参考：

<http://ifeve.com/fork-join-1/> 【并发编程网，关于fork-join框架的系列，是对Java7 Concurrency Cookbook的翻译】

http://ifeve.com/talk-concurrency-forkjoin/

# 引言

通常，当你实现一个简单的并发应用程序，你需要实现一些Runnable对象和相应的Thread对象。在程序中，你控制这些线程的创建、执行并管理线程的状态。

Java5引入了Executor和ExecutorService接口及其实现类进行了改进，执行者框架将任务的创建与执行分离，有了它，你只要实现Runnable对象(或者Callable对象)，并使用Executor对象。你提交Runnable任务给Executor，由执行器创建、管理线程来执行这些任务。

Java7更进一步，包括一个面向特定问题(即可以采用分治法处理的问题)的ExecutorService接口的额外实现(即ForkJoinPool)，它就是Fork-Join框架。

# 什么是Fork-Join框架

Fork/Join框架是Java7提供了的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

我们再通过Fork和Join这两个单词来理解下Fork/Join框架，Fork就是把一个大任务切分为若干子任务并行的执行，Join就是合并这些子任务的执行结果。

我们已经很清楚Fork/Join框架的需求了，那么我们可以思考一下，如果让我们来设计一个Fork/Join框架，该如何设计？这个思考有助于你理解Fork/Join框架的设计。

第一步分割任务。首先我们需要有一个fork类来把大任务分割成子任务，有可能子任务还是很大，所以还需要不停的分割，直到分割出的子任务足够小。

第二步执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程从队列里拿数据，然后合并这些数据。

Fork/Join使用两个类来完成以上两件事情。

ForkJoinTask：我们要使用ForkJoin框架，必须首先创建一个ForkJoin任务。它提供在任务中执行fork()和join()操作的机制，通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类。

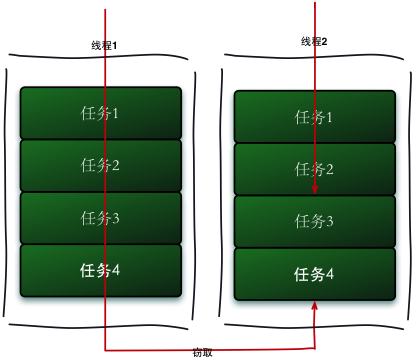
RecursiveAction用于没有返回结果的任务。

RecursiveTask用于有返回结果的任务。

ForkJoinPool：ForkJoinTask需要通过ForkJoinPool来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部；当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务，即work-stealing。

# 工作窃取算法

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。工作窃取的运行流程图如下：



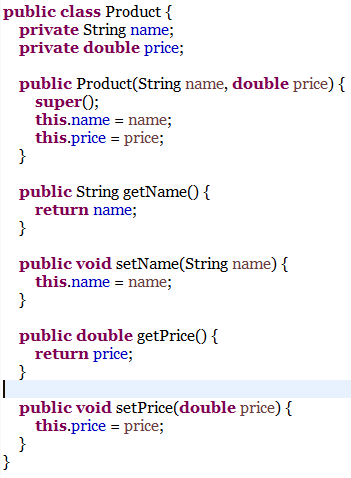
那么为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，我们可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，于是把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应，比如A线程负责处理A队列里的任务。但是有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点是充分利用线程进行并行计算，并减少了线程间的竞争，其缺点是在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且消耗了更多的系统资源，比如创建多个线程和多个双端队列。

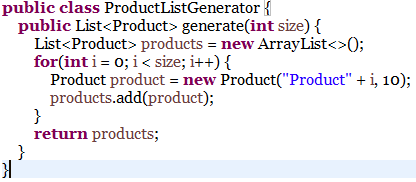
# Fork-Join框架实例一：没有返回值的任务

在这个案例中，你实现一个任务来修改产品列表的价格。任务的目的是负责更新一个队列中的所有元素，你将会使用10作为参考大小，如果一个任务要更新的元素超过10个，则这些元素将被划分成两个部分，并创建两个任务来更新每个部分中的产品的价格。

首先实现一个Product类，用来存储产品的名称和价格。



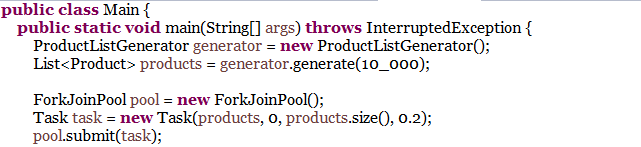
接着实现一个ProductListGenerator类，用来产生随机产品的数列。



再实现一个继承了RecursiveAction的Task类。



最后创建一个Main类，运行主程序。





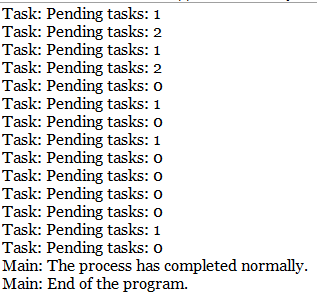
它是如何工作的…

在这个示例中，你创建了一个ForkJoinPool对象和一个在池中执行的ForkJoinTask类的子类。为了创建ForkJoinPool对象，你使用了无参构造器，所以会以默认的配置来执行，即会创建一个线程数等于计算机处理器数的池。当ForkJoinPool对象被创建时，这些线程被创建并且在池中等待，直到有任务到达让它们执行。

由于Task类没有返回结果，所以它继承RecursiveAction类。如果这个任务更新的产品个数超过10个，则将任务分隔为两个子任务，然后调用invokeAll()方法，执行每个子任务。这是一个同步调用，这个任务在继续执行之前，必须等待子任务的结束，并且当该任务正在等待它的子任务结束时，正在执行该任务的工作线程会执行其他正在等待的任务(work stealing)。在这种行为下，Fork/Join框架比Runnable和Callable对象本身提供一种更高效的任务管理。

你使用submit()方法提交任务给这个池，在这种情况下，它是一个异步调用，而主线程会继续它的执行。

你使用ForkJoinPool类的一些方法，用来检查正在运行任务的状态和变化。



最后，与执行者框架一样，你应该使用shutdown()方法结束ForkJoinPool，该方法会将已经提交的任务执行完毕后才结束。

# Fork-Join框架实例二：有返回值的任务

在这个案例中，你开发一个在文档中查找单词的应用程序。你将会实现以下两种任务类型：

一个文档任务，从文档的行集合中查找一个单词；

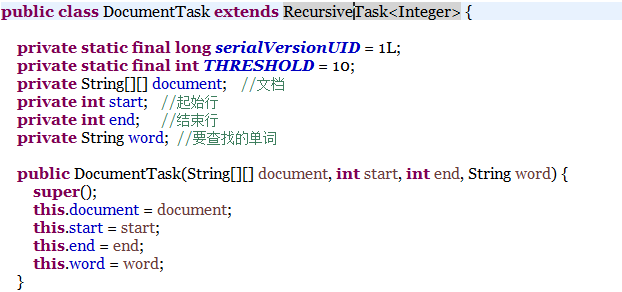
一个行任务，从一行的单词集中查找一个单词。

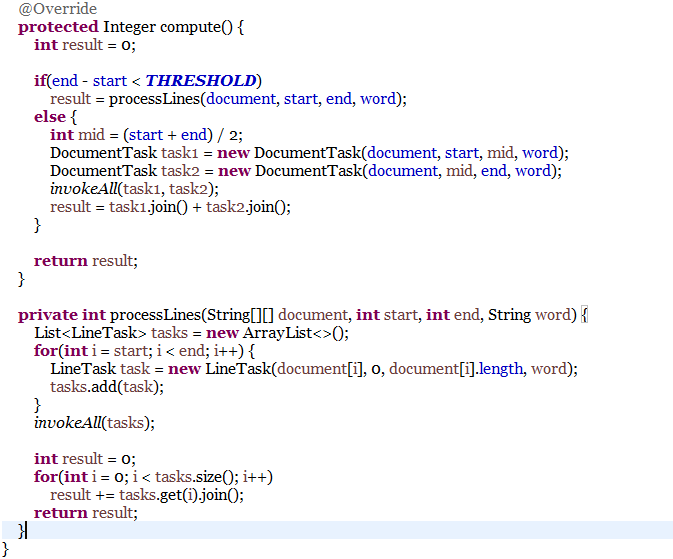
所有这些任务将返回单词在文档中出现的总次数。

首先创建一个DocumentGenerator类，用于随机生成一个文档(由行集合组成)。



接着创建一个继承RecursiveTask类的DocumentTask类，用于统计单词在文档中出现的总次数。

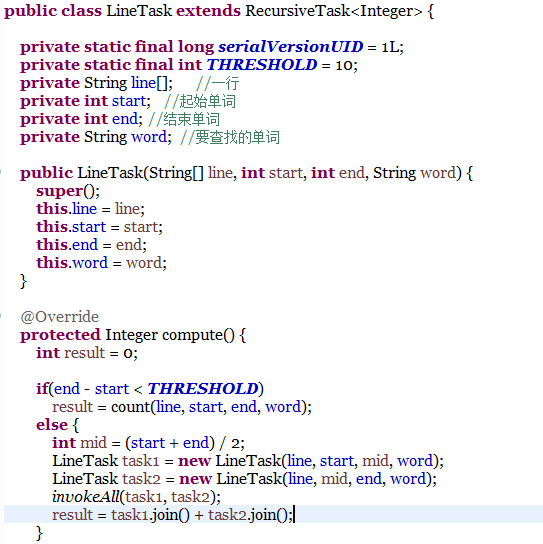


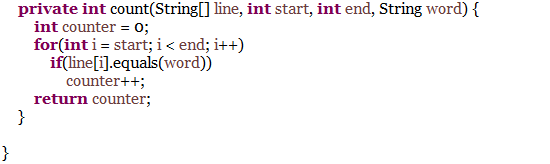


在DocumentTask的compute()方法中，如果行集合的结束位置end与起始位置start的差值大于阈值，则将行集合分隔为两个子任务，并调用invokeAll()方法来执行子任务，这是一个同步调用，这个任务在继续执行之前，必须等待子任务的结束，并且当该任务正在等待它的子任务结束时，正在执行该任务的工作线程会执行其他正在等待的任务(work stealing)，最后通过join()方法得到子任务的处理结果；否则，直接执行任务，即在每一行中查找单词出现的次数，这里是通过一个LineTask来实现的。

注：由于ForkJoinTask类实现了Future接口，因此还提供了一个get()方法可以得到当前结果(当然，这也是一个同步方法)，不过一般不太使用，因此它可能抛出检查性异常，而在compute()方法中不允许抛出这些异常。

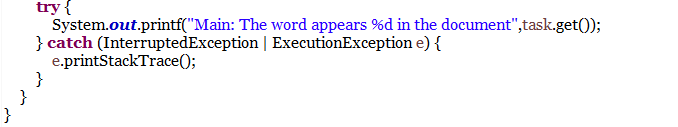
接下来创建一个继承RecursiveTask类的LineTask类，用来查找单词在一行中出现的次数。





最后，创建主程序Main类。





在这个示例中，你已经实现了两种不同的任务。

DocumentTask类：这个类的任务将处理由start和end属性决定的文档中的行组。如果这个行组的大小小于10，它为每行创建LineTask对象，并且当它们完成它们的执行时，它合计这些任务的结果，并返回这个合计值。如果这个任务要处理的行组大小不小于10，它将这个组分成两个并创建两个DocumentTask对象来处理这些新组。当这些任务完成它们的执行时，这个任务合计它们的结果，并返回这个合计值。

LineTask类：这个类的任务将处理文档中的一行的单词组。如果这个单词组小于10，这个任务直接在这个单词组中查找单词，并且返回这个单词出现的次数。否则，它将这个单词组分成两个并创建两个LineTask对象来处理。当这些任务完成它们的执行，这个任务合计这些任务的结果并返回这个合计值。

在Main类中，你已经使用默认构造器一个ForkJoinPool对象，并且你在它里面执行一个DocumentTask类，这个类将处理一个拥有100行，每行有100\_000个单词的文档。这个任务将使用其他的DocumentTask对象和LineTask对象来分解这个问题，当所有任务完成它们的执行，你可以使用启动任务来获取单词在整个文档中出现的总次数。由于任务返回一个结果，所以它们继承RecursiveTask类。

当你执行这个程序，你可以比较在控制台中的第一行和最后一行。第一行是文档生成时计算的单词出现的次数，最后一行是由Fork/Join任务计算的。

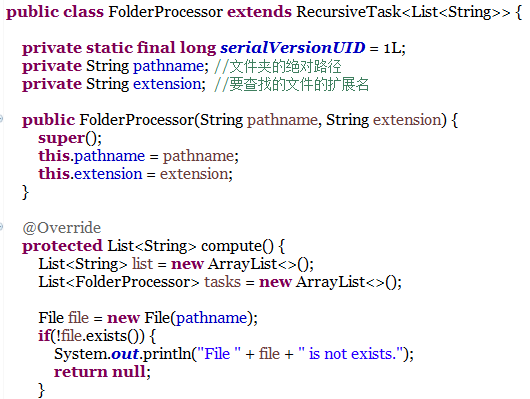
# Fork-Join框架实例三：异步执行任务

当你在ForkJoinPool中执行ForkJoinTask时，你可以使用同步或异步方式来实现。当你使用同步方式时，将任务提交给池后，会一直等到该任务完成后才会继续执行。当你使用异步方式时，将任务提交给池后会立即返回，所以这个任务可以继续执行。

你应该意识到这两个方法有很大的区别，当你使用同步方法，调用这些方法（比如：invokeAll()方法）的任务将被阻塞，直到提交给池的任务被完成后才会继续执行，这允许ForkJoinPool类使用work-stealing算法，分配一个新的任务给正在执行睡眠任务的工作线程。反之，当你使用异步方法（比如：fork()方法），这个任务将继续它的执行，所以ForkJoinPool类不能使用work-stealing算法来提高应用程序的性能，在这种情况下，只有当你调用join()或get()方法来等待任务完成时，ForkJoinPool才能使用work-stealing算法。

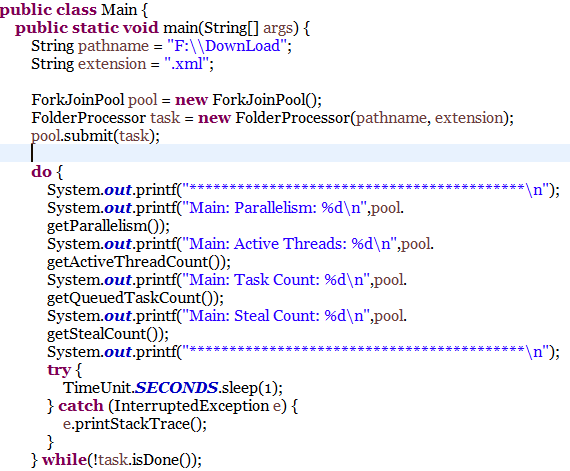
在这个指南中，你将学习如何使用ForkJoinPool和ForkJoinTask类提供的异步方法来管理任务。你将实现一个程序，在一个文件夹及其子文件夹内查找确定扩展名的文件。你将实现ForkJoinTask类来处理文件夹的内容，对于文件夹里的每个子文件夹，它将以异步的方式提交一个新的任务给ForkJoinPool类；对于文件夹里的每个文件，任务将检查文件的扩展名，如果它被处理，就把它添加到结果列表。

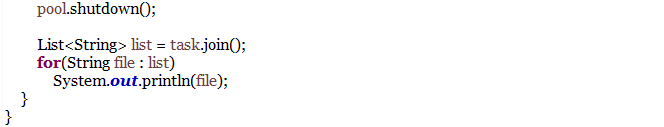
首先创建继承自RecursiveTask类的FolderProcessor类。





再创建主程序Main类。





这个例子的关键是FolderProcessor类，每个任务处理文件夹的内容。如果任务找到一个文件夹，它创建另一个Task对象来处理这个文件夹，并使用fork()方法把它（Task对象）提交给池，fork()方法会立即返回，所以这个任务可以继续处理文件夹的内容。对于每个文件，任务将它的扩展与所想要查找的（扩展）进行比较，如果它们相等，将文件名添加到结果数列。

一旦这个任务处理完指定文件夹的所有内容，它将使用join()方法等待已提交到池的所有任务的结束，最终，该任务将其所有子任务的结果和它自己的结果合并，并返回作为compute()方法的返回值返回。

# Fork-Join框架的异常处理

ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码：

if(task.isCompletedAbnormally())

{

System.out.println(task.getException());

}

getException方法返回Throwable对象，如果任务被取消了则返回CancellationException。如果任务没有完成或者没有抛出异常则返回null。