### 创建型

### 单例模式：（私有构造函数）

1. 线程同步：（懒汉式：一开始懒得加载实例，如果真正数据用到了那么才加载该实例）

private SingleTon() {  
}  
  
public static SingleTon getInstance() {  
 if (*instance* == null) {  
 synchronized (SingleTon.class) {  
 if (*instance* == null) {  
 return *instance* = new SingleTon();  
 }  
 }  
 }  
 return *instance*;  
}

2. 使用静态内部类实现单例模式：

刚开始不会实例化，在调用的时候会加载静态内部类，其静态内部类会初始化实例，所以是懒汉式，并且是线程安全的

private static class Factory{  
 private static SingleTon *instance* = new SingleTon();  
}  
private SingleTon() {  
}  
public static SingleTon getInstance() {  
 return Factory.*instance*;  
}

3. 饿汉式：一开始就饥饿的去加载数据，不管用不用到，都是实例化的

private static SingleTon *instance* = new SingleTon();  
private SingleTon() {  
}  
public static SingleTon getInstance() {  
 return *instance*;  
}

4. 枚举单例：使用枚举的特性，既是唯一性的，又是线程安全的

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

### 原型模式：

将一个对象作为原型，实现Cloneable接口，实现clone方法，返回object对象：

public class Property implements Cloneable{  
 @Override  
 protected Object clone() {  
 return new Property(); // 或者object其他对象  
 }  
}

### 抽象工厂模式：

public class RealFactory implements AbsFactory{  
 public static AbsFactory getFactory(){  
 return new RealFactory();  
 }  
 @Override  
 public Fruit getApple() {  
 return null;  
 }  
 @Override  
 public Fruit getBanana() {  
 return null;  
 }  
}

返回工厂的生产者，以及返回抽象的实例

### 建造者模式：

Builder模式，实例成员很多，拥有default默认值，可以被外部修改，如果外部不修改，采取默认值，使用Builder静态内部类实现

public static void main() {  
 MyClient myClient = new MyClient.Bulider().count(7).time(300).size(900).build();  
}

public class MyClient {  
 private int count;  
 private int size;  
 private int interval;  
 private int time;  
 private String name;  
 private MyClient() {  
 }  
}

结构型

代理模式

代理模式分2种，分别是静态代理和动态代理，特性：控制被代理对象的访问限制

静态代理

在代码编译的时候就已经确认被代理的类

比如被代理的类实现Person接口的student，那么代理对象就是被确认了，通过代理传入student，对student进行代理，然后实现相同接口，例如：

public interface Person {  
 void printMsg();  
}  
class Student implements Person{  
 @Override  
 public void printMsg() {  
 System.*out*.println("Student");  
 }  
}

class PersonStaticProxy implements Person {  
 private Person proxyObject;  
  
 private boolean shouldInterruptReal = false;  
  
 public PersonStaticProxy(Person proxyObject) {  
 this.proxyObject = proxyObject;  
 }  
  
 public void setShouldInterruptReal(boolean shouldInterruptReal) {  
 this.shouldInterruptReal = shouldInterruptReal;  
 }  
  
 @Override  
 public void printMsg() {  
 printProxy();  
 //控制代理  
 if (shouldInterruptReal) {  
 proxyObject.printMsg();  
 }  
 }  
  
 private void printProxy() {  
 System.*out*.println("PersonStaticProxy");  
 }  
}

代理对象PersonStaticProxy实现与被代理对象Student相同接口，外部在调用接口实现的时候会优先调用代理对象的接口，在其接口内部，代理对象才会真正调用被代理对象的方法

静态代理特点：

代理与被代理对象实现相同接口

代理对象引用被代理对象

类型都已经被确认

控制被代理对象访问限制

动态代理

代理类在程序运行时创建的代理方式被成为动态代理

public interface Fruit {  
 void printCategray();  
}  
class Apple implements Fruit {  
 @Override  
 public void printCategray() {  
 System.*out*.println("apple");  
 }  
}  
class FruitInvokeHandler<T extends Fruit> implements InvocationHandler {  
 private Fruit fruit;  
 public FruitInvokeHandler(Fruit fruit) {  
 this.fruit = fruit;  
 }  
 @Override  
 public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
 return method.invoke(fruit, args);  
 }  
}  
class test {  
 public static void main(String[] args) {  
 Fruit fruit = new Apple();  
 Fruit o = (Fruit) Proxy.*newProxyInstance*(Fruit.class.getClassLoader(), new Class[] {Fruit.class},  
 new FruitInvokeHandler<>(fruit));  
 o.printCategray();  
 }  
}

通过Proxy.newProxyInstance生成动态代理对象InvocationHandler的实现类，在实现类实现接口方法的时候对其调用并扩展

动态代理特点：

代理与被代理对象实现相同接口

代理对象引用/生成被代理对象

实现类型都只有加载时候才会被确认

控制被代理对象访问限制

装饰模式

public interface Person {  
 void printMsg();  
}  
class Student implements Person{  
 @Override  
 public void printMsg() {  
 System.*out*.println("Student");  
 }  
}  
class PersonStaticProxy implements Person{  
 private Person proxyObject;  
 public PersonStaticProxy(Person proxyObject) {  
 this.proxyObject = proxyObject;  
 }  
 @Override  
 public void printMsg() {  
 printProxy();  
 proxyObject.printMsg();  
 }  
   
 private void printProxy(){  
 System.*out*.println("PersonStaticProxy");  
 }  
}

装饰模式是实现相同接口的提前下，引用真正实现，并对其提供一些其他方法操作，不会拦截真正实现对象的调用

特点：

代理与被代理对象实现相同接口

代理对象引用/生成被代理对象

类型都已经被确认

不会控制被代理对象访问限制

装饰模式与静态代理模式的区别：

装饰模式主要是强调对类中代码的拓展，而代理模式则偏向于委托类的访问限制

代理模式与适配器模式的区别：

代理模式是代理原来类型相同的功能/接口，限制调用，没有新增的扩展。而适配器模式是继承扩展原来的类，新增新的接口/功能

适配器模式

在原有的基础上面扩展新方法，有3种适配方式

接口适配

public interface Fruit {  
 void printCategray();  
}  
abstract class NorFruit implements Fruit {  
 abstract void printLocation();  
}

类适配

创建新类，继承源类，并实现新接口

对象适配

创建新类，引用原类，并实现新接口

特点：

都是创建class类

实现新的接口

外观模式

外观模式是向外部提供一个高层接口，外部调用的时候并不知道内部许多具体实现，比方：按下开始键电脑启动，同时每个部分相互独立，只有全部启动完成之后才能启动完成

上面的例子针对结构型是这样，但是具体实现使我想起了ForkJoinTask任务窃取，分布式等待完成结果，或者使用CyclicBarriar同步栅栏来等待多线程执行完成，如果内部系统相对独立设计可以考虑使用外观模式+多线程等待

例如：

public class FacadeSystem {  
 private ForkJoinTask<Boolean> screenPart = new ScreenPart();  
 private ForkJoinTask<Boolean> machPart = new MachPart();  
 private ForkJoinTask<Boolean> keyPart = new KeyBoardPart();  
  
 boolean facadeStartSystem(ForkJoinPool forkJoinPool) {  
 forkJoinPool.submit(screenPart);  
 forkJoinPool.submit(machPart);  
 forkJoinPool.submit(keyPart);  
 return screenPart.join() && machPart.join() && keyPart.join();  
 }  
 public static void main(String[] args) {  
 new FacadeSystem().facadeStartSystem(new ForkJoinPool());  
 }  
}  
class ScreenPart extends RecursiveTask<Boolean> {  
 @Override  
 protected Boolean compute() {  
 // 启动屏幕  
 return true;  
 }  
}  
class MachPart extends RecursiveTask<Boolean> {  
 @Override  
 protected Boolean compute() {  
 // 启动主机  
 return true;  
 }  
}  
class KeyBoardPart extends RecursiveTask<Boolean> {  
 @Override  
 protected Boolean compute() {  
 // 启动键盘  
 return true;  
 }  
}

桥接模式：

如果在设计类/程序中存在多种独立变化的维度，相互不会影响，各个部分独立发展，使用桥接模式

举例子：

比如手机，存在手机品牌，电池容量，内存大小等，他们互不影响，每个属性有自己多个实现，这里就使用桥接模式来配置一部手机

interface Battery {  
 void printBattery();  
}  
interface Brand {  
 void printBrand();  
}  
interface Memory {  
 void printMemory();  
}  
public abstract class Phone {  
 protected Battery battery;  
 protected Brand brand;  
 protected Memory memory;  
   
 public void setBattery(Battery battery) {  
 this.battery = battery;  
 }  
 public void setBrand(Brand brand) {  
 this.brand = brand;  
 }  
 public void setMemory(Memory memory) {  
 this.memory = memory;  
 }  
 public Phone(Battery battery, Brand brand, Memory memory) {  
 setBattery(battery);  
 setBrand(brand);  
 setMemory(memory);  
 }  
  
 abstract void printMessage();  
}

定义抽象类，配置多个接口实现，下面将产生一个实现类，比如华为手机

class Huawei extends Phone {  
 public Huawei(Battery battery, Brand brand, Memory memory) {  
 super(battery,brand,memory);  
 }  
 @Override  
 void printMessage() {  
 battery.printBattery();  
 brand.printBrand();  
 memory.printMemory();  
 }  
}

特点：

各部分独立实现，定义接口

抽象基类满足单一性，动态配置实现

桥接模式，连接了各个接口，抽象类单一，动态配置具体实现类以及方便修改与替换

与外观模式的区别：

外观模式是预先定义好的，是一个顶层接口，内部实现不可以由外部动态修改，外部无法知道其内部具体实现，所以是偏向顶层的

桥接模式是中层的/底层的，是一个可以动态替换配置每个部件的具体实现，而且需要定义一个抽象类引用每个接口，提供set方法，具体实现都需要继承这个抽象类，这个抽象类提供其他抽象方法

组合模式：

这个模式只适用于整体和部分可能格式是一样的，比如文件夹系统/树形结构

享元模式

这个模式是一种内存缓存机制，举例如下：

public class SharedMemory {  
 private static Map<String, Object> *sharedMemorys* = new HashMap<>();  
  
 public static Object get(String key) {  
 Object object;  
 if (*sharedMemorys*.containsKey(key)) {  
 object = *sharedMemorys*.get(key);  
 } else {  
 object = *sharedMemorys*.put(key, new Object());  
 }  
 return object;  
 }  
}

模板方法模式：

定义一系列操作的算法骨架，将一些算法的具体实现交给子类实现

1. 定义抽象类

2. 提供部分实现，提供部分抽象接口

3. 没有新接口的实现，没有内部其他接口

与桥接模式的区别：

1. 桥接模式是一系列接口相互独立，并且提供给外部设置/变更具体实现，并且定义抽象类实现新的接口，类是抽象的定义

2. 模板方法是一个抽象类提供部分实现与部分待实现，所有子类都是继承其并实现具体方法，类也是抽象的定义

行为型：

（行为型模式相比于结构型模式，行为型更加注重功能的实现与替换，结构型注重的类型的设计，所以结构型与行为型不会冲突）

策略模式

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

2. 策略模式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

3. 恰当时候可以被子类覆盖重写修改

例如：

interface IBaseMaker{  
 void Cooking();  
}  
class WaterCooker implements IBaseMaker{  
 @Override  
 public void Cooking() {  
 System.*out*.println("水煮");  
 }  
}  
class WaterCookerV2 implements IBaseMaker{  
 @Override  
 public void Cooking() {  
 System.*out*.println("清蒸");  
 }  
}  
public class Strategy {  
 private IBaseMaker iBaseMaker;  
  
 public void setiBaseMaker(IBaseMaker iBaseMaker) {  
 this.iBaseMaker = iBaseMaker;  
 }  
 public void makeCooking(){  
 if(iBaseMaker != null) {  
 iBaseMaker.Cooking();  
 }  
 }  
}

这是一种行为型的设计模式，与结构型对比

结构型倾向于类整体设计，行为型倾向于类的功能实现

策略模式与桥接模式比较：

1. 桥接模式是设计一个类的时候考虑其每个属性/模块相互独立，没有交织并且可以有不同实现，桥接定义一个抽象类，使用不同的抽象接口实现，子类具体实现，而策略模式，类型已经确定，整体框架也已经确定，但是每个细节可能有不同实现，提供给子类进行替换实现

2. 与模板方法相比，模板也是整体框架确定，但是模板提供抽象方法，子类实现，策略模式是替换接口，并不是子类实现抽象方法

命令模式：

命令模式包含以下主要角色：

抽象命令类（Command）角色：声明执行命令的接口，拥有执行命令的抽象方法 execute()。

具体命令角色（Concrete    Command）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。

实现者/接收者（Receiver）角色：执行命令功能的相关操作，是具体命令对象业务的真正实现者。

调用者/请求者（Invoker）角色：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。

// 命令  
public interface Command {  
 void excute();  
}  
// 接收者  
class Receiver{  
 public void getResult(){  
  
 }  
}  
// 命令实现类  
class CommandV1 implements Command{  
 private Receiver receiver;  
 public CommandV1() {  
 this.receiver = new Receiver();  
 }  
 @Override  
 public void excute() {  
 // 通知接收者  
 receiver.getResult();  
 }  
}  
// 调用者，命令发送类  
class Invoker{  
 Command command;  
 public void setCommand(Command command) {  
 this.command = command;  
 }  
 public void invoke(){  
 command.excute();  
 }  
 public static void main(String[] args){  
 Invoker invoker = new Invoker();  
 invoker.setCommand(new CommandV1());  
 invoker.invoke();  
 }  
}

责任链模式：

1. 定义接口链提供处理方法

2. 定义抽象类实现当前接口并提供添加节点方法，形成链式结构

3. 实现2抽象类，并创建添加不同节点

例如：

//链中处理方法  
public interface Chain {  
 void handleMessage(String msg);  
}  
//链节点定义，抽象类  
abstract class AbstractChain implements Chain {  
 private Chain nextChain;  
  
 public void setNextChain(Chain nextChain) {  
 this.nextChain = nextChain;  
 }  
 // 当前处理之后，提供调用下一个节点进行处理方法  
 public void handleNext(String msg) {  
 if (null != nextChain) {  
 nextChain.handleMessage(msg);  
 }  
 }  
}  
//具体实现类，可能有很多  
class ChainImpl extends AbstractChain {  
 private String nodeName;  
 public ChainImpl(String nodeName) {  
 this.nodeName = nodeName;  
 }  
  
 @Override  
 public void handleMessage(String msg) {  
 System.*out*.println("nodeName is:" + nodeName);  
 handleNext(msg);  
 }  
  
 public static void main(String[] args) {  
 // 添加形成链式节点  
 AbstractChain abstractChain = new ChainImpl("001");  
 abstractChain.setNextChain(new ChainImpl("002"));  
 // 发送消息，交给链式处理  
 abstractChain.handleMessage("this is message");  
 }  
}

状态模式：

状态模式把受环境改变的对象行为包装在不同的状态对象里，其意图是让一个对象在其内部状态改变的时候，其行为也随之改变。现在我们来分析其基本结构和实现方法

主要意图就是在调用者调用方法的时候，内部状态也会随之变化

public interface State {  
 void handleState(Context context);  
}  
class StateA implements State{  
 @Override  
 public void handleState(Context context) {  
 // 状态A变更为状态B  
 context.setState(new StateB());  
 }  
}  
class StateB implements State{  
  
 @Override  
 public void handleState(Context context) {  
 // 状态B变更为状态A  
 context.setState(new StateA());  
 }  
}  
class Context{  
 private State state;  
  
 public void setState(State state) {  
 this.state = state;  
 }  
 public void externalHandle(){  
 System.*out*.println("处理事务");  
 // 内部状态随之变更  
 state.handleState(this);  
 }  
 public static void main(String[] args){  
 Context context = new Context();  
 context.setState(new StateA());  
 context.externalHandle();  
 }  
}

观察者模式：

观察者模式是一种耳熟能详的模式，其特点如下：

1. 定义观察者，定义并持有一种被观察接口

2. 观察者使用集合保存被观察接口，并提供添加与移除被观察者方法

3. 定义被观察者类，实现被观察接口，用于1通知本身改变

举例：

interface IBaseObserver<T> {  
 void onChange(T result);  
}  
class ObserverImplA implements IBaseObserver<String>{  
 @Override  
 public void onChange(String result) {  
 System.*out*.println("A has receiver changed");  
 }  
}  
class ObserverImplB implements IBaseObserver<String>{  
 @Override  
 public void onChange(String result) {  
 System.*out*.println("B has receiver changed");  
 }  
}  
  
public class ObserverManager<T> {  
 private Collection<IBaseObserver<T>> sets = new HashSet<>();  
  
 public void addObserver(IBaseObserver<T> item) {  
 sets.add(item);  
 }  
  
 public void notifyChange(T result) {  
 for (IBaseObserver<T> item : sets) {  
 item.onChange(result);  
 }  
 }  
}

中介者模式：

定义一个中介，去寻找不同的对象进行处理，自身也是属于中介持有的一部分，中介持有的抽象都是不同的实现，每个实现可以发送请求给中介，让中介寻找相同抽象但是不同实现去处理请求

比如，有一个木匠，裁缝，车夫，厨师等多个角色，木匠想要找裁缝去缝衣服，但是木匠不认识裁缝，这里就需要一个中介去帮他缝衣服，中介找到裁缝去缝衣服，裁缝找厨师也要找中介，中介找到厨师去处理

代码定义如下：

// 中介抽象接口  
interface Supporter {  
 void addIOperator(IOperator operator);  
  
 void relay(String msg, IOperator iOperator);  
}  
// 各部分抽象接口  
interface IOperator {  
 void doSomething(String args);  
 int getType();  
}  
// 具体中介实现类，保存各部分集合  
public class ThirdSupporter implements Supporter {  
 Collection<IOperator> sets = new HashSet<>();  
 public void addIOperator(IOperator operator) {  
 sets.add(operator);  
 }  
 @Override  
 public void relay(String msg, IOperator iOperator) {  
 for (IOperator operator : sets) {  
 if (iOperator.getType() == operator.getType()) {  
 operator.doSomething(msg);  
 }  
 }  
 }  
}  
// 各部分实现类不同，引用中介  
class Person implements IOperator{  
 protected final Supporter supporter;  
 public Person(Supporter supporter) {  
 this.supporter = supporter;  
 supporter.addIOperator(this);  
 }  
 @Override  
 public void doSomething(String args) {  
 // 交给中介处理  
 supporter.relay(args,this);  
 }  
 @Override  
 public int getType() {  
 return 0;  
 }  
}

中介者模式 由一个中介具体实现类，多个个体注册给中介，最后个体调用中介方法，交给中介去寻找对应相同接口个体去实现

与命令模式的区别：

命令模式偏向于策略模式，命令模式由一个调用者实现类，调用者设置一个接口指令，指令由多个实现，最后指令实现又交由实现体里面的接收者去调用，偏向一个长官，多个指令实现，每个指令内部对应唯一接收者

与策略模式却别：

策略模式是接口的多个实现，但是最终持有者只会有一个实现，比如指令，可以使用策略，最终有一个策略被调用者（长官）去调用，指令实现内部最终又交给内部接收者去实现

迭代器模式：

一般实现Iterator接口表示当前类可迭代，实现hasNext方法和Next方法，可以用作内部持有列表，数据，或者链表的foreach语法

或者实现Iterable接口，提供不同的Iterator迭代器用于迭代

迭代器源码实现如下：

for(Iterator iterator = iteratable.iterator() ; iterator.hasNext() ; ){  
 Object object = iterator.next();  
 // Object可以是泛型指定  
 // doSomething  
}

接口实现如下：

class Iteratable implements Iterable {  
 private Iterator iterator = new DefaultIterator();  
 // 策略模式，可以设置不同迭代接口实现类  
 public void setIterator(Iterator iterator) {  
 this.iterator = iterator;  
 }  
 @NonNull  
 @Override  
 public Iterator iterator() {  
 return iterator;  
 }  
 class DefaultIterator implements Iterator {  
 @Override  
 public boolean hasNext() {  
 return false;  
 }  
 @Override  
 public Object next() {  
 return null;  
 }  
 }  
}

访问者模式：

访问者模式用于一个类对其部分数据的访问，区别于该类提供public方法给外部访问的情况，这总模式可以由该类控制访问者的方法的数据，定义接口

比如：

interface Visit {  
 int getPermission();  
  
 void onMsgReplay(String msg);  
}  
abstract class VisitAdapter implements Visit{  
 @Override  
 public void onMsgReplay(String msg) {  
 System.*out*.println(msg);  
 }  
}  
interface IVisit {  
 void visit(Visit visit);  
}  
public class Visitable implements IVisit {  
 // 最高消息  
 private String msg1;  
 // 一般消息  
 private String msg2;  
 // 通用消息  
 private String msg3;  
  
 public void visit(Visit visit) {  
 switch (visit.getPermission()) {  
 case 0:  
 visit.onMsgReplay(msg1);  
 break;  
 case 1:  
 visit.onMsgReplay(msg2);  
 break;  
 default:  
 visit.onMsgReplay(msg3);  
 break;  
 }  
 }  
 public static void main(String[] args){  
 IVisit visit = new Visitable();  
 visit.visit(new VisitAdapter() {  
 @Override  
 public int getPermission() {  
 return 0;  
 }  
 });  
 visit.visit(new VisitAdapter() {  
 @Override  
 public int getPermission() {  
 return 5;  
 }  
 });  
 }  
}

相比访问者模式，个人觉得如果是同步状态下，直接提供visit方法switch并返回数据可能会更好一点，少去了接口的定义

如果是异步状态下，就是一种callback，也不能算是访问者模式

不推荐

备忘录模式：

备忘录是一种将数据临时保存在一个内存的模式，本身可以使用clone/新定义接口返回新的对象，并将当前对象保存到缓存类里面

比较简单，不介绍

解释器模式：

不推荐，不介绍