六、设计模式

1、单例模式

特点：

* 单例类只能有一个实例。
* 单例类必须自己创建自己的唯一实例，其构造函数是私有的。
* 单例类必须给所有其他对象提供这一实例。

实现：

懒汉式[单线程]【不好】

缺点：只在单线程的情况下正常运行，在多线程的情况下，就会出问题。

1. //懒汉式单例类.在第一次调用的时候实例化自己
2. **public** **class** Singleton {
3. **private** Singleton() {}
4. **private** **static** Singleton single=**null**;
5. //静态工厂方法
6. **public** **static** Singleton getInstance() {
7. **if** (single == **null**) {
8. single = **new** Singleton();
9. }
10. **return** single;
11. }
12. }

懒汉式[多线程]【不好】：在getInstance方法上加同步

缺点：每次通过getInstance方法得到singleton实例的时候都有一个试图去获取同步锁的过程。而加锁是很耗时的。能避免则避免。

1. //在getInstance方法上加同步
2. **public** **static** **synchronized** Singleton getInstance() {
3. **if** (single == **null**) {
4. single = **new** Singleton();
5. }
6. **return** single;
7. }

懒汉式[多线程]【可行】：在方法内部加同步锁，双重检查锁定，只有当instance为null时，需要获取同步锁，创建一次实例。当实例已经被创建时，则无需试图加锁。

缺点：用双重if判断，复杂，容易出错。

1. //在方法内部加同步锁，双重检查锁定
2. **public** **static** Singleton getInstance() {
3. **if** (singleton == null) {
4. synchronized (Singleton.**class**) {
5. **if** (singleton == null) {
6. singleton = **new** Singleton();
7. }
8. }
9. }
10. **return** singleton;
11. }

饿汉式【建议使用】：静态常量single只在类初始化时被创建一次，以后只需要通过这个类的静态方法获取这个对象即可。

缺点：没有实现延迟加载，无论这个对象是否被用到，对象都会被创建，降低了内存的使用率。

1. //饿汉式单例类.在类初始化时，已经自行实例化
2. **public** **class** Singleton {
3. **private** Singleton() {}
4. **private** **static** **final** Singleton single = **new** Singleton ();
5. //静态工厂方法
6. **public** **static** Singleton getInstance() {
7. **return** single;
8. }
9. }

静态内部类【建议使用】：定义了一个私有的内部类，如果不调用外部类中的静态方法时，内部类不会被加载，单例 也不会被创建，只有在当外部类中的静态方法被调用时，才会加载内部类从而创建一个实例对象，在单例对象占用资源大，需要延时加载的情况下这种方法是首选。

1. **public** **class** Singleton {
2. **private** Singleton(){ }
3. **private** **static** **class** SingletonHolder{
4. **private** **final** **static** Singleton instance=**new** Singleton();
5. }
6. **public** **static** Singleton getInstance(){
7. **return** SingletonHolder.instance;
8. }
9. }

2、工厂模式

简单工厂模式：



运算类及其子类的定义：

1. **public** **abstract** **class** Operation {
2. **public** **double** numberA;
3. **public** **double** numberB;
4. **public** **abstract** **double** getResult();
5. }
6. **class** OperationAdd **extends** Operation {
7. @Override
8. **public** **double** getResult() {
9. **return** numberA + numberB;
10. }
11. }
12. **class** OperationSub **extends** Operation {
13. @Override
14. **public** **double** getResult() {
15. **return** numberA - numberB;
16. }
17. }
18. **class** OperationMul **extends** Operation {
19. @Override
20. **public** **double** getResult() {
21. **return** numberA \* numberB;
22. }
23. }
24. **class** OperationDiv **extends** Operation {
25. @Override
26. **public** **double** getResult() {
27. **return** numberA / numberB;
28. }
29. }

工厂类的定义：

1. **public** **class** OperationFactiory {
2. **public** **static** Operation createOperate(String operate) {
3. Operation oper = **null**;
4. **switch** (operate) {
5. **case** "+":
6. oper = **new** OperationAdd();
7. **break**;
8. **case** "-":
9. oper = **new** OperationSub();
10. **break**;
11. **case** "\*":
12. oper = **new** OperationMul();
13. **break**;
14. **case** "/":
15. oper = **new** OperationDiv();
16. **break**;
17. }
18. **return** oper;
19. }
20. }

客户端的调用方式

1. Operation oper = OperationFactiory.createOperate("+");
2. oper.numberA = 1;
3. oper.numberB = 2;
4. **double** result = oper.getResult();

工厂方法模式：



工厂接口的创建：

1. **public** **interface** IFactory {
2. Operation createOperation();
3. }

每一种运算实现一个工厂：

1. **class** AddFactory **implements** IFactory {
2. @Override
3. **public** Operation createOperation() {
4. **return** **new** OperationAdd();
5. }
6. }
7. **class** SubFactory **implements** IFactory {
8. @Override
9. **public** Operation createOperation() {
10. **return** **new** OperationSub();
11. }
12. }
13. **class** MulFactory **implements** IFactory {
14. @Override
15. **public** Operation createOperation() {
16. **return** **new** OperationMul();
17. }
18. }
19. **class** DivFactory **implements** IFactory {
20. @Override
21. **public** Operation createOperation() {
22. **return** **new** OperationDiv();
23. }
24. }

客户端的调用方式

1. IFactory operFactory = new AddFactory();
2. Operation oper = operFactory.createOperate();
3. oper.numberA = 1;
4. oper.numberB = 2;
5. **double** result = oper.getResult();

简单工厂模式：

优点：工厂类中包含了必要的逻辑判断，可以根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。

缺点：

* 工厂类集中了所有实例（产品）的创建逻辑，一旦该工厂不能正常工作，整个系统都会受到影响；
* 违背“开放 - 关闭原则”，一旦添加新产品就不得不修改工厂类的逻辑，这样会造成工厂逻辑过于复杂。
* 简单工厂模式由于使用了静态工厂方法，静态方法不能被继承和重写，会造成工厂角色无法形成基于继承的等级结构。

工厂方法模式：把普通工厂中具体的工厂类，划分成两层：抽象工厂层+具体的工厂子类层。通过定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象。工厂方法模式只考虑生产同等级的产品

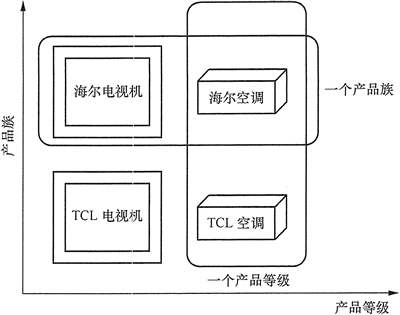
优点：

* 更符合开-闭原则，新增一种产品时，只需要增加相应的具体产品类和相应的工厂子类即可；
* 不使用静态工厂方法，可以形成基于继承的等级结构。
* 工厂模式可以说是简单工厂模式的进一步抽象和拓展，在保留了简单工厂的封装优点的同时，让扩展变得简单，让继承变得可行，增加了多态性的体现。

缺点：

* 添加新产品时，除了增加新产品类外，还要提供与之对应的具体工厂类，系统类的个数将成对增加，在一定程度上增加了系统的复杂度；同时，有更多的类需要编译和运行，会给系统带来一些额外的开销；
* 一个具体工厂只能创建一种具体产品；

抽象工厂模式：



为访问类提供一个创建一系列相互依赖对象的接口，而且访问类时无需指定产品的具体类就能得到同族的不同等级的产品。抽象工厂是工厂方法模式的升级，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

抽象工厂里面包含创建所有产品的抽象方法。抽象工厂里的创建产品的方法返回的都是最高级的抽象产品。

当系统的产品有多个产品族，每个产品族下又有多个产品。因此需要从抽象工厂继承多个具体的工厂，每个具体工厂负责一个产品族下产品的生产。有几个产品族就有几个具体工厂。

抽象工厂模式适合针对2个维度描述的产品的构造问题。取其中一个维度作为产品族（也就是对应一个具体工厂），另外一个维度作为产品族下的具体产品。从这个角度说，抽象工厂模式是在工厂模式基础上，做了一个维度的升级。



使用抽象工厂模式一般要满足以下条件。

* 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。
* 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：

易于交换产品系列，因为具体工厂只需要在创建的时候出现一次，所以改变应用的具体工厂非常容易，只需要改变具体工厂就可以实现产品系列的改变。

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。

适配器模式

定义：将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

动机：通常，客户端(client)通过目标类(target)的接口来使用它的服务。但是，在某些时候，有一些类的功能可以满足客户端的要求，但是它所提供的接口不是客户端所期望的，所以就需要一个适配器把源接口和目标接口衔接起来。然后，客户端通过调用适配器中的方法间接的使用源接口。

组件：

目标(Target)角色：客户端所期待得到的接口，

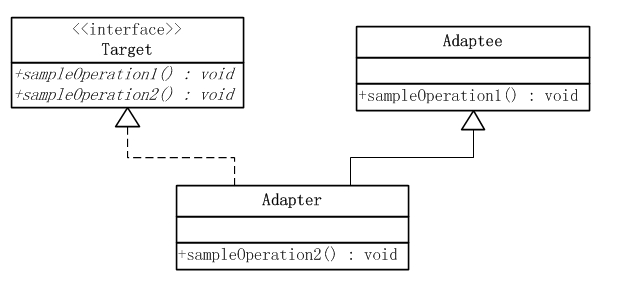
适配器(Adaper)角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

源(Adapee)角色：被适配者，现在需要被适配的接口。

Key：使用适配器模式是一种无奈之举，在软件设计之初一般不用适配器模式，但是在软件的后期维护阶段，由于介入了不同的产品或者厂家，而双方都不太容易修改自己原有的软件架构，这时就需要用到适配器模式。

结构：分为类适配器模式和对象适配器模式

类适配器模式：把适配的类的API转换成为目标类的API。



在上图中可以看出，Adaptee类并没有sampleOperation2()方法，而客户端则期待这个方法。为使客户端能够使用Adaptee类，提供一个中间环节，即类Adapter，把Adaptee的API与Target类的API衔接起来。Adapter与Adaptee是继承关系，这决定了这个适配器模式是类的：

target目标接口

1. **public** **interface** Target {
2. /\*\*
3. \* 这是源类Adaptee也有的方法
4. \*/
5. **public** **void** sampleOperation1();
6. /\*\*
7. \* 这是源类Adapteee没有的方法
8. \*/
9. **public** **void** sampleOperation2();
10. }

源接口

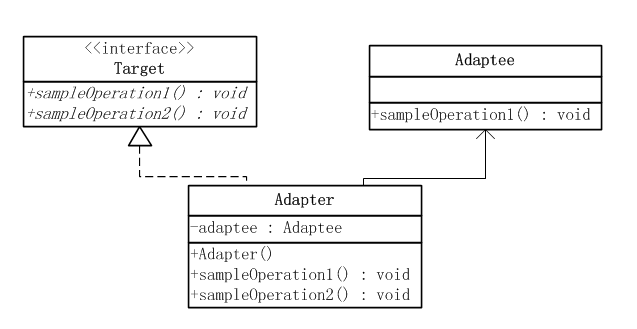
1. **public** **class** Adaptee {
2. **public** **void** sampleOperation1(){}
3. }

适配器角色Adapter扩展了Adaptee,同时又实现了目标(Target)接口。由于Adaptee没有提供sampleOperation2()方法，而目标接口又要求这个方法，因此适配器角色Adapter实现了这个方法。

1. **public** **class** Adapter **extends** Adaptee **implements** Target {
2. /\*\*
3. \* 由于源类Adaptee没有方法sampleOperation2()
4. \* 因此适配器补充上这个方法
5. \*/
6. @Override
7. **public** **void** sampleOperation2() {
8. //写相关的代码
9. }
10. }

对象适配器模式

与类的适配器模式一样，对象的适配器模式把被适配的类的API转换成为目标类的API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到Adaptee类，而是使用关联关系连接到Adaptee类。



adapter适配器的实现如下

1. **public** **class** Adapter {
2. **private** Adaptee adaptee;
3. **public** Adapter(Adaptee adaptee){
4. **this**.adaptee = adaptee;
5. }
6. /\*\*
7. \* 源类Adaptee有方法sampleOperation1
8. \* 因此适配器类直接委派即可
9. \*/
10. **public** **void** sampleOperation1(){
11. **this**.adaptee.sampleOperation1();
12. }
13. /\*\*
14. \* 源类Adaptee没有方法sampleOperation2
15. \* 因此由适配器类需要补充此方法
16. \*/
17. **public** **void** sampleOperation2(){
18. //写相关的代码
19. }
20. }

观察者模式

定义：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。

Key：在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。观察者模式用来解决当一个对象行为发生改变时，这个消息如何传递给其它对象。

实现：实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。

结构：观察者模式的主要角色如下

* 抽象主题（Subject）角色：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。
* 具体主题（Concrete Subject）角色：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。
* 抽象观察者（Observer）角色：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。
* 具体观察者（Concrete Observer）角色：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。



使用场景：当对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象需要被改变。

责任链模式

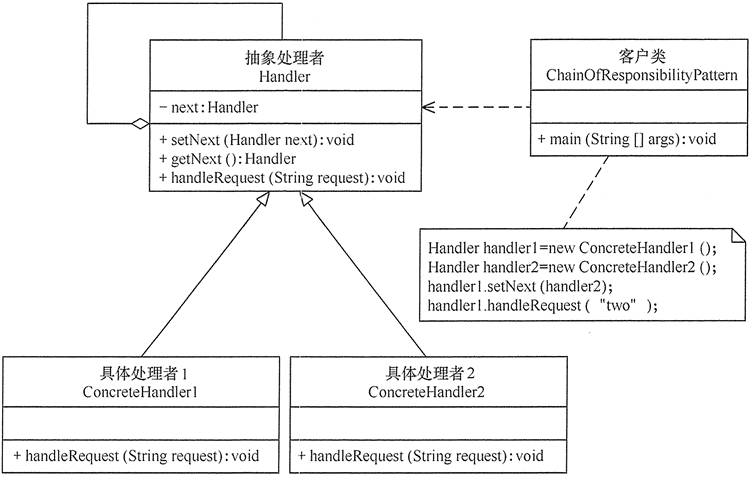
定义：为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

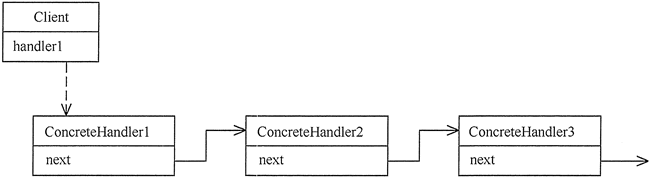
Key：在现实生活中，常常会出现这样的事例：一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。例如，学生请假这个事件，如果天数在1-2天之内，那么班主任就可以批准了；但是如果在3-10天之内，班主任就无法处理，就需要年级主任批准；如果请假的天数超过10天，班主任和年级主任都无权批准，就需要教导主任签字批准了。所以，学生必须根据自己的请假天数去找不同的领导签字批准，这样，他就必须要知道所有领导的姓名、电话和地址等信息，这增加了一定的难度。

特点：责任链模式是一种对象行为型模式，在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，所以责任链将请求的发送者和请求的处理者解耦了。

结构：职责链模式主要包含以下角色。

* 抽象处理者（Handler）角色：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。
* 具体处理者（Concrete Handler）角色：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。
* 客户类（Client）角色：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。





优点：

* 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。
* 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。
* 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。
* 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：

* 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。
* 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。
* 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。

应用场景：

* 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定。
* 在不明确指定请求处理者的情况下，向多个处理者中的一个提交请求。