17 资源有限，请排队等候—Synchronized使用、原理及缺陷

更新时间：2019-10-24 11:50:40

> 人生的旅途，前途很远，也很暗。然而不要怕，不怕的人的面前才有路。
>
> —— 鲁 迅

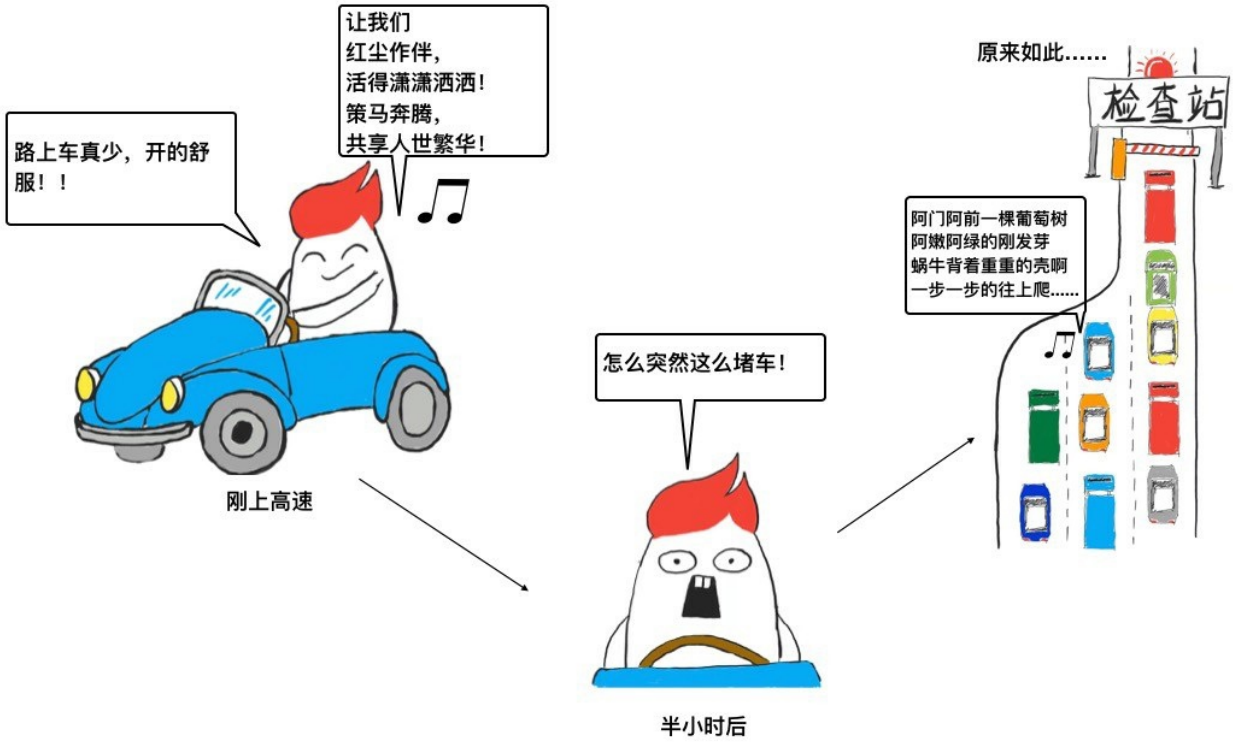
到现在为止，本专栏已经发布了近一半的内容。我还记得早在第三节就有同学留言预测下一节要讲synchronized。确实，很多讲解Java并发编程的书籍会比较早的安排讲解 synchronized 关键字。其实synchronized 使用起来非常简单，并且几乎不用做什么思考，需要确保线程安全的部分直接用就好了。所以我觉得只是使用的话没什么好讲的，随着其它内容一并介绍下就可以了。在介绍了并发的三大特性，顺便学习完Atomic 和 volatile 后，再深入学习synchronized。我认为这样的安排更能帮助读者理解。

1、synchronized 的作用

前两节我们学习了能保证原子性的 Atomic 变量以及保证可见性和有序性的 volatile 关键字。这两种方式是轻量级的同步方式，不过存在其局限性，前面已经做过总结。那么“重量级”的同步如何做呢？synchronized 代码块就是一种实现方式。在 synchronized 代码块中的代码在多线程中会同步执行，同步执行的意思就是——排队。这就像我们去体检，每个人可以并行从家里到医院，并行拿表、填表，并行走到各个检查室门口。但是，一旦要做检查了，我们就需要在检查室门口排队。这是因为只有一个大夫做检查，大夫是共享资源。对共享资源的访问我们要保证同步，否则就会出现问题。

synchronized 作用域中的代码为同步执行的，也就是并发的情况下，执行到对同一个对象加锁的 synchronized 代码块时，为串行执行的。这里注意，并不是同一个同步代码块，而是对同一个对象上锁的同步代码块。这意味着范围更广。此外 synchronized 可以确保可见性，在一个线程执行完 synchronized 代码后，所有代码中对变量值的变化都能立即被其它线程所看到。

由于 synchronized 关键字会使得代码串行执行，这就丧失了多线程的优势。并且 synchronized 关键字的使用也有相应成本。所以我们代码中能不用 synchronized 就不用。当不得不用的时候，需要尽量控制 synchronized 代码块中的代码行数。这就像高速公路上，本来三车道，所有车辆开得很快，但是突然遇到检查点，车辆只能一辆一辆通过，那么速度一下就慢了下来，必然造成堵车。我们应该尽量减少这种人为堵点。



2、synchronized 的使用

synchronized 的使用非常简单，有两种方式，第一种是同步代码块。

我们拿之前例子的代码片段回顾下：

```
synchronized (tasks) {
    if (tasks.size() > 0) {
        task = tasks.removeFirst();
        sleep(100);
        tasks.notifyAll();
    } else {
        tasks.wait();
    }
}
```

```
synchronized (tasks) {
    if (tasks.size() < MAX) {
         Task task = new Task(new Random().nextInt(3) + 1, getPunishedWord());
        tasks.addLast(task);
        System.out.println(threadName + "留了作业，抄写" + task.getWordToCopy() + " " + task.getLeftCopyCount() + "次");
        tasks.notifyAll();
    } else {
        System.out.println(threadName+"开始等待");
        tasks.wait();
        System.out.println("teacher线程 " + threadName + "线程-" + name + "等待结束");
    }
}
```

这是生产者/消费者那一节的部分代码。第一段是学生写作业的代码，第二段是老师留作业的代码。可以看到 synchronized 的使用很简单，把你需要同步的代码放入 synchronized 关键字后面的大括号中即可。

另外你肯定注意到 synchronized (tasks) ，这行代码小括号里的 tasks 对象。为什么要这么写呢？这是和 synchronized 实现的方式相关的。你是不是心里在想：这个对象一定是被加锁的对象，加了锁之后，别的线程就不能对该对象访问了。这里理解起来好像非常的自然。其实并不是这样，小括号里的对象是可以是任意的对像。之前我们讲解过这一点，这个对象相当于是同步代码块的看门人，每个对其 synchronized 的线程，它都会记录下来，然后等到同步代码块没有线程执行的时候，它就会通知其它线程来执行同步代码块。

所以我们并不是对此对象加锁，只是让它来维护秩序。这个人是谁其实并无所谓。但是我们的例子中，并发的线程并不是同样类型的 Thread，一个是 Student，还一个是 Teacher。对于不同对象的同步控制，一定要选用两个线程都持有的对象才行。否则各自使用不同的对象，相当于聘用了两个看门人，各看各的门，毫无瓜葛。那么原本想要串行执行的代码仍旧会并行执行。

第二种，使用 synchronized 关键字修饰方法：

```
public synchronized void eat(){
 .......
  .......
}
```

你是不是会好奇，这里没有锁对象，是如何加锁的呢？其实同步方法的锁对象就是 this。这和下面代码把方法中代码全部用 synchronized(this) 括起来的效果是一样的：

```
public void eat(){
 synchronized(this){
  .......
   .......
 }
}
```

如果是 synchroinized 的是静态方法，如下面代码：

```
public static synchronized void eat(){
 .......
  .......
}
```

此时同步方法为类的 Class 对象。如果上述静态方法所在的类为 Test。那么锁对象就是 Test.class。

构造方法是不能使用 synchronized 关键字修饰的。因为同步的构造方法是讲不通的，对于一个指定的对象，它只会有唯一的创建线程，所以不需要使用 synchroinzied 修饰。

下面是 synchronized 的使用总结：

1、选用一个锁对象，可以是任意对象；

2、锁对象锁的是同步代码块，并不是自己；

3、不同类型的多个 Thread 如果有代码要同步执行，锁对象要使用所有线程共同持有的同一个对象；

4、需要同步的代码放到大括号中。需要同步的意思就是需要保证原子性、可见性、有序性中的任何一种或多种。不要放不需要同步的代码进来，影响代码效率。

3、synchronized 原理

synchronized 的秘密其实都在同步对象上。就像上文所说，这个对象就是一个看门人，每次只允许一个线程进来，进门后此线程可以做任何自己想做的事情，然后再出来。此时看门人会吼一嗓子：没人了，可以进来啦！其它线程听到吼声，马上都冲了过来。但总有个敏捷值最高的线程先冲入门内，那么其它线程只好继续等待。

其实 synchronized 原理基本和上面的例子一样。下面我们真正来看看其实现原理是什么。相信如果你看懂了上面的例子，对 synchronized 原理的理解不会有任何难度。

我们一直说的同步对象，其实就是任何一个普通的对象。那么一个普通的java对象是如何来做同步这件事的呢？这是因为每个对象都关联了一个 monitor lock。

当一个线程获取了 monitor lock 后，其它线程如果运行到获取同一个 monitor 的时候就会被 block 住。当这个线程执行完同步代码，则会释放 monitor lock。在后一个线程获取锁后，happens-before 原则生效，前一个线程所做的任何修改都会被这个线程看到。

我们再深入底层一点来分析。每个 Java 对象在 JVM 的对等对象的头中保存锁状态，指向 ObjectMonitor。ObjectMonitor 保存了当前持有锁的线程引用，EntryList 中保存目前等待获取锁的线程，WaitSet 保存 wait 的线程。此外还有一个计数器，每当线程获得 monitor 锁，计数器 +1，当线程重入此锁时，计数器还会 +1。当计数器不为0时，其它尝试获取 monitor 锁的线程将会被保存到EntryList中，并被阻塞。当持有锁的线程释放了monitor 锁后，计数器 -1。当计数器归位为 0 时，所有 EntryList 中的线程会尝试去获取锁，但只会有一个线程会成功，没有成功的线程仍旧保存在 EntryList 中。**由此可以看出 monitor 锁是非公平锁**。

我们看一下前面例子中 Student 类编译之后的汇编指令。或者你也可以自己写一段简单的带有 synchronized 关键字的代码。先将其编译为.class 文件，然后使用 javap -c xxx.class 进行反汇编。我们就可以得到 java 代码对应的汇编指令。里面可以找到如下两行指令。

```
.....
15: monitorenter
.....
128: monitorexit
.....
```

这两条指令就是上面所讲述的获取锁和释放锁的关键指令。我看过使用 zookeepe r实现分布式锁的 Curator 框架源代码，Curator 的互斥锁和 monitor 锁在原理上一模一样。

4、synchronized 使用注意

1. synchronized 使用的为非公平锁，如果你需要公平锁，那么不要使用 synchronized。可以使用 ReentrantLock，设置为公平锁。关于 ReentrantLock，会在后面章节进行讲解；
2. 锁对象不能为 null。如果锁对象为 null，何谈对象头，以及保存与其关联的 monitor 锁呢？所以代码中要确保 synchronized使用的锁对象不为 null；
3. 只把需要同步的代码放入 synchronized 代码块。如果不思考，为了线程安全把方法中全部代码都放入同步代码块，那么将会丧失多线程的优势。再多的线程也只能串行执行，这完全违背了并发的初衷；
4. 只有使用同一个对象作为锁对象，才能同步。记住是同一个对象，而不是同一个类。有一种常犯的错误是，不同

线程持有的是同一个类的不同实例。那么该对象实例用作锁对象的话，多个线程并不会同步。还一种错误是使用不同类的实例作为锁对象，但是期望不同位置的同步代码块能够同步执行。这是不可能达到你想要的效果的。

5、总结

本节我们学习了Java多线程领域使用最多的同步方式 synchronized 关键字。synchronized 使用方便简单，但是一定注意其作用范围不要过大。另外 synchronized 也有其局限性。我们在后面会学习到 Lock 接口及其实现，可以解决 synchronized 存在的问题。

}

16 让你眼见为实—volatile详解

18 线程作用域内共享变量—深入解析ThreadLocal