ForkJoinPool源码分析之二

2015-10-10 14:14 feiying 1 阅读 159

1.fork/join思路

ForkJoinPool除了完成基本的线程池的任务Thread计算的功能，还可以完成map，reduce的思想；

a.fork

首先，将ForkJoinTask压入工作线程 ForkJoinWorkerThread的Queue中

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

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();
    }
}
```

CAS操作压入队列

当队列中的空位不够，进行row增长

当每一个线程对应的TaskQueue的空位不够的话，需要重新对queue的array进行扩容；

然后，调用pool的signalWork方法，这个方法主要是唤醒ForkJoinPool中的空闲的ForkJoinWorkerThread任务，使其干活：

```
/**
 * 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;
        }
    }
}
```

如果整个pool中的ForkJoinThread任务太少了，但是数量是够的，其原因就在于有一些任务是没有活，这里就需要对其进行唤醒

如果一看池中的任务根本就不多，那么这里毫无疑问的，直接进行addWorker操作

addWorker方法和JDK其他线程池的方法几乎类似，也是调用ThreadFactory创建线程：

```
/**
 * Tries to create and start a worker; minimally rolls back counts
 * on failure.
 */
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();
}
```

调用线程工厂创建线程

这里使用的是ForkJoin默认的线程工厂

线程创建完毕之后，直接s...

到这里fork的工作就完成了，整体来说总结一下，ForkJoinWorkerThread当前任务线程pushTask，然后关注线程池的其他线程够不够，如果不够的话，addWorker，如果

当前这里抛出一个疑问：

为嘛当前线程处理任务，我要关注于其他线程够不够呢，我处理完自己的事不就得了？

这个问题在下面的任务窃取算法中给出答案；

b.join

join方法和Thread的join方法差不多，也是等待最后的结果；

```
public final V join() {
    if (doJoin() != NORMAL)
        return reportResult();
    else
        return getRawResult();
}

/**
 * Primary mechanics for join, get, quietlyJoin.
 * @return status upon completion
 */
private int doJoin() {
    Thread t; ForkJoinWorkerThread w; int s; boolean completed;
    if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
        if ((s = status) < 0)
            return s;
        if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {
            try {
                completed = exec();
            } catch (Throwable rex) {
                return setExceptionalCompletion(rex);
            }
            if (completed)
                return setCompletion(NORMAL);
        }
        return w.joinTask(this);
    }
    else
        return externalAwaitDone();
}
```

当Task的status小于0的时候，说明结果已经出来了(或者是异常)，直接可以返回结果

从当前任务线程中弹出任务，调用exec方法进行执行；

如果发现task弹不出来，说明这个task正在被其他线程“帮助执行”，

这里的任务就是等待其他线程的结果；

对于join方法主要就是通过一个ForkJoinTask的status属性进行判断，任务是否完成，如果完成，那么直接返回结果，

如果没有完成，那么立刻从ForkJoinWorkerThread弹出ForkJoinTask，进行执行；

这里需要注意的是，ForkJoinTask可能被其他的ForkJoinWorkerThread所执行，这里也就是任务窃取的思路；

2.任务窃取算法

在fork，join方法中，多次提到了任务窃取算法；

首先，通过前面的源码分析，每一个ForkJoinWorkerThread都对应自己的一个local Queue队列，也就是下图：

注意上述的队列是双端队列，对于当前的，它会按照FIFO，也就是先进先出的原则，也就是从队尾取任务；

而因为是双端的，刚加入的队头也可以取任务（这也就是证明了，为什么上述分析的join方法源码，会判断该任务是否被其他线程优先取走）；

这种好处是显而易见的，可以减少并发的冲突，因为在大并发的情况下，queue的长度很长，窃取任务几乎不会遇到阻塞，采用CAS无锁算法会轻而易举的窃取成功；

如上图实现，S2中的任务不是很多，在某一个时刻已经完成了，而这个时刻S1的任务非常的多，

如果按照JDK其它的线程池中的实现，那么S2可能就会占用时间片，即使主动出让时间片，S2线程切换下来什么也不干，代价也是不小，

而ForkJoinPool不会浪费S2获取时间片的时间，S2线程从S1的queue的队头去拿task任务；

从源码分析的角度也可以看到这个算法清晰的实现，每一个ForkJoinWorkerThread的run方法会调用pool的work方法：

上述的swept的属性是从当前的状态ctl中获取的，如果它为false，说明当前ForkJoinWorkerThread没啥事，那么就通过scan去其他线程那里找找活

这个scan方法就是窃取算法的核心实现，

它会首先设置seed，随机找一个pool中的线程，如果有活，那么就偷窃任务进行执行，进入scanGuard模式；

如果真找不到活干，那么自己再扫扫自己的queue，看是否来了新活；

3.状态属性一次CAS读取

从上述的源码分解中，看到大量的关于pool，ForkJoinWorkerThread的状态的判断，

这块和JDK的线程池是同样的，减少并发的对状态的读写操作，与其使用AtomicRefernce包装一个原子对象，不如使用位的操作，

这种好处就在于long的基础数据类型是volatile语言机制所支持，如果ctl操作能编译为1条汇编指令的话，volatile效率比较高，

如下述的几个字段，都在ctl中：

AC：活动ForkJoinWorkerThread的数目

TC：总共ForkJoinWorkerThread数目（包括活动和不活动的）

ST：线程池是否关闭，=========》可以看到这里仅仅使用一个字节就OK了；

… …

但是，对于ctl的操作显然就是缺点，采用位操作，代码难懂，难以维护，例如下面的addWorker中的状态判断

总结:

ForkJoinPool同样实现了Executor接口，也和其他的JDK线程池类似

但是，ForkJoinPool实现了任务窃取算法，推出了fork，join等接口，

实现了mapreduce的分治思想，更大程度的复用了多核cpu；

分享：

阅读 159 1

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

- Web开发者应该有哪些必备的技能？ (/html/434/201605/2651560097/1.html)
- 淘宝首页性能优化实践 (/html/463/201606/2650838999/1.html)
- 如何在HTML中使用图标字体 - icon font? (/html/450/201311/10000033/1.html)
- 百花齐放——Xcode 6 GM、6.1、Swift、1.0版齐出！ (/html/334/201409/200915198/1.html)
- 第一部分::任务5:总结作业-记事本 (/html/150/201608/2650392265/1.html)