子的最终价格中增加额外费用，作为该盒子的包装费用。




组合模式以递归方式处理对象树中的所有项目

该方式的最大优点在于你无需了解构成树状结构的对象的具体类。你也无需了解对象是简单的产品还是复杂的盒子。你只需调用通用接口以相同的方式对其进行处理即可。当你调用该方法后，对象会将请求沿着树结构传递下去。

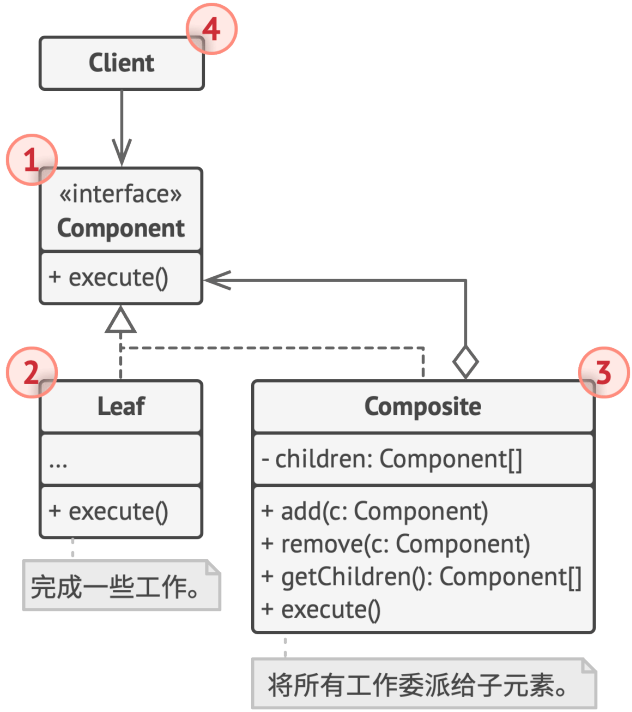
## 真实世界类比



部队结构的例子。

大部分国家的军队都采用层次结构管理。每支部队包括几个师，师由旅构成，旅由团构成，团可以继续划分为排。最后，每个排由一小队实实在在的士兵组成。军事命令由最高层下达，通过每个层级传递，直到每位士兵都知道自己应该服从的命令。

# 结构




1. **组件**（Component）接口描述了树中简单项目和复杂项目所共有的操作。

2. **叶节点**（Leaf）是树的基本结构，它不包含子项目。

   一般情况下，叶节点最终会完成大部分的实际工作，因为它们无法将工作指派给其他部分。

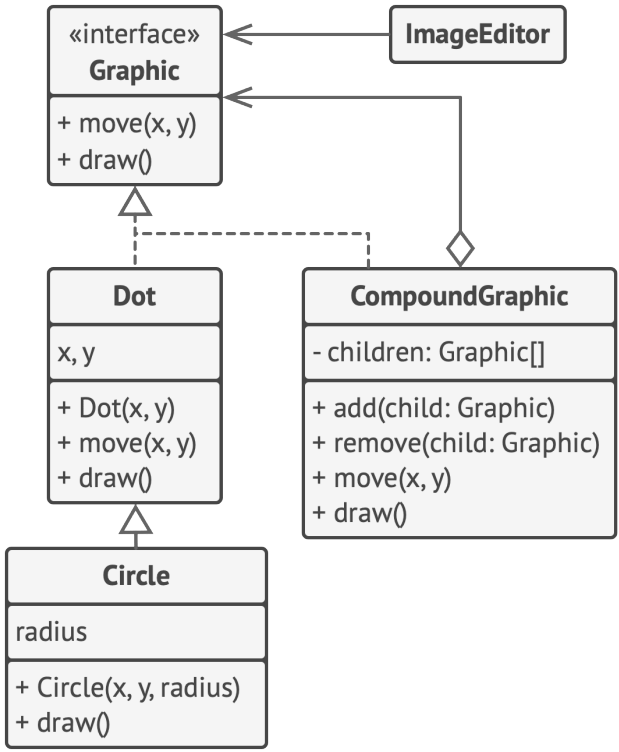
3. **容器**（Container）——又名“组合（Composite）”——是包含叶节点或其他容器等子项目的单位。容器不知道其子项目所属的具体类，它只通过通用的组件接口与其子项目交互。

容器接收到请求后会将工作分配给自己的子项目，处理中间结果，然后将最终结果返回给客户端。

4. **客户端**（Client）通过组件接口与所有项目交互。因此，客户端能以相同方式与树状结构中的简单或复杂项目交互。

# 伪代码

在本例中，我们将借助**组合**模式帮助你在图形编辑器中实现一系列的几何图形。




几何形状编辑器示例。

组合图形 CompoundGraphic 是一个容器，它可以由多个包括容器在内的子图形构成。组合图形与简单图形拥有相同的方法。但是，组合图形自身并不完成具体工作，而是将请求递归地传递给自己的子项目，然后“汇总”结果。

通过所有图形类所共有的接口，客户端代码可以与所有图形互动。因此，客户端不知道与其交互的是简单图形还是组合图形。客户端可以与非常复杂的对象结构进行交互，而无需与组成该结构的实体类紧密耦合。

```
// 组件接口会声明组合中简单和复杂对象的通用操作。
interface Graphic is
  method move(x, y)
  method draw()

// 叶节点类代表组合的终端对象。叶节点对象中不能包含任何子对象。叶节点对象
// 通常会完成实际的工作，组合对象则仅会将工作委派给自己的子部件。
class Dot implements Graphic is
  field x, y

  constructor Dot(x, y) { ... }

  method move(x, y) is
    this.x += x, this.y += y

  method draw() is
    // 在坐标位置(X,Y)处绘制一个点。

// 所有组件类都可以扩展其他组件。
```

```
class Circle extends Dot is
  field radius

  constructor Circle(x, y, radius) { ... }

  method draw() is
    // 在坐标位置(X,Y)处绘制一个半径为 R 的圆。

// 组合类表示可能包含子项目的复杂组件。组合对象通常会将实际工作委派给子项
// 目，然后“汇总”结果。
class CompoundGraphic implements Graphic is
  field children: array of Graphic

  // 组合对象可在其项目列表中添加或移除其他组件（简单的或复杂的皆可）。
  method add(child: Graphic) is
    // 在子项目数组中添加一个子项目。

  method remove(child: Graphic) is
    // 从子项目数组中移除一个子项目。

  method move(x, y) is
    foreach (child in children) do
      child.move(x, y)

  // 组合会以特定的方式执行其主要逻辑。它会递归遍历所有子项目，并收集和
  // 汇总其结果。由于组合的子项目也会将调用传递给自己的子项目，以此类推，
  // 最后组合将会完成整个对象树的遍历工作。
  method draw() is
    // 1. 对于每个子部件：
    //     - 绘制该部件。
    //     - 更新边框坐标。
    // 2. 根据边框坐标绘制一个虚线长方形。
```

```
52
53
54  // 客户端代码会通过基础接口与所有组件进行交互。这样一来，客户端代码便可同
55  // 时支持简单叶节点组件和复杂组件。
56  class ImageEditor is
57    field all: array of Graphic
58
59    method load() is
60      all = new CompoundGraphic()
61      all.add(new Dot(1, 2))
62      all.add(new Circle(5, 3, 10))
63      // ...
64
65    // 将所需组件组合为复杂的组合组件。
66    method groupSelected(components: array of Graphic) is
67      group = new CompoundGraphic()
68      foreach (component in components) do
69        group.add(component)
70        all.remove(component)
71      all.add(group)
72      // 所有组件都将被绘制。
73      all.draw()
```

## 适合应用场景

**如果你需要实现树状对象结构，可以使用组合模式。**

组合模式为你提供了两种共享公共接口的基本元素类型：简单叶节点和复杂容器。 容器中可以包含叶节点和其他容器。这使得你可以构建树状嵌套递归对象结构。

**如果你希望客户端代码以相同方式处理简单和复杂元素，可以使用该模式。**

组合模式中定义的所有元素共用同一个接口。在这一接口的帮助下，客户端不必在意其所使用的对象的具体类。

## 实现方式

1. 确保应用的核心模型能够以树状结构表示。尝试将其分解为简单元素和容器。记住，容器必须能够同时包含简单元素和其他容器。

2. 声明组件接口及其一系列方法，这些方法对简单和复杂元素都有意义。

3. 创建一个叶节点类表示简单元素。程序中可以有多个不同的叶节点类。

4. 创建一个容器类表示复杂元素。在该类中，创建一个数组成员变量来存储对于其子元素的引用。该数组必须能够同时保存叶节点和容器，因此请确保将其声明为组合接口类型。

   实现组件接口方法时，记住容器应该将大部分工作交给其子元素来完成。

5. 最后，在容器中定义添加和删除子元素的方法。

记住，这些操作可在组件接口中声明。这将会违反_接口隔离原则_，因为叶节点类中的这些方法为空。但是，这可以让客户端无差别地访问所有元素，即使是组成树状结构的元素。

## ⚖ 优缺点

✓ 你可以利用多态和递归机制更方便地使用复杂树结构。

✓ 开闭原则。无需更改现有代码，你就可以在应用中添加新元素，使其成为对象树的一部分。

✕ 对于功能差异较大的类，提供公共接口或许会有困难。在特定情况下，你需要过度一般化组件接口，使其变得令人难以理解。

## ⇄ 与其他模式的关系

- **桥接**、**状态**和**策略**（在某种程度上包括**适配器**）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。

- 你可以在创建复杂**组合**树时使用**生成器**，因为这可使其构造步骤以递归的方式运行。

- **责任链**通常和**组合**模式结合使用。在这种情况下，叶组件接收到请求后，可以将请求沿包含全体父组件的链一直传递至对象树的底部。

- 你可以使用**迭代器**来遍历**组合**树。

- 你可以使用**访问者**对整个**组合**树执行操作。

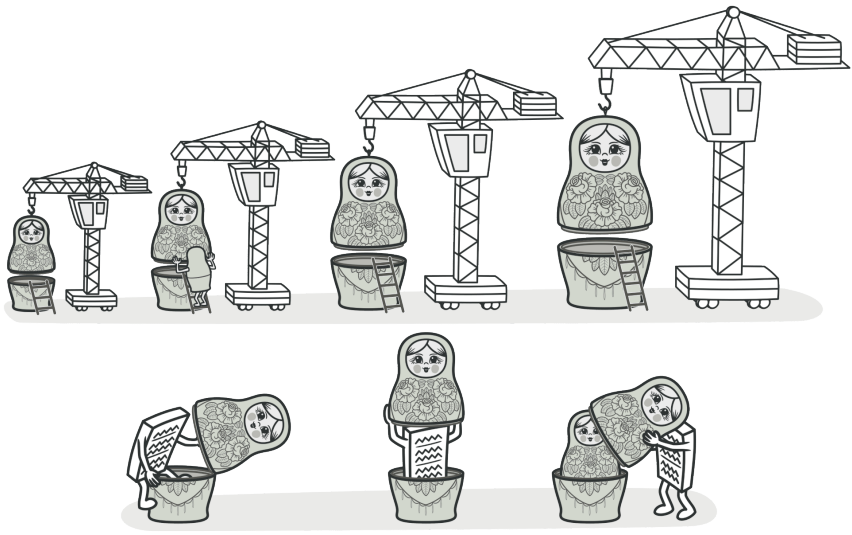
- 你可以使用**享元**实现**组合**树的共享叶节点以节省内存。

- **组合**和**装饰**的结构图很相似，因为两者都依赖递归组合来组织无限数量的对象。

  装饰类似于组合，但其只有一个子组件。此外还有一个明显不同：装饰为被封装对象添加了额外的职责，组合仅对其子节点的结果进行了“求和”。

  但是，模式也可以相互合作：你可以使用装饰来扩展组合树中特定对象的行为。

- 大量使用**组合**和**装饰**的设计通常可从对于**原型**的使用中获益。你可以通过该模式来复制复杂结构，而非从零开始重新构造。



# 装饰

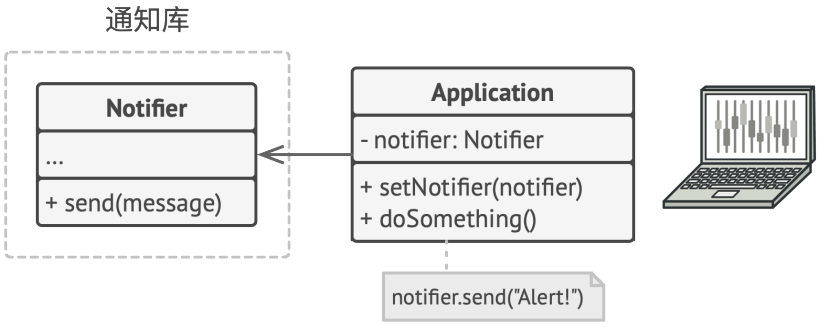
亦称：装饰者模式、装饰器模式、Wrapper、Decorator

**装饰**是一种结构型设计模式，
允许你通过将对象放入包含
行为的特殊封装对象中来为
原对象绑定新的行为。

# ☹ 问题

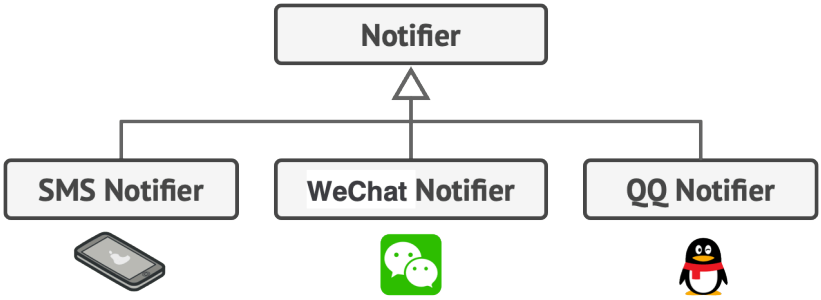
假设你正在开发一个提供通知功能的库，其他程序可使用它向用户发送关于重要事件的通知。

库的最初版本基于通知器 Notifier 类，其中只有很少的几个成员变量，一个构造函数和一个 send 发送 方法。该方法可以接收来自客户端的消息参数，并将该消息发送给一系列的邮箱，邮箱列表则是通过构造函数传递给通知器的。作为客户端的第三方程序仅会创建和配置通知器对象一次，然后在有重要事件发生时对其进行调用。




程序可以使用通知器类向预定义的邮箱发送重要事件通知。

此后某个时刻，你会发现库的用户希望使用除邮件通知之外的功能。许多用户会希望接收关于紧急事件的手机短信，还有些用户希望在微信上接收消息，而公司用户则希望在 QQ 上接收消息。

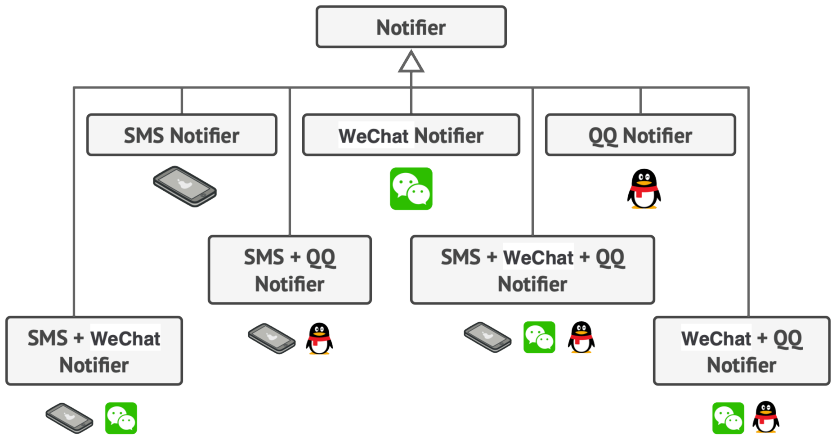



每种通知类型都将作为通知器的一个子类得以实现。

这有什么难的呢？首先扩展`通知器`类，然后在新的子类中加入额外的通知方法。现在客户端要对所需通知形式的对应类进行初始化，然后使用该类发送后续所有的通知消息。

但是很快有人会问：“为什么不同时使用多种通知形式呢？如果房子着火了，你大概会想在所有渠道中都收到相同的消息吧。”

你可以尝试创建一个特殊子类来将多种通知方法组合在一起以解决该问题。但这种方式会使得代码量迅速膨胀，不仅仅是程序库代码，客户端代码也会如此。




子类组合数量爆炸。

你必须找到其他方法来规划通知类的结构，否则它们的数量会在不经意之间打破吉尼斯纪录。

## 😊解决方案

当你需要更改一个对象的行为时，第一个跳入脑海的想法就是扩展它所属的类。但是，你不能忽视继承可能引发的几个严重问题。

- 继承是静态的。你无法在运行时更改已有对象的行为，只能使用由不同子类创建的对象来替代当前的整个对象。

- 子类只能有一个父类。大部分编程语言不允许一个类同时继承多个类的行为。

其中一种方法是用**聚合**或**组合**[1]，而不是**继承**。两者的工作方式几乎一模一样：一个对象**包含**指向另一个对象的引用，并将部分工作委派给引用对象；继承中的对象则继承了父类的行为，它们自己**能够**完成这些工作。

你可以使用这个新方法来轻松替换各种连接的“小帮手”对象，从而能在运行时改变容器的行为。一个对象可以使用多个类的行为，包含多个指向其他对象的引用，并将各种工作委派给引用对象。

聚合（或组合）组合是许多设计模式背后的关键原则（包括装饰在内）。记住这一点后，让我们继续关于模式的讨论。


继承与聚合的对比

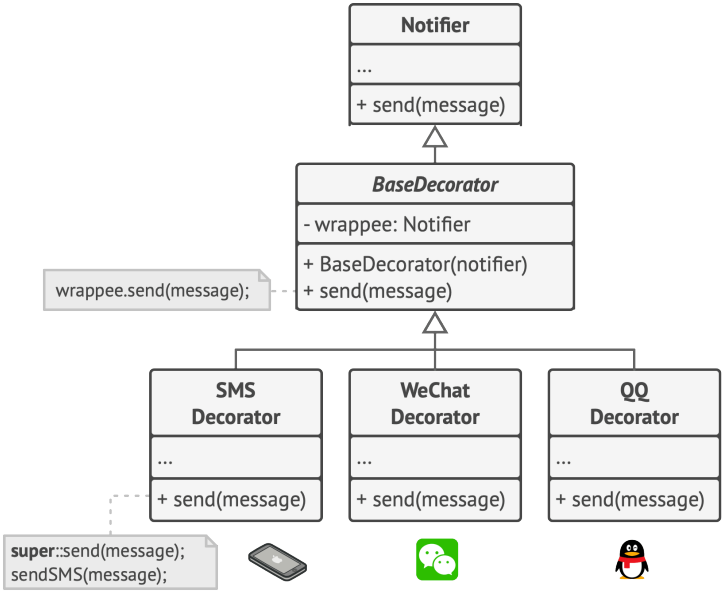
**封装器**是装饰模式的别称，这个称谓明确地表达了该模式的主要思想。“封装器”是一个能与其他“目标”对象连接的对象。封装器包含与目标对象相同的一系列方法，它会将所

1. **聚合**：对象 A 包含对象 B；B 可以独立于 A 存在。
**组合**：对象 A 由对象 B 构成；A 负责管理 B 的生命周期。B 无法独立于 A 存在。

有接收到的请求委派给目标对象。但是，封装器可以在将请求委派给目标前后对其进行处理，所以可能会改变最终结果。

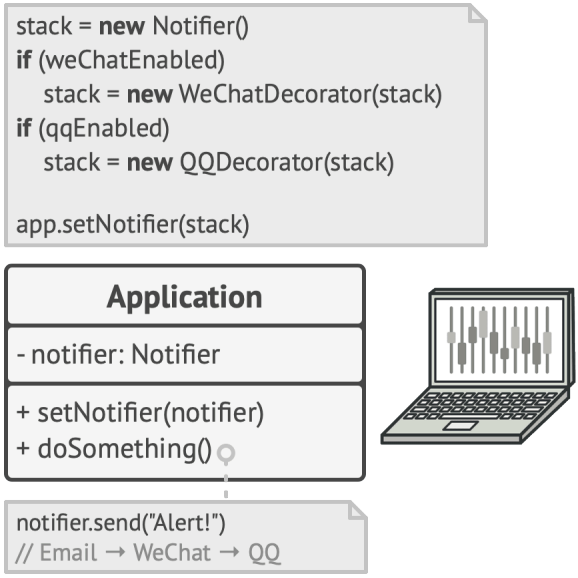
那么什么时候一个简单的封装器可以被称为是真正的装饰呢？正如之前提到的，封装器实现了与其封装对象相同的接口。因此从客户端的角度来看，这些对象是完全一样的。封装器中的引用成员变量可以是遵循相同接口的任意对象。这使得你可以将一个对象放入多个封装器中，并在对象中添加所有这些封装器的组合行为。

比如在消息通知示例中，我们可以将简单邮件通知行为放在基类`通知器`中，但将所有其他通知方法放入装饰中。




将各种通知方法放入装饰。

客户端代码必须将基础通知器放入一系列自己所需的装饰中。因此最后的对象将形成一个栈结构。

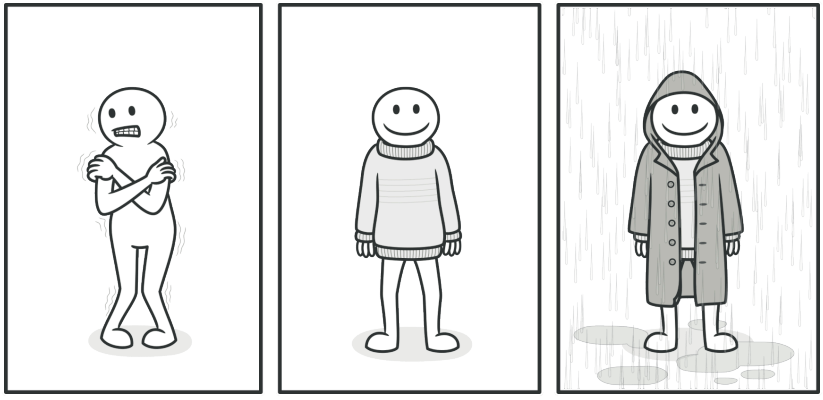



程序可以配置由通知装饰构成的复杂栈。

实际与客户端进行交互的对象将是最后一个进入栈中的装饰对象。由于所有的装饰都实现了与通知基类相同的接口，客户端的其他代码并不在意自己到底是与“纯粹”的通知器对象，还是与装饰后的通知器对象进行交互。

我们可以使用相同方法来完成其他行为（例如设置消息格式或者创建接收人列表）。只要所有装饰都遵循相同的接口，客户端就可以使用任意自定义的装饰来装饰对象。

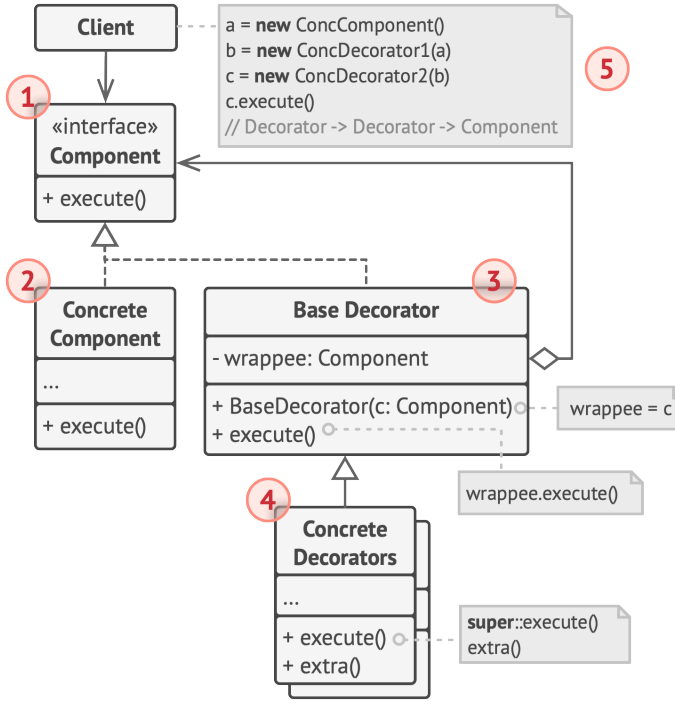
# 真实世界类比



穿上多件衣服将获得组合性的效果。

穿衣服是使用装饰的一个例子。觉得冷时，你可以穿一件毛衣。如果穿毛衣还觉得冷，你可以再套上一件夹克。如果遇到下雨，你还可以再穿一件雨衣。所有这些衣物都“扩展”了你的基本行为，但它们并不是你的一部分，如果你不再需要某件衣物，可以方便地随时脱掉。

# 结构




1. **部件**（Component）声明封装器和被封装对象的公用接口。

2. **具体部件**（Concrete Component）类是被封装对象所属的类。它定义了基础行为，但装饰类可以改变这些行为。

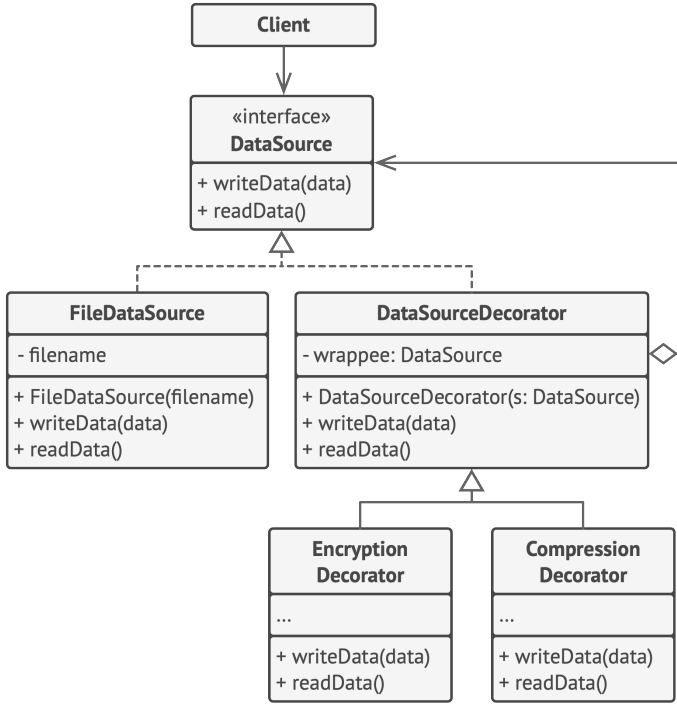
3. **基础装饰**（Base Decorator）类拥有一个指向被封装对象的引用成员变量。该变量的类型应当被声明为通用部件接口，这样它就可以引用具体的部件和装饰。装饰基类会将所有操作委派给被封装的对象。

4. **具体装饰类**（Concrete Decorators）定义了可动态添加到部件的额外行为。具体装饰类会重写装饰基类的方法，并在调用父类方法之前或之后进行额外的行为。

5. **客户端**（Client）可以使用多层装饰来封装部件，只要它能使用通用接口与所有对象互动即可。

# # 伪代码

在本例中，**装饰*模式能够对敏感数据进行压缩和加密，从而将数据从使用数据的代码中独立出来。




加密和压缩装饰的示例。

程序使用一对装饰来封装数据源对象。这两个封装器都改变了从磁盘读写数据的方式：

- 当数据即将被**写入磁盘**前，装饰对数据进行加密和压缩。在原始类对改变毫无察觉的情况下，将加密后的受保护数据写入文件。

- 当数据刚**从磁盘读出**后，同样通过装饰对数据进行解压和解密。装饰和数据源类实现同一接口，从而能在客户端代码中相互替换。

```
// 装饰可以改变组件接口所定义的操作。
interface DataSource is
  method writeData(data)
  method readData():data

// 具体组件提供操作的默认实现。这些类在程序中可能会有几个变体。
class FileDataSource implements DataSource is
  constructor FileDataSource(filename) { ... }

  method writeData(data) is
    // 将数据写入文件。

  method readData():data is
    // 从文件读取数据。

// 装饰基类和其他组件遵循相同的接口。该类的主要任务是定义所有具体装饰的封
// 装接口。封装的默认实现代码中可能会包含一个保存被封装组件的成员变量，并
// 且负责对其进行初始化。
```

```
class DataSourceDecorator implements DataSource is
  protected field wrappee: DataSource

  constructor DataSourceDecorator(source: DataSource) is
    wrappee = source

  // 装饰基类会直接将所有工作分派给被封装组件。具体装饰中则可以新增一些
  // 额外的行为。
  method writeData(data) is
    wrappee.writeData(data)

  // 具体装饰可调用其父类的操作实现，而不是直接调用被封装对象。这种方式
  // 可简化装饰类的扩展工作。
  method readData():data is
    return wrappee.readData()

// 具体装饰必须在被封装对象上调用方法，不过也可以自行在结果中添加一些内容。
// 装饰必须在调用封装对象之前或之后执行额外的行为。
class EncryptionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. 对传递数据进行加密。
    // 2. 将加密后数据传递给被封装对象 writeData（写入数据）方法。

  method readData():data is
    // 1. 通过被封装对象的 readData（读取数据）方法获取数据。
    // 2. 如果数据被加密就尝试解密。
    // 3. 返回结果。

// 你可以将对象封装在多层装饰中。
class CompressionDecorator extends DataSourceDecorator is
  method writeData(data) is
    // 1. 压缩传递数据。
```

```
51      // 2. 将压缩后数据传递给被封装对象 writeData（写入数据）方法。
52
53    method readData():data is
54      // 1. 通过被封装对象的 readData（读取数据）方法获取数据。
55      // 2. 如果数据被压缩就尝试解压。
56      // 3. 返回结果。
57
58
59  // 选项 1: 装饰组件的简单示例
60  class Application is
61    method dumbUsageExample() is
62      source = new FileDataSource("somefile.dat")
63      source.writeData(salaryRecords)
64      // 已将明码数据写入目标文件。
65
66      source = new CompressionDecorator(source)
67      source.writeData(salaryRecords)
68      // 已将压缩数据写入目标文件。
69
70      source = new EncryptionDecorator(source)
71      // 源变量中现在包含:
72      // Encryption > Compression > FileDataSource
73      source.writeData(salaryRecords)
74      // 已将压缩且加密的数据写入目标文件。
75
76
77  // 选项 2: 客户端使用外部数据源。SalaryManager（工资管理器）对象并不关心
78  // 数据如何存储。它们会与提前配置好的数据源进行交互，数据源则是通过程序配
79  // 置器获取的。
80  class SalaryManager is
81    field source: DataSource
82
```

```
  constructor SalaryManager(source: DataSource) { ... }

  method load() is
    return source.readData()

  method save() is
    source.writeData(salaryRecords)
  // ...其他有用的方法...


// 程序可在运行时根据配置或环境组装不同的装饰堆栈。
class ApplicationConfigurator is
  method configurationExample() is
    source = new FileDataSource("salary.dat")
    if (enabledEncryption)
      source = new EncryptionDecorator(source)
    if (enabledCompression)
      source = new CompressionDecorator(source)

    logger = new SalaryManager(source)
    salary = logger.load()
  // ...
```

## 适合应用场景

**如果你希望在无需修改代码的情况下即可使用对象，且希望在运行时为对象新增额外的行为，可以使用装饰模式。**

装饰能将业务逻辑组织为层次结构，你可为各层创建一个装饰，在运行时将各种不同逻辑组合成对象。由于这些对象都遵循通用接口，客户端代码能以相同的方式使用这些对象。

**如果用继承来扩展对象行为的方案难以实现或者根本不可行，你可以使用该模式。**

许多编程语言使用 final 最终 关键字来限制对某个类的进一步扩展。复用最终类已有行为的唯一方法是使用装饰模式：用封装器对其进行封装。

# 实现方式

1. 确保业务逻辑可用一个基本组件及多个额外可选层次表示。
2. 找出基本组件和可选层次的通用方法。创建一个组件接口并在其中声明这些方法。
3. 创建一个具体组件类，并定义其基础行为。
4. 创建装饰基类，使用一个成员变量存储指向被封装对象的引用。该成员变量必须被声明为组件接口类型，从而能在运行时连接具体组件和装饰。装饰基类必须将所有工作委派给被封装的对象。
5. 确保所有类实现组件接口。

6. 将装饰基类扩展为具体装饰。具体装饰必须在调用父类方法（总是委派给被封装对象）之前或之后执行自身的行为。

7. 客户端代码负责创建装饰并将其组合成客户端所需的形式。

## ⚖ 优缺点

✓ 你无需创建新子类即可扩展对象的行为。

✓ 你可以在运行时添加或删除对象的功能。

✓ 你可以用多个装饰封装对象来组合几种行为。

✓ 单一职责原则。你可以将实现了许多不同行为的一个大类拆分为多个较小的类。

✕ 在封装器栈中删除特定封装器比较困难。

✕ 实现行为不受装饰栈顺序影响的装饰比较困难。

✕ 各层的初始化配置代码看上去可能会很糟糕。

## ⇄ 与其他模式的关系

- **适配器**可以对已有对象的接口进行修改，**装饰**则能在不改变对象接口的前提下强化对象功能。此外，装饰还支持递归组合，适配器则无法实现。

- **适配器**能为被封装对象提供不同的接口，**代理**能为对象提供相同的接口，**装饰**则能为对象提供加强的接口。

- **责任链**和**装饰**模式的类结构非常相似。两者都依赖递归组合将需要执行的操作传递给一系列对象。但是，两者有几点重要的不同之处。

  **责任链**的管理者可以相互独立地执行一切操作，还可以随时停止传递请求。另一方面，各种**装饰**可以在遵循基本接口的情况下扩展对象的行为。此外，装饰无法中断请求的传递。

- **组合**和**装饰**的结构图很相似，因为两者都依赖递归组合来组织无限数量的对象。

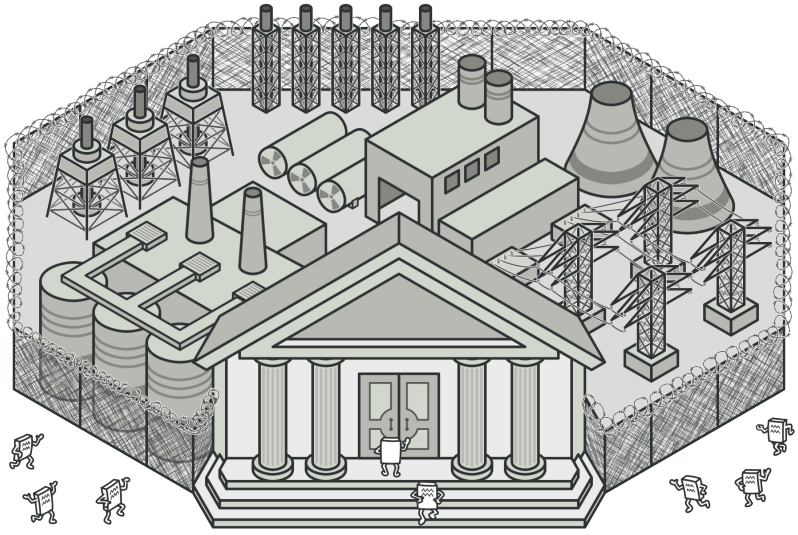
  **装饰**类似于**组合**，但其只有一个子组件。此外还有一个明显不同：**装饰**为被封装对象添加了额外的职责，**组合**仅对其子节点的结果进行了“求和”。

  但是，模式也可以相互合作：你可以使用**装饰**来扩展**组合**树中特定对象的行为。

- 大量使用**组合**和**装饰**的设计通常可从对于**原型**的使用中获益。你可以通过该模式来复制复杂结构，而非从零开始重新构造。

- **装饰**可让你更改对象的外表，**策略**则让你能够改变其本质。

- **装饰**和**代理**有着相似的结构，但是其意图却非常不同。这两个模式的构建都基于组合原则，也就是说一个对象应该将部分工作委派给另一个对象。两者之间的不同之处在于**代理**通常自行管理其服务对象的生命周期，而**装饰**的生成则总是由客户端进行控制。



# 外观

亦称：门面模式、Facade

**外观**是一种结构型设计模式，能为程序库、框架或其他复杂类提供一个简单的接口。

# ☹ 问题

假设你必须在代码中使用某个复杂的库或框架中的众多对象。正常情况下，你需要负责所有对象的初始化工作、管理其依赖关系并按正确的顺序执行方法等。

最终，程序中类的业务逻辑将与第三方类的实现细节紧密耦合，使得理解和维护代码的工作很难进行。

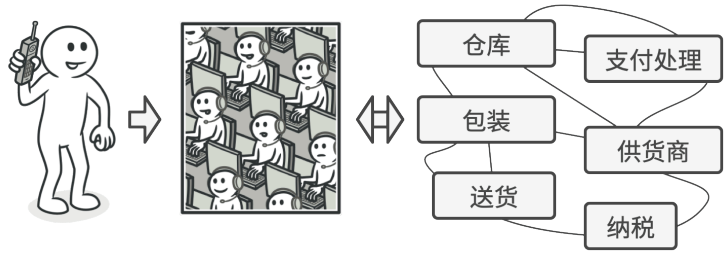
# ☺ 解决方案

外观类为包含许多活动部件的复杂子系统提供一个简单的接口。与直接调用子系统相比，外观提供的功能可能比较有限，但它却包含了客户端真正关心的功能。

如果你的程序需要与包含几十种功能的复杂库整合，但只需使用其中非常少的功能，那么使用外观模式会非常方便，

例如，上传猫咪搞笑短视频到社交媒体网站的应用可能会用到专业的视频转换库，但它只需使用一个包含 `encode(filename, format)` 方法（以文件名与文件格式为参数进行编码的方法）的类即可。在创建这个类并将其连接到视频转换库后，你就拥有了自己的第一个外观。

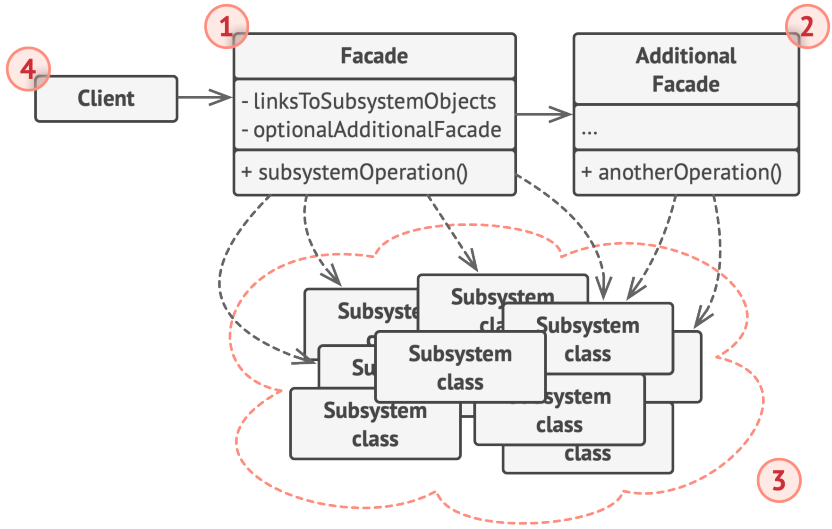
# 真实世界类比




电话购物。

当你通过电话给商店下达订单时，接线员就是该商店的所有服务和部门的外观。接线员为你提供了一个同购物系统、支付网关和各种送货服务进行互动的简单语音接口。

# 结构



1. **外观**（Facade）提供了一种访问特定子系统功能的便捷方式，其了解如何重定向客户端请求，知晓如何操作一切活动部件。

2. 创建**附加外观**（Additional Facade）类可以避免多种不相关的功能污染单一外观，使其变成又一个复杂结构。客户端和其他外观都可使用附加外观。

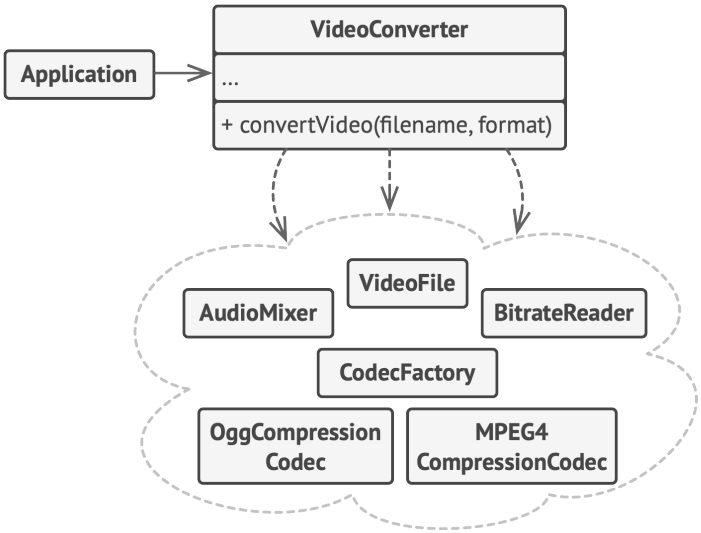
3. **复杂子系统**（Complex Subsystem）由数十个不同对象构成。如果要用这些对象完成有意义的工作，你必须深入了解子系统的实现细节，比如按照正确顺序初始化对象和为其提供正确格式的数据。

   子系统类不会意识到外观的存在，它们在系统内运作并且相互之间可直接进行交互。

4. **客户端**（Client）使用外观代替对子系统对象的直接调用。

# # 伪代码

在本例中，**外观**模式简化了客户端与复杂视频转换框架之间的交互。




使用单个外观类隔离多重依赖的示例

你可以创建一个封装所需功能并隐藏其他代码的外观类，从而无需使全部代码直接与数十个框架类进行交互。该结构还能将未来框架升级或更换所造成的影响最小化，因为你只需修改程序中外观方法的实现即可。

```
// 这里有复杂第三方视频转换框架中的一些类。我们不知晓其中的代码，因此无法
// 对其进行简化。

class VideoFile
// ...

class OggCompressionCodec
// ...

```

```
class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// 为了将框架的复杂性隐藏在一个简单接口背后，我们创建了一个外观类。它是在
// 功能性和简洁性之间做出的权衡。
class VideoConverter is
  method convert(filename, format):File is
    file = new VideoFile(filename)
    sourceCodec = new CodecFactory.extract(file)
    if (format == "mp4")
      destinationCodec = new MPEG4CompressionCodec()
    else
      destinationCodec = new OggCompressionCodec()
    buffer = BitrateReader.read(filename, sourceCodec)
    result = BitrateReader.convert(buffer, destinationCodec)
    result = (new AudioMixer()).fix(result)
    return new File(result)

// 应用程序的类并不依赖于复杂框架中成千上万的类。同样，如果你决定更换框架，
// 那只需重写外观类即可。
class Application is
  method main() is
```

```
42      convertor = new VideoConverter()
43      mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
44      mp4.save()
```

## 适合应用场景

**如果你需要一个指向复杂子系统的直接接口，且该接口的功能有限，则可以使用外观模式。**

子系统通常会随着时间的推进变得越来越复杂。即便是应用了设计模式，通常你也会创建更多的类。尽管在多种情形中子系统可能是更灵活或易于复用的，但其所需的配置和样板代码数量将会增长得更快。为了解决这个问题，外观将会提供指向子系统中最常用功能的快捷方式，能够满足客户端的大部分需求。

**如果需要将子系统组织为多层结构，可以使用外观。**

创建外观来定义子系统中各层次的入口。你可以要求子系统仅使用外观来进行交互，以减少子系统之间的耦合。

让我们回到视频转换框架的例子。该框架可以拆分为两个层次：音频相关和视频相关。你可以为每个层次创建一个外观，然后要求各层的类必须通过这些外观进行交互。这种方式看上去与**中介者**模式非常相似。

## 实现方式

1. 考虑能否在现有子系统的基础上提供一个更简单的接口。如果该接口能让客户端代码独立于众多子系统类，那么你的方向就是正确的。

2. 在一个新的外观类中声明并实现该接口。外观应将客户端代码的调用重定向到子系统中的相应对象处。如果客户端代码没有对子系统进行初始化，也没有对其后续生命周期进行管理，那么外观必须完成此类工作。

3. 如果要充分发挥这一模式的优势，你必须确保所有客户端代码仅通过外观来与子系统进行交互。此后客户端代码将不会受到任何由子系统代码修改而造成的影响，比如子系统升级后，你只需修改外观中的代码即可。

4. 如果外观变得**过于臃肿**，你可以考虑将其部分行为抽取为一个新的专用外观类。

## 优缺点

✓ 你可以让自己的代码独立于复杂子系统。

✕ 外观可能成为与程序中所有类都耦合的**上帝对象**。

# ⇄ 与其他模式的关系

- **外观**为现有对象定义了一个新接口，**适配器**则会试图运用已有的接口。适配器通常只封装一个对象，外观通常会作用于整个对象子系统上。

- 当只需对客户端代码隐藏子系统创建对象的方式时，你可以使用**抽象工厂**来代替**外观**。

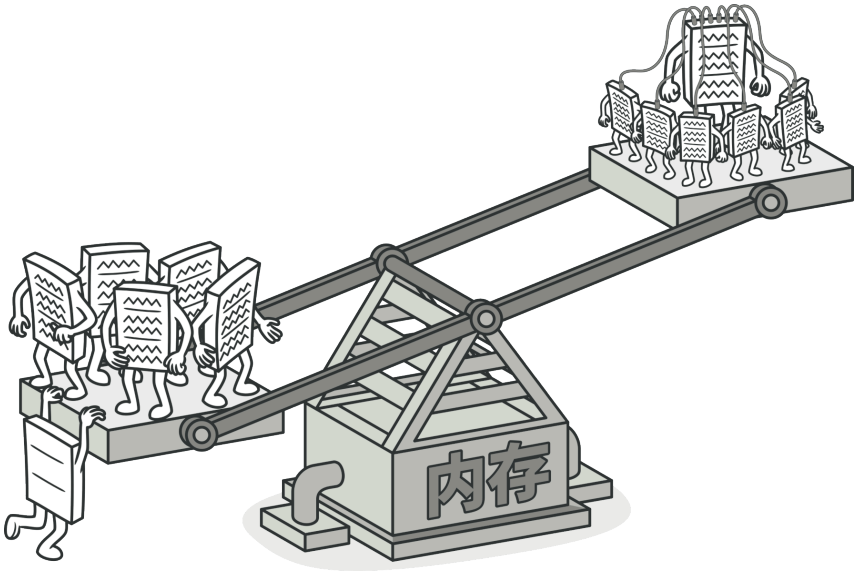
- **享元**展示了如何生成大量的小型对象，**外观**则展示了如何用一个对象来代表整个子系统。

- **外观**和**中介者**的职责类似：它们都尝试在大量紧密耦合的类中组织起合作。

  - 外观为子系统中的所有对象定义了一个简单接口，但是它不提供任何新功能。子系统本身不会意识到外观的存在。子系统中的对象可以直接进行交流。
  - 中介者将系统中组件的沟通行为中心化。各组件只知道中介者对象，无法直接相互交流。

- **外观**类通常可以转换为**单例**类，因为在大部分情况下一个外观对象就足够了。

- **外观**与**代理**的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。代理与其服务对象遵循同一接口，使得自己和服务对象可以互换，在这一点上它与外观不同。




# 享元

亦称：缓存、Cache、Flyweight

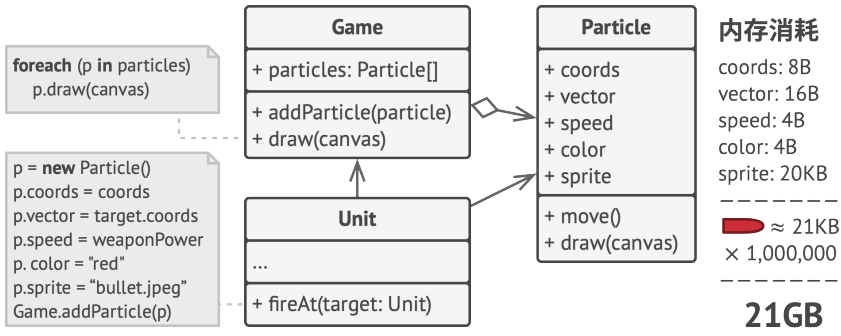
**享元**是一种结构型设计模式，
它摒弃了在每个对象中保存
所有数据的方式，通过共享
多个对象所共有的相同状态，
让你能在有限的内存容量中
载入更多对象。

# ☹ 问题

假如你希望在长时间工作后放松一下，所以开发了一款简单的游戏：玩家们在地图上移动并相互射击。你决定实现一个真实的粒子系统，并将其作为游戏的特色。大量的子弹、导弹和爆炸弹片会在整个地图上穿行，为玩家提供紧张刺激的游戏体验。

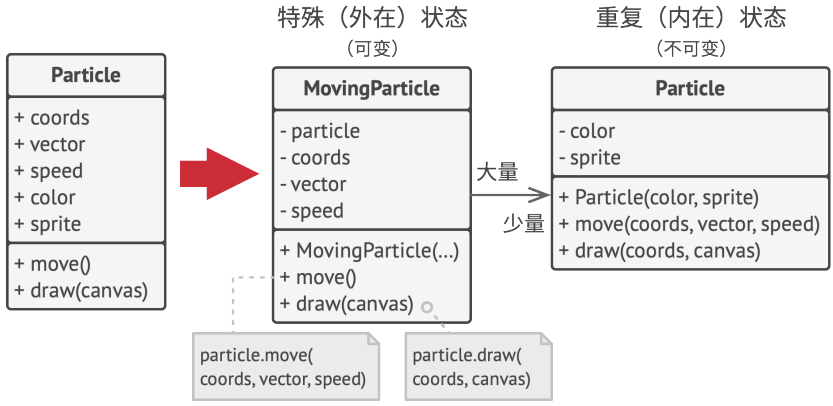
开发完成后，你推送提交了最新版本的程序，并在编译游戏后将其发送给了一个朋友进行测试。尽管该游戏在你的电脑上完美运行，但是你的朋友却无法长时间进行游戏：游戏总是会在他的电脑上运行几分钟后崩溃。在研究了几个小时的调试消息记录后，你发现导致游戏崩溃的原因是内存容量不足。朋友的设备性能远比不上你的电脑，因此游戏运行在他的电脑上时很快就会出现问题。

真正的问题与粒子系统有关。每个粒子（一颗子弹、一枚导弹或一块弹片）都由包含完整数据的独立对象来表示。当玩家在游戏中鏖战进入高潮后的某一时刻，游戏将无法在剩余内存中载入新建粒子，于是程序就崩溃了。



# 😊解决方案

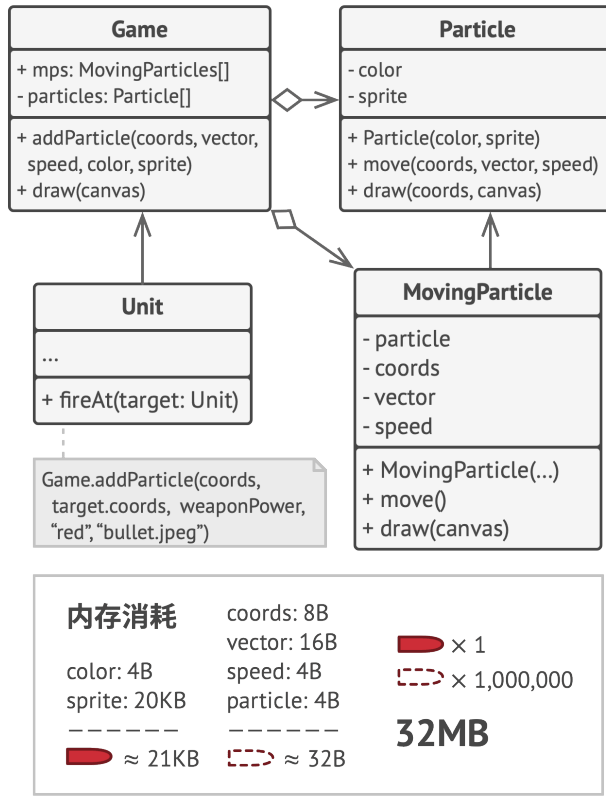
仔细观察粒子 Particle 类，你可能会注意到颜色（color）和精灵图（sprite）这两个成员变量所消耗的内存要比其他变量多得多。更糟糕的是，对于所有的粒子来说，这两个成员变量所存储的数据几乎完全一样（比如所有子弹的颜色和精灵图都一样）。




每个粒子的另一些状态（坐标、移动矢量和速度）则是不同的。因为这些成员变量的数值会不断变化。这些数据代表粒子在存续期间不断变化的情景，但每个粒子的颜色和精灵图则会保持不变。

对象的常量数据通常被称为内在状态，其位于对象中，其他对象只能读取但不能修改其数值。而对象的其他状态常常能被其他对象“从外部”改变，因此被称为外在状态。

享元模式建议不在对象中存储外在状态，而是将其传递给依赖于它的一个特殊方法。程序只在对象中保存内在状态，以方便在不同情景下重用。这些对象的区别仅在于其内在状态（与外在状态相比，内在状态的变体要少很多），因此你所需的对象数量会大大削减。

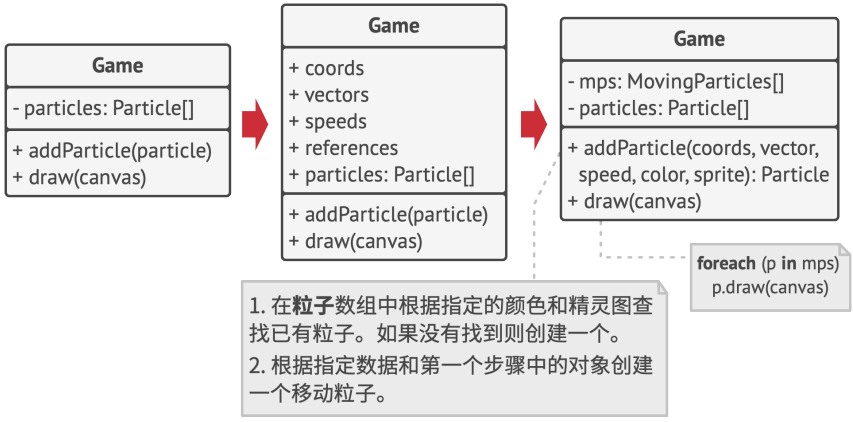



让我们回到游戏中。假如能从粒子类中抽出外在状态，那么我们只需三个不同的对象（子弹、导弹和弹片）就能表示游戏中的所有粒子。你现在很可能已经猜到了，我们将这样一个仅存储内在状态的对象称为享元。

## 外在状态存储

那么外在状态会被移动到什么地方呢？总得有类来存储它们，对不对？在大部分情况中，它们会被移动到容器对象中，也就是我们应用享元模式前的聚合对象中。

在我们的例子中，容器对象就是主要的游戏 `Game` 对象，其会将所有粒子存储在名为 粒子 `particles` 的成员变量中。为了能将外在状态移动到这个类中，你需要创建多个数组成员变量来存储每个粒子的坐标、方向矢量和速度。除此之外，你还需要另一个数组来存储指向代表粒子的特定享元的引用。这些数组必须保持同步，这样你才能够使用同一索引来获取关于某个粒子的所有数据。




更优雅的解决方案是创建独立的情景类来存储外在状态和对享元对象的引用。在该方法中，容器类只需包含一个数组。

稍等！这样的话情景对象数量不是会和不采用该模式时的对象数量一样多吗？的确如此，但这些对象要比之前小很多。消耗内存最多的成员变量已经被移动到很少的几个享元对象中了。现在，一个享元大对象会被上千个情境小对象复用，因此无需再重复存储数千个大对象的数据。

## 享元与不可变性

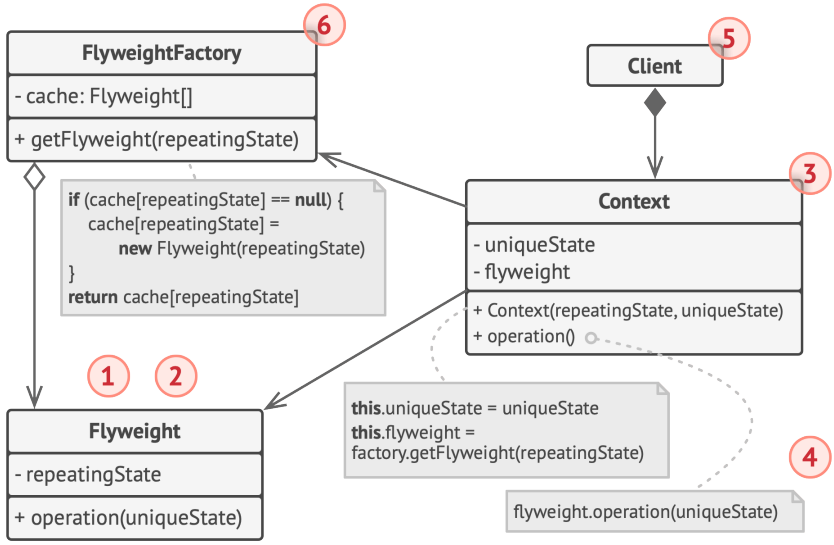
由于享元对象可在不同的情景中使用，你必须确保其状态不能被修改。享元类的状态只能由构造函数的参数进行一次性初始化，它不能对其他对象公开其设置器或公有成员变量。

## 享元工厂

为了能更方便地访问各种享元，你可以创建一个工厂方法来管理已有享元对象的缓存池。工厂方法从客户端处接收目标享元对象的内在状态作为参数，如果它能在缓存池中找到所需享元，则将其返回给客户端；如果没有找到，它就会新建一个享元，并将其添加到缓存池中。

你可以选择在程序的不同地方放入该函数。最简单的选择就是将其放置在享元容器中。除此之外，你还可以新建一个工厂类，或者创建一个静态的工厂方法并将其放入实际的享元类中。

# 结构




1. 享元模式只是一种优化。在应用该模式之前，你要确定程序中存在与大量类似对象同时占用内存相关的内存消耗问题，并且确保该问题无法使用其他更好的方式来解决。

2. **享元**（Flyweight）类包含原始对象中部分能在多个对象中共享的状态。同一享元对象可在许多不同情景中使用。享元中存储的状态被称为“内在状态”。传递给享元方法的状态被称为“外在状态”。

3. **情景**（Context）类包含原始对象中各不相同的外在状态。情景与享元对象组合在一起就能表示原始对象的全部状态。

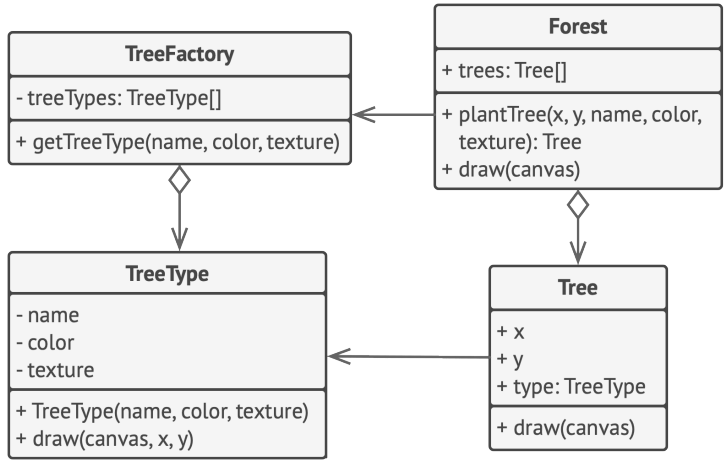
4. 通常情况下，原始对象的行为会保留在享元类中。因此调用享元方法必须提供部分外在状态作为参数。但你也可将行为移动到情景类中，然后将连入的享元作为单纯的数据对象。

5. **客户端**（Client）负责计算或存储享元的外在状态。在客户端看来，享元是一种可在运行时进行配置的模板对象，具体的配置方式为向其方法中传入一些情景数据参数。

6. **享元工厂**（Flyweight Factory）会对已有享元的缓存池进行管理。有了工厂后，客户端就无需直接创建享元，它们只需调用工厂并向其传递目标享元的一些内在状态即可。工厂会根据参数在之前已创建的享元中进行查找，如果找到满足条件的享元就将其返回；如果没有找到就根据参数新建享元。

# # 伪代码

在本例中，**享元**模式能有效减少在画布上渲染数百万个树状对象时所需的内存。



该模式从主要的 树 Tree 类中抽取内在状态，并将其移动到享元类 树种类 TreeType 之中。

最初程序需要在多个对象中存储相同数据，而现在仅需在几个享元对象中保存数据，然后在作为情景的 树 对象中连入享元即可。客户端代码使用享元工厂创建树对象并封装搜索指定对象的复杂行为，并能在需要时复用对象。

```
// 享元类包含一个树的部分状态。这些成员变量保存的数值对于特定树而言是唯一
// 的。例如，你在这里找不到树的坐标。但这里有很多树木之间所共有的纹理和颜
// 色。由于这些数据的体积通常非常大，所以如果让每棵树都其进行保存的话将耗
// 费大量内存。因此，我们可将纹理、颜色和其他重复数据导出到一个单独的对象
// 中，然后让众多的单个树对象去引用它。
class TreeType is
  field name
  field color
  field texture
  constructor TreeType(name, color, texture) { ... }
  method draw(canvas, x, y) is
    // 1. 创建特定类型、颜色和纹理的位图。
    // 2. 在画布坐标 (X,Y) 处绘制位图。

// 享元工厂决定是否复用已有享元或者创建一个新的对象。
class TreeFactory is
  static field treeTypes: collection of tree types
  static method getTreeType(name, color, texture) is
    type = treeTypes.find(name, color, texture)
    if (type == null)
      type = new TreeType(name, color, texture)
      treeTypes.add(type)
```

```
    return type

// 情景对象包含树状态的外在部分。程序中可以创建数十亿个此类对象，因为它们
// 体积很小：仅有两个整型坐标和一个引用成员变量。
class Tree is
  field x,y
  field type: TreeType
  constructor Tree(x, y, type) { ... }
  method draw(canvas) is
    type.draw(canvas, this.x, this.y)

// 树（Tree）和森林（Forest）类是享元的客户端。如果不打算继续对树类进行开
// 发，你可以将它们合并。
class Forest is
  field trees: collection of Trees

  method plantTree(x, y, name, color, texture) is
    type = TreeFactory.getTreeType(name, color, texture)
    tree = new Tree(x, y, type)
    trees.add(tree)

  method draw(canvas) is
    foreach (tree in trees) do
      tree.draw(canvas)
```

## 适合应用场景

仅在程序必须支持大量对象且没有足够的内存容量时使用享元模式。

应用该模式所获的收益大小取决于使用它的方式和情景。它在下列情况中最有效：

- 程序需要生成数量巨大的相似对象
- 这将耗尽目标设备的所有内存
- 对象中包含可抽取且能在多个对象间共享的重复状态。

# 实现方式

1. 将需要改写为享元的类成员变量拆分为两个部分：
    - 内在状态：包含不变的、可在许多对象中重复使用的数据的成员变量。
    - 外在状态：包含每个对象各自不同的情景数据的成员变量
2. 保留类中表示内在状态的成员变量，并将其属性设置为不可修改。这些变量仅可在构造函数中获得初始数值。
3. 找到所有使用外在状态成员变量的方法，为在方法中所用的每个成员变量新建一个参数，并使用该参数代替成员变量。
4. 你可以有选择地创建工厂类来管理享元缓存池，它负责在新建享元时检查已有的享元。如果选择使用工厂，客户端就只能通过工厂来请求享元，它们需要将享元的内在状态作为参数传递给工厂。

5. 客户端必须存储和计算外在状态（情景）的数值，因为只有这样才能调用享元对象的方法。为了使用方便，外在状态和引用享元的成员变量可以移动到单独的情景类中。

## ⚖ 优缺点

✓ 如果程序中有很多相似对象，那么你将可以节省大量内存。

✕ 你可能需要牺牲执行速度来换取内存，因为他人每次调用享元方法时都需要重新计算部分情景数据。

✕ 代码会变得更加复杂。团队中的新成员总是会问：“为什么要像这样拆分一个实体的状态?”。

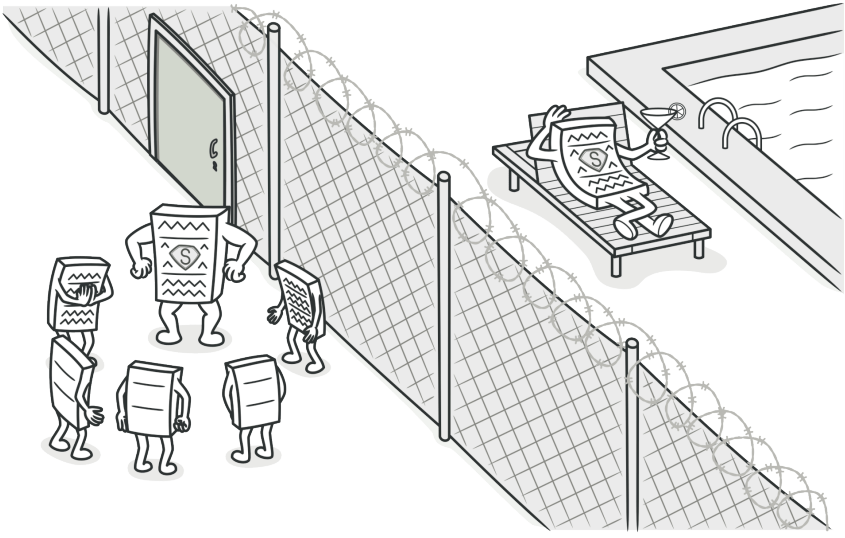
## ⇄ 与其他模式的关系

- 你可以使用**享元**实现**组合**树的共享叶节点以节省内存。

- **享元**展示了如何生成大量的小型对象，**外观**则展示了如何用一个对象来代表整个子系统。

- 如果你能将对象的所有共享状态简化为一个享元对象，那么**享元**就和**单例**类似了。但这两个模式有两个根本性的不同。

  1. 只会有一个单例实体，但是享元类可以有多个实体，各实体的内在状态也可以不同。
  2. 单例对象可以是可变的。享元对象是不可变的。



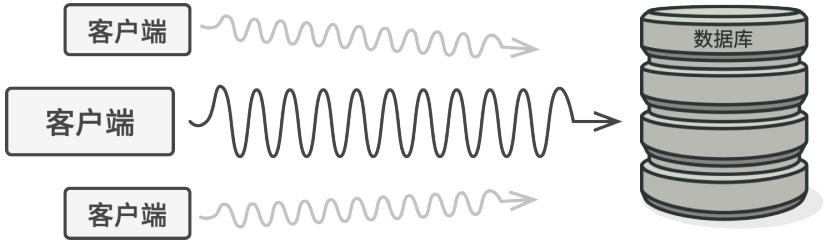
# 代理

亦称：Proxy

**代理**是一种结构型设计模式，让你能够提供对象的替代品或其占位符。代理控制着对于原对象的访问，并允许在将请求提交给对象前后进行一些处理。

# ☹ 问题

为什么要控制对于某个对象的访问呢？举个例子：有这样一个消耗大量系统资源的巨型对象，你只是偶尔需要使用它，并非总是需要。



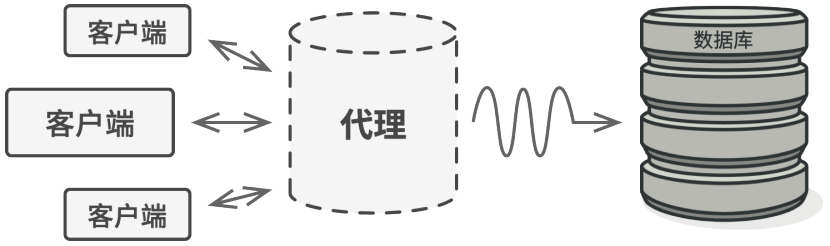

数据库查询有可能会非常缓慢。

你可以实现延迟初始化：在实际有需要时再创建该对象。对象的所有客户端都要执行延迟初始代码。不幸的是，这很可能会带来很多重复代码。

在理想情况下，我们希望将代码直接放入对象的类中，但这并非总是能实现：比如类可能是第三方封闭库的一部分。

# ☺ 解决方案

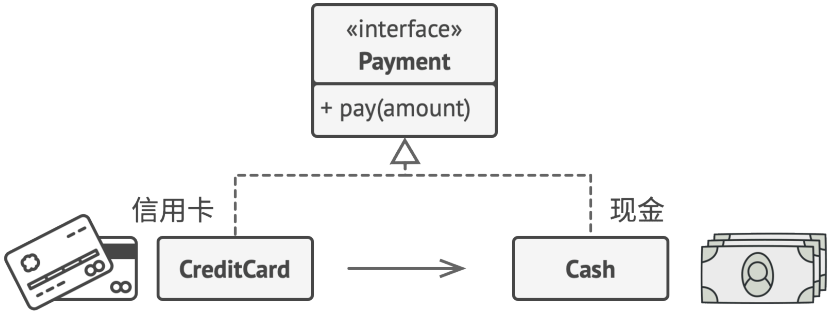
代理模式建议新建一个与原服务对象接口相同的代理类，然后更新应用以将代理对象传递给所有原始对象客户端。代理类接收到客户端请求后会创建实际的服务对象，并将所有工作委派给它。




代理将自己伪装成数据库对象，可在客户端或实际数据库对象不知情的情况下处理延迟初始化和缓存查询结果的工作。

这有什么好处呢？如果需要在类的主要业务逻辑前后执行一些工作，你无需修改类就能完成这项工作。由于代理实现的接口与原类相同，因此你可将其传递给任何一个使用实际服务对象的客户端。

## 真实世界类比

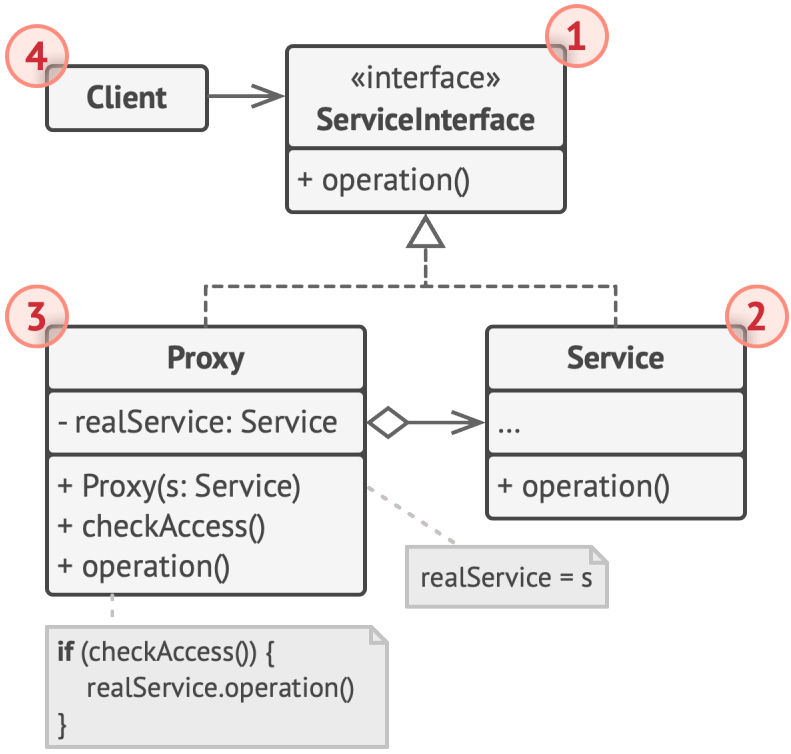



信用卡和现金在支付过程中的用处相同。

信用卡是银行账户的代理，银行账户则是一大捆现金的代理。它们都实现了同样的接口，均可用于进行支付。消费者会非

常满意，因为不必随身携带大量现金；商店老板同样会十分高兴，因为交易收入能以电子化的方式进入商店的银行账户中，无需担心存款时出现现金丢失或被抢劫的情况。

## 结构




1. **服务接口**（Service Interface）声明了服务接口。代理必须遵循该接口才能伪装成服务对象。

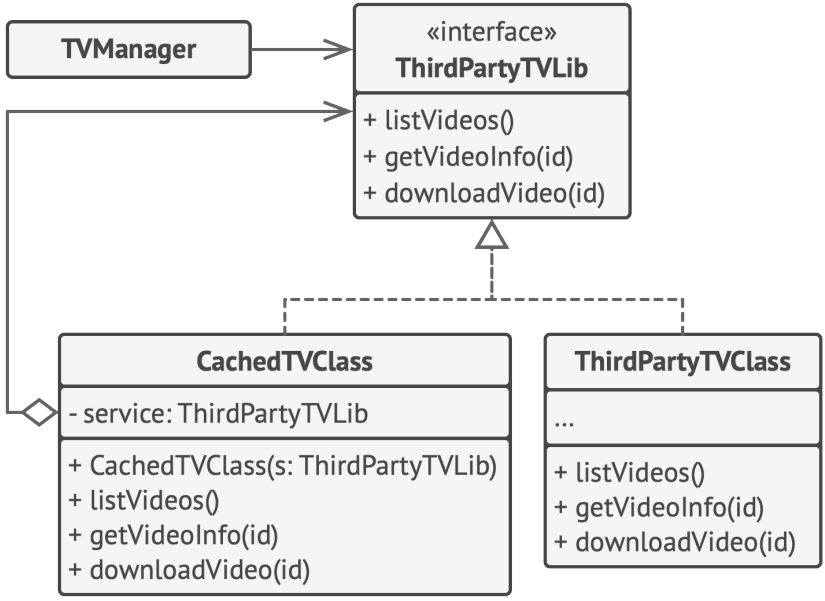
2. **服务**（Service）类提供了一些实用的业务逻辑。

3. **代理**（Proxy）类包含一个指向服务对象的引用成员变量。代理完成其任务（例如延迟初始化、记录日志、访问控制和缓存等）后会将请求传递给服务对象。通常情况下，代理会对其服务对象的整个生命周期进行管理。

4. **客户端**（Client）能通过同一接口与服务或代理进行交互，所以你可在一切需要服务对象的代码中使用代理。

# 伪代码

本例演示如何使用**代理**模式在第三方腾讯视频（TencentVideo，代码示例中记为 TV）程序库中添加延迟初始化和缓存。




使用代理缓冲服务结果。

程序库提供了视频下载类。但是该类的效率非常低。如果客户端程序多次请求同一视频，程序库会反复下载该视频，而不会将首次下载的文件缓存下来复用。

代理类实现和原下载器相同的接口，并将所有工作委派给原下载器。不过，代理类会保存所有的文件下载记录，如果程序多次请求同一文件，它会返回缓存的文件。

```
// 远程服务接口。
interface ThirdPartyTVLib is
  method listVideos()
  method getVideoInfo(id)
  method downloadVideo(id)

// 服务连接器的具体实现。该类的方法可以向腾讯视频请求信息。请求速度取决于
// 用户和腾讯视频的互联网连接情况。如果同时发送大量请求，即使所请求的信息
// 一模一样，程序的速度依然会减慢。
class ThirdPartyTVClass implements ThirdPartyTVLib is
  method listVideos() is
    // 向腾讯视频发送一个 API 请求。

  method getVideoInfo(id) is
    // 获取某个视频的元数据。

  method downloadVideo(id) is
    // 从腾讯视频下载一个视频文件。

// 为了节省网络带宽，我们可以将请求结果缓存下来并保存一段时间。但你可能无
// 法直接将这些代码放入服务类中。比如该类可能是第三方程序库的一部分或其签
// 名是`final（最终）`。因此我们会在一个实现了服务类接口的新代理类中放入
```

```
// 缓存代码。当代理类接收到真实请求后，才会将其委派给服务对象。
class CachedTVClass implements ThirdPartyTVLib is
  private field service: ThirdPartyTVLib
  private field listCache, videoCache
  field needReset

  constructor CachedTVClass(service: ThirdPartyTVLib) is
    this.service = service

  method listVideos() is
    if (listCache == null || needReset)
      listCache = service.listVideos()
    return listCache

  method getVideoInfo(id) is
    if (videoCache == null || needReset)
      videoCache = service.getVideoInfo(id)
    return videoCache

  method downloadVideo(id) is
    if (!downloadExists(id) || needReset)
      service.downloadVideo(id)

// 之前直接与服务对象交互的 GUI 类不需要改变，前提是它仅通过接口与服务对
// 象交互。我们可以安全地传递一个代理对象来代替真实服务对象，因为它们都实
// 现了相同的接口。
class TVManager is
  protected field service: ThirdPartyTVLib

  constructor TVManager(service: ThirdPartyTVLib) is
    this.service = service

```

```
    method renderVideoPage(id) is
      info = service.getVideoInfo(id)
      // 渲染视频页面。

    method renderListPanel() is
      list = service.listVideos()
      // 渲染视频缩略图列表。

    method reactOnUserInput() is
      renderVideoPage()
      renderListPanel()

// 程序可在运行时对代理进行配置。
class Application is
    method init() is
      aTVService = new ThirdPartyTVClass()
      aTVProxy = new CachedTVClass(aTVService)
      manager = new TVManager(aTVProxy)
      manager.reactOnUserInput()
```

## 适合应用场景

使用代理模式的方式多种多样，我们来看看最常见的几种。

**延迟初始化（虚拟代理）。如果你有一个偶尔使用的重量级服务对象，一直保持该对象运行会消耗系统资源时，可使用代理模式。**

你无需在程序启动时就创建该对象，可将对象的初始化延迟到真正有需要的时候。

**访问控制（保护代理）。如果你只希望特定客户端使用服务对象，这里的对象可以是操作系统中非常重要的部分，而客户端则是各种已启动的程序（包括恶意程序），此时可使用代理模式。**

代理可仅在客户端凭据满足要求时将请求传递给服务对象。

**本地执行远程服务（远程代理）。适用于服务对象位于远程服务器上的情形。**

在这种情形中，代理通过网络传递客户端请求，负责处理所有与网络相关的复杂细节。

**记录日志请求（日志记录代理）。适用于当你需要保存对于服务对象的请求历史记录时。代理可以在向服务传递请求前进行记录。**

缓存请求结果（缓存代理）。适用于需要缓存客户请求结果并对缓存生命周期进行管理时，特别是当返回结果的体积非常大时。

- 代理可对重复请求所需的相同结果进行缓存，还可使用请求参数作为索引缓存的键值。

**智能引用。可在没有客户端使用某个重量级对象时立即销毁该对象。**

代理会将所有获取了指向服务对象或其结果的客户端记录在案。代理会时不时地遍历各个客户端，检查它们是否仍在运行。如果相应的客户端列表为空，代理就会销毁该服务对象，释放底层系统资源。

代理还可以记录客户端是否修改了服务对象。其他客户端还可以复用未修改的对象。

# 实现方式

1. 如果没有现成的服务接口，你就需要创建一个接口来实现代理和服务对象的可交换性。从服务类中抽取接口并非总是可行的，因为你需要对服务的所有客户端进行修改，让它们使用接口。备选计划是将代理作为服务类的子类，这样代理就能继承服务的所有接口了。

2. 创建代理类，其中必须包含一个存储指向服务的引用的成员变量。通常情况下，代理负责创建服务并对其整个生命周期进行管理。在一些特殊情况下，客户端会通过构造函数将服务传递给代理。

3. 根据需求实现代理方法。在大部分情况下，代理在完成一些任务后应将工作委派给服务对象。

4. 可以考虑新建一个构建方法来判断客户端可获取的是代理还是实际服务。你可以在代理类中创建一个简单的静态方法，也可以创建一个完整的工厂方法。

5. 可以考虑为服务对象实现延迟初始化。

## ⚖ 优缺点

- ✓ 你可以在客户端毫无察觉的情况下控制服务对象。
- ✓ 如果客户端对服务对象的生命周期没有特殊要求，你可以对生命周期进行管理。
- ✓ 即使服务对象还未准备好或不存在，代理也可以正常工作。
- ✓ 开闭原则。你可以在不对服务或客户端做出修改的情况下创建新代理。
- ✕ 代码可能会变得复杂，因为需要新建许多类。
- ✕ 服务响应可能会延迟。

## ⇄ 与其他模式的关系

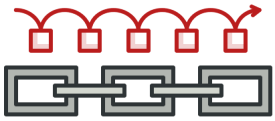
- **适配器**能为被封装对象提供不同的接口，**代理**能为对象提供相同的接口，**装饰**则能为对象提供加强的接口。

- **外观**与**代理**的相似之处在于它们都缓存了一个复杂实体并自行对其进行初始化。**代理**与其服务对象遵循同一接口，使得自己和服务对象可以互换，在这一点上它与**外观**不同。

- **装饰**和**代理**有着相似的结构，但是其意图却非常不同。这两个模式的构建都基于组合原则，也就是说一个对象应该将部分工作委派给另一个对象。两者之间的不同之处在于**代理**通常自行管理其服务对象的生命周期，而**装饰**的生成则总是由客户端进行控制。

# 行为模式

行为模式负责对象间的高效沟通和职责委派。



## 责任链

Chain of Responsibility

允许你将请求沿着处理者链进行发送。收到请求后，每个处理者均可对请求进行处理，或将其传递给链上的下个处理者。

## 命令

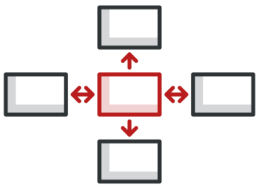
Command

它可将请求转换为一个包含与请求相关的所有信息的独立对象。该转换让你能根据不同的请求将方法参数化、延迟请求执行或将其放入队列中，且能实现可撤销操作。

## 迭代器

Iterator

让你能在不暴露集合底层表现形式（列表、栈和树等）的情况下遍历集合中所有的元素。



## 中介者

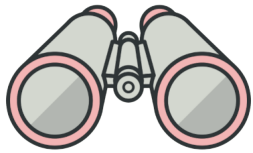
Mediator

能让你减少对象之间混乱无序的依赖关系。该模式会限制对象之间的直接交互，迫使它们通过一个中介者对象进行合作。

## 备忘录

Memento

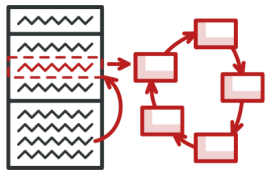
允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。



## 观察者

Observer

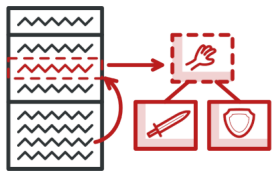
允许你定义一种订阅机制，可在对象事件发生时通知多个“观察”该对象的其他对象。



## 状态

State

让你能在一个对象的内部状态变化时改变其行为，使其看上去就像改变了自身所属的类一样。



## 策略

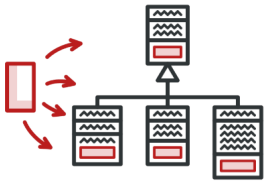
Strategy

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。

## 模板方法

Template Method

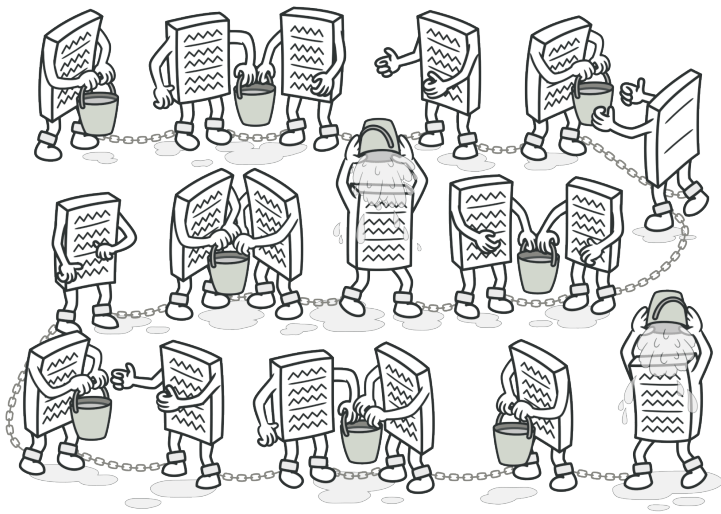
在超类中定义一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。



## 访问者

Visitor

将算法与其所作用的对象隔离开来。



# 责任链

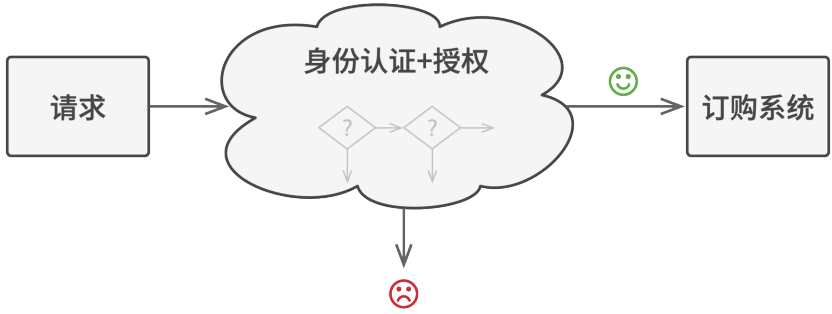
亦称：职责链模式、命令链、CoR、Chain of Command、Chain of Responsibility

**责任链**是一种行为设计模式，允许你将请求沿着处理者链进行发送。收到请求后，每个处理者均可对请求进行处理，或将其传递给链上的下个处理者。

# ☹ 问题

假如你正在开发一个在线订购系统。你希望对系统访问进行限制，只允许认证用户创建订单。此外，拥有管理权限的用户也拥有所有订单的完全访问权限。

简单规划后，你会意识到这些检查必须依次进行。只要接收到包含用户凭据的请求，应用程序就可尝试对进入系统的用户进行认证。但如果由于用户凭据不正确而导致认证失败，那就没有必要进行后续检查了。



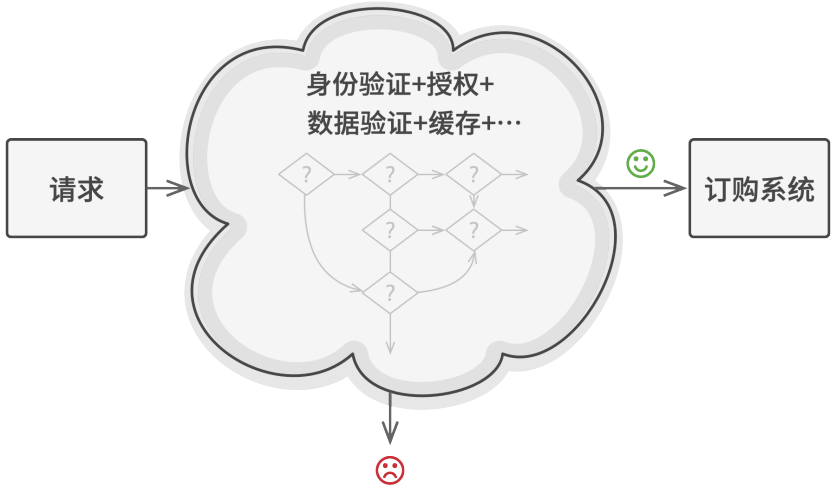

请求必须经过一系列检查后才能由订购系统来处理。

在接下来的几个月里，你实现了后续的几个检查步骤。

- 一位同事认为直接将原始数据传递给订购系统存在安全隐患。因此你新增了额外的验证步骤来清理请求中的数据。

- 过了一段时间，有人注意到系统无法抵御暴力密码破解方式的攻击。为了防范这种情况，你立刻添加了一个检查步骤来过滤来自同一 IP 地址的重复错误请求。

- 又有人提议你可以对包含同样数据的重复请求返回缓存中的结果，从而提高系统响应速度。因此，你新增了一个检查步骤，确保只有没有满足条件的缓存结果时请求才能通过并被发送给系统。




代码变得越来越多，也越来越混乱。

检查代码本来就已经混乱不堪，而每次新增功能都会使其更加臃肿。修改某个检查步骤有时会影响其他的检查步骤。最糟糕的是，当你希望复用这些检查步骤来保护其他系统组件时，你只能复制部分代码，因为这些组件只需部分而非全部的检查步骤。

系统会变得让人非常费解，而且其维护成本也会激增。你在艰难地和这些代码共处一段时间后，有一天终于决定对整个系统进行重构。

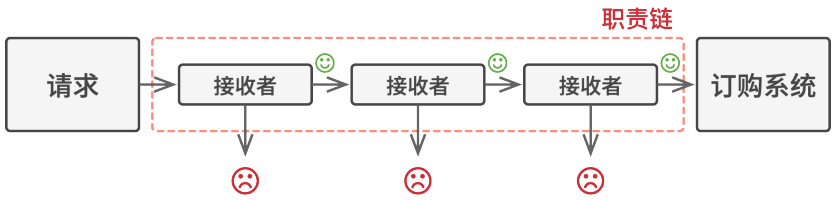
## 😊解决方案

与许多其他行为设计模式一样，**责任链**会将特定行为转换为被称作处理者的独立对象。在上述示例中，每个检查步骤都可被抽取为仅有单个方法的类，并执行检查操作。请求及其数据则会被作为参数传递给该方法。

模式建议你将这些处理者连成一条链。链上的每个处理者都有一个成员变量来保存对于下一处理者的引用。除了处理请求外，处理者还负责沿着链传递请求。请求会在链上移动，直至所有处理者都有机会对其进行处理。

最重要的是：处理者可以决定不再沿着链传递请求，这可高效地取消所有后续处理步骤。

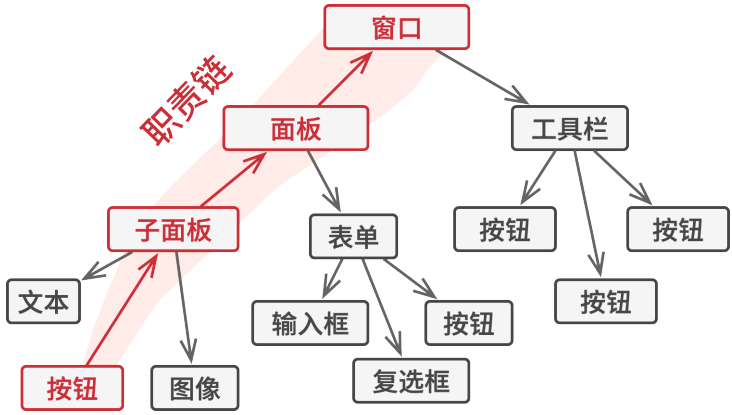
在我们的订购系统示例中，处理者会在进行请求处理工作后决定是否继续沿着链传递请求。如果请求中包含正确的数据，所有处理者都将执行自己的主要行为，无论该行为是身份验证还是数据缓存。




处理者依次排列，组成一条链。

不过还有一种稍微不同的方式（也是更经典一种），那就是处理者接收到请求后自行决定是否能够对其进行处理。如果自己能够处理，处理者就不再继续传递请求。因此在这种情况下，每个请求要么最多有一个处理者对其进行处理，要么没有任何处理者对其进行处理。在处理图形用户界面元素栈中的事件时，这种方式非常常见。

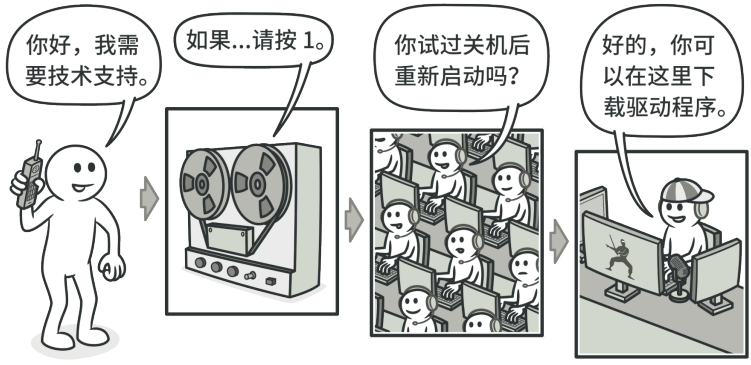
例如，当用户点击按钮时，按钮产生的事件将沿着 GUI 元素链进行传递，最开始是按钮的容器（如窗体或面板），直至应用程序主窗口。链上第一个能处理该事件的元素会对其进行处理。此外，该例还有另一个值得我们关注的地方：它表明我们总能从对象树中抽取出链来。




对象树的枝干可以组成一条链。

所有处理者类均实现同一接口是关键所在。每个具体处理者仅关心下一个包含 execute 执行 方法的处理者。这样一来，你就可以在运行时使用不同的处理者来创建链，而无需将相关代码与处理者的具体类进行耦合。

## 真实世界类比




给技术支持打电话时你可能得应对多名接听人员。

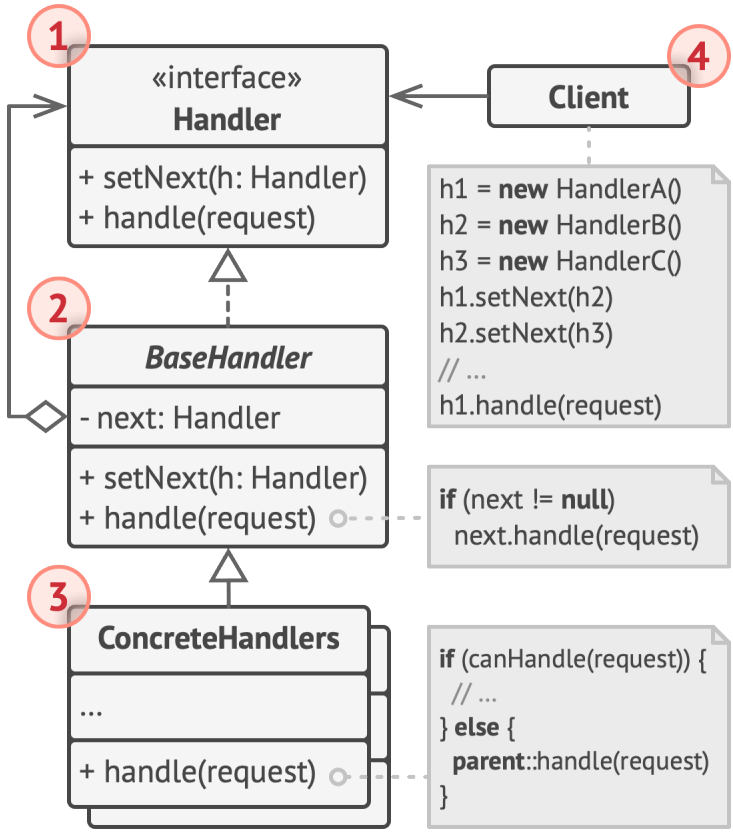
最近，你刚为自己的电脑购买并安装了一个新的硬件设备。身为一名极客，你显然在电脑上安装了多个操作系统，所以你会试着启动所有操作系统来确认其是否支持新的硬件设备。Windows 检测到了该硬件设备并对其进行了自动启用。但是你喜爱的 Linux 系统并不支持新硬件设备。抱着最后一点希望，你决定拨打包装盒上的技术支持电话。

首先你会听到自动回复器的机器合成语音，它提供了针对各种问题的九个常用解决方案，但其中没有一个与你遇到的问题相关。过了一会儿，机器人将你转接到人工接听人员处。

这位接听人员同样无法提供任何具体的解决方案。他不断地引用手册中冗长的内容，并不会仔细聆听你的回应。在第 10 次听到“你是否关闭计算机后重新启动呢?”这句话后，你要求与一位真正的工程师通话。

最后，接听人员将你的电话转接给了工程师，他或许正缩在某幢办公大楼的阴暗地下室中，坐在他所深爱的服务器机房里，焦躁不安地期待着同一名真人交流。工程师告诉了你新硬件设备驱动程序的下载网址，以及如何在 Linux 系统上进行安装。问题终于解决了！你挂断了电话，满心欢喜。

# 结构




1. **处理者**（Handler）声明了所有具体处理者的通用接口。该接口通常仅包含单个方法用于请求处理，但有时其还会包含一个设置链上下个处理者的方法。

2. **基础处理者**（Base Handler）是一个可选的类，你可以将所有处理者共用的样本代码放置在其中。

通常情况下，该类中定义了一个保存对于下个处理者引用的成员变量。客户端可通过将处理者传递给上个处理者的构造函数或设定方法来创建链。该类还可以实现默认的处理行为：确定下个处理者存在后再将请求传递给它。

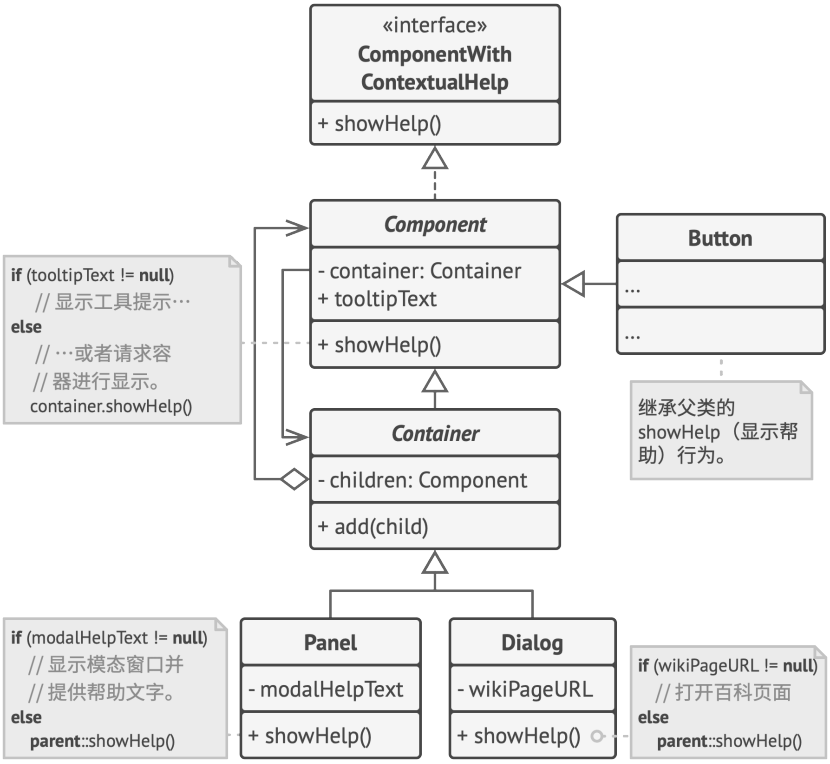
3. **具体处理者**（Concrete Handlers）包含处理请求的实际代码。每个处理者接收到请求后，都必须决定是否进行处理，以及是否沿着链传递请求。

   处理者通常是独立且不可变的，需要通过构造函数一次性地获得所有必要地数据。

4. **客户端**（Client）可根据程序逻辑一次性或者动态地生成链。值得注意的是，请求可发送给链上的任意一个处理者，而非必须是第一个处理者。

# # 伪代码

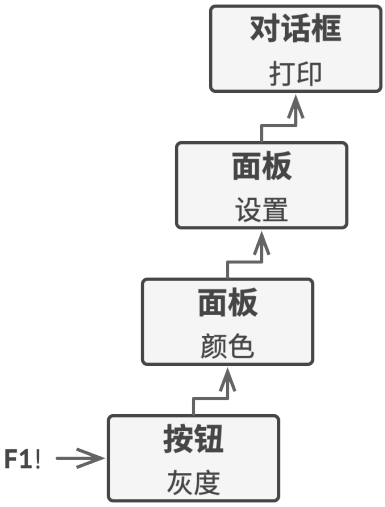
在本例中，**责任链**模式负责为活动的 GUI 元素显示上下文帮助信息。




GUI 类使用组合模式生成。每个元素都链接到自己的容器元素。你可随时构建从当前元素开始的、遍历其所有容器的元素链。

应用程序的 GUI 通常为对象树结构。例如，负责渲染程序主窗口的 对话框 类就是对象树的根节点。对话框包含 面板，而面板可能包含其他面板，或是 按钮 和 文本框 等下层元素。

只要给一个简单的组件指定帮助文本，它就可显示简短的上下文提示。但更复杂的组件可自定义上下文帮助文本的显示方式，例如显示手册摘录内容或在浏览器中打开一个网页。




帮助请求如何在 GUI 对象中移动。

当用户将鼠标指针移动到某个元素并按下 `F1` 键时，程序检测到指针下的组件并对其发送帮助请求。该请求不断向上传递到该元素所有的容器，直至某个元素能够显示帮助信息。

```
// 处理者接口声明了一个创建处理者链的方法。还声明了一个执行请求的方法。
interface ComponentWithContextualHelp is
  method showHelp()


// 简单组件的基础类。
```

```
abstract class Component implements ComponentWithContextualHelp is
  field tooltipText: string

  // 组件容器在处理者链中作为“下一个”链接。
  protected field container: Container

  // 如果组件设定了帮助文字，那它将会显示提示信息。如果组件没有帮助文字
  // 且其容器存在，那它会将调用传递给容器。
  method showHelp() is
    if (tooltipText != null)
      // 显示提示信息。
    else
      container.showHelp()


// 容器可以将简单组件和其他容器作为其子项目。链关系将在这里建立。该类将从
// 其父类处继承 showHelp（显示帮助）的行为。
abstract class Container extends Component is
  protected field children: array of Component

  method add(child) is
    children.add(child)
    child.container = this


// 原始组件应该能够使用帮助操作的默认实现...
class Button extends Component is
  // ...

// 但复杂组件可能会对默认实现进行重写。如果无法以新的方式来提供帮助文字，
// 那组件总是还能调用基础实现的（参见 Component 类）。
class Panel extends Container is
```

```
    field modalHelpText: string

    method showHelp() is
      if (modalHelpText != null)
        // 显示包含帮助文字的模态窗口。
      else
        super.showHelp()

// ...同上...
class Dialog extends Container is
    field wikiPageURL: string

    method showHelp() is
      if (wikiPageURL != null)
        // 打开百科帮助页面。
      else
        super.showHelp()


// 客户端代码。
class Application is
    // 每个程序都能以不同方式对链进行配置。
    method createUI() is
      dialog = new Dialog("预算报告")
      dialog.wikiPageURL = "http://..."
      panel = new Panel(0, 0, 400, 800)
      panel.modalHelpText = "本面板用于..."
      ok = new Button(250, 760, 50, 20, "确认")
      ok.tooltipText = "这是一个确认按钮..."
      cancel = new Button(320, 760, 50, 20, "取消")
      // ...
      panel.add(ok)
```

```
    panel.add(cancel)
    dialog.add(panel)

  // 想象这里会发生什么。
  method onF1KeyPress() is
    component = this.getComponentAtMouseCoords()
    component.showHelp()
```

## 适合应用场景

**当程序需要使用不同方式处理不同种类请求，而且请求类型和顺序预先未知时，可以使用责任链模式。**

该模式能将多个处理者连接成一条链。接收到请求后，它会“询问”每个处理者是否能够对其进行处理。这样所有处理者都有机会来处理请求。

**当必须按顺序执行多个处理者时，可以使用该模式。**

无论你以何种顺序将处理者连接成一条链，所有请求都会严格按照顺序通过链上的处理者。

**如果所需处理者及其顺序必须在运行时进行改变，可以使用责任链模式。**

如果在处理者类中有对引用成员变量的设定方法，你将能动态地插入和移除处理者，或者改变其顺序。

## 实现方式

1. 声明处理者接口并描述请求处理方法的签名。

   确定客户端如何将请求数据传递给方法。最灵活的方式是将请求转换为对象，然后将其以参数的形式传递给处理函数。

2. 为了在具体处理者中消除重复的样本代码，你可以根据处理者接口创建抽象处理者基类。

   该类需要有一个成员变量来存储指向链上下个处理者的引用。你可以将其设置为不可变类。但如果你打算在运行时对链进行改变，则需要定义一个设定方法来修改引用成员变量的值。

   为了使用方便，你还可以实现处理方法的默认行为。如果还有剩余对象，该方法会将请求传递给下个对象。具体处理者还能够通过调用父对象的方法来使用这一行为。

3. 依次创建具体处理者子类并实现其处理方法。每个处理者在接收到请求后都必须做出两个决定：

   - 是否自行处理这个请求。
   - 是否将该请求沿着链进行传递。

4. 客户端可以自行组装链，或者从其他对象处获得预先组装好的链。在后一种情况下，你必须实现工厂类以根据配置或环境设置来创建链。

5. 客户端可以触发链中的任意处理者，而不仅仅是第一个。请求将通过链进行传递，直至某个处理者拒绝继续传递，或者请求到达链尾。

6. 由于链的动态性，客户端需要准备好处理以下情况：

    - 链中可能只有单个链接。
    - 部分请求可能无法到达链尾。
    - 其他请求可能直到链尾都未被处理。

## ⚖ 优缺点

✓ 你可以控制请求处理的顺序。

✓ 单一职责原则。你可对发起操作和执行操作的类进行解耦。

✓ 开闭原则。你可以在不更改现有代码的情况下在程序中新增处理者。

✕ 部分请求可能未被处理。

# ⇄ 与其他模式的关系

- **责任链**、**命令**、**中介者**和**观察者**用于处理请求发送者和接收者之间的不同连接方式：

  - 责任链按照顺序将请求动态传递给一系列的潜在接收者，直至其中一名接收者对请求进行处理。
  - 命令在发送者和请求者之间建立单向连接。
  - 中介者清除了发送者和请求者之间的直接连接，强制它们通过一个中介对象进行间接沟通。
  - 观察者允许接收者动态地订阅或取消接收请求。

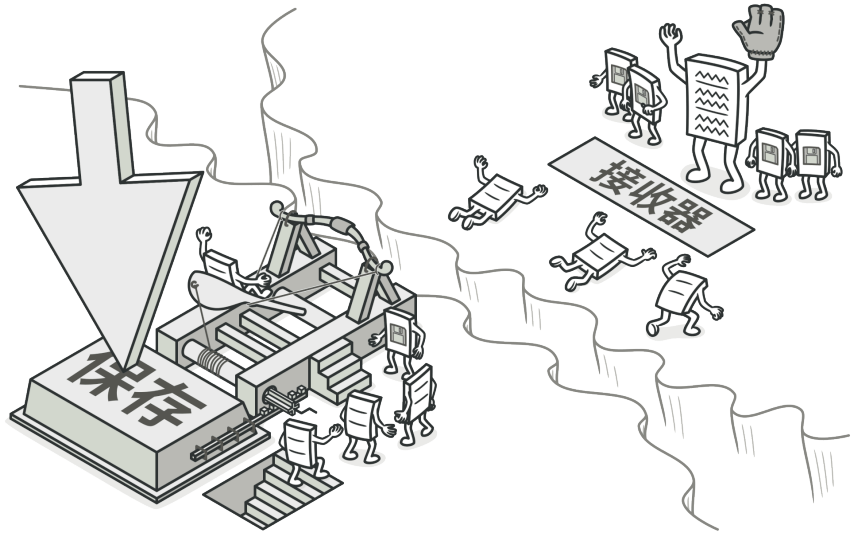
- **责任链**通常和**组合**模式结合使用。在这种情况下，叶组件接收到请求后，可以将请求沿包含全体父组件的链一直传递至对象树的底部。

- **责任链**的管理者可使用**命令**模式实现。在这种情况下，你可以对由请求代表的同一个上下文对象执行许多不同的操作。

  还有另外一种实现方式，那就是请求自身就是一个命令对象。在这种情况下，你可以对由一系列不同上下文连接而成的链执行相同的操作。

- **责任链**和**装饰**模式的类结构非常相似。两者都依赖递归组合将需要执行的操作传递给一系列对象。但是，两者有几点重要的不同之处。

  **责任链**的管理者可以相互独立地执行一切操作，还可以随时停止传递请求。另一方面，各种装饰可以在遵循基本接口的情况下扩展对象的行为。此外，装饰无法中断请求的传递。



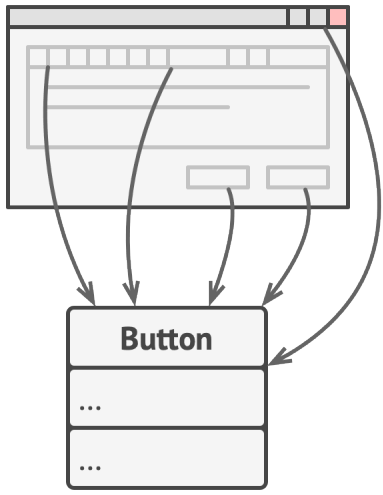

# 命令

亦称：动作、事务、Action、Transaction、Command

**命令**是一种行为设计模式，它可将请求转换为一个包含与请求相关的所有信息的独立对象。该转换让你能根据不同的请求将方法参数化、延迟请求执行或将其放入队列中，且能实现可撤销操作。

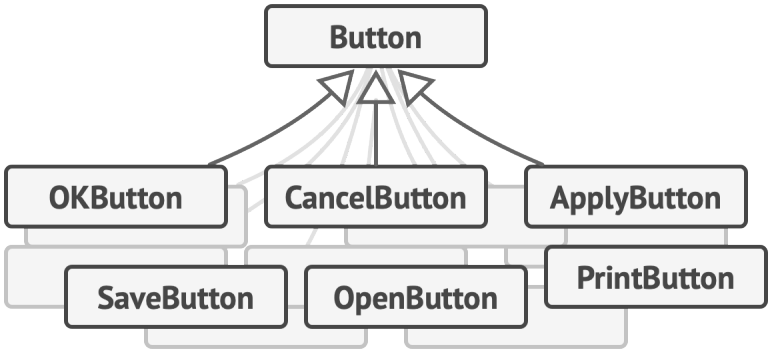
# ☹ 问题

假如你正在开发一款新的文字编辑器，当前的任务是创建一个包含多个按钮的工具栏，并让每个按钮对应编辑器的不同操作。你创建了一个非常简洁的 `按钮` 类，它不仅可用于生成工具栏上的按钮，还可用于生成各种对话框的通用按钮。




应用中的所有按钮都可以继承相同的类

尽管所有按钮看上去都很相似，但它们可以完成不同的操作（打开、保存、打印和应用等）。你会在哪里放置这些按钮的点击处理代码呢？最简单的解决方案是在使用按钮的每个地方都创建大量的子类。这些子类中包含按钮点击后必须执行的代码。




大量的按钮子类。没关系的。

你很快就意识到这种方式有严重缺陷。首先，你创建了大量的子类，当每次修改基类 `按钮` 时，你都有可能需要修改所有子类的代码。简单来说，GUI 代码以一种拙劣的方式依赖于业务逻辑中的不稳定代码。


多个类实现同一功能。

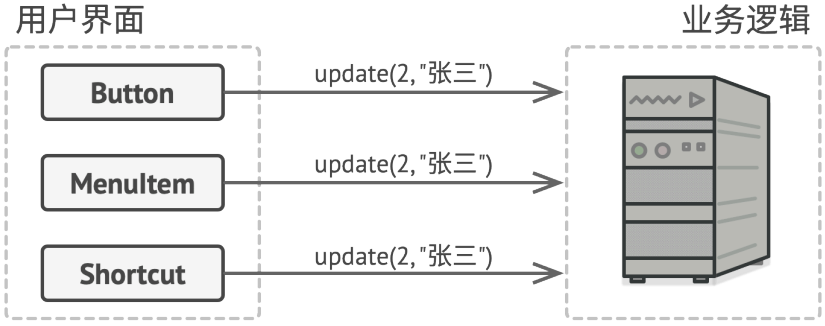
还有一个部分最难办。复制/粘贴文字等操作可能会在多个地方被调用。例如用户可以点击工具栏上小小的“复制”按钮，或者通过上下文菜单复制一些内容，又或者直接使用键盘上的 `Ctrl+C` 。

我们的程序最初只有工具栏，因此可以使用按钮子类来实现各种不同操作。换句话来说，`复制按钮 CopyButton` 子类包含复制文字的代码是可行的。在实现了上下文菜单、快捷方式和其他功能后，你要么需要将操作代码复制进许多个类中，要么需要让菜单依赖于按钮，而后者是更糟糕的选择。

## 😊解决方案

优秀的软件设计通常会将关注点进行分离，而这往往会导致软件的分层。最常见的例子：一层负责用户图像界面；另一层负责业务逻辑。GUI 层负责在屏幕上渲染美观的图形，捕获所有输入并显示用户和程序工作的结果。当需要完成一些重要内容时（比如计算月球轨道或撰写年度报告），GUI 层则会将工作委派给业务逻辑底层。

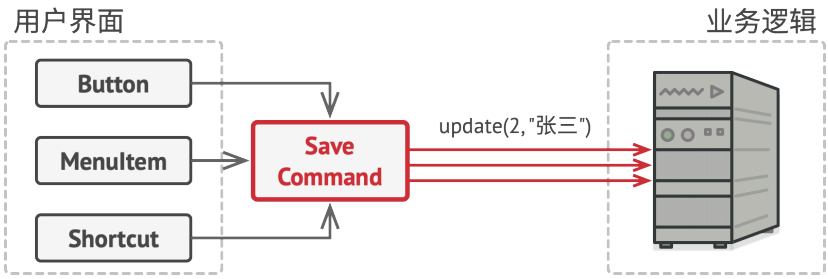
这在代码中看上去就像这样：一个 GUI 对象传递一些参数来调用一个业务逻辑对象。这个过程通常被描述为一个对象发送请求给另一个对象。




GUI 层可以直接访问业务逻辑层。

命令模式建议 GUI 对象不直接提交这些请求。你应该将请求的所有细节（例如调用的对象、方法名称和参数列表）抽取出来组成**命令**类，该类中仅包含一个用于触发请求的方法。

命令对象负责连接不同的 GUI 和业务逻辑对象。此后，GUI 对象无需了解业务逻辑对象是否获得了请求，也无需了解其对请求进行处理的方式。GUI 对象触发命令即可，命令对象会自行处理所有细节工作。

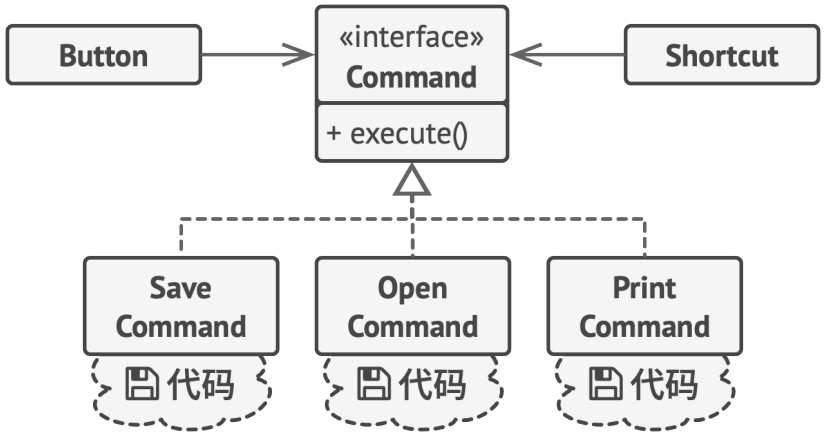



通过命令访问业务逻辑层。

下一步是让所有命令实现相同的接口。该接口通常只有一个没有任何参数的执行方法，让你能在不和具体命令类耦合的情况下使用同一请求发送者执行不同命令。此外还有额外的好处，现在你能在运行时切换连接至发送者的命令对象，以此改变发送者的行为。

你可能会注意到遗漏的一块拼图——请求的参数。GUI 对象可以给业务层对象提供一些参数。但执行命令方法没有任何

参数，所以我们如何将请求的详情发送给接收者呢？答案是：使用数据对命令进行预先配置，或者让其能够自行获取数据。




GUI 对象将命令委派给命令对象。

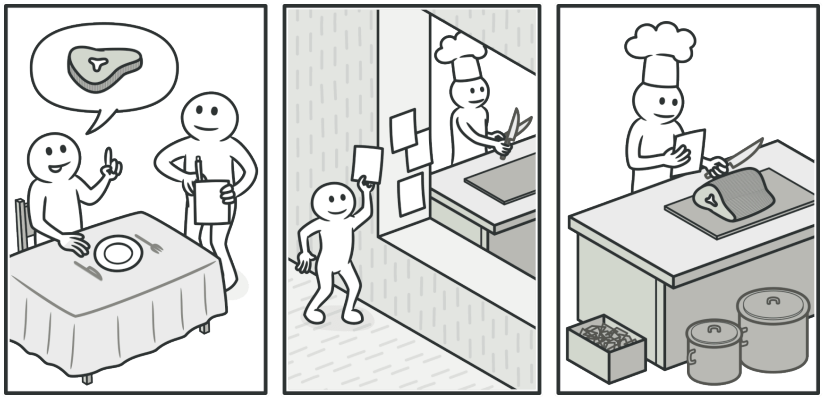
让我们回到文本编辑器。应用命令模式后，我们不再需要任何按钮子类来实现点击行为。我们只需在 按钮 Button 基类中添加一个成员变量来存储对于命令对象的引用，并在点击后执行该命令即可。

你需要为每个可能的操作实现一系列命令类，并且根据按钮所需行为将命令和按钮连接起来。

其他菜单、快捷方式或整个对话框等 GUI 元素都可以通过相同方式来实现。当用户与 GUI 元素交互时，与其连接的命令将会被执行。现在你很可能已经猜到了，与相同操作相关的元素将会被连接到相同的命令，从而避免了重复代码。

最后，命令成为了减少 GUI 和业务逻辑层之间耦合的中间层。而这仅仅是命令模式所提供的一小部分好处！

## 真实世界类比

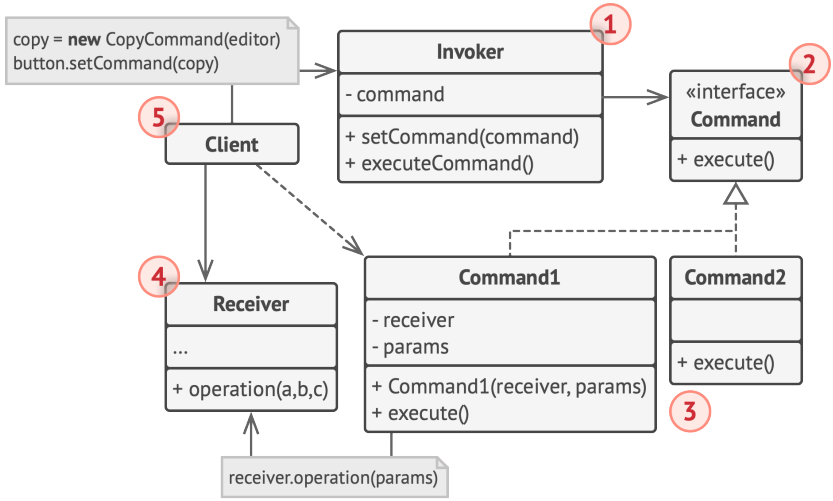


在餐厅里点餐。

在市中心逛了很久的街后，你找到了一家不错的餐厅，坐在了临窗的座位上。一名友善的服务员走近你，迅速记下你点的食物，写在一张纸上。服务员来到厨房，把订单贴在墙上。过了一段时间，厨师拿到了订单，他根据订单来准备食物。厨师将做好的食物和订单一起放在托盘上。服务员看到托盘后对订单进行检查，确保所有食物都是你要的，然后将食物放到了你的桌上。

那张纸就是一个命令，它在厨师开始烹饪前一直位于队列中。命令中包含与烹饪这些食物相关的所有信息。厨师能够根据它马上开始烹饪，而无需跑来直接和你确认订单详情。

## 结构




1. **发送者**（Sender）——亦称“触发者（Invoker）”——类负责对请求进行初始化，其中必须包含一个成员变量来存储对于命令对象的引用。发送者触发命令，而不向接收者直接发送请求。注意，发送者并不负责创建命令对象：它通常会通过构造函数从客户端处获得预先生成的命令。

2. **命令**（Command）接口通常仅声明一个执行命令的方法。

3. **具体命令**（Concrete Commands）会实现各种类型的请求。具体命令自身并不完成工作，而是会将调用委派给一个业务逻辑对象。但为了简化代码，这些类可以进行合并。

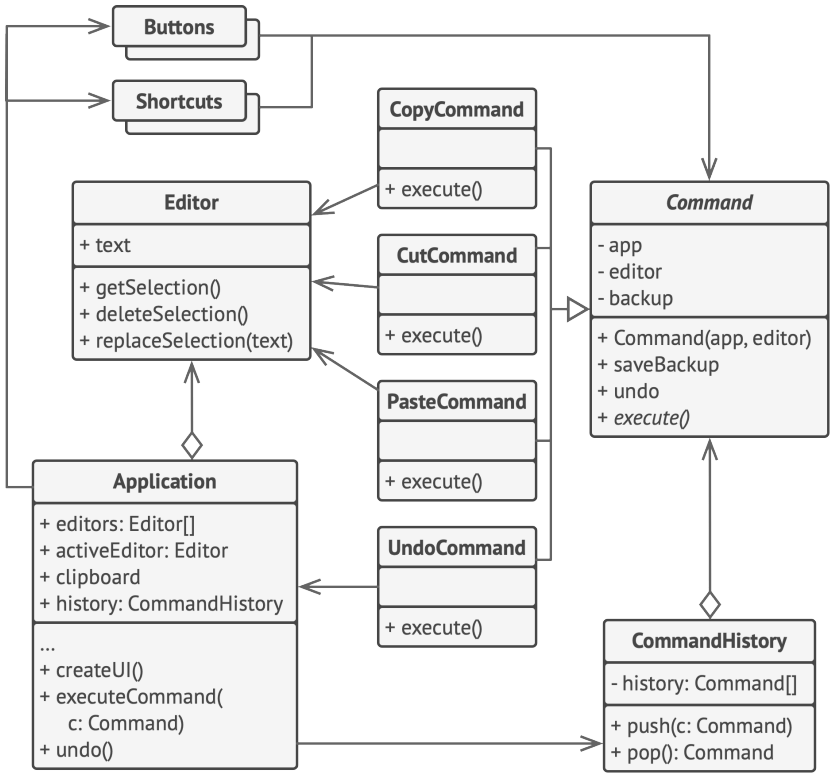
   接收对象执行方法所需的参数可以声明为具体命令的成员变量。你可以将命令对象设为不可变，仅允许通过构造函数对这些成员变量进行初始化。

4. **接收者**（Receiver）类包含部分业务逻辑。几乎任何对象都可以作为接收者。绝大部分命令只处理如何将请求传递到接收者的细节，接收者自己会完成实际的工作。

5. **客户端**（Client）会创建并配置具体命令对象。客户端必须将包括接收者实体在内的所有请求参数传递给命令的构造函数。此后，生成的命令就可以与一个或多个发送者相关联了。

# # 伪代码

在本例中，**命令**模式会记录已执行操作的历史记录，以在需要时撤销操作。




文本编辑器中的可撤销操作。

有些命令会改变编辑器的状态（例如剪切和粘贴），它们可在执行相关操作前对编辑器的状态进行备份。命令执行后会和当前点备份的编辑器状态一起被放入命令历史（命令对象栈）。此后，如果用户需要进行回滚操作，程序可从历史记录中取出最近的命令，读取相应的编辑器状态备份，然后进行恢复。

客户端代码（GUI 元素和命令历史等）没有和具体命令类相耦合，因为它通过命令接口来使用命令。这使得你能在无需修改已有代码的情况下在程序中增加新的命令。

```
// 命令基类会为所有具体命令定义通用接口。
abstract class Command is
  protected field app: Application
  protected field editor: Editor
  protected field backup: text

  constructor Command(app: Application, editor: Editor) is
    this.app = app
    this.editor = editor

  // 备份编辑器状态。
  method saveBackup() is
    backup = editor.text

  // 恢复编辑器状态。
  method undo() is
    editor.text = backup

  // 执行方法被声明为抽象以强制所有具体命令提供自己的实现。该方法必须根
  // 据命令是否更改编辑器的状态返回 true 或 false。
  abstract method execute()


// 这里是具体命令。
class CopyCommand extends Command is
  // 复制命令不会被保存到历史记录中，因为它没有改变编辑器的状态。
  method execute() is
```

```
    app.clipboard = editor.getSelection()
    return false

class CutCommand extends Command is
  // 剪切命令改变了编辑器的状态，因此它必须被保存到历史记录中。只要方法
  // 返回 true，它就会被保存。
  method execute() is
    saveBackup()
    app.clipboard = editor.getSelection()
    editor.deleteSelection()
    return true

class PasteCommand extends Command is
  method execute() is
    saveBackup()
    editor.replaceSelection(app.clipboard)
    return true

// 撤销操作也是一个命令。
class UndoCommand extends Command is
  method execute() is
    app.undo()
    return false


// 全局命令历史记录就是一个堆栈。
class CommandHistory is
  private field history: array of Command

  // 后进...
  method push(c: Command) is
    // 将命令压入历史记录数组的末尾。
```

```
60
61    // ...先出
62    method pop():Command is
63      // 从历史记录中取出最近的命令。
64
65
66  // 编辑器类包含实际的文本编辑操作。它会担任接收者的角色：最后所有命令都会
67  // 将执行工作委派给编辑器的方法。
68  class Editor is
69    field text: string
70
71    method getSelection() is
72      // 返回选中的文字。
73
74    method deleteSelection() is
75      // 删除选中的文字。
76
77    method replaceSelection(text) is
78      // 在当前位置插入剪贴板中的内容。
79
80  // 应用程序类会设置对象之间的关系。它会担任发送者的角色：当需要完成某些工
81  // 作时，它会创建并执行一个命令对象。
82  class Application is
83    field clipboard: string
84    field editors: array of Editors
85    field activeEditor: Editor
86    field history: CommandHistory
87
88    // 将命令分派给 UI 对象的代码可能会是这样的。
89    method createUI() is
90      // ...
91      copy = function() { executeCommand(
```

```
92          new CopyCommand(this, activeEditor)) }
93        copyButton.setCommand(copy)
94        shortcuts.onKeyPress("Ctrl+C", copy)
95
96        cut = function() { executeCommand(
97          new CutCommand(this, activeEditor)) }
98        cutButton.setCommand(cut)
99        shortcuts.onKeyPress("Ctrl+X", cut)
100
101       paste = function() { executeCommand(
102         new PasteCommand(this, activeEditor)) }
103       pasteButton.setCommand(paste)
104       shortcuts.onKeyPress("Ctrl+V", paste)
105
106       undo = function() { executeCommand(
107         new UndoCommand(this, activeEditor)) }
108       undoButton.setCommand(undo)
109       shortcuts.onKeyPress("Ctrl+Z", undo)
110
111     // 执行一个命令并检查它是否需要被添加到历史记录中。
112     method executeCommand(command) is
113       if (command.execute)
114         history.push(command)
115
116     // 从历史记录中取出最近的命令并运行其 undo（撤销）方法。请注意，你并
117     // 不知晓该命令所属的类。但是我们不需要知晓，因为命令自己知道如何撤销
118     // 其动作。
119     method undo() is
120       command = history.pop()
121       if (command != null)
122         command.undo()
```

# 适合应用场景

**如果你需要通过操作来参数化对象，可使用命令模式。**

命令模式可将特定的方法调用转化为独立对象。这一改变也带来了许多有趣的应用：你可以将命令作为方法的参数进行传递、将命令保存在其他对象中，或者在运行时切换已连接的命令等。

举个例子：你正在开发一个 GUI 组件（例如上下文菜单），你希望用户能够配置菜单项，并在点击菜单项时触发操作。

**如果你想要将操作放入队列中、操作的执行或者远程执行操作，可使用命令模式。**

同其他对象一样，命令也可以实现序列化（序列化的意思是转化为字符串），从而能方便地写入文件或数据库中。一段时间后，该字符串可被恢复成为最初的命令对象。因此，你可以延迟或计划命令的执行。但其功能远不止如此！使用同样的方式，你还可以将命令放入队列、记录命令或者通过网络发送命令。

**如果你想要实现操作回滚功能，可使用命令模式。**

尽管有很多方法可以实现撤销和恢复功能，但命令模式可能是其中最常用的一种。

为了能够回滚操作，你需要实现已执行操作的历史记录功能。命令历史记录是一种包含所有已执行命令对象及其相关程序状态备份的栈结构。

这种方法有两个缺点。首先，程序状态的保存功能并不容易实现，因为部分状态可能是私有的。你可以使用**备忘录**模式来在一定程度上解决这个问题。

其次，备份状态可能会占用大量内存。因此，有时你需要借助另一种实现方式：命令无需恢复原始状态，而是执行反向操作。反向操作也有代价：它可能会很难甚至是无法实现。

## 实现方式

1. 声明仅有一个执行方法的命令接口。

2. 抽取请求并使之成为实现命令接口的具体命令类。每个类都必须有一组成员变量来保存请求参数和对于实际接收者对象的引用。所有这些变量的数值都必须通过命令构造函数进行初始化。

3. 找到担任发送者职责的类。在这些类中添加保存命令的成员变量。发送者只能通过命令接口与其命令进行交互。发送者自身通常并不创建命令对象，而是通过客户端代码获取。

4. 修改发送者使其执行命令，而非直接将请求发送给接收者。

5. 客户端必须按照以下顺序来初始化对象：

    - 创建接收者。
    - 创建命令，如有需要可将其关联至接收者。
    - 创建发送者并将其与特定命令关联。

## ⚖ 优缺点

- ✓ 单一职责原则。你可以解耦触发和执行操作的类。
- ✓ 开闭原则。你可以在不修改已有客户端代码的情况下在程序中创建新的命令。
- ✓ 你可以实现撤销和恢复功能。
- ✓ 你可以实现操作的延迟执行。
- ✓ 你可以将一组简单命令组合成一个复杂命令。
- ✕ 代码可能会变得更加复杂，因为你在发送者和接收者之间增加了一个全新的层次。

## ⇄ 与其他模式的关系

- **责任链**、**命令**、**中介者**和**观察者**用于处理请求发送者和接收者之间的不同连接方式：
    - 责任链按照顺序将请求动态传递给一系列的潜在接收者，直至其中一名接收者对请求进行处理。

  - **命令**在发送者和请求者之间建立单向连接。
  - **中介者**清除了发送者和请求者之间的直接连接，强制它们通过一个中介对象进行间接沟通。
  - **观察者**允许接收者动态地订阅或取消接收请求。

- **责任链**的管理者可使用**命令**模式实现。在这种情况下，你可以对由请求代表的同一个上下文对象执行许多不同的操作。

  还有另外一种实现方式，那就是请求自身就是一个**命令**对象。在这种情况下，你可以对由一系列不同上下文连接而成的链执行相同的操作。

- 你可以同时使用**命令**和**备忘录**来实现“撤销”。在这种情况下，命令用于对目标对象执行各种不同的操作，备忘录用来保存一条命令执行前该对象的状态。

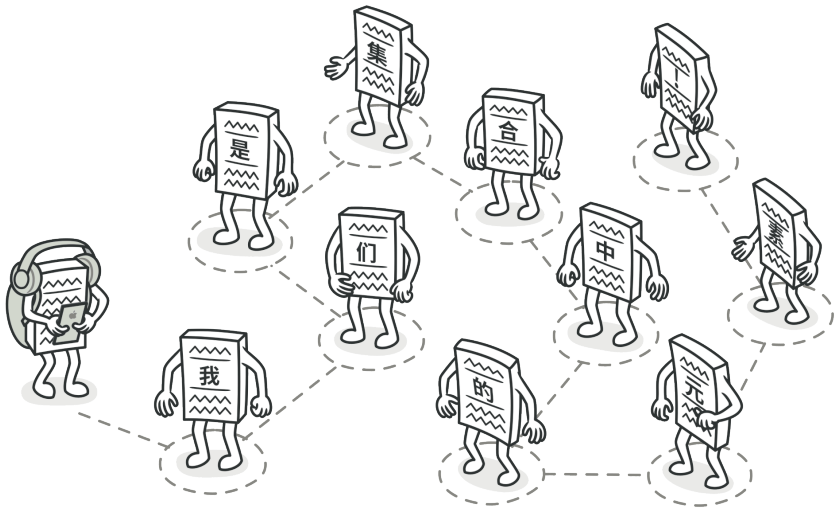
- **命令**和**策略**看上去很像，因为两者都能通过某些行为来参数化对象。但是，它们的意图有非常大的不同。

  - 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。

- **原型**可用于保存**命令**的历史记录。

- 你可以将**访问者**视为**命令**模式的加强版本，其对象可对不同类的多种对象执行操作。



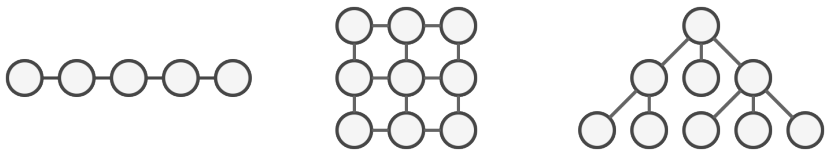

# 迭代器

亦称：Iterator

**迭代器**是一种行为设计模式，让你能在不暴露集合底层表现形式（列表、栈和树等）的情况下遍历集合中所有的元素。

# ☹ 问题

集合是编程中最常使用的数据类型之一。尽管如此，集合只是一组对象的容器而已。

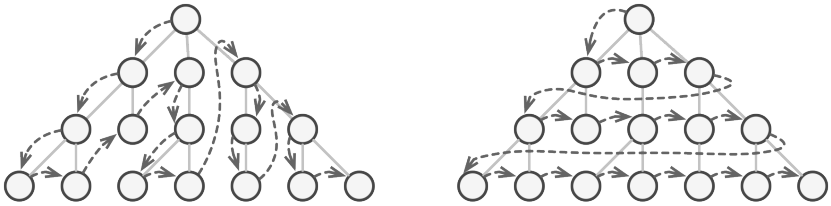


各种类型的集合。

大部分集合使用简单列表存储元素。但有些集合还会使用栈、树、图和其他复杂的数据结构。

无论集合的构成方式如何，它都必须提供某种访问元素的方式，便于其他代码使用其中的元素。集合应提供一种能够遍历元素的方式，且保证它不会周而复始地访问同一个元素。

如果你的集合基于列表，那么这项工作听上去仿佛很简单。但如何遍历复杂数据结构（例如树）中的元素呢？例如，今天你需要使用深度优先算法来遍历树结构，明天可能会需要广度优先算法；下周则可能会需要其他方式（比如随机存取树中的元素）。



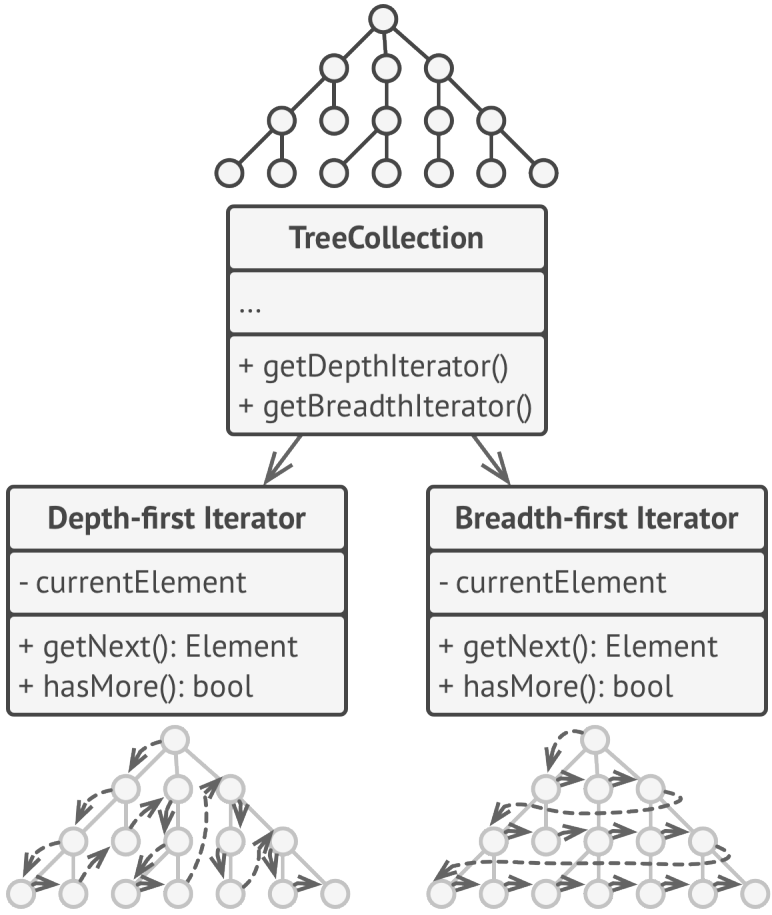
可通过不同的方式遍历相同的集合。

不断向集合中添加遍历算法会模糊其“高效存储数据”的主要职责。此外，有些算法可能是根据特定应用订制的，将其加入泛型集合类中会显得非常奇怪。

另一方面，使用多种集合的客户端代码可能并不关心存储数据的方式。不过由于集合提供不同的元素访问方式，你的代码将不得不与特定集合类进行耦合。

## 😊解决方案

迭代器模式的主要思想是将集合的遍历行为抽取为单独的**迭代器**对象。



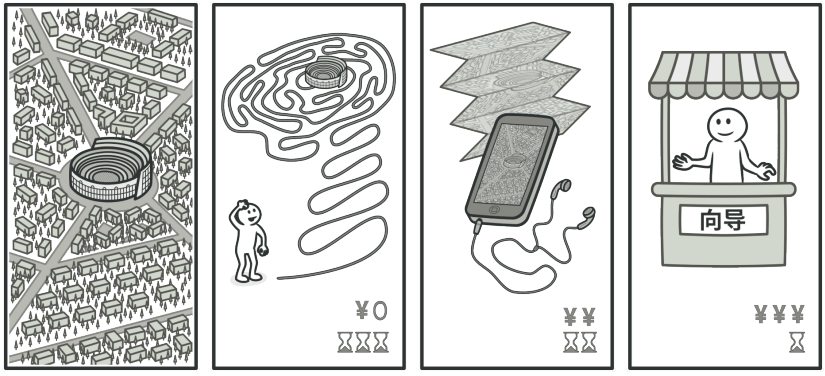

迭代器可实现多种遍历算法。多个迭代器对象可同时遍历同一个集合。

除实现自身算法外，迭代器还封装了遍历操作的所有细节，例如当前位置和末尾剩余元素的数量。因此，多个迭代器可以在相互独立的情况下同时访问集合。

迭代器通常会提供一个获取集合元素的基本方法。客户端可不断调用该方法直至它不返回任何内容，这意味着迭代器已经遍历了所有元素。

所有迭代器必须实现相同的接口。这样一来，只要有合适的迭代器，客户端代码就能兼容任何类型的集合或遍历算法。如果你需要采用特殊方式来遍历集合，只需创建一个新的迭代器类即可，无需对集合或客户端进行修改。

## 真实世界类比




漫步罗马的不同方式。

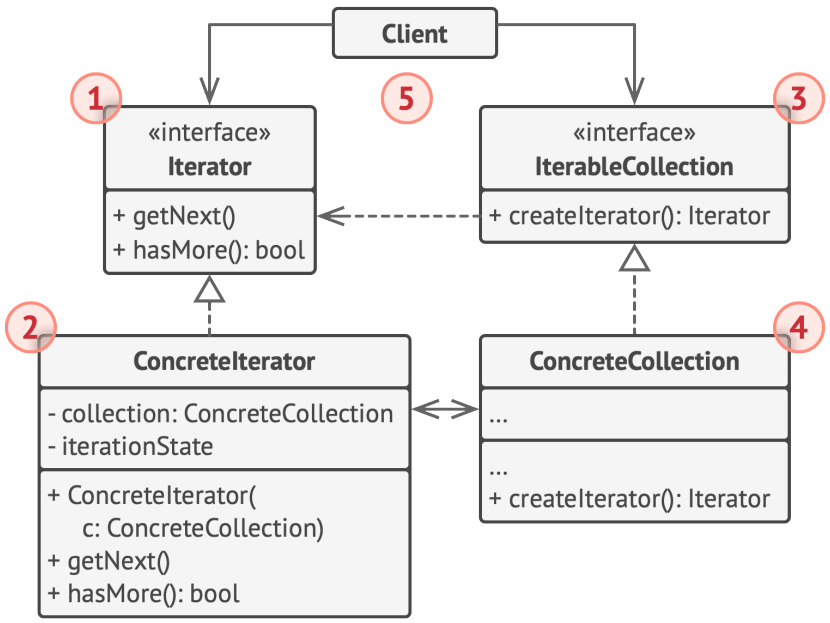
你计划在罗马游览数天，参观所有主要的旅游景点。但在到达目的地后，你可能会浪费很多时间绕圈子，甚至找不到罗马斗兽场在哪里。

或者你可以购买一款智能手机上的虚拟导游程序。这款程序非常智能而且价格不贵，你想在景点待多久都可以。

第三种选择是用部分旅行预算雇佣一位对城市了如指掌的当地向导。向导能根据你的喜好来安排行程，为你介绍每个景点并讲述许多激动人心的故事。这样的旅行可能会更有趣，但所需费用也会更高。

所有这些选择(自由漫步、智能手机导航或真人向导)都是这个由众多罗马景点组成的集合的迭代器。

## 结构



1. **迭代器**（Iterator）接口声明了遍历集合所需的操作：获取下一个元素、获取当前位置和重新开始迭代等。

2. **具体迭代器**（Concrete Iterators）实现遍历集合的一种特定算法。迭代器对象必须跟踪自身遍历的进度。这使得多个迭代器可以相互独立地遍历同一集合。

3. **集合**（Collection）接口声明一个或多个方法来获取与集合兼容的迭代器。请注意，返回方法的类型必须被声明为迭代器接口，因此具体集合可以返回各种不同种类的迭代器。

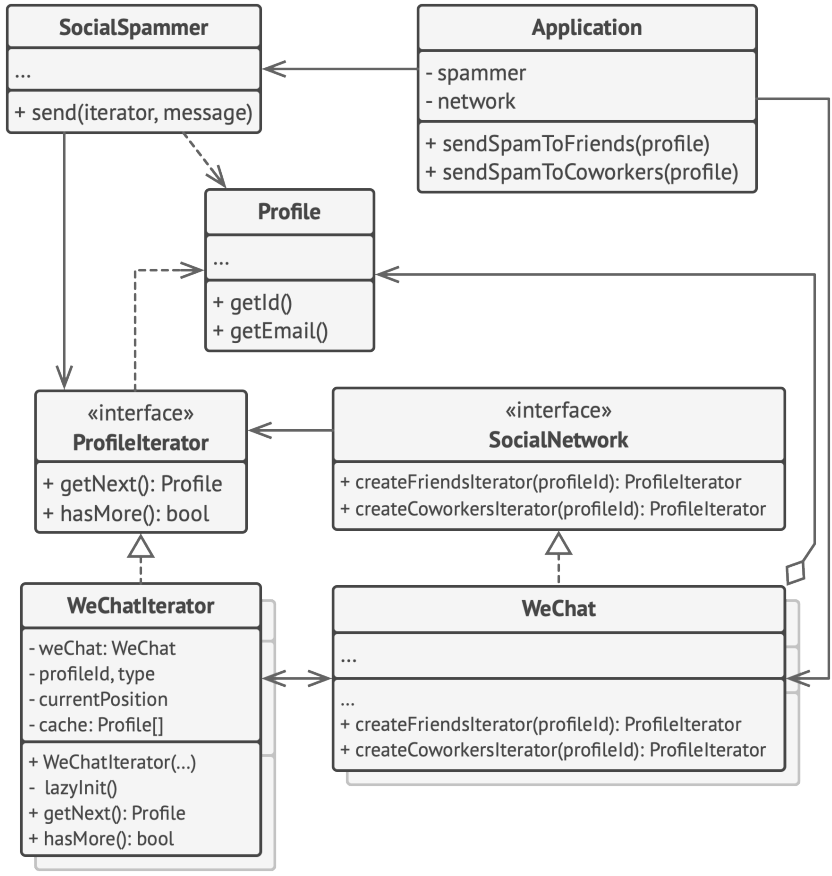
4. **具体集合**（Concrete Collections）会在客户端请求迭代器时返回一个特定的具体迭代器类实体。你可能会琢磨，剩下的集合代码在什么地方呢？不用担心，它也会在同一个类中。只是这些细节对于实际模式来说并不重要，所以我们将其省略了而已。

5. **客户端**（Client）通过集合和迭代器的接口与两者进行交互。这样一来客户端无需与具体类进行耦合，允许同一客户端代码使用各种不同的集合和迭代器。

   客户端通常不会自行创建迭代器，而是会从集合中获取。但在特定情况下，客户端可以直接创建一个迭代器（例如当客户端需要自定义特殊迭代器时）。

# 伪代码

在本例中，**迭代器**模式用于遍历一个封装了访问微信好友关系功能的特殊集合。该集合提供使用不同方式遍历档案资料的多个迭代器。




遍历社交档案的示例

“好友（friends）”迭代器可用于遍历指定档案的好友。“同事（colleagues）”迭代器也提供同样的功能，但仅包括与目标用户在同一家公司工作的好友。这两个迭代器都实现了同一个通用接口，客户端能在不了解认证和发送 REST 请求等实现细节的情况下获取档案。

客户端仅通过接口与集合和迭代器交互，也就不会同具体类耦合。如果你决定将应用连接到全新的社交网络，只需提供新的集合和迭代器类即可，无需修改现有代码。

```
// 集合接口必须声明一个用于生成迭代器的工厂方法。如果程序中有不同类型的迭
// 代器，你也可以声明多个方法。
interface SocialNetwork is
  method createFriendsIterator(profileId):ProfileIterator
  method createCoworkersIterator(profileId):ProfileIterator


// 每个具体集合都与其返回的一组具体迭代器相耦合。但客户并不是这样的，因为
// 这些方法的签名将会返回迭代器接口。
class WeChat implements SocialNetwork is
  // ...大量的集合代码应该放在这里...

  // 迭代器创建代码。
  method createFriendsIterator(profileId) is
    return new WeChatIterator(this, profileId, "friends")
  method createCoworkersIterator(profileId) is
    return new WeChatIterator(this, profileId, "coworkers")


```

```
// 所有迭代器的通用接口。
interface ProfileIterator is
  method getNext():Profile
  method hasMore():bool


// 具体迭代器类。
class WeChatIterator implements ProfileIterator is
  // 迭代器需要一个指向其遍历集合的引用。
  private field weChat: WeChat
  private field profileId, type: string

  // 迭代器对象会独立于其他迭代器来对集合进行遍历。因此它必须保存迭代器
  // 的状态。
  private field currentPosition
  private field cache: array of Profile

  constructor WeChatIterator(weChat, profileId, type) is
    this.weChat = weChat
    this.profileId = profileId
    this.type = type

  private method lazyInit() is
    if (cache == null)
      cache = weChat.socialGraphRequest(profileId, type)

  // 每个具体迭代器类都会自行实现通用迭代器接口。
  method getNext() is
    if (hasMore())
      currentPosition++
      return cache[currentPosition]

```

```
    method hasMore() is
      lazyInit()
      return currentPosition < cache.length


// 这里还有一个有用的绝招：你可将迭代器传递给客户端类，无需让其拥有访问整
// 个集合的权限。这样一来，你就无需将集合暴露给客户端了。
//
// 还有另一个好处：你可在运行时将不同的迭代器传递给客户端，从而改变客户端
// 与集合互动的方式。这一方法可行的原因是客户端代码并没有和具体迭代器类相
// 耦合。
class SocialSpammer is
  method send(iterator: ProfileIterator, message: string) is
    while (iterator.hasMore())
      profile = iterator.getNext()
      System.sendEmail(profile.getEmail(), message)


// 应用程序（Application）类可对集合和迭代器进行配置，然后将其传递给客户
// 端代码。
class Application is
  field network: SocialNetwork
  field spammer: SocialSpammer

  method config() is
    if working with WeChat
      this.network = new WeChat()
    if working with LinkedIn
      this.network = new LinkedIn()
    this.spammer = new SocialSpammer()

  method sendSpamToFriends(profile) is
```

```
    iterator = network.createFriendsIterator(profile.getId())
    spammer.send(iterator, "非常重要的消息")

  method sendSpamToCoworkers(profile) is
    iterator = network.createCoworkersIterator(profile.getId())
    spammer.send(iterator, "非常重要的消息")
```

## 适合应用场景

**当集合背后为复杂的数据结构，且你希望对客户端隐藏其复杂性时（出于使用便利性或安全性的考虑），可以使用迭代器模式。**

迭代器封装了与复杂数据结构进行交互的细节，为客户端提供多个访问集合元素的简单方法。这种方式不仅对客户端来说非常方便，而且能避免客户端在直接与集合交互时执行错误或有害的操作，从而起到保护集合的作用。

**使用该模式可以减少程序中重复的遍历代码。**

重要迭代算法的代码往往体积非常庞大。当这些代码被放置在程序业务逻辑中时，它会让原始代码的职责模糊不清，降低其可维护性。因此，将遍历代码移到特定的迭代器中可使程序代码更加精炼和简洁。

**如果你希望代码能够遍历不同的甚至是无法预知的数据结构，可以使用迭代器模式。**

该模式为集合和迭代器提供了一些通用接口。如果你在代码中使用了这些接口，那么将其他实现了这些接口的集合和迭代器传递给它时，它仍将可以正常运行。

## 实现方式

1. 声明迭代器接口。该接口必须提供至少一个方法来获取集合中的下个元素。但为了使用方便，你还可以添加一些其他方法，例如获取前一个元素、记录当前位置和判断迭代是否已结束。

2. 声明集合接口并描述一个获取迭代器的方法。其返回值必须是迭代器接口。如果你计划拥有多组不同的迭代器，则可以声明多个类似的方法。

3. 为希望使用迭代器进行遍历的集合实现具体迭代器类。迭代器对象必须与单个集合实体链接。链接关系通常通过迭代器的构造函数建立。

4. 在你的集合类中实现集合接口。其主要思想是针对特定集合为客户端代码提供创建迭代器的快捷方式。集合对象必须将自身传递给迭代器的构造函数来创建两者之间的链接。

5. 检查客户端代码，使用迭代器替代所有集合遍历代码。每当客户端需要遍历集合元素时都会获取一个新的迭代器。

## ⚖ 优缺点

✓ 单一职责原则。通过将体积庞大的遍历算法代码抽取为独立的类，你可对客户端代码和集合进行整理。

✓ 开闭原则。你可实现新型的集合和迭代器并将其传递给现有代码，无需修改现有代码。

✓ 你可以并行遍历同一集合，因为每个迭代器对象都包含其自身的遍历状态。

✓ 相似的，你可以暂停遍历并在需要时继续。

✕ 如果你的程序只与简单的集合进行交互，应用该模式可能会矫枉过正。

✕ 对于某些特殊集合，使用迭代器可能比直接遍历的效率低。

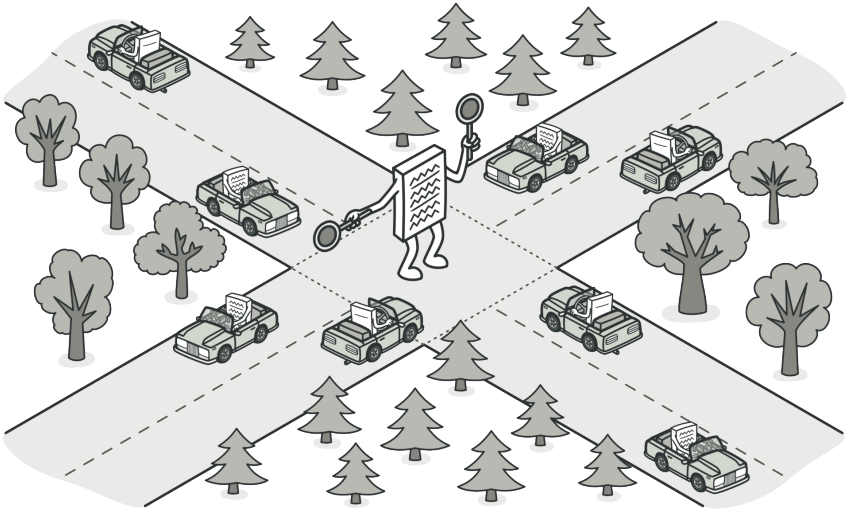
## ⇄ 与其他模式的关系

- 你可以使用**迭代器**来遍历**组合**树。

- 你可以同时使用**工厂方法**和**迭代器**来让子类集合返回不同类型的迭代器，并使得迭代器与集合相匹配。

- 你可以同时使用**备忘录**和**迭代器**来获取当前迭代器的状态，并且在需要的时候进行回滚。

- 可以同时使用**访问者**和**迭代器**来遍历复杂数据结构，并对其中的元素执行所需操作，即使这些元素所属的类完全不同。



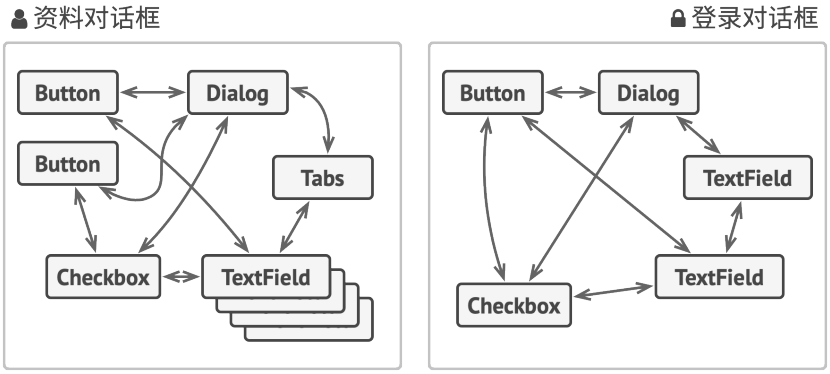
# 中介者

亦称：调解人、控制器、Intermediary、Controller、Mediator

**中介者**是一种行为设计模式，能让你减少对象之间混乱无序的依赖关系。该模式会限制对象之间的直接交互，迫使它们通过一个中介者对象进行合作。

# 问题

假如你有一个创建和修改客户资料的对话框，它由各种控件组成，例如文本框（TextField）、复选框（Checkbox）和按钮（Button）等。




用户界面中各元素间的关系会随程序发展而变得混乱。

某些表单元素可能会直接进行互动。例如，选中“我有一只狗”复选框后可能会显示一个隐藏文本框用于输入狗狗的名字。另一个例子是提交按钮必须在保存数据前校验所有输入内容。

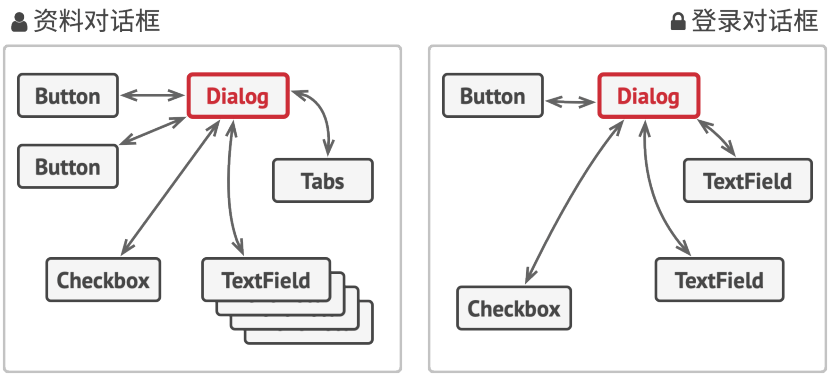

元素间存在许多关联。因此，对某些元素进行修改可能会影响其他元素。

如果直接在表单元素代码中实现业务逻辑，你将很难在程序其他表单中复用这些元素类。例如，由于复选框类与狗狗的文本框相耦合，所以将无法在其他表单中使用它。你要么使用渲染资料表单时用到的所有类，要么一个都不用。

## 😊解决方案

中介者模式建议你停止组件之间的直接交流并使其相互独立。这些组件必须调用特殊的中介者对象，通过中介者对象重定向调用行为，以间接的方式进行合作。最终，组件仅依赖于一个中介者类，无需与多个其他组件相耦合。

在资料编辑表单的例子中，对话框（Dialog）类本身将作为中介者，其很可能已知自己所有的子元素，因此你甚至无需在该类中引入新的依赖关系。



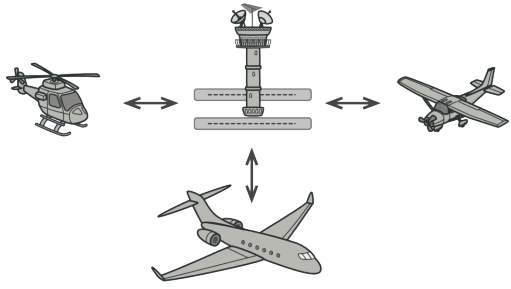

UI 元素必须通过中介者对象进行间接沟通。

绝大部分重要的修改都在实际表单元素中进行。让我们想想提交按钮。之前，当用户点击按钮后，它必须对所有表单元素数值进行校验。而现在它的唯一工作是将点击事件通知给对话框。收到通知后，对话框可以自行校验数值或将任务委派给各元素。这样一来，按钮不再与多个表单元素相关联，而仅依赖于对话框类。

你还可以为所有类型的对话框抽取通用接口，进一步削弱其依赖性。接口中将声明一个所有表单元素都能使用的通知方法，可用于将元素中发生的事件通知给对话框。这样一来，所有实现了该接口的对话框都能使用这个提交按钮了。

采用这种方式，中介者模式让你能在单个中介者对象中封装多个对象间的复杂关系网。类所拥有的依赖关系越少，就越易于修改、扩展或复用。

## 真实世界类比

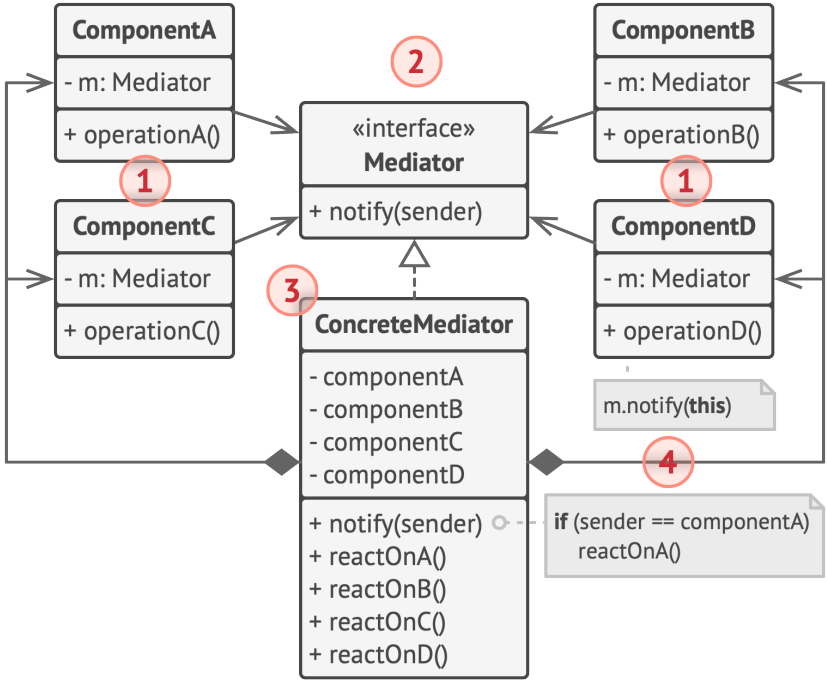


飞行器驾驶员之间不会通过相互沟通来决定下一架降落的飞机。所有沟通都通过控制塔台进行。

飞行器驾驶员们在靠近或离开空中管制区域时不会直接相互交流。但他们会与飞机跑道附近，塔台中的空管员通话。如果没有空管员，驾驶员就需要留意机场附近的所有飞机，并与数十位飞行员组成的委员会讨论降落顺序。那恐怕会让飞机坠毁的统计数据一飞冲天吧。

塔台无需管制飞行全程，只需在航站区加强管控即可，因为该区域的决策参与者数量对于飞行员来说实在太多了。

## 结构



1. **组件**（Component）是各种包含业务逻辑的类。每个组件都有一个指向中介者的引用，该引用被声明为中介者接口类型。组件不知道中介者实际所属的类，因此你可通过将其连接到不同的中介者以使其能在其他程序中复用。

2. **中介者**（Mediator）接口声明了与组件交流的方法，但通常仅包括一个通知方法。组件可将任意上下文（包括自己的对象）作为该方法的参数，只有这样接收组件和发送者类之间才不会耦合。

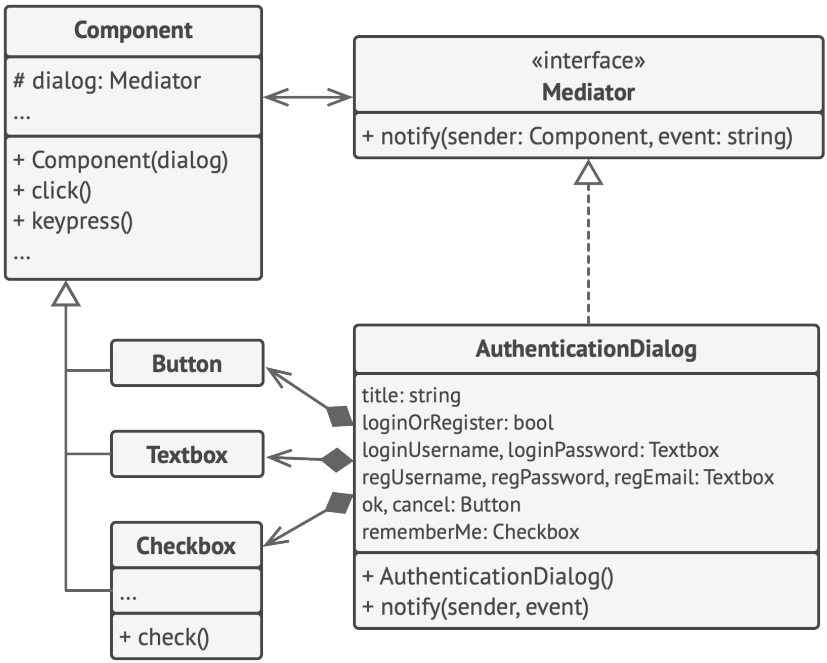
3. **具体中介者**（Concrete Mediator）封装了多种组件间的关系。具体中介者通常会保存所有组件的引用并对其进行管理，甚至有时会对其生命周期进行管理。

4. 组件并不知道其他组件的情况。如果组件内发生了重要事件，它只能通知中介者。中介者收到通知后能轻易地确定发送者，这或许已足以判断接下来需要触发的组件了。

   对于组件来说，中介者看上去完全就是一个黑箱。发送者不知道最终会由谁来处理自己的请求，接收者也不知道最初是谁发出了请求。

# # 伪代码

在本例中，**中介者**模式可帮助你减少各种 UI 类（按钮、复选框和文本标签）之间的相互依赖关系。




UI 对话框类的结构

用户触发的元素不会直接与其他元素交流，即使看上去它们应该这样做。相反，元素只需让中介者知晓事件即可，并能在发出通知时同时传递任何上下文信息。

本例中的中介者是整个认证对话框。对话框知道具体元素应如何进行合作并促进它们的间接交流。当接收到事件通知后，对话框会确定负责处理事件的元素并据此重定向请求。

```
1  // 中介者接口声明了一个能让组件将各种事件通知给中介者的方法。中介者可对这
2  // 些事件做出响应并将执行工作传递给其他组件。
3  interface Mediator is
```

```
    method notify(sender: Component, event: string)


// 具体中介者类可解开各组件之间相互交叉的连接关系并将其转移到中介者中。
class AuthenticationDialog implements Mediator is
    private field title: string
    private field loginOrRegisterChkBx: Checkbox
    private field loginUsername, loginPassword: Textbox
    private field registrationUsername, registrationPassword,
            registrationEmail: Textbox
    private field okBtn, cancelBtn: Button

    constructor AuthenticationDialog() is
        // 创建所有组件对象并将当前中介者传递给其构造函数以建立连接。

    // 当组件中有事件发生时，它会通知中介者。中介者接收到通知后可自行处理，
    // 也可将请求传递给另一个组件。
    method notify(sender, event) is
        if (sender == loginOrRegisterChkBx and event == "check")
            if (loginOrRegisterChkBx.checked)
                title = "登录"
                // 1. 显示登录表单组件。
                // 2. 隐藏注册表单组件。
            else
                title = "注册"
                // 1. 显示注册表单组件。
                // 2. 隐藏登录表单组件。

        if (sender == okBtn && event == "click")
            if (loginOrRegister.checked)
                // 尝试找到使用登录信息的用户。
                if (!found)
```

```
            // 在登录字段上方显示错误信息。
        else
          // 1. 使用注册字段中的数据创建用户账号。
          // 2. 完成用户登录工作。 …


// 组件会使用中介者接口与中介者进行交互。因此只需将它们与不同的中介者连接
// 起来，你就能在其他情境中使用这些组件了。
class Component is
  field dialog: Mediator

  constructor Component(dialog) is
    this.dialog = dialog

  method click() is
    dialog.notify(this, "click")

  method keypress() is
    dialog.notify(this, "keypress")

// 具体组件之间无法进行交流。它们只有一个交流渠道，那就是向中介者发送通知。
class Button extends Component is
  // ...

class Textbox extends Component is
  // ...

class Checkbox extends Component is
  method check() is
    dialog.notify(this, "check")
  // ...
```

# 适合应用场景

**当一些对象和其他对象紧密耦合以致难以对其进行修改时，可使用中介者模式。**

该模式让你将对象间的所有关系抽取成为一个单独的类，以使对于特定组件的修改工作独立于其他组件。

**当组件因过于依赖其他组件而无法在不同应用中复用时，可使用中介者模式。**

应用中介者模式后，每个组件不再知晓其他组件的情况。尽管这些组件无法直接交流，但它们仍可通过中介者对象进行间接交流。如果你希望在不同应用中复用一个组件，则需要为其提供一个新的中介者类。

**如果为了能在不同情景下复用一些基本行为，导致你需要被迫创建大量组件子类时，可使用中介者模式。**

由于所有组件间关系都被包含在中介者中，因此你无需修改组件就能方便地新建中介者类以定义新的组件合作方式。

# 实现方式

1. 找到一组当前紧密耦合，且提供其独立性能带来更大好处的类（例如更易于维护或更方便复用）。

2. 声明中介者接口并描述中介者和各种组件之间所需的交流接口。在绝大多数情况下，一个接收组件通知的方法就足够了。如果你希望在不同情景下复用组件类，那么该接口将非常重要。只要组件使用通用接口与其中介者合作，你就能将该组件与不同实现中的中介者进行连接。

3. 实现具体中介者类。该类可从自行保存其下所有组件的引用中受益。

4. 你可以更进一步，让中介者负责组件对象的创建和销毁。此后，中介者可能会与**工厂**或**外观**类似。

5. 组件必须保存对于中介者对象的引用。该连接通常在组件的构造函数中建立，该函数会将中介者对象作为参数传递。

6. 修改组件代码，使其可调用中介者的通知方法，而非其他组件的方法。然后将调用其他组件的代码抽取到中介者类中，并在中介者接收到该组件通知时执行这些代码。

## ⚖ 优缺点

- ✓ 单一职责原则。你可以将多个组件间的交流抽取到同一位置，使其更易于理解和维护。
- ✓ 开闭原则。你无需修改实际组件就能增加新的中介者。
- ✓ 你可以减轻应用中多个组件间的耦合情况。

✓ 你可以更方便地复用各个组件。

✕ 一段时间后，中介者可能会演化成为**上帝对象**。

## ⇄ 与其他模式的关系

- **责任链**、**命令**、**中介者**和**观察者**用于处理请求发送者和接收者之间的不同连接方式：
  - 责任链按照顺序将请求动态传递给一系列的潜在接收者，直至其中一名接收者对请求进行处理。
  - 命令在发送者和请求者之间建立单向连接。
  - 中介者清除了发送者和请求者之间的直接连接，强制它们通过一个中介对象进行间接沟通。
  - 观察者允许接收者动态地订阅或取消接收请求。
- **外观**和**中介者**的职责类似：它们都尝试在大量紧密耦合的类中组织起合作。
  - 外观为子系统中的所有对象定义了一个简单接口，但是它不提供任何新功能。子系统本身不会意识到外观的存在。子系统中的对象可以直接进行交流。
  - 中介者将系统中组件的沟通行为中心化。各组件只知道中介者对象，无法直接相互交流。

- **中介者**和**观察者**之间的区别往往很难记住。在大部分情况下，你可以使用其中一种模式，而有时可以同时使用。让我们来看看如何做到这一点。

  中介者的主要目标是消除一系列系统组件之间的相互依赖。这些组件将依赖于同一个中介者对象。观察者的目标是在对象之间建立动态的单向连接，使得部分对象可作为其他对象的附属发挥作用。

  有一种流行的中介者模式实现方式依赖于观察者。中介者对象担当发布者的角色，其他组件则作为订阅者，可以订阅中介者的事件或取消订阅。当中介者以这种方式实现时，它可能看上去与观察者非常相似。

  当你感到疑惑时，记住可以采用其他方式来实现中介者。例如，你可永久性地将所有组件链接到同一个中介者对象。这种实现方式和观察者并不相同，但这仍是一种中介者模式。

  假设有一个程序，其所有的组件都变成了发布者，它们之间可以相互建立动态连接。这样程序中就没有中心化的中介者对象，而只有一些分布式的观察者。


# 备忘录

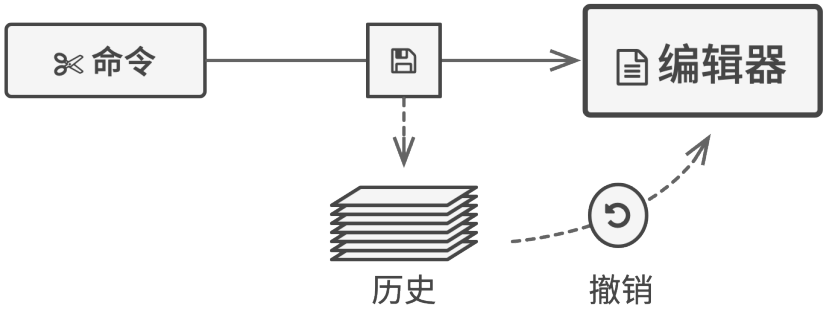
亦称：快照、Snapshot、Memento

**备忘录**是一种行为设计模式，允许在不暴露对象实现细节的情况下保存和恢复对象之前的状态。

# 问题

假如你正在开发一款文字编辑器应用程序。除了简单的文字编辑功能外，编辑器中还要有设置文本格式和插入内嵌图片等功能。

后来，你决定让用户能撤销施加在文本上的任何操作。这项功能在过去几年里变得十分普遍，因此用户期待任何程序都有这项功能。你选择采用直接的方式来实现该功能：程序在执行任何操作前会记录所有的对象状态，并将其保存下来。当用户此后需要撤销某个操作时，程序将从历史记录中获取最近的快照，然后使用它来恢复所有对象的状态。

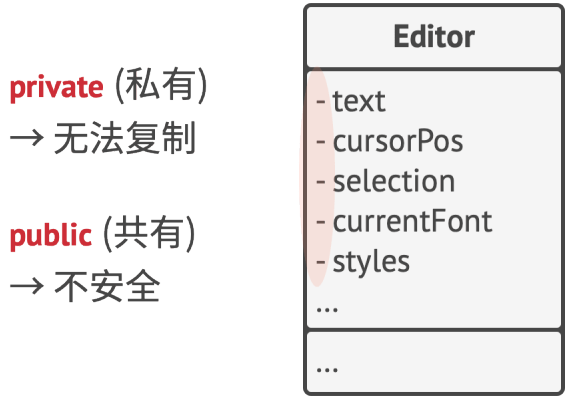



程序在执行操作前保存所有对象的状态快照，稍后可通过快照将对象恢复到之前的状态。

让我们来思考一下这些状态快照。首先，到底该如何生成一个快照呢？很可能你会需要遍历对象的所有成员变量并将其数值复制保存。但只有当对象对其内容没有严格访问权限限制的情况下，你才能使用该方式。不过很遗憾，绝大部分对

象会使用私有成员变量来存储重要数据，这样别人就无法轻易查看其中的内容。

现在我们暂时忽略这个问题，假设对象都像嬉皮士一样：喜欢开放式的关系并会公开其所有状态。尽管这种方式能够解决当前问题，让你可随时生成对象的状态快照，但这种方式仍存在一些严重问题。未来你可能会添加或删除一些成员变量。这听上去很简单，但需要对负责复制受影响对象状态的类进行更改。




如何复制对象的私有状态？

还有更多问题。让我们来考虑编辑器（Editor）状态的实际“快照”，它需要包含哪些数据？至少必须包含实际的文本、光标坐标和当前滚动条位置等。你需要收集这些数据并将其放入特定容器中，才能生成快照。

你很可能会将大量的容器对象存储在历史记录列表中。这样一来，容器最终大概率会成为同一个类的对象。这个类中几乎没有任何方法，但有许多与编辑器状态一一对应的成员变量。为了让其他对象能保存或读取快照，你很可能需要将快照的成员变量设为公有。无论这些状态是否私有，其都将暴露一切编辑器状态。其他类会对快照类的每个小改动产生依赖，除非这些改动仅存在于私有成员变量或方法中，而不会影响外部类。

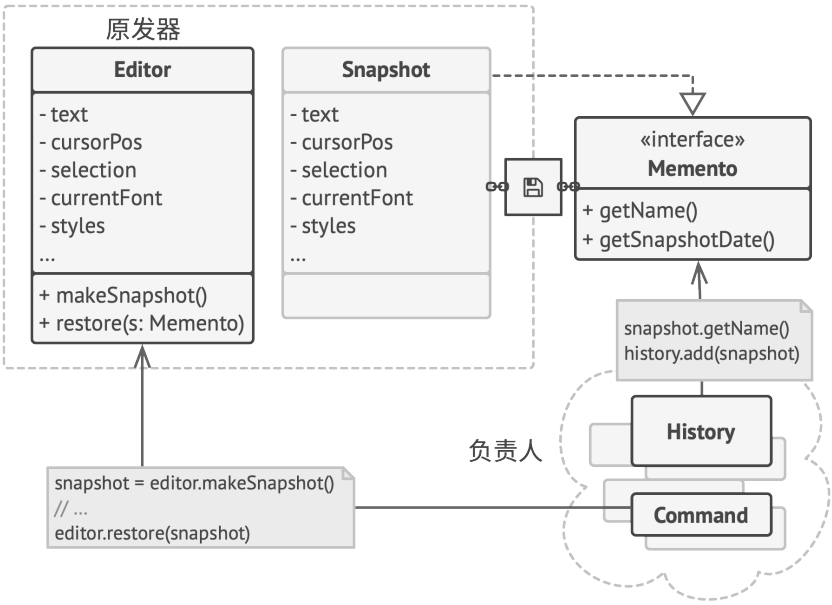
我们似乎走进了一条死胡同：要么会暴露类的所有内部细节而使其过于脆弱；要么会限制对其状态的访问权限而无法生成快照。那么，我们还有其他方式来实现“撤销”功能吗？

## 😊解决方案

我们刚才遇到的所有问题都是封装“破损”造成的。一些对象试图超出其职责范围的工作。由于在执行某些行为时需要获取数据，所以它们侵入了其他对象的私有空间，而不是让这些对象来完成实际的工作。

备忘录模式将创建状态快照（Snapshot）的工作委派给实际状态的拥有者原发器（Originator）对象。这样其他对象就不再需要从“外部”复制编辑器状态了，编辑器类拥有其状态的完全访问权，因此可以自行生成快照。

模式建议将对象状态的副本存储在一个名为**备忘录**（Memento）的特殊对象中。除了创建备忘录的对象外，任何对象都不能访问备忘录的内容。其他对象必须使用受限接口与备忘录进行交互，它们可以获取快照的元数据（创建时间和操作名称等），但不能获取快照中原始对象的状态。




原发器拥有对备忘录的完全访问权限，负责人则只能访问元数据。

这种限制策略允许你将备忘录保存在通常被称为**负责人**（Caretakers）的对象中。由于负责人仅通过受限接口与备忘录互动，故其无法修改存储在备忘录内部的状态。同时，原发器拥有对备忘录所有成员的访问权限，从而能随时恢复其以前的状态。

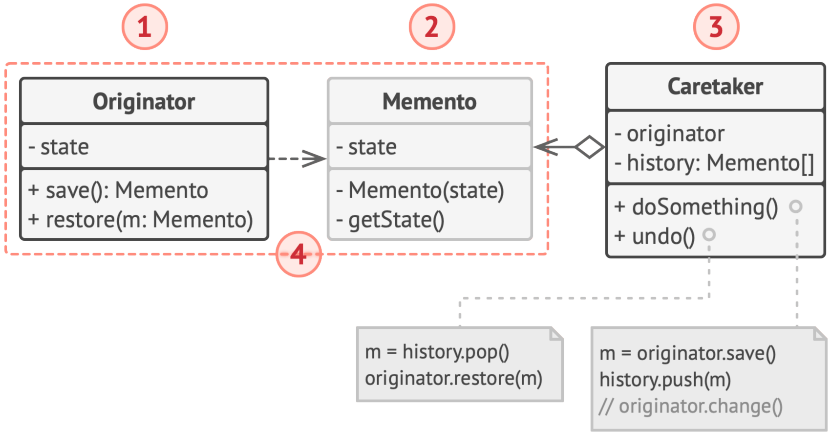
在文字编辑器的示例中，我们可以创建一个独立的历史（History）类作为负责人。编辑器每次执行操作前，存储在负责人中的备忘录栈都会生长。你甚至可以在应用的 UI 中渲染该栈，为用户显示之前的操作历史。

当用户触发撤销操作时，历史类将从栈中取回最近的备忘录，并将其传递给编辑器以请求进行回滚。由于编辑器拥有对备忘录的完全访问权限，因此它可以使用从备忘录中获取的数值来替换自身的状态。

# 结构

## 基于嵌套类的实现

该模式的经典实现方式依赖于许多流行编程语言（例如 C++、C# 和 Java）所支持的嵌套类。



1. **原发器**（Originator）类可以生成自身状态的快照，也可以在需要时通过快照恢复自身状态。

2. **备忘录**（Memento）是原发器状态快照的值对象（value object）。通常做法是将备忘录设为不可变的，并通过构造函数一次性传递数据。

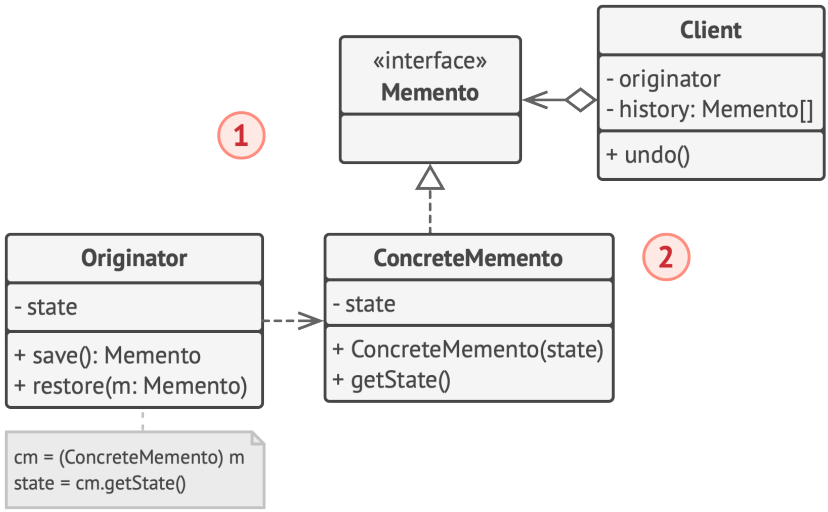
3. **负责人**（Caretaker）仅知道“何时”和“为何”捕捉原发器的状态，以及何时恢复状态。

   负责人通过保存备忘录栈来记录原发器的历史状态。当原发器需要回溯历史状态时，负责人将从栈中获取最顶部的备忘录，并将其传递给原发器的恢复（restoration）方法。

4. 在该实现方法中，备忘录类将被嵌套在原发器中。这样原发器就可访问备忘录的成员变量和方法，即使这些方法被声明为私有。另一方面，负责人对于备忘录的成员变量和方法的访问权限非常有限：它们只能在栈中保存备忘录，而不能修改其状态。

### 基于中间接口的实现

另外一种实现方法适用于不支持嵌套类的编程语言（没错，我说的就是 PHP）。

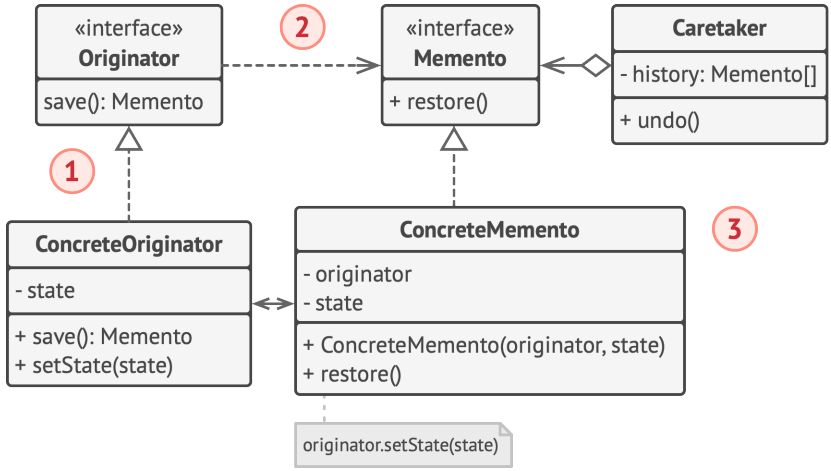



1. 在没有嵌套类的情况下，你可以规定负责人仅可通过明确声明的中间接口与备忘录互动，该接口仅声明与备忘录元数据相关的方法，限制其对备忘录成员变量的直接访问权限。

2. 另一方面，原发器可以直接与备忘录对象进行交互，访问备忘录类中声明的成员变量和方法。这种方式的缺点在于你需要将备忘录的所有成员变量声明为公有。

### 封装更加严格的实现

如果你不想让其他类有任何机会通过备忘录来访问原发器的状态，那么还有另一种可用的实现方式。



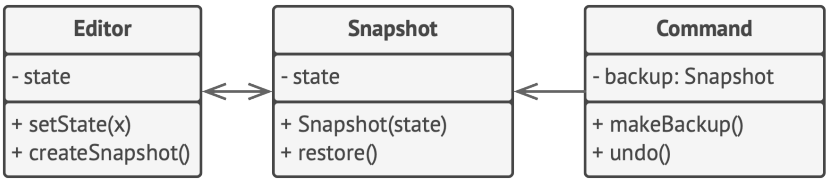

1. 这种实现方式允许存在多种不同类型的原发器和备忘录。每种原发器都和其相应的备忘录类进行交互。原发器和备忘录都不会将其状态暴露给其他类。

2. 负责人此时被明确禁止修改存储在备忘录中的状态。但负责人类将独立于原发器，因为此时恢复方法被定义在了备忘录类中。

3. 每个备忘录将与创建了自身的原发器连接。原发器会将自己及状态传递给备忘录的构造函数。由于这些类之间的紧密联系，只要原发器定义了合适的设置器（setter），备忘录就能恢复其状态。

# # 伪代码

本例结合使用了**命令**模式与备忘录模式，可保存复杂文字编辑器的状态快照，并能在需要时从快照中恢复之前的状态。




保存文字编辑器状态的快照。

命令（command）对象将作为负责人，它们会在执行与命令相关的操作前获取编辑器的备忘录。当用户试图撤销最近的命令时，编辑器可以使用保存在命令中的备忘录来将自身回滚到之前的状态。

备忘录类没有声明任何公有的成员变量、获取器（getter）和设置器，因此没有对象可以修改其内容。备忘录与创建自己的编辑器相连接，这使得备忘录能够通过编辑器对象的设置器传递数据，恢复与其相连接的编辑器的状态。由于备忘录与特定的编辑器对象相连接，程序可以使用中心化的撤销栈实现对多个独立编辑器窗口的支持。

```
1  // 原发器中包含了一些可能会随时间变化的重要数据。它还定义了在备忘录中保存
2  // 自身状态的方法，以及从备忘录中恢复状态的方法。
3  class Editor is
```

```
    private field text, curX, curY, selectionWidth

    method setText(text) is
      this.text = text

    method setCursor(x, y) is
      this.curX = curX
      this.curY = curY

    method setSelectionWidth(width) is
      this.selectionWidth = width

    // 在备忘录中保存当前的状态。
    method createSnapshot():Snapshot is
      // 备忘录是不可变的对象；因此原发器会将自身状态作为参数传递给备忘
      // 录的构造函数。
      return new Snapshot(this, text, curX, curY, selectionWidth)

// 备忘录类保存有编辑器的过往状态。
class Snapshot is
    private field editor: Editor
    private field text, curX, curY, selectionWidth

    constructor Snapshot(editor, text, curX, curY, selectionWidth) is
      this.editor = editor
      this.text = text
      this.curX = curX
      this.curY = curY
      this.selectionWidth = selectionWidth

    // 在某一时刻，编辑器之前的状态可以使用备忘录对象来恢复。
    method restore() is
```

```
    editor.setText(text)
    editor.setCursor(curX, curY)
    editor.setSelectionWidth(selectionWidth)

// 命令对象可作为负责人。在这种情况下，命令会在修改原发器状态之前获取一个
// 备忘录。当需要撤销时，它会从备忘录中恢复原发器的状态。
class Command is
  private field backup: Snapshot

  method makeBackup() is
    backup = editor.createSnapshot()

  method undo() is
    if (backup != null)
      backup.restore()
  // ...
```

## 适合应用场景

**当你需要创建对象状态快照来恢复其之前的状态时，可以使用备忘录模式。**

备忘录模式允许你复制对象中的全部状态（包括私有成员变量），并将其独立于对象进行保存。尽管大部分人因为“撤销”这个用例才记得该模式，但其实它在处理事务（比如需要在出现错误时回滚一个操作）的过程中也必不可少。

**当直接访问对象的成员变量、获取器或设置器将导致封装被突破时，可以使用该模式。**

备忘录让对象自行负责创建其状态的快照。任何其他对象都不能读取快照，这有效地保障了数据的安全性。

# 实现方式

1. 确定担任原发器角色的类。重要的是明确程序使用的一个原发器中心对象，还是多个较小的对象。

2. 创建备忘录类。逐一声明对应每个原发器成员变量的备忘录成员变量。

3. 将备忘录类设为不可变。备忘录只能通过构造函数一次性接收数据。该类中不能包含设置器。

4. 如果你所使用的编程语言支持嵌套类，则可将备忘录嵌套在原发器中；如果不支持，那么你可从备忘录类中抽取一个空接口，然后让其他所有对象通过接口来引用备忘录。你可在该接口中添加一些元数据操作，但不能暴露原发器的状态。

5. 在原发器中添加一个创建备忘录的方法。原发器必须通过备忘录构造函数的一个或多个实际参数来将自身状态传递给备忘录。

该方法返回结果的类型必须是你在上一步中抽取的接口（如果你已经抽取了）。实际上，创建备忘录的方法必须直接与备忘录类进行交互。

6. 在原发器类中添加一个用于恢复自身状态的方法。该方法接受备忘录对象作为参数。如果你在之前的步骤中抽取了接口，那么可将接口作为参数的类型。在这种情况下，你需要将输入对象强制转换为备忘录，因为原发器需要拥有对该对象的完全访问权限。

7. 无论负责人是命令对象、历史记录或其他完全不同的东西，它都必须要知道何时向原发器请求新的备忘录、如何存储备忘录以及何时使用特定备忘录来对原发器进行恢复。

8. 负责人与原发器之间的连接可以移动到备忘录类中。在本例中，每个备忘录都必须与创建自己的原发器相连接。恢复方法也可以移动到备忘录类中，但只有当备忘录类嵌套在原发器中，或者原发器类提供了足够多的设置器并可对其状态进行重写时，这种方式才能实现。

## ⚖ 优缺点

- ✓ 你可以在不破坏对象封装情况的前提下创建对象状态快照。
- ✓ 你可以通过让负责人维护原发器状态历史记录来简化原发器代码。

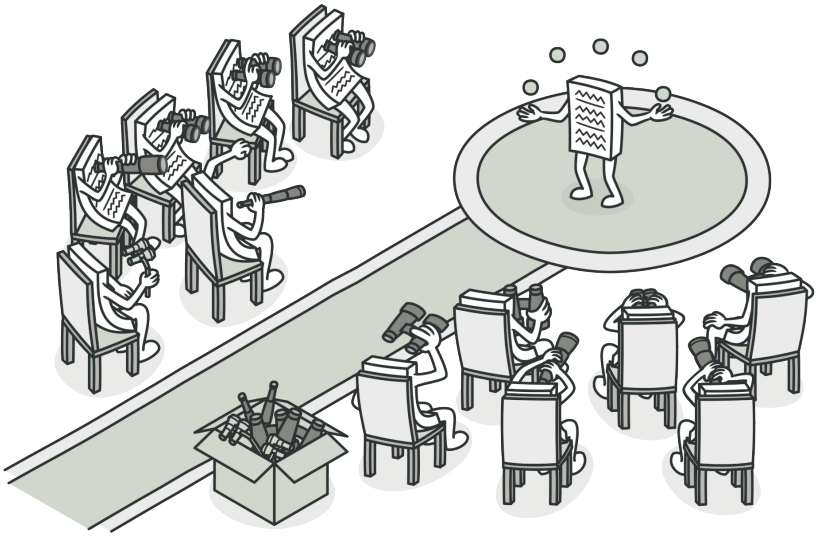
- ✕ 如果客户端过于频繁地创建备忘录，程序将消耗大量内存。
- ✕ 负责人必须完整跟踪原发器的生命周期，这样才能销毁弃用的备忘录。
- ✕ 绝大部分动态编程语言（例如 PHP、 Python 和 JavaScript）不能确保备忘录中的状态不被修改。

## ⇄ 与其他模式的关系

- 你可以同时使用**命令**和**备忘录**来实现“撤销”。在这种情况下，命令用于对目标对象执行各种不同的操作，备忘录用来保存一条命令执行前该对象的状态。

- 你可以同时使用**备忘录**和**迭代器**来获取当前迭代器的状态，并且在需要的时候进行回滚。

- 有时候**原型**可以作为**备忘录**的一个简化版本，其条件是你需要在历史记录中存储的对象的状态比较简单，不需要链接其他外部资源，或者链接可以方便地重建。



# 观察者

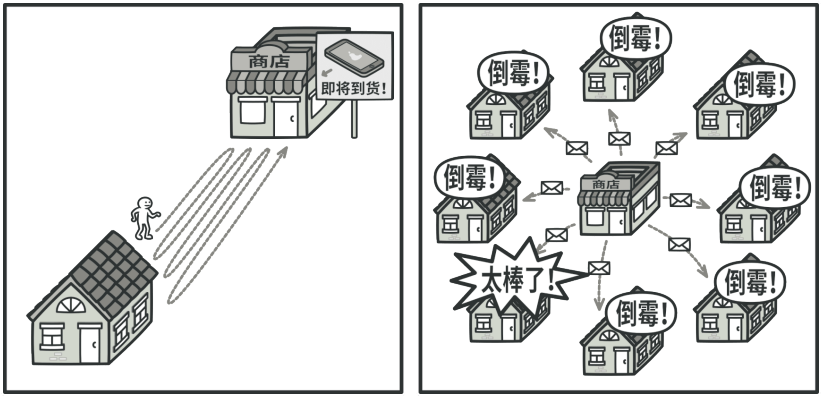
亦称：事件订阅者、监听者、Event-Subscriber、Listener、Observer

**观察者**是一种行为设计模式，
允许你定义一种订阅机制，
可在对象事件发生时通知多
个“观察”该对象的其他对象。

# ☹ 问题

假如你有两种类型的对象：`顾客`和`商店`。顾客对某个特定品牌的产品非常感兴趣（例如最新型号的 iPhone 手机），而该产品很快将会在商店里出售。

顾客可以每天来商店看看产品是否到货。但如果商品尚未到货时，绝大多数来到商店的顾客都会空手而归。




前往商店和发送垃圾邮件

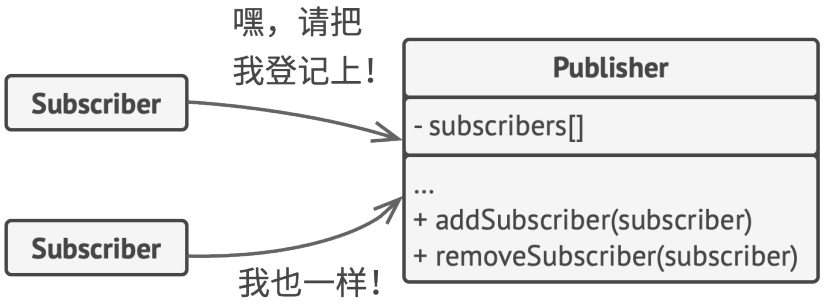
另一方面，每次新产品到货时，商店可以向所有顾客发送邮件（可能会被视为垃圾邮件）。这样，部分顾客就无需反复前往商店了，但也可能会惹恼对新产品没有兴趣的其他顾客。

我们似乎遇到了一个矛盾：要么让顾客浪费时间检查产品是否到货，要么让商店浪费资源去通知没有需求的顾客。

# 😊解决方案

拥有一些值得关注的状态的对象通常被称为**目标**，由于它要将自身的状态改变通知给其他对象，我们也将其称为**发布者**（publisher）。所有希望关注发布者状态变化的其他对象被称为**订阅者**（subscribers）。

观察者模式建议你为发布者类添加订阅机制，让每个对象都能订阅或取消订阅发布者事件流。不要害怕！这并不像听上去那么复杂。实际上，该机制包括 1）一个用于存储订阅者对象引用的列表成员变量；2）几个用于添加或删除该列表中订阅者的公有方法。



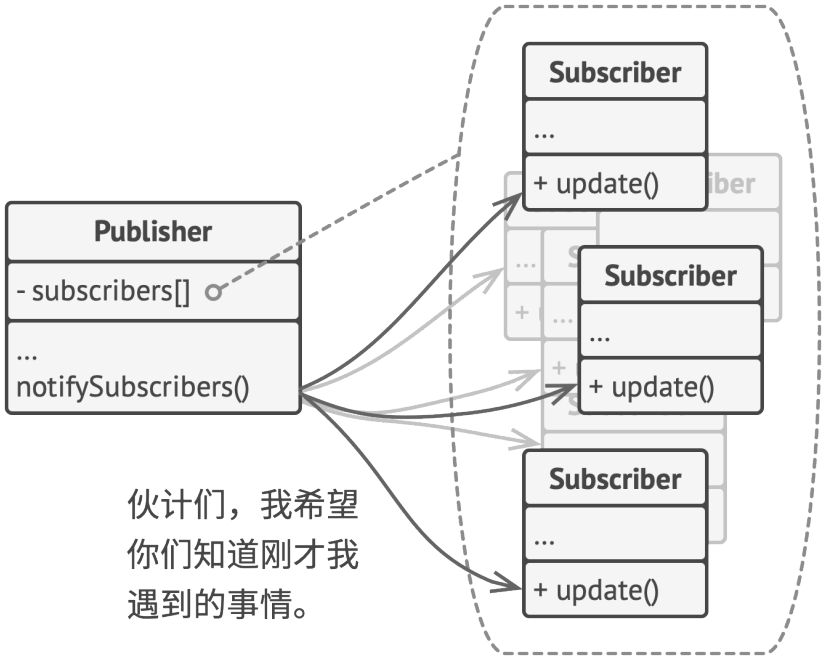

订阅机制允许对象订阅事件通知。

现在，无论何时发生了重要的发布者事件，它都要遍历订阅者并调用其对象的特定通知方法。

实际应用中可能会有十几个不同的订阅者类跟踪着同一个发布者类的事件，你不会希望发布者与所有这些类相耦合的。

此外如果他人会使用发布者类，那么你甚至可能会对其中的一些类一无所知。

因此，所有订阅者都必须实现同样的接口，发布者仅通过该接口与订阅者交互。接口中必须声明通知方法及其参数，这样发布者在发出通知时还能传递一些上下文数据。

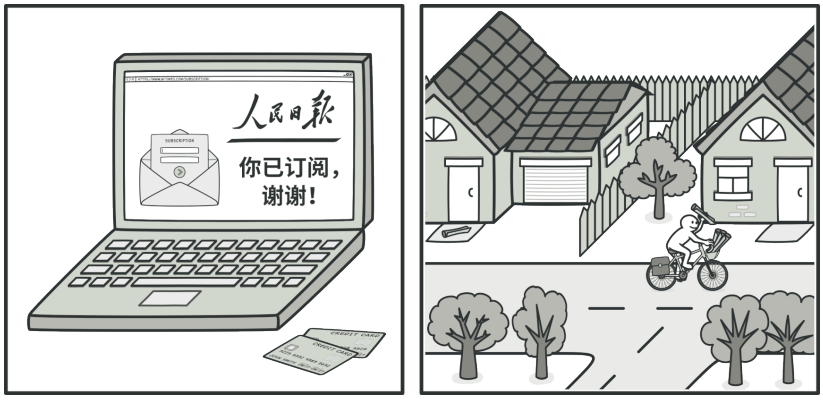



发布者调用订阅者对象中的特定通知方法来通知订阅者。

如果你的应用中有多个不同类型的发布者，且希望订阅者可兼容所有发布者，那么你甚至可以进一步让所有订阅者遵循同样的接口。该接口仅需描述几个订阅方法即可。这样订阅

者就能在不与具体发布者类耦合的情况下通过接口观察发布者的状态。

## 真实世界类比

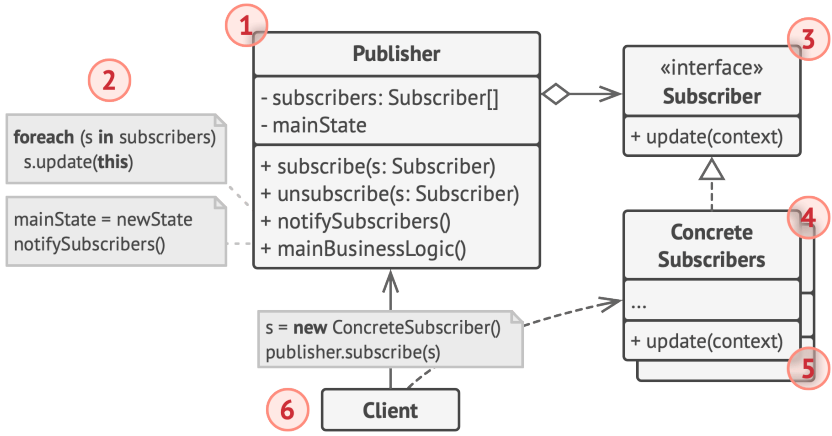



杂志和报纸订阅。

如果你订阅了一份杂志或报纸，那就不需要再去报摊查询新出版的刊物了。出版社（即应用中的“发布者”）会在刊物出版后（甚至提前）直接将最新一期寄送至你的邮箱中。

出版社负责维护订阅者列表，了解订阅者对哪些刊物感兴趣。当订阅者希望出版社停止寄送新一期的杂志时，他们可随时从该列表中退出。

# 结构




1. **发布者**（Publisher）会向其他对象发送值得关注的事件。事件会在发布者自身状态改变或执行特定行为后发生。发布者中包含一个允许新订阅者加入和当前订阅者离开列表的订阅构架。

2. 当新事件发生时，发送者会遍历订阅列表并调用每个订阅者对象的通知方法。该方法是在订阅者接口中声明的。

3. **订阅者**（Subscriber）接口声明了通知接口。在绝大多数情况下，该接口仅包含一个 update 更新 方法。该方法可以拥有多个参数，使发布者能在更新时传递事件的详细信息。

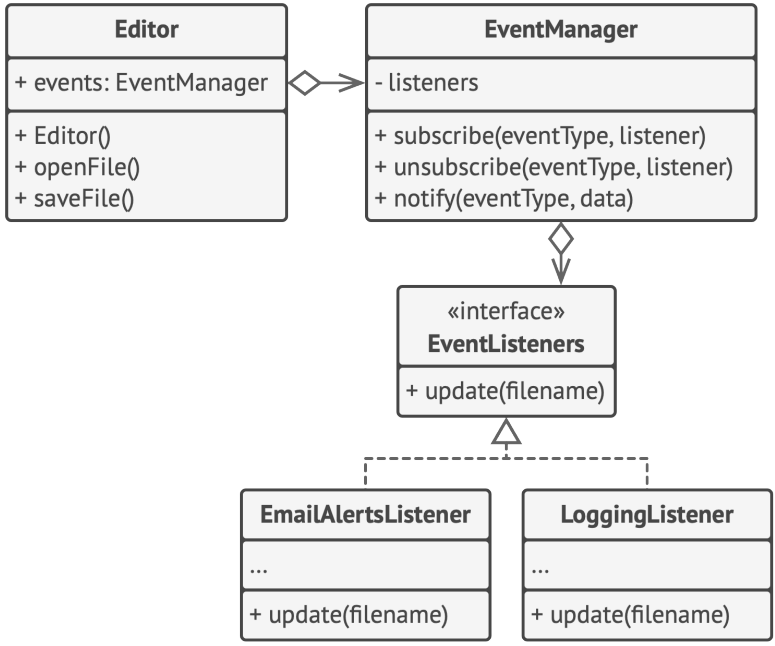
4. **具体订阅者**（Concrete Subscribers）可以执行一些操作来回应发布者的通知。所有具体订阅者类都实现了同样的接口，因此发布者不需要与具体类相耦合。

5. 订阅者通常需要一些上下文信息来正确地处理更新。因此，发布者通常会将一些上下文数据作为通知方法的参数进行传递。发布者也可将自身作为参数进行传递，使订阅者直接获取所需的数据。

6. **客户端**（Client）会分别创建发布者和订阅者对象，然后为订阅者注册发布者更新。

# 伪代码

在本例中，**观察者**模式允许文本编辑器对象将自身的状态改变通知给其他服务对象。




将对象中发生的事件通知给其他对象。

订阅者列表是动态生成的：对象可在运行时根据程序需要开始或停止监听通知。

在本实现中，编辑器类自身并不维护订阅列表。它将工作委派给专门从事此工作的一个特殊帮手对象。你还可将该对象升级为中心化的事件分发器，允许任何对象成为发布者。

只要发布者通过同样的接口与所有订阅者进行交互，那么在程序中新增订阅者时就无需修改已有发布者类的代码。

```
// 发布者基类包含订阅管理代码和通知方法。
class EventManager is
  private field listeners: hash map of event types and listeners

  method subscribe(eventType, listener) is
    listeners.add(eventType, listener)

  method unsubscribe(eventType, listener) is
    listeners.remove(eventType, listener)

  method notify(eventType, data) is
    foreach (listener in listeners.of(eventType)) do
      listener.update(data)

// 具体发布者包含一些订阅者感兴趣的实际业务逻辑。我们可以从发布者基类中扩
// 展出该类，但在实际情况下并不总能做到，因为具体发布者可能已经是子类了。
// 在这种情况下，你可用组合来修补订阅逻辑，就像我们在这里做的一样。
class Editor is
  public field events: EventManager
```

```
    private field file: File

    constructor Editor() is
      events = new EventManager()

    // 业务逻辑的方法可将变化通知给订阅者。
    method openFile(path) is
      this.file = new File(path)
      events.notify("open", file.name)

    method saveFile() is
      file.write()
      events.notify("save", file.name)

    // ...


// 这里是订阅者接口。如果你的编程语言支持函数类型，则可用一组函数来代替整
// 个订阅者的层次结构。
interface EventListener is
    method update(filename)

// 具体订阅者会对其注册的发布者所发出的更新消息做出响应。
class LoggingListener implements EventListener is
    private field log: File
    private field message

    constructor LoggingListener(log_filename, message) is
      this.log = new File(log_filename)
      this.message = message

    method update(filename) is
```

```
    log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
  private field email: string

  constructor EmailAlertsListener(email, message) is
    this.email = email
    this.message = message

  method update(filename) is
    system.email(email, replace('%s',filename,message))


// 应用程序可在运行时配置发布者和订阅者。
class Application is
  method config() is
    editor = new Editor()

    logger = new LoggingListener(
      "/path/to/log.txt",
      "有人打开了文件：%s");
    editor.events.subscribe("open", logger)

    emailAlerts = new EmailAlertsListener(
      "admin@example.com",
      "有人更改了文件：%s")
    editor.events.subscribe("save", emailAlerts)
```

# 适合应用场景

**当一个对象状态的改变需要改变其他对象，或实际对象是事先未知的或动态变化的时，可使用观察者模式。**

当你使用图形用户界面类时通常会遇到一个问题。比如，你创建了自定义按钮类并允许客户端在按钮中注入自定义代码，这样当用户按下按钮时就会触发这些代码。

观察者模式允许任何实现了订阅者接口的对象订阅发布者对象的事件通知。你可在按钮中添加订阅机制，允许客户端通过自定义订阅类注入自定义代码。

**当应用中的一些对象必须观察其他对象时，可使用该模式。但仅能在有限时间内或特定情况下使用。**

订阅列表是动态的，因此订阅者可随时加入或离开该列表。

# 实现方式

1. 仔细检查你的业务逻辑，试着将其拆分为两个部分：独立于其他代码的核心功能将作为发布者；其他代码则将转化为一组订阅类。

2. 声明订阅者接口。该接口至少应声明一个 `update` 方法。

3. 声明发布者接口并定义一些接口来在列表中添加和删除订阅对象。记住发布者必须仅通过订阅者接口与它们进行交互。

4. 确定存放实际订阅列表的位置并实现订阅方法。通常所有类型的发布者代码看上去都一样，因此将列表放置在直接扩展自发布者接口的抽象类中是显而易见的。具体发布者会扩展该类从而继承所有的订阅行为。

   但是，如果你需要在现有的类层次结构中应用该模式，则可以考虑使用组合的方式：将订阅逻辑放入一个独立的对象，然后让所有实际订阅者使用该对象。

5. 创建具体发布者类。每次发布者发生了重要事件时都必须通知所有的订阅者。

6. 在具体订阅者类中实现通知更新的方法。绝大部分订阅者需要一些与事件相关的上下文数据。这些数据可作为通知方法的参数来传递。

   但还有另一种选择。订阅者接收到通知后直接从通知中获取所有数据。在这种情况下，发布者必须通过更新方法将自身传递出去。另一种不太灵活的方式是通过构造函数将发布者与订阅者永久性地连接起来。

7. 客户端必须生成所需的全部订阅者，并在相应的发布者处完成注册工作。

# ⚖ 优缺点

✓ 开闭原则。你无需修改发布者代码就能引入新的订阅者类（如果是发布者接口则可轻松引入发布者类）。

✓ 你可以在运行时建立对象之间的联系。

✗ 订阅者的通知顺序是随机的。

# ⇄ 与其他模式的关系

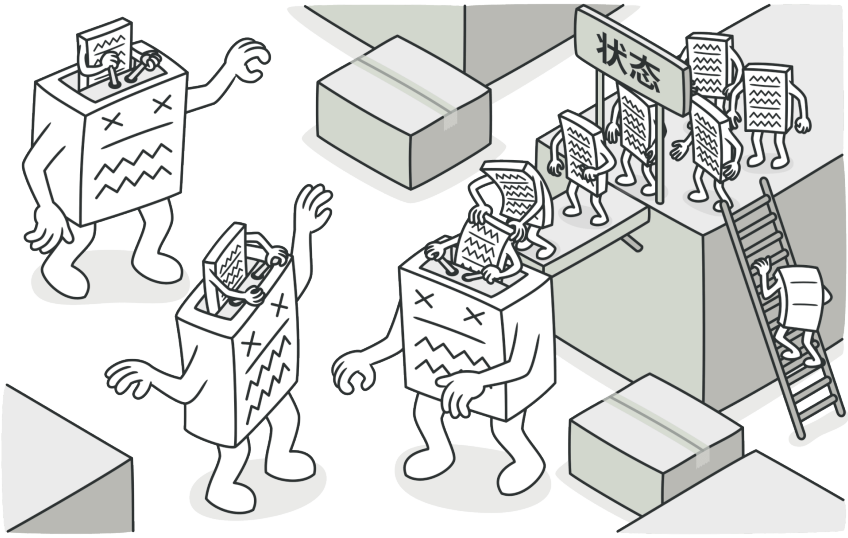
- **责任链**、**命令**、**中介者**和**观察者**用于处理请求发送者和接收者之间的不同连接方式：
  - 责任链按照顺序将请求动态传递给一系列的潜在接收者，直至其中一名接收者对请求进行处理。
  - 命令在发送者和请求者之间建立单向连接。
  - 中介者清除了发送者和请求者之间的直接连接，强制它们通过一个中介对象进行间接沟通。
  - 观察者允许接收者动态地订阅或取消接收请求。
- **中介者**和**观察者**之间的区别往往很难记住。在大部分情况下，你可以使用其中一种模式，而有时可以同时使用。让我们来看看如何做到这一点。

中介者的主要目标是消除一系列系统组件之间的相互依赖。这些组件将依赖于同一个中介者对象。观察者的目标是在对象之间建立动态的单向连接，使得部分对象可作为其他对象的附属发挥作用。

有一种流行的中介者模式实现方式依赖于观察者。中介者对象担当发布者的角色，其他组件则作为订阅者，可以订阅中介者的事件或取消订阅。当中介者以这种方式实现时，它可能看上去与观察者非常相似。

当你感到疑惑时，记住可以采用其他方式来实现中介者。例如，你可永久性地将所有组件链接到同一个中介者对象。这种实现方式和观察者并不相同，但这仍是一种中介者模式。

假设有一个程序，其所有的组件都变成了发布者，它们之间可以相互建立动态连接。这样程序中就没有中心化的中介者对象，而只有一些分布式的观察者。



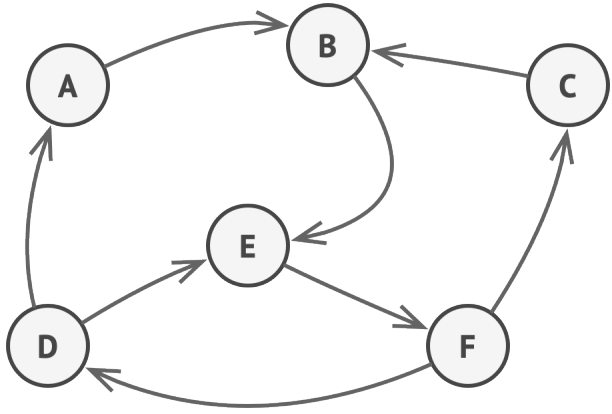

# 状态

亦称：State

**状态**是一种行为设计模式，
让你能在一个对象的内部
状态变化时改变其行为，
使其看上去就像改变了自
身所属的类一样。

# ☹ 问题

状态模式与**有限状态机**的概念紧密相关。



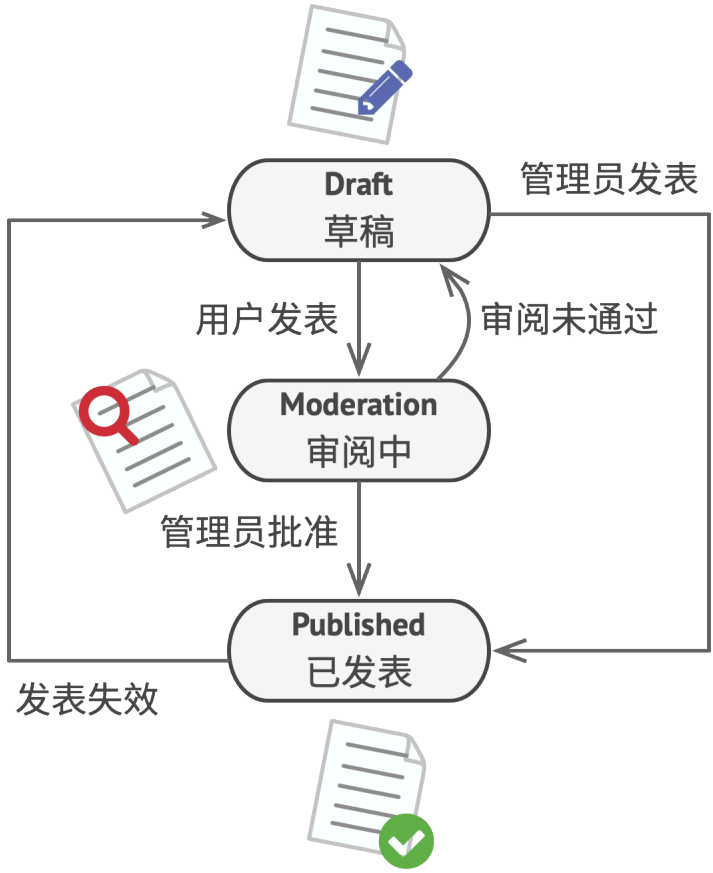

有限状态机。

其主要思想是程序在任意时刻仅可处于几种有限的状态中。在任何一个特定状态中，程序的行为都不相同，且可瞬间从一个状态切换到另一个状态。不过，根据当前状态，程序可能会切换到另外一种状态，也可能会保持当前状态不变。这些数量有限且预先定义的状态切换规则被称为转移。

你还可将该方法应用在对象上。假如你有一个 文档 Document 类。文档可能会处于 草稿 Draft、审阅中 Moderation 和 已发布 Published 三种状态中的一种。文档的 publish 发布 方法在不同状态下的行为略有不同：

- 处于 草稿 状态时，它会将文档转移到审阅中状态。

- 处于 `审阅中` 状态时，如果当前用户是管理员，它会公开发布文档。
- 处于 `已发布` 状态时，它不会进行任何操作。




文档对象的全部状态和转移。

状态机通常由众多条件运算符（`if` 或 `switch`）实现，可根据对象的当前状态选择相应的行为。“状态”通常只是对象中的一组成员变量值。即使你之前从未听说过有限状态机，

你也很可能已经实现过状态模式。下面的代码应该能帮助你回忆起来。

```
class Document is
  field state: string
  // ...
  method publish() is
    switch (state)
      "draft":
        state = "moderation"
        break
      "moderation":
        if (currentUser.role == 'admin')
          state = "published"
        break
      "published":
        // 什么也不做。
        break
  // ...
```

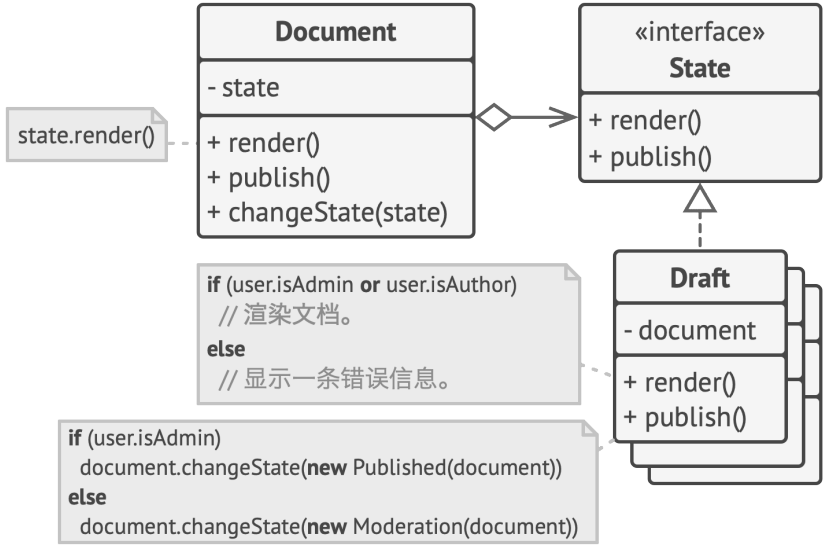
当我们逐步在 文档 类中添加更多状态和依赖于状态的行为后，基于条件语句的状态机就会暴露其最大的弱点。为了能根据当前状态选择完成相应行为的方法，绝大部分方法中会包含复杂的条件语句。修改其转换逻辑可能会涉及到修改所有方法中的状态条件语句，导致代码的维护工作非常艰难。

这个问题会随着项目进行变得越发严重。我们很难在设计阶段预测到所有可能的状态和转换。随着时间推移，最初仅包含有限条件语句的简洁状态机可能会变成臃肿的一团乱麻。

## 😊解决方案

状态模式建议为对象的所有可能状态新建一个类，然后将所有状态的对应行为抽取到这些类中。

原始对象被称为**上下文**（context），它并不会自行实现所有行为，而是会保存一个指向表示当前状态的状态对象的引用，且将所有与状态相关的工作委派给该对象。




文档将工作委派给一个状态对象。

如需将上下文转换为另外一种状态，则需将当前活动的状态对象替换为另外一个代表新状态的对象。采用这种方式是有前提的：所有状态类都必须遵循同样的接口，而且上下文必须仅通过接口与这些对象进行交互。

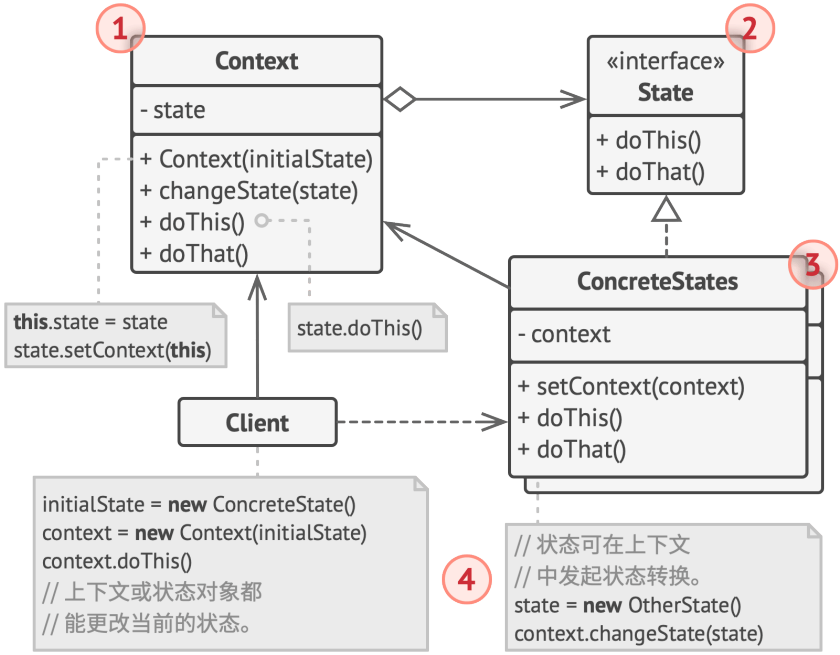
这个结构可能看上去与**策略**模式相似，但有一个关键性的不同——在状态模式中，特定状态知道其他所有状态的存在，且能触发从一个状态到另一个状态的转换；策略则几乎完全不知道其他策略的存在。

## 真实世界类比

智能手机的按键和开关会根据设备当前状态完成不同行为：

- 当手机处于解锁状态时，按下按键将执行各种功能。
- 当手机处于锁定状态时，按下任何按键都将解锁屏幕。
- 当手机电量不足时，按下任何按键都将显示充电页面。

# 结构




1. **上下文**（Context）保存了对于一个具体状态对象的引用，并会将所有与该状态相关的工作委派给它。上下文通过状态接口与状态对象交互，且会提供一个设置器用于传递新的状态对象。

2. **状态**（State）接口会声明特定于状态的方法。这些方法应能被其他所有具体状态所理解，因为你不希望某些状态所拥有的方法永远不会被调用。

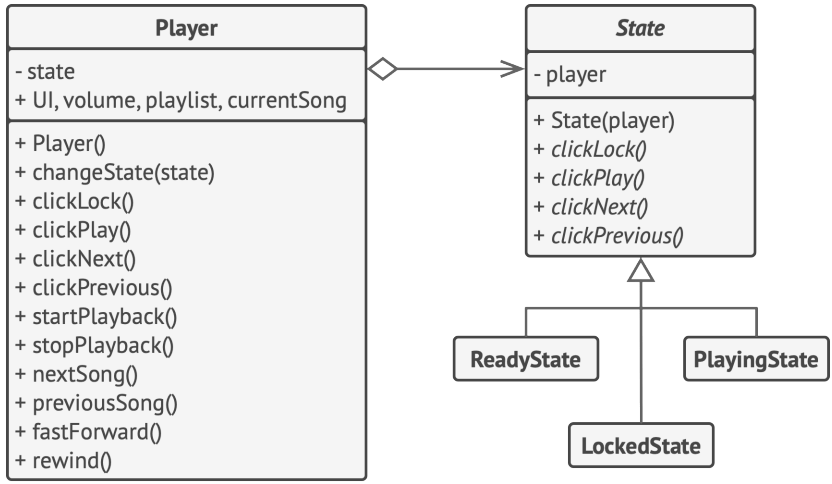
3. **具体状态**（Concrete States）会自行实现特定于状态的方法。为了避免多个状态中包含相似代码，你可以提供一个封装有部分通用行为的中间抽象类。

   状态对象可存储对于上下文对象的反向引用。状态可以通过该引用从上下文处获取所需信息，并且能触发状态转移。

4. 上下文和具体状态都可以设置上下文的下个状态，并可通过替换连接到上下文的状态对象来完成实际的状态转换。

# 伪代码

在本例中，**状态**模式将根据当前回放状态，让媒体播放器中的相同控件完成不同的行为。




使用状态对象更改对象行为的示例。

播放器的主要对象总是会连接到一个负责播放器绝大部分工作的状态对象中。部分操作会更换播放器当前的状态对象，以此改变播放器对于用户互动所作出的反应。

```
// 音频播放器（AudioPlayer）类即为上下文。它还会维护指向状态类实例的引用，
// 该状态类则用于表示音频播放器当前的状态。
class AudioPlayer is
  field state: State
  field UI, volume, playlist, currentSong

  constructor AudioPlayer() is
    this.state = new ReadyState(this)

    // 上下文会将处理用户输入的工作委派给状态对象。由于每个状态都以不
    // 同的方式处理输入，其结果自然将依赖于当前所处的状态。
    UI = new UserInterface()
    UI.lockButton.onClick(this.clickLock)
    UI.playButton.onClick(this.clickPlay)
    UI.nextButton.onClick(this.clickNext)
    UI.prevButton.onClick(this.clickPrevious)

  // 其他对象必须能切换音频播放器当前所处的状态。
  method changeState(state: State) is
    this.state = state

  // UI 方法会将执行工作委派给当前状态。
  method clickLock() is
    state.clickLock()
  method clickPlay() is
    state.clickPlay()
  method clickNext() is
```

```
      state.clickNext()
    method clickPrevious() is
      state.clickPrevious()

    // 状态可调用上下文的一些服务方法。
    method startPlayback() is
      // ...
    method stopPlayback() is
      // ...
    method nextSong() is
      // ...
    method previousSong() is
      // ...
    method fastForward(time) is
      // ...
    method rewind(time) is
      // ...


// 所有具体状态类都必须实现状态基类声明的方法，并提供反向引用指向与状态相
// 关的上下文对象。状态可使用反向引用将上下文转换为另一个状态。
abstract class State is
    protected field player: AudioPlayer

    // 上下文将自身传递给状态构造函数。这可帮助状态在需要时获取一些有用的
    // 上下文数据。
    constructor State(player) is
      this.player = player

    abstract method clickLock()
    abstract method clickPlay()
    abstract method clickNext()
```

```
60    abstract method clickPrevious()
61
62
63  // 具体状态会实现与上下文状态相关的多种行为。
64  class LockedState extends State is
65
66    // 当你解锁一个锁定的播放器时，它可能处于两种状态之一。
67    method clickLock() is
68      if (player.playing)
69        player.changeState(new PlayingState(player))
70      else
71        player.changeState(new ReadyState(player))
72
73    method clickPlay() is
74      // 已锁定，什么也不做。
75
76    method clickNext() is
77      // 已锁定，什么也不做。
78
79    method clickPrevious() is
80      // 已锁定，什么也不做。
81
82
83  // 它们还可在上下文中触发状态转换。
84  class ReadyState extends State is
85    method clickLock() is
86      player.changeState(new LockedState(player))
87
88    method clickPlay() is
89      player.startPlayback()
90      player.changeState(new PlayingState(player))
91
```

```
    method clickNext() is
      player.nextSong()

    method clickPrevious() is
      player.previousSong()


class PlayingState extends State is
    method clickLock() is
      player.changeState(new LockedState(player))

    method clickPlay() is
      player.stopPlayback()
      player.changeState(new ReadyState(player))

    method clickNext() is
      if (event.doubleclick)
        player.nextSong()
      else
        player.fastForward(5)

    method clickPrevious() is
      if (event.doubleclick)
        player.previous()
      else
        player.rewind(5)
```

# 适合应用场景

**如果对象需要根据自身当前状态进行不同行为，同时状态的数量非常多且与状态相关的代码会频繁变更的话，可使用状态模式。**

模式建议你将所有特定于状态的代码抽取到一组独立的类中。这样一来，你可以在独立于其他状态的情况下添加新状态或修改已有状态，从而减少维护成本。

**如果某个类需要根据成员变量的当前值改变自身行为，从而需要使用大量的条件语句时，可使用该模式。**

状态模式会将这些条件语句的分支抽取到相应状态类的方法中。同时，你还可以清除主要类中与特定状态相关的临时成员变量和帮手方法代码。

**当相似状态和基于条件的状态机转换中存在许多重复代码时，可使用状态模式。**

状态模式让你能够生成状态类层次结构，通过将公用代码抽取到抽象基类中来减少重复。

# 实现方式

1. 确定哪些类是上下文。它可能是包含依赖于状态的代码的已有类；如果特定于状态的代码分散在多个类中，那么它可能是一个新的类。

2. 声明状态接口。虽然你可能会需要完全复制上下文中声明的所有方法，但最好是仅把关注点放在那些可能包含特定于状态的行为的方法上。

3. 为每个实际状态创建一个继承于状态接口的类。然后检查上下文中的方法并将与特定状态相关的所有代码抽取到新建的类中。

   在将代码移动到状态类的过程中，你可能会发现它依赖于上下文中的一些私有成员。你可以采用以下几种变通方式：

   - 将这些成员变量或方法设为公有。
   - 将需要抽取的上下文行为更改为上下文中的公有方法，然后在状态类中调用。这种方式简陋却便捷，你可以稍后再对其进行修补。
   - 将状态类嵌套在上下文类中。这种方式需要你所使用的编程语言支持嵌套类。

4. 在上下文类中添加一个状态接口类型的引用成员变量，以及一个用于修改该成员变量值的公有设置器。

5. 再次检查上下文中的方法，将空的条件语句替换为相应的状态对象方法。

6. 为切换上下文状态，你需要创建某个状态类实例并将其传递给上下文。你可以在上下文、各种状态或客户端中完成这项工作。无论在何处完成这项工作，该类都将依赖于其所实例化的具体类。

## ⚖ 优缺点

✓ 单一职责原则。将与特定状态相关的代码放在单独的类中。

✓ 开闭原则。无需修改已有状态类和上下文就能引入新状态。

✓ 通过消除臃肿的状态机条件语句简化上下文代码。

✗ 如果状态机只有很少的几个状态，或者很少发生改变，那么应用该模式可能会显得小题大作。

## ⇄ 与其他模式的关系

- **桥接**、**状态**和**策略**（在某种程度上包括**适配器**）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。

- **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。

# 策略

亦称：Strategy

**策略**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。

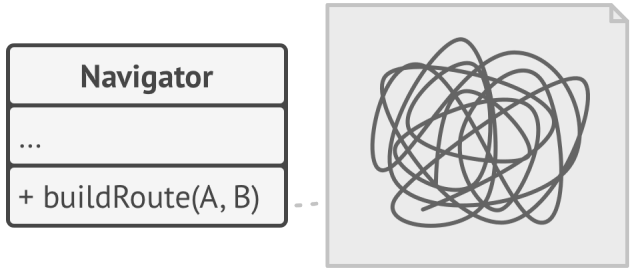
# ☹ 问题

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。

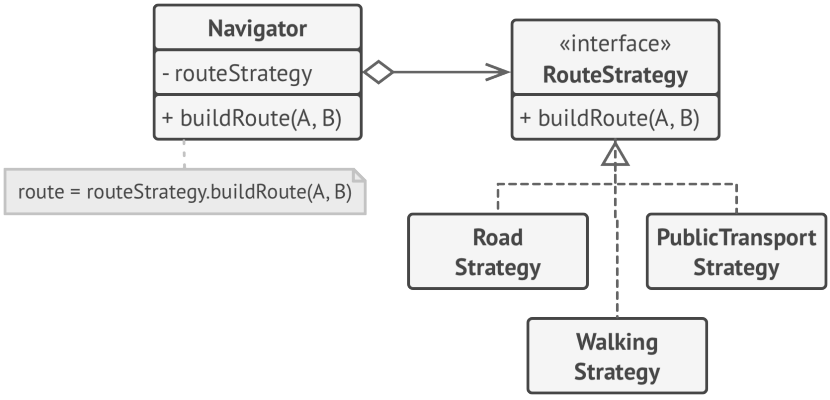
## 😊解决方案

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。

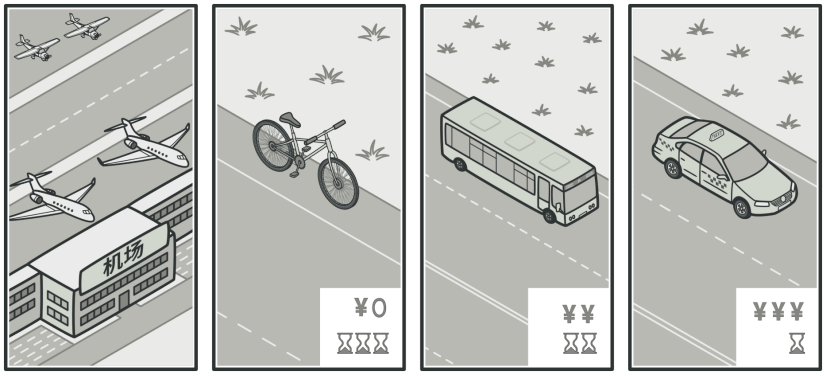



路线规划策略。

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。

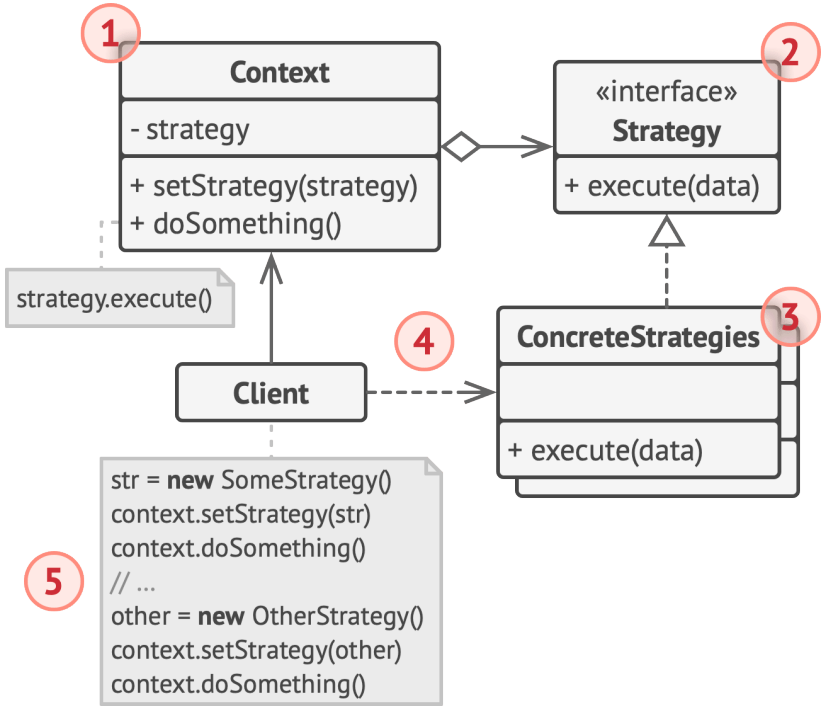
# 真实世界类比




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。

# 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。
```

# 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。

## ⚖ 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**、**状态**和**策略**（在某种程度上包括**适配器**）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**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。

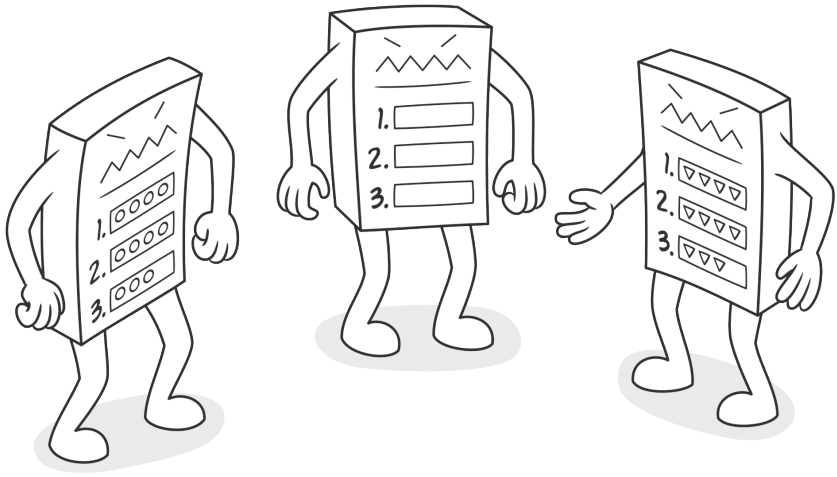
  - 你可以使用**命令**来将任何操作转换为对象。操作的参数将成为对象的成员变量。你可以通过转换来延迟操作的执行、将操作放入队列、保存历史命令或者向远程服务发送命令等。

  - 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**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**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。




# 模板方法

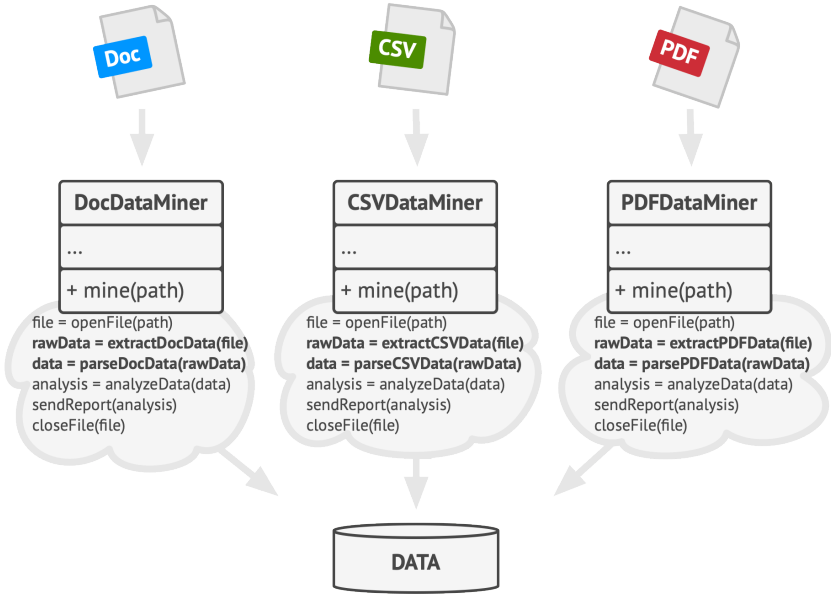
亦称：Template Method

**模板方法**是一种行为设计模式，它在超类中定义了一个算法的框架，允许子类在不修改结构的情况下重写算法的特定步骤。

# ☹ 问题

假如你正在开发一款分析公司文档的数据挖掘程序。用户需要向程序输入各种格式（PDF、DOC 或 CSV）的文档，程序则会试图从这些文件中抽取有意义的数据，并以统一的格式将其返回给用户。

该程序的首个版本仅支持 DOC 文件。在接下来的一个版本中，程序能够支持 CSV 文件。一个月后，你“教会”了程序从 PDF 文件中抽取数据。




数据挖掘类中包含许多重复代码。

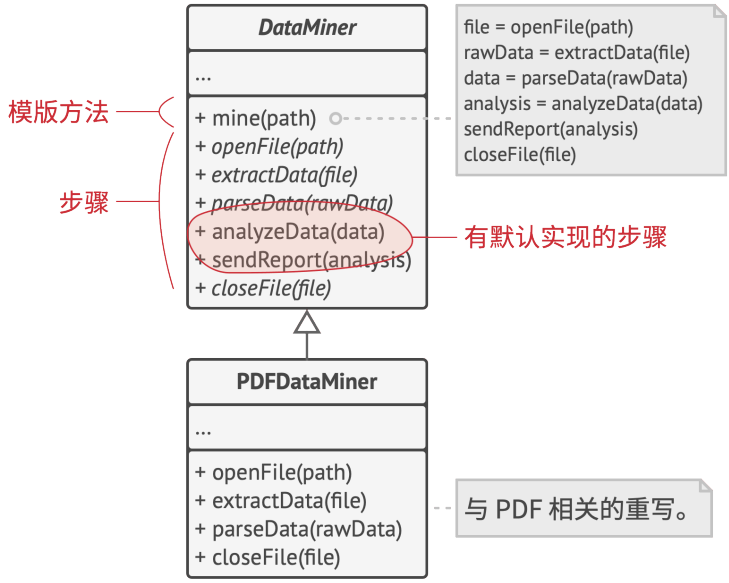
一段时间后，你发现这三个类中包含许多相似代码。尽管这些类处理不同数据格式的代码完全不同，但数据处理和分析的代码却几乎完全一样。如果能在保持算法结构完整的情况下去除重复代码，这难道不是一件很棒的事情吗？

还有另一个与使用这些类的客户端代码相关的问题：客户端代码中包含许多条件语句，以根据不同的处理对象类型选择合适的处理过程。如果所有处理数据的类都拥有相同的接口或基类，那么你就可以去除客户端代码中的条件语句，转而使用多态机制来在处理对象上调用函数。

## 😊解决方案

模板方法模式建议将算法分解为一系列步骤，然后将这些步骤改写为方法，最后在“模板方法”中依次调用这些方法。步骤可以是抽象的，也可以有一些默认的实现。为了能够使用算法，客户端需要自行提供子类并实现所有的抽象步骤。如有必要还需重写一些步骤（但这一步中不包括模板方法自身）。

让我们考虑如何在数据挖掘应用中实现上述方案。我们可为图中的三个解析算法创建一个基类，该类将定义调用了一系列不同文档处理步骤的模板方法。




模板方法将算法分解为步骤，并允许子类重写这些步骤，而非重写实际的模板方法。

首先，我们将所有步骤声明为 抽象 类型，强制要求子类自行实现这些方法。在我们的例子中，子类中已有所有必要的实现，因此我们只需调整这些方法的签名，使之与超类的方法匹配即可。

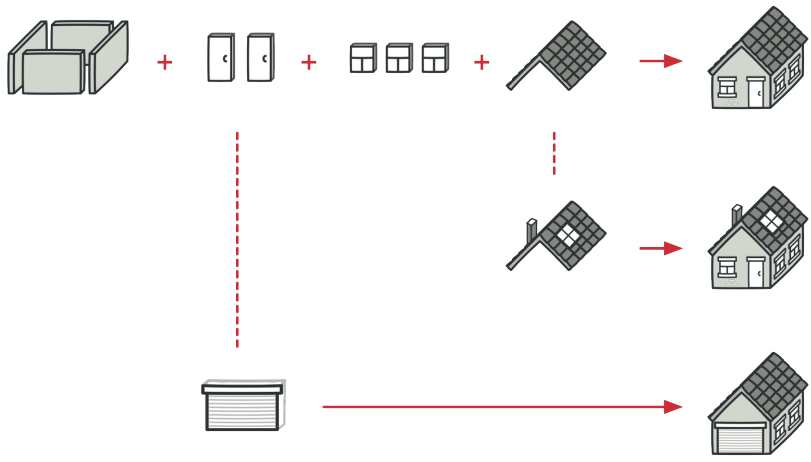
现在，让我们看看如何去除重复代码。对于不同的数据格式，打开和关闭文件以及抽取和解析数据的代码都不同，因此无需修改这些方法。但分析原始数据和生成报告等其他步骤的实现方式非常相似，因此可将其提取到基类中，以让子类共享这些代码。

正如你所看到的那样，我们有两种类型的步骤：

- **抽象步骤**必须由各个子类来实现
- **可选步骤**已有一些默认实现，但仍可在需要时进行重写

还有另一种名为**钩子**的步骤。**钩子**是内容为空的可选步骤。即使不重写钩子，模板方法也能工作。钩子通常放置在算法重要步骤的前后，为子类提供额外的算法扩展点。

## 真实世界类比

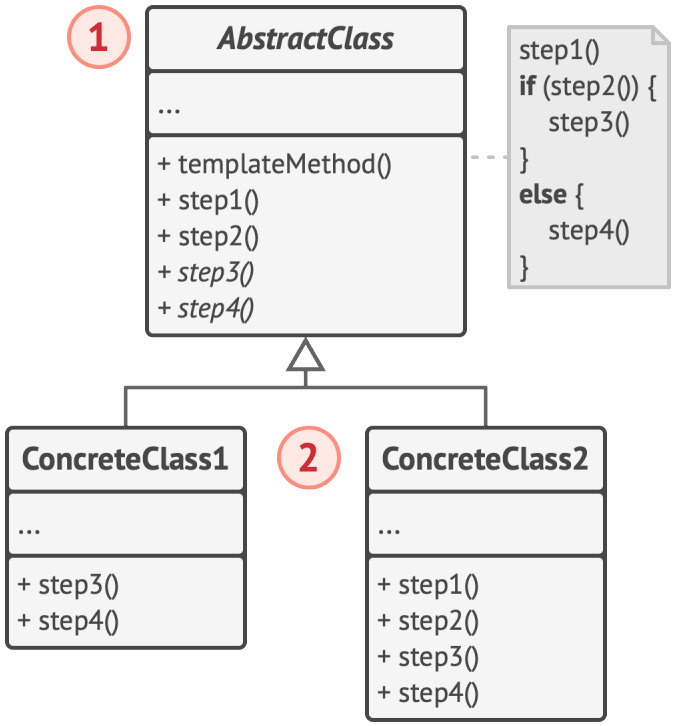


可对典型的建筑方案进行微调以更好地满足客户需求。

模板方法可用于建造大量房屋。标准房屋建造方案中可提供几个扩展点，允许潜在房屋业主调整成品房屋的部分细节。

每个建造步骤（例如打地基、建造框架、建造墙壁和安装水电管线等）都能进行微调，这使得成品房屋会略有不同。

## 结构

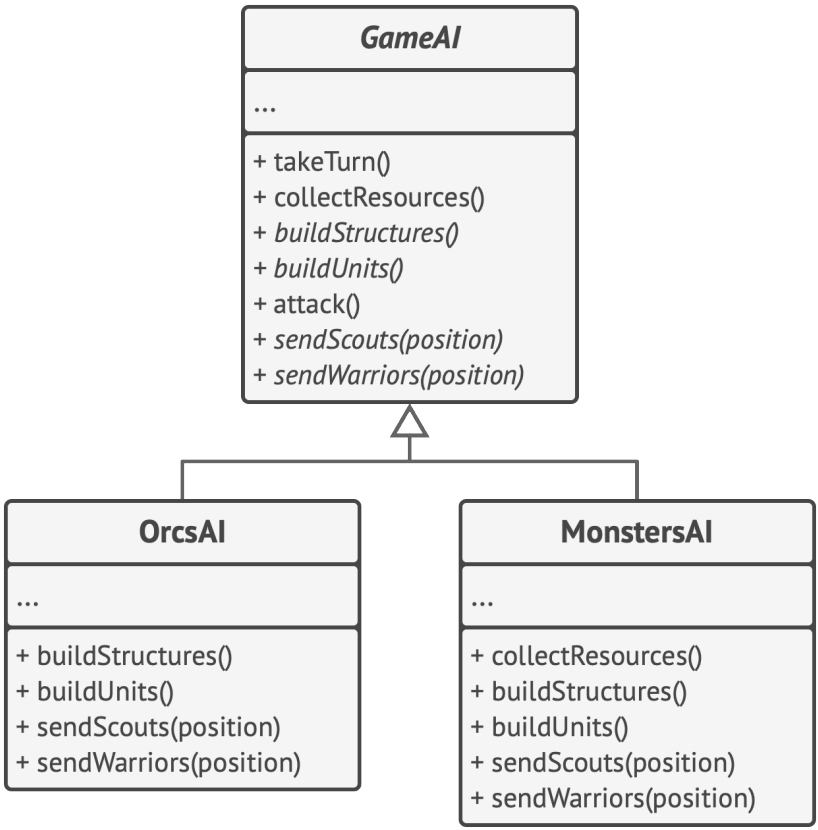



1. **抽象类**（AbstractClass）会声明作为算法步骤的方法，以及依次调用它们的实际模板方法。算法步骤可以被声明为抽象类型，也可以提供一些默认实现。

2. **具体类**（ConcreteClass）可以重写所有步骤，但不能重写模板方法自身。

# 伪代码

本例中的**模板方法**模式为一款简单策略游戏中人工智能的不同分支提供“框架”。




一款简单游戏的 AI 类。

游戏中所有的种族都有几乎同类的单位和建筑。因此你可以在不同的种族上复用相同的 AI 结构，同时还需要具备重写一些细节的能力。通过这种方式，你可以重写半兽人的 AI 使其更富攻击性，也可以让人类侧重防守，还可以禁止怪物建造建筑。在游戏中新增种族需要创建新的 AI 子类，还需要重写 AI 基类中所声明的默认方法。

```
// 抽象类定义了一个模板方法，其中通常会包含某个由抽象原语操作调用组成的算
// 法框架。具体子类会实现这些操作，但是不会对模板方法做出修改。
class GameAI is
  // 模板方法定义了某个算法的框架。
  method turn() is
    collectResources()
    buildStructures()
    buildUnits()
    attack()

  // 某些步骤可在基类中直接实现。
  method collectResources() is
    foreach (s in this.builtStructures) do
      s.collect()

  // 某些可定义为抽象类型。
  abstract method buildStructures()
  abstract method buildUnits()

  // 一个类可包含多个模板方法。
  method attack() is
    enemy = closestEnemy()
    if (enemy == null)
```

```
      sendScouts(map.center)
    else
      sendWarriors(enemy.position)

  abstract method sendScouts(position)
  abstract method sendWarriors(position)

// 具体类必须实现基类中的所有抽象操作，但是它们不能重写模板方法自身。
class OrcsAI extends GameAI is
  method buildStructures() is
    if (there are some resources) then
      // 建造农场，接着是谷仓，然后是要塞。

  method buildUnits() is
    if (there are plenty of resources) then
      if (there are no scouts)
        // 建造苦工，将其加入侦查编组。
      else
        // 建造兽族步兵，将其加入战士编组。

  // ...

  method sendScouts(position) is
    if (scouts.length > 0) then
      // 将侦查编组送到指定位置。

  method sendWarriors(position) is
    if (warriors.length > 5) then
      // 将战斗编组送到指定位置。

// 子类可以重写部分默认的操作。
class MonstersAI extends GameAI is
```

```
56    method collectResources() is
57      // 怪物不会采集资源。
58
59    method buildStructures() is
60      // 怪物不会建造建筑。
61
62    method buildUnits() is
63      // 怪物不会建造单位。
```

## 适合应用场景

**当你只希望客户端扩展某个特定算法步骤，而不是整个算法或其结构时，可使用模板方法模式。**

模板方法将整个算法转换为一系列独立的步骤，以便子类能对其进行扩展，同时还可让超类中所定义的结构保持完整。

**当多个类的算法除一些细微不同之外几乎完全一样时，你可使用该模式。但其后果就是，只要算法发生变化，你就可能需要修改所有的类。**

在将算法转换为模板方法时，你可将相似的实现步骤提取到超类中以去除重复代码。子类间各不同的代码可继续保留在子类中。

# 实现方式

1. 分析目标算法，确定能否将其分解为多个步骤。从所有子类的角度出发，考虑哪些步骤能够通用，哪些步骤各不相同。

2. 创建抽象基类并声明一个模板方法和代表算法步骤的一系列抽象方法。在模板方法中根据算法结构依次调用相应步骤。可用 final 最终 修饰模板方法以防止子类对其进行重写。

3. 虽然可将所有步骤全都设为抽象类型，但默认实现可能会给部分步骤带来好处，因为子类无需实现那些方法。

4. 可考虑在算法的关键步骤之间添加钩子。

5. 为每个算法变体新建一个具体子类，它必须实现所有的抽象步骤，也可以重写部分可选步骤。

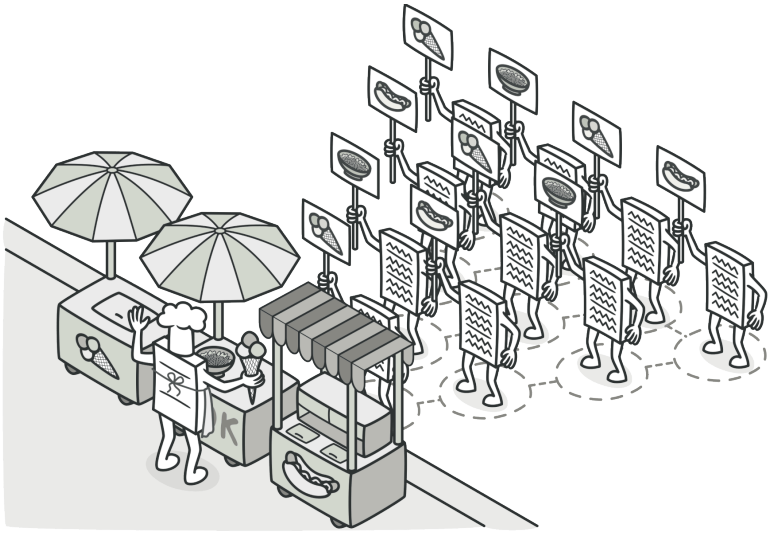
# 优缺点

- ✓ 你可仅允许客户端重写一个大型算法中的特定部分，使得算法其他部分修改对其所造成的影响减小。
- ✓ 你可将重复代码提取到一个超类中。
- × 部分客户端可能会受到算法框架的限制。
- × 通过子类抑制默认步骤实现可能会导致违反_里氏替换原则_。
- × 模板方法中的步骤越多，其维护工作就可能会越困难。

# ⇄ 与其他模式的关系

- **工厂方法**是**模板方法**的一种特殊形式。同时，工厂方法可以作为一个大型模板方法中的一个步骤。

- **模板方法**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。



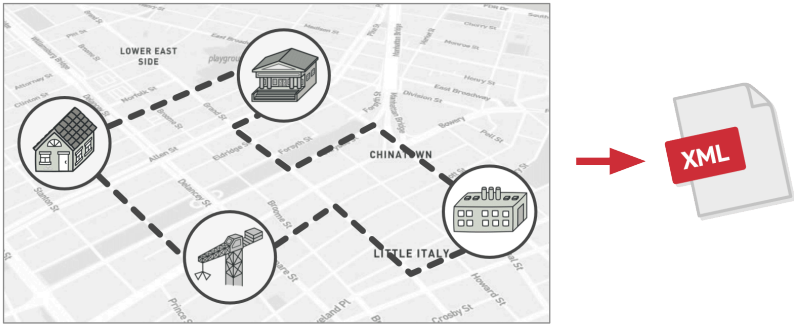
# 访问者

亦称：Visitor

**访问者**是一种行为设计模式，
它能将算法与其所作用的对
象隔离开来。

# ☹ 问题

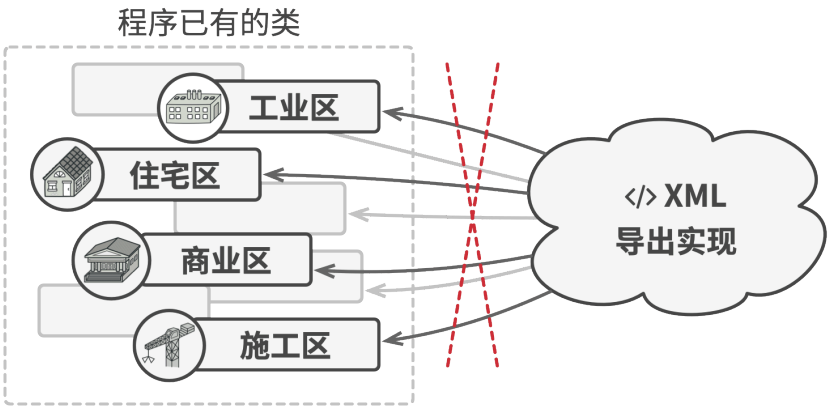
假如你的团队开发了一款能够使用巨型图像中地理信息的应用程序。图像中的每个节点既能代表复杂实体（例如一座城市），也能代表更精细的对象（例如工业区和旅游景点等）。如果节点代表的真实对象之间存在公路，那么这些节点就会相互连接。在程序内部，每个节点的类型都由其所属的类来表示，每个特定的节点则是一个对象。




将图像导出为 XML。

一段时间后，你接到了实现将图像导出到 XML 文件中的任务。这些工作最初看上去非常简单。你计划为每个节点类添加导出函数，然后递归执行图像中每个节点的导出函数。解决方案简单且优雅：使用多态机制可以让导出方法的调用代码不会和具体的节点类相耦合。

但你不太走运，系统架构师拒绝批准对已有节点类进行修改。他认为这些代码已经是产品了，不想冒险对其进行修改，因为修改可能会引入潜在的缺陷。




所有节点的类中都必须添加导出至 XML 文件的方法，但如果在修改代码的过程中引入了任何缺陷，那么整个程序都会面临风险。

此外，他还质疑在节点类中包含导出 XML 文件的代码是否有意义。这些类的主要工作是处理地理数据。导出 XML 文件的代码放在这里并不合适。

还有另一个原因，那就是在此项任务完成后，营销部门很有可能会要求程序提供导出其他类型文件的功能，或者提出其他奇怪的要求。这样你很可能会被迫再次修改这些重要但脆弱的类。

# 😊解决方案

访问者模式建议将新行为放入一个名为**访问者**的独立类中，而不是试图将其整合到已有类中。现在，需要执行操作的原始对象将作为参数被传递给访问者中的方法，让方法能访问对象所包含的一切必要数据。

如果现在该操作能在不同类的对象上执行会怎么样呢？比如在我们的示例中，各节点类导出 XML 文件的实际实现很可能会稍有不同。因此，访问者类可以定义一组（而不是一个）方法，且每个方法可接收不同类型的参数，如下所示：

```
class ExportVisitor implements Visitor is
  method doForCity(City c) { ... }
  method doForIndustry(Industry f) { ... }
  method doForSightSeeing(SightSeeing ss) { ... }
  // ...
```

但我们究竟应该如何调用这些方法（尤其是在处理整个图像方面）呢？这些方法的签名各不相同，因此我们不能使用多态机制。为了可以挑选出能够处理特定对象的访问者方法，我们需要对它的类进行检查。这是不是听上去像个噩梦呢？

```
foreach (Node node in graph)
  if (node instanceof City)
    exportVisitor.doForCity((City) node)
  if (node instanceof Industry)
    exportVisitor.doForIndustry((Industry) node)
  // ...
}
```

你可能会问，我们为什么不使用方法重载呢？就是使用相同的方法名称，但它们的参数不同。不幸的是，即使我们的编程语言（例如 Java 和 C#）支持重载也不行。由于我们无法提前知晓节点对象所属的类，所以重载机制无法执行正确的方法。方法会将 `节点` 基类作为输入参数的默认类型。

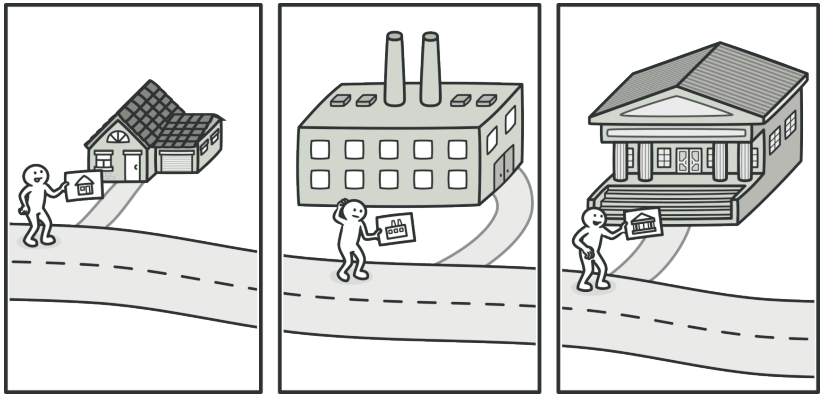
但是，访问者模式可以解决这个问题。它使用了一种名为**双分派**的技巧，不使用累赘的条件语句也可下执行正确的方法。与其让客户端来选择调用正确版本的方法，不如将选择权委派给作为参数传递给访问者的对象。由于该对象知晓其自身的类，因此能更自然地在访问者中选出正确的方法。它们会“接收”一个访问者并告诉其应执行的访问者方法。

```
// 客户端代码
foreach (Node node in graph)
  node.accept(exportVisitor)

// 城市
class City is
  method accept(Visitor v) is
    v.doForCity(this)
  // ...

// 工业区
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this)
  // ...
```

我承认最终还是修改了节点类，但毕竟改动很小，且使得我们能够在后续进一步添加行为时无需再次修改代码。

现在，如果我们抽取出所有访问者的通用接口，所有已有的节点都能与我们在程序中引入的任何访问者交互。如果需要引入与节点相关的某个行为，你只需要实现一个新的访问者类即可。

# 真实世界类比

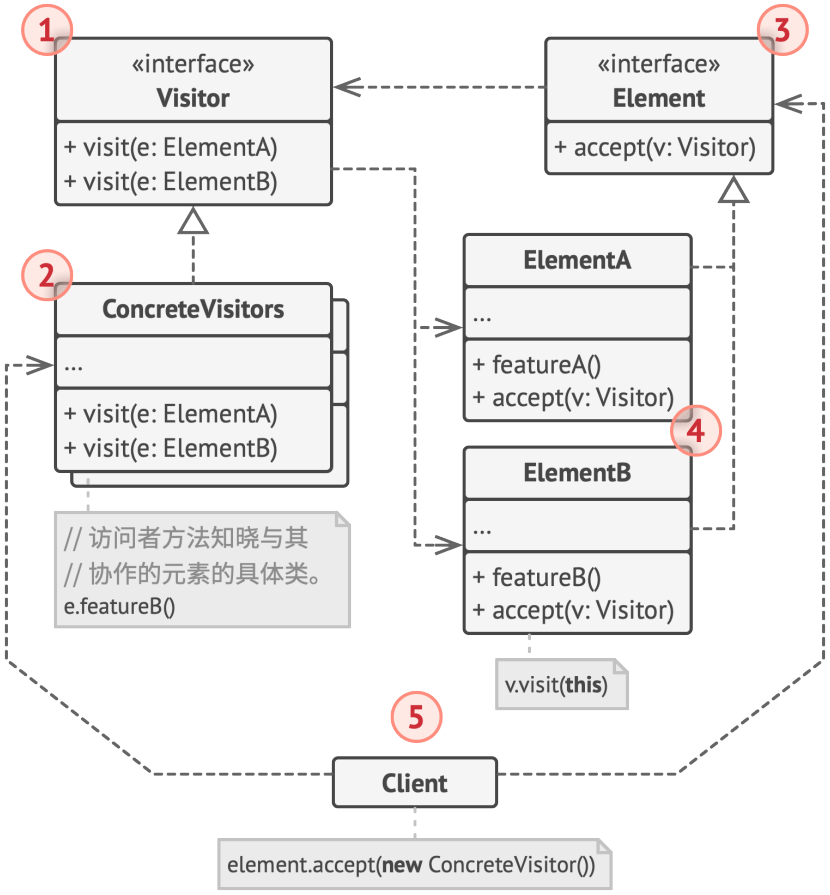


优秀的保险代理人总能为不同类型的团体提供不同的保单。

假如有这样一位非常希望赢得新客户的资深保险代理人。他可以拜访街区中的每栋楼，尝试向每个路人推销保险。所以，根据大楼内组织类型的不同，他可以提供专门的保单：

- 如果建筑是居民楼，他会推销医疗保险。
- 如果建筑是银行，他会推销失窃保险。
- 如果建筑是咖啡厅，他会推销火灾和洪水保险。

# 结构




1. **访问者**（Visitor）接口声明了一系列以对象结构的具体元素为参数的访问者方法。如果编程语言支持重载，这些方法的名称可以是相同的，但是其参数一定是不同的。

2. **具体访问者**（Concrete Visitor）会为不同的具体元素类实现相同行为的几个不同版本。

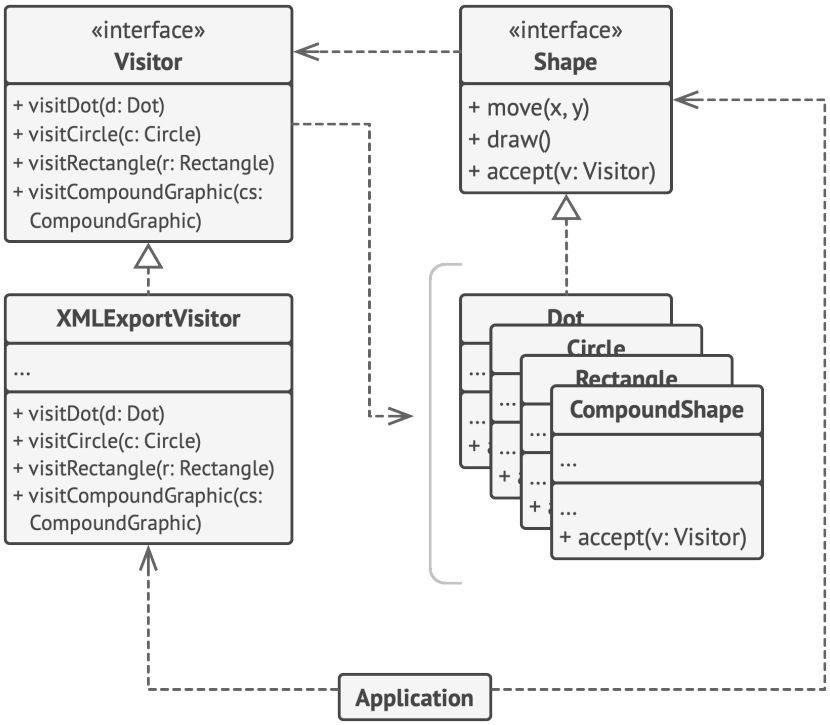
3. **元素**（Element）接口声明了一个方法来“接收”访问者。该方法必须有一个参数被声明为访问者接口类型。

4. **具体元素**（Concrete Element）必须实现接收方法。该方法的目的是根据当前元素类将其调用重定向到相应访问者的方法。请注意，即使元素基类实现了该方法，所有子类都必须对其进行重写并调用访问者对象中的合适方法。

5. **客户端**（Client）通常会作为集合或其他复杂对象（例如一个**组合**树）的代表。客户端通常不知晓所有的具体元素类，因为它们会通过抽象接口与集合中的对象进行交互。

# # 伪代码

在本例中，**访问者**模式为几何图像层次结构添加了对于 XML 文件导出功能的支持。




通过访问者对象将各种类型的对象导出为 XML 格式文件。

```
// 元素接口声明了一个`accept（接收）`方法，它会将访问者基础接口作为一个参
// 数。
interface Shape is
  method move(x, y)
  method draw()
  method accept(v: Visitor)

// 每个具体元素类都必须以特定方式实现`accept`方法，使其能调用相应元素类的
// 访问者方法。
class Dot implements Shape is
  // ...
```

```
12
13    // 注意我们正在调用的`visitDot（访问点）`方法与当前类的名称相匹配。
14    // 这样我们能让访问者知晓与其交互的元素类。
15    method accept(v: Visitor) is
16      v.visitDot(this)
17
18  class Circle implements Shape is
19    // ...
20    method accept(v: Visitor) is
21      v.visitCircle(this)
22
23  class Rectangle implements Shape is
24    // ...
25    method accept(v: Visitor) is
26      v.visitRectangle(this)
27
28  class CompoundShape implements Shape is
29    // ...
30    method accept(v: Visitor) is
31      v.visitCompoundShape(this)
32
33
34  // 访问者接口声明了一组与元素类对应的访问方法。访问方法的签名能让访问者准
35  // 确辨别出与其交互的元素所属的类。
36  interface Visitor is
37    method visitDot(d: Dot)
38    method visitCircle(c: Circle)
39    method visitRectangle(r: Rectangle)
40    method visitCompoundShape(cs: CompoundShape)
41
42  // 具体访问者实现了同一算法的多个版本，而且该算法能与所有具体类进行交互。
43  //
```

```
// 访问者模式在复杂对象结构（例如组合树）上使用时能发挥最大作用。在这种情
// 况下，它可以存储算法的一些中间状态，并同时在结构中的不同对象上执行访问
// 者方法。这可能会非常有帮助。
class XMLExportVisitor implements Visitor is
  method visitDot(d: Dot) is
    // 导出点（dot）的 ID 和中心坐标。

  method visitCircle(c: Circle) is
    // 导出圆（circle）的 ID 、中心坐标和半径。

  method visitRectangle(r: Rectangle) is
    // 导出长方形（rectangle）的 ID 、左上角坐标、宽和长。

  method visitCompoundShape(cs: CompoundShape) is
    // 导出图形（shape）的 ID 和其子项目的 ID 列表。


// 客户端代码可在不知晓具体类的情况下在一组元素上运行访问者操作。“接收”操
// 作会将调用定位到访问者对象的相应操作上。
class Application is
  field allShapes: array of Shapes

  method export() is
    exportVisitor = new XMLExportVisitor()

    foreach (shape in allShapes) do
      shape.accept(exportVisitor)
```

如果你并不十分理解为何本例中需要使用 accept 接收 方法，我的一篇文章**访问者和双分派**详细解释了这个问题。

# 适合应用场景

**如果你需要对一个复杂对象结构（例如对象树）中的所有元素执行某些操作，可使用访问者模式。**

访问者模式通过在访问者对象中为多个目标类提供相同操作的变体，让你能在属于不同类的一组对象上执行同一操作。

**可使用访问者模式来清理辅助行为的业务逻辑。**

该模式会将所有非主要的行为抽取到一组访问者类中，使得程序的主要类能更专注于主要的工作。

**当某个行为仅在类层次结构中的一些类中有意义，而在其他类中没有意义时，可使用该模式。**

你可将该行为抽取到单独的访问者类中，只需实现接收相关类的对象作为参数的访问者方法并将其他方法留空即可。

# 实现方式

1. 在访问者接口中声明一组“访问”方法，分别对应程序中的每个具体元素类。

2. 声明元素接口。如果程序中已有元素类层次接口，可在层次结构基类中添加抽象的“接收”方法。该方法必须接受访问者对象作为参数。

3. 在所有具体元素类中实现接收方法。这些方法必须将调用重定向到当前元素对应的访问者对象中的访问者方法上。

4. 元素类只能通过访问者接口与访问者进行交互。不过访问者必须知晓所有的具体元素类，因为这些类在访问者方法中都被作为参数类型引用。

5. 为每个无法在元素层次结构中实现的行为创建一个具体访问者类并实现所有的访问者方法。

   你可能会遇到访问者需要访问元素类的部分私有成员变量的情况。在这种情况下，你要么将这些变量或方法设为公有，这将破坏元素的封装；要么将访问者类嵌入到元素类中。后一种方式只有在支持嵌套类的编程语言中才可能实现。

6. 客户端必须创建访问者对象并通过“接收”方法将其传递给元素。

## 优缺点

✓ 开闭原则。你可以引入在不同类对象上执行的新行为，且无需对这些类做出修改。

✓ 单一职责原则。可将同一行为的不同版本移到同一个类中。

✓ 访问者对象可以在与各种对象交互时收集一些有用的信息。当你想要遍历一些复杂的对象结构（例如对象树），并在结构中的每个对象上应用访问者时，这些信息可能会有所帮助。

✕ 每次在元素层次结构中添加或移除一个类时，你都要更新所有的访问者。

✕ 在访问者同某个元素进行交互时，它们可能没有访问元素私有成员变量和方法的必要权限。

## ⇄ 与其他模式的关系

- 你可以将**访问者**视为**命令**模式的加强版本，其对象可对不同类的多种对象执行操作。

- 你可以使用**访问者**对整个**组合**树执行操作。

- 可以同时使用**访问者**和**迭代器**来遍历复杂数据结构，并对其中的元素执行所需操作，即使这些元素所属的类完全不同。

# 结语

## 祝贺你已经读到了本书的最后部分！

但是，世界上还有许多其他的模式。我希望本书能够成为你学习模式和获取程序设计超能力的起点。

以下的几个想法会对你决定下一步去做什么有所帮助。

- </>不要忘记你还可以**访问存档**[1]以下载不同编程语言的代码示例。
- 阅读约书亚·克里耶夫斯基撰写的《**重构与模式**[2]》。
- 对重构一无所知？**我有一门适合你的课程**[3]。
- 将这些模**式速查表**[4]打印出来，并将其放在你随时能够看到的地方。
- 给本书提出**意见和建议**[5]。我非常期待了解你的想法，即使是严肃的批评也没关系 ☺

---

1. 访问存档：**https://refactoringguru.cn/home**
2. 《重构与模式》：**https://refactoringguru.cn/ref-to-patterns-book**
3. 重构：**https://refactoringguru.cn/refactoring**
4. 式速查表：**https://refactoringguru.cn/design-patterns/cheatsheets**
5. 联系我们：**https://refactoringguru.cn/refactoring/feedback**