文章已收录我的仓库：Java学习笔记

线程池原理

池化技术

我们平时使用线程时，都是额外创建一个线程(Thread)去执行任务(run方法)，run 方法执行完毕后就会销毁线程，频繁的创建和销毁无疑增加了开销，而池化技术则可以帮我们很好的管理线程。

使用线程池后，使用线程池后，**线程执行完一个 run 方法不再是无脑销毁，而是根据需要可能会留在池中，进而继续执行下次任务**，降低资源消耗。并且，使用线程池还可以让我们更方便的管理，例如我们可以根据系统资源指定最大线程数，根据需要延时执行任务；又例如，线程池会帮我们统计已经运行了多少个任务、最多同时运行了几个任务等，便于我们排错，简言之，线程池具有如下好处：

- 避免频繁创建销毁，降低资源消耗，提高任务响应速度。
- 显示指定参数，便于管理线程。
- 线程池会记录一些信息，便于管理、排错。

线程池管理的是线程，我们提交给线程池的是任务(run 方法，线程池中称为命令)，线程池根据参数配置以及当前情况，决定是新建线程还是用已有的线程去执行任务，这一点是学习线程池的前提。

> 本文 JDK 版本 11

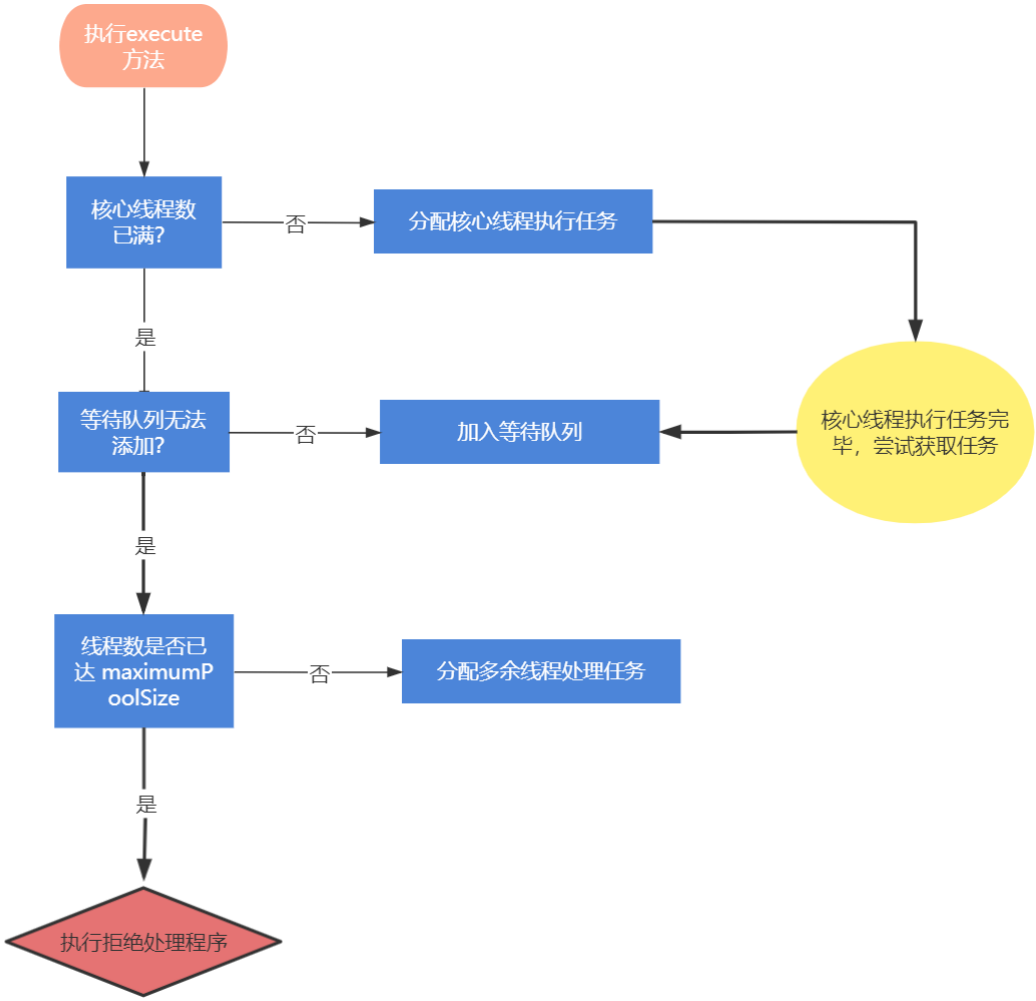
参数

```
使用给定的初始参数创建一个新的ThreadPoolExecutor 。
参数:  corePoolSize – 要保留在池中的线程数，即使它们处于空闲状态，除非设置了
       allowCoreThreadTimeOut
       maximumPoolSize – 池中允许的最大线程数
       keepAliveTime – 当线程数大于核心数时，这是多余空闲线程在终止前等待新任务的最长时
       间。
       unit – keepAliveTime参数的时间单位
       workQueue – 用于在执行任务之前保存任务的队列。 这个队列将只保存execute方法提交
       的Runnable任务。
       threadFactory – 执行程序创建新线程时使用的工厂
       handler – 执行被阻塞时使用的处理程序，因为达到了线程边界和队列容量
抛出:  IllegalArgumentException – 如果以下情况之一成立:
       corePoolSize < 0
       keepAliveTime < 0
       maximumPoolSize <= 0
       maximumPoolSize < corePoolSize
       NullPointerException – 如果workQueue或threadFactory或handler为 null
public ThreadPoolExecutor( @Range(from = 0, to = java.lang.Integer.MAX_VALUE) int corePoolSize,
                           @Range(from = 1, to = java.lang.Integer.MAX_VALUE) int maximumPoolSize,
                           @Range(from = 0, to = java.lang.Long.MAX_VALUE) long keepAliveTime,
                           @NotNull TimeUnit unit,
                           @NotNull BlockingQueue<Runnable> workQueue,
                           @NotNull ThreadFactory threadFactory,
                           @NotNull RejectedExecutionHandler handler) {
```

- corePoolSize：要保留在池中的线程数，即线程池中最少拥有的线程数(不考虑懒加载)，即使这些线程都是空闲的，也不会被回收，除非显式的设置了 allowCoreThreadTimeOut = true，这些线程被称为核心线程。
- maximumPoolSize：池中允许的最大线程数。
- keepAliveTime： 当线程数大于核心数时，这是多余空闲线程在终止前等待新任务的最长时间，线程池只会回收非核心线程(默认情况下)。
- unit：keepAliveTime参数的时间单位。
- workQueue：用于在执行任务之前保存任务的队列，这个队列将只保存execute方法提交的Runnable任务。当核心线程都在工作时，新任务被加进该队列进行缓存，等待核心线程空闲。如果队列已满并且当前线程数小于 maximumPoolSize，则会直接创建一个新的线程执行任务。
- threadFactory：执行程序创建新线程时使用的工厂。
- handler：执行被阻塞时使用的处理程序，因为达到了线程边界和队列容量，即线程数达到了 maximumPoolSize，并且 workQueue 容量也满，此时线程池处于饱和状态，拒绝该任务。

原理

执行流程



发现线程池有一个思想是：**能用核心线程处理就用核心线程处理，核心线程满了就加入等待队列等等核心线程处理完毕，要是队列已满，迫不得已才创建新线程处理。**

名称	描述
ArrayBlockingQueue	一个用数组实现的有界阻塞队列，此队列按照先进先出(FIFO)的原则对元素进行排序。支持公平锁和非公平锁。
LinkedBlockingQueue	一个由链表结构组成的有界队列，此队列按照先进先出(FIFO)的原则对元素进行排序。此队列的默认长度为Integer.MAX_VALUE，所以默认创建的该队列有容量危险。
PriorityBlockingQueue	一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序。
DelayQueue	一个实现PriorityBlockingQueue实现延迟获取的无界队列，在创建元素时，可以指定多久才能从队列中获取当前元素。只有延时期满后才能从队列中获取元素。
SynchronousQueue	一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素。支持公平锁和非公平锁。SynchronousQueue的一个使用场景是在线程池里。Executors.newCachedThreadPool()就使用了SynchronousQueue，这个线程池根据需要（新任务到来时）创建新的线程，如果有空闲线程则会重复使用，线程空闲了60秒后会被回收。
LinkedTransferQueue	一个由链表结构组成的无界阻塞队列，相当于其它队列，LinkedTransferQueue队列多了transfer和tryTransfer方法。
LinkedBlockingDeque	一个由链表结构组成的双向阻塞队列。队列头部和尾部都可以添加和移除元素，多线程并发时，可以将锁的竞争最多降到一半。

像前面两种堵塞队列都是有界的，因此存在添加队列失败的情况。

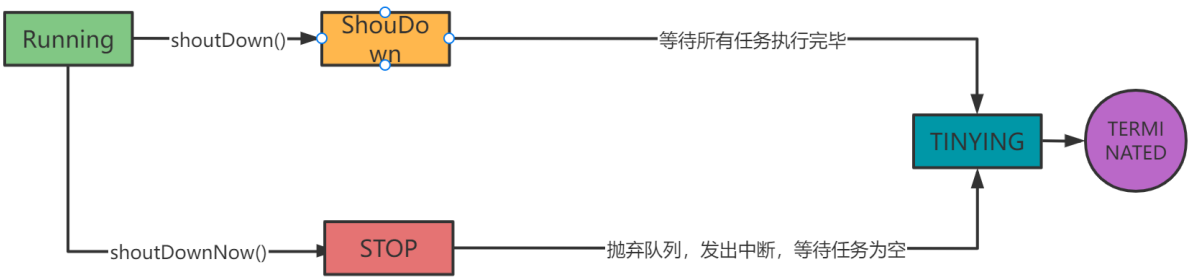
线程池状态

```
        private final AtomicInteger ctl = new
AtomicInteger(ctlOf(RUNNING, 0));
    private static final int COUNT_BITS = Integer.SIZE - 3;
    private static final int COUNT_MASK = (1 << COUNT_BITS) - 1;

    // runState is stored in the high-order bits
    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;

    // Packing and unpacking ctl
    private static int runStateOf(int c)     { return c & ~COUNT_MASK; }
    private static int workerCountOf(int c)  { return c & COUNT_MASK; }
    private static int ctlOf(int rs, int wc) { return rs | wc; }
        private static boolean runStateLessThan(int c, int s) {return c <
s; }
    private static boolean runStateAtLeast(int c, int s) {return c >= s;}
    private static boolean isRunning(int c) {  return c < SHUTDOWN; }
```

- RUNNING：能接受新提交的任务，并且也能处理阻塞队列中的任务，即存在核心线程空闲。
- SHUTDOWN：指调用了 shutdown() 方法，不再接受新提交的任务，但**却可以继续处理既有的任务以及阻塞队列中已保存的任务**。
- STOP：指调用了 shutdownNow() 方法，不再接受新提交的任务，同时**抛弃阻塞队列里的所有任务并中断所有正在执行任务**。
- TIDYING： 所有任务都执行完毕，workerCount 有效线程数为 0。
- TERMINATED：终止状态，当执行 terminated() 后会更新为这个状态。



研究源码发现，大哥李(线程池类编写者，并发大神)定义了一个原子数 ctl，这个数前 3 位保存线程池的 5 大状态，后 29 位保存 workerCount，即当前有效线程数。以及一系列方法来判断当前线程状态以及获取有效线程数量。

Worker

```
private final class Worker extends AbstractQueuedSynchronizer implements
Runnable {
    /** 当前 Worker 的线程 */
    final Thread thread;
    /** 任务，可能为空 */
    Runnable firstTask;
    /** 任务计数器，即该 Worker 执行了几个任务 */
    volatile long completedTasks;
    Worker(Runnable firstTask) {
        setState(-1); // inhibit interrupts until runWorker
        this.firstTask = firstTask;
        this.thread = getThreadFactory().newThread(this);
    }
    public void run() {
        runWorker(this);
    }
    // 省略 AQS 方法
}
```

Worker 类在线程池中就是我们所说的线程，线程池管理线程，事实上就是管理一个一个的 Worker，在这个 Worker 内管理了一个线程类，注意看 Worker 本身就是实现了 Runnable，**该线程类实例 thread 是调用我们传入的线程工厂以 Worker 本身创建的，因此 thread.start() 会调用 Worker 的 run 方法**。

Worker 还实现了 AQS 类，主要目的有 2 个：

- 将锁的粒度细化到每个 Worker。如果多个 Worker 使用同一个锁，那么一个 Worker Running 持有锁的时候，其他 Worker 就无法执行，这显然是不合理的。
- 使用不可重入锁。因为 Worker 可能调用控制线程池的方法，这显然是不合理的，我们不希望它重新获取锁。

执行任务，线程复用

如何做到线程复用？这如下两个问题：

1. 如何让一个线程运行多个任务？
2. 线程执行 run 方法后就会进入中止状态，如何复用？

我们来回答这两个问题：

第一个问题：这简单，将多个任务看作是一个任务即可，在线程池中，线程会依次从队列中取出任务执行。

第二个问题：答案是我们无法让线程死而复生，如果你理解了第一个问题的答案，你应该会想到既然无法让线程死而复生，就干脆不要让他死，没错，使用 while 循环让线程一直活下去！

嗯？while 循环太消耗资源？对，没错，所以线程池的参数是堵塞队列而不是其他队列，当队列为空时线程会堵塞，依赖于堵塞队列的底层实现（例如 ArrayBlockingQueue 使用条件变量），线程可能陷入休眠，释放 CPU，节省资源。

当一个 Worker 被添加至线程池中，线程中会执行如下代码(addWorker 方法中，后面会说)：

```
Thread t = Worker.thread;
t.start();
```

而我们都知道 thread.start() 其实内部会调用 run 方法，上面讲过 thread.start() 会调用 Worker 的 run 方法，即 `runWorker(this);`，现在来看看这个关键代码（省略部分）：

```
final void runWorker(Worker w) {
        Runnable task = w.firstTask;
        w.firstTask = null;
        w.unlock(); // allow interrupts
        boolean completedAbruptly = true;
        try {
            // 线程不死，无限循环，可以人为干预中断
            // getTask 方法从堵塞队列中取任务，如果线程数小于核心线程数，会直接分配 task，因此这里的 task 初始可能不为空
            while (task != null || (task = getTask()) != null) {
                w.lock();
                if (线程池是终止状态)
                    中断线程
                try {
                    task.run(); // 执行任务
                } catch (Throwable ex) {
                    throw ex;
                } finally {
                    task = null;
                    w.completedTasks++; // 记录完成任务数
                    w.unlock();
                }
            }
            completedAbruptly = false;
        } finally {
            // 线程跳出循环，即将死亡，将这个 Worker 移出线程池
            processWorkerExit(w, completedAbruptly);
        }
    }
```

这短短 28 行代码就是 Worker 工作的核心代码，是不是很神奇？

你可能会问，线程如何跳出循环？例如，我们传递了 keepAliveTime，我们期望非核心线程超时能够停止。

当 getTask() == null 时会跳出循环，来看看大哥李的注释文档：

> 执行阻塞或定时等待任务，具体取决于当前配置设置，或者如果由于以下任何原因而必须退出此工作程序，则返回 null：1. 有超过 maximumPoolSize 的工作程序（由于调用 setMaximumPoolSize）。2. 池停止。3. 池关闭，队列为空。4. 这个worker等待任务超时，超时的worker在定时等待前后都会被终止（即allowCoreThreadTimeOut || workerCount > corePoolSize），如果队列不为空，这个worker不是池中的最后一个线程。
>
> 返回：任务，如果工人必须退出，则为 null，在这种情况下，workerCount 递减

```
private Runnable getTask() {
```

即，如下情况返回 null，终止 Worker 循环：

- 超过 maximumPoolSize，这个参数可能会被用户可能动态减小。
- 线程池处于终止状态（STOP）。
- 线程池处于关闭状态（SHOTDOWN），并且队列为空。
- 如果 workerCount > corePoolSiz，此时存在非核心线程，因此当线程运行时间超过 keepAliveTime 视为超时，返回 null（**注意没有标记哪个线程是非核心线程，谁先来并且符合超时条件就会暂停谁**）；如果用户设置 allowCoreThreadTimeOut = true，则核心线程同样处理，否则，允许核心线程永久存在。

至此，我们基本搞明白了 Worker 工作原理，**还要注意，虽然我们一直在说核心线程和非核心线程，但事实上并没有任何字段标记 Worker 是否是核心的，所有 Worker 都是一样的，只是会根据核心线程数和最大线程数的关系去逻辑的认为谁是核心线程谁是非核心线程，例如当 workerCount <= corePoolSiz，认为所有的线程都是核心线程；而当 当 workerCount > corePoolSiz，谁先进入 getTask 判断并且符合超时条件就会暂停谁，那么我们就认为这是非核心线程。**

现在我们来看看 Worker 何时开始工作。

Worker 何时开始工作

我们从用户代码开始探究：

```
ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newCachedThreadPool();
executor.execute(()->{
    System.out.println(Thread.currentThread());
});
```

execute 函数

```
public void execute(Runnable command) {
        if (command == null)
            throw new NullPointerException();
        int c = ctl.get();
        // 如果此时有效线程数小于核心线程数的话，addWorker 第二个参数为 true
```

```
        if (workerCountOf(c) < corePoolSize) {
            if (addWorker(command, true))
                return;
            c = ctl.get();
        }
        // 否则，核心线程数已满，如果当前处于 RUNNING，说明核心线程能够处理队列中
的任务
        // 添加任务到等待队列
        if (isRunning(c) && workQueue.offer(command)) {
            int recheck = ctl.get();
            // 如果添加队列成功，再次判断一下线程池状态，如果线程池终止的话，拒绝
            if (! isRunning(recheck) && remove(command))
                reject(command);
            // 如果此时的工作线程为 0
            else if (workerCountOf(recheck) == 0)
                addWorker(null, false);
        }
        // 队列添加失败时，注意这里参数，第二个是 false
        // 如果 addWorker 失败返回 false，则拒绝处理
        else if (!addWorker(command, false))
            reject(command);
    }
```

发现还有很大一部分逻辑隐藏在 addWorker 函数中。

addWorker 函数

```
private boolean addWorker(Runnable firstTask, boolean core) {
    retry:
    for (int c = ctl.get();;) {
        if (健壮性检查)
            return false;

        for (;;) {
            // 注意第二个参数，如果 true 则和 corePoolSize 比较，否则和
maximumPoolSize 比较
            if (workerCountOf(c) >= ((core ? corePoolSize :
maximumPoolSize) & COUNT_MASK))
                return false;
            // CAS 尝试增加线程数，增加成功后则跳出两层循环，使用 CAS 防止并发问
题
            if (compareAndIncrementWorkerCount(c))
                break retry;
            c = ctl.get();  // Re-read ctl

            // 如果线程可能处于关闭状态(大于等于)，则重新开始外层循环，外层循环会
进行健壮性检查，如果线程池确实关闭，会返回 false
            if (runStateAtLeast(c, SHUTDOWN))
                continue retry; // 这个语句是再次进入外层循环的意识
            // else CAS failed due to workerCount change; retry inner
loop
        }
    }
```

```
        // 此时线程数已经成功增加

    boolean workerStarted = false;
    boolean workerAdded = false;
    Worker w = null;
    try {
        // 构造工人
        w = new Worker(firstTask);
        final Thread t = w.thread;
        if (t != null) {
            final ReentrantLock mainLock = this.mainLock;
            mainLock.lock();
            try {
                // Recheck while holding lock.
                // Back out on ThreadFactory failure or if
                // shut down before lock acquired.
                int c = ctl.get();

                // 如果线程池是 RUNNING状态
                // 或者是小于 STOP 状态（SHUTDOWN）并且 firstTask 为 null
                if (isRunning(c)||(runStateLessThan(c, STOP) && firstTask
== null)) {
                    // 添加到线程池 workers 中，这是个 HashSet<Worker>
                    workers.add(w);
                    workerAdded = true;
                    int s = workers.size();
                    if (s > largestPoolSize)
                        largestPoolSize = s;
                }
            } finally {
                mainLock.unlock();
            }
            // 如果成功添加，执行 start，注意这就是我们上面说的Worker工作启动
            if (workerAdded) {
                t.start();
                workerStarted = true;
            }
        }
    } finally {
        if (! workerStarted)
            addWorkerFailed(w);
    }
    return workerStarted;
}
```

可能会有点懵，但我们根据 execute 引用的 addWorker 来依次判断一下：

1. **addWorker(commend, true)：**当线程数小于核心线程数时调用，core = true，因此在 addWorker 中会与 corePoolSize 重新比较，防止并发情况下线程数已经大于 corePoolSize，在第二个循环中上锁判断，正常来说当线程数小于核心线程数时，线程状态是 RUNNING，因此会正常创建 Worker 并运作 Worker。

2. **addWorker(null, false)：**在 execute 中，这个调用是在任务已经添加到队列中，但突然工作线程为 0 时(并发问题)调用，此时工作线程为 0，因此我们至少要添加一个 Worker，而由于我们的命令 Task 已经添加至堵塞队列了，所以这里的 task = null，Worker 会自动从队列中获取任务。
3. **addWorker(commend, false)：**这个没啥说的，就是第二个参数变了，此在 addWorker 中会与 maximumPoolSize 重新比较，这里的 Worker 应该是非核心线程。

从结果上说，addWorker 就是添加了一个 Worker，不过函数中多了很多 CAS 操作防止并发问题。

那么整个添加运行过程就讲完了，流程确实与我们流程图画的一致，不过是多了些并发问题判断。

下面说说关闭。

Worker 关闭

线程的 stop 方法已经被废弃了，因为直接 stop 线程可能会导致某些锁未释放等 BUG 出现。

取而代之的是 interrupt 方法，即设置中断标志，当线程检查到中断时会抛出中断异常，从而我门可以捕获，进行一些操作，例如释放锁。

> 注意中断并不能是线程立即退出，当线程在休眠时则无能为力，因为中断标志是需要线程主动轮询的。

shutdown

shoutdown 方法首先设置线程为 **SHUTDOWN** 状态，然后主要调用了 interruptIdleWorkers(false) 方法，我们来看看这个方法：

```
private void interruptIdleWorkers(boolean onlyOne) {
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        for (Worker w : workers) {
            Thread t = w.thread;
            // 尝试获取 worker 的锁
            if (!t.isInterrupted() && w.tryLock()) {
                try {
                        // 获取成功了，中断线程
                    t.interrupt();
                } catch (SecurityException ignore) {
                } finally {
                    w.unlock();
                }
            }
            if (onlyOne)
```

```
                    break;
            }
        } finally {
            mainLock.unlock();
        }
    }
```

这个方法逻辑很简单，用 CAS 操作尝试获取每个 worker 的锁，获取成功了说明它们不在运行中，则设置中断；否则，**worker 正在运行，无法获取锁，让其继续运行下去。**

由于线程状态为 SHUTDOWM，根据 getTask() 的函数返回，如果还有 Worker 存活，他会不停执行任务直到队列为空，**因此 SHUTDOWM 方法不会停止正在运行的任务并且会继续运行队列中存在的任务。**

shutdownNow

这个函数会设置线程状态为 STOP，然后调用 interruptWorkers 方法中断所有线程。

```
    /**
     * Interrupts all threads, even if active. Ignores SecurityExceptions
     * (in which case some threads may remain uninterrupted).
     */
    private void interruptWorkers() {
        // assert mainLock.isHeldByCurrentThread();
        for (Worker w : workers)
            w.interruptIfStarted();
    }
```

注意看大哥李的注释，翻译是 `中断所有线程，即使是活动线程。忽略 SecurityExceptions（在这种情况下，某些线程可能保持不间断）`，**发现该方法忽略了 SecurityExceptions 异常，因此如果存在 SecurityExceptions，可能会导致部分线程未被中断，面试要考！**

具体源码就不看了。

线程池工厂

通过线程池工厂便捷的创建线程池，常用的是：

newFixedThreadPool

```
创建一个线程池，该线程池重用固定数量的线程在共享的无界队列中运行。 在任何时候，最多有
nThreads线程是活动的处理任务。 如果在所有线程都处于活动状态时提交了额外的任务，它们将在
队列中等待，直到有线程可用。 如果任何线程在关闭前的执行过程中由于失败而终止，则在需要执
行后续任务时，将有一个新线程代替它。 池中的线程将一直存在，直到它被明确shutdown 。
参数： nThreads – 池中的线程数
返回： 新创建的线程池
抛出： IllegalArgumentException – 如果nThreads <= 0
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

该方法核心线程数和最大线程数相等，因此没有业余线程，而核心线程不会消息，因此它是一个**固定线程数量的线程池。**

newCachedThreadPool

```
创建一个线程池，根据需要创建新线程，但在可用时将重用先前构造的线程。 这些池通常会提高执
行许多短期异步任务的程序的性能。 如果可用，调用execute将重用先前构造的线程。 如果没有可
用的现有线程，则会创建一个新线程并将其添加到池中。 60 秒内未使用的线程将被终止并从缓存中
删除。 因此，保持空闲足够长时间的池不会消耗任何资源。 请注意，可以使用
ThreadPoolExecutor构造函数创建具有相似属性但不同细节（例如，超时参数）的
ThreadPoolExecutor 。
返回： 新创建的线程池
@NotNull
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
                                   keepAliveTime: 60L, TimeUnit.SECONDS,
                                   new SynchronousQueue<Runnable>());
}
```

该方法核心线程数为 0，也就是说如果没有任务，则不会存在任何线程在线程池中。由于队列是 Sync，这个队列的特性是无容量的，即如果没有消费者正在 take，添加永远都会失败，因此每个任务会启动一个新的线程去运行，所以我们说这是一个**可根据实际任务情况调整线程个数的线程池。**

newSingleThreadExecutor

```
创建一个 Executor，它使用单个工作线程在无界队列中运行。 （但是请注意，如果这个单线程在关闭之前的执行过程中由于失败而终止，如果需要执行后续任务，一个新线程将取而代之。）保证任务按顺序执行，并且不会超过一个任务处于活动状态在任何给定的时间。 与其他等效的 newFixedThreadPool(1)，返回的执行程序保证不可重新配置以使用其他线程。

返回： 新创建的单线程 Executor

@NotNull
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor( corePoolSize: 1,  maximumPoolSize: 1,
                                 keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                 new LinkedBlockingQueue<Runnable>()));
}
```

这将使用单个线程串行的执行每个任务，是一个一个线程数量为 1 的线程池。

除此之外，还有对线程池的扩展，例如 newScheduledThreadPool 可以返回一个可延迟执行的线程池，等等。