Chapter 12 작업 실행과 스케줄링

: SPRING은 작업 실행 및 작업 Scheduling을 위한 Interface 구현 및 Class를 제공하고 있으며, 이를 통해 실제 사용하는 내부 기술에 상관없이 동일한

Interface를 이용해서 작업을 실행할 수 있게 해 주고 있다. 특히 SPRING 3.0 버전에 추가된 task 네임스페이스와 Annotation을 통해서 손쉽게 작업을

설정하고 실행할 수 있게 되었다.

본 장에서는 SPRING 3에 추가된 task 네임스페이스와 Annotation을 중심으로 TaskExecutor 및 TaskScheduler를 설정하고 작업을 실행하는 방법을

살펴보도록 하겠다.

1. <task:executor>를 이용한 작업 실행

: 작업 실행과 관련된 핵심 Interface는 TaskExecutor이다. TaskExecutor Interface 및 하위 Interface는 작업 실행과 관련된 Interface를 제공하고 있다.

SPRING 3버전은 TaskExecutor 빈을 손쉽게 설정할 수 있는 <task:executor> 태그를 제공하고 있다.

TaskExecutor를 이용해서 작업을 실행하려면 다음 같은 두 가지 Code를 작성하면 된다.

- <task:executor> 태그를 이용한 TaskExecutor 빈 설정

- TaskExecutor 빈 객체에 Runnable 구현 객체를 전달해서 작업 실행

각각의 내용을 차례대로 살펴보도록 하자.

1.1 <task:executor> 태그를 이용한 TaskExecutor 빈 설정

: <task:executor> 태그는 TaskExecutor 빈을 생성할 때 사용되는 태그로서, 다음과 같이 사용된다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:task="http://www.springframework.org/schema/task" xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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-3.0.xsd">

<task:executor id="executor" keep-alive="5" pool-size="1-2" queue-capacity="1" rejection-policy="ABORT" />

</beans>

<task:executor> 태그를 사용하기 위해서는 task 네임스페이스를 설정해 주어야 하며, id 속성을 이용서 빈 식별 값을 설정하게 된다.

<task:executor> 태그의 주요 속성은 아래와 같다.

속 성 설 명

================================================================================

id 생성할 빈의 식별 값을 지정한다.

================================================================================

pool-size 쓰레드 풀의 개수를 지정한다. '최소크기-최대크기'으로 지정하거나 '개수'를 지정한다. 예를 들어,

5-20으로 지정하거나 15와 같이 지정한다. '개수' 형식으로 지정할 경우, 최소 크기와 최대 크기가

동일한 값을 갖는다. 기본 값은 최소 크기가 1이고, 최대 크기는 Integer.MAX\_VALUE이다.

================================================================================

queue-capacity 작업이 저장되는 쓰레드의 최대 유효 시간. 이 시간 동안 새로운 작업을 실행하지 않으면 풀에서

제거된다. 기본값은 Integer.MAX\_VALUE이다.

================================================================================

keep-alive 풀에 있는 쓰레드의 최대 유효 시간. 이 시간 동안 새로운 작업을 실행하지 않으면 풀에서 제거된다.

단위는 초이다.

================================================================================

rejection-policy 큐가 다 차서 더 이상 작업을 받을 수 없을 때 작업을 어떻게 처리할지를 결정한다. 설정할 수 있는 값은

다음과 같다.

- ABORT : 작업을 거부하고 예외는 발생시킨다.

- CALLER\_RUNS : 호출한 쓰레드를 이용해서 실행한다.

- DISCARD : 작업을 실행하지않고 무시한다.

- DISCARD\_OLDEST : 큐의 헤드에서 하나를 제거하고 작업을 추가한다.

기본 값은 ABORT이다.

================================================================================

<task:executor> 태그를 사용하면 java.util.concurrent.ThreadPoolExecutor를 사용해서 작업을 실행하게 되는데, ThreadPoolExecutor는

다음과 같은 규칙을 이용해서 풀의 크기를 관리한다.

- 풀에 최소 크기 보다 작은 개수의 쓰레드가 존재할 경우, 쓰레드를 새롭게 생성한다.

- 풀에 최소 크기와 같거나 많은 개수의 쓰레드가 존재할 경우, 작업을 큐에 저장한다.

- 작업을 큐에 보관할 수 없을 경우, 풀에 최대 크기보다 작은 개수의 쓰레드가 존재할 경우 쓰레드를 새롭게 생성한다. 그렇지 않을 경우 작업을 거부한다.

위 규칙에 따라 보면 큐의 개수가 Integer.MAX\_VALUE인 경우, 풀에 객체를 저장하는 과정에서 메모리 부족 현상이 발생할 수도 있다. 또한, 쓰레드의

최대 개수가 Integer.MAX\_VALUE인 경우 불필요하게 많은 쓰레드가 생성되어 오히려 전반적인 처리 속도가 저하될 수 있다. 따라서, 풀의 최대/최소 크기와

개수는 처리할 작업의 상태에 따라 알맞은 개수를 지정해 주어야 한다.

1.2 TaskExecutor/AsyncTaskExecutor Interface를 이용한 작업 실행

: <task:executor> 태그를 이용해서 TaskExecutor 빈을 설정했다면, 해당 빈을 이용해서 작업을 실행할 수 있다. <task:executor> 태그는

org.springframework.scheduling.concurrent 패키지에 위치한 ThreadpoolTaskExecutor 객체를 생성하는데, ThreadPoolTaskExecutor Class가 상속받는

Interface 계층구조는 다음과 같다.

<<Interface>>

TaskExecutor

+execute(task : Runnable)

||

||

<<Interface>>

AsyncTaskExecutor

+execute(task : Runnable. startTimeout : long)

+submit(task : Runnable) : Future<?>

+submit(task : Callable<T>) : Future<T>

||

||

<<Interface>>

SchedulingTaskExecutor

+prefersShortLivedTasks() : boolean

||

||

ThreadPoolTaskExecutor

ThreadPoolTaskExecutor Class가 구현하고 있는 TaskExecutor Interface와 AsyncTaskExecutor Interface는 각각 작업을 실행하는데 필요한 메서드를

정의하고 있으며, 각 메서드는 다음과 같다.

- void execute(Runnable task) : task를 실행한다.

- void execute(Runnable task, long startTimeout) : task를 실행한다. 지정한 시간 이내에 작업이 시작되지 않을 경우 TaskRejectedException 예외를 발생시킨다.

- Future<?> submit(Runnable task) : task를 실행한다. Future를 통해 작업이 완료될 때 처리 결과를 확인할 수 있다.

- Future<T> submit(Callable<T> task) : task를 실행한다. Future를 통해 작업이 완료된 이후 처리 결과 및 리턴 값을 확인할 수 있다.

ThreadPoolTaskExecutor Class의 execute() 메서드와 submit() 메서드는 비동기로 작업을 수행한다. 따라서, execute() 메서드나 submit() 메서드를 실행하면

작업 실행 여부와 상관없이 메서드가 즉시 리턴되므로, 이 점에 유의해야 한다.

아래 Code는 <task:executor> 태그를 이용해서 생성한 ThreadPookTaskExecutor Class를 사용해서 작업을 실행하는 Code의 작성 예이다.

import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

public class Processor {

private ThreadPoolTaskExecutor executor;

public void setExecutor(ThreadPoolTaskExecutor executor) {

this.executor = executor;

}

public void process(final Work work) {

Future<?> future = executor.submit(new Runnable() {

@Override

public void run() {

work.run();

}

});

try {

future.get();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

return;

}

}

위 Code의 Processor Class를 초기화해 주는 SPRING 설정은 다음과 같을 것이다.

<bean class="madvirus.spring.chap12.Processor" p:executor="executor"/>

<task:executor id="executor" .../>

1.3 <task:executor>로 생성한 빈의 종료 처리

: <task:executor> 태그를 이용하면 TaskExecutionFactoryBean Class를 이용해서 ThreadPoolTaskExecutor 객체를 생성하게 된다. 그런데, SPRING은

FactoryBean에 의해서 생성된 빈 객체에 대해서는 라이프사이클 처리를 해 주지 않는다. 예를 들어, ThreadPoolExecutor는 InitializingBean Interface와

DisposableBean Interface를 모두 구현하고 있는데, FactoryBean을 통해서 생성된 ThreadPoolTaskExecutor 객체는 SPRING Container가 종료되더라도

DisposableBean Interface의 destroy() 메서드가 호출되지 않는다.

여기서 문제는 ThreadPoolTaskExecutor가 내부적으로 쓰레드 풀을 사용하고 있고, 풀에 생성되어 있는 쓰레드로 인해 컨텍스트를 종료하더라도

JVM 프로세스가 종료되지 않는다는 점이다. 예를 들어, 아래와 같은 Code를 보자.

--SPRING 설정 파일

<bean class="madvirus.spring.chap12.Processor" p:executor-ref="executor" />

<task:executor id="executor" ... />

--Java Code

AbstractApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

Processor processor = context.getBean(Processor.class);

processor.process(new Work());

context.close();

//Container를 종료해도 ThreadPoolTaskExecutor가 종료되지 않아 JVM 프로세스가 종료되지 않는다.

앞서 말했듯이 context.close() 메서드를 호출해도 <task:executor> 태그를 사용해서 생성된 ThreadPoolTaskExecutor 빈의 destroy() 메서드가 실행되지

않기 때문에 그리고 ThreadPoolTaskExecutor가 생성한 쓰레드가 살아 있기 때문에, 모든 Code의 실행이 끝나도 JVM 프로세스는 종료되지 않는다.

이런 문제를 없애려면 다음과 같이 ThreadPoolTaskExecutor 빈 객체의 shutdown() 메서드를 호출하면 된다.

AbstractApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");

context.registerShutdownHook();

Processor processor = context.getBean(Processor.class);

processor.process(new Work());

ThreadPoolTaskExecutor executor = context.getBean("executor", ThreadPoolTaskExecutor.class);

executor.shutdown();

ThreadPoolTaskExecutor의 shutdown() 메서드를 호출하면 내부적으로 생성한 쓰레드 풀을 종료하며, 이 때 관련된 쓰레드도 함께 종료된다. 따라서,

모든 Code의 실행이 끝난 뒤에 작업 실행을 위한 쓰레드 때문에 JVM 프로세스가 종료되지 않는 문제점을 해소 할 수 있다.

<task:executor> 태그를 사용하지 않고 다음과 같이 직접 ThreadPoolTaskExecutor Class를 사용해서 빈을 설정하면 그프링 Container가 종료될 때

ThreadPoolTaskExecutor Class의 destroy() 메서드를 통해서 shutdown() 메서드가 실행되므로 별도로 shutdown() 메서드를 호출하지 않아도 된다.

<bean id="executor"

class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"

p:corePoolSize="1"

p:maxPoolSize="2"

p:keepAliveSeconds="10"

p:queueCapacity="10">

<property name="rejectedExecutionHandler">

<bean class="java.util.concurrent.ThreadPoolExecutor.AbortPolicy" />

</property>

</bean>

2. <task:scheduler> 를 이용한 스케줄러 사용

: SPRING 3버전에 작업 Scheduling을 위한 TaskScheduler Interface가 새롭게 추가되었다. TaskScheduler Interface는 지정된 시간 또는 반복적으로 작업을

실행하기 위한 Interface를 제공하고 있으며, <task:scheduler> 태그를 이용해서 스케쥴러 빈을 생성할 수 있다. 본 절에서는 <task:scheduler> 태그를

이용해서 스케쥴러 빈을 생성하고, 생성된 빈을 이용해서 작업을 Scheduling 하는 방법을 살펴보도록 하겠다.

2.1 <task:scheduler>를 이용한 스케쥴러 생성

: <task:scheduler> 태그를 이용해서 스케쥴러 빈을 생성하는 방법은 매우 간단하다. 다음과 같이 생성할 빈의 식별값과 쓰레드 풀의 개수만 입력해주면 된다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:task="http://www.springframework.org/schema/task" xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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-3.0.xsd">

<task:scheduler id="scheduler" pool-size="20" />

...

</bean>

pool-size를 지정하지 않을 경우 쓰레드 풀의 기본값은 1이다. 풀에 생성된 쓰레드를 이용해서 작업을 Scheduling 하기 때문에, 스캐쥴링 될 작업 개수에 따라

알맞게 풀의 크기를 지정해 주면 된다.

2.2 TaskScheduler Interface를 이용한 작업 Scheduling

: <task:scheduler> 태그를 이용해서 TaskScheduler 빈을 생성했다면, TaskScheduler Interface가 제공하는 메서드를 이용해서 작업 실행을 Scheduling 할 수 있다.

TaskScheduler Interface가 제공하는 메서드는 다음과 같다. 참고로, 모든 메서드의 리턴 타입은 ScheduledFuture이며, period 파라미터와 delay 파라미터의

단위는 1/1000초이다.

- schedule(Runnable task, Trigger trigger) : Trigger가 지정한 시간에 작업을 실행한다.

- schedule(Runnable task, Date startTime) : startTime에 작업을 한 번 실행한다.

- scheduleAtFixedRate(Runnable task, Date startTime, long period) : startTime부터 period 시간마다 작업을 실행한다.

- scheduleAtFixedRate(Runnable task, long period) : 가능한 빨리 작업을 실행하고, 이 후 period 시간마다 작업을 실행한다.

- scheduleWithFixedDelay(Runnable task, Date startTime, long delay) : startTime부터 작업을 delay 시간 간격으로 작업을 실행한다.

- scheduleWithFixedDelay(Runnable task, long delay) : 가능한 빨리 작업을 실행하고, 이후 delay 시간 간격으로 작업을 실행한다.

아래 Code는 <task:scheduler> 태그로 생성한 TaskScheduler의 사용 예를 보여주고 있다.

--SPRING 설정 파일

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:task="http://www.springframework.org/schema/task" xmlns:p="http://www.springframework.org/schema/p"

...>

<task:scheduler id="scheduler" pool-size="5" />

...

</bean>

--Java Code

ThreadPoolTaskScheduler scheduler = context.getBean("scheduler", ThreadPoolTaskScheduler.class);

Calendar calendar = Calendar.getInstance();

calendar.add(Calendar.SECOND, 5);

scheduler.schedule(cacheInitializeRunner, calendar.getTime());

scheduler.scheduleAtFixedRate(statusMonitorRunner, 1000);

(1) CronTrigger를 이용한 Scheduling 설정

: schedule(Runnable task, Trigger trigger) 메서드의 두 번째 파라미터 타입은 org.springframework.scheduling.Trriger Interface인데,

Trigger Interface는 작업의 다음 실행 시간을 결정해주는 역할을 제공한다.

SPRING은 Trigger Interface 구현 Class로서 다음의 두가지를 기본으로 제공하고 있다.

- org.springframework.scheduling.support.CronTrigger

- org.springframework.scheduling.support.PeriodicTrigger

이름에서 유추할 수 있듯이 Trigger는 각각 cron 방식의 Scheduling과 주기적인 실행을 위한 Scheduling을 이용할 때 사용된다. 본 절에서는 이 두 가지

중에서 CronTrigger의 사용방법을 살펴보도록 하겠다.

CronTrigger는 cron 표현식을 이용해서 작업 실행 시간을 제공한다. 아래 Code는 CronTrigger를 이용해서 작업 실행을 Scheduling하는 Code의 예를

보여주고 있다.

CronTrigger trigger = new CronTrigger("0 30 0 \* \* \*");

scheduler.scheduler(logCollector, trigger);

위 Code는 매일 0시 30분에 logCollector를 실행하도록 설정한다.

cron 표현식은 다음과 같이 구성된다.

- 초 분 시 일 월 요일

각 시간 단위 값은 공백으로 구분되며, 다음의 표현식을 통해서 시간 간격, 주기 등을 표현할 수 있다.

: \* - 전체 값을 의미.

: 특정 값 - 해당 시간을 정확히 지정. 예) "0", "10", "20"

: 값1 - 값2 - 값1 부터 값2 사이를 표현. 예) "0-10"

: 값1, 값2, 값3 - 콤마로 구분하여 특정 값 목록을 지정. 예) "0,15,30"

: 범위/숫자 - 범위에 속한 값 중 숫자만큼 간격으로 값 목록을 지정. 예를 들어, "0-23/2"는 0부터 23까지 2 간격으로 값을 설정한다.

즉, 0,2,4,6,~,20,22의 값을 표현한다. \*을 사용해서 \*/2와 같이 표현할 수도 있다.

허용되는 값의 범위는 다음과 같다.

: 초 - 0-59

: 분 - 0-59

: 시 - 0-23

: 일 - 1-31

: 월 - 1-12

: 요일 - 0-7(0 또는 7은 일요일)

다음은 cron 표현식의 몇 가지 예를 보여주고 있다.

: 0 0 \* \* \* \* : 매일 매시 정각

: \*/10 \* \* \* \* : 0, 10, 20, 30, 40, 50초

: 0 0 8-10 \* \* \* : 매일 8시, 9시, 10시 정각

: 0 0/30 8-10 \* \* \* : 매일 8시, 8시30분, 9시, 9시30분, 10시

: 0 0 9-18 \* \* 1-5 : 매주 월요일부터 금요일의 9시부터 오후 6시까지 매

3. <task:scheduled-tasks>를 이용한 작업 스케줄링

: <task:scheduled-tasks> 태그를 이용하면 손쉽게 스케쥴러에 의해서 지정한 스케쥴에 작업을 설정할 수 있다. 다음 Code는 <task:scheduled-tasks> 태그의

설정 방법은 다음과 같다.

<task:scheduler id="scheduler" pool-size="5" />

<task:scheduled-tasks scheduler="scheduler">

<task:scheduled ref="logCollector" method="collect"

cron="\*/10 \* \* \* \* \*" />

</task:scheduled-tasks>

<bean id="logCollector" class="madvirus.spring.chap12.LogCollector" />

<task:scheduled-tasks> 태그의 scheduler 속성은 작업을 실행할 스케쥴러 빈을 설정한다.

<task:scheduled-tasks> 태그는 한 개 이상의 <task:scheduled> 태그를 가질 수 있다. <task:scheduled> 태그는 스케쥴러를 통해서 실행될 작업을 설정한다.

<task:scheduled> 태그의 ref 속성과 method 속성은 각각 실행할 빈 객체와 메서드를 지정한다.

<task:scheduled> 태그는 작업을 언제 실행할지의 여부를 지정하기 위해 다음의 세 속성 중 한 가지를 사용한다.

: cron - cron 표현식을 이용해서 실행 시간을 표현한다.

: fixed-delay - 지정된 시간 간격으로 작업을 실행한다.

: fixed-rate : 지정한 시간 주기로 작업을 실행한다.

위 속성을 여러 개 지정할 경우 적용 우선 순위는 cron, fixed-delay, fixed-rate 순이다.

4. Annotation을 이용한 작업 실행 설정

: SPRING3은 <task:scheduled-tasks> 태그를 사용하지 않고 Annotation을 사용해서 특정 빈 객체의 메서드를 주기적으로 실행하거나 비동기로 실행할 수 있도록

지원하고 있다. 본 절에서는 마지막으로 SPRING이 제공하는 @Scheduled Annotation과 @Async Annotation을 이용해서 작업을 실행하는 방법을 살펴보도록 하겠다.

@Scheduled Annotation과 @Async Annotation을 사용하려면 먼저 <task:annotation-driven> 태그를 이용해서 작업을 실행할 TaskExecutor와 TaskScheduler를

지정해주어야 한다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:task="http://www.springframework.org/schema/task" xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

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-3.0.xsd">

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

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

<task:executor id="executor" keep-alive="5" pool-size="1-2" queue-capacity="1" rejection-policy="ABORT" />

...

</beans>

4.1 @Scheduled Annotation을 이용한 작업 실행 설정

: @Scheduled Annotation은 빈 객체의 특정 메서드를 스케쥴러를 이용해서 실행할 때 사용된다. 다음 Code는 @Scheduled Annotation의 적용 예이다.

import org.springframework.scheduling.annotation.Scheduled;

public class LogProcessor {

@Scheduled(fixedRate = 5000)

public void process() {

System.out.println("로그 처리");

}

}

위 Code에서 @Scheduled는 fixed-rate의 값으로 5000을 주었는데, 이는 LogProcessor의 process() 메서드를 5초 주기로 실행한다는 것을 의미한다.

<task:annotation-driven> 태그와 함께 빈 객체로 등록해주면, 지정한 시간 주기로 process() 메서드가 실행된다.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:task="http://www.springframework.org/schema/task"

...>

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

<bean class="madvirus.spring.chap12.LogProcessor/>

...

</beans>

@Scheduled Annotation은 리턴 타입이 void이고 파라미터를 갖지 않는 메서드에 적용되며, Scheduling 설정을 위해 cron, fixedRate, fixedDelay 속성을 지정할 수 있다.

4.2 @Async Annotation을 이용한 비동기 실행

: @Async Annotation은 지정한 메서드를 비동기 실행으로 변환해 준다. 다음은 @Async Annotation의 적용 예를 보여주고 있다.

import org.springframework.scheduling.annotation.Async;

public class MessageSender {

@Async

public Future<String> send(String message) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Message 발송: " + message);

return new AsyncResult<String>(message);

}

}

@Scheduled Annotation과 마찬가지로 @Async Annotation이 적용된 메서드를 호출하면 바로 리턴 되며, 실제 메서드는 비동기로 실행된다.

MessageSender sender = context.getBean("messageSender", MessageSender.class);

// send 메서드 실행시 바로 리턴 됨.

// 실제 MessageSender.send() 메서드는 비동기로 실행

@Async Annotation이 적용된 메서드는 파라미터를 가질 수 있으며, 리턴 타입으로 void나 Future 타입을 가질 수 있다. 따라서, 비동기로 실행된 메서드의

결과를 필요로 할 경우 Future를 리턴 타입으로 사용하면 된다.

import java.util.concurrent.Future;

import org.springframework.scheduling.annotation.Async;

import org.springframework.scheduling.annotation.AsyncResult;

public class MessageSender {

@Async

public Future<String> send(String message) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Message 발송: " + message);

return new AsyncResult<String>(message);

}

}

AsyncResult Class는 SPRING이 제공하는 Future Interface의 구현 Class로서, Future 타입을 리턴해야 하는 경우 AsyncResult Class를 이용면 된다.

아래 Code는 Future 타입을 리턴하는 @Async 메서드의 사용 예를 보여주고 있다.

Future<String> future = messageSender.send("로그");

...

String result = future.get();