## 什么是spring？

Spring 是分层的 Java SE/EE full-stack 轻量级开源框架，以 IoC（Inverse of Control，控制反转）和 AOP（Aspect Oriented Programming，面向切面编程）为内核，使用基本的 JavaBean 完成以前只可能由 EJB 完成的工作，取代了 EJB 臃肿和低效的开发模式。在实际开发中，通常服务器端采用三层体系架构，分别为表现层（web）、业务逻辑层（service）、持久层（dao）。Spring 对每一层都提供了技术支持，在表现层提供了与 [Struts2](http://c.biancheng.net/struts2/) 框架的整合，在业务逻辑层可以管理事务和记录日志等，在持久层可以整合 [Hibernate](http://c.biancheng.net/hibernate/) 和 JdbcTemplate 等技术。正是因为这些特性，[Spring](http://c.biancheng.net/spring/) 是一个轻量级、功能强大、性能卓越更自由的开发应用框架，吸引了大批的程序员，迅速成为主流的 [Java](http://c.biancheng.net/java/) Web 开发框架，并且被大部分公司作为 Java 项目开发的首选框架。

#### Spring 框架的主要优点

具体如下。

**1）方便解耦，简化开发**

Spring 就是一个大工厂，可以将所有对象的创建和依赖关系的维护交给 Spring 管理。

**2）方便集成各种优秀框架**

Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如 Struts2、Hibernate、MyBatis 等）的直接支持。

**3）降低 Java EE API 的使用难度**

Spring 对 Java EE 开发中非常难用的一些 API（JDBC、JavaMail、远程调用等）都提供了封装，使这些 API 应用的难度大大降低。

**4）方便程序的测试**

Spring 支持 JUnit4，可以通过注解方便地测试 Spring 程序。

**5）AOP 编程的支持**

Spring 提供面向切面编程，可以方便地实现对程序进行权限拦截和运行监控等功能。

**6）声明式事务的支持**

只需要通过配置就可以完成对事务的管理，而无须手动编程。

#### 什么是IOC DI？

控制反转，（Inversion of Control，缩写为IoC），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。IoC不是一种技术， 只是一种思想，能够指导我们设计出更优良的程序。传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间的高耦合，难于测试；有了IoC容器后，创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，从而降低了对象之间的耦合度，同时也方便测试，利于功能复用，并使得程序的整体结构变得更加灵活。

其实说白了，将所有的模块交个一个第三方控制，我只需要满足第三方的需求能够被管理，那么和其他的模块的交汇就可以统一被这个第三发处理。Spring就是充当这个第三方，不仅管理对象的生命周期，还提供了全部对象间的依赖自动查找注入。这样的思想其实很容易明白，就是各个模块如果直接拼凑在一起使用，就需要耦合，需要两个连接方互相配合，就像我们公司前期的PLM和NC的接口一样，如果NC做了业务调整，接口调整，PLM的接口服务就需要做相应的调整，如果有更多的部门，那么这些部门都需要对业务进行调整，这样带来的难处就是，做一个小改动，需要各个使用方同时协作调整，大大增加了开发功能、发展业务的难度。

DI—Dependency Injection，即“依赖注入”。如果说IOC偏向指是一种解耦的思想，那么DI就是实现这种思想如何反转控制的一种方式。既然反转控制需要将控制权交给第三方，那么，第三方如何代理组件间的耦合呢？答案就是依赖注入，Spring解析我们声明的依赖，再通过Spring容器将依赖主动注入到对象（一般是对象主动去获取依赖，现在是容器主动将依赖注入给对象，所以称之为反转），Spring就是一个对象容器，管理这个对象的生命周期，管理着对象之间的依赖，当我们获取对象，检查对象的依赖关系，反向间依赖的对象交给获取的对象（被称为注入），对象就通过这样的方式获取所需要的对象外部资源（包括对象、资源、常量数据）。其实这也是使用代理的基本思想，Spring代理了对象的包括创建的生命周期，代理的对象创建过程中，统一给所有的对象增加一道创建工序：解析依赖并注入。这样的好处就是程序员只需声明依赖，不需要自己去管理对象。

#### 什么是AOP？

AOP（Aspect Oriented Programming），即面向切面编程，通过[预编译](https://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91/3191547)方式和运行期间动态代理实现程序功能的统一维护的一种技术，可以说是OOP（Object Oriented Programming，面向对象编程）的补充和完善。OOP的一个弊端就是所有的功能都需要在继承关系中去实现，继承是一种垂直的关系，如果业务从垂直关系中去管理，那么就需要在所有的管理对象去增加业务逻辑，重复且不易管理，这时候就需要提炼**一个独立于继承关系之外功能入口，这就是切面（这个厉害了，我自创的说法）**。如日志记录，性能统计，安全控制，事务处理，[异常处理](https://baike.baidu.com/item/%E5%BC%82%E5%B8%B8%E5%A4%84%E7%90%86)等等功能，在继承关系中去实现使用接口是最合适的，但是需要在每个被管控的对象中实现功能，这样造成冗余，面向切面编程就是，开辟一个切面，找到目标对象的功能的触发点（切点），插入新功能的模块代码。

AOP核心概念：

1、**横切关注点**

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横切关注点

2、**切面（aspect）**

类是对物体特征的抽象，切面就是对横切关注点的抽象

3、**连接点（joinpoint）**

被拦截到的点，因为Spring只支持方法类型的连接点，所以在Spring中连接点指的就是被拦截到的方法，实际上连接点还可以是字段或者构造器

4、**切入点（pointcut）**

对连接点进行拦截的定义

5、**通知（advice）**

所谓通知指的就是指拦截到连接点之后要执行的代码，通知分为前置、后置、异常、最终、环绕通知五类

6、**目标对象** 代理的目标对象

7、**织入（weave）**

将切面应用到目标对象并导致代理对象创建的过程

8、**引入（introduction）**

在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

**Spring中AOP代理**由Spring的IOC容器负责生成、管理，其依赖关系也由IOC容器负责管理。因此，AOP代理可以直接使用容器中的其它bean实例作为目标，这种关系可由IOC容器的依赖注入提供。Spring创建代理的规则为：

1、默认使用Java动态代理来创建AOP代理，这样就可以为任何接口实例创建代理了

2、当需要代理的类不是代理接口的时候，Spring会切换为使用CGLIB代理，也可强制使用CGLIB

AOP编程，程序员只需要参与三个部分：

1、定义普通业务组件（bean）

2、定义切入点，一个切入点可能横切多个业务组件

3、定义增强处理，增强处理就是在AOP框架为普通业务组件织入的处理动作

## Spring架构

S[pring](http://c.biancheng.net/spring/) 框架采用分层架构，根据不同的功能被划分成了多个模块，这些模块大体可分为 Data Access/Integration、Web、AOP、Aspects、Messaging、Instrumentation、Core Container 和 Test。（来源于资料）



#### Data Access/Integration（数据访问／集成）

数据访问/集成层包括 JDBC、ORM、OXM、JMS 和 Transactions 模块，具体介绍如下。

|  |  |
| --- | --- |
| JDBC 模块 | 提供了一个 JDBC 的抽象层，大幅度减少了在开发过程中对数据库操作的编码 |
| ORM 模块 | 对流行的对象关系映射 API，包括 JPA、JDO、[Hibernate](http://c.biancheng.net/hibernate/) 和 iBatis 提供了的集成层 |
| OXM 模块 | 提供了一个支持对象/XML 映射的抽象层实现，如 JAXB、Castor、XMLBeans、JiBX 和 XStream |
| JMS 模块 | 指 [Java](http://c.biancheng.net/java/) 消息服务，包含的功能为生产和消费的信息 |
| 事务模块 | 支持编程和声明式事务管理实现特殊接口类，并为所有的 POJO |

#### Web 模块

Spring 的 Web 层包括 Web、[Servlet](http://c.biancheng.net/servlet/)、Struts 和 Portlet 组件，具体介绍如下。

|  |  |
| --- | --- |
| Web 模块 | 提供了基本的 Web 开发集成特性，例如多文件上传功能、使用的 Servlet 监听器的 IoC 容器初始化以及 Web 应用上下文 |
| Servlet模块 | 包括 Spring 模型—视图—控制器（MVC）实现 Web 应用程序 |
| Struts 模块 | 包含支持类内的 Spring 应用程序，集成了经典的 Struts Web 层 |
| Portlet 模块 | 提供了在 Portlet 环境中使用 MV C实现，类似 Web-Servlet 模块的功能 |

#### Core Container（核心容器）

Spring 的核心容器是其他模块建立的基础，由 Beans 模块、Core 核心模块、Context 上下文模块和 Expression Language 表达式语言模块组成，具体介绍如下。

|  |  |
| --- | --- |
| Beans 模块 | 提供了 BeanFactory，是工厂模式的经典实现，Spring 将管理对象称为 Bean |
| Core 模块 | 提供了 Spring 框架的基本组成部分，包括 IoC 和 DI 功能 |
| Context 模块 | 建立在核心和 Beans 模块的基础之上，它是访问定义和配置任何对象的媒介。ApplicationContext 接口是上下文模块的焦点 |
| Expression Language 模块 | 是运行时查询和操作对象图的强大的表达式语言 |

#### 其他模块

Spring的其他模块还有 AOP、Aspects、Instrumentation 以及 Test 模块，具体介绍如下。

AOP 模块：提供了面向切面编程实现，允许定义方法拦截器和切入点，将代码按照功能进行分离，以降低耦合性。

|  |  |
| --- | --- |
| Aspects 模块 | 提供与 AspectJ 的集成，是一个功能强大且成熟的面向切面编程（AOP）框架 |
| Instrumentation 模块 | 提供了类工具的支持和类加载器的实现，可以在特定的应用服务器中使用 |
| Test 模块 | 支持 Spring 组件，使用 JUnit 或 TestNG 框架的测试 |

## Spring的IOC容器

IoC 是指在程序开发中，实例的创建不再由调用者管理，而是由 Spring 容器创建。Spring 容器会负责控制程序之间的关系，而不是由程序代码直接控制，因此，控制权由程序代码转移到了 Spring 容器中，控制权发生了反转。Spring 提供了两种 IoC 容器，分别为 BeanFactory 和 ApplicationContext。（二者的主要区别在于，BeanFactory不检查依赖是否注入，而ApplicationContext 则在初始化时自检，这样有利于检查所依赖的属*性是否注入*）。

IOC容器其实就是Spring管理Bean生命周期的容器，包括初始化Bean，管理Bean，销毁Bean，并处理Bean之间的依赖耦合关系。虽然获取对象实例简化为简单的一句，但是Spring后台做的操作是非常庞大的步骤，解析，对象管理池，依赖关系管理池等等，这样的具体步骤我还没有很清楚的理清楚，后续实际使用Bean的时候可以深入调研一下。

#### BeanFactory工厂

BeanFactory 是基础类型的 IoC 容器，它由 (org.springframework.beans.facytory) BeanFactory 接口定义，并提供了完整的 IoC 服务支持。简单来说，BeanFactory 就是一个管理 Bean 的工厂，它主要负责初始化各种 Bean，并调用它们的生命周期方法。BeanFactory 接口有多个实现类，最常见的是xml.XmlBeanFactory，它是根据 XML 配置文件中的定义装配 Bean 的。

创建 BeanFactory 实例时，需要提供 Spring 所管理容器的详细配置信息，这些信息通常采用 XML 文件形式管理。其加载配置信息的代码具体如下所示：

BeanFactory beanFactory = new XmlBeanFactory(new FileSystemResource("D://applicationContext.xml"));

#### ApplicationContext上下文

ApplicationContext 是 BeanFactory 的子接口，也被称为应用上下文。该接口的全路径为 org.springframework.context.ApplicationContext，它不仅提供了 BeanFactory 的所有功能，还添加了对 i18n（国际化）、资源访问、事件传播等方面的良好支持。ApplicationContext 接口有两个常用的实现类，具体如下。

**1）ClassPathXmlApplicationContext**

该类从类路径 ClassPath 中寻找指定的 XML 配置文件，找到并装载完成 ApplicationContext 的实例化工作，具体如下所示。

ApplicationContext applicationContext = new ClassPathXmlApplicationContext(String configLocation);

在上述代码中，configLocation 参数用于指定 Spring 配置文件的名称和位置，如 applicationContext.xml。

**2）FileSystemXmlApplicationContext**

该类从指定的文件系统路径中寻找指定的 XML 配置文件，找到并装载完成 ApplicationContext 的实例化工作，具体如下所示。

ApplicationContext applicationContext = new FileSystemXmlApplicationContext(String configLocation);

它与 ClassPathXmlApplicationContext 的区别是：在读取 Spring 的配置文件时，FileSystemXmlApplicationContext 不再从类路径中读取配置文件，而是通过参数指定配置文件的位置，它可以获取类路径之外的资源，如“F：/workspaces/applicationContext.xml”。

## Spring Bean

JAVA中everything is A object，任何东西都是一个对象，那么Spring需要托管对象的一切，这种被声明由Spring容器托管的所有对象称之为Bean。Spring是Java概念下延展出来的对象托管容器，spring将被托管的对象称之为Bean。

#### Bean配置

Bean的配置前期是使用文件配置，后面全面升级使用注解配置，使用注解可以实现XML配置的同样的功能（只需要配置容器扫描的类，解析注解即可，但是对于入门XML更容易理解，所以使用XML为例介绍装配）。XML 格式配置文件的根元素是 <beans>，该元素包含了多个 <bean> 子元素，每一个 <bean> 子元素定义了一个 Bean，并描述了该 Bean 如何被装配到 Spring 容器中。

定义 Bean 的示例代码如下所示：

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

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

<!-- 使用id属性定义person1，其对应的实现类为com.mengma.person1 -->

<bean id="person1" class="com.mengma.damain.Person1" />

<!--使用name属性定义person2，其对应的实现类为com.\*.Person2-->

<bean name="Person2" class="com.\*.Person2"/>

</beans>

Bean标签还有很多其他的属性：

|  |  |
| --- | --- |
| **属性名称** | **描述** |
| id | 是一个 Bean 的唯一标识符，Spring 容器对 Bean 的配置和管理都通过该属性完成 |
| name | Spring 容器同样可以通过此属性对容器中的 Bean 进行配置和管理，name 属性中可以为 Bean 指定多个名称，每个名称之间用逗号或分号隔开 |
| class | 该属性指定了 Bean 的具体实现类，它必须是一个完整的类名，使用类的全限定名 |
| scope | 用于设定 Bean 实例的作用域，其属性值有 singleton（单例）、prototype（原型）、request、session 和 global Session。其默认值是 singleton |
| constructor-arg | <bean>元素的子元素，可以使用此元素传入构造参数进行实例化。该元素的 index 属性指定构造参数的序号（从 0 开始），type 属性指定构造参数的类型 |
| property | <bean>元素的子元素，用于调用 Bean 实例中的 Set 方法完成属性赋值，从而完成依赖注入。该元素的 name 属性指定 Bean 实例中的相应属性名 |
| ref | <property> 和 <constructor-arg> 等元素的子元索，该元素中的 bean 属性用于指定对 Bean 工厂中某个 Bean 实例的引用 |
| value | <property> 和 <constractor-arg> 等元素的子元素，用于直接指定一个常量值 |
| list | 用于封装 List 或数组类型的依赖注入 |
| set | 用于封装 Set 类型属性的依赖注入 |
| map | 用于封装 Map 类型属性的依赖注入 |
| entry | <map> 元素的子元素，用于设置一个键值对。其 key 属性指定字符串类型的键值，ref 或 value 子元素指定其值 |

#### Bean实例化

如果没有指定配置Bean的构造方法，Spring会调用默认的无参构造器（也可以不用写这个无参构造器）；如果需要静态工厂方法生成实例，就需要单独配置factory-method（类默认为Bean的类）；如果是使用实例工厂方法，则需要使用factory-bean，factory-method指定工厂类和工厂方法。配置区别如下：

只指定bean的类路径，表示使用无参构造器

<bean id="person1" class="com.\*.Person1" />

静态工厂方法方式生成实例，指定工厂类和静态方法

<bean id="person2" class="com.\*.MyBeanFactory" factory-method="createBean" />

实例工厂方法方式生成实例，指定工厂Bean和工厂方法

<bean id="myBeanFactory" class="com.mengma.instance.factory.MyBeanFactory" />

<bean id="person3" factory-bean="myBeanFactory" factory-method="createBean" />

#### Bean作用域

Spring 容器在初始化一个 Bean 的实例时，同时会指定该实例的作用域。Spring3 为 Bean 定义了五种作用域，具体如下。（不同作用域的区别不仅仅是带有的状态信息，还有生命周期时间节点不同，尤其是对象销毁时间）

**1）singleton（无状态）**

单例模式，使用 singleton 定义的 Bean 在 Spring 容器中只有一个实例，这也是 Bean 默认的作用域。这种单例模式对于无会话状态的 Bean（如 DAO 层、Service 层）来说，是最理想的选择

**2）prototype（有状态）**

原型模式，每次通过 Spring 容器获取 prototype 定义的 Bean 时，容器都将创建一个新的 Bean 实例。对需要保持会话状态的 Bean（如 [Struts2](http://c.biancheng.net/struts2/) 的 Action 类）应该使用 prototype 作用域

**3）request（WEB 有HTTP状态）**

在一次 HTTP 请求中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Request 内有效。

**4）session（WEB有会话状态，一次会话可能多次HTTP请求）**

在一次 HTTP Session 中，容器会返回该 Bean 的同一个实例。而对不同的 HTTP 请求，会返回不同的实例，该作用域仅在当前 HTTP Session 内有效。

**5）****global Session（WEB 服务启动后所有的会话都能使用？）**

在一个全局的 HTTP Session 中，容器会返回该 Bean 的同一个实例。该作用域仅在使用 portlet context 时有效。

#### Bean生命周期

对于Spring中的Bean来说，程序员是不能直接干预Bean的生命周期的，因为Bean已经完全托管给了容器处理，上面讲到Bean的作用域，不同作用域的范围，生命周期的管理也不同。

**创建**：根据配置情况调用Bean构造方法或工厂方法实例化Bean，利用依赖注入完成Bean中所有属性值的配置注入，这是创建Bean的基本流程。下面罗列一些额外的创建的流程：

如果Bean实现了BeanNameAware接口，则Spring调用Bean的setBeanName()方法传入当前Bean的id值。

如果 Bean 实现了 BeanFactoryAware 接口，则 Spring 调用 setBeanFactory() 方法传入当前工厂实例的引用。

如果Bean实现了ApplicationContextAware接口，则Spring调用setApplicationContext() 方法传入当前ApplicationContext实例的引用。

如果BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的预初始化方法 postProcessBeforeInitialzation() 对 Bean 进行加工操作，此处非常重要，Spring 的 AOP 就是利用它实现的。

如果 Bean 实现了 InitializingBean 接口，则 Spring 将调用 afterPropertiesSet() 方法

如果在配置文件中通过 init-method 属性指定了初始化方法，则调用该初始化方法。

如果 BeanPostProcessor 和 Bean 关联，则 Spring 将调用该接口的初始化方法 postProcessAfterInitialization()

**销毁**：对于单例模式下的Bean，Spring能准确知道Bean何时被创建，何时被销毁，所以单例模式的Bean全权由Spring容器创建销毁；对于原型模式下的Bean，Spring创建后，交给Bean的调用者，由调用者销毁。**Spring还提供指定销毁的方式**：其一如果 Bean 实现了 DisposableBean 接口，则 Spring 会调用 destory() 方法将 Spring 中的 Bean 销毁；其二在配置文件中通过 destory-method 属性指定了 Bean 的销毁方法，则 Spring 将调用该方法对 Bean 进行销毁。

## Spring XML配置和注解

Spring 配置文件支持两种不同的格式，分别是 XML 文件格式和 Properties 文件格式。Properties一般用于配置值。Xml用于配置Bean。

#### XML配置

XML的配置有头声明和Beans主成，其中Beans里是核心的配置内容，其下常见的标签有以下几个：Bean、Context、tx:\*、AOP:\*，其中bean是基本内容，context配置Spring环境的上下文，用于配置Spring的配置文档等，tx:\*配置的都是事务管理的属性（声明式的事务管理需要配置一个transactionManager的Bean），aop:\*配置切面相关的内容，指定切面，切入点和通知目标。

Beans元素：配置的根节点，属性有default-lazy-init是否懒加载，default-dependency-check依赖检查，default-autowire自动装配。子元素有两个：import导入环境中的资源如配置文件，bean配置Bean。

Bean元素：配置bean的标签，ID/name/class/parent基本属性，factory-class/factory-method可以指定构造方法，scope指定作用域，property指定bean的属性。

Tx元素配置Spring内置的事务管理器的属性。

Aop元素，配置定义切面，面向切面编程会详说。

#### 注解

在 [Spring](http://c.biancheng.net/spring/) 中，尽管使用 XML 配置文件可以实现 Bean 的装配工作，但如果应用中 Bean 的数量较多，会导致 XML 配置文件过于臃肿，从而给维护和升级带来一定的困难。[Java](http://c.biancheng.net/java/) 从 JDK 5.0 以后，提供了 Annotation（注解）功能，Spring 也提供了对 Annotation 技术的全面支持。Spring3 中定义了一系列的 Annotation（注解），常用的注解如下。

|  |  |
| --- | --- |
| 注解 | 描述 |
| @Component | 可以使用此注解描述 Spring 中的 Bean，但它是一个泛化的概念，仅仅表示一个组件（Bean），并且可以作用在任何层次。使用时只需将该注解标注在相应类上即可。 |
| @Repository | 用于将数据访问层（DAO层）的类标识为 Spring 中的 Bean，其功能与 @Component 相同。 |
| @Service | 通常作用在业务层（Service 层），用于将业务层的类标识为 Spring 中的 Bean，其功能与 @Component 相同。 |
| **@Controller** | 通常作用在控制层（如 [Struts2](http://c.biancheng.net/struts2/) 的 Action），用于将控制层的类标识为 Spring 中的 Bean，其功能与 @Component 相同。 |
| @Autowired | 用于对 Bean 的属性变量、属性的 Set 方法及构造函数进行标注，配合对应的注解处理器完成 Bean 的自动配置工作。默认按照 Bean 的类型进行装配。自动装配 |
| @Resource | 其作用与 Autowired 一样。其区别在于 @Autowired 默认按照 Bean 类型装配，而 @Resource 默认按照 Bean 实例名称进行装配。 @Resource 中有两个重要属性：name 和 type。 Spring 将 name 属性解析为 Bean 实例名称，type 属性解析为 Bean 实例类型。如果指定 name 属性，则按实例名称进行装配；如果指定 type 属性，则按 Bean 类型进行装配。 如果都不指定，则先按 Bean 实例名称装配，如果不能匹配，则再按照 Bean 类型进行装配；如果都无法匹配，则抛出 NoSuchBeanDefinitionException 异常。 |
| @Qualifier | 与 @Autowired 注解配合使用，会将默认的按 Bean 类型装配修改为按 Bean 的**实例名称装配**，Bean 的实例名称由 @Qualifier 注解的参数指定。 |
| @Required | 此注解用于JavaBean的setter方法上，表示此属性是必须的，必须在配置阶段注入，否则会抛出BeanInitializationException。 |
| @configuration | 配置 |
| @lazy | 此注解使用在Spring的组件类上。默认的，Spring中Bean的依赖一开始就被创建和配置。如果想要延迟初始化一个bean，那么可以在此类上使用Lazy注解，表示此bean只有在第一次被使用的时候才会被创建和初始化。 |

还有比较常用的注解，自动生成Getter和Setter，虽然不是Spring自带的，但是可以简化代码。

## Spring依赖注入的实现方式

Spring依赖注入是概念方面的描述，对Bean对象来说就叫装配，Spring通过依赖注入的方式对Bean管理进行资源装配（创建应用对象之间的协作关系），其实依赖注入就是装配的意思。Bean装配的方式有三种：自动装配，代码显式装配和XML显式装配（按照推荐度由高到低）；

#### 自动装配

Spring从两个角度来实现自动化装配；组件扫描（Spring自动发现应用中上下文所需要的创建的Bean），自动装配（Spring自动满足Bean之间的依赖）。自动装配需要匹配类路径或者Bean名称，可以在注解中设置属性。

使用@Component将普通Java类配置成SpringBean；使用@Autowired（自动装配）使Spring满足Bean的依赖。注解好后，使用注解@ComponentScan提示Spring扫描包自动装配，等同于XML配置context:component-scan指定包路径。

#### 代码显式装配

我们也可以在Spring的Java配置类中对SpringBean进行配置，配置方法的返回值为Bean。同时MyArticle也被依赖注入到User中。

@Configuration

@ComponentScan("com.jimisun")

public class WebConfig {

@Bean

public User user() {

return new User(myArticle());

}

@Bean

public MyArticle myArticle() {

return new MyArticle();

}

}

#### XML显式装配

通过XML标签进行显示装配,使用构造参数标签，要求需要装配myArticle Bean，实现依赖注入。XML配置的通病就是，当项目到一定的复杂度后，XML会非常的庞大，不好维护。所以推荐度最低。

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

<beans …>

<bean class="com.\*.User" id="user">

<constructor-arg ref="myArticle"/>

</bean>

<bean class="com.\*.MyArticle" id="myArticle"></bean>

</beans>

## Spring AOP和Aspectj

目前最流行的 AOP 框架有两个，分别为 Spring AOP 和 AspectJ。Spring AOP 使用纯 [Java](http://c.biancheng.net/java/) 实现，不需要专门的编译过程和类加载器，在运行期间通过代理方式向目标类植入增强的代码。AspectJ 扩展了 Java 语言，提供了一个专门的编译器，在编译时提供横向代码的植入。切面编程的基本概念如下：

|  |  |
| --- | --- |
| **名称** | **说明** |
| Joinpoint（连接点） | 指那些被拦截到的点，在 Spring 中，可以被动态代理拦截目标类的方法。 |
| Pointcut（切入点） | 指要对哪些 Joinpoint 进行拦截，即被拦截的连接点。 |
| Advice（通知） | 指拦截到 Joinpoint 之后要做的事情，即对切入点增强的内容。 |
| Target（目标） | 指代理的目标对象。 |
| Weaving（植入） | 指把增强代码应用到目标上，生成代理对象的过程。 |
| Proxy（代理） | 指生成的代理对象。 |
| Aspect（切面） | 切入点和通知的结合。 |

#### JDK动态代理

下面举一个AOP的例子，展示如何使用代理的方式，在特定的切入点植入切面：

public class MyBeanFactory {

public static CustomerDao getBean() {

// 准备目标类

final CustomerDao customerDao = new CustomerDaoImpl();

// 创建切面类实例

final MyAspect myAspect = new MyAspect();

// 使用代理类，进行增强

return (CustomerDao) Proxy.newProxyInstance(

MyBeanFactory.class.getClassLoader(),

new Class[] { CustomerDao.class }, new InvocationHandler() {

public Object invoke(Object proxy, Method method,

Object[] args) throws Throwable {

myAspect.myBefore(); // 前增强

Object obj = method.invoke(customerDao, args);

myAspect.myAfter(); // 后增强

return obj;

}

});

}

}

这个工厂生成的customerDao的bean在执行任何方法前后，都分别会执行到MyAspect的myBefore和myAfter方法。其中customerDao是目标、dao内的方法就是切入点、myBefore/myAfter是通知、这个工厂方法getBean就是切面、方法执行的过程就叫植入。

#### Spring AOP

在Spring中不需要自己写这个代理方法，只需要配置一个AOP即可，自动解析自动代理。下面是一个XML配置AOP的实例：

<beans …>

    <!--目标类 -->

    <bean id="customerDao" class="com.\*.CustomerDaoImpl" />

    <!-- 通知 advice -->

    <bean id="myAspect" class="com.\*.MyAspect" />

    <!--生成代理对象 -->

    <bean id="customerDaoProxy"

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

     <!--代理实现的接口 -->

        <property name="proxyInterfaces" value="com.\*.CustomerDao" />

        <!--代理的目标对象 -->

        <property name="target" ref="customerDao" />

        <!--用通知增强目标 -->

        <property name="interceptorNames" value="myAspect" />

        <!-- 如何生成代理，true:使用cglib; false :使用jdk动态代理 -->

        <property name="proxyTargetClass" value="true" />

    </bean>

</beans>

通过XML的一段配置，就可以自动实现相同的AOP，并且不用重复写代理的代码。

#### 使用AspectJ

AspectJ和Spring AOP类似，不同的是，AspectJ提供了更多切入点类型：前置、后置、环绕、抛出和最终通知。一段配置实例如下：

<!--目标类 -->

<bean id="customerDao" class="com.mengma.dao.CustomerDaoImpl" />

<!--切面类 -->

<bean id="myAspect" class="com.mengma.aspectj.xml.MyAspect"></bean>

<!--AOP 编程 -->

<aop:config>

<aop:aspect ref="myAspect">

<!-- 配置切入点，通知最后增强哪些方法 -->

<aop:pointcut expression="execution ( \* com.mengma.dao.\*.\* (..))"

id="myPointCut" />

<!--前置通知，关联通知 Advice和切入点PointCut -->

<aop:before method="myBefore" pointeut-ref="myPointCut" />

<!--后置通知，在方法返回之后执行，可获得returning 属性 -->

<aop:after-returning method="myAfterReturning"

pointcut-ref="myPointCut" returning="returnVal" />

<!--环绕通知 -->

<aop:around method="myAround" pointcut-ref="myPointCut" />

<!--抛出通知：用于处理程序发生异常，接收当前方法产生的异常 -->

<!-- \*注意：如果程序没有异常，则不会执行增强 -->

<!-- \* throwing属性：用于设置通知第二个参数的名称，类型Throwable -->

<aop:after-throwing method="myAfterThrowing"

pointcut-ref="myPointCut" throwing="e" />

<!--最终通知：无论程序发生任何事情，都将执行 -->

<aop:after method="myAfter" pointcut-ref="myPointCut" />

</aop:aspect>

</aop:config>

AspectJ 框架为 AOP 开发提供了另一种开发方式——基于 Annotation 的声明式。AspectJ 允许使用注解定义切面、切入点和增强处理，而 Spring 框架则可以识别并根据这些注解生成 AOP 代理。可以使用的注解如下：

|  |  |
| --- | --- |
| **名称** | **说明** |
| @Aspect | 用于定义一个切面。 |
| @Before | 用于定义前置通知，相当于 BeforeAdvice。 |
| @AfterReturning | 用于定义后置通知，相当于 AfterReturningAdvice。 |
| @Around | 用于定义环绕通知，相当于MethodInterceptor。 |
| @AfterThrowing | 用于定义抛出通知，相当于ThrowAdvice。 |
| @After | 用于定义最终final通知，不管是否异常，该通知都会执行。 |
| @DeclareParents | 用于定义引介通知，相当于IntroductionInterceptor（没懂这是啥？） |

## Spring事务

Spring提供的事务管理是声明式事务管理，通过在容器中注册驱动后，通过注解@Transactional就可以实现事务管理。这样的事务管理也是通过代理实现的。事务管理的三个核心接口：PlatformTransactionManager、TransactionDefinition 和 TransactionStatus。

前几天看到一段视屏讲Spring这个注解的坑有两个：遇到一般的异常不会释放事务（除了Error和运行时异常）；还有就是方法放回事务才会提交，对于复杂的业务逻辑会出现并发问题（高并发下需要使用编程式事务管理）。

## Spring集成的其他框架

Spring集成了Struct2、Hibernate、MyBatis等，大部分都是通过XML配置文件，注册框架的驱动Bean+指定其他框架的配置文件这样的方式集成其他框架，这部分要上手实操才行，其他框架的内容暂时不在本文档内赘述。

## Spring Boot是什么？

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化Spring应用初始搭建以及开发过程。简单理解来说，就是Spring的进阶版本，提供更多的模板，支持更多的注解，来帮助程序员快速搭建程序(推荐注解的方式配置Spring)。（特点: **约定大于配置**）

#### Spring的核心功能

1）独立运行的 Spring 项目

Spring Boot 可以以 jar 包的形式独立运行，运行一个 Spring Boot 项目只需通过 java–jar xx.jar 来运行。

2）内嵌 Servlet 容器

Spring Boot 可选择内嵌 Tomcat、Jetty 或者 Undertow，这样我们无须以 war 包形式部署项目。

3）提供 starter 简化 [Maven](http://c.biancheng.net/maven/) 配置

Spring 提供了一系列的 starter pom 来简化 Maven 的依赖加载，例如，当你使用了spring-boot-starter-web 时，会自动加入如图 1 所示的依赖包。

4）自动配置 Spring

Spring Boot 会根据在类路径中的 jar 包、类，为 jar 包里的类自动配置 Bean，这样会极大地减少我们要使用的配置。当然，Spring Boot 只是考虑了大多数的开发场景，并不是所有的场景，若在实际开发中我们需要自动配置 Bean，而 Spring Boot 没有提供支持，则可以自定义自动配置。

5）准生产的应用监控

Spring Boot 提供基于 http、ssh、telnet 对运行时的项目进行监控。

6）无代码生成和 xml 配置

Spring Boot 的神奇的不是借助于代码生成来实现的，而是通过条件注解来实现的， Spring Boot 不需要任何 xml 配置即可实现 Spring 的所有配置。

#### Spring Boot特点

1. 搭建项目快，几秒钟就可以搭建完成；
2. 让测试变的简单，内置了JUnit、Spring Boot Test等多种测试框架，方便测试；
3. Spring Boot让配置变的简单，Spring Boot的核心理念：约定大约配置，约定了某种命名规范，可以不用配置，就可以完成功能开发，比如模型和表名一致就可以不用配置，直接进行CRUD（增删改查）的操作，只有表名和模型不一致的时候，配置名称即可；
4. 内嵌容器，省去了配置Tomcat的繁琐；
5. 方便监控，使用Spring Boot Actuator组件提供了应用的系统监控，可以查看应用配置的详细信息；

自动化的最大的缺点是，难调试。

#### Spring boot常见注解

Springboot的初衷就是简化繁复的XML配置，使用注解的方式，提供更快速的搭建项目模板，所以关键在于boot支持的注解，就可以大致看出提供了什么功能。

|  |  |
| --- | --- |
| @SpringBootApplication | @SpringBootApplication：包含了@ComponentScan、@Configuration和@EnableAutoConfiguration注解。其中@ComponentScan让spring Boot扫描到Configuration类并把它加入到程序上下文。 |
| @Configuration | @Configuration 等同于spring的XML配置文件；使用Java代码可以检查类型安全。 |
| @EnableAutoConfiguration | 自动配置 |
| @ComponentScan | 组件扫描，可自动发现和装配一些Bean。 |
| @Component | 可配合CommandLineRunner使用，在程序启动后执行一些基础任务。 |
| @RestController | 注解是@Controller和@ResponseBody的合集,表示这是个控制器bean,并且是将函数的返回值直 接填入HTTP响应体中,是REST风格的控制器。 |
| @Autowired | 自动导入。 |

其中@SpringBootApplication就是SpringBoot提供的快速启动项目的集合型的注解，需要使用maven引入stater的jar包,提供了一个约定的项目启动方式，即使没有详细的配置，当时boot使用它的约定，使用默认的配置启动。

## 下面是使用Visio画的思维导图：







