从javacreed / java-fork-join-example
分叉的
advanced-javas / java-fork-join-example

Java Creed - Java Fork加入示例 http://www.javacreed.com/java-fork-jo ...

编辑

管理主题

10次 提交 | 2个 分支 | 0 发布 | 1个 贡献者 | Apache的2.0

科： 硕士 | 新拉请求 | 创建新文件 | 上传文件 | 查找文件 | 克隆或下载

这个分支甚至与javacreed：master。 拉请求 比较

javacreed 修复了自述文件格式的一些问题 最新提交 d7851a0 on 27 Nov 2018

IMG	更新了自述文件并添加了图像	4 months ago
src / main	初始导入	4 years ago
的.gitignore	更新了自述文件并添加了图像	4 months ago
.travis.yml	配置Travis CI	2 years ago
执照	包括开源许可证	2 years ago
README.md	修复了自述文件格式的一些问题	4 months ago
的pom.xml	更新了自述文件并添加了图像	4 months ago

README.md

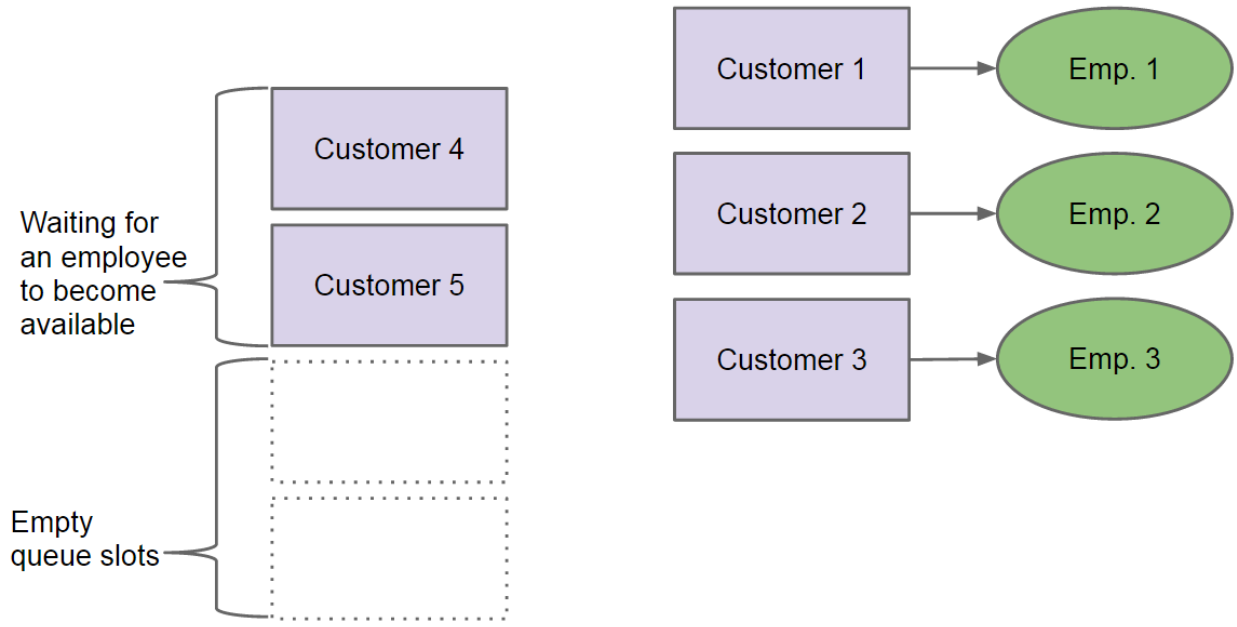
Java 7引入了一种名为**Fork / Join Framework**（Tutorial）的新型 `ExecutorService`（Java Doc），它在处理递归算法方面表现优异。与其他实现不同，Fork / Join Framework使用工作窃取算法（Paper），它可以最大化线程利用率，并提供一种更简单的方法来处理产生其他任务的任务（称为*子任务*）。 `ExecutorService`

下面列出的所有代码均可在以下网址获得：https：//github.com/javacreed/java-fork-join-example。大多数示例将不包含整个代码，并且可能省略与所讨论的示例无关的片段。读者可以从上面的链接下载或查看所有代码。

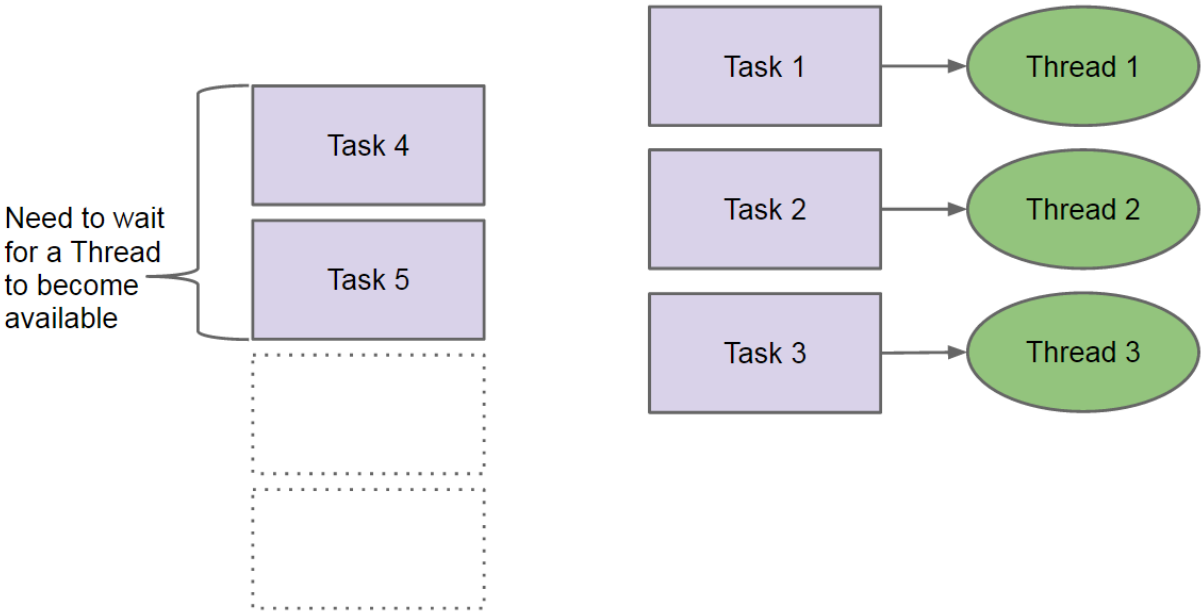
本文简要介绍了所谓的传统执行程序服务（以便称之为传统执行程序服务）以及它们如何工作。然后介绍了Fork / Join框架，并描述了它与传统执行程序服务的区别。本文的第三部分显示了Fork / Join Framework的一个实际示例，并演示了其大部分主要组件。

执行人服务

银行或邮局有几个柜台，客户在分行服务。当计数器与当前客户完成时，队列中的下一个客户将取代其位置，柜台后面的人员（也称为*员工*）开始为新客户提供服务。员工只在给定时间点为一个客户服务，队列中的客户需要等待轮到他们。此外，员工非常有耐心，永远不会要求他们的客户离开或离开，即使他们正在等待其他事情发生。下图显示了客户等待的简单视图以及在队列头部为客户提供服务的员工。



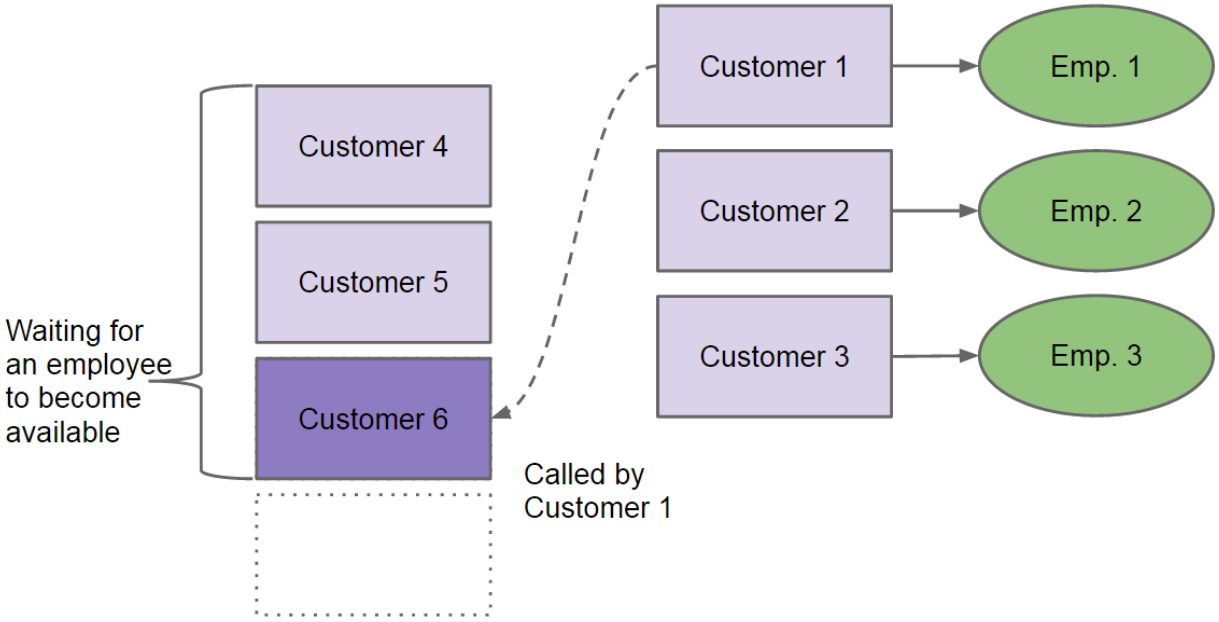
类似的事情发生在多线程程序中，其中 `Thread` s（Java Doc）代表员工，而要执行的任务是客户。以下图像与上面的图像相同，只更新了标签以使用编程术语。



这应该有助于您将这两个方面联系起来，并更好地可视化正在讨论的场景。

大多数线程池（Tutorial）和执行器服务都以这种方式工作。A `Thread` 被分配了一个*任务*，只有在手头的*任务完成*后才会移动到下一个*任务*。任务可能需要很长时间才能完成，并可能阻止等待其他事情发生。这在许多情况下效果很好，但是在需要递归解决的问题上会出现严重问题。

让我们使用我们之前用过的客户在队列中等待的相同类比。假设由*员工1*提供服务的*客户1*需要来自尚未排队的*客户6的*一些信息。他或她（*客户1*）打电话给他们的朋友（*客户6*）并等待他或她（*客户6*）来到银行。与此同时，*客户1*留在占用*员工1*的柜台。如前所述，员工非常有耐心，绝不会将客户送回队列或要求他们退出，直到他或她的所有依赖关系得到解决。 *客户6* 到达并排队如下所示。



随着*1号客户*仍然占据雇员，并为这一论点的缘故其他客户，*客户2*和*客户3*，太不相同（也就是等待一些东西，排队），然后我们有一个僵局。所有员工都被等待发生事情的客户所占据。因此，员工永远不会自由地为其他客户服务。

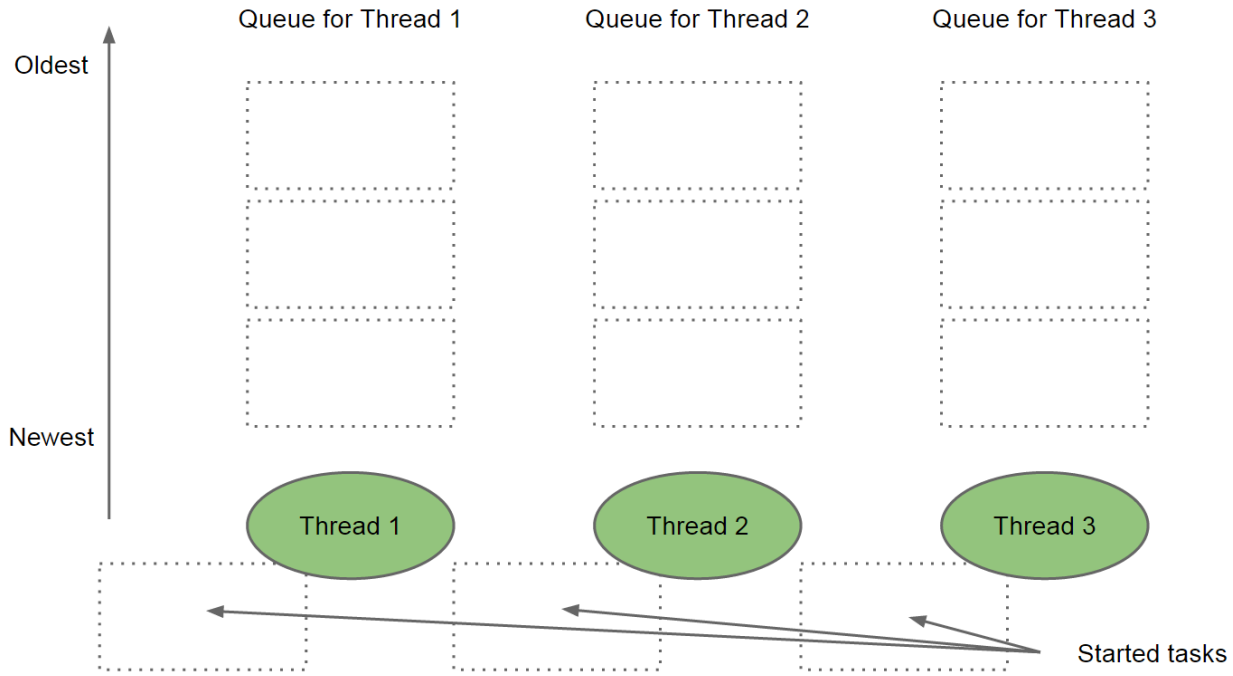
在这个例子中，我们在处理任务时看到了传统执行程序服务的弱点，而这些任务又依赖于由它们创建的其他任务（称为子任务）。这在诸如河内塔（维也纳）之类的递归算法或探索数据结构之类的树（计算目录的总大小）中非常常见。Fork / Join框架旨在解决此类问题，我们将在下一节中看到。在本文的后面，我们还将看到本节中讨论的问题的示例。

分叉/加入框架

传统执行程序服务实现在处理任务时的主要弱点是，线程无法将任务放回队列或侧面，然后服务/执行新任务，从而依赖于其他子任务。Fork / Join Framework通过在任务和执行它们的线程之间引入另一个层来解决这个限制，这允许线程将阻塞的任务放在一边并在执行所有依赖项时处理它们。换句话说，如果*任务1*依赖于*任务6*，*任务1*创建了哪个任务（*任务6*），那么*任务1*放在一边，只执行一次*任务6*被执行。这将从*任务1*释放线程，并允许它执行其他任务，这是传统执行程序服务实现无法实现的。

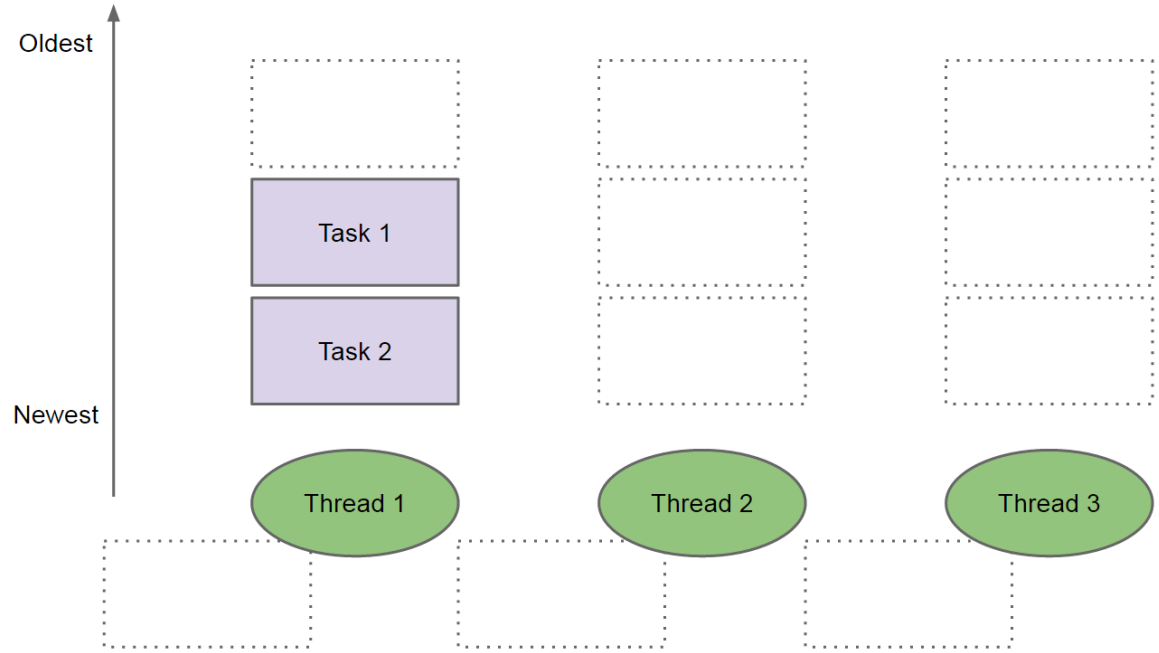
这是通过使用框架提供的*fork*和*join*操作实现的（因此名称为Fork / Join）。 *任务1*分叉*任务6*，然后加入它以等待结果。fork操作将*任务6*放入队列，而join操作允许*线程1*将*任务1*放在一边，直到*任务6*完成。这就是fork / join的工作方式，fork将新内容推送到队列，而join将导致当前任务处于双向，直到它可以继续，从而阻止没有线程。

Fork / Join Framework使用一种称为 `ForkJoinPool` （Java Doc）的特殊线程池，它将其与其他线程池区分开来。 `ForkJoinPool` 实现工作窃取算法并可以执行 `ForkJoinTask` （Java Doc）对象。所述 `ForkJoinPool` 保持多个线程，其数量通常是基于可用的CPU的数量。每个线程都有一种特殊的队列，即 `Deque` （Java Doc），其中放置了所有任务。这是一个非常重要的要点。线程不共享公共队列，但每个线程都有自己的队列，如下所示。

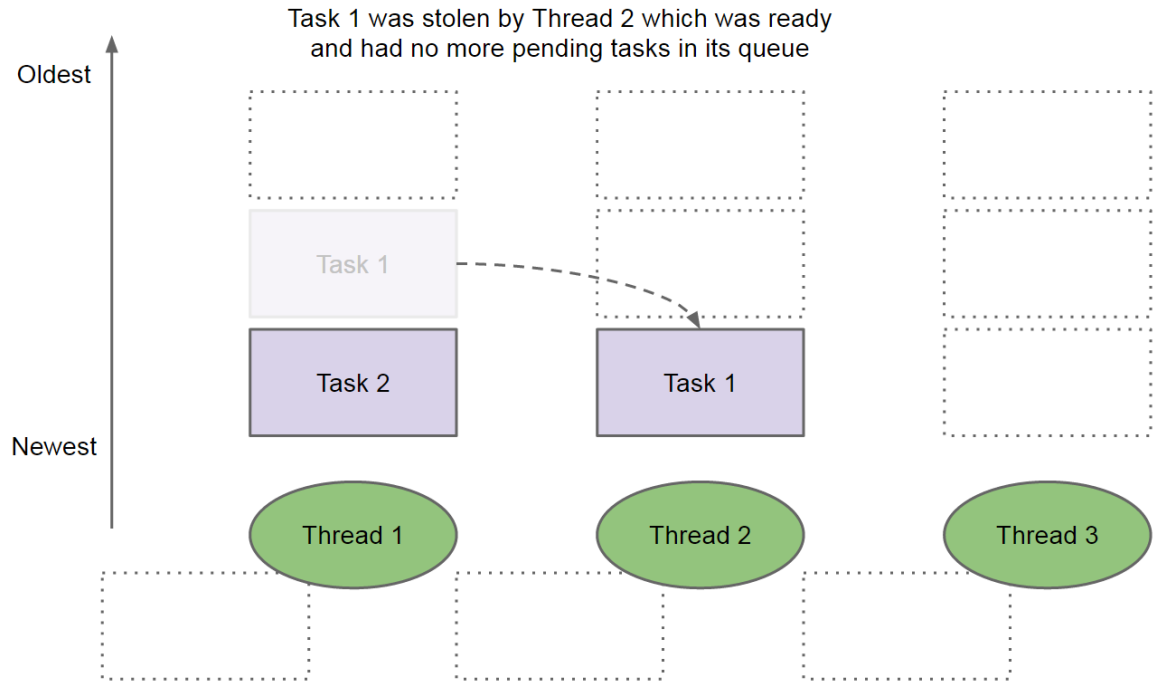


上图说明了每个线程具有的另一个队列（图像的下半部分）。这个队列，所以要调用它，允许线程放下被阻止等待其他事情发生的任务。换句话说，如果当前任务无法继续（因为它在子任务上执行*连接*），那么它将被置于此队列中，直到其所有依赖项都准备就绪。

新任务被添加到线程的队列中（使用*fork*操作），每个线程总是处理添加到其队列的最后一个任务。这非常重要。如果线程队列有两个任务，则首先处理添加到队列的最后一个任务。这被称为后进先出，LIFO（Wiki）。



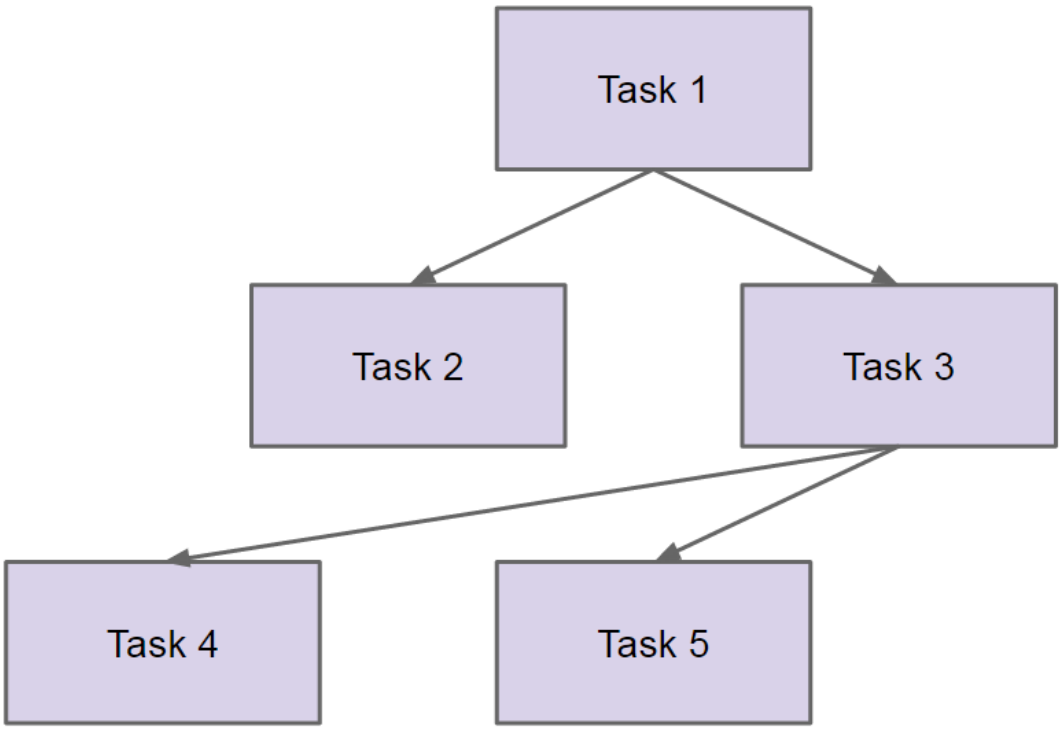
在上图中，*线程1*在其队列中有两个任务，其中*任务1*在*任务2*之前被添加到队列中。因此，*任务2*将首先由*线程1*执行，然后执行*任务1*。任何空闲线程都可以从其他线程队列中获取任务（如果可用），即工作窃取。线程将始终从其他线程的队列中窃取最旧的任务，如下图所示。



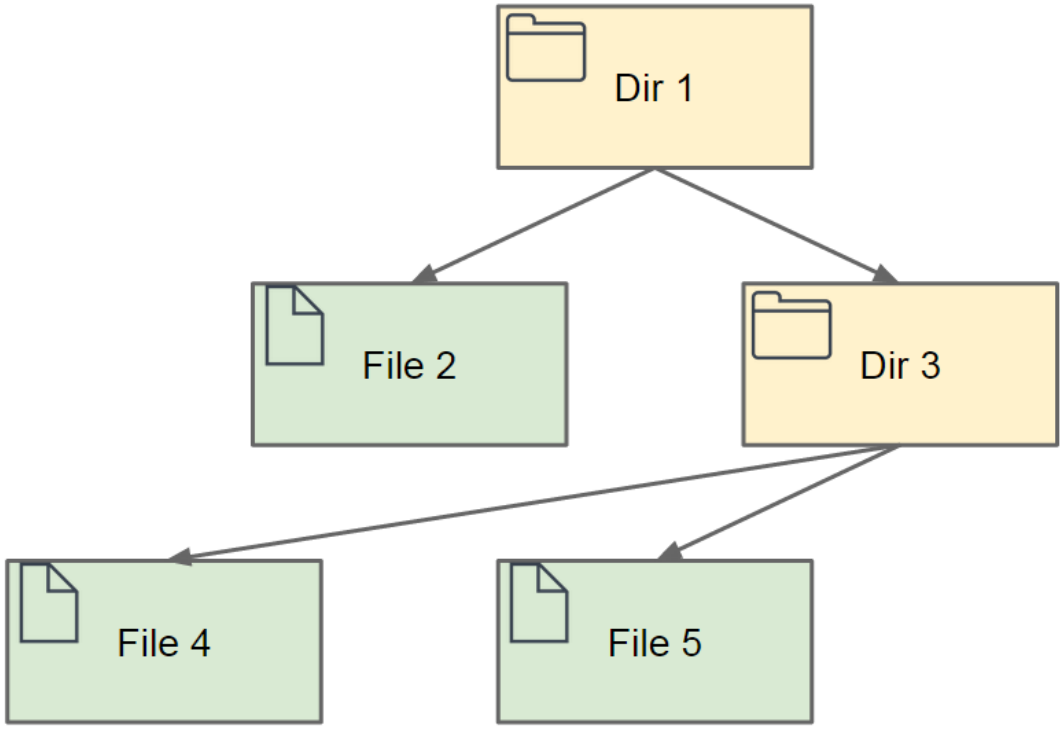
如上图所示，*线程2*从*线程1*中窃取了最早的任务，*任务1*。根据经验，线程将始终尝试从其相邻线程窃取，以最小化在工作窃取期间可能产生的争用。

任务执行和被盗的顺序非常重要。理想情况下，工作窃取不会发生很多，因为这有成本。当任务从一个线程移动到另一个线程时，与此任务相关的上下文需要从一个线程的堆栈移动到另一个线程。线程可能是（并且Fork / Join框架在所有CPU上扩展工作）在另一个CPU上。将线程上下文从一个CPU移动到另一个CPU甚至可能更慢。因此，Fork / Join Framework最小化了这一点，如下所述。

递归算法从一个大问题开始，并采用分而治之的技术将问题分解为更小的部分，直到它们足够小以便直接求解。添加到队列的第一个任务是最大的任务。第一个任务将问题分解为一组较小的任务，这些任务将添加到队列中，如下所示。

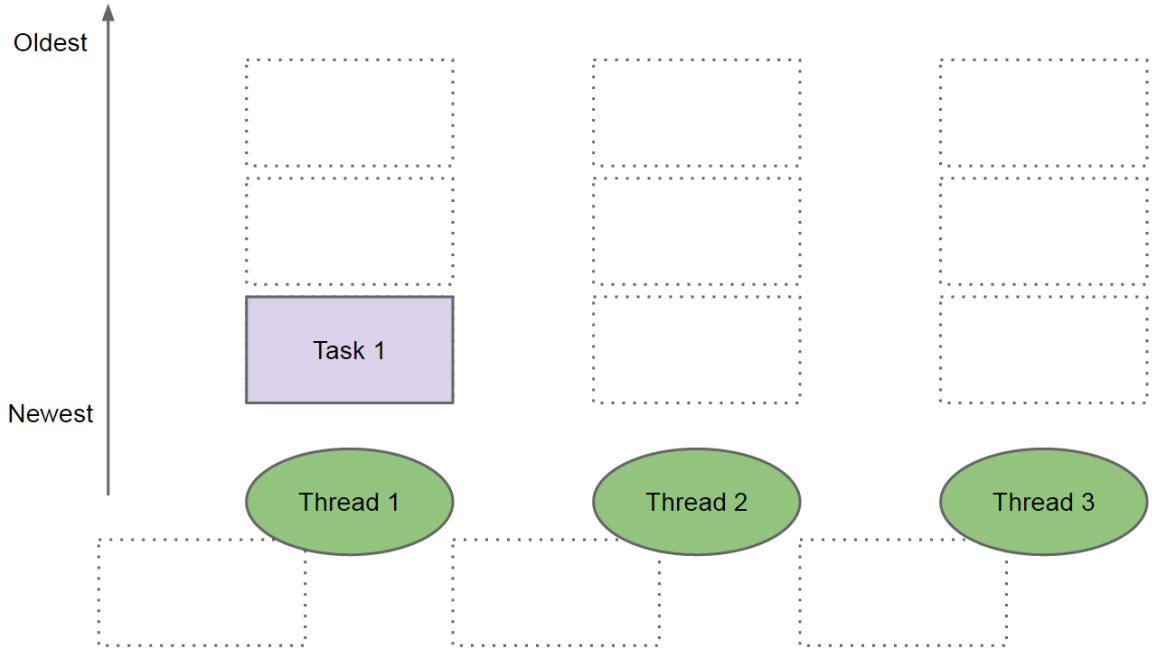


*任务1*代表我们的问题，它分为两个任务， *任务2*足够小，可以按原样解决，但*任务3*需要进一步划分。任务*任务4*和*任务5*足够小，这些不需要进一步拆分。这表示典型的递归算法，可以将其拆分为较小的部分，然后在准备好时聚合结果。这种算法的一个实际例子是计算目录的大小。我们知道目录的大小等于其文件的大小。

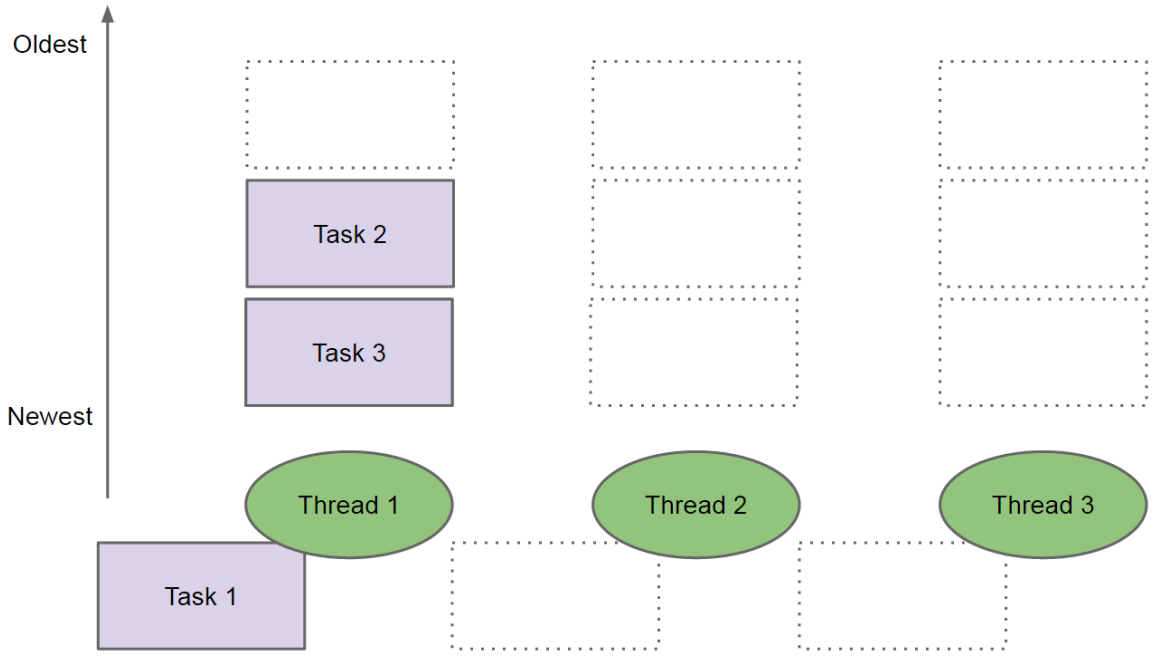


因此，*目录1*的大小等于*文件2*的大小加上*目录3*的大小。由于*目录3*是目录，因此其大小等于其内容的大小。换句话说，*目录3*的大小等于*文件4*的大小加上*文件5*的大小。

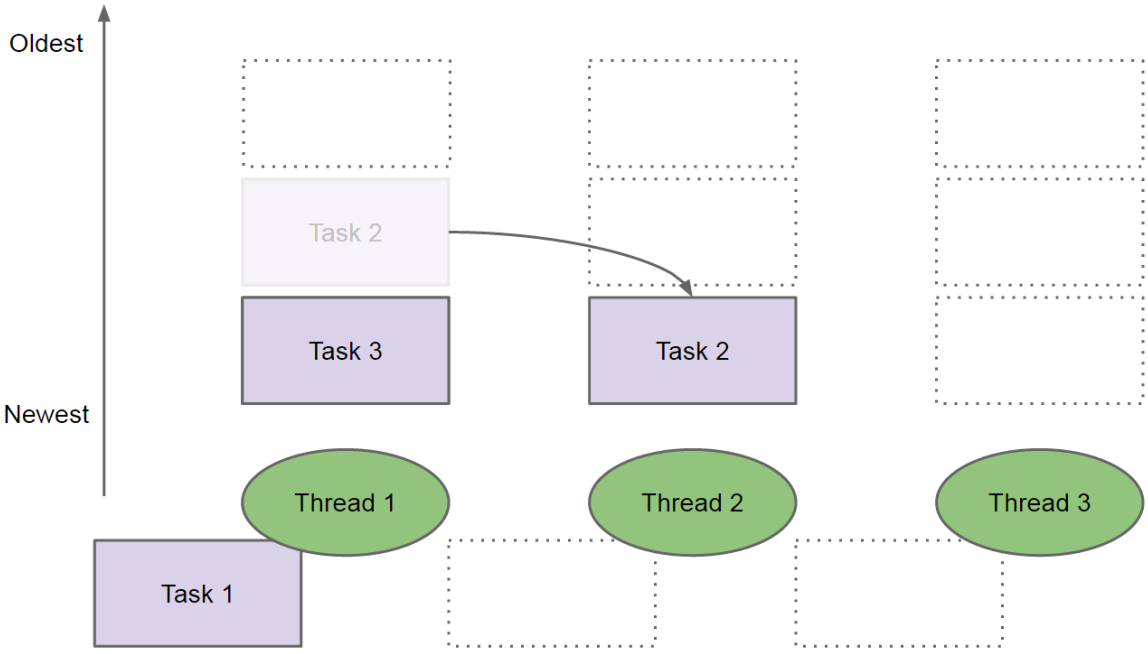
让我们看看这是如何执行的。我们从一个任务开始，即计算目录的大小，如下图所示。



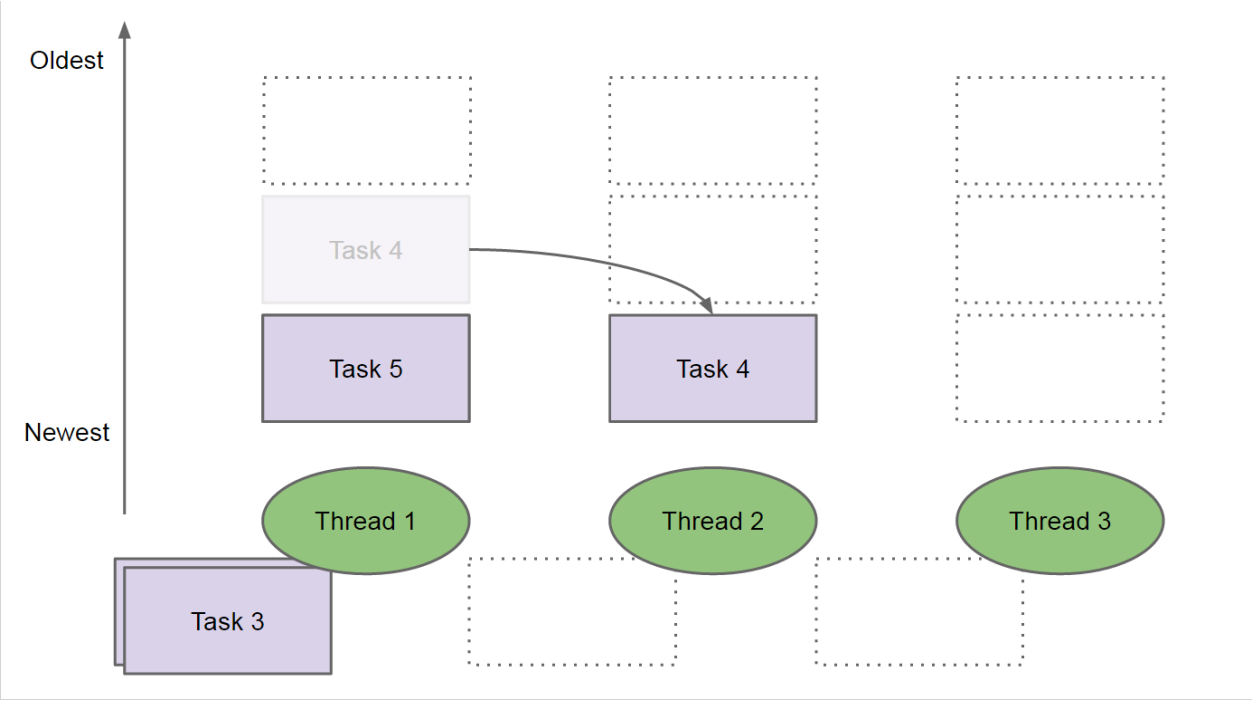
*线程1*将执行*任务1*，任务分叉两个其他子任务。这些任务将添加到*线程1*的队列中，如下图所示。



*任务1*正在等待子任务，*任务2*和*任务3*完成，因此被推到一边释放*线程1*。要使用更好的术语，*任务1*加入子*任务任务2*和*任务3*。 *线程1*开始执行*任务3*（添加到其队列中的最后一个任务），而*线程2*窃取*任务2*。

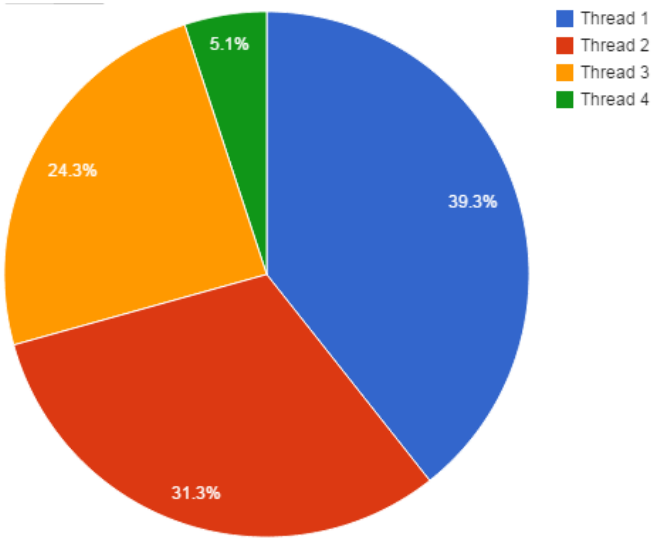


请注意，*线程1*已经开始处理其第二个任务，而*线程3*仍处于空闲状态。正如我们稍后将看到的，线程将不会执行相同数量的工作，并且第一个线程将始终产生比最后一个线程更多的工作。 *任务3*分叉两个子任务，这些子任务被添加到正在执行它的线程的队列中。因此，另外两个任务被添加到*线程1*的队列中。 从*任务2*准备好的*线程2*再次窃取另一个任务，如下所示。



在上面的例子中，我们看到*Thread 3*从未执行任务。这是因为我们只有很少的子任务。一旦*任务4*和*任务5*是准备好了，他们的结果被用来计算*任务3*，然后*任务1*。

如前所述，工作并不是均匀分布在线程中。下图显示了在计算相当大的目录的大小时如何在线程之间分配工作。



在上面的例子中，使用了四个线程。正如预期的那样，线程1执行了近40%的工作，而线程4（最后一个线程）执行的工作略多于5%。这是另一个需要理解的重要原则。Fork / Join Framework不会在线程之间均匀分配工作，并会尽量减少使用的线程数。第二个线程只会从第一个线程开始工作，这不是cooping。如前所述，在线程之间移动任务具有框架试图最小化的成本。

本节详细描述了Fork / Join Framework如何工作以及线程如何从其他线程的队列中窃取工作。在下一节中，我们将看到Fork / Join Framework的几个实际示例，并将分析获得的结果。

计算目录总大小

为了演示Fork / Join Framework的使用，我们将计算目录的大小，这个问题可以递归地解决。文件大小可以通过方法 `length()`（Java Doc）确定。目录的大小等于其所有文件的大小。

我们将使用几种方法来计算目录的大小，其中一些将使用Fork / Join Framework，我们将在每种情况下分析获得的结果。

使用单线程（非并发）

第一个例子不会使用线程，并且简单地定义了使用的算法。

```
package  com.javacreed.examples.concurrency.part1 ;

import  java.io.File ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

公共 课 DirSize {

  private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（DirSize 。类）;

  public  static  long  sizeOf（final  file  file）{
     DirSize 。记录器。debug（“计算大小：{} ”，文件）;

    长尺寸=  0 ;

    / *忽略不属于文件和文件显示目录* /
    如果（文件。 ISFILE（））{
      size =文件。长度（）;
    } else {
       final  File [] children = file 。listFiles（）;
      if（children ! =  null）{
         for（final  File child ：  children）{
          size + =  DirSize 。整型尺寸（孩子）;
        }
      }
    }

    回报大小;
  }
}
```

该类 `DirSize` 有一个名为的方法 `sizeOf()` ，该方法将 `File` （ Java Doc ）实例作为其参数。如果此实例是文件，则该方法返回文件的长度，否则，如果这是一个目录，则此方法为目录 `sizeOf()` 中的每个文件调用方法并返回总大小。

以下示例显示如何使用 `FilePath.TEST_DIR` 常量定义的文件路径运行此示例。

```
package  com.javacreed.examples.concurrency.part1 ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

import  com.javacreed.examples.concurrency.utils.FilePath ;

公共 类 Example1 {

  private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（例1 。类）;

  public  static  void  main（final  String [] args）{
     final  long start =  System 。nanoTime（）;
    最终 长尺寸=  DirSize 。整型尺寸（文件路径。 TEST_DIR）;
    最后 长期采取=  系统。nanoTime（）-开始;

    例1 。记录器。调试（“的'{}'尺寸：{}字节（在{}纳米）”，文件路径。 TEST_DIR，大小，采取）;
  }
}
```

上面的示例将计算目录的大小，并在打印总大小之前打印所有访问过的文件。以下片段仅显示最后一行，即测试文件夹（ ）下的 *下载*目录的大小以及 `C:\Test` 计算大小所用的时间。

```
...
16:55:38.045 [main] INFO Example1.java:38 - Size of 'C:\Test\': 113463195117 bytes (in 4503253988 nano)
```

要为每个访问的文件禁用日志，只需将日志级别更改为 `INFO` （ 在 `log4j.properties` ）中，日志将仅显示最终结果。

```
log4j.rootCategory=warn, R
log4j.logger.com.javacreed=info, stdout
```

请注意，日志记录只会使速度变慢。实际上，如果运行没有日志的示例（或将日志设置为 `INFO` ），则计算目录的大小会更快。

为了获得更可靠的结果，我们将多次运行相同的测试并返回如下所示的平均时间。

```
package  com.javacreed.examples.concurrency.part1 ;

import  java.util.concurrent.TimeUnit ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

import  com.javacreed.examples.concurrency.utils.FilePath ;
import  com.javacreed.examples.concurrency.utils.Results ;

公共 类 Example2 {

  private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（例1 。类）；

  public  static  void  main（final  String [] args）{
     final  Results results =  new  Results（）；
    for（int i =  0 ; i <  5 ; i ++）{
      结果。开始时间（）；
      最终 长尺寸=  DirSize 。整型尺寸（文件路径。 TEST_DIR）；
      最后 长期采取=结果。时间结束（）；
      例2 。记录器。信息（“大小'{}'：{}字节（在{}纳米）”，文件路径。 TEST_DIR，大小，采取）；
    }

    最终的 长期被采纳=结果。getAverageTime（）；
    例2 。记录器。信息（“平均：{}纳米（{}秒）”，takenInNano，TIMEUNIT 。纳秒。 toSeconds（takenInNano））；
  }
}
```

相同的测试执行五次，平均结果最后打印，如下所示。

```
16:58:00.496 [main] INFO Example2.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 4266090211 nano)
16:58:04.728 [main] INFO Example2.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 4228931534 nano)
16:58:08.947 [main] INFO Example2.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 4224277634 nano)
16:58:13.205 [main] INFO Example2.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 4253856753 nano)
16:58:17.439 [main] INFO Example2.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 4235732903 nano)
16:58:17.439 [main] INFO Example2.java:46 - Average: 4241777807 nano (4 seconds)
```

RecursiveTask

Fork / Join Framework提供两种类型的任务， `RecursiveTask` （Java Doc）和 `RecursiveAction` （Java Doc）。在本节中我们只讨论 `RecursiveTask` 。 `RecursiveAction` 稍后将对此进行讨论。

A `RecursiveTask` 是一项任务，在执行时返回一个值。因此，这样的任务返回计算结果。在我们的示例中，该任务返回它所代表的文件或目录的大小。该类 `DirSize` 被修改为使用 `RecursiveTask` 如下所示。

```
package  com.javacreed.examples.concurrency.part2 ;

import  java.io.File ;
import  java.util.ArrayList ;
import  java.util.List ;
import  java.util.Objects ;
import  java.util.concurrent.ForkJoinPool ;
import  java.util.concurrent.RecursiveTask ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

公共 课 DirSize {

  private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（DirSize 。类）；

  private  static  class  SizeOfFileTask  extends  RecursiveTask < Long > {

    private  static  final  long serialVersionUID =  - 1965224082913439 51L ;

    私人 最终 文件的文件;

    public  SizeOfFileTask（最终 文件 文件）{
       this 。file =  对象。requireNonNull（文件）；
    }
```

```
    @Override
    protected  Long  compute（）{
       DirSize 。记录器。debug（“计算大小：{} ”，文件）；

      如果（文件。 ISFILE（））{
         回报文件。长度（）；
      }

      final  List < SizeOfFileTask > tasks =  new  ArrayList <>（）；
      final  File [] children = file 。listFiles（）；
      if（children ! =  null）{
         for（final  File child ：  children）{
           final  SizeOfFileTask task =  new  SizeOfFileTask（child）；
          任务。叉子（）；
          任务。添加（任务）；
        }
      }

      长尺寸=  0 ；
      for（final  SizeOfFileTask task ：  tasks）{
        大小+ =任务。加入（）；
      }

      回报大小；
    }
  }

  public  static  long  sizeOf（final  file  file）{
     final  ForkJoinPool pool =  new  ForkJoinPool（）；
    尝试 {
       返回池。invoke（new  SizeOfFileTask（file））；
    } finally {
      游泳池。关掉（）；
    }
  }

  私人 DirSize（）{}

}
```

让我们将这个类分成更小的部分并分别描述它们。

1. 该类有一个私有构造函数，因为它不是要初始化的。因此没有必要初始化它（创建这种类型的对象）并且为了防止某人初始化它，我们制作了构造函数 `private` 。所有方法都是静态的，可以直接对类调用。

   ```
   私人 DirSize（）{}
   ```

2. 该方法 `sizeOf()` 不计算文件或目录的大小。相反，它创建了一个实例 `ForkJoinPool` 并启动计算过程。它等待计算目录大小，最后在退出之前关闭池。

   ```
   public  static  long sizeOf（final  file file）{
      final  ForkJoinPool pool =  new  ForkJoinPool（）；
     尝试 {
        返回池。invoke（new  SizeOfFileTask（file））；
     } finally {
       游泳池。关掉（）；
     }
   }
   ```

   `ForkJoinPool` 默认情况下，由守护程序线程创建的线程。有些文章建议不要关闭此池，因为这些线程不会阻止VM关闭。话虽如此，我建议在不再需要时关闭并妥善处理任何物体。即使不再需要这些守护程序线程，这些守护程序线程也可能长时间处于空闲状态。

3. 该方法 `sizeOf()` 创建一个实例 `SizeOfFileTask` ，该类扩展 `RecursiveTask<Long>` 。因此，invoke方法将返回此任务返回的对象/值。

   ```
   返回池。invoke（new  SizeOfFileTask（file））；
   ```

请注意，上述代码将阻塞，直到计算目录的大小。换句话说，上面的代码将等待任务（和所有子任务）在继续之前完成工作。

4. 该类 `SizeOfFileTask` 是类中的内部 `DirSize` 类。

```
private  static  class  SizeOfFileTask  extends  RecursiveTask < Long > {

  private  static  final  long serialVersionUID =  - 1965224082913439 51L ;

  私人 最终 文件的文件;

  public  SizeOfFileTask（最终 文件 文件）{
     this 。file =  对象。requireNonNull（文件）;
  }

  @Override
  protected  Long  compute（）{
     / *为简洁而删除* /
  }
}
```

它接受文件（可以是一个目录），其大小将被计算为其唯一构造函数的参数，该文件不能 `null` 。该 `compute()` 方法负责计算此任务的工作。在这种情况下，文件或目录的大小，接下来讨论哪种方法。

5. 该 `compute()` 方法确定传递给其构造函数的文件是文件还是目录，并相应地起作用。

```
@Override
protected  Long compute（）{
   DirSize 。记录器。debug（“计算大小：{} ”，文件）;

  如果（文件。 ISFILE（））{
     回报文件。长度（）;
  }

  final  List < SizeOfFileTask > tasks =  new  ArrayList <> （）;
  final  File [] children = file 。listFiles（）;
  if（children ! =  null）{
     for（final  File child ：  children）{
       final  SizeOfFileTask task =  new  SizeOfFileTask（child）;
      任务。叉子（）;
      任务。添加（任务）;
    }
  }

  长尺寸=  0 ;
  for（final  SizeOfFileTask task ：  tasks）{
    大小+ =任务。加入（）;
  }

  回报大小;
}
```

如果文件是文件，则该方法只返回其大小，如下所示。

```
如果（文件。 ISFILE（））{
   回报文件。长度（）;
}
```

否则，如果文件是目录，则列出其所有子文件，并 `SizeOfFileTask` 为每个子文件创建新实例。

```
final  List < SizeOfFileTask > tasks =  new  ArrayList <> （）;
final  File [] children = file 。listFiles（）;
if（children ! =  null）{
   for（final  File child ：  children）{
     final  SizeOfFileTask task =  new  SizeOfFileTask（child）;
    任务。叉子（）;
    任务。添加（任务）;
  }
}
```

对于创建的每个实例，调用 `SizeOfFileTask` 该 `fork()` 方法。该 `fork()` 方法导致将新实例 `SizeOfFileTask` 添加到此线程的队列中。所有创建的实例 `SizeOfFileTask` 都保存在名为的列表中 `tasks` 。最后，当所有任务都分叉时，我们需要等待他们完成他们的值的总结。

```
长尺寸=  0 ;
for（final  SizeOfFileTask task :  tasks）{
  大小+ =任务。加入（）;
}

回报大小;
```

这是通过该 `join()` 方法完成的。这 `join()` 将强制此任务停止，如果需要则退出，并等待子任务完成。所有子任务返回的值都将添加到变量的值中，该值 `size` 将作为此目录的大小返回。

与不使用多线程的更简单版本相比，Fork / Join框架更复杂。这是一个公平的观点，但更简单的版本慢了4倍。Fork / Join示例平均花费一秒钟来计算大小，而非线程版本平均花费4秒钟，如下所示。

```
16:59:19.557 [main] INFO Example3.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 2218013380 nano)
16:59:21.506 [main] INFO Example3.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 1939781438 nano)
16:59:23.505 [main] INFO Example3.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 2004837684 nano)
16:59:25.363 [main] INFO Example3.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 1856820890 nano)
16:59:27.149 [main] INFO Example3.java:42 - Size of 'C:\Test\': 113463195117 bytes (in 1782364124 nano)
16:59:27.149 [main] INFO Example3.java:46 - Average: 1960363503 nano (1 seconds)
```

在本节中，我们了解了多线程如何帮助我们提高程序的性能。在下一节中，我们将看到多线程的不当使用会使事情变得更糟。

ExecutorService的

在前面的例子中，我们看到了并发性如何提高了算法的性能。如果误用，多线程可能会产生不良结果，我们将在本节中看到。我们将尝试使用传统的执行程序服务来解决此问题。

请注意，本节中显示的代码已损坏，不起作用。它会永远挂起，仅用于演示目的。

这个类 `DirSize` 进行了修改一起工作 `ExecutorService` 和 `Callable` （Java文档）。

```
package  com.javacreed.examples.concurrency.part3 ;

import  java.io.File ;
import  java.util.ArrayList ;
import  java.util.List ;
import  java.util.Objects ;
import  java.util.concurrent.Callable ;
import  java.util.concurrent.ExecutorService ;
import  java.util.concurrent.Executors ;
import  java.util.concurrent.Future ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

/ **
 *此示例已损坏且遇到死锁，仅用于文档目的。
*
* @author Albert Attard
* / public class DirSize {


  私有 静态 类 SizeOfFileCallable  实现 Callable < Long > {

    私人 最终 文件的文件;
    private  final  ExecutorService executor;

    public  SizeOfFileCallable（最终 文件 文件，最终 ExecutorService  执行程序）{
       this 。file =  对象。requireNonNull（文件）;
      这个。executor =  Objects 。requireNonNull（执行）;
    }

    @Override
    public  Long  call（）抛出 Exception {
       DirSize 。记录器。debug（“计算大小：{} ”，文件）;
      长尺寸=  0 ;
```

```
        如果（文件。 ISFILE（））{
          size =文件。长度（）；
        } 否则 {
           最终 列表< 未来< 龙 > >期货= 新 的ArrayList <> （）；
          对于（最后 文件子：文件。 listFiles（））{
            期货。添加（遗嘱执行人。提交（新 SizeOfFileCallable（儿童，执行人））；
          }

          for（最终 未来< 长 >未来：未来）{
            尺寸+ =未来。得到（）；
          }
        }

        回报大小；
      }
    }

    public  static  < T >  long  sizeOf（final  file  file）{
       final  int threads =  Runtime 。getRuntime（）。availableProcessors（）；
      DirSize 。记录器。debug（“使用{}线程创建执行程序”，线程）；
      最终 ExecutorService executor =  Executors 。的newFixedThreadPool（线程）；
      试试 {
         return executor 。提交（new  SizeOfFileCallable（file，executor））。得到（）；
      } catch（final  Exception e）{
         throw  new  RuntimeException（“无法计算dir大小”，e）；
      } finally {
        执行人。关掉（）；
      }
    }

    private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（DirSize 。类）；

    私人 DirSize（）{}

  }
```

这个想法与以前非常相似。内部类 `SizeOfFileCallable` 将 `Callable<Long>` 其子任务的计算扩展并委托给 `ExecutorService` 传递给其构造函数的实例。在处理时，这不是必需的 `RecursiveTask` ，后者会自动将其子类添加到线程的队列中以便执行。

我们不会更详细地讨论这个问题，以使本文专注于Fork / Join框架。如前所述，一旦所有线程被占用，此方法就会阻塞，如下所示。

```
17:22:39.216 [main] DEBUG DirSize.java:78 - Creating executor with 4 threads
17:22:39.222 [pool-1-thread-1] DEBUG DirSize.java:56 - Computing size of: C:\Test\
17:22:39.223 [pool-1-thread-2] DEBUG DirSize.java:56 - Computing size of: C:\Test\Dir 1
17:22:39.223 [pool-1-thread-4] DEBUG DirSize.java:56 - Computing size of: C:\Test\Dir 2
17:22:39.223 [pool-1-thread-3] DEBUG DirSize.java:56 - Computing size of: C:\Test\Dir 3
```

此示例在Core i5计算机上执行，该计算机具有四个可用处理器（如 `Runtime.getRuntime().availableProcessors()` Java Doc所示）。一旦所有四个线程都被占用，这种方法将永远阻止，就像我们在本文开头的bank分支示例中看到的那样。所有线程都被占用，因此不能用于解决其他任务。人们可以建议使用更多线程。虽然这似乎是一个解决方案，但Fork / Join Framework仅使用四个线程解决了同样的问题。此外，线程并不便宜，因为他或她选择了不合适的技术，因此不应该简单地产生数千个线程。

虽然多线程这个短语在编程社区中被过度使用，但多线程技术的选择很重要，因为有些选项在我们上面看到的某些场景中根本不起作用。

RecursiveAction

Fork / Join Framework支持两种类型的任务。第二类任务是 `RecursiveAction` 。这些类型的任务并不意味着返回任何东西。这些对于您想要执行操作的情况（例如删除文件而不返回任何内容）非常理想。通常，您无法删除空目录。首先，您需要先删除所有文件。在这种情况下， `RecursiveAction` 可以在每个操作删除文件的位置使用，或者首先删除所有目录内容，然后删除目录本身。

以下是我们在本文中的最后一个示例。它显示了修改后的版本 `DirSize` ，它使用 `SizeOfFileAction` 内部类来计算目录的大小。

```
package  com.javacreed.examples.concurrency.part4 ;

import  java.io.File ;
import  java.util.Objects ;
```

```
import  java.util.concurrent.ForkJoinPool ;
import  java.util.concurrent.ForkJoinTask ;
import  java.util.concurrent.RecursiveAction ;
import  java.util.concurrent.atomic.AtomicLong ;

import  org.slf4j.Logger ;
import  org.slf4j.LoggerFactory ;

公共 课 DirSize {

  private  static  class  SizeOfFileAction  extends  RecursiveAction {

    private  static  final  long serialVersionUID =  - 1965224082913439 51L ;

    私人 最终 文件的文件;
    私人 决赛 AtomicLong sizeAccumulator;

    public  SizeOfFileAction（最终 文件 文件，最终 AtomicLong  sizeAccumulator）{
       this 。file =  对象。requireNonNull（文件）;
      这个。sizeAccumulator =  对象。requireNonNull（sizeAccumulator）;
    }

    @Override
    protected  void  compute （）{
       DirSize 。记录器。debug（“计算大小：{} ”，文件）;

      如果（文件。 ISFILE （））{
        sizeAccumulator 。（文件addAndGet 。长度 （））;
      } else {
         final  File [] children = file 。listFiles （）;
        if（children ! =  null）{
           for（final  File child ：  children）{
             ForkJoinTask 。invokeAll（new  SizeOfFileAction（child，sizeAccumulator））;
          }
        }
      }
    }
  }

  public  static  long  sizeOf（final  file  file）{
     final  ForkJoinPool pool =  new  ForkJoinPool （）;
    尝试 {
       final  AtomicLong sizeAccumulator =  new  AtomicLong （）;
      游泳池。invoke（new  SizeOfFileAction（file，sizeAccumulator））;
      return sizeAccumulator 。得到 （）;
    } finally {
      游泳池。关掉 （）;
    }
  }

  private  static  final  Logger  LOGGER  =  LoggerFactory 。getLogger（DirSize 。类）;

  私人 DirSize （）{}

}
```

这个类与它的前辈非常相似。主要区别在于返回最终值（文件或目录的大小）的方式。请记住， `RecursiveAction` 不能返回值。相反，所有任务将共享一个共同的类型计数器， `AtomicLong` 这些将增加此公共计数器，而不是返回文件的大小。

让我们将这个类分成更小的部分，然后逐个完成每个部分。我们将跳过之前已经解释过的部分，以免重复。

1. 该方法 `sizeOf()` 使用 `ForkJoinPool` 如前所述。名为common的公共计数器 `sizeAccumulator` 也在此方法中初始化并传递给第一个任务。此实例将与所有子任务共享，并且所有子任务都将增加此值。

```
    public  static  long sizeOf（final  file file）{
       final  ForkJoinPool pool =  new  ForkJoinPool （）;
      尝试 {
         final  AtomicLong sizeAccumulator =  new  AtomicLong （）;
        游泳池。invoke（new  SizeOfFileAction（file，sizeAccumulator））;
        return sizeAccumulator 。得到 （）;
      } finally {
        游泳池。关掉 （）;
      }
    }
```

像以前一样，此方法将阻塞，直到所有子任务都准备好，之后返回总大小。

2. 内部类 `SizeOfFileAction` 扩展 `RecursiveAction` ，其构造函数有两个参数。

```
private static class SizeOfFileAction extends RecursiveAction {

  private static final long serialVersionUID = -1965224082913439 51L ;

  私人 最终 文件的文件;
  私人 决赛 AtomicLong sizeAccumulator;

  public SizeOfFileAction（最终 文件 文件，最终 AtomicLong sizeAccumulator）{
     this 。file = 对象。requireNonNull（文件）;
    这个。sizeAccumulator = 对象。requireNonNull（sizeAccumulator）;
  }

  @Override
  protected void compute（）{
     / *为简洁而删除* /
  }
}
```

第一个参数是将计算大小的文件（或目录）。第二个参数是共享计数器。

3. 这里的计算方法稍微简单一些，因为它不必等待子任务。如果给定文件是文件，则它递增公共计数器（称为 `sizeAccumulator` ）。否则，如果此文件是目录，则会 `SizeOfFileAction` 为每个子文件分配新实例。

```
protected void compute（）{
   DirSize 。记录器。debug（“计算大小：{} ”，文件）;

  如果（文件。 ISFILE（））{
    sizeAccumulator 。（文件addAndGet 。长度（））;
  } else {
     final File [] children = file 。listFiles（）;
    if（children ! = null）{
       for（final File child ： children）{
         ForkJoinTask 。invokeAll（new SizeOfFileAction（child，sizeAccumulator））;
      }
    }
  }
}
```

在这种情况下，方法 `invokeAll()` （Java Doc）用于分叉任务。

这种方法大约需要11秒才能完成，使其成为所有三种中最慢的一种，如下所示。

```
19:04:39.925 [main] INFO Example5.java:40 - Size of 'C:\Test': 113463195117 bytes (in 11445506093 nano)
19:04:51.433 [main] INFO Example5.java:40 - Size of 'C:\Test': 113463195117 bytes (in 11504270600 nano)
19:05:02.876 [main] INFO Example5.java:40 - Size of 'C:\Test': 113463195117 bytes (in 11442215513 nano)
19:05:15.661 [main] INFO Example5.java:40 - Size of 'C:\Test': 113463195117 bytes (in 12784006599 nano)
19:05:27.089 [main] INFO Example5.java:40 - Size of 'C:\Test': 113463195117 bytes (in 11428115064 nano)
19:05:27.226 [main] INFO Example5.java:44 - Average: 11720822773 nano (11 seconds)
```

这对许多人来说可能是一个惊喜。当使用多个线程时，这怎么可能？这是一个普遍的误解。多线程并不能保证更好的性能。在这种情况下，我们有一个设计缺陷，理想情况下我们避免。名为的公共计数器 `sizeAccumulator` 在所有线程之间共享，从而导致线程之间的争用。这实际上违背了划分和征服技术的目的，因为瓶颈被创造出来。

结论

本文提供了Fork / Join框架的详细说明以及如何使用它。它提供了一个实际的例子并比较了几种方法。Fork / Join框架是递归算法的理想选择，但它不会均匀地在线程之间分配负载。除了join之外，任务和子任务不应该阻止其他任何事情，并且应该使用fork委托工作。避免任务中的任何阻塞IO操作并最小化可变共享状态，尤其是尽可能多地修改变量，因为这会对整体性能产生负面影响。