# Spring Framework Overview

(Version 5.0.20.RELEASE)

Spring ایجاد برنامه های سازمانی جاوا را آسان می کند. با پشتیبانی از Groovy و Kotlin به‌عنوان زبان‌های جایگزین در JVM، و با انعطاف‌پذیری برای ایجاد انواع معماری‌ها بسته به نیازهای یک برنامه، هر آنچه را که برای پذیرش زبان جاوا در یک محیط سازمانی نیاز دارید، فراهم می‌کند. از Spring Framework 5.0، Spring به JDK 8+ (Java SE 8+) نیاز دارد و از قبل از JDK 9 پشتیبانی خارج از جعبه را فراهم می کند.

Spring طیف گسترده ای از سناریوهای کاربردی را پشتیبانی می کند. در یک شرکت بزرگ، برنامه‌ها اغلب برای مدت طولانی وجود دارند و باید روی یک JDK و سرور برنامه‌ای اجرا شوند که چرخه ارتقا آن خارج از کنترل توسعه‌دهنده است. برخی دیگر ممکن است به صورت یک jar با سرور تعبیه شده اجرا شوند، احتمالاً در یک محیط ابری. با این حال، برخی دیگر ممکن است برنامه‌های مستقل باشند (مانند بارهای کاری دسته‌ای یا یکپارچه) که نیازی به سرور ندارند.

بهار منبع باز است. دارای یک جامعه بزرگ و فعال است که بازخورد مستمری را بر اساس طیف متنوعی از موارد استفاده در دنیای واقعی ارائه می دهد. این به بهار کمک کرده است تا در مدت زمان طولانی با موفقیت تکامل یابد.

## 1. What We Mean by "Spring"

اصطلاح "بهار" به معنای چیزهای متفاوت در زمینه های مختلف است. می توان از آن برای اشاره به خود پروژه Spring Framework استفاده کرد، جایی که همه چیز از آنجا شروع شد. با گذشت زمان، پروژه‌های بهار دیگری در بالای چارچوب Spring ساخته شده‌اند. اغلب، وقتی مردم می گویند "بهار"، منظور آنها کل خانواده پروژه ها است. این مستندات مرجع بر اساس تمرکز دارد: خود چارچوب بهار.

Spring Framework به ماژول ها تقسیم می شود. برنامه ها می توانند ماژول های مورد نیاز خود را انتخاب کنند. در قلب ماژول های core container، از جمله مدل پیکربندی و مکانیزم تزریق وابستگی قرار دارند. فراتر از آن، Spring Framework پشتیبانی اساسی برای معماری‌های کاربردی مختلف، از جمله پیام‌رسانی، داده‌های تراکنش و پایداری، و وب فراهم می‌کند. همچنین شامل چارچوب وب Spring MVC مبتنی بر Servlet و به طور موازی، چارچوب وب واکنشی Spring WebFlux است.

نکته ای در مورد ماژول ها: جارهای فریمورک Spring اجازه استقرار در مسیر ماژول JDK 9 ("Jigsaw") را می دهند. برای استفاده در برنامه‌های دارای قابلیت Jigsaw، شیشه‌های Spring Framework 5 دارای ورودی‌های مانیفست «Automatic-Module-Name» هستند که نام‌های ماژول در سطح زبان پایدار («spring.core»، «spring.context» و غیره) را مستقل از آرتیفکت jar تعریف می‌کنند. نام‌ها (شیشه‌ها از الگوی نام‌گذاری یکسانی با "-" به جای "." پیروی می‌کنند، به عنوان مثال، "spring-core" و "spring-context"). البته، جارهای فریمورک Spring به خوبی در مسیر کلاس در JDK 8 و 9 کار می کنند.

## 2. History of Spring and the Spring Framework

بهار در سال 2003 به عنوان پاسخی به پیچیدگی مشخصات اولیه J2EE به وجود آمد. در حالی که برخی جاوا EE و Spring را با هم رقابت می کنند، Spring در واقع مکمل جاوا EE است. مدل برنامه نویسی Spring مشخصات پلت فرم Java EE را در بر نمی گیرد. بلکه با مشخصات فردی با دقت انتخاب شده از چتر EE ادغام می شود

Servlet API (JSR 340)

WebSocket API (JSR 356)

Concurrency Utilities (JSR 236)

JSON Binding API (JSR 367)

Bean Validation (JSR 303)

JPA (JSR 338)

JMS (JSR 914)

as well as JTA/JCA setups for transaction coordination, if necessary.

چارچوب Spring همچنین از مشخصات Dependency Injection (JSR 330) و Common Annotations (JSR 250) پشتیبانی می کند، که توسعه دهندگان برنامه ممکن است به جای مکانیسم های خاص Spring ارائه شده توسط Spring Framework استفاده کنند.

از Spring Framework 5.0، Spring حداقل به سطح Java EE 7 (به عنوان مثال Servlet 3.1+، JPA 2.1+) نیاز دارد - در حالی که در عین حال یکپارچه سازی خارج از جعبه با API های جدیدتر در سطح Java EE 8 ارائه می کند. (به عنوان مثال Servlet 4.0، JSON Binding API) هنگامی که در زمان اجرا با آن مواجه می شوید. این امر اسپرینگ را کاملاً با به عنوان مثال سازگار نگه می دارد. Tomcat 8 و 9، WebSphere 9 و JBoss EAP 7.

با گذشت زمان، نقش Java EE در توسعه اپلیکیشن تکامل یافته است. در روزهای اولیه Java EE و Spring، برنامه‌ها برای استقرار در سرور برنامه ایجاد شدند. امروزه، با کمک Spring Boot، برنامه‌ها به روشی توسعه‌دهنده و سازگار با فضای ابری ایجاد می‌شوند، با ظرف Servlet تعبیه‌شده و تغییر آن بی‌اهمیت است. از Spring Framework 5، یک برنامه WebFlux حتی از Servlet API مستقیماً استفاده نمی کند و می تواند روی سرورهایی (مانند Netty) که کانتینرهای Servlet نیستند اجرا شود.

بهار به نوآوری و تکامل خود ادامه می دهد. فراتر از چارچوب Spring، پروژه های دیگری مانند Spring Boot، Spring Security، Spring Data، Spring Cloud، Spring Batch و غیره وجود دارد. مهم است که به یاد داشته باشید که هر پروژه مخزن کد منبع، ردیاب مسئله و آهنگ انتشار خود را دارد. برای لیست کامل پروژه های Spring.io/projects را ببینید.

## 3. Design Philosophy

وقتی در مورد یک چارچوب یاد می گیرید ، مهم است که نه تنها چه کاری انجام می دهد بلکه چه اصولی را دنبال می کند. در اینجا اصول راهنما چارچوب بهار آورده شده است:

انتخاب را در هر سطح ارائه دهید. بهار به شما امکان می دهد تا هر چه زودتر تصمیمات طراحی را به تعویق بیندازید. به عنوان مثال ، شما می توانید ارائه دهندگان پایداری را از طریق پیکربندی بدون تغییر کد خود تغییر دهید. در مورد بسیاری از نگرانی های زیرساخت های دیگر و ادغام با API های شخص ثالث نیز همین مسئله صادق است.

دیدگاههای متنوعی را در خود جای دهید. بهار انعطاف پذیری را در بر می گیرد و در مورد نحوه انجام کارها نظر نمی شود. این برنامه از طیف گسترده ای از نیازهای برنامه با دیدگاه های مختلف پشتیبانی می کند.

سازگاری قوی به عقب را حفظ کنید. تکامل بهار با دقت موفق شده است تا تغییرات کمی بین نسخه ها را مجبور کند. بهار از طیف وسیعی از نسخه های JDK و کتابخانه های شخص ثالث برای تسهیل نگهداری برنامه ها و کتابخانه هایی که به بهار بستگی دارند ، پشتیبانی می کند.

در مورد طراحی API مراقبت کنید. تیم بهار فکر و زمان زیادی را برای ساخت API هایی که شهودی هستند و در بسیاری از نسخه ها و سالهای زیادی حفظ می کنند ، می گذارد.

استانداردهای بالایی را برای کیفیت کد تنظیم کنید. چارچوب بهار تأکید جدی بر Javadoc معنی دار ، فعلی و دقیق دارد. این یکی از پروژه های بسیار معدودی است که می تواند ساختار کد تمیز و بدون وابستگی دایره ای بین بسته ها را ادعا کند.

## 4. Feedback and Contributions

برای سؤالات چگونه یا مشکلات تشخیص یا اشکال زدایی ، پیشنهاد می کنیم از StackOverflow استفاده کنید ، و یک صفحه سؤالی داریم که برچسب های پیشنهادی را برای استفاده در لیست قرار می دهد. اگر کاملاً مطمئن هستید که در چارچوب بهار مشکلی وجود دارد یا می خواهید یک ویژگی را پیشنهاد دهید ، لطفاً از ردیاب شماره JIRA استفاده کنید.

اگر راه حل در ذهن دارید یا یک مشکل پیشنهادی دارید ، می توانید درخواست کشش را در GitHub ارسال کنید. با این حال ، لطفاً بخاطر داشته باشید که ، برای همه اما بی اهمیت ترین مسائل ، ما انتظار داریم بلیط در ردیاب موضوع ثبت شود ، جایی که بحث ها در آن صورت می گیرد و سابقه ای را برای مرجع بعدی باقی می گذارد.

برای اطلاعات بیشتر به دستورالعمل های صفحه پروژه کمک کننده و سطح بالا مراجعه کنید.

## 5. Getting Started

اگر به تازگی با Spring شروع کرده اید، ممکن است بخواهید با ایجاد یک برنامه مبتنی بر Spring Boot استفاده از Spring Framework را شروع کنید. Spring Boot یک راه سریع (و با نظر) برای ایجاد یک برنامه کاربردی مبتنی بر Spring آماده تولید ارائه می دهد. این بر اساس چارچوب Spring است، قرارداد را به پیکربندی ترجیح می دهد، و طراحی شده است تا شما را در سریع ترین زمان ممکن راه اندازی کند.

می‌توانید از start.spring.io برای تولید یک پروژه اساسی استفاده کنید یا یکی از راهنماهای «شروع به کار» را دنبال کنید، مانند شروع به ساخت یک وب سرویس RESTful. این راهنماها علاوه بر اینکه هضم آسان‌تری دارند، بر روی کار متمرکز هستند و اکثر آنها بر اساس Spring Boot هستند. آنها همچنین پروژه های دیگری از مجموعه بهار را پوشش می دهند که ممکن است بخواهید هنگام حل یک مشکل خاص در نظر بگیرید.

# Core Technologies

این بخش از مستندات مرجع، تمام آن فناوری‌هایی را پوشش می‌دهد که کاملاً در چارچوب Spring Framework هستند.

مهمترین آنها کانتینر Spring Framework's Inversion of Control (IoC) است. بررسی کامل ظرف IoC Spring Framework با پوشش جامع فناوری های برنامه نویسی جنبه گرا (AOP) Spring دنبال می شود. Spring Framework چارچوب AOP خود را دارد که از نظر مفهومی قابل درک است و با موفقیت 80% نیازهای AOP را در برنامه نویسی سازمانی جاوا برطرف می کند.

پوشش ادغام Spring با AspectJ (در حال حاضر غنی ترین - از نظر ویژگی ها - و مطمئناً بالغ ترین پیاده سازی AOP در فضای سازمانی جاوا) نیز ارائه شده است.

# 1. The IoC container

### 1.1. Introduction to the Spring IoC container and beans

این فصل اجرای چارچوب Spring از اصل وارونگی کنترل (IoC) [1] را پوشش می دهد. IoC همچنین به عنوان تزریق وابستگی (DI) شناخته می شود. این فرآیندی است که طی آن اشیاء وابستگی های خود را تعریف می کنند، یعنی اشیایی که با آنها کار می کنند، فقط از طریق آرگومان های سازنده، آرگومان های یک متد کارخانه، یا ویژگی هایی که روی نمونه شی پس از ساخته شدن یا بازگرداندن آن از یک متد کارخانه ای تنظیم می شوند. . سپس ظرف زمانی که لوبیا را ایجاد می کند آن وابستگی ها را تزریق می کند. این فرآیند اساساً معکوس است، از این رو Inversion of Control (IoC) نامیده می شود، که خود bean با استفاده از ساخت مستقیم کلاس ها یا مکانیزمی مانند الگوی Service Locator، نمونه یا مکان وابستگی های خود را کنترل می کند.

بسته‌های org.springframework.beans و org.springframework.context مبنایی برای کانتینر IoC Spring Framework هستند. رابط BeanFactory مکانیزم پیکربندی پیشرفته ای را ارائه می دهد که قادر به مدیریت هر نوع شی است. ApplicationContext یک رابط فرعی BeanFactory است. ادغام آسان تر با ویژگی های AOP Spring را اضافه می کند. مدیریت منابع پیام (برای استفاده در بین المللی)، انتشار رویداد. و زمینه های خاص لایه برنامه مانند WebApplicationContext برای استفاده در برنامه های کاربردی وب.

به طور خلاصه، BeanFactory چارچوب پیکربندی و عملکردهای اساسی را فراهم می کند، و ApplicationContext عملکردهای ویژه سازمانی بیشتری را اضافه می کند. ApplicationContext یک ابر مجموعه کامل از BeanFactory است و به طور انحصاری در این فصل در توضیحات کانتینر IoC Spring استفاده می شود. برای اطلاعات بیشتر در مورد استفاده از BeanFactory به جای ApplicationContext، به The BeanFactory مراجعه کنید.

در Spring، اشیایی که ستون فقرات برنامه شما را تشکیل می دهند و توسط کانتینر Spring IoC مدیریت می شوند، beans نامیده می شوند. یک لوبیا شیئی است که توسط یک کانتینر Spring IoC نمونه سازی، مونتاژ و مدیریت می شود. در غیر این صورت، یک لوبیا به سادگی یکی از بسیاری از اشیاء در برنامه شما است. لوبیاها و وابستگی‌های موجود در میان آن‌ها، در فراداده‌های پیکربندی استفاده‌شده توسط یک ظرف منعکس می‌شوند.

## 1.2. Container overview

رابط org.springframework.context.ApplicationContext نشان دهنده ظرف Spring IoC است و مسئول نمونه سازی، پیکربندی و مونتاژ دانه های فوق الذکر است. ظرف با خواندن فراداده های پیکربندی دستورالعمل های خود را در مورد مواردی که باید نمونه سازی، پیکربندی و مونتاژ شود، دریافت می کند. فراداده پیکربندی در XML، حاشیه نویسی جاوا یا کد جاوا نشان داده می شود. این به شما امکان می دهد اشیایی که برنامه شما را تشکیل می دهند و وابستگی های متقابل غنی بین چنین اشیایی را بیان کنید.

چندین پیاده سازی از رابط ApplicationContext به صورت خارج از جعبه با Spring عرضه می شود. در برنامه های مستقل ایجاد یک نمونه از ClassPathXmlApplicationContext یا FileSystemXmlApplicationContext معمول است. در حالی که XML فرمت سنتی برای تعریف فراداده پیکربندی بوده است، می‌توانید با ارائه مقدار کمی از پیکربندی XML برای فعال کردن پشتیبانی از این فرمت‌های فراداده اضافی، به ظرف دستور دهید که از حاشیه‌نویسی یا کد جاوا به عنوان قالب فراداده استفاده کند.

در اکثر سناریوهای برنامه کاربردی، برای نمونه سازی یک یا چند نمونه از یک ظرف Spring IoC، به کد کاربر صریح نیازی نیست. به عنوان مثال، در یک سناریوی برنامه وب، یک هشت خط (یا بیشتر) ساده از XML توصیفگر وب boilerplate در فایل web.xml برنامه معمولاً کافی است (نمونه سازی مناسب ApplicationContext برای برنامه های کاربردی وب را ببینید). اگر از محیط توسعه مبتنی بر Spring Tool Suite Eclipse استفاده می کنید، این پیکربندی دیگ بخار را می توان به راحتی با چند کلیک ماوس یا فشار دادن کلید ایجاد کرد.

نمودار زیر نمای سطح بالایی از نحوه عملکرد Spring است. کلاس های برنامه شما با ابرداده های پیکربندی ترکیب می شوند تا پس از ایجاد و تنظیم اولیه ApplicationContext، یک سیستم یا برنامه کاملاً پیکربندی شده و قابل اجرا داشته باشید.



#### 1.2.1. Configuration metadata

همانطور که نمودار قبل نشان می دهد، محفظه Spring IoC شکلی از فراداده پیکربندی را مصرف می کند. این فراداده پیکربندی نشان می‌دهد که چگونه شما به‌عنوان یک توسعه‌دهنده برنامه به کانتینر Spring می‌گویید که اشیاء را در برنامه شما نمونه‌سازی، پیکربندی و مونتاژ کند.

فراداده پیکربندی به طور سنتی در قالب XML ساده و شهودی ارائه می‌شود، که بیشتر این فصل از آن برای انتقال مفاهیم و ویژگی‌های کلیدی کانتینر Spring IoC استفاده می‌کند.

فراداده مبتنی بر XML تنها فرم مجاز فراداده پیکربندی نیست. خود محفظه Spring IoC کاملاً از قالبی که این فراداده پیکربندی واقعاً در آن نوشته شده است جدا شده است. این روزها بسیاری از توسعه دهندگان پیکربندی مبتنی بر جاوا را برای برنامه های Spring خود انتخاب می کنند.

فراداده مبتنی بر XML تنها فرم مجاز فراداده پیکربندی نیست. خود محفظه Spring IoC کاملاً از قالبی که این فراداده پیکربندی واقعاً در آن نوشته شده است جدا شده است. این روزها بسیاری از توسعه دهندگان پیکربندی مبتنی بر جاوا را برای برنامه های Spring خود انتخاب می کنند.

برای کسب اطلاعات در مورد استفاده از سایر اشکال فراداده با ظرف Spring، نگاه کنید به:

پیکربندی مبتنی بر حاشیه نویسی: Spring 2.5 پشتیبانی از ابرداده پیکربندی مبتنی بر حاشیه نویسی را معرفی کرد.

Java bean config: با شروع Spring 3.0، بسیاری از ویژگی های ارائه شده توسط پروژه Spring JavaConfig بخشی از چارچوب اصلی Spring Framework شدند. بنابراین شما می توانید با استفاده از جاوا به جای فایل های XML، beans را برای کلاس های برنامه خود تعریف کنید. برای استفاده از این ویژگی‌های جدید، به حاشیه‌نویسی‌های @Configuration، @Bean، @Import و @DependsOn مراجعه کنید.

پیکربندی فنری شامل حداقل یک تعریف و معمولاً بیش از یک تعریف است که ظرف باید آن را مدیریت کند. فراداده پیکربندی مبتنی بر XML این دانه‌ها را به‌عنوان عناصر <bean/> در یک عنصر سطح بالای <beans/> پیکربندی می‌کند. پیکربندی جاوا معمولاً از روش‌های حاشیه‌نویسی @Bean در یک کلاس @Configuration استفاده می‌کند.

این تعاریف bean با اشیاء واقعی که برنامه شما را تشکیل می دهند مطابقت دارد. معمولاً اشیاء لایه سرویس، اشیاء دسترسی به داده (DAO)، اشیاء ارائه مانند نمونه‌های Struts Action، اشیاء زیرساخت مانند Hibernate SessionFactories، صف‌های JMS و غیره را تعریف می‌کنید. به طور معمول، یکی اشیاء دامنه ریز را در کانتینر پیکربندی نمی کند، زیرا معمولاً مسئولیت DAO ها و منطق تجاری ایجاد و بارگذاری اشیاء دامنه است. با این حال، می توانید از ادغام Spring با AspectJ برای پیکربندی اشیایی که خارج از کنترل یک ظرف IoC ایجاد شده اند استفاده کنید. استفاده از AspectJ برای وابستگی تزریق اشیاء دامنه با Spring را ببینید.

مثال زیر ساختار اصلی فراداده های پیکربندی مبتنی بر XML را نشان می دهد:

<?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

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

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

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

*<!-- collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions go here -->*

</beans>

خصیصه id رشته ای است که از آن برای شناسایی تعریف منحصر به فرد bean استفاده می کنید. ویژگی class نوع bean را تعریف می کند و از نام کلاس کاملاً واجد شرایط استفاده می کند. مقدار مشخصه id به اشیاء همکار اشاره دارد. XML برای ارجاع به اشیاء همکار در این مثال نشان داده نشده است. برای اطلاعات بیشتر به Dependencies مراجعه کنید.

#### 1.2.2. Instantiating a container

نمونه سازی یک کانتینر Spring IoC ساده است. مسیر یا مسیرهای ارائه‌شده به سازنده ApplicationContext در واقع رشته‌های منبعی هستند که به ظرف اجازه می‌دهند ابرداده‌های پیکربندی را از انواع منابع خارجی مانند سیستم فایل محلی، از Java CLASSPATH و غیره بارگیری کند.

ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");

پس از آشنایی با کانتینر IoC Spring، ممکن است بخواهید بیشتر در مورد انتزاع منابع Spring بدانید، همانطور که در Resources توضیح داده شده است، که مکانیزم مناسبی برای خواندن یک جریان ورودی از مکان های تعریف شده در یک نحو URI ارائه می دهد. به طور خاص، مسیرهای منبع برای ساختن زمینه های برنامه ها همانطور که در زمینه های برنامه و مسیرهای منبع توضیح داده شده است، استفاده می شود.

مثال زیر فایل پیکربندی اشیاء لایه سرویس (services.xml) را نشان می دهد:

<?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

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

*<!-- services -->*

<bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="itemDao" ref="itemDao"/>

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for services go here -->*

</beans>

مثال زیر فایل پیکربندی اشیاء لایه سرویس (services.xml) را نشان می دهد:

<?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

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

<bean id="accountDao"

class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">

*<!-- additional collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions for data access objects go here -->*

</beans>

در مثال قبل، لایه سرویس از کلاس PetStoreServiceImpl و دو شیء دسترسی به داده از نوع JpaAccountDao و JpaItemDao (براساس استاندارد نگاشت شیء/رابطه JPA) تشکیل شده است. عنصر نام ویژگی به نام ویژگی JavaBean و عنصر ref به نام تعریف دیگری از bean اشاره دارد. این پیوند بین عناصر id و ref وابستگی بین اشیاء همکار را بیان می کند. برای جزئیات پیکربندی وابستگی‌های یک شی، به Dependencies مراجعه کنید.

Composing XML-based configuration metadata

این می تواند مفید باشد که تعاریف bean چندین فایل XML را در بر گیرند. اغلب هر فایل پیکربندی XML فردی یک لایه یا ماژول منطقی در معماری شما را نشان می دهد.

شما می توانید از سازنده زمینه برنامه برای بارگذاری تعاریف bean از تمام این قطعات XML استفاده کنید. همانطور که در بخش قبل نشان داده شد، این سازنده چندین مکان منبع را می گیرد. از طرف دیگر، از یک یا چند مورد از عنصر <import/> برای بارگیری تعاریف bean از یک فایل یا فایل دیگر استفاده کنید. مثلا:

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

در مثال قبلی، تعاریف خارجی bean از سه فایل بارگذاری می‌شوند: services.xml، messageSource.xml، و themeSource.xml. همه مسیرهای مکان نسبت به فایل تعریفی هستند که وارد کردن را انجام می دهد، بنابراین services.xml باید در همان دایرکتوری یا مکان کلاس مسیر با فایلی که وارد کردن را انجام می دهد، قرار داشته باشد، در حالی که messageSource.xml و themeSource.xml باید در یک مکان منابع زیر مکان قرار داشته باشند. از فایل وارد کننده همانطور که می بینید، یک اسلش پیشرو نادیده گرفته می شود، اما با توجه به نسبی بودن این مسیرها، بهتر است که به هیچ وجه از اسلش استفاده نکنید. محتویات فایل‌هایی که وارد می‌شوند، از جمله عنصر سطح بالای <beans/>، باید طبق طرح Spring دارای تعاریف معتبر XML bean باشند.

ارجاع فایل ها در فهرست های والد با استفاده از مسیر نسبی "../" امکان پذیر است، اما توصیه نمی شود. انجام این کار باعث ایجاد وابستگی به فایلی می شود که خارج از برنامه فعلی است. به طور خاص، این مرجع برای URL های "classpath:" توصیه نمی شود (به عنوان مثال، "classpath:../services.xml")، که در آن فرآیند حل زمان اجرا "نزدیک ترین" ریشه classpath را انتخاب می کند و سپس به فهرست اصلی آن نگاه می کند. تغییرات پیکربندی Classpath ممکن است منجر به انتخاب دایرکتوری متفاوت و نادرستی شود.

همیشه می‌توانید از مکان‌های منابع کاملاً واجد شرایط به جای مسیرهای نسبی استفاده کنید: به عنوان مثال، "file:C:/config/services.xml" یا "classpath:/config/services.xml". با این حال، توجه داشته باشید که پیکربندی برنامه خود را به مکان های مطلق خاصی متصل می کنید. به طور کلی ترجیح داده می شود که برای چنین مکان های مطلق، یک جهت غیر مستقیم حفظ شود، به عنوان مثال، از طریق متغیرهای "${…​}" که در زمان اجرا با ویژگی های سیستم JVM حل می شوند.

دستورالعمل واردات یک ویژگی است که توسط خود فضای نام beans ارائه شده است. ویژگی های پیکربندی بیشتر فراتر از تعاریف ساده bean در مجموعه ای از فضاهای نام XML ارائه شده توسط Spring، به عنوان مثال، موجود است. فضای نام "context" و "util".

دستورالعمل واردات یک ویژگی است که توسط خود فضای نام beans ارائه شده است. ویژگی های پیکربندی بیشتر فراتر از تعاریف ساده bean در مجموعه ای از فضاهای نام XML ارائه شده توسط Spring، به عنوان مثال، موجود است. فضای نام "context" و "util".

The Groovy Bean Definition DSL

به عنوان مثالی دیگر برای ابرداده های پیکربندی خارجی، تعاریف bean را می توان در Spring's Groovy Bean Definition DSL، همانطور که از چارچوب Grails شناخته می شود، بیان کرد. به طور معمول، چنین پیکربندی در یک فایل ".groovy" با ساختاری به شرح زیر است:

beans {

dataSource(BasicDataSource) {

driverClassName = "org.hsqldb.jdbcDriver"

url = "jdbc:hsqldb:mem:grailsDB"

username = "sa"

password = ""

settings = [mynew:"setting"]

}

sessionFactory(SessionFactory) {

dataSource = dataSource

}

myService(MyService) {

nestedBean = { AnotherBean bean ->

dataSource = dataSource

}

}

}

این سبک پیکربندی تا حد زیادی معادل تعاریف XML Bean است و حتی از فضاهای نام پیکربندی XML Spring پشتیبانی می‌کند. همچنین اجازه می دهد تا فایل های تعریف XML bean را از طریق دستورالعمل "importBeans" وارد کنید.

#### 1.2.3. Using the container

ApplicationContext رابطی برای یک کارخانه پیشرفته است که قادر به حفظ رجیستری از دانه های مختلف و وابستگی های آنها است. با استفاده از روش T getBean(String name, Class<T> requiredType) می توانید نمونه هایی از bean های خود را بازیابی کنید.

ApplicationContext شما را قادر می سازد تا تعاریف bean را بخوانید و به صورت زیر به آنها دسترسی داشته باشید:

*// create and configure beans*

ApplicationContext context = **new** ClassPathXmlApplicationContext("services.xml", "daos.xml");

*// retrieve configured instance*

PetStoreService service = context.getBean("petStore", PetStoreService.class);

*// use configured instance*

List<String> userList = service.getUsernameList();

با پیکربندی Groovy، bootstrapping بسیار شبیه به نظر می رسد، فقط یک کلاس پیاده سازی زمینه متفاوت که Groovy-aware است (اما تعاریف XML bean را نیز درک می کند):

ApplicationContext context = **new** GenericGroovyApplicationContext("services.groovy", "daos.groovy");

منعطف ترین نوع GenericApplicationContext در ترکیب با نمایندگان خواننده است، به عنوان مثال. با XmlBeanDefinitionReader برای فایل های XML:

GenericApplicationContext context = **new** GenericApplicationContext();

**new** XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");

context.refresh();

یا با GroovyBeanDefinitionReader برای فایل های Groovy:

GenericApplicationContext context = **new** GenericApplicationContext();

**new** GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");

context.refresh();

چنین نمایندگان خواننده را می توان در همان ApplicationContext با هم ترکیب کرد و در صورت تمایل، تعاریف bean را از منابع پیکربندی متنوع خواند.

سپس می توانید از getBean برای بازیابی نمونه هایی از لوبیاهای خود استفاده کنید. رابط ApplicationContext چند روش دیگر برای بازیابی لوبیا دارد، اما در حالت ایده آل کد برنامه شما هرگز نباید از آنها استفاده کند. در واقع، کد برنامه شما نباید به هیچ وجه به متد ()getBean فراخوانی کند، و بنابراین اصلاً وابستگی به API های Spring نداشته باشد. به عنوان مثال، ادغام Spring با چارچوب‌های وب، تزریق وابستگی را برای اجزای مختلف چارچوب وب مانند کنترل‌کننده‌ها و دانه‌های مدیریت‌شده با JSF فراهم می‌کند و به شما این امکان را می‌دهد که وابستگی به یک Bean خاص را از طریق ابرداده (مثلاً یک حاشیه‌نویسی سیم‌کشی خودکار) اعلام کنید.

## 1.3. Bean overview

ظرف Spring IoC یک یا چند لوبیا را مدیریت می کند. این دانه‌ها با ابرداده‌های پیکربندی که به کانتینر ارائه می‌دهید، ایجاد می‌شوند، به عنوان مثال، در قالب تعاریف XML <bean/>.

در خود ظرف، این تعاریف bean به عنوان اشیاء BeanDefinition نشان داده می‌شوند که شامل (در میان اطلاعات دیگر) فوق داده‌های زیر است:

نام کلاس واجد شرایط بسته: معمولاً کلاس پیاده سازی واقعی bean تعریف می شود.

عناصر پیکربندی رفتاری Bean، که بیان می‌کند که bean چگونه باید در کانتینر رفتار کند (محدوده، تماس‌های چرخه حیات، و غیره).

ارجاع به دانه های دیگری که برای انجام کار لوبیا مورد نیاز است. به این مراجع همکار یا وابستگی نیز می گویند.

سایر تنظیمات پیکربندی برای تنظیم در شی جدید ایجاد شده، به عنوان مثال، تعداد اتصالات برای استفاده در یک Bean که یک مخزن اتصال را مدیریت می کند، یا محدودیت اندازه استخر.

این ابرداده به مجموعه ای از ویژگی ها ترجمه می شود که هر تعریف bean را تشکیل می دهد.

| Table 1. The bean definition | |
| --- | --- |
| **Property** | **Explained in…​** |
| class | [Instantiating beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-class) |
| name | [Naming beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-beanname) |
| scope | [Bean scopes](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes) |
| constructor arguments | [Dependency Injection](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-collaborators) |
| properties | [Dependency Injection](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-collaborators) |
| autowiring mode | [Autowiring collaborators](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-autowire) |
| lazy-initialization mode | [Lazy-initialized beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lazy-init) |
| initialization method | [Initialization callbacks](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lifecycle-initializingbean) |
| destruction method | [Destruction callbacks](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-lifecycle-disposablebean) |
|  |  |

علاوه بر تعاریف bean که حاوی اطلاعاتی در مورد نحوه ایجاد یک bean خاص است، پیاده سازی های ApplicationContext همچنین اجازه ثبت اشیاء موجود را که خارج از کانتینر توسط کاربران ایجاد می شوند را می دهد. این کار با دسترسی به BeanFactory ApplicationContext از طریق متد getBeanFactory() انجام می شود که اجرای BeanFactory DefaultListableBeanFactory را برمی گرداند. DefaultListableBeanFactory این ثبت را از طریق متدهای registerSingleton(..) و registerBeanDefinition(..) پشتیبانی می کند. با این حال، برنامه‌های کاربردی معمولی تنها با لوبیاهایی که از طریق تعاریف متادیتا bean تعریف شده‌اند، کار می‌کنند.

فراداده‌های Bean و نمونه‌های تک‌تنه ارائه‌شده به‌صورت دستی باید در اسرع وقت ثبت شوند تا ظرف به درستی درباره آنها در طول سیم‌کشی خودکار و سایر مراحل درون‌نگاری استدلال کند. در حالی که نادیده گرفتن ابرداده‌های موجود و نمونه‌های singleton موجود تا حدی پشتیبانی می‌شود، ثبت bean جدید در زمان اجرا (همزمان با دسترسی زنده به کارخانه) به طور رسمی پشتیبانی نمی‌شود و ممکن است منجر به استثناهای دسترسی همزمان و/یا حالت ناسازگار در ظرف bean شود. .

### 1.3.1. Naming beans

هر لوبیا یک یا چند شناسه دارد. این شناسه ها باید در ظرفی که لوبیا را میزبانی می کند منحصر به فرد باشد. یک لوبیا معمولاً فقط یک شناسه دارد، اما اگر به بیش از یک شناسه نیاز داشته باشد، موارد اضافی را می توان نام مستعار در نظر گرفت.

در فراداده پیکربندی مبتنی بر XML، از ویژگی‌های id و/یا نام برای تعیین شناسه(های) bean استفاده می‌کنید. ویژگی id به شما امکان می دهد دقیقا یک شناسه را مشخص کنید. معمولاً این نام‌ها حروف عددی هستند ("myBean"، "fooService"، و غیره)، اما ممکن است شامل کاراکترهای خاصی نیز باشند. اگر می‌خواهید نام‌های مستعار دیگری را به bean معرفی کنید، می‌توانید آن‌ها را در ویژگی name که با کاما (،)، نقطه ویرگول (;) یا فاصله سفید از هم جدا شده‌اند، مشخص کنید. به عنوان یک یادداشت تاریخی، در نسخه های قبل از Spring 3.1، ویژگی id به عنوان یک نوع xsd:ID تعریف شده بود که کاراکترهای ممکن را محدود می کرد. از 3.1، به عنوان یک نوع xsd:string تعریف شده است. توجه داشته باشید که منحصر به فرد بودن شناسه bean هنوز توسط کانتینر اعمال می شود، البته دیگر توسط تجزیه کننده های XML اعمال نمی شود.

شما نیازی به ارائه نام یا شناسه برای لوبیا ندارید. اگر هیچ نام یا شناسه ای به صراحت ارائه نشده باشد، ظرف یک نام منحصر به فرد برای آن لوبیا ایجاد می کند. با این حال، اگر می‌خواهید به آن bean با نام اشاره کنید، از طریق استفاده از عنصر ref یا جستجوی سبک Service Locator، باید یک نام ارائه کنید. انگیزه های عدم ارائه نام مربوط به استفاده از ایننر لوبیا و همکاران سیم کشی خودکار است.

Bean Naming Conventions

قرارداد استفاده از قرارداد استاندارد جاوا برای مثال نام فیلدها هنگام نامگذاری دانه ها است. یعنی نام لوبیا با حروف کوچک شروع می شود و از آن به بعد شتری است. نمونه هایی از این نام ها (بدون نقل قول) 'accountManager'، 'accountService'، 'userDao'، 'loginController' و غیره خواهد بود.

نام گذاری دانه ها به طور مداوم پیکربندی شما را برای خواندن و درک آسان تر می کند، و اگر از Spring AOP استفاده می کنید، هنگام اعمال توصیه به مجموعه ای از دانه های مرتبط با نام کمک زیادی می کند.

با اسکن کامپوننت در مسیر کلاس، Spring نام‌های bean را برای مؤلفه‌های بدون نام تولید می‌کند، از قوانین بالا پیروی می‌کند: اساساً، نام کلاس ساده را می‌گیرد و کاراکتر اولیه آن را به حروف کوچک تبدیل می‌کند. با این حال، در حالت خاص (غیر معمول) که بیش از یک کاراکتر وجود دارد و هر دو کاراکتر اول و دوم بزرگ هستند، پوشش اصلی حفظ می‌شود. اینها همان قوانینی هستند که توسط java.beans.Introspector.decapitalize (که Spring در اینجا از آن استفاده می کند) تعریف شده است.

Aliasing a bean outside the bean definition

در خود تعریف bean، با استفاده از ترکیبی از حداکثر یک نام مشخص شده توسط ویژگی id، و هر تعداد نام دیگر در ویژگی name، می توانید بیش از یک نام برای bean ارائه دهید. این نام‌ها می‌توانند نام مستعار معادل همان bean باشند، و برای برخی موقعیت‌ها مفید هستند، مانند اجازه دادن به هر جزء در یک برنامه کاربردی برای ارجاع به یک وابستگی مشترک با استفاده از نام bean که مختص خود آن جزء است.

با این حال، مشخص کردن همه نام‌های مستعار که در آن bean واقعاً تعریف شده است، همیشه کافی نیست. گاهی اوقات توصیه می شود نام مستعار برای یک لوبیا معرفی شود که در جای دیگری تعریف شده است. معمولاً در سیستم‌های بزرگ که پیکربندی بین هر زیرسیستم تقسیم می‌شود، هر زیرسیستم مجموعه‌ای از تعاریف شی خاص خود را دارد. در فراداده پیکربندی مبتنی بر XML، می‌توانید از عنصر <alias/> برای انجام این کار استفاده کنید.

<alias name="fromName" alias="toName"/>

در این مورد، یک لوبیا (در همان ظرف) با نام fromName نیز ممکن است پس از استفاده از این تعریف مستعار، به عنوان toName نامیده شود.

به عنوان مثال، فراداده پیکربندی برای زیرسیستم A ممکن است به یک DataSource با نام subsystemA-dataSource اشاره کند. فراداده پیکربندی برای زیرسیستم B ممکن است به یک DataSource با نام subsystemB-dataSource اشاره کند. هنگام نوشتن برنامه اصلی که از هر دو زیرسیستم استفاده می کند، برنامه اصلی به DataSource با نام myApp-dataSource اشاره می کند. برای اینکه هر سه نام به یک شی اشاره داشته باشند، می توانید تعاریف نام مستعار زیر را به فراداده پیکربندی اضافه کنید:

<alias name="myApp-dataSource" alias="subsystemA-dataSource"/>

<alias name="myApp-dataSource" alias="subsystemB-dataSource"/>

Java-configuration

اگر از پیکربندی جاوا استفاده می‌کنید، می‌توان از حاشیه‌نویسی @Bean برای ارائه نام‌های مستعار استفاده کرد، برای جزئیات بیشتر به استفاده از حاشیه‌نویسی @Bean مراجعه کنید.

### 1.3.2. Instantiating beans

تعریف لوبیا در اصل دستور العمل ایجاد یک یا چند شی است. کانتینر در صورت درخواست از دستور العمل یک لوبیای نامگذاری شده نگاه می کند و از ابرداده پیکربندی محصور شده توسط آن تعریف لوبیا برای ایجاد (یا به دست آوردن) یک شیء واقعی استفاده می کند.

اگر از ابرداده پیکربندی مبتنی بر XML استفاده می کنید ، نوع (یا کلاس) شیء را که باید در ویژگی کلاس عنصر <bean/> فوری شود ، مشخص می کنید. این ویژگی کلاس ، که در داخل یک ویژگی کلاس در نمونه Beandefinition است ، معمولاً اجباری است. (برای استثنائات ، با استفاده از یک روش کارخانه نمونه و وراثت تعریف لوبیا به فوری مراجعه کنید.) شما از یکی از دو روش از ویژگی کلاس استفاده می کنید:

به طور معمول ، برای مشخص کردن کلاس لوبیا در موردی که خود ظرف به طور مستقیم لوبیا را با فراخوانی سازنده خود به طور انعکاس ، تا حدودی معادل کد جاوا با استفاده از اپراتور جدید ایجاد می کند ، ایجاد می شود.

برای مشخص کردن کلاس واقعی حاوی روش کارخانه استاتیک که برای ایجاد جسم مورد استفاده قرار می گیرد ، در مورد کمتر متداول که ظرف یک روش کارخانه استاتیک را در یک کلاس فراخوانی می کند تا لوبیا ایجاد شود. نوع شیء که از فراخوانی روش کارخانه استاتیک برگشته است ممکن است همان کلاس یا کلاس دیگری باشد.

Inner class names

تعریف لوبیا در اصل دستور العمل ایجاد یک یا چند شی است. کانتینر در صورت درخواست از دستور العمل یک لوبیای نامگذاری شده نگاه می کند و از ابرداده پیکربندی محصور شده توسط آن تعریف لوبیا برای ایجاد (یا به دست آوردن) یک شیء واقعی استفاده می کند.

اگر از ابرداده پیکربندی مبتنی بر XML استفاده می کنید ، نوع (یا کلاس) شیء را که باید در ویژگی کلاس عنصر <bean/> فوری شود ، مشخص می کنید. این ویژگی کلاس ، که در داخل یک ویژگی کلاس در نمونه Beandefinition است ، معمولاً اجباری است. (برای استثنائات ، با استفاده از یک روش کارخانه نمونه و وراثت تعریف لوبیا به فوری مراجعه کنید.) شما از یکی از دو روش از ویژگی کلاس استفاده می کنید:

به طور معمول ، برای مشخص کردن کلاس لوبیا در موردی که خود ظرف به طور مستقیم لوبیا را با فراخوانی سازنده خود به طور انعکاس ، تا حدودی معادل کد جاوا با استفاده از اپراتور جدید ایجاد می کند ، ایجاد می شود.

برای مشخص کردن کلاس واقعی حاوی روش کارخانه استاتیک که برای ایجاد جسم مورد استفاده قرار می گیرد ، در مورد کمتر متداول که ظرف یک روش کارخانه استاتیک را در یک کلاس فراخوانی می کند تا لوبیا ایجاد شود. نوع شیء که از فراخوانی روش کارخانه استاتیک برگشته است ممکن است همان کلاس یا کلاس دیگری باشد.

Instantiation with a constructor

وقتی یک bean با رویکرد سازنده ایجاد می‌کنید، همه کلاس‌های عادی توسط Spring قابل استفاده و سازگار هستند. یعنی کلاسی که توسعه می‌یابد نیازی به پیاده‌سازی رابط‌های خاص یا کدگذاری به روش خاصی ندارد. صرفاً مشخص کردن کلاس bean کافی است. با این حال، بسته به نوع IoC که برای آن Bean خاص استفاده می‌کنید، ممکن است به یک سازنده پیش‌فرض (خالی) نیاز داشته باشید.

کانتینر Spring IoC می تواند تقریباً هر کلاسی را که می خواهید مدیریت کند، مدیریت کند. این به مدیریت جاوابین واقعی محدود نمی شود. اکثر کاربران Spring JavaBeans واقعی را تنها با سازنده پیش‌فرض (بدون آرگومان) و تنظیم‌کننده‌ها و دریافت‌کننده‌های مناسب که بر اساس ویژگی‌های موجود در ظرف مدل‌سازی شده‌اند، ترجیح می‌دهند. همچنین می‌توانید کلاس‌های عجیب‌تری به سبک غیر لوبیا در ظرف خود داشته باشید. به عنوان مثال، اگر شما نیاز به استفاده از یک استخر اتصال قدیمی دارید که مطلقاً با مشخصات JavaBean مطابقت ندارد، Spring می‌تواند آن را نیز مدیریت کند.

با فراداده پیکربندی مبتنی بر XML می توانید کلاس bean خود را به صورت زیر مشخص کنید:

<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

برای جزئیات در مورد مکانیسم ارائه آرگومان‌ها به سازنده (در صورت نیاز) و تنظیم ویژگی‌های نمونه شی پس از ساخته شدن شی، به تزریق وابستگی‌ها مراجعه کنید.

Instantiation with a static factory method

هنگام تعریف bean که با روش کارخانه ایستا ایجاد می کنید، از ویژگی class برای تعیین کلاس حاوی متد static factory و از ویژگی با نام factory-method برای تعیین نام خود متد کارخانه استفاده می کنید. شما باید بتوانید این متد را فراخوانی کنید (با آرگومان های اختیاری همانطور که در ادامه توضیح داده شد) و یک شی زنده را برگردانید، که متعاقباً طوری رفتار می شود که گویی از طریق سازنده ایجاد شده است. یکی از کاربردهای چنین تعریفی، فراخوانی کارخانه‌های استاتیک در کدهای قدیمی است.

تعریف bean زیر مشخص می کند که bean با فراخوانی یک روش کارخانه ایجاد می شود. تعریف نوع (کلاس) شیء برگشتی را مشخص نمی کند، فقط کلاس حاوی متد کارخانه را مشخص می کند. در این مثال، متد ()createInstance باید یک متد ثابت باشد.

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

**public** **class** **ClientService** {

**private** **static** ClientService clientService = **new** ClientService();

**private** ClientService() {}

**public** **static** ClientService createInstance() {

**return** clientService;

}

}

برای جزئیات بیشتر درباره مکانیسم ارائه آرگومان‌های (اختیاری) به روش کارخانه و تنظیم ویژگی‌های نمونه شی پس از بازگرداندن شی از کارخانه، به جزئیات وابستگی‌ها و پیکربندی مراجعه کنید.

Instantiation using an instance factory method

مشابه نمونه سازی از طریق روش کارخانه ایستا، نمونه سازی با روش کارخانه نمونه از یک روش غیر استاتیک از یک دانه موجود از ظرف برای ایجاد یک دانه جدید فراخوانی می کند. برای استفاده از این مکانیسم، صفت کلاس را خالی بگذارید، و در ویژگی factory-bean، نام یک bean را در ظرف فعلی (یا والد/اجداد) که حاوی متد نمونه‌ای است که قرار است برای ایجاد شیء فراخوانی شود، مشخص کنید. نام خود روش کارخانه را با ویژگی factory-method تنظیم کنید.

*<!-- the factory bean, which contains a method called createInstance() -->*

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

*<!-- the bean to be created via the factory bean -->*

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

}

همانطور که در اینجا نشان داده شده است، یک کلاس کارخانه می تواند بیش از یک روش کارخانه را نیز در خود جای دهد:

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

*<!-- inject any dependencies required by this locator bean -->*

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

**public** **class** **DefaultServiceLocator** {

**private** **static** ClientService clientService = **new** ClientServiceImpl();

**private** **static** AccountService accountService = **new** AccountServiceImpl();

**public** ClientService createClientServiceInstance() {

**return** clientService;

}

**public** AccountService createAccountServiceInstance() {

**return** accountService;

}

}

این رویکرد نشان می دهد که خود کارخانه لوبیا را می توان از طریق تزریق وابستگی (DI) مدیریت و پیکربندی کرد. به جزئیات وابستگی ها و پیکربندی ها مراجعه کنید.

در مستندات Spring، Factory Bean به Bean اطلاق می‌شود که در ظرف Spring پیکربندی شده است که از طریق یک نمونه یا روش کارخانه ایستا، اشیاء را ایجاد می‌کند. در مقابل، FactoryBean (به حروف بزرگ توجه کنید) به FactoryBean مخصوص Spring اشاره دارد.

## 1.4. Dependencies

یک برنامه کاربردی معمولی سازمانی از یک شی (یا در اصطلاح بهار) تشکیل نشده است. حتی ساده ترین برنامه دارای چند شی است که با هم کار می کنند تا آنچه را که کاربر نهایی به عنوان یک برنامه منسجم می بیند ارائه دهد. این بخش بعدی توضیح می‌دهد که چگونه از تعریف تعدادی از تعاریف bean که به تنهایی می‌مانند، به یک کاربرد کاملاً تحقق‌یافته که در آن اشیاء برای رسیدن به یک هدف با یکدیگر همکاری می‌کنند، می‌روید.

### 1.4.1. Dependency Injection

تزریق وابستگی (DI) فرآیندی است که طی آن اشیاء وابستگی‌های خود، یعنی سایر اشیاء را که با آنها کار می‌کنند، تنها از طریق آرگومان‌های سازنده، آرگومان‌های یک متد کارخانه یا ویژگی‌هایی که بر روی نمونه شی پس از ساخته شدن یا برگرداندن آن تنظیم می‌شوند، تعریف می‌کنند. از روش کارخانه ای سپس ظرف زمانی که لوبیا را ایجاد می کند آن وابستگی ها را تزریق می کند. این فرآیند اساساً معکوس است، از این رو Inversion of Control (IoC) نامیده می شود، که خود bean نمونه یا مکان وابستگی های خود را به تنهایی با استفاده از ساخت مستقیم کلاس ها یا الگوی Service Locator کنترل می کند.

کد با اصل DI تمیزتر است و جداسازی زمانی موثرتر است که اشیا با وابستگی های خود ارائه شوند. شیء وابستگی های خود را جستجو نمی کند و مکان یا کلاس وابستگی ها را نمی داند. به این ترتیب، تست کلاس‌های شما آسان‌تر می‌شود، به‌ویژه زمانی که وابستگی‌ها بر روی رابط‌ها یا کلاس‌های پایه انتزاعی هستند، که اجازه می‌دهد از پیاده‌سازی‌های خرد یا ساختگی در تست‌های واحد استفاده شود.

DI در دو نوع اصلی وجود دارد، تزریق وابستگی مبتنی بر سازنده و تزریق وابستگی مبتنی بر Setter.

#### Constructor-based dependency injection

DI مبتنی بر سازنده با فراخوانی یک سازنده با تعدادی آرگومان که هر کدام یک وابستگی را نشان می‌دهند، انجام می‌شود. فراخوانی یک روش کارخانه ایستا با آرگومان های خاص برای ساخت bean تقریباً معادل است، و این بحث آرگومان ها را برای سازنده و روش کارخانه ایستا به طور مشابه مورد بررسی قرار می دهد. مثال زیر کلاسی را نشان می‌دهد که فقط با تزریق سازنده می‌تواند وابستگی تزریق شود. توجه داشته باشید که هیچ چیز خاصی در مورد این کلاس وجود ندارد، این یک POJO است که هیچ وابستگی به رابط های خاص کانتینر، کلاس های پایه یا حاشیه نویسی ندارد.

**public** **class** **SimpleMovieLister** {

*// the SimpleMovieLister has a dependency on a MovieFinder*

**private** MovieFinder movieFinder;

*// a constructor so that the Spring container can inject a MovieFinder*

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

#### Constructor argument resolution

تطبیق وضوح آرگومان سازنده با استفاده از نوع آرگومان رخ می دهد. اگر هیچ ابهامی بالقوه در آرگومان‌های سازنده تعریف bean وجود نداشته باشد، ترتیبی که آرگومان‌های سازنده در تعریف bean تعریف می‌شوند، ترتیبی است که در آن آرگومان‌ها به سازنده مناسب زمانی که bean در حال نمونه‌سازی است، ارائه می‌شود. کلاس زیر را در نظر بگیرید:

**package** x.y;

**public** **class** **Foo** {

**public** Foo(Bar bar, Baz baz) {

*// ...*

}

}

هیچ ابهامی بالقوه وجود ندارد، با این فرض که کلاس های Bar و Baz با وراثت به هم مرتبط نیستند. بنابراین، پیکربندی زیر به خوبی کار می‌کند، و نیازی نیست که شاخص‌ها و/یا انواع آرگومان سازنده را به طور صریح در عنصر <constructor-arg/> مشخص کنید.

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

هنگامی که به bean دیگری ارجاع داده می شود، نوع آن مشخص می شود و تطبیق می تواند رخ دهد (همانطور که در مثال قبل رخ داد). وقتی از یک نوع ساده مانند <value>true</value> استفاده می‌شود، Spring نمی‌تواند نوع مقدار را تعیین کند و بنابراین نمی‌تواند بدون کمک با نوع آن مطابقت داشته باشد. کلاس زیر را در نظر بگیرید:

**package** examples;

**public** **class** **ExampleBean** {

*// Number of years to calculate the Ultimate Answer*

**private** **int** years;

*// The Answer to Life, the Universe, and Everything*

**private** String ultimateAnswer;

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Constructor argument type matching

در سناریوی قبلی، اگر شما به صراحت نوع آرگومان سازنده را با استفاده از ویژگی type مشخص کنید، کانتینر می تواند از تطبیق نوع با انواع ساده استفاده کند. مثلا:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

Constructor argument index

از ویژگی index برای مشخص کردن صریح شاخص آرگومان های سازنده استفاده کنید. مثلا:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

علاوه بر حل ابهام چندین مقدار ساده، مشخص کردن یک شاخص ابهام را در جایی که سازنده دو آرگومان از یک نوع داشته باشد برطرف می کند. توجه داشته باشید که شاخص بر اساس 0 است.

Constructor argument name

همچنین می‌توانید از نام پارامتر سازنده برای ابهام‌زدایی مقدار استفاده کنید:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

به خاطر داشته باشید که برای انجام این کار خارج از جعبه، کد شما باید با فعال بودن پرچم اشکال زدایی کامپایل شود تا Spring بتواند نام پارامتر را از سازنده جستجو کند. اگر نمی‌توانید کد خود را با پرچم اشکال‌زدایی کامپایل کنید (یا نمی‌خواهید)، می‌توانید از حاشیه‌نویسی @ConstructorProperties JDK برای نام‌گذاری صریح آرگومان‌های سازنده خود استفاده کنید. سپس کلاس نمونه باید به صورت زیر باشد:

**package** examples;

**public** **class** **ExampleBean** {

*// Fields omitted*

@ConstructorProperties({"years", "ultimateAnswer"})

**public** ExampleBean(**int** years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

##### **Setter-based dependency injection**

DI مبتنی بر تنظیم توسط ظرفی که متدهای تنظیم کننده را روی دانه های شما فراخوانی می کند، پس از فراخوانی یک سازنده بدون آرگومان یا روش کارخانه ایستا بدون آرگومان برای نمونه سازی bean شما انجام می شود.

مثال زیر کلاسی را نشان می‌دهد که فقط با استفاده از تزریق ستتر خالص می‌توان به آن وابستگی تزریق کرد. این کلاس جاوای معمولی است. این یک POJO است که هیچ وابستگی به رابط های خاص کانتینر، کلاس های پایه یا حاشیه نویسی ندارد.

**public** **class** **SimpleMovieLister** {

*// the SimpleMovieLister has a dependency on the MovieFinder*

**private** MovieFinder movieFinder;

*// a setter method so that the Spring container can inject a MovieFinder*

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// business logic that actually uses the injected MovieFinder is omitted...*

}

ApplicationContext از DI مبتنی بر سازنده و مبتنی بر تنظیم کننده برای دانه هایی که مدیریت می کند پشتیبانی می کند. همچنین پس از تزریق برخی وابستگی ها از طریق رویکرد سازنده، از DI مبتنی بر تنظیم کننده پشتیبانی می کند. شما وابستگی ها را در قالب BeanDefinition پیکربندی می کنید، که در ارتباط با نمونه های PropertyEditor برای تبدیل ویژگی ها از یک فرمت به فرمت دیگر استفاده می کنید. با این حال، اکثر کاربران Spring مستقیماً با این کلاس‌ها کار نمی‌کنند (یعنی به صورت برنامه‌نویسی) بلکه با تعاریف XML Bean، اجزای حاشیه‌نویسی شده (به عنوان مثال، کلاس‌های حاشیه‌نویسی شده با @Component، @Controller، و غیره) یا روش‌های @Bean در جاوا کار می‌کنند. configuration@کلاس . سپس این منابع به صورت داخلی به نمونه هایی از BeanDefinition تبدیل می شوند و برای بارگیری کل نمونه کانتینر Spring IoC استفاده می شوند.

Constructor-based or setter-based DI?

از آنجایی که می‌توانید DI مبتنی بر سازنده و تنظیم‌کننده را با هم ترکیب کنید، استفاده از سازنده‌ها برای وابستگی‌های اجباری و روش‌های تنظیم‌کننده یا روش‌های پیکربندی برای وابستگی‌های اختیاری یک قانون خوب است. توجه داشته باشید که استفاده از حاشیه‌نویسی @Required در یک متد تنظیم کننده می‌تواند برای تبدیل ویژگی به یک وابستگی ضروری استفاده شود.

تیم Spring عموماً از تزریق سازنده حمایت می کند زیرا به فرد امکان می دهد اجزای برنامه را به عنوان اشیاء تغییرناپذیر پیاده سازی کند و اطمینان حاصل کند که وابستگی های مورد نیاز خالی نیستند. علاوه بر این، مؤلفه های تزریق شده توسط سازنده همیشه در یک حالت کاملاً اولیه به کد مشتری (در حال فراخوان) بازگردانده می شوند. به عنوان یک نکته جانبی، تعداد زیادی از آرگومان‌های سازنده بوی کد بدی دارند، به این معنی که کلاس احتمالاً مسئولیت‌های زیادی دارد و باید برای پرداختن به تفکیک صحیح نگرانی‌ها مجدداً اصلاح شود.

تزریق ستر در درجه اول باید فقط برای وابستگی های اختیاری استفاده شود که بتوان مقادیر پیش فرض معقولی را در کلاس به آنها اختصاص داد. در غیر این صورت، در هر جایی که کد از وابستگی استفاده می کند، باید بررسی های غیر تهی انجام شود. یکی از مزایای تزریق تنظیم کننده این است که روش های تنظیم کننده، اشیاء آن کلاس را برای پیکربندی مجدد یا تزریق مجدد بعداً آماده می کنند. بنابراین مدیریت از طریق JMX MBeans یک مورد استفاده قانع کننده برای تزریق ستر است.

از سبک DI استفاده کنید که برای یک کلاس خاص بیشترین معنا را دارد. گاهی اوقات، هنگام برخورد با کلاس های شخص ثالث که منبع آنها را ندارید، انتخاب برای شما انجام می شود. به عنوان مثال، اگر یک کلاس شخص ثالث هیچ روش تنظیم کننده را نشان ندهد، تزریق سازنده ممکن است تنها شکل موجود DI باشد.

##### **Dependency resolution process**

container تفکیک وابستگی لوبیا را به صورت زیر انجام می دهد:

ApplicationContext با فراداده های پیکربندی که همه دانه ها را توصیف می کند ایجاد و مقداردهی اولیه می شود. فراداده پیکربندی را می توان از طریق XML، کد جاوا یا حاشیه نویسی مشخص کرد.

برای هر bean، وابستگی‌های آن در قالب ویژگی‌ها، آرگومان‌های سازنده یا آرگومان‌هایی به روش static-factory بیان می‌شود، اگر از آن به‌جای یک سازنده معمولی استفاده می‌کنید. این وابستگی ها در زمانی که لوبیا واقعاً ایجاد می شود به Bean ارائه می شود.

هر ویژگی یا آرگومان سازنده یک تعریف واقعی از مقداری است که باید تنظیم شود، یا ارجاع به bean دیگری در ظرف است.

هر ویژگی یا آرگومان سازنده که یک مقدار است از قالب مشخص شده خود به نوع واقعی آن ویژگی یا آرگومان سازنده تبدیل می شود. به‌طور پیش‌فرض Spring می‌تواند یک مقدار ارائه‌شده در قالب رشته را به همه انواع داخلی مانند int، long، string، boolean و غیره تبدیل کند.

کانتینر Spring پیکربندی هر دانه را به هنگام ایجاد ظرف تأیید می کند. با این حال، خود خواص لوبیا تا زمانی که لوبیا واقعاً ایجاد نشود تنظیم نمی شود. لوبیاهایی که به صورت singleton هستند و به‌صورت پیش‌فرض تنظیم شده‌اند، هنگام ایجاد ظرف ایجاد می‌شوند. محدوده ها در محدوده های Bean تعریف شده اند. در غیر این صورت، لوبیا فقط در صورت درخواست ایجاد می شود. ایجاد یک bean به طور بالقوه باعث می شود که یک نمودار از bean ایجاد شود، زیرا وابستگی های bean و وابستگی های آن (و غیره) ایجاد و تخصیص داده می شوند. توجه داشته باشید که عدم تطابق وضوح بین آن وابستگی‌ها ممکن است دیر ظاهر شود، یعنی در اولین ایجاد bean تحت تأثیر.

Circular dependencies

اگر عمدتاً از تزریق سازنده استفاده می کنید، ممکن است یک سناریوی وابستگی دایره ای حل نشدنی ایجاد کنید.

به عنوان مثال: کلاس A به یک نمونه از کلاس B از طریق تزریق سازنده نیاز دارد، و کلاس B به نمونه ای از کلاس A از طریق تزریق سازنده نیاز دارد. اگر دانه‌ها را برای کلاس‌های A و B پیکربندی کنید تا به یکدیگر تزریق شوند، ظرف Spring IoC این مرجع دایره‌ای را در زمان اجرا شناسایی می‌کند و BeanCurrentlyInCreationException را پرتاب می‌کند.

یک راه حل ممکن این است که کد منبع برخی از کلاس ها را ویرایش کنید تا توسط تنظیم کننده ها به جای سازنده ها پیکربندی شوند. روش دیگر، از تزریق سازنده خودداری کنید و فقط از تزریق ستر استفاده کنید. به عبارت دیگر، اگرچه توصیه نمی شود، اما می توانید وابستگی های دایره ای را با تزریق ستر پیکربندی کنید.

برخلاف حالت معمولی (بدون وابستگی دایره‌ای)، وابستگی دایره‌ای بین لوبیا A و لوبیا B باعث می‌شود یکی از لوبیاها قبل از اینکه کاملاً مقداردهی شود به دیگری تزریق شود (یک سناریوی کلاسیک مرغ/تخم مرغ).

به طور کلی می توانید به بهار اعتماد کنید تا کار درست را انجام دهد. مشکلات پیکربندی، مانند ارجاع به دانه‌های موجود و وابستگی‌های دایره‌ای را در زمان بارگیری ظرف شناسایی می‌کند. اسپرینگ ویژگی ها را تنظیم می کند و وابستگی ها را تا جایی که ممکن است دیرتر برطرف می کند، زمانی که لوبیا واقعاً ایجاد می شود. این بدان معنی است که یک کانتینر Spring که به درستی بارگیری شده است، می‌تواند بعداً در صورت درخواست یک شی، در صورت وجود مشکل در ایجاد آن شی یا یکی از وابستگی‌های آن، استثنا ایجاد کند. به عنوان مثال، bean یک استثنا را در نتیجه یک ویژگی مفقود یا نامعتبر پرتاب می کند. این مشاهده بالقوه تأخیر در برخی از مسائل پیکربندی به همین دلیل است که پیاده سازی های ApplicationContext به طور پیش فرض از پیش نمونه های singleton bean می کنند. با صرف مقداری زمان و حافظه اولیه برای ایجاد این دانه‌ها قبل از نیاز واقعی، مشکلات پیکربندی را زمانی که ApplicationContext ایجاد می‌شود، کشف می‌کنید، نه بعداً. همچنان می‌توانید این رفتار پیش‌فرض را نادیده بگیرید تا لوبیاهای سینگلتون به جای اینکه از قبل آماده شوند، تنبلی اولیه شوند.

اگر هیچ وابستگی دایره‌ای وجود نداشته باشد، هنگامی که یک یا چند دانه همکار به یک دانه وابسته تزریق می‌شود، هر یک از دانه‌های همکار قبل از تزریق به دانه وابسته کاملاً پیکربندی می‌شود. این بدان معناست که اگر bean A وابستگی به bean B داشته باشد، محفظه Spring IoC قبل از فراخوانی روش تنظیم کننده در Bean A، به طور کامل bean B را پیکربندی می کند. وابستگی ها تنظیم می شوند و روش های چرخه حیات مربوطه (مانند روش init پیکربندی شده یا روش InitializingBean callback) فراخوانی می شوند.

##### Examples of dependency injection

مثال زیر از فراداده های پیکربندی مبتنی بر XML برای DI مبتنی بر تنظیم کننده استفاده می کند. بخش کوچکی از فایل پیکربندی Spring XML برخی از تعاریف bean را مشخص می کند:

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- setter injection using the nested ref element -->*

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

*<!-- setter injection using the neater ref attribute -->*

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** **void** setBeanOne(AnotherBean beanOne) {

this.beanOne = beanOne;

}

**public** **void** setBeanTwo(YetAnotherBean beanTwo) {

this.beanTwo = beanTwo;

}

**public** **void** setIntegerProperty(**int** i) {

this.i = i;

}

}

در مثال قبل، تنظیم کننده ها با ویژگی های مشخص شده در فایل XML مطابقت دارند. مثال زیر از DI مبتنی بر سازنده استفاده می کند:

<bean id="exampleBean" class="examples.ExampleBean">

*<!-- constructor injection using the nested ref element -->*

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

*<!-- constructor injection using the neater ref attribute -->*

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg type="int" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

**private** AnotherBean beanOne;

**private** YetAnotherBean beanTwo;

**private** **int** i;

**public** ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

آرگومان های سازنده مشخص شده در تعریف bean به عنوان آرگومان های سازنده ExampleBean استفاده خواهند شد.

حال یک نوع از این مثال را در نظر بگیرید، جایی که به جای استفاده از سازنده، به Spring گفته می شود که یک متد کارخانه ای استاتیک را فراخوانی کند تا نمونه ای از شی را برگرداند:

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

**public** **class** **ExampleBean** {

*// a private constructor*

**private** ExampleBean(...) {

...

}

*// a static factory method; the arguments to this method can be*

*// considered the dependencies of the bean that is returned,*

*// regardless of how those arguments are actually used.*

**public** **static** ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, **int** i) {

ExampleBean eb = **new** ExampleBean (...);

*// some other operations...*

**return** eb;

}

}

آرگومان‌های روش کارخانه ایستا از طریق عناصر <constructor-arg/> ارائه می‌شوند، دقیقاً مانند اینکه یک سازنده واقعاً استفاده شده باشد. نوع کلاسی که با روش کارخانه برگردانده می شود، لازم نیست از همان نوع کلاسی باشد که متد کارخانه ایستا را در بر می گیرد، اگرچه در این مثال چنین است. یک روش نمونه (غیر ایستا) کارخانه به شکلی اساساً یکسان استفاده می شود (به غیر از استفاده از ویژگی Factory-bean به جای ویژگی کلاس)، بنابراین جزئیات در اینجا مورد بحث قرار نمی گیرند.

### 1.4.2. Dependencies and configuration in detail

همانطور که در بخش قبل ذکر شد، می‌توانید ویژگی‌های bean و آرگومان‌های سازنده را به‌عنوان ارجاع به سایر bean‌های مدیریت‌شده (همکاران)، یا به‌عنوان مقادیر تعریف‌شده درون خطی تعریف کنید. ابرداده پیکربندی مبتنی بر XML Spring از انواع عناصر فرعی در عناصر <property/> و <constructor-arg/> برای این منظور پشتیبانی می کند.

#### Straight values (primitives, Strings, and so on)

ویژگی مقدار عنصر <property/> یک ویژگی یا آرگومان سازنده را به عنوان نمایش رشته قابل خواندن توسط انسان مشخص می کند. سرویس تبدیل Spring برای تبدیل این مقادیر از یک رشته به نوع واقعی ویژگی یا آرگومان استفاده می شود.

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

*<!-- results in a setDriverClassName(String) call -->*

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

مثال زیر از فضای نام p برای پیکربندی XML حتی مختصرتر استفاده می کند.

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

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

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

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

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

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

XML قبلی مختصرتر است. با این حال، اشتباهات تایپی در زمان اجرا کشف می شوند تا زمان طراحی، مگر اینکه از یک IDE مانند IntelliJ IDEA یا Spring Tool Suite (STS) استفاده کنید که از تکمیل خودکار ویژگی ها هنگام ایجاد تعاریف bean پشتیبانی می کند. چنین کمک های IDE به شدت توصیه می شود.

همچنین می توانید یک نمونه java.util.Properties را به صورت زیر پیکربندی کنید:

<bean id="mappings"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

*<!-- typed as a java.util.Properties -->*

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

</bean>

ظرف Spring متن داخل عنصر <value/> را با استفاده از مکانیسم JavaBeans PropertyEditor به یک نمونه java.util.Properties تبدیل می کند. این یک میانبر خوب است و یکی از معدود جاهایی است که تیم Spring استفاده از عنصر تودرتوی <value/> را بر سبک ویژگی ارزش ترجیح می دهد.

The idref element

عنصر idref به سادگی یک روش ضد خطا برای ارسال id (مقدار رشته - نه یک مرجع) یک bean دیگر در ظرف به عنصر <constructor-arg/> یا <property/> است.

<bean id="theTargetBean" class="..."/>

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

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

قطعه تعریف bean بالا دقیقاً معادل قطعه زیر (در زمان اجرا) است:

<bean id="theTargetBean" class="..." />

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

<property name="targetName" value="theTargetBean"/>

</bean>

شکل اول بر دومی ارجحیت دارد، زیرا استفاده از تگ idref به کانتینر اجازه می دهد تا در زمان استقرار اعتبار سنجی کند که bean ارجاع شده با نام واقعاً وجود دارد. در تغییر دوم، هیچ اعتبارسنجی روی مقداری که به ویژگی targetName از کلاینت bean ارسال می‌شود، انجام نمی‌شود. اشتباهات تایپی تنها زمانی کشف می شوند (با به احتمال زیاد نتایج کشنده) که کلاینت bean واقعاً نمونه سازی شود. اگر کلاینت bean یک نمونه اولیه باشد، این اشتباه تایپی و استثنای ناشی از آن ممکن است مدت‌ها پس از استقرار کانتینر کشف شود.

ویژگی محلی در عنصر idref دیگر در 4.0 beans xsd پشتیبانی نمی‌شود، زیرا دیگر مقداری نسبت به یک مرجع bean معمولی ارائه نمی‌کند. به سادگی هنگام ارتقاء به طرح 4.0، مراجع محلی idref موجود خود را به idref bean تغییر دهید.

یک مکان رایج (حداقل در نسخه‌های زودتر از Spring 2.0) که عنصر <idref/> ارزش می‌آورد، در پیکربندی رهگیرهای AOP در تعریف Bean ProxyFactoryBean است. استفاده از عناصر <idref/> زمانی که نام رهگیر را مشخص می‌کنید، از املای اشتباه شناسه رهگیر جلوگیری می‌کند.

عنصر ref عنصر نهایی درون عنصر تعریف <constructor-arg/> یا <property/> است. در اینجا مقدار خاصیت مشخص شده یک bean را به عنوان مرجعی به bean دیگری (یک همکار) که توسط ظرف مدیریت می شود تنظیم می کنید. bean ارجاع شده یک وابستگی به bean است که ویژگی آن تنظیم می شود و در صورت نیاز قبل از تنظیم ویژگی مقداردهی اولیه می شود. (اگر همکار یک لوبیای تکی باشد، ممکن است قبلاً توسط ظرف مقداردهی اولیه شده باشد.) همه ارجاعات در نهایت ارجاع به شی دیگری هستند. محدوده و اعتبارسنجی به این بستگی دارد که آیا شما شناسه/نام شی دیگر را از طریق صفات bean، local یا والد مشخص کنید.

تعیین bean هدف از طریق ویژگی bean تگ <ref/> عمومی ترین شکل است و اجازه می دهد تا یک مرجع به هر bean در همان ظرف یا ظرف اصلی ایجاد شود، صرف نظر از اینکه در همان فایل XML باشد یا خیر. مقدار مشخصه bean ممکن است با ویژگی id bean هدف یا به عنوان یکی از مقادیر در ویژگی name bean هدف یکسان باشد.

<ref bean="someBean"/>

مشخص کردن bean هدف از طریق ویژگی والد، ارجاعی به bean ایجاد می‌کند که در ظرف والد ظرف فعلی است. مقدار خصیصه والد ممکن است با ویژگی id دانه هدف، یا یکی از مقادیر در ویژگی نام دانه هدف یکسان باشد و دانه هدف باید در ظرف والد موجود باشد. شما از این نوع مرجع bean عمدتاً زمانی استفاده می‌کنید که سلسله مراتبی از کانتینرها دارید و می‌خواهید یک bean موجود را در یک ظرف والد با یک پروکسی بپیچید که همان نام باقلا اصلی است.

*<!-- in the parent context -->*

<bean id="accountService" class="com.foo.SimpleAccountService">

*<!-- insert dependencies as required as here -->*

</bean>

*<!-- in the child (descendant) context -->*

<bean id="accountService" <!-- bean name is the same as the parent bean -->

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> *<!-- notice how we refer to the parent bean -->*

</property>

*<!-- insert other configuration and dependencies as required here -->*

</bean>

#### Inner beans

یک عنصر <bean/> در داخل عناصر <property/> یا <constructor-arg/> یک به اصطلاح inner bean را تعریف می کند.

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

*<!-- instead of using a reference to a target bean, simply define the target bean inline -->*

<property name="target">

<bean class="com.example.Person"> *<!-- this is the inner bean -->*

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

#### Collections

در عناصر <list/>، <set/>، <map/> و <props/>، به ترتیب ویژگی‌ها و آرگومان‌های فهرست، مجموعه، نقشه و ویژگی‌های مجموعه جاوا را تنظیم می‌کنید.

<bean id="moreComplexObject" class="example.ComplexObject">

*<!-- results in a setAdminEmails(java.util.Properties) call -->*

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

*<!-- results in a setSomeList(java.util.List) call -->*

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

*<!-- results in a setSomeMap(java.util.Map) call -->*

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

*<!-- results in a setSomeSet(java.util.Set) call -->*

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

مقدار یک کلید نقشه یا مقدار، یا یک مقدار مجموعه نیز می‌تواند یکی از عناصر زیر باشد:

bean | ref | idref | list | set | map | props | value | null

###### **Collection merging**

ظرف Spring همچنین از ادغام مجموعه ها پشتیبانی می کند. یک توسعه‌دهنده برنامه می‌تواند یک عنصر <list/>، <map/>، <set/> یا <props/> به سبک پدر تعریف کند و <list/>، <map/>، <set/> یا به سبک فرزند داشته باشد. عناصر <props/> مقادیر را از مجموعه والد به ارث می برند و نادیده می گیرند. یعنی مقادیر مجموعه فرزند نتیجه ادغام عناصر مجموعه والد و فرزند است، با عناصر مجموعه فرزند بر مقادیر تعیین شده در مجموعه والد.

این بخش در مورد ادغام مکانیسم والد-فرزند لوبیا را مورد بحث قرار می دهد. خوانندگانی که با تعاریف والد و بچه لوبیا آشنا نیستند ممکن است بخواهند قبل از ادامه بخش مربوطه را بخوانند.

مثال زیر ادغام مجموعه را نشان می دهد:

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

*<!-- the merge is specified on the child collection definition -->*

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

به استفاده از ویژگی merge=true در عنصر <props/> از ویژگی adminEmails در تعریف child bean توجه کنید. هنگامی که کودک bean توسط کانتینر حل و فصل شد، نمونه به دست آمده دارای یک مجموعه AdminEmails Properties است که حاوی نتیجه ادغام مجموعه ایمیل های مدیر فرزند با مجموعه ایمیل های admin والدین است.

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

مجموعه ارزش مجموعه ویژگی های فرزند همه عناصر ویژگی را از والد <props/> به ارث می برد و مقدار فرزند برای مقدار پشتیبانی بر مقدار مجموعه والد غلبه می کند.

این رفتار ادغام به طور مشابه برای انواع مجموعه <list/>، <map/> و <set/> اعمال می شود. در مورد خاص عنصر <list/>، معنایی مرتبط با نوع مجموعه List، یعنی مفهوم مجموعه مرتب شده از مقادیر، حفظ می شود. مقادیر والدین مقدم بر همه مقادیر فهرست فرزند هستند. در مورد انواع مجموعه Map، Set و Properties، هیچ ترتیبی وجود ندارد. بنابراین هیچ معنایی ترتیبی برای انواع مجموعه‌ای که زیربنای انواع پیاده‌سازی Map، Set و Properties مرتبط هستند که کانتینر به صورت داخلی استفاده می‌کند، وجود ندارد.

##### Limitations of collection merging

شما نمی توانید انواع مختلف مجموعه (مانند نقشه و فهرست) را ادغام کنید، و اگر سعی کنید این کار را انجام دهید، یک استثنای مناسب ایجاد می شود. ویژگی ادغام باید در تعریف فرزند پایین تر، ارثی مشخص شود. تعیین ویژگی ادغام در تعریف مجموعه والد اضافی است و منجر به ادغام مورد نظر نخواهد شد.

##### Strongly-typed collection

با معرفی انواع عمومی در جاوا 5، می توانید از مجموعه های تایپ شده قوی استفاده کنید. یعنی می توان یک نوع Collection را طوری تعریف کرد که فقط شامل عناصر String باشد (مثلا). اگر از Spring برای تزریق وابستگی یک مجموعه قوی تایپ شده به یک لوبیا استفاده می‌کنید، می‌توانید از پشتیبانی تبدیل نوع Spring به گونه‌ای استفاده کنید که عناصر نمونه‌های مجموعه قوی تایپ شده شما قبل از اضافه شدن به نوع مناسب تبدیل شوند. مجموعه.

**public** **class** **Foo** {

**private** Map<String, Float> accounts;

**public** **void** setAccounts(Map<String, Float> accounts) {

this.accounts = accounts;

}

}

<beans>

<bean id="foo" class="x.y.Foo">

<property name="accounts">

<map>

<entry key="one" value="9.99"/>

<entry key="two" value="2.75"/>

<entry key="six" value="3.99"/>

</map>

</property>

</bean>

</beans>

هنگامی که ویژگی حساب های foo bean برای تزریق آماده می شود، اطلاعات عمومی در مورد نوع عنصر نقشه قوی تایپ شده <String, Float> با بازتاب در دسترس است. بنابراین زیرساخت تبدیل نوع Spring، عناصر ارزش مختلف را از نوع Float تشخیص می‌دهد و مقادیر رشته‌ای 9.99، 2.75 و 3.99 به یک نوع Float واقعی تبدیل می‌شوند.

#### Null and empty string values

Spring آرگومان های خالی برای خواص و موارد مشابه را به عنوان رشته های خالی در نظر می گیرد. قطعه فراداده پیکربندی مبتنی بر XML زیر، ویژگی ایمیل را روی مقدار خالی رشته ("") تنظیم می کند.

<bean class="ExampleBean">

<property name="email" value=""/>

</bean>

مثال قبلی معادل کد جاوا زیر است:

exampleBean.setEmail("");

عنصر <null/> مقادیر null را مدیریت می کند. مثلا:

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

پیکربندی فوق معادل کد جاوا زیر است:

exampleBean.setEmail(null);

#### XML shortcut with the p-namespace

فضای نام p به شما این امکان را می دهد تا از ویژگی های عنصر bean به جای عناصر <property/> تودرتو، برای توصیف مقادیر ویژگی و/یا bean های همکار خود استفاده کنید.

Spring از فرمت های پیکربندی توسعه پذیر با فضاهای نام پشتیبانی می کند که بر اساس تعریف طرحواره XML هستند. قالب پیکربندی beans که در این فصل مورد بحث قرار گرفت در یک سند XML Schema تعریف شده است. با این حال، فضای نام p در یک فایل XSD تعریف نشده است و فقط در هسته Spring وجود دارد.

مثال زیر دو قطعه XML را نشان می دهد که نتیجه یکسانی دارند: اولی از فرمت استاندارد XML و دومی از فضای نام p استفاده می کند.

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

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

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

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

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

<bean name="classic" class="com.example.ExampleBean">

<property name="email" value="foo@bar.com"/>

</bean>

<bean name="p-namespace" class="com.example.ExampleBean"

p:email="foo@bar.com"/>

</beans>

مثال یک ویژگی را در فضای نام p به نام ایمیل در تعریف bean نشان می دهد. این به Spring می گوید که یک اظهارنامه دارایی را شامل شود. همانطور که قبلا ذکر شد، فضای نام p تعریف طرحواره ندارد، بنابراین می توانید نام ویژگی را به نام ویژگی تنظیم کنید.

این مثال بعدی شامل دو تعریف دیگر از لوبیا است که هر دو به یک لوبیا دیگر اشاره دارند:

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

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

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

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

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

<bean name="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

همانطور که می بینید، این مثال نه تنها شامل یک مقدار ویژگی با استفاده از فضای نام p است، بلکه از یک فرمت خاص برای اعلام ارجاعات ویژگی استفاده می کند. در حالی که اولین تعریف bean از <property name="spouse" ref="jane"/> برای ایجاد یک مرجع از bean john به bean jane استفاده می کند، تعریف دوم bean از p:spouse-ref="jane" به عنوان یک ویژگی برای انجام استفاده می کند. دقیقا همان چیزی در این مورد spouse نام دارایی است، در حالی که قسمت -ref نشان می دهد که این یک مقدار مستقیم نیست بلکه ارجاع به bean دیگری است.

فضای نام p به اندازه فرمت استاندارد XML انعطاف پذیر نیست. به عنوان مثال، فرمت برای اعلان ارجاعات دارایی با ویژگی هایی که به Ref ختم می شوند در تضاد است، در حالی که فرمت استاندارد XML اینطور نیست. توصیه می کنیم رویکرد خود را با دقت انتخاب کنید و این موضوع را به اعضای تیم خود در میان بگذارید تا از تولید اسناد XML که از هر سه رویکرد به طور همزمان استفاده می کنند اجتناب کنید.

#### XML shortcut with the c-namespace

مشابه میانبر XML با فضای نام p، فضای نام c، که به تازگی در Spring 3.1 معرفی شده است، اجازه می دهد تا از ویژگی های خطی برای پیکربندی آرگومان های سازنده به جای عناصر سازنده-آرگ تو در تو استفاده شود.

بیایید نمونه‌های تزریق وابستگی مبتنی بر سازنده را با فضای نام c: مرور کنیم:

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

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

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

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

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

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

*<!-- traditional declaration -->*

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

*<!-- c-namespace declaration -->*

<bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com"/>

</beans>

فضای نام c: از همان قراردادهایی مانند p: one (تریل -ref برای ارجاعات bean) برای تنظیم آرگومان های سازنده با نام آنها استفاده می کند. و به همان اندازه، باید اعلام شود، حتی اگر در یک طرح XSD تعریف نشده باشد (اما در هسته Spring وجود دارد).

برای موارد نادری که نام آرگومان سازنده در دسترس نیست (معمولاً اگر بایت کد بدون اطلاعات اشکال زدایی کامپایل شده باشد)، می توان از بازگشت به نمایه های آرگومان استفاده کرد:

*<!-- c-namespace index declaration -->*

<bean id="foo" class="x.y.Foo" c:\_0-ref="bar" c:\_1-ref="baz"/>

با توجه به دستور زبان XML، نماد شاخص نیاز به حضور پیشرو دارد.

در عمل، مکانیسم تفکیک سازنده در تطبیق آرگومان‌ها کاملاً کارآمد است، بنابراین، مگر اینکه واقعاً نیاز باشد، توصیه می‌کنیم از نماد نام در سراسر پیکربندی خود استفاده کنید.

#### Compound property names

زمانی که ویژگی های bean را تنظیم می کنید، می توانید از نام های ترکیبی یا تو در تو استفاده کنید، تا زمانی که تمام اجزای مسیر به جز نام ویژگی نهایی تهی نباشند. تعریف لوبیا زیر را در نظر بگیرید.

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

foo bean یک خاصیت fred دارد که دارای یک خاصیت bob است که دارای یک ویژگی sammy است و آن خاصیت sammy نهایی روی مقدار 123 تنظیم می شود. برای اینکه این ویژگی کار کند، ویژگی fred foo و خاصیت bob fred نباید بعد از ساخت bean، یا NullPointerException خالی باشد.

### 1.4.3. Using depends-on

اگر یک لوبیا وابستگی به دیگری باشد، معمولاً به این معنی است که یک دانه به عنوان ویژگی دیگری تنظیم می شود. معمولاً این کار را با عنصر <ref/> در فراداده پیکربندی مبتنی بر XML انجام می دهید. با این حال، گاهی اوقات وابستگی بین لوبیا کمتر مستقیم است. برای مثال، یک شروع کننده استاتیک در یک کلاس باید راه اندازی شود، مانند ثبت نام درایور پایگاه داده. مشخصه بستگی به صراحتاً می تواند یک یا چند دانه را مجبور کند قبل از اینکه bean با استفاده از این عنصر مقداردهی اولیه شود. مثال زیر از صفت بستگی-on برای بیان وابستگی به یک دانه استفاده می کند:

<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />

برای بیان وابستگی به چند دانه، فهرستی از نام‌های لوبیا را به‌عنوان مقدار ویژگی وابسته به همراه با کاما، فاصله سفید و نیم‌ویرگول که به‌عنوان جداکننده معتبر استفاده می‌شوند، ارائه کنید:

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

<bean id="manager" class="ManagerBean" />

<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />

ویژگی وابسته در تعریف bean می‌تواند هم یک وابستگی زمان اولیه و هم فقط در مورد دانه‌های تک تن، یک وابستگی زمان تخریب متناظر را مشخص کند. لوبیاهای وابسته که یک رابطه وابسته را با یک دانه معین تعریف می‌کنند، ابتدا از بین می‌روند، قبل از اینکه خود لوبیا معین از بین برود. بنابراین بستگی به می تواند ترتیب خاموش شدن را نیز کنترل کند.

### 1.4.4. Lazy-initialized beans

به‌طور پیش‌فرض، پیاده‌سازی‌های ApplicationContext مشتاقانه همه دانه‌های singleton را به عنوان بخشی از فرآیند اولیه‌سازی ایجاد و پیکربندی می‌کنند. به طور کلی، این پیش نمونه سازی مطلوب است، زیرا خطاها در پیکربندی یا محیط اطراف، بر خلاف ساعت ها یا حتی روزها بعد، بلافاصله کشف می شوند. هنگامی که این رفتار مطلوب نیست، می‌توانید با علامت‌گذاری تعریف لوبیا به‌عنوان lazy-initialized از پیش‌نمونه‌سازی لوبیا تک‌تنه جلوگیری کنید. یک bean با مقدار اولیه تنبل به ظرف IoC می گوید که در اولین درخواست، به جای راه اندازی، یک نمونه bean ایجاد کند.

در XML، این رفتار توسط ویژگی lazy-init در عنصر <bean/> کنترل می شود. مثلا:

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

هنگامی که پیکربندی قبلی توسط یک ApplicationContext مصرف می‌شود، زمانی که ApplicationContext در حال راه‌اندازی است، bean با نام lazy مشتاقانه از پیش نمونه‌سازی نمی‌شود، در حالی که bean not.lazy مشتاقانه از قبل نمونه‌سازی می‌شود.

با این حال، زمانی که یک لوبیای تنبل اولیه، وابستگی به یک لوبیای تک تنه است که با مقدار اولیه تنبلی انجام نمی‌شود، ApplicationContext در راه‌اندازی، لوبیا اولیه‌سازی شده تنبل را ایجاد می‌کند، زیرا باید وابستگی‌های تک تن را برآورده کند. لوبیای تنبل اولیه به لوبیای تک تنی در جای دیگری تزریق می شود که تنبلی اولیه نشده است.

همچنین می‌توانید با استفاده از ویژگی پیش‌فرض-lazy-init در عنصر <beans/>، راه‌اندازی تنبل را در سطح ظرف کنترل کنید. مثلا:

<beans default-lazy-init="true">

*<!-- no beans will be pre-instantiated... -->*

</beans>

### 1.4.5. Autowiring collaborators

ظرف Spring می تواند روابط بین دانه های همکار را خودکار کند. می توانید به Spring اجازه دهید تا با بررسی محتوای ApplicationContext، همکاران (دیگر bean) را به طور خودکار برای bean خود حل کند. سیم کشی خودکار دارای مزایای زیر است:

سیم کشی خودکار می تواند نیاز به مشخص کردن ویژگی ها یا آرگومان های سازنده را به میزان قابل توجهی کاهش دهد. (مکانیسم‌های دیگری مانند الگوی لوبیا که در جای دیگر این فصل مورد بحث قرار گرفته‌اند نیز در این زمینه ارزشمند هستند.)

سیم‌کشی خودکار می‌تواند پیکربندی را با تکامل اشیاء شما به‌روزرسانی کند. به عنوان مثال، اگر شما نیاز به اضافه کردن یک وابستگی به یک کلاس دارید، آن وابستگی می تواند به طور خودکار برآورده شود بدون اینکه نیازی به تغییر پیکربندی باشد. بنابراین سیم‌کشی خودکار می‌تواند به ویژه در طول توسعه مفید باشد، بدون اینکه گزینه تغییر به سیم‌کشی صریح در زمانی که پایه کد پایدارتر می‌شود را رد کند.

هنگام استفاده از فراداده پیکربندی مبتنی بر XML [2]، حالت autowire را برای تعریف bean با ویژگی autowire عنصر <bean/> مشخص می‌کنید. قابلیت سیم کشی خودکار دارای چهار حالت است. سیم‌کشی خودکار به ازای هر لوبیا را مشخص می‌کنید و بنابراین می‌توانید انتخاب کنید که کدام یک را سیم‌کشی خودکار کنید.

| *Table 2. Autowiring modes* | |
| --- | --- |
| **Mode** | **Explanation** |
| no | (Default) No autowiring. Bean references must be defined via a ref element. Changing the  default  setting is not recommended for larger deployments, because specifying collaborators explicitly  gives greater control and clarity. To some extent, it documents the structure of a system. |
| byName | Autowiring by property name. Spring looks for a bean with the same name as the property that  needs to be autowired. For example, if a bean definition is set to autowire by name, and it  contains  a master property (that is, it has a setMaster(..) method), Spring looks for a bean definition  named  master, and uses it to set the property. |
| byType | Allows a property to be autowired if exactly one bean of the property type exists in the  container.  If more than one exists, a fatal exception is thrown, which indicates that you may not use  byType autowiring for that bean. If there are no matching beans, nothing happens; the property  is not set. |
| constructor | Analogous to byType, but applies to constructor arguments. If there is not exactly  one bean of the  constructor argument type in the container, a fatal error is raised. |

با byType یا حالت سیم کشی خودکار سازنده، می توانید آرایه ها و مجموعه های تایپ شده را سیم کشی کنید. در چنین مواردی همه نامزدهای autowire در ظرف که با نوع مورد انتظار مطابقت دارند برای ارضای وابستگی ارائه می شوند. اگر نوع کلید مورد انتظار String باشد، می‌توانید Maps با تایپ قوی را به صورت خودکار سیم‌کشی کنید. مقادیر Maps سیم‌کشی شده خودکار شامل تمام نمونه‌های bean خواهد بود که با نوع مورد انتظار مطابقت دارند و کلیدهای Maps حاوی نام‌های bean مربوطه خواهند بود.

می‌توانید رفتار سیم‌کشی خودکار را با بررسی وابستگی، که پس از تکمیل سیم‌کشی خودکار انجام می‌شود، ترکیب کنید.

##### Limitations and disadvantages of autowiring

سیم‌کشی خودکار زمانی بهترین کار را انجام می‌دهد که به طور مداوم در یک پروژه استفاده شود. اگر سیم‌کشی خودکار به طور کلی استفاده نمی‌شود، ممکن است برای توسعه‌دهندگان استفاده از آن برای سیم‌کشی تنها یک یا دو تعریف لوبیا گیج‌کننده باشد.

محدودیت ها و معایب سیم کشی خودکار را در نظر بگیرید:

وابستگی های صریح در تنظیمات ویژگی و constructor-arg همیشه سیم کشی خودکار را لغو می کند. شما نمی توانید به اصطلاح خصوصیات ساده مانند primitives، string ها و class ها (و آرایه هایی از این ویژگی های ساده) را به صورت خودکار سیم کشی کنید. این محدودیت توسط طراحی است.

سیم کشی خودکار نسبت به سیم کشی صریح دقیق تر است. اگرچه همانطور که در جدول بالا ذکر شد، Spring مراقب است در صورت وجود ابهامی که ممکن است نتایج غیرمنتظره ای داشته باشد، از حدس زدن خودداری کند، روابط بین اشیاء مدیریت شده توسط Spring شما دیگر به صراحت مستند نمی شود.

اطلاعات سیم‌کشی ممکن است برای ابزارهایی که ممکن است اسنادی را از ظرف Spring ایجاد کنند در دسترس نباشد.

تعاریف چندگانه bean در ظرف ممکن است با نوع مشخص شده توسط روش تنظیم کننده یا آرگومان سازنده برای سیم کشی خودکار مطابقت داشته باشد. برای آرایه ها، مجموعه ها یا نقشه ها، این لزوماً یک مشکل نیست. با این حال، برای وابستگی هایی که انتظار یک مقدار واحد دارند، این ابهام خودسرانه حل نمی شود. اگر هیچ تعریف منحصر به فردی در دسترس نباشد، یک استثنا ایجاد می شود.

در سناریوی دوم، چندین گزینه دارید:

سیم کشی خودکار را به نفع سیم کشی صریح کنار بگذارید.

با تنظیم ویژگی‌های autowire-candidate آن بر روی false همانطور که در بخش بعدی توضیح داده شد، از سیم‌کشی خودکار برای تعریف bean اجتناب کنید.

با تنظیم ویژگی اولیه عنصر <bean/> آن بر روی true، یک تعریف bean را به عنوان نامزد اصلی تعیین کنید.

همانطور که در پیکربندی کانتینر مبتنی بر حاشیه‌نویسی توضیح داده شده است، کنترل دقیق‌تر موجود با پیکربندی مبتنی بر حاشیه‌نویسی را اجرا کنید.

##### Excluding a bean from autowiring

بر اساس هر لوبیا، می توانید یک دانه را از سیم کشی خودکار حذف کنید. در قالب XML Spring، ویژگی autowire-candidate عنصر <bean/> را روی false تنظیم کنید. ظرف باعث می‌شود که تعریف bean خاص برای زیرساخت سیم‌کشی خودکار (از جمله تنظیمات سبک حاشیه‌نویسی مانند @Autowired) در دسترس نباشد.

ویژگی autowire-candidate طوری طراحی شده است که فقط بر سیم کشی خودکار مبتنی بر نوع تأثیر بگذارد. این بر ارجاعات صریح با نام تأثیر نمی گذارد، حتی اگر bean مشخص شده به عنوان نامزد خودکار علامت گذاری نشده باشد، حل می شود. در نتیجه، سیم‌کشی خودکار با نام با این وجود، اگر نام مطابقت داشته باشد، یک دانه تزریق می‌کند.

همچنین می‌توانید نامزدهای خودکار را بر اساس تطبیق الگو با نام‌های لوبیا محدود کنید. عنصر سطح بالای <beans/> یک یا چند الگو را در ویژگی default-autowire-candidates خود می پذیرد. به عنوان مثال، برای محدود کردن وضعیت نامزدی autowire به هر Bean که نام آن به Repository ختم می‌شود، مقدار \*Repository را ارائه کنید. برای ارائه الگوهای متعدد، آنها را در یک لیست جدا شده با کاما تعریف کنید. مقدار صریح true یا false برای مشخصه autowire-candidate تعریف bean همیشه اولویت دارد و برای چنین دانه هایی قوانین تطبیق الگو اعمال نمی شود.

این تکنیک ها برای beans مفید هستند که هرگز نمی خواهید با autowiring به دانه های دیگر تزریق شوند. این بدان معنا نیست که یک bean حذف شده خود را نمی توان با استفاده از سیم کشی خودکار پیکربندی کرد. بلکه خود bean کاندیدای autowire سایر bean ها نیست.

#### 1.4.6. Method injection

در بیشتر سناریوهای کاربردی، بیشتر حبوبات موجود در ظرف singleton هستند. زمانی که یک لوبیا singleton نیاز به همکاری با لوبیا singleton دیگری دارد، یا یک لوبیا غیر singleton نیاز به همکاری با لوبیای singleton دیگری دارد، شما معمولاً با تعریف یک bean به عنوان ویژگی دیگری، وابستگی را مدیریت می کنید. زمانی که چرخه‌های عمر bean متفاوت است، مشکل ایجاد می‌شود. فرض کنید لوبیای تک تن A نیاز به استفاده از bean غیر singleton (نمونه اولیه) B دارد، شاید در هر روش فراخوانی روی A. ظرف فقط یک بار لوبیای singleton A را ایجاد می کند، و بنابراین فقط یک فرصت برای تنظیم ویژگی ها به دست می آورد. container نمی تواند هر بار که به bean A نیاز است نمونه جدیدی از لوبیا B را ارائه کند.

راه حل این است که از برخی وارونگی کنترل چشم پوشی کنید. می‌توانید با پیاده‌سازی رابط ApplicationContextAware، Bean A را از ظرف آگاه کنید، و با برقراری تماس getBean("B") با ظرف، هر زمانی که Bean A به آن نیاز دارد، یک نمونه Bean B (معمولاً جدید) بخواهید. در زیر نمونه ای از این رویکرد آورده شده است:

*// a class that uses a stateful Command-style class to perform some processing*

**package** fiona.apple;

*// Spring-API imports*

**import** org.springframework.beans.BeansException;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.ApplicationContextAware;

**public** **class** **CommandManager** **implements** ApplicationContextAware {

**private** ApplicationContext applicationContext;

**public** Object process(Map commandState) {

*// grab a new instance of the appropriate Command*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

**protected** Command createCommand() {

*// notice the Spring API dependency!*

**return** this.applicationContext.getBean("command", Command.class);

}

**public** **void** setApplicationContext(

ApplicationContext applicationContext) **throws** BeansException {

this.applicationContext = applicationContext;

}

}

مورد قبلی مطلوب نیست، زیرا کد کسب و کار از چارچوب Spring آگاه است و با آن همراه است. Method Injection، یک ویژگی تا حدی پیشرفته از کانتینر Spring IoC، اجازه می دهد تا این مورد استفاده به روشی تمیز انجام شود.

##### Lookup method injection

تزریق روش جستجو، توانایی کانتینر برای نادیده گرفتن روش‌ها بر روی دانه‌های مدیریت‌شده کانتینر، برای برگرداندن نتیجه جستجو برای دانه‌ای با نام دیگر در ظرف است. جستجو معمولاً شامل یک نمونه اولیه bean مانند سناریویی است که در بخش قبل توضیح داده شد. Spring Framework این تزریق روش را با استفاده از تولید بایت کد از کتابخانه CGLIB برای تولید پویا یک زیر کلاس که روش را لغو می کند، پیاده سازی می کند.

برای اینکه این زیر کلاس پویا کار کند، کلاسی که ظرف Spring bean آن را زیر کلاس می‌گذارد نمی‌تواند نهایی باشد و روشی که باید لغو شود نیز نمی‌تواند نهایی باشد.

تست واحد کلاسی که دارای متد انتزاعی است، مستلزم آن است که خودتان کلاس را زیر کلاس قرار دهید و یک پیاده‌سازی خرد از متد انتزاعی ارائه دهید.

روش‌های بتن نیز برای اسکن اجزا که نیاز به کلاس‌های بتنی دارد، ضروری هستند.

یک محدودیت کلیدی دیگر این است که متدهای جستجو با متدهای کارخانه و به ویژه با متدهای @Bean در کلاس‌های پیکربندی کار نمی‌کنند، زیرا کانتینر مسئول ایجاد نمونه در آن حالت نیست و بنابراین نمی‌تواند یک زیر کلاس تولید شده توسط زمان اجرا ایجاد کند. در پرواز.

**package** fiona.apple;

*// no more Spring imports!*

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

در کلاس کلاینت حاوی متدی که باید تزریق شود (در این مورد CommandManager)، متدی که باید تزریق شود نیاز به امضای فرم زیر دارد:

<public|protected> [abstract] <return-type> theMethodName(no-arguments);

اگر روش انتزاعی باشد، زیر کلاسی که به صورت پویا ایجاد شده است، روش را پیاده سازی می کند. در غیر این صورت، زیر کلاسی که به صورت پویا ایجاد می شود، روش مشخص تعریف شده در کلاس اصلی را لغو می کند. مثلا:

*<!-- a stateful bean deployed as a prototype (non-singleton) -->*

<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">

*<!-- inject dependencies here as required -->*

</bean>

*<!-- commandProcessor uses statefulCommandHelper -->*

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="myCommand"/>

</bean>

Bean شناسایی شده به عنوان commandManager هر زمان که به یک نمونه جدید از bean myCommand نیاز داشته باشد، متد خود را ()createCommand فرا می خواند. شما باید مراقب باشید که myCommand bean را به عنوان یک نمونه اولیه اجرا کنید، اگر واقعاً آن چیزی است که مورد نیاز است. اگر به صورت singleton باشد، هر بار همان نمونه از bean myCommand برگردانده می شود.

از طرف دیگر، در مدل مؤلفه مبتنی بر حاشیه‌نویسی، می‌توانید یک روش جستجو را از طریق حاشیه‌نویسی @Lookup اعلام کنید:

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

Command command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup("myCommand")

**protected** **abstract** Command createCommand();

}

یا، به صورت اصطلاحی تر، ممکن است به حل شدن bean هدف در برابر نوع بازگشت اعلام شده از روش جستجو تکیه کنید:

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

MyCommand command = createCommand();

command.setState(commandState);

**return** command.execute();

}

@Lookup

**protected** **abstract** MyCommand createCommand();

}

توجه داشته باشید که معمولاً چنین روش‌های جستجوی مشروح را با اجرای خرد بتن اعلام می‌کنید تا با قوانین اسکن مؤلفه Spring که در آن کلاس‌های انتزاعی به‌طور پیش‌فرض نادیده گرفته می‌شوند، سازگار باشند. این محدودیت در مورد طبقات لوبیا ثبت شده صریح یا صریحاً وارد شده اعمال نمی شود.

#### Arbitrary method replacement

یک شکل کمتر مفید از method injection نسبت lookup method injection ، توانایی جایگزینی روش‌های دلخواه در یک bean مدیریت‌شده با پیاده‌سازی method دیگری است. کاربران می توانند با خیال راحت از ادامه این بخش صرف نظر کنند تا زمانی که عملکرد واقعا مورد نیاز باشد.

با فراداده پیکربندی مبتنی بر XML، می‌توانید از عنصر روش جایگزین برای جایگزینی یک روش موجود با روش دیگری برای یک Bean استفاده کنید. کلاس زیر را با متد computeValue در نظر بگیرید که می‌خواهیم آن را لغو کنیم:

**public** **class** **MyValueCalculator** {

**public** String computeValue(String input) {

*// some real code...*

}

*// some other methods...*

}

کلاسی که رابط org.springframework.beans.factory.support.MethodReplacer را پیاده سازی می کند، تعریف method جدید را ارائه می دهد.

*/\*\**

*\* meant to be used to override the existing computeValue(String)*

*\* implementation in MyValueCalculator*

*\*/*

**public** **class** **ReplacementComputeValue** **implements** MethodReplacer {

**public** Object reimplement(Object o, Method m, Object**[]** args) **throws** Throwable {

*// get the input value, work with it, and return a computed result*

String input = (String) args[0];

...

return ...;

}

}

تعریف bean برای استقرار کلاس اصلی و تعیین override متد به صورت زیر است:

<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

*<!-- arbitrary method replacement -->*

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>

می توانید از یک یا چند عنصر <arg-type/> در عنصر <replaced-method/> برای نشان دادن امضای متد متد در حال لغو استفاده کنید. امضای آرگومان ها تنها در صورتی ضروری است که متد بیش از حد بارگذاری شده باشد و چندین گونه در کلاس وجود داشته باشد. برای راحتی، رشته نوع آرگومان ممکن است زیر رشته ای از نام نوع کاملا واجد شرایط باشد. به عنوان مثال، موارد زیر همه با java.lang.String مطابقت دارند:

java.lang.String

String

Str

از آنجایی که تعداد آرگومان ها اغلب برای تمایز بین هر گزینه ممکن کافی است، این میانبر می تواند در تایپ زیادی صرفه جویی کند، زیرا به شما اجازه می دهد فقط کوتاه ترین رشته ای را که با یک نوع آرگومان مطابقت دارد تایپ کنید.

## 1.5. Bean scopes

هنگامی که یک تعریف bean ایجاد می کنید، یک دستور العمل برای ایجاد نمونه های واقعی از کلاس تعریف شده توسط آن تعریف bean ایجاد می کنید. این ایده که تعریف لوبیا یک دستور غذا است مهم است، زیرا به این معنی است که مانند یک کلاس، می‌توانید نمونه‌های شی بسیاری را از یک دستور غذا ایجاد کنید.

شما می‌توانید نه تنها وابستگی‌ها و مقادیر پیکربندی مختلفی را که قرار است به یک شی که از یک تعریف bean خاص ایجاد می‌شود، وصل کنید، بلکه دامنه اشیاء ایجاد شده از یک تعریف bean خاص را نیز کنترل کنید. این رویکرد از این جهت قدرتمند و انعطاف‌پذیر است که می‌توانید محدوده اشیایی را که ایجاد می‌کنید از طریق پیکربندی انتخاب کنید، به‌جای اینکه مجبور باشید در محدوده یک شی در سطح کلاس جاوا بپزید. Beans را می توان به گونه ای تعریف کرد که در یکی از تعدادی از حوزه ها به کار گرفته شوند: خارج از جعبه، Spring Framework از شش محدوده پشتیبانی می کند که چهار مورد از آنها فقط در صورت استفاده از یک ApplicationContext آگاه از وب در دسترس هستند.

حوزه های زیر خارج از جعبه پشتیبانی می شوند. همچنین می توانید یک محدوده سفارشی ایجاد کنید.

| *Table 3. Bean scopes* | |
| --- | --- |
| **Scope** | **Description** |
| [singleton](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes-singleton) | (پیش‌فرض) یک تعریف تک لوبیا را به یک نمونه شی در هر ظرف Spring IoC محدود می‌کند. |
| [prototype](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes-prototype) | یک تعریف تکی را به هر تعداد نمونه شی ارائه می دهد. |
| [request](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes-request) | یک تعریف واحد را به چرخه عمر یک درخواست HTTP اختصاص می دهد. یعنی هر درخواست HTTP نمونه مخصوص به خود را از یک Bean دارد که در پشت یک تعریف bean ایجاد شده است. فقط در زمینه یک Spring ApplicationContext آگاه از وب معتبر است. |
| [session](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes-session) | یک تعریف واحد را به چرخه حیات یک جلسه HTTP اختصاص می دهد. فقط در زمینه یک Spring ApplicationContext آگاه از وب معتبر است. |
| [application](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-scopes-application) | یک تعریف تک لوبیا را به چرخه حیات یک ServletContext ارائه می دهد. فقط در زمینه یک Spring ApplicationContext آگاه از وب معتبر است. |
| [websocket](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web.html#websocket-stomp-websocket-scope) | یک تعریف واحد را به چرخه حیات یک WebSocket ارائه می دهد. فقط در زمینه یک Spring ApplicationContext آگاه از وب معتبر است. |

فقط یک نمونه مشترک از یک لوبیا تکی مدیریت می‌شود، و همه درخواست‌های لوبیا با شناسه یا شناسه‌هایی که با آن تعریف لوبیا مطابقت دارند، منجر به برگرداندن آن یک نمونه از دانه‌های خاص توسط ظرف Spring می‌شود.

### 1.5.1. The singleton scope

به بیان دیگر، زمانی که شما یک تعریف bean تعریف می کنید و آن را به صورت singleton در نظر می گیرید، کانتینر Spring IoC دقیقاً یک نمونه از شی تعریف شده توسط آن تعریف bean را ایجاد می کند. این نمونه منفرد در حافظه پنهانی از چنین دانه‌های تکی ذخیره می‌شود، و تمام درخواست‌ها و مراجع بعدی برای آن bean نام‌گذاری شده، شی ذخیره‌شده را برمی‌گرداند.



مفهوم اسپرینگ از لوبیا تک تن با الگوی سینگلتون که در کتاب الگوهای باند چهار (GoF) تعریف شده است متفاوت است. GoF Singleton محدوده یک شی را به گونه‌ای کدگذاری می‌کند که یک و تنها یک نمونه از یک کلاس خاص در هر ClassLoader ایجاد می‌شود. دامنه تک تنه اسپرینگ به بهترین وجه در هر ظرف و در هر لوبیا توصیف می شود. این بدان معنی است که اگر یک bean برای یک کلاس خاص در یک ظرف Spring تعریف کنید، ظرف Spring یک و تنها یک نمونه از کلاس تعریف شده توسط آن تعریف bean را ایجاد می کند. محدوده ی تک تن، محدوده پیش فرض در بهار است. برای تعریف یک لوبیا به عنوان تک تن در XML، برای مثال می‌نویسید:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

*<!-- the following is equivalent, though redundant (singleton scope is the default) -->*

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

### 1.5.2. The prototype scope

محدوده غیر singleton و نمونه اولیه استقرار bean منجر به ایجاد یک نمونه bean جدید در هر بار درخواست برای آن bean خاص می شود. یعنی bean به bean دیگری تزریق می‌شود یا شما آن را از طریق فراخوانی متد getBean() روی container درخواست می‌کنید. به عنوان یک قاعده، از prototype scope برای همه لوبیاهای حالت دار و singleton scop برای لوبیاهای بدون حالت استفاده کنید.

نمودار زیر محدوده نمونه اولیه Spring را نشان می دهد. یک شی دسترسی به داده (DAO) معمولاً به عنوان یک prototype پیکربندی نمی شود، زیرا یک DAO معمولی هیچ حالت مکالمه ای را نگه نمی دارد. استفاده مجدد از هسته نمودار سینگلتون برای این نویسنده ساده تر بود.



مثال زیر یک bean را به عنوان یک نمونه اولیه در XML تعریف می کند:

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

برخلاف سایر حوزه‌ها، اسپرینگ چرخه حیات کامل prototype را مدیریت نمی‌کند: کانتینر یک شی نمونه اولیه را نمونه‌سازی می‌کند، پیکربندی می‌کند و در غیر این صورت یک شی نمونه اولیه را مونتاژ می‌کند، و آن را به مشتری تحویل می‌دهد، بدون اینکه سابقه دیگری از آن نمونه اولیه وجود داشته باشد. بنابراین، اگرچه روش‌های بازگشت به تماس چرخه حیات اولیه سازی بر روی همه اشیا بدون در نظر گرفتن دامنه فراخوانی می‌شوند، در مورد نمونه‌های اولیه، تماس‌های چرخه حیات تخریب پیکربندی شده فراخوانی نمی‌شوند. کد مشتری باید اشیاء با محدوده نمونه اولیه را پاکسازی کند و منابع گرانقیمتی را که لوبیا(های) نمونه اولیه در اختیار دارند، آزاد کند. برای اینکه ظرف Spring منابعی را که توسط لوبیاهای با محدوده نمونه اولیه نگهداری می شود آزاد کند، سعی کنید از یک پس پردازشگر سفارشی bean استفاده کنید، که اشاره ای به دانه هایی دارد که باید تمیز شوند.

از برخی جهات، نقش کانتینر Spring در مورد یک bean با محدوده نمونه اولیه، جایگزینی برای اپراتور جدید جاوا است. تمام مدیریت چرخه عمر گذشته از آن نقطه باید توسط مشتری انجام شود. (برای جزئیات بیشتر در مورد چرخه زندگی یک لوبیا در ظرف Spring، به تماس های چرخه حیات مراجعه کنید.)

### 1.5.3. Singleton beans with prototype-bean dependencies

هنگامی که از لوبیاهای singleton با وابستگی به prototype beanاستفاده می‌کنید، توجه داشته باشید که وابستگی‌ها در زمان instatiation حل می‌شوند. بنابراین، اگر شما یک bean با محدوده prototype را به یک bean با دامنه singleton تزریق کنید، یک نمونه اولیه لوبیا جدید نمونه سازی می شود و سپس وابستگی به لوبیای singleton تزریق می شود. prototype instanceتنها نمونه ای است که تا به حال به لوبیای singleton عرضه شده است.

با این حال، فرض کنید می‌خواهید در زمان اجرا، Bean تک‌تنه‌ای نمونه جدیدی از لوبیا با prototype-scopedرا به‌طور مکرر در زمان اجرا به‌دست آورد. شما نمی توانید یک لوبیا با prototype-scopedرا به لوبیا singleton خود تزریق کنید، زیرا این تزریق تنها یک بار انجام می شود، زمانی که container اسپرینگ در حال نمونه برداری از لوبیا singleton و رفع و تزریق وابستگی های آن است. اگر به یک نمونه جدید از یک نمونه اولیه در زمان اجرا بیش از یک بار نیاز دارید، به روش تزریق مراجعه کنید.

### 1.5.4. Request, session, application, and WebSocket scopes

دامنه request، session، application و websocket فقط در صورتی در دسترس است که از پیاده‌سازی Spring ApplicationContext آگاه از وب (مانند XmlWebApplicationContext) استفاده کنید. اگر از این محدوده‌ها با کانتینرهای Spring IoC معمولی مانند ClassPathXmlApplicationContext استفاده کنید، یک IllegalStateException با شکایت از محدوده bean ناشناخته پرتاب می‌شود.

#### Initial web configuration

برای پشتیبانی از محدوده لوبیاها در سطوح درخواست، جلسه، برنامه و سوکت وب (دانه‌های دارای محدوده وب)، قبل از تعریف کردن دانه‌ها، به پیکربندی اولیه جزئی نیاز است. (این راه‌اندازی اولیه برای اسکوپ‌های استاندارد، تک‌تنه و نمونه اولیه مورد نیاز نیست.)

نحوه انجام این تنظیمات اولیه به محیط Servlet خاص شما بستگی دارد.

اگر در Spring Web MVC به beans دسترسی داشته باشید، در واقع، در یک درخواستی که توسط Spring DispatcherServlet پردازش می‌شود، هیچ راه‌اندازی خاصی لازم نیست: DispatcherServlet در حال حاضر تمام وضعیت مربوطه را نشان می‌دهد.

اگر از یک کانتینر وب Servlet 2.5 استفاده می‌کنید، با درخواست‌هایی که خارج از Spring’s DispatcherServlet پردازش می‌شوند (به عنوان مثال، هنگام استفاده از JSF یا Struts)، باید org.springframework.web.context.request.RequestContextListener ServletRequestListener را ثبت کنید. برای Servlet 3.0+، این را می توان به صورت برنامه نویسی از طریق رابط WebApplicationInitializer انجام داد. یا برای کانتینرهای قدیمی‌تر، اعلان زیر را به فایل web.xml برنامه وب خود اضافه کنید:

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

از طرف دیگر، اگر مشکلاتی در تنظیم شنونده شما وجود دارد، از Spring's RequestContextFilter استفاده کنید. نگاشت فیلتر به پیکربندی برنامه وب اطراف بستگی دارد، بنابراین باید آن را در صورت لزوم تغییر دهید.

<web-app>

...

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>requestContextFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

...

</web-app>

DispatcherServlet، RequestContextListener و RequestContextFilter همگی دقیقاً همین کار را انجام می دهند، یعنی شی درخواست HTTP را به Thread که در حال سرویس دهی به آن درخواست است، متصل می کنند. این باعث می‌شود دانه‌هایی که در محدوده درخواست و جلسه هستند در ادامه زنجیره تماس در دسترس باشند.

#### Request scope

پیکربندی XML زیر را برای تعریف bean در نظر بگیرید:

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

ظرف Spring یک نمونه جدید از LoginAction bean با استفاده از تعریف loginAction bean برای هر درخواست HTTP ایجاد می کند. یعنی loginAction bean در سطح درخواست HTTP قرار دارد. شما می توانید وضعیت داخلی نمونه ای که ایجاد می شود را به اندازه دلخواه تغییر دهید، زیرا نمونه های دیگر ایجاد شده از همان تعریف loginAction bean این تغییرات را در حالت مشاهده نخواهند کرد. آنها برای یک درخواست فردی خاص هستند. هنگامی که پردازش درخواست کامل شد، لوبیایی که در محدوده درخواست قرار می‌گیرد دور انداخته می‌شود.

هنگام استفاده از مولفه های حاشیه نویسی یا پیکربندی جاوا، حاشیه نویسی @RequestScope می تواند برای اختصاص یک جزء به محدوده درخواست استفاده شود.

**@RequestScope**

@Component

**public** **class** **LoginAction** {

*// ...*

}

#### Session scope

پیکربندی XML زیر را برای تعریف bean در نظر بگیرید:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

ظرف Spring با استفاده از تعریف bean userPreferences برای طول عمر یک جلسه HTTP نمونه جدیدی از UserPreferences ایجاد می کند. به عبارت دیگر، userPreferences bean به طور موثر در سطح جلسه HTTP قرار می گیرد. همانند دانه‌های با محدوده درخواست، می‌توانید وضعیت داخلی نمونه‌ای که ایجاد می‌شود را تا جایی که می‌خواهید تغییر دهید، با توجه به اینکه سایر نمونه‌های جلسه HTTP که از نمونه‌های ایجاد شده از همان تعریف bean userPreferences نیز استفاده می‌کنند، این تغییرات را در وضعیت نمی‌بینند. ، زیرا آنها مختص یک جلسه HTTP فردی هستند. هنگامی که جلسه HTTP در نهایت دور ریخته می شود، دانه ای که به آن جلسه HTTP خاص اختصاص داده می شود نیز دور ریخته می شود.

هنگام استفاده از مولفه های حاشیه نویسی یا پیکربندی جاوا، از حاشیه نویسی @SessionScope می توان برای اختصاص یک جزء به محدوده جلسه استفاده کرد.

**@SessionScope**

@Component

**public** **class** **UserPreferences** {

*// ...*

}

#### Application scope

پیکربندی XML زیر را برای تعریف bean در نظر بگیرید:

<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>

ظرف Spring یک نمونه جدید از AppPreferences با استفاده از تعریف appPreferences bean یک بار برای کل برنامه وب ایجاد می کند. یعنی، appPreferences bean در سطح ServletContext، به عنوان یک ویژگی ServletContext معمولی ذخیره می شود. این تا حدودی شبیه به یک اسپرینگ تک لوبیا است، اما از دو جهت متفاوت است: این یک تک تن در هر ServletContext است، نه برای Spring 'ApplicationContext' (که ممکن است چندین مورد در هر برنامه وب خاص وجود داشته باشد)، و در واقع در معرض دید قرار می گیرد و بنابراین به عنوان یک ویژگی ServletContext قابل مشاهده است.

هنگام استفاده از مولفه های حاشیه نویسی یا پیکربندی جاوا، از حاشیه نویسی @ApplicationScope می توان برای اختصاص یک جزء به محدوده برنامه استفاده کرد.

**@ApplicationScope**

@Component

**public** **class** **AppPreferences** {

*// ...*

}

#### Scoped beans as dependencies

ظرف Spring IoC نه تنها نمونه سازی اشیاء شما (لوبیا)، بلکه سیم کشی همکاران (یا وابستگی ها) را نیز مدیریت می کند. اگر می‌خواهید (به عنوان مثال) یک Bean scoped درخواست HTTP را به Bean دیگری با دامنه طولانی‌تر تزریق کنید، می‌توانید یک پراکسی AOP را به جای Bean scoped تزریق کنید. به این معنا که باید یک شی پراکسی تزریق کنید که همان واسط عمومی را با شی scope شده در معرض نمایش بگذارد، اما می‌تواند شی هدف واقعی را از محدوده مربوطه (مانند درخواست HTTP) بازیابی کند و فراخوانی‌های متد را به شی واقعی واگذار کند.

همچنین می‌توانید از <aop:scoped-proxy/> بین دانه‌هایی استفاده کنید که به صورت تک‌تنه تعیین می‌شوند، و سپس مرجع از یک پروکسی میانی عبور می‌کند که قابل سریال‌سازی است و بنابراین می‌تواند در زمان سریال‌زدایی، bean هدف را دوباره به دست آورد.

هنگام اعلان <aop:scoped-proxy/> در برابر نمونه اولیه bean of scope، هر فراخوانی متد بر روی پراکسی مشترک منجر به ایجاد یک نمونه هدف جدید می شود که تماس سپس به آن ارسال می شود.

همچنین، پراکسی‌های scoped تنها راه برای دسترسی به bean از محدوده‌های کوتاه‌تر به روشی امن برای چرخه زندگی نیستند. همچنین می‌توانید به سادگی نقطه تزریق خود (یعنی آرگومان سازنده/تنظیم کننده یا فیلد خودکار) را به‌عنوان ObjectFactory<MyTargetBean> اعلام کنید، و به یک فراخوانی getObject() اجازه می‌دهید تا هر زمان که نیاز است نمونه فعلی را بر اساس تقاضا بازیابی کنید - بدون نگه داشتن روی به عنوان مثال یا ذخیره آن به طور جداگانه.

به عنوان یک نوع توسعه یافته، می توانید ObjectProvider<MyTargetBean> را اعلام کنید که چندین نوع دسترسی اضافی از جمله getIfAvailable و getIfUnique را ارائه می دهد.

نوع JSR-330 آن Provider نامیده می شود که با یک اعلان Provider<MyTargetBean> و یک فراخوان get() مربوطه برای هر تلاش بازیابی استفاده می شود. برای جزئیات بیشتر در مورد JSR-330 به طور کلی اینجا را ببینید.

پیکربندی در مثال زیر تنها یک خط است، اما درک "چرا" و همچنین "چگونه" پشت آن مهم است.

<?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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

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

*<!-- an HTTP Session-scoped bean exposed as a proxy -->*

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

*<!-- instructs the container to proxy the surrounding bean -->*

<aop:scoped-proxy/>

</bean>

*<!-- a singleton-scoped bean injected with a proxy to the above bean -->*

<bean id="userService" class="com.foo.SimpleUserService">

*<!-- a reference to the proxied userPreferences bean -->*

<property name="userPreferences" ref="userPreferences"/>

</bean>

</beans>

برای ایجاد چنین پراکسی، یک عنصر فرزند <aop:scoped-proxy/> را در یک تعریف scoped bean وارد می‌کنید (به انتخاب نوع پروکسی برای ایجاد و پیکربندی مبتنی بر طرحواره XML مراجعه کنید). چرا تعاریف bean که در سطوح درخواست، جلسه و محدوده سفارشی محدوده می شوند، به عنصر <aop:scoped-proxy/> نیاز دارند؟ بیایید تعریف لوبیای سینگلتون زیر را بررسی کنیم و آن را با آنچه که باید برای حوزه‌های فوق‌الذکر تعریف کنید مقایسه کنیم (توجه داشته باشید که تعریف لوبیا UserPreferences زیر ناقص است).

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

در مثال قبل، userManager singleton bean با ارجاع به userPreferences با محدوده HTTP Session تزریق شده است. نکته برجسته در اینجا این است که userManager bean یک تکه است: دقیقاً یک بار در هر ظرف نمونه سازی می شود و وابستگی های آن (در این مورد فقط یکی، userPreferences bean) نیز فقط یک بار تزریق می شود. این بدان معنی است که userManager bean فقط روی همان شی userPreferences کار می‌کند، یعنی همان چیزی که در ابتدا با آن تزریق شده است.

این رفتاری نیست که شما می‌خواهید هنگام تزریق یک دانه با طول عمر کوتاه‌تر به یک دانه با طول عمر طولانی‌تر، برای مثال تزریق یک دانه مشارکتی با محدوده HTTP Session به عنوان یک وابستگی به دانه تک‌تن. در عوض، شما به یک شی userManager نیاز دارید، و برای طول عمر یک جلسه HTTP، به یک شی userPreferences نیاز دارید که مخصوص جلسه HTTP گفته شده باشد. بنابراین کانتینر یک شی ایجاد می‌کند که دقیقاً همان رابط عمومی کلاس UserPreferences را نشان می‌دهد (در حالت ایده‌آل یک شی که یک نمونه UserPreferences است) که می‌تواند شی UserPreferences واقعی را از مکانیسم محدوده (درخواست HTTP، Session و غیره) واکشی کند. کانتینر این شی پراکسی را به userManager bean تزریق می‌کند، که نمی‌داند این مرجع UserPreferences یک پروکسی است. در این مثال، وقتی یک نمونه UserManager متدی را بر روی شی UserPreferences تزریق شده با وابستگی فراخوانی می‌کند، در واقع در حال فراخوانی متدی در پروکسی است. سپس پروکسی شی UserPreferences واقعی را از (در این مورد) HTTP Session واکشی می کند و فراخوانی متد را به شی UserPreferences واقعی بازیابی شده واگذار می کند.

بنابراین، هنگام تزریق bean با محدوده درخواست و جلسه به اشیاء همکار، به پیکربندی صحیح و کامل زیر نیاز دارید:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

<aop:scoped-proxy/>

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

##### Choosing the type of proxy to create

به‌طور پیش‌فرض، وقتی کانتینر Spring یک پروکسی برای یک bean ایجاد می‌کند که با عنصر <aop:scoped-proxy/> علامت‌گذاری شده است، یک پروکسی کلاس مبتنی بر CGLIB ایجاد می‌شود.

پراکسی های CGLIB فقط تماس های متد عمومی را رهگیری می کنند! روش های غیر عمومی را در چنین پروکسی فراخوانی نکنید. آنها به شی هدف با محدوده واقعی تفویض نمی شوند.

همچنین، می‌توانید کانتینر Spring را برای ایجاد پراکسی‌های استاندارد مبتنی بر رابط JDK برای چنین دانه‌هایی با محدوده، با تعیین false برای مقدار ویژگی proxy-target-class عنصر <aop:scoped-proxy/> پیکربندی کنید. استفاده از پراکسی های مبتنی بر رابط JDK به این معنی است که برای انجام چنین پروکسی به کتابخانه های اضافی در مسیر کلاس برنامه خود نیاز ندارید. با این حال، همچنین به این معنی است که کلاس bean scoped باید حداقل یک رابط را پیاده سازی کند، و همه همکارانی که bean scoped به آن تزریق می شود باید از طریق یکی از رابط های آن به bean ارجاع دهند.

*<!-- DefaultUserPreferences implements the UserPreferences interface -->*

<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">

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

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

### 1.5.5. Custom scopes

مکانیسم محدوده لوبیا قابل توسعه است. شما می‌توانید محدوده‌های خود را تعریف کنید، یا حتی محدوده‌های موجود را دوباره تعریف کنید، اگرچه دومی عمل بدی محسوب می‌شود و نمی‌توانید محدوده‌های تک‌تنه داخلی و نمونه اولیه را نادیده بگیرید.

#### Creating a custom scope

برای ادغام دامنه(های) سفارشی خود در ظرف Spring، باید رابط org.springframework.beans.factory.config.Scope را پیاده سازی کنید که در این بخش توضیح داده شده است. برای آشنایی با نحوه پیاده‌سازی دامنه‌های خود، به پیاده‌سازی‌های Scope که با خود Spring Framework و Scope javadocs ارائه شده‌اند، مراجعه کنید، که روش‌هایی را که باید برای پیاده‌سازی با جزئیات بیشتر توضیح می‌دهد، ببینید.

رابط Scope دارای چهار روش برای دریافت اشیا از محدوده، حذف آنها از محدوده و اجازه از بین بردن آنها است.

روش زیر شی را از محدوده زیرین باز می گرداند. برای مثال، اجرای session scope bean با محدوده جلسه را برمی‌گرداند (و اگر وجود نداشته باشد، این روش نمونه جدیدی از bean را پس از اینکه آن را به جلسه برای مراجعات بعدی متصل کرد، برمی‌گرداند).

Object get(String name, ObjectFactory objectFactory)

روش زیر شیء را از محدوده زیرین حذف می کند. برای مثال، اجرای Session Scope، Bean با محدوده جلسه را از جلسه اصلی حذف می‌کند. شی باید برگردانده شود، اما اگر شی با نام مشخص شده یافت نشد، می توانید null را برگردانید.

Object remove(String name)

روش زیر تماس‌هایی را که scope باید در هنگام از بین رفتن یا زمانی که شی مشخص شده در محدوده از بین می‌رود، اجرا کند، ثبت می‌کند. برای اطلاعات بیشتر در مورد تماس‌های تخریب، به javadocs یا پیاده‌سازی Spring Scope مراجعه کنید.

**void** registerDestructionCallback(String name, Runnable destructionCallback)

روش زیر شناسه مکالمه را برای محدوده زیرین به دست می آورد. این شناسه برای هر محدوده متفاوت است. برای پیاده سازی با محدوده جلسه، این شناسه می تواند شناسه جلسه باشد.

String getConversationId()

#### Using a custom scope

پس از نوشتن و آزمایش یک یا چند پیاده سازی Scope سفارشی، باید ظرف Spring را از محدوده (های) جدید خود آگاه کنید. روش زیر روش مرکزی برای ثبت یک Scope جدید با کانتینر Spring است:

**void** registerScope(String scopeName, Scope scope);

این روش در رابط ConfigurableBeanFactory اعلام شده است، که در اکثر پیاده سازی های بتن ApplicationContext که با Spring از طریق ویژگی BeanFactory ارائه می شوند، موجود است.

اولین آرگومان متد registerScope(..) نام منحصر به فرد مرتبط با یک scope است. نمونه‌هایی از این نام‌ها در خود ظرف اسپرینگ تک‌تنه و نمونه اولیه هستند. آرگومان دوم متد registerScope(..) یک نمونه واقعی از پیاده سازی سفارشی Scope است که می خواهید ثبت و استفاده کنید.

فرض کنید پیاده سازی Scope سفارشی خود را می نویسید و سپس آن را به صورت زیر ثبت می کنید.

مثال زیر از SimpleThreadScope استفاده می کند که همراه Spring گنجانده شده است، اما به طور پیش فرض ثبت نشده است. دستورالعمل ها برای پیاده سازی های Scope سفارشی شما یکسان خواهد بود.

Scope threadScope = **new** SimpleThreadScope();

beanFactory.registerScope("thread", threadScope);

سپس تعاریف bean را ایجاد می کنید که به قوانین محدوده محدوده سفارشی شما پایبند باشد:

<bean id="..." class="..." scope="thread">

با پیاده سازی Scope سفارشی، شما محدود به ثبت برنامه ای دامنه نیستید. همچنین می توانید با استفاده از کلاس CustomScopeConfigurer ثبت نام Scope را به صورت اعلامی انجام دهید:

<?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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

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

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">

<property name="scopes">

<map>

<entry key="thread">

<bean class="org.springframework.context.support.SimpleThreadScope"/>

</entry>

</map>

</property>

</bean>

<bean id="bar" class="x.y.Bar" scope="thread">

<property name="name" value="Rick"/>

<aop:scoped-proxy/>

</bean>

<bean id="foo" class="x.y.Foo">

<property name="bar" ref="bar"/>

</bean>

</beans>

وقتی <aop:scoped-proxy/> را در یک پیاده‌سازی FactoryBean قرار می‌دهید، این خود Factory Bean است که محدوده‌بندی می‌شود، نه شیء برگشتی از getObject().

## 1.6. Customizing the nature of a bean

#### 1.6.1. Lifecycle callbacks

برای تعامل با مدیریت کانتینر چرخه حیات bean، می توانید رابط های Spring InitializingBean و DisposableBean را پیاده سازی کنید. ظرف برای اولی afterPropertiesSet را فراخوانی می‌کند و برای دومی death() را فراخوانی می‌کند تا به bean اجازه دهد تا اقدامات خاصی را پس از مقداردهی اولیه و از بین بردن bean انجام دهد.

حاشیه‌نویسی‌های JSR-250 @PostConstruct و @PreDestroy معمولاً بهترین روش برای دریافت تماس‌های چرخه حیات در یک برنامه مدرن Spring در نظر گرفته می‌شوند. استفاده از این حاشیه نویسی به این معنی است که دانه های شما با رابط های خاص Spring جفت نمی شوند. برای جزئیات بیشتر به @PostConstruct و @PreDestroy مراجعه کنید.

اگر نمی‌خواهید از حاشیه‌نویسی‌های JSR-250 استفاده کنید، اما همچنان به دنبال حذف کوپلینگ هستید، استفاده از متادیتای تعریف شیء init-method و definition-method را در نظر بگیرید.

در داخل، Spring Framework از پیاده سازی های BeanPostProcessor برای پردازش هر واسط برگشتی که بتواند پیدا کند و روش های مناسب را فراخوانی کند، استفاده می کند. اگر به ویژگی‌های سفارشی یا سایر رفتارهای چرخه حیات نیاز دارید که Spring خارج از جعبه ارائه نمی‌دهد، می‌توانید خودتان BeanPostProcessor را پیاده‌سازی کنید. برای اطلاعات بیشتر، به نقاط گسترش کانتینر مراجعه کنید.

علاوه بر فراخوانی اولیه و تخریب، اشیاء مدیریت شده توسط Spring همچنین ممکن است رابط Lifecycle را پیاده سازی کنند تا آن اشیاء بتوانند در فرآیند راه اندازی و خاموش شدن که توسط چرخه حیات خود کانتینر هدایت می شود، شرکت کنند.

رابط های برگشت تماس چرخه حیات در این بخش توضیح داده شده است.

##### Initialization callbacks

رابط org.springframework.beans.factory.InitializingBean به یک Bean اجازه می دهد تا پس از تنظیم تمام ویژگی های ضروری روی bean توسط ظرف، کار مقداردهی اولیه را انجام دهد. رابط InitializingBean یک روش واحد را مشخص می کند:

**void** afterPropertiesSet() **throws** Exception;

توصیه می شود از رابط InitializingBean استفاده نکنید زیرا به طور غیر ضروری کد را با Spring جفت می کند. متناوبا، از حاشیه نویسی @PostConstruct استفاده کنید یا یک روش اولیه سازی POJO را مشخص کنید. در مورد فراداده پیکربندی مبتنی بر XML، از ویژگی init-method برای تعیین نام روشی استفاده می‌کنید که دارای امضای بدون آرگومان void است. با پیکربندی جاوا، از ویژگی initMethod @Bean استفاده می‌کنید، به دریافت تماس‌های چرخه حیات مراجعه کنید. به عنوان مثال موارد زیر:

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

**public** **class** **ExampleBean** {

**public** **void** init() {

*// do some initialization work*

}

}

… دقیقاً مشابه …

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

**public** **class** **AnotherExampleBean** **implements** InitializingBean {

**public** **void** afterPropertiesSet() {

*// do some initialization work*

}

}

اما کد را با Spring جفت نمی کند.

##### Destruction callbacks

پیاده‌سازی رابط org.springframework.beans.factory.DisposableBean به Bean اجازه می‌دهد تا زمانی که ظرف حاوی آن از بین می‌رود، یک تماس دریافت کند. رابط DisposableBean یک روش واحد را مشخص می کند:

**void** destroy() **throws** Exception;

توصیه می‌شود از رابط تماس برگشتی DisposableBean استفاده نکنید زیرا به طور غیرضروری کد را با Spring جفت می‌کند. از طرف دیگر، از حاشیه نویسی @PreDestroy استفاده کنید یا یک روش عمومی را مشخص کنید که توسط تعاریف bean پشتیبانی می شود. با فراداده پیکربندی مبتنی بر XML، شما از ویژگی death-method در <bean/> استفاده می کنید. با پیکربندی جاوا، شما از ویژگی deathMethod @Bean استفاده می‌کنید، به دریافت تماس‌های چرخه حیات مراجعه کنید. به عنوان مثال، تعریف زیر:

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

**public** **class** **ExampleBean** {

**public** **void** cleanup() {

*// do some destruction work (like releasing pooled connections)*

}

}

دقیقاً مشابه است:

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

**public** **class** **AnotherExampleBean** **implements** DisposableBean {

**public** **void** destroy() {

*// do some destruction work (like releasing pooled connections)*

}

}

اما کد را با Spring جفت نمی کند.

می‌توان به ویژگی death-method عنصر <bean> یک مقدار خاص (استنباط‌شده) اختصاص داد که به Spring دستور می‌دهد تا به طور خودکار یک روش بسته یا خاموش کردن عمومی را در کلاس bean خاص (هر کلاسی که java.lang.AutoCloseable یا java را پیاده‌سازی می‌کند) شناسایی کند. بنابراین io.Closeable مطابقت دارد). این مقدار ویژه (استنباط شده) همچنین می تواند بر روی ویژگی default-destroy-method عنصر <beans> تنظیم شود تا این رفتار را در کل مجموعه ای از bean اعمال کند (به روش های اولیه سازی و نابودی پیش فرض مراجعه کنید). توجه داشته باشید که این رفتار پیش فرض با پیکربندی جاوا است.

##### Default initialization and destroy methods

هنگامی که مقداردهی اولیه را می نویسید و فراخوان های متد را که از واسط های تماس InitializingBean و DisposableBean خاص Spring استفاده نمی کنند، می نویسید، معمولاً متدهایی را با نام هایی مانند init()، initialize()، dispose() و غیره می نویسید. در حالت ایده‌آل، نام چنین روش‌های بازگشت به تماس چرخه حیات در سراسر یک پروژه استاندارد می‌شود تا همه توسعه‌دهندگان از نام روش‌های یکسانی استفاده کنند و از ثبات اطمینان حاصل کنند.

می‌توانید کانتینر Spring را طوری پیکربندی کنید که به دنبال مقداردهی اولیه نام‌گذاری شده باشد و نام روش‌های برگشت تماس را در هر Bean از بین ببرد. این بدان معنی است که شما به عنوان یک توسعه دهنده برنامه، می توانید کلاس های برنامه خود را بنویسید و از یک فراخوان اولیه به نام init() استفاده کنید، بدون اینکه نیازی به پیکربندی ویژگی init-method="init" با هر تعریف bean داشته باشید. کانتینر Spring IoC آن روش را زمانی فراخوانی می‌کند که bean ایجاد می‌شود (و مطابق با قرارداد بازگشت تماس چرخه عمر استاندارد که قبلاً توضیح داده شد). این ویژگی همچنین یک قرارداد نام‌گذاری ثابت را برای مقداردهی اولیه و از بین بردن تماس‌های متد اعمال می‌کند.

فرض کنید که متدهای برگشتی فراخوانی اولیه شما با نام init() و متدهای نابود کننده callback با نام death() هستند. کلاس شما شبیه کلاس در مثال زیر خواهد بود.

**public** **class** **DefaultBlogService** **implements** BlogService {

**private** BlogDao blogDao;

**public** **void** setBlogDao(BlogDao blogDao) {

this.blogDao = blogDao;

}

*// this is (unsurprisingly) the initialization callback method*

**public** **void** init() {

**if** (this.blogDao == null) {

**throw** **new** IllegalStateException("The [blogDao] property must be set.");

}

}

}

<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

وجود ویژگی پیش‌فرض-init-method در ویژگی عنصر سطح بالای <beans/> باعث می‌شود که ظرف Spring IoC روشی به نام init روی beans را به‌عنوان فراخوان روش اولیه تشخیص دهد. هنگامی که یک bean ایجاد و مونتاژ می شود، اگر کلاس bean چنین روشی داشته باشد، در زمان مناسب فراخوانی می شود.

شما با استفاده از ویژگی پیش‌فرض-destroy-method در عنصر سطح بالای <beans/>، بازخوانی‌های متد تخریب را به طور مشابه پیکربندی می‌کنید (یعنی در XML).

در جایی که کلاس‌های bean موجود قبلاً متدهای برگشتی دارند که با قرارداد نامگذاری شده‌اند، می‌توانید با تعیین نام متد (در XML) با استفاده از ویژگی‌های init-method و death-method از <bean/>، پیش‌فرض را لغو کنید. خود

ظرف Spring تضمین می‌کند که یک تماس اولیه پیکربندی شده بلافاصله پس از ارائه تمام وابستگی‌ها به یک Bean فراخوانی می‌شود. بنابراین فراخوانی اولیه بر روی مرجع bean خام فراخوانی می شود، به این معنی که رهگیرهای AOP و غیره هنوز روی bean اعمال نشده اند. ابتدا یک Bean به طور کامل ایجاد می شود، سپس یک پراکسی AOP (به عنوان مثال) با زنجیره رهگیر آن اعمال می شود. اگر هدف و پراکسی به طور جداگانه تعریف شده باشند، کد شما حتی می تواند با دور زدن پراکسی با دانه هدف خام تعامل داشته باشد. از این رو، اعمال رهگیرها در روش init متناقض است، زیرا انجام این کار باعث می شود چرخه حیات bean هدف با پراکسی/رهگیرهای آن مرتبط شود و هنگامی که کد شما مستقیماً با bean هدف خام تعامل می کند، معنای عجیبی باقی می ماند.

##### Combining lifecycle mechanisms

از Spring 2.5، شما سه گزینه برای کنترل رفتار چرخه حیات bean دارید: واسط های InitializingBean و DisposableBean. متدهای سفارشی init() و death(); و حاشیه نویسی @PostConstruct و @PreDestroy. شما می توانید این مکانیسم ها را برای کنترل یک دانه معین ترکیب کنید.

اگر چندین مکانیسم چرخه حیات برای یک Bean پیکربندی شده باشد، و هر مکانیزم با نام متد متفاوتی پیکربندی شده باشد، آنگاه هر روش پیکربندی شده به ترتیب ذکر شده در زیر اجرا می شود. با این حال، اگر همان نام متد - برای مثال، init() برای یک روش اولیه سازی - برای بیش از یکی از این مکانیسم های چرخه حیات پیکربندی شده باشد، آن متد یک بار اجرا می شود، همانطور که در بخش قبل توضیح داده شد.

مکانیسم های چرخه عمر چندگانه پیکربندی شده برای همان لوبیا ، با روش های مختلف اولیه سازی ، به شرح زیر است:

روشهای حاشیه نویسی با postconstruct

AfterPropertiesset () همانطور که توسط InitializingBean callback interface تعریف شده است

یک روش INIT () پیکربندی شده سفارشی

روشهای destroy به همان ترتیب خوانده می شوند:

روشهای حاشیه نویسی با predestroy

destroy () همانطور که توسط DisposableBean callback interface تعریف شده است

یک روش destroy () پیکربندی شده سفارشی

تماس های راه اندازی و خاموش کردن

رابط چرخه عمر روشهای اساسی را برای هر شیء که نیاز به چرخه عمر خاص خود دارد تعریف می کند (به عنوان مثال شروع و متوقف کردن برخی از فرایندهای پس زمینه):

**public** **interface** **Lifecycle** {

**void** start();

**void** stop();

**boolean** isRunning();

}

هر شیء مدیریت شده توسط Spring ممکن است آن رابط را پیاده سازی کند. سپس، هنگامی که خود ApplicationContext سیگنال های شروع و توقف را دریافت می کند، به عنوان مثال. برای یک سناریوی توقف/راه‌اندازی مجدد در زمان اجرا، آن تماس‌ها را به تمام پیاده‌سازی‌های چرخه حیات تعریف‌شده در آن زمینه آبشاری می‌کند. این کار را با تفویض اختیار به Lifecycle Processor انجام می دهد:

**public** **interface** **LifecycleProcessor** **extends** Lifecycle {

**void** onRefresh();

**void** onClose();

}

توجه داشته باشید که LifecycleProcessor خود توسعه‌ای از رابط Lifecycle است. همچنین دو روش دیگر برای واکنش به متنی که در حال refresh و close است اضافه می کند.

توجه داشته باشید که رابط معمولی org.springframework.context.Lifecycle فقط یک قرارداد ساده برای اعلان‌های شروع/توقف صریح است و به معنای راه‌اندازی خودکار در زمان تازه‌سازی متن نیست. اجرای org.springframework.context.SmartLifecycle را به‌جای کنترل دقیق راه‌اندازی خودکار یک bean خاص (از جمله مراحل راه‌اندازی) در نظر بگیرید. همچنین، لطفاً توجه داشته باشید که اعلان‌های توقف قبل از نابودی تضمین نمی‌شوند: در حالت خاموش شدن منظم، همه لوبیاهای Lifecycle ابتدا قبل از انتشار تماس‌های کلی تخریب، یک اعلان توقف دریافت می‌کنند. با این حال، در بازخوانی داغ در طول عمر یک زمینه یا در تلاش‌های به‌روزرسانی لغو شده، تنها روش‌های destroy فراخوانی می‌شوند.

ترتیب فراخوانی راه اندازی و خاموش شدن می تواند مهم باشد. اگر یک رابطه " depends-on " بین هر دو شی وجود داشته باشد، طرف وابسته بعد از وابستگی شروع می شود و قبل از وابستگی متوقف می شود. با این حال، گاهی اوقات وابستگی مستقیم ناشناخته است. ممکن است فقط بدانید که اشیاء از یک نوع خاص باید قبل از اشیاء نوع دیگری شروع شوند. در این موارد، رابط SmartLifecycle گزینه دیگری را تعریف می کند، یعنی متد getPhase() همانطور که در سوپرواسط آن، Phased تعریف شده است.

**public** **interface** **Phased** {

**int** getPhase();

}

**public** **interface** **SmartLifecycle** **extends** Lifecycle, Phased {

**boolean** isAutoStartup();

**void** stop(Runnable callback);

}

هنگام شروع ، اشیاء با کمترین مرحله ابتدا شروع می شوند و هنگام توقف ، ترتیب معکوس دنبال می شود. بنابراین ، شیئی که SmartLifecycle را پیاده سازی می کند و روش getPhase () آن را به صورت صحیح باز می گرداند. min\_value جزو اولین کسانی است که شروع می کند و آخرین بار متوقف می شود. در انتهای دیگر طیف ، مقدار فاز عدد صحیح. max\_value نشان می دهد که باید آخرین شیء شروع شود و ابتدا متوقف شود (احتمالاً به این دلیل که بستگی به سایر فرآیندهای در حال اجرا دارد). هنگام در نظر گرفتن مقدار فاز ، همچنین مهم است که بدانید که مرحله پیش فرض برای هر شیء چرخه عمر "عادی" که SmartLifecycle را اجرا نمی کند 0 خواهد بود. بنابراین ، هر مقدار فاز منفی نشان می دهد که یک شی باید قبل از آن اجزای استاندارد شروع شود (و پس از آنها متوقف شوید) و برعکس برای هر مقدار فاز مثبت.

همانطور که می بینید روش توقف تعریف شده توسط SmartLifecycle پاسخ به تماس را می پذیرد. هر اجرای باید پس از اتمام روند خاموش کردن اجرای ، از روش پاسخ (پاسخ () پاسخ استفاده کند. این امر باعث خاموش شدن ناهمزمان در صورت لزوم می شود زیرا اجرای پیش فرض رابط LifecycleProcessor ، DefaultLifecycleProcessor ، منتظر مقدار زمان بندی خود برای گروه اشیاء در هر مرحله خواهد بود تا از آن پاسخ تماس بگیرند. زمان پیش فرض در هر فاز 30 ثانیه است. شما می توانید با تعریف لوبیا به نام "LifecycleProcessor" در متن ، نمونه پردازنده چرخه پیش فرض را نادیده بگیرید. اگر فقط می خواهید مدت زمان را تغییر دهید ، تعریف موارد زیر کافی خواهد بود:

<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">

*<!-- timeout value in milliseconds -->*

<property name="timeoutPerShutdownPhase" value="10000"/>

</bean>

همانطور که گفته شد، رابط LifecycleProcessor روش‌هایی را برای بازخوانی و بستن متن نیز تعریف می‌کند. دومی به سادگی فرآیند خاموش شدن را به گونه ای هدایت می کند که گویی stop() به صراحت فراخوانی شده باشد، اما زمانی اتفاق می افتد که context بسته شود. از سوی دیگر، «بازخوانی مجدد» یکی دیگر از ویژگی های لوبیاهای SmartLifecycle را فعال می کند. هنگامی که context به‌روزرسانی می‌شود (پس از اینکه همه اشیا نمونه‌سازی و مقداردهی اولیه شدند)، آن فراخوانی فراخوانی می‌شود، و در آن نقطه پردازشگر چرخه حیات پیش‌فرض مقدار بولی بازگردانده شده توسط روش ()isAutoStartup هر شی SmartLifecycle را بررسی می‌کند. اگر "درست" باشد، آن شی در آن نقطه شروع می شود به جای اینکه منتظر فراخوانی صریح متد ()start مربوط به متن باشد (برخلاف تجدید متن، شروع متن به طور خودکار برای اجرای متن استاندارد اتفاق نمی افتد) . مقدار "فاز" و همچنین هر رابطه "وابسته به" ترتیب راه اندازی را به همان روشی که در بالا توضیح داده شد تعیین می کند.

##### Shutting down the Spring IoC container gracefully in non-web applications

این بخش فقط برای برنامه های غیر وب اعمال می شود. پیاده‌سازی‌های ApplicationContext مبتنی بر وب Spring قبلاً دارای کدهایی هستند که می‌توانند ظرف Spring IoC را به‌خوبی هنگامی که برنامه وب مربوطه بسته می‌شود، خاموش کنند.

اگر از کانتینر Spring's IoC در یک محیط برنامه غیر وب استفاده می کنید. به عنوان مثال، در یک محیط دسکتاپ مشتری غنی. شما یک قلاب خاموش کردن را با JVM ثبت می کنید. انجام این کار یک خاموشی دلپذیر را تضمین می‌کند و روش‌های تخریب مربوطه را روی دانه‌های سینگلتون فراخوانی می‌کند تا همه منابع آزاد شوند. البته، شما همچنان باید این callbacks های نابود کننده را به درستی پیکربندی و پیاده سازی کنید.

برای ثبت یک hook shutdown، متد registerShutdownHook() را که در رابط ConfigurableApplicationContext اعلام شده است فراخوانی می کنید:

**import** org.springframework.context.ConfigurableApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **final** **class** **Boot** {

**public** **static** **void** main(**final** String**[]** args) **throws** Exception {

ConfigurableApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

*// add a shutdown hook for the above context...*

ctx.registerShutdownHook();

*// app runs here...*

*// main method exits, hook is called prior to the app shutting down...*

}

}

#### 1.6.2. ApplicationContextAware and BeanNameAware

هنگامی که یک ApplicationContext یک نمونه شی ایجاد می کند که رابط org.springframework.context.ApplicationContextAware را پیاده سازی می کند، نمونه با ارجاع به آن ApplicationContext ارائه می شود.

**public** **interface** **ApplicationContextAware** {

**void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException;

}

بنابراین beanها می توانند ApplicationContext را که آنها را ایجاد کرده است، از طریق رابط ApplicationContext یا با فرستادن ارجاع به زیر کلاس شناخته شده این رابط، مانند ConfigurableApplicationContext، که عملکردهای اضافی را نشان می دهد، دستکاری کنند. یکی از کاربردها، بازیابی برنامه‌ای سایر لوبیاها است. گاهی اوقات این قابلیت مفید است. با این حال، به طور کلی باید از آن اجتناب کنید، زیرا کد را با Spring جفت می کند و از سبک Inversion of Control پیروی نمی کند، جایی که همکاران به عنوان ویژگی به beans ارائه می شوند. روش‌های دیگر ApplicationContext دسترسی به منابع فایل، انتشار رویدادهای برنامه و دسترسی به MessageSource را فراهم می‌کنند. این ویژگی های اضافی در قابلیت های اضافی ApplicationContext توضیح داده شده است

از Spring 2.5، سیم کشی خودکار جایگزین دیگری برای به دست آوردن مرجع به ApplicationContext است. حالت‌های سیم‌کشی خودکار سازنده «سنتی» و byType (همانطور که در Autowiring همکاران توضیح داده شد) می‌توانند به ترتیب وابستگی از نوع ApplicationContext را برای یک آرگومان سازنده یا پارامتر متد تنظیم‌کننده فراهم کنند. برای انعطاف‌پذیری بیشتر، از جمله امکان سیم‌کشی خودکار فیلدها و روش‌های پارامترهای متعدد، از ویژگی‌های جدید سیم‌کشی خودکار مبتنی بر حاشیه‌نویسی استفاده کنید. اگر این کار را انجام دهید، اگر فیلد، سازنده یا روش مورد نظر دارای حاشیه نویسی @Autowired باشد، ApplicationContext به صورت خودکار به یک فیلد، آرگومان سازنده یا پارامتر متد متصل می شود که انتظار نوع ApplicationContext را دارد. برای اطلاعات بیشتر، @Autowired را ببینید.

هنگامی که یک ApplicationContext کلاسی ایجاد می کند که رابط org.springframework.beans.factory.BeanNameAware را پیاده سازی می کند، کلاس با ارجاع به نام تعریف شده در تعریف شی مرتبط با آن ارائه می شود.

**public** **interface** **BeanNameAware** {

**void** setBeanName(String name) **throws** BeansException;

}

فراخوانی پس از جمع‌بندی ویژگی‌های bean معمولی، اما قبل از یک فراخوانی اولیه مانند InitializingBean afterPropertiesSet یا یک روش init سفارشی فراخوانی می‌شود.

#### 1.6.3. Other Aware interfaces

علاوه بر ApplicationContextAware و BeanNameAware که در بالا مورد بحث قرار گرفت، Spring طیف گسترده‌ای از واسط‌های برگشت تماس Aware را ارائه می‌دهد که به bean‌ها اجازه می‌دهد به ظرف نشان دهند که به یک وابستگی زیرساخت خاصی نیاز دارند. مهم‌ترین رابط‌های Aware در زیر خلاصه می‌شوند - به عنوان یک قاعده کلی، نام نشانه خوبی از نوع وابستگی است:

| *Table 4. Aware interfaces* | | | |
| --- | --- | --- | --- |
| **Name** | **Injected Dependency** | **Explained in…​** |
| ApplicationContextAware | Declaring ApplicationContext | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-aware) |
| ApplicationEventPublisherAware | Event publisher of the enclosing ApplicationContext | [Additional capabilities of the ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#context-introduction) |
| BeanClassLoaderAware | Class loader used to load the bean classes. | [Instantiating beans](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-class) |
| BeanFactoryAware | Declaring BeanFactory | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-aware) |
| BeanNameAware | Name of the declaring bean | [ApplicationContextAware and BeanNameAware](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#beans-factory-aware) |
| BootstrapContextAware | Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware ApplicationContexts | [JCA CCI](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/integration.html#cci) |
| LoadTimeWeaverAware | Defined *weaver* for processing class definition at load time | [Load-time weaving with AspectJ in the Spring Framework](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#aop-aj-ltw) |
| MessageSourceAware | Configured strategy for resolving messages (with support for parametrization and internationalization) | [Additional capabilities of the ApplicationContext](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#context-introduction) |
| NotificationPublisherAware | Spring JMX notification publisher | [Notifications](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/integration.html#jmx-notifications) |
| ResourceLoaderAware | Configured loader for low-level access to resources | [Resources](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#resources) |
| ServletConfigAware | Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext | [Spring MVC](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web.html#mvc) |
| ServletContextAware | Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext | [Spring MVC](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web.html#mvc) |

مجدداً توجه داشته باشید که استفاده از این رابط‌ها کد شما را به Spring API مرتبط می‌کند و از سبک Inversion of Control پیروی نمی‌کند. به این ترتیب، آنها برای دانه های زیرساختی که نیاز به دسترسی برنامه ای به کانتینر دارند توصیه می شوند.

## 1.7. Bean definition inheritance

تعریف bean می‌تواند حاوی اطلاعات پیکربندی زیادی باشد، از جمله آرگومان‌های سازنده، مقادیر ویژگی‌ها و اطلاعات خاص ظرف مانند روش اولیه، نام روش کارخانه ایستا و غیره. یک تعریف child، داده‌های پیکربندی را از تعریف والد به ارث می‌برد. تعریف فرزند می تواند برخی از مقادیر را نادیده بگیرد، یا در صورت نیاز، مقادیر دیگری را اضافه کند. استفاده از تعاریف والد و فرزند می تواند در تایپ کردن بسیار صرفه جویی کند. در واقع، این یک شکل از قالب است.

اگر با یک رابط ApplicationContext به صورت برنامه‌نویسی کار می‌کنید، تعاریف کودک با کلاس ChildBeanDefinition نمایش داده می‌شوند. اکثر کاربران در این سطح با آنها کار نمی کنند، در عوض تعاریف bean را به صورت اعلامی در چیزی مانند ClassPathXmlApplicationContext پیکربندی می کنند. هنگامی که از فراداده پیکربندی مبتنی بر XML استفاده می‌کنید، با استفاده از ویژگی والد، تعریف دانه‌های فرزند را نشان می‌دهید، و دانه والد را به عنوان مقدار این ویژگی مشخص می‌کنید.

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name" value="override"/>

*<!-- the age property value of 1 will be inherited from parent -->*

</bean>

یک تعریف فرزند از کلاس bean از تعریف والد استفاده می کند، اگر هیچ کدام مشخص نشده باشد، اما می تواند آن را لغو کند. در حالت دوم، کلاس child bean باید با والد سازگار باشد، یعنی باید مقادیر ویژگی والدین را بپذیرد.

یک تعریف کودک، دامنه، مقادیر آرگومان سازنده، مقادیر ویژگی و روش را از والد به ارث می برد، با گزینه ای برای افزودن مقادیر جدید. هر دامنه، روش مقداردهی اولیه، روش تخریب و/یا تنظیمات روش کارخانه ایستا که مشخص می کنید، تنظیمات والد مربوطه را لغو می کند.

تنظیمات باقی‌مانده همیشه از تعریف فرزند گرفته می‌شوند: بسته به، حالت سیم‌کشی خودکار، بررسی وابستگی، تک‌تن، شروع تنبل.

مثال قبلی با استفاده از ویژگی abstract به صراحت تعریف bean والد را به عنوان انتزاعی مشخص می کند. اگر تعریف والد کلاسی را مشخص نکرده باشد، علامت گذاری صریح تعریف والد به عنوان انتزاعی ضروری است، به شرح زیر:

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name" value="override"/>

*<!-- age will inherit the value of 1 from the parent bean definition-->*

</bean>

لوبیای والد نمی‌تواند به تنهایی نمونه‌سازی شود، زیرا ناقص است، و همچنین به‌صراحت به‌عنوان انتزاعی علامت‌گذاری شده است. هنگامی که یک تعریف مانند این انتزاعی است، تنها به عنوان یک تعریف لوبیای الگوی خالص که به عنوان تعریف والد برای تعاریف فرزند عمل می کند، قابل استفاده است. تلاش برای استفاده از چنین Bean انتزاعی به تنهایی، با ارجاع به آن به عنوان یک ویژگی ref bean دیگر یا انجام یک فراخوانی واضح getBean() با شناسه bean والد، یک خطا برمی‌گرداند. به طور مشابه، متد داخلی ()preInstantiateSingletons از تعاریف bean که به صورت انتزاعی تعریف شده‌اند، چشم پوشی می‌کند.

ApplicationContext به صورت پیش‌فرض تمام تک‌تون‌ها را از قبل نمونه‌سازی می‌کند. بنابراین، مهم است (حداقل برای لوبیاهای سینگلتون) که اگر یک تعریف bean (والد) دارید که می‌خواهید فقط به عنوان الگو از آن استفاده کنید و این تعریف یک کلاس را مشخص می‌کند، باید مطمئن شوید که ویژگی abstract را روی true تنظیم کنید. ، در غیر این صورت applicationContextدر واقع (تلاش برای) از پیش نمونه برداری از bean انتزاعی خواهد بود.

## 1.8. Container Extension Points

به طور معمول، یک توسعه‌دهنده برنامه نیازی به زیر کلاس‌بندی کلاس‌های پیاده‌سازی ApplicationContext ندارد. در عوض، کانتینر Spring IoC را می توان با وصل کردن پیاده سازی های رابط های یکپارچه سازی ویژه گسترش داد. چند بخش بعدی این رابط های یکپارچه سازی را شرح می دهد.

#### 1.8.1. Customizing beans using a BeanPostProcessor

رابط BeanPostProcessor روش‌هایی را تعریف می‌کند که می‌توانید برای ارائه منطق نمونه‌سازی، منطق وابستگی-رزولوشن، و غیره پیاده‌سازی کنید (یا از پیش‌فرض کانتینر استفاده کنید). اگر می‌خواهید پس از اتمام نمونه‌سازی، پیکربندی و مقداردهی اولیه یک Bean کانتینر، برخی از منطق سفارشی را پیاده‌سازی کنید، می‌توانید یک یا چند پیاده‌سازی سفارشی BeanPostProcessor را وصل کنید.

شما می توانید چندین نمونه BeanPostProcessor را پیکربندی کنید و می توانید با تنظیم ویژگی order ترتیب اجرای این BeanPostProcessor را کنترل کنید. شما می توانید این ویژگی را تنها در صورتی تنظیم کنید که BeanPostProcessor رابط Ordered را پیاده سازی کند. اگر BeanPostProcessor خود را می نویسید، باید رابط Ordered را نیز در نظر بگیرید. برای جزئیات بیشتر، به جاوادوکس رابط های BeanPostProcessor و Ordered مراجعه کنید. همچنین به یادداشت زیر در مورد ثبت برنامه‌ای BeanPostProcessors مراجعه کنید.

BeanPostProcessors بر روی نمونه های bean (یا شی) عمل می کنند. یعنی محفظه Spring IoC یک نمونه bean را نمونه سازی می کند و سپس BeanPostProcessors کار خود را انجام می دهند.

BeanPostProcessors در هر ظرف محدوده هستند. این تنها زمانی مرتبط است که از سلسله مراتب کانتینر استفاده می کنید. اگر BeanPostProcessor را در یک کانتینر تعریف کنید، فقط دانه‌های موجود در آن ظرف را post-processed می‌کند. به عبارت دیگر، دانه هایی که در یک ظرف تعریف می شوند توسط BeanPostProcessor تعریف شده در ظرف دیگری post-processedنمی شوند، حتی اگر هر دو کانتینر بخشی از یک سلسله مراتب باشند.

برای تغییر تعریف واقعی bean (یعنی طرحی که bean را تعریف می‌کند)، در عوض باید از یک BeanFactoryPostProcessor همانطور که در سفارشی کردن فراداده پیکربندی با BeanFactoryPostProcessor توضیح داده شده است استفاده کنید.

رابط org.springframework.beans.factory.config.BeanPostProcessor دقیقاً از دو روش callbackتشکیل شده است. وقتی چنین کلاسی به‌عنوان یک post-processor همراه container ثبت می‌شود، برای هر نمونه bean که توسط کانتینر ایجاد می‌شود، post-processor هم قبل از روش‌های مقداردهی اولیه (مانند InitializingBean's afterPropertiesSet() یا هر یک از این کانتینر یک فراخوانی دریافت می‌کند. روش init اعلام شده) و همچنین پس از هر تماس اولیه bean فراخوانی می شوند. پس پردازشگر می تواند هر اقدامی را با نمونه bean انجام دهد، از جمله نادیده گرفتن callback method کامل. یک post-processor bean معمولاً واسط‌های برگشت تماس را بررسی می‌کند یا ممکن است یک bean را با یک پروکسی بپیچد. برخی از کلاس‌های زیرساخت Spring AOP به‌عنوان post-processor bean به منظور ارائه منطق بسته‌بندی پروکسی پیاده‌سازی می‌شوند.

یک ApplicationContext به طور خودکار هر دانه‌هایی را که در فراداده‌های پیکربندی که رابط BeanPostProcessor را پیاده‌سازی می‌کنند، شناسایی می‌کند. ApplicationContext این دانه‌ها را به‌عنوان post-processor ثبت می‌کند تا بعداً پس از ایجاد bean بتوان آنها را فراخوانی کرد. post-processor لوبیا را می توان مانند سایر دانه ها در ظرف مستقر کرد.

توجه داشته باشید که هنگام اعلان BeanPostProcessor با استفاده از روش کارخانه @Bean در یک کلاس پیکربندی، نوع برگشتی متد کارخانه باید خود کلاس پیاده سازی یا حداقل رابط org.springframework.beans.factory.config.BeanPostProcessor باشد که به وضوح نشان دهنده ماهیت post-processor آن bean در غیر این صورت، ApplicationContext نمی‌تواند قبل از ایجاد کامل آن را بر اساس نوع آن شناسایی کند. از آنجایی که یک BeanPostProcessor برای اعمال اولیه‌سازی سایر دانه‌ها در context، نیاز به نمونه‌سازی اولیه دارد، این تشخیص نوع اولیه بسیار مهم است.

BeanPostProcessors و AOP Auto-Proxying

کلاس هایی که رابط BeanPostProcessor را پیاده سازی می کنند خاص هستند و توسط کانتینر با آنها متفاوت رفتار می شود. تمام BeanPostProcessor ها و bean هایی که مستقیماً به آنها ارجاع می دهند، در هنگام راه اندازی، به عنوان بخشی از مرحله راه اندازی ویژه ApplicationContext، نمونه سازی می شوند. در مرحله بعد، تمام BeanPostProcessor ها به صورت مرتب شده ثبت می شوند و برای تمام دانه های دیگر در ظرف اعمال می شوند. از آنجایی که پروکسی خودکار AOP به عنوان خود BeanPostProcessor پیاده‌سازی می‌شود، نه BeanPostProcessor و نه Beanهایی که مستقیماً به آنها ارجاع می‌دهند واجد شرایط پروکسی خودکار نیستند و بنابراین جنبه‌هایی در آنها وجود ندارد.

برای هر گونه چنین bean، باید یک پیام گزارش اطلاعاتی را مشاهده کنید: "Bean foo برای پردازش توسط تمام رابط های BeanPostProcessor واجد شرایط نیست (به عنوان مثال: واجد شرایط برای پروکسی خودکار نیست)".

توجه داشته باشید که اگر با استفاده از سیم‌کشی خودکار یا @Resource (که ممکن است به سیم‌کشی خودکار بازگردد)، Bean به BeanPostProcessor خود متصل کرده‌اید، ممکن است Spring هنگام جستجوی نامزدهای وابستگی مطابق با نوع، به دانه‌های غیرمنتظره دسترسی پیدا کند و بنابراین آنها را برای پروکسی خودکار یا سایر انواع دیگر واجد شرایط نمی‌کند. لوبیا پس از پردازش به عنوان مثال، اگر یک وابستگی مشروح شده با @Resource داشته باشید که در آن نام فیلد/تنظیم کننده مستقیماً با نام اعلام شده یک bean مطابقت ندارد و از ویژگی name استفاده نمی شود، آنگاه Spring برای تطبیق آنها بر اساس نوع، به سایر دانه ها دسترسی خواهد داشت.

مثال‌های زیر نحوه نوشتن، ثبت و استفاده از BeanPostProcessors را در یک ApplicationContext نشان می‌دهند.

##### Example: Hello World, BeanPostProcessor-style

این مثال اول کاربرد اساسی را نشان می دهد. این مثال یک پیاده‌سازی سفارشی BeanPostProcessor را نشان می‌دهد که متد toString() هر bean را همانطور که توسط کانتینر ایجاد می‌شود فراخوانی می‌کند و رشته حاصل را در کنسول سیستم چاپ می‌کند.

در زیر تعریف کلاس پیاده سازی سفارشی BeanPostProcessor را بیابید:

**package** scripting;

**import** org.springframework.beans.factory.config.BeanPostProcessor;

**public** **class** **InstantiationTracingBeanPostProcessor** **implements** BeanPostProcessor {

*// simply return the instantiated bean as-is*

**public** Object postProcessBeforeInitialization(Object bean, String beanName) {

**return** bean; *// we could potentially return any object reference here...*

}

**public** Object postProcessAfterInitialization(Object bean, String beanName) {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

**return** bean;

}

}

<?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:lang="http://www.springframework.org/schema/lang"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/lang/spring-lang.xsd">

<lang:groovy id="messenger"

script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

<lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>

</lang:groovy>

*<!--*

*when the above bean (messenger) is instantiated, this custom*

*BeanPostProcessor implementation will output the fact to the system console*

*-->*

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

توجه کنید که چگونه InstantiationTracingBeanPostProcessor به سادگی تعریف شده است. حتی نامی هم ندارد و چون لوبیا است می‌توان آن را مانند هر لوبیا دیگری تزریق کرد. (پیکربندی قبلی همچنین یک Bean را تعریف می‌کند که توسط یک اسکریپت Groovy پشتیبانی می‌شود. پشتیبانی از زبان پویا Spring در فصلی با عنوان پشتیبانی از زبان پویا توضیح داده شده است.)

برنامه ساده جاوا زیر کد و پیکربندی قبلی را اجرا می کند:

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** org.springframework.scripting.Messenger;

**public** **final** **class** **Boot** {

**public** **static** **void** main(**final** String**[]** args) **throws** Exception {

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("scripting/beans.xml");

Messenger messenger = (Messenger) ctx.getBean("messenger");

System.out.println(messenger);

}

}

خروجی برنامه قبلی شبیه موارد زیر است:

Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961

org.springframework.scripting.groovy.GroovyMessenger@272961

##### Example: The RequiredAnnotationBeanPostProcessor

استفاده از واسط های پاسخ به تماس یا حاشیه نویسی در ارتباط با پیاده سازی BeanPostProcessor سفارشی، وسیله ای رایج برای گسترش کانتینر Spring IoC است. یک مثال Spring’s RequiredAnnotationBeanPostProcessor است - یک پیاده‌سازی BeanPostProcessor که با توزیع Spring ارسال می‌شود و تضمین می‌کند که ویژگی‌های JavaBean روی دانه‌هایی که با حاشیه‌نویسی (خودسرانه) مشخص شده‌اند، در واقع (پیکربندی شده‌اند) وابستگی با یک مقدار تزریق می‌شوند.

#### 1.8.2. Customizing configuration metadata with a BeanFactoryPostProcessor

نقطه گسترش بعدی که به آن نگاه خواهیم کرد org.springframework.beans.factory.config.BeanFactoryPostProcessor است. معنایی این رابط مشابه با BeanPostProcessor است، با یک تفاوت عمده: BeanFactoryPostProcessor بر روی متادیتای پیکربندی bean عمل می کند. یعنی محفظه Spring IoC به BeanFactoryPostProcessor اجازه می‌دهد تا ابرداده‌های پیکربندی را بخواند و به طور بالقوه آن را تغییر دهد، قبل از اینکه کانتینر هر گونه beanغیر از BeanFactoryPostProcessors را نمونه‌سازی کند.

می‌توانید چندین BeanFactoryPostProcessor را پیکربندی کنید و می‌توانید ترتیب اجرای این BeanFactoryPostProcessors را با تنظیم ویژگی order کنترل کنید. با این حال، تنها در صورتی می توانید این ویژگی را تنظیم کنید که BeanFactoryPostProcessor رابط Ordered را پیاده سازی کند. اگر BeanFactoryPostProcessor خود را می نویسید، باید رابط Ordered را نیز پیاده سازی کنید. برای جزئیات بیشتر با جاوادوکس رابط های BeanFactoryPostProcessor و Ordered مشورت کنید.

اگر می‌خواهید نمونه‌های واقعی bean را تغییر دهید (یعنی اشیایی که از فراداده‌های پیکربندی ایجاد می‌شوند)، در عوض باید از BeanPostProcessor استفاده کنید (در بالا در Customizing beans با استفاده از BeanPostProcessor توضیح داده شد). در حالی که از نظر فنی می توان با نمونه های bean در یک BeanFactoryPostProcessor کار کرد (به عنوان مثال، با استفاده از BeanFactory.getBean())، انجام این کار باعث نمونه سازی زودرس bean می شود و چرخه عمر کانتینر استاندارد را نقض می کند. این ممکن است عوارض جانبی منفی مانند دور زدن لوبیا پس از پردازش ایجاد کند.

همچنین، BeanFactoryPostProcessors در هر ظرف محدوده هستند. این تنها زمانی مرتبط است که از سلسله مراتب کانتینر استفاده می کنید. اگر یک BeanFactoryPostProcessor را در یک ظرف تعریف کنید، فقط برای تعاریف bean در آن ظرف اعمال می شود. تعاریف Bean در یک کانتینر توسط BeanFactoryPostProcessors در ظرف دیگری پس پردازش نمی شوند، حتی اگر هر دو کانتینر بخشی از یک سلسله مراتب باشند.

یک post-processorکارخانه لوبیا زمانی که در داخل یک ApplicationContext اعلان می شود به صورت خودکار اجرا می شود تا تغییراتی را در متادیتای پیکربندی که ظرف را تعریف می کند اعمال کند. Spring شامل تعدادی از post-processorکارخانه لوبیا از پیش تعریف شده مانند PropertyOverrideConfigurer و PropertyPlaceholderConfigurer است. یک BeanFactoryPostProcessor سفارشی نیز می تواند مورد استفاده قرار گیرد، برای مثال، برای ثبت ویرایشگرهای دارایی سفارشی.

یک ApplicationContext به طور خودکار هر دانه ای را که در آن مستقر شده و رابط BeanFactoryPostProcessor را پیاده سازی می کند، شناسایی می کند. در زمان مناسب از این دانه ها به عنوان پس پردازشگرهای کارخانه لوبیا استفاده می کند. شما می توانید این دانه های پس از پردازش را مانند هر لوبیای دیگری به کار ببرید.

یک پس پردازشگر کارخانه لوبیا زمانی که در داخل یک ApplicationContext اعلان می شود به صورت خودکار اجرا می شود تا تغییراتی را در متادیتای پیکربندی که ظرف را تعریف می کند اعمال کند. Spring شامل تعدادی از پیش پردازشگرهای کارخانه لوبیا از پیش تعریف شده مانند PropertyOverrideConfigurer و PropertyPlaceholderConfigurer است. یک BeanFactoryPostProcessor سفارشی نیز می تواند مورد استفاده قرار گیرد، برای مثال، برای ثبت ویرایشگرهای دارایی سفارشی.

یک ApplicationContext به طور خودکار هر دانه ای را که در آن مستقر شده و رابط BeanFactoryPostProcessor را پیاده سازی می کند، شناسایی می کند. در زمان مناسب از این دانه ها به عنوان پس پردازشگرهای کارخانه لوبیا استفاده می کند. شما می توانید این دانه های پس از پردازش را مانند هر لوبیای دیگری به کار ببرید.

همانند BeanPostProcessors، معمولاً نمی خواهید BeanFactoryPostProcessors را برای مقداردهی اولیه تنبل پیکربندی کنید. اگر هیچ bean دیگری به Bean(Factory)PostProcessor ارجاع ندهد، آن پس پردازشگر اصلاً نمونه سازی نمی شود. بنابراین، علامت گذاری آن برای مقداردهی اولیه تنبل نادیده گرفته می شود و Bean(Factory)PostProcessor مشتاقانه نمونه سازی می شود، حتی اگر ویژگی default-lazy-init را روی true در اعلان عنصر <beans /> خود تنظیم کنید.

##### Example: the Class name substitution PropertyPlaceholderConfigurer

شما از PropertyPlaceholderConfigurer برای بیرونی کردن مقادیر ویژگی از یک تعریف bean در یک فایل جداگانه با استفاده از فرمت استاندارد Java Properties استفاده می کنید. انجام این کار به شخصی که یک برنامه کاربردی را راه‌اندازی می‌کند قادر می‌سازد تا ویژگی‌های محیطی خاص مانند URLهای پایگاه داده و گذرواژه‌ها را بدون پیچیدگی یا خطر تغییر فایل یا فایل‌های تعریف XML برای ظرف، سفارشی کند.

قطعه فراداده پیکربندی مبتنی بر XML زیر را در نظر بگیرید، جایی که یک DataSource با مقادیر مکان‌نما تعریف شده است. مثال ویژگی های پیکربندی شده از یک فایل Properties خارجی را نشان می دهد. در زمان اجرا، یک PropertyPlaceholderConfigurer به ابرداده اعمال می شود که جایگزین برخی از ویژگی های DataSource می شود. مقادیری که باید جایگزین شوند به عنوان متغیرهایی از فرم ${property-name} که از سبک Ant / log4j / JSP EL پیروی می کند، مشخص می شوند.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

مقادیر واقعی از فایل دیگری در قالب استاندارد Java Properties آمده است:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

بنابراین، رشته ${jdbc.username} در زمان اجرا با مقدار 'sa' جایگزین می‌شود، و همین امر برای سایر مقادیر متغیرهایی که با کلیدهای موجود در فایل خواص مطابقت دارند، صدق می‌کند. PropertyPlaceholderConfigurer وجود متغیرهایی را در اکثر ویژگی ها و ویژگی های یک تعریف bean بررسی می کند. علاوه بر این، پیشوند و پسوند مکان نگهدار را می توان سفارشی کرد.

با فضای نام متنی که در Spring 2.5 معرفی شد، می توان مکان نگهدارنده های ویژگی را با یک عنصر پیکربندی اختصاصی پیکربندی کرد. یک یا چند مکان را می توان به عنوان یک لیست جدا شده با کاما در ویژگی مکان ارائه کرد.

<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

PropertyPlaceholderConfigurer نه تنها به دنبال خواص در فایل Properties که شما مشخص کرده اید می گردد. به‌طور پیش‌فرض، اگر نتواند خاصیتی را در فایل‌های ویژگی‌های مشخص شده پیدا کند، ویژگی‌های سیستم جاوا را نیز بررسی می‌کند. می توانید این رفتار را با تنظیم ویژگی systemPropertiesMode پیکربندی با یکی از سه مقدار صحیح پشتیبانی شده زیر سفارشی کنید:

never (0): هرگز ویژگی های سیستم را بررسی نکنید

بازگشتی (1): اگر در فایل‌های ویژگی‌های مشخص شده قابل حل نیست، ویژگی‌های سیستم را بررسی کنید. این پیش فرض است.

override (2): ابتدا ویژگی های سیستم را بررسی کنید، قبل از اینکه فایل های ویژگی های مشخص شده را امتحان کنید. این به ویژگی های سیستم اجازه می دهد تا هر منبع ویژگی دیگری را نادیده بگیرند.

برای اطلاعات بیشتر با PropertyPlaceholderConfigurer javadocs مشورت کنید.

می‌توانید از PropertyPlaceholderConfigurer برای جایگزینی نام کلاس‌ها استفاده کنید، که گاهی اوقات برای انتخاب یک کلاس پیاده‌سازی خاص در زمان اجرا مفید است. مثلا:

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>classpath:com/foo/strategy.properties</value>

</property>

<property name="properties">

<value>custom.strategy.class=com.foo.DefaultStrategy</value>

</property>

</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>

اگر کلاس را نتوان در زمان اجرا به یک کلاس معتبر حل کرد، وضوح bean زمانی که در شرف ایجاد است، شکست می خورد، که در مرحله preInstantiateSingletons() یک ApplicationContext برای یک bean غیر تنبلی است.

##### Example: the PropertyOverrideConfigurer

PropertyOverrideConfigurer، یکی دیگر از پس پردازشگرهای کارخانه bean، شبیه PropertyPlaceholderConfigurer است، اما برخلاف دومی، تعاریف اصلی می توانند مقادیر پیش فرض داشته باشند یا اصلاً مقادیری برای ویژگی های bean نداشته باشند. اگر یک فایل Properties دارای ورودی برای خاصیت bean خاصی نباشد، از تعریف زمینه پیش‌فرض استفاده می‌شود.

توجه داشته باشید که تعریف bean از override شدن آگاه نیست، بنابراین بلافاصله از فایل تعریف XML مشخص نیست که از پیکربندی override استفاده شده است. در مورد چندین نمونه PropertyOverrideConfigurer که مقادیر متفاوتی را برای یک ویژگی bean تعریف می‌کنند، آخرین مورد برنده می‌شود، به دلیل مکانیسم overriding.

خطوط پیکربندی فایل Properties این فرمت را دارند:

beanName.property=value

مثلا:

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

این فایل مثال را می توان با یک تعریف ظرف استفاده کرد که حاوی یک Bean به نام dataSource است که دارای ویژگی های درایور و URL است.

تا زمانی که هر مؤلفه path به جز ویژگی final که باطل می شود، از قبل غیر تهی باشد (احتمالاً توسط سازنده ها مقداردهی اولیه شده است) نام ویژگی های مرکب نیز پشتیبانی می شود. در این مثال…

foo.fred.bob.sammy=123

خاصیت سامی خاصیت bob از خاصیت fred از foo bean روی مقدار اسکالر 123 تنظیم شده است.

مقادیر override مشخص شده همیشه مقادیر literal هستند. آنها به bean refrence ترجمه نمی شوند. این قرارداد همچنین زمانی اعمال می شود که مقدار اصلی در تعریف XML bean یک مرجع bean را مشخص کند.

با فضای نام متنی که در Spring 2.5 معرفی شد، می‌توان با یک عنصر پیکربندی اختصاصی، ویژگی‌های overriding را پیکربندی کرد:

<context:property-override location="classpath:override.properties"/>

#### 1.8.3. Customizing instantiation logic with a FactoryBean

رابط org.springframework.beans.factory.FactoryBean را برای اشیایی که خودشان کارخانه هستند پیاده سازی کنید.

رابط FactoryBean یک نقطه قابل اتصال به منطق نمونه سازی کانتینر Spring IoC است. اگر کد اولیه پیچیده ای دارید که در جاوا بهتر بیان می شود و در مقابل مقدار (به طور بالقوه) پرمخاطب XML، می توانید FactoryBean خود را ایجاد کنید، مقدار اولیه اولیه پیچیده را در آن کلاس بنویسید و سپس FactoryBean سفارشی خود را به ظرف متصل کنید.

رابط FactoryBean سه روش را ارائه می دهد:

Object getObject(): نمونه ای از آبجکتی را که این کارخانه ایجاد می کند برمی گرداند. بسته به اینکه این کارخانه تک‌تون‌ها یا نمونه‌های اولیه را برمی‌گرداند، می‌توان این نمونه را به اشتراک گذاشت.

boolean isSingleton(): true را برمی گرداند اگر FactoryBean تک تن ها را برگرداند، در غیر این صورت false.

کلاس getObjectType(): نوع شی را که توسط متد getObject() برگردانده شده برمی گرداند یا اگر نوع از قبل شناخته شده نباشد null را برمی گرداند.

مفهوم و رابط FactoryBean در تعدادی از مکان‌ها در چارچوب Spring استفاده می‌شود. بیش از 50 پیاده سازی رابط FactoryBean با خود Spring ارسال می شود.

هنگامی که باید از یک ظرف بخواهید یک نمونه واقعی FactoryBean را به جای bean که تولید می کند، بخواهید، هنگام فراخوانی متد getBean() در ApplicationContext، علامت علامت (&) را با علامت علامت (&) وارد کنید. بنابراین برای یک FactoryBean معین با شناسه myBean، فراخوانی getBean ("myBean") در ظرف، محصول FactoryBean را برمی‌گرداند. در حالی که فراخوانی getBean ("&myBean") خود نمونه FactoryBean را برمی گرداند.

## 1.9. Annotation-based container configuration

Are annotations better than XML for configuring Spring?

معرفی پیکربندی‌های مبتنی بر حاشیه‌نویسی این سوال را مطرح کرد که آیا این رویکرد بهتر از XML است یا خیر. پاسخ کوتاه این است که بستگی دارد. پاسخ طولانی این است که هر رویکردی مزایا و معایب خود را دارد و معمولاً این به توسعه‌دهنده بستگی دارد که تصمیم بگیرد کدام استراتژی برای آنها مناسب‌تر است. به دلیل نحوه تعریف آنها، حاشیه نویسی ها زمینه زیادی را در اعلان خود ارائه می دهند که منجر به پیکربندی کوتاه تر و مختصرتر می شود. با این حال، XML در سیم کشی اجزای بدون دست زدن به کد منبع یا کامپایل مجدد آنها برتری دارد. برخی از توسعه دهندگان ترجیح می دهند سیم کشی را نزدیک منبع داشته باشند در حالی که برخی دیگر استدلال می کنند که کلاس های حاشیه نویسی دیگر POJO نیستند و علاوه بر این، پیکربندی غیرمتمرکز می شود و کنترل آن سخت تر می شود.

صرف نظر از انتخاب، بهار می تواند هر دو سبک را در خود جای دهد و حتی آنها را با هم ترکیب کند. شایان ذکر است که Spring از طریق گزینه JavaConfig خود اجازه می دهد تا از حاشیه نویسی ها به روشی غیرتهاجمی و بدون دست زدن به کد منبع اجزای هدف استفاده شود و از نظر ابزار، همه سبک های پیکربندی توسط Spring Tool Suite پشتیبانی می شوند.

Annotation injection قبل از تزریق XML انجام می‌شود، بنابراین پیکربندی دوم برای ویژگی‌هایی که از طریق هر دو رویکرد سیم‌کشی شده‌اند، مورد استفاده قرار می‌گیرد.

مثل همیشه، می‌توانید آن‌ها را به‌عنوان تعاریف منحصربه‌فرد bean ثبت کنید، اما می‌توان آنها را به طور ضمنی با گنجاندن تگ زیر در یک پیکربندی Spring مبتنی بر XML ثبت کرد (به گنجاندن فضای نام زمینه توجه کنید):

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

</beans>

(پس پردازشگرهای ثبت شده ضمنی عبارتند از AutowiredAnnotationBeanPostProcessor، CommonAnnotationBeanPostProcessor، PersistenceAnnotationBeanPostProcessor، و همچنین RequiredAnnotationBeanPostProcessor فوق الذکر.)

<context:annotation-config/> فقط به دنبال حاشیه نویسی روی bean در همان زمینه برنامه ای است که در آن تعریف شده است. این بدان معناست که اگر <context:annotation-config/> را در یک WebApplicationContext برای DispatcherServlet قرار دهید، فقط @Autowired bean را در کنترلرهای شما بررسی می کند، نه خدمات شما را. برای اطلاعات بیشتر به The DispatcherServlet مراجعه کنید.

#### 1.9.1. @Required

حاشیه‌نویسی @Required برای setter methods ویژگی bean اعمال می‌شود، مانند مثال زیر:

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Required

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

این حاشیه نویسی به سادگی نشان می دهد که ویژگی bean تحت تأثیر باید در زمان پیکربندی، از طریق یک مقدار مشخصه مشخصه در تعریف bean یا از طریق سیم کشی خودکار، پر شود. اگر خاصیت باقالی آسیب دیده پر نشده باشد، ظرف یک استثنا ایجاد می کند. این اجازه می دهد تا برای شکست مشتاق و صریح، اجتناب از NullPointerExceptions یا مانند آن در آینده. همچنان توصیه می شود که اظهارات را در خود کلاس bean قرار دهید، به عنوان مثال، در یک متد init. انجام این کار آن ارجاعات و مقادیر مورد نیاز را حتی زمانی که از کلاس خارج از یک کانتینر استفاده می کنید، اعمال می کند.

#### 1.9.2. @Autowired

حاشیه نویسی @Inject در JSR 330 می تواند به جای حاشیه نویسی @Autowired Spring در مثال های زیر استفاده شود. برای جزئیات بیشتر اینجا را ببینید.

می توانید حاشیه نویسی @Autowired را برای سازنده ها اعمال کنید:

**public** **class** **MovieRecommender** {

**private** **final** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

از Spring Framework 4.3، اگر Bean هدف فقط یک سازنده را برای شروع تعریف کند، حاشیه‌نویسی @Autowired در چنین سازنده‌ای دیگر ضروری نیست. با این حال، اگر چندین سازنده در دسترس باشد، حداقل باید یکی از آنها حاشیه نویسی شود تا به ظرف آموزش داده شود که از کدام یک استفاده کند.

همانطور که انتظار می رود، می توانید حاشیه نویسی @Autowired را برای روش های تنظیم کننده "سنتی" نیز اعمال کنید:

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Autowired

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

همچنین می‌توانید حاشیه‌نویسی را برای روش‌هایی با نام‌های دلخواه و/یا چندین آرگومان اعمال کنید:

**public** **class** **MovieRecommender** {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** **void** prepare(MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

می‌توانید @Autowired را در فیلدها نیز اعمال کنید و حتی آن را با سازنده‌ها ترکیب کنید:

**public** **class** **MovieRecommender** {

**private** **final** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**private** MovieCatalog movieCatalog;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

اطمینان حاصل کنید که اجزای هدف شما (مانند MovieCatalog، CustomerPreferenceDao) به طور پیوسته بر اساس نوعی که برای نقاط تزریق @Autowired-annotated خود استفاده می کنید، اعلام شده است. در غیر این صورت تزریق ممکن است به دلیل عدم تطابق نوع در زمان اجرا شکست بخورد.

برای دانه‌های تعریف‌شده با XML یا کلاس‌های مؤلفه که از طریق اسکن مسیر کلاس یافت می‌شوند، ظرف معمولاً نوع بتن را از قبل می‌شناسد. با این حال، برای روش های کارخانه @Bean، باید مطمئن شوید که نوع بازگشتی اعلام شده به اندازه کافی رسا است. برای مؤلفه‌هایی که چندین رابط را پیاده‌سازی می‌کنند یا برای مؤلفه‌هایی که بالقوه بر اساس نوع پیاده‌سازی‌شان به آنها اشاره می‌شود، مشخص‌ترین نوع بازگشت را در روش کارخانه خود در نظر بگیرید (حداقل به اندازه‌ای که نقاط تزریق مربوط به bean شما مشخص است).

همچنین می‌توان با افزودن حاشیه‌نویسی به یک فیلد یا روشی که آرایه‌ای از آن نوع را انتظار دارد، همه دانه‌های یک نوع خاص را از ApplicationContext ارائه کرد:

**public** **class** **MovieRecommender** {

@Autowired

**private** MovieCatalog**[]** movieCatalogs;

*// ...*

}

همین امر در مورد مجموعه های تایپ شده نیز صدق می کند:

**public** **class** **MovieRecommender** {

**private** Set<MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

دانه‌های هدف شما می‌توانند رابط org.springframework.core.Ordered را پیاده‌سازی کنند یا اگر می‌خواهید موارد موجود در آرایه یا لیست به ترتیب خاصی مرتب شوند، از @Order یا حاشیه‌نویسی استاندارد @Priority استفاده کنند. در غیر این صورت سفارش آنها از ترتیب ثبت تعاریف لوبیا هدف مربوطه در ظرف پیروی می کند.

حاشیه‌نویسی @Order ممکن است در سطح کلاس هدف اما همچنین در روش‌های @Bean اعلام شود، به طور بالقوه برای هر تعریف bean بسیار مجزا است (در صورت تعاریف متعدد با یک کلاس bean). مقادیر @Order ممکن است بر اولویت‌ها در نقاط تزریق تأثیر بگذارد، اما لطفاً توجه داشته باشید که آنها بر ترتیب راه‌اندازی تک‌تنه تأثیر نمی‌گذارند که یک نگرانی متعامد است که توسط روابط وابستگی و اعلان‌های @DependsOn تعیین می‌شود.

توجه داشته باشید که حاشیه نویسی استاندارد javax.annotation.Priority در سطح @Bean در دسترس نیست زیرا نمی توان آن را در روش ها اعلام کرد. معناشناسی آن را می توان از طریق مقادیر @Order در ترکیب با @Primary در هر نوع لوبیا مدل کرد.

حتی نقشه‌های تایپ‌شده را می‌توان به صورت خودکار سیم‌کشی کرد تا زمانی که نوع کلید مورد انتظار String باشد. مقادیر Map شامل تمام دانه‌ها از نوع مورد انتظار خواهد بود و کلیدها حاوی نام‌های bean مربوطه خواهند بود:

**public** **class** **MovieRecommender** {

**private** Map<String, MovieCatalog> movieCatalogs;

@Autowired

**public** **void** setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

*// ...*

}

به‌طور پیش‌فرض، سیم‌کشی خودکار هر زمان که دانه‌های کاندید صفر در دسترس باشد، از کار می‌افتد. رفتار پیش‌فرض این است که روش‌ها، سازنده‌ها و فیلدهای حاشیه‌نویسی را به‌عنوان نشان‌دهنده وابستگی‌های مورد نیاز در نظر بگیریم. این رفتار را می توان همانطور که در زیر نشان داده شده تغییر داد.

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Autowired(required = false)

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

فقط یک سازنده از هر کلاس bean می‌تواند @Autowired را با صفت مورد نیاز که روی true تنظیم شده است، اعلام کند، که نشان می‌دهد سازنده زمانی که به عنوان یک Spring bean استفاده می‌شود، autowire می‌شود. علاوه بر این، اگر مشخصه مورد نیاز روی true تنظیم شود، تنها یک سازنده ممکن است با @Autowired حاشیه نویسی شود. اگر چندین سازنده غیر ضروری حاشیه نویسی را اعلام کنند، آنها به عنوان کاندیدای سیم کشی خودکار در نظر گرفته می شوند. سازنده با بیشترین تعداد وابستگی که می توان با تطبیق دانه ها در ظرف اسپرینگ ارضا کرد، انتخاب می شود. اگر هیچ یک از نامزدها نتوانند راضی شوند، از سازنده اولیه/پیش‌فرض (در صورت وجود) استفاده می‌شود. اگر یک کلاس برای شروع فقط یک سازنده واحد را اعلام کند، همیشه از آن استفاده می شود، حتی اگر حاشیه نویسی نشده باشد. یک سازنده حاشیه نویسی لازم نیست عمومی باشد.

ویژگی مورد نیاز @Autowired روی حاشیه نویسی @Required در روش های تنظیم کننده توصیه می شود. ویژگی مورد نیاز نشان می دهد که ملک برای اهداف سیم کشی خودکار مورد نیاز نیست. اگر امکان سیم‌کشی خودکار وجود نداشته باشد، دارایی نادیده گرفته می‌شود. از سوی دیگر، @Required از این جهت قوی‌تر است که به هر وسیله‌ای که توسط کانتینر پشتیبانی می‌شود، ویژگی تنظیم را اعمال می‌کند. اگر مقداری تعریف نشده باشد، یک استثنای مربوطه مطرح می شود.

از طرف دیگر، می‌توانید ماهیت غیر ضروری یک وابستگی خاص را از طریق Java 8 java.util.Optional بیان کنید:

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

از Spring Framework 5.0، می‌توانید از حاشیه‌نویسی @Nullable (از هر نوع در هر بسته، به عنوان مثال javax.annotation.Nullable از JSR-305) استفاده کنید:

**public** **class** **SimpleMovieLister** {

@Autowired

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

همچنین می‌توانید از @Autowired برای واسط‌هایی که وابستگی‌های قابل حل شناخته‌شده هستند استفاده کنید: BeanFactory، ApplicationContext، Environment، ResourceLoader، ApplicationEventPublisher و MessageSource. این رابط‌ها و رابط‌های توسعه‌یافته آن‌ها، مانند ConfigurableApplicationContext یا ResourcePatternResolver، به‌طور خودکار حل می‌شوند، بدون نیاز به تنظیم خاصی.

**public** **class** **MovieRecommender** {

@Autowired

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

حاشیه‌نویسی‌های @Autowired، @Inject، @Resource و @Value توسط پیاده‌سازی‌های Spring BeanPostProcessor مدیریت می‌شوند که به نوبه خود به این معنی است که شما نمی‌توانید این حاشیه‌نویسی‌ها را در انواع BeanPostProcessor یا BeanFactoryPostProcessor خود اعمال کنید (در صورت وجود). این انواع باید به صراحت از طریق XML یا با استفاده از روش Spring @Bean "سیم کشی شوند".

#### 1.9.3. Fine-tuning annotation-based autowiring with @Primary

از آنجا که سیم کشی خودکار بر اساس نوع ممکن است منجر به چندین نامزد شود، اغلب لازم است که کنترل بیشتری بر روند انتخاب داشته باشیم. یکی از راه‌های انجام این کار، حاشیه‌نویسی Spring @ Primary است. @Primary نشان می‌دهد که وقتی چندین bean کاندید سیم‌کشی خودکار به یک وابستگی تک ارزشی هستند، باید به یک bean خاص ترجیح داده شود. اگر دقیقاً یک دانه «اولیه» در میان نامزدها وجود داشته باشد، آن مقدار سیم‌کشی خودکار خواهد بود.

بیایید فرض کنیم پیکربندی زیر را داریم که firstMovieCatalog را به عنوان MovieCatalog اصلی تعریف می‌کند.

@Configuration

**public** **class** **MovieConfiguration** {

@Bean

**@Primary**

**public** MovieCatalog firstMovieCatalog() { ... }

@Bean

**public** MovieCatalog secondMovieCatalog() { ... }

*// ...*

}

با چنین پیکربندی، MovieRecommender زیر به صورت خودکار با firstMovieCatalog سیم‌کشی می‌شود.

**public** **class** **MovieRecommender** {

@Autowired

**private** MovieCatalog movieCatalog;

*// ...*

}

تعاریف مربوط به لوبیا به صورت زیر ظاهر می شود.

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog" primary="true">

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

#### 1.9.4. Fine-tuning annotation-based autowiring with qualifiers

@Primary یک روش موثر برای استفاده از سیم‌کشی خودکار بر اساس نوع با چندین نمونه است که یک نامزد اصلی را می‌توان تعیین کرد. هنگامی که کنترل بیشتری بر فرآیند انتخاب مورد نیاز است، می توان از حاشیه نویسی @Qualifier Spring استفاده کرد. می‌توانید مقادیر واجد شرایط را با آرگومان‌های خاص مرتبط کنید، مجموعه‌ای از تطابق‌های نوع را محدود کنید تا یک bean خاص برای هر آرگومان انتخاب شود. در ساده ترین حالت، این می تواند یک مقدار توصیفی ساده باشد:

**public** **class** **MovieRecommender** {

@Autowired

**@Qualifier("main")**

**private** MovieCatalog movieCatalog;

*// ...*

}

حاشیه نویسی @Qualifier همچنین می تواند بر روی آرگومان های سازنده مجزا یا پارامترهای متد مشخص شود:

**public** **class** **MovieRecommender** {

**private** MovieCatalog movieCatalog;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** **void** prepare(**@Qualifier("main")**MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

*// ...*

}

تعاریف مربوط به لوبیا به صورت زیر ظاهر می شود. bean با مقدار واجد شرایط "main" با آرگومان سازنده سیم‌کشی می‌شود که با همان مقدار واجد شرایط است.

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="main"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier value="action"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

برای تطبیق مجدد، نام bean یک مقدار مقدماتی پیش‌فرض در نظر گرفته می‌شود. بنابراین شما می توانید bean را با یک شناسه "main" به جای عنصر تعیین کننده تودرتو تعریف کنید، که منجر به همان نتیجه تطبیقی ​​می شود. با این حال، اگرچه می‌توانید از این قرارداد برای اشاره به دانه‌های خاص با نام استفاده کنید، @Autowired اساساً در مورد تزریق مبتنی بر نوع با واجد شرایط معنایی اختیاری است. این بدان معنی است که مقادیر واجد شرایط، حتی با نام bean backback، همیشه دارای معنای محدودکننده در مجموعه ای از مطابقت های نوع هستند. آنها از نظر معنایی ارجاع به یک شناسه منحصر به فرد bean را بیان نمی کنند. مقادیر واجد شرایط خوب «اصلی» یا «EMEA» یا «دائم» هستند، که ویژگی‌های یک مؤلفه خاص مستقل از شناسه bean را بیان می‌کنند، که ممکن است در صورت تعریف bean ناشناس مانند نمونه قبل، به‌طور خودکار ایجاد شود. .

واجد شرایط برای مجموعه‌های تایپ‌شده نیز اعمال می‌شود، همانطور که در بالا توضیح داده شد، برای مثال، برای Set<MovieCatalog>. در این حالت تمام دانه های همسان با توجه به صلاحیت های اعلام شده به صورت مجموعه تزریق می شوند. این بدان معناست که واجد شرایط لازم نیست منحصر به فرد باشند. آنها به سادگی معیارهای فیلتر را تشکیل می دهند. به عنوان مثال، می‌توانید چندین دانه MovieCatalog را با همان مقدار واجد شرایط "action" تعریف کنید، که همه آنها به یک Set<MovieCatalog> مشروح با @Qualifier("action") تزریق می‌شوند.

اجازه دادن به مقادیر واجد شرایط در مقابل نام‌های لوبیا هدف، در نامزدهای مطابق با نوع، حتی نیازی به حاشیه‌نویسی @Qualifier در نقطه تزریق ندارد. اگر هیچ نشانگر وضوح دیگری (به عنوان مثال یک واجد شرایط یا یک نشانگر اولیه) وجود نداشته باشد، برای یک وضعیت وابستگی غیر منحصر به فرد، Spring نام نقطه تزریق (یعنی نام فیلد یا نام پارامتر) را با نام‌های دانه هدف مطابقت می‌دهد و همان را انتخاب می‌کند. نامزد نامگذاری شده، در صورت وجود

گفته شد، اگر می‌خواهید تزریق مبتنی بر حاشیه‌نویسی را با نام بیان کنید، در اصل از @Autowired استفاده نکنید، حتی اگر قادر به انتخاب با نام bean در بین نامزدهای مطابق با نوع باشد. در عوض، از حاشیه نویسی JSR-250 @Resource استفاده کنید، که از نظر معنایی برای شناسایی یک جزء هدف خاص با نام منحصر به فردش تعریف شده است، با اینکه نوع اعلام شده برای فرآیند تطبیق نامربوط است. @Autowired دارای معنایی نسبتاً متفاوتی است: پس از انتخاب دانه‌های کاندید بر اساس نوع، مقدار تعیین‌شده String فقط در آن نامزدهای انتخاب شده در نظر گرفته می‌شود، به عنوان مثال. مطابقت یک واجد شرایط "حساب" با لوبیاهایی که با همان برچسب واجد شرایط مشخص شده اند.

برای دانه‌هایی که خودشان به‌عنوان مجموعه/نقشه یا نوع آرایه تعریف می‌شوند، @Resource راه‌حل خوبی است که به مجموعه یا آرایه‌ای خاص با نام منحصربه‌فرد اشاره می‌کند. گفتنی است، از 4.3، انواع مجموعه/نقشه و آرایه را می‌توان از طریق الگوریتم تطبیق نوع @Autowired نیز مطابقت داد، تا زمانی که اطلاعات نوع عنصر در امضاهای نوع بازگشتی @Bean یا سلسله‌مراتب وراثت مجموعه حفظ شود. در این مورد، همان طور که در پاراگراف قبلی ذکر شد، می توان از مقادیر واجد شرایط برای انتخاب از میان مجموعه های همتا استفاده کرد.

از 4.3، @Autowired همچنین ارجاعات خود را برای تزریق در نظر می گیرد، یعنی ارجاع به دانه ای که در حال حاضر تزریق شده است. توجه داشته باشید که خود تزریق یک بازگشت است. وابستگی های منظم به اجزای دیگر همیشه اولویت دارند. از این نظر، ارجاعات به خود در انتخاب نامزدهای منظم شرکت نمی‌کنند و به‌طور خاص هرگز اولیه نیستند. در مقابل، آنها همیشه به پایین ترین اولویت ختم می شوند. در عمل، از ارجاعات خود به عنوان آخرین راه حل استفاده کنید، به عنوان مثال. برای فراخوانی روش‌های دیگر در همان نمونه از طریق پراکسی تراکنش bean: در چنین سناریویی، روش‌های تحت‌تاثیر را در یک bean جداگانه در نظر بگیرید. از طرف دیگر، از @Resource استفاده کنید که ممکن است یک پروکسی به bean فعلی با نام منحصر به فرد خود دریافت کند.

@Autowired برای فیلدها، سازنده‌ها و روش‌های چند آرگومان اعمال می‌شود و امکان محدود کردن حاشیه‌نویسی واجد شرایط را در سطح پارامتر فراهم می‌کند. در مقابل، @Resource فقط برای فیلدها و روش‌های تنظیم‌کننده ویژگی bean با یک آرگومان واحد پشتیبانی می‌شود. در نتیجه، اگر هدف تزریق شما یک سازنده یا یک روش چند آرگومان است، از واجد شرایط استفاده کنید.

شما می توانید حاشیه نویسی واجد شرایط سفارشی خود را ایجاد کنید. به سادگی یک حاشیه نویسی تعریف کنید و حاشیه نویسی @Qualifier را در تعریف خود ارائه دهید:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

**@Qualifier**

**public** @interface Genre {

String value();

}

سپس می توانید qualifier سفارشی را در فیلدها و پارامترهای سیم کشی شده خودکار ارائه دهید:

**public** **class** **MovieRecommender** {

@Autowired

**@Genre("Action")**

**private** MovieCatalog actionCatalog;

**private** MovieCatalog comedyCatalog;

@Autowired

**public** **void** setComedyCatalog(**@Genre("Comedy")** MovieCatalog comedyCatalog) {

this.comedyCatalog = comedyCatalog;

}

*// ...*

}

سپس، اطلاعاتی را برای تعاریف لوبیا کاندید ارائه دهید. می توانید تگ های <qualifier/> را به عنوان عناصر فرعی تگ <bean/> اضافه کنید و سپس نوع و مقدار را برای مطابقت با حاشیه نویسی های واجد شرایط سفارشی خود تعیین کنید. نوع با نام کلاس کاملاً واجد شرایط حاشیه نویسی مطابقت دارد. یا اگر خطری برای نام‌های متضاد وجود ندارد، می‌توانید از نام کلاس کوتاه استفاده کنید. هر دو رویکرد در مثال زیر نشان داده شده است.

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Genre" value="Action"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

**<qualifier type="example.Genre" value="Comedy"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

در اسکن Classpath و اجزای مدیریت شده، جایگزینی مبتنی بر حاشیه نویسی برای ارائه فراداده واجد شرایط در XML خواهید دید. به طور خاص، به ارائه فراداده واجد شرایط با حاشیه نویسی مراجعه کنید.

در برخی موارد، ممکن است استفاده از حاشیه نویسی بدون مقدار کافی باشد. این ممکن است زمانی مفید باشد که حاشیه نویسی هدف کلی تری را دنبال می کند و می تواند در چندین نوع مختلف وابستگی اعمال شود. برای مثال، ممکن است یک کاتالوگ آفلاین ارائه کنید که در صورت عدم دسترسی به اینترنت جستجو شود. ابتدا یک حاشیه نویسی ساده را تعریف کنید:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface Offline {

}

سپس حاشیه نویسی را به فیلد یا ویژگی مورد نظر برای سیم کشی خودکار اضافه کنید:

**public** **class** **MovieRecommender** {

@Autowired

**@Offline**

**private** MovieCatalog offlineCatalog;

*// ...*

}

اکنون تعریف bean فقط به یک qualifier typeشرایط نیاز دارد:

<bean class="example.SimpleMovieCatalog">

**<qualifier type="Offline"/>**

*<!-- inject any dependencies required by this bean -->*

</bean>

همچنین می‌توانید حاشیه‌نویسی‌های واجد شرایط سفارشی را تعریف کنید که ویژگی‌های نام‌گذاری‌شده را علاوه بر یا به‌جای ویژگی مقدار ساده بپذیرند. اگر چندین مقدار مشخصه روی یک فیلد یا پارامتری که قرار است به صورت خودکار سیم‌کشی شود، مشخص شود، یک تعریف bean باید با همه این مقادیر مشخصه مطابقت داشته باشد تا به عنوان کاندید سیم‌کشی خودکار در نظر گرفته شود. به عنوان مثال، تعریف حاشیه نویسی زیر را در نظر بگیرید:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

**public** @interface MovieQualifier {

String genre();

Format format();

}

در این مورد Format یک enum است:

**public** **enum** Format {

VHS, DVD, BLURAY

}

فیلدهایی که قرار است به صورت خودکار سیم‌کشی شوند با custom qualifier حاشیه‌نویسی می‌شوند و شامل مقادیری برای هر دو ویژگی هستند: genre and format

**public** **class** **MovieRecommender** {

@Autowired

@MovieQualifier(format=Format.VHS, genre="Action")

**private** MovieCatalog actionVhsCatalog;

@Autowired

@MovieQualifier(format=Format.VHS, genre="Comedy")

**private** MovieCatalog comedyVhsCatalog;

@Autowired

@MovieQualifier(format=Format.DVD, genre="Action")

**private** MovieCatalog actionDvdCatalog;

@Autowired

@MovieQualifier(format=Format.BLURAY, genre="Comedy")

**private** MovieCatalog comedyBluRayCatalog;

*// ...*

}

در نهایت، تعاریف bean باید حاوی مقادیر واجد شرایط باشد. این مثال همچنین نشان می دهد که می توان از ویژگی های متا bean به جای عناصر فرعی <qualifier/> استفاده کرد. در صورت موجود بودن، <qualifier/> و ویژگی‌های آن اولویت دارند، اما مکانیزم سیم‌کشی خودکار بر روی مقادیر ارائه‌شده در تگ‌های <meta/> برمی‌گردد، در صورتی که چنین واجد شرایطی وجود نداشته باشد، مانند دو تعریف قبلی در مثال زیر.

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Action"/>

</qualifier>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

*<!-- inject any dependencies required by this bean -->*

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="BLURAY"/>

<meta key="genre" value="Comedy"/>

*<!-- inject any dependencies required by this bean -->*

</bean>

</beans>

#### 1.9.5. Using generics as autowiring qualifiers

علاوه بر حاشیه نویسی @Qualifier، همچنین می توان از انواع عمومی جاوا به عنوان فرم ضمنی صلاحیت استفاده کرد. برای مثال، فرض کنید تنظیمات زیر را دارید:

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**public** StringStore stringStore() {

**return** **new** StringStore();

}

@Bean

**public** IntegerStore integerStore() {

**return** **new** IntegerStore();

}

}

با فرض اینکه bean های بالا یک رابط عمومی را پیاده سازی می کنند، یعنی Store<String> و Store<Integer>، می توانید واسط Store @Autowire را انجام دهید و عمومی به عنوان یک واجد شرایط استفاده می شود:

@Autowired

**private** Store<String> s1; *// <String> qualifier, injects the stringStore bean*

@Autowired

**private** Store<Integer> s2; *// <Integer> qualifier, injects the integerStore bean*

در هنگام سیم‌کشی خودکار autowiring Lists, Maps and Arrays نیز اعمال می‌شود:

*// Inject all Store beans as long as they have an <Integer> generic*

*// Store<String> beans will not appear in this list*

@Autowired

**private** List<Store<Integer>> s;

#### 1.9.6. CustomAutowireConfigurer

CustomAutowireConfigurer یک BeanFactoryPostProcessor است که به شما امکان می دهد انواع custom qualifier annotation types خود را ثبت کنید، حتی اگر با حاشیه نویسی Spring @Qualifier حاشیه نویسی نشده باشند.

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

AutowireCandidateResolver کاندیداهای autowire را توسط:

• مقدار autowire-candidate هر تعریف لوبیا

• هر الگو(های) پیش فرض autowire-candidates موجود در عنصر <beans/>

• وجود حاشیه نویسی @Qualifier و هرگونه حاشیه نویسی سفارشی ثبت شده در CustomAutowireConfigurer

هنگامی که چندین دانه واجد شرایط به عنوان کاندید autowire هستند، تعیین یک "اولیه" به شرح زیر است: اگر دقیقاً یک تعریف bean در بین نامزدها دارای یک ویژگی اولیه باشد که روی true تنظیم شده باشد، انتخاب خواهد شد.

#### 1.9.7. @Resource

Spring همچنین از تزریق با استفاده از حاشیه نویسی JSR-250 @Resource در فیلدها یا روش های تنظیم کننده ویژگی bean پشتیبانی می کند. این یک الگوی رایج در جاوا EE 5 و 6 است، برای مثال در JSF 1.2 مدیریت لوبیا یا JAX-WS 2.0 endpoints. Spring از این الگو برای اشیاء مدیریت شده توسط Spring نیز پشتیبانی می کند.

@Resource یک ویژگی name را می گیرد و به طور پیش فرض Spring آن مقدار را به عنوان نام bean برای تزریق تفسیر می کند. به عبارت دیگر، همانطور که در این مثال نشان داده شده است، از معنای نامی پیروی می کند:

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

**@Resource(name="myMovieFinder")**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

اگر هیچ نامی به طور صریح مشخص نشده باشد، نام پیش‌فرض از نام فیلد یا روش تنظیم‌کننده مشتق شده است. در مورد فیلد، نام فیلد را می گیرد. در صورت وجود یک متد setter، نام ویژگی bean را می گیرد. بنابراین در مثال زیر، لوبیا با نام "movieFinder" به روش تنظیم کننده آن تزریق می شود:

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

**@Resource**

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

نام ارائه شده با حاشیه نویسی به عنوان یک نام لوبیا توسط ApplicationContext که CommonAnnotationBeanPostProcessor از آن آگاه است، حل می شود. اگر به طور صریح Spring's SimpleJndiBeanFactory را پیکربندی کنید، نام ها را می توان از طریق JNDI حل کرد. با این حال، توصیه می‌شود که به رفتار پیش‌فرض تکیه کنید و به سادگی از قابلیت‌های جستجو JNDI Spring برای حفظ سطح غیرمستقیم استفاده کنید.

در مورد انحصاری استفاده از @Resource بدون نام صریح و مشابه @Autowired، @Resource به جای یک bean با نام خاص، یک تطابق نوع اولیه پیدا می کند و وابستگی های قابل حل شناخته شده را حل می کند: BeanFactory، ApplicationContext، ResourceLoader، ApplicationEventPublisher، و رابط های MessageSource.

بنابراین در مثال زیر، فیلد customerPreferenceDao ابتدا به دنبال یک لوبیا با نام customerPreferenceDao می گردد، سپس به یک مطابقت نوع اولیه برای نوع CustomerPreferenceDao برمی گردد. فیلد "context" بر اساس نوع وابستگی قابل حل شناخته شده ApplicationContext تزریق می شود.

**public** **class** **MovieRecommender** {

@Resource

**private** CustomerPreferenceDao customerPreferenceDao;

@Resource

**private** ApplicationContext context;

**public** MovieRecommender() {

}

*// ...*

}

#### 1.9.8. @PostConstruct and @PreDestroy

CommonAnnotationBeanPostProcessor نه تنها حاشیه نویسی @Resource بلکه حاشیه نویسی چرخه حیات JSR-250 را نیز تشخیص می دهد. پشتیبانی از این حاشیه نویسی که در Spring 2.5 معرفی شد، جایگزین دیگری را برای مواردی که در تماس های اولیه و تماس های تخریبی توضیح داده شده است، ارائه می دهد. مشروط بر اینکه CommonAnnotationBeanPostProcessor در Spring ApplicationContext ثبت شده باشد، روشی که یکی از این حاشیه نویسی ها را حمل می کند در همان نقطه ای از چرخه حیات به عنوان روش رابط چرخه حیات Spring مربوطه یا روش برگشت فراخوانی به صراحت اعلام شده است. در مثال زیر، حافظه نهان پس از مقداردهی اولیه پر می شود و پس از تخریب پاک می شود.

**public** **class** **CachingMovieLister** {

@PostConstruct

**public** **void** populateMovieCache() {

*// populates the movie cache upon initialization...*

}

@PreDestroy

**public** **void** clearMovieCache() {

*// clears the movie cache upon destruction...*

}

}

### 1.10. Classpath scanning and managed components

بیشتر نمونه‌های این فصل از XML برای تعیین فراداده‌های پیکربندی که هر BeanDefinition را در ظرف Spring تولید می‌کند، استفاده می‌کنند. بخش قبلی (پیکربندی کانتینر مبتنی بر حاشیه‌نویسی) نشان می‌دهد که چگونه می‌توان بسیاری از فراداده‌های پیکربندی را از طریق حاشیه‌نویسی‌های سطح منبع ارائه کرد. با این حال، حتی در آن مثال‌ها، تعاریف bean "پایه" به صراحت در فایل XML تعریف شده‌اند، در حالی که حاشیه‌نویسی‌ها فقط تزریق وابستگی را هدایت می‌کنند. این بخش گزینه ای را برای شناسایی ضمنی اجزای کاندید با اسکن مسیر کلاس توضیح می دهد. اجزای کاندید کلاس هایی هستند که با معیارهای فیلتر مطابقت دارند و یک تعریف bean مربوطه در ظرف ثبت شده است. این نیاز به استفاده از XML برای انجام ثبت bean را از بین می برد. در عوض می‌توانید از حاشیه‌نویسی‌ها (به عنوان مثال @Component)، عبارات نوع AspectJ یا معیارهای فیلتر سفارشی خود استفاده کنید تا انتخاب کنید کدام کلاس‌ها دارای تعاریف bean ثبت شده در ظرف باشند.

با شروع Spring 3.0، بسیاری از ویژگی های ارائه شده توسط پروژه Spring JavaConfig بخشی از چارچوب اصلی Spring Framework هستند. این به شما این امکان را می‌دهد تا بجای استفاده از فایل‌های XML سنتی، beans را با استفاده از جاوا تعریف کنید. برای مثال‌هایی از نحوه استفاده از این ویژگی‌های جدید، به حاشیه‌نویسی‌های @Configuration، @Bean، @Import و @DependsOn نگاهی بیندازید.

#### 1.10.1. @Component and further stereotype annotations

حاشیه نویسی @Repository نشانگری برای هر کلاسی است که نقش یا کلیشه یک مخزن (همچنین به عنوان Data Access Object یا DAO شناخته می شود) را انجام می دهد. از جمله کاربردهای این نشانگر، ترجمه خودکار استثنائات است که در ترجمه Exception توضیح داده شده است.

Spring حاشیه نویسی های کلیشه ای بیشتری را ارائه می دهد: @Component، @Service، و @Controller. @Component یک کلیشه کلی برای هر مؤلفه ای است که توسط Spring مدیریت می شود. @Repository، @Service و @Controller تخصص های @Component برای موارد استفاده خاص تر هستند، به عنوان مثال، به ترتیب در لایه های ماندگاری، سرویس و ارائه. بنابراین، می‌توانید کلاس‌های کامپوننت خود را با @Component حاشیه‌نویسی کنید، اما با حاشیه‌نویسی آن‌ها با @Repository، @Service، یا @Controller، کلاس‌های شما برای پردازش توسط ابزارها یا ارتباط با جنبه‌ها مناسب‌تر هستند. به عنوان مثال، این حاشیه‌نویسی‌های کلیشه‌ای، اهداف ایده‌آلی برای نقاط برش داده می‌شوند. همچنین ممکن است که @Repository، @Service، و @Controller ممکن است معنای بیشتری را در نسخه‌های بعدی Spring Framework داشته باشند. بنابراین، اگر بین استفاده از @Component یا @Service برای لایه سرویس خود انتخاب می کنید، @Service به وضوح انتخاب بهتری است. به طور مشابه، همانطور که در بالا ذکر شد، @Repository قبلاً به عنوان نشانگری برای ترجمه استثنایی خودکار در لایه پایداری شما پشتیبانی می‌شود.

#### 1.10.2. Meta-annotations

بسیاری از حاشیه نویسی های ارائه شده توسط Spring می توانند به عنوان meta-annotation در کد خود استفاده شوند. یک meta-annotationصرفاً یک حاشیه نویسی است که می تواند برای حاشیه نویسی دیگر اعمال شود. به عنوان مثال، حاشیه‌نویسی service @ که در بالا ذکر شد، با @Component دارای حاشیه‌نویسی شده است:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**@Component** *// Spring will see this and treat @Service in the same way as @Component*

**public** @interface Service {

*// ....*

}

متا حاشیه‌نویسی‌ها را نیز می‌توان برای ایجاد حاشیه‌نویسی ترکیبی ترکیب کرد. به عنوان مثال، حاشیه نویسی @RestController از Spring MVC از @Controller و @ResponseBody تشکیل شده است.

علاوه بر این، حاشیه نویسی های ترکیبی ممکن است به صورت اختیاری ویژگی های meta-annotationsرا مجدداً اعلام کنند تا امکان سفارشی سازی کاربر فراهم شود. این می تواند به ویژه زمانی مفید باشد که بخواهید فقط زیرمجموعه ای از ویژگی های meta-annotations را در معرض دید قرار دهید. به عنوان مثال، حاشیه‌نویسی @SessionScope Spring، نام دامنه را به جلسه هارد کد می‌کند، اما همچنان اجازه سفارشی‌سازی حالت proxy را می‌دهد.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Scope(WebApplicationContext.SCOPE\_SESSION)

**public** @interface SessionScope {

*/\*\**

*\* Alias for {@link Scope#proxyMode}.*

*\* <p>Defaults to {@link ScopedProxyMode#TARGET\_CLASS}.*

*\*/*

@AliasFor(annotation = Scope.class)

ScopedProxyMode proxyMode() **default** ScopedProxyMode.TARGET\_CLASS;

}

سپس @SessionScope می‌تواند بدون اعلام حالت proxy به صورت زیر استفاده شود:

@Service

**@SessionScope**

**public** **class** **SessionScopedService** {

*// ...*

}

یا با یک مقدار override شده برای proxyMode به شرح زیر:

@Service

**@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)**

**public** **class** **SessionScopedUserService** **implements** UserService {

*// ...*

}

#### 1.10.3. Automatically detecting classes and registering bean definitions

Spring می تواند به طور خودکار کلاس های کلیشه ای را تشخیص دهد و BeanDefinition های مربوطه را با ApplicationContext ثبت کند. به عنوان مثال، دو کلاس زیر برای چنین تشخیص خودکار واجد شرایط هستند:

@Service

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Autowired

**public** SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

@Repository

**public** **class** **JpaMovieFinder** **implements** MovieFinder {

*// implementation elided for clarity*

}

برای شناسایی خودکار این کلاس ها و ثبت دانه های مربوطه، باید @ComponentScan را به کلاس Configuration@ خود اضافه کنید، جایی که ویژگی basePackages یک بسته والد مشترک برای دو کلاس است. (به طور متناوب، می توانید یک لیست با کاما/نقطه ویرگول/فضا جدا شده که شامل بسته والد هر کلاس است را مشخص کنید.)

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

زیر یک جایگزین با استفاده از XML است

<?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:context="http://www.springframework.org/schema/context"

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

https://www.springframework.org/schema/beans/spring-beans.xsd

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

https://www.springframework.org/schema/context/spring-context.xsd">

<context:component-scan base-package="org.example"/>

</beans>

استفاده از <context:component-scan> به طور ضمنی عملکرد <context:annotation-config> را فعال می کند. هنگام استفاده از <context:component-scan> معمولاً نیازی به اضافه کردن عنصر <context:annotation-config> نیست.

اسکن بسته های classpath به وجود ورودی های دایرکتوری مربوطه در classpath نیاز دارد. هنگامی که JAR ها را با Ant می سازید، مطمئن شوید که سوئیچ فقط فایل وظیفه JAR را فعال نکنید. همچنین، دایرکتوری های classpath ممکن است بر اساس سیاست های امنیتی در برخی محیط ها، به عنوان مثال، در معرض دید قرار نگیرند. برنامه‌های مستقل در JDK 1.7.0\_45 و بالاتر (که نیاز به راه‌اندازی «Trusted-Library» در مانیفست‌های شما دارد؛ https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources را ببینید).

در مسیر ماژول JDK 9 (Jigsaw)، اسکن مسیر کلاس Spring به طور کلی همانطور که انتظار می رود کار می کند. با این حال، لطفاً مطمئن شوید که کلاس‌های مؤلفه شما در توصیفگرهای اطلاعات ماژول شما صادر می‌شوند. اگر انتظار دارید Spring اعضای غیر عمومی کلاس های شما را فراخوانی کند، مطمئن شوید که آنها "باز" هستند (یعنی استفاده از یک اعلان باز به جای یک اعلان صادرات در توصیفگر اطلاعات ماژول).

علاوه بر این، AutowiredAnnotationBeanPostProcessor و CommonAnnotationBeanPostProcessor هر دو به طور ضمنی در هنگام استفاده از عنصر component-scan گنجانده می شوند. این بدان معناست که دو مؤلفه به صورت خودکار شناسایی و با هم سیم‌کشی می‌شوند - همه بدون هیچ گونه ابرداده پیکربندی bean ارائه شده در XML.

#### 1.10.4. Using filters to customize scanning

به‌طور پیش‌فرض، کلاس‌هایی که با @Component، @Repository، @Service، @Controller یا یک حاشیه‌نویسی سفارشی که خود با @Component حاشیه‌نویسی شده‌اند، تنها مؤلفه‌های کاندید شناسایی‌شده هستند. با این حال، شما می توانید این رفتار را به سادگی با اعمال فیلترهای سفارشی اصلاح و گسترش دهید. آنها را به عنوان پارامترهای includeFilters یا excludeFilters در حاشیه نویسی @ComponentScan (یا به عنوان عناصر فرعی include-filter یا exclude-filter عنصر جزء اسکن) اضافه کنید. هر عنصر فیلتر به ویژگی های نوع و عبارت نیاز دارد. جدول زیر گزینه های فیلترینگ را توضیح می دهد.

| *Table 5. Filter Types* | | |
| --- | --- | --- |
| **Filter Type** | **Example Expression** | **Description** |
| annotation (default) | org.example.SomeAnnotation | An annotation to be present at the type level in target components. |
| assignable | org.example.SomeClass | A class (or interface) that the target components are assignable to (extend/implement). |
| aspectj | org.example..\*Service+ | An AspectJ type expression to be matched by the target components. |
| regex | org\.example\.Default.\* | A regex expression to be matched by the target components class names. |
| custom | org.example.MyTypeFilter | A custom implementation of the org.springframework.core.type .TypeFilter interface. |

مثال زیر پیکربندی را نشان می‌دهد که تمام حاشیه‌نویسی‌های @Repository را نادیده می‌گیرد و به جای آن از مخازن "stub" استفاده می‌کند.

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

**public** **class** **AppConfig** {

...

}

و معادل آن با استفاده از XML

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

همچنین می‌توانید فیلترهای پیش‌فرض را با تنظیم useDefaultFilters=false در حاشیه‌نویسی یا ارائه use-default-filters="false" به عنوان ویژگی عنصر <component-scan/> غیرفعال کنید. این در واقع تشخیص خودکار کلاس های حاشیه نویسی شده با @Component، @Repository، @Service، @Controller، یا @Configuration را غیرفعال می کند.

1.10.5. Defining bean metadata within components

اجزای فنری همچنین می‌توانند به ابرداده تعریف لوبیا در ظرف کمک کنند. شما این کار را با همان حاشیه‌نویسی @Bean که برای تعریف فراداده bean در کلاس‌های حاشیه‌نویسی @Configuration استفاده می‌شود، انجام می‌دهید. در اینجا یک مثال ساده است:

@Component

**public** **class** **FactoryMethodComponent** {

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

**public** **void** doWork() {

*// Component method implementation omitted*

}

}

این کلاس یک جزء Spring است که دارای کد مخصوص برنامه کاربردی در متد ()doWork آن است. با این حال، یک تعریف bean نیز ارائه می‌کند که دارای یک متد کارخانه‌ای است که به متد publicInstance() اشاره دارد. حاشیه‌نویسی @Bean روش کارخانه و سایر ویژگی‌های تعریف bean، مانند مقدار واجد شرایط را از طریق حاشیه‌نویسی @Qualifier شناسایی می‌کند. حاشیه‌نویسی‌های سطح روش دیگری که می‌توان مشخص کرد عبارتند از @Scope، @Lazy و حاشیه‌نویسی واجد شرایط سفارشی.

حاشیه نویسی @Lazy علاوه بر نقش اولیه آن برای مقداردهی اولیه مؤلفه، ممکن است در نقاط تزریق که با @Autowired یا @Inject مشخص شده اند نیز قرار گیرد. در این زمینه، منجر به تزریق یک پروکسی با وضوح تنبل می شود.

فیلدها و روش‌های سیم‌کشی خودکار همانطور که قبلاً بحث شد، با پشتیبانی اضافی برای سیم‌کشی خودکار روش‌های @Bean پشتیبانی می‌شوند:

@Component

**public** **class** **FactoryMethodComponent** {

**private** **static** **int** i;

@Bean

@Qualifier("public")

**public** TestBean publicInstance() {

**return** **new** TestBean("publicInstance");

}

*// use of a custom qualifier and autowiring of method parameters*

@Bean

**protected** TestBean protectedInstance(

@Qualifier("public") TestBean spouse,

@Value("#{privateInstance.age}") String country) {

TestBean tb = **new** TestBean("protectedInstance", 1);

tb.setSpouse(spouse);

tb.setCountry(country);

**return** tb;

}

@Bean

**private** TestBean privateInstance() {

**return** **new** TestBean("privateInstance", i++);

}

@Bean

@RequestScope

**public** TestBean requestScopedInstance() {

**return** **new** TestBean("requestScopedInstance", 3);

}

}

این مثال، کشور پارامتر متد String را به مقدار ویژگی age در یک bean دیگر به نام privateInstance متصل می‌کند. یک عنصر Spring Expression Language مقدار ویژگی را از طریق نماد #{ <expression> } تعریف می کند. برای حاشیه‌نویسی‌های @Value، یک بیانگر از پیش پیکربندی شده است تا در هنگام حل متن عبارت به دنبال نام‌های bean بگردد.

از Spring Framework 4.3، همچنین می‌توانید یک پارامتر روش کارخانه از نوع InjectionPoint (یا زیر کلاس خاص‌تر DependencyDescriptor) را برای دسترسی به نقطه تزریق درخواستی که باعث ایجاد Bean فعلی می‌شود، اعلام کنید. توجه داشته باشید که این فقط در مورد ایجاد واقعی نمونه‌های bean اعمال می‌شود، نه برای تزریق نمونه‌های موجود. در نتیجه، این ویژگی برای لوبیاهای محدوده نمونه اولیه بسیار منطقی است. برای سایر حوزه‌ها، روش کارخانه تنها نقطه تزریق را مشاهده می‌کند که باعث ایجاد یک نمونه لوبیا جدید در محدوده داده شده شده است: به عنوان مثال، وابستگی که باعث ایجاد یک لوبیای تنبل شد. در چنین سناریوهایی از فراداده نقطه تزریق ارائه شده با مراقبت معنایی استفاده کنید.

@Component

**public** **class** **FactoryMethodComponent** {

@Bean @Scope("prototype")

**public** TestBean prototypeInstance(InjectionPoint injectionPoint) {

**return** **new** TestBean("prototypeInstance for " + injectionPoint.getMember());

}

}

متدهای @Bean در یک جزء معمولی Spring متفاوت از همتایان خود در داخل کلاس Spring @Configuration پردازش می‌شوند. تفاوت این است که کلاس های @Component با CGLIB برای رهگیری فراخوانی متدها و فیلدها تقویت نمی شوند. پروکسی CGLIB وسیله ای است که به وسیله آن فراخوانی متدها یا فیلدهای درون متدهای @Bean در کلاس های @Configuration ارجاعات فراداده bean به اشیاء همکار را ایجاد می کند. چنین روش‌هایی با معناشناسی معمولی جاوا احضار نمی‌شوند، بلکه از طریق ظرف عبور می‌کنند تا مدیریت چرخه حیات معمول و پروکسی Spring beans را حتی زمانی که از طریق فراخوانی‌های برنامه‌ای به روش‌های @Bean به Bean دیگر ارجاع می‌دهند، ارائه کنند. در مقابل، فراخوانی یک متد یا فیلد در متد @Bean در یک کلاس ساده @Component دارای معنایی استاندارد جاوا است، بدون پردازش خاص CGLIB یا سایر محدودیت‌ها.

می‌توانید متدهای @Bean را ثابت اعلام کنید، و به آنها اجازه می‌دهید بدون ایجاد کلاس پیکربندی حاوی آنها به عنوان نمونه فراخوانی شوند. این امر در هنگام تعریف دانه های پس از پردازش، به عنوان مثال، منطقی است. از نوع BeanFactoryPostProcessor یا BeanPostProcessor، زیرا چنین دانه‌هایی در اوایل چرخه حیات کانتینر مقداردهی اولیه می‌شوند و باید از فعال کردن سایر بخش‌های پیکربندی در آن نقطه اجتناب کنند.

توجه داشته باشید که تماس‌های متدهای @Bean ثابت هرگز توسط کانتینر رهگیری نمی‌شود، حتی در کلاس‌های پیکربندی @ (به بالا مراجعه کنید). این به دلیل محدودیت‌های فنی است: طبقه‌بندی فرعی CGLIB فقط می‌تواند روش‌های غیراستاتیک را لغو کند. در نتیجه، فراخوانی مستقیم به روش @Bean دیگر معنایی استاندارد جاوا خواهد داشت و در نتیجه یک نمونه مستقل مستقیماً از روش کارخانه بازگردانده می‌شود.

قابل مشاهده بودن زبان جاوا متدهای @Bean تاثیر فوری بر تعریف bean حاصل در کانتینر Spring ندارد. می‌توانید آزادانه روش‌های کارخانه‌ای خود را همانطور که می‌دانید در کلاس‌های non-@Configuration و همچنین برای روش‌های استاتیک در هر جایی اعلام کنید. با این حال، روش‌های معمولی @Bean در کلاس‌های @Configuration باید قابل لغو باشند، یعنی نباید به‌عنوان خصوصی یا نهایی اعلام شوند.

روش‌های @Bean همچنین بر روی کلاس‌های پایه یک جزء یا کلاس پیکربندی مشخص، و همچنین در روش‌های پیش‌فرض جاوا 8 که در رابط‌های پیاده‌سازی شده توسط کامپوننت یا کلاس پیکربندی اعلام شده‌اند، کشف می‌شوند. این امکان انعطاف پذیری زیادی را در ایجاد تنظیمات پیکربندی پیچیده فراهم می کند، حتی ارث بری چندگانه از طریق روش های پیش فرض جاوا 8 از Spring 4.2 امکان پذیر است.

در نهایت، توجه داشته باشید که یک کلاس واحد ممکن است چندین متد @Bean را برای یک bean نگه دارد، به عنوان ترتیبی از چندین روش کارخانه برای استفاده بسته به وابستگی های موجود در زمان اجرا. این همان الگوریتمی است که برای انتخاب «طمع‌آمیزترین» سازنده یا روش کارخانه در سناریوهای پیکربندی دیگر است: گونه‌ای که بیشترین تعداد وابستگی‌های رضایت‌بخش را دارد در زمان ساخت انتخاب می‌شود، مشابه نحوه انتخاب کانتینر بین چندین سازنده @Autowired.

#### 1.10.6. Naming autodetected components

هنگامی که یک جزء به عنوان بخشی از فرآیند اسکن به طور خودکار شناسایی می شود، نام bean آن توسط استراتژی BeanNameGenerator که برای آن اسکنر شناخته شده است، تولید می شود. به‌طور پیش‌فرض، هر گونه حاشیه‌نویسی کلیشه‌ای Spring (Component@، @Repository، @Service، و @Controller) که حاوی یک مقدار نام باشد، در نتیجه آن نام را به تعریف bean مربوطه ارائه می‌کند.

اگر چنین حاشیه‌نویسی حاوی هیچ مقدار نامی نباشد یا برای هر جزء شناسایی‌شده دیگری (مانند مواردی که توسط فیلترهای سفارشی کشف شده‌اند)، تولیدکننده نام bean پیش‌فرض، نام کلاس بدون حروف بزرگ را برمی‌گرداند. به عنوان مثال، اگر کلاس های مؤلفه زیر شناسایی شوند، نام ها myMovieLister و movieFinderImpl خواهد بود:

هنگامی که یک جزء به عنوان بخشی از فرآیند اسکن به طور خودکار شناسایی می شود، نام bean آن توسط استراتژی BeanNameGenerator که برای آن اسکنر شناخته شده است، تولید می شود. به‌طور پیش‌فرض، هر گونه حاشیه‌نویسی کلیشه‌ای Spring (Component@، @Repository، @Service، و @Controller) که حاوی یک مقدار نام باشد، در نتیجه آن نام را به تعریف bean مربوطه ارائه می‌کند.

اگر چنین حاشیه‌نویسی حاوی هیچ مقدار نامی نباشد یا برای هر جزء شناسایی‌شده دیگری (مانند مواردی که توسط فیلترهای سفارشی کشف شده‌اند)، تولیدکننده نام bean پیش‌فرض، نام کلاس بدون حروف بزرگ را برمی‌گرداند. به عنوان مثال، اگر کلاس های مؤلفه زیر شناسایی شوند، نام ها myMovieLister و movieFinderImpl خواهد بود:

@Service("myMovieLister")

**public** **class** **SimpleMovieLister** {

*// ...*

}

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

اگر نمی‌خواهید به استراتژی پیش‌فرض نام‌گذاری لوبیا تکیه کنید، می‌توانید یک استراتژی نام‌گذاری لوبیا سفارشی ارائه دهید. ابتدا رابط BeanNameGenerator را پیاده سازی کنید و مطمئن شوید که یک سازنده پیش فرض بدون آرگ قرار دهید. سپس، هنگام پیکربندی اسکنر، نام کلاس کاملاً واجد شرایط را ارائه دهید:

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

به عنوان یک قاعده کلی، هر زمان که سایر اجزا ممکن است اشاره صریحی به آن کنند، نام را با حاشیه‌نویسی مشخص کنید. از سوی دیگر، نام های تولید شده به صورت خودکار هر زمان که کانتینر مسئول سیم کشی باشد، کافی است.

#### 1.10.7. Providing a scope for autodetected components

همانند اجزای مدیریت شده با فنر به طور کلی، پیش‌فرض و رایج‌ترین حوزه برای مؤلفه‌های شناسایی خودکار تک‌تنه است. با این حال، گاهی اوقات شما به یک محدوده متفاوت نیاز دارید که می تواند از طریق حاشیه نویسی @Scope مشخص شود. به سادگی نام دامنه را در حاشیه نویسی وارد کنید:

@Scope("prototype")

@Repository

**public** **class** **MovieFinderImpl** **implements** MovieFinder {

*// ...*

}

حاشیه نویسی @Scope فقط در کلاس bean بتن (برای اجزای مشروح شده) یا روش کارخانه (برای روش های @Bean) درون نگری می شود. برخلاف تعاریف XML bean، مفهومی از وراثت تعریف bean وجود ندارد و سلسله مراتب وراثت در سطح کلاس برای اهداف ابرداده نامربوط است.

برای جزئیات بیشتر در مورد دامنه های خاص وب مانند "درخواست"/"جلسه" در زمینه Spring، به درخواست، جلسه، برنامه کاربردی و محدوده WebSocket مراجعه کنید. مانند حاشیه‌نویسی‌های از پیش ساخته شده برای آن حوزه‌ها، می‌توانید حاشیه‌نویسی‌های محدوده خود را نیز با استفاده از رویکرد فرا حاشیه‌نویسی Spring ایجاد کنید: به عنوان مثال. یک حاشیه نویسی سفارشی که با @Scope ("نمونه اولیه") مشروح شده است، که احتمالاً حالت پروکسی با محدوده سفارشی را نیز اعلام می کند.

برای ارائه یک استراتژی سفارشی برای وضوح دامنه به جای تکیه بر رویکرد مبتنی بر حاشیه‌نویسی، رابط ScopeMetadataResolver را پیاده‌سازی کنید و مطمئن شوید که یک سازنده پیش‌فرض بدون آرگ را در نظر بگیرید. سپس، هنگام پیکربندی اسکنر، نام کلاس کاملاً واجد شرایط را ارائه دهید:

@Configuration

@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example" scope-resolver="org.example.MyScopeResolver"/>

</beans>

هنگام استفاده از محدوده‌های غیر تکی خاص، ممکن است لازم باشد که پروکسی‌هایی برای اشیاء محدوده‌دار تولید شود. این استدلال در Scoped beans به عنوان وابستگی توصیف شده است. برای این منظور، یک ویژگی scoped-proxy در عنصر کامپوننت اسکن موجود است. سه مقدار ممکن عبارتند از: no، interfaces و targetClass. به عنوان مثال، پیکربندی زیر منجر به پروکسی های پویا JDK استاندارد می شود:

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

**public** **class** **AppConfig** {

...

}

<beans>

<context:component-scan base-package="org.example" scoped-proxy="interfaces"/>

</beans>

#### 1.10.8. Providing qualifier metadata with annotations

حاشیه نویسی @Qualifier در سیم کشی خودکار مبتنی بر حاشیه نویسی تنظیم دقیق با واجد شرایط مورد بحث قرار می گیرد. مثال‌های موجود در آن بخش، استفاده از حاشیه‌نویسی @Qualifier و حاشیه‌نویسی واجد شرایط سفارشی را برای ارائه کنترل دقیق هنگام حل و فصل نامزدهای autowire نشان می‌دهند. از آنجایی که آن مثال‌ها بر اساس تعاریف XML Bean بودند، فراداده واجد شرایط بر روی تعاریف bean کاندید با استفاده از واجد شرایط یا فراعناصر فرعی عنصر bean در XML ارائه شد. هنگام تکیه بر اسکن مسیر کلاس برای شناسایی خودکار مؤلفه‌ها، فراداده واجد شرایط را با حاشیه‌نویسی در سطح نوع در کلاس کاندید ارائه می‌کنید. سه مثال زیر این تکنیک را نشان می دهد:

@Component

**@Qualifier("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Genre("Action")**

**public** **class** **ActionMovieCatalog** **implements** MovieCatalog {

*// ...*

}

@Component

**@Offline**

**public** **class** **CachingMovieCatalog** **implements** MovieCatalog {

*// ...*

}

مانند بسیاری از گزینه های مبتنی بر حاشیه نویسی، به خاطر داشته باشید که ابرداده حاشیه نویسی به خود تعریف کلاس محدود می شود، در حالی که استفاده از XML به چندین دانه از همان نوع اجازه می دهد تا تغییراتی را در فراداده واجد شرایط خود ارائه دهد، زیرا این ابرداده به ازای هر یک ارائه می شود. نمونه به جای هر کلاس.

#### 1.10.9. Generating an index of candidate components

در حالی که اسکن مسیر کلاس بسیار سریع است، می‌توان عملکرد راه‌اندازی برنامه‌های بزرگ را با ایجاد یک لیست ثابت از نامزدها در زمان تدوین بهبود بخشید. در این حالت، همه ماژول های برنامه باید از این مکانیسم استفاده کنند، زیرا زمانی که ApplicationContext چنین شاخصی را شناسایی می کند، به جای اسکن مسیر کلاس، به طور خودکار از آن استفاده می کند.

برای تولید ایندکس، به سادگی یک وابستگی اضافی به هر ماژول اضافه کنید که حاوی مؤلفه هایی است که برای دستورالعمل های اسکن مؤلفه هدف هستند:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.0.20.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

یا با استفاده از Gradle:

dependencies {

compileOnly("org.springframework:spring-context-indexer:5.0.20.RELEASE")

}

این فرآیند یک فایل META-INF/spring.components ایجاد می کند که قرار است در jar گنجانده شود.

هنگام کار با این حالت در IDE خود، فهرست‌کننده بهار متنی باید به‌عنوان یک پردازشگر حاشیه‌نویسی ثبت شود تا هنگام به‌روزرسانی مؤلفه‌های کاندید، ایندکس به‌روز باشد.

هنگامی که یک META-INF/spring.components در مسیر کلاس یافت می شود، ایندکس به طور خودکار فعال می شود. اگر فهرستی برای برخی از کتابخانه‌ها (یا موارد استفاده) تا حدی در دسترس است، اما نمی‌توان آن را برای کل برنامه ایجاد کرد، می‌توانید با تنظیم spring.index.ignore به یک ترتیب کلاسی معمولی (یعنی اصلاً فهرستی وجود نداشت) بازگردید. به true، یا به عنوان یک ویژگی سیستم یا در یک فایل spring.properties در ریشه classpath.

### 1.11. Using JSR 330 Standard Annotations

با شروع Spring 3.0، Spring از حاشیه نویسی های استاندارد JSR-330 (تزریق وابستگی) پشتیبانی می کند. آن حاشیه نویسی ها به همان روشی که حاشیه نویسی های Spring اسکن می شوند. فقط باید jarهای مربوطه را در کلاس خود داشته باشید.

#### 1.11.1. Dependency Injection with @Inject and @Named

به جای @Autowired، @javax.inject.Inject ممکن است به صورت زیر استفاده شود:

**import** javax.inject.Inject;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.findMovies(...);

...

}

}

همانند @Autowired، امکان استفاده از @Inject در سطح فیلد، سطح متد و سطح سازنده-آرگومنت وجود دارد. علاوه بر این، می‌توانید نقطه تزریق خود را به‌عنوان یک ارائه‌دهنده اعلام کنید، که امکان دسترسی بر اساس تقاضا به دانه‌هایی با دامنه‌های کوتاه‌تر یا دسترسی تنبل به سایر دانه‌ها را از طریق فراخوانی Provider.get() فراهم می‌کند. به عنوان گونه ای از مثال بالا:

**import** javax.inject.Inject;

**import** javax.inject.Provider;

**public** **class** **SimpleMovieLister** {

**private** Provider<MovieFinder> movieFinder;

@Inject

**public** **void** setMovieFinder(Provider<MovieFinder> movieFinder) {

this.movieFinder = movieFinder;

}

**public** **void** listMovies() {

this.movieFinder.get().findMovies(...);

...

}

}

اگر می خواهید از یک نام واجد شرایط برای وابستگی که باید تزریق شود استفاده کنید، باید از حاشیه نویسی @Named به صورت زیر استفاده کنید:

**import** javax.inject.Inject;

**import** javax.inject.Named;

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

مانند @Autowired، @Inject نیز می تواند با java.util.Optional یا @Nullable استفاده شود. این در اینجا حتی بیشتر قابل استفاده است زیرا @Inject ویژگی require را ندارد.

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

**public** **class** **SimpleMovieLister** {

@Inject

**public** **void** setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

#### 1.11.2. @Named and @ManagedBean: standard equivalents to the @Component annotation

به جای @Component، @javax.inject.Named یا javax.annotation.ManagedBean ممکن است به صورت زیر استفاده شود:

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named("movieListener") *// @ManagedBean("movieListener") could be used as well*

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

استفاده از @Component بدون تعیین نام برای کامپوننت بسیار رایج است. @Named را می توان به روشی مشابه استفاده کرد:

**import** javax.inject.Inject;

**import** javax.inject.Named;

@Named

**public** **class** **SimpleMovieLister** {

**private** MovieFinder movieFinder;

@Inject

**public** **void** setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

*// ...*

}

هنگام استفاده از @Named یا @ManagedBean، می توان از اسکن کامپوننت دقیقاً به همان روشی که هنگام استفاده از حاشیه نویسی Spring استفاده می شود، استفاده کرد:

@Configuration

@ComponentScan(basePackages = "org.example")

**public** **class** **AppConfig** {

...

}

برخلاف @Component، حاشیه‌نویسی‌های JSR-330 @Named و JSR-250 ManagedBean قابل ترکیب نیستند. لطفاً از مدل کلیشه ای Spring برای ایجاد حاشیه نویسی اجزای سفارشی استفاده کنید.

#### 1.11.3. Limitations of JSR-330 standard annotations

هنگام کار با حاشیه نویسی استاندارد، مهم است که بدانید برخی از ویژگی های مهم مانند جدول زیر در دسترس نیستند:

| *Table 6. Spring component model elements vs. JSR-330 variants* | | |
| --- | --- | --- |
| **Spring** | **javax.inject.\*** | **javax.inject restrictions / comments** |
| @Autowired | @Inject | @Inject has no 'required' attribute; can be used with Java 8’s Optional instead. |
| @Component | @Named / @ManagedBean | JSR-330 does not provide a composable model, just a way to identify named components. |
| @Scope("singleton") | @Singleton | The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a [@Scope](https://download.oracle.com/javaee/6/api/javax/inject/Scope.html) annotation. Nevertheless, this one is only intended to be used for creating your own annotations. |
| @Qualifier | @Qualifier / @Named | javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named. |
| @Value | - | no equivalent |
| @Required | - | no equivalent |
| @Lazy | - | no equivalent |
| ObjectFactory | Provider | javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring’s @Autowired or with non-annotated constructors and setter methods. |

### 1.12. Java-based container configuration

#### 1.12.1. Basic concepts: @Bean and @Configuration

مصنوعات مرکزی در پشتیبانی از پیکربندی جدید جاوا Spring's، کلاس‌های @Configuration-annotated و @Bean-annotated متدها هستند.

حاشیه نویسی @Bean برای نشان دادن این که یک روش یک شی جدید را برای مدیریت کردن توسط کانتینر Spring IoC نمونه سازی، پیکربندی و مقداردهی اولیه می کند استفاده می شود. برای کسانی که با پیکربندی <beans/> XML Spring آشنایی دارند، حاشیه نویسی @Bean همان نقش عنصر <bean/> را ایفا می کند. می‌توانید از روش‌های حاشیه‌نویسی @Bean با هر component@Spring استفاده کنید، با این حال، این روش‌ها اغلب در @Configuration bean استفاده می‌شوند.

حاشیه نویسی یک کلاس با @Configuration نشان می دهد که هدف اصلی آن منبعی از تعاریف bean است. علاوه بر این، کلاس‌های configuration @ به سادگی با فراخوانی روش‌های @Bean در همان کلاس، می‌توانند وابستگی‌های بین‌بین را تعریف کنند. ساده ترین کلاس @Configuration ممکن است به صورت زیر خوانده شود:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** MyService myService() {

**return** **new** MyServiceImpl();

}

}

کلاس AppConfig بالا معادل Spring <beans/> XML زیر خواهد بود:

<beans>

<bean id="myService" class="com.acme.services.MyServiceImpl"/>

</beans>

Full @Configuration vs 'lite' @Bean mode?

هنگامی که متدهای @Bean در کلاس‌هایی که با @Configuration حاشیه‌نویسی نشده‌اند، اعلام می‌شوند، به‌عنوان در حال پردازش در حالت "lite" نامیده می‌شوند. متدهای Bean اعلام شده در یک @Component یا حتی در یک کلاس قدیمی ساده، 'lite' در نظر گرفته می‌شوند، با هدف اصلی متفاوتی از کلاس حاوی و یک روش @Bean فقط نوعی امتیاز در آنجا است. برای مثال، مؤلفه‌های سرویس ممکن است نماهای مدیریتی را از طریق یک روش @Bean اضافی در هر کلاس مؤلفه قابل اجرا در معرض کانتینر قرار دهند. در چنین سناریوهایی، روش‌های @Bean یک مکانیسم روش کارخانه همه منظوره ساده هستند.

برخلاف کامل @Configuration، روش‌های lite @Bean نمی‌توانند وابستگی‌های inter-bean را اعلام کنند. در عوض، آنها بر روی حالت داخلی مؤلفه حاوی خود و به صورت اختیاری بر روی آرگومان هایی که ممکن است اعلام کنند عمل می کنند. بنابراین چنین روش @Bean نباید سایر روش های @Bean را فراخوانی کند. هر یک از این روش ها به معنای واقعی کلمه فقط یک روش کارخانه ای برای یک مرجع bean خاص است، بدون هیچ گونه معنایی زمان اجرا خاصی. اثر جانبی مثبت در اینجا این است که در زمان اجرا نیازی به استفاده از کلاس فرعی CGLIB نیست، بنابراین هیچ محدودیتی از نظر طراحی کلاس وجود ندارد (یعنی کلاس حاوی ممکن است با این حال نهایی باشد و غیره).

در سناریوهای رایج، روش‌های @Bean باید در کلاس‌های configuration @ اعلان شوند، تا اطمینان حاصل شود که حالت «کامل» همیشه استفاده می‌شود و بنابراین مراجع متقاطع به مدیریت چرخه حیات کانتینر هدایت می‌شوند. این از فراخوانی تصادفی همان روش @Bean از طریق یک تماس معمولی جاوا جلوگیری می‌کند که به کاهش باگ‌های ظریفی کمک می‌کند که ردیابی آنها هنگام کار در حالت "لایت" دشوار است.

حاشیه‌نویسی‌های @Bean و @Configuration به طور عمیق در بخش‌های زیر مورد بحث قرار خواهند گرفت. با این حال، ابتدا روش های مختلف ایجاد یک محفظه فنری با استفاده از پیکربندی مبتنی بر جاوا را پوشش خواهیم داد.

#### 1.12.2. Instantiating the Spring container using AnnotationConfigApplicationContext

AnnotationConfigApplicationContext، در Spring 3.0 جدید است . این پیاده‌سازی همه کاره ApplicationContext نه تنها می‌تواند کلاس‌های @Configuration را به عنوان ورودی بپذیرد، بلکه می‌تواند کلاس‌های @Component ساده و کلاس‌های حاشیه‌نویسی شده با فراداده JSR-330 را نیز بپذیرد.

وقتی کلاس های @Configuration به عنوان ورودی ارائه می شوند، خود کلاس @Configuration به عنوان یک تعریف bean ثبت می شود و تمام متدهای @Bean اعلام شده در کلاس نیز به عنوان تعریف bean ثبت می شوند.

وقتی کلاس‌های @Component و JSR-330 ارائه می‌شوند، به‌عنوان تعاریف bean ثبت می‌شوند، و فرض بر این است که ابرداده‌های DI مانند @Autowired یا @Inject در آن کلاس‌ها در صورت لزوم استفاده می‌شوند.

##### Simple construction

تقریباً به همان روشی که فایل‌های Spring XML به‌عنوان ورودی هنگام نمونه‌سازی ClassPathXmlApplicationContext استفاده می‌شوند، کلاس‌های Configuration@ ممکن است به عنوان ورودی هنگام نمونه‌سازی AnnotationConfigApplicationContext استفاده شوند. این امکان استفاده کاملاً بدون XML از کانتینر Spring را فراهم می کند:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

همانطور که در بالا ذکر شد، AnnotationConfigApplicationContext فقط به کار با کلاس های @Configuration محدود نمی شود. هر کلاس مشروح شده @Component یا JSR-330 ممکن است به عنوان ورودی به سازنده ارائه شود. مثلا:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

در بالا فرض می شود که MyServiceImpl، Dependency1 و Dependency2 از حاشیه نویسی تزریق وابستگی Spring مانند @Autowired استفاده می کنند.

##### Building the container programmatically using register(Class<?>…​)

یک AnnotationConfigApplicationContext ممکن است با استفاده از یک سازنده بدون arg نمونه سازی شود و سپس با استفاده از متد register() پیکربندی شود. این رویکرد به ویژه هنگام ساختن برنامه‌نویسی AnnotationConfigApplicationContext مفید است.

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

##### Enabling component scanning with scan(String…​)

برای فعال کردن اسکن کامپوننت، کافی است کلاس @Configuration خود را به صورت زیر حاشیه نویسی کنید:

@Configuration

@ComponentScan(basePackages = "com.acme")

**public** **class** **AppConfig** {

...

}

کاربران باتجربه Spring با معادل اعلان XML از متن Spring آشنا هستند: فضای نام

<beans>

<context:component-scan base-package="com.acme"/>

</beans>

در مثال بالا، بسته com.acme اسکن می‌شود و به دنبال کلاس‌های @Component-annotated می‌گردد، و آن کلاس‌ها به عنوان تعاریف Spring bean در ظرف ثبت می‌شوند. AnnotationConfigApplicationContext روش اسکن (String…​) را نشان می‌دهد تا عملکرد اسکن مؤلفه یکسانی را انجام دهد:

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

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

به یاد داشته باشید که کلاس های @Configuration با @Component دارای حاشیه نویسی می شوند، بنابراین آنها کاندیدای اسکن کامپوننت هستند! در مثال بالا، با فرض اینکه AppConfig در بسته com.acme (یا هر بسته زیر آن) اعلان شده باشد، در طول فراخوانی به scan() برداشته می شود و پس از refresh() تمام متدهای @Bean آن پردازش می شود و به عنوان تعاریف لوبیا در ظرف ثبت شده است.

##### Support for web applications with AnnotationConfigWebApplicationContext

یک نوع WebApplicationContext از AnnotationConfigApplicationContext با AnnotationConfigWebApplicationContext موجود است. این پیاده سازی ممکن است هنگام پیکربندی شنونده سرویس Spring ContextLoaderListener، Spring MVC DispatcherServlet، و غیره استفاده شود. به استفاده از contextClass context-param و init-param توجه کنید:

<web-app>

*<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

*<!-- Configuration locations must consist of one or more comma- or space-delimited*

*fully-qualified @Configuration classes. Fully-qualified packages may also be*

*specified for component-scanning -->*

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

*<!-- Bootstrap the root application context as usual using ContextLoaderListener -->*

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

*<!-- Declare a Spring MVC DispatcherServlet as usual -->*

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

*<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext*

*instead of the default XmlWebApplicationContext -->*

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

*<!-- Again, config locations must consist of one or more comma- or space-delimited*

*and fully-qualified @Configuration classes -->*

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.web.MvcConfig</param-value>

</init-param>

</servlet>

*<!-- map all requests for /app/\* to the dispatcher servlet -->*

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

#### 1.12.3. Using the @Bean annotation

@Bean یک حاشیه نویسی در سطح روش و آنالوگ مستقیم عنصر XML <bean/> است. حاشیه نویسی برخی از ویژگی های ارائه شده توسط <bean/> را پشتیبانی می کند، مانند: init-method، death-method، autowiring و name.

می‌توانید از حاشیه‌نویسی @Bean در یک کلاس @Configuration-annotated یا در یک کلاس @Component-annotated استفاده کنید.

##### Declaring a bean

برای اعلام یک bean، به سادگی یک روش را با حاشیه نویسی @Bean حاشیه نویسی کنید. شما از این روش برای ثبت تعریف bean در یک ApplicationContext از نوع مشخص شده به عنوان مقدار برگشتی متد استفاده می کنید. به طور پیش فرض، نام bean با نام متد یکسان خواهد بود. مثال زیر یک مثال ساده از اعلان متد @Bean است:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferServiceImpl transferService() {

**return** **new** TransferServiceImpl();

}

}

پیکربندی قبلی دقیقاً معادل Spring XML زیر است:

<beans>

<bean id="transferService" class="com.acme.TransferServiceImpl"/>

</beans>

هر دو اعلان یک Bean به نام transferService را در ApplicationContext در دسترس قرار می دهند که به یک نمونه شی از نوع TransferServiceImpl متصل می شود:

transferService -> com.acme.TransferServiceImpl

همچنین می توانید متد @Bean خود را با نوع بازگشت رابط (یا کلاس پایه) اعلام کنید:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl();

}

}

با این حال، این قابلیت دید برای پیش‌بینی نوع پیشرفته را به نوع رابط مشخص شده (TransferService) محدود می‌کند و نوع کامل (TransferServiceImpl) را تنها زمانی که لوبیا singletonآسیب‌دیده نمونه‌سازی شد، برای ظرف شناخته می‌شود. لوبیاهای سینگلتون غیر تنبل بر اساس ترتیب اعلامی خود نمونه سازی می شوند، بنابراین بسته به زمانی که جزء دیگری سعی می کند با یک نوع غیراعلام شده مطابقت داشته باشد، ممکن است نتایج تطبیق نوع متفاوتی را مشاهده کنید (مانند @Autowired TransferServiceImpl که فقط یک بار "transferService" حل می شود. bean نمونه شده است).

##### Bean dependencies

یک روش حاشیه‌نویسی @Bean می‌تواند دارای تعدادی پارامتر دلخواه باشد که وابستگی‌های مورد نیاز برای ساخت آن bean را توصیف می‌کند. به عنوان مثال، اگر TransferService ما به یک AccountRepository نیاز داشته باشد، می‌توانیم این وابستگی را از طریق پارامتر متد محقق کنیم:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

##### Receiving lifecycle callbacks

هر کلاسی که با حاشیه‌نویسی @Bean تعریف می‌شود، از تماس‌های معمولی چرخه حیات پشتیبانی می‌کند و می‌تواند از حاشیه‌نویسی‌های @PostConstruct و @PreDestroy از JSR-250 استفاده کند، برای جزئیات بیشتر به حاشیه‌نویسی‌های JSR-250 مراجعه کنید.

تماس‌های عادی چرخه عمر Spring نیز به طور کامل پشتیبانی می‌شوند. اگر یک bean InitializingBean، DisposableBean یا Lifecycle را پیاده سازی کند، متدهای مربوطه آنها توسط ظرف فراخوانی می شود.

مجموعه استاندارد رابط های \*Aware مانند BeanFactoryAware، BeanNameAware، MessageSourceAware، ApplicationContextAware و غیره نیز به طور کامل پشتیبانی می شوند.

حاشیه‌نویسی @Bean از مشخص کردن روش‌های بازخوانی اولیه و تخریب دلخواه پشتیبانی می‌کند، دقیقاً مانند ویژگی‌های init-method و death-method Spring XML در عنصر bean:

**public** **class** **Foo** {

**public** **void** init() {

*// initialization logic*

}

}

**public** **class** **Bar** {

**public** **void** cleanup() {

*// destruction logic*

}

}

@Configuration

**public** **class** **AppConfig** {

@Bean(initMethod = "init")

**public** Foo foo() {

**return** **new** Foo();

}

@Bean(destroyMethod = "cleanup")

**public** Bar bar() {

**return** **new** Bar();

}

}

به‌طور پیش‌فرض، دانه‌هایی که با استفاده از پیکربندی جاوا تعریف می‌شوند و دارای روش بسته یا خاموش کردن عمومی هستند، به‌طور خودکار با یک تماس تخریب ثبت می‌شوند. اگر روش بسته یا خاموش کردن عمومی دارید و نمی‌خواهید هنگام خاموش شدن کانتینر فراخوانی شود، کافی است @Bean(destroyMethod="") را به تعریف bean خود اضافه کنید تا حالت پیش‌فرض (استنتاج شده) غیرفعال شود.

ممکن است بخواهید این کار را به طور پیش فرض برای منبعی انجام دهید که از طریق JNDI به دست می آورید زیرا چرخه حیات آن خارج از برنامه مدیریت می شود. به ویژه، مطمئن شوید که همیشه این کار را برای یک DataSource انجام دهید، زیرا در سرورهای برنامه Java EE مشکل ساز است.

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** NamingException {

**return** (DataSource) jndiTemplate.lookup("MyDS");

}

همچنین، با روش‌های @Bean، شما معمولاً استفاده از جستجوهای برنامه‌ای JNDI را انتخاب می‌کنید: یا با استفاده از کمک‌های Spring's JndiTemplate/JndiLocatorDelegate یا استفاده مستقیم از JNDI InitialContext، اما نه نوع JndiObjectFactoryBean که شما را مجبور می‌کند نوع بازگشتی را به‌جای نوع FactoryBean اعلام کنید. نوع هدف واقعی، استفاده از تماس های ارجاع متقابل را در سایر روش های @Bean که قصد مراجعه به منبع ارائه شده در اینجا را دارند دشوارتر می کند.

البته، در مورد Foo بالا، فراخوانی مستقیم متد init() در حین ساخت به همان اندازه معتبر خواهد بود:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

Foo foo = **new** Foo();

foo.init();

**return** foo;

}

*// ...*

}

هنگامی که مستقیماً در جاوا کار می کنید، می توانید هر کاری را که دوست دارید با اشیاء خود انجام دهید و همیشه نیازی به تکیه بر چرخه حیات کانتینر ندارید!

##### Specifying bean scope

###### Using the @Scope annotation

می‌توانید مشخص کنید که دانه‌های شما که با حاشیه‌نویسی @Bean تعریف می‌شوند، باید محدوده خاصی داشته باشند. می توانید از هر یک از محدوده های استاندارد مشخص شده در قسمت Bean Scopes استفاده کنید.

دامنه پیش‌فرض singleton است، اما می‌توانید با حاشیه‌نویسی @Scope آن را لغو کنید:

@Configuration

**public** **class** **MyConfiguration** {

@Bean

**@Scope("prototype")**

**public** Encryptor encryptor() {

*// ...*

}

}

###### @Scope and scoped-proxy

Spring راه مناسبی برای کار با وابستگی های محدوده از طریق پراکسی های scoped ارائه می دهد. ساده ترین راه برای ایجاد چنین پروکسی در هنگام استفاده از پیکربندی XML عنصر <aop:scoped-proxy/> است. پیکربندی beans خود در جاوا با حاشیه نویسی @Scope پشتیبانی معادل با ویژگی proxyMode را ارائه می دهد. پیش‌فرض هیچ پروکسی نیست (ScopedProxyMode.NO)، اما می‌توانید ScopedProxyMode.TARGET\_CLASS یا ScopedProxyMode.INTERFACES را مشخص کنید.

اگر نمونه پروکسی scoped را از مستندات مرجع XML (به پیوند قبلی مراجعه کنید) به @Bean ما با استفاده از جاوا پورت کنید، به شکل زیر خواهد بود:

*// an HTTP Session-scoped bean exposed as a proxy*

@Bean

**@SessionScope**

**public** UserPreferences userPreferences() {

**return** **new** UserPreferences();

}

@Bean

**public** Service userService() {

UserService service = **new** SimpleUserService();

*// a reference to the proxied userPreferences bean*

service.setUserPreferences(userPreferences());

**return** service;

}

##### Customizing bean naming

به‌طور پیش‌فرض، کلاس‌های پیکربندی از نام متد @Bean به عنوان نام bean استفاده می‌کنند. با این حال، این عملکرد را می توان با ویژگی name لغو کرد.

@Configuration

**public** **class** **AppConfig** {

@Bean(name = "myFoo")

**public** Foo foo() {

**return** **new** Foo();

}

}

##### Bean aliasing

همانطور که در Naming beans بحث شد، گاهی اوقات مطلوب است که به یک لوبیا چند نام داده شود، که در غیر این صورت به عنوان bean aliasing شناخته می شود. ویژگی نام حاشیه نویسی @Bean یک آرایه String را برای این منظور می پذیرد.

@Configuration

**public** **class** **AppConfig** {

@Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})

**public** DataSource dataSource() {

*// instantiate, configure and return DataSource bean...*

}

}

##### Bean description

گاهی اوقات ارائه توضیحات متنی دقیق تر از لوبیا مفید است. این می تواند به ویژه زمانی مفید باشد که لوبیاها (شاید از طریق JMX) برای اهداف نظارتی در معرض قرار می گیرند.

برای افزودن توضیحات به @Bean می توان از حاشیه نویسی @Description استفاده کرد:

@Configuration

**public** **class** **AppConfig** {

@Bean

**@Description("Provides a basic example of a bean")**

**public** Foo foo() {

**return** **new** Foo();

}

}

#### 1.12.4. Using the @Configuration annotation

@Configuration یک حاشیه نویسی در سطح کلاس است که نشان می دهد یک شی منبع تعاریف bean است. کلاس‌های پیکربندی @Beans را از طریق روش‌های مشروح عمومی @Bean اعلام می‌کنند. فراخوانی به روش‌های @Bean در کلاس‌های configuration @ نیز می‌تواند برای تعریف وابستگی‌های بین استفاده شود. برای معرفی کلی به مفاهیم اساسی: @Bean و @Configuration مراجعه کنید.

##### Injecting inter-bean dependencies

وقتی @Beans به یکدیگر وابستگی دارند، بیان این وابستگی به سادگی فراخوانی یک متد bean است:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** Foo foo() {

**return** **new** Foo(bar());

}

@Bean

**public** Bar bar() {

**return** **new** Bar();

}

}

در مثال بالا، foo bean یک مرجع به نوار از طریق تزریق سازنده دریافت می کند.

این روش برای اعلان وابستگی های inter-bean فقط زمانی کار می کند که متد @Bean در یک کلاس configuration@ اعلان شود. شما نمی توانید وابستگی های inter-bean را با استفاده از کلاس های @Component ساده اعلام کنید.

##### Lookup method injection

همانطور که قبلا ذکر شد، [lookup method injection](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html" \l "beans-factory-method-injection)یک ویژگی پیشرفته است که به ندرت باید از آن استفاده کنید. در مواردی مفید است که یک لوبیا با singleton-scop به یک لوبیا با prototype-scop وابستگی دارد. استفاده از جاوا برای این نوع پیکربندی ابزاری طبیعی برای پیاده سازی این الگو فراهم می کند.

**public** **abstract** **class** **CommandManager** {

**public** Object process(Object commandState) {

*// grab a new instance of the appropriate Command interface*

Command command = createCommand();

*// set the state on the (hopefully brand new) Command instance*

command.setState(commandState);

**return** command.execute();

}

*// okay... but where is the implementation of this method?*

**protected** **abstract** Command createCommand();

}

با استفاده از پشتیبانی از java configuration، می‌توانید یک زیر کلاس از CommandManager ایجاد کنید که در آن متد انتزاعی createCommand () به گونه‌ای که یک شی دستور جدید (نمونه اولیه) را جستجو می‌کند، بازنویسی می‌شود:

@Bean

@Scope("prototype")

**public** AsyncCommand asyncCommand() {

AsyncCommand command = **new** AsyncCommand();

*// inject dependencies here as required*

**return** command;

}

@Bean

**public** CommandManager commandManager() {

*// return new anonymous implementation of CommandManager with command() overridden*

*// to return a new prototype Command object*

**return** **new** CommandManager() {

**protected** Command createCommand() {

**return** asyncCommand();

}

}

}

##### Further information about how Java-based configuration works internally

مثال زیر یک متد مشروح @Bean را نشان می دهد که دو بار فراخوانی شده است:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** ClientService clientService1() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientService clientService2() {

ClientServiceImpl clientService = **new** ClientServiceImpl();

clientService.setClientDao(clientDao());

**return** clientService;

}

@Bean

**public** ClientDao clientDao() {

**return** **new** ClientDaoImpl();

}

}

clientDao() یک بار در clientService1() و یک بار در clientService2() فراخوانی شده است. از آنجایی که این روش یک نمونه جدید از ClientDaoImpl ایجاد می کند و آن را برمی گرداند، معمولاً انتظار دارید 2 نمونه (یک نمونه برای هر سرویس) داشته باشید. این قطعاً مشکل‌ساز خواهد بود: در بهار، لوبیاهای نمونه به طور پیش‌فرض دارای یک محدوده تک تنی هستند. اینجاست که جادو وارد می‌شود: تمام کلاس‌های configuration@ در زمان راه‌اندازی با CGLIB زیر کلاس‌بندی می‌شوند. در کلاس فرعی، متد فرزند ابتدا محفظه را برای هر گونه دانه های ذخیره شده (scoped) قبل از فراخوانی متد والد بررسی می کند و یک نمونه جدید ایجاد می کند. توجه داشته باشید که از Spring 3.2 دیگر نیازی به اضافه کردن CGLIB به مسیر کلاس خود نیست زیرا کلاس های CGLIB تحت org.springframework.cglib مجدداً بسته بندی شده اند و مستقیماً در JAR هسته فنری گنجانده شده اند.

رفتار ممکن است با توجه به محدوده لوبیا شما متفاوت باشد. ما در اینجا در مورد singleton صحبت می کنیم.

به دلیل این واقعیت که CGLIB به صورت پویا ویژگی‌ها را در زمان راه‌اندازی اضافه می‌کند، محدودیت‌هایی وجود دارد، به ویژه اینکه کلاس‌های پیکربندی نباید نهایی باشند. با این حال، از 4.3، هر سازنده ای در کلاس های پیکربندی مجاز است، از جمله استفاده از @Autowired یا یک اعلان سازنده غیر پیش فرض برای تزریق پیش فرض.

اگر ترجیح می‌دهید از هرگونه محدودیت تحمیلی CGLIB اجتناب کنید، روش‌های @Bean خود را در کلاس‌های non-@Configuration اعلام کنید، به عنوان مثال. در عوض در کلاس‌های ساده @Component. تماس‌های متقاطع بین روش‌های @Bean در آن زمان رهگیری نمی‌شوند، بنابراین باید منحصراً به تزریق وابستگی در سطح سازنده یا متد آنجا تکیه کنید.

#### 1.12.5. Composing Java-based configurations

##### Using the @Import annotation

همان‌طور که عنصر <import/> در فایل‌های Spring XML برای کمک به ماژولار کردن پیکربندی‌ها استفاده می‌شود، حاشیه‌نویسی @Import اجازه بارگیری تعاریف @Bean را از کلاس پیکربندی دیگر می‌دهد:

@Configuration

**public** **class** **ConfigA** {

@Bean

**public** A a() {

**return** **new** A();

}

}

@Configuration

@Import(ConfigA.class)

**public** **class** **ConfigB** {

@Bean

**public** B b() {

**return** **new** B();

}

}

اکنون، به جای نیاز به تعیین ConfigA.class و ConfigB.class هنگام نمونه سازی متن، فقط ConfigB باید به طور صریح ارائه شود:

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(ConfigB.class);

*// now both beans A and B will be available...*

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

این رویکرد نمونه سازی کانتینر را ساده می کند، زیرا فقط باید با یک کلاس برخورد شود، به جای اینکه توسعه دهنده نیاز به یادآوری تعداد بالقوه زیادی از کلاس های @Configuration در طول ساخت داشته باشد.

###### Injecting dependencies on imported @Bean definitions

مثال بالا کار می کند، اما ساده است. در اکثر سناریوهای عملی، beans در کلاس‌های پیکربندی به یکدیگر وابستگی دارند. هنگام استفاده از XML، این موضوع به خودی خود مشکلی نیست، زیرا هیچ کامپایلری در کار نیست، و می توان به سادگی ref="someBean" را اعلام کرد و اعتماد کرد که Spring آن را در طول اولیه سازی کانتینر حل خواهد کرد. البته، هنگام استفاده از کلاس‌هایconfiguration@، کامپایلر جاوا محدودیت‌هایی را بر روی مدل پیکربندی اعمال می‌کند، به این صورت که ارجاع‌ها به سایر دانه‌ها باید نحو جاوا معتبر باشند.

خوشبختانه حل این مشکل ساده است. همانطور که قبلاً بحث کردیم، روش @Bean می‌تواند دارای تعداد دلخواه پارامترهایی باشد که وابستگی‌های bean را توصیف می‌کنند. بیایید یک سناریوی واقعی تری با چندین کلاس @Configuration در نظر بگیریم که هر کدام به دانه های اعلام شده در سایرین بستگی دارد:

@Configuration

**public** **class** **ServiceConfig** {

@Bean

**public** TransferService transferService(AccountRepository accountRepository) {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

@Bean

**public** AccountRepository accountRepository(DataSource dataSource) {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

راه دیگری برای رسیدن به نتیجه مشابه وجود دارد. به یاد داشته باشید که کلاس‌های پیکربندی @ در نهایت فقط یک bean دیگر در ظرف هستند: این بدان معنی است که آنها می‌توانند از مزایای @Autowired و @Value injection و غیره مانند هر bean دیگری استفاده کنند!

اطمینان حاصل کنید که وابستگی هایی که به این روش تزریق می کنید فقط از ساده ترین نوع هستند. @Configuration کلاس‌ها در طول اولیه‌سازی زمینه بسیار زود پردازش می‌شوند و وادار کردن وابستگی به این روش ممکن است منجر به مقداردهی اولیه غیرمنتظره شود. در صورت امکان، مانند مثال بالا به تزریق مبتنی بر پارامتر متوسل شوید.

همچنین، به ویژه در مورد تعاریف BeanPostProcessor و BeanFactoryPostProcessor از طریق @Bean دقت کنید. آنها معمولاً باید به‌عنوان متدهای @Bean ثابت اعلام شوند، نه اینکه نمونه‌سازی کلاس پیکربندی حاوی آنها را راه‌اندازی کنند. در غیر این صورت، @Autowired و @Value روی خود کلاس پیکربندی کار نخواهند کرد زیرا خیلی زود به عنوان یک نمونه bean ایجاد شده است.

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** AccountRepository accountRepository;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(accountRepository);

}

}

@Configuration

**public** **class** **RepositoryConfig** {

**private** **final** DataSource dataSource;

@Autowired

**public** RepositoryConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return new DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

*// everything wires up across configuration classes...*

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

تزریق سازنده در کلاس های @Configuration فقط در Spring Framework 4.3 پشتیبانی می شود. همچنین توجه داشته باشید که اگر bean هدف فقط یک سازنده را تعریف کند، نیازی به تعیین @Autowired نیست. در مثال بالا، @Autowired در سازنده RepositoryConfig ضروری نیست.

Fully-qualifying imported beans for ease of navigation

در سناریوی بالا، استفاده از @Autowired به خوبی کار می کند و ماژولاریت مورد نظر را فراهم می کند، اما تعیین اینکه دقیقاً کجا تعاریف autowired bean اعلام شده است هنوز تا حدودی مبهم است. به عنوان مثال، به عنوان یک توسعه‌دهنده که به ServiceConfig نگاه می‌کند، چگونه می‌دانید که Bean @Autowired AccountRepository دقیقا کجا اعلام شده است؟ در کد مشخص نیست، و ممکن است خوب باشد. به یاد داشته باشید که Spring Tool Suite ابزاری را ارائه می دهد که می تواند نمودارهایی را نشان دهد که چگونه همه چیز سیم کشی شده است - این ممکن است تمام چیزی باشد که شما نیاز دارید. همچنین، Java IDE شما به راحتی می‌تواند تمام اعلان‌ها و موارد استفاده از نوع AccountRepository را پیدا کند و به سرعت مکان متدهای @Bean را که آن نوع را برمی‌گردانند، به شما نشان می‌دهد.

در مواردی که این ابهام قابل قبول نیست و می‌خواهید از داخل IDE خود از یک کلاس @Configuration به کلاس دیگر پیمایش مستقیم داشته باشید، خود کلاس‌های پیکربندی را به صورت خودکار سیم‌کشی کنید:

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

*// navigate 'through' the config class to the @Bean method!*

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

در وضعیت بالا، کاملاً مشخص است که AccountRepository کجا تعریف شده است. با این حال، ServiceConfig اکنون به شدت با RepositoryConfig همراه شده است. این معامله است این جفت سخت را می توان با استفاده از کلاس های @Configuration مبتنی بر واسط یا مبتنی بر کلاس انتزاعی تا حدودی کاهش داد. موارد زیر را در نظر بگیرید:

@Configuration

**public** **class** **ServiceConfig** {

@Autowired

**private** RepositoryConfig repositoryConfig;

@Bean

**public** TransferService transferService() {

**return** **new** TransferServiceImpl(repositoryConfig.accountRepository());

}

}

@Configuration

**public** **interface** **RepositoryConfig** {

@Bean

AccountRepository accountRepository();

}

@Configuration

**public** **class** **DefaultRepositoryConfig** **implements** RepositoryConfig {

@Bean

**public** AccountRepository accountRepository() {

**return** **new** JdbcAccountRepository(...);

}

}

@Configuration

@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) *// import the concrete config!*

**public** **class** **SystemTestConfig** {

@Bean

**public** DataSource dataSource() {

*// return DataSource*

}

}

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(SystemTestConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

اکنون ServiceConfig با توجه به DefaultRepositoryConfig بتن به‌طور ضعیف جفت می‌شود، و ابزار IDE داخلی هنوز مفید است: برای توسعه‌دهنده آسان است که سلسله مراتبی از پیاده‌سازی‌های RepositoryConfig را دریافت کند. به این ترتیب، پیمایش در کلاس‌های Configuration@ و وابستگی‌های آن‌ها تفاوتی با روند معمول پیمایش کدهای مبتنی بر رابط ندارد.

اگر می‌خواهید بر ترتیب ایجاد استارت‌آپ برخی از دانه‌ها تأثیر بگذارید، برخی از آن‌ها را به‌عنوان @Lazy (برای ایجاد در اولین دسترسی به جای راه‌اندازی) یا به‌عنوان @DependsOn در برخی از bean‌های دیگر (مطمئن شوید که دانه‌های خاص دیگر مشخص می‌شوند) در نظر بگیرید. ایجاد شده قبل از لوبیا فعلی، فراتر از آنچه که وابستگی مستقیم دومی دلالت می کند).

##### Conditionally include @Configuration classes or @Bean methods

فعال یا غیرفعال کردن یک کلاس کامل @Configuration یا حتی روش‌های فردی @Bean، بر اساس برخی از حالت‌های سیستم دلخواه، اغلب مفید است. یکی از مثال‌های رایج این مورد استفاده از حاشیه‌نویسی @Profile برای فعال کردن bean تنها زمانی است که یک نمایه خاص در محیط Spring Environment فعال شده باشد (برای جزئیات به پروفایل‌های تعریف Bean مراجعه کنید).

حاشیه نویسی @Profile در واقع با استفاده از حاشیه نویسی بسیار انعطاف پذیرتر به نام @Conditional پیاده سازی می شود. حاشیه‌نویسی Conditional @ نشان‌دهنده پیاده‌سازی‌های خاص org.springframework.context.annotation.Condition است که باید قبل از ثبت @Bean با آنها مشورت شود.

پیاده‌سازی رابط Condition به سادگی یک متد matches (…) ارائه می‌کند که true یا false را برمی‌گرداند. به عنوان مثال، در اینجا پیاده سازی Condition واقعی مورد استفاده برای @Profile است:

@Override

**public** **boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

**if** (context.getEnvironment() != null) {

*// Read the @Profile annotation attributes*

MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());

**if** (attrs != null) {

**for** (Object value : attrs.get("value")) {

**if** (context.getEnvironment().acceptsProfiles(((String**[]**) value))) {

**return** true;

}

}

**return** false;

}

}

**return** true;

}

##### Combining Java and XML configuration

هدف از پشتیبانی کلاس @Configuration Spring's جایگزینی 100٪ کامل برای Spring XML نیست. برخی از امکانات مانند فضاهای نام Spring XML یک راه ایده آل برای پیکربندی کانتینر باقی می ماند. در مواردی که XML مناسب یا ضروری است، یک انتخاب دارید: یا با استفاده از ClassPathXmlApplicationContext، محفظه را به روشی «XML محور» نمونه‌سازی کنید، یا به‌صورت «جاوا محور» با استفاده از AnnotationConfigApplicationContext و حاشیه‌نویسی @ImportResource. XML را در صورت نیاز وارد کنید.

###### XML-centric use of @Configuration classes

ممکن است ترجیح داده شود که کانتینر Spring از XML راه‌اندازی شود و کلاس‌های configuration @ به صورت ad-hoc گنجانده شود. به عنوان مثال، در یک پایگاه کد بزرگ موجود که از Spring XML استفاده می کند، ایجاد کلاس های @Configuration بر اساس نیاز و گنجاندن آنها از فایل های XML موجود آسان تر خواهد بود. در زیر گزینه‌های استفاده از کلاس‌های پیکربندی@ را در این نوع موقعیت «XML محور» خواهید یافت.

*Declaring @Configuration classes as plain Spring*<bean/>*elements*

به یاد داشته باشید که کلاس های @Configuration در نهایت فقط تعاریف bean در ظرف هستند. در این مثال، یک کلاس @Configuration به نام AppConfig ایجاد می کنیم و آن را در system-test-config.xml به عنوان تعریف <bean/> قرار می دهیم. از آنجایی که <context:annotation-config/> روشن است، کانتینر حاشیه نویسی @Configuration را می شناسد و روش های @Bean اعلام شده در AppConfig را به درستی پردازش می کند.

<beans>

*<!-- enable processing of annotations such as @Autowired and @Configuration -->*

<context:annotation-config/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

**jdbc.properties**:

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

در system-test-config.xml بالا، AppConfig <bean/> عنصر id را اعلام نمی کند. در حالی که انجام این کار قابل قبول است، اما با توجه به اینکه هیچ لوبیا دیگری هرگز به آن اشاره نمی کند، غیر ضروری است، و بعید است که به صراحت با نام از ظرف برداشته شود. به همین ترتیب با DataSource bean - فقط بر اساس نوع سیم‌کشی خودکار انجام می‌شود، بنابراین یک شناسه bean صریح به شدت مورد نیاز نیست.

*Using <context:component-scan/> to pick up*@Configuration*classes*

از آنجایی که @Configuration با @Component متا حاشیه نویسی می شود، کلاس های @Configuration-annotated به طور خودکار کاندیدای اسکن کامپوننت هستند. با استفاده از همان سناریوی بالا، می‌توانیم system-test-config.xml را برای استفاده از قابلیت اسکن مؤلفه‌ها دوباره تعریف کنیم. توجه داشته باشید که در این مورد، ما نیازی به بیان صریح <context:annotation-config/> نداریم، زیرا <context:component-scan/> همین عملکرد را فعال می‌کند.

**system-test-config.xml**:

<beans>

*<!-- picks up and registers AppConfig as a bean definition -->*

<context:component-scan base-package="com.acme"/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

در برنامه هایی که کلاس های @Configuration مکانیسم اصلی برای پیکربندی کانتینر هستند، احتمالاً همچنان استفاده از حداقل مقداری XML ضروری است. در این سناریوها، به سادگی از @ImportResource استفاده کنید و فقط به اندازه نیاز XML تعریف کنید. انجام این کار به یک رویکرد "جاوا محور" برای پیکربندی کانتینر دست می یابد و XML را به حداقل می رساند.

@Configuration

@ImportResource("classpath:/com/acme/properties-config.xml")

**public** **class** **AppConfig** {

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Bean

**public** DataSource dataSource() {

**return** **new** DriverManagerDataSource(url, username, password);

}

}

properties-config.xml

<beans>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

</beans>

jdbc.properties

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

*// ...*

}

### 1.13. Environment abstraction

Environment یک انتزاع ادغام شده در ظرف است که دو جنبه کلیدی محیط برنامه را مدل می کند: پروفایل ها و ویژگی ها.

نمایه یک گروه منطقی و نامگذاری شده از تعاریف bean است که فقط در صورتی که نمایه داده شده فعال باشد، در ظرف ثبت می شود. Beans ممکن است به یک نمایه اختصاص داده شود، چه در XML تعریف شده باشد و چه از طریق حاشیه نویسی. نقش شیء Environment در رابطه با پروفایل ها در تعیین این است که کدام پروفایل ها (در صورت وجود) در حال حاضر فعال هستند و کدام پروفایل ها (در صورت وجود) باید به طور پیش فرض فعال باشند.

ویژگی‌ها تقریباً در همه برنامه‌ها نقش مهمی دارند و ممکن است از منابع مختلفی نشأت بگیرند: فایل‌های ویژگی‌ها، ویژگی‌های سیستم JVM، متغیرهای محیط سیستم، JNDI، پارامترهای زمینه سرولت، اشیاء ویژگی‌های ad-hoc، نقشه‌ها و غیره. نقش شیء Environment در رابطه با خصوصیات این است که یک رابط سرویس مناسب برای پیکربندی منابع دارایی و حل خواص از آنها به کاربر ارائه دهد.

#### 1.13.1. Bean definition profiles

پروفایل های تعریف لوبیا مکانیزمی در محفظه هسته است که امکان ثبت دانه های مختلف در محیط های مختلف را فراهم می کند. کلمه محیط برای کاربران مختلف می تواند معانی مختلفی داشته باشد و این ویژگی می تواند در موارد استفاده بسیاری از جمله:

• کار در برابر یک منبع داده درون حافظه در توسعه در مقابل جستجوی همان منبع داده از JNDI در هنگام QA یا تولید

• ثبت زیرساخت نظارت فقط هنگام استقرار یک برنامه کاربردی در یک محیط عملکرد

• ثبت پیاده سازی های سفارشی سازی شده لوبیا برای استقرار مشتری A در مقابل مشتری B

بیایید اولین مورد استفاده را در یک برنامه کاربردی که به یک DataSource نیاز دارد، در نظر بگیریم. در یک محیط آزمایشی، پیکربندی ممکن است به صورت زیر باشد:

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("my-schema.sql")

.addScript("my-test-data.sql")

.build();

}

حال بیایید در نظر بگیریم که چگونه این برنامه در یک محیط QA یا تولید مستقر می شود، با این فرض که منبع داده برای برنامه در فهرست JNDI سرور برنامه تولید ثبت می شود. اکنون dataSource bean ما به شکل زیر است:

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

مشکل این است که چگونه بین استفاده از این دو تغییر بر اساس محیط فعلی جابجا شوید. با گذشت زمان، کاربران Spring راه‌های زیادی برای انجام این کار ابداع کرده‌اند، که معمولاً به ترکیبی از متغیرهای محیط سیستم و دستورات XML <import/> حاوی توکن‌های ${placeholder} تکیه می‌کنند که بسته به مقدار به مسیر فایل پیکربندی صحیح می‌رسند. از یک متغیر محیطی پروفایل های تعریف لوبیا یک ویژگی کانتینر اصلی است که راه حلی برای این مشکل ارائه می دهد.

اگر مثال استفاده از تعاریف خاص محیطی را تعمیم دهیم، در نهایت به نیاز به ثبت تعاریف خاص bean در زمینه‌های خاص می‌رسیم، در حالی که در موارد دیگر نه. می‌توانید بگویید که می‌خواهید یک نمایه خاص از تعاریف bean را در وضعیت A ثبت کنید، و یک نمایه متفاوت در موقعیت B. ابتدا ببینیم چگونه می‌توانیم پیکربندی خود را برای منعکس کردن این نیاز به‌روزرسانی کنیم.

##### @Profile

حاشیه نویسی Profile@ به شما امکان می دهد زمانی که یک یا چند نمایه مشخص شده فعال هستند، نشان دهید که یک مؤلفه برای ثبت نام واجد شرایط است. با استفاده از مثال بالا، می‌توانیم پیکربندی dataSource را به صورت زیر بازنویسی کنیم:

@Configuration

**@Profile("development")**

**public** **class** **StandaloneDataConfig** {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

}

@Configuration

**@Profile("production")**

**public** **class** **JndiDataConfig** {

@Bean(destroyMethod="")

**public** DataSource dataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

همانطور که قبلاً ذکر شد، با روش‌های @Bean، شما معمولاً استفاده از جستجوهای برنامه‌ای JNDI را انتخاب می‌کنید: یا با استفاده از کمک‌های Spring's JndiTemplate/JndiLocatorDelegate یا استفاده مستقیم از JNDI InitialContext نشان داده شده در بالا، اما نه نوع JndiObjectFactoryBean که شما را مجبور به اعلام نوع برگشتی به عنوان نوع FactoryBean

@Profile را می توان به عنوان یک meta-annotationبرای ایجاد یک حاشیه نویسی سفارشی استفاده کرد. مثال زیر یک حاشیه نویسی سفارشی @Production را تعریف می کند که می تواند به عنوان جایگزینی برای @Profile ("production") استفاده شود:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

**@Profile("production")**

**public** @interface Production {

}

**ا**گر یک کلاس Configuration@ با Profile@ علامت‌گذاری شود، تمام روش‌های @Bean و حاشیه‌نویسی‌های @Import مرتبط با آن کلاس دور زده می‌شوند مگر اینکه یک یا چند نمایه مشخص‌شده فعال باشند. اگر یک کلاس @Component یا @Configuration با @Profile({"p1"، "p2"}) علامت گذاری شده باشد، آن کلاس ثبت/پردازش نمی شود مگر اینکه پروفایل های 'p1' و/یا 'p2' فعال شده باشند. اگر پیشوند یک نمایه با عملگر NOT (!) باشد، اگر نمایه فعال نباشد، عنصر حاشیه نویسی ثبت می شود. به عنوان مثال، با توجه به @Profile({"p1"، "!p2"})، اگر نمایه 'p1' فعال باشد یا اگر نمایه 'p2' فعال نباشد، ثبت نام انجام می شود.

@Profile همچنین می تواند در سطح متد اعلان شود تا فقط یک bean خاص از یک کلاس پیکربندی را شامل شود، به عنوان مثال. برای انواع جایگزین یک لوبیا خاص:

@Configuration

**public** **class** **AppConfig** {

@Bean("dataSource")

**@Profile("development")**

**public** DataSource standaloneDataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

@Bean("dataSource")

**@Profile("production")**

**public** DataSource jndiDataSource() **throws** Exception {

Context ctx = **new** InitialContext();

**return** (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

با @Profile در روش‌های @Bean، ممکن است یک سناریوی خاص اعمال شود: در مورد روش‌های @Bean با نام روش جاوا یکسان (مشابه با سربارگذاری سازنده)، یک شرط Profile@ باید به طور مداوم در همه روش‌های سربارگذاری شده اعلام شود. اگر شرایط ناسازگار باشند، فقط شرط اولین اعلام در بین روش‌های اضافه بار اهمیت دارد. بنابراین نمی توان از @Profile برای انتخاب یک روش بارگذاری شده با امضای آرگومان خاص روی دیگری استفاده کرد. وضوح بین تمام روش‌های کارخانه برای یک bean از الگوریتم وضوح سازنده Spring در زمان ایجاد پیروی می‌کند.

اگر می‌خواهید دانه‌های جایگزین با شرایط پروفایل متفاوت تعریف کنید، از نام‌های متد جاوا متمایز استفاده کنید که از طریق ویژگی @Bean name به همان نام bean اشاره می‌کنند، همانطور که در مثال بالا نشان داده شده است. اگر امضاهای آرگومان همگی یکسان باشند (مثلاً همه انواع متدهای کارخانه ای بدون آرگ دارند)، این تنها راه برای نمایش چنین ترتیبی در یک کلاس جاوا معتبر در وهله اول است (زیرا فقط یک روش می تواند وجود داشته باشد. یک نام خاص و امضای استدلال

##### XML bean definition profiles

همتای XML ویژگی نمایه عنصر <beans> است. پیکربندی نمونه ما در بالا را می توان در دو فایل XML به شرح زیر بازنویسی کرد:

<beans profile="development"

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

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

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

xsi:schemaLocation="...">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production"

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

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

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

xsi:schemaLocation="...">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

همچنین می توان از تقسیم و قرار دادن عناصر <beans/> در یک فایل جلوگیری کرد:

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

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

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

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

xsi:schemaLocation="...">

*<!-- other bean definitions -->*

<beans profile="development">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

</beans>

Spring-bean.xsd محدود شده است تا چنین عناصری را فقط به عنوان آخرین آنها در فایل مجاز کند. این باید به ارائه انعطاف پذیری بدون درهم ریختگی در فایل های XML کمک کند.

##### Activating a profile

اکنون که پیکربندی خود را به روز کرده ایم، هنوز باید به Spring دستور دهیم که کدام نمایه فعال است. اگر برنامه نمونه خود را همین الان شروع کنیم، یک NoSuchBeanDefinitionException پرتاب می شود، زیرا ظرف نمی تواند Spring bean با نام dataSource را پیدا کند.

فعال‌سازی یک نمایه را می‌توان به روش‌های مختلفی انجام داد، اما ساده‌ترین آن این است که آن را به صورت برنامه‌نویسی در برابر Environment API که از طریق یک ApplicationContext در دسترس است، انجام دهید:

AnnotationConfigApplicationContext ctx = **new** AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);

ctx.refresh();

علاوه بر این، نمایه‌ها همچنین می‌توانند به صورت اعلامی از طریق ویژگی spring.profiles.active فعال شوند که ممکن است از طریق متغیرهای محیط سیستم، ویژگی‌های سیستم JVM، پارامترهای زمینه سرولت در web.xml، یا حتی به‌عنوان ورودی در JNDI (نگاه کنید به انتزاع PropertySource) مشخص شود. . در تست های یکپارچه سازی، پروفایل های فعال را می توان از طریق حاشیه نویسی ActiveProfiles@ در ماژول تست فنری اعلام کرد (به پیکربندی زمینه با پروفایل های محیط مراجعه کنید).

توجه داشته باشید که پروفایل ها یک گزاره "یا یا" نیستند. این امکان وجود دارد که چندین پروفایل را همزمان فعال کنید. از نظر برنامه‌ریزی، به سادگی چندین نام پروفایل را برای متد ()setActiveProfiles ارائه دهید که String… varargs را می‌پذیرد:

ctx.getEnvironment().setActiveProfiles("profile1", "profile2");

به طور اعلامی، spring.profiles.active ممکن است فهرستی از نام های نمایه را که با کاما جدا شده اند بپذیرد:

-Dspring.profiles.active="profile1,profile2"

##### Default profile

نمایه پیش فرض نمایه ای را نشان می دهد که به طور پیش فرض فعال است. موارد زیر را در نظر بگیرید:

@Configuration

**@Profile("default")**

**public** **class** **DefaultDataConfig** {

@Bean

**public** DataSource dataSource() {

**return** **new** EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.build();

}

}

اگر هیچ پروفایلی فعال نباشد، dataSource بالا ایجاد می شود. این می تواند به عنوان راهی برای ارائه یک تعریف پیش فرض برای یک یا چند لوبیا دیده شود. اگر هر نمایه ای فعال باشد، نمایه پیش فرض اعمال نخواهد شد.

نام نمایه پیش‌فرض را می‌توان با استفاده از setDefaultProfiles() در Environment یا با استفاده از ویژگی spring.profiles.default تغییر داد.

#### 1.13.2. PropertySource abstraction

ApplicationContext ctx = **new** GenericApplicationContext();

Environment env = ctx.getEnvironment();

**boolean** containsFoo = env.containsProperty("foo");

System.out.println("Does my environment contain the 'foo' property? " + containsFoo);

در قطعه بالا، یک روش سطح بالا برای پرسیدن از Spring می بینیم که آیا ویژگی foo برای محیط فعلی تعریف شده است یا خیر. برای پاسخ به این سوال، شیء Environment جستجویی را روی مجموعه ای از اشیاء PropertySource انجام می دهد. PropertySource یک انتزاع ساده بر روی هر منبعی از جفت‌های کلید-مقدار است، و Spring's StandardEnvironment با دو شیء PropertySource پیکربندی شده است - یکی نشان دهنده مجموعه ویژگی‌های سیستم JVM (a la System.getProperties()) و یکی نشان دهنده مجموعه محیط سیستم است. متغیرها (a la System.getenv()).

این منابع ویژگی پیش‌فرض برای StandardEnvironment برای استفاده در برنامه‌های مستقل وجود دارد. StandardServletEnvironment با منابع ویژگی های پیش فرض اضافی از جمله پیکربندی servlet و پارامترهای زمینه servlet پر شده است. می تواند به صورت اختیاری یک JndiPropertySource را فعال کند. برای جزئیات بیشتر به javadocs مراجعه کنید

جستجوی انجام شده سلسله مراتبی است. به‌طور پیش‌فرض، ویژگی‌های سیستم نسبت به متغیرهای محیط اولویت دارند، بنابراین اگر ویژگی foo در هر دو مکان در طول تماس با env.getProperty ("foo") تنظیم شود، مقدار ویژگی سیستم "برنده" می‌شود و ترجیحاً برگردانده می‌شود. متغیر محیطی. توجه داشته باشید که مقادیر ویژگی با هم ادغام نمی شوند، بلکه به طور کامل توسط یک ورودی قبلی لغو می شوند.

برای یک StandardServletEnvironment مشترک، سلسله مراتب کامل به صورت زیر است، با بالاترین اولویت در بالا:

* ServletConfig parameters (if applicable, e.g. in case of a DispatcherServlet context)
* ServletContext parameters (web.xml context-param entries)
* JNDI environment variables ("java:comp/env/" entries)
* JVM system properties ("-D" command-line arguments)

JVM system environment (operating system environment variables)

مهمتر از همه، کل مکانیسم قابل تنظیم است. شاید شما یک منبع سفارشی از خواص داشته باشید که بخواهید آن را در این جستجو ادغام کنید. مشکلی نیست—به سادگی PropertySource خود را پیاده سازی و نمونه سازی کنید و آن را به مجموعه PropertySources برای محیط فعلی اضافه کنید:

ConfigurableApplicationContext ctx = **new** GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(**new** MyPropertySource());

#### 1.13.3. @PropertySource

در کد بالا، MyPropertySource با بالاترین اولویت در جستجو اضافه شده است. اگر دارای یک ویژگی foo باشد، قبل از هر ویژگی foo در هر PropertySource دیگری شناسایی و برگردانده می شود. MutablePropertySources API تعدادی روش را نشان می دهد که امکان دستکاری دقیق مجموعه منابع ویژگی را فراهم می کند.

@Configuration

**@PropertySource("classpath:/com/myco/app.properties")**

**public** **class** **AppConfig** {

@Autowired

Environment env;

@Bean

**public** TestBean testBean() {

TestBean testBean = **new** TestBean();

testBean.setName(env.getProperty("testbean.name"));

**return** testBean;

}

}

هر متغیر ${…} موجود در یک مکان منبع @PropertySource با مجموعه منابع دارایی که قبلاً در برابر محیط ثبت شده است حل می شود. مثلا:

با فرض اینکه "my.placeholder" در یکی از منابع دارایی قبلاً ثبت شده وجود دارد، به عنوان مثال. ویژگی های سیستم یا متغیرهای محیطی، مکان نگهدار به مقدار مربوطه حل می شود. اگر نه، «پیش‌فرض/مسیر» به‌عنوان پیش‌فرض استفاده می‌شود. اگر هیچ پیش‌فرضی مشخص نشده باشد و یک ویژگی قابل حل نباشد، یک IllegalArgumentException پرتاب می‌شود.

حاشیه نویسی @PropertySource طبق قراردادهای جاوا 8 قابل تکرار است. با این حال، تمام این حاشیه‌نویسی‌های @PropertySource باید در یک سطح اعلام شوند: یا مستقیماً در کلاس پیکربندی یا به‌عنوان meta-annotationsدر همان حاشیه‌نویسی سفارشی. اختلاط حاشیه نویسی مستقیم و متا حاشیه نویسی توصیه نمی شود زیرا حاشیه نویسی مستقیم به طور موثری فرا حاشیه نویسی را لغو می کند.

#### 1.13.4. Placeholder resolution in statements

از لحاظ تاریخی، مقدار متغیرهای مکان در عناصر فقط در برابر ویژگی های سیستم JVM یا متغیرهای محیطی قابل حل است. دیگر این مورد نیست. از آنجایی که انتزاع محیطی در سراسر کانتینر یکپارچه شده است، مسیریابی تفکیک مکان‌ها از طریق آن آسان است. این بدان معناست که شما می‌توانید فرآیند تفکیک را به هر شکلی که دوست دارید پیکربندی کنید: اولویت جستجو از طریق ویژگی‌های سیستم و متغیرهای محیطی را تغییر دهید یا آنها را به طور کامل حذف کنید. منابع دارایی خود را در صورت لزوم به ترکیب اضافه کنید.

به طور مشخص، بیانیه زیر بدون توجه به اینکه دارایی مشتری در کجا تعریف شده است، تا زمانی که در محیط موجود باشد، کار می کند:

<beans>

<import resource="com/bank/service/${customer}-config.xml"/>

</beans>

### 1.14. Registering a LoadTimeWeaver

LoadTimeWeaver توسط Spring برای تبدیل دینامیک کلاسها در هنگام بارگذاری آنها در ماشین مجازی جاوا (JVM) استفاده می شود.

برای فعال کردن بافت در زمان بارگذاری، @EnableLoadTimeWeaving را به یکی از کلاس‌های configuration@ خود اضافه کنید:

@Configuration

@EnableLoadTimeWeaving

**public** **class** **AppConfig** {

}

روش دیگر برای پیکربندی XML از عنصر context:load-time-weaver استفاده کنید:

<beans>

<context:load-time-weaver/>

</beans>

پس از پیکربندی برای ApplicationContext. هر Bean در آن ApplicationContext ممکن است LoadTimeWeaverAware را پیاده سازی کند، در نتیجه مرجعی به نمونه بافنده بارگذاری دریافت می کند. این به ویژه در ترکیب با پشتیبانی از JPA Springs مفید است که در آن بافت در زمان بار ممکن است برای تبدیل کلاس JPA ضروری باشد. برای جزئیات بیشتر با LocalContainerEntityManagerFactoryBean javadocs مشورت کنید. برای اطلاعات بیشتر در مورد بافندگی در زمان بارگذاری AspectJ، به بافت در زمان بارگذاری با AspectJ در چارچوب Spring مراجعه کنید.

### 1.15. Additional capabilities of the ApplicationContext

همانطور که در مقدمه فصل مورد بحث قرار گرفت، بسته org.springframework.beans.factory قابلیت های اساسی را برای مدیریت و دستکاری دانه ها، از جمله به روش برنامه ریزی شده، فراهم می کند. بسته org.springframework.context اینترفیس ApplicationContext را اضافه می کند که رابط BeanFactory را گسترش می دهد، علاوه بر این، رابط های دیگر را گسترش می دهد تا عملکردهای اضافی را در سبکی مبتنی بر چارچوب برنامه کاربردی تر ارائه دهد. بسیاری از مردم از ApplicationContext به صورت کاملاً اعلامی استفاده می‌کنند، حتی آن را به صورت برنامه‌نویسی ایجاد نمی‌کنند، اما در عوض به کلاس‌های پشتیبانی مانند ContextLoader برای نمونه‌سازی خودکار یک ApplicationContext به عنوان بخشی از فرآیند عادی راه‌اندازی یک برنامه وب Java EE تکیه می‌کنند.

برای تقویت عملکرد BeanFactory در سبکی چارچوب محورتر، بسته متنی عملکرد زیر را نیز ارائه می دهد:

• دسترسی به پیام ها به سبک i18n، از طریق رابط MessageSource.

• دسترسی به منابع، مانند URL ها و فایل ها، از طریق رابط ResourceLoader.

• انتشار رویداد به دانه هایی که رابط ApplicationListener را پیاده سازی می کنند، از طریق استفاده از رابط ApplicationEventPublisher.

• بارگذاری زمینه های متعدد (سلسله مراتبی)، که به هر یک از آنها اجازه می دهد از طریق رابط HierarchicalBeanFactory بر روی یک لایه خاص، مانند لایه وب یک برنامه، متمرکز شوند.

#### 1.15.1. Internationalization using MessageSource

رابط ApplicationContext رابطی به نام MessageSource را گسترش می دهد و بنابراین عملکرد بین المللی (i18n) را ارائه می دهد. Spring همچنین رابط HierarchicalMessageSource را ارائه می دهد که می تواند پیام ها را به صورت سلسله مراتبی حل کند. این رابط‌ها با هم پایه‌ای را فراهم می‌کنند که Spring بر وضوح پیام تأثیر می‌گذارد. روش های تعریف شده در این رابط ها عبارتند از:

• String getMessage (کد رشته، آرگ‌های Object[]، پیش‌فرض رشته، Locale loc): روش اصلی برای بازیابی پیام از MessageSource. هنگامی که هیچ پیامی برای منطقه مشخص شده یافت نشد، از پیام پیش فرض استفاده می شود. هر آرگومان ارسال شده با استفاده از قابلیت MessageFormat ارائه شده توسط کتابخانه استاندارد به مقادیر جایگزین تبدیل می شود.

• String getMessage(String code, Object[] args, String default, Locale loc)اساساً مانند روش قبلی است، اما با یک تفاوت: هیچ پیام پیش فرضی قابل تعیین نیست. اگر پیام پیدا نشد، یک NoSuchMessageException پرتاب می شود.

• String getMessage (MessageSourceResolvable resolvable, Locale Locale): تمام خصوصیات استفاده شده در متدهای قبلی نیز در کلاسی به نام MessageSourceResolvable قرار می گیرند که می توانید با این روش از آن استفاده کنید.

هنگامی که یک ApplicationContext بارگذاری می‌شود، به‌طور خودکار یک MessageSource bean تعریف شده در متن را جستجو می‌کند. لوبیا باید نام messageSource را داشته باشد. اگر چنین bean پیدا شود، همه فراخوانی‌های روش‌های قبلی به منبع پیام واگذار می‌شوند. اگر هیچ منبع پیامی پیدا نشد، ApplicationContext تلاش می‌کند تا والد حاوی یک لوبیا با همان نام را پیدا کند. اگر اینطور باشد، از آن bean به عنوان MessageSource استفاده می کند. اگر ApplicationContext نتواند هیچ منبعی را برای پیام‌ها پیدا کند، یک DelegatingMessageSource خالی ایجاد می‌شود تا بتواند تماس‌های روش‌های تعریف‌شده در بالا را بپذیرد.

Spring دو پیاده سازی MessageSource، ResourceBundleMessageSource و StaticMessageSource را ارائه می دهد. هر دو HierarchicalMessageSource را برای انجام پیام‌های تودرتو پیاده‌سازی می‌کنند. StaticMessageSource به ندرت مورد استفاده قرار می گیرد اما راه های برنامه ریزی شده برای افزودن پیام ها به منبع را ارائه می دهد. ResourceBundleMessageSource در مثال زیر نشان داده شده است:

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

در مثال فرض شده است که شما سه بسته منبع دارید که در مسیر کلاس خود تعریف شده اند به نام فرمت، استثنا و ویندوز. هر درخواستی برای حل یک پیام به روش استاندارد JDK برای حل پیام ها از طریق ResourceBundles رسیدگی می شود. برای اهداف مثال، فرض کنید محتویات دو فایل از فایل‌های بسته منبع بالا…

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

برنامه ای برای اجرای عملکرد MessageSource در مثال بعدی نشان داده شده است. به یاد داشته باشید که تمام پیاده‌سازی‌های ApplicationContext نیز پیاده‌سازی MessageSource هستند و بنابراین می‌توانند به رابط MessageSource ارسال شوند.

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

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

خروجی حاصل از برنامه فوق به صورت…

Alligators rock!

بنابراین به طور خلاصه، MessageSource در فایلی به نام beans.xml تعریف شده است که در ریشه classpath شما وجود دارد. تعریف messageSource bean به تعدادی از بسته‌های منبع از طریق ویژگی نام پایه آن اشاره دارد. سه فایلی که در لیست به ویژگی basenames منتقل می‌شوند به‌عنوان فایل‌هایی در ریشه مسیر کلاس شما وجود دارند و به ترتیب format.properties،exceptions.properties و windows.properties نامیده می‌شوند.

مثال بعدی آرگومان های ارسال شده به جستجوی پیام را نشان می دهد. این آرگومان‌ها به رشته‌ها تبدیل می‌شوند و در مکان‌هایی در پیام جستجو درج می‌شوند.

<beans>

*<!-- this MessageSource is being used in a web application -->*

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

*<!-- lets inject the above MessageSource into this POJO -->*

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

**public** **class** **Example** {

**private** MessageSource messages;

**public** **void** setMessages(MessageSource messages) {

this.messages = messages;

}

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

String message = this.messages.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", null);

System.out.println(message);

}

}

خروجی حاصل از فراخوانی متد ()execute خواهد بود…

The userDao argument is required.

با توجه به بین‌المللی‌سازی (i18n)، پیاده‌سازی‌های مختلف MessageSource Spring از همان وضوح محلی و قوانین بازگشتی مانند استاندارد JDK ResourceBundle پیروی می‌کنند. به طور خلاصه، و با مثال messageSource که قبلاً تعریف شده است، ادامه دهید، اگر می‌خواهید پیام‌ها را بر خلاف زبان انگلیسی (en-GB) حل کنید، به ترتیب فایل‌هایی به نام‌های format\_en\_GB.properties،exceptions\_en\_GB.properties و windows\_en\_GB.properties ایجاد می‌کنید.

به طور معمول، وضوح محلی توسط محیط اطراف برنامه مدیریت می شود. در این مثال، محلی که پیام های (بریتانیایی) با آن حل می شوند به صورت دستی مشخص شده است.

# in exceptions\_en\_GB.properties

argument.required=Ebagum lad, the {0} argument is required, I say, required.

**public** **static** **void** main(**final** String**[]** args) {

MessageSource resources = **new** ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("argument.required",

**new** Object **[]** {"userDao"}, "Required", Locale.UK);

System.out.println(message);

}

خروجی حاصل از اجرای برنامه فوق به صورت…

Ebagum lad, the 'userDao' argument is required, I say, required.

همچنین می توانید از رابط MessageSourceAware برای به دست آوردن یک مرجع به هر MessageSource تعریف شده استفاده کنید. هر Bean که در یک ApplicationContext تعریف شده است که رابط MessageSourceAware را پیاده سازی می کند، با MessageSource زمینه برنامه زمانی که bean ایجاد و پیکربندی می شود، تزریق می شود.

به عنوان جایگزینی برای ResourceBundleMessageSource، Spring یک کلاس ReloadableResourceBundleMessageSource را ارائه می دهد. این نوع از فرمت فایل بسته یکسانی پشتیبانی می‌کند، اما نسبت به اجرای استاندارد ResourceBundleMessageSource مبتنی بر JDK انعطاف‌پذیرتر است. به طور خاص، امکان خواندن فایل‌ها را از هر مکان منبع Spring (نه فقط از مسیر کلاس) فراهم می‌کند و از بارگیری مجدد فایل‌های ویژگی بسته‌ای (در حالی که به طور موثر آنها را در حافظه پنهان می‌کند) پشتیبانی می‌کند. برای جزئیات، javadocs ReloadableResourceBundleMessageSource را بررسی کنید.

#### 1.15.2. Standard and custom events

مدیریت رویداد در ApplicationContext از طریق کلاس ApplicationEvent و رابط ApplicationListener ارائه می شود. اگر یک Bean که رابط ApplicationListener را پیاده‌سازی می‌کند در متن مستقر شود، هر بار که یک ApplicationEvent در ApplicationContext منتشر می‌شود، آن bean مطلع می‌شود. در اصل، این الگوی استاندارد طراحی Observer است.

از بهار 4.2، زیرساخت رویداد به طور قابل توجهی بهبود یافته است و یک مدل مبتنی بر حاشیه نویسی و همچنین توانایی انتشار هر رویداد دلخواه را ارائه می دهد، که یک شی است که لزوماً از ApplicationEvent گسترش نمی یابد. هنگامی که چنین شیئی منتشر می شود، آن را در یک رویداد برای شما قرار می دهیم.

بهار رویدادهای استاندارد زیر را ارائه می دهد:

| *Table 7. Built-in Events* | |
| --- | --- |
| **Event** | **Explanation** |
| ContextRefreshedEvent | Published when the ApplicationContext is initialized or refreshed, for example, using the refresh() method on the ConfigurableApplicationContext interface. "Initialized" here means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not. |
| ContextStartedEvent | Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContext interface. "Started" here means that all Lifecycle beans receive an explicit start signal. Typically this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart , for example, components that have not already started on initialization. |
| ContextStoppedEvent | Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContext interface. "Stopped" here means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call. |
| ContextClosedEvent | Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContext interface. "Closed" here means that all singleton beans are destroyed. A closed context reaches its end of life; it cannot be refreshed or restarted. |
| RequestHandledEvent | A web-specific event telling all beans that an HTTP request has been serviced. This event is published *after* the request is complete. This event is only applicable to web applications using Spring’s DispatcherServlet. |

همچنین می توانید رویدادهای سفارشی خود را ایجاد و منتشر کنید. این مثال یک کلاس ساده را نشان می دهد که کلاس پایه Spring's ApplicationEvent را گسترش می دهد:

**public** **class** **BlackListEvent** **extends** ApplicationEvent {

**private** **final** String address;

**private** **final** String content;

**public** BlackListEvent(Object source, String address, String content) {

super(source);

this.address = address;

this.content = content;

}

*// accessor and other methods...*

}

برای انتشار یک ApplicationEvent سفارشی، متد publicEvent() را در یک ApplicationEventPublisher فراخوانی کنید. معمولاً این کار با ایجاد کلاسی انجام می شود که ApplicationEventPublisherAware را پیاده سازی می کند و آن را به عنوان Spring bean ثبت می کند. مثال زیر چنین کلاسی را نشان می دهد:

**public** **class** **EmailService** **implements** ApplicationEventPublisherAware {

**private** List<String> blackList;

**private** ApplicationEventPublisher publisher;

**public** **void** setBlackList(List<String> blackList) {

this.blackList = blackList;

}

**public** **void** setApplicationEventPublisher(ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

**public** **void** sendEmail(String address, String content) {

**if** (blackList.contains(address)) {

publisher.publishEvent(**new** BlackListEvent(this, address, content));

**return**;

}

*// send email...*

}

}

در زمان پیکربندی، کانتینر Spring تشخیص می‌دهد که EmailService ApplicationEventPublisherAware را پیاده‌سازی می‌کند و به طور خودکار setApplicationEventPublisher را فراخوانی می‌کند. در واقع، پارامتر ارسال شده در خود ظرف Spring خواهد بود. شما به سادگی از طریق رابط ApplicationEventPublisher با زمینه برنامه تعامل دارید.

برای دریافت ApplicationEvent سفارشی، کلاسی ایجاد کنید که ApplicationListener را پیاده سازی کند و آن را به عنوان Spring bean ثبت کنید. مثال زیر چنین کلاسی را نشان می دهد:

**public** **class** **BlackListNotifier** **implements** ApplicationListener<BlackListEvent> {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

**public** **void** onApplicationEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

توجه داشته باشید که ApplicationListener به طور کلی با نوع رویداد سفارشی شما، BlackListEvent، پارامتر شده است. این بدان معنی است که متد onApplicationEvent() می تواند از نظر نوع ایمن باقی بماند و از هرگونه نیاز به downcast اجتناب کند. می‌توانید هر تعداد شنونده رویداد را که می‌خواهید ثبت کنید، اما توجه داشته باشید که به‌طور پیش‌فرض شنوندگان رویداد رویدادها را به صورت همزمان دریافت می‌کنند. این بدان معناست که متد publicEvent() تا زمانی که همه شنوندگان پردازش رویداد را به پایان نرسانند مسدود می‌شود. یکی از مزیت های این رویکرد همزمان و تک رشته ای این است که وقتی شنونده رویدادی را دریافت می کند، در صورت موجود بودن زمینه تراکنش، در داخل بافت تراکنش ناشر عمل می کند. اگر استراتژی دیگری برای انتشار رویداد ضروری است، به javadoc برای رابط Spring's ApplicationEventMulticaster مراجعه کنید.

مثال زیر تعاریف bean مورد استفاده برای ثبت و پیکربندی هر یک از کلاس های بالا را نشان می دهد:

<bean id="emailService" class="example.EmailService">

<property name="blackList">

<list>

<value>known.spammer@example.org</value>

<value>known.hacker@example.org</value>

<value>john.doe@example.org</value>

</list>

</property>

</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">

<property name="notificationAddress" value="blacklist@example.org"/>

</bean>

با کنار هم گذاشتن همه اینها، زمانی که متد sendEmail() از emailService bean فراخوانی می شود، اگر ایمیل هایی وجود داشته باشد که باید در لیست سیاه قرار گیرند، یک رویداد سفارشی از نوع BlackListEvent منتشر می شود. blackListNotifier bean به‌عنوان ApplicationListener ثبت می‌شود و بنابراین BlackListEvent را دریافت می‌کند، در این مرحله می‌تواند به اشخاص مربوطه اطلاع دهد.

##### Annotation-based event listeners

از بهار 4.2، event listenerرا می توان در هر روش عمومی یک لوبیا مدیریت شده از طریق حاشیه نویسی EventListener ثبت کرد. BlackListNotifier را می توان به صورت زیر بازنویسی کرد:

**public** **class** **BlackListNotifier** {

**private** String notificationAddress;

**public** **void** setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

@EventListener

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

}

همانطور که در بالا می بینید، امضای متد یک بار دیگر نوع رویدادی را که به آن گوش می دهد، اعلام می کند، اما این بار با یک نام انعطاف پذیر و بدون اجرای یک رابط شنونده خاص. تا زمانی که نوع رویداد واقعی پارامتر عمومی شما را در سلسله مراتب پیاده سازی خود حل کند، نوع رویداد نیز می تواند از طریق ژنریک محدود شود.

اگر روش شما باید به چندین رویداد گوش دهد یا اگر می‌خواهید آن را بدون هیچ پارامتری تعریف کنید، می‌توانید event type(s)را نیز در خود حاشیه‌نویسی مشخص کنید:

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})

**public** **void** handleContextStart() {

...

}

همچنین می‌توان فیلتر زمان اجرا اضافی را از طریق ویژگی شرط حاشیه‌نویسی اضافه کرد که یک عبارت SpEL را تعریف می‌کند که باید در واقع متد را برای یک رویداد خاص فراخوانی کند.

به عنوان مثال، اعلان کننده ما می تواند بازنویسی شود تا فقط در صورتی فراخوانی شود که ویژگی محتوای رویداد برابر با foo باشد:

@EventListener(condition = "#blEvent.content == 'foo'")

**public** **void** processBlackListEvent(BlackListEvent blEvent) {

*// notify appropriate parties via notificationAddress...*

}

هر عبارت SpEL در برابر یک زمینه اختصاصی ارزیابی می شود. جدول بعدی مواردی را که در اختیار زمینه قرار داده شده اند فهرست می کند تا بتوان از آنها برای پردازش رویداد شرطی استفاده کرد:

| *Table 8. Event SpEL available metadata* | | | |
| --- | --- | --- | --- |
| **Name** | **Location** | **Description** | **Example** |
| Event | root object | The actual ApplicationEvent | #root.event |
| Arguments array | root object | The arguments (as array) used for invoking the target | #root.args[0] |
| *Argument name* | evaluation context | Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where *#arg* stands for the argument index (starting from 0). | #blEvent or #a0 (one can also use #p0 or #p<#arg> notation as an alias). |

توجه داشته باشید که #root.event به شما امکان می دهد به رویداد اصلی دسترسی داشته باشید، حتی اگر امضای متد شما در واقع به یک شی دلخواه که منتشر شده است اشاره کند.

اگر لازم است رویدادی را در نتیجه پردازش رویدادی دیگر منتشر کنید، فقط امضای روش را تغییر دهید تا رویدادی که باید منتشر شود، بازگردانده شود، چیزی شبیه به:

@EventListener

**public** ListUpdateEvent handleBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress and*

*// then publish a ListUpdateEvent...*

}

این ویژگی برای [asynchronous listeners](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#context-functionality-events-async).پشتیبانی نمی شود.

این روش جدید یک ListUpdateEvent جدید برای هر BlackListEvent که توسط روش بالا مدیریت می شود منتشر می کند. اگر نیاز به انتشار چندین رویداد دارید، کافی است به جای آن مجموعه ای از رویدادها را برگردانید.

##### Asynchronous Listeners

اگر می خواهید شنونده خاصی رویدادها را به صورت ناهمزمان پردازش کند، به سادگی از پشتیبانی معمولی @Async استفاده مجدد کنید:

@EventListener

@Async

**public** **void** processBlackListEvent(BlackListEvent event) {

*// BlackListEvent is processed in a separate thread*

}

هنگام استفاده از رویدادهای ناهمزمان از محدودیت های زیر آگاه باشید:

اگر شنونده رویداد یک Exception را پرتاب کند به تماس گیرنده منتشر نمی شود، برای جزئیات بیشتر AsyncUncaughtExceptionHandler را بررسی کنید.

چنین شنونده رویدادی نمی تواند پاسخ ارسال کند. اگر باید رویداد دیگری را در نتیجه پردازش ارسال کنید، ApplicationEventPublisher را تزریق کنید تا رویداد به صورت دستی ارسال شود.

##### Ordering listeners

اگر نیاز دارید که شنونده قبل از شنونده دیگری فراخوانی شود، فقط حاشیه نویسی @Order را به اعلان متد اضافه کنید:

@EventListener

@Order(42)

**public** **void** processBlackListEvent(BlackListEvent event) {

*// notify appropriate parties via notificationAddress...*

}

##### Generic events

همچنین می توانید از ژنریک برای تعریف بیشتر ساختار رویداد خود استفاده کنید. یک EntityCreatedEvent<T>> را در نظر بگیرید که در آن T نوع موجودیت واقعی است که ایجاد شده است. شما می توانید تعریف شنونده زیر را برای دریافت EntityCreatedEvent برای یک شخص ایجاد کنید:

@EventListener

**public** **void** onPersonCreated(EntityCreatedEvent<Person> event) {

...

}

به دلیل پاک کردن نوع، این تنها در صورتی کار می‌کند که رویدادی که اجرا می‌شود، پارامتر(های) عمومی را که شنونده رویداد روی آن فیلتر می‌کند، حل کند (یعنی چیزی شبیه به کلاس PersonCreatedEvent که EntityCreatedEvent<Person> {…​} را گسترش می‌دهد).

در شرایط خاص، اگر همه رویدادها از یک ساختار پیروی کنند (همانطور که باید در مورد رویداد بالا باشد)، ممکن است کاملا خسته کننده شود. در چنین حالتی، می توانید ResolvableTypeProvider را برای هدایت چارچوب فراتر از آنچه که محیط زمان اجرا ارائه می دهد، پیاده سازی کنید:

**public** **class** **EntityCreatedEvent**<T> **extends** ApplicationEvent **implements** ResolvableTypeProvider {

**public** EntityCreatedEvent(T entity) {

super(entity);

}

@Override

**public** ResolvableType getResolvableType() {

**return** ResolvableType.forClassWithGenerics(getClass(),

ResolvableType.forInstance(getSource()));

}

}

این نه تنها برای ApplicationEvent بلکه برای هر شی دلخواه که به عنوان رویداد ارسال می کنید، کار می کند.

#### 1.15.3. Convenient access to low-level resources

برای استفاده بهینه و درک زمینه های برنامه، کاربران باید به طور کلی با انتزاع منابع Spring، همانطور که در فصل منابع توضیح داده شده است، آشنا شوند.

زمینه برنامه یک ResourceLoader است که می تواند برای بارگیری منابع استفاده شود. یک منبع اساساً یک نسخه غنی‌تر از ویژگی‌های کلاس JDK java.net.URL است، در واقع، پیاده‌سازی‌های منبع در صورت لزوم نمونه‌ای از java.net.URL را می‌پیچانند. یک منبع می‌تواند منابع سطح پایین را تقریباً از هر مکانی به صورت شفاف به دست آورد، از جمله از مسیر کلاس، مکان سیستم فایل، هر جایی که با یک URL استاندارد قابل توصیف باشد، و برخی تغییرات دیگر. اگر رشته محل منبع یک مسیر ساده و بدون هیچ پیشوند خاصی باشد، جایی که این منابع از آن می آیند خاص و مناسب با نوع زمینه برنامه کاربردی واقعی است.

شما می توانید یک Bean مستقر در زمینه برنامه را برای پیاده سازی رابط تماس ویژه، ResourceLoaderAware، پیکربندی کنید تا به طور خودکار در زمان اولیه سازی با خود زمینه برنامه به عنوان ResourceLoader ارسال شود. همچنین می‌توانید ویژگی‌های نوع Resource را برای دسترسی به منابع استاتیک استفاده کنید. مانند سایر خواص به آن تزریق خواهند شد. می‌توانید آن ویژگی‌های Resource را به‌عنوان مسیرهای رشته ساده تعیین کنید، و به یک JavaBean PropertyEditor که به‌طور خودکار توسط زمینه ثبت می‌شود، تکیه کنید تا آن رشته‌های متنی را به اشیاء واقعی Resource تبدیل کنید.

مسیر مکان یا مسیرهای ارائه شده به سازنده ApplicationContext در واقع رشته های منبع هستند و به شکل ساده به طور مناسب برای پیاده سازی زمینه خاص رفتار می شوند. ClassPathXmlApplicationContext یک مسیر مکان ساده را به عنوان یک مکان کلاس در نظر می گیرد. همچنین می‌توانید از مسیرهای مکان (رشته‌های منبع) با پیشوندهای خاص برای بارگیری تعاریف از مسیر کلاس یا URL، صرف نظر از نوع بافت واقعی، استفاده کنید.

#### 1.15.4. Convenient ApplicationContext instantiation for web applications

می‌توانید نمونه‌های ApplicationContext را با استفاده از، برای مثال، ContextLoader به‌صورت اعلامی ایجاد کنید. البته شما همچنین می توانید نمونه های ApplicationContext را به صورت برنامه نویسی با استفاده از یکی از پیاده سازی های ApplicationContext ایجاد کنید.

می توانید یک ApplicationContext را با استفاده از ContextLoaderListener به صورت زیر ثبت کنید:

<context-param>

<param-name>contextConfigLocation</param-name>

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

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

شنونده پارامتر contextConfigLocation را بررسی می کند. اگر پارامتر وجود نداشته باشد، شنونده از /WEB-INF/applicationContext.xml به عنوان پیش فرض استفاده می کند. هنگامی که پارامتر وجود داشته باشد، شنونده با استفاده از جداکننده های از پیش تعریف شده (کاما، نقطه ویرگول و فضای سفید) رشته را جدا می کند و از مقادیر به عنوان مکان هایی استفاده می کند که زمینه های برنامه جستجو می شود. الگوهای مسیر به سبک مورچه نیز پشتیبانی می شوند. مثال‌ها عبارتند از: /WEB-INF/\*Context.xml برای همه فایل‌هایی که نام‌های آنها به «Context.xml» ختم می‌شود، که در فهرست «WEB-INF» قرار دارند، و /WEB-INF/\*\*/\*Context.xml، برای همه. چنین فایل هایی در هر زیر شاخه از "WEB-INF".

#### 1.15.5. Deploying a Spring ApplicationContext as a Java EE RAR file

امکان استقرار Spring ApplicationContext به عنوان یک فایل RAR، کپسوله‌سازی متن و تمام کلاس‌های bean مورد نیاز و JAR‌های کتابخانه در یک واحد استقرار Java EE RAR وجود دارد. این معادل راه‌اندازی یک ApplicationContext مستقل است که به تازگی در محیط Java EE میزبانی شده و قادر به دسترسی به امکانات سرورهای Java EE است. استقرار RAR جایگزین طبیعی تری برای سناریوی استقرار یک فایل WAR بدون سر است، در واقع یک فایل WAR بدون هیچ نقطه ورودی HTTP که فقط برای بوت استرپ یک Spring ApplicationContext در محیط Java EE استفاده می شود.

استقرار RAR برای زمینه های برنامه ای ایده آل است که به نقاط ورودی HTTP نیاز ندارند، بلکه فقط شامل نقاط پایانی پیام و کارهای برنامه ریزی شده هستند. Beans در چنین زمینه‌ای می‌تواند از منابع سرور برنامه مانند مدیر تراکنش JTA و JDBC DataSources و JMS ConnectionFactory متصل به JNDI استفاده کند و همچنین ممکن است با سرور JMX پلتفرم ثبت نام کند - همه از طریق مدیریت تراکنش استاندارد Spring و امکانات پشتیبانی JNDI و JMX. اجزای برنامه همچنین می توانند از طریق انتزاع Spring’s TaskExecutor با JCA WorkManager سرور برنامه تعامل داشته باشند.

برای جزئیات پیکربندی مربوط به استقرار RAR، جاوادوک کلاس SpringContextResourceAdapter را بررسی کنید.

برای استقرار ساده یک Spring ApplicationContext به عنوان یک فایل Java EE RAR: تمام کلاس های برنامه را در یک فایل RAR، که یک فایل JAR استاندارد با پسوند فایل متفاوت است، بسته بندی کنید. همه JAR های کتابخانه مورد نیاز را به ریشه آرشیو RAR اضافه کنید. یک توصیفگر استقرار "META-INF/ra.xml" (همانطور که در SpringContextResourceAdapters javadoc نشان داده شده است) و فایل(های) تعریف Spring XML مربوطه (معمولا "META-INF/applicationContext.xml") اضافه کنید و فایل RAR حاصل را رها کنید. به دایرکتوری استقرار سرور برنامه شما.

چنین واحدهای استقرار RAR معمولاً مستقل هستند. آنها اجزای سازنده را در معرض دنیای بیرون قرار نمی دهند، حتی در ماژول های دیگر همان برنامه. تعامل با ApplicationContext مبتنی بر RAR معمولاً از طریق مقاصد JMS که با ماژول‌های دیگر به اشتراک می‌گذارد، اتفاق می‌افتد. یک ApplicationContext مبتنی بر RAR همچنین ممکن است، برای مثال، برخی از کارها را برنامه ریزی کند، به فایل های جدید در سیستم فایل (یا موارد مشابه) واکنش نشان دهد. اگر نیاز به اجازه دسترسی همزمان از خارج داشته باشد، می‌تواند برای مثال نقاط پایانی RMI را صادر کند، که البته ممکن است توسط ماژول‌های کاربردی دیگر در همان دستگاه استفاده شود.

### 1.16. The BeanFactory

BeanFactory API اساس زیربنای عملکرد IoC Spring را فراهم می کند. قراردادهای خاص آن بیشتر در ادغام با سایر بخش‌های Spring و چارچوب‌های شخص ثالث مرتبط استفاده می‌شود و اجرای DefaultListableBeanFactory آن یک نماینده کلیدی در ظرف سطح بالاتر GenericApplicationContext است.

BeanFactory و رابط‌های مرتبط مانند BeanFactoryAware، InitializingBean، DisposableBean نقاط ادغام مهم برای سایر اجزای چارچوب هستند: بدون نیاز به هیچ حاشیه‌نویسی یا حتی بازتابی، امکان تعامل بسیار کارآمد بین ظرف و اجزای آن را فراهم می‌کنند. دانه‌های سطح برنامه ممکن است از همان واسط‌های برگشت تماس استفاده کنند، اما معمولاً به جای آن، تزریق وابستگی اعلامی را ترجیح می‌دهند، یا از طریق حاشیه‌نویسی یا از طریق پیکربندی برنامه‌ای.

توجه داشته باشید که سطح هسته BeanFactory API و اجرای DefaultListableBeanFactory آن فرضیاتی را در مورد قالب پیکربندی یا هیچ حاشیه نویسی جزء مورد استفاده ایجاد نمی کند. همه این طعم‌ها از طریق برنامه‌های افزودنی مانند XmlBeanDefinitionReader و AutowiredAnnotationBeanPostProcessor وارد می‌شوند که بر روی اشیاء BeanDefinition به‌عنوان یک نمایش ابرداده اصلی کار می‌کنند. این ماهیت چیزی است که کانتینر اسپرینگ را بسیار انعطاف پذیر و قابل گسترش می کند.

بخش زیر تفاوت‌های بین سطوح ظرف BeanFactory و ApplicationContext و پیامدهای مربوط به راه‌اندازی را توضیح می‌دهد.

#### 1.16.1. BeanFactory or ApplicationContext?

از ApplicationContext استفاده کنید مگر اینکه دلیل خوبی برای انجام ندادن آن داشته باشید، با GenericApplicationContext و زیر کلاس آن AnnotationConfigApplicationContext به عنوان پیاده سازی های رایج برای بوت استرپ سفارشی. اینها نقاط ورود اولیه به کانتینر اصلی Spring برای همه اهداف مشترک هستند: بارگیری فایل‌های پیکربندی، راه‌اندازی اسکن مسیر کلاس، ثبت برنامه‌ای تعاریف bean و کلاس‌های حاشیه‌نویسی، و از نسخه 5.0 نیز ثبت تعاریف عملکردی bean.

از آنجا که یک ApplicationContext شامل تمام عملکردهای یک BeanFactory است، به طور کلی روی یک BeanFactory ساده توصیه می شود، به جز برای سناریوهایی که کنترل کامل بر پردازش bean مورد نیاز است. در یک ApplicationContext مانند اجرای GenericApplicationContext، چندین نوع لوبیا بر اساس قرارداد (به عنوان مثال با نام لوبیا یا نوع لوبیا)، به ویژه پس از پردازشگرها شناسایی می شوند، در حالی که یک DefaultListableBeanFactory ساده نسبت به هر گونه لوبیا خاص آگنوستیک است.

برای بسیاری از ویژگی های کانتینر توسعه یافته مانند پردازش حاشیه نویسی و پروکسی AOP، نقطه توسعه BeanPostProcessor ضروری است. اگر فقط از یک DefaultListableBeanFactory ساده استفاده می کنید، چنین پس پردازشگرهایی به طور پیش فرض شناسایی و فعال نمی شوند. این وضعیت می‌تواند گیج‌کننده باشد، زیرا هیچ چیز در پیکربندی bean شما مشکلی ندارد. این کانتینر است که در چنین سناریویی باید به طور کامل از طریق تنظیمات اضافی بوت شود.

جدول زیر ویژگی های ارائه شده توسط رابط ها و پیاده سازی های BeanFactory و ApplicationContext را فهرست می کند.

| *Table 9. Feature Matrix* | | |
| --- | --- | --- |
| **Feature** | BeanFactory | ApplicationContext |
| Bean instantiation/wiring | Yes | Yes |
| Integrated lifecycle management | No | Yes |
| Automatic BeanPostProcessor registration | No | Yes |
| Automatic BeanFactoryPostProcessor registration | No | Yes |
| Convenient MessageSource access (for internalization) | No | Yes |
| Built-in ApplicationEvent publication mechanism | No | Yes |

برای ثبت صریح یک Bean post-processor با DefaultListableBeanFactory، باید addBeanPostProcessor را به صورت برنامه‌نویسی فراخوانی کنید:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

*// populate the factory with bean definitions*

*// now register any needed BeanPostProcessor instances*

factory.addBeanPostProcessor(**new** AutowiredAnnotationBeanPostProcessor());

factory.addBeanPostProcessor(**new** MyBeanPostProcessor());

*// now start using the factory*

برای اعمال یک BeanFactoryPostProcessor به یک DefaultListableBeanFactory ساده، باید متد postProcessBeanFactory آن را فراخوانی کنید:

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(**new** FileSystemResource("beans.xml"));

*// bring in some property values from a Properties file*

PropertyPlaceholderConfigurer cfg = **new** PropertyPlaceholderConfigurer();

cfg.setLocation(**new** FileSystemResource("jdbc.properties"));

*// now actually do the replacement*

cfg.postProcessBeanFactory(factory);

در هر دو مورد، مراحل ثبت صریح ناخوشایند هستند، به همین دلیل است که انواع مختلف ApplicationContext به یک DefaultListableBeanFactory ساده در برنامه‌های کاربردی دارای پشتوانه Spring ترجیح داده می‌شوند، به‌ویژه زمانی که به BeanFactoryPostProcessors و BeanPostProcessors برای عملکرد کانتینر گسترده در یک راه‌اندازی معمولی سازمانی متکی هستند.

AnnotationConfigApplicationContext دارای تمام پس پردازشگرهای حاشیه نویسی رایج است که خارج از جعبه ثبت شده اند و ممکن است از طریق حاشیه نویسی های پیکربندی مانند @EnableTransactionManagement، پردازنده های اضافی را در زیر جلدها وارد کنند. در سطح انتزاعی مدل پیکربندی مبتنی بر حاشیه نویسی اسپرینگ، مفهوم پس پردازشگرهای لوبیا به یک جزئیات ظرف داخلی صرف تبدیل می شود.

# 2. Resources

### 2.1. Introduction

متأسفانه کلاس java.net.URL استاندارد جاوا و کنترل‌کننده‌های استاندارد برای پیشوندهای URL مختلف به اندازه کافی برای همه دسترسی‌ها به منابع سطح پایین کافی نیستند. به عنوان مثال، هیچ پیاده سازی URL استاندارد شده ای وجود ندارد که ممکن است برای دسترسی به منبعی که باید از مسیر کلاس یا نسبت به ServletContext به دست آید، استفاده شود. در حالی که امکان ثبت کنترل‌کننده‌های جدید برای پیشوندهای URL تخصصی وجود دارد (شبیه به کنترل‌کننده‌های موجود برای پیشوندهایی مانند http:)، این معمولاً بسیار پیچیده است و رابط URL همچنان فاقد برخی عملکردهای مطلوب است، مانند روشی برای بررسی وجود از منبعی که به آن اشاره شده است.

### 2.2. The Resource interface

رابط Spring's Resource به این معناست که رابطی توانمندتر برای دسترسی انتزاعی به منابع سطح پایین باشد.

**public** **interface** **Resource** **extends** InputStreamSource {

**boolean** exists();

**boolean** isOpen();

URL getURL() **throws** IOException;

File getFile() **throws** IOException;

Resource createRelative(String relativePath) **throws** IOException;

String getFilename();

String getDescription();

}

**public** **interface** **InputStreamSource** {

InputStream getInputStream() **throws** IOException;

}

برخی از مهم ترین روش های رابط منابع عبارتند از:

• getInputStream(): منبع را مکان یابی و باز می کند و یک InputStream را برای خواندن از منبع برمی گرداند. انتظار می رود که هر فراخوانی یک جریان ورودی جدید را برگرداند. مسئولیت بستن جریان به عهده تماس گیرنده است.

• exists(): یک بولی را برمی گرداند که نشان می دهد آیا این منبع واقعاً به شکل فیزیکی وجود دارد یا خیر.

• isOpen(): یک بولی برمی گرداند که نشان می دهد آیا این منبع یک دسته با یک جریان باز را نشان می دهد یا خیر. اگر درست باشد، InputStream را نمی توان چندین بار خواند و باید فقط یک بار خوانده شود و سپس بسته شود تا از نشت منابع جلوگیری شود. برای همه پیاده سازی های معمول منابع، به استثنای InputStreamResource، نادرست خواهد بود.

• getDescription(): توضیحاتی را برای این منبع برمی گرداند تا برای خروجی خطا هنگام کار با منبع استفاده شود. این اغلب نام فایل کاملا واجد شرایط یا URL واقعی منبع است.

روش‌های دیگر به شما امکان می‌دهند یک URL واقعی یا شی فایل به دست آورید که منبع را نشان می‌دهد (اگر پیاده‌سازی زیربنایی سازگار است و از آن عملکرد پشتیبانی می‌کند).

انتزاع منبع به طور گسترده در خود Spring استفاده می شود، به عنوان یک نوع آرگومان در بسیاری از امضاهای متد زمانی که به یک منبع نیاز است. روش‌های دیگر در برخی از APIهای Spring (مانند سازنده‌های پیاده‌سازی‌های مختلف ApplicationContext)، رشته‌ای را می‌گیرند که به شکل ساده یا بدون تزئین برای ایجاد یک منبع مناسب برای اجرای متن استفاده می‌شود، یا از طریق پیشوندهای خاص در مسیر String، به تماس‌گیرنده اجازه می‌دهد. برای تعیین اینکه یک پیاده سازی منبع خاص باید ایجاد و استفاده شود.

در حالی که رابط Resource با Spring و Spring بسیار استفاده می شود، در واقع استفاده از آن به عنوان یک کلاس ابزار عمومی به تنهایی در کد خود، برای دسترسی به منابع، حتی زمانی که کد شما هیچ چیز دیگری را نمی شناسد یا به آن اهمیت نمی دهد، بسیار مفید است. بخش هایی از بهار در حالی که این کد شما را با Spring جفت می کند، در واقع فقط آن را با مجموعه کوچکی از کلاس های کاربردی جفت می کند، که به عنوان جایگزینی بهتر برای URL عمل می کنند، و می توانند معادل هر کتابخانه دیگری که برای این منظور استفاده می کنید در نظر گرفته شود.

مهم است که توجه داشته باشید که انتزاع منبع جایگزین عملکرد نمی شود: آن را در جایی که ممکن است بسته بندی می کند. برای مثال، یک UrlResource یک URL را می‌پیچد و از URL پیچیده برای انجام کار خود استفاده می‌کند.

### 2.3. Built-in Resource implementations

تعدادی از پیاده سازی منابع وجود دارد که مستقیماً از جعبه در Spring عرضه می شوند:

#### 2.3.1. UrlResource

UrlResource یک java.net.URL را می پوشاند و ممکن است برای دسترسی به هر شی که معمولاً از طریق URL قابل دسترسی است استفاده شود، مانند فایل ها، یک هدف HTTP، یک هدف FTP و غیره. همه URL ها دارای یک نمایش رشته استاندارد شده هستند، به طوری که پیشوندهای استاندارد شده مناسب برای نشان دادن یک نوع URL از دیگری استفاده می شود. این شامل فایل: برای دسترسی به مسیرهای سیستم فایل، http: برای دسترسی به منابع از طریق پروتکل HTTP، ftp: برای دسترسی به منابع از طریق FTP و غیره است.

یک UrlResource توسط کد جاوا به طور صریح و با استفاده از سازنده UrlResource ایجاد می‌شود، اما اغلب زمانی که یک متد API را فراخوانی می‌کنید که آرگومان String را می‌گیرد و به‌منظور نمایش یک مسیر است، به‌طور ضمنی ایجاد می‌شود. برای مورد دوم، JavaBeans PropertyEditor در نهایت تصمیم می گیرد که کدام نوع منبع را ایجاد کند. اگر رشته مسیر حاوی چند پیشوند شناخته شده (یعنی) مانند classpath: باشد، یک منبع تخصصی مناسب برای آن پیشوند ایجاد می کند. با این حال، اگر پیشوند را تشخیص ندهد، فرض می‌کند که این فقط یک رشته URL استاندارد است و یک UrlResource ایجاد می‌کند.

#### 2.3.2. ClassPathResource

این کلاس منبعی را نشان می دهد که باید از classpath به دست آید. این کار از بارگذار کلاس بافت رشته، بارگذار کلاس معین یا کلاس معین برای بارگذاری منابع استفاده می کند.

اگر منبع مسیر کلاس در سیستم فایل قرار داشته باشد، این پیاده‌سازی منبع از وضوح به‌عنوان java.io.File پشتیبانی می‌کند، اما نه برای منابع classpath که در یک jar قرار دارند و (توسط موتور servlet یا هر محیط دیگری که هست) گسترش نیافته‌اند. سیستم فایل برای رفع این مشکل، پیاده‌سازی منابع مختلف همیشه از وضوح به عنوان java.net.URL پشتیبانی می‌کند.

یک ClassPathResource توسط کد جاوا به طور صریح و با استفاده از سازنده ClassPathResource ایجاد می‌شود، اما اغلب زمانی که یک متد API را فراخوانی می‌کنید که یک آرگومان String را می‌گیرد که برای نمایش یک مسیر است، به طور ضمنی ایجاد می‌شود. برای مورد دوم، JavaBeans PropertyEditor پیشوند کلاس مسیر: را در مسیر رشته تشخیص می‌دهد و در آن حالت یک ClassPathResource ایجاد می‌کند.

#### 2.3.3. FileSystemResource

این یک پیاده‌سازی منبع برای دسته‌های java.io.File است. واضح است که از وضوح به عنوان یک فایل و به عنوان یک URL پشتیبانی می کند.

#### 2.3.4. ServletContextResource

این یک پیاده سازی منبع برای منابع ServletContext است که مسیرهای نسبی را در دایرکتوری اصلی برنامه وب مربوطه تفسیر می کند.

این همیشه از دسترسی جریانی و دسترسی به URL پشتیبانی می‌کند، اما فقط زمانی اجازه دسترسی به فایل java.io.File را می‌دهد که بایگانی برنامه وب گسترش یافته و منبع به صورت فیزیکی روی سیستم فایل باشد. این که آیا گسترش یافته و در سیستم فایل مانند این است یا نه، یا مستقیماً از JAR یا جای دیگری مانند DB قابل دسترسی است یا نه، در واقع به ظرف Servlet بستگی دارد.

#### 2.3.5. InputStreamResource

پیاده سازی منبع برای یک جریان ورودی معین. این تنها در صورتی باید مورد استفاده قرار گیرد که هیچ پیاده سازی منبع خاصی قابل اجرا نباشد. به طور خاص، ByteArrayResource یا هر یک از پیاده سازی منابع مبتنی بر فایل را در صورت امکان ترجیح دهید.

برخلاف سایر پیاده‌سازی‌های Resource، این یک توصیفگر برای یک منبع از قبل باز شده است - بنابراین مقدار true را از ()isOpen برمی‌گرداند. اگر نیاز دارید توصیفگر منبع را در جایی نگه دارید یا اگر نیاز دارید یک جریان را چندین بار بخوانید از آن استفاده نکنید.

#### 2.3.6. ByteArrayResource

این یک پیاده سازی منبع برای یک آرایه بایت معین است. یک ByteArrayInputStream برای آرایه بایت داده شده ایجاد می کند.

این برای بارگیری محتوا از هر آرایه بایت معین، بدون نیاز به توسل به یک InputStreamResource یکبار مصرف مفید است.

### 2.4. The ResourceLoader

رابط ResourceLoader به این معناست که توسط اشیایی که می‌توانند نمونه‌های منبع را برگردانند (یعنی بارگذاری) پیاده‌سازی شوند.

**public** **interface** **ResourceLoader** {

Resource getResource(String location);

}

رابط ResourceLoader به این معناست که توسط اشیایی که می‌توانند نمونه‌های منبع را برگردانند (یعنی بارگذاری) پیاده‌سازی شوند.

هنگامی که getResource() را در یک زمینه برنامه خاص فراخوانی می‌کنید، و مسیر مکان مشخص‌شده دارای پیشوند خاصی نیست، یک نوع Resource را دریافت خواهید کرد که برای آن زمینه برنامه خاص مناسب است. به عنوان مثال، فرض کنید قطعه کد زیر در برابر یک نمونه ClassPathXmlApplicationContext اجرا شده است:

Resource template = ctx.getResource("some/resource/path/myTemplate.txt");

چیزی که برگردانده می شود یک ClassPathResource خواهد بود. اگر همین روش در برابر یک نمونه FileSystemXmlApplicationContext اجرا شود، یک FileSystemResource را برمی‌گردانید. برای WebApplicationContext، می‌توانید یک ServletContextResource و غیره را پس بگیرید.

به این ترتیب، می توانید منابع را به شیوه ای متناسب با زمینه برنامه خاص بارگیری کنید.

از سوی دیگر، می‌توانید ClassPathResource را بدون در نظر گرفتن نوع زمینه برنامه، با تعیین پیشوند classpath ویژه مجبور به استفاده کنید:

Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");

به طور مشابه، می‌توان با تعیین هر یک از پیشوندهای استاندارد java.net.URL، یک UrlResource را مجبور به استفاده کرد:

Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");

Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");

جدول زیر استراتژی تبدیل رشته ها به منابع را خلاصه می کند:

| *Table 10. Resource strings* | | |
| --- | --- | --- |
| **Prefix** | **Example** | **Explanation** |
| classpath: | classpath:com/myapp/config.xml | Loaded from the classpath. |
| file: | [file:///data/config.xml](file:///\\data\config.xml) | Loaded as a URL, from the  filesystem. [[3](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/core.html#_footnote_3)] |
| http: | <https://myserver/logo.png> | Loaded as a URL. |
| (none) | /data/config.xml | Depends on the underlying  ApplicationContext. |

### 2.5. The ResourceLoaderAware interface

رابط ResourceLoaderAware یک واسط برگشت تماس ویژه است که اجزایی را که انتظار دارند با یک مرجع ResourceLoader ارائه شوند را شناسایی می کند:

**public** **interface** **ResourceLoaderAware** {

**void** setResourceLoader(ResourceLoader resourceLoader);

}

هنگامی که یک کلاس ResourceLoaderAware را پیاده سازی می کند و در یک زمینه برنامه (به عنوان یک Bean مدیریت شده توسط Spring) مستقر می شود، توسط زمینه برنامه به عنوان ResourceLoaderAware شناخته می شود. سپس زمینه برنامه، setResourceLoader (ResourceLoader) را فراخوانی می کند، و خود را به عنوان آرگومان ارائه می کند (به یاد داشته باشید، تمام زمینه های برنامه در Spring رابط ResourceLoader را پیاده سازی می کنند).

البته، از آنجایی که یک ApplicationContext یک ResourceLoader است، bean می‌تواند رابط ApplicationContextAware را نیز پیاده‌سازی کند و از زمینه برنامه ارائه‌شده مستقیماً برای بارگیری منابع استفاده کند، اما به طور کلی، اگر تمام آنچه نیاز است، بهتر است از رابط تخصصی ResourceLoader استفاده کنید. کد فقط با رابط بارگیری منبع همراه می شود، که می تواند یک رابط کاربردی در نظر گرفته شود، و نه کل رابط Spring ApplicationContext.

از بهار 2.5، می توانید به سیم کشی خودکار ResourceLoader به عنوان جایگزینی برای پیاده سازی رابط ResourceLoaderAware اعتماد کنید. حالت‌های سیم‌کشی خودکار سازنده «سنتی» و byType (همانطور که در Autowiring همکاران توضیح داده شد) اکنون می‌توانند یک وابستگی از نوع ResourceLoader را به ترتیب برای آرگومان سازنده یا پارامتر متد تنظیم‌کننده فراهم کنند. برای انعطاف‌پذیری بیشتر (از جمله قابلیت سیم‌کشی خودکار فیلدها و روش‌های پارامترهای متعدد)، از ویژگی‌های جدید سیم‌کشی خودکار مبتنی بر حاشیه‌نویسی استفاده کنید. در آن صورت، ResourceLoader به صورت خودکار به یک فیلد، آرگومان سازنده یا پارامتر متد متصل می شود که انتظار نوع ResourceLoader را دارد تا زمانی که فیلد، سازنده یا متد مورد نظر دارای حاشیه نویسی @Autowired باشد. برای اطلاعات بیشتر، @Autowired را ببینید.

### 2.6. Resources as dependencies

اگر خود bean قرار است مسیر منبع را از طریق نوعی فرآیند پویا تعیین و عرضه کند، احتمالا منطقی است که bean از رابط ResourceLoader برای بارگیری منابع استفاده کند. به عنوان مثال بارگذاری یک الگو را در نظر بگیرید که در آن منبع خاصی که مورد نیاز است به نقش کاربر بستگی دارد. اگر منابع ثابت هستند، منطقی است که استفاده از رابط ResourceLoader را به طور کامل حذف کنید، و فقط باید از bean خصوصیات Resource مورد نیاز خود را نشان دهد و انتظار داشته باشید که آنها به آن تزریق شوند.

چیزی که تزریق این ویژگی‌ها را بی‌اهمیت می‌کند، این است که همه زمینه‌های برنامه ثبت می‌شوند و از JavaBeans PropertyEditor خاصی استفاده می‌کنند که می‌تواند مسیرهای رشته را به اشیاء منبع تبدیل کند. بنابراین اگر myBean دارای یک ویژگی الگو از نوع Resource باشد، می توان آن را با یک رشته ساده برای آن منبع به صورت زیر پیکربندی کرد:

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

<property name="template" value="some/resource/path/myTemplate.txt"/>

</bean>

توجه داشته باشید که مسیر منبع هیچ پیشوندی ندارد، بنابراین از آنجایی که خود زمینه برنامه به عنوان ResourceLoader استفاده می شود، خود منبع بسته به نوع دقیق زمینه، از طریق یک ClassPathResource، FileSystemResource یا ServletContextResource (در صورت لزوم) بارگذاری می شود. .

اگر نیاز به اجبار یک نوع منبع خاص برای استفاده وجود داشته باشد، ممکن است از یک پیشوند استفاده شود. دو مثال زیر نشان می‌دهد که چگونه یک ClassPathResource و یک UrlResource (که دومی برای دسترسی به فایل سیستم استفاده می‌شود) مجبور شود.

<property name="template" value="classpath:some/resource/path/myTemplate.txt">

<property name="template" value="file:///some/resource/path/myTemplate.txt"/>

### 2.7. Application contexts and Resource paths

#### 2.7.1. Constructing application contexts

یک سازنده زمینه برنامه (برای یک نوع زمینه برنامه خاص) به طور کلی یک رشته یا آرایه از رشته ها را به عنوان مسیر(های) محل منبع(ها) مانند فایل های XML که تعریف زمینه را تشکیل می دهند، می گیرد.

هنگامی که چنین مسیر مکانی دارای پیشوند نباشد، نوع منبع خاصی که از آن مسیر ساخته شده و برای بارگذاری تعاریف bean استفاده می‌شود، بستگی به زمینه برنامه خاص دارد و متناسب با آن است. به عنوان مثال، اگر یک ClassPathXmlApplicationContext به صورت زیر ایجاد کنید:

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("conf/appContext.xml");

تعاریف bean از classpath بارگذاری می‌شوند، زیرا از یک ClassPathResource استفاده می‌شود. اما اگر یک FileSystemXmlApplicationContext را به صورت زیر ایجاد کنید:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/appContext.xml");

تعریف bean از یک مکان سیستم فایل بارگذاری می شود، در این مورد نسبت به دایرکتوری کاری فعلی.

توجه داشته باشید که استفاده از پیشوند classpath ویژه یا پیشوند URL استاندارد در مسیر مکان، نوع پیش‌فرض منبع ایجاد شده برای بارگذاری تعریف را لغو می‌کند. بنابراین این FileSystemXmlApplicationContext…

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("classpath:conf/appContext.xml");

در واقع تعاریف bean خود را از classpath بارگذاری می کند. با این حال، هنوز یک FileSystemXmlApplicationContext است. اگر بعداً به عنوان یک ResourceLoader استفاده شود، هر مسیر بدون پیشوند همچنان به عنوان مسیرهای سیستم فایل تلقی می شود.

##### Constructing ClassPathXmlApplicationContext instances - shortcuts

ClassPathXmlApplicationContext تعدادی سازنده را برای فعال کردن نمونه سازی راحت در معرض نمایش می گذارد. ایده اصلی این است که فرد صرفاً یک آرایه رشته ای حاوی نام فایل های خود فایل های XML (بدون اطلاعات مسیر اصلی) و همچنین یک کلاس ارائه می کند. ClassPathXmlApplicationContext اطلاعات مسیر را از کلاس ارائه شده استخراج می کند.

یک مثال امیدواریم این موضوع را روشن کند. یک طرح دایرکتوری را در نظر بگیرید که به شکل زیر است:

com/

foo/

services.xml

daos.xml

MessengerService.class

یک نمونه ClassPathXmlApplicationContext متشکل از دانه‌های تعریف‌شده در «services.xml» و «daos.xml» می‌تواند به این صورت نمونه‌سازی شود…

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(

**new** String**[]** {"services.xml", "daos.xml"}, MessengerService.class);

#### 2.7.2. Wildcards in application context constructor resource paths

مسیرهای منبع در مقادیر سازنده زمینه برنامه ممکن است یک مسیر ساده (همانطور که در بالا نشان داده شده است) باشد که دارای یک نگاشت یک به یک به یک منبع هدف است، یا به طور متناوب ممکن است حاوی پیشوند ویژه "classpath\*:" و/یا Ant- داخلی باشد. عبارات منظم سبک (با استفاده از ابزار Spring's PathMatcher مطابقت داده شده است). هر دوی دومی به طور موثر وایلدکارت هستند

یکی از کاربردهای این مکانیسم هنگام مونتاژ برنامه به سبک جزء است. همه مؤلفه‌ها می‌توانند قطعات تعریف زمینه را در یک مسیر مکان شناخته شده «انتشار» کنند، و زمانی که زمینه برنامه نهایی با استفاده از همان مسیر با پیشوند از طریق classpath\*: ایجاد می‌شود، همه قطعات مؤلفه به‌طور خودکار انتخاب می‌شوند.

توجه داشته باشید که این حروف عام مخصوص استفاده از مسیرهای منبع در سازندگان زمینه برنامه (یا هنگام استفاده مستقیم از سلسله مراتب کلاس ابزار PathMatcher) است و در زمان ساخت حل می شود. ربطی به خود نوع Resource ندارد. استفاده از پیشوند classpath\*: برای ساختن یک منبع واقعی امکان پذیر نیست، زیرا یک منبع در هر زمان فقط به یک منبع اشاره می کند.

##### Ant-style Patterns

هنگامی که مکان مسیر شامل یک الگوی Ant-style باشد، برای مثال:

/WEB-INF/\*-context.xml

com/mycompany/\*\*/applicationContext.xml

file:C:/some/path/\*-context.xml

classpath:com/mycompany/\*\*/applicationContext.xml

resolver از یک روش پیچیده‌تر اما تعریف‌شده پیروی می‌کند تا بخواهد علامت عام را حل کند. این یک منبع برای مسیر تا آخرین بخش غیر عام ایجاد می کند و یک URL از آن به دست می آورد. اگر این URL یک jar: URL یا نوع خاص ظرف نباشد (به عنوان مثال zip: در WebLogic، wsjar در WebSphere، و غیره)، یک java.io.File از آن به دست می‌آید و با عبور از سیستم فایل برای حل عام استفاده می‌شود. . در مورد URL jar، حل‌کننده یا یک اتصال java.net.JarURLC از آن دریافت می‌کند یا به صورت دستی URL jar را تجزیه می‌کند و سپس محتویات فایل jar را پیمایش می‌کند تا عبارات را حل کند.

###### Implications on portability

اگر مسیر مشخص شده قبلاً یک URL فایل است (به طور صریح یا ضمنی زیرا ResourceLoader پایه یک سیستم فایل است)، پس تضمین می شود که wildcarding به شکل کاملاً قابل حمل کار کند.

اگر مسیر مشخص شده یک مکان classpath باشد، پس حل‌کننده باید آخرین URL بخش مسیر غیر عام را از طریق فراخوانی ()Classloader.getResource بدست آورد. از آنجایی که این فقط یک گره از مسیر است (نه فایل در انتها)، در واقع تعریف نشده است (در جاوادوکس ClassLoader) دقیقاً چه نوع URL در این مورد بازگردانده می شود. در عمل، همیشه یک java.io.File نشان‌دهنده دایرکتوری است، جایی که منبع classpath به یک مکان فایل سیستم حل می‌شود، یا یک URL jar که در آن منبع classpath به یک مکان jar حل می‌شود. با این حال، نگرانی قابل حمل در مورد این عملیات وجود دارد.

اگر یک URL jar برای آخرین بخش غیر عام به دست می آید، حل کننده باید بتواند یک اتصال java.net.JarURLC را از آن دریافت کند، یا به صورت دستی URL jar را تجزیه کند تا بتواند محتویات jar را طی کند و حل کند. عام این در اکثر محیط‌ها کار می‌کند، اما در محیط‌های دیگر با شکست مواجه می‌شود، و اکیداً توصیه می‌شود که وضوح عام منابعی که از jars می‌آیند، قبل از اینکه به آن تکیه کنید، به‌طور کامل در محیط خاص شما آزمایش شود.

##### The classpath\*: prefix

هنگام ساختن یک زمینه برنامه مبتنی بر XML، یک رشته مکان ممکن است از پیشوند کلاس\*: استفاده کند:

ApplicationContext ctx =

**new** ClassPathXmlApplicationContext("classpath\*:conf/appContext.xml");

این پیشوند خاص مشخص می کند که تمام منابع classpath که با نام داده شده مطابقت دارند باید به دست آیند (در داخل، این اساساً از طریق فراخوانی ClassLoader.getResources (…​) اتفاق می افتد)، و سپس ادغام شوند تا تعریف زمینه برنامه نهایی را تشکیل دهند.

مسیر کلاسی wildcard به متد getResources() کلاس لودر زیرین متکی است. از آنجایی که امروزه اکثر سرورهای برنامه کاربردی کلاس لودر خود را ارائه می کنند، رفتار ممکن است به خصوص در هنگام کار با فایل های jar متفاوت باشد. یک آزمایش ساده برای بررسی اینکه آیا classpath\* کار می‌کند یا نه، استفاده از کلاس‌لودر برای بارگذاری یک فایل از داخل یک jar در classpath است: getClass().getClassLoader().getResources("<someFileInsideTheJar>"). این تست را با فایل هایی که نام یکسانی دارند اما در دو مکان متفاوت قرار می گیرند، امتحان کنید. در صورت بازگشت نتیجه نامناسب، اسناد سرور برنامه را برای تنظیماتی که ممکن است بر رفتار کلاس‌لودر تأثیر بگذارد بررسی کنید.

پیشوند classpath\*: همچنین می تواند با یک الگوی PathMatcher در بقیه مسیر مکان ترکیب شود، برای مثال classpath\*:META-INF/\*-beans.xml. در این مورد، استراتژی وضوح نسبتاً ساده است: یک فراخوانی ClassLoader.getResources در آخرین بخش مسیر غیر عام استفاده می‌شود تا تمام منابع منطبق در سلسله مراتب کلاس loader را دریافت کند، و سپس هر منبع همان استراتژی وضوح PathMatcher را خاموش کند. شرح داده شده در بالا برای زیرمسیر عام استفاده می شود.

##### Other notes relating to wildcards

لطفاً توجه داشته باشید که classpath\*: هنگامی که با الگوهای Ant-style ترکیب می‌شود، قبل از شروع الگو فقط با حداقل یک فهرست ریشه کار می‌کند، مگر اینکه فایل‌های هدف واقعی در سیستم فایل قرار داشته باشند. این به این معنی است که الگویی مانند classpath\*:\*.xml ممکن است فایل ها را از ریشه فایل های jar بازیابی نکند، بلکه فقط از ریشه دایرکتوری های توسعه یافته است.

توانایی Spring برای بازیابی ورودی‌های classpath از متد ClassLoader.getResources () JDK سرچشمه می‌گیرد که فقط مکان‌های سیستم فایل را برای یک رشته خالی ارسال شده (که ریشه‌های بالقوه برای جستجو را نشان می‌دهد) برمی‌گرداند. Spring پیکربندی زمان اجرا URLClassLoader و مانیفست "java.class.path" را در فایل‌های jar نیز ارزیابی می‌کند، اما تضمین نمی‌شود که منجر به رفتار قابل حمل شود.\

اسکن بسته های classpath به وجود ورودی های دایرکتوری مربوطه در classpath نیاز دارد. هنگامی که JAR ها را با Ant می سازید، مطمئن شوید که سوئیچ فقط فایل وظیفه JAR را فعال نکنید. همچنین، دایرکتوری های classpath ممکن است بر اساس سیاست های امنیتی در برخی محیط ها، به عنوان مثال، در معرض دید قرار نگیرند. برنامه‌های مستقل در JDK 1.7.0\_45 و بالاتر (که نیاز به راه‌اندازی «Trusted-Library» در مانیفست‌های شما دارد؛ https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources را ببینید).

در مسیر ماژول JDK 9 (Jigsaw)، اسکن مسیر کلاس Spring به طور کلی همانطور که انتظار می رود کار می کند. قرار دادن منابع در یک دایرکتوری اختصاصی در اینجا نیز بسیار توصیه می شود و از مشکلات حمل و نقل ذکر شده در جستجوی سطح ریشه فایل jar اجتناب می شود.

Ant-style با classpath: اگر بسته ریشه برای جستجو در مکان‌های مسیر کلاس چندگانه موجود باشد، منابع تضمینی برای یافتن منابع منطبق نیستند. این به این دلیل است که منبعی مانند

com/mycompany/package1/service-context.xml

ممکن است فقط در یک مکان باشد، اما زمانی که مسیری مانند

classpath:com/mycompany/\*\*/service-context.xml

برای حل آن استفاده می شود، حل کننده با URL (اولین) برگشتی توسط getResource("com/mycompany") کار می کند. اگر این گره بسته پایه در چندین مکان کلاس لودر وجود داشته باشد، ممکن است منبع نهایی واقعی زیر آن نباشد. بنابراین، ترجیحاً در چنین حالتی از "`classpath\*:" با همان الگوی Ant-style استفاده کنید، که تمام مکان های مسیر کلاس را که حاوی بسته root هستند جستجو می کند.

#### 2.7.3. FileSystemResource caveats

یک FileSystemResource که به FileSystemApplicationContext متصل نیست (یعنی FileSystemApplicationContext ResourceLoader واقعی نیست) مسیرهای مطلق و نسبی را همانطور که انتظار دارید رفتار می کند. مسیرهای نسبی نسبت به دایرکتوری کاری فعلی هستند، در حالی که مسیرهای مطلق نسبت به ریشه سیستم فایل هستند.

اما به دلایل سازگاری با عقب (تاریخی)، زمانی که FileSystemApplicationContext ResourceLoader باشد، این تغییر می کند. FileSystemApplicationContext به سادگی تمام نمونه های FileSystemResource پیوست شده را مجبور می کند تا همه مسیرهای مکان را نسبی در نظر بگیرند، خواه با یک اسلش اصلی شروع شوند یا نه. در عمل، این بدان معنی است که موارد زیر معادل هستند:

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("conf/context.xml");

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("/conf/context.xml");

مانند موارد زیر: (اگرچه متفاوت بودن آنها منطقی است، زیرا یک مورد نسبی و دیگری مطلق است.)

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("some/resource/path/myTemplate.txt");

FileSystemXmlApplicationContext ctx = ...;

ctx.getResource("/some/resource/path/myTemplate.txt");

در عمل، اگر به مسیرهای سیستم فایل مطلق واقعی نیاز دارید، بهتر است از استفاده از مسیرهای مطلق با FileSystemResource / FileSystemXmlApplicationContext صرف نظر کنید و فقط با استفاده از پیشوند file: URL، از UrlResource استفاده کنید.

*// actual context type doesn't matter, the Resource will always be UrlResource*

ctx.getResource("file:///some/resource/path/myTemplate.txt");

*// force this FileSystemXmlApplicationContext to load its definition via a UrlResource*

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext("file:///conf/context.xml");

# 3. Validation, Data Binding, and Type Conversion

### 3.1. Introduction

JSR-303/JSR-349 Bean Validation

Spring Framework 4.0 از Bean Validation 1.0 (JSR-303) و Bean Validation 1.1 (JSR-349) از لحاظ پشتیبانی از راه اندازی پشتیبانی می کند، همچنین آن را با رابط اعتبار سنجی Spring تطبیق می دهد.

یک برنامه کاربردی می تواند انتخاب کند که اعتبار سنجی Bean را یک بار در سطح جهانی فعال کند، همانطور که در اعتبار سنجی Spring توضیح داده شده است، و از آن به طور انحصاری برای همه نیازهای اعتبار سنجی استفاده کند.

یک برنامه همچنین می‌تواند نمونه‌های Spring Validator اضافی را در هر نمونه DataBinder ثبت کند، همانطور که در پیکربندی DataBinder توضیح داده شده است. این ممکن است برای وصل کردن منطق اعتبارسنجی بدون استفاده از حاشیه‌نویسی مفید باشد.

در نظر گرفتن اعتبار سنجی به عنوان منطق تجاری، جوانب مثبت و منفی وجود دارد، و اسپرینگ طرحی را برای اعتبارسنجی (و اتصال داده ها) ارائه می دهد که هیچ یک از آنها را حذف نمی کند. به طور خاص اعتبار سنجی نباید به لایه وب گره بخورد، باید به راحتی بومی سازی شود و باید بتوان هر اعتبارسنجی موجود را وصل کرد. با توجه به موارد فوق، Spring یک رابط Validator ارائه کرده است که هم پایه و هم قابل استفاده در هر لایه از یک برنامه کاربردی است.

اتصال داده برای اجازه دادن به ورودی کاربر به صورت پویا به مدل دامنه یک برنامه کاربردی (یا هر شیئی که برای پردازش ورودی کاربر استفاده می کنید) مفید است. Spring به اصطلاح DataBinder را برای انجام این کار فراهم می کند. Validator و DataBinder بسته اعتبار سنجی را تشکیل می دهند که عمدتاً در چارچوب MVC استفاده می شود اما محدود به آن نیست.

BeanWrapper یک مفهوم اساسی در چارچوب Spring است و در بسیاری از مکان ها استفاده می شود. با این حال، احتمالاً نیازی به استفاده مستقیم از BeanWrapper نخواهید داشت. با این حال، از آنجایی که این مستندات مرجع است، ما احساس کردیم که ممکن است توضیحی لازم باشد. ما BeanWrapper را در این فصل توضیح خواهیم داد زیرا، اگر اصلاً می‌خواستید از آن استفاده کنید، به احتمال زیاد این کار را هنگام تلاش برای اتصال داده‌ها به اشیا انجام می‌دادید.

Spring's DataBinder و BeanWrapper سطح پایین هر دو از PropertyEditors برای تجزیه و قالب بندی مقادیر ویژگی استفاده می کنند. مفهوم PropertyEditor بخشی از مشخصات JavaBeans است و همچنین در این فصل توضیح داده شده است. Spring 3 یک بسته "core.convert" را معرفی می کند که یک تسهیلات تبدیل نوع عمومی و همچنین یک بسته "فرمت" سطح بالاتر برای قالب بندی مقادیر فیلد رابط کاربری ارائه می دهد. این بسته های جدید ممکن است به عنوان جایگزین های ساده تری برای PropertyEditors استفاده شوند و همچنین در این فصل مورد بحث قرار خواهند گرفت.

### 3.2. Validation using Spring’s Validator interface

Spring دارای یک رابط Validator است که می توانید از آن برای اعتبارسنجی اشیاء استفاده کنید. واسط Validator با استفاده از یک شی Errors کار می کند به طوری که در حین اعتبارسنجی، اعتبار سنجی ها می توانند خرابی های اعتبار سنجی را به شی Errors گزارش دهند.

بیایید یک شی داده کوچک را در نظر بگیریم:

**public** **class** **Person** {

**private** String name;

**private** **int** age;

*// the usual getters and setters...*

}

ما با اجرای دو روش زیر از رابط org.springframework.validation.Validator رفتار اعتبار سنجی را برای کلاس Person ارائه می کنیم:

supports(Class) - آیا این Validator می تواند نمونه های کلاس ارائه شده را تأیید کند؟

validate(Object, org.springframework.validation.Errors) - شی داده شده را تایید می کند و در صورت خطاهای اعتبار سنجی، مواردی را با شی Errors داده شده ثبت می کند.

پیاده سازی Validator نسبتاً ساده است، به خصوص زمانی که از کلاس کمکی ValidationUtils که Spring Framework نیز ارائه می کند، اطلاع دارید.

**public** **class** **PersonValidator** **implements** Validator {

*/\*\**

*\* This Validator validates \*just\* Person instances*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Person.class.equals(clazz);

}

**public** **void** validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

**if** (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} **else** **if** (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

همانطور که می بینید، متد static rejectIfEmpty(..) در کلاس ValidationUtils برای رد ویژگی 'name' در صورتی که null یا رشته خالی باشد استفاده می شود. به ValidationUtils javadocs نگاهی بیندازید تا ببینید علاوه بر مثالی که قبلا نشان داده شد، چه عملکردی را ارائه می دهد.

در حالی که مطمئناً می‌توان یک کلاس Validator را برای اعتبارسنجی هر یک از اشیاء تودرتو در یک شی غنی پیاده‌سازی کرد، شاید بهتر باشد که منطق اعتبارسنجی برای هر کلاس تودرتو از شی را در پیاده‌سازی Validator خودش کپسوله کنیم. یک مثال ساده از یک شی "rich" یک مشتری است که از دو ویژگی String (نام اول و دوم) و یک شی پیچیده Address تشکیل شده است. اشیاء آدرس ممکن است مستقل از اشیاء مشتری استفاده شوند، بنابراین یک AddressValidator مجزا پیاده سازی شده است. اگر می‌خواهید CustomerValidator شما از منطق موجود در کلاس AddressValidator بدون استفاده از کپی و چسباندن مجددا استفاده کند، می‌توانید یک AddressValidator را در CustomerValidator خود وابستگی تزریق کنید یا نمونه‌سازی کنید و از آن به‌صورت زیر استفاده کنید:

**public** **class** **CustomerValidator** **implements** Validator {

**private** **final** Validator addressValidator;

**public** CustomerValidator(Validator addressValidator) {

**if** (addressValidator == null) {

**throw** **new** IllegalArgumentException("The supplied [Validator] is " +

"required and must not be null.");

}

**if** (!addressValidator.supports(Address.class)) {

**throw** **new** IllegalArgumentException("The supplied [Validator] must " +

"support the validation of [Address] instances.");

}

this.addressValidator = addressValidator;

}

*/\*\**

*\* This Validator validates Customer instances, and any subclasses of Customer too*

*\*/*

**public** **boolean** supports(Class clazz) {

**return** Customer.class.isAssignableFrom(clazz);

}

**public** **void** validate(Object target, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");

Customer customer = (Customer) target;

**try** {

errors.pushNestedPath("address");

ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);

} **finally** {

errors.popNestedPath();

}

}

}

خطاهای اعتبار سنجی به شی Errors که به اعتبارسنجی ارسال می شود گزارش می شود. در مورد Spring Web MVC می‌توانید از تگ <spring:bind/> برای بررسی پیام‌های خطا استفاده کنید، اما البته می‌توانید شیء خطا را خودتان نیز بررسی کنید. اطلاعات بیشتر در مورد روش هایی که ارائه می دهد را می توانید در javadocs پیدا کنید.

### 3.3. Resolving codes to error messages

ما در مورد Databinding و اعتبار سنجی صحبت کرده ایم. خروجی پیام های مربوط به خطاهای اعتبارسنجی آخرین چیزی است که باید در مورد آن بحث کنیم. در مثالی که در بالا نشان دادیم، نام و قسمت سن را رد کردیم. اگر می‌خواهیم پیام‌های خطا را با استفاده از یک منبع پیام ارسال کنیم، این کار را با استفاده از کد خطایی که هنگام رد کردن فیلد داده‌ایم انجام می‌دهیم («نام» و «سن» در این مورد). هنگامی که شما (مستقیم یا غیرمستقیم با استفاده از کلاس ValidationUtils) rejectValue یا یکی از روش‌های رد کردن دیگر از رابط Errors را فراخوانی می‌کنید، پیاده‌سازی زیربنایی نه تنها کدی را که ارسال کرده‌اید ثبت می‌کند، بلکه تعدادی از کدهای خطای اضافی کدهای خطایی که ثبت می کند توسط MessageCodesResolver که استفاده می شود تعیین می شود. به طور پیش‌فرض، DefaultMessageCodesResolver استفاده می‌شود، که برای مثال نه تنها پیامی را با کدی که داده‌اید ثبت می‌کند، بلکه پیام‌هایی را نیز شامل نام فیلدی است که به روش رد ارسال کرده‌اید. بنابراین در صورتی که فیلدی را با استفاده از rejectValue ("age"، "too.darn.old") رد کنید، به غیر از کد too.darn.old، Spring نیز Too.darn.old.age و too.darn.old را ثبت می کند. .age.int (بنابراین اولی شامل نام فیلد و دومی شامل نوع فیلد خواهد بود). این به عنوان یک راحتی برای کمک به توسعه دهندگان در هدف قرار دادن پیام های خطا و موارد مشابه انجام می شود.

اطلاعات بیشتر در مورد MessageCodesResolver و استراتژی پیش‌فرض را می‌توانید به ترتیب در جاوادوک MessageCodesResolver و DefaultMessageCodesResolver به صورت آنلاین پیدا کنید.

### 3.4. Bean manipulation and the BeanWrapper

بسته org.springframework.beans به استاندارد JavaBeans ارائه شده توسط Oracle پایبند است. یک JavaBean به سادگی یک کلاس با یک سازنده پیش‌فرض بدون آرگومان است که از یک قرارداد نام‌گذاری پیروی می‌کند که در آن (به عنوان مثال) یک ویژگی به نام bingoMadness دارای یک متد تنظیم کننده setBingoMadness (..) و یک متد دریافت کننده getBingoMadness () است. برای اطلاعات بیشتر در مورد JavaBeans و مشخصات، لطفاً به وب سایت Oracle (javabeans) مراجعه کنید.

یکی از کلاس های بسیار مهم در بسته beans، رابط BeanWrapper و پیاده سازی مربوط به آن (BeanWrapperImpl) است. همانطور که از javadocs نقل شده است، BeanWrapper عملکردی را برای تنظیم و دریافت مقادیر دارایی (به صورت جداگانه یا به صورت عمده)، دریافت توصیفگرهای ویژگی، و پرس و جو از خصوصیات برای تعیین خواندن یا نوشتن آنها ارائه می دهد. همچنین، BeanWrapper از ویژگی‌های تودرتو پشتیبانی می‌کند، و امکان تنظیم ویژگی‌ها را روی ویژگی‌های فرعی تا عمق نامحدود فراهم می‌کند. سپس BeanWrapper از قابلیت افزودن استاندارد JavaBeans PropertyChangeListeners و VetoableChangeListeners بدون نیاز به کد پشتیبانی در کلاس هدف پشتیبانی می کند. آخرین اما نه کم‌اهمیت، BeanWrapper از تنظیم ویژگی‌های نمایه‌شده پشتیبانی می‌کند. BeanWrapper معمولاً توسط کد برنامه به طور مستقیم استفاده نمی شود، بلکه توسط DataBinder و BeanFactory استفاده می شود.

نحوه کار BeanWrapper تا حدی با نام آن مشخص می شود: یک bean را می پیچد تا اقداماتی مانند تنظیم و بازیابی ویژگی ها را روی آن bean انجام دهد.

#### 3.4.1. Setting and getting basic and nested properties

تنظیم و دریافت خصوصیات با استفاده از متدهای setPropertyValue و getPropertyValue(ها) انجام می شود که هر دو با چند نوع اضافه بار همراه هستند. همه آنها در javadocs Spring با جزئیات بیشتر توضیح داده شده اند. آنچه مهم است بدانید این است که چند قرارداد برای نشان دادن ویژگی های یک شی وجود دارد. یکی دو مثال:

| *Table 11. Examples of properties* | |
| --- | --- |
| **Expression** | **Explanation** |
| name | Indicates the property name corresponding to the methods getName() or isName() and setName(..) |
| account.name | Indicates the nested property name of the property account corresponding e.g. to the methods getAccount().setName() or getAccount().getName() |
| account[2] | Indicates the *third* element of the indexed property account. Indexed properties can be of type array, list or other *naturally ordered* collection |
| account[COMPANYNAME] | Indicates the value of the map entry indexed by the key *COMPANYNAME* of the Map property account |

در زیر نمونه هایی از کار با BeanWrapper برای دریافت و تنظیم ویژگی ها را مشاهده خواهید کرد.

(اگر قصد ندارید مستقیماً با BeanWrapper کار کنید، این بخش بعدی برای شما اهمیت حیاتی ندارد. اگر فقط از DataBinder و BeanFactory و پیاده سازی خارج از جعبه آنها استفاده می کنید، باید به ادامه مطلب بروید. بخش مربوط به PropertyEditors.)

دو کلاس زیر را در نظر بگیرید:

**public** **class** **Company** {

**private** String name;

**private** Employee managingDirector;

**public** String getName() {

**return** this.name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** Employee getManagingDirector() {

**return** this.managingDirector;

}

**public** **void** setManagingDirector(Employee managingDirector) {

this.managingDirector = managingDirector;

}

}

**public** **class** **Employee** {

**private** String name;

**private** **float** salary;

**public** String getName() {

**return** this.name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**float** salary) {

this.salary = salary;

}

}

قطعه کد زیر نمونه هایی از نحوه بازیابی و دستکاری برخی از ویژگی های شرکت ها و کارمندان نمونه را نشان می دهد:

BeanWrapper company = **new** BeanWrapperImpl(**new** Company());

*// setting the company name..*

company.setPropertyValue("name", "Some Company Inc.");

*// ... can also be done like this:*

PropertyValue value = **new** PropertyValue("name", "Some Company Inc.");

company.setPropertyValue(value);

*// ok, let's create the director and tie it to the company:*

BeanWrapper jim = **new** BeanWrapperImpl(**new** Employee());

jim.setPropertyValue("name", "Jim Stravinsky");

company.setPropertyValue("managingDirector", jim.getWrappedInstance());

*// retrieving the salary of the managingDirector through the company*

Float salary = (Float) company.getPropertyValue("managingDirector.salary");

#### 3.4.2. Built-in PropertyEditor implementations

Spring از مفهوم PropertyEditors برای اعمال تبدیل بین یک شی و یک رشته استفاده می کند. اگر در مورد آن فکر کنید، گاهی اوقات ممکن است مفید باشد که بتوانید ویژگی ها را به روشی متفاوت از خود شی نشان دهید. به عنوان مثال، یک تاریخ را می توان به روشی قابل خواندن برای انسان نشان داد (به عنوان رشته '09-14-2007')، در حالی که ما هنوز می توانیم شکل قابل خواندن انسان را به تاریخ اصلی تبدیل کنیم (یا حتی بهتر از آن: تبدیل هر کدام تاریخ وارد شده به شکل قابل خواندن توسط انسان، بازگشت به اشیاء تاریخ). این رفتار را می توان با ثبت ویرایشگرهای سفارشی، از نوع java.beans.PropertyEditor به دست آورد. ثبت ویرایشگرهای سفارشی بر روی BeanWrapper یا متناوب در یک ظرف IoC خاص همانطور که در فصل قبل ذکر شد، به آن دانش چگونگی تبدیل ویژگی ها به نوع مورد نظر را می دهد. اطلاعات بیشتر در مورد PropertyEditors را در javadocs بسته java.beans ارائه شده توسط Oracle بخوانید.

چند مثال که در آن ویرایش ویژگی در Spring استفاده می شود:

تنظیم خواص روی bean با استفاده از PropertyEditors انجام می شود. هنگامی که از String به عنوان مقدار خاصیت برخی از bean که در فایل XML اعلام می‌کنید نام می‌برید، Spring (اگر تنظیم‌کننده ویژگی مربوطه دارای پارامتر Class باشد) از ClassEditor برای تلاش برای حل پارامتر به یک شی Class استفاده می‌کند.

تجزیه پارامترهای درخواست HTTP در فریم ورک Spring’s MVC با استفاده از انواع PropertyEditors انجام می شود که می توانید به صورت دستی در تمام زیر کلاس های CommandController متصل شوید.

Spring دارای تعدادی PropertyEditors داخلی برای آسان کردن زندگی است. هر یک از آن‌ها در زیر فهرست شده‌اند و همگی در بسته org.springframework.beans.propertyeditors قرار دارند. بیشتر، اما نه همه (همانطور که در زیر نشان داده شده است)، به طور پیش فرض توسط BeanWrapperImpl ثبت شده اند. در جایی که ویرایشگر ویژگی به نوعی قابل تنظیم است، البته هنوز هم می توانید نوع خود را برای لغو نسخه پیش فرض ثبت کنید:

| *Table 12. Built-in PropertyEditors* | |
| --- | --- |
| **Class** | **Explanation** |
| ByteArrayPropertyEditor | Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by BeanWrapperImpl. |
| ClassEditor | Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown. Registered by default by BeanWrapperImpl. |
| CustomBooleanEditor | Customizable property editor for Boolean properties. Registered by default by BeanWrapperImpl, but, can be overridden by registering custom instance of it as custom editor. |
| CustomCollectionEditor | Property editor for Collections, converting any source Collection to a given target Collection type. |
| CustomDateEditor | Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format. |
| CustomNumberEditor | Customizable property editor for any Number subclass like Integer, Long, Float, Double. Registered by default by BeanWrapperImpl, but can be overridden by registering custom instance of it as a custom editor. |
| FileEditor | Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl. |
| InputStreamEditor | One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource) an InputStream, so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl. |
| LocaleEditor | Capable of resolving Strings to Locale objects and vice versa (the String format is *[country]*[variant], which is the same thing the toString() method of Locale provides). Registered by default by BeanWrapperImpl. |
| PatternEditor | Capable of resolving Strings to java.util.regex.Pattern objects and vice versa. |
| PropertiesEditor | Capable of converting Strings (formatted using the format as defined in the javadocs of the java.util.Properties class) to Properties objects. Registered by default by BeanWrapperImpl. |
| StringTrimmerEditor | Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed. |
| URLEditor | Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl. |

Spring از java.beans.PropertyEditorManager برای تنظیم مسیر جستجو برای ویرایشگرهای ویژگی که ممکن است مورد نیاز باشد استفاده می کند. مسیر جستجو همچنین شامل sun.bean.editors می‌شود که شامل پیاده‌سازی PropertyEditor برای انواعی مانند Font، Color و بسیاری از انواع اولیه است. همچنین توجه داشته باشید که زیرساخت استاندارد JavaBeans به طور خودکار کلاس‌های PropertyEditor را (بدون نیاز به ثبت صریح آن‌ها) در صورتی که در بسته‌ای مشابه با کلاسی که مدیریت می‌کند، باشند و همنام آن کلاس باشند و 'Editor' ضمیمه شده باشند، پیدا می‌کند. برای مثال، می‌توان ساختار کلاس و بسته زیر را داشت، که برای شناسایی و استفاده از کلاس FooEditor به عنوان PropertyEditor برای ویژگی‌های Foo-typed کافی است.

com

chank

pop

Foo

FooEditor // the PropertyEditor for the Foo class

توجه داشته باشید که می توانید از مکانیسم استاندارد BeanInfo JavaBeans نیز در اینجا استفاده کنید (در اینجا با جزئیات نه چندان شگفت انگیز توضیح داده شده است). در زیر نمونه ای از استفاده از مکانیسم BeanInfo برای ثبت صریح یک یا چند نمونه PropertyEditor با ویژگی های یک کلاس مرتبط را بیابید.

com

chank

pop

Foo

FooBeanInfo // the BeanInfo for the Foo class

در اینجا کد منبع جاوا برای کلاس FooBeanInfo ارجاع شده است. این یک CustomNumberEditor را با ویژگی age کلاس Foo مرتبط می کند.

**public** **class** **FooBeanInfo** **extends** SimpleBeanInfo {

**public** PropertyDescriptor**[]** getPropertyDescriptors() {

**try** {

**final** PropertyEditor numberPE = **new** CustomNumberEditor(Integer.class, true);

PropertyDescriptor ageDescriptor = **new** PropertyDescriptor("age", Foo.class) {

**public** PropertyEditor createPropertyEditor(Object bean) {

**return** numberPE;

};

};

**return** **new** PropertyDescriptor**[]** { ageDescriptor };

}

**catch** (IntrospectionException ex) {

**throw** **new** Error(ex.toString());

}

}

}

##### Registering additional custom PropertyEditors

هنگامی که ویژگی‌های bean را به‌عنوان مقدار رشته تنظیم می‌کنیم، یک ظرف Spring IoC در نهایت از JavaBeans PropertyEditors استاندارد برای تبدیل این رشته‌ها به نوع پیچیده ویژگی استفاده می‌کند. Spring تعدادی PropertyEditors سفارشی را از قبل ثبت می کند (به عنوان مثال، برای تبدیل یک نام کلاس که به عنوان یک رشته بیان شده است به یک شی Class واقعی). علاوه بر این، مکانیسم جستجوی استاندارد JavaBeans PropertyEditor جاوا اجازه می دهد تا یک PropertyEditor برای یک کلاس به سادگی نامگذاری مناسبی داشته باشد و در همان بسته کلاسی که از آن پشتیبانی می کند قرار گیرد و به طور خودکار پیدا شود.

اگر نیاز به ثبت سایر PropertyEditor های سفارشی وجود دارد، مکانیسم های مختلفی در دسترس است. دستی ترین روش، که معمولاً راحت یا توصیه نمی شود، استفاده از روش registerCustomEditor() رابط ConfigurableBeanFactory است، با فرض اینکه شما یک مرجع BeanFactory دارید. مکانیسم دیگر، کمی راحت‌تر، استفاده از یک پس‌پردازنده مخصوص کارخانه‌ی لوبیا به نام CustomEditorConfigurer است. اگرچه می‌توان از پس‌پردازنده‌های کارخانه bean با پیاده‌سازی‌های BeanFactory استفاده کرد، CustomEditorConfigurer دارای یک تنظیمات تودرتو است، بنابراین اکیداً توصیه می‌شود که از آن با ApplicationContext استفاده شود، جایی که ممکن است به روشی مشابه هر Bean دیگری مستقر شود و به طور خودکار شناسایی شود. و اعمال شد.

توجه داشته باشید که تمام کارخانه‌های bean و زمینه‌های کاربردی به‌طور خودکار از تعدادی ویرایشگر ویژگی داخلی استفاده می‌کنند، از طریق استفاده از چیزی به نام BeanWrapper برای مدیریت تبدیل‌های دارایی. ویرایشگرهای ویژگی استانداردی که BeanWrapper ثبت می‌کند در بخش قبل فهرست شده‌اند. علاوه بر این، ApplicationContexts همچنین تعداد ویرایشگرهای اضافی را برای رسیدگی به جستجوی منابع به شیوه ای مناسب با نوع زمینه برنامه خاص لغو یا اضافه می کند.

نمونه های استاندارد JavaBeans PropertyEditor برای تبدیل مقادیر ویژگی بیان شده به صورت رشته به نوع پیچیده واقعی ویژگی استفاده می شود. CustomEditorConfigurer، یک پس پردازشگر کارخانه bean، ممکن است برای افزودن راحت پشتیبانی از نمونه های PropertyEditor اضافی به ApplicationContext استفاده شود.

یک کلاس کاربری ExoticType و یک کلاس دیگر DependsOnExoticType را در نظر بگیرید که به مجموعه ExoticType به عنوان ویژگی نیاز دارد:

**package** example;

**public** **class** **ExoticType** {

**private** String name;

**public** ExoticType(String name) {

this.name = name;

}

}

**public** **class** **DependsOnExoticType** {

**private** ExoticType type;

**public** **void** setType(ExoticType type) {

this.type = type;

}

}

هنگامی که چیزها به درستی تنظیم می شوند، می خواهیم بتوانیم ویژگی type را به عنوان یک رشته اختصاص دهیم که یک PropertyEditor در پشت صحنه آن را به یک نمونه ExoticType واقعی تبدیل می کند:

<bean id="sample" class="example.DependsOnExoticType">

<property name="type" value="aNameForExoticType"/>

</bean>

اجرای PropertyEditor می تواند شبیه به این باشد:

*// converts string representation to ExoticType object*

**package** example;

**public** **class** **ExoticTypeEditor** **extends** PropertyEditorSupport {

**public** **void** setAsText(String text) {

setValue(**new** ExoticType(text.toUpperCase()));

}

}

در نهایت، ما از CustomEditorConfigurer برای ثبت PropertyEditor جدید با ApplicationContext استفاده می کنیم، که سپس می تواند در صورت نیاز از آن استفاده کند:

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="customEditors">

<map>

<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>

</map>

</property>

</bean>

###### Using PropertyEditorRegistrars

مکانیزم دیگر برای ثبت ویرایشگرهای اموال با کانتینر Spring، ایجاد و استفاده از PropertyEditorRegistrar است. این رابط به ویژه زمانی مفید است که شما نیاز به استفاده از یک مجموعه ویرایشگرهای دارایی در چندین موقعیت مختلف دارید: یک ثبت کننده مربوطه بنویسید و در هر مورد مجدداً از آن استفاده کنید. PropertyEditorRegistrars در ارتباط با رابطی به نام PropertyEditorRegistry کار می کند، رابطی که توسط Spring BeanWrapper (و DataBinder) پیاده سازی می شود. PropertyEditorRegistrars زمانی که همراه با CustomEditorConfigurer استفاده می‌شود (که در اینجا معرفی می‌شود) بسیار راحت هستند، که خصوصیتی به نام setPropertyEditorRegistrars(..) را نشان می‌دهد: PropertyEditorRegistrars که به این روش به CustomEditorConfigurer اضافه می‌شود، می‌تواند به راحتی با DataBinder و Spring MVC Controller به اشتراک گذاشته شود. علاوه بر این، از نیاز به همگام‌سازی در ویرایشگرهای سفارشی جلوگیری می‌کند: انتظار می‌رود که یک PropertyEditorRegistrar برای هر تلاش ایجاد bean نمونه‌های PropertyEditor جدیدی ایجاد کند.

استفاده از PropertyEditorRegistrar شاید بهتر باشد با یک مثال نشان داده شود. ابتدا باید PropertyEditorRegistrar خود را ایجاد کنید:

**package** com.foo.editors.spring;

**public** **final** **class** **CustomPropertyEditorRegistrar** **implements** PropertyEditorRegistrar {

**public** **void** registerCustomEditors(PropertyEditorRegistry registry) {

*// it is expected that new PropertyEditor instances are created*

registry.registerCustomEditor(ExoticType.class, **new** ExoticTypeEditor());

*// you could register as many custom property editors as are required here...*

}

}

همچنین به org.springframework.beans.support.ResourceEditorRegistrar برای مثال اجرای PropertyEditorRegistrar مراجعه کنید. توجه کنید که چگونه در پیاده سازی متد registerCustomEditors(..) نمونه های جدیدی از هر ویرایشگر ویژگی ایجاد می کند.

سپس یک CustomEditorConfigurer را پیکربندی می کنیم و نمونه ای از CustomPropertyEditorRegistrar خود را به آن تزریق می کنیم:

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="propertyEditorRegistrars">

<list>

<ref bean="customPropertyEditorRegistrar"/>

</list>

</property>

</bean>

<bean id="customPropertyEditorRegistrar"

class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>

در نهایت، و با کمی فاصله گرفتن از تمرکز این فصل، برای کسانی از شما که از چارچوب وب Spring’s MVC استفاده می‌کنید، استفاده از PropertyEditorRegistrars در ارتباط با کنترل‌کننده‌های اتصال داده (مانند SimpleFormController) می‌تواند بسیار راحت باشد. در زیر مثالی از استفاده از PropertyEditorRegistrar در پیاده سازی متد initBinder(..) بیابید:

**public** **final** **class** **RegisterUserController** **extends** SimpleFormController {

**private** **final** PropertyEditorRegistrar customPropertyEditorRegistrar;

**public** RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {

this.customPropertyEditorRegistrar = propertyEditorRegistrar;

}

**protected** **void** initBinder(HttpServletRequest request,

ServletRequestDataBinder binder) **throws** Exception {

**this.customPropertyEditorRegistrar.registerCustomEditors(binder);**

}

*// other methods to do with registering a User*

}

این سبک از ثبت نام PropertyEditor می تواند منجر به کد مختصر شود (پیاده سازی initBinder(..) فقط یک خط طول دارد!)، و اجازه می دهد تا کد ثبت نام PropertyEditor معمولی در یک کلاس کپسوله شود و سپس در بین کنترلرهای مورد نیاز به اشتراک گذاشته شود.

### 3.5. Spring Type Conversion

Spring 3 یک بسته core.convert را معرفی می کند که یک سیستم تبدیل نوع کلی را ارائه می دهد. این سیستم یک SPI را برای پیاده سازی منطق تبدیل نوع، و همچنین یک API برای اجرای تبدیل نوع در زمان اجرا تعریف می کند. در داخل یک کانتینر Spring، این سیستم می تواند به عنوان جایگزینی برای PropertyEditors برای تبدیل رشته های ارزش ویژگی bean خارجی به انواع ویژگی های مورد نیاز استفاده شود. API عمومی همچنین ممکن است در هر جایی از برنامه شما که نیاز به تبدیل نوع است استفاده شود.

#### 3.5.1. Converter SPI

SPI برای اجرای منطق تبدیل نوع ساده و به شدت تایپ شده است:

**package** org.springframework.core.convert.converter;

**public** **interface** **Converter**<S, T> {

T convert(S source);

}

برای ایجاد مبدل خود، به سادگی رابط بالا را پیاده سازی کنید. S را به عنوان نوعی که از آن تبدیل می کنید و T را به عنوان نوعی که به آن تبدیل می کنید، پارامتر کنید. اگر مجموعه یا آرایه ای از S باید به آرایه یا مجموعه ای از T تبدیل شود، می توان چنین مبدلی را نیز به صورت شفاف اعمال کرد، مشروط بر اینکه مبدل آرایه/مجموعه تفویض کننده نیز ثبت شده باشد (که DefaultConversionService به طور پیش فرض انجام می دهد).

برای هر فراخوانی برای تبدیل (S)، آرگومان منبع تضمین می شود که NOT null باشد. مبدل شما ممکن است در صورت عدم موفقیت تبدیل، هر گونه استثنای بررسی نشده را ایجاد کند. به طور خاص، یک IllegalArgumentException باید برای گزارش یک مقدار منبع نامعتبر پرتاب شود. مراقب باشید که اجرای مبدل شما از نظر موضوع ایمن باشد.

چندین پیاده سازی مبدل در بسته core.convert.support به عنوان یک راحتی ارائه شده است. اینها شامل مبدل‌هایی از رشته‌ها به اعداد و سایر انواع رایج است. StringToInteger را به عنوان مثالی برای اجرای Converter معمولی در نظر بگیرید:

**package** org.springframework.core.convert.support;

**final** **class** **StringToInteger** **implements** Converter<String, Integer> {

**public** Integer convert(String source) {

**return** Integer.valueOf(source);

}

}

#### 3.5.2. ConverterFactory

هنگامی که نیاز دارید منطق تبدیل را برای کل سلسله مراتب کلاس متمرکز کنید (به عنوان مثال، هنگام تبدیل از String به اشیاء Enum)، می توانید ConverterFactory را پیاده سازی کنید، همانطور که مثال زیر نشان می دهد:

S را به گونه ای که از آن تبدیل می کنید و R را به عنوان نوع پایه تعیین کنید که محدوده کلاس هایی را که می توانید به آن تبدیل کنید، پارامتر کنید. سپس getConverter (Class<T>) را پیاده سازی کنید، که در آن T یک زیر کلاس از R است.

به عنوان مثال StringToEnumConverterFactory را در نظر بگیرید:

**package** org.springframework.core.convert.support;

**final** **class** **StringToEnumConverterFactory** **implements** ConverterFactory<String, Enum> {

**public** <T **extends** Enum> Converter<String, T> getConverter(Class<T> targetType) {

**return** **new** StringToEnumConverter(targetType);

}

**private** **final** **class** **StringToEnumConverter**<T **extends** Enum> **implements** Converter<String, T> {

**private** Class<T> enumType;

**public** StringToEnumConverter(Class<T> enumType) {

this.enumType = enumType;

}

**public** T convert(String source) {

**return** (T) Enum.valueOf(this.enumType, source.trim());

}

}

}

#### 3.5.3. GenericConverter

هنگامی که به یک پیاده سازی مبدل پیچیده نیاز دارید، از رابط GenericConverter استفاده کنید. GenericConverter با امضای انعطاف‌پذیرتر اما با شدت کمتری نسبت به Converter، از تبدیل بین چندین نوع منبع و هدف پشتیبانی می‌کند. علاوه بر این، GenericConverter زمینه منبع و هدف زمینه را در دسترس قرار می دهد که می توانید هنگام پیاده سازی منطق تبدیل خود از آنها استفاده کنید. چنین زمینه ای اجازه می دهد تا یک تبدیل نوع توسط یک حاشیه نویسی فیلد یا اطلاعات عمومی اعلام شده در یک امضای فیلد انجام شود. لیست زیر تعریف رابط GenericConverter را نشان می دهد:

**package** org.springframework.core.convert.converter;

**public** **interface** **GenericConverter** {

**public** Set<ConvertiblePair> getConvertibleTypes();

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

برای پیاده‌سازی GenericConverter، از getConvertibleTypes() بخواهید جفت‌های نوع منبع → هدف پشتیبانی شده را برگرداند. سپس تبدیل (Object، TypeDescriptor، TypeDescriptor) را پیاده سازی کنید تا منطق تبدیل شما را در بر بگیرد. منبع TypeDescriptor دسترسی به فیلد منبعی را فراهم می کند که مقدار تبدیل شده را در خود نگه می دارد. هدف TypeDescriptor دسترسی به فیلد هدف را فراهم می کند که در آن مقدار تبدیل شده قرار است تنظیم شود.

یک مثال خوب از GenericConverter مبدلی است که بین یک آرایه جاوا و یک مجموعه تبدیل می شود. چنین ArrayToCollectionConverter، فیلدی را که نوع مجموعه هدف را اعلام می‌کند، بررسی می‌کند تا نوع عنصر مجموعه را حل کند. این اجازه می دهد تا قبل از تنظیم مجموعه در فیلد هدف، هر عنصر در آرایه منبع به نوع عنصر مجموعه تبدیل شود.

از آنجایی که GenericConverter یک رابط SPI پیچیده تر است، فقط زمانی که به آن نیاز دارید از آن استفاده کنید. Favor Converter یا ConverterFactory برای نیازهای اصلی تبدیل نوع.

##### ConditionalGenericConverter

گاهی اوقات شما فقط می خواهید یک Converter اجرا شود که یک شرط خاص درست باشد. به عنوان مثال، ممکن است تنها زمانی بخواهید یک مبدل را اجرا کنید که یک حاشیه نویسی خاص در فیلد مورد نظر وجود داشته باشد. یا ممکن است تنها زمانی بخواهید یک Converter را اجرا کنید که یک متد خاص، مانند یک متد static valueOf، روی کلاس هدف تعریف شده باشد. ConditionalGenericConverter ترکیبی از رابط های GenericConverter و ConditionalConverter است که به شما امکان می دهد چنین معیارهای تطبیق سفارشی را تعریف کنید:

**public** **interface** **ConditionalConverter** {

**boolean** matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}

**public** **interface** **ConditionalGenericConverter** **extends** GenericConverter, ConditionalConverter {

}

یک مثال خوب از ConditionalGenericConverter یک EntityConverter است که بین یک شناسه موجودیت دائمی و یک مرجع موجودیت تبدیل می شود. چنین EntityConverter تنها زمانی ممکن است مطابقت داشته باشد که نوع موجودیت هدف یک روش یاب ثابت را اعلام کند، به عنوان مثال. FindAccount (طولانی). شما می توانید چنین بررسی روش یاب را در اجرای مطابقت ها انجام دهید (TypeDescriptor، TypeDescriptor).

#### 3.5.4. ConversionService API

ConversionService یک API یکپارچه برای اجرای منطق تبدیل نوع در زمان اجرا تعریف می کند. مبدل ها اغلب در پشت این رابط نما اجرا می شوند:

**package** org.springframework.core.convert;

**public** **interface** **ConversionService** {

**boolean** canConvert(Class<?> sourceType, Class<?> targetType);

<T> T convert(Object source, Class<T> targetType);

**boolean** canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

اکثر پیاده سازی های ConversionService ConverterRegistry را نیز پیاده سازی می کنند که یک SPI برای ثبت مبدل ها ارائه می دهد. در داخل، یک پیاده سازی ConversionService به مبدل های ثبت شده خود واگذار می شود تا منطق تبدیل نوع را انجام دهند.

یک پیاده سازی ConversionService قوی در بسته core.convert.support ارائه شده است. GenericConversionService یک پیاده سازی همه منظوره است که برای استفاده در اکثر محیط ها مناسب است. ConversionServiceFactory یک کارخانه مناسب برای ایجاد تنظیمات ConversionService معمولی فراهم می کند.

#### 3.5.5. Configuring a ConversionService

ConversionService یک شیء بدون حالت است که برای نمونه سازی در هنگام راه اندازی برنامه طراحی شده است، سپس بین چندین رشته به اشتراک گذاشته می شود. در یک برنامه Spring، شما معمولاً یک نمونه ConversionService را در هر ظرف Spring (یا ApplicationContext) پیکربندی می کنید. آن ConversionService تا Spring دریافت می‌شود و سپس هر زمان که نیاز باشد یک تبدیل نوع توسط فریم‌ورک انجام شود، استفاده می‌شود. همچنین می توانید این ConversionService را به هر یک از دانه های خود تزریق کنید و مستقیماً آن را فراخوانی کنید.

اگر هیچ ConversionService با Spring ثبت نشده باشد، از سیستم اصلی مبتنی بر PropertyEditor استفاده می شود.

برای ثبت یک ConversionService پیش فرض با Spring، تعریف bean زیر را با id conversionService اضافه کنید:

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean"/>

یک ConversionService پیش‌فرض می‌تواند بین رشته‌ها، اعداد، فهرست‌ها، مجموعه‌ها، نقشه‌ها و دیگر انواع رایج تبدیل کند. برای تکمیل یا نادیده گرفتن مبدل های پیش فرض با مبدل(های) سفارشی خود، ویژگی مبدل ها را تنظیم کنید. مقادیر ویژگی ممکن است یکی از رابط های Converter، ConverterFactory یا GenericConverter را پیاده سازی کنند.

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="example.MyCustomConverter"/>

</set>

</property>

</bean>

همچنین استفاده از ConversionService در برنامه Spring MVC معمول است. تبدیل و قالب‌بندی را در فصل Spring MVC ببینید.

در شرایط خاصی ممکن است بخواهید قالب بندی را در طول تبدیل اعمال کنید. برای جزئیات بیشتر در مورد استفاده از FormattingConversionServiceFactoryBean به FormatterRegistry SPI مراجعه کنید.

3.5.6. Using a ConversionService programmatically

برای کار با یک نمونه ConversionService به صورت برنامه نویسی، کافی است مانند هر Bean دیگری، یک مرجع به آن تزریق کنید:

@Service

**public** **class** **MyService** {

@Autowired

**public** MyService(ConversionService conversionService) {

this.conversionService = conversionService;

}

**public** **void** doIt() {

this.conversionService.convert(...)

}

}

برای بیشتر موارد استفاده، روش تبدیل که targetType را مشخص می‌کند می‌تواند استفاده شود، اما با انواع پیچیده‌تر مانند مجموعه‌ای از یک عنصر پارامتری کار نمی‌کند. برای مثال، اگر می‌خواهید فهرستی از عدد صحیح را به صورت برنامه‌نویسی به فهرست رشته تبدیل کنید، باید یک تعریف رسمی از نوع منبع و هدف ارائه دهید.

خوشبختانه، TypeDescriptor گزینه های مختلفی را برای ساده کردن آن ارائه می دهد:

DefaultConversionService cs = **new** DefaultConversionService();

List<Integer> input = ....

cs.convert(input,

TypeDescriptor.forObject(input), *// List<Integer> type descriptor*

TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));

توجه داشته باشید که DefaultConversionService مبدل هایی را که برای اکثر محیط ها مناسب هستند به صورت خودکار ثبت می کند. این شامل مبدل های مجموعه، مبدل های اسکالر، و همچنین مبدل های اصلی Object به String می شود. همان مبدل‌ها را می‌توان با هر ConverterRegistry با استفاده از روش addDefaultConverters استاتیک در کلاس DefaultConversionService ثبت کرد.

مبدل‌های انواع مقادیر برای آرایه‌ها و مجموعه‌ها مجدداً استفاده خواهند شد، بنابراین نیازی به ایجاد مبدل خاصی برای تبدیل از مجموعه S به مجموعه T نیست، با این فرض که مدیریت مجموعه استاندارد مناسب است.

3.6. Spring Field Formatting

همانطور که در بخش قبل بحث شد، core.convert یک سیستم تبدیل نوع همه منظوره است. این یک ConversionService API یکپارچه و همچنین یک Converter SPI با تایپ قوی برای اجرای منطق تبدیل از یک نوع به نوع دیگر ارائه می دهد. یک کانتینر Spring از این سیستم برای اتصال مقادیر ویژگی bean استفاده می کند. علاوه بر این، هر دو زبان عبارت Spring (SpEL) و DataBinder از این سیستم برای اتصال مقادیر فیلد استفاده می کنند. به عنوان مثال، زمانی که SpEL برای تکمیل یک تلاش express.setValue (Object bean، Object value) باید یک Short به Long را وادار کند، سیستم core.convert این اجبار را انجام می دهد.

اکنون نیازهای تبدیل نوع یک محیط کلاینت معمولی مانند برنامه وب یا دسکتاپ را در نظر بگیرید. در چنین محیط‌هایی، شما معمولاً از String برای پشتیبانی از فرآیند postback مشتری، و همچنین برای پشتیبانی از فرآیند رندر view به String تبدیل می‌کنید. علاوه بر این، شما اغلب نیاز دارید که مقادیر String را بومی سازی کنید. کلی تر core.convert Converter SPI به طور مستقیم به چنین الزامات قالب بندی نمی پردازد. برای پرداختن مستقیم به آنها، Spring 3 یک Formatter SPI مناسب را معرفی می کند که جایگزینی ساده و قوی برای PropertyEditors برای محیط های کلاینت ارائه می دهد.

به طور کلی، زمانی که نیاز به پیاده سازی منطق تبدیل نوع همه منظوره دارید، می توانید از Converter SPI استفاده کنید—مثلاً برای تبدیل بین یک java.util.Date و یک Long. هنگامی که در یک محیط کلاینت (مانند یک برنامه وب) کار می کنید و نیاز به تجزیه و چاپ مقادیر فیلد محلی دارید، می توانید از Formatter SPI استفاده کنید. ConversionService یک API تبدیل نوع یکپارچه برای هر دو SPI فراهم می کند.

#### 3.6.1. Formatter SPI

Formatter SPI برای پیاده سازی منطق قالب بندی فیلد ساده و به شدت تایپ شده است:

**package** org.springframework.format;

**public** **interface** **Formatter**<T> **extends** Printer<T>, Parser<T> {

}

جایی که Formatter از رابط های بلوک ساختمانی Printer و Parser گسترش می یابد:

**public** **interface** **Printer**<T> {

String print(T fieldValue, Locale locale);

}

**import** java.text.ParseException;

**public** **interface** **Parser**<T> {

T parse(String clientValue, Locale locale) **throws** ParseException;

}

برای ایجاد Formatter خود، رابط Formatter را که قبلا نشان داده شده بود، پیاده سازی کنید. T را پارامتر کنید تا نوع شی ای باشد که می خواهید قالب بندی کنید——مثلاً java.util.Date. عملیات print() را برای چاپ نمونه ای از T برای نمایش در محلی مشتری اجرا کنید. عملیات parse() را برای تجزیه نمونه ای از T از نمایش فرمت شده برگردانده شده از محل کلاینت اجرا کنید. اگر تلاش تجزیه ناموفق باشد، Formatter شما باید یک ParseException یا یک IllegalArgumentException پرتاب کند. مراقب باشید که پیاده سازی Formatter شما از نظر موضوع ایمن باشد.

بسته‌های فرعی قالب چندین پیاده‌سازی Formatter را به عنوان یک راحتی فراهم می‌کنند. بسته اعداد NumberStyleFormatter، CurrencyStyleFormatter و PercentStyleFormatter را برای قالب بندی اشیاء Number که از java.text.NumberFormat استفاده می کنند، ارائه می دهد. بسته datetime یک DateFormatter برای قالب بندی اشیاء java.util.Date با یک java.text.DateFormat ارائه می دهد. بسته datetime.joda پشتیبانی جامع قالب‌بندی تاریخ بر اساس کتابخانه Joda-Time را فراهم می‌کند.

DateFormatter زیر نمونه ای از پیاده سازی Formatter است:

**package** org.springframework.format.datetime;

**public** **final** **class** **DateFormatter** **implements** Formatter<Date> {

**private** String pattern;

**public** DateFormatter(String pattern) {

this.pattern = pattern;

}

**public** String print(Date date, Locale locale) {

**if** (date == null) {

**return** "";

}

**return** getDateFormat(locale).format(date);

}

**public** Date parse(String formatted, Locale locale) **throws** ParseException {

**if** (formatted.length() == 0) {

**return** null;

}

**return** getDateFormat(locale).parse(formatted);

}

**protected** DateFormat getDateFormat(Locale locale) {

DateFormat dateFormat = **new** SimpleDateFormat(this.pattern, locale);

dateFormat.setLenient(false);

**return** dateFormat;

}

}

#### 3.6.2. Annotation-driven Formatting

همانطور که خواهید دید، قالب بندی فیلد را می توان بر اساس نوع فیلد یا حاشیه نویسی پیکربندی کرد. برای اتصال Annotation به فرمت‌کننده، AnnotationFormatterFactory را پیاده‌سازی کنید:

**package** org.springframework.format;

**public** **interface** **AnnotationFormatterFactory**<A **extends** Annotation> {

Set<Class<?>> getFieldTypes();

Printer<?> getPrinter(A annotation, Class<?> fieldType);

Parser<?> getParser(A annotation, Class<?> fieldType);

}

A را پارامتر کنید تا annotationنوعی فیلد باشد. از getFieldTypes () انواع فیلدهایی را که ممکن است حاشیه نویسی در آنها استفاده شود، بازگردانید. از getPrinter() یک چاپگر برای چاپ مقدار یک فیلد حاشیه نویسی بخواهید. از getParser() یک Parser برای تجزیه یک clientValue برای یک فیلد حاشیه نویسی بخواهید.

مثال اجرای AnnotationFormatterFactory زیر، Annotation@NumberFormat را به قالب‌کننده متصل می‌کند. این حاشیه‌نویسی به شما امکان می‌دهد یک سبک یا الگوی عددی را مشخص کنید:

**public** **final** **class** **NumberFormatAnnotationFormatterFactory**

**implements** AnnotationFormatterFactory<NumberFormat> {

**public** Set<Class<?>> getFieldTypes() {

**return** **new** HashSet<Class<?>>(asList(**new** Class<?>**[]** {

Short.class, Integer.class, Long.class, Float.class,

Double.class, BigDecimal.class, BigInteger.class }));

}

**public** Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {

**return** configureFormatterFrom(annotation, fieldType);

}

**public** Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {

**return** configureFormatterFrom(annotation, fieldType);

}

**private** Formatter<Number> configureFormatterFrom(NumberFormat annotation, Class<?> fieldType) {

**if** (!annotation.pattern().isEmpty()) {

**return** **new** NumberStyleFormatter(annotation.pattern());

} **else** {

Style style = annotation.style();

**if** (style == Style.PERCENT) {

**return** **new** PercentStyleFormatter();

} **else** **if** (style == Style.CURRENCY) {

**return** **new** CurrencyStyleFormatter();

} **else** {

**return** **new** NumberStyleFormatter();

}

}

}

}

برای راه اندازی قالب بندی، به سادگی فیلدها را با @NumberFormat حاشیه نویسی کنید:

**public** **class** **MyModel** {

@NumberFormat(style=Style.CURRENCY)

**private** BigDecimal decimal;

}

#### 3.6.3. FormatterRegistry SPI

FormatterRegistry یک SPI برای ثبت فرمت‌کننده‌ها و مبدل‌ها است. FormattingConversionService پیاده سازی FormatterRegistry است که برای اکثر محیط ها مناسب است. این پیاده سازی ممکن است به صورت برنامه نویسی یا به صورت اعلامی به صورت Spring bean با استفاده از FormattingConversionServiceFactoryBean پیکربندی شود. از آنجا که این پیاده‌سازی ConversionService را نیز پیاده‌سازی می‌کند، می‌توان آن را مستقیماً برای استفاده با Spring’s DataBinder و Spring Expression Language (SpEL) پیکربندی کرد.

FormatterRegistry SPI را در زیر مرور کنید:

**package** org.springframework.format;

**public** **interface** **FormatterRegistry** **extends** ConverterRegistry {

**void** addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

**void** addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

**void** addFormatterForFieldType(Formatter<?> formatter);

**void** addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}

همانطور که در بالا نشان داده شد، Formatters را می توان توسط fieldType یا annotation ثبت کرد.

FormatterRegistry SPI به شما این امکان را می دهد که به جای کپی کردن چنین تنظیماتی در کنترلرهای خود، قوانین قالب بندی را به صورت مرکزی پیکربندی کنید. به عنوان مثال، ممکن است بخواهید اعمال کنید که تمام فیلدهای تاریخ به روش خاصی قالب بندی شوند یا فیلدهایی با حاشیه نویسی خاص به روش خاصی قالب بندی شوند. با یک FormatterRegistry مشترک، شما این قوانین را یک بار تعریف می کنید و هر زمان که نیاز به قالب بندی باشد، اعمال می شوند.

#### 3.6.4. FormatterRegistrar SPI

FormatterRegistrar یک SPI برای ثبت فرمت‌کننده‌ها و مبدل‌ها از طریق FormatterRegistry است:

**package** org.springframework.format;

**public** **interface** **FormatterRegistrar** {

**void** registerFormatters(FormatterRegistry registry);

}

یک FormatterRegistrar هنگام ثبت چندین مبدل و فرمت‌کننده مرتبط برای یک دسته قالب‌بندی خاص، مانند قالب‌بندی تاریخ، مفید است. همچنین در مواردی که ثبت اظهاری کافی نیست، می تواند مفید باشد. به عنوان مثال زمانی که یک قالب‌کننده باید تحت یک نوع فیلد خاص متفاوت از <T> خودش ایندکس شود یا هنگام ثبت یک جفت چاپگر/تجزیه‌کننده. بخش بعدی اطلاعات بیشتری در مورد ثبت مبدل و فرمت کننده ارائه می دهد.

#### 3.6.5. Configuring Formatting in Spring MVC

See [Conversion and Formatting](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web.html#mvc-config-conversion) in the Spring MVC chapter.

### 3.7. Configuring a global date & time format

به طور پیش‌فرض، فیلدهای تاریخ و زمان که با @DateTimeFormat حاشیه‌نویسی نشده‌اند، با استفاده از سبک DateFormat.SHORT از رشته‌ها تبدیل می‌شوند. اگر ترجیح می دهید، می توانید این مورد را با تعریف قالب جهانی خود تغییر دهید.

باید مطمئن شوید که Spring فرمت‌کننده‌های پیش‌فرض را ثبت نمی‌کند و در عوض باید همه فرمت‌کننده‌ها را به صورت دستی ثبت کنید. بسته به اینکه از کتابخانه Joda-Time استفاده می کنید از کلاس org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar یا org.springframework.format.datetime.DateFormatterRegistrar استفاده کنید.

به عنوان مثال، پیکربندی جاوا زیر فرمت جهانی «yyyyMMdd» را ثبت می‌کند. این مثال به کتابخانه Joda-Time بستگی ندارد:

@Configuration

**public** **class** **AppConfig** {

@Bean

**public** FormattingConversionService conversionService() {

*// Use the DefaultFormattingConversionService but do not register defaults*

DefaultFormattingConversionService conversionService = **new** DefaultFormattingConversionService(false);

*// Ensure @NumberFormat is still supported*

conversionService.addFormatterForFieldAnnotation(**new** NumberFormatAnnotationFormatterFactory());

*// Register date conversion with a specific global format*

DateFormatterRegistrar registrar = **new** DateFormatterRegistrar();

registrar.setFormatter(**new** DateFormatter("yyyyMMdd"));

registrar.registerFormatters(conversionService);

**return** conversionService;

}

}

اگر پیکربندی مبتنی بر XML را ترجیح می دهید، می توانید از FormattingConversionServiceFactoryBean استفاده کنید. در اینجا همان مثال است، این بار با استفاده از Joda Time:

<?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

https://www.springframework.org/schema/beans/spring-beans.xsd>

<bean id="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="registerDefaultFormatters" value="false" />

<property name="formatters">

<set>

<bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">

<property name="dateFormatter">

<bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">

<property name="pattern" value="yyyyMMdd"/>

</bean>

</property>

</bean>

</set>

</property>

</bean>

</beans>

Joda-Time انواع مجزای مجزا برای نمایش مقادیر تاریخ، زمان و تاریخ-زمان ارائه می دهد. ویژگی‌های dateFormatter، timeFormatter و dateTimeFormatter JodaTimeFormatterRegistrar باید برای پیکربندی قالب‌های مختلف برای هر نوع استفاده شود. DateTimeFormatterFactoryBean راه مناسبی برای ایجاد فرمت‌کننده‌ها فراهم می‌کند.

اگر از Spring MVC استفاده می کنید، به یاد داشته باشید که سرویس تبدیل مورد استفاده را به صراحت پیکربندی کنید. برای @Configuration مبتنی بر جاوا، این به معنای گسترش کلاس WebMvcConfigurationSupport و لغو روش mvcConversionService() است. برای XML باید از ویژگی «تبدیل-سرویس» عنصر mvc:annotation-driven استفاده کنید. برای جزئیات بیشتر به تبدیل و قالب بندی مراجعه کنید.

### 3.8. Spring Validation

Spring 3 چندین پیشرفت را در پشتیبانی اعتبار سنجی خود معرفی می کند. ابتدا، JSR-303 Bean Validation API اکنون به طور کامل پشتیبانی می شود. دوم، زمانی که Spring's DataBinder به صورت برنامه‌ریزی استفاده می‌شود، اکنون می‌تواند اشیا را اعتبارسنجی کند و همچنین به آنها متصل شود. سوم، Spring MVC در حال حاضر از تأیید اعتبار اعلامی ورودی های @Controller پشتیبانی می کند.

#### 3.8.1. Overview of the JSR-303 Bean Validation API

JSR-303 اعلان محدودیت اعتبارسنجی و ابرداده را برای پلتفرم جاوا استاندارد می کند. با استفاده از این API، ویژگی های مدل دامنه را با محدودیت های اعتبار سنجی اعلامی حاشیه نویسی می کنید و زمان اجرا آنها را اجرا می کند. تعدادی محدودیت داخلی وجود دارد که می توانید از آنها استفاده کنید. شما همچنین می توانید محدودیت های سفارشی خود را تعریف کنید.

برای نشان دادن، یک مدل PersonForm ساده با دو ویژگی را در نظر بگیرید:

**public** **class** **PersonForm** {

**private** String name;

**private** **int** age;

}

JSR-303 به شما اجازه می دهد تا محدودیت های اعتبار سنجی اعلامی را در برابر چنین ویژگی هایی تعریف کنید:

**public** **class** **PersonForm** {

@NotNull

@Size(max=64)

**private** String name;

@Min(0)

**private** **int** age;

}

هنگامی که نمونه ای از این کلاس توسط اعتبارسنجی JSR-303 تأیید می شود، این محدودیت ها اعمال می شوند.

برای اطلاعات کلی در مورد JSR-303/JSR-349، به وب سایت Bean Validation مراجعه کنید. برای اطلاعات در مورد قابلیت های خاص پیاده سازی مرجع پیش فرض، به مستندات Hibernate Validator مراجعه کنید. برای یادگیری نحوه تنظیم ارائه‌دهنده اعتبار سنجی لوبیا به‌عنوان Spring bean، به خواندن ادامه دهید.

#### 3.8.2. Configuring a Bean Validation Provider

Spring از Bean Validation API پشتیبانی کامل می کند. این شامل پشتیبانی مناسب برای راه‌اندازی یک ارائه‌دهنده اعتبارسنجی Bean JSR-303/JSR-349 به‌عنوان Spring bean است. این اجازه می دهد تا یک javax.validation.ValidatorFactory یا javax.validation.Validator در هر جایی که نیاز به اعتبارسنجی در برنامه شما باشد تزریق شود.

از LocalValidatorFactoryBean برای پیکربندی یک Validator پیش فرض به عنوان یک Spring Bean استفاده کنید:

<bean id="validator"

class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

پیکربندی اصلی بالا، اعتبار Bean Validation را با استفاده از مکانیزم پیش‌فرض بوت استرپ، راه‌اندازی می‌کند. انتظار می رود یک ارائه دهنده JSR-303/JSR-349، مانند Hibernate Validator، در مسیر کلاس حضور داشته باشد و به طور خودکار شناسایی شود.

##### Injecting a Validator

LocalValidatorFactoryBean هر دو javax.validation.ValidatorFactory و javax.validation.Validator و همچنین Spring's org.springframework.validation.Validator را پیاده سازی می کند. شما می توانید یک مرجع به هر یک از این رابط ها را به bean هایی که نیاز به احضار منطق اعتبار سنجی دارند، تزریق کنید.

اگر ترجیح می دهید مستقیماً با Bean Validation API کار کنید، یک مرجع به javax.validation.Validator تزریق کنید:

**import** javax.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

اگر Bean شما به Spring Validation API نیاز دارد، یک مرجع به org.springframework.validation.Validator تزریق کنید:

**import** org.springframework.validation.Validator;

@Service

**public** **class** **MyService** {

@Autowired

**private** Validator validator;

}

##### Configuring Custom Constraints

هر محدودیت اعتبار سنجی Bean از دو بخش تشکیل شده است. ابتدا، یک حاشیه نویسی @Constraint که محدودیت و ویژگی های قابل تنظیم آن را اعلام می کند. دوم، پیاده‌سازی رابط javax.validation.ConstraintValidator که رفتار محدودیت را پیاده‌سازی می‌کند. برای مرتبط کردن یک اعلان با یک پیاده سازی، هر حاشیه نویسی @Constraint به یک کلاس پیاده سازی ConstraintValidator مربوطه ارجاع می دهد. در زمان اجرا، یک ConstraintValidatorFactory پیاده سازی ارجاع شده را زمانی که حاشیه نویسی محدودیت در مدل دامنه شما مواجه می شود، نمونه سازی می کند.

به طور پیش فرض، LocalValidatorFactoryBean یک SpringConstraintValidatorFactory را پیکربندی می کند که از Spring برای ایجاد نمونه های ConstraintValidator استفاده می کند. این به ConstraintValidators سفارشی شما اجازه می دهد تا مانند هر Spring bean دیگری از تزریق وابستگی بهره مند شوند.

در زیر مثالی از یک اعلان @Constraint سفارشی نشان داده شده است، به دنبال آن یک پیاده سازی ConstraintValidator مرتبط است که از Spring برای تزریق وابستگی استفاده می کند:

@Target({ElementType.METHOD, ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy=MyConstraintValidator.class)

**public** @interface MyConstraint {

}

**import** javax.validation.ConstraintValidator;

**public** **class** **MyConstraintValidator** **implements** ConstraintValidator {

@Autowired;

**private** Foo aDependency;

...

}

همانطور که می بینید، یک پیاده سازی ConstraintValidator ممکن است وابستگی های خود را مانند هر Spring bean دیگری @Autowired داشته باشد.

##### Spring-driven Method Validation

ویژگی اعتبارسنجی روش پشتیبانی شده توسط Bean Validation 1.1، و به عنوان یک پسوند سفارشی نیز توسط Hibernate Validator 4.3، می تواند از طریق یک تعریف Bean MethodValidationPostProcessor در یک زمینه Spring ادغام شود:

<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>

برای واجد شرایط بودن برای اعتبارسنجی روش مبتنی بر Spring، همه کلاس‌های هدف باید با حاشیه‌نویسی @Validated Spring مشروح شوند و به‌صورت اختیاری گروه‌های اعتبارسنجی را برای استفاده اعلام کنند. برای جزئیات راه اندازی با ارائه دهندگان Hibernate Validator و Bean Validation 1.1، javadocs MethodValidationPostProcessor را بررسی کنید.

##### **Additional Configuration Options**

پیکربندی پیش‌فرض LocalValidatorFactoryBean باید برای اکثر موارد کافی باشد. تعدادی گزینه پیکربندی برای ساختارهای مختلف اعتبارسنجی Bean، از درونیابی پیام تا وضوح پیمایش وجود دارد. برای اطلاعات بیشتر در مورد این گزینه ها به جاوادوکس LocalValidatorFactoryBean مراجعه کنید.

#### **3.8.3. Configuring a DataBinder**

یک DataBinder همچنین می تواند با چندین نمونه Validator از طریق dataBinder.addValidators و dataBinder.replaceValidators پیکربندی شود. این زمانی مفید است که Bean Validation با پیکربندی جهانی را با یک Spring Validator که به صورت محلی روی یک نمونه DataBinder پیکربندی شده است، ترکیب کنید. [validation-mvc-configuring] را ببینید.

#### **3.8.4. Spring MVC 3 Validation**

See [Validation](https://docs.spring.io/spring-framework/docs/5.0.x/spring-framework-reference/web.html#mvc-config-validation) in the Spring MVC chapter.

# **4. Spring Expression Language (SpEL)**

### **4.1. Introduction**

Spring Expression Language (به اختصار SpEL) یک زبان عبارت قدرتمند است که از پرس و جو و دستکاری یک نمودار شی در زمان اجرا پشتیبانی می کند. نحو زبان مشابه Unified EL است، اما ویژگی‌های اضافی را ارائه می‌دهد، از جمله مهم‌ترین آنها فراخوانی روش و قابلیت‌های اولیه قالب رشته.

در حالی که چندین زبان بیان جاوا دیگر در دسترس هستند - OGNL، MVEL، و JBoss EL، برای نام بردن چند نفر - زبان Spring Expression برای ارائه یک زبان بیانی واحد به خوبی پشتیبانی شده برای جامعه Spring ایجاد شد که می تواند در همه محصولات استفاده شود. نمونه کارها بهار ویژگی‌های زبان آن بر اساس الزامات پروژه‌های مجموعه Spring، از جمله الزامات ابزار برای پشتیبانی از تکمیل کد در مجموعه ابزار Spring مبتنی بر Eclipse، هدایت می‌شود. با این اوصاف، SpEL مبتنی بر یک API آگنوستیک فناوری است که در صورت لزوم، امکان ادغام سایر پیاده‌سازی‌های زبان بیان را فراهم می‌کند.

در حالی که SpEL به عنوان پایه ای برای ارزیابی بیان در پورتفولیوی Spring عمل می کند، مستقیماً به Spring وابسته نیست و می تواند به طور مستقل استفاده شود. بسیاری از مثال‌های این فصل برای اینکه خود شامل باشند، از SpEL به‌گونه‌ای استفاده می‌کنند که گویی یک زبان بیان مستقل است. این امر مستلزم ایجاد چند کلاس زیرساخت بوت استرپینگ مانند تجزیه کننده است. اکثر کاربران Spring نیازی به سر و کار داشتن با این زیرساخت ندارند و در عوض فقط رشته های عبارت نویسنده را برای ارزیابی می خواهند. یک مثال از این استفاده معمولی، ادغام SpEL در ایجاد XML یا تعاریف bean مبتنی بر حاشیه‌نویسی است، همانطور که در بخش پشتیبانی بیان برای تعریف تعاریف bean نشان داده شده است.

این فصل ویژگی های زبان عبارت، API و نحو زبان آن را پوشش می دهد. در چندین مکان از کلاس های Inventor و Inventor’s Society به عنوان اشیاء هدف برای ارزیابی بیان استفاده می شود. این اعلان‌های کلاس و داده‌های مورد استفاده برای تکمیل آنها در انتهای فصل فهرست شده‌اند.

زبان عبارت از عملکرد زیر پشتیبانی می کند:

* Literal expressions
* Boolean and relational operators
* Regular expressions
* Class expressions
* Accessing properties, arrays, lists, maps
* Method invocation
* Relational operators
* Assignment
* Calling constructors
* Bean references
* Array construction
* Inline lists
* Inline maps
* Ternary operator
* Variables
* User defined functions
* Collection projection
* Collection selection
* Templated expressions

### 4.2. Evaluation

این بخش استفاده ساده از رابط های SpEL و زبان بیان آن را معرفی می کند. مرجع کامل زبان را می توانید در بخش مرجع زبان پیدا کنید.

کد زیر SpEL API را برای ارزیابی عبارت رشته تحت اللفظی "Hello World" معرفی می کند.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'**");

String message = (String) exp.getValue();

مقدار متغیر پیام به سادگی "Hello World" است.

کلاس‌ها و رابط‌های SpEL که احتمالاً استفاده می‌کنید در بسته‌های org.springframework.expression و بسته‌های فرعی آن مانند spel.support قرار دارند.

رابط ExpressionParser مسئول تجزیه یک رشته عبارت است. در این مثال عبارت عبارت string یک رشته تحت اللفظی است که با تک نقل قول های اطراف نشان داده می شود. Interface Expression مسئول ارزیابی رشته عبارت تعریف شده قبلی است. هنگام فراخوانی parser.parseExpression و exp.getValue دو استثنا وجود دارد که می‌توان آنها را پرتاب کرد، ParseException و EvaluationException.

SpEL از طیف گسترده ای از ویژگی ها مانند روش های فراخوانی، دسترسی به ویژگی ها و فراخوانی سازنده ها پشتیبانی می کند.

به عنوان مثالی از فراخوانی متد، متد concat را در رشته لفظی می نامیم.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**'Hello World'.concat('!')**");

String message = (String) exp.getValue();

ارزش پیام اکنون «سلام جهان!» است.

به عنوان نمونه ای از فراخوانی یک ویژگی JavaBean، ویژگی String Bytes را می توان مطابق شکل زیر فراخوانی کرد.

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes()'*

Expression exp = parser.parseExpression("**'Hello World'.bytes**");

**byte[]** bytes = (**byte[]**) exp.getValue();

SpEL همچنین با استفاده از علامت نقطه استاندارد (مانند prop1.prop2.prop3) و همچنین تنظیمات مربوط به مقادیر ویژگی، از ویژگی های تو در تو پشتیبانی می کند. امکان دسترسی به فیلدهای عمومی نیز وجود دارد.

مثال زیر نحوه استفاده از نماد نقطه برای به دست آوردن طول یک حرف را نشان می دهد:

ExpressionParser parser = **new** SpelExpressionParser();

*// invokes 'getBytes().length'*

Expression exp = parser.parseExpression("**'Hello World'.bytes.length**");

**int** length = (Integer) exp.getValue();

سازنده String را می توان به جای استفاده از رشته Literal فراخوانی کرد.

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**new String('hello world').toUpperCase()**");

String message = exp.getValue(String.class);

به استفاده از روش عمومی public <T> T getValue(Class<T> ResultType مطلوب) توجه کنید. استفاده از این روش نیاز به ریختن مقدار عبارت به نوع نتیجه دلخواه را برطرف می کند. اگر مقدار را نتوان به نوع T فرستاد یا با استفاده از مبدل نوع ثبت شده تبدیل کرد، یک EvaluationException پرتاب می‌شود.

رایج‌ترین استفاده از SpEL، ارائه یک رشته عبارت است که در برابر یک نمونه شی خاص (به نام شی ریشه) ارزیابی می‌شود. مثال نشان می دهد که چگونه می توان ویژگی name را از نمونه ای از کلاس Inventor بازیابی کرد یا یک شرط بولی ایجاد کرد:

*// Create and set a calendar*

GregorianCalendar c = **new** GregorianCalendar();

c.set(1856, 7, 9);

*// The constructor arguments are name, birthday, and nationality.*

Inventor tesla = **new** Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = **new** SpelExpressionParser();

Expression exp = parser.parseExpression("**name**");

String name = (String) exp.getValue(tesla);

*// name == "Nikola Tesla"*

exp = parser.parseExpression("name == 'Nikola Tesla'");

**boolean** result = exp.getValue(tesla, Boolean.class);

*// result == true*

**4.2.1. EvaluationContext**

واسط EvaluationContext هنگام ارزیابی یک عبارت برای حل ویژگی ها، روش ها یا فیلدها و کمک به انجام تبدیل نوع استفاده می شود. دو پیاده سازی خارج از جعبه وجود دارد.

SimpleEvaluationContext - زیر مجموعه ای از ویژگی های ضروری زبان SpEL و گزینه های پیکربندی را برای دسته هایی از عبارات که به وسعت کامل نحو زبان SpEL نیاز ندارند و باید به طور معنی داری محدود شوند را نشان می دهد. مثال‌ها شامل عبارات اتصال داده‌ها، فیلترهای مبتنی بر ویژگی و موارد دیگر هستند، اما محدود به آن نیستند.

StandardEvaluationContext — مجموعه کاملی از ویژگی های زبان SpEL و گزینه های پیکربندی را در معرض دید قرار می دهد. می توانید از آن برای تعیین یک شی ریشه پیش فرض و پیکربندی هر استراتژی موجود مرتبط با ارزیابی استفاده کنید.

SimpleEvaluationContext برای پشتیبانی از تنها زیرمجموعه ای از نحو زبان SpEL طراحی شده است. ارجاعات نوع جاوا، سازنده ها و ارجاعات bean را مستثنی می کند. همچنین مستلزم آن است که به صراحت سطح پشتیبانی از ویژگی ها و روش ها در عبارات را انتخاب کنید. به‌طور پیش‌فرض، متد create() static factory فقط دسترسی خواندن به ویژگی‌ها را فعال می‌کند. همچنین می‌توانید سازنده‌ای برای پیکربندی سطح دقیق پشتیبانی مورد نیاز، با هدف قرار دادن یک یا ترکیبی از موارد زیر دریافت کنید:

Custom PropertyAccessor only (no reflection)

Data binding properties for read-only access

Data binding properties for read and write

##### **Type conversion**

SpEL به طور پیش فرض از سرویس تبدیل موجود در Spring core (org.springframework.core.convert.ConversionService) استفاده می کند. این سرویس تبدیل با بسیاری از مبدل‌های ساخته شده برای تبدیل‌های رایج ارائه می‌شود، اما همچنین کاملاً قابل توسعه است، بنابراین می‌توان تبدیل‌های سفارشی بین انواع را اضافه کرد. علاوه بر این، دارای قابلیت کلیدی است که ژنریک ها از آن آگاه هستند. این بدان معنی است که هنگام کار با انواع عمومی در عبارات، SpEL برای حفظ صحت نوع برای هر شیئی که با آن روبرو می شود، تبدیل ها را انجام می دهد.

این در عمل به چه معناست؟ فرض کنید انتساب با استفاده از setValue() برای تنظیم ویژگی List استفاده می شود. نوع خاصیت در واقع List<Boolean> است. SpEL تشخیص می دهد که عناصر لیست باید قبل از قرار گرفتن در آن به Boolean تبدیل شوند. یک مثال ساده:

**class** **Simple** {

**public** List<Boolean> booleanList = **new** ArrayList<Boolean>();

}

Simple simple = **new** Simple();

simple.booleanList.add(true);

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// "false" is passed in here as a String. SpEL and the conversion service*

*// will recognize that it needs to be a Boolean and convert it accordingly.*

parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

*// b will be false*

Boolean b = simple.booleanList.get(0);

#### **4.2.2. Parser configuration**

پیکربندی تجزیه کننده عبارت SpEL با استفاده از یک شی پیکربندی تجزیه کننده (org.springframework.expression.spel.SpelParserConfiguration) امکان پذیر است. شیء پیکربندی رفتار برخی از اجزای عبارت را کنترل می کند. به عنوان مثال، اگر نمایه سازی در یک آرایه یا مجموعه و عنصر موجود در شاخص مشخص شده تهی باشد، امکان ایجاد خودکار عنصر وجود دارد. این هنگام استفاده از عبارات ساخته شده از زنجیره ای از ارجاعات دارایی مفید است. اگر نمایه سازی در یک آرایه یا لیست و تعیین شاخصی که فراتر از اندازه فعلی آرایه یا لیست باشد، می توان به طور خودکار آرایه یا لیست را برای تطبیق با آن شاخص رشد داد.

**class** **Demo** {

**public** List<String> list;

}

*// Turn on:*

*// - auto null reference initialization*

*// - auto collection growing*

SpelParserConfiguration config = **new** SpelParserConfiguration(true,true);

ExpressionParser parser = **new** SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = **new** Demo();

Object o = expression.getValue(demo);

*// demo.list will now be a real collection of 4 entries*

*// Each entry is a new empty String*

همچنین می توان رفتار کامپایلر عبارت SpEL را پیکربندی کرد.

#### **4.2.3. SpEL compilation**

Spring Framework 4.1 شامل یک کامپایلر عبارت پایه است. عبارات معمولاً تفسیر می شوند که انعطاف پذیری پویا زیادی را در طول ارزیابی فراهم می کند اما عملکرد بهینه را ارائه نمی دهد. برای استفاده از بیان گاه به گاه، این خوب است، اما، هنگامی که توسط اجزای دیگر مانند Spring Integration استفاده می شود، عملکرد می تواند بسیار مهم باشد و نیازی واقعی به پویایی وجود ندارد.

کامپایلر SpEL برای رفع این نیاز در نظر گرفته شده است. در طول ارزیابی، کامپایلر یک کلاس جاوا تولید می کند که رفتار عبارت را در زمان اجرا مجسم می کند و از آن کلاس برای دستیابی به ارزیابی بیان بسیار سریعتر استفاده می کند. به دلیل عدم تایپ در اطراف عبارات، کامپایلر از اطلاعات جمع‌آوری‌شده در طی ارزیابی‌های تفسیری یک عبارت هنگام انجام کامپایل استفاده می‌کند. به عنوان مثال، نوع مرجع ویژگی را صرفاً از روی عبارت نمی‌داند، اما در اولین ارزیابی تفسیر شده، متوجه می‌شود که چیست. البته، کامپایل کردن بر اساس چنین اطلاعات مشتق شده می تواند بعداً در صورت تغییر انواع عناصر بیان مختلف در طول زمان، مشکل ایجاد کند. به همین دلیل، کامپایل برای عباراتی مناسب است که اطلاعات نوع آنها در ارزیابی های مکرر تغییر نمی کند.

برای یک عبارت اساسی مانند این:

someArray[0].someProperty.someOtherProperty <0.1

که شامل دسترسی به آرایه، برخی از ویژگی‌های دفاع زدایی و عملیات عددی است، افزایش عملکرد می‌تواند بسیار قابل توجه باشد. در یک نمونه میکرو بنچمارک اجرا شده با 50000 تکرار، ارزیابی تنها با استفاده از مفسر 75 میلی‌ثانیه و با استفاده از نسخه کامپایل‌شده عبارت فقط 3 میلی‌ثانیه طول می‌کشد.

##### **Compiler configuration**

کامپایلر به طور پیش فرض روشن نیست، اما دو راه برای روشن کردن آن وجود دارد. می توان آن را با استفاده از فرآیند پیکربندی تجزیه کننده که قبلاً توضیح داده شد یا از طریق یک ویژگی سیستمی هنگامی که استفاده از SpEL در داخل مؤلفه دیگری تعبیه شده است روشن شود. این بخش هر دوی این گزینه ها را مورد بحث قرار می دهد.

درک این نکته مهم است که چند حالت وجود دارد که کامپایلر می‌تواند در یک enum ثبت شود (org.springframework.expression.spel.SpelCompilerMode). حالت ها به شرح زیر است:

OFF - کامپایلر خاموش است. این پیش فرض است.

IMMEDIATE - در حالت فوری عبارات در اسرع وقت کامپایل می شوند. این معمولاً پس از اولین ارزیابی تفسیر شده است. اگر عبارت کامپایل شده با شکست مواجه شود (معمولاً به دلیل تغییر نوع، همانطور که در بالا توضیح داده شد) فراخوان دهنده ارزیابی عبارت یک استثنا دریافت خواهد کرد.

MIXED - در حالت ترکیبی، عبارات به مرور زمان بین حالت تفسیر شده و کامپایل شده جابجا می شوند. پس از تعدادی از اجراهای تفسیر شده، آنها به فرم کامپایل شده تغییر می کنند و اگر مشکلی در فرم کامپایل شده پیش بیاید (مانند تغییر نوع، همانطور که در بالا توضیح داده شد) سپس عبارت به طور خودکار دوباره به فرم تفسیر شده برمی گردد. مدتی بعد ممکن است یک فرم کامپایل شده دیگر ایجاد کند و به آن سوئیچ کند. اساساً استثنایی که کاربر در حالت IMMEDIATE دریافت می کند، در عوض به صورت داخلی رسیدگی می شود.

حالت IMMEDIATE وجود دارد زیرا حالت MIXED می تواند مشکلاتی را برای عباراتی ایجاد کند که دارای عوارض جانبی هستند. اگر یک عبارت کامپایل شده پس از موفقیت جزئی منفجر شود، ممکن است قبلاً کاری انجام داده باشد که بر وضعیت سیستم تأثیر گذاشته است. اگر این اتفاق افتاده باشد، ممکن است تماس گیرنده نخواهد که در حالت تفسیری بی صدا دوباره اجرا شود زیرا ممکن است بخشی از عبارت دو بار اجرا شود.

پس از انتخاب یک حالت، از SpelParserConfiguration برای پیکربندی تجزیه کننده استفاده کنید:

SpelParserConfiguration config = **new** SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,

this.getClass().getClassLoader());

SpelExpressionParser parser = **new** SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = **new** MyMessage();

Object payload = expr.getValue(message);

وقتی حالت کامپایلر را مشخص می‌کنید، می‌توانید یک کلاس‌لودر را نیز تعیین کنید (گذراندن null مجاز است). عبارات کامپایل شده در یک کلاس بارگذار فرزند که تحت هر کدام ارائه شده است، تعریف می شوند. مهم است که اطمینان حاصل شود که اگر یک کلاس لودر مشخص شده باشد، می تواند تمام انواع درگیر در فرآیند ارزیابی عبارت را ببیند. اگر کلاس بارگذاری را مشخص نکنید، از یک کلاس بارگذار پیش فرض استفاده می شود (معمولاً کلاس بارگذار زمینه برای رشته ای که در حین ارزیابی عبارت اجرا می شود).

راه دوم برای پیکربندی کامپایلر برای استفاده زمانی است که SpEL در داخل یک جزء دیگر تعبیه شده است و ممکن است پیکربندی آن از طریق یک شی پیکربندی امکان پذیر نباشد. در این موارد، امکان استفاده از ویژگی سیستم وجود دارد. می توانید ویژگی spring.expression.compiler.mode را روی یکی از مقادیر enum SpelCompilerMode (خاموش، فوری یا مخلوط) قرار دهید.

##### **Compiler limitations**

از Spring Framework 4.1، چارچوب کامپایل اولیه وجود دارد. با این حال، چارچوب هنوز از کامپایل کردن هر نوع عبارتی پشتیبانی نمی کند. تمرکز اولیه روی عبارات رایجی بوده است که احتمالاً در زمینه‌های حیاتی عملکرد مورد استفاده قرار می‌گیرند. در حال حاضر نمی توان انواع عبارت زیر را جمع آوری کرد:

expressions involving assignment

expressions relying on the conversion service

expressions using custom resolvers or accessors

expressions using selection or projection

در آینده انواع بیشتری از عبارت قابل کامپایل خواهند بود.

### **4.3. Expressions in bean definitions**

می‌توانید از عبارات SpEL با ابرداده‌های پیکربندی مبتنی بر XML یا مبتنی بر حاشیه‌نویسی برای تعریف نمونه‌های BeanDefinition استفاده کنید. در هر دو مورد، سینتکس برای تعریف عبارت به شکل #{ <expression string> }.است.

#### **4.3.1. XML configuration**

یک ویژگی یا مقدار آرگومان سازنده را می توان با استفاده از عبارات تنظیم کرد، همانطور که مثال زیر نشان می دهد:

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

همه دانه‌ها در زمینه برنامه به‌عنوان متغیرهای از پیش تعریف‌شده با نام مشترکشان در دسترس هستند. این شامل دانه‌های بافت استاندارد مانند محیط (از نوع org.springframework.core.env.Environment) و همچنین systemProperties و systemEnvironment (از نوع Map<String, Object>) برای دسترسی به محیط زمان اجرا می‌شود.

مثال زیر دسترسی به systemProperties bean را به عنوان یک متغیر SpEL نشان می دهد:

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">

<property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

*<!-- other properties -->*

</bean>

توجه داشته باشید که در اینجا مجبور نیستید متغیر از پیش تعریف شده را با نماد # پیشوند قرار دهید.

همانطور که در مثال زیر نشان می دهد، می توانید به سایر خواص لوبیا با نام اشاره کنید:

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

*<!-- other properties -->*

</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">

<property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

*<!-- other properties -->*

</bean>

#### **4.3.2. Annotation config**

برای تعیین یک مقدار پیش‌فرض، می‌توانید حاشیه‌نویسی @Value را روی فیلدها، متدها و پارامترهای متد یا سازنده قرار دهید.

مثال زیر مقدار پیش فرض یک متغیر فیلد را تعیین می کند:

**public** **static** **class** **FieldValueTestBean**

@Value("#{ systemProperties['user.region'] }")

**private** String defaultLocale;

**public** **void** setDefaultLocale(String defaultLocale) {

this.defaultLocale = defaultLocale;

}

**public** String getDefaultLocale() {

**return** this.defaultLocale;

}

}

مثال زیر روش معادل اما بر روی یک تنظیم کننده ویژگی را نشان می دهد:

**public** **static** **class** **PropertyValueTestBean**

**private** String defaultLocale;

@Value("#{ systemProperties['user.region'] }")

**public** **void** setDefaultLocale(String defaultLocale) {

this.defaultLocale = defaultLocale;

}

**public** String getDefaultLocale() {

**return** this.defaultLocale;

}

}

**public** **class** **MovieRecommender** {

**private** String defaultLocale;

**private** CustomerPreferenceDao customerPreferenceDao;

@Autowired

**public** MovieRecommender(CustomerPreferenceDao customerPreferenceDao,

@Value("#{systemProperties['user.country']}") String defaultLocale) {

this.customerPreferenceDao = customerPreferenceDao;

this.defaultLocale = defaultLocale;

}

*// ...*

}

### **4.4. Language Reference**

#### **4.4.1. Literal expressions**

انواع عبارات تحت اللفظی پشتیبانی شده عبارتند از رشته ها، مقادیر عددی (int، real، hex)، boolean و null. رشته ها با نقل قول های منفرد مشخص می شوند. برای قرار دادن خود یک نقل قول در یک رشته، از دو کاراکتر تک نقل قول استفاده کنید.

فهرست زیر استفاده ساده از لفظ را نشان می دهد. معمولاً آنها به صورت مجزا مانند این مورد استفاده قرار نمی گیرند، بلکه به عنوان بخشی از یک عبارت پیچیده تر، به عنوان مثال استفاده از یک لفظ در یک طرف یک عملگر مقایسه منطقی استفاده می شود.

ExpressionParser parser = **new** SpelExpressionParser();

*// evals to "Hello World"*

String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

**double** avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

*// evals to 2147483647*

**int** maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

**boolean** trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();

اعداد از استفاده از علامت منفی، نماد نمایی و اعشار پشتیبانی می کنند. به طور پیش فرض اعداد واقعی با استفاده از Double.parseDouble () تجزیه می شوند.

#### **4.4.2. Properties, Arrays, Lists, Maps, Indexers**

پیمایش با ارجاعات دارایی آسان است: فقط از یک نقطه برای نشان دادن مقدار ویژگی تودرتو استفاده کنید. نمونه‌های کلاس Inventor، pupin و tesla با داده‌های فهرست شده در بخش کلاس‌های مورد استفاده در مثال‌ها پر شدند. برای پیمایش "پایین" و دریافت سال تولد تسلا و شهر تولد پوپین از عبارات زیر استفاده می شود.

*// evals to 1856*

**int** year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);

عدم حساسیت حروف کوچک و بزرگ برای حرف اول نام اموال مجاز است. محتویات آرایه ها و لیست ها با استفاده از علامت براکت به دست می آیند.

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

*// Inventions Array*

*// evaluates to "Induction motor"*

String invention = parser.parseExpression("inventions[3]").getValue(

context, tesla, String.class);

*// Members List*

*// evaluates to "Nikola Tesla"*

String name = parser.parseExpression("Members[0].Name").getValue(

context, ieee, String.class);

*// List and Array navigation*

*// evaluates to "Wireless communication"*

String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(

context, ieee, String.class);

محتویات نقشه ها با تعیین مقدار کلید تحت اللفظی درون براکت ها به دست می آید. در این حالت، چون کلیدهای نقشه افسرها رشته‌ای هستند، می‌توانیم حرف‌های رشته را تعیین کنیم.

*// Officer's Dictionary*

Inventor pupin = parser.parseExpression("Officers['president']").getValue(

societyContext, Inventor.class);

*// evaluates to "Idvor"*

String city = parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(

societyContext, String.class);

*// setting values*

parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(

societyContext, "Croatia");

#### **4.4.3. Inline lists**

لیست ها را می توان مستقیماً در یک عبارت با استفاده از نماد {} بیان کرد.

*// evaluates to a Java list containing the four numbers*

List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

{} به خودی خود به معنای یک لیست خالی است. به دلایل عملکرد، اگر لیست به طور کامل از لفظ های ثابت تشکیل شده باشد، به جای ساختن یک لیست جدید در هر ارزیابی، یک لیست ثابت برای نمایش عبارت ایجاد می شود.

#### **4.4.4. Inline Maps**

نقشه ها همچنین می توانند مستقیماً در یک عبارت با استفاده از نماد {key:value} بیان شوند.

*// evaluates to a Java map containing the two entries*

Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);

{:} به خودی خود به معنای یک نقشه خالی است. به دلایل عملکرد، اگر نقشه خود از لفظهای ثابت یا دیگر ساختارهای ثابت تو در تو (لیست ها یا نقشه ها) تشکیل شده باشد، به جای ساختن یک نقشه جدید در هر ارزیابی، یک نقشه ثابت برای نمایش عبارت ایجاد می شود. نقل قول از کلیدهای نقشه اختیاری است، مثال های بالا از کلیدهای نقل قول استفاده نمی کنند.

#### **4.4.5. Array construction**

آرایه‌ها را می‌توان با استفاده از نحو آشنای جاوا ساخت، که به‌صورت اختیاری یک مقداردهی اولیه برای پر شدن آرایه در زمان ساخت ارائه می‌کند.

**int[]** numbers1 = (**int[]**) parser.parseExpression("new int[4]").getValue(context);

*// Array with initializer*

**int[]** numbers2 = (**int[]**) parser.parseExpression("new int[]{1,2,3}").getValue(context);

*// Multi dimensional array*

**int[][]** numbers3 = (**int[][]**) parser.parseExpression("new int[4][5]").getValue(context);

در حال حاضر ارائه اولیه ساز هنگام ساخت یک آرایه چند بعدی مجاز نیست.

#### **4.4.6. Methods**

روش ها با استفاده از نحو برنامه نویسی جاوا معمولی فراخوانی می شوند. همچنین می‌توانید از روش‌هایی در لفظ استفاده کنید. Varargs نیز پشتیبانی می شود.

*// string literal, evaluates to "bc"*

String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

*// evaluates to true*

**boolean** isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(

societyContext, Boolean.class);

#### **4.4.7. Operators**

##### **Relational operators**

عملگرهای رابطه ای؛ برابر، نه مساوی، کمتر از، کمتر یا مساوی، بزرگتر، و بزرگتر از یا مساوی با استفاده از نماد عملگر استاندارد پشتیبانی می شوند.

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class);

مقایسه بزرگتر/کمتر از تهی از یک قانون ساده پیروی می کند: در اینجا null به عنوان هیچ تلقی می شود (یعنی نه به عنوان صفر). در نتیجه، هر مقدار دیگری همیشه بزرگتر از null است (X > null همیشه درست است) و هیچ مقدار دیگری کمتر از هیچ نیست (X < null همیشه نادرست است).

اگر مقایسه‌های عددی را ترجیح می‌دهید، لطفاً از مقایسه‌های تهی مبتنی بر اعداد به نفع مقایسه با صفر خودداری کنید (مثلاً X > 0 یا X < 0).

علاوه بر عملگرهای رابطه‌ای استاندارد، SpEL از عملگرهای instanceof و منطبق بر بیان منظم پشتیبانی می‌کند.

*// evaluates to false*

**boolean** falseValue = parser.parseExpression(

"'xyz' instanceof T(Integer)").getValue(Boolean.class);

*// evaluates to true*

**boolean** trueValue = parser.parseExpression(

"'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

*//evaluates to false*

**boolean** falseValue = parser.parseExpression(

"'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

مراقب انواع ابتدایی باشید زیرا آنها بلافاصله تا نوع wrapper در جعبه قرار می گیرند، بنابراین 1 نمونه از T(int) به نادرست و 1 نمونه T (Integer) به درستی ارزیابی می شود، همانطور که انتظار می رود.

هر عملگر نمادین را می توان به عنوان یک معادل کاملاً الفبایی نیز مشخص کرد. این امر از مشکلاتی که در آن نمادهای استفاده شده برای نوع سندی که عبارت در آن تعبیه شده است، معنای خاصی دارند (مثلاً یک سند XML) جلوگیری می کند. معادل‌های متنی در اینجا نشان داده شده‌اند: lt (<)، gt (>)، le (<=)، ge (>=)، eq (==)، ne (!=)، div (/)، mod (%) ، نه (!). اینها به حروف بزرگ و کوچک حساس هستند.

##### **Logical operators**

عملگرهای منطقی که پشتیبانی می شوند عبارتند از و، یا، و نه. استفاده از آنها در زیر نشان داده شده است.

*// -- AND --*

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

*// evaluates to true*

String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- OR --*

*// evaluates to true*

**boolean** trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

*// evaluates to true*

String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";

**boolean** trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

*// -- NOT --*

*// evaluates to false*

**boolean** falseValue = parser.parseExpression("!true").getValue(Boolean.class);

*// -- AND and NOT --*

String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";

**boolean** falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

##### **Mathematical operators**

عملگر جمع را می توان هم روی اعداد و هم برای رشته ها استفاده کرد. تفریق، ضرب و تقسیم فقط روی اعداد قابل استفاده است. سایر عملگرهای ریاضی پشتیبانی شده مدول (%) و توان نمایی (^) هستند. اولویت عملگر استاندارد اعمال می شود. این عملگرها در زیر نشان داده شده اند.

*// Addition*

**int** two = parser.parseExpression("1 + 1").getValue(Integer.class); *// 2*

String testString = parser.parseExpression(

"'test' + ' ' + 'string'").getValue(String.class); *// 'test string'*

*// Subtraction*

**int** four = parser.parseExpression("1 - -3").getValue(Integer.class); *// 4*

**double** d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class); *// -9000*

*// Multiplication*

**int** six = parser.parseExpression("-2 \* -3").getValue(Integer.class); *// 6*

**double** twentyFour = parser.parseExpression("2.0 \* 3e0 \* 4").getValue(Double.class); *// 24.0*

*// Division*

**int** minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class); *// -2*

**double** one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class); *// 1.0*

*// Modulus*

**int** three = parser.parseExpression("7 % 4").getValue(Integer.class); *// 3*

**int** one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); *// 1*

*// Operator precedence*

**int** minusTwentyOne = parser.parseExpression("1+2-3\*8").getValue(Integer.class); *// -21*

#### **4.4.8. Assignment**

تنظیم یک ویژگی با استفاده از عملگر انتساب انجام می شود. این معمولاً در یک فراخوانی به setValue انجام می‌شود، اما می‌تواند در تماس با getValue نیز انجام شود.

Inventor inventor = **new** Inventor();

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

parser.parseExpression("Name").setValue(context, inventor, "Aleksandar Seovic");

*// alternatively*

String aleks = parser.parseExpression(

"Name = 'Aleksandar Seovic'").getValue(context, inventor, String.class);

#### **4.4.9. Types**

عملگر T ویژه را می توان برای تعیین نمونه ای از java.lang.Class (نوع) استفاده کرد. روش های استاتیک نیز با استفاده از این عملگر فراخوانی می شوند. StandardEvaluationContext از TypeLocator برای یافتن انواع استفاده می کند و StandardTypeLocator (که می تواند جایگزین شود) با درک بسته java.lang ساخته شده است. این بدان معناست که ارجاعات T() به انواع در java.lang نیازی به واجد شرایط بودن کامل ندارند، اما همه مراجع نوع دیگر باید واجد شرایط باشند.

Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

**boolean** trueValue = parser.parseExpression(

"T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")

.getValue(Boolean.class);

#### **4.4.10. Constructors**

سازنده ها را می توان با استفاده از اپراتور جدید فراخوانی کرد. نام کلاس کاملاً واجد شرایط باید برای همه استفاده شود به جز نوع اولیه و String (جایی که می توان از int، float و غیره استفاده کرد).

Inventor einstein = p.parseExpression(

"new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")

.getValue(Inventor.class);

*//create new inventor instance within add method of List*

p.parseExpression(

"Members.add(new org.spring.samples.spel.inventor.Inventor(

'Albert Einstein', 'German'))").getValue(societyContext);

#### **4.4.11. Variables**

متغیرها را می توان در عبارت با استفاده از نحو #variableName ارجاع داد. متغیرها با استفاده از متد setVariable در پیاده سازی های EvaluationContext تنظیم می شوند.

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context, tesla);

System.out.println(tesla.getName()) *// "Mike Tesla"*

##### **The #this and #root variables**

متغیر #this همیشه تعریف می‌شود و به شی ارزیابی فعلی اشاره می‌کند (در مقابل آن مراجع غیرمجاز حل می‌شوند). متغیر #root همیشه تعریف می شود و به شیء متن ریشه اشاره دارد. اگرچه #این ممکن است با ارزیابی اجزای یک عبارت متفاوت باشد، #root همیشه به ریشه اشاره دارد.

*// create an array of integers*

List<Integer> primes = **new** ArrayList<Integer>();

primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

*// create parser and set variable 'primes' as the array of integers*

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();

context.setVariable("primes", primes);

*// all prime numbers > 10 from the list (using selection ?{...})*

*// evaluates to [11, 13, 17]*

List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression(

"#primes.?[#this>10]").getValue(context);

#### **4.4.12. Functions**

می‌توانید با ثبت توابع تعریف‌شده توسط کاربر که می‌توانند در رشته عبارت فراخوانی شوند، SpEL را گسترش دهید. تابع از طریق EvaluationContext ثبت می شود.

Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("myFunction", method);

به عنوان مثال، یک متد کاربردی برای معکوس کردن یک رشته در زیر نشان داده شده است:

**public** **abstract** **class** **StringUtils** {

**public** **static** String reverseString(String input) {

StringBuilder backwards = **new** StringBuilder(input.length());

**for** (**int** i = 0; i < input.length(); i++)

backwards.append(input.charAt(input.length() - 1 - i));

}

**return** backwards.toString();

}

}

سپس روش فوق را می توان به صورت زیر ثبت و استفاده کرد:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("reverseString",

StringUtils.class.getDeclaredMethod("reverseString", String.class));

String helloWorldReversed = parser.parseExpression(

"#reverseString('hello')").getValue(context, String.class);

#### **4.4.13. Bean references**

اگر زمینه ارزیابی با یک حل‌کننده bean پیکربندی شده باشد، می‌توان با استفاده از نماد @، beans را از یک عبارت جستجو کرد.

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"foo") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("@foo").getValue(context);

برای دسترسی به خود کارخانه، نام لوبیا باید با علامت & پیشوند باشد.

ExpressionParser parser = **new** SpelExpressionParser();

StandardEvaluationContext context = **new** StandardEvaluationContext();

context.setBeanResolver(**new** MyBeanResolver());

*// This will end up calling resolve(context,"&foo") on MyBeanResolver during evaluation*

Object bean = parser.parseExpression("&foo").getValue(context);

#### **4.4.14. Ternary Operator (If-Then-Else)**

می توانید از عملگر سه تایی برای اجرای منطق شرطی if-then-else در داخل عبارت استفاده کنید. یک مثال حداقلی این است:

String falseString = parser.parseExpression(

"false ? 'trueExp' : 'falseExp'").getValue(String.class);

در این مورد، خطای بولی، مقدار رشته "falseExp" را برمی گرداند. یک مثال واقعی تر در زیر نشان داده شده است.

parser.parseExpression("Name").setValue(societyContext, "IEEE");

societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " +

"+ Name + ' Society' : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString = parser.parseExpression(expression)

.getValue(societyContext, String.class);

*// queryResultString = "Nikola Tesla is a member of the IEEE Society"*

#### **4.4.15. The Elvis Operator**

عملگر Elvis کوتاه کردن دستور عملگر سه تایی است و در زبان Groovy استفاده می شود. با دستور عملگر سه تایی، معمولاً باید یک متغیر را دو بار تکرار کنید، همانطور که مثال زیر نشان می دهد:

String name = "Elvis Presley";

String displayName = (name != null ? name : "Unknown");

در عوض، می توانید از اپراتور الویس (که به دلیل شباهت به مدل موی الویس نامگذاری شده است) استفاده کنید. مثال زیر نحوه استفاده از عملگر Elvis را نشان می دهد:

ExpressionParser parser = **new** SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(**new** Inventor(), String.class);

System.out.println(name); *// 'Unknown'*

فهرست زیر مثال پیچیده تری را نشان می دهد:

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Nikola Tesla*

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); *// Elvis Presley*

#### **4.4.16. Safe Navigation operator**

عملگر Safe Navigation برای اجتناب از NullPointerException استفاده می شود و از زبان Groovy می آید. معمولاً وقتی به یک شی ارجاع دارید، ممکن است لازم باشد قبل از دسترسی به متدها یا خصوصیات شیء، تأیید کنید که آن شیء تهی نیست. برای جلوگیری از این امر، اپراتور ناوبری ایمن به جای ایجاد یک استثنا، به سادگی null را برمی گرداند.

ExpressionParser parser = **new** SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = **new** Inventor("Nikola Tesla", "Serbian");

tesla.setPlaceOfBirth(**new** PlaceOfBirth("Smiljan"));

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); *// Smiljan*

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); *// null - does not throw NullPointerException!!!*

عملگر Elvis را می توان برای اعمال مقادیر پیش فرض در عبارات استفاده کرد، به عنوان مثال. در یک عبارت @Value:

@Value("#{systemProperties['pop3.port'] ?: 25}")

این یک ویژگی سیستمی pop3.port را اگر تعریف شده باشد یا 25 را در غیر این صورت تزریق می کند.

#### **4.4.17. Collection Selection**

Selection یک ویژگی زبان عبارت قدرتمند است که به شما امکان می دهد با انتخاب از ورودی های آن، برخی از مجموعه های منبع را به دیگری تبدیل کنید.

انتخاب از نحو .?[selectionExpression] استفاده می کند. با این کار مجموعه فیلتر شده و مجموعه جدیدی حاوی زیرمجموعه ای از عناصر اصلی را برمی گرداند. به عنوان مثال، انتخاب به ما امکان می دهد به راحتی لیستی از مخترعان صرب را دریافت کنیم:

List<Inventor> list = (List<Inventor>) parser.parseExpression(

"Members.?[Nationality == 'Serbian']").getValue(societyContext);

انتخاب بر روی هر دو لیست و نقشه امکان پذیر است. در حالت اول، معیارهای انتخاب بر اساس هر عنصر فهرست جداگانه ارزیابی می‌شوند، در حالی که در برابر یک نقشه، معیارهای انتخاب در برابر هر ورودی نقشه (اشیاء از نوع Map.Entry جاوا) ارزیابی می‌شوند. ورودی های نقشه کلید و ارزش خود را به عنوان ویژگی برای استفاده در انتخاب قابل دسترسی هستند.

این عبارت یک نقشه جدید متشکل از عناصر نقشه اصلی که مقدار ورودی کمتر از 27 است را برمی گرداند.

Map newMap = parser.parseExpression("map.?[value<27]").getValue();

علاوه بر برگرداندن تمام عناصر انتخاب شده، امکان بازیابی فقط اولین یا آخرین مقدار وجود دارد. برای به دست آوردن اولین ورودی مطابق با انتخاب، نحو .^[selectionExpression] است، در حالی که برای به دست آوردن آخرین انتخاب منطبق، نحو .$[selectionExpression] است.

#### **4.4.18. Collection Projection**

طرح ریزی به یک مجموعه اجازه می دهد تا ارزیابی یک عبارت فرعی را هدایت کند و نتیجه یک مجموعه جدید است. نحو برای طرح ریزی .![projectionExpression] است. با مثال به راحتی قابل درک است، فرض کنید فهرستی از مخترعان داریم اما لیست شهرهایی را می خواهیم که در آن متولد شده اند. عملاً می‌خواهیم «placeOfBirth.city» را برای هر ورودی در فهرست مخترعان ارزیابی کنیم. استفاده از طرح ریزی:

*// returns ['Smiljan', 'Idvor' ]*

List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");

یک نقشه همچنین می تواند برای هدایت طرح استفاده شود و در این مورد بیان طرح ریزی در برابر هر ورودی در نقشه ارزیابی می شود (به عنوان یک Java Map.Entry نشان داده می شود). نتیجه یک طرح ریزی در سراسر یک نقشه، لیستی است که شامل ارزیابی بیان طرح ریزی در برابر هر ورودی نقشه است.

#### **4.4.19. Expression templating**

قالب‌های بیان اجازه می‌دهند متن تحت اللفظی با یک یا چند بلوک ارزیابی ترکیب شوند. هر بلوک ارزیابی با کاراکترهای پیشوندی و پسوندی که می توانید تعریف کنید، مشخص می شود، یک انتخاب رایج استفاده از #{ } به عنوان جداکننده است. مثلا،

String randomPhrase = parser.parseExpression(

"random number is #{T(java.lang.Math).random()}",

**new** TemplateParserContext()).getValue(String.class);

*// evaluates to "random number is 0.7038186818312008"*

رشته با الحاق متن تحت اللفظی 'عدد تصادفی است' با نتیجه ارزیابی عبارت داخل جداکننده #{}، در این مورد نتیجه فراخوانی متد تصادفی() ارزیابی می‌شود. آرگومان دوم متد parseExpression() از نوع ParserContext است. رابط ParserContext برای تأثیرگذاری بر نحوه تجزیه عبارت به منظور پشتیبانی از عملکرد قالب عبارت استفاده می شود. تعریف TemplateParserContext در زیر نشان داده شده است.

**public** **class** **TemplateParserContext** **implements** ParserContext {

**public** String getExpressionPrefix() {

**return** "#{";

}

**public** String getExpressionSuffix() {

**return** "}";

}

**public** **boolean** isTemplate() {

**return** true;

}

}

### **4.5. Classes used in the examples**

Inventor.java

**package** org.spring.samples.spel.inventor;

**import** java.util.Date;

**import** java.util.GregorianCalendar;

**public** **class** **Inventor** {

**private** String name;

**private** String nationality;

**private** String**[]** inventions;

**private** Date birthdate;

**private** PlaceOfBirth placeOfBirth;

**public** Inventor(String name, String nationality) {

GregorianCalendar c= **new** GregorianCalendar();

this.name = name;

this.nationality = nationality;

this.birthdate = c.getTime();

}

**public** Inventor(String name, Date birthdate, String nationality) {

this.name = name;

this.nationality = nationality;

this.birthdate = birthdate;

}

**public** Inventor() {

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** String getNationality() {

**return** nationality;

}

**public** **void** setNationality(String nationality) {

this.nationality = nationality;

}

**public** Date getBirthdate() {

**return** birthdate;

}

**public** **void** setBirthdate(Date birthdate) {

this.birthdate = birthdate;

}

**public** PlaceOfBirth getPlaceOfBirth() {

**return** placeOfBirth;

}

**public** **void** setPlaceOfBirth(PlaceOfBirth placeOfBirth) {

this.placeOfBirth = placeOfBirth;

}

**public** **void** setInventions(String**[]** inventions) {

this.inventions = inventions;

}

**public** String**[]** getInventions() {

**return** inventions;

}

}

PlaceOfBirth.java

**package** org.spring.samples.spel.inventor;

**public** **class** **PlaceOfBirth** {

**private** String city;

**private** String country;

**public** PlaceOfBirth(String city) {

this.city=city;

}

**public** PlaceOfBirth(String city, String country) {

this(city);

this.country = country;

}

**public** String getCity() {

**return** city;

}

**public** **void** setCity(String s) {

this.city = s;

}

**public** String getCountry() {

**return** country;

}

**public** **void** setCountry(String country) {

this.country = country;

}

}

Society.java

**package** org.spring.samples.spel.inventor;

**import** java.util.\*;

**public** **class** **Society** {

**private** String name;

**public** **static** String Advisors = "advisors";

**public** **static** String President = "president";

**private** List<Inventor> members = **new** ArrayList<Inventor>();

**private** Map officers = **new** HashMap();

**public** List getMembers() {

**return** members;

}

**public** Map getOfficers() {

**return** officers;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** **boolean** isMember(String name) {

**for** (Inventor inventor : members) {

**if** (inventor.getName().equals(name)) {

**return** true;

}

}

**return** false;

}

}

# **5. Aspect Oriented Programming with Spring**

## **5.1. Introduction**

برنامه نویسی جنبه گرا (AOP) برنامه نویسی شی گرا (OOP) را با ارائه روش دیگری برای تفکر در مورد ساختار برنامه تکمیل می کند. واحد کلیدی مدولاریت در OOP کلاس است، در حالی که در AOP واحد مدولاریت AOP است. جنبه‌ها، مدولار کردن نگرانی‌هایی مانند مدیریت تراکنش‌ها را ممکن می‌سازد که انواع و اشیاء متعدد را برش می‌دهد. (در ادبیات AOP چنین نگرانی هایی معمولاً نگرانی های مقطعی نامیده می شوند.)

یکی از اجزای کلیدی Spring چارچوب AOP است. در حالی که کانتینر Spring IoC به AOP وابسته نیست، به این معنی که اگر نمی‌خواهید نیازی به استفاده از AOP ندارید، AOP مکمل Spring IoC برای ارائه یک راه‌حل میان‌افزار بسیار توانمند است.

Spring 2.0+ AOP

Spring 2.0 یک روش ساده تر و قدرتمندتر برای نوشتن جنبه های سفارشی با استفاده از رویکرد مبتنی بر طرحواره یا سبک حاشیه نویسی @AspectJ معرفی کرد. هر دوی این سبک ها توصیه های کاملاً تایپ شده و استفاده از زبان نقطه برش AspectJ را ارائه می دهند، در حالی که هنوز از Spring AOP برای بافتن استفاده می کنند.

پشتیبانی از طرحواره Spring 2.0+ و AOP مبتنی بر @AspectJ در این فصل مورد بحث قرار گرفته است. پشتیبانی سطح پایین AOP، همانطور که معمولاً در برنامه‌های Spring 1.2 در معرض دید قرار می‌گیرد، در فصل بعد مورد بحث قرار می‌گیرد.

AOP در چارچوب Spring برای… استفاده می شود

ارائه خدمات سازمانی اعلامی، به ویژه به عنوان جایگزینی برای خدمات اعلامی EJB. مهمترین چنین خدماتی مدیریت تراکنش های اعلامی است.

... به کاربران اجازه می دهد تا جنبه های سفارشی را پیاده سازی کنند و استفاده از OOP را با AOP تکمیل کنند.

اگر فقط به خدمات اعلامی عمومی یا سایر سرویس‌های میان‌افزار اعلامی از پیش بسته‌بندی شده مانند pooling علاقه دارید، نیازی به کار مستقیم با Spring AOP ندارید و می‌توانید بیشتر این فصل را نادیده بگیرید.

#### **5.1.1. AOP concepts**

اجازه دهید با تعریف برخی از مفاهیم و اصطلاحات مرکزی AOP شروع کنیم. این اصطلاحات مختص بهار نیستند… متاسفانه، اصطلاحات AOP به طور خاص بصری نیستند. با این حال، اگر اسپرینگ از اصطلاحات خاص خود استفاده کند، گیج کننده تر خواهد بود.

Aspect: مدولار کردن یک نگرانی که در کلاس های متعدد قرار می گیرد. مدیریت تراکنش نمونه خوبی از یک نگرانی بین بخشی در برنامه های جاوا سازمانی است. در Spring AOP، جنبه‌ها با استفاده از کلاس‌های منظم (رویکرد مبتنی بر طرحواره) یا کلاس‌های معمولی مشروح‌شده با حاشیه‌نویسی @Aspect (سبک @AspectJ) پیاده‌سازی می‌شوند.

Join point: نقطه ای در حین اجرای یک برنامه، مانند اجرای یک متد یا مدیریت یک استثنا. در Spring AOP، یک نقطه اتصال همیشه اجرای یک متد را نشان می دهد.

Advice: اقدامی که توسط یک جنبه در یک نقطه اتصال خاص انجام می شود. انواع مختلف نصیحت شامل نصیحت «در اطراف»، «قبل» و «بعد» است. (انواع مشاوره در زیر مورد بحث قرار می گیرد.) بسیاری از چارچوب های AOP، از جمله Spring، یک توصیه را به عنوان یک رهگیر مدل می کنند و زنجیره ای از رهگیرها را در اطراف نقطه اتصال حفظ می کنند.

Pointcut: گزاره ای که با نقاط پیوسته مطابقت دارد. Advice با یک عبارت pointcut مرتبط است و در هر نقطه اتصالی که با pointcut مطابقت دارد اجرا می شود (به عنوان مثال، اجرای یک متد با یک نام خاص). مفهوم اتصال نقاط مطابق با عبارات نقطه‌بردار در AOP مرکزی است و Spring به‌طور پیش‌فرض از زبان عبارت نقطه‌بردار AspectJ استفاده می‌کند.

Introduction: اعلام متدها یا فیلدهای اضافی از طرف یک نوع. Spring AOP به شما اجازه می دهد تا رابط های جدید (و پیاده سازی مربوطه) را به هر شی توصیه شده معرفی کنید. برای مثال، می‌توانید از مقدمه‌ای استفاده کنید تا یک Bean یک رابط IsModified را پیاده‌سازی کنید تا ذخیره‌سازی را ساده کنید. (یک مقدمه در جامعه AspectJ به عنوان یک اعلامیه بین نوع شناخته می شود.)

Target object: شیئی که توسط یک یا چند جنبه توصیه می شود. همچنین به عنوان شی توصیه شده نامیده می شود. از آنجایی که Spring AOP با استفاده از پراکسی های زمان اجرا پیاده سازی می شود، این شی همیشه یک شی پراکسی خواهد بود.

AOP proxy: یک شی ایجاد شده توسط چارچوب AOP به منظور اجرای قراردادهای جنبه (مشاوره در اجرای روش و غیره). در چارچوب Spring، یک پراکسی AOP یک پروکسی پویا JDK یا یک پروکسی CGLIB خواهد بود.

Weaving: پیوند دادن جنبه ها با انواع دیگر برنامه ها یا اشیاء برای ایجاد یک شی توصیه شده. این را می توان در زمان کامپایل (برای مثال با استفاده از کامپایلر AspectJ)، زمان بارگذاری یا در زمان اجرا انجام داد. Spring AOP، مانند سایر فریم ورک های خالص Java AOP، بافت را در زمان اجرا انجام می دهد.

در مورد نصیحت عمومی ترین نوع نصیحت است. از آنجایی که Spring AOP، مانند AspectJ، طیف کاملی از انواع مشاوره را ارائه می دهد، توصیه می کنیم از کم قدرت ترین نوع مشاوره استفاده کنید که بتواند رفتار مورد نیاز را اجرا کند. به عنوان مثال، اگر فقط نیاز به به روز رسانی یک کش با مقدار بازگشتی یک متد دارید، بهتر است یک توصیه پس از بازگشت را پیاده سازی کنید تا یک توصیه پیرامون، اگرچه یک مشاوره در اطراف می تواند همان کار را انجام دهد. استفاده از خاص ترین نوع مشاوره، مدل برنامه نویسی ساده تری را با پتانسیل کمتری برای خطا فراهم می کند. به عنوان مثال، شما نیازی به فراخوانی متد () () در JoinPoint مورد استفاده برای مشاوره پیرامون ندارید و بنابراین نمی توانید آن را فراخوانی نکنید.

در Spring 2.0، تمام پارامترهای مشاوره به صورت ایستا تایپ می شوند، به طوری که شما با پارامترهای توصیه از نوع مناسب (مثلاً نوع مقدار بازگشتی از اجرای روش) به جای آرایه های Object کار می کنید.

مفهوم نقاط اتصال، که با نقاط برش مطابقت دارد، کلید AOP است که آن را از فناوری‌های قدیمی‌تری که فقط رهگیری ارائه می‌دهند متمایز می‌کند. نقطه‌برها این امکان را فراهم می‌کنند که توصیه‌ها مستقل از سلسله مراتب شی گرا مورد هدف قرار گیرند. به عنوان مثال، یک توصیه پیرامون ارائه مدیریت تراکنش های اعلامی را می توان برای مجموعه ای از روش ها اعمال کرد که چندین شی را در بر می گیرد (مانند تمام عملیات تجاری در لایه خدمات).

#### **5.1.2. Spring AOP capabilities and goals**

Spring AOP در جاوا خالص پیاده سازی شده است. نیازی به فرآیند کامپایل خاصی نیست. Spring AOP نیازی به کنترل سلسله مراتب کلاس loader ندارد و بنابراین برای استفاده در کانتینر Servlet یا سرور برنامه مناسب است.

Spring AOP در حال حاضر فقط از نقاط اتصال اجرای روش (توصیه به اجرای متدها در Spring beans) پشتیبانی می کند. رهگیری میدانی اجرا نشده است، اگرچه پشتیبانی برای رهگیری میدانی می‌تواند بدون شکستن APIهای هسته Spring AOP اضافه شود. اگر نیاز به راهنمایی برای دسترسی میدانی و به روز رسانی نقاط اتصال دارید، زبانی مانند AspectJ را در نظر بگیرید.

رویکرد Spring AOP به AOP با سایر چارچوب های AOP متفاوت است. هدف ارائه کامل ترین اجرای AOP نیست (اگرچه Spring AOP کاملاً قادر است). بلکه ایجاد یکپارچگی نزدیک بین اجرای AOP و Spring IoC برای کمک به حل مشکلات رایج در برنامه های سازمانی است.

بنابراین، برای مثال، عملکرد AOP Spring Framework معمولاً همراه با کانتینر Spring IoC استفاده می‌شود. جنبه‌ها با استفاده از نحو تعریف معمولی bean پیکربندی می‌شوند (اگرچه این قابلیت‌های قدرتمند «خودپروکسی» را امکان‌پذیر می‌کند): این یک تفاوت اساسی با سایر پیاده‌سازی‌های AOP است. برخی از کارها وجود دارد که نمی توانید به راحتی یا کارآمد با Spring AOP انجام دهید، مانند توصیه به اشیاء بسیار ریز دانه (مانند اشیاء دامنه معمولا): AspectJ بهترین انتخاب در چنین مواردی است. با این حال، تجربه ما این است که Spring AOP یک راه حل عالی برای اکثر مشکلات در برنامه های کاربردی جاوا سازمانی ارائه می دهد که قابل پذیرش AOP هستند.

Spring AOP هرگز برای رقابت با AspectJ برای ارائه یک راه حل جامع AOP تلاش نمی کند. ما معتقدیم که هر دو چارچوب مبتنی بر پروکسی مانند Spring AOP و فریمورک‌های کامل مانند AspectJ ارزشمند هستند، و مکمل یکدیگر هستند، نه در رقابت. Spring به طور یکپارچه Spring AOP و IoC را با AspectJ ادغام می کند تا تمام کاربردهای AOP را در یک معماری برنامه کاربردی مبتنی بر Spring سازگار انجام دهد. این ادغام بر Spring AOP API یا AOP Alliance API تأثیر نمی‌گذارد: Spring AOP سازگار با عقب باقی می‌ماند. برای بحث در مورد Spring AOP های AOP به فصل زیر مراجعه کنید.

یکی از اصول اصلی چارچوب بهار، عدم تهاجم است. این ایده ای است که نباید مجبور شوید کلاس ها و رابط های خاص چارچوب را در مدل کسب و کار/دامنه خود وارد کنید. با این حال، در برخی مکان‌ها، Spring Framework به شما این امکان را می‌دهد که وابستگی‌های خاص Spring Framework را در پایگاه کد خود معرفی کنید: دلیل منطقی در ارائه چنین گزینه‌هایی به این دلیل است که در سناریوهای خاص ممکن است خواندن یا کدگذاری بخشی خاص ساده‌تر باشد. عملکرد به این صورت اگرچه چارچوب Spring (تقریباً) همیشه این انتخاب را به شما ارائه می دهد: شما این آزادی را دارید که آگاهانه تصمیم بگیرید که کدام گزینه به بهترین وجه مناسب مورد یا سناریوی خاص شما است.

یکی از این گزینه‌ها که مربوط به این فصل است، این است که کدام چارچوب AOP (و کدام سبک AOP) را انتخاب کنیم. شما می‌توانید AspectJ و/یا Spring AOP را انتخاب کنید، و همچنین می‌توانید رویکرد سبک حاشیه‌نویسی @AspectJ یا رویکرد سبک پیکربندی Spring XML را انتخاب کنید. این واقعیت که این فصل ابتدا رویکرد سبک @AspectJ را معرفی می‌کند نباید به عنوان نشانه‌ای در نظر گرفته شود که تیم Spring از رویکرد سبک حاشیه‌نویسی @AspectJ نسبت به سبک پیکربندی Spring XML حمایت می‌کند.

برای بحث کامل تر در مورد چرایی ها و دلایل هر سبک به انتخاب سبک اعلان AOP برای استفاده مراجعه کنید.

#### **5.1.3. AOP Proxies**

SpringAOP به طور پیش فرض از پراکسی های پویا JDK استاندارد برای پراکسی های AOP استفاده می کند. این اجازه می دهد تا هر رابط (یا مجموعه ای از اینترفیس ها) پروکسی شود.

Spring AOP همچنین می تواند از پروکسی های CGLIB استفاده کند. این برای کلاس های پروکسی به جای رابط ها ضروری است. CGLIB به طور پیش‌فرض در صورتی استفاده می‌شود که یک شی تجاری یک رابط را پیاده‌سازی نکند. از آنجایی که برنامه نویسی به اینترفیس ها به جای کلاس ها تمرین خوبی است. کلاس های تجاری معمولاً یک یا چند رابط تجاری را پیاده سازی می کنند. استفاده از CGLIB در مواردی (امیدوارم نادر) که نیاز به توصیه روشی دارید که در یک رابط اعلام نشده است، یا در مواردی که باید یک شی پراکسی را به یک روش به عنوان یک نوع مشخص ارسال کنید، ممکن است.

درک این واقعیت مهم است که Spring AOP مبتنی بر پروکسی است. برای بررسی دقیق اینکه دقیقاً این جزئیات پیاده سازی واقعاً چه معنایی دارد، به درک پراکسی های AOP مراجعه کنید.

### **5.2. @AspectJ support**

برای استفاده از جنبه‌های @AspectJ در پیکربندی Spring، باید پشتیبانی Spring را برای پیکربندی Spring AOP بر اساس جنبه‌های @AspectJ، و bean‌های خودکار را بر اساس اینکه آیا توسط آن جنبه‌ها توصیه می‌شود یا نه، فعال کنید. منظور ما از پروکسی خودکار این است که اگر Spring تشخیص دهد که یک bean توسط یک یا چند جنبه توصیه می شود، به طور خودکار یک پروکسی برای آن bean ایجاد می کند تا فراخوانی های متد را قطع کند و اطمینان حاصل کند که توصیه ها در صورت لزوم اجرا می شوند.

پشتیبانی @AspectJ را می توان با پیکربندی سبک XML یا جاوا فعال کرد. در هر صورت، باید مطمئن شوید که کتابخانه AspectJ's aspectjweaver.jar در مسیر کلاس برنامه شما (نسخه 1.8 یا بالاتر) قرار دارد. این کتابخانه در دایرکتوری 'lib' یک توزیع AspectJ یا از طریق مخزن مرکزی Maven موجود است.

فعال کردن پشتیبانی @AspectJ با پیکربندی جاوا

برای فعال کردن پشتیبانی AspectJ@ با Java @Configuration، حاشیه نویسی @EnableAspectJAutoProxy را اضافه کنید:

@Configuration

@EnableAspectJAutoProxy

**public** **class** **AppConfig** {

}

##### **Enabling @AspectJ Support with XML configuration**

برای فعال کردن پشتیبانی @AspectJ با پیکربندی مبتنی بر XML از عنصر aop:aspectj-autoproxy استفاده کنید:

<aop:aspectj-autoproxy/>

این فرض می کند که شما از پشتیبانی طرحواره همانطور که در پیکربندی مبتنی بر طرحواره XML توضیح داده شده است استفاده می کنید. برای نحوه وارد کردن برچسب ها در فضای نام aop به طرح AOP مراجعه کنید.

#### **5.2.2. Declaring an aspect**

با فعال بودن پشتیبانی @AspectJ، هر Bean تعریف شده در زمینه برنامه شما با کلاسی که جنبه @AspectJ دارد (دارای حاشیه نویسی @Aspect) به طور خودکار توسط Spring شناسایی شده و برای پیکربندی Spring AOP استفاده می شود. مثال زیر حداقل تعریف مورد نیاز برای یک جنبه نه چندان مفید را نشان می دهد:

یک تعریف معمولی bean در زمینه برنامه، به یک کلاس bean اشاره می کند که حاشیه نویسی @Aspect دارد:

<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">

*<!-- configure properties of aspect here as normal -->*

</bean>

و تعریف کلاس NotVeryUsefulAspect، با حاشیه نویسی org.aspectj.lang.annotation.Aspect.

**package** org.xyz;

**import** org.aspectj.lang.annotation.Aspect;

@Aspect

**public** **class** **NotVeryUsefulAspect** {

}

جنبه‌ها (کلاس‌هایی که با @Aspect حاشیه‌نویسی شده‌اند) ممکن است مانند هر کلاس دیگری متدها و فیلدهایی داشته باشند. آنها همچنین ممکن است حاوی نقطه برش، توصیه، و اعلامیه های معرفی (بین نوع) باشند.

شناسایی خودکار *aspects*ها از طریق اسکن کامپوننت

می‌توانید کلاس‌های *aspects*را به‌عنوان دانه‌های معمولی در پیکربندی Spring XML خود ثبت کنید، یا آنها را از طریق اسکن مسیر کلاس به طور خودکار شناسایی کنید - درست مانند هر Bean دیگری که توسط Spring مدیریت می‌شود. با این حال، توجه داشته باشید که حاشیه نویسی @Aspect برای تشخیص خودکار در مسیر کلاس کافی نیست: برای این منظور، باید یک حاشیه نویسی @Component جداگانه (یا متناوباً یک حاشیه نویسی کلیشه ای سفارشی که مطابق با قوانین اسکنر مؤلفه Springs واجد شرایط باشد) اضافه کنید.

*aspects* های مشاوره با جنبه های دیگر؟

در Spring AOP، این امکان وجود ندارد که خود *aspects* از جنبه‌های دیگر هدف مشاوره باشند. حاشیه نویسی @Aspect در یک کلاس آن را به عنوان یک *aspect*علامت گذاری می کند و از این رو آن را از پروکسی خودکار خارج می کند.

#### **5.2.3. Declaring a pointcut**

به یاد بیاورید که نقطه‌برها نقاط مورد علاقه را تعیین می‌کنند و بنابراین ما را قادر می‌سازند تا زمان اجرای توصیه را کنترل کنیم. Spring AOP فقط از نقاط اتصال به روش اجرای متد برای Spring beans پشتیبانی می کند، بنابراین می توانید نقطه برش را مطابق با اجرای متدها در Spring beans در نظر بگیرید. یک اعلام نقطه دارای دو بخش است: یک امضا شامل یک نام و هر پارامتر، و یک عبارت نقطه‌بر که دقیقاً مشخص می‌کند که ما به کدام روش‌های اجرایی علاقه‌مندیم. در سبک حاشیه‌نویسی @AspectJ AOP، یک امضای نقطه برش با یک روش معمولی ارائه می‌شود. تعریف، و عبارت pointcut با استفاده از حاشیه نویسی @Pointcut نشان داده می شود (روشی که به عنوان امضای نقطه برش عمل می کند باید دارای یک نوع بازگشتی باطل باشد).

یک مثال کمک می کند تا این تمایز بین امضای نقطه برش و عبارت نقطه برش روشن شود. مثال زیر نقطه‌ای به نام «anyOldTransfer» را تعریف می‌کند که با اجرای هر روشی به نام «transfer» مطابقت دارد:

@Pointcut("execution(\* transfer(..))")*// the pointcut expression*

**private** **void** anyOldTransfer() {}*// the pointcut signature*

عبارت pointcut که مقدار حاشیه‌نویسی @Pointcut را تشکیل می‌دهد یک عبارت معمولی AspectJ 5 pointcut است. برای بحث کامل در مورد زبان پوینت‌کات AspectJ، به راهنمای برنامه‌نویسی AspectJ (و برای برنامه‌های افزودنی، دفترچه برنامه‌نویسان AspectJ 5) یا یکی از کتاب‌های مربوط به AspectJ مانند "Eclipse AspectJ" توسط Colyer et مراجعه کنید. al. یا "AspectJ in Action" اثر Ramnivas Laddad.

##### **Supported Pointcut Designators**

Spring AOP از نشانگرهای نقطه‌بر AspectJ (PCD) زیر برای استفاده در عبارات نقطه‌بردار پشتیبانی می‌کند:

Other pointcut types

زبان نقطه برش کامل AspectJ از تعیین کننده های نقطه برش اضافی پشتیبانی می کند که در بهار پشتیبانی نمی شوند. اینها عبارتند از: call، get، set، preinitialization، staticinialization، مقداردهی اولیه، handler، consultexecution، insidecode، cflow، cflowbelow، if، @this و @withincode. استفاده از این نشانگرهای نقطه‌بر در عبارات نقطه‌بر تفسیر شده توسط Spring AOP منجر به ایجاد یک IllegalArgumentException می‌شود.

مجموعه ای از تعیین کننده های نقطه ای پشتیبانی شده توسط Spring AOP ممکن است در نسخه های بعدی برای پشتیبانی بیشتر از تعیین کننده های نقطه ای AspectJ گسترش یابد.

execution - برای تطبیق دادن روش اجرای نقاط اتصال، این نشانگر نقطه‌بر اصلی است که هنگام کار با Spring AOP استفاده می‌کنید.

within - تطابق برای اتصال به نقاط در انواع خاصی (به سادگی اجرای یک روش اعلام شده در یک نوع تطبیق هنگام استفاده از Spring AOP)

this - تطابق با نقاط اتصال را محدود می کند (اجرای روش ها هنگام استفاده از Spring AOP) که در آن مرجع bean (پراکسی Spring AOP) نمونه ای از نوع داده شده است.

target - تطابق با نقاط اتصال را محدود می کند (اجرای روش ها در هنگام استفاده از Spring AOP) که در آن شی هدف (شیء برنامه در حال پروکسی) نمونه ای از نوع داده شده است.

args - محدودیت های تطبیق با نقاط اتصال (اجرای روش ها هنگام استفاده از Spring AOP) که در آن آرگومان ها نمونه هایی از انواع داده شده هستند.

@target - تطابق با نقاط اتصال را محدود می‌کند (اجرای روش‌ها هنگام استفاده از Spring AOP) که در آن کلاس شیء اجراکننده دارای حاشیه‌نویسی از نوع داده شده است.

@args - محدودیت‌های تطبیق برای نقاط اتصال (اجرای روش‌ها هنگام استفاده از Spring AOP) که در آن نوع زمان اجرا آرگومان‌های واقعی ارسال شده دارای حاشیه‌نویسی از نوع(های) معین است.

@within - محدود کردن تطابق برای اتصال به نقاط در انواعی که دارای حاشیه‌نویسی هستند (اجرای روش‌هایی که در انواع با حاشیه‌نویسی داده شده در هنگام استفاده از Spring AOP اعلام شده است)

@annotation - تطابق را برای نقاطی که موضوع نقطه اتصال (روش در حال اجرا در Spring AOP) دارای حاشیه‌نویسی داده شده است، محدود می‌کند.

از آنجایی که Spring AOP تطبیق را فقط به نقاط اتصال اجرای روش محدود می کند، بحث در مورد تعیین کننده های نقطه برش در بالا، تعریف محدودتری نسبت به آنچه در راهنمای برنامه نویسی AspectJ می بینید، ارائه می دهد. علاوه بر این، AspectJ خود دارای معناشناسی مبتنی بر نوع است و در یک نقطه اتصال اجرا، هم این و هم هدف به یک شی ارجاع می‌دهند - شیئی که متد را اجرا می‌کند. Spring AOP یک سیستم مبتنی بر پروکسی است و بین خود شی پراکسی (که به آن متصل است) و شی هدف پشت پراکسی (محصول به هدف) تفاوت قائل می شود.

به دلیل ماهیت مبتنی بر پروکسی چارچوب AOP Spring، فراخوانی‌های درون شیء هدف طبق تعریف قطع نمی‌شوند. برای پراکسی‌های JDK، فقط فراخوانی‌های متد رابط عمومی روی پراکسی قابل رهگیری هستند. با CGLIB، فراخوانی‌های متد عمومی و محافظت‌شده روی پراکسی رهگیری می‌شوند، و در صورت لزوم، حتی روش‌های قابل مشاهده بسته‌ها. با این حال، تعاملات مشترک از طریق پروکسی ها باید همیشه از طریق امضای عمومی طراحی شود.

توجه داشته باشید که تعاریف نقطه برش عموماً با هر روش رهگیری مطابقت دارند. اگر نقطه برش صرفاً عمومی باشد، حتی در یک سناریوی پروکسی CGLIB با تعاملات غیرعمومی بالقوه از طریق پراکسی ها، باید بر این اساس تعریف شود.

اگر نیازهای رهگیری شما شامل فراخوانی متد یا حتی سازنده در کلاس هدف است، به جای فریم ورک AOP مبتنی بر پروکسی Spring، از بافت AspectJ بومی Spring-driven استفاده کنید. این یک حالت متفاوت استفاده از AOP با ویژگی‌های مختلف را تشکیل می‌دهد، بنابراین قبل از تصمیم‌گیری، ابتدا با بافتن آشنا شوید.

bean(idOrNameOfBean)

توکن idOrNameOfBean می تواند نام هر Spring bean باشد: پشتیبانی از حروف عام محدود با استفاده از کاراکتر \* ارائه شده است، بنابراین اگر برخی از قراردادهای نامگذاری را برای Spring bean خود ایجاد کنید، می توانید به راحتی یک عبارت PCD bean برای انتخاب آنها بنویسید. همانطور که در مورد دیگر تعیین کننده های نقطه ای وجود دارد، PCD لوبیا را می توان &&'ed، ||'ed، و ! (نفی) نیز.

لطفاً توجه داشته باشید که PCD لوبیا فقط در Spring AOP پشتیبانی می شود - و نه در بافت بومی AspectJ. این یک پسوند خاص Spring برای PCD های استاندارد است که AspectJ تعریف می کند و بنابراین برای جنبه های اعلام شده در مدل @Aspect در دسترس نیست.

PCD لوبیا در سطح نمونه (بر اساس مفهوم نام لوبیا بهار) به جای فقط در سطح نوع (که AOP مبتنی بر بافندگی به آن محدود می شود) عمل می کند. تعیین‌کننده‌های نقطه‌بر مبتنی بر نمونه قابلیت ویژه چارچوب AOP مبتنی بر پروکسی Spring و ادغام نزدیک آن با کارخانه Spring bean است، جایی که شناسایی دانه‌های خاص با نام طبیعی و ساده است.

##### **Combining pointcut expressions**

عبارات برش نقطه ای را می توان با استفاده از '&&'، '||' ترکیب کرد. و '!'. همچنین می توان به عبارات نقطه برش با نام اشاره کرد. مثال زیر سه عبارت pointcut را نشان می‌دهد: anyPublicOperation (که در صورتی مطابقت دارد که یک روش اجرای متد اجرای هر متد عمومی را نشان دهد). inTrading (که با اجرای یک روش در ماژول معاملاتی مطابقت دارد)، و tradingOperation (که در صورتی مطابقت دارد که اجرای یک روش نشان دهنده هر روش عمومی در ماژول معاملاتی باشد).

@Pointcut("execution(public \* \*(..))")

**private** **void** anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..\*)")

**private** **void** inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")

**private** **void** tradingOperation() {}

ساخت عبارات نقطه برش پیچیده تر از اجزای نامگذاری شده کوچکتر همانطور که در بالا نشان داده شده است بهترین روش است. هنگام اشاره به نقاط بر اساس نام، قوانین دید معمولی جاوا اعمال می شود (شما می توانید نقطه برش های خصوصی را در همان نوع، نقاط برش محافظت شده در سلسله مراتب، نقاط برش عمومی در هر کجا و غیره مشاهده کنید). قابلیت مشاهده بر تطابق برش نقطه تأثیری ندارد.

##### **Sharing common pointcut definitions**

هنگام کار با برنامه های سازمانی، اغلب می خواهید از چند جنبه به ماژول های برنامه و مجموعه های خاصی از عملیات رجوع کنید. توصیه می‌کنیم یک جنبه «SystemArchitecture» تعریف کنید که عبارات نقطه‌بردار رایج را برای این منظور می‌گیرد. یک چنین جنبه معمولی به صورت زیر است:

**package** com.xyz.someapp;

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Pointcut;

@Aspect

**public** **class** **SystemArchitecture** {

*/\*\**

*\* A join point is in the web layer if the method is defined*

*\* in a type in the com.xyz.someapp.web package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.web..\*)")

**public** **void** inWebLayer() {}

*/\*\**

*\* A join point is in the service layer if the method is defined*

*\* in a type in the com.xyz.someapp.service package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.service..\*)")

**public** **void** inServiceLayer() {}

*/\*\**

*\* A join point is in the data access layer if the method is defined*

*\* in a type in the com.xyz.someapp.dao package or any sub-package*

*\* under that.*

*\*/*

@Pointcut("within(com.xyz.someapp.dao..\*)")

**public** **void** inDataAccessLayer() {}

*/\*\**

*\* A business service is the execution of any method defined on a service*

*\* interface. This definition assumes that interfaces are placed in the*

*\* "service" package, and that implementation types are in sub-packages.*

*\**

*\* If you group service interfaces by functional area (for example,*

*\* in packages com.xyz.someapp.abc.service and com.xyz.someapp.def.service) then*

*\* the pointcut expression "execution(\* com.xyz.someapp..service.\*.\*(..))"*

*\* could be used instead.*

*\**

*\* Alternatively, you can write the expression using the 'bean'*

*\* PCD, like so "bean(\*Service)". (This assumes that you have*

*\* named your Spring service beans in a consistent fashion.)*

*\*/*

@Pointcut("execution(\* com.xyz.someapp..service.\*.\*(..))")

**public** **void** businessService() {}

*/\*\**

*\* A data access operation is the execution of any method defined on a*

*\* dao interface. This definition assumes that interfaces are placed in the*

*\* "dao" package, and that implementation types are in sub-packages.*

*\*/*

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

**public** **void** dataAccessOperation() {}

}

نقطه‌برهایی که در چنین جنبه‌ای تعریف شده‌اند، می‌توانند به هر جایی که نیاز به عبارت نقطه‌بری داشته باشید، ارجاع داده شوند. به عنوان مثال، برای تراکنشی کردن لایه سرویس، می توانید بنویسید:

<aop:config>

<aop:advisor

pointcut="com.xyz.someapp.SystemArchitecture.businessService()"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

عناصر <aop:config> و <aop:advisor> در پشتیبانی AOP مبتنی بر طرحواره مورد بحث قرار می گیرند. عناصر تراکنش در مدیریت تراکنش مورد بحث قرار می گیرند.

##### **Examples**

کاربران Spring AOP احتمالاً بیشتر از execution pointcut استفاده می کنند. فرمت یک عبارت اجرایی به صورت زیر است:

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)

**throws**-pattern?)

همه قسمت ها به جز الگوی نوع برگشتی (الگوی ret-type در قطعه بالا)، الگوی نام و الگوی پارامترها اختیاری هستند. الگوی نوع برگشتی تعیین می کند که نوع برگشتی روش باید چه باشد تا یک نقطه اتصال مطابقت داشته باشد. اغلب شما از \* به عنوان الگوی نوع برگشتی استفاده می کنید که با هر نوع برگشتی مطابقت دارد. یک نام نوع کاملاً واجد شرایط تنها زمانی مطابقت دارد که متد نوع داده شده را برگرداند. الگوی نام با نام روش مطابقت دارد. می توانید از علامت \* به عنوان تمام یا بخشی از یک الگوی نام استفاده کنید. اگر یک الگوی نوع اعلان را مشخص می کنید، یک دنباله اضافه کنید. برای پیوستن آن به مولفه الگوی نام. الگوی پارامترها کمی پیچیده‌تر است: () با روشی مطابقت دارد که هیچ پارامتری ندارد، در حالی که (..) با هر تعداد پارامتر (صفر یا بیشتر) مطابقت دارد. الگوی (\*) با روشی مطابقت دارد که یک پارامتر از هر نوع را دریافت می کند، (\*,String) با روشی که دو پارامتر دارد مطابقت دارد، اولی می تواند از هر نوع باشد، دومی باید یک رشته باشد. برای اطلاعات بیشتر به بخش معناشناسی زبان در راهنمای برنامه نویسی AspectJ مراجعه کنید.

چند نمونه از عبارات pointcut رایج در زیر آورده شده است.

• اجرای هر روش public:

execution(**public** \* \*(..))

• اجرای هر متد با نامی که با "set" شروع می شود:

execution(\* set\*(..))

• اجرای هر روشی که توسط رابط AccountService تعریف شده است:

execution(\* com.xyz.service.AccountService.\*(..))

• اجرای هر روشی که در پکیج service تعریف شده است:

execution(\* com.xyz.service.\*.\*(..))

• اجرای هر روشی که در پکیج سرویس یا یک sub-package تعریف شده است:

execution(\* com.xyz.service..\*.\*(..))

• هر join point (اجرای روش فقط در Spring AOP) در بسته سرویس:

within(com.xyz.service.\*)

• هر join point (اجرای روش فقط در Spring AOP) در بسته سرویس یا sub-package:

within(com.xyz.service..\*)

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که در آن پراکسی رابط AccountService را پیاده سازی می کند:

this(com.xyz.service.AccountService)

"this" بیشتر به صورت الزام آور استفاده می شود: - بخش زیر را در مورد توصیه برای نحوه در دسترس قرار دادن شی پراکسی در بدنه مشاوره ببینید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که در آن شی هدف رابط AccountService را پیاده سازی می کند:

target(com.xyz.service.AccountService)

"target" بیشتر به صورت الزام آور استفاده می شود: - به بخش زیر در مورد مشاوره برای نحوه در دسترس قرار دادن شی هدف در بدنه مشاوره مراجعه کنید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که یک پارامتر واحد را می گیرد و آرگومان ارسال شده در زمان اجرا Serializable است:

args(java.io.Serializable)

"args" بیشتر به صورت الزام آور استفاده می شود: - بخش زیر را در مورد توصیه برای نحوه در دسترس قرار دادن آرگومان های روش در بدنه مشاوره ببینید.

توجه داشته باشید که نقطه برش داده شده در این مثال با execution متفاوت است (\* \*(java.io.Serializable)): اگر آرگومان ارسال شده در زمان اجرا Serializable باشد، نسخه args مطابقت دارد، اگر امضای متد یک پارامتر واحد را اعلام کند نسخه اجرا مطابقت دارد. Serializable را تایپ کنید

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که در آن شیء هدف دارای حاشیه نویسی @Transactional است:

@target(org.springframework.transaction.annotation.Transactional)

'@target' همچنین می‌تواند به صورت الزام‌آور استفاده شود: - بخش زیر را در مورد توصیه‌ها برای نحوه در دسترس قرار دادن شی حاشیه‌نویسی در متن مشاوره ببینید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که در آن نوع اعلام شده شی هدف دارای حاشیه نویسی @Transactional است:

@within(org.springframework.transaction.annotation.Transactional)

«@within» همچنین می‌تواند به صورت الزام‌آور استفاده شود: - بخش زیر در مورد توصیه‌ها را برای نحوه در دسترس قرار دادن شی حاشیه‌نویسی در متن مشاوره ببینید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که در آن روش اجرا دارای حاشیه نویسی @Transactional است:

@annotation(org.springframework.transaction.annotation.Transactional)

'@ annotation' همچنین می تواند به صورت الزام آور استفاده شود: - بخش زیر را در مورد توصیه برای نحوه در دسترس قرار دادن شی حاشیه نویسی در متن مشاوره ببینید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) که یک پارامتر واحد را می گیرد و در آن نوع زمان اجرا آرگومان ارسال شده دارای حاشیه نویسی Classified @ است:

@args(com.xyz.security.Classified)

«@args» را می‌توان به صورت الزام‌آور نیز استفاده کرد: - بخش زیر را در مورد توصیه‌ها برای نحوه در دسترس قرار دادن شی (های) حاشیه‌نویسی در بدنه مشاوره ببینید.

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) در یک Spring Bean به نام tradeService:

bean(tradeService)

• هر نقطه اتصال (اجرای روش فقط در Spring AOP) در Spring beans دارای نام هایی که با عبارت wildcard مطابقت دارد \*Service:

bean(\*Service)

##### **Writing good pointcuts**

در طول کامپایل، AspectJ نقاط برش را پردازش می کند تا عملکرد مطابقت را بهینه کند. بررسی کد و تعیین اینکه آیا هر نقطه اتصال (به صورت ایستا یا پویا) با یک نقطه برش داده شده مطابقت دارد یا خیر، فرآیند پرهزینه ای است. (تطابق پویا به این معنی است که مطابقت را نمی توان به طور کامل از تجزیه و تحلیل استاتیک تعیین کرد و آزمایشی در کد قرار می گیرد تا مشخص شود که آیا مطابقت واقعی در هنگام اجرای کد وجود دارد یا خیر). در اولین برخورد با یک اعلان نقطه برش، AspectJ آن را به شکلی بهینه برای فرآیند تطبیق بازنویسی می کند. این یعنی چی؟ اساسا نقاط برش در DNF بازنویسی می شوند (فرم عادی جدایی) و اجزای نقطه برش به گونه ای مرتب می شوند که ابتدا آن دسته از مؤلفه هایی که ارزیابی ارزان تر هستند بررسی شوند. این بدان معناست که شما نیازی به نگرانی در مورد درک عملکرد تعیین کننده های مختلف نقطه برش ندارید و ممکن است آنها را به هر ترتیبی در یک اعلامیه نقطه برش ارائه کنید.

با این حال، AspectJ فقط می‌تواند با آنچه گفته می‌شود کار کند، و برای عملکرد بهینه تطبیق، باید به آنچه که آنها در تلاش برای دستیابی هستند فکر کنید و فضای جستجوی مسابقات را تا حد امکان در تعریف محدود کنید. تعیین کننده های موجود به طور طبیعی در یکی از سه گروه قرار می گیرند: مهربان، محدوده و زمینه:

نشانگرهای مهربان آنهایی هستند که نوع خاصی از نقطه اتصال را انتخاب می کنند. به عنوان مثال: اجرا، دریافت، تنظیم، تماس، کنترل کننده

تعیین کننده های محدوده آنهایی هستند که گروهی از نقاط مورد علاقه (احتمالاً انواع مختلف) را انتخاب می کنند. به عنوان مثال: درون، درون کد

تعیین‌کننده‌های متنی آنهایی هستند که بر اساس زمینه مطابقت دارند (و به صورت اختیاری متصل می‌شوند). به عنوان مثال: این، هدف، @annotation

یک نقطه برش به خوبی نوشته شده باید سعی کند حداقل دو نوع اول (نوع و محدوده) را شامل شود، در حالی که اگر بخواهید بر اساس بافت نقطه اتصال مطابقت داشته باشد، تعیین‌کننده‌های متنی ممکن است گنجانده شوند، یا آن زمینه را برای استفاده در توصیه‌ها پیوند دهند. ارائه فقط یک نشانگر دلخواه یا فقط یک نشانگر متنی کارساز خواهد بود، اما می تواند بر عملکرد بافت (زمان و حافظه استفاده شده) به دلیل پردازش و تجزیه و تحلیل اضافی تأثیر بگذارد. تعیین‌کننده‌های محدوده برای تطبیق بسیار سریع هستند و استفاده از آنها به این معنی است که AspectJ می‌تواند به سرعت گروه‌هایی از نقاط پیوست را که نباید بیشتر پردازش شوند، رد می‌کند - به همین دلیل است که یک نقطه برش خوب همیشه باید در صورت امکان شامل یکی باشد.

Advice با یک عبارت pointcut همراه است و قبل، بعد یا در اطراف اجرای متد مطابق با pointcut اجرا می شود. عبارت pointcut ممکن است یک ارجاع ساده به یک pointcut نامگذاری شده باشد، یا یک عبارت pointcut اعلام شده در جای خود.

##### **Before advice**

قبل از اینکه توصیه در یک جنبه با استفاده از حاشیه نویسی @Before اعلام شود:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

@Aspect

**public** **class** **BeforeExample** {

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doAccessCheck() {

*// ...*

}

}

اگر از یک عبارت pointcut در محل استفاده می‌کنیم، می‌توانیم مثال بالا را به صورت زیر بازنویسی کنیم:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Before;

@Aspect

**public** **class** **BeforeExample** {

@Before("execution(\* com.xyz.myapp.dao.\*.\*(..))")

**public** **void** doAccessCheck() {

*// ...*

}

}

##### **After returning advice**

مشاوره پس از بازگشت زمانی اجرا می شود که اجرای متد منطبق به طور عادی برگردد. با استفاده از حاشیه نویسی @AfterReturning اعلام می شود:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doAccessCheck() {

*// ...*

}

}

نکته: البته امکان داشتن چندین اظهارنامه مشاوره و سایر اعضا نیز وجود دارد که همه در یک جنبه هستند. ما فقط یک بیانیه توصیه واحد را در این مثال ها نشان می دهیم تا روی موضوع مورد بحث در آن زمان تمرکز کنیم.

گاهی اوقات شما نیاز دارید که در بدنه مشاوره به مقدار واقعی که برگردانده شده است دسترسی داشته باشید. می توانید از فرم @AfterReturning استفاده کنید که مقدار بازگشتی را برای این پیوند می دهد:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterReturning;

@Aspect

**public** **class** **AfterReturningExample** {

@AfterReturning(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

returning="retVal")

**public** **void** doAccessCheck(Object retVal) {

*// ...*

}

}

نام مورد استفاده در ویژگی بازگشتی باید با نام پارامتر در روش مشاوره مطابقت داشته باشد. هنگامی که اجرای یک روش برمی گردد، مقدار بازگشتی به عنوان مقدار آرگومان مربوطه به متد مشاوره ارسال می شود. یک بند برگشتی همچنین تطبیق را فقط به اجرای متدهایی محدود می کند که مقداری از نوع مشخص شده را برمی گرداند (در این مورد Object که با هر مقدار بازگشتی مطابقت دارد).

لطفاً توجه داشته باشید که هنگام استفاده از مشاوره پس از بازگشت، امکان بازگشت یک مرجع کاملاً متفاوت وجود ندارد.

##### **After throwing advice**

پس از پرتاب توصیه زمانی اجرا می شود که اجرای متد منطبق با پرتاب یک استثنا خارج می شود. با استفاده از حاشیه نویسی @AfterThrowing اعلام شده است:

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterThrowing;

@Aspect

**public** **class** **AfterThrowingExample** {

@AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doRecoveryActions() {

*// ...*

}

}

اغلب شما می‌خواهید که مشاوره فقط زمانی اجرا شود که استثناهایی از یک نوع معین وجود داشته باشند، و همچنین اغلب نیاز به دسترسی به استثنای پرتاب شده در بدنه مشاوره دارید. برای محدود کردن تطابق (در صورت تمایل، از Throwable به عنوان نوع استثنا در غیر این صورت از Throwable استفاده کنید) و استثنای پرتاب شده را به یک پارامتر مشاوره متصل کنید، از ویژگی throwing استفاده کنید.

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.AfterThrowing;

@Aspect

**public** **class** **AfterThrowingExample** {

@AfterThrowing(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

throwing="ex")

**public** **void** doRecoveryActions(DataAccessException ex) {

*// ...*

}

}

نام مورد استفاده در ویژگی پرتاب باید با نام پارامتر در روش مشاوره مطابقت داشته باشد. هنگامی که اجرای یک متد با پرتاب یک استثنا خارج می شود، استثنا به عنوان مقدار آرگومان مربوطه به متد مشاوره منتقل می شود. یک عبارت پرتابی نیز تطبیق را فقط به اجرای متدهایی محدود می‌کند که استثنایی از نوع مشخص شده ایجاد می‌کنند (در این مورد DataAccessException).

##### **After (finally) advice**

پس از (در نهایت) مشاوره اجرا می شود، با این حال اجرای متد منطبق خارج می شود. با استفاده از حاشیه نویسی @After اعلام شده است. پس از مشاوره باید برای رسیدگی به شرایط بازگشت عادی و استثنایی آماده باشید. معمولاً برای انتشار منابع و غیره استفاده می شود.

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.After;

@Aspect

**public** **class** **AfterFinallyExample** {

@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

**public** **void** doReleaseLock() {

*// ...*

}

}

##### **Around advice**

نوع آخر نصیحت پیرامون نصیحت است. در اطراف مشاوره اجرا می شود "در اطراف" یک روش همسان اجرای. این فرصت را دارد که هم قبل و هم بعد از اجرای متد کار انجام دهد، و تعیین کند که چه زمانی، چگونه و حتی اگر واقعاً متد اجرا شود. در صورتی که نیاز به اشتراک گذاری حالت قبل و بعد از اجرای متد به روشی ایمن (برای مثال شروع و توقف تایمر) داشته باشید، اغلب از توصیه های Around استفاده می شود. همیشه از کم‌توان‌ترین شکل مشاوره‌ای که نیازهای شما را برآورده می‌کند استفاده کنید (یعنی از توصیه‌هایی که قبل از توصیه ساده هستند، استفاده نکنید).

مشاوره پیرامون با استفاده از حاشیه نویسی @Around اعلام می شود. پارامتر اول روش مشاوره باید از نوع ProceedingJoinPoint باشد. در متن توصیه‌ها، فراخوانی () progress در ProceedingJoinPoint باعث می‌شود که متد زیربنایی اجرا شود. متد ادامه را می‌توان انتقال در یک شی[] نیز نامید - از مقادیر موجود در آرایه به عنوان آرگومان‌های اجرای متد در هنگام ادامه استفاده می‌شود.

رفتار اقدام زمانی که با یک شی [] فراخوانی می شود کمی متفاوت از رفتار ادامه برای توصیه هایی است که توسط کامپایلر AspectJ کامپایل شده است. برای توصیه‌هایی که با استفاده از زبان سنتی AspectJ نوشته می‌شوند، تعداد آرگومان‌های ارسال شده برای ادامه باید با تعداد آرگومان‌های ارسال شده به توصیه اطراف (نه با تعداد آرگومان‌های گرفته‌شده توسط نقطه اتصال زیربنایی)، و مقدار ارسال شده برای ادامه در یک مطابقت داشته باشد. موقعیت آرگومان داده شده جایگزین مقدار اصلی در نقطه اتصال برای موجودیتی که مقدار به آن مقید شده است می شود (اگر این در حال حاضر منطقی نیست نگران نباشید!). رویکردی که اسپرینگ اتخاذ کرده است، ساده‌تر است و با معناشناسی مبتنی بر پروکسی و فقط اجرای آن مطابقت بهتری دارد. فقط در صورتی باید از این تفاوت آگاه باشید که جنبه های @AspectJ نوشته شده برای Spring را کامپایل می کنید و از ادامه با آرگومان های با کامپایلر و بافنده AspectJ استفاده می کنید. راهی برای نوشتن چنین جنبه‌هایی وجود دارد که 100% با Spring AOP و AspectJ سازگار است، و این در بخش بعدی در مورد پارامترهای مشاوره مورد بحث قرار گرفته است.

**import** org.aspectj.lang.annotation.Aspect;

**import** org.aspectj.lang.annotation.Around;

**import** org.aspectj.lang.ProceedingJoinPoint;

@Aspect

**public** **class** **AroundExample** {

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

**public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {

*// start stopwatch*

Object retVal = pjp.proceed();

*// stop stopwatch*

**return** retVal;

}

}

مقدار برگشتی توسط اطلاعات اطراف، مقدار بازگشتی خواهد بود که توسط فراخوان متد مشاهده می شود. به عنوان مثال، یک جنبه ذخیره سازی ساده می تواند یک مقدار را از یک کش در صورت داشتن یک مقدار برگرداند، و در صورت عدم وجود، produce () را فراخوانی کند. توجه داشته باشید که ادامه ممکن است یک بار، چندین بار، یا اصلاً در بدنه توصیه های اطراف فراخوانی نشود، همه اینها کاملاً قانونی هستند.

##### **Advice parameters**

Spring توصیه های کاملاً تایپ شده ارائه می دهد - به این معنی که شما پارامترهای مورد نیاز خود را در امضای advice اعلام می کنید (همانطور که برای مثال های برگشتی و پرتابی در بالا دیدیم) به جای اینکه همیشه با آرایه های Object[] کار کنید. ما خواهیم دید که چگونه استدلال و سایر ارزش های زمینه ای را در یک لحظه در دسترس نهاد مشاوره قرار دهیم. ابتدا بیایید نگاهی به نحوه نوشتن توصیه های عمومی بیندازیم که می تواند در مورد روشی که مشاوره در حال حاضر توصیه می کند مطلع شود.

###### **Access to the current JoinPoint**

Spring توصیه های کاملاً تایپ شده ارائه می دهد - به این معنی که شما پارامترهای مورد نیاز خود را در امضای مشاوره اعلام می کنید (همانطور که برای مثال های برگشتی و پرتابی در بالا دیدیم) به جای اینکه همیشه با آرایه های Object[] کار کنید. ما خواهیم دید که چگونه استدلال و سایر ارزش های زمینه ای را در یک لحظه در دسترس نهاد مشاوره قرار دهیم. ابتدا بیایید نگاهی به نحوه نوشتن توصیه های عمومی بیندازیم که می تواند در مورد روشی که مشاوره در حال حاضر توصیه می کند مطلع شود.

###### **Passing parameters to advice**

قبلاً نحوه اتصال مقدار بازگشتی یا مقدار استثنا را دیده‌ایم (استفاده از پس از بازگشت و پس از پرتاب توصیه). برای در دسترس قرار دادن مقادیر آرگومان برای بدن مشاوره، می توانید از شکل binding args استفاده کنید. اگر یک نام پارامتر به جای نام نوع در عبارت args استفاده شود، آنگاه مقدار آرگومان مربوطه به عنوان مقدار پارامتر هنگام فراخوانی توصیه ارسال می شود. یک مثال باید این را واضح تر کند. فرض کنید می‌خواهید اجرای عملیات dao را که یک آبجکت Account را به عنوان پارامتر اول در نظر می‌گیرند، توصیه کنید و نیاز به دسترسی به حساب در بدنه مشاوره دارید. می توانید موارد زیر را بنویسید:

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**public** **void** validateAccount(Account account) {

*// ...*

}

بخش args(account,..) از عبارت pointcut دو هدف را دنبال می کند: اولاً، تطبیق را فقط به اجرای متدهایی که متد حداقل یک پارامتر را در آن می گیرد، محدود می کند و آرگومان ارسال شده به آن پارامتر، نمونه ای از Account است. ثانیاً، شیء حساب واقعی را از طریق پارامتر حساب در اختیار مشاوره قرار می دهد.

راه دیگر نوشتن این است که یک نقطه برش را اعلام کنید که وقتی با یک نقطه اتصال مطابقت دارد، مقدار شی Account را "ارائه" می کند، و سپس فقط به نقطه برش نامگذاری شده از توصیه مراجعه کنید. این به شکل زیر خواهد بود:

@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**private** **void** accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")

**public** **void** validateAccount(Account account) {

*// ...*

}

خواننده علاقه مند یک بار دیگر برای جزئیات بیشتر به راهنمای برنامه نویسی AspectJ مراجعه می کند.

شیء پراکسی (این)، شیء هدف (هدف)، و حاشیه‌نویسی‌ها (@within، @target، @annotation، @args) همگی می‌توانند به روشی مشابه محدود شوند. مثال زیر نشان می دهد که چگونه می توانید اجرای روش های حاشیه نویسی شده را با حاشیه نویسی @Auditable مطابقت دهید و کد حسابرسی را استخراج کنید.

ابتدا تعریف حاشیه نویسی @Auditable:

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @interface Auditable {

AuditCode value();

}

و سپس توصیه ای که با اجرای روش های @Auditable مطابقت دارد:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")

**public** **void** audit(Auditable auditable) {

AuditCode code = auditable.value();

*// ...*

}

###### **Advice parameters and generics**

Spring AOP می تواند ژنریک های مورد استفاده در اعلان های کلاس و پارامترهای متد را مدیریت کند. فرض کنید یک نوع عمومی مانند زیر دارید:

**public** **interface** **Sample**<T> {

**void** sampleGenericMethod(T param);

**void** sampleGenericCollectionMethod(Collection<T> param);

}

شما می توانید رهگیری انواع روش را به انواع پارامترهای خاصی محدود کنید، به سادگی با تایپ پارامتر توصیه به نوع پارامتری که می خواهید روش را رهگیری کنید:

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(MyType param) {

*// Advice implementation*

}

همانطور که قبلاً در بالا به آن پرداختیم، این کار کاملاً واضح است. با این حال، شایان ذکر است که این برای مجموعه‌های عمومی کار نخواهد کرد. بنابراین شما نمی توانید یک نقطه برش را به این شکل تعریف کنید:

@Before("execution(\* ..Sample+.sampleGenericCollectionMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(Collection<MyType> param) {

*// Advice implementation*

}

برای انجام این کار، باید همه عناصر مجموعه را بررسی کنیم، که معقول نیست، زیرا ما همچنین نمی‌توانیم تصمیم بگیریم که به طور کلی چگونه با مقادیر تهی رفتار کنیم. برای دستیابی به چیزی شبیه به این، باید پارامتر را در Collection<?> تایپ کنید و به صورت دستی نوع عناصر را بررسی کنید.

###### **Determining argument names**

اتصال پارامتر در فراخوان‌های مشاوره به تطبیق نام‌های مورد استفاده در عبارات نقطه‌بر با نام پارامترهای اعلام‌شده در امضاهای متد (مشاوره و نقطه‌برش) متکی است. نام پارامترها از طریق بازتاب جاوا در دسترس نیستند، بنابراین Spring AOP از استراتژی های زیر برای تعیین نام پارامترها استفاده می کند:

اگر نام پارامترها توسط کاربر به صراحت مشخص شده باشد، نام پارامترهای مشخص شده استفاده می شود: هر دو حاشیه نویسی مشاوره و نقطه برش دارای یک ویژگی اختیاری "argNames" هستند که می تواند برای تعیین نام آرگومان های متد مشروح استفاده شود - این آرگومان ها نام ها در زمان اجرا در دسترس هستند. مثلا:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

**public** **void** audit(Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code and bean*

}

اگر اولین پارامتر از نوع JoinPoint، ProceedingJoinPoint یا JoinPoint.StaticPart باشد، می‌توانید نام پارامتر را از مقدار ویژگی "argNames" حذف کنید. برای مثال، اگر توصیه قبلی را برای دریافت شیء نقطه اتصال تغییر دهید، ویژگی "argNames" لازم نیست آن را شامل شود:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

**public** **void** audit(JoinPoint jp, Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code, bean, and jp*

}

رفتار ویژه ای که به پارامتر اول انواع JoinPoint، ProceedingJoinPoint و JoinPoint.StaticPart داده می شود، مخصوصاً برای توصیه هایی که هیچ زمینه نقطه اتصال دیگری را جمع آوری نمی کنند، راحت است. در چنین شرایطی، ممکن است به سادگی ویژگی "argNames" را حذف کنید. به عنوان مثال، توصیه زیر نیازی به اعلان ویژگی "argNames" ندارد:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")

**public** **void** audit(JoinPoint jp) {

*// ... use jp*

}

استفاده از ویژگی 'argNames' کمی ناشیانه است، بنابراین اگر ویژگی 'argNames' مشخص نشده باشد، Spring AOP به اطلاعات اشکال زدایی کلاس نگاه می کند و سعی می کند نام پارامترها را از جدول متغیر محلی تعیین کند. این اطلاعات تا زمانی وجود خواهد داشت که کلاس ها با اطلاعات اشکال زدایی (حداقل '-g:vars') کامپایل شده باشند. پیامدهای کامپایل با این پرچم روشن عبارتند از: (1) درک کد شما کمی آسان تر خواهد بود (مهندس معکوس)، (2) اندازه فایل های کلاس بسیار کمی بزرگتر خواهد بود (معمولا بی اهمیت)، (3) بهینه سازی برای حذف متغیرهای محلی استفاده نشده توسط کامپایلر شما اعمال نمی شوند. به عبارت دیگر، در ساختن با این پرچم نباید با هیچ مشکلی مواجه شوید.

اگر یک جنبه @AspectJ توسط کامپایلر AspectJ (ajc) حتی بدون اطلاعات اشکال زدایی کامپایل شده باشد، دیگر نیازی به افزودن ویژگی argNames نیست زیرا کامپایلر اطلاعات مورد نیاز را حفظ می کند.

اگر کد بدون اطلاعات اشکال زدایی لازم کامپایل شده باشد، آنگاه Spring AOP سعی خواهد کرد جفت شدن متغیرهای اتصال به پارامترها را استنتاج کند (به عنوان مثال، اگر فقط یک متغیر در عبارت pointcut محدود شده باشد، و روش مشاوره فقط یک پارامتر را می گیرد، جفت شدن واضح است!). اگر اتصال متغیرها با توجه به اطلاعات موجود مبهم باشد، یک AmbiguousBindingException پرتاب می شود.

اگر همه استراتژی های بالا شکست بخورند، یک IllegalArgumentException پرتاب می شود.

###### **Proceeding with arguments**

قبلاً اشاره کردیم که نحوه نوشتن یک فراخوانی ادامه با آرگومان هایی که به طور مداوم در Spring AOP و AspectJ کار می کند را شرح می دهیم. راه حل این است که اطمینان حاصل شود که امضای مشاوره هر یک از پارامترهای روش را به ترتیب متصل می کند. مثلا:

@Around("execution(List<Account> find\*(..)) && " +

"com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +

"args(accountHolderNamePattern)")

**public** Object preProcessQueryPattern(ProceedingJoinPoint pjp,

String accountHolderNamePattern) **throws** Throwable {

String newPattern = preProcess(accountHolderNamePattern);

**return** pjp.proceed(**new** Object**[]** {newPattern});

}

در بسیاری از موارد به هر حال این کار را انجام خواهید داد (مانند مثال بالا).

##### **Advice ordering**

چه اتفاقی می‌افتد وقتی چندین توصیه همه بخواهند در یک نقطه اتصال اجرا شوند؟ Spring AOP از همان قوانین تقدم AspectJ برای تعیین ترتیب اجرای مشاوره پیروی می کند. توصیه بالاترین اولویت ابتدا "در راه ورود" اجرا می شود (بنابراین با توجه به دو توصیه قبل، یکی با بالاترین اولویت اول اجرا می شود). "در راه خروج" از نقطه اتصال، بالاترین اولویت توصیه آخرین اجرا می شود (بنابراین با توجه به دو توصیه پس از آن، یکی با بالاترین اولویت دوم خواهد بود).

وقتی دو توصیه در جنبه‌های مختلف تعریف می‌شوند، هر دو باید در یک نقطه اتصال اجرا شوند، مگر اینکه در غیر این صورت ترتیب اجرا را مشخص نکرده باشید. با تعیین اولویت می توانید ترتیب اجرا را کنترل کنید. این به روش عادی Spring یا با پیاده سازی رابط org.springframework.core.Ordered در کلاس جنبه یا حاشیه نویسی آن با حاشیه نویسی Order انجام می شود. با توجه به دو جنبه، جنبه ای که مقدار کمتری را از Ordered.getOrder() برمی گرداند (یا مقدار حاشیه نویسی) اولویت بیشتری دارد.

وقتی دو توصیه تعریف شده در یک جنبه، هر دو باید در یک نقطه اتصال اجرا شوند، ترتیب تعریف نشده است (زیرا هیچ راهی برای بازیابی دستور اعلامیه از طریق بازتاب برای کلاس‌های کامپایل شده با جاواک وجود ندارد). در نظر بگیرید که چنین روش‌های مشاوره را به یک روش مشاوره در هر نقطه اتصال در هر کلاس جنبه تقسیم کنید، یا این توصیه‌ها را به کلاس‌های جنبه جداگانه تبدیل کنید - که می‌توانند در سطح جنبه سفارش داده شوند.

#### **5.2.5. Introductions**