

# 第1章 Spring Boot 概要

## 1.1 Spring Boot 介绍

- 随着动态语言的流行 ( Ruby、Scala、Node.js ) , Java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整合难度大。
- 在上述环境下，Spring Boot由此诞生，它的设计是为了使您能够尽可能快地启动和运行。它使用“习惯优于配置”（项目中存在大量的配置，而 Spring Boot 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 容器）、准生产强力的基于 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。提供了 J2EE 开发的一站式解决方案。

## 1.2 Spring Boot 优点

- 快速构建独立运行的Spring项目；
- 无须依赖外部Servlet容器，应用无需打成WAR包；项目可以打成jar包独自运行；
- 提供一系列 starter pom 来简化 Maven 的依赖加载；
- 大量的自动配置，对主流开发框架的无配置集成；
- 无须配置XML，开箱即用，简化开发，同时也可以修改默认值来满足特定的需求；
- 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；
- 极大提高了开发、部署效率。

# 第2章 Spring Boot 入门开发

技术点要求：

- 熟练 Spring 框架的使用
- 熟练 Maven 依赖管理与项目构建
- 熟练使用 Eclipse 或 IDEA

## 2.1 环境要求

- jdk1.8 (Spring Boot 推荐jdk1.8及以上)：java version "1.8.0\_151"
- Maven 3.x (maven 3.2 以上版本)：Apache Maven 3.3.9
- IntelliJ IDEA：IntelliJ IDEA 2018.2.2 x64
- SpringBoot 使用当前最新稳定版本：第5章web开发前 2.0.6.RELEASE，后面使用 2.1.0.RELEASE

# Spring Boot 2.1.0



Overview Learn Samples

## Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

2.1.0 **CURRENT** **GA**

Reference Doc.

API Doc.

2.1.1 **SNAPSHOT**

Reference Doc.

API Doc.

2.0.7 **SNAPSHOT**

Reference Doc.

API Doc.

2.0.6 **GA**

Reference Doc.

API Doc.

1.5.18 **SNAPSHOT**

Reference Doc.

API Doc.

Web开发后，使用2.1.0版本

Web开发前，使用2.0.6版本

## 2.2 修改Maven配置文件

注意：本课程所有的配置内容和代码如果要复制，请复制老师源码里面的。不要复制此pdf文档，因为有格式问题。

- 在 Maven 安装目录下的 settings.xml 配置文件中,添加如下配置

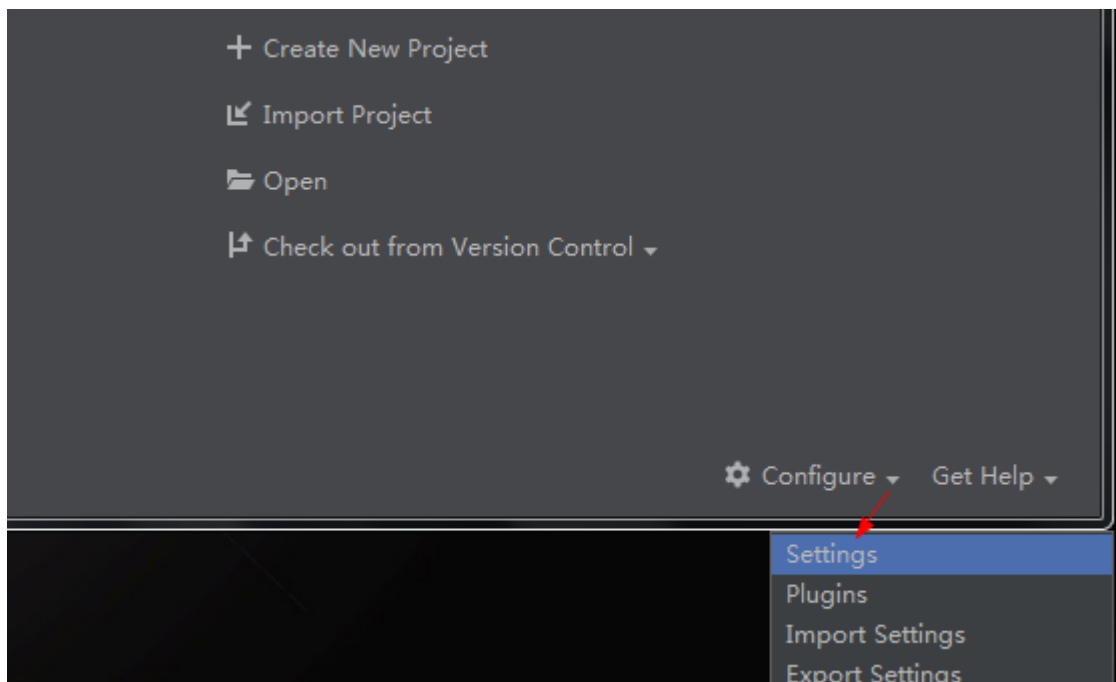
可以参考网盘下载的 04-软件\settings.xml

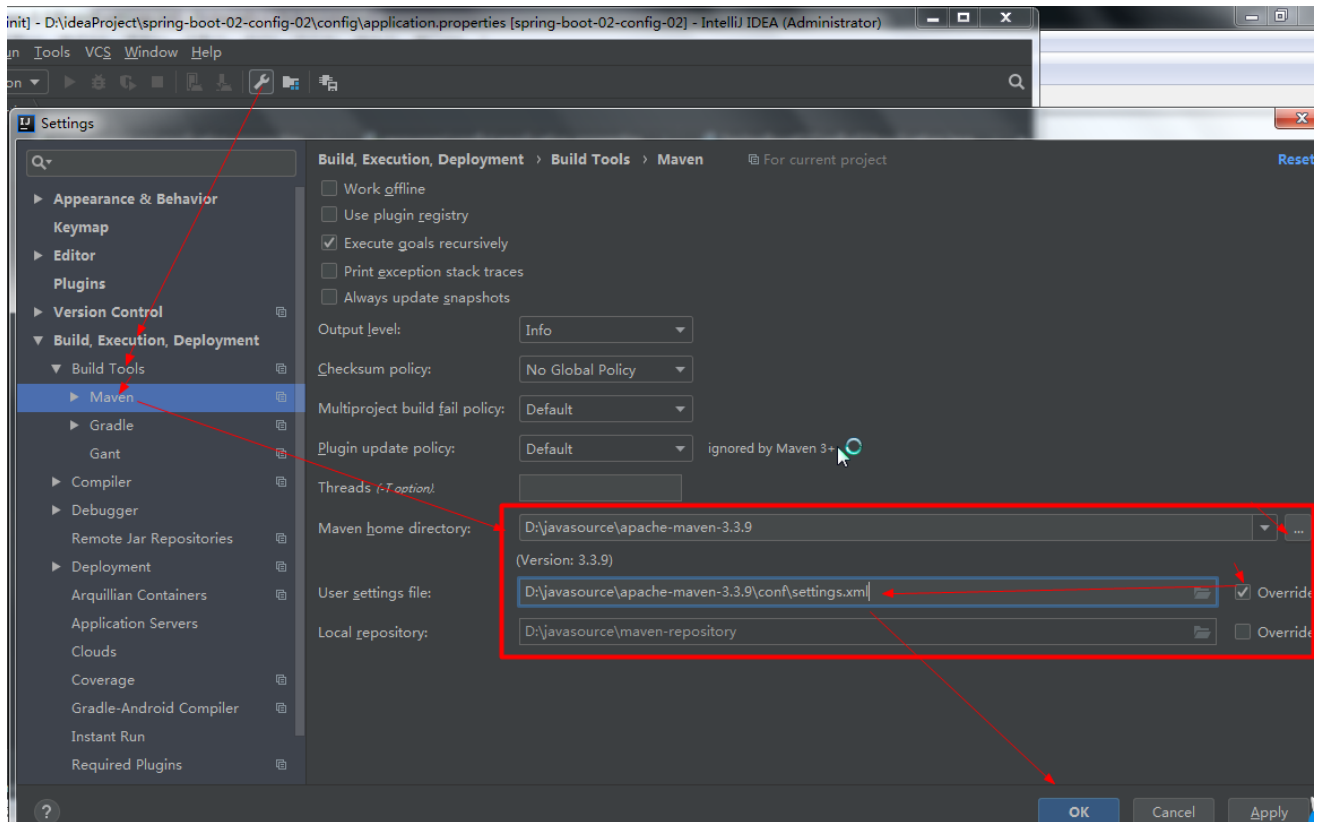
```
1 <!--开始处更改下载依赖的存放路径，以下目录需要已经创建-->
2 <localRepository>D:\javasource\maven-repository</localRepository>
3
4 <mirrors>
5   <!--在 mirrors 标签下 添加阿里云maven私服库-->
6   <mirror>
7     <id>alimaven</id>
8     <mirrorOf>central</mirrorOf>
9     <name>aliyun maven</name>
10    <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
11  </mirror>
12 </mirrors>
13
14 <profiles>
15   <!-- 在 profiles 标签下指定jdk版本 -->
16   <profile>
17     <id>jdk-1.8</id>
18     <activation>
```

```
19     <activeByDefault>true</activeByDefault>
20     <jdk>1.8</jdk>
21 </activation>
22 <properties>
23     <maven.compiler.source>1.8</maven.compiler.source>
24     <maven.compiler.target>1.8</maven.compiler.target>
25     <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
26 </properties>
27 </profile>
28 </profiles>
```

## 2.3 IntelliJ IDEA 设置

- 在idea上将 maven 环境添加进来



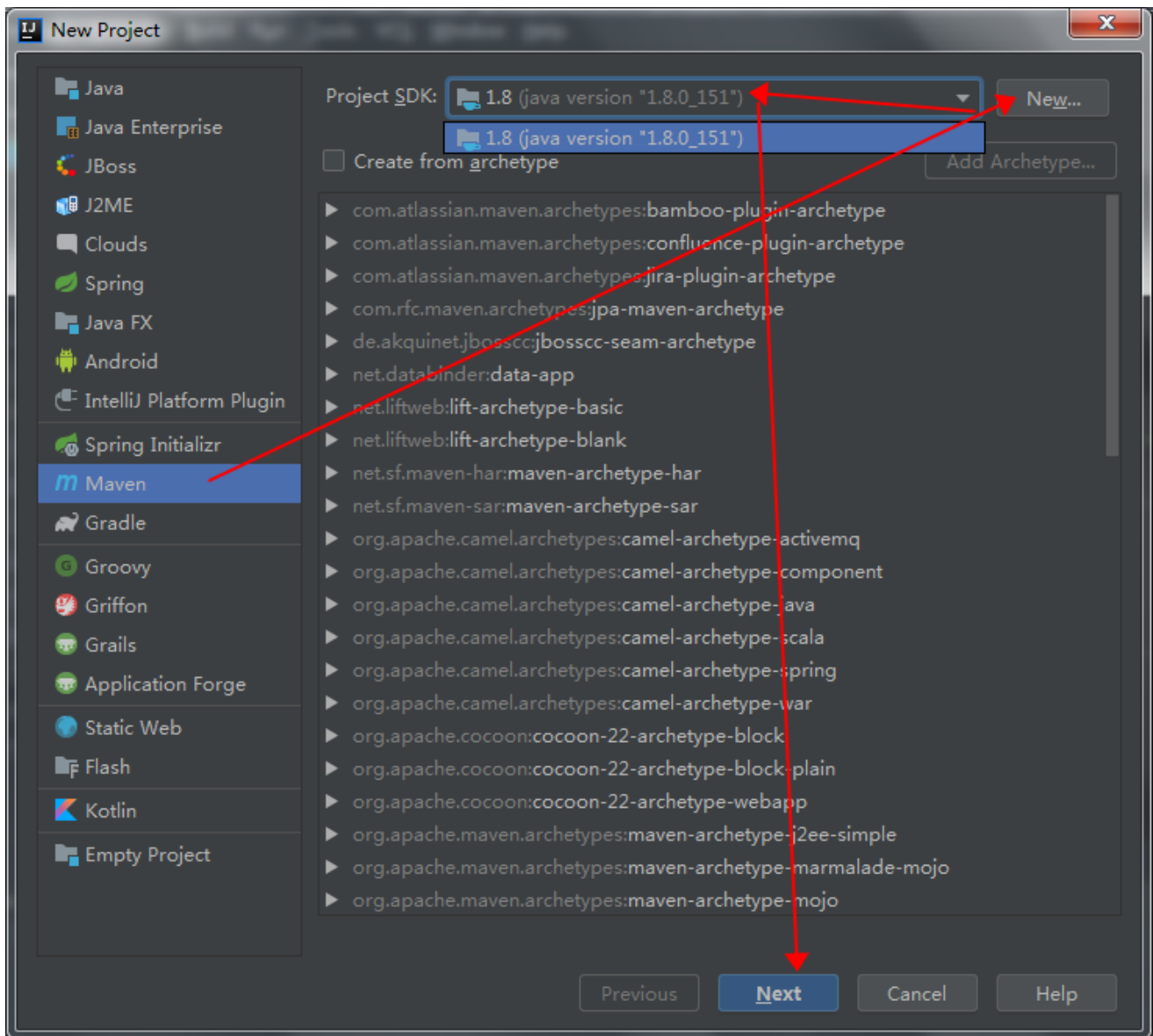




## 2.4 快速构建 Spring Boot 项目

- 需求：浏览器发送 /hello 请求，服务器接受请求并处理，响应 Hello World 字符串
- 分析：构建 Spring Boot 项目，事实上建立的就是一个 Maven 项目

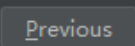

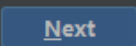
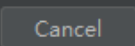
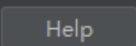
### 2.4.1 创建 Maven工程

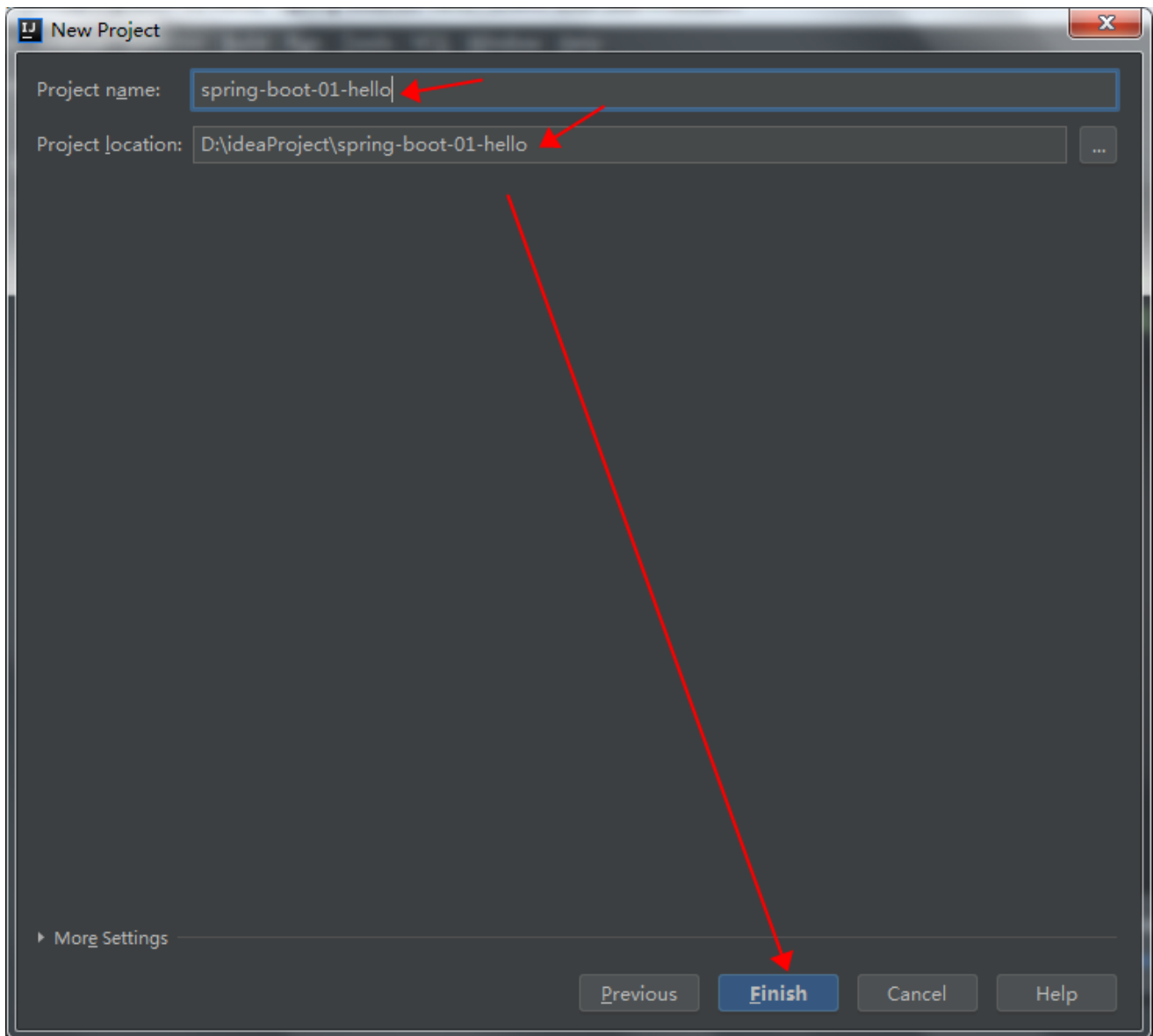
- 在 IDEA上新建一个空的jar类型 的 maven 工程



 New Project 

GroupId	<input type="text" value="com.mengxuegu"/> 坐标	<input checked="" type="checkbox"/> Inherit
ArtifactId	<input type="text" value="spring-boot-01-hello"/> 工程名	
Version	<input type="text" value="1.0-SNAPSHOT"/>	<input checked="" type="checkbox"/> Inherit



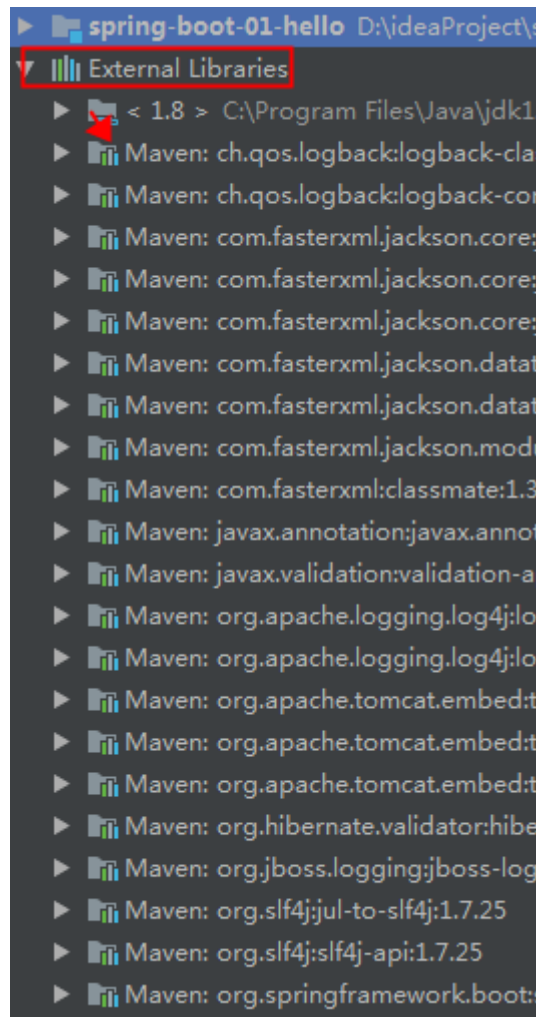
## 2.4.2 修改 pom.xml

- 在 pom.xml 中添加 Spring Boot 相关的父级依赖，`spring-boot-starter-parent` 是一个特殊的starter，它提供了项目相关的默认依赖，使用它之后，常用的包依赖可以省去 version 标签。
- 在 dependencies 添加构建 Web 项目相关的依赖

```
1 <!-- Inherit defaults from Spring Boot -->
2 <parent>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-parent</artifactId>
5   <version>2.0.6.RELEASE</version>
6 </parent>
7
8 <!-- Add typical dependencies for a web application -->
9 <dependencies>
10  <dependency>
11    <groupId>org.springframework.boot</groupId>
12    <artifactId>spring-boot-starter-web</artifactId>
13  </dependency>
```

14 </dependencies>

- 我们会惊奇地发现，我们的工程自动添加了好多好多jar包，这些jar包正是开发时需要导入的jar包



### 2.4.3 创建控制器 Controller

```
1 package com.mengxuegu.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.ResponseBody;
6
7 @Controller
8 public class HelloController {
9
10     @ResponseBody
11     @RequestMapping("/hello")
12     public String hello() {
13         return "HelloWorld...";
14     }
15 }
16
```



## 2.4.4 创建一个引导类

- 主要作用是作为启动 Spring Boot 项目的入口

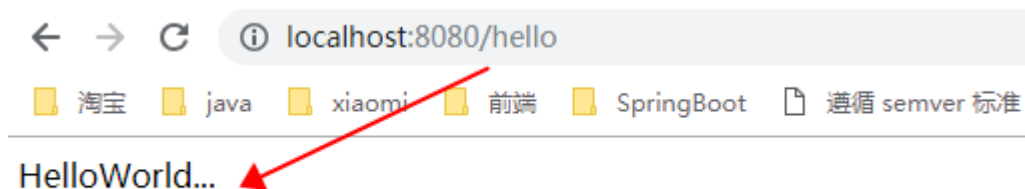
```
1 package com.mengxuegu;  
2  
3 import org.springframework.boot.SpringApplication;  
4 import org.springframework.boot.autoconfigure.SpringBootApplication;  
5  
6 /**  
7  * @SpringBootApplication 用于标识一个主程序类,说明当前是Spring Boot项目  
8  * @Description: com.mengxuegu  
9  * @Author: www.mengxuegu.com  
10 * @Version: 1.0  
11 */  
12 @SpringBootApplication  
13 public class HelloMailAppliation {  
14  
15     public static void main(String[] args) {  
16         SpringApplication.run(HelloMailAppliation.class, args);  
17     }  
18  
19 }  
20
```

## 2.4.5 运行效果

- 运行 引导类 后，会出现 如下一个标识，你能看出来下边这个图是什么东西？🤔



- 在浏览器地址栏输入 `localhost:8080/hello` 即可看到运行结果

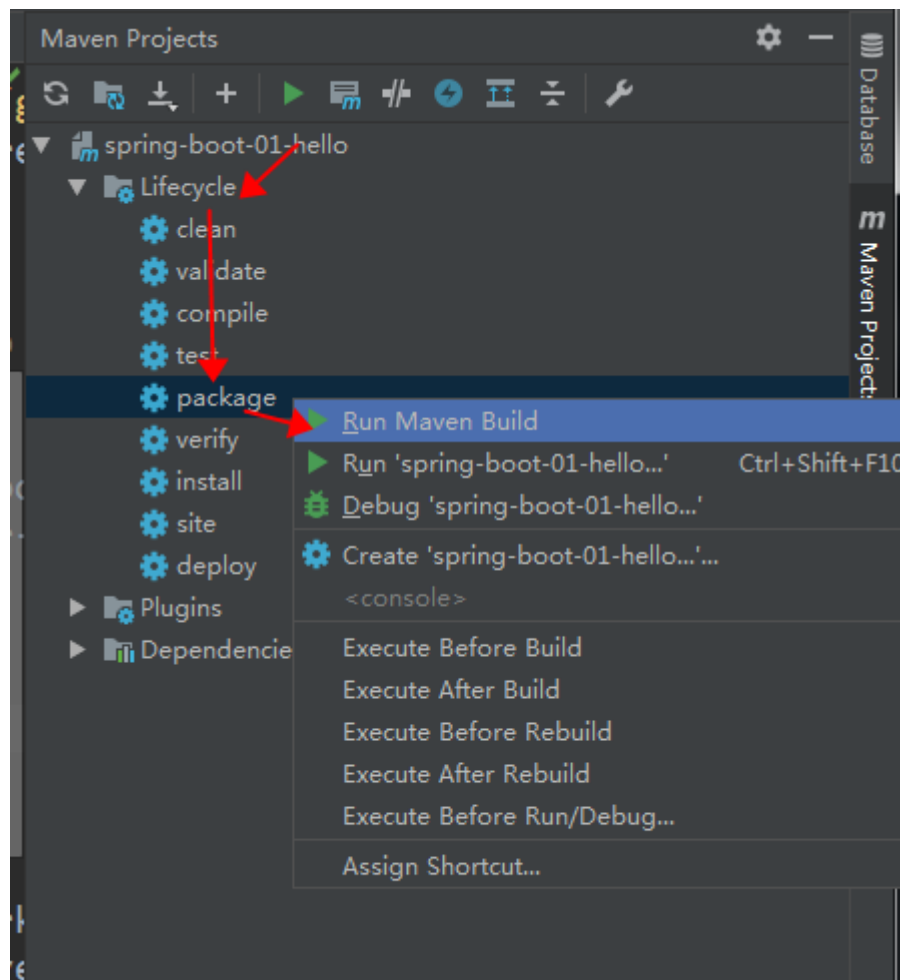


## 2.4.6 简化部署

- 在 pom.xml 添加如下插件后，将这个工程打成 jar 包后，可直接通过 `java -jar` 的命令运行

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.springframework.boot</groupId>
5       <artifactId>spring-boot-maven-plugin</artifactId>
6     </plugin>
7   </plugins>
8 </build>
```

- 如下操作进行打成 jar 包，从控制台可发现 打成的jar包所在目录



## 2.5 Spring Boot项目底层原理

### 2.5.1 pom.xml文件

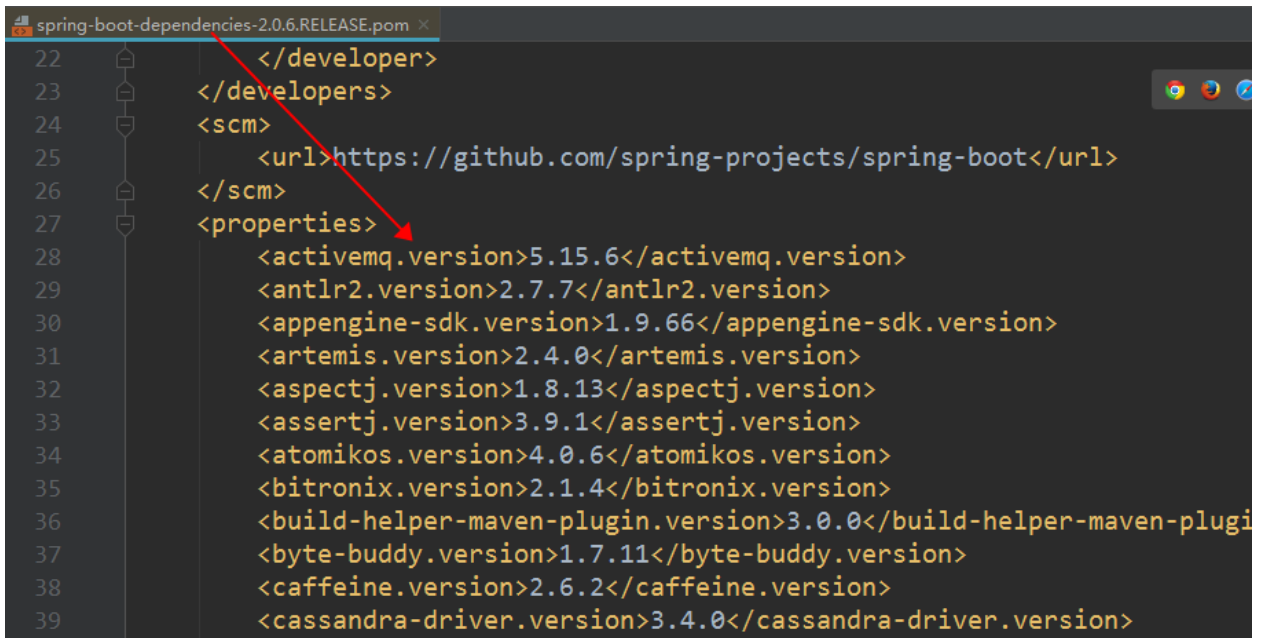
- spring-boot-starter-parent 是当前项目的父级依赖

```
1 当前hello项目的 父级依赖
2 <parent>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-parent</artifactId>
5   <version>2.0.6.RELEASE</version>
6 </parent>
```

- `spring-boot-starter` : Spring Boot 场景启动器，Spring Boot将所有的功能场景抽取出来，做成一个个的 starters ( 启动器 )，只需项目里引入相关场景的starter，就会将它所有依赖导入进来。要有什么功能就导入什么场景的启动器。（各种启动器可参见官方文档 starter）
- `spring-boot-starter-parent` : 它父依赖 `spring-boot-dependencies`，参见下面：

```
1 spring-boot-starter-parent 的 父级依赖
2 <parent>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-dependencies</artifactId>
5   <version>2.0.6.RELEASE</version>
6   <relativePath>../spring-boot-dependencies</relativePath>
7 </parent>
```

- `spring-boot-dependencies` 是管理了 Spring Boot项目中的所有依赖版本



```
spring-boot-dependencies-2.0.6.RELEASE.pom
22 </developer>
23 </developers>
24 <scm>
25   <url>https://github.com/spring-projects/spring-boot</url>
26 </scm>
27 <properties>
28   <activemq.version>5.15.6</activemq.version>
29   <antlr2.version>2.7.7</antlr2.version>
30   <appengine-sdk.version>1.9.66</appengine-sdk.version>
31   <artemis.version>2.4.0</artemis.version>
32   <aspectj.version>1.8.13</aspectj.version>
33   <assertj.version>3.9.1</assertj.version>
34   <atomikos.version>4.0.6</atomikos.version>
35   <bitronix.version>2.1.4</bitronix.version>
36   <build-helper-maven-plugin.version>3.0.0</build-helper-maven-plugin.version>
37   <byte-buddy.version>1.7.11</byte-buddy.version>
38   <caffeine.version>2.6.2</caffeine.version>
39   <cassandra-driver.version>3.4.0</cassandra-driver.version>
```

- 以后我们导入依赖默认不需要写版本号，也就是可以省去 version 标签；（当前没有在dependencies里面管理的依赖自然需要声明版本号）

## • spring-boot-starter-web

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

- 依赖导入了 Web 项目运行所依赖的组件；如 Tomcat / SpringMVC等

## 2.5.2 引导类

```
1  /**
2   * @SpringBootApplication 用于标识一个引导类,说明当前是Spring Boot项目
3   * @Author: www.mengxuegu.com
4   */
5   @SpringBootApplication
6   public class HelloMailAppliation {
7       public static void main(String[] args) {
8           SpringApplication.run(HelloMailAppliation.class, args);
9       }
10  }
```

- 通常有一个名为 \*Application 的入口类，里面定义一个main方法，使用 SpringApplication.run(HelloMailAppliation.class, args); 来启动 SpringBoot 应用项目
- @SpringBootApplication 注解主要组合了 @SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan
- @SpringBootApplication 注解说明：
  - 标注在某个类上，说明这个类是 Spring Boot 的引导类，Spring Boot 就应该运行这个类的main方法来启动 SpringBoot 应用；

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Documented
4  @Inherited
5  @SpringBootConfiguration
6  @EnableAutoConfiguration
7  @ComponentScan(excludeFilters = {
8      @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
9      @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
10 public @interface SpringBootApplication {
```

- @SpringBootApplication 是以下3个注解的总和：
  - @SpringBootConfiguration：用于定义一个Spring Boot的配置类(配置类等同配置文件)
    - 引用了 @Configuration 注解，是Spring底层的一个注解，用于定义 Spring 的配置类。
    - 配置类也是容器中的一个组件 @Component
  - @EnableAutoConfiguration：
    - 告诉Spring Boot开启自动配置功能，这样Spring Boot会自动根据你导入的依赖jar包来自动配置项目。

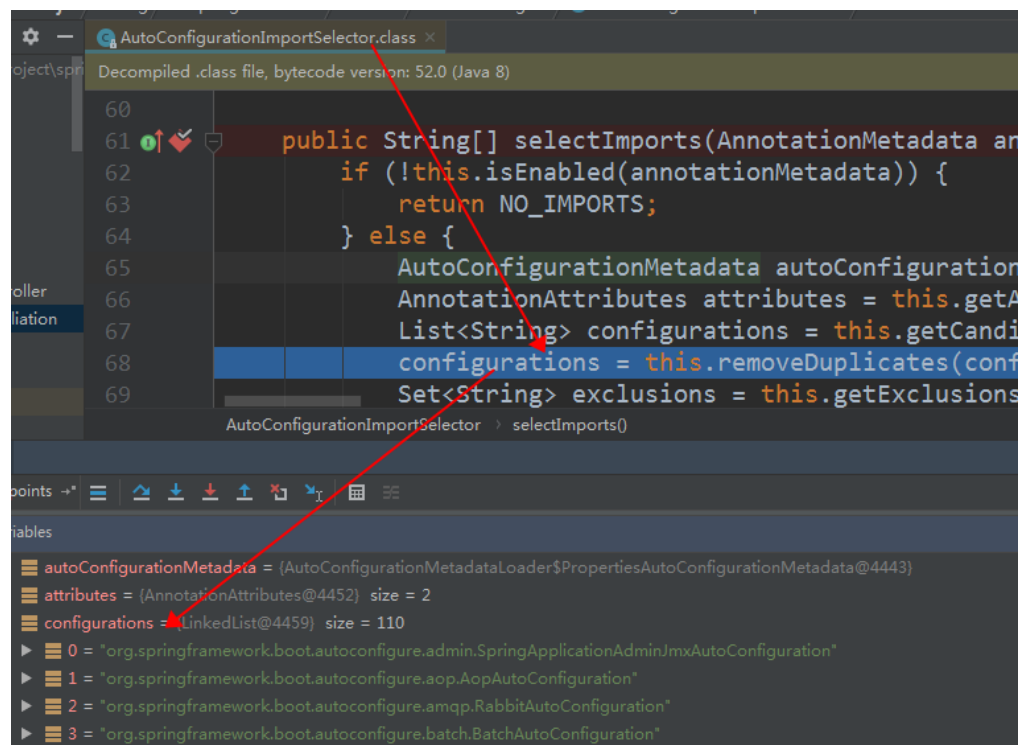
```
1 @AutoConfigurationPackage
2 @Import({AutoConfigurationImportSelector.class})
3 public @interface EnableAutoConfiguration {
```

■ @AutoConfigurationPackage ——》 @Import({Registrar.class})

- 会将引导类（@SpringBootApplication标注的类）所在的包及下面所有子包里面的所有组件扫描到Spring容器；

■ @Import({AutoConfigurationImportSelector.class})

- 将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中
- 会给容器导入非常多的自动配置类(xxxxAutoConfiguration),就是导入并配置好当前项目中所需要的组件,省去我们手动编写配置去注入组件.



- Spring Boot在启动的时候从(**spring-boot-autoconfigure-2.0.6.RELEASE.jar**)类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作；以前我们需要自己配置的文件，自动配置类都帮我们完成了；

```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    return (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyMap());
}

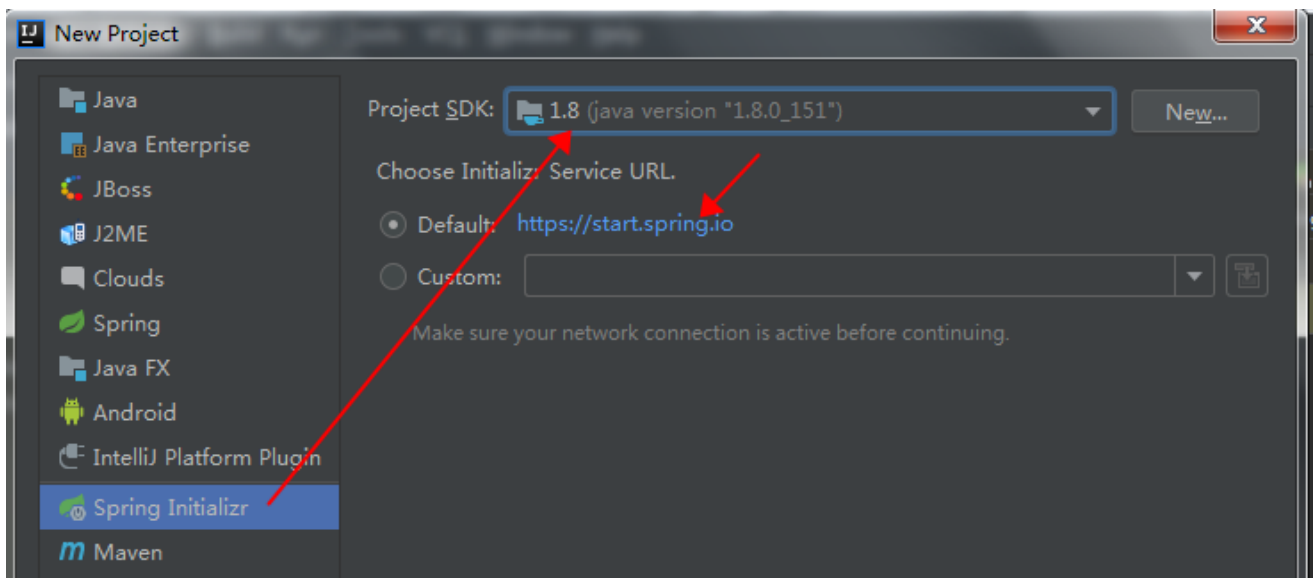
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
    MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
    if (result != null) {
        return result;
    } else {
        try {
            Enumeration<URL> urls = classLoader != null ? classLoader.getResources("META-INF/spring.factories") : null;
            MultiValueMap<String, String> result = new MultiValueMap<>();
            while (urls != null && urls.hasMoreElements()) {
                URL url = urls.nextElement();
                try {
                    InputStream is = url.openStream();
                    try {
                        Properties properties = new Properties(is);
                        String propertyName = properties.getProperty("org.springframework.boot.autoconfigure.EnableAutoConfiguration");
                        if (propertyName != null) {
                            String[] factories = properties.getProperty(propertyName).split(",");
                            for (String factory : factories) {
                                result.add(factory, url.toURI().toURL().toExternalForm());
                            }
                        }
                    } finally {
                        is.close();
                    }
                } catch (IOException ex) {
                    // ignore
                }
            }
        } catch (IOException ex) {
            // ignore
        }
    }
    return result;
}
```

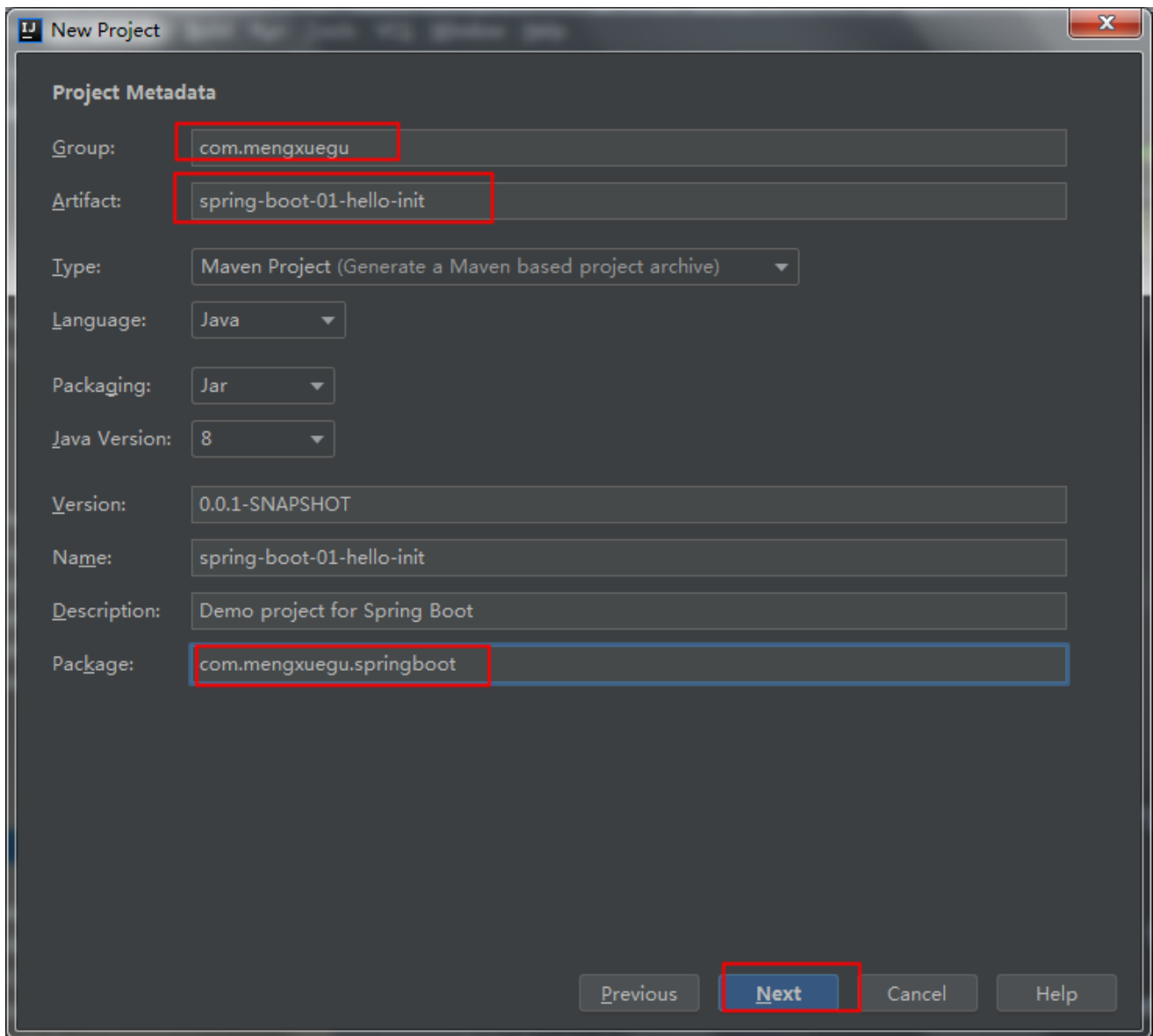
spring-boot-autoconfigure-2.0.6.RELEASE.jar library root  
META-INF  
additional-spring-configuration-metadata.json  
MANIFEST.MF  
spring.factories  
spring-autoconfigure-metadata.properties  
spring-configuration-metadata.json  
org  
aven: org.springframework.boot:spring-boot-starter:2.0.6.RELEASE

- **@ComponentScan**：该注解标识的类，会被 Spring 自动扫描并且装入bean容器。

## 2.6 使用Spring初始化器创建Spring Boot项目

- 注：初始化向导需要联网创建Spring Boot项目





The image shows a 'New Project' dialog box with a dark theme. It contains several input fields and dropdown menus for project configuration. The 'Group' field is set to 'com.mengxuegu', 'Artifact' to 'spring-boot-01-hello-init', 'Type' to 'Maven Project (Generate a Maven based project archive)', 'Language' to 'Java', 'Packaging' to 'Jar', 'Java Version' to '8', 'Version' to '0.0.1-SNAPSHOT', 'Name' to 'spring-boot-01-hello-init', 'Description' to 'Demo project for Spring Boot', and 'Package' to 'com.mengxuegu.springboot'. The 'Next' button at the bottom right is highlighted with a red border.

**New Project**

**Project Metadata**

Group: com.mengxuegu

Artifact: spring-boot-01-hello-init

Type: Maven Project (Generate a Maven based project archive)

Language: Java

Packaging: Jar

Java Version: 8

Version: 0.0.1-SNAPSHOT

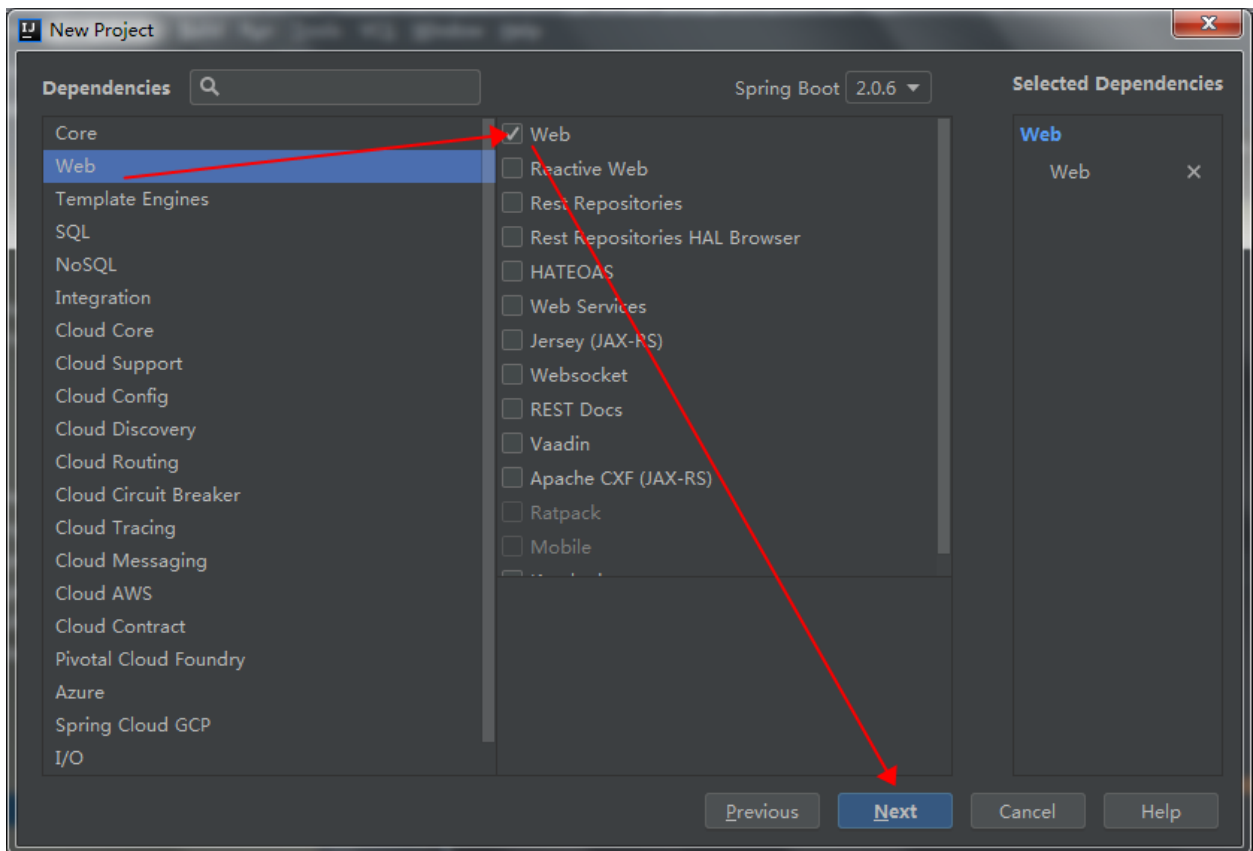
Name: spring-boot-01-hello-init

Description: Demo project for Spring Boot

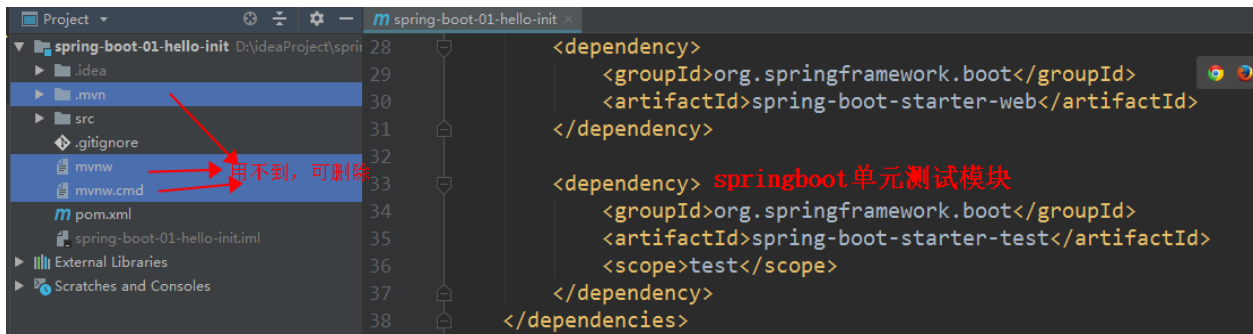
Package: com.mengxuegu.springboot

Previous **Next** Cancel Help

- 引入功能模块

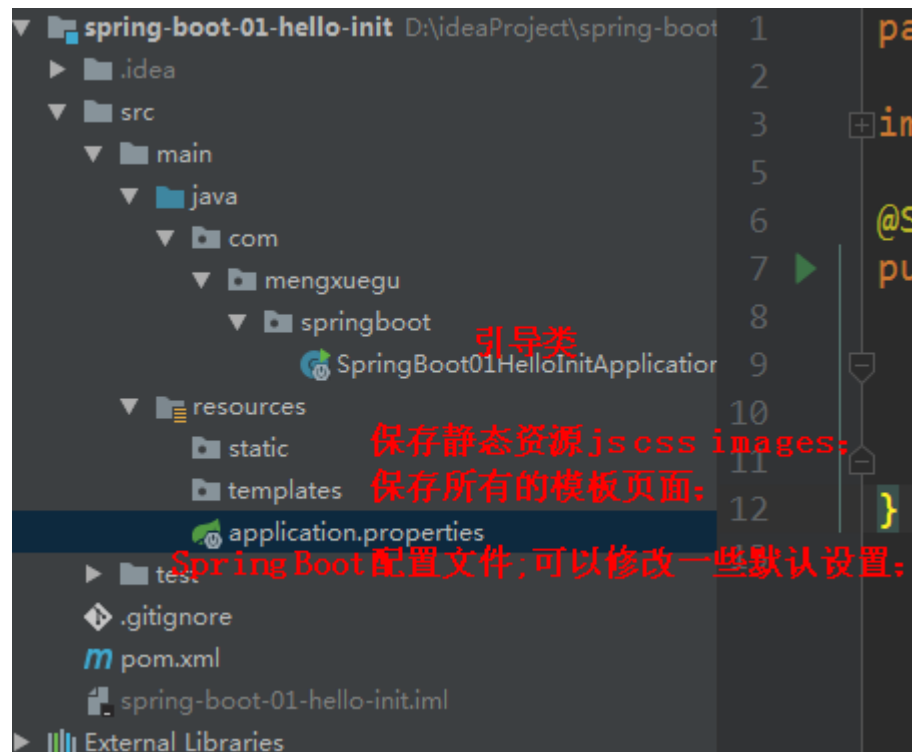


- 默认生成的Spring Boot项目；



- 删除不需要的文件与目录，`spring-boot-starter-test` 是 Spring Boot 单元测试模块
- 引导类已经生成了，我们只需要实现业务即可





- resources 文件夹中目录结构：
  - static：保存所有的静态资文件，js css images
  - templates：保存所有的模板页面（Spring Boot默认jar包使用嵌入式的Tomcat，默认不支持JSP页面），可以使用模板引擎（freemarker、thymeleaf）；
  - application.properties：Spring Boot应用的配置文件；可以修改一些默认设置；  
如修改默认端口 `server.port=8081`

## 第3章 Spring Boot 核心配置

### 3.1 Spring Boot的配置文件

- Spring Boot 使用一个全局配置文件，放置在 `src/main/resources` 目录或类路径的 `/config` 下；
  - application.properties
  - application.yml
- 配置文件的作用：修改 Spring Boot 自动配置的默认值；
- yml 是 YAML(YAML Ain't Markup Language)不是一个标记语言；
  - 标记语言：以前的配置文件；大多都使用的是 `xxxxx.xml` 文件；

```
1 <server>
2   <port>8081</port>
3 </server>
```

- YAML：以数据为中心，配置数据的时候具有面向对象的特征；比 json、xml 等更适合做配置文件；

```
1 server:  
2   port: 8081
```

## 3.2 YAML语法格式

### 3.2.1 YAML基本语法

- `key: value` 表示一对键值对 (冒号后面必须要有空格)
- 使用空格缩进表示层级关系
- 左侧缩进的空格数目不重要，只要同一层级的元素左侧对齐即可
- `key` 与 `value` 大小写敏感

```
1 server:  
2   port: 8081  
3   contextPath: /info
```

### 3.2.2 YMAL常用写法

- 字面量：数值，字符串，布尔，日期
  - 字符串 默认不用加上引号；
    - `""` 使用 双引号 不会转义特殊字符，特殊字符最终会转成本来想表示含义输出  
name: "mengxuegu \n jiaoyu" 输出： mengxuegu 换行 jiaoyu
    - `"` 使用 单引号 会转义特殊字符，特殊字符当作一个普通的字符串输出  
name: 'mengxuegu \n jiaoyu' 输出： mengxuegu \n jiaoyu
- 对象 & Map
  - `key: value` value存储对象，每个值换一行写，注意值要左对齐

```
1 emp:  
2   lastName: xiaomeng  
3   age: 22  
4   salary: 10000
```

- 行内写法：

```
1 emp: {lastName: xiaomeng age: 22 salary: 10000}
```

- 数组 (List、Set)
  - 用 `-` 值表示数组中的一个元素

```
1 fortes:
2   - java
3   - python
4   - hadoop
```

- 行内写法

```
1 fortes: [java python hadoop]
```

### 3.3 yaml 配置文件注入值

- 编写 JavaBean

```
1  /**
2   * 1、@ConfigurationProperties 告诉SpringBoot将配置文件中对应属性的值，映射到这个组件类中,进行——绑定
3   *   prefix = "emp": 配置文件中的前缀名，哪个前缀与下面的所有属性进行——映射
4   * 2、@Component 必须将当前组件作为SpringBoot中的一个组件，才能使用容器提供的
   @ConfigurationProperties功能；
5   *
6   * @Author: www.mengxuegu.com
7   */
8   @Component
9   @ConfigurationProperties(prefix = "emp")
10  public class Emp {
11      private String lastName;
12      private Integer age;
13      private Double salary;
14      private Boolean boss;
15      private Date birthday;
16
17      private Map map;
18      private List list;
19
20      //特长
21      private Forte forte;
22
23      getter/setter.....
24  }
25
26  public class Forte {
27      private String name;
28      private Integer time;
29
30      getter/setter.....
31  }
```

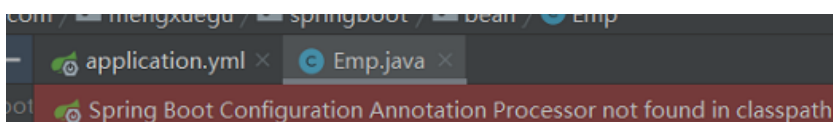
- 编写 application.yml 配置文件

```
1 emp:
2   lastName: zhangsan
3   age: 28
4   salary: 20000
5   boss: true
6   birthday: 1991/10/10
7   map:
8     key1: value1
9     key2: value2
10  list:
11    - one
12    - two
13    - three
14  forte:
15    name: java
16    time: 8
```

- 使用 SpringBoot 单元测试类进行测试

```
1 /**
2  * SpringBoot单元测试：
3  * 在测试时，可以直接将对象注入到容器中使用
4  */
5 @RunWith(SpringRunner.class)
6 @SpringBootTest
7 public class SpringBoot02ConfigApplicationTests {
8     @Autowired
9     Emp emp;
10
11     @Test
12     public void contextLoads() {
13         System.out.println(emp);
14     }
15 }
```

- 提示处理器没有发现：



在 pom.xml 导入配置文件处理器，然后重新运行测试类，在编写配置文件时就会提示（如果没有关闭打开的文件和重启idea）

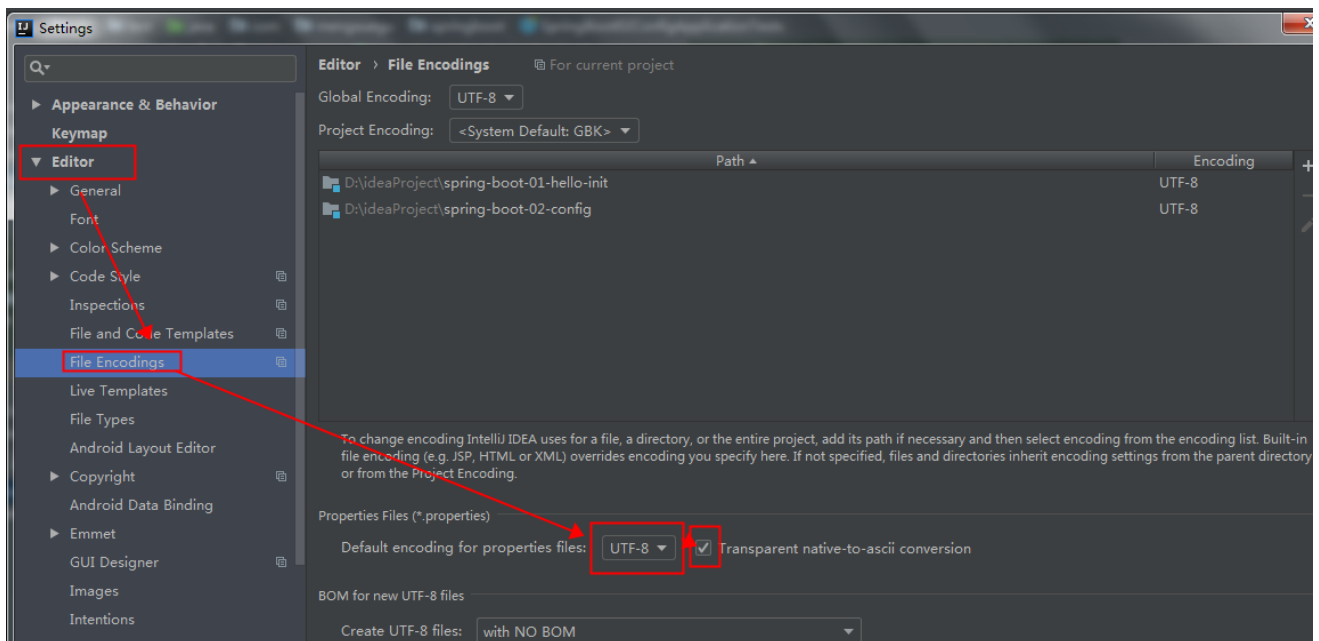
```
1 <!--导入配置文件处理器，在编写配置文件时就会提示-->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-configuration-processor</artifactId>
5   <optional>true</optional>
6 </dependency>
```

## 3.4 properties 配置文件注入值

- 将 application.yml 配置文件中的内容注释掉
- 编写 application.properties 配置文件

```
1 #配置emp的值
2 emp.last-name=李四
3 emp.age=30
4 emp.birthday=1989/9/12
5 emp.boss=false
6 emp.salary=23000
7 emp.map.key1=value1
8 emp.map.key2=value2
9 emp.list=one, two, three
10 emp.forte.name=python
11 emp.forte.time=3
```

- 运行后，发现 properties 文件在 idea 上中文乱码，进行如下设置就不会乱码



## 3.5 比较 @Value 和 @ConfigurationProperties 获取值

- 使用 @Value 获取值，把 Emp 类中的 @ConfigurationProperties 注释掉

```
1 @Component
2 //@ConfigurationProperties(prefix = "emp")
3 public class Emp {
4     /**
5      * 类似于 Spring 中的 xml 配制文件中的数据注入方式：
6      * <bean class="Emp">
```

```
7 * <property name="lastName" value="字面量/ #{SpEL}Spring表达式/ ${key}从配制文件取值">
8 * </property>
9 * </bean>
10 */
11 @Value("${emp.last-name}")
12 private String lastName;
13 @Value("#{10*2}")
14 private Integer age;
15 @Value("8000")
16 private Double salary;
17 private Boolean boss;
18 private Date birthday;
19
20 private Map map;
21 private List list;
22 //特长
23 private Forte forte;
```

- 总结 @Value 与 @ConfigurationProperties 获取值的差异

	@ConfigurationProperties	@Value	示例
实现功能	批量注入配置文件的属性值	一个一个指定	
松散绑定（松散语法）	支持	不支持	last_name == lastName last-name == lastName
SpEL	不支持	支持	<code>#{10*2}</code>
复杂类型封装	支持	不支持	<code>\${emp.map}</code>
JSR303数据校验	支持	不支持	如下3.6

## 3.6 JSR303数据校验\_配置文件注入的值

- 校验是否为合法的邮箱地址:
  - 取消 @ConfigurationProperties(prefix = "emp") 前面的注释
  - 在Emp 类上添加 @Validated 数据校验注解
  - 在 lastName 属性上添加 @Email 注解
  - 验证 @ConfigurationProperties 会进行校验, 而 @Value 不会进行校验值

```
1 // Email是这个包下面的类
2 import javax.validation.constraints.Email;
3
4 @Component
5 @ConfigurationProperties(prefix = "emp")
6 @Validated
7 public class Emp {
```

```
8
9  /**
10  * 类似于 Spring 中的 xml 配制文件中的数据注入方式：
11  * <bean class="Emp">
12  *   <property name="name" value="普通数据类型/ #{SpEL}Spring表达式/ ${key}从配制文件取值">
13  *   </property>
14  * </bean>
15  */
16 // @Value("${emp.last-name}")
17 @Email
18 private String lastName;
19 // @Value("#{10*2}")
20 private Integer age;
21 // @Value("8000")
22 private Double salary;
23 private Boolean boss;
24 private Date birthday;
25
26 private Map map;
27 private List list;
28 //特长
29 private Forte forte;
```

- 总结 使用场景:

- 如果只是在某个业务逻辑中需要获取配置文件中的某个属性值，就使用 `@Value`

```
1  @Controller
2  public class EmpController {
3
4      @Value("${emp.last-name}")
5      private String name;
6
7      @ResponseBody
8      @RequestMapping("/say")
9      public String sayHello() {
10         return "hello " + name;
11     }
12 }
```

- 如果专门使用javaBean和配置文件进行映射，就使用 `@ConfigurationProperties`

## 3.7 加载指定配置文件

### 3.7.1 `@PropertySource` 加载局部配置文件

@ConfigurationProperties 默认从全局配置文件(application.properties/application.yml)中获取值, 所有配置数据写在全局配置文件中, 显得太臃肿了, 可将它们抽取出来, 放到其他局部配置文件中。

- **@PropertySource** : 用于加载局部配置文件;

- 1. 将 全局配置文件 中的emp相关配置数据 抽取 到 resources/ emp.properties 文件中

```
1 emp.last-name=李四
2 emp.age=30
3 emp.birthday=1989/9/12
4 emp.boss=false
5 emp.salary=23000
6 emp.map.key1=value1
7 emp.map.key2=value2
8 emp.list=one, two, three
9 emp.forte.name=python
10 emp.forte.time=3
```

- 2. @PropertySource : 加载指定的配置文件; value 属性是数组类型, 用于指定文件位置

```
1
2 @PropertySource(value = {"classpath:emp.properties"})
3 @Component
4 @ConfigurationProperties(prefix = "emp")
5 @Validated
6 public class Emp {
7
8     /**
9      * 类似于 Spring 中的 xml 配制文件中的数据注入方式 :
10     * <bean class="Emp">
11     *   <property name="name" value="普通数据类型/ #{SpEL}Spring表达式/ ${key}从配制文件取值">
12     *   </property>
13     * </bean>
14     */
15     // @Value("${emp.last-name}")
16     // @Email
17     private String lastName;
18     // @Value("#{10*2}")
19     private Integer age;
20     // @Value("8000")
21     private Double salary;
22     private Boolean boss;
23     private Date birthday;
24
25     private Map map;
26     private List list;
27     //特长
28     private Forte forte;
29 }
```



### 3.7.2 @ImportResource 使用xml配置

- SpringBoot提倡零配置，即无xml配置，但是在实际开发中，可能有一些特殊要求必须使用 xml 配置；这时我们可以通过 Spring 提供的 @ImportResource 来加载 xml 配置文件。

- @ImportResource : 加载Spring的xml配置文件内容加载到容器中使用；

- 创建业务类：com.mengxuegu.springboot.service.EmpService

```
1 public class EmpService {  
2     public void add(){  
3     }  
}
```

- 创建 resources/spring01.xml 文件，添以下内容

```
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     xsi:schemaLocation="http://www.springframework.org/schema/beans  
5         http://www.springframework.org/schema/beans/spring-beans.xsd">  
6     <bean id="empService" class="com.mengxuegu.springboot.service.EmpService">  
7     </bean>  
8  
9 </beans>
```

- 将Spring的配置文件加载到容器中，使用 @ImportResource 标注在一个配置类上，下面是主配置类

```
1 @ImportResource(locations = {"classpath:spring01.xml"})  
2 @SpringBootApplication  
3 public class SpringBoot02ConfigApplication {  
4     public static void main(String[] args) {  
5         SpringApplication.run(SpringBoot02ConfigApplication.class, args);  
6     }  
7 }
```

- 单元测试

```
1 @RunWith(SpringRunner.class)  
2 @SpringBootTest  
3 public class SpringBoot02ConfigApplicationTests {  
4     @Autowired  
5     ApplicationContext context;  
6     @Test  
7     public void testXml(){  
8         //没有找到就报错  
9         System.out.println("empService: " + context.getBean("empService"));
```

```
10    }  
11  
12    @Autowired  
13    Emp emp;  
14    @Test  
15    public void contextLoads() {  
16        System.out.println(emp);  
17    }  
18 }  
19
```

### 3.7.3 自定义配置类向容器注入组件

- Spring Boot 推荐使用注解的方式向容器中注入组件，操作如下：
  - 使用 `@Configuration` 配置类，来表示对应Spring配置文件
  - 使用 `@Bean` 向容器中注入组件对象
  - 把上面 `@importResource` 注解注释掉测试

```
1  /**  
2   * @Configuration 用于标识当前类是一个配置类, 来表示对应的Spring配置文件  
3   * @Author: www.mengxuegu.com  
4   */  
5  @Configuration  
6  public class EmpConfig {  
7      /**  
8       * @Bean 标识的方法用于向容器注入组件  
9       * 1. 方法的返回值就是注入容器中的组件对象,  
10      * 2. 方法名是这个组件对象的 id 值  
11      */  
12      @Bean  
13      public EmpService empService2() {  
14          System.out.println("@Bean 注解已经将 EmpService 组件注入");  
15          return new EmpService();  
16      }  
17  }
```

- 再次测试

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest
3 public class SpringBoot02ConfigApplicationTests {
4     @Autowired
5     ApplicationContext context;
6     @Test
7     public void testXml(){
8         // System.out.println("empService: " + context.getBean("empService"));
9         System.out.println("empService2: " + context.getBean("empService2"));
10    }
11 }
```

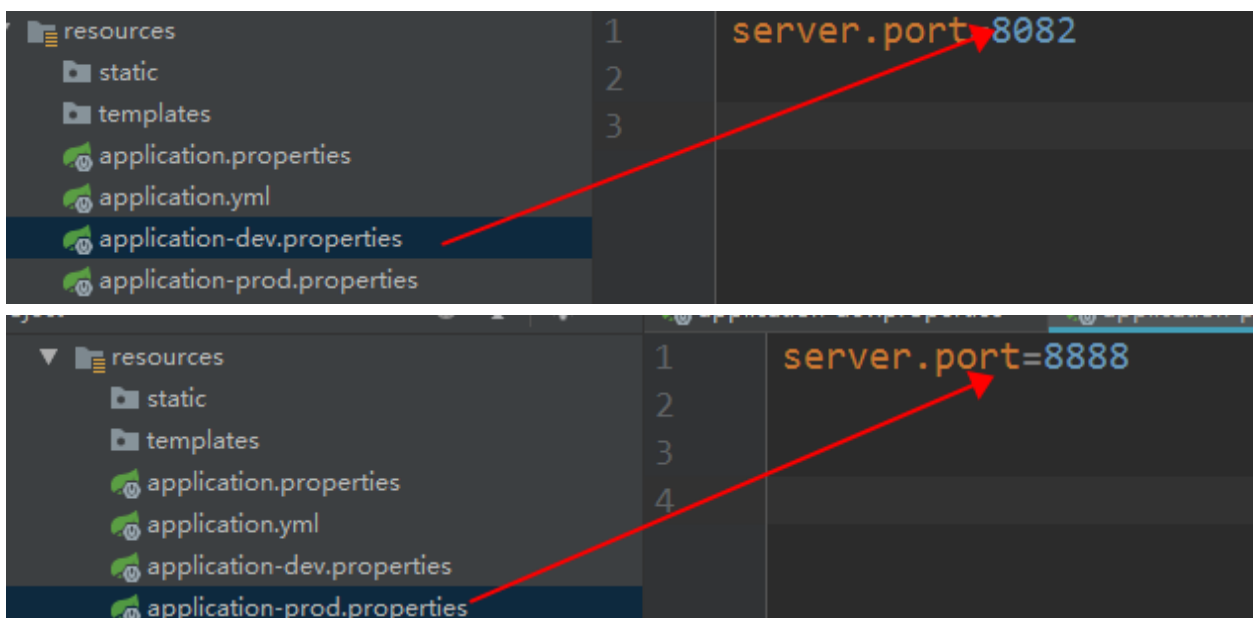
## 3.8 Profile 多环境支持

### 3.8.1 Profile介绍

- Profile 是 Spring 用来针对不同的环境要求，提供不同的配置支持，全局 Profile 配置使用的文件名可以是 `application-{profile}.properties` / `application-{profile}.yaml` ;
  - 如: `application-dev.properties` / `application-prod.properties`
- 演示案例：我们的项目环境分为 开发（dev）和生产（prod）环境，开发环境下端口号为 8082，生产环境下端口号为 8888。

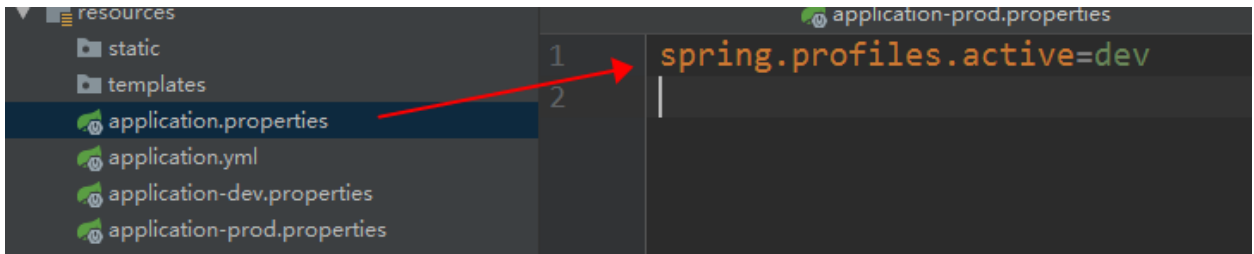
### 3.8.2 properties 文件演示案例

- 创建两个文件 `application-dev.properties` 与 `application-prod.properties`



- 激活指定profile

- 在主配置文件 `application.properties` 中指定 `spring.profiles.active=dev`



- 未指定哪个profile文件时, 默认使用 `application.properties` 中的配置。

### 3.8.3 yml 文件演示案例

- 在 `application.yml` 中配置
  - yml 支持多文档块方式 `---`

```
1 server:
2   port: 8081 # 默认端口号
3   spring:
4     profiles:
5       active: prod #激活哪个profile, 当前激活的是 dev 开发环境
6   ---
7   server:
8     port: 8082
9     spring:
10      profiles: dev #指定属于哪个环境, dev 环境时使用
11  ---
12  server:
13    port: 8888
14    spring:
15      profiles: prod #指定属于哪个环境, prod 环境时使用
```

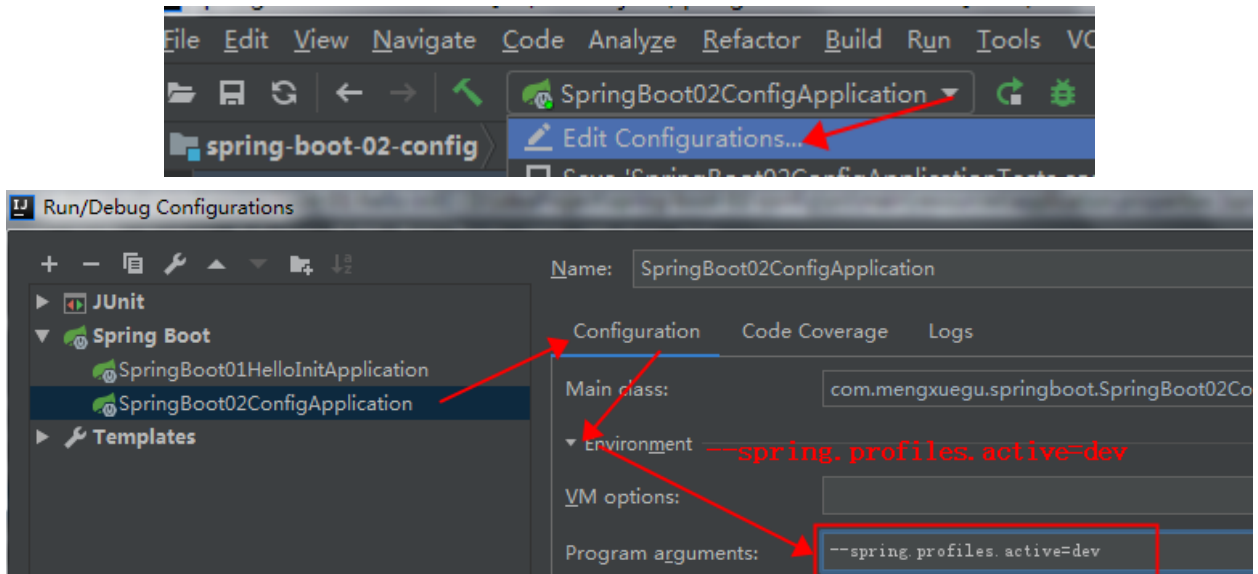
### 3.8.4 多种方式激活指定profile

- 方式1: 在主配置文件中指定

```
1 #application.properties
2 spring.profiles.active=dev
3
4 #application.yml
5 spring:
6   profiles:
7     active: prod
```

- 方式2: 命令行参数指定

- 可以直接在测试的时候, 配置传入命令行参数 `--spring.profiles.active=dev`

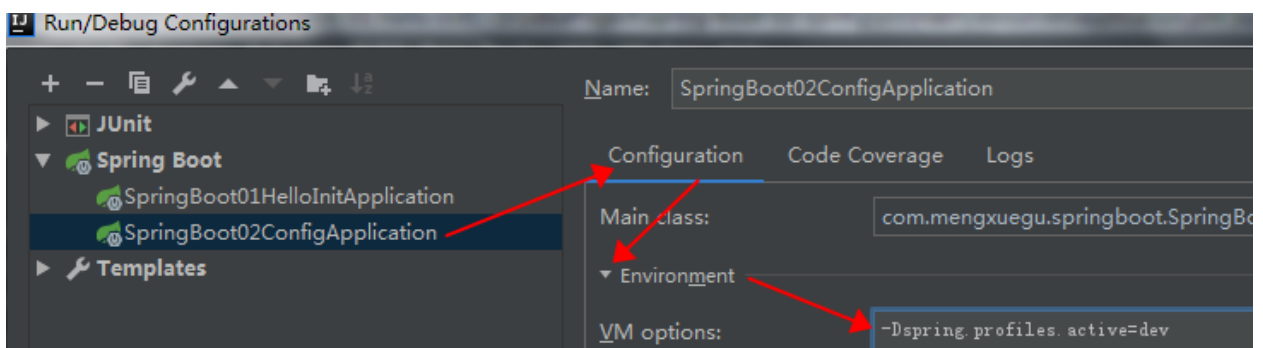


- 打jar包运行

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

- 方式3: 虚拟机参数指定

```
1 -Dspring.profiles.active=dev
```

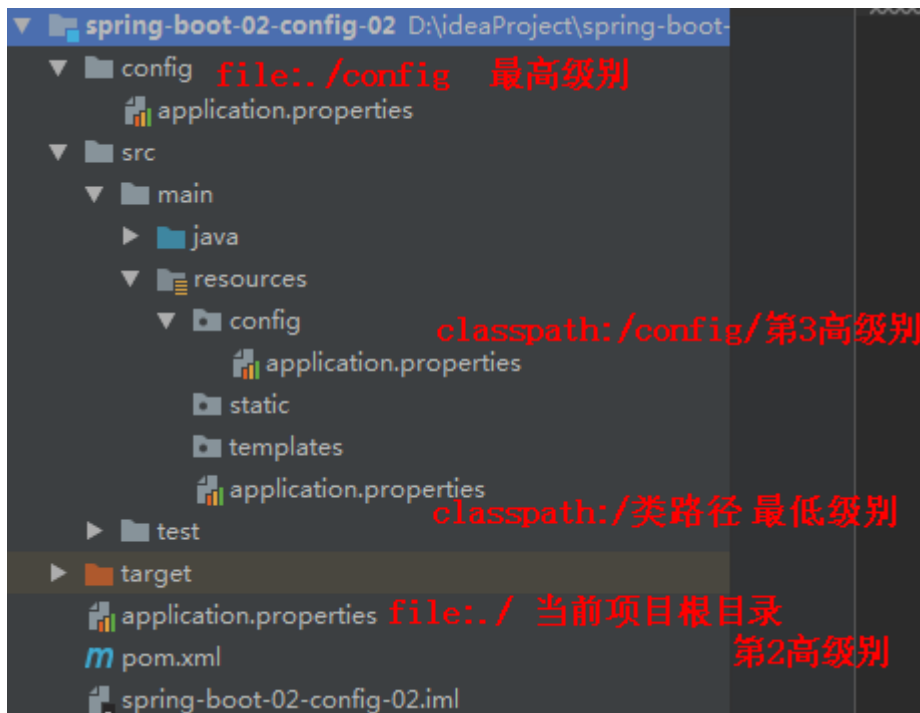


## 3.9 配置文件加载位置

- SpringBoot 启动时，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：

配置文件位置	说明
file:./config/	当前项目的config目录下（最高级别）
file:./	当前项目的根目录下
classpath:/config/	类路径的config目录下
classpath:/	类路径的根目录下（最低级别）

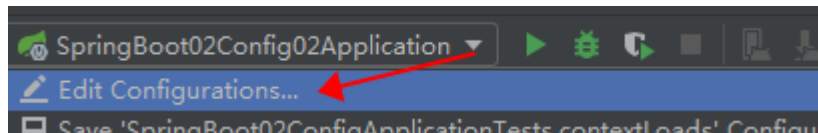
- 以上按照**优先级从低到高**的顺序，将所有位置的配置文件全部加载，**高优先级的配置内容会覆盖低优先级的配置内容**。
- 演示效果如下：

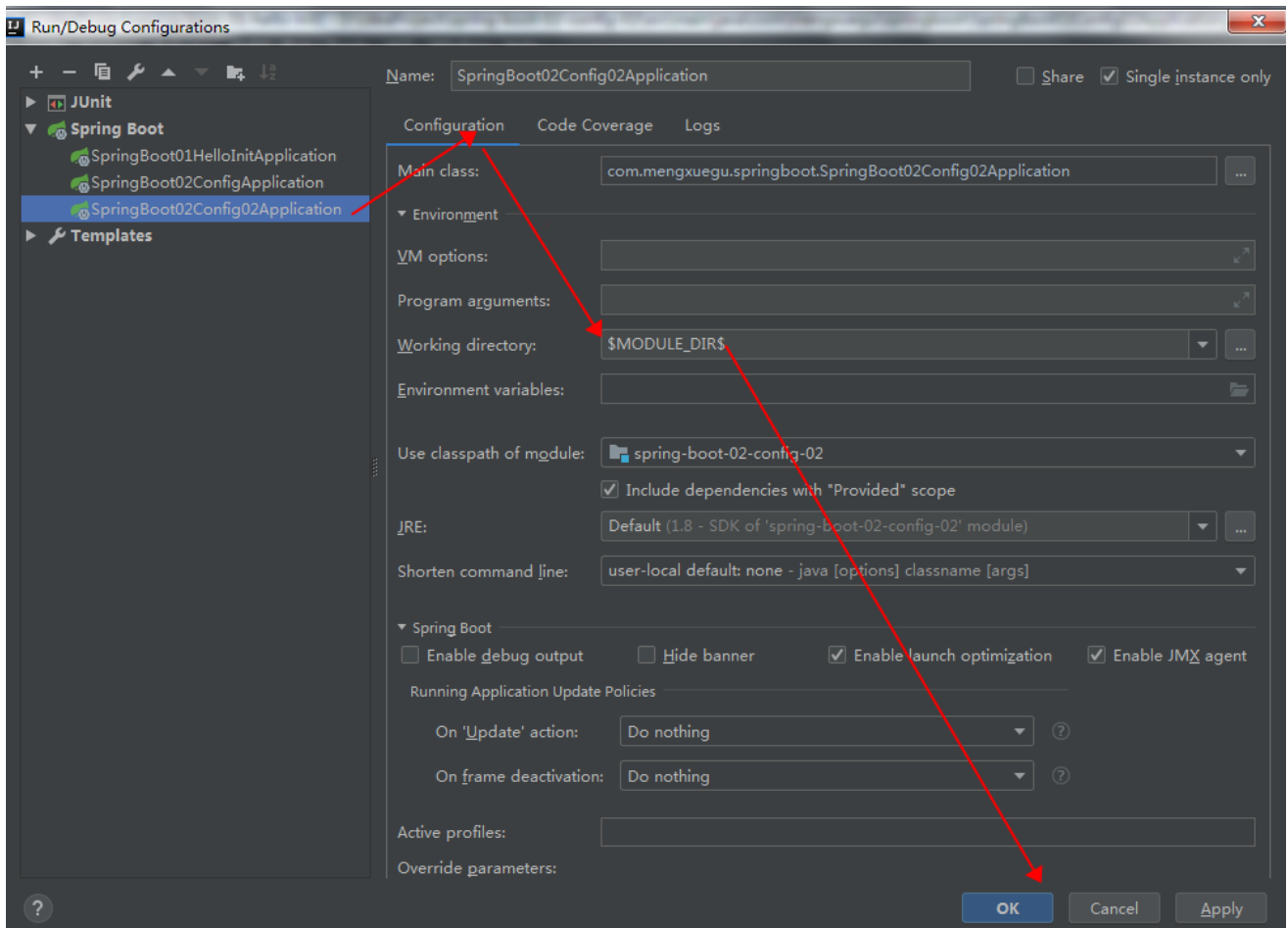


- **注意：如果使用IDEA创建的项目是 Module（如果是 Project 则忽略），当前项目的根目录不是你这个项目所有目录（是Project所在目录），这样使用 file: 存放配置文件时会找不到配置**

解决方式：更改工作路径直接为Module所有目录 `$MODULE_DIR$`

```
1 通过 System.getProperty("user.dir") 获取的是module的路径
```





- 配置文件到底能写什么？怎么写？

[配置文件能配置的属性参照](#)

<https://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/htmlsingle/#common-application-properties>

## 第4章 Spring Boot 日志配置

- 在市场上存在非常多的日志框架：

日志抽象层	日志实现
jboss-logging（不适合企业项目开发使用） JCL（Jakarta Commons Logging）（2014年后不再维护） <b>SLF4j</b> （Simple Logging Facade for Java）（与log4j Logback 同一个人开发）	JUL（java.util.logging）（担心被抢市场，推出的） Log4j（存在性能问题） <b>Logback</b> （Log4j同一个人开发的新框架，做了重大升级） Log4j2（apache开发的很强大，借了名log4j的名，但当前很多框架未适配上）

Spring Boot 采用了 **slf4j+logback** 的组合形式，Spring Boot也提供对JUL、log4j2、Logback提供了默认配置

- Spring官网参考文档:

<https://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/htmlsingle/#boot-features-logging>

## 4.1 默认日志配置

- SpringBoot默认配置好了日志，只要启动 Spring Boot 项目就会在控制台输出日志信息。

```
1 package com.mengxuegu.springboot;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.boot.test.context.SpringBootTest;
8 import org.springframework.test.context.junit4.SpringRunner;
9
10 @RunWith(SpringRunner.class)
11 @SpringBootTest
12 public class SpringBoot03LoggingApplicationTests {
13
14     //日志记录器
15     Logger logger = LoggerFactory.getLogger(getClass());
16
17     @Test
18     public void contextLoads() {
19         //1. 以下日志级别，由低到高：trace < debug < info < warn < error
20         //2. Spring Boot默认设定的是 info 级别日志，（日志默认级别也称为root级别）。
21         //   可修改默认级别日志：logging.level.level=级别名
22         //3. 可以进行调整日志级别，设定某个级别后，就只打印设定的这个级别及后面高级别的日志信息
23         //   没有指定级别的就用SpringBoot默认规定的级别：root级别
24         //4. 可修改指定包的日志级别：
25         //   指定某个包下面的所有日志级别：logging.level.包名=级别名
26
27         //跟踪运行信息
28         logger.trace("这是 trace 日志信息！");
29         //调试信息
30         logger.debug("这是 debug 日志信息！");
31
32         //自定义信息
```



```
31 logger.info("这是 info 日志信息");
32 //警告信息：如果运行结果是不预期的值，则可以进行警告
33 logger.warn("这是 warn 日志信息");
34 //错误信息：出现异常捕获时
35 logger.error("这是 error 日志信息");
36
37 }
38 }
39
```

- 修改日志默认级别

```
1 # 调整日志级别：trace < debug < info < warn < error
2 # com.mengxuegu包下的级别
3 logging.level.com.mengxuegu=trace
4
5 # 设置root级别
6 logging.level.root=debug
```

## 4.2 修改日志默认配置

`application.properties` 中修改日志默认配置

### 4.2.1 修改日志文件生成路径

logging.file	logging.path	示例	说明
(none)	(none)		只在控制台输出
指定文件名	(none)	springboot.log	输出到当前项目根路径下的 springboot.log 文件中
(none)	指定目录	/springboot/log	输出到当前项目所在磁盘根路径下的 /springboot/log目录中的 spring.log 文件中
指定文件名	指定目录		当两个同时指定时,采用的是 logging.file 指定。 推荐使用 <code>logging.file</code> 设置即可，因为它可自定义文件名

```
1 #输出到当前项目根路径下的 springboot.log 文件中
2 #logging.file=springboot.log
3
4 #输出到当前项目所在磁盘根路径下的 /springboot/log目录中的 spring.log 文件中,
5 logging.path=springboot/log
```

### 4.2.2 修改日志输出的格式

```
1 # 日志输出格式说明：
2 # %d 输出日期时间，
3 # %thread 输出当前线程名，
4 # %-5level 输出日志级别，左对齐5个字符宽度
5 # %logger{50} 输出全类名最长50个字符，超过按照句点分割
6 # %msg 日志信息
7 # %n 换行符
8
9 # 修改控制台输出的日志格式
10 logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %msg%n
11
12 # 修改文件中输出的日志格式
13 logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss.SSS} >>> [%thread] >>> %-5level >>> %logger{50} >>> %msg%n
```

## 4.3 分析日志底层实现

- 在web项目当中引用了 spring-boot-starter-web 依赖

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4 </dependency>
5
6 spring-boot-starter-web 中引入了 spring-boot-starter 启动器
7 <dependency>
8   <groupId>org.springframework.boot</groupId>
9   <artifactId>spring-boot-starter</artifactId>
10  <version>2.0.6.RELEASE</version>
11  <scope>compile</scope>
12 </dependency>
13
14 spring-boot-starter 中引入了 spring-boot-starter-logging 日志启动器
15 <dependency>
16   <groupId>org.springframework.boot</groupId>
17   <artifactId>spring-boot-starter-logging</artifactId>
18 </dependency>
19
20 spring-boot-starter-logging 日志启动器 采用的是 logback 日志框架
21 <dependency>
22   <groupId>ch.qos.logback</groupId>
23   <artifactId>logback-classic</artifactId>
24   <version>1.2.3</version>
25   <scope>compile</scope>
26 </dependency>
```

- 总结：SpringBoot中默认日志启动器为 spring-boot-starter-logging，默认采用的是 logback 日志框架

- 在 spring-boot-2.0.6.RELEASE.jar! \org\springframework\boot\logging\logback\base.xml 做了日志的默认配置

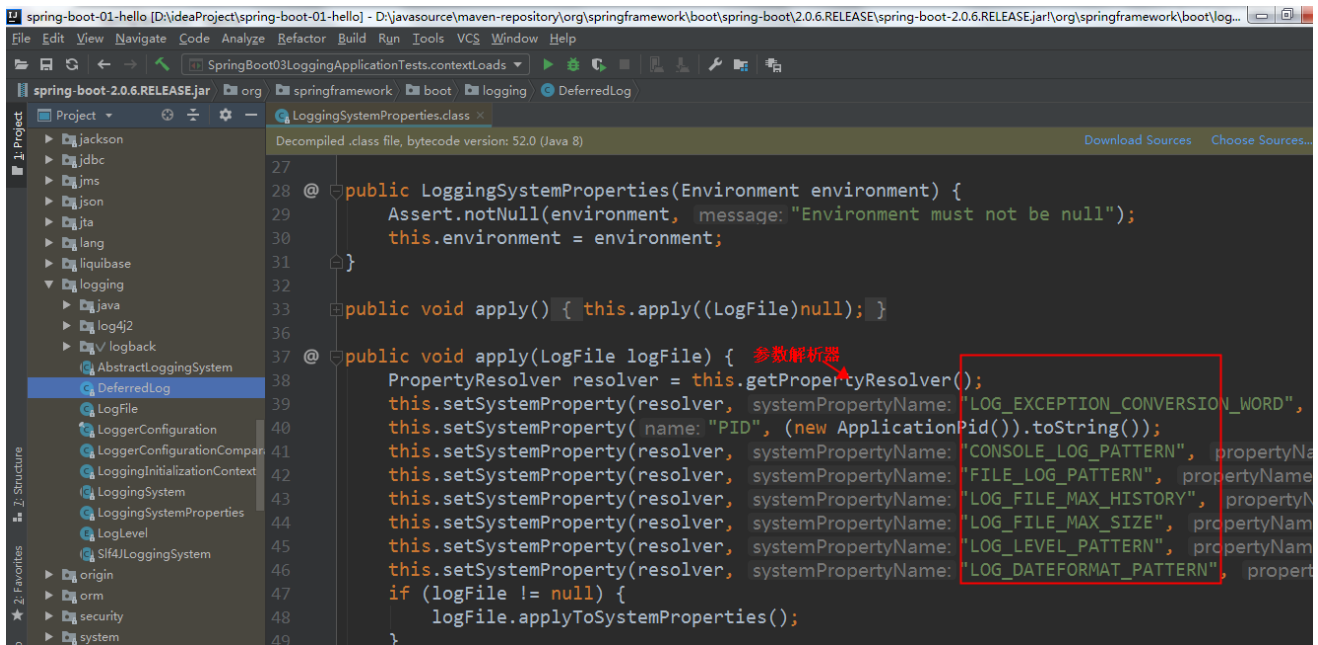
```
1 <included>
2 <!--日志格式默认规定-->
3 <include resource="org/springframework/boot/logging/logback/defaults.xml" />
4 <!--日志文件默认生成路径-->
5 <property name="LOG_FILE"
6 value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/tmp}}}/spring.log}" />
7 <!--控制台日志信息默认配置-->
8 <include resource="org/springframework/boot/logging/logback/console-appender.xml" />
9 <!--文件中日志信息默认配置-->
10 <include resource="org/springframework/boot/logging/logback/file-appender.xml" />
11 <!--日志级别默认为：info -->
12 <root level="INFO">
13 <appender-ref ref="CONSOLE" />
14 <appender-ref ref="FILE" />
15 </root>
16 </included>
```

- 日志文件采用方式为：滚动文件追加器



- 在下面类中会读取上面xml中配置的信息

spring-boot-2.0.6.RELEASE.jar!org.springframework.boot.logging.LoggingSystemProperties



- 如果spring boot的日志功能无法满足我们的需求(比如异步日志记录等)，我们可以自己定义的日志配置文件。

## 4.4 自定义日志配置

### 4.4.1 自定义Logback日志配置

- 在类路径下，存放对应日志框架的自定义配置文件即可；SpringBoot就不会使用它默认的日志配置文件了。

Logging System	自定义日志配置文件名
Logback	logback-spring.xml, logback-spring.groovy, logback.xml, or logback.groovy
Log4j2	log4j2-spring.xml or log4j2.xml
JDK (Java Util Logging)	logging.properties

- 在 `resources` 目录下创建 `logback.xml`，文件内容如下，SpringBoot就会采用以下日志配置：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!-- 梦学谷 www.mengxuegu.com
3
4 scan：当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。
5 scanPeriod：设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒当scan为true时，
   此属性生效。默认的时间间隔为1分钟。
6 debug：当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。
7 -->
8 <configuration scan="false" scanPeriod="60 seconds" debug="false">
9   <!-- 定义日志的根目录 -->
10   <property name="LOG_HOME" value="/logs/log" />
11   <!-- 定义日志文件名称 -->
12   <property name="appName" value="mengxuegu-spring-boot"></property>
13   <!-- ch.qos.logback.core.ConsoleAppender 表示控制台输出 -->
  
```

```
14 <appender name="stdout" class="ch.qos.logback.core.ConsoleAppender">
15   <!--
16     日志输出格式说明：
17       %d    输出日期时间
18       %thread 输出当前线程名
19       %-5level 输出日志级别，左对齐5个字符宽度
20       %logger{50} 输出全类名最长50个字符，超过按照句点分割
21       %msg    日志信息
22       %n      换行符
23   -->
24   <layout class="ch.qos.logback.classic.PatternLayout">
25     <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} - [%thread] - %-5level - %logger{50} - %msg%n</pattern>
26   </layout>
27 </appender>
28
29 <!-- 滚动记录文件，先将日志记录到指定文件，当符合某个条件时，将日志记录到其他文件 -->
30 <appender name="appLogAppender" class="ch.qos.logback.core.rolling.RollingFileAppender">
31   <!-- 指定日志文件的名称 -->
32   <file>${LOG_HOME}/${appName}.log</file>
33   <!--
34     当发生滚动时，决定 RollingFileAppender 的行为，涉及文件移动和重命名
35     TimeBasedRollingPolicy：最常用的滚动策略，它根据时间来制定滚动策略，既负责滚动也负责出发滚动。
36   -->
37   <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
38     <!--
39       滚动时产生的文件的存放位置及文件名称 %d{yyyy-MM-dd}：按天进行日志滚动
40       %i：当文件大小超过maxFileSize时，按照i进行文件滚动
41     -->
42     <fileNamePattern>${LOG_HOME}/${appName}-%d{yyyy-MM-dd}-%i.log</fileNamePattern>
43     <!--
44       可选节点，控制保留的归档文件的最大数量，超出数量就删除旧文件。
45       假设设置每天滚动，且maxHistory是365，则只保存最近365天的文件，删除之前的旧文件。
46       注意，删除旧文件是，那些为了归档而创建的目录也会被删除。
47     -->
48     <MaxHistory>365</MaxHistory>
49     <!--
50       当日志文件超过maxFileSize指定的大小是，根据上面提到的%i进行日志文件滚动 注意此处配置
51       SizeBasedTriggeringPolicy是无法实现按文件大小进行滚动的，必须配置timeBasedFileNamingAndTriggeringPolicy
52     -->
53     <timeBasedFileNamingAndTriggeringPolicy
54     class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
55       <maxFileSize>100MB</maxFileSize>
56     </timeBasedFileNamingAndTriggeringPolicy>
57   </rollingPolicy>
58   <!-- 日志输出格式： -->
59   <layout class="ch.qos.logback.classic.PatternLayout">
60     <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] [%-5level] [%logger{50}] : %line ] -
61     %msg%n</pattern>
62   </layout>
63 </appender>
64
65 <!--
66   logger主要用于存放日志对象，也可以定义日志类型、级别
67 -->
```

```
64     name : 表示匹配的logger类型前缀，也就是包的前半部分
65     level : 要记录的日志级别，包括 TRACE < DEBUG < INFO < WARN < ERROR
66     additivity : 作用在于children-logger是否使用 rootLogger配置的appender进行输出，
67     false : 表示只用当前logger的appender-ref，true :
68     表示当前logger的appender-ref和rootLogger的appender-ref都有效
69 -->
70 <!-- hibernate logger -->
71 <logger name="com.mengxuegu" level="debug" />
72 <!-- Spring framework logger -->
73 <logger name="org.springframework" level="debug" additivity="false"></logger>
74
75
76 <!--
77 root与logger是父子关系，没有特别定义则默认为root，任何一个类只会和一个logger对应，
78 要么是定义的logger，要么是root，判断的关键在于找到这个logger，然后判断这个logger的appender和level。
79 -->
80 <root level="info">
81     <appender-ref ref="stdout" />
82     <appender-ref ref="appLogAppender" />
83 </root>
84 </configuration>
```

logback.xml : 是直接就被日志框架加载了。

**logback-spring.xml** : 配置项不会被日志框架直接加载，而是由 SpringBoot 解析日志配置文件，进而可以使用 SpringBoot 的 Profile 特殊配置

## 4.4.2 使用 Profile 特殊配置

- 使用日志 Profile 特殊配置, 可根据不同的环境激活不同的日志配置

- 将 自定义日志配置文件名 logback.xml 改为 logback-spring.xml
- 修改 日志配置文件中 第25行，如下：

```
1 <springProfile name="dev">
2     <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} === [%thread] === %-5level ==== %logger{50} -
    %msg%n</pattern>
3 </springProfile>
4 <springProfile name="!dev">
5     <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} >>> [%thread] >>> %-5level >>> %logger{50} -
    %msg%n</pattern>
6 </springProfile>
```

- 指定运行环境：--spring.profiles.active=dev

如果使用 logback.xml 作为日志配置文件，还指定 Profile 特殊配置，则会有以下错误

```
1 ERROR in ch.qos.logback.core.joran.spi.Interpreter@28:40 - no applicable action for [springProfile], current
  ElementPath is [[configuration][appender][layout][springProfile]]
2 at
  org.springframework.boot.logging.logback.LogbackLoggingSystem.loadConfiguration(LogbackLoggingSystem.java:169)
3 .....
```

## 4.5 切换日志框架

- 将SpringBoot默认的 logback 切换为 log4j2 日志框架，[参考文档](#)

Table 13.3. Spring Boot technical starters

Name	Description	Pom
spring-boot-starter-jetty	Starter for using Jetty as the embedded servlet container. An alternative to <a href="#">spring-boot-starter-tomcat</a>	<a href="#">Pom</a>
spring-boot-starter-log4j2	Starter for using Log4j2 for logging. An alternative to <a href="#">spring-boot-starter-logging</a>	<a href="#">Pom</a>
spring-boot-starter-logging	Starter for logging using Logback. Default logging starter用的是 LogBack 框架，默认启动器	<a href="#">Pom</a>
spring-boot-starter-reactor-netty	Starter for using Reactor Netty as the embedded reactive HTTP server.	<a href="#">Pom</a>
spring-boot-starter-tomcat	Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by <a href="#">spring-boot-starter-web</a>	<a href="#">Pom</a>
spring-boot-starter-undertow	Starter for using Undertow as the embedded servlet container. An alternative to <a href="#">spring-boot-starter-tomcat</a>	<a href="#">Pom</a>

- 在项目的 pom.xml 切换log4j2

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   先排除 spring-boot-starter-logging 日志启动器
5   <exclusions>
6     <exclusion>
7       <groupId>org.springframework.boot</groupId>
8       <artifactId>spring-boot-starter-logging</artifactId>
9     </exclusion>
10  </exclusions>
11 </dependency>
12 再使用 log4j2 日志启动器
13 <dependency>
14   <groupId>org.springframework.boot</groupId>
15   <artifactId>spring-boot-starter-log4j2</artifactId>
16 </dependency>
```

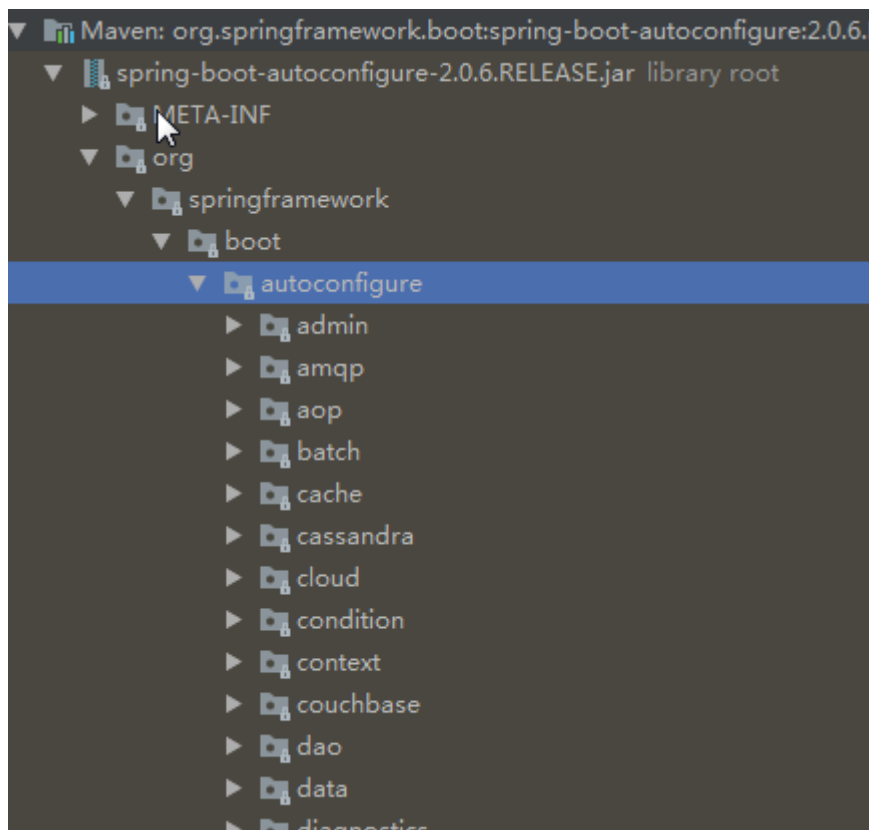


## 第5章 Spring Boot 的Web开发

- Web 开发是项目实战中至关重要的一部分，Web开发的核心内容主要包括嵌入的 Servlet 容器和 SpringMVC
- Web开发官方文档：  
<https://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/htmlsingle/#boot-features-spring-mvc>

### 5.1 Web开发支持

- Spring Boot 为 Web 开发提供了 `spring-boot-starter-web` 启动器作为基本支持，为我们提供了嵌入的Tomcat 以及 Spring MVC 的依赖支持。（参考：pom.xml）
- 也提供了很多不同场景的自动配置类，让我们只需要在配置文件中指定少量的配置即可启动项目。自动配置类存储在 `spring-boot-autoconfigure.jar` 的 `org.springframework.boot.autoconfigure` 包下。



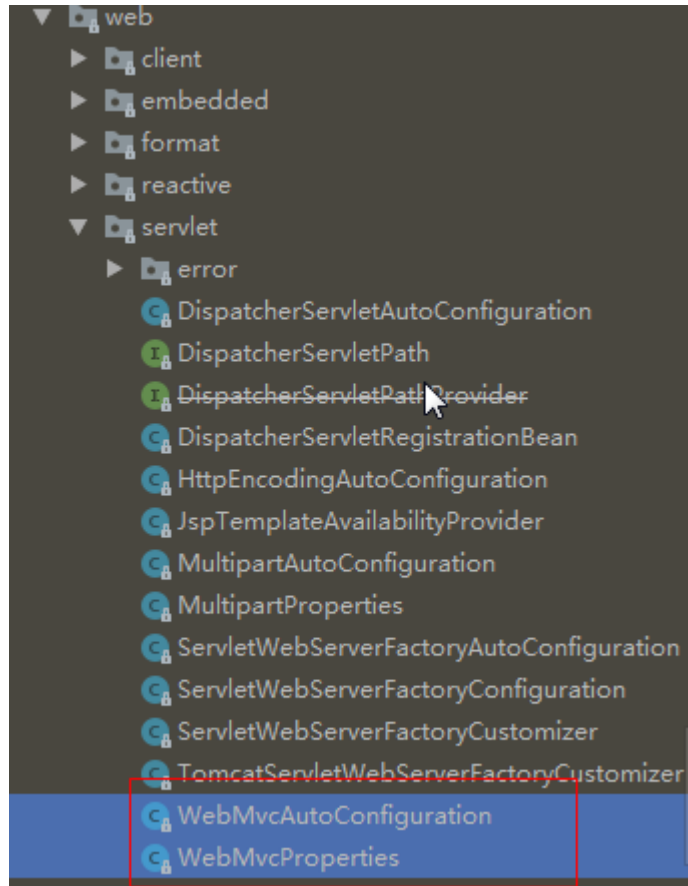
- **思考自动配置原理：**自动配置场景 SpringBoot 帮我们配置了什么？是否修改？能修改哪些配置？是否可以扩展？.....
- **自动配置类举例：**



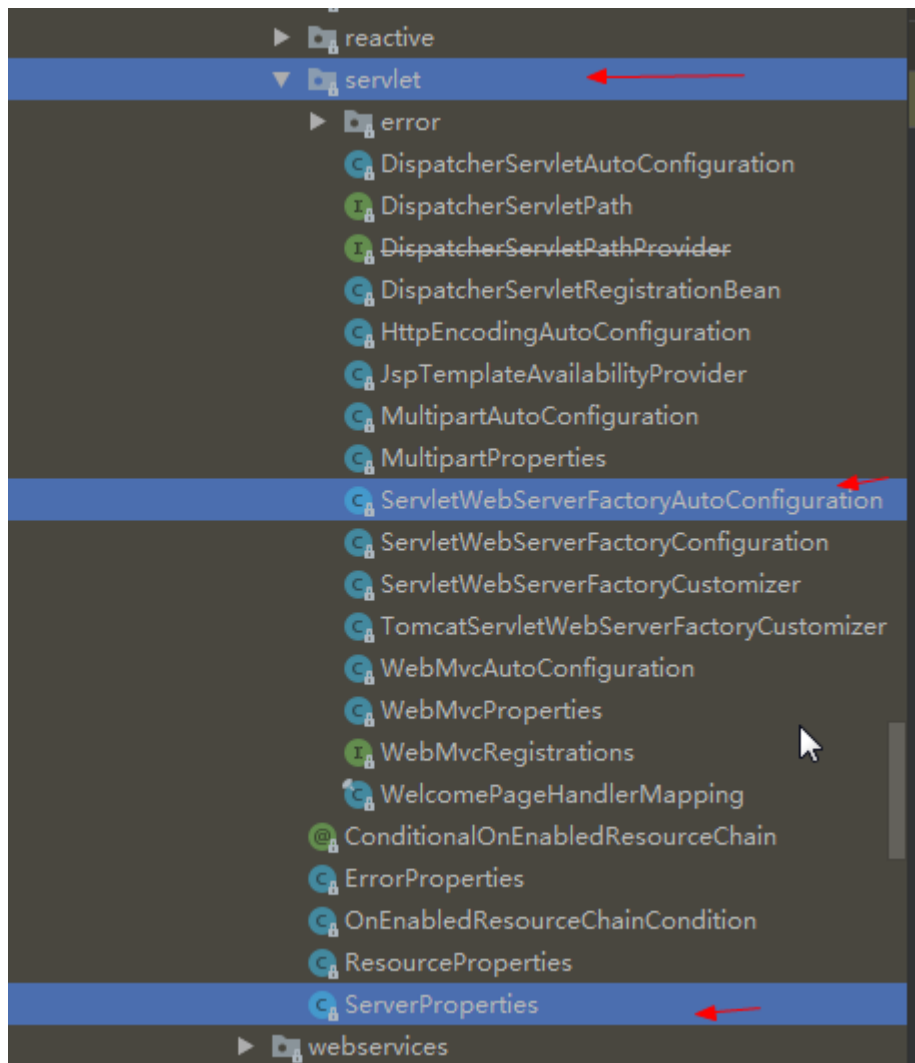
- 文件名可以看出

```
1 xxxxAutoConfiguration : 向容器中添加自动配置组件
2 xxxxProperties :使用自动配置类 来封装 配置文件的内容
```

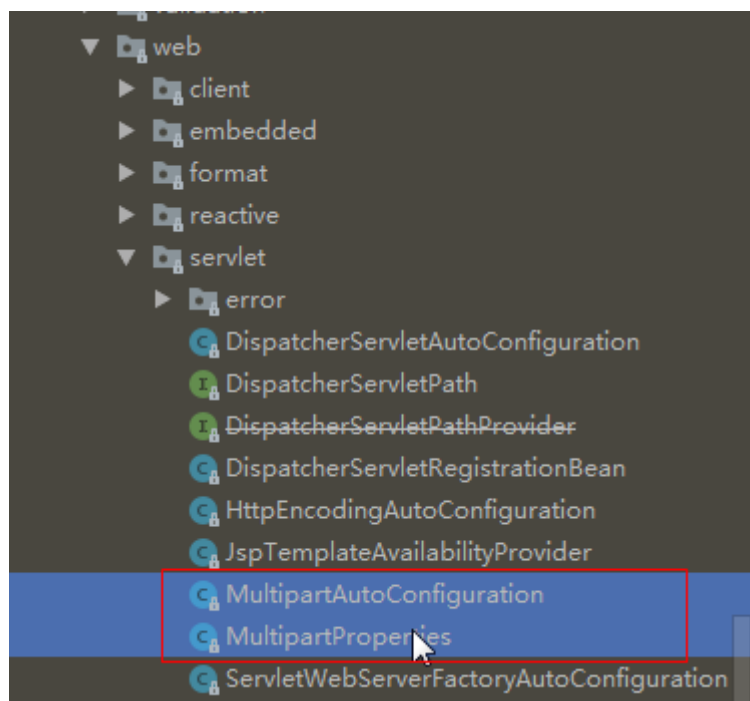
- SpringMVC配置** : WebMvcAutoConfiguration 和 WebMvcProperties



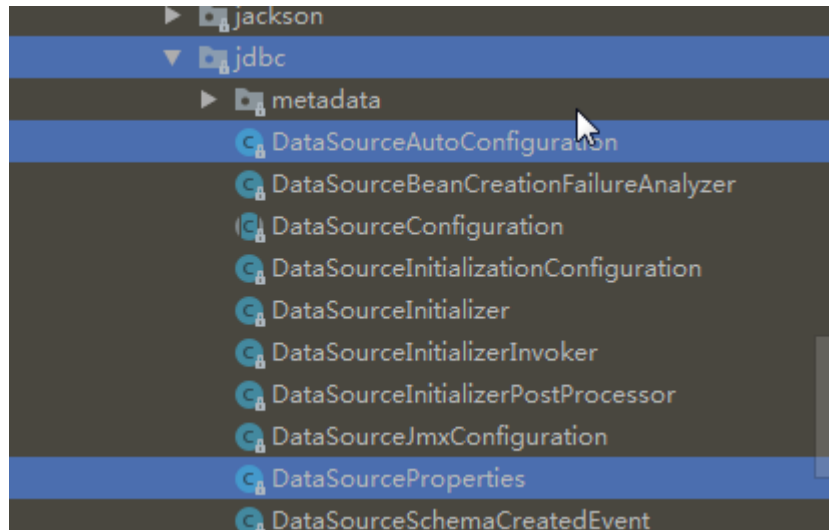
- 内嵌 Servlet 容器** : ServletWebServerFactoryAutoConfiguration 和 ServerProperties



- 上传文件的属性：MultipartAutoConfiguration 和 MultipartProperties



- JDBC：DataSourceAutoConfiguration 和 DataSourceProperties



o 等等.....

## 5.2 静态资源的映射规则

- 对静态资源的映射规则，可通过分析 WebMvcAutoConfiguration 自动配置类得到

### 5.2.1 webjars 资源映射

- 在 WebMvcAutoConfiguration.addResourceHandlers() 分析webjars 资源映射

```
1 public void addResourceHandlers(ResourceHandlerRegistry registry) {
2     if (this.resourceProperties.isAddMappings()) {
3         logger.debug("Default resource handling disabled");
4     } else {
5         Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
6         CacheControl cacheControl =
7             this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();
8         if (!registry.hasMappingForPattern("/webjars/**")) {
9             //收到 /webjars/**请求后，会去classpath:/META-INF/resources/webjars/ 查找资源文件
10            this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]
11                {"/webjars/**"}).addResourceLocations(new String[]{"classpath:/META-
12                INF/resources/webjars/"}).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));
13        }
14        String staticPathPattern = this.mvcProperties.getStaticPathPattern();
15        if (!registry.hasMappingForPattern(staticPathPattern)) {
16            this.customizeResourceHandlerRegistration(registry.addResourceHandler(new String[]
17                {staticPathPattern}).addResourceLocations(getResourceLocations(this.resourceProperties.getStaticLocations()))
18                .setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));
19        }
20    }
```

```
17     }  
18 }
```

1. 所有 `/webjars/**` 请求，都去 `classpath:/META-INF/resources/webjars/` 目录找对应资源文件

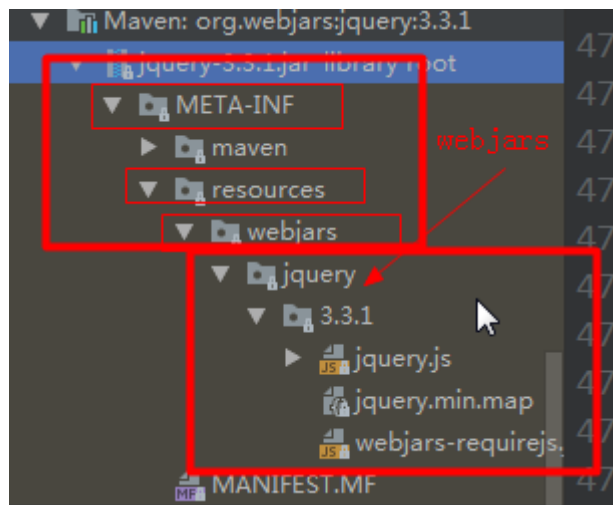
2. **webjars**：以jar包的方式引入静态资源

**webjars官网**：<https://www.webjars.org/>

3. 在官网打开资源文件的依赖配置信息，然后粘贴到 pom.xml 中

```
1 <!--引入 jquery webjars-->  
2 <dependency>  
3   <groupId>org.webjars</groupId>  
4   <artifactId>jquery</artifactId>  
5   <version>3.3.1</version>  
6 </dependency>
```

4. 访问 `localhost:8080/webjars/jquery/3.3.1/jquery.js` 会在下面路径 中查找



## 5.2.2 其他静态资源映射

- 在 `WebMvcAutotConfiguration.addResourceHandlers()` 分析 访问其他资源映射

```
1 public void addResourceHandlers(ResourceHandlerRegistry registry) {  
2     if (this.resourceProperties.isAddMappings()) {  
3         logger.debug("Default resource handling disabled");  
4     } else {  
5         Duration cachePeriod = this.resourceProperties.getCache().getPeriod();  
6         CacheControl cacheControl =  
7         this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();  
8         if (!registry.hasMappingForPattern("/webjars/**")) {  
9             this.customizeResourceHandlerRegistration(registry.addHandler(new String[]  
10                {"/webjars/**"}).addResourceLocations(new String[]{"classpath:/META-  
11                INF/resources/webjars/"}).setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));  
12         }  
13     }  
14 }  
15 // 接收/**
```

```
11     String staticPathPattern = this.mvcProperties.getStaticPathPattern();
12     if (!registry.hasMappingForPattern(staticPathPattern)) {
13         this.customizeResourceHandlerRegistration(registry.addHandler(new String[]
14             {staticPathPattern}).addResourceLocations(getResourceLocations(this.resourceProperties.getStaticLocations()))
15             .setCachePeriod(this.getSeconds(cachePeriod)).setCacheControl(cacheControl));
16     }
17 }
```

- staticPathPattern 处理其他访问的静态路径，从 WebMvcProperties 构造器中获取到 /\*\*

```
1 public WebMvcProperties() {
2     this.localeResolver = WebMvcProperties.LocaleResolver.ACCEPT_HEADER;
3     this.dispatchTraceRequest = false;
4     this.dispatchOptionsRequest = true;
5     this.ignoreDefaultModelOnRedirect = true;
6     this.throwExceptionIfNoHandlerFound = false;
7     this.logResolvedException = false;
8     =====接收 /**请求
9     this.staticPathPattern = "/**";
10    this.async = new WebMvcProperties.Async();
11    this.servlet = new WebMvcProperties.Servlet();
12    this.view = new WebMvcProperties.View();
13    this.contentnegotiation = new WebMvcProperties.Contentnegotiation();
14    this.pathmatch = new WebMvcProperties.Pathmatch();
15 }
```

- ResourceProperties 根据请求查找资源文件, 从以下 四个路径 中 查找( 静态资源目录 )

```
1 @ConfigurationProperties(
2     prefix = "spring.resources",
3     ignoreUnknownFields = false
4 )
5 public class ResourceProperties {
6     private static final String[] CLASSPATH_RESOURCE_LOCATIONS = new String[]{"classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/"};
7     private String[] staticLocations;
8     private boolean addMappings;
9     private final ResourceProperties.Chain chain;
10    private final ResourceProperties.Cache cache;
11 }
```

```
1 "classpath:/META-INF/resources/",
2 "classpath:/resources/",
3 "classpath:/static/",
4 "classpath:/public/"
```

- 总结:

- 当接受到 `/**` 请求访问资源时, 会被映射到下面4个 类路径下的静态资源目录中 查找

```
1 classpath: META-INF/resources/  
2 classpath: resources/  
3 classpath: /static/  
4 classpath: /public/
```

- 访问 `localhost:8080/style.css` 会在上面四个静态资源路径 中查找文件

### 5.2.3 欢迎页映射

- 在 `WebMvcAutotConfiguration.welcomePageHandlerMapping()` 分析 欢迎页映射

```
1 @Bean  
2 public WelcomePageHandlerMapping welcomePageHandlerMapping(ApplicationContext applicationContext) {  
3     return new WelcomePageHandlerMapping(new TemplateAvailabilityProviders(  
4         applicationContext), applicationContext  
5         =====查找欢迎页=====  
6         , this.getWelcomePage()  
7         , this.mvcProperties.getStaticPathPattern());  
8 }
```

- `getWelcomePage()` 方法获取 欢迎页面 可存储路径

```
1 private Optional<Resource> getWelcomePage() {  
2     String[] locations =  
3         ===2. 上面说的4个静态资源路径加上 "/" 路径  
4         getResourceLocations(  
5         =====1. 获取上面说的4个静态资源路径  
6         this.resourceProperties.getStaticLocations());  
7         =====在上面路径下查找 index.html 页面  
8     return Arrays.stream(locations).map(this::getIndexHtml).  
9         filter(this::isReadable).findFirst();  
10 }  
11  
12 // 上面获取的路径中查找 index.html 页面  
13 private Resource getIndexHtml(String location) {  
14     return this.resourceLoader.getResource(location + "index.html");  
15 }
```

- 分析后, 会从 4个静态资源目录 + 根路径 `/` 中 查找 `index.html` 页面

```
1 classpath: META-INF/resources/  
2 classpath: resources/  
3 classpath: /static/  
4 classpath: /public/  
5 /: 当前项目根路径下
```

- 会在 静态资源目录下 与 根路径查找 (按该顺序) index.html页面；收到 "/" 请求映射
- 访问 localhost:8080/ 会在上面5个目录中查找 index.html 页面(因为/也属于 /\*\* )

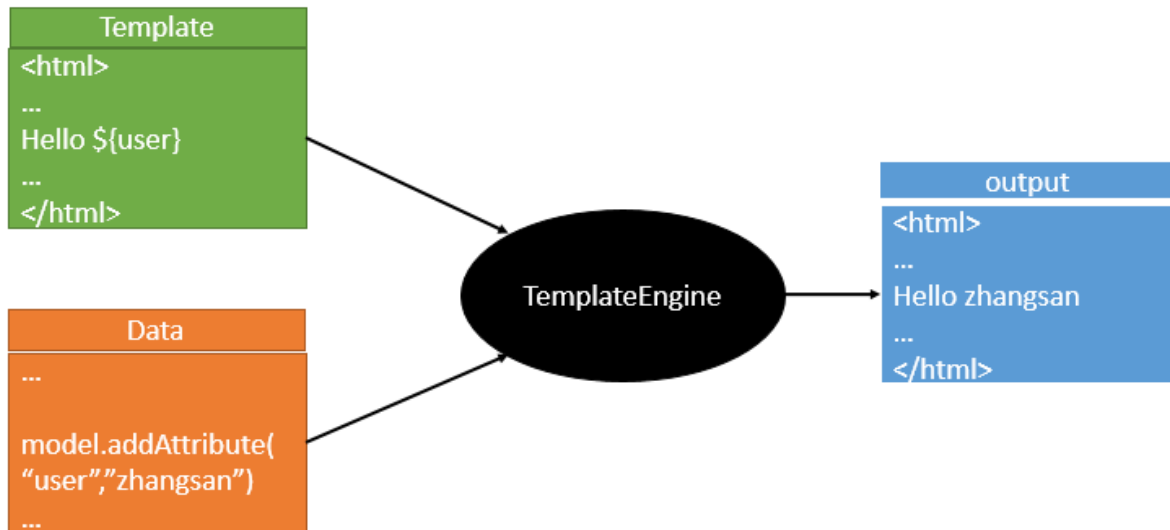
## 5.2.4 图标映射

- Spring Boot 会在静态资源目录下 与 根路径(按该顺序) 查找 favicon.ico 页面；  
如果存在这样的文件，Spring Boot 会自动将其设置为应用图标。

```
1 classpath: META-INF/resources/  
2 classpath: resources/  
3 classpath: /static/  
4 classpath: /public/  
5 /: 当前项目根路径下
```

## 5.3 Thymeleaf 模板引擎

Spring Boot 官方不推荐使用JSP，因为内嵌的 Tomcat、Jetty 容器不支持以 jar 形式运行 JSP。Spring Boot 中提供了大量模板引擎，包含 Freemarker、Mastache、Thymeleaf 等。而 Spring Boot 官方推荐使用 Thymeleaf 作为模板引擎，因为 Thymeleaf 提供了完美的 SpringMVC 的支持。



### 5.3.1 引入 Thymeleaf

- pom.xml 加入 Thymeleaf 启动器

```
1 <!-- thymeleaf 模板启动器 -->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-thymeleaf</artifactId>
5 </dependency>
```

### 5.3.2 使用 Thymeleaf

- 模板文件放在哪里？

```
1 @ConfigurationProperties( prefix = "spring.thymeleaf" )
2 public class ThymeleafProperties {
3   private static final Charset DEFAULT_ENCODING;
4   public static final String DEFAULT_PREFIX = "classpath:/templates/";
5   public static final String DEFAULT_SUFFIX = ".html";
6 }
```

- 通过上面分析发现，将 HTML 页面放到 **classpath:/templates/** 目录下，Thymeleaf 就能自动渲染

```
1 @RequestMapping("/execute")
2 public String execute(Map<String, Object> map) {
3   map.put("name", "梦学谷");
4   // classpath:/templates/success.html
5   return "success";
6 }
```

- 发送 <http://localhost:8080/execute> 后，通过上面代码转到 classpath:/templates/success.html



- 导入 Thymeleaf 的名称空间

在 html 页面加上以下名称空间, 使用 Thymeleaf 时就有语法提示。

```
1 <html xmlns:th="http://www.thymeleaf.org">
```

- 演示 Thymeleaf 语法

```
1 <!DOCTYPE html>
2 <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="UTF-8">
5   <title>hello</title>
6 </head>
7 <body>
8   <h2>成功</h2>
9   <!--th:text 设置p标签的标签体内容-->
10  <p th:text="${name}">这里显示名字</p>
11 </body>
12 </html>
```

## 5.3.3 Thymeleaf 语法

### 5.3.3.1 常用属性

- 参考 Thymeleaf 官方文档 [10 Attribute Precedence](#)

优先级	属性名	作用
1	th:insert th:replace	引入片段，与th:fragment声明组合使用；类似于jsp:include
2	th:each	遍历，类似于c:forEach
3	th:if th:unless th:switch th:case	条件判断，类似于c:if
4	th:object th:with	声明变量，类似于c:set
5	th:attr th:attrprepend th:attrappend	修改任意属性，prepend前面追加，append后面追加
6	th:value th:href th:src ...	<b>修改任意html原生属性值</b>
7	th:text th:utext	修改标签体中的内容， th:text 转义特殊字符，即 h1 标签以文本显示出来 th:utext 是不转义特殊字符，即 h1 标签展现出本来效果
8	th:fragment	声明片段
9	th:remove	移除片段

### 5.3.3.2 标准表达式语法

- 参考 Thymeleaf 官方文档 [4 Standard Expression Syntax](#)

```

1  一、Simple expressions ( 表达式语法 )
2  1. Variable Expressions(变量表达式): ${...} ( 参考： 4.2 Variables )
3    1)、获取变量值；使用OGNL表达式；
4    2)、获取对象的属性, 调用方法
5    3)、使用内置的基本对象：
6      #ctx : the context object.(当前上下文对象)
7      #vars: the context variables.(当前上下文里的变量)
8      #locale : the context locale. (当前上下文里的区域信息)
9      下面是Web环境下的隐式对象
10     #request : (only in Web Contexts) the HttpServletRequest object.
11     #response : (only in Web Contexts) the HttpServletResponse object.
12     #session : (only in Web Contexts) the HttpSession object.
13     #servletContext : (only in Web Contexts) the ServletContext object.
14     示例: ${session.foo} (用法参考: 18 Appendix A: Expression Basic Objects)
15
  
```

```
16 4)、使用内置的工具对象:(用法参考: 19 Appendix B: Expression Utility Objects)
17 #execInfo : information about the template being processed.
18 #messages : methods for obtaining externalized messages inside variables expressions, in the same
    way as they would be obtained using #{...} syntax.
19 #uris : methods for escaping parts of URLs/URIs
20 #conversions : methods for executing the configured conversion service (if any).
21 #dates : methods for java.util.Date objects: formatting, component extraction, etc.
22 #calendars : analogous to #dates , but for java.util.Calendar objects.
23 #numbers : methods for formatting numeric objects.
24 #strings : methods for String objects: contains, startsWith, prepending/appending, etc.
25 #objects : methods for objects in general.
26 #booleans : methods for boolean evaluation.
27 #arrays : methods for arrays.
28 #lists : methods for lists.
29 #sets : methods for sets.
30 #maps : methods for maps.
31 #aggregates : methods for creating aggregates on arrays or collections.
32 #ids : methods for dealing with id attributes that might be repeated (for example, as a result of an
    iteration).
33
34 2. Selection Variable Expressions(选择表达式): *{...}
35 (参考 : 4.3 Expressions on selections )
36 1)、和${}在功能上是一样，额外新增：配合 th:object 使用
37 <div th:object="${session.user}">
38     省得每次写${session.user.firstName}, 直接取出对象，然后写对象名即可
39     <p>Name: <span th:text="*{firstName}">Sebastian</span> </p>
40     <p>Email: <span th:text="*{email}">Saturn</span> </p>
41 </div>
42 3. Message Expressions ( 获取国际化内容 ) : #{...} (参考 : 4.1 Messages )
43 4. Link URL Expressions ( 定义URL ) : @{...} (参考 : 4.4 Link URLs )
44 5. Fragment Expressions ( 片段引用表达式 ) : ~{...} (参考 : 4.5 Fragments )
45 <div th:insert="~{commons :: main}">...</div>
46
47 二、Literals ( 字面量 ) (参考 : 4.6 Literals )
48 1. Text literals: 'one text', 'Another one!' ,...
49 2. Number literals: 0 , 34 , 3.0 , 12.3 ,...
50 3. Boolean literals: true , false
51 4. Null literal: null
52 5. Literal tokens: one , sometext , main ,...
53
54 三、Text operations ( 文本操作 ) (参考 : 4.7 Appending texts )
55 1. String concatenation: +
56 2. Literal substitutions: |The name is ${name}|
57
58 四、Arithmetic operations ( 数学运算 ) (参考 : 4.9 Arithmetic operations )
59 1. Binary operators: +, -, *, /, %
60 2. Minus sign (unary operator): -
61
62 五、Boolean operations ( 布尔运算 )
63 1. Binary operators: and , or
64 2. Boolean negation (unary operator): !, not
65
66 五、Comparisons and equality ( 比较运算 ) (参考 : 4.10 Comparators and Equality )
```

```
67 1. Comparators: >, <, >=, <= ( gt, lt, ge, le )
68 2. Equality operators: ==, != ( eq, ne )
69
70 六、Conditional operators(条件表达式;三元运算符) ( 参考 : 4.11 Conditional expressions )
71 1. If-then: (if) ? (then)
72 2. If-then-else: (if) ? (then) : (else)
73 3. Default: (value) ?: (defaultvalue)
74
75 七、Special tokens ( 特殊操作 ) (参考 : 4.13 The No-Operation token)
76 1. No-Operation: _
77
```

## 5.3.4 实例代码演示

### 5.3.4.1 声明与引入公共片段

```
1 <!--header.html-->
2 <body>
3   <!--声明公共片段-->
4   <!-- 方式1 : -->
5   <div th:fragment="header_common">
6     这是th:fragment声明公共片段
7   </div>
8   <!-- 方式2 : 选择器写法-->
9   <div id="header_common_id">
10    这是id选择器声明公共片段
11  </div>
12 </body>
13
14 <!-- success.html 引入头部公共片段 -->
15 <!--方式1 :
16   header : 公共片段所在模板的文件名
17   header_common : 声明代码片段名 -->
18 <div th:replace="header :: header_common"></div>
19 <!--方式2 : 选择器写法
20   header : 公共片段所在模板的文件名
21   #header_common_id: 声明代码片的id值
22 -->
23 <div th:replace="header :: #header_common_id"></div>
24 <!--
25   th:insert 和 th:replace的区别
26   th:insert和th:replace都可以引入片段，两者的区别在于
27   th:insert : 保留引入时使用的标签
28   th:replace : 不保留引入时使用的标签, 将声明片段直接覆盖当前引用标签
29 -->
30 <h2 th:insert="header :: #header_common_id"></h2>
```

练习：将项目中的 公共模块抽取出来到 public.html 中

### 5.3.4.2 迭代 th:each

- 常用迭代方式

- HelloController

```
1 @RequestMapping("/study")
2 public String study(Map<String, Object> map, HttpServletRequest request) {
3     List<User> userList = new ArrayList<>();
4     userList.add(new User("小梦", 1));
5     userList.add(new User("小李", 2));
6     userList.add(new User("小张", 1));
7     map.put("userList", userList);
8
9     return "study";
10 }
```

- study.html

```
1 <table border="1px">
2     <tr>
3         <th>姓名</th>
4     </tr>
5     <!--方式1 : -->
6     <tr th:each="user : ${userList}">
7         <!--每次迭代都会生成一个当前标签-->
8         <td th:text="${user}">mengxuegu</td>
9     </tr>
10 </table>
11
12 <hr/>
13 <ul>
14     <!--方式2 : -->
15     <!--作用在同一个标签上, 每次迭代生成一个当前标签-->
16     <li th:each="user : ${userList}" th:text="${user}"></li>
17 </ul>
```

- 获取迭代状态

```
1 <table border="1px">
2     <tr>
3         <th>编号</th>
4         <th>姓名</th>
5         <th>总数</th>
6         <th>偶数/奇数</th>
7         <th>第一个元素</th>
8         <th>最后一个元素</th>
9     </tr>
10     <!--
11     user : 第1个值,代表每次迭代出对象,名字任意取
12
13     iterStat : 第2个值,代表每次迭代器内置对象,名字任意取, 并有如下属性:
```

```
13      index : 当前迭代下标 0 开始
14      count : 当前迭代下标 1 开始
15      size : 获取总记录数
16      current : 当前迭代出的对象
17      even/odd : 当前迭代是偶数还是奇数 (1开始算,返回布尔值)
18      first : 当前是否为第一个元素
19      last : 当前是否为最后一个元素
20      -->
21      <tr th:each="user, iterStat : ${userList}">
22          <td th:text="${iterStat.count}">0</td>
23          <td th:text="${user.username}">mengxuegu</td>
24          <td th:text="${user.gender == 1 ? '女' : '男'}">未知</td>
25          <td th:text="${iterStat.size}">0</td>
26          <td th:text="${iterStat.even}? '偶数' : '奇数'"></td>
27          <td th:text="${iterStat.first}"></td>
28          <td th:text="${iterStat.last}"></td>
29      </tr>
30 </table>
31
```

- 练习 : 供应商管理 查询页面

### 5.3.4.3 条件判断

- th:if 不仅判断返回为 true 的表达式, 还判断一些特殊的表达式。

- 如果值不是Null, 以下情况均返回 true :

- 如果值是boolean类型并且值为true.
- 如果值是数值类型并且值不为0.
- 如果值是字符类型并且值不为空.
- 如果值是字符串并且内容不为 "false", "off" 或者 "no".
- 如果值不是上述类型也返回true.

- 如果值是NULL, 则返回false

```
1 <hr/>
2 下面加not
3 <h3 th:if="not ${#lists.isEmpty(userList)}">th:if判断,如果此文字显示说明有价值</h3>
4 <h3 th:unless="${#lists.isEmpty(userList)}">th:unless判断,如果此文字显示说明有价值</h3>
5
```

- th:unless 与 th:if 作用正好相反。
- th:switch th:case

```
1 @RequestMapping("/study")
2 public String study(Map<String, Object> map, HttpServletRequest request) {
3     List<User> userList = new ArrayList<>();
4     userList.add(new User("小梦", 1));
5     userList.add(new User("小李", 2));
6     userList.add(new User("小张", 1));
```

```
7 map.put("userList", userList);
8
9 // 1女, 2男
10 map.put("sex", 1);
11 map.put("man", 2);
12
13 return "study";
14 }
```

```
1 <div th:switch="${sex}">
2   <!-- 1女, 2男 -->
3   <p th:case="1">女</p>
4   <!-- 判断sex的值和下面取出man的值是否相等, 相等则显示 -->
5   <p th:case="${man}">男</p>
6   <!-- 如果值都不在上述case里, 则th:case="*"语句生效。 -->
7   <p th:case="*">未知</p>
8 </div>
```

#### 5.3.4.4 显示标签体内容

- **th:text** 转义特殊字符, 即 h1 标签以文本显示出来
- **th:utext** 不转义特殊字符, 即 h1 标签展现出本来效果

```
1 @RequestMapping("/study")
2 public String study(Map<String, Object> map, HttpServletRequest request) {
3     List<User> userList = new ArrayList<>();
4     userList.add(new User("小梦", 1));
5     userList.add(new User("小李", 2));
6     userList.add(new User("小张", 1));
7     map.put("userList", userList);
8
9     // 1女, 2男
10    map.put("sex", 1);
11    map.put("man", 2);
12
13    // th:text th:utext
14    map.put("desc", "欢迎来到<h1>梦学谷</h1>");
15
16    return "study";
17 }
```

```
1 <hr/>
2 <div th:text="${desc}"> </div>
3 <div th:utext="${desc}"> </div>
```

- 补充: Thymeleaf 行内表达式双中括号: `[[表达式]]` (就是不在标签上使用属性, 参考12 Inlining)

```
1 <input type="checkbox" /> [[${desc}]]
2 <p>Hello, [[${desc}]]。。。</p>
```

### 5.3.4.5 th:object 直接取出对象

- 使用th:object 直接取出对象，然后写对象里的属性名即可获取属性值

```
1 @RequestMapping("/study")
2 public String study(Map<String, Object> map, HttpServletRequest request) {
3     List<User> userList = new ArrayList<>();
4     userList.add(new User("小梦", 1));
5     userList.add(new User("小李", 2));
6     userList.add(new User("小张", 1));
7     map.put("userList", userList);
8
9     // 1女, 2男
10    map.put("sex", 1);
11    map.put("man", 2);
12
13    // th:text th:utext
14    map.put("desc", "欢迎来到<h1>梦学谷</h1>");
15
16    request.getSession().setAttribute("user", new User("小不点", 2));
17
18    return "study";
19 }
```

```
1 <!--使用th:object 直接取出对象，然后写对象里的属性名即可获取属性值-->
2 <div th:object="${session.user}">
3     <p>
4         姓名:<span th:text="*{username}">xxxx</span>
5     </p>
6     <p>
7         性别:<span th:text="*{gender == 1 ? '女' : '男'}">xxxx</span>
8     </p>
9 </div>
```

## 5.4 SpringBoot 热部署

- 默认情况下，在开发中我们修改一个项目文件后，想看到效果不得不重启应用，这会导致浪费大量时间，我们希望不重启应用的情况下，程序可以自动部署（热部署）。
- 如何实现热部署？

### 1. 关于模板引擎

- 在 Spring Boot 开发环境下禁用模板缓存

```
1 #开发环境下关闭thymeleaf模板缓存，thymeleaf默认是开启状态
2 spring.thymeleaf.cache=false
```



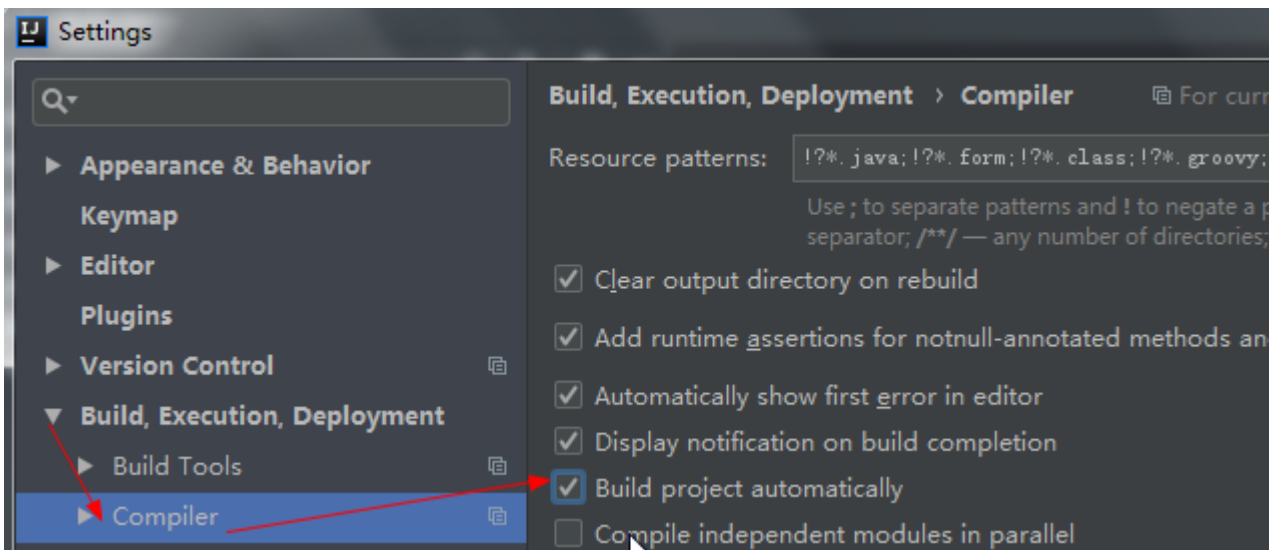
## 2. 添加 Spring Boot Devtools 热部署依赖

```
1 <!--热部署-->
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-devtools</artifactId>
5 </dependency>
```

## 3. IntelliJ IDEA和Eclipse不同，IntelliJ IDEA必须做一些小调整:

- 在 Eclipse 中，修改文件后要手动进行保存，它就会自动编译，就触发热部署现象。
- 在 IntelliJ IDEA 中，修改文件后都是自动保存，默认不会自动编译文件，  
需要手动编译按 `Ctrl + F9` (**推荐使用**) 或 `Build -> Build Project` ;  
或者进行以下设置才会自动编译 (**效果不明显**)

(File -> Settings -> Build, Execution, Deployment -> Compiler -> 勾选 Build project automatically)



## 5.5 分析 SpringMVC 自动配置

Spring Boot 为 Spring MVC 提供了适用于多数应用的自动配置功能 (`WebMvcAutoConfiguration`) 。

在Spring默认基础上，自动配置添加了以下特性：

- 引入 `ContentNegotiatingViewResolver` 和 `BeanNameViewResolver` beans.
  - 自动配置了视图解析器ViewResolver (根据方法返回值获取视图对象View，视图对象决定如何渲染？重定向Or 转发)
  - `ContentNegotiatingViewResolver` : 组合所有的视图解析器的 (通过源码可分析出)

```
1 public class ContentNegotiatingViewResolver
2   //146
3   public View resolveViewName(String viewName, Locale locale) throws Exception {
4     RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
5     Assert.state(attrs instanceof ServletRequestAttributes
6       , "No current ServletRequestAttributes");
```

```
7 List<MediaType> requestedMediaTypes =
8     this.getMediaTypes(((ServletRequestAttributes)attrs).getRequest());
9 if (requestedMediaTypes != null) {
10     //选择所有候选的视图对象
11     List<View> candidateViews = this.getCandidateViews(viewName, locale,
12                                                         requestedMediaTypes);
13     //从候选中选择最合适的视图对象
14     View bestView = this.getBestView(candidateViews, requestedMediaTypes,
15                                     attrs);
16
17     //存入所有视图解析器
18     private List<ViewResolver> viewResolvers;
19     107
20     protected void initServletContext(ServletContext servletContext) {
21         Collection<ViewResolver> matchingBeans =
22             BeanFactoryUtils.beansOfTypeIncludingAncestors(
23                 //从容器中获取所有的视图解析器
24                 this.obtainApplicationContext(), ViewResolver.class).values();
25     }
```

- 自定义视图解析器：可以@Bean向容器中添加一个我们自定义的视图解析器，即可被容器管理使用

```
1 @Bean
2 public ViewResolver myViewResolver () {
3     return new MyViewResolver();
4 }
5
6 private class MyViewResolver implements ViewResolver {
7     @Override
8     public View resolveViewName(String s, Locale locale) throws Exception {
9         return null;
10    }
11 }
12
13 // DispatcherServlet.doDispatch 断点后,发送任意请求,可查看已被容器自动管理了
```

- 自动注册 Converter, GenericConverter, and Formatter beans。
  - Converter：转换器；如: 文本类型转换目标类型，true 转 boolean类型
  - GenericConverter：转换器，Spring内部在注册时，会将Converter先转换为GenericConverter之后，再统一对GenericConverter注册。
  - Formatter：格式化器；如：2017/12/17 格式化 Date类型

```
1 @Bean
2 public FormattingConversionService mvcConversionService() {
3     //传入日期格式，spring.mvc.date-format配置日期格式
4     WebConversionService conversionService =
5         new WebConversionService(this.mvcProperties.getDateFormat());
6     this.addFormatters(conversionService);
7     return conversionService;
8 }
9 //将格式化器添加容器中
10 protected void addFormatters(FormatterRegistry registry) {
11     this.configurators.addFormatters(registry);
12 }
```

- 对 `HttpMessageConverters` 的支持。
  - SpringMVC 用它来转换Http请求和响应的；User\_json User\_xml
  - 可以通过@Bean向容器中添加一个我们自定义 `HttpMessageConverters`，即可被容器管理使用
- 自动注册 `MessageCodesResolver`。
  - 定义错误代码生成规则
- 自动注册 `ConfigurableWebBindingInitializer`。
  - 初始化所有 Web数据绑定器 对象，比如 请求数据 ——》JavaBean
- 对静态资源的支持，包括对 Webjars 的支持。
- 对静态首页 index.html 的支持。
- 对 自定义 Favicon 图标的支持。

如果想保留 Spring Boot MVC的特性，而且还想扩展新的功能（拦截器, 格式化器, 视图控制器等），你可以在你自定义的 `WebMvcConfigurer` 类上增加 `@Configuration` 注解。

如果你想全面控制SpringMVC（也就是不使用默认配置功能），你在自定义的Web配置类上添加 `@Configuration` 和 `@EnableWebMvc` 注解。

## 5.6 扩展 SpringMVC 功能

- 扩展一个视图解析器功能

```
1 <mvc:view-controller path="/mengxuegu" view-name="success"/>
2 <mvc:interceptors>
3     <mvc:interceptor>
4         <mvc:mapping path="/hello"/>
5         <bean></bean>
6     </mvc:interceptor>
7 </mvc:interceptors>
```

- 如果想保留 Spring Boot MVC的特性，而且还想扩展新的功能（拦截器, 格式化器, 视图控制器等），你可以在你自定义的 `WebMvcConfigurer` 类上增加 `@Configuration` 注解。

**自定义配置类保留了所有的自动配置, 也能用我们扩展的功能**

```
1 package com.mengxuegu.springboot.config;
2 .....
3 /**
4  * @Author: www.mengxuegu.com
5  */
6 @Configuration
7 public class MySpringMvcConfigurer implements WebMvcConfigurer{
8     @Override
9     public void addViewControllers(ViewControllerRegistry registry) {
10         // super.addViewControllers(registry);
11         //发送 /mengxuegu 请求来到 success.html
12         registry.addViewController("/mengxuegu").setViewName("success");
13     }
14 }
```

原理：

1. 自定义WebMvcConfigurer自动配置时会导入；

```
@Import({WebMvcAutoConfiguration.EnableWebMvcConfiguration.class})
```

```
1 导入EnableWebMvcConfiguration.class
2 @Import({WebMvcAutoConfiguration.EnableWebMvcConfiguration.class})
3 @EnableConfigurationProperties({WebMvcProperties.class, ResourceProperties.class})
4 @Order(0)
5 public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer, ResourceLoaderAware {
```

3. EnableWebMvcConfiguration 继承了 DelegatingWebMvcConfiguration

```
1 @Configuration
2 public static class EnableWebMvcConfiguration extends DelegatingWebMvcConfiguration {
```

4. 分析 DelegatingWebMvcConfiguration，会将所有web配置组件加到WebMvcConfigurerComposite中，

```
1 @Configuration
2 public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
3     //存储所有的mvc配置类组件
4     private final WebMvcConfigurerComposite configurers =
5         new WebMvcConfigurerComposite();
6
7     @Autowired( required = false )
8     public void setConfigurers(List<WebMvcConfigurer> configurers) {
9         if ( CollectionUtils.isEmpty(configurers)) {
10             this.configurers.addWebMvcConfigurers(configurers);
11         }
12         /*
13          一个参考实现；将所有的WebMvcConfigurer相关配置都来一起调用；
14          public void addViewControllers(ViewControllerRegistry registry) {
15              Iterator var2 = this.delegates.iterator();
16
17              while(var2.hasNext()) {
18                  WebMvcConfigurer delegate = (WebMvcConfigurer)var2.next();
```

```
18         delegate.addViewControllers(registry);
19     }
20 }
21 */
22 }
23 }
```

5. 保留原来的配置类，也添加了新的配置类，所有的WebMvcConfigurer都会一起起作用
6. 效果：SpringMVC的自动配置和我们的扩展配置都会起作用；

## 5.7 全面控制 SpringMVC

如果你想全面控制SpringMVC（SpringBoot对SpringMVC的自动配置都废弃），在自定义的Web配置类上添加 `@Configuration` 和 `@EnableWebMvc` 注解。

```
1
2 /**
3  * @Author: www.mengxuegu.com
4  */
5 @EnableWebMvc
6 @Configuration
7 public class MySpringMvcConfigurer implements WebMvcConfigurer{
8     @Override
9     public void addViewControllers(ViewControllerRegistry registry) {
10         // super.addViewControllers(registry);
11         //发送 /mengxuegu 请求来到 success.html
12         registry.addViewController("/mengxuegu").setViewName("success");
13     }
14 }
```

**原理：**为什么添加 `@EnableWebMvc` 自动配置就失效了？

1. `@EnableWebMvc` 的核心

```
1 @Import(DelegatingWebMvcConfiguration.class)
2 public @interface EnableWebMvc {
```

2. 先记住继承了WebMvcConfigurationSupport类

```
1 @Configuration
2 public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
```

3. 而在 WebMvcAutoConfiguration 上使用了 `@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)`

```
1 容器中没有这个组件的时候，这个自动配置类才生效
2  @ConditionalOnMissingBean({WebMvcConfigurationSupport.class})
3  @AutoConfigureOrder(-2147483638)
4  @AutoConfigureAfter({DispatcherServletAutoConfiguration.class, ValidationAutoConfiguration.class})
5  public class WebMvcAutoConfiguration {
```

```
1  **而 @ConditionalOnMissingBean 表示的是没有WebMvcConfigurationSupport这个组件,**
2
3  **WebMvcAutoConfiguration自动配置类才会生效.**
```

- 相反 @EnableWebMvc 将 WebMvcConfigurationSupport 组件导入进来, 使得 WebMvcAutoConfiguration就失效了
- WebMvcConfigurationSupport 只是SpringMVC最基本的功能;

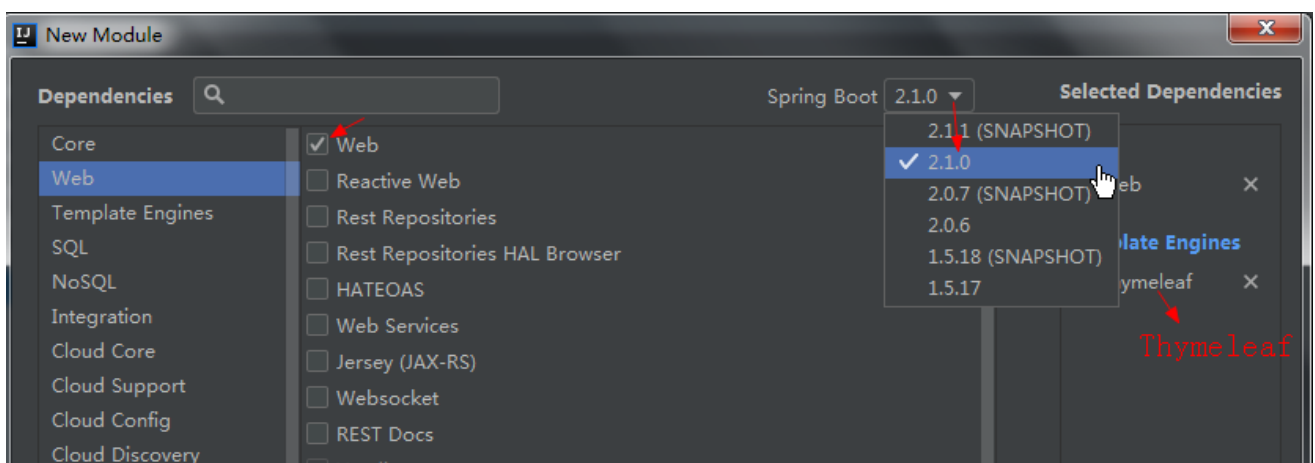
## 5.8 总结 SpringMVC 配置

- 在Spring Boot中自己配置组件的时候，先看容器中有没有公司自己配置的（@Bean、@Component），如果有就用公司自己配置的；如果没有，才自动配置。
- 在Spring Boot中会有非常多的xxxConfigurer帮助我们进行扩展配置。
- 在Spring Boot中会有很多的xxxCustomizer帮助我们进行定制配置。

# 第6章 项目实战-帐单管理系统

## 6.1 初始化项目

### 6.1.1 创建并引入项目资源



## 6.1.2 Thymeleaf修改资源路径

- 使用 `th:href` 修改资源路径；好处是：会自动获取应用名

```
1 <head lang="en" th:fragment="public_head">
2   <meta charset="UTF-8">
3   <title>梦学谷账单管理系统1</title>
4   <link rel="stylesheet" th:href="@{/css/public.css}" href="../css/public.css"/>
5   <link rel="stylesheet" th:href="@{/css/style.css}" href="../css/style.css"/>
6 </head>
7
8 <td>
9   <a href="view.html"></a>
10  <a href="update.html">
11  <a href="#" class="removeUser"></a>
12 </td>
13
14 <!--webjars方式引入-->
15 <script th:src="@{/webjars/jquery/3.3.1/jquery.js}" src="../js/jquery.js"></script>
16 <script th:src="@{/js/js.js}" src="../js/js.js"></script>
17
18 # 上面会自动获取到应用名 /bill
19 server.servlet.context-path=/bill
```

## 6.1.3 Thymeleaf引入片段时传入参数

```
1 <div class="left" id="public_left">
2   <h2 class="leftH2"><span class="span1"></span>功能列表 <span></span></h2>
3   <nav>
4     <ul class="list">
5       <li><a href="../bill/list.html">账单管理</a></li>
6       <li><a href="../provider/list.html">供应商管理</a></li>
7       <!-- 接收引入时传入的activeUri参数值-->
8       <li th:id="${activeUri == 'user' ? 'active' : ''}" id="active">
9         <a th:href="@{/user/list}" href="/user/list">用户管理</a>
10      </li>
11      <li><a href="../main/password.html">密码修改</a></li>
12      <li><a href="../main/login.html">退出系统</a></li>
13    </ul>
14  </nav>
15 </div>
16
17 引入公共片段处, 传入参数
18 <div class="left" th:replace="main/public :: #public_left(activeUri='user')">
```



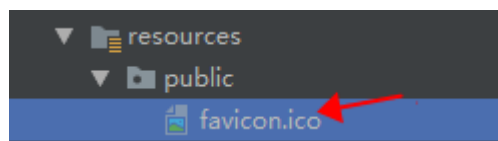


## 6.2 默认访问欢迎页

- 默认访问的欢迎页是 login.html

```
1 @Configuration
2 public class MySpringMvcConfigurer {
3
4
5     @Bean
6     public WebMvcConfigurer webMvcConfigurer() {
7         return new WebMvcConfigurer(){
8             //添加视图控制
9             @Override
10             public void addViewControllers(ViewControllerRegistry registry) {
11                 registry.addViewController("").setViewName("main/login");
12                 registry.addViewController("/index.html").setViewName("main/login");
13             }
14         };
15     }
16
17 }
```

- 更改图标



## 6.3 国际化信息

### 6.3.1 SpringMVC国际化步骤

- 编写国际化配置文件，需要显示的国际化内容写到配置中
- 使用 `ResourceBundleMessageSource` 管理国际化资源文件
- 在 JSP 页面中使用 `<fmt:message>` 标签取出国际化内容

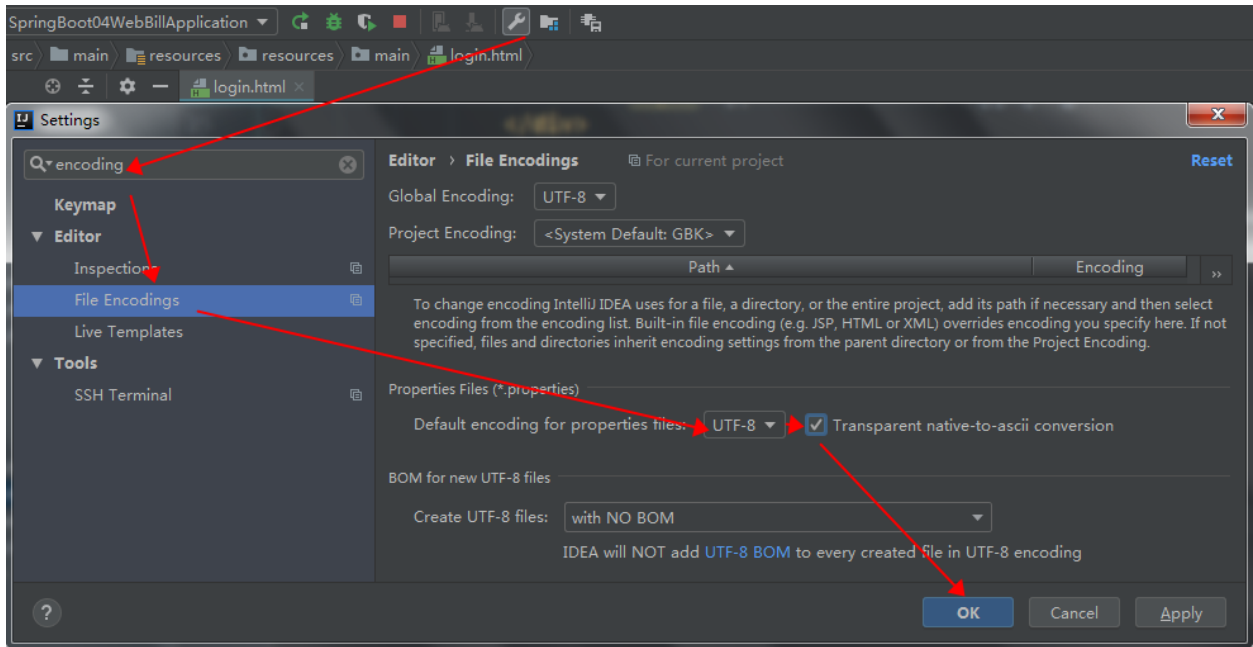
### 6.3.2 SpringBoot国际化步骤

1. 编写国际化配置文件，需要要显示的国际化内容写到配置中

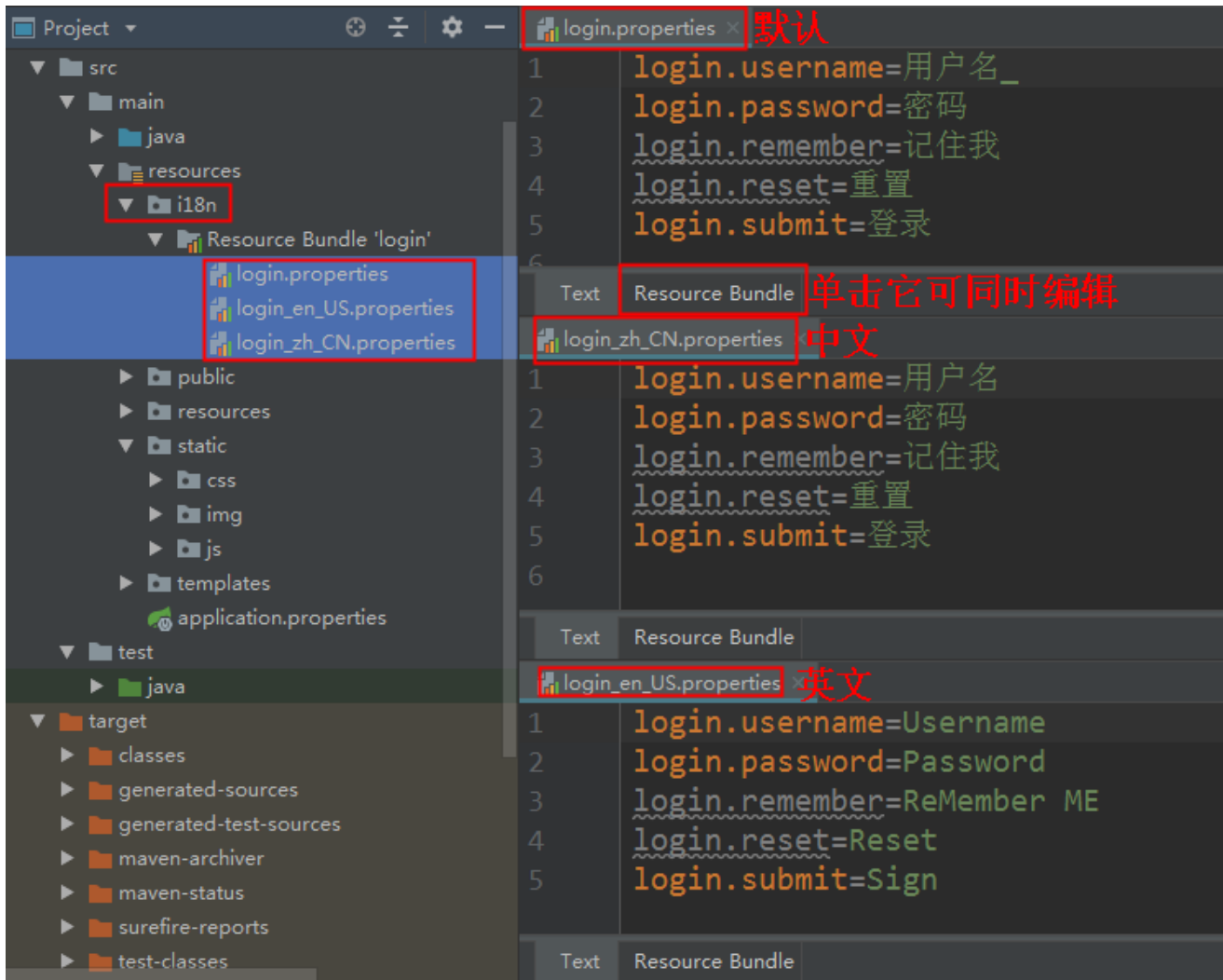


- 1 #类路径下创建 i18n 目录存放配置文件(i18n 是“国际化”的简称)
- 2 login.properties (默认国际化文件)
- 3 #login\_语言代码\_国家代码.properties
- 4 login\_zh\_CN.properties (中文\_中国 国际化文件)
- 5 login\_en\_US.properties (英文\_美国 国际化文件)

- 先修改 properties 文件的字符编码，不然出现乱码，进行如下设置：



- 类路径下创建 i18n 目录存放配置文件



- 2. Spring Boot 已经自动配置了管理国际化资源文件的组件 `MessageSourceAutoConfiguration`

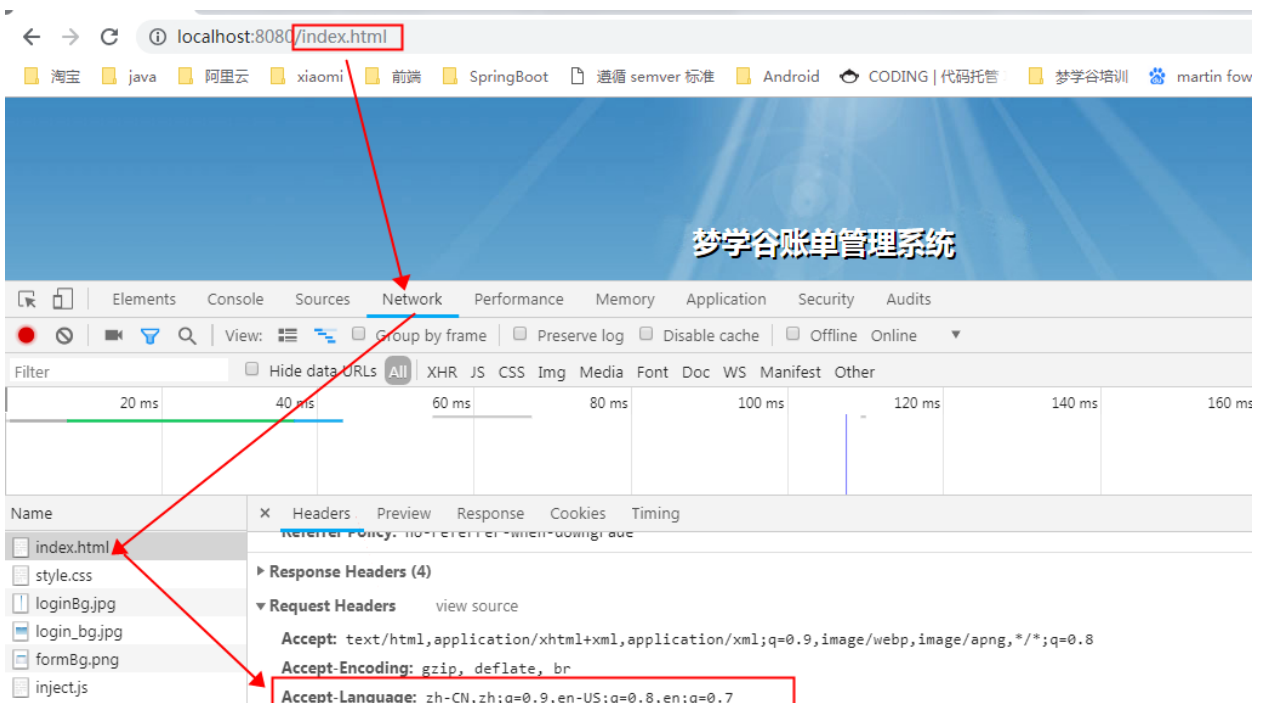
```

1 public class MessageSourceAutoConfiguration {
2
3     @Bean
4     @ConfigurationProperties(prefix = "spring.messages")
5     public MessageSourceProperties messageSourceProperties() {
6         return new MessageSourceProperties();
7     }
8
9     @Bean
10    public MessageSource messageSource() {
11        // 国际化资源相关属性
12        MessageSourceProperties properties = this.messageSourceProperties();
13        //管理国际化资源的组件
14        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
15        == public class MessageSourceProperties { =====
16        == private String basename = "messages"; =====
17        默认国际化资源文件的基础名(就是去掉 语言_国家代码 之后的名称,上面自定义的是login)
18        即 如果我们定义为 messages.properties 就可以放在类路径下,就可不做任何配置,
19        就会被直接被加载

```

仅供购买者学习，禁止盗版、转卖、传播课程

```
1 public class WebMvcAutoConfiguration {
2     @Bean
3     @ConditionalOnMissingBean
4     @ConditionalOnProperty(prefix = "spring.mvc", name = {"locale"})
5     public LocaleResolver localeResolver() {
6         if (this.mvcProperties.getLocaleResolver() == LocaleResolver.FIXED) {
7             return new FixedLocaleResolver(this.mvcProperties.getLocale());
8         } else {
9             //1. 根据请求头来获取区域信息
10            AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();
11            localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
12            return localeResolver;
13        }
14    }
15
16    //2. 请求头区域信息解析器
17    public class AcceptHeaderLocaleResolver implements LocaleResolver {
18        public Locale resolveLocale(HttpServletRequest request) {
19            public Locale resolveLocale(HttpServletRequest request) {
20                Locale defaultLocale = this.getDefaultLocale();
21                //
22                if (defaultLocale != null && request.getHeader("Accept-Language") != null) {
23                    return defaultLocale;
24                } else {
25                    //3. 获取当前收到的请求区域信息, 从而来选择国际化语言
26                    Locale requestLocale = request.getLocale();
27                }
28            }
29        }
30    }
31 }
```



- 通过上面分析, 是根据请求头带来的区域信息来选择对应的国际化信息, 即我们可以自定义区域信息解析器

## 5. 点击链接切换国际化

- 请求参数中设置区域信息

[illegible]

- 自定义区域信息解析器来进行设置区域信息

```

1 package com.mengxuegu.springboot.component;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.stereotype.Component;
5 import org.springframework.util.StringUtils;
6 import org.springframework.web.servlet.LocaleResolver;
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9 import java.util.Locale;
10 /**
11  * 自定义解析器来切换国际化信息,
12  * 需要再注入到容器器
13  * @Auther: www.mengxuegu.com
14  */
15 public class MyLocaleResolver implements LocaleResolver {
16     /**解析区域信息*/
17     @Override
18     public Locale resolveLocale(HttpServletRequest httpServletRequest) {
19         System.out.println("区域信息。。。");
20         //获取请求头中的l参数值
21         String l = httpServletRequest.getParameter("l");
22         //获取浏览器上的区域信息
23         Locale locale = httpServletRequest.getLocale();
24         //获取当前操作系统 默认的区域信息
25         // Locale locale = Locale.getDefault();
26
27         //参数有区域信息，则用参数里的区域信息
28         if (!StringUtils.isEmpty(l)) {
29             String[] split = l.split("_");
30             //参数：语言代码，国家代码
31             locale = new Locale(split[0], split[1]);
32         }
33         return locale;
34     }
35
36     @Override
37     public void setLocale(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, Locale locale) {
38     }
39 }
40

```

- 需要替换mvc自动配置类中区域信息解析器,(返回值与方法名要和下面保持必须一致)

```
1 package com.mengxuegu.springboot.config;
2
3 @Configuration
4 public class MySpringMvcConfigurer{
5     //需要替换mvc自动配置类中区域解析器,
6     @Bean
7     public LocaleResolver localeResolver() {
8         return new MyLocaleResolver();
9     }
10 }
11
```

## 6.4 登录模块开发

- 登录控制层

```
1 @Controller
2 public class LoginController {
3
4     @PostMapping("/login")
5     // @RequestMapping(value = "/user/login", method = RequestMethod.POST)
6     public String login(@RequestParam("username") String username,
7                         @RequestParam("password") String password,
8                         Map<String, Object> map) {
9         if (!StringUtils.isEmpty(username) && "123".equals(password)) {
10             //登录成功,
11             //防止表单重复提交, 通过重定向到主页, 需要添加一个视图
12             return "redirect:/main.html";
13         }
14         //登录失败
15         map.put("msg", "用户名或密码错误!");
16         return "/main/login";
17     }
18 }
```

- 页面

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head lang="en">
4     <meta charset="UTF-8">
5     <title>系统登录 - 梦学谷账单管理系统</title>
6     <link rel="stylesheet" th:href="@{/css/style.css}" href="../../css/style.css"/>
7 </head>
8 <body class="login_bg">
9     <section class="loginBox">
```

## 6.5 自定义拦截器-登录校验

- 仅供购买者学习，禁止盗版、转卖、传播课程

```
8     if (loginUser != null) {
9         //已经登录，放行请求
10        return true;
11    }
12    //未登录, 转发到登录页面
13    request.setAttribute("msg", "无权限，请登录后访问！");
14    request.getRequestDispatcher("/index.html").forward(request, response);
15    return false;
16 }
17 }
```

- 添加拦截器到容器中

```
1 package com.mengxuegu.springboot.config;
2 .....
3 @Configuration
4 public class MySpringMvcConfigurer {
5
6     //所有的WebMvcConfigurer组件都会一起起作用
7     @Bean //将当前组件添加到容器当前才可生效
8     public WebMvcConfigurer webMvcConfigurer(){
9         WebMvcConfigurer webMvcConfigurer = new WebMvcConfigurer() {
10             //添加视图控制
11             @Override
12             public void addViewControllers(ViewControllerRegistry registry) {
13                 registry.addViewController("/").setViewName("main/login");
14                 registry.addViewController("/index.html").setViewName("main/login");
15                 registry.addViewController("/main.html").setViewName("main/index");
16             }
17
18             //添加拦截器
19             @Override
20             public void addInterceptors(InterceptorRegistry registry) {
21                 registry.addInterceptor(new LoginHandlerInterceptor())
22                     // 拦截所有请求 /**
23                     .addPathPatterns("/**")
24                     // 排除不需要拦截的请求
25                     .excludePathPatterns("/", "/index.html", "/login")
26                     //SpringBoot2+中要排除静态资源路径, 因访问时不会加/static，所以配置如下
27                     ,"/css/**", "/img/**", "/js/**");
28             }
29         };
30         return webMvcConfigurer;
31     }
32
33     @Bean
34     public LocaleResolver localeResolver() {
35         return new MyLocaleResolver();
36     }
37
38 }
39
```



## 6.6 主页模块开发-退出系统

- 右上角和主页显示登录用户名

```
1 <span style="color: #fff21b"> [[${session.loginUser}]]</span>, 欢迎你 !
2
3 <div class="wFont">
4   <h2 th:text="${session.loginUser}">Admin</h2>
5   <p>欢迎来到梦学谷账单管理系统!</p>
6   <span id="hours"></span>
7 </div>
```

- 点击 `退出`，退出系统

```
1 //退出系统
2 @GetMapping("/logout")
3 public String logout(HttpSession session) {
4     System.out.println("logout被调用。。。");
5     session.removeAttribute("loginUser");
6     session.invalidate();
7     //回到登录页面
8     return "redirect:/index.html";
9 }
```

## 6.7 分析 Restful 架构

1. Restful 架构: 通过HTTP请求方式来区分对资源CRUD操作, 请求 URI 是 `/资源名称/资源标识`,

对比下:

	普通CRUD	RestfulCRUD
查询	getProvider	provider---GET
添加	addProvider?xxx	provider---POST
修改	updateProvider?id=xxx	provider/{id}---PUT
删除	deleteProvider?id=1	provider/{id}---DELETE

2. 项目使用Rest处理架构

项目功能	请求URI	请求方式
查询所有供应商	providers	GET
查询某个供应商详情	provider/1	GET
来到修改页面（查出供应商进行信息回显）	provider/1	GET
修改供应商	provider	PUT
前往添加页面	provider	GET
添加供应商	provider	POST
删除供应商	provider/1	DELETE

## 6.8 供应商列表查询

```
1  /**
2   * 供应商控制层
3   * @Author: 梦学谷
4   */
5   @Controller
6   public class ProviderController {
7       Logger logger = LoggerFactory.getLogger(getClass());
8
9       @Autowired
10      ProviderDao providerDao;
11      //查询所有供应商并响应列表页面
12      @GetMapping("/providers")
13      public String list(@RequestParam(value = "providerName", required = false) String providerName,
14      Map<String, Object> map) {
15          logger.info("providerName = " + providerName);
16
17          Collection<Provider> providers = providerDao.getAll(providerName);
18          map.put("providers", providers);
19
20          return "provider/list";
21      }
22  }
```

```
1  <div class="right">
2      <div class="location">
3          <strong>你现在所在的位置是:</strong>
4          <span>供应商管理页面</span>
5      </div>
6      <form id="searchForm" th:method="get" th:action="@{/providers}">
7          <div class="search" >
8              <span>供应商名称 : </span>
```

```
9      <input type="text" name="providerName" placeholder="请输入供应商的名称"/>
10     <input type="button" onclick="$('#searchForm').submit();" value="查询"/>
11     <a href="add.html">添加供应商</a>
12 </div>
13 </form>
14 <!--供应商操作表格-->
15 <table class="providerTable" cellpadding="0" cellspacing="0">
16     <tr class="firstTr">
17         <th width="10%">供应商编码</th>
18         <th width="20%">供应商名称</th>
19         <th width="10%">联系人</th>
20         <th width="10%">联系电话</th>
21         <th width="10%">传真</th>
22         <th width="10%">创建时间</th>
23         <th width="30%">操作</th>
24     </tr>
25     <tr th:each="p : ${providers}">
26         <td th:text="${p.getPid()}">PRO-CODE—001</td>
27         <td th:text="${p.providerName}">测试供应商001</td>
28         <td th:text="${p.people}">韩露</td>
29         <td th:text="${p.phone}">15918230478</td>
30         <td th:text="${p.fax}">15918230478</td>
31         <td th:text="${#dates.format( p.createDate, 'yyyy-MM-dd')}">2015-11-12</td>
32         <td>
33             <a href="view.html"></a>
34             <a href="update.html"></a>
35             <a href="#" class="removeProvider"></a>
36         </td>
37     </tr>
38 </table>
39 </div>
```

## 6.9 供应商详情查询

```
1 //查看某个供应商详情
2 @GetMapping("/view/{pid}")
3 public String view(@PathVariable("pid") Integer pid, Map<String, Object> map) {
4     Provider provider = providerDao.getProvider(pid);
5     map.put("provider", provider);
6     //详情页面
7     return "provider/view";
8 }
```

```
1 provider/list.html
2 <a href="view.html" th:href="@{/view/} + ${p.pid}"></a>
3
4 provider/view.html
5 <div class="providerView">
```

```
6 <p><strong>供应商编码：</strong><span th:text="${provider.pid}">PRO-CODE</span></p>
7 <p><strong>供应商名称：</strong><span th:text="${provider.providerName}">测试供应商 </span></p>
8 <p><strong>联系人：</strong><span th:text="${provider.people}">韩露</span></p>
9 <p><strong>联系电话：</strong><span th:text="${provider.phone}">1591**478</span>
10 </p>
11 <p><strong>传真：</strong><span th:text="${provider.fax}">15918230478</span></p>
12 <p><strong>描述：</strong><span th:text="${provider.describe}">描述</span></p>
13
14 <a th:href="@{/providers}" href="list.html">返回</a>
15 </div>
```

## 6.10 供应商修改

- 发送put请求修改供应商信息

1. 在SpringMVC中配置HiddenHttpMethodFilter ( SpringBoot自动配置好了 )
2. 页面创建一个method="post"表单
3. 创建一个input标签 name="\_method" , value="指定请求方式"

- 前往 修改页面, 方法重用详情查询的方法

- 方法改造

```
1 /**
2  * 查看某个供应商详情
3  * type=null 默认view详情页面，type=update 修改页面
4  */
5 @GetMapping("/provider/{pid}")
6 public String view(@RequestParam(value = "type", defaultValue = "view")
7     String type,
8     @PathVariable("pid") Integer pid,
9     Map<String, Object> map) {
10     Provider provider = providerDao.getProvider(pid);
11     map.put("provider", provider);
12
13     // type=null 默认view详情页面，type=update 修改页面
14     return "provider/" + type;
15 }
16
17 //修改供应商信息
18 @PutMapping("/provider")
19 public String update(Provider provider) {
20     logger.info("修改供应商信息: " + provider);
21     provider.setCreateDate(new Date());
22     providerDao.save(provider);
23     //重定向到列表页
24     return "redirect:/providers";
25 }
```

```
1 list.html
```

```
2 <a href="view.html" th:href="@{/provider/} + ${p.pid}"></a>
3 <a href="update.html" th:href="@{/provider/} + ${p.pid} + '?type=update'"></a>
4
5 update.html
6 <form action="#" id="updateForm" th:method="post" th:action="@{/provider}">
7 <!--
8 发送put请求修改供应商信息
9 1. 在SpringMVC中配置HiddenHttpMethodFilter ( SpringBoot自动配置好了 )
10 2. 页面创建一个method="post"表单
11 3. 创建一个input标签 name="_method" , value="指定请求方式"
12 -->
13 <input type="hidden" name="_method" value="put"/>
14 <input type="hidden" name="pid" th:value="${provider.pid}"/>
15
16 <!--div的class 为error是验证错误，ok是验证成功-->
17 <div class="">
18 <label for="providerName">供应商名称：</label>
19 <input type="text" name="providerName" th:value="${provider.providerName}"
id="providerName"/>
20 <span>*请输入供应商名称</span>
21 </div>
22 <div>
23 <label for="people">联系人：</label>
24 <input type="text" name="people" id="people" th:value="${provider.people}"/>
25 <span>*请输入联系人</span>
26
27 </div>
28 <div>
29 <label for="phone">联系电话：</label>
30 <input type="text" name="phone" id="phone" th:value="${provider.phone}"/>
31 <span>*请输入联系电话</span>
32 </div>
33 <div>
34 <label for="address">联系地址：</label>
35 <input type="text" name="address" id="address" th:value="${provider.address}"/>
36 <span></span>
37 </div>
38 <div>
39 <label for="fax">传真：</label>
40 <input type="text" name="fax" id="fax" th:value="${provider.fax}"/>
41 <span></span>
42 </div>
43 <div>
44 <label for="describe">描述：</label>
45 <input type="text" name="describe" id="describe" th:value="${provider.describe}"/>
46 </div>
47 <div class="providerAddBtn">
48 <input type="button" value="保存" onclick="$('#updateForm').submit();"/>
49 <input type="button" value="返回" onclick="history.back(-1)"/>
50 </div>
51 </form>
```

## 6.11 供应商添加

- 前往添加供应商页面

```
1 //前往添加供应商页面
2 @GetMapping("/provider")
3 public String toAddPage() {
4     //前往添加供应商页面
5     return "provider/add";
6 }
```

- 提交供应商数据

```
1 //处理添加供应商请求
2 @PostMapping("/provider")
3 public String addProvider(Provider provider) {
4     //SpringMVC会自动将请求参数与形参对象的属性——绑定
5     //要求：请求参数名要与形参对象的属性名相同
6     logger.info("添加供应商信息：" + provider);
7
8     provider.setCreateDate(new Date());
9     providerDao.save(provider);
10
11     //添加完成,回到供应商列表页面
12     //通过redirect重定向 或forward转发到一个请求地址, / 代表当前项目路径
13     return "redirect:/providers";
14 }
```

## 6.12 供应商删除

```
1 //删除操作
2 @DeleteMapping("provider/{pid}")
3 public String delete(@PathVariable("pid") Integer pid) {
4     logger.info("删除供应商：" + pid);
5     providerDao.delete(pid);
6     return "redirect:/providers";
7 }
```

```
1 <a href="#" th:attr="del_uri=@{/provider/}+${p.pid}" class="delete"></a>
```

## 第7章 SpringBoot 错误处理机制

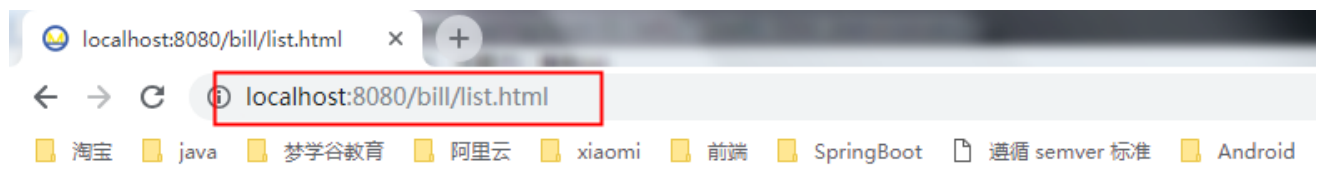
我们所开发的项目大多是直接面向用户的，而程序出现异常往往又是不可避免的，那该如何减少程序异常对用户体验的影响呢？

那么下面来介绍下 SpringBoot 为我们提供的处理方式。

### 7.1 默认的错误处理机制

#### 7.1.1 出现错误时页面效果

- 浏览器发送一个不存在的请求时，会报404



### Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

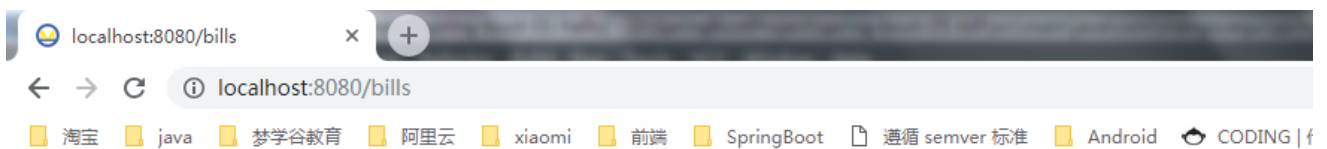
Tue Nov 09 20:48:42 CST 2088

There was an unexpected error (type=Not Found, status=404).

No message available

- 服务器内部发生错误的时候，页面会返回什么呢？

```
1 @Controller
2 public class BillController {
3
4     @GetMapping("/bills")
5     public String list() {
6         //模拟500错误
7         int i=1/0;
8         return "bill/list";
9     }
10 }
11
```



## Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Nov 09 20:51:24 CST 2088

There was an unexpected error (type=Internal Server Error, status=500).  
/ by zero

java.lang.ArithmeticException: / by zero at com.mengxuegu.springboot.controller.BillController.list(BillController.java:7)  
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(  
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.Me  
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:215)

- 通过上面，我们会发现无论是发生什么错误，SpringBoot 都会返回一个状态码以及一个错误页面，这个错误页面是怎么来的呢？

### 7.1.2 底层原理分析

底层原理关注 `ErrorMvcAutoConfiguration` 错误自动配置类

第1步：

`ErrorPageCustomizer` 错误页面定制器

```
1 private static class ErrorPageCustomizer implements ErrorPageRegistrar, Ordered {
2
3     public void registerErrorPages(ErrorPageRegistry errorPageRegistry) {
4         ErrorPage errorPage = new ErrorPage(
5             this.dispatcherServletPath.getRelativePath(
6                 // 出现错误后来到 /error 请求进行处理 (类似web.xml注册的错误页面规则)
7                 this.properties.getError().getPath()); //private String path = "/error";
8         errorPageRegistry.addErrorPages(new ErrorPage[]{errorPage});
9     }
10
11     public int getOrder() {
12         return 0;
13     }
14 }
```



15 }

当应用出现了4xx或5xx之类的错误, `ErrorPageCustomizer` 就会被激活, 它主要用于定制错误处理的响应规则, 就会发送一个/error请求, 它会交给 `BasicErrorController` 进行处理

### 第2步：

`BasicErrorController` 就会接收 /error 请求处理。

```
1  @Controller
2  @RequestMapping("${server.error.path:${error.path:/error}}")
3  public class BasicErrorController extends AbstractErrorController {
4
5      //通过请求头判断调用下面哪个访求： text/html
6      //响应 html 类型的数据；接收浏览器发送的请求
7      @RequestMapping(produces = {"text/html"})
8      public ModelAndView errorHtml(HttpServletRequest request,
9                                  HttpServletResponse response) {
10         HttpStatus status = this.getStatus(request);
11         Map<String, Object> model =
12             Collections.unmodifiableMap(this.getErrorAttributes(request, this.isIncludeStackTrace(request,
13                                                         MediaType.TEXT_HTML)));
14         response.setStatus(status.value());
15         //去哪个页面作为错误页面，包括 页面地址与页面内容，里面有一个ErrorViewResolver
16         ModelAndView modelAndView = this.resolveErrorView(request, response, status, model);
17         return modelAndView != null ? modelAndView
18             : new ModelAndView("error", model);
19     }
20
21     //通过请求头判断： */*
22     @RequestMapping
23     @ResponseBody //响应 json 类型的数据；接收 其他客户端 发送的请求
24     public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
25         Map<String, Object> body = this.getErrorAttributes(request, this.isIncludeStackTrace(request,
26                                                         MediaType.ALL));
27         HttpStatus status = this.getStatus(request);
28         return new ResponseEntity(body, status);
29     }
30 }
```

`BasicErrorController` 会接收一个/error请求, 两个 方法处理, 第1个errorHtml响应html数据, 还有一个error用来响应 json数据 的, 使用了 `ErrorViewResolver` (`DefaultErrorViewResolver`) 组件进行封装视图

### 第3步：

`DefaultErrorViewResolver` 去解析具体响应的错误页面。

```
1  public class DefaultErrorViewResolver implements ErrorViewResolver, Ordered {
2      public ModelAndView resolveErrorView(HttpServletRequest request,
3
4          HttpStatus status, Map<String, Object> model) {
```

```
4
5     ModelAndView modelAndView = this.resolve(String.valueOf(status), model);
6     if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
7         //找4xx 5xx页面
8         modelAndView = this.resolve((String)SERIES_VIEWS.get(
9             status.series()), model);
10    }
11
12    return modelAndView;
13 }
14
15
16 private ModelAndView resolve(String viewName, Map<String, Object> model) {
17     //SpringBoot默认根据状态码响应状态码页面，如 error/404 (templates/error/404.html)
18     String errorViewName = "error/" + viewName;
19     //如果模板引擎解析这个页面地址，则使用模板引擎解析
20     TemplateAvailabilityProvider provider =
21         this.templateAvailabilityProviders.getProvider(errorViewName,
22             this.applicationContext);
23     //如果模板引擎可用，返回errorViewName指定的视图
24     return provider != null ? new ModelAndView(errorViewName, model)
25         //如果模板引擎不可用，则调以下方法，在静态资源目录下找errorViewName对应的页面
26         : this.resolveResource(errorViewName, model);
27 }
28
29 private ModelAndView resolveResource(String viewName, Map<String, Object> model) {
30     //从静态资源目录下找状态码的错误页面，如 404.html
31     String[] var3 = this.resourceProperties.getStaticLocations();
32     int var4 = var3.length;
33
34     for(int var5 = 0; var5 < var4; ++var5) {
35         String location = var3[var5];
36         try {
37
38             Resource resource = this.applicationContext.getResource(location);
39             resource = resource.createRelative(viewName + ".html");
40             if (resource.exists()) {
41                 return new ModelAndView(
42                     new DefaultErrorViewResolver.HtmlResourceView(resource)
43                     , model);
44             }
45         } catch (Exception var8) {
46         }
47     }
48     return null;
49 }
50
51 //还可以定义 4xx , 5xx的页面
52 static {
53     Map<Series, String> views = new EnumMap(Series.class);
54     views.put(Series.CLIENT_ERROR, "4xx");
55     views.put(Series.SERVER_ERROR, "5xx");
56
57     SERIES_VIEWS = Collections.unmodifiableMap(views);
58 }
```

57 }

通过以上分析则可以自定义错误页面

第4步：

DefaultErrorAttributes 错误页面可获取到的数据信息

通过 BasicErrorController 的方法中响应的 module 可定位到响应哪些数据，从而引出 ErrorAttributes 的实现类 DefaultErrorAttributes，DefaultErrorAttributes 中绑定的所有值都可在页面获取到。

```
1 public abstract class AbstractErrorController implements ErrorController {
2     //以下接口实现类 DefaultErrorAttributes 封装了响应的错误数据。
3     private final ErrorAttributes errorAttributes;
```

```
1 public class DefaultErrorAttributes implements ErrorAttributes, HandlerExceptionResolver, Ordered {
2     public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {
3         Map<String, Object> errorAttributes = new LinkedHashMap();
4         errorAttributes.put("timestamp", new Date());
5         this.addStatus(errorAttributes, webRequest);
6         this.addErrorDetails(errorAttributes, webRequest, includeStackTrace);
7         this.addPath(errorAttributes, webRequest);
8         return errorAttributes;
9     }
10 }
```

```
11 下面省略一大波可获取的数据
12  timestamp：时间戳
13  status：状态码
14  error：错误提示
15  exception：异常对象
16  message：异常消息
17  errors：JSR303数据校验出现的错误
```

## 7.2 自定义错误响应页面

### • 第1种：有模板引擎

- error/状态码：精确匹配，将错误页面命名为 错误状态码.html 放在模板引擎目录 templates 下的 error 目录下，发生对应状态码错误时，就会响应对应的模板页面
- error/4xx、error/5xx：模糊匹配，可以将错误页面命名为 4xx 和 5xx，有来匹配对应类型的所有错误
- 采用精确优先
- 错误页面可获取到的数据信息

```
1 timestamp：时间戳
2 status：状态码
3 error：错误提示
4 exception：异常对象
5 message：异常消息
6 errors：JSR303数据校验出现的错误
```

- 第2种：没有模板引擎（模板引擎找不到对应错误页面）

- 静态资源目录下的 error 目录中找

- 第3种：模板目录与静态目录下都找不到对应错误页面，就响应 SpringBoot 默认的错误页面

通过 `BasicErrorController` 的 `errorHtml` 方法最后一行可知，没有找到则找 `error` 视图对象，`error` 定义在 `ErrorMvcAutoConfiguration` 的 `defaultErrorView` 中

```
1 protected static class WhitelabelErrorViewConfiguration {
2     private final ErrorMvcAutoConfiguration.SpelView defaultErrorView = new
      ErrorMvcAutoConfiguration.SpelView("<html><body><h1>Whitelabel Error Page</h1><p>This application has
      no explicit mapping for /error, so you are seeing this as a fallback.</p><div id='created'>${timestamp}</div>
      <div>There was an unexpected error (type=${error}, status=${status}).</div><div>${message}</div></body>
      </html>");
3
4     protected WhitelabelErrorViewConfiguration() {
5     }
6
7     @Bean( name = {"error"})
8     @ConditionalOnMissingBean( name = {"error"})
9     public View defaultErrorView() {
10         return this.defaultErrorView;
11     }
```

## 7.3 自定义数据进行响应

- 分析：

出现错误以后，会发送 `/error` 请求，会被 `BasicErrorController` 处理，而响应的数据是由 `getErrorAttributes` 封装的(就是 `ErrorController` 的实现类 `AbstractErrorController.getErrorAttributes` 的方法)，所以我们只需要自定义 `ErrorAttributes` 实现类即可

- 自定义 `ErrorAttributes`

```
1 @Component //向容器中添加该组件
2 public class MyErrorAttributes extends DefaultErrorAttributes {
3     /**
4      * 自定义数据进行响应
5      */
6     @Override
7     public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {
8         Map<String, Object> map = super.getErrorAttributes(webRequest, includeStackTrace);
9         map.put("company", "mengxuegu.com");
10        return map;
11    }
12
13 }
```

- 错误页面获取

```
1 <body>
2     4xx错误。。
3     <h2>[[${company}]]</h2>
4 </body>
```

## 第8章 嵌入式Servlet容器自定义配置

### 8.1 注册Servlet三大组件 Servlet/Filter/Listener

- 以前 Web 应用使用外置Tomcat 容器部署，可在 web.xml 文件中注册 Servlet 三大组件；
- 而由于 Spring Boot 默认是以 jar 包的方式运行嵌入式Servlet容器来启动应用，没有web.xml文件，Spring提供以下Bean来注册三大组件：
  - ServletRegistrationBean：注册自定义Servlet
  - FilterRegistrationBean：注册自定义Filter
  - ServletListenerRegistrationBean：注册自定义Listener
- Servlet 组件

```
1 package com.mengxuegu.springboot.servlet;
2 import javax.servlet.*;
3 import java.io.IOException;
4
5 //自定义Servlet组件
6 public class HelloServlet extends HttpServlet {
7     //处理git请求
8     @Override
9     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
10     IOException {
11         resp.getWriter().write("HelloServlet success。。。。");
12     }
13     @Override
14     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
15     IOException {
16         super.doGet(req, resp);
17     }
18 }
```

```
1 package com.mengxuegu.springboot.config;
2 ...
```

```
3 //注册Servlet相关组件
4 @Configuration
5 public class MyServletConfig {
6     //注册Servlet组件
7     @Bean
8     public ServletRegistrationBean helloServlet() {
9         //参数1：自定义Servlet，参数2：url映射
10         ServletRegistrationBean<HelloServlet> bean =
11             new ServletRegistrationBean<>(new HelloServlet(), "/hello");
12         //设置servlet组件参数配置，如下面加载顺序
13         bean.setLoadOnStartup(1);
14         return bean;
15     }
16 }
```

- Filter 组件

```
1 package com.mengxuegu.springboot.filter;
2 import javax.servlet.*;
3 import java.io.IOException;
4
5 //自定义过滤器
6 public class MyFilter implements Filter {
7     @Override
8     public void init(FilterConfig filterConfig) throws ServletException {
9         System.out.println("filter初始化");
10    }
11    @Override
12    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain) throws IOException, ServletException {
13        System.out.println("MyFilter过滤完成");
14        //放行
15        filterChain.doFilter(servletRequest, servletResponse);
16    }
17    @Override
18    public void destroy() {
19        System.out.println("filter销毁");
20    }
21 }
```

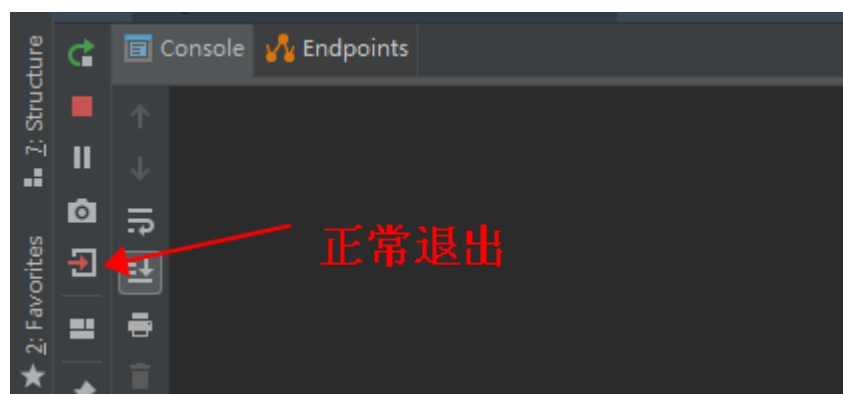
```
1 package com.mengxuegu.springboot.config;
2
3 //注册Servlet相关组件
4 @Configuration
5 public class MyServletConfig {
6     //注册Filter组件
7     @Bean
8     public FilterRegistrationBean myFilter() {
9         FilterRegistrationBean bean = new FilterRegistrationBean();
10        //指定过滤器
11        bean.setFilter(new MyFilter());
```

```
12    //过滤哪些请求
13    bean.setUrlPatterns(Arrays.asList("/hello"));
14    return bean;
15 }
16 }
```

- Listener 组件

```
1 package com.mengxuegu.springboot.listener;
2 import javax.servlet.*;
3
4 //监听应用启动与销毁
5 public class MyListener implements ServletContextListener {
6     @Override
7     public void contextInitialized(ServletContextEvent sce) {
8         System.out.println("SpringBoot.Servlet应用启动");
9     }
10    @Override
11    public void contextDestroyed(ServletContextEvent sce) {
12        System.out.println("SpringBoot.Servlet应用销毁");
13    }
14 }
```

```
1 @Configuration
2 public class MyServletConfig {
3     //注册Listener
4     @Bean
5     public ServletListenerRegistrationBean myListener() {
6         return new ServletListenerRegistrationBean(new MyListener());
7     }
8 }
```



## 8.2 分析自动注册的SpringMVC前端控制器

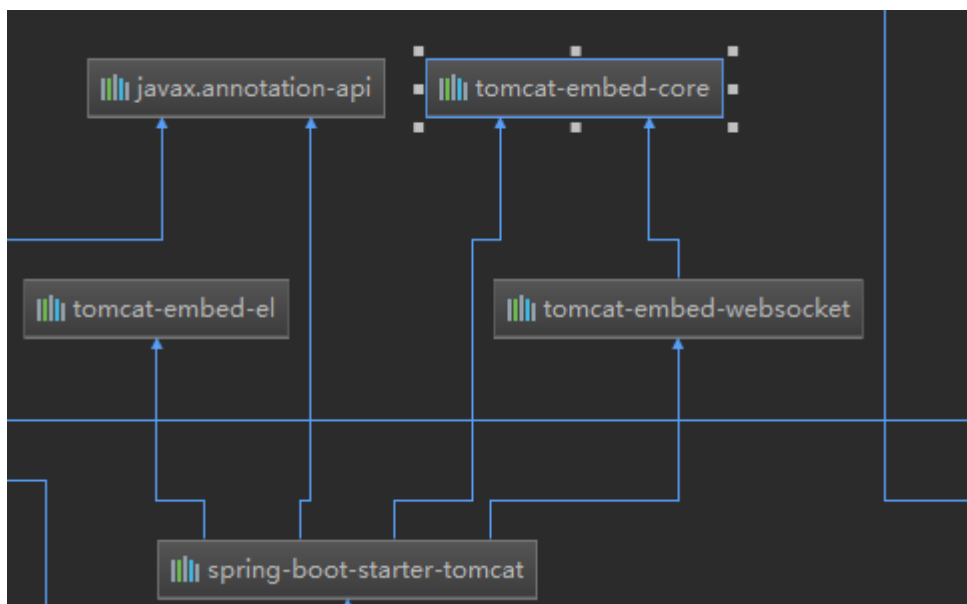
SpringBoot 在 `DispatcherServletAutoConfiguration` 自动配置中，帮我们注册 SpringMVC 的前端控制器：  
`DispatcherServlet`

```
1 @Bean( name = {"dispatcherServletRegistration"})
2 @ConditionalOnBean(value = {DispatcherServlet.class},name={"dispatcherServlet"})
3 //注册了前端控制器
4 public DispatcherServletRegistrationBean
5     dispatcherServletRegistration(DispatcherServlet dispatcherServlet) {
6     DispatcherServletRegistrationBean registration =
7         new DispatcherServletRegistrationBean(dispatcherServlet,
8         // / 拦截所有请求(包括静态资源);但不会拦截jsp请求; 而 /* 会拦截jsp
9         this.webMvcProperties.getServlet().getPath());// /
10     registration.setName("dispatcherServlet");
11     registration.setLoadOnStartup(this.webMvcProperties.getServlet().getLoadOnStartup());
12     if (this.multipartConfig != null) {
13         registration.setMultipartConfig(this.multipartConfig);
14     }
15
16     return registration;
17 }
18 }
```

## 8.3 修改Servlet容器配置

参考 pom.xml 可知，SpringBoot 默认使用 Tomcat 作为嵌入式的 Servlet 容器，

SpringBoot2.1版本默认使用的是Tomcat9.0.12版本的容器



### 8.3.1 修改Servlet容器配置

- 方式1：在 application 全局配置文件中，修改 `server` 开头有关的配置【ServerProperties】



```
1 #项目服务相关
2 server.port=8080
3
4 #修改Servlet相关配置 server.servlet.xxx
5 server.servlet.context-path=/servlet
6
7 #修改Tomcat相关配置 server.tomcat.xxx
8 server.tomcat.uri-encoding=utf-8
```

### 8.3.2 使用定制器修改Servlet容器配置 ( spring1.x与spring2.x不同 )

#### Spring Boot 1.x :

通过实现 嵌入式的Servlet容器定制器 `EmbeddedServletContainerCustomizer` 的 `customize` 方法, 来修改Servlet容器的配置

```
1 @Configuration
2 public class MyServletConfig {
3     //Spring Boot 1.x :
4     @Bean //一定要将这个定制器加入到容器中
5     public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer(){
6         return new EmbeddedServletContainerCustomizer() {
7             //定制嵌入式的Servlet容器相关的规则
8             @Override
9             public void customize(ConfigurableEmbeddedServletContainer container) {
10                 container.setPort(8083);
11             }
12         };
13     }
14 }
```

#### Spring Boot 2.x :

在2.x版本改为实现 `WebServerFactoryCustomizer` 接口的 `customize` 方法

```
1 //springboot2.x
2 @Bean
3 public WebServerFactoryCustomizer webServerFactoryCustomizer() {
4     return new WebServerFactoryCustomizer() {
5         @Override
6         public void customize(WebServerFactory factory) {
7             ConfigurableWebServerFactory serverFactory = (ConfigurableWebServerFactory)factory;
8             serverFactory.setPort(8081);
9         }
10     };
11 }
```

## 8.4 切换为其他嵌入式Servlet容器

- SpringBoot 默认针对Servlet容器提供以下支持：
  - Tomcat ( 默认使用 )
  - Jetty : 支持长连接项目 ( 如: 聊天页面 )
  - Undertow : 不支持 JSP , 但是并发性能高 , 是高性能非阻塞的容器
- 默认Tomcat容器

```
1 在spring-boot-starter-web启动器中默认引入了tomcat容器
2 <dependency>
3   <groupId>org.springframework.boot</groupId>
4   <artifactId>spring-boot-starter-tomcat</artifactId>
5   <version>2.1.0.RELEASE</version>
6   <scope>compile</scope>
7 </dependency>
```

- 切换 Jetty 容器

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <!-- 排除tomcat容器 -->
5   <exclusions>
6     <exclusion>
7       <artifactId>spring-boot-starter-tomcat</artifactId>
8       <groupId>org.springframework.boot</groupId>
9     </exclusion>
10  </exclusions>
11 </dependency>
12
13 <!--引入其他的Servlet容器-->
14 <dependency>
15   <artifactId>spring-boot-starter-jetty</artifactId>
16   <groupId>org.springframework.boot</groupId>
17 </dependency>
```

- 切换 Undertow 容器

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <!-- 排除tomcat容器 -->
5   <exclusions>
6     <exclusion>
7       <artifactId>spring-boot-starter-tomcat</artifactId>
8       <groupId>org.springframework.boot</groupId>
9     </exclusion>
10  </exclusions>
```

```
11 </dependency>
12
13 <!--引入其他的undertow容器-->
14 <dependency>
15   <artifactId>spring-boot-starter-undertow</artifactId>
16   <groupId>org.springframework.boot</groupId>
17 </dependency>
```

## 第9章 使用外置Servlet容器\_Tomcat9.x

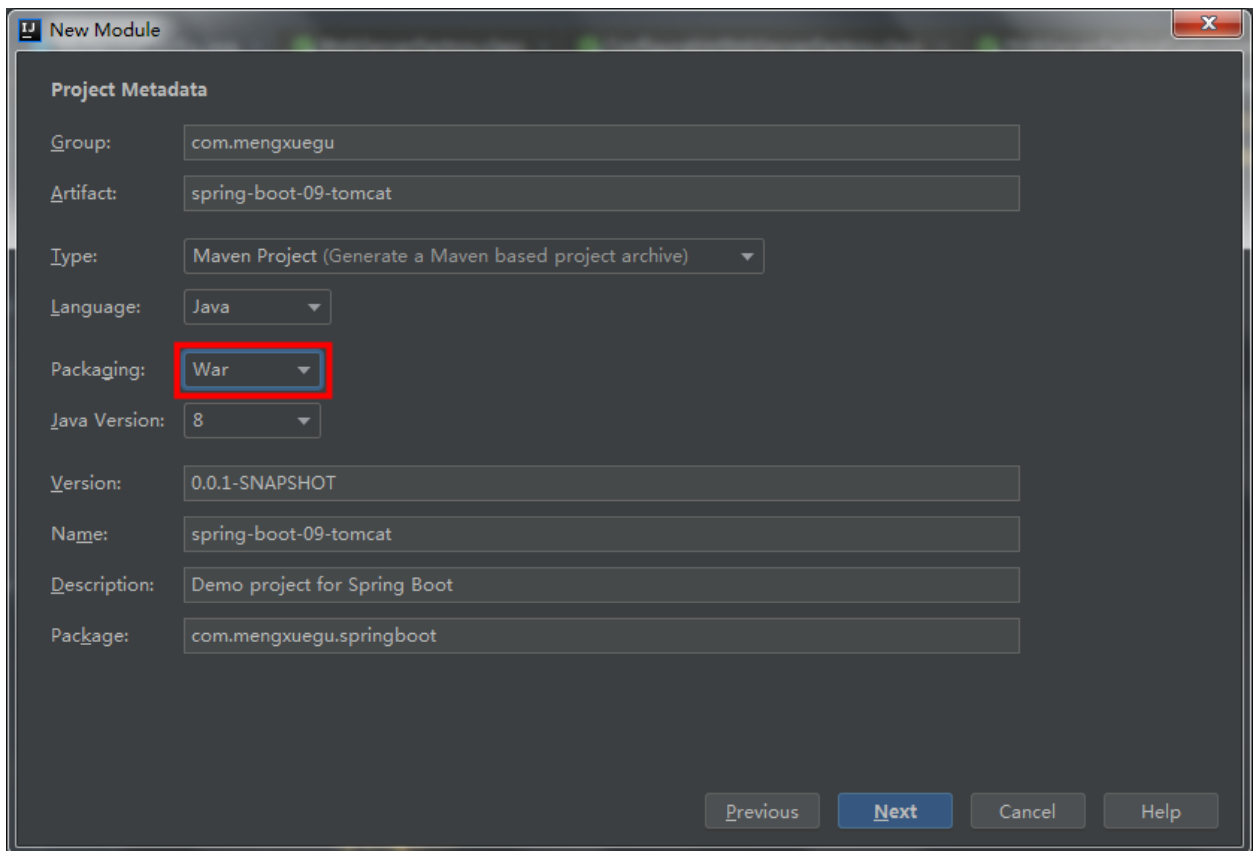
### 9.1 比较嵌入式与外置Servlet容器

- 嵌入式Servlet容器：运行启动类就可启动，或将项目打成可执行的 jar 包
  - 优点：简单、快捷；
  - 缺点：默认不支持JSP、优化定制比较复杂使用定制器, 还需要知道 每个功能 的底层原理
- 外置Servlet容器：配置 Tomcat, 将项目部署到Tomcat中运行

### 9.2 使用Tomcat9.x作为外置Servlet容器

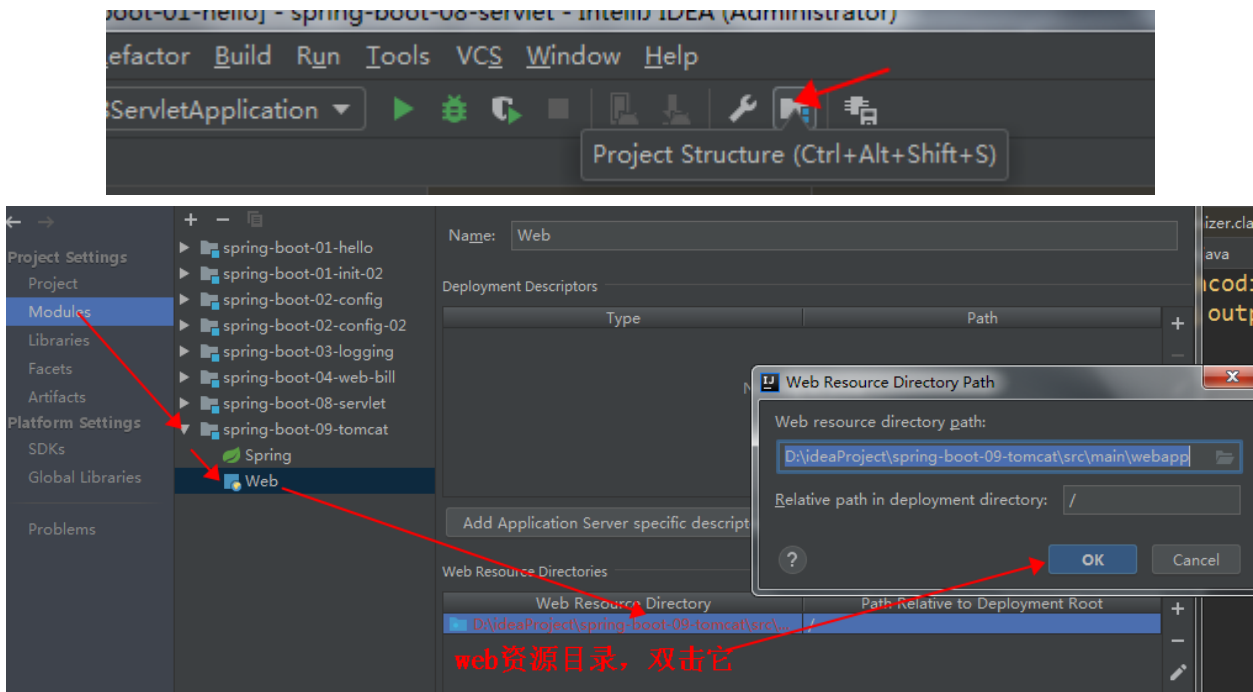
操作步骤:

1. 必须创建一个 war 类型项目

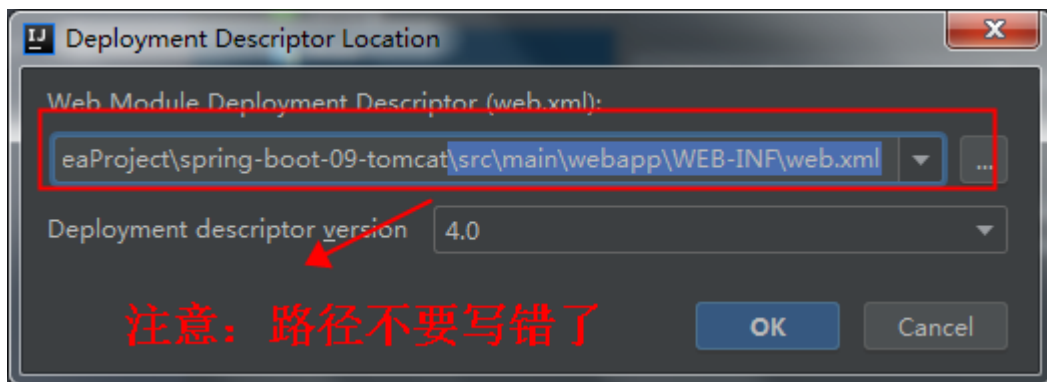
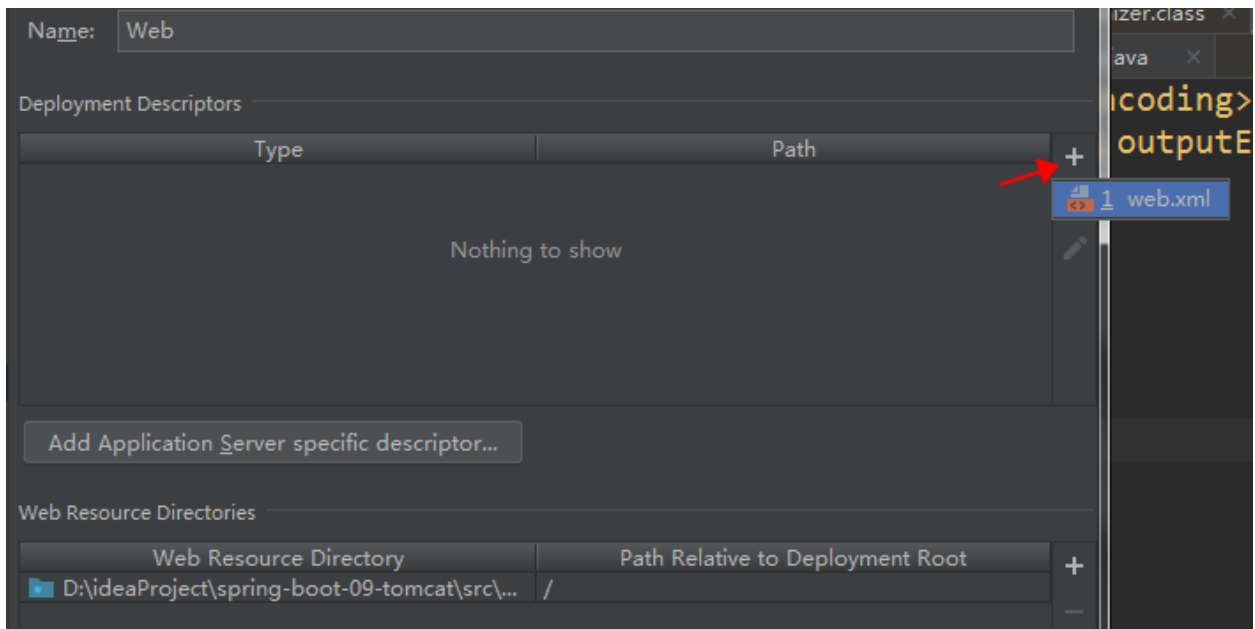


## 2. idea 上指定web.xml 与 修改好目录结构

- 指定webapp目录

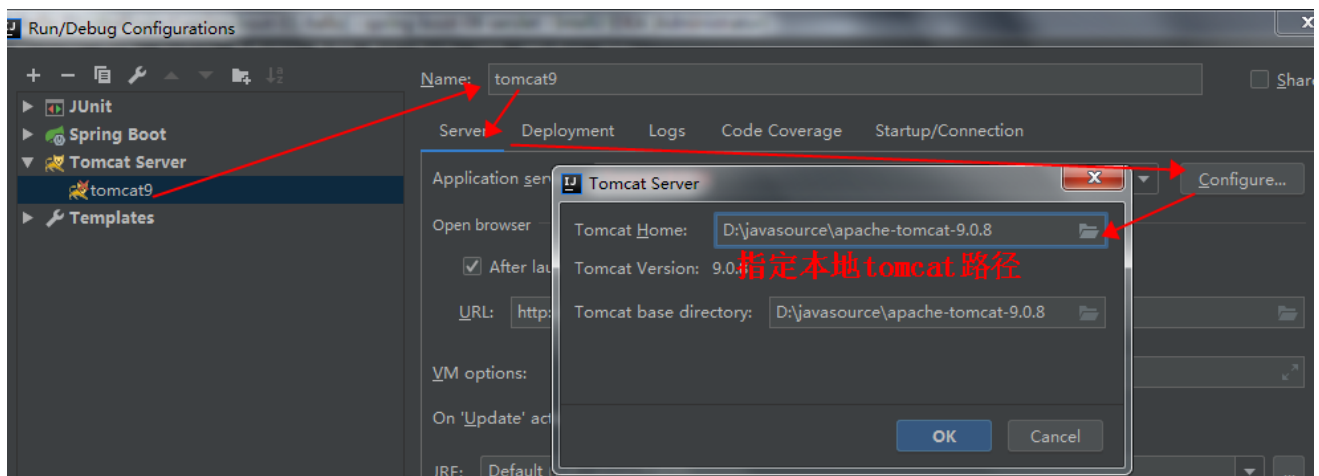
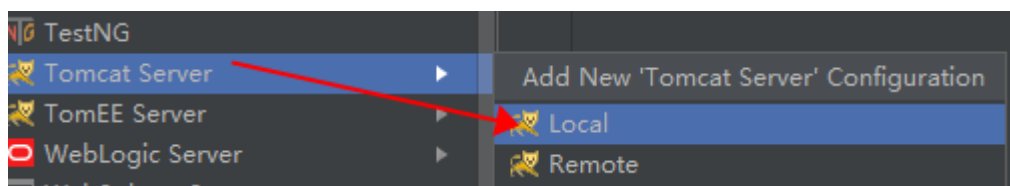
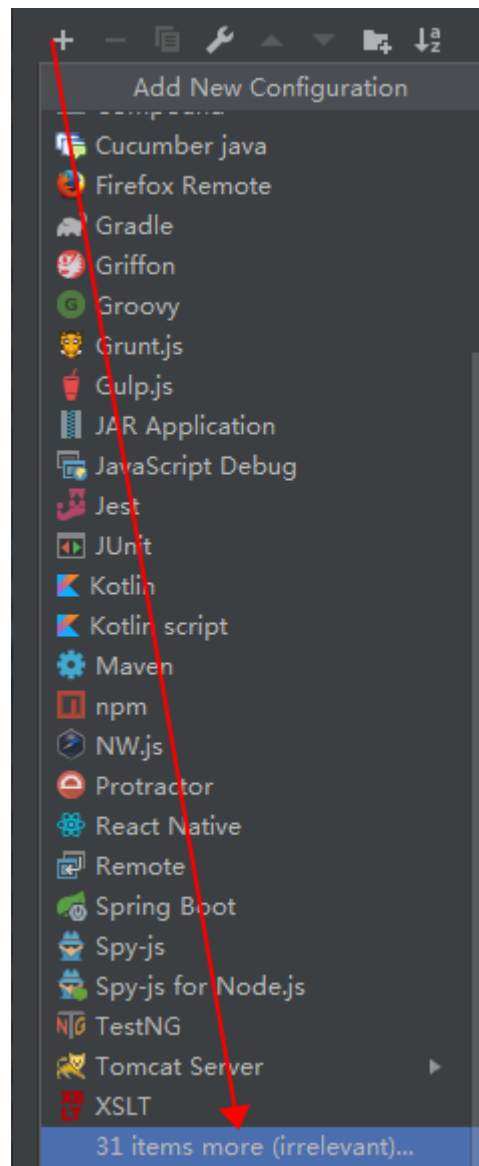


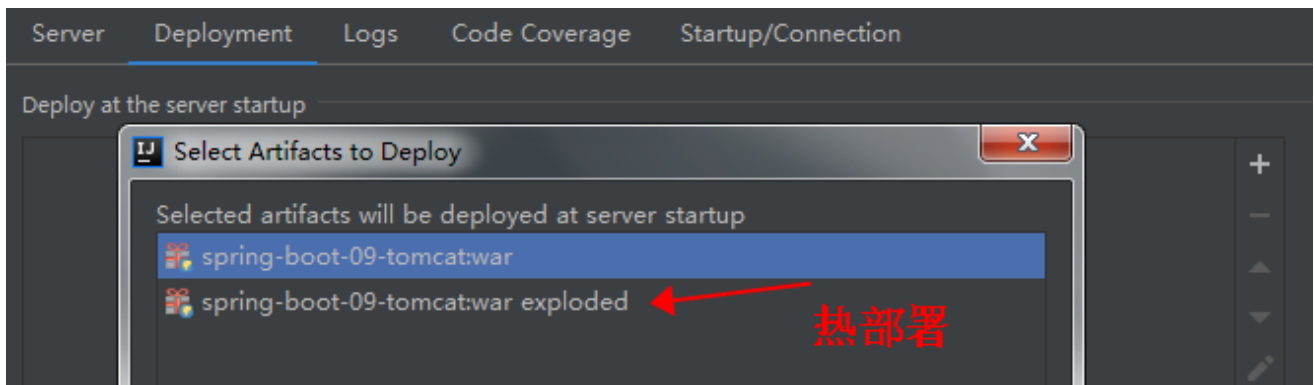
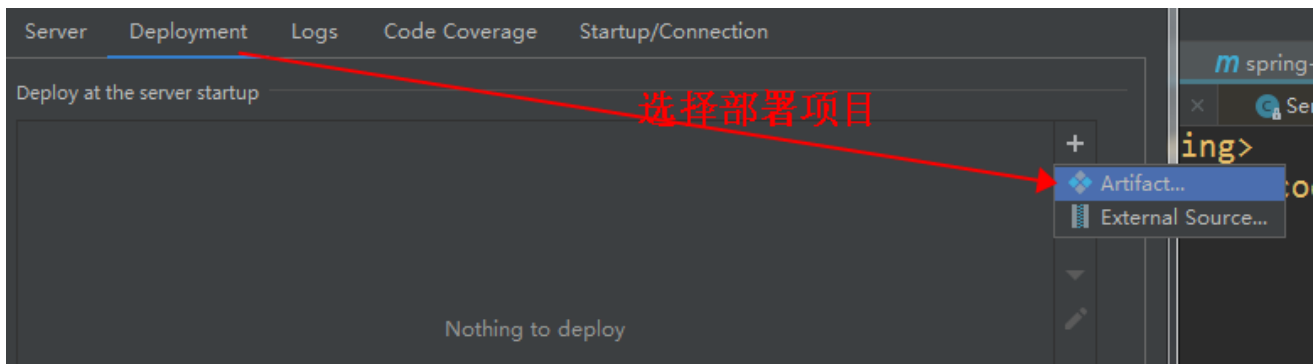
- 指定web.xml位置



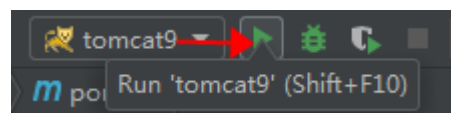
1 src\main\webapp\WEB-INF\web.xml

- 添加外置tomcat





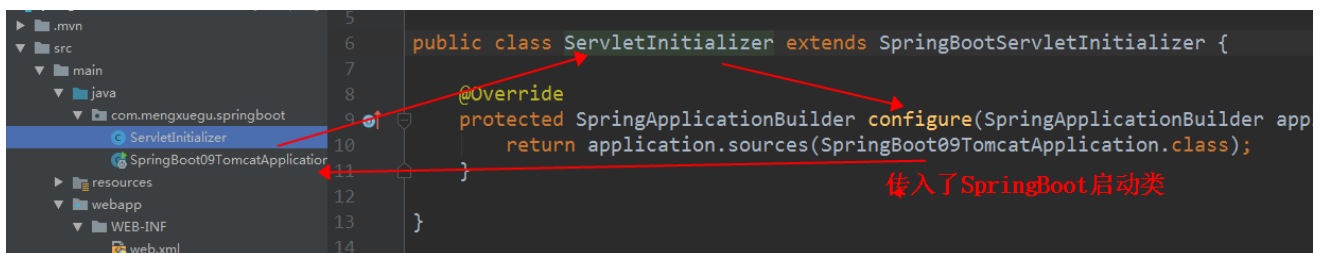
启动



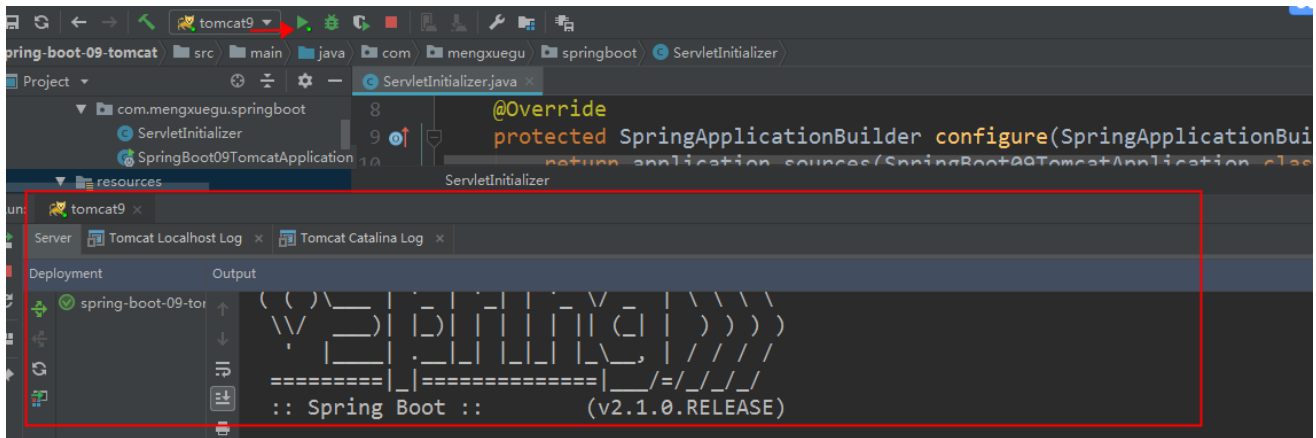
3. 在 pom.xml 将嵌入式的Tomcat指定为provided (Spring初始化器已经默认指定了)

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-tomcat</artifactId>
4   <scope>provided</scope>
5 </dependency>
```

4. Spring初始化器 自动创建了 `SpringBootServletInitializer` 的子类，调用 `configure` 方法

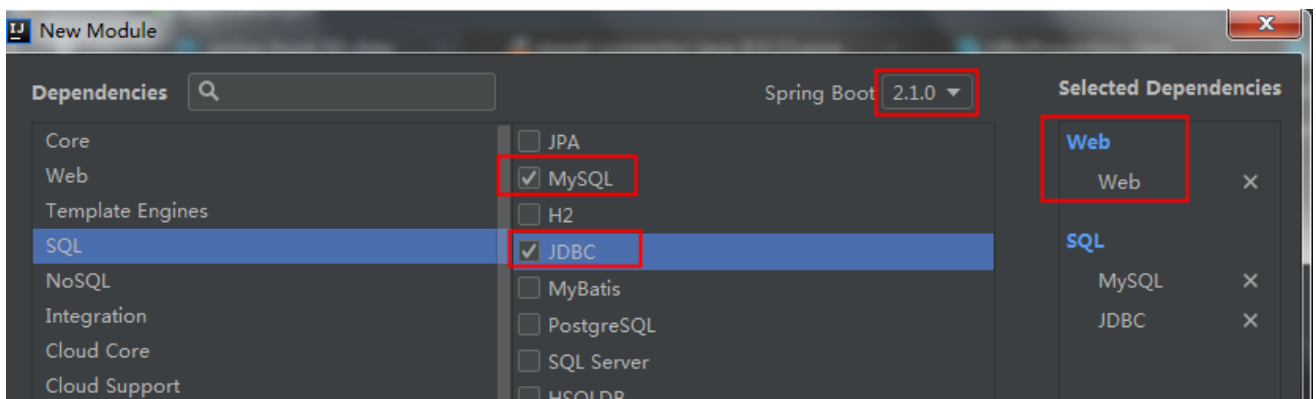


5. 直接开发项目功能即可，然后启动tomcat即可访问



## 第10章 SpringBoot 数据访问操作

### 10.1 整合 JDBC 实战



#### 10.1.1 JDBC相关配置

- pom.xml

```
1 <!--mysql驱动包-->
2 <dependency>
3   <groupId>mysql</groupId>
4   <artifactId>mysql-connector-java</artifactId>
5   <scope>runtime</scope>
6 </dependency>
7 <!--jdbc启动器-->
8 <dependency>
9   <groupId>org.springframework.boot</groupId>
10  <artifactId>spring-boot-starter-jdbc</artifactId>
11 </dependency>
```



- application.yml

注意：mysql 8.x版本驱动包，要使用 `com.mysql.cj.jdbc.Driver` 作为驱动类

```
1 spring:
2   datasource:
3     username: root
4     password: root
5     #使用 MySQL连接驱动是8.0以上，需要在Url后面加上时区，GMT%2B8代表中国时区，不然报时区错误
6     url: jdbc:mysql://127.0.0.1:3306/jdbc?serverTimezone=GMT%2B8
7     # 注意：新版本驱动包，要使用以下类作为驱动类
8     driver-class-name: com.mysql.cj.jdbc.Driver
```

- 测试类

```
1 @RunWith(SpringRunner.class)
2 @SpringBootTest
3 public class SpringBootTestApplicationTests {
4     @Autowired
5     DataSource datasource;
6
7     @Test
8     public void contextLoads() throws SQLException {
9         // 默认采用的数据源连接池：com.zaxxer.hikari.HikariDataSource
10        System.out.println("datasource: " + datasource.getClass());
11        Connection connection = datasource.getConnection();
12        System.out.println(connection);
13        connection.close();
14    }
15
16 }
```

- 运行结果

- SpringBoot 默认采用的数据源连接池是： `com.zaxxer.hikari.HikariDataSource`
- 数据源相关配置都在 `DataSourceProperties` 中；

## 10.1.2 常见错误

1. 以下说明mysql服务器没有启动，需要启动mysql服务，你用navicat连接试试看是否可以连接，不可以说明没有启动

The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.

1. 时区异常：

需要配置文件中指定时区：`jdbc:mysql://127.0.0.1:3306/jdbc?serverTimezone=GMT%2B8`

The server time zone value 'ÖÐ'±±±±±±±±±± is unrecognized or represents more than one

### 10.1.3 JDBC自动配置原理：

1. 支持的数据源, 提供了 Hikari.class, Tomcat.class, Dbc2.class, Generic.class

各种连接池数据源相关配置：DataSourceConfiguration

可以通过spring.datasource.type 修改数据源

```
1 @Configuration
2 @Conditional({DataSourceAutoConfiguration.PooledDataSourceCondition.class})
3 @ConditionalOnMissingBean({DataSource.class, XADataSource.class})
4 //提供了 Hikari.class, Tomcat.class, Dbc2.class, Generic.class
5 @Import({Hikari.class, Tomcat.class, Dbc2.class, Generic.class, DataSourceJmxConfiguration.class})
6 protected static class PooledDataSourceConfiguration {
7     protected PooledDataSourceConfiguration() {
8     }
9 }
```

2. JdbcTemplateAutoConfiguration 自动配置类提供了 JdbcTemplate 操作数据库

```
1 @Controller
2 public class ProviderController {
3     @Autowired
4     JdbcTemplate jdbcTemplate;
5
6     @ResponseBody
7     @GetMapping("/providers")
8     public Map<String, Object> list() {
9         List<Map<String, Object>> maps =
10             jdbcTemplate.queryForList("select * from provider");
11         System.out.println(maps);
12         return maps.get(0);
13     }
14 }
```

## 10.2 高级配置 Druid 连接池与监控管理

Hikari 性能上比 Druid 更好，但是 Druid 有配套的监控安全管理功能

### 10.2.1 整合 Druid 操作步骤

1. 引入 Druid 依赖

```
1 <dependency>
2     <groupId>com.alibaba</groupId>
3     <artifactId>druid</artifactId>
4     <version>1.1.12</version>
5 </dependency>
```

## 2. Druid 全局配置

```
1  spring
2  datasource:
3  # 数据源基本配置
4  username: root
5  password: root
6  url: jdbc:mysql://127.0.0.1:3306/jdbc?serverTimezone=GMT%2B8
7  # 8.x版本驱动包，要使用以下类作为驱动类
8  driver-class-name: com.mysql.cj.jdbc.Driver
9  # 指定 Druid 数据源
10 type: com.alibaba.druid.pool.DruidDataSource
11
12 # 数据源其他配置, DataSourceProperties中没有相关属性,默认无法绑定
13 initialSize: 8
14 minIdle: 5
15 maxActive: 20
16 maxWait: 60000
17 timeBetweenEvictionRunsMillis: 60000
18 minEvictableIdleTimeMillis: 300000
19 validationQuery: SELECT 1 FROM DUAL
20 testWhileIdle: true
21 testOnBorrow: false
22 testOnReturn: false
23 poolPreparedStatements: true
24 # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
25 filters: stat,wall,logback
26 maxPoolPreparedStatementPerConnectionSize: 25
27 useGlobalDataSourceStat: true
28 connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

3. 通过测试类测试，发现数据源已经切换为 DruidDataSource，但是配置中的属性没有与它绑定上

4. 自定义配置类，将配置中属性与 DruidDataSource 属性绑

```
1  package com.mengxuegu.springboot.config;
2
3  /**
4   * Druid 配置类
5   * @Author: 梦学谷
6   */
7  @Configuration
8  public class DruidConfig {
9
10     //绑定数据源配置
11     @ConfigurationProperties(prefix = "spring.datasource")
12     @Bean
13     public DataSource druid() {
14         return new DruidDataSource();
15     }
16
17 }
```

## 10.2.2 配置Druid监控

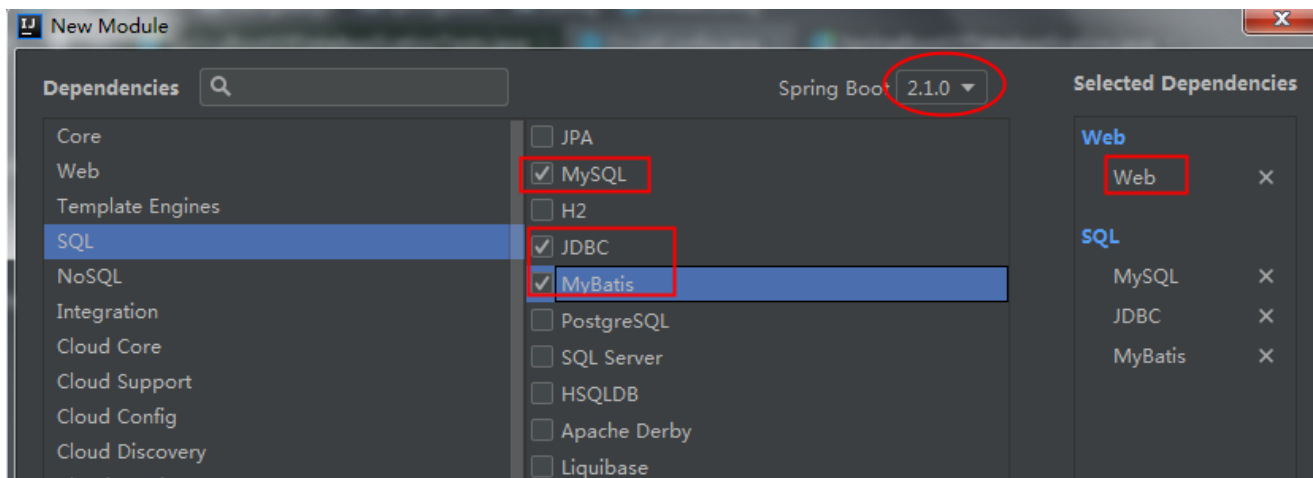
```
1 package com.mengxuegu.springboot.config;
2
3 /**
4  * Druid 配置类
5  * @Author: 梦学谷
6  */
7 @Configuration
8 public class DruidConfig {
9
10     //绑定数据源配置
11     @ConfigurationProperties(prefix = "spring.datasource")
12     @Bean
13     public DataSource druid() {
14         return new DruidDataSource();
15     }
16
17     /**
18     * 配置Druid监控
19     * 1. 配置一个管理后台的Servlet
20     * 2. 配置一个监控的filter
21     */
22     @Bean // 1. 配置一个管理后台的Servlet
23     public ServletRegistrationBean statViewServlet() {
24         //StatViewServlet是 配置管理后台的servlet
25         ServletRegistrationBean<StatViewServlet> bean =
26             new ServletRegistrationBean<>(new StatViewServlet(), "/druid/*");
27         //配置初始化参数
28         Map<String, String> initParam = new HashMap<>();
29         //访问的用户名密码
30         initParam.put(StatViewServlet.PARAM_NAME_USERNAME, "root");
31         initParam.put(StatViewServlet.PARAM_NAME_PASSWORD, "123");
32         //允许访问的ip，默认所有ip访问
33         initParam.put(StatViewServlet.PARAM_NAME_ALLOW, "");
34         //禁止访问的ip
35         initParam.put(StatViewServlet.PARAM_NAME_DENY, "192.168.10.1");
36
37         bean.setInitParameters(initParam);
38         return bean;
39     }
40
41     //2. 配置一个监控的filter
42     @Bean
43     public FilterRegistrationBean filter() {
44         FilterRegistrationBean<Filter> bean = new FilterRegistrationBean<>();
45         bean.setFilter(new WebStatFilter());
46
47         //配置初始化参数
48         Map<String, String> initParam = new HashMap<>();
49         //排除请求
```

```
50     initParam.put(WebStatFilter.PARAM_NAME_EXCLUSIONS, "*.js,*.css,/druid/*");
51
52     //拦截所有请求
53     bean.setUrlPatterns(Arrays.asList("/*"));
54
55     return bean;
56 }
57
58 }
```

## 10.3 整合 MyBatis3.x 注解版本实战

### 10.3.1 搭建 MyBatis 环境

- 创建Module



- 导入 Druid 数据源依赖，创建后自动会引入 MyBatis 启动器，是由 MyBatis 官方提供的

```
1  <!--导入 mybatis 启动器-->
2  <dependency>
3      <groupId>org.mybatis.spring.boot</groupId>
4      <artifactId>mybatis-spring-boot-starter</artifactId>
5      <version>1.3.2</version>
6  </dependency>
7  <!--druid数据源-->
8  <dependency>
9      <groupId>com.alibaba</groupId>
10     <artifactId>druid</artifactId>
11     <version>1.1.12</version>
12 </dependency>
```

- 配置 Druid 数据源（application.yml修改为mybatis库）与监控 [参考10.2章节]
- 创建 mybatis 库与导入表和数据、实体类

## 10.3.2 注解版 MyBatis 操作

```
1 package com.mengxuegu.springboot.mapper;
2
3 import com.mengxuegu.springboot.entities.Provider;
4 import org.apache.ibatis.annotations.*;
5
6 /**
7  * 使用Mybatis注解版本
8  * @Author: 梦学谷
9  */
10 // @Mapper 指定这是操作数据的Mapper
11 public interface ProviderMapper {
12
13     @Select("select * from provider where pid=#{pid}")
14     Provider getProviderByPid(Integer pid);
15
16     // useGeneratedKeys 是否使用自增主键，keyProperty 指定实体类中的哪一个属性封装主键值
17     @Options(useGeneratedKeys = true, keyProperty = "pid")
18     @Insert("insert into provider(providerName) values(#{providerName})")
19     int addProvider(Provider provider);
20
21     @Delete("delete from provider where pid=#{pid}")
22     int deleteProviderByPid(Integer pid);
23
24     @Update("update provider set providerName=#{providerName} where pid=#{pid}")
25     int updateProvider(Provider provider);
26 }
27
```

**注：**上面@Insert插入数据时，使用 @Options 接收插入的主键值：

useGeneratedKeys 是否自增主键，keyProperty 指定实体中哪个属性封装主键

```
@Options(useGeneratedKeys = true, keyProperty = "pid")
```

```
1 @Controller
2 public class ProviderController {
3
4     @Autowired
5     ProviderMapper providerMapper;
6
7     @ResponseBody
8     @GetMapping("/provider/{pid}")
9     public Provider getProvider(@PathVariable("pid") Integer pid) {
10         Provider providerByPid = providerMapper.getProviderByPid(pid);
11         return providerByPid;
12     }
13
14     @ResponseBody
15     @GetMapping("/provider")
16     public Provider addProvider(Provider provider) {
17
18     }
```

```
17     providerMapper.addProvider(provider);
18     return provider;
19 }
20
21
22 }
```

- 自定义MyBatis配置类, 替代mybatis配置文件

- 开启驼峰命名方式, 使用, 不然 provider\_code 不会自动转成 providerCode

```
1
2 import org.mybatis.spring.boot.autoconfigure.ConfigurationCustomizer;
3 import org.springframework.context.annotation.Bean;
4 import org.apache.ibatis.session.Configuration;
5 /**
6  * MyBatis注解版-配置类替换配置文件
7  * @Author: 梦学谷
8  */
9 @org.springframework.context.annotation.Configuration
10 public class MyBatisConfig {
11
12     @Bean
13     public ConfigurationCustomizer configurationCustomizer() {
14         return new ConfigurationCustomizer(){
15             @Override
16             public void customize(Configuration configuration) {
17                 //开启驼峰命名方式
18                 configuration.setMapUnderscoreToCamelCase(true);
19             }
20         };
21     }
22
23 }
24
```

- 使用 @MapperScan("包名") 自动装配指定包下所有Mapper, 省得在每个Mapper接口上写 @Mapper

```
1 //会自动装配指定包下面所有Mapper, 省得在每个Mapper上面写@Mapper
2 @MapperScan("com.mengxuegu.springboot.mapper")
3 @SpringBootApplication
4 public class SpringBoot08DataMybatisApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(SpringBoot08DataMybatisApplication.class, args);
8     }
9 }
```

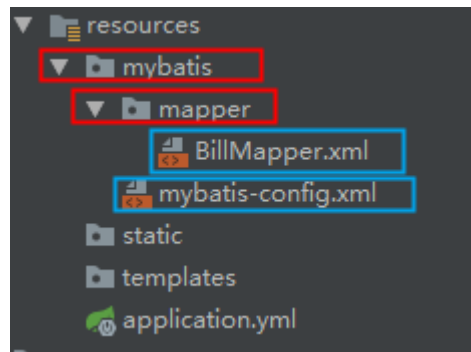
## 10.4 整合 MyBatis3.x 配置文件版实战

Mybatis官网: <http://www.mybatis.org/mybatis-3/zh/index.html>

- Mapper接口

```
1  ...
2  /**
3   * MyBatis 配置文件版
4   * @Author: 梦学谷
5   */
6  //@Mapper 或 @MapperScan 扫描Mapper接口装配到容器中
7  public interface BillMapper {
8      Bill getBillByBid(Integer bid);
9
10     int insertBill(Bill bill);
11 }
```

- 在 resources 创建以下目录和核心配置文件与Mapper映射文件



- mybatis 核心配置文件

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE configuration
3      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-config.dtd">
5  <configuration>
6      <!--mybatis核心配置文件-->
7
8  </configuration>
```

- BillMapper 映射文件

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE mapper
3      PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4      "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5  <mapper namespace="com.mengxuegu.springboot10datamybatis.entities">
6
7      <select id="getBillByBid" resultType="com.mengxuegu.springboot10datamybatis.entities.Bill">
8          select * from bill where bid = #{bid}
```



```
9     </select>
10
11     <insert id="addBill">
12         insert into bill(bill_code, bill_name) values(#{billCode}, #{billName})
13     </insert>
14
15 </mapper>
```

- application.yml 中指定配置文件路径

```
1 # Mybatis相关配置
2 mybatis:
3   #核心配置文件路径
4   config-location: classpath:mybatis/mybatis-config.xml
5   #映射配置文件路径
6   mapper-locations: classpath:mybatis/mapper/*.xml
```

- 创建 BillController 来测试

```
1 @Controller
2 public class BillController {
3
4     @Autowired
5     BillMapper billMapper;
6
7     @ResponseBody
8     @GetMapping("/bill/{bid}")
9     public Bill getBill(@PathVariable Integer bid) {
10         return billMapper.getBillByBid(bid);
11     }
12
13     @ResponseBody
14     @GetMapping("/bill")
15     public Bill addBill(Bill bill) {
16         billMapper.addBill(bill);
17         return bill;
18     }
19 }
```

访问 <http://localhost:8080/bill/1> 后发现 billCode、billName等没有获取到，需要配置文件中开启驼峰命名

```
1 {"bid":1,"billCode":null,"billName":null,"billCom":null,"billNum":null,"money":400000.0,"provider":null,"pay":null,"createDate":null}
```

- mybatis-config.xml 开启驼峰命名

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE configuration
3     PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-config.dtd">
5 <configuration>
6     <!--mybatis核心配置文件-->
7     <settings>
8         <!--开启驼峰命名-->
9         <setting name="mapUnderscoreToCamelCase" value="true"/>
10    </settings>
11 </configuration>
```

- 控制台打印sql语句

```
1 # 打印sql
2 logging:
3     level:
4         com.mengxuegu.springboot10datamybatis.mapper : debug
```

## 10.5 整合 Spring Data JPA 实战

### 10.5.1 什么是 Spring Data

Spring Data 是 Spring Boot 底层默认进行数据访问的技术，为了简化构建基于 Spring 框架应用的数据访问技术，包括非关系数据库、Map-Reduce 框架、云数据服务等；另外也包含对关系数据库的访问支持。

Spring Data 包含多个模块：

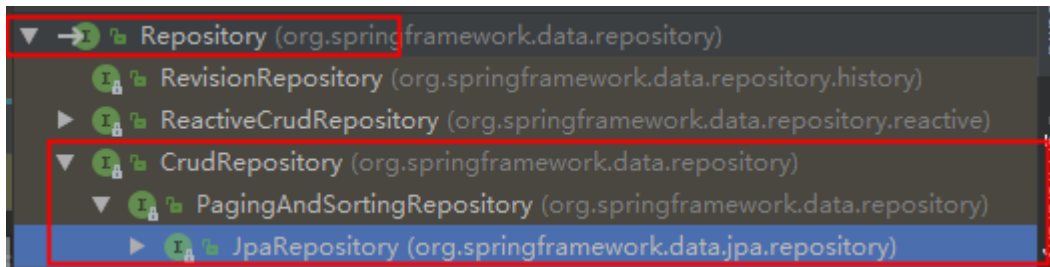
- Spring Data Commons 提供共享的基础框架，适合各个子项目使用，支持跨数据库持久化
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- Spring Data for Apache Cassandra
- Spring Data for Apache Geode
- Spring Data for Apache Solr
- Spring Data for Pivotal GemFire
- Spring Data Couchbase (community module)
- Spring Data Elasticsearch (community module)
- Spring Data Neo4j (community module)

### Spring Data 特点

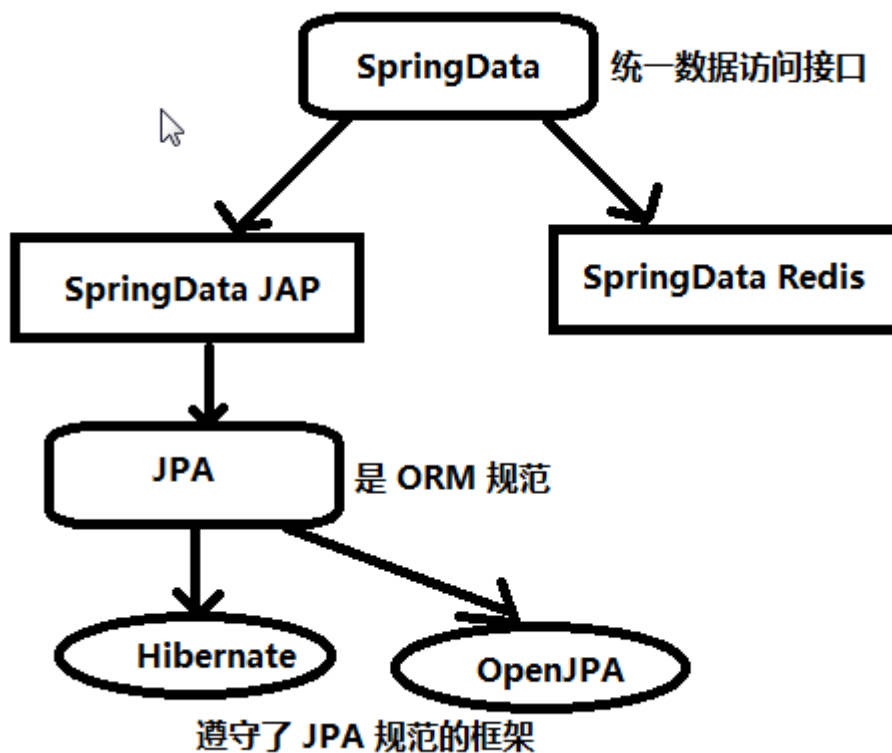
- Spring Data 项目为大家提供统一的API来对不同数据访问层进行操作；

## Spring Data 统一的核心接口

1. `Repository` : 统一的根接口，其他接口继承该接口
2. `CrudRepository` : 基本的增删改查接口,提供了最基本的对实体类CRUD操作
3. `PagingAndSortingRepository` : 增加了分页和排序操作
4. `JpaRepository` : 增加了批量操作，并重写了父接口一些方法的返回类型
5. `JpaSpecificationExecutor` : 用来做动态查询，可以实现带查询条件的分页(不属于Repository体系，支持 JPA Criteria 查询相关的方法)



## Spring Data JPA、JPA与Hibernate 关系

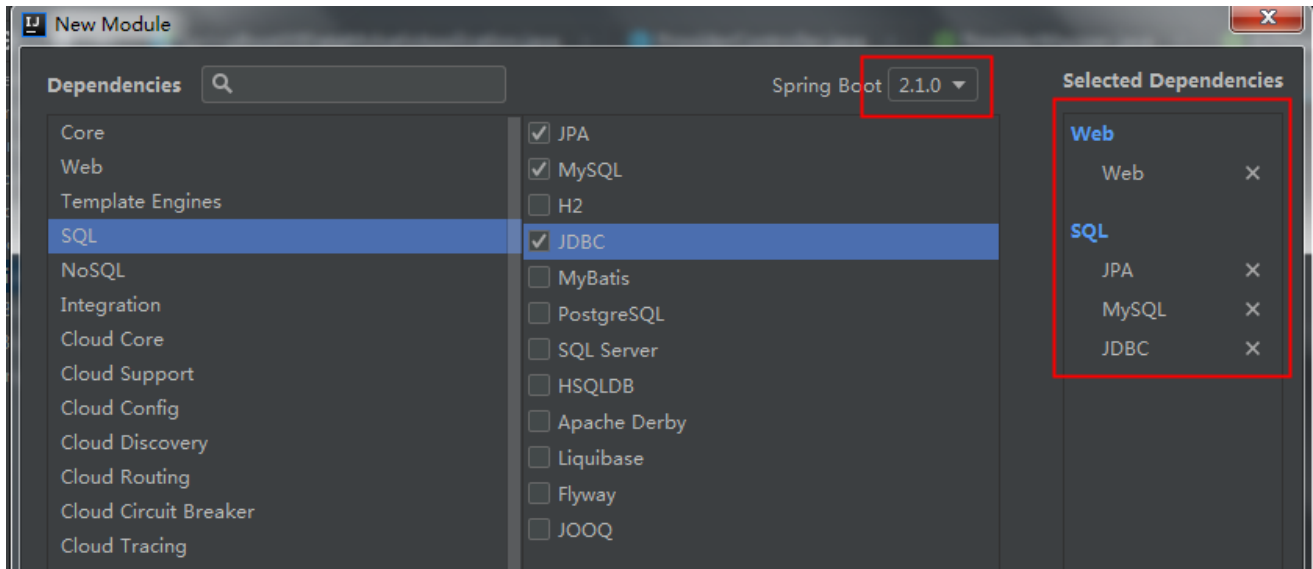


- JPA是一种规范，而Hibernate是实现这种规范的底层实现，Spring Data JPA对持久化接口 JPA 再抽象一层，针对持久层业务再进一步统一简化。

## 10.5.2 整合 Spring Data JPA 实战

JPA的底层遵守是ORM(对象关系映射)规范，因此JPA其实也就是java实体对象和关系型数据库建立起映射关系，通过面向对象编程的思想操作关系型数据库的规范。

### 1、创建Module



### 2. 添加数据源, 新建 jpa 数据库

```
1 spring
2 datasource:
3   # 数据源基本配置
4   username: root
5   password: root
6   url: jdbc:mysql://127.0.0.1:3306/jpa?serverTimezone=GMT%2B8
7   # 8.x版本驱动包, 要使用以下类作为驱动类
8   driver-class-name: com.mysql.cj.jdbc.Driver
```

### 3. 创建实体类，并使用JPA注解进行配置映射关系

- 类上使用 JPA注解 `@Entity` 标注，说明它是和数据表映射的类；`@Table(name="表名")` 指定对应映射的表名，省略默认表名就是类名。
- `@Id` 标识主键，`@GeneratedValue(strategy = GenerationType.IDENTITY)` 标识自增长主键
- `@Column` 标识字段

```
1 import javax.persistence.*;
2
3 //使用JPA注解进行配置映射关系
4 @Entity //说明它是和数据表映射的类
5 @Table(name = "tbl_user") //指定对应映射的表名，省略默认表名就是类名
6 public class User {
7   @Id //标识主键
8   @GeneratedValue(strategy = GenerationType.IDENTITY) //标识自增长主键
9   private Integer id;
10
11   @Column(name = "user_name",length = 5) //这是和数据表对应的一个列
```

```
12 private String userName;
13
14 @Column //省略默认列名就是属性名
15 private String password;
16
17 setter,getter
18 }
```

4. 创建 `UserRepository` 接口继承 `JpaRepository`，就会crud及分页等基本功能

```
1 package com.mengxuegu.springboot.dao;
2
3 import com.mengxuegu.springboot.entity.User;
4 import org.springframework.data.jpa.repository.JpaRepository;
5
6 /**
7  * 自定义接口继承JpaRepository，就会crud及分页等基本功能
8  * @Author: 梦学谷 www.mengxuegu.com
9  */
10 //指定的泛型<操作的实体类，主键的类型>
11 public interface UserRepository extends JpaRepository<User, Integer> {
12 }
```

5. JPA 配置在全局配置文件中添加（spring.jpa.\* 开头）

```
1 spring
2 datasource:
3   # 数据源基本配置
4   username: root
5   password: root
6   url: jdbc:mysql://127.0.0.1:3306/jpa
7   # 8.x版本驱动包，要使用以下类作为驱动类
8   driver-class-name: com.mysql.cj.jdbc.Driver
9
10 # jpa相关配置 spring.jpa.*
11 jpa:
12   # 控制台显示SQL
13   showSql: true
14   hibernate:
15     # 会根据映射实体类自动创建或更新数据表
16     ddl-auto: update
17   # 默认创建表类型是MyISAM，是非事务安全的，所以无法实现事物回滚
18   # 指定如下方言：创建的表类型是InnoDB，才可以进行对事物的回滚。
19   database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
```

6. 测试方法

```
1 @RestController
2 public class UserController {
3
4   @Autowired
```

```
5 UserRepository userRepository;  
6  
7 @GetMapping("/user/{id}")  
8 public User getUserById(@PathVariable("id") Integer id) {  
9     return userRepository.findById(id).get();  
10 }  
11  
12 @GetMapping("/user")  
13 public User addUser(User user) {  
14     return userRepository.save(user);  
15 }  
16 }
```

## 10.6 Spring Boot中的事务管理

### 10.6.1 什么是事务

我们在开发企业应用时，对于业务人员的一个操作实际是对数据读写的多步操作的结合。由于数据操作在顺序执行的过程中，任何一步操作都有可能发生异常，异常会导致后续操作无法完成，此时由于业务逻辑并未正确的完成，之前成功操作数据的并不可靠，需要在这种情况下进行回退。

事务的作用就是为了保证用户的每一个操作都是可靠的，事务中的每一步操作都必须成功执行，只要有发生异常就回退到事务开始未进行操作的状态。

事务管理是Spring框架中最为常用的功能之一，我们在使用Spring Boot开发应用时，大部分情况下也都需要使用事务。

### 10.6.2 事务管理操作

在Spring Boot中，当我们使用了spring-boot-starter-jdbc或spring-boot-starter-data-jpa依赖的时候，框架会自动默认分别注入DataSourceTransactionManager或JpaTransactionManager。所以我们不需要任何额外配置就可以用@Transactional注解进行事务的使用。

1. 强调 Hibernate 在创建表时，

- 默认创建表类型是MyISAM,是非事务安全的，所以无法实现事物回滚; Innodb才可以进行对事物的回滚。
- 需要指定 spring.jpa.database-platform=org.hibernate.dialect.MySQL57Dialect

```
1 jpa:
2 # 控制台显示SQL
3 showSql: true
4 hibernate:
5 # 会根据就映射实体类自动创建或更新数据表
6 ddl-auto: update
7 # 默认创建表类型是MyISAM，是非事务安全的，所以无法实现事物回滚
8 # 指定如下方言: 创建的表类型是InnoDB，才可以进行对事物的回滚。
9 database-platform: org.hibernate.dialect.MySQL57Dialect
```

## 2. 创建 Service 层

```
1 public interface IUserService {
2     Boolean addUser(User user);
3 }
4
5 import org.springframework.transaction.annotation.Transactional;
6
7 @Service
8 public class UserServiceImpl implements IUserService {
9     @Autowired
10    UserRepository userRepository;
11
12    /*
13     事务管理：
14     1. 在启动类上，使用 @EnableTransactionManagement 开启注解方式事务支持
15     2. 在 Service层方法上添加 @Transactional 进行事务管理
16    */
17    @Transactional
18    @Override
19    public Boolean addUser(User user) {
20        userRepository.save(new User("1","1"));
21        userRepository.save(new User("12","2"));
22        userRepository.save(new User("123","3"));
23        userRepository.save(new User("1234","4"));
24        userRepository.save(new User("12345","5"));
25        //用户名长度大于5会报错，应该回滚事务的
26        //userRepository.save(new User("123456","6"));
27        //userRepository.save(user);
28        return true;
29    }
30 }
```

- 以上添加 用户名长度大于5会报错时，应该回滚

### 事务管理步骤：

1. 在启动类上，使用 @EnableTransactionManagement 开启注解方式事务支持
2. 在 Service层方法上添加 @Transactional 进行事务管理

```
1 package com.mengxuegu.springboot;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.transaction.annotation.EnableTransactionManagement;
6
7 @EnableTransactionManagement //开启注解的事务管理
8 @SpringBootApplication
9 public class SpringBoot09DataJpaApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SpringBoot09DataJpaApplication.class, args);
13     }
14 }
```

3. 如果使用 JPA 创建表则需要指定数据库引擎为 Innodb

## 10.6.3 事务的隔离级别和传播行为

- 除了指定事务管理器之后，还能对事务进行隔离级别和传播行为的控制，下面分别详细解释：

### 10.6.3.1 隔离级别

隔离级别是指在发生并发的事务之间的隔离程度，与我们开发时候主要相关的场景包括：脏读、不可重复读、幻读。

**脏读**：A事务执行过程中修改了id=1的数据，未提交前，B事务读取了A修改的id=1的数据，而A事务却回滚了，这样B事务就形成了脏读。

**不可重复读**：A事务先读取了一条数据，然后执行逻辑的时候，B事务将这条数据改变了，然后A事务再次读取的时候，发现数据不匹配了，就是所谓的不可重复读了。

**幻读**：A事务先根据条件查询到了N条数据，然后B事务新增了M条符合A事务查询条件的数据，导致A事务再次查询发现有N+M条数据了，就产生了幻读。

- 我们可以看 `org.springframework.transaction.annotation.Isolation` 枚举类中定义了五个表示隔离级别的值：

```
1 public enum Isolation {
2     DEFAULT(-1),
3     READ_UNCOMMITTED(1),
4     READ_COMMITTED(2),
5     REPEATABLE_READ(4),
6     SERIALIZABLE(8);
7 }
```

- `DEFAULT`：这是默认值，表示使用底层数据库的默认隔离级别。对大部分数据库而言，通常这值就是：`READ_COMMITTED`。



- `READ_UNCOMMITTED`：该隔离级别表示一个事务可以读取另一个事务修改但还没有提交的数据。该级别不能防止脏读和不可重复读，因此很少使用该隔离级别。
  - `READ_COMMITTED`：该隔离级别表示一个事务只能读取另一个事务已经提交的数据。该级别可以防止脏读，这也是大多数情况下的推荐值，性能最好。
  - `REPEATABLE_READ`：该隔离级别表示一个事务在整个过程中可以多次重复执行某个查询，并且每次返回的记录都相同。即使在多次查询之间有新增的数据满足该查询，这些新增的记录也会被忽略。该级别可以防止脏读和不可重复读。
  - `SERIALIZABLE`：所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。
- 指定方式：通过使用 `isolation` 属性设置，例如：

```
1 @Transactional(isolation = Isolation.DEFAULT)
```

### 10.6.3.2 传播行为

传播行为是指，如果在开始当前事务之前，已经存在一个事务，此时可以指定这个要开始的这个事务的执行行为。

- 我们可以看 `org.springframework.transaction.annotation.Propagation` 枚举类中定义了6个表示传播行为的枚举值：

```
1 public enum Propagation {  
2     REQUIRED(0),  
3     SUPPORTS(1),  
4     MANDATORY(2),  
5     REQUIRES_NEW(3),  
6     NOT_SUPPORTED(4),  
7     NEVER(5),  
8     NESTED(6);  
9 }
```

- `REQUIRED`：（默认）如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
  - `SUPPORTS`：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
  - `MANDATORY`：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
  - `REQUIRES_NEW`：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
  - `NOT_SUPPORTED`：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
  - `NEVER`：以非事务方式运行，如果当前存在事务，则抛出异常。
  - `NESTED`：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 `REQUIRED`。
- 指定方式：通过使用 `propagation` 属性设置，例如：

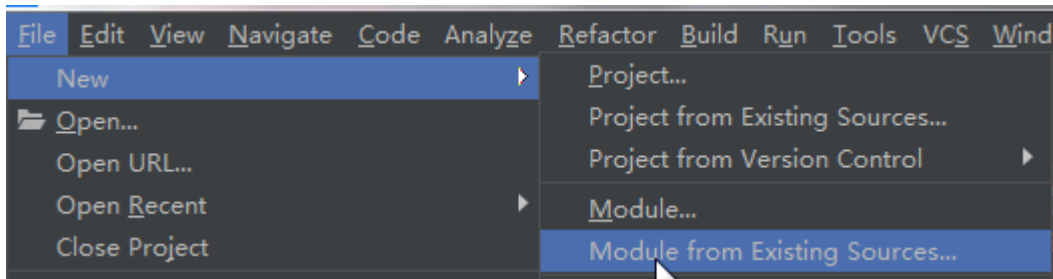
```
1 @Transactional(propagation = Propagation.REQUIRED)
```

## 第11章 项目实战-帐单管理系统完整版

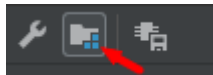
- 数据访问层采用 MyBatis配置文件版

### 11.1 项目环境搭建

1. 构建新项目，复制 spring-boot-05-bill，粘贴为 spring-boot-10-bill，然后导入spring-boot-10-bill



2. 在 Project Structure 中重命名Module名字

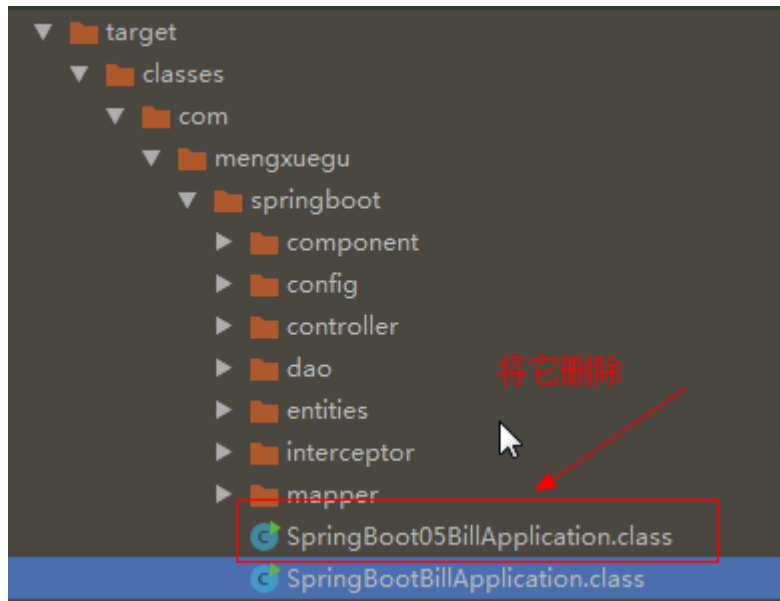


3. 修改pom.xml 的 artifactId 和 name 值

```
6      <groupId>com.mengxuegu</groupId>
7      <artifactId>spring-boot-10-bill</artifactId>
8      <version>0.0.1-SNAPSHOT</version>
9      <packaging>jar</packaging>
10
11      <name>spring-boot-10-bill</name>
12      <description>Demo project for Spring Boot</description>
```

4. Shift+F6 重命名启动类SpringBootBillApplication 与测试类 SpringBootBillApplicationTests
5. 重命名后，看下当前 Module 目录下是否存在原来的 class 文件，有则按delete删除它，不然运行测试类会报如下错误：

```
1 java.lang.IllegalStateException: Found multiple @SpringBootApplication annotated classes [Generic
  bean: class [com.mengxuegu.springboot.SpringBoot05BillApplication];
```



6. 启动 SpringBootBillApplication ，测试是否正常访问
7. 删除 resources\static\error 错误页面，因为保留 templates\error 即可

## 11.2 数据源相关配置

1. 添加依赖，使用 Mybatis 作为 数据数据访问层

```
1 <!--数据源相关-->
2 <dependency>
3   <groupId>org.mybatis.spring.boot</groupId>
4   <artifactId>mybatis-spring-boot-starter</artifactId>
5   <version>1.3.2</version>
6 </dependency>
7 <dependency>
8   <groupId>mysql</groupId>
9   <artifactId>mysql-connector-java</artifactId>
10  <scope>runtime</scope>
11 </dependency>
12 <dependency>
13   <groupId>com.alibaba</groupId>
14   <artifactId>druid</artifactId>
15   <version>1.1.12</version>
16 </dependency>
```

2. 创建 bill 数据库，导入创建表与数据脚本 bill.sql
3. 指定 Druid 数据源，application.yml（要修改库名）与 DruidConfig.java，参考 10.2.2 配置Druid连接池
4. resources类路径下添加 Mybatis 配置,并在配置中指定路径

核心配置文件：mybatis/mybatis-config.xml

映射配置文件：mybatis/mapper/ProviderMapper.xml

```
1 #配置mybatis相关文件路径
```

```
2 mybatis:
3   #映射配置文件路径
4   mapper-locations: classpath:mybatis/mapper/*.xml
5   #核心配置文件路径
6   config-location: classpath:mybatis/mybatis-config.xml
7
8   # 控制台打印sql
9   logging:
10    level:
11      com.mengxuegu.springboot.mapper: debug
12
13   #数据源相关配置
14   spring:
15     datasource:
16       username: root
17       password: root
18       #mysql8版本以上的驱动包，需要指定以下时区
19       url: jdbc:mysql://127.0.0.1:3306/bill?serverTimezone=GMT%2B8
20       #mysql8版本以上指定新的驱动类
21       driver-class-name: com.mysql.cj.jdbc.Driver
22       #引入Druid数据源
23       type: com.alibaba.druid.pool.DruidDataSource
24
25       # 数据源其他配置, DataSourceProperties中没有相关属性,默认无法绑定
26       initialSize: 8
27       minIdle: 5
28       maxActive: 20
29       maxWait: 60000
30       timeBetweenEvictionRunsMillis: 60000
31       minEvictableIdleTimeMillis: 300000
32       validationQuery: SELECT 1 FROM DUAL
33       testWhileIdle: true
34       testOnBorrow: false
35       testOnReturn: false
36       poolPreparedStatements: true
37       # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
38       filters: stat,wall,logback
39       maxPoolPreparedStatementPerConnectionSize: 25
40       useGlobalDataSourceStat: true
41       connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
```

5. 访问Druid监控后台

## 11.3 完成供应商管理模块

1. 创建数据访问层 mapper.ProviderMapper

```
1 package com.mengxuegu.springboot.mapper;
2
3 import com.mengxuegu.springboot.entities.Provider;
```

```
4 import java.util.List;
5
6 /**
7  * @Author: 梦学谷
8  */
9 // @Mapper 或 @MapperScan("com.mengxuegu.springboot.mapper")
10 public interface ProviderMapper {
11
12     List<Provider> getProviders(Provider provider);
13
14     Provider getProviderByPid(Integer pid);
15
16     int addProvider(Provider provider);
17
18     int deleteProviderByPid(Integer pid);
19
20     int updateProvider(Provider provider);
21
22 }
23
```

2. 扫描 Mapper，在启动类上添加 `@MapperScan("com.mengxuegu.springboot.mapper")`

3. 在 mybatis/mapper/ProviderMapper.xml 添加 SQL 语句

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org/DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.mengxuegu.springboot.mapper.ProviderMapper">
6
7     <select id="getProviders" resultType="com.mengxuegu.springboot.entities.Provider">
8         select * from provider where 1=1
9         <if test="providerName != null and providerName != ''">
10             <!-- ${} 用于字符串拼接 -->
11             and providerName like '%${providerName}%'
12         </if>
13     </select>
14
15     <select id="getProviderByPid" resultType="com.mengxuegu.springboot.entities.Provider">
16         select * from provider where pid=#{pid}
17     </select>
18
19     <insert id="addProvider">
20         INSERT INTO `provider` ( `provider_code`, `providerName`, `people`, `phone`, `address`, `fax`,
21             `describe`, `create_date`)
22         VALUES ( #{providerCode}, #{providerName}, #{people}, #{phone}, #{address}, #{fax},#{describe},
23             now())
24     </insert>
25
26     <update id="updateProvider">
27         UPDATE `bill`.`provider`
28
29         SET `provider_code`=#{providerCode}, `providerName`=#{providerName}, `people`=#{people},
```

```
27     `phone`=#{phone}, `address`=#{address}, `fax`=#{fax},  
28     `describe`=#{describe}, `create_date`=now()  
29     WHERE `pid`=#{pid}  
30 </update>  
31  
32 <delete id="deleteProviderByPid">  
33     delete from provider where pid = #{pid}  
34 </delete>  
35  
36 </mapper>
```

#### 4. 测试 mapper

```
1 @RunWith(SpringRunner.class)  
2 @SpringBootTest  
3 public class SpringBootTestApplicationTests {  
4  
5     @Autowired  
6     ProviderMapper providerMapper;  
7  
8     @Test  
9     public void contextLoads() {  
10         List<Provider> providers = providerMapper.getProviders(null);  
11         System.out.println(providers.get(0));  
12  
13         Provider provider = providerMapper.getProviderByPid(1);  
14         System.out.println(provider);  
15  
16         provider.setProviderCode("P_11111");  
17         int size = providerMapper.updateProvider(provider);  
18         System.out.println(size);  
19  
20         providerMapper.addProvider(new Provider(null, "PR-AA", "梦学谷供应商111", "小张", "18888666981",  
21             "深圳软件园", "0911-0123456", "品质A"));  
22  
23         providerMapper.deleteProviderByPid(5);  
24     }  
25 }  
26 }
```

#### 5. 修改 ProviderController

```
1 package com.mengxuegu.springboot.controller;  
2  
3 import com.mengxuegu.springboot.dao.ProviderDao;  
4 import com.mengxuegu.springboot.entities.Provider;  
5 import com.mengxuegu.springboot.mapper.ProviderMapper;  
6 import org.slf4j.Logger;  
7 import org.slf4j.LoggerFactory;  
8 import org.springframework.beans.factory.annotation.Autowired;  
9 import org.springframework.stereotype.Controller;  
10 import org.springframework.web.bind.annotation.*;  
11
```

```
12 import java.util.Collection;
13 import java.util.List;
14 import java.util.Map;
15
16 /**
17  * 供应商的控制层
18  * @Author: 梦学谷
19  */
20 @Controller
21 public class ProviderController {
22     Logger logger = LoggerFactory.getLogger(getClass());
23
24     @Autowired
25     ProviderDao providerDao;
26
27     @Autowired
28     ProviderMapper providerMapper;
29
30
31     @GetMapping("/providers")
32     public String list(Map<String, Object> map, Provider provider) {
33         logger.info("供应商列表查询。。。" + provider);
34
35         List<Provider> providers = providerMapper.getProviders(provider);
36
37         map.put("providers", providers);
38         map.put("providerName", provider.getProviderName());
39
40         return "provider/list";
41     }
42
43     /**
44      * type = null 进入查看详情页面view.html ,
45      * type=update 则是进入update.html
46      * @param pid 供应商id
47      * @param type
48      * @param map
49      * @return
50      */
51     @GetMapping("/provider/{pid}")
52     public String view(@PathVariable("pid") Integer pid,
53                       @RequestParam(value="type", defaultValue="view") String type,
54                       Map<String, Object> map) {
55         logger.info("查询" + pid + "的供应商详细信息");
56
57         Provider provider = providerMapper.getProviderByPid(pid);
58
59         map.put("provider", provider);
60
61         // type = null 则进入view.html , type=update 则是进入update.html
62         return "provider/" + type;
63     }
64
```

```
65 //修改供应商信息
66 @PutMapping("/provider")
67 public String update(Provider provider) {
68     logger.info("更改供应商信息。。。");
69     //更新操作
70     providerMapper.updateProvider(provider);
71
72     return "redirect:providers";
73 }
74
75 //前往添加 页面
76 @GetMapping("/provider")
77 public String toAddPage() {
78     return "provider/add";
79 }
80
81
82 //添加数据
83 @PostMapping("/provider")
84 public String add(Provider provider) {
85     logger.info("添加供应商数据" + provider);
86     //保存数据操作
87     providerMapper.addProvider(provider);
88
89     return "redirect:/providers";
90 }
91
92 //删除供应商
93 @DeleteMapping("/provider/{pid}")
94 public String delete(@PathVariable("pid") Integer pid) {
95     logger.info("删除操作, pid=" + pid);
96     providerMapper.deleteProviderByPid(pid);
97     return "redirect:/providers";
98 }
99 }
100
```

## 11.4 帐单管理模块

1. 改造 实体类，因为列表需要 供应商名称

Bill 类中添加 `private Integer pid;`



```
1 private Integer pid;
2 public Integer getPid() {
3     return pid;
4 }
5
6 public void setPid(Integer pid) {
7     this.pid = pid;
8 }
```

BillProvider 继承 Bill 后，BillProvider包含了Bill的所有属性，只需要新增 供应商 的信息属性即可

```
1 public class BillProvider extends Bill{
2     private String providerName;
3
4     public String getProviderName() {
5         return providerName;
6     }
7     public void setProviderName(String providerName) {
8         this.providerName = providerName;
9     }
10 }
```

## 2. BillMapper

```
1 package com.mengxuegu.springboot.mapper;
2
3 import com.mengxuegu.springboot.entities.Bill;
4 import com.mengxuegu.springboot.entities.BillProvider;
5 import com.mengxuegu.springboot.entities.Provider;
6
7 import java.util.List;
8
9 /**
10  * @Author: 梦学谷
11  */
12 // @Mapper 或 @MapperScan("com.mengxuegu.springboot.mapper")
13 public interface BillMapper {
14
15     List<BillProvider> getBills(Bill bill);
16
17     BillProvider getBillByBid(Integer bid);
18
19     int addBill(Bill bill);
20
21     int updateBill(Bill bill);
22
23     int deleteBillByBid(Integer bid);
24
25 }
26
```

## 3. BillMapper.xml

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE mapper
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
5 <mapper namespace="com.mengxuegu.springboot.mapper.BillMapper">
6
7     <select id="getBills" resultType="com.mengxuegu.springboot.entities.BillProvider">
8         select b.*, p.providerName from bill b left join provider p on b.pid = p.pid
9         where 1=1
10        <if test="billName != null and billName != "">
11            and b.bill_name like '%${billName}%'
12        </if>
13        <if test="pid != null ">
14            and b.pid = #{pid}
15        </if>
16
17        <if test="pay != null ">
18            and b.pay = #{pay}
19        </if>
20    </select>
21
22    <select id="getBillByBid" resultType="com.mengxuegu.springboot.entities.BillProvider">
23        select b.*, p.providerName from bill b left join provider p on b.pid = p.pid
24        where b.bid = #{bid}
25    </select>
26
27    <insert id="addBill" >
28        INSERT INTO `bill` ( `bill_code`, `bill_name`, `bill_com`, `bill_num`, `money`, `pay`, `pid`,
29        `create_date`)
30        VALUES ( #{billCode}, #{billName}, #{billCom}, #{billNum}, #{money}, #{pay}, #{pid}, now());
31    </insert>
32
33    <update id="updateBill">
34        UPDATE `bill`
35        SET `bill_code`=#{billCode}, `bill_name`=#{billName}, `bill_com`=#{billCom}, `bill_num`=#{
36        billNum}, `money`=#{money}, `pay`=#{pay}, `pid`=#{pid}, `create_date`='2018-11-17 15:22:03'
37        WHERE `bid`=#{bid}
38    </update>
39
40    <delete id="deteleBillByBid">
41        delete from bill where bid=#{bid}
42    </delete>
43
44 </mapper>
```

## 4. 测试

```
1
2 @Autowired
3 BillMapper billMapper;
```

```
4  @Test
5  public void testBill() {
6      Bill b = new Bill();
7      b.setBillName("com");
8      List<BillProvider> bills = billMapper.getBills(b);
9      System.out.println(bills.get(0));
10
11     BillProvider billProvider = billMapper.getBillByBid(4);
12     System.out.println(billProvider);
13
14     Bill bill = (Bill) billProvider;
15     bill.setBillName("cn域名...");
16     billMapper.updateBill(bill);
17
18     //billMapper.addBill(new Bill(3001, "Bi-AA11", "粮油aaa", "斤", 80,480.8, new Provider(null, "PR-BB",
19     "梦学谷供应商222", "小李", "18888666982", "深圳软件园", "0911-0123453", "品质B"), 1));
20     billMapper.deleteBillByBid(7);
21 }
```

5. 修改 public.html 中账单管理请求路径

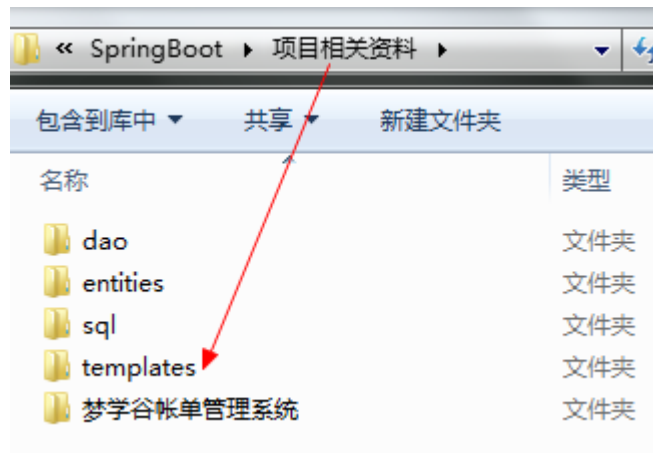
```
1  <a th:href="@{/bills}" href="../bill/list.html">账单管理</a>
```

6. BillController

```
1  @Controller
2  public class BillController {
3      Logger logger = LoggerFactory.getLogger(getClass());
4      @Autowired
5      BillMapper billMapper;
6      @Autowired
7      ProviderMapper providerMapper;
8
9      @GetMapping("/bills")
10     public String list(Map<String, Object> map, BillProvider bp) {
11         logger.info("账单列表查询。。。" + bp);
12         //获取所有供应商，
13         List<Provider> providers = providerMapper.getProviders(null);
14         //查询账单
15         Collection<BillProvider> billProviders = billMapper.getBills(bp);
16
17         map.put("providers", providers);
18         map.put("billProviders", billProviders);
19         //回显
20         map.put("billName", bp.getBillName());
21         map.put("pid", bp.getPid());
22         map.put("pay", bp.getPay());
23         return "bill/list";
24     }
25     /**
26      * type = null 进入查看详情页面view.html，
27      * type=update 则是进入update.html
```

```
28     */
29     @GetMapping("/bill/{bid}")
30     public String view(@PathVariable("bid") Integer bid,
31         @RequestParam(value="type", defaultValue = "view") String type,
32         Map<String, Object> map) {
33         logger.info("查询" + bid + "的帐单详细信息");
34
35         BillProvider billProvider = billMapper.getBillByBid(bid);
36         map.put("billProvider", billProvider);
37
38         //查询所有供应商
39         if("update".equals(type)) {
40             map.put("providers", providerMapper.getProviders(null));
41         }
42
43         // type = null 则进入view.html , type=update 则是进入update.html
44         return "bill/" + type;
45     }
46
47     //修改
48     @PutMapping("/bill")
49     public String update(Bill bill) {
50         logger.info("更改帐单信息。。。");
51         //更新操作
52         billMapper.updateBill(bill);
53         return "redirect:bills";
54     }
55
56     //前往添加 页面
57     @GetMapping("/bill")
58     public String toAddPage(Map<String, Object> map) {
59         //查询所有供应商
60         map.put("providers", providerMapper.getProviders(null));
61         return "bill/add";
62     }
63
64     //添加数据
65     @PostMapping("/bill")
66     public String add(Bill bill) {
67         logger.info("添加帐单数据" + bill);
68         //保存数据操作
69         billMapper.addBill(bill);
70         return "redirect:/bills";
71     }
72
73     //删除
74     @DeleteMapping("/bill/{bid}")
75     public String delete(@PathVariable("bid") Integer bid) {
76         logger.info("删除操作, bid=" + bid);
77         billMapper.deleteBillByBid(bid);
78         return "redirect:/bills";
79     }
80 }
```

7. 模块页面直接复制 项目相关资料 下的 templates 对应 bill



## 11.5 用户管理模块

1. UserMapper.java

```
1 public interface UserMapper {  
2  
3     List<User> getUsers(User user);  
4  
5     User getUserById(Integer id);  
6  
7     int addUser(User user);  
8  
9     int updateUser(User user);  
10  
11    int deleteUserById(Integer id);  
12  
13 }
```

2. UserMapper.xml

```
1 <?xml version="1.0" encoding="UTF-8" ?>  
2 <!DOCTYPE mapper  
3     PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
4     "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
5 <mapper namespace="com.mengxuegu.springboot.mapper.UserMapper">  
6  
7     <select id="getUsers" resultType="com.mengxuegu.springboot.entities.User">  
8         select * from `user` where 1=1  
9         <if test="username != null and username != ''">  
10             <!-- ${}用于字符串拼接使用-->  
11             and username like '%${username}%'  
12         </if>  
13     </select>
```

```
13     </select>
14
15     <select id="getUserById" resultType="com.mengxuegu.springboot.entities.User">
16         select * from `user` where id = #{id}
17     </select>
18
19     <insert id="addUser" >
20         INSERT INTO `user` ( `username`, `real_name`, `password`, `gender`, `birthday`, `user_type`)
21         VALUES ( #{username}, #{realName}, #{password}, #{gender}, #{birthday}, #{userType})
22     </insert>
23
24     <update id="updateUser">
25         UPDATE `user`
26         SET `username`=#{username}, `real_name`=#{realName}, `password`=#{password},
27             `gender`=#{gender}, `birthday`=#{birthday}, `user_type`=#{userType}
28         WHERE `id` = #{id}
29     </update>
30
31     <delete id="deleteUserById">
32         delete from `user`
33         where id = #{id}
34     </delete>
35
36 </mapper>
```

### 3. 测试

```
1  @Autowired
2  UserMapper userMapper;
3  @Test
4  public void testUser() {
5      User u = new User();
6      // u.setUsername("zhang");
7
8      List<User> users = userMapper.getUsers(u);
9      System.out.println(users.get(0));
10
11     User user = userMapper.getUserById(1);
12     System.out.println(user);
13
14     user.setUsername("admin");
15     int size = userMapper.updateUser(user);
16     System.out.println(size);
17
18     billMapper.deleteBillByBid(4);
19 }
```

### 4. 控制层

```
1  @Controller
2  public class UserController {
3      Logger logger = LoggerFactory.getLogger(getClass());
```

```
4
5  @Autowired
6  UserMapper userMapper;
7
8  @GetMapping("/users")
9  public String list(Map<String, Object> map, User user) {
10     logger.info("用户列表查询。。。" + user);
11
12     List<User> users = userMapper.getUsers(user);
13
14     map.put("users", users);
15     map.put("username", user.getUsername());
16
17     return "user/list";
18 }
19
20 /**
21  * type = null 进入查看详情页面view.html ,
22  * type=update 则是进入update.html
23  */
24 @GetMapping("/user/{id}")
25 public String view(@PathVariable("id") Integer id,
26     @RequestParam(value="type", defaultValue = "view") String type,
27     Map<String, Object> map) {
28     logger.info("查询" + id + "的用户详细信息");
29
30     User user = userMapper.getUserById(id);
31
32     map.put("user", user);
33
34     // type = null 则进入view.html , type=update 则是进入update.html
35     return "user/" + type;
36 }
37 //修改
38 @PutMapping("/user")
39 public String update(User user) {
40     logger.info("更改用户信息。。。");
41     //更新操作
42     userMapper.updateUser(user);
43
44     return "redirect:users";
45 }
46 //前往添加 页面
47 @GetMapping("/user")
48 public String toAddPage() {
49     return "user/add";
50 }
51 //添加数据
52 @PostMapping("/user")
53 public String add(User user) {
54     logger.info("添加用户数据" + user);
55     //保存数据操作
56
57     userMapper.addUser(user);
```

```
57     return "redirect:/users";
58 }
59 //删除
60 @DeleteMapping("/user/{id}")
61 public String delete(@PathVariable("id") Integer id) {
62     logger.info("删除操作, pid=" + id);
63     userMapper.deleteUserById(id);
64     return "redirect:/users";
65 }
66 }
```

#### 5. public.html 修改路径

```
1 <a th:href="@{/users}" href="../user/list.html">用户管理</a>
```

#### 6. 模板页面 直接复制 项目相关资源 下的 templates 对应 user

注意：新增 修改页面有生日是Date类型，springboot默认识别 dd/MM/yyyy 格式

但是我们传入的是其他格式，如 yyyy-MM-dd，则需要在配置中修改日期格式

```
1 #指定日期格式
2 spring.mvc.date-format=yyyy-MM-dd
```

## 11.6 重构登录功能

#### 1. UserMapper.java 增加一个方法

```
1 User getUserByUsername(String username);
```

#### 2. UserMapper.xml

```
1 <select id="getUserByUsername" resultType="com.mengxuegu.springboot.entities.User">
2     select * from `user` where upper(username) = upper("#{username})
3 </select>
```

#### 3. LoginController.login(...)

```
1 @Controller
2 public class LoginController {
3     @Autowired
4     UserMapper userMapper;
5
6     @PostMapping("/login")
7     public String login (HttpSession session, String username, String password, Map<String, Object> map) {
8
9         if( StringUtils.isEmpty(username)
```



```
10         && !StringUtils.isEmpty(password)) {  
11  
12         //查询数据库用户是否存在  
13         User user = userMapper.getUserByUsername(username);  
14         if(user != null && password.equals(user.getPassword())) {  
15             //登录成功  
16             session.setAttribute("loginUser", user.getUsername());  
17             //重定向 redirect：可以重定向到任意一个请求中（包括其他项目），地址栏改变  
18             return "redirect:/main.html";  
19         }  
20     }  
21     map.put("msg", "用户名或密码错误");  
22     return "";  
23 }
```

## 11.7 密码修改模块

需求：先使用Ajax异步校验输入的原密码是否正确，正确则JS校验新密码输入是否一致，一致则提交修改，然后注销重新回到登录页面。

1. Session存入User对象并重构 主页 用户名 显示
2. main/password.html 抽取公共代码片段

```
1 <div class="left" th:replace="main/public :: #public_left(activeUri='pwd')">
```

3. JS，注意js中引入thymeleaf行内表达式

```
1 <script type="text/javascript" th:inline="javascript">  
2 // 要使用thymeleaf行内表达式则上面需要使用：th:inline="javascript" 标识  
3 $(function () {  
4     var isCheck = false;  
5     //原密码失去焦点  
6     $("#oldPassword").blur(function () {  
7         var oldPassword = $(this).val().trim();  
8         if(oldPassword) {  
9             $('#pwdText').css('color', 'red');  
10            isCheck = false;  
11            return ;  
12        }  
13        //thymeleaf行内表达式  
14        var url = [{"@{/user/pwd}"}] + oldPassword;  
15        //异步判断密码是否正确  
16        $.ajax({  
17            url: url,  
18            dataType: 'json',  
19            method: 'GET',  
20            success: function (data) {  
21                isCheck = data;  
22                data ? $("#pwdText").replaceWith("<span id='pwdText'>*原密码正确</span>")
```

```
23         : $("#pwdText").replaceWith("<span id='pwdText' style='color: red'>*原密码错误</span>");
24         return;
25     },
26     error: function () {
27         $("#pwdText").html("校验原密码异常");
28         isChecked = false;
29         return;
30     }
31 });
32 });
33
34 $("#save").click(function () {
35     if(isCheck) {
36         if($("#newPassword").val() && $("#reNewPassword").val()
37             && $("#newPassword").val() == $("#reNewPassword").val()) {
38             $("#pwdForm").submit();
39         }else{
40             $("#reNewPwdText").replaceWith("<span id='reNewPwdText' style='color: red'>*保证和新密码
一致</span>");
41         }
42     }
43 });
44 });
45
46 </script>
```

#### 4. 控制层

```
1  @Controller
2  public class UserController {
3      //前往密码修改页面
4      @GetMapping("user/pwd")
5      public String toPwdUpdatePage() {
6          return "/main/password";
7      }
8
9      //校验密码是否正确
10     @ResponseBody
11     @GetMapping("user/pwd/{oldPwd}")
12     public Boolean checkPwd(@PathVariable("oldPwd") String oldPwd,
13         HttpSession session) {
14         logger.info("输入的旧密码为 : " + oldPwd);
15         User user = (User) session.getAttribute("loginUser");
16         if(user.getPassword().equals(oldPwd)) {
17             return true;
18         }
19         return false;
20     }
21
22     @PostMapping("/user/pwd")
23     public String updatePwd(HttpSession session, String password) {
24         //获取session中的登录信息
```

```
25     User user = (User) session.getAttribute("loginUser");
26     //更新密码
27     user.setPassword(password);
28     userMapper.updateUser(user);
29
30     //注销重新登录
31     return "redirect:/logout";
32 }
33 }
```

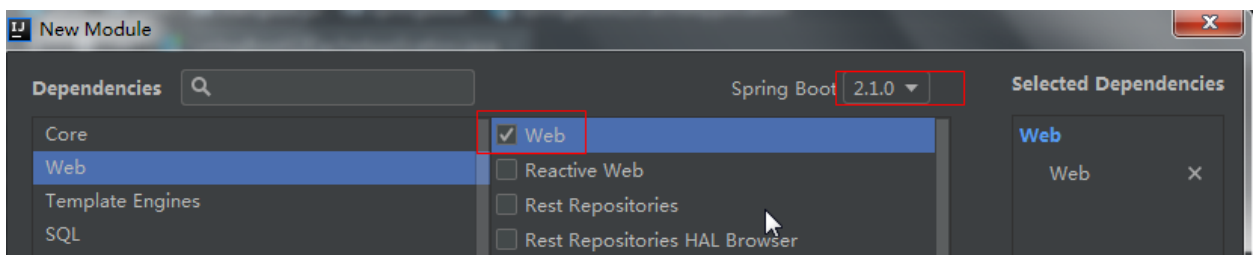
## 第12章 Spring Boot 异步任务与定时任务实战

### 12.1 Spring Boot 异步任务实战

- 在项目开发中，绝大多数情况下都是通过同步方式处理业务逻辑的，但是比如批量处理数据，批量发送邮件，批量发送短信等操作 容易造成阻塞的情况，之前大部分都是使用多线程来完成此类任务。

而在Spring 3+之后，就已经内置了 `@Async` 注解来完美解决这个问题，从而提高效率。

- 使用的注解：
  - `@EnableAysnc` 启动类上开启基于注解的异步任务
  - `@Aysnc` 标识的方法会异步执行
- 异步任务实战操作如下：



```
1 package com.mengxuegu.springboot.service;
2 import org.springframework.scheduling.annotation.Async;
3 import org.springframework.stereotype.Service;
4
5 /**
6  * 异步任务批量处理
7  * @Author: 梦学谷
8  */
9 @Service
10 public class AsyncService {
11     //批量新增操作
12     @Async
13     public void batchAdd() {
14         try {
```

```
15     Thread.sleep(3*1000);
16 } catch (InterruptedException e) {
17     e.printStackTrace();
18 }
19 System.out.println("批量保存数据中....");
20 }
21 }
```

```
1 @RestController
2 public class AsyncController {
3
4     @Autowired
5     AsyncService asyncService;
6
7     @GetMapping("/hello")
8     public String hello() {
9         asyncService.batchAdd();
10        return "success";
11    }
12
13 }
```

```
1 @EnableAsync //开启基于注解的异步处理
2 @SpringBootApplication
3 public class SpringBoot11TaskApplication {
4     public static void main(String[] args) {
5         SpringApplication.run(SpringBoot12TaskApplication.class, args);
6     }
7 }
```

## 12.2 Spring Boot 定时任务调度实战

- 在项目开发中，经常需要执行一些定时任务，比如 每月1号凌晨需要汇总上个月的数据分析报表; 每天凌晨分析前一天的日志信息等定时操作。Spring 为我们提供了异步执行定时任务调度的方式。
- 使用的注解：
  - `@EnableScheduling` 启动类上开启基于注解的定时任务
  - `@Scheduled` 标识的方法会进行定时处理
    - 需要通过 cron 属性来指定 cron 表达式：秒 分 时 日 月 星期几

```

1  @Service
2  public class ScheduledService {
3      private static int count = 1;
4      /**
5       *   秒 分 时 日 月 星期几
6       * 比如: "0 * * * * MON-FRI" 周一到周五, 每次0秒执行(即每分钟执行一次)
7       */
8      @Scheduled(cron = "*/* * * * MON-FRI")
9      public void dataCount() {
10         System.out.println("数据统计第" + count++ + "次");
11     }
12
13 }

```

- cron表达式

位置	取值范围	可指定的特殊字符
秒	0-59	, - * /
分	0-59	, - * /
小时	0-23	, - * /
日期	1-31	, - * ? / L W C
月份	1-12	, - * /
星期	0-7或SUN-SAT 0和7都是周日, 1-6是周一到周六	, - * ? / L C #

特殊字符	代表含义
,	枚举, 一个位置上指定多个值, 以逗号 , 分隔
-	区间
*	任意
/	步长, 每隔多久执行一次
?	日/星期冲突匹配, 指定哪个值, 另外个就是?, 比如: <code>***? * 1</code> 每周1执行, 则日用 ? 不能用 *, 不是每一天都是周一; <code>*** * 2 *</code> 每月2号, 则星期不能用*
L	最后
W	工作日
C	和calendar联系后计算过的值
#	这个月的第几个星期几, 4#2, 第2个星期四

- 在线生成cron表达式 <http://cron.qqe2.com/>

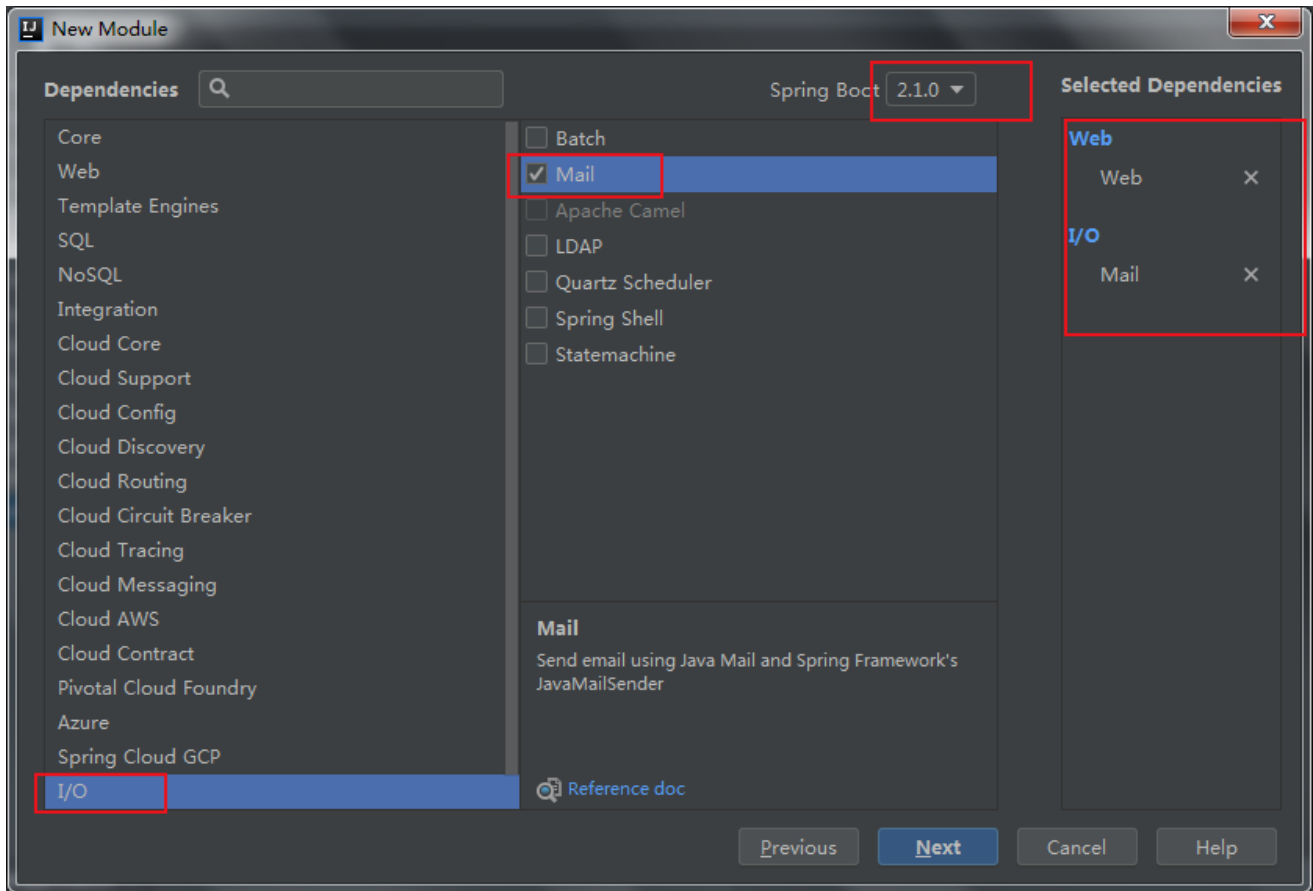
```
1 1-5 * * * * 1到5秒, 每秒都触发任务
2 */5 * * * * 每隔5秒执行一次
3 0 */1 * * * 每隔1分钟执行一次
4 0 0 5-15 * * 每天5-15点整点触发
5 0 0-5 14 * * 在每天下午2点到下午2:05期间的每1分钟触发
6 0 0/5 14 * * 在每天下午2点到下午2:55期间的每5分钟触发
7 0 0/5 14,18 * * 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发
8 0 0/30 9-17 * * 朝九晚五工作时间内每半小时
9 0 0 12 ? * WED 表示每个星期三中午12点
10 0 10,44 14 ? 3 WED 每年三月的星期三的下午2:10和2:44触发
11 0 0 23 L * ? 每月最后一天23点执行一次
12 0 15 10 LW * ? 每个月最后一个工作日的10点15分0秒触发任务
13 0 15 10 ? * 5#3 每个月第三周的星期五的10点15分0秒触发任务
```

## 第13章 Spring Boot 邮件发送实战

### 13.1 邮件发送环境准备



实战操作步骤：



1. 引入邮件启动器： `spring-boot-starter-mail`

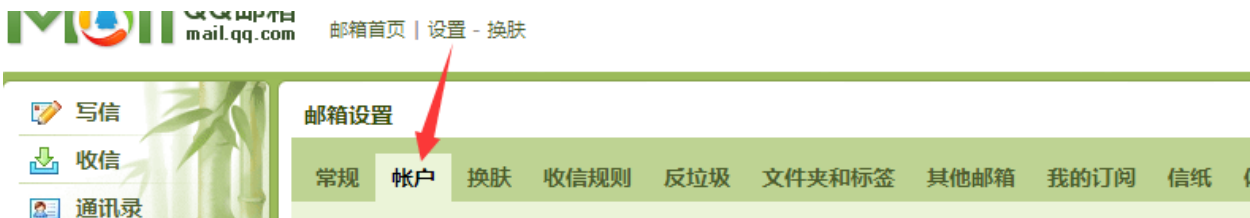
```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-mail</artifactId>
4 </dependency>
```

2. Spring Boot 提供了自动配置类 `MailSenderAutoConfiguration`
3. 在 `application.properties` 中配置邮箱信息, 参考 `MailProperties`

4.
 

```
1 spring.mail.username=736486962@qq.com
2 spring.mail.password="指定qq生成的授权码"
3 spring.mail.host=smtp.qq.com
4 #需要开启ssl安全连接
5 spring.mail.properties.smtp.ssl.enable=true
```

密码不写明文在配置中，在QQ邮箱中进行获取制授权码，如下操作



#### POP3/IMAP/SMTP/Exchange/CardDAV/CalDAV服务

全部开启，如果不能开启，则根据提示操作

开启服务：	POP3/SMTP服务 (如何使用 Foxmail 等软件收发邮件?)	已关闭	开启
	IMAP/SMTP服务 (什么是 IMAP，它又是如何设置?)	已关闭	开启
	Exchange服务 (什么是Exchange，它又是如何设置?)	已关闭	开启
	CardDAV/CalDAV服务 (什么是CardDAV/CalDAV，它又是如何设置?)	已关闭	开启
	(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置?)		

#### POP3/IMAP/SMTP/Exchange/CardDAV/CalDAV服务

开启服务：	POP3/SMTP服务 (如何使用 Foxmail 等软件收发邮件?)	已开启	关闭
	IMAP/SMTP服务 (什么是 IMAP，它又是如何设置?)	已开启	关闭
	Exchange服务 (什么是Exchange，它又是如何设置?)	已开启	关闭
	CardDAV/CalDAV服务 (什么是CardDAV/CalDAV，它又是如何设置?)	已开启	关闭
	(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置?)		

温馨提示：登录第三方客户端时，密码框请输入“授权码”进行验证⑦。生成授权码

- 上面开通后，点击“生成授权码”，根据提示发送短信验证，会生成邮件密码，修改密码会重新生成

加锁“文件夹区域”...

“文件夹区域”是由“我的文件夹”、“其他邮箱”...

/Exchange/CardDAV/CalDAV服

POP3/SMTP服务 (如何使用 Foxmail 等软件)

IMAP/SMTP服务 (什么是 IMAP，它又是如何)

Exchange服务 (什么是Exchange，它又是如何)

CardDAV/CalDAV服务 (什么是CardDAV/C

(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置?)

温馨提示：登录第三方客户端时，密码框请输入“授权码”进行验证⑦。生成授权码

短信验证 ?

请先用密保手机 [REDACTED] 发短信，然后点“我已发送”按钮

发短信：配置邮件客户端

到号码：1069 0700 69

短信费用

短信用不了？

验不了，试试其他 ▲

我已发送

## 13.2 邮件发送实战操作

### 1. Spring Boot 自动装配 JavaMailSenderImpl 进行发送邮件

```
1 package com.mengxuegu.springboot;  
2  
3 import org.junit.Test;  
4 import org.junit.runner.RunWith;  
5 import org.springframework.beans.factory.annotation.Autowired;  
6 import org.springframework.boot.test.context.SpringBootTest;  
7 import org.springframework.mail.SimpleMailMessage;  
8 import org.springframework.mail.javamail.JavaMailSenderImpl;  
9 import org.springframework.mail.javamail.MimeMailMessage;
```



```
10 import org.springframework.mail.javamail.MimeMessageHelper;
11 import org.springframework.test.context.junit4.SpringRunner;
12
13 import javax.mail.MessagingException;
14 import javax.mail.internet.MimeMessage;
15 import java.io.File;
16
17 @RunWith(SpringRunner.class)
18 @SpringBootTest
19 public class SpringBoot12TaskApplicationTests {
20
21     @Autowired
22     JavaMailSenderImpl javaMailSender;
23
24     @Test
25     public void testSimpleMail() {
26         //封装简单的邮件内容
27         SimpleMailMessage message = new SimpleMailMessage();
28         //邮件主题
29         message.setSubject("放假通知");
30         message.setText("春节放假7天");
31
32         //发件人
33         message.setFrom("736486962@qq.com");
34         message.setTo("mengxuegu666@163.com");
35
36         javaMailSender.send(message);
37     }
38
39     //发送复杂邮件带附件和html的邮件
40     @Test
41     public void testMimeMail() throws MessagingException {
42         //创建一个发送复杂消息对象
43         MimeMessage mimeMessage = javaMailSender.createMimeMessage();
44         //通过消息帮助对象，来设置发送的内容
45         MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage, true);
46         //邮件主题
47         messageHelper.setSubject("放假通知");
48         //第2个参数为true表示是html
49         messageHelper.setText("<h2 style='color:red'>春节放假7天</h2>", true);
50
51         //上传文件 (文件名，File或IO对象)
52         messageHelper.addAttachment("1.jpg", new File("D:\\images\\1.jpg"));
53         messageHelper.addAttachment("2.jpg", new File("D:\\images\\2.jpg"));
54         messageHelper.addAttachment("3.jpg", new File("D:\\images\\3.jpg"));
55
56         //发件人
57         messageHelper.setFrom("736486962@qq.com");
58         messageHelper.setTo("mengxuegu666@163.com");
59
60         javaMailSender.send(mimeMessage);
61     }
62 }
```

## 第14章 Spring Boot 整合缓存实战

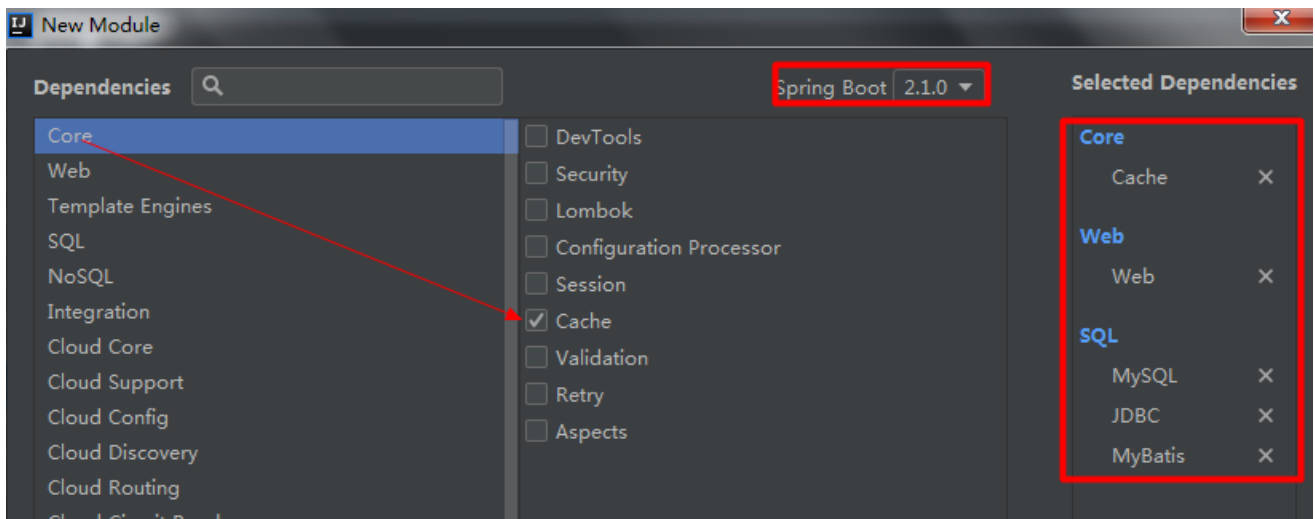
### 14.1 缓存 简介

- 缓存是每一个系统都应该考虑的功能，它用于加速系统的访问，以及提速系统的性能。比如：
  - 经常访问的高频热点数据：
    - 电商网站的商品信息：每次查询数据库耗时，可以引入缓存。
    - 微博阅读量、点赞数、热点话题等
  - 临时性的数据：发送手机验证码，1分钟有效，过期则删除，存数据库负担有点大，这些临时性的数据也可以放到缓存中，直接从缓存中存取数据。

### 14.2 SpringBoot 整合缓存

- Spring从3.1后定义了 `org.springframework.cache.CacheManager` 和 `org.springframework.cache.Cache` 接口来统一不同的缓存技术；
  - `CacheManager` 缓存管理器，用于管理各种Cache缓存组件
  - `Cache` 定义了缓存的各种操作，Spring在Cache接口下提供了各种xxxCache的实现；  
比如EhCacheCache，RedisCache，ConcurrentMapCache.....
- Spring 提供了缓存注解：`@EnableCaching`、`@Cacheable`、`@CachePut`

整合缓存步骤：



1. 引入 缓存 启动器：spring-boot-starter-cache
2. 创建 cache 数据库，导入 bill.sql 与 实体对象，创建注解版 mapper、service 与 Controller
3. @EnableCaching：在启动类上，开启基于注解的缓存
4. @Cacheable：标在方法上，返回的结果会进行缓存(先查缓存中的结果，没有则调用方法并将结果放到缓存中)
  - 属性：
    - value/cacheNames：缓存的名字
    - key：作为缓存中的Key值，可自己使用 SpEL表达式指定(不指定就是参数值)，缓存结果是方法返回值

名字	描述	示例
methodName	当前被调用的方法名	#root.methodName
target	当前被调用的目标对象	#root.target
targetClass	当前被调用的目标对象类	#root.targetClass
args	当前被调用的方法的参数列表	#root.args[0]
caches	当前方法调用使用的缓存列表（如@Cacheable(value={ "cache1", "cache2"})），则有两个cache	#root.caches[0].name
argument name	方法参数的名字. 可以直接 #参数名，也可以使用 #p0或#a0 的形式，0代表参数的索引；	#iban、#a0、#p0
result	方法执行后的返回值（仅当方法执行之后的判断有效,在 @CachePut 使用于更新数据后可用）	#result

5. @CachePut：保证方法被调用后，又将对缓存中的数据更新（先调用方法，调完方法再将结果放到缓存）  
 比如：修改了表中某条数据后，同时更新缓存中的数据，使得别人查询这条更新的数据时直接从缓存中获取
  - 测试更新User数据效果：
    1. 先查询id=1的用户，放在缓存中；
    2. 后面查询id=1的用户直接从缓存中查询；
    3. 更新id=1的用户，同时会更新缓存数据；
    4. 再查询id=1的用户应该是更新后的数据，是从缓存中查询，因为在更新时同时再新了缓存数据

**注意：需要指定key属性：** key="#user.id" 参数对象的id 或 key = "#result.id" 返回值对象id

6. @CacheEvict：清除缓存
  - 属性
    - key：指要清除的数据，如 key="#id"
    - allEntries = true：指定清除这个缓存中所有数据。
    - beforeInvocation = true：true在方法之前执行；默认false在方法之后执行,出现异常则不会清除缓存
7. @CacheConfig 指定缓存公共属性值  
 @CacheConfig(cacheNames = "user") 指定在类上，其他方法上就不需要写缓存名。

底层原理分析：















```
1 package com.mengxuegu.springboot;
2
3 import org.mybatis.spring.annotation.MapperScan;
4 import org.springframework.boot.SpringApplication;
5 import org.springframework.boot.autoconfigure.SpringBootApplication;
6 import org.springframework.cache.annotation.EnableCaching;
7
8 /**
9  * 0 = "org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration"
10  * 1 = "org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration"
11  * 2 = "org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration"
12  * 3 = "org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration"
13  * 4 = "org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration"
14  * 5 = "org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration"
15  * 6 = "org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration"
16  * 7 = "org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration"
17  * 8 = "org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration"[默认缓存]
18  * 9 = "org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration"
19  * 分析源码：
20  * 1.默认采用的是SimpleCacheConfiguration 使用 ConcurrentMapCacheManager
21  * 2. getCache 获取的是 ConcurrentMapCache 缓存对象进行存取数据,它使用ConcurrentMap<Object, Object>对
   象进行缓存数据
22  * @Cacheable(cacheNames = "user")
23  * 第一次请求时:
24  * 3.当发送第一次请求时,会从cacheMap.get(name)中获取有没有ConcurrentMapCache缓存对象,如果没有则创建
   出来,
25  * 并且创建出来的key就是通过@Cacheable(cacheNames = "user")标识的name值
26  * 4.接着会从ConcurrentMapCache里面调用lookup获取缓存数据,通过key值获取的,
27  * 默认采用的是service方法中的参数值,如果缓存中没有获取到,则调用目标方法进行获取数据,获取之后则再将它放到
   缓存中(key=参数值,value=返回值)
28  *
29  * 第二次请求:
30  * 5. 如果再次调用 则还是先ConcurrentMapCacheManager.getCache()获取缓存对象,如果有则直接返回,如果没有
   则创建
31  * 6. 然后再调用 ConcurrentMapCache.lookup方法从缓存中获取数据, 如果缓存有数据则直接响应回去,不会再去调
   用目标方法,
32  *
33  * 第三次请求与第二次请求一样.
34  * 如果缓存中没有缓存管理器,则与第一次请求一致
35  *
36  */
37 @EnableCaching //开启注解版的缓存
38 @MapperScan("com.mengxuegu.springboot.mapper")
39 @SpringBootApplication
40 public class SpringBoot13CacheApplication {
41
42     public static void main(String[] args) {
43         SpringApplication.run(SpringBoot13CacheApplication.class, args);
44     }
45 }
```

## 第15章 Spring Boot 整合 Redis 实战




- 在实际开发中，一般使用缓存中间件：redis、ehcache、memcache；导入了对应的组件依赖，就可以使用对应的缓存。
- 我们使用 Spring Boot 整合 Redis 作为缓存

### 15.1 安装 Redis 服务与客户端

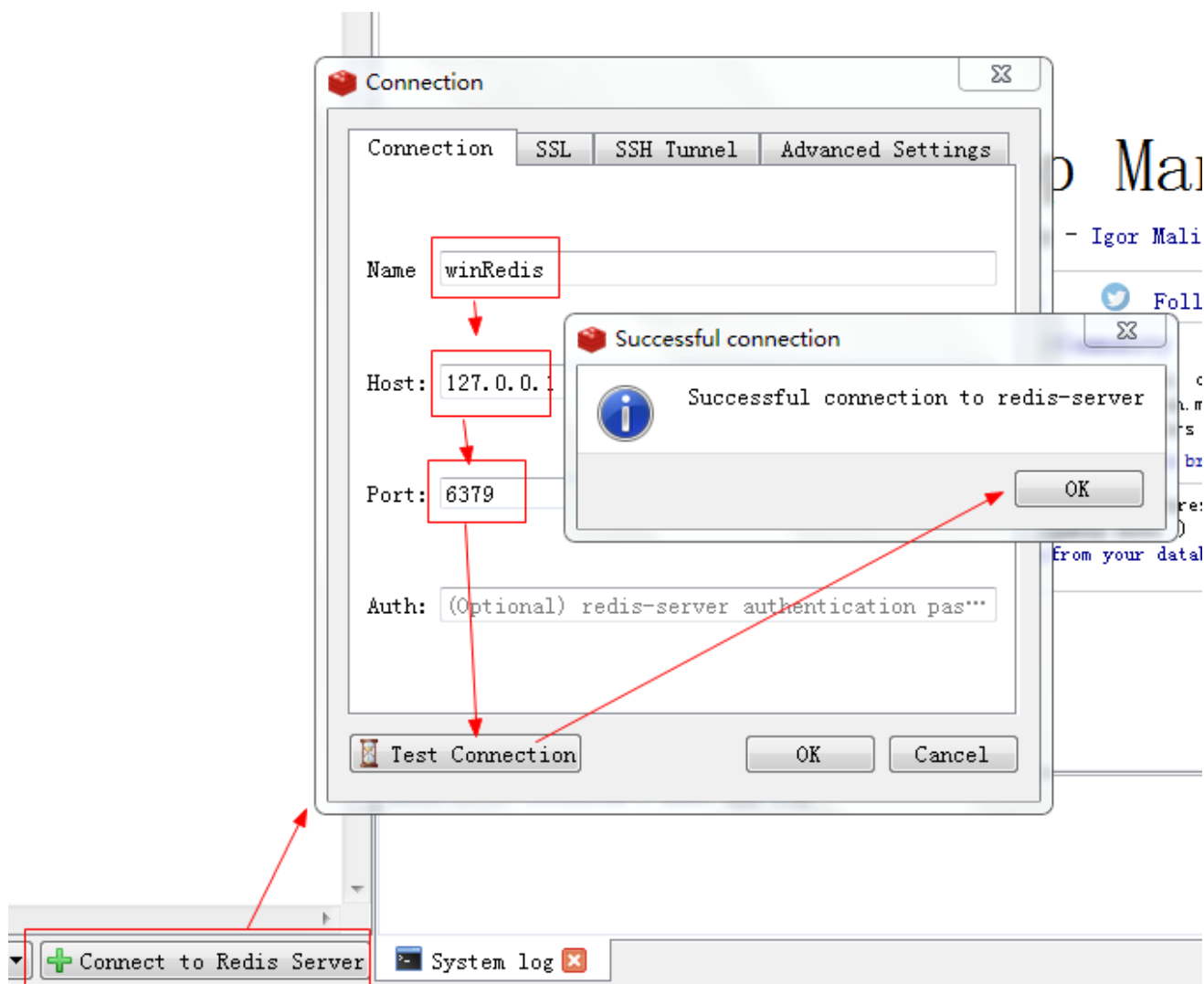
- Redis 中文官网：<http://www.redis.cn/>
- Redis 下载：
  - Linux版本：<http://download.redis.io/releases/>
  - Windows（微软开发维护的）：<https://github.com/MicrosoftArchive/redis/releases>
- Redis直接解压 Redis-x64-3.2.100.zip 即可，点击以下 redis-server.exe, 默认端口号：6379

	EventLog.dll	应用程序扩展	1 KB
	Redis on Windows Release Notes.do...	DOCX 文档	13 KB
	Redis on Windows.docx	DOCX 文档	17 KB
	redis.windows.conf	配置文件	48 KB
	redis.windows-service.conf	CONF 文件	48 KB
	redis-benchmark.exe	应用程序	400 KB
	redis-benchmark.pdb	PDB 文件	4,268 KB
	redis-check-aof.exe	应用程序	251 KB
	redis-check-aof.pdb	PDB 文件	3,436 KB
	redis-cli.exe	应用程序	488 KB
	redis-cli.pdb	PDB 文件	4,420 KB
	redis-server.exe	应用程序	1,628 KB
	redis-server.pdb	PDB 文件	6,916 KB
	Windows Service Documentation.docx	DOCX 文档	14 KB

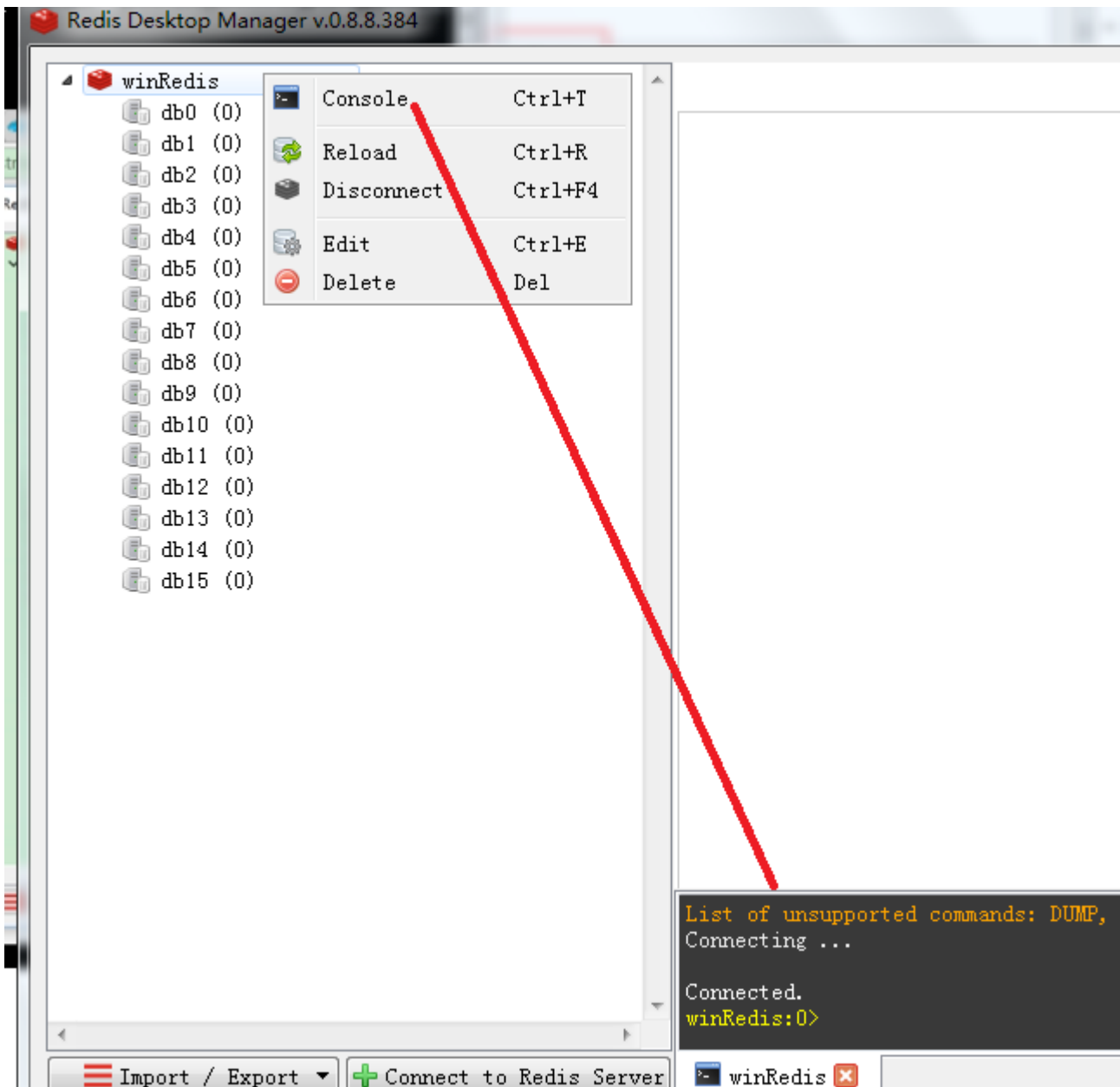
- 安装 Redis 可视化客户端 redis-desktop-manager-0.8.8.384.exe

	redis-5.0.0.tar.gz	360压缩	1,903 KB
	redis-desktop-manager-0.8.8.384.exe	应用程序	27,828 KB
	Redis-x64-3.2.100.zip	360压缩 ZIP 文件	5,102 KB

- 连接 redis 服务器



- 打开操作 Redis 命令行窗口



## 15.2 Redis 五种数据类型

Redis中所有的数据都是字符串。命令不区分大小写，key是区分大小写的。

- **五种数据类型:**

- String : <key, value>
- Hash : <key, fields-values>
- List : 有顺序可重复
- Set : 无顺序不可重复
- Sorted Sets (zset) : 有顺序, 不能重复

- **String : <key, value>**

- set、get

- incr : 加一 (生成id)
- decr : 减一
- append : 追加内容

```
1 winRedis:0>set id 5
2 "OK"
3 winRedis:0>get id
4 "5"
5 winRedis:0>incr id
6 "6"
7 winRedis:0>decr id
8 "5"
9
10 winRedis:0>set desc hello
11 "OK"
12 winRedis:0>append desc world
13 "10"
14 winRedis:0>get desc
15 "helloworld"
```

- del key\_name : 删除指定
- keys \* : 查看所有的 key
- **Hash : <key , fields-values>**
  - 相当于一个key对于一个Map , Map中还有key-value, 使用hash对key进行归类。
  - Hset : 向hash中添加内容
  - Hget : 从hash中取内容

```
1 winRedis:0> HSET myhash f1 hello
2 (integer) 1
3 winRedis:0> HGET myhash f1
4 "Hello"
```

- **List : 有顺序可重复**
  - lpush : 向List中左边添加元素
  - lrange : 查询指定范围的所有元素
  - rpush : 向List中右边添加元素
  - lpop : 弹出List左边第一个元素
  - rpop : 弹出List右边第一个元素

```
1 winRedis:0>lpush list1 a b
2 "2"
3
4 winRedis:0>lrange list1 0 -1
5 1) "b"
6 2) "a"
7
```



```
8 winRedis:0>rpush list1 1 2
9 "4"
10
11 winRedis:0>lrange list1 0 -1
12 1) "b"
13 2) "a"
14 3) "1"
15 4) "2"
16
17 winRedis:0>lpop list1
18 "b"
19
20 winRedis:0>lrange list1 0 -1
21 1) "a"
22 2) "1"
23 3) "2"
24
25 winRedis:0>rpop list1
26 "2"
27
28 winRedis:0>lrange list1 0 -1
29 1) "a"
30 2) "1"
```

- **Set : 元素无顺序，不能重复**

- sadd : 添加一个或多个元素到集合中
- smembers : 获取所有元素
- srem : 移除指定的元素

```
1 winRedis:0>sadd set1 a b c
2 "3"
3
4 winRedis:0>smembers set1
5 1) "b"
6 2) "a"
7 3) "c"
8
9 winRedis:0>srem set1 a
10 "1"
11
12 winRedis:0>smembers set1
13 1) "b"
14 2) "c"
```

- **SortedSet ( zset ) : 有顺序，不能重复**

- zadd key值 元素得分 元素 : 添加一个或多个元素到有序列Set中，按元素得分由小到大排列
- zrange : 查询指定范围的所有元素
- zrem : 移除指定的元素

```
1 winRedis:0>zadd zset1 3 a 5 b 1 c 4 d
2 "4"
3
4 winRedis:0>zrange zset1 0 -1
5 1) "c"
6 2) "a"
7 3) "d"
8 4) "b"
9
10 winRedis:0>zrem zset1 a
11 "1"
12
13 winRedis:0>zrange zset1 0 -1
14 1) "c"
15 2) "d"
16 3) "b"
17
18 #查询所有的元素并显示得分
19 winRedis:0>zrange zset1 0 -1 withscores
20 1) "c"
21 2) "1"
22 3) "d"
23 4) "4"
24 5) "b"
25 6) "5"
```

## 15.3 Spring Boot 整合 Redis

1. 引入 Redis 启动器：`spring-boot-starter-data-redis`
2. 在 `application.properties` 中配置 redis 连接地址
3. 使用 `RedisTemplate` 操作 Redis，参考 `RedisAutoConfiguration`
  - `redisTemplate.opsForValue();` //操作String
  - `redisTemplate.opsForHash();` //操作Hash
  - `redisTemplate.opsForList();` //操作List集合
  - `redisTemplate.opsForSet();` //操作Set集合
  - `redisTemplate.opsForZSet();` //操作有序Set集合
4. 自定义 Redis 配置类，指定Json序列化器

```
1 package com.mengxuegu.springboot;
2
3 import com.mengxuegu.springboot.entities.User;
4 import com.mengxuegu.springboot.service.UserService;
5 import org.junit.Test;
6 import org.junit.runner.RunWith;
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.boot.test.context.SpringBootTest;
9 import org.springframework.data.redis.core.RedisTemplate;
```

```
10 import org.springframework.data.redis.core.StringRedisTemplate;
11 import org.springframework.test.context.junit4.SpringRunner;
12
13 import java.util.List;
14
15 @RunWith(SpringRunner.class)
16 @SpringBootTest
17 public class SpringBoot13CacheApplicationTests {
18
19     //操作的是复杂类型，User
20     @Autowired
21     RedisTemplate redisTemplate;
22
23     //针对的都是操作字符串
24     @Autowired
25     StringRedisTemplate stringRedisTemplate;
26
27     //自定义的json序列化器
28     @Autowired
29     RedisTemplate jsonRedisTemplate;
30
31     @Autowired
32     UserService userService;
33
34     /**
35      * 五大数据类型
36      *      stringRedisTemplate.opsForValue();//操作字符串
37      *      stringRedisTemplate.opsForList();//操作List
38      *      stringRedisTemplate.opsForSet();//操作Set
39      *      stringRedisTemplate.opsForZSet();//操作ZSet
40      *      stringRedisTemplate.opsForHash();//操作Hash
41      */
42     @Test
43     public void contextLoads() {
44         // stringRedisTemplate.opsForValue().set("name", "mengxuegu");
45         String name = stringRedisTemplate.opsForValue().get("name");
46         System.out.println(name);//mengxuegu
47
48         // stringRedisTemplate.opsForList().leftPush("myList", "a");
49         // stringRedisTemplate.opsForList().leftPushAll("myList", "b", "c", "d");
50         List<String> myList = stringRedisTemplate.opsForList().range("myList", 0, -1);
51         System.out.println(myList);//[d, c, b, a]
52
53     }
54
55
56     @Test
57     public void testRedis() {
58         //当我们导入了reids的启动器之后，springboot会采用redis作为默认缓存，simple缓存就没有匹配上了
59         User user = userService.getUserById(4);
60         //保存的数据对象必须序列化 implements Serializable
61         //因为redisTemplate默认采用的是jdk序列化器
62
63         // redisTemplate.opsForValue().set("user", user);
```

```
63     User user1 = (User)redisTemplate.opsForValue().get("user");
64     System.out.println(user1);
65
66     jsonRedisTemplate.opsForValue().set("user2", user);
67
68 }
69 }
70
```

```
5. 1 package com.mengxuegu.springboot.config;
2
3 import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6 import org.springframework.data.redis.connection.RedisConnectionFactory;
7 import org.springframework.data.redis.core.RedisTemplate;
8 import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
9
10 import java.net.UnknownHostException;
11
12 /**
13  * @Author: 梦学谷
14  */
15 @Configuration
16 public class RedisConfig {
17
18
19     @Bean
20     public RedisTemplate<Object, Object> jsonRedisTemplate(
21         RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
22         RedisTemplate<Object, Object> template = new RedisTemplate<>();
23         template.setDefaultSerializer(new Jackson2JsonRedisSerializer(Object.class));
24         template.setConnectionFactory(redisConnectionFactory);
25         return template;
26     }
27
28
29 }
```

- Redis工具类

```
1 package com.mengxuegu.springboot.utils;
2
3 /**
4  * @Author: 梦学谷
5  */
6
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.data.redis.core.RedisTemplate;
9 import org.springframework.stereotype.Component;
10 import org.springframework.util.CollectionUtils;
```

```
11
12 import java.util.List;
13 import java.util.Map;
14 import java.util.Set;
15 import java.util.concurrent.TimeUnit;
16
17 /**
18  * Redis工具类
19  */
20 @Component
21 public class RedisClient {
22
23     @Autowired
24     private RedisTemplate redisTemplate;
25
26     /**
27      * 指定缓存失效时间
28      * @param key 键
29      * @param time 时间(秒)
30      * @return
31      */
32     public boolean expire(String key, long time) {
33         try {
34             if (time > 0) {
35                 redisTemplate.expire(key, time, TimeUnit.SECONDS);
36             }
37             return true;
38         } catch (Exception e) {
39             e.printStackTrace();
40             return false;
41         }
42     }
43
44     /**
45      * 根据key 获取过期时间
46      * @param key 键 不能为null
47      * @return 时间(秒) 返回0代表为永久有效
48      */
49     public long getExpire(String key) {
50         return redisTemplate.getExpire(key, TimeUnit.SECONDS);
51     }
52
53     /**
54      * 判断key是否存在
55      * @param key 键
56      * @return true 存在 false不存在
57      */
58     public boolean hasKey(String key) {
59         try {
60             return redisTemplate.hasKey(key);
61         } catch (Exception e) {
62             e.printStackTrace();
63             return false;
64         }
65     }
66 }
```

```
64     /**
65      * 删除缓存
66      * @param key 可以传一个值 或多个
67      */
68     @SuppressWarnings("unchecked")
69     public void del(String... key) {
70         if (key != null && key.length > 0) {
71             if (key.length == 1) {
72                 redisTemplate.delete(key[0]);
73             } else {
74                 redisTemplate.delete(CollectionUtils.arrayToList(key));
75             }
76         }
77     }
78
79     public void del(Integer key) {
80         this.del(String.valueOf(key));
81     }
82
83     // =====String=====
84     /**
85      * 普通缓存获取
86      * @param key 键
87      * @return 值
88      */
89     public Object get(String key) {
90         return key == null ? null : redisTemplate.opsForValue().get(key);
91     }
92
93     public Object get(Integer key) {
94         return this.get(String.valueOf(key));
95     }
96     /**
97      * 普通缓存放入
98      * @param key 键
99      * @param value 值
100     * @return true成功 false失败
101     */
102     public boolean set(String key, Object value) {
103         try {
104             redisTemplate.opsForValue().set(key, value);
105             return true;
106         } catch (Exception e) {
107             e.printStackTrace();
108             return false;
109         }
110     }
111
112     public boolean set(Integer key, Object value) {
113         return this.set(String.valueOf(key), value);
114     }
115     /**
116     * 普通缓存放入并设置时间
```

```
117     * @param key 键
118     * @param value 值
119     * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
120     * @return true成功 false 失败
121     */
122     public boolean set(String key, Object value, long time) {
123         try {
124             if (time > 0) {
125                 redisTemplate.opsForValue().set(key, value, time, TimeUnit.SECONDS);
126             } else {
127                 set(key, value);
128             }
129             return true;
130         } catch (Exception e) {
131             e.printStackTrace();
132             return false;
133         }
134     }
135     /**
136     * 递增
137     * @param key 键
138     * @param delta 要增加几(大于0)
139     * @return
140     */
141     public long incr(String key, long delta) {
142         if (delta < 0) {
143             throw new RuntimeException("递增因子必须大于0");
144         }
145         return redisTemplate.opsForValue().increment(key, delta);
146     }
147     /**
148     * 递减
149     * @param key 键
150     * @param delta 要减少几(小于0)
151     * @return
152     */
153     public long decr(String key, long delta) {
154         if (delta < 0) {
155             throw new RuntimeException("递减因子必须大于0");
156         }
157         return redisTemplate.opsForValue().increment(key, -delta);
158     }
159     // =====Map=====
160     /**
161     * HashGet
162     * @param key 键 不能为null
163     * @param item 项 不能为null
164     * @return 值
165     */
166     public Object hget(String key, String item) {
167         return redisTemplate.opsForHash().get(key, item);
168     }
169     /**
```

```
170     * 获取hashCode对应的所有键值
171     * @param key 键
172     * @return 对应的多个键值
173     */
174     public Map<Object, Object> hmget(String key) {
175         return redisTemplate.opsForHash().entries(key);
176     }
177     /**
178     * HashSet
179     * @param key 键
180     * @param map 对应多个键值
181     * @return true 成功 false 失败
182     */
183     public boolean hmset(String key, Map<String, Object> map) {
184         try {
185             redisTemplate.opsForHash().putAll(key, map);
186             return true;
187         } catch (Exception e) {
188             e.printStackTrace();
189             return false;
190         }
191     }
192     /**
193     * HashSet 并设置时间
194     * @param key 键
195     * @param map 对应多个键值
196     * @param time 时间(秒)
197     * @return true成功 false失败
198     */
199     public boolean hmset(String key, Map<String, Object> map, long time) {
200         try {
201             redisTemplate.opsForHash().putAll(key, map);
202             if (time > 0) {
203                 expire(key, time);
204             }
205             return true;
206         } catch (Exception e) {
207             e.printStackTrace();
208             return false;
209         }
210     }
211     /**
212     * 向一张hash表中放入数据,如果不存在将创建
213     * @param key 键
214     * @param item 项
215     * @param value 值
216     * @return true 成功 false失败
217     */
218     public boolean hset(String key, String item, Object value) {
219         try {
220             redisTemplate.opsForHash().put(key, item, value);
221             return true;
222         } catch (Exception e) {
```



```
223         e.printStackTrace();
224         return false;
225     }
226 }
227 /**
228  * 向一张hash表中放入数据,如果不存在将创建
229  * @param key 键
230  * @param item 项
231  * @param value 值
232  * @param time 时间(秒) 注意:如果已存在的hash表有时间,这里将会替换原有的时间
233  * @return true 成功 false失败
234  */
235 public boolean hset(String key, String item, Object value, long time) {
236     try {
237         redisTemplate.opsForHash().put(key, item, value);
238         if (time > 0) {
239             expire(key, time);
240         }
241         return true;
242     } catch (Exception e) {
243         e.printStackTrace();
244         return false;
245     }
246 }
247 /**
248  * 删除hash表中的值
249  * @param key 键 不能为null
250  * @param item 项 可以使多个 不能为null
251  */
252 public void hdel(String key, Object... item) {
253     redisTemplate.opsForHash().delete(key, item);
254 }
255 /**
256  * 判断hash表中是否有该项的值
257  * @param key 键 不能为null
258  * @param item 项 不能为null
259  * @return true 存在 false不存在
260  */
261 public boolean hHasKey(String key, String item) {
262     return redisTemplate.opsForHash().hasKey(key, item);
263 }
264 /**
265  * hash递增 如果不存在,就会创建一个 并把新增后的值返回
266  * @param key 键
267  * @param item 项
268  * @param by 要增加几(大于0)
269  * @return
270  */
271 public double hincr(String key, String item, double by) {
272     return redisTemplate.opsForHash().increment(key, item, by);
273 }
274 /**
275  * hash递减
```

```
276     * @param key 键
277     * @param item 项
278     * @param by 要减少记(小于0)
279     * @return
280     */
281     public double hincr(String key, String item, double by) {
282         return redisTemplate.opsForHash().increment(key, item, -by);
283     }
284
285     // =====set=====
286     /**
287      * 根据key获取Set中的所有值
288      * @param key 键
289      * @return
290      */
291     public Set<Object> sGet(String key) {
292         try {
293             return redisTemplate.opsForSet().members(key);
294         } catch (Exception e) {
295             e.printStackTrace();
296             return null;
297         }
298     }
299     /**
300      * 根据value从一个set中查询,是否存在
301      * @param key 键
302      * @param value 值
303      * @return true 存在 false不存在
304      */
305     public boolean sHasKey(String key, Object value) {
306         try {
307             return redisTemplate.opsForSet().isMember(key, value);
308         } catch (Exception e) {
309             e.printStackTrace();
310             return false;
311         }
312     }
313     /**
314      * 将数据放入set缓存
315      * @param key 键
316      * @param values 值 可以是多个
317      * @return 成功个数
318      */
319     public long sSet(String key, Object... values) {
320         try {
321             return redisTemplate.opsForSet().add(key, values);
322         } catch (Exception e) {
323             e.printStackTrace();
324             return 0;
325         }
326     }
327     /**
328      * 将set数据放入缓存
```

```
329      * @param key 键
330      * @param time 时间(秒)
331      * @param values 值 可以是多个
332      * @return 成功个数
333      */
334      public long sSetAndTime(String key, long time, Object... values) {
335          try {
336              Long count = redisTemplate.opsForSet().add(key, values);
337              if (time > 0)
338                  expire(key, time);
339              return count;
340          } catch (Exception e) {
341              e.printStackTrace();
342              return 0;
343          }
344      }
345      /**
346       * 获取set缓存的长度
347       * @param key 键
348       * @return
349       */
350      public long sGetSetSize(String key) {
351          try {
352              return redisTemplate.opsForSet().size(key);
353          } catch (Exception e) {
354              e.printStackTrace();
355              return 0;
356          }
357      }
358      /**
359       * 移除值为value的
360       * @param key 键
361       * @param values 值 可以是多个
362       * @return 移除的个数
363       */
364      public long setRemove(String key, Object... values) {
365          try {
366              Long count = redisTemplate.opsForSet().remove(key, values);
367              return count;
368          } catch (Exception e) {
369              e.printStackTrace();
370              return 0;
371          }
372      }
373      // =====list=====
374      /**
375       * 获取list缓存的内容
376       * @param key 键
377       * @param start 开始
378       * @param end 结束 0 到 -1代表所有值
379       * @return
380       */
381      public List<Object> lGet(String key, long start, long end) {
```

```
382     try {
383         return redisTemplate.opsForList().range(key, start, end);
384     } catch (Exception e) {
385         e.printStackTrace();
386         return null;
387     }
388 }
389 /**
390  * 获取list缓存的长度
391  * @param key 键
392  * @return
393  */
394 public long getListSize(String key) {
395     try {
396         return redisTemplate.opsForList().size(key);
397     } catch (Exception e) {
398         e.printStackTrace();
399         return 0;
400     }
401 }
402 /**
403  * 通过索引 获取list中的值
404  * @param key 键
405  * @param index 索引 index>=0时， 0 表头，1 第二个元素，依次类推；index<0时，-1，表尾，-2倒数第二
406  * 个元素，依次类推
407  * @return
408  */
409 public Object getIndex(String key, long index) {
410     try {
411         return redisTemplate.opsForList().index(key, index);
412     } catch (Exception e) {
413         e.printStackTrace();
414         return null;
415     }
416 }
417 /**
418  * 将list放入缓存
419  * @param key 键
420  * @param value 值
421  * @return
422  */
423 public boolean lSet(String key, Object value) {
424     try {
425         redisTemplate.opsForList().rightPush(key, value);
426         return true;
427     } catch (Exception e) {
428         e.printStackTrace();
429         return false;
430     }
431 }
432 /**
433  * 将list放入缓存
434  * @param key 键
```

```
434      * @param value 值
435      * @param time 时间(秒)
436      * @return
437      */
438      public boolean lSet(String key, Object value, long time) {
439          try {
440              redisTemplate.opsForList().rightPush(key, value);
441              if (time > 0)
442                  expire(key, time);
443              return true;
444          } catch (Exception e) {
445              e.printStackTrace();
446              return false;
447          }
448      }
449      /**
450       * 将list放入缓存
451       * @param key 键
452       * @param value 值
453       * @return
454       */
455      public boolean lSet(String key, List<Object> value) {
456          try {
457              redisTemplate.opsForList().rightPushAll(key, value);
458              return true;
459          } catch (Exception e) {
460              e.printStackTrace();
461              return false;
462          }
463      }
464      /**
465       * 将list放入缓存
466       * @param key 键
467       * @param value 值
468       * @param time 时间(秒)
469       * @return
470       */
471      public boolean lSet(String key, List<Object> value, long time) {
472          try {
473              redisTemplate.opsForList().rightPushAll(key, value);
474              if (time > 0)
475                  expire(key, time);
476              return true;
477          } catch (Exception e) {
478              e.printStackTrace();
479              return false;
480          }
481      }
482      /**
483       * 根据索引修改list中的某条数据
484       * @param key 键
485       * @param index 索引
486       * @param value 值
```

```
487     * @return
488     */
489     public boolean IUpdateIndex(String key, long index, Object value) {
490         try {
491             redisTemplate.opsForList().set(key, index, value);
492             return true;
493         } catch (Exception e) {
494             e.printStackTrace();
495             return false;
496         }
497     }
498     /**
499     * 移除N个值为value
500     * @param key 键
501     * @param count 移除多少个
502     * @param value 值
503     * @return 移除的个数
504     */
505     public long IRemove(String key, long count, Object value) {
506         try {
507             Long remove = redisTemplate.opsForList().remove(key, count, value);
508             return remove;
509         } catch (Exception e) {
510             e.printStackTrace();
511             return 0;
512         }
513     }
514 }
```

- ProviderService

```
1 package com.mengxuegu.springboot.service;
2
3 import com.mengxuegu.springboot.entities.Provider;
4 import com.mengxuegu.springboot.mapper.ProviderMapper;
5 import com.mengxuegu.springboot.utils.RedisClient;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.stereotype.Service;
8
9 /**
10  * http://localhost:8080/updateProvider?
11  * providerCode=123&providerName=aaa&people=xxx&phone=8888&address=99&pid=4
12  * @Author: 梦学谷
13  */
14 @Service
15 public class ProviderService {
16     @Autowired
17     RedisClient redisClient;
18
19     @Autowired
20     ProviderMapper providerMapper;
```

```
21
22 public Provider getProviderByPid(Integer pid){
23     Object obj = redisClient.get(pid);
24     if(obj != null) {
25         return (Provider) obj;
26     }
27     Provider provider = providerMapper.getProviderByPid(pid);
28     redisClient.set(pid, provider);
29     return provider;
30 }
31
32 public Integer updateProvider(Provider provider) {
33     int size = providerMapper.updateProvider(provider);
34     if(size > 0) {
35         redisClient.set(provider.getPid(), provider);
36     }
37     return size;
38 }
39
40 public Integer addProvider(Provider provider){
41     int size = providerMapper.addProvider(provider);
42     if(size > 0) {
43         redisClient.set(provider.getPid(), provider);
44     }
45     return size;
46 }
47
48 public Integer deleteProviderByPid(Integer pid) {
49     int size = providerMapper.deleteProviderByPid(pid);
50     if(size > 0) {
51         redisClient.del(pid);
52     }
53     return pid;
54 }
55
56 }
```

## 第16章 阿里云服务器部署项目与MySQL

主页：<https://www.aliyun.com/>

登录：<https://account.aliyun.com/login/login.htm>

### 16.1 介绍阿里云ESC服务器

- 停止、重启服务器



## 16.2 服务器与域名绑定

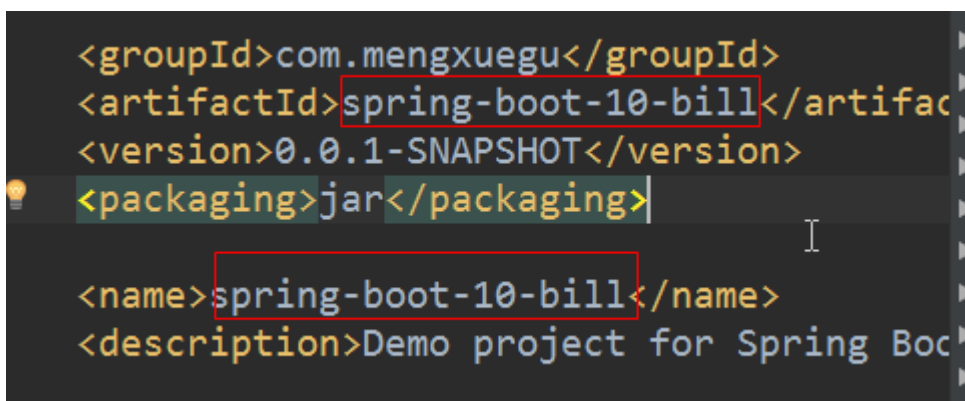






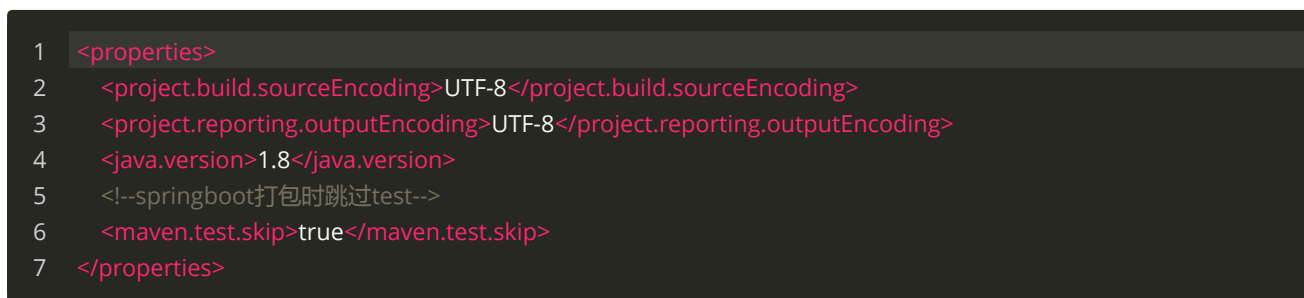
## 16.3 打包常见错误

1. 如果 maven projects 找不到Module ,看下pom.xml的名字是否正确



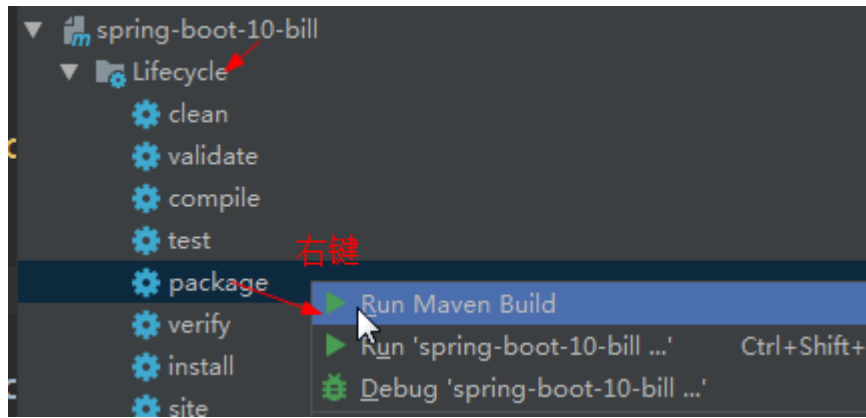
1. 如果 测试类中有错误，则无法正常打包，可在 pom.xml 中忽略测试类

在 <project> 标签下的 <properties> 标签中加入 <maven.test.skip>true</maven.test.skip>



## 16.4 打包本地部署项目

- 打成 jar包



如下效果，则打包成功

```
[INFO] --- spring-boot-maven-plugin:2.1.0.RELEASE:repackage (repackage) @ spring-boot-10-bill ---
[INFO] Replacing main artifact with repackaged binary
[INFO] ---
[INFO] BUILD SUCCESS
```

所在路径

- 本地测试

cmd 进入jar包所在目录，运行jar包启动项目，指定80端口号，直接访问ip即可

```
1 ...spring-boot-10-bill\target>java -jar spring-boot-10-bill-0.
2 0.1-SNAPSHOT.jar --server.port=80
```

访问：<http://localhost/>

## 16.5 阿里云部署 MySQL 服务

参考文档：软件\mysql相关安装包\阿里云服务器CentOS7.x安装MySQL5.7.2版本.doc

## 16.6 阿里云服务器安装JDK

- 上传解压：

1. 用FileZilla 上传将 JDK安装包上传到服务器 /opt目录里面，这里安装jdk1.8

2. 解压文件：

```
# cd /opt
```

```
# tar -zxvf jdk的文件名
```

3. ls 查看解压后的jdk

4. 将jdk移动到/home目录

```
# mv jdk1.8.0_171/ /home/
```

- 配置环境变量

```
# vim /etc/profile
```

在末尾行添加，打开后按编辑，按ctrl+c停止编辑，然后 :wq 保存退出

```
1 export JAVA_HOME=/home/jdk1.8.0_171
2 export PATH=$PATH:$JAVA_HOME/bin:
```

使更改的配置立即生效

```
# source /etc/profile
```

- java -version 查看JDK版本信息，如果显示出1.8.0证明成功

## 16.6 阿里云发布项目

- 向mysql服务器执行项目sql脚本
- 添加安全组规则-开放端口号8080端口与80端口

官网手册: [https://help.aliyun.com/document\\_detail/25471.html?spm=5176.100241.0.0.lnejPl](https://help.aliyun.com/document_detail/25471.html?spm=5176.100241.0.0.lnejPl)



- 后台进程进行项目

```
1 [root@izwz9e7 opt]# nohup java -jar spring-boot-10-bill-0.0.1-SNAPSHOT.jar --server.port=80 &
```

- 停止运行项目

```
1 查看进程
2 [root@izwz9e7 opt]# ps -ef|grep java
3 root    4903  4876  23 19:35 pts/1    00:00:13 java -jar spring-boot-10-bill-0.0.1-SNAPSHOT.jar
4 root    4939  4876   0 19:36 pts/1    00:00:00 grep --color=auto java
5
6 结束进程
7 [root@izwz9e7 opt]# kill -9 4903
```