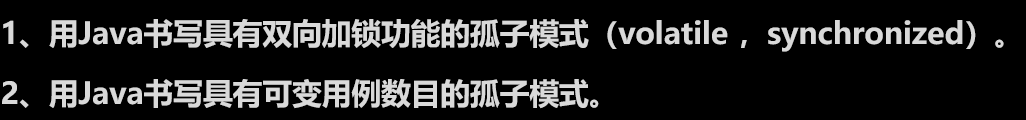
**作业六**

****

本次实验主要实现了两种单例模式的设计与实现。

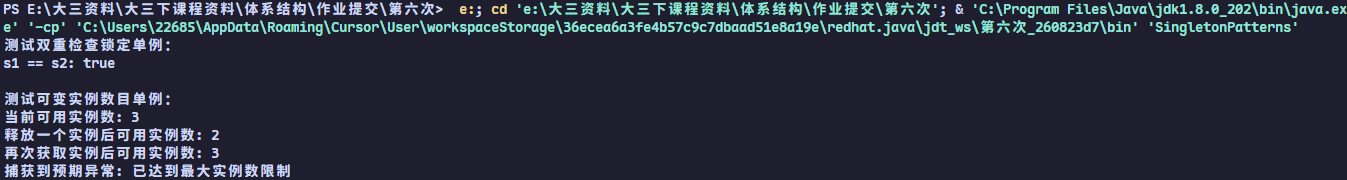
第一种是使用双重检查锁定机制的单例模式，第二种是支持可变实例数目的单例模式。

在双重检查锁定单例模式中，我们通过volatile关键字确保多线程环境下的可见性，并使用synchronized关键字进行同步控制。该实现采用了双重检查机制，即在获取实例时先进行一次无锁检查，如果实例不存在再进行加锁检查，这种设计既保证了线程安全，又提高了性能。

在可变实例数目的单例模式中，我们设计了一个支持最多3个实例的单例类，通过数组来管理实例，并提供了实例的获取和释放机制。该实现使用synchronized关键字确保线程安全，并通过availableInstances计数器来跟踪当前可用的实例数量。当请求超过最大实例数时，会抛出IllegalStateException异常。

通过测试代码的验证，这两种实现都能正确工作，双重检查锁定单例模式确保了全局唯一实例，而可变实例数目单例模式则实现了对实例数量的精确控制。这些实现展示了单例模式在不同场景下的灵活应用，同时也体现了Java并发编程中的一些重要概念，如可见性、原子性和线程安全等。

**系统运行截图：**



**Code:**

public class *SingletonPatterns* {

*/\*\**

*\* 1. 双重检查锁定的单例模式实现*

*\*/*

    public static class *DoubleCheckSingleton* {

*// volatile关键字确保instance变量的可见性*

        private static volatile *DoubleCheckSingleton* instance;

*// 私有构造函数防止外部实例化*

        private *DoubleCheckSingleton*() {

        }

*// 获取实例的公共方法*

        public static *DoubleCheckSingleton* *getInstance*() {

*// 第一次检查，不加锁*

            if (instance == null) {

*// 同步块，加锁*

                synchronized (*DoubleCheckSingleton*.class) {

*// 第二次检查，加锁*

                    if (instance == null) {

                        instance = new *DoubleCheckSingleton*();

                    }

                }

            }

            return instance;

        }

    }

*/\*\**

*\* 2. 改进的可变实例数目单例模式实现*

*\*/*

    public static class *LimitedInstanceSingleton* {

        private static final int MAX\_INSTANCES = 3; *// 最大实例数*

        private static final *LimitedInstanceSingleton*[] instances = new *LimitedInstanceSingleton*[MAX\_INSTANCES];

        private static int availableInstances = 0; *// 当前可用的实例数*

        private static final *Object* lock = new *Object*();

*// 私有构造函数*

        private *LimitedInstanceSingleton*() {

        }

*// 获取实例的公共方法*

        public static *LimitedInstanceSingleton* *getInstance*() {

            synchronized (lock) {

*// 如果还有可用的实例，直接返回*

                if (availableInstances < MAX\_INSTANCES) {

                    if (instances[availableInstances] == null) {

                        instances[availableInstances] = new *LimitedInstanceSingleton*();

                    }

                    return instances[availableInstances++];

                }

*// 如果没有可用实例，抛出异常*

                throw new *IllegalStateException*("已达到最大实例数限制");

            }

        }

*// 释放实例的方法*

        public static void *releaseInstance*(*LimitedInstanceSingleton* *instance*) {

            synchronized (lock) {

                for (int i = 0; i < availableInstances; i++) {

                    if (instances[i] == *instance*) {

*// 将实例移到最后*

*LimitedInstanceSingleton* temp = instances[i];

                        instances[i] = instances[availableInstances - 1];

                        instances[availableInstances - 1] = temp;

                        availableInstances--;

                        return;

                    }

                }

                throw new *IllegalArgumentException*("实例不存在");

            }

        }

*// 获取当前可用实例数*

        public static int *getAvailableInstances*() {

            return availableInstances;

        }

    }

*// 测试代码*

    public static void *main*(*String*[] *args*) {

*// 测试双重检查锁定单例*

*System*.out.*println*("测试双重检查锁定单例：");

*DoubleCheckSingleton* s1 = *DoubleCheckSingleton*.*getInstance*();

*DoubleCheckSingleton* s2 = *DoubleCheckSingleton*.*getInstance*();

*System*.out.*println*("s1 == s2: " + (s1 == s2));

*// 测试可变实例数目单例*

*System*.out.*println*("\n测试可变实例数目单例：");

        try {

*LimitedInstanceSingleton* i1 = *LimitedInstanceSingleton*.*getInstance*();

*LimitedInstanceSingleton* i2 = *LimitedInstanceSingleton*.*getInstance*();

*LimitedInstanceSingleton* i3 = *LimitedInstanceSingleton*.*getInstance*();

*System*.out.*println*("当前可用实例数: " + *LimitedInstanceSingleton*.*getAvailableInstances*());

*// 释放一个实例*

*LimitedInstanceSingleton*.*releaseInstance*(i2);

*System*.out.*println*("释放一个实例后可用实例数: " + *LimitedInstanceSingleton*.*getAvailableInstances*());

*// 再次获取实例*

*LimitedInstanceSingleton* i4 = *LimitedInstanceSingleton*.*getInstance*();

*System*.out.*println*("再次获取实例后可用实例数: " + *LimitedInstanceSingleton*.*getAvailableInstances*());

*// 尝试获取超过限制的实例*

*LimitedInstanceSingleton* i5 = *LimitedInstanceSingleton*.*getInstance*();

        } catch (*IllegalStateException* *e*) {

*System*.out.*println*("捕获到预期异常: " + e.*getMessage*());

        }

}

public class SingletonPatterns {

/\*\*

\* 1. 双重检查锁定的单例模式实现

\*/

public static class DoubleCheckSingleton {

// volatile关键字确保instance变量的可见性

private static volatile DoubleCheckSingleton instance;

// 私有构造函数防止外部实例化

private DoubleCheckSingleton() {

}

// 获取实例的公共方法

public static DoubleCheckSingleton getInstance() {

// 第一次检查，不加锁

if (instance == null) {

// 同步块，加锁

synchronized (DoubleCheckSingleton.class) {

// 第二次检查，加锁

if (instance == null) {

instance = new DoubleCheckSingleton();

}

}

}

return instance;

}

}

/\*\*

\* 2. 改进的可变实例数目单例模式实现

\*/

public static class LimitedInstanceSingleton {

private static final int MAX\_INSTANCES = 3; // 最大实例数

private static final LimitedInstanceSingleton[] instances = new LimitedInstanceSingleton[MAX\_INSTANCES];

private static int availableInstances = 0; // 当前可用的实例数

private static final Object lock = new Object();

// 私有构造函数

private LimitedInstanceSingleton() {

}

// 获取实例的公共方法

public static LimitedInstanceSingleton getInstance() {

synchronized (lock) {

// 如果还有可用的实例，直接返回

if (availableInstances < MAX\_INSTANCES) {

if (instances[availableInstances] == null) {

instances[availableInstances] = new LimitedInstanceSingleton();

}

return instances[availableInstances++];

}

// 如果没有可用实例，抛出异常

throw new IllegalStateException("已达到最大实例数限制");

}

}

// 释放实例的方法

public static void releaseInstance(LimitedInstanceSingleton instance) {

synchronized (lock) {

for (int i = 0; i < availableInstances; i++) {

if (instances[i] == instance) {

// 将实例移到最后

LimitedInstanceSingleton temp = instances[i];

instances[i] = instances[availableInstances - 1];

instances[availableInstances - 1] = temp;

availableInstances--;

return;

}

}

throw new IllegalArgumentException("实例不存在");

}

}

// 获取当前可用实例数

public static int getAvailableInstances() {

return availableInstances;

}

}

// 测试代码

public static void main(String[] args) {

// 测试双重检查锁定单例

System.out.println("测试双重检查锁定单例：");

DoubleCheckSingleton s1 = DoubleCheckSingleton.getInstance();

DoubleCheckSingleton s2 = DoubleCheckSingleton.getInstance();

System.out.println("s1 == s2: " + (s1 == s2));

// 测试可变实例数目单例

System.out.println("\n测试可变实例数目单例：");

try {

LimitedInstanceSingleton i1 = LimitedInstanceSingleton.getInstance();

LimitedInstanceSingleton i2 = LimitedInstanceSingleton.getInstance();

LimitedInstanceSingleton i3 = LimitedInstanceSingleton.getInstance();

System.out.println("当前可用实例数: " + LimitedInstanceSingleton.getAvailableInstances());

// 释放一个实例

LimitedInstanceSingleton.releaseInstance(i2);

System.out.println("释放一个实例后可用实例数: " + LimitedInstanceSingleton.getAvailableInstances());

// 再次获取实例

LimitedInstanceSingleton i4 = LimitedInstanceSingleton.getInstance();

System.out.println("再次获取实例后可用实例数: " + LimitedInstanceSingleton.getAvailableInstances());

// 尝试获取超过限制的实例

LimitedInstanceSingleton i5 = LimitedInstanceSingleton.getInstance();

} catch (IllegalStateException e) {

System.out.println("捕获到预期异常: " + e.getMessage());

}

}

}