主讲老师：Fox

> 有道笔记地址：https://note.youdao.com/s/3cQn63uH

1. 线程池简介

线程池（Thread Pool）是一种基于池化思想管理线程的工具，经常出现在多线程服务器中，如Tomcat。

线程过多会带来额外的开销，其中包括创建销毁线程的开销、调度线程的开销等等，同时也降低了计算机的整体性能。线程池维护多个线程，等待监督管理者分配可并发执行的任务。这种做法，一方面避免了处理任务时创建销毁线程开销的代价，另一方面避免了线程数量膨胀导致的过分调度问题，保证了对内核的充分利用。

线程池有如下的优势：

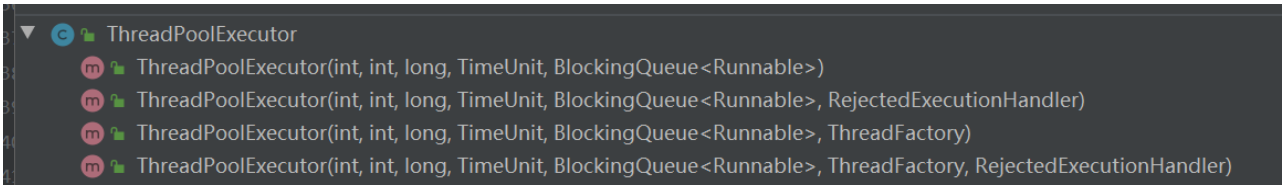
- **降低资源消耗**：通过池化技术重复利用已创建的线程，降低线程创建和销毁造成的损耗。
- **提高响应速度**：任务到达时，无需等待线程创建即可立即执行。
- **提高线程的可管理性**：线程是稀缺资源，如果无限制创建，不仅会消耗系统资源，还会因为线程的不合理分布导致资源调度失衡，降低系统的稳定性。使用线程池可以进行统一的分配、调优和监控。
- **提供更多更强大的功能**：线程池具备可拓展性，允许开发人员向其中增加更多的功能。比如延时定时线程池ScheduledThreadPoolExecutor，就允许任务延期执行或定期执行。

2. 线程池的使用

创建线程池

ThreadPoolExecutor——推荐使用

利用ThreadPoolExecutor提供的构造方法创建线程池



主要看参数最全的构造方法，其他构造方法最终还是会调用该改造方法

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
```

线程池的核心参数

corePoolSize：核心线程数，线程池初始化时默认是没有线程的，当任务来临时才开始创建线程去执行任务

maximumPoolSize：最大线程数，在核心线程数已满，且队列已满时，如果池子里的工作线程数小于maximumPoolSize，则会创建非核心线程执行任务

keepAliveTime：非核心线程数的空闲时间超过keepAliveTime就会被自动终止回收掉，但在corePoolSize=maximumPoolSize时，该值无效，因为不存在非核心线程

unit：keepAliveTime的时间单位

workQueue：用于保存线程任务的队列，主要分为无界、有界、同步移交等队列，当池子里的工作线程数大于corePoolSize，就会将新进来的线程任务放入队列中

- ArrayBlockingQueue(有界队列)：队列长度有限，当队列满了就需要创建非核心线程执行任务，如果最大线程数已满，则执行拒绝策略
- LinkedBlockingQueue(无界队列)：队列长度无限，当任务处理速度跟不上任务创建速度，可能会导致内存占用过多或OOM
- SynchronousQueue(同步队列)：队列不作为任务的缓冲处理，队列长度为0

threadFactory：

- 创建线程的工厂接口，默认使用Executors.defaultThreadFactory()
- 另外可以实现ThreadFactory接口，自定义线程工厂

handler：线程池无法继续接收任务时(workQueue已满和maximumPoolSize已满)的拒绝策略

- AbortPolicy：默认拒绝策略，中断抛出RejectedExecutionException异常
- CallerRunsPolicy：让提交任务的主线程来执行任务
- DiscardOldestPolicy：丢弃在队列中存在时间最久的任务，重复执行
- DiscardPolicy：丢弃任务，不进行任何通知
- 另外可以实现RejectedExecutionHandler接口，自定义拒绝策略

	名称	描述
1	ThreadPoolExecutor.AbortPolicy	丢弃任务并抛出RejectedExecutionException异常。 这是线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐使用此拒绝策略，这样子在系统不能承载更大的并发量的时候，能够及时的通过异常发现。
2	ThreadPoolExecutor.DiscardPolicy	丢弃任务，但是不抛出异常。 使用此策略，可能会使我们无法发现系统的异常状态。建议是一些无关紧要的业务采用此策略。
3	ThreadPoolExecutor.DiscardOldestPolicy	丢弃队列最前面的任务，然后重新提交被拒绝的任务。是否要采用此种拒绝策略，还得根据实际业务是否允许丢弃老任务来认真衡量。
4	ThreadPoolExecutor.CallerRunsPolicy	由调用线程（提交任务的线程）处理该任务。这种情况是需要让所有任务都执行完毕，那么就适合大量计算的任务类型去执行，多线程仅仅是增大吞吐量的手段，最终必须要让每个任务都执行完毕。

以下是一个使用ThreadPoolExecutor创建线程池的示例：

```java
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor.AbortPolicy;

public class ThreadPoolExample2 {

    public static void main(String[] args) {
        // 线程池的核心线程数
        int corePoolSize = 5;
        // 线程池的最大线程数
        int maximumPoolSize = 10;
        // 线程池的任务队列
        ArrayBlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(100);
        // 线程池保持空闲的时间
        long keepAliveTime = 60L;
        // 时间单位
        TimeUnit unit = TimeUnit.SECONDS;
        // 线程池的拒绝策略
        RejectedExecutionHandler handler = new ThreadPoolExecutor.AbortPolicy();

        // 创建线程池
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                corePoolSize,
                maximumPoolSize,
                keepAliveTime,
                unit,
                workQueue,
                handler
        );

        // 提交任务到线程池
        for (int i = 0; i < 15; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println("Task " + taskId + " is running by " +
Thread.currentThread().getName());
            });
```

```
        }

        // 关闭线程池
        executor.shutdown();
        try {
            // 等待所有任务完成，超时时间为60秒
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                // 如果超时后任务仍未完成，则强制关闭线程池
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            // 如果等待过程中被中断，也强制关闭线程池
            executor.shutdownNow();
        }

        System.out.println("All tasks are done or interrupted.");
    }
}
```

在这个示例中，我们直接通过ThreadPoolExecutor的构造器来创建线程池，并设置了核心线程数、最大线程数、任务队列、空闲线程存活时间以及拒绝策略。然后，我们向线程池提交任务，并在所有任务提交完毕后调用shutdown()方法来关闭线程池。我们还使用awaitTermination()方法来等待所有任务完成，如果超时或等待过程中被中断，则调用shutdownNow()方法强制关闭线程池。

Executors——不推荐使用

Executors提供了各种线程池类型创建的静态方法，如常见的newFixedThreadPool、newSingleThreadExecutor、newCachedThreadPool、newSingleThreadScheduledExecutor。

类型	池内 线程类型	池内 线程数量	处理特点	应用场景
定长线程池 (FixedThreadPool)	核心线程	固定	• 核心线程处于空闲状态时也不会被回收，除非线程被关闭 • 当所有线程都处于活动状态时，新的任务都会处于等待状态，直到有线程空闲出来 • 任务队列无大小限制 （超出的线程任务会在队列中等待）	控制线程最大并发数
定时线程池 (ScheduledThreadPool)	核心 & 非核心线程	• 核心线程 固定 • 非核心线程 无限制	当非核心线程闲置时，则会被立即回收	执行定时 / 周期性 任务
可缓存线程池 (CachedThreadPool)	非核心线程	不固定 （可无限大）	• 优先利用闲置线程处理新任务 （即 会重用线程） • 无线程可用时，即 新建线程 （即 任何线程任务到来都会立刻执行，不需要等待） • 灵活回收空闲线程 （具备超时机制=60s，即空闲超过60s才回收，全部回收时几乎不占系统资源）	执行数量多、耗时少的线程任务
单线程化线程池 (SingleThreadExecutor)	核心线程	1个	• 保证所有任务按照指定顺序在一个线程中执行 （相当于顺序执行任务） • 不需要处理线程同步的问题	单线程 （不适合并发但可能引起IO阻塞性及影响UI线程响应的操作，如数据库操作等）

以下是一个使用Java中的ExecutorService和Executors创建线程池的简单示例：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolExample {

    public static void main(String[] args) {
        // 创建一个固定大小的线程池
        ExecutorService executor = Executors.newFixedThreadPool(5);

        // 提交任务到线程池
        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " is running by " + Thread.currentThread().getName());
            });
        }

        // 关闭线程池
        executor.shutdown();
        while (!executor.isTerminated()) {
            // 等待所有任务完成
        }

        System.out.println("All tasks are done.");
    }
}
```

但不提倡使用该种方式创建线程池，在阿里巴巴JAVA开发手册，对于线程池创建要求：

4．【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

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.MAX_VALUE，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.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

提交任务

线程池提交任务有两种方法：

- 无返回值的任务使用 public void execute(Runnable command) 方法提交；
- 有返回值的任务使用：
  - Future<?> submit(Runnable task)： 提交Runnable任务
  - Future<T> submit(Runnable task, T result)： 提交Runnable任务并指定执行结果
  - Future<T> submit(Callable<T> task)： 提交Callable任务

如何执行批量任务

```
# 执行批量任务，返回它们的执行结果
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws InterruptedException
#执行批量任务，返回指定时间内完成的执行结果，取消未完成的任务
public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit)
# 执行批量任务，返回最先完成的执行结果
public <T> T invokeAny(Collection<? extends Callable<T>> tasks) throws InterruptedException, ExecutionException
#执行批量任务，返回指定时间内最先完成的执行结果，取消未完成的任务
public <T> T invokeAny(Collection<? extends Callable<T>> tasks,long timeout, TimeUnit unit)
```

示例

```java
public class InvokeAllDemo {

    public static void main(String[] args) {
        List<Task> tasks = new ArrayList<>();
        for (int i = 1; i <= 10; i++) {
            tasks.add(new Task(i));
        }

        //创建线程池
        ExecutorService threadpool = Executors.newFixedThreadPool(5);
        try {
            //批量提交任务
            List<Future<Integer>> futures = threadpool.invokeAll(tasks);
            for (Future<Integer> future : futures){
                System.out.println(future.get());
            }

        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }finally {
            //关闭线程池
            threadpool.shutdown();
        }


    }


    static class Task implements Callable<Integer> {

        private int index;

        Task(int index) {
            this.index = index;
        }

        @Override
```

```
        public Integer call() throws Exception {
            Thread.sleep(1000);
            return index;
        }
    }

}
```

如何执行定时、延时任务

```
#具备执行定时、延时、周期性任务的线程池
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor implements ScheduledExecutorService {
#延时执行Runnable任务，只执行一次
public ScheduledFuture<?> schedule(Runnable command,long delay,TimeUnit unit)
#延时执行Callable任务，只执行一次
public <V> ScheduledFuture<V> schedule(Callable<V> callable,long delay,TimeUnit unit)
```

如何执行周期、重复性任务

```
#廷时一段时间后，周期性执行Runnable任务，周期为固定时间
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit)
#廷时一段时间后，周期性执行Runnable任务，周期为间隔时间

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit)
```

关闭线程池

- shutdownNow()：立即关闭线程池，正在执行中的任务和队列中的任务都会被中断，同时返回被中断的队列中的任务列表。

- shutdown()：关闭线程池，正在执行中的任务和队列中的任务都能执行完成，后续进来的新任务会被执行拒绝策略。
- isTerminated()：当正在执行的任务和队列中的任务全部都执行完时返回true。

shutdown和shutdownNow的区别

	shutdown	**shutdownNow**
立即关闭线程池	否	是
延时关闭线程池	是	否
不再接收新任务	是	是
继续执行完任务队列中的任务	是	否
返回任务队列中的任务	否	是
线程池状态	SHUTDOWN	STOP

线程池的参数设计分析

核心线程数(corePoolSize)

核心线程数的设计需要依据任务的处理时间和每秒产生的任务数量来确定。
例如：一个线程执行一个任务需要0.1秒，1秒就执行10个任务；系统80%的时间每秒都会产生100个任务，那么要想在1秒内处理完这100个任务，就需要10个线程。
此时我们就可以设计核心线程数为10;
当然实际情况不可能这么平均，所以我们一般按照二八原则设计即可，即按照80%的情况设计核心线程数，剩下的20%可以利用最大线程数处理;

任务队列长度(workQueue)

任务队列长度，也就是设计 阻塞队列 能缓存多少个任务。
任务队列长度一般设计为:
核心线程数 / 单个任务执行时间 *2 即可;
例如上面的场景中,核心线程数设计为10，单个任务执行时间为0.1秒，则队列长度可以设计为200 ;

最大线程数(maximumPoolSize)

最大线程数的设计除了需要参照核心线程数的条件外，还需要参照系统每秒产生的最大任务数决定;
例如：上述环境中，如果系统每秒最大产生的任务是1000个，那么：

最大线程数 = (最大任务数 - 任务队列长度) * 单个任务执行时间;
即： 最大线程数 = (1000 - 200) * 0.1 = 80个;

最大空闲时间(keepAliveTime)

这个参数的设计完全参考系统运行环境和硬件压力设定，没有固定的参考值，用户可以根据经验和系统产生任务的时间间隔合理设置一个值即可。

3. 线程池原理分析

线程池执行任务的具体流程是怎样的?

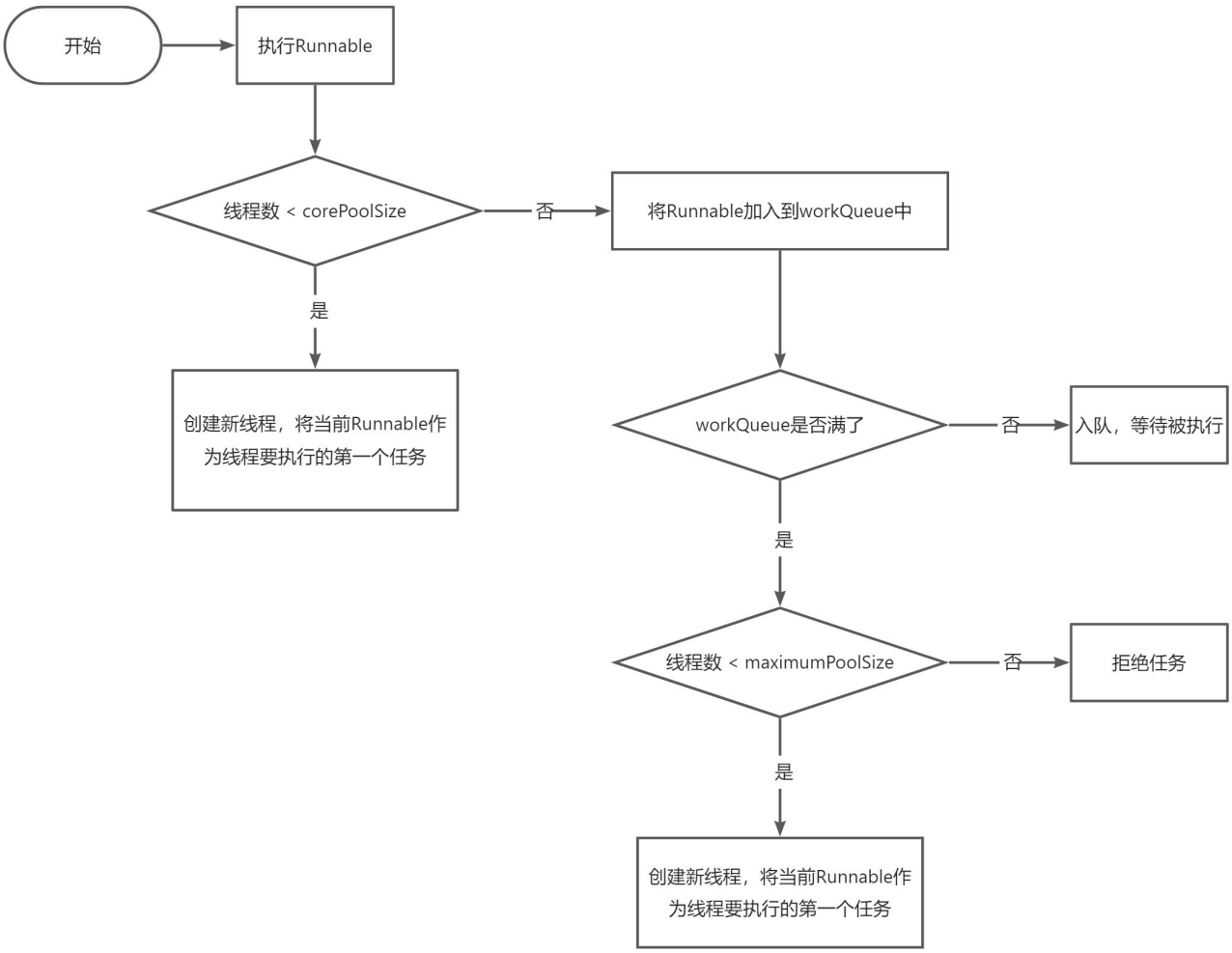
ThreadPoolExecutor中提供了两种执行任务的方法:

1. void execute(Runnable command)
2. Future<?> submit(Runnable task)

实际上submit中最终还是调用的execute()方法，只不过会返回一个Future对象，用来获取任务执行结果:

```
public Future<?> submit(Runnable task) {
    if (task == null) throw new NullPointerException();
 RunnableFuture<Void> ftask = newTaskFor(task, null);
 execute(ftask);
 return ftask;
}
```

execute(Runnable command)方法执行时会分为三步:



注意：提交一个Runnable时，不管当前线程池中的线程是否空闲，只要数量小于核心线程数就会创建新线程。

注意：ThreadPoolExecutor相当于是非公平的，比如队列满了之后提交的Runnable可能会比正在排队的Runnable先执行。

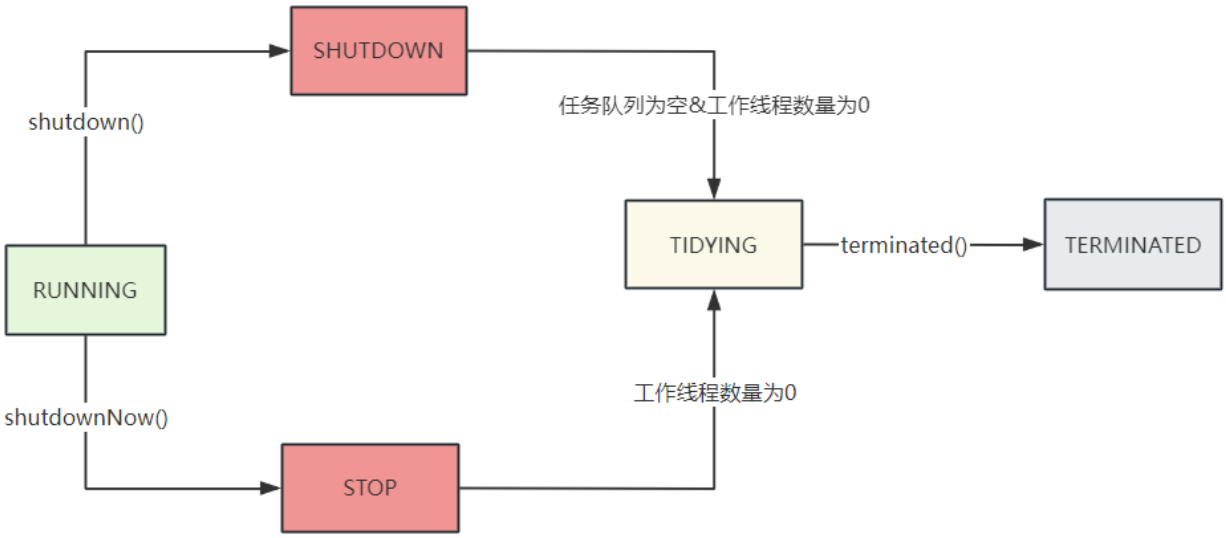
线程池的五种状态是如何流转的?

线程池有五种状态：

- RUNNING：**会**接收新任务并且**会**处理队列中的任务
- SHUTDOWN：**不会**接收新任务并且**会**处理队列中的任务
- STOP：**不会**接收新任务并且**不会**处理队列中的任务，并且会中断在处理的任务（注意：一个任务能不能被中断得看任务本身）
- TIDYING：所有任务都终止了，线程池中也没有线程了，这样线程池的状态就会转为TIDYING，一旦达到此状态，就会调用线程池的terminated()
- TERMINATED：terminated()执行完之后就会转变为TERMINATED

这五种状态并不能任意转换，只会有以下几种转换情况：

1. RUNNING -> SHUTDOWN：手动调用shutdown()触发，或者线程池对象GC时会调用finalize()从而调用shutdown()
2. (RUNNING or SHUTDOWN) -> STOP：调用shutdownNow()触发，如果先调shutdown()紧着调shutdownNow()，就会发生SHUTDOWN -> STOP
3. SHUTDOWN -> TIDYING：队列为空并且线程池中没有线程时自动转换
4. STOP -> TIDYING：线程池中没有线程时自动转换（队列中可能还有任务）
5. TIDYING -> TERMINATED：terminated()执行完后就会自动转换



线程池中的线程是如何关闭的?

我们一般会使用thread.start()方法来开启一个线程，那如何停掉一个线程呢?
线程池中就是利用线程中断机制来停止线程的。比如shutdownNow()方法中会调用：

```
void interruptIfStarted() {
    Thread t;
    if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
        try {
            t.interrupt();
        } catch (SecurityException ignore) {
        }
    }
}
```

线程池为什么一定得是阻塞队列?

线程池中的线程在运行过程中，执行完创建线程时绑定的第一个任务后，就会不断的从队列中获取任务并执行，那么如果队列中没有任务了，线程为了不自然消亡，就会阻塞在获取队列任务时，等着队列中有任务过来就会拿到任务从而去执行任务。

通过这种方法能最终确保，线程池中能保留指定个数的核心线程数，关键代码为：

```
try {
    Runnable r = timed ?
        workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
        workQueue.take();
    if (r != null)
        return r;
    timedOut = true;
} catch (InterruptedException retry) {
    timedOut = false;
}
```

某个线程在从队列获取任务时，会判断是否使用超时阻塞获取，我们可以认为非核心线程会poll()，核心线程会take()，非核心线程超过时间还没获取到任务后面就会自然消亡了。

线程发生异常，会被移出线程池吗？

答案是会的，那有没有可能核心线程数在执行任务时都出错了，导致所有核心线程都被移出了线程池？

```
                    wt.interrupt();
                try {
                    beforeExecute(wt, task);
                    try {
                        task.run();
                        afterExecute(task, t: null);
                    } catch (Throwable ex) {
                        afterExecute(task, ex);
                        throw ex;
                    }
                } finally {
                    task = null;
                    w.completedTasks++;
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            processWorkerExit(w, completedAbruptly);
        }
    }
```

在源码中，当执行任务时出现异常时，最终会执行processWorkerExit()，执行完这个方法后，当前线程也就自然消亡了，但是！processWorkerExit()方法中会额外再新增一个线程，这样就能维持住固定的核心线程数。

4. 线程池源码分析

线程池源码的基础属性和方法

在线程池的源码中，会通过一个AtomicInteger类型的变量**ctl**，来表示**线程池的状态**和**当前线程池中的工作线程数量**。

一个Integer占4个字节，也就是32个bit，线程池有5个状态：

1. RUNNING
2. SHUTDOWN
3. STOP
4. TIDYING
5. TERMINATED

2个bit能表示4种状态，那5种状态就至少需要三个bit位，比如在线程池的源码中就是这么来表示的：

```
private static final int COUNT_BITS = Integer.SIZE - 3;

private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;
```

Integer.SIZE为32，所以COUNT_BITS为29，最终各个状态对应的二级制为：

1. RUNNING：**111**00000 00000000 00000000 00000000
2. SHUTDOWN：**000**00000 00000000 00000000 00000000
3. STOP：**001**00000 00000000 00000000 00000000
4. TIDYING：**010**00000 00000000 00000000 00000000
5. TERMINATED：**011**00000 00000000 00000000 00000000

所以，只需要使用一个Integer数字的最高三个bit，就可以表示5种线程池的状态，而剩下的29个bit就可以用来表示工作线程数，比如，假如ctl为：**111**00000 00000000 00000000 0000**1**0**1**0，就表示线程池的状态为RUNNING，线程池中目前在工作的线程有10个，这里说的“在工作”意思是线程活着，要么在执行任务，要么在阻塞等待任务。

同时，在线程池中也提供了一些方法用来获取线程池状态和工作线程数，比如：

```
// 29，二进制为00000000 00000000 00000000 00011101
private static final int COUNT_BITS = Integer.SIZE - 3;

// 00011111 11111111 11111111 11111111
private static final int CAPACITY   = (1 << COUNT_BITS) - 1;

// ~CAPACITY为11100000 00000000 00000000 00000000
// &操作之后，得到就是c的高3位
private static int runStateOf(int c)     {
        return c & ~CAPACITY;
}

// CAPACITY为00011111 11111111 11111111 11111111
// &操作之后，得到的就是c的低29位
private static int workerCountOf(int c)  {
        return c & CAPACITY;
}
```

同时，还有一个方法：

```
private static int ctlOf(int rs, int wc) {
        return rs | wc;
}
```

就是用来把运行状态和工作线程数量进行合并的一个方法，不过传入这个方法的两个int数字有限制，rs的低29位都得为0，wc的高3位都得为0，这样经过或运算之后，才能得到准确的ctl。

同时，还有一些相关的方法：

```
private static final int RUNNING    = -1 << COUNT_BITS;
private static final int SHUTDOWN   =  0 << COUNT_BITS;
private static final int STOP       =  1 << COUNT_BITS;
private static final int TIDYING    =  2 << COUNT_BITS;
private static final int TERMINATED =  3 << COUNT_BITS;

// c状态是否小于s状态，比如RUNNING小于SHUTDOWN
private static boolean runStateLessThan(int c, int s) {
        return c < s;
}

// c状态是否大于等于s状态，比如STOP大于SHUTDOWN
private static boolean runStateAtLeast(int c, int s) {
        return c >= s;
}

// c状态是不是RUNNING，只有RUNNING是小于SHUTDOWN的
private static boolean isRunning(int c) {
        return c < SHUTDOWN;
}

// 通过cas来增加工作线程数量，直接对ctl进行加1
// 这个方法没考虑是否超过最大工作线程数的（2的29次方）限制，源码中在调用该方法之前会进行判断的
private boolean compareAndIncrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect + 1);
}

// 通过cas来减少工作线程数量，直接对ctl进行减1
private boolean compareAndDecrementWorkerCount(int expect) {
        return ctl.compareAndSet(expect, expect - 1);
}
```

前面说到线程池有5个状态，这5个状态分别表示：

1. RUNNING：线程池正常运行中，可以正常的接受并处理任务
2. SHUTDOWN：线程池关闭了，**不能接受新任务**，但是**线程池会把阻塞队列中的剩余任务执行完**，剩余任务都处理完之后，会中断所有工作线程
3. STOP：线程池停止了，**不能接受新任务**，并且也**不会处理阻塞队列中的任务**，会中断所有工作线程

4. TIDYING：当前线程池中的工作线程都被停止后，就会进入TIDYING
5. TERMINATED：线程池处于TIDYING状态后，会执行terminated()方法，执行完后就会进入TERMINATED状态，在ThreadPoolExecutor中terminated()是一个空方法，可以自定义线程池重写这个方法

execute方法

当执行线程池的execute方法时：

```
public void execute(Runnable command) {

        if (command == null)
                throw new NullPointerException();

        // 获取ctl
        // ctl初始值是ctlOf(RUNNING, 0)，表示线程池处于运行中，工作线程数为0
        int c = ctl.get();

        // 工作线程数小于corePoolSize，则添加工作线程，并把command作为该线程要执行的任务
        if (workerCountOf(c) < corePoolSize) {
                // true表示添加的是核心工作线程，具体一点就是，在addWorker内部会判断当前工作线程数是不是超过了corePoolSize
                // 如果超过了则会添加失败，addWorker返回false，表示不能直接开启新的线程来执行任务，而是应该先入队
                if (addWorker(command, true))
                        return;

                // 如果添加核心工作线程失败，那就重新获取ctl，可能是线程池状态被其他线程修改了
                // 也可能是其他线程也在向线程池提交任务，导致核心工作线程已经超过了corePoolSize
                c = ctl.get();
        }

        // 线程池状态是否还是RUNNING，如果是就把任务添加到阻塞队列中
        if (isRunning(c) && workQueue.offer(command)) {

                // 在任务入队时，线程池的状态可能也会发生改变
                // 再次检查线程池的状态，如果线程池不是RUNNING了，那就不能再接受任务了，就得把任务从队列中移除，并进行拒绝策略

                // 如果线程池的状态没有发生改变，仍然是RUNNING，那就不需要把任务从队列中移除掉
                // 不过，为了确保刚刚入队的任务有线程会去处理它，需要判断一下工作线程数，如果为0，那就添加一个非核心的工作线程
                // 添加的这个线程没有自己的任务，目的就是从队列中获取任务来执行
                int recheck = ctl.get();
                if (! isRunning(recheck) && remove(command))
                        reject(command);
                else if (workerCountOf(recheck) == 0)
```

```
35                          addWorker(null, false);
36          }
37          // 如果线程池状态不是RUNNING，或者线程池状态是RUNNING但是队列满了，则去添加一个非核心工作线程
38          // 实际上，addWorker中会判断线程池状态如果不是RUNNING，是不会添加工作线程的
39          // false表示非核心工作线程，作用是，在addWorker内部会判断当前工作线程数已经超过了maximumPoolSize，如果超过了则会添加不成功，执行拒绝策略
40          else if (!addWorker(command, false))
41                  reject(command);
42  }
```

addWorker方法

addWorker方法是核心方法，是用来添加线程的，core参数表示添加的是核心线程还是非核心线程。

在看这个方法之前，我们不妨先自己来分析一下，什么是添加线程？

实际上就要开启一个线程，不管是核心线程还是非核心线程其实都只是一个普通的线程，而核心和非核心的区别在于：

1. 如果是要添加**核心工作线程**，那么就得判断目前的工作线程数是否超过corePoolSize
    a. 如果没有超过，则直接开启新的工作线程执行任务
    b. 如果超过了，则不会开启新的工作线程，而是把任务进行入队
1. 如果要添加的是**非核心工作线程**，那就要判断目前的工作线程数是否超过maximumPoolSize
    a. 如果没有超过，则直接开启新的工作线程执行任务
    b. 如果超过了，则拒绝执行任务

所以在addWorker方法中，首先就要判断工作线程有没有超过限制，如果没有超过限制再去开启一个线程。

并且在addWorker方法中，还得判断线程池的状态，如果线程池的状态不是RUNNING状态了，那就没必要要去添加线程了，当然有一种特例，就是线程池的状态是SHUTDOWN，但是队列中有任务，那此时还是需要添加添加一个线程的。

那这种特例是如何产生的呢？

我们前面提到的都是开启新的工作线程，那么工作线程怎么回收呢？不可能开启的工作线程一直活着，因为如果任务由多变少，那也就不需要过多的线程资源，所以线程池中会有机制对开启的工作线程进行回收，如何回收的，后文会提到，我们这里先分析，有没有可能线程池中所有的线程都被回收了，答案的是有的。

首先非核心工作线程被回收是可以理解的，那核心工作线程要不要回收掉呢？其实线程池存在的意义，就是提前生成好线程资源，需要线程的时候直接使用就可以，而不需要临时去开启线程，所以正常情况下，开启的核心工作线程是不用回收掉的，就算暂时没有任务要处理，也不用回收，就让核心工作线程在那等着就可以了。

但是！在线程池中有这么一个参数：**allowCoreThreadTimeOut**，表示是否允许核心工作线程超时，意思就是**是否允许核心工作线程回收**，默认这个参数为false，但是我们可以调用allowCoreThreadTimeOut(boolean value)来把这个参数改为true，只要改了，那么核心工作线程也就会被回收了，那这样线程池中的所有工作线程都可能被回收掉，那如果所有工作线程都被回收掉之后，阻塞队列中来了一个任务，这样就形成了特例情况。

```
private boolean addWorker(Runnable firstTask, boolean core) {
        retry:
        for (;;) {
                int c = ctl.get();
                int rs = runStateOf(c);

                // 线程池如果是SHUTDOWN状态并且队列非空则创建线程，如果队列为空则不创建线程了
                // 线程池如果是STOP状态则直接不创建线程了
                // Check if queue empty only if necessary.
                if (rs >= SHUTDOWN &&
                        ! (rs == SHUTDOWN &&
                           firstTask == null &&
                           ! workQueue.isEmpty()))
                        return false;

                // 判断工作线程数是否超过了限制
                // 如果超过限制了，则return false
                // 如果没有超过限制，则修改ctl，增加工作线程数，cas成功则退出外层retry循环，去创建新的工作线程
                // 如果cas失败，则表示有其他线程也在提交任务，也在增加工作线程数，此时重新获取ctl
                // 如果发现线程池的状态发生了变化，则继续回到retry，重新判断线程池的状态是不是SHUTDOWN或STOP
                // 如果状态没有变化，则继续利用cas来增加工作线程数，直到cas成功
                for (;;) {
                        int wc = workerCountOf(c);
                        if (wc >= CAPACITY ||
                                wc >= (core ? corePoolSize : maximumPoolSize))
                                return false;
                        if (compareAndIncrementWorkerCount(c))
                                break retry;
                        c = ctl.get();  // Re-read ctl
                        if (runStateOf(c) != rs)
                                continue retry;
                        // else CAS failed due to workerCount change; retry inner loop
                }
        }

        // ctl修改成功，也就是工作线程数+1成功
```

```
37          // 接下来就要开启一个新的工作线程了
38
39          boolean workerStarted = false;
40          boolean workerAdded = false;
41          Worker w = null;
42          try {
43                  // Worker实现了Runnable接口
44                  // 在构造一个Worker对象时，就会利用ThreadFactory新建一个线程
45                  // Worker对象有两个属性：
46                  // Runnable firstTask：表示Worker待执行的第一个任务，第二个任务会从阻塞队列
中获取
47                  // Thread thread：表示Worker对应的线程，就是这个线程来获取队列中的任务并执行
的
48                  w = new Worker(firstTask);
49
50                  // 拿出线程对象，还没有start
51                  final Thread t = w.thread;
52                  if (t != null) {
53                          final ReentrantLock mainLock = this.mainLock;
54                          mainLock.lock();
55                          try {
56                                  // Recheck while holding lock.
57                                  // Back out on ThreadFactory failure or if
58                                  // shut down before lock acquired.
59                                  int rs = runStateOf(ctl.get());
60
61                                  // 如果线程池的状态是RUNNING
62                                  // 或者线程池的状态变成了SHUTDOWN，但是当前线程没有自己的第
一个任务，那就表示当前调用addWorker方法是为了从队列中获取任务来执行
63                                  // 正常情况下线程池的状态如果是SHUTDOWN，是不能创建新的工作
线程的，但是队列中如果有任务，那就是上面说的特例情况
64                                  if (rs < SHUTDOWN ||
65                                          (rs == SHUTDOWN && firstTask == null)) {
66
67                                          // 如果Worker对象对应的线程已经在运行了，那就有问
题，直接抛异常
68                                          if (t.isAlive()) // precheck that t is
startable
69                                                  throw new
IllegalThreadStateException();
70
```

```
                                        // workers用来记录当前线程池中工作线程，调用线程池的shutdown方法时会遍历worker对象中断对应线程
                                        workers.add(w);

                                        // largestPoolSize用来跟踪线程池在运行过程中工作线程数的峰值
                                        int s = workers.size();
                                        if (s > largestPoolSize)
                                                largestPoolSize = s;
                                        workerAdded = true;
                                }
                        } finally {
                                mainLock.unlock();
                        }

                        // 运行线程
                        if (workerAdded) {
                                t.start();
                                workerStarted = true;
                        }
                }
        } finally {
                // 在上述过程中如果抛了异常，需要从works中移除所添加的work，并且还要修改ctl，工作线程数-1，表示新建工作线程失败
                if (! workerStarted)
                        addWorkerFailed(w);
        }

        // 最后表示添加工作线程成功
        return workerStarted;
}
```

所以，对于addWorker方法，核心逻辑就是：

1. 先判断工作线程数是否超过了限制
2. 修改ctl，使得工作线程数+1
3. 构造Work对象，并把它添加到workers集合中
4. 启动Work对象对应的工作线程

runWorker方法

那工作线程在运行过程中，到底在做什么呢?

我们看看Work的构造方法:

```
Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
}
```

在利用ThreadFactory创建线程时，会把this，也就是当前Work对象作为Runnable传给线程，所以工作线程运行时，就会执行Worker的run方法:

```
public void run() {
        // 这个方法就是工作线程运行时的执行逻辑
        runWorker(this);
}
```

```
final void runWorker(Worker w) {
        // 就是当前工作线程
        Thread wt = Thread.currentThread();

        // 把Worker要执行的第一个任务拿出来
        Runnable task = w.firstTask;
        w.firstTask = null;

        // 这个地方，后面单独分析中断的时候来分析
        w.unlock(); // allow interrupts

        boolean completedAbruptly = true;
        try {

                // 判断当前线程是否有自己的第一个任务，如果没有就从阻塞队列中获取任务
                // 如果阻塞队列中也没有任务，那线程就会阻塞在这里
                // 但是并不会一直阻塞，在getTask方法中，会根据我们所设置的keepAliveTime来设置阻塞时间
                // 如果当前线程去阻塞队列中获取任务时，等了keepAliveTime时间，还没有获取到任务，则getTask方法返回null，相当于退出循环
                // 当然并不是所有线程都会有这个超时判断，主要还得看allowCoreThreadTimeOut属性和当前的工作线程数等等，后面单独分析
                // 目前，我们只需要知道工作线程在执行getTask()方法时，可能能直接拿到任务，也可能阻塞，也可能阻塞超时最终返回null
                while (task != null || (task = getTask()) != null) {
                        // 只要拿到了任务，就要去执行任务

                        // Work先加锁，跟shutdown方法有关，先忽略，后面会分析
                        w.lock();


                        // 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

                        // 下面这个if，最好把整篇文章都看完之后再来看这个if的逻辑

                        // 工作线程在运行过程中
```

```
                        // 如果发现线程池的状态变成了STOP，正常来说当前工作线程的中断标记应该为true，如果发现中断标记不为true，则需要中断自己

                        // 如果线程池的状态不是STOP，要么是RUNNING，要么是SHUTDOWN
                        // 但是如果发现中断标记为true，那是不对的，因为线程池状态不是STOP，工作线程仍然是要正常工作的，不能中断掉
                        // 就算是SHUTDOWN，也要等任务都执行完之后，线程才结束，而目前线程还在执行任务的过程中，不能中断
                        // 所以需要重置线程的中断标记，不过interrupted方法会自动清空中断标记
                        // 清空为中断标记后，再次判断一下线程池的状态，如果又变成了STOP，那就仍然中断自己

                        // 中断了自己后，会把当前任务执行完，在下一次循环调用getTask()方法时，从阻塞队列获取任务时，阻塞队列会负责判断当前线程的中断标记
                        // 如果发现中断标记为true，那就会抛出异常，最终退出while循环，线程执行结束
                        if ((runStateAtLeast(ctl.get(), STOP) ||
                                (Thread.interrupted() &&
                                 runStateAtLeast(ctl.get(), STOP))) &&
                                !wt.isInterrupted())
                                wt.interrupt();


                        try {
                                // 空方法，给自定义线程池来实现
                                beforeExecute(wt, task);
                                Throwable thrown = null;
                                try {
                                        // 执行任务
                                        // 注意执行任务时可能会抛异常，如果抛了异常会先依次执行三个finally，从而导致completedAbruptly = false这行代码没有执行
                                        task.run();
                                } catch (RuntimeException x) {
                                        thrown = x; throw x;
                                } catch (Error x) {
                                        thrown = x; throw x;
                                } catch (Throwable x) {
                                        thrown = x; throw new Error(x);
                                } finally {
                                        // 空方法，给自定义线程池来实现
                                        afterExecute(task, thrown);
                                }
```

```
                        } finally {
                                task = null;
                                w.completedTasks++; // 跟踪当前Work总共执行了多少了任务
                                w.unlock();
                        }
                }

                // 正常退出了While循环
                // 如果是执行任务的时候抛了异常，虽然也退出了循环，但是是不会执行这行代码的，只会直接进去下面的finally块中

                // 所以，要么是线程从队列中获取任务时阻塞超时了从而退出了循环会进入到这里
                // 要么是线程在阻塞的过程中被中断了，在getTask()方法中会处理中断的情况，如果被中断了，那么getTask()方法会返回null，从而退出循环
                // completedAbruptly=false，表示线程正常退出
                completedAbruptly = false;
        } finally {
                // 因为当前线程退出了循环，如果不做某些处理，那么这个线程就运行结束了，就是上文说的回收（自然消亡）掉了，线程自己运行完了也就结束了
                // 但是如果是由于执行任务的时候抛了异常，那么这个线程不应该直接结束，而应该继续从队列中获取下一个任务
                // 可是代码都执行到这里了，该怎么继续回到while循环呢，怎么实现这个效果呢？
                // 当然，如果是由于线程被中断了，或者线程阻塞超时了，那就应该正常的运行结束
                // 只不过有一些善后工作要处理，比如修改ctl，工作线程数-1
                processWorkerExit(w, completedAbruptly);
        }
}
```

processWorkerExit方法

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {

        // 如果completedAbruptly为true，表示是执行任务的时候抛了异常，那就修改ctl，工作线程数-1
        // 如果completedAbruptly为false，表示是线程阻塞超时了或者被中断了，实际上也要修改ctl，工作线程数-1
        // 只不过在getTask方法中已经做过了，这里就不用再做一次了
        if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
                decrementWorkerCount();

        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                // 当前Work要运行结束了，将完成的任务数累加到线程池上
                completedTaskCount += w.completedTasks;

                // 将当前Work对象从workers中移除
                workers.remove(w);
        } finally {
                mainLock.unlock();
        }

        // 因为当前是处理线程退出流程中，所以要尝试去修改线程池的状态为TINDYING
        tryTerminate();


        int c = ctl.get();
        // 如果线程池的状态为RUNNING或者SHUTDOWN，则可能要替补一个线程
        if (runStateLessThan(c, STOP)) {

                // completedAbruptly为false，表示线程是正常要退出了，则看是否需要保留线程
                if (!completedAbruptly) {

                        // 如果allowCoreThreadTimeOut为true，但是阻塞队列中还有任务，那就至少得保留一个工作线程来处理阻塞队列中的任务
                        // 如果allowCoreThreadTimeOut为false，那min就是corePoolSize，表示至少得保留corePoolSize个工作线程活着
                        int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
                        if (min == 0 && ! workQueue.isEmpty())
                                min = 1;
```

```
37
38                          // 如果当前工作线程数大于等于min，则表示符合所需要保留的最小线程数，那就直接return，不会调用下面的addWorker方法新开一个工作线程了
39                          if (workerCountOf(c) >= min)
40                                  return; // replacement not needed
41                  }
42
43                  // 如果线程池的状态为RUNNING或者SHUTDOWN
44                  // 如果completedAbruptly为true，表示当前线程是执行任务时抛了异常，那就得新开一个工作线程
45                  // 如果completedAbruptly为false，但是不符合所需要保留的最小线程数，那也得新开一个工作线程
46                  addWorker(null, false);
47          }
48  }
```

总结一下，某个工作线程正常情况下会不停的循环从阻塞队列中获取任务来执行，正常情况下就是通过阻塞来保证线程永远活着，但是会有一些特殊情况：

1. 如果线程被中断了，那就会退出循环，然后做一些善后处理，比如ctl中的工作线程数-1，然后自己运行结束
2. 如果线程阻塞超时了，那也会退出循环，此时就需要判断线程池中的当前工作线程够不够，比如是否有corePoolSize个工作线程，如果不够就需要新开一个线程，然后当前线程自己运行结束，这种看上去效率比较低，但是也没办法，当然如果当前工作线程数足够，那就正常，自己正常的运行结束即可
3. 如果线程是在执行任务的时候抛了移除，从而退出循环，那就直接新开一个线程作为替补，当然前提是线程池的状态是RUNNING

getTask方法

上面一直提到了getTask这个放，我们来看看这个方法。

```
private Runnable getTask() {
        boolean timedOut = false; // Did the last poll() time out?

        for (;;) {
                int c = ctl.get();
                int rs = runStateOf(c);

                // Check if queue empty only if necessary.
                // 如果线程池状态是STOP，表示当前线程不需要处理任务了，那就修改ctl工作线程数-1
                // 如果线程池状态是SHUTDOWN，但是阻塞队列中为空，表示当前任务没有任务要处理了，那就修改ctl工作线程数-1
                // return null表示当前线程无需处理任务，线程退出
                if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
                        decrementWorkerCount();
                        return null;
                }

                // 当前工作线程数
                int wc = workerCountOf(c);

                // Are workers subject to culling?
                // 用来判断当前线程是无限阻塞还是超时阻塞，如果一个线程超时阻塞，那么一旦超时了，那么这个线程最终就会退出
                // 如果是无限阻塞，那除非被中断了，不然这个线程就一直等着获取队列中的任务

                // allowCoreThreadTimeOut为true，表示线程池中的所有线程都可以被回收掉，则当前线程应该直接使用超时阻塞，一旦超时就回收
                // allowCoreThreadTimeOut为false，则要看当前工作线程数是否超过了corePoolSize，如果超过了，则表示超过部分的线程要用超时阻塞，一旦超时就回收

                boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

                // 如果工作线程数超过了工作线程的最大限制或者线程超时了，则要修改ctl，工作线程数减1，并且return null
                // return null就会导致外层的while循环退出，从而导致线程直接运行结束
                // 直播课程里会细讲timed && timedOut
                if ((wc > maximumPoolSize || (timed && timedOut))
                        && (wc > 1 || workQueue.isEmpty())) {
                        if (compareAndDecrementWorkerCount(c))
```

```
35                                  return null;
36                          continue;
37                  }
38
39
40                  try {
41                          // 要么超时阻塞，要么无限阻塞
42                          Runnable r = timed ? workQueue.poll(keepAliveTime,
TimeUnit.NANOSECONDS) : workQueue.take();
43
44                          // 表示没有超时，在阻塞期间获取到了任务
45                          if (r != null)
46                                  return r;
47
48                          // 超时了，重新进入循环，上面的代码会判断出来当前线程阻塞超时了，最后
return null，线程会运行结束
49                          timedOut = true;
50                  } catch (InterruptedException retry) {
51                          // 从阻塞队列获取任务时，被中断了，也会再次进入循环，此时并不是超时，
但是重新进入循环后，会判断线程池的状态
52                          // 如果线程池的状态变成了STOP或者SHUTDOWN，最终也会return null，线
程会运行结束
53                          // 但是如果线程池的状态仍然是RUNNING，那当前线程会继续从队列中去获取
任务，表示忽略了本次中断
54                          // 只有通过调用线程池的shutdown方法或shutdownNow方法才能真正中断线
程池中的线程
55                          timedOut = false;
56                  }
57          }
58  }
```

特别注意：只有通过调用线程池的shutdown方法或shutdownNow方法才能真正中断线程池中的线程。

因为在java，中断一个线程，只是修改了该线程的一个标记，并不是直接kill了这个线程，被中断的线程到底要不要消失，由被中断的线程自己来判断，比如上面代码中，线程遇到了中断异常，它可以选择什么都不做，那线程就会继续进行外层循环，如果选择return，那就退出了循环，后续就会运行结束从而消失。

shutdown方法

调用线程池的shutdown方法，表示要关闭线程池，不接受新任务，但是要把阻塞队列中剩余的任务执行完。

根据前面execute方法的源码，只要线程池的状态不是RUNNING，那么就表示线程池不接受新任务，所以shutdown方法要做的第一件事情就是修改线程池状态。

那第二件事情就是要中断线程池中的工作线程，这些工作线程要么在执行任务，要么在阻塞等待任务：

1. 对于在阻塞等待任务的线程，直接中断即可，
2. 对于正在执行任务的线程，其实只要等它们把任务执行完，就可以中断了，因为此时线程池不能接受新任务，所以正在执行的任务就是最后剩余的任务

```
public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                checkShutdownAccess();
                // 修改ctl，将线程池状态改为SHUTDOWN
                advanceRunState(SHUTDOWN);
                // 中断工作线程
                interruptIdleWorkers();
                // 空方法，给子类扩展使用
                onShutdown(); // hook for ScheduledThreadPoolExecutor
        } finally {
                mainLock.unlock();
        }
        // 调用terminated方法
        tryTerminate();
}
```

```
private void interruptIdleWorkers() {
        interruptIdleWorkers(false);
}


private void interruptIdleWorkers(boolean onlyOne) {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                // 遍历所有正在工作的线程，要么在执行任务，要么在阻塞等待任务
                for (Worker w : workers) {
                        Thread t = w.thread;

                        // 如果线程没有被中断，并且能够拿到锁，就中断线程
                        // Worker在执行任务时会先加锁，执行完任务之后会释放锁
                        // 所以只要这里拿到了锁，就表示线程空出来了，可以中断了
                        if (!t.isInterrupted() && w.tryLock()) {
                                try {
                                        t.interrupt();
                                } catch (SecurityException ignore) {
                                } finally {
                                        w.unlock();
                                }
                        }
                        if (onlyOne)
                                break;
                }
        } finally {
                mainLock.unlock();
        }
}
```

不过还有一个种情况，就是目前所有工作线程都在执行任务，但是阻塞队列中还有剩余任务，那逻辑应该就是这些工作线程执行完当前任务后要继续执行队列中的剩余任务，但是根据我们看到的shutdown方法的逻辑，发现这些工作线程在执行完当前任务后，就会释放锁，那就可能会被中断掉，那队列中剩余的任务怎么办呢?

工作线程一旦被中断，就会进入processWorkerExit方法，根据前面的分析，我们发现，在这个方法中会会线程池状态为SHUTDOWN进行判断，会重新生成新的工作线程，那么这样就能保证队列中剩余的任务一定会被执行完。

shutdownNow方法

看懂了shutdown方法，再来看shutdownNow方法就简单了。

```
public List<Runnable> shutdownNow() {
        List<Runnable> tasks;
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                checkShutdownAccess();
                // 修改ctl，将线程池状态改为STOP
                advanceRunState(STOP);
                // 中断工作线程
                interruptWorkers();
                // 返回阻塞队列中剩余的任务
                tasks = drainQueue();
        } finally {
                mainLock.unlock();
        }

        // 调用terminated方法
        tryTerminate();
        return tasks;
}
```

```
private void interruptWorkers() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
                // 中断所有工作线程，不管有没有在执行任务
                for (Worker w : workers)
                        w.interruptIfStarted();
        } finally {
                mainLock.unlock();
        }
}


void interruptIfStarted() {
        Thread t;

        // 只要线程没有被中断，那就中断线程，中断的线程虽然也会进入processWorkerExit方法，但是该方法中判断了线程池的状态
        // 线程池状态为STOP的情况下，不会再开启新的工作线程了
        // 这里getState>-0表示，一个工作线程在创建好，但是还没运行时，这时state为-1，可以看看Worker的构造方法就知道了
        // 表示一个工作线程还没开始运行，不能被中断，就算中断也没意义，都还没运行
        if (getState() >= 0 && (t = thread) != null && !t.isInterrupted()) {
                try {
                        t.interrupt();
                } catch (SecurityException ignore) {
                }
        }
}
```

mainLock

在上述源码中，发现很多地方都会用到mainLock，它是线程池中的一把全局锁，主要是用来控制workers集合的并发安全，因为如果没有这把全局锁，就有可能多个线程公用同一个线程池对象，如果一个线程在向线程池提交任务，一个线程在shutdown线程池，如果不做并发控制，那就有可能线程池shutdown了，但是还有工作线程没有被中断，如果1个线程在shutdown，99个线程在提交任务，那么最终就可能导致线程池关闭了，但是线程池中的很多线程都没有停止，仍然在运行，这肯定是不行，所以需要这把全局锁来对workers集合的操作进行并发安全控制。

到此，线程池中的所有核心方法的源码都分析一遍，自我感觉良好，不知道你啥感觉，哈哈，希望你能看懂，看不懂的欢迎交流。