https://mp.weixin.qq.com/s/w31BE7cWBSWWiw-gFu3vEQ

# JUC中的Executor框架详解1

## 本文主要内容

1. 介绍Executor框架相关内容
2. 介绍Executor
3. 介绍ExecutorService
4. 介绍线程池ThreadPoolExecutor及案例
5. 介绍定时器ScheduledExecutorService及案例
6. 介绍Excecutors类的使用
7. 介绍Future接口
8. 介绍Callable接口
9. 介绍FutureTask的使用
10. 获取异步任务的执行结果的几种方法

## Executors框架介绍

Executors框架是Doug Lea的神作，通过这个框架，可以很容易的使用线程池高效地处理并行任务。

**Excecutor框架主要包含3部分的内容：**

1、任务相关的：包含被执行的任务要实现的接口：Runnable接口或Callable接口

2、任务的执行相关的：包含任务执行机制的核心接口Executor，以及继承自Executor的ExecutorService接口。Executor框架中有两个关键的类实现了ExecutorService接口（ThreadPoolExecutor和ScheduleThreadPoolExecutor）

3、异步计算结果相关的：包含接口Future和实现Future接口的FutureTask类

**Executors框架包括：**

* Executor
* ExecutorService
* ThreadPoolExecutor
* Executors
* Future
* Callable
* FutureTask
* CompletableFuture
* CompletionService
* ExecutorCompletionService

下面我们来一个个介绍其用途和使用方法。

## **Executor接口**

Executor接口中定义了方法execute(Runable able)接口，该方法接受一个Runable实例，他来执行一个任务，任务即是一个实现了Runable接口的类。

## **ExecutorService接口**

ExecutorService继承于Executor接口，他提供了更为丰富的线程实现方法，比如ExecutorService提供关闭自己的方法，以及为跟踪一个或多个异步任务执行状况而生成Future的方法。

**ExecutorService有三种状态：运行、关闭、终止**。创建后便进入运行状态，当调用了shutdown()方法时，便进入了关闭状态，此时意味着ExecutorService不再接受新的任务，但是他还是会执行已经提交的任务，当所有已经提交了的任务执行完后，便达到终止状态。如果不调用shutdown方法，ExecutorService方法会一直运行下去，系统一般不会主动关闭。

## **ThreadPoolExecutor类**

线程池类，实现了ExecutorService接口中所有方法，该类也是我们经常要用到的，非常重要，关于此类有详细的介绍，可以移步第18篇。

## **ScheduleThreadPoolExecutor定时器**

ScheduleThreadPoolExecutor继承自ScheduleThreadPoolExecutor，他主要用来延迟执行任务，或者定时执行任务。功能和Timer类似，但是ScheduleThreadPoolExecutor更强大、更灵活一些。Timer后台是单个线程，而ScheduleThreadPoolExecutor可以在创建的时候指定多个线程。

常用方法介绍：

### schedule:延迟执行任务1次

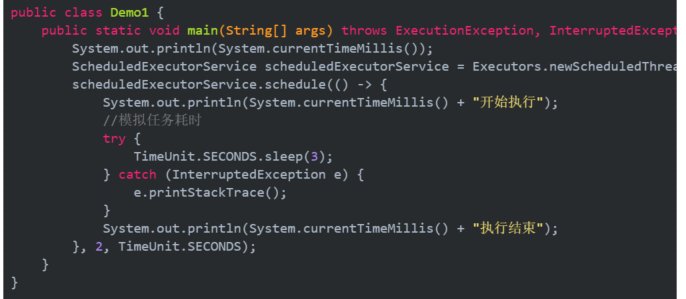
使用ScheduleThreadPoolExecutor的schedule方法，看一下这个方法的声明：

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)

3个参数：

* + command：需要执行的任务
  + delay：需要延迟的时间
  + unit：参数2的时间单位，是个枚举，可以是天、小时、分钟、秒、毫秒、纳秒等

**示例代码：**



输出：



### scheduleAtFixedRate:固定的频率执行任务

使用ScheduleThreadPoolExecutor的scheduleAtFixedRate方法，该方法设置了执行周期，下一次执行时间相当于是上一次的执行时间加上period，任务每次执行完毕之后才会计算下次的执行时间。

看一下这个方法的声明：

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit)

4个参数：

* command：表示要执行的任务
* initialDelay：表示延迟多久执行第一次
* period：连续执行之间的时间间隔
* unit：参数2和参数3的时间单位，是个枚举，可以是天、小时、分钟、秒、毫秒、纳秒等

假设系统调用scheduleAtFixedRate的时间是T1，那么执行时间如下：

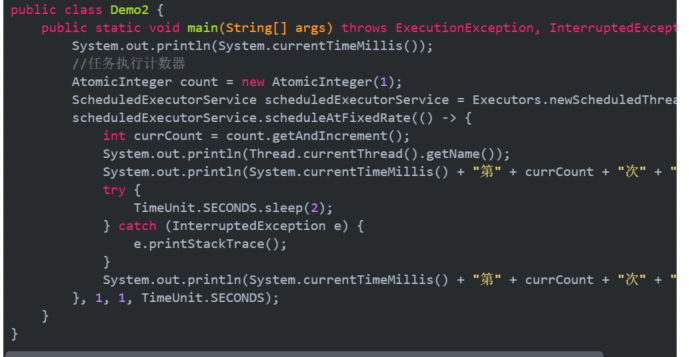
第1次：T1+initialDelay

第2次：T1+initialDelay+period

第3次：T1+initialDelay+2\*period

第n次：T1+initialDelay+(n-1)\*period

**示例代码：**



前面6次输出结果：



代码中设置的任务第一次执行时间是系统启动之后延迟一秒执行。后面每次时间间隔1秒，从输出中可以看出系统启动之后过了1秒任务第一次执行（1、3行输出），输出的结果中可以看到任务第一次执行结束时间和第二次的结束时间一样，为什么会这样？前面有介绍，任务当前执行完毕之后会计算下次执行时间，下次执行时间为上次执行的开始时间+period，第一次开始执行时间是1564576405247，加1秒为1564576406247，这个时间小于第一次结束的时间了，说明小于系统当前时间了，会立即执行。

### scheduleWithFixedDelay:固定的间隔执行任务

使用ScheduleThreadPoolExecutor的scheduleWithFixedDelay方法，该方法设置了执行周期，与scheduleAtFixedRate方法不同的是，下一次执行时间是上一次任务执行完的系统时间加上period，因而具体执行时间不是固定的，但周期是固定的，是采用相对固定的延迟来执行任务。看一下这个方法的声明：

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit)

4个参数：

* + - command：表示要执行的任务
    - initialDelay：表示延迟多久执行第一次
    - period：表示下次执行时间和上次执行结束时间之间的间隔时间
    - unit：参数2和参数3的时间单位，是个枚举，可以是天、小时、分钟、秒、毫秒、纳秒等

假设系统调用scheduleAtFixedRate的时间是T1，那么执行时间如下：

第1次：T1+initialDelay，执行结束时间：E1

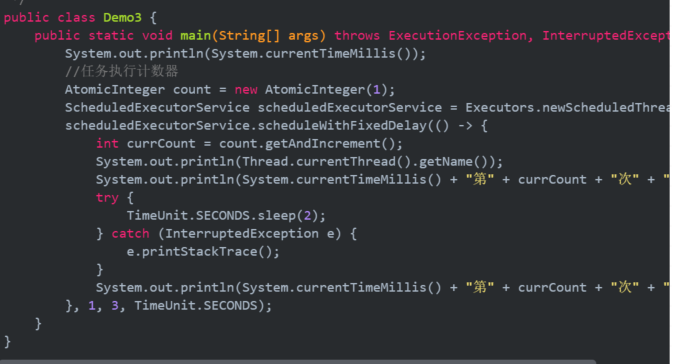
第2次：E1+period，执行结束时间：E2

第3次：E2+period，执行结束时间：E3

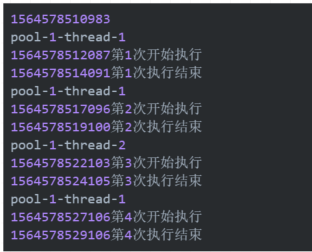
第4次：E3+period，执行结束时间：E4

第n次：上次执行结束时间+period

**示例代码：**



前几次输出如下：

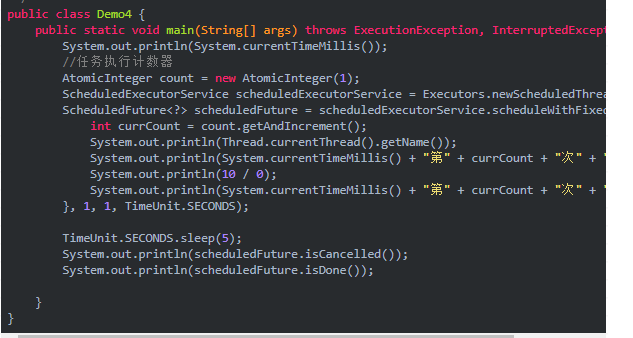


延迟1秒之后执行第1次，后面每次的执行时间和上次执行结束时间间隔3秒。

scheduleAtFixedRate和scheduleWithFixedDelay示例建议多看2遍。

### 定时任务有异常会怎么样？

示例代码：



系统输出如下内容就再也没有输出了：

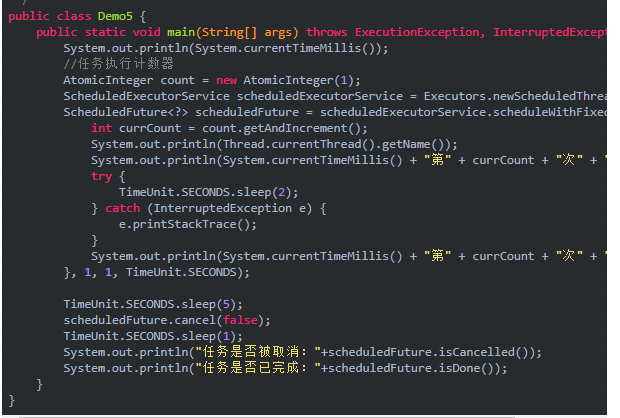


**先说补充点知识**：schedule、scheduleAtFixedRate、scheduleWithFixedDelay这几个方法有个返回值ScheduledFuture，通过ScheduledFuture可以对执行的任务做一些操作，如判断任务是否被取消、是否执行完成。

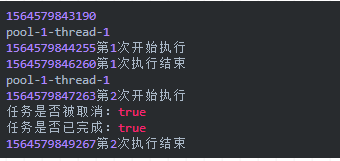
再回到上面代码，任务中有个10/0的操作，会触发异常，发生异常之后没有任何现象，被ScheduledExecutorService内部给吞掉了，然后这个任务再也不会执行了，scheduledFuture.isDone()输出true，表示这个任务已经结束了，再也不会被执行了。**所以如果程序有异常，开发者自己注意处理一下，不然跑着跑着发现任务怎么不跑了，也没有异常输出。**

### 取消定时任务的执行

可能任务执行一会，想取消执行，可以调用ScheduledFuture的cancel方法，参数表示是否给任务发送中断信号。



输出：



输出中可以看到任务被取消成功了。

## **Executors类**

Executors类，提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。常用的方法有：

### newSingleThreadExecutor

public static ExecutorService newSingleThreadExecutor()  
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。**如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它**。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。内部使用了无限容量的LinkedBlockingQueue阻塞队列来缓存任务，任务如果比较多，单线程如果处理不过来，会导致队列堆满，引发OOM。

### newFixedThreadPool

public static ExecutorService newFixedThreadPool(int nThreads)  
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，在提交新任务，任务将会进入等待队列中等待。如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。内部使用了无限容量的LinkedBlockingQueue阻塞队列来缓存任务，任务如果比较多，如果处理不过来，会导致队列堆满，引发OOM。

### newCachedThreadPool

public static ExecutorService newCachedThreadPool()  
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，

那么就会回收部分空闲（60秒处于等待任务到来）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池的最大值是Integer的最大值(2^31-1)。内部使用了SynchronousQueue同步队列来缓存任务，此队列的特性是放入任务时必须要有对应的线程获取任务，任务才可以放入成功。如果处理的任务比较耗时，任务来的速度也比较快，会创建太多的线程引发OOM。

### newScheduledThreadPool

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

在《阿里巴巴java开发手册》中指出了线程资源必须通过线程池提供，不允许在应用中自行显示的创建线程，这样一方面是线程的创建更加规范，可以合理控制开辟线程的数量；另一方面线程的细节管理交给线程池处理，优化了资源的开销。而线程池不允许使用Executors去创建，而要通过ThreadPoolExecutor方式，这一方面是由于jdk中Executor框架虽然提供了如newFixedThreadPool()、newSingleThreadExecutor()、newCachedThreadPool()等创建线程池的方法，但都有其局限性，不够灵活；另外由于前面几种方法内部也是通过ThreadPoolExecutor方式实现，使用ThreadPoolExecutor有助于大家明确线程池的运行规则，创建符合自己的业务场景需要的线程池，避免资源耗尽的风险。

## **Future、Callable接口**

Future接口定义了操作异步异步任务执行一些方法，**如获取异步任务的执行结果、取消任务的执行、判断任务是否被取消、判断任务执行是否完毕**等。

Callable接口中定义了需要有返回的任务需要实现的方法。

@FunctionalInterface

public interface Callable<V> {

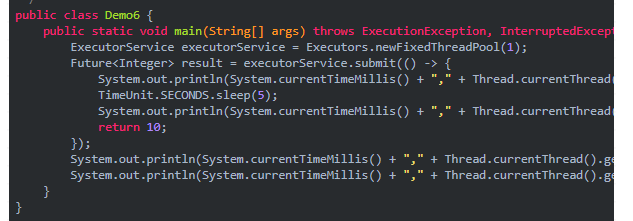
V call() throws Exception;

}

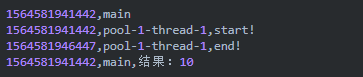
比如主线程让一个子线程去执行任务，子线程可能比较耗时，启动子线程开始执行任务后，主线程就去做其他事情了，过了一会才去获取子任务的执行结果。

### 获取异步任务执行结果

**示例代码：**



输出：



代码中创建了一个线程池，调用线程池的submit方法执行任务，submit参数为Callable接口：表示需要执行的任务有返回值，submit方法返回一个Future对象，Future相当于一个凭证，可以在任意时间拿着这个凭证去获取对应任务的执行结果（调用其get方法），代码中调用了result.get()方法之后，此方法会阻塞当前线程直到任务执行结束。

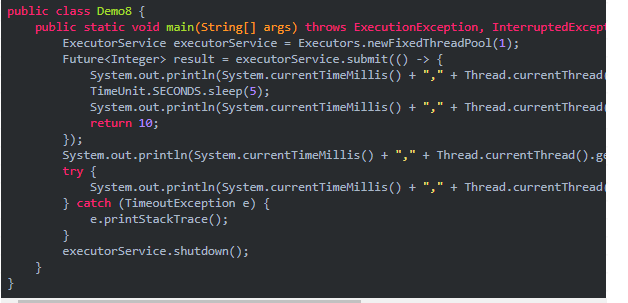
### 超时获取异步任务执行结果

可能任务执行比较耗时，比如耗时1分钟，我最多只能等待10秒，如果10秒还没返回，我就去做其他事情了。

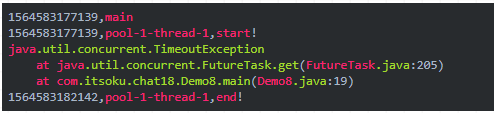
刚好get有个超时的方法，声明如下：

V get(long timeout, TimeUnit unit)  
        throws InterruptedException, ExecutionException, TimeoutException;

**示例代码：**



输出：



任务执行中休眠了5秒，get方法获取执行结果，超时时间是3秒，3秒还未获取到结果，get触发了TimeoutException异常，当前线程从阻塞状态苏醒了。

### Future其他方法

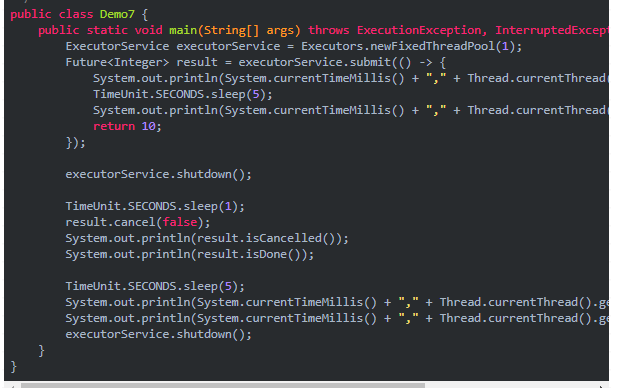
**cancel**：取消在执行的任务，参数表示是否对执行的任务发送中断信号，方法声明如下：

boolean cancel(boolean mayInterruptIfRunning);

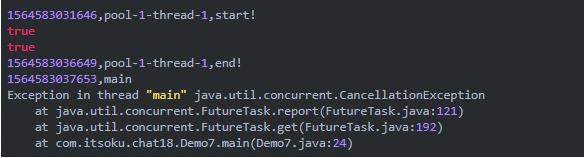
**isCancelled**：用来判断任务是否被取消

**isDone**：判断任务是否执行完毕。

**cancel方法来个示例：**



输出：



输出2个true，表示任务已被取消，已完成，最后调用get方法会触发CancellationException异常。

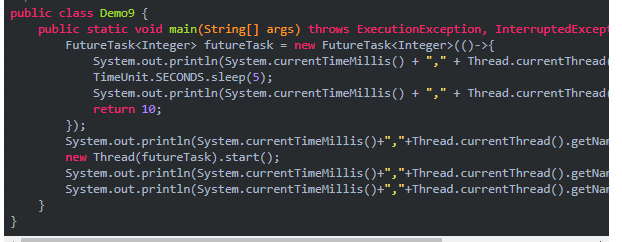
**总结：从上面可以看出Future、Callable接口需要结合ExecutorService来使用，需要有线程池的支持。**

## **FutureTask类**

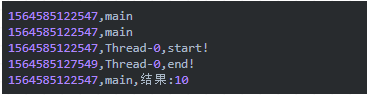
FutureTask除了实现Future接口，还实现了Runnable接口，因此FutureTask可以交给Executor执行，也可以交给线程执行执行（**Thread有个Runnable的构造方法**），**FutureTask**表示带返回值结果的任务。

上面我们演示的是通过线程池执行任务然后获取执行结果。

这次我们通过FutureTask类，自己启动一个线程来获取执行结果，示例如下：



输出：



**大家可以回过头去看一下上面用线程池的submit方法返回的Future实际类型正是FutureTask对象，有兴趣的可以设置个断点去看看。**