# 课程思考：

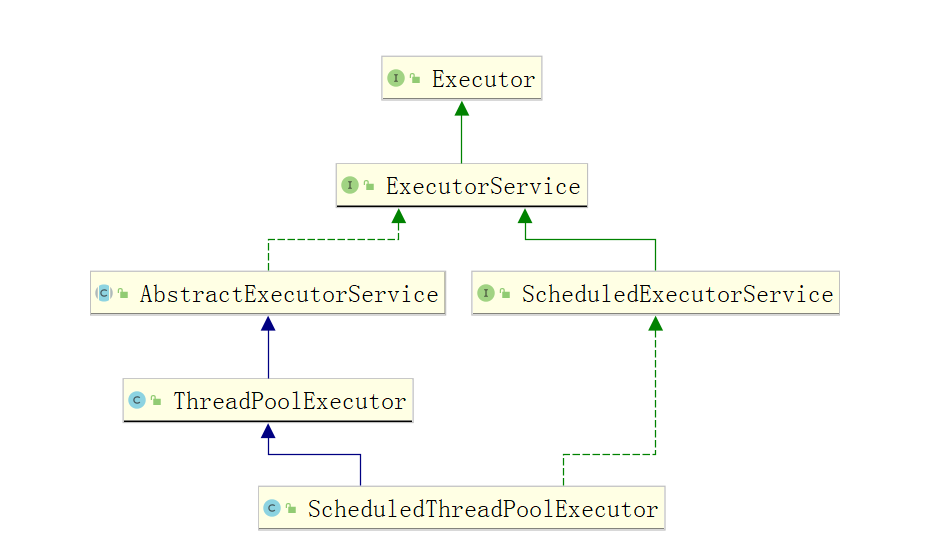
**线程池的好处和不足在哪儿？**

**线程池底层实现原理是？**

**日常开发中如何用好线程池？**

# 队列：

队列是先进先出的数据结构，就是先进入队列的数据，先被获取。但是有一种特殊的队列叫做优先级队列，它会对插入的数据进行优先级排序，保证优先级越高的数据首先被获取，与数据的插入顺序无关。

**今天跟我一起来学习线程池的原理和源码，这儿先给大家提供点预习资料，具体细节我们在今天晚上课程聊，有时间一定要来听课哦，保证让你干货满满。**

|  |
| --- |
| **public** ThreadPoolExecutor(**int** corePoolSize,核心线程数  **int** maximumPoolSize, 非核心线程数  **long** keepAliveTime,时间  TimeUnit unit,时间单位  BlockingQueue<Runnable> workQueue,队列  ThreadFactory threadFactory,线程工厂  RejectedExecutionHandler handler拒绝策略) { |

**含义：**

**1.corePoolSize -> 该线程池中核心线程数最大值**

核心线程：在创建完线程池之后，核心线程先不创建，在接到任务之后创建核心线程。并且会一直存在于线程池中（即使这个线程啥都不干），有任务要执行时，如果核心线程没有被占用，会优先用核心线程执行任务。数量一般情况下设置为CPU核数的二倍即可。

**2.maximumPoolSize -> 该线程池中线程总数最大值**

线程总数=核心线程数+非核心线程数

非核心线程：简单理解，即核心线程都被占用，但还有任务要做，就创建非核心线程

**3.keepAliveTime -> 非核心线程闲置超时时长**

这个参数可以理解为，任务少，但池中线程多，非核心线程不能白养着，超过这个时间不工作的就会被干掉，但是核心线程会保留。

**4.TimeUnit -> keepAliveTime的单位**

TimeUnit是一个枚举类型，其包括:  
NANOSECONDS ： 1微毫秒 = 1微秒 / 1000  
MICROSECONDS ： 1微秒 = 1毫秒 / 1000  
MILLISECONDS ： 1毫秒 = 1秒 /1000  
SECONDS ： 秒  
MINUTES ： 分  
HOURS ： 小时  
DAYS ： 天

**5.BlockingQueue workQueue -> 线程池中的任务队列**

默认情况下，任务进来之后先分配给核心线程执行，核心线程如果都被占用，并不会立刻开启非核心线程执行任务，而是将任务插入任务队列等待执行，核心线程会从任务队列取任务来执行，任务队列可以设置最大值，一旦插入的任务足够多，达到最大值，才会创建非核心线程执行任务。

**workQueue有四种：**

1.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

2.LinkedBlockingQueue：这个队列接收到任务的时候，如果当前已经创建的核心线程数小于线程池的核心线程数上限，则新建线程(核心线程)处理任务；如果当前已经创建的核心线程数等于核心线程数上限，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

3.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，或是执行实现定义好的饱和策略

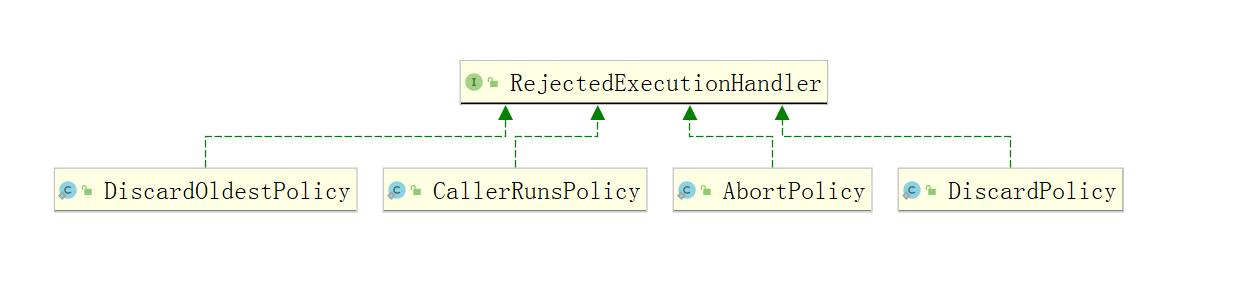
4.DelayQueue：队列内元素必须实现Delayed接口，这就意味着你传进去的任务必须先实现Delayed接口。这个队列接收到任务时，首先先入队，只有达到了指定的延时时间，才会执行任务

**6.ThreadFactory threadFactory -> 创建线程的工厂**

可以用线程工厂给每个创建出来的线程设置名字。一般情况下无须设置该参数。

**7.RejectedExecutionHandler handler -> 饱和策略**

这是当任务队列和线程池都满了时所采取的应对策略，默认是AbordPolicy， 表示无法处理新任务，并抛出 RejectedExecutionException 异常。此外还有3种策略，它们分别如下。  
（1）CallerRunsPolicy：用调用者所在的线程来处理任务。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度。  
（2）DiscardPolicy：不能执行的任务，并将该任务删除。  
（3）DiscardOldestPolicy：丢弃队列最近的任务，并执行当前的任务。

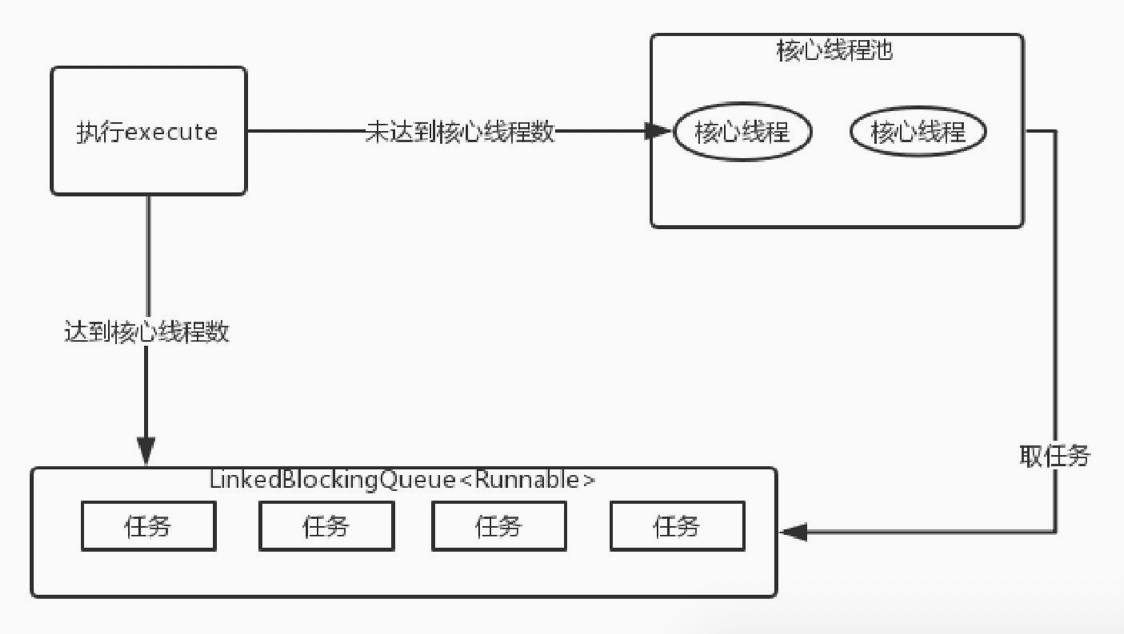


#### FixedThreadPool

可重用固定线程数的线程池，超出的线程会在队列中等待，在Executors类中我们可以找到创建方式：

|  |
| --- |
| public static ExecutorService newFixedThreadPool(int nThreads) {  return new ThreadPoolExecutor(nThreads, nThreads,  0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>());  } |

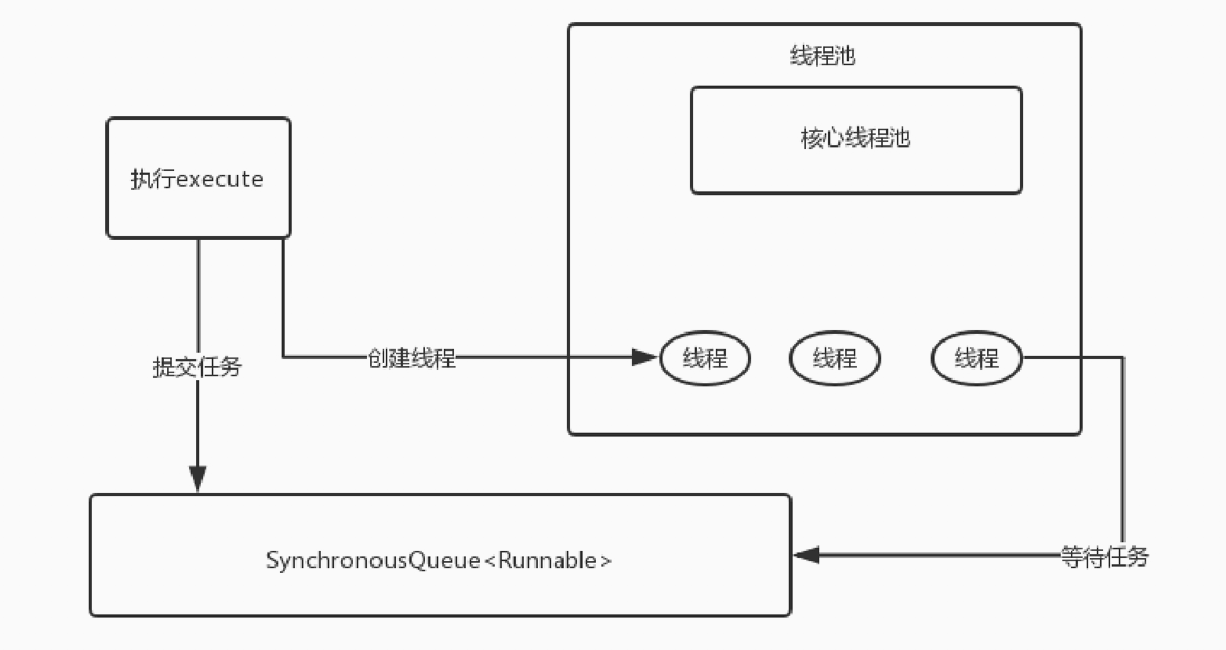
FixedThreadPool的corePoolSize和maximumPoolSize都设置为参数nThreads，也就是只有固定数量的核心线程，不存在非核心线程。keepAliveTime为0L表示多余的线程立刻终止，因为不会产生多余的线程，所以这个参数是无效的。FixedThreadPool的任务队列采用的是LinkedBlockingQueue。



CachedThreadPool是一个根据需要创建线程的线程池

|  |
| --- |
| public static ExecutorService newCachedThreadPool() {  return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,  60L, TimeUnit.SECONDS,  new SynchronousQueue<Runnable>());  } |

CachedThreadPool的corePoolSize是0，maximumPoolSize是Int的最大值，也就是说CachedThreadPool没有核心线程，全部都是非核心线程，并且没有上限。keepAliveTime是60秒，就是说空闲线程等待新任务60秒，超时则销毁。此处用到的队列是阻塞队列SynchronousQueue,这个队列没有缓冲区，所以其中最多只能存在一个元素,有新的任务则阻塞等待。

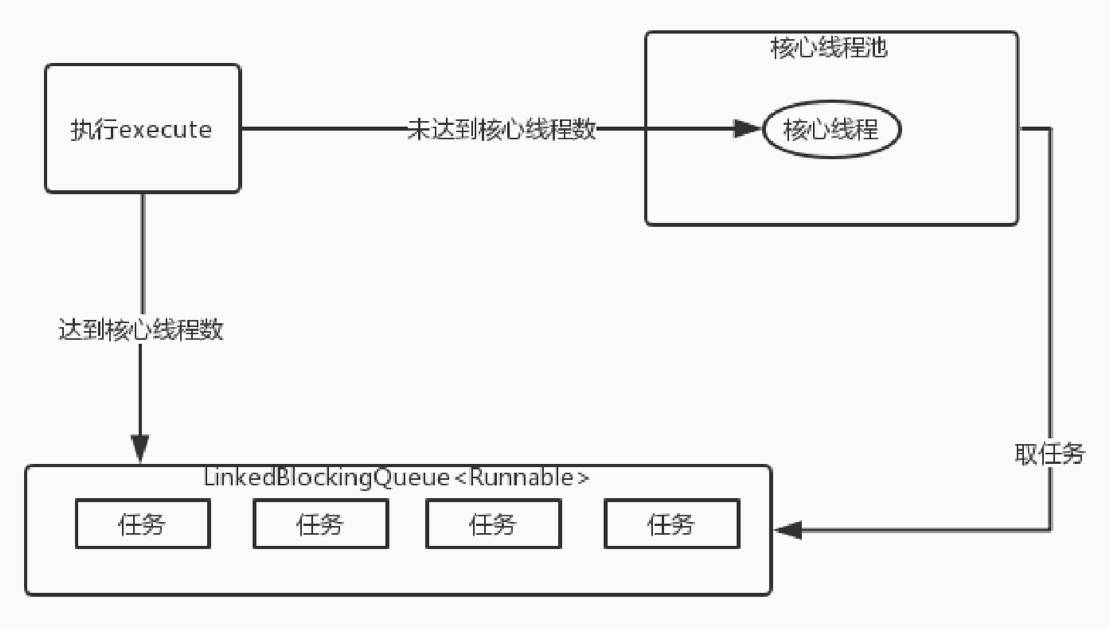


#### SingleThreadExecutor

SingleThreadExecutor是使用单个线程工作的线程池。其创建源码如下：

|  |
| --- |
| public static ExecutorService newSingleThreadExecutor() {  return new FinalizableDelegatedExecutorService  (new ThreadPoolExecutor(1, 1,  0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>()));  } |

我们可以看到总线程数和核心线程数都是1，所以就只有一个核心线程。该线程池才用链表阻塞队列LinkedBlockingQueue，先进先出原则，所以保证了任务的按顺序逐一进行。



### Execute方法

#### 

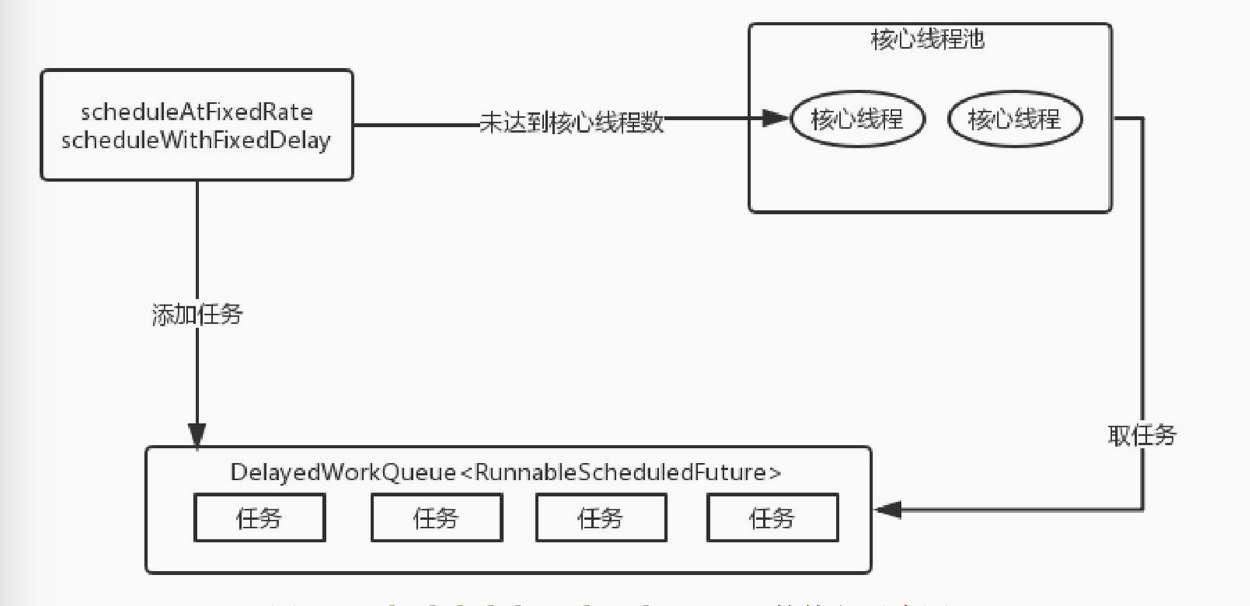
## ScheduledThreadPool

ScheduledThreadPool是一个能实现定时和周期性任务的线程池，它的创建源码如下：

这里创建了ScheduledThreadPoolExecutor，继承自ThreadPoolExecutor，主要用于定时延时或者定期处理任务。ScheduledThreadPoolExecutor的构造如下：

|  |
| --- |
| public ScheduledThreadPoolExecutor(int corePoolSize) {  super(corePoolSize, Integer.MAX\_VALUE,  DEFAULT\_KEEPALIVE\_MILLIS, MILLISECONDS,  new DelayedWorkQueue());  } |

可以看出corePoolSize是传进来的固定值，maximumPoolSize无限大，因为采用的队列DelayedWorkQueue是无解的，所以maximumPoolSize参数无效。该线程池执行如下：



当执行scheduleAtFixedRate或者scheduleWithFixedDelay方法时，会向DelayedWorkQueue添加一个实现RunnableScheduledFuture接口的ScheduledFutureTask(任务的包装类)，并会检查运行的线程是否达到corePoolSize。如果没有则新建线程并启动ScheduledFutureTask，然后去执行任务。如果运行的线程达到了corePoolSize时，则将任务添加到DelayedWorkQueue中。DelayedWorkQueue会将任务进行排序，先要执行的任务会放在队列的前面。在跟此前介绍的线程池不同的是，当执行完任务后，会将ScheduledFutureTask中的time变量改为下次要执行的时间并放回到DelayedWorkQueue中。

# DelayedWorkQueue成员属性

// 初始时，数组长度大小。

private static final int INITIAL\_CAPACITY = 16;

// 使用数组来储存队列中的元素。

private RunnableScheduledFuture<?>[] queue =

new RunnableScheduledFuture<?>[INITIAL\_CAPACITY];

// 使用lock来保证多线程并发安全问题。

private final ReentrantLock lock = new ReentrantLock();

// 队列中储存元素的大小

private int size = 0;

//特指队列头任务所在线程

private Thread leader = null;

// 当队列头的任务延时时间到了，或者有新的任务变成队列头时，用来唤醒等待线程

private final Condition available = lock.newCondition();

### 插入元素排序siftUp方法

private void siftUp(int k, RunnableScheduledFuture<?> key) {

// 当k==0时，就到了堆二叉树的根节点了，跳出循环

while (k > 0) {

// 父节点位置坐标, 相当于(k - 1) / 2

int parent = (k - 1) >>> 1;

// 获取父节点位置元素

RunnableScheduledFuture<?> e = queue[parent];

// 如果key元素大于父节点位置元素，满足条件，那么跳出循环

// 因为是从小到大排序的。

if (key.compareTo(e) >= 0)

break;

// 否则就将父节点元素存放到k位置

queue[k] = e;

// 这个只有当元素是ScheduledFutureTask对象实例才有用，用来快速取消任务。

setIndex(e, k);

// 重新赋值k，寻找元素key应该插入到堆二叉树的那个节点

k = parent;

}

// 循环结束，k就是元素key应该插入的节点位置

queue[k] = key;

setIndex(key, k);

}

### 移除元素排序siftDown方法

private void siftDown(int k, RunnableScheduledFuture<?> key) {

int half = size >>> 1;

// 通过循环，保证父节点的值不能小于子节点。

while (k < half) {

// 左子节点, 相当于 (k \* 2) + 1

int child = (k << 1) + 1;

// 左子节点位置元素

RunnableScheduledFuture<?> c = queue[child];

// 右子节点, 相当于 (k \* 2) + 2

int right = child + 1;

// 如果左子节点元素值大于右子节点元素值，那么右子节点才是较小值的子节点。

// 就要将c与child值重新赋值

if (right < size && c.compareTo(queue[right]) > 0)

c = queue[child = right];

// 如果父节点元素值小于较小的子节点元素值，那么就跳出循环

if (key.compareTo(c) <= 0)

break;

// 否则，父节点元素就要和子节点进行交换

queue[k] = c;

setIndex(c, k);

k = child;

}

queue[k] = key;

setIndex(key, k);

}

**Offer**

public boolean offer(Runnable x) {

if (x == null)

throw new NullPointerException();

RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x;

// 使用lock保证并发操作安全

final ReentrantLock lock = this.lock;

lock.lock();

try {

int i = size;

// 如果要超过数组长度，就要进行数组扩容

if (i >= queue.length)

// 数组扩容

grow();

// 将队列中元素个数加一

size = i + 1;

// 如果是第一个元素，那么就不需要排序，直接赋值就行了

if (i == 0) {

queue[0] = e;

setIndex(e, 0);

} else {

// 调用siftUp方法，使插入的元素变得有序。

siftUp(i, e);

}

// 表示新插入的元素是队列头，更换了队列头，

// 那么就要唤醒正在等待获取任务的线程。

if (queue[0] == e) {

leader = null;

// 唤醒正在等待等待获取任务的线程

available.signal();

}

} finally {

lock.unlock();

}

return true;

}

 元素个数超过数组长度，就会调用grow()方法，进行数组扩容。

 将新元素e添加到优先级队列中对应的位置，通过siftUp方法，保证按照元素的优先级排序。

 如果新插入的元素是队列头，即更换了队列头，那么就要唤醒正在等待获取任务的线程。这些线程可能是因为原队列头元素的延时时间没到，而等待的。

### Grow方法数组扩容

private void grow() {

int oldCapacity = queue.length;

// 每次扩容增加原来数组的一半数量。

int newCapacity = oldCapacity + (oldCapacity >> 1); // grow 50%

if (newCapacity < 0) // overflow

newCapacity = Integer.MAX\_VALUE;

// 使用Arrays.copyOf来复制一个新数组

queue = Arrays.copyOf(queue, newCapacity);

}

### Poll方法

public RunnableScheduledFuture<?> poll() {

final ReentrantLock lock = this.lock;

lock.lock();

try {

RunnableScheduledFuture<?> first = queue[0];

// 队列头任务是null，或者任务延时时间没有到，都返回null

if (first == null || first.getDelay(NANOSECONDS) > 0)

return null;

else

// 移除队列头元素

return finishPoll(first);

} finally {

lock.unlock();

}

}

当队列头任务是null，或者任务延时时间没有到，表示这个任务还不能返回，因此直接返回null。否则调用finishPoll方法，移除队列头元素并返回。

// 移除队列头元素

private RunnableScheduledFuture**<?**> finishPoll(RunnableScheduledFuture<**?>** f) {

// 将队列中元素个数减一

int s = --size;

// 获取队列末尾元素x

RunnableScheduledFuture**<?**> x = queue[s];

// 原队列末尾元素设置为null

queue[s] = null;

if (s != 0)

// 因为移除了队列头元素，所以进行重新排序。

siftDown(0, x);

setIndex(f, -1);

return f;

}

1. 先将队列中元素个数减一。
2. 将原队列末尾元素设置成队列头元素，再将队列末尾元素设置为null。
3. 调用siftDown(0, x)方法，保证按照元素的优先级排序。

### Take方法

public RunnableScheduledFuture<?> take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

for (;;) {

RunnableScheduledFuture<?> first = queue[0];

// 如果没有任务，就让线程在available条件下等待。

if (first == null)

available.await();

else {

// 获取任务的剩余延时时间

long delay = first.getDelay(NANOSECONDS);

// 如果延时时间到了，就返回这个任务，用来执行。

if (delay <= 0)

return finishPoll(first);

// 将first设置为null，当线程等待时，不持有first的引用

first = null; // don't retain ref while waiting

// 如果还是原来那个等待队列头任务的线程，

// 说明队列头任务的延时时间还没有到，继续等待。

if (leader != null)

available.await();

else {

// 记录一下当前等待队列头任务的线程

Thread thisThread = Thread.currentThread();

leader = thisThread;

try {

// 当任务的延时时间到了时，能够自动超时唤醒。

available.awaitNanos(delay);

} finally {

if (leader == thisThread)

leader = null;

}

}

}

}

} finally {

if (leader == null && queue[0] != null)

// 唤醒等待任务的线程

available.signal();

lock.unlock();

}

}