

SpringBoot

SpringBoot	1
1. 环境变量	9
1.1. JAVA_HOME C:\Program Files\Java\jdk1.8.0_162\bin JRE_HOME C:\Program Files\Java\jdk1.8.0_162\jre CATALINA_HOME D:\Java\Tomcat\apache-tomcat-8.5.27 Path C:\Program Files\Java\jdk1.8.0_162\bin D:\Java\Tomcat\apache-tomcat-8.5.27\bin.....	9
2. HelloWorld	9
2.4. 选择使用SpringBoot APP启动 public static void main(String[] args) { SpringApplication.run(Application.class, args); } 它会启动SpringBoot内嵌的Tomcat,默认端口号为8080	11
2.5. main方法.....	12
2.5.1. 应用程序的最后部分是main方法，这是一个标准的方法，它遵循Java 对于一个应用程序入口点的约定。我们的main方法通过调用run，将业务委托给了Spring Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat web服务器。我们需要将Example.class作为参数传递给run方法，以此告诉SpringApplication谁是主要的Spring组件，并传递args数组以暴露所有的命令行参数。	12
3. 核心配置文件	12
3.1. SpringBoot默认支持properties和YAML两种格式的配置文件。前者格式简单，但是只支持键值对。如果需要表达列表，最好使用YAML格式。SpringBoot支持自动加载约定名称的配置文件，例如application.yml。如果是自定义名称的配置文件，就要另找方法了。可惜的是，不像前者有@PropertySource这样方便的加载方式，后者的加载必须借助编码逻辑来实现。	12
3.2. application.properties.....	12
3.2.1. #自定义配置 boot.name=Raynor宗 boot.age=23 #server 编码 spring.http.encoding.charset=utf-8 spring.http.encoding.enabled=true spring.http.encoding.force=true	12
3.3. appliaction.yml.....	12
3.3.1. boot: age: 23 name: Raynor宗 server: http: encoding: charset: utf-8 enabled: true force: true # spring mvc spring: mvc: view: prefix: / suffix: .jsp.....	13
4. 配置项的使用	13
4.1. 直接从applicaiton.properties取值.....	13

4.1.1.	@Value("\${boot.name}") private String name;	13
4.2.	applicaiton.properties属性绑定JavaBean.....	13
4.2.1.	<p>只要是加载到Spring容器中的配置项都可以直接使用@Value("\${key}")的方式来引用，一般将其配置在字段顶部，表示将配置项的值赋值给该字段。</p> <p>13</p>	
4.2.2.	@Component	
	<p>//把普通pojo实例化到spring容器中，相当于配置文件中的 <bean id="" class=""/> @Configuration //默认去找application.properties中的 属性</p> <p>//@PropertySource("classpath:user.properties")</p> <p>@ConfigurationProperties(prefix="user") public class User{ private String name; private String sex; private String age; //省略set get 方法 }</p>	
5.	SpringMVC	14
5.1.	@ResponseBody //返回Json数据类型,并不会返回视图	
	@RequestMapping("/boot/hello") public @ResponseBody String hello() { return "hello world"; }	14
5.1.1.	<p>@ResponseBody 注解表示该方法的返回的结果直接写入 HTTP 响应正文（ResponseBody）中，一般在异步获取数据时使用，通常是在使用 @RequestMapping 后，返回值通常解析为跳转路径，加上 @ResponseBody 后返回结果不会被解析为跳转路径，而是直接写入HTTP 响应正文中。</p>	
5.2.	@RestController // = @Controller + @ResponseBody	14
5.3.	@GetMapping("/boot/getUser1") //只支持get请求	
	//相当于上面的requestMapping 加上 method=RequestMethod.GET	
5.4.	@PostMapping("/boot/getUser2") //只支持post请求	
	//相当于上面的requestMapping 加上 method=RequestMethod.POST	
	//直接在地址栏输入地址相当于get请求,这里会报405错误,不支持该请求	
5.5.	@PathVariable =====	
	@RequestMapping("/boot/restful/{id}") public Object user(@PathVariable("id") Integer id) { return ""; }	15
5.6.	@RequestParam	
	<p>http://localhost:8080/springmvc/hello/101?param1=10&param2=20 public String getDetails(@RequestParam(value="param1", required=true) String param1, @RequestParam(value="param2", required=false) String param2){ ... }</p>	
6.	JSP	15
6.1.	所需依赖.....	15
6.1.1.	<p><!-- 引入spring-boot内嵌的Tomcat对J s p的解析包 --> <dependency></p> <p><groupId>org.apache.tomcat.embed</groupId> <artifactId>tomcat-embed-jasper</artifactId> </dependency> <!-- servlet.API --> <dependency></p> <p><groupId>javax.servlet</groupId> <artifactId>javax.servlet-api</artifactId></p> <p></dependency> <!-- servlet.jsp-API --> <dependency></p> <p><groupId>javax.servlet.jsp</groupId> <artifactId>javax.servlet.jsp-api</artifactId></p>	

<version>2.3.1</version> <scope>provided</scope> </dependency> <!-- JSTL依赖 --> <dependency> <groupId>javax.servlet</groupId> <artifactId>jstl</artifactId> </dependency>	15
6.2. 在src/main下新建webapp目录,并在该目录下新建jsp.....	16
6.3. SpringMVC配置(在application.yml中配置)	16
6.3.1. # spring mvc spring: mvc: view: prefix: / suffix: .jsp	16
6.4. demo	16
6.4.1. @Controller public class JSPController { @RequestMapping("/boot/index") public String index(Model model) { model.addAttribute("msg", "spring boot jsp"); return "index"; } }	16
7. Mybatis	17
7.1. 1.依赖.....	17
7.1.1. <!-- 集成Mybatis --> <!-- mybatis-spring-boot-starter --> <dependency> <groupId>org.mybatis.spring.boot</groupId> <artifactId>mybatis-spring-boot-starter</artifactId> <version>1.3.1</version> </dependency> <!-- mysql-connector-java --> <dependency> <groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> </dependency>	17
7.2. 2.在application.properties中配置mybatis的Mapper.xml文件所在位置 mybatis.mapper-locations=classpath:com/demo/mapper/*.xml	17
7.3. 3.配置数据库数据源.....	17
7.3.1. spring.datasource.username=root spring.datasource.password=123456 spring.datasource.driver-class-name=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-8&useSSL=false	17
7.4. 4.在mybatis的Mapper接口中添加@Mapper注解或者在运行的主类上添加@MapperScan("com.dmeo.mapper")注解扫描包	18
8. SSM	18
8.1. Mapper(DAO).....	18
8.1.1. @Mapper // 必要的注解,相当于配置文件中spring 中配置bean public interface UserMapper { List<User> selectAll(); User selectById(Integer id); }	18
8.1.2. <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" > <mapper namespace="com.demo.mapper.UserMapper"> <select id="selectAll" resultType="com.demo.model.User"> select t_id as id,t_username as username,t_password as password from t_user </select> <!-- 根据id查询数据表中的一条记录,并封装User对象 --> <select id="selectById" resultType="com.demo.model.User"> select t_id as id,t_username as username,t_password as password from t_user where t_id=#{id}; </select>	18
8.2. Service	18
8.2.1. public interface UserService { public List<User> selectAll(); public User selectById(Integer id); }	18

8.2.2.	@Service public class UserServiceImpl implements UserService{ @Autowired private UserMapper userMapper; public List<User> selectAll() { return userMapper.selectAll(); } public User selectById(Integer id) { return userMapper.selectById(id); } }.....	19
8.3.	Controller	19
8.3.1.	@RestController public class MybatisController { @Autowired private UserService userService; @GetMapping("/boot/getUsers") public Object users() { return userService.selectAll(); } }	19
9.	事务支持	19
9.1.	1.在入口类中使用注解@EnableTransactionManagement开启事务支持	19
9.2.	2.在访问数据库的service方法上增加注解@Transactional	19
10.	热部署插件	19
10.1.	<!-- 热部署插件spring-boot-devtools --> <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot- devtools</artifactId> <optional>true</optional> </dependency>.....	20
11.	RESTFul API	20
11.1.	1.@PathVariable.....	20
11.1.1.	// http://localhost:8080/boot/restful/2/Raynor @RequestMapping("/boot/restful/{id}/{name}") public Object user1(@PathVariable("id") Integer id, @PathVariable("name") String name) { User user = new User(); user.setId(id); user.setUsername(name); return user; }.....	20
11.2.	2.增加POST方法	20
11.2.1.	@PostMapping.....	20
11.2.2.	接受和处理post请求.....	20
11.3.	3.删除DELETE方法	20
11.3.1.	@DeleteMapping	20
11.3.2.	接收delete方式请求,可以用GetMapping替代	20
11.4.	4.修改PUT方法	20
11.4.1.	@PutMapping 接收put方式请求,可以用PostMapping替代	21
11.5.	5.查询GET方法.....	21
11.5.1.	@GetMapping 接收get方式请求.....	21
12.	Redis 集成	21
12.1.	1.依赖	21
12.1.1.	<dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-redis</artifactId> </dependency>	21
12.2.	2.redis 配置	21
12.2.1.	spring.redis.host=localhost spring.redis.port=6379 spring.redis.password= //默认密码为空 (redis)	21

12.3.	3.在使用的类中注入// 注入一个spring boot 配置好的 RedisTemplate (Service Impl中) @Autowired private RedisTemplate<Object, Object> redisTemplate; //这里的泛型只能是Object,Object 或者String,String	21
12.4.	4.demo.....	21
12.4.1.	@Override public List<User> selectAll() { // 使用Redis缓存 //字符串序列化器 RedisSerializer redisSerializer = new StringRedisSerializer(); redisTemplate.setKeySerializer(redisSerializer); //对key进行String序列化,增强可读性 // 查询缓存 @SuppressWarnings("unchecked") List<User> list = (List<User>) redisTemplate.opsForValue().get("allUsers"); //User要实现序列化接口 //因为存在redis中的数据都是序列化后的 if (list == null) { // 查询为空 再查询一次 list = userMapper.selectAll(); // 放到redis 缓存中 redisTemplate.opsForValue().set("allUsers", list); } return list; }	21
12.5.	5.在高并发的情况下,出现缓存穿透的问题	22
12.5.1.	假设有10000个人同时进到 上述方法中,都从redis中查询一次均为空,此时所有人都要进入对DB进行查询,DB的压力过大. 解决1:10000个人中,先让一个人查询数据库完成后放到redis中,其余人再从redis中取数据 A.在 方法前面加入 synchronized 关键字,10000个人进到这个方法里面只有一个人有锁 B.对if增加锁	22
13.	Interceptor	23
13.1.	1.按照SpringMVC写一个拦截器类	23
13.1.1.	public class LoginInterceptor implements HandlerInterceptor{ @Override public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception { System.out.println("进入Login拦截器....."); //返回为真则可以通过,返回为false response.sendRedirect("http://localhost:8080/index.jsp"); return false; } }	23
13.2.	2.编写一个配置类继承WebMvcConfigurerAdapter类 SpringBoot 2.0 因为Spring5弃用了上述类, -改为继承WebMvcConfigurationSupport -implements WebMvcConfigurer	24
13.2.1.	public void addInterceptors(InterceptorRegistry registry) { String [] PathPatterns = { // "/boot/**" }; String [] exPathPatterns = { "/boot/hello" }; registry.addInterceptor(new LoginInterceptor()) .addPathPatterns(PathPatterns).excludePathPatterns(exPathPatterns); }	24
13.3.	3.为该配置类添加@Configuration注解,标注此类为一个配置类,让SpringBo ot扫描到	24
13.4.	4.覆盖其中的方法,并添加已经写好的拦截器	24
14.	Servlet	25

14.1.	方式一	25
14.1.1.	1.写一个Servlet	25
14.1.2.	2.再main方法前加扫描注解	25
14.2.	方式二	25
14.2.1.	1.写一个Servlet	25
14.2.2.	2.写一个配置类.....	26
15.	Filter	26
15.1.	方式一	26
15.1.1.	1.写一个Filter	26
15.1.2.	2.再main方法前加扫描注解	26
15.2.	方式二	27
15.2.1.	写一个Filter	27
15.2.2.	写一个配置类.....	27
16.	项目编码配置	27
16.1.	1.方式一	27
16.1.1.	使用字符编码过滤器(Filter)//等价于 @Bean public FilterRegistrationBean filterRegistration() { FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean(); CharacterEncodingFilter encodingFilter =new CharacterEncodingFilter(); encodingFilter.setForceEncoding(true); encodingFilter.setEncoding("UTF-8"); filterRegistrationBean.setFilter(encodingFilter); filterRegistrationBean.addUrlPatterns("//*"); return filterRegistrationBean; }	
	注意:只有当spring.http.encoding.enabled=false 配置成false才会生效	27
16.1.2.	然后在主类上需要扫描此过滤器,扫描包.....	28
16.2.	2.方式二	28
16.2.1.	在核心配置文件中配置:(SpringBoot1.4.2之后才有的) spring.http.encoding.charset=utf-8 spring.http.encoding.enable=false spring.http.encoding.force=true	28
17.	部署	28
17.1.	WAR	28
17.1.1.	1.程序入口类需要继承 SpringBootServletInitializer类	28
17.1.2.	2.程序入口类覆盖如下方法.....	28
17.1.3.	3.配置插件.....	28
17.1.4.	4.将POM中的<packaging>jar</packaging>改为war	29
17.1.5.	5.在maven install中在本地仓库中安装成一个war包,然后将war部署到tomcat中运行	29
17.2.	JAR	29
17.2.1.	配置插件.....	29
17.2.2.	install	29

17.2.3.	java -jar spring-boot-demo.jar	29
18.	执行器 (Actuator)	29
18.1.	<p>在生产环境中,需要定时或者定期监控服务的可用性,springboot的actuator功能提供了很多所需要的接口</p> <p>是springboot提供的对应用系统的自省和监控的功能,可以对应用系统进行配置查看,健康查看,相关统计等功能</p>	
18.2.	1.依赖	29
18.2.1.	<code><dependency> <groupId>org.springframework.boot</groupId></code> <code> <artifactId>spring-boot-starter-actuator</artifactId> </dependency></code>	30
18.3.	<p>2.application.properties中指定监控的HTTP端口:</p> <p>如果不指定,则使用和server相同的端口 #服务运行的端口 server.port=8080</p> <p>#actuator端口配置 #actuator监控的上下文,不配置默认使用tomcat的上下文</p> <p>#management.server.servlet.context-path=/项目名 #不配置,默认使用tomcat的端口</p> <p>management.server.port=8080</p> <p>#默认只开启了health和info,设置为*,则包含所有web入口端点</p> <p>management.endpoints.web.exposure.include=*</p>	
18.4.	3.访问http://localhost:8080/actuator/health	30
19.	日志	30
19.1.	日志接口层(抽象层).....	30
19.1.1.	JCL/SLF4j/jboss-logging	30
19.2.	日志实现层	30
19.2.1.	Log4j/Log4j2/Logback/JUL	30
19.2.2.	调用的时候,调用接口层的方法;配置的时候,需要配置实现层.....	31
19.3.	统一日志记录	31
19.3.1.	question: a(slf4j+logback) :spring(common-loggng),hibernate(jboss-loggng) 31	
19.3.2.	使用原框架的替换jar包:jcl-over-slf4j.jar	31
19.3.3.	SpringBoot统一:(slf4j).....	31
19.4.	SpringBoot日志关系	32
19.4.2.	1.默认使用slf4j/logback 2.已经替换过 jul/log4j	
	3.如果要引入其它的日志框架,一定要移除默认的日志依赖	32
19.5.	SpringBoot使用slf4j-logback.....	32
19.5.1.	<code>Logger logger=LoggerFactory.getLogger(getClass()); @Test public void contextLoads() { //日志的级别;</code> <code>//由低到高 trace<debug<info<warn<error</code> <code>//可以调整输出的日志级别; 日志就只会在这个级别以以后的高级别生效</code> <code>logger.trace("这是trace日志..."); logger.debug("这是debug日志...");</code> <code>//SpringBoot默认给我们使用的是info级别的, 没有指定级别的就用SpringBoot</code>	

默认规定的级别；root级别	logger.info("这是info日志...");	
	logger.warn("这是warn日志..."); logger.error("这是error日志..."); }	33
19.5.2.	配置方式	33
19.6.	切换日志框架	34
19.6.1.	slf4j+log4j(默认的logback性能优于log4j)	34
19.6.2.	slf4j+log4j2	35
20.	静态资源	37
20.1.	webjar	37
20.1.1.	所有 /webjars/**，都去 classpath:/META-INF/resources/webjars/找资源；	37
20.1.2.	选用webjar来管理前台资源文件，因为通过手工进行管理，容易导致文件混乱、版本不一致等问题。而WebJars是将这些通用的Web前端资源打包成Java的Jar包，然后借助Maven工具对其管理，保证这些web资源版本唯一性，升级也比较容易。 http://www.webjars.org/	37
20.1.4.	<link href="/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css" rel="stylesheet"> <script src="/webjars/jquery/3.1.1/jquery.min.js"></script>	38
20.2.	"/**" 访问当前项目的任何资源，都去（静态资源的文件夹）找映射	38
20.2.1.	"classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/" "/"：当前项目的根路径 localhost:8080/abc ==> 去静态资源文件夹里面找abc	38
20.3.	欢迎页	38
20.3.1.	静态资源文件夹下的所有index.html页面；被"/**"映射； localhost:8080/ 找index页面	38
20.4.	所有的 **/favicon.ico 都是在静态资源文件下找	38
20.5.	用户css/js	38
20.5.2.	<link href="js/signin.css" rel="stylesheet">	38
21.	集成Dubbo	38
21.1.	1.依赖	38
21.1.1.	<dependency> <groupId>com.alibaba.boot</groupId> <artifactId>dubbo-spring-boot-starter</artifactId> <version>0.2.0</version> </dependency>	39
21.2.	2.集成	39
21.2.1.	二级maven结构	39
21.2.2.	开发Dubbo服务接口	40
21.2.3.	开发Dubbo服务提供者	40
21.2.4.	开发Dubbo服务消费者	43
21.2.5.	开启zookeeper服务,然后分别运行dubbo-provider和dubbo-consumer	45

1. 环境变量

1.1. JAVA_HOME

C:\Program Files\Java\jdk1.8.0_162\bin

JRE_HOME

C:\Program Files\Java\jdk1.8.0_162\jre

CATALINA_HOME

D:\Java\Tomcat\apache-tomcat-8.5.27

Path

C:\Program Files\Java\jdk1.8.0_162\bin

D:\Java\Tomcat\apache-tomcat-8.5.27\bin


2. HelloWorld

2.1.

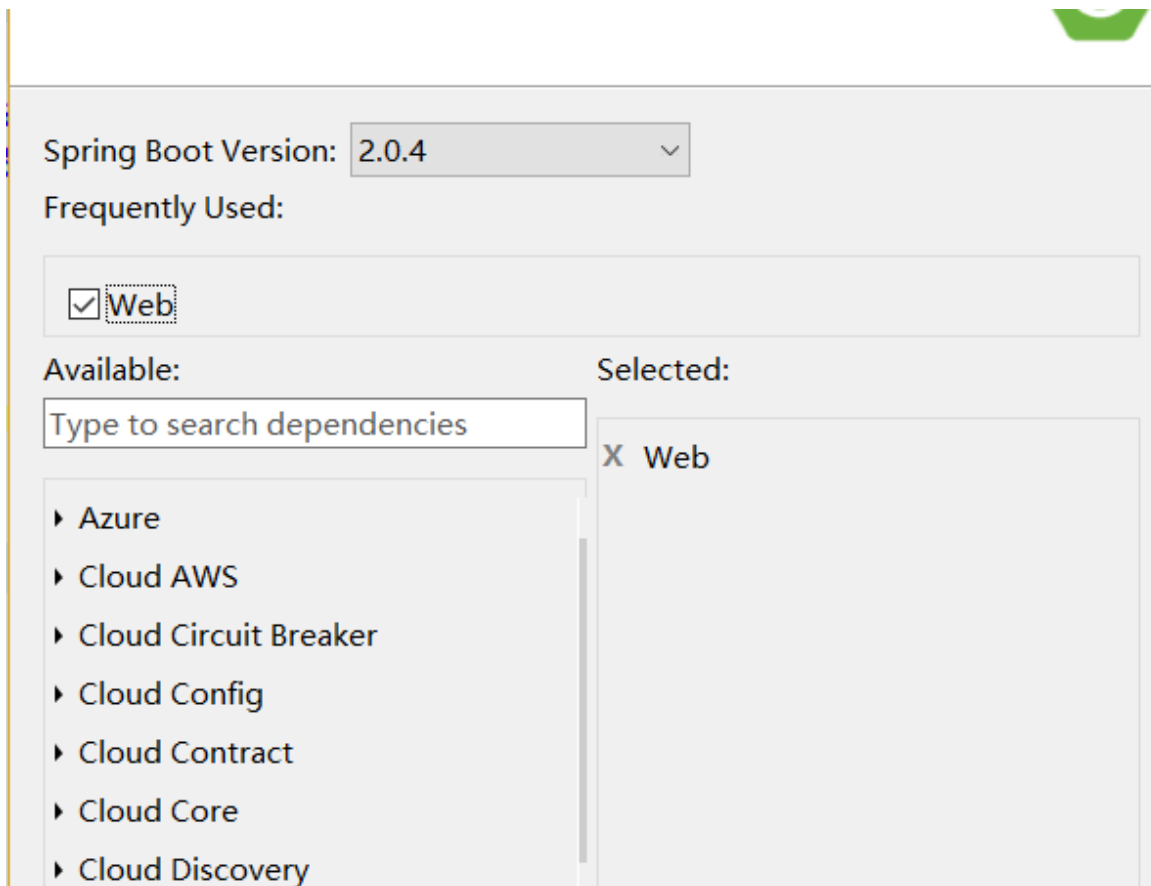
New Spring Starter Project



Service URL	<input type="text" value="https://start.spring.io"/>		
Name	<input type="text" value="02-springboot"/>		
<input checked="" type="checkbox"/> Use default location			
Location	<input type="text" value="D:\Mist\SpringBoot\02-springboot"/>	<input type="button" value="Browse"/>	
Type:	<input type="text" value="Maven"/>	Packaging:	<input type="text" value="Jar"/>
Java Version:	<input type="text" value="8"/>	Language:	<input type="text" value="Java"/>
Group	<input type="text" value="com.demo"/>		
Artifact	<input type="text" value="02-springboot"/>		
Version	<input type="text" value="1.0.0"/>		
Description	<input type="text" value="Demo project for Spring Boot"/>		
Package	<input type="text" value="com.demo"/>		
Working sets			
<input type="checkbox"/> Add project to working sets	<input type="button" value="New..."/>		
Working sets:	<input type="text"/>	<input type="button" value="Select..."/>	



2.2.



2.3.



2.4. 选择使用SpringBoot APP启动

```

public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}

```

它会启动SpringBoot内嵌的Tomcat,默认端口号为8080

2.5. main方法

2.5.1. 应用程序的最后部分是main方法，这是一个标准的方法，它遵循Java对于一个应用程序入口点的约定。我们的main方法通过调用run，将业务委托给了Spring

Boot的SpringApplication类。SpringApplication将引导我们的应用，启动Spring，相应地启动被自动配置的Tomcat

web服务器。我们需要将Example.class作为参数传递给run方法，以此告诉SpringApplication谁是主要的Spring组件，并传递args数组以暴露所有的命令行参数。

3. 核心配置文件

3.1. SpringBoot默认支持properties和YAML两种格式的配置文件。前者格式简单，但是只支持键值对。如果需要表达列表，最好使用YAML格式。SpringBoot支持自动加载约定名称的配置文件，例如application.yml。如果是自定义名称的配置文件，就要另找方法了。可惜的是，不像前者有@PropertySource这样方便的加载方式，后者的加载必须借助编码逻辑来实现。

3.2. application.properties

3.2.1. #自定义配置

```
boot.name=Raynor宗
boot.age=23
#server 编码
spring.http.encoding.charset=utf-8
spring.http.encoding.enabled=true
spring.http.encoding.force=true
```

3.3. application.yml

3.3.1. boot:

age: 23

name: Raynor宗

server:

http:

encoding:

charset: utf-8

enabled: true

force: true

spring mvc

spring:

mvc:

view:

prefix: /

suffix: .jsp

4. 配置项的使用

4.1. 直接从applicaiton.properties取值

4.1.1. @Value("\${boot.name}")

```
private String name;
```

4.2. applicaiton.properties属性绑定JavaBean

4.2.1. 只要是加载到Spring容器中的配置项都可以直接使用@Value("\${key}")的方式来引用，一般将其配置在字段顶部，表示将配置项的值赋值给该字段。

4.2.2. @Component //把普通pojo实例化到spring容器中，相当于配置文件中的
//<bean id="" class=""/>)

```
@Configuration
//默认去找application.properties中的 属性
//@PropertySource("classpath:user.properties")
```

```
@ConfigurationProperties(prefix="user")
public class User{
    private String name;
    private String sex;
    private String age;
    //省略set get 方法
}
```

5. SpringMVC

5.1. @ResponseBody

```
//返回Json数据类型,并不会返回视图
@RequestMapping("/boot/hello")
public @ResponseBody String hello() {
    return "hello world";
}
```

5.1.1. @ResponseBody 注解表示该方法的返回的结果直接写入 HTTP 响应正文（ResponseBody）中，一般在异步获取数据时使用，通常是在使用 @RequestMapping 后，返回值通常解析为跳转路径，加上 @ResponseBody 后返回结果不会被解析为跳转路径，而是直接写入HTTP 响应正文中。

5.2. @RestController

```
// = @Controller + @ResponseBody
```

5.3. @GetMapping("/boot/getUser1")

```
//只支持get请求
//相当于上面的requestMapping 加上 method=RequestMethod.GET
```

5.4. @PostMapping("/boot/getUser2")

//只支持post请求

//相当于上面的requestMapping 加上 method=RequestMethod.POST

//直接在地址栏输入地址相当于get请求,这里会报405错误,不支持该请求

5.5. @PathVariable

=====

```
@RequestMapping("/boot/restful/{id}")
public Object user(@PathVariable("id") Integer id) {
    return "";
}
```

5.6. @RequestParam

http://localhost:8080/springmvc/hello/101?param1=10¶m2=20

```
public String getDetails(
    @RequestParam(value="param1", required=true) String param1,
    @RequestParam(value="param2", required=false) String param2){
    ...
}
```

6. JSP

6.1. 所需依赖

6.1.1. <!-- 引入spring-boot内嵌的Tomcat对J s p的解析包 -->

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<!-- servlet.API -->
<dependency>
    <groupId>javax.servlet</groupId>
```

```

        <artifactId>javax.servlet-api</artifactId>
    </dependency>
    <!-- servlet.jsp-API -->
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL依赖 -->
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>

```

6.2. 在src/main下新建webapp目录,并在该目录下新建jsp

6.3. SpringMVC配置(在application.yml中配置)

6.3.1. # spring mvc

spring:

mvc:

view:

prefix: /

suffix: .jsp

6.4. demo

6.4.1. @Controller

```

public class JSPController {
    @RequestMapping("/boot/index")
    public String index(Model model) {
        model.addAttribute("msg", "spring boot jsp");
    }
}

```



```

        return "index";
    }
}

```

7. Mybatis

7.1.1. 依赖

7.1.1. <!-- 集成Mybatis -->

```

<!-- mybatis-spring-boot-starter -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>
<!-- mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

```

7.2.2. 在application.properties中配置mybatis的Mapper.xml文件所在位置

mybatis.mapper-locations=classpath:com/demo/mapper/*.xml

7.3.3. 配置数据库数据源

7.3.1. spring.datasource.username=root

spring.datasource.password=123456

spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=utf-8&useSSL=false

7.4.4.在mybatis的Mapper接口中添加@Mapper注解或者在运行的主类上添加@MapperScan("com.dmeo.mapper")注解扫描包

8. SSM

8.1. Mapper(DAO)

8.1.1. @Mapper // 必要的注解,相当于配置文件中spring 中配置bean

```
public interface UserMapper {  
    List<User> selectAll();  
    User selectById(Integer id);  
}
```

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

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >

<mapper namespace="com.demo.mapper.UserMapper">

<select id="selectAll" resultType="com.demo.model.User">

select t_id as id,t_username as username,t_password as password from t_user

</select>

<!-- 根据id查询数据表中的一条记录，并封装User对象 -->

<select id="selectById" resultType="com.demo.model.User">

select t_id as id,t_username as username,t_password password from t_user

where t_id=#{id};

</select>

8.2. Service

8.2.1. public interface UserService {

public List<User> selectAll();

public User selectById(Integer id);

}

8.2.2. @Service

```
public class UserServiceImpl implements UserService{  
    @Autowired  
    private UserMapper userMapper;  
    public List<User> selectAll() {  
        return userMapper.selectAll();  
    }  
    public User selectById(Integer id) {  
        return userMapper.selectById(id);  
    }  
}
```

8.3. Controller

8.3.1. @RestController

```
public class MybatisController {  
    @Autowired  
    private UserService userService;  
    @GetMapping("/boot/getUsers")  
    public Object users() {  
        return userService.selectAll();  
    }  
}
```

9. 事务支持

9.1.1.在入口类中使用注解@EnableTransactionManagement开启事务支持

9.2.2.在访问数据库的service方法上增加注解@Transactional

10. 热部署插件

10.1. <!-- 热部署插件spring-boot-devtools -->

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

11. RESTFul API

11.1. 1.@PathVariable

11.1.1. // http://localhost:8080/boot/restful/2/Raynor

@RequestMapping("/boot/restful/{id}/{name}")

public Object user1(@PathVariable("id") Integer id, @PathVariable("name")

String name) {

 User user = new User();

 user.setId(id);

 user.setUsername(name);

 return user;

}

11.2. 2.增加POST方法

11.2.1. @PostMapping

11.2.2. 接受和处理post请求

11.3. 3.删除DELETE方法

11.3.1. @DeleteMapping

11.3.2. 接收delete方式请求,可以用GetMapping替代

11.4. 4.修改PUT方法

11.4.1. @PutMapping

接收put方式请求,可以用PostMapping替代

11.5. 5.查询GET方法

11.5.1. @GetMapping

接收get方式请求

12. Resdis 集成

12.1. 1.依赖

12.1.1. <dependency>

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

```
<artifactId>spring-boot-starter-data-redis</artifactId>
```

```
</dependency>
```

12.2. 2.redis 配置

12.2.1. spring.redis.host=localhost

```
spring.redis.port=6379
```

```
spring.redis.password=
```

```
//默认密码为空 (redis)
```

12.3. 3.在使用的类中注入// 注入一个spring boot 配置好的 RedisTemplate

(Service Impl中)

```
@Autowired
```

```
private RedisTemplate<Object, Object> redisTemplate;
```

```
//这里的泛型只能是Object,Object 或者String,String
```

12.4. 4.demo

12.4.1. @Override

```
public List<User> selectAll() { // 使用Redis缓存
```

```

//字符串序列化器
RedisSerializer redisSerializer = new StringRedisSerializer();

redisTemplate.setKeySerializer(redisSerializer);//对key进行String序列化,增强
可读性

// 查询缓存
@SuppressWarnings("unchecked")
List<User> list = (List<User>)
redisTemplate.opsForValue().get("allUsers");//User要实现序列化接口
//因为存在redis中的数据都是序列化后的
if (list == null) {
    // 查询为空 再查询一次
    list = userMapper.selectAll();
    // 放到redis 缓存中
    redisTemplate.opsForValue().set("allUsers", list);
}
return list;
}

```

12.5. 5.在高并发的情况下,出现缓存穿透的问题

12.5.1. 假设有10000个人同时进到

上述方法中,都从redis中查询一次均为空,此时所有人都要进入对DB进行查询,DB的压力过大.

解决1:10000个人中,先让一个人查询数据库完成后放到redis中,其余人再从redis中取数据

A.在 方法前面加入 synchronized

关键字,10000个人进到这个方法里面只有一个人有锁

B.对if增加锁

```

public List<User> selectAll() { // 使用Redis缓存
    // 字符串序列化器
    @SuppressWarnings("rawtypes")
    RedisSerializer redisSerializer = new StringRedisSerializer();
    redisTemplate.setKeySerializer(redisSerializer); //
    对key进行String序列化,增强可读性

    // 查询缓存
    // 问题:在高并发的情况下,出现缓存穿透的问题
    List<User> list = (List<User>) redisTemplate.opsForValue().get("allUsers");
    //双重检测
    if (null == list) {
        synchronized (this) { //解决缓存穿透的问题
            list = (List<User>)
                redisTemplate.opsForValue().get("allUsers"); //再查询一次
            if (list == null) {
                // 查询为空 再查询一次
                list = userMapper.selectAll();
                // 放到redis 缓存中
                redisTemplate.opsForValue().set("allUsers", list);
            }
        }
    }

    return list;
}

```

13. Interceptor

13.1. 1.按照SpringMVC写一个拦截器类

```

13.1.1. public class LoginInterceptor implements HandlerInterceptor{
    @Override

```

```

    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
        throws Exception {
        System.out.println("进入Login拦截器.....");
        //返回为真则可以通过,返回为false
        response.sendRedirect("http://localhost:8080/index.jsp");
        return false;
    }
}

```

13.2. 2.编写一个配置类继承WebMvcConfigurerAdapter类

SpringBoot 2.0 因为Spring5弃用了上述类,
 -改为继承WebMvcConfigurationSupport
 -implements WebMvcConfigurer

```

13.2.1. public void addInterceptors(InterceptorRegistry registry) {
    String [] PathPatterns = {
//        "/boot/**"
    };
    String [] exPathPatterns = {
        "/boot/hello"
    };
    registry.addInterceptor(new LoginInterceptor())
        .addPathPatterns(PathPatterns).excludePathPatterns(exPathPatterns);
}

```

13.3. 3.为该配置类添加@Configuration注解,标注此类为一个配置类,让SpringBoot扫描到

13.4. 4.覆盖其中的方法,并添加已经写好的拦截器

14. Servlet

14.1. 方式一

14.1.1. 1.写一个Servlet

```
@WebServlet("/myservlet")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        doPost(req, resp);
    }
    public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        resp.getWriter().print("myServlet");
        resp.getWriter().flush();
        resp.getWriter().close();
    }
}
```

14.1.2. 2.再main方法前加扫描注解

```
@ServletComponentScan(basePackages="com.demo.servlet")
```

14.2. 方式二

14.2.1. 1.写一个Servlet

```
@WebServlet("/myservlet")
public class MyServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
```

```

        doPost(req, resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        resp.getWriter().print("myServlet");
        resp.getWriter().flush();
        resp.getWriter().close();
    }
}

```

14.2.2. 2.写一个配置类

```

/*<bean id="ServletRegistrationBean"
class="org.springframework.boot.web.servlet.ServletRegistrationBean">
*
* </bean>
*
* */
@Bean
public ServletRegistrationBean<MyServlet2> heServletRegistrationBean(){
    ServletRegistrationBean<MyServlet2> bean = new
ServletRegistrationBean<>(new MyServlet2(), "/myservlet2");
    return bean;    }

```

15. Filter

15.1. 方式一

15.1.1. 1.写一个Filter

15.1.2. 2.再main方法前加扫描注解

```
@ServletComponentScan(basePackages=
{"com.demo.servlet","com.demo.filter"})
```

15.2. 方式二

15.2.1. 写一个Filter

15.2.2. 写一个配置类

```
@Configuration
public class ServletConfig {
    /*<bean id="FilterRegistrationBean"
    class="org.springframework.boot.web.servlet.FilterRegistrationBean">
    * </bean>
    * */
    @Bean
    public FilterRegistrationBean<MyFilter2> heFilterRegistration() {
        FilterRegistrationBean<MyFilter2> filterRegistrationBean = new
        FilterRegistrationBean<MyFilter2>(new MyFilter2());
        filterRegistrationBean.addUrlPatterns("/");
        return filterRegistrationBean;
    }
}
```

16. 项目编码配置

16.1. 1.方式一

16.1.1. 使用字符编码过滤器(Filter)//等价于

```
@Bean
public FilterRegistrationBean filterRegistration() {
    FilterRegistrationBean filterRegistrationBean = new FilterRegistrationBean();
    CharacterEncodingFilter encodingFilter =new CharacterEncodingFilter();
```

```

        encodingFilter.setForceEncoding(true);
        encodingFilter.setEncoding("UTF-8");
        filterRegistrationBean.setFilter(encodingFilter);
        filterRegistrationBean.addUrlPatterns("/*");
        return filterRegistrationBean;
    }

```

注意:只有当spring.http.encoding.enabled=false 配置成false才会生效

16.1.2. 然后在主类上需要扫描此过滤器,扫描包

16.2. 2.方式二

16.2.1. 在核心配置文件中配置:(SpringBoot1.4.2之后才有的)

```
spring.http.encoding.charset=utf-8
```

```
spring.http.encoding.enable=false
```

```
spring.http.encoding.force=true
```

17. 部署

17.1. WAR

17.1.1. 1.程序入口类需要继承 `SpringBootServletInitializer`类

17.1.2. 2.程序入口类覆盖如下方法

```

public SpringApplicationBuilder configure(SpringApplicationBuilder application)
{
    return application.sources(SpringBootApplication.class);
}

```

17.1.3. 3.配置插件

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
</dependency>
```

17.1.4. 4.将POM中的<packaging>jar</packaging>改为war

17.1.5. 5.在maven

install中在本地仓库中安装成一个war包,然后将war部署到tomcat中运行

17.2. JAR

17.2.1. 配置插件

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <version>1.4.2</version>//需要用这个版本
</dependency>
```

17.2.2. install

17.2.3. java -jar spring-boot-demo.jar

18. 执行器（Actuator）

18.1. 在生产环境中,需要定时或者定期监控服务的可用性,springboot的actuator功能提供了很多所需要的接口

是springboot提供的对应用系统的自省和监控的功能,可以对应用系统进行配置查看,健康查看,相关统计等功能

18.2. 1.依赖

18.2.1. <dependency>

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

18.3. 2.application.properties中指定监控的HTTP端口:

如果不指定,则使用和server相同的端口

#服务运行的端口

server.port=8080

#actuator端口配置

#actuator监控的上下文,不配置默认使用tomcat的上下文

#management.server.servlet.context-path=/项目名

#不配置,默认使用tomcat的端口

management.server.port=8080

#默认只开启了health和info,设置为*,则包含所有web入口端点

management.endpoints.web.exposure.include=*

18.4. 3.访问http://localhost:8080/actuator/health

19. 日志

19.1. 日志接口层(抽象层)

19.1.1. JCL/SLF4j/jboss-logging

Spring默认使用JCL(common-logging)

19.2. 日志实现层

19.2.1. Log4j/Log4j2/Logback/JUL

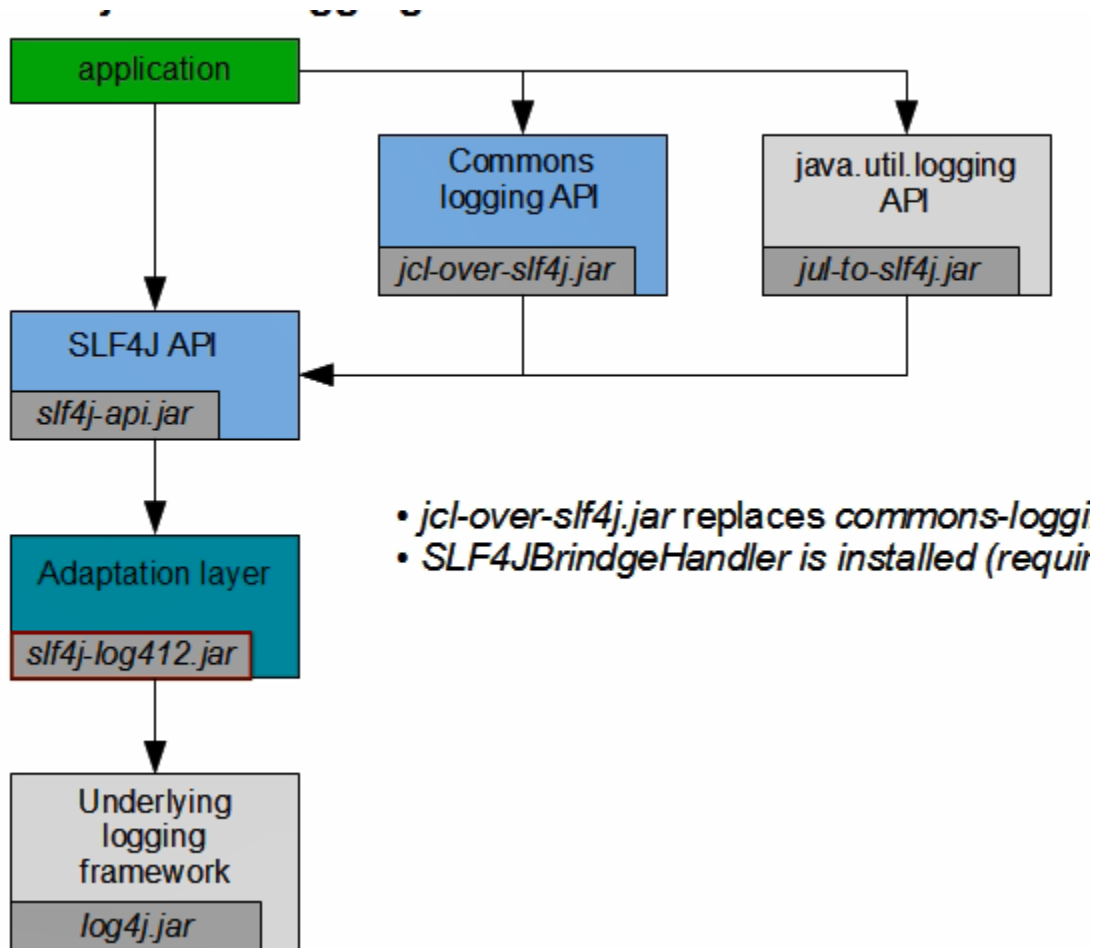
SpringBoot默认使用Logback

19.2.2. 调用的时候,调用接口层的方法;配置的时候,需要配置实现层

19.3. 统一日志记录

19.3.1. question: a(slf4j+logback) :spring(common-loggng),hibernate(jboss-loggng)

19.3.2. 使用原框架的替换jar包:jcl-over-slf4j.jar



19.3.3. SpringBoot统一:(slf4j)









先排除其它日志

中间包替换

导入slf4j的其它实现

19.4. SpringBoot日志关系

19.4.1.

- ✓  spring-boot-starter-web : 2.0.4.RELEASE [compile]
- ✓  spring-boot-starter : 2.0.4.RELEASE [compile]
 -  spring-boot : 2.0.4.RELEASE (omitted for conflict with 2.0.4.RELEASE)
 -  spring-boot-autoconfigure : 2.0.4.RELEASE (omitted for conflict with 2.0.4.RELEASE)
- ✓  spring-boot-starter-logging : 2.0.4.RELEASE [compile]
 - >  logback-classic : 1.2.3 [compile]
 - >  log4j-to-slf4j : 2.10.0 [compile]
 - >  jul-to-slf4j : 1.7.25 [compile]

19.4.2. 1.默认使用slf4j/logback

2.已经替换过 jul/log4j

3.如果要引入其它的日志框架,一定要移除默认的日志依赖

就像SpringBoot移除common-logging一样

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <exclusions>
    <exclusion>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

19.5. SpringBoot使用slf4j-logback

19.5.1. `Logger logger=LoggerFactory.getLogger(getClass());`

```
@Test
public void contextLoads() {
    //日志的级别;
    //由低到高 trace<debug<info<warn<error
    //可以调整输出的日志级别; 日志就只会在这个级别以以后的高级别生效

    logger.trace("这是trace日志...");
    logger.debug("这是debug日志...");

    //SpringBoot默认给我们使用的是info级别的, 没有指定级别的就用SpringBoot默认规定的级别; root级别

    logger.info("这是info日志...");
    logger.warn("这是warn日志...");
    logger.error("这是error日志...");

}
```

19.5.2. 配置方式

核心配置文件

```
logging.level.com.atguigu=trace
#logging.path=
# 不指定路径在当前项目下生成springboot.log日志
# 可以指定完整的路径;
#logging.file=G:/springboot.log
# 在当前磁盘的根路径下创建spring文件夹和里面的log文件夹; 使用 spring.log 作为默认文件
logging.path=/spring/log
```

```
# 在控制台输出的日志的格式
logging.pattern.console=%d{yyyy-MM-dd} [%thread] %-5level %logger{50} - %
msg%n
# 指定文件中日志输出的格式
logging.pattern.file=%d{yyyy-MM-dd} === [%thread] === %-5level === %logger
{50} ===== %msg%n
```

指定配置

Logging System	Customization
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

logback.xml: 直接就被日志框架识别了;

logback-

spring.xml: 日志框架就不直接加载日志的配置项, 由SpringBoot解析日志配置, 可以使用SpringBoot的高级Profile功能

19.6. 切换日志框架

19.6.1. slf4j+log4j(默认的logback性能优于log4j)

```
<dependency>
```

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

```
<artifactId>spring-boot-starter-web</artifactId>
```

```
<exclusions>
```

```
<exclusion>
```

```

    <artifactId>logback-classic</artifactId>
    <groupId>ch.qos.logback</groupId>
  </exclusion>
  <exclusion>
    <artifactId>log4j-over-slf4j</artifactId>
    <groupId>org.slf4j</groupId>
  </exclusion>
</exclusions>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
</dependency>

```

19.6.2. slf4j+log4j2

```

<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>

```

SpringBoot2 中要额外移除:

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

在resources中新建 log4j2-spring.xml, springboot会自动加载

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <properties>
    <!-- 文件输出格式 -->
    <property name="PATTERN">log4j2|%d{yyyy-MM-dd HH:mm:ss.SSS} | -
%-5level [%thread] %c [%L] - | %msg%n</property>
  </properties>

  <appenders>
    <Console name="CONSOLE" target="system_out">
      <PatternLayout pattern="{PATTERN}" />
    </Console>
  </appenders>

  <loggers>
    <logger name="com.roncoo.education" level="debug" />
    <root level="info">
```

```

        <appenderref ref="CONSOLE" />
    </root>
</loggers>

</configuration>

```

20. 静态资源

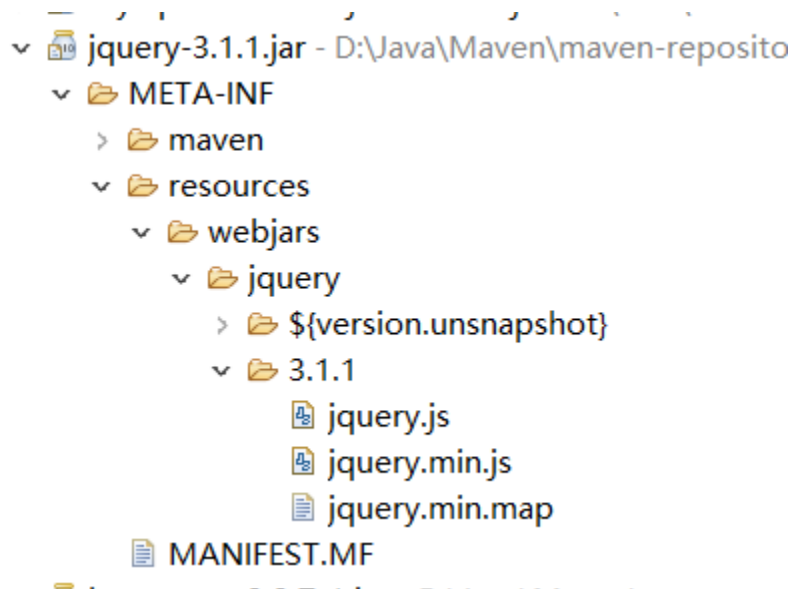
20.1. webjar

20.1.1. 所有 /webjars/**，都去 classpath:/META-INF/resources/webjars/ 找资源；

20.1.2. 选用webjar来管理前台资源文件，因为通过手工进行管理，容易导致文件混乱、版本不一致等问题。而WebJars是将这些通用的Web前端资源打包成Java的Jar包，然后借助Maven工具对其管理，保证这些web资源版本唯一性，升级也比较容易。

<http://www.webjars.org/>

20.1.3.



20.1.4. `<link href="/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css" rel="stylesheet">`
`<script src="/webjars/jquery/3.1.1/jquery.min.js"></script>`

20.2. `"/**"` 访问当前项目的任何资源，都去（静态资源的文件夹）找映射

20.2.1. `"classpath:/META-INF/resources/"`,
`"classpath:/resources/"`,
`"classpath:/static/"`,
`"classpath:/public/"`

`"/`：当前项目的根路径

`localhost:8080/abc` === 去静态资源文件夹里面找`abc`

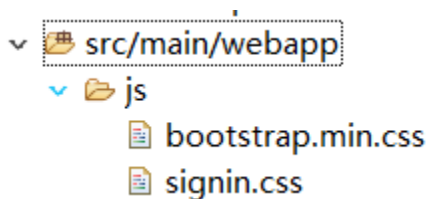
20.3. 欢迎页

20.3.1. 静态资源文件夹下的所有`index.html`页面；被`"/**"`映射；
`localhost:8080/` 找`index`页面

20.4. 所有的 `**/favicon.ico` 都是在静态资源文件下找

20.5. 用户`css/js`

20.5.1.



20.5.2. `<link href="js/signin.css" rel="stylesheet">`

21. 集成Dubbo

21.1. 1. 依赖

21.1.1. <dependency>

```
<groupId>com.alibaba.boot</groupId>  
<artifactId>dubbo-spring-boot-starter</artifactId>  
<version>0.2.0</version>  
</dependency>
```

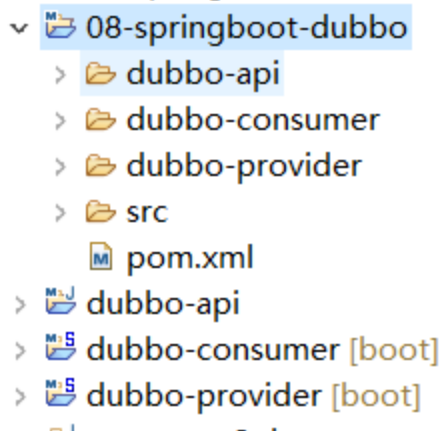
21.2. 2.集成

21.2.1. 二级maven结构

1新建maven project:

```
<packaging>pom</packaging>  
<parent>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-parent</artifactId>  
  <version>2.0.4.RELEASE</version>  
  <relativePath /> <!-- lookup parent from repository -->  
</parent>  
<modules>  
  <module>dubbo-api</module>  
  <module>dubbo-provider</module>  
  <module>dubbo-consumer</module>  
</modules>
```

2.新建maven子工程



21.2.2. 开发Dubbo服务接口

```
<modelVersion>4.0.0</modelVersion>
```

```
<artifactId>dubbo-api</artifactId>
```

```
<name>dubbo-api</name>
```

```
<parent>
```

```
<groupId>com.demo</groupId>
```

```
<artifactId>08-springboot-dubbo</artifactId>
```

```
<version>1.0.0</version>
```

```
</parent>
```

```
<packaging>jar</packaging>
```

```
</project>
```

创建service和model包,并写两个类

maven运行 install

成功之后会在maven本地仓库的com.demo中,找到名为dubbo-api-

1.0.0的jar包,这个包作为提供者 and 消费者都需要导入的

21.2.3. 开发Dubbo服务提供者

1.创建提供者springboot项目:dubbo-provider

2.加入springboot与dubbo集成的起步依赖

```
<dependency>
  <groupId>com.alibaba.boot</groupId>
  <artifactId>dubbo-spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>com.101tec</groupId>
  <artifactId>zkclient</artifactId>
  <version>0.10</version>
</dependency>

<dependency>
  <groupId>com.demo</groupId>
  <artifactId>dubbo-api</artifactId>
  <version>${project.parent.version}</version>
</dependency>
```

3.在springboot的核心配置文件中配置dubbo的信息

```
# Spring boot application
spring.application.name = dubbo-provider-demo
server.port = 9090

# Service version
demo.service.version = 1.0.0

# Base packages to scan Dubbo Components (e.g @Service , @Reference)
dubbo.scan.basePackages = com.demo.provider.service

# Dubbo Config properties
## ApplicationConfig Bean
```

```
#提供发应用名称,用于计算依赖关系
dubbo.application.id = dubbo-provider-demo
dubbo.application.name = dubbo-provider-demo
```

```
## ProtocolConfig Bean
```

```
#使用dubbo协议,在12345端口暴露服务
```

```
dubbo.protocol.id = dubbo
dubbo.protocol.name = dubbo
dubbo.protocol.port = 12345
```

```
## RegistryConfig Bean
```

```
#使用zookeeper注册中心暴露服务地址
```

```
dubbo.registry.id = my-registry
dubbo.registry.address = zookeeper://127.0.0.1:2181
#dubbo.registry.address = N/A
```

4.service

```
package com.demo.provider.service.impl;
import org.springframework.stereotype.Component;
import com.alibaba.dubbo.config.annotation.Service;
import com.dubbo.model.User;
import com.dubbo.service.UserService;
```

```
@Component
```

```
@Service(
    version = "${demo.service.version}",
    application = "${dubbo.application.id}",
    protocol = "${dubbo.protocol.id}",
    registry = "${dubbo.registry.id}"
```

```

)
public class UserServiceImpl implements UserService{
    @Override
    public User getUser(int id) {
        return null;
    }
    @Override
    public String sayHi(String name) {
        // TODO Auto-generated method stub
        return "hello dubbo";
    }
}

```

21.2.4. 开发Dubbo服务消费者

1.pom

```

<dependency>
    <groupId>com.alibaba.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>com.demos</groupId>
    <artifactId>dubbo-api</artifactId>
    <version>${project.parent.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>

```

```
<groupId>com.101tec</groupId>  
<artifactId>zkclient</artifactId>  
<version>0.10</version>  
</dependency>
```

2.核心配置

Spring boot application

```
spring.application.name = dubbo-consumer-demo  
server.port = 8080
```

Service Version

```
demo.service.version = 1.0.0
```

```
#dubbo.scan.basePackages = com.demo.consumer.controller
```

Dubbo Config properties

ApplicationConfig

Bean消费方应用名称,用于计算依赖关系,不是匹配条件,不能和提供方名称一样

```
dubbo.application.id = dubbo-consumer-demo  
dubbo.application.name = dubbo-consumer-demo
```

```
dubbo.registry.address = zookeeper://127.0.0.1:2181  
dubbo.registry.check=false
```

ProtocolConfig Bean

```
dubbo.protocol.id = dubbo  
dubbo.protocol.name = dubbo  
dubbo.protocol.port = 12345
```

3.控制器

```
package com.demo.consumer.controller;

import java.io.IOException;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import com.alibaba.dubbo.config.annotation.Reference;
import com.dubbo.service.UserService;

@Controller
public class UserController {
    @Reference(version = "${demo.service.version}")
    private UserService userService;

    @ResponseBody
    @RequestMapping("/hello")
    public Object hello() throws IOException {
        String a = userService.sayHi("aaaaa");
        return a;
    }
}
```

21.2.5. 开启zookeeper服务,然后分别运行dubbo-provider和dubbo-consumer