**程序设计模式**

**1.什么是设计模式？**

编写软件的过程中，程序员面临着来自耦合性、内聚性、可维护性、可拓展性、重用性及灵活性等多方面的挑战。 设计模式可以让程序（软件），具有更好的

1.代码重用性（相同功能的代码，不用多次编写）

2.可读性（编程规范性，便于其他程序员阅读和理解）

3.可拓展性（需要新增功能时，非常方便）

4.可靠性（增加新的功能后，对原来的功能没有影响）

5.使程序呈现高内聚、低耦合的特性

**2.设计模式的原则?**

程序员在编程时，应当遵守的原则（设计模式这样设计的依据）

**1.单一职责原则**

对类来说，即一个类应该只负责一项职责；如果一个类负责两个不同的职责，当其中一个职责的需求发送变化时,可能造成另外一个职责执行错误。所以需要将类的粒度分解为两个类;

注意事项：

1.降低类的复杂度，一个类只负责一项职责；

2.提高类的可读性，可维护性；

3.降低变更应用的风险；

4.通常，我们应该遵守单一职责原则，只有逻辑足够简单，才可以在代码级违反单一职责原则。

**2.接口隔离原则**

客户端依赖它不需要的接口，即一个类对另外一个类的依赖应该建立在最小的接口上；

**3.依赖倒转（倒置）原则**

依赖倒置的中心思想是面向接口编程，抽象不应该依赖细节，细节应该依赖抽象；

依赖关系传递的三种方式

1.接口传递

2.构造方法传递

3.Setter方式传递

**4.里氏替换原则**

子类可以扩展父类的功能，但不能改变父类原有的功能。也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。

注意事项：

1.在子类继承父类时尽量不去重写父类的方法；

2.继承增强了两个类之间耦合性，在适当的情况下可以通过聚合、组合、依赖来解决；

在实际编程中，我们常会通过重写父类的方法实现新的功能，这样写起虽然简单，但是整个继承体系的复用性会比较差，特别是运行多态比较频繁的时候。

通常的做法是：

原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合、组合等关系替代；

**5.开闭原则**

开闭原则是编程中最基础、最重要的设计原则；

一个软件的实体类、模块和函数之间应该对拓展开放（对提供方），对修改关闭（使用方），用抽象构建框架，用实现拓展细节；当软件需要变化时，尽量通过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实现变化。

**6.迪米特原则**

一个对象应该对其他对象保持最少的了解,类与类的关系越密切，耦合越大；迪米特原则,又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的public方法，不对外泄露任何信息;

迪米特法则简单定义：只与直接的朋友通信；每个对象都会与其他对象有耦合关系，只有两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式有很多如依赖、关联、聚合等，其中，我们称出现成员变量、方法参数、方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接朋友，也就是说陌生的类最好不要以局部变量的方式出现在内的内部；

迪米特法则的核心是降低类之间的耦合；由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低类之间（对象间）耦合关系，并不是要求完全没有依赖关系；

**7.合成复用原则**

尽量使用合成、聚合的方式，而不是使用继承；

**3.设计原则的核心思想？**

1.找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起；

2.针对接口编程，而不是针对实现编程；

3.为了交互对象之间的松耦合设计而努力；

**4.UML**

UML是一种用于软件系统分析和设计的语言工具，用于帮助软件开发人员进行思考和记录

**5.UML类图**

1.用于描述系统中类（对象）本身的组成和类（对象）之间的各种静态关系；

2.类之间的关系：依赖、泛化(继承)、实现、关联、聚合、组合

**1.依赖**

依赖：只有是在类中用到了对方，他们之间就存在依赖关系;如果没有对方，连编译都通过不了；

1.类中用到了对方；

2.如果是类的成员属性；

3.如果是方法的返回类型；

4.是方法接收的参数类型

5.方法中使用到的；

**2.泛化**

泛化：泛化实际上就是继承关系，是依赖关系的特例；如果A类继承了B类，我们就说A和B存在泛化关系；

**3.实现**

实现关系：实际上就是A类实现了B类，是依赖关系的特例；

**4.关联**

关联关系(Association)：类与类之间的联系(一对一、一对多),是依赖关系的特例;

**5.聚合**

聚合关系：表示的是整体和部分的关系,整体和部分可以分开。聚合关系是关联关系的特例，如一台电脑有键盘、显示器，鼠标组成；组成电脑的各个配件是可以从电脑上分离出来的，使用带空心的菱形表示；

**6.组合**

组合：表示的是整体和部分的关系，但是整体和部分不可以分开;在程序中定义Person实体类； 如Person和IDCard是聚合，Person和Head是组合；但是如果在程序中Person实体中定义了对IDcard进行级联删除，即删除Person时连同IDCard一起删除，那么IDCard和Person就是组合关系；

# 设计模式-单例模式

## 1.什么是单例模式

单例模式是采用一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得实例的方法(静态方法)。

## 2.单例模式实现思路

1.私有化构造函数 （防止new）

2.类的内部创建对象

3.暴露一个静态的公共方法

## 2.饿汉式（两种方式）

### 1.优缺点

1.优点：写法简单，在类装载的时候完成实例化，避免了线程同步问题。

2.缺点：在类装载的时候就完成实例化，没有达到懒加载的效果。如果从开始至终从未使用过这个实例，则会造成内存的浪费。

3.总结：这种单例模式可以用,可能造成内存浪费；

### 2.实现

#### 1.静态常量方式

public class Singleton1 {

*//1.构造私有化*

private Singleton1() {

}

*//2.在本类中创建类的实例对象*

private static  Singleton1 singleton = new Singleton1();

*//3.暴露一个静态的公共方法，返回实例对象*

public static Singleton1 newInstance() {

 return singleton;

}

}

#### 2.静态代码块方式

public class Singleton2 {

private static Singleton2 singleton;

*//1.构造私有化*

private Singleton2() {

}

*//2.使用静态代码块创建对象实例*

static {

 singleton = new Singleton2();

}

*//3.暴露静态的公共方法，返回对象实例*

Public static Singleton2 newInstance() {

 return singleton;

}

}

## 3.懒汉式

### 1.懒汉式（线程不安全）

#### 1.代码实现

public class Singleton3 {

private static Singleton3 instance;

*//1.私有化构造函数*

private Singleton3() {

}

*//2.提供公共的静态方法，返回实例对象*

public static Singleton3 getInstance() {

 if (null == instance) {

  instance = new Singleton3();

 }

 return instance;

}

}

#### 2. 优缺点

1.起到了懒加载的效果，但是只能在单线程下使用；

2.如果在多线程下，一个线程进入if (null == instance)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这是便会产生多个实例。所以在多线程环境不可以使用这种方式。

3.结论：在实际开发中，不要使用这种方式；

### 2.懒汉式（线程安全，同步方法）

#### 1.代码实现

public class Singleton3 {

private static Singleton3 instance;

*//1.私有化构造函数*

private Singleton3() {

}

*//2.加入同步代码块，解决线程不安全问题*

public static synchronized Singleton3 getInstance() {

 if (null == instance) {

  instance = new Singleton3();

 }

 return instance;

}

}

#### 2. 优缺点

1.解决了线程不安全问题；

2.效率低，每个线程在想获得类的实例的的时候，执行getInstance()方法都要进行同步，而其实这个方法只执行一次实例化代码就够了，后面的想获取该类的实例，直接return就行了。方法进行同步效率太低。

3.结论：在实际开发中，不推荐使用这种方式

### 3.懒汉式（线程安全，同步方法）

#### 1.代码实现

public class Singleton4 {

private static Singleton4 instance;

*//1.私有化构造函数*

private Singleton4() {

}

*//2.加入同步代码块，解决线程不安全问题*

public static  Singleton4 getInstance() {

 if (null == instance) {

  synchronized (Singleton4.class) {

   instance = new Singleton4();

  }

 }

 return instance;

}

}

#### 2. 优缺点

1.这种方式，本意是想对第前面方式的改进，因为前面同步方法效率太低，改为同步产生实例化的代码块；

2.但是这种同步并不能起到线程同步的作用；如果一个线程进入了if (null == instance) 判断语句块，还未来得及往下执行，另一个线程也会通过这个判断语句,这时便会产生多个实例；

3.结论：在时间开发中，不能使用这种方式

## 4.双重检验方式

### 1.代码实现

public class Singleton4 {

private static Singleton4 instance;

*//1.私有化构造函数*

private Singleton4() {

}

*//2.加入同步代码块，解决线程不安全问题*

public static synchronized Singleton4 getInstance() {

 if (null == instance) {

  synchronized (Singleton4.class) {

   if (null == instance) {

    instance = new Singleton4();

   }

  }

 }

 return instance;

}

### 1.优缺点

1.Double-Check概念是多线程开发中常使用到的，如代码中所示，我们进行了两次if (null == instance)检查，这样可以保证线程安全了。

2.这样，实例化代码只用执行一次，后面再次访问，判断if (null == instance)，返回实例对象，也避免反复进行方法同步；

3.线程安全；延迟加载；效率更高;

4.结论：在实际开发中推荐使用此种单例模式

## 5.静态内部内方式

知识储备：需要对静态内部类有了解

1.当Singleton类被加载时，内部类SingeltonInstance不会被加载；

2.当调用Singeleton中getInstance时使用内部类中的静态变量时会导致内部类装载,只会装载一次;

### 1.代码实现

public class Singleton5 {

*//1.私有化构造函数*

private Singleton5() {

}

*//2.在静态内部类中实例会对象*

private static class SingletonInstance{

 private static final Singleton5 INSTANCE = new Singleton5();

}

*//3.返回对象*

public static Singleton5 getInstance() {

 return  SingletonInstance.INSTANCE;

}

}

### 2.优缺点

1.这种方式采用了类装载的机制来保证初始化实例时只有一个线程；

2.静态内部类的方式在Singleton5 类被加载时并不会立即实例化,而是在需要实例化时，调用getInstance方法，才会被装载SingletonInstance类，从而完成Singleton5 的实例化；

3.类的静态属性只会在第一次加载类的时候初始化，所有在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程无法进入的；

4.优点：避免了线程不安全，利用静态内部类特点实现延迟加载，效率高；

5.结论：推荐使用

## 6.枚举方式

### 1.代码实现

public enum Singleton {

INSTANCE;

public void method() {

*//方法实现*

 System.out.println("hello");

}

public static void main(String[] args) {

 Singleton instance = Singleton.INSTANCE;

 instance.method();

}

}

### 2.优缺点

1.这借助JKD1.5中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止序列化重新创建新的对象;

2.结论：推荐使用

## 7.单例模式使用说明

1.单例模式保证类系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统的性能。

2.当想实例化一个单例类的时候,必须要记住使用相应的获取对象的方法，而不是使用new。

3.单例模式使用的场景：需要频繁的进行创建和销毁的对象、创建对象时耗时过多或耗费资源过多（即：重量级对象）,但又经常用到的对象、工具类对象、频繁访问数据库或文件的对象（比如数据源、session工厂等）

**设计模式-代理模式(Proxy)**

**1.静态代理**

**1.优缺点**

1.在不修改目标对象的方法的前提下，能通过代理对象对目标对象进行拓展；

2.因为目标对象和代理对象需要实现相同的接口，一旦接口增加方法，目标对象和代理对象都需要维护;

**2.实例**

1.定义接口

public interface HouseProvider {

 void rentHouse();

}

2.创建目标对象（被增强对象）

public class HouseProviderImpl implements HouseProvider {

@Override

public void rentHouse() {

*// TODO Auto-generated method stub*

System.out.println("房屋出租");

}

}

3.创建代理对象（增强对象）

public class Houseproxy implements HouseProvider {

private HouseProvider target;

public Houseproxy(HouseProvider target) {

 this.target = target;

}

 @Override

 public void rentHouse() {

*// TODO Auto-generated method stub*

  System.out.println("代理-房产中介-拓展逻辑");

  target.rentHouse();

 }

}

4.创建客户端

public class Client {

public static void main(String[] args) {

 HouseProvider house = new HouseProviderImpl();

  Houseproxy houseproxy = new Houseproxy(house);

  houseproxy.rentHouse();

}

}

**2.JDK动态代理**

使用JDK动态代理，利用Proxy类里面的newProxyInstance方法创建代理对象；

**实例**

1.创建接口

public interface HouseProvider {

void buyHouse();

}

2.创建目标对象（增强类）

public class HouseProviderImpl implements HouseProvider {

 @Override

 public void buyHouse() {

*// TODO Auto-generated method stub*

  System.out.println("房屋出售");

 }

}

3.构建ProxyFactory

public class ProxyFactory {

*//目标对象(被增强的对象)*

private Object target;

public ProxyFactory(Object target) {

 this.target = target;

}

*/\*\**

 \*  public static Object newProxyInstance(ClassLoader loader,

                                          Class<?>[] interfaces,

                                          InvocationHandler h)

 \* loader:指定目标对象使用的类加载器，获取加载器的方式固定;

 \* interfaces:目标对象实现的接口类型,使用泛型方法确认类型

 \* h:事情处理:执行目标对象的方法时，会触发事情处理器方法

 \*/

public  Object  getProxyInstance() {

 return Proxy.newProxyInstance(target.getClass().getClassLoader(), target.getClass().getInterfaces(), new InvocationHandler() {

*//增强逻辑*

  @Override

  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

*// TODO Auto-generated method stub*

   System.out.println("增强的方法执行前执行");

*//执行被增强的方法*

   Object result = method.invoke(target, args);

   System.out.println("增强的方法执行后执行");

   return result;

  }

 });

}

}

4.创建client

public class Client {

public static void main(String[] args) {

HouseProviderImpl houseProviderImpl = new HouseProviderImpl();

HouseProvider proxyInstance = (HouseProvider)new ProxyFactory(houseProviderImpl).getProxyInstance();

proxyInstance.buyHouse();

}

}

**3.CGLib代理**

**1.概述**

1.静态代理和JDK代理模式都要求目标对象实现一个接口，但是有时候目标对象只是一个单独的对象，并没有实现任何接口，这个时候可以使用可以使用目标对象子类实现代理，可以使用Cglib代理;

2.Cglib代理也叫子类代理，他是在内存中构建的一个子类对象从而实现对目标对象功能的拓展，CgLib代理也归属与动态代理;

3.Cglib是一个强大的高性能的代码生成包，它可以产生在运行期拓展java类与实现java接口。它广泛的被许多AOP的框架使用，例如spring AOP，事项方法拦截

4.在Aop编程中如何选择代理模式：

1.）目标对象需要实现接口，用JDK代理

2.）目标对象不需要实现接口，使用Cglib代理

5.Cglib包的底层是通过使用字节码处理框架ASM来转换字节码并生成新的类;

**2.实现：**

1.需要引入Cglib的jar文件

2.在内存中动态构建子类，注意代理的类不能为final，否则报错 java.lang.IIIegalArgumentException

3.目标对象的方法如果是final/static,那么就不会被拦截，即不会执行目标对象额外的业务方法

**3.实例：**

1.创建目标对象

public class HouseProvider {

 public void show() {

  System.out.println("你好，房屋出售");

 }

}

2.构建ProxyFactory

public class ProxyFactory implements MethodInterceptor {

*//目标对象*

private Object target;

public ProxyFactory(Object target) {

 this.target = target;

}

public Object getProxyInstance() {

*//1.创建一个工具类*

    Enhancer enhancer = new Enhancer();

*//2.设置父类*

    enhancer.setSuperclass(target.getClass());

*//3.设置回调函数*

    enhancer.setCallback(this);

*//4.创建子类对象,即代理对象*

    return enhancer.create();

}

 @Override

 public Object intercept(Object arg0, Method method, Object[] arg2, MethodProxy arg3) throws Throwable {

*// TODO Auto-generated method stub*

  System.out.println("Cglib代理-->被增强方法之前执行");

*//执行被增强的方法*

  Object result = method.invoke(target, arg2);

  System.out.println("Cglib代理-->被增强方法之后执行");

  return result;

 }

}

3.创建client

public class Client {

public static void main(String[] args) {

 HouseProvider provider = new HouseProvider();

 HouseProvider proxyInstance = (HouseProvider)new ProxyFactory(provider).getProxyInstance();

 proxyInstance.show();

}

}

**4.代理模式的变体**

**1．防火墙代理**

内网通过代理穿透防火墙，实现对内网的访问.

**2．缓存代理**

如当请求图片文件等资源时，先到缓存代理取，如果取不到资源，再到公网或者数据库去，然后缓存；

**3．远程代理**

远程对象的本地代表，通过它可以把远程对象本地对象来调用。远程代理通过网络和真正的远程对象沟通信息；

**4．同步代理**

同步代理主要使用在多线程编程中，完成多线程间同步工作；

**设计模式-工厂模式**

[设计模式](http://localhost:9093/itour/work/detail?id=133)

阅读  最后发布于 2022-09-07 16:22

**1.简单（静态）工厂模式**

**1.概述**

简单工厂模式是属于创建型模式，是工厂模式的一种。简单工厂模式是由一个工厂对象决定创建出哪一种产品的实例。简单工厂模式是工厂模式家族中的最简单的模式。

在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就可以使用工厂模式来构建对象。

**2.实现**

1.通过抽象类(继承)或接口实现多态;

public interface ComputerProvider {

}

2.定义一个创建对象的类，由这个类来封装实例化对象的行为。

public class ComputerFactory {

public static ComputerProvider getComputer(String type) {

 if("Dell".equals(type)) {

  return new SimpleDellProviderImpl();

 }else if("Lenovo".equals(type)){

  return new SimpleLenovoproviderImpl();

 }

 return null;

}

}

3.实现产品

public class SimpleDellProviderImpl implements ComputerProvider {

*//具体实现*

}

public class SimpleLenovoproviderImpl implements ComputerProvider {

*//具体实现*

}

4.使用产品

public class Client {

 public static void main(String[] args) {

  SimpleDellProviderImpl computer =(SimpleDellProviderImpl) ComputerFactory.getComputer("Dell");

  System.out.println(computer);

 }

}

**3.优缺点**

1.容易理解，操作简单。

2.程序的可拓展性、和可维护性不好。违反了设计模式的ocp原则,即对扩展开放,对修改关闭。即当我们给类新增功能的时候，尽量不修改代码或者尽可能少修改代码；

3.如果我们需要增加一个电脑的种类需要修改ComputerFactory中代码;

**2.工厂方法模式**

**1.概述**

定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将对象的实例化推迟到子类。

**2.实现**

1.定义抽象方法

public abstract class AbstractComputer {

  abstract Computer create();

}

2.创建子类实例化对象

public class SimpleLenovoFactory extends AbstractComputer {

 @Override

 public Computer create() {

*// TODO Auto-generated method stub*

  return new Computer("Lenovo");

 }

}

3.使用对象

public class Client {

public static void main(String[] args) {

 SimpleDellFactory simpleDellFactory = new SimpleDellFactory();

 Computer make = simpleDellFactory.create();

}

**3.抽象工厂模式**

**1.概述**

抽象工厂模式定义一个接口用于创建相关或有依赖关系的对象，而无需指明具体的类。可以将简单工厂模式和工厂方法模式进行整合。

从设计层面来看，抽象工厂模式就是对简单工厂模式的改进（或者称为进一步抽象）。将工厂抽象成两层，抽象工厂和具有实现的工厂子类。程序可以根据创建对象的类型使用对应的工厂子类。更便于代码的维护和拓展。

**2.实现**

1.定义一个接口用于创建相关或有依赖关系的对象，而无需指明具体的类；

public interface FactoryProvider {

  Computer create();

}

2.创建子类实例化对象，将实例化操作交给自己完成；

public class DellDeskTopFactoryImpl implements FactoryProvider {

 @Override

 public Computer create() {

*// TODO Auto-generated method stub*

  return new Computer("Dell 台式机");

 }

}

public class DellLaptopFactoryImpl implements FactoryProvider {

 @Override

 public Computer create() {

*// TODO Auto-generated method stub*

  return new Computer("Dell笔记本");

 }

}

3.使用对象

public class Client {

public static void main(String[] args) {

 DellDeskTopFactoryImpl desktop = new DellDeskTopFactoryImpl();

 Computer create = desktop.create();

 System.out.println(create.getBrand());

}

**4.优缺点**

1.将对象的实例化，交给了子类自己实现，提高了代码的拓展性；

2.会产生很多的子类；

**5.抽象工厂模式应用场景**

1.Calenadar；

**4.总结**

1.工厂模式的意义：将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的依赖关系的解耦。从而提高项目的拓展和维护性。

2.创建对象实例时，不直接new类，而是把这个类的new类的动作放到一个工厂的方法中，并返回。（变量不要直接持有具体类的引用）

3.不要让类继承具体类，而是继承抽象类或者实现接口。

4.不要覆盖基类中已经实现的方法。

**设计模式-建造者模式**

**1.传统方式**

**1.实现**

1.使用抽象类或接口定义产品

public interface HouseProvider {

  void buildBasic();*//打地基*

  void buildWalls();*//砌墙*

  void roofed();*//封顶*

  House build();

}

2.实现产品

public class HouseProviderImpl  implements HouseProvider{

 @Override

 public House build() {

*// TODO Auto-generated method stub*

   buildBasic();

   buildWalls();

   roofed();

  return new House("商品房");

 }

 @Override

 public void buildBasic() {

*// TODO Auto-generated method stub*

  System.out.println("打地基完成");

 }

 @Override

 public void buildWalls() {

*// TODO Auto-generated method stub*

  System.out.println("砌墙完成");

 }

 @Override

 public void roofed() {

*// TODO Auto-generated method stub*

  System.out.println("封顶完成");

 }

}

3.使用产品

public class Client {

public static void main(String[] args) {

 House build = new HouseProviderImpl().build();

}

}

**2.优缺点**

1.优点：容易理解,操作简单。

2.缺点：设计的程序结构过于简单，没有设计缓存层对象，程序的拓展和维护不好（这种设计，把产品（房子）和创建产品的流程（建房子的过程）封装在了一起，耦合性增强了）

3.解决方案：将产品和产品构建过程解耦=》建造者模式；

**2.使用建造者模式**

**1.概述**

1.建造者模式（Builder Pattern）又叫生成器模式，是一种对象构建模式。他可以将复杂的对象的建造过程抽取出来（抽象类）级别，使这个抽象过程的不同实现方法可以构造出不同表现(属性)的对象;

2.建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就来构建他们，用户不需要知道内部的具体构建细节。

3.建造者模式的四个角色：

1.)Product（产品角色）：一个具体的产品对象；

2.)Builder(产品建造者)：创建一个Product对象的各个部件指定的接口|抽象类；

3.)ConcreteBuilder（具体建造者）：实现接口，构建和转配各个部件

4.)Director(指挥者):构建一个使用Builder接口的对象。主要用于创建一个复杂的对象；

4.主要有两个作用：

1.)隔离客户和对象的生成过程；

2.)负责控制产品对象的生产过程；

**2.实现**

1.具体的产品

public class House {

private String type;

public String getType() {

 return type;

}

public void setType(String type) {

 this.type = type;

}

public House() {

}

public House(String type) {

 this.type = type;

}

}

2.产品构建者

public interface HouseBuild {

  void buildBasic();*//打地基*

  void buildWalls();*//砌墙*

  void roofed();*//封顶*

}

3.具体建造者

public class HouseBuilder  implements HouseBuild{

 @Override

 public void buildBasic() {

*// TODO Auto-generated method stub*

  System.out.println("打地基完成");

 }

 @Override

 public void buildWalls() {

*// TODO Auto-generated method stub*

  System.out.println("砌墙完成");

 }

 @Override

 public void roofed() {

*// TODO Auto-generated method stub*

  System.out.println("封顶完成");

 }

}

4.Director(指挥者)

public class HouseDirector {

private HouseBuild houseBuild;

public HouseDirector(HouseBuild houseBuild) {

 this.houseBuild = houseBuild;

}

public void setHouseBuild(HouseBuild houseBuild) {

 this.houseBuild = houseBuild;

}

public House constructHouse() {

 return new House();

}

}

5.使用产品

public class Client {

public static void main(String[] args) {

House constructHouse = new HouseDirector(new HouseBuilder()).constructHouse();

}

}

**3.使用场景**

StringBuilder