设计模式

设计模式是什么

设计模式是一个通过定义、使用、测试去解决特定问题的方法，是针对软件设计中在给定条件下会重复性发生的问题而提出的一种通用性的可重用解决方案，设计模式不是可以直接转化为代码的完整设计，它是用于描述在不同情况下解决问题的通用方案。

设计模式的作用

设计模式通过提供经过验证的行之有效的开发范式加快开发过程，预防重大的隐患问题，提高代码可读性。

设计模式的分类

这里主要讨论GoF所提出的23种设计模式，可将其分为三种类型：

1. 创造型设计模式
2. 结构型设计模式
3. 行为型设计模式

创造型设计模式

注重完成对象的实例化，相比于直接实例化对象，根据实际情况选择合适的设计模式完成对象的实例化，可以为复杂的业务场景带来更高的灵活性。
创造型设计模式主要包括以下几种：

1. 抽象工厂设计模式
2. 生成器设计模式
3. 工厂方法设计模式
4. 原型设计模式
5. 单例设计模式

结构型设计模式

结构型设计模式用于指导我们完成对代码的结构划分，如此，代码结构会更加清晰，更易理解，也提高了软件的可维护性。
结构型设计模式主要包括以下几种：

1. 适配器设计模式
2. 桥接设计模式
3. 组合设计模式
4. 装饰设计模式
5. 门面设计模式
6. 享元设计模式
7. 代理设计模式

行为型设计模式

行为型设计模式主要用于定义对象之间的通信与流程控制，主要的设计模式都非常注重优化对象之间的数据交互方式。
行为型设计模式主要包括以下几种：

1. 职责链设计模式
2. 命令设计模式
3. 解释器设计模式
4. 迭代器设计模式
5. 中介者设计模式
6. 备忘录设计模式
7. 观察者设计模式
8. 策略设计模式
9. 状态设计模式
10. 模板方法设计模式
11. 访问者设计模式

如何学习设计模式

- 模式名称是什么？
- 模式类型是什么？是创造型，结构型，还是行为型？
- 模式的目的是什么？（作用是什么？解决了什么问题？）
- 模式的别名是什么？
- 什么情况下使用该模式？
- 该模式的基本示例
- 该模式的UML图是什么样的？是类图还是交互图？
- 都有那些对象在模式中参与活动？列出设计模式中使用的类和对象，并说明他们各自的角色
- 模式中的类和对象是怎么进行交互的？
- 通过应用设计模式能获取什么好处，有哪些坏处？如何权衡？
- 如何实现该模式
- 与该模式相近的设计模式是什么？这几种相近的模式有哪些异同点？

正确看待设计模式

恰当使用设计模式能够提高代码的复用性，但是由于复用性往往会引入封装与间接调用，这些会降低系统性能，增加代码复杂程度。因此，除非设计模式能够帮助我们完成代码的实现或者后续的维护工作，否则没有必要去引入设计模式。
学习设计模式的关键并不在于学习设计模式本身，而是在于识别应用场景与潜在的风险，并将设计模式用之有道，这般，设计模式才能算作得心应手的工具。
在没有必要的情况大可不必去使用设计模式，因为设计模式有可能会牺牲代码的简洁性，而且滥用设计模式多会引入新的问题却没有解决原来的问题。
保持代码的整洁，模块化和可读性，同时不要让各类之间过度耦合。

创造型设计模式

创造型设计模式主要关注的是类的实例化，也就是说体现的是对象的创建方法，利用这些模式，我们可以在适当的情况下以适当的形式创建对象，创造型设计模式通过控制对象的创建来解决设计中的问题。
创造型设计模式主要包含以下子类别：

1. 对象创造型设计模式：

主要完成对象创建，并将对象中部分内容放到其他对象中创建。

2. 类创造型设计模式：

主要完成类的实例化，并将类中的部分对象放到子类中创建，此类模式在实例化过程中高效地利用了继承机制
创造型设计模式主要包含以下5种具体的设计模式：

1. 抽象工厂设计模式

提供一个用于创建相关对象或相互依赖对象的接口，无需指定对象的具体类

2. 生成器设计模式

将复杂对象的构建与其表示相互分离，使得同样的构建过程可以创建不同的表示

3. 工厂方法设计模式

允许在子类中实现本类的实例化类

4. 原型设计模式

使用一个原型实例来指定创建对象的种类，然后通过拷贝这些原型实现新对象的创建

5. 单例模式

确保某个类在系统中仅有一个实例，并提供一个访问它的全局访问点

对象创造型设计模式	类创造型设计模式
抽象工厂设计模式	工厂方法设计模式
生成器设计模式	
原型设计模式	
单例设计模式	

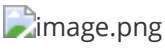
工厂方法设计模式

工厂方法的作用是创建对象，用来从一组实现特定逻辑的类中实例化某个对象。

模式中包括的类

- 产品类（Product）中定义了工厂方法创建的对象接口。
- 具体产品类（ Concrete Product）实现产品类接口。
- 工厂类（ Creator，因为由它来创建产品类，所以叫作工厂类）声明工厂方法，返回一个产品类对象。可用于调用创建产品类对象的生成方法。
- 具体工厂类（ Concrete Creator）重写用于创建具体产品类对象的方法。

UML图



功能及应用场景

- 当需要创建一个类，而在编程时不能确定这个类的类型时（需要运行时确定）。
- 当一个类希望由其子类来指定所创建对象的具体类型时。
- 当我们想要定位被创建类，并获取相关信息时。

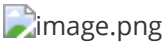
抽象工厂设计模式

抽象工厂模式相比于工厂方法模式的抽象层次更高。这意味着抽象工厂返回的是一组类的工厂。与工厂方法模式类似（返回多个子类中的一个），此方法会返回一个工厂，而这个工厂会返回多个子类中的一个。简单来说，抽象工厂是一个工厂对象，该对象又会返回若干工厂中的一个。
工厂模式是创造型模式的典型示例。抽象工厂设计模式是工厂方法模式的扩展，从而使我们无须担心所创建对象的实际类就能够创建对象。抽象工厂模式扩展了工厂方法模式，允许创建更多类型的对象。

模式中包括的类

- 抽象工厂（AbstractFactory）声明一个用于完成抽象产品对象创建操作的接口。
- 具体工厂（ConcreteFactory）实现创建具体产品对象的操作。
- 抽象产品（AbstractProduct）声明一个用于一类产品对象的接口。
- 具体产品（ConcreteProduct）定义由相应的具体工厂来创建的产品对象。
- 客户端（Client）使用由抽象工厂和抽象产品类声明的唯一接口。

UML图



功能及应用场景

抽象工厂模式的主要优点之一是它屏蔽了这些具体类的创建方法。实际应用的类名称不需要再让客户端（将客户端与具体类解耦）知道。由于具体类是屏蔽的，因此我们可以在不同的工厂（实现方法）之间进行切换。

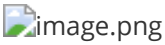
生成器设计模式

生成器模式，能够从简单的对象一步一步生成复杂的对象。生成器模式是一种用来逐步构建复杂对象并在最后一步返回对象的创造型模式。
构造一个对象的过程是通过泛型实现的，以便它能够用于对同一对象创建不同的表示形式。

模式中包括的类

- 生成器类（ Builder）提供一个接口用于创建产品的各个组成部件。具体生成器（Concrete Builder）提供此接的实现。
- 具体生成器（ ConcreteBuilder）会跟踪其所创建对象的表现形式，并在创建对象的同时提供一个接口获取产品（Product）。
- 导演类（ Director）通过生成器提供的接口构造对象。产品类用于表示被构造的复杂对象。这包括对我们构建的所有类进行定义。

UML图



功能及应用场景

生成器模式隐藏了产品构建过程中的内部细节。各个生成器之间都是相互独立的。这提高了代码的模块化，并使其他的生成器更方便地创建对象。因为每个生成器都能够逐步创建对象，这让我们能够很好地对最终产品进行掌控。

单例设计模式

在应用程序的整个生命周期中，对象只有一个实例的时候，就会使用单例设计模式。单例类总是在第一次被访问时完成实例化，直至应用程序退出之前，都只会使用同一个实例。单一实例创建策略：我们通过限制构造函数（通过设置其为私有）从而限制单例类的实例化。之后在定义类时包含一个该类的静态私有对象，以便创建单例类的实例。
在单例模式中，最棘手的部分是对单一实例的实现和管理。
在单例模式的定义过程中，有两点需要注意的地方：

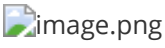
- 该类仅允许存在一个实例。
- 需要为该单一实例提供一个全局访问点。

单例模式中的主动实例化和被动实例化（饿汉、懒汉）
线程安全的单例：双重同步锁、静态变量、枚举

模式中包括的类

- 单例类

UML图



功能及应用场景

在应用程序的整个生命周期中，对象只有一个实例的时候，就会使用单例设计模式。

原型设计模式

相比于以往创建一个复杂对象总是费时费力，原型模式只需要复制现有的相似对象，并根据需要做适当修改。原型意味着使用克隆方法。克隆方法是一种复制对象的操作。克隆出的对象副本被初始化为调用克隆方法时原始对象的当前状态。这意味着对象的克隆避免了创建新对象。如果创建一个新对象的开销很大，而且有可能引起资源紧张时，我们就克隆对象。

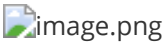
- 浅层复制：当原始对象变化时，新对象也跟着改变。这主要是因为浅层复制并没有实际复制新的对象，而只是对原有对象的一个引用。
- 深层复制：当原始对象变化时，新对象不受影响，因为原始对象所包含的所有参数、对象和引用在复制新对象的过程中都建立了新的拷贝。

使用克隆方法来复制对象时，具体是使用浅层复制还是深层复制是由业务需求来决定的。在使用原型模式时，使用克隆方法来复制对象仅仅是一个设计上的决策。克隆方法对于原型模式来说并不是强制性的最佳选择。

模式中包括的类

- 客户端（Client）：通过调用原型类的克隆操作创建一个新对象。
- 原型类（Prototype）：声明一个接口用于克隆自己。
- 具体原型类（Concrete Prototype）：实现克隆自己的操作。

UML图



功能及应用场景

1. 当一个系统应该独立于其产品的创建、组合和表示。
2. 当需要实例化的类是在运行时定义的，例如动态加载，或避免建立一个平行于产品类继承层次的工厂类继承层次时。
3. 当一个类的实例仅可以拥有若干不同的状态组合中的一个时。使用原型模式建立相应数量的原型和克隆方法，会比每次都手动实例化类并配置相应状态更加方便。

主要难点：

- 每个原型类的子类都必须实现克隆操作。这实现起来可能有难度。例如，当类已经存在的时候添加克隆方法可能比较困难。
- 对象内部包含其他不支持克隆的对象或具有循环引用的对象时，实现克隆方法会比较困难。

优点：

- 原型模式意味着使用克隆方法。克隆方法是一种复制对象的操作。相比于耗时的复制对象创建过程，原型模式仅复制类似的现有对象，再根据需要对复制出的副本进行修改。
- 客户端可以在运行时添加或移除原型对象。
- 通过各种参数来定义新对象：高度动态的系统允许我们通过使用对象组合来定义新的特征，例如为对象变量指定相应的参数值，而不是重新定义一个类。我们通过实例化现有类可以有效地定义新类型的对象，并为客户端对象注册原型实例。客户端可以通过向原型类委派某个责任而使其具有新的特征。这种设计允许用户无须大量编程就能轻松定义新的类。事实上，克隆一个原型本质上是类似于类的实例化的。但原型模式能够大大降低系统所需的类的数量。

副作用：

- 使用原型模式，我们可以根据需要通过对象克隆来实现运行时对象的添加和删除。我们可以根据程序运行情况在运行时修改类的内部数据表示形式。
- 在Java中实现原型模式的一大困难是如果这些类已经存在，我们未必能够通过添加所需要的克隆方法或深层克隆方法对类进行修改。此外，那些与其他类具有循环引用关系的类并不能真正实现克隆。
- 需要在这些类中具有足够的数据访问权限或方法，以便在克隆完成后对相应的数据进行修改。这可能需要在这些原型类中添加相应的数据访问方法，以便我们对类完成克隆之后可以修改数据。

结构型设计模式

结构型模式主要描述如何将对象和类组合在一起以组成更复杂的结构。在软件工程中结构型模式是用于帮助设计人员通过简单的方式来识别和实现对象之间关系的设计模式。结构型模式会以组的形式组织程序。这种划分形式使代码更加清晰，维护更加简便。结构型模式用于代码和对象的结构组织。
结构型模式会以组的形式组织程序。这种划分形式使代码更加清晰，维护更加简便。
结构型模式又分为以下子类别：

1. 对象结构型模式：用于对象之间相互关联与组织，以便形成更大、更复杂的结构。
2. 类结构型模式：用于实现基于继承的代码抽象，并且会介绍如何通过该模式提供更有用的程序接口。

具体包括：

对象结构型模式	类结构型模式
桥接模式	类适配器模式
组合模式	
装饰模式	
门面模式	
享元模式	
对象适配器模式	
代理模式	

1. 组合模式：它能够为客户端处理各种复杂和灵活的树状结构。这些树结构可以由各种不同类型的容器和叶节点组成，其深度或组合形式能够在运行时调整或确定。
2. 装饰模式：允许我们通过附加新的功能或修改现有功能，在运行时动态地修改对象。
3. 门面模式：允许我们为客户端创建一个统一的接口以访问不同子系统的不同接口，从而简化客户端。
4. 享元模式：客户端调用类时会在运行时创建大量对象，该模式会重新设计类以优化内存开销。
5. 代理模式：为其他对象提供一种代理以控制对这个对象的访问。这种模式的目的是一个对象不适合或者不能直接引用另一个对象，简化客户端并实现对象访问，同时避免任何副作用。
6. 适配器模式：允许我们为一个已有的类提供一个新的接口，并在客户端请求不同接口时实现类的重用。
7. 桥接模式：允许我们将类与其接口相互解耦。允许类及其接口随着时间相互独立变化，增加类重用的次数，提高后续可扩展性。它也允许运行时对接口的不同实现方式动态切换，使代码更加灵活。

适配器设计模式

软件适配器的工作原理也和插座适配器完全一样。我们也经常需要在程序中使用到不同的类或模块。假设有一段代码写得很烂，如果我们直接将这些代码集成到程序中，会将现有的代码搞乱。但是我们又不得不调用这段代码，因为我们需要实现相关的功能，而从头写起会耽误很多宝贵的时间。这时的最佳实践就是编写适配器，并将所需要的代码包装进去。这样我们就能够使用自定义的接口，从而降低对外部代码的依赖。
适配器模式会将现有接口转换为新的接口，已实现对应用程序中不相关的类的兼容性和可重用性的目标。适配器模式也被称为包装模式。适配器模式能够帮助那些因为接口不兼容而无法一起工作的类，以便它们能够一同工作。
适配器模式也负责将数据转换成适当的形式。当客户端在接口中指定了其对数据格式的要求时，我们通常可以创建新的类以实现现有类的接口和子类。这种实现方式也会通过创建类适配器，实现对客户端调用命和现有类中被调用方法之间接口的转换。

模式中包括的类

- 客户端（Client）调用目标类的类或程序。
- 目标类（Target）客户端想要使用的接口。
- 适配对象类（Adapetee）需要进行适配的类或对象。
- 适配器类（ Adapter）按照目标类接口的要求对适配对象接口实现接口形式的适配转换。
- request方法：客户端想要执行的操作。
- specificRequest方法：适配对象中能够完成 request方法功能的实现。

UML图

功能及应用场景

在具体实践上，有两种实际应用适配器模式的方法：

1. 使用继承［类适配器］
2. 使用关联［对象适配器］

应用场景：

- 我们想要使用现有的类，但它的接口不符合我们的需要。
- 我们想要创建一个可重用的类，能够与一些无关的类或不可预见的类进行协作，同时这个类无须具有兼容的接口。
- （仅适用于对象适配器）我们需要使用多个已经存在的子类，而我们为每一个子类都做接口适配显然是不切实际的。使用对象适配器可以直接适配其父类的接口。

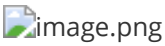
桥接设计模式

桥接模式是结构型模式中的另一个典型模式。桥接模式用于将类的接口与接口的实现相互解耦。这样做提高了系统的灵活性使得接口和实现两者均可独立变化。
举一个例子，让我们想一下家用电器及其开关。例如，风扇的开关。开关是电器的控制接口，而一旦闭合开关，实际让风扇运转的是风扇电机。
所以，在这个示例中，开关和风扇之间是彼此独立的。如果我们将开关接到电灯泡的供电线路上，那么我们还可以选用其他开关来控制风扇。

模式中包括的类

- 抽象化对象（Abstraction）桥接设计模式的核心，并定义了关键症结所在。包含对实现化对象的引用。
- 扩充抽象化对象（RefinedAbstraction）扩展抽象化对象，并将抽象化对象细化到新的层次。对实现化对象隐藏细节元素。
- 实现化对象（Implementor）该接口比抽象化对象的层次更高。只对基本操作进行定义。
- 具体实现化对象（ConcreteImplementor）通过提供具体实现来执行实现化对象的具体功能。

UML图



功能及应用场景

桥接模式主要适用于系统的多个维度上都经常发生变化的情况。桥接模式能够将不同的抽象维度进行衔接。通过桥接模式，抽象化对象和实现化对象不会在编译时进行绑定，而能够在各自的类被调用时独立扩展。
当你经常需要在运行时在多个实现之间进行切换时，桥接模式也非常有用。

组合设计模式

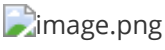
在大部分系统开发过程中，程序员都会遇到某个组件既可以是独立的个体对象，也能够作为对象集合的情况。组合模式就用于此类情况的设计。简单来说，组合模式是一组对象的集合，而这组对象中的每一个对象本身也是一个组合模式构成的对象，或者只是一个原始对象。
组合模式中存在着一个树形结构，并且在该结构中的分支节点和叶节点上都能够执行相同的操作。树形结构中每一个分支节点都包含子节点的类（能继承出叶节点和分支节点），这样的分支节点本身就是一

个组合模式构成的节点。树形结构中的叶子节点仅是一个原始对象，其没有子节点（不能继承出叶节点和分支节点）。组合模式的子类（下一级节点）可以是叶子节点或其他组合模式。

模式中包括的类

- 组件对象：（Component，结构）
    - 组件对象在整个继承结构的最顶端。它是对组合模式的抽象。
    - 它声明了组合模式中的对象接口。
    - 可以选择性地定义一个接口，以便对递归结构中组件的父类进行访问，并在需要的时候实现该接口。
- 叶子节点：（Leaf，原始对象）
    - 树形结构的末端且不会再有子节点。
    - 定义了组合结构中单个对象的行为。
- 分支节点类：（Composite，组）
    - 包含了子组件并为它们定义行为。
    - 实现子节点的相关操作。

UML图



功能及应用场景

- 当对象的集合需要采用与单个对象相同的处理方式时。
- 操纵单个对象的方式与操纵一组对象的方式类似时。
- 注意存在能够组合的递归结构或树形结构。
- 客户端能够通过组件对象访问整个继承结构，而它们却不会知道自己所处理的是叶子节点还是分支节点。

组合模式的目的是能够使独立对象（单个分支节点或叶子节点）和对象集合（子树）都能够以同样的方式组织起来。组合模式中所有的对象都来自于其本身（成为一种嵌套结构）。组合模式允许我们使用递归的方式将类似的对象组合成一种树形结构，来实现复杂结构对象的构建。

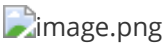
装饰者设计模式

装饰设计模式用来在运行时扩展或修改一个实例的功能。一般来说，继承可以扩展类的功能（用于类的所有实例）。但与继承不同的是，通过装饰模式，我们可以选择一个类的某个对象，并对其进行修改，而不会影响这个类中其他的实例。继承会直接为类增加功能，而装饰模式则会通过将对象与其他对象进行包装的方式将功能添加到类。

模式中包括的类

- 抽象组件（Component）给出一个抽象接口，用于能够动态添加功能的对象。
- 具体组件（Concrete Component）定义一个实现组件接口的对象。这是实际需要加以装饰的对象，但其对装饰的过程一无所知。

UML图



功能及应用场景

装饰设计模式用来在运行时扩展或修改一个实例的功能。一般来说，继承可以扩展类的功能（用于类的所有实例）。但与继承不同的是，通过装饰模式，我们可以选择一个类的某个对象，并对其进行修改，而不会影响这个类中其他的实例。继承会直接为类增加功能，而装饰模式则会通过将对象与其他对象进行包装的方式将功能添加到类。

门面设计模式

许多业务流程都会涉及复杂的业务类操作。由于流程很复杂，所以其涉及了多个业务对象，这往往会导致各个类之间的紧密耦合，从而降低系统的灵活性和设计的清晰度。底层业务组件间的复杂关系会使客户端的代码编写变得很困难。
门面模式简化了到复杂系统的外部接口。为此它会对所有的类进行整合，并构建一个复杂系统的子系统。
门面模式能够将用户与系统内部复杂的细节相互屏蔽，并只为用户提供简化后的更容易使用的外部接口。同时它也将系统内部代码与接口子系统的代码相互解耦，以便修改和升级系统代码。
相比于其他设计模式，门面模式更注重实现代码的解耦。它所强调的是代码设计中很重要的一点，即代码抽象。通过提供一个简单的接口并隐藏其后的复杂性，从而实现抽象。
在这种方式下，代码的实现完全交由门面层处理。客户端只会与一个接口交互，同时也只有和这个接口交互的权限。这样就能隐藏全部系统的复杂性。总而言之，门面模式通过提供一个简单的接口为客户端简化了与复杂系统的交互。
从另一方面看，门面模式也保证了能够在不修改客户端代码的情况下对具体实现方法进行修改。

模式中包括的类

- 门面层（Facade）：它知道子系统内各个类的具体功能，并将客户端请求转换成对系统内部对象的调用。
- 系统内部类（ComplicatedClass）：这些类会实现系统功能，处理门面层对象分配的各项工作任务。它们本身并不知道门面层的存在，也没有对其进行任何的引用。

UML图


image.png

功能及应用场景

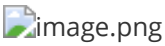
- 想要为一个复杂的子系统提供一个简单接口。子系统随着其自身的发展往往变得越来越复杂。它们应用的大多数的设计模式会导致类的数量更多、代码段更小。这使得该子系统可重用更好，也更容易进行自定义。而对于某些无法自定义的客户端来说，它也变得难以使用。门面层可以提供对大多数客户端来说足够好的简化的调用接口。只有极少数高度定制化的客户端需要直接调用门面层之后的底层代码。
- 在客户端和抽象层的实现类之间存在大量的依赖关系。引入一个门面层能够将客户端的子系统与其他子系统进行解耦，从而促进子系统的独立性和可移植性。
- 你想要为你的子系统增加层级。使用一个门面层对每个子系统级别分别定义一个入口点。如果子系统之间存在依赖关系，那么你可以通过令这些子系统之间的交互全部需要经由门面层来简化彼此的依赖关系。

代理设计模式

根据目的不同，有各种不同类型的代理。例如，有保护性代理，控制对某个对象的访问权限；有虚拟代理，处理开销过大而难以创建的对象，并通过远程访问控制来访问远程对象。

模式中包括的类

UML图



功能及应用场景

代理模式主要用于当我们需要用一个简单对象来表示一个复杂对象的情形。如果创建对象的开销很大，那么可以推迟其创建，并使用一个简单对象来代理其功能直到必须立即创建的时候。这个简单对象就可以称为复杂对象的代理。

享元设计模式

享元模式能够减少用于创建和操作大量相似的细碎对象所花费的成本。享元模式主要用在需要创建大量类似性质的对象时。大量的对象会消耗高内存，享元模式给出了一个解决方案，即通过共享对象来减少内存负载它的具体实现则是根据对象属性将对象分成两种类型：内蕴状态和外蕴状态。
共享是享元模式的关键。

模式中包括的类

- 抽象享元角色（ Flyweight）声明一个为具体享元角色规定了必须实现的接口，而外蕴状态就是以参数的形式通过此方法传入。
- 具体享元角色（ Concrete Flyweight）实现享元模式接口，并存储内蕴状态。具体享元角色必须是共享的。具体享元角色必须保持其内蕴状态不变，并且能够操纵外蕴状态。
- 享元工厂角色（FlyweightFactory）负责创建和管理享元角色。此外，该工厂确保了享元角色的共享。工厂维护了不同的享元对象池，并负责在对象创建完成时从对象池返回对象，以及向对象池添加对象。
- 客户端（Client）维护对所有享元对象的引用，而且需要存储和计算对应的外蕴状态。

UML图

功能及应用场景

当我们选择享元模式的时候，需要考虑以下因素：

- 需要创建大量的对象时。
- 由于对象众多，内存的开销是一个制约因素。
- 大多数对象属性可以分为内蕴状态和外蕴状态。
- 应用程序需要使用多种对象，且创建对象后需要多次重复使用。
- 外蕴状态最好是通过计算得到的，而不需要进行存储。

行为型模式

行为型模式是一类主要关注对象间相互通信（交互）的设计模式。这些对象之间的相互作用既能保证对象间能够交换数据，同时对象间仍然能够保持松耦合。
紧耦合一般会发生在一组紧密关联（相互依赖）的类之间。在面向对象的设计过程中，耦合引用的数量和设计过程中类与类之间的相互依赖是成正比的。用通俗的话讲，就是当一个类变化的时候，有多少可

能需要同时修改其他类呢?
松耦合是软件架构设计的关键。在行为型模式中，功能实现与调用该实现的客户端之间应该是松耦合的，以避免硬编码和依赖性。
行为型模式处理不同的对象之间的通信关系，为其提供基本的通信方式，并提供实现这种通信方式的最常用、最灵活的解决方案。
行为型模式描述的不仅是类或对象的模式，同时也包括了它们之间的通信模式。行为型模式能够用来避免硬编码和依赖性。
行为型模式又分为以下子类别：

1. 对象行为型模式：对象行为型模式使用对象组合而非继承。描述一组对象如何合作执行部分任务，而单个对象无法执行这些任务。
2. 类行为型模式：类行为型模式使用继承而不是对象组合来描述算法和流程控制。

具体包括：

- 职责链模式（COR）：在一系列对象链之间传递请求的方法。
- 命令模式：命令模式主要用于在需要向对象发出请求的情况，发出请求的对象无须了解请求的操作内容，也无须了解请求的实际接收对象。
- 解释器模式：解释器提供了在代码中使用特定语言的一种方法。解释器模式就是一种用于在程序中解析特定语言的设计模式。
- 迭代器模式：迭代器用于顺序访问集合（组合）对象中的元素，而无须了解其内部结构。
- 中介者模式：定义简单的类间通信。
- 备忘录模式：捕获和恢复对象的内部状态。
- 观察者模式：一种通知多个类进行改变的方式。
- 状态模式：当一个对象状态改变时改变其功能。
- 策略模式：在类中进行算法封装。
- 模板方法模式：将算法中的部分步骤延迟到子类中进行计算。
- 访问者模式：在不改变类的条件下为该类定义一个新的操作。

对象行为型模式	类行为型模式
职责链模式	
解释器模式	
命令模式	模板方法模式
迭代器模式	
中介者模式	
备忘录模式	
观察者模式	
状态模式	
策略模式	
访问者模式	

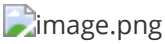
职责链设计模式

在职责链模式中，由发送端发送一个请求到一个对象链中，链中的对象自行处理请求。如果链中的对象决定不响应请求，它会将请求转发给链中的下一个对象。
职责链的目的是通过特定设计对请求的发送者和接收者之间进行解耦。解耦是软件设计中很重要的一个方面。通过该设计模式能够使我们彻底地将发送者和接收者之间完全解耦。发送者是用于调用操作的对象，接收者是接收请求并执行相关操作的对象。通过解耦，发送者不需要关心接收者的接口。
在职责链模式中，职责是前后传递的。对于链中的对象，决定谁来响应请求的责任由整个链中左侧的对象来承担。这就像问答测验的时候传递问题一样。当提问者向一个人提问，如果他不知道答案，他就把问题传给下一个人，以此类推。当一个人回答了问题，问题就会停止向下传递。有时，也可能到达最后一个人时，还是没有人能回答问题。
我们能举出若干个职责链模式的例子：硬币分拣机、ATM取款机、Servlet过滤器和Java的异常处理机制。
在Java中，我们可以在catch语句中列出的异常序列时就抛出一个异常，catch列表从上到下逐条扫描。如果赶上第一个进行异常处理就可以立刻完成任务，否则责任转移到下一行，直到最后一行。

模式中包括的类

- 抽象处理者（Handler）：定义用于处理请求的接口。
- 具体处理者（ Concrete Handler）：它负责处理请求。如果它能够处理这样的要求就会自行处理，否则会将请求发送到下一个处理者。
- 客户端（Client）：将命令发送到职责链中第一个能够处理该请求的对象。

UML图



功能及应用场景

- 发送者并不知道在链中的哪个对象会响应请求。
- 职责链中的每一个对象都有责任决定是否对请求进行响应，如果这些对象有能力响应请求就会响应请求。
- 如果对象（或节点）决定向后传递请求它需要具有选择下一个节点和继续传递的能力。
- 也有可能没有任何一个节点能够响应请求（有些请求可能无法得到处理）
- 会在运行时确定哪些对象能够响应请求。

命令设计模式

命令模式（也称为行动模式、业务模式）是一个对象行为型模式。
这使我们能够实现发送者和接收者之间完全解耦。发送者是调用操作的对象，接收者是接收请求并执行特定操作的对象。通过解耦，发送者无须了解接收者的接口。在这里，请求的含义是需要被执行的命令。

模式中包括的类

- 抽象命令类（Command）：在类中对需要执行的命令接口进行声明。
- 具体命令类（ ConcreteCommand）：将接收者对象和行为之间进行绑定。它通过调用接收者中相应的操作实现 execute方法。
- 客户端（ Client）：客户端完成对命令对象的实例化并提供随后需要调用的方法的信息。
- 调用者（Invoker）：调用者决定合适的调用方法。
- 接收者（Receiver）：接收者是包含方法代码的类的一个实例。这意味着它知道如何处理一个请求并执行相应的操作。任何一个类都可以作为接收者。

UML图

功能及应用场景

- 通过参数化对象实现功能执行。命令是面向对象式的，而不是回调函数式的。
- 指定消息队列并在不同的时间执行请求一个命令对象可以有独立于原始请求的生命周期。如果一个请求的接收者可以由一个独立地址空间的方式来表示，那么你可以将请求对应的命令对象转换到不同的进程并在其中完成请求。
- 支持撤销。命令的执行操作可以作为状态进行存储，并在需要时实现命令撤销。命令接口必须增加一个unexecute操作，支持撤销之前命令调用的执行效果。执行的命令存储在命令的历史列表中。无限次数的撤销和重做是通过遍历这个列表并分别调用 unexecute和 execute来实现的。
- 支持日志记录变化，在系统崩溃的情况下使命令可以重新应用。通过增加load和 store操作命令接口参数，你可以保存一个持续变化的日志。从系统崩溃中恢复需要从磁盘重新加载日志命令和使用Execute作重新执行这些命令。
- 通过在原生操作基础上的高层操作构建系统。这样的结构在支持交易操作的信息系统中很常见。一个交易事务封装一组变化的数据。命令模式提供了一种交易模型。命令都有一个共同的接口，允许你使用相同的方式调用所有的交易。这种模式也使得它很容易与新的交易系统进行交互扩展。

注意事项：

- 目的：将请求封装为一个对象，从而使客户端可以将不同的请求、队列、日志请求及其他支持撤销的操作进行参数化。
- 发出请求的对象无须知道请求对应的操作或请求接收者的任何信息。
- 后果：
    - 将调用操作的对象和执行操作的对象之间对命令进行解耦。即调用者和接收者之间解耦。
    - 命令转换为一类对象。使其可以像其他对象那样进行操作和扩展。
    - 我们可以将命令组合成一个组合命令。一般来说，组合命令是一个组合模式的实例。
    - 很容易添加新的命令，因为我们无须改变现有的类。

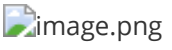
解释器设计模式

解释器模式是一种用于在程序中解析特定语法的设计模式。解释器模式是组合模式的一种应用。
对于特定的某种语言，解释器模式能够定义针对其语法表示形式的解释器，并实现对该语言语句的翻译和解释。

模式中包括的类

- 内容类（Context）：包含解释器的全局信息
- 表达式（AbstractExpression）：带有名叫 interpret抽象方法的抽象类。它会声明执行操作的接口。
- 终结符表达式（TerminalExpression）：就是带有终结符的表达式。
- 非终结符表达式（ NonterminalExpression）：在两个终结符表达式或非终结符表达式之间实现逻辑运算（与或运算）的表达式。
- 客户端（Client）：建立抽象树，并调用抽象树中的 interpret方法。

UML图



功能及应用场景

解释器模式的适用范围非常有限。我们可以说解释器模式仅仅用于需要进行正式语法解释的地方，但这些领域往往已经有了更好的标准的解决方法，因此，在实际使用中，并不会经常使用该模式。该模式可以用于解释使用了特定语法的表达式或者建立某个简单的规则引擎的时候。

迭代器设计模式

迭代器模式也是一种行为型模式。迭代器模式允许对一组对象元素的遍历（也叫收集）以完成功能实现。

模式中包括的类

- 迭代器（Iterator）：它会实现一个用于定义迭代器的抽象迭代器接口。
- 具体迭代器（Concretel）：这是迭代器的实现（实现迭代器接口）。
- 抽象容器（Container）：这是用于定义聚合关系的接口。
- 具体容器（ConcreteContainer）：一个聚合关系的实现。

UML图

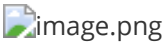


image.png

功能及应用场景

- 需要访问一个聚合（也称为容器）对象的内容，而无须了解其内部表示。-
- 支持对聚合对象的多种遍历方式。
- 为遍历不同的聚合结构提供统一的接口（即支持多态迭代）。
- 迭代器模式允许我们访问集合对象中的内容，而无须暴露其内部数据结构。
- 支持多个迭代器同时遍历集合对象。这意味着我们可以对相同的集合创建多个独立的迭代器。
- 为遍历不同的集合提供统一的接口。

中介者设计模式

中介者模式主要是关于数据交互的设计模式。中介者设计模式很容易理解，却难以实现。该模式的核心是一个中介者对象，负责协调一系列对象之间一系列不同的数据请求。这一系列对象称为同事类。
同事类会让中介者知道它们会发生变化这样中介者负责处理变化对不同对象之间交互的影响。

模式中包括的类

- 中介者接口（Mediator）：它定义了一个接口来实现同事类对象之间的沟通。
- 具体中介者（ConcreteMediator）：它知道各个同事类，并和这些同事类保持相互引用。它实现了与同事类之间的通信和消息传递。
- 同事类（Colleague）：这些类保存了对中介者的引用。无论它们想和任何其他同事类进行交互，都必须通过与中介类通信来实现。

UML图


image.png

功能及应用场景

- 一组对象使用了标准的通信方式，但整体通信的连接都非常复杂。由此产生的相互依赖的结果导致系统难以结构化，也很难理解。
- 由于对象之间的通信和相互引用，导致对象难以重用。
- 分布在多个类中的行为能够被统一定制化，而无须创建过多的子类。

需要注意的问题：
实际使用中介者模式的时候，反而会让问题变得越来越复杂。所以最佳的实践是仅让中介者类负责对象之间的通信部分。

- 定义一个对象来负责一系列对象之间的交互。
- 同事类发送和接收请求都需要通过中介者。

功能：

- 它对同事类进行解耦。中介类实现了同事类之间的松耦合。你可以相互独立地对不同的同事类进行修改和重用。
- 它简化了对象协议。中介者取代了许多交互作用，而实现了与多个同事类之间一对多的通信方式。一对多关系更容易理解、维护和扩展。
- 它集中了控制。中介者模式在中介者中集成了系统交互的复杂性。因此通过中介封装协议之后，它会比任何单一的同事类都更为复杂。这会使中介者作为一个整体也很难维护。
- 门面模式不同于中介者模式的是，它抽象了对象的子系统以提供一个更方便的接口。该种抽象是单向的。也就是说，门面对象会向子系统中的各个类发出请求，反之则不会。相比之下，中介者模式更像是同事类对象之间通过中介者的合作行为，系统的交互都是多向的。
- 当各个同事类只和一个中介者对象交互时，没有必要再去定义一个抽象的中介者类。抽象中介者只用于多个同事类通过多个抽象中介者的子类进行交互的情况，反之则不同。

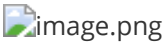
备忘录设计模式

我们每天至少会使用一次这种模式。备忘录模式提供了一种使对象恢复到其以前状态的能力（通过回滚撤销）。备忘录模式是通过两个对象实现的：发起者和管理者。发起者是具有内部状态的某个对象。管理者则会对发起者执行一些操作，并实现撤销更改。

模式中包括的类

- 发起者（Originator）：发起者知道如何保存自己。这是我们想要保存状态的类。
- 管理者（Caretaker）：管理者是用于管理发起者进行状态保存的对象，具体处理发起者何时、如何、为何对状态进行存储。管理员应能够对发起者进行修改，同时也能够撤销这些修改。
- 备忘录（Memento）：备忘录会保存发起人的状态信息，而这些状态不能由管理者修改。

UML图

 image.png

功能及应用场景

当我们在实际应用中需要提供撤销机制，当一个对象有可能需要在后续操作中恢复其内部状态时，就需要使用备忘录模式。结合本设计模式实现对象状态序列化，能够使其易于保存对象的状态并进行状态回滚。
当一个对象状态的快照必须被存储，且在后续操作过程中需要被恢复时，就可以使用备忘录模式。

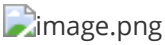
观察者设计模式

在观察者模式中，一种叫作被观察者的对象维护了观察者对象的集合。当被观察者对象变化时，它会通知观察者。
在被观察者对象所维护的观察者集合中能够添加或删除观察者。被观察者的状态变化能够传递给观察者。这样观察者能够根据被观察者的状态变化做出相应的改变。

模式中包括的类

- 被观察者（Listener）：定义了向客户端添加和移除观察者操作的接口或抽象类。
- 具体被观察者（ ConcreteListener）：具体被观察者类。它维护了对象的状态，并在当其状态改变时通知各个观察者。
- 观察者（Observer）：定义了用于通知对象的接口或抽象类。
- 具体观察者（ ConcreteObserver）：具体实现了观察者。

UML图



功能及应用场景

- 当一个对象的改变需要其他对象同时改变，而我们并不知道需要有多少个对象一起改变时。
- 当一个对象必须通知其他对象，而无须了解这些对象是谁时。
- 当一个抽象包含有两个方面，其中一个依赖于另一个。将这些方面封装成独立的对象，以便我们独立改变和重复使用它们时。

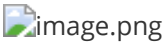
状态设计模式

状态模式是一种行为型模式。状态模式背后的理念是根据其状态变化来改变对象的行为。状态模式允许对象根据内部状态（内容类）实现不同的行为。内容类可以具有大量的内部状态，每当对内容类调用request方法时，消息就被委托给状态类进行处理。
状态类接口定义了一个对所有具体状态类都有效的通用接口，并在其中封装了与特定状态相关的所有操作。具体状态类对请求提供各自具体的实现。当内容类的状态变化时，那么与之关联的具体状态类也会发生一定相应的改变。

模式中包括的类

- 内容类（ Context）：内容类主要用于状态模式的客户端。客户端并不直接访问对象的状态。内容类拥有一个具体的状态对象并根据其当前状态提供所需实现的行为。
- 抽象状态类（ State）：这个抽象类是所有具体状态类的基类。状态类定义了一个通用接口。内容类对象能够通过使用该接口实现对不同功能的改变。在状态类及其子类的各个条目或属性中，本身并没有任何的状态。
- 具体状态类（ ConcreteState）：具体状态类根据内容类所提供的状态实现真正的功能改变。每个状态类所提供的行为都适用于内容类对象的某一个状态。它们也包含着由内容类状态变化所下发的指令。

UML图



功能及应用场景

- 状态模式为对象提供了一个清晰的状态表示。
- 它允许一个对象在运行时部分清晰明了地改变其类型。

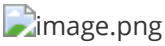
策略设计模式

策略模式主要用于需要使用不同的算法来处理不同的数据（对象）时。这意味着策略模式定义了一系列算法，并且使其可以替换使用。策略模式是一种可以在运行时选择算法的设计模式。
本模式可以使算法独立于调用算法的客户端。策略模式也称为政策模式。在使用多种不同的算法（每种算法都可以对应一个单独的类，而每个类的功能又各不相同）时可以运用策略模式。

模式中包括的类

- 抽象策略类（Strategy）：定义一个所有算法都支持的通用接口。内容类会使用这个接口来调用由具体策略类定义的各个算法。
- 具体策略类（ ContreteStrategy）：每个具体策略类都会实现一个相应的算法。
- 内容类（ Context）：包含一个对策略对象的引用。它可以定义一个用于策略类访问内容类数据的接口。内容类对象包含了对将要使用的具体策略对象的引用。当需要进行特定操作时，会从对应的策略类对象中运行相应的算法。内容类本身觉察不到策略类的执行。如果有必要的话，还可以定义专用的对象来传递从内容类对象到策略类的数据。内容类对象接收来自客户端的请求，并将其委托给策略类对象。通常具体策略类是由客户端创建，并传递给内容类。从这一点来讲，客户端仅与内容类进行交互。

UML图



功能及应用场景

当我们有多种不同的算法可供选择（每种算法都可以对应一个单独的类，而每个类的功能又各不相同）时，可以应用策略模式。策略模式会定义一组算法并能够使其相互替代使用。

模板方法设计模式

模板方法会定义算法的各个执行步骤。算法的一个或多个步骤可以由子类通过重写来实现，同时保证算法的完整性并能够实现多种不同的功能。
类行为型模式使用继承来实现模式的功能。在模板方法模式中，会有一个方法（ Template method方法）来定义算法的各个步骤。这些步骤（即方法）的具体实现会放到子类中。也就是说，在模板方法中定义了特定算法，但该算法的具体步骤仍然需要通过子类来定义。模板方法会由一个抽象类来实现在这个抽象类中还会声明该算法的各个步骤（方法），最后将其具体实现的方法声明实现为抽象类的子类。

模式中包括的类

- 抽象类（ AbstractClass）：定义了算法的抽象操作，并交由具体的子类完成这些操作的具体实现。它实现了一个模板方法，它该方法包含了算法的各个步骤。该模板方法还会在抽象类中定义各个相应步骤的基本操作。
- 具体类（ ConcreteClass）：他们通过执行基本操作来实现算法类的具体步骤。当调用一个具体类时，模板方法代码会从基类执行，而模板方法所使用的各个方法由派生类实现和调用。

UML图

功能及应用场景

应用场景：

- 当一个算法的功能需要能够改变，并通过在子类中对功能重写来实现这种改变。
- 当我们要避免代码重复时，能够在子类中实现算法不同的变化。
- 在一开始，模板方法可能不是一个显而易见的选择。最明显的现象会是当我们发现几乎完全一样的类在执行某些类似的逻辑。这时，我们就应该考虑使用模板方法模式来清理现有代码。

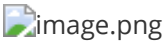
访问者设计模式

访问者模式用来简化对象相关操作的分组。这些操作是由访问者来执行的，而不是把这些代码放在被访问的类中。由于访问的操作是由访问者执行的，而不是由被访问的类，这样执行操作的代码会集中在访问者中，而不是分散在对象分组中。这为代码提供了更好的可维护性。访问者模式也避免了使用instanceof运算符对相似的类执行计算。

模式中包括的类

- 访问者（Visitor）：包括一个接口或抽象类，用于声明在所有类型的可访问对象中访问哪些操作。通常操作的名称是相同的，而是由该方法的参数来区分不同的操作。由输入对象类型来决定访问该方法中的哪一个。
- 具体访问者（Concrete Visitor）：用于实现各个类型的访问者和各个类型的访问方法。它在抽象访问者中进行声明，并各自独立实现。每一个具体访问者会负责实现不同的功能。当定义一个新的访问者时，只需要将其传递给对象结构即可。
- 元素类（Element）：一个抽象对象用于声明所接受的操作。它是一个入口点，能够允许哪一类访问者对象访问。在集合中的每个对象都需要实现该抽象对象，以便相应访问者能够实现对其进行访问。
- 具体元素类（Concrete Element）：这些类实现了抽象元素类的接口或类，并定义了所接受的操作。通过其可接受的操作，能够将访问者对象传递给该对象。
- 结构对象（ObjectStruture）：这是一个包含了所有可访问对象的类。它提供了一种机制来遍历所有元素。这种结构不一定是一个集合，也可以是一个极其复杂的结构，如组合对象。

UML图



功能及应用场景

在 visitCollection()方法中，我们调用 Visitable.accept(this)来实现对正确的访问者方法进行调用。这叫作双重分派。访问者调用元素类中的方法，又会回到对访问者类中进行调用。

模式问题：

在使用访问者模式的情况下，要想添加新的具体元素（数据结构）会更加困难。添加一个ConcreteElement会涉及向访问者接口添加新的操作和在每一个具体访问者实现中添加对应的实现。**访问者模式更适用于对象结构非常稳定，而对象的操作却需要经常变化的情况下。**

访问者模式只提供处理每种数据类型的方法，并且让数据对象确定调用哪个方法。由于数据对象本质上都知道其自身的类型，所以在访问者模式中算法决定所调用的方法所起到的作用是微不足道的。

因此，数据的整体处理包括对数据对象的分发以及通过对适当的访问者处理方法的二次分发。这就叫作双重分派。

使用访问者模式的一个主要优点是，对于在我们的数据结构中添加需要执行的新的操作来说，是非常容易的。我们所要做的就是创建一个新的访问者，并定义相应的操作。

访问者模式的主要问题是，因为每个访问者需要有相应的方法来处理每一种可能的具体数据，那么一旦实现了访问者模式，其具体类的数量和类型就不能被轻易改变。