

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

2018-06-07 杨晓峰

Java核心技术36讲

[进入课程 >](#)



讲述：黄洲君

时长 09:36 大小 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。

 复制代码

```
1 public class ThreadSafeSample {
2     public int sharedState;
3     public void nonSafeAction() {
```

```

4      while (sharedState < 100000) {
5          int former = sharedState++;
6          int latter = sharedState;
7          if (former != latter - 1) {
8              System.out.printf("Observed data race, former is " +
9                  former + ", " + "latter is " + latter);
10         }
11     }
12 }
13
14 public static void main(String[] args) throws InterruptedException {
15     ThreadSafeSample sample = new ThreadSafeSample();
16     Thread threadA = new Thread(){
17         public void run(){
18             sample.nonSafeAction();
19         }
20     };
21     Thread threadB = new Thread(){
22         public void run(){
23             sample.nonSafeAction();
24         }
25     };
26     threadA.start();
27     threadB.start();
28     threadA.join();
29     threadB.join();
30 }
31 }

```

下面是在我的电脑上的运行结果：


 复制代码

```

1 C:\>c:\jdk-9\bin\java ThreadSafeSample
2 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`。

 复制代码


```

1 synchronized (this) {
2     int former = sharedState ++;
3     int latter = sharedState;

```

```
4      // ...
5  }
```


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：

 复制代码

```
1 11: astore_1
2 12: monitorenter
3 13: aload_0
4 14: dup
5 15: getfield    #2           // Field sharedState:I
6 18: dup_x1
7 ...
8 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：

 复制代码

```
1 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。


 复制代码

```
1 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。

 复制代码

```
1 ReentrantLock fairLock = new ReentrantLock(true); // 这里是演示创建公平锁，一般情况不需要。
2 fairLock.lock();
3 try {
4     // do something
5 } finally {
6     fairLock.unlock();
7 }
```

`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：

 复制代码

```
1 /** Condition for waiting takes */
2 private final Condition notEmpty;
3
4 /** Condition for waiting puts */
5 private final Condition notFull;
6
7 public ArrayBlockingQueue(int capacity, boolean fair) {
8     if (capacity <= 0)
9         throw new IllegalArgumentException();
10    this.items = new Object[capacity];
11    lock = new ReentrantLock(fair);
12    notEmpty = lock.newCondition();
13    notFull = lock.newCondition();
14 }
```

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：

 复制代码

```
1 public E take() throws InterruptedException {
2     final ReentrantLock lock = this.lock;
3     lock.lockInterruptibly();
4     try {
5         while (count == 0)
6             notEmpty.await();
7         return dequeue();
8     } finally {
9         lock.unlock();
10    }
11 }
```

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：

```
1 private void enqueue(E e) {
2     // assert lock.isHeldByCurrentThread();
3     // assert lock.getHoldCount() == 1;
4     // assert items[putIndex] == null;
5     final Object[] items = this.items;
6     items[putIndex] = e;
7     if (++putIndex == items.length) putIndex = 0;
8     count++;
9     notEmpty.signal(); // 通知等待的线程，非空条件已经满足
10 }
```

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。

一课一练

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。

Java核心技术36讲

—— 前 Oracle 首席工程师
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 第14讲 | 谈谈你知道的设计模式？

下一篇 周末福利 | 谈谈我对Java学习和面试的看法

精选留言 (52)

写留言



公号-代码...

2018-06-07

87

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...

展开 ▾



逐梦之音

2018-06-07

44

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。ReentrantLock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， ...
展开 ▾

作者回复: 正解



BY

2018-06-07

👍 16

要是早看到这篇文章，我上次面试就过了。。

展开 ▾

作者回复: 加油



Kyle

2018-06-07

👍 11

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。

展开 ▾

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...



Daydayup

2018-06-13

👍 7

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

展开 ▾

作者回复: 非常感谢



灰飞灰猪不...

2018-06-07

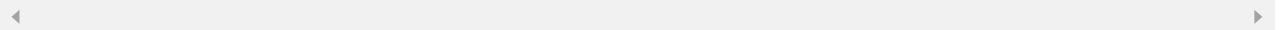
👍 6

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？

展开 ∨

作者回复: 嗯，并发库里都是靠自己的synchronizer



Jerry银银

2019-02-01

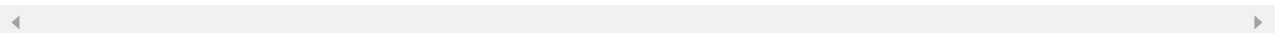
👍 5

先说说点学习感受：

并发领域的知识点很多也很散，并且知识点之间交错的。比如：synchronized，这个小小的关键字，能够彻底理解它，需要的知识储备有：基本使用场景、对锁的理解、对线程安全的理解、对同步语义的理解、对JMM的理解等等。有时候，一个知识点暂时做不到透彻理解，可能是正常的，需要再继续学习其它的知识点，等到一定时候，回过头来重新学...

展开 ∨

作者回复: 非常感谢反馈



猪哥灰

2018-06-29

👍 5

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

展开 ∨



xinfangke

2018-06-08

👍 5

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

作者回复: 这个我没用过，哪位读者熟悉？



木瓜芒果

2018-06-19

👍 3

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

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



sunlight00...

2018-06-07

👍 3

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

展开 ∨

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”



Neil

2019-01-07

👍 2

可以理解为synchronized是悲观锁 另一个是乐观锁

展开 ∨

作者回复: 基本如此，乐观、悲观是两种不同的处理策略



李飞

2018-06-08

👍 2

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



lincoln...
2019-02-25

1

看到留言区，有个同学问：new ReentrantLock()能不能写到这里，我看了回复，不是很认同，不知道对不对。其实：new lock和lock api的调用得写到try外面，写到这里会有问题，如下：

lock() api 可能会抛出异常，如果放到try里面，在finally里面unlock会再抛出异常(因为当前状态不对)，这个时候 "解锁异常"会隐藏"加锁异常"，也就是异常堆栈只有 "解锁异...

展开



leleba
2018-11-02

1

这里怎么没有精彩的留言了呢？很难吗，反正我觉得难，但必须要学



PP
2018-08-14

1

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...

展开



时间总漫不...
2018-08-10

1

老师，jmm什么时候将工作内存的值写入到主内存中呢？

作者回复: volatile读写、同步块这种



clz134152...
2018-08-05

1

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。ReentrantLock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...

展开 ∨



Daydayup

2018-06-13

👍 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

展开 ∨



Miaoze

2018-06-12

👍 1

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。

ReentrantLock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。

展开 ∨

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

