Java任务调度线程池ScheduledThreadPoolExecutor原理解析

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/u011983531/article/details/79071171

　　ScheduledThreadPoolExecutor是JDK在ThreadPoolExecutor的基础上实现的任务调度线程池。   
　　ScheduledThreadPoolExecutor的构造函数全部是调用父类(也就是ThreadPoolExecutor)的构造函数。其中，核心线程数是必须设置的，最大线程数是Integer.MAX\_VALUE，空闲工作线程生存时间是0，阻塞队列是DelayedWorkQueue。   
　　DelayedWorkQueue内部使用一个初始容量为16的数组来保存任务，容量不够时会扩容，所以可以任务DelayedWorkQueue是一个无界队列，那么最大线程数的设置也是没有意义的。

关于ThreadPoolExecutor的详细解释，可以参考：   
<http://blog.csdn.net/u011983531/article/details/49369489>

*//构造函数*

public ScheduledThreadPoolExecutor(int corePoolSize) {

super(corePoolSize, Integer.MAX\_VALUE, 0, TimeUnit.NANOSECONDS,

new DelayedWorkQueue());

}

　　既然ScheduledThreadPoolExecutor的构造函数全部使用父类的，那么又是如何实现定时调度的呢？对比ScheduledThreadPoolExecutor与ThreadPoolExecutor，不同之处主要是下面两点：

1. 任务不同。ScheduledThreadPoolExecutor的任务统一被封装成了**ScheduledFutureTask**对象，而ThreadPoolExecutor执行的还是原始的Runnable的对象。
2. 阻塞队列不同。ScheduledThreadPoolExecutor使用的是DelayedWorkQueue，顾名思义，这是一个延时队列。

我们以scheduleAtFixedRate()方法为例来看看具体是如何实现的。   
scheduleAtFixedRate的大致逻辑如下：

1. 将任务封装成一个ScheduledFutureTask对象
2. 将ScheduledFutureTask对象放到延时队列中

/\*\*

\* 主要任务：

\* 1.封装一个ScheduledFutureTask对象

\* 2.执行delayedExecute()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));

RunnableScheduledFuture<Void> t = decorateTask(command, sft);

sft.outerTask = t;

delayedExecute(t);

return t;

}

/\*\*

\* 主要任务：

\* 1.将task添加到队列中

\* /

private void delayedExecute(RunnableScheduledFuture<?> task) {

if (isShutdown())

reject(task);

else {

super.getQueue().add(task);

if (isShutdown() &&

!canRunInCurrentRunState(task.isPeriodic()) &&

remove(task))

task.cancel(false);

else

ensurePrestart();

}

}

　　所以，下面最重要的应该是延时队列DelayedWorkQueue的offer和take方法了，来看看是怎么实现的。   
　　DelayedWorkQueue内部使用数组去维护任务队列的，那么数组是怎么保证任务有序呢？   
　　其实仔细看代码，我们能发现，这里的实现是用一个二叉堆去对数组元素进行排序。确切的说是小顶堆。那么小顶堆是依据什么来排序的呢？   
　　因为ScheduledFutureTask实现了Comparable接口，是按照任务执行的时间来倒叙排序的。

*//首先判断容量，如果容量不够就扩容，接着判断是不是第一个元素，如果是，*

*//那么直接放在index为0的位置，不是的话进行上滤操作。接下来判断添加的元素是不是*

*//在堆顶，如果是那么需要进行优先调度，那么进行signal*

public boolean offer(Runnable x) {

if (x == null)

throw new NullPointerException();

RunnableScheduledFuture e = (RunnableScheduledFuture)x;

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);

} else {

*//根据任务的下一次执行时间比较，将最近需要执行的任务放到前面*

siftUp(i, e);

}

if (queue[0] == e) {

leader = null;

available.signal();

}

} finally {

lock.unlock();

}

return true;

}

*//毫无疑问，take中直接获取queue[0]，它是距离目前最近的要被执行的任务，*

*//先检测一下还有多长时间，任务会被执行，如果小于0，那么立刻弹出，*

*//并且做一个下滤操作，重新找出堆顶元素。如果不小于0，那么证明时间还没到，*

*//那么available.awaitNanos(delay);等到delay时间后自动唤醒，*

*//或者因为添加了一个更加紧急的任务即offer中的signal被调用了，那么唤醒，*

*//重新循环获取最优先执行的任务，如果delay小于0，那么直接弹出任务。*

public RunnableScheduledFuture take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly();

try {

for (;;) {

RunnableScheduledFuture first = queue[0];

if (first == null)

available.await();

else {

long delay = first.getDelay(TimeUnit.NANOSECONDS);

if (delay <= 0)

*//时间已到，取出*

return finishPoll(first);

else if (leader != null)

*//等待*

available.await();

else {

Thread thisThread = Thread.currentThread();

leader = thisThread;

try {

available.awaitNanos(delay);

} finally {

if (leader == thisThread)

leader = null;

}

}

}

}

} finally {

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

available.signal();

lock.unlock();

}

}

　　弄清楚了**延时**的实现原理，下面最关键的就是周期调度的原理了。这个是在ScheduledFutureTask的run方法里面实现的。   
　　判断是否是周期执行的，如果不是，直接执行，如果是，先执行，然后计算下一次执行时间，将任务重新添加到延时队列中。

public void run() {

boolean periodic = isPeriodic();

if (!canRunInCurrentRunState(periodic))

cancel(false);

else if (!periodic)

ScheduledFutureTask.super.run();

else if (ScheduledFutureTask.super.runAndReset()) {

setNextRunTime();

reExecutePeriodic(outerTask);

}

}