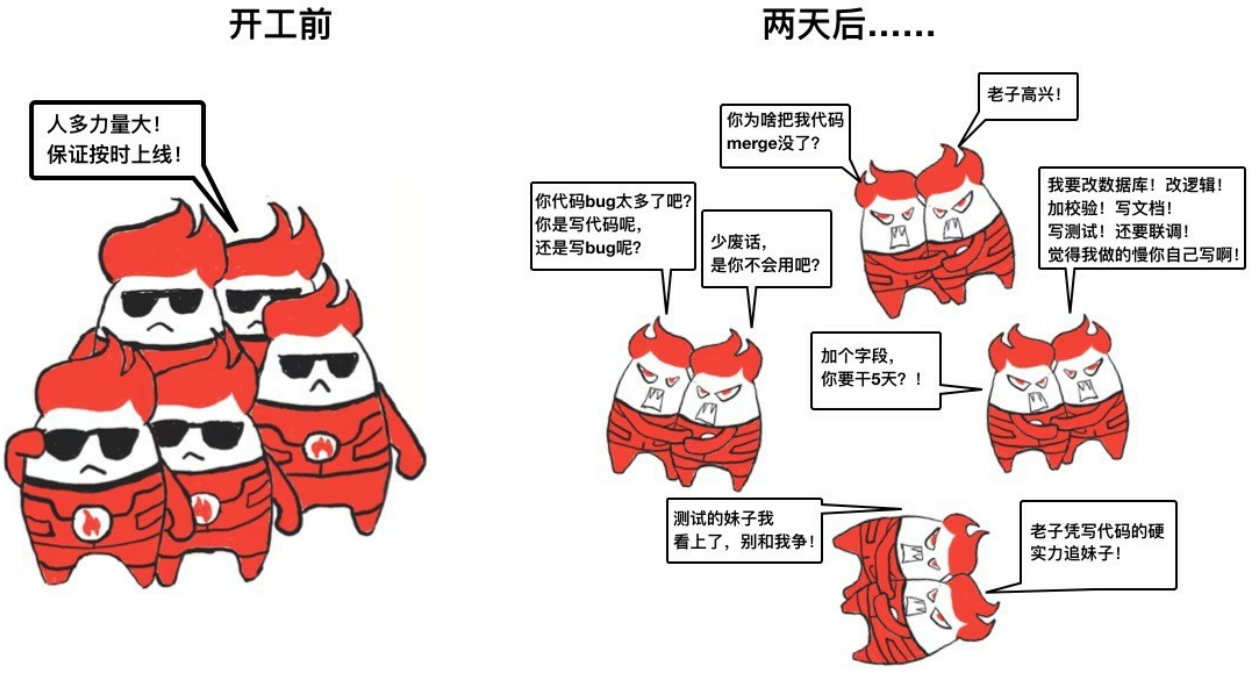
04 人多力量未必大—并发可能会遇到的问题

更新时间：2019-09-09 19:19:49

> 不安于小成，然后足以成大器；不诱于小利，然后可以立远功。
>
> ——方孝孺

专栏已经写到第三篇，但我们貌似还没体验到多线程带来的好处，反倒是惹出了一些麻烦。上一篇专栏中，我们试图采用多线程并发，加快抄写单词的速度，但是事与愿违，不但没有提升，还做了无用功。最糟糕的是程序执行结果是错误的。人多虽然力量大，但也会有各种问题和麻烦。



如上图，如果问题处理得不得当，人越多反而会越乱。上一节并发抄写单词程序的问题不知大家思考的如何。其实很简单，问题出在共享资源的访问上。

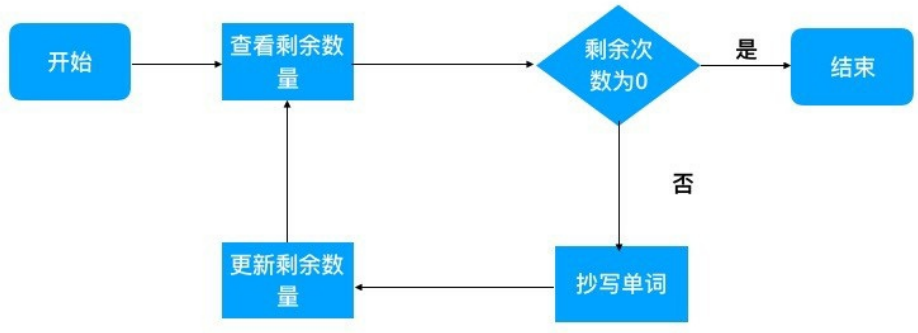
并发抄写单词问题分析

回到抄单词这个问题上，我们试图引入更多的学生来一块完成任务，那么这些学生怎么知道目前抄写多少单词了？自己是否还需要继续抄写呢？我们看相关代码：

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

我用通俗的方式来说明这段代码的逻辑。为了让参与抄写单词的学生知道剩余抄写的数量，我们找来了一块小黑板，然后把剩余的总量写在上面，每个学生抄写之前先看一眼黑板，如果剩余的数量大于零，那么还需要继续抄写，抄写完后，擦掉黑板上的数字，把剩余数量-1，写上去。

流程图如下：



OK，一个人按照这个流程抄写是没问题的，但是多个人同时抄写，问题就多了。

1. 读取次数和抄完更新次数之间有时间间隔，此时别的学生也会读到同样的剩余次数，那么这次抄写就是多余的；
2. 在更新leftCopyCount的时候，可能其它多个线程已经更新过了，也就是说此时leftCopyCount并不是你当初取出来的值，那么可能会把剩余数量更新的比此时还要大。这样其它线程的抄写就白做了。因为剩余数量被更新了回去。

尝试解决并发问题

为了解决这两个问题我们修改copyWord方法代码如下：

```
public void copyWord() {
    int count = 0;
    String threadName = Thread.currentThread().getName();

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

    System.out.println(threadName+"线程-"+name + "一共抄写了" + count + "次！");
}
```

可以看到代码中主要有两个变化：

1. 取得剩余次数后马上更新-1后的次数。看似是避免了读取和更新间的时间间隔。
2. 更新剩余次数前先判断自己的更新次数是否为最新，避免更新后次数反而变大的问题。

这么修改后看起来好像没有问题了，那么我们再来试一下。

执行以下main方法：

```
public static void main(String[] args) {
    Punishment punishment = new Punishment(100,"internationalization");

    Student xiaoming = new Student("小明",punishment);
    xiaoming.start();

    Student xiaozhang = new Student("小张",punishment);
    xiaozhang.start();

     Student xiaozhao = new Student("小赵",punishment)
     xiaozhao.start();
}
```

我们启动三个线程并发抄写。可以在输出中找到如下关键信息：

```
小赵线程-小赵一共抄写了48次！
小明线程-小明一共抄写了25次！
小张线程-小张一共抄写了27次！
```

总数是100，问题解决了！等等，真的解决了吗？我们回过头再看改后的copyWord代码，虽然程序读取剩余次数后，马上更新，并且加了小于才更新的判断。但是仔细想想，这样并不是万全之策，因为小明和小张很可能恰巧同时去看剩余次数，取得剩余次数n后，各自计算剩余次数为n-1，但是假如小明正好计算的快一点，小明先把剩余次数更新为了n-1，虽然小张不符合更新条件，但是在剩余第n次的这次抄写上，小明和小张各抄写了一次，也就是说多抄写了一次。

线程安全

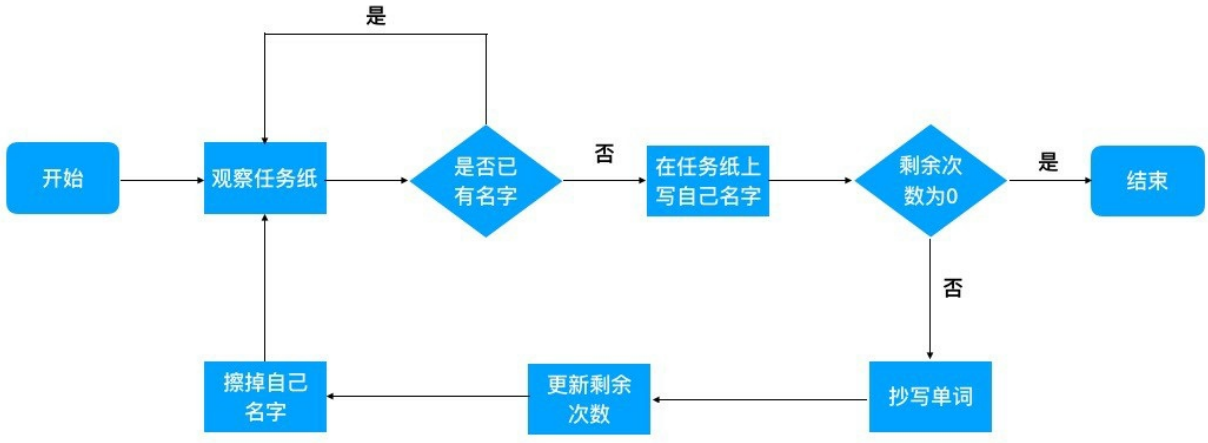
为什么上面代码打出的日志中，三人抄写总和是正确的100呢？有没有可能是抄写数量太小，全部抄写完也没有发生上面描述的两人同时去查看剩余次数的情况？为了验证这个推论，我们把抄写次数增多，看是否会出现问题。

在我的电脑上，抄写数量加大到1000，三人抄写总和依然是正确的。但加大到10000时，问题出现了，有时会出现三人抄写次数大于10000的现象。我继续加大到1000000，此时基本每次执行，三人执行总和都要超出1-5次。以上实验结果，根据实验电脑的不同会有所区别。现在已经能够得出结论了，这样修改是不行的，原因前文已经说明，因为有小概率两人甚至三人同时查看剩余次数，导致重复抄写。

以上所描述的问题，就是大家耳熟能详的线程安全问题。线程安全问题来源于并发时对共享资源的操作。在本例中，我们把剩余次数写在黑板上，大家都去黑板上读取剩余次数并更新。那么共享资源就是黑板上的剩余抄写次数。

我们先不谈代码如何修改，我们来想一想现实生活中如何解决上述问题。

问题出在小明读取剩余次数的同时，小张、小赵也可以读取，三人很可能读到同样的次数。并且读取完，三人都会根据自己的计算去更新剩余次数，所以才会乱了套。我们可以改为谁要读取次数时先做个标记（比如在次数边上写上自己名字），代表自己在操作，此时别人只能等待。详细流程如下图：



1、读取剩余次数前，先看纸上是否有名字。没有名字，在纸上写上自己的名字；

2、如果纸上已经有名字则等待，并且一直观察纸上名字是否被擦除；

3、成功写上自己名字的同学，更新次数为n-1；

4、擦掉自己的名字；

5、其他等待者观察到名字被擦掉，则抢着写上自己的名字。

这样确保了同一时间只有一个人在操作剩余次数，再也不会乱套了。

其它多线程相关概念

以上流程引入了多线程中的一个重要概念–**同步**。所谓的同步就是某一段流程同时只能有一个线程执行，其它线程需要等待。对于本例，读取剩余次数，并更新剩余次数这两步操作需要做同步控制。操作剩余次数之前需要写名字代表自己在做操作，这是在**加锁**。而擦除名字则是**释放锁**。假如小明先成功写上自己的名字，而小张和小赵按照先来后到的顺序排队，那么就是**公平锁**。但假如两人并不排队，而是通过争抢获取写名字的权利，那么这就是**非公平锁**。在这种情况下，如果小张很瘦弱，既抢不过小赵，也抢不过小明，那么小张永远无法读取剩余次数，也就无法抄写单词，这种情况就叫做**线程饿死**。

线程安全问题通过加锁可以得到解决。Java也提供了特定场景下更为轻量级的解决方法。后面的文章会有更为详尽的描述。本篇文章只是抛出了多线程开发中，可能存在的问题，及问题产生的原因。上文还通过例子引申出多个多线程中的概念，相信结合例子都很容易理解，后面的文章还会反复提及并有专门的详解。

总结

多线程开发，复杂就复杂在处理线程安全问题上。如果代码写得不好，就会有各种同步相关的问题产生，而且很难调试。不过好在我们有很多种方式能够解决。后面的文章会逐一讲解。

专栏写到这里，其实我们学习多线程的两个主要目标已经清楚：

1. 如何实现多线程；
2. 如何解决线程安全问题。

后面的文章都会围绕这两个问题进行讲解。我们会先学习如何实现多线程，再看如何解决多线程中的问题。难点在第2点上。要想知道如何解线程安全问题，就要深刻理解问题产生的根本原因是什么。另外要深刻理解解决问题的原理，而不仅仅是记住解决方法。这样才能做到一通百通，遇到问题灵活应对。其实我们学习每一样技术都是同样的道理，千万不要只停留在会用层面，否则遇到问题只会从表面现象入手，但n种表面现象可能是同一个根本原因。如果我们了解原理，遇到再多问题也无所畏惧。因此，本专栏绝不会停留在使用层面，而是会深入到底层原理。前三篇只算是个开胃菜。目的让读者了解多线程的概念及简单实现，同时知晓多线程存在的问题。后面的文章难度会越来越大，不过不用担心，只要跟着专栏认真学习下来，你一定能轻松掌握。

}

03 多线程开发如此简单—Java中如何编写多线程程序

05 看若兄弟，实如父子—Thread和Runnable详解