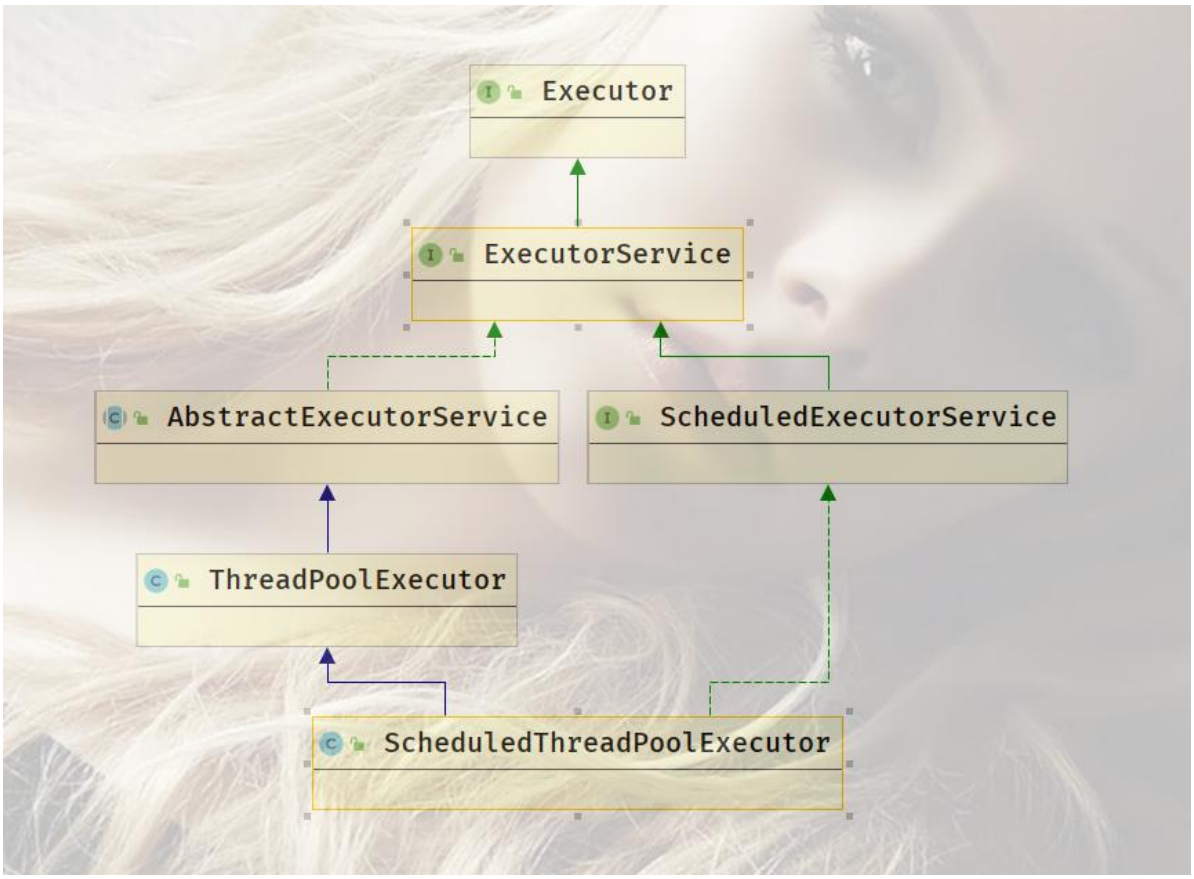
一，线程池的基本使用

1，线程池的介绍

线程复用，控制最大并发数，管理线程。

优点：提高响应速度，避免每次都去创建线程。便于管理，降低资源消耗。



2，常用的线程池

`1.Executors.newFixedThreadPool();`
执行长期任务性能好，创建一个线程池，一池有N个固定的线程，有固定的线程数的线程
`2.Executors.newSingleThreadExecutor();`
一个任务一个任务的执行，一池一线程
`3.Executors.newCachedThreadPool();`
执行很多短期异步任务，线程池根据需要创建新线程，但在先前构建的线程可用时将重用它们。可扩容，遇强则强。

线程池的核心其实都是同一个类ThreadPoolExecutor。

注意

线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，规避资源消耗。
说明：Executors返回的线程池对象的弊端如下：
`1）FixedThreadPool和SingleThreadPool；`
允许的请求队列长度为Integer的最大值，可能会堆积大量的请求，从而导致OOM。
`2）CachedThreadPool和ScheduledThreadPool`
允许的创建线程数量为Integer最大值，可能会创建大量的线程，从而导致OOM。

```
    public static void main(String[] args) {
        ExecutorService pool1 = Executors.newFixedThreadPool(5);//一个银行网点5个受理业务窗口
        ExecutorService pool2 = Executors.newSingleThreadExecutor();//一个银行网点1个受理业务窗口
        ExecutorService pool3 = Executors.newCachedThreadPool();//一个银行网点n个受理业务窗口
        //3个顾客
        try {
            for (int i = 0; i < 30; i++) {
                pool3.execute(()->{
                    System.out.println(Thread.currentThread().getName()+"\t线程办理业务！");
                });
                //pool1.submit(()->{});
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            pool3.shutdown();
        }
    }
```

3，ThreadPoolExecutor 的7大参数

`int corePoolSize`，线程池中的常驻核心线程数

`int maximumPoolSize`，能够容纳同时执行的最大线程数，必须大于1

`long keepAliveTime`，多余空闲线程存活时间

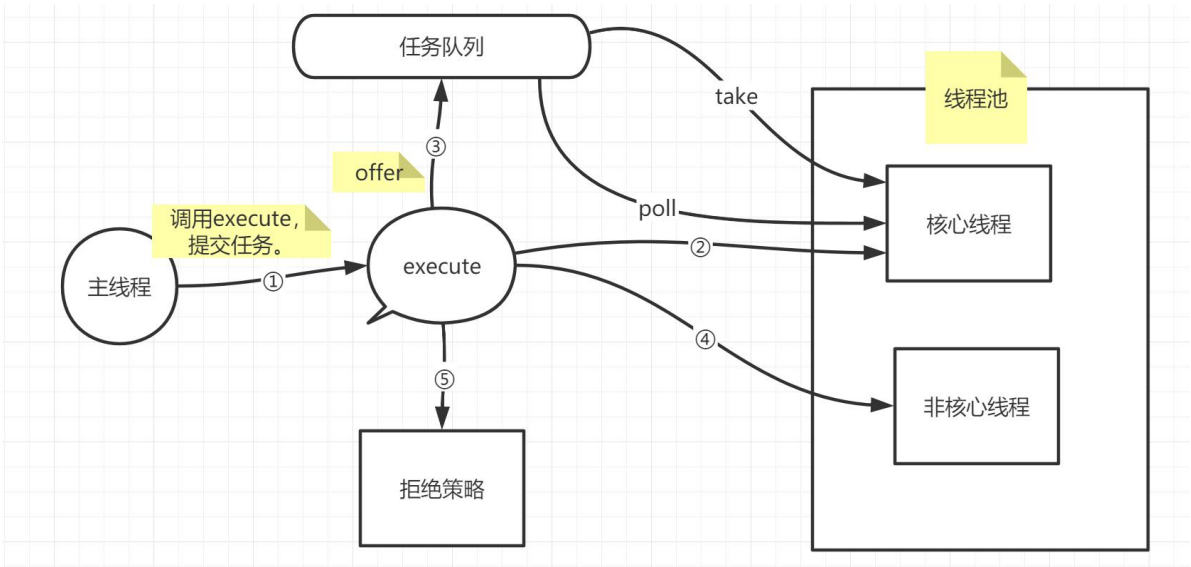
`TimeUnit unit`，上个参数的单位

`BlockingQueue<Runnable> workQueue`，任务队列，被提交但是尚未执行的任务

`ThreadFactory threadFactory`，表示生成线程池中工作线程的线程工厂，用于创建线程，一般默认

`RejectedExecutionHandler handler`，拒绝策略，表示当队列满了，并且工作线程大于等于线程池的最大连接数时，如何来拒绝请求执行的runnable的策略。

4，线程池的工作原理



1、在创建了线程池后，线程池中的线程数为零。

2、当调用execute ()方法添加一个请求任务时，线程池会做出如下判断:

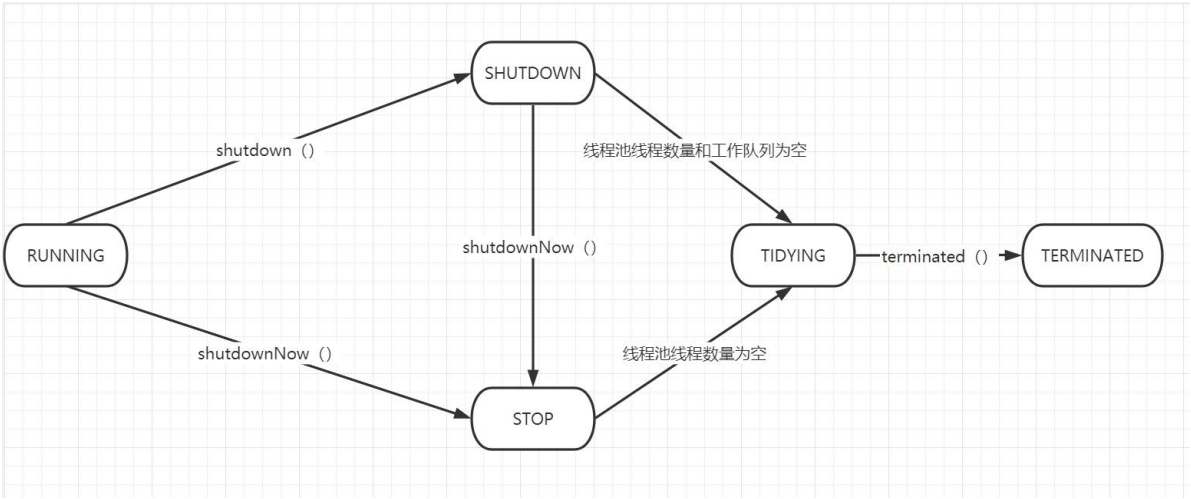
2.1如果正在运行的线程数量小于corePoolSize,那么马上创建线程运行这个任务;
2.2如果正在运行的线程数量大于或等于corePoolSize,那么将这个任务放入队列;
2.3如果这个时候队列满了且正在运行的线程数量还小于maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务;
2.4如果队列满了且正在运行的线程数量大于或等于max imumPoolSize,那么线程池会启动饱和拒绝策略来执行。

3、 当一个线程完成任务时，它会从队列中取下一个任务来执行。

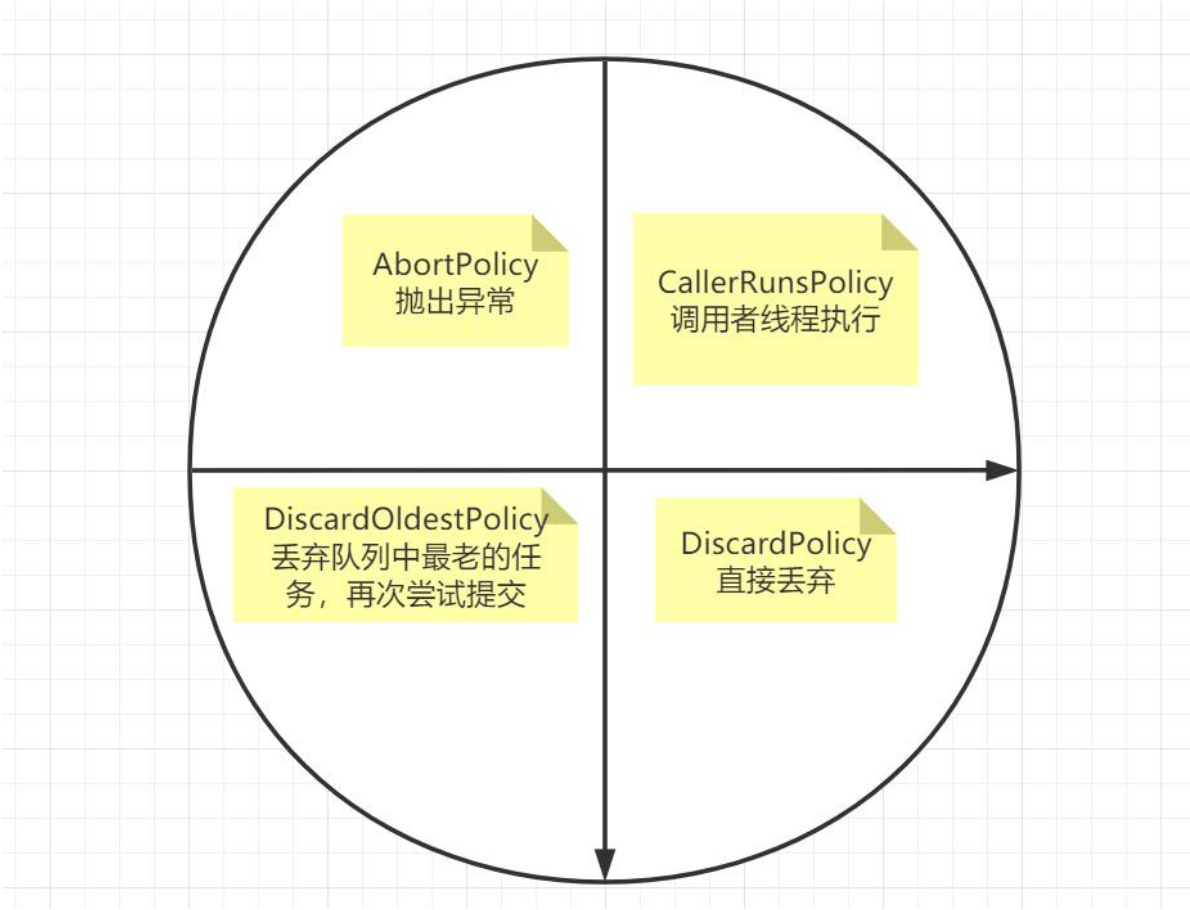
4、当一个线程无事可做超过一定的时间(keepAliveTime) 时，线程会判断:

如果当前运行的线程数大于corePoolSize.那么这个线程就被停掉。 所以线程池的所有任务完成后，它最终会收缩到corePoolSize的大小。

1）线程池的状态



2）线程池的拒绝策略



5，自定义线程池

```
    public static void main(String[] args) {
        ExecutorService pool = new ThreadPoolExecutor(
                2,
                5,
                3l,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(3),
                Executors.defaultThreadFactory(),
                //new ThreadPoolExecutor.AbortPolicy()   //抛异常
                //new ThreadPoolExecutor.CallerRunsPolicy()  //main 线程办理业务！
                //new ThreadPoolExecutor.DiscardOldestPolicy()  //就处理能处理的，剩下的老的直接丢了。
                new ThreadPoolExecutor.DiscardPolicy()  //如果新来的处理不了，直接就扔了。
        );
        try {
            for (int i = 0; i < 30; i++) {
                pool.execute(() -> {
                    System.out.println(Thread.currentThread().getName() + "\t线程办理业务！");
                });
                //pool1.submit(()->{});
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            pool.shutdown();
        }
    }
```

二，线程池源码分析

1，成员属性/静态属性/构造方法

```
//高三位：表示当前线程池运行状态  除去高三位之后的低位：表示当前线程池中所拥有的的线程数量
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
//表示在ctl中，低COUNT_BITS位适用于存放当前线程数量的位。
private static final int COUNT_BITS = Integer.SIZE - 3;
//线程池所能存放的最大容量
private static final int CAPACITY = (1 << COUNT_BITS) - 1;

//-1左移29位 负数  111
private static final int RUNNING    = -1 << COUNT_BITS;
// 0   000
private static final int SHUTDOWN   =  0 << COUNT_BITS;
//001
private static final int STOP       =  1 << COUNT_BITS;
//010
private static final int TIDYING    =  2 << COUNT_BITS;
//011
private static final int TERMINATED =  3 << COUNT_BITS;

// 获取当前线程池运行状态
private static int runStateOf(int c)     { return c & ~COUNT_MASK; }
//获取当前线程池线程数量
private static int workerCountOf(int c)  { return c & COUNT_MASK; }
//用在重置当前线程ctl值时，会用到  rs表示线程池状态  wc表示线程池worker数量
private static int ctlOf(int rs, int wc) { return rs | wc; }

//cas的方式让ctl值+1
private boolean compareAndIncrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect + 1);
}

//cas的方式让ctl-1
private boolean compareAndDecrementWorkerCount(int expect) {
    return ctl.compareAndSet(expect, expect - 1);
}

//将ctl值-1
private void decrementWorkerCount() {
    ctl.addAndGet(-1);
}
//当线程池中的线程达到核心线程数时，在提交任务，就会提交到任务队列
private final BlockingQueue<Runnable> workQueue;

//线程池中的全局锁，增加worker，减少worker，修改线程池运行状态
private final ReentrantLock mainLock = new ReentrantLock();

//真正存放worker->thread的地方
private final HashSet<Worker> workers = new HashSet<>();

//当外部线程调用awaitTermination()时，外部线程会等待当前线程池状态为Termination为止
```

```
    //等待是如何实现的？将外部线程封装成waitNode放入到condition队列中，外部线程会被park，处于WAITTING状态
    //当线程状态变为Termination时，会唤醒这些线程，通过Termination。signallAll() ，唤醒之后，这些线程会进入到等待队列，然后头结点回去抢占mainLock，
    //抢占到的线程会继续执行awaitTermination()后面的程序。这些线程最后都会正常执行了。
    private final Condition termination = mainLock.newCondition();
    //记录线程池生命周期内，线程最大值
    private int largestPoolSize;

    //记录线程池所完成的任务总数，当worker退出时，会将worker完成的任务累积到这里
    private long completedTaskCount;

    //创建线程池会使用到线程工厂，当我们使用Executor.newFix ... /newCache ... 使用的DefaultThreadFactory，生成的线程名不容易分析执行的是哪里的业务
    //一般不建议使用使用自带的，推荐自己实现这个接口
    private volatile ThreadFactory threadFactory;

    //拒绝策略，默认是采用抛出异常
    private volatile RejectedExecutionHandler handler;

    //空闲线程存活时间 ：allowCoreThreadTimeOut=false时，会维护核心线程数量内的线程存活，超出部分会超时。
    //allowCoreThreadTimeOut=true时，核心数量内的线程也会被回收。
    private volatile long keepAliveTime;

    //控制核心线程是否可以被回收，true可以，false不可以。
    private volatile boolean allowCoreThreadTimeOut;

    //核心线程数限制
    private volatile int corePoolSize;

    //最大线程数限制
    private volatile int maximumPoolSize;

    //默认的拒绝策略，抛异常的方式
    private static final RejectedExecutionHandler defaultHandler =
        new AbortPolicy();

        public ThreadPoolExecutor(int corePoolSize, //核心线程数
                                  int maximumPoolSize, //最大线程数
                                  long keepAliveTime, //空闲等待时间
                                  TimeUnit unit,    //时间单位
                                  BlockingQueue<Runnable> workQueue, //任务队列
                                  ThreadFactory threadFactory, //线程工厂
                                  RejectedExecutionHandler handler //拒绝策略
                                  ) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        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;
    }
```

2，内部类Worker

线程池中的线程实际上都是封装成了一个个Worker来执行。

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

        private static final long serialVersionUID = 6138294804551838833L;

        //worker内部封装的工作线程
        @SuppressWarnings("serial")
        final Thread thread;
        //假设firstTask不为空，当worker启动后（worker内部的线程启动后）会优先执行
firstTask，
        //当执行完firstTask后，会去queue中去获取下一个任务
        @SuppressWarnings("serial")
        Runnable firstTask;
        //记录当前worker所完成的任务数量
        volatile long completedTasks;

        //firstTask可以为空，启动后会到queue中获取
        Worker(Runnable firstTask) {
            setState(-1); // 设置aqs独占模式为 初始化中状态，不能被抢占锁
            this.firstTask = firstTask;
            this.thread = getThreadFactory().newThread(this);
            //使用线程工厂创建了一个线程，并且将当前worker指定为
            //runnable，并且说当前thread启动的时候，会以worker.run()为入口。
        }

        /**  */
        public void run() {
            //调用了threadPoolExecutor的方法
            runWorker(this);
        }


        //当前worker的独占锁是否被独占
        protected boolean isHeldExclusively() {
            return getState() != 0;
        }
        //尝试去占用worker的独占锁
        protected boolean tryAcquire(int unused) {
            if (compareAndSetState(0, 1)) {
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }
        //尝试去释放worker的独占锁
        protected boolean tryRelease(int unused) {
            setExclusiveOwnerThread(null);
            setState(0);
```

```
            return true;
        }

        public void lock()        { acquire(1); }
        public boolean tryLock()  { return tryAcquire(1); }
        public void unlock()      { release(1); }
        public boolean isLocked() { return isHeldExclusively(); }

        void interruptIfStarted() {
            Thread t;
            if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
                try {
                    t.interrupt();
                } catch (SecurityException ignore) {
                }
            }
        }
    }
```

3，execute()

当调用execute()来提交一个任务的时候，首先会判断如果当前线程数量小于核心线程数，此次提交任务会创建一个核心线程执行任务。如果提交失败，会继续判断：如果当前线程池处于RUNNING状态，尝试将task放入到workQueue中。如果还是失败，会继续判断：达到了最大线程数，所以执行拒绝策略。

```
    /*
        command可以是Runnable实现类，也可以是FutureTask
    */
    public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        //获取ctl最新值 高三位表示此线程池状态，低位表示当前线程池的线程数量
        int c = ctl.get();
        //如果当前线程数量小于核心线程数，此次提交任务会创建一个核心线程执行任务
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                //创建成功以后，会直接返回，addWorker()里面会启动新创建的worker，将
firstTask执行。
                return;
            c = ctl.get();//addWorker失败，执行到这里
            /*
                1.存在并发
                2.当前线程池的状态发生改变，线程池状态是非RUNNING，addWorker()一定会失败。
                （shutdown状态下，也有可能创建成功，前提是firstTask==null，且queue!
=null）
            */
        }
        /*
            执行到这里的情况：
            1.当前线程数已经达到核心线程数
            2.addWorker失败
        */
        //如果当前线程池处于RUNNING状态，尝试将task放入到workQueue中
        if (isRunning(c) && workQueue.offer(command)) {
            //执行到这里说明offer提交成功
```

```
            int recheck = ctl.get();
            //条件1：提交到队列后，线程池状态被外部给修改，此时，需要删除刚刚提交的任务
            //条件2：删除成功，说明任务还没执行，删除失败说明任务被核心线程执行了。
            if (! isRunning(recheck) && remove(command))
                reject(command);//拒绝策略
            /*
                来到这里的情况：
                1.当前线程是RUNNING
                2.当前线程非RUNNING，且移除任务失败

                担心线程池是RUNNING，但是线程池的线程数量为0
            */
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        /*
            执行到这里，有几种情况：
            1.offer失败（说明队列满了，如果未达到最大线程数，创建线程执行任务）
            2.当前线程池是非running状态（这个时候command! =null addworker一定返回
false）

        */

        else if (!addWorker(command, false))
            //达到了最大线程数，所以执行拒绝策略
            reject(command);
    }
```

4，addWorker()

自旋去申请一个工作线程，申请成功以后，将线程包装成一个worker，加入到worker队列中，并启动任务。

```
    //firstTask可以为空，自动取queue拿任务， core：采用的线程数限制，true，核心线程数，
false，非核心线程
    private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (int c = ctl.get();;) {//获取当前ctl保存到c

            if (runStateAtLeast(c, SHUTDOWN)//当前线程池不是RUNNING
                && (runStateAtLeast(c, STOP) || firstTask != null ||
workQueue.isEmpty()))
                    //当前线程池状态为SHUTDOWN || 任务不为空 || 队列为空
                return false;
            //自旋：来到这里说明线程池的状态允许执行任务
            for (;;) {
                //如果当前线程池的数量大于等于最大线程数 && 工作线程数达到5亿
                if (workerCountOf(c)
                    >= ((core ? corePoolSize : maximumPoolSize) & COUNT_MASK))
                    return false;
                //如果cas使工作线程数+1成功，相当于申请到了一个工作线程
                if (compareAndIncrementWorkerCount(c))
                    break retry;
                c = ctl.get();
                //来到这里说明：
                //1.并发导致申请工作线程失败
                //2.线程池状态改变
```

```
                if (runStateAtLeast(c, SHUTDOWN))//如果线程池状态没问题，再次尝试获取工
作线程
                    continue retry;
            }
        }

        boolean workerStarted = false;//任务是否开始执行
        boolean workerAdded = false;//任务是否添加到池子中
        Worker w = null;
        try {
            w = new Worker(firstTask); //包装成一个worker
            final Thread t = w.thread;
            if (t != null) {
                final ReentrantLock mainLock = this.mainLock;
                mainLock.lock();
                try {

                    int c = ctl.get();
                    //如果当前线程池状态是RUNNING，任务不为空
                    if (isRunning(c) || (runStateLessThan(c, STOP) && firstTask
== null)) {
                        //如果线程状态不是就绪
                        if (t.getState() != Thread.State.NEW)
                            throw new IllegalThreadStateException();
                        workers.add(w);//加入到队列
                        workerAdded = true;
                        int s = workers.size();
                        if (s > largestPoolSize)
                            largestPoolSize = s;
                    }
                } finally {
                    mainLock.unlock();
                }
                if (workerAdded) {
                    t.start();//启动任务
                    workerStarted = true;
                }
            }
        } finally {
            if (! workerStarted)
                //如果启动失败，释放掉刚才自旋获取的工作线程
                addWorkerFailed(w);
        }
        return workerStarted;
    }
```

5，runWorker()

执行任务的线程去获取独占锁执行任务，执行完成后将线程完成的任务书+1并释放锁，然后执行退出逻辑。

```
    final void runWorker(Worker w) {
        Thread wt = Thread.currentThread();
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); //为什么先调用unlock()?为了强制刷新当前持有锁的线程为空。
```

```
        boolean completedAbruptly = true;//是否突然退出？true表示发生异常，当前线程突然退出的，false表示正常退出。
        try {
            //任务不为空 || 在queue中获取任务成功
            while (task != null || (task = getTask()) != null) {
                w.lock();//设置独占锁为当前线程
                //为什么要设置独占锁？防止shutdown时会判断当前work状态
                //如果线程池状态为STOP ||
                //（获取当前线程中断状态并给当前线程设置中断信号未false && 线程池状态为STOP）&& 线程未设置中断状态
                if ((runStateAtLeast(ctl.get(), STOP) ||
                     (Thread.interrupted() &&
                      runStateAtLeast(ctl.get(), STOP))) &&
                    !wt.isInterrupted())
                    wt.interrupt();
                try {
                    //钩子方法
                    beforeExecute(wt, task);
                    try {
                        //执行任务
                        task.run();
                        afterExecute(task, null);//钩子方法
                    } catch (Throwable ex) {
                        afterExecute(task, ex);//钩子方法
                        throw ex;
                    }
                } finally {
                    task = null;
                    w.completedTasks++;//将线程完成的任务数+1
                    w.unlock();//释放锁
                }
            }
            completedAbruptly = false;//异常打断标记设置为false
        } finally {
            //执行退出逻辑
            processWorkerExit(w, completedAbruptly);
        }
    }
```

6，getTask()

自旋的方式去队列获取任务

```
    private Runnable getTask() {
        boolean timedOut = false;

        for (;;) {
            int c = ctl.get();

            //如果当前线程池是非RUNNING && 当前状态大于等于 STOP || queue为空
            if (runStateAtLeast(c, SHUTDOWN)
                && (runStateAtLeast(c, STOP) || workQueue.isEmpty())) {
                decrementWorkerCount(); //将获取到的工作线程还回去
                return null;
            }
            /*
                能够来到这里的情况：
```

```
                1.RUNNING
                2.SHUTDOWN但是队列不为空
            */
            int wc = workerCountOf(c); //获取到worker数量

            //如果是核心线程就不需要超时时间
            boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;
            //条件1说明 当前工作线程数已经大于最大线程数
            //条件2说明  当前线程可以被回收  ||wc==1且任务队列已经空了，当前线程可以放心退出。
            if ((wc > maximumPoolSize || (timed && timedOut))
                && (wc > 1 || workQueue.isEmpty())) {
                if (compareAndDecrementWorkerCount(c))//如果成功减少worker数量
                    return null;
                continue;
            }

            try {
                Runnable r = timed ?
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    workQueue.take();//去队列获取任务
                if (r != null)//任务不为空，返回任务
                    return r;
                timedOut = true;//否则标记为超时
            } catch (InterruptedException retry) {
                timedOut = false;
            }
        }
    }
```

7，shutdown()

首先获取到全局锁，自旋设置线程池状态为SHUTDOWN，然后遍历所有线程，中断空闲线程，等待队列的任务会执行完，然后退出。

```
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();//获取全局锁
        try {
            checkShutdownAccess();//权限判断
            advanceRunState(SHUTDOWN);//自旋设置线程池状态为SHUTDOWN
            interruptIdleWorkers();//遍历所有的线程，中断空闲线程
            onShutdown(); // 空方法，子类可以扩展
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }
```

8，shutdownNow()

首先获取到全局锁，然后自旋设置线程池状态为STOP，尝试遍历中断线程池中所有线程，导出所有未处理的任务。

```
    public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
```

```
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(STOP);//自旋设置线程池状态为STOP
            interruptWorkers();//遍历中断线程池中所有线程
            tasks = drainQueue();//导出所有未处理的任务
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
        return tasks;
    }
```

9，tryTerminate()

通过自旋的方式，当非空闲线程执行完任务，就会来到这里被中断，最后一个退出的线程，设置线程池状态为TIDYING，最终设置为设置为TERMINATED，并调用termination.signalAll();唤醒调用awaitTermination()的线程继续执行程序。

```
    final void tryTerminate() {
        for (;;) {//自旋
            int c = ctl.get();
            /*
                线程池RUNNING ||线程池大于等于TIDYING(说明已经有线程在执行这个方法)
                || 线程池大于STOP状态 且 队列不为空
            */
            if (isRunning(c) ||
                runStateAtLeast(c, TIDYING) ||
                (runStateLessThan(c, STOP) && ! workQueue.isEmpty()))
                return;
            /*
                什么时候执行到这里？
                1.线程池状态大于等于STOP
                2.线程池状态大于等于STOP并且队列为空

                当前线程池中的线程数量>0
            */
            if (workerCountOf(c) != 0) {
                //非空闲线程当执行完任务，最终也会执行到这里
                interruptIdleWorkers(ONLY_ONE);//中断一个空闲线程
                return;
            }
            //最后一个退出的线程会来到这里
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                //强制设置线程池状态为TIDYING
                if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
                    try {
                        terminated(); //钩子方法
                    } finally {
                        //设置为TERMINATED
                        ctl.set(ctlOf(TERMINATED, 0));
                        //唤醒调用awaitTermination()的线程
                        termination.signalAll();
```

```
                    }
                    return;
                }
            } finally {
                mainLock.unlock();
            }
            // else retry on failed CAS
        }
    }
```