尚马教育 JAVA 高级课程

SpringBoot

文档编号：C16

创建日期： 2018-10-18

最后修改日期：2021-02-26

版 本 号：V1.0

电子版文件名：尚马教育-第三阶段-15.activiti 工作流引擎.docx

文档修改记录：

更新日期	更新作者	更新说明	版本号
2018-10-18	王绍成	初始版本	V1.0
2019-09-12	王绍成	添加跨域配置	V2.0
2021-02-26	冯勇涛	添加 fastdfs-client	V3.0

目录

1.1 SpringBoot 简介

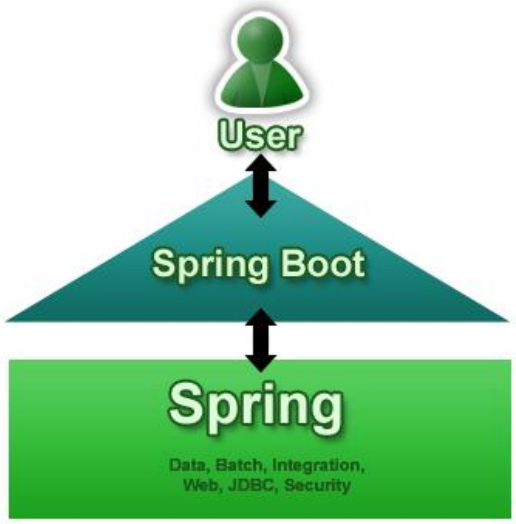
1.1.1 什么是 SpringBoot

- JavaEE 的一站式解决方案，一套真正的 Spring 全家桶应用
- Spring Boot 是为了简化 Spring 应用的创建、运行、调试、部署等而出现的。

- 使用它可以做到专注于 Spring 应用的开发，无需过多关注 XML 的配置。
- 它提供了一堆依赖包，并已经按照使用习惯解决了依赖问题。使用默认方式实现快速开发。
- 提供大多数项目所需的非功能特性，诸如：嵌入式服务器(Jetty)、安全、心跳检查、外部配置等。
- Spring Boot 不生成代码，完全无需 XML 配置，创建即用。

1.1.2 SpringBoot 开发者简介

SpringBoot 默认集成 Spring 生态中的所有框架，只需要通过一些简短的配置即可使用。开发者只需要使用 SpringBoot 开发而不再需要关注 Spring 生态中的各种框架配置。



1.1.3 SpringBoot 2.4 环境要求

- SpringBoot2.4 基于 JDK8 开发，JDK1.7 无法运行
- Maven3 以上
- 使用 IntelliJ IDEA 或 Spring 官方的 STS 开发工具

1.1.4 SpringBoot 2.4 特性

- 创建独立的 Spring 应用程序
- 大大简化了安全自动配置
- 入门的“入门”依赖项，以简化构建配置
- 支持嵌入式 Netty

- Tomcat, Undertow 和 Jetty 均已支持 HTTP/2

1.2 第一个 SpringBoot 项目

1.2.1 使用 IDEA Maven 创建

1. 在 pom 中加入 SpringBoot 依赖

```
<!-- Inherit defaults from Spring Boot -->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.0</version>
        <relativePath/>
    </parent>

<!-- Add typical dependencies for a web application -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<!-- Package as an executable jar -->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

2. 创建 SpringBoot 启动类（必须在所有类的最上层包中）

```
@SpringBootApplication
public class SpringBootTestApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootTestApplication.class,args);
    }
}
```

3. 创建一个 SpringMVC 的控制类

```java
@Controller
@RequestMapping("/hello")
public class HelloController {

    @RequestMapping("/test1")
    @ResponseBody
    public String test1() {
        return "index";
    }
}
```

1.2.1.1 在 IDEA 中直接执行

4. 运行 SpringBoot 启动类（直接右键 Run 即可）

5. 访问 localhost:8080/hello/test1

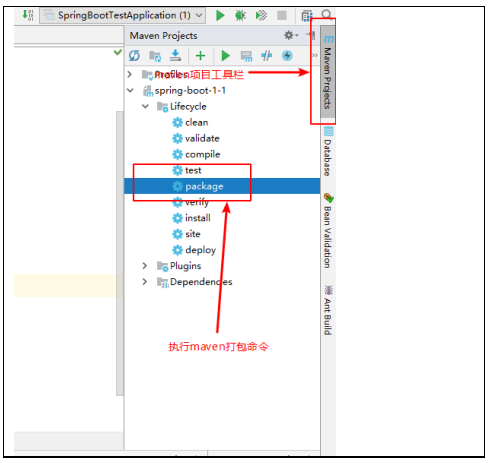
1.2.1.2 使用打包成 Jar 的方式执行

1. 检查 pom 文件中是否有当前 maven 打包插件支持

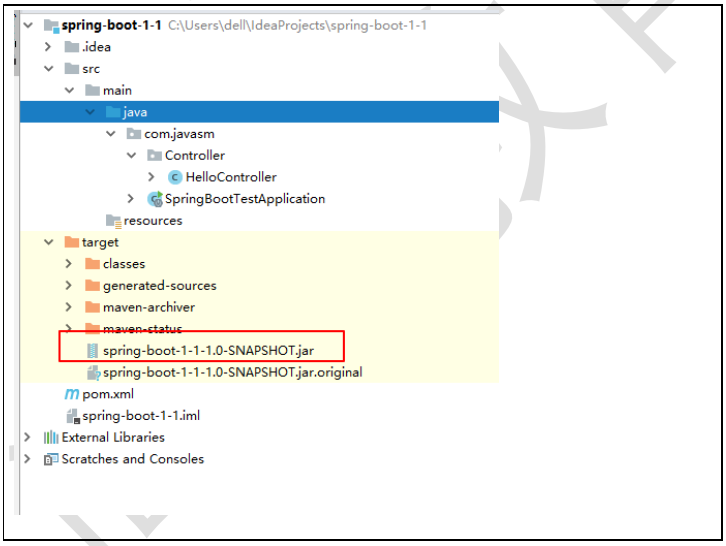
```xml
<!-- Package as an executable jar -->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

SpringBoot项目最初引入的pom配置中包含了Maven插件

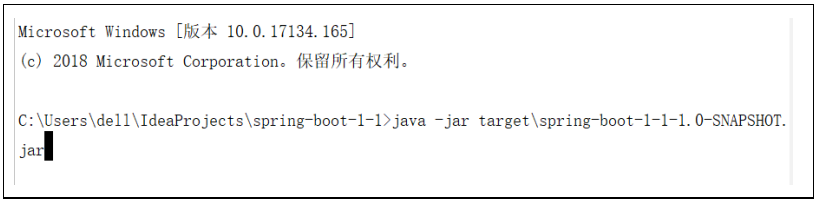
2. 使用 maven 工具将当前项目打包



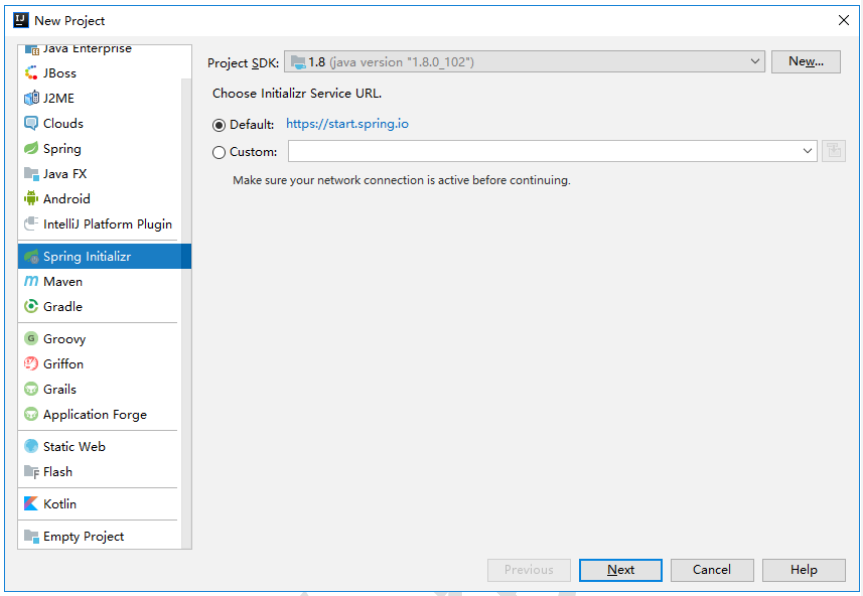
3. 在当前项目中的 target 目录中找到生成好的 Jar 包

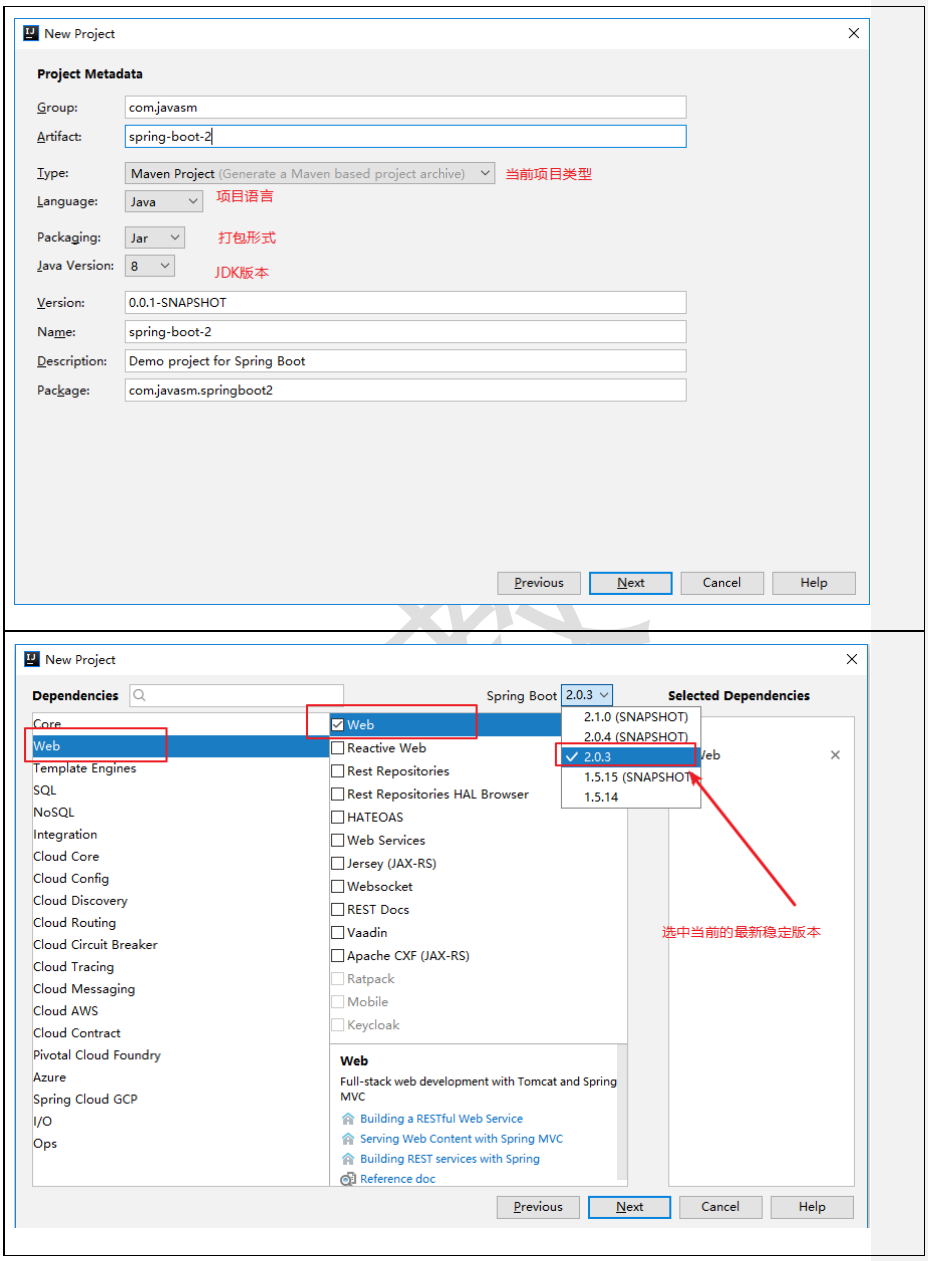


4. 可以将 Jar 拷贝至其他地方使用 cmd 执行启动命令

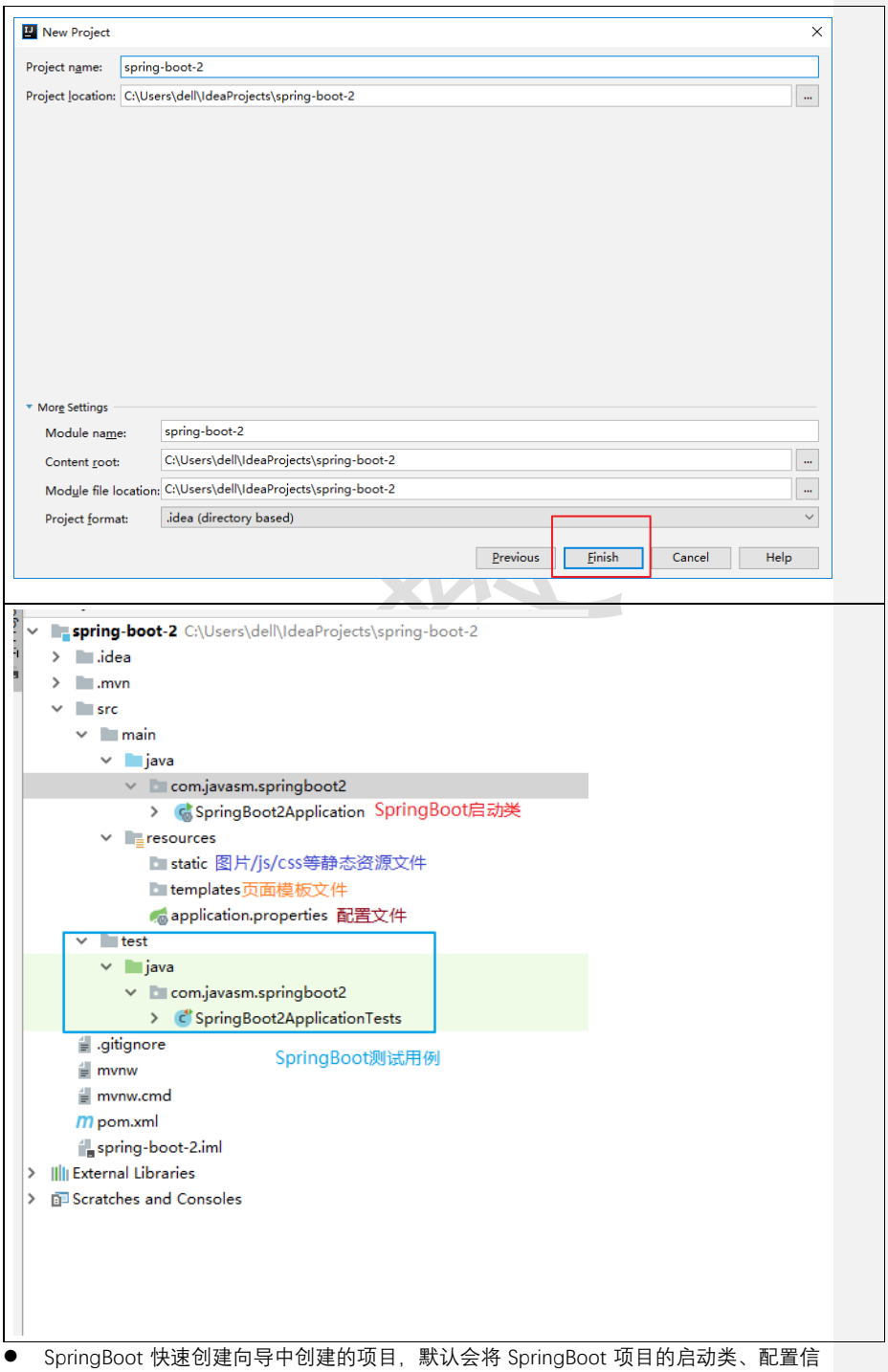


1.2.2 使用 IDEA 中的 Spring 创建向导来创建




New Project
Project Metadata
Group: com.javasm
Artifact: spring-boot-2
Type: Maven Project (Generate a Maven based project archive)
当前项目类型
Language: Java
项目语言
Packaging: Jar
打包形式
Java Version: 8
JDK版本
Version: 0.0.1-SNAPSHOT
Name: spring-boot-2
Description: Demo project for Spring Boot
Package: com.javasm.springboot2
Previous
Next
Cancel
Help

New Project
Dependencies
Spring Boot 2.0.3
Selected Dependencies
2.1.0 (SNAPSHOT)
2.0.4 (SNAPSHOT)
2.0.3
1.5.15 (SNAPSHOT)
1.5.14
Core
Web
Template Engines
SQL
NoSQL
Integration
Cloud Core
Cloud Config
Cloud Discovery
Cloud Routing
Cloud Circuit Breaker
Cloud Tracing
Cloud Messaging
Cloud AWS
Cloud Contract
Pivotal Cloud Foundry
Azure
Spring Cloud GCP
I/O
Ops
Web
Reactive Web
Rest Repositories
Rest Repositories HAL Browser
HATEOAS
Web Services
Jersey (JAX-RS)
Websocket
REST Docs
Vaadin
Apache CXF (JAX-RS)
Ratpack
Mobile
Keycloak
Web
Full-stack web development with Tomcat and Spring MVC
Building a RESTful Web Service
Serving Web Content with Spring MVC
Building REST services with Spring
Reference doc
选中当前的最新稳定版本
Previous
Next
Cancel
Help



- SpringBoot 快速创建向导中创建的项目，默认会将 SpringBoot 项目的启动类、配置信

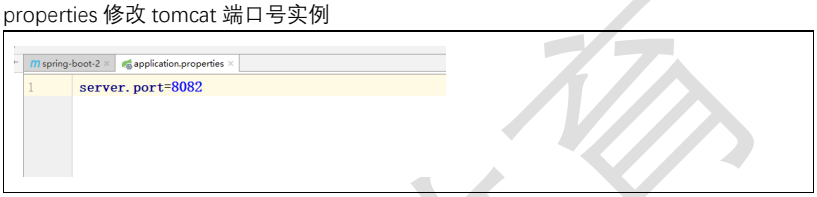
息、测试用例等一并创建。

- SpringBoot 默认不支持 JSP，官方推荐使用 thymeleaf 页面模板引擎，这里是 templates 就是用来存放页面模板的。
- application.properties 是 SpringBoot 的配置文件。关于 SpringBoot 内嵌 tomcat 端口号以及其他信息都是在这儿进行配置。同时 SpringBoot 也可以使用 yml 文件来描述配置。

1.3 配置文件详解

SpringBoot 使用 application.properties 来声明当前框架的配置信息。同时在 SpringBoot 中也可以使用 aplication.yml 类描述配置。

- properties 修改 tomcat 端口号实例



- 配置文件的优先级（优先级由高到低，优先使用高级别的配置信息）
  - 项目路径下的 config 文件夹
  - 项目根目录下
  - resources 目录下的 config 文件夹
  - resouces 目录下

1.3.1 第一个 SpringBoot 程序解析

1.3.1.1 POM 文件

```
<!-- Inherit defaults from Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
</parent>

<!-- Add typical dependencies for a web application -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

SpringBoot的父项目依赖

SpringBoot web启动器

1. 父项目的依赖中包含一个新的父依赖 spring-boot-dependencies,在这个父依赖中包含了当前版本的 SpringBoot 所有依赖 Jar 包的版本配置信息
2. SpringBoot 将所有的功能都抽取出来，做成一个个 starters，当我们需要使用对应的功

能时，导入对应的 starters 就可以使用【完整 starters】

Table 13.1. Spring Boot application starters

Name	Description
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis
spring-boot-starter-batch	Starter for using Spring Batch
spring-boot-starter-cache	Starter for using Spring Framework's caching support
spring-boot-starter-cloud-connectors	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra
spring-boot-starter-data-cassandra-reactive	Starter for using Cassandra distributed database and Spring Data Cassandra Reactive
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase
spring-boot-starter-data-couchbase-reactive	Starter for using Couchbase document-oriented database and Spring Data Couchbase Reactive
spring-boot-starter-data-elasticsearch	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch
spring-boot-starter-data-jpa	Starter for using Spring Data JPA with Hibernate
spring-boot-starter-data-ldap	Starter for using Spring Data LDAP
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB
spring-boot-starter-data-mongodb-reactive	Starter for using MongoDB document-oriented database and Spring Data MongoDB Reactive
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client

1.3.1.2 Application 类解析

```
@SpringBootApplication
public class SpringBootTestApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootTestApplication.class,args);
    }
}
```

- 在 SpringBoot 主程序中只有简短的几行代码即可完成整个应用的启动，完成这些的关键点在于**@SpringBootApplication 注解**

1.3.1.2.1 SpringBootApplication 注解解析

- 被@SpringBootApplication 标注的类就是当前 SpringBoot 应用的入口类。SpringBoot 会在当前类中查找 main 方法并执行

```
@SpringBootConfiguration      标注当前类是一个Spring配置类
@EnableAutoConfiguration      开启自动配置信息
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

- 在 SpringBootApplication 类中，使用了两个注解分别标注当前类是配置类且开启自动配置

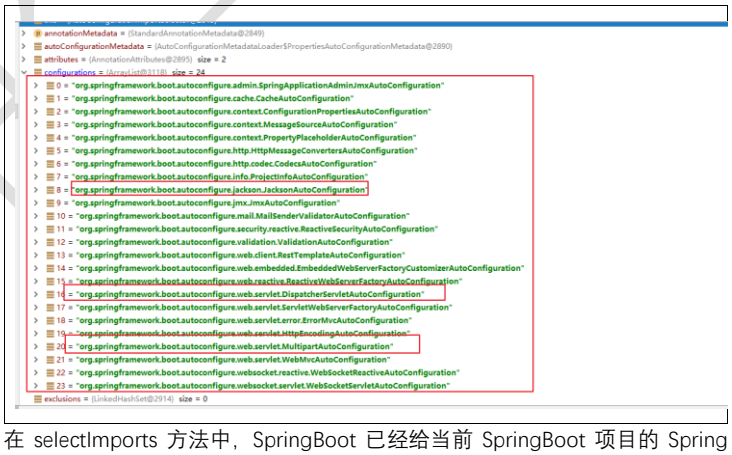
```
@AutoConfigurationPackage                          自动配置包
@Import(AutoConfigurationImportSelector.class)     导入需要使用的组件
public @interface EnableAutoConfiguration {
```

- 在 EnableAutoConfiguration 中配置了自动配置包和需要导入的组件。
  - AutoConfigurationPackage 实现了所有的包注册以及包扫描
    -

```
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {

}
```

自动配置包扫描注册器

    - 在 Registrar 静态内部类中，SpringBoot 会查找到当前启动类下所在包下的所有子类，并将他们注册到 Spring 容器中
  - AutoConfigurationImportSelector 实现了给当前容器自动添加框架配置



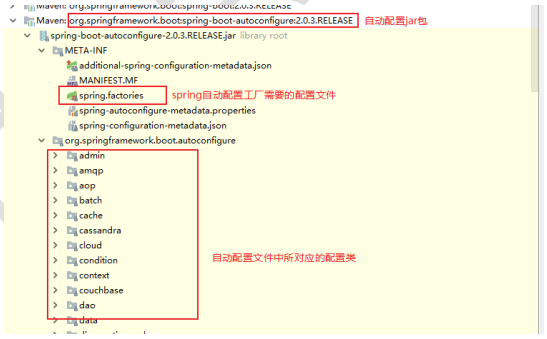
    - 在 selectImports 方法中，SpringBoot 已经给当前 SpringBoot 项目的 Spring Framework 的配置了默认配置。

```
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
    configurations = removeDuplicates(configurations);                    获取可匹配的配置信息
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    checkExcludedClasses(configurations, exclusions);
    configurations.removeAll(exclusions);
    configurations = filter(configurations, autoConfigurationMetadata);
    fireAutoConfigurationImportEvents(configurations, exclusions);
    return StringUtils.toStringArray(configurations);
}
```

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
        AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
            getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());        加载配置类到Spring容器中
    Assert.notEmpty(configurations,
            message: "No auto configuration classes found in META-INF/spring.factories. If you "
                    + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

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

```
RL> urls = classLoader != null ? classLoader.getResources( name: "META-INF/spring.factories") : ClassLoader.getSystemResources( name: "META-INF/spring.factories");
lueMap result = new LinkedMultiValueMap();
                                    从当前的类加载器中获取所有的spring.factories文件
sMoreElements()) {
(URL)urls.nextElement();
ce resource = new UrlResource(url);
s properties = PropertiesLoaderUtils.loadProperties(resource);
var6 = properties.entrySet().iterator();
```



- 且在 selectImport 中的 loadMetadata 方法中引入了一个 Path 常量。而该 Path 常量所指向的文件就是所有自动配置的属性配置文件

1.3.1.3 什么是 YML

YAML（YAML Ain't Markup Language）是“YAML 不是一种标记语言”的外语缩写；但为了强调这种语言以数据做为中心，而不是以置标语言为重点，。它是一种直观的能够被电脑识别的数据序列化格式，是一个可读性高并且容易被人类阅读，容易和脚本语言交互，用来表达资料序列的编程语言。

- 语法规则
  1、大小写敏感
  2、使用缩进表示层级关系
  3、禁止使用 tab 缩进，只能使用空格键
  4、缩进长度没有限制，只要元素对齐就表示这些元素属于一个层级。
  5、使用#表示注释
  6、字符串可以不用引号标注

1.3.1.4 语法实例

YML 可以方便直观的描述字符串、数字、时间、List 和 Map 类型的数据。

- 案例

配置 application.yml

```yaml
user:
    username: 张三    字符串类型
    age: 18    数值类型
    birthday: 2018/7/17 12:12:12    时间类型
    hobbys:    List集合
      - 打篮球
      - 踢足球
    des: {sex: 男, address: 郑州}    map类型
```

创建实体类来测试 yml 数据描述

```
Spring Boot Configuration Annotation Processor not found in classpath    Open Documentation...
@Component    将当前实体类注册为Spring容器组件
@ConfigurationProperties(prefix = "user")    从属性文件中获取配置信息，且只读取前缀为user的文档属性
public class UserInfo {
    private String username;
    private int age;
    private Date birthday;
    private List<String> hobbys;
    private Map<String,String> des;
```

当创建完对应的实体类之后，IDEA 会给出红色警告

按照官网提示，在 pom 文件中加入对应的可选依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

在当前项目的测试用例中进行测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot2ApplicationTests {

    @Resource
    private UserInfo user;
    @Test
    public void contextLoads() {
        System.out.println(user);
    }

}
```

声明容器中的UserInfo对象

输出容器中的属性的值

1.3.1.5 SpringBoot 配置属性一览

官方文档中给了 properties 所有可配置的选项。【参考文档】

常见配置如下：

```
#内嵌服务器端口号
server.port=8081
#当前项目访问路径
server.servlet.context-path=/javasm
```

1.4 SpringBoot 开发 Web

1.4.1 SpringBoot 静态资源

SpringBoot 中的静态资源存放在 resources 文件夹中，SpringBoot 默认识别该文件夹下的 public、static、resources 子文件夹。当出现同名文件时优先级由低到高。
当需要自定义静态文件目录时,可以在属性配置文件中加入

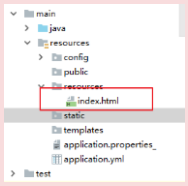
```
spring.resources.static-locations=hello,res
```

批注 [j1]: 可参考 WebMvcAutoConfigurationAdapter 下的 addResourceHandlers 方法

1.4.1.1 增加当前项目的欢迎页

在 SpringBoot 中，只要在上述三个静态资源文件夹下创建一个 index.html 即可作为欢迎页来使用。

批注 [j2]:



1.4.1.2 修改当前项目的浏览器图标

在 SpringBoot 中，开发者可以在静态资源文件夹中添加 favicon.ico 来更改浏览器中显示的图标

1.4.2 Thymeleaf 模板引擎

在 SpringBoot 官方推荐的是使用 Jar 包启动，让每一个项目更加轻量，更加方便部署运行。由于使用了内嵌的轻量 tomcat，所以 SpringBoot 默认不支持 JSP 页面引擎。
在 SpringBoot 中,官方更加推荐开发者使用 Thymeleaf 模板引擎来进行页面开发。Thymeleaf 提供了更简单易读的标签，来替代我们在传统 JSP 中所使用的 EL/JSTL 等标签技术。
SpringBoot 在 WebMVC 的视图解析中，默认查找的文件夹是在 templates 文件夹下查找。

1.4.2.1 什么是 Thymeleaf

Thymeleaf 是一个 Java 服务器端的模板引擎，Thymeleaf 的主要目标让开发者已更优雅的方式使用模板引擎，Thymeleaf 已 html 的形式存在，它可以在服务器中运行也可以作为独立的静态页面运行，这样让前端开发和后端开发能够更加快速的协作。同时它可以与 Spring 框架进行集成，同时它还也完美兼容目前的 HTML5 特性。当然对于 thymeleaf 来说，它可以做的事情不仅仅如此。【参考网站】

1.4.2.2 Thymeleaf 环境准备

在前面我们做 pom.xml 解析时，我们提到了 SpringBoot 将所有的功能抽取出来做成一个个的 starter 启动器。而我们在 SpringBoot 中需要使用 Thymeleaf 也就只需要引入对于的 starter 就可以使用了。

1.4.2.3 pom 添加 Thymeleaf 依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

1.4.2.4 Thymeleaf 的使用

1.4.2.4.1 基本使用案例

1.4.2.4.1.1 创建一个 controller

创建一个 controller 并往响应页面中传递一个参数

```
@Controller
public class TestController {
    @RequestMapping("/test")
    public String test(Model model){
        model.addAttribute("flag", true);
        model.addAttribute("msg","你好,Thymeleaf");
        return "test";
    }
}
```

1.4.2.4.1.2 在 html 中添加 thymeleaf 环境

在 templates 文件夹下创建一个 html 文件,并在 html 文件头增加以下标签引用

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

1.4.2.4.1.3 页面使用

在 body 中使用 thymeleaf 属性来显示后端传递的值

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>测试页面</title>
</head>
<body>
    <h4 th:text="${msg}"></h4>
</body>
</html>
```

当前网页的环境支持

使用thymeleaf获取当前后端传递的值并显示

1.4.2.4.1.4 开发小技巧

在实际的开发过程中,每次修改模板都需要能够快速的看到结果,也就是实时编译.而 thymeleaf 在集成中默认是开启了模板缓存的.可以通过下面的步骤来提高开发效率.

1. 在配置文件中关闭 thymeleaf 缓存.

```
spring.thymeleaf.cache=false
```

2. 每次修改完 thymeleaf 后使用 ctrl+F9 快捷键, 重新进行编译.

1.4.2.5 语法规则

1.4.2.5.1 th 属性概览

在 Thymeleaf 官方文档中第 10 章 **Attribute Precedence** 中对所有的 th 语法介绍.th 可以操作 html 中所有的属性,具体 demo 在文档第 4 章如下图.

Order	Feature	Attributes
1	Fragment inclusion	th:insert 引入/替换html片段：相当于jsp:include th:replace jsp:replace
2	Fragment iteration	th:each 循环遍历：相当于c:forEach
3	Conditional evaluation	th:if th:unless 条件判断：相当于c:if th:switch th:case
4	Local variable definition	th:object 变量声明：相当于c:set th:with
5	General attribute modification	th:attr th:attrprepend 添加或修改属性 th:attrappend
6	Specific attribute modification	th:value th:href 修改html标签的属性值 th:src ...
7	Text (tag body modification)	th:text 文本：转义特殊字符 th:utext 不转义特殊字符
8	Fragment specification	th:fragment 文档片段
9	Fragment removal	th:remove 删除文档元素

- 第一个应用示例
  - 在 html 中显示值文本是我们最常用的场景之一.

```
#HTML 标签文本方式
<!--在任意标签中插入文本(不解析特殊字符)-->
<h4 th:text="${msg}">  </h4>
<!--在任意标签中插入文本(解析特殊字符)-->
<h4 th:utext="${msg}"></h4>
#HTML 行内文本方式
<!--在 html 中直接显示值(不解析特殊字符)-->
[[${msg}]]
<!--在 html 中直接显示值(解析特殊字符)-->
[(${msg})]
```

  - 当应用在标签中时,想要拼接字符串则可以使用以下两种方式

```
<!--使用字符串+号连接的拼接方式-->
<h4 th:text="'这是 th:text 标签 : ' + ${msg}"></h4>
<!--使用||包含字符串和表达式进行拼接-->
<h4 th:text="|这是 th:text 标签 ${msg}|"></h4>
```

1.4.2.5.2 thymeleaf 表达式

在 thymeleaf 中也跟 el 一样,会有丰富的表达式支持.在 thymeleaf 官方文档的第四章 **Standard Expression Syntax** 对所有的表达式进行了一个总结.常用表达式如下:

语法名称	语法规则	语法示例	语法介绍

属性表达式(更多用法参照 4.2 节用法)	${...}	//获取对象属性
		${userInfo.name}
		${userInfo['name']}
		//获取数组下标为 0 的 name
		${userInfo[0].name}
		//获取 session 中的属性.作用域还有 param 和 application
		${session.username}
		//当作用域前面加了#则表示使用对象中的方法,其他作用域还有#request 和 #servletContext
		${#session.getAttribute('username')}
		//将当前请求中的变量进行日期转换
		${#calendars.format(date,'yyyy-MM-dd HH:mm:ss')}
			属性表达式的实质是 OGNL 表达式,使用方法类似于 el 表达式.开发者可以很方便的取出当前容器作用域中的值

选择表达式(更多用法参考第 4.3 节用法)	*{···}	 //在 th:object 声明下的子集可以直接使用属性选择器获取属性值 Name: Sebastian. Surname: Pepper. Nationality: Saturn. 	选择表达式一般搭配 object 和属性表达式一起使用.当 HTML 中发父级元素中声明了 object 则在子标签中可以使用选择表达式来直接获取 object 中的属性
URL 表达式(更多用法参考第 4.4 节用法)	@{···}	//使用(参数名=参数值,参数名=参数值)来拼接参数,会自动追加?号 <!-- 'http://localhost:8080/gtvg/order/details?orderId=3' --> <a th:href="@{http://localhost:8080/gtvg/order/details(orderId=${o.id})}">view <!-- '/gtvg/order/details?orderId=3'--> <a th:href="@{/order/details(orderId=${o.id})}">view <!-- '/gtvg/order/3/details?orderId=3'--> <a th:href="@{'/order/'+${orderId}+'/details'(orderId=${o.id})}">view	URL 表达式可以更加方便的声明超连接,可以在超链接中加载变量及拼接,并且会自动追加当前项目的访问路径

文档表达式
(更多用法参考第 8 节用法)
	~{…}	//使用~{文档文件名(可不写) :: 文档名};当文档名不写则表示引用当前 html 中的文档
		//创建 thymeleaf 公共模板
		#tmp.html
		`<div th:fragment="publicTemplate">`
		`hello ,公共文档`
		`</div>`
		//使用文档表达式进行文档引用.
		#test.html
		`<!—将文档中的代码插入到当前 div 中-->`
		`<div th:insert="~{tmp :: publicTemplate }"></div>`
		`<!—使用文档代码替换当前 div-->`
		`<div th:replace="~{tmp :: publicTemplate }"></div>`
		//运行结果
		`<div><div> hello ,公共文档</div></div>`
		`<div>hello ,公共文档</div>`
		也可以直接写 html 地址,下面路径会自动引用/templates/common/left.html
		`<div th:insert="common/left"></div>`
		`<div th:replace="common/left"></div>`
		`<div th:include="common/left"></div>`
			文档表达式相当于 jsp 中的 include, 可以任意引用其他公共的 html 代码,减少前端代码冗余

1.4.2.5.3 th 常用标签介绍

th 可以用来做数据显示、HTML 属性操作、逻辑判断、集合迭代、文档引用等.可以理解为是 JSP 中的 EL 和 JSTL 的更强替代品.

Th:value th:text th:action

标签名称	标签规则	标签示例	标签介绍

条件判断
(更多用法参考第 7 节)
	th:if	//判定值,满足条件则显示当前 div
		`<div th:if="${counts>= 100}">`
		`大于等于 100`
		`</div>`
			If 标签的使用方法与 c:if 基本一致

	th:switch	//使用 switch 标签与属性表达式来判定
		`<div th:switch="${user.role}">`
		//case 中可以直接进行值或表达式值判定
		`<p th:case="'admin'">系统管理员</p>`
		`<p th:case="${roles.manager}">主管</p>`
		//case 中的值为*号时表示默认
		`<p th:case="*">其他</p>`
		`</div>`
			Switch-case 标签组合的基本使用与 Java 一致.能够更好的判断不同的值给出不同的显示情况.也同样包含默认值.

<table>
<tr>
<td>集合迭代
(更多用法参考第 6 节)</td>
<td>th:each</td>
<td>//th:each="迭代元素名 : 迭代的数组"
#基本迭代示例
<table>
<tr>
<td>名称</td>
<td>性别</td>
</tr>
<tr th:each="user : ${users}">
<td th:text="${user.username}"></td>
<td th:text="${user.sex}"></td>
</tr>
</table>

//th:each="迭代元素名,迭代状态名 : 迭代的数组"
#迭代状态示例
<table>
<tr>
<td>名称</td>
<td>性别</td>
</tr>
<tr th:each="user,userStat : ${users}">
<td th:text="${user.username}"></td>
<td th:text="${user.sex}"></td>
<td th:text="${userStat.index}">状态变量:索引(0 开始)</td>
<td th:text="${userStat.count}">状态变量：当前行</td>
<td th:text="${userStat.size}">状态变量：总长度</td>
<td th:text="${userStat.even}">状态变量：是偶数?</td>
<td th:text="${userStat.odd}">状态变量：是奇数?</td>
<td th:text="${userStat.first}">状态变量：是第一位?</td>
<td th:text="${userStat.last}">状态变量：是最后一位?</td>
</tr>
</table></td>
<td>th:each 与 c:forEach 标签用法也差不多,在 each 中,想要获得当前数组的遍历标记,则需要在遍历对象名后增加一个新的参数名即可使用.</td>
</tr>
</table>

1.4.3 SpingBoot 扩展

在 SpringBoot2.0 版本中,使用了一个核心接口 WebMvcConfigurer 来定义用户自定义的扩展. 比如表单提交中的时间转换,默认访问路径映射,拦截器等.

1.4.3.1 控制器映射

在SpringBoot的开发中,静态资源中的文件都是可以直接访问的,而我们的页面在某些情况下是不希望用户能直接通过在浏览器中填写 URL 直接访问,那么我们就会把页面放在用户不可直接访问的目录,由所有的 controller 来做一一映射跳转.
那么在这种情况下,我们可能就会需要定义多个跳转页面的 Controller methed 来做页面映射. 而现在在 SpringBoot 中的 WebMvcConfigurer 中直接提供了添加多个控制器的映射的方法.

1.4.3.1.1 addViewControllers 的使用

```
@Configuration//标志当前类在启动的时候会被加载
public class MyConfiguration implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        //urlPath:浏览器输入的地址  viewName: 该地址跳转到的页面, 不加templates文件夹和.html
        registry.addViewController( urlPath: "/left").setViewName("common/left");
    }
```

批注 [j3]: 在 WebMvcAutoConfiguration 类中,初始化时会将自动配置中的路径映射以及用户自定义的扩展都会加入到 Spring 容器中

1.4.3.2 拦截器

SpringBoot 中声明拦截器与SpringMVC没什么区别,拦截器本身使用的仍然还是SpringMVC那一套。在 SpringBoot 中只需要通过 WebMvcConfigurer 的 addInterceptors 注册一下拦截器即可。

1.4.3.2.1 自定义登录拦截器

```
public class SessionIntercept implements HandlerInterceptor {


    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        Object userInfo = request.getSession().getAttribute("userInfo");
        if(userInfo != null){
            return true;
        }else{
            request.getRequestDispatcher("/login").forward(request,response );
```

```
            return false;
        }
    }
}
```

1.4.3.2.2 注册拦截器

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new SessionIntercept()).addPathPatterns("/**");
}
```
添加自定义拦截器　　配置拦截路径

在 SpringBoot 中拦截器的添加与视图控制映射的添加差不多。通过 addPathPatterns 可以添加当前拦截器需要拦截的请求路径。

1.4.3.2.3 排除路径

在实际的开发中，拦截路径是需要排除登录页面以及登录请求等入口页面及请求的。在 SpringBoot 中也可以直接使用 excludePathPatterns 方法来排除请求路径。

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new SessionIntercept()).addPathPatterns("/**")
            .excludePathPatterns("/login","/","/index.html","/user/login")
            .excludePathPatterns("/bootstrap/**","/jquery/**","/login/**");
}
```
登录相关
静态资源相关

批注 [j4]: 在 SpringBoot 2.0 之前,静态资源是不会被拦截器所拦截的,在 SpringBoot2.0 之后，静态资源映射 WebMvcConfigurationSupport.resourceHandlerMapping 会加入自定义拦截器,静态资源也会被拦截

1.4.3.3 格式转换器

```
@RequestMapping(value = "/user/login",method = RequestMethod.POST)
public String login(String username, String password, Date date, HttpSession session){
```

当从前端表单中传递的参数中携带有 Date 字段,而我们又希望能够使用 SpringMVC 提供的自动接收参数时,我们就需要使用自定义一个参数格式转换器.

1.4.3.3.1 自定义时间转换器

开发者可以自己定义一个实现 org.springframework.core.convert.converter.Converter 的子类.在实现时定义参数类型以及转换类型.在 Convert 中对需要转换的类型进行转换即可.

```
public class MyDateFormat implements Converter<String, Date> {
    private static final String dateFormat = "yyyy/MM/dd";
```

```
    @Override
    public Date convert(String source) {
        if (StringUtils.isEmpty(source)) {
            return null;
        }
        try {
            SimpleDateFormat formatter = new SimpleDateFormat(dateFormat);
            Date date = formatter.parse(source);
            return date;
        } catch (Exception e) {
            throw new RuntimeException(String.format("parser %s to Date fail", source));
        }
    }
}
```

1.4.3.3.2 配置时间转换器

自定义时间格式转换器需要添加到 SpringMVC 容器中才能生效。在 SpringBoot 在 WebMvcConfigurer 中提供了 addFormatters 来供用户提供添加自定义格式化转换器的方法。

```
@Override
public void addFormatters(FormatterRegistry registry) {
    registry.addConverter(new MyDateFormat());
}
```

1.4.4 自定义错误

在实际的项目开发中我们遇到的最多的错误就是 400 错误和 500 错误。
在 SpringBoot 中,MVC 错误由 ErrorMvcAutoConfiguration 来负责自动配置错误页面及错误信息返回，也就是我们看到的 4XX、5XX 这些错误页面信息的网页,且可以将默认的错误信息显示在自定义错误页面中。当我们通过其他接口访问测试工具，如 postman 这些访问时，所有的报错页面会变成错误 json。

```
Whitelabel Error Page

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

Tue Jul 24 18:59:37 CST 2018
There was an unexpected error (type=Not Found, status=404).
No message available
```

```
{
    "timestamp": "2018-07-24T09:50:13.300+0000",
    "status": 404,
    "error": "Not Found",
    "message": "No message available",
    "path": "/javasm/llll"
}
```

1.4.4.1 自定义错误页面

1.4.4.1.1 创建错误页面

当我们需要自定义 404 错误时，我们只需要在 templates 文件夹下创建 error 文件夹，并在 error 中创建一个 404.html 即可。同时我们也可以创建一个 4xx.html，当服务器返回的只要是 400 类型的错误就会自动跳转到 4xx.html 中。

1.4.4.1.1.1 原理解析

在 SpringBoot 中，自动配置机制会加载 ErrorMvcAutoConfiguration 类，在该类中由 basicErrorController 主要负责发生错误后的页面跳转，而 basicErrorController 中的 errorHtml() 和 error()方法就分别是返回 html 和 json 的两个控制器。

在 ErrorMvcAutoConfiguration 中的 basicErrorController() 方法会返回一个 BasicErrorController

```
@Bean
@ConditionalOnMissingBean(value = ErrorController.class, search = SearchStrategy.CURRENT)
public BasicErrorController basicErrorController(ErrorAttributes errorAttributes) {
    return new BasicErrorController(errorAttributes, this.serverProperties.getError(),
            this.errorViewResolvers);
}
```

#在 basicErrorController 上我们可以看到这是一个控制器,且映射路径是可以手工配置的

通过application配置文件可以对错误页面路径进行配置,默认是error

```
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {
```

#基本错误控制器中的两个方法的调用场景

```java
@RequestMapping(produces = "text/html")
public ModelAndView errorHtml(HttpServletRequest request,
        HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections.unmodifiableMap(getErrorAttributes(
            request, isIncludeStackTrace(request, MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    ModelAndView modelAndView = resolveErrorView(request, response, status, model);
    return (modelAndView != null ? modelAndView : new ModelAndView( viewName: "error", model));
}

@RequestMapping
@ResponseBody
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    Map<String, Object> body = getErrorAttributes(request,
            isIncludeStackTrace(request, MediaType.ALL));
    HttpStatus status = getStatus(request);
    return new ResponseEntity<>(body, status);
}
```

当请求头中的 Accept 为text/html时调用本方法

返回的是页面视图以及错误字段

没有什么produces时,当请求头中 Accept 的值非text/html时,则调用本方法
并返回json

返回的是对象且使用了@ResponseBody注解

#在错误网页方法中,依靠一个错误页面解析器方法来解析并返回视图模型

```java
@RequestMapping(produces = "text/html")
public ModelAndView errorHtml(HttpServletRequest request,
        HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections.unmodifiableMap(getErrorAttributes(
            request, isIncludeStackTrace(request, MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    ModelAndView modelAndView = resolveErrorView(request, response, status, model);
    return (modelAndView != null ? modelAndView : new ModelAndView( viewName: "error", model));
}
```

错误页面解析器方法

```java
*/
protected ModelAndView resolveErrorView(HttpServletRequest request,
        HttpServletResponse response, HttpStatus status, Map<String, Object> model) {
    for (ErrorViewResolver resolver : this.errorViewResolvers) {
        ModelAndView modelAndView = resolver.resolveErrorView(request, status, model);
        if (modelAndView != null) {
            return modelAndView;
        }
    }
    return null;
}
```

接口类型,该接口有一个默认的实现类 DefaultErrorViewResolver

#模型视图名称组装规则

```java
@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status,
        Map<String, Object> model) {
    ModelAndView modelAndView = resolve(String.valueOf(status), model);
    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
        modelAndView = resolve(SERIES_VIEWS.get(status.series()), model);
    }
    return modelAndView;
}

private ModelAndView resolve(String viewName, Map<String, Object> model) {
    String errorViewName = "error/" + viewName;
    TemplateAvailabilityProvider provider = this.templateAvailabilityProviders
            .getProvider(errorViewName, this.applicationContext);
    if (provider != null) {
        return new ModelAndView(errorViewName, model);
    }
    return resolveResource(errorViewName, model);
}
```

根据状态码创建视图模型

根据状态码和错误文件夹来组装视图名称

1.4.4.1.2 自定义异常处理器

在自定义页面中,我们可以使用 SpringBoot 的自动错误处理机制去方便的写自己的个性化错误页面,但是字段我们却不能个性化添加。如果我们需要个性化添加错误字段以及提示文本,则我们需要自定义一个异常处理器来解决。根据错误页面自动配置相同的原理，错误字段属性也在 ErrorMvcAutoConfiguration 中的 errorAttributes 方法中自动注册。而在该方法中定义了一个@ConditionalOnMissingBean 的注解，这个注解的意思是，当容器中没有找到自定义的类时，就使用默认的实现（DefaultErrorAttributes）来处理。
而在 basicErrorController 中对于页面错误处理都有相同的获取错误属性操作，且根据该方法的调用连最终就能找到 DefaultErrorAttributes 类的默认实现。

```java
@RequestMapping(produces = "text/html")
public ModelAndView errorHtml(HttpServletRequest request,
        HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model = Collections.unmodifiableMap(getErrorAttributes(
            request, isIncludeStackTrace(request, MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    ModelAndView modelAndView = resolveErrorView(request, response, status, model);
    return (modelAndView != null ? modelAndView : new ModelAndView( viewName: "error", model));
}

@RequestMapping
@ResponseBody
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    Map<String, Object> body = getErrorAttributes(request,
            isIncludeStackTrace(request, MediaType.ALL));
    HttpStatus status = getStatus(request);
    return new ResponseEntity<>(body, status);
}
```

拉取错误信息,并保存至map中

基于以上原理，我们只需要给 DefaultErrorAttributes 创建一个子类即可实现自定义属性的功能。

1.4.4.1.2.1 自定义异常属性方法

```java
@Component
public class MyExceptionHandler extends DefaultErrorAttributes {

    @Override
    public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {
        Map<String, Object> errorAttributes = super.getErrorAttributes(webRequest, includeStackTrace);
        errorAttributes.put("message", "系统繁忙");
        return errorAttributes;
    }
}
```

注册到Spring容器中

继承默认错误属性处理类

重用父类已有的属性字段

覆盖父类字段的字段或添加其他错误提示字段

1.4.5 跨域配置

```java
@Configuration
public class MyConfiguration {

 //springmvc 中  没有 FilterRegistrationBean 这个类
   @Bean
   public FilterRegistrationBean 这个类  corsFilter() {
      UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
      CorsConfiguration config = new CorsConfiguration();
      config.setAllowCredentials(true);
      config.addAllowedOrigin("http://domain1.com");
```

```
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        source.registerCorsConfiguration("/**", config);
        FilterRegistrationBean bean = new FilterRegistrationBean(new CorsFilter(source));
        bean.setOrder(0);
        return bean;
    }
}
```

1.5 日志

SpringBoot 默认集成有日志功能，使用的是 Apache 的 commons-logging 做日志的输出功能，且使用的 logback 的日志机制。除了默认的以外还提供一个 Log4J2 的 starter 供开发者选择，当我们引入相应的 starter 则默认日志应用就能被切换。【官方文档】

1.5.1.1 使用默认日志

SpringBoot 默认不配置的情况下可以直接使用 commons-logging 来进行日志打印,可以直接调用相应的方法进行日志打印功能。

```
Log log = LogFactory.getLog(UserController.class);
```

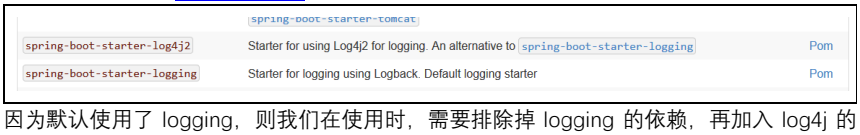
同时可以在 application 中对日志规则进行设定。

```
#log 配置信息
logging:
  #配置日志目录以及级别,root 表示根目录
  level: {root: debug}
  #配置日志输出目录的盘符路径
  path: D:/logs/
  #配置日志的名,相对路径(与 path 只能选其一)
  file: logs/test.log
```

使用默认的日志功能，只需要上述简单的配置即可完成对于的设置功能。

1.5.1.2 Log4j2 替换默认日志

在 SpringBoot 中，官方默认为我们预先实现了 log4j 和 logback，我们只需要引入相关的 starter 即可使用。【官方文档】



因为默认使用了 logging，则我们在使用时，需要排除掉 logging 的依赖，再加入 log4j 的

starter

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <artifactId>spring-boot-starter-logging</artifactId>
                <groupId>org.springframework.boot</groupId>
            </exclusion>
        </exclusions>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-log4j2</artifactId>
    </dependency>
```

最后在项目的 resources 文件夹下加入 log4j2.xml 即可使用.

理论上来说，加入 exclusion 就已经排除了 logging 的 jar 包，但是实际使用中，logging 偶尔会依然存在，这样会导致 jar 包冲突，配置文件使用失败。需要进入 maven 的视图，找到对应的 jar 包手动移除。

1.6 Mybatis

1.6.1.1 集成环境配置

在当前项目中加入 Mybatis 集成需要依赖的 Jar。

```
<!--jdbc 依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!-- mybatis 依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
<!-- mysql 驱动依赖-->
<dependency>
    <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<!--durid 依赖-->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
```

```
<!—build 标签下加入下面代码 解决 idea 下不自动编译 xml 文件的问题 -->
<resources>
        <resource>
                <directory>src/main/java</directory>
                <includes>
                        <include>**/*.xml</include>
                </includes>
        </resource>
        <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
        </resource>
</resources>
```

1.6.1.2 配置 Druid 连接池

Druid 是阿里巴巴开源的一款连接池工具，是目前 Java 语言中最好的数据库连接池工具。代码发布在 github 上。

1.6.1.2.1 配置简介

在 application 配置文件中加入 Druid 的配置信息。

```
spring:
  datasource:
      name: mysql_test
      type: com.alibaba.druid.pool.DruidDataSource
      #druid 相关配置
      druid:
        #监控统计拦截的 filters
        filters: stat
        driver-class-name: com.mysql.jdbc.Driver
        #基本属性
        url: jdbc:mysql://localhost:3306/javasm?useUnicode=true&characterEncoding=UTF-
```

```
8&allowMultiQueries=true&serverTimezone=GMT%2B8&useSSL=false
    username: root
    password: root
    #配置初始化大小/最小/最大
    initial-size: 1
    min-idle: 1
    max-active: 20
    #获取连接等待超时时间
    max-wait: 60000
    #间隔多久进行一次检测，检测需要关闭的空闲连接
    time-between-eviction-runs-millis: 60000
    #一个连接在池中最小生存的时间
    min-evictable-idle-time-millis: 300000
    validation-query: SELECT 'x'
    #空闲连接是否被回收
    test-while-idle: true
    #申请连接时是否检测有效性
    test-on-borrow: false
    #归还连接时是否检测有效性
    test-on-return: false
```

1.6.1.2.2 配置解析

在 SpringBoot 中默认采用 DataSourceAutoConfiguration 来配置数据源（DataSource）。官方默认只需要在 application 配置文件中使用简单的配置就可以连接数据库。
当加入了 Druid-starter 之后，它的 DruidDataSourceAutoConfigure 默认初始化了一个创建 DataSource 的方法。而这个 DataSource 默认就是读取当前 application 配置文件中的 spring.datasource.druid 的值，由此创建了基于 Druid 的数据源。

```
#SpringBoot 自带的 DataSourceAutoConfiguration
@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
        DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

    @Configuration
```

当没有开发者自定义的DataSource对象时,加载

读取配置文件中的JDBC相关配置

```java
@Configuration
@ConditionalOnClass(DruidDataSource.class)
@AutoConfigureBefore(DataSourceAutoConfiguration.class)
@EnableConfigurationProperties({DruidStatProperties.class, DataSourceProperties.class})
@Import({DruidSpringAopConfiguration.class,
    DruidStatViewServletConfiguration.class,
    DruidWebStatFilterConfiguration.class,
    DruidFilterConfiguration.class})
public class DruidDataSourceAutoConfigure {

    private static final Logger LOGGER = LoggerFactory.getLogger(DruidDataSourceAutoConfigure.class);

    @Bean(initMethod = "init")
    @ConditionalOnMissingBean
    public DataSource dataSource() {
        LOGGER.info("Init DruidDataSource");
        return new DruidDataSourceWrapper();
    }
}
```

加载本类之前先加载SB默认的DataSource配置

加载Druid的属性配置文件

初始化时加载,且在当前容器中没有开发者实现时

创建Druid数据源

1.6.1.3 测试 Druid 连接池对象是否被成功创建

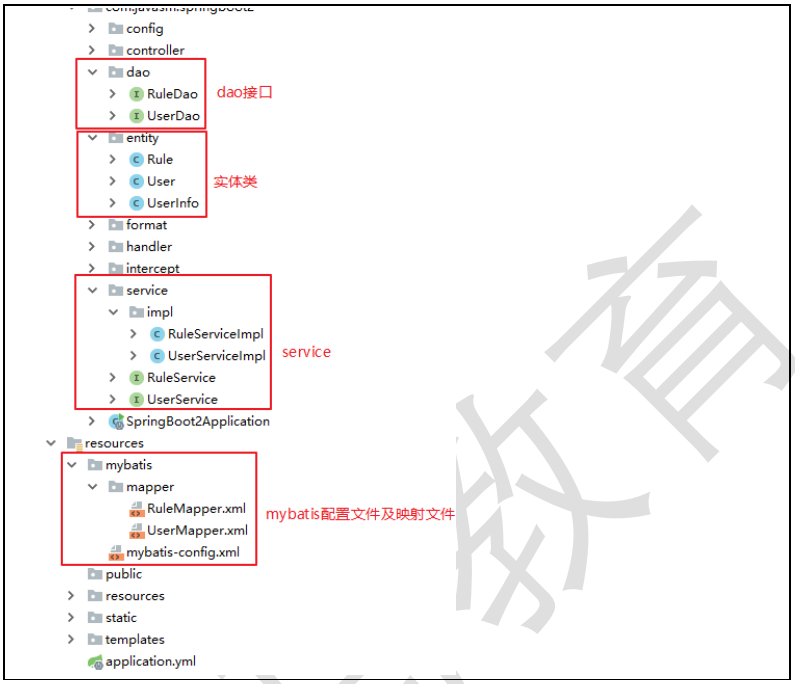
```java
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBoot2ApplicationTests {

    @Resource
    DataSource dataSource;

    @Test
    public void contextLoads() throws SQLException {
        System.out.println(dataSource.getClass());
        System.out.println(dataSource.getConnection());
    }

}
```

1.6.1.4 Mybatis 集成

在前面的集成环境配置中，我们已经加入了 mybatis 相关的 starter，也就是说，mybatis 已经集成好了。我们仅仅只需要一段简短的配置即可完成整合并开发。

1.6.1.4.1 目录结构文件

文件创建规则与 SSM 框架整合中使用一致。



1.6.1.4.2 Mybatis 核心文件配置及 Mapper 映射

在之前我们整合 SSM 时，我们会将 Mapper 映射文件以及 Mybatis 核心配置文件交给 Spring 来管理并加载。在 Mybatis 的 starter 中，按照 SpringBoot 的规范，同样也由一个 MybatisAutoConfiguration 来负责自动加载配置文件，并注册配置类。

```
mybatis:
  #配置Mybatis核心文件，方便后续对MyBatis进行扩展
  config-location: classpath:mybatis/mybatis-config.xml
  #配置本地Mapper.xml的映射路径
  mapper-locations: classpath:mybatis/mapper/*.xml
  #配置实体模型路径，完成自动类名转换
  type-aliases-package: com.javasm.springboot2.entity
```

1.6.1.4.3 将 Dao 层托管给 Spring

1.6.1.4.3.1 使用 Mapper 注解

```
@Mapper
public interface PersonDao {
```

在 dao 层，给所有的 dao 方法加上 Mapper 注解，将当前的类标注为 Mybatis 的 mapper 接口托管给 Spring

1.6.1.4.3.2 使用 MapperScan 注解

```
@SpringBootApplication
@MapperScan("com.javasm.springboot1.dao")
public class SpringBoot1Application {
```

使用 MapperScan 注解来扫描接口包路径

1.6.1.4.4 解决 java 路径下 xml 不编译的问题

```
<resource>
    <directory>src/main/java</directory>
    <includes>
        <include>**/*.xml</include>
    </includes>
</resource>
```

解决样式字体不生效问题

```
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
            <excludes>
                <exclude>**/*.woff</exclude>
                <exclude>**/*.woff2</exclude>
                <exclude>**/*.ttf</exclude>
            </excludes>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>false</filtering>
            <includes>
```

```
            <include>**/*.woff</include>
            <include>**/*.woff2</include>
            <include>**/*.ttf</include>
        </includes>
    </resource>
```

1.6.1.5 分页插件整合

1.6.1.5.1 加入插件依赖

在 POM 文件中加入分页插件的依赖。PageHelper 的作者也同样为 SpringBoot 创建了对应的 starter。加入 starter 就可以使用 Mybatis 的分页功能了。如果需要定制其他功能则可以按照 github 上的文档进行配置。【文档】

```
<!-- 分页插件 -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.2.5</version>
</dependency>
```

1.6.1.5.2 使用 PageHelper 插件

PageHelper 在项目的使用中基本上对已有的代码没有任何干扰。只需要在查询方法前调用一个静态方法即可完成分页查询。

```
//调用分页静态方法，传入当前页数和每页显示条数
PageHelper.startPage( pageNum: 1,  pageSize: 5);
List<User> all = userService.findAll();
//将查询结果传入PageInfo对象，会自动生成分页信息
PageInfo info = new PageInfo(all);
```

1.6.1.6 事务整合

在 SpringBoot 中使用事务也很简单，开发者只需要在启动类上加上 @EnableTransactionManagement 注解开启事务管理,并在需要做增删改的操作上加上 @Transactional 注解即可完成事务的配置及使用。

1.7 Redis 整合

在实际的开发中,我们会将用户经常访问请求的数据放到 redis 服务中,用来提升服务器响应速度及性能。SpringBoot 作为 Spring 全家桶的代表性框架,也同样为我们默认集成了 Redis,按照之前我们提到的,Spring 也同样为 Redis 做了一个 Starter。而这个 starter 基于 Spring-Data 框架。

SpringData 框架是 Spring 出的一款数据访问框架,它包括许多子项目,基本上涵盖了目前企业开发所需要使用的主流数据存储应用,包括搜索引擎。这个项目同时也包含了 Spring 想一统 JavaEE 企业开发的野心。但是由于它提供的 ORM 框架-JPA 对于复杂 SQL 以及后期维护等,并不是很友好,所以目前现阶段来讲,无法去替代 MyBatis 的地位。

1.7.1 环境准备

与其他框架整合相同,需要继承 redis 也仅仅只需要引入 redis 的 starter,然后在 application 配置文件中简单配置下即可使用

spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Lettuce client	Pom

1.7.1.1 引入依赖

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
        <exclusions>
                <exclusion>
                        <groupId>io.lettuce</groupId>
                        <artifactId>lettuce-core</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
</dependency>
```

```


<!-- Fastjson -->
<dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.2.7</version>
</dependency>
```

1.7.1.2 修改 application 配置文件

如果当前 redis 实例连接的是本地且是默认端口 6379，则无需配置。具体参数可以参考 RedisAutoConfiguration 类中引入的 RedisProperties 类

```yaml
#redis相关配置
redis:
  host: localhost
  port: 6379
  timeout:
    seconds: 5000
```

1.7.1.3 基本使用示例

由于 SpringBoot 已自动帮我们创建好了 Redis 的连接以及操作模板，开发者只需要引入模板类即可对 Redis 进行操作。

```java
@Resource
StringRedisTemplate stringRedisTemplate;

@RequestMapping("/test")
public Object findAll(){
    stringRedisTemplate.opsForValue().set("haha", "张三");
    return "ok";
}
```

1.7.1.4 使对象自动序列化成 JSON

在实际的开发中，我们会经常讲用户信息或其他对象信息以 json 的方式保存至 redis 中。而每次我们都需要在代码中将对象给重新序列化成 json，这样就会很麻烦。实际上在 RedisAutoConfiguration 中，我们可以看到以下两个方法，他们分别向容器中注册了 redisTemplate 和 stringRedisTemplate 两个方法，并向容器中注册了两个对象。而 StringRedisTemplate 就是我们上述使用的字符串操作对象。
RedisTemplate 就是可以直接保存对象的模板。而在 RedisTemplate 类中我们可以看到如下信息：

```
* @see StringRedisTemplate
*/
public class RedisTemplate<K, V> extends RedisAccessor implements RedisOperations<K, V>, BeanClassLoaderAware {

    private boolean enableTransactionSupport = false;
    private boolean exposeConnection = false;
    private boolean initialized = false;
    private boolean enableDefaultSerializer = true;
    private @Nullable RedisSerializer<?> defaultSerializer;
    private @Nullable ClassLoader classLoader;

    /rawtypes/ private @Nullable RedisSerializer keySerializer = null;
    /rawtypes/ private @Nullable RedisSerializer valueSerializer = null;
    /rawtypes/ private @Nullable RedisSerializer hashKeySerializer = null;
    /rawtypes/ private @Nullable RedisSerializer hashValueSerializer = null;
    private RedisSerializer<String> stringSerializer = new StringRedisSerializer();
```

该类分别由两个成员对象，key 序列化和值序列化对象。基于这样的情况，我们只需要在 Spring 初始化的时候，自己创建一个 RedisTemplate 对象并给它设置自定义的值序列化对象。

```
@Bean    在任何配置类中加入自定义bean
public RedisTemplate<String, Object> redisTemplate(
        RedisConnectionFactory redisConnectionFactory) throws UnknownHostException {
    RedisTemplate<String, Object> template = new RedisTemplate<>();
    template.setConnectionFactory(redisConnectionFactory);
    template.setKeySerializer(new StringRedisSerializer());    名称使用字符串序列化对象
    template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
    return template;                                    值可以直接使用jackson队列化对象
}
```

当我们重新定义了 RedisTemplate 之后，我们就可以直接进行对象保存了。

1.7.2 注解方式使用

1.7.2.1 修改启动类

```
@SpringBootApplication
@MapperScan("com.javasm.bootdemo.*.dao")//扫描dao目录下的mapper
@EnableTransactionManagement//事务开启
@EnableCaching //启动缓存注解
public class BootdemoApplication {
```

加入@EnableCaching

1.7.2.2 在需要使用缓存的位置使用注解

注解可以加在接口的方法上，也可以加在接口实现类的方法上，一个地方加就可以

```java
public interface UserService {

    @Cacheable(cacheNames = "users", key = "#id", unless = "#result==null", condition = "#id>0")
    User selectUserById(Integer id);
}
```

```java
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    UserMapper userDao;

    // cacheNames是缓存中的key
    // key代表着users:后的key部分 #id表示参数名称 如果参数是对象Role role 可以写成#role.id
    // unless方法被调用后被执行，结果为null的数据不存入缓存
    // condition="#name.length() < 32" 符合条件的数据才会加入缓存
    @Cacheable(cacheNames = "users", key = "#id", unless = "#result==null", condition = "#id>0")
    @Override
    public User selectUserById(Integer id) {
        // User实体类要序列化，否则报错
        return userDao.selectByPrimaryKey(id);
    }

}
```

注解	描述
@Cacheable	表明在 Spring 调用之前，首先应该在缓存中查找方法的返回值，如果这个值能够找到，就会返回缓存的值，否则这个方法会被调用，返回值会放到缓存中
@CachePut	表明 Spring 应该将该方法返回值放到缓存中，在方法调用前不会检查缓存，方法始终会被调用
@CacheEvict	表明 Spring 应该在缓存中清除一个或多个条目
@Caching	分组注解，能够同时应用多个其他的缓存注解
@CacheConfig	可以在类层级配置一些共有的缓存配置

1.8 定时任务

Spring Schedule 是 Spring 自带的任务框架，是一个简化版本的 Quartz

1.8.1 环境准备

1.8.1.1 引入依赖

```
<!-- Quartz 定时任务调度 -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-quartz</artifactId>
</dependency>
```

1.8.1.2 修改启动类

```
@SpringBootApplication
@MapperScan("com.javasm.bootdemo.*.dao")//扫描dao目录下的mapper
@EnableTransactionManagement//事务开启
@EnableCaching //启动缓存注解
@EnableScheduling//开启定时任务
public class BootdemoApplication {
```

1.8.1.3 创建任务类

```
@Component//当前类注入到Spring容器中
public class TimmerLogJob {
    //API参考前面quartz课件
    @Scheduled(cron = "0/5 * * * * ?")//每5秒执行一次
    public void reportCurrentTimeCron() throws InterruptedException {
        System.out.println("任务执行！");
    }
}
```

1.9 Solr 整合

1.10Idea 开发时的小技巧

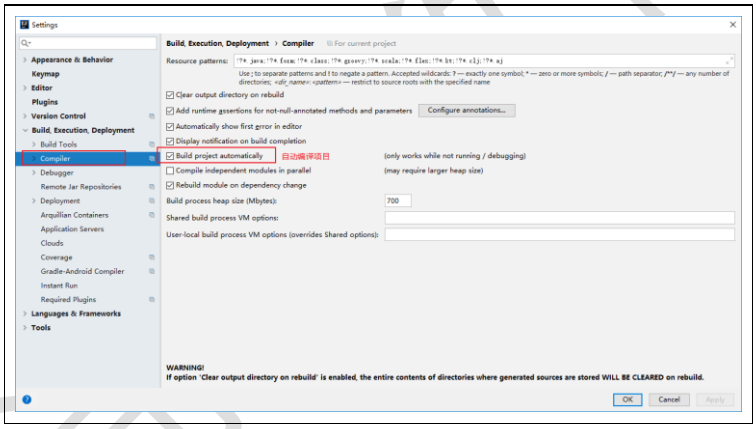
1.10.1 自动编译

Idea 默认是自动保存的，但是不会自动编译。在开发 SpingBoot 时，我们可以添加一个开发工具依赖简单配置一下即可完成修改文件实时生效。

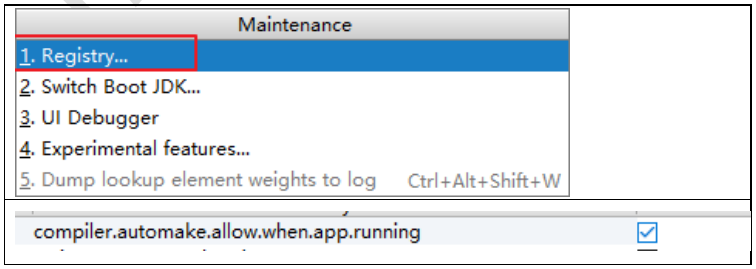
1. 添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

2. 配置 idea



3. 使用 ctrl+alt+shitf+/ 换出 idea 的维护设置



注：改文件名，添加新的文件，仍然还是需要自己手动重启。
开发工具也有各种各样的bug存在,当出现各种奇幻的情况时,可以选择手动重启项目、开发工具、操作系统。

1.11兼容 jsp

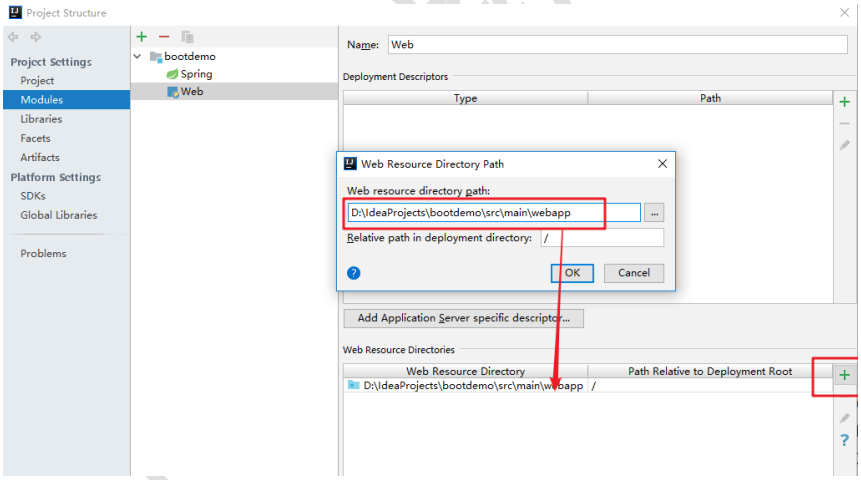
官方不推荐使用 jsp 作为前端页面，这里介绍一下如何兼容 jsp
想要访问 jsp 需要外部 tomcat 的支持，tomcat 版本必须是 8 以上
发布的 jar 包改成 war

1.11.1 视图解析

不使用默认的视图解析路径，手动修改视图解析
修改 yml 配置文件，根据项目实际路径修改

```yaml
spring:
  mvc:
    view:
      prefix: /WEB-INF/jsp/
      suffix: .jsp
```

Idea 创建的 springboot 项目默认不能创建 jsp 文件，项目设置中配置 web'目录



1.11.2 引入依赖

```
<!--用于编译 jsp-->
<dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
        <scope>provided</scope>
</dependency>
```

```
<!--jsp 页面使用 jstl 标签  高版本 Tomcat/Myeclipse 以下可以省略-->
<dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
</dependency>
<!-- 避免外部 Tomcat 和内嵌 Tomcat 冲突-->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
</dependency>
```

```
<!--为了兼容 jsp 的使用 注释掉 thymeleaf
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>-->
```

1.11.3 修改启动类

```java
//相当于@Controller @EnableAutoConfiguration @ComponentScan
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    //集成Tomcat必须继承SpringBootServletInitializer
    //重写configure方法
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication application = new SpringApplication(Application.class);
        //设置关闭spring文字输出
        //application.setBannerMode(Mode.OFF);
        application.run(args);
        //SpringApplication.run(Application.class, args);
    }
}
```

1.11.4 发布到 tomcat

启动 tomcat

1.11.5 一些错误

- 低版本Myeclipse启动找不到配置文件
  - 修改java build path--> resource --> include 删除inclode规则即可
- 低版本Myeclipse下Tomcat启动成功，但是没有SpringBoot日志和页面
  - 检查Deployment并且确认发布的是war包

- Tomcat启动报错ClassNotFoundException
  - 确认pom.xml文件正确
  - 报错的包是否存在
  - 也可能是maven下载jar包时，网络不稳，造成jar包下载失败
- @Controller注入的类如果找不到
  - 启动类是否为@SpringBootApplication
  - 如果是其他注解别忘了加上@ComponentScan