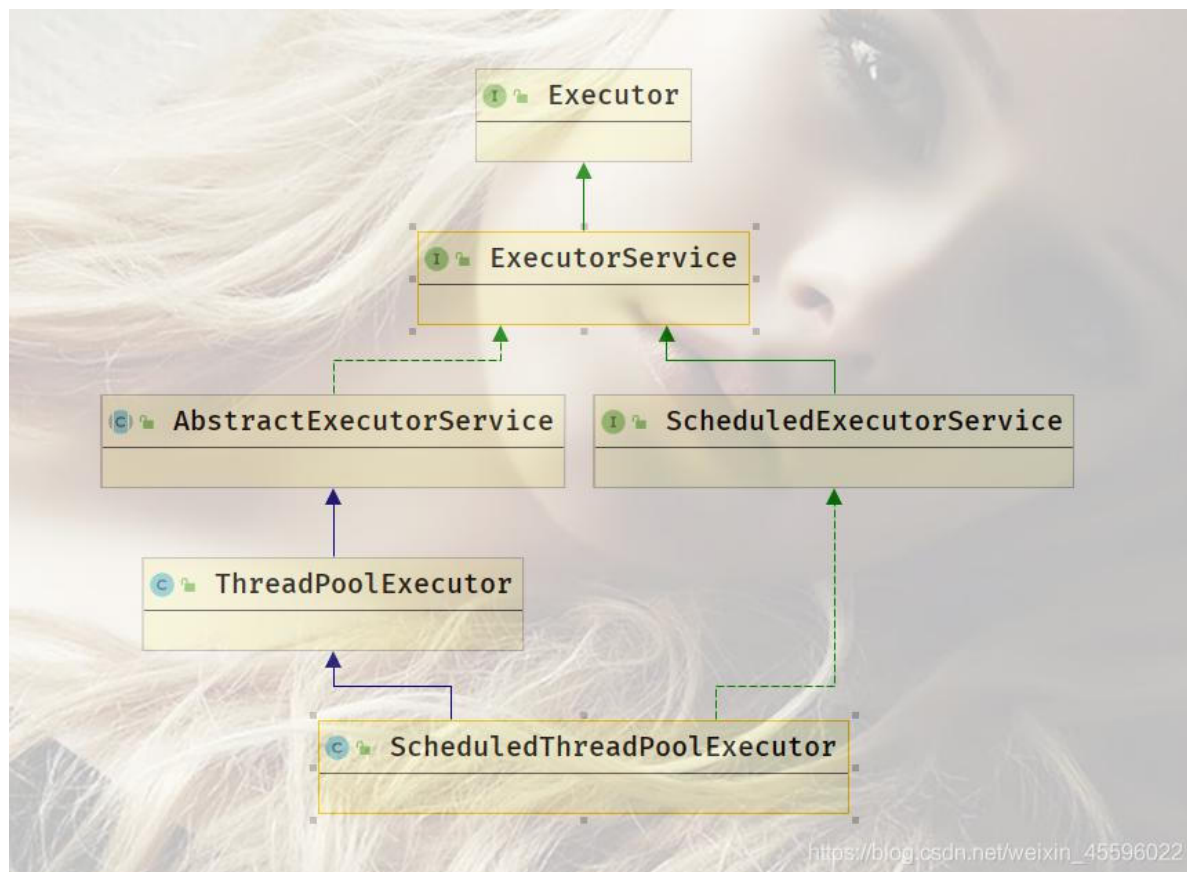


一，线程池的基本使用

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¹

`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。

1) 生产中线程池参数如何设置

为什么IO密集型的设置为 2n, 计算密集型设置为 n+1不对?

因为核心线程数设置多少要具体情况具体分析, 使用线程池的业务场景不同, 解决方案自然是不一样的。

场景假设

1、假设现在要给 100w 用户发放优惠券, 通过线程池异步发送

2、假设某线程池执行发优惠券的任务共耗时 50ms, 其中 45ms 在io, 5ms 在进行计算(真正的 io 耗时 计算耗时可以通过 记录log 判断时间差值计算出来 取平均值即可)

3、如何设置线程池的参数快速的将这 100w 张券发完?

核心线程数 = CPU核数 * ((Io耗时 / 计算耗时) + 1)
 核心线程数 = 8C * ((45ms / 5ms) + 1) = 80个

45ms / 5ms 是什么意思?

CPU 在等待 IO 返回时完全可以将 CPU 时间片拿出来去做其他的计算,45ms 可以多处理 9 个计算任务,再加上原本就有一个 5ms 在计算,也就是说: 一个CPU 核在执行这个 50ms 发券任务时,可以并发的起10 个线程去处理任务! 那8C CPU 最多同时可以有 8个核心并行的处理任务, $8 * 10 = 80$, 一秒钟一个线程可以处理 $1000ms / 50ms = 20$ 个任务可以算出线程池执行任务的峰值 $qps = 20 * 80 = 1600$, 发完 100w 张券所需时间: $100w / 1600 = 625S$,也就是说大概 10分钟左右就能发完 100w 张券。

不太正确的结论: 核心线程数在处理这个任务的情况下可以设置为 80 用来极限的压榨机器CPU 的性能。

核心线程数设置为 80,这几乎吃完了所有的 CPU 时间片, CPU 的负载将会达到 100%; 试想一下生产环境如果你的机器 CPU 负载是 100%, 慌不慌? (CPU 负载打满机器不会宕机, 但没有 CPU 资源来处理用户的请求,表现为服务假死/机器请求半天无反应)

设置线程池核心线程数要考虑 CPU 的使用要素

- 1、每台机器操作系统需要消耗一些 CPU 资源; 假设用了 2% 的CPU 资源;
- 2、如果是面向用户的服务,处理用户的请求也是要消耗CPU 资源的,可以通过一些监控系统,看看平时 CPU 在繁忙时间段的负载是多少; 假设用了 10% 的资源;
- 3、如果除了发券任务的线程池还有其他线程池在运行,就得把其他线程池消耗的CPU资源也算上,假设用了 13% 的资源;
- 4、实际情况一些中间件框架也会用线程池,也会吃一些CPU 资源。

为什么用线程池没考虑上下文的切换?

$1ms = 1000us$, 一次上下文的切换大概是 $1us$, 上下文切换的时间跟执行任务的时间比起来可以忽略不计。

结论: $CPU核数 * ((Io耗时 / 计算耗时) + 1)$

这是机器 CPU 负载 100% 时极限的值, 乘以期望的 CPU 负载百分比即可算出实际情况最佳的线程数。

2) 8C16G 的机器需要几台可以抗起 3W 的qps?

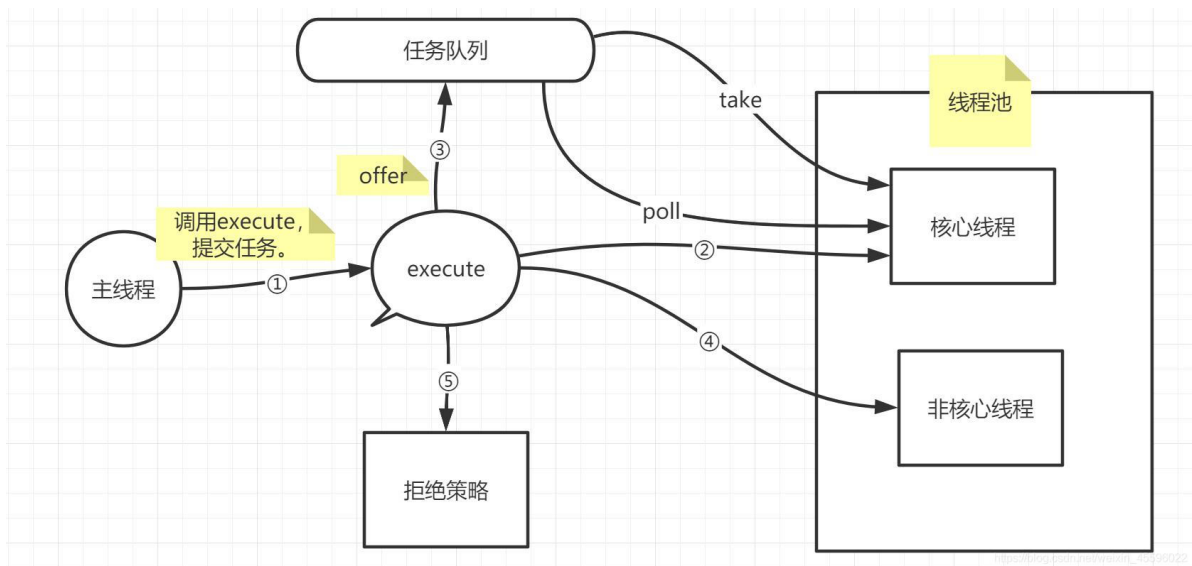
假设一个 用户领券系统的 qps 在3w左右大部分服务通常的部署在 Tomcat 上, Tomcat 内部也是通过线程来处理用户的请求,Tomcat 也是通过线程池来管理线程, 实际上算出 Tomcat 实际的并发和理想状态能支持的并发就好了。

上个问题分析出来发券接口 50ms 耗时, 8C 的CPU 占用 100%, 不考虑内存 磁盘 网络等其他开销, 线程池极限的QPS 是1600, 这里也不考虑有没有其他线程池或者七七八八的东西消耗 CPU 资源了。假设 CPU 只能维持在 70% 左右的负载; 单台机器的 qps 就只能是 $1600 * 70\% = 1120$, 就算 1100, $3w / 1100 = 27.27$ 向上取整 大概需要 28 台机器。作为一个有经验的开发人员实际部署的时候绝对要多扩容几台服务器来兜底, 推荐部署 32 - 36 台机器分两个集群部署。

3) 线程池可以先启动最大线程数再将任务放到阻塞队列里么?

启动最大线程数再将任务放到阻塞队列的诀窍就在 workQueue 的 offer 方法;我们可以用自己实现的阻塞队列在重写 offer 方法; 在 offer 方法中判断 当前线程数是否大于等于最大线程数, 如果不大于就返回 false, 这样就跳过了 execute 方法的第二步, 来到了第三步的创建最大线程数的逻辑。dubbo就是这么干的。

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;

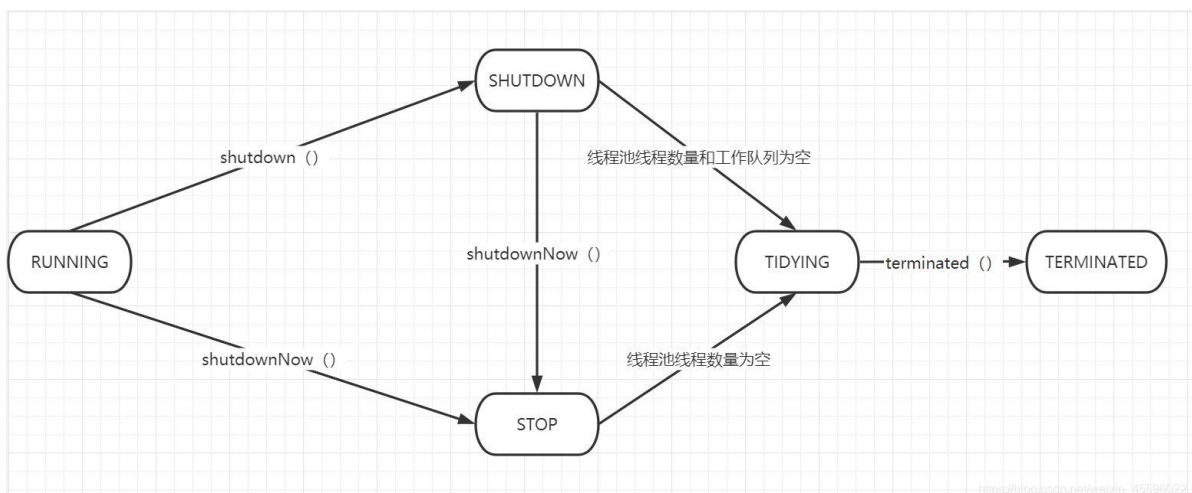
2.4如果队列满了且正在运行的线程数量大于或等于maximumPoolSize,那么线程池会启动饱和和拒绝策略来执行。

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。

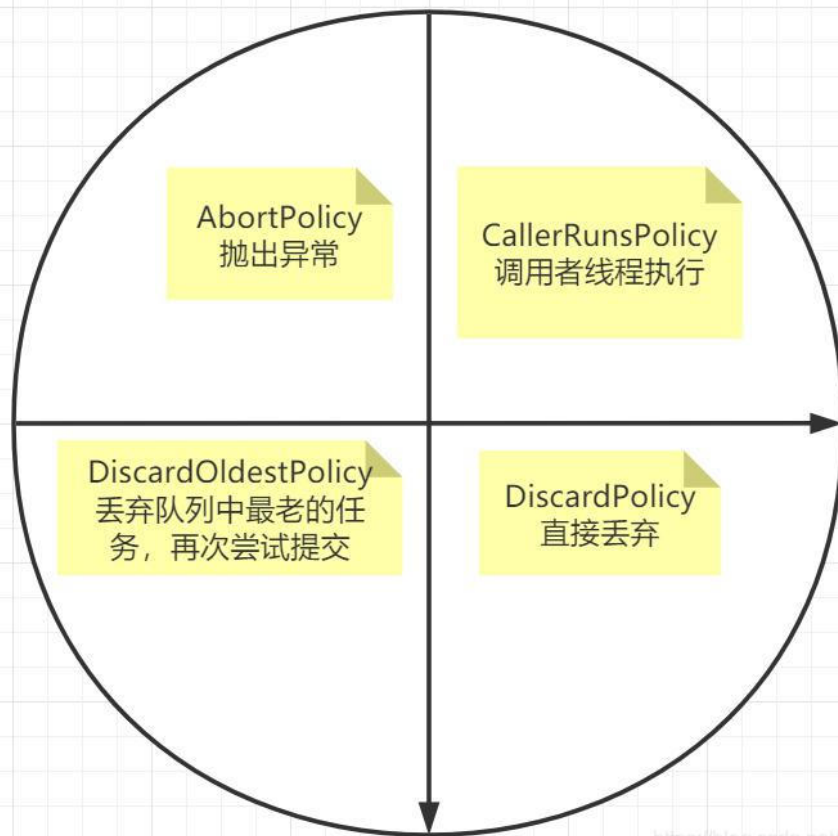
4、 当一个线程无事可做超过一定的时间(keepAliveTime) 时，线程会判断:

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



https://blog.csdn.net/yaixin_45596022

5, 自定义线程池

```
public static void main(String[] args) {
    ExecutorService pool = new ThreadPoolExecutor(
        2,
        5,
        31,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<>(3),
        Executors.defaultThreadFactory(),
        //new ThreadPoolExecutor.AbortPolicy()    //抛出异常
        //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<>();

/*
```

线程池中提供了一个对外方法，`awaitTermination(long time,TimeUnit)`，该方法调用会被阻塞，并且在以下几种情况任意一种发生时都会导致该方法的执行：即`shutdown`方法被调用之后，或者参数中定义的`timeout`时间到达或者当前线程被打断，这几种情况任意一个发生了都会导致该方法在所有任务完成之后才执行。第一个参数是`long`类型的超时时间，第二个参数可以为该时间指定单位。

```
*/
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()

他会先尝试去停止所有空闲的线程，然后执行tryTerminate(), tryTerminate()会通过自旋的方式判断，当队列里面的任务都执行完了，销毁掉执行完任务的线程，当最后一个线程退出的时候，将线程池状态修改为TIDYING，最终修改为TERMINATED。

```

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()

自旋设置线程池状态为STOP，遍历中断（interrupt的方式）线程池中所有线程，导出所有未处理的任任务，然后执行tryTerminate()。

```

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
}
}
```

1. 此处问题答案来源于携程大佬，仅作为个人学习使用。 [🔗](#)