第15讲 | synchronized和ReentrantLock有什么区别呢？

2018-06-07 杨晓峰

第15讲 | synchronized和ReentrantLock有什么区别呢？
朗读人：黄洲君　09′36″ | 4.40M

从今天开始，我们将进入 Java 并发学习阶段。软件并发已经成为现代软件开发的基础能力，而 Java 精心设计的高效并发机制，正是构建大规模应用的基础之一，所以考察并发基本功也成为各个公司面试 Java 工程师的必选项。

今天我要问你的问题是， synchronized 和 ReentrantLock 有什么区别？有人说 synchronized 最慢，这话靠谱吗？

典型回答

synchronized 是 Java 内建的同步机制，所以也有人称其为 Intrinsic Locking，它提供了互斥的语义和可见性，当一个线程已经获取当前锁时，其他试图获取的线程只能等待或者阻塞在那里。

在 Java 5 以前，synchronized 是仅有的同步手段，在代码中， synchronized 可以用来修饰方法，也可以使用在特定的代码块儿上，本质上 synchronized 方法等同于把方法全部语句用 synchronized 块包起来。

ReentrantLock，通常翻译为再入锁，是 Java 5 提供的锁实现，它的语义和 synchronized 基本相

同。再入锁通过代码直接调用 lock() 方法获取，代码书写也更加灵活。与此同时，ReentrantLock 提供了很多实用的方法，能够实现很多 synchronized 无法做到的细节控制，比如可以控制 fairness，也就是公平性，或者利用定义条件等。但是，编码中也需要注意，必须要明确调用 unlock() 方法释放，不然就会一直持有该锁。

synchronized 和 ReentrantLock 的性能不能一概而论，早期版本 synchronized 在很多场景下性能相差较大，在后续版本进行了较多改进，在低竞争场景中表现可能优于 ReentrantLock。

考点分析

今天的题目是考察并发编程的常见基础题，我给出的典型回答算是一个相对全面的总结。

对于并发编程，不同公司或者面试官面试风格也不一样，有个别大厂喜欢一直追问你相关机制的扩展或者底层，有的喜欢从实用角度出发，所以你在准备并发编程方面需要一定的耐心。

我认为，锁作为并发的基础工具之一，你至少需要掌握：

- 理解什么是线程安全。
- synchronized、ReentrantLock 等机制的基本使用与案例。

更近一步，你还需要：

- 掌握 synchronized、ReentrantLock 底层实现；理解锁膨胀、降级；理解偏斜锁、自旋锁、轻量级锁、重量级锁等概念。
- 掌握并发包中 java.util.concurrent.lock 各种不同实现和案例分析。

知识扩展

专栏前面几期穿插了一些并发的概念，有同学反馈理解起来有点困难，尤其对一些并发相关概念比较陌生，所以在这一讲，我也对会一些基础的概念进行补充。

首先，我们需要理解什么是线程安全。

我建议阅读 Brain Goetz 等专家撰写的《Java 并发编程实战》（Java Concurrency in Practice），虽然可能稍显学究，但不可否认这是一本非常系统和全面的 Java 并发编程书籍。按照其中的定义，线程安全是一个多线程环境下正确性的概念，也就是保证多线程环境下**共享的**、**可修改的**状态的正确性，这里的状态反映在程序中其实可以看作是数据。

换个角度来看，如果状态不是共享的，或者不是可修改的，也就不存在线程安全问题，进而可以推理出保证线程安全的两个办法：

- 封装：通过封装，我们可以将对象内部状态隐藏、保护起来。

- 不可变：还记得我们在专栏第 3 讲强调的 final 和 immutable 吗，就是这个道理，Java 语言目前还没有真正意义上的原生不可变，但是未来也许会引入。

线程安全需要保证几个基本特性：

- 原子性，简单说就是相关操作不会中途被其他线程干扰，一般通过同步机制实现。
- 可见性，是一个线程修改了某个共享变量，其状态能够立即被其他线程知晓，通常被解释为将线程本地状态反映到主内存上，volatile 就是负责保证可见性的。
- 有序性，是保证线程内串行语义，避免指令重排等。

可能有点晦涩，那么我们看看下面的代码段，分析一下原子性需求体现在哪里。这个例子通过取两次数值然后进行对比，来模拟两次对共享状态的操作。

你可以编译并执行，可以看到，仅仅是两个线程的低度并发，就非常容易碰到 former 和 latter 不相等的情况。这是因为，在两次取值的过程中，其他线程可能已经修改了 sharedState。

```
public class ThreadSafeSample {
    public int sharedState;
    public void nonSafeAction() {
        while (sharedState < 100000) {
            int former = sharedState++;
            int latter = sharedState;
            if (former != latter - 1) {
                System.out.printf("Observed data race, former is " +
                        former + ", " + "latter is " + latter);
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadSafeSample sample = new ThreadSafeSample();
        Thread threadA = new Thread(){
            public void run(){
                sample.nonSafeAction();
            }
        };
        Thread threadB = new Thread(){
```

```
            public void run(){
                sample.nonSafeAction();
            }
        };
        threadA.start();
        threadB.start();
        threadA.join();
        threadB.join();
    }
}
```

下面是在我的电脑上的运行结果：

```
C:\>c:\jdk-9\bin\java ThreadSafeSample
Observed data race, former is 13097, latter is 13099
```

将两次赋值过程用 synchronized 保护起来，使用 this 作为互斥单元，就可以避免别的线程并发的去修改 sharedState。

```
synchronized (this) {
    int former = sharedState ++;
    int latter = sharedState;
    // …
}
```

如果用 javap 反编译，可以看到类似片段，利用 monitorenter/monitorexit 对实现了同步的语义：

```
11: astore_1
12: monitorenter
13: aload_0
14: dup
15: getfield      #2                  // Field sharedState:I
18: dup_x1
…
56: monitorexit
```

我会在下一讲，对 synchronized 和其他锁实现的更多底层细节进行深入分析。

代码中使用 synchronized 非常便利，如果用来修饰静态方法，其等同于利用下面代码将方法体囊括进来：

```
synchronized (ClassName.class) {}
```

再来看看 ReentrantLock。你可能好奇什么是再入？它是表示当一个线程试图获取一个它已经获取的锁时，这个获取动作就自动成功，这是对锁获取粒度的一个概念，也就是锁的持有是以线程为单位而不是基于调用次数。Java 锁实现强调再入性是为了和 pthread 的行为进行区分。

再入锁可以设置公平性（fairness），我们可在创建再入锁时选择是否是公平的。

```
ReentrantLock fairLock = new ReentrantLock(true);
```

这里所谓的公平性是指在竞争场景中，当公平性为真时，会倾向于将锁赋予等待时间最久的线程。公平性是减少线程“饥饿”（个别线程长期等待锁，但始终无法获取）情况发生的一个办法。

如果使用 synchronized，我们根本无法进行公平性的选择，其永远是不公平的，这也是主流操作系统线程调度的选择。通用场景中，公平性未必有想象中的那么重要，Java 默认的调度策略很少会导致 “饥饿”发生。与此同时，若要保证公平性则会引入额外开销，自然会导致一定的吞吐量下降。所以，我建议只有当你的程序确实有公平性需要的时候，才有必要指定它。

我们再从日常编码的角度学习下再入锁。为保证锁释放，每一个 lock() 动作，我建议都立即对应一个 try-catch-finally，典型的代码结构如下，这是个良好的习惯。

```
ReentrantLock fairLock = new ReentrantLock(true);// 这里是演示创建公平锁，一般情况不需要。
try {
    // do something
} finally {
     fairLock.unlock();
}
```

ReentrantLock 相比 synchronized，因为可以像普通对象一样使用，所以可以利用其提供的各种便利方法，进行精细的同步操作，甚至是实现 synchronized 难以表达的用例，如：

- 带超时的获取锁尝试。

- 可以判断是否有线程，或者某个特定线程，在排队等待获取锁。
- 可以响应中断请求。
- ...

这里我特别想强调条件变量（java.util.concurrent.Condition），如果说 ReentrantLock 是 synchronized 的替代选择，Condition 则是将 wait、notify、notifyAll 等操作转化为相应的对象，将复杂而晦涩的同步操作转变为直观可控的对象行为。

条件变量最为典型的应用场景就是标准类库中的 ArrayBlockingQueue 等。

我们参考下面的源码，首先，通过再入锁获取条件变量：

```
/** Condition for waiting takes */
private final Condition notEmpty;

/** Condition for waiting puts */
private final Condition notFull;

public ArrayBlockingQueue(int capacity, boolean fair) {
    if (capacity <= 0)
        throw new IllegalArgumentException();
    this.items = new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
}
```

两个条件变量是从同一再入锁创建出来，然后使用在特定操作中，如下面的 take 方法，判断和等待条件满足：

```
public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        while (count == 0)
            notEmpty.await();
```

```
            return dequeue();
        } finally {
            lock.unlock();
        }
    }
```

当队列为空时，试图 take 的线程的正确行为应该是等待入队发生，而不是直接返回，这是 BlockingQueue 的语义，使用条件 notEmpty 就可以优雅地实现这一逻辑。

那么，怎么保证入队触发后续 take 操作呢？请看 enqueue 实现：

```
private void enqueue(E e) {
    // assert lock.isHeldByCurrentThread();
    // assert lock.getHoldCount() == 1;
    // assert items[putIndex] == null;
    final Object[] items = this.items;
    items[putIndex] = e;
    if (++putIndex == items.length) putIndex = 0;
    count++;
    notEmpty.signal(); // 通知等待的线程，非空条件已经满足
}
```

通过 signal/await 的组合，完成了条件判断和通知等待线程，非常顺畅就完成了状态流转。注意，signal 和 await 成对调用非常重要，不然假设只有 await 动作，线程会一直等待直到被打断（interrupt）。

从性能角度，synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大。但是在 Java 6 中对其进行了非常多的改进，可以参考性能对比，在高竞争情况下，ReentrantLock 仍然有一定优势。我在下一讲进行详细分析，会更有助于理解性能差异产生的内在原因。在大多数情况下，无需纠结于性能，还是考虑代码书写结构的便利性、可维护性等。

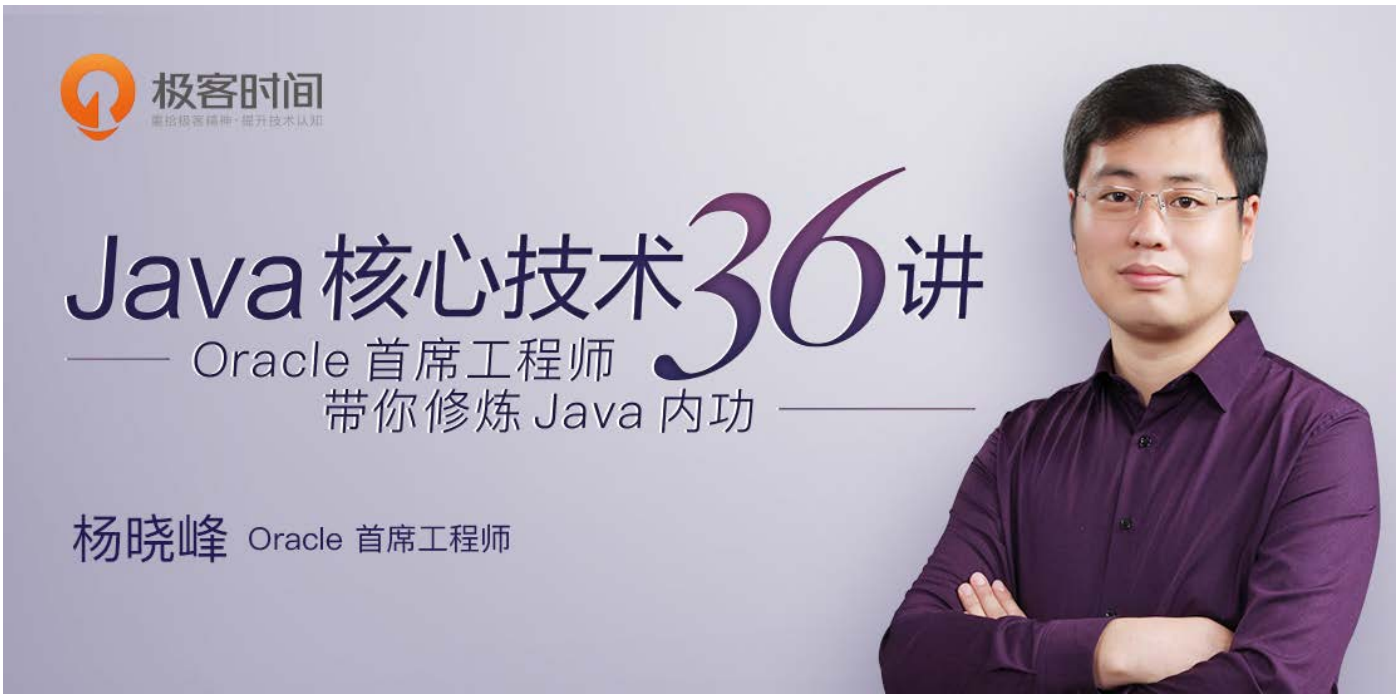
今天，作为专栏进入并发阶段的第一讲，我介绍了什么是线程安全，对比和分析了 synchronized 和 ReentrantLock，并针对条件变量等方面结合案例代码进行了介绍。下一讲，我将对锁的进阶内容进行源码和案例分析。

—课—练

关于今天我们讨论的 synchronized 和 ReentrantLock 你做到心中有数了吗？思考一下，你使用过 ReentrantLock 中的哪些方法呢？分别解决什么问题？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。



精选留言

公号-Java大后端 11

ReentrantLock是Lock的实现类，是一个互斥的同步器，在多线程高竞争条件下，ReentrantLock比synchronized有更加优异的性能表现。

1 用法比较
Lock使用起来比较灵活，但是必须有释放锁的配合动作
Lock必须手动获取与释放锁，而synchronized不需要手动释放和开启锁
Lock只适用于代码块锁，而synchronized可用于修饰方法、代码块等

2 特性比较
ReentrantLock的优势体现在：
具备尝试非阻塞地获取锁的特性：当前线程尝试获取锁，如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁
能被中断地获取锁的特性：与synchronized不同，获取到锁的线程能够响应中断，当获取到锁的线程被中断时，中断异常将会被抛出，同时锁会被释放
超时获取锁的特性：在指定的时间范围内获取锁；如果截止时间到了仍然无法获取锁，则返回

3 注意事项
在使用ReentrantLock类的时，一定要注意三点：
在finally中释放锁，目的是保证在获取锁之后，最终能够被释放
不要将获取锁的过程写在try块内，因为如果在获取锁时发生了异常，异常抛出的同时，也会导致锁无故被释放。
ReentrantLock提供了一个newCondition的方法，以便用户在同一锁的情况下可以根据不同的情况执行等待或唤醒的动作。

2018-06-07

逐梦之音 7

一直在研究JUC方面的。所有的Lock都是基于AQS来实现了。AQS和Condition各自维护了不同的队列，在使用lock和condition的时候，其实就是两个队列的互相移动。如果我们想自定义一个同步器，可以实现AQS。它提供了获取共享锁和互斥锁的方式，都是基于对state操作而言的。ReentranLock这个是可重入的。其实要弄明白它为啥可重入的呢，咋实现的呢。其实它内部自定义了同步器Sync，这个又实现了AQS，同时又实现了AOS，而后者就提供了一种互斥锁持有的方式。其实就是每次获取锁的时候，看下当前维护的那个线程和当前请求的线程是否一样，一样就可重入了。

2018-06-07

作者回复

正解

2018-06-08

Kyle 7

最近刚看完《Java 并发编程实战》，所以今天看这篇文章觉得丝毫不费力气。开始觉得，极客时间上老师讲的内容毕竟篇幅有限，更多的还是需要我们课后去深入钻研。希望老师以后讲完课也能够适当提供些参考书目，谢谢。

2018-06-07

作者回复

后面会对实现做些源码分析，其实还有各种不同的锁...

2018-06-07

BY 4

要是早看到这篇文章，我上次面试就过了。。

2018-06-07

作者回复

加油

2018-06-08

灰飞灰猪不会灰飞.烟灭 3

ReentrantLock 加锁的时候通过cas算法，将线程对象放到一个双向链表中，然后每次取出链表中的头节点，看这个节点是否和当前线程相等。是否相等比较的是线程的ID。
老师我理解的对不对啊？

2018-06-07

作者回复

嗯，并发库里都是靠自己的synchronizer

2018-06-08

xinfangke 2

老师 问你个问题 在spring中 如果标注一个方法的事务隔离级别为序列化 而数据库的隔离级别是默认的隔离级别 此时此方法中的更新 插入语句是如何执行的？能保证并发不出错吗

2018-06-08

作者回复

这个我没用过，哪位读者熟悉？

2018-06-08

sunlight001 2

老师这里说的低并发和高并发的场景，大致什么数量级的算低并发？我们做管理系统中用到锁的情况基本都算低并发吧

2018-06-07

作者回复

还真不知道有没有具体标准，但从逻辑上，低业务量不一定是“低竞争”，可能因为程序设计原因变成了“高竞争”

2018-06-07

木瓜芒果 1

杨老师，您好，synchronized在低竞争场景下可能优于retrantlock，这里的什么程度算是低竞争场景呢？

2018-06-19

作者回复

这个精确的标准我还真不知道，我觉得可以这么理解：如果大部分情况，每个线程都不需要真的获取锁，就是低竞争；反之，大部分都要获取锁才能正常工作，就是高竞争

2018-06-19

Daydayup 1

我用过读写分离锁，读锁保证速度，写锁保证安全问题。再入锁还是挺好用的。老师写的很棒，学到不少知识。感谢

2018-06-13

作者回复

非常感谢

2018-06-13

Daydayup 1

我用过读写分离锁，读锁保证速度，写锁保证安全问题。再入锁还是挺好用的。老师写的很棒，学到不少知识。感谢

2018-06-13

1

李飞

老师，可以问您一个课外题吗。具备怎样的能力才算是java高级开发

2018-06-08

Libra 1

希望后面能讲下lock源码整个的设计思想。

2018-06-07

wang_bo 1

该怎么系统学习java并发？

2018-06-07

Allen 0

为什么ReentranLock会如此高效？

2018-07-08

张小小的席大da 0

杨老师 很感谢 在您这了解到了很多以前没注意的知识点 时刻关注着您 希望您会一直讲下去

2018-07-02

猪哥灰 0

为了研究java的并发，我先把考研时候的操作系统教材拿出来再仔细研读一下，可见基础之重要性，而不管是什么语言，万变不离其宗

2018-06-29

飞鱼 0

之前有被问到synchronize和ReetrantLock底层实现上的区别，笼统的答了下前者是基于JVM实现的，后者依赖于CPU底层指令的实现，关于这个，请问有更详细的解答吗？

2018-06-17

作者回复

synchronized已经介绍了，后面我会介绍AQS，reentrantlock等多种同步结构都是利用它实现的；其实你说的靠计算机之类也对，我想你的意思是cas的实现？

2018-06-19

云学 0

锁是针对数据的，不是针对代码，一个数据一把锁，syncrise似乎违背了这一原则

2018-06-15

云学 0

看完还是觉得c++11的Lockguard比较优雅，难怪耗子哥说学习java是为了更好的用c++

2018-06-14

作者回复

互相影响，底层也有很多一致的地方，类似jmm之类也是c++掉的坑，别人吸取了教训

2018-06-14

Miaozhe 0

杨老师，问个问题，看网上有说Condition的await和signal方法，等同于Object的wait和notify，看了一下源码，没有直接的关系。
ReentractLock是基于双向链表的对接和CAS实现的，感觉比Object增加了很多逻辑，怎么会比Synchronized效率高？有疑惑。

2018-06-12

作者回复

你看到的很对，如果从单个线程做的事来看，也许并没有优势，不管是空间还是时间，但ReentrantLock这种所谓cas，或者叫lock-free，方式的好处，在于高竞争情况的扩展性，而原来那种频繁的上下文切换则会导致吞吐量迅速下降

2018-06-12