02锁相关

由 黎金平创建, 最后修改于六月 13, 2019

两种锁机制对比

简要介绍

- **synchronized**关键字
- **java.util.concurrent.Lock**(Lock是一个接口，ReentrantLock是该接口一个很常用的实现)

这两种机制的底层原理存在一定的差别

- synchronized 关键字通过一对字节码指令 monitorenter/monitorexit 实现（同步语句块的情况）；和 ACC_SYNCHRONIZED 标识来实现；有兴趣的可以使用javap -c翻译看看
- java.util.concurrent.Lock 通过 Java 代码搭配sun.misc.Unsafe 中的本地调用实现的

Synchroinzed

三种使用场景

- **修饰实例方法，**作用于当前**对象实例加锁**，进入同步代码前要获得当前对象实例的锁

```java
public class SynchronizedTest implements Runnable {
    // 共享变量
    int count = 0;
    @Override
    public synchronized  void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + count++);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        SynchronizedTest synchronizedTest = new SynchronizedTest();
        // 此处实例锁就是synchronizedTest
        Thread thread1 = new Thread(synchronizedTest, name: "thread1");
        Thread thread2 = new Thread(synchronizedTest,  name: "thread2");
        thread1.start();
        thread2.start();
```

SynchronizedTest › main()

SynchronizedTest ×

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
thread1:0
thread1:1
thread1:2
thread1:3
thread1:4
thread2:5
thread2:6
thread2:7
thread2:8
thread2:9
```

thread1、thread2分别访问不同实例对象

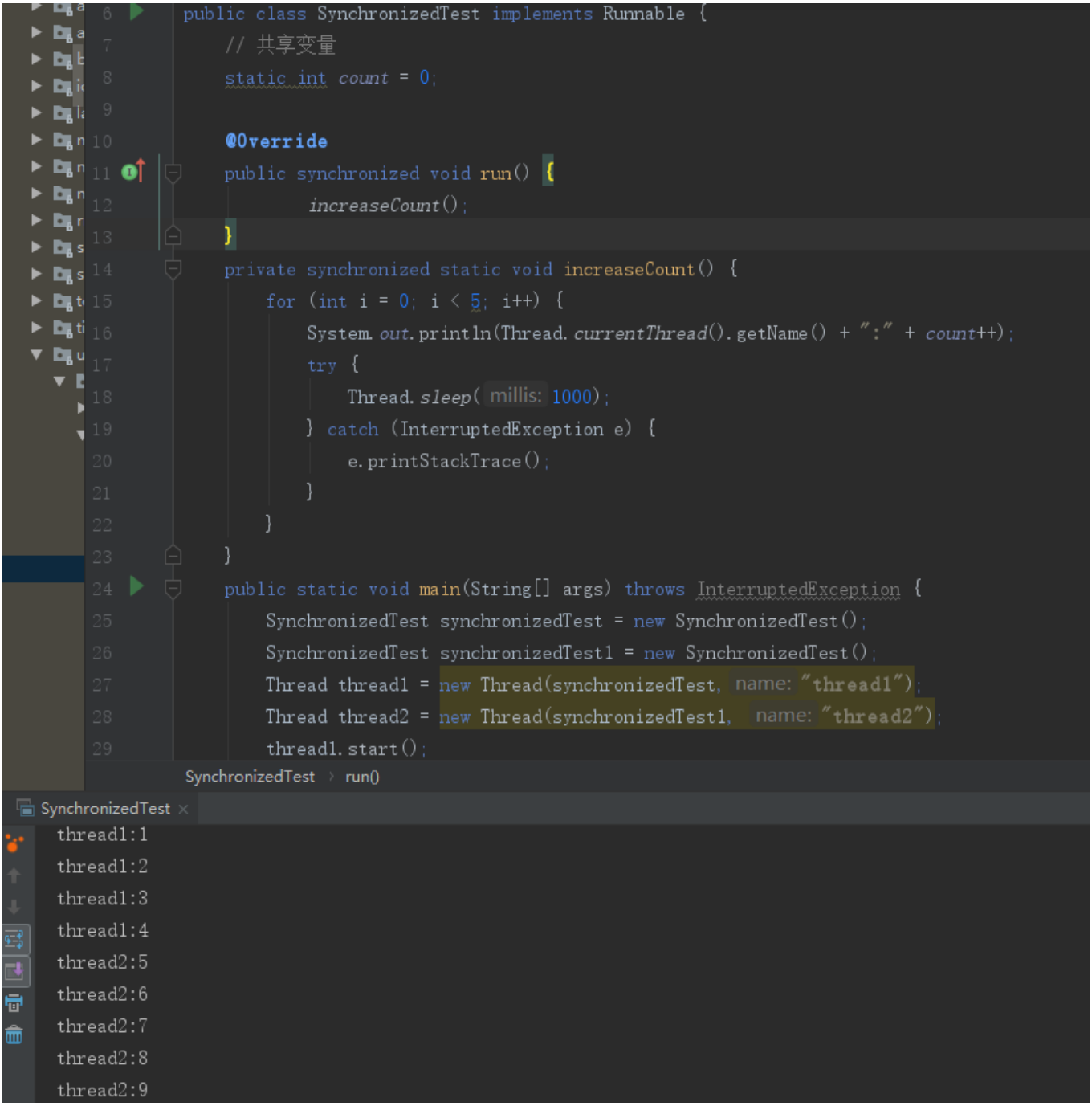
```java
public class SynchronizedTest implements Runnable {
    // 共享变量
    int count = 0;
    @Override
    public synchronized void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + count++);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        SynchronizedTest synchronizedTest = new SynchronizedTest();
        SynchronizedTest synchronizedTest1 = new SynchronizedTest();
        // 此处thread1实例锁就是synchronizedTest;thread2实例锁就是synchronizedTest1
        Thread thread1 = new Thread(synchronizedTest, name: "thread1");
        Thread thread2 = new Thread(synchronizedTest1, name: "thread2");
        thread1.start();
        thread2.start();
    }
}
```

SynchronizedTest › main()

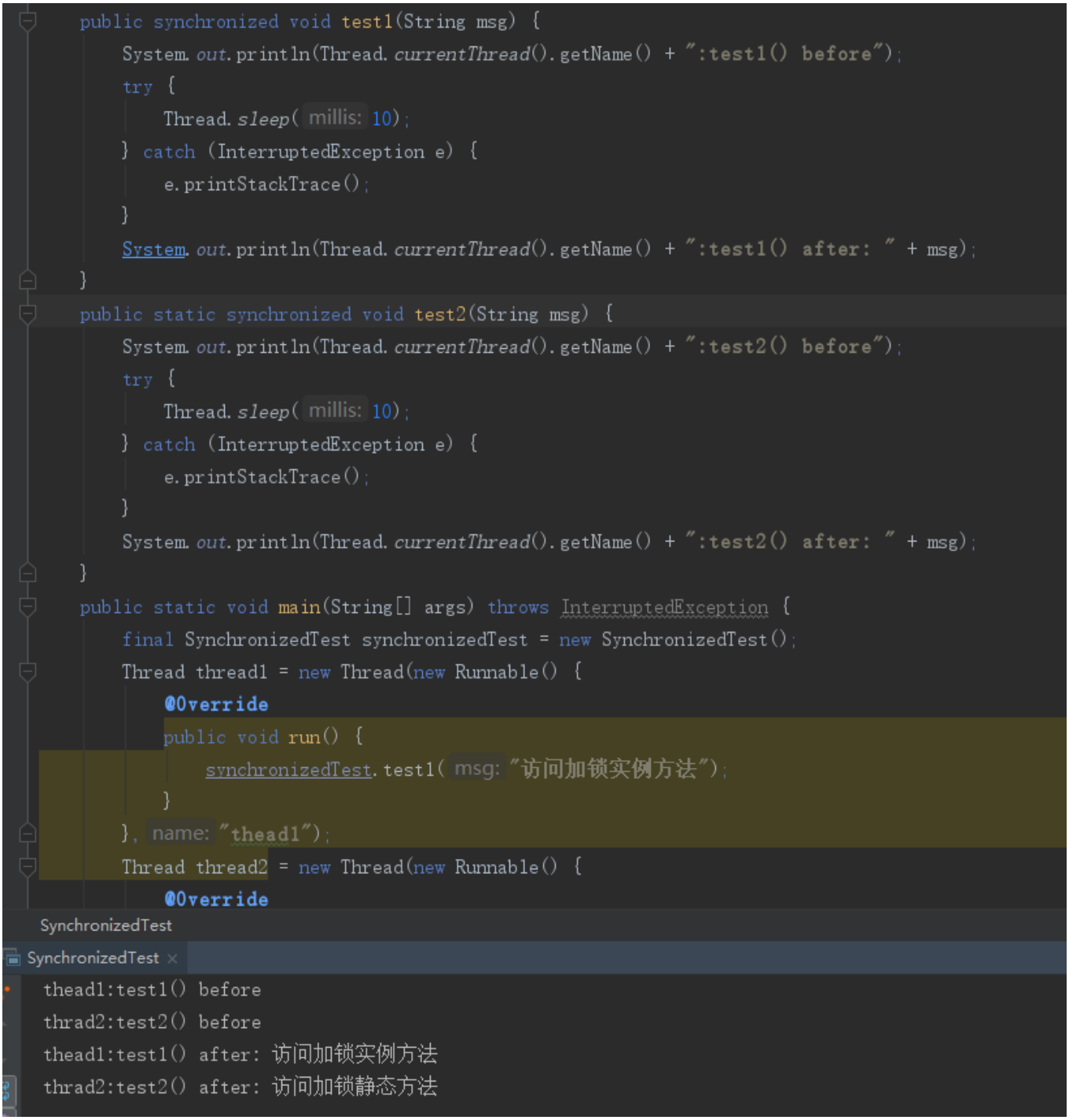
SynchronizedTest ×

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
thread2:0
thread1:0
thread2:1
thread1:1
thread2:2
thread1:2
thread2:3
thread1:3
thread2:4
thread1:4
```

- **修饰静态方法，**作用于**当前类对象加锁，**进入同步代码前要获得当前类对象的锁 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。**

- 

- 一个线程获取到一个实例中的加锁方法的锁时，另一个线程依然可以访问静态加锁方法（即实例的锁与静态方法的锁是不同的，两者不影响）



- **修饰代码块**，指定加锁对象，对**给定对象加锁**，进入同步代码库前要获得给定对象的锁。 和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓冲功能！

```java
// 共享变量
static int count = 0;
@Override
public synchronized void run() {
    increaseCount();
}
private void increaseCount() {
    //锁定当前方法实例对象
    synchronized (this) {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + ":" + count++);
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public static void main(String[] args) throws InterruptedException {
    SynchronizedTest synchronizedTest = new SynchronizedTest();
    SynchronizedTest synchronizedTest1 = new SynchronizedTest();
    Thread thread1 = new Thread(synchronizedTest, name: "thread1");
    Thread thread2 = new Thread(synchronizedTest1, name: "thread2");
    thread1.start();
    thread2.start();
```

SynchronizedTest › increaseCount()

SynchronizedTest

```
thread1:0
thread2:1
thread2:2
thread1:3
thread1:4
thread2:4
thread1:5
```

-

sychronized里面可以是任意对象

```
    // 共享变量
    static int count = 0;
    private byte[] mBytes = new byte[0];
    @Override
    public synchronized void run() {
        increaseCount();
    }
    private void increaseCount() {
        //锁定当前方法实例对象
        synchronized (mBytes) {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + ":" + count++);
                try {
                    Thread.sleep( millis: 1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    public static void main(String[] args) throws InterruptedException {
        SynchronizedTest synchronizedTest = new SynchronizedTest();
        SynchronizedTest synchronizedTest1 = new SynchronizedTest();
        Thread thread1 = new Thread(synchronizedTest, name: "thread1");
        Thread thread2 = new Thread(synchronizedTest1, name: "thread2");
        thread1.start();
```

```
SynchronizedTest › main()
SynchronizedTest ×
thread2:6
thread1:7
thread2:8
thread1:8
```

ReentrantLock

特性(对比Synchroinzed)

- 尝试获得锁
- 获取到锁的线程能够响应中断

ReentrantLock(重入锁)

```
public class MyService {

    private Lock lock = new ReentrantLock();

    public void testMethod() {
        lock.lock();
        for (int i = 0; i < 5; i++) {
            System.out.println("ThreadName=" + Thread.currentThread().getName()
                    + (" " + (i + 1)));
        }
        lock.unlock();
    }

}
```

效果和synchronized一样，都可以同步执行，**lock方法获得锁，unlock方法释放锁**

await方法

```
public class MyService {

    private Lock lock = new ReentrantLock();
    private Condition condition=lock.newCondition();
    public void testMethod() {

        try {
            lock.lock();
            System.out.println("开始wait");
            condition.await();
            for (int i = 0; i < 5; i++) {
                System.out.println("ThreadName=" + Thread.currentThread().getName()
                        + (" " + (i + 1)));
            }
        } catch (InterruptedException e) {
            // TODO 自动生成的 catch 块
            e.printStackTrace();
        }
        finally
        {
            lock.unlock();
        }
    }

}
```

通过创建Condition对象来使线程wait，必须先执行lock.lock方法获得锁

signal方法

```
public void signal()
    {
        try
        {
            lock.lock();
            condition.signal();
        }
        finally
        {
            lock.unlock();
        }
    }
```

condition对象的signal方法可以唤醒wait线程

对比

① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

举例：

```
public synchronized void testA() throws InterruptedException {
    Thread.sleep( millis: 1000);
    testB();
}

public synchronized void testB() throws InterruptedException {
    Thread.sleep( millis: 1000);
}
```

果synchronized不是可重入锁的话，testB就不会被当前线程执行，从而形成死锁；可重入锁的一个好处是可一定程度避免死锁

② synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReentrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReentrantLock增加了一些高级功能。主要来说主要有三点：**①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**

- ReentrantLock提供了一种能够中断等待锁的线程的机制，通过**lock.lockInterruptibly()**来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。
- ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。 ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的`ReentrantLock(boolean fair)`构造方法来制定是否是公平的。
- synchronized关键字与**wait()和notify()/notifyAll()**方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于**Condition接口与newCondition()** 方法；

  1.在使用notify()/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，

2.用ReentrantLock类结合Condition实例可以实现“选择性通知”，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

锁的种类

独享锁/共享锁

独享锁是指该锁一次只能被一个线程所持有----ReentrantLock、 Synchronized

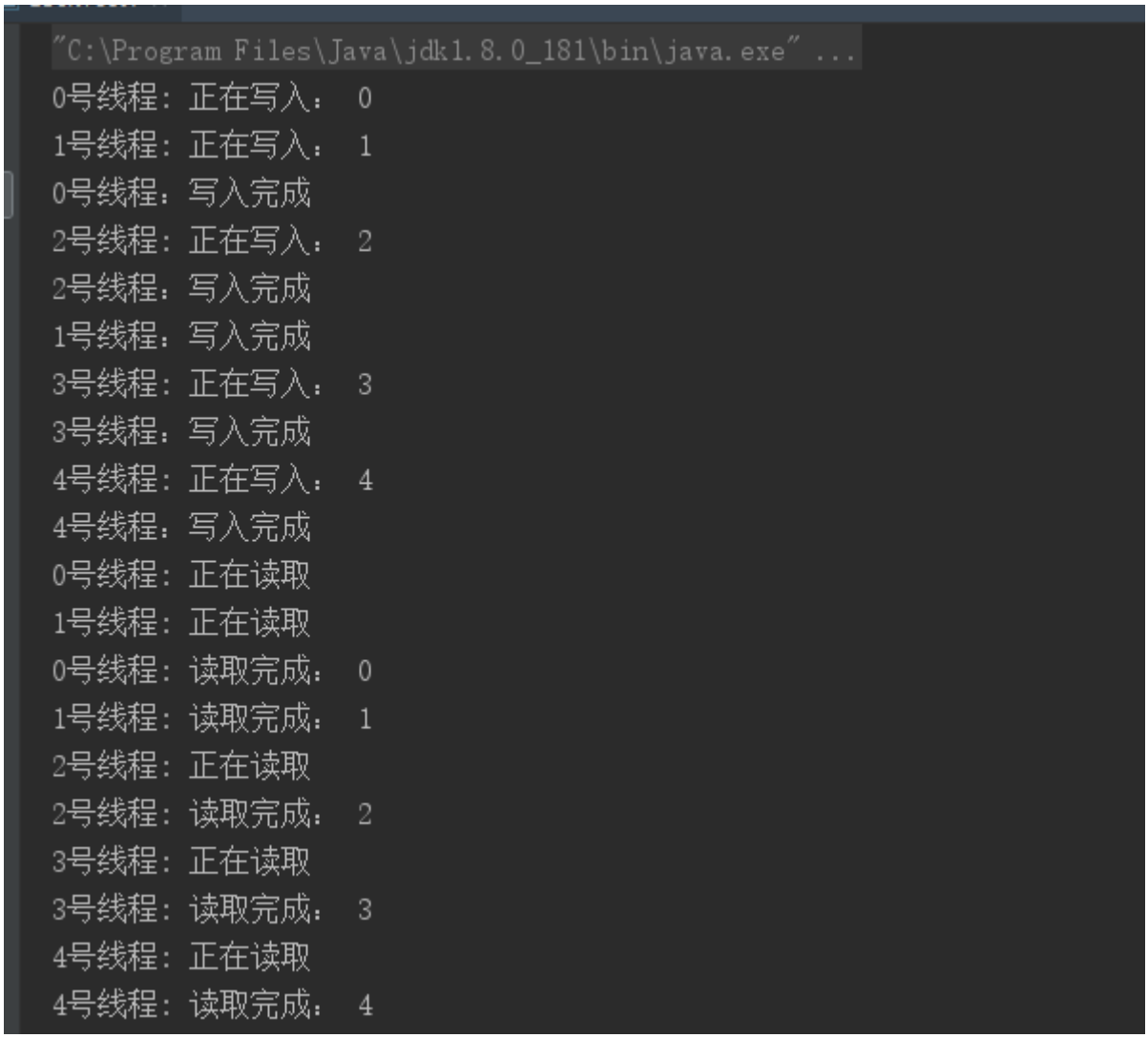
共享锁是指该锁可被多个线程所持有

ReentrantReadWriteLock的读锁是共享锁，写锁是独占锁。这样可以保证读操作的并发性。读&读可以共享锁，读&写不能共存，写&写也不能共存。

demo01 折叠源码

```
/**
 * @description:
 * @author:jinping.li(jinping.li@ucarinc.com)
 * @date: 2019-06-13 14:36
 */
public class LockTest1 {
    private volatile Map<String, Object> map = new HashMap<>();

    public void put(String key, Object value){
        System.out.println(Thread.currentThread().getName() + ": 正在写入：  "+ value );
        map.put(key, value);
        System.out.println(Thread.currentThread().getName() + ": 写入完成");
    }

    public void get(String key){
        System.out.println(Thread.currentThread().getName() + ": 正在读取");
        Object result = map.get(key);
        System.out.println(Thread.currentThread().getName() + ": 读取完成：  " + result
    }
    public static void main(String[] args){
        final LockTest1 myCache = new LockTest1();
        for (int i=0; i<5; i++){
            final Integer temp = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    myCache.put(temp+"", temp);
                }
            }, temp + "号线程").start();
        }

        for (int i=0; i<5; i++){
            final Integer temp = i;
            new Thread(new Runnable() {
                @Override
```

```
                public void run() {
                    myCache.get(temp+"");
                }
            }, temp + "号线程").start();
        }
    }

}
```

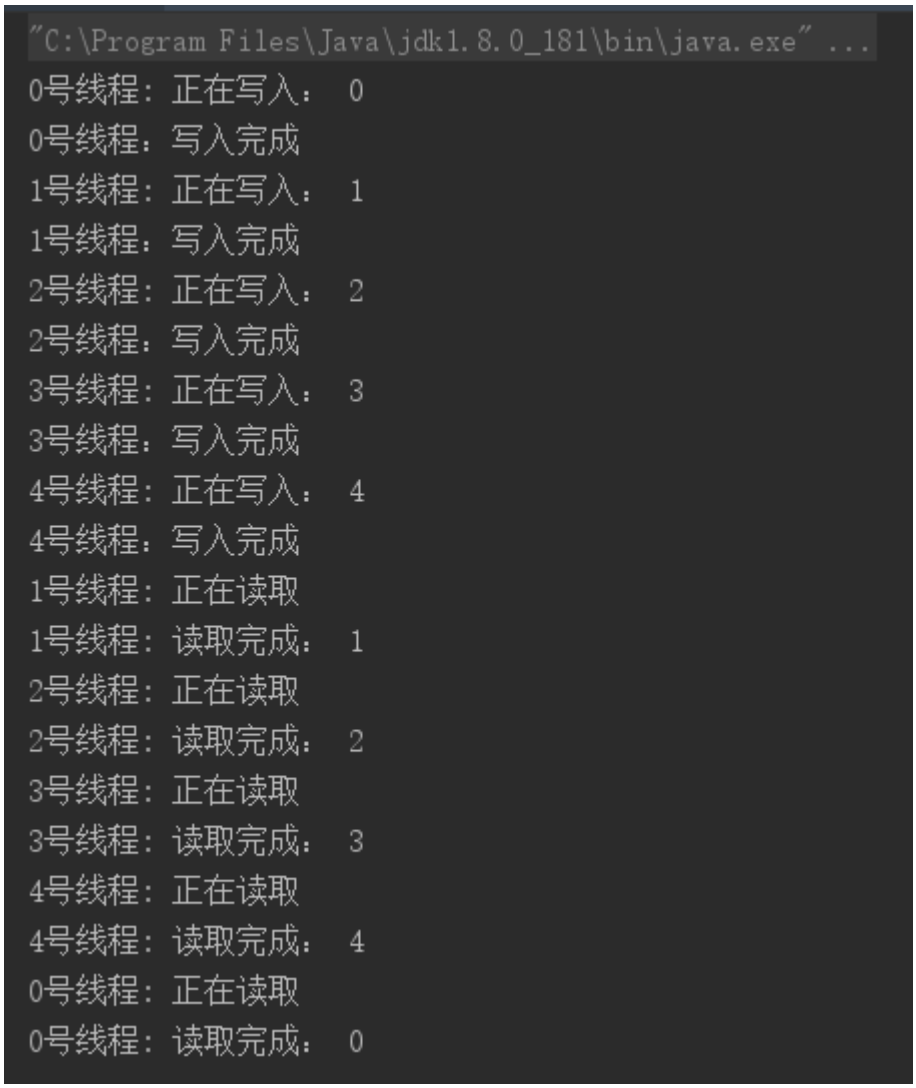


可以看到，0号线程还没写入完成，1号线程进来了，1号线程还没完成，2号线程又进来了。这就没有保证写操作是独占的，没有保证写操作的原子性。看看使用ReentrantReadWriteLock如何解决

demo02 折叠源码

```
public class LockTest1 {
    private volatile Map<String, Object> map = new HashMap<>();

    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    public void put(String key, Object value){
        lock.writeLock().lock();
        System.out.println(Thread.currentThread().getName() + ": 正在写入:  "+ value );
        map.put(key, value);
        System.out.println(Thread.currentThread().getName() + ": 写入完成");
        lock.writeLock().unlock();
    }
```

```
    public void get(String key){
        lock.readLock().lock();
        System.out.println(Thread.currentThread().getName() + ": 正在读取");
        Object result = map.get(key);
        System.out.println(Thread.currentThread().getName() + ": 读取完成:  " + result
        lock.readLock().unlock();
    }
    public static void main(String[] args){
        final LockTest1 myCache = new LockTest1();
        for (int i=0; i<5; i++){
            final Integer temp = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    myCache.put(temp+"", temp);
                }
            }, temp + "号线程").start();
        }

        for (int i=0; i<5; i++){
            final Integer temp = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    myCache.get(temp+"");
                }
            }, temp + "号线程").start();
        }
    }

}
```



可以看到，写操作的时候，保证了每次写操作的原子性，是独占锁；读的时候可以多个线程同时访问，是共享锁

可重入锁

自己可以再次获取自己的内部锁

自旋锁

一句话说明：采用循环的方式去尝试获取锁

概念：是一种乐观锁的优化；竞争锁的失败的线程，并不会真实的在操作系统层面挂起等待，而是JVM会让当前线程不停地的在循环体内执行实现的（基于预测在不久的将来就能获得)；当循环的条件被其他线程改变时才能进入临界区(已获得锁)，如果还不能获得锁，才会真实的将线程在操作系统层面进行挂起(升级为重量级锁)

比喻：一辆车遇到红灯，一看还有5秒就变绿了，那就不熄火等一下。发动机类比CPU，先占一会，就能获取锁（绿灯），如果要等5分钟，就熄火等待，时间到了再打火成本也合算

好处、坏处：好处就是减少上下文切换的消耗，但是比较消耗CPU资源

无标签