设计模式简介

设计模式（Design pattern）代表了**最佳的实践**，软件设计中常见问题的典型解决方案。这些解决方案是众多软件开发人员经过**相当长的一段时间的试验和错误总结**出来的。

设计模式是**一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结**。

模式包含哪些内容?

- **意图**部分简单描述问题和解决方案。
- **动机**部分将进一步解释问题并说明模式会如何提供解决方案。
- **结构**部分展示模式的每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。

为什么以及如何学习设计模式?

- 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。

- 让代码更容易被他人理解

  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。

- 为了重用代码、保证代码可靠性。

设计模式分类

- **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。
  - 工厂模式
    - **工厂方法模式**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。
  - 单列模式
    - **单例模式**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。
  - 抽象工厂模式
    - **抽象工厂模式**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。
  - 生成器模式
    - **生成器模式**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。
  - 原型模式
    - **原型模式**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。

- **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。
  - 装饰

        - **装饰模式**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。
    - 代理
        - **代理模式**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。
    - 适配器
        - **适配器模式**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。
    - 桥接
        - **桥接模式**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。
    - 组合
        - **组合模式**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。
    - 外观
        - **外观模式**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。
    - 享元
        - **享元模式**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。
- **行为模式**负责对象间的高效沟通和职责委派。
    - 观察者
        - **观察者模式**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。
    - 策略
        - **策略模式**是一种行为设计模式， 它能让你定义一系列算法， 并将每种算法分别放入独立的类中， 以使算法的对象能够相互替换。
    - 责任链
        - **责任链模式**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。
    - 命令
        - **命令模式**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。
    - 迭代器
        - **迭代器模式**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。
    - 中介者
        - **中介者模式**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。
    - 备忘录
        - **备忘录模式**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。
    - 状态

        - **状态模式**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。
    - 模型方法
        - **模板方法模式**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。
    - 访问者
        - **访问者模式**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。

工厂方法模式

工厂方法模式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。

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。

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。

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。

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。

工厂方法模式优缺点

优点

- 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

缺点

- 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。

单例模式

单例模式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。

单例模式适合应用场景

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。

如果你需要更加严格地控制全局变量， 可以使用单例模式。

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。

单例模式优缺点

优点

- 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。

缺点

- 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。

代理模式

代理模式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。

适合应用场景

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。**

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。

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。

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。

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。

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。

代理模式优缺点

优点：

- 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。

缺点：

- 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。

装饰模式

装饰模式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。

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。

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。

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。

装饰模式优缺点

优点：

- 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。

缺点：

- 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。

观察者模式

观察者模式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。

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。

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。

观察者模式优缺点

优点：

- *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。

缺点：

* 订阅者的通知顺序是随机的,不能控制。

策略模式

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

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。

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。

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。

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。

实现方式

1. 从上下文类中找出修改频率较高的算法 （也可能是用于在运行时选择某个算法变体的复杂条件运算符）。
2. 声明该算法所有变体的通用策略接口。
3. 将算法逐一抽取到各自的类中， 它们都必须实现策略接口。
4. 在上下文类中添加一个成员变量用于保存对于策略对象的引用。 然后提供设置器以修改该成员变量。 上下文仅可通过策略接口同策略对象进行交互， 如有需要还可定义一个接口来让策略访问其数据。
5. 客户端必须将上下文类与相应策略进行关联， 使上下文可以预期的方式完成其主要工作。

策略模式优缺点

优点：

- 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。

缺点：

- 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。