## Spring Boot

http://c.biancheng.net/spring\_boot/

## 1. 微服务（Microservice）是什么？为什么会出现微服务？

SOA（面向服务的架构）的演化趋势

|  |  |
| --- | --- |
| Monolith | 微服务 |
| 早些年的服务实现和实施思路是将很多功能从开发到交付都打包成一个很大的服务单元（一般称为 Monolith）， | 而微服务实现和实施思路则更强调功能趋向单一，服务单元小型化和微型化。 |
| 在开发阶段，如果我们遵循 Monolith 的服务化理念，通常会将所有功能的实现都统一归到一个开发项目下，但随着功能的膨胀，这些功能一定会分发给不同的研发人员进行开发，造成的后果就是，大家在提交代码的时候频繁冲突并需要解决这些冲突，单一的开发项目成为了开发期间所有人的工作瓶颈。 | 为了减轻这种苦恼，我们自然会将项目按照要开发的功能拆分为不同的项目，从而负责不同功能的研发人员就可以在自己的代码项目上进行开发，从而解决了大家无法在开发阶段并行开发的苦恼。 |
| 到了软件交付阶段，如果我们遵循 Monolith 的服务化理念，那么，我们一定是将所有这些开发阶段并行开发的项目集合到一起进行交付。  这就涉及服务化早期实践中比较有名的“火车模型”，即交付的服务就像一辆火车，而这个服务相关的所有功能对应的项目成果，就是要装上火车车厢的一件件货物，交付的列车只有等到所有项目都开发测试完成后才可以装车出发，完成整个服务的交付。  很显然，只要有一个车厢没有准备好货物（即功能项目未开发测试完成），火车就不能发车，服务就不能交付，这大大降低了服务的交付效率。如果每个功能项目可以各自独立交付，那么就不需要都等同一辆火车，各自出发就可以了。 | 顺着这个思路，自然而然地，大家逐渐各自独立，每一个功能或者少数相近的功能作为单一项目开发完成后将作为一个独立的服务单元进行交付，从而在服务交付阶段，大家也能够并行不悖，各自演化而不受影响。  所以，随着服务和系统的复杂度逐渐飙升，为了能够在整个软件的交付链路上高效扩展，将独立的功能和服务单元进行拆分，从而形成一个一个的微服务是自然而然发生的事情。 |

## 2. 微服务的好处（优点）有哪些？

* 开发独立-快速迭代，多语言生态
* 交付独立-快速交付
* 部署运行独立---易扩展，隔离性好

## 3. 实现微服务会带来哪些挑战？

完备的微服务研发和交付体系基础设施建设就成为了实施微服务的终极挑战-🡪标准化

* 开发环境-收缩语言生态选择范围
* 交付链路
* 减少采购成本
* 标准协议格式治理维护微服务

## 4. Spring框架的起源

* EJB 1 的时代（EJB 简单来说就是把已经编写好的程序/类打包放在服务器上执行）
* Spring 初版《Expert One-on-One J2EE Design and Development》—统一而简洁的设计，对 Java 应用开发中常用的技术进行了合理的设计和封装。

 Spring 被称为 J2EE 的春天，是一个开源的轻量级的 Java 开发框架， 具有控制反转（IoC）和面向切面（AOP）两大核心。

## 5. Spring IoC介绍

* IoC（Inversion Of Control）:
  + DI（Dependency Injection）当前软件实体被动接受其依赖的其他组件被 IoC 容器注入
  + DL （Dependency Lookup依赖查找）当前软件实体主动去某个服务注册地查找其依赖的那些服务
* Spring IoC， Spring 框架提供的 IoC 容器实现（IoC Container）
  + 典型代码片段：

public class App {

public static void main(String[] args) {

ApplicationContext context = new FileSystemXmlApplication-Context("...");

// ...

MockService service = context.getBean(MockService.class);

service.doSomething();

}

}

* Spring IoC 容器的依赖注入
  + 收集和注册

*我们可以通过 XML 或者 Java 代码的方式定义一些 bean，然后通过手动组装或者让容器基于某些机制自动扫描的形式，将这些 bean 定义收集到 IoC 容器中。*

*假设我们以 XML 配置的形式来收集并注册单一 bean，一般形式如下：*

<bean id="mockService" class="..MockServiceImpl"> ...</bean>

*如果嫌逐个收集 bean 定义麻烦，想批量地收集并注册到 IoC 容器中，我们也可以通过 XML Schema 形式的配置进行批量扫描并采集和注册：*

<context:component-scan base-package="com.keevol">

* + 分析和组装

*分析这些已经在 IoC 容器之中的 bean，然后根据它们之间的依赖关系先后组装它们。至于分析和组装的依据，Spring 框架最早是通过 XML 配置文件的形式来描述 bean 与 bean 之间的关系的，随着 Java 业界研发技术和理念的转变，基于 Java 代码和 Annotation 元信息的描述方式也日渐兴盛（比如 @Autowired 和 @Inject）.*

## 6. Spring JavaConfig和常见Annotation

### 6.1 Spring JavaConfig

[Java](http://c.biancheng.net/java/) 5 的推出，加上当年基于纯 Java Annotation 的依赖注入框架 Guice 的出现，使得 [Spring](http://c.biancheng.net/spring/) 框架及其社区也“顺应民意”，推出并持续完善了基于 Java 代码和 Annotation 元信息的依赖关系绑定描述方式，即 JavaConfig 项目。  
  
基于 JavaConfig 方式的依赖关系绑定描述基本上映射了最早的基于 XML 的配置方式，比如：

#### 1）表达形式层面 @Configuration

基于 XML 的配置方式是这样的：

1. <?xml version="1.0" encoding="UTF-8"?>
2. **<beans** xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:context="http://www.springframework.org/schema/context"
5. xsi:schemaLocation="http://www.springframework.org/schema/beans
6. http://www.springframework.org/schema/beans/spring-beans.xsd
7. http://www.springframework.org/schema/context
8. http://www.springframework.org/schema/context/spring-context.xsd"**>**
10. <!-- bean定义 -->
11. **</beans>**

而基于 JavaConfig 的配置方式是这样的：

1. @Configuration
2. **public** **class** MockConfiguration{
3. // bean定义
4. }

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

#### 2）注册 bean 定义层面 @Bean

基于 XML 的配置形式是这样的：

<bean id="mockService" class="..MockServiceImpl"> ...</bean>

而基于 JavaConfig 的配置形式是这样的：

1. @Configuration
2. **public** **class** MockConfiguration {
3. @Bean
4. **public** MockService mockService() {
5. **return** **new** MockServiceImpl();
6. }
7. }

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，方法名将默认成为该 bean 定义的 id。

#### 3）表达依赖注入关系层面

为了表达 bean 与 bean 之间的依赖关系，在 XML 形式中一般是这样的：

1. **<bean** id="mockService" class="..MockServiceImpl"**>**
2. **<property** name="dependencyService" ref="dependencyService" **/>**
3. **</bean>**
4. **<bean** id="dependencyService" class="DependencyServiceImpl" **/>**

而在 JavaConfig 中则是这样的：

1. @Configuration
2. **public** **class** MockConfiguration {
3. @Bean
4. **public** MockService mockService() {
5. **return** **new** MockServiceImpl(dependencyService());
6. }
7. @Bean
8. **public** DependencyService dependencyService() {
9. **return** **new** DependencyServiceImpl();
10. }
11. }

如果一个 bean 的定义依赖其他 bean，则直接调用对应 JavaConfig 类中依赖 bean 的创建方法就可以了。

### 6.2 那些高曝光率的 Annotation

至于 @Configuration，我想前面已经提及过了，这里不再赘述，下面我们看几个其他比较常见的 Annotation，便于为后面更好地理解 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 框架的奥秘做准备。

#### 1. @ComponentScan

@ComponentScan 对应 XML 配置形式中的 <context：component-scan> 元素，用于配合一些元信息 Java Annotation，比如 @Component 和 @Repository 等，将标注了这些元信息 Annotation 的 bean 定义类批量采集到 Spring 的 IoC 容器中。  
  
我们可以通过 basePackages 等属性来细粒度地定制 @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。  
  
@ComponentScan 是 SpringBoot 框架魔法得以实现的一个关键组件，大家可以重点关注，我们后面还会遇到它。

#### 2. @PropertySource 与 @PropertySources

@PropertySource 用于从某些地方加载 \*.properties 文件内容，并将其中的属性加载到 IoC 容器中，便于填充一些 bean 定义属性的占位符（placeholder），当然，这需要 PropertySourcesPlaceholderConfigurer 的配合。  
  
如果我们使用 Java 8 或者更高版本开发，那么，我们可以并行声明多个 @PropertySource：

1. @Configuration
2. @PropertySource("classpath:1.properties")
3. @PropertySource("classpath:2.properties")
4. @PropertySource("...")
5. **public** **class** XConfiguration{
6. ...
7. }

如果我们使用低于 Java 8 版本的 Java 开发 Spring 应用，又想声明多个 @PropertySource，则需要借助 @PropertySources 的帮助了，代码如下所示：

1. @PropertySources({ @PropertySource("classpath:1.properties"), @PropertySource("classpath:2.properties"), ...})
2. **public** **class** XConfiguration{
3. ...
4. }

**3. @Import 与 @ImportResource**

在 XML 形式的配置中，我们通过 <import resource="XXX.xml"/> 的形式将多个分开的容器配置合到一个配置中，在 JavaConfig 形式的配置中，我们则使用 @Import 这个 Annotation 完成同样目的：

1. @Configuration
2. @Import(MockConfiguration.**class**)
3. **public** **class** XConfiguration {
4. ...
5. }

@Import 只负责引入 JavaConfig 形式定义的 IoC 容器配置，如果有一些遗留的配置或者遗留系统需要以 XML 形式来配置（比如 dubbo 框架），我们依然可以通过 @ImportResource 将它们一起合并到当前 JavaConfig 配置的容器中。

## 7. SpringBoot是什么？SpringBoot的优缺点有哪些？

着动态语言的流行（Ruby、Groovy、Scala、Node.js），[Java](http://c.biancheng.net/java/" \t "_blank) 的开发显得格外的笨重，繁多的配置、低下的开发效率、复杂的部署流程以及第三方技术集成难度大。  
  
在上述环境下，[Spring](http://c.biancheng.net/spring/) Boot 应运而生。它使用“习惯优于配置”（项目中存在大量的配置，此外还内置一个习惯性的配置，让你无须手动进行配置）的理念让你的项目快速运行起来。  
  
使用 Spring Boot 很容易创建一个独立运行（运行 jar，内嵌 [Servlet](http://c.biancheng.net/servlet/) 容器）、准生产级别的基于 Spring 框架的项目，使用 Spring Boot 你可以不用或者只需要很少的 Spring 配置。

SpringBoot 是 Spring 框架对“约定优先于配置（Convention Over Configuration）”理念的最佳实践的产物，一个典型的 SpringBoot 应用本质上其实就是一个基于 Spring 框架的应用.

### 7.1 Spring Boot 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，这是 Spring 4.x 提供的新特性。Spring 4.x 提倡使用 Java 配置和注解配置组合，而 Spring Boot 不需要任何 xml 配置即可实现 Spring 的所有配置。

### 7.2 Spring Boot的优缺点

**1）优点**

* 快速构建项目。
* 对主流开发框架的无配置集成。
* 项目可独立运行，无须外部依赖Servlet容器。
* 提供运行时的应用监控。
* 极大地提高了开发、部署效率。
* 与[云计算](http://c.biancheng.net/cloud_computing/)的天然集成。

**2）缺点**

* 版本迭代速度很快，一些模块改动很大。
* 由于不用自己做配置，报错时很难定位。
* 网上现成的解决方案比较少。

## 8. SpringBoot快速搭建

1）访问[http://start.spring.io/](http://start.spring.io/" \t "_blank)

9. SpringBoot中@SpringBootApplication注解的三体结构解析

10. SpringApplication.run执行流程详解

11. Spring Boot的自动配置

12. Spring-Boot-Starter常用依赖模块详解

13. spring-boot-starter-logging和spring-boot-starter-web

14. spring-boot-starter-web和快速Web应用开发

15. spring-boot-starter-jdbc与数据访问

16. spring-boot-starter-aop及其使用场景说明

17. spring-boot-starter-security与应用安全

18. spring-boot-starter-actuator与应用监控

19. 创建基于Dubbo框架的SpringBoot微服务

20. 使用SpringBoot快速构建Web API

21. 使用SpringBoot构建其他形式的微服务

22. SpringBoot微服务的发布与部署（3种方式）

23. SpringBoot微服务的注册与发现

24. SpringBoot微服务的监控与运维

25. SpringBoot中使用Scala开发

26. 使用Maven构建和发布基于SpringBoot的Scala应用

27. 简化基于Maven的Scala项目创建详解

28. 简化基于Scala的Web API开发

29. 使用SBT构建和发布基于SpringBoot的Scala应用