## 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)

2）创建一个最简单的依赖 Web 模块的 SpringBoot 应用，填写项目信息，如图 1 所示。

我们在此以 Maven 作为项目构建方式，Spring Boot 还支持以 Gradle 作为项目构建工具。部署形式以 jar 包形式，当然也可以用传统的 war 包形式。Spring Boot 选择2.1.6，Spring boot 还支持以 Groovy 语言开发，应用中选择 Java 作为开发语言。

3）选择完之后，下载代码

一般情况下，我们会得到一个 SpringBoot 应用的启动类，如下面代码所示：

1. import org.springframework.boot.SpringApplication;
2. import org.springframework.boot.autoconfigure.SpringBootApplication;
3. @SpringBootApplication
4. **public** **class** DemoApplication {
5. **public** **static** void main(String[] args) {
6. SpringApplication.run(DemoApplication.**class**, args);
7. }
8. }

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

@[SpringBoot](http://c.biancheng.net/spring_boot/)Application 是一个“三体”结构，实际上它是一个复合 Annotation：

@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Inherited  
@Configuration  
@EnableAutoConfiguration  
@ComponentScanpublic  
@interface  
[Spring](http://c.biancheng.net/spring/)BootApplication{...}

### @Configuration 创世纪

启动类标注了 @Configuration 之后，本身其实也是一个 IoC 容器的配置类！

将上面的 SpringBoot 启动类拆分为两个独立的 Java 类，整个形势就明朗了：

*@Configuration*

*@EnableAutoConfiguration*

*@ComponentScan*

*public class DemoConfiguration {*

*@Bean*

*public Controller controller() {*

*return new Controller();*

*}*

*}*

*public class DemoApplication {*

*public static void main(String[] args) {*

*SpringApplication.run(DemoConfiguration.class, args);*

*}*

*}*

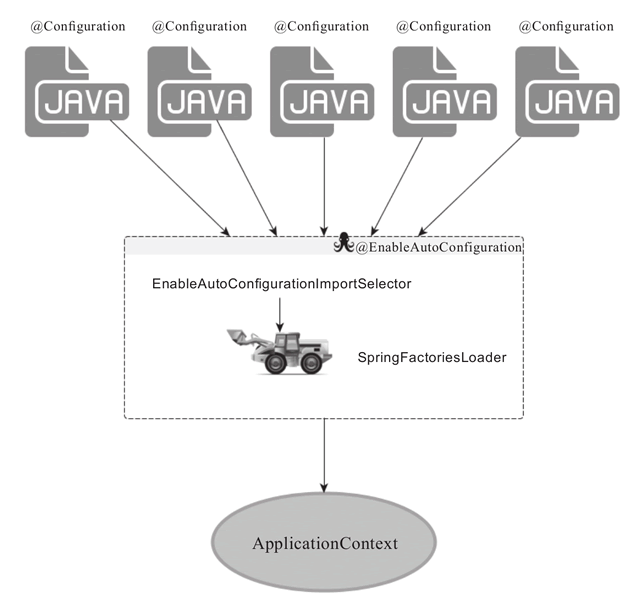
### @EnableAutoConfiguration 的功效

借助 @Import 的帮助，将所有符合自动配置条件的 bean 定义加载到 IoC 容器

@EnableAutoConfiguration 作为一个复合 Annotation，其自身定义关键信息如下：

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

借助 EnableAutoConfigurationImportSelector，@EnableAutoConfiguration 可以帮助 SpringBoot 应用将所有符合条件的 @Configuration 配置都加载到当前 SpringBoot 创建并使用的 IoC 容器，就跟一只“八爪鱼”一样（如图 1 所示）。

  
图 1  EnableAutoConfiguration得以生效的关键组件关系图

借助于 Spring 框架原有的一个工具类：SpringFactoriesLoader 的支持，@EnableAutoConfiguration 可以“智能”地自动配置功效才得以大功告成！

### 可有可无的@ComponentScan

因为原则上来说，作为 Spring 框架里的“老一辈革命家”，@ComponentScan 的功能其实就是自动扫描并加载符合条件的组件或 bean 定义，最终将这些 bean 定义加载到容器中。加载 bean 定义到 Spring 的 IoC 容器，我们可以手工单个注册，不一定非要通过批量的自动扫描完成，所以说 @ComponentScan 是可有可无的。

## SpringApplication.run执行流程详解

[Spring](http://c.biancheng.net/spring/)Application 将一个典型的 Spring 应用启动的流程“模板化”（这里是动词），在没有特殊需求的情况下，默认模板化后的执行流程就可以满足需求了但有特殊需求也没关系，SpringApplication 在合适的流程结点开放了一系列不同类型的扩展点，我们可以通过这些扩展点对 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 程序的启动和关闭过程进行扩展。  
  
最“肤浅”的扩展或者配置是 SpringApplication 通过一系列设置方法（setters）开放的定制方式，比如，我们之前的启动类的 main 方法中只有一句：

SpringApplication.run（DemoApplication.class，args）;

但如果我们想通过 SpringApplication 的一系列设置方法来扩展启动行为，则可以用如下方式进行：

1. **public** **class** DemoApplication {
2. **public** **static** void main(String[] args) {
3. // SpringApplication.run(DemoConfiguration.class, args);
4. SpringApplication bootstrap = **new** SpringApplication(Demo - Configuration.**class**);
5. bootstrap.setBanner(**new** Banner() {
6. @Override
7. **public** void printBanner(Environment environment, Class<?> aClass, PrintStream printStream) {
8. // 比如打印一个我们喜欢的ASCII Arts字符画
9. }
10. });
11. bootstrap.setBannerMode(Banner.Mode.CONSOLE);
12. // 其他定制设置...
13. bootstrap.run(args);
14. }
15. }

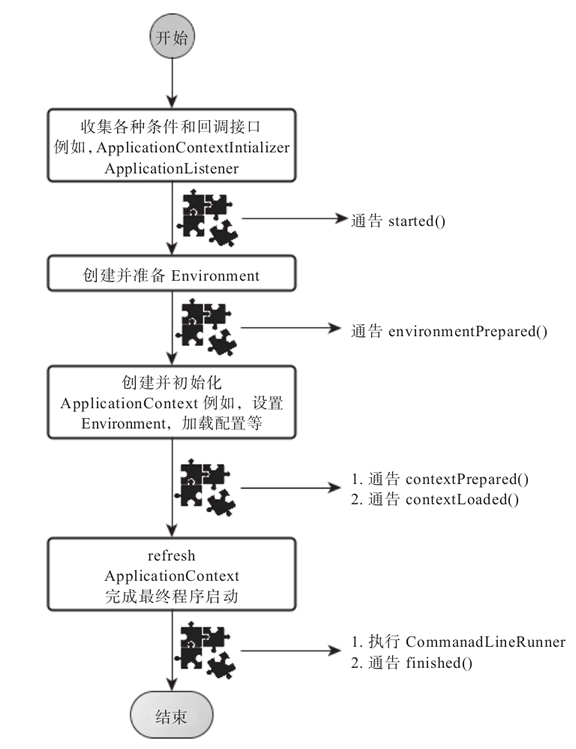
设置自定义 banner 最简单的方式其实是把 ASCII Art 字符画放到一个资源文件，然后通过 ResourceBanner 来加载：

bootstrap.setBanner(new ResourceBanner(new ClassPathResource("banner.txt")));

大部分情况下，SpringApplication 已经提供了很好的默认设置，所以，我们不再对这些表层进行探究了，因为对表层之下的东西进行探究才是我们的最终目的。

### 10.1深入探索 SpringApplication 执行流程

SpringApplication 的 run 方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下：  
  
1）如果我们使用的是 SpringApplication 的静态 run 方法，那么，这个方法里面首先需要创建一个 SpringApplication 对象实例，然后调用这个创建好的 SpringApplication 的实例 run方 法。在 SpringApplication 实例初始化的时候，它会提前做几件事情：  
  
根据 classpath 里面是否存在某个特征类（org.springframework.web.context.ConfigurableWebApplicationContext）来决定是否应该创建一个为 Web 应用使用的 ApplicationContext 类型，还是应该创建一个标准 Standalone 应用使用的 ApplicationContext 类型。  
  
使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationContextInitializer。  
  
使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationListener。  
  
推断并设置 main 方法的定义类。  
  
2）SpringApplication 实例初始化完成并且完成设置后，就开始执行 run 方法的逻辑了，方法执行伊始，首先遍历执行所有通过 SpringFactoriesLoader 可以查找到并加载的 SpringApplicationRunListener，调用它们的 started() 方法，告诉这些 SpringApplicationRunListener，“嘿，SpringBoot 应用要开始执行咯！”。  
  
3）创建并配置当前 SpringBoot 应用将要使用的 Environment（包括配置要使用的 PropertySource 以及 Profile）。  
  
4）遍历调用所有 SpringApplicationRunListener 的 environmentPrepared（）的方法，告诉它们：“当前 SpringBoot 应用使用的 Environment 准备好咯！”。  
  
5）如果 SpringApplication的showBanner 属性被设置为 true，则打印 banner（SpringBoot 1.3.x版本，这里应该是基于 Banner.Mode 决定 banner 的打印行为）。这一步的逻辑其实可以不关心，我认为唯一的用途就是“好玩”（Just For Fun）。  
  
6）根据用户是否明确设置了applicationContextClass 类型以及初始化阶段的推断结果，决定该为当前 SpringBoot 应用创建什么类型的 ApplicationContext 并创建完成，然后根据条件决定是否添加 ShutdownHook，决定是否使用自定义的 BeanNameGenerator，决定是否使用自定义的 ResourceLoader，当然，最重要的，将之前准备好的 Environment 设置给创建好的 ApplicationContext 使用。  
  
7）ApplicationContext 创建好之后，SpringApplication 会再次借助 Spring-FactoriesLoader，查找并加载 classpath 中所有可用的 ApplicationContext-Initializer，然后遍历调用这些 ApplicationContextInitializer 的 initialize（applicationContext）方法来对已经创建好的 ApplicationContext 进行进一步的处理。  
  
8）遍历调用所有 SpringApplicationRunListener 的 contextPrepared（）方法，通知它们：“SpringBoot 应用使用的 ApplicationContext 准备好啦！”  
  
9）最核心的一步，将之前通过 @EnableAutoConfiguration 获取的所有配置以及其他形式的 IoC 容器配置加载到已经准备完毕的 ApplicationContext。  
  
10）遍历调用所有 SpringApplicationRunListener 的 contextLoaded() 方法，告知所有 SpringApplicationRunListener，ApplicationContext "装填完毕"！  
  
11）调用 ApplicationContext 的 refresh() 方法，完成 IoC 容器可用的最后一道工序。  
  
12）查找当前 ApplicationContext 中是否注册有 CommandLineRunner，如果有，则遍历执行它们。  
  
13）正常情况下，遍历执行 SpringApplicationRunListener 的 finished() 方法，告知它们：“搞定！”。（如果整个过程出现异常，则依然调用所有 SpringApplicationRunListener 的 finished() 方法，只不过这种情况下会将异常信息一并传入处理）。  
  
至此，一个完整的 SpringBoot 应用启动完毕！  
  
整个过程看起来冗长无比，但其实很多都是一些事件通知的扩展点，如果我们将这些逻辑暂时忽略，那么，其实整个 SpringBoot 应用启动的逻辑就可以压缩到极其精简的几步，如图 1 所示。

  
图 1  SpringBoot应用启动步骤简要示意图

前后对比我们就可以发现，其实 SpringApplication 提供的这些各类扩展点近乎“喧宾夺主”，占据了一个 Spring 应用启动逻辑的大部分“江山”，除了初始化并准备好 ApplicationContext，剩下的大部分工作都是通过这些扩展点完成的，所以，我们接下来对各类扩展点进行逐一剖析。

### 10.2 SpringApplicationRunListener

SpringApplicationRunListener 是一个只有 SpringBoot 应用的 main 方法执行过程中接收不同执行时点事件通知的监听者：

1. **public** **interface** SpringApplicationRunListener {
2. void started();
3. void environmentPrepared(ConfigurableEnvironment environment);
4. void contextPrepared(ConfigurableApplicationContext context);
5. void contextLoaded(ConfigurableApplicationContext context);
6. void finished(ConfigurableApplicationContext context, Throwable exception);
7. }

对于我们来说，基本没什么常见的场景需要自己实现一个 Spring-ApplicationRunListener，即使 SpringBoot 默认也只是实现了一个 org.spring-framework.boot.context.event.EventPublishingRunListener，用于在 SpringBoot 启动的不同时点发布不同的应用事件类型（ApplicationEvent），如果有哪些 ApplicationListener 对这些应用事件感兴趣，则可以接收并处理。  
  
假设我们真的有场景需要自定义一个 SpringApplicationRunListener 实现，那么有一点需要注意，即任何一个 SpringApplicationRunListener 实现类的构造方法（Constructor）需要有两个构造参数，一个构造参数的类型就是我们的 org.springframework.boot.SpringApplication，另外一个就是 args 参数列表的 String[]：

1. **public** **class** DemoSpringApplicationRunListener **implements** SpringApplicationRunListener {
2. @Override
3. **public** void started() {
4. // do whatever you want to do
5. }
6. @Override
7. **public** void environmentPrepared(ConfigurableEnvironment environment) {
8. // do whatever you want to do
9. }
10. @Override
11. **public** void contextPrepared(ConfigurableApplicationContext context) {
12. // do whatever you want to do
13. }
14. @Override
15. **public** void contextLoaded(ConfigurableApplicationContext context) {
16. // do whatever you want to do
17. }
18. @Override
19. **public** void finished(ConfigurableApplicationContext context, Throwable exception) {
20. // do whatever you want to do
21. }
22. }

之后，我们可以通过 SpringFactoriesLoader 立下的规矩，在当前 SpringBoot 应用的 classpath 下的 META-INF/spring.factories 文件中进行类似如下的配置：

org.springframework.boot.SpringApplicationRunListener=\com.keevol.springboot.demo.DemoSpringApplicationRunListener

然后 SpringApplication 就会在运行的时候调用它啦！

### 10.3 ApplicationListener

ApplicationListener 其实是老面孔，属于 Spring 框架对 [Java](http://c.biancheng.net/java/) 中实现的监听者模式的一种框架实现，这里唯一值得着重强调的是，对于初次接触 SpringBoot，但对 Spring 框架本身又没有过多接触的开发者来说，可能会将这个名字与 SpringApplicationRunListener 混淆。  
  
关于 ApplicationListener 我们就不做过多介绍了，如果感兴趣，请参考 Spring 框架相关的资料和书籍。  
  
如果我们要为 SpringBoot 应用添加自定义的 ApplicationListener，有两种方式：

* 通过 SpringApplication.addListeners（..）或者 SpringApplication.setListeners（..）方法添加一个或者多个自定义的 ApplicationListener。
* 借助 SpringFactoriesLoader 机制，在 META-INF/spring.factories 文件中添加配置（以下代码是为 SpringBoot 默认注册的 ApplicationListener 配置）。

org.springframework.context.ApplicationListener=  
\org.springframework.boot.builder.ParentContextCloserApplicationListener,  
\org.springframework.boot.cloudfoundry.VcapApplicationListener,  
\org.springframework.boot.context.FileEncodingApplicationListener,  
\org.springframework.boot.context.config.AnsiOutputApplicationListener,  
\org.springframework.boot.context.config.ConfigFileApplicationListener,  
\org.springframework.boot.context.config.DelegatingApplicationListener,  
\org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicat-ionListener,  
\org.springframework.boot.logging.ClasspathLoggingApplicationListener,  
\org.springframework.boot.logging.LoggingApplicationListener

关于 ApplicationListener，我们就说这些。

### 10.4 ApplicationContextInitializer

ApplicationContextInitializer 也是 Spring 框架原有的概念，这个类的主要目的就是在 ConfigurableApplicationContext 类型（或者子类型）的 ApplicationContext 做 refresh 之前，允许我们对 ConfigurableApplicationContext 的实例做进一步的设置或者处理。  
  
实现一个 ApplicationContextInitializer 很简单，因为它只有一个方法需要实现：

1. **public** **class** DemoApplicationContextInitializer **implements** ApplicationContextInitializer {
2. @Override
3. **public** void initialize(ConfigurableApplicationContext applicationContext) {
4. // do whatever you want with applicationContext,
5. // e.g.
6. applicationContext.registerShutdownHook();
7. }
8. }

不过，一般情况下我们基本不会需要自定义一个 ApplicationContext-Initializer，即使 SpringBoot 框架默认也只是注册了三个实现：

org.springframework.context.ApplicationContextInitializer=  
\org.springframework.boot.context.ConfigurationWarningsApplication-ContextInitializer,  
\org.springframework.boot.context.ContextIdApplicationContextInitia-lizer,  
\org.springframework.boot.context.config.DelegatingApplicationContex-tInitializer

如果我们真的需要自定义一个 ApplicationContextInitializer，那么只要像上面这样，通过 SpringFactoriesLoader 机制进行配置，或者通过 SpringApplication.addInitializers（..）设置即可。

### 10.5 CommandLineRunner

CommandLineRunner 是很好的扩展接口，不是 Spring 框架原有的“宝贝”，它属于 SpringBoot 应用特定的回调扩展接口。源码如下所示：

1. **public** **interface** CommandLineRunner {
2. void run(String... args) **throws** Exception;
3. }

CommandLineRunner 需要大家关注的其实就两点：  
  
1）所有 CommandLineRunner 的执行时点在 SpringBoot 应用的 Application-Context 完全初始化开始工作之后（可以认为是 main 方法执行完成之前最后一步）。  
  
2）只要存在于当前 SpringBoot 应用的 ApplicationContext 中的任何 Command-LineRunner，都会被加载执行（不管你是手动注册这个 CommandLineRunner 到 IoC 容器，还是自动扫描进去的）。  
  
与其他几个扩展点接口类型相似，建议 CommandLineRunner 的实现类使用 @org.springframework.core.annotation.Order 进行标注或者实现 org.springframework.core.Ordered 接口，便于对它们的执行顺序进行调整，这其实十分重要，我们不希望顺序不当的 CommandLineRunner 实现类阻塞了后面其他 CommandLineRunner 的执行。

## 11. Spring Boot的自动配置

### 基于条件的自动配置

基于条件的自动配置来源于 Spring 框架中“基于条件的配置”这一特性。在 Spring 框架中，我们可以使用 @Conditional 这个 Annotation 配合 @Configuration 或者 @Bean 等 Annotation 来干预一个配置或者 bean 定义是否能够生效，其最终实现的效果或者语义类似于如下伪代码：

if(符合 @Conditional 规定的条件){  
    加载当前配置(enable current Configuration)或者注册当前bean定义；  
}

要实现基于条件的配置，我们只要通过 @Conditional 指定自己的 Condition 实现类就可以了（可以应用于类型 Type 的标注或者方法 Method 的标注）：

@Conditional({MyCondition1.class, MyCondition2.class, ...})

最主要的是，@Conditional 可以作为一个 Meta Annotation 用来标注其他 Annotation 实现类，从而构建各色的复合 Annotation，比如 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 的 autoconfigure 模块就基于这一优良的革命传统，实现了一批 Annotation（位于 org.springframework.boot.autoconfigure.condition 包下），条件注解如下：

* @ConditionalOnBean：当容器里有指定的 Bean 的条件下。
* @ConditionalOnClass：当类路径下有指定的类的条件下。
* @ConditionalOnExpression：基于 SpEL 表达式作为判断条件。
* @ConditionalOnJava：基于 JVM 版本作为判断条件。
* @ConditionalOnJndi：在 JNDI 存在的条件下查找指定的位置。
* @ConditionalOnMissingBean：当容器里没有指定 Bean 的情况下。
* @ConditionalOnMissingClass：当类路径下没有指定的类的条件下。
* @ConditionalOnNotWebApplication：当前项目不是 Web 项目的条件下。
* @ConditionalOnProperty：指定的属性是否有指定的值。
* @ConditionalOnResource：类路径是否有指定的值。
* @ConditionalOnSingleCandidate：当指定 Bean 在容器中只有一个，或者虽然有多个但是指定首选的 Bean。
* @ConditionalOnWebApplication：当前项目是 Web 项目的条件下。

有了这些复合 Annotation 的配合，我们就可以结合 @EnableAuto-Configurationn 实现基于条件的自动配置了。  
  
SpringBoot 能够风靡，很大一部分功劳需要归功于它预先提供的一系列自动配置的依赖模块，而这些依赖模块都是基于以上 @Conditional 复合 Annotation 实现的，这也意味着所有的这些依赖模块都是按需加载的，只有符合某些特定条件，这些依赖模块才会生效，这也就是我们所谓的“智能”自动配置。

### 调整自动配置的顺序

在实现自动配置的过程中，除了可以提供基于条件的配置，我们还可以对当前要提供的配置或者组件的加载顺序进行相应调整，从而让这些配置或者组件之间的依赖分析和组装可以顺利完成。  
  
我们可以使用 @org.springframework.boot.autoconfigure.AutoConfigureBefore 或者 @org.springframework.boot.autoconfigure.AutoConfigureAfter 让当前配置或者组件在某个其他组件之前或者之后进行，比如，假设我们希望某些 JMX 操作相关的 bean 定义在 MBeanServer 配置完成之后进行，那么我们就可以提供一个类似如下的配置：

[纯文本复制](http://c.biancheng.net/view/4637.html)

1. @Configuration
2. @AutoConfigureAfter(JmxAutoConfiguration.**class**)
3. **public** **class** AfterMBeanServerReadyConfiguration {
4. @AutoWired
5. MBeanServer mBeanServer;
6. //通过 @Bean 添加必要的 bean 定义
7. }

至此，我们对 SpringBoot 的核心组件完成了基本的剖析，综合来看，大部分的东西都是 Spring 框架背后原有的一些概念和实践方式，SpringBoot 只是在这些概念和实践方式上对特定的场景实现进行了固化和升华，而也恰恰是这些固化让我们开发基于 Spring 框架的应用更加方便高效

## 12. Spring-Boot-Starter常用依赖模块详解

一般认为，[SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 微框架从两个主要层面影响 [Spring](http://c.biancheng.net/spring/) 社区的开发者们：

* 基于 Spring 框架的“约定优先于配置（COC）”理念以及最佳实践之路。
* 提供了针对日常企业应用研发各种场景的 spring-boot-starter 自动配置依赖模块，如此多“开箱即用”的依赖模块，使得开发各种场景的 Spring 应用更加快速和高效。

## 13. spring-boot-starter-logging

[Java](http://c.biancheng.net/java/) 的日志系统多种多样，从 java.util 默认提供的日志支持，到 log4j，log4j2，commons logging 等，复杂繁多，所以，应用日志系统的配置就会比较特殊，从而 spring-boot-starter-logging 也比较特殊一些，下面将其作为我们第一个了解的自动配置依赖模块。  
  
假如 maven 依赖中添加了 spring-boot-starter-logging，如以下代码所示：

1. **<dependency>**
2. **<groupId>** org.springframework.boot **</groupId>**
3. **<artifactId>** spring-boot-starter-logging **</artifactId>**
4. **</dependency>**

那么，我们的 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 应用将自动使用 logback 作为应用日志框架，[Spring](http://c.biancheng.net/spring/" \t "_blank)Boot 启动的时候，由 org.springframework.boot.logging.Logging-Application-Listener 根据情况初始化并使用。  
  
SpringBoot 为我们提供了很多默认的日志配置，所以，只要将 spring-boot-starter-logging 作为依赖加入到当前应用的 classpath，则“开箱即用”，不需要做任何多余的配置，但假设我们要对默认 SpringBoot 提供的应用日志设定做调整，则可以通过几种方式进行配置调整：

* 遵循 logback 的约定，在 classpath 中使用自己定制的 logback.xml 配置文件。
* 在文件系统中任何一个位置提供自己的 logback.xml 配置文件，然后通过 logging.config 配置项指向这个配置文件来启用它，比如在 application.properties 中指定如下的配置。

logging.config=/{some.path.you.defined}/any-logfile-name-I-like.log

SpringBoot 默认允许我们通过在配置文件或者命令行等方式使用 logging.file 和 logging.path 来自定义日志文件的名称和存放路径，不过，这只是允许我们在 SpringBoot 框架预先定义的默认日志系统设定的基础上做有限的设置，如果我们希望更灵活的配置，最好通过框架特定的配置方式提供相应的配置文件，然后通过 logging.config 来启用。  
  
如果大家更习惯使用 log4j 或者 log4j2，那么也可以采用类似的方式将它们对应的 spring-boot-starter 依赖模块加到 [Maven](http://c.biancheng.net/maven/) 依赖中即可：

1. **<dependency>**
2. **<groupId>** org.springframework.boot **</groupId>**
3. **<artifactId>** spring-boot-starter-log4j **</artifactId>**
4. **</dependency>**

或者

1. **<dependency>**
2. **<groupId>** org.springframework.boot **</groupId>**
3. **<artifactId>** spring-boot-starter-log4j2 **</artifactId>**
4. **</dependency>**

但一定不要将这些完成同一目的的 spring-boot-starter 都加到依赖中。

## 14. spring-boot-starter-web和快速Web应用开发

在这个互联网时代，使用 Spring 框架除了开发少数的独立应用，大部分情况下实际上在使用 SpringMVC 开发 web 应用，为了帮我们简化快速搭建并开发一个 Web 项目，SpringBoot 为我们提供了 spring-boot-starter-web 自动配置模块。  
  
只要将 spring-boot-starter-web 加入项目的 maven 依赖：

1. **<dependency>**
2. **<groupId>**org.springframework.boot**</groupId>**
3. **<artifactId>**spring-boot-starter-web**</artifactId>**
4. **</dependency>**

我们就得到了一个直接可执行的 Web 应用，当前项目下运行 mvn spring-boot：run 就可以直接启动一个使用了嵌入式 tomcat 服务请求的 Web 应用，只不过，我们还没有提供任何服务 Web 请求的 Controller，所以，访问任何路径都会返回一个 SpringBoot 默认提供的错误页面（一般称其为 whitelabel error page），我们可以在当前项目下新建一个服务根路径 Web 请求的 Controller 实现：

1. @RestController
2. **public** **class** IndexController {
3. @RequestMapping("/")
4. **public** String index() {
5. **return** "hello, there";
6. }
7. }

重新运行 mvn spring-boot：run 并访问 http://localhost：8080，错误页面将被我们的 Controller 返回的消息所替代，一个简单的 Web 应用就这样完成了。  
  
但是，简单的背后，其实却有很多“潜规则”（约定），我们只有充分了解了这些“潜规则”，才能更好地应用 spring-boot-starter-web。

**项目结构层面的约定**

项目结构层面与传统打包为 war 的 Java Web 应用的差异在于，静态文件和页面模板的存放位置变了，原来是放在 src/main/webapp 目录下的一系列资源，现在都统一放在 src/main/resources 相应子目录下，比如：

* src/main/resources/static 用于存放各类静态资源，比如 css，js 等。
* src/main/resources/templates 用于存放模板文件，比如 \*.vm。

当然，如果还是希望以 war 包的形式，而不是 SpringBoot 推荐使用的独立 jar 包形式发布 Web 应用，也可以继续原来 Java Web 应用的项目结构约定。

**SpringMVC 框架层面的约定和定制**

spring-boot-starter-web 默认将为我们自动配置如下一些 SpringMVC 必要组件：

* 必要的 ViewResolver，比如 ContentNegotiatingViewResolver 和 Bean-NameViewResolver。
* 将必要的 Converter、GenericConverter 和 Formatter 等 bean 注册到 IoC 容器。
* 添加一系列的 HttpMessageConverter 以便支持对 Web 请求和相应的类型转换。
* 自动配置和注册 MessageCodesResolver。
* 其他。

任何时候，如果我们对默认提供的 SpringMVC 组件设定不满意，都可以在 IoC 容器中注册新的同类型的 bean 定义来替换，或者直接提供一个基于 WebMvcConfigurerAdapter 类型的 bean 定义来定制，甚至直接提供一个标注了 @EnableWebMvc 的 @Configuration 配置类完全接管所有 SpringMVC 的相关配置，自己完全重新配置。

## 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

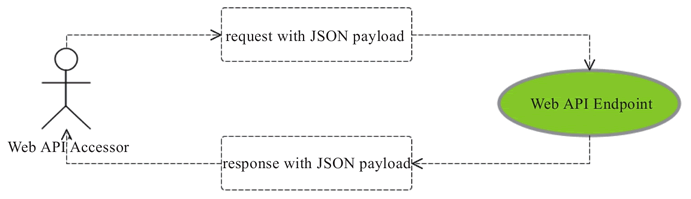
Dubbo 框架虽然有很多优点，也确实面向高强度的互联网应用场景，且在多家知名的互联网企业的生产环境得到验证，但也并非没有缺点：

* 只限于 Java 应用之间的服务调用。
* 服务访问方需要依赖 API 以及关联依赖，在很多场景下导致依赖管理混乱的问题。
* 核心项目人员转岗或者离职之后，项目不再有人专职投入维护和升级，虽然功能够用，但任何一个开源项目无推动者，无活跃社区的情况下，其生命走向只有一个，而这几乎是国内开源项目的共同宿命。

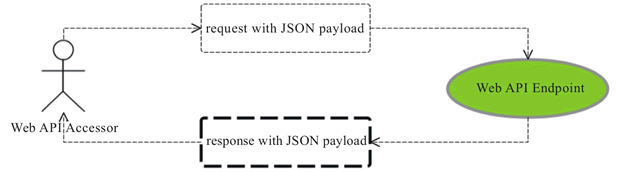
作为一名理性的研发人员，在项目技术选型的时候，需要综合考虑多种方案的优缺点，并根据现状进行权衡，实际上，对于大部分项目来说，性能可能并非技术选型的核心因素，开放和互通或许才是。  
  
是要以互通性作为核心因素进行技术选型并构建一套开放繁荣的生态体系，还是以性能为核心因素进行技术选型构建一套封闭高效的生态体系，需要大家灵活把握，而本节我们将更多以 Web API 的形式，向大家展示如何基于 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 构建一套开放、互通、稳定的 Web API 微服务体系。  
  
使用 [Spring](http://c.biancheng.net/spring/" \t "_blank)Boot 构建 Web API 有几种选择，要么使用 spring-boot-starter-jersey 构建 RESTful 风格的 Web API，要么选择 spring-boot-starter-hateoas 构建更加有关联性和相对“智能”的 Web API，但笔者认为这些都有点儿“阳春白雪”。  
  
对于大部分开发人员来说，HTTP 协议的 GET 和 POST 是直觉上最自然的选择，所以，我们选择使用最“下里巴人”的方式来构建 Web API。  
  
Web API 强调统一和互通，所以，首先我们需要定义一套内外认知一致的 Web API 开发和访问规范，在 JSON 盛行、社群庞大的背景下，我们的 Web API 方案采用 JSON 作为数据交互格式并定义统一的协议格式，然后通过 HTTP 以及周边支持完成微服务的对外服务和开放访问。

**1. 定义 Web API 规范**

首先从服务访问的交互上来说，我们可以选择较为纯粹的 JSON RPC Over HTTP 的方式，如图 1 所示。

  
图 1  JSON RPC Over HTTP 示意图

也可以选择约束相对松一些的 RPC Over HTTP 方式，如图 2 所示。

  
图 2  一般意义上基于HTTP的RPC交互示意图

相对于纯粹的 JSON RPC Over HTTP 方案，后者对请求格式不做任何限制（所以也同样支持纯粹 JSON 形式的请求格式），只对响应（Response）做 JSON 格式上的统一规定。  
  
好处是，客户端各种工具都能够很好的支持，服务器端 SpringMVC 也可以少做 HttpMessage 转换，给服务的开发者和访问者都提供了比较灵活的操作余地，至于请求的类型差异，我们可以通过配套生成的 API 文档进行补足。  
  
不管怎么样，我们选择基于后一种方案进行说明，现在剩下的主要工作就是定义服务响应格式，只有规范和统一了服务的响应格式，才能让内部和外部的服务访问者形成统一的认知。  
  
以上面同样的方式“复制”对我们提供的任何 Web API 的访问行为，减少用户的接入成本，所以，姑且我们简单规定一个服务的响应格式如下：

{ "code" : 1, "error" : "XXXXX", "data" : { ... }}

其中，code 表示调用结果的状态，0 表示成功，非 0 表示失败，并且失败情况下 error 字段将提供对应的错误信息描述，data 字段用于规范定义特定于 Web API 的响应内容。  
  
有了这样的规范定义，不同的开发者就可以根据情况选择打造对应的工具或者 SDK 了。而 Web API 的服务提供者也同样可以根据该规范考虑如何简化 Web API 的开发，或者通过约束减少规范认知不足可能导致的问题。  
  
既然是使用 SpringBoot 构建 Web API，那么显然我们现在更加关注后者。

**2. 根据规范构建 Web API**

针对同样的汇率查询服务，这回我们采用 Web API 的形式对外提供服务。  
  
使用 [http://start.spring.io](http://start.spring.io/) 构建新的 SpringBoot 项目，使其依赖 spring-boot-starter-web 模块：

1. <?xml version="1.0" encoding="UTF-8"?>
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
5. **<modelVersion>**4.0.0**</modelVersion>**
6. **<groupId>**com.keevol.springboot.chapter4**</groupId>**
7. **<artifactId>**currency-webapi**</artifactId>**
8. **<version>**0.0.1-SNAPSHOT**</version>**
9. **<packaging>**jar**</packaging>**
10. **<name>**currency-webapi**</name>**
11. **<description>**Demo project for Spring Boot**</description>**
12. **<parent>**
13. **<groupId>**org.springframework.boot**</groupId>**
14. **<artifactId>**spring-boot-starter-parent**</artifactId>**
15. **<version>**1.3.1.RELEASE**</version>**
16. **<relativePath** **/>** <!-- lookup parent from repository -->
17. **</parent>**
18. **<properties>**
19. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
20. **<java.version>**1.8**</java.version>**
21. **</properties>**
22. **<dependencies>**
23. **<dependency>**
24. **<groupId>**org.springframework.boot**</groupId>**
25. **<artifactId>**spring-boot-starter-web**</artifactId>**
26. **</dependency>**
27. **<dependency>**
28. **<groupId>**com.keevol.springboot**</groupId>**
29. **<artifactId>**currency-rates-service**</artifactId>**
30. **<version>**1.0-SNAPSHOT**</version>**
31. **</dependency>**
32. **</dependencies>**
33. **<build>**
34. **<plugins>**
35. **<plugin>**
36. **<groupId>**org.springframework.boot**</groupId>**
37. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
38. **</plugin>**
39. **</plugins>**
40. **</build>**
41. **</project>**

因为我们已经实现了 CurrencyRateService，所以，可以直接将其作为项目依赖的一部分（当然，这样也让我们的 Web API 看起来更像一个适配网关了）。  
  
我们直接使用 SpringMVC 构建对应的 Controller 对外提供 Web API 的访问如下：

1. @Controller
2. **public** **class** CurrencyRateQueryController {
3. @Autowired
4. **private** CurrencyRateService currencyRateService;
5. @RequestMapping(value = "/", method = RequestMethod.GET)
6. @ResponseBody
7. **public** ExchangeRate quote(String symbol) **throws** IOException {
8. **return** currencyRateService.quote(CurrencyPair.from(symbol));
9. }
10. }

spring-boot-starter-web 默认会提供一系列的 HttpMessageConverter 用于对请求参数和响应结果做类型转换。所以，ExchangeRate 类型将通过默认 HttpMessageConverter 序列中的 MappingJackson2HttpMessageConverter 转换成对应的 JSON 响应结果，类似于：

{ currencyPair: { symbol: “USD/CNY” }, bidPrice: 6.67, askPrice: 6.56}

整个 Web API 的功能流程算是跑通了，但跟我们之前定义的 Web API 规范却没有关系，所以，下一步我们要做的事情就是在此基础上规范 HTTP 响应格式，使其遵循我们之前定义的 Web API 规范，从而任何访问我们提供的 Web API 访问者都可以相同的认知使用这些 Web API，进而也可以打造和沉淀相应的工具或者类库。  
  
我们定义的 Web API 规范并非最优，也并非必要，如果团队成员的认知差不多，那么直接使用 HTTP Status Code 结合直接的值类型响应就可以了，我们给出的 Web API 规范考虑了更多因素后做出的一个折中方案，但任何方案设计是否完美并非最主要的，执行才是。  
  
要开发符合我们的 Web API 规范的 Web API，最少有两种方案可以选择：

* 显式的强类型封装方式（explicit type wrapper）
* 隐式的自动转换方式（implicit conversion）

显式的强类型封装方式的出发点是说，既然 spring-boot-starter-web 已经提供了 MappingJackson2HttpMessageConverter 用于对象类型到 JSON 的类型转换，那么，我们只要提供针对 Web API 规范的 Java 对象类型作为所有 Web API 处理方法的返回值就可以了，比如：

1. **public** **class** WebApiResponse<T> {
2. **public** **static** **final** int SUCCESS\_CODE = 0;
3. **public** **static** **final** int ERROR\_CODE = 1;
4. **private** int code;
5. **private** String error;
6. **private** T data;
7. // getters, setters, toString(), etc.
8. }

然后，所有的 Web API 的处理方法统一定义为返回 WebApiResponse 作为结果类型：

1. @RequestMapping(value = "/", method = RequestMethod.GET)
2. @ResponseBody
3. **public** WebApiResponse<ExchangeRate> quote(String symbol) **throws** IOException {
4. WebApiResponse<ExchangeRate> response = **new** WebApiResponse<>();
5. response.setCode(WebApiResponse.SUCCESS\_CODE);
6. response.setData(currencyRateService.quote(CurrencyPair.from(symbol)));
7. **return** response;
8. }

不过，这种模式过于强调规范的管控，对开发者来说不是太友好，即使我们通过 Builder 模式来简化 WebApiResponse 的构造过程，比如：

1. **public** **class** WebApiResponse<T> {
2. **public** **static** **final** int SUCCESS\_CODE = 0;
3. **public** **static** **final** int ERROR\_CODE = 1;
4. **private** int code;
5. **private** String error;
6. **private** T data;
7. **public** **static** <T> WebApiResponse<T> success(T data) {
8. WebApiResponse<T> response = **new** WebApiResponse<>();
9. response.setCode(SUCCESS\_CODE);
10. response.setData(data);
11. **return** response;
12. }
13. **public** **static** <T> WebApiResponse<T> error(String errorMessage) {
14. **return** WebApiResponse.<T>error(errorMessage, ERROR\_CODE);
15. }
16. // ...
17. @RequestMapping(value = "/", method = RequestMethod.GET)
18. @ResponseBody
19. **public** WebApiResponse<ExchangeRate> quote(String symbol) **throws** IOException {
20. **return** WebApiResponse.success(currencyRateService.quote(CurrencyPair.from(symbol)));
21. }
22. }

但从 API 的使用者角度来看，这种设计并非最优，最好的方式其实应该是隐式的自动转换方式。在隐式的自动转换方式下，用户的 Web API 处理方法定义保持不变，直接返回最原始的值类型（比如 ExchangeRate）：

1. @RequestMapping(value = "/", method = RequestMethod.GET)
2. @ResponseBody
3. **public** ExchangeRate quote(String symbol) **throws** IOException {
4. **return** currencyRateService.quote(CurrencyPair.from(symbol));
5. }

通过在框架层面对原始的值类型进行符合规范行为的封装，最终返回给用户的响应结果“自动”的或者说以“不打扰 API 开发者”的形式变成了符合我们 Web API 规范的响应结果形式。  
  
要达到隐式的自动转换方式的效果，最简单粗暴的做法就是完全覆盖 Web 应用的配置，只配置一个自定义处理 JSON 转换的 HttpMessageConverter 实现，比如：

1. **public** **class** JsonHttpMessageConverter **extends** AbstractHttpMessage-Converter<Object> {
2. @Override
3. **protected** boolean supports(Class<?> clazz) {
4. **return** !clazz.isPrimitive();
5. }
6. @Override
7. **protected** Object readInternal(Class<?> aClass, HttpInputMessage httpInputMessage) **throws** IOException, HttpMessageNotReadableException {
8. **return** **null**;
9. }
10. @Override
11. **protected** void writeInternal(Object o, HttpOutputMessage httpOutputMessage) **throws** IOException, HttpMessageNotWritableException {
12. httpOutputMessage.getHeaders().add("Content-Type", "application/json");
13. // 其他header设置
14. // toJson()方法中可以使用jackson或者fastjson等类库完成对象到json的转换
15. httpOutputMessage.getBody().write(toJson(o));
16. httpOutputMessage.getBody().flush();
17. }
18. }

但是，这会导致一些问题或者不便：  
1）打破了 SpringBoot 对 SpringMVC 的完备支持，对于大部分已经很熟悉 SpringMVC 框架中各种功能和类库使用的读者来说，这些可能不再有效；  
  
2）SpringBoot 提供的 spring-boot-starter-web 模块的默认配置项都不再有效，比如 SpringBoot 参考文档中的 spring.jackson.serialization.indent\_output=true 之类的配置项，这显然是在舍弃已有的良好文档和功能支持；  
  
3）因为现在只有一个 HttpMessageConverter 处理单一类型的 Web 请求和响应，如果同一项目中有类似视图渲染的需求，则无法满足需求。  
  
所以，为了能够不打破开发者对 SpringMVC 框架以及 SpringBoot 提供的 Web 应用各项功能支持的认知，最稳妥的做法是，在 SpringBoot 原有 Web 应用默认配置的基础上增加新的 HttpMessageConverter，专门处理 Web API 响应结果使其符合我们的 Web API 规范形式。  
  
要达到这个目的，我们可以提供自定义的配置：

1. @Configuration
2. **public** **class** WebApiConfiguration **extends** WebMvcConfigurerAdapter {
3. @Override
4. **public** void extendMessageConverters(List<HttpMessageConvert-er<?>> converters) {
5. // 添加或者插入我们自定义的HttpMessageConverter实现类
6. // converters.add(converter)或者converters.add(0, converter)
7. }
8. }

extendMessageConverters 属于已经添加过默认 HttpMessageConverter 序列的参数（比如针对 String 的 HttpMessageConverter，或者针对 byte[] 的 Http-MessageConverter 等），所以，我们只要在其基础上添加或者插入我们的 HttpMessageConverter 实现类就可以了。  
  
不过，这里有一个比较尴尬的地方，这可能也是 Spring 框架多处设计中都存在的尴尬，即循环条件判断应用哪个类的时候，条件判断 API 开放不足：

1. **for**(HttpMessageConverter converter: converters){
2. **if**(converter.canWrite(clazz, media)) {
3. converter.write(..);
4. }
5. }

在 HttpMessageConverter 的场景中就是，我们只能根据目标对象的类型以及 mediaType 来判断是否应该使用当前这个 HttpMessageConverter，如果需要在这两种判断条件都相同的情况下，还要根据其他条件来判定是否应该使用当前 HttpMessageConverter，此时这种设计显然就无法满足需求了。尴尬之处就在于此。  
  
对于我们的 Web API 规范这个实现场景来说，如果想继续享受原有的 MappingJackson2HttpMessageConverter 提供的功能和配置，就不得不继承并覆写（Override）相应方法，而不是略过 MappingJackson2HttpMessageConverter，然后在另一个 HttpMessageConverter 中只是必要的时候引用它（组合优于继承）。  
  
不管怎么样，我们推荐使用隐式的自动转换方式为用户提供透明的 Web API 规范行为。

**3. Web API 的短板和补足**

相对于 Dubbo 这种强类型的服务框架，Web API 有强类型支持（Not Typesafe），在开发过程中，自然也无法享受到像 IDE 自动提示之类的功能，所以，对于 Web API 的使用者来说，需要与 Web API 的提供者沟通之后才能知道如何访问 Web API 的详细信息，比如应该传哪些参数，返回的响应结果又应该是什么格式的。  
  
为了缓解这个问题，我们可以使用自动根据代码元信息生成 API 文档的方式来补足这块短板，像 [Swagger](http://swagger.io/) 这样的项目，已经是比较成熟的 API 文档方案了。  
  
不过，让每一个 Web API 项目都自己去初始化 API 文档相关的设置显然并不是很好的用户体验，为了服务到位，我们可以遵循 SpringBoot 的行事风格，新建一个 spring-boot-starter-webapi 这样的自动配置模块，其提供的主要特性包括但不限于：

* 提供针对我们 Web API 规范的功能支持，即提供显式的强类型封装方式或者隐式的自动转换方式的功能实现。
* 提供 API 文档相关功能的配置和设置。
* 提供统一的 Web API 访问错误处理逻辑。

这样，任何 Web API 的开发者和提供者只要新建 SpringBoot 应用，然后依赖 spring-boot-starter-webapi，就可以自动享有以上所有特性支持了。  
  
以下是一个 spring-boot-starter-webapi 原型项目的 pom.xml 定义：

1. <?xml version="1.0" encoding="UTF-8"?>
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
5. **<modelVersion>**4.0.0**</modelVersion>**
6. **<parent>**
7. **<groupId>**org.springframework.boot**</groupId>**
8. **<artifactId>**spring-boot-starters**</artifactId>**
9. **<version>**1.2.5.RELEASE**</version>**
10. **</parent>**
11. **<groupId>**com.keevol.springboot**</groupId>**
12. **<artifactId>**spring-boot-starter-webapi**</artifactId>**
13. **<version>**1.0.0-SNAPSHOT**</version>**
14. **<packaging>**jar**</packaging>**
15. **<name>**spring-boot-starter-webapi**</name>**
16. **<url></url>**
17. **<properties>**
18. **<java.version>**1.8**</java.version>**
19. **<file.encoding>**UTF-8**</file.encoding>**
20. **</properties>**
21. **<build>**
22. **<plugins>**
23. **<plugin>**
24. **<groupId>**org.apache.maven.plugins**</groupId>**
25. **<artifactId>**maven-compiler-plugin**</artifactId>**
26. **<version>**3.2**</version>**
27. **<configuration>**
28. **<source>**${java.version}**</source>**
29. **<target>**${java.version}**</target>**
30. **<encoding>**${file.encoding}**</encoding>**
31. **</configuration>**
32. **</plugin>**
33. **</plugins>**
34. **</build>**
35. **<dependencies>**
36. **<dependency>**
37. **<groupId>**org.springframework.boot**</groupId>**
38. **<artifactId>**spring-boot-starter-web**</artifactId>**
39. **</dependency>**
40. **<dependency>**
41. **<groupId>**io.springfox**</groupId>**
42. **<artifactId>**springfox-swagger2**</artifactId>**
43. **<version>**2.1.2**</version>**
44. **</dependency>**
45. **<dependency>**
46. **<groupId>**io.springfox**</groupId>**
47. **<artifactId>**springfox-swagger-ui**</artifactId>**
48. **<version>**2.1.2**</version>**
49. **</dependency>**
50. **<dependency>**
51. **<groupId>**javax.servlet**</groupId>**
52. **<artifactId>**javax.servlet-api**</artifactId>**
53. **<version>**${servlet-api.version}**</version>**
54. **</dependency>**
55. **</dependencies>**
56. **</project>**

以及对应的 JavaConfig 配置类示例：

1. @Configuration
2. @EnableSwagger2
3. @ComponentScan("com.wacai.springboot.webapi.errors")
4. @AutoConfigureAfter(WebMvcAutoConfiguration.**class**)
5. **public** **class** WebApiAutoConfiguration **extends** WebMvcConfigurerAdapter {
6. **protected** Logger logger = LoggerFactory.getLogger(WebApiAuto - Configuration.**class**);
7. @Value("${springfox.api.group:[your api group name]}")
8. **private** String apiGroupName;
9. @Value("${springfox.api.title:[set a api title via 'springfox.api.title']}")
10. **private** String title;
11. @Value("${springfox.api.description:[add your api description via 'springfox.api.description'}]")
12. **private** String desc;
13. @Value("${springfox.api.version:[set specific api version via 'springfox.api.version'}]")
14. **private** String version;
15. @Value("${springfox.api.termsOfServiceUrl:[set termsOf-ServiceUrl via 'springfox.api.termsOfServiceUrl']}")
16. **private** String termsOfServiceUrl;
17. @Value("${springfox.api.contact:[set contact via 'springfox.api.contact'}]")
18. **private** String contact;
19. @Value("${springfox.api.license:Your WebAPI License}")
20. **private** String license;
21. @Value("${springfox.api.licenseUrl:http://keevol.com}")
22. **private** String licenseUrl;
23. @Autowired
24. **private** TypeResolver typeResolver;
25. @Bean
26. **public** Docket api() {
27. **return** **new** Docket(DocumentationType.SWAGGER\_2).groupName(apiGroupName)
28. .apiInfo(**new** ApiInfo(title, desc, version, termsOf - ServiceUrl, contact, license, licenseUrl)).select()
29. .apis(RequestHandlerSelectors.any()).paths(excludedPathSelector()).build().pathMapping("/")
30. .directModelSubstitute(Date.**class**, String.**class**).genericModelSubstitutes(ResponseEntity.**class**)
31. .alternateTypeRules(newRule(
32. typeResolver.resolve(DeferredResult.**class**,
33. typeResolver.resolve(ResponseEntity.**class**, WildcardType.**class**)),
34. typeResolver.resolve(WildcardType.**class**)))
35. .useDefaultResponseMessages(**false**)
36. .globalResponseMessage(RequestMethod.GET, newArrayList(**new** ResponseMessageBuilder().code(500)
37. .message("服务出错啦~").responseModel(**new** ModelRef("Error")).build()))
38. .forCodeGeneration(**true**);
39. }
40. // ...
41. }

关于如何将 WebApiAutoConfiguration 配置到 META-INF/spring.factories 并发布项目则不再赘述。有了 spring-boot-starter-webapi 之后，Web API 形式的微服务开发者所要做的仅仅是把它加为项目依赖：

[纯文本复制](http://c.biancheng.net/view/4670.html)

1. **<dependency>**
2. **<groupId>**com.keevol.springboot**</groupId>**
3. **<artifactId>**spring-boot-starter-webapi**</artifactId>**
4. **<version>**1.0.0-SNAPSHOT**</version>**
5. **</dependency>**

然后像往常那样写 SpringMVC 的 @Controller 或者 @RestController 就可以了，现在，我们可以直接享受 API 文档的自动生成。

## 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）

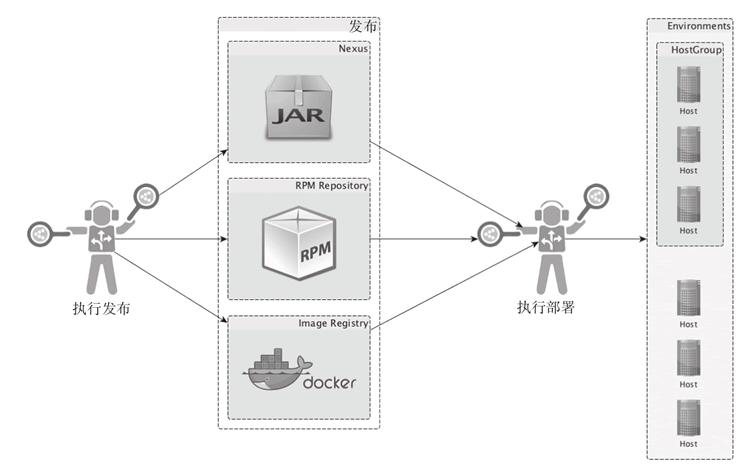
基于 [SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 的微服务开发完成之后，现在到了把它们发布并部署到相应的环境去运行的时候了。  
  
[Spring](http://c.biancheng.net/spring/)Boot 框架只提供了一套基于可执行 jar 包（executable jar）格式的标准发布形式，但并没有对部署做过多的界定，而且为了简化可执行 jar 包的生成，SpringBoot 提供了相应的 [Maven](http://c.biancheng.net/maven/) 项目插件：

1. **<build>**
2. **<plugins>**
3. **<plugin>**
4. **<groupId>**org.springframework.boot**</groupId>**
5. **<artifactId>**spring-boot-maven-plugin**</artifactId>**
6. **</plugin>** <!-- 其他插件定义 -->
7. **</plugins>**
8. **</build>**

然后只要我们运行 mvn package，当前 SpringBoot 项目就会被打包成一个包含了其所有项目依赖以及该项目本身的可执行 jar 包，通过 scp 或者 rsync 等方式将这个可执行 jar 包部署到目标环境的服务器之后，就可以通过 java-jar your-project.jar 启动 SpringBoot 应用了。  
  
整个流程看起来很简单，也很符合大部分开发人员的认知，但是，相对于一套较为严谨的软件交付流程来说，以上流程则难免过于粗糙了。  
  
软件的发布和部署可以有多种不同的形式，这更多由软件项目的属性决定，比如：

* 这个项目使用的是什么语言？
* 这个项目属于类库项目还是可独立运行的项目？
* 这个项目是面向什么平台和环境的项目？

此外，我们希望使用什么样的形式进行软件的交付，这里则涉及生态管理以及技术选型的喜好等因素，所以，为了降低讲解的复杂度，我们还是先将发布和部署分开来说吧。  
  
首先，大家应该都知道，发布并不等于部署，这是两个阶段的事情，如图 1 所示。

  
图 1  发布与部署示意图

发布一般是将项目以指定的格式打包成某种可直接交付的形式，然后放置到预先指定的交付地点。  
  
比如对于 [Java](http://c.biancheng.net/java/) 类库（Java Library）来说，我们一般将其打包成 jar 包，然后 mvn deploy 到公司内部的 Maven 仓库中（Maven Repository），像 Nexus Repository Manager 或者 JFrog Artifactory 以及 Apache Archiva。  
  
而对于可独立运行的程序，比如 SpringBoot 微服务或者一般的 Java Standalone 程序，我们既可以将它们打包成 RPM、DEB 等面向特定目标系统的发布形式，也可以将它们制作成一个个的 docker images，然后将制作完成的发布成品存储到相应的仓库中（Repository）去。  
  
部署一般紧接着发布完成之后进行，它的主要职能就是将已经发布好的成品从仓库中拿出来，然后分发到目标环境的指定资源池（比如物理机结点，虚拟机结点，docker 宿主机等），并最终启动服务。  
  
软件成品分发的手段和工具可以有很多种，从最常见的 scp、rsync，到 Chef、Puppet，进而再到最新的 saltstack、ansible 等，一般根据团队对这些工具的把控力度和喜好进行选型。  
  
下面我们就几种典型的发布和部署形式跟大家一起探索相应的实践。

### spring-boot-starter 的发布与部署方式

spring-boot-starter（s）属于 Java 类库性质的组件，只被其他可独立运行的程序依赖使用，自身不可独立运行，对于这种性质的软件实体，我们一般将其发布到公司内部的软件仓库或者以开源形式发布到 Maven 的中央仓库（Maven Central Repository）。  
  
下面我们就以 spring-boot-starter-metrics 为例，向大家展示如何将类似 spring-boot-starter-metrics 这样的 Java 类库发布到自己公司内部搭建的 Nexus 服务器上。  
  
首先，你要有一套已经搭建完成并运行的 Nexus 服务器，然后我们需要对 spring-boot-starter-metrics 的 pom.xml 附加一点儿发布相关的内容：

1. <?xml version="1.0" encoding="UTF-8"?>
2. **<project** xmlns="http://maven.apache.org/POM/4.0.0"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
5. **<modelVersion>**4.0.0**</modelVersion>**
6. **<groupId>**com.keevol**</groupId>**
7. **<artifactId>**spring-boot-starter-metrics**</artifactId>**
8. **<version>**0.0.1-SNAPSHOT**</version>**
9. **<packaging>**jar**</packaging>**
10. **<name>**spring-boot-starter-metrics**</name>**
11. **<description>**auto configuration module for dropwizard metrics**</description>**
12. **<parent>**
13. **<groupId>**org.springframework.boot**</groupId>**
14. **<artifactId>**spring-boot-starter-parent**</artifactId>**
15. **<version>**1.3.0.RELEASE**</version>**
16. **<relativePath** **/>** <!-- lookup parent from repository -->
17. **</parent>**
18. **<distributionManagement>**
19. **<repository>**
20. **<id>**deployment**</id>**
21. **<name>**internal repository for releases**</name>**
22. **<url>**http://{内部nexus服务器地址}/nexus/content/repositories/ releases/**</url>**
23. **</repository>**
24. **<snapshotRepository>**
25. **<id>**deployment**</id>**
26. **<name>**internal repository for snapshots**</name>**
27. **<url>**http://{内部nexus服务器地址}/nexus/content/repositories/ snapshots/**</url>**
28. **</snapshotRepository>**
29. **</distributionManagement>**
30. **<properties>**
31. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
32. **<java.version>**1.8**</java.version>**
33. **<metrics.version>**3.1.2**</metrics.version>**
34. **</properties>** <!--其他配置 -->
35. **<dependencies>**
36. **<dependency>**
37. **<groupId>**org.springframework.boot**</groupId>**
38. **<artifactId>**spring-boot-starter**</artifactId>**
39. **</dependency>**
40. **<dependency>**
41. **<groupId>**org.springframework.boot**</groupId>**
42. **<artifactId>**spring-boot-starter-aop**</artifactId>**
43. **</dependency>**
44. **<dependency>**
45. **<groupId>**org.springframework.boot**</groupId>**
46. **<artifactId>**spring-boot-starter-actuator**</artifactId>**
47. **</dependency>**
48. **<dependency>**
49. **<groupId>**io.dropwizard.metrics**</groupId>**
50. **<artifactId>**metrics-core**</artifactId>**
51. **<version>**${metrics.version}**</version>**
52. **</dependency>**
53. **<dependency>**
54. **<groupId>**io.dropwizard.metrics**</groupId>**
55. **<artifactId>**metrics-annotation**</artifactId>**
56. **<version>**${metrics.version}**</version>**
57. **</dependency>**
58. **<dependency>**
59. **<groupId>**org.aspectj**</groupId>**
60. **<artifactId>**aspectjrt**</artifactId>**
61. **<version>**1.8.7**</version>**
62. **</dependency>**
63. **</dependencies>**
64. **</project>**

主要关注我们添加的 distributionManagement 相关内容，用于将我们的当前项目与内部的 Nexus 服务器进行关联，这样，就可以将当前项目不同阶段的成品（比如 SNAPSHOT 版本或者 RELEASE 版本）发布到特定的仓库路径下。  
  
但是，只是在项目的 pom.xml 中添加 distributionManagement 相关内容还不够，对发布服务器的安全管控等敏感信息不能与 pom.xml 一同公开，所以，还需要在 ~/.m2/settings.xml 配置文件中添加 Nexus 服务器访问和认证信息：

1. **<server>**
2. **<id>**deployment**</id>**
3. **<username>**deployment**</username>**
4. **<password>**\*\*\*\*\*\*\*\***</password>**
5. **</server>**

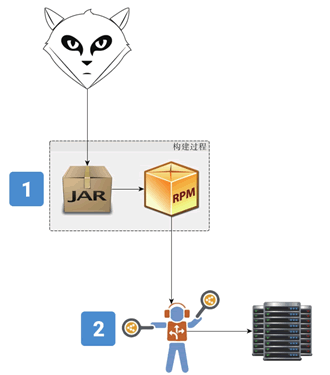
因为我们前面 distributionManagement 定义的 repository 和 snapshotRepository 的 id 都是 deployment，所以，这里的 server 的 id 也是匹配性地指定为 deployment，至于 username 和 password，则完全是我们内部的 nexus 服务器对应的安全认证用的用户名和密码啦。  
  
将内部 Nexus 服务器的认证信息放到 maven 的 settings.xml 中并非什么好的实践，纯粹是为了便利性而牺牲安全性，二者之间需要根据情况做出权衡，如果对安全性要求比较高的公司或者组织，最好将这些认证信息移除，并只在管控的范围内使用。  
  
比如将这些认证信息回收到发布和部署平台这一可控的小范围环境中，而所有开发人员使用的 settings.xml 属于“消毒”后无安全认证等敏感信息的版本。  
  
当 pom.xml 中的 distributionManagement 以及 settings.xml 中对应的 server 设定都准备好之后，我们就可以直接 mvn deploy 将 spring-boot-starter-metrics 或者类似的 Java Library 项目发布到内部 Nexus 仓库了。  
  
对于 Java 类库类型的项目来说，并无明确的部署过程，如果说有，也是存在于可独立运行项目的开发过程中，比如使用 lib 目录或者结合 Ant“部署”为项目的依赖，或者直接享受 maven、gradle、sbt 等编译工具提供的“透明”的依赖部署过程。

### 基于RPM的发布与部署方式

部署的目标服务器从硬件到系统软件，一般情况下都应该是尽量相同，这与软件的标准化目的相同，一个是可以减少应对不同类型实体的复杂度，另一个就是标准化硬件和软件之后，就可以通过工具批量化以“边际成本递减”近乎为 0 的做法来提升效率，减少成本。  
  
所以，对于大部分互联网公司来说，在硬件标准化的基础上，还会使操作系统尽量统一，比如大多数都是使用稳定性和可靠性经过长期验证过的 Red Hat CentOS 系统，而 CentOS 本身经过长期的沉淀也有一套自己的系统管理工具，比如像 YUM 或者 RPM 这样的系统包依赖管理器（Debian/Ubuntu 等 Linux 发行版也有对应的 deb 形式的包管理器）。  
  
像 RPM 这样的包管理器对系统软件包的依赖和配置提供了很好的支持，如果我们的微服务等可独立执行实体要部署到像 CentOS 这样的目标环境中，使用 RPM 完成微服务的发布和部署，对于运维人员来说几乎就是无缝衔接的。  
  
而且，对于 SpringBoot 微服务来说，单单一个可执行的 jar 包实际上是远远无法达到发布和部署要求的，如果只是发布一个可执行的 jar 包，那就意味着在部署阶段，运维要做更多的事情来弥补某些缺失，比如：

* 启动参数是否调整？
* 配置文件是否修改？
* 安装部署结构如何规范？
* 资源的对接和映射要不要做？

但是，如果我们能够将整个软件交付体系标准化和规范化，然后通过 RPM 这样的发布形式将这些标注和规范固化到发布包中，那么，整个部署过程就可以简化为一条命令，这其实也是使用 RPM 这种系统原生包管理工具完成交付部署的好处：自动化的 orcherstration，减少不必要的人工干预中间环节。  
  
使用 RPM 发布 SpringBoot 微服务，我们简单将这一个过程划分为几步，如图 2 所示。

  
图 2  使用 RPM 交付的 SpringBoot 微服务发布流程图

首先，我们需要有一个特定的编译和项目构建环境，可以不管这个编译和项目构建环境是搭建在本地（比如你的开发机上），还是搭建在特定的一台服务器上，但这个编译和项目构建环境需要安装 rpmbuild，用来构建 rpm 包。  
  
其次，我们不推荐 RPM 编译和构建的过程使用某些定义在项目编译脚本中的插件来完成，这样会让一些通用的逻辑散落在所有需要发布的项目中而不好治理。所以，我们建议使用一个外部化的独立的编译服务器完成整个 SpringBoot 微服务的 RPM 发布和部署。  
  
SpringBoot 微服务的 rpmbuild 脚本构建过程主要分几个主要步骤（如图 2 中 1 所标注）：  
  
1）调用标准的 mvn package 完成可执行 jar 包的打包。  
  
2）根据软件交付规范，构建标准发布格式的 rpm 包，如下所示：

* 使用 bin 目录存放根据脚本模板以及环境变量生成的启停脚本。
* 使用 config 目录（SpringBoot 默认文件系统中的配置目录名）或者 conf 目录存放特定的配置文件。
* 使用 docs 目录存放文档。
* 使用 agents 目录存放某些 javaagent。
* ……

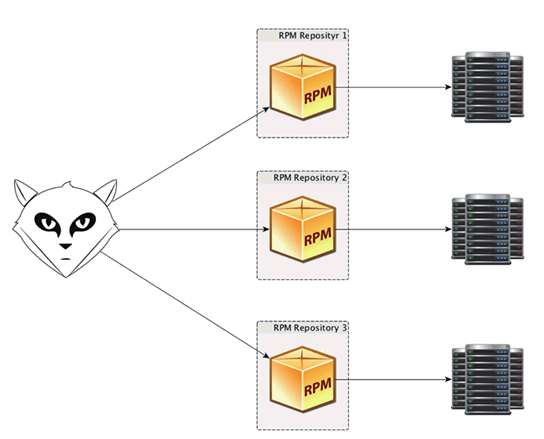
3）在标准发布格式的基础上，生成从标准发布格式到具体目标环境的映射。  
  
比如原来应用的日志是默认打印到当前项目部署目录，而根据要求，我们希望打印到 /var/logs/{projectId}/ 或者其他服务器磁盘容量分配更大的分区，这个时候，可以在 rpmbuild 过程中指定安装类似的安装规则。  
  
下面是一个简化的 SpringBoot 微服务的 rpmbuild 脚本定义：

Summary: metrics autoconfigure module for spring boot  
applicationName: spring-boot-starter-metricsVersion: {version}Release: 1Copyright: ...Group: Applications/ProductivitySource: ...URL: ...Distribution: Vendor: KEEp eVOLution, Inc.Packager: Darren <afoo@keevol.com>%description%prep%buildgit clone ....  
# 检出代码到本地cd {project folder}mvn package ...%install%clean%files/{install\_location}/{projectId}/bin/start.sh/{install\_location}  
/{projectId}/bin/stop.sh/{install\_location}/{projectId}/agents/jolokia-jvm-1.3.1-agent.jar...%attr(755, user, group) /{install\_location}/{projectId}/agents/jolokia-jvm-1.3.1-agent.jar...%doc%changelog

然后需要调用 rpmbuild 的 spec 定义完成最终 rpm 包的构建：

rpmbuild -bb {projectId}.spec#

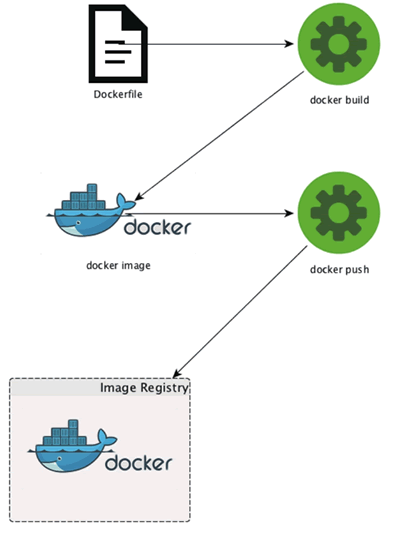
然后 scp or sftp 生成的 rpm 包到指定的 rpm 仓库 我们可以像上面那样直接执行 rpmbuild 命令完成最终的 rpm 包构建，也可以将这些逻辑纳入编译构建脚本并部署到像 Jenkins 这样现成的持续集成服务器上，总之，执行完成后，打包好的 rpm 就发布到目标环境对应的 rpm 仓库了。  
  
rpm 包发布到 rpm 仓库之后，就可以执行部署，比如通过 Salt 或者 ansible 在目标环境执行 rpm 或者 yum 命令，但具体的部署行为可能因为不同开发者的习惯和理念而有所不同。  
  
有的开发者喜欢将不同目标环境的配置都一股脑地打包到发布包中，然后通过配置文件的命名和启动程序时单独指定一个环境变量来决定如何启用哪一个配置文件，对于这种做法，只需要打一个 rpm 包，同时也只需要搭建一个内部的 rpm 仓库，部署的时候，则需要运维人员根据具体的操作环境传递相应的环境变量来启停程序。  
  
有的开发者则认为，一个软件实体发布的时候就应该是针对目标环境“装配”完备的，rpm 包中的各项配置都是针对特定目标环境配置好的，只要将 rpm 包部署到目标环境，就可以直接启动，启停完全无差别操作，唯一的差别是，rpm 包分别是根据目标环境发布和部署到不同的 rpm 仓库的，如图 3 所示。

  
图 3   “根据不同交付目标环境，设置不同RPM仓库作为交付地点”示意图

以上两种策略并无优劣之分，但却有各自适合的场景：  
  
1）在团队小，以人为本的时候，前者更适合，原因在于，整个软件交付链路更多是通过开发人员来协调和完成的。所以，开发、测试、运维一把抓，即使是不同环境的配置文件，也都是为了开发人员方便，直接放到了项目目录下一起管理和修改。  
  
当到了线上，开发人员同时担当运维人员的角色，启停程序的可控性也很高，所以，可以在熟知自身程序的前提下，很好地完成整个链路的工作。  
  
2）随着团队规模的扩张，职能更加明确，交付链路要承担的关注点也更多的时候，为了保证整个软件的交付质量，需要引入规范化的流程来关联和约束整个链路上各个环节和团队之间的工作。  
  
这个时候，开发人员的职责范围将缩小到明确的范围，测试团队、安全团队、应用运维团队等也将加入并根据流程各司其职，每个团队之间的工作需要横向关联的同时，又需要垂直隔离。  
  
这个时候，我们就需要从交付的源头一直到发布和部署，根据环境进行隔离，每个人即使只关注自己负责的事情，也可以让整个软件交付链路很好地工作，这考虑的是规范和流程对整体粒度上的把控和支撑。  
  
对于小团队来说，微服务并不是什么太好的选择，高内聚的应用开发和部署单元，对整个交付链路的要求没那么高，也不需要更多自动化和平台化层面的投入和支持。  
  
而一旦你选择了微服务的软件交付策略，数量庞大的微服务治理将耗费更多资源在支撑整个交付链路的自动化和平台化建设层面。否则，如此数量上的差异化的实体管理，单纯还靠人工拼苦劳是“捞不着好果子吃的”。  
  
所以，我们要标准化和规范化微服务的开发、交付、部署以及运维，从而收敛整条链路的治理复杂度，以近乎无差别的方式，完成各个环节上的工作。这个时候，规范、流程、平台是核心，对人的要求则适当降低。  
  
说了这么多，其实就一点，如果团队要转向微服务的交付策略，那么，标准化、单一化的微服务发布和部署行为是大家努力的方向，虽然我们在说基于 RPM 这种特定的微服务发布和部署形式，但并不意味着我们只应该关注这一点或者这单一环节，只有系统的从整体的微服务交付链路和体系层面考虑，才能够在各个单一环节落地的时候选择合适的方案。  
  
基于 RPM 的发布与部署方式就说这么多，希望对大家有所启发。

### 基于 [Docker](http://c.biancheng.net/docker/) 的发布与部署方式

随着资源虚拟化技术的持续精进，一种基于容器（container）的方式开始风行，Docker 就属于当下这个风口上最耀眼的明星，而且，很多微服务相关的文章也是言必提 Docker，好像没有 Docker 的微服务就不是正宗的微服务了，所以，我们自然也要适当提及一下如何结合 Docker 发布和部署我们的 SpringBoot 微服务。  
  
我们知道，RPM 包的构建是使用系统的 rpmbuild 工具完成的，该工具需要一个构建描述文件，即 .spec 文件，rpmbuild 工具读取 .spec 构建描述文件之后，根据构建描述文件生成一个 rpm 包，然后我们就可以把 rpm 包发布到相应的 RPM 仓库（RPM Repository）。  
  
使用 Docker 其实也是类似的过程，如图 4 所示。

  
图 4  基于Docker的SpringBoot微服务发布流程图

我们需要提供一个 Dockerfile 用于描述 Docker 发布成品的构建过程，这类似于 rpmbuild 需要的 .spec 文件。  
  
在编写好要使用的 Dockerfile 之后，我们使用 docker build 命令读取 Dockerfile 开始构建一个 Docker 的 image，docker build 完成 rpmbuild 类似的功能，Docker 的 image 则类似于 rpm 包，即 Docker 的软件发布成品。  
  
有了 docker image 之后，我们就可以将其发布到一个 Docker 的 image registry，这里的 image registry 就类似于 RPM 仓库（RPM Repository），而将 docker image 发布到 image registry 的过程，可以通过 docker push 完成。  
  
所以，假设要以 Docker 的形式发布我们的汇率查询 SpringBoot 微服务，首先需要编写一个对应的 Dockerfile 来构建相应的 docker image：

FROM java:8MAINTAINER AFOO <afoo@afoo.me>LABEL  
groupId=...LABEL artifactId=...LABEL  
version=...LABEL ...USER deployerEXPOSE 8080ENV  
{key}={value}...VOLUME ...RUN mkdir /{group}/{projectId}/configRUN mkdir /{group}/{projectId}/agentRUN mkdir /{group}/{projectId}/docsRUN mkdir /{group}/{projectId}/libCOPY target/docker-springboot-chapter4-0.0.1-SNAPSHOT.jar /{group}/{projectId}/lib/docker-springboot-chapter4-0.0.1-SNAPSHOT.jarCOPY conf/application.properties /{group}/{projectId}/config/application.propertiesCOPY agent/jolokia-jvm-1.3.3-agent.jar /{group}/{projectId}/jolokia-jvm-1.3.3-agent.jar...ENTRYPOINT ["java", "...", "-jar", "lib/docker-springboot-chapter4-0.0.1-SNAPSHOT.jar"]

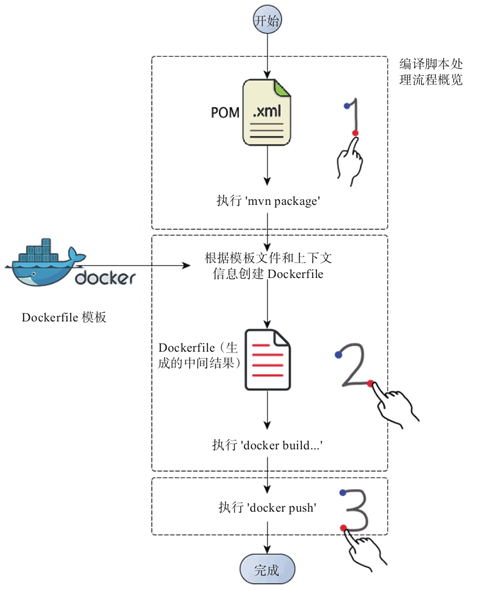
之后，我们就可以使用这个 dockerfile 来进行构建并发布了：

$ docker build . -t "{groupId}/{artifactId}:{version}

$ docker push 基于 Docker 的微服务部署与 RPM 类似，都是从发布仓库中拉取发布的成品，并在目标环境安装部署，一般情况下我们也同样是使用 Salt 或者 Ansible 之类的工具执行如下类似的命令完成基于 Docker 的微服务的部署：

$ ansible {cluster} -m shell -a "cd /microservices/{groupId}/{artifactId}; docker pull"

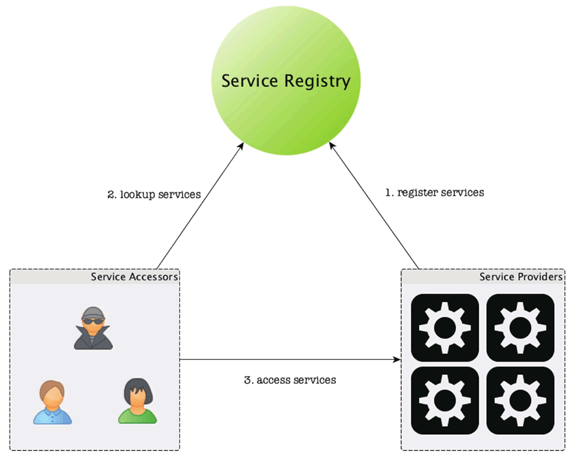
为了简化基于 Docker 的发布和部署流程，实际上，以上演示的只是单人单微服务项目的方法，在讲究集团军作战的微服务场景下，我们希望的是能够快速、批量且标准化的形式完成数量巨大的微服务发布和部署。  
  
这就要求不能只盯着单一项目内部去思考如何实现发布和部署，而应该将视线从单一项目内部抽取出来，以更高的视角来审视如何快速地完成批量微服务的发布和部署。  
  
笔者建议的一个思路是适当地弱化 Docker 属性，将发布和部署逻辑外部化到发布脚本中。  
  
外部化后的发布脚本将集中协调 Docker 基础设施，要发布的微服务上下文信息以及其他中间步骤，将微服务项目与 Docker 挂钩的唯一纽带也仅仅是一个模板化、标注化后的 Dockerfile，整个过程如图 5 所示。

  
图 5  外部化的基于Docker的SpringBoot微服务发布脚步逻辑流程图

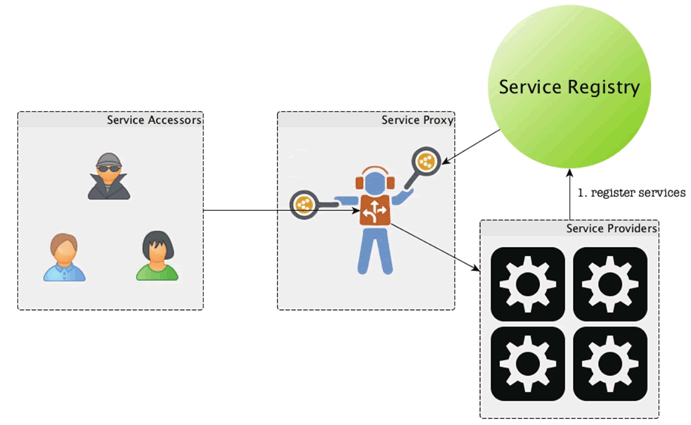
如此一来，对于所有希望以 Docker 形式发布的标准化的微服务来说，一套发布脚本即可完成所有微服务的发布和部署，而不需要每一个微服务自己去编写 Dockerfile 甚至发布脚本。  
  
上面的 Docker 实践并非 Docker 社区建议的最佳实践方式，这里更多只是为了简化说明和对比，Docker 背后是一套更为庞大完备的体系，比如 Docker 容器的注册和发现，容器的编排及调度等功能和系统，限于篇幅和内容定位，这里不再赘述。  
  
总的来说，其实基于 Docker 的微服务发布和部署与其他形式从本质上来说没有太大差别，唯一的差别只是各自方案特定的实现不同而已。  
  
大部分人选择和认同 Docker 方案，更多是从系统资源利用率以及 Docker 对整个软件交付链路的支撑体系比较完备这些角度考虑的，而微服务自身的很多特点以及需求（比如隔离、轻量），恰好与 Docker 能提供的相互吻合，或许这就是二者经常被“相提并论”的原因。  
  
不过，对于 Java 应用和微服务来说，Docker 能给予的好处可能没有想象的那么多。

## 23. SpringBoot微服务的注册与发现

[SpringBoot](http://c.biancheng.net/spring_boot/) 微服务部署到相应环境之后，即开始启动并对外服务。既然选择了微服务，那么就意味着很少会让某个微服务单兵作战，而是同一个微服务启动多个实例形成服务集群。  
  
然后让服务集群作为一个逻辑服务主体提供对外服务，而构建这个逻辑服务主体的过程，就是微服务的注册过程，服务的访问者如何访问到这个逻辑服务主体的过程，则是服务发现的过程。  
  
服务的注册与发现从结构上来说是很简单的三角关系，如图 1 所示。

  
图 1  服务注册与发现功能实现的典型结构图

其中，提供服务注册的服务是整个三角关系的核心，它负责登记和保存哪些服务是可以对外提供服务的，当服务访问者（Service Accessors）想要访问某个服务的时候，需要先向服务注册服务（Service Registry）询问一下：“我想访问服务 A，请问现在都有哪些服务结点提供在线的 A 服务？”，然后服务注册服务就会返回服务 A 在线的服务结点，这样服务访问者就可以根据这些服务结点的信息直接访问相应的服务了。  
  
[Spring](http://c.biancheng.net/spring/)Boot 微服务的注册与发现由 SpringBoot 微服务提供服务的方式来决定，比如，如果我们使用的是基于 Dubbo 框架的微服务实现方式，那么，我们的 SpringBoot 微服务可能采用的是基于 [Redis](http://c.biancheng.net/redis/) 或者 Zookeeper 的注册与发现机制。  
  
如果我们提供的是 Web API 形式的微服务，那么，我们的 SpringBoot 微服务采用的则更多可能是基于 DNS 的服务注册与发现机制。  
  
虽然服务注册与发现的机制从原理上来说都是相同的，但具体实施的时候，方案上可以进行一些微调，这种微调主要以服务的发现者与服务的访问者角色和位置是否分离来区分。  
  
一般情况下，服务的访问者同时也是服务的发现者，这通常以分布式系统的客户端为典型代表，比如 [HBase](http://c.biancheng.net/hbase/) 的客户端，以及 Dubbo 的客户端代理。  
  
这种模式的典型拓扑结构是，服务的访问者同时需要了解或者持有服务发现的逻辑，从而需要根据服务的发现逻辑首先去请求注册服务获得服务提供方的相应信息，然后再使用这些信息去访问服务提供者，这种模式很常见，但从服务访问者角度来看（实际上也是产品的用户角度），这不会带来最好的产品体验。  
  
如果我们将提供的服务也看成一种（技术）产品，那么，一种好的产品设计应该是尽量减少用户的学习成本。也就是说，服务的访问者就是产品的使用者，他们不应该也不需要了解服务端有多少服务实例，这些服务实例又是使用什么方式实现注册的。  
  
注册之后又是通过什么方式发现这些实例并使用的，服务的访问者原始需求就一个，给我一个服务接口，我直接调用，不管你接口之后结点如何增减，部署结构如何复杂，都不需要关注，如果遵循这样的技术产品设计与实现理念，我们可以将服务的注册与发现完全屏蔽在服务端，这通常以一种服务代理的形式实现，如图 2 所示。

  
图 2  向用户屏蔽服务注册与发现实现细节的技术产品设计示意图

经过这样的微调之后，服务的访问者只需要与服务代理打交道，至于服务代理如何发现服务并路由服务的访问，对于服务的访问者来说都是透明的。  
  
产品接口与产品实现要分得清楚，服务接口与服务实现也是同样的道理。原来将服务的访问与服务的发现都放在客户端的做法，我认为更多是一种偏工程师思维的实现方式，而如果我们稍微结合一些产品思维，我认为后者是一种更为合理的服务注册发现实践方式。  
  
从这个角度来看，在服务器端，HTTP 配合 DNS 或者 DNS-SD 或许是当下最有产品“范儿”的普适方案了，像 [consul](https://www.consul.io/) 就是这种类型的服务注册与发现服务。  
  
当然，SpringBoot 不会对任何一种服务注册与发现方案“挑食”，最终还是看大家选择的具体服务注册与发现方案是哪种，但尽量不要将实现逻辑外溢到服务、系统或者产品之外。

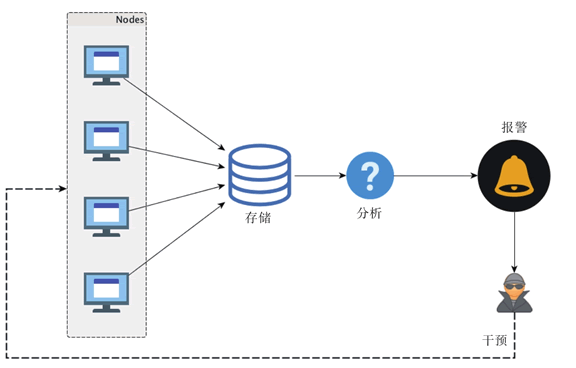
## 24. SpringBoot微服务的监控与运维

与大部分应用和系统一样，[SpringBoot](http://c.biancheng.net/spring_boot/" \t "_blank) 微服务的开发、发布与部署只占其生命周期的一小部分，应用和系统运维才是重中之重。而运维过程中，监控工作更是占据重要位置。  
  
运维的目的之一是为了保证系统的平稳运行，进而保障公司业务能持续对外服务，为了达到这一目的，我们需要对系统的状态进行持续地观测，以期望一有风吹草动就能发现并作出应对，监控作为一种手段，就是以此为生。  
  
我们会从以下多个层面对 [Spring](http://c.biancheng.net/spring/" \t "_blank)Boot 微服务进行监控：

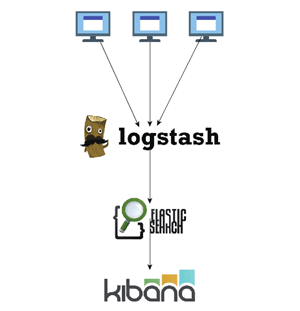
* 硬件层面
* 网络层面
* 系统层面
* SpringBoot 微服务的应用层面
* 服务访问层面

我们会从所有这些层面采集相应的状态数据，然后汇总，存储，并分析，一旦某项指标超出规定的阈值，则报警，在接收到报警通知之后，我们需要做出应对以改变现在系统状态不健康的局面，这一般通过预置的调控开关来调整应用状态，要么重启或者服务降级，也就是执行监控的“控”，整个过程如图 1 所示。  
  
硬件、网络以及系统层面的监控，现有的一些监控系统和方案已经可以很好地提供支持，比如开源的 Zabbix 系统或者以报警为强项的 Nagios 系统。  
  
本节不对这些层面的监控做过多介绍，我们将更多对 SpringBoot 微服务应用层面的监控进行实践方案的探索。SpringBoot 微服务的内部状态，通过多种方式或者渠道可以知道。  
  
打印的应用日志是一种 SpringBoot 微服务运行状态的反映形式。

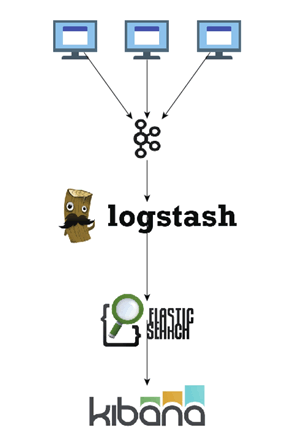
* SpringBoot 微服务内部设置的一系列“传感器”可以为外界提供某些指标的状态数据。
* SpringBoot 微服务内部追踪的一些 metrics 数据也是反映运行状态的一种方式。

  
图 1  微服务基本监控体系示意图

任何可以反映 SpringBoot 微服务运行状态的数据，对于监控来说都是十分重要的“财产”，都应该尽量采集上来进行分析，从而在分析的基础上谋求对 SpringBoot 微服务的改进。  
  
对于应用日志来说，在单机单结点的年代，我们只要登录应用部署的服务器，然后使用 tail-f 之类的命令就可以实时地查看应用日志信息，并决定如何做出应对。  
  
但对于 SpringBoot 微服务来说，数量上的特征已经决定了单机单结点的方法已经行不通了，如果还是一台台地去查看应用日志，我们不但会“疲于奔命”，而且还无法及时有效地发现微服务作为一个逻辑服务集群整体上的状态特征现在是什么样的，我们这时候需要的是一种集中式的日志采集、存储和分析平台。  
  
对于 [Java](http://c.biancheng.net/java/) 开发者来说，ELK 技术栈正是为此而生的（E=ElasticSearch，L=Logstash，K=Kibana），整个功能链路如图 2 所示。

  
图 2  基本ELK技术栈功能链路示意图

不过，鉴于 ElasticSearch 对高频度的写入并没有很高的承受力，在正式的生产环境中，我们一般会采用如图 3 所示的部署结构。

  
图 3  生产环境典型 ELK 技术栈功能链路部署示意图

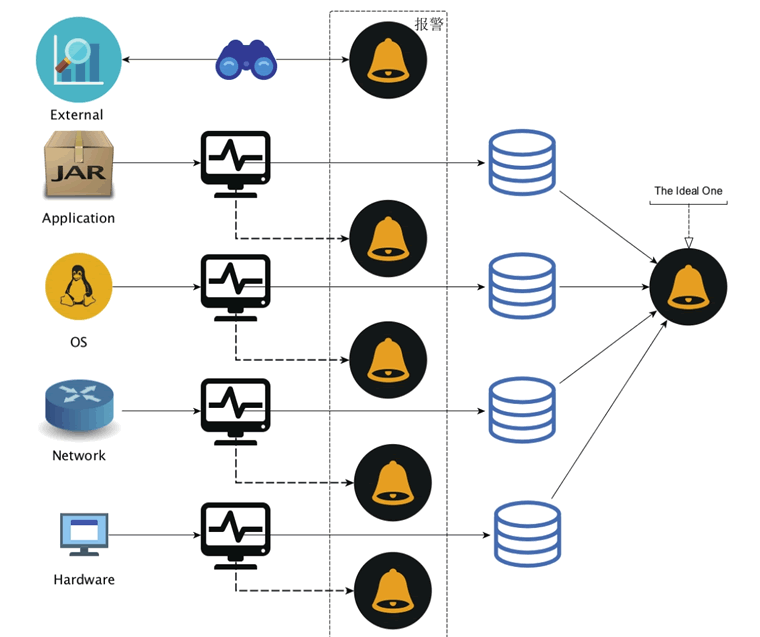
即我们使用 Kafka 作为数据采集的缓冲区，以便减轻从大量应用结点采集日志并写入 ElasticSearch 的负担。  
  
除了应用日志，应用内置的“传感器”数据以及 metrics 度量数据也都是比较重要的状态数据，对于 Java 程序来说（当然也包括 SpringBoot 微服务），JMX 是一项标准的监控手段。  
  
我们声称的应用内置“传感器”说白了就是一个个的 MBean。所以，我们要做的其实就是采集 SpringBoot 微服务的各种 MBean 的状态数据，然后分析或者报警。  
  
要将集群中各个 SpringBoot 微服务的 MBean 状态数据集中采集上来，我们当然可以采用 JMX 默认的 JSR160 规范实现远程访问，而且像 Zabbix 这样的监控系统还默认提供了对 JMX 的访问支持。  
  
但是鉴于 JMX 远程访问与防火墙之间的“纠葛”，以及 2015 年 Java 序列化漏洞暴露的不小危害，我更倾向于关闭 JMX 的远程访问，转而使用像 [Jolokia](https://jolokia.org/) 这样的方案。  
  
Jolokia 的好处在于，它可以无侵入的方式提供一种基于 HTTP 的 JMX MBean 访问通道。而且，它的 JMX 访问协议是统一的 JSON 格式，任何遵循 Jolokia JSON 协议规范的工具都可以对 MBean 的访问结果进行处理。  
  
对于 metrics 度量数据来说，也是一样的。我们建议使用 [dropwizard metrics](http://metrics.dropwizard.io/" \t "_blank) 来度量 SpringBoot 微服务的相应状态指标，然后通过 JMX+Jolokia 的方案统一采集并集中起来分析或者报警，从而避免多条处理链路上维护各自方案的资源投入和运维成本。  
  
不管是应用日志，还是内置“传感器”，甚至 metrics 度量状态数据，这些反应的都是我们对 SpringBoot 微服务内部明确剖析后的感知，即以“白盒”的形式对 SpringBoot 微服务进行监控。  
  
但是，从内部看没问题，不意味着从外部看也没有问题，毕竟，我们可以覆盖并设置很多内部指标，但永远无法通过覆盖所有的指标来反映 SpringBoot 微服务的运行状态。而且，SpringBoot 微服务并非孤立的存在，它的周边对其服务也有影响，比如网络是否通畅。  
  
所以，要对 SpringBoot 微服务进行完备的监控，我们不但要从内部以“白盒”的形式进行监控，还需要从外部服务访问者的视角来对 SpringBoot 微服务进行模拟访问（即“黑盒”形式），从而内外兼修地构建一套针对 SpringBoot 微服务的应用监控体系。  
  
实际上，SpringBoot 的 actuator 模块提供的健康检查（health check）功能就是一种允许以外部服务访问者的角度来监控微服务状态的现成方案。当然，更直接有效的方式则是直接发起对 SpringBoot 微服务的访问，从访问返回的结果来判断 SpringBoot 微服务整体上的运行状态是否良好。

### 推还是拉，这一直是个问题

我们发现，在各个监控层级各自的数据采集链路中，有些数据采集链路上采用数据客户端主动上报的方式（即 PUSH 方式），而有些则采用采集服务器端主动拉取的方式（即 PULL 方式）。  
  
比如，基于 ELK 技术栈的应用日志采集采用的是基于 logstash agent 的主动推送上报应用日志到数据采集服务的方式，dropwizard metrics 库的 GraphiteReporter 或者 GangliaReporter 也是同样的道理。  
  
而基于 JMX 或者 JMX+Jolokia 则是需要数据采集服务器端采用定时拉取（PULL）的方式采集状态或者 metrics 度量数据。至于 Zabbix，则是主动模式和被动模式都支持。  
  
那么，到底哪一种方式更好呢？  
  
实际上，可能这两种方式并没有好坏之分，只是两种不同的理念而已。只不过这两种理念散落到各个链路的不同地方，就会造成不小的困扰。好在，二者之间可以一定程度上适配，从而一定程度上减轻“脑裂”的痛苦。  
  
比如，虽然 JMX 或者 JMX+Jolokia 使用采集服务器端主动拉取状态数据的方式最自然，但我们也可以在 JMX 应用端设置一个 agent，由这个 agent 来本地拉取数据并上报，从而将拉取模式（PULL）转变为推送上报模式（PUSH），进而统一各个链路的数据采集模式。  
  
不过，在系统设计以及实现的时候，团队内部的理念最好统一，然后才能朝着一个方向使劲。其实，不管选择哪种模式，围绕这种模式构建系统和工具都能形成一套完备有效的方案，而一旦分散，则团队的技术生态体系中就会成长出实际上是完成同一目的，却要耗费双倍人力的方案。  
  
前面我们以监控系统中服务于“监”目的的系统为例说明了两种理念下产出的不同方案，而秉承同样的两种理念，在服务于“控”目的的系统设计上，也会出现两种系统样貌。  
  
比如，如果我们采用监控平台主动推送消息（PUSH）的方式对目标集群中的各个 SpringBoot 微服务进行行为干预，那么就需要我们的监控平台在主动推送消息这一功能上要有很强的并行执行能力，像 eBay 的 [parallec](https://github.com/eBay/parallec" \t "_blank) 系统就是遵循这样的理念设计并实现的。  
  
而如果我们采用 SpringBoot 微服务端自行拉取控制信息或者状态的方式来完成对 SpringBoot 微服务的控制行为，那么我们就会设计出像“配置中心”那样类似的共享状态的方案出来。  
  
所以，推还是拉，其实一直是一个问题，这个问题不在于谁优谁劣，而在于二者无法互相理解。

### 从局部性触发式报警到系统性智能化报警

要为 SpringBoot 微服务构建一套行之有效的监控体系，报警的有效性在其中至关重要。  
  
大部分情况下，监控体系中各个层次都各自为战，监控系统一旦接收到本层次关心的状态数据，并且发现超出了规定的阈值，就会即刻报警，如图 4 所示。

  
图 4  局部性触发式报警层次示意图

这种报警处理方式最大的问题就在于，它会生成很多的噪音：同一错误在多个 SpringBoot 微服务实例上报到监控系统之后，监控系统会无差别地悉数报警，而重复的报警往往会让运维或者应用负责人烦躁不已。  
  
实际上，一种有效的报警体系应该是能够系统化地综合考虑所有监控层级的数据，然后汇总和分析，最终发出一个行之有效的报警。  
  
我们称前一种为局部性触发式报警，而后一种叫系统性智能化报警。显然，系统性智能化报警应该是我们追求的最高目标。只有在条件有限的情况下，我们才会适当地容忍局部性触发式报警。

## 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