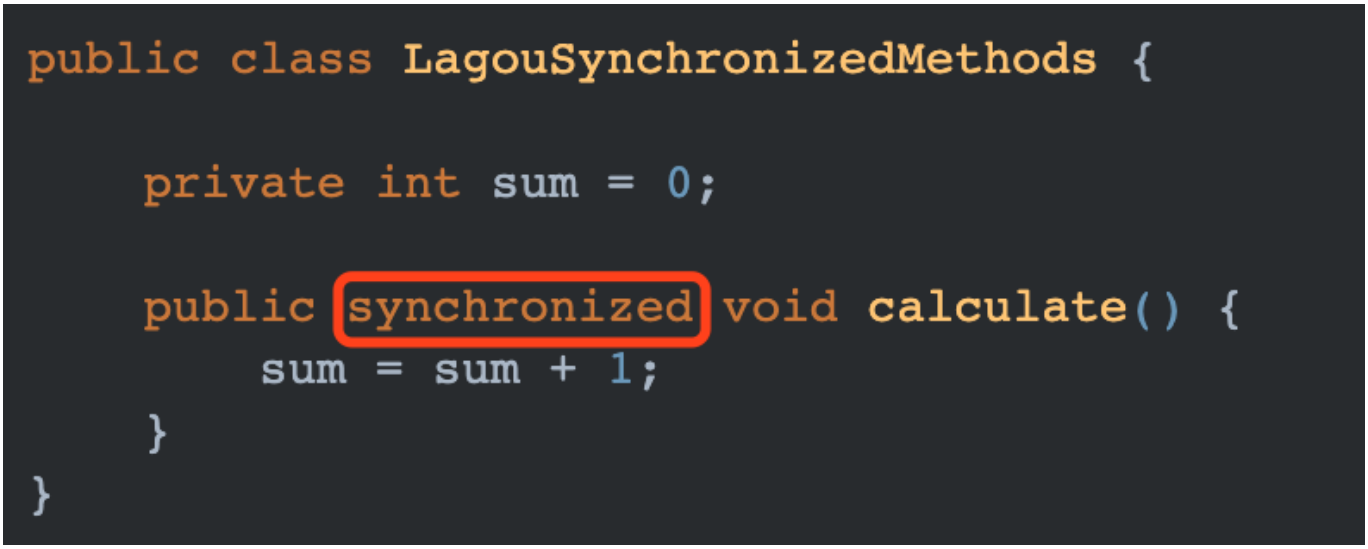
synchronized关键字相信每一位Java工程师都不会陌生。而ReentrantLock作为大神DougLea编写的concurrent包中的重要一员，也在众多项目中发挥重要重要功能。因为这两者实在是太重要，所以专门使用一课时的内容来对它们做一个详细的比较。后续两课时将会着重介绍它们各自的实现细节与原理。

1. synchronized

synchronized 可以用来修饰以下 3 个层面：

1. 修饰实例方法
2. 修饰静态类方法
3. 修饰代码块

1.1 synchronized 修饰实例方法



这种情况下的 **锁对象** 是 **当前实例对象**，因此只有同一个实例对象调用此方法才会产生互斥效果，**不同实例对象之间不会有互斥效果**。比如如下代码：

```java
public class LagouSynchronizedMehtods{
    public static void main(String[] args){
        LagouSynchronizedMehtods l1 = new LagouSynchronizedMehtods();
        LagouSynchronizedMehtods l2 = new LagouSynchronizedMehtods();

        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run(){
                l2.printLog();
            }
        });

        t1.start();
        t2.start();
    }

    public void printLog(){
        try{
            for(int i = 0; i < 5; i ++) {
                System.out.println(Thread.currentThread().getName() + " is printing " + i);
                Thread.sleep(300);
            }
        } catch(Exception e){}
    }
}
```

上述代码，在不同的线程中调用的是不同对象的 printLog 方法，因此彼此之间不会有排斥。运行效果如下：

```
➜  danny_folder java LagouSynchronizedMehtods
Thread-0 is printing 0
Thread-1 is printing 0
Thread-1 is printing 1
Thread-0 is printing 1
Thread-1 is printing 2
Thread-0 is printing 2
Thread-1 is printing 3
Thread-0 is printing 3
Thread-0 is printing 4
Thread-1 is printing 4
➜  danny_folder
```

可以看出，两个线程是 **交互执行** 的。

如果将代码进行如下修改，两个线程调用同一个对象的 printLog 方法：

```java
public class LagouSynchronizedMehtods{
    public static void main(String[] args){
        LagouSynchronizedMehtods l1 = new LagouSynchronizedMehtods();

        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });

        t1.start();
        t2.start();
    }

    public synchronized void printLog(){
        for(int i = 0; i < 5; i ++) {
            System.out.println(Thread.currentThread().getName() + " is printing " + i);
        }
    }
}
```

具体的执行结果如下：

```
➜  danny_folder java LagouSynchronizedMehtods
Thread-0 is printing 0
Thread-0 is printing 1
Thread-0 is printing 2
Thread-0 is printing 3
Thread-0 is printing 4
Thread-1 is printing 0
Thread-1 is printing 1
Thread-1 is printing 2
Thread-1 is printing 3
Thread-1 is printing 4
➜  danny_folder
```

可以看出：只有某一个线程中的代码执行完之后，才会调用另一个线程中的代码。也就是说此时 **两个线程间是互斥** 的。

1.2 修饰静态类方法

如果 synchronized 修饰的是静态方法，则 **锁对象** 是 **当前类的 Class 对象**。因此即使 在不同线程中调用不同实例对象，也会有 **互斥效果**。

将 LagouSynchronizedMehtods 中的 printLog 修改为静态方法，如下：

```
public class LagouSynchronizedMehtods{
    public static void main(String[] args){
        LagouSynchronizedMehtods l1 = new LagouSynchronizedMehtods();
        LagouSynchronizedMehtods l2 = new LagouSynchronizedMehtods();

        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run(){
                l2.printLog();
            }
        });

        t1.start();
        t2.start();
    }

    public static synchronized void printLog(){
        for(int i = 0; i < 5; i ++) {
            System.out.println(Thread.currentThread().getName() + " is printing " + i);
        }
    }
}
```

执行后的打印效果如下：

```
➜  danny_folder java LagouSynchronizedMehtods
Thread-0 is printing 0
Thread-0 is printing 1
Thread-0 is printing 2
Thread-0 is printing 3
Thread-0 is printing 4
Thread-1 is printing 0
Thread-1 is printing 1
Thread-1 is printing 2
Thread-1 is printing 3
Thread-1 is printing 4
➜  danny_folder
```

可以看出，两个线程还是依次执行的。

1.3 synchronized 修饰代码块

除了直接修饰方法之外，synchronized 还可以作用于代码块，如下代码所示：

```
public class LagouSynchronizedMehtods{
    private Object lock = new Object();

    public static void main(String[] args){
        LagouSynchronizedMehtods l1 = new LagouSynchronizedMehtods();

        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });

        t1.start();
        t2.start();
    }

    public  void printLog(){
        synchronized(lock) {
            for(int i = 0; i < 5; i ++) {
                System.out.println(Thread.currentThread().getName() + " is printing " + i);
            }
        }
    }
}
```

synchronized 作用于代码块时，锁对象就是跟在后面括号中的对象。上图中可以看出任何 Object 对象都可以当作锁对象。

1.4 实现细节

synchronized 既可以作用于方法，也可以作用于某一代码块。但在实现上是有区别的。 比如如下代码，使用 synchronized 作用于代码块：

```
1  public class Foo{
2      private int number;
3      public void test1() {
4          int i = 0;
5          synchronized(this){
6              number = i + 1;
7          }
8      }
9  }
```

使用 javap 查看上述 test1 方法的字节码，可以看出，编译而成的字节码中会包含 `monitorenter` 和 `monitorexit` 这两个字节码指令。如下所示：

```
public void test1();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=4, args_size=1
         0: iconst_0
         1: istore_1
         2: aload_0
         3: dup
         4: astore_2
         5: monitorenter
         6: iload_1
         7: iconst_1
         8: iadd
         9: istore_1
        10: aload_2
        11: monitorexit
        12: goto          20
        15: astore_3
        16: aload_2
        17: monitorexit
        18: aload_3
        19: athrow
        20: return
```

你可能已经发现了，上面字节码中有1个monitorenter和2个monitorexit。这是因为虚拟机需要保证当异常发生时也能释放锁。因此 2 个 monitorexit 一个是代码正常执行结束后释放锁，一个是在代码执行异常时释放锁。

再看下 synchronized 修饰方法有哪些区别：

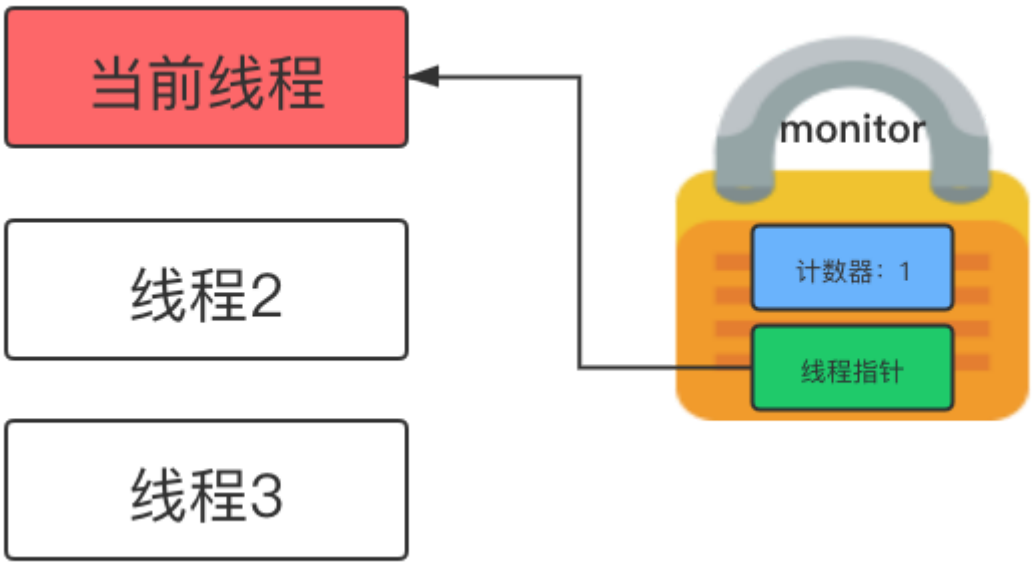
```
public synchronized void test1() {
        int i = 0;
            i = i + 1;
}

使用javap查看上面方法编译后的字节码

public synchronized void test1();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED
    Code:
      stack=2, locals=2, args_size=1
         0: iconst_0
         1: istore_1
         2: iload_1
         3: iconst_1
         4: iadd
         5: istore_1
         6: return
```

从图中可以看出，被 synchronized 修饰的方法在被编译为字节码后，在方法的 flags 属性中会被标记为 ACC_SYNCHRONIZED 标志。当虚拟机访问一个被标记为 ACC_SYNCHRONIZED 的方法时，会 **自动在方法的开始和结束（或异常）位置添加 monitorenter 和 monitorexit 指令**。

关于 monitorenter 和 monitorexit，可以理解为一把具体的锁。在这个锁中保存着两个比较重要的属性：计数器和指针。

- 计数器代表当前线程一共访问了几次这把锁；
- 指针指向持有这把锁的线程。

用一张图表示如下：



锁计数器默认为0，当执行monitorenter指令时，如锁计数器值为0 说明这把锁并没有被其它线程持有。那么这个线程会将计数器加1，并将锁中的指针指向自己。当执行monitorexit指令时，会将计数器减1。

2. ReentrantLock

2.1 ReentrantLock 基本使用

ReentrantLock 的使用同 synchronized 有点不同，它的 **加锁和解锁操作都需要手动完成**，如下所示：

```
public class LagouReentrantLockTest {

    ReentrantLock lock = new ReentrantLock();

    public static void main(String[] args){
        LagouReentrantLockTest l1 = new LagouReentrantLockTest();

        Thread t1 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });
        Thread t2 = new Thread(new Runnable(){
            @Override
            public void run(){
                l1.printLog();
            }
        });

        t1.start();
        t2.start();
    }

    public void printLog(){
        try{
            lock.lock();
            for(int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + " is printing " + i);
            }
        }catch(Exception e){
        } finally{
            lock.unlock();
        }
    }
}
```

图中 lock() 和 unlock() 分别是加锁和解锁操作。运行效果如下：

```
➜  danny_folder java LagouReentrantLockTest
Thread-0 is printing 0
Thread-0 is printing 1
Thread-0 is printing 2
Thread-0 is printing 3
Thread-0 is printing 4
Thread-1 is printing 0
Thread-1 is printing 1
Thread-1 is printing 2
Thread-1 is printing 3
Thread-1 is printing 4
➜  danny_folder
```

可以看出，使用 ReentrantLock 也能实现同 synchronized 相同的效果。

你可能已经注意到，在上面 ReentrantLock 的使用中，我将 `unlock` 操作放在 finally 代码块中。这是因为 ReentrantLock 与 synchronized 不同，当异常发生时 synchronized 会自动释放锁，但是 ReentrantLock 并不会自动释放锁。因此好的方式是将 `unlock` 操作放在 finally 代码块中，保证任何时候锁都能够被正常释放掉。

2.2 公平锁实现

ReentrantLock 有一个带参数的构造器，如下：

```
/**
 * Creates an instance of {@code ReentrantLock} with the
 * given fairness policy.
 *
 * @param fair {@code true} if this lock should use a fair ordering policy
 */
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

所谓 **公平锁** 就是通过 **同步队列** 来实现多个线程按照 **申请锁的顺序** 获取锁。

默认情况下，synchronized和ReentrantLock都是 **非公平锁**。但是 ReentrantLock 可以通过传入 true **来创建一个公平锁。**

公平锁使用实例如下：

```
public class LagouFairLockTest implements Runnable{
    private int sharedNumber = 0;
    //创建公平锁
    private static ReentrantLock lock=new ReentrantLock(true);
    public void run() {
        while(sharedNumber < 20){
            lock.lock();
            try{
                sharedNumber++;
                System.out.println(Thread.currentThread().getName()
                        + " 获得锁, sharedNumber is " + sharedNumber);
            }finally{
                lock.unlock();
            }
        }
    }
    public static void main(String[] args) {
        LagouFairLockTest lft = new LagouFairLockTest();
        Thread th1=new Thread(lft);
        Thread th2=new Thread(lft);
        Thread th3=new Thread(lft);
        th1.start();
        th2.start();
        th3.start();
    }
}
```

运行效果如下：

```
→  danny_folder
→  danny_folder java LagouFairLockTest
Thread-0 获得锁, sharedNumber is 1
Thread-1 获得锁, sharedNumber is 2
Thread-2 获得锁, sharedNumber is 3
Thread-0 获得锁, sharedNumber is 4
Thread-1 获得锁, sharedNumber is 5
Thread-2 获得锁, sharedNumber is 6
Thread-0 获得锁, sharedNumber is 7
Thread-1 获得锁, sharedNumber is 8
Thread-2 获得锁, sharedNumber is 9
Thread-0 获得锁, sharedNumber is 10
Thread-1 获得锁, sharedNumber is 11
Thread-2 获得锁, sharedNumber is 12
Thread-0 获得锁, sharedNumber is 13
Thread-1 获得锁, sharedNumber is 14
Thread-2 获得锁, sharedNumber is 15
Thread-0 获得锁, sharedNumber is 16
Thread-1 获得锁, sharedNumber is 17
Thread-2 获得锁, sharedNumber is 18
Thread-0 获得锁, sharedNumber is 19
Thread-1 获得锁, sharedNumber is 20
Thread-2 获得锁, sharedNumber is 21
Thread-0 获得锁, sharedNumber is 22
→  danny_folder
```

网上对公平锁有一段例子很经典：假设有一口水井，有管理员看守，管理员有一把锁，只有拿到锁的人才能够打水，打完水要把锁还给管理员。每个过来打水的人都要得到管理员的允许并拿到锁之后才能去打水，如果前面有人正在打水，那么这个想要打水的人就必须排队。管理员会查看下一个要去打水的人是不是队伍里排最前面的人，如果是的话，才会给你锁让你去打水；如果你不是排第一的人，就必须去队尾排队，这就是公平锁。

3. 读写锁（ReentrantReadWriteLock）

在常见的开发中，我们经常会定义一个线程间共享的用作缓存的数据结构，比如一个较大的Map。缓存中保存了全部的城市Id和城市name对应关系。这个大 Map 绝大部分时间提供读服务（根据城市 Id 查询城市名称等）。而写操作占有的时间很少，通常是在服务启动时初始化，然后可以每隔一定时间再刷新缓存的数据。但是 **写操作开始到结束之间，不能再有其他读操作进来**，并且写操作完成之后的更新数据需要对 **后续的读操作可见**。

在没有读写锁支持的时候，如果想要完成上述工作就需要使用 Java 的等待通知机制，就是当写操作开始时，所有晚于写操作的读操作均会进入等待状态，只有写操作完成并进行通知之后，所有等待的读操作才能继续执行。这样做的目的是使读操作能读取到正确的数据，不会出现脏读。

但是如果使用 concurrent 包中的读写锁（ReentrantReadWriteLock）实现上述功能，**就只需要在读操作时获取读锁，写操作时获取写锁即可**。当 **写锁被获取到时**，后续的 **读写锁都会被阻塞**，写锁释放之后，所有操作继续执行，这种编程方式相对于使用等待通知机制的实现方式而言，变得简单明了。

接下来，我们看下读写锁（ReentrantReadWriteLock）如何使用：

1. 创建读写锁对象：

```
ReadWriteLock rwLock = newReentrantReadWriteLock();
```

2. 通过读写锁对象分别获取读锁（ReadLock）和写锁（WriteLock）：

```
ReentrantReadWriteLock.ReadLock readLock = rwLock.readLock();
ReentrantReadWriteLock.WriteLock writeLock = rwLock.writeLock();
```

3. 使用读锁（ReadLock）同步缓存的读操作，使用写锁（WriteLock）同步缓存的写操作：

```
// 读操作
readLock.lock();
try {
    // 从缓存中读取数据
} finally {
    readLock.unlock();
}

// 写操作
writeLock.lock();
try {
    // 向缓存中写入数据
} finally {
    writeLock.unlock();
}
```

具体实现，参考如下代码片段：

```java
public class ReadWriteTest {
    private static final ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);
    private static String number = "0";

    public static void main(String[] args) throws InterruptedException{
        Thread t1 = new Thread(new Reader(),"读线程 1");
        Thread t2 = new Thread(new Reader(),"读线程 2");  1
        Thread t3 = new Thread(new Writer(),"写线程");
        t1.start();
        t2.start();
        t3.start();
    }
    static class Reader implements Runnable {
        public void run() {
            for(int i = 0; i<= 10; i ++) {
                lock.readLock().lock();  2
                System.out.println(Thread.currentThread().getName() + " ---> Number is " + number);
                lock.readLock().unlock();
            }
        }
    }
    static class Writer implements Runnable {
        public void run() {
            for(int i = 1; i<= 7; i+=2) {
                try {
                    lock.writeLock().lock();  3
                    System.out.println(Thread.currentThread().getName()  + " 正在写入 " + i);
                    number = number.concat("" + i);
                } finally {
                    lock.writeLock().unlock();
                }
            }
        }
    }
}
```

解释说明：

- 图中的 number 是线程中共享的数据，用来模拟缓存数据；
- 图中①处，分别创建 2 个 Reader 线程并从缓存中读取数据，和 1 个 Writer 将数据写入缓存中；
- 图中②处，使用读锁（ReadLock）将读取数据的操作加锁；
- 图中③处，使用写锁（WriteLock）将写入数据到缓存中的操作加锁。

```
➜  danny_folder java ReentrantReadWriteLockExampleMain
读线程 2 ---> Number is 0
读线程 1 ---> Number is 0
写线程 正在写入 1
读线程 2 ---> Number is 01
读线程 1 ---> Number is 01
写线程 正在写入 3
读线程 2 ---> Number is 013
读线程 1 ---> Number is 013
写线程 正在写入 5
读线程 2 ---> Number is 0135
读线程 1 ---> Number is 0135
写线程 正在写入 7
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
读线程 1 ---> Number is 01357
读线程 2 ---> Number is 01357
```

仔细查看日志，可以看出当写入操作在执行时，读取数据的操作会被阻塞。当写入操作执行成功后，读取数据的操作继续执行，并且读取的数据也是最新写入后的实时数据。

总结

这课时我们主要学习了Java中两个实现同步的方式synchronized和ReentrantLock。其中synchronized使用更简单，加锁和释放锁都是由虚拟机自动完成，而 ReentrantLock 需要开发者手动去完成。但是很显然 ReentrantLock 的使用场景更多，公平锁还有读写锁都可以在复杂场景中发挥重要作用。