一、定时任务概念

1. 基本介绍

1.1 应用场景：

闹钟程序或任务提醒，指定时间叫床或在指定日期提醒还信用卡

监控系统，每隔一段时间采集下系统数据，对异常事件报警

统计系统，一般凌晨一定时间统计昨日的各种数据指标

1.2 实现方式

使用java.util包中的Timer和TimerTask。

使用Java并发包中的ScheduledExecutorService。

二、Timer和TimerTask

1. 基本用法

1.1 TimerTask表示一个定时任务，它是一个抽象类，实现了Runnable，具体的定时任务需要继承该类，实现run方法。

public abstract class TimerTask extends Object implements Runnable;

1.2 Timer是一个具体类，它负责定时任务的调度和执行，它有如下主要方法：

public class Timer extends Object;

//在指定绝对时间time运行任务task

public void schedule(TimerTask task, Date time)

//在当前时间延时delay毫秒后运行任务task

public void schedule(TimerTask task, long delay)

//固定延时重复执行，第一次计划执行时间为firstTime，后一次的计划执行时间为前一次"实际"执行时间加上period

public void schedule(TimerTask task, Date firstTime, long period)

//同样是固定延时重复执行，第一次执行时间为当前时间加上delay

public void schedule(TimerTask task, long delay, long period)

//固定频率重复执行，第一次计划执行时间为firstTime，后一次的计划执行时间为前一次"计划"执行时间加上period

public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)

//同样是固定频率重复执行，第一次计划执行时间为当前时间加上delay

public void scheduleAtFixedRate(TimerTask task, long delay, long period)

(1) 注意：

① 需要注意固定延时(fixed-delay)与固定频率(fixed-rate)的区别，都是重复执行，但后一次任务执行相对的时间是不一样的，对于固定延时，它是基于上次任务的"实际"执行时间来算的，如果由于某种原因，上次任务延时了，则本次任务也会延时，而固定频率会尽量补够运行次数。

② 需要注意的是，如果第一次计划执行的时间firstTime是一个过去的时间，则任务会立即运行，对于固定延时的任务，下次任务会基于第一次执行时间计算，而对于固定频率的任务，则会从firstTime开始算，有可能加上period后还是一个过去时间，从而连续运行很多次，直到时间超过当前时间。

(2) 实例代码：

public class TimerFixedDelay {

//任务定义

static class LongRunningTask extends TimerTask {

@Override

public void run() {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

}

System.out.println("long running finished");

}

}

static class FixedDelayTask extends TimerTask {

@Override

public void run() {

System.out.println(System.currentTimeMillis());

}

}

public static void main(String[] args) throws InterruptedException {

Timer timer = new Timer();

//延时1s后执行任务

timer.schedule(new LongRunningTask (), 1000);

//固定延时重复

timer.schedule(new LongRunningTask(), 10);

timer.schedule(new FixedDelayTask(), 100, 1000);

//固定频率重复

timer.schedule(new LongRunningTask(), 10);

timer.scheduleAtFixedRate(new FixedRateTask(), 100, 1000);

}

}

2. 基本原理

(1) Timer内部主要由两部分组成，任务队列和Timer线程。任务队列是一个基于堆实现的优先级队列，按照下次执行的时间排优先级。Timer线程负责执行所有的定时任务，需要强调的是，一个Timer对象只有一个Timer线程，所以，对于上面的例子，任务才会被延迟。

(2) Timer线程主体是一个循环，从队列中拿任务，如果队列中有任务且计划执行时间小于等于当前时间，就执行它，如果队列中没有任务或第一个任务延时还没到，就睡眠。如果睡眠过程中队列上添加了新任务且新任务是第一个任务，Timer线程会被唤醒，重新进行检查。

(3) 在执行任务之前，Timer线程判断任务是否为周期任务，如果是，就设置下次执行的时间并添加到优先级队列中，对于固定延时的任务，下次执行时间为当前时间加上period，对于固定频率的任务，下次执行时间为上次计划执行时间加上period。

(4) 需要强调是，下次任务的计划是在执行当前任务之前就做出了的，对于固定延时的任务，延时相对的是任务执行前的当前时间，而不是任务执行后，这与后面讲到的ScheduledExecutorService的固定延时计算方法是不同的，后者的计算方法更合乎一般的期望。另一方面，对于固定频率的任务，它总是基于最先的计划计划的，所以，很有可能会出现前面例子中一下子执行很多次任务的情况。

3. 死循环

一个Timer对象只有一个Timer线程，这意味着，定时任务不能耗时太长，更不能是无限循环

4. 异常处理

(1) 在执行任何一个任务的run方法时，一旦run抛出异常，Timer线程就会退出，从而所有定时任务都会被取消。

(2) 如果希望各个定时任务不互相干扰，一定要在run方法内捕获所有异常。

5. 总结

(1) 背后只有一个线程在运行

(2) 固定频率的任务被延迟后，可能会立即执行多次，将次数补够

(3) 固定延时任务的延时相对的是任务执行前的时间

(4) 不要在定时任务中使用无限循环

(5) 一个定时任务的未处理异常会导致所有定时任务被取消

二、ScheduledExecutorService （接口）

1. 接口和类定义

(1) 由于Timer/TimerTask的一些问题，Java并发包引入了ScheduledExecutorService，它是一个接口，其定义为：

public interface ScheduledExecutorService extends ExecutorService {

//单次执行，在指定延时delay后运行command

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

//单次执行，在指定延时delay后运行callable

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

//固定频率重复执行

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

//固定延时重复执行

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

}

(2) 返回类型都是ScheduledFuture，它是一个接口，扩展了Future和Delayed，没有定义额外方法。这些方法的大部分语义与Timer中的基本是类似的。

(3) 对于固定频率的任务，第一次执行时间为initialDelay后，第二次为initialDelay+period，第三次initialDelay+2\*period，依次类推。不过，对于固定延时的任务，它是从任务执行后开始算的，第一次为initialDelay后，第二次为第一次任务执行结束后再加上delay。与Timer不同，它不支持以绝对时间作为首次运行的时间。

2. 主要实现类（ScheduledThreadPoolExecutor）

2.1 构造方法：

线程池ThreadPoolExecutor的子类，是基于线程池实现的，它的主要构造方法是：

public ScheduledThreadPoolExecutor(int corePoolSize)

(1) 此外，还有构造方法可以接受参数ThreadFactory和RejectedExecutionHandler，含义与ThreadPoolExecutor一样它的任务队列是一个无界的优先级队列，所以最大线程数对它没有作用，即使corePoolSize设为0，它也会至少运行一个线程。

2.2 创建方法 -- 工厂类Executors

(1) 基本方法

//单线程的定时任务执行服务

public static ScheduledExecutorService newSingleThreadScheduledExecutor()

public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)

//多线程的定时任务执行服务

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,

ThreadFactory threadFactory)

2.3 实例代码：

由于可以有多个线程执行定时任务，一般任务就不会被某个长时间运行的任务所延迟。

public static void main(String[] args) throws InterruptedException {

//可以创建多个线程来执行定时任务，多个任务不会被阻塞

ScheduledExecutorService timer = Executors.newScheduledThreadPool(10);

timer.schedule(new LongRunningTask(), 10, TimeUnit.MILLISECONDS);

timer.scheduleWithFixedDelay(new FixedDelayTask(), 100, 1000,

TimeUnit.MILLISECONDS);

}

(1) 与Timer不同，单个定时任务的异常不会再导致整个定时任务被取消了，即使背后只有一个线程执行任务。

3. 基本原理

ScheduledThreadPoolExecutor的实现思路与Timer基本是类似的，都有一个基于堆的优先级队列，保存待执行的定时任务，它的主要不同是：

(1) 底层实现是线程池，可以有多个线程执行任务

(2) 在任务执行后再设置下次执行的时间，对于固定延时的任务更为合理

(3) 任务执行线程会捕获任务执行过程中的所有异常，一个定时任务的异常不会影响其他定时任务，但发生异常的任务也不再被重新调度，即使它是一个重复任务