自定义实现Java线程池-模拟jdk线程池执行流程

Doug Lea大神镇楼

> 本篇文章将追寻ThreadPoolExecutor类的代码思路模拟jdk下线程池的执行策略，实现一个自己的线程池，文章目的主要是为了研究jdk下ThreadPoolExecutor的设计思想，理解作者为什么要这样设计，这么设计有什么好处，而不是为了重复造轮子，建议在工作中尽量使用类库，具体原因请参见《Effective Java》第二版第47条，废话不多说，看代码
> 在JDK 5.0的时候引入了对Java并发编程的规范提案，在JDK中该规范由java.util.concurrent包实现，此包关于线程池的最顶层接口是Executor，代码如下：

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

> 本文章将实现此接口来实现一个自己的线程池，将不考虑线程池的生命周期变化，只考虑其执行任务流程，类中主要属性和构造方法如下：

```
// 线程数量
private int wc = 0;
// 核心线程数
private volatile int coreThreadPool;
// 最大线程数
private volatile int maxThreadPool;
// 线程存活时间
private volatile long keepAliveTime;
// 工作队列
private volatile BlockingQueue<Runnable> workQueue;

// 构造方法，暂不考虑线程工厂和拒绝执行策略
public SampleThreadPoolExecutor(int coreThreadPool, int maxThreadPool, long keepAliveTime,
            TimeUnit unit, BlockingQueue<Runnable> workQueue) {
    if (coreThreadPool < 0 || coreThreadPool > maxThreadPool || maxThreadPool <= 0
            || keepAliveTime < 0)
        throw new IllegalArgumentException("arg is illegal ...");
    if (workQueue == null)
        throw new NullPointerException("work queue is null ...");
    this.coreThreadPool = coreThreadPool;
    this.maxThreadPool = maxThreadPool;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.workQueue = workQueue;
}
```

> 根据线程池执行策略分析，execute方法一旦运行，将有以下几中操作发生：
>
> 1. 初始化线程执行提交的任务
> 2. 任务执行不过来，放入工作队列

3. 任务过多，线程和队列均处理不过来，拒绝执行（本文中将抛出RejectedExecutionException作为拒绝执行策略）

总结以上三点，简单代码如下：

```
@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 (wc < coreThreadPool)
    // 初始化线程执行任务
      addThread(command);
    else if (!workQueue.offer(command) && wc < maxThreadPool)
        // 任务放入阻塞队列失败且实际线程数大于等于核心线程数且小于最大线程数
        // 初始化非核心线程执行任务
        addThread(command);
    else if (wc >= 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 void addThread(Runnable task) {
    // 为代码健壮性考虑，常规参数校验
    if (task == null)
        throw new NullPointerException("taks is null ...");

    // 增加一条线程
    wc++;
    new Thread(task).start();
}
```

以上代码仔细检查之后会发现有以下几个问题：
问题1：addThread方法和wc的自增操作不保证原子性，在多线程环境下，wc的值和实际线程数量不吻合，关于wc的判断将不准确，影响总体线程池执行策略
针对问题1，可以考虑将对wc变量的修改置为原子性，以此类推就得将对wc的访问也置为原子性，最后还得保证wc++和new Thread两个操作具有原子性，修改后代码如下：

```
@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，
    // 不直接读取wc的值，而是通过同步方法getWorkCount来读取
    if (getWorkCount() < coreThreadPool)
        // 初始化线程执行任务
        addThread(command);
    else if (!workQueue.offer(command) && getWorkCount() < maxThreadPool)
        // 任务放入阻塞队列失败且实际线程数大于等于核心线程数且小于最大线程数
        // 初始化非核心线程执行任务
        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 ...");
}

// 给addThread加锁保证wc时刻与线程数量一致（前提是thread不死亡）
private synchronized void addThread(Runnable task) {
    // 为代码健壮性考虑，常规参数校验
    if (task == null)
        throw new NullPointerException("taks is null ...");

    // 增加一条线程
    increaseWork();
    // 此处会衍生出一个问题，那就是此线程执行完成死亡wc的值并未自减1，此问题后续给出答案
    new Thread(task).start();
}

// 通过加锁来保证修改wc原子性
```

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

// 修改wc加锁，那么读取wc不加锁也是不安全的
private synchronized int getWorkCount() {
    return wc;
};
```

> 问题2：这套策略虽然使用了workQueue，但是只将任务放进去了，并没有取出来执行，导致任务积压在队列中，队列此时倒起了副作用
> 针对问题2的情况分析，所欠缺的只是从workQueue中获取任务交给任务执行，此分为两步，从队列中取任务和将任务交给执行线程，所以修改代码后如下：

```
@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) && getWorkCount() < maxThreadPool)
        // 任务放入阻塞队列失败且实际线程数大于等于核心线程数且小于最大线程数
        // 初始化非核心线程执行任务
        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) {
    // 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 Thread(new Runnable() {
        Runnable r = task;

        @Override
        public void run() {
            // 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;
            }
        }
    }).start();
}

// 提供getTask方法从队列中取值
private Runnable getTask() {
    try {
        // 从队列中取值
        Runnable task = workQueue.take();
        if (task != null)
            return task;
    } catch (InterruptedException e) {
    }
    return null;
}

private synchronized int increaseWork() {
```

```
    return ++wc;
};

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

此时代码主要变化如下，不再将线程池提交的Runnable直接放大线程中初始化启动，而是给此Runnable找了一个代理匿名Runnable对象，再次匿名Runnable的run方法中执行提交的任务，加入代理类好处如下：

1. 可以在代理类中执行任务前后加入逻辑，将getTask方法嵌入此代理线程中，试想一下，如果没有代理类，那么getTask方法将很难放置以达到前文所说的，从队列中取任务和将任务交给执行线程这两点
2. 在代理Runnable中加入while循环将会保证执行线程的活性，只要while循环一直满足那么线程run方法就执行不完，线程就不会死

基于以上两点说明引入代理Runnable对象是合理的，也就完全可以将此匿名类封装为一个实现Runnable接口的内部类，也回答了上篇文章的第一个问题：
线程池为什么要封装一个内部类Worker，既然execute方法参数是Runnable，为什么不直接调用线程工厂初始化线程执行？

问题3：在问题2中已经保证了线程池内线程复用，那么下一步就是保证非核心线程的存活时间，也就是keepAliveTime参数的问题，没有保证复用也就更不能保证线程存活时间？
针对问题3的情况分析，要想保证非核心线程存活时间只能从代理线程的run方法入手，只有run方法执行完成线程就会死亡，所以只要控制run方法的耗时就可以，具体代码如下：

```
@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) && getWorkCount() < maxThreadPool)
        // 任务放入阻塞队列失败且实际线程数大于等于核心线程数且小于最大线程数
        // 初始化非核心线程执行任务
        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) {
    // 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 Thread(new Runnable() {
        Runnable r = task;

        @Override
        public void run() {
            // 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 ...");
        }
    }).start();
```

```
}

// 提供getTask方法从队列中取值
private Runnable getTask() {
    boolean timed = getWorkCount() > coreThreadPool;
    try {
        // 从队列中取值
        Runnable task = timed ? workQueue.poll(keepAliveTime,
                TimeUnit.NANOSECONDS) : workQueue.take();
        if (task != null)
            return task;
    } catch (InterruptedException e) {
    }
    // 线程死亡，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;
}

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

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

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

问题3提到的非核心线程生命周期的控制由workQueue的带有时限控制的poll方法来实现，也就是说非核心线程从阻塞队列中获取任务只停留keepAliveTime纳秒，超时未取到任务将返回null，至于线程是否是核心线程的判断是由timed变量实现，初看之下似乎没有问题，但是测试时会发现核心线程也会死亡，也就是说核心线程也执行了workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)，导致这个问题的原因是多线程情况下的不同步造成的，有getTask方法没有上锁，假设此时有2个核心线程和2个非核心线程都进入此方法，此时会在getWorkCount方法处排队等待锁以获取timed变量，假设此时前2条线程已经获取到timed，此时timed为4 > 2为true（也就是此两条线程为非核心线程），那么就会执行workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)方法，此时假如workQueue内已没有任务，这2条线程会一直阻塞直到超时，那么在阻塞期间其它两条核心线程也获取timed变量，由于其它两条线程还在阻塞，并未执行decreaseWork方法修改wc变量，所以2条核心线程也获取到为true的timed变量，最后也进入到行workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)方法中，也就造成上文说的核心线程也受到keepAliveTime参数影响，这违背了线程池的执行策略。
综上所述，就是因为workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)方法和decreaseWork之间缺乏原子性，这两个方法执行之间有其他线程进来获取错误的timed值，这可以将getTask方法上锁来解决，不过上锁导致两个问题：

1. getTask方法将串行执行，影响线程池性能。
2. 没有任务时核心线程池内的存活的核心线程只有一条会进入getTask会在workQueue处阻塞，其它核心线程将在执行getTask方法获取锁的过程中阻塞而不是在workQueue中阻塞，新任务加入到队列中是要先获取锁在从队列取任务，这会导致线程多走一段冤枉路。另一方面线程在获取锁时阻塞会影响线程中断。

综上两点所述给getTask上锁这种办法不可取，到此时有三个问题要处理：

1. 保证核心线程不死
2. 保证非核心线程超时死亡
3. 保证所有存活线程在workQueue处等待任务

针对此3个问题，解决代码如下：

```
@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) && getWorkCount() < maxThreadPool)
        // 任务放入阻塞队列失败且实际线程数大于等于核心线程数且小于最大线程数
        // 初始化非核心线程执行任务
        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) {
    // 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 Thread(new Runnable() {
        Runnable r = task;

        @Override
        public void run() {
            // 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 ...");
        }
    }).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 synchronized int decreaseWork() {
    return --wc;
};

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

private synchronized int getWorkCount() {
```

```
    return wc;
};
```

> 到目前为止存在一个问题，假如初始化一个如下参数的线程池new SampleThreadPoolExecutor(0, 4, 3, TimeUnit.SECONDS, new ArrayBlockingQueue(2))，但是只提交两个任务（任务数不超过工作队列），那么结果将不会有任务会被取出执行，因为核心线程数为0，线程池满了之后再没有任务提交，不会初始化非核心线程，所以任务将永久停留在队列中，针对以上情况修改代码如下：

```
@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 Thread(new Runnable() {
        Runnable r = task;

        @Override
        public void run() {
            // 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 ...");
        }
    }).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 synchronized int decreaseWork() {
    return --wc;
};

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

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

再将代理Runnable类封装进内部类Worker中，重构代码如下：

```
@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;
    };
```

> 到处为止，线程池的基本执行策略将大致模拟出来，当然此类还有许多的缺点，例如对于异常的处理，过多的锁导致性能下降等问题（此部分读者可参考jdk下ThreadPoolExecutor类自行实现），再次建议读者尽量使用类库中的代码，本文手写实现线程池也并非为了投入使用，只是向读者展示一下我对jdk下线程池的设计思想的一些理解，在实现过程中也有许多思想是参照jdk下的解决思想，最后感谢Doug Lea大神为我们提供这么神奇的代码