一、单例模式：

特点：类必须保证只有一个实例存在。

优点：

（1）减少内存开支。

（2）对于消耗资源较多的对象，可以采用启动时直接产生一个单例对象，然后永久驻留内存的方式，减少系统性能开销。

（3）避免对资源的多重占用，例如写文件操作，利用单例模式可以避免对同一个文件的多重操作。

缺点：

（1）如果持有Context对象容易引发内存泄露，因此传递给单例的Context最好是Application Context。可采用以下方法：

this.context = context.getApplicationContext();

实现方法：将构造函数私有化，通过公用静态方法获取类的实例。

（1）饿汉模式：声明时即完成初始化。

**public** **class** Singleton {

**private** **static** **final** Singleton ***instance*** = **new** Singleton();

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**return** ***instance***;

}

}

（2）懒汉模式：首次使用时完成初始化。为保证线程安全，需给获取实例的方法加上同步锁。

**public** **class** Singleton {

**private** **static** Singleton *instance*;

**private** Singleton() {

}

**public** **static** **synchronized** Singleton getInstance() {

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

**return** *instance*;

}

}

（3）Double CheckLock（DCL）模式（推荐）：既能够保证在需要时初始化，又能够保证线程安全，且初始化后获取实例不进行同步锁。由于构造函数非原子操作，需要给实例加上volatile关键字。

**public** **class** Singleton {

**private** **volatile static** Singleton *instance* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){

**synchronized** (Singleton.**class**) {

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

（4）静态内部类方式（推荐）：初次使用时初始化，final关键字保证实例单一性。

**public** **class** Singleton {

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**return** SingletonHolder.***instance***;

}

**private** **static** **class** SingletonHolder{

**private** **static** **final** Singleton ***instance*** = **new** Singleton();

}

}

（5）枚举方式（最佳）:同时保证单例、线程安全、反序列安全。

**public** **class** Singleton{

**private** Singleton(){

}

**public** **static** Singleton getInstance(){

**return** SingletonEnum.***INSTANCE***.getInstance();

}

**private** **static** **enum** SingletonEnum{

***INSTANCE***;

**private** Singleton instance;

**private** SingletonEnum(){

instance = **new** Singleton();

}

**public** Singleton getInstance(){

**return** instance;

}

}

}

若不需要继承于某个类则可直接将类声明成enum类型，并提供一个INSTANCE。

补充：

（1）除枚举方式外其余几种方式均不能保证反序列安全，如有需要需加上以下方法：

**private** Object readResolve() **throws** ObjectStreamException{

**return** instance;

}

（2）可以创建一个容器类管理各个单例：

**public** **class** SingletonManager{

**private** **static** **final** HashMap<String, Object> ***instanceMap*** = **new** HashMap<>();

**private** SingletonManager(){

}

**public** **static** **void** registerService(String key,Object instance){

**if**(!***instanceMap***.containsKey(key)){

***instanceMap***.put(key, instance);

}

}

**public** **static** Object getInstanceService(String key){

**return** ***instanceMap***.get(key);

}

}

二、Builder模式

特点：提供一系列方法用于描述目标对象的各个特征与属性。

优点：能够精细地控制目标对象，并且对外封装了具体实现，扩展性强。

缺点：需要编写较多额外的代码。

实现方法：无固定模式，一般将Builder类作为静态内部类使用，提供细节描述方法（返回值为调用方法的Builder对象自身）与创建方法（返回值为目标对象）。

**public** **class** Computer {

**private** ComputerParams params;

**protected** Computer(ComputerParams params){

**this**.params = params;

}

@Override

**public** String toString() {

**return** **new** StringBuilder()

.append("CPU:"+params.cpu+"\n")

.append("Monitor:"+params.monitor+"\n")

.append("Disk:"+params.disk)

.toString();

}

**private** **static** **class** ComputerParams{

**private** String cpu = "default";

**private** String monitor = "default";

**private** String disk = "default";

**private** **void** setCpu(String cpu){

**this**.cpu = cpu;

}

**private** **void** setMonitor(String monitor){

**this**.monitor = monitor;

}

**private** **void** setDisk(String disk){

**this**.disk = disk;

}

}

**public** **static** **class** ComputerBuilder{

ComputerParams params;

**public** ComputerBuilder() {

params = **new** ComputerParams();

}

**public** ComputerBuilder setCpu(String cpu){

params.setCpu(cpu);

**return** **this**;

}

**public** ComputerBuilder setMonitor(String monitor){

params.setMonitor(monitor);

**return** **this**;

}

**public** ComputerBuilder setDisk(String disk){

params.setDisk(disk);

**return** **this**;

}

**public** Computer create(){

**return** **new** Computer(params);

}

}

}

三、原型模式

特点：用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

优点：构建大量对象的时候，用拷贝已有对象的方法比一个个new性能要好。

缺点：不会执行构造函数，可能造成一些麻烦。

实现方法：一般通过实现Cloneable接口。

**public** **class** Doc **implements** Cloneable {

**private** String mText;

**private** ArrayList<String> mList;

**public** Doc(String mText) {

**this**.mText = mText;

mList = **new** ArrayList<>();

}

@Override

**protected** Doc clone(){

**try** {

Doc doc = (Doc)**super**.clone();

doc.mText = mText;

doc.mList = new ArrayList<>(mList);

**return** doc;

} **catch** (Exception e) {

e.printStackTrace();

}

**return** **null**;

}

}

补充：

（1）要注意浅拷贝（只复制引用）和深拷贝（重建对象）的问题。若使用浅拷贝，对拷贝对象的操作会影响到原对象。

（2）String对象在内存中是不可变的，修改时会重新分配一块内存，而不是在原来的上面更改。因而即使使用的是浅拷贝，改变了某个对象中String的内容也不会影响到其他对象。在clone时可将String与基本类型作相似处理。

（3）对于需要提供对象的读权限，但又需要限制对象的写权限的情况，提供对象的拷贝对象（保护性拷贝）是一个较好的解决方案。

（4）推荐的clone()实现标准：

The general intent is that, for any object x, the expression:

x.clone() != x

will be true, and that the expression:

x.clone().getClass() == x.getClass()

will be true, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, this is not an absolute requirement.

1、由于x.clone()与x不是同一个对象的引用，因此x.clone() != x；

2、拷贝前后的两个对象必然是同一个类的实例，因此x.clone().getClass() == x.getClass()；

3、需要提前实现equals()方法。由于新对象是原对象的拷贝，因而x.clone().equals(x) == true。

四、工厂模式

特点：定义一个用于创建对象的接口，让其子类决定创建对象的具体类型。

优点：扩展性强，满足开闭原则。

缺点：额外代码量大,需要更多类文件。

实现方法：可以为每个具体的产品类创建一个对应的工厂类（构建过程较复杂），也可以创建一个公用工厂类，通过传入具体产品类的class对象并反射其构造器来获取对应实例（构建过程较简单）。

（1）多工厂模式

**abstract** **class** Factory{

**abstract** Product create();

}

**class** FactoryA **extends** Factory{

Product create() {

**return** **new** ProductA();

}

}

（2）单工厂模式

**class** Factory{

public static <T **extends** Product> T create(Class<T> clz) {

Product product = **null**;

**try** {

product = (Product)Class.*forName*(clz.getName()).newInstance();

} **catch** (Exception e) {

e.printStackTrace();

}

**return** (T)product;

}

}

五、策略模式

特点：将同类型问题的不同具体解决方案封装在不同的类中，并提供一个统一的接口。

优点：将算法实现与算法使用分离开，结构清晰，扩展性强。

缺点：需要较多子类支持。

实现方法：各个算法提供一个统一的接口供客户端使用。

**interface** InterPolator{

ArrayList<Double> getXs(**int** time);

}

**class** LinearPolator **implements** InterPolator{

**private** **double** speed = 1.0;

@Override

**public** ArrayList<Double> getXs(**int** time) {

ArrayList<Double> list = **new** ArrayList<>(time + 1);

**for**(**int** i = 0;i <= time;i++){

list.add(speed \* i);

}

**return** list;

}

}

**class** AcceleratePolator **implements** InterPolator{

**private** **double** acceleration = 2.0;

@Override

**public** ArrayList<Double> getXs(**int** time) {

ArrayList<Double> list = **new** ArrayList<>(time + 1);

**for**(**int** i = 0;i <= time;i++){

list.add(0.5 \* acceleration \* i \* i);

}

**return** list;

}

}

**public** **class** StrategyTest {

**private** InterPolator strategy;

**public** StrategyTest(InterPolator strategy) {

**this**.strategy = strategy;

}

**public** **void** Print(**int** time){

System.***out***.println(strategy.getXs(time));

}

}

六、状态模式

特点：对象在不同的状态下会产生不同的行为。

优点：客户端使用方便，没有switch与if-else等冗余语句，便于维护。

缺点：对于在某种状态下无效的方法也必须显式实现，显著增大代码量。

实现方法：将不同状态下的行为封装到独立的状态类中，并提供统一的接口供使用。

**interface** State{

**void** prevChannel();

**void** nextChannel();

**void** volumnUp();

**void** volumnDown();

}

**class** PowerOnState **implements** State{

@Override

**public** **void** prevChannel() {

System.***out***.println("上一个频道");

}

@Override

**public** **void** nextChannel() {

System.***out***.println("下一个频道");

}

@Override

**public** **void** volumnUp() {

System.***out***.println("增大音量");

}

@Override

**public** **void** volumnDown() {

System.***out***.println("降低音量");

}

}

**class** PowerOffState **implements** State{

@Override

**public** **void** prevChannel() {

}

@Override

**public** **void** nextChannel() {

}

@Override

**public** **void** volumnUp() {

}

@Override

**public** **void** volumnDown() {

}

}

**class** RemoteControl{

**private** State mState;

**public** RemoteControl() {

mState = **new** PowerOffState();

}

**void** powerOn(){

**if**(!PowerOnState.**class**.isInstance(mState)){

mState = **new** PowerOnState();

System.***out***.println("开机");

}

}

**void** powerOff(){

**if**(!PowerOffState.**class**.isInstance(mState)){

mState = **new** PowerOffState();

System.***out***.println("关机");

}

}

**public** **void** prevChannel() {

mState.prevChannel();

}

**public** **void** nextChannel() {

mState.nextChannel();

}

**public** **void** volumnUp() {

mState.volumnUp();

}

**public** **void** volumnDown() {

mState.volumnDown();

}

}

七、责任链模式

特点：多个对象都有机会处理请求。

优点：避免了请求的发送者与接收者之间的耦合关系。

缺点：处理者数量增多会导致递归层次增多，影响性能。

实现方法：将对象连成一条链，并沿着这条链传递请求，直至有对象将其处理为止。

**abstract** **class** AbstractRequest{

**public** AbstractRequest() {

}

**abstract** **int** getRequestLevel();

}

**class** Handler{

**private** **int** level;

**protected** Handler succseeor = **null**;

**public** Handler(**int** level) {

**this**.level = level;

}

**public** **final** **void** handleRequest(AbstractRequest request){

**if**(request.getRequestLevel() == getHandlerLevel()){

handle(request);

}**else** {

**if**(succseeor != **null**){

succseeor.handleRequest(request);

}**else** {

System.***out***.println("请求未处理");

}

}

}

**private** **void** handle(AbstractRequest request){

System.***out***.println("已处理");

}

**public** Handler setNext(**int** level){

succseeor = **new** Handler(level);

**return** succseeor;

}

**int** getHandlerLevel(){

**return** level;

}

}

**class** Request1 **extends** AbstractRequest{

**public** Request1() {

**super**();

}

@Override

**int** getRequestLevel() {

**return** 1;

}

}

**class** Request2 **extends** AbstractRequest{

**public** Request2() {

**super**();

}

@Override

**int** getRequestLevel() {

**return** 2;

}

}

**class** Request3 **extends** AbstractRequest{

**public** Request3() {

**super**();

}

@Override

**int** getRequestLevel() {

**return** 3;

}

}

补充：

（1）若对请求的处理规则不确定，则可将Handler对象中的handle()方法声明成abstract的。对于某个处理者是否可以处理某个请求的判断同理。

八、解释器模式

特点：提供对遵循某种特定文法的语言的解释。

优点：若需增加语言特性，只需提供新的非终结符解释器，扩展性强。

缺点：对于复杂文法的解释较为困难。

实现方法：构建统一的解释器接口，并提供所需的终结符解释器与非终结符解释器。

**abstract** **class** ArithmeticExpression{

**abstract** **int** interpreter();

}

**class** NumExpression **extends** ArithmeticExpression{

**private** **int** num;

**public** NumExpression(**int** num) {

**this**.num = num;

}

@Override

**int** interpreter() {

**return** num;

}

}

**abstract** **class** OperatorExpression **extends** ArithmeticExpression{

**protected** ArithmeticExpression exp1,exp2;

**public** OperatorExpression(ArithmeticExpression exp1,

ArithmeticExpression exp2) {

**this**.exp1 = exp1;

**this**.exp2 = exp2;

}

}

**class** AdditionExpression **extends** OperatorExpression{

**public** AdditionExpression(ArithmeticExpression exp1, ArithmeticExpression exp2) {

**super**(exp1, exp2);

}

@Override

**int** interpreter() {

**return** exp1.interpreter() + exp2.interpreter();

}

}

**class** Calculator{

**private** Stack<ArithmeticExpression> stack = **new** Stack<>();

**public** Calculator(String expression){

String[] elements = expression.split(" ");

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

**switch** (elements[i].charAt(0)) {

**case** '+':

ArithmeticExpression exp1 = stack.pop();

ArithmeticExpression exp2 = **new**

NumExpression(Integer.*valueOf*(elements[++i]));

stack.push(**new** AdditionExpression(exp1, exp2));

**break**;

**default**:

stack.push(**new** NumExpression(Integer.*valueOf*(elements[i])));

**break**;

}

}

}

**public** **int** calc(){

**return** stack.peek().interpreter();

}

}

**public** **class** InterpreterTest {

**public** **static** **void** main(String[] args) {

System.***out***.println(**new** Calculator("1 + 22 + 333").calc());

}

}

补充：

（1）终结符：无法再继续向下推导的符号，对应非终结符。

九、命令模式

特点：用户只需发出命令，无需考虑命令如何执行。

优点：耦合度低，开发方便，便于设计组合命令。

缺点：过于冗余，

实现方法：主要由Receiver（处理具体逻辑）、Command（单个命令）、Invoker（发出命令）组成，客户端仅仅与Invoker交互。

**class** Controller{

**void** toLeft(){

System.***out***.println("左");

}

**void** toRight(){

System.***out***.println("右");

}

**void** toUp(){

System.***out***.println("上");

}

**void** toDown(){

System.***out***.println("下");

}

}

**interface** Command{

**void** execute();

}

**class** LeftCommand **implements** Command{

**private** Controller controller;

**public** LeftCommand(Controller controller) {

**this**.controller = controller;

}

@Override

**public** **void** execute() {

controller.toLeft();

}

}

**class** RightCommand **implements** Command{

**private** Controller controller;

**public** RightCommand(Controller controller){

**this**.controller = controller;

}

@Override

**public** **void** execute() {

controller.toRight();

}

}

**class** UpCommand **implements** Command{

**private** Controller controller;

**public** UpCommand(Controller controller){

**this**.controller = controller;

}

@Override

**public** **void** execute() {

controller.toUp();

}

}

**class** DownCommand **implements** Command{

**private** Controller controller;

**public** DownCommand(Controller controller){

**this**.controller = controller;

}

@Override

**public** **void** execute() {

controller.toDown();

}

}

**class** Invoker{

**private** LeftCommand leftCommand;

**private** RightCommand rightCommand;

**private** UpCommand upCommand;

**private** DownCommand downCommand;

**public** **void** setLeftCommand(LeftCommand leftCommand) {

**this**.leftCommand = leftCommand;

}

**public** **void** setRightCommand(RightCommand rightCommand) {

**this**.rightCommand = rightCommand;

}

**public** **void** setUpCommand(UpCommand upCommand) {

**this**.upCommand = upCommand;

}

**public** **void** setDownCommand(DownCommand downCommand) {

**this**.downCommand = downCommand;

}

**public** **void** toLeft(){

leftCommand.execute();

}

**public** **void** toRight(){

rightCommand.execute();

}

**public** **void** toUp(){

upCommand.execute();

}

**public** **void** toDown(){

downCommand.execute();

}

}

**public** **class** CommandTest {

**public** **static** **void** main(String args[]){

Controller controller = **new** Controller();

Invoker invoker = **new** Invoker();

invoker.setLeftCommand(**new** LeftCommand(controller));

invoker.setRightCommand(**new** RightCommand(controller));

invoker.setUpCommand(**new** UpCommand(controller));

invoker.setDownCommand(**new** DownCommand(controller));

invoker.toLeft();

invoker.toRight();

invoker.toUp();

invoker.toDown();

}

}

十、观察者模式

特点：当被观察者的状态发生改变时，相应观察者会得到通知并自动更新。

优点：观察者与被观察者间耦合度低。

缺点：

实现方法：主要利用Observer接口与Observable类。

**class** Coder **implements** Observer{

@Override

**public** **void** update(Observable o, Object arg) {

System.***out***.println("已收到" + arg);

}

}

**class** WebSite **extends** Observable{

**public** **void** postNewNews(String news){

setChanged();

notifyObservers(news);

}

}

十一、备忘录模式

特点：在对象之外保存对象的内容，并可恢复。

优点：

缺点：

实现方法：主要由备忘录类Memento与备忘录存储类CareTaker构成，主类中提供构建备忘录的方法以及根据备忘录恢复内容的方法。

**class** Game{

**private** **int** mLifePoint;

**private** **int** mLevel;

**private** **int** mType;

Game(**int** lifePoint,**int** level,**int** type){

**this**.mLifePoint = lifePoint;

**this**.mLevel = level;

**this**.mType = type;

}

Game() {

**this**(0, 0, 0);

}

Memento createMemento(){

**return** **new** Memento(mLifePoint, mLevel, mType);

}

**void** restore(Memento memento){

**this**.mLifePoint = memento.mLifePoint;

**this**.mLevel = memento.mLevel;

**this**.mType = memento.mType;

}

@Override

**public** String toString() {

**return** String.*format*("lifepoint: %d\nlevel: %d\ntype: %d\n"

,mLifePoint,mLevel,mType);

}

}

**class** Memento{

**int** mLifePoint;

**int** mLevel;

**int** mType;

Memento(**int** lifePoint,**int** level,**int** type){

**this**.mLifePoint = lifePoint;

**this**.mLevel = level;

**this**.mType = type;

}

}

**class** Caretaker{

**private** Memento memento;

**public** **void** archive(Memento memento){

**this**.memento = memento;

}

**public** Memento getMemento(){

**return** memento;

}

}

十二、迭代器模式

特点：在不暴露容器内部细节的条件下提供遍历容器元素的方法。

优点：弱化了遍历算法与容器的耦合。

缺点：

实现方法：一般通过实现Iterable<T>接口。

十三、模板方法模式

特点：定义了一个算法的框架，而将每步的实现延迟到了子类中。

优点：子类只需重写某些方法即可实现自定义，代码复用率高。

缺点：造成阅读困难。

实现方法：由父类提供一个声明为final的模板方法。

**abstract** **class** GoToSchool{

**abstract** **protected** **void** longJourney();

**abstract** **protected** **void** shortJourney();

**abstract** **protected** **void** walk();

**public** **final** **void** go(){

longJourney();

shortJourney();

walk();

}

}

**class** Plan1 **extends** GoToSchool{

@Override

**protected** **void** longJourney() {

System.***out***.println("Plane");

}

@Override

**protected** **void** shortJourney() {

System.***out***.println("Bus");

}

@Override

**protected** **void** walk() {

System.***out***.println("500m");

}

}

**class** Plan2 **extends** GoToSchool{

@Override

**protected** **void** longJourney() {

System.***out***.println("Train");

}

@Override

**protected** **void** shortJourney() {

System.***out***.println("Taxi");

}

@Override

**protected** **void** walk() {

System.***out***.println("1km");

}

}

十四、访问者模式

特点：不同访问者访问同一对象池时会产生不同的行为。

优点：添加新访问者比较方便。

缺点：对被访问者内部进行修改会影响到访问者，违反了依赖倒置原则。访问者需要知道被访问者的细节，违反了迪米特原则。

实现方法：将被访问者与访问者均声明为抽象类或接口，访问者提供对每种被访问者子类的访问方法。

**abstract** **class** Staff{

**public** String name;

**public** **int** kpi;

**public** Staff(String name){

**this**.name = name;

kpi = **new** Random().nextInt(10);

}

**public** **abstract** **void** accept(Visitor visitor);

}

**interface** Visitor{

**void** visit(Engineer engineer);

**void** visit(Manager manager);

}

**class** Engineer **extends** Staff{

**public** Engineer(String name) {

**super**(name);

}

@Override

**public** **void** accept(Visitor visitor) {

visitor.visit(**this**);

}

**public** **int** getCodeLines(){

**return** **new** Random().nextInt(100000);

}

}

**class** Manager **extends** Staff{

**public** Manager(String name) {

**super**(name);

}

@Override

**public** **void** accept(Visitor visitor) {

visitor.visit(**this**);

}

**public** **int** getProducts(){

**return** **new** Random().nextInt(5);

}

}

**class** CEO **implements** Visitor{

@Override

**public** **void** visit(Engineer engineer) {

System.***out***.println(engineer.name + "工程师：KPI:" + engineer.kpi);

}

@Override

**public** **void** visit(Manager manager) {

System.***out***.println(manager.name + "经理：KPI:" + manager.kpi

+ "新产品：" + manager.getProducts());

}

}

**class** CTO **implements** Visitor{

@Override

**public** **void** visit(Engineer engineer) {

System.***out***.println(engineer.name + "工程师：KPI:" + engineer.kpi

+ "代码量：" + engineer.getCodeLines());

}

@Override

**public** **void** visit(Manager manager) {

System.***out***.println(manager.name + "经理：KPI:" + manager.kpi);

}

}

**class** Reporter{

**private** ArrayList<Staff> staffs = **new** ArrayList<>();

**public** Reporter() {

staffs.add(**new** Engineer("小明"));

staffs.add(**new** Engineer("小华"));

staffs.add(**new** Manager("小李"));

staffs.add(**new** Manager("小刘"));

}

**public** **void** report(Visitor visitor){

**for**(Staff staff : staffs)

staff.accept(visitor);

}

}

十五、中介者模式

特点：将多对多的关系网转变成多对一的关系网。

优点：避免了关系复杂度的膨胀，便于维护。

缺点：随着类的增加，中介者本身可能变得过于复杂。

实现方法：每个具体对象中保存一个中介者对象引用，中介者对象保存所有具体对象引用，具体对象发生变化时通过调用中介者提供的方法与其他对象交互。

十六、代理模式

特点：将对具体对象的调用交给代理对象处理。

优点：保证了对客户端的透明。

缺点：

实现方法：代理类与被代理类实现相同的接口，客户端直接操作接口（实际上是代理类的引用）。

作者：Intopass

链接：https://www.zhihu.com/question/20794107/answer/75164285

来源：知乎

**public** **interface** FontProvider {

Font getFont(String name);

}

**public** **abstract** **class** ProviderFactory {

**public** **static** FontProvider getFontProvider() {

**return** **new** FontProviderFromDisk();

}

}

（1）静态代理：直接指定

**public** **class** CachedFontProvider **implements** FontProvider {

**private** FontProvider fontProvider;

**private** static Map<String, Font> cached;

**public** CachedFontProvider(FontProvider fontProvider) {

**this**.fontProvider = fontProvider;

}

**public** Font getFont(String name) {

Font font = cached.get(name);

**if** (font == **null**) {

font = fontProvider.getFont(name);

cached.put(name, font);

}

**return** font;

}

}

**public** **abstract** **class** ProviderFactory {

**public** **static** FontProvider getFontProvider() {

**return** **new** CachedFontProvider(**new** FontProviderFromDisk());

}

}

（2）动态代理：InvocationHandler接口与Proxy类静态方法*newProxyInstance（）*。

**public** **class** CachedProviderHandler **implements** InvocationHandler {

**private** static Map<String, Object> cached = **new** HashMap<>();

**private** Object target;

**public** CachedProviderHandler(Object target) {

**this**.target = target;

}

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

Type[] types = method.getParameterTypes();

**if** (method.getName().matches("get.+") && (types.length == 1) &&

(types[0] == String.**class**)) {

String key = (String) args[0];

Object value = cached.get(key);

**if** (value == **null**) {

value = method.invoke(target, args);

cached.put(key, value);

}

**return** value;

}

**return** method.invoke(target, args);

}

}

**public** **abstract** **class** ProviderFactory {

**public** **static** FontProvider getFontProvider() {

Class<FontProvider> targetClass = FontProvider.**class**;

**return** (FontProvider) Proxy.newProxyInstance(targetClass.getClassLoader(),

**new** Class[] { targetClass },

**new** CachedProviderHandler(**new** FontProviderFromDisk()));

}

}

十七、组合模式

特点：根据一个树状结构组合对象，并提供统一的方法来管理和访问对象。

优点：能够让高层忽略层次的差异，便于控制。

缺点：见下文。

实现方法：为枝干和叶子定义统一的抽象。

（1）安全组合模式：叶子和枝干分开实现。客户端需调用具体的枝干对象，但可保证类型安全。

**abstract** **class** Component{

**protected** String name;

**public** Component(String name) {

**this**.name = name;

}

**abstract** **void** doSomething();

}

**class** Composite **extends** Component{

**private** List<Component> components = **new** ArrayList<>();

**public** Composite(String name) {

**super**(name);

}

@Override

**void** doSomething() {

System.***out***.println(name);

**if**(components.isEmpty()) **return**;

**for**(Component component : components)

component.doSomething();

}

**public** **void** addChild(Component component){

components.add(component);

}

}

**class** Leaf **extends** Component{

**public** Leaf(String name) {

**super**(name);

}

@Override

**void** doSomething() {

System.***out***.println(name);

}

}

（2）透明组合模式：客户端依赖于抽象，某个节点是枝干还是叶子需自行判断。

**abstract** **class** Component{

**protected** String name;

**public** Component(String name) {

**this**.name = name;

}

**abstract** **void** doSomething();

**abstract** **void** addChild(Component component);

}

**class** Branch **extends** Component{

**private** List<Component> components = **new** ArrayList<>();

**public** Branch(String name) {

**super**(name);

}

@Override

**void** doSomething() {

System.***out***.println(name);

**if**(components.isEmpty()) **return**;

**for**(Component component : components)

component.doSomething();

}

@Override

**void** addChild(Component component) {

components.add(component);

}

}

**class** Leaf **extends** Component{

**public** Leaf(String name) {

**super**(name);

}

@Override

**void** doSomething() {

System.***out***.println(name);

}

@Override

**void** addChild(Component component) {

**throw** **new** UnsupportedOperationException("叶子结点无法添加子结点");

}

}

十八、适配器模式

特点：利用原类或对象产生需要的接口

优点：满足开闭原则，提供了不同类协同工作的一种模式。

缺点：系统中使用过多的适配器会导致结构凌乱，阅读困难。

实现方法：在现有代码的基础之上实现目标接口。

（1）类适配器

**class** PowCalculator{

**int** x,y;

**public** PowCalculator(**int** x,**int** y) {

**this**.x = x;

**this**.y = y;

}

**public** **int** calc(){

**int** res = 1;

**for**(**int** i=0;i<y;i++){

res \*= x;

}

**return** res;

}

}

**interface** Report{

**void** report();

}

**class** PowCalculatorAdapter **extends** PowCalculator **implements** Report{

**public** PowCalculatorAdapter(**int** x, **int** y) {

**super**(x, y);

}

@Override

**public** **void** report() {

System.***out***.println(calc());

}

}

（2）对象适配器

**class** PowCalculator{

**int** x,y;

**public** PowCalculator(**int** x,**int** y) {

**this**.x = x;

**this**.y = y;

}

**public** **int** calc(){

**int** res = 1;

**for**(**int** i=0;i<y;i++){

res \*= x;

}

**return** res;

}

}

**interface** Report{

**void** report();

}

**class** PowCalculatorAdapter **implements** Report{

**private** PowCalculator powCalculator;

**public** PowCalculatorAdapter(PowCalculator powCalculator) {

**this**.powCalculator = powCalculator;

}

@Override

**public** **void** report() {

System.***out***.println(powCalculator.calc());

}

}

十九、装饰器模式

特点：对对象的功能进行增强。

优点：

缺点：

实现方法：与静态代理类似。

**class** Person{

String name;

**public** Person(String name){

**this**.name = name;

}

}

**class** CanWalk{

**private** Person person;

**public** CanWalk(Person person) {

**this**.person = person;

}

**public** **void** walk(){

System.***out***.println(person.name+" is walking");

}

}

二十、享元模式

特点:为可共享的对象创建缓存池，避免创建重复对象。

优点：节约内存，提升性能。

缺点：需要区分内部状态（不变）与外部状态（可变），外部状态由客户端维护。

实现方法：创建包含于外部状态相关的接口，具体类实现该接口，并提供一个带有缓存功能的工厂生产具体对象。

**interface** Ticket{

**enum** Type{

***Top***{

@Override

**public** String toString() {

**return** **new** String("上铺");

}

},

***Middle***{

@Override

**public** String toString() {

**return** **new** String("中铺");

}

},

***Bottom***{

@Override

**public** String toString() {

**return** **new** String("下铺");

}

};

}

**void** showInfos(Type type);

}

**class** ConcreteTicket **implements** Ticket{

**private** String from,to;

**public** ConcreteTicket(String from,String to) {

**this**.from = from;

**this**.to = to;

}

@Override

**public** **void** showInfos(Type type) {

System.***out***.println("从" + from + "到" + to + "的" + type + "车票");

}

}

**class** ConcreteTicketFactory{

**private** **static** HashMap<String, Ticket> *map* = **new** HashMap<>();

**public** **static** Ticket getTicket(String from,String to){

String key = from + '-' + to;

**if**(*map*.containsKey(key)){

**return** *map*.get(key);

}**else** {

Ticket ticket = **new** ConcreteTicket(from, to);

*map*.put(key,ticket);

**return** ticket;

}

}

}

二十一、外观模式

特点：为用户提供高层接口，屏蔽内部细节。

优点：便于使用。

缺点：可能造成外观类方法过多，并且添加新方法时需修改外观类，违反开闭原则。

实现方法：仅外观类提供调用接口。

二十二、桥接模式

特点：相连的两部分都可以独立变化。

优点：易于扩展

缺点：

实现方法：两部分在抽象层相连。

**abstract** **class** Coffee{

CoffeeAdditive additive;

**public** Coffee(CoffeeAdditive additive) {

**this**.additive = additive;

}

**abstract** **void** makeCoffee();

}

**abstract** **class** CoffeeAdditive{

**public** **abstract** String what();

}

**class** LargeCoffee **extends** Coffee{

**public** LargeCoffee(CoffeeAdditive additive) {

**super**(additive);

}

@Override

**void** makeCoffee() {

System.***out***.println("大杯" + additive.what() + "咖啡");

}

}

**class** SmallCoffee **extends** Coffee{

**public** SmallCoffee(CoffeeAdditive additive) {

**super**(additive);

}

@Override

**void** makeCoffee() {

System.***out***.println("小杯" + additive.what() + "咖啡");

}

}

**class** Sugar **extends** CoffeeAdditive{

@Override

**public** String what() {

**return** "焦糖";

}

}

**class** Milk **extends** CoffeeAdditive{

@Override

**public** String what() {

**return** "奶昔";

}

}