基于 JDK1.5 的定时任务执行方案的改进①

Improvement of Execution Scheme for Timing Task Based on JDK1.5

任冬艳 廖建新 王 纯 (北京邮电大学网络与交换技术国家重点实验室 北京 100876)
(东信北邮信息技术有限公司 北京 100083)

摘 要: 针对 JDK1.5(Java Development Kit)在定时任务执行方案机制中存在的不足,分析其原因,结合实践过程总结的定时任务特点,对现有的定时任务执行方案加以改进,形成一种比较全面灵活的对定时任务执行的处理机制。从而扩大其适用范围,增加其灵活性,使定时任务执行更简单,任务形式更多样。

关键词: JDK 定时任务

1 引言

定时任务的执行是目前很多 java 应用都支持的操作,随着应用复杂度的提高,定时任务的形式也越来越多样化,尤其在多线程的运行环境下,定时任务的线程安全性是一个重要的考虑因素。JDK1.5(Java Development Kit)包含两个解决定时任务执行的方案,一个是 java.util.Timer 类,一个是 java.util.concurrent.ScheduledThreadPoolExecutor[1]类。Timer 类可安排任务执行一次,或者定期重复执行,该类是单线程实现方案,无法保证有大量定时任务情况下的稳定运行。ScheduledThreadPoolExecutor 类通过线程池解决了 Timer 类单线程的问题,在队列管理、多线程执行方面有了较大改进[2]。

2 存在的问题

JDK1.5 提供的两个定时任务执行方案对于仅执行一次的定时任务都提供了很好的支持,但对于周期性定时任务又都存在不足。

(1) 周期性定时任务的周期性表示抽象,不直观。

Timer 类通过调用下面三个函数实现周期性定时任务的周期执行:

表 1 Timer 类定时任务执行函数

void	schedule(TimerTask task, long delay, long period) 安排指定的任务从指定的延迟后开始进行重复的固定延迟执行。
void	scheduleAtFixedRate(TimerTask task, Date firstTime, long period) 安排指定的任务在指定的时间开始进行重复的固定速率执行。
void	scheduleAtFixedRate(TimerTask task, long delay, long period) 安排指定的任务在指定的延迟后开始进行重复的固定速率执行。

由表 1 可以看到,参数 period 是任务执行的周期间隔(以毫秒为单位),对于所有的周期性定时任务(不管周期是以秒、小时还是天为单位)都需要用户自己计算任务的周期间隔,如 1 小时是 60 * 60 * 1000 毫秒,1 天是 24 * 60 * 60 * 1000 毫秒等,对于定义任务的用户来说不仅容易出错,而且容易混淆,而其他用户则需要通过计算才能明白任务的周期性。也即任务的周期性比较抽象,不直观。同样的,ScheduledThreadPoolExecutor 类调用以下两个方法执行周期性定时任务:

① 基金项目:基金项目:国家杰出青年科学基金(No. 60525110);国家 973 计划项目(No. 2007CB307100, 2007CB307103);新世纪优秀人才支持计划(No. NCET-04-0111);电子信息产业发展基金项目(基于 3G 的移动业务应用系统);电子信息产业发展基金重点项目(下一代网络核心业务平台)。

表 2　ScheduledThreadPoolExecutor 类定时任务执行函数

Scheduled Future <? >	scheduleAtFixedRate (Runnable command, long initialDelay, long period, 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,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。
Scheduled Future <? >	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。

从表 2 的方法描述中我们可以发现定时任务的周期性也存在抽象不直观的问题。

(2) 不保存定时任务的执行结果,不便于对定时任务的运行状况进行监控和统计。

不管是周期性定时任务还是非周期性定时任务,从表 1 和表 2 可以看到任务执行函数返回的结果一个是 void(无值),一个是 ScheduledFuture(只保存了任务的执行时间和周期)。任务执行的结果没有保存,在一些需要提取任务执行结果以便进行下一步操作的应用中无疑让用户感到遗憾。

(3) 定时任务执行开始时间的相对性和绝对性不直观。

在有些应用中,任务执行时间的相对性和绝对性要求比较明确,虽然在现有的方案中用户可以通过自己计算时间差来确定时间的相对性和绝对性,但是如果任务执行类本身可以提供这个功能,那么用户会感觉很方便。

(4) 周期性定时任务每次执行的时间起点确定比较模糊。

某些时候,定时任务用户希望任务下次开始执行的时间起点是从上次任务执行结束的时间算起,而其他时候是从上次任务执行开始的时间算起。表 1 的第一个函数和表 2 的第二个函数虽然提供了这个功能,但是读者很容易发现两个方案都是通过用"delay"(延迟) 和"period"(周期) 这两个长整型参数代表的含义的不同来区分执行时间起点的。这种方法不仅晦涩难懂,不容易让用户掌握,而且容易出错,产生用户不期望的结果。

针对上述提出的几个问题,我们对 JDK1. 5 的 ScheduledThreadPoolExecutor 类进行了改进。

3　解决方案

J2SE5. 0 (Java 2 Platform, Standard Edition) 提供了枚举类型(enum) 和泛型(Generic), 在改进的 ScheduledThreadPoolExecutor 类中充分利用这两种新特性解决上述提出的几个问题。

(1) 周期性任务的迭代周期可以分为八种情况:按年迭代、按月迭代、按星期迭代、按日迭代、按小时迭代、按分钟迭代、按秒迭代、按毫秒迭代。其中以年和月为迭代周期的运行方式在已有 JDK 中尚未支持,由于小于毫秒级别的时间控制与 JVM(Java Virtual Machine) 及平台和操作系统有很大关系,无法保证各平台时间精度的一致性,因此改进类只提供最小为毫秒级的迭代。任务的周期性由周期类型(PERIODTYPE) 和周期(period) 两个参数决定,通过在调用 ScheduledThreadPoolExecutor 类的任务执行方法时显式指定参数,用户不需要进行烦琐的计算,计算过程交给 ScheduledThreadPoolExecutor 实现。

(2) 改进的 ScheduledThreadPoolExecutor 类通过定义内部类 ExecuteInfo < V > 来记录任务执行的开始时间、结束时间和执行结果(V)。并通过重写原类的 ScheduledFutureTask 内部类,使 ScheduledFutureTask 成为任务运行控制核心类,所有任务的运行都在此类的控制下进行,周期性任务由此类负责计算下次触发时间,并保存各种运行信息。而原类的 ScheduledFutureTask 内部类只保存了任务执行的时间和周期。

(3) 任务运行方式可分为四种:以相对时间为任务起始时间,按照一定时间间隔不间断执行;以相对时间为任务起始时间,只执行一次;以绝对时间为任务起始时间,按照一定时间间隔不间断执行;以绝对时间为任务起始时间,只执行一次。改进的 ScheduledThreadPoolExecutor 类用枚举类型定义了这四种执行方式,不同执行方式通过调用不同的函数来体现。

(4) 任务执行时间起点的类型分两种情况:从上次任务执行开始时间为起点;从上次任务执行结束时间为起点。通过显式定义不同类型并在函数调用时指定参数,用户无需再为"delay"和"period"的模糊区别而烦恼。

此外,改进类的构造函数与 JDK1. 5 的原类比较在

时间粒度上由纳秒改为了毫秒，这是基于上述提到的小于毫秒级的时间控制与 JVM 及平台和操作系统有关而改进的。改进类还通过 schedule 函数的修改提高类执行任务的能力。最后，改进类实现了线程安全的单态模式，对多线程的系统应用提供了很好的支持。

4 设计实现

4.1 任务属性

在改进的 ScheduledThreadPoolExecutor 类中定义了三个新增的任务属性：分别是任务执行类型、周期类型、执行时间起点类型，每个值都可以顾名思义，如 PERIODTYPE 的 YEAR 即任务周期类型的按年迭代、RUNTYPE 的 FROMSTARTTIME 即执行时间起点类型的从上次任务执行开始时间为起点计算下次任务执行时间。各个类型定义如下：

```
public enum EXECTYPE { TIMEROLL, TIMEONCE, DATEROLL, DATEONCE };
public enum PERIODTYPE { YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND };
public enum RUNTYPE { FROMSTARTTIME, FROMRUNTIME };
```

4.2 ScheduledFutureTask 内部类

改进的 ScheduledThreadPoolExecutor 类中的 ScheduledFutureTask 内部类实现了 Runnable，和 ScheduledFuture <V> 接口。该内部类不仅定义了任务的各种属性以及任务执行属性（如周期类型、开始执行时间、下次执行时间、执行结果队列等），还定义了 Sync 内部类。Sync 类是作为 ScheduledFutureTask 类的同步控制子类，继承 AbstractQueuedSynchronizer 类，并使用这个类的同步功能。Sync 类作为 ScheduledFutureTask 的非静态子类，是为了可以调用 ScheduledFutureTask 类的 done 函数以便在任务完成时可以通知父类记录相关的信息。Sync 类的实现和 java.util.concurrent 包中 FutureTask 类的 Sync 内部类实现类似。我们可以调用构造函数

```
public ScheduledFutureTask(String name, Callable <V> callable, Calendar startDate, PERIODTYPE periodType, long period,RUNTYPE runtype)
```

来构造一个 ScheduledFutureTask 对象，其中各个参数定义为：name（任务名称）、callable（任务，必须实现 Callable 接口）、startDate（任务开始执行时间）、peirodType（周期类型）、period（周期）、runtype（任务执行起点类型）。其他构造函数的参数定义类似。

JDK1.5 的原 ScheduledThreadPoolExecutor 类中 ScheduledFutureTask 内部类继承 FutureTask 类并实现了 ScheduledFuture <V> 接口。

4.3 ScheduledFutureTask 内部类

```
public static class ExecuteInfo <V>
{ private Calendar startTime; //开始时间
  private Calendar endTime; //结束时间
  private V result; //运行结果
  private Throwable exception; //运行时异常
  public Calendar getEndTime( ) {
    return endTime;}
  public Throwable getException( ) {
    return exception; }
  public V getResult( ) {
    return result; }
  public Calendar getStartTime( ) {
    return startTime;}
}
```

该类记录了任务执行的信息和执行结果，其中 V 是 JDK1.5 的泛型（Generic）定义，V 可以是任何继承了 Object 类的子类。

4.4 构造函数

原类的构造函数如下：

```
ProtectedScheduledThreadPoolExecutor ( int corePoolSize)
{ super ( corePoolSize, Integer. MAX_VALUE, 0, TimeUnit. MILLISECONDS, new DelayedWorkQueue( ) );
}
```

改进类的构造函数如下：

```
protected ScheduledThreadPoolExecutor ( int corePoolSize)
{ super ( corePoolSize, Integer. MAX_VALUE, 0, TimeUnit. NANOSECONDS, new DelayedWorkQueue ( ) ); }
```

可以发现，除了 TimeUnit 的粒度变化外，其他参数基本相同。其他三个构造函数也是类似情形。

4.5 schedule 函数

JDK1.5 的原 ScheduledThreadPoolExecutor 类提供了两个 schedule 函数，函数定义如下：

表 3 原 ScheduledThreadPoolExecutor 的 schedule 函数

<V > Scheduled Future <V >	schedule(Callable <V > callable, long delay, TimeUnit unit) 创建并执行在给定延迟后启用的 Scheduled-Future。
Scheduled Future <? >	schedule (Runnable command, long delay, TimeUnit unit) 创建并执行在给定延迟后启用的一次性操作

而改进类中定义了四个函数:

表 4 改进 ScheduledThreadPoolExecutor 类的 schedule 函数

<V > Scheduled FutureTask <V >	schedule(String name, Callable <V > callable, long delay, PERIODTYPE periodType, int period, RUNTYPE runtype)
<V > Scheduled FutureTask <V >	schedule(String name, Callable <V > callable, long delay)
<V > Scheduled FutureTask <V >	schedule(String name, Callable <V > callable, Calendar startTime, PERIODTYPE periodType, int period, RUNTYPE runtype)
<V > Scheduled FutureTask <V >	schedule(String name, Callable <V > callable, Calendar startTime)

从表中可以看出不仅函数的形式参数发生了变化,函数的返回值也发生了变化。返回值是一个包含了任务执行结果信息的对象,从而可以对定时任务的执行过程进行监控和统计。

5 案例

一个定时任务 task(task 的实现根据不同的执行方式要继承或者实现不同的基类或接口,但是核心方法一样),执行时间从 2008 年 8 月 8 号 8 时开始,任务周期为 2 小时,任务下次运行时间计算方式是按照每次执行结束后的时间为起点开始计算,用 Timer 类的方式执行,调用过程为:

```
Calendar cal = Calendar. getInstance( ) ;
cal. set( 2008 ,7 ,8 ,8 ,0) ;
long delay = cal. getTimeInMillis( ) ;
long period = 2 * 60 * 60 * 1000;
Timer timer = new Timer( ) ;
timer. schedule( task, delay, period)
```

使用 JDK1. 5 的 Executor 方法,调用过程为:

```
Calendar cal = Calendar. getInstance( ) ;
cal. set( 2008 ,7 ,8 ,8 ,0) ;
long delay = cal. getTimeInMillis( ) ;
long period = 2 * 60 * 60 * 1000
ScheduledThreadPoolExecutor executor =
new ScheduledThreadPoolExecutor( 5) ; executor.
scheduleWithFixedDelay( task, delay, period, TimeUnit.
MILLISECONDS) ;
```

可以看到,代码量是一样的,在执行效率上,ScheduledThreadPoolExecutor 的效率要高一些,因为它是基于线程池的方式,尤其在多任务的时候,效率会有显著不同。在新改进的 ScheduledThreadPoolExecutor 类下的调用过程为:

```
Calendar cal = Calendar. getInstance( ) ;
cal. set( 2008 ,7 ,8 ,8 ,0) ;
ScheduledThreadPoolExecutor executor = Sched-
uledThreadPoolExecutor. getInstance( ) ;
executor. schedule( " task" ,task, cal, PERIODTYPE.
HOUR,2, RUNTYPE. FROMENDTIME) ;
```

很明显,在没有改变执行时间的前提下,在新改进类的调用过程中,用户需要写的代码更少,而且调用参数一目了然,不容易引起歧义

6 总结

本方案在原 JDK1. 5 提供的定时任务执行方案下进行改进,实现了定时任务的多样化和执行的简单化,方便用户理解和使用。改进方案在很多应用中都完美地完成了用户提出的各种定时任务需求,并且运行稳定。对于需要执行定时任务的应用不仅可以减少开发量,而且避免了出错的可能性,可以给开发人员提供更大的灵活性,使系统更稳定和更易扩展。

参考文献

1 JavaTM 2 Platform Standard Edition 5. 0 API. 2004. 09.

2 Scott Oaks & Henry Wong. JAVA THREADS, 3E. O' Reilly, 2006. 3.