# 第一讲 设计模式

## 1什么是设计模式

设计模式是无数码农在实际的生产项目中经过不断的踩坑, 爬坑, 填坑之后的经历总结出来的经验教训, 经过系统性的总结所提出的针对某一类问题的最佳解决方案. 能够帮助后来的开发者重复同样的错误或者弯路.

设计模式的熟练运用,帮助开发者将变化的部分和不变化的部分分开, 写出更高质量, 更健壮的代码.

## 2设计模式的六大原则:

开闭原则:

模块应尽量在不修改原代码(闭)的情况下进行扩展(开)。即在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。

里氏替换原则:

如果调用的是父类的话，那么换成子类也完全可以运行。 派生类能够在基类的基础上增加新的行为。只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

依赖导致原则:

1. 高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。

2、抽象不应该依赖于具体实现，具体实现应该依赖于抽象。要求对抽象/接口进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

接口隔离原则:

每一个接口应该是一种角色，不干不该干的事，该干的事都要干。降低类之间的耦合度。

合成复用原则:

要尽量使用组合/ 聚合，尽量不要使用继承。只有“Is - A” 关系才符合继承关系，“Has- A” 关系应当使用聚合来描述。

迪米特原则:

一个对象应对其它对象有尽可能少的了解。即一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

# 第二讲 单例模式

## 什么是单例

* 核心作用:

保证一个类只有一个实例,并且提供一个访问该实例的全局访问点

* 应用场景:

对象的创建需要占用系统资源, 在某些场景下, 比如任务管理器, 线程池, 缓存, 日志对象,数据连接池等

* 优点:

由于单例模式只生成一个实例,减少了系统的性能开销.

## 如何实现单例

* 将类的构造器私有化
* 在本类中创建唯一实例
* 提供一个全局静态的访问点

## 1饿汉式单例

* 特点:

当类被类加载器加载的时候,就创建了类的唯一实例

通过classloade的机制保证了.线程安全,调用效率高,但是不能延时加载

|  |
| --- |
| /\*\*  \*  \* @author marswing  \*饿汉式单例  \*类一加载就实例化对象  \*优点:避免了多线程同步问题  \*缺点:如果对象没有使用,造成内存浪费  \*/  public class Singleton {  private static Singleton instance = new Singleton();  private Singleton(){}  public static Singleton getInstance(){  return instance;  }  } |

类一加载就初始化,避免了多线程的同步问题

## 2懒汉式单例

* 特点:

线程不安全,但是可以延时加载

可以通过synchronized保证线程安全,但是效率比较低

因为百分之九十的情况不需要同步

|  |
| --- |
| /\*\*  \*  \* @author marswing  \*懒汉式单例  \*当需要的时候,才去创建对象  \*/  public class Singleton {  private static Singleton instance = null;  private Singleton(){}  public static Singleton getInstance(){  if(instance == null){  instance = new Singleton();  }  return instance;  }  } |

## 3静态内部类(推荐使用)

* 特点:

线程安全

高效调用

延迟加载

|  |
| --- |
| /\*\*  \*  \* @author marswing  \*静态内部类的方式实现单例  \*线程安全  \*实现了懒加载  \*/  public class Singleton {  private static class SingletonHolder{  private static final Singleton instance =new Singleton();  }  private Singleton(){}  public static final Singleton getInstance(){  return SingletonHolder.instance;  }  } |

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，这种方式是Singleton类被装载了，instance没有被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式就显得很合理。

# 第三讲 工厂模式

工厂模式目的是让对象的创建更加的灵活和易于维护.实现了创建者和调用者的分离

原则是:客户类只使用服务,而不承担维护的责任

## 1 什么时候需要工厂模式?

1.创建对象的过程比较复杂, 需要其他类的辅助, 需要大量计算,需要大量配置信息来获取等(比如connection).需要工厂模式来支持

2.当对象的实例有可能发生改变的时候

## 2 工厂模式的分类:

* 简单工厂:

用来创建同一等级(同一个接口的不同实现类)结构中的任意产品

弊端:对于增加新的产品,需要修改工厂中的逻辑

* 工厂方法:

弥补了简单工厂模式的缺点(不完全满足开闭原则)

工厂方法和简单工厂最大的不同在于,简单工厂只有一个工厂类,而工厂方法有一组实现了相同接口的工厂类

抽象工厂

* 抽象工厂:

特点:用来生产不同产品族的全部产品(对于增加新的产品,无能为力;支持增加产品族)

产品族:一系列有特定关系的产品.比如汽车工厂的发动机,轮胎和座椅

## 简单工厂和工厂方法的比较:

* 结构复杂度:

简单工厂在结构复杂度上比较有优势,只需要一个工厂类就可以,而工厂方法随着产品的增加,工厂也会增加,类结构将会越来越复杂

* 代码复杂度:

代码复杂度和结构复杂度是互斥的,简单工厂正在结构方面相对简洁,但是在代码复杂度上比工厂模式复杂.简单工厂随着产品增多,工厂类的逻辑也越来越臃肿,而工厂方法中每个工厂只负责实例化一个产品,职责单一,代码简洁

* 管理上的难度:

从扩展性上来说,工厂方法完全满足OCP(开闭原则),即具有良好的扩展性,那是否就说明了简单工厂没有扩展性呢,答案是否定的,简单工厂同样也具有良好的扩展性,只不过扩展的时候需要修改工厂类少量的代码,尽管没有完全满足OCP,但是如果能够在合理范围内,做一点妥协,牺牲一点原则,也是可以的,具体采用哪种设计模式,需要看对应的业务场景

工厂模式的要点总结:

简单工厂(静态工厂):

某种程度来说,虽然不完全符合设计原则,但是项目中使用比较多

工厂方法:

符合设计开闭原则,能在不修改已有类的前提下,通过增加新的工厂来实现扩展

抽象工厂(了解):

不可以增加产品,可以增加产品族(比如新增一条中配汽车的生产线:需要中端发动机,中端座椅和轮胎);

# 第四讲 代理模式

## 1 什么是代理模式

## 2 代理模式的应用场景

如果需要在已有代码的基础上扩展功能,比如日志记录,性能统计,事务管理,直接在现有代码中添加代码, 会导致职责不够单一, 代码复用性不好,不方便后期维护,此时需要考虑采用代理模式

## jdk静态代理

* 核心作用:

通过代理,控制对对象的访问

可以在调用目标标方法之前做一些其他的事情,增强目标类的业务方法

* 实现

静态代理在使用时,需要定义接口或者父类,被代理对象(房东)与代理对象()中介)一起实现相同的接口或者是继承相同父类.

代理类需要维护一个委托类的属性

* 代理模式在生活中的应用:

邀请明星唱歌,需要和经纪人联系,经纪人负责前期沟通演出费用,时间,场地,安排机票住宿,然后安排明星唱歌,然后收钱.

找中介租房子

原则:

1. 代理类和委托类行为相似(实现同一个接口)
2. 代理增强了委托类的行为(代理类拥有委托类的实例)

缺点:

因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.

## JDK动态代理:运行的时候, 动态生成代理

* 实现:

Java.lang.reflect.Proxy:

作用:生成代理类和对象

Java.lang.reflect.InvocationHandler(处理器接口):

可以通过invoke方法对目标进行调用

|  |
| --- |
| public class ProxyFactory {  //维护一个Object类型的委托类  private Object target;  public ProxyFactory(Object target) {  super();  this.target = target;  }  public Object getProxyInstance(){  return Proxy.newProxyInstance(  target.getClass().getClassLoader(),  target.getClass().getInterfaces(),  new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.out.println("打印日志");  Object result = method.invoke(target, args);  return result;  }  });  }  } |

JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数,完整的写法是:

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )

注意该方法是在Proxy类中是静态方法,且接收的三个参数依次为:

1.ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是固定的

2.Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型

3.InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入

注意:jdk动态代理要求委托类必须有接口.但是不要求代理类实现为委托类的接口

## 5 CGLIB动态代理(code generator libaray)

|  |
| --- |
| public class CglibProxyFactory implements MethodInterceptor{  //维护目标对象  private Object target;  public CglibProxyFactory(Object target) {  super();  this.target = target;  }  public Object getProxyInstance(){  //1.工具类,用于创建动态代理类  Enhancer en = new Enhancer();  //2.设置父类  en.setSuperclass(target.getClass());  //3.设置回调函数  en.setCallback(this);  //4.创建子类(代理对象)  return en.create();  }  @Override  public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {  System.out.println("开始事务...");  Object returnValue = method.invoke(target, args);  return returnValue;  }  } |

CGLIB动态代理功能更强大,不要求委托类必须有接口

而是通过继承目标类,通过目标来的子类来完成功能增强

Enhancer允许为非接口类型创建一个代理对象,Enhancer动态创建了给定类型的子类但是拦截了所有方法,和jdk的动态代理不同,不管目标类是否有接口都能正常工作

注意:目标对象不能被final修饰

需要导入jar包

cglib

asm

# 第五讲 ioc(inverse of control)

IOC是一种编程思想, 也算是一种设计模式.主要目的是为了解耦

在软件开发当中, 为了系统易于维护, 要求职责单一, 一个对象只负责处理自己范围类的业务,比如在MVC模式当中,Controller只负责分发请求, Serivice只负责逻辑处理, Dao只负责持久化数据库. 但是为了完成功能, 一般是需要两个以上的对象合作完成来实现业务. 使得每个对象都和其他对象产生了关联,也就是耦合. 如果说一个对象持有的引用需要靠自身来创建(new的方式), 代码耦合过高难以测试和维护.

IOC的模式提供了解耦的方案.

模拟IOC:

思路:

工厂模式

单例

解析xml(JAXB)(Java Architecturefor XML Binding)

注解:

反射创建对象

反射获取属性的set方法注入属性

核心:

核心方法 就是调用 setXxxx(xxxx);

需要反射构造setter方法 然后执行设置值

构造setter方法的时候 需要知道 属性名称 和 属性类型

在执行方法的时候 需要知道 方法所属对象 以及参数