二

# 15 synchronized和ReentrantLock有什么区别呢？-极客时间

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。

更进一步，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
fairLock.lock();
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。

## 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。

上一页 下一页