

1 认识设计模式

1.1 设计模式简介

软件设计模式 (Software Design Pattern)，俗称设计模式，**设计模式**是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。

使用设计模式的**目的**是为了代码重用、让代码更容易被他人理解、保证代码可靠性。

设计模式：

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。

设计模式使用场景：

- 1、在程序软件架构设计上会使用到设计模式
- 2、在软件架构设计上会使用到设计模式

设计模式的目的：

- 1、提高代码的可重用性
- 2、提高代码的可读性
- 3、保障代码的可靠性

在《设计模式》这本书的最大部分是一个目录，该目录列举并描述了 23 种设计模式。



1.2 设计模式的六大原则

单一职责原则：类或者接口要实现职责单一

里氏替换原则：使用子类来替换父类，做出通用的编程

依赖倒置原则：面向接口编程

接口隔离原则：接口的设计需要精简单一

迪米特法则：降低依赖之间耦合

开闭原则：对扩展开放，对修改关闭

1.3 单例模式

单例模式（Singleton Pattern）是 Java 中最常见的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

单例模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。该类还提供了一种访问它唯一对象的方式，其他类可以直接访问该方法获取该对象实例而不需要实例化该类的对象。

单例模式特点：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。

单例模式优点：

- 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。
- 2、避免对资源的多重占用（比如写文件操作）。

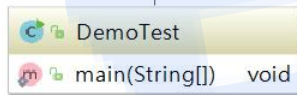
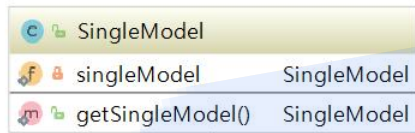
单例模式真实应用场景：

- 1、网站的计数器
- 2、应用程序的日志应用
- 3、数据库连接池设计
- 4、多线程的线程池设计

1.3.1 单例模式-饿汉式

创建一个单例对象`SingleModel`，`SingleModel` 类有它的私有构造函数和本身的一个静态实例。

`SingleModel` 类提供了一个静态方法，供外界获取它的静态实例。`DesignTest` 我们的演示类使用`SingleModel`类来获取 `SingleModel` 对象。



SingleModel的唯一实例由SingleModel自己创建
DemoTest调用SingleModel中的方法获取唯一实例

singleton/SingletonModel.java

```
package com.open.design.singleton;

public class SingletonModel {

    //整个应用程序中只有一个自己的实例
    private static SingletonModel singleModel = new SingletonModel();

    //只能自己创建自己
    private SingletonModel() {
    }

    //需要提供一个方法让外界调用获取实例
    public static SingletonModel getInstance(){
        return singleModel;
    }
}
```

1.3.2 多种单例模式讲解

单例模式有多种创建方式，刚才创建方式没有特别的问题，但是程序启动就需要创建对象，不管你用不用到对象，都会创建对象，都会消耗一定内存。因此在单例的创建上出现了多种方式。

懒汉式：

懒汉式有这些特点：

- 1、延迟加载创建，也就是用到对象的时候，才会创建
- 2、线程安全问题需要手动处理(不添加同步方法，线程不安全，添加了同步方法，效率低)
- 3、实现容易

singleton/SingletonModel1.java

```
package com.open.design.singleton;

public class SingletonModel1 {
    //整个应用程序中只有一个自己的实例
    private static SingletonModel1 singleModel;

    //只能自己创建自己
    private SingletonModel1() {
    }

    //需要提供一个方法让外界调用获取实例
    public static synchronized SingletonModel1 getInstance(){
        if(singleModel==null){
            singleModel = new SingletonModel1();
        }
        return singleModel;
    }
}
```

测试

singleton/SingletonTest.java

```
package com.open.design.singleton;

import com.carrotsearch.sizeof.RamUsageEstimator;

import java.util.ArrayList;
import java.util.List;
public class SingletonTest {
    //测试单例
    /*public static void main(String[] args) {
        SingletonModel singleModel1 = SingletonModel.getInstance();
        SingletonModel singleModel2 = SingletonModel.getInstance();
        System.out.println(singleModel1);
        System.out.println(singleModel2);
    }*/
}
```

```

}*/
//比较单例和非单例创建对象对内存的占用情况
/* public static void main(String[] args) {
    //存储创建的新实例对象
    List<SingleModel> array = new ArrayList<SingleModel>();

    //获取实例
    SingleModel singleModel1 = SingleModel.getInstance();
    array.add(singleModel1);

    for (int i = 0; i <100000 ; i++) {
        SingleModel singleModel2 = SingleModel.getInstance();
        if(singleModel1!=singleModel2){
            array.add(singleModel2);
        }
    }
    System.out.println(RamUsageEstimator.sizeOf(array));
}*/

public static void main(String[] args) {
    //存储创建的新实例对象
    List<Cat> array = new ArrayList<Cat>();

    //获取实例
    Cat cat1 = new Cat();
    array.add(cat1);

    for (int i = 0; i <100000 ; i++) {
        Cat cat2 = new Cat();
        if(cat2!=cat1){
            array.add(cat2);
        }
    }
    System.out.println(RamUsageEstimator.sizeOf(array));
}
}

```

我们测试创建10万个对象，用单例模式创建，仅占内存： 104 字节，而如果用传统方式创建10万个对象，占内存大小为 2826904 字节。

```

public class SingleModel1 {

    //不实例化
    private static SingleModel1 instance;    ①不直接实例化

    //让构造函数为 private，这样该类就不会被实例化
    private SingleModel1() {}

    //获取唯一可用的对象
    public static SingleModel1 getInstance() {
        //instance为空的时候才创建对象，这里线程不安全
        if(instance==null){
            instance = new SingleModel1();
        }
        return instance;
    }

    public void useMessage() {
        System.out.println("Single Model!");
    }
}

```

instance对象为空的时候，才创建对象
如果是多线程场景下，这里不安全

如果在创建对象实例的方法上添加同步synchronized，但是这种方案效率低，代码如下：

```
//获取唯一可用的对象 添加了同步可以解决多线程安全问题，但这么做明显会降低获取对象实例的效率。
public static synchronized SingleModel1 getInstance() {
    //instance为空的时候才创建对象，这里线程不安全
    if(instance==null){
        instance = new SingleModel1();
    }
    return instance;
}
```

双重校验锁：SingleModel2

这种方式采用双锁机制，安全且在三线程情况下能保持高性能。

如果有A，B两个线程，第一次判断null,如果A和B为null,这样A和B就是并发场景。A获取锁进入下一个null判断，判断为null就会创建实例,释放锁。B获取锁进来，如果没有第二次判断为null，则B还会创建实例。

singleton/SingleModel2.java

```
package com.open.design.singleton;

public class SingleModel2 {
    //整个应用程序中只有一个自己的实例
    private static SingleModel2 singleModel;

    //只能自己创建自己
    private SingleModel2() {
    }

    //需要提供一个方法让外界调用获取实例
    public static SingleModel2 getInstance(){
        if(singleModel==null){
            //如果singleModel为空，则创建对象实例
            synchronized (SingleModel2.class){
                if(singleModel==null){
                    singleModel = new SingleModel2();
                }
            }
        }
        return singleModel;
    }
}
```

2 常用设计模式剖析

2.1 观察者模式

定义：

对象之间存在一对多或者一对一依赖，当一个对象改变状态，依赖它的对象会收到通知并自动更新。

MQ其实就属于一种观察者模式，发布者发布信息，订阅者获取信息，订阅了就能收到信息，没订阅就收不到信息。

优点：

- 1、观察者和被观察者是抽象耦合的。
- 2、建立一套触发机制。

缺点：

- 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
- 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

observer/AbstrackInfo.java

```
package com.open.design.observer;

public abstract class AbstrackInfo {
    //被监听的对象
    private Clock clock;
    abstract void message();
}
```

observer/EatInfo.java

```
package com.open.design.observer;

public class EatInfo extends AbstrackInfo {
    @Override
    public void message() {
        System.out.println("大家吃饭了!");
    }
}
```

observer/SleepInfo.java

```
package com.open.design.observer;

public class SleepInfo extends AbstrackInfo{
    @Override
    public void message() {
        System.out.println("大家午睡了!");
    }
}
```

observer/Clock.java

```
package com.open.design.observer;

import java.util.ArrayList;
import java.util.List;

public class Clock {
    //一对多，需要接到通知的对象
    private List<AbstrackInfo> infos = new ArrayList<AbstrackInfo>();

    public void say() {
        System.out.println("上班了!");
        //通知
        update();
    }

    //添加要得到通知的对象
    public void addInfo(AbstrackInfo info) {
        infos.add(info);
    }

    //通知
    public void update() {
        for (AbstrackInfo info : infos) {
            info.message();
        }
    }

    public static void main(String[] args) {
        Clock clock = new Clock();
        AbstrackInfo eat = new EatInfo();
        AbstrackInfo sp = new SleepInfo();
    }
}
```



```

        clock.addInfo(eat);
        clock.addInfo(sp);

        clock.say();
    }
}

```

2.2 代理模式

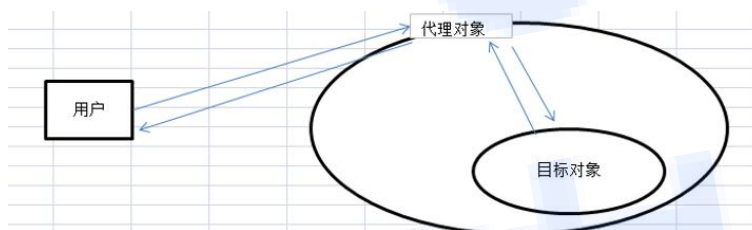
1) 代理模式定义

给目标对象提供一个代理对象，并由代理对象控制对目标对象的引用；目的：

(1) 通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要复杂性；

(2) 通过代理对象对原有的业务增强；

举例：明星 ---经纪人<-----用户



代理模式有静态代理和动态代理两种实现方式。

2) 静态代理

这种代理方式需要代理对象和目标对象实现一样的接口。

优点：可以在不修改目标对象的前提下扩展目标对象的功能。

缺点：

冗余。由于代理对象要实现与目标对象一致的接口，会产生过多的代理类。

不易维护。一旦接口增加方法，目标对象与代理对象都要进行修改。

staticproxy/IUserDao.java

```

package com.open.design.staticproxy;

public interface IUserDao {
    void save();
}

```

staticproxy/UserDao.java

```

package com.open.design.staticproxy;
//目标对象
public class UserDao implements IUserDao{
    @Override
    public void save() {
        System.out.println("保存数据");
    }
}

```

staticproxy/UserDaoProxy.java

```

package com.open.design.staticproxy;
//代理对象
public class UserDaoProxy implements IUserDao{
    private IUserDao target;//null 目标对象
    public UserDaoProxy(IUserDao target){
        this.target=target;
    }
}

```



```

@Override
public void save() {
    System.out.println("开启事务");
    target.save();
    System.out.println("提交事务");
}
}

```

staticproxy/App.java

```

package com.open.design.staticproxy;

public class App {
    public static void main(String[] args) {
        IUserDao target=new UserDao(); //多态目标对象
        IUserDao proxy=new UserDaoProxy(target); //代理对象
        //target.save();
        proxy.save();
    }
}

```

3) 动态代理

动态代理利用了JDK API，动态地在内存中构建代理对象，从而实现对目标对象的代理功能。

动态代理又被称为JDK代理或接口代理。

dynamicproxy/IUserDao.java

```

package com.open.design.dynamicproxy;

// 接口
public interface IUserDao {
    void save();
}

```

dynamicproxy/UserDao.java

```

package com.open.design.dynamicproxy;
//目标对象
public class UserDao implements IUserDao{
    @Override
    public void save() {
        System.out.println("-----已经保存数据!!! -----");
    }
}

```

dynamicproxy/ProxyFactory.java

```

package com.open.design.dynamicproxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

/**
 * 给所有的 dao 创建代理对象【动态代理】
 * 代理对象，不需要实现接口
 *
 */
public class ProxyFactory {
    private Object target; //目标对象

    public ProxyFactory(Object target){
        this.target=target;
    }
}

```

```

//返回一个代理对象
public Object getProxyInstance() {
    return Proxy.newProxyInstance(
        //指定当前目标对象是哪个类
        target.getClass().getClassLoader(), //userDao
        //userDao 接口是那个
        target.getClass().getInterfaces(), //IUserDao
        //找目标对象中的那个方法去调用
        new InvocationHandler() {
            //proxy 目标对象
            //method 调用方法
            //args 参数
            @Override
            public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                System.out.println("开启事务");
                Object returnValue=method.invoke(target, args);//执
行目标对象中的方法

                System.out.println("提交事务");
                return returnValue;
            }
        }
    );
}
}

```

dynamicproxy/App.java

```

package com.open.design.dynamicproxy;

public class App {
    public static void main(String[] args) {
        IUserDao target=new UserDao(); //目标对象
        System.out.println(target.getClass());
        //代理对象
        IUserDao proxy=(IUserDao) new
ProxyFactory(target).getProxyInstance();
        proxy.save();
    }
}

```

4) 静态代理与动态代理的区别

1. 静态代理在编译时就已经实现，编译完成后代理类是一个实际的class 文件
2. 动态代理是在运行时动态生成的，即编译完成后没有实际的 class 文件，而是在运行时动态生成类字节码，并加载到 JVM 中

注意：动态代理对象不需要实现接口，但是要求目标对象必须实现接口，否则不能使用动态代理。

JDK 中生成代理对象主要涉及两个类：

第一个类为 java.lang.reflect.Proxy，通过静态方法 newProxyInstance 生成代理对象，
第二个为 java.lang.reflect.InvocationHandler 接口，通过 invoke 方法对业务进行增强；

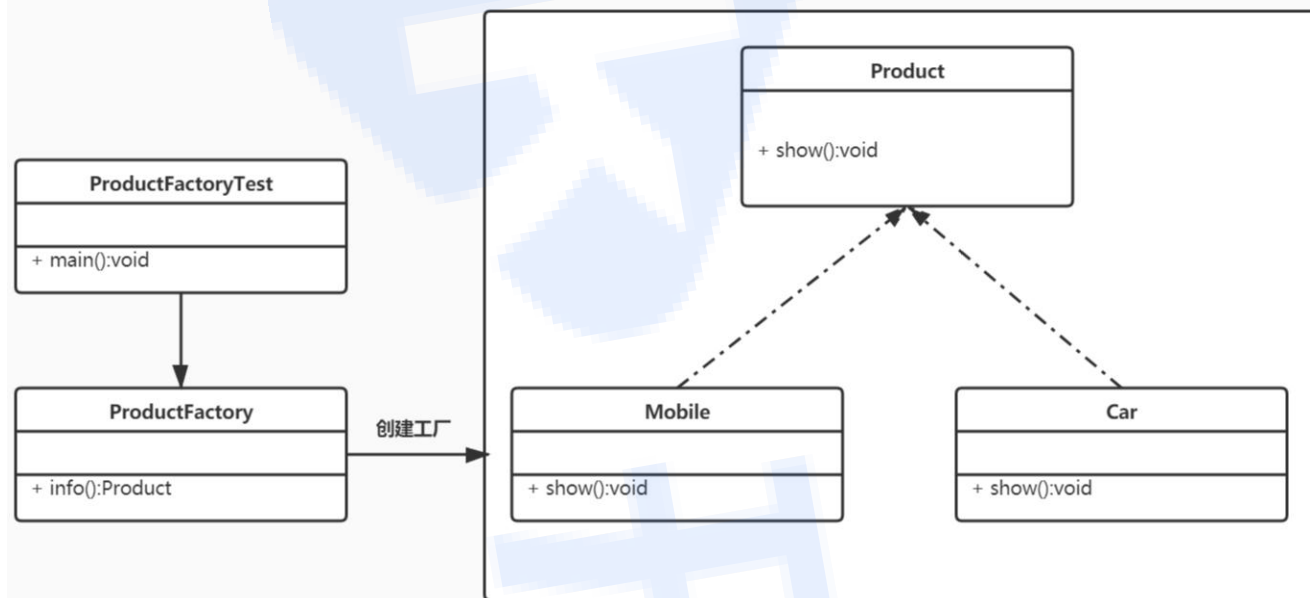
2.3 工厂设计模式

定义：

工厂模式 (Factory Pattern) 是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。

优点:

- 1、一个调用者想创建一个对象，只要知道其名称就可以了。
- 2、屏蔽产品的具体实现，调用者只关心产品的接口。
- 3、降低了耦合度



我们来做这么一个案例，创建一个接口Product和Product的实现类Mobile以及Car，再定义一个具体的工厂对象ProductFactory，并通过ProductFactory来获取指定的Product。

Product接口：

factory/Product.java

```
package com.open.design.factory;

public interface Product {
    //查看产品详情
    void show();
}
```

创建接口实现类Mobile:

factory/Mobile.java

```
package com.open.design.factory;

public class Mobile implements Product {
    //产品详情
    @Override
    public void show() {
        System.out.println("手机: HUAWEI P40 Pro +");
    }
}
```

创建接口实现类Car:

factory/Car.java

```
package com.open.design.factory;

public class Car implements Product {
    @Override
    public void show() {
        System.out.println("汽车: 比亚迪!");
    }
}
```

创建工厂ProductFactory, 根据参数创建指定产品对象:

factory/ProductFactory.java

```
package com.open.design.factory;

public class ProductFactory {
    //根据用户的需求创建不同的产品
    public static Product getBean(String name){
        if(name.equals("mobile")){
            return new Mobile();
        }else if(name.equals("car")){
            return new Car();
        }
        return null;
    }
}
```

使用工厂ProductFactory创建指定对象:

factory/ProductFactoryTest.java

```
package com.open.design.factory;

public class ProductFactoryTest {

    public static void main(String[] args) {
        Product mobile = ProductFactory.getBean("mobile");
        Product car = ProductFactory.getBean("car");
        mobile.show();
        car.show();
    }
}
```

运行结果如下:



```
Run: ProductFactoryTest x
C:\java\jdk1.8.0_151\bin\java.exe ...
手机: HUAWEI P40 Pro +
汽车: 比亚迪!
```

2.4 适配器模式

定义:

将一个类的接口转换成客户希望的另外一个接口, 使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

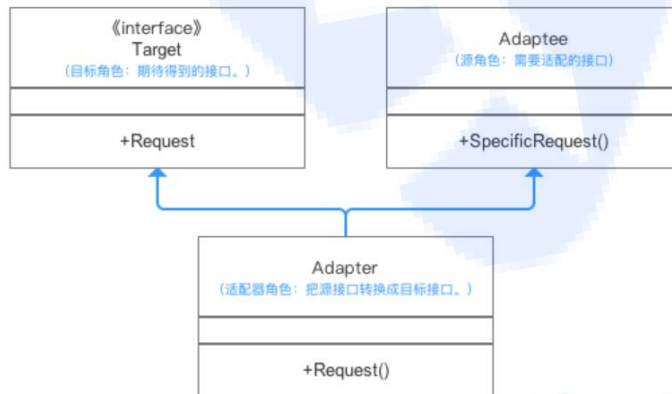
优点:

- 1、可以让任何两个没有关联的类一起运行。
- 2、提高了类的复用。
- 3、灵活性好。

缺点:

过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。

日志模块的第一个需求是一个典型的使用适配器模式的场景，适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁，将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作；类图如下



冲突：Target期待调用Request方法，而Adaptee并没有（这就是所谓的不兼容了）。

解决方案：为使Target能够使用Adaptee类里的SpecificRequest方法，故提供一个中间环节Adapter类（**继承Adaptee & 实现Target接口**），把Adaptee的API与Target的API衔接起来（适配）。（国外电器用的电源转换器）

案例：

背景：小成买了一个进口的电视机

冲突：进口电视机要求电压（110V）与国内插头标准输出电压（220V）不兼容

解决方案：设置一个适配器将插头输出的220V转变成110V

步骤1：创建Target接口（期待得到的插头）：能输出110V

Target.java

```
package com.open.design.adapter;

public interface Target {
    //将 220v 转换输出 110v
    public void Convert_110v();
}
```

步骤2：创建源类（原有的插头）；

PowerPort220V.java

```
package com.open.design.adapter;

public class PowerPort220V {
    //原有插头只能输出 220v
    public void Output_220v() {
    }
}
```

步骤3：创建适配器类（Adapter）

Adapter220V.java

```
package com.open.design.adapter;

class Adapter220V extends PowerPort220V implements Target{
    //期待的插头要求调用 Convert_110v(), 但原有插头没有, 因此适配器补充上这个方
```

法名

```
//但实际上 Convert_110v() 只是调用原有插头的 Output_220v() 方法的内容
//所以适配器只是将 Output_220v() 作了一层封装, 封装成 Target 可以调用的
Convert_110v() 而已
@Override
public void Convert_110v() {
    System.out.println("适配器将 220v 转为 110v");
    this.Output_220v();
}
}
```

测试

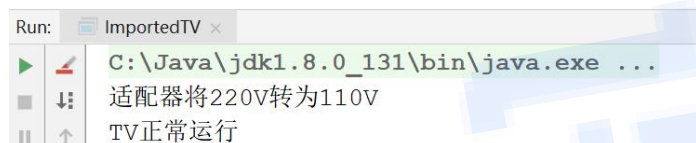
ImportedTV.java

```
package com.open.design.adapter;

//通过 Adapter 类从而调用所需要的方法
public class ImportedTV {
    public static void main(String[] args) {
        Target mAdapter220V = new Adapter220V();
        ImportedTV mImportedMachine = new ImportedTV();
        //用户拿着进口机器插上适配器 (调用 Convert_110v() 方法)
        //再将适配器插上原有插头 (Convert_110v() 方法内部调用 Output_220v() 方法输出 220V)
        //适配器只是个外壳, 对外提供 110V, 但本质还是 220V 进行供电
        mAdapter220V.Convert_110v();
        mImportedMachine.Work();
    }

    public void Work() {
        System.out.println("TV 正常运行");
    }
}
}
```

测试效果如下:



Run: ImportedTV x

C:\Java\jdk1.8.0_131\bin\java.exe ...

适配器将220V转为110V

TV正常运行