设计模式解析

*单例模式

1，概述

单例模式通俗的讲就是单例对象的类必须保证只有一个实例存在。整个系统中就只有一个类的实例。

2，实现方式

- 饿汉式单例：在类加载的过程中。就主动创建实例。

  饿汉式单例代码：

```
public class singleclass {
    public static void main(String[] args) {
        T x = T.getT();
    }
}

class T {
    private static T singleT = new T();
    public T(){}
    public static T getT() {
        return singleT;
    }
}
```

- 懒汉式单例：等到真正使用的时候才会去创建实例。

```
public class singleclass2 {
    public static void main(String[] srgs) {
        T x = T.getT();
    }
}

class T {
    private static T singleT;

    public T() {
    }

    public static T getT() {
        if (singleT == null)
            singleT = new T();
        return singleT;
    }
```

```
}
```

3,单例模式的优点

- 在内存当中，只有一个对象，节省空间。
- 避免频繁的创建销毁对象，可以提高性能。
- 避免对共享资源的多重占用。
- 为整个系统提供一个全局的访问点。

4，单例模式的线程安全问题

- 由于饿汉式是再类加载的时候就创建出一个类实例对象，因此这种方式具有线程安全的特性。
- 但是对于懒汉式来说，在多线程的环境下则不宜动就是安全的。因为在在懒汉式实现单例模式的时候，有这样一句话：if(singleT==null) singleT=new T()；这句话当处于多个线程的时候，可能会在if判断句的地方发生歧义，不能保证线程安全。

  解决办法：

  双重校验线程安全

```
import com.sun.xml.internal.txw2.output.StaxSerializer;

public class DoubleCheckThreadsafe {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            new Thread(){
                @Override
                public void run() {
                    System.out.println(Test.getTest());
                }
            }.start();
        }
    }
}

// 饿汉模式
class Test {

    public static volatile Test test = null;
    public static Test getTest() {
        if (test == null) {                //第一次检查
            synchronized (Test.class) {
                if (test == null)          //第二次检查
                    test = new Test();
                return test;
            }
        }
        return test;
    }

}
```

*适配器模式

1，类适配器模式

```
interface A{
    void test();
}
interface B{
    void good();
}
class BB implements B{
    void good(){
        System.out.println("Very Good !!!");
    }
}
```

需求：A中的test想要访问B接口中的good方法。
类适配器模式：
在创建一个适配器类

```
class Adapter extends B implements A{
    void test(){
        good();
    }
}
```

2，对象适配器模式

与类适配器模式差不多，只是该模式是通过对象的实例来进行适配的。

```
interface A{
    void test();
}
interface B{
    void good();
}
class BB implements B {
    public void good(){
        System.out.println("Very Good!!!");
    }
}
```

需求与上面的一样

```
class Adapter implements A{
    private BB b;  //通过一个对象的实例来进行适配。
    Adapter(BB b){
        this.b = b;
    }
    void test(){
        this.b.good();
    }
}
```

*工厂模式

1，普通工厂模式

```
interface A{
    void test();
}
class B implements A{
    public void test(){
        System.out.println("B Class");
    }
}
class C implements A {
    public void test(){
        System.out.println("C Class");
    }
}
//工厂可以根据get方法中参数的类型来进行相应的实例化
class Factory{
    public A get(String Type){
        if("C".equals(Type)){
            return new C();
        } else if("B".equals(Type)){
            return new B();
        } else{
            System.out.println("输入有误！！！");
        }
    }


}
```

2，多个工厂模式

```
/*
单个工厂模式当输入的Type不合法的时候，那么就不能创建对象
A,B,C不变
改变Factory即可
*/
class Factory{
    public A Get_B(){
        return new B();
    }
    public A Get_C(){
        return new C();
    }
}
```

3，静态工厂方法模式

```
/*
    就是将Factory中的方法设置成static即可。不用创建工厂类的实例就可以调用。
*/
class Factory{
    public static A Get_B(){
        return new B();
    }
    public static A Get_C(){
        return new C();
    }
}
```

工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。

4，抽象工厂模式

```
/**
    理解：就是当增加一个新的功能的时候，就创建一个工厂类去完成，不需要在原来的类中去修改相应的代码
**/
interface A{
    void test();
}
class B implements A{
    public void test(){
        System.out.println("B Class");
    }
}
class C implements A {
    public void test(){
        System.out.println("C Class");
    }
}
interface Factory{
    A get();
}
//工厂可以根据get方法中参数的类型来进行相应的实例化
class Factory1 implements Factory{
    public A get(){
        return new B();
    }
}
class Factory2 implements Factory{
    public A get(){
        return new B();
    }
}
//这种方式扩展性较好
```