Executor框架

#### 前言

在Java中，使用线程来异步执行任务。  
Java线程的创建与销毁需要一定的开销，如果我们为每一个任务创建一个新线程来执行，这些线程的创建与销毁将消耗大量的计算资源。  
同时，为每一个任务创建一个新线程来执行，这种策略可能会使处于高负荷状态的应用最终崩溃。

Java的线程既是工作单元，也是执行机制。

从 **JDK 5** 开始，把 **工作单元** 与 **执行机制** 分离开来。

* **工作单元** :
  + Runnable
  + Callable
* **执行机制**：
  + Executor框架

#### **一、Executor框架简介**

##### 1.1 Executor框架的两级调度模型

在HotSpot VM的线程模型中，Java线程（java.lang.Thread）被 **一对一映射为本地操作系统线程**。Java线程启动时会创建一个本地操作系统线程；当该Java线程终止时，这个操作系统线程也会被回收。  
操作系统会调度所有线程并将它们分配给可用的CPU。  
在上层，Java多线程程序通常把应用分解为若干个任务，然后使用用户级的调度器（Executor框架）将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。这种两级调度模型的示意图下面有介绍。  
从下图中可以看出，应用程序通过Executor框架控制上层的调度；而下层的调度由操作系统内核控制，下层的调度不受应用程序的控制。

Diagram

Description automatically generated

##### 1.2 Executor框架的结构

* **任务**。包括被执行任务需要实现的接口：Runnable接口 或 Callable接口。
* **任务的执行**。包括任务执行机制的核心接口Executor，以及继承自Executor的ExecutorService接口。Executor框架有两个关键类实现了ExecutorService接口（ThreadPoolExecutor 和 ScheduledThreadPoolExecutor）。
* **异步计算的结果**。包括接口Future和实现Future接口的FutureTask类。
* Executor框架包含的主要的类与接口如下图所示：

Diagram

Description automatically generated

* 下面是这些类和接口的简介：
  + Executor是一个接口，它是Executor框架的基础，它将任务的提交与任务的执行分离开来。
  + ThreadPoolExecutor 是线程池的核心实现类，用来执行被提交的任务。
  + ScheduledThreadPoolExecutor 是一个实现类，可以在给定的延迟后运行命令，或者定期执行命令。ScheduledThreadPoolExecutor比Timer更灵活，功能更强大。
  + Future接口和实现Future接口的FutureTask类，代表异步计算的结果。
  + Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor 或ScheduledThreadPoolExecutor执行。

##### 1.3 Executor框架的使用

Diagram

Description automatically generated

主线程首先要创建实现Runnable或者Callable接口的任务对象。  
工具类Executors可以通过以下两个方法把一个Runnable对象封装为一个Callable对象：

* Executors.callable(Runnable task)
* Executors.callable(Runnable task, Object resule)。

然后可以把Runnable对象直接交给ExecutorService执行ExecutorService.execute(Runnable command)；或者也可以把Runnable对象或Callable对象提交给ExecutorService 执行 ExecutorService.submit(Runnable task) 或 ExecutorService.submit(Callable<T>task)。

如果执行ExecutorService.submit(…)，ExecutorService 将返回一个实现 Future 接口的对象（到目前为止的JDK中，返回的是FutureTask对象）。由于FutureTask实现了Runnable，程序员也可以创建FutureTask，然后直接交给ExecutorService执行。

最后，主线程可以执行 FutureTask.get() 方法来等待任务执行完成。主线程也可以执行  
FutureTask.cancel(boolean mayInterruptIfRunning)来取消此任务的执行。

##### 1.4 Executor框架的成员

这里将介绍Executor框架的主要成员：

* ThreadPoolExecutor
* ScheduledThreadPoolExecutor
* Future接口
* Runnable接口
* Callable接口
* Executors

##### 1.5 ThreadPoolExecutor

通常使用工厂类Executors来创建。Executors可以创建3种类型的ThreadPoolExecutor：

* SingleThreadExecutor：适用于需要保证顺序地执行各个任务；并且在任意时间点，不会有多个线程是活动的应用场景。  
  下面是Executors提供的，创建使用单个线程的SingleThreadExecutor的API。

public static ExecutorService newSingleThreadExecutor() {

return new Executors.FinalizableDelegatedExecutorService(

new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue())

);

}

public static ExecutorService newSingleThreadExecutor(ThreadFactory var0) {

return new Executors.FinalizableDelegatedExecutorService(

new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue(), var0)

);

}

* FixedThreadPool：适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，它适用于负载比较重的服务器。  
  下面是Executors提供的，创建使用固定线程数的FixedThreadPool的API：

public static ExecutorService newFixedThreadPool(int var0) {

return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue());

}

public static ExecutorService newFixedThreadPool(int var0, ThreadFactory var1) {

return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue(), var1);

}

* CachedThreadPool：是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。  
  下面是Executors提供的，创建一个会根据需要创建新线程的CachedThreadPool的API：

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue());

}

public static ExecutorService newCachedThreadPool(ThreadFactory var0) {

return new ThreadPoolExecutor(0, 2147483647, 60L, TimeUnit.SECONDS, new SynchronousQueue(), var0);

}

* ScheduledThreadPoolExecutor  
  ScheduledThreadPoolExecutor通常使用工厂类Executors来创建。  
  Executors可以创建2种类型的ScheduledThreadPoolExecutor，如下:
  + ScheduledThreadPoolExecutor。包含若干个线程的ScheduledThreadPoolExecutor。  
    适用于需要多个后台线程执行周期任务，同时为了满足资源管理的需求而需要限制后台线程的数量的应用场景。  
    下面是工厂类Executors提供的，创建 **固定个数** 线程的ScheduledThreadPoolExecutor的API。

public static ScheduledExecutorService newScheduledThreadPool(int var0) {

return new ScheduledThreadPoolExecutor(var0);

}

public static ScheduledExecutorService newScheduledThreadPool(int var0, ThreadFactory var1) {

return new ScheduledThreadPoolExecutor(var0, var1);

}

* + SingleThreadScheduledExecutor。只包含一个线程的ScheduledThreadPoolExecutor。  
    适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。  
    下面是Executors提供的，创建单个线程的SingleThreadScheduledExecutor的API。

public static ScheduledExecutorService newSingleThreadScheduledExecutor() {

return new Executors.DelegatedScheduledExecutorService(new ScheduledThreadPoolExecutor(1));

}

public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory var0) {

return new Executors.DelegatedScheduledExecutorService(new ScheduledThreadPoolExecutor(1, var0));

}

##### 1.6 Future接口

Future接口和实现Future接口的FutureTask类用来表示异步计算的结果。  
当我们把Runnable接口或Callable接口的实现类提交（submit）给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，ThreadPoolExecutor或 ScheduledThreadPoolExecutor会向我们返回一个FutureTask对象。  
下面是对应的API：

<T> Future<T> submit(Callable<T> task)

<T> Future<T> submit(Runnable task, T result)

Future<> submit(Runnable task)

有一点需要读者注意，到目前最新的JDK 8为止，Java通过上述API返回的是一个FutureTask对象。但从API可以看到，Java仅仅保证返回的是一个实现了Future接口的对象。在将来的JDK实现中，返回的可能不一定是FutureTask。

##### 1.7 Runnable接口和 Callable接口

Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。  
它们之间的区别是Runnable不会返回结果，而Callable可以返回结果。  
除了可以自己创建实现Callable接口的对象外，还可以使用工厂类Executors来把一个Runnable包装成一个Callable。

下面是Executors提供的，把一个Runnable包装成一个Callable的API。

public static Callable<Object> callable(Runnable task) // 假设返回对象Callable1

下面是Executors提供的，把一个Runnable和一个待返回的结果包装成一个Callable的API。

public static <T> Callable<T> callable(Runnable task, T result) // 假设返回对象Callable2

前面讲过，当我们把一个Callable对象（比如上面的Callable1或Callable2）提交给ThreadPoolExecutor或ScheduledThreadPoolExecutor执行时，submit(…)会向我们返回一个FutureTask对象。我们可以执行FutureTask.get()方法来 **等待任务执行完成**。当任务成功完成后FutureTask.get()将返回该任务的结果。

例如，如果提交的是对象Callable1，FutureTask.get()方法将返回null；如果提交的是对象Callable2，FutureTask.get()方法将返回result对象。

#### **二、ThreadPoolExecutor详解**

Executor框架最核心的类是ThreadPoolExecutor，它是线程池的实现类，主要由下列4个组件构成。

* corePool：核心线程池的大小。
* maximumPool：最大线程池的大小。
* BlockingQueue：用来暂时保存任务的工作队列。
* RejectedExecutionHandler：当ThreadPoolExecutor已经关闭或ThreadPoolExecutor已经饱和时（达到了最大线程池大小且工作队列已满），execute()方法将要调用的Handler。

通过Executor框架的工具类Executors，可以创建3种类型的ThreadPoolExecutor。

* FixedThreadPool
* SingleThreadExecutor
* CachedThreadPool

##### 2.1 FixedThreadPool详解

FixedThreadPool被称为可重用固定线程数的线程池。下面是FixedThreadPool的源代码实现。

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>());

}

FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。

当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止。这里把keepAliveTime设置为0L，意味着多余的空闲线程会被立即终止。

FixedThreadPool的execute()方法的运行示意图如下图所示。

Diagram

Description automatically generated

* 图中1：如果当前运行的线程数少于corePoolSize，则创建新线程来执行任务。
* 图中2：在线程池完成预热之后（当前运行的线程数等于corePoolSize），将任务加入LinkedBlockingQueue。
* 图中3：线程执行完1中的任务后，会在循环中反复从LinkedBlockingQueue获取任务来执行。

FixedThreadPool使用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。  
使用无界队列作为工作队列会对线程池带来如下影响：

* 当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize。
* 由于上一点，使用无界队列时maximumPoolSize将是一个无效参数。
* 由于前面两点，使用无界队列时keepAliveTime将是一个无效参数。
* 由于使用无界队列，运行中的FixedThreadPool（未执行方法shutdown()或shutdownNow()）不会拒绝任务（不会调用RejectedExecutionHandler.rejectedExecution方法）。

##### 2.2 SingleThreadExecutor详解

SingleThreadExecutor是使用单个worker线程的Executor。  
下面是SingleThreadExecutor的源代码实现：

public static ExecutorService newSingleThreadExecutor() {

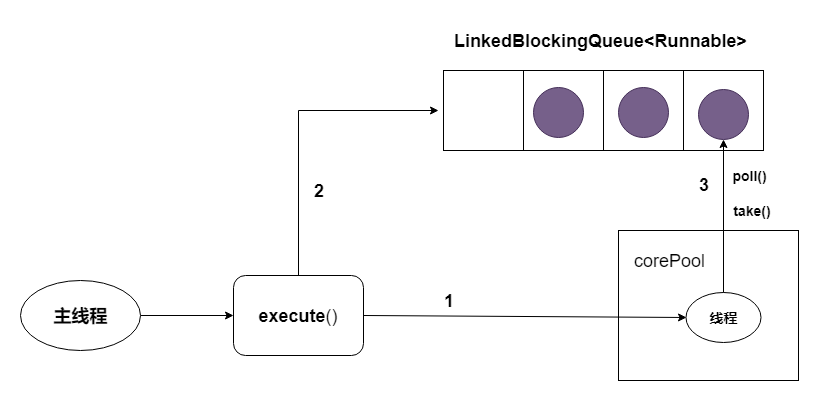
return new FinalizableDelegatedExecutorService(

new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>())

);

}

SingleThreadExecutor的corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。SingleThreadExecutor使用无界队列作为工作队列对线程池带来的影响与FixedThreadPool相同，这里就不赘述了。

SingleThreadExecutor的运行示意图如下图所示：  


* 上图1:如果当前运行的线程数少于corePoolSize（即线程池中无运行的线程），则创建一个新线程来执行任务。
* 上图2:在线程池完成预热之后（当前线程池中有一个运行的线程），将任务加入LinkedBlockingQueue。
* 上图3:线程执行完上图1中的任务后，会在一个无限循环中反复从LinkedBlockingQueue获取任务来执行。

##### 2.3 CachedThreadPool详解

CachedThreadPool是一个会根据需要创建新线程的线程池。  
下面是创建CachedThreadPool的源代码。

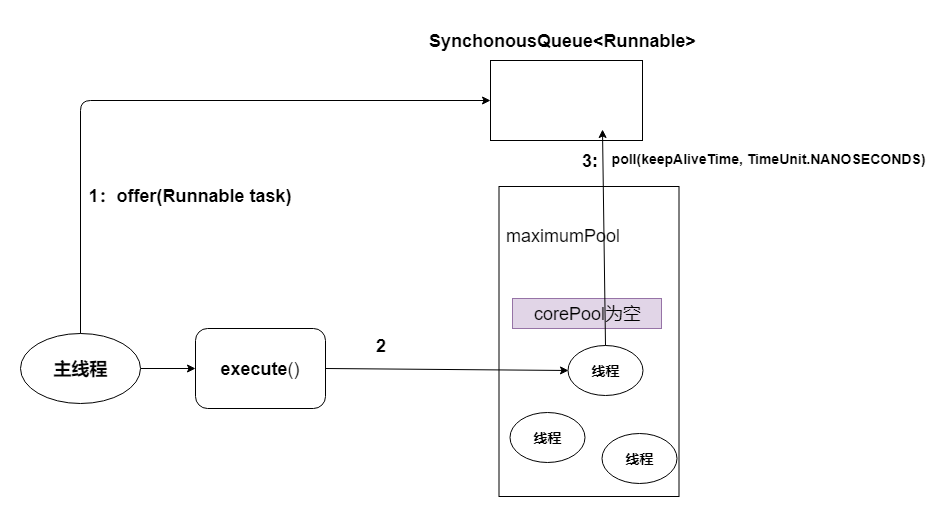
public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS,new SynchronousQueue<Runnable>());

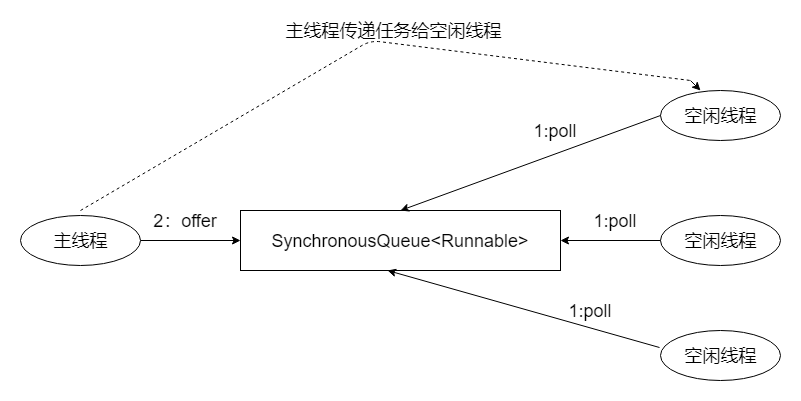
}

CachedThreadPool的corePoolSize被设置为0，即corePool为空；maximumPoolSize被设置为Integer.MAX\_VALUE，即maximumPool是无界的。这里把keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。

FixedThreadPool和SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列。  
CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。  
这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。  
极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。

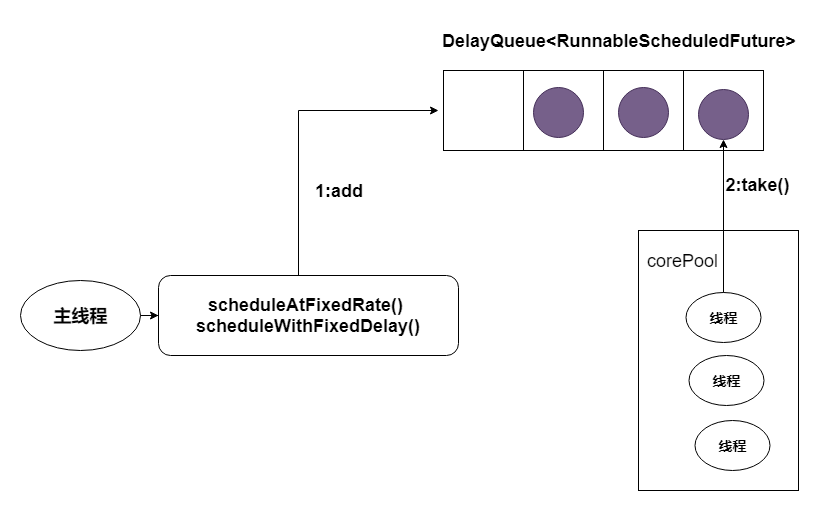
CachedThreadPool的execute()方法的执行示意图如下图所示：  


* 上图1： 首先执行SynchronousQueue.offer(Runnable task)。如果当前maximumPool中有空闲线程正在执行SynchronousQueue.poll(keepAliveTime，TimeUnit.NANOSECONDS)，那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成；否则执行下面的步骤2。
* 上图2：当初始maximumPool为空，或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll(keepAliveTime，TimeUnit.NANOSECONDS)。这种情况下，步骤1将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。
* 上图3：.在步骤2中新创建的线程将任务执行完后，会执行SynchronousQueue.poll(keepAliveTime，TimeUnit.NANOSECONDS)。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务（主线程执行步骤1），那么这个空闲线程将执行主线程提交的新任务；否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用任何资源。

前面提到过，SynchronousQueue是一个没有容量的阻塞队列。每个插入操作必须等待另一个线程的对应移除操作，反之亦然。CachedThreadPool使用SynchronousQueue，把主线程提交的任务传递给空闲线程执行。  
CachedThreadPool中任务传递的示意图如下图所示：  


##### 2.4 ScheduledThreadPoolExecutor详解

ScheduledThreadPoolExecutor继承自ThreadPoolExecutor。它主要用来在给定的延迟之后运行任务，或者定期执行任务。ScheduledThreadPoolExecutor的功能与Timer类似，但ScheduledThreadPoolExecutor功能更强大、更灵活。Timer对应的是单个后台线程，而ScheduledThreadPoolExecutor可以在构造函数中指定多个对应的后台线程数。

* ScheduledThreadPoolExecutor的运行机制  
  ScheduledThreadPoolExecutor的执行示意图（基于JDK 6）如下图所示：  
  

DelayQueue是一个无界队列，所以ThreadPoolExecutor的maximumPoolSize在ScheduledThreadPoolExecutor中没有什么意义（设置maximumPoolSize的大小没有什么效果）。

ScheduledThreadPoolExecutor的执行主要分为两大部分。

* 当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或者scheduleWithFixedDelay()方法时，会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了RunnableScheduledFutur接口的ScheduledFutureTask。
* 线程池中的线程从DelayQueue中获取ScheduledFutureTask，然后执行任务。

ScheduledThreadPoolExecutor为了实现周期性的执行任务，对ThreadPoolExecutor做了如下的修改。

* 使用DelayQueue作为任务队列。
* 获取任务的方式不同（后文会说明）。
* 执行周期任务后，增加了额外的处理（后文会说明）。

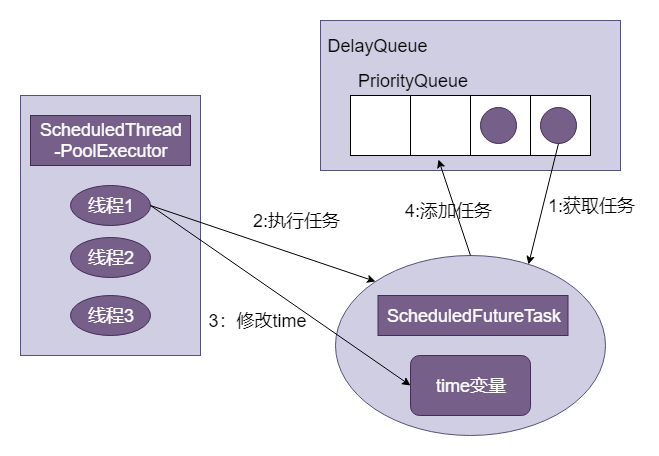
##### 2.5 ScheduledThreadPoolExecutor的实现

前面我们提到过，ScheduledThreadPoolExecutor会把待调度的任务（ScheduledFutureTask）放到一个DelayQueue中。

ScheduledFutureTask主要包含3个成员变量，如下。

* long型成员变量time，表示这个任务将要被执行的具体时间。
* long型成员变量sequenceNumber，表示这个任务被添加到ScheduledThreadPoolExecutor中的序号。
* long型成员变量period，表示任务执行的间隔周期。

DelayQueue封装了一个PriorityQueue，这个PriorityQueue会对队列中的ScheduledFutureTask进行排序。排序时，time小的排在前面（时间早的任务将被先执行）。如果两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面（也就是说，如果两个任务的执行时间相同，那么先提交的任务将被先执行）。

首先，让我们看看ScheduledThreadPoolExecutor中的线程执行周期任务的过程。下图是ScheduledThreadPoolExecutor中的线程1执行某个周期任务的4个步骤。  


下面是对这4个步骤的说明：

* 上图1： 线程·1从DelayQueue中获取已到期的ScheduledFutureTask 的 DelayQueue.take()。到期任务是指ScheduledFutureTask的time大于等于当前时间。
* 上图2： 线程1执行这个ScheduledFutureTask。
* 上图3： 线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。
* 上图4： 线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中DelayQueue.add()。

接下来，让我们看看上面的步骤1获取任务的过程。下面是DelayQueue.take()方法的源代码实现：

public E take() throws InterruptedException {

final ReentrantLock lock = this.lock;

lock.lockInterruptibly(); // 1

try {

for (; ; ) {

E first = q.peek();

if (first == null) {

available.await(); // 2.1

} else {

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

if (delay > 0) {

long tl = available.awaitNanos(delay); // 2.2

} else {

E x = q.poll(); // 2.3.1

assert x != null;

if (q.size() != 0)

available.signalAll(); // 2.3.2

return x;

}

}

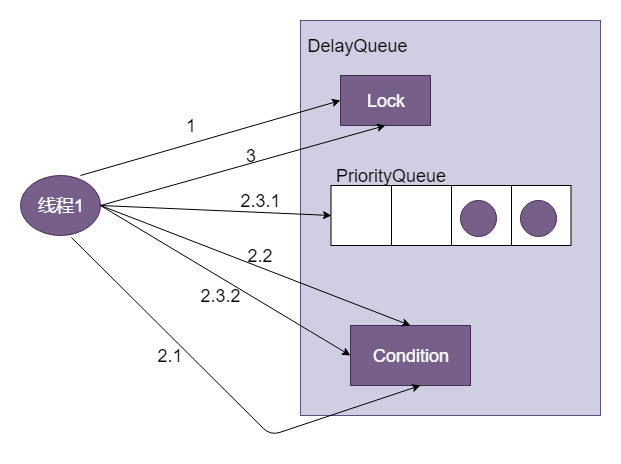
}

} finally {

lock.unlock(); // 3

}

}

下图是DelayQueue.take()的执行示意图：  
  
如图所示，获取任务分为3大步骤。

* 获取Lock。
* 获取周期任务。
  + 如果PriorityQueue为空，当前线程到Condition中等待；否则执行下面的2.2。
  + 如果PriorityQueue的头元素的time时间比当前时间大，到Condition中等待到time时间；否则执行下面的2.3。
  + 获取PriorityQueue的头元素（2.3.1）；如果PriorityQueue不为空，则唤醒在Condition中等待的所有线程（2.3.2）。
* 释放Lock。

ScheduledThreadPoolExecutor在一个循环中执行步骤2，直到线程从PriorityQueue获取到一个元素之后（执行2.3.1之后），才会退出无限循环（结束步骤2）。

最后，让我们看看ScheduledThreadPoolExecutor中的线程执行任务的步骤4，把ScheduledFutureTask放入DelayQueue中的过程。

下面是DelayQueue.add()的源代码实现：

public boolean offer(E e) {

final ReentrantLock lock = this.lock;

lock.lock(); // 1

try {

E first = q.peek();

q.offer(e); // 2.1

if (first == null || e.compareTo(first) < 0) {

available.signalAll(); // 2.2

}

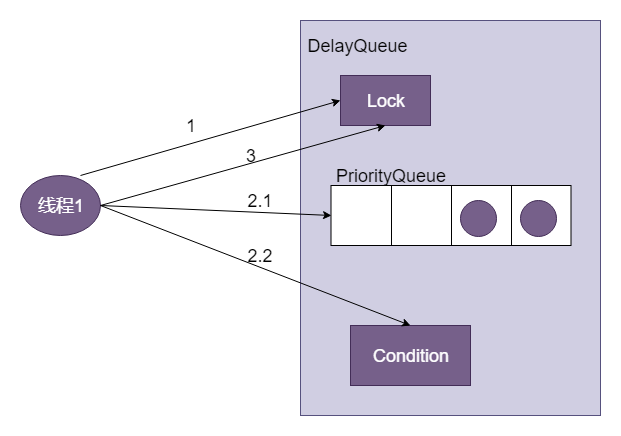
return true;

} finally {

lock.unlock(); // 3

}

}

下图是DelayQueue.add()的执行示意图：  
  
如图所示，添加任务分为3大步骤。

* 获取Lock。
* 添加任务。
  + 向PriorityQueue添加任务。
  + 如果在上面2.1中添加的任务是PriorityQueue的头元素，唤醒在Condition中等待的所有线程。
* 释放Lock。

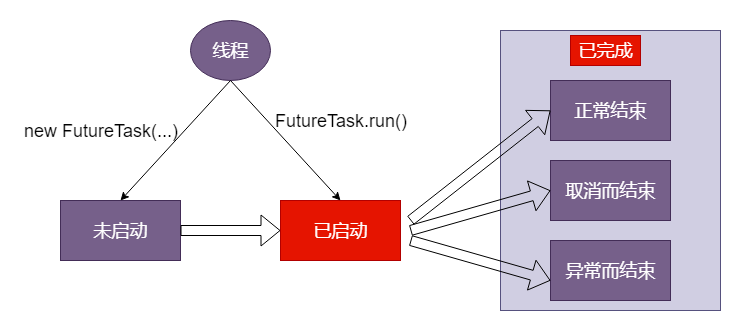
#### **三、FutureTask详解**

Future接口和实现Future接口的FutureTask类，代表 **异步计算的结果**。

##### 3.1 FutureTask简介

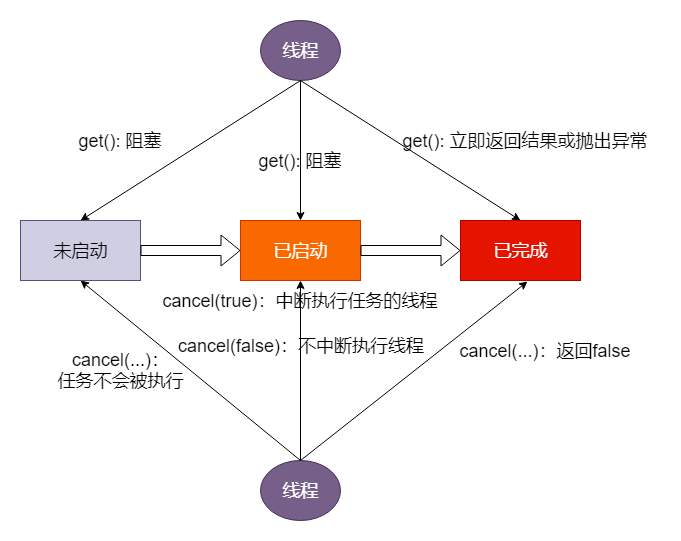
FutureTask除了实现Future接口外，还实现了Runnable接口。因此，FutureTask可以交给Executor执行，也可以由调用线程直接执行FutureTask.run()。根据FutureTask.run()方法被执行的时机，FutureTask可以处于下面3种状态。

* **未启动**。FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态。当创建一个FutureTask，且没有执行FutureTask.run()方法之前，这个FutureTask处于未启动状态。
* **已启动**。FutureTask.run()方法被执行的过程中，FutureTask处于已启动状态。
* **已完成**。FutureTask.run()方法执行完后正常结束，或被取消FutureTask.cancel(…)，或执行FutureTask.run()方法时抛出异常而异常结束，FutureTask处于已完成状态。

下图是FutureTask的 **状态迁移** 的示意图。  


当FutureTask处于未启动或已启动状态时，执行FutureTask.get()方法将导致调用线程阻塞；当FutureTask处于已完成状态时，执行FutureTask.get()方法将导致调用线程立即返回结果或抛出异常。

当FutureTask处于未启动状态时，执行FutureTask.cancel()方法将导致此任务永远不会被执行；当FutureTask处于已启动状态时，执行FutureTask.cancel(true)方法将以中断执行此任务线程的方式来试图停止任务；当FutureTask处于已启动状态时，执行FutureTask.cancel(false)方法将不会对正在执行此任务的线程产生影响（让正在执行的任务运行完成）；当FutureTask处于已完成状态时，执行FutureTask.cancel(…)方法将返回false。

下图是get方法和cancel方法的执行示意图：  


##### 3.2 FutureTask的使用

可以把FutureTask交给Executor执行；也可以通过ExecutorService.submit(...)方法返回一个FutureTask，然后执行FutureTask.get()方法或FutureTask.cancel(...)方法。除此以外，还可以单独使用FutureTask。

当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行，此时可以使用FutureTask。假设有多个线程执行若干任务，每个任务最多只能被执行一次。当多个线程试图同时执行同一个任务时，只允许一个线程执行任务，其他线程需要等待这个任务执行完后才能继续执行。

下面是对应的示例代码：

private final ConcurrentMap<Object, Future<String>> taskCache = new ConcurrentHashMap<>();

private String executionTask(final String taskName)

throws ExecutionException, InterruptedException {

while (true) {

Future<String> future = taskCache.get(taskName); // 1.1, 2.1

if (future == null) {

Callable<String> task = new Callable<String>() {

@Override

public String call() throws InterruptedException {

return taskName;

}

};

FutureTask<String> futureTask = new FutureTask<String>(task);

future = taskCache.putIfAbsent(taskName, futureTask); // 1.3

if (future == null) {

future = futureTask;

futureTask.run(); // 1.4执行任务

}

}

try {

return future.get(); // 1.5, 2.2

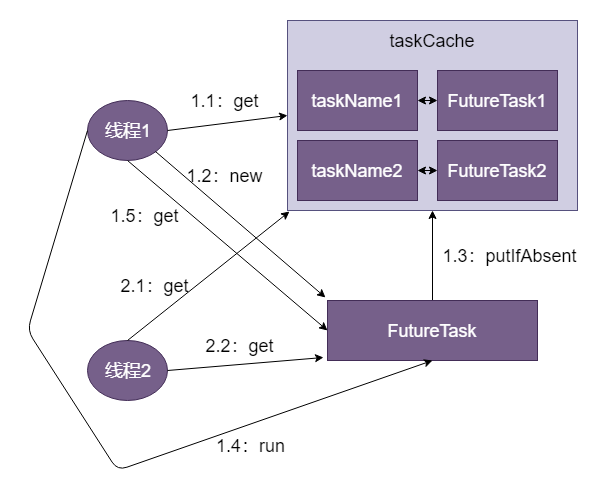
} catch (CancellationException e) {

taskCache.remove(taskName, future);

}

}

}

上述代码的执行示意图如下图所示：  


当两个线程试图同时执行同一个任务时，如果线程1执行1.3后线程2执行2.1，那么接下来线程2将在2.2等待，直到线程1执行完1.4后线程2才能2.2（FutureTask.get()）返回。

##### 3.3 FutureTask的实现

FutureTask的实现基于AbstractQueuedSynchronizer（以下简称为 **AQS**）。java.util.concurrent中的很多可阻塞类（比如ReentrantLock）都是基于AQS来实现的。AQS是一个同步框架，它提供通用机制来原子性管理同步状态、阻塞和唤醒线程，以及维护被阻塞线程的队列。  
**JDK 6** 中AQS被广泛使用，基于AQS实现的同步器包括：

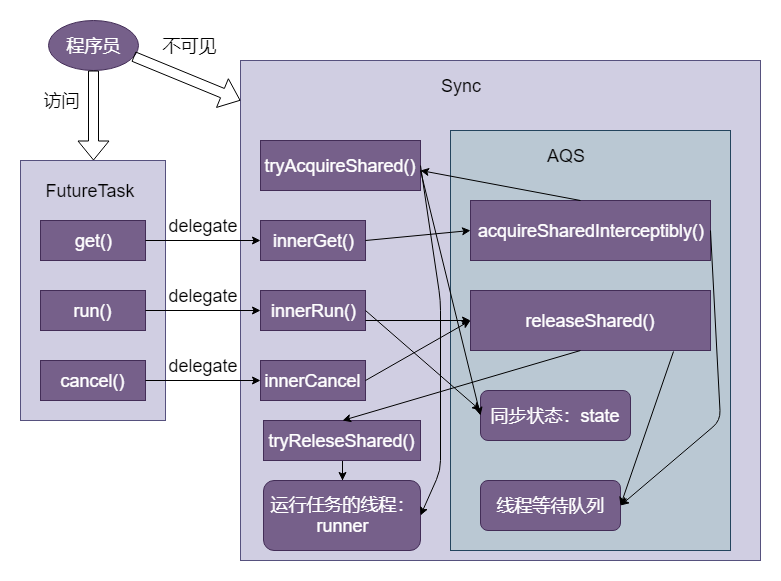
* ReentrantLock
* Semaphore
* ReentrantReadWriteLock
* CountDownLatch
* FutureTask。

每一个基于AQS实现的同步器都会包含两种类型的操作，如下:

* 至少一个acquire操作。这个操作阻塞调用线程，除非/直到AQS的状态允许这个线程继续执行。FutureTask的acquire操作为get() / get(long timeout，TimeUnit unit)方法调用。
* 至少一个release操作。这个操作改变AQS的状态，改变后的状态可允许一个或多个阻塞线程被解除阻塞。FutureTask的release操作包括run()方法和cancel(...)方法。

基于“ **复合优先于继承** ”的原则，FutureTask声明了一个内部私有的继承于AQS的子类Sync，对FutureTask所有公有方法的调用都会委托给这个内部子类。

AQS被作为“模板方法模式”的基础类提供给FutureTask的内部子类Sync，这个内部子类只需要实现状态检查和状态更新的方法即可，这些方法将控制FutureTask的获取和释放操作。具体来说，Sync实现了AQS的tryAcquireShared(int)方法和tryReleaseShared(int)方法，Sync通过这两个方法来检查和更新同步状态。

FutureTask的设计示意图如下图所示：  


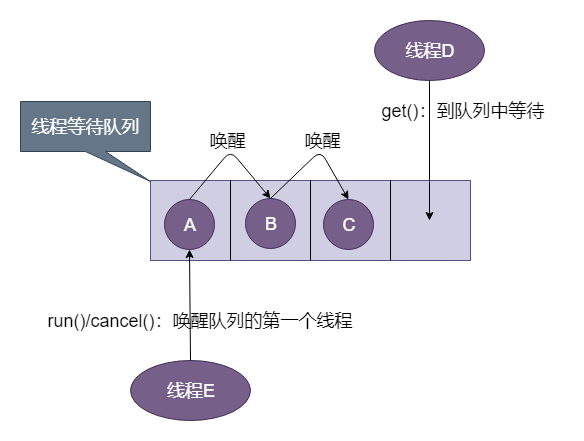
如图所示，Sync是FutureTask的内部私有类，它继承自AQS。创建FutureTask时会创建内部私有的成员对象Sync，FutureTask所有的的公有方法都直接委托给了内部私有的Sync。

FutureTask.get()方法会调用AQS.acquireSharedInterruptibly(int arg)方法，这个方法的执行过程如下：

* 调用AQS.acquireSharedInterruptibly(int arg)方法，这个方法首先会回调在子类Sync中实现的tryAcquireShared()方法来判断acquire操作是否可以成功。acquire操作可以成功的条件为：state为执行完成状态RAN或已取消状态CANCELLED，且runner不为null。
* 如果成功则get()方法立即返回。如果失败则到线程等待队列中去等待其他线程执行  
  release操作。
* 当其他线程执行release操作（比如FutureTask.run()或FutureTask.cancel(…)）唤醒当前线程后，当前线程再次执行tryAcquireShared()将返回正值1，当前线程将离开线程等待队列并唤醒它的后继线程（这里会产生级联唤醒的效果，后面会介绍）。
* 最后返回计算的结果或抛出异常。

FutureTask.run()的执行过程如下:

* 执行在构造函数中指定的任务（Callable.call()）。
* 以原子方式来更新同步状态（调用AQS.compareAndSetState(int expect，int update)，设置state为执行完成状态RAN）。如果这个原子操作成功，就设置代表计算结果的变量result的值为Callable.call()的返回值，然后调用AQS.releaseShared(int arg)。
* AQS.releaseShared(int arg)首先会回调在子类Sync中实现的tryReleaseShared(arg)来执行release操作（设置运行任务的线程runner为null，然会返回true）；AQS.releaseShared(int arg)，然后唤醒线程等待队列中的第一个线程。
* 调用FutureTask.done()。

当执行FutureTask.get()方法时，如果FutureTask不是处于执行完成状态RAN或已取消状态CANCELLED，当前执行线程将到AQS的线程等待队列中等待（见下图的线程A、B、C和D）。当某个线程执行FutureTask.run()方法或FutureTask.cancel(...)方法时，会唤醒线程等待队列的第一个线程（见下图所示的线程E唤醒线程A）。  


假设开始时FutureTask处于未启动状态或已启动状态，等待队列中已经有3个线程（A、B和C）在等待。此时，线程D执行get()方法将导致线程D也到等待队列中去等待。

当线程E执行run()方法时，会唤醒队列中的第一个线程A。线程A被唤醒后，首先把自己从队列中删除，然后唤醒它的后继线程B，最后线程A从get()方法返回。线程B、C和D重复A线程的处理流程。最终，在队列中等待的所有线程都被级联唤醒并从get()方法返回。