Java设计模式

1、单例模式

1-1、饿汉模式

单例类

```
public class SingleTon
{

    //1、将构造函数私有化，让外界无法创建对象
    private SingleTon()
    {

    }

    //2、由类的内部提供一个私有的静态的对象,我们只提供get方法，而不提供set方法，因为构造函数被私有化了，外界无法创建该对象，没有对象就无法完成赋值，
    // 那么要Set就没有意义
    //缺点：无论用户是否使用该对象，该对都已经被创建出来了，且占据了内存空间
    private static SingleTon st = new SingleTon();  //在本类中，private还是可以被访问

    private int age;

    //  Get ~ Set


    //3、只给外界提供get方法就可以了
    public static SingleTon getSt()
    {
        return st;
    }

    public int getAge()
    {
        return age;
    }

    public void setAge(int age)
    {
        this.age = age;
    }
}
```

测试类

```
public class Main
```

```
{
    public static void main(String[] args)
    {
        //可以创建对象是因为有无参构造函数,当构造函数被private修饰后，外界就无法正常创建对象了
//        SingleTon st = new SingleTon();
//        SingleTon st2 = new SingleTon();
//        System.out.println(st.st);
//        System.out.println(st2.st);;

        //未封装属性时，可以使用
//        System.out.println(SingleTon.st);
//        System.out.println(SingleTon.st);
//
//        //想给对象中的属性赋值
//        SingleTon.st.age = 10;
//        SingleTon.st.age = 20;
//        System.out.println(SingleTon.st.age);

        //使用单例方法操作
        SingleTon.getSt().setAge(10);
        System.out.println(SingleTon.getSt().getAge());

    }
}
```

1-2、静态内部

单例类

```
public class SingleTon
{
    private SingleTon()
    {

    }

    //私有静态内部类,内部类也属于类的内部，所以可以使用外部类中被private修饰的东西
    private static class innerClass
    {
        //final的用意是让这个变量不可以修改
        private final static SingleTon SINGLE_TON = new SingleTon();
    }

    public static SingleTon getInstance()
    {
        return innerClass.SINGLE_TON;
    }
}
```

测试类

```
public class Main
{
    public static void main(String[] args)
    {
        SingleTon s1 = SingleTon.getInstance();
        SingleTon s2 =SingleTon.getInstance();

        System.out.println(s1);
        System.out.println(s2);
    }
}
```

1-3、懒汉模式

单例类

```
public class Person
{
    //1、私有化构造函数
    private Person()
    {

    }
    //2、提供一个静态的对象，保证了对象只有一个
    private static Person person = null;
    //3、提供一个静态的get方法，供外界使用
    //4、加上synchronized就是线程安全的，在多线程下也可以正常使用
    //5、如果没有synchronized，就是非线程安全，那么再多线程情况下可能会同时产生多个对象
    public synchronized static Person getPerson()
    {
        //在用户第一次使用的时候，检查对象是否为Null，如果为null在给它开辟空间
        //区别就是饿汉模式一直有。懒汉模式需要的时候才有
        //同步代码块，注意因为操作是静态变量，所以要锁类.class
//        synchronized (Person.class)
//        {
//
//        }
        if(person == null)
        {
            person = new Person();
        }
        return person;
    }

    //正常的属性
    private int age;
    private String name;

    public int getAge()
    {

        return age;
```

```
    }

    public void setAge(int age)
    {
        this.age = age;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}
```

测试类

```
public class Main
{
    public static void main(String[] args)
    {
        Person p1 = Person.getPerson();
        Person p2 = Person.getPerson();

        System.out.println(p1);
        System.out.println(p2);
    }
}
```

2、模板模式

抽象父类

```
public abstract class Teacher
{
    //交由子类去完成
    public  abstract void beike();
    public abstract  void neirong();

    public void shangke()
    {
        System.out.println("老师开始上课...");
    }
    public void xiake()
    {
        System.out.println("老师下课");
    }

```

```
    public void all()
    {
        beike();   //父类没有实现
        shangke();
        neirong();  //父类也没有实现
        xiake();
    }
}

```

具体子类：语文老师

```
public class ChineseTeacher extends Teacher
{
    @Override
    public void beike()
    {
        System.out.println("备课语文");
    }

    @Override
    public void neirong()
    {
        System.out.println("授课语文");
    }
}
```

具体子类：英语老师

```
public  class EnglishTeacher extends Teacher
{
    @Override
    public void beike()
    {
        System.out.println("备课英语");
    }

    @Override
    public void neirong()
    {
        System.out.println("授课英语");
    }
}

```

测试类

```
public class Main
{
    public static void main(String[] args)
    {
        Teacher chineseTeacher = new ChineseTeacher();

        Teacher englishTeacher = new EnglishTeacher();

        chineseTeacher.all();
        englishTeacher.all();
    }
}
```

3、工厂模式

3-1、简单工厂

父类：宠物

```
public class Pet
{

    public void eat()
    {

    }
}
```

子类：猫

```
public class Cat extends Pet
{
    //让子类重写父类的eat方法

    @Override
    public void eat()
    {
        System.out.println("吃小鱼");
    }
}

```

子类：狗

```
public class Dog extends Pet
{
    @Override
    public void eat()
    {
        System.out.println("爱吃骨头");
    }
}
```

工厂类:用来生产pet的子类

```

public class PetFactory
{
    //注意和单例不同，每调用一次方法，就会产生一个新的对象
    public static Pet getPet(String type)
    {
        //提供一个静态的方法，来返回子类的对象
        Pet pet = null;
        switch (type)
        {
            case "cat":
                pet = new Cat();
                break;
            case "dog":
                pet = new Dog();
                break;

        }

        return pet;

    }
}
```

测试类

```
public class Main
{
    public static void main(String[] args)
    {
        //通过工程来获取对象
        Pet p1 = PetFactory.getPet("cat");
        Pet p2 = PetFactory.getPet("cat");
        System.out.println(p1);
        System.out.println(p2);

        p1.eat();
        p2.eat();

        Pet p3 = PetFactory.getPet("dog");
```

```
        Pet p4 = PetFactory.getPet("dog");

        p3.eat();
        p4.eat();

    }
}
```

3-2、复杂工厂

猫类接口:

```
public interface ICat
{
    public void sleep();
}
```

子类：猫

```
public class Cat implements ICat
{
    @Override
    public void sleep()
    {
        System.out.println("猫猫Zzzz");
    }
}
```

工厂：猫

```
public class CatFactory
{
    public static ICat getCat()
    {
        return new Cat();
    }
}
```

狗类接口:

```
public interface IDog
{
    public void eat();
}

```

子类：狗

```
public class Dog implements IDog
{
    @Override
    public void eat()
    {
        System.out.println("狗狗爱吃骨头");
    }
}
```

子类：哈士奇

```
public class HaShiQiDog implements IDog
{

    @Override
    public void eat()
    {
        System.out.println("哈士奇吃饭如打仗");
    }
}
```

工厂：狗

```
public class DogFactory
{
    public static IDog getDog()
    {
        return new Dog();
    }
    public static IDog getHaShiQi()
    {
        return new HaShiQiDog();
    }
}
```

4、代理模式

4-1、静态代理

统一的接口

```
public interface IHouse
{
    //收租的方法
    void rent();
}

```

房主

```
public class Host implements IHouse
{
    @Override
    public void rent()
    {
        System.out.println("房主要1500/月");
    }
}

```

代理

```
public class HostProxy implements IHouse
{
    //在代理中获取到一个房主的对象，这个方法对象地位很高
    Host host;
    //构建代理的时候，必须要有本主
    public HostProxy(Host host)
    {
        this.host = host;
    }


    @Override
    public void rent()
    {
        System.out.println("代理对象收费1800/元");
        //本主收费
        host.rent();

        System.out.println("本次代理盈利300/月");
    }
}
```

测试

```
public class Main
{
    public static void main(String[] args)
    {
        Host house = new Host();
//        house.rent();

        IHouse house2 = new HostProxy(house);
        house2.rent();
    }
}
```

4-2、动态代理

统一的接口

```
public interface IHouse
{
    //收租的方法
    void rent();
}
```

房主

```
public class Host implements IHouse
{
    @Override
    public void rent()
    {
        System.out.println("房主要1500/月");
    }
}

```

统一的代理对象

```
public class HostProxy implements InvocationHandler
{
    IRent rent;

    public HostProxy(IRent rent)
    {
        this.rent = rent;
    }

    //通过反射动态创建代理对象

    public Object getProxy()
```

```
    {
        //JDK通过反射 获取到了当前类（HostProxy）,获取到了房东类rent,获取到了他父类 IRent.class
        Object o = Proxy.newProxyInstance(this.getClass().getClassLoader(),new Class[]
{IRent.class},this);

        return o;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        System.out.println("中介收费2000/月");

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

        return result;
    }
}
```

测试

```
public static void main(String[] args)
    {
        IRent h = new Host();
        //JDK
        HostProxy proxy = new HostProxy(h);
        //获取自己的代理对象
        Object obj = proxy.getProxy();
        if(obj instanceof  IRent)
        {
            IRent rentProxy = (IRent)obj;

            rentProxy.rent();
        }

    }
```