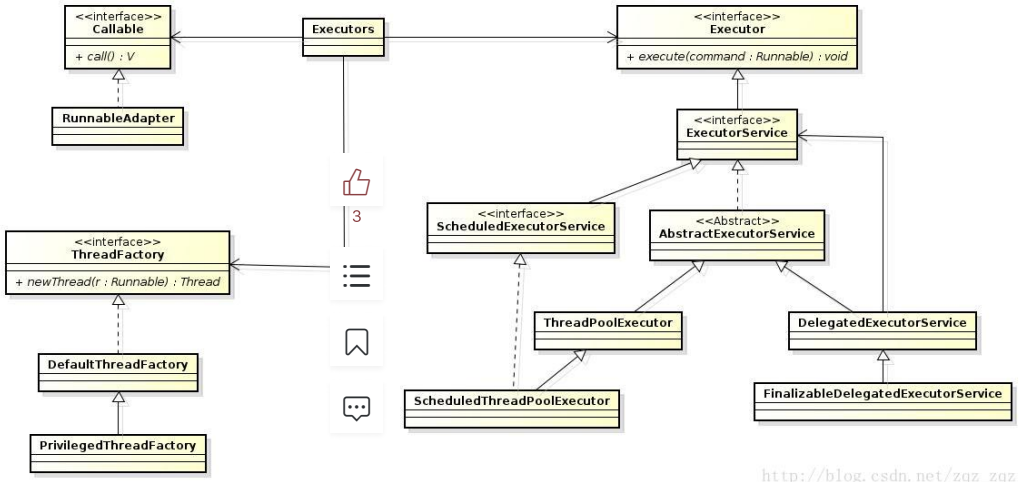
java程序员必精--从源码讲解java线程池ThreadPoolExecuter的实现原理

原创 2017年04月06日 23:33:56 线程池 / 源码 / ThreadPoolExecutor 2512

线程池基本在每个应用中都会用到，而线程池涉及到的细节非常多，要想用好它，仅仅是了解它的api调用是不行的，而且如果你经常分析java线程堆栈，不了解线程池，那么涉及到线程池堆栈的代码也很难看懂，所以作为java程序员，应该好好研究下线程池的实现！

类结构图



示例

```
1  public class ThreadPoolTest {
2      //固定大小的线程池：
3      //初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockin
4  gQuene作为阻塞队列，当线程池没有可执行任务时，也不会释放线程。
5      private static ExecutorService executor = Executors.newFixedThreadPool(10);
6      private static ThreadPoolExecutor executor_ = new ThreadPoolExecutor(10, 10, 0L,
7   TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());
8
9      //缓存线程池：
10     //初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX_VALUE，即214
11 7483647，内部使用SynchronousQueue作为阻塞队列；
12     //和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空
13 闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任
14 务，会导致一定的系统开销；
15     //所以，使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程可能导致严重的性能问题；
16     private static ExecutorService executor2 = Executors.newCachedThreadPool();
17     private static ExecutorService executor2_ = new ThreadPoolExecutor(0,
18 Integer.MAX_VALUE,60L, TimeUnit.SECONDS,new SynchronousQueue<Runnable>());
19
20     //初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的
21 线程可以保证所提交任务的顺序执行，内部使用LinkedBlockingQueue作为阻塞队列。
22     private static ExecutorService executor3 = Executors.newSingleThreadExecutor();
23 //   因为FinalizableDelegatedExecutorService类是不可直接访问的，这样写会报错，所以注释掉
24 //   private static ExecutorService executor3_ = new FinalizableDelegatedExecutorServ
25 ice
26 //            (new ThreadPoolExecutor(1, 1,
27 //                    0L, TimeUnit.MILLISECONDS,
28 //                    new LinkedBlockingQueue<Runnable>()));
29
30     //定时任务线程池：
31     //初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池
32 定期的同步数据。
33     private static ScheduledExecutorService executor4 = Executors.newScheduledThread
34 Pool(5);
35 //   因为new DelayedWorkQueue()这个类是内部类所以这里也不可以直接这样写，这样写是为了让大家了解
36 它的实现本质
37 //   private static ExecutorService executor4_ = new ThreadPoolExecutor(1, Integer.MA
38 X_VALUE, 0, NANOSECONDS,
39 //            new DelayedWorkQueue());
40
41
42     public static void main(String[] args){
43         if(!executor.isShutdown())
               executor.execute(new Task());
           Future f = executor.submit(new Task());
```

```
    static class Task implements Runnable{
        public void run(){
            System.out.println(Thread.currentThread().getName());
        }
    }
```

基础参数

我们先来总结一下线程池的这些参数 ，不再上源码就好理解了
core,maxPoolSize,keepalive
执行任务时
1. 如果线程池中线程数量 < core，新 个线程执行任务；
2. 如果线程池中线程数量 >= core ,则 务放入任务队列
3. 如果线程池中线程数量 >= core 且 xPoolSize，则创建新的线程；
4. 如果线程池中线程数量 > core ,当线程空闲时间超过了keepalive时，则会销毁线程；由此可见线程池的队列如果是无界队列，那么设置线程池最大数量是无效的；

自带线程池的各种坑

- Executors.newFixedThreadPool(10);

固定大小的线程池，它的实现

```
new ThreadPoolExecutor(10, 10, 0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());
```

初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockingQuene作为阻塞队列，超时时间为0，当线程池没有可执行任务时，也不会释放线程。
因为队列LinkedBlockingQueue大小为默认的Integer.MAX_VALUE，可以无限的往里面添加任务，直到内存溢出；

- Executors.newCachedThreadPool();

缓存线程池，它的实现:

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

初始化一个可以缓存线程的线程池，默认超时时间60s，线程池的最小线程数时0，但是最大线程数为Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；

因为线程池的最大值了Integer.MAX_VALUE，会导致无限创建线程；所以，使用该线程池时，一定要注意控制并发的任务数，如果短时有大量任务要执行，就会创建大量的线程，导致严重的性能问题（线程上下文切换带来的开销），线程创建占用堆外内存，如果任务对象也不小，它就会使堆外内存和堆内内存其中的一个先耗尽，导致oom;

- Executors.newSingleThreadExecutor()

单线程线程池，它的实现

```
new FinalizableDelegatedExecutorService(
    new ThreadPoolExecutor(1, 1,0L,
                           TimeUnit.MILLISECONDS,
                           new LinkedBlockingQueue<Runnable>()
                           )
);
```

同newFixedThreadPool线程池一样，队列用的是LinkedBlockingQueue，队列大小为默认的Integer.MAX_VALUE，可以无限的往里面添加任务，直到内存溢出；

源码分析java.util.concurrent.ThreadPoolExecutor

构造方法

它的构造方法有很多，但是最终调用的都是下面这个构造方法

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolS    = 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTim    )
        throw new Il    ArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

参数说明

- corePoolSize（核心线程池大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。
- maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
- ThreadFactory：用于设置创建线程的工厂。默认使用Executors内部类DefaultThreadFactory，可以通过实现ThreadFactory接口，写自己的Factory，通过线程工厂给每个创建出来的线程设置更有意义的名字，Debug和定位问题时非常又帮助；
- keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
- TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。
- workQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

```
1.ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
2.LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
3.SynchronousQueue：一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
4.PriorityBlockingQueue：一个具有优先级得无限阻塞队列。
```

- RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。

```
这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是提供的四种策略。
1.AbortPolicy：直接抛出异常。默认策略
2.CallerRunsPolicy：只用调用者所在线程来运行任务。
3.DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
```

7 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。

重要的成员变量

先看下重要的成员变量ctl及其相关常量

ctl

它记录了当前线程池的运行状态和线程池内的线程数；一个变量是怎么记录两个值的呢？它是一个AtomicInteger 类型，有32个字节，这个32个字节高3位用来标识线程池的运行状态，低29位用来标识线程池内当前存在的线程数；

```
1 //利用低29位表示线程池中线程数，通过高3位表示线程池的运行状态:
2 private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
```

线程池状态

线程池有5种状态，这五种状态由五个静态常量标识，每种状态的值的大小
RUNNING < shutdown < stop < tidying < terminated;

```
1  //32-3 = 29 ，低位29位存储线程池中线程数
2  private static final int COUNT_BITS = Integer.SIZE - 3;
3  //线程池最多可以有536870911个线程，一般绝对创建不到这么大
4  private static final int CAPACITY   = (1 << COUNT_BITS) - 1;
5
6  //RUNNING线程池能接受新任务（只有running状态才会接收新任务），并且可以运行队列中的任务
7  //-1的二进制为32个1，移位后为：11100000000000000000000000000000
8  private static final int RUNNING = -1 << COUNT_BITS;
9
10 //SHUTDOWN不再接受新任务，但仍可以执行队列中的任务
11 //0的二进制为32个0，移位后还是全0
12 private static final int SHUTDOWN = 0 << COUNT_BITS;
13
14 //STOP不再接受新任务，不再执行队列中的任务，而且要中断正在处理的任务
15 //1的二进制为前面31个0，最后一个1，移位后为：00100000000000000000000000000000
16 private static final int STOP = 1 << COUNT_BITS;
17
18 //TIDYING所有任务均已终止，workerCount的值为0，转到TIDYING状态的线程即将要执行terminated()钩
19 子方法.
20 //2的二进制为01000000000000000000000000000000
21 private static final int TIDYING = 2 << COUNT_BITS;
22
23 //TERMINATED terminated()方法执行结束.
24 //3移位后01100000000000000000000000000000
25 private static final int TERMINATED = 3 << COUNT_BITS;
```

要牢记以下几点：

1. 只有RUNNING状态下才会接收新任务；
2. 只有RUNNING状态和SHUTDOWN状态才会执行任务队列中的任务；
3. 其它状态都不会接收新任务，不会执行任务队列中的任务；

状态之间转换关系如下

- RUNNING -> SHUTDOWN
  调用了shutdown方法，线程池实现了finalize方法，在里面调用了shutdown方法，因此shutdown可能是在finalize中被隐式调用的
  (RUNNING or SHUTDOWN) -> STOP
  调用了shutdownNow方法
- SHUTDOWN -> TIDYING
  当队列和线程池均为空的时候

- TIDYING -> TERMINATED
  处于TIDYING状态后最终会进入TERMINATED状态

与ctl相关的三个方法

```
//获取线程池的状态,也就是将ctl低29位都置为0后的值
private static int runSt    (int c)     { return c & ~CAPACITY; }
//获取线程池中线程数，也就是    29位的值
private static int workerCountOf(int c)  { return c & CAPACITY; }
//设置ctl的值，rs为线程池状    为线程数;
private static int ctlOf     rs, int wc) { return rs | wc; }
```

workers

```
用来存储线程池中的线程，线程    装成了Worker对象
private final HashSet<Worker> workers = new HashSet<Worker>();
```

completedTaskCount

```
//记录了已经销毁的线程，完成的任务总数;
private long completedTaskCount;
```

线程池的运行

前面内容都是理解源码的基础，下面开始讲解重要的运行方法，阅读前了解前面的内容才能更好的理解下面方法的运行原理;

添加任务execute方法

线程池是调用该方法来添加任务的，所以我们就从这个方法看起;
它传入的参数为实现了Runnable接口的对象，要执行的任务写在它的run方法中;

```
    //添加新任务
    public void execute(Runnable command) {
        //如果任务为null直接抛出异常
        if (command == null)
            throw new NullPointerException();
        //获取当前线程池的ctl值，不知道它作用的看前面说明
        int c = ctl.get();

        //如果当前线程数小于核心线程数，这时候任务不会进入任务队列，会创建新的工作线程直接执行任务;
        if (workerCountOf(c) < corePoolSize) {
            //添加新的工作线程执行任务，addWorker方法后面分析
            if (addWorker(command, true))
                return;
            //addWorker操作返回false，说明添加新的工作线程失败，则获取当前线程池状态；（线程池数量小于corePoolSize情况下，创建新的工作线程失败，是因为线程池的状态发生了改变，已经处于非Running状态，或shutdown状态且任务队列为空）
            c = ctl.get();
        }

        //以下两种情况继续执行后面代码
        //1.前面的判断中，线程池中线程数小于核心线程数，并且创建新的工作线程失败;
        //2.前面的判断中，线程池中线程数大于等于核心线程数

        //线程池处于RUNNING状态，说明线程池中线程已经>=corePoolSize,这时候要将任务放入队列中，等待执行;
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            //再次检查线程池的状态，如果线程池状态变了，非RUNNING状态下不会接收新的任务，需要将任务移除，成功从队列中删除任务，则执行reject方法处理任务;
            if (! isRunning(recheck) && remove(command))
                reject(command);
```

```
程（这时候新任务已经在队列中，所以下面创建worker线程时第一个参数，要执行的任务为null，只是创建一个新的工作线程并启动它，让它自己去队列中取任务执行）
                //2.线程池处于非RUNNING状态但是任务移除失败,导致任务队列中仍然有任务，但是线程池中的线程数为0，则创建新的工作线程，处理队列中的任务；
                    addWorker(null, false);
            // 两种情况执行下面分支：
            // 1.非RUNNING状态拒绝新的任务,并且无法创建新的线程，则拒绝任务
            // 2.线程池处于RUN                状态，线程池线程数量已经大于等于coresize，任务就需要放入队列，如果任务入队失败，说明队列满了，                建新的线程，创建成功则新线程继续执行任务，如果创建失败说明线程池中线程数已经超过maximumPoolSize，则拒绝任务
            }else if (!addWo                command, false))
                reject(comma
        }
```

往线程池添加线程addWor 方法

往线程池中添加工作线程，线程会被封装成Worker对象，放入到works线程池中（可以先看下一小节“内部类Worker”的实现后再看这个方法，也可以先不用管Worker类，先看addWorker的实现过程）；
它的执行过程如下：

- 增加线程时，先判断当前线程池的状态允不允许创建新的线程，如果允许再判断线程池有没有达到 限制，如果条件都满足，才继续执行；
- 先增加线程数计数ctl，增加计数成功后，才会去创建线程；
- 创建线程是通过work对象来创建的，创建成功后，将work对象放入到works线程池中（就是一个hashSet）；
- 添加完成后，更新largestPoolSize值（线程池中创建过的线程最大数量），最后启动线程，如果参数firstTask不为null，则执行第一个要执行的任务，然后循环去任务队列中取任务来执行；

成功添加worker工作线程需要线程池处于以下两种状态中的一种

1. 线程池处于RUNNING状态
2. 线程池处于SHUTDOWN状态，且创建线程的时候没有传入新的任务（此状态下不接收新任务），且任务队列不为空（此状态下，要执行完任务队列中的剩余任务才能关闭）；

```

    private boolean addWorker(Runnable firstTask, boolean core) {
        //以下for循环，增加线程数计数，ctl，只增加计数，不增加线程，只有增加计数成功，才会增加线程
        retry:
        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);
            //这个代码块的判断，如果是STOP，TIDYING和TERMINATED这三种状态，都会返回false。(这几种状态不会接收新任务，也不再执行队列中的任务，中断当前执行的任务)
            //如果是SHUTDOWN，firstTask不为空（SHUTDOWN状态下，不会接收新任务）或 者workQueue是空（队列里面都没有任务了，也就不需要线程了），返回false。
            if (rs >= SHUTDOWN &&
                ! (rs == SHUTDOWN &&
                   firstTask == null &&
                   ! workQueue.isEmpty()))
                return false;
            //只有满足以下两种条件才会继续创建worker线程对象
            //1.RUNNING状态，
            //2.shutdown状态，且firstTask为null（因为shutdown状态下不再接收新任务），队列不是空（shutdown状态下需要继续处理队列中的任务）
             通过自旋的方式增加线程池线程数
            for (;;) {
                int wc = workerCountOf(c);
                //1.如果线程数大于最大可创建的线程数CAPACITY，直接返回false；
                //2.判断当前是要根据corePoolSize，还是maximumPoolSize进行创建线程（corePoolSize是基本线程池大小，未达到corePoolSize前按照corePollSize来限制线程池大小,达到corePoolSize后，并且任务队列也满了，才会按照maximumPoolSize限制线程池大小）
                if (wc >= CAPACITY ||
```

```
                if (compareAndIncrementWorkerCount(c))//将WorkerCount通过CAS操作增加
1，成功的话直接跳出两层循环；
                    break retry;
                c = ctl.get();  // Re-read ctl
                if (runStateOf(c) != rs)//否则则判断当前线程池的状态，如果现在获取到的状态与
进入自旋的状态不一致的话，那么则通过continue retry重新进行状态的判断
                    continue retry;
                // else CAS failed due to workerCount change; retry inner loop
            }
        }

        //以下代码块是创建Worker线程对象，并启动

        boolean workerStarted = false;
        boolean workerAdded = false;
        Worker w = null;
        try {
            w = new Worker(firstTask); //创建一个新的Worker对象
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock(); //获取线程池的重入锁后，
                try {
                    // Recheck while holding lock.
                    // Back out on ThreadFactory failure or if
                    // shut down before lock acquired.
                    int rs = runStateOf(ctl.get());

                    // RUNNING状态 || SHUTDONW状态下，没有新的任务，只是处理任务队列中剩余的
任务；
                    if (rs < SHUTDOWN ||
                        (rs == SHUTDOWN && firstTask == null)) {
                        //如果线程是活动状态，直接抛出异常，因为线程刚创建，还没有执行start方
法，一定不会是活动状态；
                        if (t.isAlive())
                            throw new IllegalThreadStateException();
                        // 将新启动的线程添加到线程池中
                        workers.add(w);
                        // 更新largestPoolSize的值，largestPoolSize成员变量保存线程池中创
建过的线程最大数量
                        int s = workers.size();
                        //将线程池中创建过的线程最大数量，设置给largestPoolSize，可以通过get
LargestPoolSize()方法获取，注意这个方法只能在 ThreadPoolExecutor中调用，Executer，Executer
Service，AbstractExecutorService中都是没有这个方法的
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                        workerAdded = true;
                    }
                } finally {
                    mainLock.unlock();
                }
                // 启动新添加的线程，这个线程首先执行firstTask，然后不停的从队列中取任务执行
                // 当等待keepAlieTime还没有任务执行则该线程结束。见runWoker和getTask方法的
代码。
                if (workerAdded) {
                    t.start();
                    workerStarted = true;
                }
            }
        }finally {
            if (! workerStarted)
                addWorkerFailed(w);
            }
            return workerStarted;
        }
    }
```

内部类Worker

```
private final class Worker  extends AbstractQueuedSynchronizer implements Runnable
```

由它的定义可以知它实现了Runnable接口，是一个线程，还继承了AQS类，实现了加锁机制；

它利用AQS框架实现了一个简单的非重入的互斥锁， 实现互斥锁主要目的是为了中断的时候判断线程是在空闲还是运行，它的state只有三个值 ，初始状态为不可加锁状态-1，无锁状态为0，加锁状态为1，可以看shutdown、shutdownNow、runWorker方法对它锁的作用。

Worker的构造方法

构造方法里面要重点关注一下getThr ctory()这个方法

```
        //参数为Worker线程 第一个要执行的任务
        Worker(Runnable firstTask) {
            //设置ASQ的st -1  设置worker处于不可加锁的状态，看后面的tryAcquire方法，只有state为0时才允许加锁，worker线程运行以后才会把state置为0
            setState(-1);
            //设置第一个运行的任务
            this.firstTask = firstTask;
            //创建线程，将this自己传入进去；getThreadFactory()见后面详解
            this.thread = getThreadFactory().newThread(this);
        }
```

线程的创建getThreadFactory();

默认会在构造方法中传入Executors.defaultThreadFactory()，该方法然会一个DefaultThreadFactory();

```
public static ThreadFactory defaultThreadFactory() {
        return new DefaultThreadFactory();
}
```

```
 static class DefaultThreadFactory implements ThreadFactory {
        //线程池编号
        private static final AtomicInteger poolNumber = new AtomicInteger(1);
        //线程池中线程所属线程组
        private final ThreadGroup group;
        //线程池中线程编号
        private final AtomicInteger threadNumber = new AtomicInteger(1);
        //线程名称前缀
        private final String namePrefix;

        DefaultThreadFactory() {
            SecurityManager s = System.getSecurityManager();
            group = (s != null) ? s.getThreadGroup() :
                                  Thread.currentThread().getThreadGroup();
            //设置线程名称为"pool-线程池的编号-thread-线程的编号"
            namePrefix = "pool-" +
                         poolNumber.getAndIncrement() +
                        "-thread-";
        }

        //创建新的线程
        public Thread newThread(Runnable r) {
            Thread t = new Thread(group, r,
                                  namePrefix + threadNumber.getAndIncrement(),
                                  0);
            //设置为非守护线程
            if (t.isDaemon())
                t.setDaemon(false);
            //设置优先级为NORMAL为5
            if (t.getPriority() != Thread.NORM_PRIORITY)
                t.setPriority(Thread.NORM_PRIORITY);
            return t;
        }
    }
```

```
//被封装的线程，就是它自己；
final Thread thread;
//传入的它要执行的第一个任务，如果firstTask为空就从任务队列中取任务执行
Runnable firstTask;
//记录执行完成的任务数量，如果执行任务过程中出现异常，仍然会计数；
volatile long completedTasks
```

worker线程的加锁解锁

worker的加锁解锁机制是基于AQS框架的，要完全弄明白它的加锁解锁机制请看AQS框架的实现，在这里只是简单介绍一下：

```
        //尝试加锁方法，将state设置为1；如果不是0则加锁失败,在worker线程没有启动前是-1状态，无法加锁
        //该方法重写了父类AQS的同名方法
        protected boolean tryAcquire(int unused) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        //尝试释放锁的方法，直接将state置为0
        //该方法重写了父类AQS的同名方法
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }
        //注意：tryAcquire与tryRelease是重写了AQS父类的方法，且不可以直接调用，它们被以下方法调用实现加锁解锁操作

        //加锁:acquire法是它父类AQS类的方法,会调用tryAcquire方法加锁
        public void lock()        { acquire(1); }
        //尝试加锁
        public boolean tryLock()  { return tryAcquire(1); }
        //解锁:release方法是它父类AQS类的方法，会调用tryRelease方法
        public void unlock()      { release(1); }
        //返回锁状态
        public boolean isLocked() { return isHeldExclusively(); }
```

Worker线程执行任务runWorker （重要）

看完了Worker线程的创建，再来看看Worker线程的运行，Worker的run方法中会调用runWorker方法来获循环取任务并执行；

```
   final void runWorker(Worker w) {
        //当前线程
        Thread wt = Thread.currentThread();
        //获取当前Worker线程创建时，指定的第一个要执行的任务，也可以不指定任务，那么它自己就会去任务队列中取任务；
        Runnable task = w.firstTask;
        w.firstTask = null;
        // 在构造方法里面将state设置为了-1，执行该方法就将state置为了0，这样就可以加锁了，-1状态下是无法加锁的，看Worker类的tryAcquire方法
        w.unlock();
        //该变量代表任务执行是否发生异常，默认值为true发生了异常，后面会用到这个变量
        boolean completedAbruptly = true;
        try {
            //如果创建worker时传入了第一个任务，则执行第一个任务，否则 从任务队列中获取任务getTask(), getTask()后面分析；
            while (task != null || (task = getTask()) != null) {
                //线程加锁
                w.lock();
                /**
                 * 先判断线程池状态是否允许继续执行任务：
```

```
                 * 2.shutingdown, runing , 处于中断状态（并复位中断标志），如果这个时候其它线
程执行了shutdownNow方法，shutdownNow方法会把状态设置为STOP
                 *
                 * 这个时候则中断线程
                 **/
                if ((
                        runStateAtLeast(ctl.get(), STOP) ||
                        Thread.interrupted() && runStateAtLeast(ctl.get(), STOP)
                    )
                    &&
                    !wt.isInterrupted())
                    wt.interrupt();

                /**
                 *开始执行任务
                 */

                try {
                    //任务执行前要做的处理：这个方法是空的，什么都不做，一般会通过继承ThreadPo
olExecute类后重写该方法实现自己的功能；传入参数为当前线程与要执行的任务
                    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 {
                        //任务执行后要做的处理：这个方法也是空的，什么都不做,一般会通过继承Thr
eadPoolExecute类后重写该方法实现自己的功能；参数为当前任务和执行任务时抛出的异常
                        afterExecute(task, thrown);
                    }
                } finally {
                    task = null;
                    //增加完成任务计数
                    w.completedTasks++;
                    w.unlock();
                }
            }

            /**
             *退出while循环，线程结束；
             **/

            //判断task.run()方法是否抛出了异常，如果没有则设置它为false，如果发生了异常，前面会
直接抛出，中断方法继续执行，就不会执行下面这句；
            completedAbruptly = false;
        } finally {
            /**
             * 线程退出后的处理
             */
            processWorkerExit(w, completedAbruptly);
        }
    }
```

worker线程从任务队列里面获取任务getTask

从任务队列中获取任务

```
这是个for循环
1.先判断线程池状态是否允许取任务，不允许直接将线程数量减1 ，直接返回null；
```

3.若没有超时，则去任务队列取任务，取到的话返回任务，若超时则设置超时状态，继续循环，在下次循环中处理超时状态

```
    private Runnable getTask() {
        // 如果判断当前线程池状态需要启用超时操作，那么任务队列取任务时使用的是带有超时的workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS)方法，如果超时，则会将timeOut 变量设置为true，在下次执行for循环时根据timeOut执行超时操作；
        boolean timedOut = false;

        for (;;) {
            int c = ctl.get();
            int rs = runStateOf(c);
            /**
            * 以下分支在state为shutdown、stop、tidying、terminated状态，或者在SHUTDOWN状态且任务队列为空时退出当前线程
            *
            * 判断线程池状态是否允许继续获取任务：
            * RUNNING<shutdown<stop<tidying<terminated;
            * rs >= SHUTDOWN，包含两部分判断操作
            *1.如果是rs > SHUTDOWN，即状态为stop、tidying、terminated；这时不再处理队列中的任务，直接返回null
            *2.如果是rs = SHUTDOWN , rs>=STOP不成立，这时还需要处理队列中的任务除非队列为空，没有任务要处理，则返回null
            */
            // Check if queue empty only if necessary.
            if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                //自旋锁将ctl减1（也就是将线程池中的线程数减1）
                decrementWorkerCount();
                return null;
            }
            /**
             * 在RUNNING状态 或 shutdown状态且任务队列不为空时继续往下执行执行
             */

            /**
             *  以下做线程超时控制：
             * 启用超时控制需要满足至少一个条件
             * 1.allowCoreThreadTimeOut为true代表核心线程数可以做超时控制；
             * 2.如果当前线程数>corePoolSize核心线程数,也可以做超时控制；
             * 在以上前提下，再判断当前线程是否需要销毁：
             * 1.如果当前线程数大于maximumPoolSize，这肯定是不允许的，需要销毁当前线程；
             * 2.如果当前线程上次执行循环时，取任务操作超时，任务队列是空，需要销毁当前线程；
             */

            //获取线程池中线程数量
            int wc = workerCountOf(c);

            // timed变量用于判断是否需要进行超时控制。
            // allowCoreThreadTimeOut默认是false，也就是核心线程不允许进行超时；
            // wc > corePoolSize，表示当前线程池中的线程数量大于核心线程数量；
            // 对于超过核心线程数量的这些线程，需要进行超时控制；
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

            /*
            * 超时销毁线程需要先满足以下两个条件之一
            * 1. wc > maximumPoolSize的情况是因为可能在此方法执行阶段同时执行了setMaximumPoolSize方法；
            * 2. timed && timedOut 如果为true，表示当前操作需要进行超时控制，并且上次循环当前线程从任务队列中获取任务发生了超时，没有取到任务；
            *  满足上面两个条件之一的情况下，接下来判断，如果线程数量大于1，或者线程队列是空的，那么尝试将workerCount减1，减1成功则返回null，退出当前线程； 如果减1失败，则返回继续执行循环操作，重试。
            */
            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                //尝试将线程池线程数量减一
                if (compareAndDecrementWorkerCount(c))
```

```
                continue;
            }


            /**
             * 如果没有超时，则继续去任务队列取任务执行；
             *取任务操作
             */
            try {
                //根据timed（是否启用超时控制）来判断执行poll操作还是执行take()操作还是执行有
时间限制的poll操作，并返回获 任务；
                Runnable  timed ?
                    work .poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    work .take();
                if (r != null)
                    retu
                //如果poll操作等待超时,没有取到任务；则将timeOut设置为true；
                timedOut = true;
            } catch (InterruptedException retry) {
                //如果是因为线程中断导致没有取到任务；则设置timedOut=false继续执行循环，取任务
                timedOut = false;
            }
        }
    }
```

Worker线程的退出processWorkerExit

如果是处理任务发生异常导致的退出，则以自旋锁的方式将线程数减1；
将当前worker执行完成的任务数，累加到completedTaskCount上；
将当前线程移出线程池；
尝试终止线程池；
判断是否要新建workder线程；
1.如果是RUNNING或SHUTDOWN状态，且worker是异常结束，会直接执行AddWorker操作；
2.如果是RUNNING或SHUTDOWN状态，且worker是没有任务可做结束的，且allowCoreThreadTimeOut=false，且当前线程池中的线程数小于corePoolSize，则会创建addWorker线程；
3.判断是否要添加一个新的线程：线程池是RUNNING或SHUTDOWN状态，worker线程如果是异常结束的，则直接添加一个新线程；如果当前线程池中的线程数小于最小线程数，也会创建一个新线程；

```
 private void processWorkerExit(Worker w, boolean completedAbruptly) {
         // 如果任务运行异常导致则completedAbruptly=true，则将线程池worker线程数减1，如果是
没有获取到任务导致的completedAbruptly=false，则会在getTask()方法里面将线程数减1；
        if (completedAbruptly)
            //自旋锁将ctl减1（也就是将线程池中的线程数减1）
            decrementWorkerCount();

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            //退出前，将本线程已完成的任务数量，添加到已经完成任务的总数中；
            completedTaskCount += w.completedTasks;
            //线程队列中移除当前线程
            workers.remove(w);
        } finally {
            mainLock.unlock();
        }

        //尝试停止线程池
        tryTerminate();

       /**
        *判断是否要增加新的线程
        *如果满足以下条件则新增线程：
        * 一、当线程池是RUNNING或SHUTDOWN状态，且worker是异常结束，那么会直接addWorker；
        * 二、当线程池是RUNNING或SHUTDOWN状态，且worker是没有任务可做结束的；
        *    1.如果allowCoreThreadTimeOut=true，则判断等待队列不为空  ，且当前线程数是否小于
1；
```

```
        int c = ctl.get();
        //当线程池是RUNNING或SHUTDOWN状态，
        if (runStateLessThan(c, STOP)) {
            //如果非异常状况completedAbruptly=false，也就是没有获取到可执行的任务，则获取线程池允许的最小线程数，如果allowCoreThreadTimeOut为true说明允许核心线程超时，则最小线程数为0，否则最小线程数为corePoolSize;
            if (!completedAbruptly) {
                int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                //如果allowCoreThreadTimeOut=true，且任务队列有任务要执行，则将最最小线程数设置为1
                if (min == 0 && ! workQueue.isEmpty())
                    min = 1;
                //如果当前线程数大于等于最小线程数，则直接返回
                if (workerCountOf(c) >= min)
                    return; // replacement not needed
            }
            //以下两种情况会添加一个新的线程
            //1.worker是异常结束；
            //2.如果是非异常结束，且任务队列里面还有任务，
            addWorker(null, false);
        }
    }
```

线程池的关闭

线程池的关闭有两个方法shutdown() 与shutdownNow()；

shutdown会将线程池状态设置为SHUTDOWN状态，然后中断所有空闲线程，然后执行tryTerminate()方法（tryTerminate这个方法很重要，会在后面分析），来尝试终止线程池;

shutdownNow会将线程池状态设置为STOP状态，然后中断所有线程（不管有没有执行任务都设置为中断状态），然后执行tryTerminate()方法，来尝试终止线程池;

```
public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            // 线程池状态设为SHUTDOWN，如果已经是shutdown<stop<tidying<terminated，也就是非RUNING状态则直接返回
            advanceRunState(SHUTDOWN);
            // 中断空闲的没有执行任务的线程
            interruptIdleWorkers();
            onShutdown(); //空方法，子类覆盖实现
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }

    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            // STOP状态：不再接受新任务且不再执行队列中的任务。
            advanceRunState(STOP);
            // 中断所有线程，无论空闲还是在执行任务
            interruptWorkers();
            // 将任务队列清空，并返回队列中还没有被执行的任务。
            tasks = drainQueue();
        }finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }
```

这两个方法可以直接调用，来关闭线程池；shutdown方法还会在线程池被垃圾回收时调用,因为ThreadPoolExecuter重写了finalize方法

```
protected void finalize() {
        shutdown();
}
```

关于finalize方法说明：
垃圾回收时，如果判断对象不可达，且覆盖了finalize方法，则会将对象放入到F-Queue队列，有一个名为"Finalizer"的守护线程执行finalize方法，它先级为8，做最后的清理工作，执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象复活
注意：网上很多人说，Finalizer线程先级低，个人认为这是不对的，Finalizer线程在jdk1.8的优先级是8，比我们创建线程默认优先级5要高，之前其它版本的jdk我记得导出的线程栈信息里面优先级是5，忘记是哪个版本的jdk了，即使是5优先级也不比自己线程默认优先级低，总之我没见过优先级低于5的Finalizer线程；
这个线程会不停的循环等待java.lang.ref.Finalizer.ReferenceQueue中的新增对象。一旦Finalizer线程发现队列中出现了新的对象，它会弹出该对象，调用它的finalize()方法，将该引用从Finalizer类中移除，因此下次GC再执行的时候，这个Finalizer实例以及它引用的那个对象就可以回垃圾回收掉了。
大多数时候，Finalizer线程能够赶在下次GC带来更多的Finalizer对象前清空这个队列,但是当它的处理速度没法赶上新对象创建的速度，对象创建的速度要比Finalizer线程调用finalize()结束它们的速度要快，这导致最后堆中所有可用的空间都被耗尽了；
当我们大量线程频繁创建重写了finalizer（）方法的对象的情况下，高并发情况下，它可能会导致你内存的溢出；虽然Finalizer线程优先级高，但是毕竟它只有一个线程；最典型的例子就是数据库连接池,proxool，对要释放资源的操作加了锁，并在finalized方法中调用该加锁方法，在高并发情况下，锁竞争严重，finalized竞争到锁的几率减少，finalized无法立即释放资源，越来越多的对象finalized()方法无法被执行，资源无法被回收，最终导致导致oom；所以覆盖finalized方法，执行一定要快，不能有锁竞争的操作，否则在高并发下死的很惨；
（proxool使用了cglib，它用WrappedConnection代理实际的Conneciton。在运行WrappedConnection的方法时，包括其finalize方法，都会调用Conneciton.isClosed()方法去判断是否真的需要执行某些操作。不幸的是JDBC中的这个方法是同步的，锁是连接对象本身。于是， Finalizer线程回收刚执行过的WrappedConnection对象时就总会与还在使用Connection的各个工作线程争用锁。）

线程池中线程的中断

线程池的中断也有两个方法
interruptIdleWorkers 中断没有执行任务的线程；
interruptWorkers 中断所有线程，不管线程有没有执行任务；

```
//中断空闲线程，没有执行任务的线程会被中断，onlyOne参数用来标识是否只中断一个线程；
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        //遍历所有的Worker线程
        for (Worker w : workers) {
            Thread t = w.thread;
            //如果线程没有被中断，w.tryLock()会调用tryAcquire()方法尝试加锁，加锁成功后会中断线程
            //为什么要w.tryLock(),因为在runWorker()方法的while循环执行任务之前会加锁，如果已经被加锁说明线程正在执行任务，不能被中断；
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                    //中断线程
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            //如果 onlyOne为true， for循环只执行一次就退出
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock();
```

```
    /****
     * 中断所有正在运行的线程，注意，这里与interruptIdelWorkers()方法不同的是，没有使用worker
的AQS锁
     */
    private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            for (Worker w : workers)
                w.interruptIfStarted();
        } finally {
            mainLock.unlock();
        }
    }
```

尝试终止线程池tryTerminate

该方法会在很多地方调用，如添加worker线程失败的addWorkerFailed（)方法，worker线程跳出执行任务的while 循环退出时的processWorkerExit（）方法，关闭线程池的shutdown()和shutdownNow()方法，从任务队列移除任务的remove()方法；

该方法的作用是检测当前线程池的状态是否可以将线程池终止，如果可以终止则尝试着去终止线程，否则直接返回；

STOP-》TIDYING 与SHUTDOWN-》TIDYING状态的转换，就是在该方法中实现的，最终执行terminated()方法后会把线程状态设置为TERMINATED的状态；

尝试终止线程池执行过程；

一、重点内容先判断线程池的状态是否允许被终止

以下状态不可被终止：

```
1.如果线程池的状态是RUNNING（不可终止）
   或者是TIDYING（该状态一定执行过了tryTerminate方法，正在执行或即将执行terminated()方法，所以
不需要重复执行），
   或者是TERMINATED（该状态已经执行完成terminated()钩子方法，已经是被终止状态了），
   以上三种状态直接返回。
2.如果线程池状态是SHUTDOWN，而且任务队列不是空的（该状态需要继续处理任务队列中的任务，不可被终
止），也直接返回。
```

以下两种状态线程池可以被终止：

```
1.如果线程池状态是SHUTDOWN，而且任务队列是空的（shutdown状态下，任务队列为空，可以被终止），向下
进行。
2.如果线程池状态是STOP（该状态下，不接收新任务，不执行任务队列中的任务，并中断正在执行中的线程，可
以被终止），向下进行。
```

二、线程池状态可以被终止，如果线程池中仍然有线程，则尝试中断线程池中的线程

则尝试中断一个线程然后返回,被中断的这个线程执行完成退出后，又会调用tryTerminate()方法，中断其它线程，直到线程池中的线程数为0，则继续往下执行；

三、如果线程池中的线程为0，则将状态设置为TIDYING，设置成功后执行 terminated()方法，最后将线程状态设置为TERMINATED

源码如下：

```
    final void tryTerminate() {
        for (;;) {
            int c = ctl.get();
            //先判断是否满足终止线程池的条件
            //1.如果线程池的状态是RUNNING（不可终止）或者是TIDYING（该状态的线程池即将要执行或
正在执行terminated()钩子方法），TERMINATED（该状态已经执行完成terminated()钩子方法），直接返
回。
            //2.如果线程池状态是SHUTDOWN，而且任务队列不是空的（该状态需要继续处理任务队列中的任
务，不可被终止），也直接返回。
```

```
                return;

            //以下状态才会继续执行：
            //1.如果线程池状态是SHUTDOWN，而且任务队列是空的（shutdown状态下，任务队列为空，可以被终止），向下进行。
            //2.如果线程池状态是STOP（该状态下，不接收新任务，不执行任务队列中的任务，并中断正在执行中的线程，可以被终止），向下进行。

            // workerCou                0则还不能停止线程池,而且这时线程都处于空闲等待的状态
            // 需要中断让线                "过来，醒过来的线程才能继续处理shutdown的信号。
            if (workerCo                (c) != 0) { // Eligible to terminate
                // runWo                法中w.unlock就是为了可以被中断,getTask方法也处理了中断。
                // ONLY_                这里只需要中断1个线程去处理shutdown信号就可以了。
                interrup            Workers(ONLY_ONE);
                return;
            }

            //满足以下两个条件才会继续执行
            //1.线程池状态是STOP且 工作线程池中的线程wc是0
            //2.线程池状态是SHUTDOWN而且工作线程池wc(pool)和任务队列(queue)都是空的
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                //进入TIDYING状态，线程池的状态被原子操作ctl.compareAndSet(c, ctlOf(TIDYING, 0)将状态设置为TIDYING，(因为tryTerminate方法会在多处调用，存在竞争)
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        terminated();
                    } finally {
                        //进入TERMINATED状态
                        //进一步在terminated结束之后的finally块中通过ctl.set(ctlOf(TERMINATED, 0))设置为TERMINATED。
                        ctl.set(ctlOf(TERMINATED, 0));
                        termination.signalAll();  //最后执行termination.signalAll(),会唤醒awaitTermination方法中由于执行termination.awaitNanos(nanos)操作进入等待状态的线程
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }
```

未完待续

进阶篇:定时任务执行之ScheduledThreadPoolExecuter(十六)

定时任务这个恐怕很多时候我们都需要用到吧... 比如我们想间隔一天后执行某个定义好的任务，又或者间隔一天后执行完某个任务后，再每间1小时执行一次...当我们有这种需求的时候，ScheduledThre...

cy_Alone 2017年04月13日 22:54 1346

ThreadPoolExecutor 原理机制

paul_wei2008 2015年11月21日 13:05 1037