Spring2.5的新特性：第一部分

作者 Mark Fisher 译者 沙晓兰 发布于 2008年1月7日 下午7时43分

社区 **Java** 主题 **Web框架** 标签 **Spring框架**

分享 |

简介

从诞生之初，Spring框架就坚守它的宗旨：简化企业级应用开发，同时给复杂问题提供强大的、非侵入性解决方案。一年前发布的Spring2.0就把这些主题推到了一个新的高度。XML Schema的支持和自定义命名空间的使用大大减少了基于XML的配置。使用Java5及更新版本java的开发人员如今可以利用植入了像泛型（generic）和注解等新语言特性的Spring库。最近，和AspectJ表达式语言的紧密集成，使得以非侵入方式添加跨越定义良好的Spring管理对象分组的行为成为可能。

新发布的Spring2.5继续坚持了这个发展趋向，特别是为那些使用Java 5或更新版本java的开发人员提供了进一步简化而强大的新特性。这些新特性包括：注解驱动的依赖性注入（annotation-driven dependency injection），使用注解而非XML元数据来自动侦测classpath上的Spring组件，注解对生命周期方法的支持，一个新的web控制器模型将请求映射到加注解的方法上，在测试框架中支持Junit4，Spring XML命名空间的新增内容，等等。

本文是探讨这些新特性的3篇系列文章中的第一篇。本文将主要关注于简化的配置和在Spring应用程序上下文（application context）核心新增的基于注解的功能；第二篇文章将涵盖web层可用的新特性；最后一篇文章将着重介绍集成和测试的新增性能。这一系列的三篇文章中引用的例子都基于Spring PetClinic应用程序范例。此范例最近被重构以用于展示Spring最新功能，并被包含于Spring 2.5的发布下载包中，可以从Spring Framework 下载网页下载。查看"samples/petclinic"目录下的"readme.txt"文件可以得知关于如何构建和部署PetClinic应用程序，掌握本文提到的新技术的最佳方法也许就是对PetClinic应用程序中所展示的特性进行试验。

Spring支持JSR-250注解

Java EE5中引入了"Java平台的公共注解（Common Annotations for the Java Platform）"，而且该公共注解从Java SE 6一开始就被包含其中。2006年5月，BEA系统宣布了他们在一个名为Pitchfork的项目上与Interface21的合作，该项目提供了基于Spring的Java EE 5编程模型的实现，包括支持用于注入（injection）、拦截（interception）和事务处理（transactions）的JSR-250注解和EJB 3注解(JSR-220)。 在2.5版本中，Spring框架的核心（core）现在支持以下JSR-250注解：

- @Resource
- @PostConstruct
- @PreDestroy

结合Spring，这些注解在任何开发环境下都可以使用——无论是否有应用程序服务器——甚至是集成测试环境都可以。激活这样的支持仅仅是注册一个单独的Spring post-processor的事情：

```
<bean
class="org.springframework.context.annotation.CommonAnnotationBeanPo
/>
```

@Resource注解

@Resource 注解被用来激活一个命名资源（named resource）的依赖注入，在JavaEE应用程序中，该注解被典型地转换为绑定于JNDI context中的一个对象。Spring确实支持使用@Resource通过JNDI lookup来解析对象，默认地，拥有与@Resource注解所提供名字相匹配的"bean name（bean名字）"的Spring管理对象会被注入。 在下面的例子中，Spring会向加了注解的setter方法传递bean名为"dataSource"的Spring管理对象的引用。

```
@Resource(name="dataSource")
 public void setDataSource(DataSource dataSource) {
   this.dataSource = dataSource;
}
```

直接使用@Resource注解一个域（field）同样是可能的。通过不暴露setter方法，代码愈发紧凑并且还提供了域不可修改的额外益处。正如下面将要证明的，@Resource注解甚至不需要一个显式的字符串值，在没有提供任何值的情况下，域名将被当作默认值。

```
@Resource
private DataSource dataSource; // inject the bean named
'dataSource'
```

该方式被应用到setter方法的时候，默认名是从相应的属性衍生出来，换句话说，命名为'setDataSource'的方法被用来处理名为'dataSource'的属性。

```
private DataSource dataSource;
@Resource
public void setDataSource(DataSource dataSource) {
   this.dataSource = dataSource;
}
```

当@Resource没有显式提供名字的时候，如果根据默认名字找不到对应的Spring管理对象，注入机制会回滚至类型匹配（type-match）。如果刚好只有一个Spring管理对象符合该依赖的类型，那么它会被注入。通过设置CommonAnnotationBeanPostProcessor的'fallbackToDefaultTypeMatch'属性为"false"（默认值是"true"）可以禁用这一特性。

```
<bean
class="org.springframework.context.annotation.CommonAnnotationBeanP
>
   <property name="fallbackToDefaultTypeMatch" value="false"/>
</bean>
```

正如上文所提到的，在解析标有@Resource注解的依赖时，Spring支持JNDI-lookup。如若要强制对所有使用@Resource注解的依赖进行JNDI lookup，那也只要将CommonAnnotationBeanPostProcessor的'alwaysUseJndiLookup' 标识设置为true就可以了（默认值是false）。

```
<bean
class="org.springframework.context.annotation.CommonAnnotationBeanP
>
   <property name="alwaysUseJndiLookup" value="true"/>
</bean>
```

另一个选择是，激活指定为'resource-ref-mappings'的依据全局JNDI名的查找，在@Resource注解内提供'mappedName'属性。即使目标对象实际上是一个JNDI资源，仍然推荐引入一个Spring管理对象，这样可以提供一个间接层并且因此降低耦合程度。自Spring2.0开始添加命名空间以来，定义一个委托Spring处理JNDI lookup的bean也变得愈发简练：

```
<jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/petclinic"/>
```

这个方法的优点在于间接层带来了巨大的部署弹性。比如说，一个单独的系统测试环境应该不再需要JNDI注册。在这种情况下，在系统测试配置中可以提供如下的bean定义：

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>
```

顺便提一下，上面的例子中，实际的JDBC连接属性从一个属性文件（properties file）解析而来，在这个属性文件里，关键字与提供的${占位符}互相对应，这需要注册一个名为PropertyPlaceholderConfigurer的BeanFactoryPostProcessor实现来完成。这是具体化那些属性（通常是针对特定环境的属性）常用的技术，这些属性可能比其他配置修改得更为频繁。

```
<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderC
>
   <property name="location" value="classpath:jdbc.properties"/>
</bean>
```

Srping2.5中新加入了'context'命名空间，这个命名空间让我们能够得到更为简洁的方式来实现属性占位符（property placeholder）的配置：

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

William El Kaim，Carlson Wagonlit的首席架构师，详细描述了他与同事们在构建企业的面向服务架构过程中的思考与选择。

SOA，William El Kaim
2010年5月26日

更早的 >

生命周期注解：@PostConstruct和@PreDestroy

@PostConstruct 和@PreDestroy注解分别用来触发Spring的初始化和销毁回调。这个特性在原有基础上得到了扩展，但并没有替代在Spring2.5之前版本中提供的同样的回调的另两个选项。第一个选项是实现Spring的InitializingBean和DisposableBean 接口中的一个或两个。这两个接口都需要一个回调方法的实现(分别是afterPropertiesSet()和destroy())。这种基于接口的方法利用了Spring自动识别任何实现这些接口的Spring管理对象的能力，因而不再需要另外的配置。另一方面，Spring的一个关键目标是尽可能的非侵入。因此，许多Spring用户并不采用实现这些Spring特定接口的方法，而利用第二个选项，那就是提供他们自己的初始化和销毁方法。尽管入侵性小，但缺点在于使用这个方式的话就必须显式声明bean元素的init-method或destroy-method属性。显式配置有时候是必须的，例如当回调需要在开发人员控制能力之外的代码上被调用的时候。PetClinic应用程序很好地说明了这个场景。当它和JDBC配置一起运行的时候，会用到一个第三方DataSource，并且它显式声明了一个destroy-method。另外要注意到的是，单独的连接池数据源是dataSource的另一个部署选项，并且不需要修改任何代码。

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>
```

在使用Spring2.5的过程中，如果一个对象需要调用一个初始化的回调方法的话，这个回调方法可以采用@PostConstruct来注解。例如一个假想的例子，一个后台任务需要在启动的时候就开始对一个文件目录进行轮询：

```
public class FilePoller {

   @PostConstruct
   public void startPolling() {
         ...
   }
   ...
}
```

类似地，一个在Spring管理对象上用@PreDestroy注解的方法会在这个对象寄宿的应用程序上下文（application context）关闭的时候被调用。

```
public class FilePoller {

   @PreDestroy
   public void stopPolling() {
         ...
   }
   ...
}
```

在添加了对JSR-250注解的支持以后，现在的Spring2.5结合前面提到的两种生命周期方法的长处。将@PostConstruct和@PreDestroy作为方法层注解加入，足可以实现在受Spring管理的上下文（context）中触发回调。换句话说，不需要另外基于XML的配置。同时，这两个注解是Java语言本身的一部分（甚至被包括在Java SE版本6中），所以无需引入特定Spring包。这两个注解拥有在其他环境中也能理解的标识语义的优点，随着时间的推移，Java开发人员可能会发现这些注解在第三方开发库中被越来越多的运用到。最后，基于注解生命周期回调的其中一个有趣的结果是，不止一个方法可以带有这两个注解中的任何一个，并且所有注解了的方法会被调用。

激活刚刚描述的关于@Resource、@PostConstruct和@PreDestroy注解的所有行为，正如上文提到的，需要为Spring的CommonAnnotationBeanPostProcessor提供一个bean定义。但另一个更简练的方法则可能是使用2.5中的新的context命名空间：

```
<context:annotation-config/>
```

引入这个单个元素将不单单注册一个CommonAnnotationBeanPostProcessor，也会像下文将叙述的那样激活自动装配（autowire）行为。CommonAnnotationBeanPostProcessor也为@WebServiceRef 和@EJB注解提供支持。这些将在本文系列的第三篇中和Spring2.5为企业集成提供的其他新特性一起讨论。

利用注解来优化细粒度自动装配

涵盖Spring对自动装配支持的文档中常常会提到由于自动装配机制的粗粒度而伴随有很多限制性。Spring2.5之前，自动装配可以通过很多不同的方式来配置：构造器，类型setter，名字setter，或者自动侦测（在该方式中Spring选择自动装配一个构造器或者类型setter）。这些不同的选择确实提供了很大程度的灵活性，但它们中没有一个方

法能够提供细粒度控制。换句话说，Spring2.5之前还不可能自动装配某个对象setter方法的特定子集，或者通过类型或名字来自动装配它的一些属性。结果，许多Spring用户意识到将自动装配应用到构建原型和测试中的好处，但当提到在产品中维护和支持系统时，大部分人认为，加入冗长的显式配置对于澄清它所担负的职责是非常值得的。

然而，Spring2.5大幅度地改变了布局。如上文所述，自动配置选项现在已经被扩展，支持JSR-250 @Resource注解来激活在每个方法或域基础上被命名资源的自动装配。然而，@Resource注解若单独使用的话有很多限制。因此，Sring2.5引进了一个名为@Autowired的注解进一步提高控制级别。为激活这里所讲的行为需要注册一个单独的bean定义：

```
<bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotat
/>
```

另外如上文提到的，context命名空间提供了一个更简明的方法。它将激活本文所讨论的两个post-processor（AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor）和我们在Spring2.0中引入的基于注解的post-processor：RequiredAnnotationBeanPostProcessor和PersistenceAnnotationBeanPostProcessor。

```
<context:annotation-config/>
```

利用@Autowired 注解可以对相应类型注入依赖。域、构造器和方法都可以激活此行为。实际上，aotowired方法并不一定要是setter方法，且可以接受多个参数。下面这个例子是完整的可接受的用法：

```
@Autowired
public void setup(DataSource dataSource, AnotherObject o) { ... }
```

默认地，标有@Autowired注解的依赖被认为是必须的。然而，也可以将required属性值设置为false来声明它们中的任何一个。在下面这个例子中，DefaultStrategy只有在context命名空间中没有SomeStrategy类型的Spring管理对象时才能被使用。

```
@Autowired(required=false)
private SomeStrategy strategy = new DefaultStrategy();
```

通过类型进行的自动装配明显地在Spring context包含多于一个期望类型的对象的时候造成歧义。默认地，如果一个必须的依赖没不是恰好一个bean与之对应的话，自动装配机制就会失败。同样的，对于任何一个可选属性，如果它拥有一个以上的候选，也都会失败（如果属性可选且没有任何候选可用的话，该属性则会被简单地跳过）。有很多不同的配置选项可以避免这些冲突。

若Context中拥有一个指定类型的一个主关键实例，对这个类型定义的bean定义应该包含'primary'属性。当Context中含有其他可用实例的时候这个方法就很适用，但那些非主关键实例总是显式配置的。

```
<bean id="dataSource" primary="true" ... />
```

在需要更多控制的时候，任何autowired的域、构造参数、或者方法参数可以进一步加注@Qualifier注解。qualifier可以包含一个字符串值，在这种情况下，Spring会试图通过名字来找到对应的对象。

```
@Autowired
@Qualifier("primaryDataSource")
private DataSource dataSource;
```

@Qualifier作为一个独立注解存在的主要原因是它可以被应用在构造器参数或方法参数上，但上文提到的@Autowired注解只能运用在构造器或方法本身。

```
@Autowired
public void setup(@Qualifier("primaryDataSource") DataSource
dataSource, AnotherObject o) { ... }
```

事实上，@Qualifier作为一个单独的注解在定制化方面提供了更多的好处。用户自定义的注解在自动装配过程中也可以起到qualifier的作用，最简单的实现方式是在运用自定义注解的同时将@Qualifier作为它的元注解。

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

自定义注解可以选择包含一个值来提供通过名字匹配的功能，但更普遍的用法是将它

作为“标记”注解或定义一个对qualifier过程提供一些更多含义的值。例如，下面这个摘录则描绘了一个域，它应该和通过名字匹配得到的结果中合格的对象进行自动装配。

```
@Autowired
@VetSpecialty("dentistry")
private Clinic dentistryClinic;
```

在使用XML配置来达到依赖解析的目标时，'qualifier' 子元素可以被加注到bean定义中。在下文的组件扫描部分，我们将呈现一个可供选择的非XML方法。

```
<bean id="dentistryClinic" class="samples.DentistryClinic">
   <qualifier type="example.VetSpecialty" value="dentistry"/>
</bean>
```

为了避免对@Qualifier注解的任何依赖性，可以在Spring context中提供一个CustomAutowireConfigurer的bean定义并直接注册所有自定义注解类型：

```
<bean
class="org.springframework.beans.factory.annotation.CustomAutowireC
>
   <property name="customQualifierTypes">
         <set>
               <value>example.VetSpecialty</value>
         </set>
   </property>
</bean>
```

现在，自定义修饰符被显式声明了，就不再需要@Qualifier这个元注解符了。

```
@Target({ElementType.FIELD, ElementType.PARAMETER,
ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface VetSpecialty { ... }
```

其实，在配置AutowiredAnnotationBeanPostProcessor的时候，取代@Autowired注解都是有可能的。

```
<bean
class="org.springframework.beans.factory.annotation.AutowiredAnnotat
>
   <property name="autowiredAnnotationType"
value="example.Injected"/>
</bean>
```

大部分情况下，定义自定义'标记'注解的能力结合通过名字或其他文法值进行匹配选项，足以完成自动装配过程的细粒度控制。但Spring还支持在qualifier注解上任意数目的任意属性。比如，下面是一个极为细粒度修饰的例子。

```
@SpecializedClinic(species="dog", breed="poodle")
private Clinic poodleClinic;
```

自定义修饰符的实现应该定义这些属性：

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

   String species();

   String breed();

}
```

自定义修饰符属性可以匹配那些XML中bean定义的qualifier注解的属性子元素。这些元素通常以键／值对方式提供。

```
<bean id="poodleClinic" class="example.PoodleClinic">
   <qualifier type="example.SpecializedClinic">
         <attribute key="species" value="dog"/>
         <attribute key="breed" value="poodle"/>
   </qualifier>
</bean>
```

目前为止，关于autowire的描述都只是针对单独的实例，其实也支持集合。在任何需要得到所有context中某种特定类型的Spring管理对象的时候，只需要简单地在一个强类型（strongly-typed）集合上加注@Autowired 注解。

```
@Autowired
private List<Clinic> allClinics;
```

本章节最后一个值得指出的特性是自动装配的使用替代了Spring的Aware接口。在Spring2.5之前，如果某个对象需要一个Spring context的ResourceLoader的引用，它可以通过实现ResourceLoaderAware的方式使得Spring通过setResourceLoader(ResourceLoader resourceLoader)方法来提供该依赖。借助同样的方法可以得到Spring管理的MessageSource的引用，甚至可以得到ApplicationContext本身。对于Spring2.5用户而言，这个行为现在通过autowiring得到全面支持（需要指出的是包含这些Spring特定依赖的时候应该考虑周到，特别是它们只能用于从业务逻辑清楚地分割出来的基础构架代码中）。

```
@Autowired
private MessageSource messageSource;

@Autowired
private ResourceLoader resourceLoader;

@Autowired
private ApplicationContext applicationContext;
```

自动侦测Spring组件

从2.0版本开始，Spring引入了构造型（stereotype）注解的概念以及将@Repository注解作为数据访问代码的标记的方法。在此基础上，Spring2.5又加入了两个新的注解 —— @Service和@Controller 来完成为通常的三层架构（数据访问对象、服务、web控制器）角色委任。Spring2.5也引入了泛型@Component注解，其他构造型可从逻辑上对其进行扩展。通过清晰地指明应用程序的角色，这些构造型方便了Spring AOP和post-processor的使用，这些post-processor给基于这些角色的加了注解的对象提供了附加行为。比如，Spring2.0引入了PersistenceExceptionTranslationPostProcessor对任何带有@Repository 注解的对象自动激活其数据访问异常转换。

这些注解同样可以结合Spring2.5其他一些新性能来使用：自动侦测classpath上的组件。尽管XML已经成为最常见的Spring元数据的格式，但它决不是唯一选择。实际上，Spring容器内的元数据是由纯Java来表示的，当XML被用来定义Spring管理对象时，在实例化过程之前，那些定义会被解析并转化成Java对象。Spring2.5的一个巨大的新功能是支持从源码层注解读取元数据。因而，上文描述的自动装配机制使用注解的元数据来注入依赖，但它仍然需要注册至少一个bean定义以便提供每个Spring管理对象的实现类。组件扫描功能则使得这个XML中最起码的bean定义都不再存在需求性。

正如上面所示，Spring注解驱动的自动装配可以在不牺牲细粒度控制的前提下极大程度地减少XML的使用。组件侦测机制将这个优点更发扬光大。全面替代XML中的配置不再必要，组件扫描反而可以处理XML元数据来简化整体配置。结合XML和注解驱动技术可以得到一个平衡优化的方法，这在2.5版本的PetClinic范例中有详细阐述。在该范例中，基础构架组件（数据源、事务管理等）结合上文提到的外化属性在XML中定义。数据访问层对象也有部分在XML中定义，它们的配置也都利用了@Autowired注解来简化依赖注入。最后，web层控制器完全不在XML中显式定义，相反，下面提供的这段配置被用来触发所有web控制器的自动侦测：

```
<context:component-scan base-
package="org.springframework.samples.petclinic.web"/>
```

需要注意到的是这段示例中使用到了base-package属性。组件扫描的默认匹配规则会递归侦测该包（多个包可以以逗号分隔的list方式提供）内的所有类的所有Spring构造型注解。正因为如此，PetClinic应用程序范例中的各类控制器的实现都采用了@Controller注解（Spring的内置构造型之一）。请看下面这个例子：

```
@Controller
public class ClinicController {

   private final Clinic clinic;

   @Autowired
   public ClinicController(Clinic clinic) {
         this.clinic = clinic;
   }
...
```

自动侦测组件在Spring容器中注册，就像它们在XML中被定义一样。如上所示，那些对象可以轮流利用注解驱动的自动装配。

组件扫描的匹配规则可以通过过滤器（filter）来自定义，以根据类型、AspectJ表达式、或针对命名模式的正则表达式来决定包含或不包含哪些组件。默认的构造型也可以被禁用。比如这里有一个配置的例子，这个配置会忽略默认的构造型，但会自动侦测名字以Stub打头或者包含@Mock注解的所有类：

```
<context:component-scan base-package="example" use-default-
filters="false">
   <context:include-filter type="aspectj"
expression="example..Stub*"/>
```

```
   <context:include-filter type="annotation"
expression="example.Mock"/>
</context:component-scan>
```

类型匹配的限制性也可以用排他的过滤器控制。例如，除了@Repository注解外其他都依赖于默认过滤器，那么就需要加入一个排他过滤器(exclude-filter)。

```
<context:component-scan base-package="example">
   <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

很明显，有很多方法可以扩展组件扫描来注册自定义的类型。构造型注解是最简单的选择，所以构造型概念本身也是可扩展的。像先前提到的，@Component是泛型模型，@Repository、@Service,和@Controller注解都从该构造型逻辑扩展而得。正因为如此，@Component可被用来作为元注解（也就是说，在另外的注解上声明的注解），所有具有@Component元注解的自定义注解都会被默认扫描匹配规则自动侦测到。一个例子就有希望让你领会到其实它根本没有听起来那么难。

让我们回想一下在讲@PostConstruct和@PreDestroy生命周期注解的时候的假想的后台任务。也许一个应用程序有很多很多这样的后台任务，这些任务实例需要XML bean定义以便在Spring context里注册并使它们自己的生命周期方法在正确时候被调用。利用组件扫描就不再需要这些显式的XML bean定义。如果这些后台任务都实现一个相同的接口或者都沿用同样的命名惯例，那么可以用include-filters。然而，更简单的方法是为这些任务对象创建一个注解并提供@Component元注解。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface BackgroundTask {
   String value() default "";
}
```

然后在所有后台任务的类定义中提供自定义构造型注解。

```
@BackgroundTask
public class FilePoller {

   @PostConstruct
   public void startPolling() {
         ...
   }

   @PreDestroy
   public void stopPolling() {
         ...
   }
   ...
}
```

泛型@Component注解可以像例子中提供的那样简单使用，自定义注解技术则提供了一个使用更具涵义的、领域特定的名字的机会。这些领域特定注解提供更深入的机会，比如使用AspectJ切点表达式来识别所有后台任务，以便增加advice来监控这些任务的活动性。

默认的，组件被侦测到的时候，Spring会自动生成一个没有修饰符的类名作为bean名字。上一个例子中，生成的bean名字会是filePoller。但是，任何加注了Spring构造型注解（@Component、@Repository、@Service或 @Controller）或是加注了其他的以@Component作为元注解的注解（比如上面例子中的@BackgroundTask）的类，构造型注解的value属性可以被显式指定，实例将该值作为它的bean名字注册到context中。接下来的例子里，实例名应该是petClinic而不是默认生成的名字simpleJdbcClinic。

```
@Service("petClinic")
public class SimpleJdbcClinic {
   ...
}
```

同样的，在下面修正版的FilePoller例子里，生成的bean名字应该是poller而不是filePoller。

```
@BackgroundTask("poller")
 public class FilePoller {
   ...
}
```

虽然所有Spring管理对象都被默认地当作单例实例来处理，但有些时候还是有必要为某个对象指明一个备用的范围（scope）。举个例子来说，在web层，一个Spring管理对象可能捆绑到request或session的范围。对于2.0版本，Spring的scope机制更具延展性，这样一来，自定义scope可以被注册到应用程序上下文（application

context）。在XML配置中，仅仅是简单地包含进scope属性及该scope的名字就可以了。

```
<bean id="shoppingCart" class="example.ShoppingCart"
scope="session">
   ...
</bean>
```

Spring2.5中，为被扫描的组件提供@Scope注解可以起到同样的作用。

```
@Component
@Scope("session")
public class ShoppingCart {
   ...
}
```

这里要指出的最后一点是使用组件扫描时qualifier注解应用是多么的简单。在上一节，下面这个对象曾被作为使用自定义qualifier注解进行自动装配的例子：

```
@VetSpecialty("dentistry")
private Clinic dentistryClinic;
```

同样的例子接着展现了在XML内使用'qualifier'元素为依赖提供指定目标bean定义。在使用组件扫描时，XML元数据不是必须的。但自定义修饰符也许在目标类定义中被作为类型层注解而引入。另一个将被扫描的@Repository实例作为依赖的例子如下：

```
@Repository
@VetSpecialty("dentistry")
public class DentistryClinic implements Clinic {
   ...
}
```

最终，因为前面的例子展现了自定义注解及其属性的例子，相等同的非XML表示依赖目标的方法如下：

```
@Repository
@SpecializedClinic(species="dog", breed="poodle")
public class PoodleClinic implements Clinic {
   ...
}
```

小结

Spring2.5在很多方面都提供了很有意义的新功能。本文主要关注于怎样通过掌控Java注解的力量将配置简化。就如在JSR-250中定义的那样，Spring支持公共注解（Common Annotations），同时为自动装配过程的更细粒度的控制提供了额外注解。Spring2.5也扩展了从Spring2.0的@Repository就开始的构造型（stereotype）注解，并且所有这些构造型注解都可以和新的组件扫描功能结合使用。Spring2.5仍然全面支持基于XML的配置，同时它又引进了一个新的context命名空间对常见配置场景提供更精要的文法。实际上，支持XML和基于注解配置的无缝结合最终产生一个更为平衡的全面的方法。基本构架的复杂配置可以在模块XML文件中定义，而应用程序栈日益增多地更高层配置可以更多的从基于注解的技术中获益——前提是都在同一个Spring2.5应用程序context内。

在接下来的文章中，我们将讨论到在Spring web层强大的基于注解的新功能项。敬请关注该系列的下一篇文章。

查看英文原文：What's New in Spring 2.5: Part 1

10 条回复　　关注此讨论　　回复

Spring2.5的新特性：第一部分 发表人 sw pollux 发表于 2008年1月7日 下午11时8分

- Re: Spring2.5的新特性：第一部分 发表人 sw pollux 发表于 2008年1月7日 下午11时36分
- Spring2.5的新特性：第一部分 发表人 陈 昌瀚 发表于 2008年1月14日 下午9时23分
  - Re: Spring2.5的新特性：第一部分 发表人 edon wei 发表于 2008年1月16日 上午1时39分
    - Re: Spring2.5的新特性：第一部分 发表人 taoist war 发表于 2009年2月10日 上午3时25分

链接无法打开 发表人 Qu Jinlong 发表于 2008年1月8日 上午2时30分

- Re: 链接无法打开 发表人 玮 宋 发表于 2008年1月8日 上午2时52分

不支持基本类型注入 发表人 Andy Yao 发表于 2008年1月8日 下午11时41分

Spring2.5的新特性：第一部分 发表人 feng xiao 发表于 2008年1月9日 下午8时41分

其他的part那？ 发表人 micheal lee 发表于 2009年6月11日 上午1时56分

按日期倒序排列

Spring2.5的新特性：第一部分

2008年1月7日 下午11时8分 发表人 sw pollux

感觉现在infoq cn翻译速度也在加快!
赞一个!

回复

Re: Spring2.5的新特性：第一部分

2008年1月7日 下午11时36分 发表人 sw pollux

配置段好象不完整，都丢了 <bean 这个部分喔 呵呵>

回复

链接无法打开

2008年1月8日 上午2时30分 发表人 Qu Jinlong

【阅读全文：Spring2.5的新特性：第一部分】的链接无法打开，
找不到此页。

Oops! The server couldn't find the requested information (/zh/articles/spring-2.5-part-1).

回复

Re: 链接无法打开

2008年1月8日 上午2时52分 发表人 玮 宋

感谢sw pollux的提醒，发布时出了点问题，已经好了。
刚才正在修改sw pollux所提的问题，因此无法打开链接

回复

不支持基本类型注入

2008年1月8日 下午11时41分 发表人 Andy Yao

目前版本不支持基本类型注入，
而且在Annotation不支持XML配置方式中的PropertyPlaceholderConfigurer。

回复

Spring2.5的新特性：第一部分

2008年1月9日 下午8时41分 发表人 feng xiao

太好了，感觉2.5使用也越来越简单了！！！
希望可以有完整版本出来！期待中！！！

回复

Spring2.5的新特性：第一部分

2008年1月14日 下午9时23分 发表人 陈 昌瀚

这真是好东西啊

回复

Re: Spring2.5的新特性：第一部分

2008年1月16日 上午1时39分 发表人 edon wei

学习中。

回复

Re: Spring2.5的新特性：第一部分

2009年2月10日 上午3时25分 发表人 taoist war

想问下如果用Spring2.5的新特性：自动扫描,没有了配置文件，那spring还有什么灵活性可言呢？

回复

其他的part那？

2009年6月11日 上午1时56分 发表人 micheal lee

不是有三篇吗？怎么就一文那？

回复