

20 其实不用造轮子—Executor框架详解

更新时间：2019-11-05 10:29:05



“人的一生可能燃烧也可能腐朽，我不能腐朽，我愿意燃烧起来！”

——奥斯特洛夫斯基”

上一节我们动手实现了一个非常简单的线程池。其实 JDK 已经为我们准备了功能丰富的线程池工具。本章我们就来学习一下 JDK 中的线程池—Executor 框架。

1、Executor 框架的使用

我们首先来看看 Executor 框架是如何实用的。看如下代码：

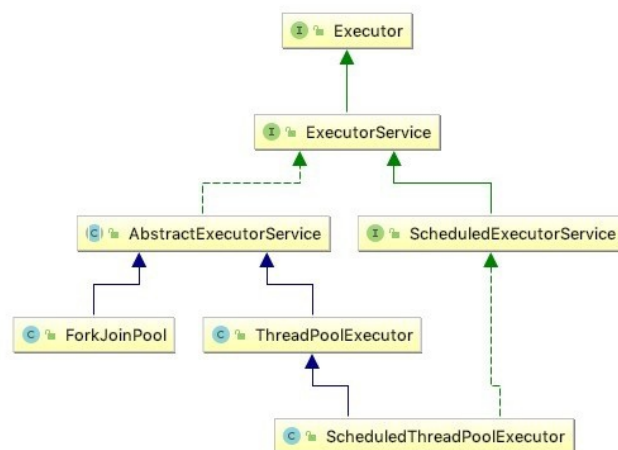
```
public class Client {  
    public static Executor executor = Executors.newFixedThreadPool(10);  
  
    public static void main(String[] args) {  
        Stream.iterate(1, item -> item + 1).limit(20).forEach(item -> {  
            executor.execute(() -> {  
                System.out.println(Thread.currentThread().getName() + " hello!");  
            });  
        });  
    }  
}
```

可以看到在使用上和我们自己实现的线程池几乎一模一样。只不过在声明 Executor 的时候，没有直接 new 对象。而是通过Executors的静态方法 newFixedThreadPool 来创建 Executor。

而执行任务的方式则是和我们自己实现的一模一样。都是调用 executor 方法，传入 Runnable 接口的实现，也就是运行的逻辑。那么它的内部实现是否也和我们实现的一样呢？先别急，我们一点点展开来分析。

2、Executor 框架设计简介

我们先看下Executor框架的继承关系：



1、Executor

可以看到最顶层是 **Executor** 的接口。这个接口很简单，只有一个 **execute** 方法。此接口的目的是为了把任务提交和任务执行解耦。

2、ExecutorService

这还是一个接口，继承自 **Executor**，它扩展了 **Executor** 接口，定义了更多线程池相关的操作。

3、AbstractExecutorService

提供了 **ExecutorService** 的部分默认实现。

4、ThreadPoolExecutor

实际上我们使用的线程池的实现是 **ThreadPoolExecutor**。它实现了线程池工作的完整机制。也是我们接下来分析的重点对象。

5、ForkJoinPool

实现 Fork/Join 模式的线程池，后面会有小节专门讲解。本节不做深入分析。

6、ScheduledExecutorService

这个接口扩展了 **ExecutorService**，定义个延迟执行和周期性执行任务的方法。

7、ScheduledThreadPoolExecutor

此接口则是在继承 **ThreadPoolExecutor** 的基础上实现 **ScheduledExecutorService** 接口，提供定时和周期执行任务的特性。

Executors

Executor 框架还提供 Executors 对象。注意看这个对象比 Executor 接口后面多了个 s，要区分开，不要搞混。Executors 是一个工厂及工具类。提供了例如 newFixedThreadPool(10) 的方法，来创建各种不同的 Executor。

3、Executor 框架源码分析

Executor 设计的类和实现比较多。本节对 Executor 框架的源码分析以 ThreadPoolExecutor 作为主线，其它的内容也会有所提及，不过请同学们抓住重点，别偏离了主线。

3.1 Executor


代码如下：

```
public interface Executor {  
    void execute(Runnable command);  
}
```

很简单，只是为了把提交任务解耦出来。

3.2 ExecutorService

ExecutorService 定义了线程池管理和更多执行任务的方法，如下：



The screenshot shows the ExecutorService interface with the following methods listed:

- awaitTermination(long, TimeUnit): boolean
- invokeAll(Collection<? extends Callable<T>>): List<Future<T>>
- invokeAll(Collection<? extends Callable<T>>, long, TimeUnit): List<Future<T>>
- invokeAny(Collection<? extends Callable<T>>): T
- invokeAny(Collection<? extends Callable<T>>, long, TimeUnit): T
- isShutdown(): boolean
- isTerminated(): boolean
- shutdown(): void
- shutdownNow(): List<Runnable>
- submit(Callable<T>): Future<T>
- submit(Runnable): Future<?>
- submit(Runnable, T): Future<T>

挑选几个重点的说一下：

shutdown 方法

终止 executorService，不再执行任务新的任务，已经执行的任务会被执行完。

shutdownNow 方法

不等待正在执行的任务完成，强行关闭。不过此方法并不保证正在执行的任务能被强行终止。返回从来没有被执行的任务列表。

submit 方法

对 execute 方法的扩展，会返回一个 Future 对象，持有任务执行结果。

invokeAll 方法

执行一组任务，所有任务都返回或者 timeout 的时候，invokeAll 方法返回执行结果列表。该方法一旦返回结果，没有完成的任务则被取消。

invokeAny 方法

执行一组任务，任意一个任务有返回时，`invokeAny` 返回该任务的执行结果。其余没有完成的任务则被取消。

3.3 AbstractExecutorService

提供了 `newTaskFor` 方法对 `Runnable` 进行包装：

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T value) {  
    return new FutureTask<T>(runnable, value);  
}
```

它对 `submit` 的实现，就是过 `newTaskFor` 方法，代码如下：

```
public Future<?> submit(Runnable task) {  
    if (task == null) throw new NullPointerException();  
    RunnableFuture<Void> ftask = newTaskFor(task, null);  
    execute(ftask);  
    return ftask;  
}
```

这里用到的 `RunnableFuture`，就是为了这个功能而生，它实现了 `Runnable` 接口及 `Future` 接口。所以它可以被传入 `execute` 方法，从而添加进任务列表。此外它还保存了执行的结果，并被返回。

3.4 构造 ThreadPoolExecutor

下面才是本节的重头戏，对 `ThreadPoolExecutor` 的源代码分析。我们从 `ThreadPoolExecutor` 的创建开始。

```
Executors.newFixedThreadPool(10)
```

可以看到是通过 `Executors` 的工厂方法来创建的，`Executor` 提供了多种工厂方法创建 `ThreadPool`。其实根本是调用 `ThreadPoolExecutor` 构造方法时传入参数不同。我们以 `newFixedThreadPool` 方法为例，看一下代码：

```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        0L, TimeUnit.MILLISECONDS,  
        new LinkedBlockingQueue<Runnable>());  
}
```

`ThreadPoolExecutor` 构造方法如下：

```
public ThreadPoolExecutor(int corePoolSize,  
    int maximumPoolSize,  
    long keepAliveTime,  
    TimeUnit unit,  
    BlockingQueue<Runnable> workQueue) {  
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
        Executors.defaultThreadFactory(), defaultHandler);  
}
```

现在我们可以翻一下 `newFixedThreadPool` 定义了一个什么样的线程池：

核心线程数量为 `n`，最大线程数量也为 `n` 的线程池。线程池中线程永远存活。线程池创建线程使用 `defaultThreadFactory`。当无法创建线程时，使用 `defaultHandler`。

corePoolSize 即线程池的核心线程数量，其实也是最小线程数量。不设置 **allowCoreThreadTimeOut** 的情况下，核心线程数量范围内的线程一直存活。

maximumPoolSize 即线程池的最大线程数量。受限于线程池的 **CAPACITY**。线程池的 **CAPACITY** 为 2 的 29 次方 -1。这是由于线程池把线程数量和状态保存在一个整形原子变量中。状态保存在高位，占据了两位，所以线程池中线程数量最多到 2 的 29 次方 -1。

workQueue 是一个阻塞的 **queue**，用来保存线程池要执行的所有任务。

Executors.defaultThreadFactory()，我们看下源代码，发现其最终返回了一个 **DefaultThreadFactory**。代码如下：

```
static class DefaultThreadFactory implements ThreadFactory {
    private static final AtomicInteger poolNumber = new AtomicInteger(1);
    private final ThreadGroup group;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    private 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;
    }
}
```

其实就是规范了生成的 **Thread**。避免调用 **new Thread** 创建，导致创建出来的 **Thread** 可能存在差异。在 **Executor** 中，对线程的创建都是通过 **ThreadFactory**，禁止使用 **new Thread** 来创建。

ThreadPoolExecutor 中还有个重要的属性：

```
/**
 * Set containing all worker threads in pool. Accessed only when
 * holding mainLock.
 */
private final HashSet<Worker> workers = new HashSet<Worker>();
```

通过注释可以看出，这个 **HashSet** 中存的是 **Thread**。而 **Worker** 其实就是对 **Thread** 的进一步封装。

我们再回过头来，看一下 **ThreadPoolExecutor** 的构造函数中做了什么事情：

```

public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
        null :
        AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}

```

可以看到只是对属性的赋值，并没有启动任何线程。这样做是很好的设计，因为没有任何任务添加时就启动线程，是对系统资源的浪费。

通过以上分析，我们对 `ThreadPoolExecutor` 的结构应该比较清晰了，其实核心和我们自己实现的线程池是一样的。`ThreadPoolExecutor` 也有一个任务的列表 `workQueue`，还有一个线程的列表 `worker`。

那么按照我们自己实现的逻辑，线程池应该是通过启动线程轮询从 `workQueue` 中获取任务执行来实现线程池的运转。接下来我们看看猜想是否正确。

3.5 启动 `ThreadPoolExecutor`

既然在创建 `ThreadPoolExecutor` 时并没有启动线程池，那么线程池是何时被启动的呢？我猜应该是添加第一个任务的时候，也就是调用 `execute` 方法时。我们来看看 `execute` 方法的代码：

```

public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (!isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    else if (!addWorker(command, false))
        reject(command);
}

```

源代码中有一段关键的注释我没有贴进来，下面我先把这段关键的注释翻译讲解下：

分三步做处理：

- 1、如果运行的线程数量小于 `corePoolSize`，那么尝试创建新的线程，并把传入的 `command` 作为它的第一个 `task` 来执行。调用 `addWorker` 会自动检查 `runState` 和 `workCount`，以此来防止在不应该添加线程时添加线程的错误警告；
- 2、即使 `task` 可以被成功加入队列，我们仍旧需要再次确认我们是否应该添加 `thread`（因为最后一次检查之后可能有线程已经死掉了）还是线程池在进入此方法后已经停掉了。所以我们会再次检查状态，如果有必要的话，可以回滚队列。或者当没有线程时，开启新的 `thread`；
- 3、如果无法将 `task` 加入 `queue`，那么可以尝试添加新的 `thread`。如果添加失败，这是因为线程池被关闭或者已经饱和了，所以拒绝这个 `task`。

以上是原文的翻译。结合代码，其实就是如下三步：

- 1、线程数量不足 `corePoolSize`时，添加新线程作为 `core thread` 执行 `command`；
- 2、将 `command` 加入 `workQueue`，然后再次检查线程池状态。如果不是 `isRunning`，则移除 `command` 并且 `reject command`。如果线程数量已经为 0，那么则再次 `addWorker`；
- 3、如果无法将 `task` 加入 `workQueue`，则尝试 `addWorker`。但不作为 `core thread`。如果添加失败，则 `reject command`（由于没有加入 `workQueue`，所以不需要从 `queue` 中移除 `command`）。

可以看到 `execute` 流程的核心方法为 `addWorker`。我们继续分析 `addWorker`方法。

`addWork` 中主要执行如下逻辑：

- 1、更新 `worker` 的数量，代码如下：

```

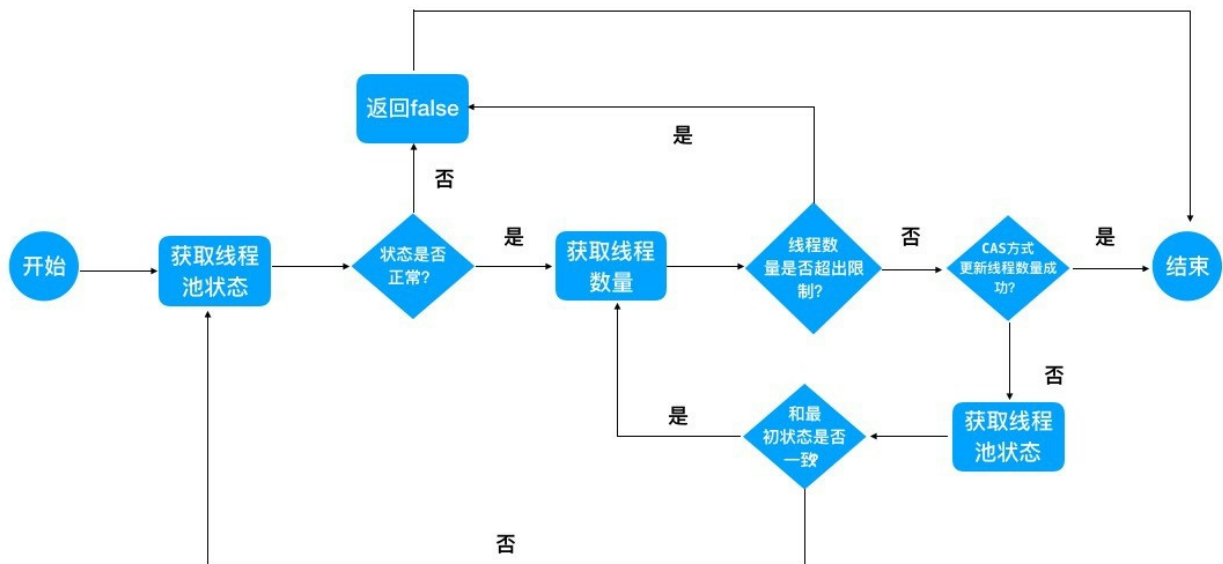
retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // Check if queue empty only if necessary.
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
                firstTask == null &&
                ! workQueue.isEmpty()))
            return false;

        for (;;) {
            int wc = workerCountOf(c);
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get(); // Re-read ctl
            if (runStateOf(c) != rs)
                continue retry;
            // else CAS failed due to workerCount change; retry inner loop
        }
    }
}

```

retry 是一个标记，和循环配合使用，**continue retry** 的时候，会跳到 **retry** 的地方再次执行。如果 **break retry**，则跳出整个循环体。前文提到过，**ThreadPoolExecutor** 把状态和线程池数量两个属性存在了一个 **Atomic** 变量中，就是这里用到的 **ctl**。源码中先检查了状态，然后根据创建线程类型的不同，进行数量的校验。在通过 **CAS** 方式更新 **ctl**，成功的话则跳出循环。否则再次取得线程池状态，如果和最初已经不一致，那么从头开始执行。如果状态并未改变则继续更新 **worker** 的数量。流程参考下图：



2、添加 **worker** 到 **workers** 的 **set** 中。并且启动 **worker** 中持有的线程。代码如下：


```

boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            // Recheck while holding lock.
            // Back out on ThreadFactory failure or if
            // shut down before lock acquired.
            int rs = runStateOf(ctl.get());

            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive()) // precheck that t is startable
                    throw new IllegalThreadStateException();
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                workerAdded = true;
            }
        } finally {
            mainLock.unlock();
        }
        if (workerAdded) {
            t.start();
            workerStarted = true;
        }
    }
} finally {
    if (! workerStarted)
        addWorkerFailed(w);
}
return workerStarted;

```

可以看到添加 **work** 时需要先获得锁，这样确保多线程并发安全。如果添加 **worker** 成功，那么调用 **worker** 中线程的 **start** 方法启动线程。如果启动失败则调用 **addWorkerFailed** 方法进行回滚。过程比较简单，这里就不再提流程图了。

分析到这里，我们先进行下总结。

- 1、**ThreadPoolExecutor** 在初始化后并没有启动和创建任何线程；
- 2、在调用 **execute** 方法时才会调用 **addWorker** 创建线程，并且把 **command** 加入到 **workQueue**（如果已经拥有超过 **core** 数量的线程，则不会再调用 **addWorker** 创建线程）；
- 3、**addWorker** 方法中会创建新的 **worker**，并启动其持有的线程来执行任务。

第二步中，如果线程数量已经达到 **corePoolSize**，则只会把 **command** 加入到 **workQueue** 中，那么加入到 **workQueue** 中的 **command** 是如何被执行的呢？我们下面来分析 **Worker** 的源代码。

3.6 Worker

Worker 封装了线程，是 **executor** 中的工作单元。**worker** 继承自 **AbstractQueuedSynchronizer**，并实现 **Runnable**。

worker 中的属性如下：

```
/** Thread this worker is running in. Null if factory fails. */
final Thread thread;
/** Initial task to run. Possibly null. */
Runnable firstTask;
/** Per-thread task counter */
volatile long completedTasks;
```

如果存在 **firstTask**，那么 **worker** 中线程启动时，会先执行 **firstTask**。

构造方法如下：

```
Worker(Runnable firstTask) {
    setState(-1);
    this.firstTask = firstTask;
    this.thread = getThreadFactory().newThread(this);
}
```

可以看到通过 **ThreadFactory** 创建线程，并没有直接 **new**。原因上文已经将结果。此处还需要特别注意的是，创建 **thread** 时把 **worker** 自己作为 **Runnable** 的实现传入了 **thread** 中。那么 **addWork** 时调用的 **t.start()**，实际上运行的是 **t** 所属 **worker** 的 **run** 方法。**worker** 的 **run** 方法如下：

```
public void run() {
    runWorker(this);
}
```

实际运行的是 **ThreadPoolExecutor** 的 **runWorker** 方法，代码如下：

```

final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
    boolean completedAbruptly = true;
    try {
        while (task != null || (task = getTask()) != null) {
            w.lock();
            if ((runStateAtLeast(ctl.get(), STOP) ||
                (Thread.interrupted() &&
                 runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt();
            try {
                beforeExecute(wt, task);
                Throwable thrown = null;
                try {
                    task.run();
                } catch (RuntimeException x) {
                    thrown = x; throw x;
                } catch (Error x) {
                    thrown = x; throw x;
                } catch (Throwable x) {
                    thrown = x; throw new Error(x);
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
                task = null;
                w.completedTasks++;
                w.unlock();
            }
        }
        completedAbruptly = false;
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}

```

主流程如下：

- 1、先取出 worker 中的 firstTask，并清空；
- 2、如果没有 firstTask，则调用 getTask 方法，从 workQueue 中获取task；
- 3、获取锁；
- 4、执行 beforeExecute。这里是空方法，如有需要在子类实现；
- 5、执行 task.run；
- 6、执行 afterExecute。这里是空方法，如有需要在子类实现；
- 7、清空 task，completedTasks++，释放锁；
- 8、当有异常或者没有 task 可执行时，会进入外层 finally 代码块。调用 processWorkerExit 退出当前 worker。从 works 中移除本 worker 后，如果 worker 数量小于 corePoolSize，则创建新的 worker，以维持 corePoolSize 大小的线程数。

这行代码 `while (task != null || (task = getTask()) != null)`，确保了 `worker` 不停地从 `workQueue` 中取得 `task` 执行。`getTask` 方法会从 `BlockingQueue workQueue` 中 `poll` 或者 `take` 其中的 `task` 出来。

到这里关于 `executor` 如何创建并启动线程执行 `task` 的过程已经分析清楚了。其实和我们自己实现的线程池的核心思想一致，都是通过维护一定数量的线程，并且不断从任务队列取得任务执行来实现线程池的运转。但是 `Executor` 框架考虑得更为全面，健壮性也要好很多。我们在实际开发中不要自己再去设计线程池，请直接使用 `executor`。

4、总结

本节的内容相对比较多，源代码阅读也比较枯燥。我们在阅读源代码时一定要抓住核心流程，从高层级逻辑开始自顶向下分析和阅读。不要过多纠缠于细节，等到大体能够读懂时，再去看感兴趣的细节实现。否则很容易在层层嵌套的源代码中迷失了方向，陷入某个细节不能自拔。其实关于 `ThreadPoolExecutor` 还有些方法，本节没有给出分析，比如 `shutdown` 和 `shutdownNow`，大家可以尝试自己分析下。

}

