## 一、设计模式：

创建型模式：单一实例

结构型模式：按某种布局组成更大结构

行为型模式：之间相互协作共同完成单一无法解决

|  |  |  |
| --- | --- | --- |
| **名 称** | **类模型** | **对象模型** |
| 创建型模式 | 工厂方法 | 单例、原型、抽象工厂、建造者 |
| 结构型模式 | 适配器 | 代理、适配器、桥接、装饰、外观、享元、组合 |
| 行为型模式 | 模板方法、解释器 | 策略、命令、责任链、状态、观察者、中介者、  迭代器、访问者、备忘录 |

## 二、23种设计模式

### 1、单例模式

原理：某个类只能生成一个实例，该类提供一个全局访问点供外部获取该实例

特点：

1、只有一个实例；

2、单例对象必须由单例自己创建；

3、单例必须对外提供一个访问该单例的全局访问点

|  |  |
| --- | --- |
|  |  |

**第一种懒汉模式：**

类加载时候并没有生成单例，只是第一次调用getInstance()方法时候才去创建这个单例

**第二种饿汉模式：**

类一旦加载就创建一个单例，保证在调用getInstance()方法之前单例已经存在

### 2、原型模式

原理：将一个对象作为原型，通过对其复制而克隆出多个和原型类似的新实例

特点：

1、抽象原型类：规定了具体原型对象必须实现的接口(Cloneable)

2、具体原型类：实现抽象原型类的clone()方法，他是可被复制的对象

3、访问类：使用具体原型类中的clone()方法来复制新的对象

|  |
| --- |
| //具体原型类  **class** Realizetype **implements** Cloneable{  Realizetype(){  System.***out***.println("具体原型创建成功！");  }  **public** Object clone() **throws** CloneNotSupportedException{  System.***out***.println("具体原型复制成功！");  **return** (Realizetype)**super**.clone();  }  }  //原型模式的测试类  **public** **class** PrototypeTest{  **public** **static** **void** main(String[] args)**throws** CloneNotSupportedException{  Realizetype obj1=**new** Realizetype();  Realizetype obj2=(Realizetype)obj1.clone();  System.***out***.println("obj1==obj2?"+(obj1==obj2));  }  } |

案例

|  |
| --- |
| **class** SunWukong **extends** JPanel **implements** Cloneable{  **private** **static** **final** **long** ***serialVersionUID*** = 5543049531872119328L;  **public** SunWukong(){ JLabel l1=**new** JLabel(**new** ImageIcon("src/Wukong.jpg")); **this**.add(l1);}  **public** Object clone(){  SunWukong w=**null**;  **try**{ w=(SunWukong)**super**.clone(); }**catch**(CloneNotSupportedException e){  System.***out***.println("拷贝悟空失败!");  }  **return** w;  }  }  **public** **class** ProtoTypeWukong{  **public** **static** **void** main(String[] args){  JFrame jf=**new** JFrame("原型模式测试");  jf.setLayout(**new** GridLayout(1,2));  Container contentPane=jf.getContentPane();  SunWukong obj1=**new** SunWukong();  contentPane.add(obj1);  SunWukong obj2=(SunWukong)obj1.clone();  contentPane.add(obj2);  jf.pack();  jf.setVisible(**true**);  jf.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);  }  }  输出结果 |

### 3、工厂方法模式

原理：定义一个用于创建产品的接口，由子类决定生产什么产品

主要优点：

1、用户只要知道具体工厂名称就可以得到所要产品，无需知道创建具体过程

2、新增产品只需要添加具体产品类和对应的具体工厂类，无须修改原工厂，满足开闭原则

结构:

1、抽象工厂：提供创建产品的接口，通过他访问具体工厂的工厂方法

2、具体工厂：实现抽象工厂中抽象方法，完成具体产品创建

3、抽象产品：定义产品规范，描述产品的主要特性和功能

4、具体产品：实现抽象产品所定义的接口，由具体工厂创建，他与具体工厂一一对应

备注：只考虑一类产品的生产

|  |
| --- |
| **public** **interface** AbstractFactory { AbstractShuiguo newShuiguo(); } //抽象工厂  **public** **class** ImplFactory **implements** AbstractFactory{ // 具体工厂  **private** AbstractShuiguo shuiguo;  **public** ImplFactory(AbstractShuiguo shuiguo) { **this**.shuiguo = shuiguo; }  **public** AbstractShuiguo newShuiguo() {System.***out***.println("具体工厂正在加工"); **return** shuiguo;}  }  **public** **interface** AbstractShuiguo { **void** show();} // 抽象产品  **public** **class** PingguoShuiguo **implements** AbstractShuiguo{ // 具体产品1  **public** **void** show() { System.***out***.println("苹果显示"); }  }  **public** **class** LiziShuiguo **implements** AbstractShuiguo{ // 具体产品2  **public** **void** show() { System.***out***.println("梨子显示"); }  }  **public** **class** Test { // 测试  **public** **static** **void** main(String[] args) {  AbstractFactory factory = **new** ImplFactory(**new** PingguoShuiguo());  factory.newShuiguo().show();  factory = **new** ImplFactory(**new** LiziShuiguo());  factory.newShuiguo().show();  }  } |

### 4、抽象工厂模式

原理：是一种为访问类提供一个创建一组相关或相互依赖对象的接口，访问类无需指定所要产品的具体类就能得到同组的不同等级的产品的模式结构

条件：

1、系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品

2、系统一次只可能消费其中某一族产品，即同族的产品一起使用

优点：

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、当增加一个新的产品族时不需要修改原代码，满足开闭原则。

缺点：

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改

### 5、建造者模式

原理：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建他们，最后构建成改复杂对象

优点：

1、各个具体的建造者相互独立，有利于系统的扩展

2、客户端不必知道产品内部组成的细节，便于控制细节风险

缺点:

1、产品的组成部分必须相同，这限制了其使用范围

2、如果产品的内部变化复杂，该模式会增加很多的建造者类

与工厂模式比较：

建造者注重零部件的组装过程。

工厂模式更注重零部件的创建过程

结构：

1、产品角色：包含多个组成部件的复杂对象，由具体建造者创建其各个零部件

2、抽象建造者：包含创建产品各个子部件的抽象方法的接口

3、具体建造者：完成复杂产品的各个部件的具体创建方法

4、指挥者：调用建造者对象中的部件构造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的信息

|  |
| --- |
| **public** **class** Product { // 产品角色  **private** String partA;  **private** String partB;  **private** String partC;  **public** String getPartA() { **return** partA;}  **public** **void** setPartA(String partA) { **this**.partA = partA; }  **public** String getPartB() { **return** partB; }  **public** **void** setPartB(String partB) { **this**.partB = partB; }  **public** String getPartC() { **return** partC; }  **public** **void** setPartC(String partC) { **this**.partC = partC; }  **public** **void** show() { System.***out***.println(partA + ":" + partB + ":" + partC); }  }  **public** **abstract** **class** Builder { // 抽象建造者  **public** Product product = **new** Product();  **public** **abstract** **void** buildPartA();  **public** **abstract** **void** buildPartB();  **public** **abstract** **void** buildPartC();  **public** Product resoult() { **return** product; }  }  **public** **class** ExBuilder **extends** Builder { //具体建造者  **public** **void** buildPartA() { product.setPartA("建造者A"); }  **public** **void** buildPartB() { product.setPartB("建造者B"); }  **public** **void** buildPartC() { product.setPartC("建造者C");}  }  **public** **class** Director { // 指挥者  **private** Builder builder;  **public** Director(Builder builder) { **this**.builder = builder; }  **public** Product construct(){  builder.buildPartA();  builder.buildPartB();  builder.buildPartC();  **return** builder.resoult();  }  }  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Director director = **new** Director(**new** ExBuilder());  Product product = director.construct();  product.show();  }  } |

### 6、代理模式

原理：为某对象提供一种代理以控制对该对象的访问

优点：

1、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、可以扩展目标对象的功能

3、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

缺点：

1、在客户端与目标对象之间增加一个代理对象，会造成请求处理速度变慢

2、增加了系统的复杂度

结构：

1、抽象主题：通过接口与抽象声明真实主题和代理对象实现的业务方法

2、真实主题：实现抽象主题的具体业务，是代理对象所代表真实对象，最终引用对象

3、代理类：提供与真实主题相同接口，内部含有对真实主题引用，可以访问、控制、扩展真实主题的功能

|  |
| --- |
| **public** **interface** Subject { **void** request(); } // 抽象主题  **public** **class** RealSubject **implements** Subject{ // 真实主题  **public** **void** request() { System.***out***.println("访问真实主题方法"); }  }  **public** **class** Proxy **implements** Subject{ //代理类  **private** RealSubject subject;  **public** **void** request() {  **if**(subject == **null**) {subject = **new** RealSubject();}  preRequest();  subject.request();  afterRequest();  }  **public** **void** preRequest() {System.***out***.println("访问真实主题之前");}  **public** **void** afterRequest() {System.***out***.println("访问真实主题之后");}  }  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Proxy proxy = **new** Proxy();  proxy.request();  }  } |

### 7、适配器模式

原理：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

优点：

1、客户端通过适配器可以透明地调用目标接口

2、复用了现存的类，程序员不要修改原有代码而重用现有的适配者类

3、将目标类和适配者类解耦，解决了目标类和适配者类接口不一致问题

缺点：

对类适配器来说，跟换适配器的实现过程比较复杂

结构：

1、目标接口：当前系统业务所期待的接口

2、适配者：他是被访问和适配的现存组件库中的组件接口

3、适配器：通过继承或引用适配者对象，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

|  |
| --- |
| 类适配器  **public** **interface** Target { **void** request(); } // 目标接口  **public** **class** Adapter { // 适配者接口  **void** speciRequest() { System.***out***.println("适配者业务代码被调用"); }  }  **public** **class** ClassAdapter **extends** Adapter **implements** Target{ // 类适配器类  **public** **void** request() { speciRequest(); }  }  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Target target = **new** ClassAdapter();  target.request();  }  }  对象适配器  **class** ObjectAdapter **implements** Target{ //对象适配器类  **private** Adapter adaptee;  **public** ObjectAdapter(Adapter adaptee){ **this**.adaptee=adaptee;}  **public** **void** request(){ adaptee.specificRequest(); }  }  **public** **class** ObjectAdapterTest{ //客户端代码  **public** **static** **void** main(String[] args){  System.***out***.println("对象适配器模式测试：");  Adapter adaptee = **new** Adapter ();  Target target = **new** ObjectAdapter(adaptee);  target.request();  }  } |

### 8、桥接模式

原理：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来实现，从而降低了抽象和实现这两个可变维度的耦合度

优点：

1、由于抽象和实现分离，所有扩展能力强

2、其实现细节对客户透明

缺点：

由于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，增加了系统的理解与设计难度

结构：

1、抽象化：定义抽象类，并包含一个对实现化对象的引用

2、扩展抽象化：是抽象化的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法

3、实现化：定义实现化角色的接口，供扩展抽象化调用

4、具体实现化：是实现化接口的具体实现

|  |
| --- |
| **public** **interface** Implmentor { // 实现化  **void** OperationImpl();  }  **public** **class** ImplmentorImpl **implements** Implmentor{ // 具体实现化  **public** **void** OperationImpl() { System.***out***.println("具体实现化调用"); }  }  **public** **abstract** **class** Abstractor { // 抽象化  **protected** Implmentor imple;  **public** Abstractor(Implmentor imple) { **this**.imple = imple; }  **public** **abstract** **void** Operation();  }  **public** **class** AbstractorEx **extends** Abstractor{ //扩展抽象化  **public** AbstractorEx(Implmentor imple) { **super**(imple); }  **public** **void** Operation() {  System.***out***.println("扩展抽象化被访问");  imple.OperationImpl();  }  }  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Implmentor tor = **new** ImplmentorImpl();  Abstractor ab = **new** AbstractorEx(tor);  ab.Operation();  }  } |

### 9、装饰模式

原理：动态的给对象增加一些职责，即增加其额外的功能

优点：

1、采用装饰模式扩展对象的功能比采用继承方式更加灵活。

2、可以设计出多个不同的具体装饰类，创造出多个不同行为的组合

缺点：

装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

结构：

1、抽象构件：定义一个抽象接口以规范准备接收附加责任的对象。

2、具体构件：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：继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。

4、具体装饰：实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

|  |
| --- |
| **interface** Component { //抽象构件角色  **public** **void** operation();  }  **class** ConcreteComponent **implements** Component { //具体构件角色  **public** ConcreteComponent() { System.***out***.println("创建具体构件角色"); }  **public** **void** operation() { System.***out***.println("调用具体构件角色的方法operation()"); }  }  **class** Decorator **implements** Component { //抽象装饰角色  **private** Component component;  **public** Decorator(Component component) { **this**.component = component; }  **public** **void** operation() { component.operation(); }  }  **class** ConcreteDecorator **extends** Decorator { //具体装饰角色  **public** ConcreteDecorator(Component component) { **super**(component); }  **public** **void** operation() { **super**.operation(); addedFunction(); }  **public** **void** addedFunction() { System.***out***.println("为具体构件角色增加额外的功能");}  } |

### 10、外观模式

原理：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

优点：

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

2、对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。

3、编译一个子系统不会影响其他的子系统、外观对象,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，

缺点：

不能很好地限制客户使用子系统类。

增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”

结构：

1、外观：为多个子系统对外提供一个共同的接口。

2、子系统：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：通过一个外观角色访问各个子系统的功能。

|  |
| --- |
| **public** **class** FacadePattern {  **public** **static** **void** main(String[] args) { Facade f = **new** Facade();f.method();}  }  **class** Facade { //外观角色  **private** SubSystem01 obj1 = **new** SubSystem01();  **private** SubSystem02 obj2 = **new** SubSystem02();  **private** SubSystem03 obj3 = **new** SubSystem03();  **public** **void** method() {obj1.method1();obj2.method2();obj3.method3();}  }  **class** SubSystem01 { //子系统角色  **public** **void** method1() {System.***out***.println("子系统01的method1()被调用！");}  }  **class** SubSystem02 { //子系统角色  **public** **void** method2() {System.***out***.println("子系统02的method2()被调用！");}  }  **class** SubSystem03 { //子系统角色  **public** **void** method3() {System.***out***.println("子系统03的method3()被调用！");}  } |

### 11、享元模式

原理：运用共享技术来有効地支持大量细粒度对象的复用。它通过共享已经存在的又橡来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率。

优点：

相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

缺点：

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

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: 是所有的具体享元类的基类，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：实现抽象享元角色中所规定的接口

3、非享元：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：负责创建和管理享元角色。当客户对象请求一个享元对象时，享元工厂检査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。

|  |
| --- |
| **public** **class** FlyweightPattern {  **public** **static** **void** main(String[] args) {  FlyweightFactory factory = **new** FlyweightFactory();  Flyweight f01 = factory.getFlyweight("a");  Flyweight f02 = factory.getFlyweight("a");  Flyweight f03 = factory.getFlyweight("a");  Flyweight f11 = factory.getFlyweight("b");  Flyweight f12 = factory.getFlyweight("b");  f01.operation(**new** UnsharedConcreteFlyweight("第1次调用a。"));  f02.operation(**new** UnsharedConcreteFlyweight("第2次调用a。"));  f03.operation(**new** UnsharedConcreteFlyweight("第3次调用a。"));  f11.operation(**new** UnsharedConcreteFlyweight("第1次调用b。"));  f12.operation(**new** UnsharedConcreteFlyweight("第2次调用b。"));  }  }  **class** UnsharedConcreteFlyweight {//非享元角色  **private** String info;  UnsharedConcreteFlyweight(String info) { **this**.info = info; }  **public** String getInfo() { **return** info; }  **public** **void** setInfo(String info) { **this**.info = info; }  }  **interface** Flyweight { //抽象享元角色  **public** **void** operation(UnsharedConcreteFlyweight state);  }  **class** ConcreteFlyweight **implements** Flyweight {//具体享元角色  **private** String key;  ConcreteFlyweight(String key) {  **this**.key = key;  System.***out***.println("具体享元" + key + "被创建！");  }  **public** **void** operation(UnsharedConcreteFlyweight outState) {  System.***out***.print("具体享元" + key + "被调用，");  System.***out***.println("非享元信息是:" + outState.getInfo());  }  }  **class** FlyweightFactory {//享元工厂角色  **private** HashMap<String, Flyweight> flyweights = **new** HashMap<String, Flyweight>();  **public** Flyweight getFlyweight(String key) {  Flyweight flyweight = (Flyweight) flyweights.get(key);  **if** (flyweight != **null**) {  System.***out***.println("具体享元" + key + "已经存在，被成功获取！");  } **else** {  flyweight = **new** ConcreteFlyweight(key);  flyweights.put(key, flyweight);  }  **return** flyweight;  }  } |

### 12、组合模式

原理：是一种将对象组合成树状的层次结构的模式，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的访问性。

优点：

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，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码

更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”

缺点：

设计较复杂，客户端需要花更多时间理清类之间的层次关系

不容易限制容器中的构件

不容易用继承的方法来增加构件的新功能

结构：

抽象构件：它的主要作用是为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

树叶构件：是组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中声明的公共接口

树枝构件：是组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明的接口，它的主要作用是存储和管理子部件，通常包含 Add()、Remove()、GetChild() 等方法。

|  |
| --- |
| **public** **class** CompositePattern {  **public** **static** **void** main(String[] args) {  Component c0 = **new** Composite();  Component c1 = **new** Composite();  Component leaf1 = **new** Leaf("1");  Component leaf2 = **new** Leaf("2");  Component leaf3 = **new** Leaf("3");  c0.add(leaf1);  c0.add(c1);  c1.add(leaf2);  c1.add(leaf3);  c0.operation();  }  }  **interface** Component {//抽象构件  **public** **void** add(Component c);  **public** **void** remove(Component c);  **public** Component getChild(**int** i);  **public** **void** operation();  }  **class** Leaf **implements** Component {//树叶构件  **private** String name;  **public** Leaf(String name) { **this**.name = name;}  **public** **void** add(Component c) {}  **public** **void** remove(Component c) {}  **public** Component getChild(**int** i) { **return** **null**; }  **public** **void** operation() { System.***out***.println("树叶" + name + "：被访问！"); }  }  **class** Composite **implements** Component { //树枝构件  **private** ArrayList<Component> children = **new** ArrayList<Component>();  **public** **void** add(Component c) { children.add(c); }  **public** **void** remove(Component c) { children.remove(c); }  **public** Component getChild(**int** i) { **return** children.get(i); }  **public** **void** operation() { **for** (Object obj : children) { ((Component) obj).operation(); } }  } |

### 13、模板方法模式

原理：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。

优点：

1、它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展

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的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则

缺点：

1、对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的结果，这导致一种反向的控制结构，它提高了代码阅读的难度

结构：

1、抽象类：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

2、具体子类：实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤

|  |
| --- |
| **public** **class** TemplateMethodPattern {  **public** **static** **void** main(String[] args) { AbstractClass tm = **new** ConcreteClass();tm.TemplateMethod();}  }  **abstract** **class** AbstractClass { // 抽象类  **public** **void** TemplateMethod() { SpecificMethod(); abstractMethod1(); abstractMethod2();}// 模板方法  **public** **void** SpecificMethod() {System.***out***.println("抽象类中的具体方法被调用..."); }// 具体方法  **public** **abstract** **void** abstractMethod1(); // 抽象方法1  **public** **abstract** **void** abstractMethod2(); // 抽象方法2  }  **class** ConcreteClass **extends** AbstractClass { // 具体子类  **public** **void** abstractMethod1() { System.***out***.println("抽象方法1的实现被调用..."); }  **public** **void** abstractMethod2() { System.***out***.println("抽象方法2的实现被调用..."); }  } |

### 14、策略模式

原理：该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，且算法的变化不会影响使用算法的客户。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

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条件语句

2、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，从而避免重复的代码

3、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、提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法

5、把算法的使用放到环境类中，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：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，一般使用接口或抽象类实现

2、具体策略：实现了抽象策略定义的接口，提供具体的算法实现

3、环境类：持有一个策略类的引用，最终给客户端调用

|  |
| --- |
| **public** **class** StrategyPattern {  **public** **static** **void** main(String[] args) {  Context c = **new** Context();  Strategy s = **new** ConcreteStrategyA();  c.setStrategy(s);  c.strategyMethod();  s = **new** ConcreteStrategyB();  c.setStrategy(s);  c.strategyMethod();  }  }  //抽象策略类  **interface** Strategy { **public** **void** strategyMethod();} // 策略方法  //具体策略类A  **class** ConcreteStrategyA **implements** Strategy {  **public** **void** strategyMethod() { System.***out***.println("具体策略A的策略方法被访问！"); }  }  //具体策略类B  **class** ConcreteStrategyB **implements** Strategy {  **public** **void** strategyMethod() { System.***out***.println("具体策略B的策略方法被访问！"); }  }  **class** Context { //环境类  **private** Strategy strategy;  **public** Strategy getStrategy() { **return** strategy; }  **public** **void** setStrategy(Strategy strategy) { **this**.strategy = strategy; }  **public** **void** strategyMethod() { strategy.strategyMethod(); }  } |

### 15、命令模式

原理：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。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储存、传递、调用、增加与管理

优点：

1、降低系统的耦合度。命令模式能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，它满足“开闭原则”，对扩展比较灵活

3、可以实现宏命令。命令模式可以与[组合模式](http://c.biancheng.net/view/1373.html" \t "http://c.biancheng.net/view/_blank)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](http://c.biancheng.net/view/1400.html" \t "http://c.biancheng.net/view/_blank)结合，实现命令的撤销与恢复

缺点：可能产生大量具体命令类。因为计对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：声明执行命令的接口，拥有执行命令的抽象方法 execute()

2、具体命令角色：是抽象命令类的具体实现类，它拥有接收者对象，并通过调用接收者的功能来完成命令要执行的操作

3、实现者/接收者：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：是请求的发送者，它通常拥有很多的命令对象，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

|  |
| --- |
| **public** **class** CommandPattern {  **public** **static** **void** main(String[] args) {  Command cmd = **new** ConcreteCommand();  Invoker ir = **new** Invoker(cmd);  System.***out***.println("客户访问调用者的call()方法...");  ir.call();  }  }  **class** Invoker { //调用者  **private** Command command;  **public** Invoker(Command command) { **this**.command = command; }  **public** **void** setCommand(Command command) { **this**.command = command; }  **public** **void** call() { System.***out***.println("调用者执行命令command..."); command.execute(); }  }  **interface** Command { **public** **abstract** **void** execute(); } //抽象命令  **class** ConcreteCommand **implements** Command { //具体命令  **private** Receiver receiver;  ConcreteCommand() { receiver = **new** Receiver(); }  **public** **void** execute() { receiver.action(); }  }  **class** Receiver { //接收者  **public** **void** action() { System.***out***.println("接收者的action()方法被调用"); }  } |

### 16、责任链模式

原理：为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止

优点：

1、降低了对象之间的耦合度。该模式使得一个对象无须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须拥有对方的明确信息

2、增强了系统的可扩展性。可以根据需要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，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任

4、责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 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，符合类的单一职责原则

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处理对象，系统性能将受到一定影响

3、职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，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、抽象处理者：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。

2、具体处理者：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者

3、客户类：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程

|  |
| --- |
| **public** **class** ChainOfResponsibilityPattern {  **public** **static** **void** main(String[] args) {  Handler handler1 = **new** ConcreteHandler1(); // 组装责任链  Handler handler2 = **new** ConcreteHandler2();  handler1.setNext(handler2);  handler1.handleRequest("two"); // 提交请求  }  }  **abstract** **class** Handler { //抽象处理者角色  **private** Handler next;  **public** **void** setNext(Handler next) { **this**.next = next; }  **public** Handler getNext() { **return** next; }  **public** **abstract** **void** handleRequest(String request); // 处理请求的方法  }  **class** ConcreteHandler1 **extends** Handler { //具体处理者角色1  **public** **void** handleRequest(String request) {  **if** (request.equals("one")) { System.***out***.println("具体处理者1负责处理该请求！"); }  **else** {  **if** (getNext() != **null**) { getNext().handleRequest(request); }  **else** { System.***out***.println("没有人处理该请求！"); }  }  }  }  **class** ConcreteHandler2 **extends** Handler { //具体处理者角色2  **public** **void** handleRequest(String request) {  **if** (request.equals("two")) { System.***out***.println("具体处理者2负责处理该请求！"); }  **else** {  **if** (getNext() != **null**) { getNext().handleRequest(request); }  **else** { System.***out***.println("没有人处理该请求！"); }  }  }  } |

### 16、状态模式

原理：对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为

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、有利于程序的扩展。通过定义新的子类很容易地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的混乱

结构：

1、环境：也称为上下文，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：定义一个接口，用以封装环境对象中的特定状态所对应的行为。

3、具体状态：实现抽象状态所对应的行为

|  |
| --- |
| **public** **class** StatePatternClient {  **public** **static** **void** main(String[] args) {  Context context = **new** Context(); // 创建环境  context.Handle(); // 处理请求  context.Handle();  context.Handle();  context.Handle();  }  }  **class** Context { //环境类  **private** State state;  **public** Context() { **this**.state = **new** ConcreteStateA(); } // 定义环境类的初始状态  **public** **void** setState(State state) { **this**.state = state; } // 设置新状态  **public** State getState() { **return** (state); } // 读取状态  **public** **void** Handle() { state.Handle(**this**); } // 对请求做处理  }  **abstract** **class** State { **public** **abstract** **void** Handle(Context context); } // 抽象状态类  **class** ConcreteStateA **extends** State { // 具体状态A类  **public** **void** Handle(Context context) {  System.***out***.println("当前状态是 A.");  context.setState(**new** ConcreteStateB());  }  }  **class** ConcreteStateB **extends** State { // 具体状态B类  **public** **void** Handle(Context context) {  System.***out***.println("当前状态是 B.");  context.setState(**new** ConcreteStateA());  }  } |

### 17、观察者模式

原理：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，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

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了一套触发机制

缺点：

1、目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用

2、当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率

结构：

1、抽象主题：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：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改变时，通知所有注册过的观察者对象

3、抽象观察者：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：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态

|  |
| --- |
| **public** **class** ObserverPattern {  **public** **static** **void** main(String[] args) {  Subject subject = **new** ConcreteSubject();  Observer obs1 = **new** ConcreteObserver1();  Observer obs2 = **new** ConcreteObserver2();  subject.add(obs1);  subject.add(obs2);  subject.notifyObserver();  }  }  **abstract** **class** Subject { // 抽象目标  **protected** List<Observer> observers = **new** ArrayList<Observer>();  **public** **void** add(Observer observer) { observers.add(observer); } // 增加观察者方法  **public** **void** remove(Observer observer) { observers.remove(observer); } // 删除观察者方法  **public** **abstract** **void** notifyObserver(); // 通知观察者方法  }  **class** ConcreteSubject **extends** Subject { //具体目标  **public** **void** notifyObserver() {  **for** (Object obs : observers) { ((Observer) obs).response(); }  }  }  **interface** Observer { **void** response(); } // 抽象观察者  **class** ConcreteObserver1 **implements** Observer { // 具体观察者1  **public** **void** response() { System.***out***.println("具体观察者1作出反应！"); }  }  **class** ConcreteObserver2 **implements** Observer { // 具体观察者1  **public** **void** response() { System.***out***.println("具体观察者2作出反应！"); }  } |

### 18、中介者模式

原理：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，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

优点：

1、降低了对象之间的耦合性，使得对象易于独立地被复用

2、将对象间的一对多关联转变为一对一的关联提高系统的灵活性，使得系统易于维护和扩展

缺点：

当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护

结构：

1、抽象中介者：它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。

2、具体中介者：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：定义同事类的接口，保存中介者对象，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：是抽象同事类的实现者，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

|  |
| --- |
| **public** **class** MediatorPattern {  **public** **static** **void** main(String[] args) {  Mediator md = **new** ConcreteMediator();  Colleague c1, c2;  c1 = **new** ConcreteColleague1();  c2 = **new** ConcreteColleague2();  md.register(c1);  md.register(c2);  c1.send();  c2.send();  }  }  **abstract** **class** Mediator {//抽象中介者  **public** **abstract** **void** register(Colleague colleague);  **public** **abstract** **void** relay(Colleague cl); // 转发  }  **class** ConcreteMediator **extends** Mediator { //具体中介者  **private** List<Colleague> colleagues = **new** ArrayList<Colleague>();  **public** **void** register(Colleague colleague) {  **if** (!colleagues.contains(colleague)) { colleagues.add(colleague); colleague.setMedium(**this**); }  }  **public** **void** relay(Colleague cl) {  **for** (Colleague ob : colleagues) { **if** (!ob.equals(cl)) { ((Colleague) ob).receive(); } }  }  }  **abstract** **class** Colleague { //抽象同事类  **protected** Mediator mediator;  **public** **void** setMedium(Mediator mediator) { **this**.mediator = mediator; }  **public** **abstract** **void** receive();  **public** **abstract** **void** send();  }  **class** ConcreteColleague1 **extends** Colleague { //具体同事类  **public** **void** receive() { System.***out***.println("具体同事类1收到请求。"); }  **public** **void** send() { System.***out***.println("具体同事类1发出请求。"); mediator.relay(**this**); }// 请中介者转发  }  **class** ConcreteColleague2 **extends** Colleague { //具体同事类  **public** **void** receive() { System.***out***.println("具体同事类2收到请求。"); }  **public** **void** send() { System.***out***.println("具体同事类2发出请求。"); mediator.relay(**this**); }// 请中介者转发  } |

### 19、迭代器模式

原理：提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

优点：

1、访问一个聚合对象的内容而无须暴露它的内部表示

2、遍历任务交由迭代器完成，这简化了聚合类

3、它支持以不同方式遍历一个聚合，甚至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类都很方便，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：

增加了类的个数，这在一定程度上增加了系统的复杂性

结构：

1、抽象聚合：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：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：定义访问和遍历聚合元素的接口

4、具体迭代器：实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历，记录遍历的当前位置

|  |
| --- |
| **public** **class** IteratorPattern {  **public** **static** **void** main(String[] args) {  Aggregate ag = **new** ConcreteAggregate();  ag.add("中山大学");  ag.add("华南理工");  ag.add("韶关学院");  System.***out***.print("聚合的内容有：");  Iterator it = ag.getIterator();  **while** (it.hasNext()) { Object ob = it.next(); System.***out***.print(ob.toString()); }  Object ob = it.first();  System.***out***.println("\nFirst：" + ob.toString());  }  }  **interface** Aggregate { // 抽象聚合  **public** **void** add(Object obj);  **public** **void** remove(Object obj);  **public** Iterator getIterator();  }  **class** ConcreteAggregate **implements** Aggregate { // 具体聚合  **private** List<Object> list = **new** ArrayList<Object>();  **public** **void** add(Object obj) { list.add(obj); }  **public** **void** remove(Object obj) { list.remove(obj); }  **public** Iterator getIterator() { **return** (**new** ConcreteIterator(list)); }  }  **interface** Iterator { Object first(); Object next(); **boolean** hasNext(); } // 抽象迭代器  **class** ConcreteIterator **implements** Iterator { // 具体迭代器  **private** List<Object> list = **null**;  **private** **int** index = -1;  **public** ConcreteIterator(List<Object> list) { **this**.list = list; }  **public** **boolean** hasNext() {  **if** (index < list.size() - 1) { **return** **true**; }  **else** { **return** **false**; }  }  **public** Object first() { index = 0; Object obj = list.get(index); **return** obj; }  **public** Object next() {  Object obj = **null**;  **if** (**this**.hasNext()) { obj = list.get(++index); }  **return** obj;  }  } |

### 20、访问者模式

原理：将作用于某种数据结构中的各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方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

优点：

1、扩展性好。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都比较单一

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类中增加相应的具体操作，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：定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 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：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：声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数

4、具体元素：实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this) ，另外具体元素中可能还包含本身业务逻辑的相关操作

5、对象结构：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的所有元素的方法，通常由 List、Set、Map 等聚合类实现

|  |
| --- |
| **public** **class** VisitorPattern {  **public** **static** **void** main(String[] args) {  ObjectStructure os = **new** ObjectStructure();  os.add(**new** ConcreteElementA());  os.add(**new** ConcreteElementB());  Visitor visitor = **new** ConcreteVisitorA();  os.accept(visitor);  visitor = **new** ConcreteVisitorB();  os.accept(visitor);  }  }  **interface** Visitor { //抽象访问者  **void** visit(ConcreteElementA element);  **void** visit(ConcreteElementB element);  }  **class** ConcreteVisitorA **implements** Visitor { // 具体访问者A类  **public** **void** visit(ConcreteElementA element) { System.***out***.println(element.operationA()); }  **public** **void** visit(ConcreteElementB element) { System.***out***.println(element.operationB()); }  }  **class** ConcreteVisitorB **implements** Visitor { // 具体访问者B类  **public** **void** visit(ConcreteElementA element) { System.***out***.println(element.operationA()); }  **public** **void** visit(ConcreteElementB element) { System.***out***.println(element.operationB()); }  }  **interface** Element { **void** accept(Visitor visitor); } // 抽象元素类  **class** ConcreteElementA **implements** Element { // 具体元素A类  **public** **void** accept(Visitor visitor) { visitor.visit(**this**); }  **public** String operationA() { **return** "具体元素A的操作。"; }  }  **class** ConcreteElementB **implements** Element { // 具体元素B类  **public** **void** accept(Visitor visitor) { visitor.visit(**this**); }  **public** String operationB() { **return** "具体元素B的操作。"; }  }  **class** ObjectStructure { // 对象结构角色  **private** List<Element> list = **new** ArrayList<Element>();  **public** **void** accept(Visitor visitor) {  Iterator<Element> i = list.iterator();  **while** (i.hasNext()) { ((Element) i.next()).accept(visitor); }  }  **public** **void** add(Element element) { list.add(element); }  **public** **void** remove(Element element) { list.remove(element); }  } |

### 21、备忘录模式

原理：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。

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的状态

2、实现了内部状态的封装。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、简化了发起人类。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：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息

2、备忘录：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：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

|  |
| --- |
| **public** **class** MementoPattern {  **public** **static** **void** main(String[] args) {  Originator or = **new** Originator();  Caretaker cr = **new** Caretaker();  or.setState("S0");  System.***out***.println("初始状态:" + or.getState());  cr.setMemento(or.createMemento()); // 保存状态  or.setState("S1");  System.***out***.println("新的状态:" + or.getState());  or.restoreMemento(cr.getMemento()); // 恢复状态  System.***out***.println("恢复状态:" + or.getState());  }  }  //备忘录  **class** Memento {  **private** String state;  **public** Memento(String state) { **this**.state = state; }  **public** **void** setState(String state) { **this**.state = state; }  **public** String getState() { **return** state; }  }  //发起人  **class** Originator {  **private** String state;  **public** **void** setState(String state) { **this**.state = state; }  **public** String getState() { **return** state; }  **public** Memento createMemento() { **return** **new** Memento(state); }  **public** **void** restoreMemento(Memento m) { **this**.setState(m.getState()); }  }  //管理者  **class** Caretaker {  **private** Memento memento;  **public** **void** setMemento(Memento m) { memento = m; }  **public** Memento getMemento() { **return** memento; }  } |

### 22、解释器模式

原理：给分析对象定义一个语言，并定义该语言的文法表示，再设计一个解析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。

优点：

1、扩展性好。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机制来改变或扩展文法

2、容易实现。在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、执行效率较低。解释器模式中通常使用大量的循环和递归调用，当要解释的句子较复杂时，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，类的个数将急剧增加，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：

1、抽象表达式：定义解释器的接口，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：是抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应

3、非终结符表达式：也是抽象表达式的子类，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：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：主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树，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

|  |
| --- |
| //抽象表达式类  **interface** AbstractExpression {  **public** Object interpret(String info); // 解释方法  }  // 终结符表达式类  **class** TerminalExpression **implements** AbstractExpression {  **public** Object interpret(String info) { } // 对终结符表达式的处理  }  // 非终结符表达式类  **class** NonterminalExpression **implements** AbstractExpression {  **private** AbstractExpression exp1;  **private** AbstractExpression exp2;  **public** Object interpret(String info) { }// 非对终结符表达式的处理  }  // 环境类  **class** Context {  **private** AbstractExpression exp;  **public** Context() { }// 数据初始化  **public** **void** operation(String info) { }// 调用相关表达式类的解释方法  } |