深入浅出 Java Concurrency (32): 线程池 part 5 周期性任务调度

▪ 周期性任务调度前世

在 JDK 5.0 之前，java.util.Timer/TimerTask 是唯一的内置任务调度方法，而且在很长一段时间里很热衷于使用这种方式进行周期性任务调度。

首先研究下 Timer/TimerTask 的特性（至于 javax.swing.Timer 就不再研究了）。

```
public void schedule(TimerTask task, long delay, long period) {
    if (delay < 0)
        throw new IllegalArgumentException("Negative delay.");
    if (period <= 0)
        throw new IllegalArgumentException("Non-positive period.");
    sched(task, System.currentTimeMillis()+delay, -period);
}
public void scheduleAtFixedRate(TimerTask task, long delay, long period) {
    if (delay < 0)
        throw new IllegalArgumentException("Negative delay.");
    if (period <= 0)
        throw new IllegalArgumentException("Non-positive period.");
    sched(task, System.currentTimeMillis()+delay, period);
}
public class Timer {

    private TaskQueue queue = new TaskQueue();
    /**
     * The timer thread.
     */
    private TimerThread thread = new TimerThread(queue);
```

java.util.TimerThread.mainLoop()

```
private void mainLoop() {
    while (true) {
        try {
            TimerTask task;
            boolean taskFired;
            synchronized(queue) {
                // Wait for queue to become non-empty
```

```
            while (queue.isEmpty() && newTasksMayBeScheduled)
                queue.wait();
            if (queue.isEmpty())
                break; // Queue is empty and will forever remain; die
。。。。。。

            if (!taskFired) // Task hasn't yet fired; wait
                queue.wait(executionTime - currentTime);
          }
          if (taskFired)  // Task fired; run it, holding no locks
            task.run();
        } catch(InterruptedException e) {
        }
    }
}
```

上面三段代码反映了 Timer/TimerTask 的以下特性：

- Timer 对任务的调度是基于绝对时间的。
- 所有的 TimerTask 只有一个线程 TimerThread 来执行，因此同一时刻只有一个 TimerTask 在执行。
- 任何一个 TimerTask 的执行异常都会导致 Timer 终止所有任务。
- 由于基于绝对时间并且是单线程执行，因此在多个任务调度时，长时间执行的任务被执行后有可能导致短时间任务快速在短时间内被执行多次或者干脆丢弃多个任务。

由于 Timer/TimerTask 有这些特点（缺陷），因此这就导致了需要一个更加完善的任务调度框架来解决这些问题。

■ 周期性任务调度今生

java.util.concurrent.ScheduledExecutorService 的出现正好弥补了 Timer/TimerTask 的缺陷。

```
public interface ScheduledExecutorService extends ExecutorService {
    public ScheduledFuture<?> schedule(Runnable command,
                    long delay, TimeUnit unit);

    public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                    long delay, TimeUnit unit);
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                      long initialDelay,
                      long period,
                      TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,
```

```
                long initialDelay,
                long delay,
                TimeUnit unit);
}
```

首先 ScheduledExecutorService 基于 ExecutorService，是一个完整的线程池调度。另外在提供线程池的基础上增加了四个调度任务的 API。

- schedule(Runnable command,long delay, TimeUnit unit)：在指定的延迟时间一次性启动任务（Runnable），没有返回值。
- schedule(Callable<V> callable, long delay, TimeUnit unit)：在指定的延迟时间一次性启动任务（Callable），携带一个结果。
- scheduleAtFixedRate(Runnable command,long initialDelay,long period,TimeUnit unit)：建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期；也就是将在 initialDelay 后开始执行，然后在 initialDelay+period 后执行，接着在 initialDelay + 2 * period 后执行，依此类推。如果任务的任何一个执行遇到异常，则后续执行都会被取消。否则，只能通过执行程序的取消或终止方法来终止该任务。如果此任务的任何一个执行要花费比其周期更长的时间，则将推迟后续执行，但不会同时执行。
- scheduleWithFixedDelay(Runnable command,long initialDelay,long delay,TimeUnit unit)：创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟。如果任务的任一执行遇到异常，就会取消后续执行。否则，只能通过执行程序的取消或终止方法来终止该任务。

上述 API 解决了以下几个问题：

- ScheduledExecutorService 任务调度是基于相对时间，不管是一次性任务还是周期性任务都是相对于任务加入线程池（任务队列）的时间偏移。
- 基于线程池的 ScheduledExecutorService 允许多个线程同时执行任务，这在添加多种不同调度类型的任务是非常有用的。
- 同样基于线程池的 ScheduledExecutorService 在其中一个任务发生异常时会退出执行线程，但同时会有新的线程补充进来进行执行。
- ScheduledExecutorService 可以做到不丢失任务。

下面的例子演示了一个任务周期性调度的例子。

```
package xylz.study.concurrency.executor;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ScheduledExecutorServiceDemo {
   public static void main(String[] args) throws Exception{
      ScheduledExecutorService execService =  Executors.newScheduledThreadPool(3);
      execService.scheduleAtFixedRate(new Runnable() {
```

```
            public void run() {
                System.out.println(Thread.currentThread().getName()+" -> "+System.currentTimeMi
llis());
                try {
                    Thread.sleep(2000L);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, 1, 1, TimeUnit.SECONDS);
        //
        execService.scheduleWithFixedDelay(new Runnable() {
            public void run() {
                System.out.println(Thread.currentThread().getName()+" -> "+System.currentTimeMi
llis());
                try {
                    Thread.sleep(2000L);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }, 1, 1, TimeUnit.SECONDS);
        Thread.sleep(5000L);
        execService.shutdown();
    }
}
```

一次可能的输出如下：

```
pool-1-thread-1 -> 1294672392657
pool-1-thread-2 -> 1294672392659
pool-1-thread-1 -> 1294672394657
pool-1-thread-2 -> 1294672395659
pool-1-thread-1 -> 1294672396657
```

在这个例子中启动了默认三个线程的线程池，调度两个周期性任务。第一个任务是每隔 1 秒执行一次，第二个任务是以固定 1 秒的间隔执行，这两个任务每次执行的时间都是 2 秒。上面的输出有以下几点结论：

- 任务是在多线程中执行的，同一个任务应该是在同一个线程中执行。
- scheduleAtFixedRate 是每次相隔相同的时间执行任务，如果任务的执行时间比周期还长，那么下一个任务将立即执行。例如这里每次执行时间 2 秒，而周期时间只有 1 秒，那么每次任务开始执行的间隔时间就是 2 秒。
- scheduleWithFixedDelay 描述是下一个任务的开始时间与上一个任务的结束时间间隔相同。流入这里每次执行时间 2 秒，而周期时间是 1 秒，那么两个任务开始执行的间隔时间就是 2+1=3 秒。

事实上 ScheduledExecutorService 的实现类 ScheduledThreadPoolExecutor 是继承线程池类 ThreadPoolExecutor 的，因此它拥有线程池的全部特性。但是同时又是一种特殊的线程池，这个线程池的线程数大小不限，任务队列是基于 DelayQueue 的无限任务队列。具体的结构和算法在以后的章节中分析。

由于 ScheduledExecutorService 拥有 Timer/TimerTask 的全部特性，并且使用更简单，支持并发，而且更安全，因此没有理由继续使用 Timer/TimerTask，完全可以全部替换。需要说明的一点事构造 ScheduledExecutorService 线程池的核心线程池大小要根据任务数来定，否则可能导致资源的浪费。

上一篇：线程池 part 4 线程池任务拒绝策略 **目 录** **下一篇**： 线程池 part 6 线程池的实现及原理 (1)