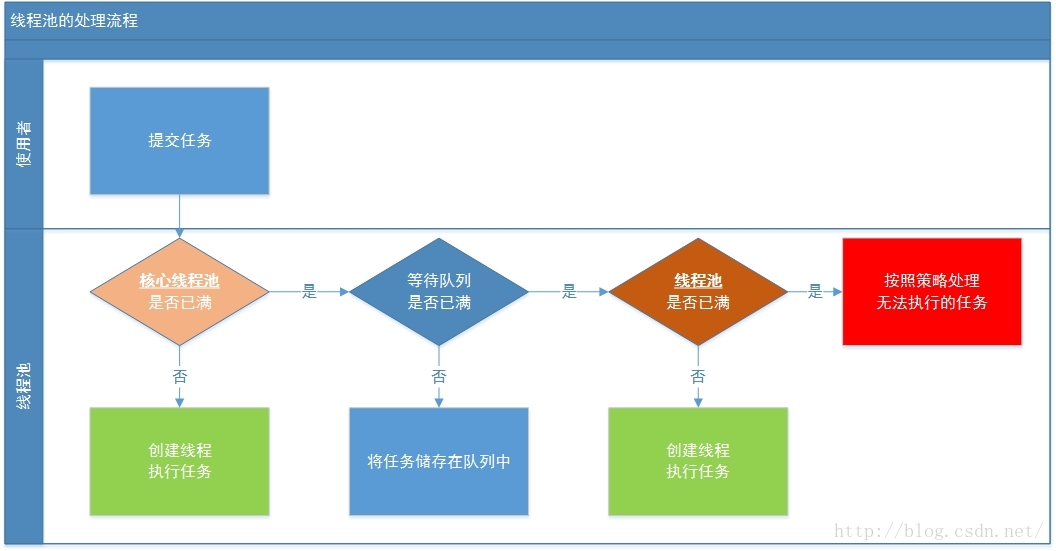
# Java中的线程池——ThreadPoolExecutor的原理

1 线程池的处理流程

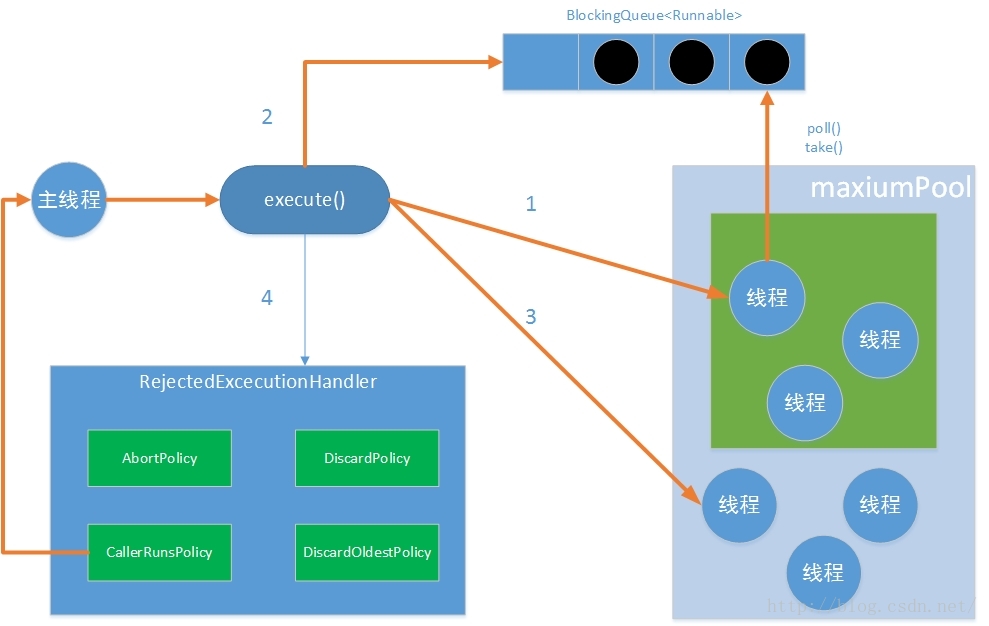
向线程池提交一个任务后，它的主要处理流程如下图所示



一个线程从被提交（submit）到执行共经历以下流程：

* 线程池判断核心线程池里是的线程是否都在执行任务，如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下一个流程
* 线程池判断工作队列是否已满。如果工作队列没有满，则将新提交的任务储存在这个工作队列里。如果工作队列满了，则进入下一个流程。
* 线程池判断其内部线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已满了，则交给饱和策略来处理这个任务。

线程池在执行excute方法时，主要有以下四种情况



1 如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（需要获得全局锁）

2 如果运行的线程等于或多于corePoolSize ,则将任务加入BlockingQueue

3 如果无法将任务加入BlockingQueue(队列已满)，则创建新的线程来处理任务（需要获得全局锁）

4 如果创建新线程将使当前运行的线程超出maxiumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

线程池采取上述的流程进行设计是为了减少获取全局锁的次数。在线程池完成预热（当前运行的线程数大于或等于corePoolSize）之后，几乎所有的excute方法调用都执行步骤2。

# 2 线程池的源码分析

## 2.1 定义的几个变量

1. private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
2. private static final int COUNT\_BITS = Integer.SIZE - 3;
3. private static final int CAPACITY = (1 << COUNT\_BITS) - 1;
5. *// runState is stored in the high-order bits*
6. private static final int RUNNING = -1 << COUNT\_BITS;
7. private static final int SHUTDOWN = 0 << COUNT\_BITS;
8. private static final int STOP = 1 << COUNT\_BITS;
9. private static final int TIDYING = 2 << COUNT\_BITS;
10. private static final int TERMINATED = 3 << COUNT\_BITS;
12. *// Packing and unpacking ctl*
13. private static int runStateOf(int c) { return c & ~CAPACITY; }
14. private static int workerCountOf(int c) { return c & CAPACITY; }
15. private static int ctlOf(int rs, int wc) { return rs | wc; }

在分析源码前有必要理解一个变量ctl。这是Java大神们为了把工作线程数量和线程池状态放在一个int类型变量里存储而设置的一个原子类型的变量。 在ctl中，低位的29位表示工作线程的数量，高位用来表示RUNNING、SHUTDOWN、STOP等状态。 因此一个线程池的数量也就变成了(2^29)-1，大约500 million，而不是(2^31)-1，2billion。上面定义的三个方法只是为了计算得到线程池的状态和工作线程的数量。

## 2.2 Execute 方法提交任务

### 2.2.1 Execute方法

1. public void execute(Runnable command) {
2. *//如果提交了空的任务 抛出异常*
3. if (command == null)
4. throw new NullPointerException();
5. int c = ctl.get();*//获取当前线程池的状态*
6. *//检查当前工作线程数量是否小于核心线程数量*
7. if (workerCountOf(c) < corePoolSize) {
8. *//通过addWorker方法提交任务*
9. if (addWorker(command, true))
10. return;
11. c = ctl.get();*//如果提交失败 需要二次检查状态*
12. }
13. *//向工作线程提交任务*
14. if (isRunning(c) && workQueue.offer(command)) {
15. *// 再次检查状态*
16. int recheck = ctl.get();
18. if (! isRunning(recheck) && remove(command))
19. reject(command);
20. else if (workerCountOf(recheck) == 0)
21. addWorker(null, false);
22. }
23. *//扩容失败 则拒绝任务*
24. else if (!addWorker(command, false))
25. reject(command);
26. }

从源码中可以看到提交任务的这一过程基本与第二个图的四个流程是一致的，需要检查的是当前工作线程的数量与核心线程数量的关系，来决定提交任务的方式或者是拒绝任务提交。而具体任务的提交工作是在addWorker方法中。在这里面看到了recheck这样的变量，这是在执行了一些动作失败后再次检查线程池的状态，因为在这期间可能有线程池关闭获得线程池饱和等状态的改变。

### 2.2.2 addWorker方法

这个方法是任务提交的一个核心方法。在里面完成了状态检查、新建任务、执行任务等一系列动作。它有两个参数，第一个参数是提交的任务，第二个参数是一个标识符，标识在检查工作线程数量的时候是应该与corePoolSize对比还是应该maximumPoolSize对比。

1. private boolean addWorker(Runnable firstTask, boolean core) {
2. retry:
3. *//死循环更新状态*
4. for (;;) {
5. int c = ctl.get();
6. int rs = runStateOf(c);*//获取运行状态*
8. *//检查线程池是否处于关闭状态*
9. if (rs >= SHUTDOWN &&
10. ! (rs == SHUTDOWN &&
11. firstTask == null &&
12. ! workQueue.isEmpty()))
13. return false;
15. for (;;) {
16. *//获取当前工作线程数量*
17. int wc = workerCountOf(c);
18. *//如果已经超过corePoolSize获取maximumPoolSize 返回false*
19. if (wc >= CAPACITY ||
20. wc >= (core ? corePoolSize : maximumPoolSize))
21. return false;
22. *//CAS增加一个工作线程*
23. if (compareAndIncrementWorkerCount(c))
24. break retry;
25. *//再次获取状态*
26. c = ctl.get(); *// Re-read ctl*
27. *//如果状态更新失败 则循环更新*
28. if (runStateOf(c) != rs)
29. continue retry;
30. *// else CAS failed due to workerCount change; retry inner loop*
31. }
32. }
34. boolean workerStarted = false;
35. boolean workerAdded = false;
36. Worker w = null;
37. try {
38. w = new Worker(firstTask);*//初始化一个工作线程*
39. final Thread t = w.thread;
40. if (t != null) {
41. *//获得锁*
42. final ReentrantLock mainLock = this.mainLock;
43. mainLock.lock();
44. try {
45. *// Recheck while holding lock.*
46. *// Back out on ThreadFactory failure or if*
47. *// shut down before lock acquired.*
48. int rs = runStateOf(ctl.get());
50. if (rs < SHUTDOWN ||
51. (rs == SHUTDOWN && firstTask == null)) {
52. if (t.isAlive()) *// precheck that t is startable*
53. throw new IllegalThreadStateException();
54. *//添加工作这到hashset中保存*
55. workers.add(w);
56. int s = workers.size();
57. if (s > largestPoolSize)
58. largestPoolSize = s;
59. workerAdded = true;
60. }
61. } finally {
62. mainLock.unlock();
63. }
64. if (workerAdded) {
65. *//工作线程启动 执行第一个任务 就是新提交的任务*
66. t.start();
67. workerStarted = true;
68. }
69. }
70. } finally {
71. if (! workerStarted)
72. addWorkerFailed(w);
73. }
74. return workerStarted;
75. }

这个方法可以分为两个阶段来看，第一个阶段是判断是否有必要新增一个工作线程，如果有则利用CAS更新工作线程的数量；第二部分是将提交的任务封装成一个工作线程Worker然后加入到线程池的容器中，开始执行新提交的任务。这个Worker在执行完任务后，还会循环地获取工作队列里的任务来执行。下面来看一下Worker的构造方法就能更好地理解上面的代码了

1. */\*\**
2. *\* Creates with given first task and thread from ThreadFactory.*
3. *\* @param firstTask the first task (null if none)*
4. *\*/*
5. Worker(Runnable firstTask) {
6. setState(-1); *// inhibit interrupts until runWorker*
7. this.firstTask = firstTask;
8. this.thread = getThreadFactory().newThread(this);
9. }

### 2.2.3 runWorker方法

在addWorker方法快要结束的地方，调用了t.start()方法，我们知道它实际执行的就是Worker对象的run()方法，而worker的run()方法是这样定义的：

1. */\*\* Delegates main run loop to outer runWorker \*/*
2. public void run() {
3. runWorker(this);
4. }

它实际上是将自己委托给线程池的runWorker方法

1. final void runWorker(Worker w) {
3. Thread wt = Thread.currentThread();
4. Runnable task = w.firstTask;
5. w.firstTask = null;
6. w.unlock(); *// allow interrupts*
7. boolean completedAbruptly = true;
8. try {
9. *//不断地从blockingQueue获取任务*
10. while (task != null || (task = getTask()) != null) {
11. w.lock();
12. *// If pool is stopping, ensure thread is interrupted;*
13. *// if not, ensure thread is not interrupted. This*
14. *// requires a recheck in second case to deal with*
15. *// shutdownNow race while clearing interrupt*
16. if ((runStateAtLeast(ctl.get(), STOP) ||
17. (Thread.interrupted() &&
18. runStateAtLeast(ctl.get(), STOP))) &&
19. !wt.isInterrupted())
20. wt.interrupt();
21. try {
22. *//执行beforeExecute方法*
23. beforeExecute(wt, task);
24. Throwable thrown = null;
25. try {
26. *//调用Runable的run方法*
27. task.run();
28. } catch (RuntimeException x) {
29. thrown = x; throw x;
30. } catch (Error x) {
31. thrown = x; throw x;
32. } catch (Throwable x) {d
33. thrown = x; throw new Error(x);
34. } finally {
35. *// 执行aferExecute方法*
36. afterExecute(task, thrown);
37. }
38. } finally {
39. task = null;
40. w.completedTasks++;
41. w.unlock();
42. }
43. }
44. completedAbruptly = false;
45. } finally {
46. processWorkerExit(w, completedAbruptly);
47. }
48. }

这个方法呢也比较好理解，它在不断执行我们提交的任务的run方法。而这个任务可能是我们新提交的，也有可能是从等待队列中获取的。这样就实现了线程池的完成逻辑。

1438\*(4.45\*(4.45+3.91)\*2+3.91\*4.45\*2+4.45\*4.45\*3.91)