高并发技术之Java线程池源码解析

目录

[题纲 1](#_Toc494055212)

[问题 2](#_Toc494055213)

[线程池框架 2](#_Toc494055214)

[第一层结构 2](#_Toc494055215)

[接口简介 3](#_Toc494055216)

[核心实现类 3](#_Toc494055217)

[辅助类 4](#_Toc494055218)

[完成服务 4](#_Toc494055219)

[源码原理解析 4](#_Toc494055220)

[线程池执行原理 4](#_Toc494055221)

[调度线程池原理 7](#_Toc494055222)

[异步结果源码分析 10](#_Toc494055223)

# 题纲

1.线程池的模块结构

2.示例&原理解析

# 问题

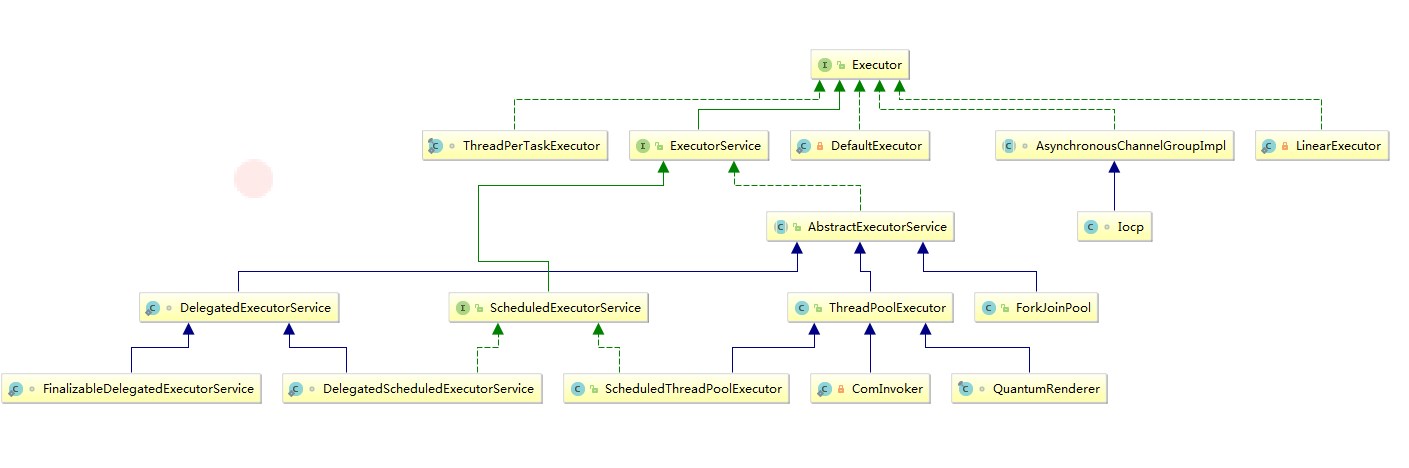
1.线程池包含哪些东西？

2.线程池的运作原理？

3.调度线程池的运作原理？

4.调度线程池怎么实现FixRate，FixDelay？和他们之间的区别？

5.怎么取消的？

线程池框架

## 第一层结构

sun.nio.ch.AsynchronousChannelGroupImpl(Iocp) 异步channel –AIO相关实现

java.util.concurrent.CompletableFuture.ThreadPerTaskExecutor （启动一个线程执行）

sun.net.httpserver.ServerImpl.DefaultExecutor （more执行器，直接执行）

com.sun.jmx.remote.internal.ClientNotifForwarder.LinearExecutor （线性执行器）

java.util.concurrent.ExecutorService （核心执行器服务）

## 接口简介

java.util.concurrent.Executor （执行器，执行方法）

java.util.concurrent.ExecutorService （执行服务） 包含服务的生命周期

java.util.concurrent.ScheduledExecutorService （调度相关的服务）

## 核心实现类

java.util.concurrent.ThreadPoolExecutor （普通的的线程池实现类）

java.util.concurrent.ScheduledThreadPoolExecutor （调度的核心实现类）

## 辅助类

java.util.concurrent.Executors

## 完成服务

java.util.concurrent.CompletionService

java.util.concurrent.ExecutorCompletionService

# 源码原理解析

## 线程池执行原理

构造器

核心数量，任务队列容器，存活时间，线程工厂，处理器。

|  |
| --- |
| *//固定线程池* ExecutorService executorService = Executors.*newFixedThreadPool*(2); executorService.execute(**new** Runnable() {  **public void** run() {   } });//runnable接口 executorService.submit(**new** Callable<String>() {  **public** String call() **throws** Exception {  **return "abc"**;  } });//callable接口 |

初始化构造器

|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,  **int** maximumPoolSize,  **long** keepAliveTime,  TimeUnit unit,  BlockingQueue<Runnable> workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) {  **if** (corePoolSize < 0 ||  maximumPoolSize <= 0 ||  maximumPoolSize < corePoolSize ||  keepAliveTime < 0)  **throw new** IllegalArgumentException();  **if** (workQueue == **null** || threadFactory == **null** || handler == **null**)  **throw new** NullPointerException();  **this**.**acc** = System.*getSecurityManager*() == **null** ?  **null** :  AccessController.*getContext*();  **this**.**corePoolSize** = corePoolSize;  **this**.**maximumPoolSize** = maximumPoolSize;  **this**.**workQueue** = workQueue;  **this**.**keepAliveTime** = unit.toNanos(keepAliveTime);  **this**.**threadFactory** = threadFactory;  **this**.**handler** = handler; } |

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); |

java.util.concurrent.ThreadPoolExecutor#addWorker

|  |
| --- |
| 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.如果非运行状态 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(); *// Re-read ctl 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; |

|  |
| --- |
| 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  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); } |

java.util.concurrent.ThreadPoolExecutor#runWorker

java.util.concurrent.ThreadPoolExecutor#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**); } |

## 调度线程池原理

调度核心构造器

DelayedWorkQueue延迟队列

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

java.util.concurrent.ScheduledThreadPoolExecutor#delayedExecute

|  |
| --- |
| **if** (isShutdown())  reject(task); **else** {  **super**.getQueue().add(task);//增加任务  **if** (isShutdown() &&  !canRunInCurrentRunState(task.isPeriodic()) &&  remove(task))  task.cancel(**false**);  **else** ensurePrestart(); } |

通过DelayedWorkQueue 延迟队列实现 offer获取对象的延迟

java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue#offer(java.lang.Runnable)

|  |
| --- |
| **if** (x == **null**)  **throw new** NullPointerException(); RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x; //当前对象 **final** ReentrantLock lock = **this**.**lock**; lock.lock(); **try** {  **int** i = **size**;  **if** (i >= **queue**.**length**) //扩容  grow();  **size** = i + 1;  **if** (i == 0) {  **queue**[0] = e;  setIndex(e, 0); //第一个直接设置索引和下标0  } **else** {  siftUp(i, e); //筛选到上边  }  **if** (**queue**[0] == e) {  **leader** = **null**;  **available**.signal(); //唤醒所有的被挤压的wait线程  } } **finally** {  lock.unlock(); } **return true**; |

java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue#siftUp （二叉堆算法）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); |

java.util.concurrent.ScheduledThreadPoolExecutor.ScheduledFutureTask#compareTo

|  |
| --- |
| **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; |

确保有work执行

java.util.concurrent.ThreadPoolExecutor#ensurePrestart

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

work运行的时候调用queue的take方法

java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue#take

|  |
| --- |
| **final** ReentrantLock lock = **this**.**lock**; lock.lockInterruptibly(); **try** {  **for** (;;) {  RunnableScheduledFuture<?> first = **queue**[0];//获取第一个对象  **if** (first == **null**)  **available**.await();  **else** {  **long** delay = first.getDelay(***NANOSECONDS***);//延迟时间  **if** (delay <= 0)//到时间了  **return** finishPoll(first);  first = **null**; *// don't retain ref while waiting* **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  } **finally** {  **if** (**leader** == thisThread)  **leader** = **null**;  }  }  }  } } **finally** {  **if** (**leader** == **null** && **queue**[0] != **null**)  **available**.signal();  lock.unlock(); } |

java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue#finishPoll

|  |
| --- |
| **int** s = --**size**; RunnableScheduledFuture<?> x = **queue**[s]; //重排序队列 **queue**[s] = **null**; **if** (s != 0)  siftDown(0, x); setIndex(f, -1); **return** f; |

怎么实现固定率的？

java.util.concurrent.ScheduledThreadPoolExecutor.ScheduledFutureTask#run

|  |
| --- |
| **boolean** periodic = isPeriodic(); **if** (!canRunInCurrentRunState(periodic))  cancel(**false**); **else if** (!periodic)  ScheduledFutureTask.**super**.run(); **else if** (ScheduledFutureTask.**super**.runAndReset()) {//有period的要执行成功设置下次执行时间和增加额外任务  setNextRunTime();  reExecutePeriodic(**outerTask**); } |

scheduleAtFixedRate 和*scheduleWithFixedDelay* 有什么区别吗？

java.util.concurrent.ScheduledThreadPoolExecutor.ScheduledFutureTask#setNextRunTime

|  |
| --- |
| **long** p = **period**; **if** (p > 0)  **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 |

# 异步结果源码分析

怎么拿到的异步任务结果？

java.util.concurrent.FutureTask#awaitDone

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

什么时候回填的结果那？

java.util.concurrent.FutureTask#run

|  |
| --- |
| **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); } |

什么时候取消生效那？

java.util.concurrent.FutureTask#cancel

|  |
| --- |
| **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**; |

图灵学院（tuling.ke.qq.com）出品