Java单例

制作人: Tami

单例模式： 就是整个程序有且仅有一个实例。该类负责创建自己的对象，同时确保只有一个对象被创建。在Java，一般常用在工具类的实现或创建对象需要消耗资源时使用单例模式。

- 类构造器私有
- 持有自己类型的属性
- 对外提供获取实例的静态方法

1. 饿汉模式

```
/**
* 饿汉单例
*/
public class SingletonEh {
    //1.定义静态的私有对象,主动创建
    private static SingletonEh singletonEh =new SingletonEh();
    //2.私有化无参构造方法
    private SingletonEh() {
    }
    //3.以自己的实例返回静态的共有方法
    public static SingletonEh getInstance(){
        return singletonEh;
    }
}
```

- 优点
  - 写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题(线程安全的)。
- 缺点
  - 在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费

2.懒汉模式

```
/**
* 懒汉单例模式
*/
public class SingletonLh {
    //1.私有化自身的静态引用
    private static SingletonLh singletonLh;
    //2.私有化构造方法
    private SingletonLh(){}
```

```
    //3.公开以自身实例为返回值的静态方法
    public static SingletonLh getInstance(){
        if(singletonLh==null){
            singletonLh = new SingletonLh();
        }
        return singletonLh;
    }
}
```

- 优点
  - 单个实例被延迟加载,真正需要时,才会去实例化一个对象交给自己的引用
- 缺点
  - 在多线程的情况下,线程不安全

验证懒汉单例的线程安全问题:

```
//创建一个可以存放20个线程的线程池
ExecutorService threadPool = Executors.newFixedThreadPool(20);
for(int i=1;i<=20;i++){
    threadPool.execute(new Runnable() {
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName()+":"+
SingletonLh.getInstance());
        }
    });
}
threadPool.shutdown();
```

运行结果:

```
pool-1-thread-13:com.singleton.SingletonLh@517f8175
pool-1-thread-17:com.singleton.SingletonLh@517f8175
pool-1-thread-5:com.singleton.SingletonLh@517f8175
pool-1-thread-14:com.singleton.SingletonLh@517f8175
pool-1-thread-20:com.singleton.SingletonLh@517f8175
pool-1-thread-19:com.singleton.SingletonLh@517f8175
pool-1-thread-4:com.singleton.SingletonLh@517f8175
pool-1-thread-15:com.singleton.SingletonLh@517f8175
pool-1-thread-11:com.singleton.SingletonLh@517f8175
pool-1-thread-3:com.singleton.SingletonLh@517f8175
pool-1-thread-7:com.singleton.SingletonLh@517f8175
pool-1-thread-6:com.singleton.SingletonLh@517f8175
pool-1-thread-1:com.singleton.SingletonLh@2a0f1662
pool-1-thread-8:com.singleton.SingletonLh@517f8175
pool-1-thread-18:com.singleton.SingletonLh@517f8175
pool-1-thread-12:com.singleton.SingletonLh@517f8175
pool-1-thread-10:com.singleton.SingletonLh@517f8175
pool-1-thread-2:com.singleton.SingletonLh@517f8175
pool-1-thread-9:com.singleton.SingletonLh@3e3fbdaa
pool-1-thread-16:com.singleton.SingletonLh@517f8175
```

- 解决懒汉单例线程不安全的问题

```
/**
* 懒汉单例模式(单重检查)
*/
public class SingletonLh {
    //1.私有化自身的静态引用
    private static SingletonLh singletonLh;
    //2.私有化构造方法
    private SingletonLh(){}
    //3.公开以自身实例为返回值的静态方法
    public static SingletonLh getInstance() {
        if (singletonLh == null) {
            synchronized (SingletonLh.class) {
               singletonLh = new SingletonLh();
            }
        }
        return singletonLh;
    }
}
```

注意:这样线程还是不安全的:原因是当线程A去判断为空时,这是线程B如果抢得了cpu资源,她也会判断为null,这是就会创建一个对象,如果线程A得到CPU资源,由于原来已经判断过了,就不再判断,直接创建对象,这是就会产生两个不一样的对象,线程还是不安全.

- 解决方案: 双重检查加锁机制

```
/**
* 懒汉单例模式测试:双重检查加锁机制
*/
public class SingletonLh {
    //1.私有化自身的静态引用
    private static SingletonLh singletonLh;
    //2.私有化构造方法
    private SingletonLh(){}
    //3.公开以自身实例为返回值的静态方法
    public static SingletonLh getInstance() {
        if (singletonLh == null) {
            synchronized (SingletonLh.class) {
                if(singletonLh==null){
                    singletonLh = new SingletonLh();
                }
            }
        }
        return singletonLh;
    }
}
```