单例模式的三要素

1. 构造方法私有化
2. 静态属性修饰的实例
3. public static的 getInstance方法，返回第二步实例引用

单例模式的四种写法

饿汉单例模式

先创建一个实例等着调用

```
        //私有化构造方法使得该类无法在外部通过new 进行实例化
        private GiantDragon(){  }

        //准备一个类属性，指向一个实例化对象。 因为是类属性，只有一个并且不需要实例即可使用
        private static GiantDragon instance = new GiantDragon();

        //public static 方法，提供给调用者获取12行定义的对象
        public static GiantDragon getInstance(){
            return instance;}
           public static void main(String[] args) {
                GiantDragon g1 = GiantDragon.getInstance();
            }
    }
```

优点：1.线程安全
缺点1.可能造成浪费，无论是否会用到这个对象，都会加载。
2.还有一个漏洞，别人可以通过反射的方式创建一个新对象。

懒汉单例模式

只有在调用getInstance的时候，才会创建实例。

```
public class GiantDragon {
    //私有化构造方法使得该类无法在外部通过new 进行实例化
    private GiantDragon(){
    }
      //准备一个类属性，用于指向一个实例化对象，但是暂时指向null
    private static GiantDragon instance;

    //public static 方法，返回实例对象
    public static GiantDragon getInstance(){
        //第一次访问的时候，发现instance没有指向任何对象，这时实例化一个对象
        if(null==instance){
            instance = new GiantDragon();
        }
```

```
        //返回 instance指向的对象
        return instance;
    }

}
```

注意这样写是线程不安全的！
问题出在直接"if(null==instance){"判断实例为null，就创建对象。
为什么说是线程不安全的呢。请模拟一下两个线程同时来创建对象结果会发生什么？
如果两个对象同时判断为空，结果就会创造了两个实例，就不是单例模式啦。
因此加入**synchronized声明。**
但如果判断语句写在同步代码后面，导致同步块包括了判断语句，这并没有必要，最重要的是很影响速度。（代码如下）

```
synchronized (Singleton.class) {
  if (instance == null) {
```

如果在同步代码写在判断语句前面，可能会出现两个线程同时判断都为空，才进入同步块，线程1创建完一个对象后，线程2还会创造一个对象，只是两个线程创建对象不是同时发生而已，并没有解决线程不安全问题（代码如下）

```
if (instance == null) {
  synchronized (Singleton.class){
```

重点来了，可以用**双重检测机制**，即在synchronized声明前后都判断一次是否为空，保证线程安全。
如果没有第一个判断：所有调用这个方法的线程都得先获取锁，不管此时实例是否为空，有没有必要。
如果没有第二个判断：如果两个线程同时判断为空，一个先取锁，一个后取锁。还是会创建两个对象，只不过是一个先一个后。第二个判断避免了两个线程** 同时判断为空 **先后获取锁创建对象的情况.线程1创建完实例后，线程2再执行的时候要经过第二次判断，此时已经有实例了，线程2就不满足创建条件。
双重检测机制不会影响效率。因为在第一次判断语句不是在同步块内，并没有影响多少效率。
第二次判断虽然在同步块内，但只有当实例为空的时候需要获取锁。

另外，实例是非原子性的，有可能出现**指令重排问题**，因此实例用**volatile**修饰。

```
    private static volatile Singleton instance = null;
    private Singleton(){};

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class){
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance; }}
```

懒汉式优点：相对于饿汉模式；来说在启动的时候，会感觉到比饿汉式略快，因为并没有做对象的实例化。 但是在第一次调用的时候，会进行实例化操作，感觉上就略慢。
懒汉式缺点：1.麻烦，需要我们来自己加锁，保证线程安全的问题。
2.还是可以通过反射的方式来破坏单例模式。

静态内部类

```
public class Singleton3 {
    //静态内部类
    private static class LazyHolder{
        private static Singleton3 instance = new Singleton3();
    }
    //私有构造器
    private Singleton3(){};
    public static Singleton3 getInstance() {
        return LazyHolder.instance;
    }
}
```

由于**外部类无法访问静态内部类**，因此只有当外部类调用Singleton.getInstance()方法的时候，才能得到instance实例。并且，instance实例对象初始化的时机并不是在Singleton被加载的时候，而是当getInstance()方法被调用的时候，静态内部类才会被加载，这时instance对象才会被初始化。并且也是线程安全的。所以，与饿汉式相比，通过静态内部类的方式，可以保证instance实例对象不会被白白浪费。但是，它仍然存在反射问题。
优点：1.线程安全
2.不会浪费
缺点：还是反射问题

枚举

```
public enum Singleton4 {
instance;}
```

枚举方式优点：1.线程安全
2.代码简单
3.反射也不能获得多个对象，因为JVM能阻止反射获取枚举类的私有构造器
枚举方式缺点：和饿汉式一样，由于一开始instance实例就被创建了，所以有可能出现白白浪费的情况。