# SpringBoot\_study

**By Brother.Lee 201712 Shanghai**

## Springboot特点:

1:为基于spring的开发提供更快的入门体验

2:创建可以 独立运行的spring的应用

3:直接嵌入Tomcat或Jetty服务器,不需要打包成WAR文件

4:提供推荐的基础POM文件(starter)来简化Apache Maven配置

5:尽可能的根据项目依赖来自动配置spring框架

6:提供可以直接在生产环境中使用的功能,如性能指标,应用信息和应用健康检查

7:开箱即用,没有代码生成,也无须xml配置,同时也可以修改默认值来满足特定的需求

8:其他大量的项目都是基于spring boot之上的,如spring cloud.

总结起来:

1:spring boot使编码变得简单

2:spring boot使配置变得简单

3:spring boot使部署变得简单

4:spring boot使监控变得简单

## Spring boot的缺点

1:依赖太多,随便一个spring boot应用都好几十M

2:缺少服务的注册和发现等解决方案

3:缺少监控集成方案,安全管理方案

4:中文的文档和资料太少且不够深入

## Spring boot的应用场景:

1:spring能够应用的场景

2:Java web应用

3:微服务

## 课程大纲:

1:spring Boot概述与课程概要介绍

2:spring4快速入门

3:spring4扩展分析

4:Spring Boot快速入门

5:Spring Boot配置分析

6:Spring Boot自动配置

7:Spring Boot @Enable\*注解的工作原理

8:Spring Boot @EnableAutoConfiguration深入分析

9:Spring Boot事件监听

10:Spring Boot扩展分析

11:Spring Boot运行流程分析

12:Spring Boot Web

13:Spring Boot订制和优化内嵌的Tomcat

14:Spring Boot JDBC

15:Spring Boot AOP

16:Spring Boot Starter

17:Spring Boot日志

18:Spring Boot监控和度量

19:Spring Boot测试

20:Spring Boot构建微服务实战

21:Spring Boot服务的注册和发现

22:Spring Boot应用的打包和部署

# SpringBoot helloworld

pom.xml

|  |
| --- |
| <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>com.springboot</groupId>  <artifactId>com.springboot.startdemo.controller</artifactId>  <version>0.0.1-SNAPSHOT</version>    <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.9.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

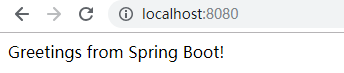
启动类MainApplication

|  |
| --- |
| **package** com.springboot.startdemo.controller;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  @EnableAutoConfiguration  **public** **class** MainApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(MainApplication.**class**, args);  }  } |

web的controller

|  |
| --- |
| package com.springboot.startdemo.controller;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  @RestController  public class HelloController {  @RequestMapping("/")  public String index() {  return "Greetings from Spring Boot!";  }  } |

启动后访问web



springboot maven插件,可以将项目打成一个jar包

|  |
| --- |
| <!‐‐ 这个插件，可以将应用打包成一个可执行的jar包；‐‐>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

# SpringBoot配置文件

### 0.1:配置文件

springboot的配置文件是一个全局的配置文件:

\*application.properties

\*application.yml

### 0.2:yml语法:

k:(空格)v：表示一对键值对（空格必须有）；

以空格的缩进来控制层级关系；只要是左对齐的一列数据，都是同一个层级的

server:

    port: 8081

    path: /hello

### 0.3:值的写法

字面量：普通的值（数字，字符串，布尔）

k: v：字面直接来写；

字符串默认不用加上单引号或者双引号；

""：双引号；不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思

name: "zhangsan \n lisi"：输出；zhangsan 换行 lisi

''：单引号；会转义特殊字符，特殊字符最终只是一个普通的字符串数据

name: ‘zhangsan \n lisi’：输出；zhangsan \n lisi

对象、Map（属性和值）（键值对）：

k: v：在下一行来写对象的属性和值的关系；注意缩进

对象还是k: v的方式

friends:

lastName: zhangsan

age: 20

行内写法

friends: {lastName: zhangsan,age: 18}

数组（List、Set）：

用- 值表示数组中的一个元素

pets:

 ‐ cat

 ‐ dog

 ‐ pig

3、配置文件值注入

配置文件

javaBean：

我们可以导入配置文件处理器，以后编写配置就有提示了

pets: [cat,dog,pig]

## 1:配置文件值注入

我们可以导入配置文件处理器，以后编写配置就有提示了

<!-- 导入配置文件处理器，配置文件进行绑定就会有提示 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring‐boot‐configuration‐processor</artifactId>

<optional>true</optional>

</dependency>

### @ConfigurationProperties(prefix = "XXX")

告诉springboot将本类中的所有属性和配置文件中相关的配置进行绑定;

### @PropertySource&@ImportResource&@Bean

@PropertySource：加载指定的配置文件；

@PropertySource(value = {"classpath:person.properties"})

@ImportResource：导入Spring的配置文件，让配置文件里面的内容生效；

Spring Boot里面没有Spring的配置文件，我们自己编写的配置文件，也不能自动识别；

想让Spring的配置文件生效，加载进来；@ImportResource标注在一个配置类上

@ImportResource(locations = {"classpath:beans.xml"})

导入Spring的配置文件让其生效

## 2:自测加载形式

#### application.yml

如果将配置写在application.yml中,则不要加载指定的配置文件了,系统会自动将其内容进行加载到内存中

application.yml

|  |
| --- |
| server:  port: 8082  person:  lastName: Jim  age: 19  boos: false  birth: 2011/12/11  maps: {k1: v1 , k2: 12}  list:  - lisi  - wangwu  dog:  name: dahuang  age: 2 |

|  |
| --- |
| package com.springboot.startdemo;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringRunner;  import com.springboot.startdemo.bean.Person;  @RunWith(SpringRunner.class)  @SpringBootTest  public class MainApplicationTest {  @Autowired  Person person;  @Test  public void contectTest() {  System.out.println(person);  }  } |

结果

Person [lastName=Jim, age=19, boss=null, birth=Sun Dec 11 00:00:00 CST 2011, maps={k1=v1, k2=12}, lists=null, dog=com.springboot.startdemo.bean.Dog@13e698c7]

#### person.yml

如果使用自定义的.yml方式加载配置文件.

@Component

@ConfigurationProperties(prefix = "person")

@PropertySource(value = "classpath:person.yml")

**public** **class** Person {

并不能加载到配置

Person{name='null', age=null, isBoss=false, birth=null, maps=null, lists=null, dog=null}

#### person.properties

# 配置person

# idea 默认UTF-8

person.name=张三

person.age=18

person.birth=2018/02/11

person.isBoss=true

person.maps.k1=v1

person.maps.k2=v2

person.lists=1,2,3

person.dog.name=dog

person.dog.age=15

person.last-name=王五

居然可以加载了--!

Person [lastName=çäº, age=18, boss=null, birth=Sun Feb 11 00:00:00 CST 2018, maps={k1=v1, k2=v2}, lists=[1, 2, 3], dog=com.springboot.startdemo.bean.Dog@5e76a2bb]

#### 总结

自定义配置文件目前只支持properties方式

## 3:SpringBoot的application.yml application.properties 区别和优先级

application.properties和application.yml文件可以放在一下四个位置：

外置，在相对于应用程序运行目录的/congfig子目录里。

外置，在应用程序运行的目录里

内置，在config包内

内置，在Classpath根目录

同样，这个列表按照优先级排序，也就是说，src/main/resources/config下application.properties覆盖src/main/resources下application.properties中相同的属性

此外，如果你在相同优先级位置同时有application.properties和application.yml，那么application.yml里面的属性就会覆盖application.properties里的属性。

**也就是说哪个文件被最后加载，哪个才具有最高级别，因为最后的，会覆盖前面所有的。**

## 4:配置文件占位符

### 随机数

${random.value}、

${random.int}、

${random.long}

${random.int(10)}、

${random.int[1024,65536]}

### 占位符获取之前配置的值，如果没有可以是用:指定默认值

person.last‐name=张三${random.uuid}

person.age=${random.int}

person.birth=2017/12/15

person.boss=false

person.maps.k1=v1

person.maps.k2=14

person.lists=a,b,c

person.dog.name=${person.hello:hello}\_dog

person.dog.age=15

## 5:Profile

### 5.1、多Profile文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml

默认使用application.properties的配置；

### 5.2、yml支持多文档块方式

server:

  port: 8081

spring:

  profiles:

    active: prod

‐‐‐

server:

  port: 8083

spring:

  profiles: dev

‐‐‐

server:

  port: 8084

spring:

  profiles: prod  #指定属于哪个环境

### 5.3、激活指定profile

1、在配置文件中指定 spring.profiles.active=dev

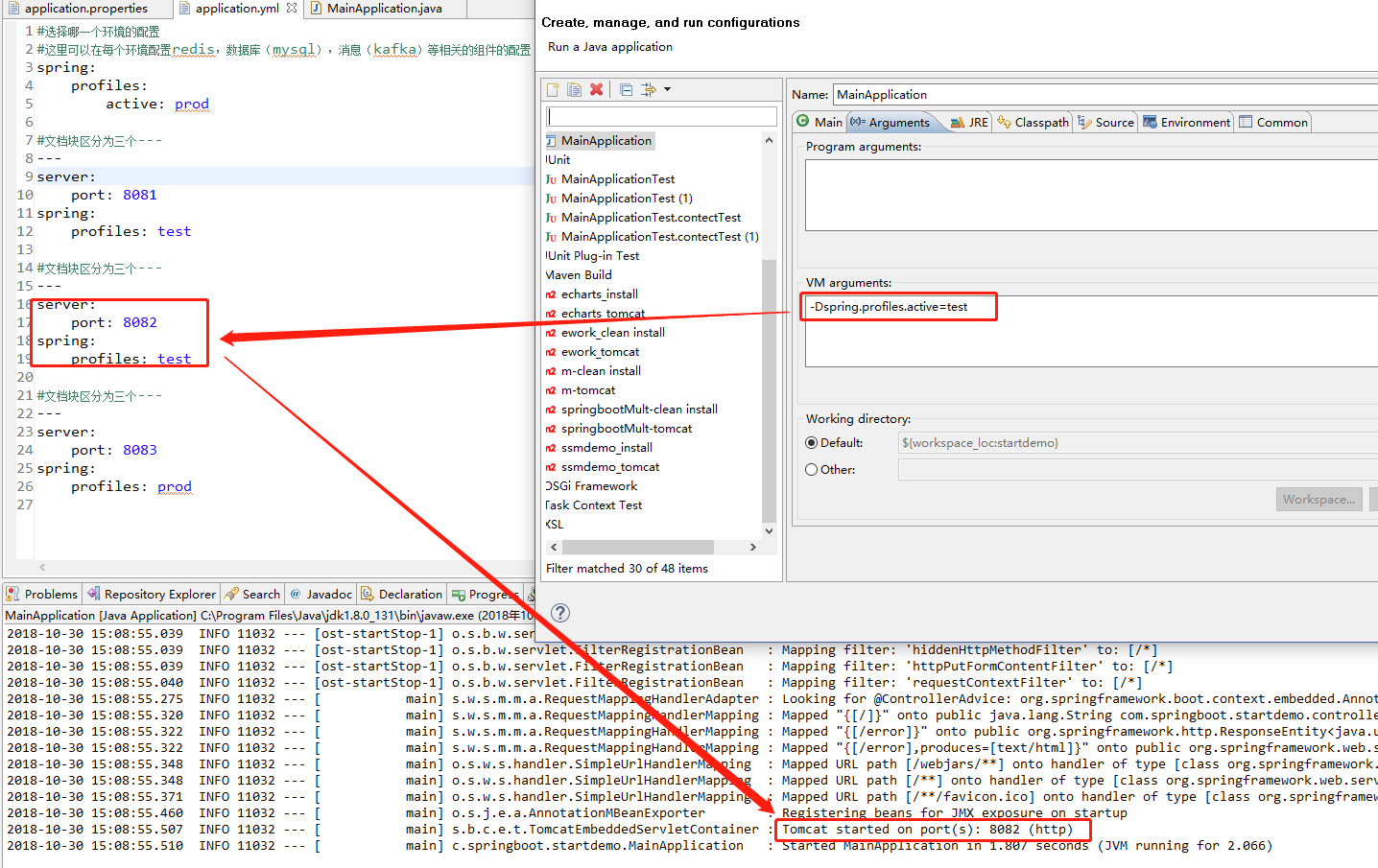
2、命令行：

java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev；

可以直接在测试的时候，配置传入命令行参数

3、虚拟机参数；

-Dspring.profiles.active=dev



|  |
| --- |
| #选择哪一个环境的配置  #这里可以在每个环境配置redis，数据库（mysql），消息（kafka）等相关的组件的配置  spring:  profiles:  active: prod  #文档块区分为三个---  ---  server:  port: 8081  spring:  profiles: test  #文档块区分为三个---  ---  server:  port: 8082  spring:  profiles: test  #文档块区分为三个---  ---  server:  port: 8083  spring:  profiles: prod |

如果虚拟机不加参数就是prod,端口监听8083

如果虚拟机加了参数,加哪一个就监听对应的那个端口

## 6:配置文件加载位置

springboot 启动会扫描以下位置的application.properties或者application.yml文件作为Spring boot的默认配置文件

–file:./config/

–file:./

–classpath:/config/

–classpath:/

优先级由高到底，高优先级的配置会覆盖低优先级的配置；

SpringBoot会从这四个位置全部加载主配置文件；互补配置；

我们还可以通过spring.config.location来改变默认的配置文件位置

项目打包好以后，我们可以使用命令行参数的形式，启动项目的时候来指定配置文件的新位置；指定配置文件和默认加载的这些配置文件共同起作用形成互补配置；

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --spring.config.location=G:/application.properties

## 7:外部配置加载顺序

SpringBoot也可以从以下位置加载配置； 优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置

1.命令行参数

所有的配置都可以在命令行上进行指定

java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar --server.port=8087 --server.context-path=/abc

多个配置用空格分开； --配置项=值

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

由jar包外向jar包内进行寻找；

优先加载带profile

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)配置文件再来加载不带profile

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

所有支持的配置加载来源；

<https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#boot-features-external-config>

## 8:自动配置原理

配置文件到底能写什么？怎么写？自动配置原理；

<https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#common-application-properties>

### 1、自动配置原理：

1）、SpringBoot启动的时候加载主配置类，开启了自动配置功能 @EnableAutoConfiguration

2）、@EnableAutoConfiguration 作用：

* 利用EnableAutoConfigurationImportSelector给容器中导入一些组件？
* 可以查看selectImports()方法的内容；
* List configurations = getCandidateConfigurations(annotationMetadata, attributes);获取候选的配置

SpringFactoriesLoader.loadFactoryNames()

扫描所有jar包类路径下  META‐INF/spring.factories

把扫描到的这些文件的内容包装成properties对象

从properties中获取到EnableAutoConfiguration.class类（类名）对应的值，然后把他们添加在容器中

将类路径下 META-INF/spring.factories 里面配置的所有EnableAutoConfiguration的值加入到了容器中；

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\

org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\

org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\

org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\

org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\

org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,

\

org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\

org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration

,\

org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\

org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration

,\

org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAutoConfiguration,\

org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration

每一个这样的 xxxAutoConfiguration类都是容器中的一个组件，都加入到容器中；用他们来做自动配置；

3）、每一个自动配置类进行自动配置功能；

4）、以HttpEncodingAutoConfiguration（Http编码自动配置）为例解释自动配置原理；

|  |
| --- |
| @Configuration   //表示这是一个配置类，以前编写的配置文件一样，也可以给容器中添加组件  @EnableConfigurationProperties(HttpEncodingProperties.**class**)  //启动指定类的ConfigurationProperties功能；将配置文件中对应的值和HttpEncodingProperties绑定起来；并把HttpEncodingProperties加入到ioc容器中  @ConditionalOnWebApplication //Spring底层@Conditional注解（Spring注解版），根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会生效；    判断当前应用是否是web应用，如果是，当前配置类生效  @ConditionalOnClass(CharacterEncodingFilter.**class**)  //判断当前项目有没有这个类CharacterEncodingFilter；SpringMVC中进行乱码解决的过滤器；  @ConditionalOnProperty(prefix = "spring.http.encoding", value = "enabled", matchIfMissing **true**)  //判断配置文件中是否存在某个配置  spring.http.encoding.enabled；如果不存在，判断也是成立的  //即使我们配置文件中不配置pring.http.encoding.enabled=true，也是默认生效的；  **public** **class** HttpEncodingAutoConfiguration {    //他已经和SpringBoot的配置文件映射了  **private** **final** HttpEncodingProperties properties;  //只有一个有参构造器的情况下，参数的值就会从容器中拿  **public** HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {  **this**.properties = properties;  }    @Bean   //给容器中添加一个组件，这个组件的某些值需要从properties中获取  @ConditionalOnMissingBean(CharacterEncodingFilter.**class**) //判断容器没有这个组件？  **public** CharacterEncodingFilter characterEncodingFilter() {  CharacterEncodingFilter filter = **new** OrderedCharacterEncodingFilter();  filter.setEncoding(**this**.properties.getCharset().name());  filter.setForceRequestEncoding(**this**.properties.shouldForce(Type.REQUEST));  filter.setForceResponseEncoding(**this**.properties.shouldForce(Type.RESPONSE));  **return** filter;  } |

根据当前不同的条件判断，决定这个配置类是否生效？

一但这个配置类生效；这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的；

5）、所有在配置文件中能配置的属性都是在xxxxProperties类中封装者‘；配置文件能配置什么就可以参照某个功能对应的这个属性类

@ConfigurationProperties(prefix = "spring.http.encoding")  //从配置文件中获取指定的值和bean的属性进行绑定

public class HttpEncodingProperties {

    public static final Charset DEFAULT\_CHARSET = Charset.forName("UTF‐8");

#### 精髓：

1）、SpringBoot启动会加载大量的自动配置类

2）、我们看我们需要的功能有没有SpringBoot默认写好的自动配置类；

3）、我们再来看这个自动配置类中到底配置了哪些组件；（只要我们要用的组件有，我们就不需要再来配置了）

4）、给容器中自动配置类添加组件的时候，会从properties类中获取某些属性。我们就可以在配置文件中指定这些属性的值；

xxxxAutoConfigurartion：自动配置类；

给容器中添加组件

xxxxProperties:封装配置文件中相关属性；

### 2、细节

1、@Conditional派生注解（Spring注解版原生的@Conditional作用）

作用：必须是@Conditional指定的条件成立，才给容器中添加组件，配置配里面的所有内容才生效；

@Conditional扩展注解 作用（判断是否满足当前指定条件）

@ConditionalOnJava 系统的java版本是否符合要求

@ConditionalOnBean 容器中存在指定Bean；

@ConditionalOnMissingBean 容器中不存在指定Bean；

@ConditionalOnExpression 满足SpEL表达式指定

@ConditionalOnClass 系统中有指定的类

@ConditionalOnMissingClass 系统中没有指定的类

@ConditionalOnSingleCandidate 容器中只有一个指定的Bean，或者这个Bean是首选Bean

@ConditionalOnProperty 系统中指定的属性是否有指定的值

@ConditionalOnResource 类路径下是否存在指定资源文件

@ConditionalOnWebApplication 当前是web环境

@ConditionalOnNotWebApplication 当前不是web环境

@ConditionalOnJndi JNDI存在指定项

自动配置类必须在一定的条件下才能生效；

我们怎么知道哪些自动配置类生效；

我们可以通过启用 debug=true属性；来让控制台打印自动配置报告，这样我们就可以很方便的知道哪些自动配置

类生效；

# Web开发

SpringMVC自动配置

Spring Boot 自动配置好了SpringMVC: WebMvcAutoConfiguration

# 错误处理机制

1）、SpringBoot默认的错误处理机制

默认效果：

1）、浏览器，返回一个默认的错误页面

