Quartz Scheduler misfireThreshold属性的意义与触发器超时后的处理策略

Quartz misfireThreshold属性的意义与触发器超时后的处理策略。

在配置quartz.properties有这么一个属性就是misfireThreshold，用来指定调度引擎设置触发器超时的"临界值"。

要弄清楚触发器超时临界值的意义，那么就要先弄清楚什么是触发器超时？打个比方：比如调度引擎中有5个线程，然后在某天的下午2点 有6个任务需要执行，那么由于调度引擎中只有5个线程，所以在2点的时候会有5个任务会按照之前设定的时间正常执行，有1个任务因为没有线程资源而被延迟执行，这个就叫触发器超时。

那么超时的时间又是如何计算的呢？还接着上面的例子说，比如一个(任务A)应该在2点的时候执行，但是在2点的时候调度引擎中的可用线程都在忙碌状态中,或者调度引擎挂了，这都有可能发生，然后再2点05秒的时候恢复（有可用线程或者服务器重新启动），也就是说（任务A）应该在2点执行 但现在晚了5秒钟。那么这5秒钟就是任务超时时间，或者叫做触发器(Trigger)超时时间。

理解了上面的内容再来看misfireThreshold值的意义，misfireThreshold是用来设置调度引擎对触发器超时的忍耐时间，简单来说 假设misfireThreshold=6000(单位毫秒)。

那么它的意思说当一个触发器超时时间如果大于misfireThreshold的值 就认为这个触发器真正的超时(也叫Misfires)。

如果一个触发器超时时间 小于misfireThreshold的值， 那么调度引擎则不认为触发器超时。也就是说调度引擎可以忍受这个超时的时间。

还是 任务A 比它应该正常的执行时间晚了5秒 那么misfireThreshold的值是6秒，那么调度引擎认为这个延迟时间可以忍受，所以不算超时(Misfires)，那么引擎会按照正常情况执行该任务。 但是如果 任务A 比它应该正常的执行时间晚了7秒 或者是6.5秒 只要大于misfireThreshold 那么调度引擎就会认为这个任务的触发器超时。

这样的话就会出现这么情况，让我们一个一个说明，并给出例子。

第一种情况:任务一切正常，即按照我们定义的触发器的预期时间执行，比如下午2点运行 时间间隔3秒 重复运行 5次等，这个没啥好说的。

第二种情况：任务出现延时，延时的时间<misfireThreshold。比如一个任务正常应该在2点运行，但是调度系统忙碌2点5秒才得空运行这个任务，这样这个任务就被耽误了5秒钟。

假设这个任务的触发器定义的是 2点执行 时间间隔为1秒 执行10次。如果正常情况 任务应该在 2点00秒 ，2点01秒，2点03秒....2点10秒触发。但这个任务在2点05秒的时候引擎在后空去执行它，这样的话就比我们预期的时间慢了5秒。那么调度引擎是如何执行这个任务的呢？不是是应该在在2点05秒开始执行 然后一直到2点15秒呢？经过测试我发现并不是这样的，而是调度引擎直接把慢了的那5次立即运行，然后再每隔1秒运行5次。

DEMO：

```
1 org.quartz.threadPool.threadCount = 1
2 org.quartz.threadPool. class : org.quartz.simpl.SimpleThreadPool
3 org.quartz.threadPool.threadPriority: 5
4 org.quartz.jobStore. class = org.quartz.simpl.RAMJobStore
5 org.quartz.jobStore.misfireThreshold = 5000
```

为了测试出效果，我们将threadCount调度引擎中的线程数设置为1，misfireThreshold超时忍受时间设置为5秒

任务一: 假设执行时间4秒

```
public void execute(JobExecutionContext context)
    throws JobExecutionException {
    System.out.println(context.getJobDetail().getKey() + new Date().toLocaleString());
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

任务二: 打印当前时间

```
public void execute(JobExecutionContext context)
    throws JobExecutionException {
    System.out.println(context.getJobDetail().getKey() + new Date().toLocaleString());
}
```

测试代码:

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();

JobDetail job = newJob(StatefulDumbJob.class)
    .withIdentity("statefulJob1", "group1")
    .build();

SimpleTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .withSchedule(simpleSchedule())
    .build();

JobDetail job2 = newJob(StatefulDumbJob2.class)
        .withIdentity("statefulJob2", "group1")
        .build();

SimpleTrigger trigger2 = newTrigger()
        .withIdentity("trigger2", "group1")
        .startNow()
        .endAt(futureDate(10, IntervalUnit.SECOND))
        .withSchedule(simpleSchedule()
                    .withIntervalInSeconds(1)
                    .withRepeatCount(10)
                )
        .build();

sched.scheduleJob(job2, trigger2);
sched.start();
```

分别在调度引擎中添加2个任务 job 和 job2， 俩个任务都是立即执行

由于job的执行时间需要4秒并且调度引擎中的可用线程只有一个，这就会导致job2延迟触发。我们观察一下控制台输出的结果。

```
1  group1.statefulJob12014-8-19 11:44:28
2  group1.statefulJob22014-8-19 11:44:32
3  group1.statefulJob22014-8-19 11:44:32
4  group1.statefulJob22014-8-19 11:44:32
5  group1.statefulJob22014-8-19 11:44:32
6  group1.statefulJob22014-8-19 11:44:32
7  group1.statefulJob22014-8-19 11:44:33
8  group1.statefulJob22014-8-19 11:44:34
9  group1.statefulJob22014-8-19 11:44:35
10 group1.statefulJob22014-8-19 11:44:36
```

可以看到job先执行在11:44:28的时候，然后4秒钟以后执行JOB2，调度引擎会立即执行(job2) 5次，然后再每隔一秒执行一次，直到执行完定义的次数。

第三种情况：任务触发器超时，延迟的时间>=misfireThreshold，那么调度引擎该如何处理这个任务呢？

答案是这样的：在定义一个任务的触发器的时候，我们可以设置它超时的处理策略，调度引擎会根据我们设置的策略来处理这个任务。

我们在定义一个任务的触发器时 最常用的就是俩种触发器：1、SimpleTrigger 2、CronTrigger。

1、SimpleTrigger 默认的策略是 Trigger.MISFIRE_INSTRUCTION_SMART_POLICY 官方的解释如下:

```
Instructs the Scheduler that upon a mis-fire situation, the updateAfterMisfire() method will be called on the Trigger to
determine the mis-fire instruction, which logic will be trigger-implementation-dependent.
```

大意是:指示调度引擎在MisFire的状态下，会去调用触发器的updateAfterMisfire的方法来确定它的超时处理策略，里面的逻辑取决于具体的实现类。

那我们在看一下updateAfterMisfire方法的说明:

```
If the misfire instruction is set to MISFIRE_INSTRUCTION_SMART_POLICY, then the following scheme will be used:
If the Repeat Count is 0, then the instruction will be interpreted as MISFIRE_INSTRUCTION_FIRE_NOW.
If the Repeat Count is REPEAT_INDEFINITELY, then the instruction will be interpreted as
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT. WARNING: using
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT with a trigger that has a non-null end-time may cause the trigger
to never fire again if the end-time arrived during the misfire time span.
If the Repeat Count is > 0, then the instruction will be interpreted as
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT.
```

大意是:

1、如果触发器的重复执行数(Repeat Count)等于0，那么会按这个(MISFIRE_INSTRUCTION_FIRE_NOW)策略执行。

2、如果触发器的重复执行次数是 SimpleTrigger.REPEAT_INDEFINITELY (常量值为-1，意思是重复无限次)，那么会按照 MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT策略执行。

3、如果触发器的重复执行次数大于0，那么按照 MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT执行。

既然是这样，那就让我们依次看一下每种处理策略都是啥意思！

1、MISFIRE_INSTRUCTION_FIRE_NOW

Instructs the `Scheduler` that upon a mis-fire situation, the `SimpleTrigger` wants to be fired now by `Scheduler`.

NOTE: This instruction should typically only be used for 'one-shot' (non-repeating) Triggers. If it is used on a trigger with a repeat count > 0 then it is equivalent to the instruction `MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT`.

指示调度引擎在MisFire的情况下，将任务(JOB)马上执行一次。

需要注意的是 这个指令通常被用做只执行一次的Triggers，也就是没有重复的情况（non-repeating），如果这个Triggers的被安排的执行次数大于0

那么这个执行与 (4) `MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT 相同`

`2、` MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT

```
1 Instructs the Scheduler that upon a mis-fire situation, the
SimpleTrigger wants to be re-scheduled to 'now' with the repeat
2 count left as-is. This does obey the Trigger end-time however, so if
'now' is after the end-time the Trigger will not fire again.
3
NOTE: Use of this instruction causes the trigger to 'forget' the
start-time and repeat-count that it was originally setup with ( this
is only an issue if you for some reason wanted to be able to tell
what the original values were at some later time).
```

指示调度引擎重新调度该任务，repeat count保持不变，并且服从trigger定义时的endTime,如果现在的时间，如果当前时间已经晚于 end-time，那么这个触发器将不会在被激发。

注意：这个状态会导致触发器忘记最初设置的 start-time 和 repeat-count，为什么这个说呢，看源代码片段:updateAfterMisfire方法中

```
else if (instr == MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT) {
Date newFireTime = new Date();
if (repeatCount != 0 && repeatCount != REPEAT_INDEFINITELY) {
setRepeatCount(getRepeatCount() - getTimesTriggered());
setTimesTriggered( 0 );
}

if (getEndTime() != null && getEndTime().before(newFireTime)) {
setNextFireTime( null );  // We are past the end time
} else {
setStartTime(newFireTime);
setNextFireTime(newFireTime);
}
```

getTimesTriggered的是获取这个触发器已经被触发了多少次，那么用原来的次数 减掉 已经触发的次数就是还要触发多少次

接下来就是判断一下触发器是否到了结束时间，如果到了的话，触发器就不会在被触发。

然后就是重新设置触发器的开始实现是 “现在” 并且立即运行。

3、MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT

```
Instructs the Scheduler that upon a mis-fire situation, the SimpleTrigger wants to be re-scheduled to the next scheduled time after 'now' - taking into account any associated Calendar, and with the repeat count set to what it would be, if it had not missed any firings.

NOTE/WARNING: This instruction could cause the Trigger to go directly to the 'COMPLETE' state if all fire-times where missed.
```

这个策略跟上面的2策略一样，唯一的区别就是设置触发器的时间 不是“现在” 而是下一个 scheduled time。解释起来比较费劲，举个例子就能说清楚了。

比如一个触发器设置的时间是 10:00 执行 时间间隔10秒 重复10次。那么当10：07秒的时候调度引擎可以执行这个触发器的任务。那么如果是策略（2），那么任务会立即运行。

那么触发器的触发时间就变成了 10:07 10:17 10:27 10:37 .....

那么如果是策略（3），那么触发器会被安排在下一个scheduled time。 也就是10：20触发。 然后10:30 10:40 10:50。这回知道啥意思了吧。

4、MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT

这个策略跟上面的策略（2）比较类似，指示调度引擎重新调度该任务，repeat count 是剩余应该执行的次数，也就是说本来这个任务应该执行10次，但是已经错过了3次，那么这个任务就还会执行7次。

下面是这个策略的源码，主要看红色的地方就能看到与策略(2)的区别，这个任务的repeat count 已经减掉了错过的次数。

```
} else if (instr == MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT) {
Date newFireTime = new Date();
int timesMissed = computeNumTimesFiredBetween(nextFireTime, newFireTime);
if (repeatCount != 0 && repeatCount != REPEAT_INDEFINITELY) {
int remainingCount = getRepeatCount() - (getTimesTriggered() + timesMissed);
if (remainingCount <= 0 ) {
remainingCount = 0 ;
}
setRepeatCount(remainingCount);
setTimesTriggered( 0 );
}
if (getEndTime() != null && getEndTime().before(newFireTime)) {
setNextFireTime( null ); // We are past the end time
} else {
setStartTime(newFireTime);
setNextFireTime(newFireTime);
}
}
```

5、MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT

```
Instructs the Scheduler that upon a mis-fire situation, the SimpleTrigger wants to be re-scheduled to the next scheduled time after 'now' - taking into account any associated Calendar, and with the repeat count set to what it would be, if it had not missed any firings.

NOTE/WARNING: This instruction could cause the Trigger to go directly to the 'COMPLETE' state if all fire-times where missed.
```

这个策略与上面的策略3比较类似，区别就是repeat count 是剩余应该执行的次数而不是全部的执行次数。比如一个任务应该在2:00执行，repeat count=5，时间间隔5秒， 但是在2:07才获得执行的机会，那任务不会立即执行，而是按照机会在2点10秒执行。

6、MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY

```
Instructs the Scheduler that the Trigger will never be evaluated for a misfire situation, and that the scheduler will simply try to fire it as soon as it can, and then update the Trigger as if it had fired at the proper time.

NOTE: if a trigger uses this instruction, and it has missed several of its scheduled firings, then several rapid firings may occur as the trigger attempt to catch back up to where it would have been. For example, a SimpleTrigger that fires every 15 seconds which has misfired for 5 minutes will fire 20 times once it gets the chance to fire.
```

这个策略是忽略所有的超时状态，和最上面讲到的 (第二种情况) 一致。

举个例子，一个SimpleTrigger 每个15秒钟触发, 但是超时了5分钟才获得执行的机会，那么这个触发器会被快速连续调用20次， 追上前面落下的执行次数。

2、CronTrigger 的默认策略也是Trigger.MISFIRE_INSTRUCTION_SMART_POLICY 官方解释如下,也就是说不指定的话默认为:MISFIRE_INSTRUCTION_FIRE_ONCE_NOW。

```
Updates the CronTrigger's state based on the MISFIRE_INSTRUCTION_XXX that was selected when the CronTrigger was created.

If the misfire instruction is set to MISFIRE_INSTRUCTION_SMART_POLICY, then the following scheme will be used:

The instruction will be interpreted as MISFIRE_INSTRUCTION_FIRE_ONCE_NOW
```

1、MISFIRE_INSTRUCTION_FIRE_ONCE_NOW

```
Instructs the Scheduler that upon a mis-fire situation, the CronTrigger wants to be fired now by Scheduler.
```

这个策略指示触发器超时后会被立即安排执行，看源码，红色标记的地方。也就是说不管这个触发器是否超过结束时间(endTime) 首选执行一次，然后就按照正常的计划执行。

```
@Override
public void updateAfterMisfire(org.quartz.Calendar cal) {
int instr = getMisfireInstruction();
if (instr == Trigger.MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY)
return ;
if (instr == MISFIRE_INSTRUCTION_SMART_POLICY) {
instr = MISFIRE_INSTRUCTION_FIRE_ONCE_NOW;
}
if (instr == MISFIRE_INSTRUCTION_DO_NOTHING) {
Date newFireTime = getFireTimeAfter( new Date());
while (newFireTime != null && cal != null
&& !cal.isTimeIncluded(newFireTime.getTime())) {
newFireTime = getFireTimeAfter(newFireTime);
}
setNextFireTime(newFireTime);
} else if (instr == MISFIRE_INSTRUCTION_FIRE_ONCE_NOW) {
setNextFireTime( new Date());
}
}
```

2、MISFIRE_INSTRUCTION_DO_NOTHING

这个策略与策略(1)正好相反，它不会被立即触发，而是获取下一个被触发的时间，并且如果下一个被触发的时间超出了end-time 那么触发器就不会被执行。

上面绿色标记的地方是源码

补充几个方法的说明:

1、getFireTimeAfter 返回触发器下一次将要触发的时间，如果在给定（参数）的时间之后，触发器不会在被触发，那么返回null。

```
Date getFireTimeAfter(Date afterTime)

Returns the next time at which the Trigger will fire, after the given time. If the trigger will not fire after the given time, null will be returned.
```

2、isTimeIncluded 判断给定的时间是否包含在quartz的日历当中，因为quartz是可以自定义日历的，设置哪些日子是节假日什么的。

```
boolean isTimeIncluded(long timeStamp)

Determine whether the given time (in milliseconds) is 'included' by the Calendar.
```