Java 线程池

1. Java 线程池构架

1. Executor 是最基础的执行接口；
2. ExecutorService 接口继承了 Executor，在其上做了一些 shutdown()、submit()的扩展，可以说是真正的线程池接口；
3. AbstractExecutorService 抽象类实现了 ExecutorService 接口中的大部分方法；
4. TheadPoolExecutor 继承了 AbstractExecutorService，是线程池的具体实现
5. ScheduledExecutorService 接口继承了 ExecutorService 接口，提供了带"周期执行"功能 ExecutorService；
6. ScheduledThreadPoolExecutor 既继承了 TheadPoolExecutor 线程池，也实现了 ScheduledExecutorService 接口，是带"周期执行"功能的线程池；
7. Executors 是线程池的静态工厂，其提供了快捷创建线程池的静态方法。

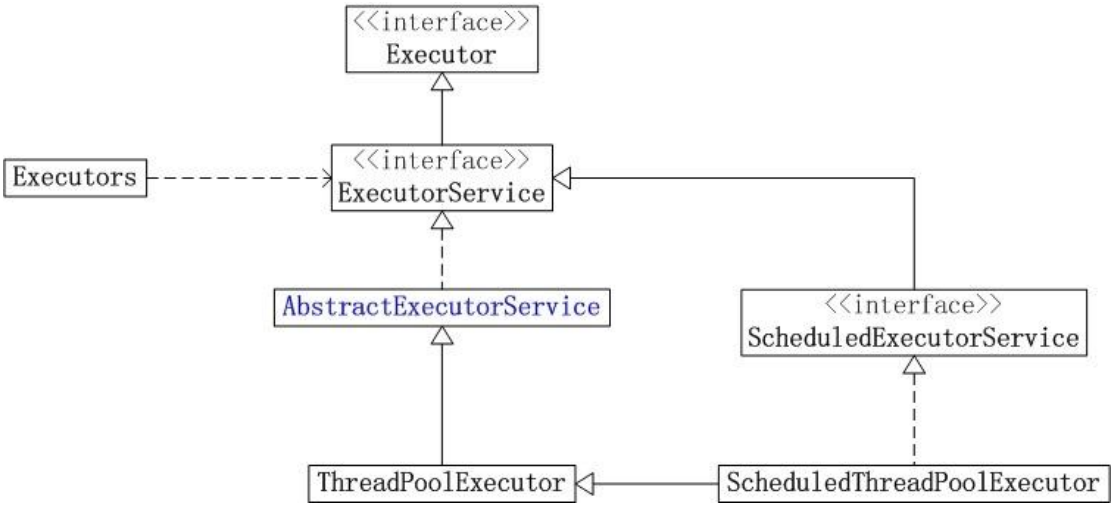


Figure 1 java **线程池构架**

1.1. Executor 接口

“执行者”接口，只提供了一个方法：void execute(Runnable command); 可以用来执行已经提交的 Runnable 任务对象，这个接口提供了一种将“任务提交”与“任务执行”解耦的方法。

1.2. ExecutorService 接口

“执行者服务”接口，可以说是真正的线程池接口，在 Executor 接口的基础上做了一些扩展，主要是：

(A) **管理任务如何终止的 shutdown 相关方法**

```
1. /**
```

```
     * 启动一次有序的关闭，之前提交的任务执行，但不接受新任务
     * 这个方法不会等待之前提交的任务执行完毕
     */
    void shutdown();

    /**
     * 试图停止所有正在执行的任务，暂停处理正在等待的任务，返回一个等待执行的任务列表
     * 这个方法不会等待正在执行的任务终止
     */
    List<Runnable> shutdownNow();

    /**
     * 如果已经被 shutdown，返回 true
     */
    boolean isShutdown();

    /**
     * 如果所有任务都已经被终止，返回 true
     * 是否为终止状态
     */
    boolean isTerminated();

    /**
     * 在一个 shutdown 请求后，阻塞的等待所有任务执行完毕
     * 或者到达超时时间，或者当前线程被中断
     */
    boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedExce
    ption;
```

(B) 可以生成用于追踪一个或多个异步任务执行结果的 Future 对象的 submit()相关方法

```
    /**
     * 提交一个可执行的任务，返回一个 Future 代表这个任务
     * 等到任务成功执行，Future#get()方法会返回 null
     */
    Future<?> submit(Runnable task);

    /**
     * 提交一个可以执行的任务，返回一个 Future 代表这个任务
     * 等到任务执行结束，Future#get()方法会返回这个给定的 result
     */
    <T> Future<T> submit(Runnable task, T result);

    /**
```

```
14.  * 提交一个有返回值的任务，并返回一个 Future 代表等待的任务执行的结果
15.  * 等到任务成功执行，Future#get()方法会返回任务执行的结果
16.  */
17. <T> Future<T> submit(Callable<T> task);
```

1.3. ScheduledExecutorService 接口

```
1.  /**
2.   * 提交一个可执行的任务，返回一个 Future 代表这个任务
3.   * 等到任务成功执行，Future#get()方法会返回 null
4.   */
5.  Future<?> submit(Runnable task);
6.
7.  /**
8.   * 提交一个可以执行的任务，返回一个 Future 代表这个任务
9.   * 等到任务执行结束，Future#get()方法会返回这个给定的 result
10.  */
11. <T> Future<T> submit(Runnable task, T result);
12.
13. /**
14.  * 提交一个有返回值的任务，并返回一个 Future 代表等待的任务执行的结果
15.  * 等到任务成功执行，Future#get()方法会返回任务执行的结果
16.  */
17. <T> Future<T> submit(Callable<T> task);
```

1. ThreadPoolExecutor

1.4. ThreadPoolExecutor 构造参数

```
1.  public ThreadPoolExecutor(int corePoolSize,
2.                            int maximumPoolSize,
3.                            long keepAliveTime,
4.                            TimeUnit unit,
5.                            BlockingQueue<Runnable> workQueue,
6.                            ThreadFactory threadFactory,
7.                            RejectedExecutionHandler handler)
```

corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于 corePoolSize；

1. 如果当前线程数为 corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

2. 如果执行了线程池的 prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于 maximumPoolSize

keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于 corePoolSize 时才有用

workQueue

workQueue 必须是 BlockingQueue 阻塞队列。当线程池中的线程数超过它的 corePoolSize 的时候，线程会进入阻塞队列进行阻塞等待。通过 workQueue，线程池实现了阻塞功能

几种排队的策略：

（1）不排队，直接提交

1. 将任务直接交给线程处理而不保持它们，可使用 SynchronousQueue
2. 如果不存在可用于立即运行任务的线程（即线程池中的线程都在工作），则试图把任务加入缓冲队列将会失败，因此会构造一个新的线程来处理新添加的任务，并将其加入到线程池中（corePoolSize-->maximumPoolSize 扩容）
3. Executors.newCachedThreadPool()采用的便是这种策略

（2）无界队列

1. 可以使用 LinkedBlockingQueue（基于链表的有界队列，FIFO），理论上是该队列可以对无限多的任务排队，将导致在所有 corePoolSize 线程都工作的情况下将新任务加入到队列中。这样，创建的线程就不会超过 corePoolSize，也因此，maximumPoolSize 的值也就无效了

（3）有界队列

1. 可以使用 ArrayBlockingQueue（基于数组结构的有界队列，FIFO），并指定队列的最大长度
2. 使用有界队列可以防止资源耗尽，但也会造成超过队列大小和 maximumPoolSize 后，提交的任务被拒绝的问题，比较难调整和控制。

threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名

```
1. /**
2.  * The default thread factory
3.  */
4. static class DefaultThreadFactory implements ThreadFactory {
5.     private static final AtomicInteger poolNumber = new AtomicInteger(1);
```

```
6.      private final ThreadGroup group;
7.      private final AtomicInteger threadNumber = new AtomicInteger(1);
8.      private final String namePrefix;
9.
10.     DefaultThreadFactory() {
11.         SecurityManager s = System.getSecurityManager();
12.         group = (s != null) ? s.getThreadGroup() :
13.                               Thread.currentThread().getThreadGroup();
14.         namePrefix = "pool-" +
15.                       poolNumber.getAndIncrement() +
16.                      "-thread-";
17.     }
18.
19.     public Thread newThread(Runnable r) {
20.         Thread t = new Thread(group, r,
21.                               namePrefix + threadNumber.getAndIncrement(),
22.                               0);
23.         if (t.isDaemon())
24.             t.setDaemon(false);
25.         if (t.getPriority() != Thread.NORM_PRIORITY)
26.             t.setPriority(Thread.NORM_PRIORITY);
27.         return t;
28.     }
29. }
```

Executors 静态工厂里默认的 threadFactory，线程的命名规则是“pool-数字-thread-数字”

RejectedExecutionHandler（饱和策略）

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了 4 种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现 RejectedExecutionHandler 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

1.5. ThreadPoolExecutor 线程池执行流程

根据 ThreadPoolExecutor 源码前面大段的注释，我们可以看出，当试图通过 execute 方法将一个 Runnable 任务添加到线程池中时，按照如下顺序来处理：

（1）如果线程池中的线程数量少于 corePoolSize，就创建新的线程来执行新添加的任务；

（2）如果线程池中的线程数量大于等于 corePoolSize，但队列 workQueue 未满，则将新添加的任务放到 workQueue 中，按照 FIFO 的原则依次等待执行（线程池中有线程空闲出来后依次将队列中的任务交付给空闲的线程执行）；

（3）如果线程池中的线程数量大于等于 corePoolSize，且队列 workQueue 已满，但线程池中的线程数量小于 maximumPoolSize，则会创建新的线程来处理被添加的任务；

（4）如果线程池中的线程数量等于了 maximumPoolSize，就用 RejectedExecutionHandler 来做拒绝处理

总结，当有新的任务要处理时，先看线程池中的线程数量是否大于 corePoolSize，再看缓冲队列 workQueue 是否满，最后看线程池中的线程数量是否大于 maximumPoolSize

另外，当线程池中的线程数量大于 corePoolSize 时，如果里面有线程的空闲时间超过了 keepAliveTime，就将其移除线程池

2. Executors 静态工厂创建几种常用线程池

```
1.  public static ExecutorService newFixedThreadPool(int nThreads) {
2.      return new ThreadPoolExecutor(nThreads, nThreads,
3.                                    0L, TimeUnit.MILLISECONDS,
4.                                    new LinkedBlockingQueue<Runnable>());
5.  }
6.
7.  public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory thr
    eadFactory) {
8.      return new ThreadPoolExecutor(nThreads, nThreads,
9.                                    0L, TimeUnit.MILLISECONDS,
10.                                   new LinkedBlockingQueue<Runnable>(),
11.                                   threadFactory);
12. }
```

newFixedThreadPool 创建一个指定工作线程数的线程池，其中参数 corePoolSize 和 maximumPoolSize 相等，阻塞队列基于 LinkedBlockingQueue

它是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是在线程池空闲时，即线程池中没有可运行任务时，它也不会释放工作线程，还会占用一定的系统资源

```
1.  public static ExecutorService newSingleThreadExecutor() {
2.      return new FinalizableDelegatedExecutorService
3.          (new ThreadPoolExecutor(1, 1,
4.                                  0L, TimeUnit.MILLISECONDS,
5.                                  new LinkedBlockingQueue<Runnable>()));
```

```
}

public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>(),
                                threadFactory));
}
```

newSingleThreadExecutor 初始化的线程池中只有一个线程，如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行，内部使用 LinkedBlockingQueue 作为阻塞队列。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}

public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>(),
                                  threadFactory);
}
```

newCachedThreadPool 创建一个可缓存工作线程的线程池，默认存活时间 60 秒，线程池的线程数可达到 Integer.MAX_VALUE，即 2147483647，内部使用 SynchronousQueue 作为阻塞队列；

在没有任务执行时，当线程的空闲时间超过 keepAliveTime，则工作线程将会终止，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {
    return new ScheduledThreadPoolExecutor(corePoolSize);
}

public static ScheduledExecutorService newScheduledThreadPool(
        int corePoolSize, ThreadFactory threadFactory) {
```

```
7.      return new ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
8.  }
```

newScheduledThreadPool 初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据

注意：

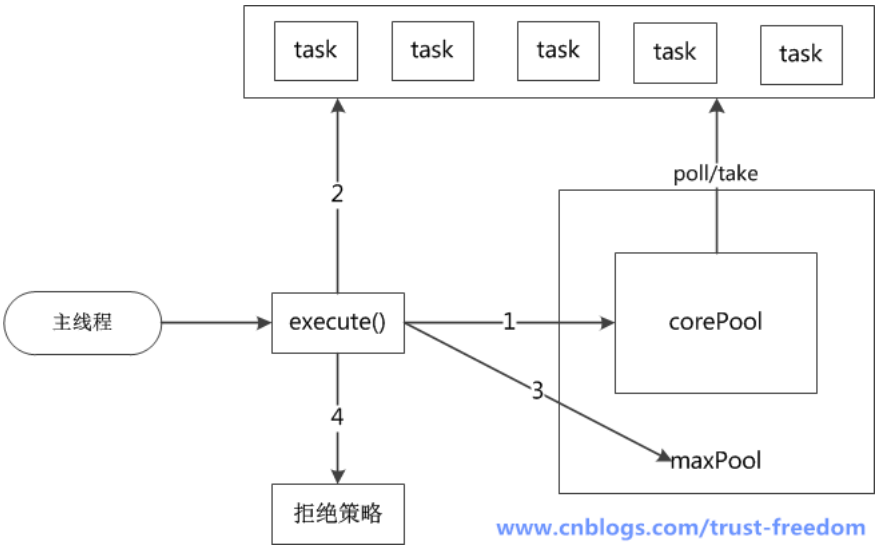
ScheduledExecutorService#scheduleAtFixedRate() 指的是“以固定的频率”执行，period（周期）指的是两次成功执行之间的时间

比如，scheduleAtFixedRate(command, 5, 2, second)，第一次开始执行是 5s 后，假如执行耗时 1s，那么下次开始执行是 7s 后，再下次开始执行是 9s 后

而 ScheduledExecutorService#scheduleWithFixedDelay() 指的是“以固定的延时”执行，delay（延时）指的是一次执行终止和下一次执行开始之间的延迟

还是上例，scheduleWithFixedDelay(command, 5, 2, second)，第一次开始执行是 5s 后，假如执行耗时 1s，执行完成时间是 6s 后，那么下次开始执行是 8s 后，再下次开始执行是 11s 后

3. 线程池执行流程



1、如果线程池中的线程数量少于 corePoolSize，就创建新的线程来执行新添加的任务

2、如果线程池中的线程数量大于等于 corePoolSize，但队列 workQueue 未满，则将新添加的任务放到 workQueue 中

3、如果线程池中的线程数量大于等于 corePoolSize，且队列 workQueue 已满，但线程池中的线程数量小于 maximumPoolSize，则会创建新的线程来处理被添加的任务

4、如果线程池中的线程数量等于了 maximumPoolSize，就用 RejectedExecutionHandler 来执行拒绝策略

4. 线程池状态

```
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;
4.
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;
11.
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; }
```

其中 ctl 这个 AtomicInteger 的功能很强大，其高 3 位用于维护线程池运行状态，低 29 位维护线程池中线程数量

1、 RUNNING：-1<<COUNT_BITS，即高 3 位为 1，低 29 位为 0，该状态的线程池会接收新任务，也会处理在阻塞队列中等待处理的任务

2、 SHUTDOWN：0<<COUNT_BITS，即高 3 位为 0，低 29 位为 0，该状态的线程池不会再接收新任务，但还会处理已经提交到阻塞队列中等待处理的任务

3、 STOP：1<<COUNT_BITS，即高 3 位为 001，低 29 位为 0，该状态的线程池不会再接收新任务，不会处理在阻塞队列中等待的任务，而且还会中断正在运行的任务

4、 TIDYING：2<<COUNT_BITS，即高 3 位为 010，低 29 位为 0，所有任务都被终止了，workerCount 为 0，为此状态时还将调用 terminated()方法

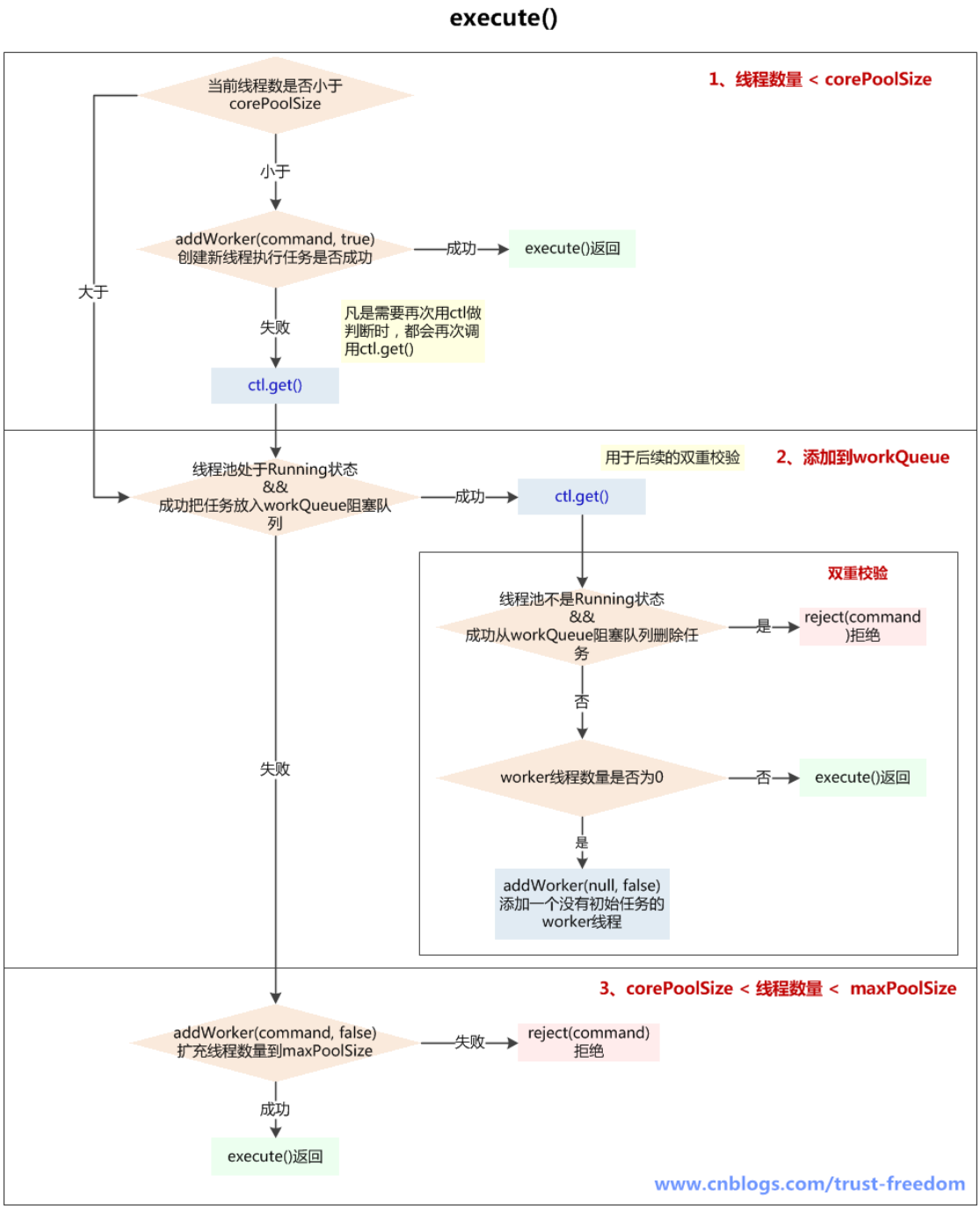
5、 TERMINATED：3<<COUNT_BITS，即高 3 位为 100，低 29 位为 0，terminated()方法调用完成后变成此状态

这些状态均由 int 型表示，大小关系为 RUNNING<SHUTDOWN<STOP<TIDYING<TERMINATED，这个顺序基本上也是遵循线程池从 运行 到 终止这个过程。

1. runStateOf(int c) 方法：c & 高 3 位为 1，低 29 位为 0 的~CAPACITY，用于获取高 3 位保存的线程池状态
2. workerCountOf(int c)方法：c & 高 3 位为 0，低 29 位为 1 的 CAPACITY，用于获取低 29 位的线程数量
3. ctlOf(int rs, int wc)方法：参数 rs 表示 runState，参数 wc 表示 workerCount，即根据 runState 和 workerCount 打包合并成 ctl

5. 任务提交内部原理

5.1. execute() -- 提交任务




```
1.  /**
2.   * Executes the given task sometime in the future.  The task
3.   * may execute in a new thread or in an existing pooled thread.
```

```
4.   * 在未来的某个时刻执行给定的任务。这个任务用一个新线程执行，或者用一个线程池中已经存在的线程执行
5.   *
6.   * If the task cannot be submitted for execution, either because this
7.   * executor has been shutdown or because its capacity has been reached,
8.   * the task is handled by the current {@code RejectedExecutionHandler}.
9.   * 如果任务无法被提交执行，要么是因为这个 Executor 已经被 shutdown 关闭，要么是已经达到其容量上限，任务会被当前的 RejectedExecutionHandler 处理
10.  *
11.  * @param command the task to execute
12.  * @throws RejectedExecutionException at discretion of
13.  *         {@code RejectedExecutionHandler}, if the task
14.  *         cannot be accepted for execution                RejectedExecutionException 是一个 RuntimeException
15.  * @throws NullPointerException if {@code command} is null
16.  */
17. public void execute(Runnable command) {
18.     if (command == null)
19.         throw new NullPointerException();
20.
21.     /*
22.      * Proceed in 3 steps:
23.      *
24.      * 1. If fewer than corePoolSize threads are running, try to
25.      * start a new thread with the given command as its first
26.      * task.  The call to addWorker atomically checks runState and
27.      * workerCount, and so prevents false alarms that would add
28.      * threads when it shouldn't, by returning false.
29.      * 如果运行的线程少于 corePoolSize，尝试开启一个新线程去运行 command，command 作为这个线程的第一个任务
30.      *
31.      * 2. If a task can be successfully queued, then we still need
32.      * to double-check whether we should have added a thread
33.      * (because existing ones died since last checking) or that
34.      * the pool shut down since entry into this method. So we
35.      * recheck state and if necessary roll back the enqueuing if
36.      * stopped, or start a new thread if there are none.
37.      * 如果任务成功放入队列，我们仍需要一个双重校验去确认是否应该新建一个线程（因为可能存在有些线程在我们上次检查后死了） 或者 从我们进入这个方法后，pool 被关闭了
38.      * 所以我们需要再次检查 state，如果线程池停止了需要回滚入队列，如果池中没有线程了，新开启 一个线程
39.      *
40.      * 3. If we cannot queue task, then we try to add a new
41.      * thread.  If it fails, we know we are shut down or saturated
```

```
42.      * and so reject the task.
43.      * 如果无法将任务入队列（可能队列满了），需要新开区一个线程（自己：往
    maxPoolSize 发展）
44.      * 如果失败了，说明线程池 shutdown 或者 饱和了，所以我们拒绝任务
45.      */
46.     int c = ctl.get();
47.
48.     /**
49.      * 1、如果当前线程数少于 corePoolSize（可能是由于 addWorker()操作已经包含对线
    程池状态的判断，如此处没加，而入 workQueue 前加了）
50.      */
51.     if (workerCountOf(c) < corePoolSize) {
52.         //addWorker()成功，返回
53.         if (addWorker(command, true))
54.             return;
55.
56.         /**
57.          * 没有成功 addWorker()，再次获取 c（凡是需要再次用 ctl 做判断时，都会再次
    调用 ctl.get()）
58.          * 失败的原因可能是：
59.          * 1、线程池已经 shutdown，shutdown 的线程池不再接收新任务
60.          * 2、workerCountOf(c) < corePoolSize 判断后，由于并发，别的线程先创建
    了 worker 线程，导致 workerCount>=corePoolSize
61.          */
62.         c = ctl.get();
63.     }
64.
65.     /**
66.      * 2、如果线程池 RUNNING 状态，且入队列成功
67.      */
68.     if (isRunning(c) && workQueue.offer(command)) {
69.         int recheck = ctl.get();//再次校验位
70.
71.         /**
72.          * 再次校验放入 workerQueue 中的任务是否能被执行
73.          * 1、如果线程池不是运行状态了，应该拒绝添加新任务，从 workQueue 中删除任
    务
74.          * 2、如果线程池是运行状态，或者从 workQueue 中删除任务失败（刚好有一个线程
    执行完毕，并消耗了这个任务），确保还有线程执行任务（只要有一个就够了）
75.          */
76.         //如果再次校验过程中，线程池不是 RUNNING 状态，并且 remove(command)--
    workQueue.remove()成功，拒绝当前 command
77.         if (! isRunning(recheck) && remove(command))
78.             reject(command);
```

```
79.         //如果当前 worker 数量为 0，通过 addWorker(null, false)创建一个线程，其任务为 null
80.         //为什么只检查运行的 worker 数量是不是 0 呢？？ 为什么不和 corePoolSize 比较呢？？
81.         //只保证有一个 worker 线程可以从 queue 中获取任务执行就行了？？
82.         //因为只要还有活动的 worker 线程，就可以消费 workerQueue 中的任务
83.         else if (workerCountOf(recheck) == 0)
84.             addWorker(null, false);  //第一个参数为 null，说明只为新建一个 worker 线程，没有指定 firstTask
85.                                      //第二个参数为 true 代表占用 corePoolSize，false 占用 maxPoolSize
86.     }
87.     /**
88.      * 3、如果线程池不是 running 状态 或者 无法入队列
89.      *   尝试开启新线程，扩容至 maxPoolSize，如果 addWork(command, false)失败了，拒绝当前 command
90.      */
91.     else if (!addWorker(command, false))
92.         reject(command);
```

execute(Runnable command)

参数：

command　　提交执行的任务，不能为空

执行流程：

1、如果线程池当前线程数量少于 corePoolSize，则 addWorker(command, true)创建新 worker 线程，如创建成功返回，如没创建成功，则执行后续步骤；

addWorker(command, true)失败的原因可能是：

A、线程池已经 shutdown，shutdown 的线程池不再接收新任务

B、workerCountOf(c) < corePoolSize 判断后，由于并发，别的线程先创建了 worker 线程，导致 workerCount>=corePoolSize

2、如果线程池还在 running 状态，将 task 加入 workQueue 阻塞队列中，如果加入成功，进行 double-check，如果加入失败（可能是队列已满），则执行后续步骤；

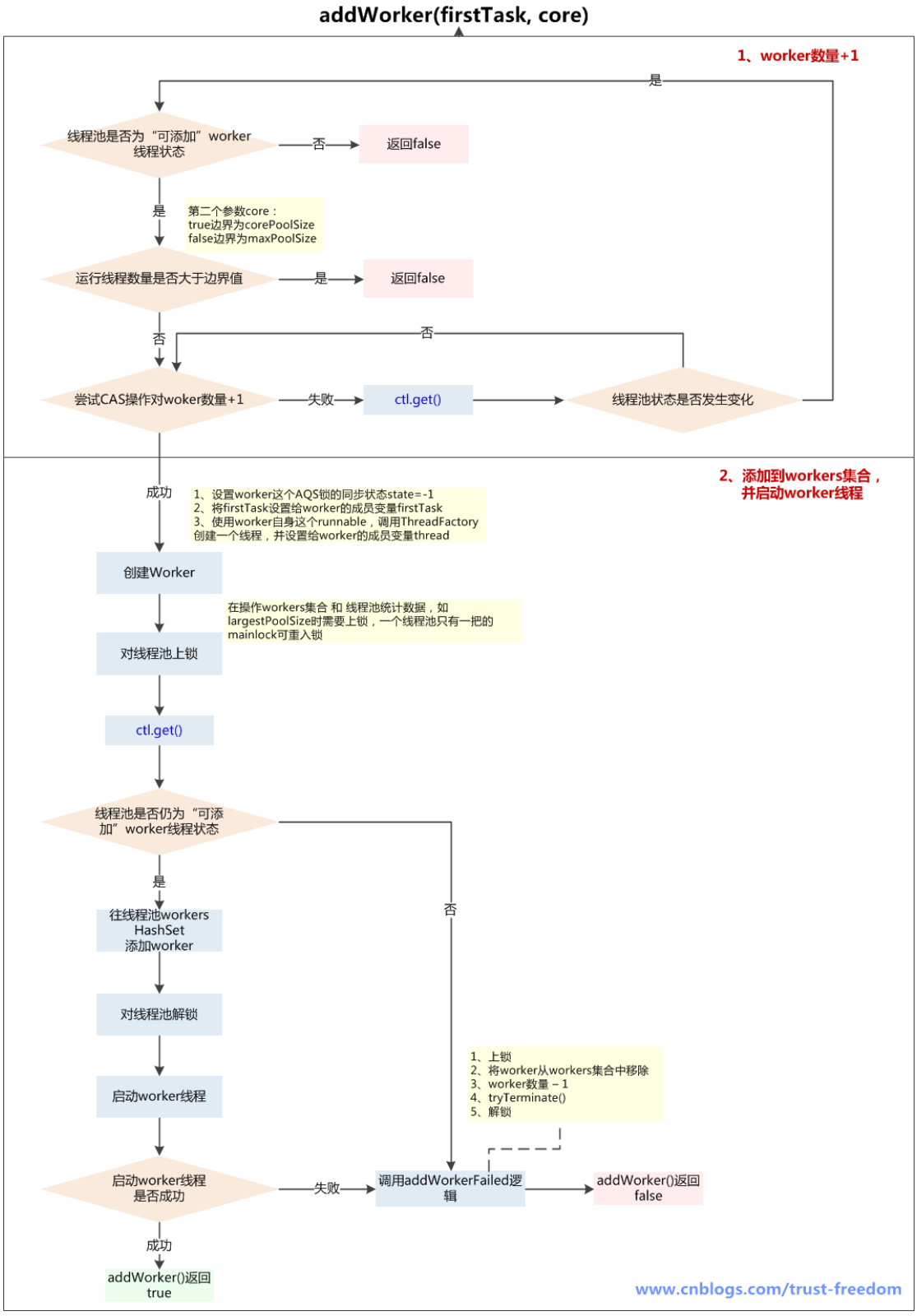
double-check 主要目的是判断刚加入 workQueue 阻塞队列的 task 是否能被执行

A、如果线程池已经不是 running 状态了，应该拒绝添加新任务，从 workQueue 中删除任务

B、如果线程池是运行状态，或者从 workQueue 中删除任务失败（刚好有一个线程执行完毕，并消耗了这个任务），确保还有线程执行任务（只要有一个就够了）

3、如果线程池不是 running 状态 或者 无法入队列，尝试开启新线程，扩容至 maxPoolSize，如果 addWork(command, false)失败了，拒绝当前 command

5.2. addWorker() -- 添加 worker 线程




```
1.  /**
2.   * Checks if a new worker can be added with respect to current
3.   * pool state and the given bound (either core or maximum). If so,
```

```
4.   * the worker count is adjusted accordingly, and, if possible, a
5.   * new worker is created and started, running firstTask as its
6.   * first task. This method returns false if the pool is stopped or
7.   * eligible to shut down. It also returns false if the thread
8.   * factory fails to create a thread when asked.  If the thread
9.   * creation fails, either due to the thread factory returning
10.  * null, or due to an exception (typically OutOfMemoryError in
11.  * Thread#start), we roll back cleanly.
12.  * 检查根据当前线程池的状态和给定的边界(core or maximum)是否可以创建一个新的
     worker
13.  * 如果是这样的话，worker 的数量做相应的调整，如果可能的话，创建一个新的 worker 并启
     动，参数中的 firstTask 作为 worker 的第一个任务
14.  * 如果方法返回 false，可能因为 pool 已经关闭或者调用过了 shutdown
15.  * 如果线程工厂创建线程失败，也会失败，返回 false
16.  * 如果线程创建失败，要么是因为线程工厂返回 null，要么是发生了 OutOfMemoryError
17.  *
18.  * @param firstTask the task the new thread should run first (or
19.  * null if none). Workers are created with an initial first task
20.  * (in method execute()) to bypass(绕开) queuing when there are fewer
21.  * than corePoolSize threads (in which case we always start one),
22.  * or when the queue is full (in which case we must bypass queue).
23.  * Initially idle threads are usually created via
24.  * prestartCoreThread or to replace other dying workers.
25.  *
26.  * @param core if true use corePoolSize as bound, else
27.  * maximumPoolSize. (A boolean indicator is used here rather than a
28.  * value to ensure reads of fresh values after checking other pool
29.  * state).
30.  * @return true if successful
31.  */
32. private boolean addWorker(Runnable firstTask, boolean core) {
33.     //外层循环，负责判断线程池状态
34.     retry:
35.     for (;;) {
36.         int c = ctl.get();
37.         int rs = runStateOf(c); //状态
38.
39.         // Check if queue empty only if necessary.
40.         /**
41.          * 线程池的 state 越小越是运行状态，runnbale=-1,
     shutdown=0,stop=1,tidying=2，terminated=3
42.          * 1、如果线程池 state 已经至少是 shutdown 状态了
43.          * 2、并且以下 3 个条件任意一个是 false
```

```
44.          *    rs == SHUTDOWN          （隐含：rs>=SHUTDOWN）false 情况： 线程池状态已经超过 shutdown，可能是 stop、tidying、terminated 其中一个，即线程池已经终止
45.          *    firstTask == null       （隐含：rs==SHUTDOWN）false 情况： firstTask 不为空，rs==SHUTDOWN 且 firstTask 不为空，return false，场景是在线程池已经 shutdown 后，还要添加新的任务，拒绝
46.          *    ! workQueue.isEmpty()   （隐含：rs==SHUTDOWN，firstTask==null）false 情况： workQueue 为空，当 firstTask 为空时是为了创建一个没有任务的线程，再从 workQueue 中获取任务，如果 workQueue 已经为空，那么就没有添加新 worker 线程的必要了
47.          * return false，即无法 addWorker()
48.          */
49.         if (rs >= SHUTDOWN &&
50.             ! (rs == SHUTDOWN &&
51.                firstTask == null &&
52.                ! workQueue.isEmpty()))
53.             return false;
54.
55.         //内层循环，负责 worker 数量+1
56.         for (;;) {
57.             int wc = workerCountOf(c); //worker 数量
58.
59.             //如果 worker 数量>线程池最大上限 CAPACITY（即使用 int 低 29 位可以容纳的最大值）
60.             //或者( worker 数量>corePoolSize 或  worker 数量>maximumPoolSize )，即已经超过了给定的边界
61.             if (wc >= CAPACITY ||
62.                 wc >= (core ? corePoolSize : maximumPoolSize))
63.                 return false;
64.
65.             //调用 unsafe CAS 操作，使得 worker 数量+1，成功则跳出 retry 循环
66.             if (compareAndIncrementWorkerCount(c))
67.                 break retry;
68.
69.             //CAS worker 数量+1 失败，再次读取 ctl
70.             c = ctl.get();  // Re-read ctl
71.
72.             //如果状态不等于之前获取的 state，跳出内层循环，继续去外层循环判断
73.             if (runStateOf(c) != rs)
74.                 continue retry;
75.             // else CAS failed due to workerCount change; retry inner loop
76.             // else CAS 失败时因为 workerCount 改变了，继续内层循环尝试 CAS 对 worker 数量+1
77.         }
78.     }
79.
```

```
80.     /**
81.      * worker 数量+1 成功的后续操作
82.      * 添加到 workers Set 集合，并启动 worker 线程
83.      */
84.     boolean workerStarted = false;
85.     boolean workerAdded = false;
86.     Worker w = null;
87.     try {
88.         final ReentrantLock mainLock = this.mainLock;
89.         w = new Worker(firstTask); //1、设置 worker 这个 AQS 锁的同步状态 state=-
    1
90.                                    //2、将 firstTask 设置给 worker 的成员变量
    firstTask
91.                                    //3、使用 worker 自身这个 runnable，调用
    ThreadFactory 创建一个线程，并设置给 worker 的成员变量 thread
92.         final Thread t = w.thread;
93.         if (t != null) {
94.             mainLock.lock();
95.             try {
96.                 //-----------------------------------------这部分代码是上锁
    的
97.                 // Recheck while holding lock.
98.                 // Back out on ThreadFactory failure or if
99.                 // shut down before lock acquired.
100.                 // 当获取到锁后，再次检查
101.                 int c = ctl.get();
102.                 int rs = runStateOf(c);
103.
104.                 //如果线程池在运行 running<shutdown 或者  线程池已经 shutdown，
    且 firstTask==null（可能是 workQueue 中仍有未执行完成的任务，创建没有初始任务的
    worker 线程执行）
105.                 //worker 数量-1 的操作在 addWorkerFailed()
106.                 if (rs < SHUTDOWN ||
107.                     (rs == SHUTDOWN && firstTask == null)) {
108.                     if (t.isAlive()) // precheck that t is startable   线程
    已经启动，抛非法线程状态异常
109.                         throw new IllegalThreadStateException();
110.
111.                     workers.add(w);//workers 是一个 HashSet<Worker>
112.
113.                     //设置最大的池大小 largestPoolSize，workerAdded 设置为
    true
114.                     int s = workers.size();
115.                     if (s > largestPoolSize)
```

```
                        largestPoolSize = s;
                    workerAdded = true;
                }
              //-----------------------------------------
            }
            finally {
                mainLock.unlock();
            }

            //如果往 HashSet 中添加 worker 成功，启动线程
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        //如果启动线程失败
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

5.3. 内部类 Worker

```
/**
 * Class Worker mainly maintains interrupt control state for
 * threads running tasks, along with other minor bookkeeping.
 * This class opportunistically extends AbstractQueuedSynchronizer
 * to simplify acquiring and releasing a lock surrounding each
 * task execution.  This protects against interrupts that are
 * intended to wake up a worker thread waiting for a task from
 * instead interrupting a task being run.  We implement a simple
 * non-reentrant mutual exclusion lock rather than use
 * ReentrantLock because we do not want worker tasks to be able to
 * reacquire the lock when they invoke pool control methods like
 * setCorePoolSize.  Additionally, to suppress interrupts until
 * the thread actually starts running tasks, we initialize lock
 * state to a negative value, and clear it upon start (in
 * runWorker).
 *
 * Worker 类大体上管理着运行线程的中断状态 和 一些指标
 * Worker 类投机取巧的继承了 AbstractQueuedSynchronizer 来简化在执行任务时的获取、释放锁
```

```
19.  * 这样防止了中断在运行中的任务，只会唤醒(中断)在等待从 workQueue 中获取任务的线程
20.  * 解释：
21.  *    为什么不直接执行 execute(command)提交的 command，而要在外面包一层 Worker 呢？？
22.  *    主要是为了控制中断
23.  *    用什么控制？？
24.  *    用 AQS 锁，当运行时上锁，就不能中断，TreadPoolExecutor 的 shutdown()方法中断前都要获取 worker 锁
25.  *    只有在等待从 workQueue 中获取任务 getTask()时才能中断
26.  * worker 实现了一个简单的不可重入的互斥锁，而不是用 ReentrantLock 可重入锁
27.  * 因为我们不想让在调用比如 setCorePoolSize()这种线程池控制方法时可以再次获取锁(重入)
28.  * 解释：
29.  *    setCorePoolSize()时可能会 interruptIdleWorkers()，在对一个线程 interrupt 时会要 w.tryLock()
30.  *    如果可重入，就可能会在对线程池操作的方法中中断线程，类似方法还有：
31.  *    setMaximumPoolSize()
32.  *    setKeppAliveTime()
33.  *    allowCoreThreadTimeOut()
34.  *    shutdown()
35.  * 此外，为了让线程真正开始后才可以中断，初始化 lock 状态为负值(-1)，在开始 runWorker()时将 state 置为 0，而 state>=0 才可以中断
36.  *
37.  *
38.  * Worker 继承了 AQS，实现了 Runnable，说明其既是一个可运行的任务，也是一把锁（不可重入）
39.  */
40. private final class Worker
41.     extends AbstractQueuedSynchronizer
42.     implements Runnable
43. {
44.     /**
45.      * This class will never be serialized, but we provide a
46.      * serialVersionUID to suppress a javac warning.
47.      */
48.     private static final long serialVersionUID = 6138294804551838833L;
49.
50.     /** Thread this worker is running in.  Null if factory fails. */
51.     final Thread thread; //利用 ThreadFactory 和 Worker 这个 Runnable 创建的线程对象
52.
53.     /** Initial task to run.  Possibly null. */
54.     Runnable firstTask;
55.
```

```
56.     /** Per-thread task counter */
57.     volatile long completedTasks;
58.
59.     /**
60.      * Creates with given first task and thread from ThreadFactory.
61.      * @param firstTask the first task (null if none)
62.      */
63.     Worker(Runnable firstTask) {
64.         //设置 AQS 的同步状态 private volatile int state，是一个计数器，大于 0 代表锁已经被获取
65.         setState(-1); // inhibit interrupts until runWorker
66.                       // 在调用 runWorker()前，禁止 interrupt 中断，在interruptIfStarted()方法中会判断 getState()>=0
67.         this.firstTask = firstTask;
68.         this.thread = getThreadFactory().newThread(this); //根据当前 worker 创建一个线程对象
69.                                                           //当前 worker 本身就是一个 runnable 任务，也就是不会用参数的 firstTask 创建线程，而是调用当前 worker.run()时调用 firstTask.run()
70.     }
71.
72.     /** Delegates main run loop to outer runWorker  */
73.     public void run() {
74.         runWorker(this); //runWorker()是 ThreadPoolExecutor 的方法
75.     }
76.
77.     // Lock methods
78.     //
79.     // The value 0 represents the unlocked state. 0 代表“没被锁定”状态
80.     // The value 1 represents the locked state. 1 代表“锁定”状态
81.
82.     protected boolean isHeldExclusively() {
83.         return getState() != 0;
84.     }
85.
86.     /**
87.      * 尝试获取锁
88.      * 重写 AQS 的 tryAcquire()，AQS 本来就是让子类来实现的
89.      */
90.     protected boolean tryAcquire(int unused) {
91.         //尝试一次将 state 从 0 设置为 1，即“锁定”状态，但由于每次都是 state 0->1，而不是+1，那么说明不可重入
92.         //且 state==-1 时也不会获取到锁
93.         if (compareAndSetState(0, 1)) {
```

```
94.             setExclusiveOwnerThread(Thread.currentThread()); //设置
    exclusiveOwnerThread=当前线程
95.             return true;
96.         }
97.         return false;
98.     }
99.
100.     /**
101.      * 尝试释放锁
102.      * 不是 state-1，而是置为 0
103.      */
104.     protected boolean tryRelease(int unused) {
105.         setExclusiveOwnerThread(null);
106.         setState(0);
107.         return true;
108.     }
109.
110.     public void lock()        { acquire(1); }
111.     public boolean tryLock()  { return tryAcquire(1); }
112.     public void unlock()      { release(1); }
113.     public boolean isLocked() { return isHeldExclusively(); }
114.
115.     /**
116.      * 中断（如果运行）
117.      * shutdownNow 时会循环对 worker 线程执行
118.      * 且不需要获取 worker 锁，即使在 worker 运行时也可以中断
119.      */
120.     void interruptIfStarted() {
121.         Thread t;
122.         //如果 state>=0、t!=null、且 t 没有被中断
123.         //new Worker()时 state==-1，说明不能中断
124.         if (getState() >= 0 && (t = thread) != null && !t.isInterrupted())
    {
125.             try {
126.                 t.interrupt();
127.             } catch (SecurityException ignore) {
128.             }
129.         }
130.     }
131. }
```

Worker 类

Worker 类本身既实现了 Runnable，又继承了 AbstractQueuedSynchronizer（以下简称 AQS），所以其既是一个可执行的任务，又可以达到锁的效果

new Worker()

1、将 AQS 的 state 置为-1，在 runWoker()前不允许中断

2、待执行的任务会以参数传入，并赋予 firstTask

3、用 Worker 这个 Runnable 创建 Thread

之所以 Worker 自己实现 Runnable，并创建 Thread，在 firstTask 外包一层，是因为要通过 Worker 控制中断，而 firstTask 这个工作任务只是负责执行业务

Worker 控制中断主要有以下几方面：

1、初始 AQS 状态为-1，此时不允许中断 interrupt()，只有在 worker 线程启动了，执行了 runWoker()，将 state 置为 0，才能中断

不允许中断体现在：

A、shutdown()线程池时，会对每个 worker tryLock()上锁，而 Worker 类这个 AQS 的 tryAcquire()方法是固定将 state 从 0->1，故初始状态 state==-1 时 tryLock()失败，没发 interrupt()

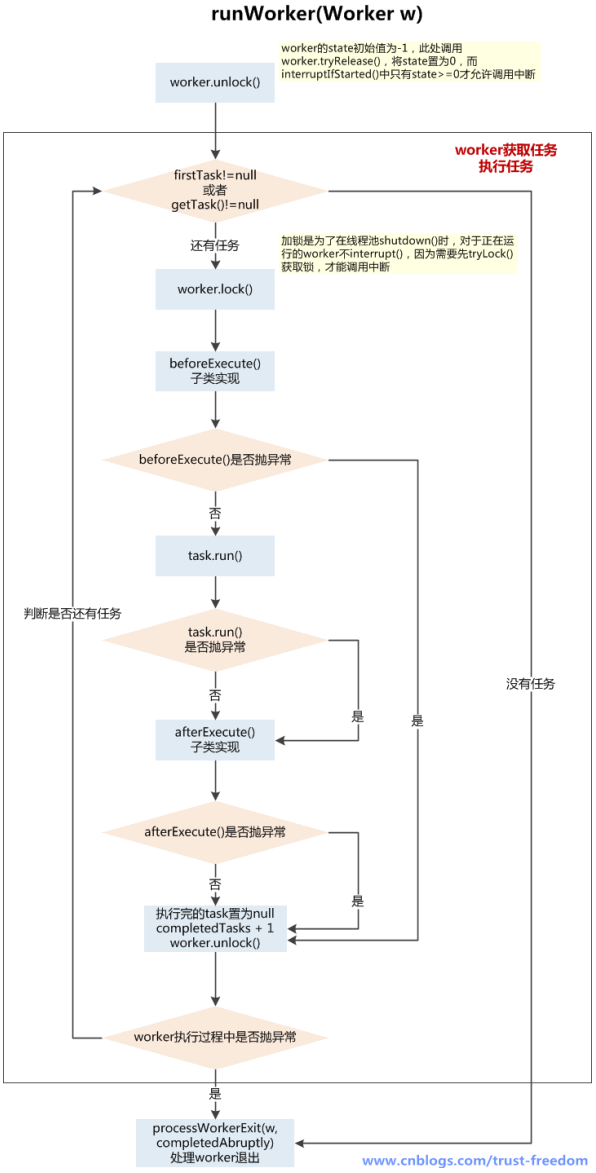
B、shutdownNow()线程池时，不用 tryLock()上锁，但调用 worker.interruptIfStarted()终止 worker，interruptIfStarted()也有 state>0 才能 interrupt 的逻辑

2、为了防止某种情况下，在运行中的 worker 被中断，runWorker()每次运行任务时都会 lock()上锁，而 shutdown()这类可能会终止 worker 的操作需要先获取 worker 的锁，这样就防止了中断正在运行的线程，Worker 实现的 AQS 为不可重入锁，为了是在获得 worker 锁的情况下再进入其它一些需要加锁的方法

Worker 和 Task 的区别：

Worker 是线程池中的线程，而 Task 虽然是 runnable，但是并没有真正执行，只是被 Worker 调用了 run 方法，后面会看到这部分的实现。

5.4. runWorker() -- 执行任务




```
1.  /**
2.   * Main worker run loop.  Repeatedly gets tasks from queue and
3.   * executes them, while coping with a number of issues:
4.   * 重复的从队列中获取任务并执行，同时应对一些问题：
5.   *
6.   * 1. We may start out with an initial task, in which case we
7.   * don't need to get the first one. Otherwise, as long as pool is
8.   * running, we get tasks from getTask. If it returns null then the
9.   * worker exits due to changed pool state or configuration
10.  * parameters.  Other exits result from exception throws in
11.  * external code, in which case completedAbruptly holds, which
12.  * usually leads processWorkerExit to replace this thread.
```

```
13.  * 我们可能使用一个初始化任务开始，即 firstTask 为 null
14.  * 然后只要线程池在运行，我们就从 getTask()获取任务
15.  * 如果 getTask()返回 null，则 worker 由于改变了线程池状态或参数配置而退出
16.  * 其它退出因为外部代码抛异常了，这会使得 completedAbruptly 为 true，这会导致在
     processWorkerExit()方法中替换当前线程
17.  *
18.  * 2. Before running any task, the lock is acquired to prevent
19.  * other pool interrupts while the task is executing, and
20.  * clearInterruptsForTaskRun called to ensure that unless pool is
21.  * stopping, this thread does not have its interrupt set.
22.  * 在任何任务执行之前，都需要对 worker 加锁去防止在任务运行时，其它的线程池中断操作
23.  * clearInterruptsForTaskRun 保证除非线程池正在 stoping，线程不会被设置中断标示
24.  *
25.  * 3. Each task run is preceded by a call to beforeExecute, which
26.  * might throw an exception, in which case we cause thread to die
27.  * (breaking loop with completedAbruptly true) without processing
28.  * the task.
29.  * 每个任务执行前会调用 beforeExecute()，其中可能抛出一个异常，这种情况下会导致线
     程 die（跳出循环，且 completedAbruptly==true），没有执行任务
30.  * 因为 beforeExecute()的异常没有 cache 住，会上抛，跳出循环
31.  *
32.  * 4. Assuming beforeExecute completes normally, we run the task,
33.  * gathering any of its thrown exceptions to send to
34.  * afterExecute. We separately handle RuntimeException, Error
35.  * (both of which the specs guarantee that we trap) and arbitrary
36.  * Throwables.  Because we cannot rethrow Throwables within
37.  * Runnable.run, we wrap them within Errors on the way out (to the
38.  * thread's UncaughtExceptionHandler).  Any thrown exception also
39.  * conservatively causes thread to die.
40.  * 假定 beforeExecute()正常完成，我们执行任务
41.  * 汇总任何抛出的异常并发送给 afterExecute(task, thrown)
42.  * 因为我们不能在 Runnable.run()方法中重新上抛 Throwables，我们将 Throwables 包装
     到 Errors 上抛（会到线程的 UncaughtExceptionHandler 去处理）
43.  * 任何上抛的异常都会导致线程 die
44.  *
45.  * 5. After task.run completes, we call afterExecute, which may
46.  * also throw an exception, which will also cause thread to
47.  * die. According to JLS Sec 14.20, this exception is the one that
48.  * will be in effect even if task.run throws.
49.  * 任务执行结束后，调用 afterExecute()，也可能抛异常，也会导致线程 die
50.  * 根据 JLS Sec 14.20，这个异常（finally 中的异常）会生效
51.  *
52.  * The net effect of the exception mechanics is that afterExecute
53.  * and the thread's UncaughtExceptionHandler have as accurate
```

```
 * information as we can provide about any problems encountered by
 * user code.
 *
 * @param w the worker
 */
final void runWorker(Worker w) {
    Thread wt = Thread.currentThread();
    Runnable task = w.firstTask;
    w.firstTask = null;
    w.unlock(); // allow interrupts
                // new Worker()是 state==-1，此处是调用 Worker 类的 tryRelease()方法，将 state 置为 0， 而 interruptIfStarted()中只有 state>=0 才允许调用中断
    boolean completedAbruptly = true; //是否“突然完成”，如果是由于异常导致的进入 finally，那么 completedAbruptly==true 就是突然完成的
    try {
        /**
         * 如果 task 不为 null，或者从阻塞队列中 getTask()不为 null
         */
        while (task != null || (task = getTask()) != null) {
            w.lock(); //上锁，不是为了防止并发执行任务，为了在 shutdown()时不终止正在运行的 worker

            // If pool is stopping, ensure thread is interrupted;
            // if not, ensure thread is not interrupted.  This
            // requires a recheck in second case to deal with
            // shutdownNow race while clearing interrupt
            /**
             * clearInterruptsForTaskRun 操作
             * 确保只有在线程 stoping 时，才会被设置中断标示，否则清除中断标示
             * 1、如果线程池状态>=stop，且当前线程没有设置中断状态，wt.interrupt()
             * 2、如果一开始判断线程池状态<stop，但 Thread.interrupted()为 true，即线程已经被中断，又清除了中断标示，再次判断线程池状态是否>=stop
             *    是，再次设置中断标示，wt.interrupt()
             *    否，不做操作，清除中断标示后进行后续步骤
             */
            if ((runStateAtLeast(ctl.get(), STOP) ||
                 (Thread.interrupted() &&
                  runStateAtLeast(ctl.get(), STOP))) &&
                !wt.isInterrupted())
                wt.interrupt(); //当前线程调用 interrupt()中断

            try {
                //执行前（子类实现）
```

```
93.                 beforeExecute(wt, task);
94.
95.                 Throwable thrown = null;
96.                 try {
97.                     task.run();
98.                 }
99.                 catch (RuntimeException x) {
100.                     thrown = x; throw x;
101.                 }
102.                 catch (Error x) {
103.                     thrown = x; throw x;
104.                 }
105.                 catch (Throwable x) {
106.                     thrown = x; throw new Error(x);
107.                 }
108.                 finally {
109.                     //执行后（子类实现）
110.                     afterExecute(task, thrown); //这里就考验 catch 和 finally
    的执行顺序了，因为要以 thrown 为参数
111.                 }
112.             }
113.             finally {
114.                 task = null; //task 置为 null
115.                 w.completedTasks++; //完成任务数+1
116.                 w.unlock(); //解锁
117.             }
118.         }
119.
120.         completedAbruptly = false;
121.     }
122.     finally {
123.         //处理 worker 的退出
124.         processWorkerExit(w, completedAbruptly);
125.     }
126. }
```

执行流程：

1、Worker 线程启动后，通过 Worker 类的 run()方法调用 runWorker(this)

2、执行任务之前，首先 worker.unlock()，将 AQS 的 state 置为 0，允许中断当前 worker 线程

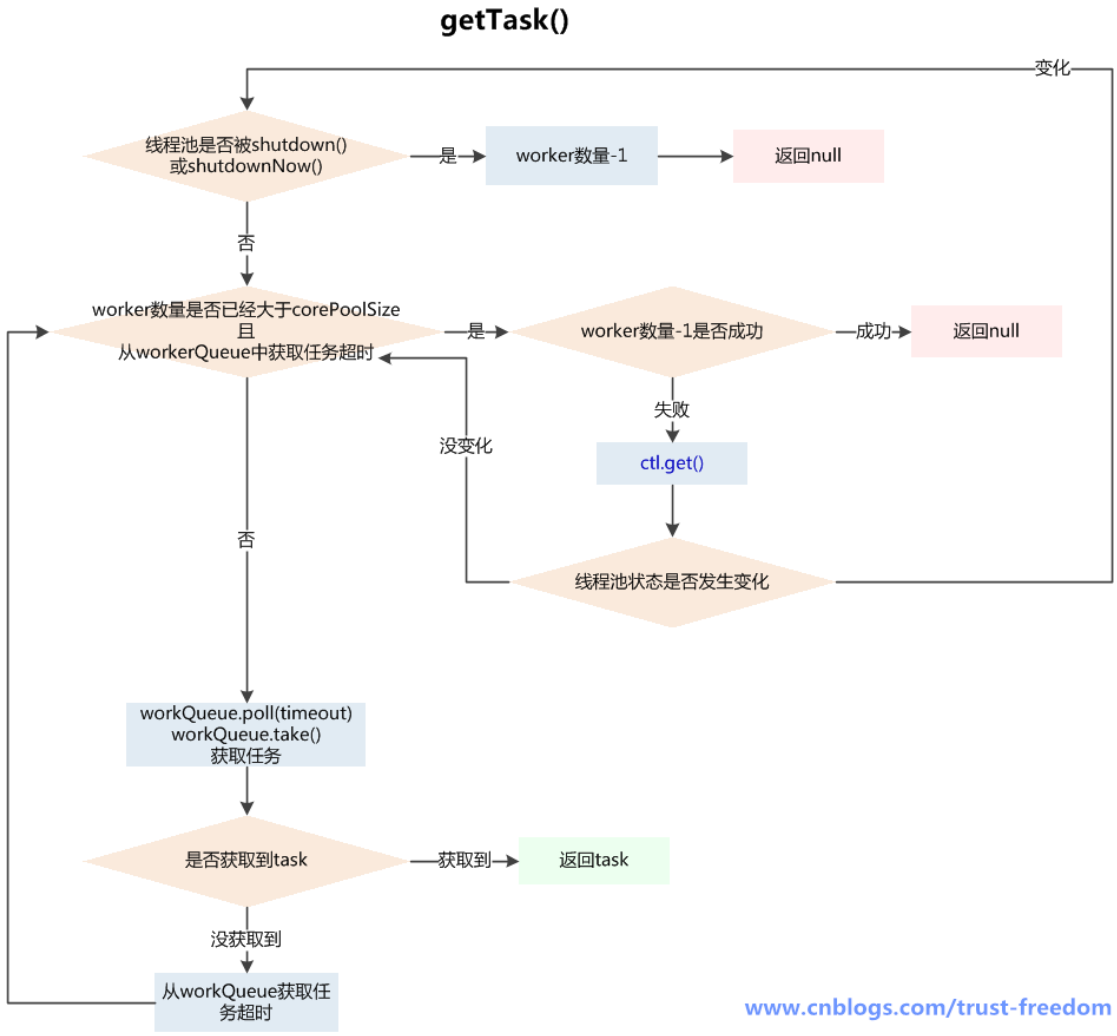
3、开始执行 firstTask，调用 task.run()，在执行任务前会上锁 wroker.lock()，在执行完任务后会解锁，为了防止在任务运行时被线程池一些中断操作中断

4、在任务执行前后，可以根据业务场景自定义 beforeExecute() 和 afterExecute()方法

5、无论在 beforeExecute()、task.run()、afterExecute()发生异常上抛，都会导致 worker 线程终止，进入 processWorkerExit()处理 worker 退出的流程

6、如正常执行完当前 task 后，会通过 getTask()从阻塞队列中获取新任务，当队列中没有任务，且获取任务超时，那么当前 worker 也会进入退出流程

5.5. getTask() -- 获取任务




```
1.  /**
2.   * Performs blocking or timed wait for a task, depending on
3.   * current configuration settings, or returns null if this worker
4.   * must exit because of any of:  以下情况会返回 null
5.   * 1. There are more than maximumPoolSize workers (due to
6.   *    a call to setMaximumPoolSize).
7.   *    超过了 maximumPoolSize 设置的线程数量（因为调用了 setMaximumPoolSize()）
8.   * 2. The pool is stopped.
9.   *    线程池被 stop
10.  * 3. The pool is shutdown and the queue is empty.
11.  *    线程池被 shutdown，并且 workQueue 空了
12.  * 4. This worker timed out waiting for a task, and timed-out
```

```
13.  *    workers are subject to termination (that is,
14.  *    {@code allowCoreThreadTimeOut || workerCount > corePoolSize})
15.  *    both before and after the timed wait.
16.  *    线程等待任务超时
17.  *
18.  * @return task, or null if the worker must exit, in which case
19.  *         workerCount is decremented
20.  *         返回 null 表示这个 worker 要结束了，这种情况下 workerCount-1
21.  */
22. private Runnable getTask() {
23.     boolean timedOut = false; // Did the last poll() time out?
24.
25.     /**
26.      * 外层循环
27.      * 用于判断线程池状态
28.      */
29.     retry:
30.     for (;;) {
31.         int c = ctl.get();
32.         int rs = runStateOf(c);
33.
34.         // Check if queue empty only if necessary.
35.         /**
36.          * 对线程池状态的判断，两种情况会 workerCount-1，并且返回 null
37.          * 线程池状态为 shutdown，且 workQueue 为空（反映了 shutdown 状态的线程池还是要执行 workQueue 中剩余的任务的）
38.          * 线程池状态为 stop（shutdownNow()会导致变成 STOP）（此时不用考虑 workQueue 的情况）
39.          */
40.         if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
41.             decrementWorkerCount(); //循环的 CAS 减少 worker 数量，直到成功
42.             return null;
43.         }
44.
45.         boolean timed;      // Are workers subject to culling?
46.                             // 是否需要定时从 workQueue 中获取
47.
48.         /**
49.          * 内层循环
50.          * 要么 break 去 workQueue 获取任务
51.          * 要么超时了，worker count-1
52.          */
53.         for (;;) {
54.             int wc = workerCountOf(c);
```

```
55.             timed = allowCoreThreadTimeOut || wc > corePoolSize; //allowCore
    ThreadTimeOut 默认为 false
56.                                                                 //如果
    allowCoreThreadTimeOut 为 true，说明 corePoolSize 和 maximum 都需要定时
57.
58.             //如果当前执行线程数<maximumPoolSize，并且 timedOut 和 timed 任一为
    false，跳出循环，开始从 workQueue 获取任务
59.             if (wc <= maximumPoolSize && ! (timedOut && timed))
60.                 break;
61.
62.             /**
63.              * 如果到了这一步，说明要么线程数量超过了 maximumPoolSize（可能
    maximumPoolSize 被修改了）
64.              * 要么既需要计时 timed==true，也超时了 timedOut==true
65.              * worker 数量-1，减一执行一次就行了，然后返回 null，在 runWorker()中
    会有逻辑减少 worker 线程
66.              * 如果本次减一失败，继续内层循环再次尝试减一
67.              */
68.             if (compareAndDecrementWorkerCount(c))
69.                 return null;
70.
71.             //如果减数量失败，再次读取 ctl
72.             c = ctl.get();  // Re-read ctl
73.
74.             //如果线程池运行状态发生变化，继续外层循环
75.             //如果状态没变，继续内层循环
76.             if (runStateOf(c) != rs)
77.                 continue retry;
78.             // else CAS failed due to workerCount change; retry inner loop
79.         }
80.
81.         try {
82.             //poll() - 使用  LockSupport.parkNanos(this, nanosTimeout) 挂起一
    段时间，interrupt()时不会抛异常，但会有中断响应
83.             //take() - 使用 LockSupport.park(this) 挂起，interrupt()时不会抛异
    常，但会有中断响应
84.             Runnable r = timed ?
85.                 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :    //
    大于 corePoolSize
86.                 workQueue.take();                                        //
    小于等于 corePoolSize
87.
88.             //如获取到了任务就返回
89.             if (r != null)
```

```
90.                 return r;
91.
92.             //没有返回，说明超时，那么在下一次内层循环时会进入 worker count 减一的步骤
93.             timedOut = true;
94.         }
95.         /**
96.              * blockingQueue 的 take()阻塞使用 LockSupport.park(this)进入 wait 状态的，对 LockSupport.park(this)进行 interrupt 不会抛异常，但还是会有中断响应
97.              * 但 AQS 的 ConditionObject 的 await()对中断状态做了判断，会报告中断状态 reportInterruptAfterWait(interruptMode)
98.              * 就会上抛 InterruptedException，在此处捕获，重新开始循环
99.              * 如果是由于 shutdown()等操作导致的空闲 worker 中断响应，在外层循环判断状态时，可能 return null
100.              */
101.         catch (InterruptedException retry) {
102.             timedOut = false; //响应中断，重新开始，中断状态会被清除
103.         }
104.     }
105. }
```

执行流程：

1、首先判断是否可以满足从 workQueue 中获取任务的条件，不满足 return null

A、线程池状态是否满足：

（a）shutdown 状态 + workQueue 为空 或 stop 状态，都不满足，因为被 shutdown 后还是要执行 workQueue 剩余的任务，但 workQueue 也为空，就可以退出了

（b）stop 状态，shutdownNow()操作会使线程池进入 stop，此时不接受新任务，中断正在执行的任务，workQueue 中的任务也不执行了，故 return null 返回

B、线程数量是否超过 maximumPoolSize 或 获取任务是否超时

（a）线程数量超过 maximumPoolSize 可能是线程池在运行时被调用了 setMaximumPoolSize()被改变了大小，否则已经 addWorker()成功不会超过 maximumPoolSize

（b）如果 当前线程数量>corePoolSize，才会检查是否获取任务超时，这也体现了当线程数量达到 maximumPoolSize 后，如果一直没有新任务，会逐渐终止 worker 线程直到 corePoolSize

2、如果满足获取任务条件，根据是否需要定时获取调用不同方法：

A、workQueue.poll()：如果在 keepAliveTime 时间内，阻塞队列还是没有任务，返回 null

B、workQueue.take()：如果阻塞队列为空，当前线程会被挂起等待；当队列中有任务加入时，线程被唤醒，take 方法返回任务

3、在阻塞从 workQueue 中获取任务时，可以被 interrupt()中断，代码中捕获了 InterruptedException，重置 timedOut 为初始值 false，再次执行第 1 步中的判断，满足就继续获取任务，不满足 return null，会进入 worker 退出的流程

5.6. processWorkerExit() -- worker 线程退出

```
1.  /**
2.   * Performs cleanup and bookkeeping for a dying worker. Called
3.   * only from worker threads. Unless completedAbruptly is set,
4.   * assumes that workerCount has already been adjusted to account
5.   * for exit.  This method removes thread from worker set, and
6.   * possibly terminates the pool or replaces the worker if either
7.   * it exited due to user task exception or if fewer than
8.   * corePoolSize workers are running or queue is non-empty but
9.   * there are no workers.
10.  *
11.  * @param w the worker
12.  * @param completedAbruptly if the worker died due to user exception
13.  */
14. private void processWorkerExit(Worker w, boolean completedAbruptly) {
15.     /**
16.      * 1、worker 数量-1
17.      * 如果是突然终止，说明是 task 执行时异常情况导致，即 run()方法执行时发生了异常，那么正在工作的 worker 线程数量需要-1
18.      * 如果不是突然终止，说明是 worker 线程没有 task 可执行了，不用-1，因为已经在 getTask()方法中-1 了
19.      */
20.     if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted 代码和注释正好相反啊
21.         decrementWorkerCount();
22.
23.     /**
24.      * 2、从 Workers Set 中移除 worker
25.      */
26.     final ReentrantLock mainLock = this.mainLock;
27.     mainLock.lock();
28.     try {
29.         completedTaskCount += w.completedTasks; //把 worker 的完成任务数加到线程池的完成任务数
30.         workers.remove(w); //从 HashSet<Worker>中移除
31.     } finally {
```

```
32.         mainLock.unlock();
33.     }
34.
35.     /**
36.      * 3、在对线程池有负效益的操作时，都需要“尝试终止”线程池
37.      * 主要是判断线程池是否满足终止的状态
38.      * 如果状态满足，但还有线程池还有线程，尝试对其发出中断响应，使其能进入退出流程
39.      * 没有线程了，更新状态为 tidying->terminated
40.      */
41.     tryTerminate();
42.
43.     /**
44.      * 4、是否需要增加 worker 线程
45.      * 线程池状态是 running 或 shutdown
46.      * 如果当前线程是突然终止的，addWorker()
47.      * 如果当前线程不是突然终止的，但当前线程数量 < 要维护的线程数量，addWorker()
48.      * 故如果调用线程池 shutdown()，直到 workQueue 为空前，线程池都会维持 corePoolSize 个线程，然后再逐渐销毁这 corePoolSize 个线程
49.      */
50.     int c = ctl.get();
51.     //如果状态是 running、shutdown，即 tryTerminate()没有成功终止线程池，尝试再添加一个 worker
52.     if (runStateLessThan(c, STOP)) {
53.         //不是突然完成的，即没有 task 任务可以获取而完成的，计算 min，并根据当前 worker 数量判断是否需要 addWorker()
54.         if (!completedAbruptly) {
55.             int min = allowCoreThreadTimeOut ? 0 : corePoolSize; //allowCoreThreadTimeOut 默认为 false，即 min 默认为 corePoolSize
56.
57.             //如果 min 为 0，即不需要维持核心线程数量，且 workQueue 不为空，至少保持一个线程
58.             if (min == 0 && ! workQueue.isEmpty())
59.                 min = 1;
60.
61.             //如果线程数量大于最少数量，直接返回，否则下面至少要 addWorker 一个
62.             if (workerCountOf(c) >= min)
63.                 return; // replacement not needed
64.         }
65.
66.         //添加一个没有 firstTask 的 worker
67.         //只要 worker 是 completedAbruptly 突然终止的，或者线程数量小于要维护的数量，就新添一个 worker 线程，即使是 shutdown 状态
68.         addWorker(null, false);
```

```
69.     }
70. }
```

processWorkerExit(Worker w, boolean completedAbruptly)

参数：

worker： 要结束的 worker

completedAbruptly： 是否突然完成（是否因为异常退出）

执行流程：

1、worker 数量-1

A、如果是突然终止，说明是 task 执行时异常情况导致，即 run()方法执行时发生了异常，那么正在工作的 worker 线程数量需要-1

B、如果不是突然终止，说明是 worker 线程没有 task 可执行了，不用-1，因为已经在 getTask()方法中-1 了

2、从 Workers Set 中移除 worker，删除时需要上锁 mainlock

3、tryTerminate()：在对线程池有负效益的操作时，都需要“尝试终止”线程池，大概逻辑：

判断线程池是否满足终止的状态

A、如果状态满足，但还有线程池还有线程，尝试对其发出中断响应，使其能进入退出流程

B、没有线程了，更新状态为 tidying->terminated

4、是否需要增加 worker 线程，如果线程池还没有完全终止，仍需要保持一定数量的线程

线程池状态是 running 或 shutdown

A、如果当前线程是突然终止的，addWorker()

B、如果当前线程不是突然终止的，但当前线程数量 < 要维护的线程数量，addWorker()

故如果调用线程池 shutdown()，直到 workQueue 为空前，线程池都会维持 corePoolSize 个线程，然后再逐渐销毁这 corePoolSize 个线程

6. shutdown() -- 温柔的终止线程池

```
1.  /**
2.   * Initiates an orderly shutdown in which previously submitted
3.   * tasks are executed, but no new tasks will be accepted.
4.   * Invocation has no additional effect if already shut down.
5.   * 开始一个有序的关闭，在关闭中，之前提交的任务会被执行（包含正在执行的，在阻塞队列中的），但新任务会被拒绝
6.   * 如果线程池已经 shutdown，调用此方法不会有附加效应
```

```
7.   *
8.   * <p>This method does not wait for previously submitted tasks to
9.   * complete execution.  Use {@link #awaitTermination awaitTermination}
10.  * to do that.
11.  * 当前方法不会等待之前提交的任务执行结束，可以使用 awaitTermination()
12.  *
13.  * @throws SecurityException {@inheritDoc}
14.  */
15. public void shutdown() {
16.     final ReentrantLock mainLock = this.mainLock;
17.     mainLock.lock(); //上锁
18.
19.     try {
20.         //判断调用者是否有权限 shutdown 线程池
21.         checkShutdownAccess();
22.
23.         //CAS+循环设置线程池状态为 shutdown
24.         advanceRunState(SHUTDOWN);
25.
26.         //中断所有空闲线程
27.         interruptIdleWorkers();
28.
29.         onShutdown(); // hook for ScheduledThreadPoolExecutor
30.     }
31.     finally {
32.         mainLock.unlock(); //解锁
33.     }
34.
35.     //尝试终止线程池
36.     tryTerminate();
37. }
```

shutdown()执行流程：

1、上锁，mainLock 是线程池的主锁，是可重入锁，当要操作 workers set 这个保持线程的 HashSet 时，需要先获取 mainLock，还有当要处理 largestPoolSize、completedTaskCount 这类统计数据时需要先获取 mainLock

2、判断调用者是否有权限 shutdown 线程池

3、使用 CAS 操作将线程池状态设置为 shutdown，shutdown 之后将不再接收新任务

4、中断所有空闲线程　interruptIdleWorkers()

5、onShutdown()，ScheduledThreadPoolExecutor 中实现了这个方法，可以在 shutdown()时做一些处理

6、解锁

7、尝试终止线程池　tryTerminate()

6.1. interruptIdleWorkers()　--　中断空闲 worker

```
1.  /**
2.   * Interrupts threads that might be waiting for tasks (as
3.   * indicated by not being locked) so they can check for
4.   * termination or configuration changes. Ignores
5.   * SecurityExceptions (in which case some threads may remain
6.   * uninterrupted).
7.   * 中断在等待任务的线程(没有上锁的)，中断唤醒后，可以判断线程池状态是否变化来决定是否继续
8.   *
9.   * @param onlyOne If true, interrupt at most one worker. This is
10.  * called only from tryTerminate when termination is otherwise
11.  * enabled but there are still other workers.  In this case, at
12.  * most one waiting worker is interrupted to propagate shutdown
13.  * signals in case(以免) all threads are currently waiting.
14.  * Interrupting any arbitrary thread ensures that newly arriving
15.  * workers since shutdown began will also eventually exit.
16.  * To guarantee eventual termination, it suffices to always
17.  * interrupt only one idle worker, but shutdown() interrupts all
18.  * idle workers so that redundant workers exit promptly, not
19.  * waiting for a straggler task to finish.
20.  *
21.  * onlyOne 如果为 true，最多 interrupt 一个 worker
22.  * 只有当终止流程已经开始，但线程池还有 worker 线程时,tryTerminate()方法会做调用 onlyOne 为 true 的调用
23.  * （终止流程已经开始指的是：shutdown 状态 且 workQueue 为空，或者 stop 状态）
24.  * 在这种情况下，最多有一个 worker 被中断，为了传播 shutdown 信号，以免所有的线程都在等待
25.  * 为保证线程池最终能终止，这个操作总是中断一个空闲 worker
26.  * 而 shutdown()中断所有空闲 worker，来保证空闲线程及时退出
27.  */
28. private void interruptIdleWorkers(boolean onlyOne) {
29.     final ReentrantLock mainLock = this.mainLock;
30.     mainLock.lock(); //上锁
31.     try {
32.         for (Worker w : workers) {
33.             Thread t = w.thread;
34.
35.             if (!t.isInterrupted() && w.tryLock()) {
36.                 try {
37.                     t.interrupt();
```

```
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
                break;
        }
    } finally {
        mainLock.unlock(); //解锁
    }
}
```

InterruptIdleWorkers() 首先会获取 mainLock 锁，因为要迭代 workers set，在中断每个 worker 前，需要做两个判断：

1、线程是否已经被中断，是就什么都不做

2、worker.tryLock() 是否成功

第二个判断比较重要，因为 Worker 类除了实现了可执行的 Runnable，也继承了 AQS，本身也是一把锁。tryLock()调用了 Worker 自身实现的 tryAcquire()方法，这也是 AQS 规定子类需要实现的尝试获取锁的方法。

```
protected boolean tryAcquire(int unused) {
    if (compareAndSetState(0, 1)) {
        setExclusiveOwnerThread(Thread.currentThread());
        return true;
    }
    return false;
}
```

tryAcquire()先尝试将 AQS 的 state 从 0-->1，返回 true 代表上锁成功，并设置当前线程为锁的拥有者。可以看到 compareAndSetState(0, 1)只尝试了一次获取锁，且不是每次 state+1，而是 0-->1，说明锁不是可重入的

6.2. tryTerminate() -- 尝试终止线程池

```
/**
 * Transitions to TERMINATED state if either (SHUTDOWN and pool
 * and queue empty) or (STOP and pool empty).  If otherwise
 * eligible to terminate but workerCount is nonzero, interrupts an
 * idle worker to ensure that shutdown signals propagate. This
 * method must be called following any action that might make
 * termination possible -- reducing worker count or removing tasks
```

```
8.   * from the queue during shutdown. The method is non-private to
9.   * allow access from ScheduledThreadPoolExecutor.
10.  *
11.  * 在以下情况将线程池变为 TERMINATED 终止状态
12.  * shutdown 且 正在运行的 worker 和 workQueue 队列 都 empty
13.  * stop 且  没有正在运行的 worker
14.  *
15.  * 这个方法必须在任何可能导致线程池终止的情况下被调用，如：
16.  * 减少 worker 数量
17.  * shutdown 时从 queue 中移除任务
18.  *
19.  * 这个方法不是私有的，所以允许子类 ScheduledThreadPoolExecutor 调用
20.  */
21. final void tryTerminate() {
22.     //这个 for 循环主要是和进入关闭线程池操作的 CAS 判断结合使用的
23.     for (;;) {
24.         int c = ctl.get();
25.
26.         /**
27.          * 线程池是否需要终止
28.          * 如果以下 3 中情况任一为 true，return，不进行终止
29.          * 1、还在运行状态
30.          * 2、状态是 TIDYING、或 TERMINATED，已经终止过了
31.          * 3、SHUTDOWN 且 workQueue 不为空
32.          */
33.         if (isRunning(c) ||
34.             runStateAtLeast(c, TIDYING) ||
35.             (runStateOf(c) == SHUTDOWN && ! workQueue.isEmpty()))
36.             return;
37.
38.         /**
39.          * 只有 shutdown 状态 且 workQueue 为空，或者 stop 状态能执行到这一步
40.          * 如果此时线程池还有线程（正在运行任务，正在等待任务）
41.          * 中断唤醒一个正在等任务的空闲 worker
42.          * 唤醒后再次判断线程池状态，会 return null，进入 processWorkerExit()流
    程
43.          */
44.         if (workerCountOf(c) != 0) { // Eligible to terminate 资格终止
45.             interruptIdleWorkers(ONLY_ONE); //中断 workers 集合中的空闲任务，参
    数为 true，只中断一个
46.             return;
47.         }
48.
49.         /**
```

```
50.          * 如果状态是 SHUTDOWN，workQueue 也为空了，正在运行的 worker 也没有了，开始 terminated
51.          */
52.         final ReentrantLock mainLock = this.mainLock;
53.         mainLock.lock();
54.         try {
55.             //CAS：将线程池的 ctl 变成 TIDYING（所有的任务被终止，workCount 为 0，为此状态时将会调用 terminated()方法），期间 ctl 有变化就会失败，会再次 for 循环
56.             if (ctl.compareAndSet(c, ctlOf(TIDYING, 0))) {
57.                 try {
58.                     terminated(); //需子类实现
59.                 }
60.                 finally {
61.                     ctl.set(ctlOf(TERMINATED, 0)); //将线程池的 ctl 变成 TERMINATED
62.                     termination.signalAll(); //唤醒调用了 等待线程池终止的线程 awaitTermination()
63.                 }
64.                 return;
65.             }
66.         }
67.         finally {
68.             mainLock.unlock();
69.         }
70.         // else retry on failed CAS
71.         // 如果上面的 CAS 判断 false，再次循环
72.     }
73. }
```

tryTerminate() 执行流程：

1、判断线程池是否需要进入终止流程（只有当 shutdown 状态+workQueue.isEmpty 或 stop 状态，才需要）

2、判断线程池中是否还有线程，有则 interruptIdleWorkers(ONLY_ONE) 尝试中断一个空闲线程（正是这个逻辑可以再次发出中断信号，中断阻塞在获取任务的线程）

3、如果状态是 SHUTDOWN，workQueue 也为空了，正在运行的 worker 也没有了，开始 terminated

会先上锁，将线程池置为 tidying 状态，之后调用需子类实现的 terminated()，最后线程池置为 terminated 状态，并唤醒所有等待线程池终止这个 Condition 的线程

7. shutdownNow() -- 强硬的终止线程池

```
1. /**
```

```
2.   * Attempts to stop all actively executing tasks, halts the
3.   * processing of waiting tasks, and returns a list of the tasks
4.   * that were awaiting execution. These tasks are drained (removed)
5.   * from the task queue upon return from this method.
6.   * 尝试停止所有活动的正在执行的任务，停止等待任务的处理，并返回正在等待被执行的任务列表
7.   * 这个任务列表是从任务队列中排出（删除）的
8.   *
9.   * <p>This method does not wait for actively executing tasks to
10.  * terminate.  Use {@link #awaitTermination awaitTermination} to
11.  * do that.
12.  * 这个方法不用等到正在执行的任务结束，要等待线程池终止可使用 awaitTermination()
13.  *
14.  * <p>There are no guarantees beyond best-effort attempts to stop
15.  * processing actively executing tasks.  This implementation
16.  * cancels tasks via {@link Thread#interrupt}, so any task that
17.  * fails to respond to interrupts may never terminate.
18.  * 除了尽力尝试停止运行中的任务，没有任何保证
19.  * 取消任务是通过 Thread.interrupt()实现的，所以任何响应中断失败的任务可能永远不会结束
20.  *
21.  * @throws SecurityException {@inheritDoc}
22.  */
23. public List<Runnable> shutdownNow() {
24.     List<Runnable> tasks;
25.     final ReentrantLock mainLock = this.mainLock;
26.     mainLock.lock(); //上锁
27.
28.     try {
29.         //判断调用者是否有权限 shutdown 线程池
30.         checkShutdownAccess();
31.
32.         //CAS+循环设置线程池状态为 stop
33.         advanceRunState(STOP);
34.
35.         //中断所有线程，包括正在运行任务的
36.         interruptWorkers();
37.
38.         tasks = drainQueue(); //将 workQueue 中的元素放入一个 List 并返回
39.     }
40.     finally {
41.         mainLock.unlock(); //解锁
42.     }
43.
```

```
44.     //尝试终止线程池
45.     tryTerminate();
46.
47.     return tasks; //返回 workQueue 中未执行的任务
48. }
```

shutdownNow() 和 shutdown()的大体流程相似，差别是：

1、将线程池更新为 stop 状态

2、调用 interruptWorkers() 中断所有线程，包括正在运行的线程

3、将 workQueue 中待处理的任务移到一个 List 中，并在方法最后返回，说明 shutdownNow()后不会再处理 workQueue 中的任务

7.1. interruptWorkers() -- 中断所有 worker

```
1.  private void interruptWorkers() {
2.      final ReentrantLock mainLock = this.mainLock;
3.      mainLock.lock();
4.      try {
5.          for (Worker w : workers)
6.              w.interruptIfStarted();
7.      } finally {
8.          mainLock.unlock();
9.      }
10. }
```

interruptWorkers() 很简单，循环对所有 worker 调用 interruptIfStarted()，其中会判断 worker 的 AQS state 是否大于 0，即 worker 是否已经开始运作，再调用 Thread.interrupt()

需要注意的是，对于运行中的线程调用 Thread.interrupt()并不能保证线程被终止，task.run()内部可能捕获了 InterruptException，没有上抛，导致线程一直无法结束

8. awaitTermination() -- 等待线程池终止

```
public boolean awaitTermination(long timeout, TimeUnit unit)
    throws InterruptedException {
    long nanos = unit.toNanos(timeout);
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (;;) {
            if (runStateAtLeast(ctl.get(), TERMINATED))
                return true;

            if (nanos <= 0)
                return false;

            nanos = termination.awaitNanos(nanos);
        }
    } finally {
        mainLock.unlock();
    }
}
```

awaitTermination() 循环的判断线程池是否terminated终止 或 是否已经超过超时时间，然后通过 termination 这个 Condition 阻塞等待一段时间

termination.awaitNanos() 是通过 LockSupport.parkNanos(this, nanosTimeout)实现的阻塞等待

阻塞等待过程中发生以下具体情况会解除阻塞（对上面 3 种情况的解释）：

1、如果发生了 termination.signalAll()（内部实现是 LockSupport.unpark()）会唤醒阻塞等待，且由于 ThreadPoolExecutor 只有在 tryTerminated()尝试终止线程池成功，将线程池更新为 terminated 状态后才会 signalAll()，故 awaitTermination()再次判断状态会 return true 退出

2、如果达到了超时时间 termination.awaitNanos() 也会返回，此时 nano==0，再次循环判断 return false，等待线程池终止失败

3、如果当前线程被 Thread.interrupt()，termination.awaitNanos() 会上抛 InterruptException，awaitTermination()继续上抛给调用线程，会以异常的形式解除阻塞

故终止线程池并需要知道其是否终止可以用如下方式：

```
executorService.shutdown();
try{
    while(!executorService.awaitTermination(500, TimeUnit.MILLISECONDS)) {
        LOGGER.debug("Waiting for terminate");
    }
}
catch (InterruptedException e) {
    //中断处理
```

```
9. }
```