一 Spring 发展史

1 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。

2 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

3 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

4 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。

二　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。

三　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

1 @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>
```

1.A 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。

2 @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。

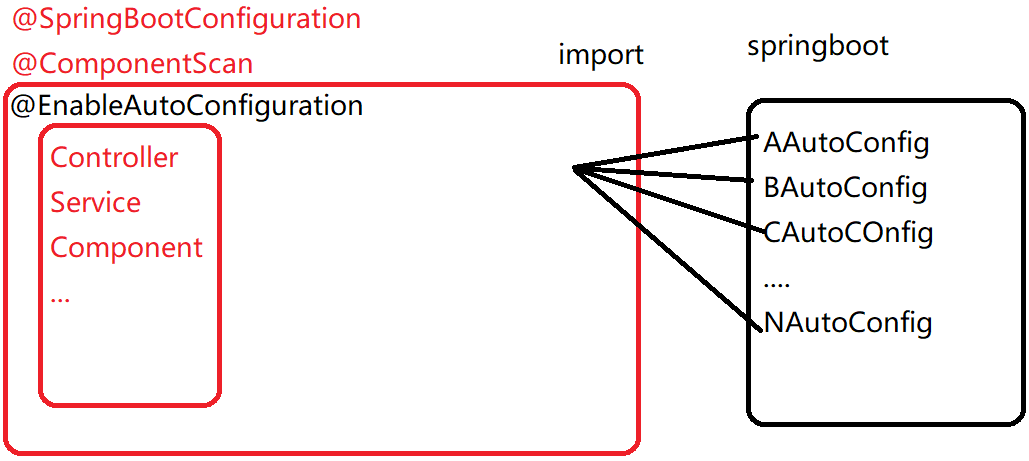
注：所以 SpringBoot 的启动类最好是放在 root package 下，因为默认不指定 basePackages。

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

3 @EnableAutoConfiguration

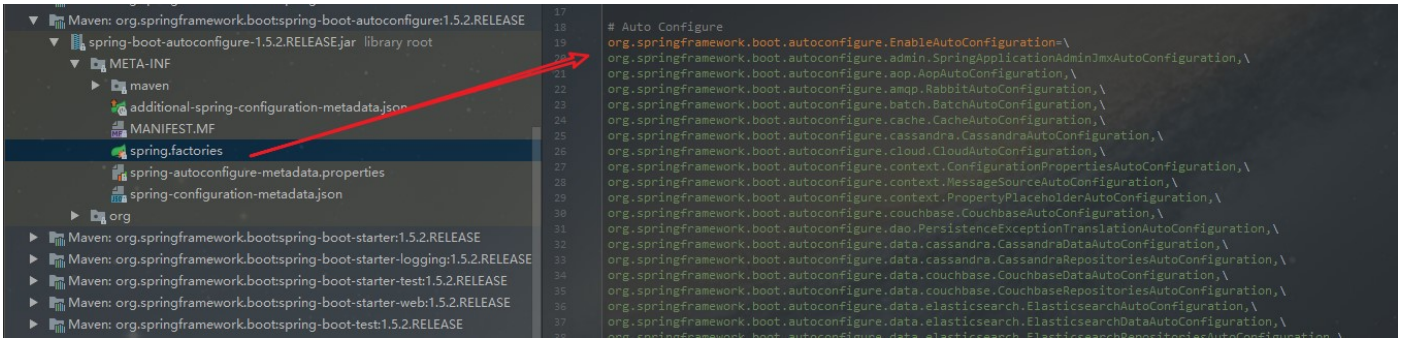
每一个版本的 springboot 都会扩展非常庞大的**AutoConfiguration 的配置类,每一个类都相当于是一个 xml.@EnableAutoConfiguration 的作用就是导入这些配置类,使得在加载一个 springboot 启动类时,不仅具备扫描自定义业务层注解的功能,还具备加载 springboot 自动配置逻辑

```
/deprecation/
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```



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



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。

四　XML 转向代码

1　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{

}
```

2　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{

}
```

3 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();
    }
}
```

4 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

五　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