深入浅出 Java Concurrency (33): 线程池 part 6 线程池的实现及原理 (1)

● 线程池数据结构与线程构造方法

由于已经看到了 ThreadPoolExecutor 的源码，因此很容易就看到了 ThreadPoolExecutor 线程池的数据结构。图 1 描述了这种数据结构。

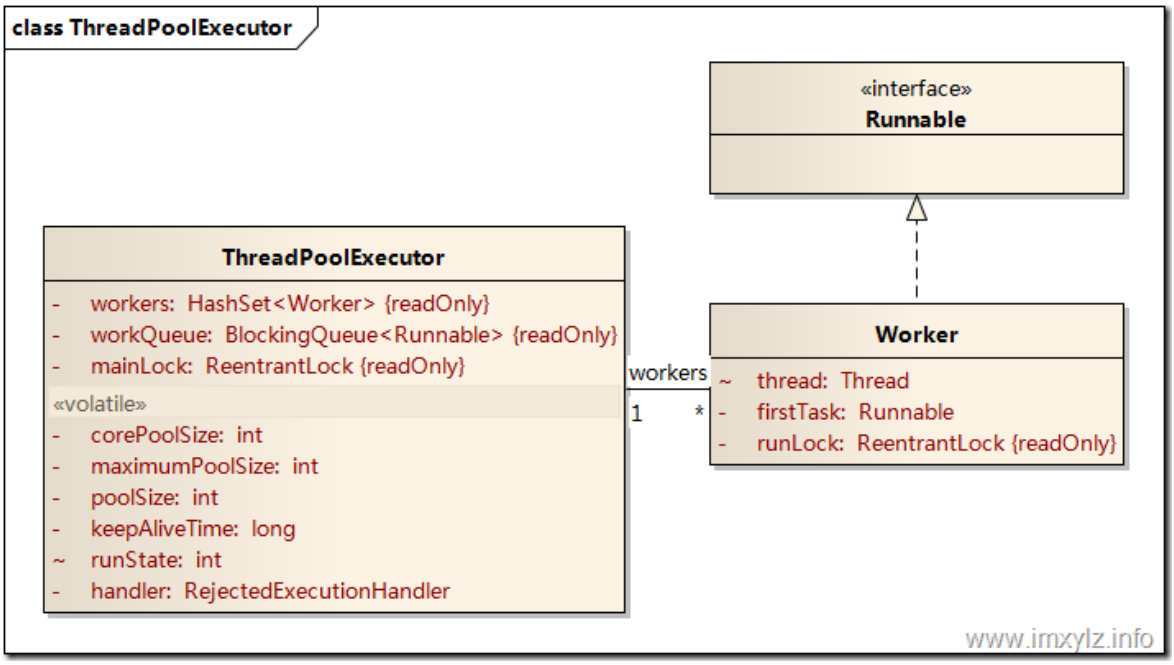


图 1 ThreadPoolExecutor 数据结构

其实，即使没有上述图形描述 ThreadPoolExecutor 的数据结构，我们根据线程池的要求也很能够猜测出其数据结构出来。

- 线程池需要支持多个线程并发执行，因此有一个线程集合 Collection<Thread>来执行线程任务；
- 涉及任务的异步执行，因此需要有一个集合来缓存任务队列 Collection<Runnable>；
- 很显然在多个线程之间协调多个任务，那么就需要一个线程安全的任务集合，同时还需要支持阻塞、超时操作，那么 BlockingQueue 是必不可少的；
- 既然是线程池，出发点就是提高系统性能同时降低资源消耗，那么线程池的大小就有限制，因此需要有一个核心线程池大小（线程个数）和一个最大线程池大小（线程个数），有一个计数用来描述当前线程池大小；
- 如果是有限的线程池大小，那么长时间不使用的线程资源就应该销毁掉，这样就需要一个线程空闲时间的计数来描述线程何时被销毁；
- 前面描述过线程池也是有生命周期的，因此需要有一个状态来描述线程池当前的运行状态；

- 线程池的任务队列如果有边界，那么就需要有一个任务拒绝策略来处理过多的任务，同时在线程池的销毁阶段也需要有一个任务拒绝策略来处理新加入的任务；
- 上面种的线程池大小、线程空闲实际那、线程池运行状态等等状态改变都不是线程安全的，因此需要有一个全局的锁（mainLock）来协调这些竞争资源；
- 除了以上数据结构以外，ThreadPoolExecutor 还有一些状态用来描述线程池的运行计数，例如线程池运行的任务数、曾经达到的最大线程数，主要用于调试和性能分析。

对于 ThreadPoolExecutor 而言，一个线程就是一个 Worker 对象，它与一个线程绑定，当 Worker 执行完毕就是线程执行完毕，这个在后面详细讨论线程池中线程的运行方式。

既然是线程池，那么就首先研究下线程的构造方法。

```
public interface ThreadFactory {
    Thread newThread(Runnable r);
}
```

ThreadPoolExecutor 使用一个线程工厂来构造线程。线程池都是提交一个任务 Runnable，然后在某一个线程 Thread 中执行，ThreadFactory 负责如何创建一个新线程。

在 J.U.C 中有一个通用的线程工厂 java.util.concurrent.Executors.DefaultThreadFactory，它的构造方式如下：

```
static class DefaultThreadFactory implements ThreadFactory {
    static final AtomicInteger poolNumber = new AtomicInteger(1);
    final ThreadGroup group;
    final AtomicInteger threadNumber = new AtomicInteger(1);
    final String namePrefix;
    DefaultThreadFactory() {
        SecurityManager s = System.getSecurityManager();
        group = (s != null)? s.getThreadGroup() :
                             Thread.currentThread().getThreadGroup();
        namePrefix = "pool-" +
                      poolNumber.getAndIncrement() +
                     "-thread-";
    }
    public Thread newThread(Runnable r) {
        Thread t = new Thread(group, r,
                              namePrefix + threadNumber.getAndIncrement(),
                              0);
        if (t.isDaemon())
            t.setDaemon(false);
        if (t.getPriority() != Thread.NORM_PRIORITY)
            t.setPriority(Thread.NORM_PRIORITY);
        return t;
```

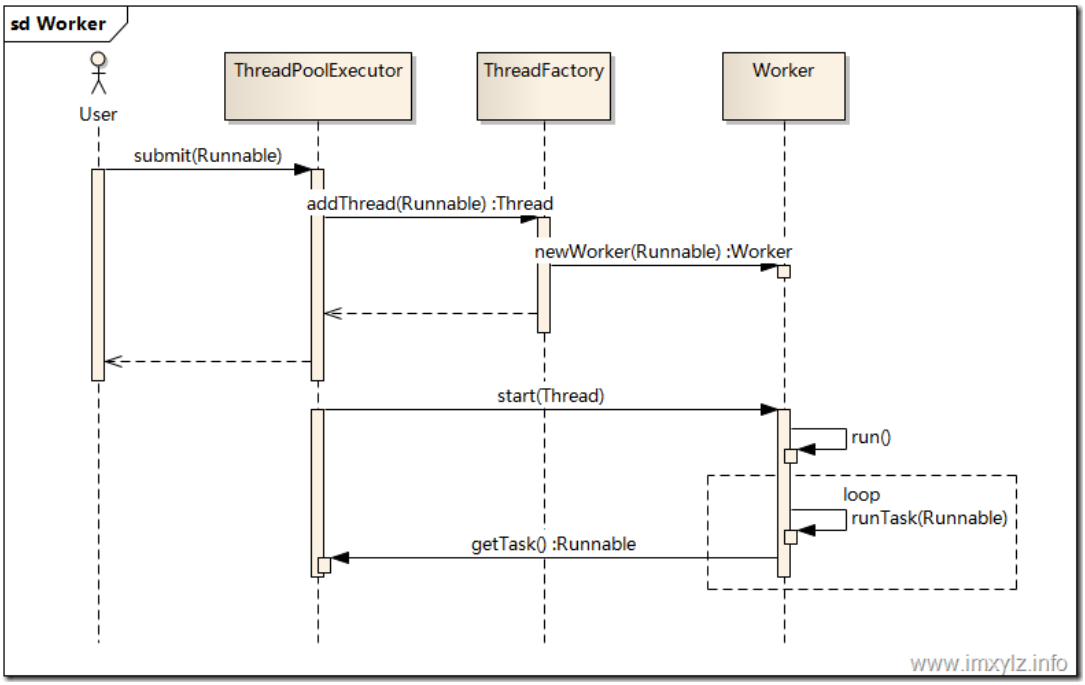
```
    }
}
```

在这个线程工厂中，同一个线程池的所有线程属于同一个线程组，也就是创建线程池的那个线程组，同时线程池的名称都是"pool-<poolNum>-thread-<threadNum>"，其中 poolNum 是线程池的数量序号，threadNum 是此线程池中的线程数量序号。这样如果使用 jstack 的话很容易就看到了系统中线程池的数量和线程池中线程的数量。另外对于线程池中的所有线程默认都转换为非后台线程，这样主线程退出时不会直接退出 JVM，而是等待线程池结束。还有一点就是默认将线程池中的所有线程都调为同一个级别，这样在操作系统角度来看所有系统都是公平的，不会导致竞争堆积。

●线程池中线程生命周期

一个线程 Worker 被构造出来以后就开始处于运行状态。以下是一个线程执行的简版逻辑。

```
private final class Worker implements Runnable {
    private final ReentrantLock runLock = new ReentrantLock();
    private Runnable firstTask;
    Thread thread;
    Worker(Runnable firstTask) {
        this.firstTask = firstTask;
    }
    private void runTask(Runnable task) {
        final ReentrantLock runLock = this.runLock;
        runLock.lock();
        try {
            task.run();
        } finally {
            runLock.unlock();
        }
    }
    public void run() {
        try {
            Runnable task = firstTask;
            firstTask = null;
            while (task != null || (task = getTask()) != null) {
                runTask(task);
                task = null;
            }
        } finally {
            workerDone(this);
        }
    }
}
```



当提交一个任务时，如果需要创建一个线程（何时需要在下一节中探讨）时，就调用线程工厂创建一个线程，同时将线程绑定到 Worker 工作队列中。需要说明的是，Worker 队列构造的时候带着一个任务 Runnable，因此 Worker 创建时总是绑定着一个待执行任务。换句话说，创建线程的前提是有必要创建线程（任务数已经超出了线程或者强制创建新的线程，至于为何强制创建新的线程后面章节会具体分析），不会无缘无故创建一堆空闲线程等着任务。这是节省资源的一种方式。

一旦线程池启动线程后（调用线程 run()）方法，那么线程工作队列 Worker 就从第 1 个任务开始执行（这时候发现构造 Worker 时传递一个任务的好处了），一旦第 1 个任务执行完毕，就从线程池的任务队列中取出下一个任务进行执行。循环如此，直到线程池被关闭或者任务抛出了一个 RuntimeException。

由此可见，线程池的基本原理其实也很简单，无非预先启动一些线程，线程进入死循环状态，每次从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已。如此反复。

其实，线程池原理看起来简单，但是复杂的是各种策略，例如何时该启动一个线程，何时该终止、挂起、唤醒一个线程，任务队列的阻塞与超时，线程池的生命周期以及任务拒绝策略等等。下一节将研究这些策略问题。

上一篇：线程池 part 5 周期性任务调度　　**目　录**　　**下一篇**：线程池 part 6 线程池的实现及原理（2）