\*\*\*스프링 강의 노트 1일차\*\*\*

Part 01 SPRING의 Fundmental

++++++++++++++++++++++++

+Chapter 01 SPRING Framework 소개 +

++++++++++++++++++++++++

------------------------------------------------------------------------------------------------------------------------------------------------------------------

: Java가 Enterprise 시장에서 각광을 받기 시작한지 어느덧 15년이 넘어가고 있다.

최초에 Servlet이 출현하면서 WEB Based Application을 구축하는데 Java가 사용되기 시작했고,

Transaction, Security등을 제공하는 EJB가 제공되면서 Java는 Enterprise Application을 구축하는데

필요한 기본 기술로 자리잡아 나갔다.

하지만, EJB 2버전까지는 개발과정이 편하지는 않았다. SessionBean이나 EntityBean의 Code를 변경해서

Test를 해야 할 경우, Compile하고 알맞은 Format으로 묶은 뒤 EJB Container에 배포를 해야만 변경한

Code에 대한 Test가 가능했기 때문에, 개발속도를 향상시키는데 한계를 갖고 있었다.

Enterprise Application을 구현하는데 있어서 EJB Container가 제공하는 기능 중 실제로 많이

사용되는 기능은 Resource 관리와 Transaction의 처리 정도였고, EntityBean이 수행하는 Database Mapping이나

MDB(Message Driven Bean)가 제공하는 Message 처리등은 사실 EJB가 아닌 다른 기술들을 통해서 구현할 수

있는 것들이다. 또한, EJB는 반드시 EJB SPEC에 정의된 Interface에 따라 Code를 작성하도록 제약하고 있기

때문에 개발자가 기존에 작성한 POJO(Plain Old Java Object)를 변경해야 한다는 단점도 갖고 있었다.

Container 없이 Test가 어렵다는 점과 개발속도가 저하되는 부분은 많은 개발자들이 개발을 진행하는 데 있어

불편을 겪는 부분이었다. 이런 시점에서 Rod Johnson은 "Expert One-on-One J2EE Development without EJB"라는

책을 통해서 책 제목처럼 EJB를 사용하지 않고 Enterprise Application을 개발하는 방법을 소개하였고,

이것이 SPRING Framework의 모태가 되었다. 이후로 SPRING Framework는 급격하게 널리 사용되기 시작하였으며,

현재는 단순한 WEB Application 구축에서부터 금융 시스템과 같은 복잡한 Enterprise Application까지

사용 범위가 확대되었다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. SPRING Framework란?

: SPRING은 간단히 말하면 Enterprise Application에서 필요로 하는 기능을 제공하는 Framework다.

SPRING은 JEE(Java Enterprise Edition)가 제공하는 다수의 기능을 지원하고 있기 때문에,

JEE를 대체하는 Framework로 자리잡고 있다.

실제로 SPRING은 JEE가 제공하는 다양한 기능을 제공하는 것뿐만이 아니라 DI(Dependency Injection)이나

AOP(Aspect Oriented Programming)와같은 기능도 지원하고 있다.

다음은 SPRING이 제공하는 주요기능과 특징을 정리한 것이다.

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

- SPRING은 경량 Container이다.

SPRING은 Java객체를 담고 있는 Container다. SPRING Container는 이들 Java 객체의 생성, 소멸과 같은 라이프 사이클을 관리하며,

SPRING Container로부터 필요한 객체를 가져와 사용할 수 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 DI(Dependency Injection)를 지원한다.

SPRING은 설정 파일이나 Annotation을 통해서 객체간의 의존관계를 설정할 수 있도록 하고 있다. 따라서, 객체는 의존하고 있는

객체를 직접 생성하거나 검색할 필요가 없다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 AOP(Aspect Oriented Programming)를 지원한다.

SPRING은 자체적으로 AOP를 지원하고 있기 때문에 Transaction이나 Logging, Security과 같이 여러 Module에서 공통으로 필요로

하지만 실제 Module의 핵심은 아닌 기능들을 분리해서 각 Module에 적용할 수 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 POJO(Plain Old Java)를 지원한다.

SPRING Container에 저장되는 Java 객체는 특정한 Interface를 구현하거나 Class를 상속받지 않아도 된다. 따라서, 기존에 작성한

Code를 수정할 필요없이 SPRING에서 사용할 수 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 Transaction 처리를 위한 일관된 방법을 제공한다.

JDBC를 사용하든, JTA를 사용하든, 또는 Container가 제공하는 Transaction을 사용하든, 설정파일을 통해 Transaction 관련 정보를

입력하기 때문에, Transaction 구현에 상관없이 동일한 Code를 여러 환경에서 사용할 수 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 영속성과 관련된 다양한 API를 지원한다.

SPRING은 JDBC를 비롯하여 iBatis, Hibernate, JPA 등 Database 처리를 위해 널리 사용되는 Library와의 연동을 지원하고 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- SPRING은 다양한 API에 대한 연동을 지원한다.

SPRING은 JMS, Mail, Scheduling 등 Enterprise Application을 개발하는데 필요한 다양한 API를 설정 파일과 Annotation을 통해서

손쉽게 사용할 수 있도록 지원하고 있다.

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

위 특징과 더불어 SPRING 자체적으로 MVC Framework를 제공하고 있다. 따라서, SPRING만 사용해도 MVC 기반의 WEB Application을

어렵지 않게 개발할 수 있다. 또한, Struts2, JSF와 같은 Framework와의 연동을 지원하고 있기 때문에 이들 Framework에서

SPRING이 제공하는 다양한 기능을 사용할 수 있다.

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

2. SPRING Framework 설치와 Module 구성

: 우리가 다룰 SPRING Framework 버전은 3.0버전이다. SPRING Framework는 아래 Site에서 Download 받을 수 있다.

http://www.springsource.org/download

이 문서를 작성하는 시점에서 현재 안정화된 최신버전은 3.0.2 버전이며, 다음과 같이 의존 및 관련 문서를 모두 포함하고 있는

파일을 Download 받으면 된다.

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

- spring-framework-3.0.2.RELEASE-dependencies.zip (sha1) 155.7 MB

: SPRING Framework에서 사용하는 의존 jar 파일 목록이 포함되어 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- spring-framework-3.0.2.RELEASE-with-docs.zip (sha1) 45.0 MB

: SPRING Framework의 Module(jar 파일), Source Code, Reference 문서, API 문서가 포함되어 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

- spring-framework-3.0.2.RELEASE.zip (sha1) 21.4 MB

: SPRING Framework Module이 포함되어 있다. 관련 문서는 포함되어 있지 않다.

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

참고로 SPRING 3버전부터는 Java5 또는 그 이상의 버전을 필요로 하므로, Java1.4나 이하 버전에서는 SPRING 3버전을 사용할 수 없다.

: 2012.05.24

- spring-framework-3.1.0.RELEASE-with-docs.zip (sha1) 51.6 MB

- spring-framework-3.1.0.RELEASE.zip (sha1) 27.3 MB

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.1 Folder 구성

: 문서를 포함한 버전인 spring-framework-3.0.2.RELEASE-with-docs.zip의 압축을 풀면 다음과 같은 Folder가 생성된다.

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

- dist : SPRING Framework의 각 Module별 jar파일을 포함하고 있다.

----------------------------------------------------------------------------------------------

- docs : API 문서 및 Reference 문서.

----------------------------------------------------------------------------------------------

- projects : 각 Module별 Source Code 및 빌드 관련 파일을 포함하고 있다.

----------------------------------------------------------------------------------------------

- src : 각 Module별 Source jar 파일을 포함 하고 있다.

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

SPRING Framework를 사용하려면 dist Folder에 포함되어 있는 SPRING Module과 SPRING Module을 사용하는 데 필요한

외부 Library가 필요하다. 예를 들어, SPRING과 Hibernate를 연동하려면 SPRING의 ORM Module과 Hibernate 3.3 이상 버전을

사용하면 된다. SPRING Framework 의존하는 외부 Module은 spring-framework-3.0.2.RELEASE-dependencies.zip 파일에

포함되어 있다.

\*\*\*http://springide.org/updatesite\*\*\*

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 Module 구성

: SPRING 배포판의 압축을 푼 뒤 dist/ Folder를 살펴보면 20개의 jar 파일을 찾을 수 있는데, 각 jar 파일은 한 개의 Module을

의미하며, 주요 Module 목록은 아래와 같다. 각 Module의 이름은 dist/ Folder에 위치한 파일의 이름과 일치한다.

예를 들어, context Module은 org.springframework.context-3.0.1.RELEASE-A.jar 파일에 포함되어 있다.

[[SPRING의 주요 Module 목록]]

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

\*\*core : DI 기능을 비롯한 Framework의 기반을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*\*beans : BeanFactory Interface를 통해 구현된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*\*expression : 객체에 접근하고 객체를 조작하기 위한 표현 언어를 제공한다. JSP2.1규약에 명시된 통합 EL을 확장하였다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*\*context : spring-core와 spring-beans Module을 확장해서 i18n, Event 처리, Resource 로딩, Servlet Container를 위한 Context 생성등의

기능을 추가로 제공한다. ApplicationContext Interface를 통해 구현된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

context.support : Ehcache, Mail, Scheduling, UI의 Velocity 지원 기능을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*aop : AOP Alliance에 호횐되는 AOP 구현을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

aspects : AspectJ와의 통합을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

web : FileUpload, Locale 처리 등 WEB을 위한 통합 기능을 제공한다. 또한, 원격 지원 기능 중 WEB 관련 기능을 지원한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*web.servlet : SPRING MVC를 제공한다. JSP, Velocity에 대한 View 연동을 지원한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\* web.struts : SPRING과 Struts 연동 기능을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

web.portlet : 포틀릿 환경에서 사용되는 MVC 구현을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*transaction : AOP를 이용한 선언적 Transaction 관리 및 Code를 이용한 Transaction 관리 기능을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*jdbc : JDBC Programming을 위한 abstract layer를 제공한다. JDBC Template을 제공함으로써 간결한 Code로 JDBC Programming을

할 수 있도록 돕는다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

\*orm : Hibernate, JPA, iBatis, JDO 등 ORM API를 위한 통합 layer를 제공한다.

SPRING이 제공하는 Transaction 관리와의 연동을 지원한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

oxm : 객체와 XML 사이의 Mapping을 처리하기 위한 abstrat layer를 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

jms : JMS의 Message를 생성하고 수신하는 기능을 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

test : JUnit이나 TestNG를 이용한 SPRING 컴포넌트의 Test를 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

instrument : Instrumentation 지원 Class를 지원한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

instrument.tomcat : 톰캣 서버를 위한 Instrumentation 지원 Class를 제공한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

asm : ASM Library를 재패키징한 Module.

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

Module간의 의존관계는 아래와 같다.

web.portlet ===> web.servlet ===> web ===> context

orm ===> jdbc ===> transaction ===> context

oxm ===> context

jms ===> context

context.support ===> context

aspects ===> context

===> test

context ===> aop ===> beans ===> core ===> asm

===> expression ===> core ===> asm

위에서 SPRING이 제공하는 spring-web.servlet Module을 사용하려면 추가로 spring-context Module, spring-beans Module,

spring-core Module, spring-asm Module을 Class 패스에 추가해주어야 한다. 또한, 각 Module이 필요로 하는 외부 jar 파일을

Class 패스에 추가해주면 된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.3 외부 의존 Module.

: SPRING의 각 Module은 다양한 외부Library를 사용하고 있다. 예를 들어, SPRING은 Commons Logging이나 SLF4J API를

이용해서 로그를 기록한다. 또한, Hibernate, JPA, iBatis 등 DB연동 지원 기능은 다양한 외부 Library를 사용하고 있다.

SPRING을 사용하는데 필요한 모든 외부 의존 Module을 Download 받으려면 다양한 Site를 방문해서 일일이 파일을 Download

받아야 한다.

SPRING은 이런 불편함을 해소하기 위해 SPRING Framework를 사용하는 데 필요한 모든 외부 의존 Module을

spring-framework-3.0.1.RELEASE-dependencies.zip 파일로 제공하고 있다. 이 파일의 압축을 풀면 javax.servlet Folder,

org.apache.commons Folder등이 존재하고 있으며, 각 하위 Folder에는 com.springsource.org.apache.commons.httpclient/3.1.0Folder와 같이

의존 Library의 이름 및 버전에 따른 디렉토리가 존재하며, 해당 디렉토리에는 com.springsource.org.apache.commons.httpclient-3.1.0.jar

와 같은 실제 의존하는 jar파일이 위치한다. 각 jar파일의 형식은 com.springsource.Module명-버전.jar이다.

예를 들어, Hibernate 3.3.1 GA 버전의 경우

org.hibernate/com.springsource.org.hibernate/3.3.1 GA Folder에 com.springsource.org.hibernate-3.3.1 GA.jar 파일 이름으로 포함되어 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.4 Maven에서의 SPRING Module 의존 설정

: 프로젝트 빌드/관리를 위해 Maven Framework가 널리 사용되고 있는데, Maven의 pom.xml 파일에 SPRING3 버전에 대한 의존을 추가하려면

groupId로 org.springframework를 사용하고, artifactId로 spring-\* 형식을 사용하면 된다.

예를 들어, aopModule을 사용하려면 다음과 같이 의존을 추가하면 된다.

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>3.0.1.RELEASE</version>

</dependency>

...

</dependencies>

artifactId의 명명규칙은 SPRING Framework 배포판에 포함된 jar파일의 이름과 차이가 있다. 배포판의 Module명에 점(.)이 포함되어 있는 경우,

Maven의 artifactId는 전 대신에 '-'을 사용한다. 예를 들어, web.servlet Module의 경우 배포판에 포함된 jar파일의 이름은 org.springframework.web.servlet인데

반해 Maven 의존에서 사용하는 artifactId는 spring-webmvc이다. 아래 Code는 web.servlet Module에 의존하는 Maven 파일의 작성 예를 보여주고 있다.

<dependencies>

...

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>3.0.1.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>3.0.1.RELEASE</version>

</dependency>

...

</dependencies>

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

3. Dependency Injection과 SPRING Framework

: Dependency Injection, 줄여서 DI는 SPRING Container가 지원하는 핵심 개념 중의 하나이다. DI라는 용어에 상관없이

DI는 이미 다양한 Framework에 적용되어 있는 기능이며, SPRING에 새롭게 추가된 개념은 아니다.

DI를 간단히 설명하면, 객체 간의 의존 관계를 객체 자신이 아닌 외부의 Assembler가 수행해 준다는 개념이다.

본 문서에서는 이에대해 좀 더 구체적으로 살펴보고 SPRING에서 DI를 어떻게 지원하는 지 예제를 통해 알아보도록 하자.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.1 Dependency Injection

: Application은 규모에 따라 수십에서 수백, 수천개의 객체로 구성되어 있다. 일반적인 WEB Application의 경우

클라이언트의 요청을 받아 주는 컨트롤러 객체, 비지니스 로직을 수행하는 서비스 객체, 데이터 접근을 수행하는 DAO 객체

등으로 구성된다. 이런 객체들은 독립적으로 기능을 수행하기 보다는, 서로 의존하여 Application의 기능을 구현하게 된다.

예를 들어, 아래를 보면 WriteArticlesServiceImpl Class는 ArticleDAO Interface에 의존하고 있다.

<<Interface>> <<Interface>>

WriteArticesService ArticleDAO

^ ^

|| ||

|| ||

WriteArticlesServiceImpl MySQLArticleDAO OracleArticleDAO

WriteArticlesServiceImpl Class의 객체는 실제로 ArticleDAO Interface를 구현한 MySQLArticleDAO Class의 객체나 OracleArticleDAO Class의

객체에 의존하게 된다. 예를 들어, ArticleDAO Interface의 구현 Class로서 MySQLArticleDAO Class를 사용한다면,

메모리에 생성된 객체는 다음과 같은 의존 관계를 형성할 것이다.

WriteArticlesServiceImpl ===> MySQLArticleDAO

위에서 볼 수 있듯이 WriteArticlesServiceImpl 객체가 의존하는 객체는 ArticleDAO Interface를 구현한 Class의 객체가 된다.

따라서, WriteArticlesServiceImpl Class는 실제로 의존할 객체를 지정할 수 있는 방법을 필요로 한다.

실제로 의존하는 객체를 지정하는 가장 간단한 방법은 Code에 직접 명시하는 것이다.

예를 들어, 다음과 같이 WriteArticlesServiceImpl Class는 Code에 직접 MySQLArticleDAO 객체에 의존한다고 명시할 수 있다.

public class WriteArticlesServiceImpl{

//Code에 직접 의존 객체 명시

private ArticleDAO articleDAO = new MySQLArticleDAO();

...

}

Code에 직접 의존 Class를 명시하는 것이 손쉬운 방법이긴 하지만 단위 Test를 어렵게 만드는 단점이 있다.

articleDAO 멤버 필드에 mock 객체를 할당할 수 없기 때문에, WriteArticlesServiceImpl Class를 Test하기 위해서는

반드시 올바르게 동작하는 MySQLArticleDAO Class가 존재해야 하고, 심지어 DBMS를 필요로 할 수도 있다.

또 다른 단점으로는 의존하는 Class가 변경되는 경우 Code를 변경해야 하는 문제가 있다. 예를 들어, 의존하는 Class를

MySQLArticleDAO에서 OracleArticleDAO로 변경해야 하는 경우 Code를 변경한 뒤 에 Compile 해야만 한다.

의존 관계를 처리하는 또 다른 방법은 Factory 패턴이나 JNDI등을 사용해서 의존 Class를 검색하는 방법이다.

Factory를 사용하는 경우 Code는 다음과 같은 형태로 바뀌게 된다.

public class WriteArticlesServiceImpl{

private ArticleDAO articleDAO = ArticleDAOFactory.create();

...

}

ArticleDAOFactory Class는 ArticleDAO Interface를 구현한 Class 중에서 어떤 Class를 사용해야 할 지의 여부를

알아내기 위해 외부의 설정 파일을 사용할 수도 있고, 시스템 프로퍼티를 사용할 수도 있을 것이다.

Factory나 JNDI를 사용하면 앞서 Code에 직접 의존 Class를 명시할 때의 두 번째 문제점(즉, 의존 Class가 변경되면

Code를 변경해야 하는 문제)을 없앨 수는 있지만, WriteArticlesServiceImpl Class를 Test 하려면 올바르게 동작하는

Factory 또는 JNDI에 등록된 객체를 필요로 한다는 점에서, Code에서 직접 의존하는 객체를 생성해 줄 때 발생하는 문제점을

완전히 극복하지는 못한다.

세 번째 방식은 외부 Assembler를 사용하는 것이다. 이 방식에서는 의존 관계에 있는 객체가 아닌 외부의 Assembler가 각 객체

사이의 의존 관계를 설정해 준다.

WriteArticlesServiceImpl

-articleDAO : ArticleDAO

||

||

<<Interface>> Assembler : DI패턴에서는 Assembler가 객체 간의 의존 관계를 관리.

ArticleDAO

||

||

MySQLArticleDAO

위에서 WriteArticlesServiceImpl Class의 Code는 MySQLArticleDAO 객체를 생성하거나 검색하기 위한 Code가 포함되어 있지 않다.

대신, Assembler의 역할을 하는 Assembler가 MySQLArticleDAO객체를 생성한 뒤 WriteArticlesServiceImpl 객체에 전달해 주게 된다.

이렇게 Assembler가 의존 관계를 관리해 주는 방식을 Dependency Injection(이하 DI) 패턴이라고 한다.

DI 패턴을 적용할 경우 WriteArticlesServiceImpl Class는 의존하는 객체를 전달받기 위한 설정 메서드(setter method)나

생성자를 제공할 뿐 WriteArticlesServiceImpl에서 직접 의존하는 Class를 찾지 않는다. 예를 들어, DI 패턴을 적용할 경우

WriteArticlesServiceImpl Class는 다음과 같이 의존하는 객체를 전달받기 위한 생성자를 제공하게 된다.

public class WriteArticlesServiceImpl{

private ArticleDAO articleDAO;

//생성자에서 의존하는 객체를 전달 받음.

public WriteArticlesServiceImpl(ArticleDAO articleDAO){

this.articleDAO = articleDAO;

}

...

}

WriteArticlesServiceImpl Class는 의존 객체를 직접 생성하지도 않고, Factory나 JNDI를 이용하지도 않는다. WriteArticlesServiceImpl Class

단지 의존 객체를 전달받을 수 있는 메서드나 생성자만을 제공할 뿐이다.

WriteArticlesServiceImpl Class가 의존하고 있는 객체를 WriteArticlesServiceImpl 객체에 전달해주는 역할은 Assembler가 맡게 된다.

Assembler는 내부적으로 다음과 같은 형태의 Code를 사용하여 WriteArticlesServiceImpl 객체에 의존 객체를 전달하게 될 것이다.

public class Assembler{

public WriteArticlesServiceImpl getWriteArticleService(){

ArticleDAO articleDAO = new MySQLArticleDAO();

WriteArticleService service = new WriteArticlesServiceImpl(articleDAO);

return service;

}

}

public class UsingService{

public void useService(){

...

//Assembler로부터 사용할 객체를 구함.

WriteArticleService service = assembler.getWriteArticleService();

service.write(..);

...

}

}

Assembler가 WriteArticlesServiceImpl에 MySQLArticleDAO 객체를 넣어 주기 때문에 WriteArticlesServiceImpl Class에는

의존 객체를 찾기 위한 Code가 필요하지 않게 된다. 또한, WriteArticlesServiceImpl 객체를 사용하는 Class는 Assembler로부터

WriteArticlesServiceImpl 객체를 구해서 사용하게 된다.

의존(Dependency)하는 객체를 Assembler가 삽입(Inject)해 주기 때문에 이 방식을 DI(Depedency Injection) 패턴이라고 부른다.

비슷한 의미로 IoC(Inversion of Control)이라고도 표현하는데, 본 문서에서는 DI라는 용어만을 사용할 것이다.

DI 패턴을 적용하려면 불필요한 의존 관게를 없애거나 줄일 수 있게 된다. 예를 들어, 의존하는 객체를 Code에서 직접 생성하는 경우

아래와 같이 Interface와 구현 Class에 대해서 의존 관계를 갖게 되지만, DI 패턴을 적용하면 Interface에만 의존 하게 된다.

따라서, WriteArticlesServiceImpl을 수정할 필요 없이 ArticleDAO 구현 Class를 MySQLArticleDAO에서 OracleArticleDAO로 교체할수 있게 된다.

<<Interface>> <<Interface>>

WriteArticesService ArticleDAO

ㅅ ㅅ

|| ||

|| ||

WriteArticlesServiceImpl MySQLArticleDAO

-articleDAO : ArticleDAO

\*\*\*의존 객체를 직접 생성할 경우 불필요한 의존 관계가 증가한다.

Factory 패턴을 사용할 경우 구현 Class에 대한 의존 관계는 줄일 수 있지만, 대신 Factory Class에 대한 의존 관계가 추가된다.

하지만, DI 패턴을 사용하면 WriteArticlesServiceImpl Class는 오직 ArticleDAO Interface에만 의존하게 되며

그 외 다른 Class에는 의존하지 않게 된다.

DI 패턴을 사용할 때의 또 다른 장점은 단위 Test를 수행하는 게 수월해진다는 점이다. 예를 들면, mock 객체를 사용하여

MySQLArticleDAO Class를 사용하지 않고도 WriteArticlesServiceImpl Class를 Test 할 수 있게 된다.

단위 Test는 Code의 품질을 향상시키고 개발 속도를 증가시키는데 도움이 되기 때문에, DI 패턴이 단위 Test를 진행하는 데

도움을 준다는 것은 중요한 장점이다.

DI 패턴을 적용하려면 각 객체들을 조립해 주는 Assembler가 필요하다. 이 Assembler는 개발자가 직접 만들 수도 있지만, 이 경우 조립해야

할 객체가 증가하고 의존 관계가 복잡해 질수록 Assembler를 만드는 일 또한 힘든 일이 된다. 따라서, Assembler를 직접 만들기 보다는

DI 패턴을 지원해주는 Framework를 사용하는 것이 좋다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 SPRING에서의 DI

: DI 패턴을 지원해 주는 다양한 Framework(또는 Container)가 존재하는데, 그 중에서 WEB Application을 개발할 때

널리 사용되고 있는 Framework는 SPRING이다. SPRING은 설정파일과 Annotation을 이용하여 손쉽게 객체 간의

의존 관계를 설정하는 기능을 제공하고 있기 때문에, 어렵지 않게 SPRING을 객체 Assembler로 사용할 수 있다.

아래 그림은 앞에의 설명에 사용한 것을 다시 보여 주고 있는데, 이 그림을 보면 Assembler가 WriteArticlesServiceImpl객체와

MySQLArticleDAO 객체를 조립해 주는 것을 알 수 있다. SPRING을 사용할 경우 어떻게 이 객체들을 조립할 수 있는 지 Code를 톨

살펴 보도록 하자.

WriteArticlesServiceImpl

-articleDAO : ArticleDAO

||

||

<<Interface>> Assembler : DI패턴에서는 Assembler가 객체 간의 의존 관계를 관리.

ArticleDAO

||

||

MySQLArticleDAO

(1) DI 패턴을 적용한 Java Code

: DI 패턴을 적용한 경우 WriteArticlesServiceImpl Class는 생성자나 설정 메서드를 이용하여 의존 객체를 전달받을 수 있다.

다음의 Source는 생성자 방식을 이용하여 의존 객체를 전달받는 WriteArticlesServiceImpl의 Code를 보여주고 있다.

[WriteArticlesServiceImpl.java]

package madvirus.spring.chap01;

public class WriteArticleServiceImpl implements WriteArticleService {

private ArticleDao articleDao;

public WriteArticleServiceImpl(ArticleDao articleDao) {

this.articleDao = articleDao;

}

@Override

public void write(Article article) {

System.out.println("WriteArticleServiceImpl.write() 메서드 실행");

articleDao.insert(article);

}

}

설정 메서드 방식을 사용할 경우 다음과 같이 set 메서드를 이용하면 된다.

public class WriteArticleServiceImpl implements WriteArticleService{

...

public void setArticleDAO(ArticleDAO articleDAO){

this.articleDAO = articleDAO;

}

...

}

WriteArticleServiceImpl Class가 전달받게 될 의존 객체 중 하나인 MySQLArticleDAO Class는 SPRING이 제공하는

DI Test를 위해 다음 Code와 같이 간단히 작성 하였다.

[MySQLArticleDAO.java]

package madvirus.spring.chap01;

public class MySQLArticleDao implements ArticleDao {

@Override

public void insert(Article article) {

System.out.println("MySQLArticleDao.insert() 실행");

}

}

(2) SPRING 설정 파일을 이용한 의존 관계 설정

: 앞서 DI 패턴을 적용하여 WriteArticleServiceImpl Class를 작성했으니, 이제 SPRING을 이용하여 WriteArticleServiceImpl객체와

MySQLArticleDAO 객체 사이의 의존 관계를 어떻게 처리할 수 있는 지 살펴보도록 하자.

SPRING은 Code나 설정 파일을 이용하여 객체간의 의존 관계를 설정할 수 있는데, 이 문서에서는 설정 파일을 이용하여

객체간의 의존 관계를 지정하는 방식에 대해서 살펴 볼 것이다. 다음 Code는 예제에서 사용할 SPRING 설정 파일로서

Class 패스에 위치 시킨다.

[applicationContext.xml]

<?xml version="1.0" encoding="UTF-8"?>

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

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">

<bean name="writeArticleService" class="madvirus.spring.chap01.WriteArticleServiceImpl">

<constructor-arg>

<ref bean="articleDao" />

</constructor-arg>

</bean>

<bean name="articleDao" class="madvirus.spring.chap01.MySQLArticleDao">

</bean>

</beans>

SPRING은 각 객체를 빈(bean)으로 관리한다. 위의 Code에서 <beans> 태그는 SPRING 설정 파일의 루트 태그이며,

<bean> 태그는 SPRING이 관리할 하나의 객체를 설정하는 데 사용된다. 일단 지금은 <bean> 태그의 name 속성이

빈의 이름을 의미하고, class 속성이 생성될 객체의 Class 타입이라는 것만 기억하자.

<constructor-arg> 태그는 'writeArticleService' 빈 객체를 생성할 때 생성자에 전달할 파라미터를 명시하기 위해 사용된다.

위 Code의 경우 생성자에 정의한 'articleDao' 빈 객체를 전달한다고 명시하였다. 이는 Code로는 다음과 같은 의미를 갖는다.

MySQLArticleDAO articleDAO = new MySQLArticleDAO();

WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl(articleDAO);

즉, WriteArticleServiceImpl 객체와 MySQLArticleDAO 객체 사이의 의존 관계를 설정하고 있다.

(3) Class패스 설정 및 ApplicationContext를 이용한 빈 객체 사용.

: 설정 파일을 작성 했으니 이제 남은 작업은 SPRING을 이용하여 설정 파일을 로딩한 뒤, 설정 파일에 명시한 빈 객체를 사용하는 것이다.

SPRING Container를 사용하려면 먼저 아래의 jar 파일을 Class 패스에 추가해 주어야 한다.

- org.springframework.asm.3.0.1.RELEASE-A.jar

- org.springframework.core.3.0.1.RELEASE-A.jar

- org.springframework.beans-3.0.1.RELEASE-A.jar

- commons-logging-1.1.1.jar

아래 Code는 앞서 작성한 설정 파일로부터 BeanFactory를 생성하고, BeanFactory로부터 필요한 빈 객체를 가져와 사용하는 예제 Code이다.

[Main.java]

1 package madvirus.spring.chap01;

2

3 import org.springframework.beans.factory.BeanFactory;

4 import org.springframework.beans.factory.xml.XmlBeanFactory;

5 import org.springframework.core.io.ClassPathResource;

6 import org.springframework.core.io.Resource;

7

8 public class Main {

9

10 public static void main(String[] args) {

11 Resource resource = new ClassPathResource("applicationContext.xml");

12 BeanFactory beanFactory = new XmlBeanFactory(resource);

13 WriteArticleService articleService = (WriteArticleService) beanFactory.getBean("writeArticleService");

14 articleService.write(new Article());

15 }

16}

라인 12에서 XmlBeanFactory Class는 resource가 나타내는 XML 파일로부터 SPRING 설정 내용을 로딩하여 빈 객체를 생성하는

BeanFactory 구현 Class이다. 일단, 지금은 BeanFactory가 빈 객체를 관리하는 Container라는 것 정도만을 기억하도록 하자.

BeanFactory 객체를 생성하면, BeanFactory로 부터 빈 객체를 가져와 사용할 수 있게 된다. 앞선 Code의 설정 파일에서 <bean> 태그의

name 속성을 이용하여 빈 객체에 이름을 부여했는데, 이때 부여한 이름을 사용하여 빈 객체를 구할 수 있다.

SPRING Container로부터 빈 객체를 가져오려면 BeanFactory.getBean(String name) 메서드를 사용하면 된다.

라인 13에서는 이름이 'writeArticleService'인 빈 객체를 가져와 'articleService' 변수에 할당하고 있다.

앞서 SPRING 설정 파일에서 MySQLArticleDAO Class 타입인 'articleDAO' 빈 객체를 WriteArticleServiceImpl Class 타입인

'writeArticleService' 빈 객체와 연결해 주었다. WriteArticleServiceImpl.write() 메서드는 콘솔에 Message를 출력한 뒤, 생성자로

전달받은 ArticleDAO 객체의 insert() 메서드를 호출하도록 하였고, MySQLArticleDAO Class의 insert() 메서드는 콘솔에 Message를

출력하도록 하였다. 따라서, Main Class를 실행하면 다음과 같은 결과가 출력된다.

WriteArticleServiceImpl.write() 메서드 실행

MySQLArticleDAO.insert() 실행

본 예제에서는 생성자를 이용하여 두 객체의 의존 관계를 정하는 것을 살펴봤는데, SPRING은 다양한 방법으로 의존 관계를

설정할 수 있도록 지원하고 있다. 이에 대한 보다 자세한 내용은 '2장. SPRING DI와 객체 관리'에서 살펴볼 것이다.

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

4. AOP와 SPRING

: Application을 개발하다 보면 여러 부분에 걸쳐서 공통으로 사용되는 기능이 필요한 경우가 있다. 예를 들어, Logging이나,

Transaction 처리, Security과 같은 기능은 대부분의 Application에서 필요로 한다.

Transaction이나 Security과 같은 기능은 어떤 특정 Module에서 필요로 하기보다는, Application 전반에 걸쳐서 필요한 기능들이며,

실제 핵심 비지니스 로직과는 구분되는 기능이기도 하다. 예를 들어, 게시글 쓰기의 경우 현재 사용자가 권한이 있는 지 검사하는

Security 처리 로직과 데이터 무결성 처리를 위한 Transaction 처리 로직은 게시글 쓰기를 수행하기 위해 필요한 로직과는 구분되는 기능이다.

Security처리와 Transaction 처리는 글쓰기 뿐만 아니라 글 삭제, 글 목록 등 다양한 기능을 구현하는 데 필요한 공통 관심 사항

(cross-cutting concern)이다.

이런 공통 관심 사항들은 객체 지향 기법(상속이나 패턴)을 사용해서 여러 Module에 효과적으로 적용하는 데는 한계가 있으며,

이런 한계를 극복하기 위해 AOP라는 기법이 소개 되었다. 본 절에서는 간단하게 SPRING이 제공하는 AOP를 이용하여

공통 관심 사항을 어떻게 구현하고 적용할 수 있는지 살펴보도록 하겠다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

4.1 AOP 소개

: Security이나 Transaction과 같은 공통 기능을 별도의 Module로 구현한 뒤, 각 기능을 필요로 하는 곳에서 사용하게 될 경우, 각 Module과

공통 Module 사이의 의존 관계는 다음 그림과 같이 형성된다.

WriteArticleServiceImpl ==================================> LoggingModule

||

ReadArticleServiceImpl ==================================> TransactionModule

||

GetArticleListServiceImpl =================================> SecurityModule

\*\*\*공통으로 사용되는 Module에 대한 복잡한 의존 관계

위에서 볼 수 있듯이 Application 전반에 걸쳐서 공통으로 사용되는 기능이 많아 질수록, 그리고 공통 Module을 사용하는

Class가 많아 질수록 의존 관계는 점점 복잡해 질 것이다. 또한, 공통 Module을 사용하는 Code가 여러 곳에서 중복되는 문제도 발생한다.

AOP(Aspect Oriented Programming)는 공통의 관심 사항을 적용해서 발생하는 의존 관계의 복잡성과 Code 중복을 해소해 주는 프로그래밍

기법이다. AOP에서는 각 Class에서 공통 관심 사항을 구현한 Module에 대한 의존 관계를 갖기 보다는, Aspect를 이용하여

핵심 로직을 구현한 각 Class에 공통 기능을 적용하게 된다.

일단 지금은 Aspect가 공통 Module을 로직을 구현할 Class에 적용된다는 것만 이해하고 '5장. SPRING AOP'에서

Aspect에 대해 자세히 살펴보도록 하겠다.

다음 그림은 Aspect를 이용할 경우 로직을 구현한 Class가 어떻게 변경되는 지를 보여 주고 있다. 핵심 로직을 구현한 Class는

더 이상 여러 공통 Module에 의존하지 않는다. 즉, 핵심 로직 구현 Class에서 공통 Module을 사용하는 Code를 포함 하지는 않는다.

LoggingModule <==== LoggingAspect -------------

|

SecurityModule <==== SecurityAspect -------------------------------> CoreConcreteImplClass

|

TransactionModule <==TransactionAspect ----------

\*\*\*AOP에서는 공통 관심 사항을 구현한 Module에 의존 관계를 갖지 않는다.

AOP에서는 핵심 로직을 구현한 Class를 실행하기 전/후에 Aspect를 적용하고, 그 결과로 핵심 로직을 수행하면 그에 앞서

공통 Module을 실행하거나 또는 로직 수행 이후에 공통 Module을 수행하는 방식으로 공통 기능을 실행하게 된다.

AOP에서 중요한 점은 Aspect가 핵심 로직 구현 Class에 의존하지 않는다는 점이다. 일반적으로 AOP에서는 설정 파일이나

설정 Class 등을 이용하여 Aspect를 개발하게 되면, Aspect를 수정할 필요 없이 여러 Class에 적용할 수 있다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

4.2 SPRING에서의 AOP

: 본 절에서는 메서드의 실행 시간을 출력해 주는 간단한 Code를 작성해 볼 것이다. AOP를 적용하지 않는다면 아마도 다음과 같은 방식으로

특정 메서드를 호출하는 Code에서 메서드의 실행 시간을 구하게 될 것이다.

public void someMethod(){

StopWatch stopWatch = new StopWatch();

stopWatch.start();

executeLogic();

stopWatch.stop();

long executionTime = stopWatch.getTotalTimeMillis();

}

그런데, 만약 실행 시간을 구해야 할 메서드가 많아 진다면 어떻게 해야 할까? 또는 조건에 따라서 실행 시간을 구해야 할 메서드를 선택해야

한다면 어떻게 해야할까? 만약 단순히 위와 같은 방식으로 실행 시간을 구하게 된다면, 요구가 변경될 때 마다 많은 Code를 변경해야 할 것이다.

위 Code에서 실행 시간을 구하는 Code는 핵심 로직과 관련된 것이라기 보다는 Application 전반에 적용되는 공통 관심 사항에 해당된다.

SPRING에서는 몇 가지 방식으로 AOP를 적용할 수 있도록 하고 있는데, 본 절에서는 그 중에서도 비교적 쉽고 간단한 POJO 기반의 예제를

작성해 보도록 하겠다.

본 절의 예제 Code를 Compile하고 실행하기 위해서는 아래 jar 파일을 Class 패스에 추가해 주어야 한다.

- org.springframework.asm.3.0.1.RELEASE-A.jar

- org.springframework.core.3.0.1.RELEASE-A.jar

- org.springframework.beans-3.0.1.RELEASE-A.jar

- org.springframework.aop-3.0.1.RELEASE-A.jar

- org.springframework.context-3.0.1.RELEASE-A.jar

- org.springframework.expression-3.0.1.RELEASE-A.jar

- aopalliance-1.0.jar

- aspectjweaver-1.6.8.jar

- commons-logging-1.1.1.jar

먼저, 공통 관심 사항을 구현한 POJO Class를 작성해보자. 본 예제에서는 SPRING Framework가 제공하는 StopWatch Class를 이용하여

메서드 실행시간을 구할 것이며, 공통 기능을 구현한 Class는 다음과 같다.

[LoggingAspect.java]

package madvirus.spring.chap01;

import org.apache.commons.logging.Log;

import org.apache.commons.logging.LogFactory;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.util.StopWatch;

public class LoggingAspect {

private Log log = LogFactory.getLog(getClass());

public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

log.info("기록 시작");

StopWatch stopWatch = new StopWatch();

try {

stopWatch.start();

Object retValue = joinPoint.proceed();

return retValue;

} catch (Throwable e) {

throw e;

} finally {

stopWatch.stop();

log.info("기록 종료");

log.info(joinPoint.getSignature().getName() + "메서드 실행 시간 : "

+ stopWatch.getTotalTimeMillis());

}

}

}

위의 logging() 메서드는 Aspect가 적용되는 메서드의 실행 시간을 구한 뒤 Log를 통해 출력하도록 구현하였다.

logging() 메서드가 파라미터로 전달받는 ProceedingJoinPoint 객체는 Aspect가 적용되는 객체 및 메서드에 대한 정보를

담고 있으며, 이 객체를 통해서 핵심 로직을 (즉, Aspect가 적용되는 메서드를) 실행할 수 있게 된다.

LoggingAspect Class는 핵심 기능을 구현한 Class나 Interface에 전혀 의존하지 않고, 오직 ProceedingJoinPoint에만

의존하는 것을 알 수 있는데, 이렇게 Aspect를 적용할 객체에 의존하지 않도록 함으로써 Code 수정 없이 LoggingAspect를

여러 Class에 적용할 수 있게 된다.

공통 기능을 구현한 Aspect Class를 작성했다면, 다음으로 할 일은 Aspect를 어떤 Class의 어떤 메서드에 적용할지를

설정하는 것이다. SPRING의 XML 설정 파일을 다음과 같이 작성함으로써 앞서 작성한 LoggingAspect가 앞에서 작성한

WriteArticleServiceImpl Class에 적용되도록 설정할 수 있다.

[commonConcern.xml]

<?xml version="1.0" encoding="UTF-8"?>

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<bean id="logging" class="madvirus.spring.chap01.LoggingAspect" />

<aop:config>

<aop:pointcut id="servicePointcut" expression="execution(\* \*..\*Service.\*(..))" />

<aop:aspect id="loggingAspect" ref="logging">

<aop:around pointcut-ref="servicePointcut" method="logging" />

</aop:aspect>

</aop:config>

</beans>

위의 Code는 XML 스키마 확장( aop 네임스페이스)를 사용하여 AOP를 설정하였다. 일단, 지금은 위와 같이 설정하면

이름이 Service로 끝나는 Interface를 구현한 모든 Class의 모든 메서드에 LoggingAspect가 적용된다는 정도로만

이해하도록 하자. 보다 자세한 내용은 '5장. SPRING AOP'에서 설명할 것이다.

공통 관심 사항을 구현한 Class를 작성했고, 공통 관심 사항을 어떻게 적용할지에 대한 내용을 설정 파일로 작성했다면,

실제로 Aspect가 적용되는 지 Test하는 일만 남았다. Test를 위한 Code는 다음과 같다.

[MainForAop.java]

package madvirus.spring.chap01;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainForAop {

public static void main(String[] args) {

String[] configLocations = new String[] { "applicationContext.xml", "commonConcern.xml" };

ApplicationContext context = new ClassPathXmlApplicationContext( configLocations);

WriteArticleService articleService = (WriteArticleService) context.getBean("writeArticleService");

articleService.write(new Article());

}

}

앞서 SPRING DI 예제인 Main.java와 비교해 보면 BeanFactory 대신 ApplicationContext를 사용한 점과 SPRING 설정 파일로

commonConcern.xml이 추가되었다는 점이다. 그 외에 getBean() 메서드를 이용하여 WriteArticleServiceImpl 객체를

SPRING Container로부터 가져온 뒤 write() 메서드를 실행하는 것은 동일하다.

articleService.write(new Article()); 가 실행되면 콘솔에 다음과 같은 결과가 출력될 것이다.

2009. 10. 20 오후 11:59:35 madvirus.spring.chap01.LoggingAspect logging

정보: 기록 시작

WriteArticleServiceImpl.write() 메서드 실행

MySQLArticleDAO.insert() 실행

2009. 10. 20 dhgn 11:59:39 madvirus.spring.chap01.LoggingAspect logging

정보: 기록 종료

2009. 10. 20 dhgn 11:59:39 madvirus.spring.chap01.LoggingAspect logging

정보: write메서드 실행 시간 : 0

위 실행 결과를 보면 LoggingAspect의 Code가 WriteArticleServiceImpl Class의 write() 메서드 앞 뒤로 실행되는 것을

확인할 수 있다. LoggingAspect.logging() 메서드의 Code를 다시 살펴보자.

public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

log.info("기록 시작");

...

Object retValue = joinPoint.proceed();

...

log.info("기록 종료");

log.info(joinPoint.getSignature().getName() + "메서드 실행 시간 : "

+ stopWatch.getTotalTimeMillis());

}

MainForAop Class에서 articleService.write(new Article()) Code가 실행되면 LoggingAspect.logging() 메서드가 실행된다.

그런 뒤, joinPoint.proceed() Code가 실행될 때 실제 WriteArticleServiceImpl Class의 write() 메서드가 실행된다.

앞서 실행 결과를 보면 LoggingAspect가 적용되었다는 것을 확인할 수 있을 것이다. 실행 순서를 정리하면...아래와 같다.

MainForAop proxy LoggingAspect joinPoint WriteArticleServiceImpl

| | | | |

| 1: write() ----> | | | |

| 2.logging()--> | -----------> StopWatch | |

| 4.start() ------> | | |

| 5.proceed() --- |--------> | |

| | | 6.write()---------------->|

| 7.stop() -------> |

|<----------------|

|<------------------|

SPRING은 프록시를 이용하여 AOP를 구현하고 있다. 위에서 MainForAop Class가 사용하는 객체는 WriteArticleServiceImpl 객체가

아니라 SPRING이 런타임에 생성한 프록시 객체가 되며, 프록시가 내부적으로 다시 LoggingAspect의 logging() 메서드를 호출하여

공통 관심 사항이 실제 WriteArticleServiceImpl 객체에 적용된다.

위의 예제에서 중요한 점은 공통 관심 사항(LoggingAspect)를 적용하는 과정에서 핵심 로직을 구현한 Class(WriteArticleServiceImpl)의

Code를 변경하지 않았다는 점이다. 단지, 공통 관심 사항을 수현한 Aspect Class를 작성하였고, 설정 파일을 이용하여 Aspect를

핵심 로직을 구현한 Class에 적용했을 뿐이다.

어떻게 간단한 설정만으로 공통 기능을 여러 Class에 적용할 수 있는 것이 AOP의 주요 장점 중의 하나이며, SPRING은 JEE Application을

구현하는 데에 필요한 수준으로 AOP를 지원하고 있다. 따라서, SPRING AOP를 이용하면 핵심 로직 Code의 수정 없이 WEB Application에

Security, Logging, Transaction과 같은 공통 관심 사항을 AOP를 이용하여 간단하게 적용할 수 있다.

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

+++++++++++++++++++++++

+Chapter 02 SPRING DI와 객체 관리+

+++++++++++++++++++++++

------------------------------------------------------------------------------------------------------------------------------------------------------------------

: 1장에서 살펴봤듯이, SPRING의 주요 기능 중의 하나는 DI 패턴을 지원한다는 것이다.

SPRING Container는 빈 객체를 저장하고 있으며, 각 객체간의 의존 관계를 관리해 준다.

이 장에서는 SPRING Container에 대해 좀더 자세히 살펴보고, 객체 간의 의존 관계를 설 정하는 방법에 대해서 보다

구체적으로 살펴보도록 하자.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. SPRING Container

: SPRING은 객체를 관리하는 Container를 제공한다.

SPRING은 Container에 객체를 담아 두고, 필요할 때에 Container로부터 객체를 가져와 사용할 수 있도록 하고 있다.

앞서 1장의 예제에서 사용한 BeanFactory와 ApplicationContext가 Container 역할을 수행하는 Interface가 된다.

Container와 관련된 주요 Interface의 관계는 다음과 같다.

<<Interface>> <<Interface>> <<Interface>>

BeanFactory <===== ApplicationContext <===== WebApplicationContext

각 Interface는 제공하는 기능과 용도가 다른데, 각 Interface 및 구현 Class에 대해서 차례대로 살펴보도록 하자.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.1 BeanFactory Interface

: org.springframwork.beans.factory.BeanFactory Interface는 빈 객체를 관리하고 각 객체간의 의존 관계를 설정해 주는

기능을 제공하는 가장 단순한 Container로서, 구현 Class로는 org.springframwork.beans.xml.XmlBeanFactory Class가

존재한다. XmlBeanFactory Class는 ApplicationContext Interface는 구현하지 않았다.

XmlBeanFactory Class는 외부 Resource으로부터 설정 정보를 읽어 와 빈 객체를 생성한다.

SPRING은 org.springframwork.core.io.Resource Interface를 사용하여 다양한 종류의 Resource을 동일한 방식으로

표현할 수 있도록 하고 있으며, Resource를 이용하여 XmlBeanFactory에 설정 정보를 전달할 수 있다.

예를 들어, 특정 파일로부터 설정 파일을 읽어 오는 XmlBeanFactory 객체를 생성하고 싶다면

다음과 같은 Code를 사용하면 된다.

Resource resource = new FileSystemResource("beans.xml");

XmlBeanFactory factory = new XmlBeanFactory(resource);

AgentService agent = (AgentService)factory.getBean("agent");

SPRING은 FileSystemResource Class뿐만 아니라 다양한 Resource 구현 Class를 제공하고 있으며,

이들 Class는 아래와 같다.

\*\*\*Resource 구현 Class

org.springframework.core.io.FileSystemResource : 파일 시스템의 특정 파일로부터 정보를 읽어 온다.

org.springframework.core.io.InputStreamResource : InputStream으로부터 정보를 읽어 온다.

org.springframework.core.io.ClassPathResource : Class 패스에 있는 Resource으로부터 정보를 읽어 온다.

org.springframework.core.io.UrlResource : 특정 URL로부터 정보를 읽어 온다.

org.springframework.web.context.support.ServletContextResource : WEB Application의 루트 디렉터리를 기준으로

지정한 경로에 위치한 Resource으로부터 정보를 읽어 온다.

특정 Resource로부터 설정 정보를 읽어와 XmlBeanFactory 객체를 생성한 뒤에는 getBean() 메서드를 이용하여

알맞은 빈을 가져와 사용하면 된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

1.2 ApplicationContext Interface와 WebApplicationContext Interface

: org.springframework.context.ApplicationContext Interface는 BeanFactory Interface를 상속받은 하위 Interface로서

BeanFactory가 제공하는 빈 관리 기능이외에 빈 객체 라이프 사이클, 파일과 같은 Resource 처리 추상화,

Message 지원 및 국제화 지원, 이벤트 지원, XML 스키마 확장을 통한 편리한 설정 등 추가적인 기능을 제공하고 있다.

따라서, SPRING을 이용하여 Application을 개발할 때에는 단순히 빈 객체 생성 기능만을 제공하는 BeanFactory보다는

ApplicationContext Interface의 구현 Class를 주로 사용하게 된다.

org.springframework.web.context.WebApplicationContext Interface는 WEB Application을 위한

ApplicationContext로서 하나의 WEB Application(즉, 하나의 ServletContext)마다 한 개 이상의

WebApplicationContext를 가질 수 있다.

ApplicationContext Interface 및 WebApplicationContext Interface를 구현한 Class 중에서 주요 Class는 다음과 같다.

- org.springframework.context.support.ClassPathXmlApplicationContext

: Class 패스에 위치한 XML파일로부터 설정 정보를 로딩한다.

- org.springframework.context.support.FileSystemXmlApplicationContext

: 파일 시스템에 위치한 XML파일로부터 설정 정보를 로딩한다.

- org.springframework.web.context.support.XmlWebApplicationContext

: WEB Application에 위치한 XML파일로부터 설정 정보를 로딩한다.

예를 들어, Class 패스에 위치한 config/ApplicationContext.xml 파일로부터 설정 정보를 읽어와 ApplicationContext

인스턴스를 생성하고 싶다면, 다음과 같이 ClassPathXmlApplicationContext Class를 사용하면 된다.

String configLoaction = "config/ApplicationContext.xml";

ApplicationContext context = new ClassPathXmlApplicationContext(configLoaction);

ParserFactory factory = (ParserFactory)context.getBean("parserFactory");

Class 패스에 위치하고 있는 여러 XML 파일을 이용하여 ApplicationContext를 생성하고 싶다면, 설정 파일 목록을

담고 있는 String 배열을 ClassPathXmlApplicationContext Class 생성자에 전달하면 된다.

String[] configLocations = new String[]{"config/ApplicationContext.xml","config/aop.xml"};

ApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

SomeBean bean = context.getBean("beanName");

XmlWebApplicationContext Class는 WEB Application에서 사용 가능한 ApplicationContext로서, WEB Application에

위치한 Resource으로부터 설정 정보를 로딩할 수 있다. 개발자가 직접 XmlWebApplicationContext 객체를 생성하는 경우는

드물며, WEB Application에서 web.xml 파일에 설정을 통해 XmlWebApplicationContext 객체를 생성하고 사용하게 된다.

<context-param>

<param-name>contextConfigLoaction</param-name>

<param-value>WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

생성된 XmlWebApplicationContext 객체는 WebApplicationContextUtils Class를 이용하여 구할 수 있다.

WebApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

WriteArticleService writeArticleService = (WriteArticleService)context.getBean("writeArticleService");

SPRING MVC를 이용하는 경우 WebApplicationContextUtils Class를 직접적으로 사용할 경우는 거의 없으며,

본 문서에서도 위와 같은 Code를 사용하는 경우는 매우 드물 것이다.

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

2. 빈(Bean) 생성과 의존 관계 설정

: SPRING은 객체를 담고 있는 Container다. Container에 저장될 빈 객체와 각 빈 객체간의 연관 관계는 앞서 1장에서

살펴봤듯이 XML파일을 통해서 설정할 수 있다. SPRING 설정 파일의 구성 요소는 매우 직관적이어서 조금만 시간을 내어

학습하면 어려움 없이 설정 파일을 작성할 수 있을 것이다. 본 절에서는 SPRING Container에서 사용될 빈 객체를 생성하고

각 객체간의 의존 관계를 설정하는 방법을 살펴보도록 하겠다.

SPRING은 XML 파일을 이용한 설정 뿐만 아니라 Annotation을 이용한 설정도 지원하고 있는데, Annotation을 이용한

설정 방법은 '4장. Annotation 기반 설정 및 Java 기반 설정'에서 살펴보도록 하겠다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.1 빈 객체 설정 및 Container를 통한 빈 객체 사용

: SPRING 객체가 관리할 빈 객체를 생성하기 위해서는 <bean>태그를 사용하면 된다.

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

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">

<bean id="articleDao" class="madvirus.spring.chap01.MySQLArticleDao">

</bean>

</beans>

<bean>태그의 class속성은 생성할 빈 객체의 완전한 Class 이름을 입력하며, id속성은 SPRING Container에서

생성된 객체를 구분하는 데 사용되는 식별 값을 의미한다. id속성 대신 name속성을 사용해도 된다.

<bean name="articleDao" class="madvirus.spring.chap01.MySQLArticleDao">

</bean>

id속성은 XML의 ID 구성 요소이기 때문에 id속성을 사용할 경우 동일한 id값을 갖는 <bean> 태그를 설정하면

XML 문서 검증 과정에서 예외가 발생한다. 또한, id속성은 슬래시('/')나 공백문자(' '), 콤마(',')와 같은 특수 문자를

포함할 수 없다. 반면에 name속성을 사용할 경우 슬래시('/')나 공백문자(' '), 콤마(',')와 같은 특수 문자를 이름으로

사용할 수 있다.

<bean>태그를 이용하여 생성할 빈 객체에 대한 정보를 설정했다면, ApplicationContext나 BeanFactory를 이용하여

SPRING Container를 생성한 뒤, Container로부터 빈 객체를 가져와 사용할 수 있다. 다음은 XmlBeanFactory Class를

이용해서 빈 객체를 가져와 사용하는 Code를 보여주고 있다.

Resource resource = new ClassPathResource("applicationContext.xml");

//SPRING Container 생성

BeanFactory beanFactory = new XmlBeanFactory(resource);

//SPRING Container로부터 빈 객체를 가져와 사용

MySQLArticleDAO articleDAO =

(MySQLArticleDAO)beanFactory.getBean("articleDAO");

XML파일에서 <bean>태그의 id속성 또는 name속성에 명시한 값이 빈 객체의 식별 값이라고 했는데, 이 식별 값을 getBean()

메서드에 전달해서 Container로부터 객체를 가져올 수 있다.

getBean() 메서드의 리턴 타입은 Object이기 때문에 위 Code와 같이 형 변환을 해 주어야 한다.

SPRING3 버전부터 타입 변환 대신 Java5의 제너릭을 이용해서 원하는 타입으로 빈을 구할 수 있는 getBean() 메서드를

제공하고 있으며, 사용법은 아래 Code와 같다.

ArticleDAO articleDAO =

beanFactory.getBean("articleDAO",ArticleDAO.class);

<bean> 태그를 사용할 때 주의할 점은 생성자와 관련된 정보(<constructor-arg>태그)를 명시하지 않았다면, SPRING은

인자가 없는 기본 생성자를 이용해서 객체를 생성한다는 점이다. 따라서, 빈으로 사용할 Class에 인자를 갖는 생성자를

정의했다면 기본 생성자도 함께 정의해 주어야 한다.

public class MySQLArticleDAO implements ArticleDAO{

//파라미터를 갖는 생성자가 정의되어 있는 경우, 기본 생성자 추가해야 함!

public MySQLArticleDAO(){}

public MySQLArticleDAO(DataSource dataSource){

...

}

}

기본 생성자가 아닌 1개 이상의 인자를 갖는 생성자를 이용하여 빈 객체를 생성하는 방법도 있는데, 이에 대해서는

본 장의 '의존 관계 설정'에서 살펴보도록 하겠다.

\*\*\*빈 팩토리 메서드

: 경우에 따라서 생성자가 아닌 static 메서드를 이용하여 객체를 생성해야 하는 경우도 있다. 예를 들어, 아래 Code와 같이

싱글톤 패턴이 적용된 Class를 생각해 보자.

public class ParserFactory{

private static ParserFactory instance = new ParserFactory();

public static ParserFactory getInstance(){

return instance;

}

// 기본 생성자 접근 막음.

private ParserFactory(){}

...

}

ParserFactory Class는 생성자를 사용하지 않고 static 메서드인 getInstance() 메서드를 이용하여 ParserFactory

객체를 생성하도록 구현되어 있다. 이렇게 static 메서드를 이용하여 객체를 구해야 하는 Class의 경우, <bean> 태그에

factory-method 속성 값으로 static 메서드를 지정함으로써 해당 메서드를 이용하여 빈을 생성하도록 설정할 수 있다.

<bean id="parserFactory" class="madvirus.spring.chap02.ParserFactory" factory-method="getInstance"/>

위와 같이 factory-method 속성을 지정하면 SPRING Container는 생성자를 사용하지 않고 getInstance() 메서드를

이용하여 ParserFactory Class의 객체를 구하게 된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 의존 관계 설정

: SPRING Container에서 각 객체 간의 의존 관계도 XML 파일을 통해 명시할 수 있다.

객체간의 의존 관계는 생성자, 프로퍼티 설정 그리고 룩업 메서드 인잭션의 세 가지 방식으로 지정할 수 있으며,

본 절에서는 각 방식(생성자 방식, 프로퍼티 설정 방식, 룩업 메서드 방식)에 대해서 차례대로 살펴보도록 하자.

(1) 생성자 방식

: 생성자 방식은 의존하는 객체를 생성자를 통해서 전달받는다. 예를 들어 아래와 같은 의존 관계가 있다고 하자.

WriteArticleServiceImpl ===============> <<Interface>>

+write(article : Article) ArticleDAO

+insert(article: Article)

||

||

OracleArticleDAO

WriteArticleServiceImpl Class는 ArticleDAO Interface에 의존하고 있는데, 다음과 같이 생성자를 통해 의존하는 객체를

전달받을 수 있을 것이다.

public class WriteArticleServiceImpl{

private ArticleDAO articleDAO;

public WriteArticleServiceImpl(ArticleDAO articleDAO){

this.articleDAO = articleDAO;

}

public void write(Article article){

...

articleDAO.insert(article);

}

}

생성자를 통해서 의존하는 객체를 전달받는 경우, <constructor-arg>태그를 이용하여 의존하는 객체를 전달할 수 있다.

아래는 <constructor-arg>태그를 이용한 생성자 방식의 설정 예를 보여 주고 있다.

<bean id="writeArticleService" class="madvirus.spring.chap02.WriteArticleServiceImpl">

<constructor-arg>

<ref bean="articleDAO"/>

</constructor-arg>

</bean>

<bean id="articleDAO" class="madvirus.spring.chap02.OracleArticleDAO">

</bean>

위 Code에서 <ref>태그는 Reference를 의미하며, 위 설정의 의미는 WriteArticleServiceImpl Class의 생성자에 식별값이

'articleDAO'인 빈 객체를 전달한다는 것을 의미한다. 위 설정을 동일한 Code로 표현하면 다음과 같다.

OracleArticleDAO articleDAO = new OracleArticleDAO();

WriteArticleServiceImpl writeArticleService = new WriteArticleServiceImpl(articleDAO);

<ref>태그를 사용하는 대신, <constructor-arg>태그의 ref 속성을 사용해도 된다.

<bean id="writeArticleService" class="madvirus.spring.chap02.WriteArticleServiceImpl">

<constructor-arg ref="articleDAO"/>

</bean>

생성자가 전달 받는 값이 int나 double과 같은 기본 데이터 타입이나 java.lang.String 타입이라면 <ref> 태그 대신에

<value>태그를 사용하여 값을 지정한다. 아래 Code는 <value> 태그의 사용 예를 보여 주고 있다.

<bean name="monitor" class="madvirus.spring.chap02.SystemMonitor">

<constructor-arg>

<value>10</value>

</constructor-arg>

</bean>

<value> 태그를 사용하지 않고 <constructor-arg> 태그의 value 속성을 이용해서 생성자에 전달할 값을 지정할 수도 있다.

<bean name="monitor" class="madvirus.spring.chap02.SystemMonitor">

<constrictor-arg value=10/>

</bean>

생성자가 전달받는 파라미터가 2개 이상인 경우에는 파라미터 개수만큼 <constructor-arg> 태그를 설정해 주어야 한다.

아래 Code는 두개의 <constructor-arg> 태그를 사용하는 설정 예이다.

<bean name="monitor" class="madvirus.spring.chap02.SystemMonitor">

<constructor-arg value=10/>

<constructor-arg ref="sender"/>

</bean>

SPRING은 <constructor-arg> 태그를 통해 전달된 값의 타입과 가장 근접한 타입의 파라미터를 갖는 생성자를 이용하여

객체를 생성한다. 예를 들어, 다음과 같은 두 개의 생성자를 갖는 Class가 있다고 하자.

public class JobExecutor{

public JobExecutor(Runnable run){

}

public JobExecutor(Job job){

}

}

Job Class는 Runnable Interface를 구현한 Class이고, SomeRunnable Class는 Runnable Interface를 구현한 Class라고 하자.

public class Job implements Runnable{...}

public class SomeRunnable implements Runnable{...}

이 상태에서 다음과 같이 SPRING 설정 파일을 작성했다고 하자.

<bean name="run" class="madvirus.SomeRunnable"/>

<bean name="job" class="madvirus.Job"/>

<bean name="executor1" class="madvirus.JobExecutor">

<constructor-arg ref="run"/>

</bean>

<bean name="executor2" class="madviurs.JobExecutor">

<constructor-arg ref="job"/>

</bean>

executor1 빈을 생성하기 위해 <constructor-arg> 태그에 전달된 SomeRunnable 객체인 run 빈은 Runnable 타입이므로

Runnable을 전달 받는 JobExecutor 생성자가 사용된다.

반면에 executor2 빈을 생성하기 위해 <constructor-arg> 태그에 전달된 job빈은 Runnable 타입이면서 Job 타입이다.

JobExecutor Class는 Job 타입을 전달받는 생성자를 제공하고 있으므로, executor2 빈을 생성할 때는 Job 타입을

전달받는 JobExecutor 생성자가 사용된다.

두 개 이상의 생성자에서 기본 데이터 타입이나 Integer와 같은 레퍼 타입을 사용해서 어떤 생성자를 사용해야 할지

결정할수 없는 경우가 있다. 예를 들어, 다음과 같이 세 개의 생성자를 제공하는 Class가 있다고 하자.

public class JobExecutor{

public JobExecutor(String name, int seconds){...}

public JobExecutor(String name, long millis){...}

public JobExecutor(String name, String seconds){...}

}

생성자 방식을 이용하여 값을 전달하기 위해 <constructor-arg> 태그를 다음과 같이 설정하게 될 것이다.

<bean name="executor" class="madvirus.spring.chap02.JobExecutor">

<constructor-arg>

<value>실행기1</value>

</constructor-arg>

<constructor-arg>

<value>3000</value>

</constructor-arg>

</bean>

이 때 사용되는 생성자는 무엇일까? SPRING은 기본적으로 <value>에 전달된 값을 java.lang.String 타입으로 처리한다.

따라서 위 Code에서는 두 번째 파라미터가 String 타입인 생성자를 사용해서 객체를 생성하게 된다.

만약 매칭되는 파라미터의 타입이 String인 생성자가 존재하지 않을 겨우 <value>에 전달된 값에 따라 알맞은 데이터 타입을

선택하게 된다. 따라서, 두 번째 파라미터 타입이 String 타입인 생성자가 존재하지 않는다면, 3000 값이 정수 타입에

해당하므로 두 번째 파라미터 타입이 int인 생성자가 사용된다.

앞서 예로 든 세 개의 생성자 중에서 두 번째 파라미터가 long 타입인 생성자를 사용하려면 설정 파일에 직접적으로 파라미터 타입을

명시해 주어야 한다. <value> 태그에 type 속성을 추가하면 직접 매칭될 파라미터의 타입을 명시할 수 있다.

<bean name="executor" class="madvirus.spring.chap02.JobExecutor">

<constructor-arg>

<value>실행기1</value>

</constructor-arg>

<constructor-arg>

<value type="long">3000</value>

</constructor-arg>

</bean>

<value> 태그뿐만 아니라 <constructor-arg>태그에 type 속성을 추가해도 동일한 동작을 한다.

(2) 프로퍼티 설정 방식

: 생성자 방식이 생성자를 사용해서 필요한 객체나 값을 전달받는다면, 프로퍼티 설정 방식은 setXXX() 형태의 설정 메서드를

사용해서 필요한 객체와 값을 전달 받는다. 앞서 작성했던 WriteArticleServiceImpl Class가 setArticleDAO() 메서드를 통해서

ArticleDAO 객체를 전달받는다면 다음과 같은 Code가 작성될 것이다.

public class WriteArticleServiceImpl implements WriteArticleService{

private ArticleDAO articleDAO;

public void setArticleDAO(ArticleDAO articleDAO){

this.articleDAO = articleDAO;

}

}

setXXX() 형태의 메서드에서 프로퍼티의 이름은 XXX가 된다. 예를 들어, 위 Code에서 setArticleDAO() 메서드는 articleDAO 프로퍼티의

값을 설정하는 메서드가 된다.

SPRING에서는 <property> 태그를 이용하여 프로퍼티에 값을 전달한다. 아래 Code는 <property> 태그를 사용해서 WriteArticleServiceImpl

빈 객체의 articleDAO 프로퍼티 값을 설정하는 예를 보여 주고 있다.

<bean name="writeArticleService" class="madvirus.spring.chap02.WriteArticleServiceImpl">

<property name="articleDAO">

<ref bean="mysqlArticleDAO"/>

</property>

</bean

<bean name="mysqlArticleDAO" class="madvirus.spring.chap02.MySQLArticleDAO"/>

<constructor-arg> 태그에서 <ref>태그를 이용하여 다른 빈 객체를 전달했듯이, <property> 태그에서도 <ref> 태그를 이용하여

빈 객체를 프로퍼티 값을 전달할 수 있다.

기본 데이터 타입의 경우에는 <ref> 태그 대신 <value> 태그를 사용하면 된다.

<bean name="lockingFailManager" class="madvirus.spring.chap02.PessimisticLockingFailManager">

<property name="retryCount">

<value>3</value>

</property>

</bean>

<property> 태그도 ref 속성과 value 속성을 이용하여 각각 빈 객체와 기본 데이터 타입 값을 전달할 수 있다.

<bean name="monitor" class="madirus.spring.chap02.SystemMonitor">

<property name="periodTime" value="10"/>

<property name="sender" ref="smsSender"/>

</bean>

(3) XML 네임스페이스를 이용한 프로퍼티 설정

: XML 네임스페이스를 이용하면 <property> 태그를 사용하지 않고 좀더 간단한 방법으로 프로퍼티 값을 설정할 수 있다.

아래 Code는 XML 네임스페이스를 이용하여 프로퍼티 값을 설정하는 예를 보여주고 있다.

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="monitor" class="madvirus.spring.chap02.SystemMonitor" p:periodTime="10" p:sender-ref="smsSender" />

<bean id="smsSender" class="madvirus.spring.chap02.SmsSender" />

</beans>

네임스페이스가 "http://www.springframework.org/schema/p"인 접두어를 사용하면 프로퍼티 값을 간편하게 설정할 수 있다.

위 Code에서는 접두어를 'p'로 설정했는데, 기본 데이터 타입을 설정할 때는 'p:프로퍼티이름' 속성을 사용하여 프로퍼티 값을

설정하고, 'p:프로퍼티이름-ref' 속성을 사용하여 빈 객체를 프로퍼티에 전달할 수 있다.

위 Code에서는 'periodTime' 프로퍼티의 값을 10으로 설정하였고, sender 프로퍼티에 smsSender 빈 객체를 전달하고 있다.

(4) 룩업 메서드 인젝션 방식

: 룩업 메서드 인젝션 방식은 필요한 객체를 제공하는 룩업(look up) 메서드를 구현하는 방식이다. 다음 Code는 룩업 메서드를

이용해서 의존 객체를 구하는 Class의 예를 보여주고 있다.

public class Processor{

public void process(String commandName){

CommandFactory factory = getCommandFactory();

Command command = factory.createCommand(commandName);

command.execute();

}

protected abstract CommandFactory getCommandFactory();

}

위 Code에서 process() 메서드는 CommandFactory를 이용해서 Command 객체를 생성하고 있는데, getCommandFactory()

메서드를 이용해서 CommandFactory Class의 인스턴스를 생성하고 있다. 여기서 getCommandFactory() 메서드가 의존 객체를

구할 때 사용되는 룩업 메서드이다.

SPRING에서 룩업 메서드에 대한 인젝션을 수행하려면 설정 파일에서 <lookup-method> 태그를 사용하면 된다.

<bean id="processor" class="madvirus.spring.chap02.Processor">

<lookup-method name="getCommandFactory" bean="commandFactory"/>

</bean>

<bean id="commandFactory" class="madvirus.spring.chap02.CommandFactory"/>

<lookup-method> 태그의 name 속성은 룩업 메서드의 이름을 입력하고, bean 속성은 룩업 메서드가 리턴할 빈 객체의 식별값을

입력한다. 예를 들어, 위 Code는 룩업 메서드로 getCommandFactory() 메서드를 지정하였고, getCommandFactory() 메서드가

commandFactory 빈을 리턴하도록 설정하였다. 따라서, Processor.process() 메서드는 commandFactory 빈을 사용하게 된다.

SPRING은 상속을 이용해서 룩업 메서드를 구현 (오버라이드)하기 때문에, 룩업 메서드는 다음과 같은 규칙을 따라야 한다.

- 접근 수식어가 public 이나 protected여야 한다.

- 리턴 타입이 void가 아니다.

- 인자를 갖지 않는다.

- 추상 메서드여도 된다.

- final이 아니다.

또한, 추가로 상속을 통해서 룩업 메서드 인젝션을 구현하기 때문에 Class가 final이어서는 안된다.

록업 메서드가 추상 메서드가 아닐 경우, 아래 Code와 같이 임의로 구현해 주면 된다.

public class Processor{

public void process(String commandName){

CommandFactory factory = getCommandFactory();

...

}

protected CommandFactory getCommandFactory(){

return null;

}

}

(5) 임의의 빈 객체 전달

: <constructor-arg> 태그나 <property> 태그는 <ref> 태그 또는 ref 속성을 이용하여 다른 빈 객체를 전달하였다.

하지만, 식별값을 갖지 않는 빈 객체를 생성해서 전달할 수도 있다. 이를 위해서는 <constructor-arg> 태그나

<property> 태그에 <bean> 태그를 중첩해서 사용하면 된다.

<bean id="monitor" class="madvirus.spring.chap02.SystemMonitor" p:periodTime="10">

<property name="sender">

<!--임의 빈 객체-->

<bean class="madvirus.spring.chap02.SmsSender">

<constructor-arg value="true"/>

</bean>

</property>

</bean>

임의 빈 객체는 식별값을 갖지 않기 때문에, 임의 빈 객체를 재사용할 수 없다는 점에 유의해야 한다. 임의 빈 객체를 위한

<bean> 태그에 id 나 name 속성을 사용하여 식별값을 준다 하더라도 해당 식별값은 사용할 수 없다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.3 컬렉션 타입 프로퍼티 설정

: 프로퍼티의 타입이 java.util.List나 java.util.Map과 같은 컬렉션 타입인 경우, SPRING은 각 타입에 알맞은 태그를 이용하여

값을 설정할 수 있도록 하고 있다. 컬렉션 타입의 값을 입력 받기 위해 사용되는 SPRING 태그 목록은 다음과 같다.

<list> : java.util.List/Java배열 : List 타입이나 배열에 값 목록을 전달할 때 사용된다.

<map> : java.util.Map : Map 타입에 <키,값> 목록을 전달할 때 사용된다.

<set> : java.util.Set : Set 타입에 값 목록을 전달할 때 사용된다.

<properties> : java.util.Properties : Properties 타입에 <프로퍼티 이름, 프로퍼티 값> 목록을 전달할 때 사용된다.

각 태그가 어떻게 사용되는 지 차례대로 살펴보도록 하자.

(1) List 타입과 배열

: java.util.List 타입의 프로퍼티나 생성자 인자에 값 목록을 전달하기 위해서는 <list> 태그를 사용하면 된다.

아래 Code와 같이 List 타입의 프로퍼티를 갖는 Class를 예로 들어 설정 방법을 설명해 보겠다.

public class ProtocolHandler{

private List<Filter> filters;

public void setFilters(List<Filter> filters){

this.filters = filters;

}

...

}

위 Code에서 ProtocolHandler Class는 List 타입의 filters 프로퍼티를 갖는다. 이 경우, <list> 태그를 이용하여

List에 저장될 객체 목록을 지정할 수 있다.

<bean name="handler" class="madvirus.spring.chap02.ProtocolHandler">

<property name="filters">

<list>

<ref bean="encryptionFilter"/>

<ref bean="zipFilter"/>

<bean class="madvirus.spring.ch02.HeaderFilter"/>

</list>

</property>

</bean>

<bean name="zipFilter" class="madvirus.spring.ch02.ZipFilter"/>

<bean name="encryptionFilter" class="madvirus.spring.ch02.EncryptionFilter"/>

<list> 태그는 List에 저장될 객체 목록을 <ref> 태그를 이용하여 전달받는다. 또한 <bean> 태그를 이용하여

식별값을 갖지 않는 임의 객체를 List에 전달할 수도 있다.

List에 저장되는 값이 Integer, Long과 같은 레퍼 타입이거나 String 타입이라면 <value> 태그를 사용하여 값을 전달할 수 있다.

<bean name="monitor" class="madvirus.spring.chap02.PerformanceMonitor">

<property name="deviations">

<list>

<value>0.2</value>

<value>0.3</value>

</list>

</property>

</bean>

List 타입의 프로퍼티가 저장하는 값들의 타입이 래퍼 타입 내지 String 타입인 경우 타입을 지정하는 것에 주의를 해야 한다.

예를 들어, 위 설정 파일과 관련된 Class의 Code가 아래와 같고, deviations 멤버 필드에 저장되는 값이 java.lanag.Double 이어야 한다고 하자.

public class PerformanceMonitor{

private List deviations;

public PerformanceMonitor(List deviations){

this.deviations = deviations;

}

}

SPRING은 기본적으로 <value> 태그에 명시된 값을 String 타입으로 처리하기 때문에, List에 전달되는 값들은 String이 된다.

따라서, List에 저장될 객체가 String이 아닌 Integer나 Boolean과 같은 래퍼 타입이라면, <list> 태그에 value-type 속성을 이용하여

List에 저장될 값들의 타입을 명시해 주어야 한다.

<bean name="monitor" class="madvirus.spring.chap02.PerformanceMonitor">

<property name="deviations">

<list value-type="java.lang.Double">

<value>0.2</value>

<value>0.3</value>

</list>

</property>

</bean>

<list> 태그가 아니라 <value> 태그의 type 속성을 이용하여 타입을 명시해도 된다.

<bean name="monitor" class="madvirus.spring.chap02.PerformanceMonitor">

<property name="deviations">

<list>

<value type="java.lang.Double">0.2</value>

<value type="java.lang.Double">0.3</value>

</list>

</property>

</bean>

Java5부터 제공하는 제너릭을 이용할 경우 <list>나 <value> 태그에 값의 타입을 명시하지 않아도 알맞게 타입을 변환해 준다.

예를 들어, 아래 Code와 같이 제너릭을 사용하여 List에 저장될 값의 타입을 명시해 주었다고 하자.

public class PerformanceMonitor{

private List<Double> deviations;

public PerformanceMonitor(List<Double> deviations){

this.deviations = deviations;

}

}

이 경우 설정 파일에 타입을 명시해 주지 않아도 된다. 즉, 아래와 같이 타입을 명시하지 않더라도 SPRING이 알맞게

<value>에 명시된 값을 java.lang.Double 타입으로 알맞게 변환해 준다.

<bean name="monitor" class="madvirus.spring.chap02.PerformanceMonitor">

<property name="deviations">

<list>

<value>0.2</value>

<value>0.3</value>

</list>

</property>

</bean>

<list> 태그는 <value>나 <ref>태그 뿐만 아니라 <list>나 <map>등의 태그를 이용하여 값을 표현할 수도 있다.

아래는 List에 저장될 값을 표현하기 위해 사용 가능한 태그 목록이다.

-<ref> : 다른 SPRING 빈 객체를 값으로 사용.

-<bean> : 임의 빈 객체를 생성해서 값으로 사용.

-<value> : 래퍼 타입이나 String을 값으로 사용.

-<list>,<map>,<props>,<set> : 컬렉션 객체를 값으로 사용.

-<null> : null Reference를 값으로 사용.

<list> 태그는 배열을 설정할 때에도 사용된다. 예를 들어, 프로퍼티가 다음 Code와 같이 Filter 타입이라고 하자.

public class ProtocolHandler{

private Filter[] filters;

public void setFilters(Filter[] filters){

this.filters=filters;

}

...

}

(2) Map 타입

: Map 타입의 프로퍼티를 설정하는 경우에는 <map> 태그를 사용하면 된다. 예를 들어, 아래와 같이 키 타입이 String이고

값 타입이 ProtocolHandler인 handlers 프로퍼티를 갖는 Class가 있다고 하자.

public class ProtocolHandlerFactory{

private Map<String, ProtocolHandler> handlers;

public void setHandlers(Map<String, ProtocolHandler> handlers){

this.handlers=handlers;

}

...

}

이때 handlers 프로퍼티에 Map 타입의 값을 전달하려면 아래와 같이 <map> 태그를 사용하면 된다.

<bean name="handlerFactory" class="madvirus.spring.chap02.ProtocolHandlerFactory">

<property name="handlers">

<map>

<entry>

<key><value>soap</value></key>

<ref bean="soapHandler"/>

</entry>

<entry>

<key><value>rest</value></key>

<ref bean="restHandler"/>

</entry>

</map>

</property>

</bean>

한 개의 <entry> 태그는 Map에 저장될 한 개의 <키,값>을 표현한다. <entry> 태그는 두 개의 자식 노드를 가지며,

각각 키와 값을 의미한다.

<map>

<entry>

<key><키태그>...</키태그></key>

<값태그>...</값태그>

</entry>

</map>

<key> 태그를 이용하여 키에 할당된 값을 지정하게 되는데, 이때 <키태그>에는 다음과 같은 태그가 위치할 수 있다.

-<ref> : 다른 SPRING 빈 객체를 키로 사용.

-<bean> : 임의 빈 객체를 생성해서 키로 사용.

-<value> : 래퍼 타입이나 String을 키로 사용.

-<list>,<map>,<props>,<set> : 컬렉션 객체를 키로 사용.

-<null> : null Reference를 키로 사용.

<값태그>에도 <키태그>와 마찬가지로 위와 동일한 태그를 사용하여 값을 지정할 수 있다. 키와 값의 타입이 각각

String과 Integer라면 <value> 태그를 사용하여 <키,값>을 지정하고, 빈 객체를 참조한다면 <ref> 테그를 이용하여

<키,값>을 지정하게 된다. 아래 Code는 설정 예를 보여 주고 있다.

<map>

<entry>

<key><value>1</value></key>

<value type="java.lang.Integer">1th</value>

</entry>

<entry>

<key><ref bean="protocol"/></key>

<ref bean="handler"/>

</entry>

</map>

Map에 저장될 하나의 <키,값> 쌍을 표현하기 위해 위 Code처럼 적지않은 분량의 Code를 사용해야 하는데, 이는 꽤 성가신

일이 될 수도 있다.그래서, SPRING은 좀더 간단한 표현을 위해 <entry> 태그의 key, key-ref, value, value-ref 속성을 이용하여

<키,값> 쌍을 표현할 수 있도록 하고 있다. 이들 속성을 이용하면 위 설정 Code를 다음과 같이 간략하게 변경할 수 있다.

<map>

<entry key="1" value="1th"/>

<entry key-ref="protocol" value-ref="handler"/>

</map>

래퍼 타입이나 String을 키나 값에 할당할 때에는 key 속성과 value 속성을 사용하고, 빈 객체를 참할 때에는 key-ref나 value-ref를

사용하여 설정하면 된다.

앞서 <list> 태그에서 제너릭을 사용하지 않은 경우, 래퍼 타입을 List에 저장하려면 value-type 속성을 이용하여 값의 타입을

명확하게 지정해 줄 필요가 있었듯이, <map> 태그를 사용할 때에도, 제너릭을 사용하지 않은 상태에서 래퍼 타입의 키나 값을

설정하려면 key-type 속성과 value-type 속성을 이용하여 키와 값의 타입을 알맞게 지정해 주어야 한다.

<property name="ratio">

<map key-type="java.lang.Integer" value-type="java.lang.Double">

<entry .../>

...

</map>

</property>

(3) Properties 타입

: java.util.Properties Class는 특별한 타입의 Map으로서 키 값이 모두 String인 Map이다. 보통 Properties Class는

환경 변수나 설정 정보와 같이 상황에 따라 변경되는 값을 저장하기 위한 용도로 주로 사용된다.

SPRING에서는 <props> 태그를 이용하여 Properties 타입의 프로퍼티를 설정한다. 아래는 <props> 태그를 이용한 설정 예이다.

<bean name="client" class="madvirus.spring.ch02.BookClient">

<property name="config">

<props>

<prop key="server">192.168.1.100</prop>

<prop key="connectionTimeout">5000</prop>

</props>

</property>

</bean>

<prop> 태그는 한 개의 프로퍼티를 표현하며, 프로퍼티의 이름은 key 속성을 이용하여 입력하고, 프로퍼티의 값은 <prop> 태그의

몸체 내용을 이용하여 입력한다. 따라서, 위 Code에서 "server" 와 "connectionTimeout"은 프로퍼티의 이름이 되고,

"192.168.1.100"과 "5000"은 각 프로퍼티의 값이 된다. 한 개의 <prop> 태그는 Properties 태그에 한 개의 프로퍼티로 저장이 되므로,

아래 Code와 같이 Properties.getProperty(String name) 메서드를 이용하여 프로퍼티 값을 사용할 수 있다.

public class BookClient{

private Properties config;

public void setConfig(Properties config){

this.config=config

}

public void connet(){

String serverIp = config.getProperty("server");

}

}

(4) Set 타입

: Java에서 중복을 허용하지 않는 집합을 표현할 때에는 java.util.Set 컬렉션 타입을 사용하는데, SPRING에서 Set 타입의 프로퍼티를

설정할 때에는 <set> 태그를 이용하면 된다. <set> 태그는 집합에 저장될 값을 <value> 태그나 <ref> 등의 태그를 이용하여 표시한다.

아래는 Integer 래퍼 타입의 값들을 원소로 갖는 집합을 설정하는 예를 보여 주고 있다.

<property name="subset">

<set value-type="java.lang.Integer">

<value>10</value>

<value>20</value>

<value>30</value>

</set>

</property>

래퍼 타입인 경우 <list>나 <map>의 경우 처럼 value-type 속성을 이용하거나 <value>태그의 type 속성을 이용하여 저장될 값의

타입을 명시해 주어야 한다.

<list>와 마찬가지로, 아래의 태그를 이용하여 값을 지정할 수 있다.

-<ref> : 다른 SPRING 빈 객체를 값으로 사용.

-<bean> : 임의 빈 객체를 생성해서 값으로 사용.

-<value> : 래퍼 타입이나 String을 값으로 사용.

-<list>,<map>,<props>,<set> : 컬렉션 객체를 값으로 사용.

-<null> : null Reference를 값으로 사용.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

2.4 의존 관계 자동 설정

: 빈 객체 간의 의존 관계를 설정하기 위해 다음과 같이 <property> 태그를 이용하여 설정한다고 하였다.

<bean name="writeArticleService" class="madvirus.spring.chap01.WriteArticleServiceImpl">

<property name="articleDAO">

<ref bean="articleDAO" />

</property>

</bean>

<bean name="articleDAO" class="madvirus.spring.chap01.MySQLArticleDao">

</bean>

위 Code는 단지 writeArticleService 빈 객체에서 articleDAO 객체로의 하나의 의존 관계만을 포함하고 있는데, 의존 관계가 복잡하거나

많아지면 설정 파일도 그 만큼 복잡하고 커지게 될 것이다.

SPRING은 의존하는 빈 객체의 타입이나 이름을 이용하여 의존 객체를 자동으로 설정할 수 있는 기능을 제공하고 있으며, 이를 통해

설정 파일의 크기를 줄일 수 있다. 자동 설정에는 다음과 같이 네 가지 방식이 존재한다.

- byName : 프로퍼티의 이름과 같은 이름을 갖는 빈 객체를 설정한다.

- byType : 프로퍼티의 타입과 같은 타입을 갖는 빈 객체를 설정한다.

- constructor : 생성자 파라미터 타입과 같은 타입을 갖는 빈 객체를 생성자에 전달한다.

- autodetect : contructor 방식을 먼저 적용하고, byType 방식을 적용하여 의존 객체를 설정한다.

위 방식 중에서 특정 방식을 이용하여 의존 객체를 설정하려면 <bean> 태그에 autowire 속성의 값을 알맞게 지정해 주면 된다.

<bean id="monitor" class="madvirus.spring.chap02.SystemMonitor" autowire="byName"/>

설정 파일에 포함된 모든 빈 객체에 대해서 특정 방식의 자동 설정을 적용하고 싶다면 <beans> 태그의 default-autowire 속성의 값에

설정 방식을 지정해 주면 된다.

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

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

default-autowire="byName">

...

</beans>

default-autowire 속성의 기본 값은 자동 설정을 하지 않는 것을 의미하는 "no"이며, <bean> 태그의 autowire 속성의 기본 값은 "default"이다.

"default"는 default-autowire 속성의 값을 사용한다는 것을 의미하며, 따라서 자동 설정과 관련하여 병도 설정을 하지 않는다면 자동 설정을

사용하지 않게 된다.

(1) 프로퍼티 이름을 이용한 의존 관계 자동 설정

: byName 방식은 프로퍼티 이름과 동일한 이름을 갖는 빈 객체를 프로퍼티 값으로 설정한다.

예를 들어, 아래 Code에서 WriteArticleServiceImpl Class가 이름이 "articleDAO"인 프로퍼티를 갖고 있다면,

이름이 "articleDAO"인 빈 객체가 프로퍼티의 값으로 전달된다.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="byName"/>

<bean name="articleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

<bean name="mysqlArticleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

프로퍼티 이름을 사용하여 의존 객체를 결정하기 때문에, Code를 리팩토링 해서 프로퍼티의 이름을 변경하는 경우 기존에 올바르게

동작하던 설정 파일이 올바르게 동작하지 않을 수 있다.

(2) 프로퍼티 타입을 이용한 의존 관계 자동 설정

: byType 방식은 프로퍼티 타입과 동일한 타입을 갖는 빈 객체를 프로퍼티 값으로 설정한다. 예를 들어, 아래 그림과 같이

WriteArticleServiceImpl Class가 ArticleDAO 타입의 "articleDAO" 프로퍼티를 갖는 경우를 생각해 보자.

WriteArticleServiceImpl =======================> <<Interface>>

+setArticleDAO(articleDAO : ArticleDAO) ArticleDAO

+insert(article : Article)

||

||

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

|| ||

OracleArticleDAO MySQLArticleDAO

\*\*\*ArticleDAO 타입의 프로퍼티를 갖는 WriteArticleServiceImpl

이 경우, 다음과 같이 byType 방식의 자동 설정을 이용하여 articleDAO 프로퍼티에 알맞은 빈 객체를 설정할 수 있다.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="byType"/>

<bean name="mysqlArticleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

하지만, 동일한 타입의 빈 객체가 두 개 이상 존재하게 되면 SPRING은 어떤 빈 객체를 사용해야 할 지 알 수 없기 때문에 예외를

발생시키게 된다. 예를 들어, 아래 Code를 보자.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="byType"/>

<bean name="mysqlArticleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

<bean name="oracleArticleDAO" class="madvirus.spring.ch02.OracleArticleDAO"/>

위 Code에서 mysqlArticleDAO 빈 객체와 oracleArticleDAO 빈 객체는 둘 다 ArticleDAO 타입이기 때문에 writeArticleService 빈 객체의

articleDAO 프로퍼티에 두 객체 중 어떤 객체를 넣어야 할 지 결정할 수가 없게 된다. 따라서 SPRING은 빈 객체를 생성할수 없게

되며 예외를 발생시킨다.

이는 byType 방식을 이용할 경우, 프로퍼티 타입과 동일한 타입의 빈 객체를 오직 한개만 설정할 수 있다는 것을 의미하며,

동일한 타입의 빈 객체를 다수 설정해야 하는 경우에는 byType 방식을 사용할 수 없게 된다.

(3) 생성자 파라미터 타입을 이용한 의존 관계 자동 설정

: constructor 방식은 생성자의 byType 방식 마찬가지로 타입을 이용하여 값을 전달하는데, 차이가 있다면 프로퍼티가 아닌

생성자의 파라미터 타입을 사용한다는 점이다. WriteArticleServiceImpl Class가 다음 같은 생성자를 갖고 있다고 하자.

public class WriteArticleServiceImpl{

public WriteArticleServiceImpl(ArticleDAO articleDAO){

...

}

}

이 상태에서 <bean> 태그의 autowire 속성 값을 constructor로 설정하자.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="constructor"/>

<bean name="mysqlArticleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

이 경우 WriteArticleServiceImpl(ArticleDAO articleDAO) 생성자의 첫 번째 파라미터에 ArticleDAO 타입인 mysqlArticleDAO

빈 객체가 전달된다.

constructor 방식의 경우, byType 방식과 마찬가지로 생성자 파라미터 타입과 일치하는 빈 객체가 두 개 이상 존재할 경우

빈 객체를 생성할 때 예외를 발생시킨다.

(4) 생성자 및 프로퍼티 타입을 이용한 자동 설정

: autodetect 방식은 constructor 방식을 먼저 적용하고, constructor 방식을 적용할 수 없는 경우 byType 방식을 적용하여

의존 관계를 설정해 준다. autodetect 방식을 적용하려면 다음과 같이 autowire 속성의 값을 "autodetect"로 설정하면 된다.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="autodetect"/>

<bean name="mysqlArticleDAO" class="madvirus.spring.ch02.MySQLArticleDAO"/>

autodetect 방식은 constructor 방식과 byType 방식을 사용하므로, 동일한 타입의 빈 객체를 2개 이상 정의할 수 없는 단점을

그대로 갖는다.

(5) 자동 설정과 직접 설정의 혼합

: 설정 파일의 크기를 줄여 준다는 측면에서는 자동 설정이 유용하긴 하지만, 경우에 따라서는 프로퍼티 중 일부는 자동 설정을

하지 않고 직접적으로 명시하고 싶을 때가 있을것이다. 이런 경우에는 자동 설정과 함께 <property> 태그를 이용하여

해당 프로퍼티의 값을 직접 설정해 주면 된다.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="byName">

<property name="eventListener" ref="emailAdapter"/>

</bean>

특정 프로퍼티의 값이 자동 설정에 의해 초기화되지 않도록 하려면, <null> 태그를 이용하여 프로퍼티의 값을 null로 설정할 수도 있다.

<bean name="writeArticleService" class="madvirus.spring.ch02.WriteArticleServiceImpl" autowire="byName">

<property name="eventListener"><null></property>

</bean>

2.5 부모 빈을 통한 설정 재사용

: 빈 정보를 설정하다 보면 <bean> 태그의 설정 정보가 중복되는 경우가 있다. 예를 들어, 아래의 설정 Code를 보자.

<bean id="doorMonitor" class="madvirus.spring.ch02.SystemMonitor">

<property name="periodTime" value="10"/>

<property name="sender" ref="smsSender"/>

<bean>

<bean id="lobbyMonitor" class="madvirus.spring.ch02.SystemMonitor">

<property name="periodTime" value="10"/>

<property name="sender" ref="smsSender"/>

<bean>

<bean id="roomMonitor" class="madvirus.spring.ch02.SystemMonitor">

<property name="periodTime" value="20"/>

<property name="sender" ref="smsSender"/>

<bean>

위 설정을 보면 세 개의 빈은 이름만 다를 뿐 나머지 설정은 대부분 동일한 것을 알 수 있다. 이렇게 중복되는 설정을 갖는 빈이

다수 존재할 경우, 중복되는 설정 정보를 담고 있는 부모 빈을 정의 한 뒤, 부모 빈 정보를 재사용하도록 설정할 수 있다.

아래 Code는 부모 빈으로부터 설정 정보를 상속 받는 예를 보여 주고 있다.

<bean id="commonMonitor" class="madvirus.spring.ch02.SystemMonitor" abstract="true">

<property name="periodTime" value="10"/>

<property name="sender" ref="smsSender"/>

<bean>

<bean id="doorMonitor" parent="commonMonitor"/>

<bean id="lobbyMonitor" parent="commonMonitor"/>

<bean id="roomMonitor" parent="commonMonitor">

<property name="periodTime" value="20"/>

</bean>

위 Code에서 commonMonitor 빈은 다수 빈에서 중복되는 설정 정보를 담고 있다. 이 설정 정보에는 빈 객체의 Class를

비롯하여 프로퍼티 설정 등이 포함된다. <bean> 태그의 abstract 속성 값을 true로 지정한 것을 알 수 있는데, abstract 속성의

값을 true로 지정하게 된면 SPRING Container는 해당 빈 객체를 생성하지 않는다. 즉, commonMonitor 빈은 설정 정보만

존재할 뿐 실제로 객체는 생성되지 않는다.

자식 빈에서 parent 속성을 사용하여 Class 및 프로퍼티 설정 정보를 물려 받을 부모 빈을 설정하면 된다.

위 Code의 경우 doorMonitor, lobbyMonitor, roomMonitor 빈이 모두 commonMonitor 빈의 설정 정보를 사용하도록

설정하고 있다. 따라서, 이 세 빈은 모두 class 속성의 값으로 madvirus.spring.ch02.SystemMonitor를 사용하게 된다.

parent 속성을 이용하여 물려 받은 설정 정보 중에서 변경하고 싶은 값이 있다면 추가로 입력해 주면 된다. 예를 들어,

commonMonitor의 periodTime 프로퍼티의 값은 10인데, 다른 값을 사용하고 싶다면 위 Code에서 roomMonitor 빈 객체처럼

사용 할 프로퍼티 값을 입력해 주면 된다.

프로퍼티 뿐만 아니라 Class도 새롭게 지정할 수 있다. class 속성을 이용하여 Class를 새롭게 지정하면 부모 빈에서

설정한 Class가 아닌 class 속성에 명시한 Class를 사용하여 빈을 생성한다.

<bean id="commonMonitor" class="madvirus.spring.ch02.SystemMonitor" abstract="true">

<property name="periodTime" value="10"/>

<property name="sender" ref="smsSender"/>

<bean>

<bean id="extMonitor" parent="commonMonitor" class="ExtendedSystemMonitor">

<property name="defaultResolution" value="high"/>

</bean>

위 설정에서 extMonitor 빈은 commonMonitor 빈의 프로퍼티 정보를 사용하게 되므로, extMonitor 빈의

설정 정보는 아래와 동일하게 된다.

<bean id="extMonitor" class="ExtendedSystemMonitor">

<property name="periodTime" value="high"/>

<property name="sender" ref="smsSender"/>

<property name="defaultResolution" value="high"/>

</bean>

------------------------------------------------------------------------------------------------------------------------------------------------------------------

3. 빈 객체 범위

: 아래와 같이 SPRING 설정 파일을 작성했다고 하자.

<bean name="monitor" class="madvirus.spring.ch02.SystemMonitor"/>

이때, monitor 빈 객체는 getBean() 메서드를 이용하여 Java Code에서 사용할 수 있다.

SystemMonitor monitor1 = (SystemMonitor)applicationContext.getBean("monitor");

SystemMonitor monitor2 = (SystemMonitor)applicationContext.getBean("monitor");

(monitor1 == monitor2); //true!!!

SPRING은 기본적으로 하나의 빈 설정에 대해 한 개의 객체만을 생성하기 때문에, 위 Code에서 monitor1과

monitor2는 동일한 빈 객체를 참조하게 된다. 이는 SPRING Container 내에서 빈 객체는 싱글톤이라는 것을 의미한다.

여기서 싱글톤은 Class 차원에서의 싱글톤을 의미하는 것이 아니다. 예를 들어, 아래 설정에서 이름이 "monitor"

인 빈 객체와 이름이 "monitorBackup" 빈 객체가 Container에 각각 한 개만 존재한다는 것을 의미한다.

<bean name="monitor" class="madvirus.spring.ch02.SystemMonitor"/>

<bean name="monitorBackup" class="madvirus.spring.ch02.SystemMonitor"/>

경우에 따라서, 빈 객체를 매번 생성해야 하는 경우가 있을 수 있는데, SPRING은 빈 객체에 범위를 지정해서

객체 생성 여부를 지정할 수 있다. 본 절에서는 빈 객체의 범위에 따라 빈 객체 생성이 어떻게 달라지는 지

살펴보도록 하자.

3.1 빈 범위 설정

: 앞서 말했듯이 SPRING은 기본적으로 Container에 한 개의 빈 객체를 생성한다. 하지만, 경우에 따라서 getBean() 메서드를

호출할 때 마다 빈 객체를 매번 생성하고 싶은 경우가 있을 것이다. 이런 경우를 위해 SPRING은 빈의 범위를 설정할 수 있는

방법을 제공하고 있다. 설정 파일에서 <bean> 태그의 scope 속성을 이용해서 빈위 범위를 설정할 수 있다.

<bean id="memory" class="madvirus.spring.ch02.computer.Memory" scope="singleton">

...

</bean>

scope 속성에 올 수 있는 값은 다음과 같다.

-singleton :SPRING Container에 한 개의 빈 객체만 존재한다.(기본값)

-prototype : 빈을 사용할 때 마다 객체를 생성한다.

-request : HTTP 요청 마다 빈 객체를 생성한다. WebApplicationContext에서만 적용 가능하다.

-session : HTTP 세션 마다 빈 객체를 생성한다. WebApplicationContext에서만 적용 가능하다.

-global-session : 글로벌 HTTP 세션에 대한 빈 객체를 생성한다. 포틀릿을 지원하는 컨텍스트에 대해서만 적용 가능하다.

아래 Code는 scope 속성의 값을 "prototype"으로 지정한 설정 예를 보여 주고 있다.

<bean id="memory" class="madvirus.spring.ch02.computer.Memory" scope="prototype">

...

</bean>

이 경우, BeanFactory나 ApplicationContext의 getBean() 메서드를 이용하여 "memory" 빈을 구하면, SPRING Container는

getBean() 메서드가 호출될 때 마다 매번 새로운 Memory 객체를 생성해서 리턴 한다.

Memory memory1 = (Memory)context.getBean("memory");

Memory memory2 = (Memory)context.getBean("memory");

위 Code에서, "memory" 빈의 scope 속성이 "prototype"인 경우, context.getBean("memory") Code는 매번 새로운 Memory

객체를 생성하게 되므로 memory1, memory2는 서로 다른 객체를 참조하게 된다.

3.2 서로 다른 범위 빈에 대한 의존 처리

: 만약 singleton 범위의 빈 객체가 prototype 범위의 빈 객체를 참조한다면 어떻게 될까? 예를 들어, Worker Class는

상태를 갖고 있어서 매번 새롭게 객체를 생성해야 쓰레드에 안전하게 동작하기 때문에, 빈의 범위를 prototype으로

지정했다고 가정해보자.

<bean id="workerBean" class="madvirus.spring.chap02.Worker" scope="prototype"/>

<bean id="executor" class="madvirus.spring.ch02.Executor">

<property name="worker" ref="workerBean" />

</bean>

그런데, 문제는 singleton 범위의 executor 빈이 prototype 범위인 workerBean 빈을 참조하면서 발생한다. 예를 들어,

Executor Class의 Code가 다음과 같다고 하자.

public class Executor{

private Worker worker;

public void setWorker(Worker worker){

this.worker=worker;

}

public void executer(WorkUtil work){

worker.work(work);//매번 새로운 Worker가 적용되지 않음.

}

}

이 경우, Executor Class는 전달받은 worker 프로퍼티가 prototype 범위의 빈 객체임에도 불구하고, 매번 새로운

Worker 객체를 사용하는 것이 아니라 한 개의 Worker 객체만 사용하게 된다.

이렇게 자신보다 생명주기가 더 긴 객체의 의존 객체로 설정되는 경우에는, 예를 들어, prototype 범위의 빈 객체가

singleton 범위의 빈 객체로부터 참조되는 경우, <aop:scoped-proxy> 태그를 이용하면 올바르게 범위를 적용시킬 수 있다.

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

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

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/aop

http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<bean id="workerBean" class="madvirus.spring.chap02.Worker" scope="prototype">

<aop:scoped-proxy />

</bean>

<bean id="executor" class="madvirus.spring.chap02.Executor">

<property name="worker" ref="workerBean" />

</bean>

</beans>

<aop:scoped-proxy /> 태그를 사용하려면 aop 네임스페이스 및 스키마 위치를 지정해주어야 한다.

<aop:scoped-proxy /> 태그를 <bean> 태그에 설정하면, SPRING은 범위에 알맞은 빈 객체에 접근할 수 있도록

해 주는 프록시 객체를 생성한다. 따라서, 아래 Code에서 setWorker() 메서드는 Worker Class의 프록시 객체를 전달받게 된다.

public class Executor{

private Worker worker;

public void setWorker(Worker worker){

this.worker=worker; //<aop:scoped-proxy />를 통해 생성된 프록시 객체.

}

public void executer(WorkUtil work){

worker.work(work);//내부적으로 매번 새로운 Worker 객체가 생성됨.

}

}

<aop:scoped-proxy /> 태그를 통해서 생성된 프록시 객체를 사용할 경우, 해당 프록시 객체에 접근할 때 마다 범위에

알맞은 빈 객체를 내부적으로 생성한다. 예를 들어, 위 Code에서 worker 프로퍼티의 값으로 prototype 범위의 빈에 대한

프록시 객체를 설정했다면, worker 객체에 접근할 때 마다(즉, 프록시 객체에 접근할 때 마다) 매번 프록시 객체는 SPRING

Container로부터 새로운 객체를 가져와 메서드를 실행한다.

<aop:scoped-proxy /> 태그는 기본적으로 CGLIB Library를 이용해서 Class에 대한 프록시 객체를 생성한다.

만약, Class가 아니라 Class가 구현하고 있는 Interface에 대한 프록시 객체를 생성하고 싶다면, 다음과 같이

proxy-target-class 속성의 값을 false로 지정해 주면 된다.

<bean id="workerBean" class="madvirus.spring.chap02.Worker" scope="prototype">

<aop:scoped-proxy proxy-target-class="false"/>

</bean>

<bean id="executor" class="madvirus.spring.chap02.Executor">

<property name="worker" ref="workerBean" />

</bean>

proxy-target-class 속성의 값을 false로 지정하면, CGLIB가 아닌 JDK의 다이나믹 프록시를 이용해서 프록시 객체를 생성하게 된다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

4. 라이프 사이클

: SPRING은 객체를 담는 Container다. 따라서, SPRING은 빈 객체를 생성하는 책임을 지게 된다. SPRING Container는 빈 객체의 생성뿐만

아니라 초기화 및 소멸을 처리하게 된다. 즉, SPRING Container에 저장되는 빈 객체는 최소한 생성, 초기화, 소멸의 라이프 사이클을

갖게 된다.

SPRING은 빈 객체의 생성, 초기화, 소멸뿐만 아니라 추가적인 단계를 제공하고 있으며, 이를 통해 라이프 사이클에 따름 빈 객체의

상태를 절묘하게 제어할 수 있다. 본 절에서는 SPRING Container가 빈 객체의 라이프 사이클을 어떻게 리하는 지 살펴보도록 하자.

4.1 빈 객체의 라이프 사이클

: SPRING Container에 생성되는 빈 객체의 라이프 사이클은 빈 Class가 구현한 Interface에 따라 달라지는데, BeanFactory를

사용할 경우 빈 객체의 완전한 라이프 사이클은 다음과 같다.

빈 객체 생성 ==> BeanNameAware.setBeanName()

==> BeanFactoryAware.setNameFactory()

==>\* BeanPostProcessor의 초기화 전처리

==> 커스텀 init-method

==> InitializingBean.afterPropertiesSet()

==> \*BeanPostProcessor의 초기화 후 처리

==> 빈 객체 사용

==> DisposableBean.destroy()

==> 커스텀 destroy-method

위에서 '\*'가 표시된 단계는 빈 Class가 아닌 특수 빈을 통해 처리되는 과정이며, 나머지 단계가 실제 빈 Class를 통해 처리되는 과정이다.

위에서 [Interface.메서드이름]으로 표시된 단계는 빈 Class가 지정한 Interface를 구현했을 경우 SPRING Container가 지정한 메서드를

호출한다는 것을 의미한다. 예를 들어, 'BeanNameAware.setBeanName()'는 빈 Class가 BeanNameAware Interface를 구현했을 경우

setBeanName() 메서드가 호출된다는 것을 의미한다.

'커스텀 init-method'는 설정 파일에서 <bean> 태그의 init-method 속성에서 지정한 메서드가 호출된다는 것을 의미하며,

'커스텀 destroy-method'는 설정 파일에서 <bean> 태그의 destroy-method 속성에 지정한 메서드가 호출된다는 것을 의미한다.

SPRING은 빈 객체를 생성하고, <property> 태그에 명시된 프로퍼티의 값을 설정한 뒤 빈 Class가 구현한 Interface에 따라 위 에서

명시한 순서대로 메서드를 호출한다. 생성이 완료되면 BeanFactory.getBean() 메서드를 이용하여 빈 객체를 사용할 수 있게 된다.

XmlBeanFactory Class는 SPRING Container에서 빈 객체를 제거할 수 있는 destroySingleton(beanName), destroyBean(beanName,beanInstance)

등의 메서드를 제공하고 있는데, 이들 메서드를 호출하면 '빈 사용'이 후의 단계를 수행하게 된다.

ApplicationContext를 이용할 경우 BeanFactory의 라이프 사이클과 비교하여 몇 가지 단계가 추가된다. 추가된 과정은 아래에서 '\*'로

표시한 부분과 같다.

... ==> BeanFactoryAware.setBeanFactory() ==> ResourceLoaderAware.setResourceLoader() ==>

ApplicationEventPublisherAware.setApplicationEventPublisher() ==> MessageSourceAware.setMessageSource() ==>

ApplicationContextAware.setApplicationContext() ==> @PostConstruct 메서드 실행 ==>

\*BeanPostProcessor의 초기화 전처리 ===> ... ===> 빈 객체 사용 ===>

DisposableBean.destroy() ==> 커스텀 destroy-method

\*\*\*ApplicationContext를 사용하는 경우 추가되는 라이프 사이클 단계.

ApplicationContext는 Resource 및 Message 처리, 이벤트 처리 등 추가적인 기능을 제공하는데, 위의 단계에는 이런 추가 기능을 빈이

사용할 수 있도록 하기 위한 단계가 추가되었다.

SPRING Framework가 제공하는 ApplicationContext의 수현 Class들은 AbstractApplicationContext Class를 상속받고 있는데,

AbstractApplicationContext Class는 destroy() 메서드를 구현하고 있다. destroy() 메서드를 호출하면 SPRING Container에

등록된 빈 객체를 제거하는 과정을 거치게 된다.

"BeanFactory/ApplicationContext에서의 라이프 사이클 단계"를 보면 라이프 사이클과 관련된 Interface가 다수 존재하는 것을

알 수 있는데, 이들 Interface가 모두 사용되는 것은 아니다. 실제로 이들 Interface를 직접적으로 구현하는 경우는 드물다.

하지만, 경우에 따라 객체를 초기화하고 검증하는 등의 과정에서 SPRING이 제공하는 라이프 사이클을 이용하는 것이 편리할 때도

있으므로, 본 절에서는 각 Interface가 어떤 용도로 사용되는 지 살펴보도록 하겠다.

4.2. BeanFactory와 ApplicationContext의 라이프사이클 제거 단계 처리

: BeanFactory Interface의 구현체인 XmlBeanFactory는 빈 객체를 Container에서 제거할 때 DisposableBean.destroy()와 커스텀

destroy 메서드를 실행한다. 빈 객체를 제거할 때 사용되는 메서드를 아래와 같다.

-XmlBeanFactory.removeBeanDefinition(String beanName)

: 이름이 beanName인 빈 객체를 Container에서 제거한다. 제거할 때 제거 단계의 라이프 사이클 메서드를 실행한다.

아래 Code는 사용 예를 보여주고 있다.

XmlBeanFactory beanFactory = new XmlBeanFactory(resource);

String beanName ="articleService";

ArticleService bean = beanFactory.getBean(beanName, ArticleService.class);

...

beanFactory.removeBeanDefinition(beanName);

ApplicationContext Interface를 구현한 Class들은 AbstractApplicationContext Class를 상속받는데, AbstractApplicationContext Class는

XmlBeanFactory Class와 달리 개별 빈을 제거하는 기능을 제공하지 않는다. 대신, Container를 종료시켜주는 close() 메서드를 제공하고

있으며, close() 메서드를 실행하면 Container에 등록된 모든 빈 객체의 제거 처리 메서드가 실행된다.

String[] configLocations = new String[]{"applicationContext.xml"};

AbstractApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

context.close(); //Container가 종료되면서 모든 빈의 종료 단계 실행.

registerShutdownHook() 메서드를 사용하면 JVM이 종료될 떼 ApplicationContext를 종료하는 작업을 수행한다.

아래 Code는 사용 예이다.

String[] configLocations = new String[]{"applicationContext.xml"};

AbstractApplicationContext context = new ClassPathXmlApplicationContext(configLocations);

context.registerShutdownHook();

4.3 BeanNameAware Interface

: SPRING 설정 파일을 보면 아래와 같이 빈 객체를 구분하기 위해 식별값을 지정하고 있다.

<bean id=beanName" class="...">

...

</bean>

id 속성이나 name 속성을 통해 지정한 이름은 SPRING Container가 내부적으로 빈 객체를 식별하는 데 사용되며 빈 객체가

이름을 알고 있지는 않다. 하지만, 경우에 따라서 빈 객체가 자신의 이름을 알아야 할 때가 있다. 예를 들어, 빈 객체의 메서드가

호출될 때 로그를 남기는 경우를 생각해 보자. 동일한 타입의 빈 객체가 다수 존재할 경우, 어떤 빈 객체에서 남긴 로그인지를 확인하려면

빈 객체의 이름을 함께 로그로 남기는 것이 좋을 것이다.

org.springframework.beans.factory.BeanNameAware Interface는 빈 객체가 자신의 이름을 알아야 하는 경우에 사용될 수 있다.

BeanNameAware Interface는 다음과 같이 정의되어 있다.

public interface BeanNameAware{

void setBeanName(String beanName);

}

SPRING Container는 빈 Class가 BeanNameAware Interface를 구현하고 있을 경우, setBeanName() 메서드를 호출하여 빈 객체의

이름을 전달한다. 따라서, 자기 자신의 이름을 필요로 하는 빈 Class는 BeanNameAware Interface를 구현하고 setBeanName()

메서드를 통해 전달받은 이름을 멤버 필드에 저장한 뒤 필요한 곳에서 사용하면 된다. 아래 Code는 BeanNameAware Interface의

구현 예를 보여 주고 있다.

public class JobExecutor implements BeanNameAware, Executor{

...

private String beanName;

public void setBeanName(String beanName){

this.beanName=beanName;

}

public void run(){

...

log.debug(beanName + ":" + ...);

}

}

4.4 BeanFactoryAware Interface와 ApplicationContextAware Interface

: SPRING Container는 생성자나 프로퍼티 설정 메서드를 이용하여 의존하는 객체를 전달해 주기 때문에 빈 객체가 직접적으로 의존하는

객체를 검색하는 경우는 흔치 않지만, 종종 빈 객체가 필요한 객체를 SPRING Container로부터 직접 검색해야 하는 경우가 있다.

또는, 빈 객체에서 ApplicationContext가 제공하는 Resource 및 Message 관련 기능을 사용하고 싶은 경우가 있다.

이렇게 빈 객체가 SPRING Container(BeanFactory나 ApplicationContext)를 직접 사용해야 할 필요가 있다면, 아래의 두 Interface를

구현해서 빈 객체를 관리하고 있는 BeanFactory나 ApplicationContext를 전달받을 수 있다.

-org.springframework.beans.factory.BeanFactoryAware : BeanFactory를 빈에 전달할 때 사용된다.

-org.springframework.context.ApplicationContextAware : ApplicationContext를 빈에 전달할 때 사용다.

두 Interface는 각각 다음과 같이 메서드를 정의하고 있다.

public interface BeanFactoryAware{

public void setBeanFactory(BeanFactory factory) throws BeansException;

}

public interface ApplicationContextAware{

public void setApplicationContext(ApplicationContext context) throws BeansException;

}

빈 객체는 BeanFactory나 ApplicationContext를 이용하여 Container로부터 필요한 정보를 구할 수 있게 된다. 아래 Code는 ApplicationContextAware

Interface를 구현한 예를 보여 주고 있다.

public class FilterAutoCreator implements ApplicationContextAware{

private ApplicationContext context;

public void setApplicationContext(ApplicationContext context){

this.context=context;

}

public void init(){

Map filterBeanMap = context.getBeansOfType(Filter.class);

Iterator filterIter = filterBeansMap.values().iterator();

}

}

4.5 InitializingBean Interface

: 빈 객체의 라이프 사이클과 관련된 Interface 중에서 가장 많이 사용되는 것 중의 하나가 org.springframework.beans.factory.InitializingBean

Interface이다. InitializingBean Interface는 객체를 생성하 프로퍼티를 초기화 하고, Container 관련 설정을 료한 뒤에 호출되는 메서드를

정의하고 있다. InitializingBean Interface가 정의한 메서드는 다음과 같다.

public interface InitializingBean{

public void afterPropertiesSet() throws Exception;

}

afterPropertiesSet() 메서드는 주로 빈 객체의 프로퍼티가 모두 올바르게 설정되었는지의 여부를 검사하는 용도로 사용된다.

아래는 실제로 SPRING Framework가 제공하고 있는 Class에서 InitializingBean Interface를 사용하여 프로퍼티 값을 검증하는

Code를 구현한 예이다.

public abstract class AbstractUrlBasedView extends AbstractView implements InitializingBean{

...

public void afterPropertiesSet() throws Exception{

if(getUrl() == null){

throw new IllegalArgumentException("Property 'url' is required");

}

}

...

}

4.6 DisposableBean Interface

: Container에서 빈 객체를 제거할 때, 빈 객체는 사용하던 Resource을 반납해야 한다. 소켓 통신을 하고 있었다면 소켓을 종료해야 하고,

쓰레드를 통해 백그라운드로 작업을 수행하고 있었다면 쓰레드를 알맞게 종료시켜야 한다.

빈 Class가 org.springframework.beans.factory.DisposableBean Interface를 구현한 경우 SPRING은 빈 객체를 Container에서

제거하기 전에 DisposableBean Interface에 정의된 메소드를 호출하여 빈 객체가 Resource을 반납할 수 있도록 하고 있다.

DisposableBean Interface는 아래와 같이 정의되어 있다.

public interface DisposableBean{

public void destroy() throws Exception;

}

4.7 커스텀 초기화 및 소멸 메서드

: 아래 Monitor Class는 모니터링을 위한 Class이고, start() 메서드를 호출하면 모니터링 작업이 시작되고 stop() 메서드를 호출하면

모니터링이 종료된다고 하자.

public class Monitor{

public void start(){

...

}

public void stop(){

...

}

}

Monitor Class를 사용하려면 다음과 같이 객체를 생성한 뒤 start() 메서드를 호출하고 사용이 끝나면 stop() 메서드를 호출할 것이다.

Monitor monitor = new Monitor();

monitor.start();

...

monitor.stop();

Monitor Class를 SPRING Container에 빈 객체로 등록하고 SPRING Container가 시작되고 종료될 때 자동으로 Monitor 빈 객체도

시작되고 종료되어야 한다고 하자. Monitor Class는 SPRING Framework가 제공하는 라이프 사이클 Interface를 구현하고 있지

않기 때문에 별도 처리를 해 주어야 한다.

SPRING은 <bean> 태그의 init-method 속성과 destroy-method 속성을 통해서 빈 객체를 초기화하고 제거할 때 메서드를 지정할 수 있다.

예를 들어, 앞서 Monitor Class를 빈 객체로 등록하고 Container가 시작될 때 start() 메서드가 호출되고 종료될 때 stop() 메서드가

호출되도록 하려면 다음 같이 init-method 속성과 destroy-method 속성 값을 설정하면 된다.

<bean id="monitor" class="madvirus.system.ch02.Monitor" init-method="start" destroy-method="stop">

...

</bean>

------------------------------------------------------------------------------------------------------------------------------------------------------------------

5. 외부 설정 프로퍼티

: PropertyPlaceholderConfigurer Class를 빈으로 등록하면 외부의 프로퍼티 파일에 저장된 정보를 SPRING 설정 파일에서

사용할 수 있다. 예를 들어, 다음과 같은 프로퍼티 파일을 작성했다고 하자.

jdbc.driver=oracle.jdbc.driver.OracleDriver

jdbc.url=jdbc:oracle:thin:@192.168.0.100:1521:ORCL

jdbc.username=ssol

jdbc.password=ssol

위 프로퍼티 파일의 정보를 SPRING 설정 파일에서 사용하고 싶다면, 다음과 같이 PropertyPlaceholderConfigurer Class를

빈으로 등록한 뒤, 프로퍼티 이름을 설정 파일에서 사용하면 된다.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>classpath:config/jdbc.properties</value>

</property>

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driver}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

locations 프로퍼티 값에는 콤마나 공백으로 구분된 프로퍼티 파일 목록이 오며, 프로퍼티 파일에 포함된 프로퍼티의 값은

'${프로퍼티이름}' 형식으로 사용할 수 있다. 즉, 위 Code에서 ${jdbc.driver}는 프로퍼티 파일의 jdbc.driver 프로퍼티의 값으로

대체된다.

한 개 이상의 프로퍼티 파일을 지정하려면 <list> 태그를 이용하여 프로퍼티 목록을 지정해 주면 된다.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:config/jdbc.properties</value>

<value>classpath:config/monitor.properties</value>

</list>

</property>

</bean>

외부 환경에 따라 설정 정보가 변경되는 경우에 프로퍼티 파일을 유용하게 사용할 수 있다. 예를 들어, 로컬에서의 Test와

Test 서버에서의 통합 Test, 그리고 실제 운영 환경에서 사용하는 Database가 다르다고 하자.

이 경우 각 환경에 맞게 SPRING 설정 파일을 변경하는 것 보다는 환경에 맞는 프로퍼티 설정 파일을 작성하고, 환경에 따라

알맞은 프로퍼티 파일을 사용하도록 배치 스크립트를 작성하는 것이 유지 및 보수에 더 편리하다.

5.1 <context:property-placeholder> 태그를 사용한 외부 설정 프로퍼티 사용

: <context:property-placeholder> 태그를 사용하여 외부 프로퍼티 파일을 로딩하도록 설정할 수도 있다.

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

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

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/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:property-placeholder location="classpath:config/jdbc.properties" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driver}"/>

..

</bean>

..

</beans>

<context:property-placeholder> ?태그를 사용하려면 위 Code와 같이 context 접두어에 대한 네임스페이스를

http://www.springframework.org/schema/context로 지정하고 네임스페이스와 관련된 XML 스키마 경로를

spring-context-3.0.xsd로 지정해 주어야 한다.

여러 개의 프로퍼티 파일을 사용하려면 각 프로퍼티 파일을 콤마로 구분하여 지정하면 된다.

<context:property-placeholder location="classpath:config/jdbc.properties, classpath:config/monitor.properties"/>

5.2 PropertyPlacaholderConfigurer 사용시 주의사항

: PropertyPlacaholderConfigurer를 사용할 때 주의할 점은 두 개 이상의 PropertyPlacaholderConfigurer 빈을 설정하면

안 된다는 점이다. 이 경우, 첫 번째 PropertyPlacaholderConfigurer의 설정이 적용되며, 두 번째 설정 내용은 적용되지

않는다. 예를 들어, 두 개의 프로퍼티 파일이 다음과 같다고 하자.

#jdbc1.properties

jdbc.driver=oracle.jdbc.driver.OracleDriver

jdbc.url=jdbc:oracle:thin:@192.168.0.100:1521:ORCL

#jdbc2.properties

jdbc.username=springbook

jdbc.password=springbook

위 Code에서 jdbc1.properties 파일은 jdbc.driver, jdbc.url 프로퍼티를 설정하고 있고, jdbc2.properties 파일은 jdbc.username

프로퍼티와 jdbc.passoword 프로퍼티를 설정하고 있다. 이 경우 아래와 같이 서로 다른 PropertyPlacaholderConfigurer 빈 객체에서

각 프로퍼티 파일을 사용하도록 설정했다고 해 보자.

<bean class="org.springframework.beans.factory.config.PropertyPlacaholderConfigurer">

<property name="locations">

<value>classpath:config/jdbc1.properties</value>

</property>

</bean>

<bean class="org.springframework.beans.factory.config.PropertyPlacaholderConfigurer">

<property name="locations">

<value>classpath:config/jdbc2.properties</value>

</property>

</bean>

<bean id="dataSourceFactory" class="madvirus.spring.chap02.DataSourceFactory">

<property name="jdbcDriver" value="${jdbc.driver}"/>

<property name="jdbcUrl" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

각 PropertyPlacaholderConfigurer 빈은 서로 프로퍼티 정보를 공유하지 않기 때문에, 위와 같이 설정한 경우, 첫 번째

PropertyPlacaholderConfigurer 빈이 ${jdbc.username}을 분석하는 과정에서 (jdbc1.properties에는 jdbc.username

프로퍼티가 없으므로) 예외가 발생하게 된다.

SPRING을 이용하다 보면 두 개 이상의 설정 파일을 사용하는 경우가 빈번한데, 이 때 서로 다른 파일에

PropertyPlacaholderConfigurer 빈이 설정되어 있을 경우 프로퍼티 값이 올바르게 처리되지 않아서 예외가 발생할 수 있다.

따라서, PropertyPlacaholderConfigurer를 사용할 때에는 두 번 설정되지 않도록 주의해야 한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

6. Container 간 계층

: BeanFactory나 ApplicationContext는 Container 간 자식-부모 계층 구조를 가질 수 있다. Container간 계층 구조를 구성하면,

자식 Container는 부모 Container에 정의된 빈 객체에 접근할 수 있다. 반면에 부모 Container에서는 자식 Container에 정의된

빈 객체에 접근할 수 없다.

ApplicationContext나 BeanFactory를 생성할 때 부모 Container를 지정하려면 아래 Code와 같이 부모 Container를 추가로 지정해 주면

된다.

String[] parentConfigLocations = new String[]{"parent.xml"};

AbstractApplicationContext parentContext = new ClassPathXmlApplicationContext(parentConfigLocations);

String[] childConfigLocations = new String[]{"child.xml"};

AbstractApplicationContext childContext = new ClassPathXmlApplicationContext(childConfigLocations, parentContext);

위 Code에서 childContext Container를 생성할 때 부모 Container로 parentContext를 지정하였다. parentContext를 생성할 때

사용한 설정 파일인 parent.xml과 childContext를 생성할 때 사용한 설정 파일인 child.xml은 아래와 같다고 해 보자.

<!--parent.xml-->

<bean id="smsSender" class="madvirus.spring.ch02.SmsSender"/>

<!--child.xml-->

<bean id="monitor" class="madvirus.spring.ch02.SystemMonitor" p:periodTime="10" p:sender-ref="smsSender"/>

위 Code를 보면 자식 Container에 정의된 monitor 빈 객체는 부모 Container에 정의된 smsSender 빈을 참조하고 있다.

위 설정에서 보듯이 자식 Container는 부모 Container에 정의된 빈 객체를 참조할 수 있다. 하지만, 반대로 부모 Container에 정의된

빈 객체는(Container 생성 시점에서 자식 Container 존재 여부를 알 수 없기 때문에) 자식 Container가 제공하는 빈 객체를

참조할 수 없다.