# 源码：

源代码解决什么问题？

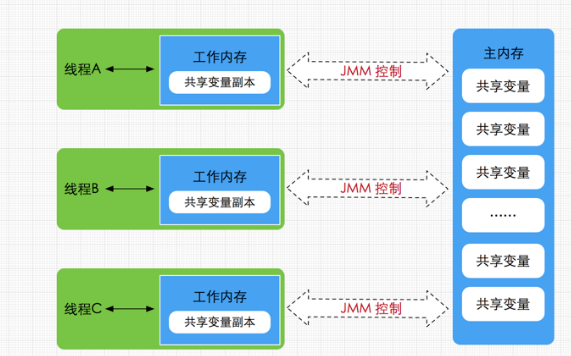
源代码通过什么原理实现

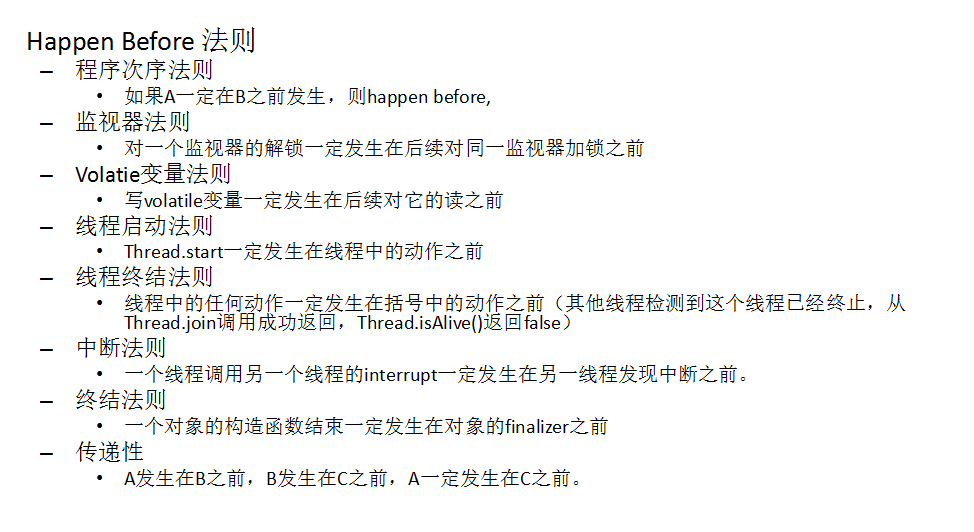
它采用了哪些接口、类、抽象类

为什么采用这些接口、类、抽象类

# JMM

java内存模型(即Java Memory Model，简称JMM)本身是一种抽象的概念，并不真实存在，它描述的是一组规则或规范，通过这组规范定义了程序中各个变量（包括实例字段，静态字段和构成数组对象的元素）的访问方式。由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(有些地方称为栈空间)，用于存储线程私有的数据，而Java内存模型中规定所有变量都存储在主内存，主内存是共享内存区域，所有线程都可以访问，但线程对变量的操作(读取赋值等)必须在工作内存中进行，首先要将变量从主内存拷贝的自己的工作内存空间，然后对变量进行操作，操作完成后再将变量写回主内存，不能直接操作主内存中的变量，工作内存中存储着主内存中的变量副本拷贝，前面说过，工作内存是每个线程的私有数据区域，因此不同的线程间无法访问对方的工作内存，线程间的通信(传值)必须通过主内存来完成，其简要访问过程如下图



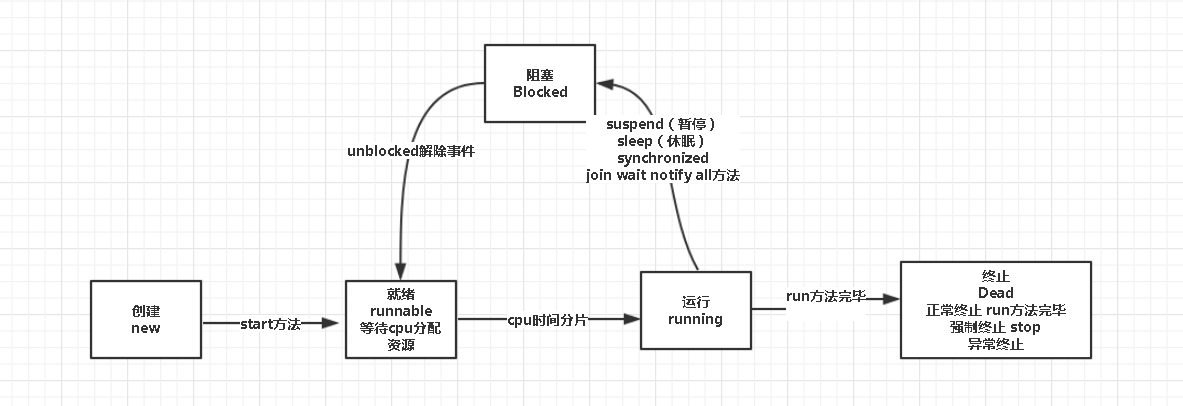


## 什么是线程？

线程是一个操作系统概念。操作系统负责这个线程的创建、挂起、运行、阻塞和终结操作。而操作系统创建线程、切换线程状态、终结线程都要进行CPU调度——这是一个耗费时间和系统资源的事情。

### 生命周期：

****Java当中，线程通常都有五种状态****，创建、就绪、运行、阻塞和死亡。  
　　第一是创建状态。在生成线程对象，并没有调用该对象的start方法，这是线程处于创建状态。  
　　第二是就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，但是此时线程调度程序还没有把该线程设置为当前线程，此时处于就绪状态。在线程运行之后，从等待或者睡眠中回来之后，也会处于就绪状态。  
　　第三是运行状态。线程调度程序将处于就绪状态的线程设置为当前线程，此时线程就进入了运行状态，开始运行run函数当中的代码。  
　　第四是阻塞状态。线程正在运行的时候，被暂停，通常是为了等待某个时间的发生(比如说某项资源就绪)之后再继续运行。sleep,suspend，wait等方法都可以导致线程阻塞。  
　　第五是死亡状态。如果一个线程的run方法执行结束或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪。



可以用过jstack 或者idea debug快照显示状态

"Low Memory Detector" 负责对可使用内存进行检测，如果发现可用内存低，分配新的内存空间。

"CompilerThread0" 用来调用JITing，实时编译装卸class。

"Signal Dispatcher" 负责分发内部事件。

"Finalizer" 负责调用Finalizer方法。

"Reference Handler" 负责处理引用。

"main" 是主线程。

"VM Thread"， "VM Periodic Task Thread"从名字上看是虚机内部线程。

### 状态描述：

NEW 状态是指线程刚创建, 尚未启动

RUNNABLE 状态是线程正在正常运行中, 当然可能会有某种耗时计算/IO等待的操作/CPU时间片切换等, 这个状态下发生的等待一般是其他系统资源, 而不是锁, Sleep等

BLOCKED  这个状态下, 是在多个线程有同步操作的场景, 比如正在等待另一个线程的synchronized 块的执行释放, 或者可重入的 synchronized块里别人调用wait() 方法, 也就是这里是线程在等待进入临界区

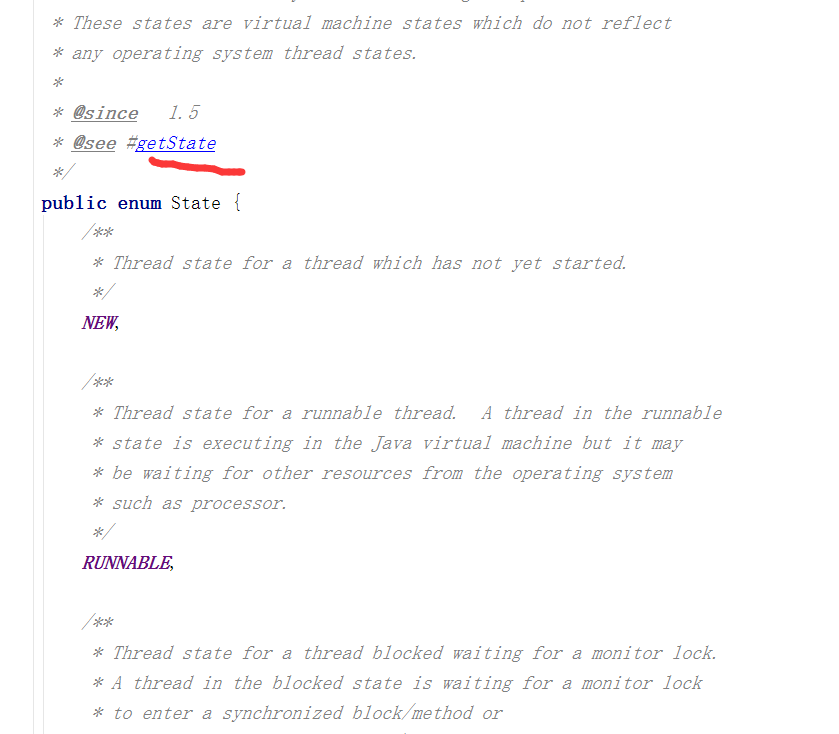
WAITING  这个状态下是指线程拥有了某个锁之后, 调用了他的wait方法, 等待其他线程/锁拥有者调用 notify / notifyAll 一遍该线程可以继续下一步操作, 这里要区分 BLOCKED 和 WATING 的区别, 一个是在临界点外面等待进入, 一个是在理解点里面wait等待别人notify, 线程调用了join方法 join了另外的线程的时候, 也会进入WAITING状态, 等待被他join的线程执行结束

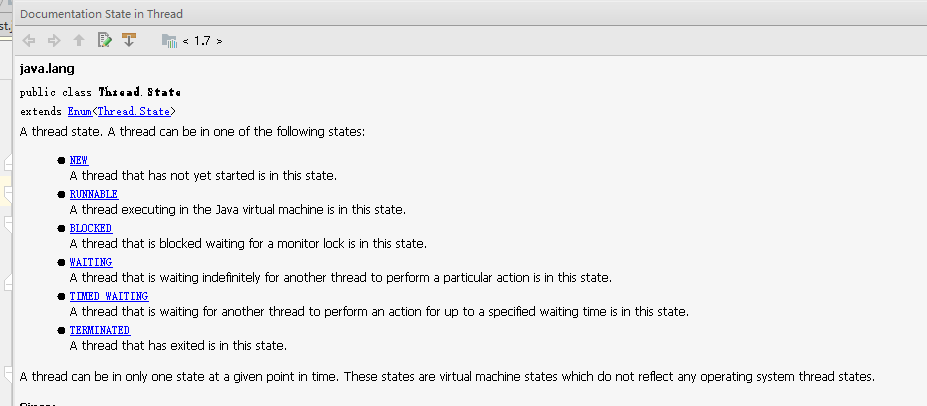
TIMED\_WAITING  这个状态就是有限的(时间限制)的WAITING, 一般出现在调用wait(long), join(long)等情况下, 另外一个线程sleep后, 也会进入TIMED\_WAITING状态

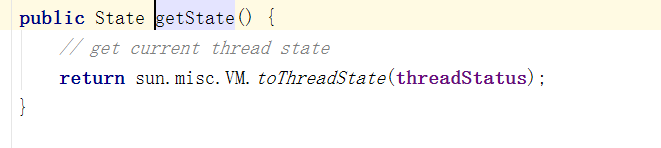
TERMINATED 这个状态下表示 该线程的run方法已经执行完毕了, 基本上就等于死亡了(当时如果线程被持久持有, 可能不会被回收)

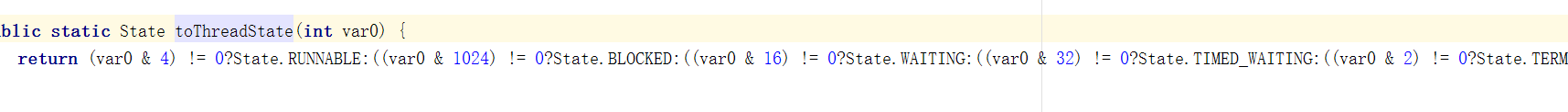
java.lang.Thread.State状态描述

视频中状态描述









### 优先级Priority：

线程的优先级是将该线程的重要性传给了调度器、cpu处理线程顺序有一定的不确定，但是调度器会倾向于优先权高的先执行

### 实现方式：

Thread、Runnable、Callable

Runnable和Thread区别：

除了单继承这个问题，还有一个问题可能大家没有感受到

Callable1.5引入，具有返回值，并且支持泛型，加入泛型是解决什么问题？并且不是run方法了是call方法哦

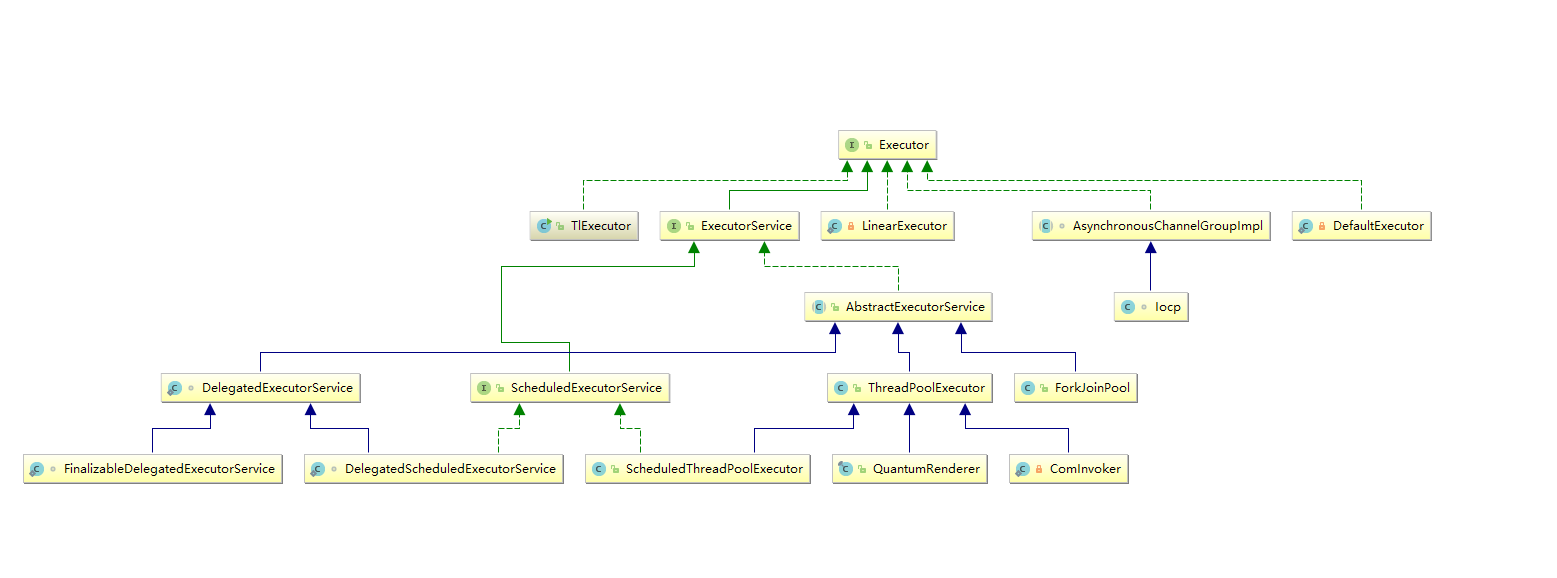
# 为什么有线程还要用线程池？

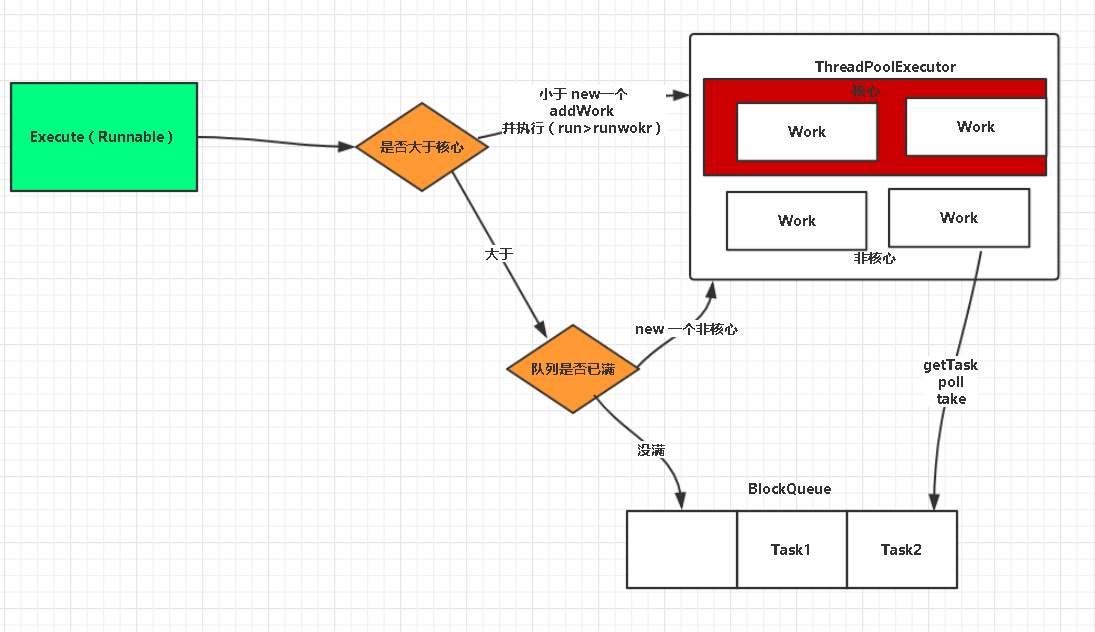
通过代码运行效率才提供线程池

# 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两种基础线程池的选择ScheduledThreadPoolExecutor和ThreadPoolExecutor





|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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** } |

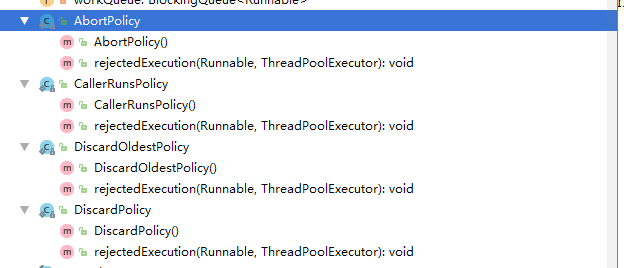
总结：获取任务、调用线程run方法

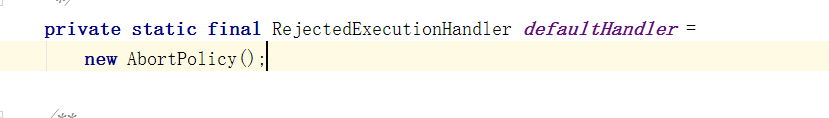
**processWorkerExit**

|  |
| --- |
| **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。 |

# Scheduledxx调度线程池原理

### 初始化ScheduledThreadPoolExecutor

调度核心构造器

|  |
| --- |
| **public** ScheduledThreadPoolExecutor(**int** corePoolSize) {  **super**(corePoolSize, Integer.***MAX\_VALUE***, 0, ***NANOSECONDS***,  **new** DelayedWorkQueue()); } |

### scheduleAtFixedRate方法

|  |
| --- |
| **public** ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  **long** initialDelay,  **long** period,  TimeUnit unit) {  **if** (command == **null** || unit == **null**)  **throw new** NullPointerException();  **if** (period <= 0)  **throw new** IllegalArgumentException();  ScheduledFutureTask<Void> sft =  **new** ScheduledFutureTask<Void>(command,  **null**,  triggerTime(initialDelay, unit),  unit.toNanos(period));**//处理时间**  RunnableScheduledFuture<Void> t = decorateTask(command, sft);   sft.**outerTask** = t;  delayedExecute(t)**;** **return** t; } |

### delayedExecute#ScheduledThreadPoolExecutor方法

|  |
| --- |
| **if** (isShutdown())  reject(task); **else** {  **super**.getQueue().add(task);**//增加任务 子类实现了**  **//总结：**add方法是通过DelayedWorkQueue（初始化时候指定的队列） 延迟队列实现 offer获取对象的延迟  **if** (isShutdown() &&  !canRunInCurrentRunState(task.isPeriodic()) && //判断是否已经停止  remove(task))  task.cancel(**false**);  **else** ensurePrestart(); } |

### Offer#DelayedWorkQueue 二叉树堆排序算法

|  |
| --- |
| **public boolean** offer(Runnable x) {  **if** (x == **null**)  **throw new** NullPointerException();  RunnableScheduledFuture e = (RunnableScheduledFuture)x; **//内部类**  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  **int** i = **size**;  **if** (i >= **queue**.**length**)**//判断是否扩容**  grow();  **size** = i + 1;  **if** (i == 0) {  **queue**[0] = e**;//这个队列是我们核心**  setIndex(e, 0); **//第一个直接设置索引和下标0**  } **else** {  siftUp(i, e);**//看这儿**   }  **if** (**queue**[0] == e) {  **leader** = **null**;  **available**.signal();**//唤醒** }  } **finally** {  lock.unlock();  }  **return true**; } |

### siftUp#DelayedWorkQueue保证相同的

|  |
| --- |
| **while** (k > 0) {  **int** parent = (k - 1) >>> 1;   RunnableScheduledFuture<?> e = **queue**[parent];  **if** (key.compareTo(e) >= 0)  **break**;   **queue**[k] = e;  setIndex(e, k);  k = parent; } **queue**[k] = key; setIndex(key, k); |

### compareTo#ScheduledFutureTask

|  |
| --- |
| **if** (other == **this**) *// compare zero if same object* **return** 0; **if** (other **instanceof** ScheduledFutureTask) {  ScheduledFutureTask<?> x = (ScheduledFutureTask<?>)other;  **long** diff = **time** - x.**time**; **//判断time**  **if** (diff < 0)  **return** -1;  **else if** (diff > 0)  **return** 1;  **else if** (**sequenceNumber** < x.**sequenceNumber**)  **return** -1;  **else  return** 1; } **long** diff = getDelay(***NANOSECONDS***) - other.getDelay(***NANOSECONDS***); **return** (diff < 0) ? -1 : (diff > 0) ? 1 : 0; |

我们在回到开始的地方,根据刚才我们跟代码可以看到执行时间的顺序已经分配好了，那如何确保work可以运行了？

确保有work执行

**ensurePrestart#ThreadPoolExecutor**

|  |
| --- |
| **int** wc = *workerCountOf*(**ctl**.get()); **if** (wc < **corePoolSize**)  addWorker(**null**, **true**); **else if** (wc == 0)  addWorker(**null**, **false**); |

放到队列 runwork take对象

## take#DelayedWorkQueue

//调用start>run>runWorker->getTask>take方法

|  |
| --- |
| **public** RunnableScheduledFuture take() **throws** InterruptedException {  **final** ReentrantLock lock = **this**.**lock**;  lock.lockInterruptibly();  **try** {  **for** (;;) {  RunnableScheduledFuture first = **queue**[0];  **if** (first == **null**) **//第一个有没有 没有等着**  **available**.await();  **else** {  **long** delay = first.getDelay(TimeUnit.***NANOSECONDS***);**//到时间了**  **if** (delay <= 0)//到时间了  **return** finishPoll(first);  **else if** (**leader** != **null**)  **available**.await();**//因为没有执行线程初始化，所以等等什么时候有了自己被他人唤醒**  **else** {  Thread thisThread = Thread.*currentThread*();  **leader** = thisThread;  **try** {  **available**.awaitNanos(delay);**//各种condition的awaitNanos 带时间的**  } **finally** {  **if** (**leader** == thisThread)  **leader** = **null**;  }  }  }  }  } **finally** {  **if** (**leader** == **null** && **queue**[0] != **null**)  **available**.signal();  lock.unlock();  } } |

### finishPoll#ScheduledThreadPoolExecutor

|  |
| --- |
| **private** RunnableScheduledFuture finishPoll(RunnableScheduledFuture f) {  **int** s = --**size**;  RunnableScheduledFuture x = **queue**[s];//重新排列  **queue**[s] = **null**;  **if** (s != 0)  siftDown(0, x);  setIndex(f, -1);  **return** f; } |

# 区别scheduleAtFixedRate 和scheduleWithFixedDelay 有什么区别吗？

（构造方法中实现了ScheduledFutureTask）

### run#ScheduledFutureTask

|  |
| --- |
| **public void** run() {  **boolean** periodic = isPeriodic(); **//判断是否是周期的**  **if** (!canRunInCurrentRunState(periodic))  cancel(**false**);  **else if** (!periodic)  ScheduledFutureTask.**super**.run();  **else if** (ScheduledFutureTask.**super**.runAndReset()) {  setNextRunTime()**;//点进去**  reExecutePeriodic(**outerTask**);**// 重置task 没有异常捕捉**  } } |

### setNextRunTime#ScheduledFutureTask

|  |
| --- |
| **long** p = **period**; **if** (p > 0)  **//看这个大于0 和小于0的区别**  **time** += p; //假如延迟了这个时间早过了，+当前时候肯定还是过的。 **else  time** = triggerTime(-p); //取的当前的任务延迟 |

# Future异步结果源码分析

awaitDone#FutureTask

|  |
| --- |
| **final long** deadline = timed ? System.*nanoTime*() + nanos : 0L; WaitNode q = **null**; **boolean** queued = **false**; **for** (;;) {  **if** (Thread.*interrupted*()) { //check线程中断  removeWaiter(q);  **throw new** InterruptedException();  }  **int** s = **state**;  **if** (s > ***COMPLETING***) { **//判断是否完成  if (q != null)**  q.**thread** = **null**;  **return** s;  }  **else if** (s == ***COMPLETING***) *// cannot time out yet* Thread.*yield*();  **else if** (q == **null**)  q = **new** WaitNode();  **//生成一个wait对象**  **else if** (!queued)  queued = ***UNSAFE***.compareAndSwapObject(**this**, ***waitersOffset***,  q.**next** = **waiters**, q)**;//链表的对象下一个置成当前的waitNode**  **else if** (timed) {  nanos = deadline - System.*nanoTime*();  **if** (nanos <= 0L) {  removeWaiter(q);  **return state**;  }  LockSupport.*parkNanos*(**this**, nanos); **//等待时间阻塞**  }  **else** LockSupport.*park*(**this**); **//一直阻塞** } |

### 拿到结果run#FutureTask什么时候回填的结果那？

|  |
| --- |
| **if** (**state** != ***NEW*** ||  !***UNSAFE***.compareAndSwapObject(**this**, ***runnerOffset***,  **null**, Thread.*currentThread*())) //如果状态不是new 或者 runner状态置不成功直接退出  **return**; **try** {  Callable<V> c = **callable**;  **if** (c != **null** && **state** == ***NEW***) {  V result;  **boolean** ran;  **try** {  result = c.call();//运行ok 的时候返回result  ran = **true**;  } **catch** (Throwable ex) {  result = **null**;  ran = **false**;  setException(ex);  }  **if** (ran)/**/正常成功set result对象**  set(result);  } } **finally** {  *// runner must be non-null until state is settled to  // prevent concurrent calls to run()* **runner** = **null**;  *// state must be re-read after nulling runner to prevent  // leaked interrupts* **int** s = **state**;  **if** (s >= ***INTERRUPTING***)  handlePossibleCancellationInterrupt(s); } |

### 取消生效cancel#FutureTask什么时候取消生效那？

|  |
| --- |
| **if** (!(**state** == ***NEW*** &&  ***UNSAFE***.compareAndSwapInt(**this**, ***stateOffset***, ***NEW***,  mayInterruptIfRunning ? ***INTERRUPTING*** : ***CANCELLED***))) **//CAS 置成stateOffset 的中断或者取消**  **return false**; **try** { *// in case call to interrupt throws exception* **if** (mayInterruptIfRunning) {  **//如果线程运行中，可能中断**  **try** {  Thread t = **runner**;  **if** (t != **null**)  t.interrupt();  } **finally** { *// final state* ***UNSAFE***.putOrderedInt(**this**, ***stateOffset***, ***INTERRUPTED***);  }  } } **finally** {  finishCompletion(); } **return true**; |