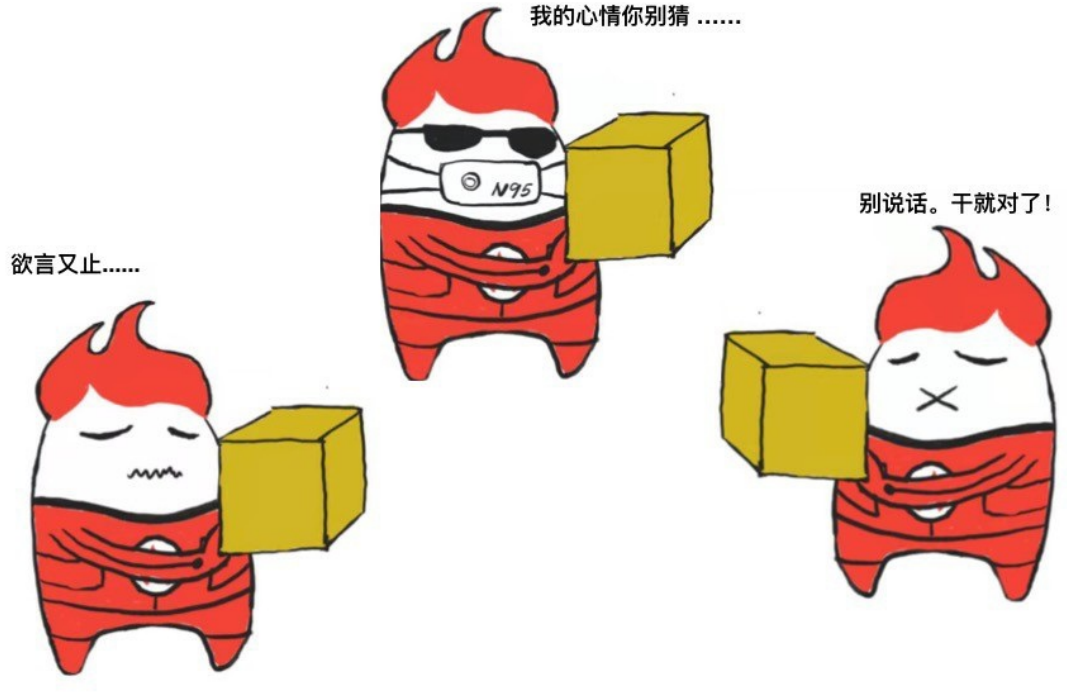
08 集体协作，什么最重要？沟通！—线程的等待和通知

更新时间：2019-09-25 15:29:02

> 世界上最宽阔的是海洋，比海洋更宽阔的是天空，比天空更宽阔的是人的胸怀。
>
> ——雨果

通过前面几节的学习，我们了解了在 Java 中如何启动一个线程，并且学习了 Thread 类的 API 以及线程的状态。假如将多线程比作多个机器人一起工作，那么我们讲到现在，所生产的机器人确实能够担当干活的责任了。每个机器人各司其职，尽职尽责地去完成自己的工作。他们每个人不断去查看自己的任务列表，有了新的任务就去工作，没有的话会持续查看。OK，这样没有问题，机器人能够一起把工作干完。但是你不觉得缺了点什么吗？或者说你不觉得机器人少了什么器官？没错，嘴巴！这个过程太安静了，居然没有机器人说话！这在现实世界是不可想象的。在现实世界里，即便最简单的两人配合工作都需要沟通和交流。



我们回忆一下之前学生抄写单词的例子。那个例子中抄写的次数提前预置，每个学生抄写前领取一次抄写的任务，然后更新剩余抄写次数，直到所有抄写次数全部完成。这个例子比较简单，但是假如今天老师发飙了，除了抄写 internationalization 这个单词，她还会一直给你新的单词抄写作业。此时，作为学生在抄写完 internationalization 后他有两个选择。一是不停的盯着老师，直到老师发给他新的作业。二是他先休息一会，老师准备好新的作业时再叫他。假如我是学生，我肯定选择第二种。因为第一种太累了，要一直盯着老师。第一种方式在程序中叫做轮询。而第二种方式就引出了我们本节要讲解的 wait/notify。

1. wait/notify 概念

我们先从概念上初步了解 wait/notify。原本 RUNNING 的线程，可以通过调用 wait 方法，进入 BLOCKING 状态。此线程会放弃原来持有的锁。而调用 notify 方法则会唤醒 wait 的线程，让其继续往下执行。

你可能有疑问，既然这两个方法都和线程有关系，为什么没有放在上一节线程 API 中讲解呢？这是因为这两个方法并不在 Thread 对象中，而是在 Object 中。也就是说所有的 Java 类都继承了这两个方法。所有 Java 类都会继承这两个方法的原因是 Java 中同步操作的需要。

2、同步

讲到这里，我们必须要对线程同步有所了解。那么什么是线程同步呢？我们先看看什么是异步，异步其实就是指多个线程同时执行。但在多个线程同时执行的过程中，可能会访问共享资源，此时我们希望确保多个线程在同一时间只能有一个线程访问，此时就称之为线程同步。

在多线程开发中最基本的同步方式就是通过 synchronized 关键字来实现。第三节中我们单词抄写的程序并没有彻底解决线程安全问题，仍旧可能出现重复抄写。这是因为我们对抄写次数这个共享资源的访问没有做同步。现在我们使用 synchronized 关键字对抄写单词的核心逻辑进行改写，如下：

```
while (true) {
    int leftCopyCount = 0;
   //在同步代码块中访问punishment，确保读取和更新数量时，只有一个线程访问到共享资源
    synchronized (punishment){
        if (punishment.getLeftCopyCount() > 0) {
            leftCopyCount = punishment.getLeftCopyCount();
            punishment.setLeftCopyCount( leftCopyCount - 1);
        }
    }

    if(leftCopyCount>0){
        System.out.println(threadName + "线程-" + name + "抄写" +  leftCopyCount + "。还要抄写" + leftCopyCount - 1 + "次");
        count++;
    }else{
        break;
    }
}
```

原来的代码如下：

```
while (true) {
    if (punishment.getLeftCopyCount() > 0) {
        int leftCopyCount = punishment.getLeftCopyCount();
        if (leftCopyCount == punishment.getLeftCopyCount()) {
            punishment.setLeftCopyCount(leftCopyCount - 1);
            System.out.println(threadName + "线程-" + name + "抄写" + punishment.getWordToCopy() + "。还要抄写" + leftCopyCount-1 + "次");
            count++;
        }
    } else {
        break;
    }
}
```

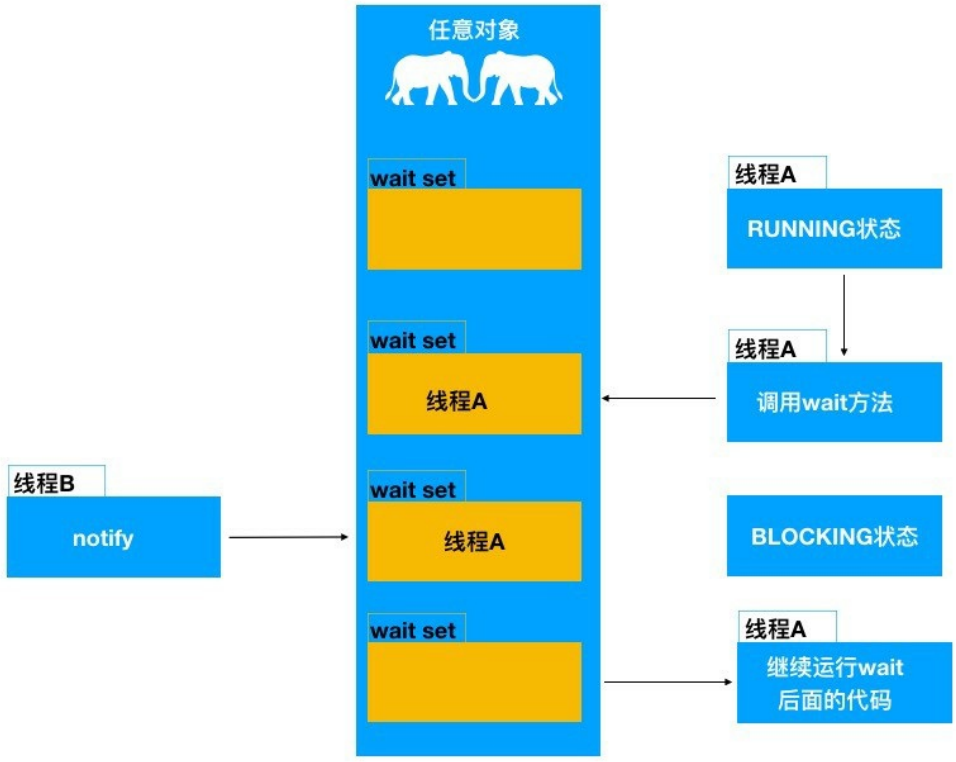
可以做一下比较。修改后的代码中，读取和更新 leftCopyCount 的两步操作放在了 synchronized 代码块中，在此代码块中的代码会确保同一时间只有一个线程能够执行。这也称之为加上了锁，只有获取锁的线程才能执行同步代码，执行完成后则会释放锁。因此在 synchronized 代码块中操作 punishment 是安全的。当前线程取出来的 leftCopyCount 值，在同步代码块结束前，也就是 set 回去前，并不会被其它线程所改变。所以它并不需要像原来代码那么啰嗦，取出来后更新前还要再比较一次。其实原来代码即使又做了比较，也无法 100% 确保更新操作前没有被别的线程修改。这在第三节的实验中已经得到证实。正确的编写方式应该把共享资源的操作放在 synchronized 代码块中，这样才能 100% 确保程序的正确性。

注意修改后的代码，并没有把输出抄写内容放到 synchronized 代码块中。因为这一步操作其实和共享资源已经无关，所以没必要再持有锁，这会延长其它线程等待锁的时间，降低了并行代码的效率。这在我们实际开发中要注意，尽量把不需要同步的代码移出 synchronized 代码块。

3、使用 wait/notify

了解完 synchronized，我们再回头看 wait 操作。synchronized 关键字需要配合一个对象使用，其实这个对象可以是任何对象，只不过为了代码好懂，这里使用了共享资源对象 punishment，语义上表示对该对象上锁，但你换成其它任何对象一样是可以的。

其实 synchronized 所使用的对象，只是用来记录等待同步操作的线程集合。他相当于一位排队管理员，所有线程都要在此排队，并接受他的管理，他说谁能进就可以进。另外他维护了一个 wait set，所有调用了 wait 方法的线程都保存于此。一旦有线程调用了同步对像的 notify 方法，那么 wait set 中的线程就会被 notify，继续执行自己的逻辑。



这也解释了为什么 synchronized 的对象并不一定是共享资源对象。这个对象只是看门人，确保同步代码块中的代码只有一个线程能够进入执行，但这个看门的工作并不一定要共享资源对象来做。任何对象都可担当此工作。

需要注意的是，我们对哪个对象做了 synchronized 操作，那么就只能在同步代码块中使用此对象进行 wait 和 notify 的操作。这也很好理解，只有当看门人在听你讲话时，他才能按你的要求去做事情。我们只有获得了和同步对象的对话权，这个对象才能听此线程的命令。无论是请求加入 wait set 还是要通知 wait set 中的线程出来，均是如此。

wait 和 notify 示例代码如下：

```
synchronized (punishment){
//do something
punishment.wait();
//continue to do something
}
```

假如此段代码在 A 线程中。这段代码会在执行一些逻辑后把 A 线程放入 punishment 对象的 wait set 中，并且 A 线程会释放持有的锁。

我们再看看另外一个 B 线程中的部分代码：

```
synchronized (punishment){
//do something
punishment.nofity();
//continue to do something
}
```

这段代码会 notify 在 punishment 对象的 wait set 中的一个线程，将其弹出。比如此时 A 线程在 wait set 中，那么 A 线程将被弹出。被弹出的 A 线程会在获取 CPU 资源后继续执行 wait 方法后面的逻辑。

最后再说一下 notifyAll 方法。我们知道 notify 可以唤醒 wait set 中的一个线程，但是如果 wait set 中存在多于一个线程时，我们并无法控制哪个线程被唤醒。假如所有线程的执行逻辑都是一样的，那么无所谓谁被唤醒，因为都是干一样的工作。

但如果是本节前面提出的问题，一个老师线程负责留作业，一个学生线程负责写作业。假如此时开启了多个学生线程，当学生写完作业后本来需要通知老师留作业，但被 notify 的并不一定的是老师线程，也可能 notify 了其他学生线程。

为了解决这个问题，我们可以使用 notifyAll 来唤醒所有在此对象的 wait set 上的线程。而获得锁的线程是否真的需要做什么工作是由自己控制的。如果学生线程先抢到 CPU 资源，但是由于作业列表为空，他又会选择 wait 进入 wait set。此时他会释放锁。而老师线程此时会获得锁，在看到作业列表为空后，则会添加新的作业。通过 wait/notifyAll 让多个线程交互，同时通过共享资源的状态，各线程控制自己的逻辑。这样的程序称之为状态驱动程序。也就是说是否真的执行逻辑，是由状态值所决定的。如果状态不满足，即使被 notify 了，也会再次进入 wait set。

4、总结

本节首先简单介绍了同步的概念，然后讲解了如何通过 wait 和 notify 实现多线程间的沟通和协调。讲了这么多，其实不如写一写代码，更容易理解。在下节中我们将采用生产者 / 消费者模式，来开发一个多位老师留作业，多个学生一起完成的作业的程序。届时，我们本节学习的内容都会被使用上。

}

07 深入Thread类—线程API精讲

09 使用多线程实现分工、解耦、缓冲—生产者、消费者实战