ForkJoinPool源码分析之一

2015-10-09 21:56 feiying 0 阅读 203

Executor池的作用就是线程池，主要处理的是无序批量的异步业务；

但是，有一种类似的业务是大数据量的任务，我们需要将其任务进行切割，发挥多核CPU的优势，让多个线程共同完成一项任务，这就是目前比较流行的mapruduce思想；

在JAVA 7以后，java.concurrent包中多了几个对应的API的class：

；

举一个例子，做1到2000000000L的加法，通常我们的程序这么写：

1.

```
a. public class CountTaskOneThread {
b.  public static void main(String[] args) {
c.  long sum=0;
d.  for(long i=1;i<=2000000000L;i++){
e.  sum+=i;
f.  }
g.  System.out.println(sum);
```

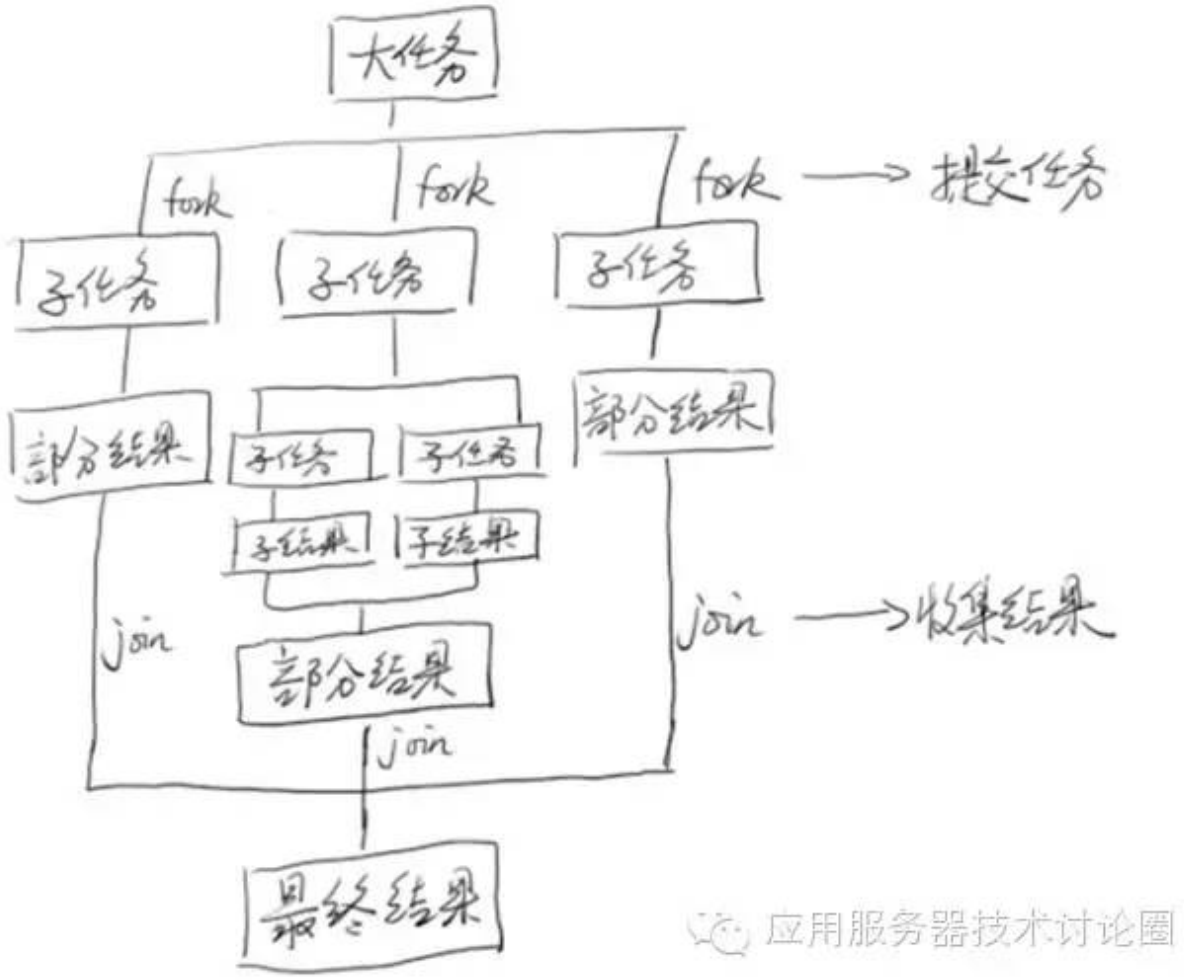
```
h.  }

i. }
```

上述程序中通过一个for循环来完成对加法的操作，程序倒没有什么问题；

因为上述程序仅仅开启了一个线程，而默认的一般情况下，一个java线程对应着一个cpu；

所以，上述这种场景，很有可能仅仅使用了一个cpu，无法发挥现有计算机一整就是16核，24核的多核优势；

我们借助于google提出的mapreduce思想，



将一个大的任务，分割成不同的任务，然后将每一个子任务放到一个cpu上去，让其运行结果，等待每个cpu的处理的结果完成后，将这些子结果进行合并，最终计算出结果，

整体阶段一共分成两个阶段，map阶段就是分割任务的阶段，这些任务被放到了不同的cpu中进行计算；

1.ForkJoinTask

JDK中线程池，它也是需要执行对应的Runable，Callable任务的；

对应的ForkJoinPool其实也是一个线程池，它同样要执行任务，因此也需要有一个ForkJoinTask做业务逻辑的封装，

类似于FutureTask，JDK中定义了下面的类图：

a. ForkJoinTask 因为是mapreduce的任务，因此要进行结果集的合并等操作，因此一定是序列化的;

b. 类似于FutureTask，ForkJoinTask 也需要继承自Future;

c. RecursiveAction，RecursiveTask是子类，

二者之间的区别在于，RecursiveAction没有返回值，也就是其compute方法是void，而RecursiveTask的compute方法是Object类型

以RecursiveTask为例，将上面的 1到2000000000L的加法 的程序重新进行编写，

这里因为要计算结果，所以需要一个返回值，因此业务逻辑就不能使用RecursiveAction，而需要使用RecursiveTask：

```
public class CountTask extends RecursiveTask<Long>{
    private static final int THRESHOLD = 10000;
    private long start;
    private long end;

    public CountTask(long start,long end){
        this.start=start;
        this.end=end;
    }

    public Long compute(){
        long sum=0;
        boolean canCompute = (end-start)<THRESHOLD;
        if(canCompute){
            for(long i=start;i<=end;i++){
                sum +=i;
            }
        }else{
            //分成100个小任务
            long step=(start+end)/100;
            ArrayList<CountTask> subTasks=new ArrayList<CountTask>();
            long pos=start;
            for(int i=0;i<100;i++){
                long lastOne=pos+step;
                if(lastOne>end)lastOne=end;
                CountTask subTask=new CountTask(pos,lastOne);
                pos+=step+1;
                subTasks.add(subTask);
                subTask.fork();
            }
            for(CountTask  t:subTasks){
                sum+=t.join();
            }
        }
        return sum;
    }
}
```


需要注意上述的三个方法的作用：

a. compute方法来完成业务逻辑的，你写的RecursiveTask主要的mapreduce的思想都在这个方法中；

如上述的程序中，start是每一次分段的起始计算值，end是每一次分段的截止计算值，每一次分段计算是在start和end之间，而计算这个程序就是简单的求和；

THRESHOLD属性是阈值，也就是通过这个属性来控制一段有多长，这个程序每一段都是固定的10000，

也就是说，分段一共包含两类：一类是可以计算，叫做计算段，用canCompute属性标识；一种类型是还需要继续分的段，也就是细分段

只有start和end在10000以内才不继续划分段，当canCompute属性为true的时候，sum+=i 可以进行计算；

当canCompute属性为false的时候，进行细分子任务，上述的程序每一次细分的粒度为100，这个属性也可以开放出来进行设置；

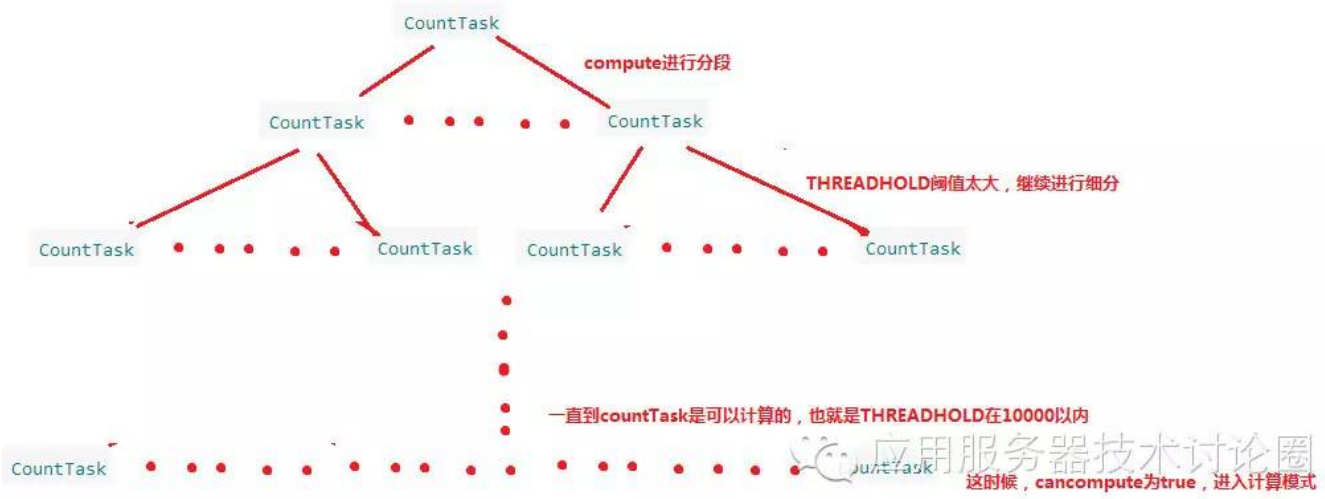
b.细分段中程序的逻辑是，分成100等分，每一份再新建一个RecursiveTask，调用fork方法；

这个fork方法可以暂时理解为将当前的子任务提交给ForkJoinPool来做了，

c.在提交完任务后，调用task的join方法，等待任务结果的返回，如果子任务计算完成了，那么这个join直接返回，

如果没有计算完，这里就阻塞一会，一直到join方法计算完成；

通过上述程序分析，RecursiveTask之所以叫做Recursive的原因，就在于这个程序的递归性，如下图所示



对于ForkJoinTask 的客户端的调用：

```
public static void main(String[]args){
    ForkJoinPool forkJoinPool = new ForkJoinPool();//forkjoin池
    CountTask task = new CountTask(0,20000L);//新建任务
    ForkJoinTask<Long> result = forkJoinPool.submit(task);//提交任务给线程池
    try{
        long res = result.get();//异步的等待结果
        System.out.println("sum="+res);
    }catch(InterruptedException e){
        e.printStackTrace();
    }catch(ExecutionException e){
        e.printStackTrace();
    }
}
```

需要将Task塞入给线程池，然后也是调用get方法进行结果的等待；

因为ForkJoinTask 也是Future，也可以做等待结果，所以也可以调用get方法，这一点既可以作为传入参数，又可以通过其返回结果，和FutureTask类似；

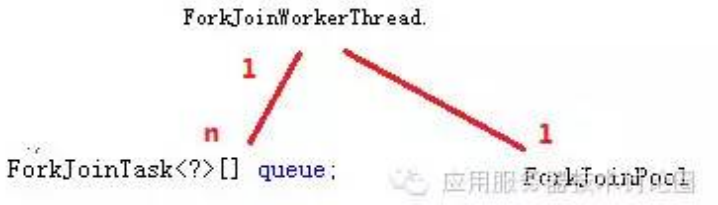
2.ForkJoinWorkerThread

JDK的集中线程池中默认启动的是 Thread，对线程没有进行包装，

而对于ForkJoin池，线程需要完成上述的mapreduce算法，因此对Thread进行了包装

```
public class ForkJoinWorkerThread extends Thread {
```

ForkJoinWorkerThread就是类似于JDK线程池中的Worker，类图如下：



ForkJoinWorkerThread持有ForkJoinPool池的一个引用，

其次需要注意的是，每一个work线程都关联一个ForkJoinTask的队列，这点和JDK的其他线程池共享一个待命的任务队列有很大的不同；

基于上述的程序进行分析：

a. 客户端提交任务（ main函数中 ）

将forkJoinTask首先压入一个forkjoinpool的队列中

```
private void addSubmission(ForkJoinTask<?> t) {
    final ReentrantLock lock = this.submissionLock;
    lock.lock();
    try {
        ForkJoinTask<?>[] q; int s, m;
        if ((q = submissionQueue) != null) {    // ignore if queue removed
            long u = (((s = queueTop) & (m = q.length-1)) << ASHIFT)+ABASE;
            UNSAFE.putOrderedObject(q, u, t);
            queueTop = s + 1;
            if (s - queueBase == m)
                growSubmissionQueue();
        }
    } finally {
        lock.unlock();
    }
    signalWork();
}
```

然后实际调用了forkJoinPool的addWorker方法进行new Thread

```
private void addWorker() {
    Throwable ex = null;
    ForkJoinWorkerThread t = null;
    try {
        t = factory.newThread(this);
    } catch (Throwable e) {
        ex = e;
    }
    if (t == null) {  // null or exceptional factory return
        long c;       // adjust counts
        do {} while (!UNSAFE.compareAndSwapLong
                     (this, ctlOffset, c = ctl,
                      (((c - AC_UNIT) & AC_MASK) |
                       ((c - TC_UNIT) & TC_MASK) |
                       (c & ~(AC_MASK|TC_MASK)))));
        // Propagate exception if originating from an external caller
        if (!tryTerminate(false) && ex != null &&
            !(Thread.currentThread() instanceof ForkJoinWorkerThread))
            UNSAFE.throwException(ex);
    }
    else
        t.start();
```

```
static class DefaultForkJoinWorkerThreadFactory
    implements ForkJoinWorkerThreadFactory {
    public ForkJoinWorkerThread newThread(ForkJoinPool pool) {
        return new ForkJoinWorkerThread(pool);
    }
}
```


这里通过默认的工厂方法，可以看出，new的就是ForkJoinWorkerThread，

之后，线程启动开始做事，会从forkjoinpool中取出刚才塞入的队列，加入自己的ForkJoinTask的Queue中；

然后调用exec方法执行业务逻辑，exec方法在RecursiveTask类中实现为compute，

```
public abstract class RecursiveTask<V> extends ForkJoinTask<V> {
    private static final long serialVersionUID = 5232453952276485270L;

    /**
     * The result of the computation.
     */
    V result;

    /**
     * The main computation performed by this task.
     */
    protected abstract V compute();

    public final V getRawResult() {
        return result;
    }

    protected final void setRawResult(V value) {
        result = value;
    }

    /**
     * Implements execution conventions for RecursiveTask.
     */
    protected final boolean exec() {
        result = compute();
        return true;
    }

}
```


b. ForkJoinWorkerThread 细分提交子任务（ForkJoinTask的compute方法中）

先看fork方法：下载《开发者大全》 下载 (/download/dev.apk)

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


这个方法也相当于客户端的提交任务，实际就是将task压入到当前的ForkJoinWorkerThread 线程中，

主要的逻辑在signalWork方法中：

```
final void pushTask(ForkJoinTask<?> t) {
    ForkJoinTask<?>[] q; int s, m;
    if ((q = queue) != null) {    // ignore if queue removed
        long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;
        UNSAFE.putOrderedObject(q, u, t);
        queueTop = s + 1;         // or use putOrderedInt
        if ((s -= queueBase) <= 2)
            pool.signalWork();
        else if (s == m)
            growQueue();
    }
}

/**
 * Wakes up or creates a worker.
 */
final void signalWork() {
    /*
     * The while condition is true if: (there is are too few total
     * workers OR there is at least one waiter) AND (there are too
     * few active workers OR the pool is terminating).  The value
     * of e distinguishes the remaining cases: zero (no waiters)
     * for create, negative if terminating (in which case do
     * nothing), else release a waiter. The secondary checks for
     * release (non-null array etc) can fail if the pool begins
     * terminating after the test, and don't impose any added cost
     * because JVMs must perform null and bounds checks anyway.
     */
    long c; int e, u;
    while ((((e = (int)(c = ctl)) | (u = (int)(c >>> 32))) &
            (INT_SIGN|SHORT_SIGN)) == (INT_SIGN|SHORT_SIGN) && e >= 0) {
        if (e > 0) {                         // release a waiting worker
            int i; ForkJoinWorkerThread w; ForkJoinWorkerThread[] ws;
            if ((ws = workers) == null ||
                (i = ~e & SMASK) >= ws.length ||
                (w = ws[i]) == null)
                break;
            long nc = (((long)(w.nextWait & E_MASK)) |
                       ((long)(u + UAC_UNIT) << 32));
            if (w.eventCount == e &&
                UNSAFE.compareAndSwapLong(this, ctlOffset, c, nc)) {
                w.eventCount = (e + EC_UNIT) & E_MASK;
                if (w.parked)
                    UNSAFE.unpark(w);
                break;
            }
        }
        else if (UNSAFE.compareAndSwapLong
                 (this, ctlOffset, c,
                  (long)(((u + UTC_UNIT) & UTC_MASK) |
                         ((u + UAC_UNIT) & UAC_MASK)) << 32)) {
            addWorker();
            break;
        }
    }
```

如果ForkJoinPool中目前有足够的ForkJoinWorkeThread线程，那么唤醒一个使用

如果池中目前没有，那么整体流程和main函数的新建线程池的流程是一样的

因为这个时刻，客户端已经启动一段了，因此pool中是存在一些已经工作的县城了，因此分为两种情况，如果有空闲的worker直接干活，如果没有再创建；

上述程序中，ForkJoinWorkerThread 的个数是基于当前负载算法和池的大小配置的，在pool中有初始化对应的大小，也就是上述判断的第一部分逻辑

而通常的情况是1个ForkJoinWorkerThread对应n个ForkJoinTask，处理完手头的任务后，回到pool中，然后当pool的signalWork方法被调用时，再接活工作；

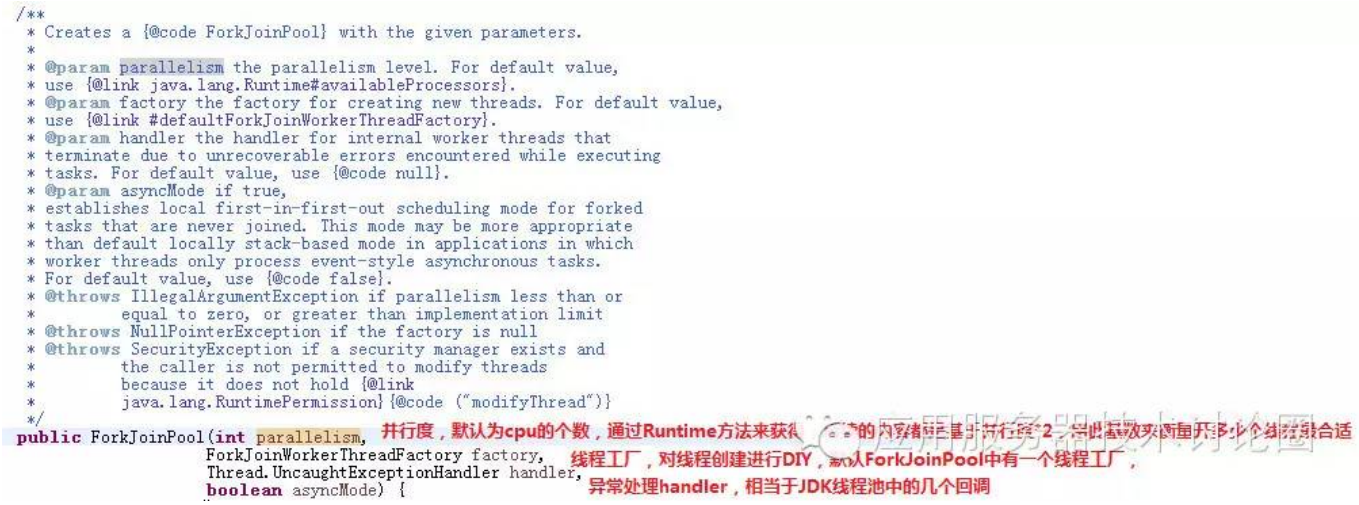
整体思路实质和JDK的线程池类似；

3.ForkJoinPool

ForkJoinPool对应的线程池，它和JDK的几个线程池不一样，没有使用ThreadPoolExecutor作为底层实现，而是自己实现；

它继承的接口也是AbstractExecutorService抽象的层次；

构造方法传入的参数也和JDK的线程池类似，



下一节再细致分析ForkJoinPool池的任务窃取算法！

分享：

阅读 203 0

应用服务器技术讨论圈 更多文章

东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)

玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)

金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)

GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)

Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢

- 分享丨海量数据处理分析的16条经验总结 (/html/303/201601/401598815/1.html)
- Spark SQL 究竟是何方神圣? (/html/295/201606/2651987669/1.html)
- C++11读书笔记-8（多线程使用简介）(一) (/html/461/201608/2650715903/1.html)
- 麻木的IT公民：你有极端的行为表现吗？（连载） (/html/49/201504/206213226/1.html)
- 心动游戏的云秘笈 (/html/362/201503/203418496/1.html)