**深入Java线程池：从设计思想到源码解读**

<https://mp.weixin.qq.com/s/siGOSBZdSOlS-0tlXyJyJQ>

**初识线程池**

我们知道，线程的创建和销毁都需要映射到操作系统，因此其代价是比较高昂的。出于避免频繁创建、销毁线程以及方便线程管理的需要，线程池应运而生。

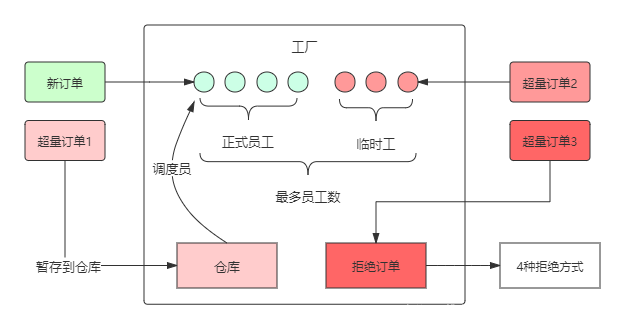
**线程池优势**

* **降低资源消耗**：线程池通常会维护一些线程（数量为 corePoolSize），这些线程被重复使用来执行不同的任务，任务完成后不会销毁。在待处理任务量很大的时候，通过对线程资源的复用，避免了线程的频繁创建与销毁，从而降低了系统资源消耗。
* **提高响应速度**：由于线程池维护了一批 alive 状态的线程，当任务到达时，不需要再创建线程，而是直接由这些线程去执行任务，从而减少了任务的等待时间。
* **提高线程的可管理性**：使用线程池可以对线程进行统一的分配，调优和监控。

**线程池设计思路**

有句话叫做艺术来源于生活，编程语言也是如此，很多设计思想能映射到日常生活中，比如面向对象思想、封装、继承，等等。今天我们要说的线程池，它同样可以在现实世界找到对应的实体——工厂。

先假想一个工厂的生产流程：



工厂中有固定的一批工人，称为正式工人，工厂接收的订单由这些工人去完成。当订单增加，正式工人已经忙不过来了，工厂会将生产原料暂时堆积在仓库中，等有空闲的工人时再处理（因为工人空闲了也不会主动处理仓库中的生产任务，所以需要调度员实时调度）。仓库堆积满了后，订单还在增加怎么办？

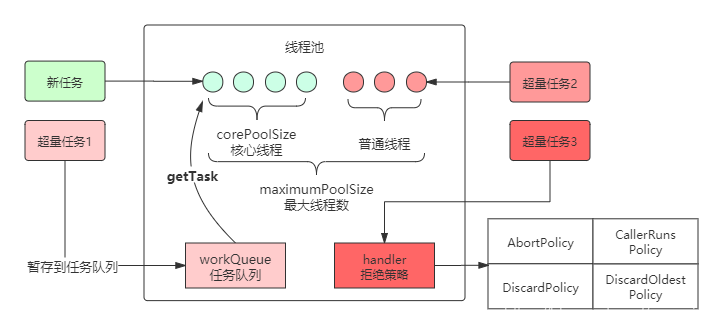
工厂只能临时扩招一批工人来应对生产高峰，而这批工人高峰结束后是要清退的，所以称为临时工。当时临时工也以招满后（受限于工位限制，临时工数量有上限），后面的订单只能忍痛拒绝了。

我们做如下一番映射：

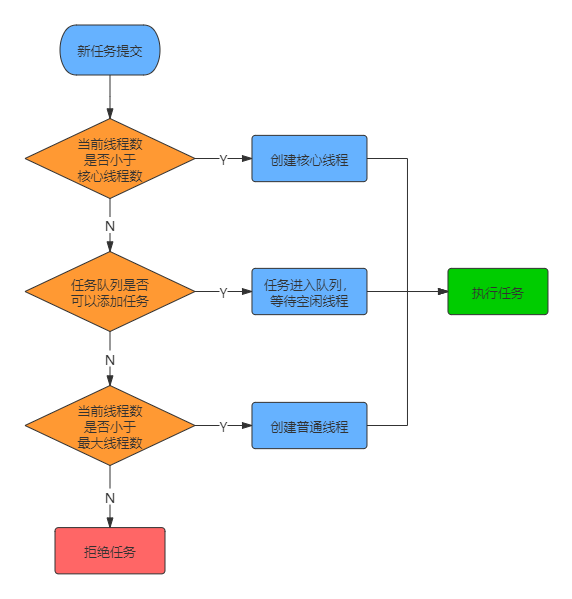
* 工厂——线程池
* 订单——任务（Runnable）
* 正式工人——核心线程
* 临时工——普通线程
* 仓库——任务队列
* 调度员——getTask()

getTask()是一个方法，将任务队列中的任务调度给空闲线程，在解读线程池有详细介绍。

映射后，形成线程池流程图如下，两者是不是有异曲同工之妙？

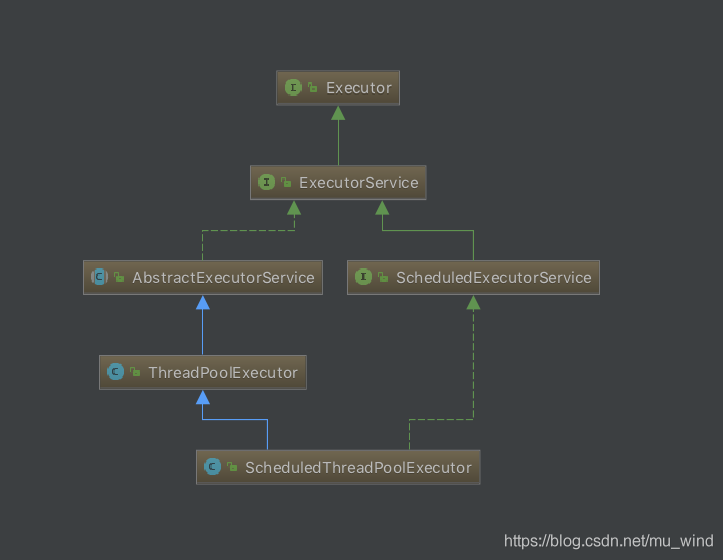


这样，线程池的工作原理或者说流程就很好理解了，提炼成一个简图：



### 深入线程池

那么接下来，问题来了，线程池是具体如何实现这套工作机制的呢？从Java线程池Executor框架体系可以看出：线程池的真正实现类是ThreadPoolExecutor，因此我们接下来重点研究这个类。



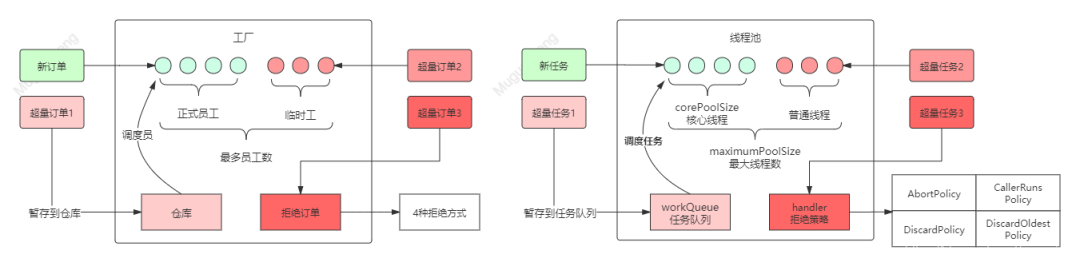
### 构造方法

研究一个类，先从它的构造方法开始。ThreadPoolExecutor提供了4个有参构造方法：

解释一下构造方法中涉及到的参数：

* **corePoolSize（必需）：** 核心线程数。即池中一直保持存活的线程数，即使这些线程处于空闲。但是将allowCoreThreadTimeOut参数设置为true后，核心线程处于空闲一段时间以上，也会被回收。
* **maximumPoolSize（必需）：** 池中允许的最大线程数。当核心线程全部繁忙且任务队列打满之后，线程池会临时追加线程，直到总线程数达到maximumPoolSize这个上限。
* **keepAliveTime（必需）：** 线程空闲超时时间。当非核心线程处于空闲状态的时间超过这个时间后，该线程将被回收。将allowCoreThreadTimeOut参数设置为true后，核心线程也会被回收。
* **unit（必需）：** keepAliveTime参数的时间单位。有：TimeUnit.DAYS（天）、TimeUnit.HOURS（小时）、TimeUnit.MINUTES（分钟）、TimeUnit.SECONDS（秒）、TimeUnit.MILLISECONDS（毫秒）、TimeUnit.MICROSECONDS（微秒）、TimeUnit.NANOSECONDS（纳秒）
* **workQueue（必需）：** 任务队列，采用阻塞队列实现。当核心线程全部繁忙时，后续由execute方法提交的Runnable将存放在任务队列中，等待被线程处理。
* **threadFactory（可选）：** 线程工厂。指定线程池创建线程的方式。
* **handler（可选）：** 拒绝策略。当线程池中线程数达到maximumPoolSize且workQueue打满时，后续提交的任务将被拒绝，handler可以指定用什么方式拒绝任务。

放到一起再看一下：



**任务队列**

使用ThreadPoolExecutor需要指定一个实现了BlockingQueue接口的任务等待队列。在ThreadPoolExecutor线程池的API文档中，一共推荐了三种等待队列，它们是：SynchronousQueue、LinkedBlockingQueue和ArrayBlockingQueue；

* **SynchronousQueue：** 同步队列。这是一个内部没有任何容量的阻塞队列，任何一次插入操作的元素都要等待相对的删除/读取操作，否则进行插入操作的线程就要一直等待，反之亦然。
* **LinkedBlockingQueue：** 无界队列（严格来说并非无界，上限是Integer.MAX\_VALUE），基于链表结构。使用无界队列后，当核心线程都繁忙时，后续任务可以无限加入队列，因此线程池中线程数不会超过核心线程数。这种队列可以提高线程池吞吐量，但代价是牺牲内存空间，甚至会导致内存溢出。另外，使用它时可以指定容量，这样它也就是一种有界队列了。
* **ArrayBlockingQueue：** 有界队列，基于数组实现。在线程池初始化时，指定队列的容量，后续无法再调整。这种有界队列有利于防止资源耗尽，但可能更难调整和控制。

另外，Java还提供了另外4种队列：

另外，Java还提供了另外4种队列：

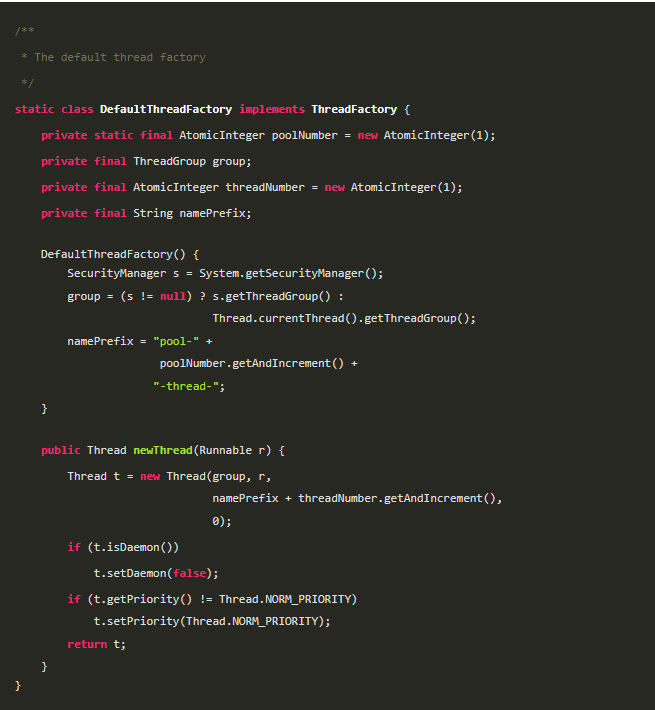
* **PriorityBlockingQueue：** 支持优先级排序的无界阻塞队列。存放在PriorityBlockingQueue中的元素必须实现Comparable接口，这样才能通过实现compareTo()方法进行排序。优先级最高的元素将始终排在队列的头部；PriorityBlockingQueue不会保证优先级一样的元素的排序，也不保证当前队列中除了优先级最高的元素以外的元素，随时处于正确排序的位置。
* **DelayQueue：** 延迟队列。基于二叉堆实现，同时具备：无界队列、阻塞队列、优先队列的特征。DelayQueue延迟队列中存放的对象，必须是实现Delayed接口的类对象。通过执行时延从队列中提取任务，时间没到任务取不出来。更多内容请见DelayQueue。
* **LinkedBlockingDeque：** 双端队列。基于链表实现，既可以从尾部插入/取出元素，还可以从头部插入元素/取出元素。
* **LinkedTransferQueue：** 由链表结构组成的无界阻塞队列。这个队列比较特别的时，采用一种预占模式，意思就是消费者线程取元素时，如果队列不为空，则直接取走数据，若队列为空，那就生成一个节点（节点元素为null）入队，然后消费者线程被等待在这个节点上，后面生产者线程入队时发现有一个元素为null的节点，生产者线程就不入队了，直接就将元素填充到该节点，并唤醒该节点等待的线程，被唤醒的消费者线程取走元素。

**拒绝策略**

线程池有一个重要的机制：拒绝策略。当线程池workQueue已满且无法再创建新线程池时，就要拒绝后续任务了。拒绝策略需要实现RejectedExecutionHandler接口，不过Executors框架已经为我们实现了4种拒绝策略：

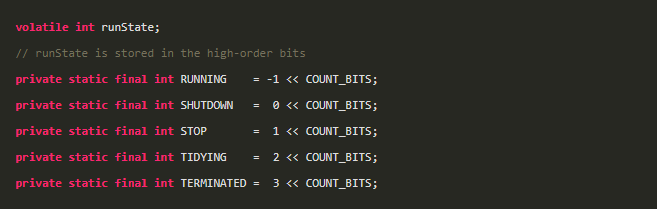
* **AbortPolicy（默认）：** 丢弃任务并抛出RejectedExecutionException异常。
* **CallerRunsPolicy：** 直接运行这个任务的run方法，但并非是由线程池的线程处理，而是交由任务的调用线程处理。
* **DiscardPolicy：** 直接丢弃任务，不抛出任何异常。
* **DiscardOldestPolicy：** 将当前处于等待队列列头的等待任务强行取出，然后再试图将当前被拒绝的任务提交到线程池执行。

线程工厂指定创建线程的方式，这个参数不是必选项，Executors类已经为我们非常贴心地提供了一个默认的线程工厂：



### 线程池状态

线程池有5种状态：



runState表示当前线程池的状态，它是一个 volatile 变量用来保证线程之间的可见性。

下面的几个static final变量表示runState可能的几个取值，有以下几个状态：

* **RUNNING：** 当创建线程池后，初始时，线程池处于RUNNING状态；
* **SHUTDOWN：** 如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；
* **STOP：** 如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；
* **TERMINATED：** 当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

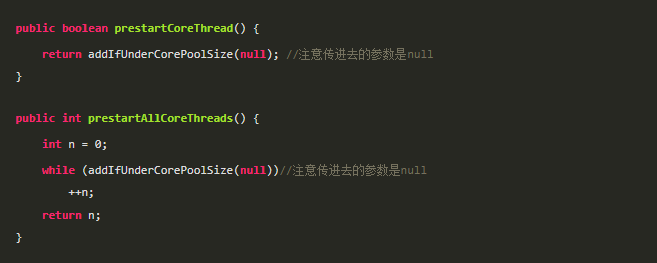
### 初始化&容量调整&关闭

#### 1、线程初始化

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。

在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

* prestartCoreThread()：boolean prestartCoreThread()，初始化一个核心线程
* prestartAllCoreThreads()：int prestartAllCoreThreads()，初始化所有核心线程，并返回初始化的线程数



#### 2、线程池关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭：

* shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
* shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

#### 3、线程池容量调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

### 使用线程池

#### ThreadPoolExecutor

通过构造方法使用ThreadPoolExecutor是线程池最直接的使用方式，下面看一个实例：



### Executors封装线程池

另外，Executors封装好了4种常见的功能线程池（还是那么地贴心）：

#### 1、FixedThreadPool

固定容量线程池。其特点是最大线程数就是核心线程数，意味着线程池只能创建核心线程，keepAliveTime为0，即线程执行完任务立即回收。任务队列未指定容量，代表使用默认值Integer.MAX\_VALUE。适用于需要控制并发线程的场景。



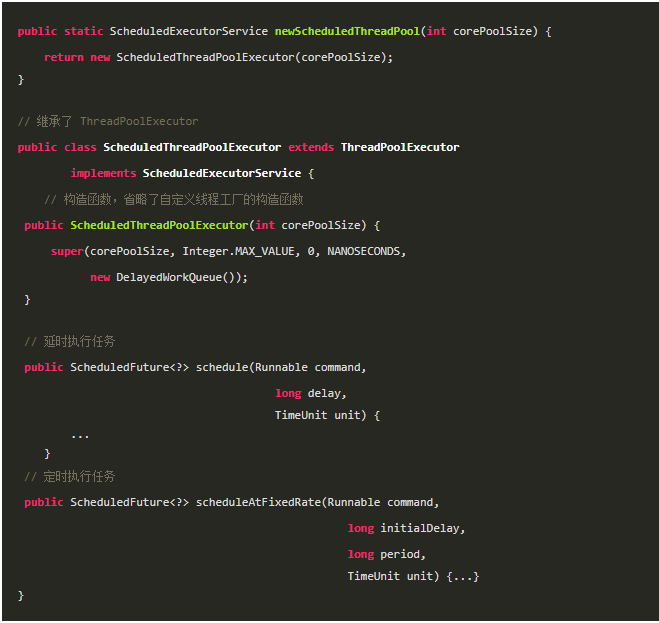
#### 2、 SingleThreadExecutor

单线程线程池。特点是线程池中只有一个线程（核心线程），线程执行完任务立即回收，使用有界阻塞队列（容量未指定，使用默认值Integer.MAX\_VALUE）



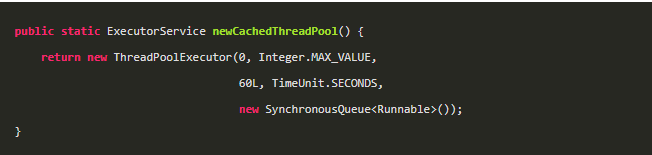
#### 3、 ScheduledThreadPool

定时线程池。指定核心线程数量，普通线程数量无限，线程执行完任务立即回收，任务队列为延时阻塞队列。这是一个比较特别的线程池，适用于执行定时或周期性的任务。



#### 4、CachedThreadPool

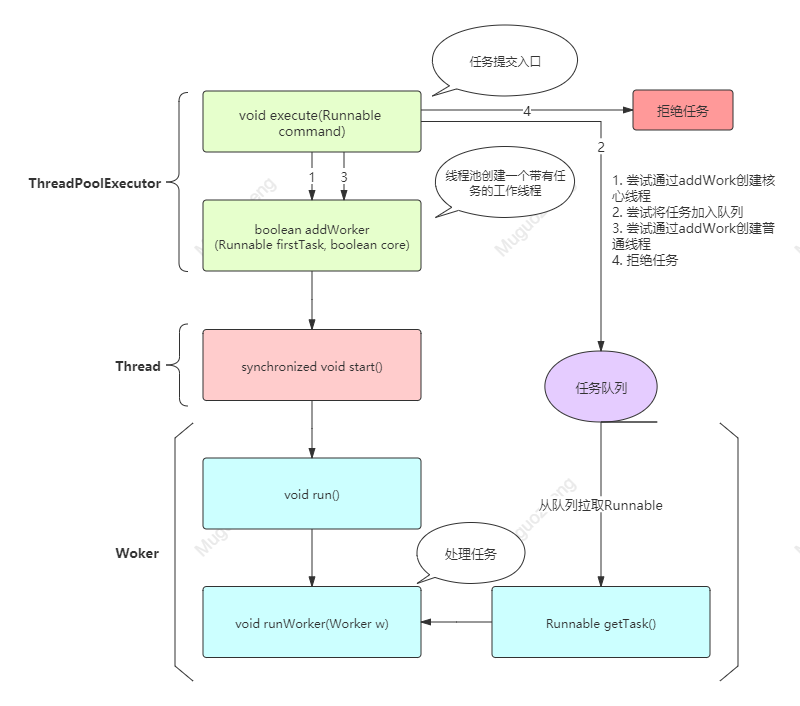
缓存线程池。没有核心线程，普通线程数量为Integer.MAX\_VALUE（可以理解为无限），线程闲置60s后回收，任务队列使用SynchronousQueue这种无容量的同步队列。适用于任务量大但耗时低的场景。



### 解读线程池

OK，相信前面内容阅读起来还算轻松愉悦吧，那么从这里开始就进入深水区了，如果后面内容能吃透，那么线程池知识就真的被你掌握了。

我们知道，向线程池提交任务是用ThreadPoolExecutor的execute()方法，但在其内部，线程任务的处理其实是相当复杂的，涉及到ThreadPoolExecutor、Worker、Thread三个类的6个方法：



### execute()

在ThreadPoolExecutor类中，任务提交方法的入口是execute(Runnable command)方法（submit()方法也是调用了execute()），该方法其实只在尝试做一件事：经过各种校验之后，调用 addWorker(Runnable command,boolean core)方法为线程池创建一个线程并执行任务，与之相对应，execute()的结果有两个：

参数说明：

* Runnable command：待执行的任务

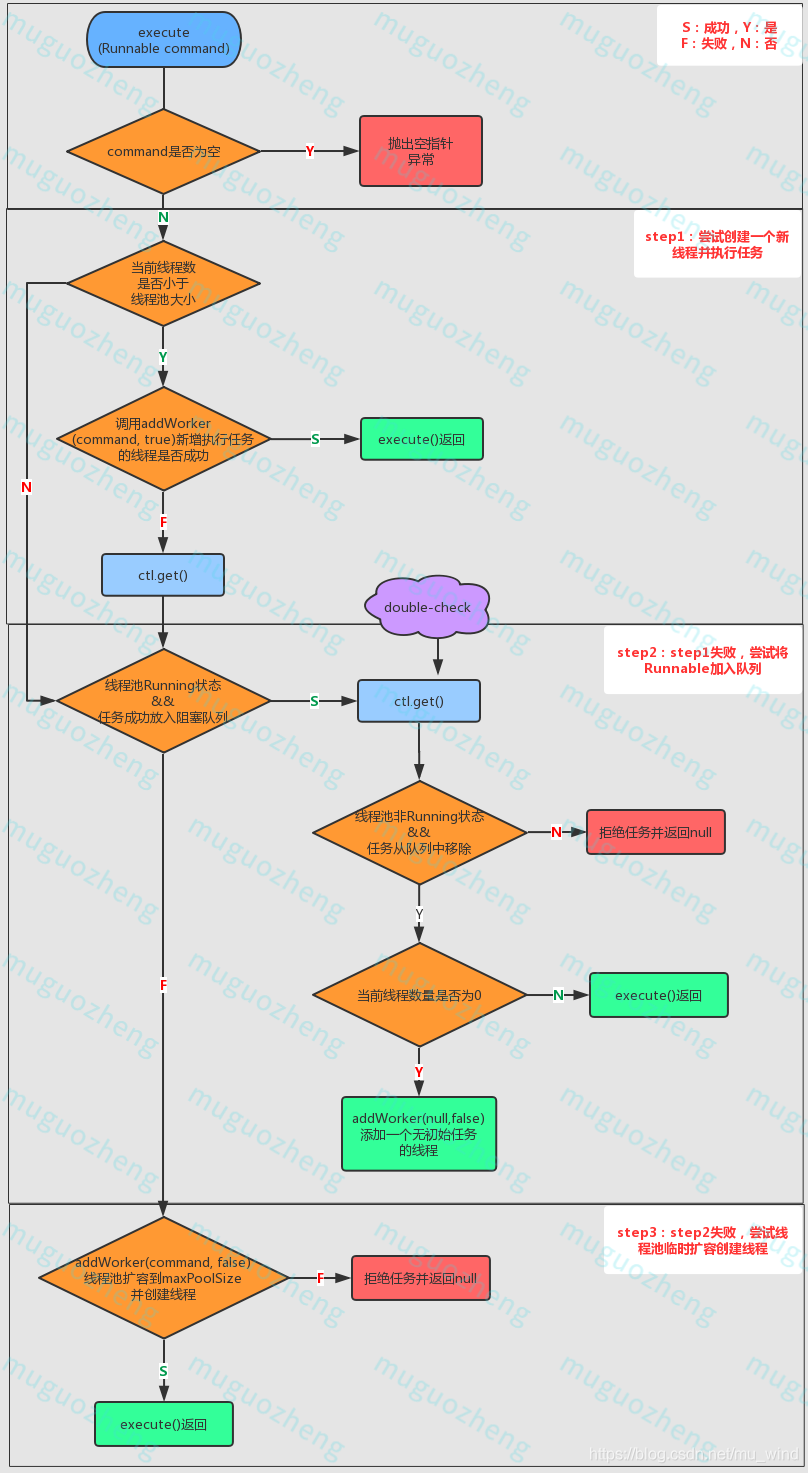
执行流程：

1、通过 ctl.get()得到线程池的当前线程数，如果线程数小于corePoolSize，则调用 addWorker(commond,true)方法创建新的线程执行任务，否则执行步骤2；

2、步骤1失败，说明已经无法再创建新线程，那么考虑将任务放入阻塞队列，等待执行完任务的线程来处理。基于此，判断线程池是否处于Running状态（只有Running状态的线程池可以接受新任务），如果任务添加到任务队列成功则进入步骤3，失败则进入步骤4；

3、来到这一步需要说明任务已经加入任务队列，这时要二次校验线程池的状态，会有以下情形：

* 线程池不再是Running状态了，需要将任务从任务队列中移除，如果移除成功则拒绝本次任务
* 线程池是Running状态，则判断线程池工作线程是否为0，是则调用 addWorker(commond,true)添加一个没有初始任务的线程（这个线程将去获取已经加入任务队列的本次任务并执行），否则进入步骤4；
* 线程池不是Running状态，但从任务队列移除任务失败（可能已被某线程获取？），进入步骤4；
* 4、将线程池扩容至maximumPoolSize并调用 addWorker(commond,false)方法创建新的线程执行任务，失败则拒绝本次任务。
* 流程图：



**addWorker()**

addWorker(Runnable firstTask, boolean core)方法，顾名思义，向线程池添加一个带有任务的工作线程。

参数说明：

* Runnable firstTask：新创建的线程应该首先运行的任务（如果没有，则为空）。
* boolean core：该参数决定了线程池容量的约束条件，即当前线程数量以何值为极限值。参数为 true 则使用corePollSize 作为约束值，否则使用maximumPoolSize。

执行流程：

1、外层循环判断线程池的状态是否可以新增工作线程。这层校验基于下面两个原则：

* 线程池为Running状态时，既可以接受新任务也可以处理任务
* 线程池为关闭状态时只能新增空任务的工作线程（worker）处理任务队列（workQueue）中的任务不能接受新任务

2、内层循环向线程池添加工作线程并返回是否添加成功的结果。

* 首先校验线程数是否已经超限制，是则返回false，否则进入下一步
* 通过CAS使工作线程数+1，成功则进入步骤3，失败则再次校验线程池是否是运行状态，是则继续内层循环，不是则返回外层循环

3、核心线程数量+1成功的后续操作：添加到工作线程集合，并启动工作线程

* 首先获取锁之后，再次校验线程池状态（具体校验规则见代码注解），通过则进入下一步，未通过则添加线程失败
* 线程池状态校验通过后，再检查线程是否已经启动，是则抛出异常，否则尝试将线程加入线程池
* 检查线程是否启动成功，成功则返回true，失败则进入 addWorkerFailed 方法

流程图：

