Quart2.2.1开发指南 -中文版

翻译　:chen
QQ　: 1185280999
主页　:github.com/chen1185280999
版本　:2.2.1

一.使用Quartz API

1.scheduler(调度器)的实例化
2.关键接口
3.任务和触发器

1.scheduler(调度器)的例子

你需要使用SchedulerFactory来实例化scheduler.

可以选择JNDI工厂的方式,或者直接使用工厂方法实例化(例如下面的方式).

一旦调度器被实例化,他就开始,关机或置于等待的模式.在你没有重新启动这个调度器的时候,这个触发器不会触发,job也不会执行.直到调度器启动,他们也会触发,而调度器处于暂停状态.

下面的代码是实例化,并启动调度器:

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

2.关键接口

Quartz API的关键接口:

Scheduler - 与scheduler交互的主要API,调度器

Job - 你想要的scheduler去实现组件接口

JobDetail - 实现job接口

Trigger - 一个在指定时间执行Job的组件

JobBuilder - 用来定义/构建 jobDetail 实例,也定义了Jobs实例.

TriggerBuilder - 用来定义/构建Trigger实例.

一个Scheduler的周期从它创建开始就是有限的,从SchedulerFactoryk开始到调用shutdown()结束.

当Scheduler实例化后,他能添加,删除,列举jobs和triggers,并且执行其他调度相关工作(例如暂停调度).然而,直到scheduler调用了start()方法,才会作用于任何的triggers和jobs.(见上页例子).

Quartz使用了"builder"特定的语言,例如下面:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

使用JobBuilder class中的方法构建job,使用TriggerBuilder class方法构建trigger-以及从SimpleScheduleBulder 类.

```
import static org.quartz.JobBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.DateBuilder.*;
```

不同的ScheduleBuilder定义不同的schedules,

3.jobs and triggers

一个job(任务)是一个实现Job接口的类,如下所示,该接口中间有一个简单的方法:

```
package org.quartz;
public interface Job {
  public void execute(JobExecutionContext context)
    throws JobExecutionException;
}
```

但一个job的trigger触发的时候,这个execute()方法就会在scheduler的工作线程中调用.通过JobExecutionContext对象将上下文传递到方法内,上下文包括:运行时的环境,调度器的实例,操作trigger方法,jobDetail对象,等等其他的对象.

JobDetail对象作用是创建一个Quartz job(你的项目),并把这个job加到scheduler. job的jobDetail包含很多属性,作为JobDataMap ,可以用来储存你的Job的实例.JobDetail本质上是描述job的实例.

Trigger对象用来触发(使运行)Jobs.当你想要计划一个job,你实例化一个trigger并且提供你的计划.

Triggers 也可能有一个JobDataMap关联.一个JobDataMap作为一个很有用的参数传递到job中去,准确的触发trigger.在Quartz中有很多的trigger的类型,但采用的 就是simpleTrigger和CronTrigger.

●SimpleTrigger 在你只想要执行一次是实用的(在给定时刻,当次执行).或在一个时间段内执行N次,使用T的延迟

●CronTrigger 在你希望基于日期触发是实用的,例如 每个星期五中午.

为什么要存在两者jobs和triggers?许多的作业调度系统没有单独的jobs和triggers的概念.一些定义的"job"就是单纯的在某个时间执行某个任务.其他很多就结合Quartz中的Job,Trigger对象.Quartz被设计分离scheduler(调度器)和work在创建计划的时候,这个设计有很多好处.

例如:For example, you can create jobs and store them in the job scheduler independent of a trigger. This enables you to associate many triggers with the same job. Another benefit of this loose-coupling is the ability to configure jobs that remain in the scheduler after their associated triggers have expired. This enables you to reschedule them later, without having to re-define them. It also allows you to modify or replace a trigger without having to re-define its associated job.

identities(身份)

Jobs和triggers在他们注册在Quartz的scheduler(调度器)时会生成一个identities(身份).Jobs和triggers的keys(JobKey 和 TriggerKey)允许他们加到一个groups(组)中,这样是有好处的,对于"报告工作"和"维护工作".Job和trigger的名字必须在group中唯一.就是说,group中完整的key(或者标识码)是由job和trigger组成的名字.

二.Jobs和JobDetails介绍

Jobs and JobDetails

job实现很简单,接口中仅有一个execute(),你仅仅需要了解一点jobs的本质,Job接口的execute(),以及关于JobDetails.

While a job class that you implement has the code that knows how do the actual work of the particular type of job, Quartz needs to be informed about various attributes that you may wish an instance of that job to have. This is done via the JobDetail class, which was mentioned briefly in the previous section. (大概讲的是使用JobDetail来实例化job)

JobDetail的实例通过JobBuilder来构建,你通常会想要使用static import(静态导入)所以的方法,为了让你的代码有DSL-feel.

```
import static org.quartz.JobBuilder.*;
```

下面代码定义了一个job,并执行:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .build();
// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

HelloJob类:

```
public class HelloJob implements Job {
  public HelloJob() {
  }
  public void execute(JobExecutionContext context)
    throws JobExecutionException
  {
    System.err.println("Hello!  HelloJob is executing.");
  }
}
```

注意我们给scheduler(调度器)一个JobDetail实例,这个实例包含了简单的job.class.每次这个调度器执行job的时候,都会实例化一个class(job.class)并调用其execute()方法.但执行完成后,调用的实例会被销毁,并被垃圾回收.

一种情况是job必须使用无参的构造函数(使用默认的JobFactory).另外一种情况是没有意义的.

你使用JobDataMap的话,可以提供一个对实例进行性能/配置的状态,这是JobDetail.

JobDataMap

这JobDataMap可用于任何你的job实例方法的数据对象(可序列化).JobDataMap是实现了java Map接口,并且补充了一些方便的方法去存取原始的数据.

下面代码是在定义/建立 JobDetail的时候(在添加添加调度器之前),将一些数据放在JobDataMap中:

```
// define the job and tie it to our DumbJob class
JobDetail job = newJob(DumbJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .usingJobData("jobSays", "Hello World!")
    .usingJobData("myFloatValue", 3.141f)
    .build();
```

下面的是在Job的实现类中使用JobDataMap的例子:

```
public class DumbJob implements Job {
    public DumbJob() {
    }
    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();
      JobDataMap dataMap = context.getJobDetail().getJobDataMap();
      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
      System.err.println("Instance " + key + " of DumbJob says: "
          + jobSays + ", and val is: " + myFloatValue);
    }
  }
```

如果你想要使用一个持久化的JobStore(在本教程后面会提到),你应该注意的JobDataMap放的是什么,因为在他里面的对象是序列化,他们会产生class-versioning问题.明显的,标准的java类型是非常安全的.但是,任何时候,有人想要改变你已经序列化的类的时候,注意不要破坏兼容性.或者我们能将JDBC-JobStore和JobDataMap放在一个只允许原始类型和string类型的map中,从而消除序列化的问题.

如果你想要添加setter方法在job类中对应的键名在JobDataMap(一个setJobSays(String val)方法在上面例子中的数据),这个Quartz默认的JobFactory的实现的实例会主动的调用setter方法,因此阻止了在你的实现方法中要明确的指出获取某个值.

Triggers也可以和JobDataMaps联系起来,这在你有一个job存储在对应多个定期/重复trigger的调度器情况下很有用,并且job对应的每个不同的trigger要使用不同的数据输入.

JobDataMap在job执行过程中的JobExecutioncontext使用很方便,jobDataMap是Jobdetail和trigger的融合,并且值为后者覆盖任何相同名字的前者.(就是map中的覆盖吗?)

下面是JobExecutionContext融合jobDataMap的例子:

```
public class DumbJob implements Job {
    public DumbJob() {
    }
    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();
      JobDataMap dataMap = context.getMergedJobDataMap();
      // Note the difference from the previous example
      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
      ArrayList state = (ArrayList)dataMap.get("myStateData");
      state.add(new Date());
      System.err.println("Instance " + key + " of DumbJob says: " + jobSays
          + ", and val is: " + myFloatValue);
    }
  }
```

如果你想使用JobFactory注入数据映射到你的类,你就应该看下面的例子:

```
public class DumbJob implements Job {
    String jobSays;
    float myFloatValue;
    ArrayList state;
    public DumbJob() {
    }
    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();
      JobDataMap dataMap = context.getMergedJobDataMap();
      // Note the difference from the previous example
      state.add(new Date());
      System.err.println("Instance " + key + " of DumbJob says: "
         + jobSays + ", and val is: " + myFloatValue);
    }
    public void setJobSays(String jobSays) {
      this.jobSays = jobSays;
    }
    public void setMyFloatValue(float myFloatValue) {
      myFloatValue = myFloatValue;
    }
    public void setState(ArrayList state) {
      state = state;
    }
  }
```

你会发现上面的类很长,但是execute()内的代码整洁的,他实际上是减少了编码,使用IDE生产setter(),不用手工生成取JobDataMap中的值的方法,这是你的选择.

job Instances(job 实例)

你可以创建单个job类,并通过创建多个JobDetail实例在调度器中存储他的"实例定义",每个都有他自己的属性和JobDataMap,并且把他们加入到调度器中.

For example, you can create a class that implements the Job interface called SalesReportJob. The job might be coded to expect parameters sent to it (via the JobDataMap) to

specify the name of the sales person that the sales report should be based on. They may then create multiple definitions (JobDetails) of the job, such as SalesReportForJoe and SalesReportForMike which have “joe” and “mike” specified in the corresponding JobDataMaps as input to the respective jobs.

当一个trigger触发的时候,与他相关联的job是通过配置在调度器JobFactory来实例化的.默认的JobFactory是简单的调用job类的newInstance(),然后尝试通过类中的setter方法在JobDataMap进行匹配.你可能想要创建自己的JobFactory来实现自己的功能,像让自己的应用程序的IOC或者DI容器制造/初始化这个job实例.

每个JobDetail是一个job的定义或JobDetail的实例,并且每个运行的job是是一个job的实例或job定义的实例.一般的,当"job"被使用,他指的是一个名称定义或JobDetail.实现job接口的类叫"job class".

job State and Concurrency(job 状态和并发)

以下是关于job的状态数据(就是JobDataMap)和Concurrency(并发).这里有两个注解,你加到你的job类中,会让Quartz的行为不一样

@DisallowConcurrentExecution(不允许同时运行多任务) is an annotation that can be added to the Job class that tells Quartz not to execute multiple instances of a given job definition (that refers to the given job class) concurrently. In the example from the previous section, if SalesReportJob has this annotation, than only one instance of SalesReportForJoe can execute at a given time, but it can execute concurrently with an instance of "SalesReportForMike". The constraint is based upon an instance definition (JobDetail), not on instances of the job class. However, it was decided (during the design of Quartz) to have the annotation carried on the class itself, because it does often make a difference to how the class is coded.

@PersistJobDataAfterExecution(保留data在运行完后)is an annotation that can be added to the Job class that tells Quartz to update the stored copy of the JobDetail’s JobDataMap after the execute() method completes successfully (without throwing an exception), such that the next execution of the same job (JobDetail) receives the updated values rather than the originally stored values. Like the @DisallowConcurrentExecution annotation, this applies to a job definition instance, not a job class instance, though it was decided to have the job class carry the attribute because it does often make a difference to how the class is coded (e.g. the “statefulness” will need to be explicitly understood by the code within the execute method).

如果你想要使用@PersistJobDataAfterExecution注解,你应该认真考虑也同时使用@DisallowConcurrentExecution注解,合理的避开可能出现的相同Job(JobDetail)留下相同实例数据的问题.

Other Attributes Of Jobs(job的其他属性)

这里是其他属性的总结,这些属性是通过Job实例的JobDetail对象定义的.

Durability(持久性) - 如果一个job是非持久性的,一旦没有任何的triggers和它相连就会自动销毁.总而言之,非持久化的job的生命周期存在他的triggers.

RequestsRecovery(请求恢复) - 如果一个job“请求恢复”，在调度器“硬关闭”（如：该进程崩溃，机器被关掉）时这个job还在执行，过后，当调度器再次启动时，他就会再次执行。在这种情况下，JobExecutionContext.isRecovering() 方法将会返回true.

JobExecutionException (job运行异常)

最后，我们来看看Job.execute(..)方法的一些细节。你能够从execute方法里抛出的仅有的异常类型就是JobExecutionException。因为这样，我们应该使用try-catch块包围整个execute方法内

容。我们还应该花一些时间看看JobExecutionException文档。当job执行发生异常时，通过设置JobExecutionException，可以让此job再次进入调度器或者今后不再运行。

三.Working with Triggers (触发器的工作原理)

Triggers
SimpleTrigger
CronTriggers

1.Triggers

像Jobs一样,triggers也很简单,但是我们还是要了解他的一些特性,Quartz提供了很多类型的trigger我们使用.你可以随意选择,以满足不同的需要,还有两个比较重要的类型:SimpleTragger和CronTriggers.详情见下面描述.

Common Trigger Attributes(常见的trigger属性)

有一个事实,即所有的trigger都有TriggerKey属性跟踪其身份,还要一些属性是所有trigger都有的.当你使用TriggerBuilder来设置你的trigger的公用属性.

下面有一些常见的trigger属性列表:

jobKey属性指向这个job运行的触发器.
startTime属性指向trigger第一次生效的时间,java.util.Date类型.
endTime属性指向的是最后一次执行的结果时间.

其他属性需要进一步讨论,在下面进行谈论.

Priority(优先)

可以对触发器经常优先设置(资源不足的时候).

Sometimes, when you have many Triggers (or few worker threads in your Quartz thread pool), Quartz may not have enough resources to immediately fire all of the Triggers that are scheduled to fire at the same time. In this case, you may want to control which of your Triggers get first crack at the available Quartz worker threads. For this purpose, you can set the priority property on a Trigger. If N Triggers are to fire at the same time, but there are only Z worker threads currently available, then the first Z Triggers with the highest priority will be executed first.
Any integer value is allowed for priority, positive or negative. If you do not set a priority on a Trigger, then it will use the default priority of 5.

Note: Priorities are only compared when triggers have the same fire time. A trigger scheduled to fire at 10:59 will always fire before one scheduled to fire at 11:00.

Note: When a trigger's job is detected to require recovery, its recovery is scheduled with the same priority as the original trigger.

Misfire Instructins(不触发指令)

触发器的另外一个重要的属性-不触发指令.如果一个持久化了的trigger因为调度器关闭或者因为在Quartz线程池中没有可以获得的线程而没有找到他的触发时间,那么一个触发都不会发生.不同的触发器有不同的不触发条件.

…待续(原文档page-17)