自定义实现Java线程池-完善异常处理和去除同步

> 本篇文章将延续上篇文章的思路继续完善线程池执行功能,主要完善execute方法线程安全的前提下去除同步锁和对任务执行异常的处理,上篇文章最终代码如下:

```
@Override
public void execute(Runnable command) {
    // 校验参数有效性
    if (command == null)
        throw new NullPointerException("command is null ...");

    // 如果实际线程数量小于核心线程数，
    if (getWorkCount() < coreThreadPool)
        // 初始化线程执行任务
        addThread(command);
    else if (workQueue.offer(command)) {
        // 任务放入队列成功，校验核心线程是否为0
        if (getWorkCount() == 0)
            // 初始化一条不携带任务的线程，让它从队列中获取任务
            addThread(null);
    } else if (getWorkCount() < maxThreadPool) {
        // 初始化非核心线程
        addThread(command);
    } else if (getWorkCount() >= maxThreadPool)
        // 提交任务过多，线程池处理过不来，抛出异常
        throw new RejectedExecutionException(
                "command id too much, reject execute ...");
}

private synchronized void addThread(Runnable task) {
    // 由于execute方法调用addThread(null)，此处参数非空校验得去掉
    // 去掉不保证会有代码恶意调用，所以此方法不能泄露，必须用private修饰

    // 为保证代码健壮性，常规参数校验
    // if (task == null)
    // throw new NullPointerException("taks is null ...");

    // 增加一条线程
    increaseWork();
    new Worker(task).thread.start();
}

// 提供getTask方法从队列中取值
private Runnable getTask() {
    boolean timeOut = false;
    for (;;) {
        boolean timed = getWorkCount() > coreThreadPool;

        // 利用timed和timeOut一起判断线程是否应该死亡
        if (timed && timeOut) {
            // 线程即将死亡，wc自减1
            decreaseWork();
            // 返回为空意味着跳出while循环，线程即将死亡
            return null;
        }
        try {
            // 从队列中取值
            Runnable task = timed ? workQueue.poll(keepAliveTime,
                    TimeUnit.NANOSECONDS) : workQueue.take();
            if (task != null)
                return task;
            // 非核心线程获取超时，直接返回null，跳到循环初始处再次和
            // timed变量判断，防止核心线程死亡
            timeOut = true;
        } catch (InterruptedException e) {
            timeOut = false;
        }
    }
}
```

```
private final class Worker implements Runnable {
    // 由于Worker只是一个Runnable需要一个thread对象来启动它
    private final Thread thread;
    private final Runnable task;

    public Worker(Runnable task) {
        this.task = task;
        // 将初始化的线程赋值给thread
        this.thread = new Thread(this);
    }

    @Override
    public void run() {
        runWorker(this);
    }
}

private void runWorker(Worker worker) {
    Runnable r = worker.task;
    // 此处切记要用while循环而不是用if判断，有以下两个原因：
    // 1. 由于addThread有task参数校验，task不可能为null，这就保证了r != null
    // 始终条件成立，如果用if，那么r = getTask()将永远不会执行
    // 2. 采用while循环也间接保持了线程的活性，后续会解释
    while (r != null || (r = getTask()) != null) {
        // 执行任务
        r.run();
        // 注意：使用while循环，r执行完成之后需要显式置空，否则将会造成活锁
        r = null;
    }
    System.out.println(Thread.currentThread().getName() + " will dead ...");
}

private synchronized int decreaseWork() {
    return --wc;
};

private synchronized int increaseWork() {
    return ++wc;
};

private synchronized int getWorkCount() {
    return wc;
};
```

以上存在两点问题:

- 在runWorker方法中任务如果执行过程中发生异常,线程将会死亡,但是ctl的值并未变化,此处存在问题

针对问题1,可以对runWorker方法做如下修改:

```
private void runWorker(Worker worker) {
    Runnable r = worker.task;
    // 用后置空,有利于gc
    worker.task = null;
    boolean taskException = true;
    try {
        // 此处切记要用while循环而不是用if判断，有以下两个原因：
        // 1. 由于addThread有task参数校验，task不可能为null，这就保证了r != null
        // 始终条件成立，如果用if，那么r = getTask()将永远不会执行
        // 2. 采用while循环也间接保持了线程的活性，后续会解释
        while (r != null || (r = getTask()) != null) {
            try {
                // 执行任务
                r.run();
            } catch (Exception e) {
                throw e;
            } finally {
                // 注意:使用while循环,r执行完成之后需要显式置空,否则将会造成活锁
                r = null;
            }
        }
        taskException = false;
```

```
    } finally {
        /**
         * 无论是否异常,都进行收尾工作,这个方法执行完成,当前线程就会死亡
         */
        processWorkerExit(worker, taskException);
    }
}

private void processWorkerExit(Worker w, boolean taskException) {
    // taskException用来判断线程是否是正常死亡,如果非正常死亡,需要将ctl值自减1
    if (taskException)
        decrementWorkerCount();
}

private void decrementWorkerCount() {
    do {
    } while (!compareAndDecrementWorkerCount(ctl.get()));
}

private boolean compareAndDecrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect - 1);
}
```

在runWorker方法中加入taskException变量来判断任务是正常结束还是异常结束,最终决定对ctl的操作.

- 程序中对wc的访问和修改均作了同步,严重影响线程池执行任务的性能.
  针对此问题,将引入jdk原子类AtomicInteger类来处理,代码如下:

```
@Override
public void execute(Runnable command) {
    // 校验参数有效性
    if (command == null)
        throw new NullPointerException("command is null ...");

    // 如果实际线程数量小于核心线程数,
    if (ctl.get() < coreThreadPool)
        // 初始化线程执行任务
        addThread(command);
    else if (workQueue.offer(command)) {
        // 任务放入队列成功，校验核心线程是否为0
        if (ctl.get() == 0)
            // 初始化一条不携带任务的线程，让它从队列中获取任务
            addThread(null);
    } else if (ctl.get() < maxThreadPool) {
        // 初始化非核心线程
        addThread(command);
    } else if (ctl.get() >= maxThreadPool)
        // 提交任务过多，线程池处理过不来，抛出异常
        throw new RejectedExecutionException(
                "command id too much, reject execute ...");
}

/**
 * 此方法加锁会严重影响线程池提交任务的性能,不加锁保证不了并发情况下wc的值和实际线程数量不相同
 * 参考ThreadPoolExecutor类的addWorker方法,重构代码如下:
 */
private void addThread(Runnable task) {
    for (;;) {
        int c = ctl.get();
        // 超出线程池容量,直接返回
        if (c > CAPACITY)
            return;
        // cas修改ctl的值
        if (compareAndIncrementWorkerCount(c))
            break;
    }
    // 到此说明cas操作成功
    boolean workerStarted = false;
    Worker w = null;
    try {
        w = new Worker(task);
        final Thread t = w.thread;
        if (t != null) {
```

```
            // 预先检查线程是否已经开始执行
            if (t.isAlive())
                throw new IllegalThreadStateException();
            t.start();
            workerStarted = true;
        }
    } finally {
        // worker添加失败,将ctl值自减1
        if (!workerStarted)
            addWorkerFailed();
    }
}

// 提供getTask方法从队列中取值
private Runnable getTask() {
    boolean timeOut = false;
    for (;;) {
        boolean timed = ctl.get() > coreThreadPool;

        // 利用timed和timeOut一起判断线程是否应该死亡
        if (timed && timeOut) {
            // 线程即将死亡，ctl自减1
            decrementWorkerCount();
            // 返回为空意味着跳出while循环，线程即将死亡
            return null;
        }
        try {
            // 从队列中取值
            Runnable task = timed ? workQueue.poll(keepAliveTime,
                    TimeUnit.NANOSECONDS) : workQueue.take();
            if (task != null)
                return task;
            // 非核心线程获取超时，直接返回null，跳到循环初始处再次和
            // timed变量判断，防止核心线程死亡
            timeOut = true;
        } catch (InterruptedException e) {
            timeOut = false;
        }
    }
}

private final class Worker implements Runnable {
    // 由于Worker只是一个Runnable需要一个thread对象来启动它
    private final Thread thread;
    private Runnable task;

    public Worker(Runnable task) {
        this.task = task;
        // 将初始化的线程赋值给thread
        this.thread = new Thread(this);
    }

    @Override
    public void run() {
        runWorker(this);
    }
}

private void runWorker(Worker worker) {
    Runnable r = worker.task;
    // 用后置空,有利于gc
    worker.task = null;
    boolean taskException = true;
    try {
        // 此处切记要用while循环而不是用if判断，有以下两个原因：
        // 1. 由于addThread有task参数校验，task不可能为null，这就保证了r != null
        // 始终条件成立，如果用if，那么r = getTask()将永远不会执行
        // 2. 采用while循环也间接保持了线程的活性，后续会解释
        while (r != null || (r = getTask()) != null) {
            try {
                // 执行任务
                r.run();
            } catch (Exception e) {
                throw e;
            } finally {
```

```
                // 注意:使用while循环,r执行完成之后需要显式置空,否则将会造成活锁
                r = null;
            }
        }
        taskException = false;
    } finally {
        /**
         * 无论是否异常,都进行收尾工作,这个方法执行完成,当前线程就会死亡
         */
        processWorkerExit(worker, taskException);
    }
}

private boolean compareAndIncrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect + 1);
}

private void addWorkerFailed() {
    decrementWorkerCount();
}

private void decrementWorkerCount() {
    do {
    } while (!compareAndDecrementWorkerCount(ctl.get()));
}

private boolean compareAndDecrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect - 1);
}

private void processWorkerExit(Worker w, boolean taskException) {
    // taskException用来判断线程是否是正常死亡,如果非正常死亡,需要将ctl值自减1
    if (taskException)
        decrementWorkerCount();
}
```

> 将addThread方法锁去除,内部采用cas操作volatile变量实现对ctl的安全自增,操作ctl成功之后,将初始化Worker类启动线程执行任务,然后引入workerStarted来判断线程是否成功启动,最终在finally块中来处理线程启动之后的一些收尾工作,addThread方法修改之后虽去除了锁,但是导致了修改ctl的值和初始化Worker不是原子操作,但是对ctl的操作是原子操作,初始化Worker操作最终也会在finally块中校验,这样做保证了ctl值和实现线程数的最终一致性.试想一下,假如在线程池中,刚有一个线程将ctl自增1,刚好线程池已到最大上限,但是还未初始化Worker,此时又有另一条线程提交任务,这时默认会抛出RejectedExecutionException,第一条线程会继续初始化Worker启动线程,最终结果一条线程正常执行,一条线程在提交任务时会抛出异常,而且ctl值和实际线程数最终也是吻合的,只不过会失去ctl值和线程数的实时一致性,极大提高了并发性,权衡之下这种结果对于线程池来说也是可以忍受的.再试想一下加入第一条线程将ctl自增1之后,初始化Worker之后启动失败,最终也会将ctl自减1来弥补,这样第二条线程提交任务会失败,但是不影响第三条线程提交任务.