스프링에서 빈이란?

: 스프링에 제어권을 가지고 직접 만들고 관계를 부여하는 오브젝트. 스프링 컨테이너가 생성, 관계설정, 사용 등을 제어해주는 IoC가 적용된 오브젝트

빈 팩토리이란?

: 빈의 생성과 관계설정과 같은 제어를 담당하는 IoC오브젝트

(이를 더 확장한 개념이 애플리케이션 컨텍스트이다)

@Configuration이란?

: 스프링이 빈 팩토리를 위한 오브젝트 설정을 담당하는 클래스라고 인식할 수 있도록 등록해주는 어노테이션

@Bean이란?

: 오브젝트를 만들어주는 메소드에 적용시키는 어노테이션

Ex) 클래스의 생성자는 해당 클래스의 오브젝트를 생성하고 초기화하여 돌려주는 역할을 함으로,

해당 어노테이션이 붙기에 적합하다.

프레임워크와 라이브러리의 차이

* 라이브러리

: 라이브러리를 사용하는 애플리케이션 코드는 애플리케이션 흐름을 직접 제어한다. 단지 동작하는 중에 필요한 기능이 있을 때 능동적으로 라이브러리를 사용할 뿐이다.

* 프레임워크

: 거꾸로 애플리케이션 코드가 프레임워크에 의해 사용된다. 보통 프레임워크 위에 개발한 클래스를 등록해두고, 프레임워크가 흐름을 주도하는 중에 개발자가 만든 애플리케이션 코드를 사용하도록 만드는 방식이다.

어노테이션 기법과 new 할당기법의 차이(결과는 동일해 보임)

: new 할당 기법으로 사용 시, 할당할 때마다 ‘동등’한 객체가 생성되지만,

어노테이션 기법으로 스프링 할당 시, 싱글턴 기법이 적용되 있기 때문에 ‘동일’한 객체가 가져와진다.(참조되어 진다, 객체는 하나만 생성)

어플리케이션 컨텍스트

: 오브젝트 팩토리와 비슷한 방식으로 동작하는 IoC 컨테이너이다. 그러면서 동시에 싱글톤을 저장하고 관리하는 싱글톤 레지스트리이다.

스프링에서 싱글턴을 사용하는 이유?

: 서버 환경에서 유동적인 오브젝트 생성 횟수가 늘어남에 따라 생기는 쓰레기값을 가비지 컬렉터가 감당할 수 없을 정도로 늘어나기 때문에 싱글턴을 활용하여 참조형태로 사용한다.

싱글톤 패턴의 단점

* Private 생성자를 갖고 있기 때문에 상속할 수 없다.

: 오직 싱글톤 클래스 자신만이 자기 오브젝트를 만들도록 제한한다.

* 싱글톤은 테스트하기가 어렵다.

: 초기화 과정에서 생성자 등을 통해 사용할 오브젝트를 만들어 사용할 수 밖에 없다. 이런 경우 테스트용 오브젝트로 대체하기가 힘들다.

* 서버 환경에서는 싱글톤이 하나만 만들어지는 것을 보장하지 못한다.

: 서버에서 클래스 로더를 어떻게 구성하고 있느냐에 따라 싱글톤 클래스임에도 하나 이상의 오브젝트가 만들어 질 수 있다.(여러 개의 JVM에 분산되어 설치되는 경우에도 각 각 독립적으로 오브젝트가 생기기 때문에 싱글톤으로서의 가치가 떨어진다.)

* 싱글톤의 사용은 전역상태를 만들 수 있기에 바람직하지 못하다.

: 싱글톤의 스태틱 메소드를 이용하는 방식은 전역상태로 사용되기 쉽기에 객체지향 프로그래밍에서 권장하지 않는 프로그래밍 모델이다.

싱글톤 레지스트리

: 스프링은 서버환경에서 싱글톤이 만들어져서 서비스 오브젝트 방식으로 사용되는 것을 적극 지지한다. 왜냐하면 스프링 직접 싱글톤 형태의 오브젝트를 만들고 관리하는 기능을 제공하기 때문이다. 이를 싱글턴 레지스트리라고 한다.(싱글턴 관리 컨테이너)

싱글톤 레지스트리 덕분에 싱글턴 방식으로 사용될 애플리케이션 클래스라도 public 생성자를 가질 수 있다.

싱글톤과 오브젝트의 상태

: 기본적으로 싱글톤이 멀티스레드 환경에서 서비스 형태의 오브젝트로 사용되는 경우에는 상태정보를 내부에 갖고 있지 않은 무상태방식으로 만들어져야 한다.

멀티 쓰레드 방식에서의 덮어쓰기 등의 방식은 위험하기에 싱글톤은 기본적으로 인스턴스 필드의 값을 변경하고 유지하는 상태유지(stateful) 방식으로 만들지 않는다.

* 동일하게 읽기전용의 속성을 가진 정보라면 싱글톤에서 인스턴스 변수로 사용해도 좋다. 물론 단순한 읽기전용 값이라면 static final이나 final로 선언하는 편이 나을 것이다.

스프링 빈의 스코프

: 빈의 스코프란 : 빈이 생성되고, 존재하고, 적용되는 범위를 뜻한다.

스프링 빈의 기본 스코프는 싱글톤이다.( 하나의 오브젝트만 만들어져, 강제로 제거하지 않는 한 스프링 컨테이너가 존재하는 동안 계속 유지된다.)

But, 경우에 따라서 싱글톤 외의 스코프를 가질 수 있다. 대표적으로 프로토타입 스코프가 있다.

프로토타입 스코프 : 싱글톤과 달리 컨테이너에 빈을 요청할 때마다 매번 새로운 오브젝트를 만들어준다.

요청(request) 스코프 : 웹을 통해 새로운 HTTP 요청이 생길 대마다 오브젝트를 생성

세션 스코프 : 웹의 세션과 스코프가 유사하다.

의존관계 주입(Dependency Injection)

: spring ioc의 대표적인 동작원리는 주로 의존관계 주입이라고 불린다.

DI는 오브젝트 레퍼런스를 외부로부터 주입받고 이를 통해 여타 오브젝트를 다이내믹하게 의존관계가 만들어지는 것이 핵심이다.

의존관계 설정

* 의존관계란?

: 의존대상(B)가 변하면 그것이 A에 영향을 미친다는 뜻이다.

B의 기능이 추가되거나 변경되거나, 형식이 바뀌거나 하면 그 영향이 A로 전달된다는 것이다.

But, 의존관계는 방향성이 있다. A가 B에 의존하고 있지만, 반대로 B는 A에 의존하지 않는다.

의존관계란 한쪽의 변화가 다른 쪽의 영향을 주는 것이라고 햇으니, 인터페이스를 통해 의존관계를 제한해주면 그만큼 변경에서 자유로워지는 셈이다.

* 런타임 의존관계(오브젝트 의존관계)

: 설계시점의 의존관계가 실체화된 것이라고 볼 수 있다.(런타임 의존관계는 모델링 시점의 의존관계와는 성격이 분명히 다르다)

* 인터페이스를 통해 설계 시점에 느슨한 의존관계를 갖는 경우에는 런타임 시에 사용할 오브젝트가 어떤 클래스로 만든 것인지 미리 알 수 없다.
* 의존 오브젝트란?

: 프로그램이 시작되고 오브젝트가 만들어지고 나서 런타임 시에 의존관계를 맺는 대상, 즉 실제 사용대상인 오브젝트

* 클라이언트

: 구체적인 의존 오브젝트를 사용할 주체

의존관계 주입(DI:Dependency Injection)란?

: 의존 오브젝트들과 클라이언트들을 연결 시켜주는 작업을 말한다.

Ex) DaoFactory는 두 오브젝트 사이의 런타임 의존관계를 설정해주는 DI를 주도하는 존재이며, 동시에 IoC방식으로 오브젝트의 생성, 초기화, 제공 등의 작업을 수행하는 컨테이너다. 따라서 DI 컨테이너라고 불러도 된다.

* 스프링에서 getBean()에서 빈이름으로 검색하는 것을 보고 의존관계 검색이라고 하기도 한다.

스프링에서 의존관계 주입 방식과 의존관계 검색 방식

: 의존관계 주입방식은 의존관계 검색 방식에 비해 깔끔하고 API나 스프링 프레임워크의 메소드가 나오지 않기 때문에 훨씬 단순하고 깔끔하다. 하지만 애플리케이션의 기동시점(main)과 서버에서의 사용자 요청을 받을 때마다, 의 경우는 의존관계 검색방식을 무조건 사용해야 한다.

왜냐하면 해당 상황에서의 DI를 이용한 오브젝트 주입을 받을 수 잇는 방법이 없기 때문이다.

의존관계 주입 방식(DI) ex)

Public UserDao(){

DaoFactory daofactory = new DaoFactory();

This.connectionMaker = DaoFactory.connectionMaker();

}

의존관계 검색 방식(DL:Dependency Lookup)

Public UserDao(){

AnnotationConfigApplicationContext context =

New AnnotationConfigApplicationContxt(DaoFactory.class);

This.connectionMaker = context.getBean(“connectionMaker”, ConnectionMaker.class);

}

* DI와 DL의 특정한 차이점

: DL은 UserDao가 굳이 스프링이 만들고 관리하는 빈일 필요가 없다. ConnectionMaker만 스프링의 빈이기만 하면 된다.(스프링에서 쓰는 것은 connectionMaker(DB)이기 때문)

하지만, DI는 UserDao와 ConnectionMaker 사이에 DI가 적용되려면 UserDao도 반드시 컨테이너가 만드는 빈 오브젝트여야 한다. 컨테이너가 UserDao에 생성과 초기화 권한을 갖고있어야 유동적 오브젝트 주입이 가능해지고, 이 뜻은 IoC방식으로 컨테이너에서 생성되는 오브젝트, 즉 빈이어야 한다.(DL은 @Bean으로 검색하기에 맞닿은 위치에 있을 필요가 없다.)

수정자를 통한 DI주입하는 방식 또한 사용할 수 있다.( 실제로 의존관계를 주입하는 시점과 방법이 달라졌을 뿐, 결과는 동일하다)

Return new CountingConnectionMaker(realConnectionMaker());

ㄴ> UserDao userDao = new UserDao();

userDao.setConnectionMaker(connectionMaker());

return userDao;

xml을 이용한 설정(애노테이션을 사용하면 애노테이션만 xml을 사용하면 xml만 거의 사용한다)

: 자바 클래스 파일 수정을 해야 한다는 단점이 있다. 그리고 대부분 틀에 박힌 구조가 반복되고 DI구성이 바뀔 때마다 자바 코드를 수정하고 클래스를 다시 컴파일하는 것도 귀찮은 작업이다.

이를 위한 방법이 xml을 사용하는 방법이다.

Xml 설정

: 스프링의 애플리케이션 컨텍스트는 xml에 담긴 DI정보를 활용할 수 있다. DI 정보가 담긴 xml 파일은 <beans>를 루트 엘리먼트로 사용한다.

* <beans>안에는 여러 개의 <bean>을 정의할 수 있다.

Xml 설정은 @Configuration과 @Bean이 붙은 자바 클래스로 만든 설정과 내용이 동일하다.

@Configuration = <beans>, @Bean = <bean>

* 빈의 이름 : @Bean 메소드의 이름이 빈의 이름. getBean()에서 사용된다(id)
* 빈의 클래스 : 빈 오브젝트를 어떤 클래스를 이용해서 만들지를 정의한다(class)
* 빈의 의존 오브젝트 : 빈의 생성자나 수정자 메소드를 통해 의존 오브젝트를 넣어준다. 의존 오브젝트도 하나의 빈이므로 이름이 있을 것이고, 그 이름에 해당하는 메소드를 호출해서 의존 오브젝트를 가져온다. 의존 오브젝트는 하나 이상일 수 있다.

빈 설정 파일

* 자바 코드 : @Configuration
* Xml : <beans>

빈의 이름

* 자바 코드 : @Bean methodName()
* Xml : <bean id=”methodNmae”>

빈의 클래스

* 자바 코드 : return new BeanClass();
* Xml : <bean class=”패키지.하위패키지.BeanClass”> // 리턴타입 변수가 아닌 클래스명

스프링 개발자가 수정자 메소드를 선호하는 이유 중에는 xml로 의존관계 정보를 만들 대 편리하다는 점도 있다. 자바빈의 관례에 따라 수정자 메소드는 프로퍼티가 된다. 프로퍼티 이름은 메소드 이름에서 set을 제외한 나머지 부분을 사용한다. Ex) setConnectionMaker() -> connectionMaker라는 프로퍼티를 갖는다.

<property> 태그는 name과 ref라는 두 개의 애트리부트를 갖는다. Name은 프로퍼티의 이름

Ref는 수정자 메소드를 통해 주입해줄 오브젝트 빈의 이름이다.

마지막으로 이 <property> 태그를 <bean> 태그 내에 넣어주면 완성

Ex)

userDao.setConnectionMaker(connectionMaker());

<property name=”connectionMaker” ref=”connectionMaker” />

<bean id=”useDao” class=”패키지.UserDao”>

<property name=”connectionMaker” ref=”connectionMaker”/>

</bean>

Property의 name 애트리부트는 di에 사용할 수정자 메소드의 프로퍼티 이름이며,

Ref는 주입할 오브젝트를 정의한 빈의 ID이다.

빈 이름도 역시 인터페이스 이름을 사용하는 경우가 많다.

* 빈의 이름이 바뀌는 경우 그 이름을 참조하는 다른 빈의 <property> ref 애트리뷰트 값도 변경시켜주어야 한다.

위의 빈의 이름 변경을 순조롭게 하기 위해 자바코드라면 이름변경 리팩토링 기능을 사용하면 간단하지만 xml은 텍스트 치환 기능을 사용해야 하는데 이는 위험하다.

때로는 같은 인터페이스를 구현한 의존 오브젝트를 여러 개 정의해두고 그 중에서원하는 걸 골라서 DI하는 경우도 있다. 이 때는 각 빈의 이름을 독립적으로 만들어두고 ref 애트리뷰트를 이용해 DI 받을 빈을지정해 주면 된다.

Ex)

<beans>

<bean id=”localDBConnectionMaker” class=”…LocalDBConnectionMaker”/>

<bean id=”testDBConnectionMaker” class=”…TestDBConnectionMaker”/>

…

<bean id=”userDao” class=”…UserDao”>

<property name=”connectionMaker” ref=”localDBConnectionMaker” />

</bean>

</beans>

* DTD와 스키마

Xml 문서는 미리 정해진 구조를 따라서 작성됐는지 검사할 수있다. Xml 문서의 구조를 정의하는 방법에는 DTD와 스키마가 잇다. 스프링의 xml에서는 이 두가지 방식을 모두 지원한다.

DTD를 사용할 경우에는 <beans> 엘리먼트 앞에 다음과 같은 DTD 선언을 넣어준다.

<!DOCTYPE beans PUBLIC “-//SPRING//DTD BEAN 2.0//EN”

“http://www.springframework.org/dtd/spring-beans-2.0.dtd>

스프링은 기본 태그인 <beans>, <bean> 외에도 특별한 목적을 위해 별도의 태그를사용할 수 있는 방법을 제공한다. 이 태그들은 각각 별개의 스키마 파일에정의되어 있고 독립적인 네임스페이스를 사용해야 한다. <beans> 태그를 기본네임스페이스로 하는 스키마 선언은 다음과 같다.

<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>”>

특별한 이유가없다면 DTD보다는 스키마를 사용하는 편이 바람직하다.

스프링 IDE의 XML 에디터를 사용하면 스키마 선언을 간단히 만들고 관리할 수 있다.

XML을 이용하는 애플리케이션 컨텍스트

어노테이션을 사용하는 AnnotationConfigApplicationContext와 달리

Xml에서는 GenericXmlApplicationContext를 사용한다.

시작하는 /가 없는 경우에도 항상 루트에서부터 시작하는 클래스 패스이다.

ApplicationContext context – new GenericXmlApplicationContext(“apllicationContext.xml”);

* ClassPathXmlApplicationContext는 클래스패스 뿐 아니라 다양한 소스로부터 설정파일을 읽어올 수 있다. 그리고 경로를 상대적으로 알아서 파악 해줘 긴 경로를 적지 않아도 된다.

Ex) new ClassPathXmlApplicationContext(“daoContext.xml”, UserDao.class);

ㄴ> 두 번째 매개변수의 클래스 파일이 있는 곳을 기준으로 상대적으로 경로를 계산한다.

-------------------Spring xml bean property 정리------------------------------------

id : spring container에서 식별할 수 있는 이름(빈의 이름)

class : 해당 bean의 full path

name : id 대신에 사용가능한 속성 ex)<bean name=”testBean”>

but, 헷갈리지 않게 bean은 id, property는 name을 쓰자

----- bean 내부 태그 -----

<bean id=”…” class=”…”>

<factory-method=”getInstance” />//해당 클래스의 객체를 반환해주는 메소드(singleton)

</bean>

<bean>

<constructor-arg ref=”testDao” />

</bean>

// 생성자를 통한 bean 넘기기

<bean>

<constructor-arg value=”10”/>

</bean>

// 생성자를 통한 특정 값 넘기기

<bean>

<property name=”setter이름” ref=”testDao”/>

</bean>

// setter를 통한 특정값넘기기ex) if setDaoMK이면 set을 빼고 앞 대문자를 소문자로 변경 daoMK

<bean>

<property name=”mappings”>

<props>

<prop key=”/login/login.mv”>loginController</prop>

</props>

</property>

</bean>

// props : java.util.properties 클래스로 key, value를 String type으로만 갖는다.

즉 위의 예는 SimpleUrlHandlerMapping 클래스의 setMapping 메소드를 통해 properties 객체를 생성하고 해당 properties 객체는 key=”/login/login.mv”, value=loginController를 갖고 있게 된다.

요약 : key-value 값을 가지는 String 값만 가질 수 있는 자료구조의 태그

Bean의 객체 범위

TestFao testDao1 = (TestDao)applicationContext.getBean(“testDao”);

TestFao testDao2 = (TestDao)applicationContext.getBean(“testDao”);

위의 두 객체는 동일한 객체지만Bean scope 명시할 경우, 서로 다른 객체로 생성이 가능하다

Scope=””

* singleton : 기본값, 컨테이너 하나의 빈 객체 생성
* prototype : 빈은 사용할 때마다 새로운 객체 생성
* request : http 요청 시 마다 새로운 객체 생성(WebApplicationContext에서 사용)
* session : 세션마다 새로운 객체 생성(WebApplicationContext에서 사용)

DataSource 인터페이스로 변환

: 자바에서는 DB커넥션을 가져오는 오브젝트의 기능을 추상화해서 비슷한 용도로 사용할 수 있게 만들어진 DataSource라는 인터페이스가 이미 존재한다. 따라서 실전에서 ConnectionMaker를 사용할 일은 거의 없을 것이다.

단, DataSource는 getConnection()이라는 DB커넥션을 가져오는 기능 외에도 여러 개의 메소드를 갖고 있어서 인터페이스를 직접 구현하기는 부담스럽다.

일반적으로 DataSource를 구현해서 DB커넥션을 제공하는 클래스로 만들 일은 거의 없다. 이미 다양한 방법으로 DB 연결과 풀링기능을 많이 갖춘 많은 DataSource 구현 클래스가 존재하고, 이를 가져다 사용하면 충분하기 때문이다.

Package javax.sql //javax.sql에 DataSource 클래스가 들어가 있다.

Public interface DataSource extends CommonDataSource, Wrapper{

Connection getConnection() throws SQLException;

} //DataSource의 실제 예상 인터페이스 구조(실제로는 구현하지 않고

// spring-jdbc 라이브러리를 가져와서 사용한다.