# 

# 快速开始：



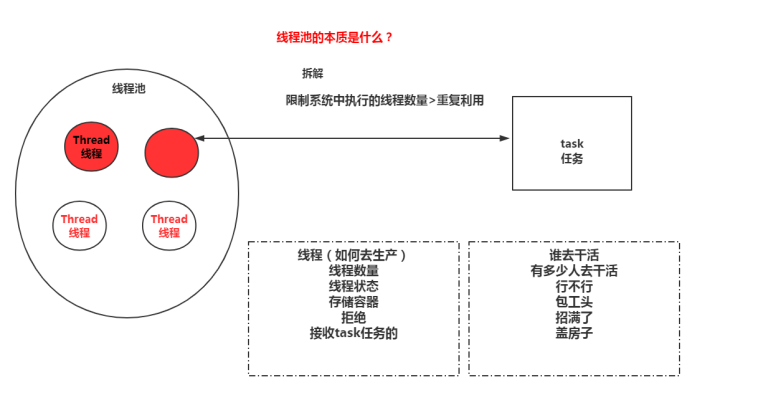
为什么？

LinkedBlockingQueue：无界队列

ArrayBlockingQueue:有界队列

SynchronousQueue：阻塞队列（不会有等待task>core max）

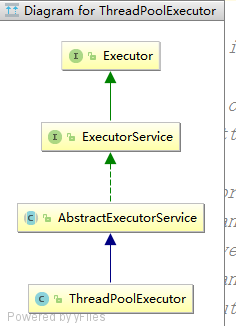
# 技术的本质：



# 手写实现：

肯定是要参考java的ThreadPoolExecutor

类的关系图





保证时间到期了 然后销毁线程（不加入队列）

### 队列：

**SynchronousQueue**：这个队列接收到任务的时候，会直接提交给线程处理，而不保留它，如果所有线程都在工作怎么办？那就新建一个线程来处理这个任务！所以为了保证不出现 线程数达到了 maximumPoolSize 而不能新建线程 的错误，使用这个类型队列的时候，maximumPoolSize 一般指定成 Integer.MAX\_VALUE，即无限大。

**LinkedBlockingQueue**：这个队列接收到任务的时候，如果当前线程数小于核心线程数，则新建线程(核心线程)处理任务；如果当前线程数等于核心线程数，则进入队列等待。由于这个队列没有最大值限制，即所有超过核心线程数的任务都将被添加到队列中，这也就导致了 maximumPoolSize 的设定失效，因为总线程数永远不会超过 corePoolSize。

**ArrayBlockingQueue**：可以限定队列的长度，接收到任务的时候，如果没有达到 corePoolSize 的值，则新建线程(核心线程)执行任务，如果达到了，则入队等候，如果队列已满，则新建线程(非核心线程)执行任务，又如果总线程数到了 maximumPoolSize，并且队列也满了，则发生错误。

**DelayQueue**：队列内元素必须实现 Delayed 接口，这就意味着你传进去的任务必须先实现 Delayed 接口。这个队列接收到任务时，首先先入队，只有达到了指定的延时时间，才会执行任务。

 ArrayBlockingQueue：构造函数一定要传大小

 LinkedBlockingQueue：构造函数不传大小会默认为Integer.MAX\_VALUE ，当大量请求任务时，容易造成 内存耗尽

 SynchronousQueue：同步队列，一个没有存储空间的阻塞队列 ，将任务同步交付给工作线程

 PriorityBlockingQueue : 优先队列

# 作业：

1、Shutdown方法

2、ThreadPoolExecutor监控数据

3、了解线程池源码

完成一个就可以，提交到源码中心来

# Executor框架体系

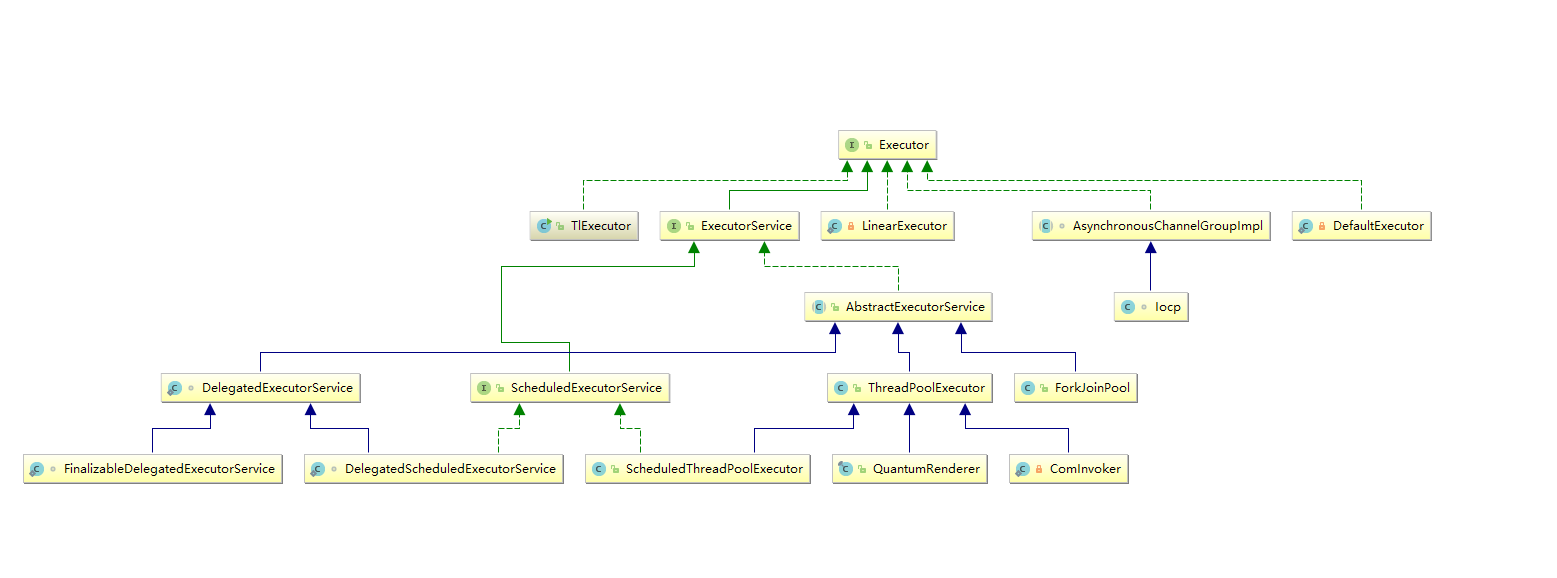
Java.util.concurrent中的Executor(中文翻译是执行器)是管理咱们之前的Thread线程的。目的是是简化并发编程

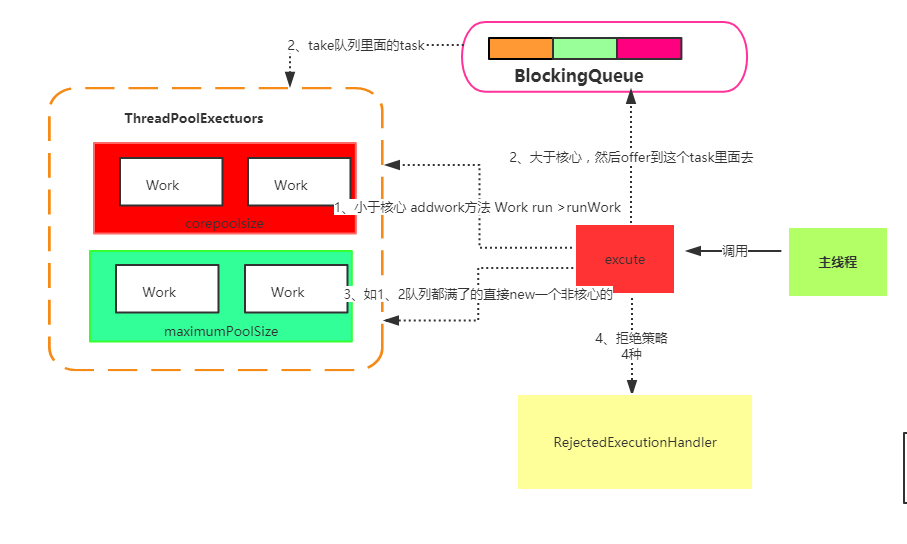
### ExecutorService（执行器）

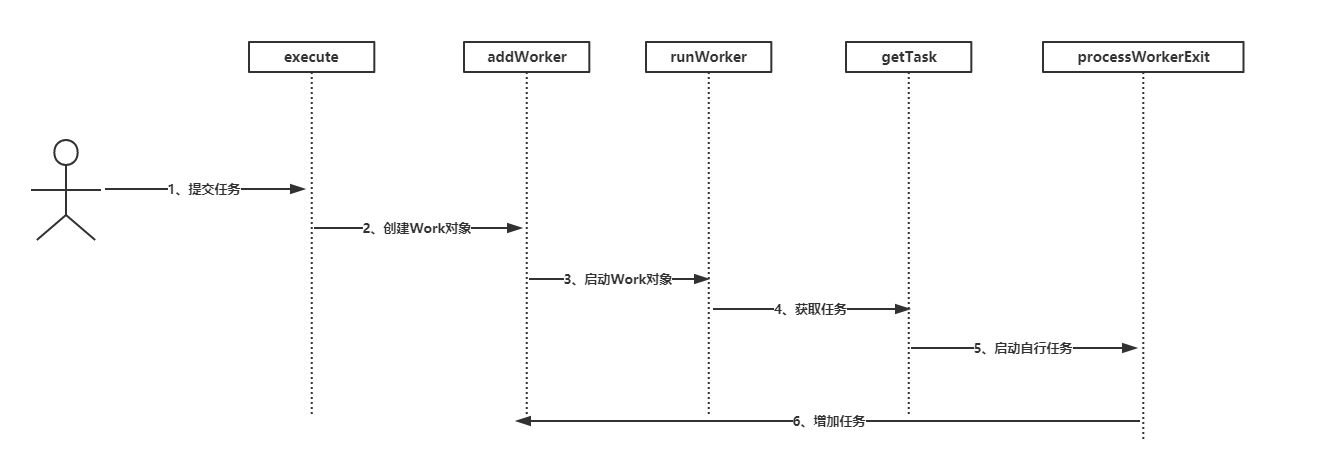
### Java两种基础线程池

普通：ThreadPoolExecutor

定时ScheduledThreadPoolExecutor







### 层级关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **层级** | **名称** | **方法** | **说明** | **类型** |
| 1 | java.util.concurrent.Executor | java.util.concurrent.Executor#execute | 执行接口 | 接口 |
| 2 | java.util.concurrent.ExecutorService | java.util.concurrent.ExecutorService#submit(java.util.concurrent.Callable<T>) | 提交接口 | 接口 |
| 3 | java.util.concurrent.AbstractExecutorService | java.util.concurrent.AbstractExecutorService#submit(java.util.concurrent.Callable<T>) | 把执行和提交接口进行合并  区别:有返回值和无返回值 | 抽象类 |
| 4 | java.util.concurrent.ThreadPoolExecutor | java.util.concurrent.ThreadPoolExecutor#execute | 调用addwork（offer>task放队列）  Run方法调用runwork方法 getTask（从队列拿数据） | 实现类 |
| 5 | java.util.concurrent.ScheduledExecutorService | Schedule、scheduleAtFixedRat、scheduleWithFixedDelay | 定义方法  定义接口 | 接口  接口 |
| 6 | java.util.concurrent.ScheduledThreadPoolExecutor | delayedExecute | 具体实现  add>task>addWork | 实现类 |

源码分析：

|  |
| --- |
| **public static void** main(String[] args) {  ExecutorService executorService= Executors.*newFixedThreadPool*(1);*//构建 并且创建线程池赋值* executorService.submit(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(**"悟空是只猴子"**);  }  }); } |

### 初始化new ThreadPoolExecutor

构造器：核心数量，任务队列容器，存活时间，线程工厂，处理器

|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,**//存活时间**  TimeUnit unit,  BlockingQueue<Runnable> workQueue, **//阻塞队列**  ThreadFactory threadFactory,**//线程工厂**  RejectedExecutionHandler handler) {**//hander处理** |

### LinkedBlockingQueue实现

### Execute（submit）方法

java.util.concurrent.ThreadPoolExecutor#execute

|  |
| --- |
| **if** (command == **null**)  **throw new** NullPointerException();**int** c = **ctl**.get();  //判断是否小于核心数量，是直接新增work成功后直接退出  **if** (*workerCountOf*(c) < **corePoolSize**) {  **if** (addWorker(command, **true**))  **return**;  c = **ctl**.get()**;// 增加失败后继续获取标记** }  **//判断是运行状态并且扔到workQueue里成功后** **if** (*isRunning*(c) && **workQueue**.offer(command)) {  **int** recheck = **ctl**.get();  **//再次check判断运行状态如果是非运行状态就移除出去&reject掉**  **if** (! *isRunning*(recheck) && remove(command))  reject(command);  **else if** (*workerCountOf*(recheck) == 0) **//否则发现可能运行线程数是0那么增加一个null的worker。**  addWorker(**null**, **false**); } **else if** (!addWorker(command, **false**)) **//直接增加worker如果不成功直接reject**  reject(command); |

**简要分析一下execute源码**：执行一个Runnable对象时，首先通过workerCountOf(c)获取线程池中线程的数量，如果池中的数量小于corePoolSize就调用addWorker添加一个线程来执行这个任务。否则通过workQueue.offer(command)方法入列。如果入列成功还需要在一次判断池中的线程数，因为我们创建线程池时可能要求核心线程数量为0，所以我们必须使用addWorker(null, false)来创建一个临时线程去阻塞队列中获取任务来执行。

### addWorker方法

|  |
| --- |
| retry: **for** (;;) {  **int** c = **ctl**.get();  **int** rs = *runStateOf*(c);   *// Check if queue empty only if necessary.* **if** (rs >= ***SHUTDOWN*** &&  ! (rs == ***SHUTDOWN*** &&  firstTask == **null** &&  ! **workQueue**.isEmpty()))  **return false**;**// 两种情况1.如果非关闭状态 2.不是这种情况（停止状态并且是null对象并且workQueue不等于null）**   **for** (;;) {  **int** wc = *workerCountOf*(c);  **if** (wc >= ***CAPACITY*** ||  wc >= (core ? **corePoolSize** : **maximumPoolSize**))  **return false**;**// 判断是否饱和容量了**  **if** (compareAndIncrementWorkerCount(c)) //增加一个work数量 然后跳出去  **break** retry;  c = **ctl**.get(); **//增加work失败后继续递归****if** (*runStateOf*(c) != rs)  **continue** retry;  *// else CAS failed due to workerCount change; retry inner loop* } }  **boolean** workerStarted = **false**; **boolean** workerAdded = **false**; Worker w = **null**; **try** {  w = **new** Worker(firstTask);**//增加一个worker**  **final** Thread t = w.**thread**;  **if** (t != **null**) {/**/判断是否 为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.*  **锁定后并重新检查下 是否存在线程工厂的失败或者锁定前的关闭** **int** rs = *runStateOf*(**ctl**.get());   **if** (rs < ***SHUTDOWN*** ||  (rs == ***SHUTDOWN*** && firstTask == **null**)) {  **if** (t.isAlive()) *// precheck that t is startable* **throw new** IllegalThreadStateException();   **workers**.add(w); **//增加work**  **int** s = **workers**.size();  **if** (s > **largestPoolSize**)  **largestPoolSize** = s;  workerAdded = **true**;  }  } **finally** {  mainLock.unlock();  }  **if** (workerAdded) { **//本次要是新增加work成功就调用start运行**  t.start();  workerStarted = **true**;  }  } } **finally** {  **if** (! workerStarted)  addWorkerFailed(w); } **return** workerStarted; |

总结：//这个两个for循环主要是判断能否增加一个线程，

    //外循环来判断线程池的状态

//内循环主要是个增加线程数的CAS操作

 第一个参数firstTask不为null，则创建的线程就会先执行firstTask对象，然后去阻塞队列中取任务，否直接到阻塞队列中获取任务来执行。第二个参数，core参数为真，则用corePoolSize作为池中线程数量的最大值；为假，则以maximumPoolSize作为池中线程数量的最大值。

T.start方法 run方法执行>调用了runWorker方法

### runWorker

|  |
| --- |
| Thread wt = Thread.*currentThread*();**//1.取到当前线程** Runnable task = w.**firstTask**; w.**firstTask** = **null**; w.unlock(); *// allow interrupts* **boolean** completedAbruptly = **true**; **try** {  **while** (task != **null** || (task = getTask()) != **null**) { **//获取任务 看看是否能拿到 点进入**  **Task中的队列poll take是真正拿到task**  **//workQueue**.poll(**keepAliveTime**, TimeUnit.***NANOSECONDS***) :  **超时取出**  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* **if** ((*runStateAtLeast*(**ctl**.get(), ***STOP***) ||  (Thread.*interrupted*() &&  *runStateAtLeast*(**ctl**.get(), ***STOP***))) &&  !wt.isInterrupted())  wt.interrupt();**// 确保线程是能中断的**  **try** {  beforeExecute(wt, task); **//开始任务前的钩子**  Throwable thrown = **null**;  **try** {  task.run();**//执行任务**  } **catch** (RuntimeException x) {  thrown = x; **throw** x;  } **catch** (Error x) {  thrown = x; **throw** x;  } **catch** (Throwable x) {  thrown = x; **throw new** Error(x);  } **finally** {  afterExecute(task, thrown); **//任务后的钩子**  }  } **finally** {  task = **null**;  w.**completedTasks**++;  w.unlock();  }  }  completedAbruptly = **false**; } **finally** {  processWorkerExit(w, completedAbruptly); **//看这儿** } |

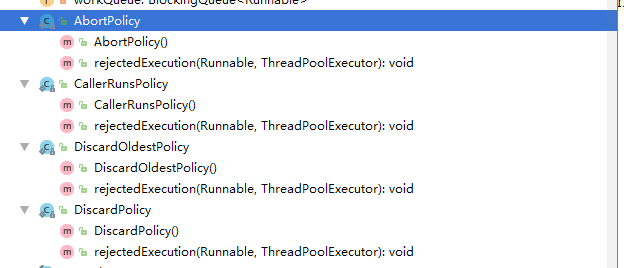
总结：获取任务、调用线程run方法

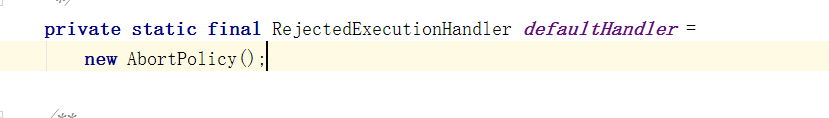
**processWorkerExit**

|  |
| --- |
| **if** (completedAbruptly) *// If abrupt, then workerCount wasn't adjusted* decrementWorkerCount();  **final** ReentrantLock mainLock = **this**.**mainLock**; mainLock.lock(); **try** {  **completedTaskCount** += w.**completedTasks**;  **workers**.remove(w);  **//移除work** } **finally** {  mainLock.unlock(); }  tryTerminate();  **int** c = **ctl**.get(); **if** (*runStateLessThan*(c, ***STOP***)) { **//判断是否还有任务**  **if** (!completedAbruptly) {  **int** min = **allowCoreThreadTimeOut** ? 0 : **corePoolSize**;  **if** (min == 0 && ! **workQueue**.isEmpty())  min = 1;  **if** (*workerCountOf*(c) >= min)  **return**; *// replacement not needed* }  addWorker(**null**, **false**); } |

### 线程池的拒绝策略

四种策略 默认是AbortPolicy





|  |  |
| --- | --- |
| 名称 | 意义 |
| AbortPolicy (默认) | 当任务添加到线程池中被拒绝时，它将抛出 RejectedExecutionException 异常。 |
| CallerRunsPolicy | 当任务添加到线程池中被拒绝时，会在线程池当前正在运行的Thread线程池中处理被拒绝的任务。 |
| DiscardOldestPolicy | 当任务添加到线程池中被拒绝时，线程池会放弃等待队列中最旧的未处理任务，然后将被拒绝的任务添加到等待队列中 |
| DiscardPolicy | 当任务添加到线程池中被拒绝时，线程池将丢弃被拒绝的任务。 |

# Java中提供的几种方法

## newFixedThreadPool

通过创建一个corePoolSize和maximumPoolSize相同的线程池。使用LinkedBlockingQuene作为阻塞队列，不过当线程池没有可执行任务时，也不会释放线程。

### newCachedThreadPool

1. 初始化一个可以缓存线程的线程池，默认缓存60s，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；  
   2、和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，  
   当线程的空闲时间超过keepAliveTime，会自动释放线程资源，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；

### 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。

### 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。除了newScheduledThreadPool的内部实现特殊一点之外，其它几个线程池都是基于ThreadPoolExecutor类实现的.