# 什么是设计模式

设计模式(Design Pattern)是一套被反复使用、多数人知晓的、经过 分类编目的、代码设计经验的总结，使用设计模式是为了可重用代码、让代码更容易被他人理解并且保证代码可靠性。

设计模式一般包含模式名称、问题、目的、解决方案、效果等组成要素，其中关键要素是模式名称、问题、解决方案和效果

# 设计模式的作用

* 设计模式来源众多专家的经验和智慧， 它们是从许多优秀的软件系统中总结出的成功的、能够实现可维护性复用的设计方案 。
* 设计模式提供了一套通用的设计词汇和一种通用的形式来方便开发人员之间沟通和交流，使得设计方案更加通俗易懂 。
* 大部分设计模式都兼顾了系统的可重用性和可扩展性， 这使得我们可以更好地重用一些已有的设计方案、 功能模块甚至一个完整的软件系统， 避免我们经常做一些重复的设计、 编写一些重复的代码 。

# 设计模式的原则

设计模式有7大原则：

* 单一职责原则 (Single Responsibility Principle, SRP) ：个类只负责一个功能领域中的相应职责。或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。实际上，方法的设计上，也应该遵循SRP。
* 接口隔离原则 (Interface Segregation Principle, ISP) ：使用多个专门的接口，而不使用单一的总接口，说白了，接口应该尽可能的小。即客户端不应该依赖那些它不需要的接口。当一个接口太大时，我们需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。
* 依赖倒转原则 (Dependence Inversion Principle, DIP)：抽象不应该依赖于细节，细节应该依赖于抽象 ，强调的是面向接口编程；

在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过依赖注入 (DependencyInjection, DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发 生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：

* 构造注 入：通过构造函数来传入具体类的对象
* 设值注入（Setter注入）：通过Setter方法来传入具体类的对象
* 接口注入：通过在接口中声明的业务方法来传入具体类的对象
* 里氏代换原则 (Liskov Substitution Principle, LSP) ：所有引用基类对象的地方能够透明地使用其子类的对象，LSP和继承的关系非常密切；
* 开闭原则 (Open-Closed Principle, OCP) ： 软件实体应对扩展开放，而对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。
* 合成复用原则 (Composite Reuse Principle, CRP) ：尽量使用对象组合，而不是继承来达到复用的目的
* 迪米特法则 (Law of Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用，其中门面模式就很好的展现了迪米特法则；

# 设计模式分类

## 创建型模式（6个）

* 简单工厂模式
* 工厂方法模式
* 抽象工厂模式
* 单例模式
* 原型模式
* 建造者模式

## 结构型模式（7个）

* 适配器模式
* 桥接模式
* 组合模式
* 装饰模式
* 外观模式
* 享元模式
* 代理模式

## 行为型模式（11个）

* 责任链模式
* 命令模式
* 解释器模式
* 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
* 模板方法模式
* 访问者模式

# 简单工厂模式

## 定义

简单工厂模式(Simple Factory Pattern)：定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。因为通常在简单工厂模式中用于创建实例的方法是 静态(static)方法，因此简单工厂模式又被称为静态工厂方法(Static Factory Method)模式。简单工厂模式属于创建型模式。

## 类图

### 标准类图

为了让工厂能够统一生产产品，这些产品通常会有一个公共的父类，这个类多半被定义成为抽象类或者是接口（当然这个不是必须的），如下Product，然后让各个具体的类去具体实现或者继承Product。



### 简化类图

为了简化简单工厂，可以将抽象的产品类和工厂类合并在一起，形成如下比较简单的方式，由于工厂需要被实例化，否则不能使用，所以Product不再是抽象类或者接口。



## 角色

* Factory（工厂类）：负责创建所有产品的核心逻辑。
* Product（抽象产品类）：工厂类所创建的所有对象的父类，封装了各种产品对象的 公有方法，它引入将提高系统的灵活性，使得在工厂类中只需定义一个通用的工厂方法，因为所有创建的具体产品对象都是其子类对象。
* ConcreteProduct（具体产品角色）：它是简单工厂模式的创建目标

## 核心代码

### 标准模式

|  |
| --- |
| 工厂类：  public class *ProductFactory* {  public static *Product* createProduct(*String arg*) {  if ("A".equals(*arg*)) {  return new ProductA();  } else if ("B".equals(*arg*)) {  return new ProductB();  } else {  throw new RuntimeException("参数错误");  }  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Product* product = *ProductFactory*.createProduct("A");  *System*.***out***.println(product.getName());  } } |

### 简化模式

|  |
| --- |
| 工厂类（产品父类）：  public class *Product* {   /\*\*  \* 需要子类重写的方法  \* @return  \*/  protected *String* getName() {  return null;  }    /\*\*  \* 工厂方法  \* @param arg  \* @return  \*/  public static *Product* createProduct(*String arg*) {  if ("A".equals(*arg*)) {  return new ProductA();  } else if ("B".equals(*arg*)) {  return new ProductB();  } else {  throw new RuntimeException("参数错误");  }  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *Product* product = *Product*.createProduct("A");  *System*.***out***.println(product.getName());  } } |

需要注意的是，这里的Product仍然可以是一个抽象类。

## 总结

### 优点

* 工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的职责，而仅仅“消费”产品，简单工厂模式实现了对象创建和使用的分离。
* 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以在一定程度减少使用者的记忆量。
* 把对象的创建封装在了一个地方，对于较复杂对象的创建，统一进行了管理，避免多个地方分别创建的不一致性；
* 可以通过配置文件参数的方式，在不修改代码的情况下传入arg，从而灵活的构建出Product对象；

### 缺点

* 由于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响。
* 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。
* 简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。
* 对于工厂类而已，每增加一种产品，就需要更改逻辑违背了开闭原则

### 适用场景

* 当需要根据不同的参数创建不同的对象，且他们有共同的抽象方法，只是具体表现行为不同的时候；
* 创建的对象相对较少，这样不会让工厂方法业务过于复杂；

### 开源项目案例

* java.util.Calendar#createCalendar

# 工厂方法模式

## 定义

工厂方法模式(Factory Method Pattern)：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为工厂模式 (Factory Pattern)，又可称作虚拟构造器模式(Virtual Constructor Pattern)或多态工厂模式 (Polymorphic Factory Pattern)，工厂模式属于创建新型模式。

工厂方法模式提供一个抽象工厂接口来声明抽象工厂方法，而由其子类来具体实现工厂方法，创建具体的产品对象。

## 类图

### 标准模式



### 隐藏工厂方法

有时候，为了进一步简化客户端的使用，还可以对客户端隐藏工厂方法，此时，在工厂类中

将直接调用产品类的业务方法。



## 角色

* Product（抽象产品）：它是定义产品的接口，是工厂方法模式所创建对象的超类型，也就
* ConcreteProduct（具体产品）：它实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，具体工厂和具体产品之间一一对应。
* Factory（抽象工厂）：在抽象工厂类中，声明了工厂方法(Factory Method)，用于返回一个产品。抽象工厂是工厂方法模式的核心，所有创建对象的工厂类都必须实现该接口。
* ConcreteFactory（具体工厂）：它是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户端调用，返回一个具体产品类的实例。

## 核心代码

### 标准模式

|  |
| --- |
| 抽象工厂：  public interface *LoggerFactory* {  *Logger* createLogger(); } |
| 具体工厂：  public class *FileLoggerFactory* implements *LoggerFactory*{  @Override  public *Logger* createLogger() {  return new FileLogger();  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {   // 可以通过XML文件来更改具体的工厂类  *LoggerFactory* loggerFactory = new FileLoggerFactory();   *Logger* logger = loggerFactory.createLogger();   logger.writeLog("这是内容");  } } |

### 简化模式

|  |
| --- |
| 抽象工厂：  public abstract class *LoggerFactory* {   /\*\*  \* 工厂方法最好修饰成protected，对客户端进行隐藏  \* @return  \*/  protected abstract *Logger* createLogger();   public void writeLog(*String content*){  *Logger* logger = this.createLogger();  logger.writeLog(*content*);  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *LoggerFactory* loggerFactory = new FileLoggerFactory();  loggerFactory.writeLog("这是内容");  } } |

## 总结

### 优点

* 客户端只需要关心工厂类抽象和产品抽象，不需要关心具体的实现细节；
* 当有新的产品加入的时候，无需修改抽象工厂类和抽象产品类以及客户端，扩展性强；

### 缺点

* 每新增加一种产品，就需要新增一个工厂类，会导致类的数量很多；

### 适用场景

同一个抽象的实现不是很多，且相对固定，对于不同的场景，可能选择不同的实现的时候，比较适合使用工厂方法模式；

### 开源项目案例

* java.util.Collection#iterator
* java.util.Collection#spliterator
* org.springframework.beans.factory.FactoryBean

# 抽象工厂模式

## 定义

抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，抽象工厂模式属于创建型模式。其实说白了，就是对工厂方法模式的一种优化，可以减少工厂类的数据；

同一个工厂类创建出来的对象被称为同一个产品族的；

## 类图



## 角色

* 产品等级结构：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机。
* 产品族：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中 的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品 等级结构中，海尔电冰箱位于电冰箱产品等级结构中，海尔电视机、海尔电冰箱构成了一个产品族。
* AbstractFactory（抽象工厂）：它声明了一组用于创建一族产品的方法，每一个方法对应一种产品。
* ConcreteFactory（具体工厂）：它实现了在抽象工厂中声明的创建产品的方法，生成一组具体产品，这些产品构成了一个产品族，每一个产品都位于某个产品等级结构中。
* AbstractProduct（抽象产品）：它为每种产品声明接口，在抽象产品中声明了产品所具有的业务方法。
* ConcreteProduct（具体产品）：它定义具体工厂生产的具体产品对象，实现抽象产品接口中 声明的业务方法。

## 核心代码

|  |
| --- |
| 抽象工厂：  public interface *SkinFactory* {   *Button* createButton();   *TextField* createTextField();   *ComboBox* createComboBox(); } |
| 具体工厂：  public class *SpringSkinFactory* implements *SkinFactory* {  @Override  public *Button* createButton() {  return new SpringButton();  }   @Override  public *TextField* createTextField() {  return new SpringTextField();  }   @Override  public *ComboBox* createComboBox() {  return new SpringComboBox();  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *SkinFactory* skinFactory = new SpringSkinFactory();   *Button* button = skinFactory.createButton();   button.displayButton();  } } |

## 总结

### 优点

* 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建；
* 当新增同一个产品族的时候，无需更改原有抽象，遵守了“开闭原则”

### 缺点

* 增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，违反了“开闭原则”。

### 适用场景

* 当系统有类似于产品族结构的时候，上面类图所示的结构，有多个产品族，每个产品族有多个产品；
* 产品类型相对固定的时候

### 开源项目案例

* java.util.Collection

# 单例模式

## 定义

单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式属于创建型模式。

单例模式有三个要点：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它 必须自行向整个系统提供这个实例。

## 类图



## 核心代码

单例模式主要有两种模式：饿汉式、懒汉式，写法比较多。

### 饿汉式

|  |
| --- |
| public class *EagerSingleon* {  private static *EagerSingleon* instance = new EagerSingleon();   private EagerSingleon() {  }   public static *EagerSingleon* getInstance() {  return instance;  } } |

这种方式是上来就直接创建，然而，如果这个类用不到的话，那么这个创建是没有意义的；

### 饿汉式变种（静态代码块）

|  |
| --- |
| public class *EagerStaticSingleton* {  private static *EagerStaticSingleton* instance = null;  static{  instance = new EagerStaticSingleton();  }  private EagerStaticSingleton(){};  public static *EagerStaticSingleton* getInstance(){  return instance;  } } |

这种方式和上面的是类似的，在类被加载的时候创建；

### 懒汉式（线程不安全）

|  |
| --- |
| public class *LazySingleton* {  private static *LazySingleton* instance = null;   private LazySingleton() {  }   public static *LazySingleton* getInstance() {  if (instance == null) {  instance = new LazySingleton();  }  return instance;  } } |

把对象的创建延迟到第一次使用的时候，然而，线程不安全，多线程环境下不能使用；

### 懒汉式（线程安全、效率低）

|  |
| --- |
| public class *SafeLazySingleton* {  private static *SafeLazySingleton* instance = null;   private SafeLazySingleton() {  }   public static synchronized *SafeLazySingleton* getInstance() {  if (instance == null) {  instance = new SafeLazySingleton();  }  return instance;  } } |

这种方式通过枷锁的方式让对象的创建，但是创建对象的情况很少，所以，很多时候加锁是没有意义的，效率较低

### 双重校验锁

|  |
| --- |
| public class *DoubleCheckLockSingleton* {  private static *DoubleCheckLockSingleton* instance = null;   private DoubleCheckLockSingleton() {  }   public static *DoubleCheckLockSingleton* getInstance() {  if (instance == null) {  synchronized (*DoubleCheckLockSingleton*.class) {  // 需要再次判断是存在，原因见下面：  if (instance == null) {  instance = new DoubleCheckLockSingleton();  }  }  }  return instance;  } } |

假如在某一瞬间线程A和线程B都在调用getInstance()方法，此时instance对象为null值，均能通 过instance == null的判断。由于实现了synchronized加锁机制，线程A进入synchronized锁定的代 码中执行实例创建代码，线程B处于排队等待状态，必须等待线程A执行完毕后才可以进入 synchronized锁定代码。但当A执行完毕时，线程B并不知道实例已经创建，将继续创建新的实 例，导致产生多个单例对象

### 静态内部类实现

|  |
| --- |
| public class *StaticInnerClassSingleton* {  private static class *SingletonHolder* {  private static final *StaticInnerClassSingleton* ***INSTANCE*** = new StaticInnerClassSingleton();  }   private StaticInnerClassSingleton() {  }   public static final *StaticInnerClassSingleton* getInstance() {  return *SingletonHolder*.***INSTANCE***;  } } |

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，和饿汉式不同的是，饿汉式创建当Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。

### 枚举实现

|  |
| --- |
| public enum *EnumSingleton* {  ***INSTANCE***;  public void doSomething() {  *System*.***out***.println("doSomething");  } } |

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，使用的时候直接通过Singleton.INSTANCE的方式便可以拿到；

## 总结

### 优点

* 单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。
* 由于在系统内存中只存在一个对象，因此可以节约系统资源；

### 缺点

* 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难
* 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色

### 适用场景

* 系统只需要一个实例对象
* 客户调用类的单个实例只允许使用一个公共访问点

### 开源项目案例

* java.lang.Runtime（饿汉模式）
* org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton（双重校验所）
* spring容器中的Bean默认都是单例模式

# 原型模式

## 定义

原型模式(Prototype Pattern)：使用原型实例指定创建对象的种类，并且通过拷贝这些 原型创建新的对象。

需要注意的是通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，通常对克隆所产生的对象进行修改对原型对象不会造成任何影响，每一个克隆对象都是相互独立的。原型模式属于创建型模式

克隆分为两种：深克隆和浅克隆

### 浅克隆

创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。

### 深克隆

创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。

总之深浅克隆都会在堆中新分配一块区域，区别在于对象属性引用的对象是否需要进行克隆（递归性的）

## 角色

* Prototype（抽象原型类）：它是声明克隆方法的接口，是所有具体原型类的公共父类，可以是抽象类也可以是接口，甚至还可以是具体实现类。
* ConcretePrototype（具体原型类）：它实现在抽象原型类中声明的克隆方法，在克隆方法中 返回自己的一个克隆对象。
* Client（客户类）：让一个原型对象克隆自身从而创建一个新的对象，在客户类中只需要直 接实例化或通过工厂方法等方式创建一个原型对象

## 类图



这里给出深克隆和浅克隆的区别图，其中pos值得是当前对象在内存中的地址。



右面的图中，颜色和左边一样的表示对象地址没有变化的，颜色不同则表示对象地址变化了，可以看到，深度克隆，所有的对象地址都不同了，而浅克隆只有最外层的对象地址不同了，但是里面的对象都还是和之前一样，引用的是相同的地址；

## 核心代码

### 浅克隆

对于浅克隆，我们通常是让这个对象类实现Cloneable接口，调用clone方法（这个方法来自Object），

|  |
| --- |
| 最深层次对象：  @Data public class *Dog* implements *Serializable*, *Cloneable* {   private *String* color;   @Override  public *Dog* clone() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }  return (*Dog*) clone;  } } |
| 最外层对象：  @Data public class *Person* implements *Serializable*, *Cloneable* {   private *String* name;   private *Son* son;   public *Person* clone() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }  *Person* result = (*Person*) clone;  return result;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Dog* dog = new Dog();  dog.setColor("black");   *Son* son = new Son();  son.setName("jim");  son.setDog(dog);   *Person* person = new Person();  person.setName("tom");  person.setSon(son);   *Person* person2 = person.clone();  // 打印true，因为地址一样  *System*.***out***.println(person.getSon() == person2.getSon());  // 打印true，因为地址一样  *System*.***out***.println(person2.getSon().getDog() == person.getSon().getDog());  } } |

实际上，我们只需要Person实现Cloneable接口即可；这里让Son和Dog也实现Cloneable，以及多实现了Serializable接口是为了后面的深克隆；

### 深克隆（利用浅克隆完成）

深克隆有两种方式实现，一种方法是每个对象在克隆的时候，除了克隆基础属性，再调用引用类型的clone方法，以Person为例，需要如下实现：

|  |
| --- |
| public *Person* deepClone1() {  *Object* clone = null;  try {  clone = super.clone();  } catch (*CloneNotSupportedException e*) {  *e*.printStackTrace();  }   // 克隆Son，需要调用它的深克隆方法  *Person* result = (*Person*) clone;  *Son* clonePerson = result.getSon().deepClone1();    result.setSon(clonePerson); // 拷贝子对象  return result; } |

这时候，需要调用deepClone1()方法进行克隆，这种克隆的弊端在于，还需要对Son的colne方法进行改造，如果层次较深，改动很多，如果引用类型的属性很多，这种做法会非常复杂

### 深克隆（字节拷贝）

有一种比较简单的方法完成深克隆，就是使用字节拷贝，这种方式适合任何对象，不过需要类实现序列化接口，但是不需要实现Cloneable 接口

|  |
| --- |
| public *Person* deepClone2() {  try {  //将对象写入到流中  *ByteArrayOutputStream* bao = new ByteArrayOutputStream();  *ObjectOutputStream* oos = new ObjectOutputStream(bao);  oos.writeObject(this);   // 从流中读出对象  *ByteArrayInputStream* bai = new ByteArrayInputStream(bao.toByteArray());  *ObjectInputStream* ois = new ObjectInputStream(bai);  return (*Person*) ois.readObject();   } catch (*Exception e*) {  *e*.printStackTrace();  }  return null; } |

这种方式我们只需要对要拷贝的对象添加对应方法即可

## 总结

### 优点

* 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过复制一个 已有实例可以提高新实例的创建效率
* 扩展性较好，由于在原型模式中提供了抽象原型类，在客户端可以针对抽象原型类进行编程，而将具体原型类写在配置文件中，增加或减少产品类对原有系统都没有任何影响
* 可以使用深克隆的方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起 来，以便在需要的时候使用（如恢复到某一历史状态）

### 缺点

* 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了“开闭原则”

### 适用场景

* 创建新对象成本较大，新的对象可以通过原型模式对已有对象进行复制来获得，如果是相似对象，则可以对 其成员变量稍作修改；
* 需要保存历史状态的场景；

### 开源项目

* Spring容器中的Bean，可以使用原型模式
* JDK中的ArrayList，就实现了clone方法，提供了浅拷贝。

# 建造者模式

## 定义

建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示

建造者模式一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于创建型模式。

## 类图

### 标准类图



### 省略指挥者

在有些情况下，为了简化系统结构，可以将Director和抽象建造者Builder进行合并，省略掉指挥者，在Builder 中提供逐步构建复杂产品对象的construct()方法。为了防止子构建者覆盖construct方法，可以考虑用final修饰



## 角色

* Builder（抽象建造者）：它为创建一个产品Product对象的各个部件指定抽象接口，在该接口中一般声明两类方法，一类方法是buildPartX()，它们用于创建复杂对象的各个部件；另一类方法是construct()，用于返回复杂对象。Builder既可以是抽象类，也可以是接口。
* ConcreteBuilder（具体建造者）：它实现了Builder接口，实现各个部件的具体构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象。
* Product（产品角色）：它是被构建的复杂对象，包含多个组成部件，具体建造者创建该产品的内部表示并定义它的装配过程
* Director（指挥者）：指挥者又称为导演类，它负责安排复杂对象的建造次序，指挥者与抽 象建造者之间存在关联关系，可以在其construct()建造方法中调用建造者对象的部件构造与装 配方法，完成复杂对象的建造。客户端一般只需要与指挥者进行交互，在客户端确定具体建 造者的类型，并实例化具体建造者对象（也可以通过配置文件和反射机制），然后通过指挥 者类的构造函数或者Setter方法将该对象传入指挥者类中

## 核心代码

### 标准模式

|  |
| --- |
| 抽象建造者：  public abstract class *ComputerBuilder* {   // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   // 返回构建的结果  public abstract *Computer* getResult(); } |
| 具体建造者：  public class *LOLComputerBuilder* extends *ComputerBuilder* {   private *Computer* computer = new Computer();   @Override  public void buildCpu() {  computer.setCpu("Intel i7");  }   @Override  public void buildHardDisk() {  computer.setHardDisk("SSD 512");  }   @Override  public void buildMemory() {  computer.setMemory("金士顿32G");  }   @Override  public void buildDisplayer() {  computer.setDisplayer("三星32英寸");  }   @Override  public *Computer* getResult() {  return computer;  } } |
| 指挥者：  public class *ComputerDirector* {   // 引入构建者  private *ComputerBuilder* computerBuilder;   public ComputerDirector(*ComputerBuilder computerBuilder*) {  this.computerBuilder = *computerBuilder*;  }   // 构建出产品  public *Computer* construct() {  computerBuilder.buildCpu();  computerBuilder.buildDisplayer();  computerBuilder.buildHardDisk();  computerBuilder.buildMemory();  return computerBuilder.getResult();  }    public *ComputerBuilder* getComputerBuilder() {  return computerBuilder;  }   public void setComputerBuilder(*ComputerBuilder computerBuilder*) {  this.computerBuilder = *computerBuilder*;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  // 可以通过配置文件的方式灵活更改Builder  *ComputerBuilder* builder = new LOLComputerBuilder();   *ComputerDirector* director = new ComputerDirector(builder);   *Computer* computer = director.construct();   *System*.***out***.println(computer);   } } |

说明：

* 如果所有的构建者构建出来的对象是相同的，那么我们实际上可以把new对象的过程放到抽象构建者里面；
* 如果构造出来的产品不止一种，那么我们需要对多种产品进行统一抽象。

### 省略Director

|  |
| --- |
| 抽象构建者：  public abstract class *ComputerDirectorBuilder* {   // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   public abstract *Computer* getResult();   // 构建并返回结果  public final *Computer* construct() {  this.buildCpu();  this.buildDisplayer();  this.buildHardDisk();  this.buildMemory();  return this.getResult();  }  } |
| 具体构建者：  public class *LOLComputerDirectorBuilder* extends *ComputerDirectorBuilder* {   protected *Computer* computer = new Computer();   @Override  public void buildCpu() {  computer.setCpu("Intel i7");  }   @Override  public void buildHardDisk() {  computer.setHardDisk("SSD 512");  }   @Override  public void buildMemory() {  computer.setMemory("金士顿32G");  }   @Override  public void buildDisplayer() {  computer.setDisplayer("三星32英寸");  }   @Override  public *Computer* getResult() {  return computer;  }  } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *ComputerDirectorBuilder* directorBuilder = new LOLComputerDirectorBuilder();   *Computer* computer = directorBuilder.construct();   *System*.***out***.println(computer);   } } |

### 钩子函数（往模板方法的过渡）

在某些场景下，可能不同的构建者略有不同，比如，某些产品没有某个部分，这时候，我们可以通过钩子函数来进行控制，如：

|  |
| --- |
| 抽象构建者：  public abstract class *ComputerDirectorBuilder* {    // 下面是各个组成部分的构建方法  public abstract void buildCpu();   public abstract void buildHardDisk();   public abstract void buildMemory();   public abstract void buildDisplayer();   /\*\*  \* 钩子函数  \*  \* @return  \*/  public boolean needCpu() {  return true;  }   public abstract *Computer* getResult();   // 构建并返回结果  public *Computer* construct() {  if (needCpu()) {  this.buildCpu();  }   this.buildDisplayer();  this.buildHardDisk();  this.buildMemory();  return this.getResult();  }  } |
| 具体构建者：  public class *LOLComputerDirectorBuilder* extends *ComputerDirectorBuilder* {   protected *Computer* computer = new Computer();   @Override  public void buildCpu() {  computer.setCpu("Intel i7");  }   @Override  public void buildHardDisk() {  computer.setHardDisk("SSD 512");  }   @Override  public void buildMemory() {  computer.setMemory("金士顿32G");  }   @Override  public void buildDisplayer() {  computer.setDisplayer("三星32英寸");  }   /\*\*  \* 覆盖钩子函数  \* @return  \*/  @Override  public boolean needCpu() {  return false;  }   @Override  public *Computer* getResult() {  return computer;  }  } |

### 链式调用

在实际应用的时候，很多情况会直接省略掉指挥者、直接使用Builder来对对象进行组装。具体使用时在每个组件装配的过程中都返回自身，实现链式调用。

|  |
| --- |
| 产品：  public class *ComputerBuilder* {   /\*\*  \* cpu  \*/  private *String* cpu;   /\*\*  \* 内存  \*/  private *String* memory;    /\*\*  \* 硬盘  \*/  private *String* hardDisk;   public *ComputerBuilder* buildCpu(*String cpu*){  this.cpu = *cpu*;  return this;  }   public *ComputerBuilder* buildMemory(*String memory*){  this.memory = *memory*;  return this;  }   public *ComputerBuilder* buildeHardDisk(*String hardDisk*){  this.hardDisk = *hardDisk*;  return this;  }  } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *ComputerBuilder* computerBuilder = new ComputerBuilder().buildCpu("inter i7").buildeHardDisk("SSD").buildMemory("32G");  *System*.***out***.println(computerBuilder);  } } |

## 总结

### 优点

* 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解 耦，使得相同的创建过程可以创建不同的产品对象
* 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体 建造者或增加新的具体建造者

### 缺点

* 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致 系统变得很庞大，增加系统的理解难度和运行成本。

### 使用场景

* 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性
* 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

## 开源项目案例

* StringBuilder的append方法
* Spring中的BeanDefinitionBuilder

# 适配器模式

## 定义

适配器模式(Adapter Pattern)：将一个接口转换成客户希望的另一个接口，使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。适配器模式是一种结构性模式

适配器模式分为如下几种：

* 对象适配器：适配器和适配者之间是关联关系
* 类适配器：适配器和适配者之间是继承（实现）关系
* 双向适配器：适配者和目标对象可以互相调用
* 缺省适配器：解决不需要实现全部接口方法的场景

## 类图

### 对象适配器



这里的Target可以是接口，也可以是抽象类

### 类适配器



由于Adapter要继承Adaptee，所有，Target只能是接口

### 双向适配器



### 缺省适配器



## 角色

* Target（目标抽象类）：目标抽象类定义客户所需接口，可以是一个抽象类或接口，也可以是具体类。
* Adapter（适配器类）：适配器可以调用另一个接口，作为一个转换器，对Adaptee和Target进 行适配，适配器类是适配器模式的核心，在对象适配器中，它通过继承Target并关联一个 Adaptee对象使二者产生联系。
* Adaptee（适配者类）：适配者即被适配的角色，它定义了一个已经存在的接口，这个接口 需要适配，适配者类一般是一个具体类，包含了客户希望使用的业务方法，在某些情况下可 能没有适配者类的源代码。

## 核心代码

### 对象适配器

|  |
| --- |
| public class *Adapter* implements *Target* {   private *Adaptee* adaptee;   public Adapter(*Adaptee adaptee*) {  this.adaptee = *adaptee*;  }   @Override  public *String* sayHelloUpCase(*String name*) {  return adaptee.sayHello(*name*).toUpperCase();  } } |

### 类适配器

|  |
| --- |
| public class *Adapter* extends *Adaptee* implements *Target* {   @Override  public *String* sayHelloUpCase(*String name*) {  return sayHello(*name*).toUpperCase();  } } |

要求目标是一个接口。

### 双向适配器

|  |
| --- |
| public class *Adapter* implements *Target*,*Adaptee* {  private *Adaptee* adaptee;  private *Target* target;   public Adapter(*Adaptee adaptee*) {  this.adaptee = *adaptee*;  }   public Adapter(*Target target*) {  this.target = *target*;  }   @Override  public *String* sayHello(*String name*) {  return target.sayHelloUpCase(*name*).toLowerCase();  }   @Override  public *String* sayHelloUpCase(*String name*) {  return adaptee.sayHello(*name*).toUpperCase();  } } |

### 缺省适配器

其实就是定义一个抽象类对接口的方法（全部或者部分）给出默认实现，用户可以通过继承这个抽象类从而减少实现不必要的接口

## 总结

### 优点

* 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构
* 增加了类的透明性和复用性，将具体的业务实现过程封装在适配者类中，对于客户端类而言是透明的，而且提高了适配者的复用性

### 缺点

* 由于JAVA的单继承特性，一次最多只能适配一个适配者类，不能同时适配多个适配者
* 类适配器模式下，适配者类不能为最终类，目标类只能是接口
* 对象适配器的实现上比类适配器要麻烦一些；

### 使用场景

* 系统需要使用一些现有的类，而这些类的接口（如方法名）不符合系统的需要，甚至没有这些类的源代码。
* 想创建一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作
* 接口方法过多，而我们只想实现其中的某几个方法的时候，可以使用缺省适配器

# 桥接模式

## 定义

桥接模式(Bridge Pattern)：将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是 一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

## 类图



## 角色

* Abstraction（抽象类）：用于定义抽象类的接口，它一般是抽象类而不是接口，其中定义了 一个Implementor（实现类接口）类型的对象并可以维护该对象，它与Implementor之间具有关 联关系，它既可以包含抽象业务方法，也可以包含具体业务方法。
* RefinedAbstraction（扩充抽象类）：扩充由Abstraction定义的接口，通常情况下它不再是抽象类而是具体类，它实现了在Abstraction中声明的抽象业务方法，在RefinedAbstraction中可以 调用在Implementor中定义的业务方法。
* Implementor（实现类接口）：定义实现类的接口，这个接口不一定要与Abstraction的接口完 全一致，事实上这两个接口可以完全不同，一般而言，Implementor接口仅提供基本操作，而 Abstraction定义的接口可能会做更多更复杂的操作
* ConcreteImplementor（具体实现类）：具体实现Implementor接口，在不同的ConcreteImplementor中提供基本操作的不同实现

## 核心代码

|  |
| --- |
| 抽象部分抽象：  @Data public abstract class *Brush* {  protected *Color* color;   public abstract *String* draw(); } |
| 扩充的抽象部分：  public class *BigBrush* extends *Brush* {  @Override  public *String* draw() {  return "大号画笔，颜色是：" + color.getColor();  } } |
| 实现部分抽象：  public interface *Color* {  *String* getColor(); } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Brush* brush = new BigBrush();  brush.setColor(new YelloColor());  *System*.***out***.println(brush.draw());  } } |

## 总结

### 优点

* 分离抽象接口及其实现部分
* 在很多情况下，桥接模式可以取代多层继承方案
* 提高了系统的扩展性

### 缺点

* 桥接模式的使用会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程。
* 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性，如何正确识别两个独立维度也需要一定的经验积累

### 适用场景

* 一个类存在两个（或多个）独立变化的维度，且这两个（或多个）维度都需要独立进行扩展。
* 对于那些不希望使用继承或因为多层继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

# 组合模式

## 定义

组合模式(Composite Pattern)：组合多个对象形成树形结构以表示具有“整体—部分”关系的层次结构。组合模式对单个对象（即叶子对象）和组合对象（即容器对象）的使用具有一致性，组合模式又可以称为“整体—部分”(Part-Whole)模式，它是一种对象结构型模式。

## 类图

### 透明模式

在透明模式中，抽象构建一视同仁，定义了容器构件和叶子构建具备的所有方法，这样，叶子构建和抽象构建就具有相同的方法。只不过，有些方法在叶子构件或者容器构件中可能不能正常工作。



### 安全模式

为了让客户端不会调用到错误的方法，安全模式下，抽象构件中只抽象了叶子构件和容器构件中都存在的方法。



## 角色

* Component（抽象构件）：它可以是接口或抽象类，为叶子构件和容器构件对象声明接口， 在该角色中可以包含所有子类共有行为的声明和实现。在抽象构件中定义了访问及管理它的子构件的方法，如增加子构件、删除子构件、获取子构件等。
* Leaf（叶子构件）：它在组合结构中表示叶子节点对象，叶子节点没有子节点，它实现了在抽象构件中定义的行为。对于那些访问及管理子构件的方法，可以通过异常等方式进行处理。
* Composite（容器构件）：它在组合结构中表示容器节点对象，容器节点包含子节点，其子节点可以是叶子节点，也可以是容器节点，它提供一个集合用于存储子节点，实现了在抽象构件中定义的行为，包括那些访问及管理子构件的方法，在其业务方法中可以递归调用其子节点的业务方法。

## 核心代码

### 透明模式

|  |
| --- |
| 抽象构件：  public abstract class *AbstractFile* {  private *String* name;  public AbstractFile(*String name*) {  this.name = *name*;  }  // 添加，容器组件才有的方法  public abstract void add(*AbstractFile abstractFile*);  //删除，容器组件才有的方法  public abstract void remove(*AbstractFile abstractFile*);  // 获取名字，容器组件和叶子组件都有的方法  public abstract void display();   public *String* getName() {  return name;  } } |
| 叶子构件：  public class *TextFile* extends *AbstractFile* {   public TextFile(*String name*) {  super(*name*);  }   public void add(*AbstractFile abstractFile*) {  throw new RuntimeException("不支持添加");  }   public void remove(*AbstractFile abstractFile*) {  throw new RuntimeException("不支持删除");  }   @Override  public void display() {  *System*.***out***.println("文本文件：" + getName());  } } |
| 容器构件：  public class *Folder* extends *AbstractFile* {   private *List*<*AbstractFile*> files = new ArrayList<>();   public Folder(*String name*) {  super(*name*);  }   @Override  public void add(*AbstractFile abstractFile*) {  files.add(*abstractFile*);  }   @Override  public void remove(*AbstractFile abstractFile*) {  files.remove(*abstractFile*);  }   @Override  public void display() {  *System*.***out***.println("文件夹：" + getName());  for (*AbstractFile* file : files) {  file.display();  }  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {   *AbstractFile* folder = new Folder("所有文件");   *AbstractFile* img1 = new ImageFile("刘兴.jpg");  *AbstractFile* img2 = new ImageFile("郭智慧.png");  *AbstractFile* imgFolder = new Folder("图片");  imgFolder.add(img1);  imgFolder.add(img2);   *AbstractFile* txtFile = new TextFile("情书.txt");  *AbstractFile* txtFolder = new Folder("情书");  txtFolder.add(txtFile);   folder.add(imgFolder);  folder.add(txtFolder);   folder.display();   } } |

### 透明模式优化版

有时候，如果方法不被支持的可能性较大，我们可以在抽象构件中提供默认实现，这样可以减少子类的实现，如：

|  |
| --- |
| public abstract class *AbstractFile* {  private *String* name;  public AbstractFile(*String name*) {  this.name = *name*;  }  // 添加，容器组件才有的方法  public void add(*AbstractFile abstractFile*){  throw new RuntimeException("不支持添加");  }  //删除，容器组件才有的方法  public void remove(*AbstractFile abstractFile*){  throw new RuntimeException("不支持删除");  }  // 获取名字，容器组件和叶子组件都有的方法  public abstract void display();   public *String* getName() {  return name;  } } |

### 安全模式

|  |
| --- |
| 抽象构件：  public abstract class *AbstractFile* {  private *String* name;  public AbstractFile(*String name*) {  this.name = *name*;  }   // 获取名字，容器组件和叶子组件都有的方法  public abstract void display();   public *String* getName() {  return name;  } } |
| 容器构件：  public class *Folder* extends *AbstractFile* {   private *List*<*AbstractFile*> files = new ArrayList<>();   public Folder(*String name*) {  super(*name*);  }   public void add(*AbstractFile abstractFile*) {  files.add(*abstractFile*);  }   public void remove(*AbstractFile abstractFile*) {  files.remove(*abstractFile*);  }   @Override  public void display() {  *System*.***out***.println("文件夹：" + getName());  for (*AbstractFile* file : files) {  file.display();  }  } } |
| 叶子构建：  public class *TextFile* extends *AbstractFile* {   public TextFile(*String name*) {  super(*name*);  }    @Override  public void display() {  *System*.***out***.println("文本文件：" + getName());  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {   *Folder* folder = new Folder("所有文件");   *ImageFile* img1 = new ImageFile("刘兴.jpg");  *ImageFile* img2 = new ImageFile("郭智慧.png");  *Folder* imgFolder = new Folder("图片");  imgFolder.add(img1);  imgFolder.add(img2);   *TextFile* txtFile = new TextFile("情书.txt");  *Folder* txtFolder = new Folder("情书");  txtFolder.add(txtFile);   folder.add(imgFolder);  folder.add(txtFolder);   folder.display();   } } |

## 总结

### 优点

* 组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，它让客户端忽略了层次的差异，方便对整个层次结构进行控制
* 客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码

### 缺点

在增加新构件时很难对容器中的构件类型进行限制。

### 适用场景

* 在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们
* 在一个使用面向对象语言开发的系统中需要处理一个树形结构。

# 装饰模式

## 定义

装饰模式(Decorator Pattern)：动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。装饰模式是一种对象结构型模式。

## 类图



## 角色

* Component（抽象构件）：它是具体构件和抽象装饰类的共同父类，声明了在具体构件中实 现的业务方法，它的引入可以使客户端以一致的方式处理未被装饰的对象以及装饰之后的对象，实现客户端的透明操作。
* ConcreteComponent（具体构件）：它是抽象构件类的子类，用于定义具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）
* Decorator（抽象装饰类）：它也是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现。它维护一个指向抽象构件对象的引用，通过该引用可以调用装饰之前构件对象的方法，并通过其子类扩展该方法，以达到装饰的目的。
* ConcreteDecorator（具体装饰类）：它是抽象装饰类的子类，负责向构件添加新的职责。每一个具体装饰类都定义了一些新的行为，它可以调用在抽象装饰类中定义的方法，并可以增加新的方法用以扩充对象的行为。

## 核心代码

### 透明装饰模式

|  |
| --- |
| 抽象构件：  public abstract class *Component* {  public abstract void display(); } |
| 抽象装饰者：  public class *ComponentDeretor* extends *Component* {   private *Component* component;   public ComponentDeretor(*Component component*) {  this.component = *component*;  }   @Override  public void display() {  component.display(); // 只是简单的调用构件的原始方法  } } |
| 具体装饰者：  public class *ScrollBarDerector* extends *ComponentDeretor* {   public ScrollBarDerector(*Component component*) {  super(*component*);  }   @Override  public void display() {  super.display();  this.setScrollBar();  }   public void setScrollBar() {  *System*.***out***.println("设置滚动条");  }  } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Component* textField = new TextField();  textField.display();  *System*.***out***.println("-----------");  *Component* scrollTxtField = new ScrollBarDerector(textField);  scrollTxtField.display();  } } |

### 半透明装饰模式

有时候，客户端需要单独的调用具体装饰类的方法，那么此时，我们就需要在客户端直接声明具体装饰者（不透明），而构件则直接使用抽象类（透明），这种情况只是客户端在使用上的不同。

|  |
| --- |
| public class *Client* {   public static void main(*String*[] *args*) {   *Component* textField = new TextField();  textField.display();  *System*.***out***.println("-----------");  *ScrollBarDerector* scrollTxtField = new ScrollBarDerector(textField);  scrollTxtField.setScrollBar();  } } |

## 总结

### 优点

* 对于扩展一个对象的功能，装饰模式比继承更加灵活性，不会导致类的个数急剧增加；
* 可以对一个对象进行多次装饰，通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合，得到功能更为强大的对象。

### 缺点

* 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同
* 装饰模式提供了一种比继承更加灵活机动的解决方案，但同时也意味着比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为繁琐

### 适用场景

* 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
* 当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式

# 外观模式

## 定义

外观模式：为子系统中的一组接口提供一个统一的入口。外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用

## 类图



## 角色

* Facade（外观角色）：在客户端可以调用它的方法，在外观角色中可以知道相关的（一个 或者多个）子系统的功能和责任；在正常情况下，它将所有从客户端发来的请求委派到相应 的子系统去，传递给相应的子系统对象处理
* SubSystem（子系统角色）：在软件系统中可以有一个或者多个子系统角色，每一个子系统可以不是一个单独的类，而是一个类的集合，它实现子系统的功能；每一个子系统都可以被客户端直接调用，或者被外观角色调用，它处理由外观类传过来的请求；子系统并不知道外观的存在，对于子系统而言，外观角色仅仅是另外一个客户端而已

## 核心代码

|  |
| --- |
| 业务子系统：  public class *FileReader* {   public *String* readFileContent(*String fileName*){  *System*.***out***.println("read content from file , fileName="+*fileName*);  return "content";  } } |
| 外观类：  public class *FileEncoderFacade* {   private *FileReader* fileReader = new FileReader();  private *StringEncoder* stringEncoder = new StringEncoder();  private *FileWriter* fileWriter = new FileWriter();   public void encodeFile(*String srcFileName*,*String destFileName*){  *String* content = fileReader.readFileContent(*srcFileName*);  *String* encode = stringEncoder.encode(content);  fileWriter.writeFile(encode,*destFileName*);  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *FileEncoderFacade* fileEncoder = new FileEncoderFacade();  fileEncoder.encodeFile("aaa.txt","bbb.txt");  } } |

## 总结

### 优点

* 它对客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易。通过引入外观模式，客户端代码将变得很简单，与之关联的对象也很少
* 它实现了子系统与客户端之间的松耦合关系，这使得子系统的变化不会影响到调用它的客户端，只需要调整外观类即可。
* 一个子系统的修改对其他子系统没有任何影响，而且子系统内部变化也不会影响到外观对象。

### 缺点

* 不能很好地限制客户端直接使用子系统类，如果对客户端访问子系统类做太多的限制则减 少了可变性和灵活 性。
* 如果设计不当，增加新的子系统可能需要修改外观类的源代码，违背了开闭原则。

### 适用场景

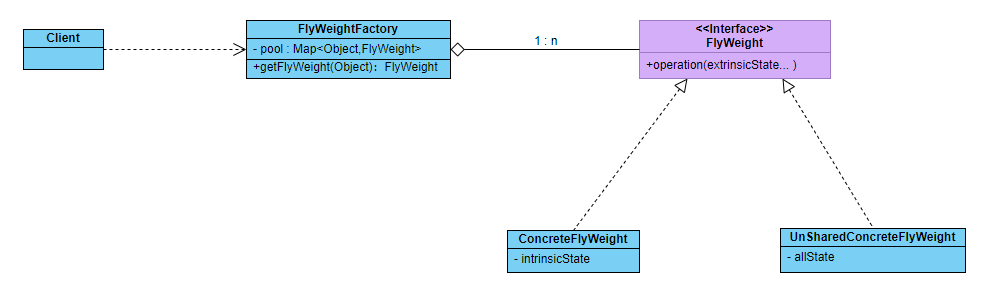
* 当要为访问一系列复杂的子系统提供一个简单入口时可以使用外观模式
* 客户端程序与多个子系统之间存在很大的依赖性。引入外观类可以将子系统与客户端解耦，从而提高子系统的独立性和可移植性
* 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

# 享元模式

## 定义

享元模式(Flyweight Pattern)：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式，它是一种对象结构型模式。

## 类图



## 角色

* 内部状态：是存储在享元对象内部并且不会随环境改变而改变的状态，内部状态可以共享；
* 外部状态是随环境改变而改变的、不可以共享的状态。享元对象的外部状态通常由客户端 保存，并在享元对象被创建之后，需要使用的时候再传入到享元对象内部。一个外部状态与 另一个外部状态之间是相互独立的。
* Flyweight（抽象享元类）：通常是一个接口或抽象类，在抽象享元类中声明了具体享元类公 共的方法，这些方法可以向外界提供享元对象的内部数据（内部状态），同时也可以通过这 些方法来设置外部数据（外部状态）
* ConcreteFlyweight（具体享元类）：它实现了抽象享元类，其实例称为享元对象；在具体享 元类中为内部状态提供了存储空间。通常我们可以结合单例模式来设计具体享元类，为每一 个具体享元类提供唯一的享元对象。
* UnsharedConcreteFlyweight（非共享具体享元类）：并不是所有的抽象享元类的子类都需要 被共享，不能被共享的子类可设计为非共享具体享元类；当需要一个非共享具体享元类的对象时可以直接通过实例化创建
* FlyweightFactory（享元工厂类）：享元工厂类用于创建并管理享元对象，它针对抽象享元 类编程，将各种类型的具体享元对象存储在一个享元池中，享元池一般设计为一个存储“键值 对”的集合（也可以是其他类型的集合），可以结合工厂模式进行设计；当用户请求一个具体 享元对象时，享元工厂提供一个存储在享元池中已创建的实例或者创建一个新的实例（如果 不存在的话），返回新创建的实例并将其存储在享元池中。

## 核心代码

|  |
| --- |
| 抽象享元类：  public abstract class *IgoChessman* {  public abstract *String* getColor();   /\*\*  \* 操作方法，可能会传入外部状态属性，如这里的position  \*  \* @param position  \*/  public void display(*Position position*) {  *System*.***out***.println("棋子颜色：" + getColor() + "，棋子位置：x=" + *position*.getX() + ",y=" + *position*.getY());  } } |
| 具体享元类：  public class *WhitIgoChessman* extends *IgoChessman* {  @Override  public *String* getColor() {  return "白色";  } } |
| 享元工厂（可以考虑使用单例实现，我这里是使用简单工厂实现）：  public class *IgoChessmanFactory* {   private static final *Map*<*String*, *IgoChessman*> ***pool*** = new ConcurrentHashMap<>();   static {  ***pool***.put("black", new BlackIgoChessman());  ***pool***.put("white", new WhitIgoChessman());  }    public static *IgoChessman* getIgoChessman(*String color*) {  return ***pool***.get(*color*);  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *IgoChessman* black = *IgoChessmanFactory*.getIgoChessman("black");   // 这里传入外部状态  black.display(new Position(12,3));  } } |

## 总结

### 优点

* 可以极大减少内存中对象的数量，使得相同或相似对象在内存中只保存一份，从而可以节 约系统资源，提高系统性能
* 享元模式的外部状态相对独立，而且不会影响其内部状态，从而使得享元对象可以在不同 的环境中被共享

### 缺点

* 享元模式使得系统变得复杂，需要分离出内部状态和外部状态，这使得程序的逻辑复杂化。
* 为了使对象可以共享，享元模式需要将享元对象的部分状态外部化，而读取外部状态将使得运行时间变长。

### 适用场景

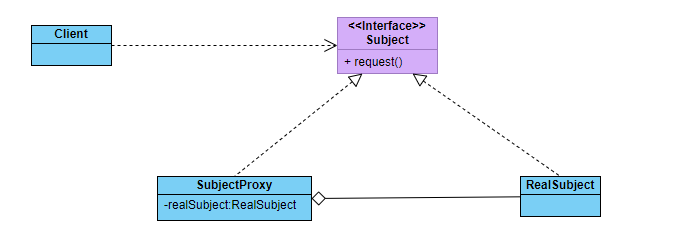
* 一个系统有大量相同或者相似的对象，造成内存的大量耗费。
* 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中
* 在使用享元模式时需要维护一个存储享元对象的享元池，而这需要耗费一定的系统资源， 因此，应当在需要多次重复使用享元对象时才值得使用享元模式。

# 代理模式

## 定义

代理模式：给某一个对象提供一个代理或占位符，并由代理对象来控制对原对象的访问。

## 类图



## 角色

* Subject（抽象主题角色）：它声明了真实主题和代理主题的共同接口，这样一来在任何使 用真实主题的地方都可以使用代理主题，客户端通常需要针对抽象主题角色进行编程。
* Proxy（代理主题角色）：它包含了对真实主题的引用，从而可以在任何时候操作真实主题 对象；在代理主题角色中提供一个与真实主题角色相同的接口，以便在任何时候都可以替代真实主题；代理主题角色还可以控制对真实主题的使用，负责在需要的时候创建和删除真实 主题对象，并对真实主题对象的使用加以约束。通常，在代理主题角色中，客户端在调用所 引用的真实主题操作之前或之后还需要执行其他操作，而不仅仅是单纯调用真实主题对象中 的操作。
* RealSubject（真实主题角色）：它定义了代理角色所代表的真实对象，在真实主题角色中实现

## 代理分类

从Java实现角度上看分为：

* 静态代理
* 动态代理（jdk代理、接口代理）
* cglib代理（子类代理）：它是在内存中构建一个子类对象从而实现对目标对象功能的扩展

从应用角度分为：

* 远程代理(Remote Proxy)：为一个位于不同的地址空间的对象提供一个本地的代理对象，这 个不同的地址空间可以是在同一台主机中，也可是在另一台主机中，远程代理又称为大使 (Ambassador)
* 虚拟代理(Virtual Proxy)：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较 小的对象来表示，真实对象只在需要时才会被真正创建
* 保护代理(Protect Proxy)：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限
* 缓冲代理(Cache Proxy)：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
* 智能引用代理(Smart Reference Proxy)：当一个对象被引用时，提供一些额外的操作，例如 将对象被调用的次数记录下来等

## 核心代码

### 静态代理

|  |
| --- |
| 抽象主题：  public interface *Searcher* {   *String* searchByName(*String username*); } |
| 真实主题：  public class *FileSercher* implements *Searcher* {  @Override  public *String* searchByName(*String username*) {  *System*.***out***.println("search from file");  return *username*.toUpperCase() + " has find !";  } } |
| 代理主题：  public class *SearcherProxy* implements *Searcher* {   private *FileSercher* fileSercher = new FileSercher();    @Override  public *String* searchByName(*String username*) {  // 1. 验证  if (*username* == null) {  throw new RuntimeException("username can't be null !");  }  // 2.记录日志  *System*.***out***.println("start search");   // 调用真实方法  *String* result = fileSercher.searchByName(*username*);   *System*.***out***.println("finish search, result = " + result);  return result;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] args) {  SearcherProxy searcherProxy = new SearcherProxy();  searcherProxy.searchByName("zhangsan!");  } } |

### 动态代理

|  |
| --- |
| 抽象主题：  public interface *Searcher* {   *String* searchByName(*String username*); } |
| 真实主题：  public class *FileSercher* implements *Searcher* {  @Override  public *String* searchByName(*String username*) {  *System*.***out***.println("search from file");  return *username*.toUpperCase() + " has find !";  } } |
| 代理工厂（可以为各种类型的真实主题生成代理）：  public class *ProxyFactory* {   //维护一个目标对象  private *Object* target;   public ProxyFactory(*Object target*) {  this.target = *target*;  }   //给目标对象生成代理对象  public *Object* getProxyInstance() {  return *Proxy*.newProxyInstance(  target.getClass().getClassLoader(),  target.getClass().getInterfaces(),  new *InvocationHandler*() {  @Override  public *Object* invoke(*Object proxy*, *Method method*, *Object*[] *args*) throws *Throwable* {  *System*.***out***.println("开始执行, args = "+ *Arrays*.toString(*args*));  //执行目标对象方法  *Object* returnValue = *method*.invoke(target, *args*);  *System*.***out***.println("结束执行");  return returnValue;  }  }  );  }  } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *FileSercher* fileSercher = new FileSercher();  *ProxyFactory* proxyFactory = new ProxyFactory(fileSercher);   *Searcher* searcher = (*Searcher*)proxyFactory.getProxyInstance();  searcher.searchByName("李四");  } } |

需要注意的是，JDK代理要求被代理对象需要实现了某个接口

### cglib代理

|  |
| --- |
| 引入依赖：  <dependency>  <groupId>cglib</groupId>  <artifactId>cglib</artifactId>  <version>3.3.0</version> </dependency> |
| 目标对象（不需要实现接口）： |
| public class *FileSercher* {  public *String* searchByName(*String username*) {  *System*.***out***.println("search from file");  return *username*.toUpperCase() + " has find !";  } } |
| 代理工厂：  public class *ProxyFactory* implements *MethodInterceptor* {  //维护目标对象  private *Object* target;   public ProxyFactory(*Object target*) {  this.target = *target*;  }   //给目标对象创建一个代理对象  public *Object* getProxyInstance() {  //1.工具类  *Enhancer* en = new Enhancer();  //2.设置父类  en.setSuperclass(target.getClass());  //3.设置回调函数  en.setCallback(this);  //4.创建子类(代理对象)  return en.create();   }   @Override  public *Object* intercept(*Object obj*, *Method method*, *Object*[] *args*, *MethodProxy proxy*) throws *Throwable* {  *System*.***out***.println("开始执行...");   //执行目标对象的方法  *Object* returnValue = *method*.invoke(target, *args*);   *System*.***out***.println("结束执行...");   return returnValue;  } } |
|  |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *FileSercher* fileSercher = new FileSercher();   *FileSercher* proxy = (*FileSercher*) new ProxyFactory(fileSercher).getProxyInstance();  *String* result = proxy.searchByName("王五");  *System*.***out***.println(result);  } } |

## 总结

### 优点

* 能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。
* 客户端可以针对抽象主题角色进行编程，增加和更换代理类无须修改源代码，符合开闭原 则，系统具有较好的灵活性和可扩展性。

### 缺点

* 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求 的处理速度变慢，例如保护代理
* 实现代理模式需要额外的工作，而且有些代理模式的实现过程较为复杂，例如远程代理。

### 适用场景

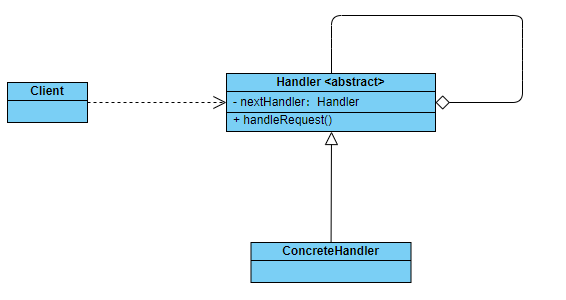
* 当客户端对象需要访问远程主机中的对象时可以使用远程代理
* 当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、 缩短运行时间时可以使用虚拟代理，
* 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问 这些结果时可以使用缓冲代理
* 当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用保护代理。
* 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理

# 责任链模式

## 定义

职责链模式(Chain of Responsibility Pattern)：避免请求发送者与接收者 耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传 递请求，直到有对象处理它为止。职责链模式是一种对象行为型模式。

## 类图



## 角色

* Handler（抽象处理者）：它定义了一个处理请求的接口，一般设计为抽象类，由于不同的 具体处理者处理请求的方式不同，因此在其中定义了抽象请求处理方法。因为每一个处理者 的下家还是一个处理者，因此在抽象处理者中定义了一个抽象处理者类型的对象；
* ConcreteHandler（具体处理者）：它是抽象处理者的子类，可以处理用户请求，在具体处理 者类中实现了抽象处理者中定义的抽象请求处理方法，在处理请求之前需要进行判断，看是 否有相应的处理权限，如果可以处理请求就处理它，否则将请求转发给后继者；在具体处理 者中可以访问链中下一个对象，以便请求的转发。

## 责任链模式分类

### 纯的职责链模式

一个纯的职责链模式要求一个具体处理者对象只能在两个行为中选择一个：要么承担全部责 任，要么将责任推给下家，不允许出现某一个具体处理者对象在承担了一部分或全部责任后 又将责任向下传递的情况

### 不纯的责任链模式

在一个不纯的职责链模式中允许某个请求被一个具体处理者部分处理后再向下传递，或者一 个具体处理者处理完某请求后其后继处理者可以继续处理该请求，而且一个请求可以最终不 被任何处理者对象所接收

## 核心代码

|  |
| --- |
| 抽象处理者：  public abstract class *StringValidator* {   protected *StringValidator* nextHandler;   public StringValidator(*StringValidator nextHandler*) {  this.nextHandler = *nextHandler*;  }   public abstract boolean verify(*String str*); } |
| 具体处理者：  public class *StringNotNullValidator* extends *StringValidator* {   public StringNotNullValidator(*StringValidator validator*) {  super(*validator*);  }   @Override  public boolean verify(*String str*) {  *System*.***out***.println("verify not null");  if (*str* == null) {  return false;  }  if (nextHandler != null) {  return nextHandler.verify(*str*);  }  return false;  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *StringValidator* stringValidator = new StringNotNullValidator(new StringLengthValidator(new PrefixStringValidator(null, "he"), 5));  boolean result = stringValidator.verify("hello");  *System*.***out***.println(result);  } } |
|  |

## 总结

### 优点

* 职责链模式使得一个对象无须知道是其他哪一个对象处理其请求，对象仅需知道该请求会 被处理即可，降低了系统的耦合度；
* 请求处理对象仅需维持一个指向其后继者的引用，而不需要维持它对所有的候选处理者的 引用，可简化对象的相互连接
* 在给对象分派职责时，职责链可以给我们更多的灵活性，可以通过在运行时对该链进行动 态的增加或修改来增加或改变处理一个请求的职责
* 在系统中增加一个新的具体请求处理者时无须修改原有系统的代码，只需要在客户端重新 建链即可，从这一点来看是符合“开闭原则”的。

### 缺点

* 由于一个请求没有明确的接收者，那么就不能保证它一定会被处理，该请求可能一直到链 的末端都得不到处理
* 对于比较长的职责链，请求的处理可能涉及到多个处理对象，系统性能将受到一定影响， 而且在进行代码调试时不太方便

### 适用场景

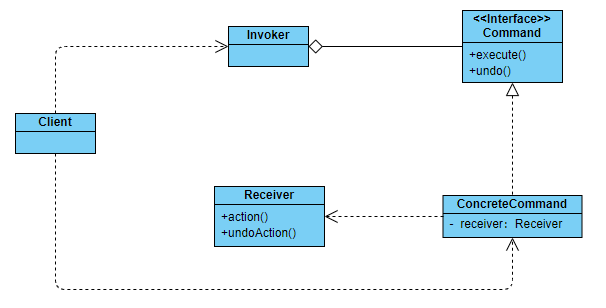
* 有多个对象可以处理同一个请求，具体哪个对象处理该请求待运行时刻再确定，客户端只 需将请求提交到链上，而无须关心请求的处理对象是谁以及它是如何处理的。
* 可动态指定一组对象处理请求，客户端可以动态创建职责链来处理请求，还可以改变链中 处理者之间的先后次序
* 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求

# 命令模式

## 定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象 行为型模式，其别名为动作(Action)模式或事务(Transaction)模式

## 类图



通常，我们的命令都会有撤销动作，但是这不是一定的。

## 角色

* Command（抽象命令类）：抽象命令类一般是一个抽象类或接口，在其中声明了用于执行 请求的execute()等方法，通过这些方法可以调用请求接收者的相关操作
* ConcreteCommand（具体命令类）：具体命令类是抽象命令类的子类，实现了在抽象命令类 中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中。在实现execute()方 法时，将调用接收者对象的相关操作(Action)。
* Invoker（调用者）：调用者即请求发送者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以 将一个具体命令对象注入其中，再调用具体命令对象的execute()方法，从而实现间接调用请求 接收者的相关操作；
* Receiver（接收者）：接收者执行与请求相关的操作，它具体实现对请求的业务处理；

## 示例代码

### 标准方式

|  |
| --- |
| 抽象Command：  public interface *CalCommand* {   /\*\*  \* 计算操作  \*  \* @param value  \* @return  \*/  int cal(int *value*);   /\*\*  \* 撤销操作  \*  \* @return  \*/  int undo(); } |
| 具体命令：  public class *AddCommand* implements *CalCommand* {   private *Adder* adder = new Adder();   private int lastValue = 0;   @Override  public int cal(int *value*) {  lastValue = *value*;  return adder.add(*value*);  }   @Override  public int undo() {  return adder.add(-lastValue);  } } |
| 接受者：  public class *Adder* {  private int init = 0;    public int add(int *value*) {  init += *value*;  return init;  } } |
| 调用者：  public class *Calculator* {   /\*\*  \* Invoker中的Command，  \*/  private *CalCommand* command;   public void setCommand(*CalCommand command*) {  this.command = *command*;  }   public int execute(int *value*) {  return command.cal(*value*);  }   public int undo() {  return command.undo();  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *Calculator* calculator = new Calculator();  calculator.setCommand(new AddCommand());  *System*.***out***.println(calculator.execute(10));  *System*.***out***.println(calculator.undo());  *System*.***out***.println(calculator.execute(3));  *System*.***out***.println(calculator.execute(5));  *System*.***out***.println(calculator.undo());  } } |

### 多个命令的使用

|  |
| --- |
| 抽象命令：  public interface *CalCommand* {   int execute(int *firstValue*, int *secondValue*);   int undo(int *lastValue*, int *undoValue*); } |
| 具体命令：  public class *AddCommand* implements *CalCommand* {   private *Adder* adder = new Adder();   @Override  public int execute(int *firstValue*, int *secondValue*) {  return adder.add(*firstValue*, *secondValue*);  }   @Override  public int undo(int *lastValue*, int *undoValue*) {  return adder.undo(*lastValue*, *undoValue*);  } } |
| 调用者：  public class *Calculator* {   private *Map*<*String*, *CalCommand*> commands;   private final *String* ADD = "add";  private final *String* MULTI = "multi";  private int lastValue, undoValue; // 记录上次数据，用于撤销操作  private *CalCommand* lastCommand;   {  commands = new HashMap<>();  commands.put(ADD, new AddCommand());  commands.put(MULTI, new MultiCommand());  }   public int add(int *a*, int *b*) {  return cal(commands.get(ADD), *a*, *b*);  }   public int multi(int *a*, int *b*) {  return cal(commands.get(MULTI), *a*, *b*);  }   public int undo() {  return lastCommand.undo(lastValue, undoValue);  }    private int cal(*CalCommand command*, int *firstValue*, int *secondValue*) {  int result = *command*.execute(*firstValue*, *secondValue*);  lastValue = result;  undoValue = *secondValue*;  lastCommand = *command*;  return result;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Calculator* calculator = new Calculator();  int a = calculator.add(3, 4);  *System*.***out***.println(a);  int b = calculator.multi(a, 5);  *System*.***out***.println(b);  int undo = calculator.undo();  *System*.***out***.println(undo);  } } |

## 总结

### 优点

* 降低系统的耦合度。由于请求者与接收者之间不存在直接引用，因此请求者与接收者之间 实现完全解耦，相同的请求者可以对应不同的接收者，同样，相同的接收者也可以供不同的 请求者使用，两者之间具有良好的独立性。
* 新的命令可以很容易地加入到系统中。由于增加新的具体命令类不会影响到其他类，因此增加新的具体命令类很容易，无须修改原有系统源代码，甚至客户类代码，满足“开闭原则”的 要求。
* 可以比较容易地设计一个命令队列或宏命令（组合命令）。
* 为请求的撤销(Undo)和恢复(Redo)操作提供了一种设计和实现方案。

### 缺点

使用命令模式可能会导致某些系统有过多的具体命令类。

### 适用场景

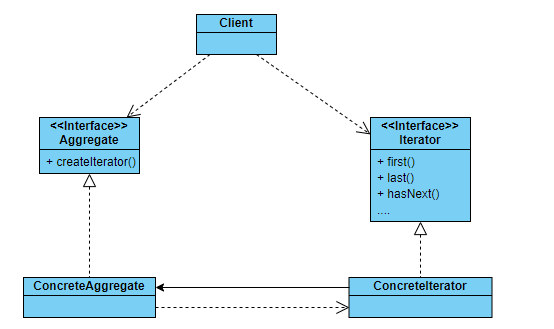
* 系统需要将请求调用者和请求接收者解耦， 使得调用者和接收者不直接交互
* 系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作
* 系统需要将一组操作组合在一起形成宏命令

# 迭代器模式

## 定义

迭代器模式(Iterator Pattern)： 提供一种方法来访问聚合对象， 而不用暴露这个对象的内部表示， 其别名为游标(Cursor)。 迭代器模式是一种对象行为型模式。

## 类图



## 角色

* Iterator（ 抽象迭代器） ： 它定义了访问和遍历元素的接口， 声明了用于遍历数据元素的方法， 例如： 用于获取第一个元素的first()方法， 用于访问下一个元素的next()方法；
* ConcreteIterator（ 具体迭代器） ： 它实现了抽象迭代器接口， 完成对聚合对象的遍历， 同时在具体迭代器中通过游标来记录在聚合对象中所处的当前位置， 在具体实现时， 游标通常是一个表示位置的非负整数；
* Aggregate（ 抽象聚合类） ： 它用于存储和管理元素对象， 声明一个createIterator()方法用于创建一个迭代器对象， 充当抽象迭代器工厂角色
* ConcreteAggregate（ 具体聚合类） ： 它实现了在抽象聚合类中声明的createIterator()方法， 该方法返回一个与该具体聚合类对应的具体迭代器ConcreteIterator实例。

## 核心代码

|  |
| --- |
| 抽象迭代器：  public interface *Iterator*<*T*> {   void add(*T t*);   *T* next();   boolean hasNext();   *T* first();   *T* last();   *String* disPlay(); } |
| 具体迭代器：  public class *ListIterator*<*E*> implements *Iterator*<*E*> {   private *ListAggregate*<*E*> listAggregate;  private *List*<*E*> list = null;   private int position = -1;   public ListIterator(*ListAggregate*<*E*> *listAggregate*) {  this.listAggregate = *listAggregate*;  this.list = *listAggregate*.getList();  }   @Override  public void add(*E e*) {  list.add(*e*);  }   @Override  public *E* next() {  return list.get(++position);  }   @Override  public boolean hasNext() {  return position < list.size()-1;  }   @Override  public *E* first() {  return list.get(0);  }   @Override  public *E* last() {  if (list.size() == 0) {  return null;  }  return list.get(list.size() - 1);  }   @Override  public *String* disPlay() {  return list.stream().map(*x* -> *x*.toString()).collect(*Collectors*.joining(",", "{", "}"));  } } |
| 抽象聚合类：  public interface *Aggregate*<*T*> {   *Iterator*<*T*> createIterator();  } |
| 具体聚合类：  public class *ListAggregate*<*T*> implements *Aggregate*<*T*> {   private *List*<*T*> list = new ArrayList<>();   @Override  public *Iterator*<*T*> createIterator() {  return new ListIterator(this);  }   public *List*<*T*> getList() {  return list;  } } |
| 客户端：  public class *Client* {   public static void main(*String*[] *args*) {  *Aggregate*<*String*> aggregate = new ListAggregate<>();  *Iterator*<*String*> iterator = aggregate.createIterator();  iterator.add("111");  iterator.add("222");  while (iterator.hasNext()) {  *System*.***out***.println(iterator.next());  }   *System*.***out***.println(iterator.disPlay());  } } |

## 总结

### 优点

* 它支持以不同的方式遍历一个聚合对象， 在同一个聚合对象上可以定义多种遍历方式。 在迭代器模式中只需要用一个不同的迭代器来替换原有迭代器即可改变遍历算法，我们也可以自己定义迭代器的子类以支持新的遍历方式；
* 迭代器简化了聚合类。 由于引入了迭代器， 在原有的聚合对象中不需要再自行提供数据遍历等方法， 这样可以简化聚合类的设计；
* 在迭代器模式中， 由于引入了抽象层， 增加新的聚合类和迭代器类都很方便， 无须修改原有代码， 满足“开闭原则”的要求

### 缺点

* 由于迭代器模式将存储数据和遍历数据的职责分离， 增加新的聚合类需要对应增加新的迭代器类， 类的个数成对增加， 这在一定程度上增加了系统的复杂性；
* 抽象迭代器的设计难度较大， 需要充分考虑到系统将来的扩展

### 适用场景

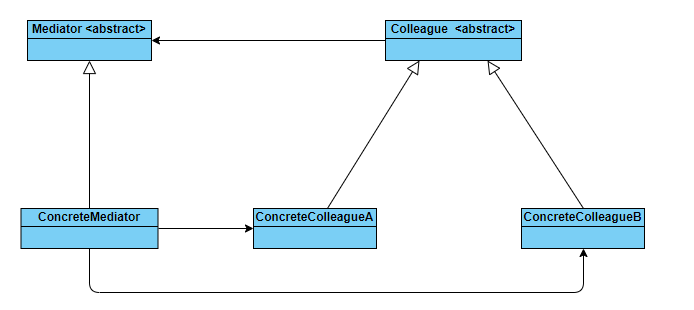
* 访问一个聚合对象的内容而无须暴露它的内部表示。 将聚合对象的访问与内部数据的存储分离， 使得访问聚合对象时无须了解其内部实现细节。
* 为遍历不同的聚合结构提供一个统一的接口， 在该接口的实现类中为不同的聚合结构提供不同的遍历方式， 而客户端可以一致性地操作该接口。

# 中介者模式

## 定义

中介者模式(Mediator Pattern)： 用一个中介对象（ 中介者） 来封装一系列的对象交互， 中介者使各对象不需要显式地相互引用， 从而使其耦合松散， 而且可以独立地改变它们之间的交互。 中介者模式又称为调停者模式， 它是一种对象行为型模式；

## 类图



## 角色

* Mediator（ 抽象中介者） ： 它定义一个接口， 该接口用于与各同事对象之间进行通信
* ConcreteMediator（ 具体中介者） ： 它是抽象中介者的子类， 通过协调各个同事对象来实现协作行为， 它维持了对各个同事对象的引用；
* Colleague（ 抽象同事类） ： 它定义各个同事类公有的方法， 并声明了一些抽象方法来供子类实现， 同时它维持了一个对抽象中介者类的引用， 其子类可以通过该引用来与中介者通信；
* ConcreteColleague（ 具体同事类） ： 它是抽象同事类的子类； 每一个同事对象在需要和其他同事对象通信时， 先与中介者通信， 通过中介者来间接完成与其他同事类的通信； 在具体同事类中实现了在抽象同事类中声明的抽象方法

## 核心代码

|  |
| --- |
| 抽象中介者：  public interface *EventBus* {  /\*\*  \* 发布事件  \* @param type 事件类型  \* @param message 消息对象  \*/  void post(*String type*, *Object message*); } |
| 具体中介者：  public class *DefaultEventBus* implements *EventBus* {  private static *DefaultEventBus* instance;  private static *Object* lockObj = new Object();  private *Map*<*String*, *List*<*EventHandler*>> eventHandlerMap;   private DefaultEventBus() {  eventHandlerMap = new HashMap<>();  }   public static *DefaultEventBus* getInstance() {  if (null == instance) {  synchronized (lockObj) {  if (null == instance) {  instance = new DefaultEventBus();  }  }  }  return instance;  }   @Override  public void post(*String type*, *Object message*) {  *List*<*EventHandler*> handlers = eventHandlerMap.get(*type*);  if (null != handlers) {  for (*EventHandler* handler : handlers) {  handler.handle(*message*);  }  }  }   /\*\*  \* 注册事件处理器  \*/  public void registerHandler(*String type*, *EventHandler eventHandler*) {  if (eventHandlerMap.containsKey(*type*)) {  *List* handlers = eventHandlerMap.get(*type*);  handlers.add(*eventHandler*);  } else {  *List* handlers = new ArrayList<>();  handlers.add(*eventHandler*);  eventHandlerMap.put(*type*, handlers);  }  } } |
| 抽象同事：  public abstract class *EventHandler* {   private *EventBus* eventBus;    public *EventBus* getEventBus() {  return eventBus;  }    public EventHandler(*EventBus eventBus*) {  this.eventBus = *eventBus*;  }   /\*\*  \* 处理事件  \* 消息对象  \*/  public abstract void handle(*Object message*);  } |
| 具体同事：  public class *KeyMapManager* extends *EventHandler* {  public KeyMapManager(*EventBus eventBus*) {  super(*eventBus*);  }   @Override  public void handle(*Object message*) {  *String* key = *message*.toString();  *System*.***out***.println("你按下了：" + key);  if ("alt+d".equals(key)) {  getEventBus().post("main\_form\_event", "hideWindow");  } else if ("alt+s".equals(key)) {  getEventBus().post("main\_form\_event", "showWindow");  }  } } |
| 客户端：  public class *App* {  public static void main(*String*[] *args*) {  *DefaultEventBus* eventBus = *DefaultEventBus*.getInstance(); // tips: type常量在程序中可以抽取到常量类中  eventBus.registerHandler("keyboard\_event", new KeyMapManager(eventBus));  eventBus.registerHandler("audio\_event", new AudioPlayer(eventBus));  eventBus.registerHandler("main\_form\_event", new MainForm(eventBus));  // 老板来了，赶紧隐藏窗口！！  eventBus.post("keyboard\_event", "alt+d");  // 老板走了，显示窗口继续玩。  eventBus.post("keyboard\_event", "alt+s");  } } |

## 总结

### 优点

* 中介者模式简化了对象之间的交互， 它用中介者和同事的一对多交互代替了原来同事之间多对多交互， 一对多关系更容易理解、 维护和扩展
* 中介者模式可将各同事对象解耦

### 缺点

* 在具体中介者类中包含了大量同事之间的交互细节， 可能会导致具体中介者类非常复杂， 使得系统难以维护；

### 适用场景

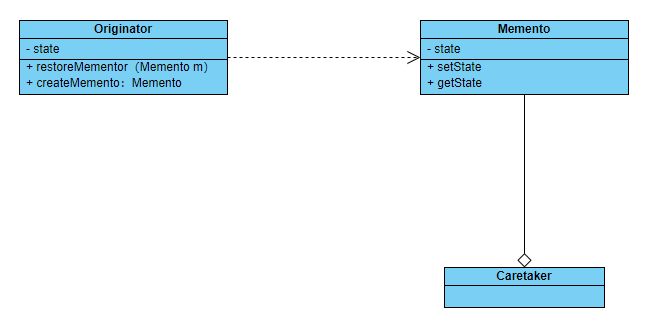
* 系统中对象之间存在复杂的引用关系， 系统结构混乱且难以理解。
* 一个对象由于引用了其他很多对象并且直接和这些对象通信， 导致难以复用该对象；
* 想通过一个中间类来封装多个类中的行为， 而又不想生成太多的子类；

# 备忘录模式

## 定义

备忘录模式(Memento Pattern)： 在不破坏封装的前提下， 捕获一个对象的内部状态， 并在该对象之外保存这个状态， 这样可以在以后将对象恢复到原先保存的状态。 它是一种对象行为型模式， 其别名为Token。

## 类图



## 角色

* Originator（ 原发器） ： 它是一个普通类， 可以创建一个备忘录， 并存储它的当前内部状态， 也可以使用备忘录来恢复其内部状态， 一般将需要保存内部状态的类设计为原发器；
* Memento（ 备忘录)： 存储原发器的内部状态， 根据原发器来决定保存哪些内部状态。 备忘录的设计一般可以参考原发器的设计， 根据实际需要确定备忘录类中的属性。 需要注意的是，除了原发器本身与负责人类之外， 备忘录对象不能直接供其他类使用。
* Caretaker（ 负责人） ： 负责人又称为管理者， 它负责保存备忘录， 但是不能对备忘录的内容进行操作或检查。 在负责人类中可以存储一个或多个备忘录对象， 它只负责存储对象， 而不能修改对象， 也无须知道对象的实现细节；

## 核心代码

这里实现较为负责的可以进行多次撤销的备忘录模式

|  |
| --- |
| 原发器：  @Data @AllArgsConstructor @ToString public class *Chessman* {   /\*\*  \* 棋子名  \*/  private *String* name;   /\*\*  \* 棋子的X坐标  \*/  private int x;  /\*\*  \* 棋子的Y坐标  \*/  private int y;    /\*\*  \* 保存状态  \*  \* @return  \*/  public *ChessmanMemento* save() {  return new ChessmanMemento(x, y);  }   public void restore(*ChessmanMemento memento*) {  this.x = *memento*.getX();  this.y = *memento*.getY();  } } |
| 备忘录：  @Data @AllArgsConstructor public class *ChessmanMemento* {   /\*\*  \* 棋子的X坐标  \*/  private int x;  /\*\*  \* 棋子的Y坐标  \*/  private int y;  }  负责人：  public class *ChessmanCaretaker* {   private *Map*<*String*, *List*<*ChessmanMemento*>> maps = new HashMap<>();   /\*\*  \* 根据棋子名和第几步来进行恢复  \*  \* @param name  \* @param i  \* @return  \*/  public *ChessmanMemento* getChessmanMemento(*String name*, int *i*) {  return maps.get(*name*).get(*i*);  }   public void setChessmanMemento(*String name*, *ChessmanMemento memento*) {  if (!maps.containsKey(*name*)) {  maps.put(*name*, new ArrayList<>());  }  maps.get(*name*).add(*memento*);  } } |
| 客户端：  public class *Client* {  // 当前是第几步  private static int index = -1;  private static *ChessmanCaretaker* caretaker = new ChessmanCaretaker();   public static void main(*String*[] *args*) {  *Chessman* chessman = new Chessman("车", 1, 1);  play(chessman);   chessman.setX(5);  play(chessman);   undo(chessman, --index);  play(chessman);    }   /\*\*  \* 下棋  \*  \* @param chessman  \*/  private static void play(*Chessman chessman*) {  caretaker.setChessmanMemento(*chessman*.getName(), *chessman*.save());  index++;  *System*.***out***.println(*chessman*);  }   /\*\*  \* 悔棋  \*  \* @param chessman  \* @param i说  \*/  private static void undo(*Chessman chessman*, int *i*) {  *chessman*.restore(caretaker.getChessmanMemento(*chessman*.getName(), *i*));  } } |

## 总结

### 优点

* 它提供了一种状态恢复的实现机制， 使得用户可以方便地回到一个特定的历史步骤， 当新的状态无效或者存在问题时， 可以使用暂时存储起来的备忘录将状态复原；
* 备忘录实现了对信息的封装， 一个备忘录对象是一种原发器对象状态的表示， 不会被其他代码所改动。 备忘录保存了原发器的状态， 采用列表、 堆栈等集合来存储备忘录对象可以实现多次撤销操作。

### 缺点

资源消耗过大， 如果需要保存的原发器类的成员变量太多， 就不可避免需要占用大量的存储空间， 每保存一次对象的状态都需要消耗一定的系统资源

### 适用场景

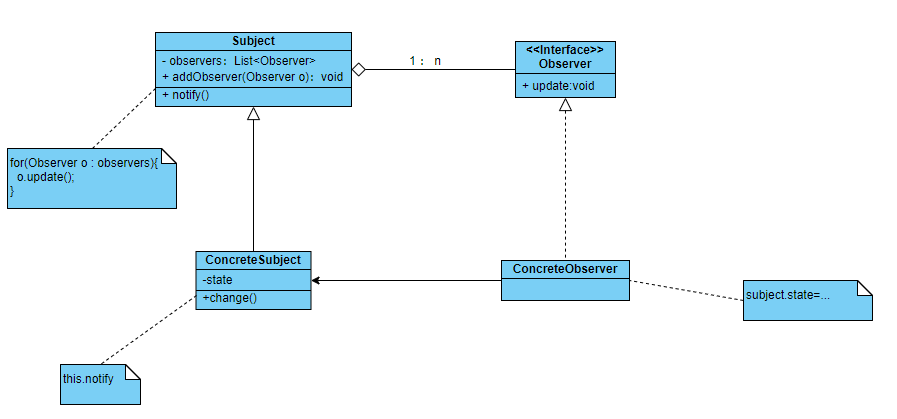
* 保存一个对象在某一个时刻的全部状态或部分状态， 这样以后需要时它能够恢复到先前的状态， 实现撤销操作；
* 防止外界对象破坏一个对象历史状态的封装性， 避免将对象历史状态的实现细节暴露给外界对象。

# 观察者模式

## 定义

观察者模式(Observer Pattern)： 定义对象之间的一种一对多依赖关系，使得每当一个对象状态发生改变时， 其相关依赖对象皆得到通知并被自动更新。 观察者模式的别名包括发布-订阅（ Publish/Subscribe） 模式、 模型-视图（ Model/View） 模式、 源-监听器（ Source/Listener） 模式或从属者（ Dependents） 模式。 观察者模式是一种对象行为型模式

## 类图



## 角色

* Subject（ 目标） ： 目标又称为主题， 它是指被观察的对象。 在目标中定义了一个观察者集合， 一个观察目标可以接受任意数量的观察者来观察， 它提供一系列方法来增加和删除观察者对象， 同时它定义了通知方法notify()。
* ConcreteSubject（ 具体目标） ： 具体目标是目标类的子类， 通常它包含有经常发生改变的数据， 当它的状态发生改变时， 向它的各个观察者发出通知； 同时它还实现了在目标类中定义的抽象业务逻辑方法（ 如果有的话） 。 如果无须扩展目标类， 则具体目标类可以省略；
* Observer（ 观察者） ： 观察者将对观察目标的改变做出反应， 观察者一般定义为接口， 该接口声明了更新数据的方法update()， 因此又称为抽象观察者；
* ConcreteObserver（ 具体观察者） ： 在具体观察者中维护一个指向具体目标对象的引用， 它存储具体观察者的有关状态， 这些状态需要和具体目标的状态保持一致； 它实现了在抽象观察者Observer中定义的update()方法。 通常在实现时， 可以调用具体目标类的attach()方法将自己添加到目标类的集合中或通过detach()方法将自己从目标类的集合中删除。

## 核心代码

|  |
| --- |
| 抽象目标：  public abstract class *Clickable* {   /\*\*  \* 观察者列表  \*/  *List*<*ClickListener*> listeners = new ArrayList<>();   /\*\*  \* 添加观察者  \*  \* @param listener  \*/  public void addListener(*ClickListener listener*) {  listeners.add(*listener*);  }   /\*\*  \* 删除观察者  \*  \* @param listener  \*/  public void removeListener(*ClickListener listener*) {  listeners.remove(*listener*);  }   /\*\*  \* 通知观察者  \*/  public void notifyListeners() {  listeners.forEach(*listener* -> *listener*.clicked(this));  }  } |
| 具体目标：  @Data @NoArgsConstructor @AllArgsConstructor @ToString public class *Button* extends *Clickable* {   /\*\*  \* 目标的一些状态信息  \*/  private *String* color;  private int size;    /\*\*  \* 业务方法  \*/  public void click() {  *System*.***out***.println("点击了按钮...");  this.notifyListeners();  } } |
| 抽象观察者：  public interface *ClickListener* {   void clicked(*Clickable clickable*); } |
| 具体观察者：  public class *ChangeColorListener* implements *ClickListener* {   private *List*<*String*> colors = *Arrays*.asList("红色", "黄色", "绿色", "紫色");   @Override  public void clicked(*Clickable clickable*) {  if (*clickable* instanceof *Button*) {  *Button* button = (*Button*) *clickable*;  *String* color = button.getColor();  int i = colors.indexOf(color);  button.setColor(colors.get((i + 1) % colors.size()));  }  } } |
| 客户端：  public class *Client* {  public static void main(*String*[] *args*) {  *Button* button = new Button("绿色",20);   // 添加监听器  button.addListener(new ChangeColorListener());  *ChangeSizeListener* sizeListener = new ChangeSizeListener();  button.addListener(sizeListener);   *System*.***out***.println(button);   button.click();  *System*.***out***.println(button);   // 移除大小改变监听器  button.removeListener(sizeListener);  button.click();  *System*.***out***.println(button);  } } |

## 总结

### 优点

* 观察者模式可以实现表示层和数据逻辑层的分离， 定义了稳定的消息更新传递机制， 并抽象了更新接口， 使得可以有各种各样不同的表示层充当具体观察者角色；
* 观察者模式在观察目标和观察者之间建立一个抽象的耦合。 观察目标只需要维持一个抽象观察者的集合， 无须了解其具体观察者。
* 观察者模式支持广播通信， 观察目标会向所有已注册的观察者对象发送通知， 简化了一对多系统设计的难度

### 缺点

* 如果一个观察目标对象有很多直接和间接观察者， 将所有的观察者都通知到会花费很多时间。
* 如果在观察者和观察目标之间存在循环依赖， 观察目标会触发它们之间进行循环调用， 可能导致系统崩溃。
* 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的， 而仅仅只是知道观察目标发生了变化。

### 适用场景

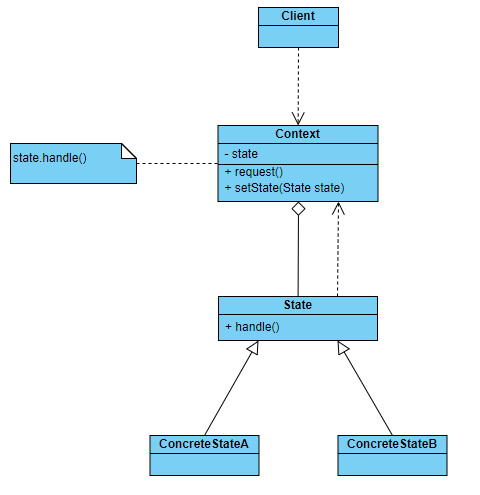
* 一个抽象模型有两个方面， 其中一个方面依赖于另一个方面， 将这两个方面封装在独立的对象中使它们可以各自独立地改变和复用；
* 一个对象的改变将导致一个或多个其他对象也发生改变， 而并不知道具体有多少对象将发生改变， 也不知道这些对象是谁。

# 状态模式

## 定义

状态模式(State Pattern)： 允许一个对象在其内部状态改变时改变它的行为， 对象看起来似乎修了它的类。 其别名为状态对象(Objects for States)， 状态模式是一种对象行为型模式

## 类图



## 角色

* Context（ 环境类） ： 环境类又称为上下文类， 它是拥有多种状态的对象。 由于环境类的状态存在多样性且在不同状态下对象的行为有所不同， 因此将状态独立出去形成单独的状态类。 在环境类中维护一个抽象状态类State的实例， 这个实例定义当前状态， 在具体实现时，它是一个State子类的对象。
* State（ 抽象状态类） ： 它用于定义一个接口以封装与环境类的一个特定状态相关的行为， 在抽象状态类中声明了各种不同状态对应的方法， 而在其子类中实现类这些方法， 由于不同状态下对象的行为可能不同， 因此在不同子类中方法的实现可能存在不同， 相同的方法可以写在抽象状态类中。
* ConcreteState（ 具体状态类） ： 它是抽象状态类的子类， 每一个子类实现一个与环境类的一个状态相关的行为， 每一个具体状态类对应环境的一个具体状态， 不同的具体状态类其行为有所不同。

## 状态的转换

状态会发生一些相应的转换，我们可以在具体的状态类中进行转换，也可以在Context中进行转换。

## 核心代码

## 总结

### 优点

* 封装了状态的转换规则， 在状态模式中可以将状态的转换代码封装在环境类或者具体状态类中， 可以对状态转换代码进行集中管理， 而不是分散在一个个业务方法中。
* 允许状态转换逻辑与状态对象合成一体， 而不是提供一个巨大的条件语句块， 状态模式可以让我们避免使用庞大的条件语句来将业务方法和状态转换代码交织在一起。

### 缺点

* 状态模式的结构与实现都较为复杂， 如果使用不当将导致程序结构和代码的混乱， 增加系统设计的难度。
* 状态模式对“开闭原则”的支持并不太好， 增加新的状态类需要修改那些负责状态转换的源代码， 否则无法转换到新增状态；

### 适用场景

* 对象的行为依赖于它的状态（ 如某些属性值） ， 状态的改变将导致行为的变化
* 在代码中包含大量与对象状态有关的条件语句， 这些条件语句的出现， 会导致代码的可维护性和灵活性变差， 不能方便地增加和删除状态， 并且导致客户类与类库之间的耦合增强，此时可以尝试使用状态模式。

# 策略模式

# 模板方法模式

# 访问者模式

# 解释器模式