

06 | 用“等待-通知”机制优化循环等待

王宝令 2019-03-12



00:00

10:56

讲述：王宝令 大小：10.03M

由上一篇文章你应该已经知道，在**破坏占用且等待条件**的时候，如果转出账本和转入账本不满足同时在文件架上这个条件，就用死循环的方式来循环等待，核心代码如下：

 复制代码

```
1 // 一次性申请转出账户和转入账户，直到成功
2 while(!actr.apply(this, target))
3     ;
4
```

如果 `apply()` 操作耗时非常短，而且并发冲突量也不大时，这个方案还挺不错的，因为这种场景下，循环上几次或者几十次就能一次性获取转出账户和转入账户了。但是如果 `apply()` 操作耗时长，或者并发冲突量大的时候，循环等待这种方案就不适用了，因为在这种场景下，可能要循环上万次才能获取到锁，大消耗 CPU 了。

主方法才能获取到锁，从而消耗 CPU 了。

其实在这种场景下，最好的方案应该是：如果线程要求的条件（转出账本和转入账本同在文件架上）不满足，则线程阻塞自己，进入**等待**状态；当线程要求的条件（转出账本和转入账本同在文件架上）满足后，**通知**等待的线程重新执行。其中，使用线程阻塞的方式就能避免循环等待消耗 CPU 的问题。

那 Java 语言是否支持这种**等待 - 通知机制**呢？答案是：一定支持（毕竟占据排行榜第一那么久）。下面我们就来看看 Java 语言是如何支持**等待 - 通知机制**的。

完美的就医流程

在介绍 Java 语言如何支持等待 - 通知机制之前，我们先看一个现实世界里面的就医流程，因为它有着完善的等待 - 通知机制，所以对比就医流程，我们就能更好地理解和应用并发编程中的等待 - 通知机制。

就医流程基本上是这样：

1. 患者先去挂号，然后到就诊门口分诊，等待叫号；
2. 当叫到自己的号时，患者就可以找大夫就诊了；
3. 就诊过程中，大夫可能会让患者去做检查，同时叫下一位患者；
4. 当患者做完检查后，拿检测报告重新分诊，等待叫号；
5. 当大夫再次叫到自己的号时，患者再去找大夫就诊。

或许你已经发现了，这个有着完美等待 - 通知机制的就医流程，不仅能够保证同一时刻大夫只为一个患者服务，而且还能够保证大夫和患者的效率。与此同时你可能也会有疑问，“这个就医流程很复杂呀，我们前面描述的等待 - 通知机制相较而言是不是太简单了？”那这个复杂度是否是必须的呢？这个是必须的，我们不能忽视等待 - 通知机制中的一些细节。

下面我们来对比看一下前面都忽视了哪些细节。

1. 患者到就诊门口分诊，类似于线程要去获取互斥锁；当患者被叫到时，类似线程已经获取到锁了。
2. 大夫让患者去做检查（缺乏检测报告不能诊断病因），类似于线程要求的条件没有满足。
3. 患者去做检查，类似于线程进入等待状态；然后**大夫叫下一个患者，这个步骤我们在前面的等待 - 通知机制中忽视了，这个步骤对应到程序里，本质是线程释放持有的互斥锁。**
4. 患者做完检查，类似于线程要求的条件已经满足；**患者拿检测报告重新分诊，类似于线程需要重新获取互斥锁，这个步骤我们在前面的等待 - 通知机制中也忽视了。**

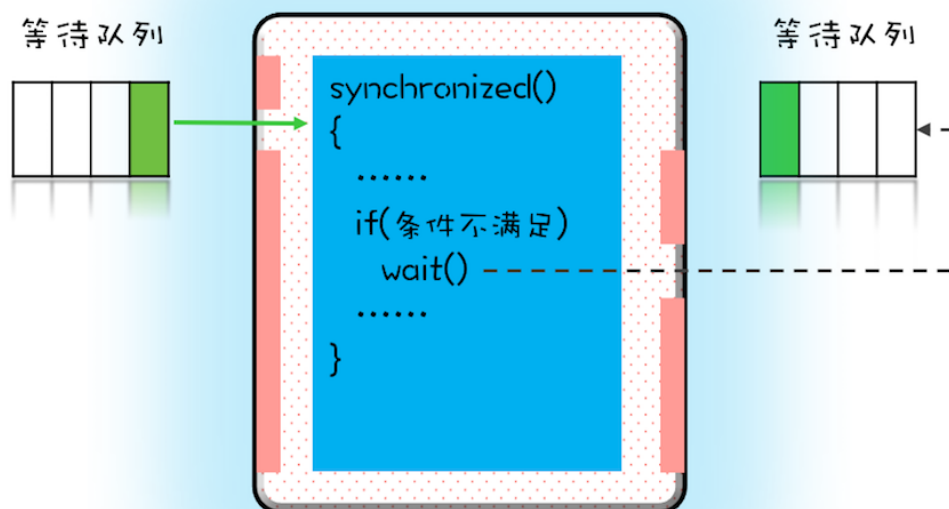
所以加上这些至关重要的细节，综合一下，就可以得出**一个完整的等待 - 通知机制：线程首先获取互斥锁，当线程要求的条件不满足时，释放互斥锁，进入等待状态；当要求的条件满足时，通**

知等待的线程，重新获取互斥锁。

用 synchronized 实现等待 - 通知机制

在 Java 语言里，等待 - 通知机制可以有多种实现方式，比如 Java 语言内置的 synchronized 配合 wait()、notify()、notifyAll() 这三个方法就能轻松实现。

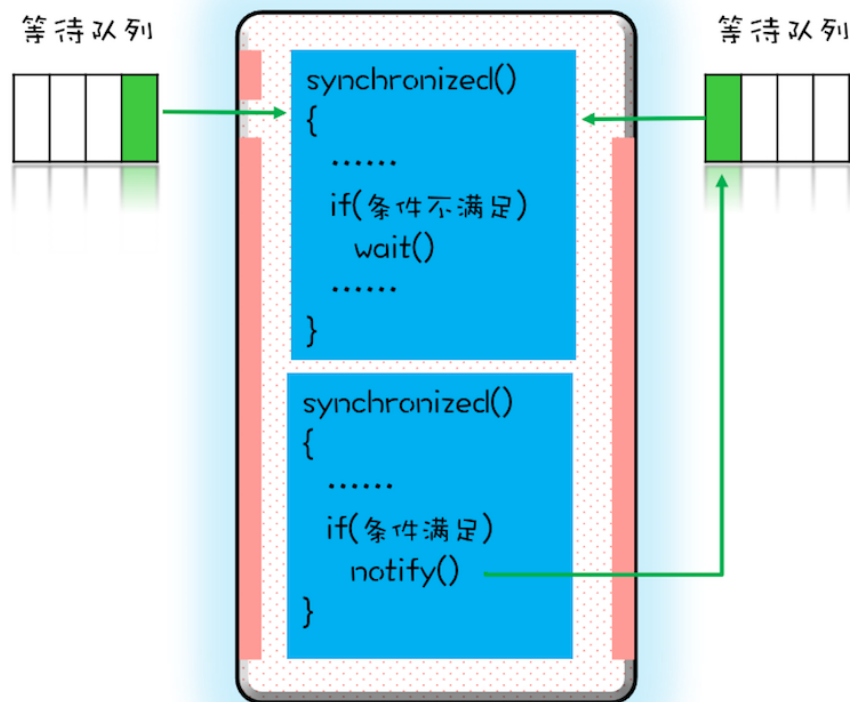
如何用 synchronized 实现互斥锁，你应该已经很熟悉了。在下面这个图里，左边有一个等待队列，同一时刻，只允许一个线程进入 synchronized 保护的临界区（这个临界区可以看作大夫的诊室），当有一个线程进入临界区后，其他线程就只能进入图中左边的等待队列里等待（相当于患者分诊等待）。**这个等待队列和互斥锁是一对一的关系，每个互斥锁都有自己独立的等待队列。**



wait() 操作工作原理图

在并发程序中，当一个线程进入临界区后，由于某些条件不满足，需要进入等待状态，Java 对象的 wait() 方法就能够满足这种需求。如上图所示，当调用 wait() 方法后，当前线程就会被阻塞，并且进入到右边的等待队列中，**这个等待队列也是互斥锁的等待队列**。线程在进入等待队列的同时，**会释放持有的互斥锁**，线程释放锁后，其他线程就有机会获得锁，并进入临界区了。

那线程要求的条件满足时，该怎么通知这个等待的线程呢？很简单，就是 Java 对象的 notify() 和 notifyAll() 方法。我在下面这个图里为你大致描述了这个过程，当条件满足时调用 notify()，会通知等待队列（**互斥锁的等待队列**）中的线程，告诉它**条件曾经满足过**。



notify() 操作工作原理图

为什么说是曾经满足过呢？因为**notify()** 只能保证在通知时间点，条件是满足的。而被通知线程的**执行时间点和通知的时间点**基本上不会重合，所以当线程执行的时候，很可能条件已经不满足了（保不齐有其他线程插队）。这一点你需要格外注意。

除此之外，还有一个需要注意的点，被通知的线程要想重新执行，仍然需要获取到互斥锁（因为曾经获取的锁在调用 wait() 时已经释放了）。

上面我们一直强调 wait()、notify()、notifyAll() 方法操作的等待队列是互斥锁的等待队列，所以如果 synchronized 锁定的是 this，那么对应的一定是 this.wait()、this.notify()、this.notifyAll()；如果 synchronized 锁定的是 target，那么对应的一定是 target.wait()、target.notify()、target.notifyAll()。而且 wait()、notify()、notifyAll() 这三个方法能够被调用的前提是已经获取了相应的互斥锁，所以我们会发现 wait()、notify()、notifyAll() 都是在 synchronized{} 内部被调用的。如果在 synchronized{} 外部调用，或者锁定的 this，而用 target.wait() 调用的话，JVM 会抛出一个运行时异常：

```
java.lang.IllegalMonitorStateException。
```


小试牛刀：一个更好地资源分配器

等待 - 通知机制的基本原理搞清楚后，我们就来看看它如何解决一次性申请转出账户和转入账户的问题吧。在这个等待 - 通知机制中，我们需要考虑以下四个要素。

1. 互斥锁：上一篇文章我们提到 Allocator 需要是单例的，所以我们可以用 this 作为互斥锁。


2. 线程要求的条件：转出账户和转入账户都没有被分配过。
3. 何时等待：线程要求的条件不满足就等待。
4. 何时通知：当有线程释放账户时就通知。

将上面几个问题考虑清楚，可以快速完成下面的代码。需要注意的是我们使用了：

 复制代码

```
1 while(条件不满足) {  
2     wait();  
3 }  
4
```

利用这种范式可以解决上面提到的**条件曾经满足过**这个问题。因为当 `wait()` 返回时，有可能条件已经发生了变化了，曾经条件满足，但是现在已经不满足了，所以要重新检验条件是否满足。范式，意味着是经典做法，所以没有特殊理由不要尝试换个写法。后面在介绍“管程”的时候，我会详细介绍这个经典做法的前世今生。

 复制代码

```
1 class Allocator {  
2     private List<Object> als;  
3     // 一次性申请所有资源  
4     synchronized void apply(  
5         Object from, Object to){  
6         // 经典写法  
7         while(als.contains(from) ||  
8             als.contains(to)){  
9             try{  
10                wait();  
11            }catch(Exception e){  
12            }  
13        }  
14        als.add(from);  
15        als.add(to);  
16    }  
17    // 归还资源  
18    synchronized void free(  
19        Object from, Object to){  
20        als.remove(from);  
21        als.remove(to);  
22        notifyAll();  
23    }  
24 }  
25
```

尽量使用 `notifyAll()`

在上面的代码中，我用的是 `notifyAll()` 来实现通知机制，为什么不使用 `notify()` 呢？这二者是有

区别的，**notify()** 是会随机地通知等待队列中的一个线程，而 **notifyAll()** 会通知等待队列中的所有线程。从感觉上来讲，应该是 **notify()** 更好一些，因为即便通知所有线程，也只有一个线程能够进入临界区。但那所谓的“感觉”往往都蕴藏着风险，实际上使用 **notify()** 也很有风险，它的风险在于可能导致某些线程永远不会被通知到。

假设我们有资源 A、B、C、D，线程 1 申请到了 AB，线程 2 申请到了 CD，此时线程 3 申请 AB，会进入等待队列（AB 分配给线程 1，线程 3 要求的条件不满足），线程 4 申请 CD 也会进入等待队列。我们再假设之后线程 1 归还了资源 AB，如果使用 **notify()** 来通知等待队列中的线程，有可能被通知的是线程 4，但线程 4 申请的是 CD，所以此时线程 4 还是会继续等待，而真正该唤醒的线程 3 就再也没有机会被唤醒了。

所以除非经过深思熟虑，否则尽量使用 **notifyAll()**。

总结

等待 - 通知机制是一种非常普遍的线程间协作的方式。工作中经常看到有同学使用轮询的方式来等待某个状态，其实很多情况下都可以用今天我们介绍的等待 - 通知机制来优化。Java 语言内置的 **synchronized** 配合 **wait()**、**notify()**、**notifyAll()** 这三个方法可以快速实现这种机制，但是它们的使用看上去还是有点复杂，所以你需要认真理解等待队列和 **wait()**、**notify()**、**notifyAll()** 的关系。最好用现实世界做个类比，这样有助于你的理解。

Java 语言的这种实现，背后的理论模型其实是管程，这个很重要，不过你不用担心，后面会有专门的一章来介绍管程。现在你只需要能够熟练使用就可以了。

课后思考

很多面试都会问到，**wait()** 方法和 **sleep()** 方法都能让当前线程挂起一段时间，那它们的区别是什么？现在你也试着回答一下吧。

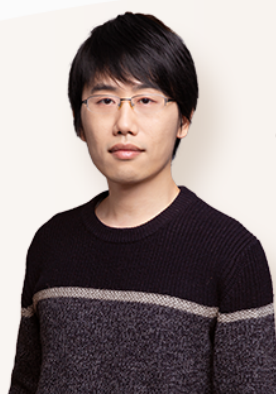
欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

猜你喜欢

玩转 Spring 全家桶

一站通关 Spring、Spring Boot 与 Spring Cloud

戳此试读 



丁雪丰
平安壹钱包高级架构师
《Spring Boot 实战》
《Spring 攻略》译者



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

Ctrl + Enter 发表

0/2000字

提交留言

精选留言(17)



邈邈的流浪剑客

wait会释放当前占有的锁，sleep不会释放锁

👍 4 2019-03-12



姜戈

wait与sleep区别在于：

1. wait会释放所有锁而sleep不会释放锁资源.
2. wait只能在同步方法和同步块中使用，而sleep任何地方都可以.
3. wait无需捕捉异常，而sleep需要.

两者相同点：都会让渡CPU执行时间，等待再次调度！

👍 3 2019-03-12



Home

补充一下姜戈同学回答；1 sleep是Thread的方法，而wait是Object类的方法；2：sleep方法调用的时候必须指定时间

👍 2 2019-03-12



Geek_e726b7

应该是!als.contains(from) || !als.contains(to)才wait()吧

👍 1 2019-03-12

作者回复: 有发现一个大bug, 多谢多谢!



峰

wait会释放当前调用对象的锁, 这也就意味着调用之前的持有这把锁, 否则着会抛出异常。
wait是Object方法, sleep是Thread的静态方法。

👍 1 2019-03-12



0bug

```
// 一次性申请转出账户和转入账户, 直到成功
while(!actr.apply(this, target))
;
```

这段代码的while可以去掉了吧?

👍 2019-03-12



张先生、

- 1、wait释放持有的锁, sleep不会
- 2、sleep设置挂起时间, wait如果不调用notify或者notifyall, 一直挂起
- 3、wait是Object方法, sleep是Thread方法

👍 2019-03-12



哲民

分布式系统中对应wait和notify有哪些方式呢?

👍 2019-03-12



crazypokerk

wait()方法与sleep()方法的不同之处在于, wait()方法会释放对象的“锁标志”。当调用某一对象的wait()方法后, 会使当前线程暂停执行, 并将当前线程放入对象等待池中, 直到调用了notify()方法后, 将从对象等待池中移出任意一个线程并放入锁标志等待池中, 只有锁标志等待池中的线程可以获取锁标志, 它们随时准备争夺锁的拥有权。当调用了某个对象的notifyAll()方法, 会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

sleep()方法需要指定等待的时间, 它可以让当前正在执行的线程在指定的时间内暂停执行, 进入阻塞状态, 该方法既可以让其他同优先级或者高优先级的线程得到执行的机会, 也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”, 也就是说如果有synchronized同步块, 其他线程仍然不能访问共享数据。

👍 2019-03-12



狂战俄洛伊

Sleep是睡一段时间就自己起来了，wait的话得等到别人叫醒他。个人感觉sleep除了增加程序运行时间以外，没别的意义呀



2019-03-12



王智

wait会主动释放锁,sleep不会;

wait()中指定毫秒值,时间到了需要使用notify()/notifyAll()唤醒; sleep()中指定毫秒值是睡眠一定时间自动唤醒.

大学操作系统学的,不知道记得对不对!!!



2019-03-12



往事随风，顺其自然

while 条件中不是与操作？怎么是或运算



2019-03-12



Zach_

老师，怎么查看synchronized(A.class)阻塞后进入的等待队列？就是，等待队列，对应的现实代码，在哪里能看到啊？



2019-03-12



渔夫

如果所有线程的等待条件都是一样的话，就可以使用 notify 来进行通知，对吗？



2019-03-12



哈呼呼

wait让线程挂起的时候，会释放自己持有的锁资源，而sleep还会继续持有锁资源。



2019-03-12



张学磊

非常棒的课程，通俗易懂！



2019-03-12



Pwhxbdk

老师 那自旋锁是不是效率更高?



2019-03-12

作者回复: 看场景, 并发量大的时候长时间拿不到锁会浪费cpu, 并发量小的时候效率高