# DesignPattern技术栈

《武林外传》中有这么一段对话：

邢捕头：手中无刀，心中有刀；

老白：错了，最高境界是手中无刀，心中也无刀。

设计模式四巨头The Gang of Four: Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides。

## 六大设计原则

### 单一职责原则

定义：

SRP，Single Responsible Principle，定义是：There should never be more than one reason for a class to change。

细节：

1. RBAC模型 Role-Based Access Control，基于角色的访问控制，通过分配和取消角色来完成用户权限的授予与取消。
2. 用户的信息需要抽取成一个BO，Business Object，行为抽取成Biz，Business Logic。

影响：

There is sometimes hard to see，生搬硬套该原则会容易使得类的剧增，给维护带来很大的麻烦，而且过细的划分使得系统变得复杂。

### 里氏替换原则

定义：

1. LSP，Liskov Substitution Principle
2. 定义一：If for each object o1 of type S there is an object o2 of type T, the behavior of P is unchanged when o1 is substituted for o2 then S is subtype of T.
3. 定义二：Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

四层含义：

1. 子类必须完全实现父类的方法。在类中调用其他类的时候必须使用父类或者接口，这样做前一个类就不必知道调用的类具体的子类是什么，如果不能使用父类或者接口，说明类的设计违反了LSP原则。
2. 子类可以有自己的个性；
3. 覆盖或者实现父类的方法时输入参数可以被放大（重载），若被缩小了，则在某些场合下调用子类时无法实现父类的方法。子类无法完全实现父类的方法则违反第一层含义，即违背LSP原则；
4. 覆盖或者实现父类的方法时输出结果可以被缩小（这是覆写的要求）

最佳实践：

在实际开发过程中，应尽量避免子类的“个性”。

### 依赖倒置原则

定义：

1. DIP，Dependence Inversion Principle，定义：High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details. Detail should depend upon abstraction.
2. 什么叫“倒置”，为什么不叫“正置”？原因是依赖正置指的是实实在在的实现类间的依赖，符合正常人的思维方式

细节：

1. 依赖关系是通过实现接口或者抽象类，或者传入参数的方式产生的，更加精简的定义是“面向接口编程”——ODD（Object-oriented Design）；
2. 在业务场景中，“抽象不应该依赖细节”，即抽象类不应该依赖于实现类；
3. 在java中，只要定义变量就会有两种类型——表面类型和实际类型，前者是在定义的时候赋予的类型，后者是对象的类型。
4. TDD Test-Driven Development，测试驱动开发就是依赖倒置最高级的应用，只要制定好接口或者抽象类就可以进行独立开发，单元测试也可以独立运行。

依赖的三种写法

1. 构造函数传递依赖对象，在构造函数中声明依赖对象，这种方式叫做构造函数注入

public class Driver(){

private ClassA classA;

public void Driver(Class classA){

this.classA=classA

}

}

1. Setter方法传递依赖对象
2. 接口声明依赖对象

最佳实践：

1. 每个类尽量都有接口和抽象类；
2. 变量的表面类型尽量是接口或者是抽象类；

但是这个也不是绝对的，比如一个工具类XXXutils一般不需接口和抽象类的，还有，如果你使用的是类的clone方法，就必须要用实现类，这是JDK提供的一个规范；

1. 任何类都不应该从具体类中派生；
2. 尽量不要覆写基类的方法；
3. 结合里氏替换原则使用；

### 接口隔离原则

定义：

1. Clients should not be forced to depend upon interface that they don’t use.
2. The dependency of one class to another one should depend on the smallest possible interface.

### 迪米特法则

定义：

1. LoD，Law of Demeter 也称最少知识原则，only talk to your immediate friends.
2. 而朋友类的定义：出现在成员变量和成员方法中的输入输出参数的类，一个类最好只与它的朋友类进行通讯，否则，会破坏类的健壮性。方法是类的一个行为，类如果不知道与其他类产生依赖关系，这是不允许的

细节：

1. 在java以及其他的面向对象设计模式中，类与类之间主要有6种关系，他们分别是：依赖、关联、聚合、组合、继承、实现。他们的耦合度依次增强。关联表现为类A为类B的成员变量，聚合关系中作为成员变量的类一般使用set方法赋值，组合关系中的成员变量一般会在构造方法中赋值；
2. 朋友间也不要太过亲密，即尽量不要对外公布太多的public方法和非静态的public方法，尽量多使用private，package-private和protected等访问权限。

最佳实践：

1. 在实际应用中，如果一个类需要跳转两次才能访问到另一个类，就需要想办法重构了。为什么是两次呢？因为跳转的次数越多，系统越复杂，维护越困难，只要跳转的次数小于两次的还可以接受，当然了具体的问题需要具体分析。

### 开闭原则

定义：

1. Software entities like classes, modules and functions should be open for extension but closed for modification.
2. 开闭原则告诉我们应尽量通过拓展软件实体的行为来实现变化，而不是通过修改已有的代码来完成变化，它是为软件实体的未来事件而制定的对先行开发进行约束的一个原则。

细节：

1. 运维尽量减少对原有代码的修改，保持历史代码的纯洁性，提高系统的稳定性；
2. 开闭原则是最基础的一个原则，前面所讲的五个原则都是开闭原则的具体形态。

## 创建类模式

### 单例模式

定义：

1. Single pattern，Ensure a class has only one instance, and provide a global point of access to it；

细节代码：皇帝类

1. 通过定义一个私有访问权限的构造函数，避免被其他类new出来一个对象，而Emperor他自己可以new一个静态对象出来，确保其他类对该类的访问都可以通过getInstance获得同一个对象。本实例又称为饿汉式单例。

public class Emperor{

private static final Emperor emperor = new Emperor();

private Emperor(){

}

public Emperor getInstance(){

return emperor;

}

}

优势：

1. 减少内存的开销，特别是对于一个对象需要频繁的创建和销毁；
2. 减少系统的性能开销；
3. 避免对资源的多重占用，比如一个写文件的动作，由于只有一个实例存在的时候，避免对同一个资源文件的同时写的操作；
4. 相当于设置全局变量；

劣势：

单例模式没有接口，很难拓展，若要拓展，除了修改代码没有别的办法，这就违反了开闭原则；

单例模式对测试是不利的，在并行开发环境中，如果单例模式没有完成，是不能进行测试的，没有接口也不能使用mock的方式虚拟一个对象；

注意事项;

1. 线程不安全的实例，在高并发的情况下，emperor new的过程需要时间，当emperor还没有来得及new出来，另外一个线程进行了emperor == null的判断，所以有可能会出现两个emperor的出现。为了避免这种情况的出现，可以在getInstance()方法前添加synchronized关键字，也可以在方法内添加synchronized关键字，这种实例又称为懒汉式单例。

public class Emperor{

private static final Emperor emperor = null;

private Emperor(){

}

public static Emperor getInstance(){

if(emperor == null){

emperor=new Emperor();

}

return emperor;

}

}

### 工厂方法模式

定义：

1. Define an interface for creating an object, but let subclasses decide which class to instance. Factory Method lets a class defer instantiation to subclasses.
2. 抽象产品类product负责定义产品的共性，实现对产品最抽象的定义，Creator为抽象创建类，也就是抽象工厂类，具体如何创建产品是由具体的实现工厂ConcreteCenter完成

细节：

1. Class对象的生成方式如下，在生成Class对象的时候，首先判断内存中是否已经加载，已加载的不再加载：
2. 类名.class 说明： JVM将使用类装载器, 将类装入内存(前提是:类还没有装入内存),不做类的初始化工作.返回Class的对象；
3. Class.forName("类名字符串") （注：类名字符串是包名+类名） 说明：装入类,并做类的静态初始化，返回Class的对象；
4. 实例对象.getClass() 说明：对类进行静态初始化、非静态初始化；返回引用运行时真正所指的对象(因为:子对象的引用可能会赋给父对象的引用变量中)所属的类的Class的对象；
5. getName()，以String形式返回次Class对象所表示的实体名称。
6. 具体的代码示例：

//抽象产品类

public abstract class Product{

//产品类的公共方法

public void method1(){

}

//抽象方法

public abstract void method2();

}

//具体产品类

public class ConcreteProduct1 extends Product {

public void method1(){

//具体的业务处理

}

@Override

public void method2() {

// TODO Auto-generated method stub

}

}

public class ConcreteProduct2 extends Product {

public void method1(){

//具体的业务处理

}

@Override

public void method2() {

// TODO Auto-generated method stub

}

}

//抽象工厂：

public abstract class Centor{

public abstract <T extends Product> T createProduct(Class<T> c);

}

//具体工厂：

public class ConcreteCentor extends Centor{

@SuppressWarnings("unchecked")

public <T extends Product> T createProduct(Class<T> c) {

Product product=null;

try {

product=(Product)Class.forName(c.getName()).newInstance();

} catch (Exception e) {

// TODO: handle exception

}

return (T)product;

}

}

//场景类

public class Client{

public static void main(String[] arg) {

Centor center=new ConcreteCentor();

Product product=center.createProduct(ConcreteProduct1.class);

product.method1();

}

}

1. 优点：

有良好的封装性，只要知道类名即可创建，无需了解具体的创建过程，降低模块间的耦合；

优秀的拓展性，在增加产品类的情况下，只要适当地修改具体工厂类或者拓展一个工厂类，就可以完成“拥抱变化”；

屏蔽产品类，产品类如何变化，使用者都不需要关心，只需要关心产品接口即可，只要接口不变，系统的上层模块就不用变化，因为产品的实例化工作是由工厂完成的。

使用场景：

首先在使用new一个对象的场合都可以使用，区别在于new适用于具体类的创建和实现，而工厂方法适用于接口和抽象类的创建，是面向接口编程的首选。但是需要谨慎考虑是否增加一个工厂类进行管理，增加代码的复杂度。

其次，当需要灵活和可拓展框架的时候也可以考虑。比如设计一个连接邮件服务器的模块的时候，定义一个接口如IConnectMail，有三种网络协议可供选择，如POP3，IMAP和HTTP，那么我们可以将此三种协议的实现看成是具体产品类。假如以后需要增加一个webService接口时，我们增加一个产品类即可。

工厂模式的拓展：

简单工厂类 只有一个工厂类，此时可以把抽象工厂去掉，然后把具体工厂的成员方法变为静态即可。又叫静态工厂模式，缺点是方便拓展，不符合开闭原则。

升级为多个工厂类，为每个产品添加一个具体工厂类；

替代单例模式 不知道是啥；

延迟初始化 不知道是啥；

### 抽象工厂模式

定义

1. Abstract Factory Pattern，定义：Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
2. 抽象工厂模式是工厂方法的升级版，相比工厂方法，在抽象工厂类的上层的抽象类编程一个接口，抽象类返回对象实例，但每次只返回一个。而接口每次则可以返回不同对象的实例。就好比女娲造人，工厂方法是指女娲造不同肤色的人。后来女娲发现原来人还有性别之分，不同肤色的人还需要分男女，此时需要添加一个造不同肤色人的接口，然后添加两个用于实现该接口的造男人和女人的具体工厂类。
3. 抽象工厂相当于生产多种产品族的方法，每个产品族内的产品相似，则可以把相似的产品抽取出来作为抽象工厂，然后具体工厂类负责生实现不同的产品族的生产。

优点：

1. 封装性 每个产品的实现类不是高层模块要关心的，他关心的是接口，是抽象，是工厂类；
2. 产品族内的约束为非公开状态，该约束对于调用工厂类的高层模块来说是透明的，具体产品类的约束是在工厂内实现的。

缺点：

1. 最大的缺点在于产品族内产品的拓展是十分困难的，不仅要改抽象工厂类，还要改每个具体工厂类，不符合开闭原则。而且我们一直在强调抽象和接口是一个契约，改变契约，所有跟契约有关的代码都需要修改——这种代码又称为“有毒代码”。

使用场景

1. 一般用于具体相同约束的产品类。如不同操作系统上的图片编辑和文本编辑器。

### 模板方法模式

定义

1. Template Method Pattern Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.
2. 将子类完全相同的方法放在抽象类中

细节

1. 为了避免被恶意的操作，一般模板方法前面都加上final关键字，不允许被覆写，代码示例：

//模板方法

public abstract class AbstractClass{

//基本方法

protected abstract void doSomething();

protected abstract void doAnything();

//模板方法

public void templateMethod() {

this.doSomething();

this.doAnything();

}

}

模板方法中的基本方法尽量使用protecte类型，符合迪特米原则。

优点：

1. 提取公共代码，便于维护；
2. 行为由父类控制，子类实现；

缺点

1. 思维方式不符合抽象类定义一般方法，实现类定义具体方法的习惯。

使用场景

1. 多个子类有公共的方法；
2. 重要的复杂的算法，可以把核心算法设计为模板方法，周边的细节功能则由各个子类实现；

拓展

钩子方法（Hook Method） 增加判断的方法用于调整模板方法执行的顺序

### 建造者模式

定义：

1. Builder Pattern 也叫生成器模式 定义：Separate the construction of a complex object from its representation so that the same construction process can create different representation.

细节：

1. 类图的使用。
2. 加号（+）表示public；减号（-）表示private；井号(#)表示protected；省略这些修饰符表示具有package（包）级别的可见性。
3. 如果属性或方法具有下划线，则说明它是静态的。
4. 描述信息使用 << 开头，使用 >> 结尾。
5. 一般化关系：表示为类与类之间的继承关系，接口与接口之间的继承，类对接口的实现关系。用一个空心箭头+实线，箭头指向父类。或空心箭头+虚线，如果父类是接口。
6. 关联关系：类与类之间的联接，它使一个类知道另一个类的属性和方法。用实线+箭头， 箭头指向被使用的类。
7. 聚合关系：空心菱形+实线+箭头，箭头指向部分。
8. 合成关系：实心菱形+实线+箭头，
9. 依赖关系：虚线+箭头 箭头指向被依赖的一方，也就是指向局部变量。
10. ArrayList和HashMap如果定义为成员变量，则在成员方法中调用该成员变量前需要作初始化操作，即调用clear()方法，防止数据混淆；
11. 还有this关键字的使用，如果加不加该关键字对结果都没有什么影响，但是不加的话可能会会清晰，这时候最好还是添加；
12. 一共有四个角色 Product产品类，Builder抽象建造者，Concretebuilder具体建造者和Director导演类，建造者其实就是控制产品的组成方式，相比于模板方法，增加了多个的模板。

优点：

1. 封装性 客户端不需要知道产品内部的组成细节；
2. 建造者独立，容易拓展；
3. 便于控制细节风险；

使用场景：

1. 相同的方法，不同的执行顺序，会产生不同的效果；
2. 一个对象中不同零部件的自由组合，会产生不同的效果；

### 原型模式

定义

Prototype Pattern，原型模式，定义：Specify the objects to create using a prototypical instance, and create new objects by copying this prototype.

细节：

1. 实现Cloneable接口，并覆写Object类中的clone方法

public class PrototypeClass implements Cloneable{

@Override

public PrototypeClass clone() {

PrototypeClass prototypeClass=null;

try {

prototypeClass=(PrototypeClass)super.clone();

} catch (Exception e) {

// TODO: handle exception

}

return prototypeClass;

}

}

使用的时候直接用

PrototypeClass prototypeClass=new PrototypeClass();

PrototypeClass prototypeClassClone= prototypeClass.clone();

1. 作用相当于new；
2. 浅拷贝与深拷贝；

浅拷贝指的是对象使用clone()方法时，拷贝的只是对象本身，对于对象内的数组和引用的对象等都不拷贝，若要进行深拷贝，可以加几行代码：

try {

prototypeClass=(PrototypeClass)super.clone();

this.arrayList= this.arrayList.clone();

} catch (Exception e) {

// TODO: handle exception

}

1. clone和final关键字是矛盾的，两者只能存其一；

优点：

1. 性能优良

原型模式的拷贝是基于内存中的二进制流的拷贝，性能要比直接new一个对象好很多，特别是对于需要循环大量创建同一对象的情况；

1. 逃避构造函数的约束

这是他的优点，同时又是他的缺点，因为clone()方法不会执行构造函数；

## 结构类模式

### 适配器模式

定义：

1. Adapter Pattern，又叫变压器模式或者包装模式，定义：Convert the interface of a class into another interface clients expert. Adapter lets classers work together that couldn’t otherwise because of incompatible interfaces.
2. 三个角色：
3. Target目标角色，也即期望接口；
4. Adaptee源角色，需要被转换的类或者对象；
5. Adapter适配器角色。

细节：

1. 代码实现：

//目标角色

public interface Target{

public void Request();

}

//目标角色的实现类

public class ConcreteTarget implements Target{

public void Request() {

System.out.println("If you need help, Please call me.");

};

}

//源角色

public class Adaptee{

public void doSomething() {

System.out.println("I'm kind of buzy,leave me alone,ple.");

}

}

//适配器角色

public class Adapter extends Adaptee implements Target{

@Override

public void Request() {

super.doSomething();

}

}

//场景类/高层类

public class Client{

public static void main(string[] args) {

//原有的业务逻辑

Target target1=new ConcreteTarget();

Target1.Request();

//现在增加适配器角色后

Target target2=new ConcreteTarget();

Target2.Request();

1. 项目必须满足依赖倒置原则和里氏替换原则，否则使用时需要很大的改造：
2. 关联关系指的是通过构造函数传递参数；
3. 当需要适配的Adaptee源角色有多个时，由于java不支持多继承功能，此时可将多继承的关系变为关联的关系，通过构造函数传递Adaptee源角色。

优点：

1. 增加了类的透明性。我们访问Target目标角色，但是具体的实现都委托了给源角色，而这些对高层此模块是透明的，也不用关心；
2. 提高了类的复用度
3. 灵活性非常好

使用场景：

当你有动机想要修改一个已经投产的接口时，比如说新建一个类，但是这个类又不符合当前的接口时

### 桥梁模式

定义：

1. Bridge Pattern，定义：Decouple an abstraction from its implementation so that the two can vary independently.
2. 四个角色
3. Abstraction抽象化角色
4. Implementor实现化角色
5. RefinedAbstraction修正抽象化角色
6. ConcreteImplementor具体实现化角色

其实就是在引用的时候用使用接口引用，然后在高层模块调用的时候才传递具体的实现类，非常简单。

### 组合模式

定义：

1. Composite Pattern，定义：Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual object and compositions of objects uniformly.
2. 三个角色：
3. Component抽象构件角色，定义组合对象共有的方法和属性，是一个接口；
4. Leaf叶子构件，其再下没有其他分支，也就是遍历的最小单位；
5. Composite树枝构件

细节：

1. 关键在于树枝和树叶共同的接口和实现共同接口的分开的接口，以及场景类的递归操作，实现代码如下

//抽象构建角色

public interface Component{

public void doSomething();

}

//抽象树叶角色

public interface class AbstractLeaf implements Component{

public void doSomething();

}

//抽象树枝角色

public interface AbstractComposite implements Component{

public void add(Component component);

public void remove(Component component);

public ArrayList<Component> getChildren();

public void doSomething() {

};

}

//树叶构建

public class Leaf implements AbstractLeaf{

@Override

public void doSomething() {

// TODO Auto-generated method stub

}

}

//树枝构建

public class Composite implements AbstractComposite{

ArrayList<Component> arrayList=new ArrayList<Component>();

@Override

public void add(Component component) {

this.arrayList.add(component);

}

@Override

public void remove(Component component) {

this.arrayList.remove(component);

}

@Override

public ArrayList<Component> getChildren() {

return this.arrayList;

}

@Override

public void doSomething() {

// TODO Auto-generated method stub

}

}

//场景类

public class Client{

public static void main(String[] args) {

Composite root=new Composite();

root.doSomething();

Composite branch=new Composite();

branch.doSomething();

Leaf leaf1=new Leaf();

Leaf leaf2=new Leaf();

root.add(branch);

branch.add(leaf1);

branch.add(leaf2);

}

public static void display(Composite root) {

root.doSomething();

for(Component c:root.getChildren) {

if(c instanceof Leaf) {

c.doSomething();

}else {

c.doSomething();

display((Composite)c);

}

}

}

}

优点：

1. 高层模块调用简单，其不必挂心自己调用的是树枝还是树叶；
2. 节点增加自由，符合开闭原则；

缺点：

1. 违反了依赖倒置的原则

由于最初接口中只有doSomething()的方法，而子类中的方法不仅有doSomething()方法，所以不能对象的表面类不能直接使用最初的接口，必须使用能容纳所有方法的最近的接口或者时父类，按照以上原则，就违反了依赖倒置的原则；

拓展：

1. 安全的组合模式

以上讲的就是安全的组合模式；

1. 透明的组合模式

其实就是把add()，remove()和getChildren()的方法搬到抽象类Component中去，后面不管是树叶还是树枝都拥有以上方法，需要判断getChildren()的返回值确定是树枝节点还是叶子节点，但是这样做的好处是符合依赖倒置的原则，而且遍历的时候不需要强转类型；

1. 组合模式的遍历

getChildren()方法换成getParent()

### 装饰模式

定义：Decorator Pattern，定义Attach additional responsibilities to an object dynamically keeping the same interface. Decroators provide a flexible alternative to subclassing for extending functionality.

四个角色：

1. Component抽象挂件
2. ConcreteComponent具体挂件
3. Decorate装饰角色
4. Concrete具体装饰角色

细节：

优点：

装饰类和被装饰类可以独立发展；

缺点：

多层装饰比较复杂，像剥洋葱一样，出现问题很难找；

### 门面模式

定义：

1. Façade Pattern，也叫外观模式，定义：Provide a unified interface to a set of interface in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

细节：

1. 其实就是找一个门面对象来关联其他业务对象，作为一个同一的接口，你想访问业务对象，必须通过门面对象，门面对象没有的对象，你无法访问得到。
2. 当一个门面的代码超过200行时，可以考虑拆分多个门面。按照功能拆分比较好，如数据库的crud操作
3. 子系统可以提供不同的访问路径；
4. 相同的代码尽量只写一次；
5. 门面不参与子系统内的业务逻辑；

优点：

1. 减少系统的相互依赖；
2. 提高灵活性；
3. 提高安全性

想让你访问子系统的哪些业务就开通哪些逻辑，不在门面上开通的方法，你休想访问得到；

缺点：

不符合开闭原则，一旦发现有小错误想要修改，必须修改门面角色的代码，这个风险相当大；

使用场景：

1. 子系统相对独立，外界对子系统的访问只要黑箱操作即可；
2. 预防低水平人员带来的风险扩散，哈哈，这是在说我吗？^\_^；

### 享元模式

定义：

1. Flyweight Pattern，池技术的重要实现方式，定义：Use sharing to support large numbers of fine-grained objects efficient.
2. 细粒度的对象：数量对且性质相近
3. 内部对象 可共享的信息，储存在享元对象内部并且不会随着环境改变而改变，如实体类中的id, postAddress等，他们可以作为一个对象的东陶附加信息，不必直接储存在具体的某个对象中，属于可以共享的部分；
4. 外部状态 是对象得以以来的标记，随环境变化而变化，不可以共享
5. 共享对象池
6. 容器定义
7. 提供客户端访问的接口，池中有可用对象的时候，可以直接从池中获取，否则建立一个新的对象，并放置到池中；
8. 四个对象
9. Flyweight抽象享元角色
10. ConcreteFlyweight具体享元角色
11. unsharedConcreteFlyweight不可具体享元角色 不存在外部对象，或者由于安全要求，如线程安全不能使用共享技术的对象，一般不会出现在享元工厂中；
12. FlyweightFactory享元工厂 生产细粒度的对象

细节：

1. 内存溢出的可能：
2. 内存泄漏，无意识的代码缺陷，导致内存泄漏，jvm不能连续获得内存空间；
3. 对象太多。
4. 内存溢出的可能：
5. @Deprecated注解，用于标识该段代码已经过时，不要再使用了。作为一个有素养的coder，写代码的时候应该保持历史的原貌，此举有助于版本向下兼容，特别是在产品级研发中；
6. 在程序开发过程中，确认只需要一次赋值的属性设置为final类型，避免无意的修改导致逻辑混乱，特别时Session级的常量或者变量；
7. 外部状态作为对象池唯一标识的key，代码如下：

//抽象享元角色

public abstract class Flyweight{

//内部状态

private String intrinsic;

//外部状态

protected final String extrinsic;

public Flyweight(String extrinsic) {

this.extrinsic=extrinsic;

}

//定义业务操作

public abstract void operate();

/\*\*

\* @return the intrinsic

\*/

public String getIntrinsic() {

return intrinsic;

}

/\*\*

\* @param intrinsic the intrinsic to set

\*/

public void setIntrinsic(String intrinsic) {

this.intrinsic = intrinsic;

}

}

//具体享元角色

public class ConcreteFlyweight1 extends Flyweight{

//接收外部状态

public ConcreteFlyweight1(String extrinsic) {

super(extrinsic);

}

//跟据外部状态进行逻辑处理

public void operate() {

//业务逻辑

}

}

public class ConcreteFlyweight2 extends Flyweight{

//接收外部状态

public ConcreteFlyweight2(String extrinsic) {

super(extrinsic);

}

//跟据外部状态进行逻辑处理

public void operate() {

//业务逻辑

}

}

//享元工厂

public class FlywightFactory{

//定义一个容器

private static HashMap<String, Flyweight> pool=new HashMap()<String, Flyweight>;

//巷元工厂

public static Flyweight getFlyweight(String extrinsic) {

Flyweight flyweight=null;

if(pool.containsKey(extrinsic)) {

flyweight=pool.get(extrinsic);

}else {

flyweight=new Flyweight(extrinsic);

pool.put(extrinsic, flyweight)

}

return flyweight

}

}

优点：

1. 大大减少应用 程序创建的对象，降低程序内存的占用，需要分离处外部状态和内部状态，而且外部状态具有固话特性（即从数量上是可以被有限穷举的）；

拓展：

1. 线程安全的问题；

主要原因是享元对象的数量太少，导致每个线程都到对象池中获得对象，然后修改其属性。目前没有什么参考的标准，只能依靠经验，在需要的地方考虑一下线程安全，在大部分场景中都不需要考虑。

1. 性能平衡；

将外部状态作为一个类来使用，但此时需要注意的是，如果一个对象作为一个Map类的键值，一定要确保重写了equals和hashCode方法，否则会出现通过简直搜索失败的情况，例如map.get(object)、map.contain(object)等会返回失败的结果，代码如下：

//外部状态类

public class ExtrinsicState{

private String subject;

private String location;

/\*\*

\* @return the subject

\*/

public String getSubject() {

return subject;

}

/\*\*

\* @return the location

\*/

public String getLocation() {

return location;

}

/\*\*

\* @param subject the subject to set

\*/

public void setSubject(String subject) {

this.subject = subject;

}

/\*\*

\* @param location the location to set

\*/

public void setLocation(String location) {

this.location = location;

}

@Override

public boolean equals(Object obj) {

if (obj instanceof ExtrinsicState) {

ExtrinsicState extrinsicState=(ExtrinsicState)obj;

return (extrinsicState.getLocation().equals(this.location) &&

extrinsicState.getSubject()().equals(this.subject));

}

return false;

}

@Override

public int hashCode() {

// TODO Auto-generated method stub

return this.location.hashCode()+this.subject.hashCode();

}

}

### 代理模式

定义：

1. Proxy Pattern，又叫委托模式，定义: Provide a surrogate or placeholder for another object to control access to it
2. 三种角色，Subject抽象主题角色，RealSubject具体主题角色，Proxy代理主题角色。
3. 具体主题角色通过Proxy代理的构造函数传入；
4. 动态代理：动态代理实现代理的职责，业务逻辑由Subject实现，两者没有必然的相互耦合关系，通知Advice从另外一个切面切入，最终在高层模块也就是client进行耦合，完成逻辑的封装任务。
5. 其实代理模式最重要的特点是可以在不改变已有代码的情况下（主题类），增强或者控制对象的行为。代理模式侧重于拦截，装饰模式侧重与加强

细节：

1. 具体代码：

//抽象主题类

public interface Subject{

public void request();

}

//真实主题类

public class RealSubject extends Subject{

public void request() {

//具体的逻辑业务

}

}

//代理类

public class Proxy extends Subject{

private Subject subject=null;

public Proxy(){

this.subject=new Proxy();

}

public Proxy(Object object){

this.subject=object;

}

public void request() {

this.before();

this.subject.request();

this.after();

}

public void before() {

//预处理

}

public void after() {

//后处理

}

}

1. Proxy.newProxyInstance

Proxy类中的方法Proxy.newProxyInstance

(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)是在运行时生成一个类，这个类会实现你指定的一组接口，而这个类没有.java文件，是在运行时生成的，你也不用去关心它是什么类型的，你只需要知道它实现了哪些接口即可。Proxy类的newInstance()方法有三个参数：

1. ClassLoader loader：它是类加载器类型，你不用去理睬它，你只需要知道怎么可以获得它就可以了：MyInterface.class.getClassLoader()就可以获取到ClassLoader对象，没错，只要你有一个Class对象就可以获取到ClassLoader对象；
2. Class[] interfaces：指定newProxyInstance()方法返回的对象要实现哪些接口，没错，可以指定多个接口，例如上面例子只我们只指定了一个接口：Class[] cs = {MyInterface.class};
3. InvocationHandler h：它是最重要的一个参数！它是一个接口！它的名字叫调用处理器！InvocationHandler接口只有一个方法，即invoke()方法！它是对代理对象所有方法的唯一实现。也就是说，无论你调用代理对象上的哪个方法，其实都是在调用InvocationHandler的invoke()方法。
4. InvocationHandler.invoke

InvocationHandler的invoke()方法的参数有三个：

1. Object proxy：代理对象，也就是Proxy.newProxyInstance()方法返回的对象，通常我们用不上它。就本人目前的水平看来，这个proxy参数并没有什么作用，在整个动态代理机制中，并没有用到InvocationHandler中invoke方法的proxy参数。而传入的这个参数实际是代理类的一个实例。我想可能是为了让程序员在invoke方法中使用反射来获取关于代理类的一些信息吧。，；
2. Method method：表示当前被调用方法的反射对象，例如mi.fun1()，那么method就是fun1()方法的反射对象；
3. Object[] args：表示当前被调用方法的参数，当然mi.fun1()这个调用是没有参数的，所以args是一个零长数组。

最后要说的是invoke()方法的返回值为Object类型，它表示当前被调用的方法的返回值，当然mi.fun1()方法是没有返回值的，所以invoke()返回的就必须是null了。

1. 想象中的类

InvocationHandler接口只有一个方法，即invoke()方法！它是对代理对象所有方法的唯一实现。也就是说，无论你调用代理对象上的哪个方法，其实都是在调用InvocationHandler的invoke()方法。

class X implements MyInterface {

　　private InvocationHandler h;

　　public X(InvocationHandler h) {

　　　　this.h = h;

　　}

　　public void fun1() {

　　　　h.invoke();

　　}

　　public void fun2() {

　　　　h.invoke();

　　}

}

拓展：

1. 透明代理

用户不用设置服务器地址，就可以直接访问，也就是代理服务器对用户来讲是透明的，不用知道他的存在。

1. 普通代理

需要用户自己设置代理服务器的IP地址，用户必须知道代理的存在。而且要求客户端只能访问代理角色，而不能访问真实角色；“屏蔽了真实角色的变更对高层模块的影响，真实角色想怎么改就怎么改，对高层次模块没有影响。”这句话让我理解就是比较方便更改真实角色的类名，或者更改真实角色的类名的实例名字，而尽量少的改动其他代码，减少有毒代码的产生。

1. 强制代理

调用者直接调用真实角色，而不用关心代理是否存在，其代理的产生由真实角色决定。一般思维都是通过代理找到真实角色，而强制代理却要“强制”必须通过真实角色查找代理角色，否则不能访问。“找明星时明星很忙让你去找他的代理，不能随便找其他的代理”

1. 代理也是有个性的

代理可以实现多个不同的接口，来完成不同的任务。代理的目的是在目标对象方法的基础上作增强，这种增强的本质是对目标对象的方法进行拦截和过滤。

1. 动态代理

年度压轴大戏，目前有一个非常流行的名称，叫AOP Aspect Oriented Programming，“面向横切面编程”，其核心就是采用了动态代理机制。该机制宣称“我已经实现该接口下的所有方法”

你只需要知道，Proxy类的静态方法newProxyInstance()方法返回的方法是实现了指定接口的实现类对象，甚至你都没有看见实现类的代码。

动态代理实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为 。相对静态代理， 如果不同接口的某些类想使用代理模式来实现相同的功能 ，静态代理需要实现多个代理类，动态代理只需要实现一个代理类。

## 行为类模式

### 中介者模式

定义

1. Mediator Pattern，定义：Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
2. 角色：
3. Mediator 抽象中介者
4. ConcreteMediator 具体中介者
5. Colleague 抽象同事
6. ConcreteColleague 具体同事 具体同事有两种方法，分别是独立方法（self-Method）和依赖方法（Dep-Method）

细节

网络的基本知识，网络的拓扑类型主要分为三种类型：总线型，环形和星型，一旦遇到蜘蛛网类型的结构最好把他梳理成星型结构；

最佳实践

一般当相关联的对象大于2的时候才考虑使用中介者模式，同时需要提防中介者的膨胀问题；

### 命令模式

定义：

1. Command Pattern，定义：Encapsulate a request as a Object, thereby letting you parameterize cliets with different requests, queue or log requests, and support undoable operations.
2. 三个角色：
3. Reciver接收者
4. Commander命令者
5. Invoke调用者角色

细节：

1. 代码实现：

//抽象接收者

public abstract class Reciver{

public abstract doSomething();

}

//具体接收者1

public class Reciver1 extends Reciver{

public void doSomething() {

}

}

//具体接收者2

public class Reciver2 extends Reciver{

public void doSomething() {

}

}

//抽象命令

public abstract class Command{

public abstract void execute();

}

//具体命令

public class Delete extends Command{

private Reciver reciver=null;

/\*\*

\* @param reciver the reciver to set

\*/

public void setReciver(Reciver reciver) {

this.reciver = reciver;

}

public abstract void execute() {

this.reciver.doSomething();

}

}

public class Copy extends Command{

private Reciver reciver=null;

/\*\*

\* @param reciver the reciver to set

\*/

public void setReciver(Reciver reciver) {

this.reciver = reciver;

}

public abstract void execute() {

this.reciver.doSomething();

}

}

//执行者

public class Invoker{

public Command command=null;

/\*\*

\* @param command the command to set

\*/

public void setCommand(Command command) {

this.command = command;

}

public void action() {

this.command.execute();

}

}

//场景类

public class Client{

Command command=new Copy();

Reciver reciver=new Reciver1();

command.setReciver(reciver);

Invoker invoker=new Invoker();

invoker.setCommand(command);

invoker.action();

}

1. 实现过程中最好不要把接收者暴露给客户；

### 责任链模式

定义：

Responsibility Chain pattern定义：Avoid coupling the sender of a request to its receiver by giving more than one object a change to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

抽象的处理者实现了三个职责：

定义一个请求的处理方法handleMessage，唯一对外开放的方法；

定义一个链的编排方法setNext，设置下一个处理者；

定义了具体请求者必须实现的两个方法：定义自己能够处理的getHanderLever和具体的处理任务echo；

细节：

1. 代码：

//抽象处理者

public abstract class Handler{

private Handler nextHandler;

public void setNext(Handler handler) {

this.nextHandler=handler;

}

public static Response handleMessage(Request request) {

Response response=null;

if(this.getHandlerLever.equal(request.getRequestLevel)) {

response=this.echo(request);

}else {

if(this.nextHandler !=null) {

response=this.nextHandler.handleMessage(request);

}else {

}

}

return response;

}

protected abstract Level getHandlerLever();

protected abstract Response echo(Request request);

}

//具体处理者

public class ConcreteHandler1 extends Handler{

protected Response echo(Request request) {

//完成逻辑的处理

return null;

}

protected Lever getHandLerLever() {

//设置自己的级别

return null;

}

}

public class ConcreteHandler2 extends Handler{

protected Response echo(Request request) {

//完成逻辑的处理

return null;

}

protected Lever getHandLerLever() {

//设置自己的级别

return null;

}

}

public class ConcreteHandler3 extends Handler{

protected Response echo(Request request) {

//完成逻辑的处理

return null;

}

protected Lever getHandLerLever() {

//设置自己的级别

return null;

}

}

//相关框架代码

public class Level{

//定义一个请求和处理登记;

}

public class Request{

//请求的等级

public Level getRequestLevel() {

return null;

}

}

public class Reponse{

//处理者返回的数据

}

//场景类

public class Client{

public static void main(Static[] args) {

Handler handler1 =new ConcreteHandler1();

Handler handler2 =new ConcreteHandler2();

Handler handler3 =new ConcreteHandler3();

handler1.setNext(handler2);

handler2.setNext(handler3);

Response response = handler1.handlerMessage(new Request());

}

}

1. 融合了模板模式的优点；

优点：

将请求和处理分开，请求者不知是谁处理的，处理者也不必知道请求的全貌，得到解耦的效果；

缺点：

1. 性能问题，每次都要从链头遍历链尾，特别是链比较长的时候，性能是一个很大的问题。为了避免无意识的长递归破坏系统，一般需要在handler中设置一个最大节点数；
2. 调试不是很方便，有时候都不知道是那个环节出了问题；

### 策略模式

定义：

1. Strategy Pattern，定义：Define a family of algorithms, encapsulate each one, and make them interchangeable.
2. 三个角色：

Context封装角色

Strategy抽象类策略角色

ConcreteStrategy具体策略角色

细节：

1. 代码示例：

//抽象的策略角色

public interface Strategy{

public void doSomething();

}

//具体的策略角色

public class ConcreteStrategy1 extends Strategy{

public void doSomething() {

System.out.println("具体策略一");

}

}

public class ConcreteStrategy2 extends Strategy{

public void doSomething() {

System.out.println("具体策略二");

}

}

//封装角色

public class Content{

private Strategy strategy;

public Content(Strategy strategy){

this.strategy=strategy;

}

public void doAnything() {

this.strategy.doSomething();

}

}

//高层模块

public class Client{

public static void main(String[] args) {

Strategy strategy=new ConcreteStrategy1;

Content content=new Content(strategy);

content.doAnything;

}

}

1. 采用面向对象的继承和多态机制；
2. 变形：枚举策略

优点：

1. 策略可以自由切换；
2. 避免使用多重条件判断；
3. 拓展性良好；

缺点：

策略类数量增多，但是复用的可能性很小；

所有的策略类都必须对外暴露，上层模块必须知道有哪些策略，然后才能决定使用哪一个策略，这与迪米特原则是相违背的。

如果一个系统的策略类数量超过4个，需要考虑使用混合模式，解决策略类膨胀和对外暴露的问题，否则日后的系统维护是一个烫手山芋；

使用场景：

算法需要自由切换的场景；

需要屏蔽算法规则的场景；

### 迭代器模式

定义：

1. Iterator pattern，定义：Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
2. 容器：能容纳对象的所有类型都可以称为对象，如collection集合类型，Set类型等；
3. 迭代器模式的产生是为了解决遍历这些容器中的元素
4. 四个角色：
5. Iterator抽象迭代器
6. ConcreteIterator具体迭代器
7. Aggregate抽象容器
8. ConcreteAggregate具体容器

细节：

1. 代码示例，注意迭代器的删除和添加方法都需要把当前图标加1：

//抽象迭代器

public interface Iterator{

public Object next();

public boolean hasNext();

public boolean remove();

}

//具体迭代器

public class ConcreteIterator implements Iterator{

private Vector vector=new Vector();

//定义当前游标

public int cursor=0;

public ConcreteIterator(Vector vector) {

this.vector=vector;

}

@Override

public Object next() {

Object result=null;

if(hasNext()) {

result=this.vector.get(this.cursor++);

}else {

result=null;

}

return result;

}

@Override

public boolean hasNext() {

// TODO Auto-generated method stub

if(this.cursor==vector.size()) {

return false;

}else {

return true;

}

}

@Override

public boolean remove() {

// TODO Auto-generated method stub

return false;

}

}

//抽象容器

public interface Aggregate{

public void add(Object object);

public void remove(Object object);

public Iterator iterator();

}

//具体容器

public class ConcreteAggregate implements Aggregate {

private Vector vector=null;

@Override

public void add(Object object) {

this.vector.add(object);

}

@Override

public void remove(Object object) {

this.vector.remove(object);

}

@Override

public Iterator iterator() {

// TODO Auto-generated method stub

return new ConcreteIterator(this.vector);

}

}

//场景类

public class Client{

public static void main(String[] args) {

Aggregate aggregate=new ConcreteAggregate;

aggregate.add("abc");

aggregate.add("def");

aggregate.add("ghi");

Iterator iterator=aggregate.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

1. 从JDK1.2开始增加java.util.Iterator这个接口，并逐步把Iterator应用到各个聚集类（collection）中；
2. 这个迭代器模式有点落后了，一些大师级的人物甚至建议把迭代器模式从23种设计模式中删除，目前基本很少有项目再独立写迭代器了，如果是自己开发，尽量不要自己去写迭代器模型；

### 观察者模式

定义：

1. Observer Pattern，也叫做发布订阅模式，又叫触发链模式，定义：Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
2. 四个角色：
3. Subject被观察者
4. Observer观察者
5. ConcreteSubject具体被观察者
6. ConcreteObserver具体观察者

细节：

1. 程序中使用的ArrayList和Vector区别不是很大，只是ArrayList是线程异步的，不安全，而Vector是线程同步的，安全，就这点区别，代码如下：

//被观察者

public abstract class Subject{

private Vector<Observer> vector=new Vector();

public void addObserver(Observer observer) {

this.vector.add(observer);

}

public void removeObserver(Observer observer) {

this.vector.remove(observer);

}

public void notifyObserver() {

for(Observer observer:this.vector)

observer.update();

}

}

//具体观察者

public class ConcreteSubject extends Subject{

public void doSomething() {

//做什么事情

super.notifyObserver();

}

}

//观察者

public interface Observer{

public void update();

}

//具体观察者

public class ConcreteObserver implements Observer{

public void update() {

System.out.println("收到信息，正在处理");

}

}

//场景类

public class Client{

public static void main(String[] args) {

ConcreteSubject concreteSubject =new ConcreteSubject();

Observer observer1=new Observer();

Observer observer2=new Observer();

concreteSubject.add(observer1);

concreteSubject.add(observer2);

concreteSubject.doSomething();

}

}

1. 一个系统中最多出现一个既是观察者又是被观察者的角色，也就是说消息最多被转发一次，这样的话比较好控制。
2. 他和责任链模式最大的区别在于消息的不变性，责任链模式在消息传递过程中消息是不变的，而观察者模式是会改变消息的；

优点：

1. 观察者和被观察者之间的抽象解耦；

观察者和被观察者的拓展都变得非常方便；

1. 建立了一套触发机制；

缺点：

1. 开发效率和运行效率的问题，由于是线程同步的，一旦一个观察者运行卡壳了，会影响整体的执行效率；

应用场景：

1. 事件的多级触发；

拓展：

1. Java.util.Observable被观察者类，可以被继承，拥有super.setChanged()，super.notifyObservers(Object obj)， super.deleteObservers()，super. addObservers (Object Observer)等方法；
2. Java.util. Observer观察者接口，可以被实现，拥有super.update (Observable o, Object arg)；

调用addObserver(观察者)方法让某指定观察者观察被观察者；

只有在setChange()被调用后，notifyObservers()才会去调用update()，否则什么都不干。一旦使用super.notifyObservers(Object arg)就会调用super.update ()的方法。Observable o,称为事件源，而传过来的 Oberverable o 其实可以通过if语句用来判断到底是谁发过来的消息。形参Object arg，对应一个由notifyObservers(Object arg)传递来的参数，当执行的是notifyObservers()时，arg为null。。在Observer对象销毁前一定要用deleteObserver将其从列表中删除，也就是在onDestroy()方法中调用deleteObserver()方法。不然因为还存在对象引用的关系，Observer对象不会被垃圾收集，造成内存泄漏，并且已死的Observer仍会被通知到，有可能造成意料外的错误，而且随着列表越来越大，notifyObservers操作也会越来越慢。

### 备忘录模式

定义：

1. Memento Pattern，也叫外观模式，定义：Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.
2. 三个角色：
3. Originator发起人角色；
4. Memento备忘录角色，备忘录被创建的时候就对发起人角色进行备份；
5. Caretaker备忘录管理员角色，用于创建和返回备忘录角色；

细节：

1. 保留和回复状态应该由另外一个类来承担，我们把这个类取名为备忘录，
2. 备忘录被创建的时候就对发起人角色进行备份，代码如下：

//发起人角色

public class Originator{

private String state="";

/\*\*

\* @return the state

\*/

public String getState() {

return state;

}

/\*\*

\* @param state the state to set

\*/

public void setState(String state) {

this.state = state;

}

public Memento creatMemento() {

return new Memento();

}

public void restoreMemento(Memento memento) {

this.setState(memento.getState());

}

}

//备忘录角色

public class Memento{

private String state="";

public Memento(String state) {

this.state= state;

}

/\*\*

\* @return the state

\*/

public String getState() {

return state;

}

/\*\*

\* @param state the state to set

\*/

public void setState(String state) {

this.state = state;

}

}

//备忘录管理员角色

public class Caretaker{

private Memento memento;

/\*\*

\* @return the memento

\*/

public Memento getMemento() {

return memento;

}

/\*\*

\* @param memento the memento to set

\*/

public void setMemento(Memento memento) {

this.memento = memento;

}

}

//场景类

public class Client{

public static void main(String[] args) {

Originator originator=new Originator();

Caretaker caretaker=new Caretaker();

//创建一个备忘录

caretaker.setMemento(originator.creatMemento);

//恢复一个备忘录

originator.restoreMemento(caretaker.getMemento);

}

}

1. 不要频繁建立备份的场景使用备忘录模式，不如在一个for循环中，原因有二：一是控制不了备忘录的对象数量，二是大对象的建立需要消耗资源，从系统的系能方面考虑。哥哥6

使用场景：

1. 提供一个可回滚（roolback）的操作；
2. 需要监控的副本场景；
3. 数据库连接的事务管理用的就是备忘录模式；

备忘录模式的拓展

1. Clone方式的备忘录——融合备忘录的发起人角色

//融合备忘录的发起人角色

//发起人角色Originator

public class Originator implements Cloneable{

private String state="";

/\*\*

\* @return the state

\*/

public String getState() {

return state;

}

/\*\*

\* @param state the state to set

\*/

public void setState(String state) {

this.state = state;

}

public Memento creatMemento() {

return this.clone();

}

public void restoreMemento(Originator originator) {

this.setState(originator.getState());

}

public Originator clone() {

try {

return (Originator)super.clone();

} catch (Exception e) {

e.printStackTrace();

}

}

}

//备忘录管理员角色

public class Caretaker{

private Originator originator;

/\*\*

\* @return the originator

\*/

public Originator getMemento() {

return originator;

}

/\*\*

\* @param originator the originator to set

\*/

public void setMemento(Originator originator) {

this.originator = originator;

}

}

1. Clone方式的备忘录——融合备忘录的发起人和管理员角色

//融合备忘录的发起人和管理员角色

//发起人角色Originator

public class Originator implements Cloneable{

private String state="";

private Originator backup=null;

/\*\*

\* @return the state

\*/

public String getState() {

return state;

}

/\*\*

\* @param state the state to set

\*/

public void setState(String state) {

this.state = state;

}

public void creatMemento() {

this.backup=this.clone();

}

public void restoreMemento() {

this.setState(this.backup.getState());

}

public Originator clone() {

try {

return (Originator)super.clone();

} catch (Exception e) {

e.printStackTrace();

}

}

}

//场景类

public class Client{

public static void main(String[] args) {

Originator originator=new Originator();

//创建一个备忘录；

originator.setState("初始状态");

//创建一个备忘录备份；

originator.creatMemento();

originator.setState("更改后的状态");

originator.restoreMemento();

}

}

1. 多备份备忘录

### 访问者模式

定义：

1. Visitor Pattern，定义：Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of elements on which it operation.
2. 五个角色：
3. Visitor抽象访问者

抽象类或者接口，声明访问者可以访问哪些元素，具体在元素中就是visit方法参数定义哪些对象可以被访问；

1. ConcreteVisitor具体访问者
2. Element抽象元素

抽象类或者接口，声明接受哪一类访问者的访问，具体在元素中就是accept方法参数定义；

1. ConcreteElement具体元素

实现accept()的方法基本上是visitor.visit(this);

1. ObjectStructure结构对象

用于产生不同的容器，如List、Set、Map等；

1. 建立自己的一套过滤器和拦截器；

细节：

1. 主要用来修饰报表的展现方式，不同的visitor就意味者不同的展现风格，具体代码如下：

//抽象元素

public abstract class Element{

public abstract void doSomething();

public abstract void accept(Visitor vistor);

}

//具体元素

public class ConcreteElement1 extends Element{

public void doSomething() {

};

public void accept(Visitor vistor) {

};

}

public class ConcreteElement2 extends Element{

public void doSomething() {

};

public void accept(Visitor vistor) {

};

}

//抽象访问者

public interface Vistor{

public abstract void visit(ConcreteElement1 concreteElement1);

public abstract void visit(ConcreteElement2 concreteElement2);

}

//具体访问者

public class ConcreteVistor implements Vistor {

public void visit(ConcreteElement1 concreteElement1) {

concreteElement1.doSomething();

};

public void visit(ConcreteElement2 concreteElement2) {

concreteElement2.doSomething();

};

}

//具体场景

public class Client{

public static void main(String[] args) {

Element element = new ConcreteElement1();

element.accept(new Visitor);

}

}

1. 双分派Double dispatch

单分派single dispatch，其处理一个操作是根据请求者的名称和接收到的参数决定的，在java中有静态绑定和动态绑定之说，它的实现是依据重载（overload）和覆写（override）实现的。重载在编译期跟据参数的个数和类型就决定了要调用那个方法，这是静态绑定。

而双分派则意味着得到执行的操作决定于请求的种类和两个接收者的类型。

优点：

1. 符合单一职责原则。具体的元素角色也就是Element抽象类的两个子类负责数据的加载，而Visitor类则负责报表的展现，两个不同职责非常明确的分离出来。
2. 优秀的拓展性

由于职责分开，继续增加对数据的操作时非常快捷的，例如现在要增加一份给老师的报表，有要求报表的格式不一样，则直接在visitor类中增加一个方法，传递数据后整理打印。

1. 灵活性非常高

缺点：

1. 具体原色对访问者公布细节

访问者要访问一个类必然要知道该类的方法和数据，也就是访问者关注了其他类的内部数据，这是迪米特法则所不允许的；

1. 具体元素变更困难
2. 违背了依赖倒置的原则

访问者依赖的是具体元素，而不是抽象元素；

使用场景：

1. 一个对象结构包含有很多类的对象，他们有不同的接口，而你想对这些对象进行具体的很多不一样的操作；
2. 业务规则需要遍历多个不同的对象（迭代器模式只能访问同类或者同接口的对象）

拓展：

1. 统计功能；
2. 多个访问者

一堆数据从不同的角度进行分析，即数据挖掘。

### 状态模式

定义：

1. Status Pattern，定义：Allow an Object to alert its internal state changes. The object will appear to change its class.
2. 三个角色：
3. State抽象状态角色；
4. ConcreteState具体状态角色；
5. Context环境角色，重点抓住该类去理解程序；

细节：

1. Context是一个环境角色，放在各状态里面可以储存各状态下转换状态的信息，并调用转换动作，但是为什么不直接调用自己的动作，而通过自己间接调用具体状态角色的动作呢？我的理解是用于跟据迪米特原则，上下文知道状态转换及动作的信息越少越好，那只好调用具体状态角色的动作喽。同时，环境角色有代理的意味，代码如下：

//抽象状态角色

public abstract class Status{

protected Context context;

public void setContext(Context context) {

this.context=context;

}

public abstract void handle1();

public abstract void handle2();

}

//具体状态角色

public class ConcreteStatus1 extends Status{

public void handle1() {

//本状态下必须处理的逻辑

};

public void handle2() {

//设置当前的状态为state2

super.Context.setCurrentStatus(Context.ConcreteStatus1);

//过渡到state2，由context实现

super.Context.handle2

};

}

public class ConcreteStatus2 extends Status{

public void handle1() {

super.context.setCurrentStatus(Context.concreteStatus2);

super.context.handle2();

};

public void handle2() {

//本状态下必须处理的逻辑

};

}

//具体环境角色

public class Context{

//定义状态

public final static ConcreteStatus1 =new ConcreteStatus1();

public final static ConcreteStatus2 =new ConcreteStatus2();

//当前状态

private Status currentStatus=null;

/\*\*

\* @return the currentStatus

\*/

public Status getCurrentStatus() {

return currentStatus;

}

/\*\*

\* @param currentStatus the currentStatus to set

\*/

public void setCurrentStatus(Status currentStatus) {

this.currentStatus = currentStatus;

}

public void handle1(){

this.currentStatus.handle1();

}

public void handle2(){

this.currentStatus.handle2();

}

}

//场景类

public class Client{

public static void main(String[] avgs) {

Context context=new Context();

//定义初始状态

context.setCurrentStatus(new ConcreteStatus1);

context.handle1;

handle1.handle2;

}

}

优点：

1. 避免过多的使用判断语句使得程序变得过于复杂，提高系统的可维护性；
2. 遵循设计原则，很好地体现开闭原则和单一职责原则，每个状态都是子类，你要增加状态或者修改状态，只要增加子类或者修改子类即可；
3. 封装性非常好，状态的变换放置到类中实现，内部的调用不知道类内是如何

缺点：

1. 类过多容易膨胀，管理是一个问题，所以类状态最好不要超过5个；

使用场景：

1. 行为随状态改变的场景；
2. 条件和分支过多的时候；

### 解释器模式

定义：

1. Interpreter Pattern，一种按照规定语法进行解析的方案，在现在项目中用的比较少，定义：Given a language, define a representation for its grammar along with an interpreter that user the representation to interpret sentences in the language.
2. 四个角色：
3. AbstractionExpression抽象解释器
4. TerminalExpression终结符表达式
5. NonterminalExpression非终结表达式
6. Context环境角色

细节：

1. 改进型，可迭代，可进行多项的四则运算；

//抽象表达式

public abstract class Expression{

public abstract Object Interpreter(Context ctx);

}

//终结符表达式

public class TerminalExpression extends Expression{

private Object parameter;

/\*\*

\* @return the parameter

\*/

public Object getParameter() {

return parameter;

}

/\*\*

\* @param parameter the parameter to set

\*/

public void setParameter(Object parameter) {

this.parameter = parameter;

}

public Object Interpreter(Context ctx) {

return null;

}

}

//非终结符表达式

public class NonTerminalExpression1 extends Expression{

public NonTerminalExpression1(Expression... expressions) {

// expressions必须是TerminalExpression类

}

public Object Interpreter(Context ctx) {

//调用TerminalExpression的Interpreter的方法继续逻辑运算

return null;

}

}

public class NonTerminalExpression2 extends Expression{

public NonTerminalExpression2(Expression... expressions) {

// expressions必须是TerminalExpression类

}

public Object Interpreter(Context ctx) {

//调用TerminalExpression的Interpreter的方法继续逻辑运算

return null;

}

}

//场景类

public class Client{

public static void main(String[] args) {

Context ctx = new Context();

char[] charArray=ctx.toCharArray();

//通常定义一个语法容器，来容纳一个具体的表达式

Stack<Expression> stack=null;

Expression left=new TerminalExpression();//实际类型是TerminalExpression

Expression right=new TerminalExpression();//实际类型是TerminalExpression

Object newLeft=new Object();

Expression exp=null;//

for(int i=0;i<charArray.length();i++) {

if (charArray[i]=="NonTerminalExpression1"){

left.setParameter(stack.pop);

right.setParameter(charArray[++i]);

exp=new NonTerminalExpression1(left,right);

stack.push(exp.Interpreter(ctx));//把每一步的结果保存下来

}else if (charArray[i]=="NonTerminalExpression2"){

left.setParameter(stack.pop);

right.setParameter(charArray[++i]);

exp=new NonTerminalExpression2(left,right);

stack.push(exp.Interpreter(ctx));//把每一步的结果保存下来

}else if(charArray[i]=="TerminalExpression") {

stack.push(charArray[i])

}

}

}

}

优点：

1. 可拓展语法，只要添加非终结符表达式类就可以

缺点：

1. 会引起类膨胀；
2. 递归的算法在调试时存在非常大的不方便；
3. 由于使用了大量的循环和递归，效率是个不容忽视的问题；

最佳实践：

1. 其实现在有很多开源的框架实现解释器的功能，实在没有必要自己从头开始；

## 创建类模式大PK

### 单例模式VS原型模式

其实这两个模式都不容易混淆，单例模式是用静态的方法返回内存唯一的对象，原型模式是继承Colonable的对象然后覆写clone()方法返回一个几乎一样的对象。

### 工厂方法模式和建造者模式

1. 工厂方法模式：注重整体对象的创建，具体生产出来的对象不必具有相同的属性和行为，生产产品的粒度比较大；
2. 建造者模式：注重部件构建的过程，旨在一步一步地精确创造处一个复杂的对象，拥有对象诸如Director(提供建造的方法，高层模块传入何种Builder就返回哪种Product)，Builder(建造方法或者建造顺序，最后返回相应的实体类)和product实体类。生产产品的粒度比较小。

### 抽象工厂模式和建造者模式

1. 抽象工厂模式：实现对产品家族的创建，具有不同分类维度产品的组合。并不关心产品生产的过程或者是顺序，只关心什么产品由什么工厂生产即可；
2. 建造者模式：注重部件构建的过程，旨在一步一步地精确创造处一个复杂的对象，拥有对象诸如Director(提供建造的方法，高层模块传入何种Builder就返回哪种Product)，Builder(建造方法或者建造顺序，最后返回相应的实体类)和product实体类。生产产品的粒度比较小。

## 结构类模式大PK

### 代理模式VS装饰模式

1. 相同点：代理类和相应的被代理类都有相同的接口，代理类使用时需要传入被代理类的对象；
2. 代理模式：可以判断是否执行被代理类的行为，就是加上if语句进行判断，如果可执行就原汁原味地执行；
3. 装饰模式：在增强原来被代理类的行为，不做准入条件的判断和准入参数的过滤

### 装饰模式VS适配器模式

1. 装饰模式：装饰具有相同血缘关系（相同接口）的类；
2. 适配器模式：修饰不具有血缘关系的类；

当出现子类继承父类的同时又去实现一个成员变量、方法与父类相同的接口时，重写的是父类的方法，接口的方法被隐藏。

如果子类没有定义跟父类、接口一样的成员变量，则需要通过super关键字调用父类的成员变量，接口的成员则无法调用，而且这个时候不用super关键的话，IDE会提示使用的成员变量是ambiguous（暧昧的、混淆的），无法通过编译；

如果子类同时还定义了跟父类、接口一样的成员变量，可以直接调它本身的成员变量，能过super关键字可以调父类的成员变量，而接口的成员变量则无法调用。

## 行为类模式大PK

### 命令模式和策略模式

1. 命令模式：多了一个接收者(Receiver)角色，侧重对动作的解耦，把一个动作的行为分为执行对象（接收者角色），执行行为（命令角色）。调用者调用命令类让接收者干活，命令类在高层模块中通过参数传入调用者，接收者则在命令类的抽象类中已经被传入，可以参考数据库中的业务类，如业务提交和业务撤销；
2. 策略模式：封装算法，本身每个算法就是一个原子业务，算法独立可以相互替换，让行为的变化独立于拥有行为的客户，使用哪种算法由高层模块决定；

### 策略模式和状态模式

1. 策略模式：封装算法，本身每个算法就是一个原子业务，算法独立可以相互替换，让行为的变化独立于拥有行为的客户，使用哪种算法由高层模块决定。环境角色主要用于在高层模块中传入参数（对象）进行切换算法；
2. 状态模式：关注的是状态按照指定的顺序进行无缝切换，旨在解决内部状态改变而引起的行为改变，它的出发点需要初始化。环境角色用于保存切换的状态。通常命令模式用于代替复杂的判断语句，虽然容易拓展，但是一般不会进行大规模的扩张和修正。

### 观察者模式和责任链模式

观察者模式 消息是属于触发传递的，类似于蝴蝶效应，传递的对象可以被改变

责任链模式 在高层模块中声明责任链的顺序，在每种情况中添加判断，如果能执行则执行，不能执行则跳转到下一级执行。链中不会改变消息对象的结构。

## 跨区域PK

### 策略模式VS桥梁模式