二

18 线程池实现“线程复用”的原理?

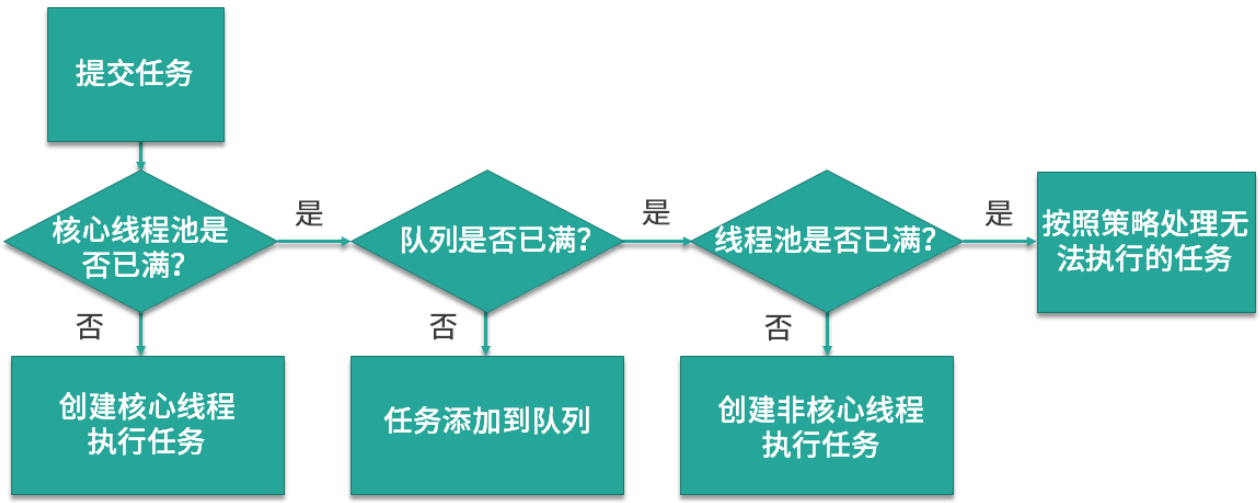
在本课时我们主要学习线程复用的原理，以及对线程池的 execute 这个非常重要的方法进行源码解析。

线程复用原理

我们知道线程池会使用固定数量或可变数量的线程来执行任务，但无论是固定数量或可变数量的线程，其线程数量都远远小于任务数量，面对这种情况线程池可以通过线程复用让同一个线程去执行不同的任务，那么线程复用背后的原理是什么呢?

线程池可以把线程和任务进行解耦，线程归线程，任务归任务，摆脱了之前通过 Thread 创建线程时的一个线程必须对应一个任务的限制。在线程池中，同一个线程可以从 BlockingQueue 中不断提取新任务来执行，其核心原理在于线程池对 Thread 进行了封装，并不是每次执行任务都会调用 Thread.start() 来创建新线程，而是让每个线程去执行一个“循环任务”，在这个“循环任务”中，不停地检查是否还有任务等待被执行，如果有则直接去执行这个任务，也就是调用任务的 run 方法，把 run 方法当作和普通方法一样的地位去调用，相当于把每个任务的 run() 方法串联了起来，所以线程数量并不增加。

我们首先来复习一下线程池创建新线程的时机和规则：



如流程图所示，当提交任务后，线程池首先会检查当前线程数，如果此时线程数小于核心线

程数，比如最开始线程数量为 0，则新建线程并执行任务，随着任务的不断增加，线程数会逐渐增加并达到核心线程数，此时如果仍有任务被不断提交，就会被放入 workQueue 任务队列中，等待核心线程执行完当前任务后重新从 workQueue 中提取正在等待被执行的任务。此时，假设我们的任务特别的多，已经达到了 workQueue 的容量上限，这时线程池就会启动后备力量，也就是 maxPoolSize 最大线程数，线程池会在 corePoolSize 核心线程数的基础上继续创建线程来执行任务，假设任务被不断提交，线程池会持续创建线程直到线程数达到 maxPoolSize 最大线程数，如果依然有任务被提交，这就超过了线程池的最大处理能力，这个时候线程池就会拒绝这些任务，我们可以看到实际上任务进来之后，线程池会逐一判断 corePoolSize 、workQueue 、maxPoolSize ，如果依然不能满足需求，则会拒绝任务。

我们接下来具体看看代码是如何实现的，我们从 execute 方法开始分析，源码如下所示。

```
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);
}
```

线程复用源码解析

这段代码短小精悍，内容丰富，接下来我们具体分析代码中的逻辑，首先看下前几行：

```
//如果传入的Runnable的空，就抛出异常
if (command == null)
    throw new NullPointerException();
```

execute 方法中通过 if 语句判断 command ，也就是 Runnable 任务是否等于 null，如果为 null 就抛出异常。

接下来判断当前线程数是否小于核心线程数，如果小于核心线程数就调用 addWorker() 方法增加一个 Worker，这里的 Worker 就可以理解为一个线程：

```
if (workerCountOf(c) < corePoolSize) {
    if (addWorker(command, true))
        return;
        c = ctl.get();
}
```

那 addWorker 方法又是做什么用的呢？addWorker 方法的主要作用是在线程池中创建一个线程并执行第一个参数传入的任务，它的第二个参数是个布尔值，如果布尔值传入 true 代表增加线程时判断当前线程是否少于 corePoolSize，小于则增加新线程，大于等于则不增加；同理，如果传入 false 代表增加线程时判断当前线程是否少于 maxPoolSize，小于则增加新线程，大于等于则不增加，所以这里的布尔值的含义是以核心线程数为界限还是以最大线程数为界限进行是否新增线程的判断。addWorker() 方法如果返回 true 代表添加成功，如果返回 false 代表添加失败。

我们接下来看下一部分代码：

```
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);
}
```

如果代码执行到这里，说明当前线程数大于或等于核心线程数或者 addWorker 失败了，那么就需要通过 if (isRunning(c) && workQueue.offer(command)) 检查线程池状态是否为 Running，如果线程池状态是 Running 就把任务放入任务队列中，也就是 **workQueue**.offer(**command**)。如果线程池已经不处于 Running 状态，说明线程池被关闭，那么就移除刚刚添加到任务队列中的任务，并执行拒绝策略，代码如下所示：

```
if (! isRunning(recheck) && remove(command))
    reject(command);
```

下面我们再来看后一个 else 分支：

```
else if (workerCountOf(recheck) == 0)
    addWorker(null, false);
```

能进入这个 else 说明前面判断到线程池状态为 Running，那么当任务被添加进来之后就需要防止没有可执行线程的情况发生（比如之前的线程被回收了或意外终止了），所以此时如果检查当前线程数为 0，也就是 workerCountOf**(**recheck**) == 0，那就执行 addWorker() 方法新建线程。

我们再来看最后一部分代码：

```
else if (!addWorker(command, false))
    reject(command);
```

执行到这里，说明线程池不是 Running 状态或线程数大于或等于核心线程数并且任务队列已经满了，根据规则，此时需要添加新线程，直到线程数达到“最大线程数”，所以此时就会再次调用 addWorker 方法并将第二个参数传入 false，传入 false 代表增加线程时判断当前线程数是否少于 maxPoolSize，小于则增加新线程，大于等于则不增加，也就是以 maxPoolSize 为上限创建新的 worker；addWorker 方法如果返回 true 代表添加成功，如果返回 false 代表任务添加失败，说明当前线程数已经达到 maxPoolSize，然后执行拒绝策略 reject 方法。如果执行到这里线程池的状态不是 Running，那么 addWorker 会失败并返回 false，所以也会执行拒绝策略 reject 方法。

可以看出，在 execute 方法中，多次调用 addWorker 方法把任务传入，addWorker 方法会添加并启动一个 Worker，这里的 Worker 可以理解为是对 Thread 的包装，Worker 内部有

一个 Thread 对象，它正是最终真正执行任务的线程，所以一个 Worker 就对应线程池中的一个线程，addWorker 就代表增加线程。线程复用的逻辑实现主要在 Worker 类中的 run 方法里执行的 runWorker 方法中，简化后的 runWorker 方法代码如下所示。

```
runWorker(Worker w) {
    Runnable task = w.firstTask;
    while (task != null || (task = getTask()) != null) {
        try {
            task.run();
        } finally {
            task = null;
        }
    }
}
```

可以看出，实现线程复用的逻辑主要在一个不停循环的 while 循环体中。

1. 通过取 Worker 的 firstTask 或者通过 getTask 方法从 workQueue 中获取待执行的任务。
2. 直接调用 task 的 run 方法来执行具体的任务（而不是新建线程）。

在这里，我们找到了最终的实现，通过取 Worker 的 firstTask 或者 getTask方法从 workQueue 中取出了新任务，并直接调用 Runnable 的 run 方法来执行任务，也就是如之前所说的，每个线程都始终在一个大循环中，反复获取任务，然后执行任务，从而实现了线程的复用。

上一页 下一页