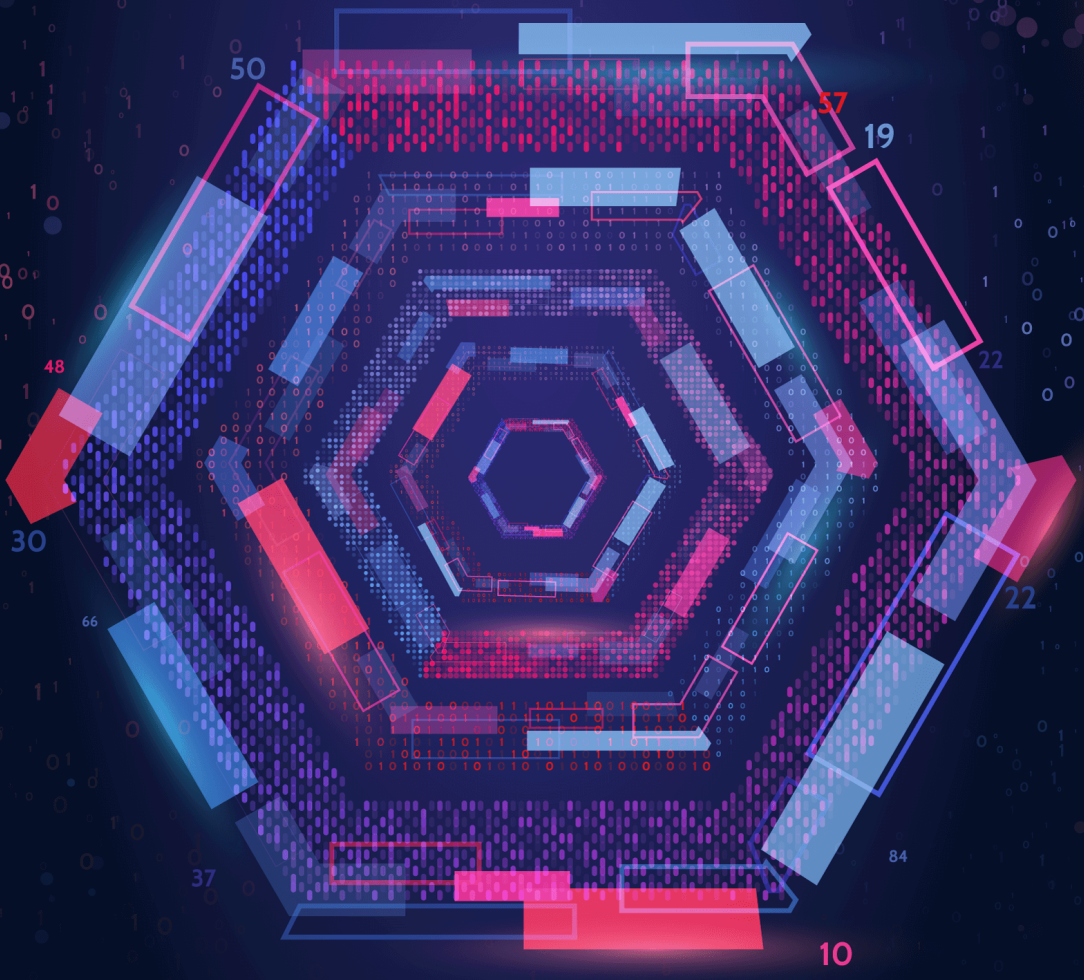
深入
设计模式



亚历山大·什韦茨 (Alexander Shvets) 著
彭力 译

REFACTORING
·GURU·

深入
设计模式

v2021-1.20

演示版本

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

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**。

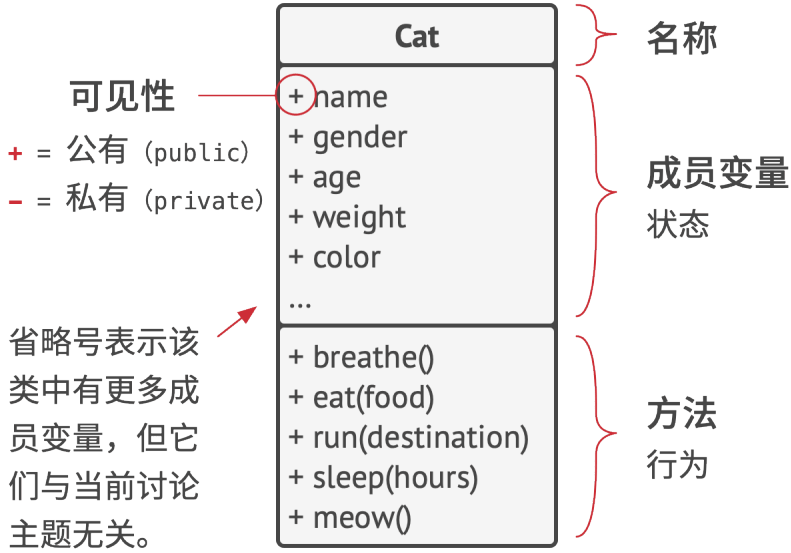
面向对象程序设计简介

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。

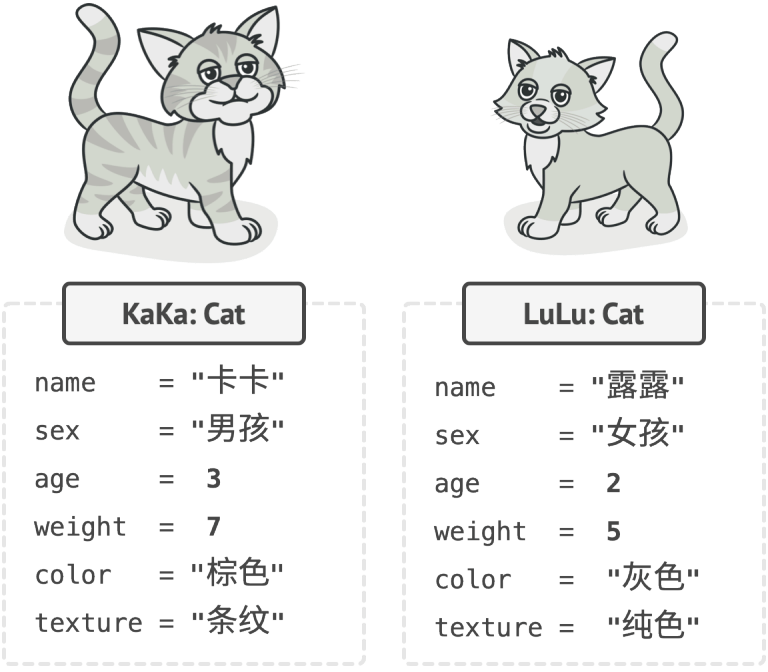


这是一个 UML 类图。你将在本书中看到许多类似的图示。

> 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。

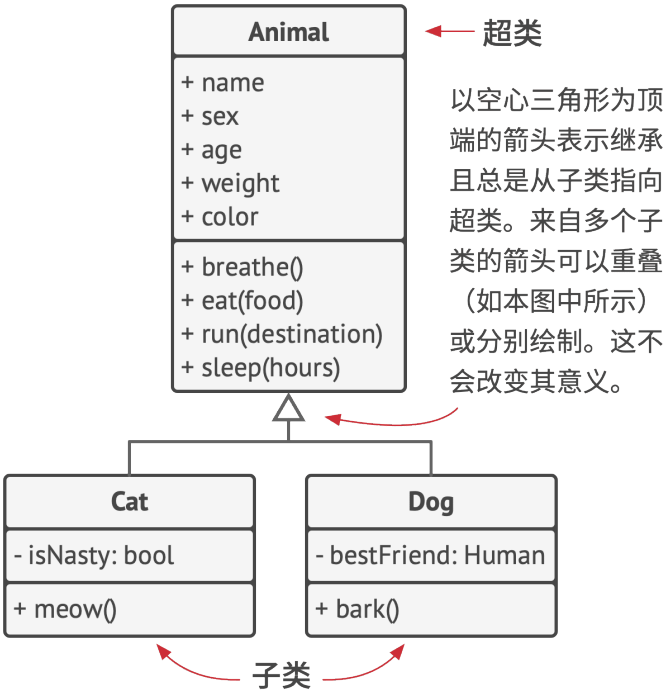
你朋友的猫“露露”也是 `猫` 这个类的一个实例。它拥有与“卡卡”相同的一组属性。不同之处在于这些属性的值：她的性别是“女孩”；她的毛色不同；体重较轻。因此类就像是定义对象结构的蓝图，而对象则是类的具体实例。

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。

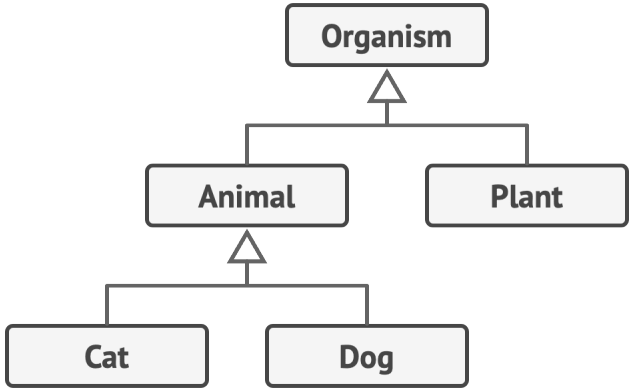
假如你的邻居有一只名为“福福”的狗。其实狗和猫有很多相同的地方：它们都有名字、性别、年龄和毛色等属性。狗和猫一样可以呼吸、睡觉和奔跑。因此似乎我们可定义一个动物 Animal 基类来列出它们所共有的属性和行为。

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。

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
```

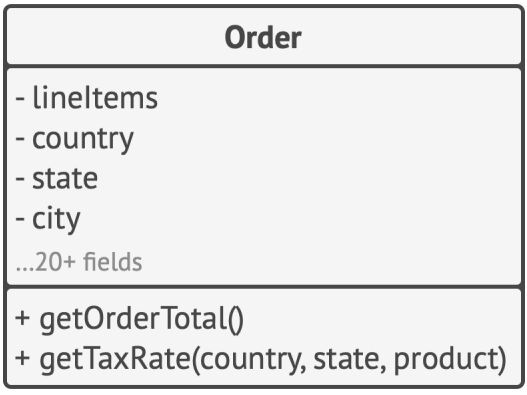
```
5
6     total += total * getTaxRate(order.country)
7
8     return total
9
10  method getTaxRate(country) is
11    if (country == "US")
12      return 0.07 // 美国营业税
13    else if (country == "EU")
14      return 0.20 // 欧洲增值税
15    else
16      return 0
```

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。

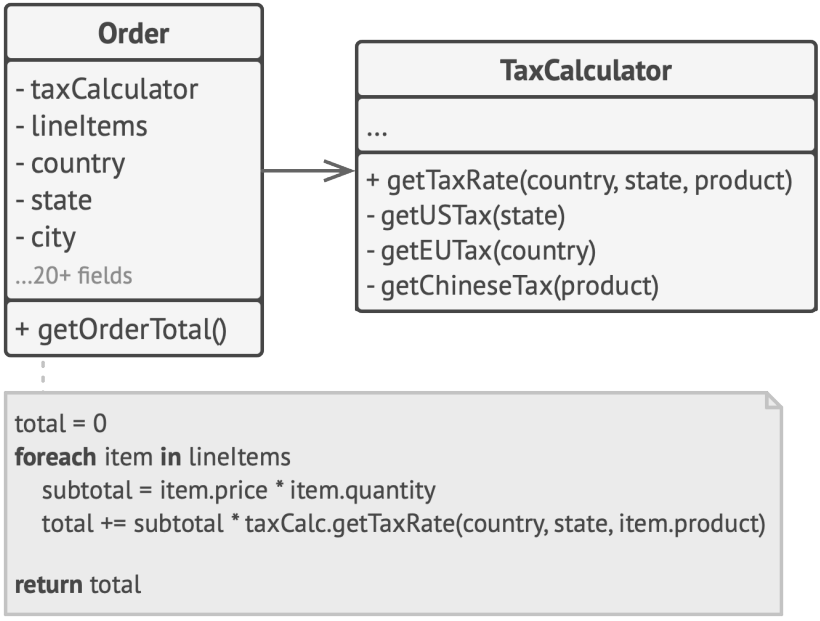
类层面的封装

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。

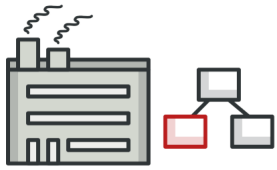


修改后：对订单类隐藏税金计算。

设计模式目录

创建型模式

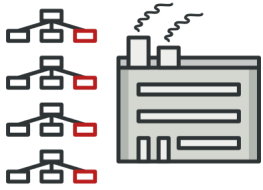
创建型模式提供了创建对象的机制，能够提升已有代码的灵活性和可复用性。



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

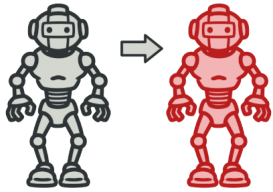
Abstract Factory

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

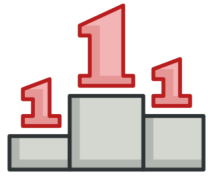
使你能够分步骤创建复杂对象。 该模式允许你使用相同的创建代码生成不同类型和形式的对象。



原型

Prototype

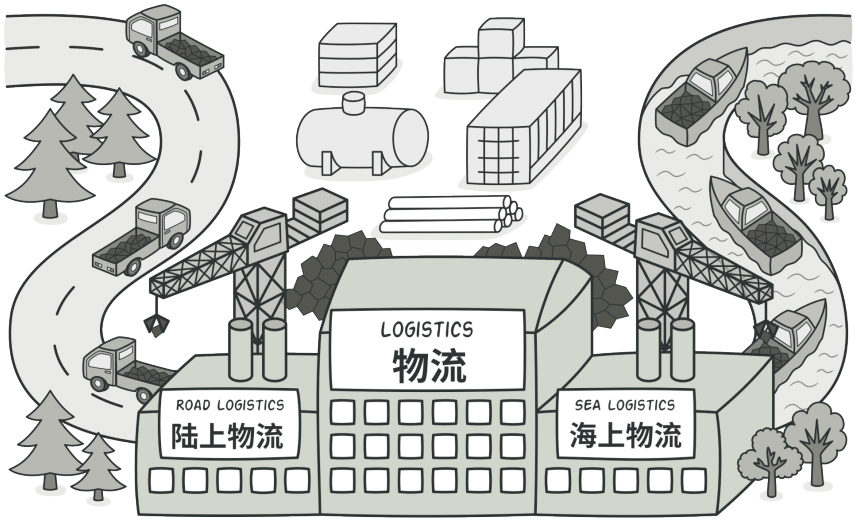
让你能够复制已有对象，而又无需使代码依赖它们所属的类。



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

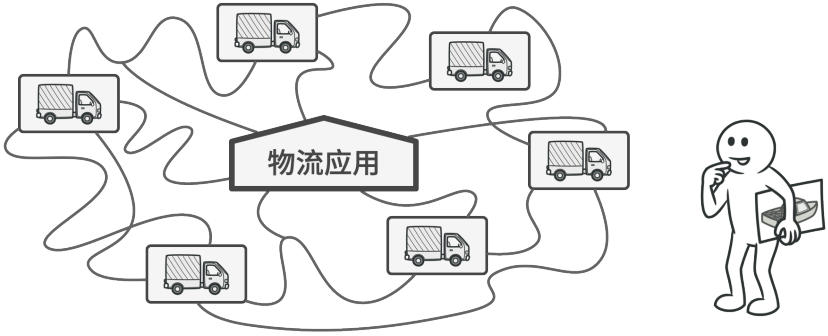
亦称：虚拟构造函数、Virtual Constructor、Factory Method

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。



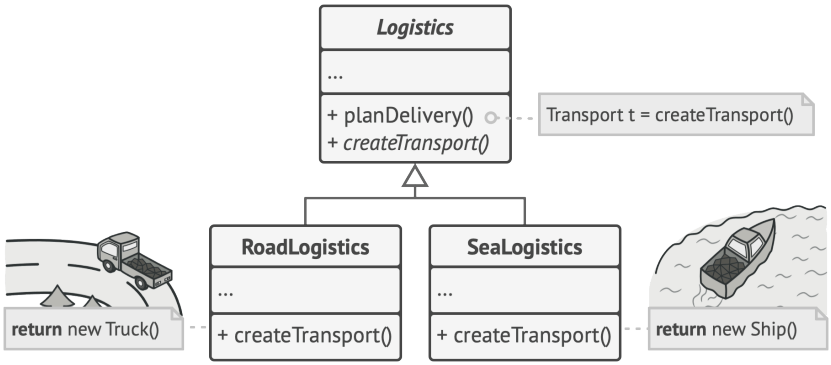
如果代码其余部分与现有类已经存在耦合关系，那么向程序中添加新类其实并没有那么容易。

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

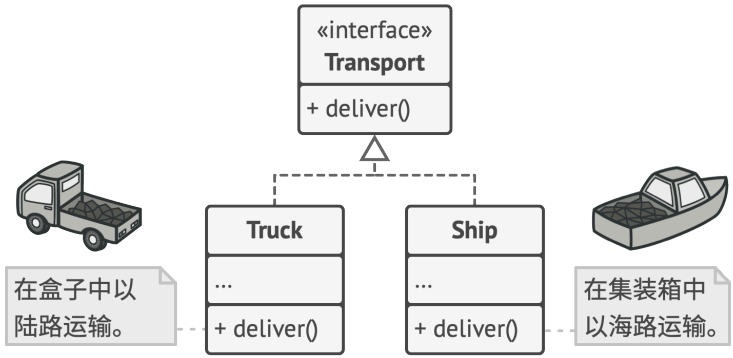
工厂方法模式建议使用特殊的工厂方法代替对于对象构造函数的直接调用（即使用 new 运算符）。不用担心，对象仍将通过 new 运算符创建，只是该运算符改在工厂方法中调用罢了。工厂方法返回的对象通常被称作“产品”。



子类可以修改工厂方法返回的对象类型。

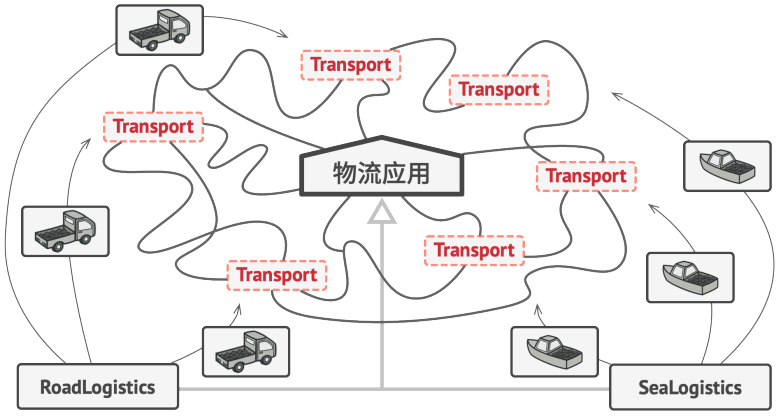
乍看之下，这种更改可能毫无意义：我们只是改变了程序中调用构造函数的位置而已。但是，仔细想一下，现在你可以在子类中重写工厂方法，从而改变其创建产品的类型。

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。

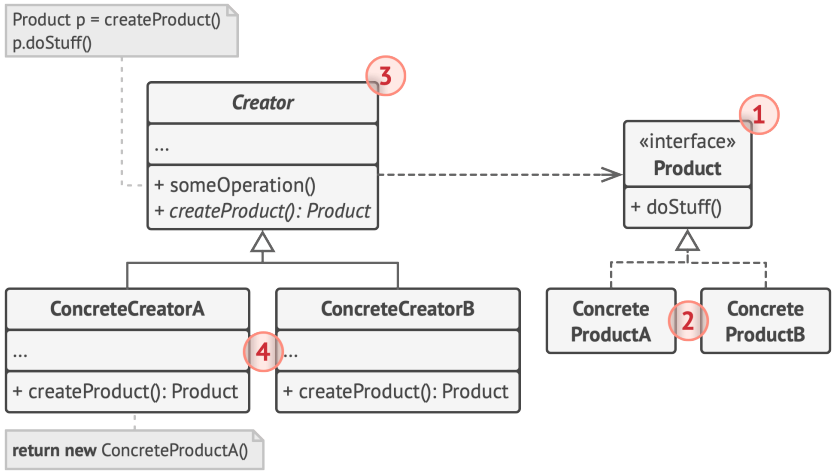
举例来说，卡车 Truck 和轮船 Ship 类都必须实现运输 Transport 接口，该接口声明了一个名为 deliver 交付的方法。每个类都将以不同的方式实现该方法：卡车走陆路交付货物，轮船走海路交付货物。陆路运输 RoadLogistics 类中的工厂方法返回卡车对象，而海路运输 SeaLogistics 类则返回轮船对象。



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。

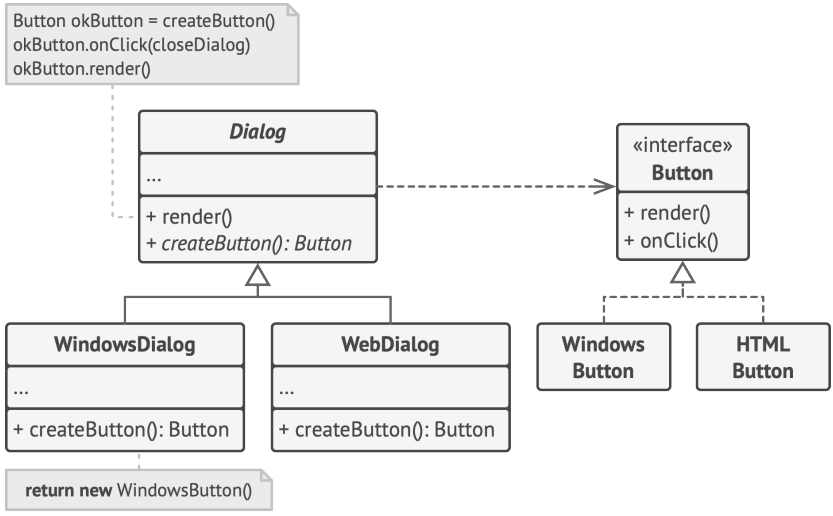
注意，尽管它的名字是创建者，但他最主要的职责并**不是**创建产品。一般来说，创建者类包含一些与产品相关的核心业务逻辑。工厂方法将这些逻辑处理从具体产品类中分离出来。打个比方，大型软件开发公司拥有程序员培训部门。但是，这些公司的主要工作还是编写代码，而非生产程序员。

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。