ForkJoinPool底层原理分析

ThreadPoolExecutor的执行流程

ForkJoinPool的执行流程

JDK19虚拟线程

ThreadPoolExecutor的执行流程

首先在构造一个ThreadPoolExecutor时，并不会创建线程，而是等到线程池接收到任务时才会创建线程。

当我们向ThreadPoolExecutor中submit任务时，ThreadPoolExecutor内部会判断当前线程池中的线程个数是否超过了corePoolSize，如果没有超过，则会创建线程，并且该线程的第一个任务就是当前submit的任务。

这里注意一点，就是submit任务时，线程池中有空闲的线程，只要线程总数没有超过corePoolSize，那就会创建新的线程，也就是说，只有在线程池中的线程个数达到了corePoolSize时，才会尝试将任务添加到队列中去。

如果队列满了，则看当前线程池中的线程个数是否超过了maximumPoolSize，如果没有超过，则创建新的线程，如果超过了则执行拒绝策略。

这个过程相信大家是比较熟悉的，也是最常用的，核心思想可以抽象为一个任务只会让一个线程来负责执行，而就是这一点，在某些时候可能是不合适的。

比如现在有一个任务，需要计算1-10000累加的和，比如这个任务的执行代码为：

```java
threadPoolExecutor.submit(new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {

        int sum = 0;

        for (int i = 1; i <= 10000; i++) {
            sum += i;
        }

        return sum;
    }
});
```

假如此时ThreadPoolExecutor有10个空闲线程，那是也只能有一个线程来执行这个任务，尽管原理上可以把这个任务进行拆分，然后由多个线程来同时执行，从而提高效率。

而ForkJoinPool这个线程池就是来处理这种情况的。

ForkJoinPool的执行流程

ForkJoinPool和ThreadPoolExecutor的最大区别就在于，ForkJoinPool在处理任务时，会触发任务拆分的逻辑，并且将拆分出来的子任务继续交给ForkJoinPool中的线程去处理，这样就可能充分利用ForkJoinPool中的线程，能更快的去执行一个大任务。

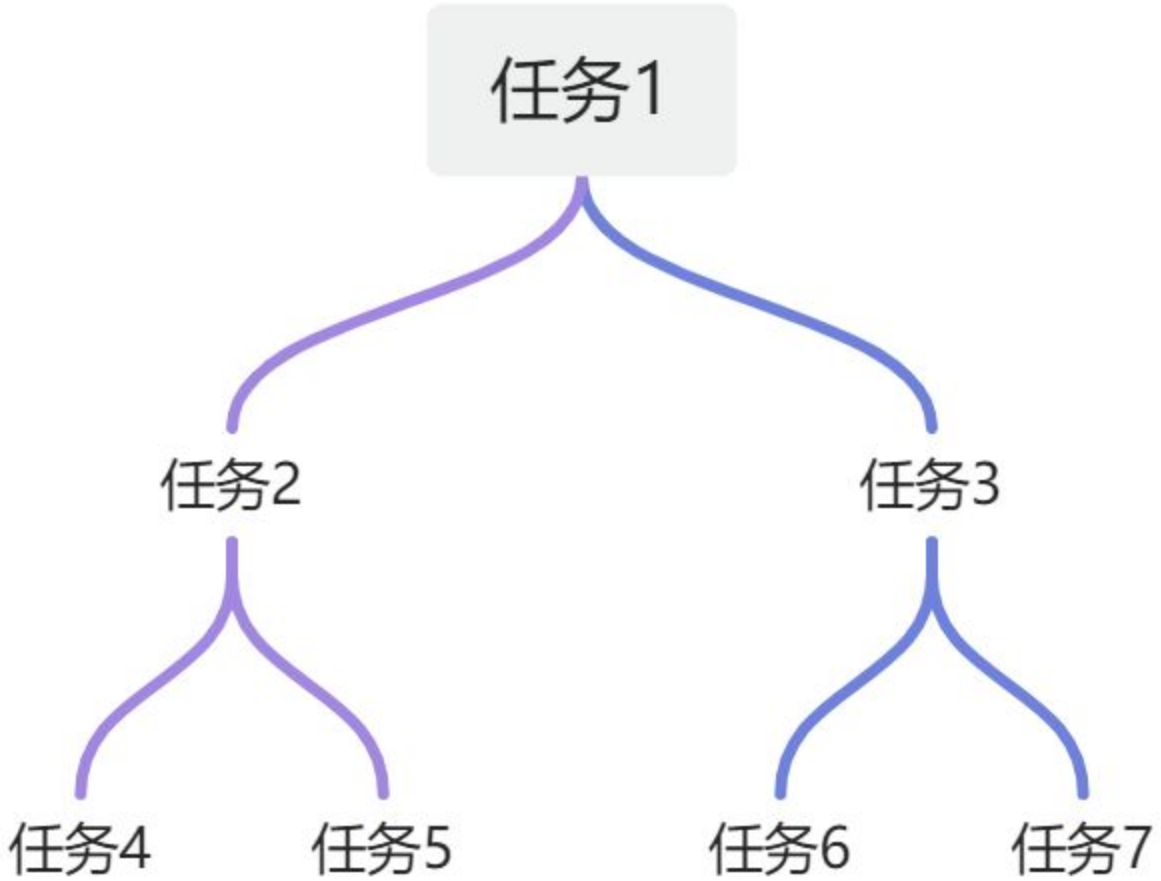
为此，ForkJoinPool在执行一个任务时，会对任务进行拆分，任务拆分的逻辑由程序员来控制，比如：

Java | 复制代码

```java
public class SumTask extends RecursiveTask<Long> {

    private final int begin;
    private final int end;

    public SumTask(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }

    @Override
    protected Long compute() {
        long sum = 0;
        if (end - begin < 100) {
            for (int i = begin; i <= end; i++) {
                sum += i;
            }
        } else {
            // 拆分逻辑

            int middle = (end + begin) / 2;

            SumTask subtask1 = new SumTask(begin, middle);
            SumTask subtask2 = new SumTask(middle + 1, end);

            subtask1.fork();
            subtask2.fork();

            // 等到子任务做完
            long sum1 = subtask1.join();
            long sum2 = subtask2.join();

            sum = sum1 + sum2;
        }
        return sum;
    }
}
```

比如SumTask表示一个数字累加任务，可以指定begin、end，而compute()方法就是在执行任务，我们可以在compute()方法中去做任务拆分的逻辑。

比如上述代码就会判断begin和end这个范围是否超过100，如果小于100则不拆分，直接进行累加运算，如果大于100，则会把这个任务拆分成两个小任务并进行fork，并且当前任务要拿到结果，就需要join等待这两个小任务执行完。



相当于，任务1的执行需要等待任务2和任务3执行完，任务2和任务3又分别需要等待它们各自的子任务执行完成。

那如果ForkJoinPool执行这个任务1，共需要多少个线程才能完成呢?

1. 线程1执行任务1，拆分出任务2和任务3，然后阻塞等待任务2和任务3执行完成
2. 线程2执行任务2，拆分出任务4和任务5，然后阻塞等待任务4和任务5执行完成
3. 线程3执行任务3，拆分出任务6和任务7，然后阻塞等待任务6和任务7执行完成
4. 线程4执行任务4
5. 线程5执行任务5

6. 线程6执行任务6
7. 线程7执行任务7

按这种思路就需要7个线程才能完成任务1的执行，其中3个线程负责拆分任务并阻塞等待子任务的结果，4个线程负责执行最小、不用拆分的任务。

那这是最合适的方案吗，当然不是，如果这个任务1，最终拆分出100个最小的子任务，难道真的去开辟100多个线程来执行吗？ForkJoinPool当然不会这么做。

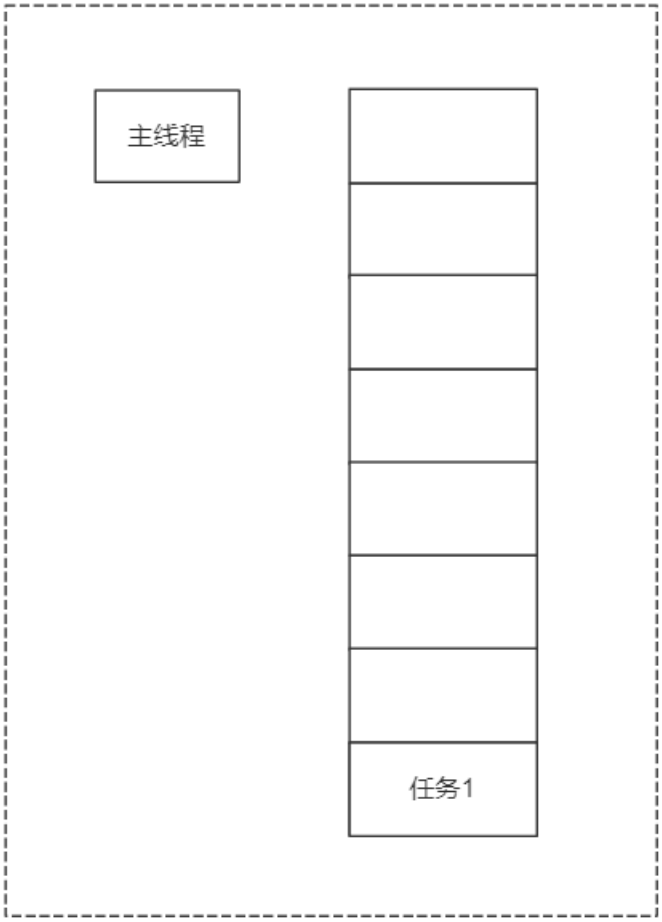
在创建ForkJoinPool时，可以设置parallelism参数，它就是用来指定ForkJoinPool中最大线程数的，默认为：Runtime.getRuntime().availableProcessors()。

那假如现在parallelism为4，也就是ForkJoinPool中最多只会有4个线程，那该如何来执行上面的那些任务呢?

很明显，如果只有4个线程，那么每个线程就不能只执行一个任务了，所以这就需要用到队列了。

比如每个线程都对应一个任务队列，线程可以从队列中获取任务来执行。

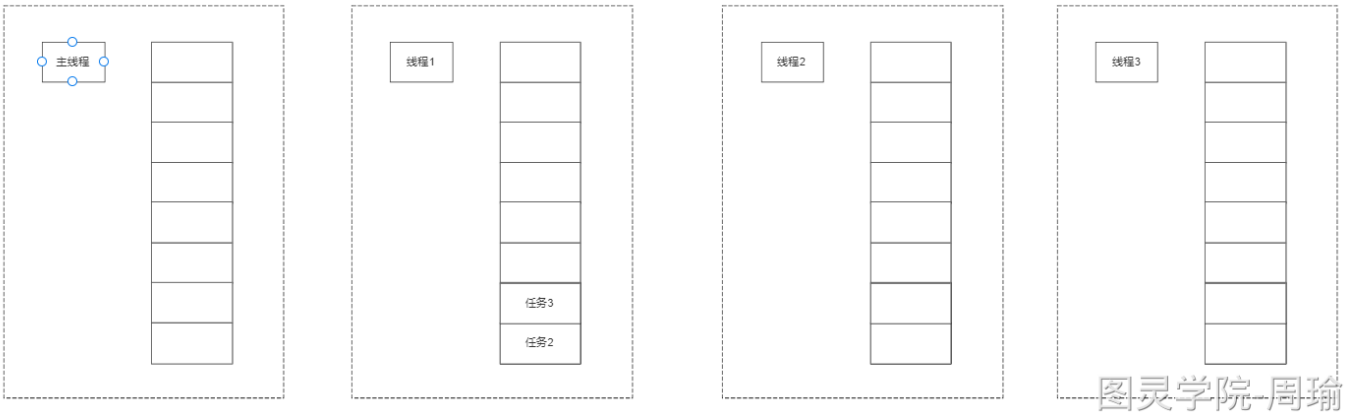
比如，当主线程提交任务1给ForkJoinPool时，ForkJoinPool就会针对主线程开辟一个队列用来存任务1



那谁来执行任务1呢？是主线程呢？还是新开一个线程呢？

如果是主线程执行任务1，那是不是主线程将需要阻塞等待？所以答案是新开一个线程来执行任务1，主线程将不会阻塞，可以继续执行其他（和ThreadPoolExecutor一样，主线程只提交任务，不执行任务）。

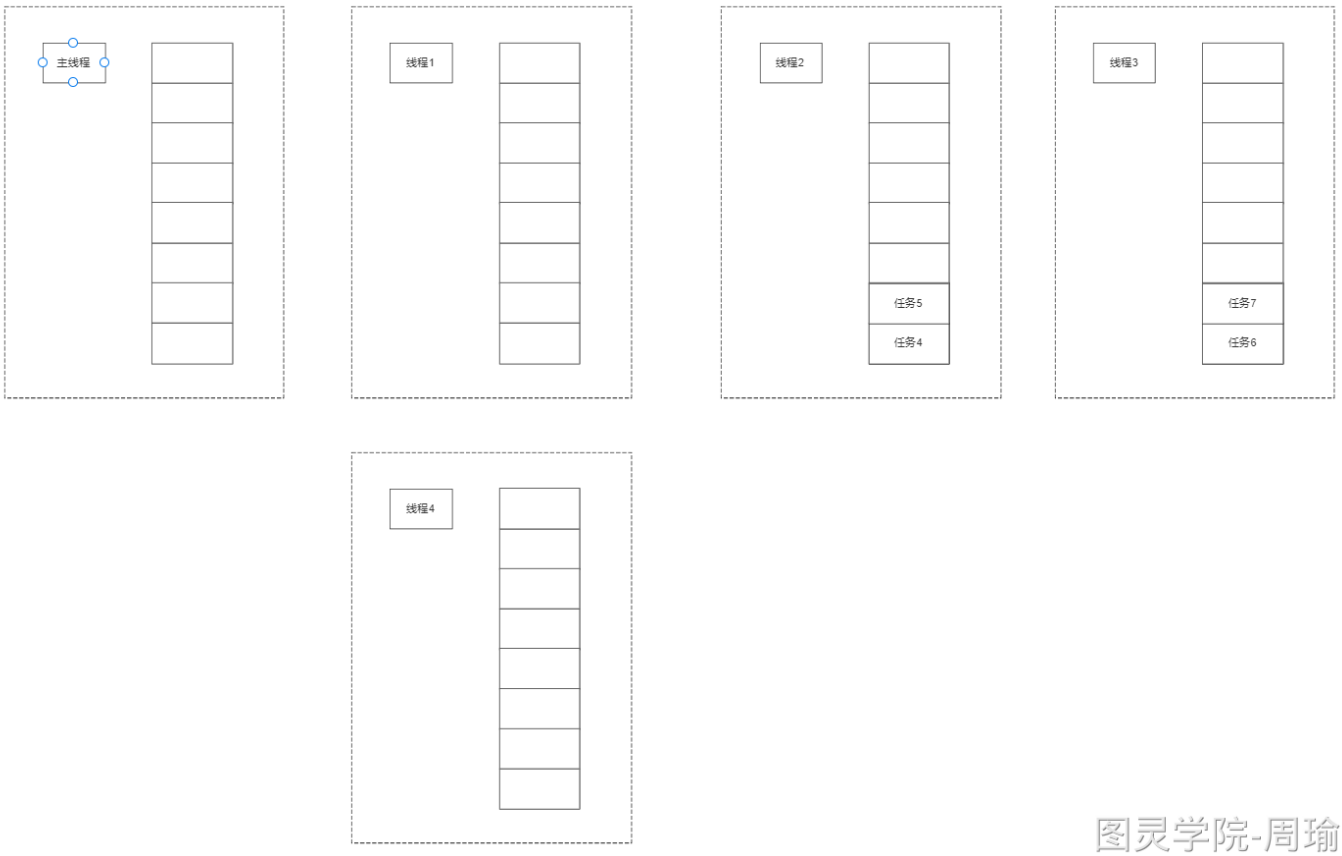
所以新开一个线程1，而线程1将会拉取出任务1，并执行任务1，并拆分出两个任务，每fork一次，就会将子任务将添加到线程1对应的队列中，并且会尝试开启另外的线程。



线程1完成了任务拆分后，就会join住（阻塞住），等待子任务的返回结果。

新开辟的线程2会负责执行任务2，新开辟的线程3会负责执行任务3。

线程2执行任务2时，又需要拆分任务，并且也会尝试去开辟线程，但是受到parallelism=4的限制，只能再开辟一个线程4了



由于线程4是执行任务4的过程中开辟出来的，所以线程4会去执行任务4，此时任务4不用拆分了，线程4执行完任务4之后，自然可以执行任务5，然后执行任务6，任务7。

那么线程4在执行任务4的过程中，其他3个线程在干嘛呢？阻塞等子任务的结果，那这三个线程不能帮忙来执行任务5、6、7吗？当然可以。

比如线程3虽然在阻塞等待任务6、7的结果，当然现在没有多余线程能来执行，所以它自己就可以执行任务6，7，这是执行自己任务队列中的任务。

或者线程1，它自己队列没有任务需要执行，它也是在阻塞等待子任务的执行结果，那它也可以帮忙执行其他任务队列中的任务，比如执行线程2中的任务2（当然任务2，线程2也可以字节执行），这种就是窃取任务执行。

所以一个任务队列中的任务，线程自己和其他线程都可能会从队列中来获取任务进行执行，所以就不要都从一端来获取任务，所以队列可以是一个双端队列，窃取其他线程队列中的任务时，就采用FIFO，相当于从队头获取任务，而自己线程从看asyncMode的配置，默认是false，表示LIFO，表示从队尾获取任务，asyncMode为true，则也是FIFO，相当于所有线程都会从队头获取任务，所以叫同步模型。

那么，一个线程在阻塞的过程中会去帮忙执行其他队列中的任务，那它如何知道自己的子任务是否执行完了呢?

很简单，利用循环来达到阻塞的效果，比如源码中是这么体现的：

Java | 复制代码

```java
final int helpComplete(WorkQueue w, CountedCompleter<?> task,
                       int maxTasks) {
        WorkQueue[] ws; int s = 0, m;
        if ((ws = workQueues) != null && (m = ws.length - 1) >= 0 &&
            task != null && w != null) {
            int mode = w.config;                // for popCC
            int r = w.hint ^ w.top;             // arbitrary seed for origin
            int origin = r & m;                 // first queue to scan
            int h = 1;                          // 1:ran, >1:contended, <0:hash
            // 死循环
            for (int k = origin, oldSum = 0, checkSum = 0;;) {
                CountedCompleter<?> p; WorkQueue q;
                // 判断任务是否完成，完成了则break
                if ((s = task.status) < 0)
                    break;
                if (h == 1 && (p = w.popCC(task, mode)) != null) {
                    p.doExec();                 // run local task
                    if (maxTasks != 0 && --maxTasks == 0)
                        break;
                    origin = k;                 // reset
                    oldSum = checkSum = 0;
                }
                else {                          // poll other queues
                    if ((q = ws[k]) == null)
                        h = 0;
                        // 从队列中poll并执行任务
                    else if ((h = q.pollAndExecCC(task)) < 0)
                        checkSum += h;
                    if (h > 0) {
                        if (h == 1 && maxTasks != 0 && --maxTasks == 0)
                            break;
                        r ^= r << 13; r ^= r >>> 17; r ^= r << 5; // xorshift
                        origin = k = r & m;     // move and restart
                        oldSum = checkSum = 0;
                    }
                    else if ((k = (k + 1) & m) == origin) {
                        if (oldSum == (oldSum = checkSum))
                            break;
                        checkSum = 0;
                    }
                }
            }
```

```
43          }
44          return s;
45      }
```

上述源码可以看出，是一个自旋操作，每次循环会检查一下当前任务是否完成，如果没有完成就会去获取队列中的任务执行，执行完之后又会进行一次循环并检查当前任务是否完成，完成了就break。

所以，这就是ForkJoinPool中的核心逻辑，总结一下：

1. 可以通过parallelism来设置ForkJoinPool中的线程个数
2. ForkJoinPool中的每个线程都对应的一个任务队列，该队列中存的任务是线程自己负责的任务，这些要么自己执行，要么被别的线程执行
3. 每个线程在执行任务时都可能拆分出其他子任务，这些子任务会加入到自己的任务队列中
4. 每个现在在执行任务时，如果拆分出了子任务，那么则需要等待这些子任务的完成，当前执行的任务才能完成，只不过在等待的过程中可以去执行其他任务队列中的任务，当前等待的任务一旦执行完成task.status则会发现改变，这样当前线程就拿到结果了，可以继续执行后续代码了，最终把当前任务执行完。

综合来看，一个ForkJoinPool中，一般会存在多个任务队列，多个线程，线程会从这写队列中获取任务来执行，并且可能会产生新的任务，任务有一个status标记，表示任务是否执行完成。

JDK19虚拟线程

JDK19中的虚拟线程就是业界的协程。

因为协程是用户态的，线程是操作系统内核态的，所以协程仍然基于的是线程，一个线程可以承载多个协程，但如果所有协程都只基于一个线程，那样效率肯定会不高，所以JDK19中协程会基于ForkJoinPool线程池，利用多个线程来支持协程的运行，并且利用ForkJoinPool，而不是普通的ThreadPoolExecutor，可以支持大任务的拆分。

JDK19中的协程底层是基于ForkJoinPool的，相当于，我们在利用协程执行Runnable时，底层会把Runnable提交到一个ForkJoinPool中去执行，我们可以通过：

- -Djdk.virtualThreadScheduler.parallelism=1
- -Djdk.virtualThreadScheduler.maxPoolSize=1

这两个参数来设置ForkJoinPool的核心线程数和最大线程数：

- parallelism默认为Runtime.getRuntime().availableProcessors()
- maxPoolSize默认为256

ForkJoinPool中的线程在执行任务过程中，一旦线程阻塞了，比如sleep、lock、io操作时，那么这个线程就会去执行ForkJoinPool中的其他任务，从而可以做到一个线程在执行过程中，也能并发的执行多个任务，达到协程并发执行任务的效果。