面试官问我ThreadPoolExecutor类的核心流程，我和他扯了半天！

程序员面试 5月31日

以下文章来源于冰河技术，作者冰河团队

前言

打开你的IDE，踏下心来，跟着文章看代码，相信你定能收货满满！！！

核心逻辑概述

ThreadPoolExecutor是Java线程池中最核心的类之一，它能够保证线程池按照正常的业务逻辑执行任务，并通过原子方式更新线程池每个阶段的状态。

ThreadPoolExecutor类中存在一个workers工作线程集合，用户可以向线程池中添加需要执行的任务，workers集合中的工作线程可以直接执行任务，或者从任务队列中获取任务后执行。ThreadPoolExecutor类中提供了整个线程池从创建到执行任务，再到消亡的整个流程方法。本文，就结合ThreadPoolExecutor类的源码深度分析线程池执行任务的整体流程。

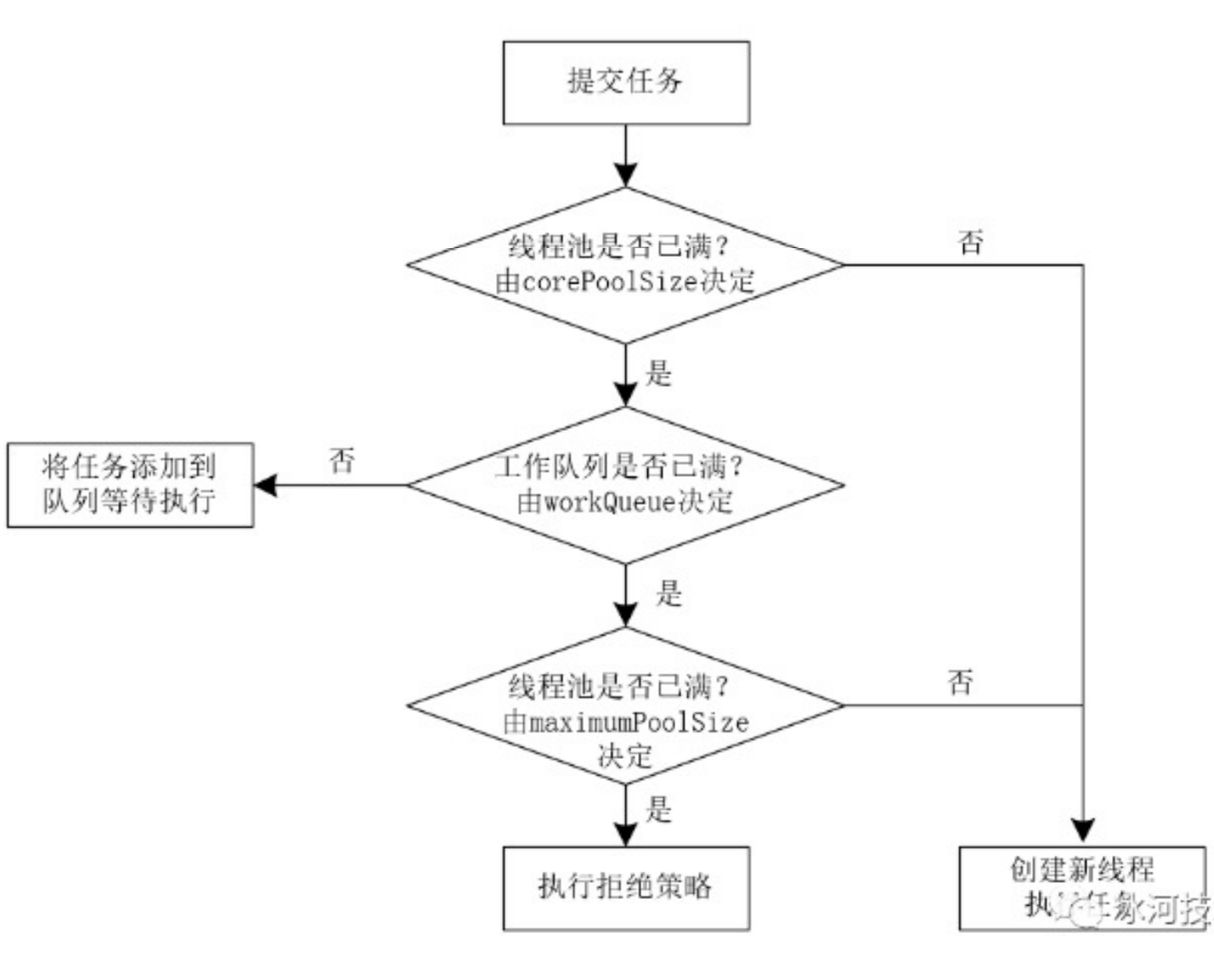
在ThreadPoolExecutor类中，线程池的逻辑主要体现在execute(Runnable)方法，addWorker(Runnable, boolean)方法，addWorkerFailed(Worker)方法和拒绝策略上，接下来，我们就深入分析这几个核心方法。

execute(Runnable)方法

execute(Runnable)方法的作用是提交Runnable类型的任务到线程池中。我们先看下execute(Runnable)方法的源码，如下所示。

```java
public void execute(Runnable command) {
    //如果提交的任务为空，则抛出空指针异常
    if (command == null)
        throw new NullPointerException();
    //获取线程池的状态和线程池中线程的数量
    int c = ctl.get();
    //线程池中的线程数量小于corePoolSize的值
    if (workerCountOf(c) < corePoolSize) {
        //重新开启线程执行任务
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    //如果线程池处于RUNNING状态，则将任务添加到阻塞队列中
    if (isRunning(c) && workQueue.offer(command)) {
        //再次获取线程池的状态和线程池中线程的数量，用于二次检查
        int recheck = ctl.get();
        //如果线程池没有未处于RUNNING状态，从队列中删除任务
        if (! isRunning(recheck) && remove(command))
            //执行拒绝策略
            reject(command);
        //如果线程池为空，则向线程池中添加一个线程
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    //任务队列已满，则新增worker线程，如果新增线程失败，则执行拒绝策略
    else if (!addWorker(command, false))
        reject(command);
}
```

整个任务的执行流程，我们可以简化成下图所示。



接下来，我们拆解execute(Runnable)方法，具体分析execute(Runnable)方法的执行逻辑。

（1）线程池中的线程数是否小于corePoolSize核心线程数，如果小于corePoolSize核心线程数，则向workers工作线程集合中添加一个核心线程执行任务。代码如下所示。

```java
//线程池中的线程数量小于corePoolSize的值
if (workerCountOf(c) < corePoolSize) {
    //重新开启线程执行任务
    if (addWorker(command, true))
        return;
    c = ctl.get();
}
```

（2）如果线程池中的线程数量大于corePoolSize核心线程数，则判断当前线程池是否处于RUNNING状态，如果处于RUNNING状态，则添加任务到待执行的任务队列中。注意：这里向任务队列添加任务时，需要判断线程池是否处于RUNNING状态，只有线程池处于RUNNING状态时，才能向任务队列添加新任务。否则，会执行拒绝策略。代码如下所示。

```java
if (isRunning(c) && workQueue.offer(command))
```

（3）向任务队列中添加任务成功，由于其他线程可能会修改线程池的状态，所以这里需要对线程池进行二次检查，如果当前线程池的状态不再是RUNNING状态，则需要将添加的任务从任务队列中移除，执行后续的拒绝策略。如果当前线程池仍然处于RUNNING状态，则判断线程池是否为空，如果线程池中不存在任何线程，则新建一个线程添加到线程池中，如下所示。

```java
//再次获取线程池的状态和线程池中线程的数量，用于二次检查
int recheck = ctl.get();
//如果线程池没有未处于RUNNING状态，从队列中删除任务
if (! isRunning(recheck) && remove(command))
    //执行拒绝策略
    reject(command);
//如果线程池为空，则向线程池中添加一个线程
else if (workerCountOf(recheck) == 0)
    addWorker(null, false);
```

（4）如果在步骤（3）中向任务队列中添加任务失败，则尝试开启新的线程执行任务。此时，如果线程池中的线程数量已经大于线程池中的最大线程数maximumPoolSize，则不能再启动新线程。此时，表示线程池中的任务队列已满，并且线程池中的线程已满，需要执行拒绝策略，代码如下所示。

```java
//任务队列已满，则新增worker线程，如果新增线程失败，则执行拒绝策略
else if (!addWorker(command, false))
    reject(command);
```

这里，我们将execute(Runnable)方法拆解，结合流程图来理解线程池中任务的执行流程就比较简单了。可以这么说，execute(Runnable)方法的逻辑基本上就是一般线程池的执行逻辑，理解了execute(Runnable)方法，就基本理解了线程池的执行逻辑。

注意：有关ScheduledThreadPoolExecutor类和ForkJoinPool类执行线程池的逻辑，在【高并发专题】系列文章中的后文中会详细说明，理解了这些类的执行逻辑，就基本全面掌握了线程池的执行流程。

在分析execute(Runnable)方法的源码时，我们发现execute(Runnable)方法中多处调用了addWorker(Runnable, boolean)方法，接下来，我们就一起分析下addWorker(Runnable, boolean)方法的逻辑。

addWorker(Runnable, boolean)方法

总体上，addWorker(Runnable, boolean)方法可以分为三部分，第一部分是使用CAS安全的向线程池中添加工作线程；第二部分是创建新的工作线程；第三部分则是将任务通过安全的并发方式添加到workers中，并启动工作线程执行任务。

接下来，我们看下addWorker(Runnable, boolean)方法的源码，如下所示。

```java
private boolean addWorker(Runnable firstTask, boolean core) {
    //标记重试的标识
    retry:
    for (;;) {
        int c = ctl.get();
        int rs = runStateOf(c);

        // 检查队列是否在某些特定的条件下为空
        if (rs >= SHUTDOWN &&
            ! (rs == SHUTDOWN &&
               firstTask == null &&
               ! workQueue.isEmpty()))
            return false;
        //下面循环的主要作用为通过CAS方式增加线程的个数
        for (;;) {
            //获取线程池中的线程数量
            int wc = workerCountOf(c);
            //如果线程池中的线程数量超出限制，直接返回false
            if (wc >= CAPACITY ||
                wc >= (core ? corePoolSize : maximumPoolSize))
                return false;
            //通过CAS方式向线程池新增线程数量
            if (compareAndIncrementWorkerCount(c))
                //通过CAS方式保证只有一个线程执行成功，跳出最外层循环
                break retry;
            //重新获取ctl的值
            c = ctl.get();
            //如果CAS操作失败了，则需要在内循环中重新尝试通过CAS新增线程数量
            if (runStateOf(c) != rs)
                continue retry;
        }
    }

    //跳出最外层for循环，说明通过CAS新增线程数量成功
    //此时创建新的工作线程
    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        //将执行的任务封装成worker
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            //独占锁，保证操作workers时的同步
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                //此处需要重新检查线程池状态
                //原因是在获得锁之前可能其他的线程改变了线程池的状态
                int rs = runStateOf(ctl.get());

                if (rs < SHUTDOWN ||
                    (rs == SHUTDOWN && firstTask == null)) {
                    if (t.isAlive())
                        throw new IllegalThreadStateException();
                    //向worker中添加新任务
                    workers.add(w);
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                    //将是否添加了新任务的标识设置为true
                    workerAdded = true;
                }
            } finally {
                //释放独占锁
                mainLock.unlock();
            }
            //添加新任务成功，则启动线程执行任务
            if (workerAdded) {
                t.start();
                //将任务是否已经启动的标识设置为true
                workerStarted = true;
            }
        }
    } finally {
        //如果任务未启动或启动失败，则调用addWorkerFailed(Worker)方法
        if (! workerStarted)
            addWorkerFailed(w);
    }
    //返回是否启动任务的标识
    return workerStarted;
}
```

乍一看，addWorker(Runnable, boolean)方法还蛮长的，这里，我们还是将addWorker(Runnable, boolean)方法进行拆解。

（1）检查任务队列是否在某些特定的条件下为空，代码如下所示。

```java
// 检查队列是否在某些特定的条件下为空
if (rs >= SHUTDOWN &&
    ! (rs == SHUTDOWN &&
       firstTask == null &&
       ! workQueue.isEmpty()))
    return false;
```

（2）在通过步骤（1）的校验后，则进入内层for循环，在内层for循环中通过CAS来增加线程池中的线程数量，如果CAS操作成功，则直接退出双重for循环。如果CAS操作失败，则查看当前线程池的状态是否发生了变化，如果线程池的状态发生了变化，则通过continue关键字重新通过外层for循环校验任务队列，检验通过再次执行内层for循环的CAS操作。如果线程池的状态没有发生变化，此时上一次CAS操作失败了，则继续尝试CAS操作。代码如下所示。

```java
for (;;) {
    //获取线程池中的线程数量
    int wc = workerCountOf(c);
    //如果线程池中的线程数量超出限制，直接返回false
    if (wc >= CAPACITY ||
        wc >= (core ? corePoolSize : maximumPoolSize))
        return false;
    //通过CAS方式向线程池新增线程数量
    if (compareAndIncrementWorkerCount(c))
        //通过CAS方式保证只有一个线程执行成功，跳出最外层循环
        break retry;
    //重新获取ctl的值
    c = ctl.get();
    //如果CAS操作失败了，则需要在内循环中重新尝试通过CAS新增线程数量
    if (runStateOf(c) != rs)
        continue retry;
}
```

（3）CAS操作成功后，表示向线程池中成功添加了工作线程，此时，还没有线程去执行任务。使用全局的独占锁mainLock来将新增的工作线程Worker对象安全的添加到workers中。

总体逻辑就是：创建新的Worker对象，并获取Worker对象中的执行线程，如果线程不为空，则获取独占锁，获取锁成功后，再次检查线线程的状态，这是避免在获取独占锁之前其他线程修改了线程池的状态，或者关闭了线程池。如果线程池关闭，则需要释放锁。否则将新增加的线程添加到工作集合中，释放锁并启动线程执行任务。将是否启动线程的标识设置为true。最后，判断线程是否启动，如果没有启动，则调用addWorkerFailed(Worker)方法。最终返回线程是否启动的标识。

```java
//跳出最外层for循环，说明通过CAS新增线程数量成功
//此时创建新的工作线程
boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
    //将执行的任务封装成worker
    w = new Worker(firstTask);
    final Thread t = w.thread;
    if (t != null) {
        //独占锁，保证操作workers时的同步
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //此处需要重新检查线程池状态
            //原因是在获得锁之前可能其他的线程改变了线程池的状态
            int rs = runStateOf(ctl.get());

            if (rs < SHUTDOWN ||
                (rs == SHUTDOWN && firstTask == null)) {
                if (t.isAlive())
                    throw new IllegalThreadStateException();
                //向worker中添加新任务
                workers.add(w);
                int s = workers.size();
                if (s > largestPoolSize)
                    largestPoolSize = s;
                //将是否添加了新任务的标识设置为true
                workerAdded = true;
            }
        } finally {
            //释放独占锁
            mainLock.unlock();
        }
        //添加新任务成功，则启动线程执行任务
        if (workerAdded) {
            t.start();
            //将任务是否已经启动的标识设置为true
            workerStarted = true;
        }
    }
} finally {
    //如果任务未启动或启动失败，则调用addWorkerFailed(Worker)方法
    if (! workerStarted)
        addWorkerFailed(w);
}
//返回是否启动任务的标识
return workerStarted;
```

addWorkerFailed(Worker)方法

在addWorker(Runnable, boolean)方法中，如果添加工作线程失败或者工作线程启动失败时，则会调用addWorkerFailed(Worker)方法，下面我们就来看看addWorkerFailed(Worker)方法的实现，如下所示。

```java
private void addWorkerFailed(Worker w) {
    //获取独占锁
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //如果Worker任务不为空
        if (w != null)
            //将任务从workers集合中移除
            workers.remove(w);
        //通过CAS将任务数量减1
        decrementWorkerCount();
        tryTerminate();
    } finally {
        //释放锁
        mainLock.unlock();
    }
}
```

addWorkerFailed(Worker)方法的逻辑就比较简单了，获取独占锁，将任务从workers中移除，并且通过CAS将任务的数量减1，最后释放锁。

拒绝策略

我们在分析execute(Runnable)方法时，线程池会在适当的时候调用reject(Runnable)方法来执行相应的拒绝策略，我们看下reject(Runnable)方法的实现，如下所示。

```java
final void reject(Runnable command) {
    handler.rejectedExecution(command, this);
}
```

通过代码，我们发现调用的是handler的rejectedExecution方法，handler又是个什么鬼，我们继续跟进代码，如下所示。

```java
private volatile RejectedExecutionHandler handler;
```

再看看RejectedExecutionHandler是个啥类型，如下所示。

```java
package java.util.concurrent;

public interface RejectedExecutionHandler {

    void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}
```

可以发现RejectedExecutionHandler是个接口，定义了一个rejectedExecution(Runnable, ThreadPoolExecutor)方法。既然RejectedExecutionHandler是个接口，那我们就看看有哪些类实现了RejectedExecutionHandler接口。

看到这里，我们发现RejectedExecutionHandler接口的实现类正是线程池默认提供的四种拒绝策略的实现类。

至于reject(Runnable)方法中具体会执行哪个类的拒绝策略，是根据创建线程池时传递的参数决定的。如果没有传递拒绝策略，则默认会执行AbortPolicy类的拒绝策略。否则会执行传递的类的拒绝策略。

在创建线程池时，除了能够传递JDK默认提供的拒绝策略外，还可以传递自定义的拒绝策略。如果想使用自定义的拒绝策略，则只需要实现RejectedExecutionHandler接口，并重写rejectedExecution(Runnable, ThreadPoolExecutor)方法即可。例如，下面的代码。

```java
public class CustomPolicy implements RejectedExecutionHandler {

    public CustomPolicy() { }

    public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
        if (!e.isShutdown()) {
            System.out.println("使用调用者所在的线程来执行任务")
            r.run();
        }
    }
}
```

使用如下方式创建线程池。

```java
new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                       60L, TimeUnit.SECONDS,
                       new SynchronousQueue<Runnable>(),
                       Executors.defaultThreadFactory(),
                new CustomPolicy());
```

至此，线程池执行任务的整体核心逻辑分析结束。