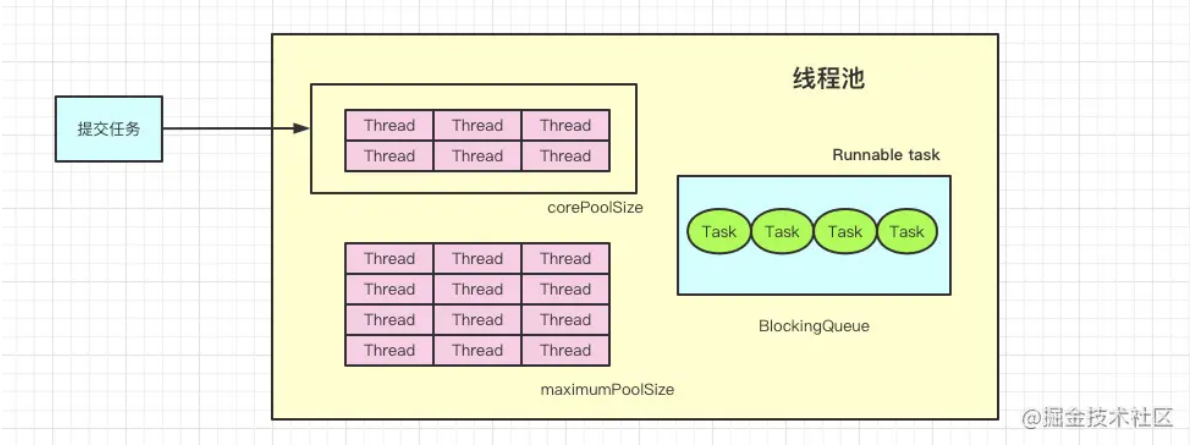
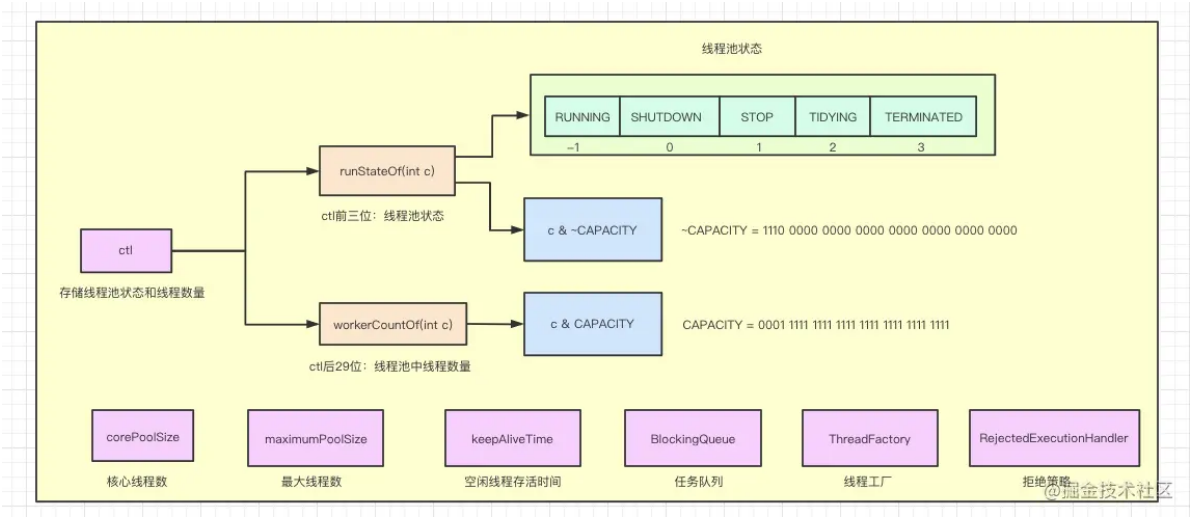
ThreadPoolExecutor



线程池是一种池化策略的产物，通过重复利用已创建的线程降低线程创建和销毁带来的开销，可以降低资源消耗；当任务到达时，直接执行不用等待线程创建，可以提高响应速度；其次通过线程池来进行线程的分配、调
优和监控，可以提高线程的可管理性．线程池主要由核心线程池(Work)、非核心线程池(Work)，阻塞队列(Task)等组成，线程池与线程类似也有运行状态．



- 线程池属性
  - ctl

    线程池中用AtomicInteger类型的ctl来记录线程池的状态和线程数量，其中高3位表示线程池状态，
    低29位表示当前线程池中的线程数量．
  - 状态/状态间转化
    - RUNNING = -1 << COUNT_BITS

      线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理．
    - SHUTDOWN = 0 << COUNT_BITS

      线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务．
    - STOP = 1 << COUNT_BITS

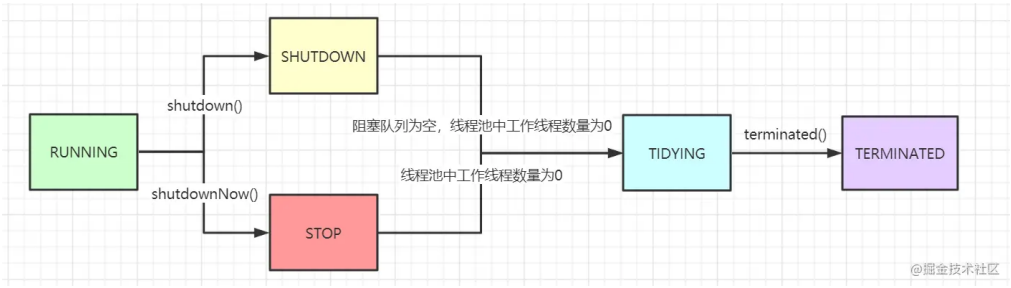
      线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务．
    - TIDYING = 2 << COUNT_BITS

当所有任务已终止，且当前ctl记录的任务数为0时，线程池变为TIDYING状态，然后执行**terminated**函数(ThreadPoolExecutor类中该函数为空，可供用户重载，进行功能扩展).

- TERMINATED = 3 << COUNT_BITS

  线程池彻底终止，就变成TERMINATED状态.

- 转换关系



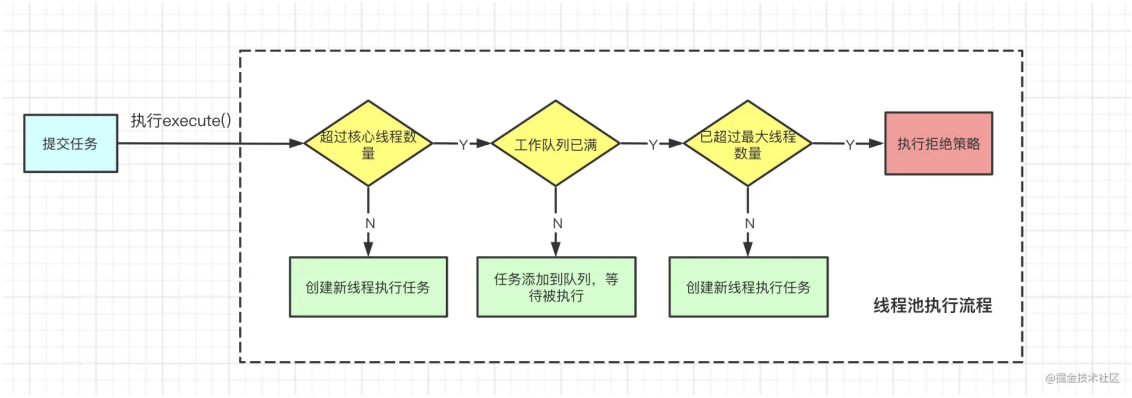
  - corePoolSize

    线程池中设定的核心线程的上界.

  - maximumPoolSize

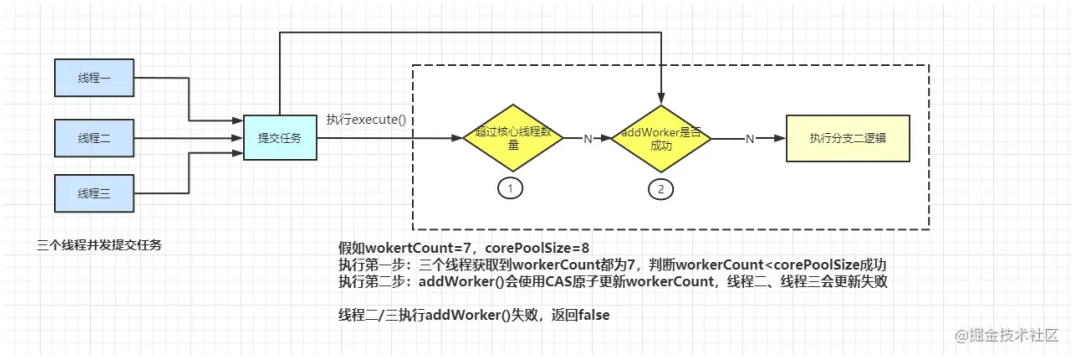
    线程池中设定的最大线程上界.

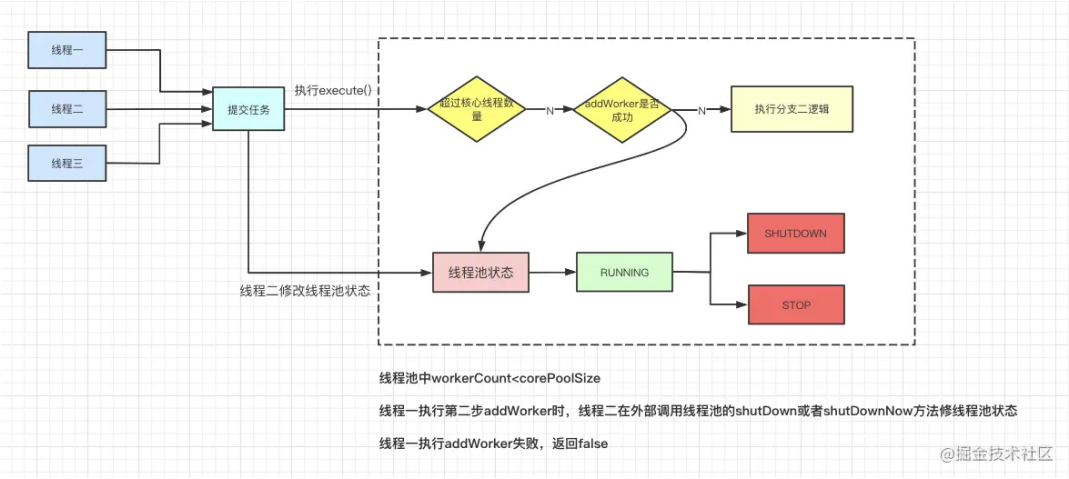
- executor(线程池任务执行流程)



任务提交到线程池的执行流程主要有三个阶段：线程数没有超过核心线程数量则创建新线程执行任务、超过核心线程数量则加入到工作队列(阻塞队列)、阻塞队列满了线程数没有超过最大线程数量则创建新线程执行任务
，上面阶段都不满足则执行拒绝策略.

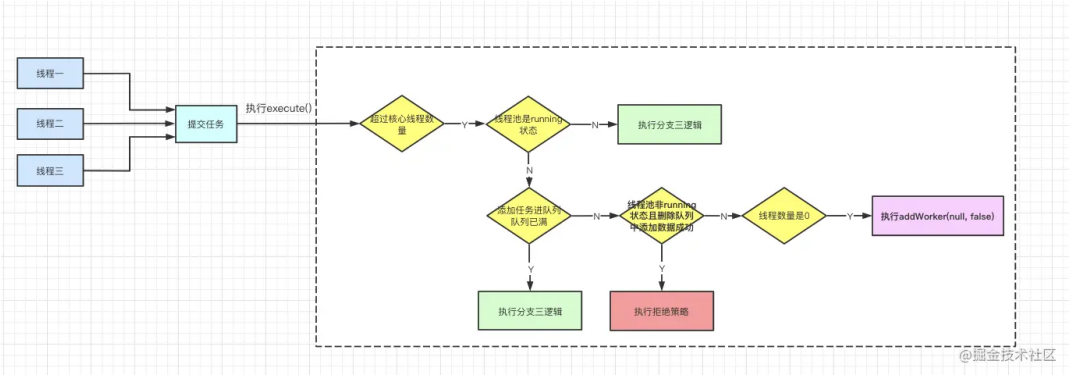
  - stage 1： workerCount < corePoolSize

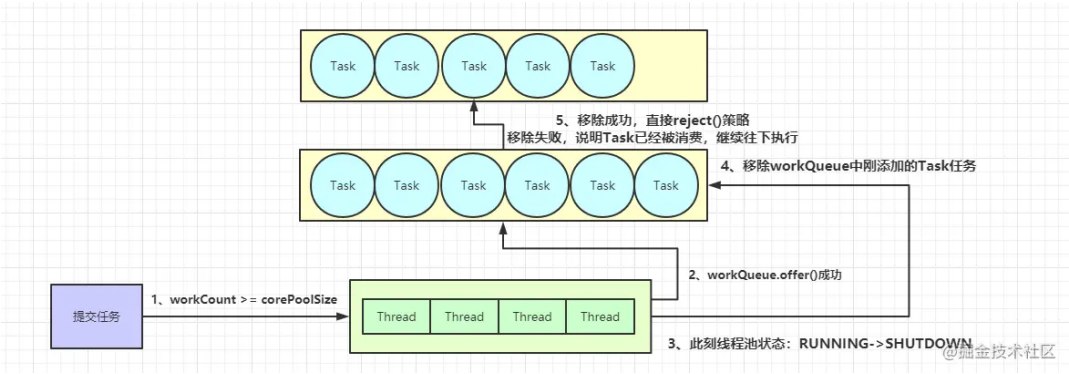




调用addWorker函数，创建一个新的worker来执行当前任务；addWorker创建新worker前会检测当前线程池的状态是否为RUNNING(其它线程调用了shutDown/shutDownNow改变了线程池状态)，并且会再次判断当前线程数量是否小于核心线程数量(多线程执行时其它线程创建了worker)，如果上述条件不满足则添加失败.

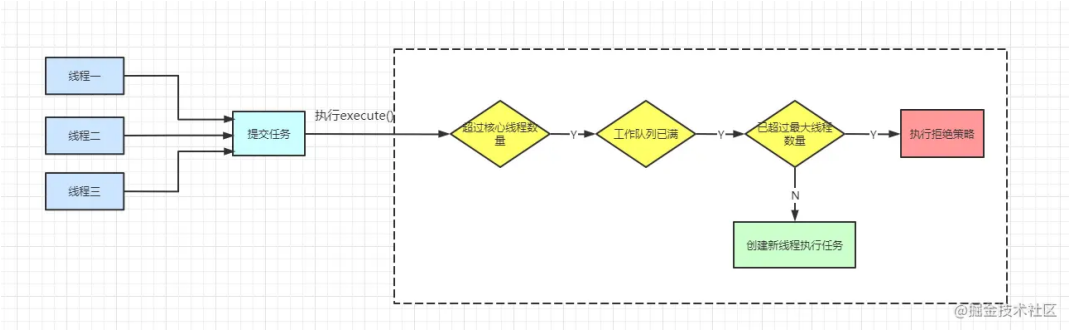
- stage 2: workerCount >= corePoolSize || addWorker -> false





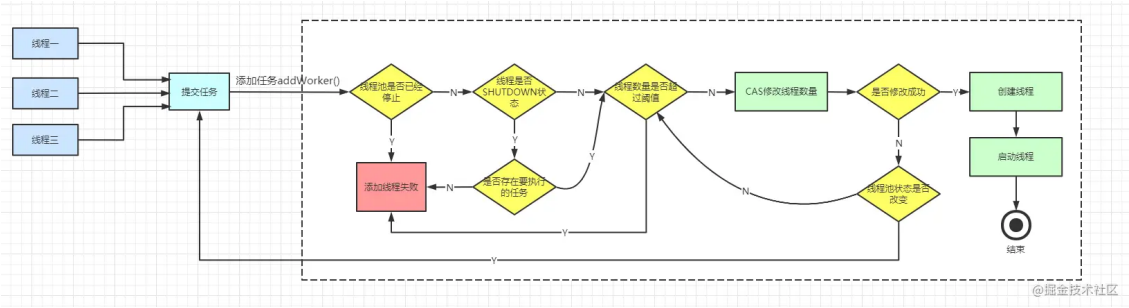
第一阶段条件不满足，进入第二阶段，因此首先判断线程池状态是否为RUNNING(addWorker导致进入第二阶段)，如果不是RUNNING则直接进入第三阶段；否则，则是因为workerCount>= corePoolSize进入第二阶段，调用workQueue.offer将当前任务task加入阻塞队列中，阻塞队列满了或者添加失败，则进入第三阶段；否则，表示task成功被加入阻塞队列，此时需要再次判断线程池状态是否为RUNNING，不为RUNNING则需要尝试删除当前添加的task(可能无法处理该task，需要拒绝执行)，执行拒绝策略，如果当前task已经被消费则会移除失败(task已经被安全执行，无需执行拒绝策略，正常退出)；如果是RUNNING状态，则判断当前线程池是否存在线程，不存在则创建一个无当前任务的worker执行阻塞队列中的task.

- stage 3: workerCount >= corePoolSize && blockQueue is full



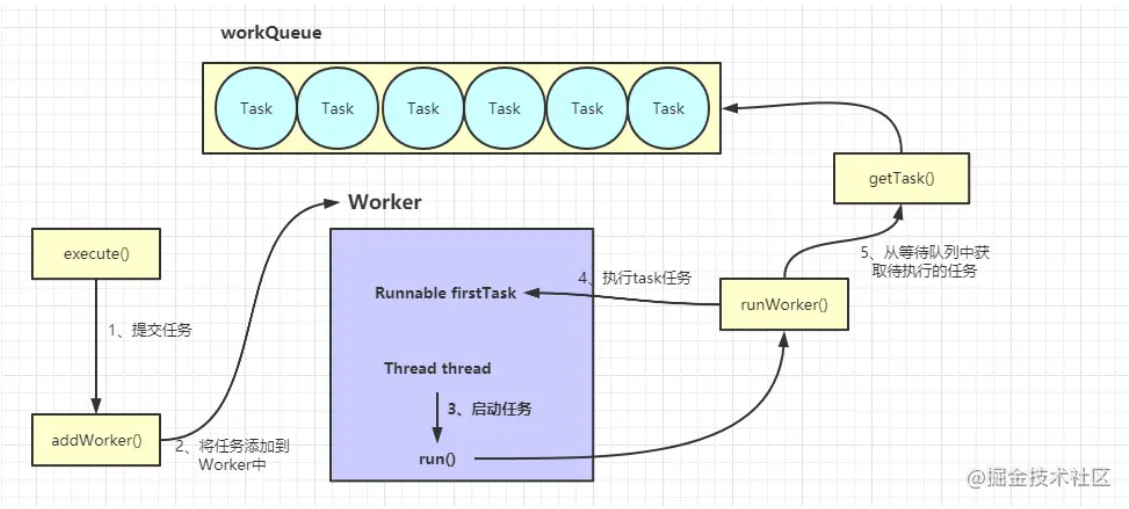
进入第三阶段，表示当前线程数已经超过核心线程数并且阻塞队列已经满了，此时只能尝试调用addWoker创建新woker来执行当前task，addWoker会判断当前线程数量是否超过maximumPoolSize，
超过则添加失败，执行拒绝策略.

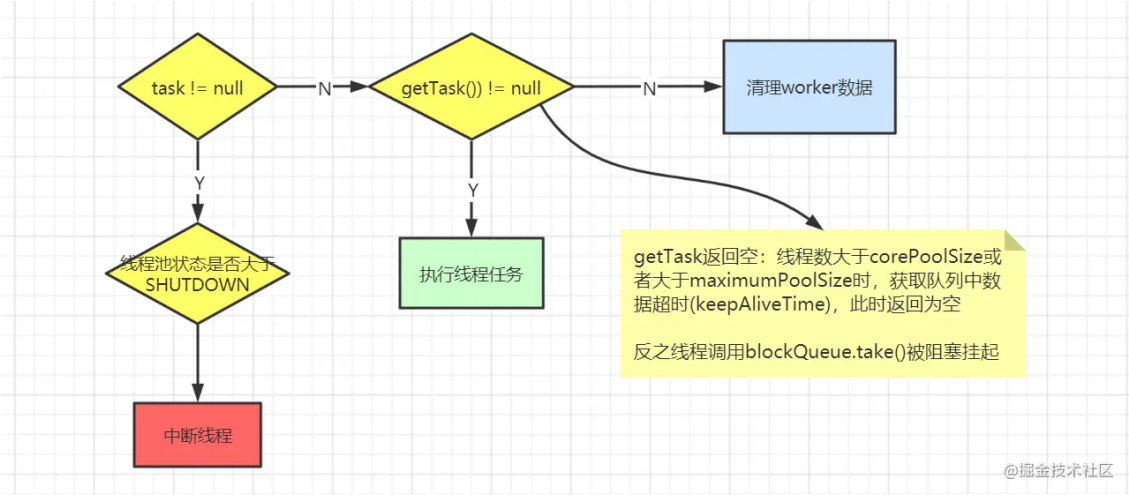
- addWorker



addWorker函数先判断线程池状态是否合法(RUNNING / SHUTDOWN && firstTask == null && ! workQueue.isEmpty)，1）线程池正常运行，那么可以创建新work来完task/firstTask，或者是2）当
前线程池处于SHUTDOWN状态因此停止处理新task(firstTask == null)，可以处理阻塞队列中的task (workQueue.isEmpty)，所以可以新建无firstTask的work；线程池状态合法后，还需要判断线程数是否超过线程阈值(根据addWorker调用阶段参数core决定stage1:corePoolSize/stage3:maximumPoolSize)，
满足条件后，自旋CAS修改ctl线程池线程数量(并发修改失败、线程池状态改变，故需要进行自旋修改和条件判断)，成功后进入Worker创建过程(加全局重入锁，因为hashset中加worker是线程不安全)，最后启动worker中的线程.

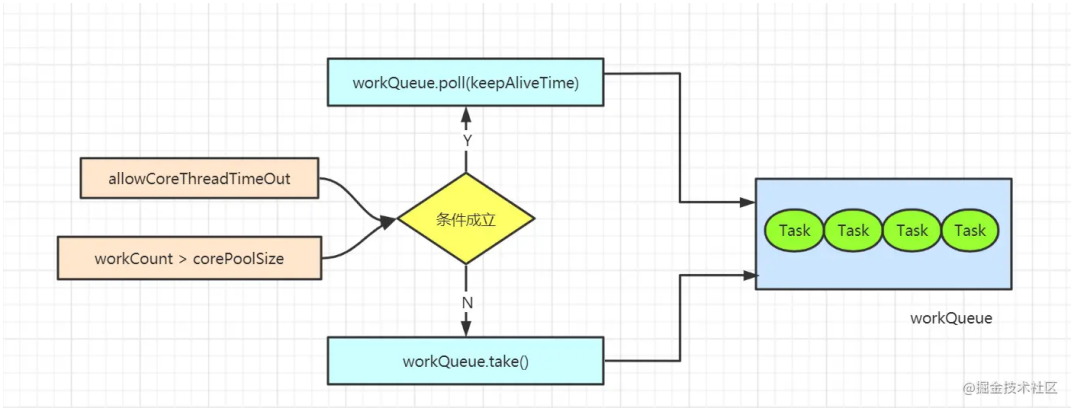
- runWorker



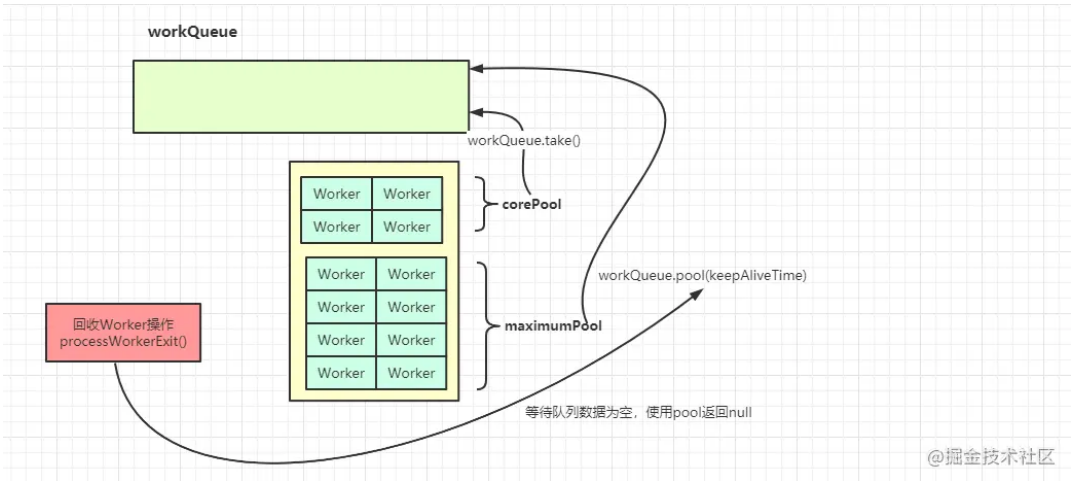


在addWorker成功添加work并启动线程后，会调用runWorker方法来执行task，主要过程是先执行firstTask任务，执行完毕后不断调用getTask函数从阻塞队列中获取任务进行执行．获取到非空task后会先判断当前线程池和线程状态是否合法，不合法则中断线程；否则执行当前task，执行task的前后会调用函数beforeExecute/afterExecute(可扩展方法)．如果getTask函数没有获取到task，则会返回null，此时会跳出不断获取和执行task循环，进入当前worker的清理函数processWorkerExit．

- getTask



getTask从阻塞队列获取task策略如下，**如果当前线程数量大于corePoolSize(存在非核心线程)**，因此当前线程使用**poll**(keepAliveTime)从队列获取task，如果在keepAliveTime内未获取到task，则返回null(进入回收当前空闲worker)；如果当前线程数量小于等于corePoolSize(均为核心线程)，因此要先判断用户设置的参数**allowCoreThreadTimeOut**来判断核心线程是否可回收，如为**true**，则也调用**poll**来获取task，否则调用**take**来获取task，阻塞队列为空，take函数获取不到task会挂起等待，而不会返回null．



- processWorkerExit

此方法的含义是清理当前线程，从线程池中移除掉刚刚添加的worker对象．

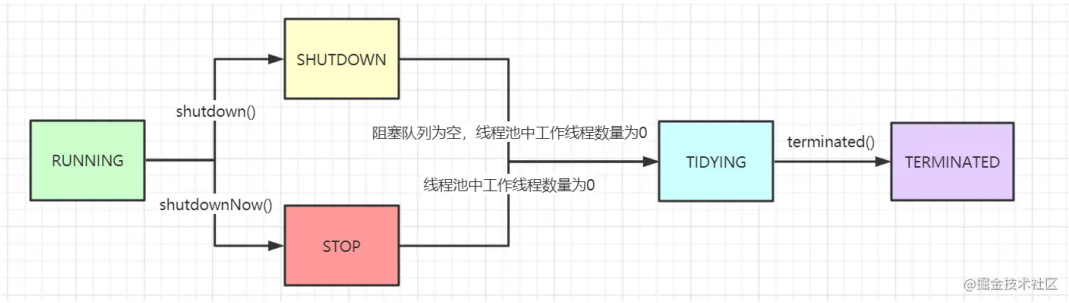
- tips
  - ThreadPoolExecutor常用参数
    - **int** corePoolSize（核心线程数量上界）
    - int maximumPoolSize(线程数量上界)
    - long keepAliveTime, TimeUnit unit(worker中poll操作从阻塞队列中等待task的最大时常，时间单位)
    - BlockingQueue workQueue(传入的task阻塞队列)
    - ThreadFactory threadFactory(用户自定义的线程工厂，便于设置统一的线程id前缀等用户参数)
    - RejectedExecutionHandler handler(拒绝执行的对象)

      AbortPolicy：默认策略，直接抛出异常.

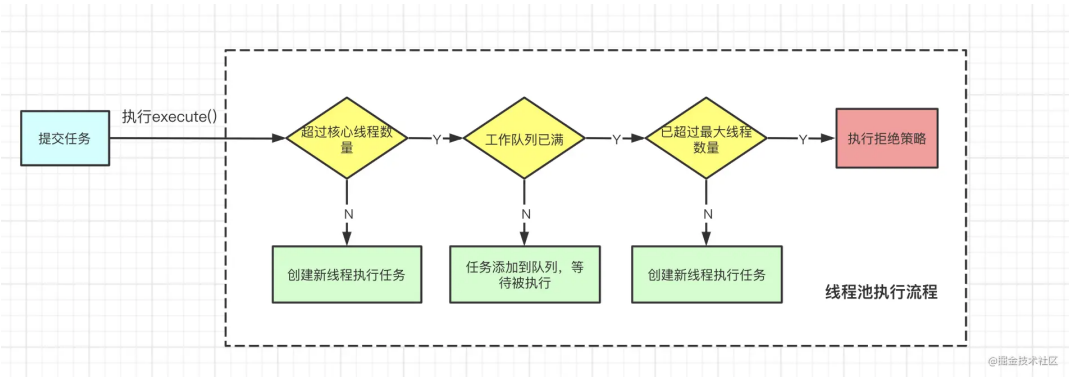
      CallerRunsPolicy：用调用者所在线程来执行当前任务.

      DiscardPolicy：直接丢弃当前任务.

      DiscardOldestPolicy：丢弃阻塞队列头，再次execute任务.
  - 线程池状态/数量属性和设置
    - ctl(AtomicInteger类型变量，高3位记录线程池状态，低28位记录线程数量)
    - corePoolSize、maximumPoolSize初始化时设置核心线程数量和最大线程数量
  - 线程池状态种类/转换



  - 线程池任务处理策略/底层原理



  - 线程池常用拒绝策略

    AbortPolicy：默认策略，直接抛出异常.

    CallerRunsPolicy：用调用者所在线程来执行当前任务.

    DiscardPolicy：直接丢弃当前任务.

    DiscardOldestPolicy：丢弃阻塞队列头，再次execute任务.
  - 线程池线程如何复用

    runWorker函数循环调用getTask函数从阻塞队列中获取任务task进行执行.
  - 线程池可扩展

    - beforeExecute()/afterExecute()（runWorker()中线程执行前和执行后会调用的钩子方法）
    - terminated（线程池的状态从TIDYING状态流转为TERMINATED状态时terminated方法会被调用的钩子方法）
    - onShutdown（执行shutdown()方法时预留的钩子方法）
- 线程池预热

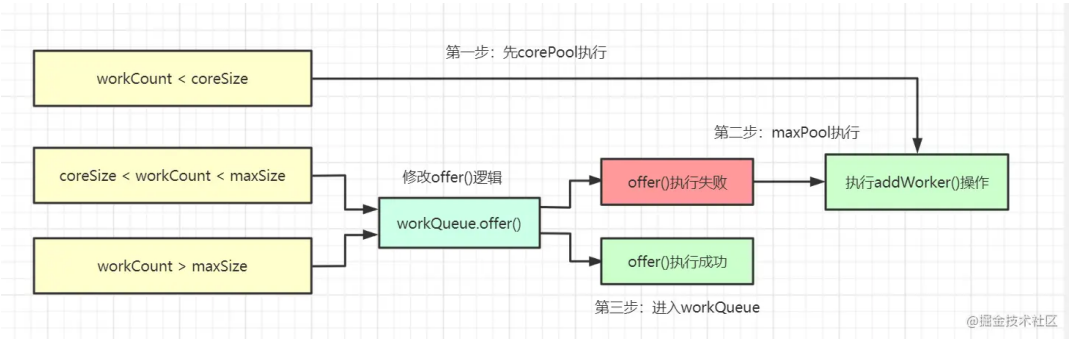
  prestartAllCoreThreads()（循环调用addWorker函数添加核心线程）.
- 线程池支持动态调整
    - setCorePoolSize()
    - setMaximumPoolSize()
    - setKeepAliveTime()
    - allowsCoreThreadTimeOut()
- 线程池线程回收

  非核心空闲线程等待超时或者核心空闲线程设置了可回收时等待超时，poll方法返回null调用processWorkerExit进行回收.
- 线程池关闭方法
    - shutdown()（将线程池状态变为SHUTDOWN，此时新任务不能被提交，workQueue中还存有的任务可以继续执行，同时会像线程池中空闲的状态发出中断信号）
    - shutdownNow()（将线程池的状态设置为STOP，此时新任务不能被提交，线程池中所有线程都会收到中断的信号，如果线程处于wait状态，那么中断状态会被清除，同时抛出InterruptedException）
- Executors线程池种类
    - FixedThreadPool 和 SingleThreadPool（允许的请求队列长度为Integer.MAX_VALUE，可能会堆积大量的请求，从而导致OOM）
    - CachedThreadPool（允许的创建线程数量为Integer.MAX_VALUE，可能会创建大量的线程，从而导致OOM）
- 线程池配置
    - CPU密集任务 threadNums = Ncpu+1
    - IO密集任务 threadNums = 2*Ncpu
    - 混合型任务 拆分CPU + IO
    - threadNums = (thread_IO_time + thread_cpu_time)/thread_cpu_time * Ncpu
- 线程池如何按照core、max、queue的顺序去执行（定制IO线程池）

  1、问题描述：[redacted]辑比较复杂，为了提高响应时间，我会把处理时间较长的业务放在异步线程池，但是响应时间未得到有效改善；

  2、原因分析：该线程池所执行的任务类型 I/O 密集型占大比例，CPU 大多处于空闲，系统资源未充分利用；

  3、解决思路：定制 I/O 线程池，来满足当前业务需求，主要改动：改写线程池入队列逻辑，强制在未达到最大线程数之前(当前线程数已大于核心线程数),优先创建线程而不是入队列。

dubbo中继承LinkedBlockingQueue自定义了一个TaskQueue，包含线程池对象成员，初始化时参数传
入当前线程池实例，重写offer时比较当前线程和maximumPoolSize关系，线程数不到时，直接offer失败，以便进入第三阶段创建新worker.

- **参考资料**
  - 【万字图文-原创】 | 学会Java中的线程池，这一篇也许就够了！ - 掘金 (juejin.cn)