设计模式（一）单例模式形式汇总与场景分析

作者：hudk
原文链接：https://ld246.com/article/1633921681260
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

设计模式中，单例模式是一种常见的代码组织形式，它的设计意义，是为了满足让程序在指定的运行境中，可以且只可以创建出某个对象的一个实例出来的需求。不同的运行环境和场景，对实现单例的求是不一样的，比如，单线程环境下，不需要考虑并发问题，所以不需要加锁就可以满足单例要求。如，多线程下，加合适的锁虽然可以解决单例要求，单特定情况下又会出现其他可能的异常；还有，知道反射方式，也是可以创建对象的，这就要考虑如何避免这种漏洞被恶意利用。

总的来说，单例模式的实现可以分为三种形式

- 饿汉式
- 静态内部类。
- 懒汉式

每种方式的实现，都要考虑对应场景可能发生的问题。好了，直接上代码，分析都在注释里！

一、简单的饿汉式单例

```
package top.hudk.design.single;

/**
 * 作用：饿汉式单例模式
 * 缺点：
 * 容易受到反射攻击，即通过反射的方式，仍然可以创建出多个实例出来。（思考如何避免反射攻击？）
 * @author hudk
 * @date 2020/3/1 15:48
 */
public class HungrySingleton {
    private HungrySingleton instance = new HungrySingleton();
    private HungrySingleton(){

    }
    public HungrySingleton getInstance(){
        return instance;
    }
}
```

静态内部类单例形式

```
package top.hudk.design.single;

/**
 * 作用：静态内部类方式实现单例模式
 *
 * @author hudk
 * @date 2020/3/1 15:11
 */
public class InnerClassSingleton {

    /**
     * 静态内部类，是在被使用的时候，JVM虚拟机才会对他进行实例化
     * 所以，静态内部类方式，设计单例，就是利用了静态内部类的这种特点
     */
```

```
    private static class InnerClassHolder{
        private static InnerClassSingleton instence = new InnerClassSingleton();
    }

    private InnerClassSingleton(){
        /**
         * 这里的判断，是为了避免反射攻击造成的多实例出现。
         */
        if(InnerClassHolder.instence != null){
            throw new RuntimeException("本对象是单例模式，系统内不允许多出现个实例");
        }
    }
    public static InnerClassSingleton getInstance(){
        //运行到这里，JVM虚拟机才会对InnerClassHolder进行实例化，进而实例化InnerClassSinglet
n
        return InnerClassHolder.instence;
    }

}
```

懒汉式单例模式

```
package top.hudk.design.single;

/**
 * 简单懒汉式单例模式
 * 适用于单线程环境
 * 缺点:
 * 1、容易受到反射攻击
 * 2、不适用于多线程
 *
 * @author hudk
 * @date 2020/3/1 14:43
 */
public class LazySingleton {
    private LazySingleton instance = null;
    /**
     * 私有构造方法，防止外部调用来实例化
     */
    private LazySingleton() {
    }
    public LazySingleton getInstance() {
        if (instance == null) {
        instance = new LazySingleton();
            return instance;
        }
        return instance;
    }
}

/**
 * 懒汉式单例模式
 * 简单加锁版
 * 适用于多线程
```

```
 * 缺点：
 * 1、容易受到反射攻击
 * 2、性能问题：每次示例化时都需要加锁，锁对性能影响很大
 */
class LazySingleton2 {
    private LazySingleton2 instance2 = null;
    /**
     * 私有构造方法，防止外部调用来实例化
     */
    private LazySingleton2() {
    }
    public synchronized LazySingleton2 getInstance2() {
        if (instance2 == null) {
            instance2 = new LazySingleton2();
            return instance2;
        }
        return instance2;
    }
}

/**
 * 懒汉式单例模式
 * 局部加锁版
 * 适用于多线程。性能问题解决
 * 缺点：
 * 1、具有反射漏洞
 * 2、容易出现指令重排导致的空指针异常问题：
 */
class LazySingleton3 {
    private LazySingleton3 instance3 = null;
    /**
     * 私有构造方法，防止外部调用来实例化
     */
    private LazySingleton3() {
    }
    public LazySingleton3 getInstance3() {
        if (instance3 == null) {
            synchronized (LazySingleton3.class) {
                if (instance3 == null) {
                    instance3 = new LazySingleton3();
                    return instance3;
                    /**
                     * 此处：new LazySingleton3();
                     * 虚拟机，会分为三步：
                     * 1、分配内存
                     * 2、初始化实例
                     * 3、为引用赋值
                     * 由于，指令重排的存在，第2、3步可能顺序颠倒。
                     * 如果顺序颠倒的话，就会出现：
                     * 当一个线程执行到为引用赋值，但是实际的实例还未初始化时，
                     * 另一个线程正好执行到上面第一次判断是否为空的位置，那么该线程就会得到实例不空的结果，
                     * 但是实例现在其实是没有初始化的，只是对应的引用已经有了实例在内存中的地址值而已。
```

```
                * 这样，第二个线程有可能带着一个没有“初始化”的“实例”去执行其他的操作，
                * 当在其他操作中对该实例的某个属性进行访问时，由于该属性并没有初始化赋值，有能会出现不可预测的空指针异常问题（虽然概率极小）。
                */
            }
        }
    }
    return instance3;
  }
}

/**
 * 懒汉式单例模式
 * 通过局部加锁版，且禁止指令重排
 * 适用于多线程。性能问题和指令重排问题被解决
 * 缺点：
 * 1、具有反射漏洞
 */
class LazySingleton4 {
  /**
   * volatile关键词使属性new时不会被指令重排
   */
  private volatile LazySingleton4 instance4 = null;
  /**
   * 私有构造方法，防止外部调用来实例化
   */
  private LazySingleton4() {
  }
  public LazySingleton4 getInstance4() {
    if (instance4 == null) {
      synchronized (LazySingleton4.class) {
        if (instance4 == null) {
          instance4 = new LazySingleton4();
          return instance4;
          /**
           * 此处：new LazySingleton4();
           * 虚拟机，会分为三步：
           * 1、分配内存
           * 2、初始化实例
           * 3、为引用赋值
           * 由于，全局成员定义时，禁止了指令重排，第2、3步的顺序不会颠倒。
           * 就不会出现不可预测的空指针异常
           */
        }
      }
    }
    return instance4;
  }
}

/**
 * 懒汉式单例模式
 * 通过局部加锁版，且禁止指令重排，加上防反射检查。
 * 适用于多线程。性能问题、指令重排、和反射问题被解决
```

```
 */
class LazySingleton5 {
    /**
     * volatile关键词使属性new时不会被指令重排
     */
    private volatile LazySingleton5 instance5 = null;
    /**
     * 私有构造方法，防止外部调用来实例化
     */
    private LazySingleton5() {
        /**
         * 这里的判断，是为了避免反射攻击造成的多实例出现。
         */
        if(instance5 != null){
            throw new RuntimeException("本对象是单例模式，系统内不允许多出现个实例");
        }
    }
    public LazySingleton5 getInstance5() {
        if (instance5 == null) {
            synchronized (LazySingleton5.class) {
                if (instance5 == null) {
                    instance5 = new LazySingleton5();
                    return instance5;
                    /**
                     * 此处：new LazySingleton5();
                     * 虚拟机，会分为三步：
                     * 1、分配内存
                     * 2、初始化实例
                     * 3、为引用赋值
                     * 由于，全局成员定义时，禁止了指令重排，第2、3步的顺序不会颠倒。
                     * 就不会出现不可预测的空指针异常
                     */
                }
            }
        }
        return instance5;
    }
}
```

总结