单例模式

一、基本介绍

1、定义

指一个类只有一个实例，且该类能自行创建这个实例的一种模式。

2、特点：

- **单例类只有一个实例对象；**
- **该单例对象必须由单例类自行创建；**
- **单例类对外提供一个访问该单例的全局访问点；**

3、结构与实现

单例模式是设计模式中最简单的模式之一。通常，普通类的构造函数是公有的，外部类可以通过“new 构造函数()”来生成多个实例。但是，如果将类的**构造函数设为私有的**，外部类就无法调用该构造函数，也就无法生成多个实例。这时该类自身必须定义一个**静态私有实例**，并向外提供一个**静态的公有函数**用于创建或获取该静态私有实例。

二、实现方式

1、饿汉式单例

1）特点：

类一旦加载就创建一个单例，保证在调用 getInstance 方法之前单例已经存在了。

2)线程安全

饿汉式单例在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以是线程安全的，可以直接用于多线程而不会出现问题。

3）缺点

- 空间使用率不高

- 类加载时实例化，意味着该类无法在程序运行过程中通过运行参数实例化，代码失去灵活性。

```
public class HungrySingleton{
  // 静态变量，类在创建之初就会执行实例化动作。
    private static final HungrySingleton instance=new HungrySingleton();
   // 私有化构造函数，使外界无法创建实例
    private HungrySingleton(){}
    // 为外界提供获取实例接口
    public static HungrySingleton getInstance()
    {
        return instance;
    }
}
```

2、懒汉式单例

1）特点（与饿汉式相反）

类加载时没有生成单例，只有当第一次调用 getlnstance 方法时才去创建这个单例。

2）缺点

如果编写的是多线程程序，则不要删除上例代码中的关键字 volatile 和 synchronized，否则将存在线程非安全的问题。

如果不删除这两个关键字就能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源

```
public class LazySingleton {
    private static volatile LazySingleton instance=null;    //保证 instance 在所有线程中同步
    private LazySingleton(){}    //private 避免类在外部被实例化
    public static synchronized LazySingleton getInstance() {
        //getInstance 方法前加同步
        if(instance==null)
        {
            instance=new LazySingleton();  // 懒加载
        }
        return instance;
    }
}
```