# 定时任务ScheduledThreadPoolExecutor：

## 介绍：

之前我们讲的ThreadPoolExecutor是java的普通线程池。而ScheduledThreadPoolExecutor是java提供的定时任务线程池。

## 使用：

常用

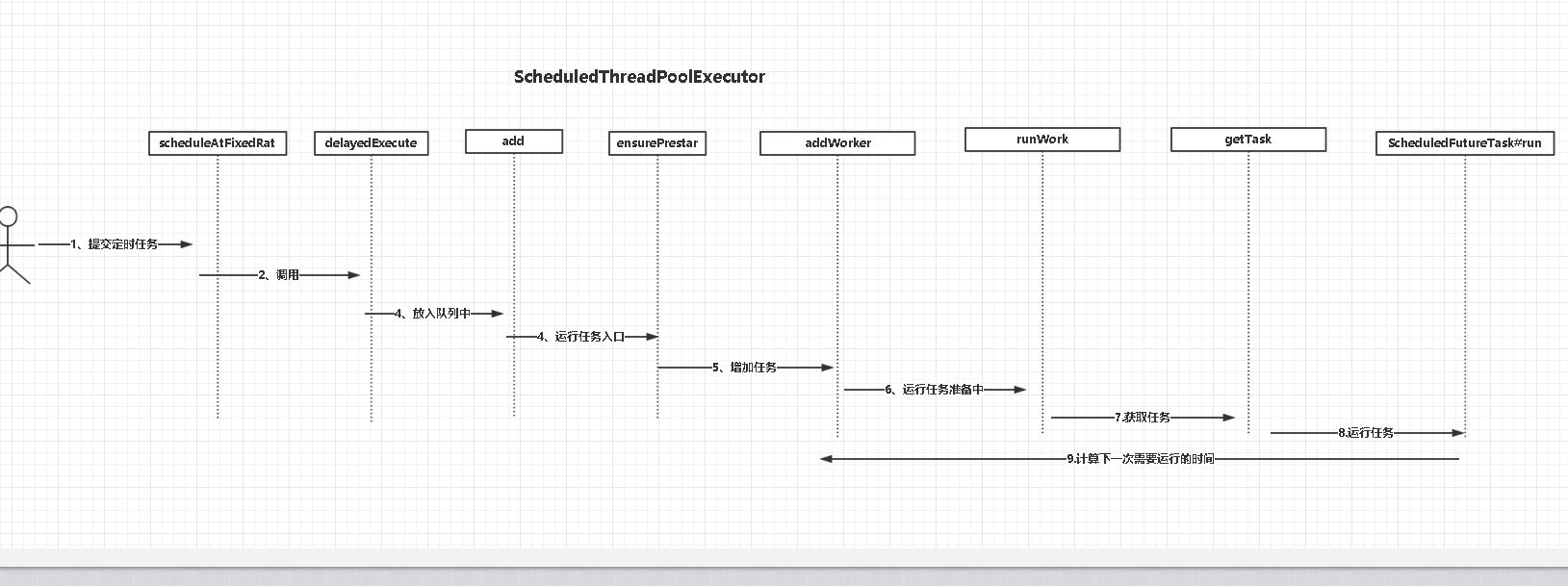
java.util.concurrent.ScheduledThreadPoolExecutor#schedule 定时任务

java.util.concurrent.ScheduledThreadPoolExecutor#scheduleAtFixedRate **固定**速率**连续**执行

java.util.concurrent.ScheduledThreadPoolExecutor#scheduleWithFixedDelay**非**固定速率**连续**执行

java.util.concurrent.ScheduledThreadPoolExecutor.DelayedWorkQueue延迟队列

## 源码分析：



### 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 |

参数

task--这是被调度的任务。

delay--这是以毫秒为单位的延迟之前的任务执行。

period--这是在连续执行任务之间的毫秒的时间。

## 实战：

实战与工作中注意事项

有异常一定要捕获，要不job不会执行了

# 单列模式（Double check lock）

双重锁定

发布与逸出这块以单列模式举例，在**高并发下**如何实现一个线程安全的单列模式。

会涉及到这些知识点1、懒汉模式、2、饿汉模式、3、synchronized 4、volatile 5、枚举、6重排序等

内部类

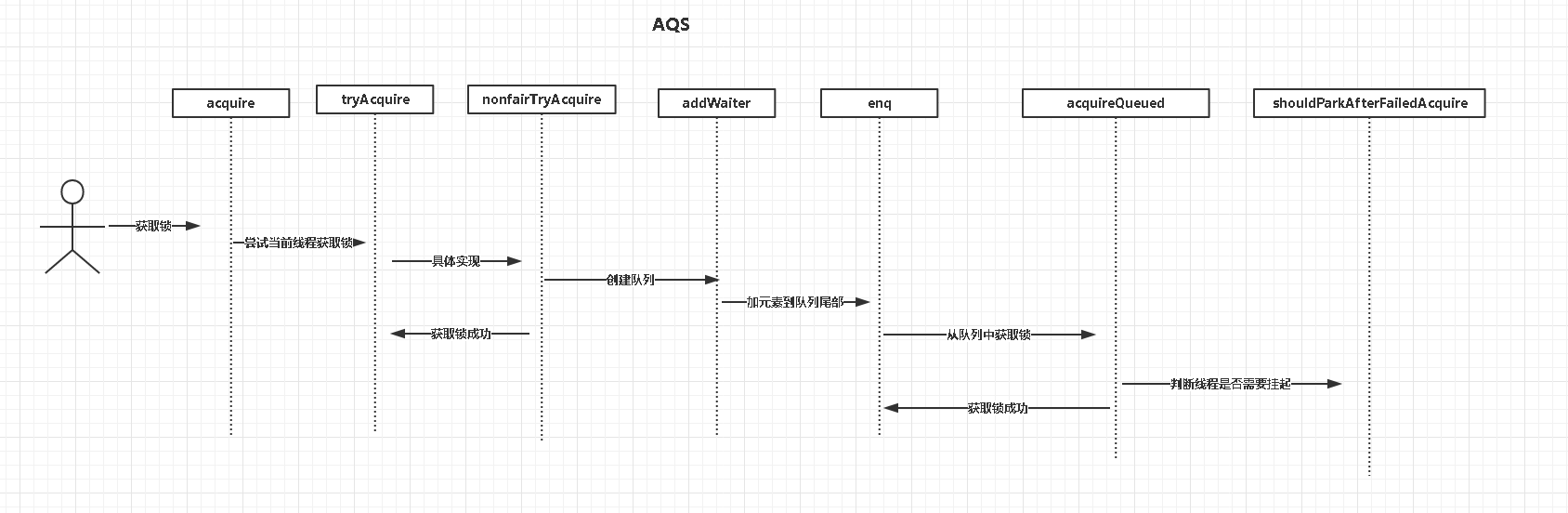
枚举类

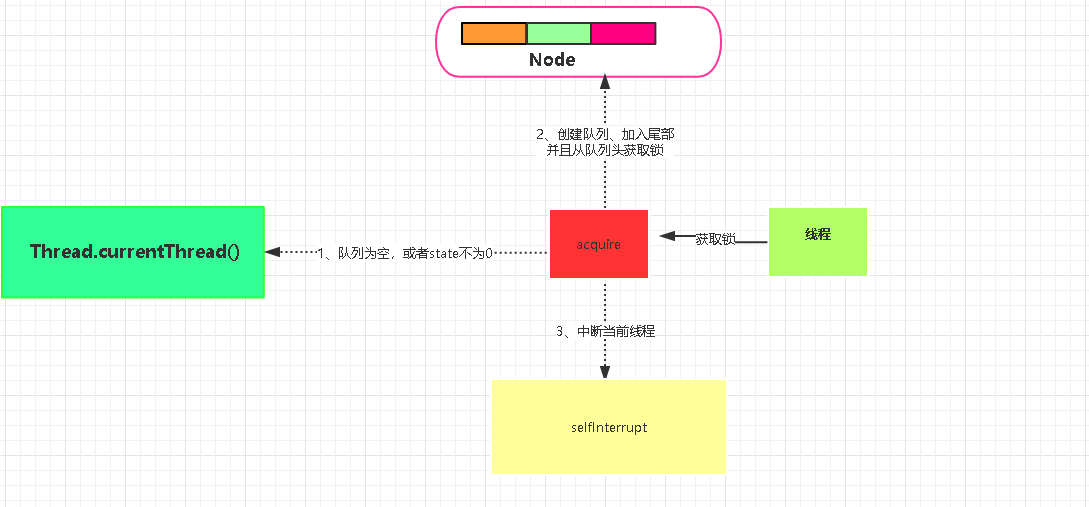
# AbstractQueuedSynchronizer(AQS)同步器:

公平锁、非公平锁

NonfairSync

cpu 随机调用task





### Java并发脑图:

https://www.processon.com/view/link/5b71947ce4b0be50eadcdad0