# SpringBoot基础

## 学习目标：

1. 能够理解Spring的优缺点
2. 能够理解SpringBoot的特点
3. 能够理解SpringBoot的核心功能
4. 能够搭建SpringBoot的环境
5. 能够完成application.properties配置文件的配置
6. 能够完成application.yml配置文件的配置
7. 能够使用SpringBoot集成Mybatis
8. 能够使用SpringBoot集成Junit
9. 能够使用SpringBoot集成SpringData JPA

# 一、SpringBoot简介

## 原有Spring优缺点分析

### Spring的优点分析

Spring是Java企业版（Java Enterprise Edition，JEE，也称J2EE）的轻量级代替品。无需开发重量级的Enterprise JavaBean（EJB），Spring为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单 的Java对象（Plain Old Java Object，POJO）实现了EJB的功能。

### Spring的缺点分析

虽然Spring的组件代码是轻量级的，但它的配置却是重量级的。一开始，Spring用XML配置，而且是很多XML配 置。Spring 2.5引入了基于注解的组件扫描，这消除了大量针对应用程序自身组件的显式XML配置。Spring 3.0引入了基于Java的配置，这是一种类型安全的可重构配置方式，可以代替XML。

所有这些配置都代表了开发时的损耗。因为在思考Spring特性配置和解决业务问题之间需要进行思维切换，所以编 写配置挤占了编写应用程序逻辑的时间。和所有框架一样，Spring实用，但与此同时它要求的回报也不少。

除此之外，项目的依赖管理也是一件耗时耗力的事情。在环境搭建时，需要分析要导入哪些库的坐标，而且还需要 分析导入与之有依赖关系的其他库的坐标，一旦选错了依赖的版本，随之而来的不兼容问题就会严重阻碍项目的开 发进度。

## SpringBoot的概述

### SpringBoot解决上述Spring的缺点

SpringBoot对上述Spring的缺点进行的改善和优化，基于约定优于配置的思想，可以让开发人员不必在配置与逻辑 业务之间进行思维的切换，全身心的投入到逻辑业务的代码编写中，从而大大提高了开发的效率，一定程度上缩短 了项目周期。

### SpringBoot的特点

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

SpringBoot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的核心功能

起步依赖

起步依赖本质上是一个Maven项目对象模型（Project Object Model，POM），定义了对其他库的传递依赖，这些东西加在一起即支持某项功能。

简单的说，起步依赖就是将具备某种功能的坐标打包到一起，并提供一些默认的功能。 自动配置

Spring Boot的自动配置是一个运行时（更准确地说，是应用程序启动时）的过程，考虑了众多因素，才决定Spring配置应该用哪个，不该用哪个。该过程是Spring自动完成的。

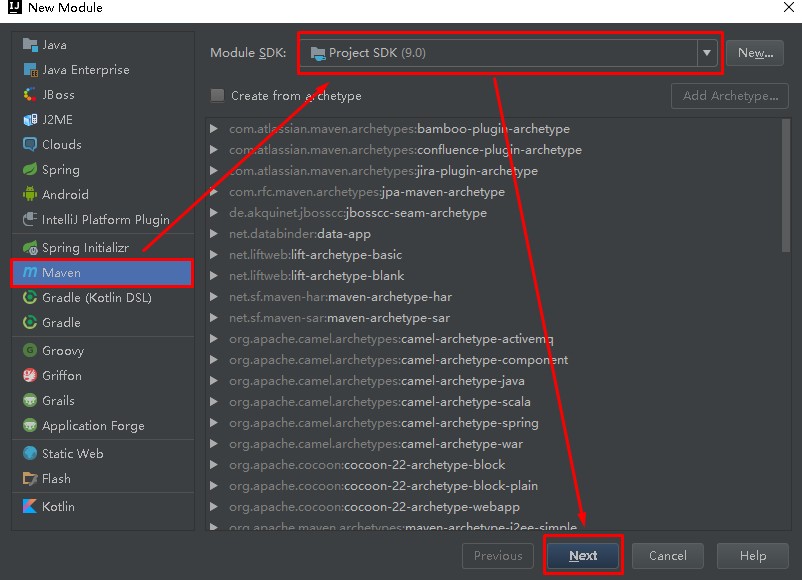
注意：起步依赖和自动配置的原理剖析会在第三章《SpringBoot原理分析》进行详细讲解

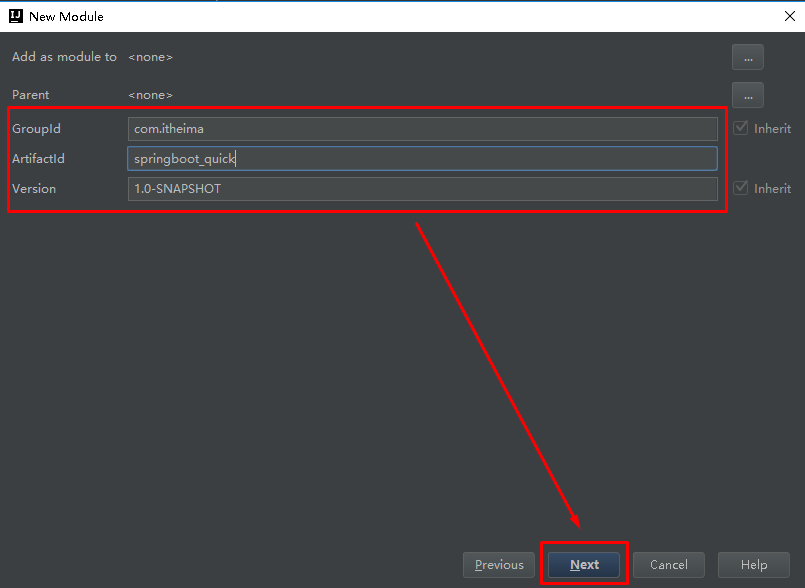
# 二、SpringBoot快速入门

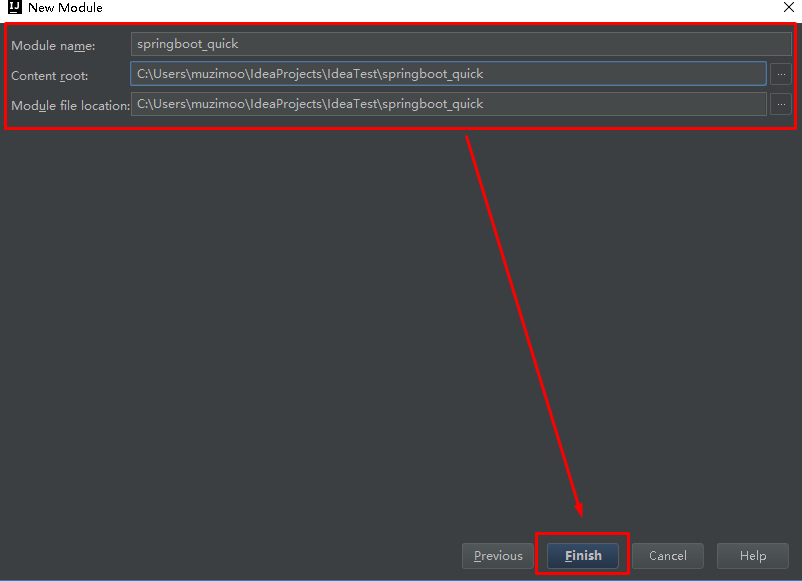
## 代码实现

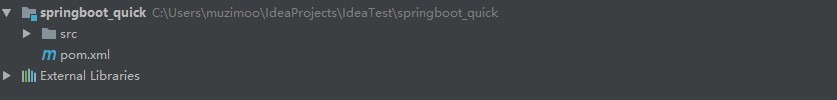
### 创建Maven工程

使用idea工具创建一个maven工程，该工程为普通的java工程即可









### 添加SpringBoot的起步依赖

SpringBoot要求，项目要继承SpringBoot的起步依赖spring-boot-starter-parent

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-parent</artifactId>
4. <version>2.0.1.RELEASE</version>
5. </parent>

SpringBoot要集成SpringMVC进行Controller的开发，所以项目要导入web的启动依赖

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

### 编写SpringBoot引导类

要通过SpringBoot提供的引导类起步SpringBoot才可以进行访问

}

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class);

}

@**SpringBootApplication**

public class MySpringBootApplication {

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

package com.itheima;

1

2

3

4

5

6

7

8

9

10

11

12

13

### 编写Controller

在引导类MySpringBootApplication同级包或者子级包中创建QuickStartController

}

@RequestMapping("/quick")

@ResponseBody

public String quick(){

return "springboot 访问成功!";

}

@Controller

public class QuickStartController {

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody;

package com.itheima.controller;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

### 测试

执行SpringBoot起步类的主方法，控制台打印日志如下：

main]

: Started MySpringBootApplication in 4.252

14

main]

: Tomcat started on port(s): 8080 (http)

13

main]

: Registering beans for JMX exposure on

2018-05-08 14:30:03.126 INFO 5672 --- [

o.s.j.e.a.AnnotationMBeanExporter startup

2018-05-08 14:30:03.196 INFO 5672 --- [

o.s.b.w.embedded.tomcat.TomcatWebServer with context path ''

2018-05-08 14:30:03.206 INFO 5672 --- [

com.itheima.MySpringBootApplication seconds (JVM running for 5.583)

12

(C:\Users\muzimoo\IdeaProjects\IdeaTest\springboot\_quick\target\classes started by

muzimoo in C:\Users\muzimoo\IdeaProjects\IdeaTest) 10 ... ... ...

11 o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/\*\*] onto handler of type

[class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

main]

: Starting MySpringBootApplication on

2018-05-08 14:29:59.714 INFO 5672 --- [

com.itheima.MySpringBootApplication DESKTOP-RRUNFUH with PID 5672

(v2.0.1.RELEASE)

:: Spring Boot ::

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) ) ' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

\_\_ \_ \_

\_

. \_\_\_\_

1

2

3

4

5

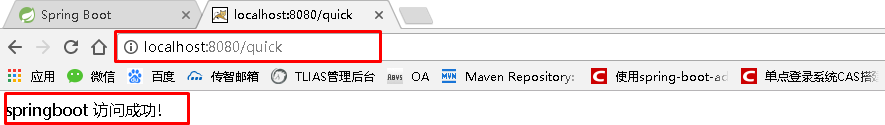
6

7

8

9

通过日志发现，Tomcat started on port(s): 8080 (http) with context path '' tomcat已经起步，端口监听8080，web应用的虚拟工程名称为空

打开浏览器访问url地址为：http://localhost:8080/quick

## 快速入门解析

### SpringBoot代码解析

@SpringBootApplication：标注SpringBoot的启动类，该注解具备多种功能（后面详细剖析） SpringApplication.run(MySpringBootApplication.class) 代表运行SpringBoot的启动类，参数为SpringBoot 启动类的字节码对象

### SpringBoot工程热部署

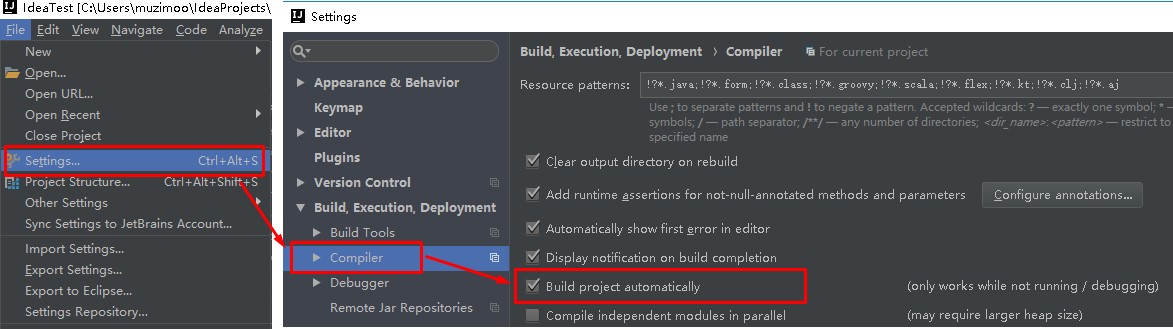
我们在开发中反复修改类、页面等资源，每次修改后都是需要重新启动才生效，这样每次启动都很麻烦，浪费了大 量的时间，我们可以在修改代码后不重启就能生效，在 pom.xml 中添加如下配置就可以实现这样的功能，我们称 之为热部署。

1 <!--热部署配置-->

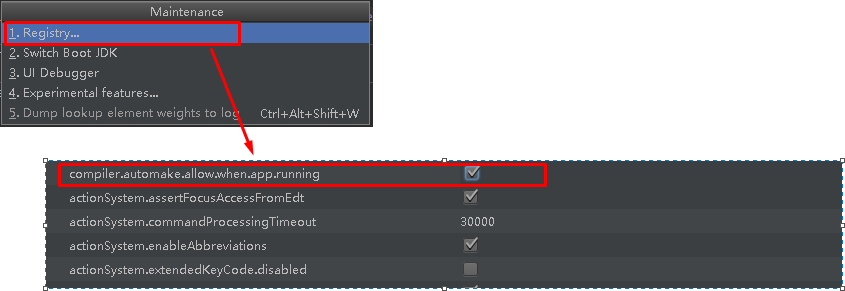
1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-devtools</artifactId>
4. </dependency>

注意：IDEA进行SpringBoot热部署失败原因

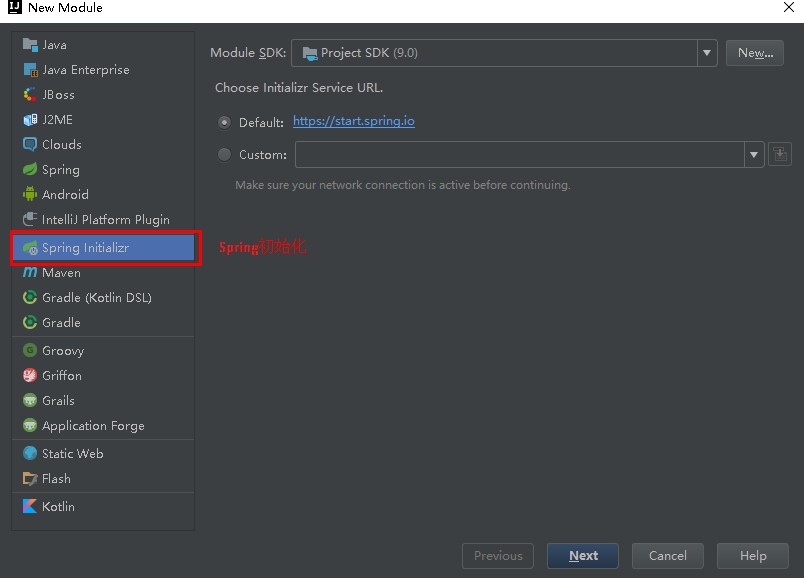
出现这种情况，并不是热部署配置问题，其根本原因是因为Intellij IEDA默认情况下不会自动编译，需要对IDEA进行自动编译的设置，如下：

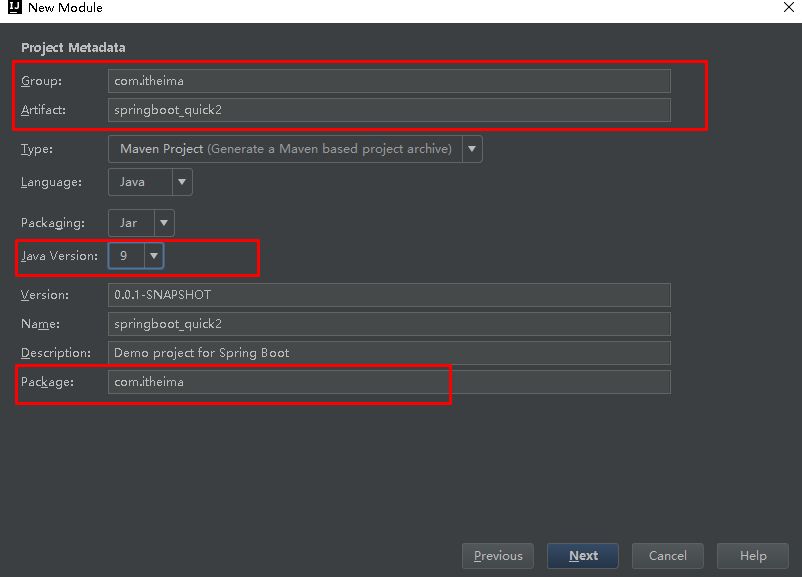


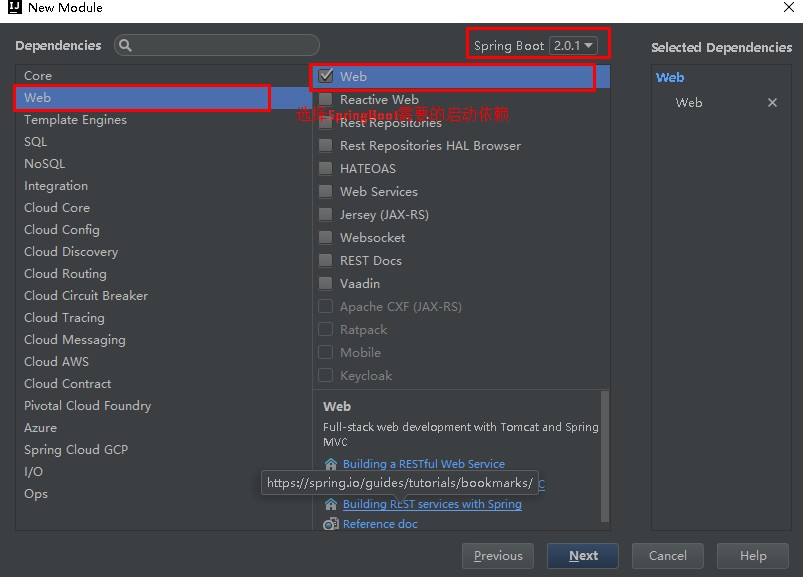
然后 Shift+Ctrl+Alt+/，选择Registry

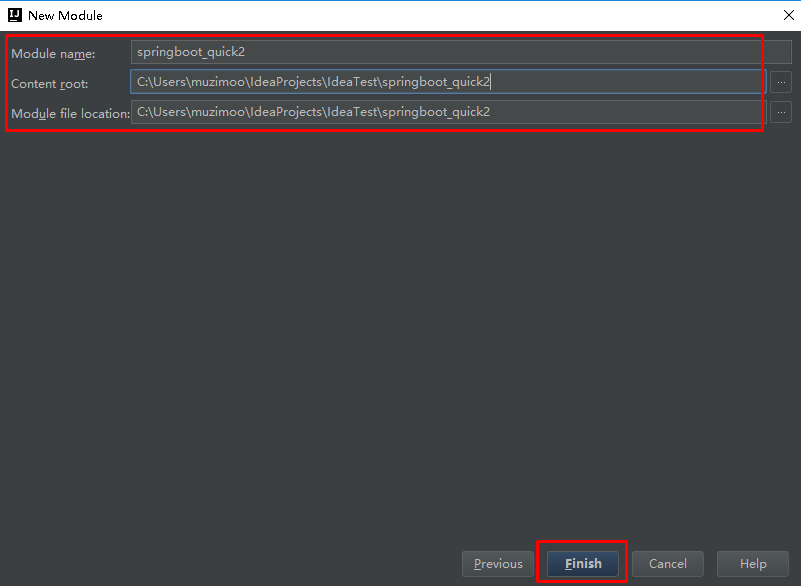


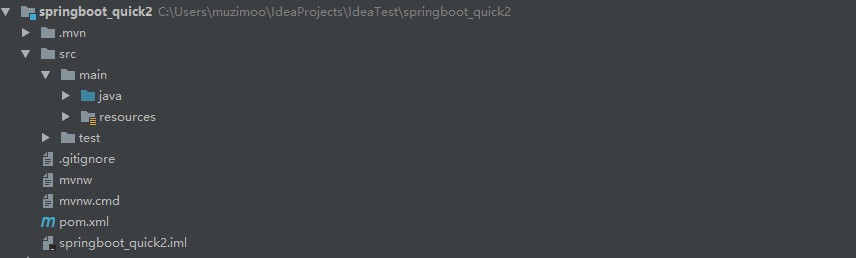
### 使用idea快速创建SpringBoot项目











通过idea快速创建的SpringBoot项目的pom.xml中已经导入了我们选择的web的起步依赖的坐标

* + - 1. <?xml version="1.0" encoding="UTF-8"?>
      2. <project xmlns="<http://maven.apache.org/POM/4.0.0>" xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
      3. xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0><http://maven.apache.org/xsd/maven-4.0.0.xsd>">
      4. <modelVersion>4.0.0</modelVersion> 5

1. <groupId>com.itheima</groupId>
2. <artifactId>springboot\_quick2</artifactId>
3. <version>0.0.1-SNAPSHOT</version>
4. <packaging>jar</packaging> 10
5. <name>springboot\_quick2</name>
6. <description>Demo project for Spring Boot</description> 13
7. <parent>
8. <groupId>org.springframework.boot</groupId>
9. <artifactId>spring-boot-starter-parent</artifactId>
10. <version>2.0.1.RELEASE</version>
11. <relativePath/> <!-- lookup parent from repository -->
12. </parent> 20
13. <properties>
14. <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15. <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
16. <java.version>9</java.version>
17. </properties> 26
18. <dependencies>
19. <dependency>
20. <groupId>org.springframework.boot</groupId>
21. <artifactId>spring-boot-starter-web</artifactId>
22. </dependency> 32
23. <dependency>
24. <groupId>org.springframework.boot</groupId>
25. <artifactId>spring-boot-starter-test</artifactId>
26. <scope>test</scope>

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

可以使用快速入门的方式创建Controller进行访问，此处不再赘述

# 三、SpringBoot原理分析

## 起步依赖原理分析

### 分析spring-boot-starter-parent

按住Ctrl点击pom.xml中的spring-boot-starter-parent，跳转到了spring-boot-starter-parent的pom.xml，xml配 置如下（只摘抄了部分重点配置）：

1. <parent>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-dependencies</artifactId>
4. <version>2.0.1.RELEASE</version>
5. <relativePath>../../spring-boot-dependencies</relativePath>
6. </parent>

按住Ctrl点击pom.xml中的spring-boot-starter-dependencies，跳转到了spring-boot-starter-dependencies的

pom.xml，xml配置如下（只摘抄了部分重点配置）：

dependencyManagement元素中声明所依赖的jar包的版本号等信息，所有子项目再次引入此依赖jar包时则无需显式的列出版本号。Maven会沿着父子层级向上寻找拥有dependencyManagement 元素的项目，然后使用它指定的版本号。

1. <properties>
2. <activemq.version>5.15.3</activemq.version>
3. <antlr2.version>2.7.7</antlr2.version>
4. <appengine-sdk.version>1.9.63</appengine-sdk.version>
5. <artemis.version>2.4.0</artemis.version>
6. <aspectj.version>1.8.13</aspectj.version>
7. <assertj.version>3.9.1</assertj.version>
8. <atomikos.version>4.0.6</atomikos.version>
9. <bitronix.version>2.1.4</bitronix.version>

10

<build-helper-maven-plugin.version>3.0.0</build-helper-maven-plugin.version>

11 <byte-buddy.version>1.7.11</byte-buddy.version> 12 ... ... ...

1. </properties>
2. <dependencyManagement>
3. <dependencies>
4. <dependency>
5. <groupId>org.springframework.boot</groupId>
6. <artifactId>spring-boot</artifactId>
7. <version>2.0.1.RELEASE</version>
8. </dependency>
9. <dependency>
10. <groupId>org.springframework.boot</groupId>
11. <artifactId>spring-boot-test</artifactId>
12. <version>2.0.1.RELEASE</version>
13. </dependency>

26 ... ... ...

1. </dependencies>
2. </dependencyManagement>
3. <build>
4. <pluginManagement>
5. <plugins>
6. <plugin>
7. <groupId>org.jetbrains.kotlin</groupId>
8. <artifactId>kotlin-maven-plugin</artifactId>
9. <version>${kotlin.version}</version>
10. </plugin>
11. <plugin>
12. <groupId>org.jooq</groupId>
13. <artifactId>jooq-codegen-maven</artifactId>
14. <version>${jooq.version}</version>
15. </plugin>
16. <plugin>
17. <groupId>org.springframework.boot</groupId>
18. <artifactId>spring-boot-maven-plugin</artifactId>
19. <version>2.0.1.RELEASE</version>
20. </plugin>

47 ... ... ...

1. </plugins>
2. </pluginManagement>
3. </build>

从上面的spring-boot-starter-dependencies的pom.xml中我们可以发现，一部分坐标的版本、依赖管理、插件管 理已经定义好，所以我们的SpringBoot工程继承spring-boot-starter-parent后已经具备版本锁定等配置了。所以 起步依赖的作用就是进行依赖的传递。

### 分析spring-boot-starter-web

按住Ctrl点击pom.xml中的spring-boot-starter-web，跳转到了spring-boot-starter-web的pom.xml，xml配置如 下（只摘抄了部分重点配置）：

1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xsi:schemaLocation="<http://maven.apache.org/POM/4.0.0>

<http://maven.apache.org/xsd/maven-4.0.0.xsd>" xmlns="<http://maven.apache.org/POM/4.0.0>"

* 1. xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>">
  2. <modelVersion>4.0.0</modelVersion>
  3. <parent>
  4. <groupId>org.springframework.boot</groupId>
  5. <artifactId>spring-boot-starters</artifactId>
  6. <version>2.0.1.RELEASE</version>
  7. </parent>
  8. <groupId>org.springframework.boot</groupId>
  9. <artifactId>spring-boot-starter-web</artifactId>
  10. <version>2.0.1.RELEASE</version>
  11. <name>Spring Boot Web Starter</name> 14

1. <dependencies>
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter</artifactId>
5. <version>2.0.1.RELEASE</version>
6. <scope>compile</scope>
7. </dependency>
8. <dependency>
9. <groupId>org.springframework.boot</groupId>
10. <artifactId>spring-boot-starter-json</artifactId>
11. <version>2.0.1.RELEASE</version>
12. <scope>compile</scope>
13. </dependency>
14. <dependency>
15. <groupId>org.springframework.boot</groupId>
16. <artifactId>spring-boot-starter-tomcat</artifactId>
17. <version>2.0.1.RELEASE</version>
18. <scope>compile</scope>
19. </dependency>
20. <dependency>
21. <groupId>org.hibernate.validator</groupId>
22. <artifactId>hibernate-validator</artifactId>
23. <version>6.0.9.Final</version>
24. <scope>compile</scope>
25. </dependency>
26. <dependency>
27. <groupId>org.springframework</groupId>
28. <artifactId>spring-web</artifactId>
29. <version>5.0.5.RELEASE</version>
30. <scope>compile</scope>
31. </dependency>
32. <dependency>
33. <groupId>org.springframework</groupId>
34. <artifactId>spring-webmvc</artifactId>
35. <version>5.0.5.RELEASE</version>
36. <scope>compile</scope>
37. </dependency>
38. </dependencies>
39. </project>

54

从上面的spring-boot-starter-web的pom.xml中我们可以发现，spring-boot-starter-web就是将web开发要使用的spring-web、spring-webmvc等坐标进行了“打包”，这样我们的工程只要引入spring-boot-starter-web起步依赖的 坐标就可以进行web开发了，同样体现了依赖传递的作用。

## 自动配置原理解析

按住Ctrl点击查看启动类MySpringBootApplication上的注解@SpringBootApplication

1. @SpringBootApplication
2. public class MySpringBootApplication {
3. public static void main(String[] args) {
4. SpringApplication.run(MySpringBootApplication.class); 5 }

6 }

注解@SpringBootApplication的源码

|  |  |  |
| --- | --- | --- |
| 14 |  | \* @return the classes to exclude |
| 15 |  | \*/ |
| 16 |  | @AliasFor(annotation = EnableAutoConfiguration.class) |
| 17 |  | Class<?>[] exclude() default {}; |
| 18 |  |  |
| 19 |  | ... ... ... |
| 20 |  |  |
| 21 | } |  |

其中，

1

2

3

4

5

6

7

8

9

10

11

12

13

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME) @Documented

@Inherited @SpringBootConfiguration @EnableAutoConfiguration @ComponentScan(excludeFilters = {

@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class), @Filter(type = FilterType.CUSTOM, classes =

AutoConfigurationExcludeFilter.class) })

public @interface SpringBootApplication {

/\*\*

\* Exclude specific auto-configuration classes such that they will never be applied.

@SpringBootConﬁguration：等同与@Conﬁguration，既标注该类是Spring的一个配置类@EnableAutoConﬁguration：SpringBoot自动配置功能开启

按住Ctrl点击查看注解@EnableAutoConﬁguration

1. @Target(ElementType.TYPE)
2. @Retention(RetentionPolicy.RUNTIME)
3. @Documented
4. @Inherited
5. @AutoConfigurationPackage
6. @Import(AutoConfigurationImportSelector.class)
7. public @interface EnableAutoConfiguration { 8 ... ... ...

9 }

其中，@Import(AutoConﬁgurationImportSelector.class) 导入了AutoConﬁgurationImportSelector类按住Ctrl点击查看AutoConﬁgurationImportSelector源码

}

return configurations;

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,

AnnotationAttributes attributes) {

List<String> configurations = SpringFactoriesLoader.loadFactoryNames( getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());

public String[] selectImports(AnnotationMetadata annotationMetadata) {

... ... ...

List<String> configurations = getCandidateConfigurations(annotationMetadata,

attributes);

configurations = removeDuplicates(configurations);

Set<String> exclusions = getExclusions(annotationMetadata, attributes); checkExcludedClasses(configurations, exclusions); configurations.removeAll(exclusions);

configurations = filter(configurations, autoConfigurationMetadata); fireAutoConfigurationImportEvents(configurations, exclusions); return StringUtils.toStringArray(configurations);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

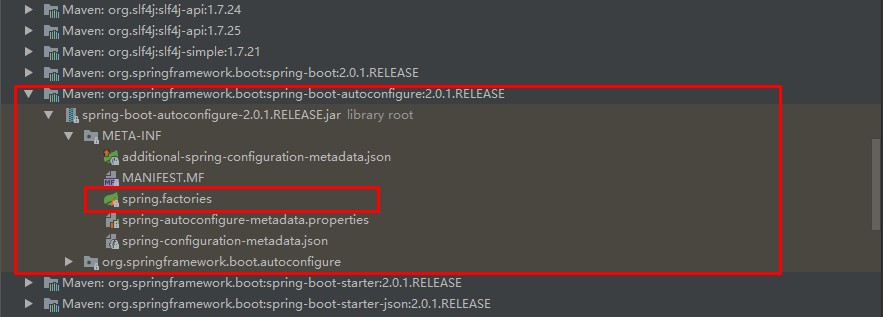
19

20

21

22

其中，SpringFactoriesLoader.loadFactoryNames 方法的作用就是从META-INF/spring.factories文件中读取指定类对应的类名称列表



spring.factories 文件中有关自动配置的配置信息如下：

... ... ...

org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConf

iguration,\ org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration

,\ org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfigu ration,\ org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\ org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\

org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\

5

6

7

8

9

10

1 ... ... ...

2

3

4

上面配置文件存在大量的以Conﬁguration为结尾的类名称，这些类就是存有自动配置信息的类，而

SpringApplication在获取这些类名后再加载

我们以ServletWebServerFactoryAutoConﬁguration为例来分析源码：

1. @Configuration
2. @AutoConfigureOrder(Ordered.HIGHEST\_PRECEDENCE)
3. @ConditionalOnClass(ServletRequest.class)
4. @ConditionalOnWebApplication(type = Type.SERVLET)
5. @EnableConfigurationProperties(ServerProperties.class)
6. @Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
7. ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
8. ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
9. ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
10. public class ServletWebServerFactoryAutoConfiguration { 11 ... ... ...

12 }

13

其中，

@EnableConﬁgurationProperties(ServerProperties.class) 代表加载ServerProperties服务器配置属性类

进入ServerProperties.class源码如下：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

@ConfigurationProperties(prefix = "server", ignoreUnknownFields = true)

public class ServerProperties {

/\*\*

\* Server HTTP port.

\*/

private Integer port;

/\*\*

\* Network address to which the server should bind.

\*/

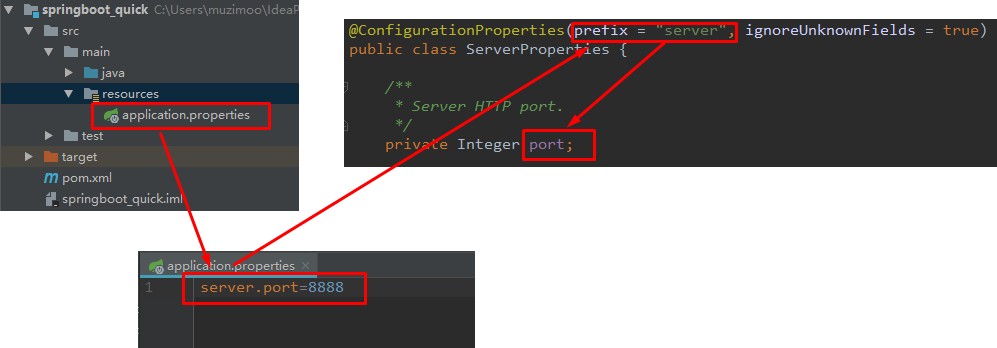
private InetAddress address;

... ... ...

}

其中，

preﬁx = "server" 表示SpringBoot配置文件中的前缀，SpringBoot会将配置文件中以server开始的属性映射到该类的字段中。映射关系如下：



# 四、SpringBoot的配置文件

## SpringBoot配置文件类型

### SpringBoot配置文件类型和作用

SpringBoot是基于约定的，所以很多配置都有默认值，但如果想使用自己的配置替换默认配置的话，就可以使用application.properties或者application.yml（application.yaml）进行配置。

SpringBoot默认会从Resources目录下加载application.properties或application.yml（application.yaml）文件

其中，application.properties文件是键值对类型的文件，之前一直在使用，所以此处不在对properties文件的格式 进行阐述。除了properties文件外，SpringBoot还可以使用yml文件进行配置，下面对yml文件进行讲解。

### application.yml配置文件

#### yml配置文件简介

YML文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的的数据数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的，可以被支持YAML库的不同的编程语言程序导 入，比如： C/C++, Ruby, Python, Java, Perl, C#, PHP等。YML文件是以数据为核心的，比传统的xml方式更加简洁。

YML文件的扩展名可以使用.yml或者.yaml。

#### yml配置文件的语法

* + - * 1. **配置普通数据**语法： key: value

示例代码：

1 name: haohao

注意：value之前有一个空格

* + - * 1. **配置对象数据**语法：

key:

key1: value1 key2: value2

或者：

key: {key1: value1,key2: value2}

示例代码：

#或者

person: {name: haohao,age: 31,addr: beijing}

person:

name: haohao age: 31

addr: beijing

1

2

3

4

5

6

7

8

注意：key1前面的空格个数不限定，在yml语法中，相同缩进代表同一个级别

* + - * 1. **配置Map数据**同上面的对象写法
        2. **配置数组（List、Set）数据**语法：

key:

value1

value2

或者：

key: [value1,value2]

示例代码：

#或者

city: [beijing,tianjin,shanghai,chongqing] #集合中的元素是对象形式

student:

* name: zhangsan age: 18

score: 100

* name: lisi age: 28

score: 88

* name: wangwu age: 38

score: 90

city:

* beijing
* tianjin
* shanghai
* chongqing

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

注意：value1与之间的 - 之间存在一个空格

### SpringBoot配置信息的查询

上面提及过，SpringBoot的配置文件，主要的目的就是对配置信息进行修改的，但在配置时的key从哪里去查询 呢？我们可以查阅SpringBoot的官方文档

[文 档 URL：https://docs.spring.io/spring-boot/docs/2.0.1.RELEASE/reference/htmlsingle/#common-application- properties](https://docs.spring.io/spring-boot/docs/2.0.1.RELEASE/reference/htmlsingle/#common-application-properties)

常用的配置摘抄如下：

1. # QUARTZ SCHEDULER (QuartzProperties)
2. spring.quartz.jdbc.initialize-schema=embedded # Database schema initialization mode.
3. spring.quartz.jdbc.schema=classpath:org/quartz/impl/jdbcjobstore/tables\_@@platform@@. sql # Path to the SQL file to use to initialize the database schema.
4. spring.quartz.job-store-type=memory # Quartz job store type.

5 spring.quartz.properties.\*= # Additional Quartz Scheduler properties. 6

|  |  |  |
| --- | --- | --- |
| 7 | # | ---------------------------------------- |
| 8 | # | WEB PROPERTIES |
| 9 | # | ---------------------------------------- |
| 10 |  |  |
| 11 | # | EMBEDDED SERVER CONFIGURATION (ServerProperties) |

1. server.port=8080 # Server HTTP port.
2. server.servlet.context-path= # Context path of the application.
3. server.servlet.path=/ # Path of the main dispatcher servlet. 15
4. # HTTP encoding (HttpEncodingProperties)
5. spring.http.encoding.charset=UTF-8 # Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.

18

1. # JACKSON (JacksonProperties)
2. spring.jackson.date-format= # Date format string or a fully-qualified date format class name. For instance, `yyyy-MM-dd HH:mm:ss`.

21

1. # SPRING MVC (WebMvcProperties)
2. spring.mvc.servlet.load-on-startup=-1 # Load on startup priority of the dispatcher servlet.
3. spring.mvc.static-path-pattern=/\*\* # Path pattern used for static resources.
4. spring.mvc.view.prefix= # Spring MVC view prefix.
5. spring.mvc.view.suffix= # Spring MVC view suffix. 27
6. # DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
7. spring.datasource.driver-class-name= # Fully qualified name of the JDBC driver. Auto- detected based on the URL by default.
8. spring.datasource.password= # Login password of the database.
9. spring.datasource.url= # JDBC URL of the database.
10. spring.datasource.username= # Login username of the database. 33
11. # JEST (Elasticsearch HTTP client) (JestProperties)
12. spring.elasticsearch.jest.password= # Login password.
13. spring.elasticsearch.jest.proxy.host= # Proxy host the HTTP client should use.
14. spring.elasticsearch.jest.proxy.port= # Proxy port the HTTP client should use.
15. spring.elasticsearch.jest.read-timeout=3s # Read timeout.
16. spring.elasticsearch.jest.username= # Login username. 40

我们可以通过配置application.poperties 或者 application.yml 来修改SpringBoot的默认配置例如：

application.properties文件

1. server.port=8888
2. server.servlet.context-path=demo

application.yml文件

1 server:

2 port: 8888

1. servlet:
2. context-path: /demo

## 配置文件与配置类的属性映射方式

### 使用注解@Value映射

我们可以通过@Value注解将配置文件中的值映射到一个Spring管理的Bean的字段上例如：

application.properties配置如下：

1. person:
2. name: zhangsan
3. age: 18

或者，application.yml配置如下：

1. person:
2. name: zhangsan
3. age: 18

实体Bean代码如下：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

@Controller

public class QuickStartController {

@Value("${person.name}")

private String name; @Value("${person.age}") private Integer age;

@RequestMapping("/quick") @ResponseBody

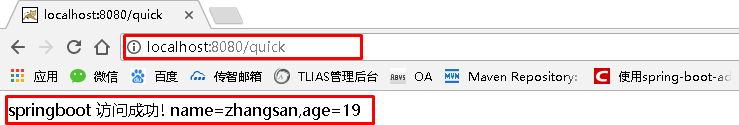
public String quick(){

return "springboot 访问成功! name="+name+",age="+age;

}

}

浏览器访问地址：http://localhost:8080/quick 结果如下：



### 使用注解@ConfigurationProperties映射

通过注解@ConﬁgurationProperties(preﬁx="配置文件中的key的前缀")可以将配置文件中的配置自动与实体进行映 射

application.properties配置如下：

1. person:
2. name: zhangsan
3. age: 18

或者，application.yml配置如下：

1. person:
2. name: zhangsan
3. age: 18

实体Bean代码如下：

}

private String name;

private Integer age;

@RequestMapping("/quick") @ResponseBody

public String quick(){

return "springboot 访问成功! name="+name+",age="+age;

}

public void setName(String name) { this.name = name;

}

public void setAge(Integer age) { this.age = age;

}

@Controller

@ConfigurationProperties(prefix = "person") public class QuickStartController {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

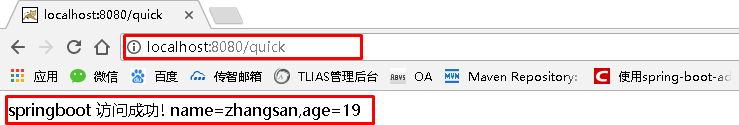
17

18

19

20

21

浏览器访问地址：http://localhost:8080/quick 结果如下：

注意：使用@ConﬁgurationProperties方式可以进行配置文件与实体字段的自动映射，但需要字段必须提供set方 法才可以，而使用@Value注解修饰的字段不需要提供set方法

# 五、SpringBoot与整合其他技术

## SpringBoot整合Mybatis

### 添加Mybatis的起步依赖

1. <!--mybatis起步依赖-->
2. <dependency>
3. <groupId>org.mybatis.spring.boot</groupId>
4. <artifactId>mybatis-spring-boot-starter</artifactId>
5. <version>1.1.1</version>
6. </dependency>

### 添加数据库驱动坐标

1. <!-- MySQL连接驱动 -->
2. <dependency>
3. <groupId>mysql</groupId>
4. <artifactId>mysql-connector-java</artifactId>
5. </dependency>

### 添加数据库连接信息

在application.properties中添加数据量的连接信息

1. #DB Configuration:
2. spring.datasource.driverClassName=com.mysql.jdbc.Driver
3. spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test? useUnicode=true&characterEncoding=utf8
4. spring.datasource.username=root
5. spring.datasource.password=root

### 创建user表

在test数据库中创建user表

-- ----------------------------

-- Records of user

-- ----------------------------

INSERT INTO `user` VALUES ('1', 'zhangsan', '123', '张三'); INSERT INTO `user` VALUES ('2', 'lisi', '123', '李四');

-- ----------------------------

-- Table structure for `user`

-- ---------------------------- DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`username` varchar(50) DEFAULT NULL,

`password` varchar(50) DEFAULT NULL,

`name` varchar(50) DEFAULT NULL, PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=10 DEFAULT CHARSET=utf8;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

### 创建实体Bean

}

//此处省略getter和setter方法 .. ..

public class User {

// 主 键

private Long id;

// 用户名

private String username;

// 密 码

private String password;

// 姓 名

private String name;

1

2

3

4

5

6

7

8

9

10

11

12

13

### 编写Mapper

1. @Mapper
2. public interface UserMapper {
3. public List<User> queryUserList(); 4 }

注意：@Mapper标记该类是一个mybatis的mapper接口，可以被spring boot自动扫描到spring上下文中

### 配置Mapper映射文件

在src\main\resources\mapper路径下加入UserMapper.xml配置文件"

1. <?xml version="1.0" encoding="utf-8" ?>
2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "<http://mybatis.org/dtd/mybatis-3-mapper.dtd>" >
3. <mapper namespace="com.itheima.mapper.UserMapper">
4. <select id="queryUserList" resultType="user">
5. select \* from user
6. </select>
7. </mapper>

### 在application.properties中添加mybatis的信息

1. #spring集成Mybatis环境
2. #pojo别名扫描包
3. mybatis.type-aliases-package=com.itheima.domain
4. #加载Mybatis映射文件
5. mybatis.mapper-locations=classpath:mapper/\*Mapper.xml

### 编写测试Controller

}

@Autowired

private UserMapper userMapper;

@RequestMapping("/queryUser") @ResponseBody

public List<User> queryUser(){

List<User> users = userMapper.queryUserList(); return users;

}

@Controller

public class MapperController {

1

2

3

4

5

6

7

8

9

10

11

12

13

14

### 测试

## 5.2 SpringBoot整合Junit

### 添加Junit的起步依赖

1. <!--测试的起步依赖-->
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-test</artifactId>
5. <scope>test</scope>
6. </dependency>

### 编写测试类

}

@Autowired

private UserMapper userMapper;

@Test

public void test() {

List<User> users = userMapper.queryUserList(); System.out.println(users);

}

@RunWith(SpringRunner.class)

@SpringBootTest(classes = MySpringBootApplication.class) public class MapperTest {

import com.itheima.MySpringBootApplication;

import com.itheima.domain.User; import com.itheima.mapper.UserMapper; 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 java.util.List;

package com.itheima.test;

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

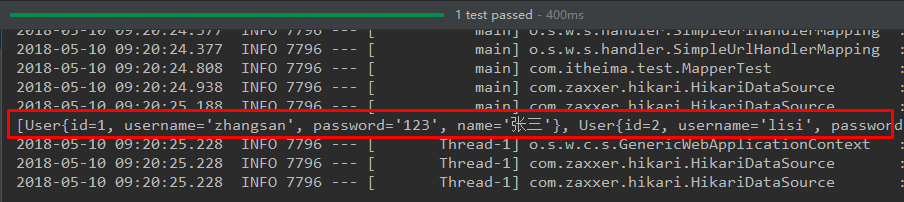
其中，

SpringRunner继承自SpringJUnit4ClassRunner，使用哪一个Spring提供的测试测试引擎都可以

1 public final class SpringRunner extends SpringJUnit4ClassRunner

@SpringBootTest的属性指定的是引导类的字节码对象

### 控制台打印信息



* 1. **SpringBoot整合Spring Data JPA**
     1. **添加Spring Data JPA的起步依赖**

1. <!-- springBoot JPA的起步依赖 -->
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-data-jpa</artifactId>
5. </dependency>
   * 1. **添加数据库驱动依赖**
6. <!-- MySQL连接驱动 -->
7. <dependency>
8. <groupId>mysql</groupId>
9. <artifactId>mysql-connector-java</artifactId>
10. </dependency>
    * 1. **在application.properties中配置数据库和jpa的相关属性**

#JPA Configuration:

spring.jpa.database=MySQL spring.jpa.show-sql=true spring.jpa.generate-ddl=true spring.jpa.hibernate.ddl-auto=update

spring.jpa.hibernate.naming\_strategy=org.hibernate.cfg.ImprovedNamingStrategy

#DB Configuration:

spring.datasource.driverClassName=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test? useUnicode=true&characterEncoding=utf8 spring.datasource.username=root spring.datasource.password=root

1

2

3

4

5

6

7

8

9

10

11

12

* + 1. **创建实体配置实体**

1 @Entity

2

3

4

5

6

7

8

9

10

11

12

13

14

15

public class User {

// 主 键

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY) private Long id;

// 用户名

private String username;

// 密 码

private String password;

// 姓 名

private String name;

//此处省略setter和getter方法... ...

}

* + 1. **编写UserRepository**

1. public interface UserRepository extends JpaRepository<User,Long>{
2. public List<User> findAll(); 3 }
   * 1. **编写测试类**

}

@Autowired

private UserRepository userRepository;

@Test

public void test(){

List<User> users = userRepository.findAll(); System.out.println(users);

}

@RunWith(SpringRunner.class)

@SpringBootTest(classes=MySpringBootApplication.class) public class JpaTest {

1

2

3

4

5

6

7

8

9

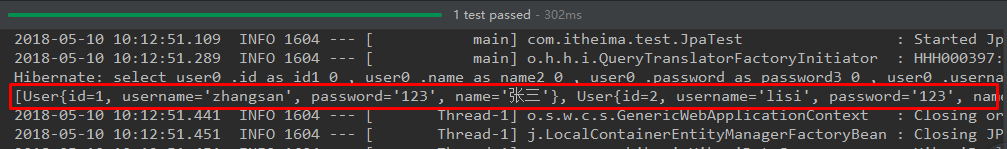
10

11

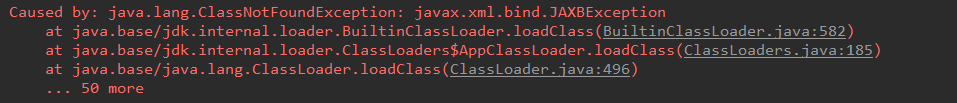
12

13

14

* + 1. **控制台打印信息**

注意：如果是jdk9，执行报错如下：



原因：jdk缺少相应的jar

解决方案：手动导入对应的maven坐标，如下：

1. <!--jdk9需要导入如下坐标-->
2. <dependency>
3. <groupId>javax.xml.bind</groupId>
4. <artifactId>jaxb-api</artifactId>
5. <version>2.3.0</version>
6. </dependency>

## SpringBoot整合Redis

### 添加redis的起步依赖

1. <!-- 配置使用redis启动器 -->
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-data-redis</artifactId>
5. </dependency>
   * 1. **配置redis的连接信息**
6. #Redis
7. spring.redis.host=127.0.0.1
8. spring.redis.port=6379
   * 1. **注入RedisTemplate测试redis操作**

1

2

3

4

5

6

7

8

9

10

11

12

@RunWith(SpringRunner.class)

@SpringBootTest(classes = SpringbootJpaApplication.class) public class RedisTest {

@Autowired

private UserRepository userRepository;

@Autowired

private RedisTemplate<String, String> redisTemplate;

13

@Test

public void test() throws JsonProcessingException {

//从redis缓存中获得指定的数据

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

String userListData = redisTemplate.boundValueOps("user.findAll").get();

//如果redis中没有数据的话if(null==userListData){

//查询数据库获得数据

List<User> all = userRepository.findAll();

//转换成json格式字符串

ObjectMapper om = new ObjectMapper(); userListData = om.writeValueAsString(all);

//将数据存储到redis中，下次在查询直接从redis中获得数据，不用在查询数据库

redisTemplate.boundValueOps("user.findAll").set(userListData); System.out.println("===============从数据库获得数据===============");

}else{

System.out.println("===============从redis缓存中获得数据===============");

}

System.out.println(userListData);

}

}