

目录(5.2.8 版)

参考文档的这一部分涵盖了 Spring 框架绝对必要的所有技术.....	1
1. IoC 容器.....	2
1.1. Spring IoC 容器和 Beans 的介绍.....	2
1.2. 容器概述.....	2
1.2.1. 配置元数据.....	3
1.2.2. 实例化一个容器.....	5
1.2.3. 使用容器.....	8
1.3. Bean 的概述.....	10
1.3.1. 命名 beans.....	11
1.3.2. 实例化 Beans.....	13
1.4. 依赖.....	18
1.4.1. 依赖注入.....	18
1.4.2. 依赖和配置详情.....	29
1.4.3. 使用 depends-on.....	39
1.4.4. 懒初始化 Beans.....	40
1.4.5. 自动装配协作者.....	41
1.4.6. 方法注入.....	43
1.5. Bean 的作用域.....	51
1.5.1. 单例作用域.....	52
1.5.2. 原型作用域.....	53
1.5.3. 有原型 bean 依赖的单例 Bean.....	54

1.5.4. 请求、会话、应用程序和 WebSocket 作用域.....	54
1.5.5. 自定义作用域.....	60
1.6. 自定义 Bean 的性质.....	63
1.6.1. 生命周期回调.....	63
1.6.2. ApplicationContextAware 和 BeanNameAware 接口.	74
1.6.3. 其他 Aware 接口.....	75
1.7. Bean 定义继承.....	76
1.8. 容器扩展点.....	78
1.8.1. 通过使用 BeanPostProcessor 自定义 Beans.....	78
1.8.2. 使用 BeanFactoryPostProcessor 自定义配置元数据	82
1.8.3. 使用 FactoryBean 自定义实例化逻辑.....	86
1.9. 基于注解的容器配置.....	87
1.9.1. @Required 注解.....	88
1.9.2. 使用@Autowired.....	89
1.9.3. 使用 @Primary 基于注解的自动装配的调整.....	98
1.9.4. 使用 Qualifiers 基于注解的自动装配的调整.....	99
1.9.5. 使用泛型作为自动装配限定符.....	109
1.9.6. 使用 CustomAutoWireConfigurer.....	111
1.9.7. 使用@Resource 注入.....	112
1.9.8. 使用@Value.....	114
1.9.9. 使用@PostConstruct 和@PreDestroy.....	118
1.10. 类路径扫描和托管组件.....	119

1.10.1. @Component 以及更多的原型注释.....	120
1.10.2. 使用元注解和合成注解.....	120
1.10.3. 自动扫描类和注册 Bean 定义.....	123
1.10.4. 使用过滤器自定义扫描.....	125
1.10.5. 在组件中定义 Bean 元数据.....	127
1.10.6. 命名自动检测组件.....	131
1.10.7. 为自动扫描的组件提供一个作用域.....	133
1.10.8. 使用组件提供限定符元数据.....	135
1.10.9. 生成候选组件的索引.....	136
1.11. 使用 JSR 330 标准注解.....	137
1.11.1. 使用@Inject 和@Named 进行依赖注入.....	138
1.11.2. @Named 和@ManagedBean 和@Component 标准等效注解.....	141
1.11.3. JSR-330 标准注解的局限.....	143
1.12. 基于 Java 的容器配置.....	144
1.12.1. 基本概念:@Bean 和@Configuration.....	145
1.12.2. 通过使用 AnnotationConfigApplicationContext 实例化 Spring 容器.....	146
1.12.3. 使用@Bean 注解.....	151
1.12.4. 使用@Configuration 注解.....	161
1.12.5. 构建基于 Java 的配置.....	166
1.13. 环境抽象概念.....	183

1.13.1. Bean 定义文件.....	183
1.13.2. PropertySource 抽象.....	192
1.13.3. 使用 @PropertySource.....	194
1.13.4. 语句中占位符解析.....	196
1.14. 注册一个 LoadTimeWeaver.....	197
1.15. ApplicationContext 的其他功能.....	198
1.15.1. Internationalization using MessageSource....	198
1.15.2. 标准和自定义事件.....	202
1.15.3. 获取低级资源的简便方法.....	212
1.15.4. 对于 Web 应用的简便应用上下文的实例化.....	213
1.15.5. 将一个 Spring 应用上下文部署成一个 Java EE RAR 文件.....	213
1.16. Bean 工厂	214
1.16.1. BeanFactory 还是 ApplicationContext?.....	215
2. 资源.....	218
2.1. 介绍.....	218
2.2. 资源接口.....	218
2.3. 内置资源实现.....	220
2.3.1. UriResource.....	221
2.3.2. ClassPathResource.....	221
2.3.3. FileSystemResource.....	221
2.3.4. ServletContextResource.....	222

2. 3. 5. InputStreamResource.....	222
2. 3. 6. ByteArrayResource.....	222
2. 4. ResourceLoader.....	222
2. 5. ResourceLoaderAware 接口.....	224
2. 6. 资源依赖.....	225
2. 7. 应用程序上下文和资源路径.....	225
2. 7. 1. 构造应用上下文.....	225
2. 7. 2. 应用程序上下文构造函数资源路径中的通配符.....	227
2. 7. 3. FileSystemResource 注意事项.....	230
3. 验证，数据绑定和类型转换.....	232
3. 1. 通过 Spring 的 Validator 接口进行验证.....	232
3. 2. 将代码解析为错误消息.....	236
3. 3. Bean 操作和 BeanWrapper.....	237
3. 3. 1. 设置和获取基本和内嵌的属性.....	237
3. 3. 2. 构建 PropertyEditor 实现.....	240
3. 4. Spring 类型转换.....	248
3. 4. 1. Converter API.....	248
3. 4. 2. 使用ConverterFactory.....	250
3. 4. 3. 使用 GenericConverter.....	251
3. 4. 4. ConversionService API.....	253
3. 4. 5. 配置一个 ConversionService.....	254
3. 4. 6. 编程式使用 ConversionService.....	255

3.5. Spring 字段格式化.....	256
3.5.1. Formatter SPI.....	257
3.5.2. 注解驱动格式化.....	260
3.5.3. FormatterRegistry SPI.....	263
3.5.4. FormatterRegistrar SPI.....	264
3.5.5. 在 Spring MVC 中配置格式化.....	265
3.6. 配置一个全局日期时间格式.....	265
3.7. Java Bean 验证.....	268
3.7.1. bean 验证概述.....	268
3.7.2. 配置一个 bean 验证 Provider.....	269
3.7.3. 配置一个 DataBinder.....	273
3.7.4. Spring MVC 3 验证.....	274
4. Spring 表达式语言 (SpEL)	275
4.1. 求值.....	276
4.1.1. 理解求值上下文.....	279
4.1.2. 解析器配置.....	281
4.1.3. SpEL 并发.....	282
4.2. bean 定义中的表达式.....	285
4.2.1. XML 配置.....	285
4.2.2. 注解配置.....	286
4.3. 语言引用.....	288
4.3.1. 文字表达式.....	289

4.3.2. 属性, 数组, 列表, Maps 和索引器.....	290
4.3.3. 内联列表.....	292
4.3.4. 内联 Maps.....	293
4.3.5. 数组结构.....	293
4.3.6. 方法.....	294
4.3.7. 操作符.....	295
4.3.8. 类型.....	301
4.3.9. 构造器.....	301
4.3.10. 变量.....	302
4.3.11. 函数.....	304
4.3.12. Bean 引用.....	305
4.3.13. 三元操作符.....	306
4.3.14. Elvis 操作符.....	307
4.3.15. 安全导航操作符.....	309
4.3.16. 集合选择器.....	309
4.3.17. 集合的投射.....	310
4.3.18. 表达式模板.....	311
4.4. 例子中的类使用.....	312
5. Spring 的面向切面编程 (AOP)	318
5.1. AOP 概念.....	318
5.2. Spring AOP 能力和目标.....	320
5.3. AOP 代理.....	321

5. 4. @AspectJ 支持.....	321
5. 4. 1. 启动@AspectJ 支持.....	322
5. 4. 2. 声明一个 Aspect.....	322
5. 4. 3. 声明一个切入点.....	324
5. 4. 4. 声明通知.....	334
5. 4. 5. 引介.....	349
5. 4. 6. Aspect 实例化模型.....	350
5. 4. 7. 一个 AOP 例子.....	351
5. 5. 基于 Schema 的 AOP 支持.....	354
5. 5. 1. 声明一个 Aspect.....	355
5. 5. 2. 声明一个切入点.....	355
5. 5. 3. 声明通知.....	358
5. 5. 4. 引介.....	366
5. 5. 5. Aspect 实例化模型.....	368
5. 5. 6. 一般切面.....	368
5. 5. 7. 一个 AOP schema 例子.....	368
5. 6. 选择 AOP 声明风格.....	372
5. 6. 1. Spring AOP 还是全 AspectJ?	372
5. 6. 2. @AspectJ 还是 XML, 哪个和 Spring AOP 更配?	372
5. 7. 混合 Aspect 类型.....	373
5. 8. 代理机制.....	374
5. 8. 1. 理解 AOP 代理.....	374

5. 9. 编程式@AspectJ 代理的创建.....	379
5. 10. 使用 Spring 应用的 AspectJ.....	379
5. 10. 1. 对 Spring 的依赖域对象使用 AspectJ.....	380
5. 10. 2. 配合 AspectJ 的其他 Spring 切面.....	384
5. 10. 3. 通过使用 Spring IOC 配置 AspectJ 切面.....	385
5. 10. 4. 在 Spring Framework 中使用 AspectJ 进行加载时编 织.....	386
5. 11. 更多资源.....	397
6. Spring AOP APIs.....	398
6. 1. Spring 里的 Pointcut(切入点) API.....	398
6. 1. 1. 概念.....	398
6. 1. 2. 切入点上的操作.....	400
6. 1. 3. AspectJ 表达式切入点.....	400
6. 1. 4. 简易切入点实现.....	400
6. 1. 5. 切入点超类.....	402
6. 1. 6. 自定义切入点.....	402
6. 2. Spring 里的 Advice(通知) API.....	402
6. 2. 1. 通知生命周期.....	403
6. 2. 2. Spring 中的通知类型.....	403
6. 3. Spring 里的 Advisor(一般切面) API.....	414
6. 4. 使用 ProxyFactoryBean(代理工厂 bean) 去创建 AOP 代理	414
6. 4. 1. 基础.....	414

6. 4. 2. JavaBean 属性.....	415
6. 4. 3. 基于 JDK 和基于 CGLIB 的代理.....	416
6. 4. 4. Proxying 接口.....	417
6. 4. 5. Proxyin 代理类.....	419
6. 4. 6. 使用“全局”一般切面.....	420
6. 5. 简洁代理定义.....	420
6. 6. 使用 ProxyFactory 编程式创建 AOP 代理.....	422
6. 7. 操作 Advised 对象.....	422
6. 8. 使用“自动代理”功能.....	425
6. 8. 1. Auto-proxy bean 定义.....	426
6. 9. 使用 TargetSource 实现.....	428
6. 9. 1. 可热插拔的目标源.....	428
6. 9. 2. 池化目标源.....	429
6. 9. 3. 原型目标源.....	431
6. 9. 4. ThreadLocal 目标源.....	431
6. 10. 定义新的通知类型.....	432
7. Null-safety（空值-安全性）.....	433
7. 1. 用例.....	433
7. 2. JSR-305 元注解.....	433
8. 数据缓冲区和编解码器.....	434
8. 1. DataBufferFactory.....	434
8. 2. DataBuffer.....	434

8. 3. PolledDataUtils.....	435
8. 4. DataBufferUtils.....	435
8. 5. Codecs.....	435
8. 6. 使用 DataBuffer.....	436
9. 附录.....	438
9. 1. XML 结构.....	438
9. 1. 1. util 结构.....	438
9. 1. 2. aop 结构.....	446
9. 1. 3. context 结构.....	447
9. 1. 4. beans 结构.....	448
9. 2. XML 架构创作.....	449
9. 2. 1. 编写 Schema.....	450
9. 2. 2. 编写一个 NamespaceHandler.....	451
9. 2. 3. 使用 BeanDefinitionParser.....	452
9. 2. 4. 注册 Handler 和 Schema.....	454
9. 2. 5. 使用 Custom Extension 在你的 Spring XML 配置文件	455
9. 2. 6. 更多详细示例.....	456

参考文档的这一部分涵盖了 Spring 框架绝对必要的所有技术

其中最重要的是 Spring 框架的控制反转（IoC）容器。对 Spring 框架的 IoC 容器进行彻底的处理之后，将全面介绍 Spring 的面向方面的编程（AOP）技术。Spring 框架具有自己的 AOP 框架，该框架在概念上易于理解，并且成功解决了 Java 企业编程中 AOP 要求的 80% 的难题。

还提供了 Spring 与 AspectJ 的集成（目前，在功能上最丰富）以及 Java 企业领域中最成熟的 AOP 实现）。

1. IoC 容器

这一章节涵盖了 Spring 的控制反转容器

1.1. Spring IoC 容器和 Beans 的介绍

这一章涵盖了 Spring Framework IoC 原理的实现。IoC 以依赖注入而闻名。借以对对象定义他们的依赖(也就是那些他们一起工作的其他类)仅仅是通过给工厂方法构造器参数,参数或者那些在它们从工厂方法被构造或返回后设置在对象实例上的属性。当容器创造这些 Bean 的时候接下来会注入那些依赖。这个过程从根本上反转(因此叫控制反转)了 Bean 本身通过直接类构造器或者注入服务定位器模式机制来控制实例化或它的依赖。

`org.springframework.beans` 和 `org.springframework.context` 包是 Spring Framework 的 IoC 容器的基础。`BeanFactory` 接口提供了一个管理任何类型的高级的配置机制能力。`ApplicationContext` 是 `BeanFactory` 的一个子接口。它增加了:

- 与 Spring AOP 特性更简单地集成
- 消息源处理(为了在国际化中使用)
- 事件发布
- 应用层特定上下文,比如用在 Web 应用中的 `WebApplicationContext`

简单来说, `BeanFactory` 提供了配置框架和基础功能性, `ApplicationContext` 增加了更多企业特定功能。`ApplicationContext` 是一个 `BeanFactory` 的超集,并仅仅使用在本章 Spring IoC 容器的描述中。想获取更多的 `BeanFactory` 使用信息而不是 `ApplicationContext` 的,看 `BeanFactory`。

在 Spring 中,那些构成你应用主体的对象和那些被 Spring IoC 容器管理的对象被称为 beans。一个 bean 是一个被实例化组装好的并被 Spring IoC 容器管理对象。反之,一个 bean 只是你应用程序众多对象中的一个。Beans 和依赖,被反射在一个容器所使用的配置元数据中。

1.2. 容器概述

`org.springframework.context.ApplicationContext` 接口代表了 Spring IoC 容器的能力,它负责实例化,配置,组装这些 beans。容器通过读取配置元数据来获取有关要实例化,配置和组装哪些对象的指令。配置元数据表现为 XML, Java 注解或者 Java 代码。

它使你能够表达组成你的应用程序的对象以及这些对象之间的丰富相互依赖关系。

若干种 `ApplicationContext` 接口的实现由 Spring 实现。在独立的应用程序中，创建一个 `ClassPathXMLApplicationContext` 或者 `FileSystemXMLApplicationContext` 的实例更为普遍。当 XML 已经传统定义了配置元数据时，你可以通过提供一小部分 XML 配置指示容器将 Java 注释或代码用作元数据格式去以声明方式启用对这些其他元数据格式的支持。

在大部分应用场景中，实例化一个 Spring IoC 容器的一个或多个实例不需要显式用户代码。例如，在一个 Web 应用程序场景中，应用程序的 `web.xml` 文件中简单的八行(约)样板 Web 描述符 XML 文件通常就足够了（见 [Convenient ApplicationContext Instantiation for Web Applications](#)）。如果使用 [Spring Tools for Eclipse](#)，则只需单击几下鼠标或击键即可轻松创建此样板配置。

下图展示了一个关于 Spring 是如何工作的高级视图。你的应用程序类与配置元数据结合在了一起以便在 `ApplicationContext` 被创建和初始化之后，你有一个完成的配置好的可执行的系统或者应用。

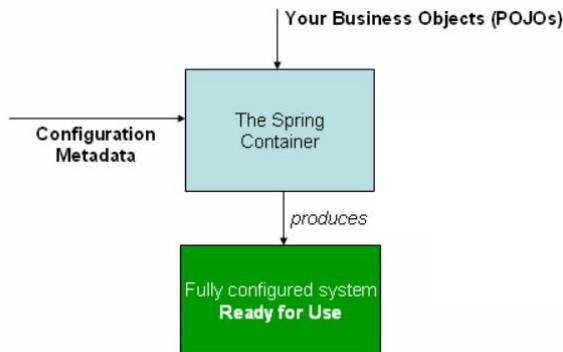


Figure 1. The Spring IoC container

1.2.1. 配置元数据

如前图所示，Spring IoC 容器接收一个配置元数据的形式。这个配置元数据表示，你，作为应用程序的开发者，告诉 Spring 容器是如何实例化，配置，装配你应用程序中的对象的。

一般来说，配置元数据以简单直观的 XML 格式提供，这是本章大部分内容用来传达 Spring IoC 容器的关键概念和功能的内容。

基于 XML 的元数据不是配置元数据的唯一允许形式。Spring IoC 容器本身与实际写入此配置元数据的格式完全脱钩。如今，许多开发人员为他们的 Spring 应用程序选择基于 Java 的配置。

有关在 Spring 容器中使用其他形式的元数据的信息，请看：

- [基于注解的配置](#): Spring 2.5 介绍了对于基于注解的元数据配置支持。
- [基于 Java 的配置](#): 从 Spring 3.0 开始许多由 Spring JavaConfig 提供的新特性变成了 Spring 框架的核心部分。因此你可以使用 Java 为你的应用程序类定义外部 bean 来替代 XML 文件。为了使用这些新特性，请看[@Configuration](#), [@Bean](#), [@Import](#), and [@DependsOn](#) 注解。

Spring 配置由至少一个，通常更多地 bean 定义组成。这些 bean 必须由 Spring 容器管理。基于 XML 的配置元数据将这些 bean 配置为顶级`<beans/>`元素内的`<bean/>`元素。Java 配置通常在[@Configuration](#) 类中使用[@Bean](#) 注解的方法。

这些 bean 的定义对应了那些实际上组成你应用程序对象。通常，你定义的服务层对象，数据访问对象，表示对象比如 Struts 的 Action 实例，基础架构对象比如 Hibernate SessionFactories, JMS Queues，以此类推。通常，不会在容器中配置细粒度的域对象，因为 DAO 和业务逻辑通常负责创建和加载域对象。然而，你可以使用集成了 AspectJ 的 Spring 去配置那些已经在 IoC 容器控制之外创建的对象。见 [Using AspectJ to dependency-inject domain object with Spring](#).

下面的例子展示了基于 XML 的元数据配置的基本结构：

```
<?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="..."> ① ②
        <!-- 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>
```

① `id` 属性是标识单个 bean 定义的字符串

② `class` 属性定义 bean 的类型并使用完全限定的类名。

`id` 属性的值是指协作对象。在此示例中未显示用于引用协作对象的 XML。有关更多信息，请参见[依赖项 \(dependencies\)](#)。

1.2.2. 实例化一个容器

提供给 `ApplicationContext` 构造函数的位置路径是资源字符串，它们让容器从各种外部资源（例如本地）加载配置元数据比如本地文件系统，Java `CLASSPATH` 等。

Java

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

Kotlin

```
val context = ClassPathXmlApplicationContext("services.xml", "daos.xml")
```

在你学习完关于 Spring IoC 容器之后，你或许想知道更多关于 Spring 的资源抽象（如[资源](#)中所述），它提供了一种方便的机制去读取一个输入流格式的以 URI 句法定义的位置[资源](#)。特别是[如应用程序上下文和资源路径](#)中所述，资源路径用于构造应用程序上下文。

下面的例子展示了服务层对象(`service.xml`)的配置文件：

```
<?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>
```

下面的例子展示了数据访问层 `daos.xml` 对象文件：

```

<?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>

```

在上面的例子中，服务层包括了 `PetStoreServiceImpl` 类和两个数据访问层类 `JpaAccountDao` 和 `JpaItemDao`(基于 JPA 对象关系映射标准)。属性名称(`property name`)元素引用 JavaBean 属性的名称，而 `ref` 元素引用另一个 bean 定义的名称。`id` 和 `ref` 元素之间的这种联系表示了协作对象之间的依赖性。对于配置一个对象的依赖的详细信息，见依赖(`Dependencies`)。

构建基于 XML 的配置元数据

使 bean 定义跨多个 XML 文件将会很有用。经常性的，在你架构中的每一个独立的 XML 配置文件表示一个逻辑层或者模块。

你可以使用应用上下文构造器去从所有的 XML 片中读取 bean 定义。如[上一节](#)中所示，此构造函数具有多个资源(`Resource`)位置。或者，使用`<import />`元素的一个或多个实例从另一个文件加载 bean 定义。以下示例显示了如何执行此操作：

```

<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>

```

在上述例子中，外部 bean 定义从三个文件中被读取到了：`service.xml`,`messageSource.xml` 和 `themeSource.xml`.所有位置路径都相对于定义文

件进行导入，因此 `services.xml` 必须与进行导入的文件位于同一目录或类路径位置，而 `messageSource.xml` 和 `themeSource.xml` 必须位于该路径下方的资源位置导入文件。如你所见，斜杠被忽略。但是，鉴于这些路径是相对的，最好不要使用任何斜线。根据 Spring Schema，导入的文件的内容（包括顶级`<beans/>`元素）必须是有效的 XML bean 定义。

可以但不建议使用相对的“`../`”路径引用父目录中的文件。这样做会创建对当前应用程序外部文件的依赖关系。特别是，不建议对 `classpath: URL`（例如，`classpath: ../ services.xml`）使用此引用，在 URL 中，运行时解析过程会选择“最近的”类路径根目录，然后查看其父目录。类路径配置的更改可能导致选择其他错误的目录。



你始终可以使用完全限定的资源位置来代替相对路径：例如，文件：`C:/config/services.xml` 或类路径：`/config/services.xml`。但是请注意，您正在将应用程序的配置耦合到特定的绝对位置。通常，最好为这样的绝对位置保留一个间接寻址-例如，通过在运行时针对 JVM 系统属性解析的“`$ {...}`”占位符。

命名空间本身提供了导入指令功能。Spring 所提供的一系列 XML 名称空间（例如，上下文和 util 名称空间）中提供了超出普通 bean 定义的其他配置功能。

Groovy Bean 定义 DSL

作为外部化配置元数据的另一个示例，Bean 定义也可以在 Spring 的 Groovy Bean 定义 DSL 中表达，这是从 Grails 框架中得知的。通常，这种配置位于“`.groovy`”文件中，其结构如以下示例所示：

```

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
        }
    }
}

```

这种配置样式在很大程度上等同于 XML Bean 定义，甚至支持 Spring 的 XML 配置名称空间。它还允许通过 `importBeans` 指令导入 XML bean 定义文件。

1.2.3. 使用容器

`ApplicationContext` 是高级工厂的接口，该工厂能够维护不同 bean 及其依赖关系的注册表。通过使用方法 `T getBean (String name, Class <T> requiredType)`，可以检索 bean 的实例。

使用 `ApplicationContext` 可以读取 bean 定义并访问它们，如以下示例所示：

Java

```

// 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();

```

Kotlin

```
import org.springframework.beans.factory.getBean  
  
// create and configure beans  
val context = ClassPathXmlApplicationContext("services.xml", "daos.xml")  
  
// retrieve configured instance  
val service = context.getBean<PetStoreService>("petStore")  
  
// use configured instance  
var userList = service.getUsernameList()
```

使用 Groovy 配置，引导看起来非常相似。 它有一个不同的上下文实现类，该类可识别 Groovy（但也了解 XML Bean 定义）。 以下示例展示了 Groovy 配置：

Java

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

Kotlin

```
val context = GenericGroovyApplicationContext("services.groovy", "daos.groovy")
```

最灵活的变体是 `GenericApplicationContext` 与读取器委托结合使用，例如，与 XML 文件的 `XmlBeanDefinitionReader` 结合使用，如以下示例所示：

Java

```
GenericApplicationContext context = new GenericApplicationContext();  
new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");  
context.refresh();
```

Kotlin

```
val context = GenericApplicationContext()  
XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml")  
context.refresh()
```

您还可以将 `GroovyBeanDefinitionReader` 用于 Groovy 文件，如以下示例所示：

Java

```
GenericApplicationContext context = new GenericApplicationContext();
new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",
"daos.groovy");
context.refresh();
```

Kotlin

```
val context = GenericApplicationContext()
GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",
"daos.groovy")
context.refresh()
```

您可以在同一 `ApplicationContext` 上混合和匹配此类阅读器委托，从不同的配置源中读取 bean 定义。

然后可以使用 `getBean` 检索 bean 的实例。`ApplicationContext` 接口有检索 beans 的一系列方法，但是，要我说，你的应用程序不该用它。确实，您的应用程序代码应该根本不调用 `getBean()` 方法，因此完全不依赖 Spring API。例如，Spring 与 Web 框架的集成的各种 Web 框架组件（例如控制器和 JSF 托管的 Bean）提供了依赖项注入，使您可以声明一个通过元数据（例如自动装配注释）对特定 bean 的依赖。

1.3. Bean 的概述

一个 Spring IoC 容器管理着一个或多个 beans。这些 beans 通过你提供给容器的配置元数据创建（比如 XML 中`<bean/>`标签定义）。

在容器内部，这些 bean 定义被表示为 `BeanDefinition` 对象，它包含了如下元数据：

- 包限定的类名：通常，定义了 Bean 的实际实现类
- Bean 行为配置元素，用于声明 Bean 在容器中的行为（作用域，生命周期回调等）。
- 引用其他 bean 进行其工作所需的 bean。这些引用也称为协作者或依赖项。
- 要在新创建的对象中设置的其他配置设置，例如，池的大小限制或在管理连接池的 bean 中使用的连接数。

该元数据转换为构成每个 bean 定义的一组属性。下表描述了这些属性：

Property	Explained in...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

除了包含有关如何创建特定 bean 的信息的 bean 定义之外，`ApplicationContext` 实现还允许注册在容器外部（由用户）创建的现有对象。这将通过 `ApplicationContext` 的 `BeanFactory` 的 `getBeanFactory()` 方法完成，它返回 `BeanFactory` 的 `DefaultListableBeanFactory` 实现。`DefaultListableBeanFactory` 通过以下方式支持此注册：`registerSingleton(...)` 和 `registerBeanDefinition(...)` 方法。但是，典型的应用程序只能与通过常规 bean 定义元数据定义的 bean 一起使用。



Bean 元数据和手动提供的单例实例需要尽早注册，以便容器在自动装配和其他自省步骤中正确地理解/解释它们。虽然在某种程度上支持覆盖现有元数据和现有单例实例，但是在运行时（与对工厂的实时访问同时）对新 bean 的注册不被正式支持，并且可能导致并发访问异常，bean 容器中的状态不一致或全部。

1.3.1. 命名 beans

每个 bean 具有一个或多个标识符。这些标识符在承载 Bean 的容器内必须唯一。一个 bean 同行只有一个标识符，然而如果它确实有超过一个的时候，另外的一个我们可以考虑当成别名。

在基于 XML 的配置元数据上，你使用 `id` 属性，`name` 属性，或者两者皆用能够特指 bean 的标识符。`id` 属性是你能够特指确定的一个 `id`。简便点，这些命名是单词数字一类 ('myBean', 'someService' 等)但是它们也会包含特殊字符。如果你想给这些 bean 加点别名，你也可以给他们制定一个 `name` 属性，由逗号，分号，空格隔开。作为历史记录，在 Spring 3.1 之前的版本中，`id` 属性定义为 `xsd:ID` 类型，该类型限制了可能的字符。自 3.1 之后它被定义为 `xsd:String` 类型。请注意，bean ID 唯一性仍由容器强制执行，尽管

不再由 XML 解析器执行。

你不再被强制提供一个 `name` 或者 `id` 给一个 bean。如果你不明确的提供一个 `name` 或者 `id` 属性的话容器将会自动生成一个唯一名字给这个 bean 然而，如果你想通过名字 `name` 参考一个 bean 的话，比如 `ref` 元素的使用或者一个服务定位风格查找表，你必须提供一个名字 `name`。不提供名称的动机与使用内部 bean 和自动装配合作者有关。

Bean 命名约定

约定是在命名 bean 时将标准 Java 约定用于实例字段名称。也就是说，bean 名称以小写字母开头，并用驼峰式大小写。此类名称的示例包括 `accountManager`, `accountService`, `userDao`, `LoginController` 等。

一致地命名 Bean 使您的配置更易于阅读和理解。另外，如果您使用 Spring AOP，则在将建议应用于按名称相关的一组 bean 时，它会很有帮助。

通过在类路径中进行组件扫描，Spring 会按照前面描述的规则为未命名的组件生成 Bean 名称：本质上，采用简单的类名称并将其初始字符转换为小写。但是，在（不寻常的）特殊情况下，如果有多个字符并且第一个和第二个字符均为大写字母，则会保留原始大小写。这些规则与 `java.beans.Introspector.decapitalize` (Spring 在此使用) 定义的规则相同。

在 Bean 的定义之外使用别名

在 bean 定义本身中，你可以通过使用 `id` 属性指定的最多一个名称和 `name` 属性中任意数量的其他名称的组合来为 bean 提供多个名称。这些名称可以是同一个 bean 的等效别名，并且在某些情况下很有用，例如，通过使用特定于该组件本身的 bean 名称，让应用程序中的每个组件都引用一个公共依赖项。但是，在实际定义 bean 的地方指定所有别名并不总是足够的。有时需要为在别处定义的 bean 引入别名。在大型系统中通常是这种情况，在大型系统中，配置在每个子系统之间分配，每个子系统都有自己的对象定义集。在基于 XML 的配置元数据中，可以使用`<alias/>`元素来完成此操作。以下示例显示了如何执行此操作：

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

在这个例子中，一个叫 `fromName` 的 bean (在同一个容器中) 在定义了别名之后也可能被参考成 `toName`。

举例说明，子系统 A 的配置元数据可能被参考为一个叫 `subsystemA-datasource` 的

DataSource。子系统 B 的配置元数据可能被参考为一个叫 `subsystemB-dataSource` 的 DataSource。组成使用这两个子系统的主应用程序时，主应用程序通过 `myApp-dataSource` 的名称引用数据源。要使所有三个名称都引用相同的对象，可以将以下别名定义添加到配置元数据中：

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

现在每个组件和主应用程序可以使用唯一的一个 name 去引用数据源了，并且保证不与任何其他定义冲突（有效地创建名称空间），但它们引用的是同一 bean。

Java-配置

如果使用 Java configuration，则 `@Bean` 注解可用于提供别名。有关详细信息，请参见[使用@Bean注解](#)。

1.3.2. 实例化 Beans

Bean 定义实质上是创建一个或多个对象的方法。当被调用时，容器将查看命名 bean 的食谱（recipe?），并使用该 bean 定义封装的配置元数据来创建（或获取）实际对象。

如果使用基于 XML 的配置元数据，则在 `<bean/>` 元素的 `class` 属性中指定要实例化的对象的类型（或类）。此类属性（在内部是 BeanDefinition 实例的 Class 属性）通常 是必需的。（有关例外，请参阅[使用实例工厂方法实例化\(Instantiation by Using an Instance Factory Method\)](#) 和 Bean 定义继承([Bean Definition Inheritance](#))）可以通过以下两种方式之一使用 Class 属性：

- 通常，在容器本身通过反射性地调用其构造函数直接创建 Bean 的情况下，指定要构造的 Bean 类，这在某种程度上等同于使用 new 运算符的 Java 代码。
- 要指定包含用于创建对象的静态工厂方法的实际类，在不太常见的情况下，容器将在类上调用静态工厂方法以创建 Bean。从静态工厂方法的调用返回的对象类型可以是同一类，也可以是完全不同的另一类。

内部类命名

如果你想为一个静态内部类命名的话，你不得不使用二进制名称。

例如，一个叫 `SomeThing` 的类名在 `com.example` 的包里，这个类还有一个叫 `OtherThing` 的内部类，它的 `class` 属性就会是 `com.example.SomeThing$OtherThing`。

请注意，名称中使用 \$ 字符将嵌套的类名与外部类名分开。

构造器实例化

当你使用构造器方法去创建一个 bean 时，所有普通的类都可以被 Spring 使用并与之兼容。也就是说，正在开发的类不需要实现任何特定的接口或以特定的方式进行编码。只需指定 bean 类就足够了。但是，根据您用于该特定 bean 的 IoC 的类型，您可能需要一个默认（空）构造函数。

Spring IoC 容器几乎可以管理您要管理的任何类。他不只限制在管理真正的 JavaBeans。大多数 Spring 用户更喜欢实际的只有一个默认构造函数的 JavaBeans 以及根据容器中的属性建模的适当的 setter 和 getter。您还可以在容器中拥有更多奇特的非 Bean 样式类。例如，您需要使用绝对不符合 JavaBean 规范的旧式连接池，Spring 也可以对其进行管理。

通过基于 XML 的配置元数据，你可以这么来定义你的 bean 类：

```
<bean id="exampleBean" class="examples.ExampleBean"/>  
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

对于有参构造机制（如果需要）和在对象被构建之后设置对象实例属性的详情，请看注入依赖([Injecting Dependencies](#))

静态工厂实例化

当你通过一个静态工厂方法创建了一个 bean 的时候，使用 `class` 属性能够指定一个包含静态工厂方法和一个叫做 `factory-method` 的属性的类去指定一个工厂方法本身的名字。你可以去调用它（可以传参稍后会讲）并返回一个活动的对象，随后将其视为已通过构造函数创建。这种 bean 定义的一种用法是在旧版代码中调用静态工厂。

以下 bean 定义指定通过调用工厂方法来创建 bean。该定义不指定返回对象的类型（类），而仅指定包含工厂方法的类。在此示例中，`createInstance()`方法必须是静态方法。以下示例显示如何指定工厂方法：

```
<bean id="clientService"  
      class="examples.ClientService"  
      factory-method="createInstance"/>
```

下面的例子展示了一个和上面的 bean 定义一起工作的类：

Java

```
public class ClientService {  
    private static ClientService clientService = new ClientService();  
    private ClientService() {}  
  
    public static ClientService createInstance() {  
        return clientService;  
    }  
}
```

Kotlin

```
class ClientService private constructor() {  
    companion object {  
        private val clientService = ClientService()  
        fun createInstance() = clientService  
    }  
}
```

有关为工厂方法提供（可选）参数并在从工厂返回对象后设置对象实例属性的机制的详细信息，请参见详细的依赖关系和配置 [Dependencies and Configuration in Detail](#)。

使用实例工厂方法实例化

和通过[静态工厂方法实例化](#)类似的，使用实例工厂方法实例化会从容器中调用现有 bean 的非静态方法来创建新 bean。为了使用这个机制，设置 **class** 属性为空，在 **factory-bean** 属性，指定一个在当前容器（或父容器或祖先容器）中类的名字，它包含了一个将要创建这个对象的实例方法。设置工厂方法本身的名字在 **factory-method** 属性中。

下面的这个例子展示了如何配置这么一个 bean：

```
<!-- 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"/>
```

下面的例子展示了对应的类：

Java

```
public class DefaultServiceLocator {  
  
    private static ClientService clientService = new ClientServiceImpl();  
  
    public ClientService createClientServiceInstance() {  
        return clientService;  
    }  
}
```

Kotlin

```
class DefaultServiceLocator {  
    companion object {  
        private val clientService = ClientServiceImpl()  
    }  
    fun createClientServiceInstance(): ClientService {  
        return clientService  
    }  
}
```

一个工厂类也能够拥有不止一个的工厂方法，就像下面这个例子：

```
<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"/>
```

下面的例子展示了对应的类：

Java

```
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;  
    }  
}
```

Kotlin

```
class DefaultServiceLocator {  
    companion object {  
        private val clientService = ClientServiceImpl()  
        private val accountService = AccountServiceImpl()  
    }  
  
    fun createClientServiceInstance(): ClientService {  
        return clientService  
    }  
  
    fun createAccountServiceInstance(): AccountService {  
        return accountService  
    }  
}
```

这种方法表明，工厂 Bean 本身可以通过依赖项注入（DI）进行管理和配置。详细信息，请参见依赖性和配置 [Dependencies and Configuration in Detail](#)。



在 Spring 文档中，“factory bean”引用为一个被配置在 Spring 容器内的，能够通过一个[实例](#)或者[静态工厂方法](#)创建对象的一个 bean。相比之下，[FactoryBean](#)（请注意大小写）是指特定于 Spring 的 [FactoryBean](#) 实现类。

确定 Bean 的运行时类型

确定特定 bean 的运行时类型并非易事。Bean 元数据定义中的指定类只是初始类引用，可能与声明的工厂方法结合使用，或者是 [FactoryBean](#) 类，这可能导致 Bean 的运行时类型不同，或者在实例级的工厂方法情况下完全不进行设置（通过指定的 [factory-bean](#) 名称解析）。此外，AOP 代理可以使用基于接口的代理包装 Bean 实例，而目标 Bean 的实际类型（仅是其实现的接口）的暴露程度有限。找出特定 bean 的实际运行时类型的推荐方法

是对指定 bean 名称的 `BeanFactory.getType` 调用。这考虑了上述所有情况，并返回了针对相同 bean 名称的 `BeanFactory.getBean` 调用将返回的对象的类型。

1.4. 依赖

一个经典的企业级应用不会包含一个单独的对象（或者 bean 在 Spring 用语中）。即使是最简单的应用程序，也有一些对象可以协同工作，以呈现最终用户视为一致的应用程序。下面这一节将解释如何从定义多个独立的 Bean 定义到实现对象协作以实现目标的完全实现的应用程序。

1.4.1. 依赖注入

依赖注入（DI）是一个过程，通过该过程，对象仅通过构造函数参数，工厂方法的参数或在构造或创建对象实例后在对象实例上设置的属性来从工厂方法定义其依赖关系（即，与它们一起工作的其他对象）。然后，容器在创建 bean 时注入那些依赖项。从根本上讲，此过程是通过使用类的直接构造或服务定位器模式来控制 bean 自身依赖关系的实例化或定位的 bean 本身的逆过程（因此称为 Control Inversion）。

使用 DI 原理，代码更简洁，当为对象提供依赖项时，解耦会更有效。该对象不查找其依赖项，也不知道依赖项的位置或类。结果，你的类变得更易于测试，尤其是当依赖项依赖于接口或抽象基类时，它们允许在单元测试中使用存根或模拟实现。

DI 存在两个主要变体：[基于构造函数的依赖注入](#) 和 [基于 Setter 的依赖注入](#)。

基于构造器的依赖注入

基于构造函数的 DI 是通过容器调用具有多个参数的构造函数来完成的，每个参数表示一个依赖项。调用带有特定参数的静态工厂方法来构造 Bean 几乎是等效的，并且本次讨论将构造函数和静态工厂方法的参数视为类似。下面的例子展示了一个仅能通过构造器注入进行依赖注入的类：

Java

```
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...  
}
```

Kotlin

```
// a constructor so that the Spring container can inject a MovieFinder  
class SimpleMovieLister(private val movieFinder: MovieFinder) {  
    // business logic that actually uses the injected MovieFinder is omitted...  
}
```

注意，该类没有什么特别的。 它是一个 POJO，不依赖于特定于容器的接口，基类或注解。

构造参数解析

构造函数参数解析匹配通过使用参数的类型进行。如果 Bean 定义的构造函数参数中不存在任何歧义，当 bean 正在被初始化的时候，bean 定义中构造器参数的顺序就是提供给适配的构造器的参数顺序。看看下面这个类：

Java

```
package x.y;  
  
public class ThingOne {  
  
    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {  
        // ...  
    }  
}
```

Kotlin

```
package x.y

class ThingOne(thingTwo: ThingTwo, thingThree: ThingThree)
```

假设 `ThingTwo` 和 `ThingThree` 类没有通过继承关联，则不存在潜在的歧义。因此，以下配置可以正常工作，并且您无需在`<constructor-arg/>`元素中显式指定构造函数参数索引或类型。

```
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

当引用另一个 bean 时，类型是已知的，并且可以发生匹配（与前面的示例一样）。当使用简单类型（例如`<value> true </ value>`）时，Spring 无法确定值的类型，因此在没有帮助的情况下无法按类型进行匹配。来看下面这个类：

Java

```
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;
    }
}
```

Kotlin

```
package examples

class ExampleBean(
    private val years: Int, // Number of years to calculate the Ultimate Answer
    private val ultimateAnswer: String// The Answer to Life, the Universe, and
    Everything
)
```

构造函数参数类型匹配

在上述情况下，如果通过使用 `type` 属性显式指定构造函数参数的类型，则容器可以使用简单类型的类型匹配。来看下面这个例子：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

构造器参数索引

您可以使用 `index` 属性来明确指定构造函数参数的索引，就像下面这个例子一样：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

除了解决多个简单值的歧义性之外，指定索引还可以解决歧义，其中构造函数具有两个相同类型的参数。



索引从 0 开始。

构造器参数名称

你也可以使用构造器参数名来消除值得歧义，来看下面这个例子：

```
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

请记住，要立即使用该功能，必须在启用调试标志的情况下编译代码，以便 Spring 可以从构造函数中查找参数名称。如果您不能或不想使用 `debug` 标志编译代码，则可以使用 `@ConstructorProperties` JDK 注释显式命名构造函数参数。然后，该示例类必须如下所

示：

Java

```
package examples;

public class ExampleBean {

    // Fields omitted

    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Kotlin

```
package examples

class ExampleBean
@ConstructorProperties("years", "ultimateAnswer")
constructor(val years: Int, val ultimateAnswer: String)
```

基于 Setter 的构造输入

通过使用无参数构造函数或无参数静态工厂方法实例化您的 bean 之后，容器通过在 bean 上调用 setter 方法来完成基于 setter 的 DI。

下面的示例显示只能通过使用纯 setter 注入来依赖注入的类。此类是常规的 Java。它是一个 POJO，不依赖于容器特定的接口，基类或注释。

Java

```
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...  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    // a late-initialized property so that the Spring container can inject a  
    // MovieFinder  
    lateinit var movieFinder: MovieFinder  
  
    // business logic that actually uses the injected MovieFinder is omitted...  
}
```

[ApplicationContext](#) 支持它管理的 bean 的基于构造函数和基于 `setter` 的依赖注入。在已经通过构造函数方法注入了某些依赖项之后，它还支持基于 `setter` 的依赖注入。您可以以 [BeanDefinition](#) 的形式配置依赖项，并与 [PropertyEditor](#) 实例结合使用以将属性从一种格式转换为另一种格式。但是，大多数 Spring 用户并不直接（即以编程方式）使用这些类，而是使用 XML bean 定义，带注解的组件（即以 @Component, @Controller 等进行注解的类）或 @Bean 方法基于 Java 配置类。然后将这些源在内部转换为 BeanDefinition 实例，并用于加载整个 Spring IoC 容器实例。

基于构造器还是基于 `setter` 进行依赖注入？

由于可以混合使用基于构造函数的 DI 和基于 `setter` 的 DI，因此将构造函数用于强制性依赖项并将 `setter` 方法或配置方法用于可选依赖性是一个很好的经验法则。注意，可以在 `setter` 方法上使用 `@Required` 批注，以使该属性成为必需的依赖项。但是，最好使用带有参数的程序验证的构造函数注入。

Spring 团队通常提倡构造函数注入，因为它可以让您将应用程序组件实现为不可变对象，并确保所需的依赖项不为 `null`。此外，注入构造函数的组件始终以完全初始化的状态返回到客户端（调用）代码。附带说明一下，大量的构造函数参数是一种不好的代码异味，这表明该类可能承担了太多的职责，应将其重构以更好地解决关注点分离问题。

`Setter` 注入主要应仅用于可以在类中分配合理的默认值的可选依赖项。否则，必须在代码使用依赖项的任何地方执行非空检查。`setter` 注入的一个好处是，`setter` 方法使该类的对象在以后可以重新配置或重新注入。因此，通过 `JMX MBean` 进行管理是用于 `setter` 注入的引人注目的用例。

使用 DI 的风格可以使一个指定的类具有最大意义。有时，在处理您没有源代码的第三方类时，将为您做出选择。例如，如果第三方类未公开任何 `setter` 方法，则构造函数注入可能是 DI 的唯一可用形式。

依赖解析过程

容器执行 bean 依赖项解析，如下所示：

使用描述所有 bean 的配置元数据创建和初始化 `ApplicationContext`。可以通过 XML，Java 代码或注释来指定配置元数据。

对于每个 bean，其依赖项都以属性，构造函数参数或 `static-factory` 方法的参数（如果使用它而不是常规构造函数）的形式表示。在实际创建 Bean 时，会将这些依赖项提供给 Bean。

每个属性或构造函数参数都是要设置的值的实际定义，或者是对容器中另一个 bean 的引用。

作为值的每个属性或构造函数参数都将从其指定的格式转换为该属性或构造函数参数的实际类型。默认情况下，Spring 可以将以字符串格式提供的值转换为所有内置类型，例如 `int`, `long`, `String`, `boolean` 等。

在创建容器时，Spring 容器会验证每个 bean 的配置。但是，在实际创建 Bean 之前，不会设置 Bean 属性本身。创建容器时，将创建具有单例作用域并设置为预先实例化（默认）的 Bean。作用域在 Bean 作用域中定义。否则，仅在请求时才创建 Bean。创建和分配 Bean

的依赖关系关系及其依赖的依赖（依此类推）时，创建 Bean 可能会导致创建一个 Bean 图。请注意，这些依赖项之间的解析不匹配可能会在后期出现，即在第一次创建受影响的 bean 时。

循环依赖

如果主要使用构造函数注入，则可能会创建无法解决的循环依赖方案。

例如：类 A 通过构造函数注入需要类 B 的实例，而类 B 通过构造函数注入获取类 A 的实例。如果您配置了将类 A 和 B 相互注入的 bean，Spring IoC 容器会在运行时检测到此循环引用，并抛出 `BeanCurrentlyInCreationException`。

一种可能的解决方案是编辑某些类的源代码，这些类的源代码由设置者而不是构造函数来配置。

或者，避免构造函数注入，而仅使用 `setter` 注入。换句话说，尽管不建议这样做，但是您可以使用 `setter` 注入配置循环依赖关系。

你可以完全相信 Spring 在做正确的事情。它在容器加载时检测配置问题，例如对不存在的 Bean 的引用和循环依赖项。在实际创建 Bean 时，Spring 设置属性并尽可能晚地解决依赖关系。这意味着如果创建对象或其依赖项之一存在问题，则已正确加载的 Spring 容器稍后可以在您请求对象时生成异常-例如，由于缺少属性或无效属性，bean 引发异常。这可能会延迟某些配置问题的可见性，这就是为什么默认情况下 `ApplicationContext` 实现会预先实例化单例 bean 的原因。在实际需要这些 bean 之前先花一些时间和内存来创建它们，您会在创建 `ApplicationContext` 时发现配置问题，而不是稍后。你仍然可以覆盖此默认行为，以便单例 bean 延迟初始化，而不是预先实例化。

如果不存在循环依赖关系，则在将一个或多个协作 Bean 注入从属 Bean 时，每个协作 Bean 都将被完全配置，然后再注入到依赖 Bean 中。这意味着，如果 bean A 依赖于 bean B，则在对 bean A 调用 `setter` 方法之前，Spring IoC 容器会完全配置 bean B。换句话说，实例化该 bean（如果它不是预先实例化的单例），则设置其依赖关系，并调用相关的生命周期方法（例如配置的 `init` 方法或 `InitializingBean` 回调方法）。

依赖注入的例子

以下示例将基于 XML 的配置元数据用于基于 `setter` 的依赖注入。一小部分的 Spring XML 配置文件指定了一些 bean 定义，如下所示：

```
<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"/>
```

以下示例显示了相应的 `ExampleBean` 类：

Java

```
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;
    }
}
```

Kotlin

```
class ExampleBean {
    lateinit var beanOne: AnotherBean
    lateinit var beanTwo: YetAnotherBean
    var i: Int = 0
}
```

在前面的示例中，声明了 `setter` 以与 XML 文件中指定的属性匹配。以下示例使用基于构造函数的依赖注入：

```
<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"/>
```

以下示例显示了相应的 `ExampleBean` 类：

Java

```
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;
    }
}
```

Kotlin

```
class ExampleBean(
    private val beanOne: AnotherBean,
    private val beanTwo: YetAnotherBean,
    private val i: Int)
```

`bean` 定义中指定的构造函数参数用作 `ExampleBean` 构造函数的参数

现在考虑该示例的一个变体，在该变体中，不是使用构造函数，而是告诉 Spring 调用静态工厂方法以返回对象的实例：

```

<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"/>

```

以下示例显示了相应的 `ExampleBean` 类：

Java

```

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;
    }
}

```

Kotlin

```

class ExampleBean private constructor() {
    companion object {
        // 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.
        fun createInstance(anotherBean: AnotherBean, yetAnotherBean: YetAnotherBean,
i: Int): ExampleBean {
            val eb = ExampleBean (...)
            // some other operations...
            return eb
        }
    }
}

```

静态工厂方法的参数由`<constructor-arg/>`元素提供，与实际使用构造函数的情况完全相同。`factory`方法返回的类的类型不必与包含静态工厂方法的类的类型相同（尽管在

此示例中他们是一样的）。实例（非静态）工厂方法可以以基本上相同的方式使用（除了使用 `factory-bean` 属性代替 `class` 属性之外），因此在此不讨论这些细节。

1.4.2. 依赖和配置详情

如上一节所述，您可以将 `bean` 属性和构造函数参数定义为对其他托管 `bean`（协作者）的引用或内联定义的值。Spring 的基于 XML 的配置元数据为此目的在其 `<property/>` 和 `<constructor-arg/>` 元素中支持子元素类型。

直接值（原语、字符串等）

`<property />` 元素的 `value` 属性将属性或构造函数参数指定为人类可读的字符串表示形式。Spring 的转换服务用于将这些值从字符串转换为属性或参数的实际类型。以下示例显示了设置的各种值：

```
<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="misterkaoli"/>
</bean>
```

以下示例将 `p`-命名空间用于更简洁的 XML 配置：

```
<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="misterkaoli"/>

</beans>
```

前面的 XML 更简洁。但是，除非在创建 `bean` 定义时使用支持自动属性完成的 IDE（例如 [IntelliJ IDEA](#) 或 [Eclipse 的 Spring Tools](#)），否则错字是在运行时而不是设计时发现的。强烈建议您使用此类 IDE 帮助。

您还可以配置 `java.util.Properties` 实例，如下所示：

```
<bean id="mappings"
      class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">

    <!-- 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>
```

Spring 容器通过使用 JavaBeans `PropertyEditor` 机制将`<value/>`元素内的文本转换为 `java.util.Properties` 实例。这是一个不错的捷径，并且是 Spring 团队偏爱使用嵌套的`<value/>`元素而不是 `value` 属性样式的几个地方之一。

idref 元素

`idref` 元素只是一种防错方法，可以将容器中另一个 bean 的 `id`（字符串值-不是引用）传递给`<constructor-arg/>`或`<property/>`元素。下面的例子展示了如何使用：

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

<bean id="theClientBean" class="..." >
    <property name="targetName">
        <idref bean="theTargetBean"/>
    </property>
</bean>
```

前面的 bean 定义代码段（在运行时）与以下代码段完全等效：

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

<bean id="client" class="..." >
    <property name="targetName" value="theTargetBean"/>
</bean>
```

第一种形式优于第二种形式，因为使用 `idref` 标记可使容器在部署时验证所引用的命名 Bean 实际上是否存在。在第二个变体中，不对传递给客户端 bean 的 `targetName` 属性的值执行验证。拼写错误仅在实际实例化客户端 bean 时发现（最有可能导致致命的结果）。如果客户端 Bean 是原型 Bean，则可能仅在部署容器后很长时间才发现此错字和所产生的

异常。

在 4.0 Bean XSD 中不再支持 `idref` 元素上的 `local` 属性，因为它不再提供常规 Bean 引用上的值。

升级到 4.0 模式时，将现有的 `idref local` 引用更改为 `idref bean`。

`<idref/>` 元素带来价值的一个常见地方（至少在 Spring 2.0 之前的版本中）是在 `ProxyFactoryBean` bean 定义中的 AOP 拦截器的配置中。指定拦截器名称时使用 `<idref/>` 元素可防止您拼写错误的拦截器 ID。

参考其他 Beans(协作者)

`ref` 元素是 `<constructor-arg/>` 或 `<property/>` 定义元素内的最后一个元素。在这里，您将一个 bean 的指定属性的值设置为对容器管理的另一个 bean (协作者) 的引用。引用的 bean 是要设置其属性的 bean 的依赖关系，并且在设置属性之前根据需要对其进行初始化。（如果协作者是单例 bean，则它可能已经由容器初始化了。）所有引用最终都是对另一个对象的引用。作用域和验证取决于您是否通过 `bean` 还是 `parent` 属性指定另一个对象的 ID 或名称。

通过 `<ref/>` 标记的 `bean` 属性指定目标 bean 是最通用的形式，并且允许创建对同一容器或父容器中任何 bean 的引用，而不管它是否在同一 XML 文件中。`bean` 属性的值可以与目标 bean 的 `id` 属性相同，也可以与目标 bean 的 `name` 属性中的值之一相同。下列例子展示了如何使用 `ref` 元素：

```
<ref bean="someBean"/>
```

通过 `parent` 属性指定目标 Bean 将创建对当前容器的父容器中的 Bean 的引用。`parent` 属性的值可以与目标 bean 的 `id` 属性或目标 bean 的 `name` 属性中的值之一相同。目标 Bean 必须位于当前容器的父容器中。主要在具有容器层次结构并且要使用与父 bean 名称相同的代理将现有 bean 封装在父容器中时，才应使用此 bean 参考变量。以下清单对显示了如何使用 `parent` 属性：

```
<!-- in the parent context -->
<bean id="accountService" class="com.something.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>
```



`ref` 元素的 `local` 属性在 4.0 Bean XSD 中不再受支持，因为它不再提供常规 Bean 引用上的值。升级到 4.0 模式时，将现有的 `ref local` 引用更改为 `ref bean`。

内部 bean

`<property/>` 或 `<constructor-arg/>` 元素内的 `<bean/>` 元素定义了一个内部 bean，如以下示例所示：

```
<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>
```

内部 bean 定义不需要定义的 ID 或名称。如果指定，则容器不使用某个值作为标识符。容器在创建时也将忽略作用域标志，因为内部 Bean 始终是匿名的，并且始终与外部 Bean 一起创建。不可能独立地访问内部 bean 或将其注入到协作 bean 中而不是封装到封闭 bean 中。

作为一个特例，可以从自定义作用域中接收销毁回调，例如对于单例 bean 中包含的请求范围内的 bean。内部 bean 实例的创建与其包含的 bean 绑定在一起，但是销毁回调使它可以参与请求范围的生命周期。这不是常见的情况。内部 bean 通常只共享其包含 bean 的作用域。

集合

`<list/>`, `<set/>`, `<map/>` 和 `<props/>` 元素分别设置 Java 集合类型 List, Set, Map 和 Properties 的属性和参数。以下示例显示了如何使用它们：

```

<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>

```

映射键或值的值或设置值也可以是以下任意元素：

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

集合合并：

Spring 容器还支持合并集合。 应用程序开发人员可以定义父`<list/>`, `<map/>`, `<set/>`或`<props/>`元素，并具有子`<list/>`, `<map/>`, `<set/>`或`<props/>`元素。 从父集合继承并覆盖值。也就是说，子集合的值是合并父集合和子集合的元素的结果，子集合的元素会覆盖父集合中指定的值。

关于合并的本节讨论了父子 bean 机制。不熟悉父级和子级 bean 定义的读者可能希望先阅读[相关部分\(1.7 Bean Definition Inheritance\)](#)，然后再继续。

下面的示例演示了集合合并：

```

<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>

```

注意子 bean 定义的 `adminEmails` 属性的 `<props/>` 元素上使用 `merge = true` 属性。当 `child` bean 由容器解析并实例化后，生成的实例具有 `adminEmails Properties` 集合，其中包含将 `child` 的 `adminEmails` 集合与父对象的 `adminEmails` 集合合并的结果。下列列表展示了结果内容：

```

administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk

```

子属性集的值集继承了父`<props />`的所有属性元素，而支持值的子值覆盖了父集合中的值。

此合并行为类似地适用于`<list/>`, `<map/>`和`<set/>`集合类型。在`<list/>`元素的特定情况下，将维护与 `List` 集合类型关联的语义（即，值的有序集合的概念）。父级的值先于子级列表的所有值。对于“`Map`”，“`Set`”和“`Properties`”集合类型，不存在任何排序。因此，对于容器内部使用的关联 `Map`, `Set` 和 `Properties` 实现类型基础的集合类型，没有任何排序语义有效。

集合合并的局限性

您不能合并不同的集合类型（例如 `Map` 和 `List`）。如果您尝试这样做，则会引发对应的异常。必须在下面的继承的子定义中指定 `merge` 属性。在父集合定义上指定 `merge` 属性

是多余的，不会导致所需的合并。

强类型集合

随着 Java 5 中泛型类型的引入，您可以使用强类型集合。也就是说，可以声明一个 `Collection` 类型，使其只能包含（例如）`String` 元素。如果使用 Spring 将强类型的集合依赖注入到 Bean 中，则可以利用 Spring 的类型转换支持，在你的强类型集合实例中的元素被添加进集合之前，预先转换为合适的类型。

以下 Java 类和 bean 定义显示了如何执行此操作：

Java

```
public class SomeClass {  
  
    private Map<String, Float> accounts;  
  
    public void setAccounts(Map<String, Float> accounts) {  
        this.accounts = accounts;  
    }  
}
```

Kotlin

```
class SomeClass {  
    lateinit var accounts: Map<String, Float>  
}
```

```
<beans>  
    <bean id="something" class="x.y.SomeClass">  
        <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>
```

当准备注入 `Something` bean 的 `accounts` 属性时，可以通过反射获得有关强类型 `Map<String, Float>` 的元素类型的泛型信息。因此，Spring 的类型转换基础结构将各种值元素识别为 `Float` 类型，并将字符串值（9.99、2.75 和 3.99）转换为实际的 `Float` 类型。

指向空或值为空的字符串

Spring 将属性等的空参数视为空 `String`。以下基于 XML 的配置元数据片段将 `email` 属性设置为空的 `String` 值（“”）。

```
<bean class="ExampleBean">
    <property name="email" value="" />
</bean>
```

前面的示例等效于以下 Java 代码：

Java

```
exampleBean.setEmail("");
```

Kotlin

```
exampleBean.email = ""
```

`<null />`元素处理空值。 以下清单显示了一个示例：

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

前面的示例等效于以下 Java 代码：

Java

```
exampleBean.setEmail(null);
```

Kotlin

```
exampleBean.email = null
```

P 命名空间的 XML 快捷方式

使用 `p-namespace`，您可以使用 `bean` 元素的属性（而不是嵌套的`<property />`元素）来描述协作 `bean` 的属性值，或同时使用这两者。

Spring 支持[带有名称空间（9.1XML Schema）](#)的可扩展配置格式，这些名称空间基于 XML Schema 定义。本章讨论的 `bean` 配置格式在 XML Schema 文档中定义。但是，`p` 命名空间未在 XSD 文件中定义，仅存在于 Spring 的核心(core)中。

下面的示例显示了两个 XML 代码段（第一个使用标准 XML 格式，第二个使用 `p` 命名空间），它们可以解析为相同的结果：

```

<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="someone@somewhere.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
          p:email="someone@somewhere.com"/>
</beans>

```

该示例显示了 p 命名空间中的一个属性，该属性在 bean 定义中称为 email。这告诉 Spring 包含一个属性声明。如前所述，p 名称空间没有架构定义，因此可以将属性名称设置为属性名称。

下一个示例包括另外两个 bean 定义，它们都引用了另一个 bean：

```

<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>

```

此示例不仅包括使用 p-namespace 的属性值，还使用特殊格式声明属性引用。第一个 bean 定义使用<property name="spouse" ref =“ jane”/>创建从 bean john 到 bean jane 的引用，而第二个 bean 定义使用 p:spouse-ref =“ jane”作为属性来执行完全一样的东西。在这种情况下，spouse 是属性名称，而-ref 部分指示这不是一个直接值，而是对另一个 bean 的引用。



p 命名空间不如标准 XML 格式灵活。例如，声明属性引用的格式与以 Ref 结尾的属性发生冲突，而标准 XML 格式则没有。我们建议您仔细选择方法，并与团队成员进行交流，以避免同时使用这三种方法生成 XML 文档。

C 命名空间的 XML 快捷方式

与带有 [p 命名空间的 XML 快捷方式](#) 类似，Spring 3.1 中引入的 c 命名空间允许使用内联属性来配置构造函数参数，而不是嵌套的 `constructor-arg` 元素。

以下示例使用 c: 命名空间执行与 [基于构造函数的依赖注入相同的操作](#):

```
<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="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

c: 命名空间使用与 p: 相同的约定（bean 引用为尾随-ref）以按名称设置构造函数参数。同样，即使未在 XSD 架构中定义它（也存在于 Spring 内核中），也需要在 XML 文件中声明它。

对于极少数情况下无法使用构造函数自变量名称的情况（通常，如果字节码是在没有调试信息的情况下编译的），则可以对参数索引使用后备，如下所示：

```
<!-- c-namespace index declaration -->
<bean id="beanOne" class="x.y.ThingOne" c:_0-ref="beanTwo" c:_1-ref="beanThree"
      c:_2="something@somewhere.com"/>
```



由于 XML 语法的原因，索引符号要求前导 `_` 的存在，因为 XML 属性名称不能以数字开头（即使某些 IDE 允许）。相应的索引符号也可用于 `<constructor-arg>` 元素，但并不常用，因为声明的普通顺序通常就足够了。

复合属性名

设置 bean 属性时，可以使用复合属性名称或嵌套属性名称，只要路径中除最终属性名称以外的所有组件都不为空即可。来看下面这个 bean 的定义：

```
<bean id="something" class="things.ThingOne">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

`something` bean 具有 `fred` 属性，该属性具有 `bob` 属性，该属性具有 `sammy` 属性，并且最终的 `sammy` 属性被设置为 `123` 的值。为了使它起作用，在构造 bean 之后，`something` 的 `fred` 属性和 `fred` 的 `bob` 属性一定不能为 `null`。否则，将引发 `NullPointerException`。

1.4.3. 使用 depends-on

如果一个 bean 是另一个 bean 的依赖项，则通常意味着将一个 bean 设置为另一个 bean 的属性。通常，您可以使用基于 XML 的配置元数据中的 `<ref/>` 元素 (References to Other Beans) 来完成此操作。但是，有时 bean 之间的依赖性不是直接的。一个例子是当在一个类的静态初始化块需要被触发，诸如用于数据库驱动的注册。`depends-on` 属性可以使使用这个元素的 bean 在初始化之前，强制一个或多个 bean 先被初始化。下面的示例使用 `depends-on` 属性来表示对单个 bean 的依赖关系：

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

要表达对多个 bean 的依赖关系，请提供一个 bean 名称列表作为 `Depends-on` 属性的值（逗号，空格和分号是有效的分隔符）：

```
<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" />
```



depends-on 属性既可以指定初始化时间依赖，也可以仅在单例 bean 的情况下指定相应的销毁时间依赖。与给定 bean 定义依赖关系的从属 bean 首先被销毁，然后再销毁给定 bean 本身。因此，依赖也可以控制关闭顺序。

1.4.4. 懒初始化 Beans

默认情况下，作为初始化过程的一部分，`ApplicationContext` 实现会急于创建和配置所有 `singleton(1.5.1)` bean。总的来说，这种预初始化是可取的，因为与数小时甚至数天后相比，会立即发现配置或周围环境中的错误。如果不希望使用此行为，则可以通过将 bean 定义标记为延迟初始化来防止单例 bean 的预实例化。延迟初始化的 bean 告诉 IoC 容器在首次请求时而不是在启动时创建一个 bean 实例。

在 XML 中，此行为由 `<bean/>` 元素上的 `lazy-init` 属性控制，如以下示例所示：

```
<bean id="lazy" class="com.something.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.something.AnotherBean"/>
```

当前面的配置写在 `ApplicationContext` 文件时，在 `ApplicationContext` 启动时不会急切地预实例化 `lazy` bean，而在 `not.lazy` Bean 中则急切地预实例化。

但是，当延迟初始化的 bean 是未延迟初始化的单例 bean 的依赖项时，`ApplicationContext` 会在启动时创建延迟初始化的 bean，因为它必须满足单例的依赖关系。延迟初始化的 bean 被注入到其他未延迟初始化的单例 bean 中。

您还可以通过使用 `<beans/>` 元素上的 `default-lazy-init` 属性在容器级别控制延迟初始化，以下示例显示：

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

1.4.5. 自动装配协作者

Spring 容器可以自动装配协作 beans 之间的关系。您可以通过检查 ApplicationContext 的内容，让 Spring 为您的 bean 自动解决协作者（其他 bean）。

自动装配有如下缺点：

- 自动装配可以大大减少指定属性或构造函数参数的需要。（在本章其他地方讨论的其他机制，例如 Bean 模板，在这方面也很有价值。）
- 随着对象的发展，自动装配可以更新配置。例如，如果您需要向类中添加一个依赖项，则无需修改配置即可自动满足该依赖项。因此，自动装配在开发过程中特别有用，而不必担心当代码库变得更稳定时切换到强制装配的选择。

使用基于 XML 的配置元数据时（请参阅“[依赖注入](#)”），可以使用 `<bean/>` 元素的 `autowire` 属性为 bean 定义指定自动装配模式。自动装配功能具有四种模式。你可以为每个 bean 指定自动装配，因此可以选择要自动装配的装配。下表描述了四种自动装配模式：

Table 2. Autowiring modes

Mode	Explanation
<code>no</code>	(Default) No autowiring. Bean references must be defined by <code>ref</code> elements. 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.
<code>byName</code>	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 <code>master</code> property (that is, it has a <code>setMaster(..)</code> method), Spring looks for a bean definition named <code>master</code> and uses it to set the property.
<code>byType</code>	Lets a property 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 <code>byType</code> autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).
<code>constructor</code>	Analogous to <code>byType</code> 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.

使用 `byType` 或构造函数自动装配模式，您可以连接数组和类型集合。在这种情况下，将提供容器中与期望类型匹配的所有自动装配候选，以满足相关性。如果期望的键类型为 `String`，则可以自动装配强类型 Map 实例。自动装配的 Map 实例的值包括与期望类型匹配的所有 bean 实例，并且 Map 实例的键包含相应的 bean 名称。

自动装配的限制和缺点

当在项目中一致使用自动装配时，自动装配效果最佳。如果通常不使用自动装配，那么使用开发人员仅连接一个或两个 bean 定义可能会使开发人员感到困惑。

我们考虑一下自动装配的限制和缺点：

- 属性和构造器参数设置中的显式依赖项始终会覆盖自动装配。您不能自动装配简单属性，例如基元，字符串和类（以及此类简单属性的数组）。此限制是设计使然。
- 自动装配不如显式装配精确。尽管如前表所述，Spring 还是小心避免在可能产生意外结果的模棱两可的情况下进行猜测。Spring 管理的对象之间的关系不再明确记录。
- 装配信息可能不适用于可能从 Spring 容器生成文档的工具。
- 容器内的多个 bean 定义可能与要自动装配的 setter 方法或构造函数参数指定的类型匹配。对于数组，集合或 Map 实例，这不一定是问题。但是，对于需要单个值的依赖项，不会武断的解决此歧义。如果没有唯一的 bean 定义可用，则引发异常。

在以后的场景中，您有几种选择：

- 放弃自动装配，转而使用显示装配。
- 通过将其 bean 的 `autowire-candidate` 属性设置为 `false`，避免自动装配 bean 定义，[如下一节](#) 所述。
- 通过将其`<bean/>`元素的 `primary` 属性设置为 `true`，将单个 bean 定义指定为主要候选对象。
- 如[基于注释的容器配置](#)中所述，通过基于注释的配置实现更细粒度的控件。

从自动装配中排除一个 Bean

在每个 bean 的基础上，您可以从自动装配中排除一个 bean。以 Spring XML 格式，将`<bean/>`元素的 `autowire-candidate` 属性设置为 `false`。容器使特定的 bean 定义不适用于自动装配基础结构（包括注释样式配置，例如`@Autowired`）。



`autowire-candidate` 属性被设计为仅影响基于类型的自动装配。它不会影响按名称显示的显式引用，即使未将指定的 Bean 标记为自动装配候选，该名称也可得到解析。因此，如果名称匹配，按名称自动装配仍会注入 Bean。

您还可以基于与 Bean 名称的模式匹配来限制自动装配候选。顶级`<beans/>`元素在其 `default-autowire-candidates` 属性内接受一个或多个模式串。例如，要将自动装配候选状态限制为名称以 `Repository` 结尾的任何 bean，请提供 `*Repository` 值。要提供多种模式，请在以逗号分隔的列表中定义它们。Bean 定义的 `autowire-candidate` 属性的显

式值 `true` 或 `false` 始终优先。对于此类 bean，模式匹配规则不适用。

这些技术对于您不希望通过自动装配将其注入其他 bean 的 bean 非常有用。这并不意味着排除的 bean 本身不能使用自动装配进行配置。相反，bean 本身不是自动装配其他 bean 的候选对象。

1.4.6. 方法注入

在大多数应用场景中，容器中的大多数 bean 是单例的。当单例 Bean 需要与另一个单例 Bean 协作或非单例 Bean 需要与另一个非单例 Bean 协作时，通常可以通过将一个 Bean 定义为另一个 Bean 的属性来处理依赖性。当 bean 的生命周期不同时会出现问题。假设单例 bean A 需要使用非单例（原型）bean B，也许是在 A 的每个方法调用上。容器仅创建一次 singleton bean A，因此只有一次机会来设置属性。每次需要一个容器时，容器都无法为 bean A 提供一个新的 bean B 实例。

一个解决方案是放弃某些控制反转。您可以通过实现 `ApplicationContextAware` 接口，并通过对容器进行 `getBean("B")` 调用来使 bean A 知道该容器，以便每次 bean A 需要它时都请求一个（通常是新的）bean B 实例。以下示例显示了此方法：

Java

```
// 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;
    }
}
```

Kotlin

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple

// Spring-API imports
import org.springframework.context.ApplicationContext
import org.springframework.context.ApplicationContextAware

class CommandManager : ApplicationContextAware {

    private lateinit var applicationContext: ApplicationContext

    fun process(commandState: Map<*, *>): Any {
        // grab a new instance of the appropriate Command
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // notice the Spring API dependency!
    protected fun createCommand() =
        applicationContext.getBean("command", Command::class.java)

    override fun setApplicationContext(applicationContext: ApplicationContext) {
        this.applicationContext = applicationContext
    }
}
```

前面的内容是不理想的，因为业务代码知道并耦合到 Spring 框架。方法注入是 Spring IoC 容器的一项高级功能，使您可以干净地处理此用例。

You can read more about the motivation for Method Injection in [this blog entry](#).

<https://spring.io>

查找方法注入

查找方法注入是容器重写被容器管理的 Bean 上的方法并返回容器中另一个已命名 Bean 的查找结果的能力。查找通常涉及原型 bean，如[上一节](#)中所述。Spring 框架通过使用从 CGLIB 库生成字节码来动态生成覆盖该方法的子类来实现此方法注入。

- 为了使此动态子类起作用，Spring Bean 容器子类的类也不能是 `final`，而要覆盖的方法也不能是 `final`。
- 对具有抽象方法的类进行单元测试需要您自己对该类进行子类化，并提供该抽象方法的存根实现。
- 组件扫描也需要具体方法，这需要具体的类别。
- 另一个关键限制是，查找方法不适用于工厂方法，尤其不适用于配置类中的`@Bean` 方法，因为在这种情况下，容器不负责创建实例，因此无法创建运行时生成的动态子类。

对于前面的代码片段中的 `CommandManager` 类，Spring 容器动态地覆盖 `createCommand()` 方法的实现。如重新编写的示例所示，`CommandManager` 类没有任何 Spring 依赖项：

Java

```
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();
}
```

Kotlin

```
package fiona.apple

// no more Spring imports!

abstract class CommandManager {

    fun process(commandState: Any): Any {
        // grab a new instance of the appropriate Command interface
        val command = createCommand()
        // set the state on the (hopefully brand new) Command instance
        command.state = commandState
        return command.execute()
    }

    // okay... but where is the implementation of this method?
    protected abstract fun createCommand(): Command
}
```

在包含要注入的方法的客户端类（在本例中为 `CommandManager`）中，要注入的方法需要以下形式的声明：

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

如果该方法是抽象的，则动态生成的子类将实现该方法。否则，动态生成的子类将覆盖原始类中定义的具体方法。请看下列示例：

```
<!-- 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>
```

每当需要新的 `myCommand` bean 实例时，标识为 `commandManager` 的 bean 就会调用其自己的 `createCommand()` 方法。如果确实需要将 `myCommand` bean 部署为原型，则必须小心。如果是单例，则每次都返回 `myCommand` bean 的相同实例。

另外，在基于注释的组件模型中，您可以通过`@Lookup` 注释声明一个查找方法，如下示例所示：

Java

```
public abstract class CommandManager {  
  
    public Object process(Object commandState) {  
        Command command = createCommand();  
        command.setState(commandState);  
        return command.execute();  
    }  
  
    @Lookup("myCommand")  
    protected abstract Command createCommand();  
}
```

Kotlin

```
abstract class CommandManager {  
  
    fun process(commandState: Any): Any {  
        val command = createCommand()  
        command.state = commandState  
        return command.execute()  
    }  
  
    @Lookup("myCommand")  
    protected abstract fun createCommand(): Command  
}
```

或者，更常用的是，你可以依赖于目标 bean 根据查找方法的声明的返回类型来解析：

Java

```
public abstract class CommandManager {  
  
    public Object process(Object commandState) {  
        MyCommand command = createCommand();  
        command.setState(commandState);  
        return command.execute();  
    }  
  
    @Lookup  
    protected abstract MyCommand createCommand();  
}
```

Kotlin

```
abstract class CommandManager {  
  
    fun process(commandState: Any): Any {  
        val command = createCommand()  
        command.state = commandState  
        return command.execute()  
    }  
  
    @Lookup  
    protected abstract fun createCommand(): Command  
}
```

请注意，通常应使用具体的存根实现声明此类带注解的查找方法，以使其与 Spring 的组件扫描规则（默认情况下抽象类会被忽略）兼容。此限制不适用于显式注册或显式导入的 Bean 类。



访问作用域不同的目标 bean 的另一种方法是 `ObjectFactory / Provider` 注入点。请参阅作用域 Bean 作为依赖项 [Scoped Beans as Dependencies \(1.5.4\)](#)。

您还可以在以下位置找到 `ServiceLocatorFactoryBean` (`org.springframework.beans.factory.config` 包) 很有用。

强制方法替换

与查找方法注入相比，方法注入的一种不太常用的形式是能够用另一种方法实现替换被管理 bean 中的任意方法。您可以放心地跳过本节的其余部分，直到您真正需要此功能为止。

借助基于 XML 的配置元数据，您可以使用 `replaced-method` 元素将现有的方法实现替换为已部署的 Bean。考虑以下类，该类具有一个我们要覆盖的名为 `computeValue` 的方法：

Java

```
public class MyValueCalculator {  
  
    public String computeValue(String input) {  
        // some real code...  
    }  
  
    // some other methods...  
}
```

Kotlin

```
class MyValueCalculator {  
  
    fun computeValue(input: String): String {  
        // some real code...  
    }  
  
    // some other methods...  
}
```

实现 `org.springframework.beans.factory.support.MethodReplacer` 接口的类提供了新的方法定义，如以下示例所示：

Java

```
/**  
 * 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 ...;  
    }  
}
```

Kotlin

```
/**  
 * meant to be used to override the existing computeValue(String)  
 * implementation in MyValueCalculator  
 */  
class ReplacementComputeValue : MethodReplacer {  
  
    override fun reimplement(obj: Any, method: Method, args: Array<out Any>): Any {  
        // get the input value, work with it, and return a computed result  
        val input = args[0] as String;  
        ...  
        return ...;  
    }  
}
```

用于部署原始类并指定方法覆盖的 Bean 定义类似于以下示例：

```

<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"/>

```

您可以在`<replaced-method/>`元素中使用一个或多个`<arg-type/>`元素，以指示要覆盖的方法的方法签名。仅当方法重载且类中存在多个变体时，才需要对参数标记。为了方便起见，参数的类型字符串可以是完全限定类型的子字符串，例如，以下所有都匹配`java.lang.String`：

```

java.lang.String
String
Str

```

因为参数的数量通常足以区分每个可能的选择，所以通过让您仅键入与参数类型匹配的最短字符串，此快捷方式可以节省很多输入。

1.5. Bean 的作用域

创建 bean 定义时，将创建一个配方来创建该 bean 定义所定义的类的实际实例。一个 bean 定义是一个配方的想法很重要，因为它意味着与类一样，您可以从一个配方中创建许多对象实例。

您不仅可以控制要插入到从特定 bean 定义创建的对象中的各种依赖项和配置值，还可以控制从特定 bean 定义创建的对象的作用域。这种方法功能强大且灵活，因为您可以选择通过配置创建的对象的范围，而不必在 Java 类级别上烘烤对象的范围。可以将 Bean 定义为部署在多个作用域之一中。Spring 框架支持六种作用域，其中只有在使用 Web 感知的 `ApplicationContext` 时才可用。您还可以创建[自定义范围（1.5.5）](#)。

下表表述了所支持的作用域：

Table 3. Bean scopes

Scope	Description
<code>singleton</code>	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
<code>prototype</code>	Scopes a single bean definition to any number of object instances.

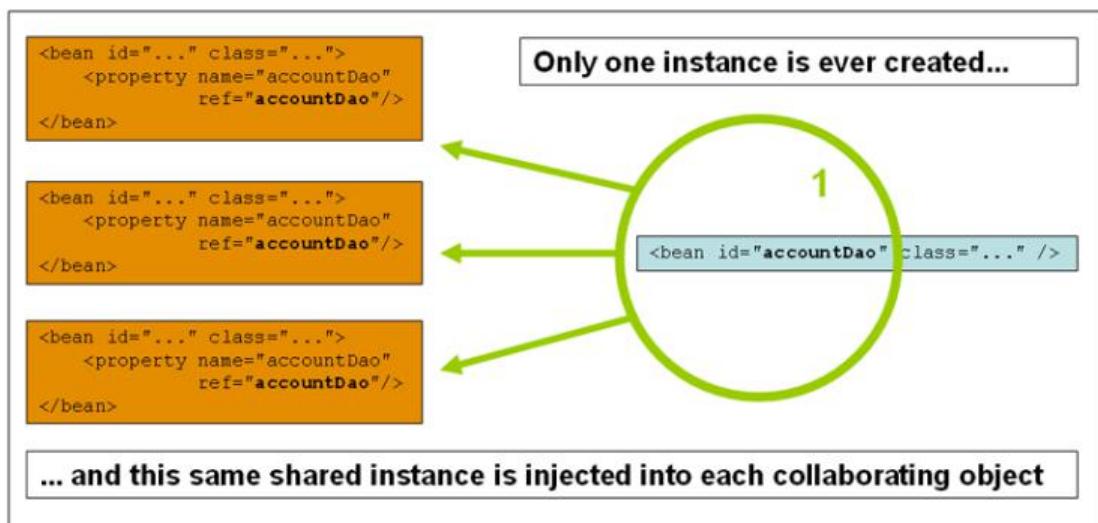
Scope	Description
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext .
session	Scopes a single bean definition to the lifecycle of an HTTP Session . Only valid in the context of a web-aware Spring ApplicationContext .
application	Scopes a single bean definition to the lifecycle of a ServletContext . Only valid in the context of a web-aware Spring ApplicationContext .
websocket	Scopes a single bean definition to the lifecycle of a WebSocket . Only valid in the context of a web-aware Spring ApplicationContext .

从 Spring 3.0 开始，线程作用域可用，但默认情况下未注册。有关更多信息，请参见 [SimpleThreadScope](#) 文档。有关如何注册此或任何其他自定义范围的说明，请参阅[使用自定义范围\(1.5.5\)](#)。

1.5.1. 单例作用域

仅管理一个 singleton bean 的一个共享实例，并且所有对具有 ID 或与该 bean 定义相匹配的 ID 的 bean 的请求都会导致该特定的 bean 实例由 Spring 容器返回。

换句话说，当您定义一个 bean 定义并且其作用域为单例时，Spring IoC 容器将为该 bean 定义所定义的对象创建一个实例。该单个实例存储在此类单例 bean 的高速缓存中，并且对该命名 bean 的所有后续请求和引用都返回该高速缓存的对象。下图显示了单例作用域如何工作：



Spring 的“单例 bean”概念与四人帮 (Gang of Four) 模式手册中定义的单例模式不

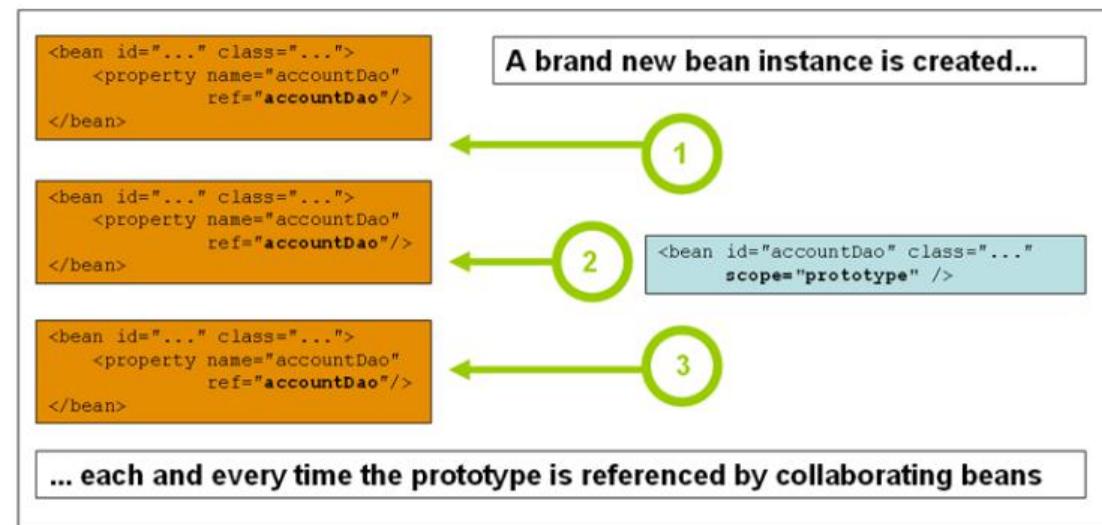
同。GoF 单例对对象的范围进行硬编码，以使每个类加载器 ClassLoader 只能创建一个特定类的一个实例。Spring 单例的作用域最好描述为每个容器和每个 bean。（这一句翻译没搞明白什么意思）。这意味着如果你在单个 Spring 容器中为特定类定义一个 bean，则 Spring 容器将根据该 bean 的定义创建有且仅有一个实例。Singleton 作用域是 Spring 中的默认作用域。要将 bean 定义为 XML 中的单例，可以定义 bean，如以下示例所示：

```
<bean id="accountService" class="com.something.DefaultAccountService"/>  
!-- the following is equivalent, though redundant (singleton scope is the default)  
-->  
<bean id="accountService" class="com.something.DefaultAccountService"  
scope="singleton"/>
```

1.5.2. 原型作用域

每次对特定 bean 提出请求时，bean 部署的非单例原型作用域内都会导致创建一个新 bean 实例。也就是说，该 Bean 被注入到另一个 Bean 中，或者您可以通过容器上的 `getBean()` 方法调用来请求它。通常，应将原型作用域用于所有有状态 Bean，将单例作用域用于无状态 Bean。

下图说明了 Spring 原型作用域：



（数据访问对象（DAO）通常不配置为原型，因为典型的 DAO 不拥有任何对话状态。对于我们而言，重用单例图的核心更为容易。）

以下示例将 bean 定义为 XML 原型：

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

与其他作用域相反，Spring 不管理原型 Bean 的完整生命周期。容器将实例化，配置或组装原型对象，然后将其交给客户端，而无需对该原型实例的进一步记录。因此，尽管在不考虑作用域的情况下在所有对象上都调用了初始化生命周期回调方法，但在原型的情况下，不会调用已配置的销毁生命周期回调。客户端代码必须清除原型作用域内的对象，并释放原型 Bean 拥有的昂贵资源。要使 Spring 容器释放原型作用域 bean 所拥有的资源，请尝试使用自定义 bean 后处理器（1.8.1），该处理器包含对需要清除的 bean 的引用。

在某些方面，Spring 容器在原型作用域内的 bean 方面的角色是 Java new 运算符的替代。超过该时间点的所有生命周期管理必须由客户端处理。（有关 Spring 容器中 bean 的生命周期的详细信息，请参阅[生命周期回调（1.6.1）](#)。）

1.5.3. 有原型 bean 依赖的单例 Bean

当您使用对原型 bean 有依赖性的单例作用域 Bean 时，请注意，依赖关系在实例化时已解析。因此，如果你把一个原型作用域 bean 注入到一个单例作用域 bean 中时，一个新的原型作用域实例将被创建并在稍后注入到单例 bean 中，原型实例是曾经提供给单例范围的 bean 的唯一实例。

但是，假设您希望单例作用域的 bean 在运行时重复获取原型作用域的 bean 的新实例。你不能将原型作用域的 bean 依赖项注入到您的单例 bean 中，因为当 Spring 容器实例化单例 bean 并解析并注入其依赖项时，该注入仅发生一次。如果在运行时不止一次需要原型 bean 的新实例，请参见[方法注入（1.4.6）](#)。

1.5.4. 请求、会话、应用程序和 WebSocket 作用域

仅当您使用可感知 Web 的 Spring ApplicationContext 实现（例如 XmlWebApplicationContext）时，request，session，application 和 websocket 范围才可用。如果将这些作用域与常规的 Spring IoC 容器（例如 ClassPathXmlApplicationContext）一起使用，则会抛出一个 IllegalStateException 异常，该错误抱怨未知的 bean 作用域。

初始化 Web 配置

为了在 request，session，application 和 websocket 级别（网络范围的 Bean）上支持 Bean 的作用域，在定义 Bean 之前，需要一些较小的初始配置。（对于标准范围：[单例](#)和[原型](#)，不需要此初始设置。）

如何完成此初始设置取决于您的特定 Servlet 环境。

如果您实际上在 Spring Web MVC 中访问由 Spring `DispatcherServlet` 处理的请求中的作用域 Bean，则不需要特殊的设置。`DispatcherServlet` 已经公开了所有相关状态。

如果您使用 Servlet 2.5 Web 容器，并且在 Spring 的 `DispatcherServlet` 外部处理请求（例如，使用 JSF 或 Struts 时），则需要注册 `org.springframework.web.context.request.RequestContextListener` `ServletRequestListener`。

对于 Servlet 3.0+，可以使用 `WebApplicationInitializer` 接口以编程方式完成此操作。或者，或者对于较旧的容器，将以下声明添加到 Web 应用程序的 `web.xml` 文件中：

```
<web-app>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

另外，如果您的监听器设置存在问题，请考虑使用 Spring 的 `RequestContextFilter`。过滤器映射取决于周围的 Web 应用程序配置，因此您必须适当地对其进行更改。以下清单显示了 Web 应用程序的过滤器部分：

```
<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` 和 `RequestContextFilter` 都做完全相同的事情，即将 HTTP 请求对象绑定到为该请求提供服务的线程。这使得在请求链和会话范围内的 Bean 可以在调用链的更下游使用。

请求作用域

考虑以下 XML 配置来定义 bean：

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

Spring 容器通过为每个 HTTP 请求使用 `loginAction` bean 定义来创建 `LoginAction` bean 的新实例。也就是说，`loginAction` bean 的作用域为 HTTP 请求级别。您可以根据需要任意更改创建的实例的内部状态，因为从同一 `loginAction` bean 定义创建的其他实例看不到状态的这些更改。它们特定于单个请求。当请求完成处理时，将限制作用于该请求的 Bean。

当使用注解驱动的组件或 Java 配置时，`@RequestScope` 注解可用于将组件分配给 [请求作用域](#)。以下示例显示了如何执行此操作：

Java

```
@RequestScope  
@Component  
public class LoginAction {  
    // ...  
}
```

Kotlin

```
@RequestScope  
@Component  
class LoginAction {  
    // ...  
}
```

会话作用域

考虑以下 XML 配置来定义 bean：

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

Spring 容器通过在单个 HTTP [会话](#) 的生存期内使用 `userPreferences` bean 定义来创建 `UserPreferences` bean 的新实例。换句话说，`userPreferences` bean 作用域在 HTTP [会话](#) 级别。与请求作用域的 Bean 一样，您可以根据需要任意更改所创建实例的内部状态，知道其他也在使用从同一 `userPreferences` Bean 定义创建的实例的 HTTP `Session` 实例不会看到这些状态更改，因为它们特定于单个 HTTP [会话](#)。当 HTTP 会话最终被丢弃时，作用于该特定 HTTP 会话的 bean 也将被丢弃。

使用注解驱动的组件或 Java 配置时，可以使用 `@SessionScope` 注解将组件分配给 [会话作用域](#)。

Java

```
@SessionScope  
@Component  
public class UserPreferences {  
    // ...  
}
```

Kotlin

```
@SessionScope  
@Component  
class UserPreferences {  
    // ...  
}
```

应用程序作用域

考虑以下 XML 配置来定义 bean:

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

Spring 容器通过对整个 Web 应用程序使用一次 `appPreferences` bean 定义来创建 `AppPreferences` bean 的新实例。也就是说，`appPreferences` bean 的作用域位于 `ServletContext` 级别，并存储为常规 `ServletContext` 属性。这有点类似于 Spring 单例 bean，但是有两个重要的区别：它是每个 `ServletContext` 的单例，而不是每个 Spring 'ApplicationContext' 的单例（在任何给定的 Web 应用程序中可能都有多个），并且它实际上是公开的，因此 可见为 `ServletContext` 属性。

使用注解驱动的组件或 Java 配置时，可以使用 `@ApplicationScope` 注解将组件分配给应用程序作用域。以下示例显示了如何执行此操作：

Java

```
@ApplicationScope  
@Component  
public class AppPreferences {  
    // ...  
}
```

Kotlin

```
@ApplicationScope  
@Component  
class AppPreferences {  
    // ...  
}
```

作用域 Bean 作为依赖

Spring IoC 容器不仅管理对象（bean）的实例化，而且还管理协作者（或依赖项）的连接。如果要将 HTTP 会话作用域的 bean 注入（例如）到作用域更长的另一个 bean 中，则可以选择注入 AOP 代理来代替作用域的 bean。也就是说，你需要注入暴露出一样的公共接口的作用域内的对象的代理对象，但也可以检索相关作用域的真正目标对象（如 HTTP 请求），并委托方法调用到真正的对象。

您还可以在作用域为单例的 bean 之间使用 `<aop: scoped-proxy/>`，然后引用通过可序列化的中间代理进行，因此能够在反序列化时重新获得目标单例 bean。

当针对作用域原型的 bean 声明 `<aop: scoped-proxy/>` 时，共享代理上的每个方法调用都会导致创建新的目标实例，然后将该调用转发到该目标实例。

同样，作用域代理不是以生命周期安全的方式从较小的作用域访问 bean 的唯一方法。您也可以将注入点（即构造函数或 `setter` 参数或自动装配的字段）声明为 `ObjectFactory <MyTargetBean>`，从而允许 `getObject()` 调用在需要时按需检索当前实例。实例或将其单独存储。

作为扩展变体，您可以声明 `ObjectProvider <MyTargetBean>`，它提供了几个附加的访问变体，包括 `getIfAvailable` 和 `getIfUnique`。

JSR-330 的这种变体称为 `Provider`，并与 `Provider <MyTargetBean>` 声明和每次检索尝试的相应 `get()` 调用一起使用。有关 JSR-330 更多详细信息，请参见 [此处\(1.11\)](#)。

以下示例中的配置仅一行，但是了解其背后的“原因”和“方式”很重要：

```
<?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.something.UserPreferences" scope="session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/> ①
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.something.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

① 定义代理的行。

要创建这样的代理，请将 `<aop: scoped-proxy />` 子元素插入到作用域 bean 定义中

(请参阅[选择要创建的代理类型](#)和[基于 XML Schema 的配置 9.1](#))。为什么在 `request`, `session` 和 `custom` 作用域级别定义的 bean 定义都需要`<aop: scopedproxy />`元素？考虑以下单例 bean 定义，并将其与需要为上述范围定义的内容进行对比（请注意，以下 `userPreferences` bean 定义不完整）：

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

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

在前面的示例中，单例 bean (`userManager`) 注入了对 HTTP 会话作用域 bean (`userPreferences`) 的引用。这里的重点是 `userManager` bean 是单例的：每个容器仅实例化一次，并且它的依赖项（在这种情况下，只有一个，`userPreferences` bean）也只注入一次。这意味着 `userManager` bean 仅在完全相同的 `userPreferences` 对象（即最初与之注入对象）上操作。

将生命周期短的作用域 bean 注入生命周期较长的作用域 bean 时，这不是想要的行为（例如，将 HTTP 会话范围的协作 bean 作为依赖项注入到 singleton bean 中）。相反，您只需要一个 `userManager` 对象，并且在 HTTP 会话的生存期内，您需要一个特定于 HTTP 会话的 `userPreferences` 对象。因此，容器创建了一个对象，该对象公开与 `UserPreferences` 类完全相同的公共接口（理想情况下是一个 `UserPreferences` 实例的对象），该对象可以从作用域机制（`HTTP request`, `Session` 等）中获取实际的 `UserPreferences` 对象。容器将此代理对象注入到 `userManager` bean 中，而后者不知道此 `UserPreferences` 引用是代理。在此示例中，当 `UserManager` 实例在注入依赖项的 `UserPreferences` 对象上调用方法时，实际上是在代理上调用方法。然后，代理从 HTTP 会话（在本例中）获取真实的 `UserPreferences` 对象，并将方法调用委托给检索到的真实的 `UserPreferences` 对象。

因此，在将请求范围和会话范围的 bean 注入到协作对象中时，您需要以下（正确和完整）配置，如以下示例所示：

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

选择要创建的代理类型

默认情况下，当 Spring 容器为使用 `<aop: scoped-proxy/>` 元素标记的 bean 创建代理时，将创建基于 CGLIB 的类代理。



CGLIB 代理仅拦截公共方法调用！不要在此类代理上调用非公共方法。它们没有被委派给实际的作用域目标对象。

另外，您可以通过为 `<aop: scoped-proxy/>` 元素的 `proxy-target-class` 属性值指定 `false`，来配置 Spring 容器为此类作用域的 bean 创建基于标准 JDK 接口的代理。使用基于 JDK 接口的代理意味着您不需要应用程序类路径中的其他库即可影响此类代理。但是，这也意味着作用域 bean 的类必须实现至少一个接口，并且作用域 bean 注入到其中的所有协作者都必须通过其接口之一引用该 bean。以下示例显示基于接口的代理：

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.stuff.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.stuff.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

1.5.5. 自定义作用域

Bean 作用域机制是可扩展的。您可以定义自己的作用域，甚至重新定义现有的作用域，尽管后者被认为是不好的做法，并且您不能覆盖内置的 `singleton` 和 `prototype` 作用域。

创建一个自定义作用域

要将自定义范围集成到 Spring 容器中，需要实现 `org.springframework.beans.factory.config.Scope` 接口，本节对此进行了介绍。有关如何实现自己的作用域的想法，请参阅 Spring 框架本身提供的 `Scope` 实现和 `Scope javadoc`，其中详细说明了需要实现的方法。

作用域接口有四种方法可以从作用域中获取对象，将它们从作用域中删除，然后销毁它

们。

例如，会话作用域实现返回会话作用域的 Bean（如果不存在，则该方法将其绑定到会话以供将来参考之后，将返回该 Bean 的新实例）。以下方法从基础范围返回对象：

Java

```
Object get(String name, ObjectFactory<?> objectFactory)
```

Kotlin

```
fun get(name: String, objectFactory: ObjectFactory<*>): Any
```

会话作用域的实现，例如，从基础会话中删除了会话作用域的 bean。应该返回该对象，但是如果找不到具有指定名称的对象，则可以返回 null。以下方法从基础作用域中删除该对象：

Java

```
Object remove(String name)
```

Kotlin

```
fun remove(name: String): Any
```

以下方法注册一个回调，当作用域被销毁或作用域中的指定对象被销毁时，作用域应调用该回调：

Java

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

Kotlin

```
fun registerDestructionCallback(name: String, destructionCallback: Runnable)
```

有关销毁回调的更多信息，请参见 [javadoc](#) 或 Spring 作用域实现。以下方法获取基础范围的会话标识符：

Java

```
String getConversationId()
```

Kotlin

```
fun getConversationId(): String
```

每个作用域的标识符都不相同。对于会话作用域的实现，此标识符可以是会话标识符。

使用一个自定义作用域

在编写和测试一个或多个自定义作用域实现之后，需要使 Spring 容器检测到您的新作用域。以下方法是在 Spring 容器中注册新范围的主要方法：

Java

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

Kotlin

```
fun registerScope(scopeName: String, scope: Scope)
```

该方法在 `ConfigurableBeanFactory` 接口上声明，该接口可通过 Spring 附带的大多数 `ApplicationContext` 的具体实现上的 `BeanFactory` 属性获得。

`registerScope(..)` 方法的第一个参数是与作用域相关联的唯一名称。Spring 容器本身中的此类名称的示例包括 `singleton` 和 `prototype`。`registerScope(..)` 方法的第二个参数是你希望注册和使用的自定义作用域实现的实际实例。

假设您编写了自定义的 `Scope` 实现，然后注册它，如下面的示例所示。



下一个示例使用 `SimpleThreadScope`，它包含在 Spring 中，但默认情况下未注册。对于您自己的自定义作用域实现，说明将是相同的。

Java

```
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

Kotlin

```
val threadScope = SimpleThreadScope()
beanFactory.registerScope("thread", threadScope)
```

然后，您可以按照您的自定义作用域的作用域规则创建 bean 定义，如下所示：

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

使用自定义作用域实现，您不仅限于以编程方式注册作用域。您还可以使用 `CustomScopeConfigurer` 类以声明方式进行作用域注册，如以下示例所示：

```

<?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="thing2" class="x.y.Thing2" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="thing1" class="x.y.Thing1">
        <property name="thing2" ref="thing2"/>
    </bean>

```

</beans>



当将 `<aop:scoped-proxy/>` 放置在 `FactoryBean` 实现中时，作用域是工厂 bean 本身，而不是从 `getObject()` 返回的对象。

1.6. 自定义 Bean 的性质

Spring 框架提供了许多接口，可用于自定义 Bean 的性质。本节将它们分组如下：

- [生命周期回调\(1.6.1\)](#)
- [ApplicationContextAware 和 BeanNameAware\(1.6.2\)](#)
- [其他 Aware 接口\(1.6.3\)](#)

1.6.1. 生命周期回调

为了与容器对 bean 生命周期的管理进行交互，您可以实现 `Spring InitializingBean` 和 `DisposableBean` 接口。容器为前者调用 `afterPropertiesSet()` 并为后者调用 `destroy()`，以使 Bean 在初始化和销毁 Bean 时执行某些操作。



通常，在现代 Spring 应用程序中，JSR-250 @PostConstruct 和 @PreDestroy 注解被认为是接收生命周期回调的最佳实践。使用这些注解意味着您的 bean 没有耦合到特定于 Spring 的接口。有关详细信息，请参见[使用@PostConstruct 和@PreDestroy\(1.9.9\)](#)。

如果您不想使用 JSR-250 注解，但仍然想解除耦合，请考虑使用 `init-method` 和 `destroy-method` Bean 定义元数据。

在内部，Spring 框架使用 `BeanPostProcessor` 实现来处理它可以找到的任何回调接口并调用适当的方法。如果您需要自定义特性或其他生命周期行为，默认情况下 Spring 不提供，则您可以自己实现 `BeanPostProcessor`。有关更多信息，请参见[容器扩展点 \(1.8\)](#)。

除了初始化和销毁回调，Spring 管理的对象还可以实现 `Lifecycle` 接口，以便这些对象可以在容器自身的生命周期的驱动下参与启动和关闭过程。

本节介绍了生命周期回调接口。

初始化回调

使用 `org.springframework.beans.factory.InitializingBean` 接口，容器在容器上设置了所有必要的属性后，即可执行初始化工作。`InitializingBean` 接口指定一个方法：

Java

```
void afterPropertiesSet() throws Exception;
```

Kotlin

```
fun afterPropertiesSet()
```

我们建议您不要使用 `InitializingBean` 接口，因为它不必要地将代码耦合到 Spring。另外，我们建议使用 `@PostConstruct` 批注或指定 POJO 初始化方法。对于基于 XML 的配置元数据，可以使用 `init-method` 属性指定具有无效无参数签名的方法的名称。通过 Java 配置，可以使用 `@Bean` 的 `initMethod` 属性。请参阅[接收生命周期回调\(1.12.3/152, 153\)](#)。请看以下示例：

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

Java

```
public class ExampleBean {  
  
    public void init() {  
        // do some initialization work  
    }  
}
```

Kotlin

```
class ExampleBean {  
  
    fun init() {  
        // do some initialization work  
    }  
}
```

前面的示例与下面的示例（由两个列表组成）几乎具有完全相同的效果：

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

Java

```
public class AnotherExampleBean implements InitializingBean {  
  
    @Override  
    public void afterPropertiesSet() {  
        // do some initialization work  
    }  
}
```

Kotlin

```
class AnotherExampleBean : InitializingBean {  
  
    override fun afterPropertiesSet() {  
        // do some initialization work  
    }  
}
```

但是，前面两个示例中的第一个示例并未将代码耦合到 Spring。

回调销毁

通过实现 `org.springframework.beans.factory.DisposableBean` 接口，当包含 bean 的容器被销毁时，它可以获取回调。DisposableBean 接口指定一个方法：

Java

```
void destroy() throws Exception;
```

Kotlin

```
fun destroy()
```

我们建议您不要使用 `DisposableBean` 回调接口，因为它不必要地将代码耦合到 Spring。另外，我们建议使用`@PreDestroy` 批注或指定 bean 定义支持的通用方法。使用基于 XML 的配置元数据时，可以在`<bean/>`标签上使用 `destroy-method` 属性。通过 Java 配置，可以使用`@Bean` 的 `destroyMethod` 属性。请参阅接收生命周期回调。请看以下定义：

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

Java

```
public class ExampleBean {  
  
    public void cleanup() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

Kotlin

```
class ExampleBean {  
  
    fun cleanup() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

前面的示例与下面的示例（由两个列表组成）几乎具有完全相同的效果：

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

Java

```
public class AnotherExampleBean implements DisposableBean {  
  
    @Override  
    public void destroy() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

Kotlin

```
class AnotherExampleBean : DisposableBean {  
  
    override fun destroy() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

但是，前面两个示例中的第一个示例并未将代码耦合到 Spring。



您可以为`<bean>`标签的 `destroy-method` 属性分配一个特殊的（推断的）值，该值指示 Spring 自动检测特定 bean 类上的公共 `close` 或 `shutdown` 方法。（因此，任何实现 `java.lang.AutoCloseable` 或 `java.io.Closeable` 的类都将匹配。）您还可以在`<beans>`元素的 `default-destroy-method` 属性上设置此特殊（推断）值，以将此行为应用于整个 bean 集（请参见[默认初始化和销毁方法\(如下\)](#)）。请注意，这是 Java 配置的默认行为。

默认的初始化和销毁方法

当编写不使用 Spring 特定的 `InitializingBean` 和 `DisposableBean` 回调接口的初始化和销毁方法回调时，通常会使用诸如 `init()`, `initialize()`, `dispose()` 之类的名字来编写方法。理想情况下，此类生命周期回调方法的名字应在整个项目中标准化，以便所有开发人员都使用相同的方法名称并确保一致性。

您可以配置 Spring 容器以“look”命名每个 bean 上的 `initialization` 和 `destroy` 回调方法名称。这意味着，作为应用程序开发人员，您可以编写应用程序类并使用称为 `init()` 的初始化回调，而不必为每个 bean 定义配置 `init-method="init"` 属性。Spring IoC 容器在创建 bean 时（并根据前面描述的标准生命周期回调协定）调用该方法。此功能还对 `initialization` 和 `destroy` 方法回调强制执行一致的命名约定。

假设您的初始化回调方法命名为 `init()`，而 `destroy` 回调方法命名为 `destroy()`。

然后，您的类类似于以下示例中的类：

Java

```
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.");
        }
    }
}
```

Kotlin

```
class DefaultBlogService : BlogService {

    private var blogDao: BlogDao? = null

    // this is (unsurprisingly) the initialization callback method
    fun init() {
        if (blogDao == null) {
            throw IllegalStateException("The [blogDao] property must be set.")
        }
    }
}
```

然后，您可以在类似于以下内容的 Bean 中使用该类：

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

    <bean id="blogService" class="com.something.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

顶级`<beans/>`标签属性上存在 `default-init-method` 属性，导致 Spring IoC 容器将 Bean 类上称为 `init` 的方法识别为初始化方法回调。在创建和组装 bean 时，如果 bean 类具有这种方法，则会在适当的时间调用它。

您可以通过使用顶级`<beans/>`标签上的 `default-destroy-method` 属性类似地（在 XML 中）配置 `destroy` 方法回调。

如果现有的 Bean 类已经具有按惯例命名的回调方法，你可以通过使用`<bean/>`本身的

`init-method` 和 `destroy-method` 属性指定（在 XML 中）方法名称来覆盖默认值。

Spring 容器保证在为 bean 提供所有依赖项后立即调用配置的初始化回调。因此，在原始 bean 引用上调用了初始化回调，这意味着 AOP 拦截器等尚未应用于 bean。首先完全创建目标 bean，然后应用带有其拦截器链的 AOP 代理（例如）。如果目标 Bean 和代理分别定义，则您的代码甚至可以绕过代理与原始目标 Bean 进行交互。因此，将拦截器应用于 `init` 方法将是不一致的，因为这样做会将目标 Bean 的生命周期耦合到其代理或拦截器，并且当您的代码直接与原始目标 Bean 进行交互时会留下奇怪的语义。

结合生命周期机制

从 Spring 2.5 开始，您可以使用三个选项来控制 Bean 生命周期行为：

- `InitializingBean` 和 `DisposableBean` 回调接口
- 自定义 `init()` 和 `destroy()` 方法
- `@PostConstruct` 和`@PreDestroy` 注解（1.9.9）。

你可以结合这些机制去控制一个给出的 bean。

如果为一个 bean 配置了多个生命周期机制，并且为每个机制配置了不同的方法名称，则每个配置的方法都将按照此注释后列出的顺序运行。
但是，如果为多个生命周期机制中的多个生命周期配置了相同的方法名称（例如，为初始化方法使用 `init()`），则该方法将运行一次，如上一节所述。

为同一个 bean 配置的具有不同初始化方法的多种生命周期机制如下：

1. `@PostConstruct` 方法注解
2. 由 `InitializingBean` 回调接口定义的 `afterPropertiesSet()`。
3. 一个自定义配置的 `init()` 方法

销毁方法以同样的顺序被调用：

1. `@PostDestroy` 方法注解
2. 由 `DisposableBean` 回调接口定义的 `destroy()`。
3. 一个自定义配置的 `destroy()` 方法

启动和关闭回调

`Lifecycle` 接口为具有自己的生命周期要求（例如启动和停止某些后台进程）的任何对象定义了基本方法：

Java

```
public interface Lifecycle {  
    void start();  
    void stop();  
    boolean isRunning();  
}
```

Kotlin

```
interface Lifecycle {  
    fun start()  
    fun stop()  
    val isRunning: Boolean  
}
```

任何 Spring 管理的对象都可以实现 `Lifecycle` 接口。然后，当 `ApplicationContext` 本身接收到启动和停止信号时（例如，对于运行时的停止/重新启动场景），它将把这些调用级联到在该上下文中定义的所有 `Lifecycle` 实现。它通过委派给 `LifecycleProcessor` 来做到这一点，如以下列表所示：

Java

```
public interface LifecycleProcessor extends Lifecycle {  
    void onRefresh();  
    void onClose();  
}
```

Kotlin

```
interface LifecycleProcessor : Lifecycle {  
    fun onRefresh()  
    fun onClose()  
}
```

请注意，`LifecycleProcessor` 本身是 `Lifecycle` 接口的扩展。它还添加了两种其他方法来响应正在刷新和关闭的上下文。



请注意，常规的 `org.springframework.context.Lifecycle` 接口是用

于显式启动和停止通知的普通协议，并不意味着在上下文刷新时自动启动。为了对特定 bean 的自动启动（包括启动阶段）进行细粒度的控制，请考虑改为实现 `org.springframework.context.SmartLifecycle`。

另外，请注意，不能保证会在销毁之前发出停止通知。在常规关闭时，在传播常规销毁回调之前，所有 `Lifecycle` bean 都会首先收到停止通知。但是，在上下文生存期内进行热刷新或停止刷新尝试时，仅调用 `destroy` 方法。

启动和关闭调用的顺序可能很重要。如果任何两个对象之间存在“依赖”关系，则依赖方在其依赖之后开始，而在依赖之前停止。但是，有时直接依赖项是未知的。您可能只知道某种类型的对象应该先于另一种类型的对象开始。在这些情况下，`SmartLifecycle` 接口定义了另一个选项，即在其父接口 `Phased` 上定义的 `getPhase()` 方法。以下清单显示了 `Phased` 接口的定义：

Java

```
public interface Phased {  
    int getPhase();  
}
```

Kotlin

```
interface Phased {  
    val phase: Int  
}
```

以下列表显示了 `SmartLifecycle` 接口的定义：

Java

```
public interface SmartLifecycle extends Lifecycle, Phased {  
  
    boolean isAutoStartup();  
  
    void stop(Runnable callback);  
}
```

Kotlin

```
interface SmartLifecycle : Lifecycle, Phased {  
    val isAutoStartup: Boolean  
    fun stop(callback: Runnable)  
}
```

启动时，相位最低的对象首先启动。停止时，这个顺序就会倒过来。因此，实现 `SmartLifecycle` 并且其 `getPhase()` 方法返回 `Integer.MIN_VALUE` 的对象将是第一个启动且最后一个停止的对象。在这组的另一端，相位值 `Integer.MAX_VALUE` 表示该对象应最后启动并首先停止（可能是因为它依赖于正在运行的其他进程）。在考虑阶段值时，重要的是要知道，任何未实现 `SmartLifecycle` 的“普通” `Lifecycle` 对象的默认阶段为 `0`。因此，任何负相位值都表明对象应在这些标准组件之前开始（并在它们之后停止）。对于任何正相位值，反之亦然。

`SmartLifecycle` 定义的 `stop` 方法接受回调。任何实现都必须在该实现的关闭过程完成后调用该回调的 `run()` 方法。这将在必要时启用异步关闭，因为 `LifecycleProcessor` 接口的默认实现 `DefaultLifecycleProcessor` 会等待每个阶段内的对象组的超时值等待调用该回调。默认的每阶段超时为 30 秒。您可以通过在上下文中定义一个名为 `lifecycleProcessor` 的 bean 来覆盖默认的生命周期处理器实例。如果只想修改超时，则定义以下内容即可：

```
<bean id="lifecycleProcessor"  
      class="org.springframework.context.support.DefaultLifecycleProcessor">  
    <!-- timeout value in milliseconds -->  
    <property name="timeoutPerShutdownPhase" value="10000"/>  
  </bean>
```

如前所述，`LifecycleProcessor` 接口还定义了用于刷新和关闭上下文的回调方法。稍后驱动关闭处理，就好像已经显式调用了 `stop()` 一样，但是它在上下文关闭时发生。另一方面，“refresh”回调启用 `SmartLifecycle` bean 的另一个功能。刷新上下文时（在所有对象都被实例化和初始化之后），该回调将被调用。届时，默认生命周期处理器将检查每个 `SmartLifecycle` 对象的 `isAutoStartup()` 方法返回的布尔值。如果为 `true`，则从那时开始启动该对象，而不是等待上下文或它自己的 `start()` 方法的显式调用（与上下文刷新不同，对于标准上下文实现，上下文启动不会自动发生）。相位值和任何“依赖”关系决定了启动顺序，如前所述。

在非 Web 应用程序中优雅地关闭 Spring IoC 容器



本节仅适用于非 Web 应用程序。Spring 的基于 Web 的 `ApplicationContext` 实现已经具有适当的代码，可以在相关 Web 应用程序关闭时正常关闭 Spring IoC 容器。

如果您在非 Web 应用程序环境中（例如，在富客户端桌面环境中）使用 Spring 的 IoC 容器，请向 JVM 注册一个关闭钩子。这样做可以确保正常关机，并在您的 Singleton bean 上调用相关的 `destroy` 方法，以便释放所有资源。您仍然必须正确配置和实现这些 `destroy` 回调。

要注册关闭挂钩，请调用在 `ConfigurableApplicationContext` 接口上声明的 `registerShutdownHook()` 方法，如以下示例所示：

Java

```
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...
    }
}
```

Kotlin

```
import org.springframework.context.support.ClassPathXmlApplicationContext

fun main() {
    val ctx = 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 和 BeanNameAware 接口

当 `ApplicationContext` 创 建 实 现
org.springframework.context.ApplicationContextAware 接口的对象实例时，将为该实例提供对该 `ApplicationContext` 的引用。以下清单显示了 `ApplicationContextAware` 接口的定义：

Java

```
public interface ApplicationContextAware {  
  
    void setApplicationContext(ApplicationContext applicationContext) throws  
    BeansException;  
}
```

Kotlin

```
interface ApplicationContextAware {  
  
    @Throws(BeansException::class)  
    fun setApplicationContext(applicationContext: ApplicationContext)  
}
```

因此，bean 可以通过 `ApplicationContext` 接口或通过将引用转换为该接口的已知子类（例如 `ConfigurableApplicationContext`，它公开了其他功能）来以编程方式操纵创建它们的 `ApplicationContext`。一种用途是通过编程方式检索其他 bean。有时，此功能很有用。但是，通常应避免使用它，因为它将代码耦合到 Spring，并且不遵循控制反转样式，在该样式中，将协作者作为属性提供给 bean。`ApplicationContext` 的其他方法提供对文件资源的访问，发布应用程序事件以及访问 `MessageSource`。这些附加功能在 `ApplicationContext` 的其他功能（1.15）中进行了描述。

自动装配是获得对 `ApplicationContext` 的引用的另一种选择。传统的构造函数和 `byType` 自动装配模式（如“[自动装配协作器（1.4.5）](#)”中所述）可以分别为构造函数参数或 `setter` 方法参数提供 `ApplicationContext` 类型的依赖项。要获得更大的灵活性，包括能够自动连接字段和使用多个参数方法，请使用基于注释的自动装配功能。如果这样做，则将 `ApplicationContext` 自动连装配需要使用 `ApplicationContext` 类型的字段，构造函数参数或方法参数中（如果有问题的字段，构造函数或方法带有`@Autowired` 批注）。有关更多信息，请参见[使用`@Autowired`（1.9.2）](#)。

当 `ApplicationContext` 创 建 一 个 实 现
org.springframework.beans.factory.BeanNameAware 接口，该类提供了对在其关联

对象定义中定义的名称的引用。以下列表显示了 BeanNameAware 接口的定义：

Java

```
public interface BeanNameAware {  
    void setBeanName(String name) throws BeansException;  
}
```

Kotlin

```
interface BeanNameAware {  
    @Throws(BeansException::class)  
    fun setBeanName(name: String)  
}
```

回调应该在一个普通 bean 的属性填充后，但也应该在一个初始化回调前，比如 InitializingBean, afterPropertiesSet, 或者一个自定义初始化方法(init-method)。

1.6.3. 其他 Aware 接口

除了 ApplicationContextAware 和 BeanNameAware(前面已经讨论过)之外，Spring 还提供了多种 Aware 回调接口，这些接口使 Bean 向容器指示它们需要某种基础结构依赖性。通常，名称表示依赖项类型。下表总结了最重要的 Aware 接口：

Table 4. Aware interfaces

Name	Injected Dependency	Explained in...
ApplicationContextAware	Declaring ApplicationContext.	ApplicationContextAware and BeanNameAware
ApplicationEventPublisherAware	Event publisher of the enclosing ApplicationContext.	Additional Capabilities of the ApplicationContext
BeanClassLoaderAware	Class loader used to load the bean classes.	Instantiating Beans
BeanFactoryAware	Declaring BeanFactory.	ApplicationContextAware and BeanNameAware
BeanNameAware	Name of the declaring bean.	ApplicationContextAware and BeanNameAware
BootstrapContextAware	Resource adapter BootstrapContext the container runs in. Typically available only in JCA-aware ApplicationContext instances.	JCA CCI
LoadTimeWeaverAware	Defined weaver for processing class definition at load time.	Load-time Weaving with AspectJ in the Spring Framework

Name	Injected Dependency	Explained in...
<code>MessageSourceAware</code>	Configured strategy for resolving messages (with support for parametrization and internationalization).	Additional Capabilities of the ApplicationContext
<code>NotificationPublisherAware</code>	Spring JMX notification publisher.	Notifications
<code>ResourceLoaderAware</code>	Configured loader for low-level access to resources.	Resources
<code>ServletConfigAware</code>	Current <code>ServletConfig</code> the container runs in. Valid only in a web-aware Spring <code>ApplicationContext</code> .	Spring MVC
<code>ServletContextAware</code>	Current <code>ServletContext</code> the container runs in. Valid only in a web-aware Spring <code>ApplicationContext</code> .	Spring MVC

再次注意，使用这些接口会将您的代码与 Spring API 绑定在一起，并且不遵循“控制反转”样式。因此，我们建议将它们用于需要以编程方式访问容器的基础结构 Bean。

1. 7. Bean 定义继承

Bean 定义可以包含许多配置信息，包括构造函数参数，属性值和特定于容器的信息，例如初始化方法，静态工厂方法名称等。子 bean 定义从父定义继承配置数据。子定义可以覆盖某些值或根据需要添加其他值。使用父 bean 和子 bean 定义可以节省很多输入。实际上，这是一种模板形式。

如果您以编程方式使用 `ApplicationContext` 接口，则子 bean 定义由 `ChildBeanDefinition` 类表示。大多数用户不在此级层级上与他们合作。相反，它们在诸如 `ClassPathXmlApplicationContext` 之类的类中声明性地配置 Bean 定义。当使用基于 XML 的配置元数据时，可以通过使用父属性来指示子 Bean 定义，并指定父 Bean 作为该属性的值。以下示例显示了如何执行此操作：当使用基于 XML 的配置元数据时，可以通过使用父属性来指示子 Bean 定义，并指定父 Bean 作为该属性的值。以下示例显示了如何执行此操作：

```

<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"> ①
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>

```

① 注意 `parent` 属性

如果未指定子 bean 定义，则使用父定义中的 bean 类，但也可以覆盖它。在后一种情况下，子 bean 类必须与父类兼容（也就是说，它必须接受父类的属性值）。

子 bean 定义从父对象继承作用域，构造函数参数值，属性值和方法覆写，并可以选择添加新值。你指定的任何作用域，初始化方法，`destroy` 方法或静态工厂方法设置都会覆盖相应的父设置。

其余设置始终从子定义中获取：依赖项，自动装配模式，依赖项检查，单例和惰性初始化。

前面的示例使用 `abstract` 属性将父 bean 定义显式标记为抽象类。如果父定义未指定为一个类，则需要将父 bean 定义显式标记为抽象类，如以下示例所示：

```

<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>

```

父 bean 不能单独实例化，因为它不完整，并且还被明确标记为抽象。当定义是抽象的时，它只能用作纯模板 bean 定义，用作子定义的父定义。如果尝试使用这么一个抽象父类 bean 于它自身，比如咱引用它自己作为另一个 bean 的 `ref` 属性或者使用父类的 bean ID 显式的调用 `getBean()` 方法将会返回一个错误。类似的，容器内部的 `preInstantiateSingletons()` 方法将会忽略那些被定义为抽象的 bean。



`ApplicationContext` 默认的将所有单例进行预初始化。因此，有个很重要（至少是对单例 bean）的事就是你有一个仅仅打算作为模板的（父类）bean 定义并且它被指定了一个类，你必须确定将 `abstract` 属性设为 `true`，否则应用上下文将会实际（或尝试）去预编译它。

1.8. 容器扩展点

通常，应用程序开发人员不需要为 `ApplicationContext` 实现类提供子类。相反，可以通过插入特殊集成接口的实现来扩展 Spring IoC 容器。接下来的几节描述了这些集成接口。

1.8.1. 通过使用 BeanPostProcessor 自定义 Beans

`BeanPostProcessor` 接口定义了回调方法，您可以实施这些回调方法以提供自己的（或覆盖容器的默认值）实例化逻辑，依赖关系解析逻辑等。如果您想在 Spring 容器完成实例化，配置和初始化 bean 之后实现一些自定义逻辑，则可以插入一个或多个自定义 `BeanPostProcessor` 实现。

您可以配置多个 `BeanPostProcessor` 实例，并且可以通过设置 `order` 属性来控制这些 `BeanPostProcessor` 实例的运行顺序。仅当 `BeanPostProcessor` 实现 `Ordered` 接口时，才可以设置此属性。如果编写自己的 `BeanPostProcessor`，则也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见 `BeanPostProcessor` 和 `Ordered` 接口的 javadoc。另请参见有关[以编程方式注册 BeanPostProcessor 实例](#)的说明。

`BeanPostProcessor` 实例在 bean（或对象）实例上运行。也就是说，Spring IoC 容器实例化一个 bean 实例，然后 `BeanPostProcessor` 实例完成其工作。



`BeanPostProcessor` 实例是按容器划分作用域的。仅在使用容器层次结构时，这才有意义。如果在一个容器中定义 `BeanPostProcessor`，它将仅对该容器中的 bean 进行后处理。换句话说，一个容器中定义的 bean 不会由另一个容器中定义的 `BeanPostProcessor` 进行后处理，即使这两个容器是同一层次结构的一部分。

要更改实际的 bean 定义（即定义 bean 的蓝图），您需要使用 `BeanFactoryPostProcessor`，如使用 [BeanFactoryPostProcessor 自定义配置元数据\(1.8.2\)](#) 中所述。

`org.springframework.beans.factory.config.BeanPostProcessor` 接口恰好由

两个回调方法组成。将此类注册为容器的 post-processor 时，对于容器创建的每个 bean 实例，后处理器都会在容器初始化方法（例如 `InitializingBean.afterPropertiesSet()` 或任何声明的 `init` 方法），并在任何 bean 初始化之后被调用。post-processor 可以对 bean 实例执行任何操作，包括完全忽略回调。Bean 后处理器通常检查回调接口，或者可以用代理包装 Bean。一些 Spring AOP 基础结构类被实现为 bean 后处理器，以提供代理包装逻辑。

`ApplicationContext` 自动检测配置元数据中实现 `BeanPostProcessor` 接口的定义的所有 bean。`ApplicationContext` 将这些 bean 注册为 post-processor，以便以后在 bean 创建时可以调用它们。Bean 后处理器可以与其他任何 Bean 相同的方式部署在容器中。

请注意，在配置类上使用 `@Bean` 工厂方法声明 `BeanPostProcessor` 时，工厂方法的返回类型应该是实现类本身，或者至少是 `org.springframework.beans.factory.config.BeanPostProcessor` 接口。指示该 bean 的后处理器性质。否则，`ApplicationContext` 无法在完全创建之前按类型自动检测它。由于需要早期实例化 `BeanPostProcessor` 以便将其应用于上下文中的其他 Bean 初始化，因此这种早期类型检测至关重要。

以编程方式注册 `BeanPostProcessor` 实例

尽管建议的 `BeanPostProcessor` 注册方法是通过 `ApplicationContext` 自动检测（如前所述），但是您可以使用 `addBeanPostProcessor` 方法以编程方式针对 `ConfigurableBeanFactory` 注册它们。当您需要在注册之前评估条件逻辑，甚至需要跨层次结构的上下文复制 `Beanpost-processor` 时，这将非常有用。但是请注意，以编程方式添加的 `BeanPostProcessor` 实例不遵守 `Ordered` 接口。这样，注册的顺序决定了执行的顺序。还要注意，以编程方式注册的 `BeanPostProcessor` 实例始终在通过自动检测注册的 `BeanPostProcessor` 实例之前进行处理，而不考虑任何明确的顺序。

`BeanPostProcessor` 实例和 AOP 自动代理

实现 `BeanPostProcessor` 接口的类是特殊的，并且容器对它们的处理方式有所不同。作为 `ApplicationContext` 特殊启动阶段的一部分，所有 `BeanPostProcessor` 实例和它们直接引用的 Bean 都会在启动时实例化。接下来，以排序方式注册所有 `BeanPostProcessor` 实例，并将其应用于容器

中的所有其他 bean。因为 AOP 自动代理是作为 BeanPostProcessor 本身实现的，所以 BeanPostProcessor 实例或它们直接引用的 bean 都没有资格进行自动代理，因此也没有编织方面的内容。

对于任何此类 Bean，您都应该看到一条信息日志消息：Bean someBean 不适合所有 BeanPostProcessor 接口进行处理（例如：不具备自动代理功能）。



如果您通过使用自动装配或@Resource（可能会退回到自动装配）将 Bean 装配到 BeanPostProcessor 中，则 Spring 在搜索类型匹配的依赖项候选对象时可能会访问意外的 Bean，因此使它们不符合自动代理或其他种类的 bean post-processing。例如，如果您有一个用@Resource 注解的依赖项，其中字段或设置器名称不直接与 bean 的声明名称相对应，并且不使用 name 属性，那么 Spring 将访问其他 bean 以按类型匹配它们。

下面的示例演示如何在 ApplicationContext 中编写，注册和使用 BeanPostProcessor 实例。

例子：Hello World, BeanPostProcessor-style

第一个示例说明了基本用法。该示例显示了一个自定义 BeanPostProcessor 实现，该实现调用由容器创建的每个 bean 的 `toString()` 方法，并将结果字符串打印到系统控制台。

以下列表显示了自定义 BeanPostProcessor 实现类的定义：

Java

```
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;
    }
}
```

Kotlin

```
import org.springframework.beans.factory.config.BeanPostProcessor

class InstantiationTracingBeanPostProcessor : BeanPostProcessor {

    // simply return the instantiated bean as-is
    override fun postProcessBeforeInitialization(bean: Any, beanName: String): Any? {
        return bean // we could potentially return any object reference here...
    }

    override fun postProcessAfterInitialization(bean: Any, beanName: String): Any? {
        println("Bean '$beanName' created : $bean")
        return bean
    }
}
```

以下 bean 元素使用 `InstantiationTracingBeanPostProcessor`:

```
<?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>
```

注意，如何单纯定义 `InstantiationTracingBeanPostProcessor`。它甚至没有名称，并且因为它是 Bean，所以可以像注入其他任何 Bean 一样对其进行依赖注入。（前面的配置还定义了一个由 Groovy 脚本支持的 bean。Spring 动态语言支持在标题为 [Dynamic Language Support](#) 的一章中有详细介绍。）

下面的 Java 应用程序运行前面的代码和配置：

Java

```
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 = ctx.getBean("messenger", Messenger.class);
        System.out.println(messenger);
    }

}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = ClassPathXmlApplicationContext("scripting/beans.xml")
    val messenger = ctx.getBean<Messenger>("messenger")
    println(messenger)
}
```

前面的应用程序的输出类似于以下内容：

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

例子:RequiredAnnotationBeanPostProcessor

将回调接口或注解与自定义 `BeanPostProcessor` 实现结合使用是扩展 Spring IoC 容器的常用方法。Spring 的 `RequiredAnnotationBeanPostProcessor` 是一个示例，它是 Spring 发行版附带的 `BeanPostProcessor` 实现，可以确保标有（任意）注解的 bean 上的 JavaBean 属性实际上（被配置为）依赖注入了一个值。

1.8.2. 使用 BeanFactoryPostProcessor 自定义配置元数据

我们 要 看 的 下 一 个 扩 展 点 是 `org.springframework.beans.factory.config.BeanFactoryPostProcessor`。该接口的语义类似于 `BeanPostProcessor` 的语义，但有一个主要区别：`BeanFactoryPostProcessor` 对 Bean 配置元数据进行操作。也就是说，Spring IoC 容器允许 `BeanFactoryPostProcessor` 读取配置元数据，并有可能在容器实例化除 `beanFactoryPostProcessor` 实例以外的任何 Bean 之前更改它。

您可以配置多个 `BeanFactoryPostProcessor` 实例，并且可以通过设置 `order` 属性来控制这些 `BeanFactoryPostProcessor` 实例的运行顺序。但是，仅当 `BeanFactoryPostProcessor` 实现 `Ordered` 接口时，才可以设置此属性。如果您编写自己的 `BeanFactoryPostProcessor`，也应该考虑实现 `Ordered` 接口。有关更多详细信息，请参见 `BeanFactoryPostProcessor` 和 `Ordered` 接口的 javadoc。

如果要更改实际的 bean 实例（即从配置元数据创建的对象），则需要使用 `BeanPostProcessor`（在前面的 [使用 BeanPostProcessor 自定义 Bean](#) 中进行了介绍）。从技术上讲，可以在 `BeanFactoryPostProcessor` 中使用 Bean 实例（例如，通过使用 `BeanFactory.getBean()`），但是这样做会导致过早的 Bean 实例化，从而违反了标准容器的生命周期。这可能会导致负面影响，例如绕过 bean 后处理。



同样，`BeanFactoryPostProcessor` 实例是按容器划分作用域的。仅在使用容器层次结构时才有意义。如果在一个容器中定义 `BeanFactoryPostProcessor`，则仅将其应用于该容器中的 Bean 定义。一个容器中的 Bean 定义不会由另一个容器中的 `BeanFactoryPostProcessor` 实例进行后处理，即使两个容器都在同一层次结构也是如此。

Bean 工厂后处理器在 `ApplicationContext` 中声明时会自动运行，以便将更改应用于定义容器的配置元数据。Spring 包含许多预定义的 bean 工厂 post-processors，例如 `PropertyOverrideConfigurer` 和 `PropertySourcesPlaceholderConfigurer`。您也可以使用自定义 `BeanFactoryPostProcessor`，例如注册自定义属性编辑器。



与 `BeanPostProcessors` 一样，您通常不想配置 `BeanFactoryPostProcessors` 用于延迟初始化。如果没有其他 bean 引用 Bean (Factory) PostProcessor，则该后处理器将完全不会实例化。因此，将其标记为延迟初始化将被忽略，即使您在 `<beans/>` 标签的声明中将 `default-lazy-init` 属性设置为 `true`，Bean (Factory) PostProcessor 也会被实例化。

例子：类名替换 `PropertySourcePlaceholderConfiguration`

您可以使用 `PropertySourcesPlaceholderConfigurer` 来通过使用标准 Java 属性格式将豆定义中的属性值外部化到单独的文件中。这样做使部署应用程序的人员可以自定义特

定于环境的属性，例如数据库 URL 和密码，而不会为修改容器的一个或多个主要 XML 定义文件带来复杂性或风险。考虑以下基于 XML 的配置元数据片段，其中定义了具有占位符值的 DataSource：

```
<bean  
    class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">  
    <property name="locations" value="classpath:com/something/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>
```

该示例显示了从外部属性文件配置的属性。在运行时，将 PropertySourcesPlaceholderConfigurer 应用于替换数据源某些属性的元数据。将要替换的值指定为 \${property-name} 格式的占位符，该格式遵循 Ant 和 log4j 和 JSP EL 样式。

实际值来自标准 Java 属性格式的另一个文件：

```
jdbc.driverClassName=org.hsqldb.jdbcDriver  
jdbc.url=jdbc:hsqldb:hsq://production:9002  
jdbc.username=sa  
jdbc.password=root
```

因此，\${jdbc.username} 字符串在运行时将被替换为值“sa”，并且与属性文件中的键匹配的其他占位符也是如此。PropertySourcesPlaceholderConfigurer 检查 Bean 定义的大多数属性和属性中的占位符。此外，您可以自定义占位符前缀和后缀。

借助 Spring 2.5 中引入的上下文名称空间，您可以使用专用配置元素配置属性占位符。您可以在 location 属性中以逗号分隔列表的形式提供一个或多个位置，如以下示例所示：

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

PropertySourcesPlaceholderConfigurer 不仅在您指定的属性文件中查找属性 properties。默认情况下，如果无法在指定的属性文件中找到属性，则会检查 Spring Environment 属性和常规 Java System 属性。

您可以使用 `PropertySourcesPlaceholderConfigurer` 替换类名称，这在您必须在运行时选择特定的实现类时有时很有用。下面这个图讲了如何去做：



```
<bean  
    class="org.springframework.beans.factory.config.PropertySourcesPlaceholderConfigurer">  
    <property name="locations">  
        <value>classpath:com/something/strategy.properties</value>  
    </property>  
    <property name="properties">  
        <value>custom.strategy.class=com.something.DefaultStrategy</value>  
    </property>  
</bean>  
  
<bean id="serviceStrategy" class="${custom.strategy.class}" />
```

如果无法在运行时将该类解析为有效的类，则在将要创建该 bean 时（在非延迟初始化 bean 的 `ApplicationContext` 的 `preInstantiateSingletons()` 阶段期间），该 bean 的解析将失败。

例子：`PropertyOverrideConfigurer`

另一个 bean 工厂 post-processor 程序 `PropertyOverrideConfigurer` 类似于 `PropertySourcesPlaceholderConfigurer`，但是与后者不同，原始定义对于 bean 属性可以具有默认值或完全没有值。如果覆盖的属性文件没有某个 bean 属性的条目，则使用默认的上下文定义。

注意，bean 定义不知道会被覆盖，因此在覆写配置器正在被使用的 XML 定义文件中不能立即看出。在多个对同一个 bean 属性定义了不同值的 `PropertyOverrideConfigurer` 实例，由于覆写机制，只有最后一个能够生效。

属性配置按照下列格式：

```
beanName.property=value
```

下列列出一个例子：

```
dataSource.driverClassName=com.mysql.jdbc.Driver  
dataSource.url=jdbc:mysql:mydb
```

此示例文件可与包含一个名为 `dataSource` 的 bean 的容器定义一起使用，该 bean 具有驱动程序和 `url` 属性。

复合属性名也被支持，只要除了最终属性被覆写之外每个路径上的组件都已经为非空（大概是由构造函数初始化的）。在下面的示例中，`tom` bean 的 `fred` 属性的 `bob` 属性的 `sammy` 属性设置为标量值 123：

```
tom.fred.bob.sammy=123
```



指定的替代值始终是字面值。它们不会转换为 bean 引用。当 XML bean 定义中的原始值指定 bean 引用时，此约定也适用。

使用 Spring 2.5 中引入的上下文(context)名称空间，可以使用专用配置元素配置属性覆盖，如以下示例所示：

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

1.8.3. 使用 FactoryBean 自定义实例化逻辑

您可以以为本身就是工厂的对象实现 `org.springframework.beans.factory.FactoryBean` 接口。

`FactoryBean` 接口是可插入 Spring IoC 容器的实例化逻辑的一点。如果您有复杂的初始化代码，而不是（可能）冗长的 XML 量，可以用 Java 更好地表达，则可以创建自己的 `FactoryBean`，在该类中编写复杂的初始化，然后将自定义 `FactoryBean` 插入容器。

`FactoryBean` 接口提供了三个方法：

- `Object getObject()`：返回一个工厂创建的实例，并根据工厂返回的是不是 `Singleton` 或者 `prototype`，这个实力可能能被共享。
- `boolean isSingleton()`：如果工厂返回单例则返回 `true`，否则返回 `false`。
- `Class getObjectType()`：根据 `getObject()` 方法返回的对象返回类类型，或者位置的情况下返回 `null`。

Spring 框架中的许多地方都使用了 `FactoryBean` 概念和接口。它起码有 50 个实现。

当您需要向容器请求一个实际的 `FactoryBean` 实例本身而不是由它产生的 bean 时，请在调用 `ApplicationContext` 的 `getBean()` 方法时在该 bean 的 ID 前面加上一个 & 符号。所以对于给定的一个 id 为 `myBean` 的 `FactoryBean` 时，调用 `getBean("myBean")` 会得到 `FactoryBean` 的产品，如果调用 `getBean("&myBean")` 则返回 `FactoryBean` 实例本身。

1.9. 基于注解的容器配置

对于配置 Spring 注解比 XML 要好吗？

基于注释的配置的引入提出了一个问题，即这种方法是否比 XML“更好”。简短的答案是“各有所长”。长话短说，每种方法都有其优缺点，通常，由开发人员决定哪种策略更适合他们。由于定义方式的不同，注释在声明中提供了很多上下文，从而使配置更短，更简洁。但是，XML 擅长装配组件组件而不接触其源代码或重新编译它们。一些开发人员更喜欢将在源代码中装配类，而另一些开发人员则认为带注解的类不再是 POJO，而且，该配置变得分散且难以控制。

无论选择如何，Spring 都可以容纳两种样式，甚至可以将它们混合在一起。值得指出的是，通过其 [JavaConfig\(1.12\)](#) 选项，Spring 允许以非侵入方式使用注释，而无需接触目标组件的源代码，并且就工具而言，[Spring Tools for Eclipse](#) 支持所有配置样式。

基于注释的配置提供了 XML 设置的替代方法，该配置依赖字节码元数据来装配组件，而不是标签声明。通过使用相关类，方法或字段声明上的注解，开发人员无需使用 XML 来描述 bean 的装配，而是将配置转入组件类本身。如[示例：RequiredBeansProcessor\(1.8.2 上面\)](#)，结合使用 [BeanPostProcessor](#) 和注释，是扩展 Spring IoC 容器的常用方法。例如，Spring2.0 引入了使用@Required 注释强制执行必需属性的可能性。Spring2.5 使得可以遵循相同的通用方法来驱动 Spring 的依赖项注入。本质上，@Autowired 注解提供的功能与[自动装配协作器（1.4.5）](#) 中所述的功能相同，但具有更细粒度的控制和更广泛的适用性。Spring2.5 还添加了对 JSR-250 批注的支持，例如@PostConstruct 和@PreDestroy。Spring3.0 增加了对 [javax.inject](#) 包中包含的 JSR-330(Java 依赖注入)注释的支持，例如@Inject 和@Named。有关这些注释的详细信息，请参见[相关章节\(1.11\)](#)。



注释注入在 XML 注入之前执行。因此，通过两种方法装配的属性 XML 配置将覆盖注解。

与往常一样，您可以将它们注册为单独的 bean 定义，但是也可以通过在基于 XML 的 Spring 配置中包含以下标记来隐式注册它们（请注意包括 context 命名空间）：

```

<?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>

```

(隐式注册的 post-processors 包括 `AutowiredAnnotationBeanPostProcessor`, `CommonAnnotationBeanPostProcessor`, `PersistenceAnnotationBeanPostProcessor` 和前述的 `RequiredAnnotationBeanPostProcessor`。)

 `<context: annotation-config/>` 仅在定义它的相同应用程序上下文中查找关于 bean 的注解。就是说，如果将 `<context : annotation-config/>` 放在 `DispatcherServlet` 的 `WebApplicationContext` 中，它将仅检查控制器中的`@Autowired` bean，而不检查服务。

1.9.1. @Required 注解

`@Required` 批注适用于 bean 属性设置器方法，如以下示例所示：

Java

```

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}

```

Kotlin

```
class SimpleMovieLister {  
  
    @Required  
    lateinit var movieFinder: MovieFinder  
  
    // ...  
}
```

此注释指示必须在配置时通过 bean 定义中的显式属性值或通过自动装配来填充受影响的 bean 属性。如果受影响的 bean 属性尚未填充，则容器将引发异常。这允许急切和显式的失败，避免以后再出现 `NullPointerException` 实例等。我们仍然建议您将断言放入 bean 类本身中（例如，放入 `init` 方法中）。这样做会强制执行那些必需的引用和值，即使您在容器外部使用该类也是如此。



从 Spring Framework 5.1 开始，`@Required` 批注已正式弃用，转而使用构造函数注入进行必需的设置（或 `InitializingBean.afterPropertiesSet()` 的自定义实现以及 bean 属性 `setter` 方法）。

1.9.2. 使用`@Autowired`



在本节中的示例中，可以使用 JSR 330 的`@Inject` 注释代替 Spring 的`@Autowired` 注释。有关更多详细信息，请参见[此处\(1.11\)](#)。

您可以将`@Autowired` 注释应用于构造函数，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    private final CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender @Autowired constructor(  
    private val customerPreferenceDao: CustomerPreferenceDao)
```

从 Spring Framework 4.3 开始，如果目标 bean 仅定义一个构造函数作为开始，则不再需要在此类构造函数上使用 @Autowired 注解。但是，如果有多个构造函数可用，并且没有主/默认构造函数，则至少一个构造函数必须使用 @Autowired 注释，以指示容器使用哪个。有关详细信息，请参见有关[构造函数解析](#)的讨论。

您还可以将 @Autowired 注解应用于传统的 setter 方法，如以下示例所示：

Java

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Autowired  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired  
    lateinit var movieFinder: MovieFinder  
  
    // ...  
}
```

您还可以将注释应用于具有任意名称和多个参数的方法，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(MovieCatalog movieCatalog,  
                        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    private lateinit var movieCatalog: MovieCatalog  
  
    private lateinit var customerPreferenceDao: CustomerPreferenceDao  
  
    @Autowired  
    fun prepare(movieCatalog: MovieCatalog,  
               customerPreferenceDao: CustomerPreferenceDao) {  
        this.movieCatalog = movieCatalog  
        this.customerPreferenceDao = customerPreferenceDao  
    }  
  
    // ...  
}
```

您也可以将@Autowired 应用于字段，甚至将其与构造函数混合使用，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    private final CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    private MovieCatalog movieCatalog;  
  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender @Autowired constructor(  
    private val customerPreferenceDao: CustomerPreferenceDao) {  
  
    @Autowired  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

确保目标组件（例如，MovieCatalog 或 CustomerPreferenceDao）由用于@Autowired 注解的注入点的类型一致地声明。否则，注入可能会由于运行时出现“找不到类型匹配(notype match found)”错误而失败。



对于通过类路径扫描找到的 XML 定义的 bean 或组件类，容器通常预先知道具体的类型。但是，对于@Bean 工厂方法，您需要确保声明的返回类型足够明显。对于实现多个接口的组件或可能由其实现类型引用的组件，请考虑在工厂方法中声明最具体的返回类型（至少根据引用您的 bean 的注入点的要求进行声明）。

您还可以通过将@Autowired 注解添加到需要该类型数组的字段或方法中，指示 Spring 从 ApplicationContext 提供特定类型的所有 bean，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    private lateinit var movieCatalogs: Array<MovieCatalog>  
  
    // ...  
}
```

如以下示例所示，这同样适用于泛型集合：

Java

```
public class MovieRecommender {  
  
    private Set<MovieCatalog> movieCatalogs;  
  
    @Autowired  
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {  
        this.movieCatalogs = movieCatalogs;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var movieCatalogs: Set<MovieCatalog>  
  
    // ...  
}
```

你的目标 beans 可以实现 org.springframework.core.Ordered 接口或者使用 @Order 或者标准 @Priority 注解，如果你希望你的数组或列表中的项按照指定顺序排列。否则，他们的顺序将按照容器中相关目标 bean 的注册顺序进行。



您可以在目标类级别和 @Bean 方法上声明 @Order 注解，这可能适用于单个 bean 定义（如果使用同一 bean 类的多个定义）。@Order 值可能会影响注入点的优先级，但请注意它们不会影响单例启动顺序，这是由依赖关系和 @DependsOn 声明确定的正交关注点。

注意，标准 javax.annotation.Priority 注解在 @Bean 级别不可用，因为无法在方法上声明它。它的语义可以通过 @Order 值与 @Primary 结合在每种类型的单个 bean 上进行建模。

只要预期的键类型为 String，甚至输入类型的 Map 实例也可以自动装配。映射值包含所有预期类型的 bean，并且键包含相应的 bean 名称，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    private Map<String, MovieCatalog> movieCatalogs;  
  
    @Autowired  
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {  
        this.movieCatalogs = movieCatalogs;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var movieCatalogs: Map<String, MovieCatalog>  
  
    // ...  
}
```

默认情况下，当给定注入点没有匹配的候选 bean 可用时，自动装配将失败。对于声明的数组、集合或映射，至少应有一个匹配元素。

默认行为是将带注解的方法和字段视为指示所需的依赖项。您可以按照以下示例中所示

的方式更改此行为,从而使框架可以通过将其标记为不需要来跳过不满足要求的注入点(即,通过将@Autowired 中的 required 属性设置为 false) :

Java

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Autowired(required = false)  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired(required = false)  
    var movieFinder: MovieFinder? = null  
  
    // ...  
}
```

如果不需要的方法(或者在多个参数的情况下,其中一个依赖项)不可用,则根本不会调用该方法。在这种情况下,完全不需要填充非必需字段,而保留其默认值。

注入的构造函数和工厂方法参数是一种特殊情况,因为由于 Spring 的构造函数解析算法可能会处理多个构造函数,因此@Autowired 中的 required 属性的含义有所不同。默认情况下,有效地需要构造函数和工厂方法参数,但是在单构造函数场景中有一些特殊规则,例如,如果没有可用的匹配 bean,则多元素注入点(数组,集合,映射)解析为空实例。这允许一种通用的实现模式,其中所有依赖项都可以在唯一的多参数构造函数中声明-例如,声明为不带@Autowired 批注的单个公共构造函数。



对于任何给定 bean 类只有一个构造函数能声明 @Autowired，并将必选属性设置为 true，这表示在用作 Spring bean 时可以进行自动装配的构造函数。因此，如果必填属性保留为其默认值 `vtrue`，则 @Autowired 只能注释单个构造函数。如果多个构造函数声明了注解，则它们都必须声明 `required = false` 才能被视为自动装配的候选对象（类似于 XML 中的 `autowire=constructor`）。可以满足匹配 Spring 容器中 bean 的带有最多数量依赖项的构造函数将被选中。如果没有一个候选者满足要求，则将使用主/默认构造函数（如果存在）。同样，如果一个类声明了多个构造函数，但都没有使用 @Autowired 进行注释，则将使用主/默认构造函数（如果存在）。如果一个类只声明一个单一的构造函数开始，即使没有注释，也将始终使用它。请注意，带注解的构造函数不必是公共的。

建议在 setter 方法上使用 @Autowired 的 required 属性，而不推荐使用已弃用的 @Required 注解。将 required 属性设置为 false 表示该属性对于自动装配不是必需的，并且如果无法自动装配，则将忽略该属性。另一方面，@Required 更为强大，因为它可以通过容器支持的任何方式强制设置属性，并且如果未定义任何值，则会引发相应的异常。

另外，您可以通过 Java 8 的 `java.util.Optional` 表示特定依赖项的非必需性质，如以下示例所示：

```
public class SimpleMovieLister {  
  
    @Autowired  
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {  
        ...  
    }  
}
```

从 Spring Framework 5.0 开始，还可以用 @Nullable 注解（在任何包中任何形式，例如 JSR-305 中的 `javax.annotation.Nullable`）或仅利用 Kotlin 内置的 nullsafety 支持：

Java

```
public class SimpleMovieLister {  
  
    @Autowired  
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {  
        ...  
    }  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    @Autowired  
    var movieFinder: MovieFinder? = null  
  
    // ...  
}
```

您也可以将 `@Autowired` 用于众所周知的可解决依赖项的接口：`BeanFactory`, `ApplicationContext`, `Environment`, `ResourceLoader`, `ApplicationEventPublisher` 和 `MessageSource`. 这些接口及其扩展接口（例如 `ConfigurableApplicationContext` 或 `ResourcePatternResolver`）将自动解析，而无需进行特殊设置。下面的示例自动装配 `ApplicationContext` 对象：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    lateinit var context: ApplicationContext  
  
    // ...  
}
```

`@Autowired`, `@Inject`, `@Value` 和 `@Resource` 注解由 Spring `BeanPostProcessor` 实现处理。这意味着您不能在自己的 `BeanPostProcessor` 或 `BeanFactoryPostProcessor` 类型（如果有）中应用这些注解。必须使用 XML 或 Spring `@Bean` 方法显式“装配”这些类型。

1.9.3. 使用 @Primary 基于注解的自动装配的调整

由于按类型自动装配可能会导致多个候选对象，因此通常有必要对选择过程进行更多控制。实现此目标的一种方法是使用 Spring 的 `@Primary` 注解。`@Primary` 表示当多个 bean 是自动装配到单值依赖项的候选对象时，应优先考虑特定的 bean。如果候选中恰好存在一个主 bean，则它将成为自动装配的值。

考虑以下将 `firstMovieCatalog` 定义为主要 `MovieCatalog` 的配置：

Java

```
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

Kotlin

```
@Configuration
class MovieConfiguration {

    @Bean
    @Primary
    fun firstMovieCatalog(): MovieCatalog { ... }

    @Bean
    fun secondMovieCatalog(): MovieCatalog { ... }

    // ...
}
```

通过前面的配置，以下 `MovieRecommender` 与 `firstMovieCatalog` 自动装配：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    private MovieCatalog movieCatalog;  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

相关 bean 定义如下：

```
<?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. 使用 Qualifiers 基于注解的自动装配的调整

当可以确定一个主要候选对象时，`@Primary` 是在几种情况下按类型使用自动装配的有效方法。当您需要对选择过程进行更多控制时，可以使用 Spring 的`@Qualifier` 批注。您可以将限定符值与特定的参数相关联，从而缩小类型匹配的范围，以便为每个参数选择特定的 bean。在最简单的情况下，这可以是简单的描述性值，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    @Qualifier("main")  
    private lateinit var movieCatalog: MovieCatalog  
  
    // ...  
}
```

您还可以在各个构造函数参数或方法参数上指定**@Qualifier** 注解，如以下示例所示：

Java

```
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;  
    }  
  
    // ...  
}
```

```

class MovieRecommender {

    private lateinit var movieCatalog: MovieCatalog

    private lateinit var customerPreferenceDao: CustomerPreferenceDao

    @Autowired
    fun prepare(@Qualifier("main") movieCatalog: MovieCatalog,
               customerPreferenceDao: CustomerPreferenceDao) {
        this.movieCatalog = movieCatalog
        this.customerPreferenceDao = customerPreferenceDao
    }

    // ...
}

```

下面的例子展示了相关 bean 的定义：

```

<?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"/> ①

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/> ②

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

```

① 值为 `main` 的 Bean 与限定有相同值的构造函数参数关联。

② 值为 `action` 的 Bean 与限定有相同值的构造函数参数关联。

对于 `fallback` 匹配，`bean` 名称被视为默认的限定符值。因此，可以定义一个 `id` 为 `main` 而不是嵌套的 `qualifier` 元素定义 `bean`，从而得到相同的匹配结果。尽管可以使用此约定按名称引用特定的 `bean`，但 `@Autowired` 基本上是关于带有可选语义限定符的类型

驱动的注入。这意味着，即使带有 Bean 名称 `fallback` 的限定符值，在类型匹配集中也始终具有狭窄的语义。它们没有在语义上表示对唯一 `bean id` 的引用。好的限定符值是主要的或 EMEA 的或持久的，表示特定组件的特性，该特性独立于 `bean id`，在使用匿名 bean 定义（例如上例中的定义）的情况下，可以自动生成这些特性。

限定符还适用于泛型集合，如前面所述（例如，应用于 `Set<MovieCatalog>`）。在这种情况下，根据声明的限定符，将所有匹配的 bean 作为集合注入。这意味着限定词不必是唯一的。相反，它们构成了过滤标准。例如，您可以使用相同的限定符值“`action`”定义多个 `MovieCatalog` Bean，所有这些都注入到注有 `@Qualifier("action")` 的 `Set<MovieCatalog>` 中。



在类型匹配的候选对象中，让限定符值根据目标 Bean 名称进行选择，在注入点不需要 `@Qualifier` 注释。如果没有其他解决方案指示符（例如限定符或主标记），则对于非唯一依赖性情况，Spring 将注入点名称（即字段名称或参数名称）与目标 Bean 名称进行匹配，然后选择同名候选（如果有）。

就是说，如果您打算按名称表示注释驱动的注入，则即使它能够在类型匹配的候选对象中按 bean 名称进行选择，也不要主要使用 `@Autowired`。而是使用 JSR-250 `@Resource` 注解，该批注的语义定义是通过其唯一名称标识特定目标组件，而声明的类型与匹配过程无关。`@Autowired` 具有不同的语义：在按类型选择候选 bean 之后，仅在那些类型选择的候选中考虑指定的 String 限定符值（例如，将 `account` 限定符与标记有相同限定符标签的 bean 进行匹配）。

对于本身定义为集合、`Map` 或数组类型的 bean，`@Resource` 是一个很好的解决方案，通过唯一名称引用特定的集合或数组 bean。就是说，从 4.3 版本开始，只要元素类型信息保留在 `@Bean` 返回类型签名或集合继承层次结构中，就可以通过 Spring 的 `@Autowired` 类型匹配算法来匹配 `Map` 和数组类型。在这种情况下，您可以使用限定符值在同类型的集合中进行选择，如上一段所述。

从 4.3 开始，`@Autowired` 还考虑了自我引用以进行注入（即，引用回当前注入的 Bean）。请注意，自我注入是一个 fallback。对其他组件的常规依赖始终优先。从这个意义上说，自我引用不参与常规的候选选择，因此尤其是绝不是主要的。相反，它们总是以最低优先级结束。实际上，应该仅将自我引用用作最后的手段（例如，通过 Bean 的事务代理在同一实例上调用其他方法）。在这种情况下，请考虑将受影响的方法分解为单独的委托 bean。

或者，您可以使用`@Resource`，它可以通过其唯一名称获取返回到当前 bean 的代理。



尝试将`@Bean` 方法的结果注入相同的配置类也实际上是一种自引用方案。要么在实际需要的方法签名中懒解析此类引用（与配置类中的自动装配字段相对），要么将受影响的`@Bean` 方法声明为静态，将其与包含的配置类实例及其生命周期脱钩。否则，仅在回退阶段考虑此类 Bean，而将其他配置类上的匹配 Bean 选作主要候选对象（如果可用）。

`@Autowired` 适用于字段，构造函数和多参数方法，从而允许在参数级别缩小限定符注解的范围。相反，只有具有单个参数的字段和 bean 属性设置器方法才支持`@Resource`。因此，如果注入目标是构造函数或多参数方法，则应坚持使用限定符。

您可以创建自己的自定义限定符注释。为此，请定义一个注释并在定义中提供`@Qualifier` 注释，如以下示例所示：

Java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {

    String value();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Genre(val value: String)
```

然后，您可以在自动连接的字段和参数上提供自定义限定符，如以下示例所示：

Java

```
public class MovieRecommender {  
  
    @Autowired  
    @Genre("Action")  
    private MovieCatalog actionCatalog;  
  
    private MovieCatalog comedyCatalog;  
  
    @Autowired  
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {  
        this.comedyCatalog = comedyCatalog;  
    }  
  
    // ...  
}
```

Kotlin

```
class MovieRecommender {  
  
    @Autowired  
    @Genre("Action")  
    private lateinit var actionCatalog: MovieCatalog  
  
    private lateinit var comedyCatalog: MovieCatalog  
  
    @Autowired  
    fun setComedyCatalog(@Genre("Comedy") comedyCatalog: MovieCatalog) {  
        this.comedyCatalog = comedyCatalog  
    }  
  
    // ...  
}
```

接下来，您可以提供有关候选 bean 定义的信息。您可以将<qualifier/>标记添加为<bean/>标记的子元素，然后指定类型和值以匹配您的自定义限定符注释。该类型与注解的完全限定的类名匹配。另外，为方便起见，如果不存在名称冲突的风险，则可以使用简短的类名。下面的示例演示了两种方法：

```

<?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>

```

在“[类路径扫描和托管组件\(1.10\)](#)”中，您可以看到基于注释的替代方法，以 XML 形式提供限定符元数据。具体来说，请参阅[为限定符元数据提供注解\(1.10.8\)](#)。

在某些情况下，使用没有值的注释就足够了。当注解用于更一般的用途并且可以应用于几种不同类型的依赖项时，这将很有用。例如，您可以提供一个离线目录，当没有 Internet 连接可用时，可以对其进行搜索。首先，定义简单的注释，如以下示例所示：

Java

```

@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {
}

```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class Offline
```

然后将注释添加到要自动装配的字段或属性，如以下示例所示：

Java

```
public class MovieRecommender {

    @Autowired
    @Offline ①
    private MovieCatalog offlineCatalog;

    // ...
}
```

① 这行加入`@Offline` 注解

Kotlin

```
class MovieRecommender {

    @Autowired
    @Offline ①
    private lateinit var offlineCatalog: MovieCatalog

    // ...
}
```

① 这行加入`@Offline` 注解

现在 bean 的定义只需要一个先定了，如下列所展示的：

```
<bean class="example.SimpleMovieCatalog">
    <qualifier type="Offline"/> ①
    <!-- inject any dependencies required by this bean -->
</bean>
```

① 这个元素指定了一个限定符

您还可以定义自定义限定符注解，该注解除了简单值属性之外或代替简单值属性，都接受命名属性。如果随后在要自动装配的字段或参数上指定了多个属性值，则 bean 定义必须与所有此类属性值匹配才能被视为自动装配候选。例如，请看以下注释定义：

Java

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier {

    String genre();

    Format format();
}
```

Kotlin

```
@Target(AnnotationTarget.FIELD, AnnotationTarget.VALUE_PARAMETER)
@Retention(AnnotationRetention.RUNTIME)
@Qualifier
annotation class MovieQualifier(val genre: String, val format: Format)
```

在这种情况下，**Format** 是一个枚举，定义如下：

Java

```
public enum Format {
    VHS, DVD, BLURAY
}
```

Kotlin

```
enum class Format {
    VHS, DVD, BLURAY
}
```

要自动装配的字段将用自定义限定符进行注解，并包括两个属性的值：体裁和格式，如以下示例所示：

Java

```
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;

    // ...
}
```

Kotlin

```
class MovieRecommender {

    @Autowired
    @MovieQualifier(format = Format.VHS, genre = "Action")
    private lateinit var actionVhsCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.VHS, genre = "Comedy")
    private lateinit var comedyVhsCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.DVD, genre = "Action")
    private lateinit var actionDvdCatalog: MovieCatalog

    @Autowired
    @MovieQualifier(format = Format.BLURAY, genre = "Comedy")
    private lateinit var comedyBluRayCatalog: MovieCatalog

    // ...
}
```

最后，bean 定义应包含匹配的限定符值。此示例还演示了可以使用 bean 元属性代替 `<qualifier/>` 元素。如果可用，`<qualifier/>` 元素及其属性优先，但是如果不存在这样的限定符，则自动装配机制将退回到`<meta/>` 标记内提供的值，如以下示例中的最后两个 bean 定义：

```

<?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. 使用泛型作为自动装配限定符

除了@Qualifier注解之外，您还可以将Java泛型类型用作限定符的隐式形式。例如，假设您具有以下配置：

Java

```
@Configuration  
public class MyConfiguration {  
  
    @Bean  
    public StringStore stringStore() {  
        return new StringStore();  
    }  
  
    @Bean  
    public IntegerStore integerStore() {  
        return new IntegerStore();  
    }  
}
```

Kotlin

```
@Configuration  
class MyConfiguration {  
  
    @Bean  
    fun stringStore() = StringStore()  
  
    @Bean  
    fun integerStore() = IntegerStore()  
}
```

假设前面的 bean 实现了通用接口（即 `Store <String>` 和 `Store <Integer>`），则可以 `@Autowired Store` 接口，并且泛型被用作了一个限定符，如以下示例所示：

Java

```
@Autowired  
private Store<String> s1; // <String> qualifier, injects the stringStore bean  
  
@Autowired  
private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean
```

Kotlin

```
@Autowired  
private lateinit var s1: Store<String> // <String> qualifier, injects the stringStore  
bean  
  
@Autowired  
private lateinit var s2: Store<Integer> // <Integer> qualifier, injects the  
integerStore bean
```

当自动装配列表，Map 实例和数组是，泛型限定符也可以使用，比如下面的例子自动装配了一个泛型列表 List：

Java

```
// 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;
```

Kotlin

```
// Inject all Store beans as long as they have an <Integer> generic  
// Store<String> beans will not appear in this list  
@Autowired  
private lateinit var s: List<Store<Integer>>
```

1.9.6. 使用 CustomAutoWireConfigurer

CustomAutoWireConfigurer 是 BeanFactoryPostProcessor，即使您没有使用 Spring 的 @Qualifier 注解来注解自己的自定义限定符注释类型，也可以使用它来注册。以下示例显示如何使用 CustomAutoWireConfigurer：

```
<bean id="customAutoWireConfigurer"  
      class="org.springframework.beans.factory.annotation.CustomAutoWireConfigurer">  
  <property name="customQualifierTypes">  
    <set>  
      <value>example.CustomQualifier</value>  
    </set>  
  </property>  
</bean>
```

AutowireCandidateResolver 通过以下方式确定自动依赖注入：

- 每个 bean 定义的 autowire-candidate 值
- <beans/> 元素上 default-autowire-candidates 模式
- @Qualifier 注解以及在 CustomAutoWireConfigurer 中注册的所有自定义注解的存储在

当多个 bean 符合自动装配候选条件时，“primary”的确定如下：如果候选对象中恰好有一个 bean 定义的 `primary` 属性设置为 `true`，则将其选中。

1.9.7. 使用`@Resource` 注入

Spring 还通过对字段或 bean 属性 `setter` 方法使用 JSR-250 `@Resource` 注解 (`javax.annotation.Resource`) 支持注入。这是 Java EE 中的常见模式：例如，在 JSF 管理的 Bean 和 JAX-WS 端点中。Spring 也为 Spring 管理的对象支持此模式。

`@Resource` 具有 `name` 属性。默认情况下，Spring 将该值解释为要注入的 Bean 名称。换句话说，它遵循 `name` 语义，如以下示例所示：

Java

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder") ①  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

① 这行插入一个`@Resource`。

Kotlin

```
class SimpleMovieLister {  
  
    @Resource(name="myMovieFinder") ①  
    private lateinit var movieFinder:MovieFinder  
}
```

① 这行插入一个`@Resource`。

如果未明确指定名称，则默认名称是从字段名称或 `setter` 方法派生的。如果是字段，则采用字段名称。在使用 `setter` 方法的情况下，它采用 bean 属性名称。以下示例将把名为 `movieFinder` 的 bean 注入其 `setter` 方法中：

Java

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
  
    @Resource  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    @Resource  
    private lateinit var movieFinder: MovieFinder  
}
```



配置注解的 name 通过 `CommonAnnotationBeanPostProcessor` 的 `ApplicationContext` 发现并解析为 Bean 名称。如果您明确配置 Spring 的 `SimpleJndiBeanFactory`, 则可以通过 JNDI 解析名称。但是, 我们建议您依赖默认行为, 并使用 Spring 的 JNDI 查找功能来保留间接级别。

在`@Resource`用法(未指定显式名称)的特殊情况下, 类似于`@Autowired`, `@Resource`查找主类型匹配而不是特定的命名 Bean, 并解析众所周知的可解决依赖项: `BeanFactory`, `ApplicationContext`, `ResourceLoader`, `ApplicationEventPublisher` 和 `MessageSource` 接口。

因此, 在以下示例中, `customerPreferenceDao` 字段首先查找名为“`customerPreferenceDao`”的 bean, 然后回退到类型为 `CustomerPreferenceDao` 的 primary 类型匹配项:

Java

```
public class MovieRecommender {  
  
    @Resource  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Resource  
    private ApplicationContext context; ①  
  
    public MovieRecommender() {  
    }  
  
    // ...  
}
```

① `context` 字段是基于已知的可解决依赖类型：`ApplicationContext` 注入的。

Kotlin

```
class MovieRecommender {  
  
    @Resource  
    private lateinit var customerPreferenceDao: CustomerPreferenceDao  
  
    @Resource  
    private lateinit var context: ApplicationContext ①  
  
    // ...  
}
```

① `context` 字段是基于已知的可解决依赖类型：`ApplicationContext` 注入的。

1.9.8. 使用@Value

`@Value` 通常用于注入外部属性：

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("${catalog.name}") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name}") private val catalog: String)
```

伴随下列配置：

Java

```
@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig { }
```

Kotlin

```
@Configuration
@PropertySource("classpath:application.properties")
class AppConfig
```

还有下列 application.properties 文件：

```
catalog.name=MovieCatalog
```

Spring 提供了一个默认的宽松内嵌值解析器。 它将尝试解析属性值，如果无法解析，则将属性名称（例如`$ {catalog.name}`）作为值注入。如果要严格控制不存在的值，则应声明一个 `PropertySourcesPlaceholderConfigurer` bean，如以下示例所示：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public static PropertySourcesPlaceholderConfigurer  
propertyPlaceholderConfigurer() {  
        return new PropertySourcesPlaceholderConfigurer();  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun propertyPlaceholderConfigurer() = PropertySourcesPlaceholderConfigurer()  
}
```



使用 `JavaConfig` 配置 `PropertySourcesPlaceholderConfigurer` 时，`@bean` 方法必须为静态。

如果任何`${}`占位符都没被解析出来的话，`name` 使用上述的配置会使 Spring 初始化失败。也可以使用 `setPlaceholderPrefix`，`setPlaceholderSuffix` 或 `setValueSeparator` 之类的方法来自定义占位符。



Spring boot 默认配置一个 `PropertySourcesPlaceholderConfigurer` bean 时，将会从 `application.properties` 和 `application.yml` 文件中获取属性值。

Spring 提供的内置转换器支持允许自动处理简单的类型转换（例如，转换为 `Integer` 或 `int`）。多个逗号分隔的值可以自动转换为 `String` 数组，而无需付出额外的努力。

可以提供如下默认值：

Java

```
@Component  
public class MovieRecommender {  
  
    private final String catalog;  
  
    public MovieRecommender(@Value("${catalog.name:defaultCatalog}") String catalog) {  
        this.catalog = catalog;  
    }  
}
```

Kotlin

```
@Component
class MovieRecommender(@Value("\${catalog.name:defaultCatalog}") private val catalog:
String)
```

Spring `BeanPostProcessor` 在后台使用 `ConversionService` 处理将`@Value` 中的 `String` 值转换为目标类型的过程。如果要为自己的自定义类型提供转换支持，则可以提供自己的 `ConversionService` bean 实例，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean
    public ConversionService conversionService() {
        DefaultFormattingConversionService conversionService = new
DefaultFormattingConversionService();
        conversionService.addConverter(new MyCustomConverter());
        return conversionService;
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun conversionService(): ConversionService {
        return DefaultFormattingConversionService().apply {
            addConverter(MyCustomConverter())
        }
    }
}
```

当`@Value` 包含 SpEL 表达式(第 4 章)时，该值将在运行时动态计算，如以下示例所示：

Java

```
@Component
public class MovieRecommender {

    private final String catalog;

    public MovieRecommender(@Value("#{systemProperties['user.catalog'] + 'Catalog' }") String catalog) {
        this.catalog = catalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{systemProperties['user.catalog'] + 'Catalog' }") private val catalog: String)
```

SpEL 还支持使用更复杂的数据结构：

Java

```
@Component
public class MovieRecommender {

    private final Map<String, Integer> countOfMoviesPerCatalog;

    public MovieRecommender(
        @Value("#{{'Thriller': 100, 'Comedy': 300}}") Map<String, Integer> countOfMoviesPerCatalog) {
        this.countOfMoviesPerCatalog = countOfMoviesPerCatalog;
    }
}
```

Kotlin

```
@Component
class MovieRecommender(
    @Value("#{{'Thriller': 100, 'Comedy': 300}}") private val countOfMoviesPerCatalog: Map<String, Int>)
```

1.9.9. 使用@PostConstruct 和@PreDestroy

CommonAnnotationBeanPostProcessor 不仅可以识别@Resource 注解，还可以识别 JSR-250 生命周期注解：javax.annotation.PostConstruct 和 javax.annotation.PreDestroy。在 Spring 2.5 中引入的对这些注解的支持提供了初始

化回调和销毁回调中描述的生命周期回调机制的替代方法。假设 `CommonAnnotationBeanPostProcessor` 在 `Spring ApplicationContext` 中注册，则在生命周期中与相应的 Spring 生命周期接口方法或显式声明的回调方法在同一点调用带有这些注解之一的方法。在以下示例中，缓存在初始化时预先填充，并在销毁时清除：

Java

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

Kotlin

```
class CachingMovieLister {  
  
    @PostConstruct  
    fun populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    fun clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

有关组合各种生命周期机制的效果的详细信息，请参见[结合生命周期机制\(69页\)](#)。



与 `@Resource` 一样，`@PostConstruct` 和 `@PreDestroy` 批注解类型是从 JDK 6 到 8 的标准 Java 库的一部分。但是，整个 `javax.annotation` 包与 JDK 9 中的核心 Java 模块分离，并最终在 JDK 11 中删除。如果需要，现在需要通过 Maven Central 获取 `javax.annotation-api` 工件，只需像其他任何库一样将其添加到应用程序的类路径中即可。

1.10. 类路径扫描和托管组件

本章中的大多数示例都使用 XML 来指定在 Spring 容器中生成每个 `BeanDefinition` 的配置元数据。上一节（[基于注解的容器配置](#)）演示了许多如何通过源码层面的注解提供配

置元数据。但是，即使在这些示例中，“base” bean 定义也已在 XML 文件中明确定义，而注解仅驱动依赖项注入。本节介绍了通过扫描类路径来隐式检测候选组件的选项。候选组件是与过滤条件匹配的类，并在容器中注册了相应的 Bean 定义。这消除了使用 XML 进行 bean 注册的需要。而是可以使用注解（例如，`@Component`），AspectJ 类型表达式或您自己的自定义过滤条件来选择哪些类已向容器注册了 bean 定义。



从 Spring 3.0 开始，Spring JavaConfig 项目提供的许多功能是 Spring Framework 核心的一部分。这使您可以使用 Java 而不是使用传统的 XML 文件来定义 bean。请查看[@Configuration](#), [@Bean](#), [@Import](#) 和 [@DependsOn](#) 注解，以获取有关如何使用这些新特性的示例。

1.10.1. `@Component` 以及更多的原型注释

`@Repository` 注解是任何实现数据库的角色或原型（也称为数据访问对象或 DAO）的类的标记。这些标记的使用将获自动转化异常，详情见[异常转换](#)的描述。

Spring 提供了进一步的原型注解：`@Component`, `@Service` 和 `@Controller`。
`@Component` 是任何 Spring 托管组件的通用原型。`@Repository`, `@Service` 和 `@Controller` 是`@Component` 的特定例，用于更特定的用例（分别在持久层，服务层和表示层中）。因此，您可以使用`@Component` 来注解标识组件类，但是通过使用`@Repository`, `@Service` 或 `@Controller` 注解来标识组件类，您的类更适合被工具或与相关联的切面处理。例如，这些原型注解成为切入点的理想目标。`@Repository`, `@Service` 和 `@Controller` 在 Spring 框架的将来版本中也可以带有其他语义。因此，如果在服务层使用`@Component` 或 `@Service` 之间进行选择，则`@Service` 显然是更好的选择。同样，如前所述，`@Repository` 已被支持作为持久层中自动异常转换的标记。

1.10.2. 使用元注解和合成注解

Spring 提供的许多注解都可以在您自己的代码中用作元注解。元注解是可以应用于另一个注解的注解。例如，前面提到的`@Service` 注解使用`@Component` 进行元注解，如以下示例所示：

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component ①
public @interface Service {

    // ...
}
```

① Component 导致@Service 的处理方式与@Component 相同。

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Component ①
annotation class Service {

    // ...
}
```

① Component 导致@Service 的处理方式与@Component 相同。

您还可以组合元注释来创建“组合注解”。例如，Spring MVC 中的@RestController 注解由@Controller 和@ResponseBody 组成。

此外，组合注解可以选择从元注解中重新声明属性，以允许自定义。当您只希望公开元注解属性的子集时，此功能特别有用。例如，Spring 的@SessionScope 注解将作用域名称硬编码为 session，但仍允许自定义 proxyMode。以下列表显示了 SessionScope 注解的定义：

Java

```
@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;

}
```

Kotlin

```
@Target(AnnotationTarget.TYPE, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Scope(WebApplicationContext.SCOPE_SESSION)
annotation class SessionScope(
    @get:AliasFor(annotation = Scope::class)
    val proxyMode: ScopedProxyMode = ScopedProxyMode.TARGET_CLASS
)
```

然后，您可以使用`@SessionScope` 而不用声明如下的 `proxyMode`：

Java

```
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

Kotlin

```
@Service
@SessionScope
class SessionScopedService {
    // ...
}
```

您还可以覆盖 `proxyMode` 的值，如以下示例所示：

Java

```
@Service  
 @SessionScope(proxyMode = ScopedProxyMode.INTERFACES)  
 public class SessionScopedUserService implements UserService {  
     // ...  
 }
```

Kotlin

```
@Service  
 @SessionScope(proxyMode = ScopedProxyMode.INTERFACES)  
 class SessionScopedUserService : UserService {  
     // ...  
 }
```

有关更多详细信息，请参见 [Spring Annotation programming model](#) Wiki 页面。

1.10.3. 自动扫描类和注册 Bean 定义

Spring 可以自动检测原型类，并向 [ApplicationContext](#) 注册相应的 [BeanDefinition](#) 实例。例如，以下两个类别能被自动检测：

Java

```
@Service  
 public class SimpleMovieLister {  
  
     private MovieFinder movieFinder;  
  
     public SimpleMovieLister(MovieFinder movieFinder) {  
         this.movieFinder = movieFinder;  
     }  
 }
```

Kotlin

```
@Service  
 class SimpleMovieLister(private val movieFinder: MovieFinder)
```

Java

```
@Repository  
 public class JpaMovieFinder implements MovieFinder {  
     // implementation elided for clarity  
 }
```

Kotlin

```
@Repository  
class JpaMovieFinder : MovieFinder {  
    // implementation elided for clarity  
}
```

要自动检测这些类并注册相应的 bean，需要将 @ComponentScan 添加到 @Configuration 类中，其中 basePackages 属性是这两个类的公共父包。（或者，您可以指定一个逗号分隔，分号分隔或空格分隔的列表，其中包括每个类的父包。）

Java

```
@Configuration  
@ComponentScan(basePackages = "org.example")  
public class AppConfig {  
    // ...  
}
```

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"])  
class AppConfig {  
    // ...  
}
```



为简便起见，前面的示例可能使用了注解的 value 属性（即，@ComponentScan("org.example")）。

以下替代方法使用 XML：

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns: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>
```



`<context:component-scan>` 的 使用 隐 式 启 用
`<context:annotation-config>`。当使用 `<context:component-scan>` 的时候一般不需要启用 `<context:annotation-config>`。



扫描类路径包需要在类路径中存在相应的目录条目。使用 Ant 构建 JAR 时，请确保未激活 JAR 任务的仅文件。此外，在某些环境中，基于安全策略可能不会公开类路径目录。例如，JDK1.7.0_45 及更高版本上的独立应用程序。（这需要在清单中设置“TrustedLibrary”请参见 <https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>）。

在 JDK 9 的模块路径（Jigsaw）上，Spring 的类路径扫描通常可以按预期进行。但是，请确保将组件类导出到模块信息描述符中。如果您希望 Spring 调用类的非公共成员，请确保它们是“opened”（也就是说，它们在模块信息描述符中使用了 opens 声明而不是 export 声明）。

此外，当您使用 component-scan 元素时，将隐式包括 `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor`。这意味着两个组件将被自动扫描并装配在一起，而所有这些都不需要 XML 中提供任何 bean 配置元数据。



您可以禁用注册 `AutowiredAnnotationBeanPostProcessor` 和 `CommonAnnotationBeanPostProcessor` 通过包含带有 false 值的注释配置属性。

1.10.4. 使用过滤器自定义扫描

默认情况下，仅使用 `@Component`, `@Repository`, `@Service`, `@Controller`, `@Configuration` 进行注释的类或使用 `@Component` 进行注解的自定义注解是唯一检测到的候选组件。但是，您可以通过应用自定义过滤器来修改和扩展此行为。将它们添加为 `@ComponentScan` 注解的 `includeFilters` 或 `excludeFilters` 属性（或作为 XML 配置中 `<context:component-scan>` 元素的 `<context:include-filter/>` 或 `<context:exclude-filter />` 子元素）。每个过滤器元素都需要类型和表达式属性。下表描述了过滤选项：

Table 5. Filter Types

Filter Type	Example Expression	Description
annotation (default)	<code>org.example.SomeAnnotation</code>	An annotation to be <i>present</i> or <i>meta-present</i> at the type level in target components.
assignable	<code>org.example.SomeClass</code>	A class (or interface) that the target components are assignable to (extend or implement).

Filter Type	Example Expression	Description
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 <code>org.springframework.core.type.TypeFilter</code> interface.

以下示例显示了忽略所有`@Repository`注解并改为使用“存根”库的配置：

Java

```
@Configuration
@ComponentScan(basePackages = "org.example",
    includeFilters = @Filter(type = FilterType.REGEX, pattern =
".*Stub.*Repository"),
    excludeFilters = @Filter(Repository.class))
public class AppConfig {
    ...
}
```

Kotlin

```
@Configuration
@ComponentScan(basePackages = "org.example",
    includeFilters = [Filter(type = FilterType.REGEX, pattern =
[".*Stub.*Repository"])],
    excludeFilters = [Filter(Repository::class)])
class AppConfig {
    // ...
}
```

下方列表展示了等效的 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>
```



您也可以通过在注解上设置 `useDefaultFilters=false` 或通过将 `use-default-filters="false"` 作为 `<component-scan/>` 元素的属性来禁用默认过滤器。这有效地禁用了对使用 `@Component`, `@Repository`, `@Service`, `@Controller`, `@RestController` 或 `@Configuration` 进行注释或元注释的类的自动检测。

1.10.5. 在组件中定义 Bean 元数据

Spring 组件还可以将 bean 定义元数据贡献给容器。您可以使用与 `@Bean` 注解类相同的 `@Configuration` 注解来定义 Bean 元数据。以下示例显示了如何执行此操作：

Java

```
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    fun doWork() {
        // Component method implementation omitted
    }
}
```

上面的类是一个，在其 `doWork()` 方法中具有特定于应用程序的代码的 Spring 组件。但是，它也提供了一个 bean 定义，该定义具有引用方法 `publicInstance()` 的工厂方法。`@Bean` 注解标识工厂方法和其他 bean 定义属性，例如通过 `@Qualifier` 注解的限定符值。可以指定的其他方法层级的注解是 `@Scope`, `@Lazy` 和自定义限定符注解。



除了用于组件初始化的角色外，还可以将注解放置在标有 `@Autowired` 或 `@Inject` 的注入点上。在这种情况下，`@Lazy` 会导致注入惰性解析代理。如前所述，支持自动装配的字段和方法，并自动装配`@Bean` 方法。以下示例显示了如何执行此操作：

Java

```
@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);
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    companion object {
        private var i: Int = 0
    }

    @Bean
    @Qualifier("public")
    fun publicInstance() = TestBean("publicInstance")

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected fun protectedInstance(
        @Qualifier("public") spouse: TestBean,
        @Value("#{privateInstance.age}") country: String) =
    TestBean("protectedInstance", 1).apply {
        this.spouse = spouse
        this.country = country
    }

    @Bean
    private fun privateInstance() = TestBean("privateInstance", i++)

    @Bean
    @RequestScope
    fun requestScopedInstance() = TestBean("requestScopedInstance", 3)
}
```

该示例将 `String` 方法参数 `country` 自动装配到另一个名为 `privateInstance` 的 bean 上 `age` 属性的值。Spring Expression Language 元素通过符号 `# {<expression>}` 定义属性的值。对于 `@Value` 注解，表达式解析程序已预先配置为在解析表达式文本时查找 bean 名称。

从 Spring Framework 4.3 开始，您还可以声明类型为 `InjectionPoint` 的工厂方法参数（或更具体的子类：`DependencyDescriptor`），以访问触发当前 bean 创建的请求注入点。注意，这仅适用于实际创建 bean 实例，而不适用于注入现有实例。因此，此功能对原型范围的 bean 最有意义。对于其他作用域，`factory` 方法仅看到在给定作用域中触发创建新 bean 实例的注入点（例如，触发创建惰性单例 bean 的依赖项）。在这种情况下，可以将提供的注入点元数据与语义一起使用。以下示例显示如何使用 `InjectionPoint`：

Java

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember());
    }
}
```

Kotlin

```
@Component
class FactoryMethodComponent {

    @Bean
    @Scope("prototype")
    fun prototypeInstance(injectionPoint: InjectionPoint) =
        TestBean("prototypeInstance for ${injectionPoint.member}")
}
```

常规 Spring 组件中的 `@Bean` 方法的处理方式与 Spring `@Configuration` 类中的 `@Bean` 方法不同。区别在于，使用 CGLIB 不能增强 `@Component` 类，以拦截方法和字段的调用。CGLIB 代理是通过调用 `@Configuration` 类中的 `@Bean` 方法中的方法或字段来创建对协作对象的 Bean 元数据引用的方法。此类方法不是使用普通的 Java 语义调用的，而是通过容器进行的，以提供一般的 Spring Bean 生命周期管理和代理，即使通过 `@Bean` 方法的编程调用引用其他 Bean 时也是如此。相反，在普通 `@Component` 类内的 `@Bean` 方法中调用方法或字段具有标准 Java 语义，而无需特殊的 CGLIB 处理或其他约束。

您可以将@Bean 方法声明为静态方法，从而允许在不将其包含配置类创建为实例的情况下调用它们。在定义 post-processor Bean（例如 BeanFactoryPostProcessor 或 BeanPostProcessor 类型）时，这特别有意义，因为此类 Bean 在容器生命周期的早期进行了初始化，并且应避免在那时触发配置的其他部分。

由于技术限制，对静态@Bean 方法的调用永远不会被容器拦截，甚至在@Configuration 类中也不会（如本节前面所述），因为技术限制：CGLIB 子类只能覆盖非静态方法。因此，直接调用另一个@Bean 方法具有标准的 Java 语义，从而导致从工厂方法本身直接返回一个独立的实例。



@Bean 方法的 Java 语言可见性不会对 Spring 容器中的最终 bean 定义产生直接影响。您可以在非@Configuration 类中自由声明自己的工厂方法，也可以在任何地方声明静态方法。但是，@Configuration 类中的常规@Bean 方法必须是可重写的—即，不得将它们声明为 private 或 final。

还可以在给定组件或配置类的基类上以及在由组件或配置类实现的接口中声明的 Java 8 默认方法上发现@Bean 方法。这为组合复杂的配置安排提供了很大的灵活性，从 Spring 4.2 开始，通过 Java 8 默认方法甚至可以实现多重继承。

最后，一个类可以为同一个 bean 保留多个@Bean 方法，这取决于在运行时可用的依赖关系，从而可以使用多个工厂方法。这与在其他配置方案中选择“最贪婪”的构造函数或工厂方法的算法相同：在构造时会选择具有最大可满足依赖关系的变量，类似于容器如何在多个@Autowired 构造函数之间进行选择。

1.10.6. 命名自动检测组件

在扫描过程中自动检测到组件时，其 bean 名称由该扫描程序已知的 BeanNameGenerator 策略生成。默认情况下，任何包含名称值的 Spring 构造型注释（@Component, @Repository, @Service 和 @Controller）都会将该名称提供给相应的 bean 定义。

如果这样的注解不包含名称值，或者不包含任何其他扫描到的组件（例如，由自定义过滤器发现的组件），则缺省 bean 名称生成器将返回不使用大写字母的非限定类名称。例如，如果检测到以下组件类，则名称将为 myMovieLister 和 movieFinderImpl：

Java

```
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

Kotlin

```
@Service("myMovieLister")
class SimpleMovieLister {
    // ...
}
```

Java

```
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

Kotlin

```
@Repository
class MovieFinderImpl : MovieFinder {
    // ...
}
```

如果不想依赖默认的 bean 命名策略，则可以提供自定义的命名策略。首先，实现 BeanNameGenerator 接口，并确保包括默认的无参构造函数。然后，在配置扫描器时提供完全限定的类名，如以下示例注解和 Bean 定义所示。



如果由于多个自动检测到的组件具有相同的非限定类名称（即，具有相同名称但位于不同程序包中的类）而导致命名冲突，则可能需要为了生成的 bean 名称配置一个 BeanNameGenerator，其默认值为全限定标准名称。从 Spring Framework 5.2.3 开始，在 org.springframework.context.annotation 包中的 FullyQualifiedAnnotationBeanNameGenerator 可以用于此类目的。

Java

```
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)
public class AppConfig {
    // ...
}
```

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"], nameGenerator = MyNameGenerator::class)  
class AppConfig {  
    // ...  
}
```

```
<beans>  
    <context:component-scan base-package="org.example"  
        name-generator="org.example.MyNameGenerator" />  
</beans>
```

通常，请考虑在其他组件可能对其进行显式引用时，使用注解指定名称。另一方面，只要容器负责装配，自动生成的名称就足够了。

1.10.7. 为自动扫描的组件提供一个作用域

通常，与 Spring 管理的组件一样，自动检测到的组件的默认范围也是最常见的范围是 **单例**。但是，有时您需要使用 **@Scope** 注解指定的其他作用域。您可以在注解中提供作用域的名称，如以下示例所示：

Java

```
@Scope("prototype")  
@Repository  
public class MovieFinderImpl implements MovieFinder {  
    // ...  
}
```

Kotlin

```
@Scope("prototype")  
@Repository  
class MovieFinderImpl : MovieFinder {  
    // ...  
}
```

 **@Scope** 批注仅在具体的 bean 类（对于带注解的组件）或工厂方法（对于 **@Bean** 方法）上进行内省。与 XML bean 定义相反，没有 bean 定义继承的概念，并且在类级别的继承层次结构与元数据目的无关。

有关特定于 Web 的作用域的详细信息，例如 Spring 上下文中的“request”或“session”，请参阅 [Request, Session, Application 和 WebSocket 作用域](#)。与这些作用域的预构建注解一样，您也可以使用 Spring 的元注解方法来编写自己的作用域注解：例如，使用 **@Scope (“prototype”)** 进行元注解的自定义注解，也可能会声明自定义的作用域。

域代理模式。

要提供用于作用域解析的自定义策略，而不是依赖于基于注解的方法，可以实现 `ScopeMetadataResolver` 接口。确保包括默认的无参数构造函数。然后，可以在配置扫描程序时提供完全限定的类名，如以下注释和 Bean 定义示例所示：

Java

```
@Configuration  
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)  
public class AppConfig {  
    // ...  
}
```

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"], scopeResolver = MyScopeResolver::class)  
class AppConfig {  
    // ...  
}
```

```
<beans>  
    <context:component-scan base-package="org.example" scope-  
    resolver="org.example.MyScopeResolver"/>  
</beans>
```

使用某些非单例作用域时，可能有必要为作用域对象生成代理。原因请在[作用域 Bean 作为依赖项](#)中查看。为此，在 `component-scan` 元素上可以使用 `scopedProxy` 属性。三个可能的值是：`no`, `interface` 和 `targetClass`。例如，以下配置产生标准的 JDK 动态代理：

Java

```
@Configuration  
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)  
public class AppConfig {  
    // ...  
}
```

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"], scopedProxy =  
    ScopedProxyMode.INTERFACES)  
class AppConfig {  
    // ...  
}
```

```
<beans>  
    <context:component-scan base-package="org.example" scoped-proxy="interfaces"/>  
</beans>
```

1.10.8. 使用组件提供限定符元数据

`@Qualifier` 注解在[使用 Qualifiers 基于注解的自动装配的调整中讨论过\(1.9.4\)](#)。该部分中的示例演示了`@Qualifier` 注解和自定义限定符注解的使用，以在解析自动装配候选时提供细粒度的控制。由于这些示例基于 XML bean 定义，因此通过使用 XML 中 bean 元素的限定符或 meta 子元素，在候选 bean 定义上提供了限定符元数据。当依靠类路径扫描来自动检测组件时，可以在候选类上为限定符元数据提供类型级别的注解。下面的三个示例演示了此技术：

Java

```
@Component  
@Qualifier("Action")  
public class ActionMovieCatalog implements MovieCatalog {  
    // ...  
}
```

Kotlin

```
@Component  
@Qualifier("Action")  
class ActionMovieCatalog : MovieCatalog
```

Java

```
@Component  
@Genre("Action")  
public class ActionMovieCatalog implements MovieCatalog {  
    // ...  
}
```

Kotlin

```
@Component  
@Genre("Action")  
class ActionMovieCatalog : MovieCatalog {  
    // ...  
}
```

Java

```
@Component  
@Offline  
public class CachingMovieCatalog implements MovieCatalog {  
    // ...  
}
```

Kotlin

```
@Component  
@Offline  
class CachingMovieCatalog : MovieCatalog {  
    // ...  
}
```



与大多数基于注释的替代方法一样，请记住，注解元数据绑定到类定义本身，而 XML 的使用允许相同类型的多个 bean 提供其限定符元数据的变体，因为每个元数据由每个实例提供而不是每个类。

1.10.9. 生成候选组件的索引

尽管类路径扫描非常快，但可以通过在编译时创建候选静态列表来提高大型应用程序的启动性能。在这种模式下，作为组件扫描目标的所有模块都必须使用此机制。



现存的@ComponentScan 或<context: component-scan 指令必须保留原样，以请求上下文扫描某些软件包中的候选对象。当 ApplicationContext 检测到这样的索引时，它将自动使用它，而不是扫描类路径。

要生成索引，请向每个包含组件的模块添加附加依赖关系，这些组件是组件扫描指令的目标。以下示例显示了如何使用 Maven 进行操作：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
        <version>5.2.8.RELEASE</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

对于 Gradle 4.5 和更早版本，依赖关系应在 `compileOnly` 配置中声明，如以下示例所示：

```
dependencies {
    compileOnly "org.springframework:spring-context-indexer:5.2.8.RELEASE"
}
```

在 Gradle 4.6 及更高版本中，依赖性应在注释处理器配置中声明，如以下示例所示：

```
dependencies {
    annotationProcessor "org.springframework:spring-context-indexer:{spring-version}"
}
```

该过程将生成一个包含在 jar 文件中的 `META-INF/spring.components` 文件。



在您的 IDE 中使用此模式时，必须将 `spring-context-indexer` 注册为注解处理器，以确保在更新候选组件时索引是最新的。



在以下位置找到 `META-INF / spring.components` 时会自动启用索引类路径。如果某些库（或用例）的索引部分可用，但无法为整个应用程序构建，你可以通过设置 `spring.index.ignore` 为 `true` 来 fallback 一个常规类路径安排（好像根本没有索引）作为系统属性或在 `classpath` 根目录下的 `spring.properties` 文件中。

1.11. 使用 JSR 330 标准注解

从 Spring 3.0 开始，Spring 提供对 JSR-330 标准注解（依赖注入）的支持。这些注解的扫描方式与 Spring 注解的扫描方式相同。要使用它们，您需要在类路径中有相关的 jar 包。

如果使用 Maven，则标准 Maven 存储库（<https://repo1.maven.org/maven2/javax/inject/javax.inject/1/>）中提供了 `javax.inject` 工件。您可以将以下依赖项添加到文件 `pom.xml` 中：



```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

1.11.1. 使用`@Inject` 和`@Named` 进行依赖注入

可以使用`@javax.inject.Inject` 代替`@Autowired`，如下所示：

Java

```
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(...);
        // ...
    }
}
```

Kotlin

```
import javax.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    fun listMovies() {
        movieFinder.findMovies(...)
        // ...
    }
}
```

与`@Autowired`一样，您可以在字段层面、方法层面和构造函数参数层面使用`@Inject`。此外，您可以将注入点声明为`Provider`，从而允许按需访问范围较小的bean，或者通过`Provider.get()`调用来懒访问其他bean。下示例提供了上述示例的变体：

Java

```
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(...);
        // ...
    }
}
```

Kotlin

```
import javax.inject.Inject

class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    fun listMovies() {
        movieFinder.findMovies(...)
        // ...
    }
}
```

如果要为应注入的依赖项使用限定名称，则应使用`@Named`批注，如以下示例所示：

Java

```
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;
    }

    // ...
}
```

Kotlin

```
import javax.inject.Inject
import javax.inject.Named

class SimpleMovieLister {

    private lateinit var movieFinder: MovieFinder

    @Inject
    fun setMovieFinder(@Named("main") movieFinder: MovieFinder) {
        this.movieFinder = movieFinder
    }

    // ...
}
```

与 `@Autowired` 一样，`@Inject` 也可以与 `java.util.Optional` 或 `@Nullable` 一起使用。这在这里更加适用，因为 `@Inject` 没有必需的属性。以下一对示例显示了如何使用 `@Inject` 和 `@Nullable`：

```
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        // ...
    }
}
```

Java

```
public class SimpleMovieLister {  
  
    @Inject  
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {  
        // ...  
    }  
}
```

Kotlin

```
class SimpleMovieLister {  
  
    @Inject  
    var movieFinder: MovieFinder? = null  
}
```

1.11.2. **@Named** 和**@ManagedBean**:和**@Component** 标准等效注解

可以使用 `@javax.inject.Named` 或 `javax.annotation.ManagedBean` 代替 `@Component`, 如以下示例所示:

Java

```
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;  
    }  
  
    // ...  
}
```

Kotlin

```
import javax.inject.Inject
import javax.inject.Named

@Named("movieListener") // @ManagedBean("movieListener") could be used as well
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

在不指定组件名称的情况下使用`@Component` 是非常常见的。可以类似的方式使用`@Named`, 如以下示例所示:

Java

```
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

Kotlin

```
import javax.inject.Inject
import javax.inject.Named

@Named
class SimpleMovieLister {

    @Inject
    lateinit var movieFinder: MovieFinder

    // ...
}
```

当使用 @Named 或 @ManagedBean 时，可以使用与使用 Spring 注解完全相同的方式来使用组件扫描，如以下示例所示：

Java

```
@Configuration  
@ComponentScan(basePackages = "org.example")  
public class AppConfig {  
    // ...  
}
```

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["org.example"])  
class AppConfig {  
    // ...  
}
```



与 @Component 相反，JSR-330 @Named 和 JSR-250 ManagedBean 注解是不可组合的。您应该使用 Spring 的构造型模型来构建自定义组件注解。

1.11.3. JSR-330 标准注解的局限

当使用标准注释时，您应该知道一些重要功能是不可用，如下表所示：

Table 6. Spring component model elements versus 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, only a way to identify named components.

Spring	javax.inject.*	javax.inject restrictions / comments
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's <code>prototype</code> . However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a <code>singleton</code> by default. In order to use a scope other than <code>singleton</code> , you should use Spring's <code>@Scope</code> annotation. <code>javax.inject</code> also provides a <code>@Scope</code> annotation. Nevertheless, this one is only intended to be used for creating your own annotations.
@Qualifier	@Qualifier / @Named	<code>javax.inject.Qualifier</code> is just a meta-annotation for building custom qualifiers. Concrete <code>String</code> qualifiers (like Spring's <code>@Qualifier</code> with a value) can be associated through <code>javax.inject.Named</code> .
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	<code>javax.inject.Provider</code> is a direct alternative to Spring's <code>ObjectFactory</code> , only with a shorter <code>get()</code> method name. It can also be used in combination with Spring's <code>@Autowired</code> or with non-annotated constructors and setter methods.

1.12. 基于 Java 的容器配置

本节介绍如何在 Java 代码中使用注释来配置 Spring 容器。它包括以下主题：

- 基本概念：@Bean 和@Configuration（1.12.1）
- 使用 AnnotationConfigApplicationContext 实例化 Spring 容器（1.12.2）
- 使用@Bean 注解（1.12.3）
- 使用@Configuration 注解（1.12.4）
- 组成基于 Java 的配置（1.12.5）
- Bean 定义配置文件（1.13.1）
- PropertySource 抽象（1.13.2）

- 使用@PropertySource (1.13.3)
- 声明中的占位符解析 (1.13.4)

1.12.1. 基本概念:@Bean 和@Configuration

Spring 的新 Java 配置支持中的主要工件是@Configuration 注解的类和@Bean 注解的方法。

@Bean 注解用于指示方法实例化，配置和初始化要由 Spring IoC 容器管理的新对象。对于那些熟悉 Spring 的<beans/> XML 配置的人来说，@ Bean 注解与<bean/>元素具有相同的作用。您可以将@Bean 注解的方法与任何 Spring @Component 一起使用。但是，它们最常与@Configuration bean 一起使用。

用@Configuration 注解表示该类的主要目的是作为 Bean 定义的来源。此外，
@Configuration 类允许通过调用同一类中的其他@Bean 方法来定义 Bean 之间的依赖关系。最简单的@Configuration 类的内容如下：

Java

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun myService(): MyService {
        return MyServiceImpl()
    }
}
```

前面的 `AppConfig` 类等同于下面的 Spring <beans/> XML：

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

完整的@Configuration 与“精简”@Bean 模式之争？

如果在未使用@Configuration 注解的类中声明@Bean 方法，则将它们称为以“精简”模式进行处理。在@Component 或普通类(plain old class)中声明的 Bean 方法被认为是“精简版”，其中包含类的主要目的不同，而@Bean 方法在那里具有某种优势。例如，服务组件可以通过每个适用组件类上的其他@Bean 方法将管理视图公开给容器。在这种情况下，@Bean 方法是一种通用的工厂方法机制。

与完整的@Configuration 不同，精简@Bean 方法无法声明 Bean 之间的依赖关系。取而代之的是，它们在其包含组件的内部状态上进行操作，并且还可以根据其可能声明的参数进行操作。因此，此类@Bean 方法不应调用其他@Bean 方法。实际上，每个这样的方法仅是用于特定 bean 的字面上的工厂方法，而没有任何特殊的运行时语义。这里的积极反作用是，不必在运行时应用 CGLIB 子类，因此在类设计方面没有任何限制(即，包含类可能是 final 类，依此类推)。

在常见情况下，@Bean 方法将在@Configuration 类中声明，以确保始终使用“完全”模式，因此跨方法引用将重定向到容器的生命周期管理。这样可以防止通过常规 Java 调用意外地调用同一@Bean 方法，这有助于减少在“精简”模式下运行时难找的细微错误。

以下各节将详细讨论@Bean 和@Configuration 注解。但是，首先，我们介绍了使用基于 Java 的配置来创建 spring 容器的各种方法。

1.12.2. 通过使用 AnnotationConfigApplicationContext 实例化 Spring 容器

以下各节介绍了 Spring 3.0 中引入的 Spring 的 AnnotationConfigApplicationContext。这种通用的 ApplicationContext 实现不仅可以接受@Configuration 类作为输入，还可以接受普通的@Component 类以及带有 JSR-330 元数据注解的类。

当提供@Configuration 类作为输入时，@Configuration 类本身将注册为 Bean 定义，并且该类中所有已声明的@Bean 方法也将注册为 Bean 定义。

提供@Component 和 JSR-330 类时，它们将注册为 Bean 定义，并且假定在必要时在这些类中使用了诸如@Autowired 或@Inject 之类的 DI 元数据。

简单结构

与实例化 `ClassPathXmlApplicationContext` 时将 Spring XML 文件用作输入的方式几乎相同，实例化 `AnnotationConfigApplicationContext` 时可以将`@Configuration`类用作输入。如下面的示例所示，这允许完全不使用 XML 来使用 Spring 容器：

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

如前所述，`AnnotationConfigApplicationContext` 不限于仅与`@Configuration`类一起使用。可以将任何`@Component` 或 JSR-330 带注解的类作为输入提供给构造函数，如以下示例所示：

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new
    AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class,
    Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(MyServiceImpl::class.java,
    Dependency1::class.java, Dependency2::class.java)
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

前面的示例假定 `MyServiceImpl`, `Dependency1` 和 `Dependency2` 使用 Spring 依赖项

注入注解，例如`@Autowired`。

通过使用 `register(Class<?>...)` 以编程方式构建容器

您可以使用无参构造函数实例化 `AnnotationConfigApplicationContext`，然后使用 `register()` 方法对其进行配置。以编程方式构建 `AnnotationConfigApplicationContext` 时，此方法特别有用。以下示例显示了怎么做：

Java

```
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();
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext()
    ctx.register(AppConfig::class.java, OtherConfig::class.java)
    ctx.register(AdditionalConfig::class.java)
    ctx.refresh()
    val myService = ctx.getBean<MyService>()
    myService.doStuff()
}
```

使用 `scan(String)` 启用扫描组件扫描

要启用组件扫描，可以按如下方式对`@Configuration`类进行注解：

Java

```
@Configuration
@ComponentScan(basePackages = "com.acme") ①
public class AppConfig {
    ...
}
```

① 此注解启用组件扫描。

Kotlin

```
@Configuration  
@ComponentScan(basePackages = ["com.acme"]) ①  
class AppConfig {  
    // ...  
}
```

- ① 此注解启用组件扫描。

经验丰富的 Spring 用户可能熟悉 Spring 的 context：名称空间中的 XML 声明，如以下示例所示：



```
<beans>  
    <context:component-scan base-package="com.acme"/>  
</beans>
```

在前面的示例中，对 `com.acme` 包进行了扫描以查找任何 `@Component` 注解的类，并将这些类注册为容器内的 Spring bean 定义。`AnnotationConfigApplicationContext` 公开了 `scan (String...)` 方法以允许相同的组件扫描功能，如以下示例所示：

Java

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```

Kotlin

```
fun main() {  
    val ctx = AnnotationConfigApplicationContext()  
    ctx.scan("com.acme")  
    ctx.refresh()  
    val myService = ctx.getBean<MyService>()  
}
```



请记住，`@Configuration` 类使用 `@Component` 进行元注解，因此它们是组件扫描的候选对象。在前面的示例中，假定 `AppConfig` 在 `com.acme` 包（或下面的任何包）中声明，则在调用 `scan()` 时将其抓取。根据 `refresh()`，其所有 `@Bean` 方法都将被处理并注册为容器内的 Bean 定义。

使用 AnnotationConfigWebApplicationContext 支持 Web 应用

AnnotationConfigWebApplicationContext 提供了 AnnotationConfigApplicationContext 的 WebApplicationContext 变体。在配置 Spring ContextLoaderListener Servlet 监听器，Spring MVC DispatcherServlet 等时，可以使用此实现。以下 web.xml 代码片段配置了典型的 Spring MVC Web 应用程序（请注意 contextClass context-param 和 init-param 的使用）：

```
<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 -->
    </servlet>

```

```

        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. 使用@Bean 注解

`@Bean` 是方法层面的注解，很像 XML `<bean/>` 标签。注解支持`<bean/>`提供的某些属性，例如`*init-method(1.6.1)*destroy-method(1.6.1)*autowiring(1.4.5) *name`。

您可以在`@Configuration` 注解或`@Component` 注解的类中使用`@Bean` 注解。

声明一个 Bean

要声明一个 bean，可以用`@Bean` 注解来标识一个方法。您可以使用此方法在类型指定为该方法的返回值的 `ApplicationContext` 中注册 Bean 定义。默认情况下，Bean 名称与方法名称相同。以下示例显示了`@Bean` 方法声明：

Java

```

@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}

```

Kotlin

```

@Configuration
class AppConfig {

    @Bean
    fun transferService() = TransferServiceImpl()
}

```

前面的配置与下面的 Spring XML 完全等效：

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

这两个声明使一个名为 `transferService` 的 bean 在 `ApplicationContext` 中可用，并绑定到类型 `TransferServiceImpl` 的对象实例，如以下文本图像所示：

```
transferService -> com.acme.TransferServiceImpl
```

您还可以使用接口（或基类）返回类型声明`@Bean` 方法，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl()
    }
}
```

但是，这将高级类型预测的可见性限制为指定的接口类型（`TransferService`）。然后，使用仅一次知道容器的完整类型（`TransferServiceImpl`），实例化受影响的单例 bean。非懒加载单例 bean 根据其声明顺序实例化，因此您可能会看到不同的类型匹配结果，具体取决于另一个组件何时尝试通过未声明的类型进行匹配（例如 `@Autowired` `TransferServiceImpl`，仅当 `transferService` bean 具有 被实例化）。



如果您通过声明的服务接口一致地引用类型，则@Bean 返回类型可以安全地加入该设计决策。但是，对于实现多个接口的组件或对于可能由其实现类型引用的组件，声明尽可能最具体的返回类型（至少与引用您的 bean 的注入点所要求的具体类型一样）是比较安全的。

Bean 依赖

@Bean 注解的方法可以具有任意数量的参数，这些参数描述构建该 bean 所需的依赖关系。例如，如果我们的 TransferService 需要一个 AccountRepository，则可以使用方法参数来实现该依赖关系，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}
```

解析机制与基于构造函数的依赖注入几乎相同。有关更多详细信息，请参见[相关部分 \(1.4.1\)](#)。

接收生命周期回调

任何使用@Bean 注解定义的类都支持常规的生命周期回调，并且可以使用 JSR-250 中的@PostConstruct 和@PreDestroy 注解。详情请见[JSR-250 注解 \(1.9.9\)](#)。

还完全支持常规的 Spring 生命周期 (1.6.1) 回调。如果 bean 实现了 InitializingBean, DisposableBean 或 Lifecycle，则容器将调用它们各自的方法。

还完全支持标准的*Aware 接口集（例如 BeanFactoryAware (1.16), BeanNameAware, MessageSourceAware (1.6.2), ApplicationContextAware (1.6.2) 等）。

@Bean 注解支持指定任意的初始化和销毁回调方法，就像 Spring XML 在 bean 元素上

的 init-method 和 destroy-method 属性一样，如以下示例所示：

Java

```
public class BeanOne {  
  
    public void init() {  
        // initialization logic  
    }  
}  
  
public class BeanTwo {  
  
    public void cleanup() {  
        // destruction logic  
    }  
}  
  
@Configuration  
public class AppConfig {  
  
    @Bean(initMethod = "init")  
    public BeanOne beanOne() {  
        return new BeanOne();  
    }  
  
    @Bean(destroyMethod = "cleanup")  
    public BeanTwo beanTwo() {  
        return new BeanTwo();  
    }  
}
```

Kotlin

```
class BeanOne {  
  
    fun init() {  
        // initialization logic  
    }  
}  
  
class BeanTwo {  
  
    fun cleanup() {  
        // destruction logic  
    }  
}  
  
@Configuration  
class AppConfig {  
  
    @Bean(initMethod = "init")  
    fun beanOne() = BeanOne()  
  
    @Bean(destroyMethod = "cleanup")  
    fun beanTwo() = BeanTwo()  
}
```

默认情况下，使用 Java 配置定义的具有公共 `close` 或 `shutdown` 方法的 bean 会自动通过销毁回调进行回收。如果您有一个公共 `close` 或 `shutdown` 方法，并且不希望在容器关闭时调用它，则可以将`@Bean(destroyMethod = "")` 添加到您的 bean 定义中以禁用默认（推断）模式。

默认情况下，您可能要对通过 JNDI 获取的资源执行此操作，因为其生命周期是在应用程序外部进行管理的。特别是，请确保始终对数据源执行此操作，因为在 Java EE 应用程序服务器上已知这是有问题的。下面的示例显示如何防止对数据源的自动销毁回调：

Java

```
@Bean(destroyMethod = "")  
public DataSource dataSource() throws NamingException {  
    return (DataSource) jndiTemplate.lookup("MyDS");  
}
```



Kotlin

```
@Bean(destroyMethod = "")  
fun dataSource(): DataSource {  
    return jndiTemplate.lookup("MyDS") as DataSource  
}
```

同样，通过`@Bean` 方法，通常使用程序化 JNDI 查找，方法是使用 Spring 的 `JndiTemplate` 或 `JndiLocatorDelegate` 帮助器，或者直接使用 JNDI `InitialContext` 用法，而不是 `JndiObjectFactoryBean` 变体（这将迫使您将返回类型声明为 `FactoryBean` 类型，而不是实际的目标类类型，因此很难在打算引用此处提供的资源的其他`@Bean` 方法中用于交叉引用调用）。

对于前面注释中的示例中的 BeanOne，在构造期间直接调用 `init()` 方法同样有效，如以下示例所示：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public BeanOne beanOne() {  
        BeanOne beanOne = new BeanOne();  
        beanOne.init();  
        return beanOne;  
    }  
  
    // ...  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun beanOne() = BeanOne().apply {  
        init()  
    }  
  
    // ...  
}
```



当您直接使用 Java 工作时，您可以对对象执行任何操作，而不必总是依赖于容器生命周期。

指定 Bean 作用域

Spring 包含 `@Scope` 注解，以便您可以指定 bean 的作用域。

- 使用 `@Scope` 注解

您可以指定使用 `@Bean` 注解定义的 bean 应该具有特定作用域。您可以使用 [Bean Scopes \(1.5\)](#) 部分中指定的任何标准作用域。

默认范围是单例，但是您可以使用 `@Scope` 注释覆盖它，如以下示例所示：

Java

```
@Configuration  
public class MyConfiguration {  
  
    @Bean  
    @Scope("prototype")  
    public Encryptor encryptor() {  
        // ...  
    }  
}
```

Kotlin

```
@Configuration  
class MyConfiguration {  
  
    @Bean  
    @Scope("prototype")  
    fun encryptor(): Encryptor {  
        // ...  
    }  
}
```

@Scope 和 scoped-proxy

Spring 提供了一种通过[作用域代理（58 页）](#)使用作用域依赖性的便捷方法。使用 XML 配置时创建此类代理的最简单方法是<aop:scoped-proxy/>元素。使用@Scope 注解在 Java 中配置 bean，可以通过 proxyMode 属性提供同等的支持。默认值为无代理（`ScopedProxyMode.NO`），但您可以设置为 `ScopedProxyMode.TARGET_CLASS` 或 `ScopedProxyMode.INTERFACES`。

如果使用 Java 从 XML 参考文档（请参阅[作用域代理（58 页）](#)）将作用域代理示例移植到我们的@Bean，则它类似于以下内容：

Java

```
// 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;
}
```

Kotlin

```
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
fun userPreferences() = UserPreferences()

@Bean
fun userService(): Service {
    return SimpleUserService().apply {
        // a reference to the proxied userPreferences bean
        setUserPreferences(userPreferences())
    }
}
```

自定义 Bean 名称

默认情况下，配置类使用@Bean 方法的名称作为结果 bean 的名称。但是，可以使用 name 属性覆盖此功能，如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean(name = "myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean("myThing")  
    fun thing() = Thing()  
}
```

Bean 别名

如在[命名 Bean \(上一节\)](#) 中讨论的那样，有时希望为单个 Bean 提供多个名称，否则称为 Bean 别名。为此，`@Bean` 注解的 `name` 属性接受一个 `String` 数组。以下示例说明如何为 bean 设置多个别名：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})  
    public DataSource dataSource() {  
        // instantiate, configure and return DataSource bean...  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean("dataSource", "subsystemA-dataSource", "subsystemB-dataSource")  
    fun dataSource(): DataSource {  
        // instantiate, configure and return DataSource bean...  
    }  
}
```

Bean 描述

有时，提供有关 bean 的更详细的文本描述会很有帮助。当出于监视目的而暴露（可能通过 JMX）bean 时，这尤其有用。

要向`@Bean` 添加描述，可以使用`@Description` 注释，如以下示例所示：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    @Description("Provides a basic example of a bean")  
    public Thing thing() {  
        return new Thing();  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    @Description("Provides a basic example of a bean")  
    fun thing() = Thing()  
}
```

1.12.4. 使用@Configuration 注解

`@Configuration` 是类级别的注解，指示对象是 Bean 定义的源。`@Configuration` 类通过公共`@Bean` 注解方法声明 bean。对`@Configuration` 类的`@Bean` 方法的调用也可以用于定义 Bean 之间的依赖关系。有关一般介绍，请参见 [基本概念：@Bean 和 @Configuration\(1.12.1\)](#)。

注入 bean 间的依赖关系

当 bean 相互依赖时，表示这种依赖关系就像让一个 bean 方法调用另一个一样简单，如以下示例所示：

Java

```
@Configuration  
public class AppConfig {  
  
    @Bean  
    public BeanOne beanOne() {  
        return new BeanOne(beanTwo());  
    }  
  
    @Bean  
    public BeanTwo beanTwo() {  
        return new BeanTwo();  
    }  
}
```

Kotlin

```
@Configuration  
class AppConfig {  
  
    @Bean  
    fun beanOne() = BeanOne(beanTwo())  
  
    @Bean  
    fun beanTwo() = BeanTwo()  
}
```

在前面的示例中，通过构造函数注入 `beanOne` 接收到 `beanTwo` 的引用。



仅当在 `@Configuration` 类中声明 `@Bean` 方法时，此声明 bean 间依赖关系的方法才有效。您不能使用简单的 `@Component` 类声明 Bean 间的依赖关系。

查找方法注入

如前所述，[查找方法注入\(1.4.6\)](#)是一项高级功能，您应该很少使用。在单例作用域的 bean 依赖于原型作用域的 bean 的情况下，这很有用。将 Java 用于这种类型的配置为实现此模式提供了原生的方法。以下示例显示如何使用查找方法注入：

Java

```
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();  
}
```

Kotlin

```
abstract class CommandManager {  
    fun process(commandState: Any): Any {  
        // grab a new instance of the appropriate Command interface  
        val 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 fun createCommand(): Command  
}
```

通过使用 Java 配置，可以创建 `CommandManager` 的子类，在该子类中，抽象的 `createCommand()` 方法将被覆盖，以便它查找新的（原型）命令对象。以下示例显示了如何执行此操作：

Java

```
@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 createCommand()  
    // overridden to return a new prototype Command object  
    return new CommandManager() {  
        protected Command createCommand() {  
            return asyncCommand();  
        }  
    }  
}
```

Kotlin

```
@Bean  
@Scope("prototype")  
fun asyncCommand(): AsyncCommand {  
    val command = AsyncCommand()  
    // inject dependencies here as required  
    return command  
}  
  
@Bean  
fun commandManager(): CommandManager {  
    // return new anonymous implementation of CommandManager with createCommand()  
    // overridden to return a new prototype Command object  
    return object : CommandManager() {  
        override fun createCommand(): Command {  
            return asyncCommand()  
        }  
    }  
}
```

关于基于 Java 配置在内部如何工作的更多信息

考虑以下示例，该示例显示了一个@Bean 注解方法被调用两次：

Java

```
@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();  
    }  
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun clientService1(): ClientService {
        return ClientServiceImpl().apply {
            clientDao = clientDao()
        }
    }

    @Bean
    fun clientService2(): ClientService {
        return ClientServiceImpl().apply {
            clientDao = clientDao()
        }
    }

    @Bean
    fun clientDao(): ClientDao {
        return ClientDaoImpl()
    }
}
```

`clientDao()` 在 `clientService1()` 中被调用一次，并在 `clientService2()` 中被调用一次。由于此方法会创建一个 `ClientDaoImpl` 的新实例并返回它，因此通常希望有两个实例（每个服务一个）。那肯定是有问题的：在 Spring 中，实例化的 bean 默认情况下具有单例作用域。这就是神奇的地方：所有 `@Configuration` 类在启动时都使用 `CGLIB` 进行了子类化。在子类中，子方法在调用父方法并创建新实例之前，首先检查容器中是否有任何缓存（作用域）的 bean。



根据 bean 的作用范围，行为可能会有所不同。 我们在这里讨论单例模式。



从 Spring 3.2 开始，不再需要将 `CGLIB` 添加到您的类路径中，因为 `CGLIB` 类已经在 `org.springframework.cglib` 下重新打包并直接包含在 `spring-core` JAR 中。

由于 CGLIB 在启动时会动态添加功能，因此存在一些限制。特别是，配置类不能是 final。但是，从 4.3 版本开始，配置类中允许使用任何构造函数，包括使用 @Autowired 或单个非默认构造函数声明进行默认注入。



如果您希望避免任何 CGLIB 施加的限制，请考虑在非@Configuration 类（例如，在普通的@Component 类上）声明@Bean 方法。然后，不会截获 @Bean 方法之间的跨方法调用，因此您必须专门依赖那里的构造函数或方法级别的依赖项注入。

1.12.5. 构建基于 Java 的配置

Spring 的基于 Java 的配置功能使您可以撰写注解，从而降低配置的复杂性。

使用@Import 注解

与 Spring XML 文件中使用<import/>元素来帮助模块化配置一样，@Import 注解允许从另一个配置类加载@Bean 定义，如以下示例所示：

Java

```
@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();  
    }  
}
```

Kotlin

```
@Configuration  
class ConfigA {  
  
    @Bean  
    fun a() = A()  
}  
  
@Configuration  
@Import(ConfigA::class)  
class ConfigB {  
  
    @Bean  
    fun b() = B()  
}
```

现在，无需在实例化上下文时同时指定 `ConfigA.class` 和 `ConfigB.class`，只需显式提供 `ConfigB`，如以下示例所示：

Java

```
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);  
}
```

Kotlin

```
import org.springframework.beans.factory.getBean  
  
fun main() {  
    val ctx = AnnotationConfigApplicationContext(ConfigB::class.java)  
  
    // now both beans A and B will be available...  
    val a = ctx.getBean<A>()  
    val b = ctx.getBean<B>()  
}
```

这种方法简化了容器的实例化，因为只需要处理一个类，而不是要求您在构造过程中记住潜在的大量`@Configuration` 类。



从 Spring Framework 4.2 开始，`@Import` 还支持对常规组件类的引用，类似于 `AnnotationConfigApplicationContext.register` 方法。如果要通过使用一些配置类作为入口点来显式定义所有组件，从而避免组件扫描，则此功能特别有用。

在导入的`@Bean` 定义上注入依赖项

前面的示例有效，但过于简单。在大多数实际情况下，Bean 在配置类之间相互依赖。使用 XML 时，这不是问题，因为不涉及任何编译器，并且您可以声明 `ref="someBean"` 并信任 Spring 在容器初始化期间进行处理。使用 `@Configuration` 类时，Java 编译器会在配置模型上施加约束，因为对其他 bean 的引用必须是有效的 Java 语法。

幸运的是解决问题非常简单。正如[我们已经讨论的](#)，`@Bean` 方法可以具有任意数量的参数来描述 Bean 的依赖关系。考虑以下具有多个 `@Configuration` 类的更真实的场景，每个类取决于在其他类中声明的 bean：

Java

```
@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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Bean
    fun transferService(accountRepository: AccountRepository): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig {

    @Bean
    fun accountRepository(dataSource: DataSource): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

还有另一种方法可以达到相同的结果。请记住，`@Configuration` 类最终仅是容器中的另一个 bean：这意味着它们可以利用`@Autowired` 和`@Value` 注入以及与任何其他 bean 相同的其他功能。

确保以这种方式注入的依赖项只是最简单的一种。`@Configuration` 类在上下文的初始化过程中很早就被处理，并且强制以这种方式注入依赖项可能导致意外的过早初始化。如上例所示，请尽可能使用基于参数的注入。



另外，通过 `@Bean` 使用 `BeanPostProcessor` 和 `BeanFactoryPostProcessor` 定义时要特别小心。通常应将这些声明为静态 `@Bean` 方法，而不触发其包含的配置类的实例化。否则，`@Autowired` 和 `@Value` 可能不适用于配置类本身，因为可以将其创建为比 `AutowiredAnnotationBeanPostProcessor` 早的 bean 实例。

下面的例子展示了一个 bean 如何被装配为另一个 bean：

Java

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    lateinit var accountRepository: AccountRepository

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(accountRepository)
    }
}

@Configuration
class RepositoryConfig(private val dataSource: DataSource) {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }
}

@Configuration
@Import(ServiceConfig::class, RepositoryConfig::class)
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return new DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    // everything wires up across configuration classes...
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```



`@Configuration` 类的构造器注入方法只在 Spring 4.3 之后被支持，

同样需要注意的是如果目标 bean 定义只有一个构造器的时候不需要特指
`@Autowired`。

对导航场景的全限定 bean 导入

在前面的场景中，使用`@Autowired` 可以很好地工作并提供所需的模块化，但是确定一定以及肯定在何处声明自动装配的 Bean 定义仍然有些模棱两可。例如，当开发人员查看 `ServiceConfig` 时，您如何确切知道`@Autowired AccountRepository` bean 的声明位置？它在代码中并不是显式的，这可能看上去很美好。请记住，[Spring Tools for Eclipse](#)

提供了可以渲染图形的工具，这些图形显示了所有事物的装配方式，这也许就是您所需要的。另外，您的 Java IDE 可以轻松找到 `AccountRepository` 类型的所有声明和使用，并快速向您显示返回该类型的`@Bean` 方法的位置。

如果这种歧义是不可接受的，并且您希望从 IDE 内部直接从一个`@Configuration` 类导航到另一个`@Configuration` 类，请考虑自动装配配置类本身。以下示例显示了如何执行此操作：

Java

```
@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());
    }
}
```

Kotlin

```
@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        // navigate 'through' the config class to the @Bean method!
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}
```

在上述情况下，定义 `AccountRepository` 的位置是完全明确的。但是，`ServiceConfig` 现在与 `RepositoryConfig` 紧密耦合，那就很坑了。所以通过使用基于接口或基于抽象类的`@Configuration` 类可以稍微缓解这种紧密耦合。请看以下示例：

Java

```
@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");
}
```

Kotlin

```
import org.springframework.beans.factory.getBean

@Configuration
class ServiceConfig {

    @Autowired
    private lateinit var repositoryConfig: RepositoryConfig

    @Bean
    fun transferService(): TransferService {
        return TransferServiceImpl(repositoryConfig.accountRepository())
    }
}

@Configuration
interface RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository
}

@Configuration
class DefaultRepositoryConfig : RepositoryConfig {

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(...)
    }
}

@Configuration
@Import(ServiceConfig::class, DefaultRepositoryConfig::class) // import the concrete config!
class SystemTestConfig {

    @Bean
    fun dataSource(): DataSource {
        // return DataSource
    }
}

fun main() {
    val ctx = AnnotationConfigApplicationContext(SystemTestConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    transferService.transfer(100.00, "A123", "C456")
}
```

现在，`ServiceConfig` 与具体的 `DefaultRepositoryConfig` 是松耦合，并且内置的 IDE 工具仍然有用：您可以轻松地获得 `RepositoryConfig` 实现的类型层次结构。以这种方式，指向`@Configuration` 类及其依赖项与指向基于接口的代码的通常过程没有什么不同。



如果要影响某些 bean 的启动创建顺序，请考虑将其中一些声明为 `@Lazy`（用于首次访问而不是在启动时创建）或声明为`@DependsOn` 某些其他 bean（确保其他特定 bean 在当前 bean 创建之前，意味着它们是直接依赖的）。

有条件地包括`@Configuration` 类或 Bean 方法

根据某些系统状态，有条件地启用或禁用完整的`@Configuration` 类甚至单个`@Bean` 方法通常很有用。一个常见的示例是仅在 Spring 环境中启用了特定的配置文件时才使用 `@Profile` 注解来激活 Bean（有关详细信息，请参见 [Bean 定义配置文\(1.13.1\)](#)）。

`@Profile` 注解实际上通过使用更灵活的调用`@Conditional` 注解来实现的。

`@Conditional` 注解在 `@Bean` 注册之前应引用的特定 `org.springframework.context.annotation.Condition` 实现。

`Condition` 接口的实现提供了一个 `matches(...)` 方法，该方法返回 `true` 或 `false`。例如，以下列表显示了用于`@Profile` 的实际 `Condition` 实现：

Java

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    // 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;
}
```

Kotlin

```
override fun matches(context: ConditionContext, metadata: AnnotatedTypeMetadata): Boolean {
    // Read the @Profile annotation attributes
    val attrs = metadata.getAllAnnotationAttributes(Profile::class.java.name)
    if (attrs != null) {
        for (value in attrs["value"]!!) {
            if (context.environment.acceptsProfiles(Profiles.of(*value as
Array<String>)))) {
                return true
            }
        }
        return false
    }
    return true
}
```

有关更多详细信息，请参见[@Conditional](#) 的 javadoc。

结合 Java 和 XML 配置

Spring 的[@Configuration](#) 类支持并非旨在完全替代 Spring XML。某些工具（例如 Spring XML 名称空间）仍然是配置容器的理想方法。在使用 XML 方便或有必要的情况下，你可以选择：使用例如以“XML 为中心”的方式凭借“[ClassPathXmlApplicationContext](#)”实例化容器，或以“Java 为中心”的方式使用 [AnnotationConfigApplicationContext](#) 实例化容器。[@ImportResource](#) 注解可根据需要导入 XML。

以 XML 中心使用[@Configuration](#) 类

或许更偏好从 XML 引导 Spring 容器，并包括用特定方式创建的[@Configuration](#) 类。例如，在使用 Spring XML 的大型现有代码库中，根据需要创建[@Configuration](#) 类并从现有 XML 文件中将它们包含在内会变得更加容易。在稍后章节，我们将介绍在这种“以 XML 为中心”的情况下使用[@Configuration](#) 类的情况。

将[@Configuration](#) 类声明为纯 Spring `<bean/>` 元素

请记住，[@Configuration](#) 类最终是容器中的 bean 定义。在本系列示例中，我们创建一个名为 [AppConfig](#) 的 [@Configuration](#) 类，并将其使用 `<bean/>` 定义包含在 `systemtest-config.xml` 中。因为`<context:annotation-config/>` 已打开，所以容器可以识别[@Configuration](#) 注解并正确处理 [AppConfig](#) 中声明的[@Bean](#) 方法。

以下示例显示了 Java 中的普通配置类：

Java

```
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Autowired
    private lateinit var dataSource: DataSource

    @Bean
    fun accountRepository(): AccountRepository {
        return JdbcAccountRepository(dataSource)
    }

    @Bean
    fun transferService() = TransferService(accountRepository())
}
```

以下示例显示了 `system-test-config.xml` 文件的一部分：

```

<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>

```

以下示例显示了可能的 `jdbc.properties` 文件：

```

jdbc.url=jdbc:hsqldb:hsq://localhost/xdb
jdbc.username=sa
jdbc.password=

```

Java

```

public static void main(String[] args) {
    ApplicationContext ctx = new
    ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}

```

Kotlin

```

fun main() {
    val ctx = ClassPathXmlApplicationContext("classpath:/com/acme/system-test-
config.xml")
    val transferService = ctx.getBean<TransferService>()
    // ...
}

```

在 `system-test-config.xml` 文件中，`AppConfig <bean >/>` 没有声明 `id` 元素。尽管这样做是可以接受的，但是由于没有其他 `bean` 曾经引用过它，因此这是不必要的，并且不太可能通过名称从容器中显式获取。同样，`DataSource` `bean` 只能按类型自动装配，因此也不严格要求显式 `bean id`。

使用 `<context: component-scan/>` 抓取 `@Configuration` 类

由于 `@Configuration` 使用 `@Component` 进行元注解，因此 `@Configuration` 注解的类自动成为组件扫描的候选对象。使用与先前示例中描述的场景相同的场景，我们可以重新定义 `system-test-config.xml` 以利用组件扫描的优势。请注意，在这种情况下，我们无需

显式声明`<context:annotation-config/>`，因为`<context:component-scan/>`可启用相同的功能。

以下示例显示了修改后的`system-test-config.xml`文件：

```
<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>
```

@Configuration 以类为中心的 XML 与@ImportResource 的使用

在`@Configuration`类是配置容器的主要机制的应用程序中，仍然有必要多少用点 XML。在这些情况下，您可以使用`@ImportResource`并仅定义所需的 XML。这样做实现了“以 Java 为中心”的方法来配置容器，并使 XML 保持在最低限度。以下示例（包括配置类，定义 Bean 的 XML 文件，属性文件和 main 类）显示了如何使用`@ImportResource`注解来实现按需使用 XML 的以 Java 为中心的配置：

Java

```
@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);
    }
}
```

Kotlin

```
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
class AppConfig {

    @Value("\${jdbc.url}")
    private lateinit var url: String

    @Value("\${jdbc.username}")
    private lateinit var username: String

    @Value("\${jdbc.password}")
    private lateinit var password: String

    @Bean
    fun dataSource(): DataSource {
        return DriverManagerDataSource(url, username, password)
    }
}
```

```
properties-config.xml
<beans>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsq://localhost/xdb
jdbc.username=sa
jdbc.password=
```

Java

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

```

import org.springframework.beans.factory.getBean

fun main() {
    val ctx = AnnotationConfigApplicationContext(AppConfig::class.java)
    val transferService = ctx.getBean<TransferService>()
    // ...
}

```

1.13. 环境抽象概念

`Environment` 接口是集成在容器中的抽象概念，可对应用程序环境的两个关键方面进行建模：[概要文件\(1.13.1\)](#)和[属性\(1.13.2\)](#)。

概要文件是仅在给定概要文件处于活动状态时才向容器注册的 Bean 定义的命名逻辑组。可以将 Bean 分配给概要文件，无论是以 XML 定义还是带有注解。与配置文件相关的环境对象的作用是确定哪些配置文件（如果有）当前处于活动状态，以及哪些配置文件（如果有）在默认情况下应处于活动状态。

属性在几乎所有应用程序中都起着重要作用，并且可能源自各种来源：属性文件，JVM 系统属性，系统环境变量，JNDI，Servlet 上下文参数，特殊属性对象，`Map` 对象等。环境对象与属性相关的作用是为用户提供方便的服务界面，用于配置属性源并从中解析属性。

1.13.1. Bean 定义文件

Bean 定义配置文件在核心容器中提供了一种机制，该机制允许在不同环境中注册不同的 Bean。“environment”一词对不同的用户可能具有不同的含义，并且此功能可以帮助解决许多用例，包括：

- 在开发中针对内存中的数据源进行工作，而不是在进行 QA 或生产时从 JNDI 查找相同的数据源。
- 仅在将应用程序部署到性能环境中时注册监视基础架构。
- 为消费者 A 针对消费者 B 部署注册 bean 的自定义实现。

在需要数据源的实际应用中考虑第一个用例。在测试环境中，配置可能类似于以下内容：

Java

```
@Bean  
public DataSource dataSource() {  
    return new EmbeddedDatabaseBuilder()  
        .setType(EmbeddedDatabaseType.HSQL)  
        .addScript("my-schema.sql")  
        .addScript("my-test-data.sql")  
        .build();  
}
```

Kotlin

```
@Bean  
fun dataSource(): DataSource {  
    return EmbeddedDatabaseBuilder()  
        .setType(EmbeddedDatabaseType.HSQL)  
        .addScript("my-schema.sql")  
        .addScript("my-test-data.sql")  
        .build()  
}
```

现在，假设该应用程序的数据源已在生产应用程序服务器的 JNDI 目录中注册，请考虑如何将该应用程序部署到 QA 或生产环境中。现在，我们的 `dataSource` bean 看起来像下面的列表：

Java

```
@Bean(destroyMethod = "")  
public DataSource dataSource() throws Exception {  
    Context ctx = new InitialContext();  
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");  
}
```

Kotlin

```
@Bean(destroyMethod = "")  
fun dataSource(): DataSource {  
    val ctx = InitialContext()  
    return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource  
}
```

问题是如何根据当前环境在使用这两种变体之间进行切换。随着时间的流逝，Spring 用户已经设计出多种方法来完成此任务，通常依赖于系统环境变量和包含`$ {placeholder}` 占位符的 XML `<import />`语句的组合，这个 XML 根据环境变量值解析为正确的配置文件路径。Bean 定义配置文件是一项核心容器功能，可提供此问题的解决方案。

如果我们概括前面特定于环境的 Bean 定义示例中所示的用例，那么最终需要在某些上下文中而不是在其他上下文中注册某些 Bean 定义。你可能会说你要在情况 A 中注册一个特

定的 bean 定义配置文件，在情况 B 中注册一个不同的配置文件。我们首先更新配置以反映这种需求。

使用 @profile 注解

`@Profile` 注解可让你在一个或多个指定的配置文件处于活动状态时指定有资格注册的组件。使用前面的示例，我们可以如下重写 dataSource 配置：

Java

```
@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();  
    }  
}
```

Kotlin

```
@Configuration  
@Profile("development")  
class StandaloneDataConfig {  
  
    @Bean  
    fun dataSource(): DataSource {  
        return EmbeddedDatabaseBuilder()  
            .setType(EmbeddedDatabaseType.HSQL)  
            .addScript("classpath:com/bank/config/sql/schema.sql")  
            .addScript("classpath:com/bank/config/sql/test-data.sql")  
            .build()  
    }  
}
```

Java

```
@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");  
     }  
 }
```

Kotlin

```
@Configuration  
 @Profile("production")  
 class JndiDataConfig {  
  
     @Bean(destroyMethod = "")  
     fun dataSource(): DataSource {  
         val ctx = InitialContext()  
         return ctx.lookup("java:comp/env/jdbc/datasource") as DataSource  
     }  
 }
```

如前所述，对于@Bean 方法，通常选择使用程序化 JNDI 查找，方法是使用 Spring 的 JndiTemplate / JndiLocatorDelegate 帮助器或前面显示的直接 JNDI InitialContext 用法，而不是 JndiObjectFactoryBean 变体，这将强制你将返回类型声明为 FactoryBean 类型。

概要文件字符串可以包含简单的概要文件名（例如，生产）或概要文件表达式。配置文件表达式允许表达更复杂的配置文件逻辑（例如，production&us-east）。 概要文件表达式中支持以下运算符：

- !：配置文件的逻辑“非”
- &：配置文件的逻辑“与”
- |：配置文件的逻辑“或”

您不能不使用括号的运算符的情况下将 & 和 | 混合使用 。 例如，

 production&us-east| eu-central 不是有效的表达式。 它必须表示为 production&(us-east|eu-central)。

您可以将@Profile 用作元注解，以创建自定义的组合注解。以下示例定义了一个自定义@Production 注解，您可以将其用作@Profile("production")的替代品：

Java

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

Kotlin

```
@Target(AnnotationTarget.TYPE)
@Retention(AnnotationRetention.RUNTIME)
@Profile("production")
annotation class Production
```



如果 `@Configuration` 类用 `@Profile` 标记，则除非该类中的一个或多个指定的配置文件处于活动状态，否则所有与该类关联的 `@Bean` 方法和 `@Import` 注解都会被忽略。如果 `@Component` 或 `@Configuration` 类标记有 `@Profile({"p1", "p2"})`，则除非已激活概要文件 'p1' 或 'p2'，否则不会注册或处理该类。如果给定的配置文件以 NOT 运算符 (!) 为前缀，则仅在该配置文件未激活时才注册带该注解的元素。例如，给定 `@Profile({"p1", "! p2"})`，如果配置文件 'p1' 处于活动状态或如果配置文件 'p2' 未处于活动状态，则会进行注册。

也可以在方法级别将 `@Profile` 声明为仅包含配置类的一个特定 bean（例如，特定 bean 的替代变体），如以下示例所示：

Java

```
@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    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") ②
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

- ① `standaloneDataSource` 方法仅在 `development` 配置文件中可用。
② `jndiDataSource` 方法仅在 `production` 配置文件中可用。

Kotlin

```
@Configuration
class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    fun standaloneDataSource(): DataSource {
        return 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") ②
    fun jndiDataSource() =
        InitialContext().lookup("java:comp/env/jdbc/datasource") as DataSource
}
```

- ① `standaloneDataSource` 方法仅在 `development` 配置文件中可用。
② `jndiDataSource` 方法仅在 `production` 配置文件中可用。

在 @Bean 方法上使用 @Profile 时，可能会出现特殊情况：如果 Java 方法名称相同的重载@Bean 方法（类似于构造函数重载），则必须在所有重载方法上一致声明 @Profile 条件。如果条件不一致，则仅重载方法中第一个声明的条件很重要。因此，@Profile 不能用于选择具有另一个参数签名的重载方法。在创建时，同一 bean 的所有工厂方法之间的解析都遵循 Spring 的构造函数解析算法。



如果要使用不同的概要文件条件定义备用 bean，请使用 @Bean name 属性使用指向相同 bean 名称的不同 Java 方法名称，如前面的示例所示。如果参数签名都相同（例如，所有变体都具有无参工厂方法），则这是首先在有效 Java 类中表示这种排列的唯一方法（因为只能有一个特定名称和参数签名的方法）。

XML Bean 定义配置文件

XML 对应项是 <beans> 元素的 profile 属性。我们前面的示例配置可以用两个 XML 文件重写，如下所示：

```
<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>
```

也可以避免在同一文件中拆分和嵌套 <beans/> 元素，如以下示例所示：

```

<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 已被限制为仅允许作为文件中的最后一个元素，这应有助于提供灵活性而不会在 XML 文件中造成混乱。

XML 对应项不支持前面描述的配置文件表达式。但是，可以使用！操作符取消配置文件。也可以通过嵌套配置文件来应用逻辑“与”，如以下示例所示：



```

<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="production">
        <beans profile="us-east">
            <jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/datasource"/>
        </beans>
    </beans>
</beans>

```

在前面的示例中，如果 **production** 和 **us-east** 配置文件都处于活动状态，则将 **dataSource** bean 对外暴露。

激活一个 Profile

现在，我们已经更新了配置，我们仍然需要指示 Spring 哪个配置文件处于活动状态。如果我们现在启动示例应用程序，将会看到抛出 `NoSuchBeanDefinitionException` 的异常消息，因为容器找不到名为 `dataSource` 的 Spring bean。

可以通过多种方式来激活配置文件，但是最直接的方法是针对可通过 `ApplicationContext` 获得的 `Environment` API 以编程方式进行配置，下面的示例演示了如何进行配置：

Java

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment(). setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

Kotlin

```
val ctx = AnnotationConfigApplicationContext().apply {
    environment.setActiveProfiles("development")
    register(SomeConfig::class.java, StandaloneDataConfig::class.java,
JndiDataConfig::class.java)
    refresh()
}
```

此外，您还可以通过 `spring.profiles.active` 属性以声明方式激活配置文件，该属性可以通过系统环境变量，JVM 系统属性，`web.xml` 中的 `servlet` 上下文参数来指定，甚至可以作为 JNDI 中的条目来指定(请参阅 [PropertySource Abstraction\(1.13.2\)](#))。在集成测试中，可以使用 `spring-test` 模块中的`@ActiveProfiles` 注解来声明活动配置文件(请参阅[环境配置文件的上下文配置](#))。

请注意，配置文件不是“非此即彼”的问题。你可以一次激活多个配置文件。通过编程，可以为 `setActiveProfiles()` 方法提供多个配置文件名称，该方法接受字符串参数。以下示例激活多个配置文件：

Java

```
ctx.getEnvironment(). setActiveProfiles("profile1", "profile2");
```

Kotlin

```
ctx.getEnvironment(). setActiveProfiles("profile1", "profile2")
```

声明性地，`spring.profiles.active` 可以接受逗号分隔的配置文件名称列表，如以

下示例所示：

```
-Dspring.profiles.active="profile1,profile2"
```

默认 Profile

默认配置文件表示默认情况下启用的配置文件。请看以下示例：

Java

```
@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();  
    }  
}
```

Kotlin

```
@Configuration  
@Profile("default")  
class DefaultDataConfig {  
  
    @Bean  
    fun dataSource(): DataSource {  
        return EmbeddedDatabaseBuilder()  
            .setType(EmbeddedDatabaseType.HSQL)  
            .addScript("classpath:com/bank/config/sql/schema.sql")  
            .build()  
    }  
}
```

如果没有配置文件处于活动状态，那么将创建 dataSource。你可以看到这是为一个或多个 bean 提供默认定义的一种方法。如果启用了任何配置文件，则默认配置文件将不适用。

您可以通过在 `Environment` 上使用 `setDefaultProfiles()` 或声明性地使用 `spring.profiles.default` 属性来更改默认配置文件的名称。

1.13.2. PropertySource 抽象

Spring 的环境抽象概念提供了可配置属性源层次结构上的搜索操作。请看以下列表：

Java

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsMyProperty = env.containsProperty("my-property");
System.out.println("Does my environment contain the 'my-property' property? " +
containsMyProperty);
```

Kotlin

```
val ctx = GenericApplicationContext()
val env = ctx.environment
val containsMyProperty = env.containsProperty("my-property")
println("Does my environment contain the 'my-property' property? $containsMyProperty")
```

在前面的代码片段中，我们看到了一种高级方式来询问 Spring 是否为当前环境定义了 `my-property` 属性。为了回答这个问题，环境对象在一组 `PropertySource` 对象上执行搜索。`PropertySource` 是对任何键值对源的简单抽象，并且 Spring 的 `StandardEnvironment` 配置有两个 `PropertySource` 对象—一个代表 JVM 系统属性的集合 (`System.getProperties()`) 和一个代表系统环境变量的集合 (`System.getenv()`)。



这些默认属性源适用于 `StandardEnvironment`，可在独立应用程序中使用。`StandardServletEnvironment` 填充了其他默认属性源，包括 `servlet` 配置和 `servlet` 上下文参数。它可以选择启用 `JndiPropertySource`。有关详细信息，请参见 javadoc。

具体来说，当您使用 `StandardEnvironment` 时，如果在运行时存在 `my-property` 系统属性或 `my-property` 环境变量，则对 `env.containsProperty("my-property")` 的调用将返回 `true`。

执行的搜索是分层的。默认情况下，系统属性优先于环境变量。因此，如果在调用 `env.getProperty ("my-property")` 时在两个地方都同时设置了 `my-property` 属性，则系统属性值“胜出”并返回。请注意，属性值不会合并，而是会被前面的条目完全覆盖。



对于常见的 `StandardServletEnvironment`，完整层次结构如下，下列表优先级自上而下：

1. `ServletConfig` 参数（如果适用，例如在 `DispatcherServlet` 上下文中）
2. `ServletContext` 参数（`web.xml` 上下文参数条目）
3. JNDI 环境变量（`java:comp/env/`条目）
4. JVM 系统属性（`-D` 命令行参数）
5. JVM 系统环境（操作系统环境变量）

最重要的是，整个机制是可配置的。也许您具有要集成到此搜索中的自定义属性源。为此，实现并实例化您自己的 `PropertySource` 并将其添加到当前环境的 `PropertySources` 集中。以下示例显示了如何执行此操作：

Java

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

Kotlin

```
val ctx = GenericApplicationContext()
val sources = ctx.environment.propertySources
sources.addFirst(MyPropertySource())
```

在前面的代码中，已在搜索中以最高优先级添加了 `MyPropertySource`。如果它包含 `my-property` 属性，则将检测并返回该属性，以支持任何其他 `PropertySource` 中的 `my-property` 属性。`MutablePropertySources` API 公开了许多方法，这些方法允许对属性源集进行精确操作。

1.13.3. 使用 `@PropertySource`

`@PropertySource` 注解为将 `PropertySource` 添加到 Spring 的 `Environment` 中提供了一种方便的声明性机制。

给定一个名为 `app.properties` 的文件，其中包含键值对 `testbean.name =`

`myTestBean`, 下面的`@Configuration`类使用`@PropertySource`, 以这样的方式调用`testBean.getName()`会返回`myTestBean`:

Java

```
@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;  
     }  
 }
```

Kotlin

```
@Configuration  
 @PropertySource("classpath:/com/myco/app.properties")  
 class AppConfig {  
  
     @Autowired  
     private lateinit var env: Environment  
  
     @Bean  
     fun testBean() = TestBean().apply {  
         name = env.getProperty("testbean.name")!!  
     }  
 }
```

`@PropertySource` 资源位置中存在的任何`...{}`占位符都是根据已经针对该环境注册的一组属性源来解析的, 如以下示例所示:

Java

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
public class AppConfig {

    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

Kotlin

```
@Configuration
@PropertySource("classpath:/com/\${my.placeholder:default/path}/app.properties")
class AppConfig {

    @Autowired
    private lateinit var env: Environment

    @Bean
    fun testBean() = TestBean().apply {
        name = env.getProperty("testbean.name")!!
    }
}
```

假设 `my.placeholder` 存在于已注册的属性源之一（例如，系统属性或环境变量）中，则将占位符解析为相应的值。如果不是，则将 `default/path` 用作默认值。如果未指定默认值并且无法解析属性，则抛出 `IllegalArgumentException`。



根据 Java 8 约定，`@PropertySource` 注解是可重复的。但是，所有此类`@PropertySource` 注解都需要在同一层面上声明，可以直接在配置类上声明，也可以在同一自定义注解中声明为元注解。

1.13.4. 语句中占位符解析

回首过去，元素中占位符的值只能根据 JVM 系统属性或环境变量来解析。今非昔比已不再是这种情况。由于环境抽象是在整个容器中集成的，因此很容易通过它来路由占位符的解析。这意味着您可以按照自己喜欢的任何方式配置解析过程。您可以更改搜索系统属性和环境变量的优先级，也可以完全删除它们。您还可以根据需要将自己的属性源添加到组合中。

具体而言，以下语句无论在何处定义 `customer` 属性，只要在 `Environment` 中可用，该语句就起作用：

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

1.14. 注册一个 LoadTimeWeaver

Spring 使用 `LoadTimeWeaver` 在将类加载到 Java 虚拟机（JVM）中时对其进行动态转换。

要启用加载时织入，可以将`@EnableLoadTimeWeaving` 添加到您的`@Configuration` 类之一，如以下示例所示：

Java

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig { }
```

Kotlin

```
@Configuration
@EnableLoadTimeWeaving
class AppConfig
```

另外，对于 XML 配置，可以使用 `context:load-time-weaver` 元素：

```
<beans>
    <context:load-time-weaver/>
</beans>
```

为 `ApplicationContext` 配置后，该 `ApplicationContext` 中的任何 bean 都可以实现 `LoadTimeWeaverAware`，从而接收对加载时 weaver 实例的引用。与 Spring 的 JPA 支持结合使用时，该功能特别有用，因为在 JPA 类转换中可能需要进行加载时编织。咨询 `LocalContainerEntityManagerFactoryBean`. 有关 AspectJ 加载时编织的更多信息，请参见使用 Spring 框架中的 AspectJ(5.10.4)。

1.15. ApplicationContext 的其他功能

如本章介绍(第一章开头)中所讨论的, `org.springframework.beans.factory` 包提供了用于管理和操作 bean 的基本功能, 包括以编程方式。`org.springframework.context` 包添加了 `ApplicationContext` 接口, 该接口扩展了 `BeanFactory` 接口, 此外还扩展了其他接口以提供更多面向应用程序框架的样式的附加功能。许多人以完全声明性的方式使用 `ApplicationContext`, 甚至没有以编程方式创建它, 而是依靠诸如 `ContextLoader` 之类的支持类来自动实例化 `ApplicationContext`, 作为 Java EE Web 应用程序正常启动过程的一部分。

为了以更加面向框架的方式增强 `BeanFactory` 的功能, 上下文包还提供以下功能:

- 通过 `MessageSource` 接口访问 i18n 样式的消息。
- 通过 `ResourceLoader` 接口访问资源, 例如 URL 和文件。
- 通过使用 `ApplicationEventPublisher` 接口, 将事件发布到实现 `ApplicationListener` 接口的 bean。
- 加载多个(分层)上下文, 使每个上下文都通过 `HierarchicalBeanFactory` 接口集中在一个特定层上, 例如应用程序的 Web 层。

1.15.1. Internationalization using MessageSource

`ApplicationContext` 接口扩展了一个称为 `MessageSource` 的接口, 因此提供了国际化 (“i18n”) 功能。Spring 还提供了 `HierarchicalMessageSource` 接口, 该接口可以分层解析消息。这些接口一起提供了 Spring 效果消息解析的基础。这些接口上定义的方法包括:

- `String getMessage(String code, Object[] args, String default, Locale loc)`: 从 `MessageSource` 检索消息的基本方法。如果找不到针对指定语言环境的消息, 则使用默认消息。使用标准库提供的 `MessageFormat` 功能, 传入的所有参数都将成为替换值。
- `String getMessage(String code, Object[] args, Locale loc)`: 与先前的方法基本相同, 但有一个区别: 不能指定默认消息。如果找不到该消息, 则抛出 `NoSuchMessageException`。
- `String getMessage(MessageSourceResolvable resolvable, Locale`

`locale`): 前述方法中使用的所有属性也都包装在一个名为 `MessageSourceResolvable` 的类中，您可以将其与该方法一起使用。

加载 `ApplicationContext` 时，它将自动搜索在上下文中定义的 `MessageSource` bean。Bean 必须具有名称 `messageSource`。如果找到了这样的 bean，则对先前方法的所有调用都将委派给消息源。两者都实现 `HierarchicalMessageSource` 以便进行嵌套消息传递。`StaticMessageSource` 很少使用，但是提供了将消息添加到源中的编程方式。下面的示例显示 `ResourceBundleMessageSource`:

```
<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>
```

该示例假定您在类路径中定义了三个资源包，分别称为 `format`, `exception` 和 `windows`。解析消息的任何请求都通过 `ResourceBundle` 对象用 JDK 标准的解析消息来处理。就本示例而言，假定上述两个资源束文件的内容如下：

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

下一个示例显示了运行 `MessageSource` 功能的程序。请记住，所有 `ApplicationContext` 实现也是 `MessageSource` 实现，因此可以转换为 `MessageSource` 接口。

Java

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", Locale.ENGLISH);
    System.out.println(message);
}
```

Kotlin

```
fun main() {
    val resources = ClassPathXmlApplicationContext("beans.xml")
    val message = resources.getMessage("message", null, "Default", Locale.ENGLISH)
    println(message)
}
```

以上程序的结果输出如下：

```
Alligators rock!
```

总而言之，`MessageSource` 是在名为 `beans.xml` 的文件中定义的，该文件位于类路径的根目录下。`messageSource` bean 定义通过其 `basenames` 属性引用了许多资源包。列表中传递给 `basenames` 属性的三个文件在类路径的根目录下以文件形式存在，分别称为 `format.properties`, `exceptions.properties` 和 `windows.properties`。

下一个示例显示了传递给消息查找的参数。这些参数将转换为 `String` 对象，并插入到查找消息中的占位符中。

```
<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.something.Example">
        <property name="messages" ref="messageSource"/>
    </bean>

</beans>
```

Java

```
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", Locale.ENGLISH);  
        System.out.println(message);  
    }  
}
```

Kotlin

```
class Example {  
  
    lateinit var messages: MessageSource  
  
    fun execute() {  
        val message = messages.getMessage("argument.required",  
            arrayOf("userDao"), "Required", Locale.ENGLISH)  
        println(message)  
    }  
}
```

调用 `execute()` 方法的结果输出如下：

```
The userDao argument is required.
```

关于国际化（“i18n”），Spring 的各种 `MessageSource` 实现遵循与标准 JDK `ResourceBundle` 相同的语言环境解析和后备规则。简而言之，并继续前面定义的示例 `messageSource`，如果您想根据英国（en-GB）语言环境解析消息，则可以分别创建名为 `format_en_GB.properties`，`exceptions_en_GB.properties` 和 `windows_en_GB.properties` 的文件。

通常，语言环境解析由应用程序的周围环境管理。在以下示例中，手动指定了针对其解析（英国）消息的语言环境：

```
# in exceptions_en_GB.properties  
argument.required=Ebagum lad, the ''{0}'' argument is required, I say, required.
```

Java

```
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);  
}
```

Kotlin

```
fun main() {  
    val resources = ClassPathXmlApplicationContext("beans.xml")  
    val message = resources.getMessage("argument.required",  
        arrayOf("userDao"), "Required", Locale.UK)  
    println(message)  
}
```

上述程序运行的结果如下：

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

您还可以使用 `MessageSourceAware` 接口获取对已定义的任何 `MessageSource` 的引用。创建和配置 bean 时，在 `ApplicationContext` 中定义的实现 `MessageSourceAware` 接口的任何 bean 都会与应用程序上下文的 `MessageSource` 一起注入。



作为 `ResourceBundleMessageSource` 的替代，Spring 提供了 `ReloadableResourceBundleMessageSource` 类。此变体支持相同的包文件格式，但比基于标准 JDK 的 `ResourceBundleMessageSource` 实现要灵活得多。特别是，它允许从任何 Spring 资源位置（不仅从类路径）读取文件，并支持热重载捆绑属性文件（同时在它们之间进行有效缓存）。有关详细信息，请参见 `ReloadableResourceBundleMessageSource` Javadoc。

1.15.2. 标准和自定义事件

通过 `ApplicationEvent` 类 和 `ApplicationListener` 接口 提供 `ApplicationContext` 中的事件处理。如果将实现 `ApplicationListener` 接口的 bean 部署到上下文中，则每次将 `ApplicationEvent` 发布到 `ApplicationContext` 时，都会通知该 bean。本质上，这是标准的观察者模式。



从 Spring 4.2 开始，事件基础架构显著改进，并提供了基于注解的模型(207 页)以及发布任意事件（即不一定从 `ApplicationEvent` 扩展的

对象) 的功能。发布此类对象后, 我们会为您包装一个事件。

下表描述了 Spring 提供的标准事件:

事件	说明
ContextRefreshedEvent	在初始化或刷新 ApplicationContext 时发布 (例如, 通过使用 ConfigurableApplicationContext 接口上的 refresh() 方法)。在这里, “initialized”是指所有 Bean 都已加载, 检测到并激活了后处理器 Bean, 已预先实例化单例并且可以使用 ApplicationContext 对象。只要尚未关闭上下文, 只要选定的 ApplicationContext 实际上支持这种“热”刷新, 就可以多次触发刷新。例如, XmlWebApplicationContext 支持热刷新, 但 GenericApplicationContext 不支持。
ContextStrartedEvent	在 ConfigurableApplicationContext 接口上使用 start() 方法启动 ApplicationContext 时发布。在这里, “Strated”是指所有 Lifecycle bean 都收到一个明确的启动信号。通常, 此信号用于在显式停止后重新启动 Bean, 但也可以用于启动尚未配置为自动启动的组件(例如, 尚未在初始化时启动的组件)。
ContextStoppedEvent	通过 使用 ConfigurableApplicationContext 接口上的 stop() 方法停止 ApplicationContext 时发布。在这里, “Stopped”表示所有 Lifecycle bean 都收到一个明确的停止信号。 停止的上下文可以通过 start() 调用重新启动。
ContextClosedEvent	通过 使用 ConfigurableApplicationContext 接口上的 close()方法关闭 ApplicationContext 或通过 JVM shutdown 钩子时发布。在这里, “Closed”意味着所有单例 bean 将被销毁。 关闭上下文后, 它将完成生命周期, 无法刷新或重新启动。
RequestHandledEvent	一个特定于 Web 的事件, 告诉所有 bean HTTP 请求已经被执行。服务。请求完成后, 将发布此事件。此事件仅适用于使用 Spring 的 DispatcherServlet 的 Web 应用程序。
ServletRequestHandledEvent	RequestHandledEvent 的子类, 添加了特定于 Servlet 的上下文信息。

您还可以创建和发布自己的自定义事件。以下示例显示了一个简单的类，该类扩展了 Spring 的 ApplicationEvent 基类：

Java

```
public class BlockedListEvent extends ApplicationEvent {  
  
    private final String address;  
    private final String content;  
  
    public BlockedListEvent(Object source, String address, String content) {  
        super(source);  
        this.address = address;  
        this.content = content;  
    }  
  
    // accessor and other methods...  
}
```

Kotlin

```
class BlockedListEvent(source: Any,  
                      val address: String,  
                      val content: String) : ApplicationEvent(source)
```

若要发布自定义 ApplicationEvent，请在 ApplicationEventPublisher 上调用 publishEvent() 方法。通常，这是通过创建一个实现 ApplicationEventPublisherAware 的类并将其注册为 Spring Bean 来完成的。以下示例显示了此类：

Java

```
public class EmailService implements ApplicationEventPublisherAware {  
  
    private List<String> blockedList;  
    private ApplicationEventPublisher publisher;  
  
    public void setBlockedList(List<String> blockedList) {  
        this.blockedList = blockedList;  
    }  
  
    public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {  
        this.publisher = publisher;  
    }  
  
    public void sendEmail(String address, String content) {  
        if (blockedList.contains(address)) {  
            publisher.publishEvent(new BlockedListEvent(this, address, content));  
            return;  
        }  
        // send email...  
    }  
}
```

Kotlin

```
class EmailService : ApplicationEventPublisherAware {  
  
    private lateinit var blockedList: List<String>  
    private lateinit var publisher: ApplicationEventPublisher  
  
    fun setBlockedList(blockedList: List<String>) {  
        this.blockedList = blockedList  
    }  
  
    override fun setApplicationEventPublisher(publisher: ApplicationEventPublisher) {  
        this.publisher = publisher  
    }  
  
    fun sendEmail(address: String, content: String) {  
        if (blockedList!!contains(address)) {  
            publisher!!.publishEvent(BlockedListEvent(this, address, content))  
            return  
        }  
        // send email...  
    }  
}
```

在配置时，Spring 容器检测到 `EmailService` 实现了 `ApplicationEventPublisherAware` 并自动调用 `setApplicationEventPublisher()`。实际上，传入的参数是 Spring 容器本身。您正在通过其 `ApplicationEventPublisher` 接口与应用程序上下文进行交互。

要接收自定义 `ApplicationEvent`，可以创建一个实现 `ApplicationListener` 的类，并将其注册为 Spring Bean。以下示例显示了此类：

Java

```
public class BlockedListNotifier implements ApplicationListener<BlockedListEvent> {  
  
    private String notificationAddress;  
  
    public void setNotificationAddress(String notificationAddress) {  
        this.notificationAddress = notificationAddress;  
    }  
  
    public void onApplicationEvent(BlockedListEvent event) {  
        // notify appropriate parties via notificationAddress...  
    }  
}
```

Kotlin

```
class BlockedListNotifier : ApplicationListener<BlockedListEvent> {  
  
    lateinit var notificationAddress: String  
  
    override fun onApplicationEvent(event: BlockedListEvent) {  
        // notify appropriate parties via notificationAddress...  
    }  
}
```

注意，`ApplicationListener` 通常使用您的自定义事件的类型（在前面的示例中为 `BlockedListEvent`）进行参数化。这意味着 `onApplicationEvent()` 方法可以保持类型安全，从而避免了向下转换的需要。您可以根据需要注册任意数量的事件监听器，但是请注意，默认情况下，事件监听器会同步接收事件。这意味着 `publishEvent()` 方法将阻塞，直到所有监听器都已完成对事件的处理为止。这种同步和单线程方法的一个优点是，当监听器收到事件时，如果有可用的事务上下文，它将在发布者的事务上下文内部进行操作。如果有必要采用其他发布事件的策略，请参见 Spring 的 `ApplicationEventMulticaster` 接口的 `javadoc` 和 `SimpleApplicationEventMulticaster` 实现的配置选项。

以下示例显示了用于注册和配置上述每个类的 Bean 定义：

```

<bean id="emailService" class="example.EmailService">
    <property name="blockedList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
</bean>

<bean id="blockedListNotifier" class="example.BlockedListNotifier">
    <property name="notificationAddress" value="blockedlist@example.org"/>
</bean>

```

将所有内容放在一起，当调用 `emailService` bean 的 `sendEmail()` 方法时，如果有任何应阻止的电子邮件，则会发布 `BlockedListEvent` 类型的自定义事件。`blockedListNotifier` bean 被注册为 `ApplicationListener` 并接收 `BlockedListEvent`，这时它可以通知适当的参与者。



Spring 的事件机制旨在在同一应用程序上下文内在 Spring bean 之间进行简单的通信。但是，对于更复杂的企业集成需求，单独维护的 [Spring Integration](#) 项目为基于著名的 Spring 编程模型构建轻量级，面向模式，事件驱动的架构提供了完整的支持。

基于注解的事件监听器

从 Spring 4.2 开始，您可以使用 `@EventListener` 注解在托管 Bean 的任何公共方法上注册事件监听器。`BlockedListNotifier` 可以重写如下：

Java

```

public class BlockedListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlockedListEvent(BlockedListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}

```

Kotlin

```
class BlockedListNotifier {  
  
    lateinit var notificationAddress: String  
  
    @EventListener  
    fun processBlockedListEvent(event: BlockedListEvent) {  
        // notify appropriate parties via notificationAddress...  
    }  
}
```

方法签名再次声明其侦听的事件类型，但是这次使用灵活的名称并且没有实现特定的监听器接口。只要实际事件类型在其实现层次结构中解析您的通用参数，也可以通过通用类型来缩小事件类型。

如果您的方法应该侦听多个事件，或者您想完全不使用任何参数来定义它，则事件类型也可以在注解本身上指定。以下示例显示了如何执行此操作：

Java

```
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})  
public void handleContextStart() {  
    // ...  
}
```

Kotlin

```
@EventListener(ContextStartedEvent::class, ContextRefreshedEvent::class)  
fun handleContextStart() {  
    // ...  
}
```

还可以通过使用定义 [SpEL 表达式\(第 4 章\)](#) 的注解的 `condition` 属性来添加其他运行时过滤，该注释应匹配以针对特定事件实际调用该方法。以下示例显示了仅当事件的 `content` 属性等于 `my-event` 时，才可以重写我们的通知程序以进行调用：

Java

```
@EventListener(condition = "#blEvent.content == 'my-event'")  
public void processBlockedListEvent(BlockedListEvent blockedListEvent) {  
    // notify appropriate parties via notificationAddress...  
}
```

Kotlin

```
@EventListener(condition = "#blEvent.content == 'my-event'")  
fun processBlockedListEvent(blockedListEvent: BlockedListEvent) {  
    // notify appropriate parties via notificationAddress...  
}
```

每个 SpEL 表达式都会根据专用上下文进行评估。 下表列出了可用于上下文的项目，以便您可以将它们用于条件事件处理：

名称	位置	描述	示例
Event	root object	实际的 ApplicationEvent。	#root.event 或者 event
Arguments array	root object	用于调用方法的参数（作为对象数组）。	# root.args 或 args; args [0] 访问第一个参数，依此类推。
Argument name	evaluation context	任何方法参数的名称。 如果由于某种原因这些名称不可用（例如，由于在编译的字节码中没有调试信息），则还可以使用#a <#arg>语法（其中<#arg>代表参数索引（从 0 开始）。	#blEvent 或 # a0 (您也可以使用 # p0 或 #p <#arg> 参数表示法作为别名)

请注意，即使您的方法签名实际上引用了已发布的任意对象，#root.event 也使您可以访问基础事件。如果由于处理另一个事件而需要发布一个事件，则可以更改方法签名以返回应发布的事件，如以下示例所示：

Java

```
@EventListener  
public ListUpdateEvent handleBlockedListEvent(BlockedListEvent event) {  
    // notify appropriate parties via notificationAddress and  
    // then publish a ListUpdateEvent...  
}
```

Kotlin

```
@EventListener  
fun handleBlockedListEvent(event: BlockedListEvent): ListUpdateEvent {  
    // notify appropriate parties via notificationAddress and  
    // then publish a ListUpdateEvent...  
}
```



异步监听器不支持此功能。

此新方法为上述方法处理的每个 `BlockedListEvent` 发布一个新的 `ListUpdateEvent`。如果需要发布多个事件，则可以返回事件的集合。

异步监听器

如果希望特定的侦听器异步处理事件，则可以重用常规的 `@Async` 支持。以下示例显示了如何执行此操作：

Java

```
@EventListener  
@Async  
public void processBlockedListEvent(BlockedListEvent event) {  
    // BlockedListEvent is processed in a separate thread  
}
```

Kotlin

```
@EventListener  
@Async  
fun processBlockedListEvent(event: BlockedListEvent) {  
    // BlockedListEvent is processed in a separate thread  
}
```

使用异步事件时，请注意以下限制：

- 如果异步事件侦听器引发 `Exception`，则不会将其传播到调用方。有关更多详细信息，请参见 `AsyncUncaughtExceptionHandler`。
- 异步事件侦听器方法无法通过返回值来发布后续事件。如果您需要发布另一个事件作为处理的结果，请注入 `ApplicationEventPublisher` 以手动发布事件。

指定监听器顺序

如果需要先调用一个监听器，则可以将 `@Order` 注解添加到方法声明中，如以下示例所示：

Java

```
@EventListener  
 @Order(42)  
 public void processBlockedListEvent(BlockedListEvent event) {  
     // notify appropriate parties via notificationAddress...  
 }
```

Kotlin

```
@EventListener  
 @Order(42)  
 fun processBlockedListEvent(event: BlockedListEvent) {  
     // notify appropriate parties via notificationAddress...  
 }
```

一般事件

您还可以使用泛型来进一步定义事件的结构。考虑使用 `EntityCreatedEvent<T>`，其中 `T` 是已创建的实际实体的类型。例如，您可以创建以下监听器定义以仅接收 `Person` 的 `EntityCreatedEvent`：

Java

```
@EventListener  
 public void onPersonCreated(EntityCreatedEvent<Person> event) {  
     // ...  
 }
```

Kotlin

```
@EventListener  
 fun onPersonCreated(event: EntityCreatedEvent<Person>) {  
     // ...  
 }
```

由于类型擦除，只有在触发的事件解析了事件监听器所依据的通用参数（即类似 `PersonCreatedEvent` 的类扩展 `EntityCreatedEvent <Person> {...}`）时，此方法才起作用。

在某些情况下，如果所有事件都遵循相同的结构，这可能会变得很乏味（就像前面示例中的事件一样）。在这种情况下，您可以实现 `ResolvableTypeProvider` 来指定框架超出运行时环境提供的范围。以下事件显示了如何执行此操作：

Java

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(),
ResolvableType.forInstancegetSource());
    }
}
```

Kotlin

```
class EntityCreatedEvent<T>(entity: T) : ApplicationEvent(entity),
ResolvableTypeProvider {

    override fun getResolvableType(): ResolvableType? {
        return ResolvableType.forClassWithGenerics(javaClass,
ResolvableType.forInstancegetSource())
    }
}
```



这个玩意不只是对 `ApplicationEvent` 而是你只要当做事件发送的任何对象他都管事。

1.15.3. 获取低级资源的简便方法

为了获得最佳用法和对应用程序上下文的理解，您应该熟悉 Spring 的 `Resource` 抽象，如[资源（第 2 章）](#)所述。

应用程序上下文是 `ResourceLoader`，可用于加载 `Resource` 对象。`Resource` 本质上是 JDK `java.net.URL` 类的功能更丰富的版本。实际上，`Resource` 的实现现在适当的地方包装了 `java.net.URL` 的实例。资源可以以透明的方式从几乎任何位置获取低级资源，包括从类路径，文件系统位置，可使用标准 URL 描述的任何位置以及其他一些变体。如果资源位置字符串是没有任何特殊前缀的简单路径，则这些资源的来源是特定的，并且适合于实际的应用程序上下文类型。

您可以配置部署到应用程序上下文中的 Bean，以实现特殊的回调接口 `ResourceLoaderAware`，以便在初始化时自动调用，并将应用程序上下文本身作为 `ResourceLoader` 传入。您还可以公开 `Resource` 类型的属性，以用于访问静态资源。它们像其他任何属性一样注入其中。您可以将那些 `Resource` 属性指定为简单的 `String` 路径，

并在部署 bean 时依靠从这些文本字符串到实际 Resource 对象的自动转换。

提供给 `ApplicationContext` 构造函数的一个或多个位置路径实际上是资源字符串，并且根据特定的上下文实现以简单的形式对其进行适当处理。例如，`ClassPathXmlApplicationContext` 将简单的位置路径视为类路径位置。您也可以使用带有特殊前缀的位置路径（资源字符串）来强制从类路径或 URL 中加载定义，而不管实际的上下文类型如何。

1.15.4. 对于 Web 应用的简便应用上下文的实例化

您可以使用例如 `ContextLoader` 声明性地创建 `ApplicationContext` 实例。当然，您也可以使用 `ApplicationContext` 实现之一以编程方式创建 `ApplicationContext` 实例。

您可以使用 `ContextLoaderListener` 注册 `ApplicationContext`，如以下示例所示：

```
<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>
```

监听器检查 `contextConfigLocation` 参数。如果参数不存在，那么监听器将使用 `/WEB-INF/applicationContext.xml` 作为默认值。当参数确实存在时，侦听器将使用预定义的定界符（逗号，分号和空格）来分隔 `String`，并将这些值用作搜索应用程序上下文的位置。还支持 Ant-style 的路径模式。示例包括 `/WEB-INF/*Context.xml`（适用于所有名称以 `Context.xml` 结尾且位于 `WEB-INF` 目录中的文件）和 `/WEB-INF/**/*Context.xml`（适用于所有此类文件）文件在 `WEB-INF` 的任何子目录中）。

1.15.5. 将一个 Spring 应用上下文部署成一个 Java EE RAR 文件

可以将 Spring `ApplicationContext` 部署为 RAR 文件，将上下文及其所有必需的 bean 类和库 JAR 封装在 Java EE RAR 部署单元中。这等效于引导独立的 `ApplicationContext`（仅托管在 Java EE 环境中）能够访问 Java EE 服务器功能。RAR 部署是部署无头 WAR 文件的方案的一种更自然的选择-实际上，这种 WAR 文件没有任何 HTTP 入口点，仅用于在 Java EE 环境中引导 `Spring ApplicationContext`。

对于不需要 HTTP 入口点而仅由消息端点和计划的作业组成的应用程序上下文，RAR 部

署是理想的选择。在这样的上下文中，Bean 可以使用应用程序服务器资源，例如 JTA 事务管理器和 JNDI 绑定的 JDBC `DataSource` 实例以及 JMS `ConnectionFactory` 实例，并且还可以在平台的 JMX 服务器上注册 - 整个过程都通过 Spring 的标准事务管理以及 JNDI 和 JMX 支持工具进行。应用程序组件还可以通过 Spring 的 `TaskExecutor` 抽象与应用程序服务器的 JCA `WorkManager` 进行交互。

有关 RAR 部署中涉及的配置详细信息，请参见 `SpringContextResourceAdapter` 类的 `javadoc`。

对于将 Spring `ApplicationContext` 作为 Java EE RAR 文件的简单部署：

- ① 将所有应用程序类打包到 RAR 文件（这是具有不同文件扩展名的标准 JAR 文件）中。将所有必需的库 JAR 添加到 RAR 归档文件的根目录中。添加一个 `META-INF / ra.xml` 部署描述符（如 `SpringContextResourceAdapter` 的 `javadoc` 中所示）和相应的 Spring XML `bean` 定义文件（通常为 `META-INF / applicationContext.xml`）。

- ② 将生成的 RAR 文件拖放到应用程序服务器的部署目录中。

此类 RAR 部署单元通常是独立的。它们不会将组件暴露给外界，甚至不会暴露给同一应用程序的其他模块。与基于 RAR 的 `ApplicationContext` 的交互通常是通过与其他模块共享的 JMS 目标进行的。例如，基于 RAR 的 `ApplicationContext` 还可以安排一些作业或对文件系统（或类似文件）中的新文件做出反应。如果需要允许从外部进行同步访问，则可以（例如）导出 RMI 端点，该端点可以由同一台计算机上的其他应用程序模块使用。

1.16. Bean 工厂

`BeanFactory` API 为 Spring 的 IoC 功能提供了基础。它的特定作用主要用于与 Spring 的其他部分以及相关的第三方框架集成，并且其 `DefaultListableBeanFactory` 实现是更高级别的 `GenericApplicationContext` 容器中的关键委托。

`BeanFactory` 和相关接口（例如 `BeanFactoryAware`, `InitializingBean`, `DisposableBean`）是其他框架组件的重要集成点。不需要任何注解甚至反射，它们甚至可以在容器及其组件之间进行非常有效的交互。应用级 Bean 可以使用相同的回调接口，但通常更喜欢通过注释或通过程序配置进行声明式依赖注入。

请注意，核心 `BeanFactory` API 级别及其 `DefaultListableBeanFactory` 实现不对

配置格式或要使用的任何组件注解进行假设。所有这些 flavors 都是通过扩展（例如 `XmlBeanDefinitionReader` 和 `AutowiredAnnotationBeanPostProcessor`）引入的，并以核心元数据表示形式对共享 `BeanDefinition` 对象进行操作。这就是使 Spring 的容器如此灵活和可扩展的本质。

1.16.1. BeanFactory 还是 ApplicationContext?

本节说明 `BeanFactory` 和 `ApplicationContext` 容器级别之间的区别以及对引导的影响。

除非有充分的理由，否则应使用 `ApplicationContext`，将 `GenericApplicationContext` 及其子类 `AnnotationConfigApplicationContext` 作为自定义引导的常见实现。这些是所有常见目的 Spring 核心容器的主要入口点：加载配置文件，触发类路径扫描，以编程方式注册 Bean 定义和带注释的类，以及（自 5.0 版本起）注册功能性 Bean 定义。

因为 `ApplicationContext` 包含 `BeanFactory` 的所有功能，所以通常建议在普通 `BeanFactory` 上使用，除非需要对 Bean 处理的完全控制。在 `ApplicationContext`（例如 `GenericApplicationContext` 实现）中，按照约定（即，按 Bean 名称或 Bean 类型（尤其是后处理器））检测到几种 Bean，而普通的 `DefaultListableBeanFactory` 不知道任何特殊的 Bean。

对于许多扩展的容器功能，例如注释处理和 AOP 代理，`BeanPostProcessor` 扩展点至关重要。如果仅使用普通的 `DefaultListableBeanFactory`，则默认情况下不会检测和激活此类后处理器。这种情况可能会造成混淆，因为您的 bean 配置实际上并没有错。而是在这种情况下，需要通过其他设置完全引导容器。下表列出了 `BeanFactory` 和 `ApplicationContext` 接口和实现所提供的功能。

特性	BeanFactory	ApplicationContext
Bean 初始化/装配	Yes	Yes
集成生命周期管理	No	Yes
自动化 <code>BeanPostProcessor</code> 注册	No	Yes
自动化 <code>BeanFactoryPostProcessor</code> 注册	No	Yes
简便 <code>MessageSource</code> 访问（为了初始化）	No	Yes
内置 <code>ApplicationEvent</code> 发布机制	No	Yes

要向 `DefaultListableBeanFactory` 显式注册 Bean 后处理器，需要以编程方式调用 `addBeanPostProcessor`，如以下示例所示：

Java

```
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
```

Kotlin

```
val factory = DefaultListableBeanFactory()
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(AutowiredAnnotationBeanPostProcessor())
factory.addBeanPostProcessor(MyBeanPostProcessor())

// now start using the factory
```

要将 `BeanFactoryPostProcessor` 应用于普通的 `DefaultListableBeanFactory`，您需要调用其 `postProcessBeanFactory` 方法，如以下示例所示：

Java

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertySourcesPlaceholderConfigurer cfg = new PropertySourcesPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

Kotlin

```
val factory = DefaultListableBeanFactory()
val reader = XmlBeanDefinitionReader(factory)
reader.loadBeanDefinitions(FileSystemResource("beans.xml"))

// bring in some property values from a Properties file
val cfg = PropertySourcesPlaceholderConfigurer()
cfg.setLocation(FileSystemResource("jdbc.properties"))

// now actually do the replacement
cfg.postProcessBeanFactory(factory)
```

在这两种情况下，显式的注册步骤都不方便，这就是为什么在 Spring 支持的应用程序中，各种 ApplicationContext 变量比普通的 DefaultListableBeanFactory 更为可取的原因，尤其是在典型企业设置中依赖 BeanFactoryPostProcessor 和 BeanPostProcessor 实例来扩展容器功能时。



AnnotationConfigApplicationContext 已注册了所有常见的注解的后处理器，并且可以通过诸如@EnableTransactionManagement之类的配置注解在后台引入其他处理器。在 Spring 基于注解的配置模型的抽象级别上，bean 后处理器的概念仅是内部容器详细信息。

2. 资源

本章介绍了 Spring 如何处理资源以及如何在 Spring 中使用资源。 它包括以下主题：

- 介绍
- 资源接口
- 内置资源实现
- `ResourceLoader`
- `ResourceLoaderAware` 接口
- 资源依赖
- 应用程序上下文和资源路径

2.1. 介绍

不幸的是，Java 的标准 `java.net.URL` 类和用于各种 URL 前缀的标准处理程序不足以满足所有对低级资源的访问。例如，没有标准化的 `URL` 实现可用于访问需要从类路径或相对于 `ServletContext` 获得的资源。尽管可以为专用 URL 前缀注册新的处理程序（类似于用于诸如 `http:` 的前缀的现有处理程序），但这通常相当复杂，并且 `URL` 接口仍然缺少某些理想的功能，例如用于检查是否存在方法指向的资源。

2.2. 资源接口

Spring `Resource` 接口旨在成为一种功能更强大的接口，用于抽象化对低级资源的访问。以下清单显示了 `Resource` 接口定义：

Java

```
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();  
}
```

Kotlin

```
interface Resource : InputStreamSource {  
    fun exists(): Boolean  
    val isOpen: Boolean  
    val url: URL  
    val file: File  
    @Throws(IOException::class)  
    fun createRelative(relativePath: String): Resource  
    val filename: String  
    val description: String  
}
```

如 `Resource` 接口的定义所示，它扩展了 `InputStreamSource` 接口。以下清单显示了 `InputStreamSource` 接口的定义：

Java

```
public interface InputStreamSource {  
    InputStream getInputStream() throws IOException;  
}
```

Kotlin

```
interface InputStreamSource {  
    val inputStream: InputStream  
}
```

Resource 接口中一些最重要的方法是：

- `getInputStream()`: 找到并打开资源，返回一个 `InputStream` 以便从资源中读取。 预期每次调用都会返回一个新的 `InputStream`。 调用者负责关闭流。
- `exists()`: 返回一个布尔值，指示此资源是否实际以物理形式存在。
- `isOpen()`: 返回一个布尔值，指示此资源是否表示具有打开流的句柄。 如果为 `true`，则不能多次读取 `InputStream`，必须只读取一次，然后将其关闭以避免资源泄漏。 对于所有常规资源实现，返回 `false`，但 `InputStreamResource` 除外。
- `getDescription()`: 返回对此资源的描述，用于在处理资源时用于错误输出。 这通常是标准文件名或资源的实际 URL。

其他方法可让您获取代表资源的实际 URL 或 `File` 对象（如果基础实现兼容并且支持该功能）。

当需要资源时，Spring 本身广泛使用 Resource 抽象作为许多方法签名中的参数类型。一些 Spring API 中的其他方法（例如，各种 ApplicationContext 实现的构造函数）采用 String 形式，该字符串以未经修饰或简单的形式用于创建适合于该上下文实现的 Resource，或者通过 String 路径上的特殊前缀，让调用者指定一个必须创建并使用特定的资源实现。

尽管 Spring 经常使用 Resource 接口或通过 `spring` 使用，但实际上，在您自己的代码中单独使用通用接口类作为通用实用工具类来访问资源也非常有用，即使您的代码不了解或不关心 Spring 的任何其他部分。虽然这会将您的代码耦合到 Spring，但实际上仅将其耦合到这套实用程序类，这些实用程序类可作为 URL 的更强大替代，并且您将用于此目的的任何其他库视为等效。



资源抽象不能替代功能。它尽可能地包装它。例如，`UrlResource` 包装一个 URL 并使用该包装的 URL 来完成其工作。

2. 3. 内置资源实现

Spring 包括了一下 `Resource` 实现

- `UrlResource`
- `ClassPathResource`
- `FileSystemResource`
- `ServletContextResource`
- `InputStreamResource`
- `ByteArrayResource`

2.3.1. `UrlResource`

`UrlResource` 包装了 `java.net.URL`，可用于访问通常以 URL 访问的任何对象，例如文件，HTTP 目标，FTP 目标等。所有 URL 都具有标准化的 `String` 表示形式，因此使用适当的标准化前缀来指示另一种 URL 类型。其中包括 `file`: 用于访问文件系统路径，`http`: 通过 HTTP 协议访问资源，`ftp`: 通过 FTP 访问资源以及其他。

`UrlResource` 是由 Java 代码通过显式使用 `UrlResource` 构造函数创建的，但通常在调用带有 `String` 参数表示路径的 API 方法时隐式创建。对于后一种情况，JavaBeans `PropertyEditor` 最终决定要创建哪种类型的资源。如果路径字符串包含众所周知的前缀（例如，`classpath:`），则它将为该前缀创建一个适当的专用资源。但是，如果它不能识别前缀，则假定该字符串是标准 URL 字符串并创建 `UrlResource`。

2.3.2. `ClassPathResource`

此类表示应从类路径获取的资源。它使用线程上下文类加载器，给定的类加载器或给定的类来加载资源。如果类路径资源驻留在文件系统中，而不是驻留在 jar 中并且尚未（通过 `servlet` 引擎或任何环境）将其扩展到文件系统的类路径资源驻留在文件系统中，则此 `Resource` 实现以 `java.io.File` 的形式支持解析。为了解决这个问题，各种 `Resource` 实现始终支持将解析作为 `java.net.URL`。Java 代码通过显式使用 `ClassPathResource` 构造函数来创建 `ClassPathResource`，但通常在调用带有 `String` 参数表示路径的 API 方法时隐式创建 `ClassPathResource`。对于后一种情况，JavaBeans `PropertyEditor` 会识别字符串路径上的特殊前缀 `classpath:`，并在这种情况下创建 `ClassPathResource`。

2.3.3. `FileSystemResource`

这是 `java.io.File` 和 `java.nio.file.Path` 句柄的 `Resource` 实现。它支持作为文件和 URL 的解析。

2.3.4. `ServletContextResource`

这是 `ServletContext` 资源的 `Resource` 实现，它解释相关 Web 应用程序根目录中的相对路径。它始终支持流访问和 URL 访问，但仅在扩展 Web 应用程序档案且资源实际位于文件系统上时才允许 `java.io.File` 访问。它是在文件系统上扩展还是在文件系统上进行扩展，或者直接从 JAR 或其他类似数据库(可以想到的)访问，实际上取决于 `Servlet` 容器。

2.3.5. `InputStreamResource`

`InputStreamResource` 是给定 `InputStream` 的 `Resource` 实现。仅当没有特定的资源实现适用时才应使用它。特别是，尽可能选择 `ByteArrayResource` 或任何基于文件的 `Resource` 实现。

与其他 `Resource` 实现相反，这是一个已经打开的资源的描述符。因此，它从 `isOpen()` 返回 `true`。如果需要将资源描述符保留在某个位置，或者需要多次读取流，请不要使用它。

2.3.6. `ByteArrayResource`

这是给定字节数组的 `Resource` 实现。它为给定的字节数组创建一个 `ByteArrayInputStream`。这对于从任何给定的字节数组加载内容很有用，而不必求助于单次使用的 `InputStreamResource`。

2.4. `ResourceLoader`

`ResourceLoader` 接口旨在由可以返回（即加载）`Resource` 实例的对象实现。以下清单显示了 `ResourceLoader` 接口定义：

Java

```
public interface ResourceLoader {  
    Resource getResource(String location);  
}
```

Kotlin

```
interface ResourceLoader {  
    fun getResource(location: String): Resource  
}
```

所有应用程序上下文都实现 `ResourceLoader` 接口。因此，所有应用程序上下文都可用于获取 `Resource` 实例。

当您在特定的应用程序上下文中调用 `getResource()`，并且指定的位置路径没有特定的前缀时，您将获得适合该特定应用程序上下文的 `Resource` 类型。例如，假设针对以下代

码运行了 `ClassPathXmlApplicationContext` 实例：

Java

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("some/resource/path/myTemplate.txt")
```

针对 `ClassPathXmlApplicationContext`，该代码返回 `ClassPathResource`。如果对 `FileSystemXmlApplicationContext` 实例运行相同的方法，它将返回 `FileSystemResource`。对于 `WebApplicationContext`，它将返回 `ServletContextResource`。类似地，它将为每个上下文返回适当的对象。

因此，你可以以适当的方式将资源加载到特定的应用上下文中。另一方面，您也可以通过指定特殊的 `classpath:` 前缀来强制使用 `ClassPathResource`，而与应用程序上下文类型无关，如下例所示：

Java

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("classpath:some/resource/path/myTemplate.txt")
```

同样，您可以通过指定任何标准 `java.net.URL` 前缀来强制使用 `UrlResource`。以下两个示例使用 `file` 和 `http` 前缀：

Java

```
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("file:///some/resource/path/myTemplate.txt")
```

Java

```
Resource template =
ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");
```

Kotlin

```
val template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt")
```

下表总结了将 `String` 对象转换为 `Resource` 对象的策略：

表 10: Resource strings

前缀	示例	解释
classpath:	classpath:com/myapp/config.xml	从 classpath 中读取
file:	file:///data/config.xml	从文件系统中读取一个 URL。请看 FileSystemResource Caveats (2.7.3) 。
http;	https://myserver/logo.png	作为一个 URL 读取
(none)	/data/config.xml	取决于底层应用程序上下文。

2.5. ResourceLoaderAware 接口

ResourceLoaderAware 接口是个特殊的回调接口，用于标识期望随 ResourceLoader 引用一起提供的组件。以下清单显示了 ResourceLoaderAware 接口的定义：

Java

```
public interface ResourceLoaderAware {
    void setResourceLoader(ResourceLoader resourceLoader);
}
```

Kotlin

```
interface ResourceLoaderAware {
    fun setResourceLoader(resourceLoader: ResourceLoader)
}
```

当一个类实现 ResourceLoaderAware 并部署到应用程序上下文中（作为 Spring 托管的 bean）时，该类被应用程序上下文识别为 ResourceLoaderAware。然后，应用程序上下文调用 `setResourceLoader(ResourceLoader)`，将自身作为参数提供（请记住，Spring 中的所有应用程序上下文都实现 ResourceLoader 接口）。

由于 ApplicationContext 是 ResourceLoader，因此 Bean 也可以实现 ApplicationContextAware 接口，并直接使用提供的应用程序上下文来加载资源。但是，通常，如果需要的话，最好使用专门的 ResourceLoader 接口。该代码将仅耦合到资源加载接口（可以视为实用程序接口），而不耦合到整个 Spring ApplicationContext 接口。

在应用组件中你还可以依靠自动装配 ResourceLoader 来实现 ResourceLoaderAware 接口。“传统”构造函数和 byType 自动装配模式（如“自动装配协作器”中所述）能够分别为构造函数参数或 setter 方法参数提供 ResourceLoader。为了获得更大的灵活性（包括自动装配字段和多个参数方法的能力），请考虑使用基于注解的自动装配功能。在这种

情况下，只要有问题的字段，构造函数或方法带有@Autowired 注解，`ResourceLoader` 就会自动装配到需要 `ResourceLoader` 类型的字段，构造函数参数或方法参数中。有关更多信息，请参见[使用@Autowired](#)。

2.6. 资源依赖

如果 Bean 本身将通过某种动态过程来确定并提供资源路径，那么对于 Bean 来说，使用 `ResourceLoader` 接口加载资源可能是有意义的。例如，考虑加载某种模板，其中所需的特定资源取决于用户的角色。如果资源是静态的，则有必要完全消除对 `ResourceLoader` 接口的使用，让 Bean 公开所需的 `Resource` 属性，并期望将其注入其中。

注入这些属性的麻烦之处在于，所有应用程序上下文都注册并使用了特殊的 JavaBeans `PropertyEditor`，可以将 `String` 路径转换为 `Resource` 对象。因此，如果 `myBean` 具有资源类型的模板属性，则可以为该资源配置个简单的字符串，如以下示例所示：

```
<bean id="myBean" class="...>
    <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

请注意，资源路径没有前缀。因此，由于应用程序上下文本身将用作 `ResourceLoader`，因此根据上下文的确切类型，通过 `ClassPathResource`，`FileSystemResource` 或 `ServletContextResource` 加载资源本身。

如果需要强制使用特定的 `Resource` 类型，则可以使用前缀。以下两个示例显示了如何强制 `ClassPathResource` 和 `UrlResource`（后者用于访问文件系统文件）：

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:///some/resource/path/myTemplate.txt"/>
```

2.7. 应用程序上下文和资源路径

本节介绍如何使用资源创建应用程序上下文，包括使用 XML 的快捷方式，如何使用通配符以及其他详细信息。

2.7.1. 构造应用上下文

应用程序上下文构造函数（针对特定的应用程序上下文类型）通常采用字符串或字符串数组作为资源的位置路径，例如构成上下文定义的 XML 文件。当这样的位置路径没有前缀时，从该路径构建并用于加载 Bean 定义的特定 `Resource` 类型取决于特定应用程序上下文，

并且适用于该特定应用程序上下文。例如，下面这个示例，该示例创建一个

`ClassPathXmlApplicationContext`:

Java

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext("conf/appContext.xml")
```

由于使用了 `ClassPathResource`，因此从类路径中加载了 Bean 定义。但是，请看以下示例，该示例创建一个 `FileSystemXmlApplicationContext`:

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("conf/appContext.xml")
```

现在，bean 定义是从文件系统位置（在这种情况下，是相对于当前工作目录）加载的。

请注意，在位置路径上使用特殊的类路径前缀或标准 URL 前缀会覆盖为加载定义而创建的资源的默认类型。请看这个示例：

Java

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("classpath:conf/appContext.xml")
```

使用 `FileSystemXmlApplicationContext` 从类路径加载 bean 定义。但是，它仍然是 `FileSystemXmlApplicationContext`。如果随后将其用作 `ResourceLoader`，则所有未前缀的路径仍将视为文件系统路径。

构造 `ClassPathXmlApplicationContext` 实例-快捷方式

`ClassPathXmlApplicationContext` 公开了许多构造函数以启用方便的实例化。基本思想是，您只能提供一个字符串数组，该字符串数组仅包含 XML 文件本身的文件名（不包含前导路径信息），并且还提供一个 Class。然后，`ClassPathXmlApplicationContext` 从提供的类中派生路径信息。

请看以下目录布局：

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

以下示例显示如何实例化由在名为 `service.xml` 和 `daos.xml`（位于类路径中）的文件中定义的 bean 组成的 `ClassPathXmlApplicationContext` 实例：

Java

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext(arrayOf("services.xml", "daos.xml"),
    MessengerService::class.java)
```

有关各种构造函数的详细信息，请参见 [ClassPathXmlApplicationContext javadoc](#)。

2.7.2. 应用程序上下文构造函数资源路径中的通配符

应用程序上下文构造函数值中的资源路径可以是简单路径（如先前所示），每个路径都具有到目标资源的一对一映射，或者可以包含特殊的“`classpath*:`”前缀或内部 Ant-样式的正则表达式（通过使用 Spring 的 `PathMatcher` 实用程序进行匹配）。后者都是有效的通配符。

这种机制的一种用途是当您需要进行组件样式的应用程序组装时。所有组件都可以将上下文定义片段“发布”到一个众所周知的位置路径，并且当使用前缀为 `classpath*:` 的相同路径创建最终应用程序上下文时，将自动抓取所有组件片段。

请注意，此通配符特定于在应用程序上下文构造函数中使用资源路径（或当您直接使用 `PathMatcher` 实用工具类层次结构时），并且在构造时已解决。它与资源类型本身无关。您不能使用 `classpath*:` 前缀来构造实际的 `Resource`，因为一个资源点一次仅指向一个资源。

Ant 风格模式串

路径位置可以包含 Ant 样式的模式，如以下示例所示：

```
/WEB-INF/*-context.xml  
com/mycompany/**/applicationContext.xml  
file:C:/some/path/*-context.xml  
classpath:com/mycompany/**/applicationContext.xml
```

路径位置包含 Ant 样式的模式时，解析程序将遵循更复杂的过程来尝试解析通配符。它为到达最后一个非通配符段的路径生成 `Resource`，并从中获取 URL。如果此 URL 不是 `jar:URL` 或特定于容器的变体（例如 WebLogic 中的 `zip`，WebSphere 中的 `wsjar` 等），则从中获取 `java.io.File` 并通过遍历文件系统用于解析通配符。对于 `jar` URL，解析器可以从中获取 `java.net.JarURLConnection` 或手动解析 `jar` URL，然后遍历 `jar` 文件的内容以解析通配符。

对可移植性的影响

如果指定的路径已经是一个文件 URL（由于基本 `ResourceLoader` 是一个文件系统，所以它是隐式的，或者是明确的），则保证通配符可以完全可移植的方式工作。

如果指定的路径是类路径位置，则解析器必须通过调用 `ClassLoader.getResource()` 获得最后的非通配符路径段 URL。由于这只是路径的一个节点（而不是末尾的文件），因此实际上（在 `ClassLoader` javadoc 中）未定义确切返回的是哪种 URL。实际上，它始终是一个 `java.io.File`，代表目录（类路径资源解析为文件系统位置）或某个 `jar` URL（类路径资源解析为 `jar` 位置）。尽管如此，此操作仍存在可移植性问题。如果为最后一个非通配符段获取了 `jar` URL，则解析程序必须能够从中获取 `java.net.JarURLConnection` 或手动解析 `jar` URL，以便能够遍历 `jar` 的内容并解析通配符。这在大多数环境中确实有效，但在其他环境中则无效，因此我们强烈建议您在依赖特定环境之前，对来自 `jars` 的资源的通配符解析进行彻底测试。

`classpath*:` 前缀

在构造基于 XML 的应用程序上下文时，位置字符串可以使用特殊的 `classpath*:` 前缀，如以下示例所示：

Java

```
ApplicationContext ctx =  
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

Kotlin

```
val ctx = ClassPathXmlApplicationContext("classpath*:conf/appContext.xml")
```

这个特殊的前缀指定必须获取与给定名称匹配的所有类路径资源（内部，这本质上是通过调用 `ClassLoader.getResources(...)` 发生的），然后合并以形成最终的应用程序上下文定义。



通配符类路径依赖于基础类加载器的 `getResources()` 方法。由于当今大多数应用程序服务器提供其自己的类加载器实现，因此行为可能有所不同，尤其是在处理 jar 文件时。检查 `classpath*` 是否有效的一个简单测试是使用 `classloader` 从 `classpath` 的 jar 中加载文件：
`getClass().getClassLoader().getResources("<someFileInsideTheJar>")`。尝试对具有相同名称但位于两个不同位置的文件进行此测试。如果返回了不合适的结果，请检查应用程序服务器文档中可能影响类加载器行为的设置。

您还可以在其余位置路径中将 `classpath*:` 前缀与 `PathMatcher` 模式结合使用（例如，`classpath*:META-INF/*-beans.xml`）。在这种情况下，解析策略非常简单：在最后一个非通配符路径段上使用 `ClassLoader.getResources()` 调用，以获取类加载器层次结构中的所有匹配资源，然后从每个资源获取相同的 `PathMatcher` 解析。前面描述的策略用于通配符子路径。

通配符的其他注意事项

请注意，当 `classpath*:` 与 Ant 样式的模式结合使用时，除非模式文件实际存在于目标文件中，否则在模式启动之前，它只能与至少一个根目录可靠地一起工作。这意味着诸如 `classpath*:*.*xml` 之类的模式可能不会从 jar 文件的根目录检索文件，而只会从扩展目录的根目录检索文件。

Spring 检索类路径条目的能力源自 JDK 的 `ClassLoader.getResources()` 方法，该方法仅返回文件系统中的空字符串位置（指示可能要搜索的根目录）。Spring 也会评估 jar 文件中的 `URLClassLoader` 运行时配置和 `java.class.path` 清单，但这不能保证会导致可移植行为。



扫描类路径包需要在类路径中存在相应的目录条目。使用 Ant 构建 JAR 时，请勿激活 JAR 任务的仅文件开关。此外，在某些环境中，基于安全策略，可能不会暴露类路径目录-例如，在 JDK 1.7.0_45 及更高版本上的独立应用程序（要求在清单中设置“受信任的库”。请参阅 <https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources>）。

在 JDK 9 的模块路径（Jigsaw）上，Spring 的类路径扫描通常可以按预期进行。强烈建议在此处将资源放入专用目录中，以避免在搜索 jar 文件根级别时遇到上述可移植性问题。

具有类路径的蚂蚁样式模式：如果要搜索的根包在多个类路径位置可用，则不能保证资源找到匹配的资源。考虑以下资源位置示例：

```
com/mycompany/package1/service-context.xml
```

现在考虑某人可能用来尝试找到该文件的 Ant 样式的路径：

```
classpath:com/mycompany/**/service-context.xml
```

这样的资源可能只在一个位置，但是当使用诸如上一示例的路径尝试对其进行解析时，解析器将处理 `getResource("com/mycompany")`；返回的（第一个）URL。如果此基本包节点存在于多个类加载器位置，则实际的最终资源可能不存在。因此，在这种情况下，您应该更喜欢使用具有相同 Ant 样式模式的 `classpath*:` 来搜索包含根包的所有类路径位置。

2.7.3. FileSystemResource 注意事项

未附加到 `FileSystemApplicationContext` 的 `FileSystemResource`（即，当 `FileSystemApplicationContext` 不是实际的 `ResourceLoader` 时）将按您期望的那样处理绝对路径和相对路径。相对路径是相对于当前工作目录的，而绝对路径是相对于文件系统的根目录。

但是，出于向后兼容性（历史）的原因，当 `FileSystemApplicationContext` 是 `ResourceLoader` 时，情况会发生变化。`FileSystemApplicationContext` 强制所有附加的 `FileSystemResource` 实例将所有位置路径都视为相对位置，无论它们是否以前斜杠开头。实际上，这意味着以下示例是等效的：

Java

```
ApplicationContext ctx =  
    new FileSystemXmlApplicationContext("conf/context.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("conf/context.xml")
```

Java

```
ApplicationContext ctx =  
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

Kotlin

```
val ctx = FileSystemXmlApplicationContext("/conf/context.xml")
```

以下示例也是等效的（即使它们有所不同也有意义，因为一种情况是相对的，另一种情况是绝对的）：

Java

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

Kotlin

```
val ctx: FileSystemXmlApplicationContext = ...
ctx.getResource("some/resource/path/myTemplate.txt")
```

Java

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

Kotlin

```
val ctx: FileSystemXmlApplicationContext = ...
ctx.getResource("/some/resource/path/myTemplate.txt")
```

实际上，如果需要真正的绝对文件系统路径，则应避免将绝对路径与 [FileSystemResource](#) 或 [FileSystemXmlApplicationContext](#) 一起使用，并通过使用 `file: URL` 前缀来强制使用 [UrlResource](#)。以下示例显示了如何执行此操作：

Java

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

Kotlin

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:///some/resource/path/myTemplate.txt")
```

Java

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:///conf/context.xml");
```

Kotlin

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
val ctx = FileSystemXmlApplicationContext("file:///conf/context.xml")
```

3. 验证，数据绑定和类型转换

考虑将验证作为业务逻辑是有利有弊，Spring 提供了一种验证（和数据绑定）设计，但并不排除其中任何一个。具体来说，验证不应与网络层绑定，并且应该易于本地化，并且应该可以插入任何可用的验证器。考虑到这些问题，Spring 提供了一个 `Validator` 包，该包既基本又突出可以在应用程序的每一层中使用。

数据绑定对于使用户输入动态绑定到应用程序（或用于处理用户输入的任何对象）域模型很有用。Spring 提供了恰当地命名为 `DataBinder` 的功能。`Validator` 和 `DataBinder` 组成了验证程序包，该程序包主要用于但不限于 Web 层。

`BeanWrapper` 是 Spring 框架中的一个基本概念，已在很多地方使用，但是您可能不需要直接使用 `BeanWrapper`。但是，由于这是参考文档，因此我们认为可能需要进行一些解释。我们将在本章中解释 `BeanWrapper`，因为如果您要使用它，那么在尝试将数据绑定到对象时最有可能使用它。

Spring 的 `DataBinder` 和较低级别的 `BeanWrapper` 都使用 `PropertyEditorSupport` 实现来解析和格式化属性值。`PropertyEditor` 和 `PropertyEditorSupport` 类型是 JavaBeans 规范的一部分，本章还将对此进行说明。Spring 3 引入了 `core.convert` 包，该包提供了通用的类型转换工具，以及用于格式化 UI 字段值的高级“format”包。您可以将这些包用作 `PropertyEditorSupport` 实现的更简单替代方案。本章还将对它们进行讨论。

Spring 通过设置基础结构和 Spring 自己的 `Validator` 包的适配器来支持 Java Bean 验证。应用程序可以全局启用一次 Bean 验证，如 [Java Bean 验证\(3.7\)](#) 中所述，并将其专用于所有验证需求。在 Web 层中，应用程序可以对每个 `DataBinder` 进一步注册控制器本地的 Spring `Validator` 实例，如[配置 DataBinder\(3.7.3\)](#) 中所述，这对于插入自定义验证逻辑很有用。

3.1. 通过 Spring 的 Validator 接口进行验证

Spring 具有 `Validator` 接口，可用于验证对象。`Validator` 接口通过使用 `Errors` 对象工作，以便在验证时，验证器可以将验证失败报告给 `Errors` 对象。

瞅瞅下面这个小小的数据对象：

Java

```
public class Person {  
  
    private String name;  
    private int age;  
  
    // the usual getters and setters...  
}
```

Kotlin

```
class Person(val name: String, val age: Int)
```

下一个示例通过实现 `org.springframework.validation.Validator` 接口的以下两个方法来提供 `Person` 类的验证行为：

- `support(Class)`:此验证程序可以验证提供的 `Class` 的实例吗？
- `validate(Object, org.springframework.validation.Errors)`:验证给定的对象，并在发生验证错误的情况下，验证给定的 `Errors` 对象的那些注册对象。

实现 `Validator` 非常简单，尤其是当您知道 Spring Framework 也提供的 `ValidationUtils` 帮助器类时。以下示例实现了用于 `Person` 实例的 `Validator`:

Java

```
public class PersonValidator implements Validator {  
  
    /**  
     * This Validator validates only 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");  
        }  
    }  
}
```

```
class PersonValidator : Validator {  
  
    /*  
     * This Validator validates only Person instances  
     */  
    override fun supports(clazz: Class<>): Boolean {  
        return Person::class.java == clazz  
    }  
  
    override fun validate(obj: Any, e: Errors) {  
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty")  
        val p = obj as Person  
        if (p.age < 0) {  
            e.rejectValue("age", "negativevalue")  
        } else if (p.age > 110) {  
            e.rejectValue("age", "too.darn.old")  
        }  
    }  
}
```

`ValidationUtils` 类上的 `static rejectIfEmpty(..)` 方法用于拒绝 name 属性(如果该属性为 `null` 或为空字符串)。看看 `ValidationUtils` javadoc，看看它除了提供前面显示的示例外还提供什么功能。

虽然可以实现单个 `Validator` 类来验证丰富对象中的每个嵌套对象，但最好将每个嵌套对象类的验证逻辑封装在其自己的 `Validator` 实现中。一个“丰富”对象的简单示例是一个由两个 `String` 属性 (第一个和第二个名称) 和一个复杂的 `Address` 对象组成的 `Customer`。`Address` 对象可以独立于客户对象使用，因此已实现了独特的 `AddressValidator`。如果要让 `CustomerValidator` 重用 `AddressValidator` 类中包含的逻辑而不求助于复制粘贴，则可以在 `CustomerValidator` 中依赖注入或实例化一个 `AddressValidator`，如以下示例所示：

Java

```
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();  
        }  
    }  
}
```

```

class CustomerValidator(private val addressValidator: Validator) : Validator {

    init {
        if (addressValidator == null) {
            throw IllegalArgumentException("The supplied [Validator] is required and
must not be null.")
        }
        if (!addressValidator.supports(Address::class.java)) {
            throw IllegalArgumentException("The supplied [Validator] must support the
validation of [Address] instances.")
        }
    }

    /*
     * This Validator validates Customer instances, and any subclasses of Customer too
     */
    override fun supports(clazz: Class<>): Boolean {
        return Customer::class.java.isAssignableFrom(clazz)
    }

    override fun validate(target: Any, errors: Errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
"field.required")
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required")
        val customer = target as Customer
        try {
            errors.pushNestedPath("address")
            ValidationUtils.invokeValidator(this.addressValidator, customer.address,
errors)
        } finally {
            errors.popNestedPath()
        }
    }
}

```

验证错误将报告给传递给验证器的 `Errors` 对象。对于 Spring Web MVC，可以使用 `<spring: bind/>` 标记检查错误消息，但也可以自己检查 `Errors` 对象。关于它提供的方法的更多信息可以在 javadoc 中找到。

3. 2. 将代码解析为错误消息

我们介绍了数据绑定和验证。本节介绍与验证错误相对应的输出消息。在上一节显示的示例中，我们拒绝了名称和年龄字段。如果要使用 `MessageSource` 输出错误消息，则可以使用拒绝字段时提供的错误代码（在这种情况下为“name”和“age”）进行输出。当您从 `Errors` 接口调用（直接或间接地，例如通过使用 `ValidationUtils` 类）`rejectValue` 或其他 `reject` 方法之一时，基础实现不仅会注册您传入的代码，还会注册其他错误代码。

`MessageCodesResolver` 确定 `Errors` 接口注册哪个错误代码。默认情况下，使用

`DefaultMessageCodesResolver`, 它（例如）不仅使用您提供的代码注册消息，而且还注册包含传递给拒绝方法的字段名称的消息。因此，如果您通过使用 `rejectValue("age", "too.darn.old")` 拒绝字段，除了 `too.darn.old` 代码外，Spring 还会注册 `too.darn.old.age` 和 `too.darn.old.age.int`（第一个包含字段名称，第二个包含字段类型）。这样做是为了方便开发人员在定位错误消息时提供帮助。

有关 `MessageCodesResolver` 和替代策略的更多信息，可以分别在 `MessageCodesResolver` 和 `DefaultMessageCodesResolver` 的 javadoc 中找到。

3.3. Bean 操作和 BeanWrapper

`org.springframework.beans` 包遵循 JavaBeans 标准。JavaBean 是具有默认无参数构造函数的类，并且遵循命名约定，在该命名约定下，例如，名为 `bingoMadness` 的属性将具有 `setter` 方法 `setBingoMadness(..)` 和 `getter` 方法 `getBingoMadness()`。有关 JavaBean 和规范的更多信息，请参见 [javabeans](#)。

Bean 包中的一个非常重要的类是 `BeanWrapper` 接口及其相应的实现(`BeanWrapperImpl`)。就像从 Javadoc 引用的那样，`BeanWrapper` 提供了以下功能：设置和获取属性值(单独或批量)，获取属性描述符以及查询属性以确定它们是否可读或可写。此外，`BeanWrapper` 还支持嵌套属性，从而可以将子属性上的属性设置为无限深度。`BeanWrapper` 还支持添加标准 JavaBeans `PropertyChangeListeners` 和 `VetoableChangeListeners` 的功能，而无需在目标类中支持代码。最后但并非最不重要的一点是，`BeanWrapper` 支持设置索引属性。`BeanWrapper` 通常不被应用程序代码直接使用，而是由 `DataBinder` 和 `BeanFactory` 使用。

`BeanWrapper` 的工作方式部分由其名称表示：它包装一个 Bean，以对该 Bean 执行操作，例如设置和检索属性。

3.3.1. 设置和获取基本和内嵌的属性

设置和获取属性是通过 `BeanWrapper` 的 `setProperty` 和 `getProperty` 重载方法完成的。有关详细信息，请参见其 Javadoc。下表显示了这些约定的一些示例：

表 11：属性举例

表达式	解释
<code>name</code>	指示对应于的属性名称 <code>getName()</code> 或 <code>isName()</code> 和 <code>setName(..)</code> 方法。

表达式	解释
<code>account.name</code>	指向属性 <code>account</code> 的内嵌 <code>name</code> 属性关联 <code>getAccount.setName()</code> 方法或者 <code>getAccount.getName()</code> 方法
<code>account[2]</code>	指示索引属性 <code>account</code> 的第三个元素。索引属性的类型可以是数组，列表或其他自然排序的集合。
<code>account[COMPANYNAME]</code>	指示由 <code>account</code> Map 属性的 <code>COMPANYNAME</code> 键索引的 map 的值。

(如果您不打算直接使用 BeanWrapper, 那么下一部分对您而言并不是至关重要的。如果仅使用 DataBinder 和 BeanFactory 及其默认实现, 则应跳到 PropertyEditors 的这一部分。)

以下两个示例类使用 BeanWrapper 来获取和设置属性:

Java

```
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;
    }
}
```

Kotlin

```
class Company {  
    var name: String? = null  
    var managingDirector: Employee? = null  
}
```

Java

```
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;  
    }  
}
```

Kotlin

```
class Employee {  
    var name: String? = null  
    var salary: Float? = null  
}
```

以下代码段显示了一些有关如何检索和操纵实例化的 Companies 和 Employee 的某些属性的示例：

Java

```
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");
```

Kotlin

```
val company = BeanWrapperImpl(Company())
// setting the company name..
company.setPropertyValue("name", "Some Company Inc.")
// ... can also be done like this:
val value = PropertyValue("name", "Some Company Inc.")
company.setPropertyValue(value)

// ok, let's create the director and tie it to the company:
val jim = BeanWrapperImpl(Employee())
jim.setPropertyValue("name", "Jim Stravinsky")
company.setPropertyValue("managingDirector", jim.wrappedInstance)

// retrieving the salary of the managingDirector through the company
val salary = company.getPropertyValue("managingDirector.salary") as Float?
```

3.3.2. 构建 PropertyEditor 实现

Spring 使用 **PropertyEditor** 的概念来实现 **Object** 和 **String** 之间的转换。不同于对象本身的方式表示属性可能很方便。例如，**Date** 可以用人类可读的方式表示（如字符串：“**2007-14-09**”），而我们仍然可以将人类可读的形式转换回原始日期（或者更好的是，转换以人类可读的形式输入的任何日期回 **Date** 对象）。通过注册类型为 **java.beans.PropertyEditor** 的自定义编辑器，可以实现此行为。在 **BeanWrapper** 上或在特定的 IoC 容器中注册自定义编辑器（如上一章所述），使您了解如何将属性转换为所需的类型。有关 **PropertyEditor** 的更多信息，请参见 Oracle 的 **java.beans** 包的 [javadoc](#)。

在 Spring 中使用属性编辑的几个示例：

- 通过使用 **PropertyEditor** 实现在 bean 上设置属性。当使用 **String** 作为在 XML 文件

中声明的某些 bean 的属性的值时, Spring (如果相应属性的设置器具有 `Class` 参数) 将使用 `ClassEditor` 尝试将参数解析为 `Class` 对象。

- 在 Spring 的 MVC 框架中, 通过使用各种 `PropertyEditor` 实现来解析 HTTP 请求参数, 您可以在 `CommandController` 的所有子类中手动绑定这些实现。

Spring 具有许多内置的 `PropertyEditor` 实现来让程序员轻松。它们都位于 `org.springframework.beans.propertyeditors` 包中。默认情况下, 大多数(但不是全部, 如下表)由 `BeanWrapperImpl` 注册。如果可以通过某种方式配置属性编辑器, 则仍可以注册自己的变体以覆盖默认变体。下表描述了 Spring 提供的各种 `PropertyEditor` 实现:

表 12. 内置的 `PropertyEditor` 实现

类	解释
<code>ByteArrayPropertyEditor</code>	字节数组的编辑器。将字符串转换为其相应的字节表示形式。默认情况下由 <code>BeanWrapperImpl</code> 注册。
<code>ClassEditor</code>	将表示类的字符串解析为实际类, 反之亦然。当找不到类时, 将抛出 <code>IllegalArgumentException</code> 。默认情况下, 由 <code>BeanWrapperImpl</code> 注册。
<code>CustomBooleanEditor</code>	<code>Boolean</code> 属性的可定制属性编辑器。默认情况下, 由 <code>BeanWrapperImpl</code> 注册, 但是可以通过注册它的自定义实例作为自定义编辑器。
<code>CustomCollectionEditor</code>	集合的属性编辑器, 可将任何源 <code>Collection</code> 转换为给定的目标 <code>Collection</code> 类型。
<code>CustomDateEditor</code>	<code>java.util.Date</code> 的可自定义属性编辑器, 支持自定义 <code>DateFormat</code> 。默认未注册。必须根据需要以适当的格式进行用户注册。
<code>CustomNumberEditor</code>	任何 <code>Number</code> 子类 (例如 <code>Integer</code> , <code>Long</code> , <code>Float</code> 或 <code>Double</code>) 的可自定义属性编辑器。默认情况下, 由 <code>BeanWrapperImpl</code> 注册, 但是可以通过将其自定义实例注册为自定义编辑器来覆盖。
<code>FileEditor</code>	将字符串解析为 <code>java.io.File</code> 对象。默认情况下, 由 <code>BeanWrapperImpl</code> 注册。

<code>InputStreamEditor</code>	单向属性编辑器，它可以采用字符串并生成(通过中间的 <code>ResourceEditor</code> 和 <code>Resource</code>)一个 <code>InputStream</code> ，以便可以将 <code>InputStream</code> 属性直接设置为字符串。请注意，默认用法不会为您关闭 <code>InputStream</code> 。默认情况下，由 <code>BeanWrapperImpl</code> 注册。
<code>LocaleEditor</code>	可以将字符串解析为 <code>Locale</code> 对象，反之亦然（字符串格式为 <code>[country] [variant]</code> ，与 <code>Locale</code> 的 <code>toString()</code> 方法相同）。 默认情况下，由 <code>BeanWrapperImpl</code> 注册。
<code>PatternEditor</code>	可以将字符串解析为 <code>java.util.regex.Pattern</code> 对象，反之亦然。
<code>PropertiesEditor</code>	可以将字符串（格式设置为 <code>java.util.Properties</code> 类的 javadoc 中定义的格式）转换为 <code>Properties</code> 对象。 默认情况下，由 <code>BeanWrapperImpl</code> 注册。
<code>StringTrimmerEditor</code>	截取字符串的属性编辑器。（可选）允许将空字符串转换为空值。 默认情况下未注册—必须是用户注册的。
<code>URLEditor</code>	可以将 URL 的字符串表示形式解析为实际的 <code>URL</code> 对象。默认情况下，由 <code>BeanWrapperImpl</code> 注册。

Spring 使用 `java.beans.PropertyEditorManager` 设置可能需要的属性编辑器的搜索路径。搜索路径还包括 `sun.bean.editors`，其中包括针对诸如 `Font`, `Color` 和大多数基本类型的类型的 `PropertyEditor` 实现。还要注意，如果标准 JavaBeans 基础结构与它们处理的类在同一包中并且与该类具有相同的名称，并附加了 `Editor`，则标准 JavaBeans 基础结构将自动发现 `PropertyEditor` 类（无需显式注册它们）。例如，一个人可能具有以下类和包结构，足以将 `SomethingEditor` 类识别出来并用作 `Something` 类型的属性的 `PropertyEditor`。

```
com
  chank
    pop
      Something
        SomethingEditor // the PropertyEditor for the Something class
```

注意，您也可以在此处使用标准的 `BeanInfo` JavaBeans 机制（在这某种程度上进行了描述）。以下示例使用 `BeanInfo` 机制使用关联类的属性显式注册一个或多个 `PropertyEditor` 实例：

```
com  
chank  
pop  
Something  
SomethingBeanInfo // the BeanInfo for the Something class
```

所引用的 `SomethingBeanInfo` 类的以下 Java 源代码将 `CustomNumberEditor` 与 `Something` 类的 `age` 属性相关联：

Java

```
public class SomethingBeanInfo extends SimpleBeanInfo {  
  
    public PropertyDescriptor[] getPropertyDescriptors() {  
        try {  
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class,  
true);  
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age",  
Something.class) {  
                public PropertyEditor createPropertyEditor(Object bean) {  
                    return numberPE;  
                };  
                return new PropertyDescriptor[] { ageDescriptor };  
            }  
            catch (IntrospectionException ex) {  
                throw new Error(ex.toString());  
            }  
        }  
    }  
}
```

Kotlin

```
class SomethingBeanInfo : SimpleBeanInfo() {  
  
    override fun getPropertyDescriptors(): Array<PropertyDescriptor> {  
        try {  
            val numberPE = CustomNumberEditor(Int::class.java, true)  
            val ageDescriptor = object : PropertyDescriptor("age",  
Something::class.java) {  
                override fun createPropertyEditor(bean: Any): PropertyEditor {  
                    return numberPE  
                };  
                return arrayOf(ageDescriptor)  
            } catch (ex: IntrospectionException) {  
                throw Error(ex.toString())  
            }  
        }  
    }  
}
```

注册其他自定义 `PropertyEditor` 实现：

当将 bean 属性设置为字符串值时，Spring IoC 容器最终会使用标准 Java Beans `PropertyEditor` 实现将这些字符串转换为属性的复杂类型。Spring 预注册了许多自定义的 `PropertyEditor` 实现（例如，将表示为字符串的类名称转换为 `Class` 对象）。此外，

Java 的标准 JavaBeans `PropertyEditor` 查找机制允许适当地命名类的 `PropertyEditor`, 并将其与提供支持的类放在同一包中, 以便可以自动找到它。

如果需要注册其他自定义 `PropertyEditor`, 则可以使用几种机制。最手动的方法(通常不方便或不建议使用)是使用 `ConfigurableBeanFactory` 接口的 `registerCustomEditor()`方法, 假设您有 `BeanFactory` 引用。另一种(稍微方便些)的机制是使用一种称为 `CustomEditorConfigurer` 的特殊 bean 工厂后处理器。尽管可以将 Bean 工厂后处理器与 `BeanFactory` 实现一起使用, 但 `CustomEditorConfigurer` 具有嵌套的属性设置, 因此我们强烈建议您将其与 `ApplicationContext` 一起使用, 在其中可以将其以与其他任何 Bean 相似的方式进行部署或可以在其中自动检测到并应用的地方。

请注意, 所有 Bean 工厂和应用程序上下文通过使用 `BeanWrapper` 处理属性转换, 都会自动使用许多内置的属性编辑器。上一节中列出了 `BeanWrapper` 注册的标准属性编辑器。另外, `ApplicationContext` 还以适合特定应用程序上下文类型的方式重写或添加其他编辑器, 以处理资源查找。

标准 JavaBeans `PropertyEditor` 实例用于将以字符串表示的属性值转换为该属性的实际复杂类型。可以使用 bean 工厂的后处理器 `CustomEditorConfigurer` 来方便地将对其他 `PropertyEditor` 实例的支持添加到 `ApplicationContext`。考虑以下示例, 该示例定义了一个名为 `ExoticType` 的用户类和另一个名为 `DependsOnExoticType` 的类, 该类需要将 `ExoticType` 设置为属性:

Java

```
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;
    }
}
```

Kotlin

```
package example

class ExoticType(val name: String)

class DependsOnExoticType {

    var type: ExoticType? = null
}
```

正确设置之后，我们希望能够将 `type` 属性分配为字符串，`PropertyEditor` 会将其转换为实际的 `ExoticType` 实例。以下 bean 定义显示了如何建立这种关系：

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type" value="aNameForExoticType"/>
</bean>
```

`PropertyEditor` 实现可能类似于以下内容：

Java

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    public void setAsText(String text) {
        setValue(new ExoticType(text.toUpperCase()));
    }
}
```

Kotlin

```
// converts string representation to ExoticType object
package example

import java.beans.PropertyEditorSupport

class ExoticTypeEditor : PropertyEditorSupport() {

    override fun setAsText(text: String) {
        value = ExoticType(text.toUpperCase())
    }
}
```

最后，以下示例显示如何使用 `CustomEditorConfigurer` 向 `ApplicationContext` 注册新的 `PropertyEditor`，然后可以根据需要使用它：

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType" value="example.ExoticTypeEditor"/>
        </map>
    </property>
</bean>
```

使用 `PropertyEditorRegistrar`

向 Spring 容器注册属性编辑器的另一种机制是创建和使用 `PropertyEditorRegistrar`。当需要在几种不同情况下使用同一组属性编辑器时，此接口特别有用。您可以编写相应的 `registrar`，并在每种情况下重复使用它。`PropertyEditorRegistrar` 实例与一个称为 `PropertyEditorRegistry` 的接口一起工作，该接口由 `Spring BeanWrapper`(和 `DataBinder`) 实现。当与 `CustomEditorConfigurer`(在此描述)结合使用时，`PropertyEditorRegistrar` 实例特别方便，该实例公开了名为 `setPropertyEditorRegistrars(..)` 的属性。以这种方式添加到 `CustomEditorConfigurer` 的 `PropertyEditorRegistrar` 实例可以轻松地与 `DataBinder` 和 Spring MVC 控制器共享。此外，它避免了在自定义编辑器上进行同步的需求：希望 `PropertyEditorRegistrar` 为每次 bean 创建尝试创建新的 `PropertyEditor` 实例。

以下示例说明如何创建自己的 `PropertyEditorRegistrar` 实现：

Java

```
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...
    }
}
```

Kotlin

```
package com.foo.editors.spring

import org.springframework.beans.PropertyEditorRegistrar
import org.springframework.beans.PropertyEditorRegistry

class CustomPropertyEditorRegistrar : PropertyEditorRegistrar {

    override fun registerCustomEditors(registry: PropertyEditorRegistry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType::class.java, ExoticTypeEditor())

        // you could register as many custom property editors as are required here...
    }
}
```

另请参阅 [org.springframework.beans.support.ResourceEditorRegistrar](#) 以获取示例 [PropertyEditorRegistrar](#) 实现。注意，在实现 `registerCustomEditor(..)` 方法时，它如何创建每个属性编辑器的新实例。

下一个示例显示如何配置 [CustomEditorConfigurer](#) 并将其注入我们的 [CustomPropertyEditorRegistrar](#) 的实例：

```
<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"/>
```

最后（对于使用 Spring MVC Web 框架的人来说，与本章的重点有所偏离），将 [PropertyEditorRegistrars](#) 与数据绑定控制器（例如 [SimpleFormController](#)）结合使用会非常方便。下面的示例在 `initBinder(..)` 方法的实现中使用 [PropertyEditorRegistrar](#)：

Java

```
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  
}
```

Kotlin

```
class RegisterUserController(  
    private val customPropertyEditorRegistrar: PropertyEditorRegistrar) :  
    SimpleFormController() {  
  
    protected fun initBinder(request: HttpServletRequest,  
        binder: ServletRequestDataBinder) {  
        this.customPropertyEditorRegistrar.registerCustomEditors(binder)  
    }  
  
    // other methods to do with registering a User  
}
```

这种样式的 `PropertyEditor` 注册可以导致代码简洁(`initBinder(..)`)的实现只有一行长)，并且可以将通用的 `PropertyEditor` 注册代码封装在一个类中，然后根据需要在许多 `Controller` 之间共享。

3. 4. Spring 类型转换

Spring 3 引入了 `core.convert` 包，该包提供了通用的类型转换系统。该系统定义了一个用于实现类型转换逻辑的 SPI 和一个用于在运行时执行类型转换的 API。在 Spring 容器中，可以使用此系统作为 `PropertyEditor` 实现的替代方法，以将外部化的 bean 属性值字符串转换为所需的属性类型。您也可以在应用程序中任何需要类型转换的地方使用公共 API

3. 4. 1. Converter API

如以下接口定义所示，用于实现类型转换逻辑的 SPI 非常简单且具有强类型。

Java

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);
}
```

Kotlin

```
package org.springframework.core.convert.converter

interface Converter<S, T> {

    fun convert(source: S): T
}
```

要创建自己的转换器，请实现 `Converter` 接口，并将 `S` 设置为要转换前的类型，并将 `T` 设置为要转换后的类型。如果需要将 `S` 的集合或数组转换为 `T` 的数组或集合，你还可以透明地应用此类提供了已经注册好委派数组或集合的转换器，（默认情况下 `DefaultConversionService` 会这样做）。

对于每次对 `convert(S)` 的调用，保证源参数不为 `null`。如果转换失败，您的转换器可能会抛出任何未经检查的异常。具体来说，它应该抛出 `IllegalArgumentException` 以报告无效的源值。注意确保您的 `Converter` 实现是线程安全的。

为了方便起见，在 `core.convert.support` 软件包中提供了几种转换器实现。这些包括从字符串到数字以及其他常见类型的转换器。下图显示了 `StringToInteger` 类，它是一个典型的 `Converter` 实现：

Java

```
package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

    public Integer convert(String source) {
        return Integer.valueOf(source);
    }
}
```

Kotlin

```
package org.springframework.core.convert.support

import org.springframework.core.convert.converter.Converter

internal class StringToInteger : Converter<String, Int> {

    override fun convert(source: String): Int? {
        return Integer.valueOf(source)
    }
}
```

3.4.2. 使用 `ConverterFactory`

当需要集中整个类层次结构的转换逻辑时（例如，从 `String` 转换为 `Enum` 对象时），可以实现 `ConverterFactory`，如以下示例所示：

Java

```
package org.springframework.core.convert.converter;

public interfaceConverterFactory<S, R> {

    <T extends R> Converter<S, T> getConverter(Class<T> targetType);
}
```

Kotlin

```
package org.springframework.core.convert.converter

interfaceConverterFactory<S, R> {

    fun <T : R> getConverter(targetType: Class<T>): Converter<S, T>
}
```

参数化 `S` 为你要转换的类型，参数化 `R` 为基本类型，定义可以转换为的类的范围。然后实现 `getConverter(Class <T>)`，其中 `T` 是 `R` 的子类。

以 `StringToEnumConverterFactory` 为例：

Java

```
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());
        }
    }
}
```

3.4.3. 使用 GenericConverter

当您需要复杂的 `Converter` 实现时，请考虑使用 `GenericConverter` 接口。与 `Converter` 相比，`GenericConverter` 具有比 `Converter` 更灵活但类型不太强的签名，支持在多个源类型和目标类型之间进行转换。此外，`GenericConverter` 使您可以在实现转换逻辑时使用可用的源字段和目标字段上下文。这种上下文允许类型转换由字段注解或在字段签名上声明的通用信息驱动。以下清单显示了 `GenericConverter` 的接口定义：

Java

```
package org.springframework.core.convert.converter;

public interface GenericConverter {

    public Set<ConvertiblePair> getConvertibleTypes();

    Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor
targetType);
}
```

Kotlin

```
package org.springframework.core.convert.converter

interface GenericConverter {

    fun getConvertibleTypes(): Set<ConvertiblePair>?

    fun convert(@Nullable source: Any?, sourceType: TypeDescriptor, targetType:
    TypeDescriptor): Any?
}
```

要实现 `GenericConverter`, 请让 `getConvertibleTypes()` 返回支持的源→目标类型对。然后实现 `convert(Object, TypeDescriptor, TypeDescriptor)` 包含您的转换逻辑。源 `TypeDescriptor` 提供对包含正在转换的值的源字段的访问。使用目标 `TypeDescriptor` 可以访问要设置转换值的目标字段。

`GenericConverter` 的一个很好的例子是在 Java 数组和集合之间进行转换的转换器。这样的 `ArrayToCollectionConverter` 会对声明目标集合类型的字段进行内省, 以解析集合的元素类型。这样就可以在将集合设置到目标字段上之前, 将源数组中的每个元素转换为集合元素类型。



由于 `GenericConverter` 是一个更复杂的 SPI 接口, 因此仅应在需要时使用它。支持 `Converter` 或 `ConverterFactory` 以满足基本的类型转换需求。

使用 `ConditionalGenericConverter`

有时, 您希望 `Converter` 仅在满足特定条件时才运行。例如, 您可能只想在目标字段上存在特定注解时才运行 `Converter`, 或者可能仅在目标类上定义了特定方法 (例如静态 `valueOf` 方法) 时才运行 `Converter`。`ConditionalGenericConverter` 是 `GenericConverter` 和 `ConditionalConverter` 接口的并集, 可用于定义以下自定义匹配条件:

Java

```
public interface ConditionalConverter {

    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);
}

public interface ConditionalGenericConverter extends GenericConverter,
ConditionalConverter {
}
```

Kotlin

```
interface ConditionalConverter {  
    fun matches(sourceType: TypeDescriptor, targetType: TypeDescriptor): Boolean  
}  
  
interface ConditionalGenericConverter : GenericConverter, ConditionalConverter
```

`ConditionalGenericConverter` 的一个很好的例子是 `EntityConverter`，它可以在持久实体标识符和实体引用之间进行转换。仅当目标实体类型声明静态查找器方法(例如 `findAccount(Long)`)时，此类 `EntityConverter` 才可能匹配。您可以在 `matchs(TypeDescriptor, TypeDescriptor)` 的实现中执行这种 `finder` 方法检查。

3.4.4. ConversionService API

`ConversionService` 定义了一个统一的 API，用于在运行时执行类型转换逻辑。转换器通常在以下外观界面后面运行：

Java

```
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);  
}
```

Kotlin

```
package org.springframework.core.convert

interface ConversionService {

    fun canConvert(sourceType: Class<*>, targetType: Class<*>): Boolean

    fun <T> convert(source: Any, targetType: Class<T>): T

    fun canConvert(sourceType: TypeDescriptor, targetType: TypeDescriptor): Boolean

    fun convert(source: Any, sourceType: TypeDescriptor, targetType: TypeDescriptor): Any

}
```

大多数 `ConversionService` 还实现 `ConverterRegistry`，该注册器提供用于注册转换器的 SPI。在内部，`ConversionService` 实现委派其注册的转换器执行类型转换逻辑。

`core.convert.support` 软件包中提供了一个强大的 `ConversionService` 实现。`GenericConversionService` 是适用于大多数环境的通用实现。`ConversionServiceFactory` 提供了一个方便的工厂来创建通用的 `ConversionService` 配置。

3.4.5. 配置一个 `ConversionService`

`ConversionService` 是无状态对象，旨在在应用程序启动时实例化，然后在多个线程之间共享。在 Spring 应用程序中，通常为每个 Spring 容(或 `ApplicationContext`)配置一个 `ConversionService` 实例。当框架需要执行类型转换时，Spring 会使用该 `ConversionService` 并使用它。您也可以将此 `ConversionService` 注入到任何 bean 中，然后直接调用它。



如果没有使用 Spring 生成任何 `ConversionService` 的话，那么将会使用基于系统的 `PropertyEditor`。

为使用 Spring 注册一个默认的 `ConversionService`，添加如下的带 `id` 属性 `conversionService`。

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean"/>
```

默认的 `ConversionService` 可以在字符串，数字，枚举，集合，映射和其他常见类型之间进行转换。要用您自己的自定义转换器补充或覆盖默认转换器，请设置 `converters`

属性。属性值可以实现 `Converter`, `ConverterFactory` 或 `GenericConverter` 接口中的任何一个。

```
<bean id="conversionService"
      class="org.springframework.context.support.ConversionServiceFactoryBean">
    <property name="converters">
      <set>
        <bean class="example.MyCustomConverter"/>
      </set>
    </property>
</bean>
```

在 Spring MVC 应用程序中使用 `ConversionService` 也很常见。参见 Spring MVC 一章中的[转换和格式化](#)。有关使用 `FormattingConversionServiceFactoryBean` 的详细信息，请参见 [FormatterRegistry SPI\(3.5.3\)](#)。

3.4.6. 编程式使用 `ConversionService`

要以编程方式使用 `ConversionService` 实例，可以像对其他任何 bean 一样注入对该实例的引用。以下示例显示了如何执行此操作：

Java

```
@Service
public class MyService {

    public MyService(ConversionService conversionService) {
        this.conversionService = conversionService;
    }

    public void doIt() {
        this.conversionService.convert(...)
    }
}
```

Kotlin

```
@Service
class MyService(private val conversionService: ConversionService) {

    fun doIt() {
        conversionService.convert(...)
    }
}
```

对于大多数用例，可以使用指定 `targetType` 的 `convert` 方法，但不适用于更复杂的类型，例如参数化元素的集合。例如，如果要以编程方式将整数列表转换为字符串列表，则需要提供源类型和目标类型的正式定义。

幸运的是，如下面的示例所示，`TypeDescriptor` 提供了各种选项来使操作变得简单明了：

Java

```
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)));
```

Kotlin

```
val cs = DefaultConversionService()

val input: List<Integer> = ...
cs.convert(input,
    TypeDescriptor.forObject(input), // List<Integer> type descriptor
    TypeDescriptor.collection(List::class.java,
    TypeDescriptor.valueOf(String::class.java)))
```

请注意，`DefaultConversionService` 自动注册适用于大多数环境的转换器。这包括集合转换器，标量转换器和基本的 `Object-to-String` 转换器。您可以使用 `DefaultConversionService` 类上的静态 `addDefaultConverters` 方法在任何 `ConverterRegistry` 中注册相同的转换器。

值类型的转换器可重用于数组和集合，因此，假设标准集合处理适当，则无需创建特定的转换器即可将 S 的集合转换为 T 的集合。

3.5. Spring 字段格式化

如前一节所述，`core.convert` 是一种通用类型转换系统。提供统一的 `ConversionService` API 和强类型的 `Converter` SPI，以实现从一种类型到另一种类型的转换逻辑。一个 Spring 容器使用这个系统来绑定 bean 属性值。此外，Spring Expression Language (SpEL) 和 `DataBinder` 都使用此系统绑定字段值。例如，当 SpEL 需要强制将 `Short` 转换为 `Long` 来完成 `expression.setValue(Object bean, Object value)` 尝试时，`core.convert` 系统将执行强制转换。

现在考虑典型客户端环境（例如 Web 或桌面应用程序）的类型转换要求。在这样的环境中，您通常会从 `String` 转换为支持客户端回发过程，然后又转换为 `String` 以支持视图渲染过程。另外，您通常需要本地化 `String` 值。更通用的 `core.convert Converter` SPI 不能直接满足此类格式化要求。为了直接解决这些问题，Spring 3 引入了便利的 `Formatter`

SPI，它为客户端环境提供了 `PropertyEditor` 实现的简单而强大的替代方案。

通常，当您需要实现通用类型转换逻辑时（例如，用于在 `java.util.Date` 和 `Long` 之间进行转换），可以使用 `Converter` SPI。在客户端环境（例如 Web 应用程序）中工作并且需要解析和打印本地化的字段值时，可以使用 `Formatter` SPI。`ConversionService` 为两个 SPI 提供统一的类型转换 API。

3.5.1. Formatter SPI

用于实现字段格式化逻辑的 `Formatter` SPI 非常简单且类型严格。以下清单显示了 `Formatter` 接口定义：

Java

```
package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> { }
```

`Formatter` 从 `Printer` 和 `Parser` 构建块接口扩展。下表显示了这两个接口的定义：

Java

```
public interface Printer<T> {

    String print(T fieldValue, Locale locale);
}
```

Kotlin

```
interface Printer<T> {

    fun print(fieldValue: T, locale: Locale): String
}
```

Java

```
import java.text.ParseException;

public interface Parser<T> {

    T parse(String clientValue, Locale locale) throws ParseException;
}
```

Kotlin

```
interface Parser<T> {  
  
    @Throws(ParseException::class)  
    fun parse(clientValue: String, locale: Locale): T  
}
```

要创建自己的格式化程序，请实现前面显示的格式化程序接口。参数化 `T` 为您希望格式化的对象类型—例如，`java.util.Date`。实现 `print()` 操作以打印 `T` 的实例以在客户端语言环境中显示。实现 `parse()` 操作，以从客户端语言环境返回的格式化表示形式解析 `T` 的实例。如果解析尝试失败，则 `Formatter` 应该抛出 `ParseException` 或 `IllegalArgumentException`。注意确保您的 `Formatter` 实现是线程安全的。

格式子包为方便起见提供了几种 `Formatter` 实现。`Number` 包提供 `NumberStyleFormatter`、`CurrencyStyleFormatter` 和 `PercentStyleFormatter` 来格式化使用 `java.text.NumberFormat` 的 `Number` 对象。`datetime` 包提供了一个 `DateFormatter`，用于使用 `java.text.DateFormat` 格式化 `java.util.Date` 对象。`datetime.joda` 包基于 [Joda-Time 库](http://www.joda.org)(www.joda.org)提供了全面的日期时间格式支持。

以下 `DateFormatter` 是 `Formatter` 实现的示例：

Java

```
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;
    }
}
```

Kotlin

```
class DateFormatter(private val pattern: String) : Formatter<Date> {

    override fun print(date: Date, locale: Locale)
        = getDateFormat(locale).format(date)

    @Throws(ParseException::class)
    override fun parse(formatted: String, locale: Locale)
        = getDateFormat(locale).parse(formatted)

    protected fun getDateFormat(locale: Locale): DateFormat {
        val dateFormat = SimpleDateFormat(this.pattern, locale)
        dateFormat.isLenient = false
        return dateFormat
    }
}
```

Spring 团队欢迎社区推动的 **Formatter** 贡献。请参阅 GitHub 问题(github.com)以做出贡献。

3.5.2. 注解驱动格式化

可以通过字段类型或注解配置字段格式。要将注解绑定到 Formatter，请实现 AnnotationFormatterFactory。以下列表显示了 AnnotationFormatterFactory 接口的定义：

Java

```
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);
}
```

Kotlin

```
package org.springframework.format

interface AnnotationFormatterFactory<A : Annotation> {

    val fieldTypes: Set<Class<*>>

    fun getPrinter(annotation: A, fieldType: Class<*>): Printer<*>

    fun getParser(annotation: A, fieldType: Class<*>): Parser<*>
}
```

要创建一个实现：将 A 参数化为要与格式逻辑关联的字段注解类型，例如 `org.springframework.format.annotation.DateTimeFormat`。让 `getFieldTypes()` 返回可在上面使用注解的字段类型。让 `getPrinter()` 返回 `Printer` 以打印带注解的字段的值。让 `getParser()` 返回 `Parser` 以解析带注解字段的 `clientValue`。

以下示例 `AnnotationFormatterFactory` 实现将 `@NumberFormat` 注解绑定到格式化程序，以指定数字样式或模式：

Java

```
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();
            }
        }
    }
}
```

Kotlin

```
class NumberFormatAnnotationFormatterFactory :  
    AnnotationFormatterFactory<NumberFormat> {  
  
    override fun getFieldTypes(): Set<Class<*>> {  
        return setOf(Short::class.java, Int::class.java, Long::class.java,  
        Float::class.java, Double::class.java, BigDecimal::class.java, BigInteger::class.java)  
    }  
  
    override fun getPrinter(annotation: NumberFormat, fieldType: Class<*>):  
        Printer<Number> {  
        return configureFormatterFrom(annotation, fieldType)  
    }  
  
    override fun getParser(annotation: NumberFormat, fieldType: Class<*>):  
        Parser<Number> {  
        return configureFormatterFrom(annotation, fieldType)  
    }  
  
    private fun configureFormatterFrom(annotation: NumberFormat, fieldType: Class<*>):  
        Formatter<Number> {  
        return if (annotation.pattern.isNotEmpty()) {  
            NumberStyleFormatter(annotation.pattern)  
        } else {  
            val style = annotation.style  
            when {  
                style === NumberFormat.Style.PERCENT -> PercentStyleFormatter()  
                style === NumberFormat.Style.CURRENCY -> CurrencyStyleFormatter()  
                else -> NumberStyleFormatter()  
            }  
        }  
    }  
}
```

要触发格式，可以使用@NumberFormat注解字段，如以下示例所示：

Java

```
public class MyModel {  
  
    @NumberFormat(style=Style.CURRENCY)  
    private BigDecimal decimal;  
}
```

Kotlin

```
class MyModel(  
    @field:NumberFormat(style = Style.CURRENCY) private val decimal: BigDecimal  
)
```

格式化注解 API

`org.springframework.format.annotation` 包中存在一个可移植的格式注解 API。您可以使用 `@NumberFormat` 格式化数字字段（例如 `Double` 和 `Long`），并使用 `@DateTimeFormat` 格式化 `java.util.Date`, `java.util.Calendar`, `Long`（用于毫秒时间戳）以及 JSR-310 `java.time` 和 Joda-Time 值类型。

以下示例使用 `@DateTimeFormat` 将 `java.util.Date` 格式化为 ISO 日期 (yyyy-MMdd)：

Java

```
public class MyModel {  
  
    @DateTimeFormat(iso=ISO.DATE)  
    private Date date;  
}
```

Kotlin

```
class MyModel(  
    @DateTimeFormat(iso= ISO.DATE) private val date: Date  
)
```

3.5.3. FormatterRegistry SPI

`FormatterRegistry` 是用于注册格式器和转换器的 SPI。`FormattingConversionService` 是适用于大多数环境的 `FormatterRegistry` 的实现。您可以通过编程方式或声明方式将此变体配置为 Spring Bean，例如，通过使用 `FormattingConversionServiceFactoryBean`。由于此实现还实现了 `ConversionService`，因此您可以直接将其配置为与 Spring 的 `DataBinder` 和 Spring 表达式语言 (SpEL) 一起使用。

以下清单显示了 `FormatterRegistry` SPI：

Java

```
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);
}
```

Kotlin

```
package org.springframework.format

interface FormatterRegistry : ConverterRegistry {

    fun addFormatterForFieldType(fieldType: Class<*>, printer: Printer<*>, parser:
Parser<*>)

    fun addFormatterForFieldType(fieldType: Class<*>, formatter: Formatter<*>)

    fun addFormatterForFieldType(formatter: Formatter<*>)

    fun addFormatterForAnnotation(factory: AnnotationFormatterFactory<*>)
}
```

如前面的清单所示，您可以按字段类型或注解注册格式化程序。

FormatterRegistry SPI 使您可以集中配置格式设置规则，而不必在控制器之间复制此类配置。例如，您可能需要强制以某种方式设置所有日期字段的格式或以某种方式设置带有特定批注的字段的格式。使用共享的 **FormatterRegistry**，您可以一次定义这些规则，并在需要格式化时应用它们。

3.5.4. FormatterRegistrar SPI

FormatterRegistrar 是一个 SPI，用于通过 **FormatterRegistry** 注册格式器和转换器。以下清单显示了其接口定义：

Java

```
package org.springframework.format;

public interface FormatterRegistrar {

    void registerFormatters(FormatterRegistry registry);
}
```

Kotlin

```
package org.springframework.format

interface FormatterRegistrar {

    fun registerFormatters(registry: FormatterRegistry)
}
```

为给定的格式类别（例如日期格式）注册多个相关的转换器和格式器时，`FormatterRegistrar` 很有用。在声明式注册不足的情况下，它也很有用。例如，当格式化程序需要在不同于其自身<T>的特定字段类型下进行索引时，或者在注册“`Printer/Parser`”对时。

3.5.5. 在 Spring MVC 中配置格式化

参见 Spring MVC 一章中的[转换和格式化](#)。

3.6. 配置一个全局日期时间格式

默认情况下，使用 `DateFormat.SHORT` 样式从字符串转换未使用`@DateTimeFormat` 注解的日期和时间字段。如果愿意，可以通过定义自己的全局格式来更改此设置。

为此，请确保 Spring 不注册默认格式器。相反，可以借助以下方法手动注册格式器：

- `org.springframework.format.datetime.standard.DateTimeFormatterRegistrar`
- `org.springframework.format.datetime.DateFormatterRegistrar`, or
`org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar` 对于 Joda-Time.

例如，以下 Java 配置注册全局 `yyyyMMdd` 格式：

Java

```
@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 JSR-310 date conversion with a specific global format
        DateTimeFormatterRegistrar registrar = new DateTimeFormatterRegistrar();
        registrar.setDateTimeFormatter(DateTimeFormatter.ofPattern("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        // Register date conversion with a specific global format
        DateFormatterRegistrar registrar = new DateFormatterRegistrar();
        registrar.setFormatter(new DateFormatter("yyyyMMdd"));
        registrar.registerFormatters(conversionService);

        return conversionService;
    }
}
```

Kotlin

```
@Configuration
class AppConfig {

    @Bean
    fun conversionService(): FormattingConversionService {
        // Use the DefaultFormattingConversionService but do not register defaults
        return DefaultFormattingConversionService(false).apply {

            // Ensure @NumberFormat is still supported
            addFormatterForFieldAnnotation(NumberFormatAnnotationFormatterFactory())

            // Register JSR-310 date conversion with a specific global format
            val registrar = DateTimeFormatterRegistrar()
            registrar.setDateTimeFormatter(DateTimeFormatter.ofPattern("yyyy-MM-dd"))
            registrar.registerFormatters(this)

            // Register date conversion with a specific global format
            val registrar = DateFormatterRegistrar()
            registrar.setFormatter(DateFormatter("yyyy-MM-dd"))
            registrar.registerFormatters(this)
        }
    }
}
```

如果 您 喜 欢 基 于 XML 的 配 置 , 则 可 以 使 用 [FormattingConversionServiceFactoryBean](#)。以下示例显示了如何执行此操作 (这次 使用 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="yyyy-MMdd"/>
                            </bean>
                        </property>
                    </bean>
                </set>
            </property>
        </bean>
    </beans>

```

请注意，在 Web 应用程序中配置日期和时间格式时，还有其他注意事项。请参阅 [WebMVC 转换和格式](#) 或 [WebFlux 转换和格式](#)。

3. 7. Java Bean 验证

Spring 框架提供了对 [Java Bean 验证\(beanvalidation.org\)](#) API 的支持。

3. 7. 1. bean 验证概述

Bean 验证为 Java 应用程序提供了通过约束声明和元数据进行验证的通用方法。要使用它，您需要使用声明性验证约束对字段模型属性进行注解，然后由运行时强制实施。有内置的约束，您也可以定义自己的自定义约束。

且看以下示例，该示例显示了具有两个属性的简单 PersonForm 模型：

Java

```
public class PersonForm {  
    private String name;  
    private int age;  
}
```

Kotlin

```
class PersonForm(  
    private val name: String,  
    private val age: Int  
)
```

Bean 验证使您可以声明约束，如以下示例所示：

Java

```
public class PersonForm {  
  
    @NotNull  
    @Size(max=64)  
    private String name;  
  
    @Min(0)  
    private int age;  
}
```

Kotlin

```
class PersonForm(  
    @get:NotNull @get:Size(max=64)  
    private val name: String,  
    @get:Min(0)  
    private val age: Int  
)
```

然后，Bean 验证验证器根据声明的约束来验证此类的实例。有关该 API 的一般信息，请参见 [Bean 验证 \(beanvalidation.org\)](#)。有关特定限制，请参见 [Hibernate Validator \(hibernate.org\)](#) 文档。要学习如何将 bean 验证提供程序设置为 Spring bean，请继续阅读。

3.7.2. 配置一个 bean 验证 Provider

Spring 提供了对 Bean 验证 API 的全面支持，包括将 Bean 验证提供程序作为 Spring Bean 进行引导。这样，您可以在应用程序中需要验证的任何地方注入 `javax.validation.ValidatorFactory` 或 `javax.validation.Validator`。

您可以使用 `LocalValidatorFactoryBean` 将默认的 Validator 配置为 Spring

Bean，如以下示例所示：

Java

```
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class AppConfig {

    @Bean
    public LocalValidatorFactoryBean validator() {
        return new LocalValidatorFactoryBean();
    }
}
```

XML

```
<bean id="validator"
      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>
```

前面示例中的基本配置触发 Bean 验证以使用其默认引导机制进行初始化。Bean 验证提供程序（例如 Hibernate Validator）应该存在于类路径中并被自动检测到。

注入一个验证器

`LocalValidatorFactoryBean` 同时实现 `javax.validation.ValidatorFactory` 和 `javax.validation.Validator` 以及 Spring 的 `org.springframework.validation.Validator`。您可以将对这些接口之一的引用注入需要调用验证逻辑的 bean 中。

如果 您 更 喜 欢 直 接 使 用 Bean Validation API，则 可 以 注 入 对 `javax.validation.Validator` 的引用，如以下示例所示：

Java

```
import javax.validation.Validator;

@Service
public class MyService {

    @Autowired
    private Validator validator;
}
```

Kotlin

```
import javax.validation.Validator;  
  
@Service  
class MyService(@Autowired private val validator: Validator)
```

如果您的 bean 需要使用 Spring Validation API，则可以注入对 org.springframework.validation.Validator 的引用，如以下示例所示：

Java

```
import org.springframework.validation.Validator;  
  
@Service  
public class MyService {  
  
    @Autowired  
    private Validator validator;  
}
```

Kotlin

```
import org.springframework.validation.Validator  
  
@Service  
class MyService(@Autowired private val validator: Validator)
```

配置自定义约束

每个 bean 验证约束都包括两个部分：

- `@Constraint` 注解，用于声明约束及其可配置属性。
- `javax.validation.ConstraintValidator` 接口的实现，用于实现约束的行为。

要将声明与实现相关联，每个 `@Constraint` 批注都引用一个对应的 `ConstraintValidator` 实现类。在运行时，当在字段模型中遇到约束注解时，`ConstraintValidatorFactory` 表示为一个引用的实现。

默认情况下，`LocalValidatorFactoryBean` 配置一个 `SpringConstraintValidatorFactory`，该工厂使用 Spring 创建 `ConstraintValidator` 实例。这使您的自定义 `ConstraintValidators` 像其他任何 Spring bean 一样受益于依赖项注入。

以下示例显示了一个自定义 `@Constraint` 声明，后跟一个关联的 `ConstraintValidator` 实现，该实现使用 Spring 进行依赖项注入：

Java

```
@Target({ElementType.METHOD, ElementType.FIELD})  
@Retention(RetentionPolicy.RUNTIME)  
@Constraint(validatedBy=MyConstraintValidator.class)  
public @interface MyConstraint {  
}
```

Kotlin

```
@Target(AnnotationTarget.FUNCTION, AnnotationTarget.FIELD)  
@Retention(AnnotationRetention.RUNTIME)  
@Constraint(validatedBy = MyConstraintValidator::class)  
annotation class MyConstraint
```

Java

```
import javax.validation.ConstraintValidator;  
  
public class MyConstraintValidator implements ConstraintValidator {  
  
    @Autowired;  
    private Foo aDependency;  
  
    // ...  
}
```

Kotlin

```
import javax.validation.ConstraintValidator  
  
class MyConstraintValidator(private val aDependency: Foo) : ConstraintValidator {  
  
    // ...  
}
```

如前面的示例所示，**ConstraintValidator** 实现可以像其他任何 Spring bean 一样具有**@Autowired** 的依赖项。

Spring 驱动方法验证

您可以通过 MethodValidationPostProcessor bean 定义将 Bean Validation 1.1 (以及作为自定义扩展，还包括 Hibernate Validator 4.3) 支持的方法验证功能集成到 Spring 上下文中：

Java

```
import org.springframework.validation.beanvalidation.MethodValidationPostProcessor;

@Configuration
public class AppConfig {

    @Bean
    public MethodValidationPostProcessor validationPostProcessor() {
        return new MethodValidationPostProcessor();
    }
}
```

XML

```
<bean
    class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>
```

为了能进行 Spring 驱动的方法验证，所有目标类都必须使用 Spring 的 `@Validated` 注解进行注解，该注解也可以选择声明要使用的验证组。`MethodValidationPostProcessor`，用于使用 Hibernate Validator 和 Bean Validation 1.1 提供程序进行设置的详细信息。



方法验证依赖于目标类周围的 AOP 代理(5.3)，即接口上方法的 JDK 动态代理或 CGLIB 代理。代理的使用存在某些限制，[了解 AOP 代理\(5.8.1\)](#) 中介绍了其中的一些限制。另外，请记住在代理类上始终使用方法和访问器；字段直接访问将不起作用。

额外的配置选项

在大多数情况下，默认的 `LocalValidatorFactoryBean` 配置就足够了。从消息插值到遍历解析，有许多用于各种 Bean 验证构造的配置选项。有关这些选项的更多信息，请参见 [LocalValidatorFactoryBean\(docs.spring.io\) Javadoc](#)。

3.7.3. 配置一个 DataBinder

从 Spring 3 开始，您可以使用 `Validator` 配置 `DataBinder` 实例。配置完成后，您可以通过调用 `binder.validate()` 来调用 `Validator`。任何验证错误都会自动添加到活绑定器的 `BindingResult` 中。

下面的示例演示如何在绑定到目标对象后，以编程方式使用 `DataBinder` 来调用验证逻辑：

Java

```
Foo target = new Foo();
DataBinder binder = new DataBinder(target);
binder.setValidator(new FooValidator());

// bind to the target object
binder.bind(propertyValues);

// validate the target object
binder.validate();

// get BindingResult that includes any validation errors
BindingResult results = binder.getBindingResult();
```

Kotlin

```
val target = Foo()
val binder = DataBinder(target)
binder.validator = FooValidator()

// bind to the target object
binder.bind(propertyValues)

// validate the target object
binder.validate()

// get BindingResult that includes any validation errors
val results = binder.bindingResult
```

您还可以通过 `dataBinder.addValidators` 和 `dataBinder.replaceValidators` 配置具有多个 `Validator` 实例的 `DataBinder`。当将全局配置的 bean 验证与在 `DataBinder` 实例上本地配置的 Spring Validator 结合使用时，这很有用。请参阅 [Spring MVC 验证配置](#)。

3.7.4. Spring MVC 3 验证

参见 Spring MVC 一章中的验证。

4. Spring 表达式语言(SpEL)

Spring 表达式语言（简称“SpEL”）是一种功能强大的表达式语言，支持在运行时查询和操作对象图。语言语法类似于 Unified EL，但提供了其他功能，最著名的是方法调用和基本的字符串模板功能。

尽管还有其他几种 Java 表达式语言可用 -OGNL, MVEL 和 JBoss EL，仅举几例-Spring 表达式语言的创建是为了向 Spring 社区提供一种受良好支持的表达式语言，该语言可用于 Spring 产品集以下版本中的所有产品。它的语言功能受 Spring 产品组合中项目的要求驱动，包括 [Spring Tools for Eclipse](#) 中代码完成支持的工具要求。也就是说，SpEL 基于与技术无关的 API，如果需要，可以将其他表达语言实现集成在一起。

虽然 SpEL 是 Spring 产品组合中表达式求值的基础，但它并不直接与 Spring 绑定，可以独立使用。为了自成一体，本章中的许多示例都将 SpEL 当作一种独立的表达语言来使用。这需要创建一些自举基础结构类，例如解析器。大多数 Spring 用户不需要处理这种基础结构，而只需编写表达式字符串进行求值。这种典型用法的一个示例是将 SpEL 集成到创建 XML 或基于注解的 Bean 定义中，如 [Expression 支持对 Bean 的定义](#) 所示。

本章介绍了表达语言，其 API 和语法规的功能。在许多地方，Inventor 和 Society 类都用作表达求值的目标对象。这些类声明以及用于填充它们的数据在本章末尾列出。

表达式语言支持以下功能：

- 文字表达式
- 属性，数组，列表，Maps 和索引器
- 内联列表
- 内联 Maps
- 数组结构
- 方法
- 操作符
- 类型
- 构造器
- 变量
- 函数

- Bean 引用
- 三元操作符
- Elvis 操作符
- 安全导航操作符
- 集合选择
- 集合 projection
- 表达式模板

4.1. 求值

本节介绍 SpEL 接口的简单用法及其表达语言。完整的信息可以在[语言参考](#)中找到。

以下代码介绍了 SpEL API，用于评估文字字符串表达式 `Hello World`。

Java

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'"); ①
String message = (String) exp.getValue();
```

① `message` 变量的值是 '`hello world`'。

Kotlin

```
val parser = SpelExpressionParser()
val exp = parser.parseExpression("'Hello World'") ①
val message = exp.value as String
```

① `message` 变量的值是 '`hello world`'。

您最可能使用的 SpEL 类和接口位于 `org.springframework.expression` 包及其子包中，例如 `spel.support`。

`ExpressionParser` 接口负责解析表达式字符串。在前面的示例中，表达式字符串是由周围的单引号表示的字符串文字。`Expression` 接口负责评估先前定义的表达式字符串。分别调用 `parser.parseExpression` 和 `exp.getValue` 时，可以引发两个异常 `ParseException` 和 `EvaluationException`。

SpEL 支持多种功能，例如调用方法，访问属性和调用构造函数。

在以下方法调用示例中，我们在字符串文字上调用 `concat` 方法：

Java

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!'))"); ①
String message = (String) exp.getValue();
```

① `message` 变量的值现在是 '`hello world`'.

Kotlin

```
val parser = SpelExpressionParser()
val exp = parser.parseExpression("'Hello World'.concat('!'))") ①
val message = exp.value as String
```

① `message` 变量的值现在是 '`hello world`'.

以下调用 JavaBean 属性的示例将调用 `String` 的 `Bytes` 属性：

Java

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes"); ①
byte[] bytes = (byte[]) exp.getValue();
```

① 这行将文字转换为字节数组。

Kotlin

```
val parser = SpelExpressionParser()

// invokes 'getBytes()'
val exp = parser.parseExpression("'Hello World'.bytes") ①
val bytes = exp.value as ByteArray
```

① 这行将文字转换为字节数组。

SpEL 还通过使用标准的点符号（例如 `prop1.prop2.prop3`）以及相应的属性值设置来支持嵌套属性。 也可以访问公共字段。

下面的示例演示如何使用点表示法获取文字的长度：

Java

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length"); ①
int length = (Integer) exp.getValue();
```

① '`hello World'.bytes.length` 给出文字的长度。

Kotlin

```
val parser = SpelExpressionParser()  
  
// invokes 'getBytes().length'  
val exp = parser.parseExpression("'Hello World'.bytes.length") ①  
val length = exp.value as Int
```

① ‘hello World’.bytes.length 给出文字的长度。

可以调用 String 的构造函数，而不是使用字符串文字，如以下示例所示：

Java

```
ExpressionParser parser = new SpelExpressionParser();  
Expression exp = parser.parseExpression("new String('hello world').toUpperCase()"); ①  
String message = exp.getValue(String.class);
```

① 从文字构造一个新的 String 并将其变为大写。

Kotlin

```
val parser = SpelExpressionParser()  
val exp = parser.parseExpression("new String('hello world').toUpperCase()") ①  
val message = exp.getValue(String::class.java)
```

① 从文字构造一个新的 String 并将其变为大写。

注意使用通用方法： `public <T> T getValue(Class<T> requiredResultType)`。

使用此方法无需将表达式的值强制转换为所需的结果类型。如果该值不能转换为 `T` 类型或无法使用已注册的类型转换器转换，则将引发 `EvaluationException`。

SpEL 的更常见用法是提供一个表达式字符串，该字符串针对特定对象实例（称为根对象）进行评估。以下示例显示如何从 `Inventor` 类的实例检索 `name` 属性或如何创建布尔条件：

```

// 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"); // Parse name as an expression
String name = (String) exp.getValue(tesla);
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(tesla, Boolean.class);
// result == true

```

Kotlin

```

// Create and set a calendar
val c = GregorianCalendar()
c.set(1856, 7, 9)

// The constructor arguments are name, birthday, and nationality.
val tesla = Inventor("Nikola Tesla", c.time, "Serbian")

val parser = SpelExpressionParser()

var exp = parser.parseExpression("name") // Parse name as an expression
val name = exp.getValue(tesla) as String
// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'")
val result = exp.getValue(tesla, Boolean::class.java)
// result == true

```

4.1.1. 理解求值上下文

在求值表达式以解析属性，方法或字段并帮助执行类型转换时，使用 `EvaluationContext` 接口。Spring 提供了两种实现。

- `SimpleEvaluationContext`: 针对不需要全部 SpEL 语言语法范围且应受到有意义限制的表达式类别，公开了 SpEL 基本语言特性和配置选项的子集。示例包括但不限于数据绑定表达式和基于属性的过滤器。
- `StandardEvaluationContext`: 公开了全套 SpEL 语言功能和配置选项。您可以使用它来指定默认的根对象，并配置每个可用的计算相关策略。

`SimpleEvaluationContext` 设计为仅支持 SpEL 语言语法的子集。它不包括 Java 类

型引用，构造函数和 Bean 引用。它还要求您明确选择对表达式中的属性和支持级别的支持级别。默认情况下，`create()`静态工厂方法仅启用对属性的读取访问。您还可以获取构建器来配置所需的确切支持级别，并针对以下一种或某些组合：

- 仅自定义 `PropertyAccessor`（无反射）
- 只读访问的数据绑定属性
- 读写的数据绑定属性

类型转换

默认情况下，SpEL 使用 Spring Core 中可用的转换服务 (`org.springframework.core.convert.ConversionService`)。此转换服务随附有许多用于常见转换的内置转换器，但它也是完全可扩展的，因此您可以在类型之间添加自定义转换。此外，它是泛型感知的。这意味着，当您在表达式中使用泛型类型时，SpEL 会尝试进行转换以维护遇到的任何对象的类型正确性。

在实践中这意味着什么？假设使用 `setValue()` 进行赋值来设置 `List` 属性。该属性的类型实际上是 `List<Boolean>`。SpEL 认识到列表中的元素在放入列表之前需要转换为布尔值。以下示例显示了如何执行此操作：

Java

```
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 is false  
Boolean b = simple.booleanList.get(0);
```

```
class Simple {  
    var booleanList: MutableList<Boolean> = ArrayList()  
}  
  
val simple = Simple()  
simple.booleanList.add(true)  
  
val 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 is false  
val b = simple.booleanList[0]
```

4.1.2. 解析器配置

可以使用解析器配置对象([org.springframework.expression.spel.SpelParserConfiguration](#))配置 SpEL 表达式解析器。配置对象控制某些表达式组件的行为。例如，如果您索引到数组或集合中并且指定索引处的元素为 `null`，则可以自动创建该元素。当使用由属性引用链组成的表达式时，这很有用。如果索引到数组或列表中，并且指定的索引超出了数组或列表当前大小的末尾，则可以自动增长数组或列表以容纳该索引。下面的示例演示如何自动增加列表：

Java

```
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
```

Kotlin

```
class Demo {  
    var list: List<String>? = null  
}  
  
// Turn on:  
// - auto null reference initialization  
// - auto collection growing  
val config = SpelParserConfiguration(true, true)  
  
val parser = SpelExpressionParser(config)  
  
val expression = parser.parseExpression("list[3]")  
  
val demo = Demo()  
  
val o = expression.getValue(demo)  
  
// demo.list will now be a real collection of 4 entries  
// Each entry is a new empty String
```

4.1.3. SpEL 并发

Spring Framework 4.1 包含一个基本的表达式编译器。通常对表达式进行解释，这在评估过程中提供了很多动态灵活性，但没有提供最佳性能。对于偶尔使用表达式，这很好，但是，当与其他组件（如 Spring Integration）一起使用时，性能可能非常重要，并且不需要动态性。

SpEL 编译器旨在满足这一需求。在评估过程中，编译器生成一个 Java 类，该类体现了运行时的表达式行为，并使用该类来实现更快的表达式评估。由于缺少在表达式周围输入内容的信息，因此编译器在执行编译时会使用在表达式的解释求值期间收集的信息。例如，它不仅仅从表达式中就知道属性引用的类型，而是在第一次解释求值时就知道它是什么。当然，如果各种表达元素的类型随时间变化，则基于此类派生信息进行编译会在以后引起麻烦。因此，编译最适合类型信息在重复求值时不会改变的表达式。

考虑以下基本表达式：

```
someArray[0].someProperty.someOtherProperty < 0.1
```

因为前面的表达式涉及数组访问，一些属性取消引用和数字运算，所以性能提升可能非常明显。在一个示例中，进行了 50000 次迭代的微基准测试，使用解释器评估需要 75 毫秒，而使用表达式的编译版本仅需要 3 毫秒。

编译期配置

默认情况下不打开编译器，但是您可以通过两种不同的方式之一来打开它。当 SpEL 用法嵌入到另一个组件中时，可以使用解析器配置过程（[前面讨论过](#)）或使用系统属性来打开它。本节讨论这两个选项。

编译器可以在 `org.springframework.expression.spel.SpelCompilerMode` 枚举中捕获的三种模式之一进行操作。模式如下：

- `OFF`(默认)：编译器关闭。
- `IMMEDIATE`：在 `immediate` 模式。表达式尽可能的快编译。通常是在第一次解析计算之后。如果编译的表达式失败（通常是由类型更改，如前所述），则表达式求值的调用者将收到异常。
- `MIXED`：在 `mixed` 模式，随着时间的推移，这些表达式会在解释模式和编译模式之间默默地切换。经过一定数量的解释运行后，它们会切换到编译形式，如果编译形式出了问题（例如，如前所述的类型更改），则表达式会自动再次切换回解释形式。稍后，它可能会生成另一个已编译的表单并切换到该表单。基本上，用户进入 `IMMEDIATE` 模式的异常是在内部处理的。

存在 `IMMEDIATE` 模式是因为 `MIXED` 模式可能会导致具有副作用的表达式出现问题。如果已编译的表达式在部分成功后就崩溃了，则它可能已经完成了影响系统状态的操作。如果发生这种情况，调用者可能不希望它在解释模式下静默地重新运行，因为表达式的一部分可

能运行了两次。

选择模式后，使用 `SpelParserConfiguration` 配置解析器。以下示例显示了如何执行此操作：

Java

```
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);
```

Kotlin

```
val config = SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,  
    this.javaClass.classLoader)  
  
val parser = SpelExpressionParser(config)  
  
val expr = parser.parseExpression("payload")  
  
val message = MyMessage()  
  
val payload = expr.getValue(message)
```

当指定编译器模式时，还可以指定一个类加载器（允许传递 `null`）。编译的表达式是在提供的任何子类加载器中定义的。重要的是要确保，如果指定了类加载器，则它可以查看表达式评估过程中涉及的所有类型。如果未指定类加载器，则使用默认的类加载器（通常是在表达式求值期间运行的线程的上下文类加载器）。第二种配置编译器的方法是将 SpEL 嵌入到其他组件中，并且可能无法通过配置对象进行配置。在这些情况下，可以使用系统属性。您可以将 `spring.expression.compiler.mode` 属性设置为 `SpelCompilerMode` 枚举值之一（`off`, `immediate` 或 `mixed`）。

编译限制

从 Spring Framework 4.1 开始，已经有了基本的编译框架。但是，该框架尚不支持编译每种表达式。最初的重点是可能在性能关键型上下文中使用的通用表达式。目前无法编译以下类型的表达式：

- 涉及赋值的表达

- 表达式依赖转换服务
- 使用自定义解析器或访问器的表达式
- 使用选择或 projection 的表达式

将来会编译更多类型的表达式。

4.2. bean 定义中的表达式

您可以将 SpEL 表达式与基于 XML 或基于注解的配置元数据一起使用，以定义 BeanDefinition 实例。在这两种情况下，用于定义表达式的语法都采用 `# {<expression string>}` 的形式。

4.2.1. XML 配置

可以使用表达式来设置属性或构造函数的参数值，如以下示例所示：

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>
    <!-- other properties -->
</bean>
```

应用程序上下文中的所有 bean 都可以使用其公共 bean 名称作为预定义变量使用。这包括标准上下文 Bean，这包括标准上下文 Bean，例如环境 (`org.springframework.core.env.Environment` 类型) 以及用于访问运行时环境的 `systemProperties` 和 `systemEnvironment(Map<String, Object>)` 类型)。

下面的示例显示了如何将 SpPro 变量作为对 `systemProperties` bean 的访问：

```
<bean id="taxCalculator" class="org.springframework.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>
    <!-- other properties -->
</bean>
```

注意，此处不必在预定义变量前加上 # 符号。

此外你也可以按名字引用其他 bean 的属性，就像下面这个例子一样：

```

<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">
    <property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

    <!-- other properties -->
</bean>

```

4.2.2. 注解配置

若要指定默认值，可以将@Value注解放置在字段、方法以及方法或构造函数参数上。

下面的示例设置字段变量的默认值：

Java

```

public class FieldValueTestBean {

    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }
}

```

Kotlin

```

class FieldValueTestBean {

    @Value("#{ systemProperties['user.region'] }")
    var defaultLocale: String? = null
}

```

以下示例显示了等效的但使用属性设置器方法的示例：

Java

```
public class PropertyValueTestBean {  
  
    private String defaultLocale;  
  
    @Value("#{ systemProperties['user.region'] }")  
    public void setDefaultLocale(String defaultLocale) {  
        this.defaultLocale = defaultLocale;  
    }  
  
    public String getDefaultLocale() {  
        return this.defaultLocale;  
    }  
}
```

Kotlin

```
class PropertyValueTestBean {  
  
    @Value("#{ systemProperties['user.region'] }")  
    var defaultLocale: String? = null  
}
```

自动装配的方法和构造函数也可以使用@Value注解，如以下示例所示：

Java

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
    private String defaultLocale;  
  
    @Autowired  
    public void configure(MovieFinder movieFinder,  
        @Value("#{ systemProperties['user.region'] }") String defaultLocale) {  
        this.movieFinder = movieFinder;  
        this.defaultLocale = defaultLocale;  
    }  
  
    // ...  
}
```

Kotlin

```
class SimpleMovieLister {

    private lateinit var movieFinder: MovieFinder
    private lateinit var defaultLocale: String

    @Autowired
    fun configure(movieFinder: MovieFinder,
                  @Value("#{ systemProperties['user.region'] }") defaultLocale: String)
    {
        this.movieFinder = movieFinder
        this.defaultLocale = defaultLocale
    }

    // ...
}
```

Java

```
public class MovieRecommender {

    private String defaultLocale;

    private CustomerPreferenceDao customerPreferenceDao;

    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao,
                           @Value("#{systemProperties['user.country']}") String defaultLocale) {
        this.customerPreferenceDao = customerPreferenceDao;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

Kotlin

```
class MovieRecommender(private val customerPreferenceDao: CustomerPreferenceDao,
                      @Value("#{systemProperties['user.country']}") private val defaultLocale:
String) {
    // ...
}
```

4.3. 语言引用

本节描述了 Spring Expression Language 的工作方式。它涵盖以下主题：

- [文字表达式 \(4.3.1\)](#)
- [属性, 数组, 列表, Maps 和索引器 \(4.3.2\)](#)
- [内联列表 \(4.3.3\)](#)
- [内联 Maps \(4.3.4\)](#)
- [数组结构 \(4.3.5\)](#)

- [方法 \(4.3.6\)](#)
- [操作符 \(4.3.7\)](#)
- [类型 \(4.3.8\)](#)
- [构造器 \(4.3.9\)](#)
- [变量 \(4.3.10\)](#)
- [函数 \(4.3.11\)](#)
- [Bean 引用 \(4.3.12\)](#)
- [三元操作符 \(4.3.13\)](#)
- [Elvis 操作符 \(4.3.14\)](#)
- [安全导航操作符 \(4.3.15\)](#)

4.3.1. 文字表达式

支持的文字表达式的类型为字符串，数值（int，实数，十六进制），布尔值和 null。字符串由单引号引起。要将单引号本身放在字符串中，请使用两个单引号字符。

以下清单显示了文字的简单用法。通常，它们不是像这样孤立地使用，而是作为更复杂的表达式的一部分使用 - 例如，在逻辑比较运算符的一侧使用文字。

Java

```
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();
```

Kotlin

```
val parser = SpelExpressionParser()

// evals to "Hello World"
val helloWorld = parser.parseExpression("'Hello World'").value as String

val avogadrosNumber = parser.parseExpression("6.0221415E+23").value as Double

// evals to 2147483647
val maxValue = parser.parseExpression("0x7FFFFFFF").value as Int

val trueValue = parser.parseExpression("true").value as Boolean

val nullValue = parser.parseExpression("null").value
```

数字支持使用负号，指数符号和小数点。默认情况下，使用 `Double.parseDouble()` 解析实数。

4.3.2. 属性，数组，列表，Maps 和索引器

使用属性引用进行导航很容易。为此，请使用句点来指示嵌套的属性值。`Inventor` 类的实例 `pupin` 和 `tesla` 用在示例部分中使用的类中列出的数据填充。要向下导航并获取 `tesla` 的出生年份和 `pupin` 的出生城市，我们使用以下表达式：

Java

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year +
1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);
```

Kotlin

```
// evals to 1856
val year = parser.parseExpression("Birthdate.Year + 1900").getValue(context) as Int

val city = parser.parseExpression("placeOfBirth.City").getValue(context) as String
```

属性名称的首字母允许不区分大小写。数组和列表的内容通过使用方括号表示法获得，如以下示例所示：

Java

```
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);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

// Inventions Array

// evaluates to "Induction motor"
val invention = parser.parseExpression("inventions[3]").getValue(
    context, tesla, String::class.java)

// Members List

// evaluates to "Nikola Tesla"
val name = parser.parseExpression("Members[0].Name").getValue(
    context, ieee, String::class.java)

// List and Array navigation
// evaluates to "Wireless communication"
val invention = parser.parseExpression("Members[0].Inventions[6]").getValue(
    context, ieee, String::class.java)
```

通过在方括号内指定文字键值可以获取映射的内容。 在下面的示例中，由于 Officer 地图的键是字符串，因此我们可以指定字符串文字：

Java

```
// 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");
```

Kotlin

```
// Officer's Dictionary

val pupin = parser.parseExpression("Officers['president']").getValue(
    societyContext, Inventor::class.java)

// evaluates to "Idvor"
val city = parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(
    societyContext, String::class.java)

// setting values
parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(
    societyContext, "Croatia")
```

4. 3. 3. 内联列表

您可以使用`{}`符号在表达式中直接表达列表。

Java

```
// 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);
```

Kotlin

```
// evaluates to a Java list containing the four numbers
val numbers = parser.parseExpression("{1,2,3,4}").getValue(context) as List<*>

val listOfLists = parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context) as
List<*>
```

{ }本身表示一个空列表。出于性能原因，如果 list 本身完全由固定的文字组成，则会创建一个常量 list 来表示该表达式（而不是在每次求值时都构建一个新 list）。

4. 3. 4. 内联 Maps

您也可以使用 {key: value} 表示法在表达式中直接表达 Map。以下示例显示了如何执行此操作：

Java

```
// 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',y
ear:1856}}").getValue(context);
```

Kotlin

```
// evaluates to a Java map containing the two entries
val inventorInfo = parser.parseExpression("{name:'Nikola',dob:'10-July-
1856'}").getValue(context) as Map<*, >

val mapOfMaps =
parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',y
ear:1856}}").getValue(context) as Map<*, *>
```

{ : }本身就是一个空的 Map。出于性能原因，如果 Map 本身由固定的文字或其他嵌套的常量结构（list 或 map）组成，则会创建一个常量 map 来表示该表达式（而不是在每次求值时都新建一个 map）。Map 的 key 的引用是可选的。上面的示例不使用带引号的键。

4. 3. 5. 数组结构

您可以使用熟悉的 Java 语法来构建数组，可以选择提供一个初始化程序以在构造时填充该数组。以下示例显示了如何执行此操作：

Java

```
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);
```

Kotlin

```
val numbers1 = parser.parseExpression("new int[4]").getValue(context) as IntArray

// Array with initializer
val numbers2 = parser.parseExpression("new int[]{1,2,3}").getValue(context) as
IntArray

// Multi dimensional array
val numbers3 = parser.parseExpression("new int[4][5]").getValue(context) as
Array<IntArray>
```

构造多维数组时，当前无法提供初始化程序。

4.3.6. 方法

您可以使用典型的 Java 编程语法来调用方法。 您还可以在文字上调用方法。还支持变量参数。 下面的示例演示如何调用方法：

Java

```
// 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);
```

Kotlin

```
// string literal, evaluates to "bc"
val bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String::class.java)

// evaluates to true
val isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(
    societyContext, Boolean::class.java)
```

4.3.7. 操作符

Spring Expression Language 支持以下几种运算符：

- 关系运算符
- 逻辑运算符
- 数学运算符
- 赋值运算符

关系运算符

使用标准运算符表示法支持关系运算符（等于，不等于，小于，小于或等于，大于和大于或等于）。以下列表显示了一些运算符示例：

Java

```
// 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);
```

Kotlin

```
// evaluates to true  
val trueValue = parser.parseExpression("2 == 2").getValue(Boolean::class.java)  
  
// evaluates to false  
val falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean::class.java)  
  
// evaluates to true  
val trueValue = parser.parseExpression("'black' <  
'block'").getValue(Boolean::class.java)
```

对 `null` 的大于和小于比较遵循一个简单的规则：`null` 被视为无（不是零）。结果，任何其他值始终大于 `null` (`X>null` 始终为 `true`)，并且其他任何值都不小于零 (`X<null` 始终为 `false`)。



如果您更喜欢使用数字比较，请避免使用基于数字的空比较，而建议使用零进行比较（例如 `X>0` 或 `X<0`）。

除了标准的关系运算符外，SpEL 还支持 `instanceof` 和基于正则表达式的匹配运算符。以下列表显示了两个示例：

Java

```
// 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);
```

Kotlin

```
// evaluates to false
val falseValue = parser.parseExpression(
    "'xyz' instanceof T(Integer)").getValue(Boolean::class.java)

// evaluates to true
val trueValue = parser.parseExpression(
    "'5.00' matches '^-?\\d+(\\.\\d{2})?$',").getValue(Boolean::class.java)

//evaluates to false
val falseValue = parser.parseExpression(
    "'5.0067' matches '^-?\\d+(\\.\\d{2})?$',").getValue(Boolean::class.java)
```



请注意基本类型，他们会被立刻转换为对应的包装类型。所以 1 instanceof T(int) 值为 false，1 instanceof T(Integer) 值为 true。每个符号运算符也可以指定为纯字母等效项。这样可以避免使用的符号对于嵌入表达式的文档类型具有特殊含义的问题（例如在 XML 文档中）。等效的文字是：

- **lt (<)**
- **gt (>)**
- **le (<=)**
- **ge (>=)**
- **eq (==)**
- **ne (!=)**
- **div (/)**
- **mod (%)**
- **no (!).**

所有的文本运算符都不区分大小写。

逻辑运算符

SpEL 支持如下逻辑操作符：

- `and (&&)`
- `or (||)`
- `not (!)`

下面的示例显示如何使用逻辑运算符：

Java

```
// -- 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);
```

Kotlin

```
// -- AND --
// evaluates to false
val falseValue = parser.parseExpression("true and
false").getValue(Boolean::class.java)

// evaluates to true
val expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')"
val trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)

// -- OR --
// evaluates to true
val trueValue = parser.parseExpression("true or false").getValue(Boolean::class.java)

// evaluates to true
val expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')"
val trueValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)

// -- NOT --
// evaluates to false
val falseValue = parser.parseExpression("!true").getValue(Boolean::class.java)

// -- AND and NOT --
val expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')"
val falseValue = parser.parseExpression(expression).getValue(societyContext,
Boolean::class.java)
```

数学运算符

您可以在数字和字符串上使用加法运算符。 只能对数字使用减，乘和除运算符。 您还可以使用模数（%）和指数幂（^）运算符。 强制执行标准运算符优先级。 以下示例显示了正在使用的数学运算符：

Java

```
// 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
```

Kotlin

```
// Addition
val two = parser.parseExpression("1 + 1").getValue(Int::class.java) // 2

val testString = parser.parseExpression(
    "'test' + ' ' + 'string'").getValue(String::class.java) // 'test string'

// Subtraction
val four = parser.parseExpression("1 - -3").getValue(Int::class.java) // 4

val d = parser.parseExpression("1000.00 - 1e4").getValue(Double::class.java) // -9000

// Multiplication
val six = parser.parseExpression("-2 * -3").getValue(Int::class.java) // 6

val twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Double::class.java)
// 24.0

// Division
val minusTwo = parser.parseExpression("6 / -3").getValue(Int::class.java) // -2

val one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double::class.java) // 1.0

// Modulus
val three = parser.parseExpression("7 % 4").getValue(Int::class.java) // 3

val one = parser.parseExpression("8 / 5 % 2").getValue(Int::class.java) // 1

// Operator precedence
val minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Int::class.java) //
-21
```

赋值运算符

要设置属性，请使用赋值运算符 (=)。这通常在对 setValue 的调用内完成，但也可以在对 getValue 的调用内完成。下面的列表显示了使用赋值运算符的两种方法：

Java

```
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);
```

Kotlin

```
val inventor = Inventor()
val context = SimpleEvaluationContext.forReadWriteDataBinding().build()

parser.parseExpression("Name").setValue(context, inventor, "Aleksandar Seovic")

// alternatively
val aleks = parser.parseExpression(
    "Name = 'Aleksandar Seovic'").getValue(context, inventor, String::class.java)
```

4.3.8. 类型

您可以使用特殊的 T 运算符来指定 `java.lang.Class`（类型）的实例。静态方法也可以通过使用此运算符来调用。`StandardEvaluationContext` 使用 `TypeLocator` 查找类型，而 `StandardTypeLocator`（可以替换）是在了解 `java.lang` 包的情况下构建的。这意味着对 `Java.lang` 中的类型的 `T()` 引用不需要完全限定，但是所有其他类型引用都必须是完全限定的。下面的示例演示如何使用 T 运算符：

Java

```
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);
```

Kotlin

```
val dateClass =
parser.parseExpression("T(java.util.Date)").getValue(Class::class.java)

val stringClass = parser.parseExpression("T(String)").getValue(Class::class.java)

val trueValue = parser.parseExpression(
    "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")
    .getValue(Boolean::class.java)
```

4.3.9. 构造器

您可以使用 `new` 运算符来调用构造函数。除基本类型（`int`, `float` 等）和 `String` 以外的所有其他类都应使用完全限定的类名。下面的示例演示如何使用 `new` 运算符调用构造函数：

Java

```
Inventor einstein = p.parseExpression(  
    "new org.springframework.spel.inventor.Inventor('Albert Einstein', 'German')")  
    .getValue(Inventor.class);  
  
//create new inventor instance within add method of List  
p.parseExpression(  
    "Members.add(new org.springframework.spel.inventor.Inventor(  
        'Albert Einstein', 'German'))").getValue(societyContext);
```

Kotlin

```
val einstein = p.parseExpression(  
    "new org.springframework.spel.inventor.Inventor('Albert Einstein', 'German')")  
    .getValue(Inventor::class.java)  
  
//create new inventor instance within add method of List  
p.parseExpression(  
    "Members.add(new org.springframework.spel.inventor.Inventor('Albert Einstein',  
        'German'))")  
    .getValue(societyContext)
```

4. 3. 10. 变量

您可以使用#variableName 语法在表达式中引用变量。通过在 EvaluationContext 实现上使用 setVariable 方法设置变量。

有效的变量名称必须由以下一个或多个受支持的字符组成。

- 字母字符: A 到 Z 和 a 到 z
- 数字: 0 到 9
- 下划线: _
- 美元符: \$

以下示例显示了如何使用变量。

Java

```
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"
```

Kotlin

```
val tesla = Inventor("Nikola Tesla", "Serbian")

val context = SimpleEvaluationContext.forReadWriteDataBinding().build()
context.setVariable("newName", "Mike Tesla")

parser.parseExpression("Name = #newName").getValue(context, tesla)
println(tesla.name) // "Mike Tesla"
```

#this 和#root 变量

#this 变量始终是定义并且引用当前的评估对象（针对解决不合格的引用）。#root 变量始终定义并引用根上下文对象。尽管#this 可能随表达式的组成部分的求值而变化，但 #root 始终引用根。以下示例说明如何使用#this 和#root 变量：

Java

```
// 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);
```

Kotlin

```
// create an array of integers
val primes = ArrayList<Int>()
primes.addAll(listOf(2, 3, 5, 7, 11, 13, 17))

// create parser and set variable 'primes' as the array of integers
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataAccess()
context.setVariable("primes", primes)

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
val primesGreaterThanTen = parser.parseExpression(
    "#primes.?[#this>10]").getValue(context) as List<Int>
```

4.3.11. 函数

您可以通过注册可以在表达式字符串中调用的用户定义函数来扩展 SpEL。该函数通过 `EvaluationContext` 注册。下面的示例演示如何注册用户定义的函数：

Java

```
Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();
context.setVariable("myFunction", method);
```

Kotlin

```
val method: Method = ...

val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()
context.setVariable("myFunction", method)
```

例如，考虑以下用于反转字符串的实用程序方法：

Java

```
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();
    }
}
```

Kotlin

```
fun reverseString(input: String): String {
    val backwards = StringBuilder(input.length)
    for (i in 0 until input.length) {
        backwards.append(input[input.length - 1 - i])
    }
    return backwards.toString()
}
```

然后，您可以注册并使用前面的方法，如以下示例所示：

Java

```
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);
```

Kotlin

```
val parser = SpelExpressionParser()

val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()
context.setVariable("reverseString", ::reverseString::javaMethod)

val helloWorldReversed = parser.parseExpression(
    "#reverseString('hello')").getValue(context, String::class.java)
```

4.3.12. Bean 引用

如果求值上下文已使用 bean 解析器配置，则可以使用 @ 符号从表达式中查找 bean。以下示例显示了如何执行此操作：

Java

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context, "something") on MyBeanResolver during
// evaluation
Object bean = parser.parseExpression("@something").getValue(context);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = StandardEvaluationContext()
context.setBeanResolver(MyBeanResolver())

// This will end up calling resolve(context, "something") on MyBeanResolver during
// evaluation
val bean = parser.parseExpression("@something").getValue(context)
```

要访问工厂 bean 本身，您应该在 bean 名称前加上 & 符号。以下示例显示了如何执行此操作：

Java

```
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);
```

Kotlin

```
val parser = SpelExpressionParser()
val context = StandardEvaluationContext()
context.setBeanResolver(MyBeanResolver())

// This will end up calling resolve(context,"&foo") on MyBeanResolver during
// evaluation
val bean = parser.parseExpression("&foo").getValue(context)
```

4.3.13. 三元操作符

您可以使用三元运算符在表达式内部执行 if-then-else 条件逻辑。以下列表显示了一个小小的示例：

Java

```
String falseString = parser.parseExpression(
    "false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

Kotlin

```
val falseString = parser.parseExpression(
    "false ? 'trueExp' : 'falseExp'").getValue(String::class.java)
```

在这种情况下，布尔值 `false` 导致返回字符串值 '`falseExp`'。一个更现实的示例如下：

Java

```
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"
```

Kotlin

```
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'"

val queryResultString = parser.parseExpression(expression)
    .getValue(societyContext, String::class.java)
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

有关三元运算符的更短语法，请参阅下一部分有关 Elvis 运算符的内容

4.3.14. Elvis 操作符

Elvis 运算符是三元运算符语法的简化，并且在 Groovy 语言中使用。使用三元运算符语法，通常必须将变量重复两次，如以下示例所示：

```
String name = "Elvis Presley";
String displayName = (name != null ? name : "Unknown");
```

取而代之的是，您可以使用 Elvis 运算符（其命名类似于 Elvis 的发型）。以下示例显示了如何使用 Elvis 运算符：

Java

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(new Inventor(),
String.class);
System.out.println(name); // 'Unknown'
```

Kotlin

```
val parser = SpelExpressionParser()

val name = parser.parseExpression("name?:'Unknown'").getValue(Inventor(),
String::class.java)
println(name) // 'Unknown'
```

下面列表秀了一个稍微复杂点的例子：

Java

```
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
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

val tesla = Inventor("Nikola Tesla", "Serbian")
var name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla,
String::class.java)
println(name) // Nikola Tesla

tesla.setName(null)
name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla,
String::class.java)
println(name) // Elvis Presley
```

您可以使用 `Elvis` 运算符在表达式中应用默认值。以下示例显示了如何在 `@Value` 表达式中使用 `Elvis` 运算符：



```
@Value("#{systemProperties['pop3.port'] ?: 25}")
```

如果定义，将注入系统属性 `pop3.port`，否则将注入 25。

4.3.15. 安全导航操作符

安全导航运算符用于避免 `NullPointerException`，它来自 Groovy(www.groovy-lang.org)语言。通常，当您引用一个对象时，可能需要在访问该对象的方法或属性之前验证其是否为 `null`。为了避免这种情况，安全导航运算符返回 `null` 而不是引发异常。下面的示例演示如何使用安全导航操作符：

Java

```
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!!!
```

Kotlin

```
val parser = SpelExpressionParser()
val context = SimpleEvaluationContext.forReadOnlyDataBinding().build()

val tesla = Inventor("Nikola Tesla", "Serbian")
tesla.setPlaceOfBirth(PlaceOfBirth("Smiljan"))

var city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla,
String::class.java)
println(city) // Smiljan

tesla.setPlaceOfBirth(null)
city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla,
String::class.java)
println(city) // null - does not throw NullPointerException!!!
```

4.3.16. 集合选择器

选择器是一种强大的表达语言功能，可让您通过从源集合中选择条目来将其转换为另一个集合。

选择使用`.?[selectionExpression]`的语法。它过滤集合并返回一个包含原始元素子集的新集合。例如，通过选择，我们可以轻松地获得 `Serbian inventor` 的列表，如以下示例所示：

Java

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(  
    "Members.? [Nationality == 'Serbian']").getValue(societyContext);
```

Kotlin

```
val list = parser.parseExpression(  
    "Members.? [Nationality == 'Serbian']").getValue(societyContext) as  
List<Inventor>
```

在 `list` 和 `Map` 上都可以选择。对于列表，将针对每个单独的列表元素计算选择标准。针对 `Map`，针对每个 `Map` 条目（`Java` 类型 `Map.Entry` 的对象）计算选择标准。每个地图条目都有其键和值，可作为属性进行访问以供选择。以下表达式返回一个新 `Map`，该 `Map` 由原始 `Map` 中键值小于 27 的那些元素组成：

Java

```
Map newMap = parser.parseExpression("map.? [value<27]").getValue();
```

Kotlin

```
val newMap = parser.parseExpression("map.? [value<27]").getValue()
```

除了返回所有选定的元素外，您只能检索第一个或最后一个值。为了获得与选择匹配的第一个条目，语法为 `.^ [selectionExpression]`。为了获得与选择匹配的最后一个条目，语法为 `.^ [selectionExpression]`。

4.3.17. 集合的投射

投射使集合可以驱动子表达式的求值，结果是一个新的集合。投影的语法为 `.! [projectionExpression]`。例如，假设我们有一个发明家列表，但想要他们出生的城市的列表。实际上，我们希望为发明人列表中的每个条目求值 `"placeOfBirth.city"`。

下面的示例使用投影来做到这一点：

Java

```
// returns ['Smiljan', 'Idvor']  
List placesOfBirth = (List)parser.parseExpression("Members.! [placeOfBirth.city]");
```

Kotlin

```
// returns ['Smiljan', 'Idvor']
val placesOfBirth = parser.parseExpression("Members.! [placeOfBirth.city]") as List<*>
```

您还可以使用 Map 来驱动投射，在这种情况下，将针对 Map 中的每个条目（以 Java Map.Entry 表示）对投射表达式进行求值。跨 Map 的投射结果是一个 list，其中包含针对每个 Map 条目的投射表达式的求值。

4.3.18. 表达式模板

表达式模板允许将文字文本与一个或多个求值块混合。每个求值块均以您可以定义的前缀和后缀字符分隔。常见选择是使用 # {} 作为分隔符，如以下示例所示：

Java

```
String randomPhrase = parser.parseExpression(
    "random number is #{T(java.lang.Math).random()}",
    new TemplateParserContext().getValue(String.class));

// evaluates to "random number is 0.7038186818312008"
```

Kotlin

```
val randomPhrase = parser.parseExpression(
    "random number is #{T(java.lang.Math).random()}",
    TemplateParserContext().getValue(String::class.java))

// evaluates to "random number is 0.7038186818312008"
```

通过将文字文本“random number is ”与求值 # {} 分隔符内的表达式的结果（在本例中为调用那个 random() 方法的结果）相连接来求值字符串。parseExpression() 方法的第二个参数的类型为 ParserContext。ParserContext 接口用于影响表达式的解析方式，以支持表达式模板功能。TemplateParserContext 的定义如下：

Java

```
public class TemplateParserContext implements ParserContext {  
    public String getExpressionPrefix() {  
        return "#{";  
    }  
  
    public String getExpressionSuffix() {  
        return "}";  
    }  
  
    public boolean isTemplate() {  
        return true;  
    }  
}
```

Kotlin

```
class TemplateParserContext : ParserContext {  
  
    override fun getExpressionPrefix(): String {  
        return "#{"  
    }  
  
    override fun getExpressionSuffix(): String {  
        return "}"  
    }  
  
    override fun isTemplate(): Boolean {  
        return true  
    }  
}
```

4.4. 例子中的类使用

本节列出了本章示例中使用的类。

```
package org.springframework.samples.spel.inventor;  
  
import java.util.Date;  
import java.util.GregorianCalendar;  
  
public class Inventor {  
  
    private String name;  
    private String nationality;
```

(与下页为同一个文件)

```
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;
}
```

(与下页为同一个文件)

```
public void setInventions(String[] inventions) {
    this.inventions = inventions;
}

public String[] getInventions() {
    return inventions;
}
```

Inventor.kt

```
class Inventor(
    var name: String,
    var nationality: String,
    var inventions: Array<String>? = null,
    var birthdate: Date = GregorianCalendar().time,
    var placeOfBirth: PlaceOfBirth? = null)
```

PlaceOfBirth.java

```
package org.springframework.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;
    }
}
```

PlaceOfBirth.kt

```
class PlaceOfBirth(var city: String, var country: String? = null) {
```

Society.java

```
package org.springframework.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;
    }
}
```

Society.kt

```
package org.springframework.samples.spel.inventor

import java.util.*

class Society {

    val Advisors = "advisors"
    val President = "president"

    var name: String? = null

    val members = ArrayList<Inventor>()
    val officers = mapOf<Any, Any>()

    fun isMember(name: String): Boolean {
        for (inventor in members) {
            if (inventor.name == name) {
                return true
            }
        }
        return false
    }
}
```

5. Spring 的面向切面编程 (AOP)

面向方面的编程 (AOP) 通过提供另一种思考程序结构的方式来补充面向对象的编程 (OOP)。OOP 中模块化的关键单元是类，而在 AOP 中模块单元是切面。切面使关注点（例如事务管理）的模块化可以跨越多种类型和对象。（这种关注在 AOP 文献中通常被称为“跨领域”关注。）

Spring 的关键组件之一是 AOP 框架。尽管 Spring IoC 容器不依赖于 AOP（这意味着您不需要使用 AOP），但 AOP 是对 Spring IoC 的补充，以提供功能强大的中间件解决方案。

Spring AOP 和 AspectJ 切入点

Spring 通过使用[基于模式的方法\(5.5\)](#)或[@AspectJ 注解风格\(5.4\)](#)，提供了编写自定义切面的简单而强大的方法。这两种样式都提供了完全类型化的通知，并使用了 AspectJ 切入点语言，同时仍使用 Spring AOP 进行织入。

本章讨论基于架构和基于@AspectJ 的 AOP 支持。[下一章](#)将讨论较低级别的 AOP 支持。

AOP 在 Spring Framework 中用于：

- 提供声明式企业服务。此类服务中最重要的是[声明式事务管理](#)。
- 让用户实现自定义方面，以 AOP 补充其对 OOP 的使用。



如果您只对通用声明性服务或其他预打包的声明性中间件服务（例如池）感兴趣，则无需直接使用 Spring AOP，并且可以跳过本章的大部分内容。

5.1. AOP 概念

让我们首先定义一些重要的 AOP 概念和术语。这些术语不是特定于 Spring 的。不幸的是，AOP 术语不太直观。但是，如果 Spring 使用自己的术语，那将更加令人困惑。

- **切面：**涉及多个类别的关注点的模块化。事务管理是企业 Java 应用程序中横切的一个很好的例子。在 Spring AOP 中，方面是通过使用常规类（[基于模式的方法](#)）或使用[@Aspect](#) 注解（[@AspectJ 样式](#)）注解的常规类来实现的。
- **连接点：**在程序执行过程中的一点，例如方法的执行或异常的处理。在 Spring AOP 中，连接点始终代表方法的执行。

- **通知：**方面在特定的连接点处采取的操作。不同类型的建议包括“环绕”，“前”和“后”建议。（通知类型将在后面讨论。）包括 Spring 在内的许多 AOP 框架都将通知建模为拦截器，并在连接点周围维护一系列拦截器。
- **切入点：**与连接点匹配的谓词。通知与切入点表达式关联，并在与该切入点匹配的任何连接点处运行（例如，执行具有特定名称的方法）。切入点表达式匹配的连接点的概念是 AOP 的核心，并且 Spring 默认使用 AspectJ 切入点表达语言。
- **引介：**代表类型声明其他方法或字段。Spring AOP 允许您向任何通知的对象引入新的接口（和相应的实现）。例如，您可以使用简介使 Bean 实现 `IsModified` 接口，以简化缓存。（在 AspectJ 社区中，引介被称为类型间声明。）
- **目标对象：**一个或多个方面通知的对象。也称为“通知对象”。由于 Spring AOP 是使用运行时代理实现的，因此该对象始终是代理对象。
- **AOP 代理：**由 AOP 框架创建的一个对象，用于实现切面协定（通知方法执行等）。在 Spring Framework 中，AOP 代理是 JDK 动态代理或 CGLIB 代理。
- **织入：**将切面与其他应用程序类型或对象链接以创建通知的对象。这可以在编译时（例如，使用 AspectJ 编译器），加载时或在运行时完成。像其他纯 Java AOP 框架一样，Spring AOP 在运行时执行织入。

Spring AOP 包含下面几类通知：

前置通知：在连接点之前运行但无法阻止执行流前进到连接点的通知（除非引发异常）。

后置返回通知：连接点正常完成后要运行的通知（如果某个方法返回而没有引发异常）。

后置抛出通知：如果方法因引发异常而退出，则要运行的通知。

后置(最终)通知：无论连接点退出的方式如何（正常或异常返回）都应运行的通知。

环绕通知：围绕连接点的通知，例如方法调用。这是最有力的通知。环绕通知可以在方法调用之前和之后执行自定义行为。它还负责选择是返回连接点还是通过返回自己的返回值或引发异常来快捷通知的方法执行。

环绕通知是最通用的通知。由于 Spring AOP 与 AspectJ 一样，提供了各种通知类型，因此我们建议您使用功能最弱的通知类型，以实现所需的行为。例如，如果您只需要使用方法的返回值更新缓存，则最好使用返回通知而不是环绕通知，尽管环绕通知可以完成相同的事情。使用最具体的通知可以提供更简单的编程模型，并减少出错的可能性。例如，您不需要在用于环绕的连接点上调用 `proceed()` 方法，因此，您不会失败。所有通知参数都是静态类型的，因此您可以使用适当类型（例如，方法执行的返回值的类型）而不是对象数组的

通知参数。

切入点匹配的连接点的概念是 AOP 的关键，它与仅提供拦截功能的旧技术有所不同。切入点使通知的目标独立于面向对象的层次结构。例如，您可以将提供声明性事务管理的环绕通知应用于跨越多个对象（例如服务层中的所有业务操作）的一组方法。

5.2. Spring AOP 能力和目标

Spring AOP 是用纯 Java 实现的。不需要特殊的编译过程。Spring AOP 不需要控制类加载器的层次结构，因此适合在 Servlet 容器或应用程序服务器中使用。

Spring AOP 当前仅支持方法执行连接点（建议在 Spring Bean 上执行方法）。尽管可以在不破坏核心 Spring AOP API 的情况下添加对字段拦截的支持，但并未实现字段拦截。如果需要通知字段访问和更新连接点，请考虑使用诸如 AspectJ 之类的语言。

Spring AOP 的 AOP 方法不同于大多数其他 AOP 框架。目的不是提供最完整的 AOP 实现（尽管 Spring AOP 相当强大）。相反，其目的是在 AOP 实现和 Spring IoC 之间提供紧密的集成，以帮助解决企业应用程序中的常见问题。

因此，例如，通常将 Spring Framework 的 AOP 功能与 Spring IoC 容器结合使用。通过使用常规 bean 定义语法来配置方面（尽管这允许强大的“自动代理”功能）。这是与其他 AOP 实现的关键区别。使用 Spring AOP 无法轻松或高效地完成某些事情，例如通知非常细粒度的对象（通常是字段对象）。在这种情况下，AspectJ 是最佳选择。但是，我们的经验是，Spring AOP 为企业 Java 应用程序中适合 AOP 的大多数问题提供了出色的解决方案。

Spring AOP 从未努力与 AspectJ 竞争以提供全面的 AOP 解决方案。我们认为，基于代理的框架（如 Spring AOP）和成熟的框架（如 AspectJ）都是有价值的，它们是互补的，而不是竞争。Spring 将 AspectsJ 无缝集成了 Spring AOP 和 IoC，以在基于 Spring 的一致应用程序架构中支持 AOP 的所有使用。这种集成不会影响 Spring AOP API 或 AOP Alliance API。Spring AOP 仍向后兼容。有关 Spring AOP API 的讨论，请参见[下一章](#)。



Spring 框架的中心宗旨之一是非侵入性。这是一个想法，不应强迫您将特定于框架的类和接口引入业务或域模型。但是，在某些地方，Spring Framework 确实为您提供了将 Spring Framework 特定的依赖项引入代码库的选项。提供此类选项的理由是，在某些情况下，以这种方式阅读或编码某些特定功能可能会变得更加容易。但是，Spring 框架（几乎）总是为您提供选择：您可以自由地就哪个选项最适合您的特定用例或场景做出明智的决定。

与本章相关的一种选择是选择哪种 AOP 框架（以及哪种 AOP 样式）。您可以选择 AspectJ, Spring AOP 或二者皆取。您还可以选择@AspectJ 注解样式方法或 Spring XML 配置样式方法。本章选择首先介绍@AspectJ 风格的方法这一事实不应被视为表明 Spring 团队比 Spring XML 配置风格更喜欢@AspectJ 注解风格的方法。有关每种样式的“刨根问底”的更完整讨论，请参见[选择要使用的 AOP 声明样式\(5.6\)](#)。

5.3. AOP 代理

Spring AOP 默认将标准 JDK 动态代理用于 AOP 代理。这使得可以代理任何接口（或一组接口）。Spring AOP 也可以使用 CGLIB 代理。这对于代理类而不是接口是必需的。默认情况下，如果业务对象未实现接口，则使用 CGLIB。由于对接口而不是对类进行编程是一种好习惯，因此业务类通常实现一个或多个业务接口。在那些需要通知在接口上未声明的方法或需要将代理对象作为具体类型传递给方法的情况下（在极少数情况下），可以强制使用 CGLIB。

掌握 Spring AOP 是基于代理的这一事实非常重要。有关完全了解此实现细节实际含义的详细信息，请参阅[了解 AOP 代理\(5.8.1\)](#)。

5.4. @AspectJ 支持

@AspectJ 是一种将通知声明为带有注解的常规 Java 类的样式。@AspectJ 样式是[AspectJ 项目](#)在 AspectJ 5 版本中引入的。Spring 使用 AspectJ 提供的用于切入点解析和匹配的库来解释与 AspectJ 5 相同的注解。但是，AOP 运行时仍然是纯 Spring AOP，并且不依赖于 AspectJ 编译器或编织器。



使用 AspectJ 编译器和 weaver 可以使用完整的 AspectJ 语言，有关[在 Spring App 中使用 AspectJ\(5.10\)](#)进行了讨论

5.4.1. 启动@AspectJ 支持

要在 Spring 配置中使用@AspectJ 切面，您需要启用 Spring 支持以基于@AspectJ 方面配置 Spring AOP，并根据这些切面是否对它们进行自动代理通知。通过自动代理，我们的意思是，如果 Spring 确定一个 bean 被一个或多个切面通知，它将自动为该 bean 生成一个代理以拦截方法调用并确保按需运行通知。

可以使用 XML 或 Java 样式的配置来启用@AspectJ 支持。无论哪种情况，您都需要确保 AspectJ 的 `Aspectjweaver.jar` 库位于应用程序的类路径（版本 1.8 或更高版本）上。该库在 AspectJ 发行版的 lib 目录中或从 Maven Central 存储库中可用。

使用 Java 配置启动@AspectJ 支持

要通过 Java `@Configuration` 启用 @AspectJ 支持，请添加 `@EnableAspectJAutoProxy` 注解，如以下示例所示：

Java

```
@Configuration  
@EnableAspectJAutoProxy  
public class AppConfig {  
}
```

Kotlin

```
@Configuration  
@EnableAspectJAutoProxy  
class AppConfig
```

使用 XML 配置启动@AspectJ 支持

要通过基于 XML 的配置启用@AspectJ 支持，请使用 `aop: aspectj-autoproxy` 元素，如以下示例所示：

```
<aop:aspectj-autoproxy/>
```

假定您使用 [基于 XML Schema 的配置\(9.1\)](#) 中所述的架构支持。有关如何在 `aop` 名称空间中导入标签的信息，请[参见 AOP 模式\(9.1.2\)](#)。

5.4.2. 声明一个 Aspect

启用@AspectJ 支持后，Spring 会自动检测在应用程序上下文中使用@AspectJ 切面（具有@Aspect 注解）的类定义的任何 bean，并将其用于配置 Spring AOP。接下来的两个示

例显示了一个不太有用的切面所需的最小定义。这两个示例中的第一个示例显示了应用程序上下文中的常规 bean 定义，该定义指向具有@Aspect 注解的 bean 类：

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
    <!-- configure properties of the aspect here -->
</bean>
```

这两个示例中的第二个示例显示了 `NotVeryUsefulAspect` 类定义，该类定义使用 `org.aspectj.lang.annotation.Aspect` 注解进行标识；

Java

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Kotlin

```
package org.xyz

import org.aspectj.lang.annotation.Aspect;

@Aspect
class NotVeryUsefulAspect
```

切面（使用@Aspect 注解的类）可以具有方法和字段，与任何其他类相同。它们还可以包含切入点、通知和引介（类型间）声明。

通过组件扫描自动检测切面



您可以将切面类注册为 Spring XML 配置中的常规 bean，也可以通过类路径扫描来自动检测它们-与其他任何 Spring 管理的 bean 一样。但是，请注意，@ Aspect 注解不足以在类路径中进行自动检测。为此，您需要添加一个单独的@Component 注解（或者，按照 Spring 的组件扫描器的规则，有条件的自定义构造注解）。



通过其他切面向切面提供通知？在 Spring AOP 中，切面本身不能成为其他切面的通知目标。类上的@Aspect 注解将其标记为一个切面，因此将其从自动代理中排除。

5.4.3. 声明一个切入点

切入点确定了感兴趣的连接点，从而使我们能够控制运行建议的时间。Spring AOP 仅支持 Spring Bean 的方法执行连接点，因此您可以将切入点视为与 Spring Bean 上的方法执行相匹配。切入点声明有两个部分：一个包含名称和任何参数的签名，以及一个切入点表达式，该切入点表达式精确确定我们感兴趣的方法执行。在 AOP 的@AspectJ 注解样式中，常规方法定义提供了切入点签名，并且通过使用@Pointcut 注解标识了切入点表达式（用作切入点签名的方法必须是 void 返回类型）。

一个示例可能有助于使切入点签名和切入点表达式之间的区别变得清晰。下面的示例定义一个名为 anyOldTransfer 的切入点，该切入点与任何名为 transfer 的方法的执行相匹配：

Java

```
@Pointcut("execution(* transfer(..))") // the pointcut expression
private void anyOldTransfer() {} // the pointcut signature
```

Kotlin

```
@Pointcut("execution(* transfer(..))") // the pointcut expression
private fun anyOldTransfer() {} // the pointcut signature
```

构成@Pointcut 注解的值的切入点表达式是一个常规的 AspectJ 5 切入点表达式。有关 AspectJ 的切入点语言的完整讨论，请参阅相关书籍。

支持切入点的选择器

Spring AOP 支持以下在切入点表达式中使用的 AspectJ 切入点指示符 (PCD)：

- **execution**: 用于匹配方法执行的连接点。这是使用 Spring AOP 时要使用的主要切入点选择器。
- **within**: 将匹配限制为某些类型内的连接点（使用 Spring AOP 时，在匹配类型内声明的方法的执行）。
- **this**: 限制匹配到连接点（使用 Spring AOP 时方法的执行）的匹配，其中 bean 引用（Spring AOP 代理）是给定类型的实例。
- **target**: 限制到连接点（使用 Spring AOP 时方法的执行）的匹配，其中目标对象（代理的应用程序对象）是给定类型的实例。
- **args**: 限制匹配到连接点（使用 Spring AOP 时方法的执行）的匹配，其中参数是给定类型的实例。
- **@target**: 限制匹配到连接点（使用 Spring AOP 时方法的执行）在执行对象有一个给定类型的注解。

- `@args`: 限制匹配的连接点（使用 Spring AOP 时方法的执行），其中传递的实际参数的运行时类型具有给定类型的注解。
- `@within`: 限制匹配到具有给定注解的类型内的连接点（使用 Spring AOP 时，使用给定注释的类型中声明的方法的执行）。
- `@annotation`: 限制匹配连接点的主题（在 Spring AOP 中运行的方法）具有给定注解的连接点。

其他切入点类型

完整的 AspectJ 切入点语言支持 Spring 不支持的其他切入点指示符：`call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, 和 @withincode`。在 Spring AOP 解释的切入点表达式中使用这些切入点选择器会导致抛出 `IllegalArgumentException`。

Spring AOP 支持的切入点选择器集合可能会在将来的版本中扩展，以支持更多的 AspectJ 切入点选择器。

因为 Spring AOP 将限制匹配为仅方法执行连接点，所以前面对切入点选择器的讨论所提供的定义比 AspectJ 编程指南中的定义要窄。此外，AspectJ 本身具有基于类型的语义，并且在执行连接点处，`this` 和 `target` 都引用同一对象：执行该方法的对象。Spring AOP 是一个基于代理的系统，区分代理对象本身（绑定到 `this`）和代理后面的目标对象（绑定到 `target`）。

由于 Spring 的 AOP 框架基于代理的性质，因此根据定义，不会拦截目标对象内的调用。对于 JDK 代理，只能拦截代理上的公共接口方法调用。使用 CGLIB 时，将拦截代理上的 `public` 方法和 `protected` 方法（甚至在必要时对程序包可见的方法）。但是，通常应通过公共签名设计通过代理进行的常见交互。



请注意，切入点定义通常与任何拦截方法匹配。如果严格地将切入点设置为仅公开使用，即使在 CGLIB 代理方案中通过代理存在潜在的非公开交互作用，也需要相应地进行定义。

如果您的拦截需要在目标类中包含方法调用甚至构造函数，请考虑使用 Spring 驱动的原生 AspectJ 织入，而不是 Spring 的基于代理的 AOP 框架。这构成了具有不同特征的 AOP 使用模式，因此请确保在做出决定之前先熟悉织入。

Spring AOP 还支持其他名为 bean 的 PCD。使用 PCD，可以将连接点的匹配限制为特定的命名 Spring Bean 或一组命名 Spring Bean（使用通配符时）。Bean PCD 具有以下

形式：

Java

```
bean(idOrNameOfBean)
```

Kotlin

```
bean(idOrNameOfBean)
```

`idOrNameOfBean` 标识可以是任何 Spring bean 的名称。提供了使用*字符的有限通配符支持，因此，如果为 Spring bean 建立了一些命名约定，则可以编写 bean PCD 表达式来选择它们。与其他切入点指示符一样，bean PCD 可以与`&&`（与），`||`（或），`!`（非）运算符共用。

Bean PCD 仅在 Spring AOP 中受支持，而在原生 AspectJ 织入中不受支持。它是 AspectJ 定义的标准 PCD 的特定于 Spring 的扩展，因此不适用于`@Aspect` 模型中声明的方面。



Bean PCD 在实例级别（基于 Spring bean 名称概念构建）上运行，而不是仅在类型级别（基于编织的 AOP 受其限制）上运行。基于实例的切入点选择器是 Spring 基于代理的 AOP 框架的特殊功能，它与 Spring bean 工厂紧密集成，因此可以自然而直接地通过名称识别特定 bean。

结合切入点表达式

您可以使用`&&`，`||`组合切入点表达式和`!`，你也可以按名称引用切入点表达式。以下示例显示了三个切入点表达式：

Java

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {} ①  
  
@Pointcut("within(com.xyz.myapp.trading..*)")  
private void inTrading() {} ②  
  
@Pointcut("anyPublicOperation() && inTrading()")  
private void tradingOperation() {} ③
```

① `anyPublicOperation` 匹配表示任何公共方法执行的一个方法执行连接点。

② `inTrading` 匹配一个在 `trading` 模块的方法执行。

③ `tradingOperation` 匹配表示任何在 `trading` 模块的方法执行的公共方法的一个方法执行连接点。

Kotlin

```
@Pointcut("execution(public * *(..))")  
private fun anyPublicOperation() {} ①  
  
@Pointcut("within(com.xyz.myapp.trading..*)")  
private fun inTrading() {} ②  
  
@Pointcut("anyPublicOperation() && inTrading()")  
private fun tradingOperation() {} ③
```

- ① `anyPublicOperation` 匹配表示任何公共方法执行的一个方法执行连接点。
- ② `inTrading` 匹配一个在 `trading` 模块的方法执行。
- ③ `tradingOperation` 匹配表示任何在 `trading` 模块的方法执行的公共方法的一个方法执行连接点。

最佳实践是从较小的命名组件中构建更复杂的切入点表达式，如先前所示。按名称引用切入点时，将应用常规的 Java 可见性规则(可以看到相同类型的 `private` 切入点，层次结构中 `protected` 的切入点，任何位置的 `public` 切入点，等等)。可见性不影响切入点匹配。

分享公共切入点定义

在使用企业应用程序时，开发人员通常希望从多个方面引用应用程序的模块和特定的操作集。我们建议为此定义一个 `CommonPointcuts` 切面，以捕获常见的切入点表达式。这样的方面通常类似于以下示例：

Java

```
package com.xyz.myapp;  
  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class CommonPointcuts {  
  
    /**  
     * A join point is in the web layer if the method is defined  
     * in a type in the com.xyz.myapp.web package or any sub-package  
     * under that.  
     */  
    @Pointcut("within(com.xyz.myapp.web..*)")  
    public void inWebLayer() {}  
  
    /**  
     * A join point is in the service layer if the method is defined  
     * in a type in the com.xyz.myapp.service package or any sub-package
```

(与下页同文件)

```

 * under that.
 */
@Pointcut("within(com.xyz.myapp.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.myapp.dao package or any sub-package
 * under that.
 */
@Pointcut("within(com.xyz.myapp.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.myapp.abc.service and com.xyz.myapp.def.service) then
 * the pointcut expression "execution(* com.xyz.myapp..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.myapp..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.myapp.dao.*(..))")
public void dataAccessOperation() {}

}

```

Kotlin

```

package com.xyz.myapp

import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Pointcut

@Aspect
class CommonPointcuts {

    /**

```

(与下页同文件)

```

* A join point is in the web layer if the method is defined
* in a type in the com.xyz.myapp.web package or any sub-package
* under that.
*/
@Pointcut("within(com.xyz.myapp.web..*)")
fun inWebLayer() {
}

/**
* A join point is in the service layer if the method is defined
* in a type in the com.xyz.myapp.service package or any sub-package
* under that.
*/
@Pointcut("within(com.xyz.myapp.service..*)")
fun inServiceLayer() {
}

/**
* A join point is in the data access layer if the method is defined
* in a type in the com.xyz.myapp.dao package or any sub-package
* under that.
*/
@Pointcut("within(com.xyz.myapp.dao..*)")
fun 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.myapp.abc.service and com.xyz.myapp.def.service) then
* the pointcut expression "execution(* com.xyz.myapp..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.myapp..service.*.*(..))")
fun 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.myapp.dao.*.*(..))")
fun dataAccessOperation() {
}

```

```
    }  
}
```

您可以在需要切入点表达式的任何地方引用此类定义的切入点。例如，要使服务层具有事务性，您可以编写以下内容：

```
<aop:config>  
    <aop:advisor  
        pointcut="com.xyz.myapp.CommonPointcuts.businessService()"  
        advice-ref="tx-advice"/>  
</aop:config>  
  
<tx:advice id="tx-advice">  
    <tx:attributes>  
        <tx:method name="*" propagation="REQUIRED"/>  
    </tx:attributes>  
</tx:advice>
```

在[基于模式的 AOP 支持](#) (5.5) 中讨论了<aop:config>和<aop:advisor>元素。事务管理中讨论了事务元素。

示例

Spring AOP 用户可能最常使用执行切入点指示符。 执行表达式的格式如下：

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-  
pattern(param-pattern)  
throws-pattern?)
```

除了返回类型模式（前面的代码片段中的 ret-type-pattern），名称模式和参数模式以外的所有部分都是可选的。返回类型模式确定该方法的返回类型必须是什么才能使连接点匹配。*最常用作返回类型模式。它匹配任何返回类型。仅当方法返回给定类型时，完全合格的类型名称才匹配。名称模式与方法名称匹配。您可以将*通配符用作名称模式的全部或一部分。如果指定声明类型模式，请在其后加上。将其加入名称模式组件。参数模式稍微复杂一些：()匹配不带参数的方法，而(..)匹配任意数量（零个或多个）的参数。(*)模式与采用任何类型的一个参数的方法匹配。(*, String)与采用两个参数的方法匹配。第一个可以是任何类型，而第二个必须是 String。有关更多信息，请查阅 AspectJ 编程指南的“语义”部分。

以下示例显示了一些常用的切入点表达式：

- 执行任何公共方法：

```
execution(public * *(..))
```

- 执行以 `set` 为开头的方法:

```
execution(* set*(..))
```

- 执行任何定义自 `AccountService` 接口的方法:

```
execution(* com.xyz.service.AccountService.*(..))
```

- 执行任何定义在 `service` 包中的方法:

```
execution(* com.xyz.service.*.*(..))
```

- 执行任何定义在 `service` 包和它一个子包中的方法:

```
execution(* com.xyz.service..*.*(..))
```

- 任何在 `service` 包里的连接点 (仅在 Spring AOP 的方法执行):

```
within(com.xyz.service.*)
```

- 任何在 `service` 包和它一个子包中的连接点 (仅在 Spring AOP 的方法执行):

```
within(com.xyz.service..*)
```

- 任何在代理实现 `AccountService` 接口的连接点(仅在 Spring AOP 的方法执行):

```
this(com.xyz.service.AccountService)
```



“`this`”通常以绑定形式使用。有关如何在通知踢中使代理对象可用的信息，请参阅“[声明通知（5.4.4）](#)”部分。

- 任何在目标对象实现 `AccountService` 接口的连接点(仅在 Spring AOP 的方法执行):

```
target(com.xyz.service.AccountService)
```



“`target`”更多以绑定形式使用。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

- 任何带有但参数且参数运行时可序列化的连接点：

```
args(java.io.Serializable)
```



“args”更多以绑定形式使用。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

请注意，此示例中给出的切入点与 `execution(* *(java.io.Serializable))` 不同。如果在运行时传递的参数为 `Serializable`，则 args 版本匹配；如果方法签名声明单个类型为 `Serializable` 的参数，则执行版本匹配。

- 任何在目标对象有一个@Transactional 注解的连接点：

```
@target(org.springframework.transaction.annotation.Transactional)
```



你也可以使用’@Target’为绑定形式。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

- 任何在目标对象声明类型上有一个@Transactional 注解的连接点：

```
@within(org.springframework.transaction.annotation.Transactional)
```



你也可以使用’@within’为绑定形式。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

- 任何执行方法有一个@Transactional 注解的连接点：

```
@annotation(org.springframework.transaction.annotation.Transactional)
```



你也可以使用’@annotation’为绑定形式。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

- 任何有单个参数并在参数 passed 时有@Classified 注解的连接点：

```
@args(com.xyz.security.Classified)
```



你也可以使用’@args’为绑定形式。有关如何在通知体中使目标对象可用的信息，请参见“[声明通知（5.4.4）](#)”部分。

- 任何在 Spring bean 上叫 tradeService 的连接点：

```
bean(tradeService)
```

- 任何在 Spring bean 上名字符合表达式*Service 的连接点：

```
bean(*Service)
```

写出好的切入点

在编译期间，AspectJ 处理切入点以优化匹配性能。检查代码并确定每个连接点是否（静态或动态）匹配给定的切入点是一个很伤的过程。（动态匹配意味着无法从静态分析中完全确定匹配，并且在代码中进行测试以确定代码运行时是否存在实际匹配）。首次遇到切入点声明时，AspectJ 将其重写为匹配过程的最佳形式。这是什么意思？基本上，切入点以 DNF（析取范式）重写，并且对切入点的组件进行排序，以便首先检查那些较简单的组件。这意味着您不必担心理解各种切入点选择器的性能，并且可以在切入点声明中以任何顺序提供它们。

但是，AspectJ 只能使用所告诉的内容。为了获得最佳的匹配性能，您应该考虑他们试图达到的目标，并在定义中尽可能缩小匹配的搜索空间。现有的指示符自然分为三类之一：同类，作用域和上下文：

- 同类选择器选择一种特殊的连接点：`execution`, `get`, `set`, `call` 和 `handler`。
- 作用域选择器选择一组感兴趣的连接点（可能是多种）：`within` 和代码内 `withincode`。
- 上下文选择器根据上下文进行匹配（并可选地绑定）：`this`, `target` 和`@annotation`.

编写正确的切入点至少应包括前两种类型（种类和作用域）。您可以包括上下文指示符以根据连接点上下文进行匹配，也可以绑定该上下文以在通知中使用。仅提供同类的选择器或仅提供上下文选择器是可行的，但是由于额外的处理和分析，可能会影响织入性能（使用的时间和内存）。范围选择器的匹配非常快，使用它们的使用意味着 AspectJ 可以非常迅速地消除不应进一步处理的连接点组。一个好的切入点应该始终包括一个切入点选择器。

5.4.4. 声明通知

通知与切入点表达式关联，并且在切入点匹配的方法执行之前，之后或环绕运行。切入点表达式可以是对命名切入点的简单引用，也可以是就地声明的切入点表达式。

前置通知

您可以使用@Before 批注在一个方面中声明先于建议：

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

    @Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Before

@Aspect
class BeforeExample {

    @Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doAccessCheck() {
        // ...
    }

}
```

如果使用就地切入点表达式，则可以将前面的示例重写为以下示例：

Java

```
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() {
        // ...
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Before

@Aspect
class BeforeExample {

    @Before("execution(* com.xyz.myapp.dao.*.*(..))")
    fun doAccessCheck() {
        // ...
    }

}
```

后置返回通知

返回通知后, 当匹配的方法执行正常返回时, 运行通知。您可以使用[@AfterReturning](#)注解进行声明:

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doAccessCheck() {
        // ...
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterReturning

@Aspect
class AfterReturningExample {

    @AfterReturning("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doAccessCheck() {
        // ...
    }
}
```



您可以在同一切面内拥有多个通知声明（以及其他成员）。在这些示例

中，我们仅显示单个通知声明，以凸显每个通知的效果。

有时，您需要在通知体中访问返回的实际值。您可以使用`@AfterReturning` 的形式绑定返回值以获取该访问权限，如以下示例所示：

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

    @AfterReturning(
        pointcut="com.xyz.myapp.CommonPointcuts.dataAccessOperation(),
        returning="retVal")
    public void doAccessCheck(Object retVal) {
        // ...
    }
}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterReturning

@Aspect
class AfterReturningExample {

    @AfterReturning(
        pointcut = "com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        returning = "retVal")
    fun doAccessCheck(retVal: Any) {
        // ...
    }

}
```

返回属性中使用的名称必须与通知方法中的参数名称相对应。当方法执行返回时，该返回值将作为相应的参数值传递到通知方法。返回子句也将匹配限制为仅返回指定类型值的方法执行（在这种情况下为 Object，它匹配任何返回值）。

请注意，后置返回通知使用时，不可能返回完全不同的引用。

后置抛出

后置抛出通知，当匹配的方法执行通过抛出异常退出时运行通知。您可以使用

`@AfterThrowing` 注解进行声明，如以下示例所示：

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doRecoveryActions() {
        // ...
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterThrowing

@Aspect
class AfterThrowingExample {

    @AfterThrowing("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doRecoveryActions() {
        // ...
    }

}
```

通常，您希望通知仅在引发给定类型的异常时才运行，并且您通常还需要访问通知体中的异常。您可以使用 `throwing` 属性来限制匹配（如果需要）（否则，请使用 `Throwable` 作为异常类型），并将抛出的异常绑定到 `advice` 参数。以下示例显示了如何执行此操作：

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
        pointcut="com.xyz.myapp.CommonPointcuts.dataAccessOperation(),
        throwing="ex")
    public void doRecoveryActions(DataAccessException ex) {
        // ...
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.AfterThrowing

@Aspect
class AfterThrowingExample {

    @AfterThrowing(
        pointcut = "com.xyz.myapp.CommonPointcuts.dataAccessOperation()",
        throwing = "ex")
    fun doRecoveryActions(ex: DataAccessException) {
        // ...
    }

}
```

throwing 属性中使用的名称必须与通知方法中的参数名称相对应。当通过抛出异常退出方法执行时，该异常将作为相应的参数值传递给通知方法。**throwing** 子句还将匹配仅限制为抛出指定类型的异常（在这种情况下为 `DataAccessException`）的方法执行。

后置(最终)通知

当匹配的方法执行退出时，后置(最终)通知运行。通过使用`@After`注解声明它。后置通知必须准备去处理所有普通黑厂返回情况。它通常用于释放资源和类似目的。以下示例显示了后置最终通知的用法：

Java

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

    @After("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    public void doReleaseLock() {
        // ...
    }

}
```

```

import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.After

@Aspect
class AfterFinallyExample {

    @After("com.xyz.myapp.CommonPointcuts.dataAccessOperation()")
    fun doReleaseLock() {
        // ...
    }

}

```

环绕通知

最后一种通知是环绕通知。 环绕通知在匹配方法的执行过程中“环绕”运行。它有机机会在该方法运行之前和之后进行工作，并确定何时，如何以及即使该方法实际上可以运行。如果需要以线程安全的方式（例如，启动和停止计时器）在方法执行之前和之后共享状态，则通常使用环绕通知。始终适用最有力的通知来符合你的需求（就是说如果前置通知能干，别使环绕通知）。

通过使用 `@Around` 注解来声明环绕通知。通知方法的第一个参数必须是 `ProceedingJoinPoint` 的属性。在通知体内部，调用 `ProceedingJoinPoint` 上的 `proceed()` 来使底层方法运行。`proceed` 方法也可以传入 `Object[]`。数组中的值用作方法执行时的参数。

当用 `Object []` 调用时，`proceed` 的行为与 AspectJ 编译器所编译的环绕通知的 `proceed` 行为略有不同。对于使用传统的 AspectJ 语言编写的环绕通知，传递给 `proceed` 的参数数量必须与传递给环绕通知的参数数量（而不是基础连接点采用的参数数量）相匹配，并且传递给给定的参数位置会取代该值绑定到的实体的连接点处的原始值（不要担心，如果这现在没有意义）。Spring 采取的方法更简单，并且更适合其基于代理的，仅执行的语义。仅当编译为 Spring 编写的`@AspectJ` 方面并使用 AspectJ 编译器和 weaver 的参数进行处理时，才需要意识到这种区别。有一种方法可以编写在 Spring AOP 和 AspectJ 之间 100% 兼容的方面，这将在[以下有关通知参数的部分](#)中进行讨论。

以下示例显示了如何使用环绕通知：

Java

```
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.CommonPointcuts.businessService()")
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
        // start stopwatch
        Object retVal = pjp.proceed();
        // stop stopwatch
        return retVal;
    }

}
```

Kotlin

```
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Around
import org.aspectj.lang.ProceedingJoinPoint

@Aspect
class AroundExample {

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    fun doBasicProfiling(pjp: ProceedingJoinPoint): Any {
        // start stopwatch
        val retVal = pjp.proceed()
        // stop stopwatch
        return retVal
    }

}
```

环绕通知返回的值是该方法的调用者看到的返回值。例如，如果一个简单的缓存方面有一个值，则可以从缓存中返回一个值，如果没有，则调用 `proceed()`。请注意，在环绕通知体中，`proceed` 可能被调用一次，多次或完全不被调用。所有这些都是合法的。

通知参数

Spring 提供了完全类型化的建议，这意味着您可以在通知签名中声明所需的参数（如我们先前在返回和抛出示例中所见），而不是一直使用 `Object []` 数组。我们将在本节的后面部分介绍如何使参数和其他上下文值可用于通知体。首先，我们看一下如何编写通用通知，以了解该通知当前正在通知的方法。

访问当前切入点

任何建议方法都可以将 `org.aspectj.lang.JoinPoint` 类型的参数声明为它的第一个

参数（请注意，环绕通知需要来声明 ProceedingJoinPoint 类型的第一个参数。它 JoinPoint 的子类。） JoinPoint 接口提供了许多有用的方法：

- `getArgs()`：返回方法参数。
- `getThis()`：返回代理对象。
- `getTarget()`：返回目标对象。
- `getSignature()`：返回所建议方法的描述。
- `toString()`：打印有关所建议方法的有用描述。

查看 [javadoc](#) 寻找更多详情。

向通知传参

我们已经看到了如何绑定返回的值或异常值（在后置返回和后置抛出通知之后使用）。要使参数值可用于通知体，可以使用 `args` 的绑定形式。如果在 `args` 表达式中使用参数名称代替类型名称，则在调用通知时会将相应参数的值作为参数值传递。一个例子应该使这一点更清楚。假设您要建议以 `Account` 对象作为第一个参数的 DAO 操作的执行，并且您需要在通知体中访问该帐户。你应该像下面这么写：

Java

```
@Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,...)")  
public void validateAccount(Account account) {  
    // ...  
}
```

Kotlin

```
@Before("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,...)")  
fun validateAccount(account: Account) {  
    // ...  
}
```

切入点表达式的 `args(account,..)` 部分有两个用途。首先，它将匹配限制为方法采用至少一个参数并且传递给该参数给参数为 `Account` 实例的方法执行。编写此内容的另一种方法是声明一个“提供” `Account` 对象值的切入点，当切入点与连接点匹配时，然后从通知中引用命名切入点。如下所示：

Java

```
@Pointcut("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,...)")  
private void accountDataAccessOperation(Account account) {}  
  
@Before("accountDataAccessOperation(account)")  
public void validateAccount(Account account) {  
    // ...  
}
```

Kotlin

```
@Pointcut("com.xyz.myapp.CommonPointcuts.dataAccessOperation() && args(account,...)")  
private fun accountDataAccessOperation(account: Account) {}  
  
@Before("accountDataAccessOperation(account)")  
fun validateAccount(account: Account) {  
    // ...  
}
```

有关更多详细信息，请参见 AspectJ 编程指南。

代理对象(`this`)，目标对象(`target`)和注解(`@within`, `@target`,`@annotation` 和 `@args`)都可以以类似的方式绑定。接下来的两个示例显示如何匹配使用`@Auditable` 注解标注的方法的执行并提取审计代码：

两个示例中的第一个示例显示`@Auditable` 注解的定义：

Java

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Auditable {  
    AuditCode value();  
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)  
@Target(AnnotationTarget.FUNCTION)  
annotation class Auditable(val value: AuditCode)
```

这两个示例中的第二个示例显示了与`@Auditable` 方法的执行相匹配的通知：

Java

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")  
public void audit(Auditable auditable) {  
    AuditCode code = auditable.value();  
    // ...  
}
```

Kotlin

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")  
fun audit(auditable: Auditable) {  
    val code = auditable.value()  
    // ...  
}
```

建议参数和泛型

Spring AOP 可以处理类声明和方法参数中使用的泛型。 假设您具有如下泛型类型：

Java

```
public interface Sample<T> {  
    void sampleGenericMethod(T param);  
    void sampleGenericCollectionMethod(Collection<T> param);  
}
```

Kotlin

```
interface Sample<T> {  
    fun sampleGenericMethod(param: T)  
    fun sampleGenericCollectionMethod(param: Collection<T>)  
}
```

您可以通过在要拦截方法的参数类型中键入 `advice` 参数，将方法类型的拦截限制为某些参数类型：

Java

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")  
public void beforeSampleMethod(MyType param) {  
    // Advice implementation  
}
```

Kotlin

```
@Before("execution(* ..Sample+.sampleGenericMethod(*)) && args(param)")  
fun beforeSampleMethod(param: MyType) {  
    // Advice implementation  
}
```

这种方法不适用于泛型集合。因此，您不能按以下方式定义切入点：

Java

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")  
public void beforeSampleMethod(Collection<MyType> param) {  
    // Advice implementation  
}
```

Kotlin

```
@Before("execution(* ..Sample+.sampleGenericCollectionMethod(*)) && args(param)")  
fun beforeSampleMethod(param: Collection<MyType>) {  
    // Advice implementation  
}
```

为了使这项工作有效，我们将不得不检查集合中的每个元素，这是不合理的，因为我们也无法决定通常如何处理 `null` 值。要实现类似的目的，您必须将参数键入 `Collection <?>` 并手动检查元素的类型。

判断参数名

通知调用中的参数绑定依赖于切入点表达式中使用的名称与通知和切入点方法签名中声明的参数名称的匹配。通过 Java 反射无法获得参数名称，因此 Spring AOP 使用以下策略来确定参数名称：

- 如果用户已明确指定参数名称，则使用指定的参数名称。通知和切入点注解都具有可选的 `argNames` 属性，您可以使用该属性来指定带注解的方法的参数名称。这些参数名称在运行时可用。以下示例显示如何使用 `argNames` 属性：

Java

```
@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
}
```

Kotlin

```
@Before(value = "com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)", argNames = "bean,auditable")
fun audit(bean: Any, auditable: Auditable) {
    val code = auditable.value()
    // ... use code and bean
}
```

如果第一个参数是 `JoinPoint`, `ProceedingJoinPoint` 或 `JoinPoint.StaticPart` 类型，则可以从 `argNames` 属性的值中忽略该参数的名称。例如，如果您修改前面的通知以接收连接点对象，则 `argNames` 属性不需要包括它：

Java

```
@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
}
```

Kotlin

```
@Before(value = "com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) &&
@annotation(auditable)", argNames = "bean,auditable")
fun audit(jp: JoinPoint, bean: Any, auditable: Auditable) {
    val code = auditable.value()
    // ... use code, bean, and jp
}
```

对 `JoinPoint`, `ProceedingJoinPoint` 和 `JoinPoint.StaticPart` 类型的第一个参数给予的特殊处理对于不收集任何其他联接点上下文的通知实例特别方便。在这种情况下，您可以省略 `argNames` 属性。例如，以下通知无需声明 `argNames` 属性：

Java

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
public void audit(JoinPoint jp) {
    // ... use jp
}
```

Kotlin

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")
fun audit(jp: JoinPoint) {
    // ... use jp
}
```

- 使用 'argNames' 属性有点笨拙，因此，如果未指定 'argNames' 属性，Spring AOP 将查看该类的调试信息，并尝试从局部变量表中确定参数名称。只要已使用调试信息（至少是"-g:vars"）编译了类，此信息就会存在。启用此标志时进行编译的结果是：(1) 您的代码稍微易于理解(逆向工程)，(2)类文件的大小略大(通常无关紧要)，(3)删除未使用的本地代码的优化变量不适用于您的编译器。换句话说，通过启用该标志，您应该不会遇到任何困难。



如果即使没有调试信息，AspectJ 编译器（ajc）都已编译@AspectJ 方面，则无需添加 argNames 属性，因为编译器会保留所需的信息。

- 如果在没有必要调试信息的情况下编译了代码，Spring AOP 将尝试推断绑定变量与参数的配对（例如，如果切入点表达式中仅绑定了一个变量，并且 advice 方法仅接受一个参数，则配对很明显）。如果在给定可用信息的情况下变量的绑定是不明确的，则抛出 AmbiguousBindingException。
- 如果以上所有策略均失败，则抛出 IllegalArgumentException。

参数处理

前面我们提到过，我们将描述如何编写一个在 Spring AOP 和 AspectJ 中始终有效的参数的 proceed 调用。解决方案是确保通知签名按顺序绑定每个方法参数。以下示例显示了如何执行此操作：

Java

```
@Around("execution(List<Account> find*(..)) && " +
    "com.xyz.myapp.CommonPointcuts.inDataAccessLayer() && " +
    "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp,
    String accountHolderNamePattern) throws Throwable {
    String newPattern = preProcess(accountHolderNamePattern);
    return pjp.proceed(new Object[] {newPattern});
}
```

Kotlin

```
@Around("execution(List<Account> find*(..)) && " +
    "com.xyz.myapp.CommonPointcuts.inDataAccessLayer() && " +
    "args(accountHolderNamePattern)")
fun preProcessQueryPattern(pjp: ProceedingJoinPoint,
                           accountHolderNamePattern: String): Any {
    val newPattern = preProcess(accountHolderNamePattern)
    return pjp.proceed(arrayOf<Any>(newPattern))
}
```

在许多情况下，无论如何都要进行这种绑定（如前面的示例所示）。

通知顺序

当多条通知都希望在同一连接点上运行时会发生什么？Spring AOP 遵循与 AspectJ 相同的优先级规则来确定通知执行的顺序。优先级最高的通知首先“入口”运行（因此，给定两条优先通知，则优先级最高的通知首先运行）。从连接点“出口”中，优先级最高的通知将最后运行（因此，给定两条后置通知，优先级最高的通知将后运行）。

当在不同方面定义的两条通知都需要在同一连接点上运行时，除非另行指定，否则执行顺序是不确定的。您可以通过指定优先级来控制执行顺序。通过在 Aspect 类中实现 org.springframework.core.Ordered 接口或使用 @Order 注解对其进行标记，可以通过普通的 Spring 方法来完成。给定两个方面，从 Ordered.getValue ()（或注解值）返回较低值的切面具有较高的优先级。



从 Spring Framework 5.2.7 开始，在相同@Aspect 类中定义的，需要在同一连接点运行的通知方法将根据其通知类型按照从高到低的优先级从高到低的顺序分配优先级：@Around, @Before@After, @AfterReturning, @AfterThrowing。但是请注意，由于 Spring 的 AspectJAwareAdvice 中的实现方式，在同一方面中的任何@AfterReturning 或@AfterThrowing 通知方法之后，都会有效地调用@After 通知方法。

当在同一@Aspect 类中定义的两个相同类型的通知（例如，两个@After 通知方法）都需要在同一连接点上运行时，其顺序是不确定的（因为无法检索源）代码反射通过 javac 编译类的声明顺序）。考虑将此类通知方法折叠为每个@Aspect 类中每个连接点的一通知议方法，或将建议重构为单独的@Aspect 类，您可以在这些方面通过 Ordered 或@Order 进行排序。

5.4.5. 引介

引介（在 AspectJ 中称为类型间声明）使切面可以声明通知对象实现给定的接口，并代表那些对象提供该接口的实现。

您可以使用@DeclareParents 注解进行介绍。此注解用于声明匹配类型具有新的父代（因此而得名）。例如，给定一个名为 UsageTracked 的接口和该接口名为 DefaultUsageTracked 的实现，以下方面声明服务接口的所有实现者也都实现了 UsageTracked 接口（例如，通过 JMX 公开统计信息）：

Java

```
@Aspect
public class UsageTracking {

    @DeclareParents(value="com.xyz.myapp.service.*",
    defaultImpl=DefaultUsageTracked.class)
    public static UsageTracked mixin;

    @Before("com.xyz.myapp.CommonPointcuts.businessService() && this(usageTracked)")
    public void recordUsage(UsageTracked usageTracked) {
        usageTracked.incrementUseCount();
    }
}
```

Kotlin

```
@Aspect
class UsageTracking {

    companion object {
        @DeclareParents(value = "com.xzy.myapp.service.*+", defaultImpl =
DefaultUsageTracked::class)
            lateinit var mixin: UsageTracked
    }

    @Before("com.xyz.myapp.CommonPointcuts.businessService() && this(usageTracked)")
    fun recordUsage(usageTracked: UsageTracked) {
        usageTracked.incrementUseCount()
    }
}
```

要实现的接口由带注解的字段的类型确定。`@DeclareParents` 注解的 `value` 属性是 AspectJ 类型的模式。匹配类型的任何 Bean 均实现 `UsageTracked` 接口。请注意，在前面示例的之前通知中，服务 Bean 可以直接用作 `UsageTracked` 接口的实现。如果以编程方式访问 bean，则应编写以下内容：

Java

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

Kotlin

```
val usageTracked = context.getBean("myService") as UsageTracked
```

5.4.6. Aspect 实例化模型



这是一个高级主题。如果您刚开始使用 AOP，则可以放心地跳过它，之后再回来看。

默认情况下，应用程序上下文中每个方面都有一个实例。AspectJ 将此称为单例实例化模型。可以使用备用生命周期来定义方面。Spring 支持 AspectJ 的 `perthis` 和 `pertarget` 实例化模型；目前不支持 `percflow`, `percflowbelow` 和 `pertypewithin`。

您可以通过在`@Aspect`注解中指定 `perthis`子句来声明 `perthis`切面。考虑以下示例：

Java

```
@Aspect("perthis(com.xyz.myapp.CommonPointcuts.businessService())")
public class MyAspect {

    private int someState;

    @Before("com.xyz.myapp.CommonPointcuts.businessService()")
    public void recordServiceUsage() {
        // ...
    }

}
```

Kotlin

```
@Aspect("perthis(com.xyz.myapp.CommonPointcuts.businessService())")
class MyAspect {

    private val someState: Int = 0

    @Before("com.xyz.myapp.CommonPointcuts.businessService()")
    fun recordServiceUsage() {
        // ...
    }

}
```

在前面的示例中，`perthis` 子句的作用是为执行业务服务的每个唯一服务对象（每个与切入点表达式匹配的连接点绑定到此的唯一对象）创建一个切面实例。方面实例是在服务对象上首次调用方法时创建的。当服务对象超出范围时，切面将超出范围。在创建方面实例之前，其中的任何通知都不会运行。创建切面实例后，在其中声明的建议将在匹配的连接点处运行，但是仅当服务对象是与此方面相关联的对象时才运行。有关每个子句的更多信息，请参见 AspectJ 编程指南。

`pertarget` 实例化模型的工作方式与 `perthis` 完全相同，但是它为每个唯一目标对象在匹配的连接点创建一个切面实例。

5.4.7. 一个 AOP 例子

既然您已经了解了所有组成部分是如何工作的，那么我们可以将它们放在一起做一些有用的事情。

有时由于并发问题（例如，死锁），业务服务的执行可能会失败。如果重试该操作，则很可能在下一次尝试中成功。对于适合在此类情况下重试的业务服务（不需要为解决冲突而需要返回给用户的幂等操作），我们希望透明地重试该操作，以避免客户端看到

`PessimisticLockingFailureException`。这项要求清楚地跨越了服务层中的多个服务，因此非常适合通过一个切面实施。因为我们想重试该操作，所以我们需要使用环绕通知，以便可以多次调用 `proceed`。以下列表显示了基本方面的实现：

Java

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

    private static final int DEFAULT_MAX_RETRIES = 2;

    private int maxRetries = DEFAULT_MAX_RETRIES;
    private int order = 1;

    public void setMaxRetries(int maxRetries) {
        this.maxRetries = maxRetries;
    }

    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }
}
```

Kotlin

```
@Aspect
class ConcurrentOperationExecutor : Ordered {

    private val DEFAULT_MAX_RETRIES = 2
    private var maxRetries = DEFAULT_MAX_RETRIES
    private var order = 1

    fun setMaxRetries(maxRetries: Int) {
        this.maxRetries = maxRetries
    }

    override fun getOrder(): Int {
        return this.order
    }

    fun setOrder(order: Int) {
        this.order = order
    }

    @Around("com.xyz.myapp.CommonPointcuts.businessService()")
    fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
        var numAttempts = 0
        var lockFailureException: PessimisticLockingFailureException
        do {
            numAttempts++
            try {
                return pjp.proceed()
            } catch (ex: PessimisticLockingFailureException) {
                lockFailureException = ex
            }
        } while (numAttempts <= this.maxRetries)
        throw lockFailureException
    }
}
```

请注意，方面实现了 `Ordered` 接口，因此我们可以将切面的优先级设置为高于事务通知（每次重试时都希望有新的事务）。`maxRetries` 和 `order` 属性均由 Spring 配置。通知的主要动作发生在 `doConcurrentOperation` 中。注意，目前，我们将重试逻辑应用于每个 `businessService()`。我们尝试继续，如果失败并出现 `PessimisticLockingFailureException`，则我们将再次尝试，除非我们用尽了所有重试尝试。

相应的 Spring 配置如下：

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
      class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>
```

为了完善切面，使其仅重试幂等运算，我们可以定义以下幂等注解：

Java

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}
```

Kotlin

```
@Retention(AnnotationRetention.RUNTIME)
annotation class Idempotent// marker annotation
```

然后，我们可以使用注解来标识服务操作的实现。切面的更改仅重试幂等操作涉及到改进切入点表达式，因此只与@Idempotent 操作匹配，如下所示：

Java

```
@Around("com.xyz.myapp.CommonPointcuts.businessService() && "
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
    // ...
}
```

Kotlin

```
@Around("com.xyz.myapp.CommonPointcuts.businessService() && "
        "@annotation(com.xyz.myapp.service.Idempotent)")
fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
    // ...
}
```

5.5. 基于 Schema 的 AOP 支持

如果您喜欢基于 XML 的格式，Spring 还提供了对使用 aop 名称空间标签定义方面的支持。支持与使用@AspectJ 样式时完全相同的切入点表达式和通知类型。因此，在本节中，我们将重点放在该语法上，并使读者参考上一节中的讨论（[@AspectJ 支持\(5.4\)](#)），以了解编写切入点表达式和通知参数的绑定。

要使用本节中描述的 aop 名称空间标签，您需要导入 spring-aop 模式，如[基于 XML](#)

Schema 的配置中(9.1)所述。有关如何在 aop 名称空间中导入标签的信息，请参见 AOP 模式(9.1.2)。

在您的 Spring 配置中，所有切面和通知元素都必须放在[`<aop:config>`](#)元素内（在应用程序上下文配置中可以有多个[`<aop:config>`](#)元素）。[`<aop:config>`](#)元素可以包含切入点，通知和 aspect 元素（请注意，必须按此顺序声明它们）。



[`<aop: config>`](#)的配置样式大量使用了 Spring 的自动代理机制。如果您已经通过使用 `BeanNameAutoProxyCreator` 或类似方法使用显式自动代理，则可能会导致问题（例如，通知未被织入）。推荐的使用模式是仅使用[`<aop: config>`](#)样式或仅使用 `AutoProxyCreator` 样式，并且不要混合使用。

5.5.1. 声明一个 Aspect

使用模式支持时，方面是在 Spring 应用程序上下文中定义为 Bean 的常规 Java 对象。状态和行为在对象的字段和方法中捕获，切入点和通知信息在 XML 中捕获。您可以使用[`<aop:aspect>`](#)元素声明一个切面，并使用 `ref` 属性引用该支持 bean，如以下示例所示：

```
<aop:config>
    <aop:aspect id="myAspect" ref="aBean">
        ...
    </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
    ...
</bean>
```

支持切面的 bean（在本例中为 `aBean`）当然可以像配置其他 Spring bean 一样进行配置并注入依赖项。

5.5.2. 声明一个切入点

您可以在[`<aop: config>`](#)元素内声明一个命名的切入点，让切入点定义在多个切面和通知之间共享。

可以定义代表服务层中任何业务服务的执行的切入点：

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

请注意，切入点表达式本身使用的是[@AspectJ 支持\(5.4\)](#)中所述的 AspectJ 切入点表达式语言。如果使用基于架构的声明样式，则可以引用在切入点表达式中的类型(@Aspects)中定义的命名切入点。定义上述切入点的另一种方法如下：

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="com.xyz.myapp.CommonPointcuts.businessService()"/>

</aop:config>
```

假如你有一个叫 CommonPointcuts 的切面以[共享公共切入点定义](#)描述。

然后，在切面中声明切入点与声明顶级切入点非常相似，如以下示例所示：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        ...

    </aop:aspect>

</aop:config>
```

与[@AspectJ](#)切面几乎相同，使用基于架构的定义样式声明的切入点可以收集连接点上下文。例如，以下切入点将收集此对象作为连接点上下文，并将其传递给通知：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">
        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) && this(service)"/>
        <aop:before pointcut-ref="businessService" method="monitor"/>
        ...
    </aop:aspect>
</aop:config>
```

必须声明通知，以通过包含匹配名称的参数来接收收集的连接点上下文，如下所示：

Java

```
public void monitor(Object service) {
    // ...
}
```

Kotlin

```
fun monitor(service: Any) {
    // ...
}
```

当与切入点子表达式结合，`&&`在 XML 文件中太笨拙了，所以你可以时候用 `and`, `or`, `not` 关键字替代`&&`, `||`和`!`，例如，上一个切入点可以更好地编写如下：

```
<aop:config>

    <aop:aspect id="myAspect" ref="aBean">

        <aop:pointcut id="businessService"
            expression="execution(* com.xyz.myapp.service.*.*(..)) and
            this(service)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>

        ...
    </aop:aspect>
</aop:config>
```

请注意，以这种方式定义的切入点由其 XML **ID** 引用，并且不能用作命名切入点以形成复合切入点。因此，基于架构的定义样式中的命名切入点支持比@AspectJ 样式所提供的更受限制。

5.5.3. 声明通知

基于模式的 AOP 支持使用与@AspectJ 样式相同的五种通知，并且它们具有完全相同的语义。

前置通知

前置通知：在运行匹配的方法之前。 使用[<aop: before>](#)元素在[<aop: aspect>](#)中声明它，如以下示例所示：

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
        pointcut-ref="dataAccessOperation"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

在这里，**dataAccessOperation** 是在最高 ([<aop: config>](#)) 级别定义的切入点的 **ID**。要定义内联切入点，请使用以下方法将 **pointcut-ref** 属性替换为 **pointcut** 属性：

```
<aop:aspect id="beforeExample" ref="aBean">  
    <aop:before  
        pointcut="execution(* com.xyz.myapp.dao.*.*(..))"  
        method="doAccessCheck"/>  
    ...  
</aop:aspect>
```

正如我们在@AspectJ 样式的讨论中所指出的那样，使用命名的切入点可以显着提高代码的可读性。

`method` 属性标识提供通知体的方法（`doAccessCheck`）。必须为包含建议的 Aspect 元素所引用的 bean 定义此方法。在执行数据访问操作（与切入点表达式匹配的方法执行连接点）之前，将调用 Aspect Bean 上的 `doAccessCheck` 方法。

后置返回通知

后置返回通知运行在一个匹配的方法完全执行完毕之后。它以与前置通知相同的方式在 `<aop:aspect>` 中声明。以下示例显示了如何声明它：

```
<aop:aspect id="afterReturningExample" ref="aBean">  
    <aop:after-returning  
        pointcut-ref="dataAccessOperation"  
        method="doAccessCheck"/>  
    ...  
</aop:aspect>
```

与@AspectJ 样式一样，您可以在通知体中获取返回值。为此，使用 `returning` 属性指定返回值应传递到的参数的名称，如以下示例所示：

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
        pointcut-ref="dataAccessOperation"
        returning="retVal"
        method="doAccessCheck"/>

    ...

</aop:aspect>
```

`doAccessCheck` 方法必须声明一个名为 `retVal` 的参数。该参数的类型以与 `@AfterReturning` 中所述相同的方式约束匹配。例如，您可以声明方法签名，如下所示：

Java

```
public void doAccessCheck(Object retVal) { ... }
```

Kotlin

```
fun doAccessCheck(retVal: Any) { ... }
```

后置抛出通知

后置抛出通知运行在一个匹配的方法以抛出一个异常为结束之后。使用 `after-throwing` 元素在 `<aop:aspect>` 中声明它，如以下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        method="doRecoveryActions"/>

    ...

</aop:aspect>
```

与 `@AspectJ` 样式一样，您可以在通知体中获取引发的异常。为此，请使用 `throwing` 属性指定异常应传递到的参数的名称，如以下示例所示：

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
        pointcut-ref="dataAccessOperation"
        throwing="dataAccessEx"
        method="doRecoveryActions"/>

    ...

</aop:aspect>
```

`doRecoveryActions` 方法必须声明一个名为 `dataAccessEx` 的参数。该参数的类型以与 `@AfterThrowing` 中所述相同的方式约束匹配。例如，方法签名可以声明如下：

Java

```
public void doRecoveryActions(DataAccessException dataAccessEx) { ... }
```

Kotlin

```
fun doRecoveryActions(dataAccessEx: DataAccessException) { ... }
```

后置(最终)通知

无论最终如何执行匹配的方法，后置(最终)通知都会运行。您可以使用 `after` 元素声明它，如以下示例所示：

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
        pointcut-ref="dataAccessOperation"
        method="doReleaseLock"/>

    ...

</aop:aspect>
```

环绕通知

最后一种通知是环绕通知。环绕通知在匹配的方法执行过程中“环绕”运行。它有机会在该方法运行之前和之后进行工作，并确定何时，如何以及是否该方法实际上可以运行。环绕通知通常用于以线程安全的方式（例如，启动和停止计时器）在方法执行之前和之后共享状态。始终使用最弱的通知形式，以满足您的要求。如果前置通知可以完成这项工作，请不要用环绕通知。

您可以使用 `aop: around` 元素声明环绕通知。通知方法的第一个参数必须是

`ProceedingJoinPoint` 类型。在通知体中，在 `ProceedingJoinPoint` 上调用 `proceed()` 会使底层方法运行。可以使用 `Object[]` 调用 `proceed` 方法。数组中的值用作方法执行时的参数。有关使用 `Object []` 作为参数调用 `proceed` 方法注意事项，请参见“[环绕通知\(340页\)](#)”。以下示例显示了如何在 XML 中环绕通知进行声明：

```
<aop:aspect id="aroundExample" ref="aBean">  
    <aop:around  
        pointcut-ref="businessService"  
        method="doBasicProfiling"/>  
  
    ...  
</aop:aspect>
```

`doBasicProfiling` 通知的实现可以与@AspectJ 示例完全相同（当然要减去注解），如以下示例所示：

Java

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {  
    // start stopwatch  
    Object retVal = pjp.proceed();  
    // stop stopwatch  
    return retVal;  
}
```

Kotlin

```
fun doBasicProfiling(pjp: ProceedingJoinPoint): Any {  
    // start stopwatch  
    val retVal = pjp.proceed()  
    // stop stopwatch  
    return pjp.proceed()  
}
```

通知参数

基于架构的声明样式以与@AspectJ 支持相同的方式支持完全类型的通知，即通过名称与通知方法参数匹配切入点参数。有关详细信息，请参见[通知参数](#)。如果您希望为通知方法明确指定参数名称（不依赖于先前描述的检测策略），则可以通过使用 `advice` 元素的 `arg-names` 属性来实现，该属性的处理方式与通知注解 `argNames` 属性相同（如确定参数名称中所述）。以下示例显示如何在 XML 中指定参数名称：

```
<aop:before  
    pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"  
    method="audit"  
    arg-names="auditable"/>
```

arg-names 属性接受以逗号分隔的参数名称列表。

以下基于 XSD 的方法中涉及程度稍高的示例显示了一些与一些强类型参数结合使用的通知：

Java

```
package x.y.service;  
  
public interface PersonService {  
  
    Person getPerson(String personName, int age);  
}  
  
public class DefaultPersonService implements PersonService {  
  
    public Person getPerson(String name, int age) {  
        return new Person(name, age);  
    }  
}
```

Kotlin

```
package x.y.service  
  
interface PersonService {  
  
    fun getPerson(personName: String, age: Int): Person  
}  
  
class DefaultPersonService : PersonService {  
  
    fun getPerson(name: String, age: Int): Person {  
        return Person(name, age)  
    }  
}
```

接下来是切面。请注意 `profile(..)` 方法接受许多强类型参数的事实，其中第一个恰好是用于进行方法调用的连接点。此参数的存在表明 `profile(..)` 将用作通知，如以下示例所示：

Java

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

    public Object profile(ProceedingJoinPoint call, String name, int age) throws
Throwable {
        StopWatch clock = new StopWatch("Profiling for '" + name + "' and '" + age +
"']");
        try {
            clock.start(call.toShortString());
            return call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
    }
}
```

Kotlin

```
import org.aspectj.lang.ProceedingJoinPoint
import org.springframework.util.StopWatch

class SimpleProfiler {

    fun profile(call: ProceedingJoinPoint, name: String, age: Int): Any {
        val clock = StopWatch("Profiling for '$name' and '$age'")
        try {
            clock.start(call.toShortString())
            return call.proceed()
        } finally {
            clock.stop()
            println(clock.prettyPrint())
        }
    }
}
```

最后，以下示例 XML 配置影响了特定连接点的上述通知的执行：

```

<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">

    <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
    <bean id="personService" class="x.y.service.DefaultPersonService"/>

    <!-- this is the actual advice itself -->
    <bean id="profiler" class="x.y.SimpleProfiler"/>

    <aop:config>
        <aop:aspect ref="profiler">

            <aop:pointcut id="theExecutionOfSomePersonServiceMethod"
                expression="execution(*
                    x.y.service.PersonService.getPerson(String,int))
                and args(name, age)"/>

            <aop:around pointcut-ref="theExecutionOfSomePersonServiceMethod"
                method="profile"/>

        </aop:aspect>
    </aop:config>

</beans>

```

考慮以下驅動程序腳本：

Java

```

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.PersonService;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
        PersonService person = (PersonService) ctx.getBean("personService");
        person.getPerson("Pengo", 12);
    }
}

```

Kotlin

```
fun main() {
    val ctx = ClassPathXmlApplicationContext("x/y/plain.xml")
    val person = ctx.getBean("personService") as PersonService
    person.getPerson("Pengo", 12)
}
```

有了这样的 Boot 类，我们将在标准输出上获得类似于以下内容的输出：

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
-----
ms      %      Task name
-----
00000  ?  execution(getFoo)
```

通知顺序

当需要在同一连接点（执行方法）上运行多个通知时，排序规则如“[通知排序](#)”中所述。方面之间的优先级是通过[`<aop: aspect>`](#)元素中的 `order` 属性或通过将[`@Order`](#)注解添加到支持该方面的 bean 或通过使 bean 实现 `Ordered` 接口来确定的。

与在同一[`@Aspect`](#)类中定义的通知方法的优先规则相反，当在同一[`<aop: aspect>`](#)元素中定义的两条通知都需要在同一连接点上运行时，优先级由在包围的[`<aop: aspect>`](#)元素中声明的通知元素中的顺序确定，从最高优先级到最低优先级。

例如，给定一个环绕通知和一个在同一[`<aop: aspect>`](#)元素中定义的，适用于同一连接点的前置通知，以确保环绕通知的优先级高于前置通知。[`<aop: around>`](#)元素必须在[`<aop: before>`](#)元素之前声明。根据一般经验，如果发现在同一[`<aop: aspect>`](#)元素中定义了多个通知，这些通知适用于同一连接点，请考虑将这些通知方法折叠为每个[`<aop: aspect>`](#)元素，或将建议重构为单独的[`<aop: aspect>`](#)元素，您可以在切面层级进行排序。

5.5.4. 引介

引介（在 AspectJ 中称为类型间声明）使切面可以声明通知的对象实现给定的接口，并代表那些对象提供该接口的实现。

您可以通过在 `aop: aspect` 中使用 `aop: declare-parents` 元素设置一个引介。您可以使用 `aop: declare-parents` 元素来声明匹配类型具有新的父代（因此而得名）。例如，给定一个名为 `UsageTracked` 的接口和该接口名为 `DefaultUsageTracked` 的实现，以下方

面声明服务接口的所有实现者也都实现了 UsageTracked 接口。（例如，为了通过 JMX 公开统计信息。）

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

    <aop:declare-parents
        types-matching="com.xyz.myapp.service.*+"
        implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
        default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

    <aop:before
        pointcut="com.xyz.myapp.CommonPointcuts.businessService()
            and this(usageTracked)"
        method="recordUsage"/>

</aop:aspect>
```

支持 usageTracking bean 的类将包含以下方法：

Java

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

Kotlin

```
fun recordUsage(usageTracked: UsageTracked) {
    usageTracked.incrementUseCount()
}
```

要实现的接口由 Implement-interface 属性确定。类型匹配属性的值是 AspectJ 类型模式。匹配类型的任何 Bean 均实现 UsageTracked 接口。请注意，在前面示例的前置通知中，服务 Bean 可以直接用作 UsageTracked 接口的实现。要以编程方式访问 bean，可以编写以下代码：

Java

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

Kotlin

```
val usageTracked = context.getBean("myService") as UsageTracked
```

5.5.5. Aspect 实例化模型

模式定义方面唯一受支持的实例化模型是单例模型。在将来的版本中可能会支持其他实例化模型。

5.5.6. 一般切面

“通知”的概念来自 Spring 中定义的 AOP 支持，并且在 AspectJ 中没有直接等效的概念。一般切面就像一个独立的小切面，只有一条建议。通知本身由 bean 表示，并且必须实现 Spring 的“通知类型”中描述的通知接口之一。通知可以利用 AspectJ 切入点表达式。

Spring 通过<aop: advisor>元素支持通知程序概念。您最常看到它与事务通知结合使用，事务通知在 Spring 中也有其自己的命名空间支持。以下示例显示了一个通知：

```
<aop:config>

    <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:advisor
        pointcut-ref="businessService"
        advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

除了前面的示例中使用的 `pointcut-ref` 属性，您还可以使用 `pointcut` 属性内联定义一个 `pointcut` 表达式。

要定义一般切面的优先级以使通知可以进行排序，请使用 `order` 属性定义顾问程序的 `Ordered` 值。

5.5.7. 一个 AOP schema 例子

本节显示使用模式支持重写时，AOP 示例中的并发锁定失败重试示例的外观。有时由于并发问题（例如，死锁），业务服务的执行可能会失败。如果重试该操作，则很可能在下一次尝试中成功。对于适合在此类情况下重试的业务服务（不需要为解决冲突而需要返回给用户的幂等操作），我们希望透明地重试该操作，以避免客户端看到

`PessimisticLockingFailureException`。这项要求清楚地跨越了服务层中的多个服务，因此非常适合通过一个切面实施。

因为我们想重试该操作，所以我们需要使用环绕建议，以便可以多次调用 `proceed`。以下列表显示了基本切面的实现（这是使用模式支持的常规 Java 类）：

Java

```
public class ConcurrentOperationExecutor implements Ordered {

    private static final int DEFAULT_MAX_RETRIES = 2;

    private int maxRetries = DEFAULT_MAX_RETRIES;
    private int order = 1;

    public void setMaxRetries(int maxRetries) {
        this.maxRetries = maxRetries;
    }

    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
        int numAttempts = 0;
        PessimisticLockingFailureException lockFailureException;
        do {
            numAttempts++;
            try {
                return pjp.proceed();
            }
            catch(PessimisticLockingFailureException ex) {
                lockFailureException = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw lockFailureException;
    }

}
```

```

class ConcurrentOperationExecutor : Ordered {

    private val DEFAULT_MAX_RETRIES = 2

    private var maxRetries = DEFAULT_MAX_RETRIES
    private var order = 1

    fun setMaxRetries(maxRetries: Int) {
        this.maxRetries = maxRetries
    }

    override fun getOrder(): Int {
        return this.order
    }

    fun setOrder(order: Int) {
        this.order = order
    }

    fun doConcurrentOperation(pjp: ProceedingJoinPoint): Any {
        var numAttempts = 0
        var lockFailureException: PessimisticLockingFailureException
        do {
            numAttempts++
            try {
                return pjp.proceed()
            } catch (ex: PessimisticLockingFailureException) {
                lockFailureException = ex
            }
        } while (numAttempts <= this.maxRetries)
        throw lockFailureException
    }
}

```

请注意，切面实现了 `Ordered` 接口，因此我们可以将切面的优先级设置为高于事务通知（每次重试时都希望有新的事务）。`maxRetries` 和 `order` 属性都由 Spring 配置。主要操作发生在通知方法周围的 `doConcurrentOperation` 中。我们尝试处理。如果由于 `PessimisticLockingFailureException` 失败，则将重试，除非我们用尽了所有重试尝试。



该类与@AspectJ 示例中使用的类相同，但是删除了注解。

相应的 Spring 配置如下：

```

<aop:config>

    <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">
        <aop:pointcut id="idempotentOperation"
            expression="execution(* com.xyz.myapp.service.*.*(..))"/>

        <aop:around
            pointcut-ref="idempotentOperation"
            method="doConcurrentOperation"/>

    </aop:aspect>
</aop:config>

<bean id="concurrentOperationExecutor"
    class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
    <property name="maxRetries" value="3"/>
    <property name="order" value="100"/>
</bean>

```

请注意，目前我们假设所有业务服务都是幂等的。如果不是这种情况，我们可以改进切面，以便通过引入等幂注解并使用该注解来注解服务操作的实现，使其仅重试真正的幂等操作，如以下示例所示：

Java

```

@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
    // marker annotation
}

```

Kotlin

```

@Retention(AnnotationRetention.RUNTIME)
annotation class Idempotent {
    // marker annotation
}

```

切面的更改仅重试幂等操作涉及到改进切入点表达式，以便使@Idempotent 操作匹配，如下所示：

```

<aop:pointcut id="idempotentOperation"
    expression="execution(* com.xyz.myapp.service.*.*(..)) and
    @annotation(com.xyz.myapp.service.Idempotent)"/>

```

5.6. 选择 AOP 声明风格

一旦确定方面是实现给定需求的最佳方法，您如何在使用 Spring AOP 或 AspectJ 以及在 Aspect 语言（代码）样式，@AspectJ 注解样式或 Spring XML 样式之间做出选择？这些决定受许多因素影响，包括应用程序需求，开发工具以及团队对 AOP 的熟悉程度。

5.6.1. Spring AOP 还是全 AspectJ？

使用最简单的方法即可。Spring AOP 比使用完整的 AspectJ 更简单，因为不需要在开发和构建过程中引入 AspectJ 编译器/编织器。如果您只需要在 Spring bean 上执行通知操作，则 Spring AOP 是正确的选择。如果您需要通知不受 Spring 容器管理的对象（通常是字段对象），则需要使用 AspectJ。如果您希望通知除简单方法执行以外的连接点（例如，字段获取或设置连接点等），则还需要使用 AspectJ。

使用 AspectJ 时，可以选择 AspectJ 语言语法（也称为“代码样式”）或 @AspectJ 注解样式。显然，如果不使用 Java 5+，则可以为您做出选择：使用代码样式。如果方面在您的设计中起着重要的作用，并且您能够将 [AspectJ 开发工具（AJDT）](#) 插件用于 Eclipse，则 AspectJ 语言语法是首选。它更干净，更简单，因为该语言是专为编写切面而设计的。如果您不使用 Eclipse 或只有少数几个切面在您的应用程序中不起作用，那么您可能要考虑使用 @AspectJ 样式当在 IDE 中坚持常规 Java 编译，并向您的构建脚本添加切面编织阶段。

5.6.2. @AspectJ 还是 XML，哪个和 Spring AOP 更配？

如果选择使用 Spring AOP，则可以选择 @AspectJ 或 XML 样式。有各种折衷考虑。

XML 样式可能是现有 Spring 用户最熟悉的，并且得到了真正的 POJO 的支持。当使用 AOP 作为配置企业服务的工具时，XML 可能是一个不错的选择（一个很好的测试是您是否将切入点表达式视为配置的一部分，您可能希望独立更改）。使用 XML 样式，可以说从您的配置中可以更清楚地了解系统中存在哪些切面。

XML 样式有两个缺点。首先，它没有完全将要解决的需求的实现封装在一个地方。DRY 原则说，系统中的任何知识都应该有一个单一，明确，权威的表示形式。当使用 XML 样式时，关于如何实现需求的说明会在配置文件中的后备 bean 类的声明和 XML 中分散。当您使用 @AspectJ 样式时，此信息封装在一个模块中：切面。其次，与 @AspectJ 样式相比，XML 样式在表达能力上有更多限制：仅支持“单例”方面实例化模型，并且无法组合以 XML 声明的命名切入点。例如，使用 @AspectJ 样式，您可以编写如下内容：

Java

```
@Pointcut("execution(* get*())")
public void propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ *(..))")
public void operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")
public void accountPropertyAccess()
```

Kotlin

```
@Pointcut("execution(* get*())")
fun propertyAccess() {}

@Pointcut("execution(org.xyz.Account+ *(..))")
fun operationReturningAnAccount() {}

@Pointcut("propertyAccess() && operationReturningAnAccount()")
fun accountPropertyAccess()
```

在 XML 样式中，您可以声明前两个切入点：

```
<aop:pointcut id="propertyAccess"
    expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

XML 方法的缺点是您无法通过组合这些定义来定义 `accountPropertyAccess` 切入点。

`@AspectJ` 样式支持其他实例化模型和更丰富的切入点组合。它具有将切面保持为模块化单元的优势。它还具有的优点是，Spring AOP 和 AspectJ 都可以理解`@AspectJ` 切面。因此，如果您以后决定需要 AspectJ 的功能来实现其他要求，则可以轻松地迁移到经典的 AspectJ 设置。总体而言，Spring 团队在自定义方面更喜欢`@AspectJ` 样式，而不是简单地配置企业服务。

5.7. 混合 Aspect 类型

通过使用自动代理支持，模式定义的`<aop: aspect>`切面，`<aop: advisor>`声明的通知程序，甚至使用同一配置中的其他样式的代理和拦截器，可以完美地混合`@AspectJ` 样式的切面。所有这些都是通过使用相同的基础支持机制实现的，并且可以毫无困难地共存。

5.8. 代理机制

Spring AOP 使用 JDK 动态代理或 CGLIB 创建给定目标对象的代理。JDK 内置了 JDK 动态代理，而 CGLIB 是常见的开源类定义库（重新包装到 spring-core 中）。

如果要代理的目标对象实现至少一个接口，则使用 JDK 动态代理。代理了由目标类型实现的所有接口。如果目标对象未实现任何接口，则将创建 CGLIB 代理。

如果要强制使用 CGLIB 代理（例如，代理为目标对象定义的每个方法，而不仅是由其接口实现的方法），都可以这样做。但是，您应该考虑以下问题：

- 使用 CGLIB，不能通知 final 方法，因为不能在运行时生成的子类中覆盖 final 方法。
- 从 Spring 4.0 开始，由于 CGLIB 代理实例是通过 Objenesis 创建的，因此不再调用代理对象的构造函数两次。仅当您的 JVM 不允许绕过构造函数时，您才可能从 Spring 的 AOP 支持中看到两次调用和相应的调试日志条目。

要强制使用 CGLIB 代理，请将 `<aop: config>` 元素的 `proxy-target-class` 属性的值设置为 `true`，如下所示：

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

要在使用 @AspectJ 自动代理支持时强制 CGLIB 代理，请将 `<aop : aspectj-autoproxy>` 元素的 `proxy-target-class` 属性设置为 `true`，如下所示：

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

多个 `<aop: config>` 部分在运行时折叠到一个统一的自动代理创建器中，该创建器将应用任何 `<aop: config />` 部分（通常来自不同 XML Bean 定义文件）指定的最强代理设置。这也适用于 `<tx: annotation-driven/>` 和 `<aop: aspectjautoproxy/>` 元素。



明确地说，在 `<tx : annotation-driven/>`，`<aop : aspectj-autoproxy/>` 或 `<aop : config/>` 元素上使用 `proxy-target-class=“true”` 会强制对所有三个元素使用 CGLIB 代理。

5.8.1. 理解 AOP 代理

Spring AOP 是基于代理的。在编写自己的切面或使用 Spring Framework 提供的任

何基于 Spring AOP 的切面之前，掌握最后一条语句实际含义的语义至关重要。

首先考虑您有一个普通的，未经代理的，没有什么特别的，直接的对象引用的情况，如以下代码片段所示：

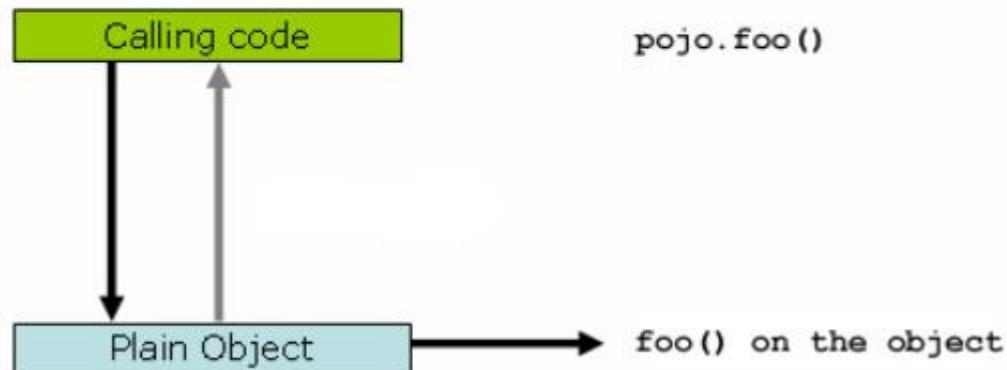
Java

```
public class SimplePojo implements Pojo {  
  
    public void foo() {  
        // this next method invocation is a direct call on the 'this' reference  
        this.bar();  
    }  
  
    public void bar() {  
        // some logic...  
    }  
}
```

Kotlin

```
class SimplePojo : Pojo {  
  
    fun foo() {  
        // this next method invocation is a direct call on the 'this' reference  
        this.bar()  
    }  
  
    fun bar() {  
        // some logic...  
    }  
}
```

如果在对象引用上调用方法，则直接在该对象引用上调用该方法，如下图和清单所示：



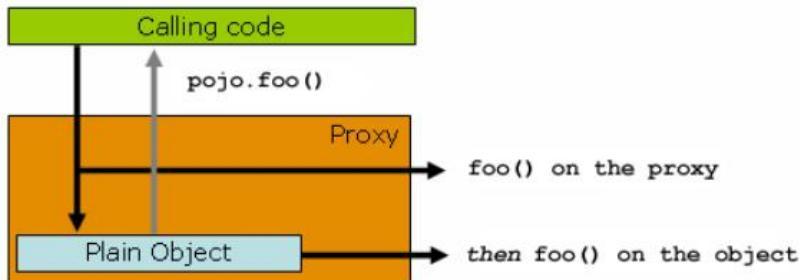
Java

```
public class Main {  
  
    public static void main(String[] args) {  
        Pojo pojo = new SimplePojo();  
        // this is a direct method call on the 'pojo' reference  
        pojo.foo();  
    }  
}
```

Kotlin

```
fun main() {  
    val pojo = SimplePojo()  
    // this is a direct method call on the 'pojo' reference  
    pojo.foo()  
}
```

当客户端代码具有的引用是代理时，情况会稍有变化。 考虑以下图表和代码片段：



Java

```
public class Main {  
  
    public static void main(String[] args) {  
        ProxyFactory factory = new ProxyFactory(new SimplePojo());  
        factory.addInterface(Pojo.class);  
        factory.addAdvice(new RetryAdvice());  
  
        Pojo pojo = (Pojo) factory.getProxy();  
        // this is a method call on the proxy!  
        pojo.foo();  
    }  
}
```

Kotlin

```
fun main() {
    val factory = ProxyFactory(SimplePojo())
    factory.addInterface(Pojo::class.java)
    factory.addAdvice(RetryAdvice())

    val pojo = factory.proxy as Pojo
    // this is a method call on the proxy!
    pojo.foo()
}
```

这里要理解的关键是 `Main` 类的 `main(..)` 方法中的客户端代码具有对代理的引用。这意味着该对象引用上的方法调用是代理上的调用。结果，代理可以委派给与该特定方法调用相关的所有拦截器（通知）。但是，一旦调用最终到达目标对象（在这种情况下为 `SimplePojo` 引用），则可能会针对它调用可能对其自身进行的任何方法调用，例如 `this.bar()` 或 `this.foo()` 针对 `this` 参考的调用，而不是代理。这具有重要的意义。这意味着自调用不会导致与方法调用相关的通知得到运行的机会。

好吧，那该怎么办？最佳方法（此处宽松地使用术语“最佳”）是重构代码，以免发生自调用。这确实需要您做一些工作，但这是最好的，侵入性最小的方法。下一种方法绝对可怕，我们正要指出这一点，恰恰是因为它是如此可怕。您可以（对我们来说是痛苦的）完全将类中的逻辑与 Spring AOP 绑定在一起，如以下示例所示：

Java

```
public class SimplePojo implements Pojo {

    public void foo() {
        // this works, but... gah!
        ((Pojo) AopContext.currentProxy()).bar();
    }

    public void bar() {
        // some logic...
    }
}
```

Kotlin

```
class SimplePojo : Pojo {

    fun foo() {
        // this works, but... gah!
        (AopContext.currentProxy() as Pojo).bar()
    }

    fun bar() {
        // some logic...
    }
}
```

这完全将您的代码与 Spring AOP 结合在一起，并且它使类本身意识到，它正在 AOP 上下文中使用，这与 AOP 背道而驰。创建代理时，还需要一些其他配置，如以下示例所示：

Java

```
public class Main {

    public static void main(String[] args) {
        ProxyFactory factory = new ProxyFactory(new SimplePojo());
        factory.addInterface(Pojo.class);
        factory.addAdvice(new RetryAdvice());
        factory.setExposeProxy(true);

        Pojo pojo = (Pojo) factory.getProxy();
        // this is a method call on the proxy!
        pojo.foo();
    }
}
```

Kotlin

```
fun main() {
    val factory = ProxyFactory(SimplePojo())
    factory.addInterface(Pojo::class.java)
    factory.addAdvice(RetryAdvice())
    factory.isExposeProxy = true

    val pojo = factory.proxy as Pojo
    // this is a method call on the proxy!
    pojo.foo()
}
```

最后，必须注意，AspectJ 没有此自调用问题，因为它不是基于代理的 AOP 框架。

5.9. 编程式@AspectJ 代理的创建

除了使用[`<aop: config>`](#)或[`<aop: aspectjautoproxy>`](#)声明配置中的各个切面外，还可以通过编程方式创建通知目标对象的代理。有关 Spring 的 AOP API 的完整详细信息，请参阅[下一章](#)。在这里，我们要重点介绍通过使用@AspectJ 方便自动创建代理的功能。您可以使用[`org.springframework.aop.aspectj.annotation.AspectJProxyFactory`](#)类为一个或多个@AspectJ 方便建议的目标对象创建代理。此类的基本用法非常简单，如以下示例所示：

Java

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be
// an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

Kotlin

```
// create a factory that can generate a proxy for the given target object
val factory = AspectJProxyFactory(targetObject)

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager::class.java)

// you can also add existing aspect instances, the type of the object supplied must be
// an @AspectJ aspect
factory.addAspect(usageTracker)

// now get the proxy object...
val proxy = factory.getProxy<Any>()
```

查看 [Javadoc](#) 获取更多信息

5.10. 使用 Spring 应用的 AspectJ

到目前为止，本章介绍的所有内容都是纯 Spring AOP。在本节中，我们将研究如果您的需求超出了 Spring AOP 所提供的功能，那么如何使用 AspectJ 编译器或编织器代替 Spring AOP 或除 Spring AOP 之外使用。

Spring 附带了一个小的 AspectJ 方面库，该库在您的发行版中可以作为 `spring-aspects.jar` 独立使用。您需要将其添加到类路径中才能使用其中的切面。[对 Spring 的依赖域对象使用 AspectJ 和配合 AspectJ 的其他 Spring 切面](#)讨论了该库的内容以及如何使用它。[使用 Spring IoC 配置 AspectJ 方面](#)讨论了如何依赖注入使用 AspectJ 编译器编织的 AspectJ 方面。最后，[Spring Framework 中使用 AspectJ 进行的加载时编织](#)为使用 AspectJ 的 Spring 应用程序提供了加载时编织的介绍。

5.10.1. 对 Spring 的依赖域对象使用 AspectJ

Spring 容器实例化并配置在您的应用程序上下文中定义的 bean。给定包含要应用的配置的 Bean 定义的名称，也可以要求 Bean 工厂配置预先存在的对象。`spring-aspects.jar` 包含一个注解驱动切面，该方面利用此功能允许依赖项注入任何对象。该支持旨在用于在任何容器的控制范围之外创建的对象。域对象通常属于此类，因为它们通常是通过数据库查询的结果，使用 `new` 操作符或 ORM 工具以编程方式创建的。

`@Configurable` 注解将一个类标记为符合 Spring 驱动的配置。在最简单的情况下，您可以将其纯粹用作标记注解，如以下示例所示：

Java

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configuration
public class Account {
    // ...
}
```

Kotlin

```
package com.xyz.myapp.domain

import org.springframework.beans.factory.annotation.Configurable

@Configuration
class Account {
    // ...
}
```

当以这种方式用作标记接口时，Spring 通过使用具有与完全限定类型名称 (`com.xyz.myapp.domain.Account`) 相同名称的 bean 定义（通常为原型作用域）来配置带注解类型的新实例（在这种情况下为 `Account`）。由于 bean 的默认名称是其类型的全限

定名，因此声明原型定义的便捷方法是省略 id 属性，如以下示例所示：

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
    <property name="fundsTransferService" ref="fundsTransferService"/>
</bean>
```

如果要显式指定要使用的原型 bean 定义的名称，则可以直接在注解中这样做，如以下示例所示：

Java

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configuration("account")
public class Account {
    // ...
}
```

Kotlin

```
package com.xyz.myapp.domain

import org.springframework.beans.factory.annotation.Configurable

@Configuration("account")
class Account {
    // ...
}
```

Spring 现在寻找一个名为 account 的 bean 定义，并将其用作配置新 Account 实例的定义。

您也可以使用自动装配来避免完全指定专用的 bean 定义。要让 Spring 应用自动装配，请使用 @Configurable 注解的 autowire 属性。您可以指定 @Configurable (autowire = Autowire.BY_TYPE) 或 @Configurable (autowire = Autowire.BY_NAME) 分别按类型或名称进行自动装配。或者，最好在字段或方法层面通过 @Autowired 或 @Inject 为 @Configurable bean 指定显式的，注解驱动的依赖项注入（有关更多详细信息，请参见 [基于注解的容器配置\(1.9\)](#)）。

最后，您可以使用 dependencyCheck 属性（例如，@Configurable(autowire = Autowire.BY_NAME, dependencyCheck=true)）为新创建和配置的对象中的对象引用启用 Spring 依赖项检查。如果将此属性设置为 true，则 Spring 在配置后验证是否已设置所有属性（不是基元或集合）。

请注意，单独使用注解不会执行任何操作。`spring-aspects.jar` 中的 `AnnotationBeanConfigurerAspect` 对注解的存在起作用。从本质上讲，该切面说：“从带有`@Configurable` 注解的类型的新的对象的初始化返回之后，使用 Spring 根据注解的属性配置新创建的对象”。在这种情况下，“初始化”是指新实例化的对象（例如，使用 `new` 运算符实例化的对象）以及正在进行反序列化（例如，通过 `readResolve()`）的 `Serializable` 对象。

上段中的关键短语之一是“本质上”。在大多数情况下，“从新对象的初始化返回之后”的确切语义是可以的。在这种情况下，“初始化之后”是指在构造对象之后注入依赖项。这意味着该依赖项不可在类的构造函数中使用。如果您希望在构造函数主体运行之前注入依赖项，从而可以在构造函数主体中使用这些依赖项，则需要在`@Configurable` 声明中对此进行定义，如下所示：



Java

```
@Configurable(preConstruction = true)
```

Kotlin

```
@Configurable(preConstruction = true)
```

您可以在《AspectJ 编程指南》的附录中找到有关 AspectJ 中各种切入点类型的语言语义的更多信息。

为此，必须将带注解的类型与 AspectJ 编织器编织在一起。您可以使用构建时的 Ant 或 Maven 任务来执行此操作（例如，参见《AspectJ 开发环境指南》），也可以使用加载时编织（请参见 Spring Framework 中的使用 AspectJ 进行加载时编织）。Spring 需要配置 `AnnotationBeanConfigurerAspect` 本身（以便获得对将用于配置新对象的 bean 工厂的引用）。如果使用基于 Java 的配置，则可以将`@EnableSpringConfigured` 添加到任何`@Configuration` 类中，如下所示：

Java

```
@Configuration  
@EnableSpringConfigured  
public class AppConfig {  
}
```

Kotlin

```
@Configuration  
@EnableSpringConfigured  
class AppConfig {  
}
```

如果您更喜欢基于 XML 的配置，则 Spring [context 名称空间\(9.1.3\)](#) 定义了一个方便的 `context: spring-configured` 元素，您可以按以下方式使用它：

```
<context:spring-configured/>
```

在配置方面之前创建的 `@Configurable` 对象实例会导致向调试日志发出消息，并且未进行对象配置。一个示例可能是 Spring 配置中的 `bean`，当它由 Spring 初始化时会创建域对象。在这种情况下，您可以使用 `depends-on` `bean` 属性来手动指定 `bean` 取决于配置切面。下面的示例显示如何使用 `depends-on` 属性：

```
<bean id="myService"  
      class="com.xzy.myapp.service.MyService"  
      depends-  
      on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">  
  
    <!-- ... -->  
  
</bean>
```



除非您真的想在运行时依赖它的语义，否则不要通过 `bean` 配置器方面激活 `@Configurable` 处理。特别是，请确保不要在通过容器注册为常规 Spring `bean` 的 `bean` 类上使用 `@Configurable`。这样做会导致两次初始化，一次遍历容器，一次遍历切面。

单元测试`@Configuration` 对象

`@Configurable` 支持的目标之一是实现域对象的独立单元测试，而不会遇到与硬编码查找相关的困难。如果 AspectJ 尚未编织 `@Configurable` 类型，则注解在单元测试期间不起作用。您可以在被测对象中设置 `mock` 或 `stub` 属性引用，然后照常进行。如果 AspectJ 编织了 `@Configurable` 类型，您仍然可以像往常一样在容器外部进行单元测试，但是每次构造 `@Configurable` 对象时，您都会看到一条警告消息，指示该对象尚未由 Spring 配置。

多应用上下文协同工作

用于实现`@Configurable` 支持的 `AnnotationBeanConfigurerAspect` 是 AspectJ 单例切面。单例切面的作用域与静态成员的作用域相同：每个类加载器都有一个切面实例来定义类型。这意味着，如果在相同的类加载器层次结构中定义多个应用程序上下文，则需要考虑`@EnableSpringConfigured` bean 和将 `spring-aspects.jar` 放置在类路径上的位置。

考虑一个典型的 Spring Web 应用程序配置，该配置具有一个共享的父应用程序上下文，该上下文定义了通用的业务服务，支持那些服务所需的一切，以及每个 `Servlet` 的一个子应用程序上下文（其中包含该 `Servlet` 的特定定义）。所有这些上下文共存于同一类加载器层次结构中，因此 `AnnotationBeanConfigurerAspect` 只能保存对其中一个的引用。在这种情况下，我们建议在共享（父）应用程序上下文中定义`@EnableSpringConfigured` bean。这定义了您可能想注入域对象的服务。结果是，您无法使用`@Configurable` 机制来配置域对象，而该域对象引用的是在子（特定于 `Servlet` 的）上下文中定义的 bean 的引用（无论如何，这可能都不是您想做的事情）。

在同一容器中部署多个 Web 应用程序时，请确保每个 Web 应用程序通过使用其自己的类加载器（例如，将 `spring-aspects.jar` 放置在“WEB-INF/lib”中）在 `spring-aspects.jar` 中加载类型。如果将 `spring-aspects.jar` 仅添加到容器范围的类路径中（并因此由共享的父类加载器加载），则所有 Web 应用程序都共享相同的切面实例（可能不是您想要的）。

5.10.2. 配合 AspectJ 的其他 Spring 切面

除了`@Configurable` 切面，`spring-aspects.jar` 还包含一个 AspectJ 切面，您可以使用该切面来驱动 Spring 的事务管理，以使用`@Transactional` 注解来注解类型和方法。这主要用于希望在 Spring 容器之外使用 Spring Framework 的事务支持的用户。

解释`@Transactional` 注解的切面是 `AnnotationTransactionAspect`。使用此切面时，必须注解实现类（或该类中的方法或两者），而不是注解该类所实现的接口（如果有）。AspectJ 遵循 Java 的规则，即不继承接口上的注释。

类上的`@Transactional` 注解指定用于执行该类中任何公共操作的默认事务语义。类中方法上的`@Transactional` 注解会覆盖类注解（如果存在）给出的默认事务语义。可以标注任何可见性的方法，包括私有方法。直接注解非公共方法是执行此类方法而获得事务划分的唯一方法。



从 Spring Framework 4.2 开始，spring-aspects 提供了一个相似的切面，为标准 javax.transaction.Transactional 注释提供了完全相同的功能。有关更多详细信息，请参见 [JtaAnnotationTransactionAspect](#)。

对于希望使用 Spring 配置和事务管理支持但又不想(或不能)使用注解的 AspectJ 程序员，spring-aspects.jar 还包含抽象切面，您可以扩展它们以提供自己的切入点定义。有关更多信息，请参见 AbstractBeanConfigurerAspect 和 AbstractTransactionAspect 的源文件。例如，以下摘录显示了如何编写方面来使用与完全限定的类名匹配的原型 Bean 定义来配置域模型中定义的对象的所有实例：

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {  
  
    public DomainObjectConfiguration() {  
        setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());  
    }  
  
    // the creation of a new bean (any object in the domain model)  
    protected pointcut beanCreation(Object beanInstance) :  
        initialization(new(..)) &&  
        CommonPointcuts.inDomainModel() &&  
        this(beanInstance);  
}
```

5.10.3. 通过使用 Spring IOC 配置 AspectJ 切面

当您将 AspectJ 方面与 Spring 应用程序一起使用时，既自然又希望能够使用 Spring 配置这些切面。AspectJ 运行时本身负责切面的创建，并且通过 Spring 配置 AspectJ 创建的切面的方式取决于方面所使用的 AspectJ 实例化模型（per-xxx 子句）。

AspectJ 的大多数方面都是单例切面。这些切面的配置很容易。您可以创建一个 bean 定义，该 bean 定义按常规引用方面类型，并包括 factorymethod = “aspectOf” bean 属性。在 Spring 架构中应用 AspectJ 的加载时织入。这可以确保 Spring 通过向 AspectJ 索要切面实例，而不是尝试自己创建实例来获得切面实例。以下示例显示如何使用 factory-method = “ aspectOf” 属性：

```
<bean id="profiler" class="com.xyz.profiler.Profiler"  
      factory-method="aspectOf"> ①  
  
    <property name="profilingStrategy" ref="jamonProfilingStrategy"/>  
</bean>
```

① 注意 `factory-method =“aspectOf”` 属性

非单例切面很难配置。但是，可以通过创建原型 Bean 定义并使用 `spring-aspects.jar` 中的`@Configurable` 支持来实现，一旦它们由 AspectJ 运行时创建了 Bean，就可以配置切面实例。

如果您有一些要与 AspectJ 编织的`@AspectJ` 切面（例如，对域模型类型使用加载时编织）以及要与 Spring AOP 结合使用的其他`@AspectJ` 切面，并且这些切面都已在 Spring 中配置，需要告诉 Spring AOP `@AspectJ` 自动代理支持，应使用配置中定义的`@AspectJ` 方面的确切子集进行自动代理。您可以通过在`<aop:aspectj-autoproxy/>` 声明中使用一个或多个`<include/>` 元素来执行此操作。每个`<include/>` 元素都指定一个名称模式，并且只有名称与至少一个模式匹配的 bean 才可用于 Spring AOP 自动代理配置。以下示例显示了如何使用`<include/>` 元素：

```
<aop:aspectj-autoproxy>
    <aop:include name="thisBean"/>
    <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

不要被`<aop:aspectj-autoproxy/>`元素的名称所迷惑。使用它可以创建 Spring AOP 代理。此处使用了`@AspectJ` 样式的声明，但是不涉及 AspectJ 运行时。

5.10.4. 在 Spring Framework 中使用 AspectJ 进行加载时编织

加载时编织(LTW)是指将 AspectJ 方面加载到应用程序的类文件中时将其编织到 Java 虚拟机(JVM)中的过程。本部分的重点是在 Spring 框架的特定上下文中配置和使用 LTW。本节不是 LTW 的总的介绍。有关 LTW 的详细信息以及仅使用 AspectJ 配置 LTW(完全不涉及 Spring)的详细信息，请参阅《AspectJ 开发环境指南》的 LTW 部分。

Spring 框架为 AspectJ LTW 带来的价值在于能够对编织过程进行更精细的控制。“Vanilla” AspectJ LTW 是通过使用 Java(5+) 代理来实现的，该代理在启动 JVM 时通过指定 VM 参数来打开。因此，它是 JVM 范围的设置，在某些情况下可能很好，但通常有点过于粗糙。启用 Spring 的 LTW 可让您基于每个类加载器开启 LTW，它更细粒度，并且在“单个 JVM- 多应用程序”环境(例如在典型的应用程序服务器中发现)中更有意义。

此外，在某些环境中，此需要添加`-javaagent: path/to/aspectjweaver.jar` 或(如本节后面所述)`-javaagent: path/to/spring-instrument.jar` 来支持可实现加载时编

织，而无需对应用程序服务器的启动脚本进行任何修改。开发人员将应用程序上下文配置为启用加载时编织，而不是依赖通常负责部署配置（例如启动脚本）的管理员。现在，销售工作已经结束，让我们首先浏览一个使用 Spring 的 AspectJ LTW 的快速示例，然后详细介绍示例中引入的元素。有关完整的示例，请参见 [Petclinic 示例应用程序](#)。

第一个例子

假设您是一位负责诊断系统中某些性能问题的原因的应用程序开发人员。与其使用分析工具，不如使用一个简单的分析切面，使我们能够快速获得一些性能指标。然后，我们可以立即在该特定区域应用更细粒度的分析工具。

此处提供的示例使用 XML 配置。 您还可以配置@AspectJ 并将其与 Java 配置一起使用。 具体来说，您可以使用@EnableLoadTimeWeaving 注解作为<context:load-timeweaver/>的替代方法（有关详细信息，请参见下文）。

下面的示例显示了配置方面的信息，这并不理想。 这是一个基于时间的探查器，它使用@AspectJ 样式的切面声明：

Java

```
package foo;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.util.StopWatch;
import org.springframework.core.annotation.Order;

@Aspect
public class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    public Object profile(ProceedingJoinPoint pjp) throws Throwable {
        StopWatch sw = new StopWatch(getClass().getSimpleName());
        try {
            sw.start(pjp.getSignature().getName());
            return pjp.proceed();
        } finally {
            sw.stop();
            System.out.println(sw.prettyPrint());
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    public void methodsToBeProfiled(){}
}
```

Kotlin

```
package foo

import org.aspectj.lang.ProceedingJoinPoint
import org.aspectj.lang.annotation.Aspect
import org.aspectj.lang.annotation.Around
import org.aspectj.lang.annotation.Pointcut
import org.springframework.util.StopWatch
import org.springframework.core.annotation.Order

@Aspect
class ProfilingAspect {

    @Around("methodsToBeProfiled()")
    fun profile(pjp: ProceedingJoinPoint): Any {
        val sw = StopWatch(javaClass.simpleName)
        try {
            sw.start(pjp.getSignature().getName())
            return pjp.proceed()
        } finally {
            sw.stop()
            println(sw.prettyPrint())
        }
    }

    @Pointcut("execution(public * foo..*.*(..))")
    fun methodsToBeProfiled() {
    }
}
```

我们还需要创建一个 `META-INF/aop.xml` 文件，以通知 AspectJ 编织器我们希望将 `ProfilingAspect` 编织到类中。此文件约定，即在 Java 类路径上称为 `META-INF/aop.xml` 的一个或多个文件的存在，是标准的 AspectJ。以下示例显示 `aop.xml` 文件：

```

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
 "https://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>

    <weaver>
        <!-- only weave classes in our application-specific packages -->
        <include within="foo.*"/>
    </weaver>

    <aspects>
        <!-- weave in just this aspect -->
        <aspect name="foo.ProfilingAspect"/>
    </aspects>

</aspectj>

```

现在，我们可以继续进行配置中特定于 Spring 的部分。我们需要配置一个 LoadTimeWeaver（稍后说明）。加载时编织器是必不可少的组件，负责将一个或多个 META-INF/aop.xml 文件中的切面配置编织到应用程序的类中。好处是，它不需要很多配置（您可以指定一些其他选项，但是稍后会详细介绍），如以下示例所示：

```

<?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">

    <!-- a service object; we will be profiling its methods -->
    <bean id="entitlementCalculationService"
        class="foo.StubEntitlementCalculationService"/>

    <!-- this switches on the load-time weaving -->
    <context:load-time-weaver/>
</beans>

```

现在，所有必需的工作（切面， META-INF/aop.xml 文件和 Spring 配置）均已就绪，我们可以使用 `main(..)` 方法创建以下驱动程序类，以演示实际的 LTW：

Java

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml",
Main.class);

        EntitlementCalculationService entitlementCalculationService =
            (EntitlementCalculationService)
ctx.getBean("entitlementCalculationService");

        // the profiling aspect is 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

Kotlin

```
package foo

import org.springframework.context.support.ClassPathXmlApplicationContext

fun main() {
    val ctx = ClassPathXmlApplicationContext("beans.xml")

    val entitlementCalculationService = ctx.getBean("entitlementCalculationService")
as EntitlementCalculationService

    // the profiling aspect is 'woven' around this method execution
    entitlementCalculationService.calculateEntitlement()
}
```

我们还有最后一件事要做。本节的引言确实说过，可以使用 Spring 在每个类加载器的基础上选择性地打开 LTW，这是事实。但是，对此示例，我们使用 Java 代理（Spring 提供）打开 LTW。我们使用以下命令运行前面显示的 Main 类：

```
java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main
```

-javaagent 是一个标志，用于指定和启用代理以对在 JVM 上运行的程序进行检测。Spring 框架附带了这样的代理工具 `InstrumentationSavingAgent`，该代理文件打包在 `spring-instrument.jar` 中，在上一示例中作为 `-javaagent` 参数的值提供。

执行主程序的输出类似于下一个示例。我已将 `Thread.sleep(..)` 语句引入了 `calculateEntitlement()` 实现中，因此探查器实际上捕获了 0 毫秒以外的内容（01234

毫秒不是 AOP 引入的开销）。以下清单显示了运行分析器时得到的输出：

```
Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234
-----
ms      %      Task name
-----
01234  100%  calculateEntitlement
```

由于此 LTW 是通过使用成熟的 AspectJ 来实现的，因此我们不仅限于通知 Spring Bean。在 Main 程序上进行以下细微改动会产生相同的结果：

Java

```
package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

    public static void main(String[] args) {
        new ClassPathXmlApplicationContext("beans.xml", Main.class);

        EntitlementCalculationService entitlementCalculationService =
            new StubEntitlementCalculationService();

        // the profiling aspect will be 'woven' around this method execution
        entitlementCalculationService.calculateEntitlement();
    }
}
```

```

package foo

import org.springframework.context.support.ClassPathXmlApplicationContext

fun main(args: Array<String>) {
    ClassPathXmlApplicationContext("beans.xml")

    val entitlementCalculationService = StubEntitlementCalculationService()

    // the profiling aspect will be 'woven' around this method execution
    entitlementCalculationService.calculateEntitlement()
}

```

请注意，在前面的程序中，我们如何引导 Spring 容器，然后完全在 Spring 上下文之外创建 `StubEntitlementCalculationService` 的新实例。分析通知仍会被应用。

诚然，这个例子很简单。但是，在较早的示例中已经介绍了 Spring 对 LTW 支持的基础，本节的其余部分详细解释了每个配置和用法背后的“原因”。

在此示例中使用的 `ProfilingAspect` 可能是基本的，但它非常有用。



这是开发时方面的一个很好的示例，开发人员可以在开发过程中使用它，然后轻松地将其从部署到 UAT 或生产中的应用程序构建中排除。

切面

您在 LTW 中使用的切面必须是 AspectJ 切面。您可以使用 AspectJ 语言本身来编写它们，也可以使用@AspectJ 风格来编写切面。这样，您的切面就是有效的 AspectJ 和 Spring AOP 切面。此外，编译的切面类需要在类路径上可用。

‘META-INF/aop.xml’

通过使用 Java 类路径上的一个或多个 `META-INF / aop.xml` 文件（直接或通常在 jar 文件中）来配置 AspectJ LTW 基础结构。

该文件的结构和内容在 [AspectJ 参考文档](#) 的 LTW 部分中进行了详细说明。由于 `aop.xml` 文件是 100% AspectJ，因此在此不再赘述。

需求库 (JARS)

至少，您需要使用以下库来使用 Spring Framework 对 AspectJ LTW 的支持

- `spring-aop.jar`
- `aspectjweaver.jar`

如果使用 [Spring 提供的代理来启用检测](#)，则还需要：

- `spring-instrument.jar`

Spring 配置

Spring 的 LTW 支持的关键组件是 `LoadTimeWeaver` 接口（在 `org.springframework.instrument.classloading` 包中），以及 Spring 发行版附带的众多实现。`LoadTimeWeaver` 负责在运行时将一个或多个 `java.lang.instrument.ClassFileTransformers` 添加到类加载器，这为各种有趣的应用程序打开了大门，其中之一就是方面的 LTW。



如果您不熟悉运行时类文件转换的概念，请在继续操作之前参阅 `java.lang.instrument` 软件包的 javadoc API 文档。虽然该文档并不全面，但是至少您可以看到关键的接口和类（在您阅读本节时作为参考）。

为特定的 `ApplicationContext` 配置 `LoadTimeWeaver` 就像添加一行代码一样容易。（请注意，您几乎可以肯定需要将 `ApplicationContext` 用作 Spring 容器 - 通常，仅 `BeanFactory` 是不够的，因为 LTW 支持使用 `BeanFactoryPostProcessors`。）

要启用 Spring Framework 的 LTW 支持，您需要配置一个 `LoadTimeWeaver`，通常通过使用`@EnableLoadTimeWeaving` 注解来完成，如下所示：

Java

```
@Configuration  
@EnableLoadTimeWeaving  
public class AppConfig {  
}
```

Kotlin

```
@Configuration  
@EnableLoadTimeWeaving  
class AppConfig {  
}
```

另外，如果您喜欢基于 XML 的配置，请使用`<context:load-time-weaver/>`元素。注意，该元素是在上下文名称空间中定义的。以下示例显示了如何使用`<context:load-time-weaver/>`：

```

<?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:load-time-weaver/>

</beans>

```

前面的配置会自动为您定义并注册许多 LTW 特定的基础结构 Bean，例如 `LoadTimeWeaver` 和 `AspectJWeavingEnabler`。默认的 `LoadTimeWeaver` 是 `DefaultContextLoadTimeWeaver` 类，它试图装饰自动检测到的 `LoadTimeWeaver`。“自动检测”的 `LoadTimeWeaver` 的确切类型取决于您的运行时环境。下表总结了各种 `LoadTimeWeaver` 实现：

表 13DefaultContextLoadTimeWeaver LoadTimeWeavers

运行时环境	<code>LoadTimeWeaver</code> 实现
运行在 Apache Tomcat	<code>TomcatLoadTimeWeaver</code>
运行在 GlassFish(限制 EAR 部署)	<code>GlassFishLoadTimeWeaver</code>
运行在 RedHat 的 JBoss AS 或者 WildFly	<code>JBossLoadTimeWeaver</code>
运行在 IBM 的 WebSphere	<code>WebSphereLoadTimeWeaver</code>
运行在 Oracle 的 Weblogic	<code>WebLogicLoadTimeWeaver</code>
使用 Spring 启动 JVM <code>InstrumentationSavingAgent(java -javaagent:path/to/spring-instrument.jar)</code>	<code>InstrumentationLoadTimeWeaver</code>
回退，期望基础类加载器遵循通用约定(即 <code>addTransformer</code> 和 可选的 <code>getThrowawayClassLoader</code> 方法)	<code>ReflectiveLoadTimeWeaver</code>

请注意，该表仅列出使用 `DefaultContextLoadTimeWeaver` 时自动检测到的 `LoadTimeWeaver`。您可以确切指定要使用的 `LoadTimeWeaver` 实现。

要使用 Java 配置指定特定的 `LoadTimeWeaver`，请实现

LoadTimeWeavingConfigurer 接口并重写 getLoadTimeWeaver () 方法。以下示例指定了 ReflectiveLoadTimeWeaver：

Java

```
@Configuration  
@EnableLoadTimeWeaving  
public class AppConfig implements LoadTimeWeavingConfigurer {  
  
    @Override  
    public LoadTimeWeaver getLoadTimeWeaver() {  
        return new ReflectiveLoadTimeWeaver();  
    }  
}
```

Kotlin

```
@Configuration  
@EnableLoadTimeWeaving  
class AppConfig : LoadTimeWeavingConfigurer {  
  
    override fun getLoadTimeWeaver(): LoadTimeWeaver {  
        return ReflectiveLoadTimeWeaver()  
    }  
}
```

如果使用基于 XML 的配置，则可以在<context:load-time-weaver/>元素上将完全限定的类名指定为 weaver-class 属性的值。同样，以下示例指定了 ReflectiveLoadTimeWeaver：

```
<?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:load-time-weaver  
        weaver-  
        class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>  
  
</beans>
```

稍后可以使用众所周知的名称 loadTimeWeaver 从 Spring 容器中检索由配置定义和注册的 LoadTimeWeaver。请记住，LoadTimeWeaver 仅作为 Spring LTW 基础结构添加一个或多个 ClassFileTransformers 的机制而存在。执行 LTW 的实际 ClassFileTransformer

是 `ClassPreProcessorAgentAdapter`(来自 `org.aspectj.weaver.loadtime` 包)类。有关更多详细信息，请参见 `ClassPreProcessorAgentAdapter` 类的类级 javadoc，因为实际上如何实现编织的细节不在本文档的讨论范围之内。

还需要讨论配置的最后一个属性：`aspectjWeaving` 属性（如果使用 XML，则为 `Aspectj-weaving`）。此属性控制是否启用 LTW。它接受三个可能的值之一，如果不存在该属性，则默认值为自动检测。下表总结了三个可能的值：

表 14: Aspect weaving 属性值

注解值	XML 值	解释
<code>ENABLED</code>	<code>on</code>	Aspect weaving 开启，切面在运行时适当时机被编织
<code>DISABLED</code>	<code>off</code>	LTW 关闭。没有切面在运行时被编织
<code>AUTODETECT</code>	<code>autodetect</code>	如果 Spring LTW 基础结构可以找到至少一个 <code>META-INF/aop.xml</code> 文件，则 AspectJ 编织已启动。否则默认关闭。

特定环境配置

最后一部分包含在应用程序服务器和 Web 容器等环境中使用 Spring 的 LTW 支持时所需的任何其他设置和配置。

Tomcat, JBoss, WebSphere, WebLogic

Tomcat, JBoss / WildFly, IBM WebSphere Application Server 和 Oracle WebLogic Server 都提供了能够在本地进行检测的通用应用程序类加载器。Spring 的原生 LTW 可以利用这些类加载器实现来提供 AspectJ 编织。如前所述，您可以简单地启用加载时编织。具体来说，您无需修改 JVM 启动脚本即可添加 `-javaagent:path/to/spring-instrument.jar`。

请注意，在 JBoss 上，您可能需要禁用应用程序服务器扫描，以防止它在应用程序实际启动之前加载类。一个快速的解决方法是将一个名为 `WEB-INF/jboss-scanning.xml` 的文件添加到您的工件中，其中包含以下内容：

```
<scanning xmlns="urn:jboss:scanning:1.0"/>
```

通用 Java 应用程序

如果特定 `LoadTimeWeaver` 实现不支持的环境中需要类检测，则 JVM 代理是通用解决方案。对于这种情况，Spring 提供了 `InstrumentationLoadTimeWeaver`，它需要特定于 Spring（但非常通用）的 JVM 代理 `spring-instrument.jar`，并通过常见的 `@EnableLoadTimeWeaving` 和`<context:load-time-weaver/>`设置自动检测到。

要使用它，您必须通过提供以下 JVM 选项来使用 Spring 代理启动虚拟机：

```
-javaagent:/path/to/spring-instrument.jar
```

请注意，这需要修改 JVM 启动脚本，这可能会阻止您在应用程序服务器环境中使用它（取决于您的服务器和您的操作策略）。就是说，对于每个 JVM 一个应用程序的部署（例如独立的 Spring Boot 应用程序），无论如何通常都可以控制整个 JVM 的设置。

5.11. 更多资源

可以在 [AspectJ 网站](#) 上找到有关 AspectJ 的更多信息。

Eclipse AspectJ, 作者: Adrian Colyer 等 (Addison-Wesley, 2005 年) 为 AspectJ 语言提供了全面的介绍和参考。

强烈推荐 Ramnivas Laddad 撰写的《AspectJ in Action》第二版 (Manning, 2009 年)。本书的重点是 AspectJ，但在一定程度上探讨了许多通用的 AOP 主题。

6. Spring AOP APIs

上一章通过@AspectJ 和基于架构的方面定义描述了 Spring 对 AOP 的支持。在本章中，我们讨论了较低级别的 Spring AOP API。对于常见的应用程序，我们建议将 Spring AOP 与 AspectJ 切入点一起使用，如上一章所述。

6.1. Spring 里的 Pointcut(切入点) API

本节描述了 Spring 如何处理关键切入点概念。

6.1.1. 概念

Spring 的切入点模型使切入点重用不受通知类型的影响。您可以使用相同的切入点来定位不同的通知。

`org.springframework.aop.Pointcut` 接口是中央接口，用于将通知定向到特定的类和方法。完整的界面如下：

Java

```
public interface Pointcut {  
    ClassFilter getClassFilter();  
    MethodMatcher getMethodMatcher();  
}
```

Kotlin

```
interface Pointcut {  
    fun getClassFilter(): ClassFilter  
    fun getMethodMatcher(): MethodMatcher  
}
```

将 `Pointcut` 接口分为两部分，可以重用类和方法匹配的部分以及细粒度的合成操作（例如与另一个方法匹配器执行“联合”）。

`ClassFilter` 接口用于将切入点限制为给定的一组目标类。如果 `matches()` 方法始终返回 `true`，则将匹配所有目标类。以下清单显示了 `ClassFilter` 接口定义：

Java

```
public interface ClassFilter {  
    boolean matches(Class clazz);  
}
```

Kotlin

```
interface ClassFilter {  
    fun matches(clazz: Class<*>): Boolean  
}
```

MethodMatcher 接口通常更重要。 完整的接口如下：

Java

```
public interface MethodMatcher {  
    boolean matches(Method m, Class targetClass);  
    boolean isRuntime();  
    boolean matches(Method m, Class targetClass, Object[] args);  
}
```

Kotlin

```
interface MethodMatcher {  
    val isRuntime: Boolean  
    fun matches(m: Method, targetClass: Class<*>): Boolean  
    fun matches(m: Method, targetClass: Class<*>, args: Array<Any>): Boolean  
}
```

matchs(Method,Class) 方法用于测试此切入点是否与目标类上的给定方法匹配。创建 AOP 代理时可以执行此评估，以避免需要对每个方法调用进行测试。如果两个参数的 **match** 方法对于给定的方法返回 **true**，并且 **MethodMatcher** 的 **isRuntime()** 方法返回 **true**，则在每次方法调用时都将调用三个参数的 **match** 方法。这使切入点可以在目标通知开始之前立即查看传递给方法调用的参数。

大多数 **MethodMatcher** 实现都是静态的，这意味着它们的 **isRuntime()** 方法返回 **false**。在这种情况下，永远不会调用三参数匹配方法。



如果可能，尝试让切入点是静态的，当 AOP 代理创建的时候允许 AOP 框架缓存切入点评估的结果。

6.1.2. 切入点上的操作

Spring 支持切入点上的操作（特别是联合和相交）。

联合表示两个切入点匹配其中一个的方法。 交集是指两个切入点都匹配的方法。 联合有时候更有用。您可以通过使用 `org.springframework.aop.support.Pointcuts` 类中的静态方法或在同一包中使用 `ComposablePointcut` 类来组成切入点。但是，使用 AspectJ 切入点表达式通常是一种更简单的方法。

6.1.3. AspectJ 表达式切入点

从 2.0 开始，Spring 使用的最重要的切入点类型是 `org.springframework.aop.aspectj.AspectJExpressionPointcut`。这是使用 `AspectJsupplied` 库来解析 AspectJ 切入点表达式字符串的切入点。

有关支持的 AspectJ 切入点原语的讨论，请参见上一章。

6.1.4. 简易切入点实现

Spring 提供了几种方便的切入点实现。您可以直接使用其中一些，其他则应在特定于应用程序的切入点中进行子类化。

静态切入点

静态切入点基于方法和目标类，并且不能考虑方法的参数。静态切入点在更多使用中都是足够而且是最好的。首次调用方法时，Spring 只能评估一次静态切入点。之后，无需在每次方法调用时再次评估切入点。

本节的其余部分描述了 Spring 附带的一些静态切入点实现。

常规表达式切入点

指定静态切入点的一种明显方法是正则表达式。除了 Spring 之外，还有几个 AOP 框架使之成为可能。`org.springframework.aop.support.JdkRegexpMethodPointcut` 是一个通用的正则表达式切入点，它使用了 JDK 中的正则表达式支持。

使用 `JdkRegexpMethodPointcut` 类，可以提供模式字符串的列表。如果其中任何一个匹配，则切入点的评估结果为 `true`。（因此，结果实际上是这些切入点的并集。）

以下示例显示如何使用 `JdkRegexpMethodPointcut`：

```
<bean id="settersAndAbsquatulatePointcut"
      class="org.springframework.aop.support.JdkRegexpMethodPointcut">
    <property name="patterns">
      <list>
        <value>.*set.*</value>
        <value>.*absquatulate</value>
      </list>
    </property>
</bean>
```

Spring 提供了一个名为 `RegexpMethodPointcutAdvisor` 的便捷类，该类使我们也可以引用一个 `Advice`（请记住，`Advice` 可以是拦截器，前置通知，抛出通知）。在后台，Spring 使用了 `JdkRegexpMethodPointcut`。使用 `RegexpMethodPointcutAdvisor` 可以简化装配，因为一个 `bean` 封装了切入点和通知，如以下示例所示：

```
<bean id="settersAndAbsquatulateAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
      <ref bean="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
      <list>
        <value>.*set.*</value>
        <value>.*absquatulate</value>
      </list>
    </property>
</bean>
```

您可以将 `RegexpMethodPointcutAdvisor` 与任何通知类型一起使用。

属性驱动切入点

静态切入点的一种重要类型是元数据驱动的切入点。这使用元数据属性的值（通常是一源级别的元数据）。

动态切入点

动态切入点比静态切入点更费事。它们考虑了方法参数以及静态信息。这意味着必须在每次方法调用时对它们进行评估，并且由于参数会有所不同，因此无法缓存结果。

主要示例是 `control flow` 切入点。

`control flow` 切入点

Spring 控制流切入点在概念上类似于 AspectJ `cflow` 切入点，但功能较弱。（当前无法指定切入点在与另一个切入点匹配的连接点下运行。）控制流切入点与当前调用堆栈匹配。例如，如果连接点是由 `com.mycompany.web` 包中的方法或 `SomeCaller` 类调用的，

则可能会触发。通过使用 `org.springframework.aop.support.ControlFlowPointcut` 类指定控制流切入点。



与其他动态切入点相比，控制流切入点在运行时进行评估耗费巨大在 Java 1.4 中，成本大约是其他动态切入点的五倍。

6.1.5. 切入点超类

Spring 提供了有用的切入点超类，以帮助您实现自己的切入点。因为静态切入点最有用，所以您可能应该子类化 `StaticMethodMatcherPointcut`。这仅需要实现一个抽象方法（尽管您可以覆盖其他方法以自定义行为）。下面的示例演示如何对 `StaticMethodMatcherPointcut` 进行子类化：

Java

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {  
  
    public boolean matches(Method m, Class targetClass) {  
        // return true if custom criteria match  
    }  
}
```

Kotlin

```
class TestStaticPointcut : StaticMethodMatcherPointcut() {  
  
    override fun matches(method: Method, targetClass: Class<*>): Boolean {  
        // return true if custom criteria match  
    }  
}
```

动态切入点也有超类。您可以将自定义切入点与任何通知类型一起使用。

6.1.6. 自定义切入点

由于 Spring AOP 中的切入点是 Java 类，而不是语言功能（如 AspectJ），因此可以声明自定义切入点，无论是静态还是动态。Spring 中的自定义切入点可以任意复杂。但是，如果可以的话，我们建议使用 AspectJ 切入点表达式语言。



更高版本的 Spring 可能提供对 JAC 提供的“语义切入点”的支持，例如，“更改目标对象中实例变量的所有方法”。

6.2. Spring 里的 Advice(通知)API

现在，我们可以检查 Spring AOP 如何处理通知。

6.2.1. 通知生命周期

每个通知都是一个 Spring bean。通知实例可以在所有通知对象之间共享，或者对于每个通知对象都是唯一的。这对应于每个类或每个实例的通知。

逐类通知最常用。适用于一般通知，例如事务一般切面。这些不依赖于代理对象的状态或添加新状态。它们仅作用于方法和参数。

逐实例通知都适合引入，以支持混入。在这种情况下，建议将状态添加到代理对象。

您可以在同一个 AOP 代理中混合使用共享和逐实例通知。

6.2.2. Spring 中的通知类型

Spring 提供了几种通知类型，并且可以扩展以支持任意通知类型。本节介绍基本概念和标准通知类型。

拦截环绕通知

Spring 中最基本的通知类型是环绕通知的拦截。对于使用方法拦截的建议，Spring 符合 AOP Alliance 接口。实现 `MethodInterceptor` 和环绕通知的类也应该实现以下接口：

Java

```
public interface MethodInterceptor extends Interceptor {  
    Object invoke(MethodInvocation invocation) throws Throwable;  
}
```

Kotlin

```
interface MethodInterceptor : Interceptor {  
    fun invoke(invocation: MethodInvocation) : Any  
}
```

`invoke()` 方法的 `MethodInvocation` 参数公开了正在调用的方法，目标连接点，AOP 代理以及该方法的参数。`invoke()` 方法应返回调用的结果：连接点的返回值。

以下示例显示了一个简单的 `MethodInterceptor` 实现：

Java

```
public class DebugInterceptor implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation) throws Throwable {  
        System.out.println("Before: invocation=[" + invocation + "]");  
        Object rval = invocation.proceed();  
        System.out.println("Invocation returned");  
        return rval;  
    }  
}
```

Kotlin

```
class DebugInterceptor : MethodInterceptor {  
  
    override fun invoke(invocation: MethodInvocation): Any {  
        println("Before: invocation=[${invocation}]")  
        val rval = invocation.proceed()  
        println("Invocation returned")  
        return rval  
    }  
}
```

请注意对 `MethodInvocation` 的 `proceed()` 方法的调用。这沿着拦截器链向下到达连接点。大多数拦截器都调用此方法并返回其返回值。但是，`MethodInterceptor` 像任何环绕通知一样，可以返回不同的值或引发异常，而不是调用 `proceed` 方法。但是，您没有充分的理由的话就不想这样做。

 `MethodInterceptor` 实现提供与其他符合 AOP Alliance 要求的 AOP 实现的互操作性。本节其余部分讨论的其他通知类型将实现常见的 AOP 概念，但以特定于 Spring 的方式。尽管使用最具体的通知类型有一个优势，但是如果可能想在另一个 AOP 框架中运行切面，则在环绕通知使用 `MethodInterceptor`。请注意，切入点当前无法在框架之间互操作，并且 AOP Alliance 当前未定义切入点接口。

前置通知

一种更简单的建议类型是前置通知。这不需要 `MethodInvocation` 对象，因为它仅在进入方法之前被调用。

前置通知的主要优点在于，无需调用 `proceed()` 方法，因此，不会因疏忽而未能沿拦截器链继续前进。

以下清单显示了 `MethodBeforeAdvice` 接口：

Java

```
public interface MethodBeforeAdvice extends BeforeAdvice {  
    void before(Method m, Object[] args, Object target) throws Throwable;  
}
```

Kotlin

```
interface MethodBeforeAdvice : BeforeAdvice {  
    fun before(m: Method, args: Array<Any>, target: Any)  
}
```

(尽管通常的对象适用于字段拦截，并且 Spring 不太可能实现它，但 Spring 的 API 设计允许字段前置通知。)

请注意，返回类型为 **void**。通知可以在联接点运行之前插入自定义行为，但不能更改返回值。如果之前的通知引发异常，它将停止进一步执行拦截器链。异常会传播回拦截器链。如果未检查它或在调用的方法的签名上，则将其直接传递给客户端。否则，它将被 AOP 代理包装在未经检查的异常中。

以下示例显示了 Spring 中的前置通知，该通知计算所有方法调用：

Java

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {  
  
    private int count;  
  
    public void before(Method m, Object[] args, Object target) throws Throwable {  
        ++count;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

Kotlin

```
class CountingBeforeAdvice : MethodBeforeAdvice {  
  
    var count: Int = 0  
  
    override fun before(m: Method, args: Array<Any>, target: Any?) {  
        ++count  
    }  
}
```



前置通知可以与任何切入点一起使用。

抛出通知

如果联接点引发异常，则在联接点返回之后调用抛出通知。Spring 提供类型化的抛出通知。请注意，这意味着 `org.springframework.aop.ThrowsAdvice` 接口不包含任何方法。它是一个标签接口，用于标识给定对象实现了一个或多个类型化的抛出通知方法。这些应采用以下形式：

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

仅最后一个参数是必需的。方法签名可以具有一个或四个自变量，具体取决于通知方法是否对该方法和自变量感兴趣。接下来的两个列表显示了类，它们是抛出通知的示例。

如果引发 `RemoteException`（包括从子类），则调用以下通知：

Java

```
public class RemoteThrowsAdvice implements ThrowsAdvice {  
  
    public void afterThrowing(RemoteException ex) throws Throwable {  
        // Do something with remote exception  
    }  
}
```

Kotlin

```
class RemoteThrowsAdvice : ThrowsAdvice {  
  
    fun afterThrowing(ex: RemoteException) {  
        // Do something with remote exception  
    }  
}
```

与前面的通知不同，下一个示例声明四个参数，以便可以访问被调用的方法，方法参数和目标对象。如果抛出 `ServletException`，则调用以下通知：

Java

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {  
  
    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {  
        // Do something with all arguments  
    }  
}
```

Kotlin

```
class ServletThrowsAdviceWithArguments : ThrowsAdvice {  
  
    fun afterThrowing(m: Method, args: Array<Any>, target: Any, ex: ServletException)  
    {  
        // Do something with all arguments  
    }  
}
```

最后一个示例说明如何在处理 RemoteException 和 ServletException 的单个类中使用这两种方法。可以将任意数量的抛出通知方法组合到一个类中。以下清单显示了最后一个示例：

Java

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {  
  
    public void afterThrowing(RemoteException ex) throws Throwable {  
        // Do something with remote exception  
    }  
  
    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {  
        // Do something with all arguments  
    }  
}
```

Kotlin

```
class CombinedThrowsAdvice : ThrowsAdvice {

    fun afterThrowing(ex: RemoteException) {
        // Do something with remote exception
    }

    fun afterThrowing(m: Method, args: Array<Any>, target: Any, ex: ServletException)
    {
        // Do something with all arguments
    }
}
```



如果抛出通知方法本身引发异常，则它将覆盖原始异常（也就是说，它将更改引发给用户的异常）。重写异常通常是 `RuntimeException`，它与任何方法签名都兼容。但是，如果抛出通知方法抛出一个已检查的异常，则它必须与目标方法的已声明异常匹配，因此在某种程度上与特定的目标方法签名耦合。**不要抛出与目标方法签名不兼容的未声明检查异常！**



前置通知可以与任何切入点一起使用。

后置返回通知

Spring 中的后置返回通知必须实现 `org.springframework.aop.AfterReturningAdvice` 接口，以下清单显示：

Java

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
        throws Throwable;
}
```

Kotlin

```
interface AfterReturningAdvice : Advice {

    fun afterReturning(returnValue: Any, m: Method, args: Array<Any>, target: Any)
}
```

后置返回通知可以访问返回值（无法修改），调用的方法，方法的参数和目标。

后置返回通知的将计数所有未引发异常的成功方法调用：

Java

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {  
  
    private int count;  
  
    public void afterReturning(Object returnValue, Method m, Object[] args, Object  
target)  
        throws Throwable {  
        ++count;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

Kotlin

```
class CountingAfterReturningAdvice : AfterReturningAdvice {  
  
    var count: Int = 0  
    private set  
  
    override fun afterReturning(returnValue: Any?, m: Method, args: Array<Any>,  
target: Any?) {  
        ++count  
    }  
}
```

该通知不会更改执行路径。如果抛出异常，则会将其抛出拦截器链，而不是返回值。



后置返回通知可以和任何切入点一起使用。

引介通知

Spring 将引介通知视为一种特殊的拦截通知。

引介需要实现以下接口的 `IntroductionAdvisor` 和 `IntroductionInterceptor`:

Java

```
public interface IntroductionInterceptor extends MethodInterceptor {  
    boolean implementsInterface(Class intf);  
}
```

Kotlin

```
interface IntroductionInterceptor : MethodInterceptor {  
    fun implementsInterface(intf: Class<*>): Boolean  
}
```

从 AOP Alliance `methodInterceptor` 接口继承的 `invoke()` 方法必须实现引介。也就是说，如果被调用的方法在引介接口上，则引介拦截器负责处理方法调用，不能调用 `proceed()`。

引介通知不能与任何切入点一起使用，因为它仅适用于类，而不适用于方法级别。您只能通过 `IntroductionAdvisor` 使用引介通知，它具有以下方法：

Java

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {  
    ClassFilter getClassFilter();  
  
    void validateInterfaces() throws IllegalArgumentException;  
}  
  
public interface IntroductionInfo {  
    Class<?>[] getInterfaces();  
}
```

Kotlin

```
interface IntroductionAdvisor : Advisor, IntroductionInfo {  
    val classFilter: ClassFilter  
  
    @Throws(IllegalArgumentException::class)  
    fun validateInterfaces()  
}  
  
interface IntroductionInfo {  
    val interfaces: Array<Class<*>>  
}
```

没有 `MethodMatcher`, 因此没有与引介通知相关的 `Pointcut`。只有类过滤是合乎逻辑的。

`getInterfaces()`方法返回此通知程序引介的接口。

在内部使用 `validateInterfaces()`方法来查看引入的接口是否可以由配置的 `IntroductionInterceptor` 实现。

考虑一下 Spring 测试套件中的一个示例，并假设我们想为一个或多个对象引入以下接口：

Java

```
public interface Lockable {  
    void lock();  
    void unlock();  
    boolean locked();  
}
```

Kotlin

```
interface Lockable {  
    fun lock()  
    fun unlock()  
    fun locked(): Boolean  
}
```

这说明了混入。我们希望能够将通知对象强制转换为 `Lockable`, 无论它们的类型和调用锁和解锁方法与否。如果我们调用 `lock()`方法, 我们希望所有 `setter` 方法都抛出一个 `LockedException`。因此, 我们可以添加一个切面, 使对象在不了解对象的情况下不可变: AOP 的一个很好的例子。

首先, 我们需要一个 `IntroductionInterceptor` 来完成繁重的工作。在这种情况下, 我们扩展了

`org.springframework.aop.support.DelegatingIntroductionInterceptor` 便利类。

我们可以直接实现 `IntroductionInterceptor`，但是在大多数情况下，最好使用 `DelegatingIntroductionInterceptor`。

`DelegatingIntroductionInterceptor` 旨在委派一个引介给一个引介接口的实际实现，从而隐瞒了使用拦截的方式。您可以使用构造函数参数将委托设置为任何对象。默认委托（使用无参构造函数时）是 `this`。因此，在下一个示例中，委托是 `DelegatingIntroductionInterceptor` 的 `LockMixin` 子类。给定一个委托（默认情况下为本身），`DelegatingIntroductionInterceptor` 实例将查找由委托实现的所有接口（`IntroductionInterceptor` 除外），并支持针对其中任何一个的引介。诸如 `LockMixin` 的子类可以调用 `preventInterface(Class intf)` 方法来禁止不应公开的接口。但是，无论 `IntroductionInterceptor` 准备支持多少个接口，`IntroductionAdvisor` 都会使用控件来实际公开哪些接口。引介接口隐藏了目标对同一接口的任何实现。

因此，`LockMixin` 扩展了 `DelegatingIntroductionInterceptor` 并实现了 `Lockable` 本身。超类会自动选择可支持 `Lockable` 的引入，因此我们不需要指定它。我们可以通过这种方式引入任意数量的接口。注意 `locked` 实例变量的使用。这有效地将附加状态添加到目标对象中保存的状态。

下面的示例显示示例 `LockMixin` 类：

Java

```
public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {
            throw new LockedException();
        }
        return super.invoke(invocation);
    }
}
```

Kotlin

```
class LockMixin : DelegatingIntroductionInterceptor(), Lockable {

    private var locked: Boolean = false

    fun lock() {
        this.locked = true
    }

    fun unlock() {
        this.locked = false
    }

    fun locked(): Boolean {
        return this.locked
    }

    override fun invoke(invocation: MethodInvocation): Any? {
        if (locked() && invocation.method.name.indexOf("set") == 0) {
            throw LockedException()
        }
        return super.invoke(invocation)
    }
}
```

通常，您不必重写 `invoke()` 方法。通常足以满足 `DelegatingIntroductionInterceptor` 实现（如果引入了方法，则调用委托方法，否则进行到连接点）。在当前情况下，我们需要添加一个检查：如果处于锁定模式，则不能调用任何 `setter` 方法。

所需的引介仅需要保存一个独特的 `LockMixin` 实例并指定所引入的接口（在这种情况下，仅是 `Lockable`）。一个更复杂的示例可能引用了引入拦截器（将被定义为原型）。在这种情况下，没有与 `LockMixin` 相关的配置，因此我们使用 `new` 创建它。以下示例显示了我们的 `LockMixinAdvisor` 类：

Java

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

Kotlin

```
class LockMixinAdvisor : DefaultIntroductionAdvisor(LockMixin(), Lockable::class.java)
```

我们可以非常简单地应用此一般切面，因为它不需要配置。（但是，如果没有 `IntroductionAdvisor`，则无法使用 `IntroductionInterceptor`。）像通常的引介一样，一般切面必须是逐实例的，因为它是有状态的。对于每个通知对象，我们需要一个 `LockMixinAdvisor` 实例，因此需要一个 `LockMixin` 实例。一般切面包含通知对象状态的一部分。

我们可以像其他任何一般切面一样，通过使用 `Advised.addAdvisor()` 方法或 XML 配置中的（推荐方式）以编程方式应用此一般切面。下面讨论的所有代理创建选择，包括“自动代理创建器”，都可以正确处理引介和有状态的混入。

6.3. Spring 里的 Advisor(一般切面)API

在 Spring 中，一般切面是一个切面，其中仅包含与切入点表达式关联的单个通知对象。

除了引介的特殊情况外，任何一般切面都可以与任何通知一起使用。`org.springframework.aop.support.DefaultPointcutAdvisor` 是最常用的一般切面类。它可以与 `MethodInterceptor`, `BeforeAdvice` 或 `ThrowsAdvice` 一起使用。

可以在同一 AOP 代理中的 Spring 中混合使用一般切面和通知类型。例如，您可以在一个代理配置中使用环绕通知的拦截，抛出通知以及在前置通知。Spring 自动创建必要的拦截器链。

6.4. 使用 ProxyFactoryBean(代理工厂 bean)去创建 AOP 代理

如果您将 Spring IoC 容器 (`ApplicationContext` 或 `BeanFactory`) 用于您的业务对象（应该如此！），则要使用 Spring 的 AOP `FactoryBean` 实现之一。（请记住，工厂 bean 引入了一个间接层，允许它创建不同类型的对象。）



Spring AOP 支持还在后台使用了工厂 bean。

在 Spring 中 创建 AOP 代理 的 基 本 方 法 是 使用 `org.springframework.aop.framework.ProxyFactoryBean`。这样可以完全控制切入点，任何适用的通知及其顺序。但是，如果您不需要这样的控制，则有一些更简单的选项。

6.4.1. 基础

像其他 Spring `FactoryBean` 实现一样，`ProxyFactoryBean` 引入了一个间接层。如果定义一个名为 `foo` 的 `ProxyFactoryBean`，则引用 `foo` 的对象将看不到 `ProxyFactoryBean` 实例本身，而是看到由 `ProxyFactoryBean` 中的 `getObject()` 方法的

实现创建的对象。此方法创建一个包装目标对象的 AOP 代理。

使用 `ProxyFactoryBean` 或另一个支持 IoC 的类创建 AOP 代理的最重要好处之一是，通知和切入点也可以由 IoC 管理。这是一项强大的功能，可实现某些其他 AOP 框架难以实现的方法。例如，受益于依赖注入提供的所有可插入性，通知本身可以引用应用程序对象（除了目标，目标应该在任何 AOP 框架中可用）。

6.4.2. JavaBean 属性

与 Spring 提供的大多数 `FactoryBean` 实现一样，`ProxyFactoryBean` 类本身就是 JavaBean。其属性用于：

- 指定你想代理的目标。
- 指定是否使用 CGLIB（稍后描述也可以参见[基于 JDK 和 CGLIB 的代理](#)）

一些关键属性继承自 `org.springframework.aop.framework.ProxyConfig`(spring 里所有 AOP 代理工厂的超类)。这些关键属性如下：

- `proxyTargetClass:true` 如果目标类被代理，而不是目标类的接口。如果属性值是 `true`，CGLIB 代理将创建（参见[基于 JDK 和 CGLIB 的代理](#)）。
- `optimize`: 控制是否主动优化 CGLIB 代理，除非您完全了解相关的 AOP 代理如何处理优化，否则不要随意使用此设置。仅用于 CGLIB 代理，JDK 动态代理无效。
- `frozen`: 如果一个代理配置是 `frozen`，就不允许改配置了。这对于进行轻微的优化以及在不希望调用者在创建代理后希望调用者能够(通过 `Advised` 接口)操纵代理的情况下都是有用的。默认值为 `false` 所以这个时候可更改(比如加额外通知)。
- `exposeProxy`: 确定是否应在 `ThreadLocal` 中公开当前代理，以便目标可以访问它。如果目标需要获取代理，并且暴露代理属性设置为 `true`，则目标可以使用 `AopContext.currentProxy()` 方法。

`ProxyFactoryBean` 特有的其他属性包括：

- `proxyInterfaces`: 字符串接口名称的数组。如果未提供，则使用目标类的 CGLIB 代理（另请参见[基于 JDK 和 CGLIB 的代理](#)）。
- `InterceptorNames`: 一般切面，拦截器或其他要应用的通知名称的 String 数组。顺序很重要，先到先得。即，列表中的第一个拦截器是第一个能够拦截调用的拦截器。

名称是当前工厂中的 bean 名称，包括祖先工厂中的 bean 名称。您不能在此提及 bean 引用，因为这样做会导致 `ProxyFactoryBean` 忽略通知的单例设置。您可以在拦截器名称

后加上星号 (*)。这样做会导致应用所有一般切面 Bean，其名称以要应用星号的部分开头。您可以在[使用“全局”一般切面](#)中找到使用此功能的示例。

单例：无论调用 `getObject()` 方法的频率如何，工厂是否应返回单个对象。一些 `FactoryBean` 实现提供了这种方法。默认值是 `true`。如果要使用有状态的通知（例如，对于有状态的混入），请使用原型通知以及单例值 `false`。

6.4.3. 基于 JDK 和基于 CGLIB 的代理

本部分是有关 `ProxyFactoryBean` 如何选择为特定目标对象（将被代理）创建基于 JDK 的代理或基于 CGLIB 的代理的权威性文档。



在 Spring 的 1.2.x 版和 2.0 版之间，`ProxyFactoryBean` 的行为与创建基于 JDK 或 CGLIB 的代理有关。`ProxyFactoryBean` 现在在自动检测接口方面表现出与 `TransactionProxyFactoryBean` 类类似的语义。

如果要代理的目标对象的类（以下简称为目标类）未实现任何接口，则将创建基于 CGLIB 的代理。这是最简单的情况，因为 JDK 代理是基于接口的，并且没有接口意味着甚至无法进行 JDK 代理。您可以插入目标 bean 并通过设置 `interceptorNames` 属性来指定拦截器列表。请注意，即使 `ProxyFactoryBean` 的 `proxyTargetClass` 属性已设置为 `false`，也会创建基于 CGLIB 的代理。（这样做没有任何意义，最好将其从 Bean 定义中删除，因为它是多余的，并且在最糟的情况下会造成混淆。）

如果目标类实现一个（或多个）接口，则创建的代理类型取决于 `ProxyFactoryBean` 的配置。

如果 `ProxyFactoryBean` 的 `proxyTargetClass` 属性已设置为 `true`，则将创建基于 CGLIB 的代理。这是有道理的，并且符合最小意外原则。即使将 `ProxyFactoryBean` 的 `proxyInterfaces` 属性设置为一个或多个完全限定的接口名称，`proxyTargetClass` 属性设置为 `true` 的事实也会导致基于 CGLIB 的代理生效。

如果 `ProxyFactoryBean` 的 `proxyInterfaces` 属性已设置为一个或多个完全限定的接口名称，则将创建一个基于 JDK 的代理。创建的代理实现了 `proxyInterfaces` 属性中指定的所有接口。如果目标类恰好实现了比 `proxyInterfaces` 属性中指定的接口更多的接口，那很好，但是返回的代理不会实现这些其他接口。

如果尚未设置 `ProxyFactoryBean` 的 `proxyInterfaces` 属性，但是目标类确实实现了一个（或多个）接口，则 `ProxyFactoryBean` 会自动检测到目标类实际上至少实现了一个

接口以及创建基于 JDK 的代理。实际代理的接口是目标类实现的所有接口。实际上，这与将目标类实现的每个接口的列表提供给 `proxyInterfaces` 属性相同。然而，它的工作量明显减少，也不容易出现排版错误。

6.4.4. Proxying 接口

考虑一个简单的 `ProxyFactoryBean` 操作示例。此示例涉及：

- 一个被代理的目标 bean。这是示例中的 `personTarget` bean 定义。
- 一个一般切面和一个拦截器用来提供通知。
- 一个 AOP 代理 bean 定义指定目标对象(`personTarget` bean)代理的接口以及适用的通知。

以下列表显示了示例：

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor"
      class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>

    <property name="target" ref="personTarget"/>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

请注意，`interceptorNames` 属性采用 `String` 列表，其中包含当前工厂中的拦截器或一般切面的 Bean 名称。你可以使用一般切面，拦截器，前置，后置返回，抛出通知对象。一般切面的顺序非常重要。



您可能想知道为什么列表不包含 bean 引用。这样做的原因是，如果 `ProxyFactoryBean` 的 `singleton` 属性设置为 `false`，则它必须能够返回独立的代理实例。如果任何一般切面本身就是原型类型，则需要返回一个独立的实例，因此必须能够从工厂获得原型的实例。保持一个引用是不够的。

可以使用前面给出的 `person` Bean 定义代替 `Person` 实现，如下所示：

Java

```
Person person = (Person) factory.getBean("person");
```

Kotlin

```
val person = factory.getBean("person") as Person;
```

与普通 Java 对象一样，在同一 IoC 上下文中的其他 bean 可以表达对此的强类型依赖性。以下示例显示了如何执行此操作：

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref bean="person"/></property>
</bean>
```

在此示例中，`PersonUser` 类公开了 `Person` 类型的属性。就其而言，可以透明地使用 AOP 代理代替“真实”人的实现。但是，其类将是动态代理类。可以将其转换为 `Advised` 接口（稍后讨论）。

您可以使用匿名内部 bean 隐藏目标和代理之间的区别。仅 `ProxyFactoryBean` 定义不同。该通知仅出于完整性考虑。以下示例显示如何使用匿名内部 bean：

```

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty" value="Custom string property value"/>
</bean>

<bean id="debugInterceptor"
class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="com.mycompany.Person"/>
    <!-- Use inner bean, not local reference to target -->
    <property name="target">
        <bean class="com.mycompany.PersonImpl">
            <property name="name" value="Tony"/>
            <property name="age" value="51"/>
        </bean>
    </property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>

```

使用匿名内部 bean 的优点是只有一个 Person 类型的对象。如果我们希望防止应用程序上下文的用户获取对未通知对象的引用，或者需要避免使用 Spring IoC 自动装配的任何歧义，这将非常有用。可以说，还有一个优点是 ProxyFactoryBean 定义是独立的。但是，有时能够从工厂获得未被通知目标实际上可能是一个优势（例如，在某些测试方案中）。

6.4.5. Proxy in 代理类

如果您需要代理一个类，而不是一个或多个接口，该怎么办？

想象一下，在我们之前的示例中，没有 Person 接口。我们需要通知一个名为 Person 的类，该类没有实现任何业务接口。在这种情况下，您可以将 Spring 配置为使用 CGLIB 代理而不是动态代理。为此，请将前面显示的 ProxyFactoryBean 的 proxyTargetClass 属性设置为 true。尽管最好对接口而不是对类进行编程，但是在处理遗留代码时，通知未实现接口的类的功能可能会很有用。（通常，Spring 并不是规定性的。虽然可以轻松地应用良好实践，但可以避免强制采用特定方法。）

如果需要，即使您有接口，也可以在任何情况下强制使用 CGLIB。CGLIB 代理通过在运行时生成目标类的子类来工作。Spring 配置此生成的子类以将方法调用委托给原始目标。子类用于实现装饰模式，并编织在通知中。

CGLIB 代理通常应对用户透明。但是，有一些问题要考虑：

- `Final` 方法不能被通知，因为他们不能被覆写。
- 不需要加 CGLIB 到你的类路径，Spring3.2 之后 CGLIB 被重新打包并收入 `spring-core.jar`。换句话说，基于 CGLIB 的 AOP “即开即用”，就像 JDK 动态代理。

CGLIB 代理和动态代理之间几乎没有性能差异。在这种情况下，性能不应作为决定性的考虑因素。

6.4.6. 使用“全局”一般切面

通过在拦截器名称后附加一个星号，所有具有与该星号之前的部分匹配的 Bean 名称的一般切面都将添加到一般切面链中。如果您需要添加标准的“全局”一般切面集，这可能会派上用场。以下示例定义了两个全局一般切面：

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="service"/>
    <property name="interceptorNames">
        <list>
            <value>global*</value>
        </list>
    </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance"
    class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

6.5. 简洁代理定义

特别是在定义事务代理时，您可能会得到许多类似的代理定义。使用父子 bean 定义以及内部 bean 定义可以使代理定义更加简洁明了。

首先，我们为代理创建父模板，bean 定义，如下所示：

```

<bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="transactionAttributes">
      <props>
        <prop key="*">PROPAGATION_REQUIRED</prop>
      </props>
    </property>
  </bean>

```

它本身从未实例化，因此实际上可能是不完整的。然后，需要创建的每个代理都是一个子 bean 定义，它将代理的目标包装为内部 bean 定义，因为无论如何该目标都不会单独使用。以下示例显示了这样的子 bean：

```

<bean id="myService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
      </bean>
    </property>
  </bean>

```

您可以从父模板覆盖属性。 在以下示例中，我们将覆盖事务传播设置：

```

<bean id="mySpecialService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
      </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readonly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readonly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readonly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>

```

请注意，在父 bean 的示例中，我们通过将 `abstract` 属性设置为 `true` 来将父 bean 定义显式标记为抽象，[如前所述](#)，因此实际上可能不会实例化它。默认情况下，应用程序上下文（但不是简单的 bean 工厂）会预先实例化所有单例。因此，重要的是（至少对于单例 bean），如果您有一个（父）bean 定义仅打算用作模板，并且此定义指定了一个类，则必须确保将 `abstract` 属性设置为 `true`。否则，应用程序上下文实际上会尝试对其进行实例化。

6.6 使用 ProxyFactory 编程式创建 AOP 代理

使用 Spring 以编程方式创建 AOP 代理很容易。这使您可以使用 Spring AOP，而无需依赖 Spring IoC。

由目标对象实现的接口将被自动代理。以下清单显示了使用一个拦截器和一个一般切面为目标对象创建代理的过程：

Java

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addAdvice(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

Kotlin

```
val factory = ProxyFactory(myBusinessInterfaceImpl)
factory.addAdvice(myMethodInterceptor)
factory.addAdvisor(myAdvisor)
val tb = factory.proxy as MyBusinessInterface
```

第一步是构造一个类型为 `org.springframework.aop.framework.ProxyFactory` 的对象。您可以使用目标对象创建此对象，如前面的示例中所示，或指定要在备用构造函数中代理的接口。

您可以添加通知（使用拦截器作为一种特殊的通知），一般切面，或同时添加两者，并在 `ProxyFactory` 的整个生命周期内对其进行操作。如果添加了 `IntroductionInterceptionAroundAdvisor`，则可以使代理实现其他接口。

`ProxyFactory`（从 `AdvisedSupport` 继承）上还有便捷的方法，可让您添加其他通知类型，例如前置和异常抛出通知。`AdvisedSupport` 是 `ProxyFactory` 和 `ProxyFactoryBean` 的超类。



在大多数应用中使用 IoC 框架集成 AOP 创建时最好的实践、我们建议你使用 AOP 从 Java 代码中外部化配置，就像你应该做的。

6.7 操作 Advised 对象

无论创建 AOP 代理，都可以使用 `org.springframework.aop.framework.Advised` 接口来操作它们。任何 AOP 代理都可以强制转换为该接口，无论它实现了哪个其他接口。该接口包括以下方法：

Java

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

Kotlin

```
fun getAdvisors(): Array<Advisor>

@Throws(AopConfigException::class)
fun addAdvice(advice: Advice)

@Throws(AopConfigException::class)
fun addAdvice(pos: Int, advice: Advice)

@Throws(AopConfigException::class)
fun addAdvisor(advisor: Advisor)

@Throws(AopConfigException::class)
fun addAdvisor(pos: Int, advisor: Advisor)

fun indexOf(advisor: Advisor): Int

@Throws(AopConfigException::class)
fun removeAdvisor(advisor: Advisor): Boolean

@Throws(AopConfigException::class)
fun removeAdvisor(index: Int)

@Throws(AopConfigException::class)
fun replaceAdvisor(a: Advisor, b: Advisor): Boolean

fun isFrozen(): Boolean
```

`getAdvisors()`方法针对已添加到工厂的每个 `Advisor`, 拦截器或其他通知类型返回一个一般切面。如果添加了 `Advisor`, 则在此索引处返回的一般切面是您添加的对象。如

果添加了拦截器或其他通知类型，Spring 会将其包装在带有指向总是返回 true 的切入点的一般切面中。因此，如果添加了 `MethodInterceptor`，则为此索引返回的一般切面是 `DefaultPointcutAdvisor`，它返回您的 `MethodInterceptor` 和与所有类和方法匹配的切入点。

`addAdvisor()`方法可用于添加任何 `Advisor`。通常，拥有切入点和通知的一般切面是通用的 `DefaultPointcutAdvisor`，您可以将其与任何通知或切入点一起使用（但不能用于引介）。

默认情况下，即使已创建代理，也可以添加或删除一般切面或拦截器。唯一的限制是不可能添加或删除引介切面，因为工厂中的现有代理不会显示接口更改。（您可以从工厂获取新的代理来避免此问题。）

以下示例显示了将 AOP 代理投射到 `Advised` 接口并检查和处理其通知：

Java

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);
```

```

val advised = myObject as Advised
val advisors = advised.advisors
val oldAdvisorCount = advisors.size
println("$oldAdvisorCount advisors")

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(DebugInterceptor())

// Add selective advice using a pointcut
advised.addAdvisor(DefaultPointcutAdvisor(mySpecialPointcut, myAdvice))

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.advisors.size)

```

 尽管无疑存在合法的使用案例，但是是否建议（无双关语）修改生产中的业务对象的通知值得怀疑。但是，它在开发中（例如在测试中）非常有用。有时我们发现以拦截器或其他通知的形式添加测试代码，并进入我们要测试的方法调用中非常有用。（例如，建议可以进入为该方法创建的事务内部，也许可以在将事务标记为回滚之前运行 SQL 以检查数据库是否已正确更新。）

根据创建代理的方式，通常可以设置 `frozen` 标志。在这种情况下，`Advised isFrozen()` 方法返回 `true`，并且任何通过添加或删除来修改通知的尝试都会导致 `AopConfigException`。冻结通知对象状态的功能在某些情况下很有用（例如，防止调用代码删除安全拦截器）。

6.8. 使用“自动代理”功能

到目前为止，我们已经考虑通过使用 `ProxyFactoryBean` 或类似的工厂 bean 来显式创建 AOP 代理。

Spring 还允许我们使用“自动代理” Bean 定义，该定义可以自动代理选定的 Bean 定义。它基于 Spring 的“bean 后处理器”基础架构，可在容器加载时修改任何 bean 定义。

在此模型中，您在 XML bean 定义文件中设置了一些特殊的 bean 定义，以配置自动代理基础结构。这使您可以声明可以进行自动代理的目标。您无需使用 `ProxyFactoryBean`。

有两个方法可以做到这样：

- 通过使用在当前上下文中引用特定 bean 的自动代理创建器。

- 自动代理创建的一种特殊情况，值得单独考虑：由源级元数据属性驱动的自动代理创建。

6.8.1. Auto-proxy bean 定义

本节介绍了 `org.springframework.aop.framework.autoproxy` 包提供的自动代理创建者。

`BeanNameAutoProxyCreator`

`BeanNameAutoProxyCreator` 类是一个 `BeanPostProcessor`，可以自动为名称与文字值或通配符匹配的 bean 创建 AOP 代理。下面的示例演示如何创建 `BeanNameAutoProxyCreator` bean：

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="jdk*,onlyJdk"/>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

与 `ProxyFactoryBean` 一样，有 `interceptorNames` 属性而不是监听器列表，以允许原型切面具有正确的行为。名为“拦截器”的可以是一般切面或任何通知类型。

通常，与自动代理一样，使用 `BeanNameAutoProxyCreator` 的要点是将相同的配置一致地应用于多个对象，并且配置量最少。将声明式事务应用于多个对象是一种流行的选择。

名称匹配的 Bean 定义，例如前面示例中的 `jdkMyBean` 和 `onlyJdk`，是带有目标类的普通旧 Bean 定义。`BeanNameAutoProxyCreator` 自动创建一个 AOP 代理。相同的通知适用于所有匹配的 bean。注意，如果使用了一般切面（而不是前面示例中的拦截器），则切入点可能会不同地应用于不同的 bean。

`DefaultAdvisorAutoProxyCreator`

`DefaultAdvisorAutoProxyCreator` 是更通用，功能极其强大的自动代理创建器。这样可以在当前上下文中自动应用符合条件的一般切面，而无需在自动代理一般切面的 Bean 定义中包括特定的 Bean 名称。与 `BeanNameAutoProxyCreator` 一样，它具有一致的配置和避免重复的优点。

使用的机制包括：

- 指定一个 `DeaultAdvisorAutoProxyCreator` bean 定义。

- 在相同或相关的上下文中指定任意数量的一般切面。请注意，这些必须是一般切面，而不是拦截器或其他通知。这是必要的，因为必须有一个评估的切入点，以检查每个通知是否符合候选 bean 定义。

`DefaultAdvisorAutoProxyCreator` 自动评估每个一般切面中包含的切入点，以查看它应应用于每个业务对象（例如示例中的 `businessObject1` 和 `businessObject2`）的通知（如果有）。

这意味着可以将任意数量的一般切面自动应用于每个业务对象。如果在任何切面中没有切入点与业务对象中的任何方法匹配，则该对象不会被代理。由于为新的业务对象添加了 Bean 定义，因此如有必要，它们会自动被代理。

通常，自动代理的优点是使调用者或依赖者无法获得未通知的对象。在此 `ApplicationContext` 上调用 `getBean("businessObject1")` 会返回 AOP 代理，而不是目标业务对象。（前面显示的“inner bean”也提供了这一好处。）

以下示例创建一个 `DefaultAdvisorAutoProxyCreator` bean 和本节中讨论的其他元素：

```
<bean  
class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>  
  
<bean  
class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">  
    <property name="transactionInterceptor" ref="transactionInterceptor"/>  
</bean>  
  
<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>  
  
<bean id="businessObject1" class="com.mycompany.BusinessObject1">  
    <!-- Properties omitted -->  
</bean>  
  
<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

如果要将相同的通知一致地应用于许多业务对象，则 `DefaultAdvisorAutoProxyCreator` 非常有用。基础结构定义到位后，您可以添加新的业务对象，而无需包括特定的代理配置。您还可以轻松地添加其他切面（例如，跟踪或性能监视方面），而对配置的更改最少。

`DefaultAdvisorAutoProxyCreator` 提供过滤器和排序支持（通过使用命名约定，以便仅评估某些顾问，从而允许在同一工厂中使用多个不同配置的

`AdvisorAutoProxyCreator`)。一般切面可以实现 `org.springframework.core.Ordered` 接口，以确保在出现问题时可以正确排序。前面示例中使用的 `TransactionAttributeSourceAdvisor` 具有可配置的排序值。默认设置为无序。

6.9. 使用 TargetSource 实现

Spring 提供了以 `org.springframework.aop.TargetSource` 接口表示 `TargetSource` 的概念。该接口负责返回实现连接点的“目标对象”。每当 AOP 代理处理方法调用时，都会向 `TargetSource` 实现请求目标实例。

使用 Spring AOP 的开发人员通常不需要直接与 `TargetSource` 实现一起工作，但这提供了一种强大的手段来支持池化，可热插拔和其他复杂的目标。例如，通过使用池管理实例，池 `TargetSource` 可以为每次调用返回不同的目标实例。

如果未指定 `TargetSource`，则使用默认实现包装本地对象。每次调用都会返回相同的目标（与您期望的一样）。

本节的其余部分描述了 Spring 随附的标准目标源以及如何使用它们。



当时用一个自定义目标源的时候，你的目标通常需要一个原型而不是一个单例 bean 定义。使得 Spring 在需要时可以创建一个新的目标实例。

6.9.1. 可热插拔的目标源

`org.springframework.aop.target.HotSwappableTargetSource` 的存在是为了允许 AOP 代理的目标切换，同时允许调用者保留对其的引用。

更改目标来源的目标会立即生效。`HotSwappableTargetSource` 是线程安全的。

您可以通过在 `HotSwappableTargetSource` 上使用 `swap()` 方法来更改目标，如以下示例所示：

Java

```
HotSwappableTargetSource swapper = (HotSwappableTargetSource)
beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

Kotlin

```
val swapper = beanFactory.getBean("swapper") as HotSwappableTargetSource
val oldTarget = swapper.swap(newTarget)
```

以下示例显示了必需的 XML 定义：

```
<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg ref="initialTarget"/>
</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="swapper"/>
</bean>
```

前面的 `swap()` 调用更改了可交换 bean 的目标。拥有对该 bean 的引用的客户端不知道更改，但立即开始达到新目标。

尽管此示例未添加任何通知（不必添加通知以使用 `TargetSource`），但可以将任何 `TargetSource` 与任意通知结合使用。

6.9.2. 池化目标源

使用池目标源提供了与无状态会话 EJB 相似的编程模型，在无状态会话 EJB 中，维护了相同实例的池，方法调用将释放池中的对象。

Spring 池和 SLSB 池之间的关键区别在于，Spring 池可以应用于任何 POJO。通常，与 Spring 一样，可以以非侵入性方式应用此服务。

Spring 提供了对 Commons Pool 2.2 的支持，该池提供了相当有效的池实现。您需要在应用程序的类路径上使用 `common-pool` Jar 才能使用此功能。您还可以将 `org.springframework.aop.target.AbstractPoolingTargetSource` 子类化以支持任何其他池化 API。



在 Spring 框架 4.2 版本 Common Pool 依旧支持但是已经被弃用。

以下列表显示了一个示例配置：

```

<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
    scope="prototype">
    ...
    </bean>

<bean id="poolTargetSource"
    class="org.springframework.aop.target.CommonsPool2TargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
    <property name="maxSize" value="25"/>
</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="targetSource" ref="poolTargetSource"/>
    <property name="interceptorNames" value="myInterceptor"/>
</bean>

```

请注意，目标对象（在前面的示例中为 `businessObjectTarget`）必须是原型。这使 `PoolingTargetSource` 实现可以创建目标的新实例，以根据需要扩展池。有关其属性的信息，请参见 `AbstractPoolingTargetSource` 的 `javadoc` 和您希望使用的具体子类。`maxSize` 是最基本的，并且始终保证存在。

在这种情况下，`myInterceptor` 是需要在同一 IoC 上下文中定义的拦截器的名称。但是，您无需指定拦截器即可使用池。如果只希望池化而没有其他通知，则完全不要设置 `interceptorNames` 属性。

您可以将 Spring 配置为能够将任何池化对象投射到 `org.springframework.aop.target.PoolingConfig` 接口，该接口通过引介来公开有关池的配置和当前大小的信息。您需要定义类似于以下内容的一般切面：

```

<bean id="poolConfigAdvisor"
    class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="poolTargetSource"/>
    <property name="targetMethod" value="getPoolingConfigMixin"/>
</bean>

```

通过在 `AbstractPoolingTargetSource` 类上调用便捷方法来获得此切面，因此可以使用 `MethodInvokingFactoryBean`。该切面的名称（在此处为 `poolConfigAdvisor`）必须位于公开池对象的 `ProxyFactoryBean` 中的拦截器名称列表中。

定义如下：

Java

```

PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());

```

```
val conf = beanFactory.getBean("businessObject") as PoolingConfig
println("Max pool size is " + conf.maxSize)
```



通常不需要合并无状态服务对象。我们认为它应该是默认选择，因为大多数无状态对象自然是线程安全的，并且如果缓存了资源，实例池会造成问题。

通过使用自动代理，可以简化池。您可以设置任何自动代理创建者使用的 TargetSource 实现。

6.9.3. 原型目标源

设置“原型”目标源类似于设置池化 TargetSource。在这种情况下，每次方法调用都会创建目标的新实例。尽管在现代 JVM 中创建新对象的成本并不高，但是连接新对象（满足其 Ioc 依赖关系）的成本可能会更高。因此，没有充分的理由就不应使用此方法。

为此，您可以修改前面显示的 poolTargetSource 定义，如下所示（为清楚起见，我们也更改了名称）：

```
<bean id="prototypeTargetSource"
class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName" ref="businessObjectTarget"/>
</bean>
```

唯一的属性是目标 Bean 的名称。在 TargetSource 实现中使用继承来确保命名一致。与池化目标源一样，目标 Bean 必须是原型 Bean 定义。

6.9.4. ThreadLocal 目标源

如果您需要为每个传入请求（每个线程）创建一个对象，则 ThreadLocal 目标源非常有用。ThreadLocal 的概念提供了 JDK 范围的功能，可以透明地将资源与线程一起存储。设置 ThreadLocalTargetSource 几乎与其他类型的目标源所说明的相同，如以下示例所示：

```
<bean id="threadlocalTargetSource"
class="org.springframework.aop.target.ThreadLocalTargetSource">
    <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```



当在多线程和多类加载器环境中错误地使用 `ThreadLocal` 实例时，它
们会带来严重的问题（可能导致内存泄漏）。您应该始终考虑在其他一些
类中包装 `threadlocal`，并且永远不要直接使用 `ThreadLocal` 本身（包
装类中除外）。另外，您应始终记住正确设置和取消设置本地资源到线程。
(后者仅涉及对 `ThreadLocal.set(null)` 的调用)。在任何情况下都应进
行取消设置，因为不取消设置可能会导致出现问题。Spring 的
`ThreadLocal` 支持为您做到了这一点，应始终考虑使用 `ThreadLocal` 实
例，而无需其他适当的处理代码。

6. 10. 定义新的通知类型

Spring AOP 被设计为可扩展的。尽管目前在内部使用拦截实现策略，但是除了在环
绕通知，前置通知，异常抛出通知和后置返回通知进行拦截之外，还可以支持任意通知类型。

`org.springframework.aop.framework.adapter` 软件包是一个 SPI 软件包，可以在
不更改核心框架的情况下添加对新的自定义建议类型的 support。对自定义 `Advice` 类型的唯一
限制是它必须实现 `org.aopalliance.aop.Advice` 标记接口。

有关更多信息，请参见 `org.springframework.aop.framework.adapter` javadoc。

7. Null-safety（空值-安全性）

尽管 Java 不允许用它的类型系统来表示 null 安全性，但是 Spring 框架现在在 `org.springframework.lang` 用于声明 API 和字段的可空性的包：

- `@Nullable`: 表示特定参数，返回值或字段可以为 `null` 的注解。
- `@NonNull`: 表示特定参数，返回值或字段不能为 `null` 的注释（分在别适用于 `@NonNullApi` 和`@NonNullFields` 的参数/返回值和字段不需要）。
- `@NonNullApi`: 程序包级别的注解，它声明非空为参数和返回值的默认语义。
- `@NonNullFields`: 程序包级别的注释，它声明非 `null` 为字段的默认语义。

Spring 框架本身利用了这些注解，但是它们也可以在任何基于 Spring 的 Java 项目中使用，以声明 null 安全的 API 和可选的 null 安全的字段。通用类型参数，`varargs` 和数组元素的可空性尚不支持，但应在即将发布的版本中提供，有关最新信息，请参见 [SPR-15942](#)。可空性声明有望在 Spring Framework 版本之间进行微调，包括次要版本。方法主体内部使用的类型的可空性超出了此功能的范围。



其他常见的库（例如 Reactor 和 Spring Data）提供了使用类似可空性设置的空安全 API，从而为 Spring 应用程序开发人员提供了一致的总体体验。

7.1. 用例

除了为 Spring Framework API 可空性提供显式声明之外，IDE（例如 IDEA 或 Eclipse）还可以使用这些注解来提供与空安全性相关的有用警告，从而避免在运行时出现 `NullPointerException`。

由于 Kotlin 原生支持 `null` 安全，因此它们还用于在 Kotlin 项目中使 Spring API 为 `null` 安全。[Kotlin 支持文档](#) 中提供了更多详细信息。

7.2. JSR-305 元注解

Spring 注解使用 JSR 305 注解（休眠但广泛使用的 JSR）进行元注释。JSR-305 元注解使工具供应商（如 IDEA 或 Kotlin）以通用方式提供了空安全支持，而无需对 Spring 注解进行硬编码支持。既不需要也不建议向项目类路径添加 JSR-305 依赖项以利用 Spring 空安全 API。只有诸如在其代码库中使用空安全注释的基于 Spring 的库之类的项目才应添加 `com.google.code.findbugs: jsr305: 3.0.2` 的 `compileOnly` Gradle 配置或 Maven 提供的范围，以避免编译警告。

8. 数据缓冲区和编解码器

Java NIO 提供了 ByteBuffer，但是许多库在顶部构建了自己的字节缓冲区 API，特别是对于网络操作，其中重用缓冲区和/或使用直接缓冲区对于性能有好处。例如，Netty 具有 ByteBuf 层次结构，Undertow 使用 XNIO，Jetty 使用具有要释放的回调的池化字节缓冲区，依此类推。spring-core 模块提供了一组抽象，可与各种字节缓冲区 API 配合使用，如下所示：

- `DataBufferFactory` 抽象数据缓冲区的创建。
- `DataBuffer` 表示一个字节缓冲区，可以将其 `池化`。
- `DataBufferUtils` 提供了用于数据缓冲区的实用程序方法。
- `编解码器` 将流数据缓冲区流解码或编码为更高级别的对象。

8.1. DataBufferFactory

`DataBufferFactory` 用于以以下两种方式之一创建数据缓冲区：

1. 分配一个新的数据缓冲区，可以选择预先指定容量（如果已知），即使 `DataBuffer` 的实现可以按需增长和缩小，该容量也会更有效。
2. 包装一个现有的 `byte []` 或 `java.nio.ByteBuffer`，它用 `DataBuffer` 实现装饰给定的数据，并且不涉及分配。

请注意，`WebFlux` 应用程序不会直接创建 `DataBufferFactory`，而是通过客户端的 `ServerHttpResponse` 或 `ClientHttpRequest` 访问它。工厂的类型取决于基础客户端或服务器，例如 `NettyDataBufferFactory` 用于 `Reactor Netty`，`DefaultDataBufferFactory` 用于其他。

8.2. DataBuffer

`DataBuffer` 接口提供与 `java.nio.ByteBuffer` 类似的基本操作，但还带来了一些其他好处，其中一些是受 `Netty ByteBuf` 启发的。以下是部分优点列表：

- 具有独立位置的读取和写入，即不需要调用 `flip()` 在读取和写入之间交替。
- 与 `java.lang.StringBuilder` 一样，容量可以按需扩展。
- 通过 `PooledDataBuffer` 进行缓冲池和引用计数。
- 将缓冲区查看为 `java.nio.ByteBuffer`，`InputStream` 或 `OutputStream`。

- 确定给定字节的索引或最后一个索引。

8.3. PolledDataUtils

如 Javadoc 中 `ByteBuffer` 所述，字节缓冲区可以是直接的也可以是非直接的。直接缓冲区可以驻留在 Java 堆之外，从而无需复制本机 I / O 操作。

这使得直接缓冲区对于通过套接字接收和发送数据特别有用，但是创建和释放它们的成本也更高，这导致了缓冲池的想法。

`PooledDataBuffer` 是 `DataBuffer` 的扩展，可帮助进行引用计数，这对于字节缓冲区池至关重要。他如何工作？分配 `PooledDataBuffer` 时，引用计数为 1。调用 `retain()` 会增加计数，而调用 `release()` 会减少计数。只要计数大于 0，就保证不会释放缓冲区。当计数减少到 0 时，可以释放池中的缓冲区，这实际上意味着将为缓冲区保留的内存返回到内存池。

请注意，与其直接在 `PooledDataBuffer` 上进行操作，在大多数情况下，最好使用 `DataBufferUtils` 中的便捷方法，仅当它是 `PooledDataBuffer` 的实例时才将释放或保留应用于 `DataBuffer`。

8.4. DataBufferUtils

`DataBufferUtils` 提供了许多实用程序方法来对数据缓冲区进行操作：

- 将数据缓冲区流连接到单个缓冲区中，可能具有零个副本，例如 通过复合缓冲区（如果基础字节缓冲区 API 支持的话）。
- 将 `InputStream` 或 NIO 通道转换为 `Flux <DataBuffer>`，反之亦然，将 `Publisher <DataBuffer>` 转换为 `OutputStream` 或 NIO 通道。
- 如果缓冲区是 `PooledDataBuffer` 的实例，则释放或保留 `DataBuffer` 的方法。
- 从字节流中跳过或获取，直到特定的字节数为止。

8.5. Codecs

`org.springframework.core.codec` 包提供以下策略接口：

- **编码器**，用于将 `Publisher <T>` 编码为数据缓冲区流。
- **解码器**，用于将 `Publisher <DataBuffer>` 解码为更高级别的对象流。

`spring-core` 模块提供 `byte[]`，`ByteBuffer`，`DataBuffer`，`Resource` 和 `String` 编码器和解码器实现。`spring-web` 模块添加了 Jackson JSON，Jackson Smile，JAXB2，协议缓冲区和其他编码器和解码器。请参阅 WebFlux 部分中的[编解码器](#)。

8.6. 使用 DataBuffer

使用数据缓冲区时，必须特别小心以确保释放缓冲区，因为它们可能会被[池化](#)。我们将使用编解码器来说明其工作原理，但这些概念会更普遍地应用。让我们看看编解码器必须在内部执行哪些操作来管理数据缓冲区。

在创建更高级别的对象之前，[解码器](#)是最后一个读取输入数据缓冲区的对象，因此，它必须按以下方式释放它们：

1. 如果解码器只是读取每个输入缓冲区并准备立即释放它，则可以通过 `DataBufferUtils.release(dataBuffer)` 这样做。
2. 如果解码器使用 Flux 或 Mono 运算符(例如 `flatMap`, `reduce`)以及其他在内部预取和缓存数据项的运算符，或使用诸如过滤器，跳过和其他省略项的运算符，则必须将 `doOnDiscard(PooledDataBuffer.class, DataBufferUtils::release)` 添加到组合链中，以确保此类缓冲区在被丢弃之前被释放，结果也可能是错误或取消信号。
3. 如果解码器以任何其他方式保留一个或多个数据缓冲区，则它必须确保在完全读取时释放它们，或者在读取和释放缓存的数据缓冲区之前发生错误或取消信号的情况下。

请注意，`DataBufferUtils # join` 提供了一种安全有效的方法来将数据缓冲区流聚合到单个数据缓冲区中。同样，`skipUntilByteCount` 和 `takeUntilByteCount` 是供解码器使用的其他安全方法。

[编码器](#)分配其他必须读取（和释放）的数据缓冲区。因此，编码器无事可做。但是，如果在用数据填充缓冲区时发生序列化错误，则[编码器](#)必须小心释放数据缓冲区。例如：

Java

```
DataBuffer buffer = factory.allocateBuffer();
boolean release = true;
try {
    // serialize and populate buffer..
    release = false;
}
finally {
    if (release) {
        DataBufferUtils.release(buffer);
    }
}
return buffer;
```

Kotlin

```
val buffer = factory.allocateBuffer()
var release = true
try {
    // serialize and populate buffer..
    release = false
} finally {
    if (release) {
        DataBufferUtils.release(buffer)
    }
}
return buffer
```

[编码器](#)的使用者负责释放其接收的数据缓冲区。在 WebFlux 应用程序中，[编码器](#)的输出用于写入 HTTP 服务器响应或客户端 HTTP 请求，在这种情况下，释放数据缓冲区是代码写入服务器响应或客户端请求。

请注意，在 Netty 上运行时，有用于[调试缓冲区泄漏](#)的调试选项。

9. 附录

9.1. XML 结构

附录的此部分列出了与核心容器相关的 XML 模式。

9.1.1. `util` 结构

顾名思义，`util` 标签处理常见的实用程序配置问题，例如配置集合，引用常量等。要在 `util` 模式中使用标签，您需要在 Spring XML 配置文件的顶部具有以下序言（代码段中的文本引用了正确的模式，以便您可以使用 `util` 命名空间中的标签）：

```
<?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:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/util
        https://www.springframework.org/schema/util/spring-util.xsd">

    <!-- bean definitions here -->

</beans>
```

使用`<util:constant/>`

请看下列 bean 定义：

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
            class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

前面的配置使用 Spring FactoryBean 实现(`FieldRetrievingFactoryBean`)将 Bean 上的隔离属性的值设置为 `java.sql.Connection.TRANSACTION_SERIALIZABLE` 常量的值。这一切都很好，但是很冗长，并且(不必要地)将 Spring 的内部管道暴露给最终用户。

以下基于 XML Schema 的版本更加简洁，清楚地表达了开发人员的意图(“注入此常量值”)，并且读起来更好：

```
<bean id="..." class="...">
    <property name="isolation">
        <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
    </property>
</bean>
```

从字段值设置 Bean 属性或构造函数参数

[FieldRetrievingFactoryBean](#) 是一个 [FactoryBean](#)，它检索静态或非静态字段值。

它通常用于检索 `public static final` 常量，然后可以将其用于为另一个 bean 设置属性值或构造函数参数。

以下示例显示如何使用 `staticField` 属性公开 `static` 字段：

```
<bean id="myField"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
    <property name="staticField"
      value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

还有一种便利用法形式，其中将静态字段指定为 Bean 名称，如以下示例所示：

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

这的确意味着不再需要选择 bean id 是什么（因此，引用它的任何其他 bean 也必须使用这个较长的名称），但是这种形式的定义非常简洁，并且可以很方便地用作内部对象。因为不必为 bean 引用指定 id，如以下示例所示：

```
<bean id="..." class="...">
    <property name="isolation">
        <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
              class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
    </property>
</bean>
```

您还可以访问另一个 bean 的非静态（实例）字段，如 [FieldRetrievingFactoryBean](#) 类的 API 文档中所述。

在 Spring 中很容易将枚举值作为属性或构造函数参数注入到 bean 中。实际上，您不必做任何事情或不了解 Spring 内部知识（甚至不了解诸如 [FieldRetrievingFactoryBean](#) 之类的类）也不知道任何事情。以下示例枚举显示了注入枚举值贼简单：

Java

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED
}
```

Kotlin

```
package javax.persistence

enum class PersistenceContextType {

    TRANSACTION,
    EXTENDED
}
```

现在考虑以下类型的 PersistenceContextType setter 和相应的 bean 定义：

Java

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

Kotlin

```
package example

class Client {

    lateinit var persistenceContextType: PersistenceContextType
}
```

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION"/>
</bean>
```

使用`<util:property-path/>`

请看下面示例：

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

前面的配置使用 `Spring FactoryBean` 实现 (`PropertyPathFactoryBean`) 创建一个名为 `testBean.age` 的 bean (类型为 `int`)，其值等于 `testBean` bean 的 `age` 属性。

现在考虑以下示例，该示例添加了一个`<util: property-path/>`元素：

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

<!-- results in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

`<property-path/>` 元素的 `path` 属性的值遵循 `beanName.beanProperty` 的形式。在这种情况下，它将获取名为 `testBean` 的 bean 的 `age` 属性。`age` 的值是 10.

使用`<util:property-path/>`设置 Bean 属性或构造函数参数

`PropertyPathFactoryBean` 是一个 `FactoryBean`，用于评估给定目标对象上的属性路径。可以直接指定目标对象，也可以通过 `bean` 名称指定目标对象。然后，您可以在另一个 `bean` 定义中将此值用作属性值或构造函数参数。

下面的示例按名称显示了针对另一个 `bean` 的路径：

```

// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
    <property name="age" value="10"/>
    <property name="spouse">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="11"/>
        </bean>
    </property>
</bean>

// results in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
      class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetBeanName" value="person"/>
    <property name="propertyPath" value="spouse.age"/>
</bean>

```

在以下示例中，针对内部 bean 评估路径：

```

<!-- results in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
      class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
    <property name="targetObject">
        <bean class="org.springframework.beans.TestBean">
            <property name="age" value="12"/>
        </bean>
    </property>
    <property name="propertyPath" value="age"/>
</bean>

```

还有一种快捷方式，其中 Bean 名称是属性路径。以下示例显示了快捷方式形式：

```

<!-- results in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
      class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

```

这种形式的确意味着在 bean 名称中没有选择。对它的任何引用也必须使用相同的 ID，即路径。如果用作内部 bean，则根本不需要引用它，如以下示例所示：

```
<bean id="..." class="...">
    <property name="age">
        <bean id="person.age"
            class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
    </property>
</bean>
```

您可以在实际定义中专门设置结果类型。对于大多数用例来说，这不是必需的，但有时可能会有用。有关此功能的更多信息，请参见 javadoc。

使用`<util:properties/>`

请看下面的例子：

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<bean id="jdbcConfiguration"
    class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

前面的配置使用 Spring `FactoryBean` 实现（`PropertyFactoryBean`）来实例化 `java.util.Properties` 实例，并从提供的 `Resource` 位置加载值。

以下示例使用 `util: properties` 元素进行更简洁的表示：

```
<!-- creates a java.util.Properties instance with values loaded from the supplied
location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-
production.properties"/>
```

使用`<util:list/>`

```
<!-- creates a java.util.List instance with values loaded from the supplied  
'sourceList' -->  
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">  
    <property name="sourceList">  
        <list>  
            <value>pechorin@hero.org</value>  
            <value>raskolnikov@slums.org</value>  
            <value>stavrogin@gov.org</value>  
            <value>porfiry@gov.org</value>  
        </list>  
    </property>  
</bean>
```

前面的配置使用 Spring `FactoryBean` 实现（`ListFactoryBean`）创建一个 `java.util.List` 实例，并使用从提供的 `sourceList` 中获取的值对其进行初始化。

以下示例使用`<util: list/>`元素进行更简洁的表示：

```
<!-- creates a java.util.List instance with the supplied values -->  
<util:list id="emails">  
    <value>pechorin@hero.org</value>  
    <value>raskolnikov@slums.org</value>  
    <value>stavrogin@gov.org</value>  
    <value>porfiry@gov.org</value>  
</util:list>
```

您还可以通过使用`<util:list/>`元素上的 `list-class` 属性来显式控制实例化和填充的 `List` 的确切类型。例如，如果我们确实需要实例化 `java.util.LinkedList`，则可以使用以下配置：

```
<util:list id="emails" list-class="java.util.LinkedList">  
    <value>jackshaftoe@vagabond.org</value>  
    <value>eliza@thinkingmanscrumpect.org</value>  
    <value>vanhoek@pirate.org</value>  
    <value>d'Arcachon@nemesis.org</value>  
</util:list>
```

如果没有提供 `list-class` 属性，则容器选择 `List` 实现。

使用`<util:map/>`

请看以下示例：

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
        <map>
            <entry key="pechorin" value="pechorin@hero.org"/>
            <entry key="raskolnikov" value="raskolnikov@slums.org"/>
            <entry key="stavrogin" value="stavrogin@gov.org"/>
            <entry key="porfiry" value="porfiry@gov.org"/>
        </map>
    </property>
</bean>
```

前面的配置使用 Spring FactoryBean 实现（`MapFactoryBean`）创建一个 `java.util.Map` 实例，该实例使用从提供的“`sourceMap`”中获取的键值对进行初始化。

以下示例使用`<util: map />`元素进行更简洁的表示：

```
<!-- creates a java.util.Map instance with the supplied key-value pairs -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

您还可以通过使用`<util: map/>`元素上的‘`map-class`’属性来显式控制实例化和填充的 Map 的确切类型。例如，如果我们确实需要实例化 `java.util.TreeMap`，则可以使用以下配置：

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

如果未提供“`map-class`”属性，则容器选择 `Map` 实现。

使用`<util:set/>`

请看下列示例：

```

<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet'
-->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
        <set>
            <value>pechorin@hero.org</value>
            <value>raskolnikov@slums.org</value>
            <value>stavrogin@gov.org</value>
            <value>porfiry@gov.org</value>
        </set>
    </property>
</bean>

```

前面的配置使用 Spring FactoryBean 实现（`SetFactoryBean`）创建一个 `java.util.Set` 实例，该实例使用从提供的 `sourceSet` 中获取的值进行初始化。

以下示例使用`<util:set/>`元素进行更简洁的表示：

```

<!-- creates a java.util.Set instance with the supplied values -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>

```

您还可以使用`<util:set/>`元素上的 `set-class` 属性来显式控制实例化和填充的 Set 的确切类型。例如，如果我们确实需要实例化 `java.util.TreeSet`，则可以使用以下配置：

```

<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>

```

如果未提供 `set-class` 属性，则容器选择 `Set` 实现。

9.1.2. aop 结构

`aop` 标签用于配置 Spring 中的所有 AOP，包括 Spring 自己的基于代理的 AOP 框架以及 Spring 与 AspectJ AOP 框架的集成。这些标签在名为“[面向方面的 Spring 编程\(第 5 章\)](#)”的章节中全面介绍。

为了完整起见，要在 `aop` 模式中使用标记，您需要在 Spring XML 配置文件的顶部具有以下前导（代码段中的文本引用了正确的模式，以便你能有效使用 `aop` 名称空间中的标

记) :

```
<?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 definitions here -->

</beans>
```

9.1.3. `context` 结构

上下文标记处理与管道有关的 `ApplicationContext` 配置，即通常不是对最终用户重要的 bean，而是在 Spring 中完成大量“艰巨”工作的 bean，例如 `BeanfactoryPostProcessors`。以下代码段引用了正确的架构，以便您可以使用上下文名称空间中的元素：

```
<?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">

    <!-- bean definitions here -->

</beans>
```

使用`<property-placeholder/>`

此元素激活`$ {…}`占位符的替换，这些占位符针对指定的属性文件（作为 Spring 资源位置）解析。该元素是为您设置 `PropertySourcesPlaceholderConfigurer` 的便捷机制。如果您需要对特定的 `PropertySourcesPlaceholderConfigurer` 设置进行更多控制，则可以自己将其明确定义为 Bean。

使用`<annotation-config/>`

此元素激活 Spring 基础结构以检测 Bean 类中的注解：

- Spring 的@`Configuration` 模型
- @`Autowired`/@`Injec` 和@`Value`
- JSR-250 的@`Resource`, @`PostConstruct` 和@`PreDestroy` (如果有)
- JPA 的@`PersistenceContext` 和@`PersistenceUnit` (如果有)
- Spring 的@`EventListener`

或者，您可以选择为这些注解显式激活各个 BeanPostProcessor。



此元素不会激活 Spring 的@`Transactional` 注解的处理；您可以为此目的使用<`tx:annotation-driven`>元素。同样，还需要明确启用 Spring 的缓存注解。

使用<`component-scan`>

有关基于注解的容器配置(1.9)的部分中详细介绍了此元素。

使用<`load-time-weaver`>

在 Spring Framework 中关于使用 AspectJ 进行加载时编织(5.10.4)的部分中详细介绍了此元素。

使用<`spring-configured`>

关于使用 AspectJ 通过 Spring 依赖注入域对象(5.10.1)的部分中详细介绍了此元素。

使用<`mbean-export`>

有关配置基于注解的 MBean 导出的部分中详细介绍了此元素。

9.1.4. beans 结构

最后但并非最不重要的是，我们有 bean 模式中的元素。这些要素自框架诞生之初就已存在于 Spring。这里没有展示 bean 模式中各种元素的示例，因为它们在依赖项和配置中有非常全面的介绍（实际上，在整个章节中）。

请注意，您可以向<`bean` /> XML 定义添加零个或多个键值对。使用此额外的元数据进行的操作（如果有的话）完全取决于您自己的自定义逻辑（因此通常仅在您按照标题为 XML Schema Authoring 的附录中所述编写自己的自定义元素时使用）。

以下示例在周围的<`bean` />上下文中显示了<`meta` />元素（请注意，由于没有任何逻辑来解释它，因此元数据实际上是毫无用处的）。

```

<?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="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/> ①
        <property name="name" value="Rick"/>
    </bean>

</beans>

```

① 这是示例 `meta` 元素

在前面的示例中，您可以假设存在一些逻辑，这些逻辑销毁了 bean 的定义，并建立了一些使用提供的元数据的缓存基础结构。

9.2. XML 架构创作

从 2.0 版开始，Spring 提供了一种机制，可以将基于架构的扩展添加到基本 Spring XML 格式中，以定义和配置 bean。本节介绍如何编写自己的自定义 XML Bean 定义解析器，以及如何将此类解析器集成到 Spring IoC 容器中。

为了促进使用支持架构的 XML 编辑器编写配置文件，Spring 的可扩展 XML 配置机制基于 XML Schema。如果您不熟悉标准 Spring 发行版随附的 Spring 当前的 XML 配置扩展，则应首先阅读标题为 [appendix.pdf](#) 的附录。

要创建新的 XML 配置扩展，请执行以下操作：

- ① 编写 XML 模式以描述您的自定义元素。
- ② 编写自定义 `NamespaceHandler` 实现。
- ③ 编写一个或多个 `BeanDefinitionParser` 实现（这是完成实际工作的地方）。
- ④ 向 Spring 注册您的新组件。

对于一个统一的示例，我们创建一个 XML 扩展（一个自定义 XML 元素），该扩展使我们可以配置类型 `SimpleDateFormat` 的对象（来自 `java.text` 包）。完成后，我们将能够如下定义 `SimpleDateFormat` 类型的 bean 定义：

```

<myns:dateFormat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>

```

（我们将在本附录后面提供更详细的示例。第一个简单示例的目的是引导您完成制作自

定义扩展的基本步骤。)

9.2.1 编写 Schema

创建用于 Spring 的 IoC 容器的 XML 配置扩展首先要编写 XML 模式来描述扩展。对于我们的示例，我们使用以下架构来配置 `SimpleDateFormat` 对象：

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.example/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.mycompany.example/schema/myns"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.springframework.org/schema/beans"/>

    <xsd:element name="dateformat">
        <xsd:complexType>
            <xsd:complexContent>
                <xsd:extension base="beans:identifiedType"> ①
                    <xsd:attribute name="lenient" type="xsd:boolean"/>
                    <xsd:attribute name="pattern" type="xsd:string" use="required"/>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

① 所指示的行包含所有可识别标签的扩展基础（这意味着它们具有 `id` 属性，我们可以将其用作容器中的 `bean` 标识符）。我们可以使用此属性，因为我们导入了 Spring 提供的 `bean` 名称空间。

前面的架构使我们可以使用 `<myns:dateformat/>` 元素直接在 XML 应用程序上下文文件中配置 `SimpleDateFormat` 对象，如以下示例所示：

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

请注意，在创建基础结构类之后，上述 XML 片段与以下 XML 片段基本相同：

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

前面两个片段中的第二个片段在容器中创建了一个 Bean（由名称 `SimpleDateFormat` 类型的 `dateFormat` 标识），并设置了几个属性。



创建配置格式的基于模式的方法允许与具有模式识别 XML 编辑器的 IDE 紧密集成。通过使用正确编写的架构，可以使用自动完成功能来让用户在枚举中定义的多个配置选项之间进行选择。

9.2.2. 编写一个 NamespaceHandler

除了模式，我们还需要一个 `NamespaceHandler` 来解析 Spring 在解析配置文件时遇到的这个特定名称空间的所有元素。对于此示例，`NamespaceHandler` 应该处理 `myns:dateFormat` 元素的解析。

`NamespaceHandler` 接口具有以下三种方法：

- `init()`: 允许初始化 `NamespaceHandler`，并在使用处理程序之前由 Spring 调用。
- `BeanDefinition parse(Element, ParserContext)`: 当 Spring 遇到顶级元素（未嵌套在 `bean` 定义或其他命名空间中）时调用。此方法本身可以注册 `Bean` 定义，返回 `Bean` 定义或两者。
- `BeanDefinitionHolder`

`decorate(Node, beanDefinitionHolder, ParserContext)`: 当 Spring 遇到另一个名称空间的属性或嵌套元素时调用。一个或多个 `bean` 定义的修饰（例如）与 Spring 支持的作用域一起使用。我们首先突出显示一个简单的示例，而不使用装饰，然后在一个更高级的示例中显示装饰。

尽管您可以为整个名称空间编写自己的 `NamespaceHandler`（从而提供解析该名称空间中每个元素的代码），但是通常情况下，Spring XML 配置文件中的每个顶级 XML 元素都产生一个 `bean` 定义（在我们的例子中，单个 `<myns:dateFormat/>` 元素指明单个 `SimpleDateFormat` `bean` 定义）。Spring 提供了许多支持这种情况的便利类。在下面的示例中，我们使用 `NamespaceHandlerSupport` 类：

Java

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new
SimpleDateFormatBeanDefinitionParser());
    }
}
```

Kotlin

```
package org.springframework.samples.xml

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class MyNamespaceHandler : NamespaceHandlerSupport {

    override fun init() {
        registerBeanDefinitionParser("dateformat",
SimpleDateFormatBeanDefinitionParser())
    }
}
```

您可能会注意到，此类中实际上没有很多解析逻辑。实际上，`NamespaceHandlerSupport` 类具有内置的委托概念。它支持注册任何数量的 `BeanDefinitionParser` 实例，在需要解析其名称空间中的元素时将其委托给该实例。这种清晰的关注点分离使 `NamespaceHandler` 可以处理其命名空间中所有自定义元素的解析编排，同时委托 `BeanDefinitionParsers` 来完成 XML 解析的繁琐工作。这意味着每个 `BeanDefinitionParser` 都只包含用于解析单个自定义元素的逻辑，正如我们在下一步中看到的那样。

9.2.3. 使用 BeanDefinitionParser

如果 `NamespaceHandler` 遇到映射到特定 bean 定义解析器（在这种情况下为 `dateformat`）的类型的 XML 元素，则使用 `BeanDefinitionParser`。换句话说，`BeanDefinitionParser` 负责解析模式中定义的一个独特的顶级 XML 元素。

在解析器中，我们可以访问 XML 元素（因此也可以访问其子元素），以便我们可以解析自定义 XML 内容，如以下示例所示：

Java

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends
AbstractSingleBeanDefinitionParser { ①

    protected Class<?> getBeanClass(Element element) {
        return SimpleDateFormat.class; ②
    }

    protected void doParse(Element element, BeanDefinitionBuilder bean) {
        // this will never be null since the schema explicitly requires that a value
        be supplied
        String pattern = element.getAttribute("pattern");
        bean.addConstructorArgValue(pattern);

        // this however is an optional property
        String lenient = element.getAttribute("lenient");
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
        }
    }
}
```

① 我们使用 Spring 提供的 `AbstractSingleBeanDefinitionParser` 来处理创建单个 `BeanDefinition` 的许多基本工作。

② 我们为 `AbstractSingleBeanDefinitionParser` 超类提供单个 `BeanDefinition` 表示的类型。

```

package org.springframework.samples.xml

import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser
import org.springframework.util.StringUtils
import org.w3c.dom.Element

import java.text.SimpleDateFormat

class SimpleDateFormatBeanDefinitionParser : AbstractSingleBeanDefinitionParser() { ①

    override fun getBeanClass(element: Element): Class<*>? { ②
        return SimpleDateFormat::class.java
    }

    override fun doParse(element: Element, bean: BeanDefinitionBuilder) {
        // this will never be null since the schema explicitly requires that a value
        be supplied
        val pattern = element.getAttribute("pattern")
        bean.addConstructorArgValue(pattern)

        // this however is an optional property
        val lenient = element.getAttribute("lenient")
        if (StringUtils.hasText(lenient)) {
            bean.addPropertyValue("lenient", java.lang.Boolean.valueOf(lenient))
        }
    }
}

```

① 我们使用 Spring 提供的 `AbstractSingleBeanDefinitionParser` 来处理创建单个 `BeanDefinition` 的许多基本工作。

② 我们为 `AbstractSingleBeanDefinitionParser` 超类提供单个 `BeanDefinition` 表示的类型。

在这种简单的情况下，这就是我们要做的全部。单个 `BeanDefinition` 的创建由 `AbstractSingleBeanDefinitionParser` 超类处理，`bean` 定义的唯一标识符的提取和设置也是如此。

9.2.4. 注册 Handler 和 Schema

编码完成。剩下要做的就是让 Spring XML 解析基础结构了解我们的自定义元素。通过在两个特殊用途的属性文件中注册我们的自定义 `namespaceHandler` 和自定义 XSD 文件来实现。这些属性文件都放置在应用程序的 META-INF 目录中，例如，可以与二进制类一起分发到 JAR 文件中。Spring XML 解析基础结构通过使用这些特殊的属性文件来自动选择您的新扩展，以下两部分将详细介绍其格式。

写 META-INF/spring.handlers

名为 `spring.handlers` 的属性文件包含 XML Schema URI 到命名空间处理器类的映射。对于我们的示例，我们需要编写以下内容：

```
http://www.mycompany.example/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

(`:`字符是 Java 属性格式中的有效分隔符，因此 URI 中的`:`字符需要用反斜杠转义。) 键值对的第一部分（键）是与您的自定义名称空间扩展关联的 URI，并且需要与您的自定义 XSD 架构中指定的 `targetNamespace` 属性值完全匹配。

写' META-INF/spring.schema'

名为 `spring.schemas` 的属性文件包含 XML 架构位置（与架构声明一起引用，在 XML 文件中使用该架构作为 `xsi: schemaLocation` 属性的一部分）与类路径资源的映射。需要使用该文件来防止 Spring 绝对使用默认的 `EntityResolver`，该默认的 `EntityResolver` 需要 Internet 访问才能检索架构文件。如果在此属性文件中指定映射，Spring 会在类路径上搜索架构（在本例中为 `org.springframework.samples.xml` 包中的 `myns.xsd`）。

以下代码段显示了我们需要为自定义架构添加的行：

```
http://www.mycompany.example/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

（注意：要被转义）

建议在类路径的 `NamespaceHandler` 和 `BeanDefinitionParser` 类旁边部署 XSD 文件。

9.2.5. 使用 Custom Extension 在你的 Spring XML 配置文件

使用您自己实现的自定义扩展与使用 Spring 提供的“自定义”扩展没有什么不同。以下示例在 Spring XML 配置文件中使用前面步骤中开发的自定义 `<dateformat/>` 元素：

```

<?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:myns="http://www.mycompany.example/schema/myns"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           https://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.mycompany.example/schema/myns
           http://www.mycompany.com/schema/myns/myns.xsd">

    <!-- as a top-level bean -->
    <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm"
lenient="true"/> ①

    <bean id="jobDetailTemplate" abstract="true">
        <property name="dateFormat">
            <!-- as an inner bean -->
            <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
        </property>
    </bean>

</beans>

```

① 一个自定义 bean。

9.2.6. 更多详细示例

本节提供了一些更详细的自定义 XML 扩展示例。

在自定义元素内嵌自定义元素

本节中的示例显示如何编写满足以下配置目标所需的各种工件：

```
<?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:foo="http://www.foo.example/schema/component"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.foo.example/schema/component
        http://www.foo.example/schema/component/component.xsd">

    <foo:component id="bionic-family" name="Bionic-1">
        <foo:component name="Mother-1">
            <foo:component name="Karate-1"/>
            <foo:component name="Sport-1"/>
        </foo:component>
        <foo:component name="Rock-1"/>
    </foo:component>

</beans>
```

前面的配置将自定义扩展相互嵌套。`<foo:component/>` 元素实际配置的类是 `Component` 类(在下一个示例中显示)。请注意，`Component` 类如何不为 `Components` 属性公开 `setter` 方法。这使得很难(或几乎不可能)通过使用 `setter` 注入为 `Component` 类配置 `bean` 定义。以下清单显示了 `Component` 类：

Java

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

    private String name;
    private List<Component> components = new ArrayList<Component>();

    // mmm, there is no setter method for the 'components'
    public void addComponent(Component component) {
        this.components.add(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Kotlin

```
package com.foo

import java.util.ArrayList

class Component {

    var name: String? = null
    private val components = ArrayList<Component>()

    // mmm, there is no setter method for the 'components'
    fun addComponent(component: Component) {
        this.components.add(component)
    }

    fun getComponents(): List<Component> {
        return components
    }
}
```

解决此问题的典型方法是创建一个自定义 `FactoryBean`, 它公开了 `component` 属性的 `setter` 属性。以下清单显示了这样的自定义 `FactoryBean`:

Java

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

    private Component parent;
    private List<Component> children;

    public void setParent(Component parent) {
        this.parent = parent;
    }

    public void setChildren(List<Component> children) {
        this.children = children;
    }

    public Component getObject() throws Exception {
        if (this.children != null && this.children.size() > 0) {
            for (Component child : children) {
                this.parent.addComponent(child);
            }
        }
        return this.parent;
    }

    public Class<Component> getObjectType() {
        return Component.class;
    }

    public boolean isSingleton() {
        return true;
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.FactoryBean
import org.springframework.stereotype.Component

class ComponentFactoryBean : FactoryBean<Component> {

    private var parent: Component? = null
    private var children: List<Component>? = null

    fun setParent(parent: Component) {
        this.parent = parent
    }

    fun setChildren(children: List<Component>) {
        this.children = children
    }

    override fun getObjectType(): Class<Component>? {
        return Component::class.java
    }

    override fun isSingleton(): Boolean {
        return true
    }
}
```

这很好用，但是向最终用户暴露了很多 Spring 管道。我们要做的是编写一个自定义扩展名，以隐藏所有此 Spring 管道。如果我们坚持[前面描述的步骤](#)，那么我们首先创建 XSD 模式来定义我们的自定义标签的结构，如下清单所示：

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.example/schema/component"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.foo.example/schema/component"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:element name="component">
    <xsd:complexType>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="component"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:ID"/>
      <xsd:attribute name="name" use="required" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

再次按照[前面描述的过程](#), 然后创建一个自定义的 **NamespaceHandler**:

Java

```

package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("component", new
ComponentBeanDefinitionParser());
    }
}

```

Kotlin

```

package com.foo

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class ComponentNamespaceHandler : NamespaceHandlerSupport() {

    override fun init() {
        registerBeanDefinitionParser("component", ComponentBeanDefinitionParser())
    }
}

```

接下来是自定义 BeanDefinitionParser。请记住，我们正在创建一个描述 ComponentFactoryBean 的 BeanDefinition。以下列表显示了我们的自定义 BeanDefinitionParser 实现：

Java

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

    protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
        return parseComponentElement(element);
    }

    private static AbstractBeanDefinition parseComponentElement(Element element) {
        BeanDefinitionBuilder factory =
        BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
        factory.addProperty("parent", parseComponent(element));

        List<Element> childElements = DomUtils.getChildElementsByTagName(element,
        "component");
        if (childElements != null && childElements.size() > 0) {
            parseChildComponents(childElements, factory);
        }

        return factory.getBeanDefinition();
    }

    private static BeanDefinition parseComponent(Element element) {
        BeanDefinitionBuilder component =
        BeanDefinitionBuilder.rootBeanDefinition(Component.class);
        component.addProperty("name", element.getAttribute("name"));
        return component.getBeanDefinition();
    }

    private static void parseChildComponents(List<Element> childElements,
    BeanDefinitionBuilder factory) {
        ManagedList<BeanDefinition> children = new
        ManagedList<BeanDefinition>(childElements.size());
```

```
        for (Element element : childElements) {
            children.add(parseComponentElement(element));
        }
        factory.addPropertyValue("children", children);
    }
}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.config.BeanDefinition
import org.springframework.beans.factory.support.AbstractBeanDefinition
import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.support.ManagedList
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser
import org.springframework.beans.factory.xml.ParserContext
import org.springframework.util.xml.DomUtils
import org.w3c.dom.Element

import java.util.List

class ComponentBeanDefinitionParser : AbstractBeanDefinitionParser() {

    override fun parseInternal(element: Element, parserContext: ParserContext):
AbstractBeanDefinition? {
        return parseComponentElement(element)
    }

    private fun parseComponentElement(element: Element): AbstractBeanDefinition {
        val factory =
BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean::class.java)
        factory.addPropertyValue("parent", parseComponent(element))

        val childElements = DomUtils.getChildElementsByTagName(element, "component")
        if (childElements != null && childElements.size > 0) {
            parseChildComponents(childElements, factory)
        }

        return factory.getBeanDefinition()
    }

    private fun parseComponent(element: Element): BeanDefinition {
        val component =
BeanDefinitionBuilder.rootBeanDefinition(Component::class.java)
        component.addPropertyValue("name", element.getAttribute("name"))
        return component.beanDefinition
    }

    private fun parseChildComponents(childElements: List<Element>, factory:
BeanDefinitionBuilder) {
        val children = ManagedList<BeanDefinition>(childElements.size)
        for (element in childElements) {
            children.add(parseComponentElement(element))
        }
        factory.addPropertyValue("children", children)
    }
}
```

最后，需要通过修改 META-INF/spring.handlers 和 META-INF/spring.schemas 文件，在 Spring XML 基础结构中注册各种工件，如下所示：

```
# in 'META-INF/spring.handlers'  
http://www.foo.example/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'  
http://www.foo.example/schema/component/component.xsd=com/foo/component.xsd
```

自定义属性在普通元素上

编写自己的自定义解析器和关联的工件并不难。但是，有时这不是正确的选择。考虑一个需要将元数据添加到已经存在的 bean 定义的场景。在这种情况下，您当然不需要编写自己的整个自定义扩展。相反，您只想向现有的 bean 定义元素添加一个附加属性。

举另一个例子，假设您为访问集群 JCache 的服务对象（它不知道）定义了一个 Bean 定义，并且您想确保在周围的集群中急切启动命名的 JCache 实例。下面展示了这么一个定义：

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"  
      jcache:cache-name="checking.account">  
    <!-- other dependencies here... -->  
</bean>
```

然后，当解析'jcache: cache-name'属性时，我们可以创建另一个 BeanDefinition。然后， BeanDefinition 为我们初始化了命名的 JCache。我们还可以为'checkingAccountService'修改现有的 BeanDefinition，以便它依赖于此新的 JCache-initializing BeanDefinition。以下清单显示了我们的 JCacheInitializer：

Java

```
package com.foo;

public class JCacheInitializer {

    private String name;

    public JCacheInitializer(String name) {
        this.name = name;
    }

    public void initialize() {
        // lots of JCache API calls to initialize the named cache...
    }
}
```

Kotlin

```
package com.foo

class JCacheInitializer(private val name: String) {

    fun initialize() {
        // lots of JCache API calls to initialize the named cache...
    }
}
```

现在我们可以进入自定义扩展了。首先，我们需要编写描述自定义属性的 XSD 架构，如下所示：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.example/schema/jcache"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.foo.example/schema/jcache"
    elementFormDefault="qualified">

    <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

接下来，我们需要创建关联的 NamespaceHandler，如下所示：

Java

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
    }

}
```

Kotlin

```
package com.foo

import org.springframework.beans.factory.xml.NamespaceHandlerSupport

class JCacheNamespaceHandler : NamespaceHandlerSupport() {

    override fun init() {
        super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            JCacheInitializingBeanDefinitionDecorator())
    }

}
```

接下来，我们需要创建解析器。请注意，在这种情况下，因为我们要解析 XML 属性，所以我们编写了 `BeanDefinitionDecorator` 而不是 `BeanDefinitionParser`。以下清单显示了我们的 `BeanDefinitionDecorator` 实现：

Java

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```

public class JCacheInitializingBeanDefinitionDecorator implements
BeanDefinitionDecorator {

    private static final String[] EMPTY_STRING_ARRAY = new String[0];

    public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,
        ParserContext ctx) {
        String initializerBeanName = registerJCacheInitializer(source, ctx);
        createDependencyOnJCacheInitializer(holder, initializerBeanName);
        return holder;
    }

    private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,
        String initializerBeanName) {
        AbstractBeanDefinition definition = ((AbstractBeanDefinition)
holder.getBeanDefinition());
        String[] dependsOn = definition.getDependsOn();
        if (dependsOn == null) {
            dependsOn = new String[]{initializerBeanName};
        } else {
            List dependencies = new ArrayList(Arrays.asList(dependsOn));
            dependencies.add(initializerBeanName);
            dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
        }
        definition.setDependsOn(dependsOn);
    }

    private String registerJCacheInitializer(Node source, ParserContext ctx) {
        String cacheName = ((Attr) source).getValue();
        String beanName = cacheName + "-initializer";
        if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
            BeanDefinitionBuilder initializer =
BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);
            initializer.addConstructorArg(cacheName);
            ctx.getRegistry().registerBeanDefinition(beanName,
initializer.getBeanDefinition());
        }
        return beanName;
    }
}

```

Kotlin

```
package com.foo

import org.springframework.beans.factory.config.BeanDefinitionHolder
import org.springframework.beans.factory.support.AbstractBeanDefinition
import org.springframework.beans.factory.support.BeanDefinitionBuilder
import org.springframework.beans.factory.xml.BeanDefinitionDecorator
import org.springframework.beans.factory.xml.ParserContext
import org.w3c.dom.Attr
import org.w3c.dom.Node

import java.util.ArrayList

class JCacheInitializingBeanDefinitionDecorator : BeanDefinitionDecorator {

    override fun decorate(source: Node, holder: BeanDefinitionHolder,
        ctx: ParserContext): BeanDefinitionHolder {
        val initializerBeanName = registerJCacheInitializer(source, ctx)
        createDependencyOnJCacheInitializer(holder, initializerBeanName)
        return holder
    }

    private fun createDependencyOnJCacheInitializer(holder: BeanDefinitionHolder,
        initializerBeanName: String) {
        val definition = holder.beanDefinition as AbstractBeanDefinition
        var dependsOn = definition.dependsOn
        dependsOn = if (dependsOn == null) {
            arrayOf(initializerBeanName)
        } else {
            val dependencies = ArrayList(listOf(*dependsOn))
            dependencies.add(initializerBeanName)
            dependencies.toTypedArray()
        }
        definition.setDependsOn(*dependsOn)
    }

    private fun registerJCacheInitializer(source: Node, ctx: ParserContext): String {
        val cacheName = (source as Attr).value
        val beanName = "$cacheName-initializer"
        if (!ctx.registry.containsBeanDefinition(beanName)) {
            val initializer =
                BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer::class.java)
                initializer.addConstructorArg(cacheName)
                ctx.registry.registerBeanDefinition(beanName,
                    initializer.getBeanDefinition())
        }
        return beanName
    }
}
```

最后，我们需要通过修改 META-INF/spring.handlers 和 META-INF/spring.schemas 文件，在 Spring XML 基础结构中注册各种工件，如下所示：

```
# in 'META-INF/spring.handlers'  
http://www.foo.example/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'  
http://www.foo.example/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```