**设计模式**

# 概述

## 描述在软件合集过程中一些不断重复发生的问题以及该问题的解决方案，特性普遍性，反复使用

# 本质/意义

## 面向对象设计原则的实际运用，对类的封装性，继承性和多态性以及类的关联关系和组合关系的充分理解

# 优点

## 1、提高思维能力，编程能力，设计能力

## 2、程序设计更加标准化，代码工程化，缩短开发工期

## 3、代码重用性提高，可读性提高，灵活性高，维护性高。

# 学习要素

## 上述优点的体现

## 应用问题

## 解决方案

## 效果

### 模式优缺点，灵活性，扩展性，可移植性，时间空间等等的权衡

## 带着要素学习

# 原则

## 开闭原则

### 需求改变时候，在不修改软件实体的源代码或者二进制代码情况下，可以拓展模块功能，使其满足需求

### 抽象约束，封装变化--扩展开放，修改关闭

### 里氏替换原则

* 子类拓展父类，但是不能修改父类
* 如果违背了，就取消原来的继承关系，重新设计它们之间的关系

## 依赖倒置原则

### 高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象

### 实现

* 1、每个类尽量提供接口或抽象类，或者两者都具备
* 2、变量声明类型尽量是接口或者是抽象类
* 3、任何类都不应该从具体类派生
* 4、使用继承时尽量遵循里氏替换原则

## 单一职责原则

### 单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分

### 实现

* 单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，再封装到不同的类或模块中

## 接口隔离职责

### 客户端不应该被迫依赖于它不使用的方法。该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上

### 实现

* 1、接口尽量小，但有限度，一个接口服务一个子模块或者业务逻辑
* 2、只提供调用者需要的方法，屏蔽不需要的方法
* 3、了解环境，每个项目有固定的环境因素，拆分的逻辑也不同
* 4、提高内聚，减少为对外交互

## 迪米特法则

### 如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性

### 实现

* 1、类的划分，创建弱耦合的类
* 2、在类的结构设计上，尽量降低成员的访问权限
* 3、在类的设计上，优先考虑将一个类设计成不变类
* 4、在对其他类的引用上，引用次数降低
* 5、不直接暴露属性成员，而是应该提供相应的访问器（get/set）
* 6、谨慎使用序列化

• 原因

• 降低实现灵活性

• 字节流编码改成导出API的一部分

• 变更最低限度访问域的准则

• 新老版本序列化不兼容

• 类的演变受到限制

• 流的唯一标识符，如果不设置自动生成(导致InvalidClassException)

• 增加bug，漏洞可能性

• 序列化反序列化用隐士构造器，会被显示构造器破坏约束条件

• 反序列化过程容易受到破坏和非法访问

• 测试负担增加

• 检查老版本序列化是否可行

• 深度学习

• 优点/注意/另外会单独整理

• 参考链接:https://my.oschina.net/u/3847203/blog/3010010

• 主要漏洞可参考:fastjson

## 合成复用原则

### 它要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。 如果要使用继承关系，则必须严格遵循里氏替换原则

### 继承复用

* 白箱复用

• 破坏类的封装

• 提高耦合度

• 限制复用的灵活性，父类继承来的实现是静态的

### 合成复用

* 黑箱复用

• 完善封装，成分对象细节不全暴露

• 新旧类耦合度低，复用依赖少

• 复用灵活度高，运行时能动态运行

### 重点:尽可能考虑组合复用，继承复用尽可能少

## 区别

### 1、单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离

### 2、单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。

# 创建型模式

## 特点

### 将对象的创建与使用分离，主要重心放在对象创建的流程上，尽可能保证创建流程与使用流程的隔离，更加关注与对象本身的细节

## 单例模式(Singleton)

### 定义

* 该类只有一个实例，且该类能自行创建这个实例的一种模式

### 解决问题

* 节省系统资源，保证内容一致性
* 类似:Windows任务管理器

### 特点

* 1、单例类只有一个实例对象
* 2、该单例对象必须由单例类自己创建
* 3、单例类对外提供一个访问单例的全局访问点

### 结构

* 1、单例类:包含一个实例且能自行创建这个实例的类
* 2、访问类:使用单例的类

### 实现

* 懒汉模式

• 调用的时候初始化一次

• 线程安全问题

• 双重检查锁

• 静态内部类

* 饿汉模式

• 类加载就初始化一次

• 线程安全

* 比较

### 场景

* 实践案例

• 全局唯一计数器

• 访问共同资源

• 日志，应用配置

• 线程池

### 拓展

* 多例模式

• 将多个单例放在ArrayList维护

## 原型模式(Prototype)

### 定义

* 把一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或者相似的新对象

### 解决问题

* 大量相同或者相似的对象的创建问题，减少用构造函数来创建对象

### 特点

* 1、指定创建对象的种类，高效创建
* 2、不关心创建细节

### 结构

* 1、抽象类原型:规定了具体原型对象必须实现的接口
* 2、具体原型类:实现抽象原型类的clone()方法，他是可复制对象
* 3、访问类:使用具体原型类中的clone()方法来复制新的对象

### 实现

* 原型类实现cloneable接口，重写clone方法

### 注意事项

* 该clone为浅度克隆，如果要深度clone则需要重写多层引用底层clone属性（或者用序列化解决）

• 深clone

• 浅clone

• 知识点扩展学习

### 场景

* 全班三十个同学的成绩单类，cloen30份
* 保护性拷贝
* 对象相似，但是属性不同
* 创建对象比较复杂，复制简单

### 拓展

* PrototypeManager 做一个原型管理器做功能拓展
* 安卓端--Intent
* 实践

## 工厂模式(Factory Method)

### 定义

* 定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂中

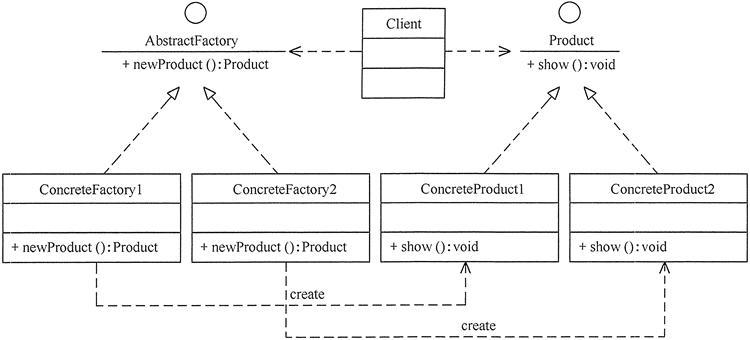
### 特点

* 1、简单工厂并不属于设计模式，新增产品时候会违背"开闭原则"，需要将简单工厂模式抽象才能满足开闭原则
* 2、用户只需要知道具体工厂的名称就可得到所想要的产品，不需要知道具体创建过程
* 3、新增产品的时候只需要添加具体产品类和具体工厂类，不需要对原工厂进行加工
* 缺点:为了满足开闭原则会导致一定程度的系统复杂度

### 结构

* 1、抽象工厂(Abstract Factory):提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法newProduct()来创建产品。
* 2、具体工厂(ConcreteFactory):主要是实现抽象工厂中的抽象方法，完成具体产品的创建
* 3、抽象产品(Product):定义了产品规范，描述了产品的主要特性和功能。
* 4、具体产品(ConcreteProduct):实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应

### 实现



### 场景

* 客户只关心或只知道工厂名，不知道具体的产品名
* 实践案例

• hibernate里通过sessionFactory创建session、通过代理方式生成ws客户端时，通过工厂构建报文中格式化数据的对象

### 拓展

* 当产品不多且不会增加，一个具体工厂能够完成就不需要抽象工厂去做退化成简单工厂

## 抽象工厂(AbstractFactory)

### 定义

* 为访问类提供一个创建一组相关或者相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构

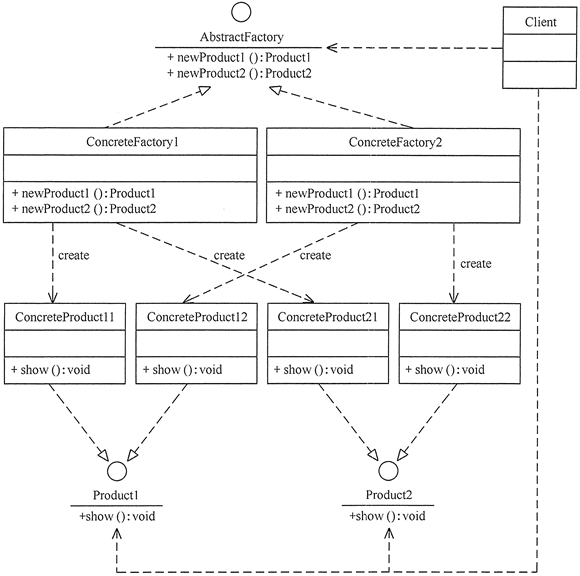
### 特点

* 1、可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理
* 2、新增加一个产品族的时候，满足开闭原则(产品族:产品等级再向上抽象一类产品等级的一个集群)
* 缺点:新增加产品的时候，所有工厂类都需要修改

### 结构

* 注意:虽然该要素还是同工厂方法一样，但是抽象工厂中方法个数不同，抽象产品的个数也不同
* 1、抽象工厂(Abstract Factory):提供了创建产品的接口，它包含多个创建产品的方法 new Product(),可以创建多个不同等级的产品
* 2、具体工厂(Concrete Factory):主要是实现多个抽象方法，完成具体产品的创建。
* 3、抽象产品(Product):定义了产品规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品
* 4、具体产品(ConcreteProduct):实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间是多对一的关系

### 实现



### 场景

* 1、当需要创建的对象是一系列相互关联或者相互依赖的产品族
* 2、系统中有多个产品族，但每次只需要使用其中一种族产品
* 3、系统中提供了产品的类库，且所有产品的接口相同。
* 实践案例

• spring--AbstrictBeanFactory

### 拓展

* 开闭原则的不全满足性质:新增族只需要新增工厂，新增产品时候需要重新修改工厂

## 建造者模式(Builder)

### 定义

* 将一个复杂对象的构造和表示分离，构建过程可以创建不同的表示

### 解决问题

* 将产品整个概念弱化，分成多个组成部分，通过自搭配来构建产品

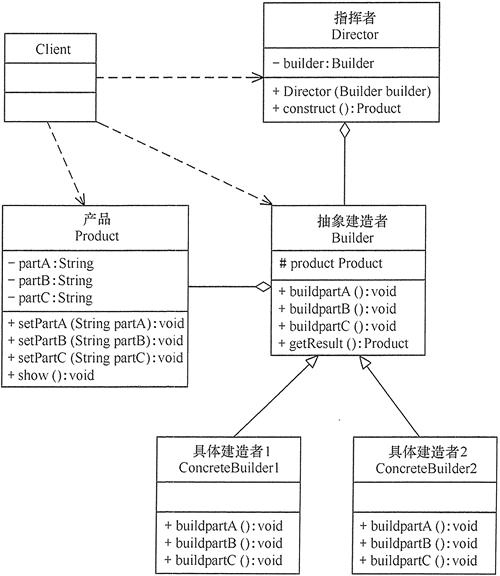
### 特点

* 1、各个具体建造者相对独立.有利于系统的拓展
* 2、客户端不需要知道产品内部组成细节，容易控制细节风险
* 3、产品的的组成部分必须相同 ，这限制了其使用范围
* 4、如果产品内部变化复杂，该模式会增加很多建造者类
* 优点
* 缺点

### 结构

* 1、产品角色(Product):包含多个组成部件的复杂对象，由具体建造者来创建其各个部件
* 2、抽象建造者(Builder):一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂的产品方法gerResult.
* 3、具体建造者(Concrete Builder):实现Builder接口，完成复杂产品的各个部件的具体创建方法.
* 4、指挥者(Director):它调用建造者对象中的部件与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品信息

### 实现



### 场景

* 1、创建的对象比较复杂，由多个部件构成，变化复杂
* 2、创建对象算法独立于组成部分以及他们的装配方式，构建过程能够解耦
* 实践案例

• Mybatis中的SqlSessionFactoryBuilder

• Lambok实现Builder pattern

### 拓展

* 当只有一个产品种类的时候能对该方法做什么改进

## 结合使用

### 设计一个任务同时使用以上模式(单例创建工厂，然后工厂配合建造者创建产品，最后在通过原型对产品进行clone)

## 关注点:

### 工厂模式注重产品创建链路而不关心产品本身，而建造者模式不关心产品创建，关心产品本身组装，可以结合使用

## 对比

# 结构型模式

## 将类或者对象按照某种布局(结构)组成更大的结构.

### 类结构行模式

* 采用继承机制来组织接口和类

### 对象结构型模式

* 采用组合或聚合来组合对象

### 继承关系相比较于组合关系耦合更高，所以对象更加具有灵活性

## 代理模式(Proxy)

### 定义

* 由于某些原因需要给某对象提供一个代理以控制对该对象的访问，这是访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介

### 优点

* 1、代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用
* 2、代理对象可以拓展目标对象的功能
* 3、 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度

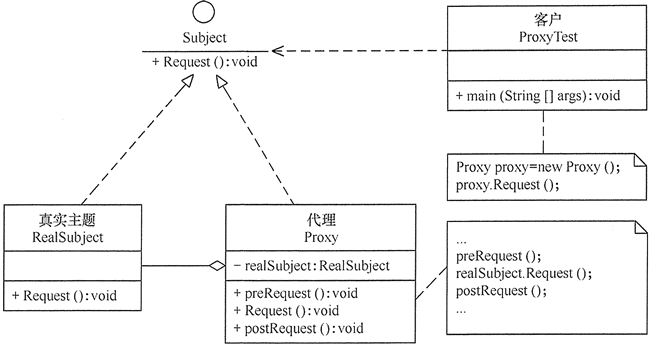
### 缺点

* 1、因为中介代理对象，会造成请求处理变慢
* 2、增加系统的复杂度

### 结构

* 抽象主题(Subject)类:通过接口或抽象类声明真实主题和代理对象实现的业务方法
* 真实主题(Real Subject)类:实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象
* 代理(Proxy)类:提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或拓展真实主题的功能

### 实现



### 应用场景

* 远程代理:隐藏目标对象存在的地址空间
* 虚拟代理:目标对象开销很大
* 安全代理:控制不同种类客户对真实对象的访问权限
* 智能指引:主要用于调用目标对象时，代理附加一些额外的处理功能,拓展代理性能
* 延迟加载:提高系统性能，延迟加载目标对象

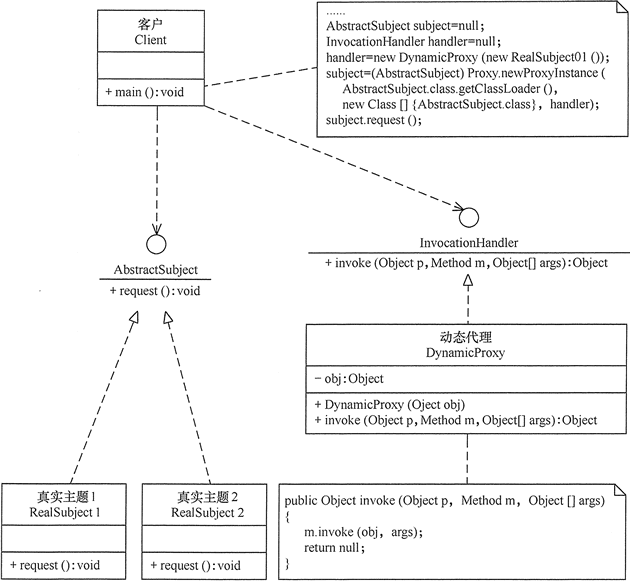
• Hibernate 中就存在属性的延迟加载和关联表的延时加载。

### 拓展

* 会造成额外代码，增加真实主题，也要增加代理类
* 真实主题必须事先存在，所以灵活性能会有很大影响

• 解决策略:动态代理,例如Spring的AOP

• AOP结构图



## 适配器模式(Adapter)

### 定义

* 将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

• 类结构模型

• 使用较少，原因:耦合度高，且要求程序员了解现有组件库中的相关组件的内部结构

• 对象结构模型

### 特点

* 优点

• 1、客户端通过适配器可以透明的调用目标接口

• 2、复用了现存的类，程序员不需要修改原有代码而重用现有的适配器

• 3、将目标类和适配器解耦，解决了目标类和适配者类接口不一致的情况

* 缺点

• 对类适配器来说,更换适配器的实现过程比较复杂

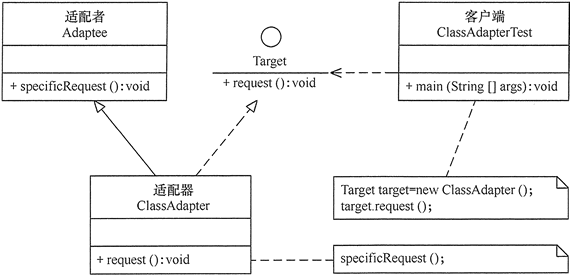
### 结构

* 1、目标接口(Target):当前系统业务所期待的接口，他可以是抽象类或者接口
* 2、适配者(Adaptee)类:它是被访问和适配的现存组件库中的组件接口
* 3、适配器(Adapter)类:它是一个转换器，通过继承或者引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者

### 实现

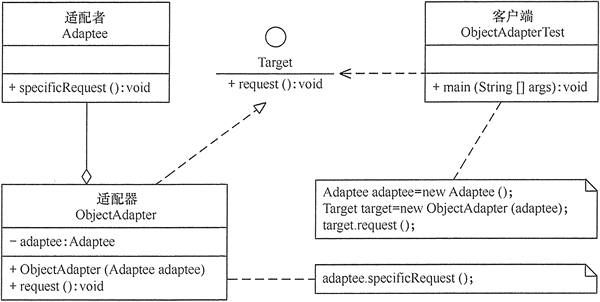
* 类结构模型

•



* 对象结构模型

•



### 场景

* 1、以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致
* 2、使用第三方提供的组件，但组件定义的接口和自己要求的接口定义不同
* spring中MethodInterceptor

### 拓展

* 双向适配器实现

## 桥接模式(Bridge)

### 定义

* 将抽象与实现分离，使他们可以独立变化.他是用组合关系代替继承关系实现，从而降低了抽象和实现这两个可变维度的耦合度

### 特点

* 优点

• 1、由于抽象与实现分离，拓展性强

• 2、其实现细节可以对客户透明

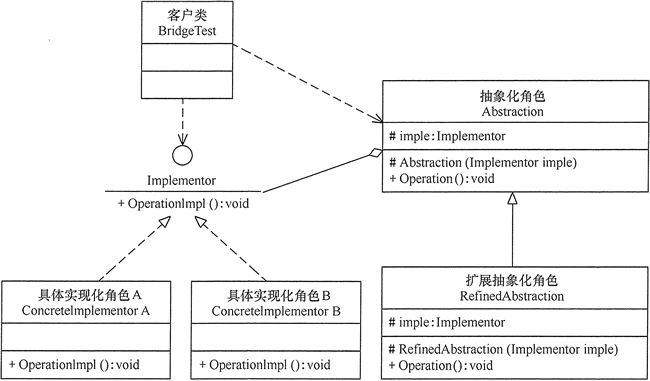
* 缺点

• 聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，这增加了系统的理解与设计难度，系统维度上的可读性

### 结构

* 1、抽象化(Abstraction)角色:定义抽象类，并包含一个对实现化对象的引用
* 2、拓展抽象化(Refined Abstraction)角色:是抽象化角色的子类，实现父类的业务方法，并通过组合关系调用实现化角色中的业务方法
* 3、实现化(Implementor)角色:定义实现化角色的接口，供拓展抽象化角色调用
* 4、具体实现化(Concrete Implementor)角色:给出实现化角色接口的具体实现

### 实现



### 应用场景

* 1、当一个类存在两个独立变化的唯独，且这两个唯独都需要进行拓展
* 2、当一个系统不希望使用聚成或者因为多层次继承导致系统类的个数急剧增加时
* 3、当一个系统需要在构建的抽象化角色和具体化角色之间增加更多的灵活性时
* spring的ViewRendererServlet
* Spring日志处理，多个日志插件做桥接

### 拓展

* 桥接模式一般配合适配器使用，当具体实现化角色的接口与现有类的接口不一致时，可以定义适配器将二者联合起来

## 装饰模式(Decorator)

### 定义

* 不改变现有对象结构的情况下，动态给对象增加一些职责(即增加其额外功能)的模式

### 特点

* 优点

• 采用装饰模式拓展对象功能比采用继承方式更加灵活

• 可以设计出多个不同的具体装饰类，创建出多个不同行为的组合

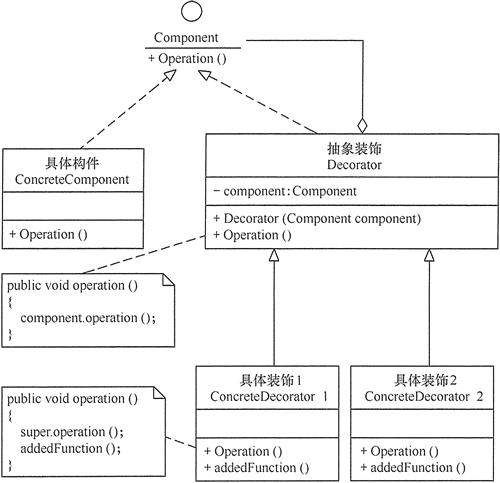
* 缺点

• 增加许多子类，使用过度会变得程序复杂

### 结构

* 1、抽象构件(Component)角色:定义一个抽象接口以规范准备接受附加责任的对象
* 2、具体构件(Concrete Component)角色:实现抽象构件，通过装饰角色为其添加一些职责
* 3、抽象装饰(Decorator)角色:继承抽象构件，并包含具体构件的实例，可以通过其子类拓展具体构件的功能
* 4、具体装饰(Concrete Decorator)角色:实现抽象装饰的相关方法，并给具体构件对象添加附加责任

### 实现



### 应用场景

* 当需要给一个现有类添加附加职责，而又不能采用生成子类方法进行扩充时
* 当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能
* 当对象的功能要求可以动态的添加，也可以在动态的撤销
* java I/O标准库抽象装饰类

### 拓展

* 装饰模式的反向引用

• 原抽象装饰应该是引用具体装饰构建，但是可能具体业务只有具体装饰构建，那么我们将抽象装饰继承具体装饰完成多种职能扩充

* 如果只有一个具体装饰时，可以将抽象装饰和具体装饰合并

### 跟代理模式比较

* 代理为内部创建对象

• 代理则是更多意义的原接口的保护，和行为的抽象

* 装饰为构造引用对象

• 装饰是能添加额外职能

## 外观模式(Facade)

### 定义

* 是一种通过多个复杂的子系统提供一个一致的接口，而使这些子系统更容易被访问的模式。

### 特点

* 优点

• 1、降低了子系统与客户端之间的耦合度，使子系统的变化不会影响调用它的客户类

• 2、对客户端屏蔽了子系统的组件，减少客户处理的对象数目，并使得子系统使用起来更加容易

• 3、降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编译一个子系统不会影响其他子系统，也不会影响外观对象

* 缺点

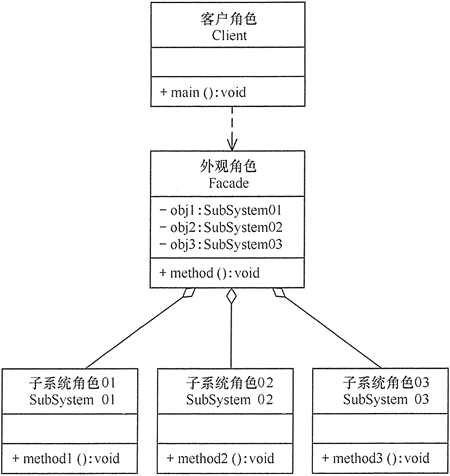
• 1、不能很好地限制客户使用子系统

• 2、增加新的子系统可能需要修改外观类或者客户端的源代码,违背了开闭原子

### 结构

* 1、外观角色(Facade):为多个子系统对外提供一个共用的接口
* 2、子系统(Sub System)角色:实现系统的部分功能，客户可以通过外观角色访问它
* 3、客户(client)角色:通过一个外观角色访问各个子系统的功能

### 实现



### 应用场景

* 1、对分层接口系统构建的时候，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系
* 2、当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问，
* 3、当客户端与多个子系统之间存在很大的联系时候，引入外观模式可以将它们分离，从而提高子系统的独立性和可移植性

### 拓展

* 我们需要修改子系统的时候需要修改外观类，违背了开闭原则，这个时候提供一个抽象外观类，如果引入抽象外观类，则在一定程度上解决了该问题

## 享元模式(Flyweight)

### 定义

* 运用共享技术来有效的支持大量细粒度对象的复用。

### 特点

* 优点

• 相同对象只要存一份，这降低了系统中的数量，从而降低了系统中细粒度对象给内存带来的压力

* 缺点

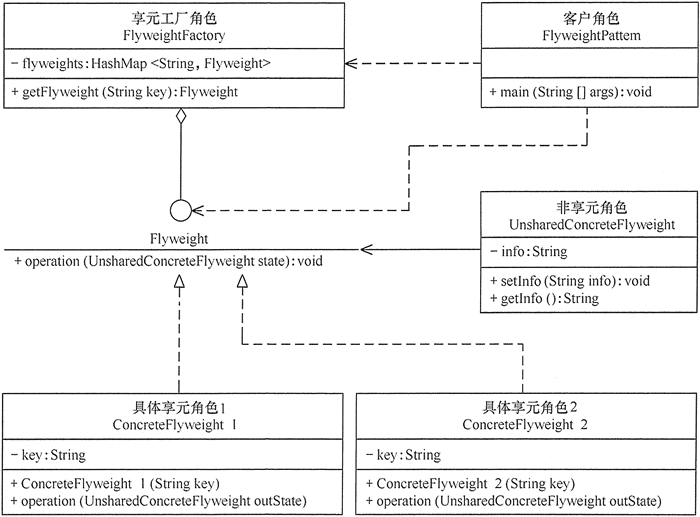
• 1、为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性

• 2、读取享元模式的外部状态会使得运行时间稍微变长

### 结构

* 1、抽象享元角色(Flyweight):是所有具体享元类的基类，为具体享元规范需要实现的公共接口，非享元的外部状态以参数的形式通过方法传入
* 2、具体享元(Concrete Flyweight)角色:实现抽象享元角色中所规定的接口
* 3、非享元(Unsharable Flyweight)角色:是不可以共享的外部状态，它以参数的形式注入具体享元的相关方法中。
* 4、享元工厂(Flyweight Factory)角色:负责创建和管理享元角色，当客户对象请求一个享元对象的时候，享元工厂检查系统中是否存在符合要求的享元对象，如果存在则提供给客户；如果不存在的话，则创建一个新的享元对象

### 实现



### 应用场景

* 1、系统中存在大量相同或者相似的对象，这些 对象耗费大量的内存资源
* 2、大部分的对象可以按照内部状态进行分组，且可将不同部分外部化，这样每个组主需要保存一个内部状态
* 3、由于享元模式需要额外维护一个保存享元的数据结构，所以应当在有足够多的享元实例时才值得使用享元模式
* jdk源码Integer 包
* String
* Apache Commons Pool2

https://blog.csdn.net/wwwdc1012/article/details/82833965

### 拓展

* 1、单纯享元模式，所有具体享元类都是可以共享的，不存在非共享的具体享元类
* 2、复合享元模式，用于将不同对象进行分解在被共享

## 组合模式(Composite)

### 定义

* 将对象组合成树状的层次结构模式，用来表示"部分-整体"的关系，使用户对单个对象和组合对象具有一致性

### 特点

* 优点

• 1、组合模式使得客户端代码可以一致地处理单个对象和组合对象，无需关心自己处理的对象是单个对象，还是组合对象，这简化了客户端代码

• 2、更容易在组合体内加入新的对象，客户端不会因为加入新的对象而更改源代码，满足开闭原则

* 缺点

• 1、设计较复杂，客户端需要花更多的时间清理类之间的层次关系

• 2、不容易限制容器中的构件

• 不容易用继承的方法增加构件的新功能

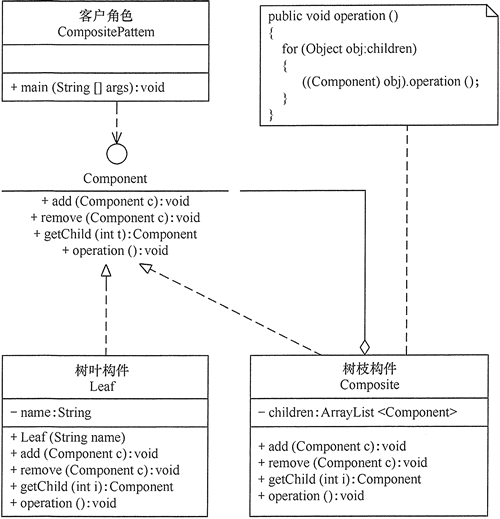
### 结构

* 1、抽象构件(Component)角色:它的作用主要是为了树叶构件和树枝构件声明公共接口，并实现它们的默认行为。在透明式的组合模式中抽象构件还声明的访问和管理子类的接口；在安全式的组合模式中不声明访问和管理子类的接口，管理工作由树枝构件完成
* 2、树叶构件(Leaf)角色:是组合中的叶节点对象，他没有子节点，用于实现抽象构件角色中声明的公共接口
* 3、树枝构件(Composite)角色:是组合中的分支节点对象，他有子节点。它实现了抽象构件角色中声明的接口，它的作用是存储和管理子部件，通常包含Add(),Remove()及GetChild()等方法

### 实现

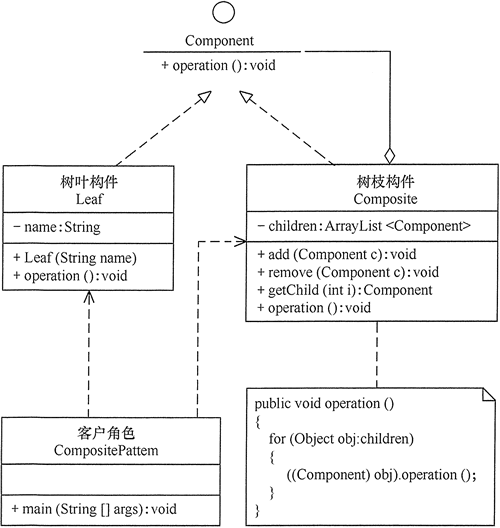
* 透明方式

•



* 安全方式

•



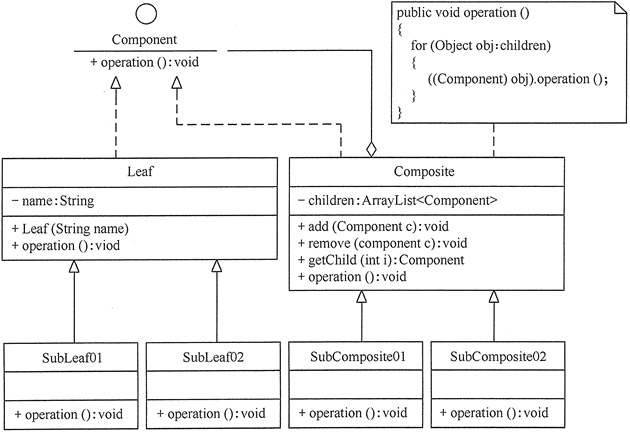
### 应用场景

* 1、在需求表示一个对象整体与部分的层次结构的场合
* 2、要求对用户隐藏组合对象与单个对象不同，用户可以统一接口使用组合结构中的所有对象的场合
* 集合如HashMap，ArrayList中的putAll()方法
* ybatis SqlNode中的组合模式

### 拓展

* 组合模式中树叶节点和树枝节点抽象，对其增加子节点，这时候组合模式成为复杂组合模式

•



## 小结

### 代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

### 适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

### 桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现的，从而降低了抽象和实现这两个可变维度的耦合度。

### 装饰（Decorator）模式：动态地给对象增加一些职责，即增加其额外的功能。

### 外观（Facade）模式：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

### 享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。

### 组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

## 区别

### 代理:主要面向代理对象，在代理内部执行对被代理对象的增强操作

### 适配器:针对接口层面，对不同的需要适配的类进行包装增强在通过接口暴露，一般不会再初始项目设计该模式，一般用作系统开发完成统一接口的时候使用

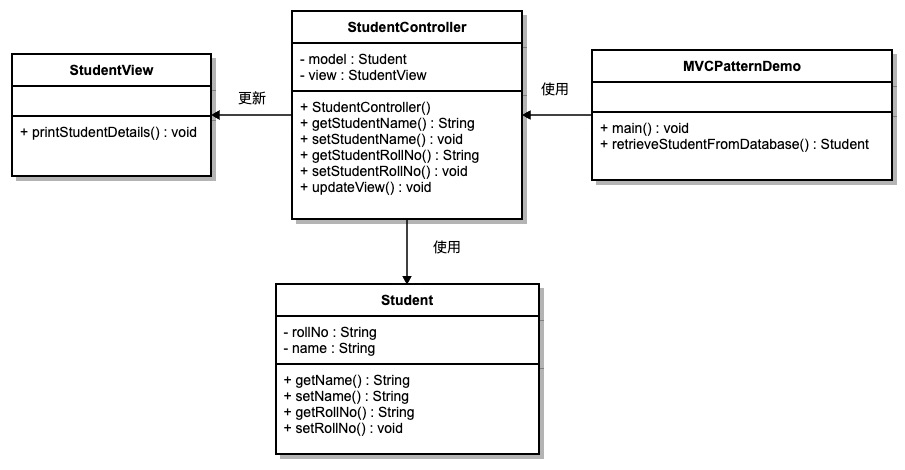
# J2EE模式

## MVC模式(MVC Pattern)

### 结构

* 模型层(Model):指从现实世界中抽象出来的对象模型，是应用逻辑的反应；它封装了数据和对数据的操作，是实际进行数据处理的地方
* 视图层(View):是应用和用户之间的接口，它负责将应用显示给用户和显示模型的状态
* 控制器(Controller)控制负责视图和模型之间的响应，响应方式和流程;它主要两方面的动作，一是用户的请求分发到相应的模型，二是把模型的改变及时反映到视图上。

### 图解



## 业务代表模式(Business Delegate Pattern)

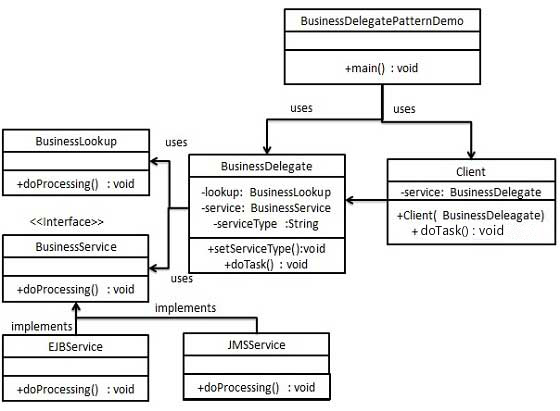
### 定义

* 表示层和业务层解耦。他基本上是用来减少通信或对表示层代码中的业务层代码的远程查询功能。

### 结构

* 客户端(Client):表示代码可以是jsp、servlet或UI java代码
* 业务代表(Business Delegate):一个客户端实体提供的入口类，它提供了对业务服务方法的访问
* 查询服务(LookUp service):查找服务对象负责获取相关业务实现，并提供业务对象对业务代表对象的访问
* 业务服务(Business Service):业务服务接口。实现了该业务服务的实体类，提供了实际的业务实现逻辑

### 实现



## 组合实体模式(Composite Entity Pattern)

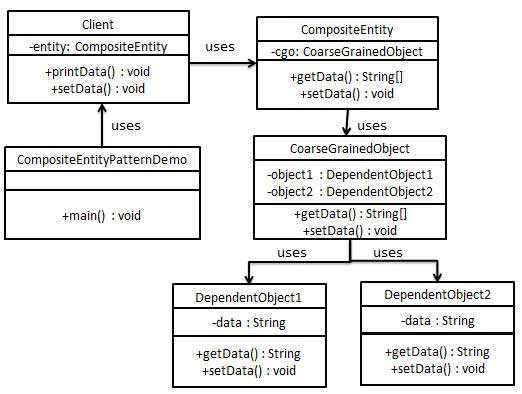
### 定义

* 一个组合实体是一个EJB实体bean，代表了对象的图解。当更新一个组合实体时，内部依赖对象beans会自动更新，因为他们是由EJB实体bean管理的

### 结构

* 组合实体(Composite Entity):它是主要实体bean，它是可以粗粒的，或者可以包含一个粗粒的对象，用于持续生命周期
* 粗粒度对象(Coarse-Grained Object)该对象包含以来对象，它有自己的生命周期。也能依赖管理对象的生命周期
* 依赖对象(Dependent Object):依赖对象是一个持续生命周期依赖于粗粒度对象的对象
* 策略(Strategies):表示如何实现组合实体

### 实现



## 数据访问对象模式(Data Access Object Pattern)

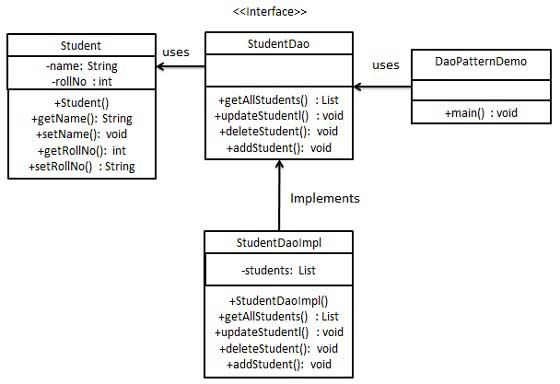
### 定义

* 用于把低级的数据访问API或操作从高级业务服务中分离出来

### 结构

* 数据访问接口对象(Data Access Object Interface) - 该接口定义了在一个模型对象上要执行的标准操作
* 数据访问对象实体类（Data Access Object concrete class） - 该类实现了上述的接口。该类负责从数据源获取数据，数据源可以是数据库，也可以是 xml，或者是其他的存储机制。
* 模型对象/数值对象（Model Object/Value Object） - 该对象是简单的 POJO，包含了 get/set 方法来存储通过使用 DAO 类检索到的数据

### 实现



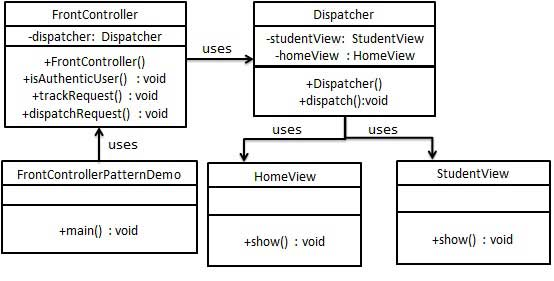
## 前端控制器模式(Front Controller Pattern)

### 定义:提供一个集中的请求处理机制，所有请求都将由一个单一的处理程序处理，该处理程序可以做认证/授权/记录日志或者跟踪请求，然后把请求传给相应的处理程序

### 结构

* 前端控制器(Front Controller):处理应用程序所有类型请求的单个处理程序，应用程序可以是基于web的应用程序，也可以是基于桌面的应用程序
* 调度器(dispather):前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序
* 视图(view):视图是为请求而创建的对象

### 实现



## 拦截过滤器模式(Intercepting Filter Pattern)

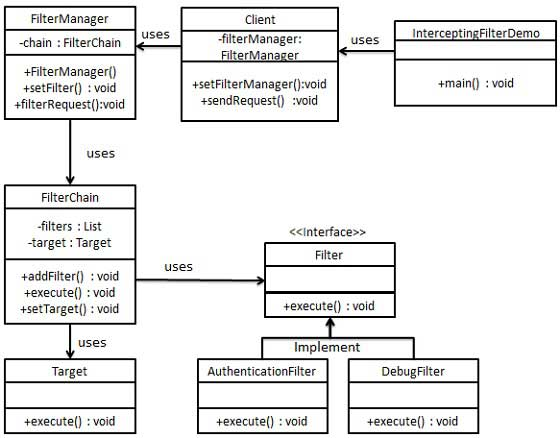
### 定义

* 用于对应应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给时机目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪求请求，然后把请求传给相应的处理程序

### 结构

* 过滤器(Fiter):过滤器在请求处理程序请求之前或之后，执行某些任务
* 过滤器链(Filter Chain):过滤器带有多个过滤器，并在Target上按照定义的顺序执行这些过滤器
* target :target对象是请求处理程序
* 过滤管理器(Filter Manager):过滤管理器管理过滤器和过滤器链
* 客户端(client):Client像Target对象发送请求的对象

### 实现



## 服务定位器模式(Aervice Locator Pattern)

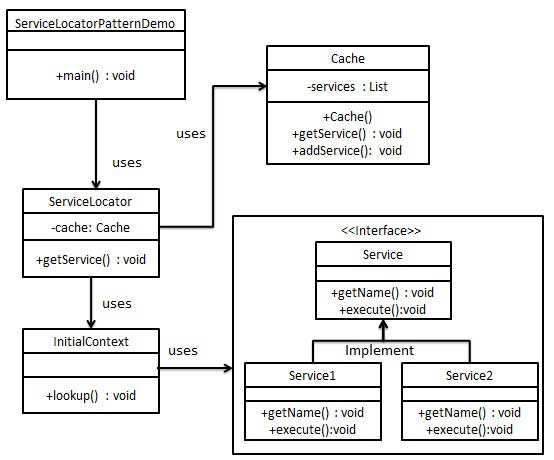
### 定义

* 我们想使用jndi查询定位各种服务的时候，考虑到为某个服务查找JNDI的代价很高，服务定位模式充分利用了缓存技术，在首次请求某个服务时，服务定位器在JNDI中查找服务，并缓存该服务对象。再次请求相同服务时，服务定位器会在它的缓存中查找，这样可以在很大程度上提高程序的性能

### 结构

* 服务(Service):实际处理请求的服务。对这种服务的引用可以再JNDI服务器中查找到
* Context/初始化Context:JNDI Conext带有对要查找的服务的引用
* 服务定位器(Service Locator):服务定位器是通过JNDI查找和缓存服务来获取服务的单点接触
* 缓存(Cache):缓存存储服务的引用，以便复用它们。
* 客户端(Client):Client是通过ServiceLocator调用服务对象

### 实现



## 传输对象模式(Transfer Object Pattern)

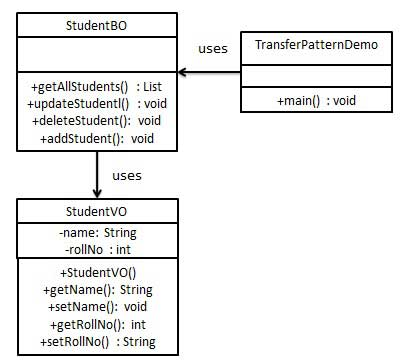
### 定义

* 用于从客户端向服务器一次性传递带有多个属性的数据。传输对象也被称为数值对象。传输对象是一个具有getter/setter方法的简单的POJO类，他是可序列化，所以它可以通过网络传输。他没有任何行为。服务器端的业务类通常从数据库读取数据，然后填充POJO，并把它发送到客户端或者按值传递它。对于客户端，传输对象是只读的。客户端可以创建自己的传输对象，并把它传递给服务器，以便一次性更新数据库中的数值

### 结构

* 业务对象(Business Object)-为传输对象填充数据的业务服务
* 传输对象(Transfer Object)-简单的POJO，只有设置/获取属性的方法
* 客户端(Client):客户端可以发送请求或者发送传输对象到业务对象

### 实现



# UML绘图关系

## 结构

### 类

* 他是一组具有相同属性、操作、关系、语义的对象的抽象。在UML中，类使用带有分隔线的矩形来表示

• 结构

• 部分(NAME)

• 属性(Attribute)

• [可见性]属性名:类型[=默认值]

• 操作部分(Operation)

• [可见性]名称(参数列表)[:返回类型]

• 公有（public）

• +

• 私有（private）

• -

• 受保护（protected）

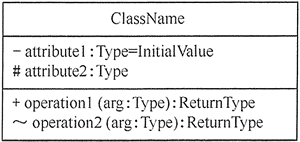
• #

• 朋友（fiiendly）

• ~

• 图例

•



### 关系

* 关系内容

• 1、依赖关系(Dependency),使用带箭头的虚线来表示，箭头从实用类指向被依赖的类

• 2、关联关系(Association),分为双向关联和单向关联两种。其中，双向关联可以用带两个箭头或者没有箭头的实线表示，单向关联用带一个箭头的实线表示，箭头从使用类指向被关联的类。还可以在关联线的两端标注角色名，补充说明他们的角色

• 3、聚合关系(Aggregation):用带空心菱形的实线表示，菱形指向整体

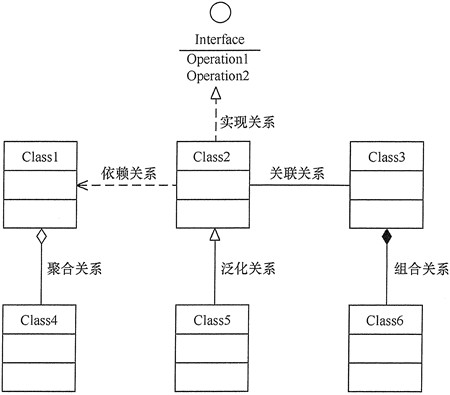
• 4、组合关系(Composition)，用带实心的菱形实线表示菱形指向整体

• 5、泛化关系(Generalization):用带空心三角箭头的实现来表示，箭头从子类指向父类

• 6、实现关系(Realization):用带空心的三角箭头虚线表示，箭头从实现类指向接口

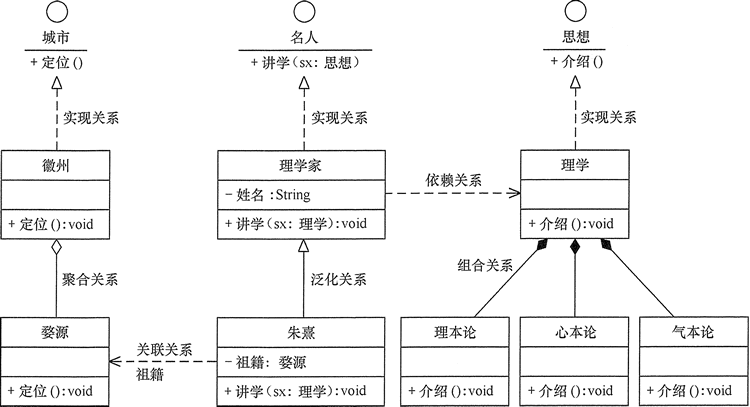
* 图例

•



## 实践例子

### 



# 行为型模式

## 定位分析

### 用于描述程序在运行时复杂的流程控制，描述多个类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，它涉及算法与对象职责分配

* 类行为

• 类间分派行为

* 对象行为

• 聚合或组合在对象间分配行为

* 对象行为模式更灵活

## 模板模式(TemplateMethod)

### 定义

* 定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤

### 特点

* 优点

• 1、封装了不变部分，拓展可变部分。他认为是不变部分的算法封装到父类中实现，而把可变部分由子类继承实现，便于子类继续拓展

• 2、它在父类中 提取了公共的部分代码，便于代码复用

• 3、部分方法是由子类实现，因此子类可以通过拓展方式增加相应的功能，符合开闭原则

* 缺点

• 1、对于每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象

• 2、父类中的抽象方法由子类实现，子类执行的结果会影响父类结果，这导致一种反向的控制结构，提高代码阅读难度

### 结构

* 1、抽象类(Abstract Class):负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成

• ①模板方法:定义了算法骨架，按照某种顺序调用其包含的基本方法。

• ②基本方法:是整个算法中的一个步骤

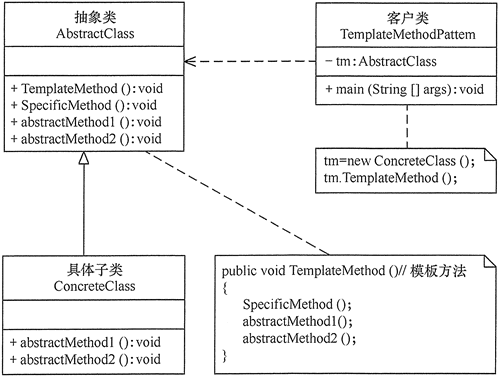
• 抽象方法:在抽象类中申明，由具体子类实现

• 具体方法:在抽象类中已经实现，再具体子类中可以继承或者重写它。

• 钩子方法:在抽象类中已经实现，包括用于判断的逻辑方法和需要的子类重写的空方法两种

* 2、具体子类(Concrete):实现抽象类中所定义的抽象方法和钩子方法，他们是一个顶级逻辑的组成步骤

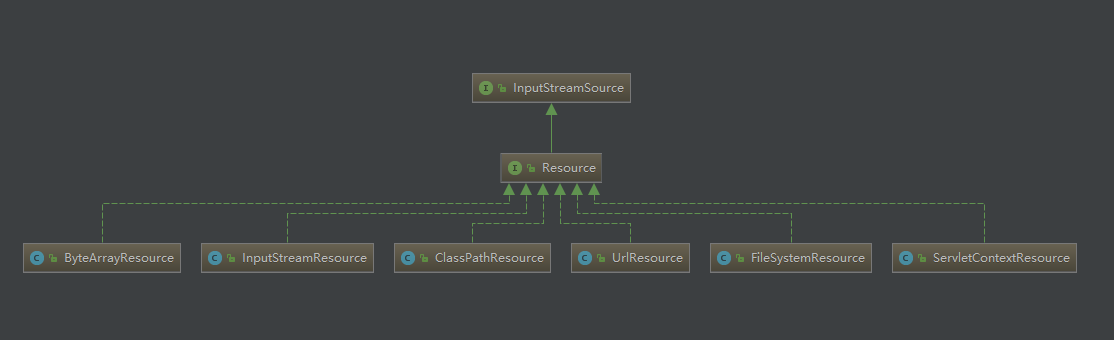
### 实现



### 应用场景

* 1、算法步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变得部分抽象出来，供子类实现
* 2、当多个子类存在公共的行为时，可以将其提取出来并集中到一个公共父类中避免代码重复。首先要识别代码的不同之处，并且将不同之处分离为新的操作，最后，用一个调用这些新的操作的模板方法来替换这些不同的代码
* 3、当需要控制子类扩展时，模板方法只在特定调用钩子操作，这样就只允许在这些点进行拓展
* 钩子编程
* spring的Resource接口

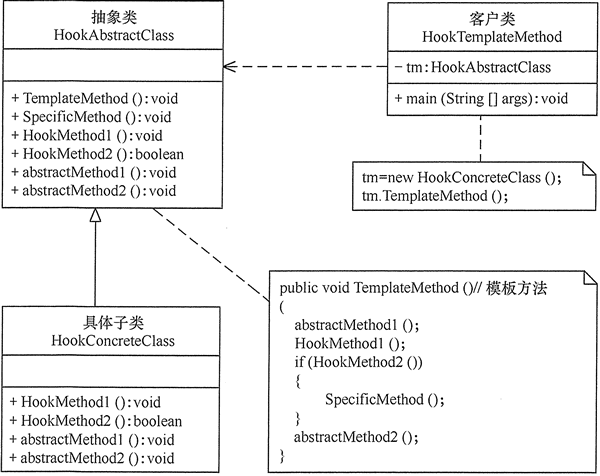
•



### 扩展

* 通过钩子方法控制父类执行策略

•



## 策略模式(StrategyMethod)

### 定义

* 定义了一些列算法封装起来，使他们可以相互替换，且算法的变化不会影响使用算法的客户。算法封装对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理

### 特点

* 优点

• 1、多重条件语句不易维护，而使用策略模式可以避免使用多重if条件语句

• 2、策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复代码

• 3、策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的

• 4、策略模式提供了对开闭原则的完美支持，可以在不修改源代码的情况下，灵活增加新的算法

• 5、策略模式把算法的使用放到了环境类中，而算法的实现移到具体策略类中，实现了二者的分离

* 缺点

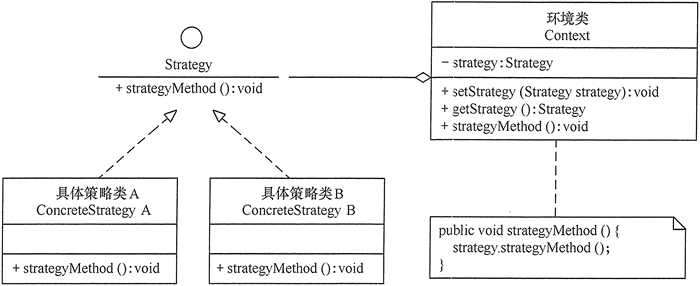
• 1、客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类

• 2、策略模式造成很多的策略类

### 结构

* 1、抽象策略(Strategy)类:定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法
* 2、具体策略(Concrete Strategy)类:实现了抽象策略定义接口。提供具体的算法实现
* 3、环境(Context)类:持有一个策略类的引用，最终给客户端调用

### 实现



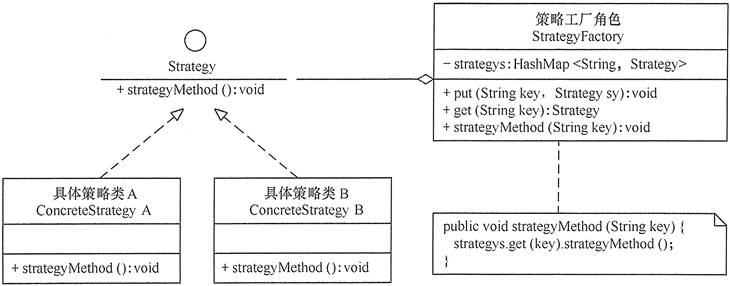
### 应用场景

* 1、一个系统需要动态在几中算法选择其中一种的时候
* 2、一个类定义了多个行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可以将每个条件分支移入它们各自的策略类中实现
* 3、系统中各算法彼此完全独立，并且要求客户隐藏具体算法的实现细节
* 4、系统中要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构
* 5、多各类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为
* java SE 容器布局管理
* spring SimpleInstantiationStrategy

### 拓展

* 在产品迭代过程中策略不断增多可以通过策略工厂进行维护，减少客户端操作

•



## 命令模式(command)

### 定义

* 将一个请求封装为一个对象，使发出请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行存储、传递、调用、增加、与管理

### 特点

* 优点

• 1、降低系统耦合度.命令模式能够将调用操作的对象与实现该操作的对象解耦

• 2、增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，满足开闭原则，拓展灵活

• 3、可以实现宏命令。命令模式可以跟组合模式结合，将多个命令装配成一个组合命令，即宏命令

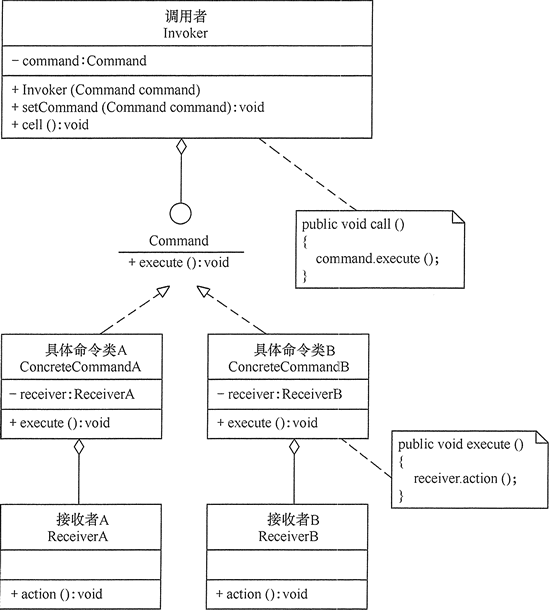
• 4、方便实现Undo和Redo操作，命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复操作

* 可能产生大量具体命令类。因为涉及每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性

### 结构

* 1、抽象命令类(Command)角色:声明执行命令的接口，拥有执行命令的抽象方法execute().
* 2、具体命令类(Concrete Command)角色:是抽象命令类的具体实现类，他拥有接受者对象，并通过调用接收者的功能来完成命令要执行的操作
* 3、实现类/接受者(Receiver)角色:执行命令功能的相关操作，是具体命令对象业务的真正实现者
* 4、调用者/请求者(Invoker)角色:是请求的发送者，他通常有用很多的命令对象，并通过访问命令对象来执行相关请求，它不直接访问接收者

### 实现



### 场景

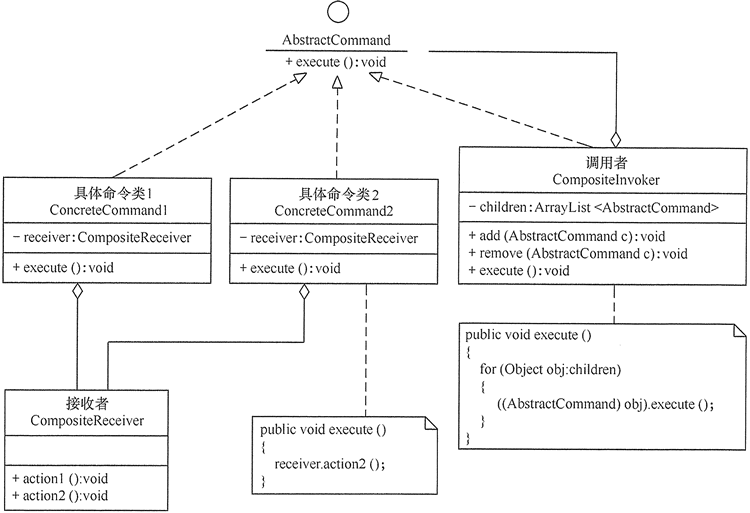
* 1、当系统需要将请求调用者与请求接受者解耦时，命令模式使得调用者和接收者不直接交互
* 2、当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能
* 3、当系统需要执行一组操作时。命令模式可以定义宏命令来实现这些功能，
* 4、当系统需要支持命令的撤销undo操作和回复redo操作时，可以将命令对象存储起来，采用备忘录模式来实现

### 拓展

* 宏命令模式拓展

• 组合模式+命令模式实现

•



## 责任链模式(Chain of Reponsibility)

### 定义

* 为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理通过前一对象记住其下一个对象的引用而连成的一条链;当请求发生时，可将请求沿着这条链传递，知道有对象处理它为止

### 特点

* 优点

• 1、降低了对象之间的耦合度。该模式使一个对象无需知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无需拥有对方的明确信息

• 2、增强了系统的可拓展性。可以根据增加新的请求处理类，满足开闭原则

• 3、增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态的改变链内的成员或者调动它们的词序，也可动态的新增或者删除责任

• 4、责任链简化了对象之间的练习，每个对象只需要保持一个指向其后继者的引用，不需要保持其他所有处理者的引用，不需保持其他所有处理者的引用，这避免了使用众多条件语句

• 5、责任分担、每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合单一职责原则

* 缺点

• 1、不能保证每个请求一定被处理，由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理

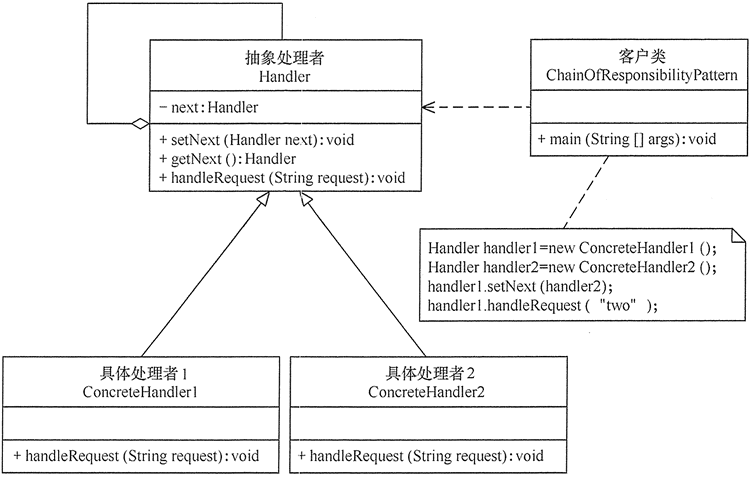
• 2、对比较长的职责链，请求的处理可能涉及多个对象处理，系统性能将受到一定影响

• 3、职责链建立的合理性需要靠客户端保证，增加客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用

### 结构

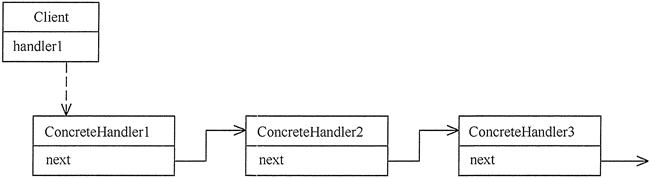
* 1、抽象矗立着(Handler)角色:定义一个处理请求的接口，包含抽象处理方法和一个后继链接
* 2、具体处理者(Concrete Handler)角色:实现抽象处理者的处理方法，判断能否处理本次请求，否则将该请求转给它的后继者
* 3、客户类(Client)角色:创建处理链，并向链头的具体处理者对象提交请求，他不关心处理细节和请求的传递过程

### 实现



* 责任链

•



### 应用场景

* 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定
* 可动态指定一组对象处理请求，或者添加新的处理者
* 再不确定请求处理者的情况下，向多个处理者中的一个提交请求
* Strut2的拦截器
* Java Servlet中的Filter
* Dobbo中的FIiter
* MyBatis中的Plugin插件

### 拓展

* 1、纯的责任链模式:一个请求必须被某一个处理者对象所接受，且一个具体处理者对某个请求的处理只能采用以下两种行为之一
* 2、不纯的责任链模式:允许出现某一个具体处理者对象在承担了请求的一部分责任后又将剩余的责任传给下家的情况，且一个请求可以最终不被任何接受端对象消费

## 状态模式(State)

### 定义

* 对有状态的对象，把复杂的"判断逻辑"提取到不同状态对象中，允许状态对象再其内部状态发生改变时改变其行为

### 特点

* 优点

• 1、状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足单一职责原则

• 2、减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖

• 3、有利于程序的拓展。通过定义新的子类很容易的增加新的状态和转换

* 缺点

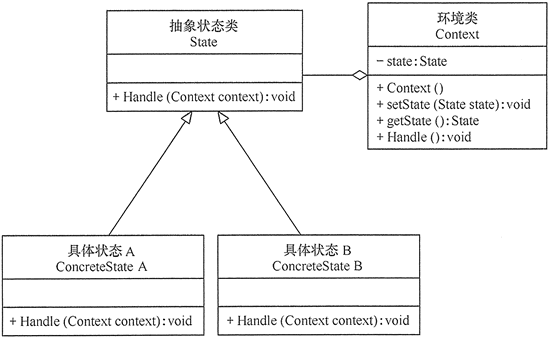
• 1、状态模式的使用必然会增加系统的类与对象的个数

• 2、状态模式的结构与实现都较为复杂，如果使用不当会导致程序结构和代码混乱

### 结构

* 1、环境角色(Context):也称作上下文，它定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理
* 2、抽象状态(State)角色:定义一个接口，用以封装环境对象中的特定的状态所对应的行为
* 3、具体状态(Concrete State)角色:实现抽象状态所对应的行为

### 实现



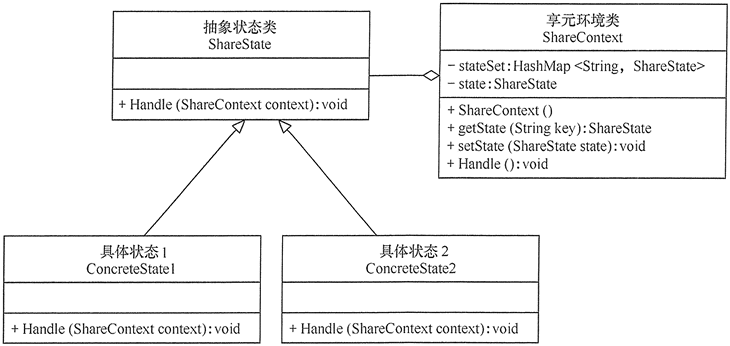
### 应用场景

* 当一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为时，就可以考虑使用状态模式
* 一个操作中含有庞大的分支结构，并且这些分支决定对象的状态
* 线程应用

### 拓展

* 有些情况下，可能有多个环境对象需要共享一组状态，这时需要引入享元模式，将这些具体状态对象放在集合中供程序共享

•



## 观察者模式(Observer)

### 定义

* 指多个对象存在一对多关系的依赖关系，当一个对象状态发生改变时，所有依赖于它的对象都得到通知并被自动更新，这种模式有时又称作发布-订阅模式，模型-视图模式，它是对象行为型模式

### 特点

* 优点

• 1、降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系

• 2、目标与观察者之间建立一套触发机制

* 缺点

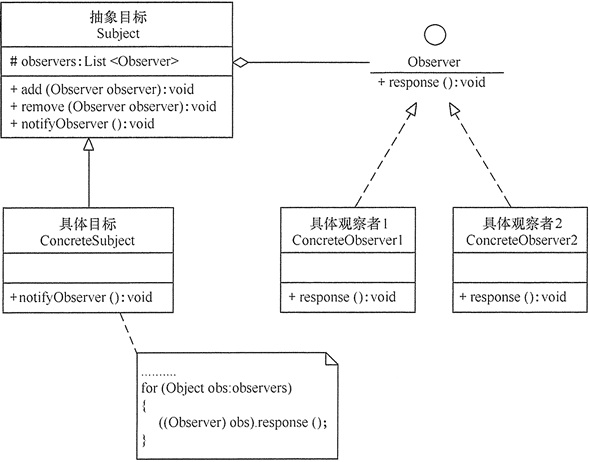
• 1、目标与观察者之间的耦合关系并没有完全解除，而且可能会出现循环依赖

• 2、当观察者模式对象很多时候，通知的发布会花费很多时间，影响程序效率

### 结构

* 1、抽象主题(Subject)角色:也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加，删除观察者对象的方法，以及通知所有观察者的抽象方法。
* 2、具体主题(Concrete Subject)角色:也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生变化的时候，通知所有注册过的观察者对象
* 3、抽象观察者(Observer)角色:它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。
* 4、具体观察者（Concrete Observer）角色:实现抽象观察者方法中定义的抽象方法，以便在得到目标更改通知时更新自身状态

### 实现



### 应用场景

* 对象间存在一对多关系，一个对象改变会影响其他对线
* 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使他们刻一个字独立的改变复用

### 拓展

* java已经支持Observable类和observer接口，直接继承或者实现就可以编写观察者模式

• Observable类

• Observable 类是抽象目标类，它有一个 Vector 向量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法。

• 1、void addObserver(Observer o) 方法：用于将新的观察者对象添加到向量中。

• 2、void notifyObservers(Object arg) 方法：调用向量中的所有观察者对象的 update。方法，通知它们数据发生改变。通常越晚加入向量的观察者越先得到通知。

• 3、void setChange() 方法：用来设置一个 boolean 类型的内部标志位，注明目标对象发生了变化。当它为真时，notifyObservers() 才会通知观察者。

• Observer 接口

• Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用 void update(Observable o,Object arg) 方法，进行相应的工作。

## 中介者模式(Mediator)

### 定义

* 定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变他们之间的交互，又称调停模式，迪米特法则的典型应用

### 特点

* 优点

• 1、降低对象之间的耦合性，使得对象容易对立被复用

• 2、将对象间的一对多关联转变成一对一的关联，提高系统的灵活性，使得系统易于独立地被复用

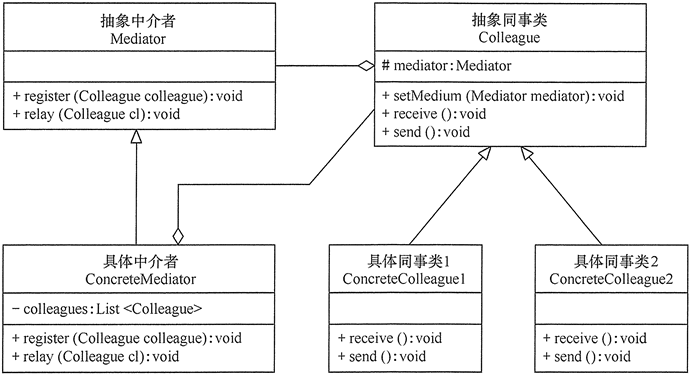
* 缺点

• 当同事类太多的时候，中介者职责很大，不利于维护

### 结构

* 1、抽象中介者(Mediator)角色:他是中介者的接口，提供了同事对象信息的抽象方法
* 2、具体中介者(ConcreteMediator)角色:实现中介者接口，定义一个list来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色
* 3、抽象同事类(Collague)角色:定义同事类的接口，保护中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能
* 4、具体同事类(Concrete Collague)角色:是抽象同事类的是闲着，当需要与其他同事对象交互时，由中介者对象负责后续的交互

### 实现



### 应用场景

* 当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用
* 当想创建一个运行于多个类之间的对象，又不想生成新得子类时候
* Mvc框架中控制层就是一个中介者模式，前端控制器
* C/S架构

### 拓展

* 简单中介者

• 1、不定义中介者接口，把具体中介者对象实现成为单例

• 2、同时对象不持有中介者，而是在需要的时候直接获取中介者对象并调用

## 迭代器模式(Iterator)

### 定义

* 提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象内部的表示。迭代器模式是一种对象行为模式

### 特点

* 优点

• 1、访问一个聚合对象的内容而无需暴露它的内部表示

• 2、遍历任务交由迭代器完成，简化聚合类

• 3、它支持以不同方式遍历一个聚合，升值可以自定义迭代器的子类以支持新的遍历

• 4、增加新的聚合类和迭代器都很方便，无须修改原有代码

• 5、封装性良好，为遍历不同的聚合结构提供一个统一的接口

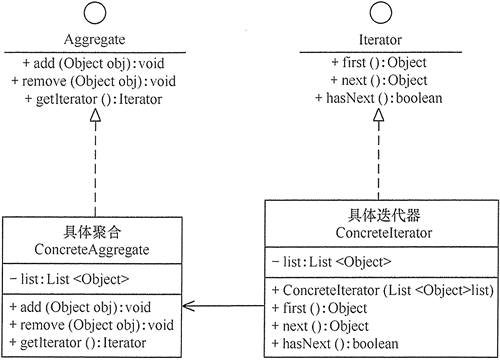
* 缺点

• 增加类的个数，在一定程度上增加系统的复杂性

### 结构

* 1、抽象聚合(Aggregate)角色:定义存储、添加、删除聚合对象以及创建迭代器对象的接口
* 2、具体聚合(ConcreteAggregate)角色:实现抽象聚合类，返回一个具体迭代器的实例
* 3、抽象迭代器(Iterator)角色:定义访问和遍历聚合元素的接口，通常包含hashNaxt()、first()、next()等方法
* 4、具体迭代器(ConcreteIterator)角色:实现抽象迭代器接口中所定义的方法，完成聚合对象的遍历，记录遍历的当前位置

### 实现



### 应用场景

* 当需要为聚合对象提供多种遍历方式时
* 当需要为遍历不同的聚合结构提供一个统一的接口的时候
* 当访问一个聚合对象的内容而无需暴露其内部实现的时候

### 拓展

* 组合模式容器构件进行访问时，经常将迭代器潜藏在组合模式的容器构成类中，当然也可以构造一个外部迭代器来对容器构件进行访问

## 访问者模式(Visitor)

### 定义

* 将作用于某种数据结构中的各个元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问，它将对数据的操作与数据结构进行分离，是行为模式中最复杂的一种模式

### 特点

* 优点

• 1、拓展性能好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能

• 2、复用性好，可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度

• 3、灵活性好，访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构

• 4、符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能比较单一

* 缺点

• 1、增加新的元素类很困难。在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者中增加具体操作，这违背了开闭原则

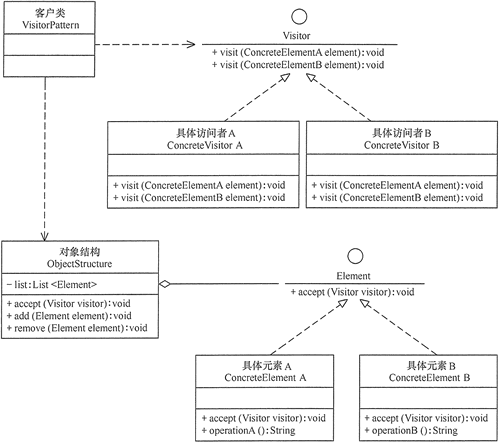
• 2、破坏封装。访问者模式中具体元素对访问者公布细节，这破坏了对象的封装性

• 3、违反了依赖倒置原则。访问者模式依赖了具体类，而没有依赖抽象类

### 结构

* 1、抽象访问者(Visitor)角色:定义一个访问具体元素的接口，为每个元素对应一个访问操作visit()，该操作中的参数类型标识了被访问的具体元素
* 2、具体访问者(ConcreteVisitor)角色:实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么
* 3、抽象元素(Element)角色:声明包含一个接受操作的accept()的接口，被接受访问对象作为accept()方法的参数
* 4、具体元素(ConcreteElement)角色:实现抽象元素角色提供的accept()操作，其方法通常都是visitor.visit(this)，另外具体元素中可能还保函业务本身业务逻辑相关操作
* 5、对象结构(Object Structure)角色:是一个包含元素角色的容器，提供让访问者对象遍历容器中所有的元素的方法，通常由List，set，map等聚合类实现

### 实现



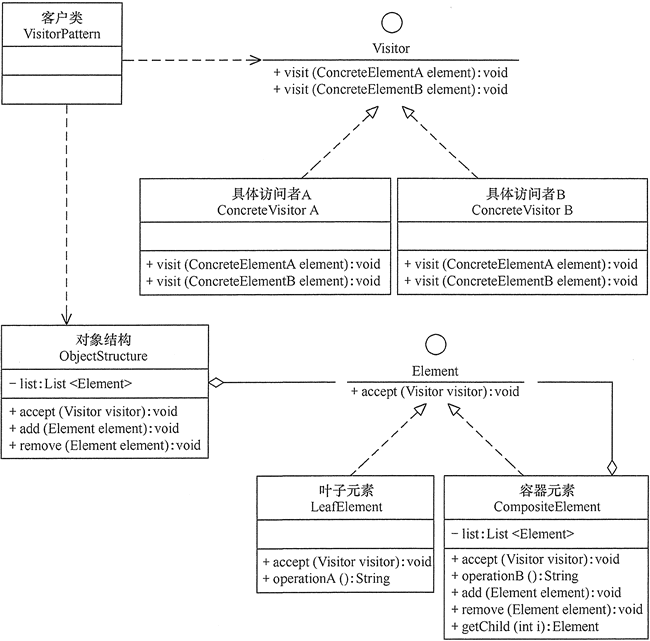
### 应用场景

* 1、对象结构相对稳定，但其操作算法经常变化的程序
* 2、对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构
* 3、对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作

### 拓展

* 访问者模式配合迭代器模式
* 访问者模式配合组合模式

•



## 备忘录模式(Memento)

### 定义

* 在不破坏封装性的前提下，捕获一个对象的内部状态，并对该对象之外的保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态

### 特点

* 优点

• 1、提供了一种可以恢复状态的机制，当用户需要时能够比较方便地将数据恢复到某个历史状态

• 2、实现内部状态的封装，除了创建它的发起人之外，其他对象都不能够访问这些状态信息

• 3、简化了发起人，发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则

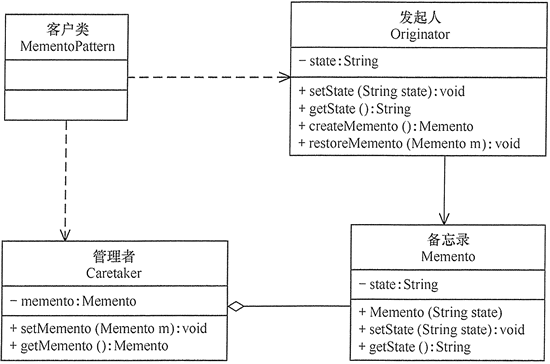
* 缺点

• 资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

### 结构

* 1、发起人(Originator)角色:记录当前时刻的内部状态信息，提供创建备忘录和回复备忘录数据的功能，实现其他业务的功能，他可以访问备忘录里的所有信息
* 2、备忘录(Memento)角色:负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人
* 3、管理者(Caretaker)角色:对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改

### 实现



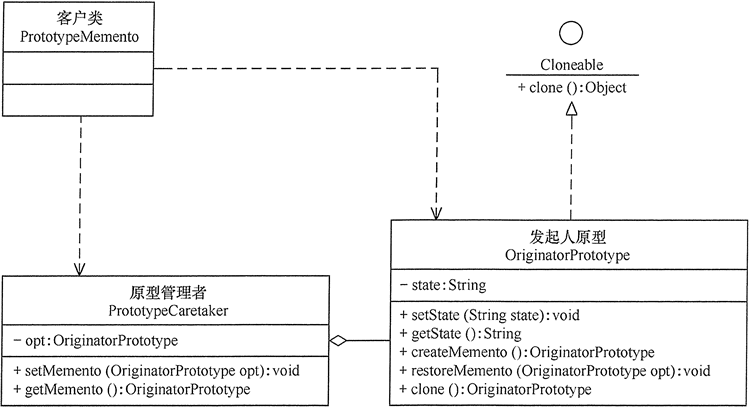
### 应用场景

* 1、需要保存与恢复数据的场景
* 2、需要提供一个可回滚的操作的场景
* 数据库redo

### 拓展

* 备忘录模式和原型模式混用，备忘录定义备份发起人信息，原型模式的clone()存储副本。这样减少备忘录类

•



## 解释器模式(Interpreter)

### 定义

* 给分析对象定义一个语言，并定义该语言的文法表示，在设计一个解释器来解释语言中的句子，也就是说，用编译语言的方式来分析应用中的实例。这种模式实现了文法表达式处理的接口，该接口解释一个特定的上下文

### 特点

* 优点

• 1、拓展性好。由于在解释器模式中使用类来表示语言的文法规则，因此可以通过继承等级制来改变或拓展文法

• 2、rongyishixian.zia语法树中的每个表达式节点类都是相似的，所以实现其文法较为容易

* 缺点

• 1、执行效率低解释器模式中通常使用大量的循环和递归调用，当要解释的句子比较复杂的时候，其运行速度很慢，且代码的调试过程也比较麻烦

• 2、会引起类膨胀。解释器模式中的每条规则至少需要定义一个类，当包含的文法规则很多时，类的个数急剧增加，导致系统难以管理与维护

• 3、可应用的场景比较少，在软件开发中，需要定义语言文法的应用实例非常少，所以这种模式很少被使用到

### 结构

* 文法概念

• 文法

〈句子〉::=〈主语〉〈谓语〉〈宾语〉〈主语〉::=〈代词〉|〈名词〉〈谓语〉::=〈动词〉〈宾语〉::=〈代词〉|〈名词〉〈代词〉你|我|他〈名词〉7大学生I筱霞I英语〈动词〉::=是|学习

• 句子

• 语法数

* 模式结构

• 1、抽象表达式(Abstract Expression)角色:定义解释器接口，约定解释器的解释操作，主要包含解释方法interpret()

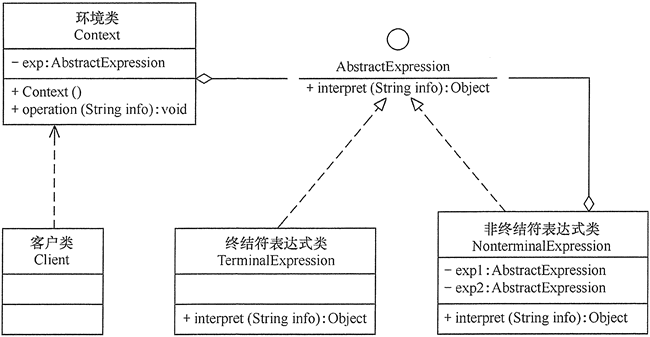
• 2、终结符表达式(Terminal Expression)角色:是抽象表达式的子类，用来实现文法中与终结符先关的操作，文法中的每一个终结符都有一个具体中介表达式与之相对

• 3、非终结符号表达式(Nonterminal Expression)角色:也是抽象表达式子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式

• 4、环境(Context)角色:通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值

• 5、客户端(Client)角色:主要任务是将需要分析的句子或者表达式转换成使用解释器对象描述的抽象语法数，然后调用解释器的解释方法，当然也可以通过环境角色间接访问解释器的解释方法

### 实现



### 应用场景

* 1、当语言的文法比较简单，且执行效率不是关键问题时候，
* 2、当问题重复出现，且可以用一种简单的语言来进行表达式时
* 3、当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如xml文档解释
* Spring EL表达式中的解释器模式

### 拓展

* 拓展

• 在项目开发中，如果要对数据表达式进行分析与计算，无须再用解释器模式进行设计了，Java 提供了以下强大的数学公式解析器：Expression4J、MESP(Math Expression String Parser) 和 Jep 等，它们可以解释一些复杂的文法，功能强大，使用简单。

## 小结

### 模板方法（Template Method）模式：定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

### 策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

### 命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

### 职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

### 状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。

### 观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

### 中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

### 迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

### 访问者（Visitor）模式：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

### 备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

### 解释器（Interpreter）模式：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

# 分类