# Quartz

[**一.簡介**](#_一.簡介)[**九.Clustering**](#_九._Clutering)

[**二.簡單範例**](#_二.簡單範例)[**十.Spring-Quartz**](#_十._Spring-Quartz)

[**三.Job和JobDetail**](#_三.Job和JobDetail)[**十一.Other**](#_十一.Other)

[**四.Trigger**](#_四.JobDetail)[**十二.參考資料**](#_十二.參考資料)

[**五.Scheduler和SchedulerFactory**](#_五._Scheduler和SchedulerFactory)

[**六.JobDataMap**](#_六._JobDataMap)

[**七.JobStore**](#_七._JobStore)

[**八.Listener**](#_八._Listener)

# 一.簡介

用途

用來建立排程時間表, 執行Java下任意數量的工作任務(程式)

版本

1.0(2002/09/13), 目前版本2.2.1(2013/09/24)

所需jar檔

1. quartz-x.y.z.jar

2. quartz-jobs-x.y.z.jar

3. slf4j-api-x.y.zz.jar

4. c3p0-w.x.y.z.jar (store scheduling data in a database)

5. mchange-commons-java-x.y.zz.jar

重要介面

|  |  |  |  |
| --- | --- | --- | --- |
| **名稱** | **角色** | **用途** | **備註** |
| ***Job*** | 工作介面 | 所有工作類別皆需實作此介面 | 將欲執行工作內容寫在execute()內 |
| **JobFactory** | 工作工廠 | 產生工作 | 定義工作實例化時, 初始化的過程 |
| ***JobDetail*** | 工作任務介面 | 宣告工作類別的實例 | 也"只能"將工作類別實例化 |
| **JobBuilder** | 工作任務建立器 | 建立及設定工作任務實例 |  |
| ***Trigger*** | 觸發器介面 | 觸發工作任務 | 可設定工作在調度器上的執行的細節 |
| **TriggerBuilder** | 觸發器建立器 | 建立及設定觸發器實例 |  |
| **ScheduleBuilder** | 調度表建立器 | 設定觸發器細節 |  |
| ***Scheduler*** | 調度器介面 | 調度工作任務及觸發器 | 可加入多個工作, 一旦被停止後, 除非重新實例化, 否則無法重新啟動 |
| **SchedulerFactory** | 調度器工廠 | 產生調度器 | 有時習慣掛在JNDI上 |

## 簡單範例

SchedulerFactory schedulerFactory = **new** StdSchedulerFactory();

Scheduler scheduler = schedulerFactory.getScheduler();

//Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

scheduler.start();

JobDetail job = JobBuilder.*newJob*(MyJob.**class**)

.withIdentity("job1", "group1")

.build();

Trigger trigger = TriggerBuilder.*newTrigger*()

.withIdentity("trigger1", "group1")

.startNow()

.withSchedule(SimpleScheduleBuilder.*simpleSchedule*()

.withIntervalInSeconds(1000)

.repeatForever())

.build();

scheduler.scheduleJob(job, trigger);

# 三.Job和JobDetail

## Job

**簡述**: 自訂的工作類別, 需實作工作介面, 並將欲執行工作內容寫在execute(JobExecutionContext context)內

\*註 : context參數為執行當下環境細節, ex. 相關的Scheduler、Trigger、JobDetail及其他訊息

**生命週期** : 執行execute()前, 會先產生一個Job物件; 執行結束後該物件會被釋放, 然後帶GC機制回收

\*註 : 因為會自動產生Job物件, 所以Job類別必須有**預設建構子**(使用預設的JobFactory時)

**Factory-Shipped Job**

說明 : Quartz提供一些通用的工作類別

**package** org.quartz.jobs

功能 : ex. 寄mail, 調用EJB

## JobDetail

**簡述** : 工作任務介面, 工作類別的代理介面, 用以宣告工作類別的實例

\*註 : 也"只能"將工作類別實例化

**建立** : JobDetail物件由JobBuilder建立. 將Job指定給JobDetail, 再將JobDetail指定給Scheduler, Scheduler就會知道要執行的Job類別

**其他** : 為Job物件提供許多細節設定屬性, 及JobDataMap

## **行為控制**

\*註

1. 寫在Job類別上, 但作用針對JobDetail

2. 發生例外時可能沒有作用

|  |  |  |
| --- | --- | --- |
| 註釋名稱 | 說明 | 備註 |
| @DisallowConcurrentExecution | 不可並發相同的JobDetail | 但可並發相同的Job |
| @PersistJobDataAfterExecution | 保存執行完後的JobDataMap資料 | 即下一個相同的JobDetail會拿到上一個相同的JobDetail結束的資料 |

## 屬性

\*註 : 由JobDetail傳給Job

|  |  |  |  |
| --- | --- | --- | --- |
| 名稱 | 對應方法 | 說明 | 備註 |
| durability | setDurability(**boolean** durability)  **boolean** isDurable() | 持久化否 | 非持久化的Job, 一旦沒有活著的Trigger關聯, 就會自動被Scheduler移除 |
| requestsRecovery | setRequestsRecovery(**boolean** shouldRecover)  **boolean** requestsRecovery() | Scheulder強制關閉重啟後自動重跑否 | 強制關閉原因ex. Thread或Server當機 |

## JobExecutionException

**簡述** : Job.execute()唯一允許拋出的例外類別

\*註 : 應該在execute()裡用try-catch將例外做適當處理

## JobFactory

**簡述** : 工作工廠, 附加在調度器上, 用以產生工作物件. 其內定義工作實例化時, 初始化的過程. 亦可自定義JobFactory, 決定Job初始化時的過程

**執行過程** : 當觸發器被觸發時, 會透過調度器指定的JobFactory將關聯的工作類別實例化

**指定方式** : scheduler.setJobFactory(jobfactory);

**SimpleJobFactory**

**簡述** : 預設的JobFactory

**執行過程** :在Job初始化時, 只是在Job類別中呼叫newInstance(), 然後試著尋找適合的setter注入JobDataMap中的資料

**自訂JobFactory步驟**

**1. 新建** : 新增一類別JobFactory

**2. 實作** : 實作org.quartz.spi.JobFactory

**3. 撰寫內容** : 在newJob()內定義初始化過程, 並回傳Job物件

**4. 指定JobFacotry** : 將自訂的JobFactory加入至調度器

**應用 - AutowiredSupportedJobFactory**

**說明** : Spring沒有對Quartz工作類別支援自動裝配(@Autowired), 自訂一JobFactory以解決此問題

\*註 : 需要加入spring-context-support-x.y.z.jar、spring-beans-x.y.z.jar

**public** **class** AutowiredSupportedJobFactory **extends** SpringBeanJobFactory {

@Autowired

**private** AutowireCapableBeanFactory beanFactory;

@Override

**protected** Object createJobInstance(TriggerFiredBundle bundle) **throws** Exception {

Object jobInstance = **super**.createJobInstance(bundle);

beanFactory.autowireBean(jobInstance);

**return** jobInstance;

}

}

# 四.Trigger

**簡述** : 觸發器介面, 用來觸發工作任務, 可設定工作在調度器上的執行的細節

**執行過程** : 當被觸發那一刻, Trigger的第一個Thread會呼叫Job的execute()

**種類**

SimpleTrigger : 適用於一次性工作任務執行, 即在指定的時間區間內只執行一次工作任務; 或是指定時間多次觸發工作任務, 每次有固定的間隔時間

CronTrigger : 日曆時間表的觸發, ex. 每周五下午、每月10號 10:15

**共同屬性**

\*註 : 透過TriggerBuilder設定

|  |  |  |  |
| --- | --- | --- | --- |
| 名稱 | 說明 | TriggerBuilder對應方法 | 備註 |
| key | Trigger識別 | withIdentity(TriggerKey) | 若無設定TriggerBuilder會隨機產生 |
| jobKey | 被觸發的工作任務識別 | forJob(JobKey) |  |
| startTime | 首次觸發時間 | startAt(Date) | 參數為Date型態, 預設為當下(立即觸發) |
| endTime | 不再被觸發時間 | endAt(Date) |  |
| priority | (同一時間點)觸發的優先權 | withPriority(**int**) | 1.當Quartz無執行緒可用時會參考此設定  2.參數為int型態(正負皆可)  3.當工作任務的requestsRecovery為true時, 優先權設定會保留  4.預設為5 |
| misfireinstruction | 觸發失敗指令 | withSchedule(XxxScheduleBuilder.xxxSchedule()  .withmisfirehandlinginstructionXxx())  ---------------------------------------------  3.觸發失敗的觸發器會在調度器啟動時被檢查到, 根據"觸發失敗指令"設定來更新觸發器 | 1.失敗原因為調度器被關閉或Quartz無執行緒可用  2.預設為Trigger.MISFIRE\_INSTRUCTION\_SMART\_POLICY, 依當下狀況動態選擇其他策略 |
| calendarName | 日曆 | modifiedByCalendar(String) |  |

## Calendar

**package** org.quartz

**簡述** : Quartz定義的日曆介面, 可以與觸發器相關聯, 設定要觸發的時段或要排除的時段

**實作類別**

**package** org.quartz.impl.calendar

1. AnnualCalendar

2. BaseCalendar

3. CronCalendar

4. DailyCalendar

5. HolidayCalendar

6. MonthlyCalendar

7. WeeklyCalendar

**範例**

HolidayCalendar calendar = **new** HolidayCalendar();

calendar.addExcludedDate(**new** Date(...));//加入排除時段

scheduler.addCalendar("日曆識別名稱", calendar, **false**, **true**);//將calendar物件加入調度器中

Trigger trigger = TriggerBuilder.newTrigger()

//...略

.modifiedByCalendar("日曆識別名稱")//此觸發器會參考剛剛加入的calendar物件

.build();

## SimpleTrigger

**特性** : 在某時點上執行一次, 或固定的間隔時間執行多次

**建立** : 用TriggerBuilder和SimpleScheduleBuilder建立

**屬性**

|  |  |  |
| --- | --- | --- |
| 名稱 | 說明 | 備註 |
| startTime | 開始時間 | 參數型態為Date, 可透過DateBuilder設定 |
| endTime | 結束時間 | 會影響repeatCount設定. 參數型態為Date, 可透過DateBuilder設定 |
| repeatCount | 重複次數 | 參數型態為int. 可以是0或正整數, SimpleTrigger.REPEAT\_INDEFINITELY(-1)代表無限次 |
| repeatInterval | 間隔時間(毫秒) | 參數型態為long. 可以是0(並發)或正整數 |

\*註 : 通常endTime跟repeatCount會擇一使用; 或是設定endTime, 然後將repeatCount設為無限次(SimpleTrigger.REPEAT\_INDEFINITELY)

**範例**

SimpleTrigger trigger = (SimpleTrigger) TriggerBuilder.*newTrigger*()

.withIdentity("trigger1", "group1")

.forJob("job1", "group1")

.startAt(**new** Date())

.endAt(**new** Date())

.withSchedule(SimpleScheduleBuilder.*simpleSchedule*()

.withRepeatCount(10)

.withIntervalInMilliseconds(1000)

.withMisfireHandlingInstructionNextWithExistingCount())

.build();

**SimpleTrigger觸發失敗指令(misfireinstruction)常數**

\*註 : 預設為Trigger.MISFIRE\_INSTRUCTION\_SMART\_POLICY

***MISFIRE\_INSTRUCTION\_IGNORE\_MISFIRE\_POLICY***

***MISFIRE\_INSTRUCTION\_FIRE\_NOW***

***MISFIRE\_INSTRUCTION\_RESCHEDULE\_NOW\_WITH\_EXISTING\_REPEAT\_COUNT***

***MISFIRE\_INSTRUCTION\_RESCHEDULE\_NOW\_WITH\_REMAINING\_REPEAT\_COUNT***

***MISFIRE\_INSTRUCTION\_RESCHEDULE\_NEXT\_WITH\_REMAINING\_COUNT***

***MISFIRE\_INSTRUCTION\_RESCHEDULE\_NEXT\_WITH\_EXISTING\_COUNT***

## CronTrigger

**特性** : 基於日曆的觸發器, 例如可設定每個星期五、下午、10:15時、每個月第十天等等

\*註 : 亦需要設定startTime和endTime

**建立** : 用TriggerBuilder和CronScheduleBuilder建立

**Cron Expression**

**說明** : 設定CronTrigger的日曆, 由7個字段組成, 以空格間隔, 字段可以是範圍或是列舉

**字段**

|  |  |  |  |
| --- | --- | --- | --- |
| 順序 | 意義 | 值 | 備註 |
| 1 | 秒 | 0~59 |  |
| 2 | 分 | 0~59 |  |
| 3 | 時 | 0~23 |  |
| 4 | 日 | 1~31 | 依每月不同 |
| 5 | 月 | 0~11或英文前三碼 | JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC |
| 6 | 週 | 1(週日)~7或英文前三碼 | SUN, MON, TUE, WED, THU, FRI, SAT |
| 7 | 年 |  | 可不設 |

**特殊符號**

|  |  |  |  |
| --- | --- | --- | --- |
| 符號 | 意義 | 適用字段 | 備註 |
| 逗號(,) | 列舉 | 全部 |  |
| 減號(-) | 範圍 | 全部 |  |
| 乘號(\*) | 任何 | 全部 |  |
| 斜線(/) | 遞增量 | 全部 | 分字段"0/15", 表示0分開始, 每隔15分鐘, 等同"0,15,30,45"或"/15" |
| 問號(?) | 不指定值 | 日、週 | 表示無具體的值 |
| 井號(#) | 第N個 | 全部 | 週字段"6#3"或"FRI#3", 表示此月第三個週五 |
| Weekday(W) | 最近工作日 | 全部 | 日字段"15W", 表示離15日最近的工作日 |
| Last(L) | 最後 | 日、週 | 1.用在月字段即表示本後最後一天; 用在週字段即為週六, 等同"7"或"SAT"  2.前面多一值,ex. "6L"或"FRIL", 則表示最後一個週五  3.跟減號(-)一起使用, 表示最後N個, ex. 日字串"L-3", 表示最後三天  4.不可跟列舉一起使用 |

**範例**

CronTrigger trigger = (CronTrigger) TriggerBuilder.*newTrigger*()

.withIdentity("trigger1", "group1")

.forJob("job1", "group1")

.startAt(**new** Date())

.endAt(**new** Date())

.withSchedule(CronScheduleBuilder.*cronSchedule*("0 0 0 30 12 ?"))

.build();

**CronTrigger的觸發失敗指令(misfireinstruction)常數**

\*註 : 預設為Trigger.MISFIRE\_INSTRUCTION\_SMART\_POLICY

***MISFIRE\_INSTRUCTION\_IGNORE\_MISFIRE\_POLICY***

***MISFIRE\_INSTRUCTION\_DO\_NOTHING***

***MISFIRE\_INSTRUCTION\_FIRE\_NOW***

# 五. Scheduler和SchedulerFactory

## Scheduler

**簡述** : 調度器介面, 調度工作任務及觸發器, 可加入多個工作, 一旦被停止後, 除非重新實例化, 否則無法重新啟動

**生命週期**

SchedulerFactory產生 -> 加入/刪除/查詢 Job或Trigger -> 其他Scheduler相關操作(ex. 中止) -> shutdown()

\*註

1. 實例化後即能啟動

2. 當啟動時, 會檢查之內所有觸發失敗的觸發器, 根據"觸發失敗指令"設定來更新觸發器

3. 未啟動或是終止狀態時不會觸發任何一個觸發器去執行工作任務

4. 當加入Job物件時, 會在Client端建立JobDetail物件

5. 一旦執行了shutdown()(關閉), 除非重新實例化, 否則就不會再啟動

## SchedulerFactory

**簡述** : 調度器工廠, 用以產生調度器

**建立** : 可掛在JNDI上, 亦可直接new成物件使用

**實作類別**

1. StdSchedulerFactory

2. DirectSchedulerFactory

### StdSchedulerFactory

**特性** : 用getScheduler()建立調度器, 用initialize(Properties props)初始化調度器

**注意事項**

1. 一般將參數值寫在.properties檔

2. 包括ThreadPool、JobStore、DataSource設定

\*註 : 在quartz-2.2.3-distribution.tar中docs/config(???)下, 有範例

### DirectSchedulerFactory

**特性** : 適用於在程式裡建立調度器

**注意事項**

1. 不建議使用

2. 需很了解內部運作

3. 不允許註釋和組態檔方式

# 六. JobDataMap

**簡述** : 儲存特定Job物件的狀態資料. Job物件被執行時, Job會收到存有資料的JobDataMap; 亦可當Trigger和Job傳遞資料的媒介. 實作JDK的

Map介面; 另有方便存取基本型態的方法

**自動映射** : 如果在Job類別中加入屬性, 且屬性名稱剛好與存入JobDataMap的key相同時, Quartz預設的JobFactory會在初始化Job物件時自動

透過setter放入資料

**注意事項**

1. 可存任何可序列化的資料, 即存入的資料皆會被序列化; 所以存入第三方定義類別的資料時可能會有風險

\*註 : 詳情請參考Java Developer Connection的Serialization In The Real World

2. 可選擇使用JDBC-JobStore和JobDataMap設計成只能存基本型態和String類別的資料, 以解決第1點問題

3. 若JobDetail和Trigger使用JobDataMap時, 使用了相同的key, 則較後面設定的資料會覆蓋前面的資料

**範例**

**存**

JobDetail job = JobBuilder.*newJob*(MyJob.**class**)

.withIdentity("job1", "group1")

.usingJobData("stringData", "string")

.usingJobData("floatData", 123.456f)

.build();

**取**(在自訂的Job類別)

**class** MyJob **implements** Job {

**private** String stringData;//初始化後資料會直接存進來

**private** **float** floatData;//初始化後資料會直接存進來

**public** **void** execute(JobExecutionContext context) **throws** JobExecutionException {

JobDataMap dataMap = context.getJobDetail().getJobDataMap();

//JobDataMap dataMap = context.getMergedJobDataMap(); <--取Job和Trigger合併的JobDataMap

String stringData = dataMap.getString("stringData");

**float** floatData = dataMap.getFloat("floatData");

}

//stringData、floatData的setter/getter here

}

# 七. JobStore

**簡述** : 提供給SchedulerFactory, 用來創建調度器實例. 追蹤調度器中所有工作的資料, 包含作業任務、觸發器、日曆等

**注意事項 :** 只需設定JobStores, 不要直接使用JobStores實例, JobStores是給Quartz背後使用

**自訂 :** 實作org.quartz.spi.JobStore介面即可

\*註 : 所有的JobStore都實作org.quartz.spi.JobStore介面

**種類**

**RAMJobStore**

**JDBCJobStore**

**TerracottaJobStore**

## RAMJobStore

**特性** : 將資料儲存在RAM中, 易使用, 效率最高. 但程式意外停止時, 調度訊息將會消失

**設定方式** : 在Quartz組態檔中加入

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore

## JDBCJobStore

**特性** : 透過JDBC將資料存在資料庫中, 設定較複雜, 效率較差, 可配合在資料庫中設定index提升效率

註 : 支援的DBMS : Oracle, PostgreSQL, MySQL, MS SQLServer, HSQLDB, DB2

**設定方式** : 在Quartz組態檔中加入

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX

或

org.quartz.jobStore.class = org.quartz.impl.jobcjobstore.JobStoreCMT

**交易管理**

**JobStoreTx** : 不把調度命令和其他交易綁定時使用(最常使用)

**JobStoreCMT** : 關聯其他交易時使用

**DriverDelegate**

**用途** : 負責完成任何JDBC的工作, 需和指定的DBMS對應, 然後再將DriverDelegate指定給JobStore

**package** org.quartz.impl.jdbcjobstore

設定方式 : org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.類別名稱

**種類**

|  |  |
| --- | --- |
| 名稱 | 對應DBMS |
| OracleDelegate | Oracle |
| MSSQLDelegate | MSSQLServer |
| DB2vxDelegate | DB2 6-、7-、8- |
| HSQLDBDelegate | HSQLDB |
| PostgreSQLDelegate | PostgreSQL |
| WeblogicDelegate | Weblogic |
| SybaseDelegate | Sybase |
| StdJDBCDelegate | 通用 |

**使用步驟**

1. **建立表格** : 在資料庫中建立一套TABLE

註

1. 在quartz-2.2.3-distribution.tar中docs/dbTables下, 有.sql檔, 依DBMS分

2. TABLE NAME預設都是以QRTZ\_開頭(前置詞), 若自訂名稱則需再Quartz組態檔中指定

org.quartz.jobStore.tablePrefix = 自訂的前置詞

2. **建立Quartz組態檔** : 在classpath下新建一檔案quartz.properties

3. 撰寫Quartz組態檔 : 在quartz.properties中加入

3.1 設定**jobStore** :

3.1.1 設定交易管理

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.類別名稱

3.1.2 設定DriverDelegate

org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.類別名稱

3.1.3 指定DataSource

org.quartz.jobStore.dataSource = 自訂DataSource名稱

註 : 自訂DataSource名稱為步驟3.3所設定

3.1.4 設定Quartz TABLE前置詞

org.quartz.jobStore.tablePrefix = QRTZ\_

或

org.quartz.jobStore.tablePrefix = 自訂的前置詞

3.1.5 設定JobDataMap內型態為String(非必要)

org.quartz.jobStore.useProperties = true

註 : 可避免一些問題, 且較安全

3.2 設定**Quartz執行緒**

3.2.1 指定類別

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool

3.2.2 設定數量

org.quartz.threadPool.threadCount = 5

3.3 設定**DataSource**

方式1 - 自訂

MySQL

org.quartz.dataSource.自訂DataSource名稱.driver = com.mysql.jdbc.Driver

org.quartz.dataSource.自訂DataSource名稱.URL = jdbc:mysql://hostName:3306/QUARTZ\_DEMO

org.quartz.dataSource.自訂DataSource名稱.user = root

org.quartz.dataSource.自訂DataSource名稱.password = 0000

org.quartz.dataSource.自訂DataSource名稱.maxConnections = 8

MSSQL Server

org.quartz.dataSource.自訂DataSource名稱.driver = com.microsoft.sqlserver.jdbc.SQLServerDriver

org.quartz.dataSource.自訂DataSource名稱.URL = jdbc:sqlserver://hostName:1433;DatabaseName=mPOS\_DEV

org.quartz.dataSource.自訂DataSource名稱.user = mPOS\_DEV

org.quartz.dataSource.自訂DataSource名稱.password = mPOS\_DEV123

org.quartz.dataSource.自訂DataSource名稱.maxConnections = 8

註 : 建議maxConnections的值要比threadCount的值多2(以上)

方式2 - JNDI

Tomcat

org.quartz.dataSource.myDS.jndiURL = java:comp/env/quartz

JBoss

org.quartz.dataSource.myDS.jndiURL = java:jboss/evn/mPOSDB

## TerracottaJobStore

**特性** : 將訊息儲存在Terracatta Server中, 效率優於JDBCJobStore, 但較RAMJobStore差

**設定方式**

org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore

org.quartz.jobStore.tcConfigUrl = localhost:9510 <--指定Terracotta Server

## quartz.properties範例

# Scheduler setting

#org.quartz.scheduler.skipUpdateCheck = true

org.quartz.scheduler.instanceName = DatabaseScheduler

org.quartz.scheduler.instanceId = AUTO

org.quartz.scheduler.jobFactory.class = org.quartz.simpl.SimpleJobFactory

# JobStore setting

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTx

org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.StdJDBCDelegate

org.quartz.jobStore.tablePrefix = QRTZ\_

org.quartz.jobStore.dataSource = quartzDataSource

org.quartz.jobStore.useProperties = true

# Thread setting

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool

org.quartz.threadPool.threadCount = 5

# DataSource setting

org.quartz.dataSource.quartzDataSource.driver = com.mysql.jdbc.Driver

org.quartz.dataSource.quartzDataSource.URL = jdbc:mysql://localhost:3306/QUARTZ\_DEMO

org.quartz.dataSource.quartzDataSource.user = root

org.quartz.dataSource.quartzDataSource.password = 0000

org.quartz.dataSource.quartzDataSource.maxConnections = 8

org.quartz.dataSource.quartzDataSource.jndiURL = java:comp/env/quartz

# 八. Listener

**簡述** : 監聽器, 用來監聽Quartz重要物件(工作任務、觸發器和調度器)相關事件

**注意事項** : 監聽器在每次執行前都需重新註冊

**種類**

JobListener

TriggerListener

SchedulerListener

## Matcher(匹配器)

**簡述** : 匹配監聽器監聽哪些對象(工作 | 觸發器). 提供多種方式匹配, ex. key匹配、group匹配、邏輯匹配、所有匹配..等等

**種類**

|  |  |  |
| --- | --- | --- |
| 名稱 | 說明 | 範例 |
| NameMatcher | 依name匹配 | NameMatcher.*nameEquals*("name") |
| GroupMatcher | 依group匹配 | GroupMatcher.*groupEquals*("group") |
| KeyMatcher | 依key匹配 | KeyMatcher.*keyEquals*(TriggerKey.*triggerKey*("name", "group"))  KeyMatcher.*keyEquals*(JobKey.*jobKey*("name", "group")) |
| AndMatcher | 同時被其他兩匹配器匹配 | AndMatcher.*and*(NameMatcher.*nameStartsWith*("startToken"), NameMatcher.*nameEndsWith*("endToken")) |
| OrMatcher | 被兩匹配器其一匹配 | OrMatcher.*or*(NameMatcher.*nameContains*("token1"), NameMatcher.*nameContains*("token2")) |
| NotMatcher | 排除匹配 | NotMatcher.*not*(NameMatcher.*nameContains*("token")) |
| EverythingMatcher | 匹配全部 | EverythingMatcher.*allTriggers*()  EverythingMatcher.*allJobs*() |

## JobListener

**簡述** : 工作監聽器, 接收關於Job的事件

**事件**

|  |  |
| --- | --- |
| 名稱 | 對應方法 |
| 即將被執行 | jobToBeExecuted(JobExecutionContext context) |
| 工作執行被否決 | jobExecutionVetoed(JobExecutionContext context) |
| 執行完畢 | jobWasExecuted(JobExecutionContext context, JobExecutionException jobException) |

**建立步驟**

**1. 建立及撰寫工作監聽器**

1.1 實作JobListener介面或繼承JobListenerSupport類別

1.2 改寫事件方法

註 : 一定要實作getName(), 並回傳非null之值. 因為監聽器需要呼叫getName()廣播自己的名稱

**2. 宣告工作監聽器**

JobListener jobListener = **new** JobListener()

**3. 註冊**

3.1 取得監聽器管理者

ListenerManager listenerManager = scheduler.getListenerManager()

3.2 註冊

listenerManager.addJobListener(jobListener, 匹配器類別名.靜態方法())

**\***補充 - 移除工作監聽器

listenerManager.removeJobListener("name")

## TriggerListener

**簡述** : 觸發監聽器, 接收關於Trigger的事件

**事件**

|  |  |  |
| --- | --- | --- |
| 名稱 | 對應方法 | 備註 |
| 觸發 | triggerFired(Trigger trigger, JobExecutionContext context) |  |
| 否決工作執行 | vetoJobExecution(Trigger trigger, JobExecutionContext context) | 若回傳true, 則工作不執行 |
| 觸發失敗 | triggerMisfired(Trigger trigger) |  |
| 觸發完成 | triggerComplete(Trigger trigger, JobExecutionContext context,  CompletedExecutionInstruction triggerInstructionCode) | 工作開始執行 |

**建立步驟**

**1. 建立及撰寫觸發監聽器**

1.1 實作TriggerListener介面或繼承TriggerListenerSupport類別

1.2 改寫事件方法

註 : 一定要實作getName(), 並回傳非null之值. 因為監聽器需要呼叫getName()廣播自己的名稱

**2. 宣告工作監聽器**

TriggerListener triggerListener = **new** TriggerListener();

**3. 註冊**

3.1 取得監聽器管理者

ListenerManager listenerManager = scheduler.getListenerManager()

3.2 註冊

listenerManager.addTriggerListener(triggerListener, 匹配器類別名.靜態方法())

\*補充 - 移除觸發監聽器

listenerManager.removeTriggerListener("name")

## SchedulerListener

**簡述** : 調度監聽器. 與觸發監聽器、工作監聽器類似, 但調度監聽器在調度器內部接收通知, 而不需關聯任何具體的觸發器或工作任務

**事件**

|  |  |  |
| --- | --- | --- |
| 名稱 | 對應方法 | 備註 |
| 工作任務加入 | jobAdded(JobDetail jobDetail) |  |
| 工作任務暫停 | jobPaused(JobKey jobKey) |  |
| 工作任務被取消暫停 | jobResumed(JobKey jobKey) |  |
| 一個群組的工作任務被暫停 | jobsPaused(String jobGroup) |  |
| 一個群組的工作任務被取消暫停 | jobsResumed(String jobGroup) |  |
| 工作任務刪除 | jobDeleted(JobKey jobKey) |  |
| 工作任務被調度 | jobScheduled(Trigger trigger) |  |
| 工作任務未在調度上 | jobUnscheduled(TriggerKey triggerKey) |  |
| 觸發器被暫停 | triggerPaused(TriggerKey triggerKey) |  |
| 觸發器被取消暫停 | triggerResumed(TriggerKey triggerKey) |  |
| 一個群組的觸發器被暫停 | triggersPaused(String triggerGroup) |  |
| 一個群組的觸發器被取消暫停 | triggersResumed(String triggerGroup) |  |
| 觸發器達到不會再被觸發的條件 | triggerFinalized(Trigger trigger) |  |
| 調度器啟動中 | schedulerStarting() |  |
| 調度器已被啟動 | schedulerStarted() |  |
| 調度器在待命模式 | schedulerInStandbyMode() |  |
| 調度器關閉中 | schedulerShuttingdown() |  |
| 調度器已關閉 | schedulerShutdown() |  |
| 調度器已清除所有物件 | schedulingDataCleared() | 物件包含Jobs、Triggers、Calendars |
| 調度器發生嚴重錯誤 | schedulerError(String msg, SchedulerException cause) | ex. JobStore重複的失敗、已觸發但無法實例化工作 |

**建立步驟**

**1. 建立及撰寫調度監聽器**

1.1 實作SchedulerListener介面或繼承SchedulerListenerSupport類別

1.2 改寫事件方法

註 : 一定要實作getName(), 並回傳非null之值. 因為監聽器需要呼叫getName()廣播自己的名稱

**2. 宣告調度監聽器**

SchedulerListener schedulerListener = **new** SchedulerListener()

**3. 註冊**

3.1 取得監聽器管理者

ListenerManager listenerManager = scheduler.getListenerManager()

3.2 註冊

listenerManager.addSchedulerListener(schedulerListener)

\*補充 - 移除調度監聽器

listenerManager.removeSchedulerListener(schedulerListener)

# 九. Clutering

**簡述** : JDBCJobStore的JobStoreTx和JobStoreCMT或是TerracottaJobStore可以叢集並發執行

**特性** : 負載平衡、工作任務切換(當JobDetail的requestsRecovery = **true**)

## JobStoreTx / JobStoreCMT叢集

**開啟** : org.quartz.jobStore.isClustered = true

**注意事項**

**相同的設定** : 同叢集的每個實例都應該使用相同的核心組態檔(quartz.properties), 除了以下兩個**例外**..

1. 實例的id : org.quartz.scheduler.instanceId = 名稱 || AUTO

\*註 : 每個實例id必需唯一, 可設成AUTO由Quartz自己產生

2. 執行緒池大小 : org.quartz.threadPool.threadCount = 整數

**時間同步** : 避免在網路環境隔離的主機上執行同個叢集, 除非使用時間同步服務(time-sync service), 誤差會在1秒內

**資料同步** : 避免叢集跟非叢集的存取相同的資料或TABLE, 否則可能導致資料問題或不穩定問題

**執行實例** : 每次觸發只會有一個實例執行, 且每次可能是不同實例. 選擇方式依負載平衡機制, Quartz會盡量選擇活著而空閒的實例

## TerracottaJobStore叢集

**注意事項** \*註 : 企業版TerracottaJobStore更多進階功能, 詳情<http://www.terracotta.org/quartz>

1. 所有調度器將全部設定叢集

2. 可考慮設定持久化(persistence)和HA

# 十. Spring-Quartz

**簡述** : Spring對Quartz的支援

**jar檔** : spring-context-support-x.y.z.jar

**package** org.springframework.scheduling.quartz

## 基本用法建立步驟

### 0. 完成Spring及Quartz基本步驟

### **1. 加入jar檔**

spring-context-support-x.y.z.jar

### 2. 建立及撰寫自訂的工作類別

2.1 建立一類別 : XxxJob

2.2 撰寫執行內容 : 撰寫一**公開的一般方法**即可

2.ex

**public** **class** FooJob {

**public** **void** execute() {

System.***out***.println("工作開始執行");

System.***out***.println("工作執行中");

System.***out***.println("工作結束");

}

}

### 3. 建立Spring-Quartz核心組態檔 : applicationContext-quartz.xml

### 4. 撰寫Spring-Quartz核心組態檔在 : applicationContext-quartz.xml, <beans>中加入..

4.1 **宣告工作**

<bean id=*"fooJob"* class=*"xxx.yyy.FooJob"* /> <--指向步驟2.1的工作類別

4.2 **宣告工作任務**

<bean id=*"fooJobDetail"* class=*"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"*>

<property name=*"targetObject"* ref=*"fooJob"* /> <--注入步驟4.1宣告的工作

<property name=*"targetMethod"* value=*"execute"* /> <--指定步驟2.2撰寫的方法名稱

</bean>

4.3 **宣告觸發器**

SimpleTrigger

<bean id=*"fooSimpleTrigger"* class=*"org.springframework.scheduling.quartz.SimpleTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"fooJobDetail"* /> <--注入步驟4.2宣告的工作任務

<property name=*"repeatInterval"* value=*"60000"* /> <--設定間隔時間(毫秒)

<property name=*"startDelay"* value=*"300000"* /> <--設定開始延遲時間(毫秒)

</bean>

CronTrigger

<bean id=*"fooCronTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"fooJobDetail"*/> <--注入步驟4.2宣告的工作任務

<property name=*"cronExpression"* value=*"0 55 \* \* \* ?"* /> <--設定CRON表示式

</bean>

4.4 **宣告調度器**

<bean class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"jobDetails"*> <--注入工作任務

<list>

<ref bean=*"fooJobDetail"* /> <--指向步驟4.2宣告的工作任務

</list>

</property>

<property name=*"triggers"*> <--注入觸發器

<list>

<ref bean=*"fooSimpleTrigger"* /> <--指向步驟4.3宣告的觸發器

</list>

</property>

</bean>

## 進階用法步驟

### 0. 完成Spring及Quartz基本步驟

### 1. 加入jar檔 : spring-context-support-x.y.z.jar

### 2. 建立及撰寫自訂的工作類別

2.1 建立一類別 : XxxJob, 實作org.quartz.Job或繼承org.springframework.scheduling.quartz.QuartzJobBean(Spring supported)

2.2 撰寫執行內容 : 將欲執行的程式寫在execute()中

2.3 宣告成Spring bean : 在class上一行加上@Component

2.ex

@Component

**public** **class** FooJob **extends** QuartzJobBean {

@Autowired

**private** MemberMapper mapper;

@Override

**protected** **void** executeInternal(JobExecutionContext context) **throws** JobExecutionException {

JobDataMap dataMap = context.getJobDetail().getJobDataMap();

mapper.selectByKey(dataMap.getInt("id"));

}

}

### 3. 建立Spring-Quartz核心組態檔 : applicationContext-quartz.xml

### 4. 撰寫Spring-Quartz核心組態檔 : 在applicationContext-quartz.xml, 在<beans>中加入..

4.1 宣告工作任務

<!-- 使用Spring support的JobDetailFactoryBean -->

<bean id=*"fooJob"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>

<property name=*"jobClass"* value=*"xxx.yyy.FooJob"* /> <--指向步驟2.1的工作類別

<property name=*"durability"* value=*"true"* /> <--設定持久化否

<!-- <property name="jobDataAsMap"> --> <--利用JobDataMap傳資料到工作任務裡

<!-- <map> -->

<!-- <entry key="鍵1" value="值1" /> -->

<!-- <entry key="鍵2" value-ref="值2(物件)" /> -->

<!-- </map> -->

<!-- </property> -->

</bean>

4.2 宣告觸發器

<!-- 使用Spring support的JobDetailFactoryBean -->

<bean id=*"fooTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"fooJob"* /> <--注入步驟4.1宣告的工作任務

<property name=*"cronExpression"* value=*"0 0 12 1/1 \* ? \*"* /> <--設定CronExpression

</bean>

4.3 宣告DataSource

<bean id=*"quartzDataSource"* class=*"org.springframework.jndi.JndiObjectFactoryBean"*>

<property name=*"jndiName"* value=*"java:comp/env/quartz"* />

</bean>

4.5 宣告調度器

<!-- 使用Spring support的Spring support的SchedulerFactoryBean -->

<bean id=*"jdbcQuartzScheduler"* lazy-init=*"false"* autowire=*"no"*

class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"configLocation"* value=*"classpath:quartz.properties"* /> <--指定Quartz組態檔

<property name=*"dataSource"* ref=*"quartzDataSource"* /> <--注入4.3宣告的DataSource

<!-- <property name="jobFactory" ref="autowiredSupportedJobFactory" /> -->

<!-- <property name="globalJobListeners" ref="jobListener1" /> --> <--指定工作監聽器(可設多個)

<!-- <property name="globalTriggerListeners" ref="triggerListener1" /> --> <--指定觸發監聽器(可設多個)

<property name=*"overwriteExistingJobs"* value=*"false"* /> <--覆蓋已存在工作物件否

<property name=*"autoStartup"* value=*"true"* /> <--自動啟動否

<property name=*"triggers"*> <--注入觸發器(可設多個)

<list>

<ref bean=*"fooTrigger"* /> <--指向步驟4.2宣告的觸發器

</list>

</property>

</bean>

### 5. 加入AutowiredSupportedJobFactory

5.1 建立及撰寫AutowiredSupportedJobFactory

**public** **class** AutowiredSupportedJobFactory **extends** SpringBeanJobFactory {

@Autowired

**private** AutowireCapableBeanFactory beanFactory;

@Override

**protected** Object createJobInstance(TriggerFiredBundle bundle) **throws** Exception {

Object jobInstance = **super**.createJobInstance(bundle);

beanFactory.autowireBean(jobInstance);

**return** jobInstance;

}

}

5.2 指定JobFactory : 在applicationContext-quartz.xml, <beans>中加入..

5.1.1 宣告自訂的JobFactory

<bean id=*"autowireJobFactory"* class=*"xxx.yyy.* *AutowiredSupportedJobFactory "* />

↑↑↑↑指向步驟5.1的JobFactory類別

5.1.2 將JobFactory加入至調度器

<bean ... class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

...

<property name=*"jobFactory"* ref=*"autowireJobFactory"* /> <--注入步驟5.1.1宣告的JobFactory

...

</bean>

### 範例 – applicationContext-quarz.xml

\*註 : quartz.properties **ALL SETTING** : <http://www.quartz-scheduler.org/documentation/quartz-2.x/configuration/>

<bean id=*"quartzDataSource"* class=*"org.springframework.jndi.JndiObjectFactoryBean"*>

<property name=*"jndiName"* value=*"java:comp/env/quartz"* />

</bean>

<bean id=*"fooJob"* class=*"org.springframework.scheduling.quartz.JobDetailFactoryBean"*>

<property name=*"jobClass"* value=*"xxx.yyy.FooJob"* />

<property name=*"durability"* value=*"true"* />

<property name=*"jobDataAsMap"*>

<map>

<entry key=*"鍵1"* value=*"值1"* />

<entry key=*"鍵2"* value-ref=*"值2(物件)"* />

</map>

</property>

</bean>

<bean id=*"fooTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>

<property name=*"jobDetail"* ref=*"fooJob"* />

<property name=*"cronExpression"* value=*"0 0 12 1/1 \* ? \*"* />

</bean>

<bean id=*"jdbcQuartzScheduler"* lazy-init=*"false"* autowire=*"no"*

class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>

<property name=*"configLocation"* value=*"classpath:quartz.properties"* />

<property name=*"dataSource"* ref=*"quartzDataSource"* />

<property name=*"jobFactory"* ref=*"autowiredSupportedJobFactory"* />

<property name=*"globalJobListeners"* ref=*"jobListener1"* />

<property name=*"globalTriggerListeners"* ref=*"triggerListener1"* />

<property name=*"overwriteExistingJobs"* value=*"false"* />

<property name=*"autoStartup"* value=*"true"* />

<property name=*"triggers"*>

<list>

<ref bean=*"fooTrigger"* />

</list>

</property>

</bean>

<bean id=*"autowiredSupportedJobFactory"* class=*"core.quartz.AutowiredSupportedJobFactory"* />

# 十一.Other

**DateBuilder** : 許多簡易的方法建立Date物件

**Identities**

1. Job和Trigger被加入至Scheduler時的識別名稱

2. 可群組化管理(group)

3. name + group必須唯一

**外掛(Plugin)**

1. 介面 : org.quartz.spi.SchedulerPlugin

2. 供外掛功能開發

3. Quartz提供的外掛在org.quartz.plugin套件中, 功能ex..

3.1 調度器啟動時自動調用工作任務

3.2 紀錄工作任務和觸發器的事件

3.3 當JVM退出時, 確保調度器關閉

**Logging**

1. Quartz預設使用SLF4J

2. 相關類別

LoggingJobHistoryPlugin : 工作任務執行紀錄

LoggingTriggerHistoryPlugin : 觸發器觸發紀錄

**JTA交易**

1. Job使用JTA

設定

全局 : org.quartz.scheduler.wrapJobExecutionInUserTransaction = true

局部 : @ExecuteInJTATransaction

說明 : JTA交易的begin()會在job.execute()**前**被呼叫, commit()則會在job.execute()執行結束**後**被呼叫

2. JobStoreCMT使用JTA

2.1 需注意調度器調用方法**前**啟動交易

2.2 可透過UserTransaction介面執行, 或是把自訂的程式放在有SessionBean的調度器中

**Configuration, Resource Usage**

**說明** : Quartz框架是模組化的, 運作前需先設定, 將各模組咬合在一起

**模組**

ThreadPool

JobStore

DataSources(非必要)

The Scheduler itself

**ThreadPool**

說明

簡述 : 提供一組執行緒給Quartz執行工作任務時使用

數量 : 池裡的執行緒越多, 可並發的工作任務數量就越多; 但數量太多可能導致系統當機. 數量的設定需考慮並發的工作任務數量和執行時間

注意 : 若觸發器的觸發時間到了, 但沒有可用的執行緒, Quartz會讓觸發器暫停(block), 直到有可用的執行緒; 可能導致延遲或觸發失敗

**package** org.quartz.spi

實作類別

SimpleThreadPool : 指定固定數量的執行緒, 設定後就不能改變. 通常使用此類別

ZeroSizeThreadPool

The Scheduler itself

1. 建立一調度器實例, 並設定名稱

2. 設定RMI(Remote Method Invocation)

2.1 允許遠端連線否(是否為RMI服務對象)

org.quartz.scheduler.rmi.proxy = true

2.2 主機名稱(2.1為true時)

org.quartz.scheduler.rmi.registryHost = localhost

2.3 埠號(2.1為true時)

org.quartz.scheduler.rmi.registryPort = 1099

\*註 : StdSchedulerFactory也可以產生調度器實例, 實際上是代理在RMI下, 建立調度器實例

3. 指定JobStore

4. 指定ThreadPool

其他

1. 有些Batch框架並沒有區分Job和Trigger, 意義上像是將Job和Trigger結合

2. Job和Trigger分開的好處 : Job和Trigger成多對多關係, 通用性佳; 另可對Job動態更換Trigger(不需重新實例化Job)

# 十二.參考資料

Quartz官方教程 : <http://www.quartz-scheduler.org/documentation/quartz-2.2.x/tutorials/>

CSDN Quartz簡中翻譯教程 : <http://so.csdn.net/so/search/s.do?q=Quartz+%E6%A1%86%E6%9E%B6+%E6%95%99%E7%A8%8B%28%E4%B8%AD%E6%96%87%E7%89%88%292.2.x+%E4%B9%8B%E7%AC%AC+%E8%AF%BE+-+CSDN%E5%8D%9A%E5%AE%A2&t=blog&o=&s=&l>

Quartz 2.2.1 API Document : <http://www.quartz-scheduler.org/api/2.2.1/>