# Spring发展史

### Spring1.x时代

在 Spring1.x 时代，开发项目是通过在项目中配置大量的xml文件，并且在每个xml文件中通过IOC DI的特点实现大量bean对象的创建，所以有很多bean标签，注入依赖的关系。随着项目不断的扩大，xml越来越多，bean越来越繁琐。

### Spring2.x时代

在spring的2.x时代，java5出现了，也就是jdk1.5，他的出现使得注解的代码广泛使用。Spring也利用jdk1.5中的javaConfig实现了很多xml配置转化向bean的代码申明和注入。极大的方便了开发效率。那么随之问题也出现，到底应该偏向于使用xml配置，还是使用代码注解实现配置。

开发习惯统一：

技术应用的配置使用xml，例如数据源，资源属性等等

业务层使用注解，例如Controller,Service等

### Spring3.x/4.x时代

从 Spring3.x 开始提供了 Java 配置方式，使用 Java 配置方式可以更好的理解你配置的 Bean，现在我们就处于这个时代，并且 Spring4.x 和 Spring boot 都推荐使用 java 配置的方式。可以完全的从xml配置转化向代码的配置编写---@Configuration

### Spring5.x时代

Spring5.x 是 Java 界首个支持响应式的 Web 框架，是 Spring 的一个重要版本，距离 Spring4.x 差不多四年。在此期间，大多数增强都是在 SpringBoot 项目中完成的，其最大的亮点就是提供了完整的端到端响应式编程的支持（新增 Spring WebFlux 模块）。

Spring WebFlux 同时支持使用旧的 Spring MVC 注解声明 Reactive Controller。和传统的 MVC Controller 不同，Reactive Controller 操作的是 非阻塞 的 ServerHttpRequest 和 ServerHttpResponse，而不再是 Spring MVC 里的 HttpServletRequest 和 HttpServletResponse。

至此也代表着 Java 正式迎来了响应式异步编程的时代。

# 启动类

我们开发任何一个Spring Boot项目，都会用到如下的启动类

|  |
| --- |
| @SpringBootApplication public class MainStarter {  public static void main(String[] args) {  SpringApplication.*run*(MainStarter.class,args);  } } |

从上面代码可以看出，Annotation定义（@SpringBootApplication）和类定义（SpringApplication.run）最为耀眼，所以要揭开SpringBoot的神秘面纱，我们要从这两位开始就可以了。

# ****SpringBootApplication(核心注解)****

@SpringBootApplication注解是Spring Boot的核心注解，它其实是一个组合注解：

|  |
| --- |
| @Target(ElementType.*TYPE*) @Retention(RetentionPolicy.*RUNTIME*) @Documented @Inherited @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(excludeFilters = {  @Filter(type = FilterType.*CUSTOM*, classes = TypeExcludeFilter.class),  @Filter(type = FilterType.*CUSTOM*, classes = AutoConfigurationExcludeFilter.class) }) public @interface SpringBootApplication { |

除了三个元注解，但实际上重要的只有三个Annotation：

* @Configuration（@SpringBootConfiguration点开查看发现里面还是应用了@Configuration）

|  |
| --- |
| @Target(ElementType.*TYPE*) @Retention(RetentionPolicy.*RUNTIME*) @Documented @Configuration public @interface SpringBootConfiguration {  } |

* @EnableAutoConfiguration
* @ComponentScan

### @Configuration

它能够标识一个类为配置类,只要spring容器加载这个配置类,相当于是加载类一个xml配置文件.

任何一个标注了@Configuration的Java类定义都是一个JavaConfig配置类。

这里的@Configuration就相当于一个xml配置文件，可以理解为等同于如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"  default-lazy-init="true">  <!--bean定义-->  </beans> |

#### 注册bean

任何一个标注了@Bean的方法，其返回值将作为一个bean定义注册到Spring的IoC容器，方法名将默认成该bean定义的id。

|  |
| --- |
| @Configuration class test{  @Bean  public String bean(){  return new String("创建的Bean");  } } |

等同于XML形式：

|  |
| --- |
| <bean id="string" class="...String"/> |

@Configuration的注解类标识这个类可以使用Spring IoC容器作为bean定义的来源。

@Bean注解告诉Spring，一个带有@Bean的注解方法将返回一个对象，该对象应该被注册为在Spring应用程序上下文中的bean。

### ****@ComponentScan****

相当于一个xml配置文件中的<context:component-scan>标签

@ComponentScan这个注解在Spring中很重要，它对应XML配置中的元素，@ComponentScan的功能其实就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义，最终将这些bean定义加载到IoC容器中。

     我们可以通过basePackages等属性来细粒度的定制@ComponentScan自动扫描的范围，如果不指定，则默认Spring框架实现会从声明@ComponentScan所在类的package进行扫描。

注：所以SpringBoot的启动类最好是放在root package下，因为默认不指定basePackages。

|  |
| --- |
| @Configuration @ComponentScan(basePackages = "cn.shu") class test{  @Bean  public String bean(){  return new String("创建的Bean");  } } |

### ****@EnableAutoConfiguration****

每一个版本的springboot都会扩展非常庞大的\*\*AutoConfiguration的配置类,每一个类都相当于是一个xml.@EnableAutoConfiguration的作用就是导入这些配置类,使得在加载一个springboot启动类时,不仅具备扫描自定义业务层注解的功能,还具备加载springboot自动配置逻辑

|  |
| --- |
|  |
| 计算机生成了可选文字: @SpringB00tConfiguration .Com/onentScan @EnabIeAut0Configuration Controller Service Component import springboot AAut0Config BAut0Config AutoCOnfig NAut0Config |

springboot当前版本都准备了哪些自动配置类,可以从一个spring.factories的文件中观察到.

在spring-boot-autoconfigure的jar包中,找到META-INF文件夹,从中观察打开spring.factories

|  |
| --- |
| https://images2017.cnblogs.com/blog/249993/201712/249993-20171208161556484-1145877030.jpg |

上图就是从SpringBoot的autoconfigure依赖包中的META-INF/spring.factories配置文件中摘录的一段内容，可以很好地说明问题。

所以，@EnableAutoConfiguration自动配置的大致过程：**从classpath中搜寻所有的META-INF/spring.factories配置文件，并将其中org.springframework.boot.autoconfigure.EnableutoConfiguration对应的配置项通过反射（Java Refletion）实例化为对应的标注了@Configuration的JavaConfig形式的IoC容器配置类，然后汇总为一个并加载到IoC容器。**

# XML转向代码

### 加载配置类

标识一个类为配置类,只要spring容器加载这个配置类,相当于是加载类一个xml配置文件.

|  |
| --- |
| @Configuration class test{  } |

### 配合包扫描

添加@ComponentScan注解,配合@Configuration相当于在一个xml中准备了<context:component-scan>

|  |
| --- |
| @Configuration @ComponentScan class test{  } |

### 配置类中配置bean对象

@Bean注解,可以作用在方法上,如果一个配置类中,某个方法存在这个@Bean,返回值会交给容器管理;

相当于之前一个<bean>

|  |
| --- |
| @Configuration @ComponentScan class test{  @Bean  public User bean(){  return new User();  } } |

### 条件注解Conditional

springboot基于spring上述机制,扩展了一些注解,这样的注解找不到xml对应关系了.其中一个核心注解就是条件注解@Conditional的各种衍生产品

|  |
| --- |
| @Configuration @ComponentScan class test{  @ConditionalOnBean(User.class)  @Bean  public User bean(){  return new User();  } } |

@ConditionalOnBean属于@Conditional一个衍生

当注解属性是一个类的反射对象时,这个方法到底要不要创建执行,取决于是否在容器中有这个User.class的对象,有则满足条件,没有则不满足条件

其他衍生注解

@ConditionalOnMissingBean:容器中没有指定的某些bean对象才会满足条件

@ConditionalOnClass:大量springboot自动配置类使用的条件注解,依赖资源有指定的类才满足条件

@ConditionalOnMissingClass:依赖没有该指定类才满足条件

@ConditionalOnWebApplication:当前进程是web工程满足条件

@ConditionalOnNotWebApplication:当前进程不是web工程满足条件

# 总结springboot自动配置原理:

1.spring完全支持从xml配置方式转向代码的配置方式

2.springboot基于这种机制,扩展了大量的配置类

3.springboot扩展了条件注解,细化内容,使得大量配置类中满足条件的配置逻辑才会加载