

Anyframe Scheduling Plugin



Version 1.0.3

저작권 © 2007-2011 삼성SDS

본 문서의 저작권은 삼성SDS에 있으며 Anyframe 오픈소스 커뮤니티 활동의 목적하에서 자유로운 이용이 가능합니다. 본 문서를 복제, 배포할 경우에는 저작권자를 명시하여 주시기 바라며 본 문서를 변경하실 경우에는 원문과 변경된 내용을 표시하여 주시기 바랍니다. 원문과 변경된 문서에 대한 상업적 용도의 활용은 허용되지 않습니다. 본 문서에 오류가 있다고 판단될 경우 이슈로 등록해 주시면 적절한 조치를 취하도록 하겠습니다.

I. Introduction	1
II. Spring Scheduling	2
1. Spring Scheduling Configuration	3
1.1. TaskExecutor	3
1.2. TaskScheduler	4
1.3. XML based Scheduling	4
1.4. Annotation based Scheduling & Asynchronous Execution	6
1.4.1. Scheduling	6
1.4.2. Asynchronous Execution	7
1.5. Resources	7
III. Quartz Scheduling	9
2. Quartz Integration	10
2.1. Quartz Scheduler	10
2.1.1. Advanced Quartz	13
2.1.2. Samples	14

I.Introduction

Scheduling Plugin은 Spring과 오픈 소스 작업 스케줄링 프레임워크인 Quartz [<http://www.opensymphony.com/quartz/>]를 연계하여 특정 작업에 대해 스케줄링하는 방법을 가이드하기 위한 샘플 코드와 이 오픈소스들을 활용하는데 필요한 참조 라이브러리들로 구성되어 있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 scheduling-plugin을 설치한다.

```
mvn anyframe:install -Dname=scheduling
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Dependent Plugins

Plugin Name	Version Range
Query [http://dev.anyframejava.org/docs/anyframe/plugin/optional/query/1.1.2/reference/htmlsingle/query.html]	2.0.0 > *

II.Spring Scheduling

Spring은 특정 Task에 대한 Asynchronous Execution과 Scheduling을 위해 TaskExecutor와 TaskScheduler(Spring 3 이후)를 제공하고 있다.

1.Spring Scheduling Configuration

본 장에서는 TaskExecutor와 TaskScheduler에 대해 간단히 살펴볼 것이다. 또한 XML과 Annotation 기
반에서 이들을 사용하는 방법에 대해 알아보도록 하자.

1.1.TaskExecutor

TaskExecutor는 java.util.concurrent.Executor를 extends하여 정의된 인터페이스로써 지정된 Task
를 실행시키기 위한 execute(Runnable task) 메소드를 정의하고 있다. Spring에서는 다양
한 TaskExecutor 구현체를 제공하고 있으며 각 구현체에 대해서는 Spring Documentation 내
의 TaskExecutor types [[http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/
html/scheduling.html#scheduling-task-executor](http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/scheduling.html#scheduling-task-executor)]를 참조하도록 한다.

Spring에서 제공하는 TaskExecutor를 사용하여 특정 Task를 실행시키는 TaskExecutor를 개발하기 위
해서는 XML 기반으로 Spring TaskExecutor의 속성을 정의한 후, 구현 대상이 되는 TaskExecutor에서
정의된 Spring TaskExecutor를 Inject하여 사용하면 된다. 해당 TaskExecutor 내에서는 Inject한 Spring
TaskExecutor의 execute() 메소드를 호출함으로써 Task를 실행할 수 있으며 Task Execution을 위한 Rule
은 자체적으로 구현해야 한다. 한편 Thread 형태로 구현된 Task는 Spring TaskExecutor 구현체의 특성
에 맞게 Thread Pool에 관리될 것이다.

```
public class PrintTaskExecutor {
    private TaskExecutor executor;

    public PrintTaskExecutor(TaskExecutor taskExecutor) {
        this.executor = taskExecutor;
    }

    public void print() {
        for (int i = 0; i < 3; i++) {
            executor.execute(new Task(i));
        }
    }

    private class Task implements Runnable {
        private int no;

        public Task(int no) {
            this.no = no;
        }

        public void run() {
            System.out.println("execute a Task" + no + " at " + new Date()
                + " with TaskExecutor");
        }
    }
}
```

위 코드는 print() 메소드 내에서 Spring TaskExecutor를 활용하여 Inner 클래스로 정의된 Task를 실행
하는 PrintTaskExecutor의 일부이다. PrintTaskExecutor의 print() 메소드를 호출하면 Thread 유형의 내부
Task에 구현된 run() 메소드가 3회 실행되는 것을 확인할 수 있을 것이다.

다음은 위에서 언급한 PrintTaskExecutor에 대한 속성 정의 내용의 일부이다.

```
<task:executor id="executor" pool-size="4" queue-capacity="4" rejection-policy="ABORT"/>

<bean id="task" class="org.anyframe.sample.scheduling.task.executor.PrintTaskExecutor">
    <constructor-arg ref="executor"/>
</bean>
```

```
</bean>
```

위에서 언급한 `PrintTaskExecutor` 샘플 코드는 본 섹션 내의 다운로드 - `anyframe-sample-scheduling`을 통해 다운로드받을 수 있다.

1.2.TaskScheduler

다음은 Spring 3에서 새롭게 제공하고 있는 `org.springframework.scheduling.TaskScheduler` 클래스의 일부 내용이다.

```
public interface TaskScheduler {
    ScheduledFuture schedule(Runnable task, Trigger trigger);

    ScheduledFuture schedule(Runnable task, Date startTime);

    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);
}
```

`TaskScheduler`는 Execution 대상이 되는 `Task`를 특정 시점 이후에 한 번 실행하거나 `fixedRate` 또는 `fixedDelay` 정보를 기반으로 주기적으로 실행할 수 있는 메소드를 제공하고 있다.

1.3.XML based Scheduling

Spring 3에서는 앞서 언급한 `TaskExecutor`(`<task:executor/>`)나 `TaskScheduler`(`<task:scheduler/>`)에 대한 속성 정의를 위해 `task`라는 Namespace를 제공한다. 또한 이를 이용하면 간편하게 `Task Scheduling`(`<task:scheduled-task/>`)을 위한 속성을 정의할 수 있게 된다. `task` Namespace를 사용하기 위해서는 해당 XML 파일 내의 `<beans>` 정의시 `spring-task.xsd`를 선언해 주어야 한다.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:task="http://www.springframework.org/schema/task"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/task
    http://www.springframework.org/schema/task/spring-task.xsd">
  ...
</beans>
```

다음은 `<task:scheduler/>`를 사용한 속성 정의의 일부이다. 다음과 같이 속성을 정의한 경우 정의된 `Pool Size`를 기반으로 `ThreadPoolTaskScheduler` 인스턴스가 생성될 것이다. 정의된 `id`는 `Pool`에 관리될 `Task Thread`의 Prefix로 사용된다.

```
<task:scheduler id="scheduler" pool-size="10"/>
```

다음은 `<task:executor/>`를 사용한 속성 정의의 일부이다. 다음과 같이 속성을 정의한 경우 `ThreadPoolTaskExecutor` 인스턴스가 생성될 것이다. 또한 정의된 `id`는 `Pool`에 관리될 `Task Thread`의 Prefix로 사용된다.

```
<task:executor id="executor" pool-size="4" queue-capacity="4" rejection-policy="ABORT"/>
```

<task:executor/>는 <task:scheduler/>에 비해 다양한 속성 정의를 지원한다. 다음에서는 정의 가능한 속성들에 대해 자세히 살펴보도록 하자.

Attribute	Description
pool-size	Thread Pool의 Size를 결정한다. 단일값으로 정의하였을 경우 Pool Size가 정의된 크기로 고정된다. min-max 형태로 정의하였을 경우 Pool Size의 범위가 지정된다.
queue-capacity	현재 실행 중인 Thread의 개수가 지정된 최소 Pool Size보다 작을 경우, TaskExecutor는 실행 대상 Task에 대해 Free Thread를 사용한다. 점점 실행 대상 Task가 증가하여 현재 실행 중인 Thread의 개수가 지정된 최소 Pool Size와 같아지는 경우, 실행 대상 Task는 Queue에 추가된다. 이 때 추가 가능한 Task의 개수는 queue-capacity와 동일하다. 정의된 Queue Capacity를 모두 사용하게 된다면 TaskExecutor는 실행 대상 Task에 대해 New Thread를 생성하여 Pool에 추가하게 된다. 현재 실행 중인 Thread의 개수가 지정된 최대 Pool Size를 초과하는 경우 비로소 TaskExecutor는 해당 Task 실행을 거부하게 된다. 이와 같이 pool-size는 queue-capacity와 같이 고려되어야 하는 속성 정보이며 pool-size와 queue-capacity의 상관 관계에 대한 자세한 정보는 ThreadPoolExecutor API [http://java.sun.com/javase/6/docs/api/java/util/concurrent/ThreadPoolExecutor.html]를 참조하도록 한다. queue-capacity 값을 정의하지 않는 경우 한계값이 정해지지 않으나 이 경우 너무 많은 실행 대상 Task가 Queuing 됨으로 인해 OutOfMemoryErrors를 초래할 수 있음에 유의하도록 한다. 또한 Queue Capacity에 대한 최대값이 존재하지 않으므로 Queue가 Full이 되는 상태가 발생하지 않아 결국 최대 Pool Size 또한 의미가 없어지게 된다.
keep-alive	최소 Pool Size 초과로 생성된 Inactive Thread에 대해 keep-alive 값으로 지정한 시간이 지난 후에 timeout된다. 만일 TaskExecutor의 pool-size가 범위로 정의되어 있고, queue-capacity가 정의되어 있지 않는다면, Pool Size가 최소 크기를 넘지 않았을 경우라도 해당 Pool에 포함된 Inactive Thread에 대해서 timeout을 적용하게 된다. (초 단위로 지정 가능)
rejection-policy	기본적으로 Task 실행이 거부되었을 경우 TaskExecutor는 TaskRejectedException을 throw하게 된다. 그러나 rejection-policy 값을 다음과 같이 정의하는 경우 정의된 Policy에 의해 다른 결과를 보여줄 수 있다. <ul style="list-style-type: none"> • ABORT : AbortPolicy 적용. rejection-policy가 정의되지 않았을 경우 기본 적용되는 Policy로 Exception을 throw한다. • CALLER_RUNS : CallerRunsPolicy 적용. 해당 Application이 과부하 상태일 경우 TaskExecutor에 의해서가 아닌 Thread에서 직접 Task를 실행시킬 수 있게 한다. • DISCARD : DiscardPolicy 적용. 모든 Task가 반드시 실행되어야 한다는 제약점이 없는 경우 적용 가능한 Policy로써 해당 Application이 과부하 상태일 경우 현재 Task의 실행을 Skip한다. • DISCARD_OLDEST : DiscardOldestPolicy 적용. 모든 Task가 반드시 실행되어야 한다는 제약점이 없는 경우 적용 가능한 Policy로써 해당 Application이 과부하 상태일 경우 Queue의 Head에 있는 Task의 실행을 Skip한다.

다음은 task Namespace의 가장 강력한 특징인 <task:scheduled-task/>를 사용한 속성 정의의 일부이다. <task:scheduled-task/>는 기본적으로 'scheduler'라는 속성을 가지고 있는데 이것은 내부에 정의된 Task를 Scheduling하기 위한 TaskScheduler Bean을 정의하기 위한 것이다. <task:scheduled-task/>는 하위에 다수의 <task:scheduled/>를 포함할 수 있으며 <task:scheduled/>의 'ref'와 'method'는 실행 대상이 되는 Bean과 해당 Bean 내에 포함된 실행 대상 메소드를 정의하기 위한 속성이다.

```
<task:scheduled-tasks scheduler="scheduler">
  <task:scheduled ref="task" method="printWithFixedDelay" fixed-delay="5000"/>
  <task:scheduled ref="task" method="printWithFixedRate" fixed-rate="10000"/>
  <task:scheduled ref="task" method="printWithCron" cron="*/8 * * * * MON-FRI"/>
</task:scheduled-tasks>
```

```
<task:scheduler id="scheduler" pool-size="10"/>
```

<task:scheduled/>는 'ref', 'method' 외에 Scheduling을 위해 필요한 속성을 가지는데 각각에 대해 알아보면 다음과 같다.

Attribute	Description
cron	<p>Cron Expression을 이용하여 Task 실행 주기 정의.</p> <p>Cron Expression은 6개의 Field로 구성되며 각 Field는 순서대로 second, minute, hour, day, month, weekday를 의미한다. 각 Field의 구분은 Space로 한다. 또한 month와 weekday는 영어로 된 단어의 처음 3개의 문자로 정의할 수 있다.</p> <ul style="list-style-type: none"> • 0 0 * * * * : 매일 매시 시작 시점 • */10 * * * * * : 10초 간격 • 0 0 8-10 * * * : 매일 8,9,10시 • 0 0/30 8-10 * * * : 매일 8:00, 8:30, 9:00, 9:30, 10:00 • 0 0 9-17 * * MON-FRI : 주중 9시부터 17시까지 • 0 0 0 25 12 ? : 매년 크리스마스 자정 <p>* org.springframework.scheduling.support.CronSequenceGenerator API 참조</p>
fixed-delay	이전에 실행된 Task의 종료 시간으로부터의 fixed-delay로 정의한 시간만큼 소비한 이후 Task 실행. (Milliseconds 단위로 정의)
fixed-rate	이전에 실행된 Task의 시작 시간으로부터 fixed-rate로 정의한 시간만큼 소비한 이후 Task 실행. (Milliseconds 단위로 정의)

위에서 언급한 PrintTaskExecutor 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

1.4.Annotation based Scheduling & Asynchronous Execution

Spring 3에서는 Task Scheduling(@Scheduled)과 Asynchronous Task Execution(@Async)을 위한 Annotation을 제공한다. 이 Annotation들을 인식할 수 있도록 하기 위해서는 다음과 같은 속성 정의가 추가되어야 한다.

```
<task:annotation-driven scheduler="scheduler" executor="executor"/>
```

다음에서는 @Scheduled와 @Async Annotation에 대해 살펴보기로 하자.

1.4.1.Scheduling

@Scheduled는 메소드 단위로 사용 가능하며 실행 주기 정의를 위한 'fixedDelay', 'fixedRate', 'cron'과 같은 속성들을 제공하고 있다. 각 속성의 의미는 XML based Scheduling에서 언급한 <task:scheduled/> 속성과 동일한 의미를 가진다. @Scheduled 메소드에 대한 실행은 TaskScheduler가 담당한다.

```
@Scheduled(fixedDelay=5000)
```

```

public void printWithFixedDelay() {
    System.out.println("execute printWithFixedDelay() of Annotated PrintTask at "
        + new Date());
}

@Scheduled(fixedRate=10000)
public void printWithFixedRate() {
    System.out.println("execute printWithFixedRate() of Annotated PrintTask at "
        + new Date());
}

@Scheduled(cron="*/8 * * * * MON-FRI")
public void printWithCron() {
    System.out.println("execute printWithCron() of Annotated PrintTask at "
        + new Date());
}

```

@Scheduled Annotation을 부여한 메소드는 입력 인자를 갖지 않고, Return 값이 없어야 함에 유의하도록 한다. 또한 메소드 로직 실행을 위해 다른 Bean을 참조로 해야 한다면 Dependency Injection에 의해 처리하도록 한다.

1.4.2. Asynchronous Execution

@Async는 메소드 단위로 사용 가능하며 비동기적으로 특정 메소드를 실행하고자 할 때 사용할 수 있다. @Async 메소드에 대한 실제적인 실행은 TaskExecutor에 의해 처리된다.

```

@Async
public void printWithAsync() throws Exception {
    System.out.println("execute printWithAsync() of AsyncPrintTask at "
        + new Date());
    Thread.sleep(5000);
}

@Async
public void printWithArg(int i) throws Exception {
    System.out.println("execute printWithArg(" + i + ") of AsyncPrintTask at "
        + new Date());
    Thread.sleep(5000);
}

@Async
public Future<String> returnVal(int i) throws Exception {
    System.out.println("execute returnVal() of AsyncPrintTask");
    Date current = new Date();
    Thread.sleep(5000);
    return new AsyncResult<String>(Integer.toString(i));
}

```

위 코드에서와 같이 @Async 메소드는 @Scheduled 메소드와 다르게 입력 인자나 리턴값을 가질 수 있다. @Scheduled의 경우에는 Spring Container에 의해 관리되는 반면에 @Async는 Caller에 의해 직접 호출되기 때문이다. 단, 리턴값의 경우 Future 타입 형태만 가능하며 Caller는 비동기적으로 실행 종료된 메소드의 결과를 Future 객체의 get() 메소드를 통해 알아낼 수 있다.

위에서 언급한 PrintTaskExecutor 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

1.5. Resources

- 다운로드

다음에서 sample 코드를 포함하고 있는 Eclipse 프로젝트 파일을 다운받은 후, 압축을 해제한다.

- Maven 기반 실행

Command 창에서 압축 해제 폴더로 이동한 후, `mvn compile exec:java -Dexec.mainClass=...`이라는 명령어를 실행시켜 결과를 확인한다. 각 Eclipse 프로젝트 내에 포함된 Main 클래스의 JavaDoc을 참고하도록 한다.

- Eclipse 기반 실행

Eclipse에서 압축 해제 프로젝트를 import한 후, `src/main/java` 폴더 하위의 `Main.java`를 선택하고 마우스 오른쪽 버튼 클릭하여 컨텍스트 메뉴에서 `Run As > Java Application`을 클릭한다. 그리고 실행 결과를 확인한다.

표 1.1. Download List

Name	Download
anyframe-sample-scheduling.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/optional/scheduling/1.0.3/reference/sample/anyframe-sample-scheduling.zip]

III.Quartz Scheduling

Spring은 JDK 1.30이후 제공되는 Timer Scheduler와 Quartz Scheduler(<http://www.opensymphony.com/quartz/>)를 이용한 스케줄링을 지원한다.

Quartz는 오픈 소스 작업 스케줄링 프레임워크이다. Quartz는 완전히 자바로 작성되어 있으며 매우 유연하고 단순한 구조를 제공하여 간단한 작업은 물론 복잡한 작업 모두에 대한 스케줄링을 작성할 수 있다. 또한 EJB, JavaMail 등을 위한 데이터베이스 지원, 클러스터링, 플러그 인, 미리 내장된 작업들을 포함하고 있다.

2.Quartz Integration

Anyframe 에서는 JDK의 Timer Scheduler보다 **Quartz Scheduler** 사용을 추천한다. Quartz Scheduler 는 JDK Timer Scheduler보다 더 유동성있고 성능이 좋다. 자세한 내용은 Spring 매뉴얼의 Scheduling 부분 [<http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/scheduling.html>]을 참고하도록 한다.

2.1.Quartz Scheduler

Quartz는 Job과 Trigger 그리고 JobDetail을 사용하여 스케줄링 기능을 수행한다. Quartz에 대한 자세한 내용은 <http://www.opensymphony.com/quartz> [<http://www.opensymphony.com/quartz/>]를 참고한다.

- **Job**은 실행해야 할 작업으로, 자동 문자메시지 전송 기능이나 어플리케이션의 자동화된 작업들이 그 예이다. Job은 언제 실행되는지에 대한 정보 없이 실행 작업의 단위로만 작성된다. Job은 Spring Framework에서 제공하는 QuartzJobBean을 상속받아서 작성할 수도 있고, 특정 API에 종속되지 않은 POJO 형태의 자바 클래스로 작성할 수 있다. 또한 **Stateful한 Job** 클래스를 사용하기 위해서 Quartz 에서 제공하는 StatefulJob 인터페이스를 구현하여 Job을 작성할 수도 있다. StatefulJob 인터페이스 를 구현하여 작성된 Job 클래스 이외의 모든 Job 클래스는 기본적으로 **Stateless Job** 클래스로 동작 한다. Stateful한 Job의 경우에만 반복되는 Job 수행 시 특정 데이터 값을 공유하여 변경할 수 있다.
- **Trigger**는 Job을 실행시키기 위한 조건으로 작업 실행 시간, 반복 횟수 그리고 실행 간격 시간 등이 조건에 해당된다. 다수의 Trigger는 동일한 Job을 공유하여 지정할 수 있으나, 하나의 Trigger는 반드시 하나의 Job을 지정해야 한다.
- **JobDetail**은 Job을 실행하기 위해 필요한 정보를 가지고 있는, 즉 Job Instance에 대한 상세 속성 정보를 가지고 있는 객체로 Trigger가 JobDetail을 이용하여 Job을 수행시킴으로써 Quartz 기반의 스케 줄링 기능이 수행된다.
- **Scheduler**는 SchedulerFactory에 의해 생성되는데, JobDetail들과 Trigger들을 관리하며 해당 Trigger 에 연관된 Job을 수행시키는 Scheduling 서비스의 핵심 역할을 담당한다.

아래 예제에서는 JobDetailBean, MethodInvokingJobDetailFactoryBean, SimpleTriggerBean, CronTriggerBean, TriggerListener, SchedulerFactoryBean 등을 이용하여 일정한 시간 간격 마다 특정 메 소드를 실행시키는 방법을 보여주고 있다.

- **JobDetailBean** : Spring Framework은 JobDetailBean이라고 불리는 클래스를 제공하는데, Job을 실행 시키기 위해 필요한 정보를 가지고 있다.
- **MethodInvokingJobDetailFactoryBean** : Job API에 종속적이지 않게 POJO 형태의 Job 클래스를 작 성하여 Scheduler에 따라 해당 클래스의 메소드를 호출할 수 있도록 해주는 역할을 수행한다. Job 과 Trigger에 관련된 정보를 DB를 통해 관리하는 경우, 즉 JDBC Job Store 방식을 사용하는 경우에는 MethodInvokingJobDetailFactoryBean이 동작하지 않으므로 사용할 수 없음에 유의하도록 한다.
- **SimpleTriggerBean, CronTriggerBean** : Spring Framework에서 Quartz를 손쉽게 사용할 수 있도록 2 개의 TriggerBean을 제공한다. SimpleTriggerBean과 CronTriggerBean은 Trigger로써 동작하는 것은 동 일하나 CronTriggerBean의 경우, 시간 실행 조건을 cron expression을 이용하여 작성하는 것이 차이 점이다. SimpleTrigger보다 상세한 스케줄링 실행 시간 조건을 설정할 수 있으면서도 설정 방법이 복잡하지 않다.
- **TriggerListener** : Trigger가 fire->execute->complete 혹은 misfire되는 시점에 특정 일을 수행하고자 할때 Scheduler에 TriggerListener를 설정한다.
- **SchedulerFactoryBean** : 사용되는 Trigger를 등록시켜주는 역할을 담당하므로 Trigger들을 triggers 속 성의 <list> 태그 하위로 등록시킨다. Trigger에 적용할 Listener 등 여러 속성들을 추가 설정할 수 있다.

위에 나열한 Bean들의 주요 속성 정의 항목부터 살펴보도록 하자. 다음은 **JobDetailBean**의 주요 속성으로 정의되어야 할 항목들에 대한 설명이다.

Property Name	Description	Required	Default Value
jobClass	Job의 클래스명을 설정한다.	Y	N/A
jobDataAsMap	이 속성값으로 설정된 데이터들은 Job 클래스 내에서 사용가능한 데이터로 Map 형태로 정의한다. Job과 연관된 Trigger에 설정된 jobDataAsMap 속성값으로 설정한 Map 데이터와 JobDetail에 설정된 jobDataAsMap 속성값으로 설정한 Map 데이터는 함께 merge되어 사용될 수 있다. 이 데이터들은 기본적으로 처음 한번 설정된 뒤 변경되지 않으나, Stateful한 Job 객체를 실행시켜서 데이터 값을 변경시키게 되는 경우에는 변경이 가능하다.	N	N/A

다음은 **MethodInvokingJobDetailFactoryBean**의 주요 속성으로 정의되어야 할 항목들에 대한 설명이다.

Property Name	Description	Required	Default Value
name	Job의 이름을 설정한다.	FactoryBean의 Bean name	N/A
targetObj	POJO로 작성된 Job 클래스명을 정의한다.	Y	N/A
targetMethod	targetObj에서 설정한 Object에서 실행하고자 하는 method 명을 입력한다.	Y	N/A
concurrent	다수의 Job을 동시에 실행시키려면 true로 설정하고 순차적으로 실행시키려면 false로 설정한다.	N	true

다음은 **SimpleTriggerBean**의 주요 속성으로 정의되어야 할 항목들에 대한 설명이다.

Property Name	Description	Required	Default Value
jobDetail	이 Trigger와 관련된 JobDetail의 클래스를 작성하는데, Spring Bean으로 작성된 경우 ref attribute를 이용하여 Bean의 id를 작성하도록 한다.	Y	N/A
repeatInterval	Trigger가 처음 수행 이후 반복 수행 시킬 Trigger의 interval 시간을 설정하는 값으로 Trigger가 반복되는 interval 시간을 milliseconds 단위로 설정한다. repeatInterval은 0이상이어야 한다.	N	0
repeatCount	Trigger가 처음 수행 이후 반복 수행 시킬 Trigger의 수행 횟수를 설정한다.	Y	0
triggerListenerNames	특정 trigger에만(즉, Non-global) TriggerListener를 설정하여 Trigger가 fire->execute->complete 혹은 misfire되는 시점에 특정 일을 수행하고자 할때 TriggerListener를 설정하는데 여기에는 triggerListenerName을 등록한다. 이 triggerListenerName은 TriggerListener 인터페이스를 구현한 클래스의 getName() 메소드에서 리턴한 이름을 작성하도록 한다.	N	N/A
jobDataAsMap	이 속성값으로 설정된 데이터들은 Trigger에 설정된 JobDetail을 통해 Job에서 해당 데이터를 이용할 수 있다.	N	N/A

Property Name	Description	Required	Default Value
	다. Trigger에 설정된 jobDataAsMap 속성값으로 설정한 Map 데이터와 JobDetail에 설정된 jobDataAsMap 속성값으로 설정한 Map 데이터는 함께 merge되어 사용될 수 있다. 이외에도 이 Trigger에 설정된 triggerListener 내부에서도 이 데이터를 이용할 수 있다. 이 데이터들은 기본적으로 처음 한번 설정된 뒤 변경되지 않으나, Stateful한 Job 객체를 실행시켜서 데이터 값을 변경시키게 되는 경우에는 변경이 가능하다.		
startTime	Trigger 수행 시작 시간을 설정한다.	N	N/A
startDelay	처음 Job이 시작되기 전 지연 시간을 설정한다. 입력한 milliseconds 단위의 시간에 현재 시간을 더하면 시작 시간이 된다. startDelay 속성은 startTime 속성이 지정되지 않은 경우에 적용된다. 그러나 Spring Container를 구동시키면, startTime은 언제나 Container가 구동된 시간 이므로 이러한 경우에는 절대적인 시간이 아닌 상대적인 시간을 설정하도록 한다.	N	0
endTime	Trigger 반복을 종료할 시간을 입력한다.	N	N/A
misfireInstructionName	Trigger는 Scheduler에 문제가 발생하였거나 Job을 수행시키는 Quartz의 Thread pool에서 현재 사용 가능한 Thread가 존재하지 않을 때 실행되지 못하는 경우가 있다. 이러한 경우 Job Store 방식으로 DB를 사용하고 있다면, 다음 Scheduler가 정상 동작 시 실행되지 못한 Job들에 대해서 misfire 시킨다. (하위 Advanced Quartz >> Job Stores 내용 참고) 기본적으로 'smart policy' instruction이 사용되고, 그외 다음과 같은 misfire instruction명을 설정할 수 있다. 자세한 내용은 Quartz Tutorial >> Misfire Instructions [http://www.opensymphony.com/quartz/wikidocs/TutorialLesson4.html]를 참조하도록 한다. <ul style="list-style-type: none"> 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 이 속성 설정 시, 아래 소개된 SchedulerFactoryBean 속성 중 dataSource 속성 값도 설정하도록 한다.	N	MISFIRE_INSTRUCTION_SMART_POLICY

다음은 **CronTriggerBean** 의 주요 속성으로 정의되어야 할 항목들에 대한 설명이다.

Property Name	Description	Required	Default Value
jobDetail	이 Trigger와 관련된 JobDetail의 클래스를 작성하는데, Spring Bean으로 작성된 경우 ref attribute를 이용하여 Bean의 id를 작성하도록 한다.	Y	N/A
cronExpression	Cron-Expression은 기본적으로 다음과 같이 7개 필드로 구성되어 있다.	N	0

Property Name	Description	Required	Default Value
	<ul style="list-style-type: none"> Seconds (Allowed Values : 0-59, Allowed Special Characters : , - * /) Minutes (Allowed Values : 0-59, Allowed Special Characters : , - * /) Hours (Allowed Values : 0-23, Allowed Special Characters : , - * /) Day-of-month (Allowed Values : 1-31, Allowed Special Characters : , - * ? / L W) Month (Allowed Values : 1-12 or JAN-DEC, Allowed Special Characters : , - * /) Day-of-week (Allowed Values : 1-7 or SUN-SAT, Allowed Special Characters : , - * ? / L #) Year (optional field) (Allowed Values : empty, 1970-2199, Allowed Special Characters : , - * /) <p>예를 들어, cron expression이 0 0 6 * * ? 으로 설정되었다면 매일 오전 6시에 실행시키라는 의미로 설정된 것이다.</p>		
startTime	Trigger 수행 시작 시간을 설정한다.	N	N/A
endTime	Trigger 반복을 종료할 시간을 입력한다.	N	N/A

다음은 **SchedulerFactoryBean**의 주요 속성으로 정의되어야 할 항목들에 대한 설명이다.

Property Name	Description	Required	Default Value
triggers	Scheduler에 사용되는 Trigger들을 등록시킨다. 여러 개의 Trigger를 등록시켜야 하므로 <list> 태그를 이용하여 설정한다.	Y	N/A
triggerListeners	특정 Trigger에 한하여 적용할 TriggerListener를 등록한다.	N	N/A
globalTriggerListeners	해당 Scheduler에 등록된 모든 Trigger들에게 적용할 TriggerListener를 등록한다. 이를 GlobalTriggerListener라고 한다.	N	N/A
dataSource	이 속성 값을 설정하게 되면, Job과 Trigger에 관련된 정보가 자동으로 DB를 통해 관리된다. 이때 저장할 DB Table이 준비되어 있어야 하는데, Table은 Quartz를 다운로드 받은 후 docs/dbTables 폴더 내의 table-creation SQL script 중 사용하고자 하는 DB(ex. Oracle의 경우, tables_oracle.sql)에 맞는 script를 수행하여 생성시키도록 한다.	N	N/A

2.1.1. Advanced Quartz

이 매뉴얼에 소개된 내용을 통해서도 Quartz의 많은 기능을 사용할 수 있으나, 이외에도 여러 고급 기능 등이 존재한다. 여기서는 Job과 Trigger에 관련된 정보를 저장하는 방법에 대해서 소개한다. 더욱 자세한 내용은 <http://www.opensymphony.com/quartz> [<http://www.opensymphony.com/quartz/>]를 참고한다.

• **Job Stores**

Quartz는 Job과 Trigger에 관련된 정보를 2가지 서로 다른 방식으로 저장할 수 있게 한다. 기본적으로는 메모리에 저장되는데, DB에 저장하도록 선택할 수도 있다. **기본(Default)적으로 RAMJobStore 클래스를 이용하여 Memory에 저장되는데**, 이 Job Store 방식을 사용할 때 모든 데이터가 Memory에 저장되기 때문에 성능이 가장 좋다. 그러나, 어플리케이션이나 시스템에 문제가 발생하면 RAM에 저장되어있던 모든 데이터가 손실될 위험성이 있다.

위의 Memory 저장 방식의 문제점을 해결하기 위해 Quartz는 **JDBCJobStore 클래스를 이용하여 데이터를 DB에 저장** 하도록 한다. 모든 데이터가 JDBC를 통해서 DB에 저장되므로 어플리케이션이나 시스템에 문제가 발생해도 데이터가 손실될 위험은 없으나 성능이 좋지 않고 복잡성이 증가한다. 데이터를 DB에 저장하는 방식을 사용하기 위해서는 위의 SchedulerFactoryBean 속성 중 dataSource 속성 값을 설정해줘야 하는데 dataSource 속성 설정 방법은 Core Plugin >> Spring >> DataSource [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html#core_spring_datasource]를 참고한다.

Job과 Trigger에 관련된 정보를 DB를 통해 관리하기 위해서는 DB Table이 준비되어 있어야 하는데, Table은 Quartz를 다운로드 받은 후 docs/dbTables 폴더 내의 table-creation SQL script 중 사용하고 자 하는 DB(ex. Oracle의 경우, tables_oracle.sql)에 맞는 script를 수행하여 생성시키도록 한다. 더욱 자세한 내용과 사용 방법은 Quartz Configure DataSources [<http://www.opensymphony.com/quartz/wikidocs/ConfigDataSources.html>]를 참고하도록 한다.

2.1.2.Samples

다음은 서로 다른 3개의 Job(jobDetail01, jobDetail02, jobDetail03)을 서로 다른 3개의 Trigger(simpleTrigger01, simpleTrigger02, cronTrigger)에 연관시킨 후, 3개의 Trigger들을 Scheduler에 등록시켜서 Job 스케줄링을 수행시킨 예이다. 수행 시간 조건에 따라 일정 시간 마다 반복적인 일을 하는데, 이때 Scheduler에 TriggerListener (non-global)와 GlobalTriggerListener을 등록하여 Trigger의 수행 순서 모습을 모니터링할 수 있게 하였다.

• **JobDetailBean with Stateless Job**

다음은 Spring Framework에서 제공하는 QuartzJobBean을 상속받아서 작성한 Job을 Job 클래스로 가지고 있는 JobDetailBean인 **jobDetail01**을 정의한 Spring 속성 정의 파일(context-scheduling.xml)의 일부이다. JobDetailBean의 jobDataAsMap 속성으로 설정한 데이터값은 Job들 사이에 공유되어 사용될 수 있으나 Stateless한 Job이므로 반복 수행되는 Job에서 값을 변경시킬 수는 없다.

```

<!-- JobDetailBean -->
<bean id=<emphasis role="bold">"jobDetail01"</emphasis>
  class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass"
  value="org.anyframe.sample.scheduling.SimpleQuartzJobBean" />
  <property name="jobDataAsMap">
    <map>
      <entry>
        <key>
          <value>count</value>
        </key>
        <value>6</value>
      </entry>
    </map>
  </property>
</bean>

```

다음은 Spring Framework에서 제공하는 QuartzJobBean을 상속받아서 작성한 테스트 용 SimpleQuartzJobBean 클래스의 구현 모습이다. Spring의 QuartzJobBean 클래스를 상속받으면 JobExecutionContext를 argument로 갖는 executeInternal 메소드를 작성하여 스케줄링 일정에 따라 반복적으로 수행될 작업을 기술한다. 이 예제에서는 JobDetailBean의 jobDataAsMap 속성으로 지정

한 count 값을 변경시켜보려고 하지만, Stateless한 Job이기 때문에 처음 Job 수행 이후 반복 수행되는 Job들에서는 값이 변경되지 않는다.

```
public class SimpleQuartzJobBean extends QuartzJobBean {
    public void setCount(int count) {
        this.count=count;
    }

    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        this.count= this.count + 1;
        context.getJobDetail().getJobDataMap().put("count",this.count);
        System.out.println("count="+this.count);
    }
}
...종략
```

위에서 언급한 SimpleQuartzJobBean, context-scheduling.xml 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

• JobDetailBean with StatefulJob

다음은 Quartz에서 제공하는 StatefulJob 인터페이스 클래스를 구현하여 작성한 Job을 Job 클래스로 가지고 있는 JobDetailBean인 **jobDetail02**을 정의한 Spring 속성 정의 파일(context-scheduling.xml)의 일부이다. JobDetailBean의 jobDataAsMap 속성으로 설정한 데이터값은 Job들 사이에 공유되어 사용될 수 있고, Stateful한 Job이므로 반복 수행되는 Job에서 값을 변경시킬 수 있다.(예제에서 count 라는 값을 1씩 증가시키고 있다.)

```
<!-- JobDetailBean -->
<bean id=<emphasis role="bold">"jobDetail02"</emphasis>
    class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="jobClass"
    value="org.anyframe.sample.scheduling.StatefulQuartzJobBean"/>
    <property name="jobDataAsMap">
        <map>
            <entry>
                <key>
                    <value>count</value>
                </key>
                <value>6</value>
            </entry>
        </map>
    </property>
</bean>
```

다음은 Quartz에서 제공하는 StatefulJob 인터페이스 클래스를 구현하여 작성한 테스트 용 StatefulQuartzJobBean 클래스의 구현 모습이다. StatefulJob 인터페이스 클래스의 메소드인 execute 메소드를 아래와 같이 JobExecutionContext를 argument로 갖도록 구현하고, 메소드 내부에서는 executeInternal 메소드를 호출하도록 한다. 이 예제에서는 JobDetailBean의 jobDataAsMap 속성으로 지정한 count 값을 변경시켜서 수행되는 동일한 Job들 사이에 값을 공유시키고 있다.

```
import org.quartz.StatefulJob;
public class StatefulQuartzJobBean implements StatefulJob {
    public void setCount(int count) {
        this.count=count;
    }

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        try {
```

```

        BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
        MutablePropertyValues pvs = new MutablePropertyValues();
        pvs.addPropertyValues(context.getScheduler().getContext());
        pvs.addPropertyValues(context.getMergedJobDataMap());
        bw.setPropertyValues(pvs, true);
    } catch (SchedulerException ex) {
        throw new JobExecutionException(ex);
    }
    executeInternal(context);
}

public void executeInternal(JobExecutionContext context)
    throws JobExecutionException {
    this.count = count + 1;
    context.getJobDetail().getJobDataMap().putAsString("count", count);
    System.out.println(this.getClass().getName() + ":@" + this.count);
    result = count;
}

...중략

```

위에서 언급한 StatefulQuartzJobBean, context-scheduling.xml 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

- **MethodInvokingJobDetailFactoryBean**

다음은 Spring Framework에서 제공하는 MethodInvokingJobDetailFactoryBean을 그대로 재사용하여 POJO 형태로 작성한 job을 Job 클래스로 가지고 있는 JobDetail인 **productSummaryJob**을 정의한 Spring 속성 정의 파일(context-scheduling.xml)의 일부이다.

```

<bean name="job" class="org.anyframe.plugin.scheduling.job.MovieStatusSummaryJob" init-
method="initialize">
    <constructor-arg index="0" ref="schedulingGenreService" />
    <constructor-arg index="1" ref="queryService" />
</bean>

<bean id="movieSummaryJob"
class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="job" />
    <property name="targetMethod" value="execute" />
    <property name="concurrent" value="false" />
</bean>

```

다음은 Job 관련 API에 종속되지 않고 POJO로 작성한 ProductStatusSummaryJob 클래스의 구현 모습이다. Trigger에 의해서 ProductStatusSummaryJob 클래스의 execute() 메소드가 수행될 것이다. execute() 메소드 내에 스케줄링 일정에 따라 반복적으로 수행될 작업을 기술한다.

```

public class MovieStatusSummaryJob {
    public void execute() throws Exception {
        removeAllMonthlyMovieStatus();
        insertMonthlyMovieStatus();
    }
    ...중략
}

```

- **SimpleTriggerBean for Stateless Job**

다음은 Spring Framework에서 제공하는 SimpleTriggerBean를 이용하여 위에서 작성한 JobDetail 중 **jobDetail01**을 설정한 Spring 속성 정의 파일(context-scheduling.xml)의 일부이다.

처음 Job 수행 이후 반복 수행 시간의 간격은 1초로, 반복 횟수는 3회로 설정되었으며 해당 trigger에 만 적용시킬 triggerListener가 triggerListenerNames 속성 정보로 정의되어 있다. triggerListener Bean 설정은 하위 내용을 참고한다.

```
<!-- SimpleTriggerBean -->
<bean id="simpleTrigger01"
      class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail" ref="jobDetail01"/>
  <property name="repeatInterval" value="1000"/>
  <property name="repeatCount" value="3"/>
  <property name="triggerListenerNames" value="triggerListener"/>
</bean>
```

위에서 언급한 context-scheduling.xml 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

- **SimpleTriggerBean for StatefulJob**

다음은 Spring Framework에서 제공하는 SimpleTriggerBean를 이용하여 위에서 작성한 JobDetail 중 **jobDetail02** 을 설정한 Spring 속성 정의 파일(context-scheduling.xml)의 일부이다.

하나의 Trigger는 하나의 JobDetail을 설정해야 하므로 위의 SimpleTriggerBean과 동일한 속성 정보를 갖는 Trigger를 추가 작성한다.

처음 Job 수행 이후 반복 수행 시간의 간격은 1초로, 반복 횟수는 3회로 설정되었으며 해당 trigger에 만 적용시킬 triggerListener가 triggerListenerNames 속성 정보로 정의되어 있다. triggerListener Bean 설정은 하위 내용을 참고한다.

```
<!-- SimpleTriggerBean -->
<bean id="simpleTrigger02"
      class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail" ref="jobDetail02"/>
  <property name="repeatInterval" value="1000"/>
  <property name="repeatCount" value="3"/>
  <property name="triggerListenerNames" value="triggerListener"/>
</bean>
```

위에서 언급한 context-scheduling.xml 샘플 코드는 본 섹션 내의 다운로드 - anyframe-sample-scheduling을 통해 다운로드받을 수 있다.

- **CronTriggerBean**

다음은 Spring Framework에서 제공하는 CronTriggerBean를 이용하여 작성 Spring 속성 정의 파일 (context-scheduling.xml)의 일부이다.

아래 예제에서 보면 SimpleTriggerBean과 달리 cron expression을 통해 스케줄링 시간을 설정한 모습을 볼 수 있는데, 현재 1분마다 실행되도록 설정한 것이다.

```
<!-- CronTriggerBean -->
<bean id="cronTrigger"
      class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail" ref="movieSummaryJob"/>
  <property name="cronExpression" value="0 0/1 * * * ?" />
</bean>
```

- **TriggerListener Beans**

다음은 Quartz에서 제공하는 TriggerListener를 구현한 GlobalTriggerListener 클래스를 설정한 Spring 속성 정의 파일(context- scheduling.xml)의 일부이다. listenerType 속성을 "Global"로 설정하면 모든

Trigger에 적용된다. (미설정 시에는 Non-global로 설정된다.) 참고로 동일한 TriggerListener 구현 클래스를 Global 과 Non-Global로 동시에 함께 사용할 수도 있다.

```
<bean id="globalTriggerListener"
      class="anyframe.sample.scheduling.listener.GlobalTriggerListener">
  <property name="listenerType" value="Global"/>
</bean>
```

다음은 Quartz에서 제공하는 TriggerListener를 구현한 GlobalTriggerListener 클래스 구현 모습이다. Trigger가 fire->execute-> complete 혹은 misfire되는 시점에 로깅을 남기는 예제이다.

```
import org.quartz.TriggerListener;

public class GlobalTriggerListener implements TriggerListener {
    Log logger = LogFactory.getLog(GlobalTriggerListener.class);
    String listenerType = "Non global";

    public void setListenerType(String listenerType) {
        this.listenerType = listenerType;
    }

    public void triggerFired(Trigger trigger, JobExecutionContext ctx) {
        logger.info("Scheduled " + trigger.getJobName() + " Fired!!");
    }

    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext ctx) {
        logger.info("Scheduled " + trigger.getJobName() + " Executed!!");
        return false;
    }

    public void triggerComplete(Trigger trigger, JobExecutionContext ctx, int arg) {
        logger.info("Scheduled " + trigger.getJobName() + " Completed!!");
    }

    public void triggerMisfired(Trigger trigger) {
        logger.error("Scheduled " + trigger.getJobName() + " Misfired!!");
    }

    public String getName() {
        return "GlobalTriggerListener";
    }
}
```

- **SchedulerFactoryBean**

다음은 Spring Framework에서 제공하는 SchedulerFactoryBean를 이용하여 위에서 작성한 Trigger(cronTrigger)와 TriggerListener(globalTriggerListener)를 등록하는 Spring 속성 정의 파일 (context-scheduling.xml)의 일부이다.

```
<bean id="schedulerFactoryBean"
      class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref bean="cronTrigger" />
    </list>
  </property>
  <property name="globalTriggerListeners" ref="globalTriggerListener" />
</bean>
```

- **TestCase**

다음은 앞서 정의한 속성 설정을 기반으로 하여 schedule를 실행하는 MonthlyMovieStatusServiceTest.java 코드의 일부이다.

```
public void findMonthlyMovieStatusList() throws Exception {
    Collection monthlyMovieStatusList = monthlyMovieStatusService.getList();

    assertEquals("fail to fetch monthly movie status list", 10,
        monthlyMovieStatusList.size());

    Iterator itr = monthlyMovieStatusList.iterator();
    while (itr.hasNext()) {
        Map monthlyMovieStatus = (Map) itr.next();
        if (monthlyMovieStatus.get("genreId").equals("GR-01")) {
            assertEquals("fail to execute scheduling job", "0",
                monthlyMovieStatus.get("janCount").toString());
        }
    }

    addNewMovie();
    Thread.sleep(60000);

    monthlyMovieStatusList = monthlyMovieStatusService.getList();
    itr = monthlyMovieStatusList.iterator();
    while (itr.hasNext()) {
        Map monthlyMovieStatus = (Map) itr.next();
        if (monthlyMovieStatus.get("genreId").equals("GR-01")) {
            assertEquals("fail to execute scheduling job", "1",
                monthlyMovieStatus.get("janCount").toString());
        }
    }
}
```