

## Homework:

### 1、 用 Java 书写具有双向加锁功能的单子模式 (volatile , synchronized)。

答: 利用 volatile 关键字和 synchronized 关键字来确保线程安全和单例对象的唯一性——volatile 关键字确保了多线程环境下该变量的可见性和部分有序性, 防止 JVM 的指令重排。synchronized 则用于在对象的实例化阶段提供互斥访问, 确保只有一个线程可以初始化单例。代码如下 (附录有完整代码):

```
1 public class Singleton {
2     // 使用volatile关键字确保多线程能够正确处理_instance变量
3     // 4 个用法
4     private static volatile Singleton _instance;
5
6     // 私有构造函数, 防止外部通过new创建实例
7     // 1 个用法
8     private Singleton() {}
9
10    // 获取单例对象的方法
11    // 2 个用法
12    public static Singleton getInstance() {
13        // 首次检查, 若实例已存在, 则直接返回
14        if (_instance == null) {
15            // 同步块, 确保线程安全
16            synchronized (Singleton.class) {
17                // 再次检查, 这是为了在null的情况下只创建一个实例
18                if (_instance == null) {
19                    _instance = new Singleton();
20                }
21            }
22        }
23        return _instance;
24    }
25 }
```

设计 SingletonTest 类用以验证: 在具有双向加锁功能的单子模式中, 无论多少线程尝试获取实例, 输出的 hashCode 应该都是相同的, 因为所有线程获取的是同一个对象实例。

```
1 public class SingletonTest {
2     // 0 个用法
3     public static void main(String[] args) {
4         // 创建多个线程, 每个线程尝试获取单例实例
5         Thread t1 = new Thread(() -> {
6             Singleton instance1 = Singleton.getInstance();
7             System.out.println("Instance 1: " + instance1.hashCode());
8         });
9         Thread t2 = new Thread(() -> {
10             Singleton instance2 = Singleton.getInstance();
11             System.out.println("Instance 2: " + instance2.hashCode());
12         });
13         // 启动线程
14         t1.start();
15         t2.start();
16         try {
17             t1.join();
18             t2.join();
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22     }
23 }
```

运行结果如下所示：



## 2、 用 Java 书写具有可变用例数目的孤子模式。

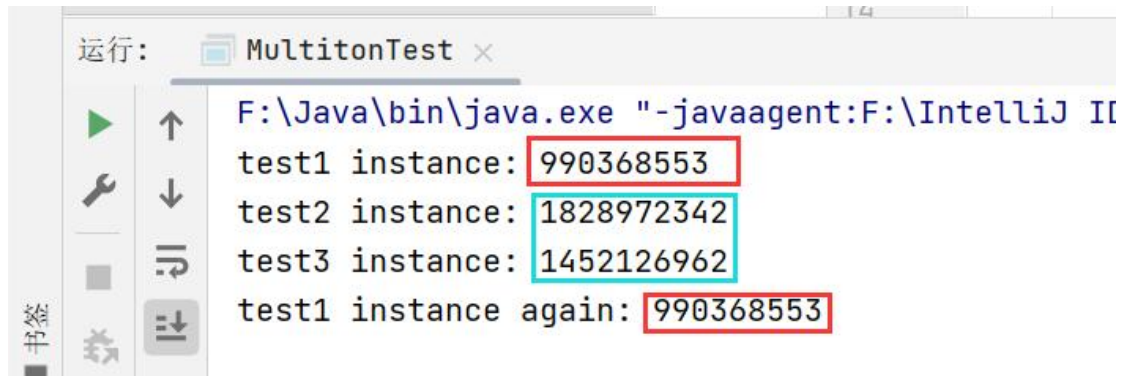
答：对于可变实例数目的孤子模式，使用一个静态的 HashMap 存储实例，键是实例的名称。getInstance 方法通过键来访问或创建实例，保证每个键只对应一个实例。使用 synchronized 关键字在方法级别上锁，确保线程安全。具体代码如下：

```
4 public class Multiton {
5     // 存储多个单例的HashMap
6     // 3 个用法
7     private static final Map<String, Multiton> instances = new HashMap<>();
8     // 1 个用法
9     private Multiton() {}
10    // 获取或创建一个命名的单例
11    // 4 个用法
12    public static synchronized Multiton getInstance(String key) {
13        // 检查实例是否存在
14        if (!instances.containsKey(key)) {
15            // 实例不存在, 创建新的实例
16            instances.put(key, new Multiton());
17        }
18        return instances.get(key);
19    }
20 }
```

编写 MultitonTest 类用于验证：在可变实例数目的孤子模式中，不同键的 hashCode 应该是不同的，但相同键的多次获取应该返回相同的实例。代码如下：

```
2 public static void main(String[] args) {
3     // 获取三个不同名字的实例
4     Multiton m1 = Multiton.getInstance(key: "test1");
5     Multiton m2 = Multiton.getInstance(key: "test2");
6     Multiton m3 = Multiton.getInstance(key: "test3");
7
8     // 打印实例的hash code来验证它们是否不同
9     System.out.println("test1 instance: " + m1.hashCode());
10    System.out.println("test2 instance: " + m2.hashCode());
11    System.out.println("test3 instance: " + m3.hashCode());
12
13    // 测试同名实例是否相同
14    Multiton m4 = Multiton.getInstance(key: "test1");
15    System.out.println("test1 instance again: " + m4.hashCode());
16 }
```

运行结果如下所示：



```
运行: MultitonTest x
F:\Java\bin\java.exe "-javaagent:F:\IntelliJ I
test1 instance: 990368553
test2 instance: 1828972342
test3 instance: 1452126962
test1 instance again: 990368553
```

### 3、 附录

#### 1) Singleton

```
public class Singleton {
    // 使用 volatile 关键字确保多线程能够正确处理_instance 变量
    private static volatile Singleton _instance;
    // 私有构造函数，防止外部通过 new 创建实例
    private Singleton() {}
    // 获取单例对象的方法
    public static Singleton getInstance() {
        // 首次检查，若实例已存在，则直接返回
        if (_instance == null) {
            // 同步块，确保线程安全
            synchronized (Singleton.class) {
                // 再次检查，在 null 的情况下只创建一个实例
                if (_instance == null) {
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
    }
}
```

#### 2) SingletonTest

```
public class SingletonTest {
    public static void main(String[] args) {
        // 创建多个线程，每个线程尝试获取单例实例
        Thread t1 = new Thread(() -> {
            Singleton instance1 = Singleton.getInstance();
            System.out.println("Instance 1: " +
instance1.hashCode());
        });
        Thread t2 = new Thread(() -> {
```

```

        Singleton instance2 = Singleton.getInstance();
        System.out.println("Instance 2: " +
instance2.hashCode());
    });
    // 启动线程
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

### 3) Multiton

```

import java.util.HashMap;
import java.util.Map;

public class Multiton {
    // 存储多个孤子的 HashMap
    private static final Map<String, Multiton> instances = new
HashMap<>();
    private Multiton() {}
    // 获取或创建一个命名的孤子
    public static synchronized Multiton getInstance(String key) {
        // 检查实例是否已存在
        if (!instances.containsKey(key)) {
            // 实例不存在, 创建新的实例
            instances.put(key, new Multiton());
        }
        return instances.get(key);
    }
}

```

### 4) MultitonTest

```

public class MultitonTest {
    public static void main(String[] args) {
        // 获取三个不同名字的实例
        Multiton m1 = Multiton.getInstance("test1");
        Multiton m2 = Multiton.getInstance("test2");
        Multiton m3 = Multiton.getInstance("test3");

        // 打印实例的 hash code 来验证它们是否不同
    }
}

```

```
System.out.println("test1 instance: " + m1.hashCode());
System.out.println("test2 instance: " + m2.hashCode());
System.out.println("test3 instance: " + m3.hashCode());

// 测试同名实例是否相同
Multiton m4 = Multiton.getInstance("test1");
System.out.println("test1 instance again: " + m4.hashCode());
    }
}
```