09 使用多线程实现分工、解耦、缓冲—生产者、消费者实战

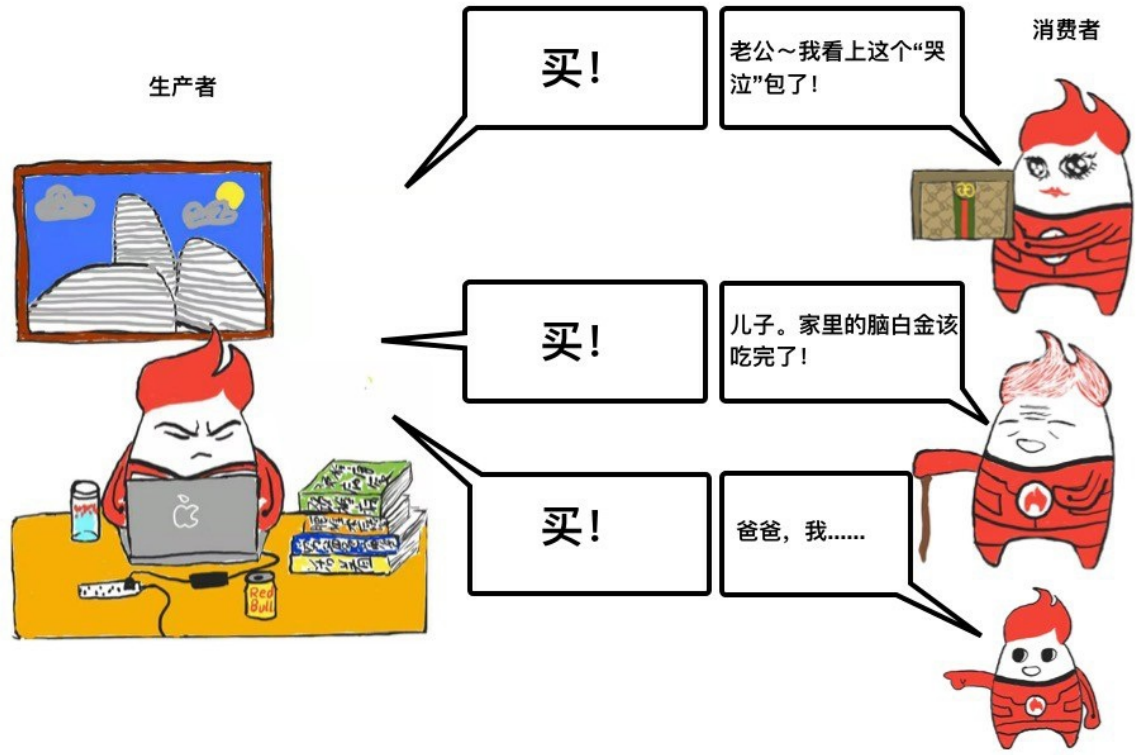
更新时间：2019-09-26 09:35:09

> 不要问你的国家能够为你做些什么，而要问你可以为国家做些什么。
>
> ——林肯

前面讲了那么多，其实我们一直还没有体会到多线程带来好处，甚至还因为使用多线程引来了一些麻烦。不过好在上一节中，我们通过同步操作，解决了抄写单词程序的线程安全问题。在本节中，我们通过一个经典的多线程设计模式–生产者 / 消费者，来体验多线程开发及其带来的好处。

1. 生产者 / 消费者介绍

生产者 / 消费者是一种经典的设计模式，被广泛应用于软件领域。生产者 / 消费者设计模式能够充分解藕，每一方只需要关注自己的职责。生产者专注生产，消费者专注消费。双方通过某种机制进行资源共享。举个例子，就像男人负责赚钱养家，女人负责貌美如花。男人可以看作生产者，不停赚钱存入银行，而老婆负责花钱买生活用品等。银行账户就是共享金钱的机制。



2. 需求分析

我们在写代码前，需要先把需求理解清楚。我们的需求是这样的：

1. 有两个角色，老师和学生。但每个角色可能有多个人。
2. 老师负责留抄写单词的作业，学生负责完成作业。一项作业是指抄写指定单词指定次数。
3. 每个学生领取一项作业，独立完成。
4. 当没有作业可领取的时候，学生等待。
5. 当有了新的作业可领取，学生被唤醒，继续领取作业。
6. 当作业数量达到上限的时候，老师停止布置作业。
7. 当作业小于上限的时候，老师被唤醒，继续布置作业。

这是一个典型的生产者 / 消费者的场景。老师负责生产作业，学生负责消化作业。当不满足某些条件的时候两种角色都可能休眠，而在特定条件下则会被唤醒。

3. Teacher 类代码

我们先看 Teacher 的全部代码：

```
public class Teacher extends Thread {
    private String name;
    private List<String> punishWords = Arrays.asList("internationalization", "hedgehog", "penicillin", "oasis", "nirvana", "miserable");
    private LinkedList<Task> tasks;
    private int MAX = 10;

    public Teacher(String name, LinkedList<Task> tasks) {
        //调用Thread构造方法，设置threadName
        super(name);
        this.name = name;
        this.tasks = tasks;
    }

    public void arrangePunishment() throws InterruptedException {
        String threadName = Thread.currentThread().getName();

        while (true) {
            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");
                }
            }
        }
    }

    //重写run方法，完成任务。
    @Override
    public void run() {
        try {
            arrangePunishment();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private String getPunishedWord() {
        return punishWords.get(new Random().nextInt(punishWords.size()));
    }
}
```

我们看核心的 arrangePunishment () 方法，在这个方法中会进入自旋，当任务数量未达 MAX 值时，teacher 会生成新的 task 加入任务列表，并唤醒所有线程。这是因为可能有学生线程因为无 task 可做已经 wait，所以需要唤醒。而当任务数量超出 MAX，当前老师线程则会进入 wait set。

4. Student 代码

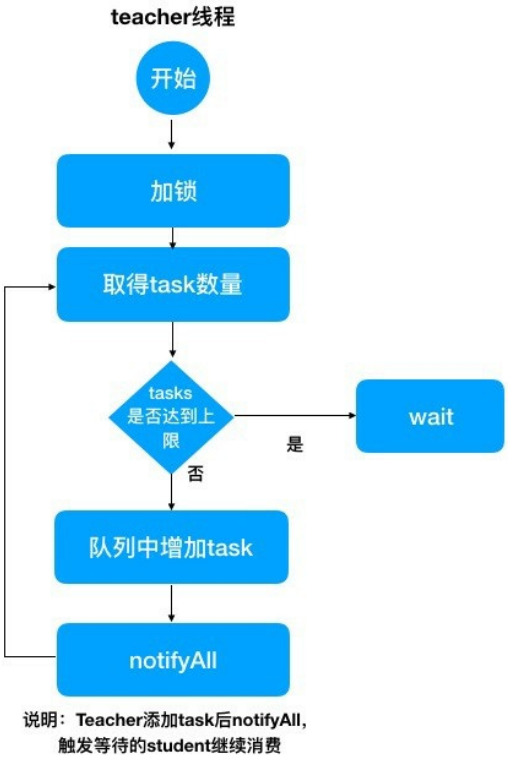
我们再看学生的全部代码：

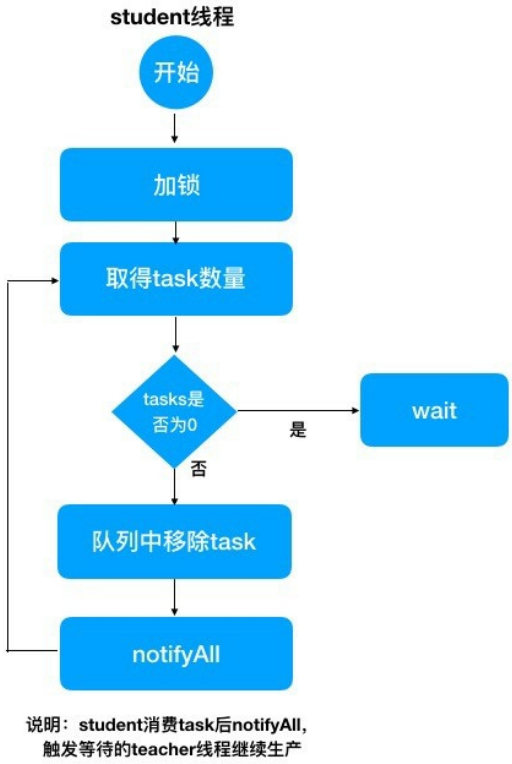
```
public class Student extends Thread {
    private String name;
    private LinkedList<Task> tasks;

    public Student(String name, LinkedList<Task> tasks) {
        //调用Thread构造方法，设置threadName
        super(name);
        this.name = name;
        this.tasks = tasks;
    }

    public void copyWord() throws InterruptedException {
        String threadName = Thread.currentThread().getName();

        while (true) {
            Task task = null;

            synchronized (tasks) {
                if (tasks.size() > 0) {
                    task = tasks.removeFirst();
                    sleep(100);
                    tasks.notifyAll();
                } else {
                    System.out.println(threadName+"开始等待");
                    tasks.wait();
                    System.out.println("学生线程 "+threadName + "线程-" + name + "等待结束");
                }
            }

            if (task != null) {
                for (int i = 1; i <= task.getLeftCopyCount(); i++) {
                    System.out.println(threadName + "线程-" + name + "抄写" + task.getWordToCopy() + "。已经抄写了" + i + "次");
                }
            }
        }
    }

    //重写run方法，完成任务。
    @Override
    public void run() {
        try {
            copyWord();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

我们也直接看核心方法 copyWord ()，在这个方法同样会进入 while 自旋，如果任务的数量大于 0，则会取出第一个任务，并从任务列表移除，同时会唤醒所有线程。这里 notifyAll 这是因为可能有 teacher 线程由于触及 MAX 值而进入了 wait set。当此次消费完成后，任务列表中任务数量会小于 MAX 值，所以需要唤醒 teacher 线程继续生成新的 task。当 task 数量为 0 时，学生线程则会进入等待，因为无 task 可做。

学生的同步代码块中让线程 sleep 了 100 毫秒，这样是为了让 teacher 生产 task 的速度更快，以让 task 数量能够达到 MAX 值，触发 teacher 线程的 wait 操作。以便我们看到 wait 和 notifyAll 的效果。

以上程序代码逻辑，可参考下图理解：





5. 客户端代码

最后我们来看客户端代码：

```
public class WaitNotifyClient {
    public static void main(String[] args) {
        LinkedList<Task> tasks = new LinkedList<>();

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

        Student xiaowang = new Student("小王", tasks);
        xiaowang.start();

        Teacher lilaoshi = new Teacher("李老师", tasks);
        lilaoshi.start();

        Teacher zhanglaoshi = new Teacher("张老师", tasks);
        zhanglaoshi.start();
    }
}
```

我们声明了两个学生线程写作业，两个老师线程留作业。

接下来我们运行客户端代码，看一下输出。

我截取了部分输出如下：

```
小明开始等待
小王开始等待
李老师留了作业，抄写hedgehog 1次
李老师留了作业，抄写nirvana 2次
李老师留了作业，抄写miserable 2次
李老师留了作业，抄写nirvana 2次
李老师留了作业，抄写nirvana 3次
李老师留了作业，抄写nirvana 2次
李老师留了作业，抄写nirvana 3次
李老师留了作业，抄写nirvana 3次
李老师留了作业，抄写penicillin 3次
李老师留了作业，抄写oasis 1次
李老师开始等待
学生线程 小王线程-小王等待结束
学生线程 小明线程-小明等待结束
小王线程-小王抄写hedgehog。已经抄写了1次
小明线程-小明抄写nirvana。已经抄写了1次
张老师留了作业，抄写hedgehog 1次
张老师留了作业，抄写miserable 3次
张老师开始等待
```

整个过程如下：

1. 学生线程先启动后，由于没有作业，开始等待。
2. 李老师线程启动后开始留作业，学生线程由于有了 task 被唤醒。
3. 而李老师线程由于 task 数量达到上限，开始等待。
4. 学生消费作业的同时，张老师线程也启动完成，开始继续留作业。
5. 此时作业数量又达 MAX 值。老师线程开始等待。

如果你本地运行了以上代码，还能看到更多有规律的输出，每当学生消费掉一个 task，那么就有一个老师的线程被唤醒。这里注意由于线程输出时会有时间差，所以在顺序上和我们想象的顺序会有所区别，但这并不代表程序运行是错误的。

我们对以上代码做一个总结。学生和老师线程都会操作任务列表 tasks，所以对 tasks 的操作需要加上同步保护。任务达到 MAX 上限时，老师 wait，当任务数量为 0 时，学生休息。无论学生和老师，在执行完自己正常生产和消费逻辑后都会执行 notifyAll，确保如果有 wait 的线程，能被唤醒。

6. 使用多线程的好处

采用生产者 / 消费者的多线程方式带来很多好处，我列了主要几条如下：

1. ** 更贴近真实世界。** 在真实世界里，每个人都像一个单独的线程，在并行工作。
2. ** 解耦。** 多线程之间不需要互相调用，并且不需要知道其他线程运行的情况。通过共享资源 tasks，让不同线程维系在一起工作。
3. ** 缓冲。**tasks 列表其实就是一个缓冲池，能够缓解生产和消费的速度不一致。有了 task 列表，我们就不需要运行完一个 task，才能继续生成下一个 task。本文的例子比较理想化，生成和消费 task 速度是一样的。但实际情况可能并不是这样，假如生产 task 速度开始比较快，那么会先把 task 列表装满。此时如果生产 task 速度慢下来了，消费者还是可以持续消费 task 列表中已有的 task，而不会停下来等待。这大大提高了程序的整体效率

7. 本章总结

本节是第二章《如何编写多线程》的最后一节。通过本章学习，我们了解了实现多线程的方式；学习了线程的状态和 API；讲解了线程间是如何通讯的。最后我们通过生产者和消费者的代码实践，对多线程开发有了更为直观的认识。这一章我们重点围绕着多线程的实现做讲解。不过在这个过程中，我们也接触到了线程安全问题。最后通过同步，我们保证了线程安全。其实在多线程开发的过程中，非常重要的一环就是确保线程安全，否则你的多线程程序是无法正常工作的。下一章我们将会讲解多线程开发中可能遇到的各种各样的问题以及问题产生的原因。只有解决了这些问题，多线程才能为我所用。否则，我们空握多线程这把利器，不但无法发挥出他的威力，还会误伤自己。

}

08 集体协作，什么最重要？沟通！—线程的等待和通知

10 有福同享，有难同当—原子性