06 | 用“等待-通知”机制优化循环等待

2019-03-12 王宝令

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：

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

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，太消耗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**。

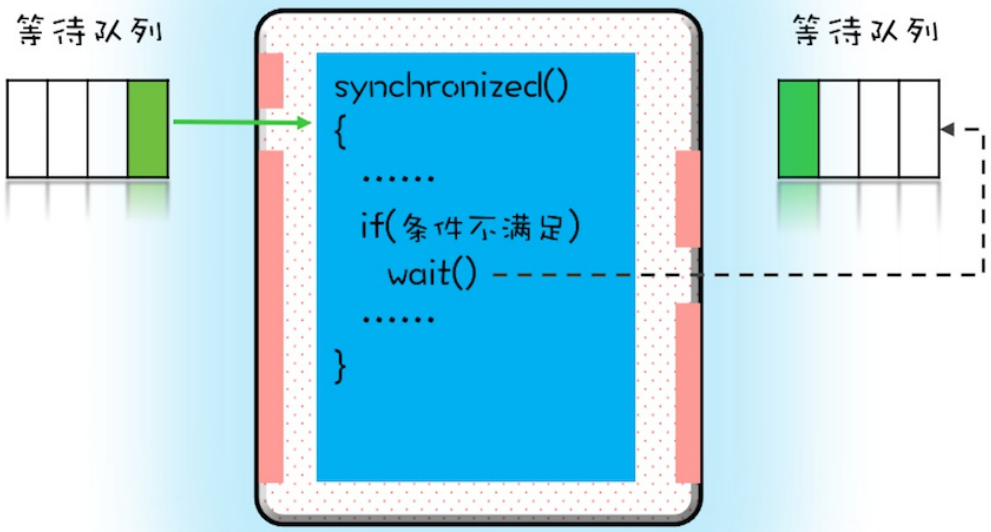
所以加上这些至关重要的细节，综合一下，就可以得出**一个完整的等待-通知机制：线程首先获取互斥锁，当线程要求的条件不满足时，释放互斥锁，进入等待状态；当要求的条件满足时，通知等待的线程，重新获取互斥锁**。

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**

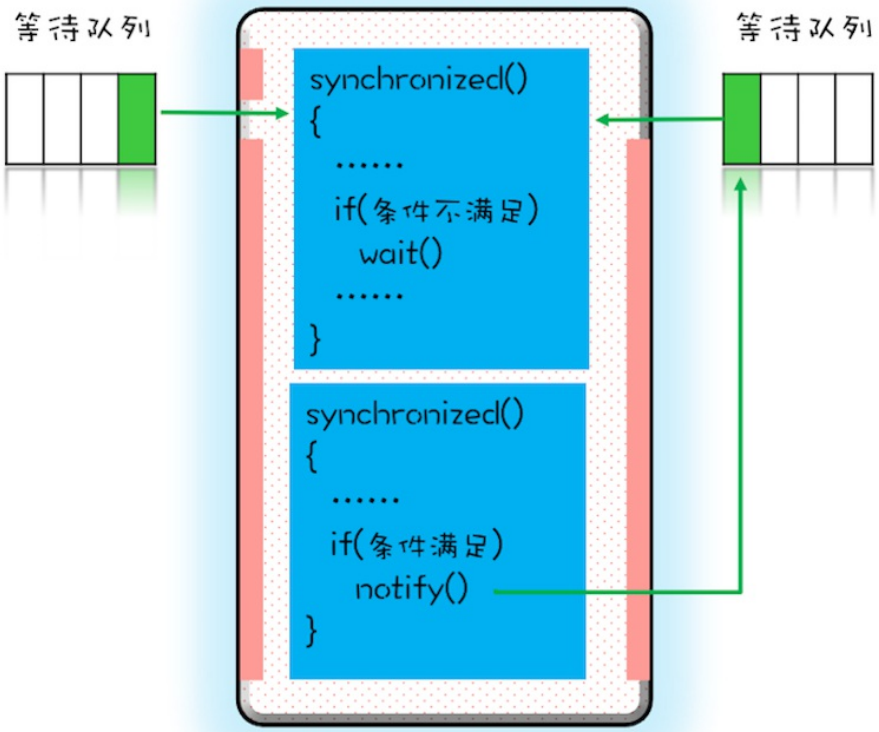
列。



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：

```
while(条件不满足) {
  wait();
}
```

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。

```
class Allocator {
  private List<Object> als;
  // 一次性申请所有资源
  synchronized void apply(
    Object from, Object to){
    // 经典写法
    while(als.contains(from) ||
         als.contains(to)){
      try{
        wait();
      }catch(Exception e){
      }
    }
    als.add(from);
    als.add(to);
  }
  // 归还资源
  synchronized void free(
    Object from, Object to){
    als.remove(from);
    als.remove(to);
    notifyAll();
  }
}
```

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。

姜戈

105

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！

2019-03-12

Home

59

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

2019-03-12

crazypokerk

37

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

作者回复

[]

2019-03-12

wang

33

```
public class MyLock {
// 测试转账的main方法
public static void main(String[] args) throws InterruptedException {
Account src = new Account(10000);
Account target = new Account(10000);
CountDownLatch countDownLatch = new CountDownLatch(9999);
for (int i = 0; i < 9999; i++) {
new Thread(()->{
src.transactionToTarget(1,target);
countDownLatch.countDown();
}).start();
}
countDownLatch.await();
```

```
System.out.println("src="+src.getBanalce() );
System.out.println("target="+target.getBanalce() );
}
static class Account{ //账户类
public Account(Integer banalce) {
this.banalce = banalce;
}
private Integer banalce;
public void transactionToTarget(Integer money,Account target){//转账方法
Allocator.getInstance().apply(this,target);
this.banalce -= money;
target.setBanalce(target.getBanalce()+money);
Allocator.getInstance().release(this,target);
}
public Integer getBanalce() {
return banalce;
}
public void setBanalce(Integer banalce) {
this.banalce = banalce;
}
}
static class Allocator { //单例锁类
private Allocator(){}
private List<Account> locks = new ArrayList<>();
public synchronized void apply(Account src,Account tag){
while (locks.contains(src)||locks.contains(tag)) {
try {
this.wait();
} catch (InterruptedException e) {
}
}
locks.add(src);
locks.add(tag);
}
public synchronized void release(Account src,Account tag){
locks.remove(src);
locks.remove(tag);
this.notifyAll();
}
public static Allocator getInstance(){
return AllocatorSingle.install;
}
static class AllocatorSingle{
```

```
public static Allocator install = new Allocator();
}
}
}
```

2019-03-12

作者回复

高手高手，让我写也这不这样好

2019-03-13

虎虎 17

困惑

1. 对于从来没有获得过互斥锁的线程 所在的等待队列 和 因为wait() 释放锁而进入了等待队列，是否是同一个等待队列？也就是图中左侧和右侧的是否为同一个队列？

2. notifyAll() 会发通知给等待队列中所有的线程吗？包括那些从未获得过互斥锁的线程吗？

3. 因为wait()被阻塞，而又因为notify()重新被唤醒后，代码是接着在wait()之后执行，还是重新执行 apply 方法？

2019-03-14

作者回复

不是一个队列

只唤醒右侧的队列

wait之后

2019-03-14

蜡笔 14

老师你不用在文章中贴出所有代码嘛，只贴出核心代码，然后把整个例子放在github上，文末的时候给出github的链接，水平基础一般的就可以去上面下载下来跑一跑调试加深印象理解，这样可以不老师

2019-03-12

Geek_e726b7 11

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

2019-03-12

郑晨Cc 10

王老师 ABCD 那个例子真没看懂 线程1释放锁为啥会通知线程4？1和3才是互斥的啊 2和4互斥 按我的理解 3和4 不应该是在同一个等待队列里啊 因为不是通一把锁（准确来时不是同样的两把锁）

就着这个例子 我还有个关互斥锁的等待队列的问题 假设还是资源ABCD 线程5 获取AB 线程6获取CD 线程7试图获取AB 线程8试图获取BC 线程9试图获取CD 那线程 7，8，9 到底是不是在一个等待队列里面，

JVM在实现 wait notify机制是时候到底存不存在真实的队列？

2019-03-12

作者回复

都是this这一把锁: synchronized void apply(){}

所以是一个等待队列

就是500个线程，也是同一个等待队列，因为锁的都是this

队列一定是存在的

2019-03-12

邋遢的流浪剑客

10

wait会释放当前占有的锁，sleep不会释放锁

2019-03-12

陈志凯

7

```
public class Allocator {
private final List<Account> als=new LinkedList<Account>();
// 一次性申请所有资源
public synchronized void apply(Account from, Account to) {
// 经典写法
while (als.contains(from) || als.contains(to)) {
try {
System.out.println("等待用户 -> "+from.getId()+"_"+to.getId());
wait();
} catch (Exception e) {
//notify + notifyAll 不会来这里
System.out.println("异常用户 -> "+from.getId()+"_"+to.getId());
e.printStackTrace();
}
}
als.add(from);
als.add(to);
}
// 归还资源
public synchronized void free(Account from, Account to) {
System.out.println("唤醒用户 -> "+from.getId()+"_"+to.getId());
als.remove(from);
als.remove(to);
notifyAll();
}
}

public class Account {
// actr 应该为单例
```

```
private final Allocator actr;
//唯一账号
private final long id;
//余额
private int balance;
public Account(Allocator actr,long id,int balance){
this.actr=actr;
this.id=id;
this.balance=balance;
}
// 转账
public void transfer(Account target, int amt) {
// 一次性申请转出账户和转入账户，直到成功
actr.apply(this, target);
try {
//TODO 有了资源管理器，这里的synchronized锁就不需要了吧？！
if (this.balance > amt) {
this.balance -= amt;
target.balance += amt;
}
//模拟数据库操作时间
try {
Thread.sleep(new Random().nextInt(2000));
} catch (InterruptedException e) {
e.printStackTrace();
}
} finally {
actr.free(this, target);
}
}
@Override
public int hashCode() {
final int prime = 31;
int result = 1;
result = prime * result + (int) (id ^ (id >>> 32));
return result;
}
/**
* 用于判断两个用户是否一致
*/
@Override
public boolean equals(Object obj) {
if (this == obj)
```

```
return true;
if (obj == null)
return false;
if (getClass() != obj.getClass())
return false;
Account other = (Account) obj;
if (id != other.id)
return false;
return true;
}
public long getId() {
return id;
}
}
```

老师，以上代码是我补的，有个疑问，以上有了Allocator管理器（见TODO部分），transfer方法的this跟target都不再需要加synchronized锁了吧？！

2019-03-13

作者回复

如果只是这个例子就不需要了，

送你俩字！优秀！！！！！

2019-03-13

老杨同志 6

点赞@姜戈 补充一下：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.（都抛出InterruptedException，wait也需要捕获异常）
4. wait()无参数需要唤醒，线程状态WAITING；wait(1000L);到时间自己醒过来或者到时间之前被其他线程唤醒，状态和sleep都是TIME_WAITING

两者相同点：都会让渡CPU执行时间，等待再次调度！

2019-03-12

作者回复

[]

2019-03-12

我是卖报小行家 6

wait和sleep区别

1：wait释放资源，sleep不释放资源

2：wait需要被唤醒，sleep不需要

3：wait需要获取到监视器，否则抛异常，sleep不需要

4：wait是object顶级父类的方法，sleep则是Thread的方法

2019-03-12

作者回复

全面

2019-03-12

lau

5

看评论也能学到很多干货

2019-03-14

陈志凯

5

强烈建议老师每个章节配上完整的demo，包括模拟多线程多个客户操作的代码，这样看效果才是最佳的，我们自己也能根据代码实际好好观察！

2019-03-12

作者回复

对于水平高的，完整的代码没必要。对于水平低的，完整的代码只能增加惰性。我就很讨厌粘贴一些无关的代码

2019-03-12

aksonic

4

老师，我昨天问了你问题后，带着疑问又去学习了下，是不是文章中的左边和右边的两个队列应该改一改名字，不应该都叫等待队列，这样对新手很容易产生误解。如果左边的叫做同步队列，右边的叫做等待队列可能更好。左边的队列是用来争夺锁的，右边的队列是等待队列，是必须被notify的，当被notify之后，就会被放入左边的队列去争夺锁。老师，你觉得呢？

2019-03-14

作者回复

你这个建议挺好，在管程里面，会重新讲这俩队列。现在就知道有俩等待队列就可以了

2019-03-15

高源

4

老师最好讲解每一章的时候配合完整的例子源代码，这样再加调试源代码，印象更深刻了

2019-03-12

作者回复

大家水平不一样，有些高水平的可能只想看到核心的代码，我怕贴多了，有人说浪费流量。自己补上剩余代码也是个不错的提高机会吧

2019-03-12

郭瑞娟

3

之前老师答复问题时，提到wait和notify是一一对应的，如果浪费了一个notify，就必然有一个wait永远没机会被唤醒。这句话怎么理解呢？

例子里面 假设之后线程 1 归还了资源 AB，使用 notify() 来通知等待队列中的线程4 申请的是 CD，程 4 还是会继续等待，此时会执行wait()吗？如果执行了，wait和notify还是一一对应的呀。如果没有执行，线程4会怎么执行呢？我看了几次文章了，还是没有理解此处，请老师帮忙，谢谢。

2019-04-02

作者回复

因为不对应了，所以就死等下去了，有借有还才行，还错了人就出问题了

2019-04-02

亮 3

3怎么可能永远通知不到呢？就算4通知到了不满足条件等待，2走完还是会通知3或者4，就算通知到4了还是会点用notify方法

2019-03-24

作者回复

一个notify对应一个wait，浪费一个wait，自然有一个永远失去机会

2019-03-24

San D Ji 3

学习这几章以后，我一直有一个问题，Javaweb端在什么样的业务场景下需要多线程的技术实现？

一直以为Javaweb端都是接收到一个请求服务器端开启一条线程独立作业，完了之后就返回一个应答。

不知道老师能否回答一下我的疑问？

2019-03-12

作者回复

比如你要做个数据库连接池，做个httpclient，做个rpc框架，用批处理处理上千万数据，一个简单的crud真的用不上

2019-03-12

^_^ 3

老师，while(als.contains(from) || als.contains(to)) 这句对吗

2019-03-12

作者回复

我确认了一下，应该是对的

只要有一个，就说明曾经被分配过

2019-03-12