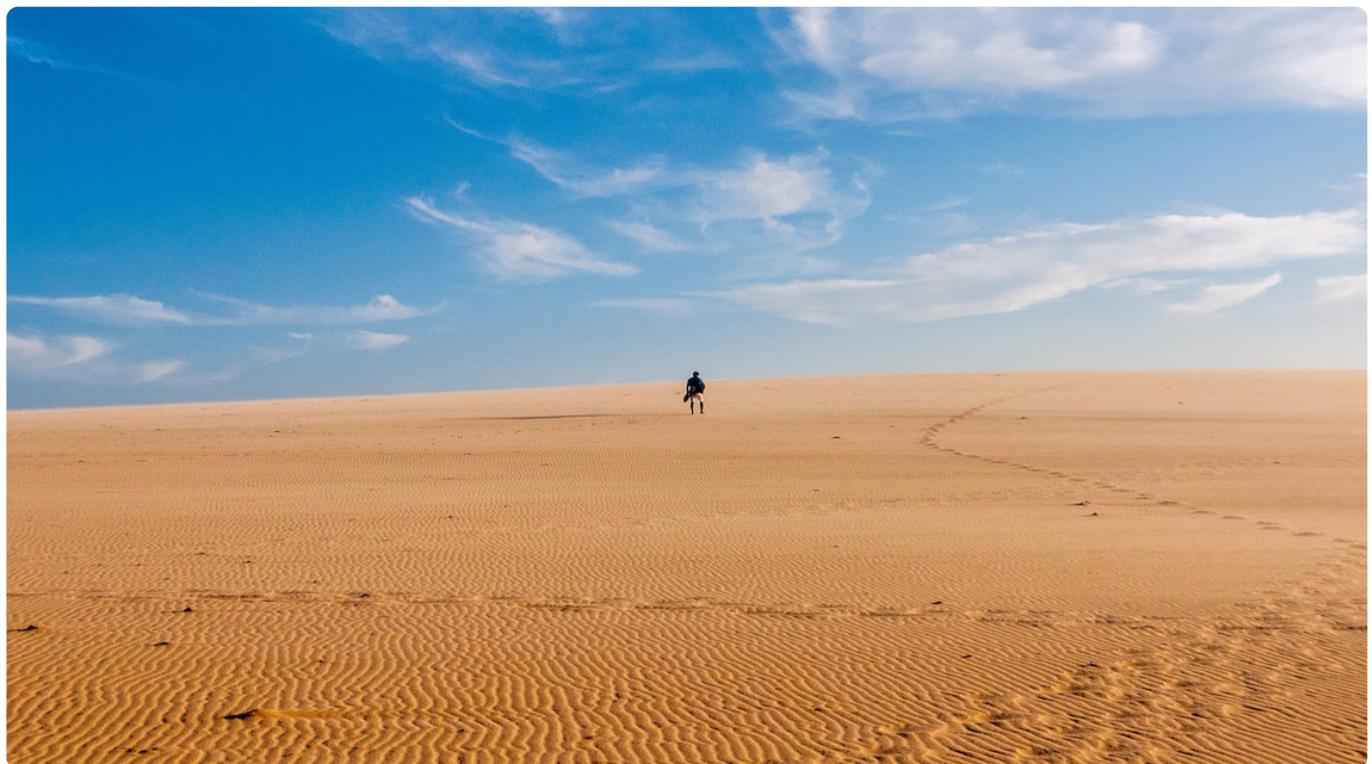


26 | Fork/Join：单机版的MapReduce

2019-04-27 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

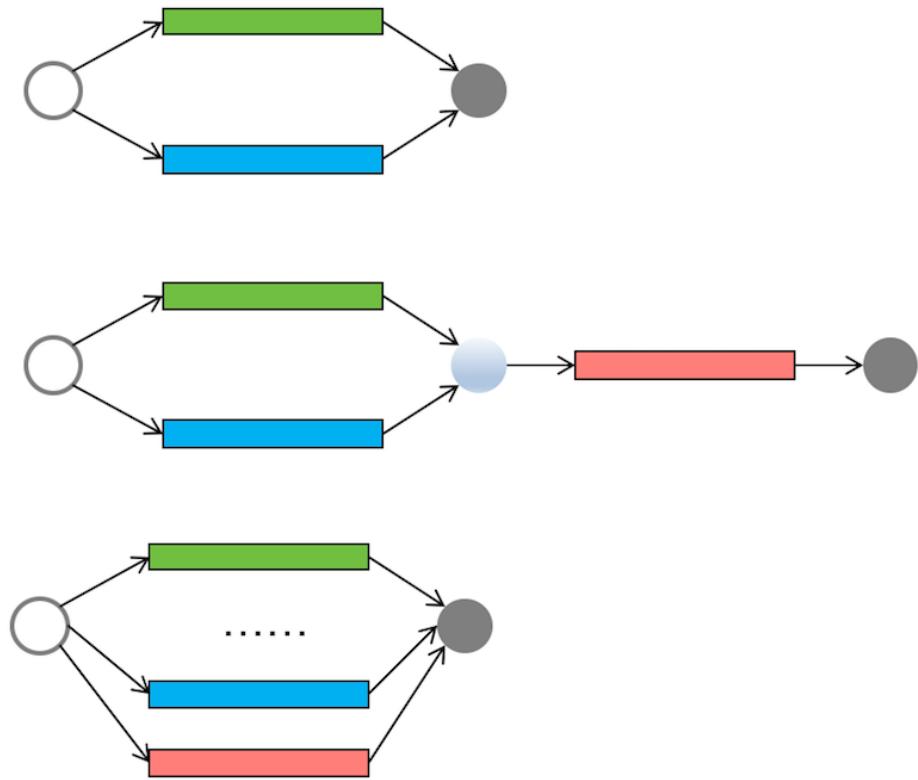
时长 09:38 大小 8.84M



前面几篇文章我们介绍了线程池、Future、CompletableFuture 和 CompletionService，仔细观察你会发现这些工具类都是在帮助我们站在任务的视角来解决并发问题，而不是让我们纠缠在线程之间如何协作的细节上（比如线程之间如何实现等待、通知等）。**对于简单的并行任务，你可以通过“线程池 + Future”的方案来解决；如果任务之间有聚合关系，无论是 AND 聚合还是 OR 聚合，都可以通过 CompletableFuture 来解决；而批量的并行任务，则可以通过 CompletionService 来解决。**

我们一直讲，并发编程可以分为三个层面的问题，分别是分工、协作和互斥，当你关注于任务的时候，你会发现你的视角已经从并发编程的细节中跳出来了，你应用的更多的是现实世界的思维模式，类比的往往是现实世界里的分工，所以我把线程池、Future、CompletableFuture 和 CompletionService 都列到了分工里面。

下面我用现实世界里的工作流程图描述了并发编程领域的简单并行任务、聚合任务和批量并行任务，辅以这些流程图，相信你一定能将你的思维模式转换到现实世界里来。



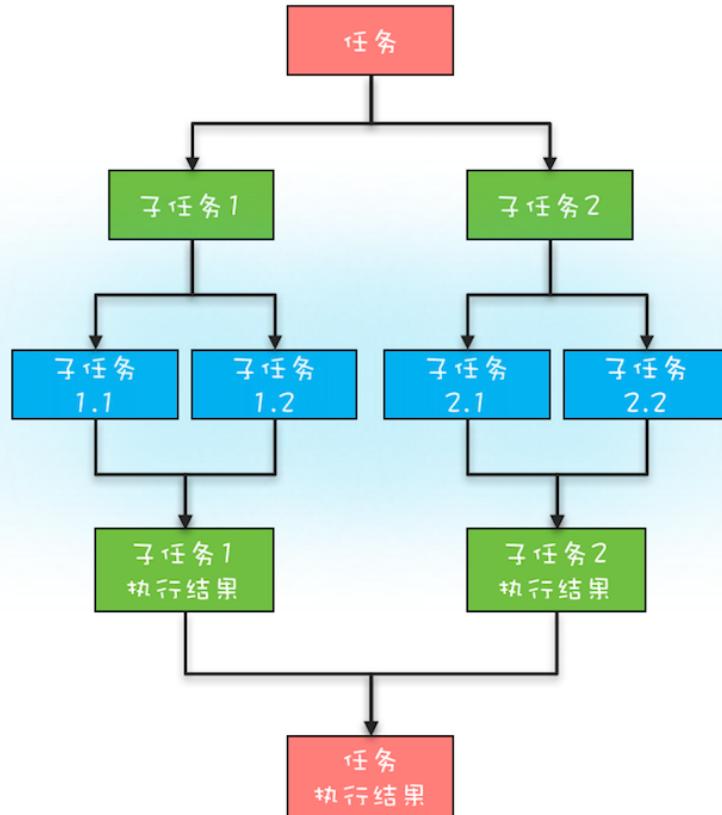
从上到下，依次为简单并行任务、聚合任务和批量并行任务示意图

上面提到的简单并行、聚合、批量并行这三种任务模型，基本上能够覆盖日常工作中的并发场景了，但还是不够全面，因为还有一种“分治”的任务模型没有覆盖到。**分治**，顾名思义，即分而治之，是一种解决复杂问题的思维方法和模式；具体来讲，指的是**把一个复杂的问题分解成多个相似的子问题，然后再把子问题分解成更小的子问题，直到子问题简单到可以直接求解**。理论上来讲，解决每一个问题都对应着一个任务，所以对于问题的分治，实际上就是对于任务的分治。

分治思想在很多领域都有广泛的应用，例如算法领域有分治算法（归并排序、快速排序都属于分治算法，二分法查找也是一种分治算法）；大数据领域知名的计算框架 MapReduce 背后的思想也是分治。既然分治这种任务模型如此普遍，那 Java 显然也需要支持，Java 并发包里提供了一种叫做 Fork/Join 的并行计算框架，就是用来支持分治这种任务模型的。

分治任务模型

这里你需要先深入了解一下分治任务模型，分治任务模型可分为两个阶段：一个阶段是**任务分解**，也就是将任务迭代地分解为子任务，直至子任务可以直接计算出结果；另一个阶段是**结果合并**，即逐层合并子任务的执行结果，直至获得最终结果。下图是一个简化的分治任务模型图，你可以对照着理解。



简版分治任务模型图

在这个分治任务模型里，任务和分解后的子任务具有相似性，这种相似性往往体现在任务和子任务的算法是相同的，但是计算的数据规模是不同的。具备这种相似性的问题，我们往往都采用递归算法。

Fork/Join 的使用

Fork/Join 是一个并行计算的框架，主要就是用来支持分治任务模型的，这个计算框架里的 **Fork** 对应的是分治任务模型里的**任务分解**，**Join** 对应的是**结果合并**。Fork/Join 计算框架主要包含两部分，一部分是**分治任务的线程池 ForkJoinPool**，另一部分是**分治任务 ForkJoinTask**。这两部分的关系类似于 ThreadPoolExecutor 和 Runnable 的关系，都可以理解为提交任务到线程池，只不过分治任务有自己独特类型 ForkJoinTask。

ForkJoinTask 是一个抽象类，它的方法有很多，最核心的是 fork() 方法和 join() 方法，其中 fork() 方法会异步地执行一个子任务，而 join() 方法则会阻塞当前线程来等待子任务的执行结果。ForkJoinTask 有两个子类——RecursiveAction 和 RecursiveTask，通过名字你就应该能知道，它们都是用递归的方式来处理分治任务的。这两个子类都定义了抽象方法 compute()，不过区别是 RecursiveAction 定义的 compute() 没有返回值，而 RecursiveTask 定义的 compute() 方法是有返回值的。这两个子类也是抽象类，在使用的时候，需要你定义子类去扩展。

接下来我们就来实现一下，看看如何用 Fork/Join 这个并行计算框架计算斐波那契数列（下面的代码源自 Java 官方示例）。首先我们需要创建一个分治任务线程池以及计算斐波那契数列的分治任务，之后通过调用分治任务线程池的 invoke() 方法来启动分治任务。由于计算斐波那契数列需要有返回值，所以 Fibonacci 继承自 RecursiveTask。分治任务 Fibonacci 需要实现 compute() 方法，这个方法里面的逻辑和普通计算斐波那契数列非常类似，区别之处在于计算 Fibonacci(n - 1) 使用了异步子任务，这是通过 f1.fork() 这条语句实现的。

 复制代码

```
1 static void main(String[] args){  
2     // 创建分治任务线程池  
3     ForkJoinPool fjp =  
4         new ForkJoinPool(4);  
5     // 创建分治任务  
6     Fibonacci fib =  
7         new Fibonacci(30);  
8     // 启动分治任务  
9     Integer result =  
10        fjp.invoke(fib);  
11    // 输出结果  
12    System.out.println(result);  
13 }  
14 // 递归任务  
15 static class Fibonacci extends  
16     RecursiveTask<Integer>{  
17     final int n;  
18     Fibonacci(int n){this.n = n;}  
19     protected Integer compute(){  
20         if (n <= 1)  
21             return n;  
22         Fibonacci f1 =  
23             new Fibonacci(n - 1);  
24         // 创建子任务  
25         f1.fork();  
26         Fibonacci f2 =  
27             new Fibonacci(n - 2);
```

```
28     // 等待子任务结果，并合并结果
29     return f2.compute() + f1.join();
30 }
31 }
```

ForkJoinPool 工作原理

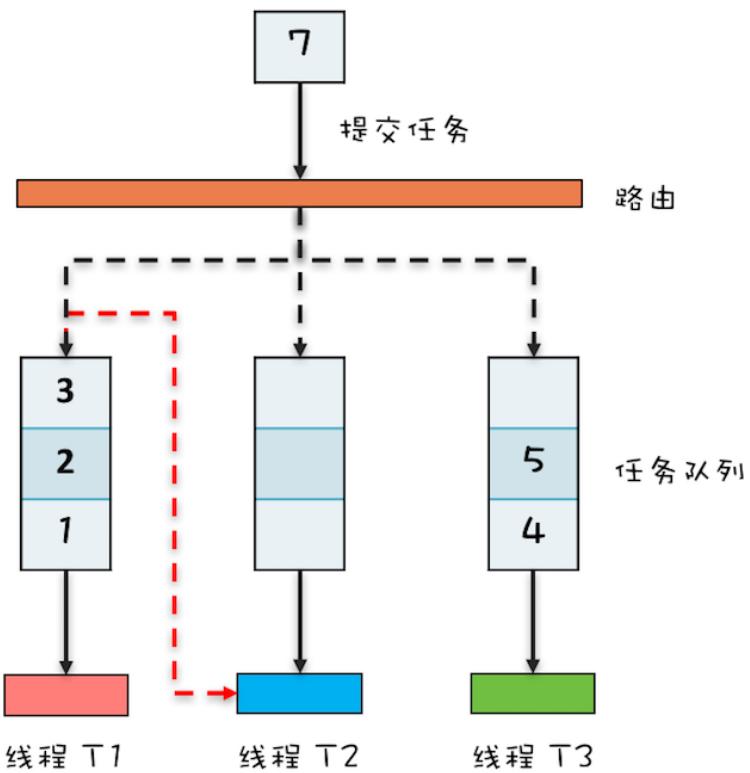
Fork/Join 并行计算的核心组件是 ForkJoinPool，所以下面我们就来简单介绍一下 ForkJoinPool 的工作原理。

通过专栏前面文章的学习，你应该已经知道 ThreadPoolExecutor 本质上是一个生产者 - 消费者模式的实现，内部有一个任务队列，这个任务队列是生产者和消费者通信的媒介； ThreadPoolExecutor 可以有多个线程，但是这些线程都共享一个任务队列。

ForkJoinPool 本质上也是一个生产者 - 消费者的实现，但是更加智能，你可以参考下面的 ForkJoinPool 工作原理图来理解其原理。 ThreadPoolExecutor 内部只有一个任务队列，而 ForkJoinPool 内部有多个任务队列，当我们通过 ForkJoinPool 的 invoke() 或者 submit() 方法提交任务时，ForkJoinPool 根据一定的路由规则把任务提交到一个任务队列中，如果任务在执行过程中会创建出子任务，那么子任务会提交到工作线程对应的任务队列中。

如果工作线程对应的任务队列空了，是不是就没活儿干了呢？不是的，ForkJoinPool 支持一种叫做“**任务窃取**”的机制，如果工作线程空闲了，那它可以“窃取”其他工作任务队列里的任务，例如下图中，线程 T2 对应的任务队列已经空了，它可以“窃取”线程 T1 对应的任务队列的任务。如此一来，所有的工作线程都不会闲下来了。

ForkJoinPool 中的任务队列采用的是双端队列，工作线程正常获取任务和“窃取任务”分别是从任务队列不同的端消费，这样能避免很多不必要的数据竞争。我们这里介绍的仅仅是简化后的原理，ForkJoinPool 的实现远比我们这里介绍的复杂，如果你感兴趣，建议去看它的源码。



ForkJoinPool 工作原理图

模拟 MapReduce 统计单词数量

学习 MapReduce 有一个入门程序，统计一个文件里面每个单词的数量，下面我们来看看如何用 Fork/Join 并行计算框架来实现。

我们可以先用二分法递归地将一个文件拆分成更小的文件，直到文件里只有一行数据，然后统计这一行数据里单词的数量，最后再逐级汇总结果，你可以对照前面的简版分治任务模型图来理解这个过程。

思路有了，我们马上来实现。下面的示例程序用一个字符串数组 `String[] fc` 来模拟文件内容，`fc` 里面的元素与文件里面的行数据一一对应。关键的代码在 `compute()` 这个方法里面，这是一个递归方法，前半部分数据 `fork` 一个递归任务去处理（关键代码 `mr1.fork()`），后半部分数据则在当前任务中递归处理（`mr2.compute()`）。

复制代码

```

1 static void main(String[] args){
2     String[] fc = {"hello world",
3                     "hello me",
4                     "hello fork",
5                     "hello join",
6                     "hello join",
7                     "hello fork",
8                     "hello me",
9                     "hello world"};
10    new ForkJoinTask<String>() {
11        public String compute() {
12            int count = 0;
13            for (String s : fc) {
14                if (s.startsWith("hello")) {
15                    count++;
16                }
17            }
18            return String.valueOf(count);
19        }
20    }.join();
21 }

```

```
6         "fork join in world");
7 // 创建 ForkJoin 线程池
8 ForkJoinPool fjp =
9     new ForkJoinPool(3);
10 // 创建任务
11 MR mr = new MR(
12     fc, 0, fc.length);
13 // 启动任务
14 Map<String, Long> result =
15     fjp.invoke(mr);
16 // 输出结果
17 result.forEach((k, v)->
18     System.out.println(k+":"+v));
19 }
20 //MR 模拟类
21 static class MR extends
22     RecursiveTask<Map<String, Long>> {
23     private String[] fc;
24     private int start, end;
25     // 构造函数
26     MR(String[] fc, int fr, int to){
27         this.fc = fc;
28         this.start = fr;
29         this.end = to;
30     }
31     @Override protected
32     Map<String, Long> compute(){
33         if (end - start == 1) {
34             return calc(fc[start]);
35         } else {
36             int mid = (start+end)/2;
37             MR mr1 = new MR(
38                 fc, start, mid);
39             mr1.fork();
40             MR mr2 = new MR(
41                 fc, mid, end);
42             // 计算子任务，并返回合并的结果
43             return merge(mr2.compute(),
44                         mr1.join());
45         }
46     }
47     // 合并结果
48     private Map<String, Long> merge(
49         Map<String, Long> r1,
50         Map<String, Long> r2) {
51         Map<String, Long> result =
52             new HashMap<>();
53         result.putAll(r1);
54         // 合并结果
55         r2.forEach((k, v) -> {
56             Long c = result.get(k);
57             if (c != null)
```

```
58         result.put(k, c+v);
59     else
60         result.put(k, v);
61 });
62 return result;
63 }
64 // 统计单词数量
65 private Map<String, Long>
66     calc(String line) {
67     Map<String, Long> result =
68         new HashMap<>();
69     // 分割单词
70     String [] words =
71         line.split("\s+");
72     // 统计单词数量
73     for (String w : words) {
74         Long v = result.get(w);
75         if (v != null)
76             result.put(w, v+1);
77         else
78             result.put(w, 1L);
79     }
80     return result;
81 }
82 }
```

总结

Fork/Join 并行计算框架主要解决的是分治任务。分治的核心思想是“分而治之”：将一个大的任务拆分成小的子任务去解决，然后再把子任务的结果聚合起来从而得到最终结果。这个过程非常类似于大数据处理中的 MapReduce，所以你可以把 Fork/Join 看作单机版的 MapReduce。

Fork/Join 并行计算框架的核心组件是 ForkJoinPool。ForkJoinPool 支持任务窃取机制，能够让所有线程的工作量基本均衡，不会出现有的线程很忙，而有的线程很闲的状况，所以性能很好。Java 1.8 提供的 Stream API 里面并行流也是以 ForkJoinPool 为基础的。不过需要你注意的是，默认情况下所有的并行流计算都共享一个 ForkJoinPool，这个共享的 ForkJoinPool 默认的线程数是 CPU 的核数；如果所有的并行流计算都是 CPU 密集型计算的话，完全没有问题，但是如果存在 I/O 密集型的并行流计算，那么很可能会因为一个很慢的 I/O 计算而拖慢整个系统的性能。所以**建议用不同的 ForkJoinPool 执行不同类型的计算任务。**

如果你对 ForkJoinPool 详细的实现细节感兴趣，也可以参考[Doug Lea 的论文](#)。

课后思考

对于一个 CPU 密集型计算程序，在单核 CPU 上，使用 Fork/Join 并行计算框架是否能够提高性能呢？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

极客时间

Java 并发编程实战

全面系统提升你的并发编程能力

王宝令
资深架构师

新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | CompletionService：如何批量执行异步任务？

下一篇 27 | 并发工具类模块热点问题答疑

精选留言 (27)

 写留言



爱吃回锅肉...

2019-04-28

 11

<https://www.liaoxuefeng.com/article/001493522711597674607c7f4f346628a761454>

老师，您好，我在廖雪峰网站中也看到forkjoin使用方式。讲解了，为啥不使用两次fork，分享出来给大家看看。

展开 ▼

作者回复: 用两次fork()在join的时候，需要用这样的顺序：a.fork(); b.fork(); b.join(); a.join();这个要求在JDK官方文档里有说明。

如果是一不小心写成a.fork(); b.fork(); a.join(); b.join();就会有大神廖雪峰说的问题。

建议还是用fork() + compute()，这种方式的执行过程普通人还是能理解的，fork() + fork()内部做了很多优化，我这个普通人看的实在是头痛。

感谢分享啊。我觉得讲的挺好的。用这篇文章的例子理解fork() + compute()很到位。

◀ ▶



锦

2019-04-27

↑ 9

CPU同一时间只能处理一个线程，所以理论上，纯cpu密集型计算任务单线程就够了。多线程的话，线程上下文切换带来的线程现场保存和恢复也会带来额外开销。但实际上可能要经过测试才知道。

展开 ▼

作者回复: ↗

◀ ▶



右耳听海

2019-04-27

↑ 5

请教老师一个问题，merge函数里的mr2.compute先执行还是mr1.join先执行，这两个参数是否可交换位置

作者回复: 我觉得不可以，如果join在前面会先让当前线程阻塞在join()上。当join()执行完才会执行mr2.compute()，这样并行度就下来了。

◀ ▶



尹圣



2019-04-29

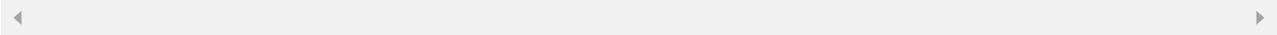


看到分治任务立马就想到归并排序，用Fork/Join又重新实现了一遍，

```
/**  
 * Ryzen 1700 8核16线程 3.0 GHz  
 */  
@Test...
```

展开▼

作者回复: 凸凸凸举一反三了😊



linqw

2019-04-27



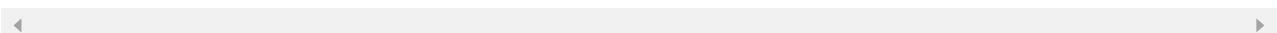
2

以前在面蚂蚁金服时，也做过类似的题目，从一个目录中，找出所有文件里面单词出现的 top100，那时也是使用服务提供者，从目录中找出一个或者多个文件（防止所有文件一次性加载内存溢出，也为了防止文件内容过小，所以每次都确保读出的行数10万行左右），然后使用fork/join进行单词的统计处理，设置处理的阈值为20000。

课后习题：单核的话，使用单线程会比多线程快，线程的切换，恢复等都会耗时，并且...

展开▼

作者回复: 凸



QQ怪

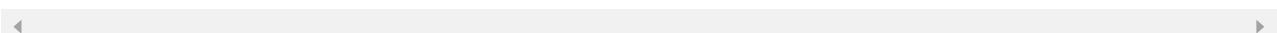
2019-05-24



1

学习了老师的分享，现在就已经在工作用到了，的确是在同事面前好好装了一次逼

作者回复: 凸说明你很有悟性😊



狂风骤雨

2019-04-29



1

好希望工作当中能有老师这样一位大牛，能为我答疑解惑

展开▼

作者回复: 我知道的就这些，都写出来了😊，显然我不是大牛😊



朱晋君

2019-04-28

1

老师，请问为什么不能merge mr1.compute和mr2..compute或者mr1.join和mr2的join呢？

作者回复: compute+compute相当于没用forkjoin，都在一个线程里跑的。如果用join+join也可以，不过jdk官方有个建议，顺序要用：a.fork(); b.fork(); b.join(); a.join();否则性能有问题。所以还是用fork+compute更简单。



ban

2019-04-27

1

“如果存在 I/O 密集型的并行流计算，那么很可能会因为一个很慢的 I/O 计算而拖慢整个系统的性能。”

老师这个问题，这句话前面的文字也看到，但是不太懂。如果共用一个线程池，但是不是有多个线程，如果一个线程操作I/O，应该不影响其他线程吧，其他线程还能继续执行，
...
展开▼

作者回复: 前提是有很多请求并发访问这个很慢的I/O计算，我们这的并发程序，往往都有很多请求同时访问的



êwěn

2019-04-27

1

老师，fork是fork调用者的子任务还是表示下面new出来的任务是子任务？

展开▼

作者回复: fork是fork调用者这个子任务加入到任务队列里



木木匠

1



2019-04-27

单核cpu上多线程会导致线程的上下文切换，还不如单核单线程处理的效率高。



蓝天白云看...

2019-06-02



join会阻塞线程吗？如果阻塞线程，而线程池里的线程个数又有线，那么递归几次之后所有线程不都全阻塞了吗！



蓝天白云看...

2019-06-02



join()如果阻塞线程的话，就不是并行任务了，任务有可能永远都完不成了吧

展开 ▼



约书亚

2019-06-02



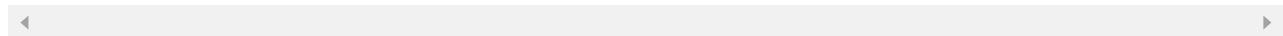
“如果任务在执行过程中会创建出子任务，那么子任务会提交到工作线程对应的任务队列中。”

抱歉来得晚了些。

上面这句话不太理解。如果“创建子任务”指的是fork的话，是不是应该提交到其他任务队列中？否则岂不是全部在一个队列中，被一个线程处理，其他线程对应的队列都是空...

展开 ▼

作者回复: 所有任务都会创建子任务，如果只有一个任务会创建子任务，那就只能steal了



null

2019-05-27



@王伟童鞋的问题，我们也有这场景：通过手机号查询商家信息。

我们是在 redis 里维护(手机号，商家号)关联关系，在 redis 里通过手机号查询商家号，就知道该去哪个库表查询商家具体信息了。

内存开销：...

展开 ▼





Geek ebda9...
2019-05-13

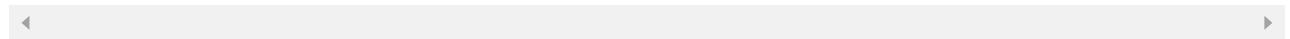


如果所有的并行流计算都是 CPU 密集型计算的话，完全没有问题，但是如果存在 I/O 密集型的并行流计算，那么很可能会因为一个很慢的 I/O 计算而拖慢整个系统的性能。

老师这里的意思是不是，如果有耗时的i/o计算，需要用单独的forkjoin pool 来处理这个计算，在程序设计的时候就要跟其他cpu密集计算的任务分开处理？

展开 ▼

作者回复: 是的



张三

2019-04-29



ForkJoinTask这个抽象类的 fork() 和 join () 底层是怎么实现的呢？

展开 ▼

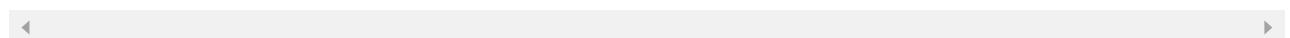


右耳听海
2019-04-28



这里用的递归调用，数据量大的时候会不会粘溢出，虽然这里用的二分，时间复杂度为 logn

作者回复: 我觉得会

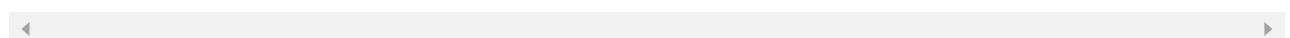


王伟
2019-04-28



老师，我现在碰到一个生产问题：用户通过微信小程序进入我们平台，我们只能需要使用用户的手机号去我们商家库中查取该用户的注册信息。在只知道用户手机号的情况下我们需要切换到所有的商家库去查询。这样非常耗时。ps：我们商家库做了分库处理而且数量很多。想请教一下您，这种查询该如何做？

作者回复: 可以加redis缓存看看，也可以加本地缓存。不要让流量直接打到数据库上





密码123456

2019-04-28



我记得之前提到过，使用线程数目大小的方法。如果io耗时过长可以多加线程数量，能够提升性能。如果io耗时过短，增加线程数量就不能，提升性能了？不知道是否能够对应，这个题目的答案？

展开 ▼