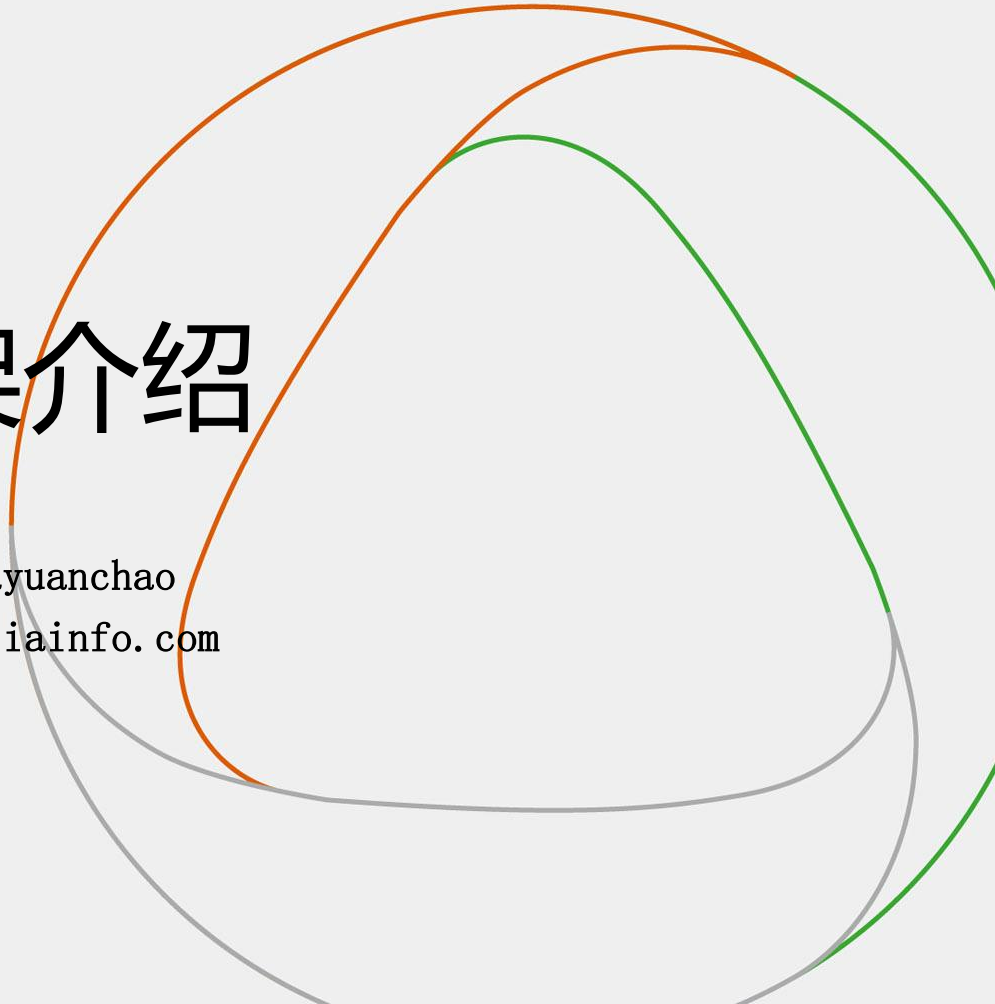
AsiaInfo 亚信

Fork/Join框架介绍

By mayuanchao

mayc@asiainfo.com

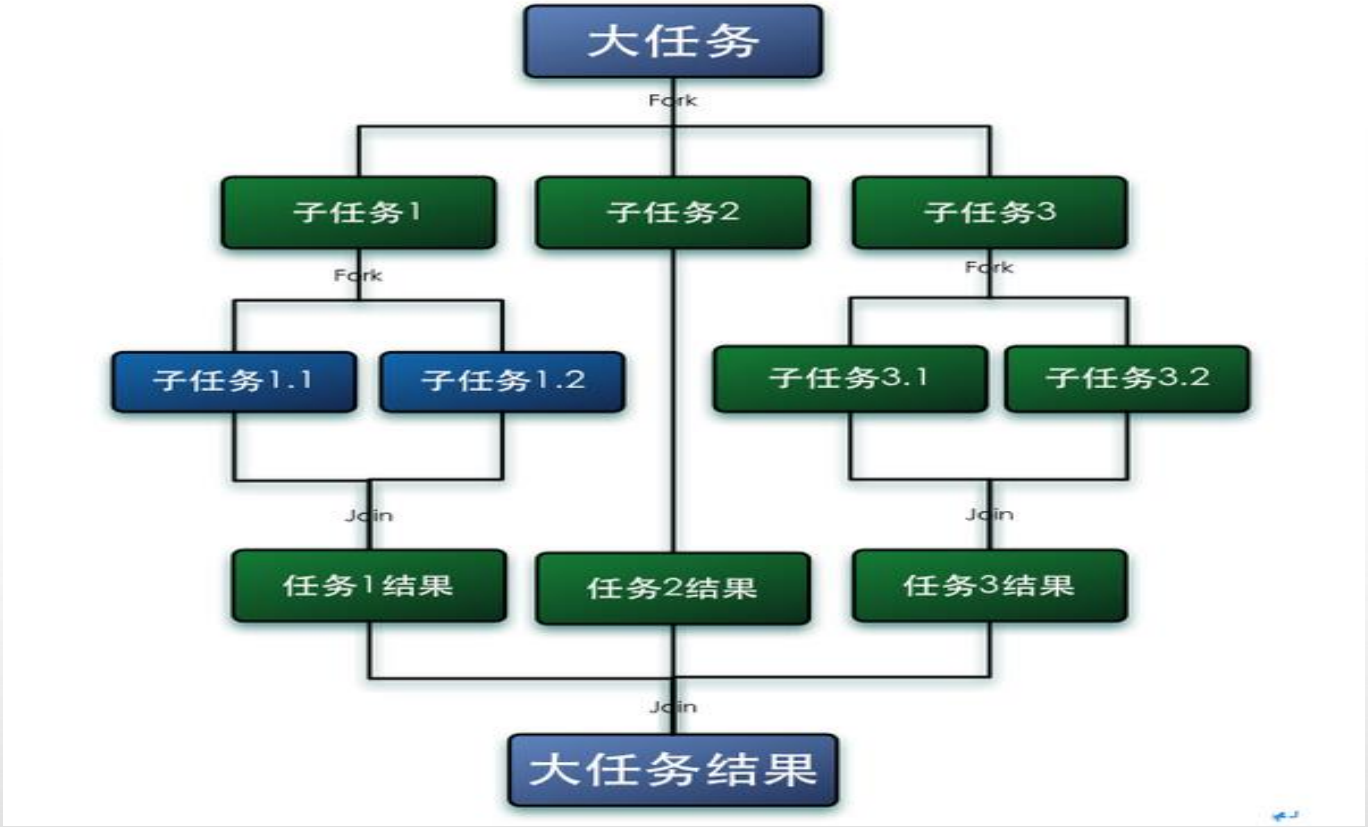


- 介绍
- 创建一个Fork/Join池
- 加入任务的结果
- 影响ForkJoin加速效果的因素
- Map/Reduce?
- 总结
- ?

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，最后得到这个大任务的结果。比如计算1+2+。。＋10000，可以分割成10个子任务，每个子任务分别对1000个数进行求和，最终汇总这10个子任务的结果。Fork/Join的运行流程图如下：

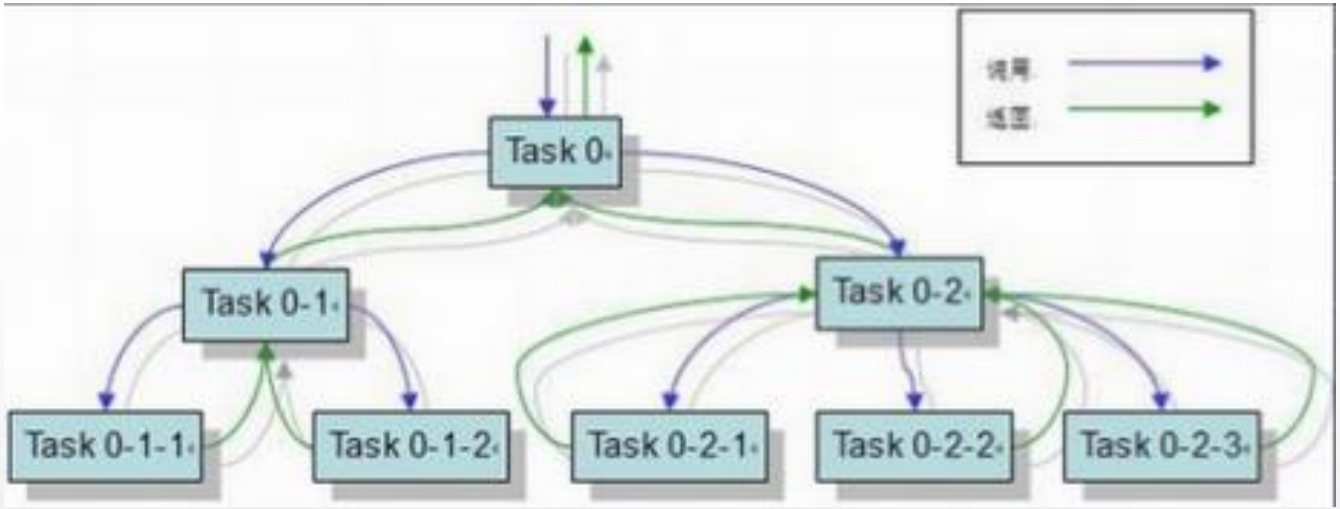

大任务
Fork
子任务1
子任务2
子任务3
Fork
Fork
子任务1.1
子任务1.2
子任务3.1
子任务3.2
Join
Join
任务1结果
任务2结果
任务3结果
Join
大任务结果

1、Divide and conquer（分而治之算法）

大家肯定听说过有一种叫 divide and conquer（分而治之算法）的策略。分而治之算法本质：

它本质上将原有的问题集合拆分成两个子问题，然后再针对这些子问题进行进一步的处理，直到子问题已经得到解决。在这些子问题解决后上面的部分再将这些问题合并起来就得到了我们想要的答案。这个问题主要是针对单线程的运行场景。在把并行的一些思想考虑进来的时候，我发现他们可以碰撞出美丽的火花。我们仔细再来看一下前面的这一类型的问题，他们的本质上是将一个问题划分成多个子问题，然后再逐个的去解决子问题。在很多情况下，他们这些子问题是互不相干的。也就是说，我们针对他们每个执行的子问题，可以让他们采用独立的线程来运行。这样的话我们可以充分的发挥现在并行处理器的优势。

下图就是divide and conquer策略的处理问题方式：

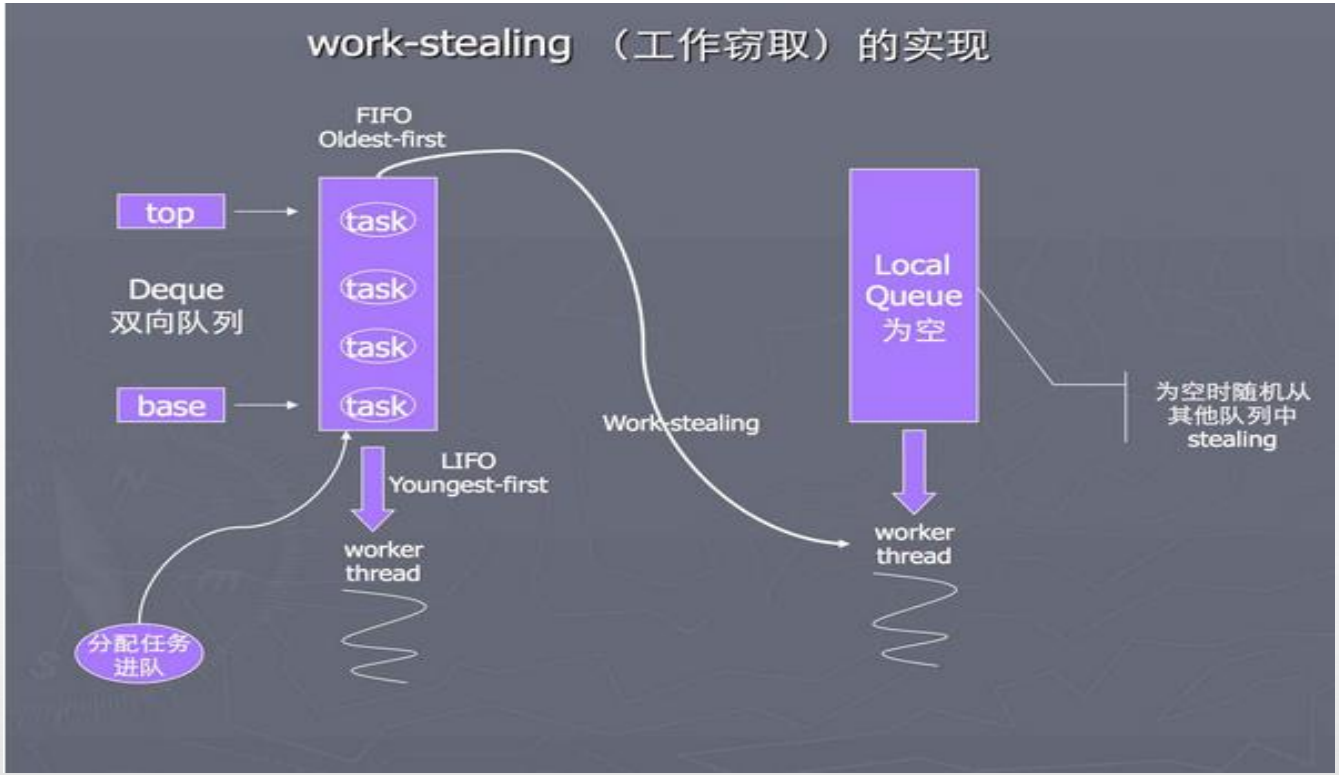


2、工作窃取算法

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：

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。


work-stealing （工作窃取）的实现
FIFO
Oldest-first
top
Deque
双向队列
base
task
task
task
task
LIFO
Youngest-first
worker
thread
分配任务
进队
Work-stealing
Local
Queue
为空
worker
thread
为空时随机从
其他队列中
stealing

3、Fork/Join框架介绍：

Fork/Join框架基于以下两种操作：

- Fork操作：把一个大任务分割成若干个小任务。
- Join操作：汇集多个小任务的结果然后得到大任务的结果。

Fork/Join 是基于Divide and conquer和work-stealing算法。当一个任务正在等待它使用join操作创建的子任务的结 束时，执行这个任务的线程（工作线程）查找其他未被执行的任务并开始它的执行。通过这种方式，线程充分利用它们的运行时间，从而提高了应用程序的性能。参看下图：

Fork/Join框架的核心是由以下两个类：

- 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。

4、Fork/Join框架实现原理：

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

ForkJoinTask的fork方法实现原理。当我们调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步的执行这个任务，然后立即返回结果。

代码如下：

```
public final ForkJoinTask fork() {
        ((ForkJoinWorkerThread) Thread.currentThread()) .pushTask(this);
        return this;
}
```

PushTask方法把当前任务存放在ForkJoinTask 数组queue里。然后再调用ForkJoinPool的signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下：

```
01 final void pushTask(ForkJoinTask t) {
02         ForkJoinTask[] q; int s, m;
03         if ((q = queue) != null) {    // ignore if queue removed
04             long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;
05             UNSAFE.putOrderedObject(q, u, t);
06             queueTop = s + 1;         // or use putOrderedInt
07             if ((s -= queueBase) <= 2)
08                 pool.signalWork();
09     else if (s == m)
10                 growQueue();
11         }
12     }
```

ForkJoinTask的join方法实现原理。Join方法的主要作用是阻塞当前线程并等待获取结果。让我们一起看看ForkJoinTask的join方法的实现，代码如下：

```
01 public final V join() {
02         if (doJoin() != NORMAL)
03             return reportResult();
04         else
05             return getRawResult();
06 }
07 private V reportResult() {
08         int s; Throwable ex;
09         if ((s = status) == CANCELLED)
10             throw new CancellationException();
11 if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)
12             UNSAFE.throwException(ex);
13         return getRawResult();
14 }
```

首先，它调用了doJoin()方法，通过doJoin()方法得到当前任务的状态来判断返回什么结果。

任务状态有四种：

1、已完成（NORMAL）；

2、被取消（CANCELLED）；

3、信号（SIGNAL）；

4、出现异常（EXCEPTIONAL）；

如果任务状态是已完成，则直接返回任务结果。

如果任务状态是被取消，则直接抛出CancellationException。

如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析下doJoin()方法的实现代码：

```
01 private int doJoin() {
02         Thread t; ForkJoinWorkerThread w; int s; boolean completed;
03         if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {
04             if ((s = status) < 0)
05  return s;
06             if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {
07                 try {
08                     completed = exec();
09                 } catch (Throwable rex) {
10                     return setExceptionalCompletion(rex);
11                 }
12                 if (completed)
13                     return setCompletion(NORMAL);
14             }
15             return w.joinTask(this);
16         }
17         else
18             return externalAwaitDone();
19     }
```

在doJoin()方法里，首先通过查看任务的状态，看任务是否已经执行完了，如果执行完了，则直接返回任务状态，如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成了，则设置任务状态为NORMAL，如果出现异常，则纪录异常，并将任务状态设置为EXCEPTIONAL。

创建一个Fork/Join池

如何使用Fork/Join框架的基本元素。它包括：

- 创建一个ForkJoinPool对象来执行任务。
- 创建一个ForkJoinPool执行的ForkJoinTask类。
- 在这个任务中，你将使用Java API文档推荐的结构：

```
If (problem size > default size){
        tasks=divide(task);
        execute(tasks);
       } else {
        resolve problem using another algorithm;
        }
```

如何做？

有这样个需求，修改产品列表的价格，这个产品列表比较大。。。

思路：

最初我们是实现一个任务来修改价格，任务最初是负责更新一个队列中的所有元素，我们使用10作为参考大小，如果一个任务必须更新超过10个元素，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。

伪代码：

```
If (Products size > default size==10){
  tasks=divide(task);
  execute(tasks);
 } else{
  product.setPrice(“修改价格”);
 }
```

我们按以下步骤来实现这个示例

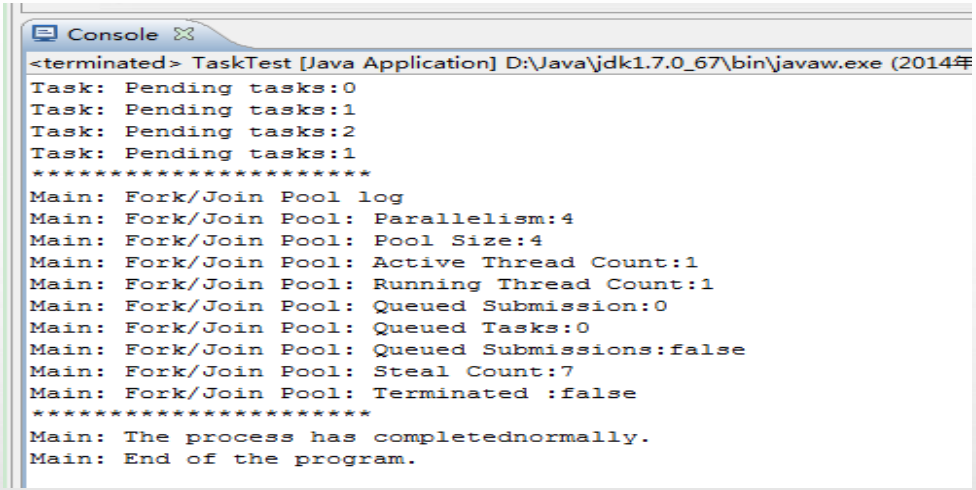
1. 创建类Product，将用来存储产品的名称和价格。
    1. public class Product {private String name;private double price;}
2. 创建ProductListGenerator类，用来产生随机产品的数列。
3. 创建Task类，指定它继承RecursiveAction类，实现完compute方法。

```
@Override
protected void compute() {
    if ((last - first) < 10) {
        updatePrices(); //如果last和first的差小于10（任务只能更新价格小于10的产品），
                        //使用updatePrices()方法递增的设置产品的价格。
    } else {//如果last和first的差大于或等于10，
            //则创建两个新的Task对象，一个处理产品的前半部分，
            //另一个处理产品的后半部分，然后在ForkJoinPool中，使用invokeAll()方法执行它们。
        int middle = (last + first) / 2;
        System.out.printf("Task: Pending tasks:%s\n", getQueuedTaskCount());
        Task t1 = new Task(products, first, middle + 1, increment);
        Task t2 = new Task(products, middle + 1, last, increment);
        invokeAll(t1, t2);
    }
}
```

3、 创建一个新的Task对象，用来更新产品队列中的产品。first参数使用值0，last参数使用值10000（产品数列的大小）。

1. Task task=new Task(products, 0, products.size(), 0.20);

4、用无参构造器创建ForkJoinPool对象 ForkJoinPool pool=new ForkJoinPool();然后使用pool.execute(task) 来执行任务。



它是如何工作的？

- 在这个示例中，我们已经创建一个ForkJoinPool对象和一个在池中执行的ForkJoinTask类的子类。为了创建ForkJoinPool对象，你已经使用了无参构造器，所以它会以默认的配置来执行。它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。在这个例子中，我们已经使用了推荐的结构来实现任务。如果这个任务更新超过10产品，它将被分解成两部分，并创建两个任务，一个任务执行一部分。你已经在Task类中使用first和last属性，用来了解这个任务要更新的产品队列的位置范围。你已经使用first和last属性，只复制产品数列一次，而不是为每个任务创建不同的数列。
- 调用invokeAll()方法，执行每个任务所创建的子任务。这是一个同步调用，这个任务在继续（可能完成）它的执行之前，必须等待子任务的结束。当任务正在等待它的子任务（结束）时，正在执行它的工作线程执行其他正在等待的任务。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。

- ForkJoinTask类的invokeAll()方法是执行者（Executor）和Fork/Join框架的一个主要区别。在执行者框架中，所有任务被提交给执行者，而在这种情况下，这些任务包括执行和控制这些任务的方法都在池内。你已经在Task类中使用invokeAll()方法，它是继承了继承ForkJoinTask类的RecursiveAction类。
- 你使用execute()方法提交唯一的任务给这个池，用来执行所有产品数列。在这种情况下，它是一个异步调用，而主线程继续它的执行。

不止这些...

- ForkJoinPool类提供其他的方法，用来执行一个任务。这些方法如下：
- execute (Runnable task)：这是在这个示例中，使用的execute()方法的另一个版本。在这种情况下，你可以提交一个Runnable对象给ForkJoinPool类。注意：ForkJoinPool类不会对Runnable对象使用work-stealing算法。它（work-stealing算法）只用于ForkJoinTask对象。
- invoke(ForkJoinTask<T> task)：当execute()方法使用一个异步调用ForkJoinPool类，正如你在本示例中所学的，invoke()方法使用同步调用ForkJoinPool类。这个调用不会（立即）返回，直到传递的参数任务完成它的执行。
- 你也可以使用在ExecutorService接口的invokeAll()和invokeAny()方法。这些方法接收一个Callable对象作为参数。ForkJoinPool类不会对Callable对象使用work-stealing算法，所以你最好使用执行者去执行它们。

不止这些...

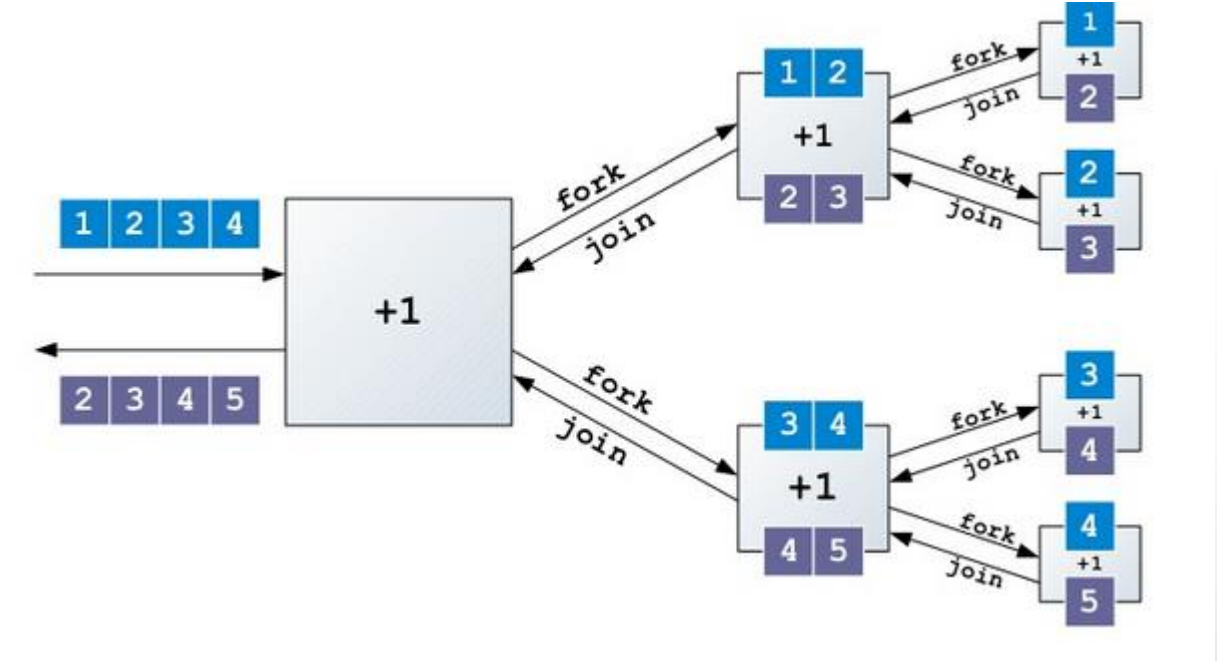
- ForkJoinTask类同样提供在示例中使用的invokeAll()的其他版本。这些版本如下：
- invokeAll(ForkJoinTask<?>... tasks)：这个版本的方法使用一个可变参数列表。你可以传入许多你想要执行的ForkJoinTask对象作为参数。
- invokeAll(Collection<T> tasks)：这个版本的方法接收一个泛型类型T对象的集合（如：一个ArrayList对象，一个LinkedList对象或者一个TreeSet对象）。这个泛型类型T必须是ForkJoinTask类或它的子类。

举个例子，使用 Fork / Join框架来实现一个数组每个数+1

实现思路：

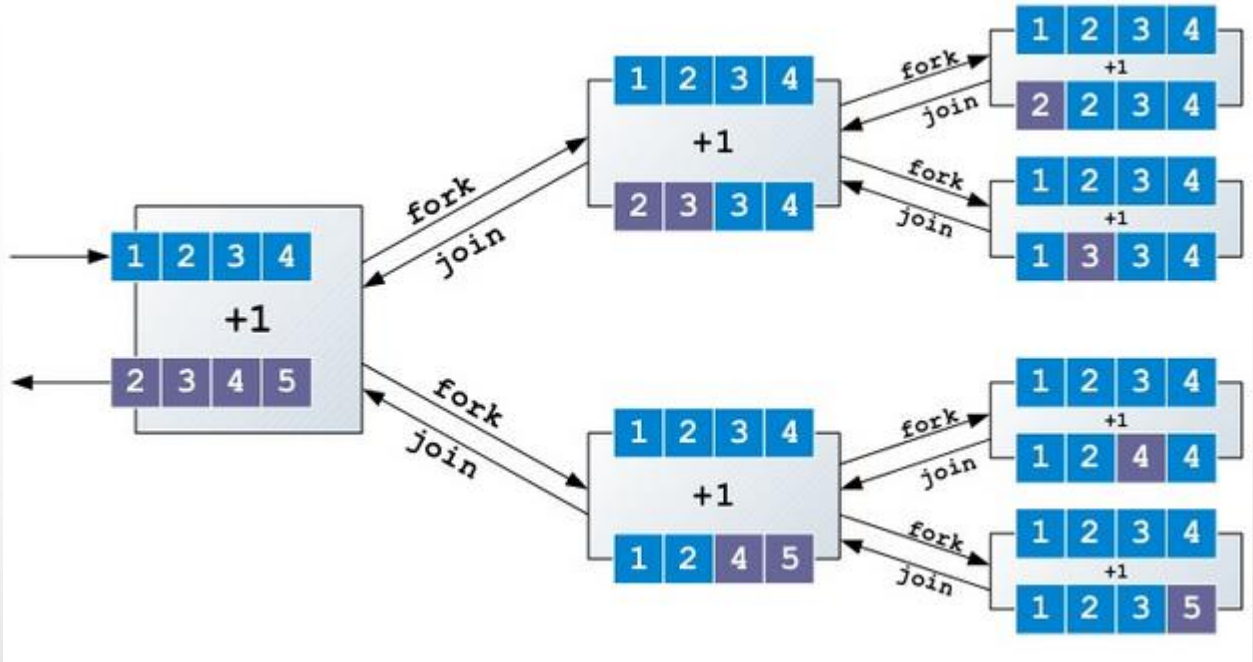
- 将数组分成2个子数组。
- 对左边的数组+1。
- 对右边的数组+1然后左右数组合并。
- 上面着几个步骤会重复递归，每个子数组都要求容量小于上面计算出来的临界值。

RecursiveTask的实现方式



```
20 public class RecursiveTask1 extends RecursiveTask<int[]> {
21     private int[] source;
22
23     RecursiveTask1(int[] source) {
24         this.source = source;
25     }
26
27     @Override
28     protected int[] compute() {
29         int a = source.length;
30         if (a <= 100) {
31             for (int i = 0; i < a; i++) {
32                 source[i] = source[i] + 1;
33             }
34         } else {
35             int middle = a / 2;
36             int[] left = split(source, 0, middle);
37             int[] right = split(source, middle, source.length);
38             RecursiveTask1 ltask = new RecursiveTask1(left);
39             RecursiveTask1 rtask = new RecursiveTask1(right);
40             ltask.fork();
41             rtask.fork();
42             int[] l = ltask.join();
43             int[] r = rtask.join();
44             for (int i = 0; i < l.length; i++) {
45                 source[i] = l[i];
46             }
47
48             for (int i = 0; i < r.length; i++) {
49                 source[i + l.length] = r[i];
50             }
51         }
52
53         return source;
```

RecursiveAction无返回值得实现方式



```
@SuppressWarnings("serial")
public class ActionRecursive extends RecursiveAction{

    private int[] a;
    private int start;
    private int end;

    ActionRecursive(int [] a,int start,int end){
        this.a = a;
        this.start = start;
        this.end = end;
    }
    @Override
    protected void compute() {
        // TODO Auto-generated method stub
        int l = end- start;
        if(l<=2){
            for(int i=start;i<end;i++){
                a[i] = a[i] + 1;
            }
        }else{
            int mid =  (start + end) / 2;
            ActionRecursive ltask = new ActionRecursive(a,start,mid);
            ActionRecursive rtask = new ActionRecursive(a,mid,end);
            ltask.fork().join();
            rtask.fork().join();
        }
    }
```

RecursiveAction无返回值的实现方式必须继承RecursiveAction类，然后实现compute方法。

再看jdk8中的Arrays.parallelSort例子

原理如下：

- 把数组分成4部分。
- 第一个两部分进行排序,然后将它们合并排序。
- 接下来的两部分进行排序,然后将它们合并排序。
- 上面的步骤递归地重复,直到部分类的大小不小于上述阈值计算。

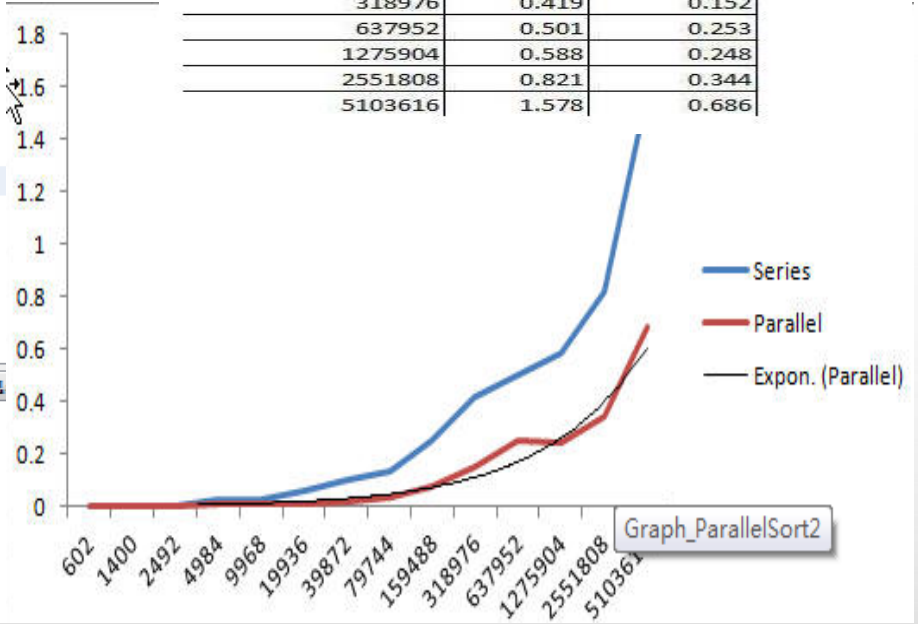
```
    }
    public static void sortCompare(List<Double> arraySourc
        System.out.println("数组大小: "+arraySource.size());
        Double[] myArray = new Double[1];
        myArray = arraySource.toArray(myArray);
        long startTime = System.currentTimeMillis();
        Arrays.sort(myArray);
        long endTime = System.currentTimeMillis();
        System.out.println("线性sort时间: "
                + (endTime - startTime) / 1000.0);

        Double[] myArray2 = new Double[1];
        myArray2 = arraySource.toArray(myArray);
        startTime = System.currentTimeMillis();
        Arrays.parallelSort(myArray2);
        endTime = System.currentTimeMillis();
        System.out.println("并行sort时间: "
                + (endTime - startTime) / 1000.0);
    }
```

制输出 | 输出 - javase8-sample | 测试结果 | HTTP 服务器监视

```
并行sort时间: 0.054
数组大小: 700000
线性sort时间: 0.219
并行sort时间: 0.076
数组大小: 800000
线性sort时间: 0.235
并行sort时间: 0.06
数组大小: 900000
线性sort时间: 0.27
```

Number of element	Serial(sec)	Parallel(sec)
602	0.001	0.004
1400	0.004	0.004
2492	0.007	0.007
4984	0.025	0.008
9968	0.027	0.011
19936	0.064	0.015
39872	0.101	0.024
79744	0.135	0.04
159488	0.253	0.077
318976	0.419	0.152
637952	0.501	0.253
1275904	0.588	0.248
2551808	0.821	0.344
5103616	1.578	0.686



影响ForkJoin加速效果的因素

理想效果是核越多加速效果越好。但是并行不一定更快，参数不对还可能更慢：

1） 并发数，即线程数。一般是可用的cpu数，默认就是这个，一般表现很好。

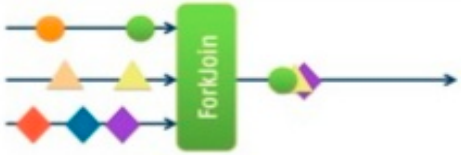
2） 任务切分的粒度。如果切分粒度等于总任务量，一个任务执行，就相当于单线程顺序执行。每个任务执行的计算量，太大的话加速效果有限，不能发挥到最好。相反，太小的话，消耗在任务管理的成本占了主要部分，导致还不如顺序执行的快。

3）需要适当平衡二者。因为还和任务本身的特定有关，所以可以做个基准测试比较一下。而总的执行时间还与任务的规模有关。

4）任务粒度应该适中，多大合适？好像在什么地方上看到说：经验上单个任务为100-10000个基本指令，当然还和任务本身的特定有关。

5）个人感觉多核cpu只适用于解决计算密集型应用，因为实际问题可能IO等其他方面的瓶颈，多核也还是无法充分利用的。

Map/Reduce?



Environment	Single JVM	Cluster
Model	Recursive forking	Often single map
Scales with	Cores/CPUs	Nodes
Worker interaction	Workstealing	No inter-node communication

ForkJoin与MapReduce两个并行计算框架的区别？

MapReduce是把大数据集切分成小数据集，并行分布计算后再合并。

ForkJoin是将一个问题递归分解成子问题，再将子问题并行运算后合并结果。

二者共同点：都是用于执行并行任务的。基本思想都是把问题分解为一个个子问题分别计算，再合并结果。应该说并行计算都是这种思想，彼此独立的或可分解的。从名字上看Fork和Map都有切分的意思，Join和Reduce都有合并的意思，比较类似。

区别：

1）环境差异，分布式 vs 单机多核：ForkJoin设计初衷针对单机多核（处理器数量很多的情况）。MapReduce一开始就明确是针对很多机器组成的集群环境的。也就是说一个是想充分利用多处理器，而另一个是想充分利用很多机器做分布式计算。这是两种不同的的应用场景，有很多差异，因此在细的编程模式方面有很多不同。

2）编程差异：MapReduce一般是：做较大粒度的切分，一开始就先切分好任务然后再执行，并且彼此间在最后合并之前不需要通信。这样可伸缩性更好，适合解决巨大的问题，但限制也更多。ForkJoin可以是较小粒度的切分，任务自己知道该如何切分自己，递归地切分到一组合适大小的子任务来执行，因为是一个JVM内，所以彼此间通信是很容易的，更像是传统编程方式。

总结

Fork/Join框架执行的任务有以下局限性：

- 任务只能使用fork()和join()操作，作为同步机制。如果使用其他同步机制，工作线程不能执行其他任务，当它们在同步操作时。比如，在Fork/Join框架中，你使任务进入睡眠，正在执行这个任务的工作线程将不会执行其他任务，在这睡眠期间内。
- 任务不应该执行I/O操作，如读或写数据文件。
- 任务不能抛出检查异常，它必须包括必要的代码来处理它们。
- 任务中不能获取其它线程变量值，比如CRM中的操作员登录信息UserInfoInterface,必须将UserInfoInterface 信息set到当前线程中。

```
 * @version $Revision: 1515866 $ $Date: 2014-10-28 17:50:30 +0800 (Tue, 28 Oct 2014) $
 */
@SuppressWarnings("serial")
public abstract class CRMRecursiveTask<V> extends RecursiveTask<V>  {

    private UserInfoInterface user;

    public CRMRecursiveTask(UserInfoInterface user){
        this.user = user;
    }
    @Override
    protected V compute() {
        ServiceManager.setServiceUserInfo(user);
        V result = scompute();
        ServiceManager.setServiceUserInfo(null);
        return result;
    }

      protected abstract V scompute();


}
```

```
private static ThreadLocal s_locale = new ThreadLocal();
private static ThreadLocal s_user = new ThreadLocal();
private static final String CURRENT_DOMAIN_ID = "Current_Domain_Id";
private static ThreadLocal s_request = new ThreadLocal();
private static ThreadLocal s_session = new ThreadLocal() {
```

```
public static void setUser(UserInfoInterface user)
{
  s_user.set(user);
}
```

- 自定义在 Fork/Join 框架中运行的任务?
- 监控Fork/Join池?
- 任务中抛出异常?
- 取消任务?

参考资料

- JDK1.7源码
- http://ifeve.com/fork-join-5/
- A Java Fork/Join Framework （pdf）
- Java 理论与实践: 应用 fork-join 框架
- Java 理论与实践: 应用 fork-join 框架，第 2 部分
- JDK 7 中的 Fork/Join 模式
- Java Fork/Join for Parallel Programming
- Fork-Join Development in JavaEE
- Java Fork/Join
- InfoQ上ForkJoin相关文章

Questions ?

AsiaInfo 亚信