有道云链接：http://note.youdao.com/noteshare?id=38ce18aa0158e7998e6e508fd52e84cd&sub=10D2D06C84FE47F0A8EDBD82D8430204

Turing College 图灵课堂 | 致敬大师 致敬未来的你

Java并发线程池底层原理详解与源码分析

Monkey

- 开源框架Flasher作者
- 京东资深架构师
- 国美技术委员会成员

课程内容

1、线程与线程池性能对比，为何要用线程池

2、Java自带几种线程池详解

3、ThreadPoolExecutor 原理与源码详解

4、ScheduledThreadPoolExecutor原理与源码详解

腾讯课堂-图灵课堂

11月03日 晚上20:00

线程池与线程对比：

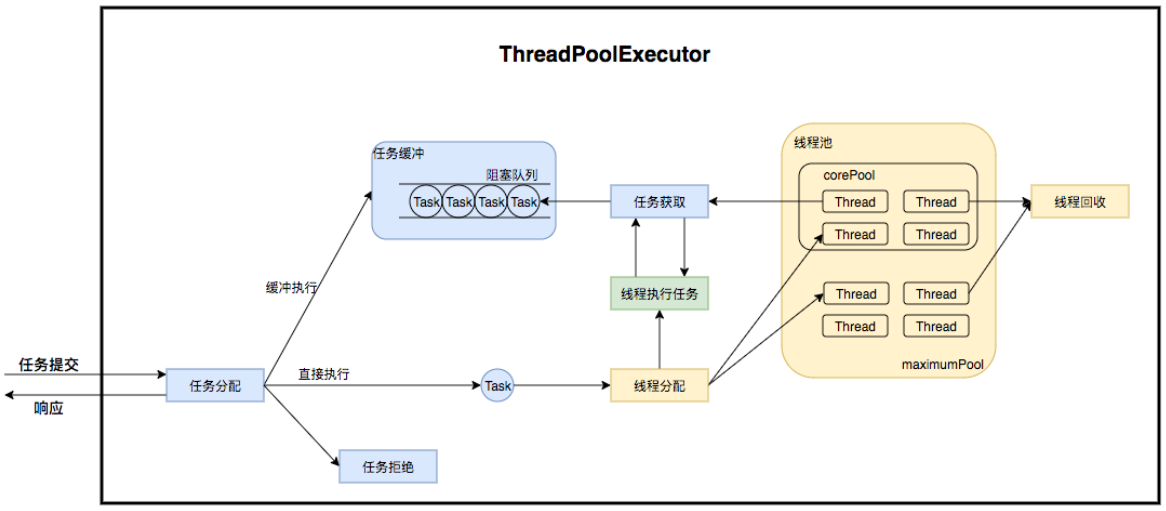
```
package bat.ke.qq.com.threadpool;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

/***
 * 使用线程的方式去执行程序
 */
public class ThreadTest {

 public static void main(String[] args) throws InterruptedException {
 Long start = System.currentTimeMillis();
 final Random random = new Random();
 final List<Integer> list = new ArrayList<Integer>();
 for (int i = 0; i < 100000; i++) {
 Thread thread = new Thread() {
 @Override
 public void run() {
 list.add(random.nextInt());

 }
 };
 thread.start();
 thread.join();
 }
 System.out.println("时间: " + (System.currentTimeMillis() - start));
 System.out.println("大小: " + list.size());

 }
}
```

线程池：

```
package bat.ke.qq.com.threadpool;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```java
import java.util.concurrent.TimeUnit;
/***
 * 线程池执行
 */
public class ThreadPoolTest {

 public static void main(String[] args) throws InterruptedException {
 Long start = System.currentTimeMillis();
 final Random random = new Random();
 final List<Integer> list = new ArrayList<Integer>();
 ExecutorService executorService = Executors.newSingleThreadExecutor();
 for (int i = 0; i < 100000; i++) {
 executorService.execute(new Runnable() {
 @Override
 public void run() {
 list.add(random.nextInt());
 }
 });
 }
 executorService.shutdown();
 executorService.awaitTermination(1, TimeUnit.DAYS);
 System.out.println("时间: "+(System.currentTimeMillis() - start));
 System.out.println("大小: "+list.size());


 }
}
```

原理解析：

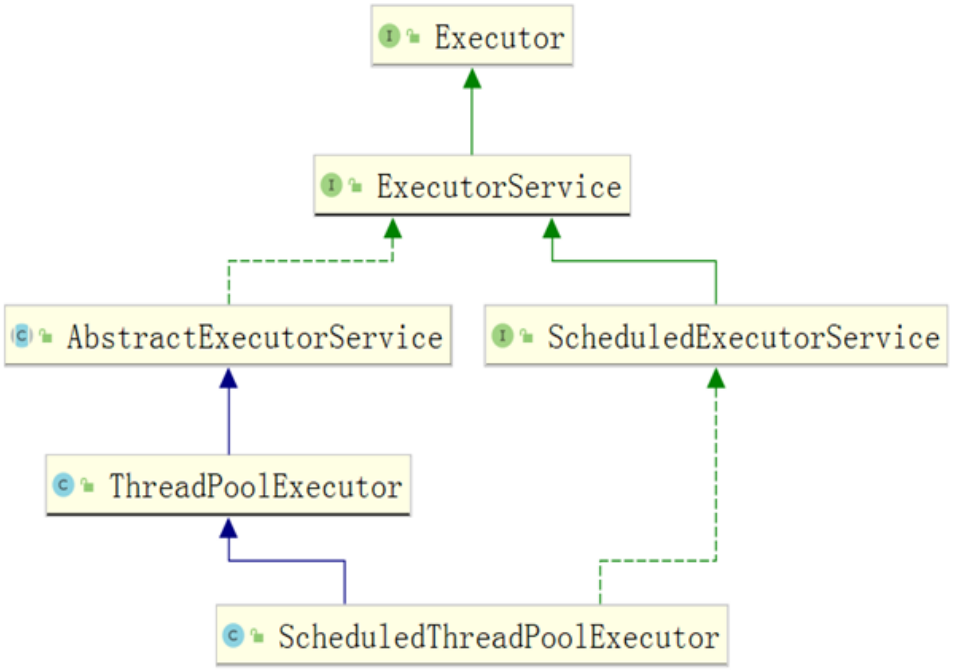


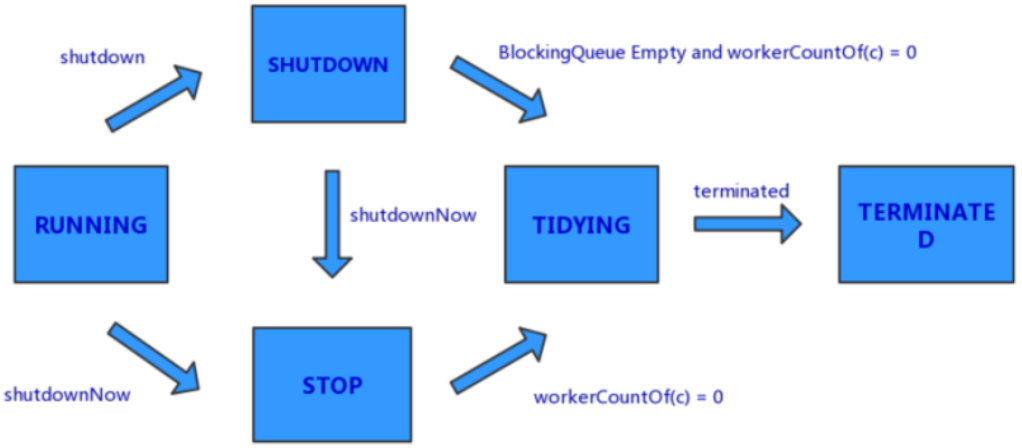
为什么阿里不推荐使用：

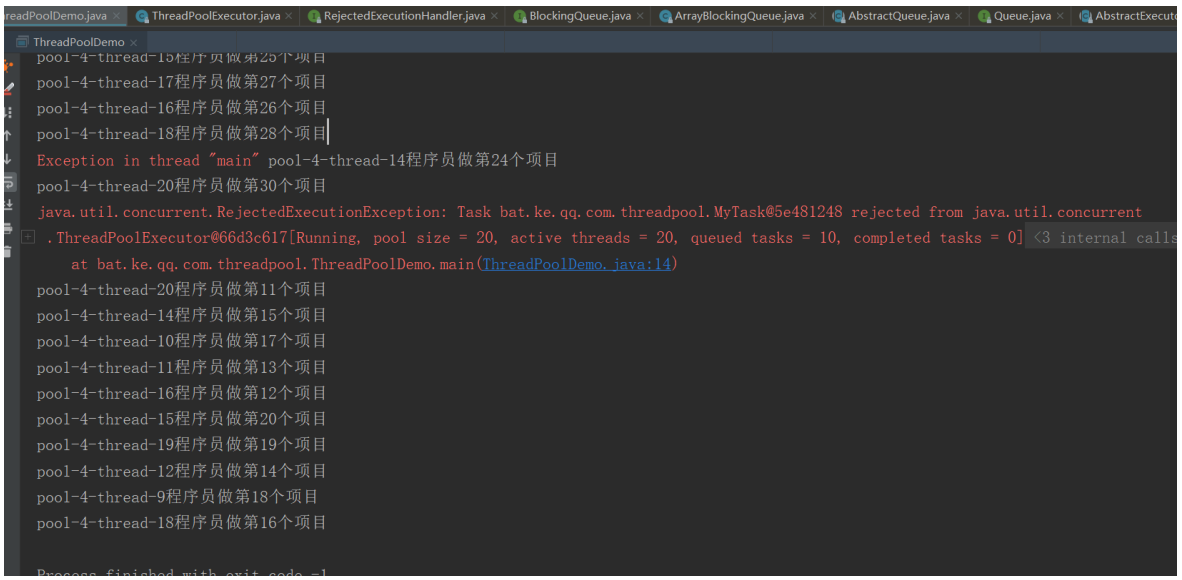
自定义线程池：

```
ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(10, 20,
 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>(10));//自定义线程
```

源码分析


Executor
ExecutorService
AbstractExecutorService
ScheduledExecutorService
ThreadPoolExecutor
ScheduledThreadPoolExecutor


shutdown
SHUTDOWN
BlockingQueue Empty and workerCountOf(c) = 0
RUNNING
shutdownNow
TIDYING
terminated
TERMINATED
shutdownNow
STOP
workerCountOf(c) = 0


ThreadPoolDemo.java
ThreadPoolExecutor.java
RejectedExecutionHandler.java
BlockingQueue.java
ArrayBlockingQueue.java
AbstractQueue.java
Queue.java
ThreadPoolDemo
pool-4-thread-17程序员做第27个项目
pool-4-thread-16程序员做第26个项目
pool-4-thread-18程序员做第28个项目
Exception in thread "main" pool-4-thread-14程序员做第24个项目
pool-4-thread-20程序员做第30个项目
java.util.concurrent.RejectedExecutionException: Task bat.ke.qq.com.threadpool.MyTask@5e481248 rejected from java.util.concurrent
.ThreadPoolExecutor@66d3c617[Running, pool size = 20, active threads = 20, queued tasks = 10, completed tasks = 0]
at bat.ke.qq.com.threadpool.ThreadPoolDemo.main(ThreadPoolDemo.java:14)
pool-4-thread-20程序员做第11个项目
pool-4-thread-14程序员做第15个项目
pool-4-thread-10程序员做第17个项目
pool-4-thread-11程序员做第13个项目
pool-4-thread-16程序员做第12个项目
pool-4-thread-15程序员做第20个项目
pool-4-thread-19程序员做第19个项目
pool-4-thread-12程序员做第14个项目
pool-4-thread-9程序员做第18个项目
pool-4-thread-18程序员做第16个项目

```
// runState is stored in the high-order bits
private static final int RUNNING = -1 << COUNT_BITS;
private static final int SHUTDOWN = 0 << COUNT_BITS;
private static final int STOP = 1 << COUNT_BITS;
private static final int TIDYING = 2 << COUNT_BITS;
private static final int TERMINATED = 3 << COUNT_BITS;
```

其中COUNT_BITS是 int 位数

private static final int COUNT_BITS = Integer.SIZE - 3; //Integer.SIZE=32

所以实际 COUNT_BITS = 29,

用上面的5个常量表示线程池的状态，实际上是使用32位中的高3位表示;

execute方法:

```
int c = ctl.get();
1、判断当前的线程数是否小于corePoolSize如果是，
使用入参任务通过addWord方法创建一个新的线程，
如果能完成新线程创建exexute方法结束，成功提交任务;
if (workerCountOf(c) < corePoolSize) {
 if (addWorker(command, true))
 return;
 c = ctl.get();
}
2、在第一步没有完成任务提交；状态为运行并且能成功加入任务到工作队列后，
再进行一次check，如果状态在任务加入队列后变为了非运行（有可能是在执行到这里线程池shutdown了）
非运行状态下当然是需要reject;
然后再判断当前线程数是否为0（有可能这个时候线程数变为了0），如是，新增一个线程;
if (isRunning(c) && workQueue.offer(command)) {
 int recheck = ctl.get();
 if (! isRunning(recheck) && remove(command))
 reject(command);
 else if (workerCountOf(recheck) == 0)
 addWorker(null, false); 判断当前工作线程池数是否为0
 如果是创建一个null任务，任务在堵塞队列存在了就会从队列中取出 这样做的意义是
 保证线程池在running状态必须有一个任务在执行


}
3、如果不能加入任务到工作队列，将尝试使用任务新增一个线程，如果失败，
则是线程池已经shutdown或者线程池已经达到饱和状态，所以reject;
从上面新增任务的execute方法也可以看出，拒绝策略不仅仅是在饱和状态下使用，
```

```
在线程池进入到关闭阶段同样需要使用到；
上面的几行代码还不能完全清楚这个新增任务的过程，
肯定还需要清楚addWorker方法才行：
else if (!addWorker(command, false))
 reject(command);

```

1、判断当前的线程数是否小于corePoolSize如果是，使用入参任务通过addWord方法创建一个新的线程，如果能完成新线程创建exexute方法结束，成功提交任务；

2、在第一步没有完成任务提交；状态为运行并且能成功加入任务到工作队列后，再进行一次check，如果状态在任务加入队列后变为了非运行（有可能是在执行到这里线程池shutdown了），非运行状态下当然是需要reject；然后再判断当前线程数是否为0（有可能这个时候线程数变为了0），如是，新增一个线程；

3、如果不能加入任务到工作队列，将尝试使用任务新增一个线程，如果失败，则是线程池已经shutdown或者线程池已经达到饱和状态，所以reject;
从上面新增任务的execute方法也可以看出，拒绝策略不仅仅是在饱和状态下使用，在线程池进入到关闭阶段同样需要使用到；

addWorker方法

```

private boolean addWorker(Runnable firstTask, boolean core) {
 retry: goto写法 用于重试
 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()))
 线程状态非运行并且非shutdown状态任务为空，队列非空就不能新增线程了

 return false;

 for (;;) {
 int wc = workerCountOf(c);
```

```
if (wc >= CAPACITY ||
wc >= (core ? corePoolSize : maximumPoolSize))
当前现场大于等于最大值
等于核心线程数 非核心大于等于线程池数 说明达到了阈值
最大线程数 就不新增线程
return false;
if (compareAndIncrementWorkerCount(c)) ctl+1 工作线程池数量+1 如果成功
就跳出死循环。
cas操作 如果为true 新增成功 退出
break retry;
c = ctl.get(); // Re-read ctl
if (runStateOf(c) != rs)
continue retry; 进来的状态和此时的状态发生改变 重头开始 重试
// else CAS failed due to workerCount change; retry inner loop
}
}
上面主要是对ctl工作现场+1

boolean workerStarted = false;
boolean workerAdded = false;
Worker w = null;
try {
w = new Worker(firstTask); 内部类 封装了线程和任务 通过threadfactory创建线程

final Thread t = w.thread; 每一个worker就是一个线程数
if (t != 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.
重新获取当前线程状态
int rs = runStateOf(ctl.get());
小于shutdown就是running状态
if (rs < SHUTDOWN ||
(rs == SHUTDOWN && firstTask == null)) {
SHUTDOWN 和firstTask 为空是从队列中处理任务 那就可以放到集合中
线程还没start 就是alive就直接异常
if (t.isAlive()) // precheck that t is startable
throw new IllegalThreadStateException();
workers.add(w);
int s = workers.size();
```

```
62  if (s > largestPoolSize)
63  largestPoolSize = s; 记录最大线程数
64  workerAdded = true;
65  }
66  } finally {
67  mainLock.unlock();
68  }
69  if (workerAdded) {
70  t.start(); 启动线程
71  workerStarted = true;
72  }
73  }
74  } finally {
75  if (! workerStarted)
76  addWorkerFailed(w);//失败回退 从wokers移除w 线程数减1 尝试结束线程池
77  }
78  return workerStarted;
79 }
```

```
1
2  private final class Worker
3   extends AbstractQueuedSynchronizer
4   implements Runnable
5  {
6   /**
7   * This class will never be serialized, but we provide a
8   * serialVersionUID to suppress a javac warning.
9   */
10   private static final long serialVersionUID = 6138294804551838833L;
11
12   /** Thread this worker is running in. Null if factory fails. */
13   正在运行woker线程
14   final Thread thread;
15   /** Initial task to run. Possibly null. */
16   传入的任务
17   Runnable firstTask;
18   /** Per-thread task counter */
19   完成的任务数 监控用
20   volatile long completedTasks;
21
22   /**
```

```
 * Creates with given first task and thread from ThreadFactory.
 * @param firstTask the first task (null if none)
 */
Worker(Runnable firstTask) {
禁止线程中断
setState(-1); // inhibit interrupts until runWorker
this.firstTask = firstTask;
this.thread = getThreadFactory().newThread(this);
}

/** Delegates main run loop to outer runWorker */
public void run() {
runWorker(this);
}
```

runwoker方法：

```
final void runWorker(Worker w) {
 Thread wt = Thread.currentThread();//获取当前线程
 Runnable task = w.firstTask;
 w.firstTask = null;
 w.unlock(); // allow interrupts 把state从-1改为0 意思是可以允许中断
 boolean completedAbruptly = true;
 try { task不为空 或者阻塞队列中拿到了任务
 while (task != null || (task = getTask()) != null) {
 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
  如果当前线程池状态等于stop 就中断
  //Thread.interrupted() 中断标志
  if ((runStateAtLeast(ctl.get(), STOP) ||
  (Thread.interrupted() &&
  runStateAtLeast(ctl.get(), STOP))) &&
  !wt.isInterrupted())
  wt.interrupt();
  try {
  beforeExecute(wt, task);
  Throwable thrown = null;
  try {
  task.run();
  } 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);
 }
 } finally {
 task = null; 这设置为空 等下次循环就会从队列里面获取
 w.completedTasks++; 完成任务数+1
 w.unlock();
 }
 }
 completedAbruptly = false;
 } finally {
 processWorkerExit(w, completedAbruptly);
 }
}
```

getTask方法：

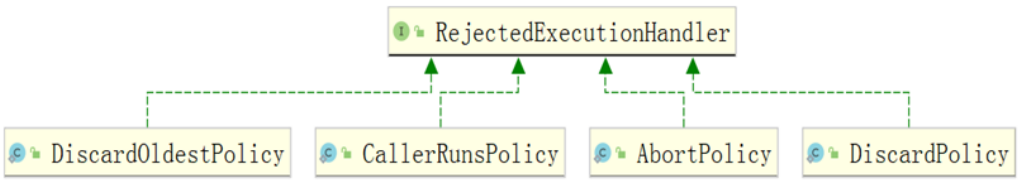
```

private Runnable getTask() {
 boolean timedOut = false; // Did the last poll() time out?

 for (;;) {
 int c = ctl.get();
 int rs = runStateOf(c);//获取线程池运行状态

 shuitdown或者weikong 那就工作现场-1 同事返回为null
 // Check if queue empty only if necessary.
 if (rs >= SHUTDOWN && (rs >= STOP || workQueue.isEmpty())) {
 decrementWorkerCount();
 return null;
 }
 重新获取工作线程数
 int wc = workerCountOf(c);
 timed是标志超时销毁
 allowCoreThreadTimeOut true 核心线程池也是可以销毁的
 // Are workers subject to culling?
 boolean timed = allowCoreThreadTimeOut || wc > corePoolSize;

```

```
 if ((wc > maximumPoolSize || (timed && timedOut))
 && (wc > 1 || workQueue.isEmpty())) {
 if (compareAndDecrementWorkerCount(c))
 return null;
 continue;
 }

 try {
 Runnable r = timed ?
 workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
 workQueue.take();
 if (r != null)
 return r;
 timedOut = true;
 } catch (InterruptedException retry) {
 timedOut = false;
 }
 }
}
```

processWorkerExit方法：

```
private void processWorkerExit(Worker w, boolean completedAbruptly) {
 if (completedAbruptly) // If abrupt, then workerCount wasn't adjusted
 decrementWorkerCount();

 final ReentrantLock mainLock = this.mainLock;
 mainLock.lock();
 try {
 completedTaskCount += w.completedTasks;
 workers.remove(w);
 } finally {
 mainLock.unlock();
 }

 tryTerminate();

 int c = ctl.get();
 if (runStateLessThan(c, STOP)) {
 if (!completedAbruptly) {
 int min = allowCoreThreadTimeOut ? 0 : corePoolSize;
 if (min == 0 && ! workQueue.isEmpty())
```

```
21   min = 1;
22   if (workerCountOf(c) >= min)
23   return; // replacement not needed
24   }
25   addWorker(null, false);
26   }
27 }
```



ThreadPoolExecutor内部有实现4个拒绝策略：

(1)、CallerRunsPolicy，由调用execute方法提交任务的线程来执行这个任务；

(2)、AbortPolicy，抛出异常RejectedExecutionException拒绝提交任务；

(3)、DiscardPolicy，直接抛弃任务，不做任何处理；

(4)、DiscardOldestPolicy，去除任务队列中的第一个任务（最旧的），重新提交；

ScheduledThreadPoolExecutor：

schedule：延迟多长时间之后只执行一次；

scheduledAtFixedRate固定：延迟指定时间后执行一次，之后按照固定的时长周期执行；

scheduledWithFixedDelay非固定：延迟指定时间后执行一次，之后按照：上一次任务执行时长 + 周期的时长 的时间去周期执行；

```
1 private void delayedExecute(RunnableScheduledFuture<?> task) {
2  //如果线程池不是RUNNING状态，则使用拒绝策略把提交任务拒绝掉
3  if (isShutdown())
4  reject(task);
5  else {
```

```
6   //与ThreadPoolExecutor不同，这里直接把任务加入延迟队列
7   super.getQueue().add(task);
8   //如果当前状态无法执行任务，则取消
9   if (isShutdown() &&
10   !canRunInCurrentRunState(task.isPeriodic()) &&
11   remove(task))
12   task.cancel(false);
13   else
14   //和ThreadPoolExecutor不一样，corePoolSize没有达到会增加Worker;
15   //增加Worker，确保提交的任务能够被执行
16   ensurePrestart();
17   }
18  }
```

offer方法：

```
1  public boolean offer(Runnable x) {
2   if (x == null)
3   throw new NullPointerException();
4   RunnableScheduledFuture<?> e = (RunnableScheduledFuture<?>)x;
5   final ReentrantLock lock = this.lock;
6   lock.lock();
7   try {
8   int i = size;
9   if (i >= queue.length)
10   // 容量扩增50%。
11   grow();
12   size = i + 1;
13   // 第一个元素,其实这里也可以统一进行sift-up操作,没必要特判。
14   if (i == 0) {
15   queue[0] = e;
16   setIndex(e, 0);
17   } else {
18   // 插入堆尾。
19   siftUp(i, e);
20   }
21   // 如果新加入的元素成为了堆顶,则原先的leader就无效了。
22   if (queue[0] == e) {
23   leader = null;
24   // 由于原先leader已经无效被设置为null了,这里随便唤醒一个线程(未必是原先的leader)来
取走堆顶任务。
25   available.signal();
```

```
 }
 } finally {
 lock.unlock();
 }
 return true;
}
```

siftup方法：

```
private void siftUp(int k, RunnableScheduledFuture<?> key) {
    // 找到父节点的索引
    while (k > 0) {
        // 获取父节点
        int parent = (k - 1) >>> 1;
        RunnableScheduledFuture<?> e = queue[parent];
        // 如果key节点的执行时间大于父节点的执行时间，不需要再排序了
        if (key.compareTo(e) >= 0)
            break;
         // 如果key.compareTo(e) < 0，
  说明key节点的执行时间小于父节点的执行时间，需要把父节点移到后面
         queue[k] = e;
         setIndex(e, k);
         // 设置索引为k
         k = parent;
    }
    // key设置为排序后的位置中
    queue[k] = key;
    setIndex(key, k);
}
```

任务执行：

```
public void run() {
 // 是否周期性，就是判断period是否为0。
 boolean periodic = isPeriodic();
 // 检查任务是否可以被执行。
 if (!canRunInCurrentRunState(periodic))
 cancel(false);
 // 如果非周期性任务直接调用run运行即可。
 else if (!periodic)
```

```
9   ScheduledFutureTask.super.run();
10   // 如果成功runAndRest，则设置下次运行时间并调用reExecutePeriodic。
11   else if (ScheduledFutureTask.super.runAndReset()) {
12   setNextRunTime();
13   // 需要重新将任务(outerTask)放到工作队列中。此方法源码会在后文介绍ScheduledThread
PoolExecutor本身API时提及。
14   reExecutePeriodic(outerTask);
15   }
16  }
```

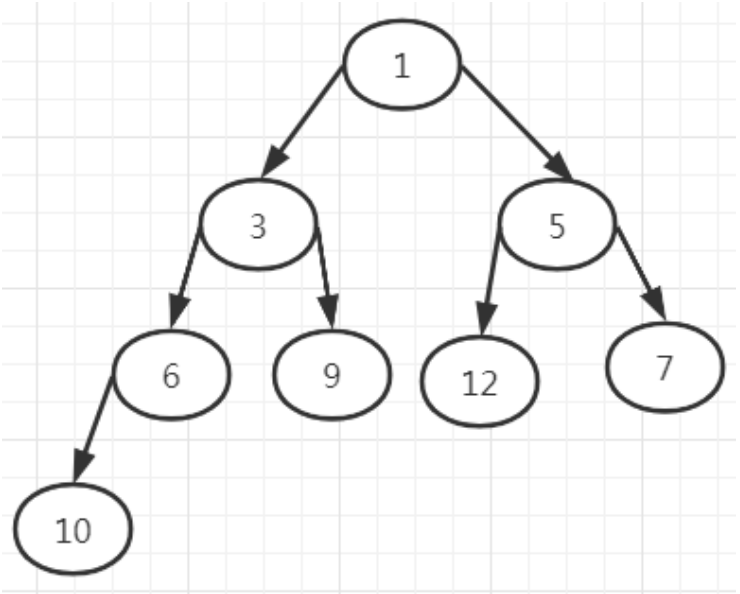
fied-rate模式和fixed-delay模式区别

```
1  private void setNextRunTime() {
2   long p = period;
3   /*
4   * fixed-rate模式，时间设置为上一次时间+p。
5   * 提一句，这里的时间其实只是可以被执行的最小时间，不代表到点就要执行。
6   * 如果这次任务还没执行完是肯定不会执行下一次的。
7   */
8   if (p > 0)
9   time += p;
10   /**
11   * fixed-delay模式，计算下一次任务可以被执行的时间。
12   * 简单来说差不多就是当前时间+delay值。因为代码走到这里任务就已经结束了,now()可以认
为就是任务结束时间。
13   */
14   else
15   time = triggerTime(-p);
16  }
17
18  long triggerTime(long delay) {
19   /*
20   * 如果delay < Long.Max_VALUE/2,则下次执行时间为当前时间+delay。
21   *
22   * 否则为了避免队列中出现由于溢出导致的排序紊乱,需要调用overflowFree来修正一下
delay(如果有必要的话)。
23   */
24   return now() + ((delay < (Long.MAX_VALUE >> 1)) ? delay :
overflowFree(delay));
25  }
26
27  /**
28   * 主要就是有这么一种情况：
29   * 某个任务的delay为负数，说明当前可以执行(其实早该执行了)。
```

```
30   * 工作队列中维护任务顺序是基于compareTo的，在compareTo中比较两个任务的顺序会用time
相减，负数则说明优先级高。
31   *
32   * 那么就有可能出现一个delay为正数,减去另一个为负数的delay，结果上溢为负数，则会导致
compareTo产生错误的结果。
33   *
34   * 为了特殊处理这种情况，首先判断一下队首的delay是不是负数，如果是正数不用管了,怎么
减都不会溢出。
35   * 否则可以拿当前delay减去队首的delay来比较看，如果不出现上溢，则整个队列都ok，排序
不会乱。
36   * 不然就把当前delay值给调整为Long.MAX_VALUE + 队首delay。
37   */
38  private long overflowFree(long delay) {
39   Delayed head = (Delayed) super.getQueue().peek();
40   if (head != null) {
41   long headDelay = head.getDelay(NANOSECONDS);
42   if (headDelay < 0 && (delay - headDelay < 0))
43   delay = Long.MAX_VALUE + headDelay;
44   }
45   return delay;
46  }
```

循环的根据key节点与它的父节点来判断，如果key节点的执行时间小于父节点，则将两个节点交换，使执行时间靠前的节点排列在队列的前面。

可以理解为一个树形的结构，最小点堆的结构；父节点一定小于子节点；



DelayQueue内部封装了一个PriorityQueue，它会根据time的先后时间排序（time小的排在前面），若time相同则根据sequenceNumber排序（ sequenceNumber小的排在前面）；