# Spring Boot 入门

Spring Boot是Spring社区较新的一个项目。该项目的目的是帮助开发者更容易的创建基于Spring的应用程序和服务，让更多人的人更快的对Spring进行入门体验，让Java开发也能够实现Ruby on Rails那样的生产效率。并为Spring生态系统提供了一种固定的、约定优于配置风格的框架。

Spring Boot具有如下特性：

* 为基于Spring的开发提供更快的入门体验
* 开箱即用，没有代码生成，也无需XML配置。同时也可以修改默认值来满足特定的需求。
* 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。
* Spring Boot并不是对Spring功能上的增强，而是提供了一种快速使用Spring的方式。
* SpringBoot简化了使用Spring的过程。其更多的是采用Java Config的方式对Spring进行配置，从而使原来的多个配置文件（.properties、.xml）减少为一个配置文件（application.yml或者application.properties）

## 1.1 简单例子

首先创建一个一般的Maven项目，有一个pom.xml和基本的src/main/java结构。

### 1.1.1 pom.xml 文件

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>wsh.springboot.sample</groupId>  <artifactId>spring-boot-sample-start</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.3.5.RELEASE</version>  </parent>    <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.7</java.version>  </properties>    <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>    <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

### 1.1.2 对pom的说明

首先是增加了<parent>，增加父pom比较简单，而且spring-boot-starter-parent包含了大量配置好的依赖管理，在自己项目添加这些依赖的时候不需要写<version>版本号。

使用父pom虽然简单，但是有些情况我们已经有父pom，不能直接增加<parent>时，可以通过如下方式：

|  |
| --- |
| <dependencyManagement>  <dependencies>  <dependency>  <!-- Import dependency management from Spring Boot -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.2.3.RELEASE</version>  <type>pom</type>  <scope>import</scope><!—这个地方-->  </dependency>  </dependencies>  </dependencyManagement> |

### 1.1.3 添加spring-boot-starter-web依赖

Spring通过添加spring-boot-starter-\*这样的依赖就能支持具体的某个功能。

我们这个示例最终是要实现web功能，所以添加的是这个依赖。

更完整的功能列表可以查看：[Using-boot-starter-poms](http://docs.spring.io/spring-boot/docs/1.2.3.RELEASE/reference/html/using-boot-build-systems.html#using-boot-starter-poms)

(TODO:对文档的完善会从列表中选择需要的依赖项进行实际测试)

除了应用程序的starters，下面的starters可以用于添加生产准备的特性。

|  |  |
| --- | --- |
| 名称 | 描述 |
| spring-boot-starter-actuator | 添加生产准备特性，比如指标和监控 |
| spring-boot-starter-remote-shell | 添加远程 ssh shell支持 |

最后，Spring Boot包含一些可用于排除或交换具体技术方面的starters

|  |  |
| --- | --- |
| 名称 | 描述 |
| spring-boot-starter-jetty | 导入Jetty HTTP引擎（作为Tomcat的替代） |
| spring-boot-starter-log4j | 对Log4J日志系统的支持 |
| spring-boot-starter-logging | 导入Spring Boot的默认日志系统（Logback） |
| spring-boot-starter-tomcat | 导入Spring Boot的默认HTTP引擎（Tomcat） |
| spring-boot-starter-undertow | 导入Undertow HTTP引擎（作为Tomcat的替代） |

### 添加spring-boot-maven-plugin插件

该插件支持多种功能，常用的有两种，第一种是打包项目为可执行的jar包。

在项目根目录下执行mvn package将会生成一个可执行的jar包，jar包中包含了所有依赖的jar包，只需要这一个jar包就可以运行程序，使用起来很方便。该命令执行后还会保留一个XXX.jar.original的jar包，包含了项目中单独的部分。

生成这个可执行的jar包后，在命令行执行java -jar xxxx.jar即可启动项目。

另外一个命令就是mvn spring-boot:run，可以直接使用tomcat（默认）启动项目。

在我们开发过程中，我们需要经常修改，为了避免重复启动项目，我们可以启用热部署。

### 1.1.6 spring-loaded热部署

Spring-Loaded项目提供了强大的热部署功能，添加/删除/修改 方法/字段/接口/枚举 等代码的时候都可以热部署，速度很快，很方便。

想在Spring Boot中使用该功能非常简单，**就是在spring-boot-maven-plugin**插件下面添加依赖：

|  |
| --- |
| <!-- 支持热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  <version>1.2.5.RELEASE</version>  </dependency> |

添加以后，通过mvn spring-boot:run启动就支持热部署了。（经过测试，这里如果使用main方法启动，将达不到热部署的效果）

注意：使用热部署的时候，需要IDE编译类后才能生效，你可以打开自动编译功能，这样在你保存修改的时候，类就自动重新加载了。

## 创建一个应用类

我们创建一个HelloController类：

|  |
| --- |
| @RestController  //@RestController==@Controller+@ResponseBody  /\*@Controller  @ResponseBody\*/  @RequestMapping("/hello")  **public** **class** HelloController {    @RequestMapping  **public** String hello(){  **return** "Hello Spring Boot 23231312323";  }    @RequestMapping("/info")  **public** Map<String, String> getInfoMap(@RequestParam String name){  Map<String, String> map = **new** HashMap<String, String>();  map.put("name", name);  **return** map;  }    @RequestMapping("/list")  **public** List<Map<String, String>> getList(){  List<Map<String, String>> list = **new** ArrayList<Map<String,String>>();  Map<String, String> map = **null**;  **for**(**int** i=0;i<5;i++){  map = **new** HashMap<String, String>();  map.put("name", "wsh"+i);  list.add(map);  }  **return** list;  } |

@RestController和@RequestMapping注解

我们的Application类上使用的第一个注解是 @RestController 。这被称为一个构造型

（stereotype）注解。对于Spring，该类扮演了一个特殊角色。等同于@ResponsBody和@Controller的集合

@RequestMapping 注解提供路由信息。它告诉Spring任何来自"/hello"路径的HTTP请求都应该被映射到 hello方法。 @RestController 注解告诉Spring以字符串的形式渲染结果，并直接返回给调用者。

创建一个SpringBootSampleApplication类

|  |
| --- |
| @SpringBootApplication  **public** **class** SpringBootSampleApplication {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  SpringApplication.*run*(SpringBootSampleApplication.**class**, args);  }  } |

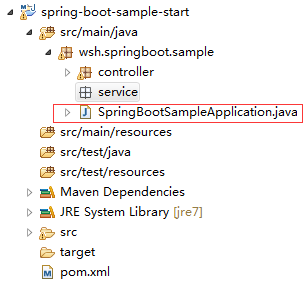
@SpringBootApplication注解

申明让spring boot自动给程序进行必要的配置，这个配置等同于：

@Configuration ，@EnableAutoConfiguration 和 @ComponentScan 三个配置。

### 1.2.1 注意

Spring Boot建议将我们main方法所在的这个主要的配置类配置在根包名下。



在SpringBootSampleApplication.java中有main方法。

这只是一个标准的方法，它遵循Java对于一个应用程序入口点的约定。我们的main方法通过调用run，将业务委托给了Spring Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat web服务器。我们需要将 SpringBootSampleApplication.class 作为参数传递给run方法来告诉SpringApplication谁是主要的Spring组件。为了暴露任何的命令行参数，args数组也会被传递过去。

### 1.2.5 启动项目SpringApplication.run

启动Spring Boot项目最简单的方法就是执行下面的方法：

SpringApplication.run(SpringBootSampleApplication.class, args);

该方法返回一个ApplicationContext对象，使用注解的时候返回的具体类型是AnnotationConfigApplicationContext或AnnotationConfigEmbeddedWebApplicationContext，当支持web的时候是第二个。

除了上面这种方法外，还可以用下面的方法：

SpringApplication application = new SpringApplication(Application.class);

application.run(args);

SpringApplication包含了一些其他可以配置的方法，如果你想做一些配置，可以用这种方式。

除了上面这种直接的方法外，还可以使用SpringApplicationBuilder：

new SpringApplicationBuilder()

.showBanner(false)

.sources(Application.class)

.run(args);

当使用SpringMVC的时候由于需要使用子容器，就需要用到SpringApplicationBuilder，该类有一个child(xxx...)方法可以添加子容器。

## 1.3 运行

在IDE中直接直接执行main方法，然后访问http://localhost:8080即可。

另外还可以用上面提到的mvn，可以打包为可执行jar包，然后执行java -jar xxx.jar。

或者执行mvn spring-boot:run运行项目。

# Spring Boot 常用注解

#### 注解(annotations)列表

1. @ResponseBody   
   用该注解修饰的函数，会将结果直接填充到HTTP的响应体中，一般用于构建RESTful的api；
2. @Controller   
   用于定义控制器类，在spring 项目中由控制器负责将用户发来的URL请求转发到对应的服务接口（service层）。
3. @RestController   
   @ResponseBody和@Controller的合集
4. @RequestMapping   
   提供路由信息，负责URL到Controller中的具体函数的映射。
5. @EnableAutoConfiguration   
   Spring Boot自动配置（auto-configuration）：尝试根据你添加的jar依赖自动配置你的Spring应用。这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。比如上面的例子中，由于 spring-boot-starter-web 添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。Starter POMs和Auto-Configuration：设计auto-configuration的目的是更好的使用"Starter POMs"，但这两个概念没有直接的联系。你可以自由地挑选starter POMs以外的jar依赖，并且Spring Boot将仍旧尽最大努力去自动配置你的应用。
6. @ComponentScan   
   表示将该类自动发现（扫描）并注册为Bean，可以自动收集所有的Spring组件，包括@Configuration类。我们经常使用@ComponentScan注解搜索beans，并结合@Autowired注解导入。
7. @Configuration   
   相当于传统的xml配置文件，如果有些第三方库需要用到xml文件，建议仍然通过@Configuration类作为项目的配置主类——可以使用@ImportResource注解加载xml配置文件。
8. @SpringBootApplication   
   相当于@EnableAutoConfiguration、@ComponentScan和@Configuration的合集。
9. @Import   
   用来导入其他配置类。
10. @ImportResource   
    用来加载xml配置文件。
11. @Autowired   
    自动导入依赖的bean
12. @Service   
    一般用于修饰service层的组件
13. @Repository   
    使用@Repository注解可以确保DAO或者repositories提供异常转译，这个注解修饰的DAO或者repositories类会被ComponetScan发现并配置，同时也不需要为它们提供XML配置项。

# [Spring Boot 属性配置和使用](http://blog.csdn.net/isea533/article/details/50281151)

Spring Boot允许外化你的配置，这样就能够在不同的环境下使用相同的代码。你可以使用properties文件、YAML文件，也可以使用环境变量和命令行参数来进行外化配置。

## Spring Boot 支持多种外部配置方式

Spring Boot支持多种方式的外部配置，其使用一个非常特别的PropertySource次序来允许对值进行合理的覆盖，具体次序如下：

1. 命令行参数

2. 来自java:comp/env的JNDI属性

3. Java系统属性（System.getProperties()）

4. 操作系统环境变量

5. RandomValuePropertySource配置的random.\*属性值

6. jar包外部的application-{profile}.properties或application.yml(带spring.profile)配置文件

7. jar包内部的application-{profile}.properties或application.yml(带spring.profile)配置文件

8. jar包外部的application.properties或application.yml(不带spring.profile)配置文件

9. jar包内部的application.properties或application.yml(不带spring.profile)配置文件

10. @Configuration注解类上的@PropertySource

11. 通过SpringApplication.setDefaultProperties指定的默认属性

打个比方：

我们的程序有一个name属性，你可以将一个application.properties文件捆绑到jar内，用来提供一个合理的默认name属性值。当运行在生产环境时，可以在jar外提供一个application.properties文件来覆盖name属性。对于一次性的测试，你可以使用特定的命令行开关启动（比如，java -jar app.jar --name="Spring"）。

### 3.1.1 命令行参数

通过java -jar app.jar --server.port=9090方式来传递参数。

参数用--xxx=xxx的形式传递。

可以使用的参数可以是我们自己定义的，也可以是Spring Boot中默认的参数。

很多人可能会关心如web端口如何配置这样的问题，这些都是Spring Boot中提供的参数，部分可用参数如下：

|  |
| --- |
| # EMBEDDED SERVER CONFIGURATION (ServerProperties)  server.port=8080  server.address= # bind to a specific NIC  server.session-timeout= # session timeout in seconds  server.context-parameters.\*= # Servlet context init parameters, e.g. server.context-parameters.a=alpha  server.context-path= # the context path, defaults to '/'  server.servlet-path= # the servlet path, defaults to '/' |

更多常见的应用属性请浏览[这里](http://docs.spring.io/spring-boot/docs/1.2.3.RELEASE/reference/html/common-application-properties.html)

**注意：**命令行参数在app.jar的后面！

可以通过SpringApplication.setAddCommandLineProperties(false)禁用命令行配置。

### 3.1.2 Java系统属性

注意Java系统属性位置java -Dname="isea533" -jar app.jar，可以配置的属性都是一样的，优先级不同。

例如java -Dname="isea533" -jar app.jar --name="Spring!"中name值为Spring!

### 3.1.3 操作系统环境变量

配置过JAVA\_HOME的应该都了解这一个。

这里需要注意的地方，有些OS可以不支持使用.这种名字，如server.port，这种情况可以使用SERVER\_PORT来配置。

### 3.1.4 RandomValuePropertySource

RandomValuePropertySource在注入随机值（比如，密钥或测试用例）时很有用。它能产生

整数，longs或字符串，例如：

my.secret=${random.value}

my.number=${random.int}

my.bignumber=${random.long}

my.number.less.than.ten=${random.int(10)}

my.number.in.range=${random.int[1024,65536]}

random.int\*支持value参数和,max参数，当提供max参数的时候，value就是最小值。

### 3.1.5 应用配置文件（.properties或.yml）

在配置文件中直接写：

name=Isea533

server.port=8080

.yml格式的配置文件如：

name: Isea533

server:

port: 8080

当有前缀的情况下，使用.yml格式的配置文件更简单。关于.yml配置文件用法请看[这里](http://docs.spring.io/spring-boot/docs/1.2.3.RELEASE/reference/html/boot-features-external-config.html#boot-features-external-config-yaml)

注意：使用.yml时，属性名的值和冒号中间必须有空格，如name: Isea533正确，name:Isea533就是错的。

#### 3.1.5.1 属性配置文件的位置

SpringApplication将从以下位置加载application.properties文件，并把该文件添加到Spring Environment中：

1、当前目录下的一个/config子目录

2、当前目录

3、一个classpath下的/config包

4、classpath根路径

这个列表是按优先级排序的，也就是列表中位置高的配置将覆盖位置低的配置文件。

### 3.1.6 @PropertySource

这个注解可以指定具体的属性配置文件，优先级比较低。

### 3.1.7 SpringApplication.setDefaultProperties

例如：

SpringApplication application = new SpringApplication(Application.class);

Map<String, Object> defaultMap = new HashMap<String, Object>();

defaultMap.put("name", "Isea-Blog");

//还可以是Properties对象

application.setDefaultProperties(defaultMap);

application.run(args);

## 3.1.8 多环境配置

我们在开发Spring Boot应用时，通常同一套程序会被应用和安装到几个不同的环境，比如：开发、测试、生产等。其中每个环境的数据库地址、服务器端口等等配置都会不同，如果在为不同环境打包时都要频繁修改配置文件的话，那必将是个非常繁琐且容易发生错误的事。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot也不例外，或者说更加简单。

在Spring Boot中多环境配置文件名需要满足application-{profile}.properties的格式，其中{profile}对应你的环境标识，比如：

   application-dev.properties：开发环境

   application-test.properties：测试环境

   application-prod.properties：生产环境

至于哪个具体的配置文件会被加载，需要在application.properties文件中通过spring.profiles.active属性来设置，其值对应{profile}值。

如：spring.profiles.active=test就会加载application-test.properties配置文件内容

下面，以不同环境配置不同的服务端口为例，进行样例实验。

针对各环境新建不同的配置文件application-dev.properties、application-test.properties、application-prod.properties

在这三个文件均都设置不同的server.port属性，如：dev环境设置为8080，test环境设置为9090，prod环境设置为80

application.properties中设置spring.profiles.active=dev，就是说默认以dev环境设置

测试不同配置的加载：

       执行java -jar xxx.jar，可以观察到服务端口被设置为8080，也就是默认的开发环境（dev）

       执行java -jar xxx.jar --spring.profiles.active=test，可以观察到服务端口被设置为9090，也就是测试环境的配置（test）

       执行java -jar xxx.jar --spring.profiles.active=prod，可以观察到服务端口被设置为80，也就是生产环境的配置（prod）

按照上面的实验，可以如下总结多环境的配置思路：

   application.properties中配置通用内容，并设置spring.profiles.active=dev，以开发环境为默认配置

   application-{profile}.properties中配置各个环境不同的内容

通过命令行方式去激活不同环境的配置。

## 应用（使用）属性

### 3.2.1 @Value(“${xxx}”)

这种方式是最简单的，通过@Value注解可以将属性值注入进来。

### 3.2.2 @ConfigurationProperties

Spring Boot 可以方便的将属性注入到一个配置对象中。例如：

my.name=Isea533

my.port=8080

my.servers[0]=dev.bar.com

my.servers[1]=foo.bar.com

对应对象：

|  |
| --- |
| @ConfigurationProperties(prefix="my")  public class Config {  private String name;  private Integer port;  private List<String> servers = new ArrayList<String>();  public String geName(){  return this.name;  }  public Integer gePort(){  return this.port;  }  public List<String> getServers() {  return this.servers;  }  } |

TODO:这里没有测试成功，需要分析原因

Spring Boot 会自动将prefix="my"前缀为my的属性注入进来。

Spring Boot 会自动转换类型，当使用List的时候需要注意在配置中对List进行初始化！

Spring Boot 还支持嵌套属性注入，例如：

name=isea533

jdbc.username=root

jdbc.password=root

...

对应的配置类：

|  |
| --- |
| @ConfigurationProperties  public class Config {  private String name;  private Jdbc jdbc;  class Jdbc {  private String username;  private String password;  //getter...  }  public Integer gePort(){  return this.port;  }  public Jdbc getJdbc() {  return this.jdbc;  }  } |

jdbc开头的属性都会注入到Jdbc对象中。

### 3.2.3 在@Bean方法上使用@ConfigurationProperties

例如：

@ConfigurationProperties(prefix = "foo")

@Bean

public FooComponent fooComponent() {

...

}

Spring Boot 会将foo开头的属性按照名字匹配注入到FooComponent对象中。

### 3.2.4 属性占位符

例如：

app.name=MyApp

app.description=${app.name} is a Spring Boot application

可以在配置文件中引用**前面**配置过的属性（优先级前面配置过的这里都能用）。

通过如${app.name:默认名称}方法还可以设置默认值，当找不到引用的属性时，会使用默认的属性。

由于${}方式会被Maven处理。如果你pom继承的spring-boot-starter-parent，

Spring Boot 已经将maven-resources-plugins默认的${}方式改为了@ @方式，例如@name@。

如果你是引入的Spring Boot，你可以修改使用[其他的分隔符](http://maven.apache.org/plugins/maven-resources-plugin/resources-mojo.html#delimiters)

### 3.2.5 通过属性占位符还能缩短命令参数

例如修改web默认端口需要使用--server.port=9090方式，如果在配置中写上：

server.port=${port:8080}

那么就可以使用更短的--port=9090，当不提供该参数的时候使用默认值8080。

### 3.2.6 属性名匹配规则

例如有如下配置对象：

@Component

@ConfigurationProperties(prefix="person")

public class ConnectionSettings {

private String firstName;

}

firstName可以使用的属性名如下：

person.firstName，标准的驼峰式命名

person.first-name，虚线（-）分割方式，推荐在.properties和.yml配置文件中使用

PERSON\_FIRST\_NAME，大写下划线形式，建议在系统环境变量中使用

### 3.2.7 属性验证

可以使用JSR-303注解进行验证，例如：

@Component

@ConfigurationProperties(prefix="connection")

public class ConnectionSettings {

@NotNull

private InetAddress remoteAddress;

// ... getters and setters

}

## ****3.3 最后****

以上是Spring Boot 属性配置和使用的内容，更全的讲解可以查看[Spring Boot完整文档](http://docs.spring.io/spring-boot/docs/1.2.3.RELEASE/reference/html/index.html) 或 [Externalized Configuration](http://docs.spring.io/spring-boot/docs/1.2.3.RELEASE/reference/html/boot-features-external-config.html)。

# Spring Boot 安全访问控制

1. 在pom文件中，引入spring-boot-starter-security的maven依赖。
2. **<dependency>**
3. **<groupId>**org.springframework.boot**</groupId>**
4. **<artifactId>**spring-boot-starter-security**</artifactId>**
5. **</dependency>**
6. 在application.properties中，配置访问的用户名和密码
7. #用户名 密码配置
8. security.user.name=admin
9. security.user.password=test

当然这只是，一个初级的配置，更复杂的配置，可以分不用角色，在控制范围上，能够拦截到方法级别的权限控制。

# Spring Boot 日志

在开发中打印内容，使用 System.out.println() 和 Log4j 应当是人人皆知的方法了。

其实在开发中我们不建议使用 System.out 因为大量的使用 System.out 会增加资源的消耗。

而Log4j 更为灵活在性能上也相比 System.out 要高，我们可以配置输出级别，可以指定多个日志文件分别记录不同的日志。

使用 System.out 是在当前线程执行的，写入文件也是写入完毕后才继续执行下面的程序。而使用Log工具不但可以控制日志是否输出，怎么输出，它的处理机制也是通知写日志，继续执行后面的代码不必等日志写完。

如非必要，建议大家不要使用控制台输出，因为控制台输出没有优先级会显得输出太乱。

Spring Boot 提供了一套日志系统，logback是最优先的选择。

在Spring Boot 中记录日志只需两步：

1、在 src/main/resources 下面创建logback.xml （根据不同环境来定义不同的日志输出，那么取名为logback-spring.xml 即可）文件，并按上面讲述的进行配置。

2、在Java代码中创建实例，并在需要输出日志的地方使用。

Logback-spring.xml配置文件如下：

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

<configuration>

<include resource=*"org/springframework/boot/logging/logback/base.xml"* />

<logger name=*"org.springframework.web"* level=*"INFO"*/>

<logger name=*"wsh.springboot.sample"* level=*"TRACE"* />

<springProfile name=*"dev"*>

<logger name=*"wsh.springboot.sample"* level=*"DEBUG"* />

</springProfile>

<springProfile name=*"test"*>

<logger name=*"wsh.springboot.sample"* level=*"INFO"* />

</springProfile>

</configuration>