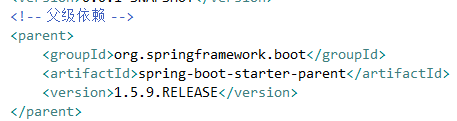
**一、新建Springboot项目，demo1**

1.新建maven项目，编辑pom文件，添加parent标签

1.1 

1.2 

1.3 

1.4 新建controller



1.5 运行结果

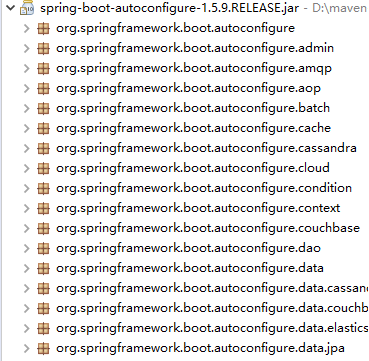


1.6分析

@EnableAutoConfiguration：启用自动配置，该注解会使SpringBoot根据项目以来的jar包自动配置项目的配置项，例如我们添加了spring-boot-starter-web的以来，项目中就会引入SpringMVC的依赖，SpringBoot就会自动配置Tomcat和SpringMVC

@EnableAutoConfiguration里面有个属性（exclude用于排除不自动配置的jar包，exclude={xx.classs}）

支持的自动配置jar包如下

等

2. 全局配置文件

全局配置文件的名称一定要命名为**application**，扩展名为properties或者yml，在resources目录下或者类路径下的/config下，一般我们放到resource下。

Yml文件



Properties文件



3. starter pom

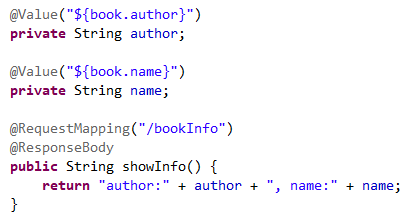
SpringBoot为我们提供了企业级开发绝大多数场景的starter pom，只要使用了应用场景所需的starter pom，就可以得到SpringBoot为我们提供的自动配置的bean

**二、springBoot项目demo2**

1. 获取自定义配置的值

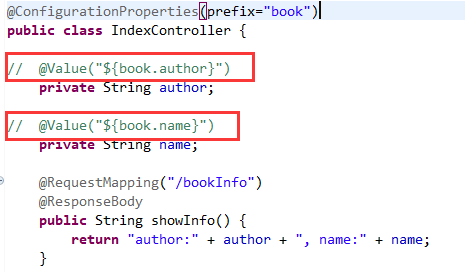
在application.properties文件中写两个变量

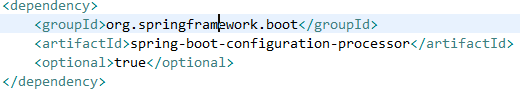




2.类型安全的配置

使用@Value注入每个自定义配置再项目中显得很麻烦，当自定义属性很多时需要注入很多次。SpringBoot还提供了基于类型安全的配置方式，通过@ConfigurationProperties将properties中的属性和一个bean属性关联，从而实现类型安全的配置，**这个前提是要生成set和get方法。**



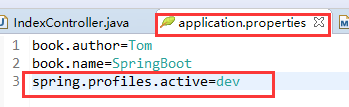


设置完前缀之后，只需要在成员变量的命名中使用properties文件中的属性名命名即可。

3. profile配置

Profile是针对不同的环境对不同的配置提供支持的，全局profile配置使用application-\*.properties(application-prod.properties, application-test.properties, application-dev.properties)

通过在application.properties中设置**spring.profiles.active=prod**来指定活动的profile。



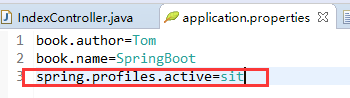




运行之后可以看到开发环境的端口被启动。



修改application.properties中的active为sit



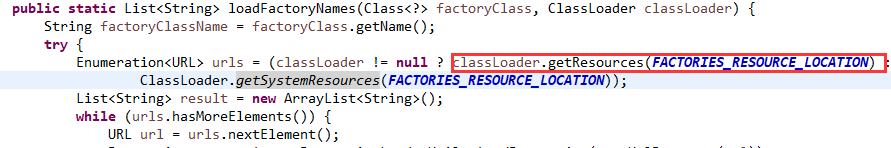
重新运行得到结果



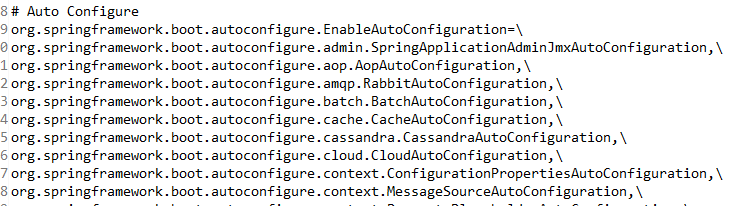
4. springboot自动配置的原理@EnableAutoConfiguration

从SpringApplication中的initialize方法中出发，调用getSpringFactoriesInstances方法，方法中调用getSpringFactoriesInstances，再到SpringFactoriesLoader.

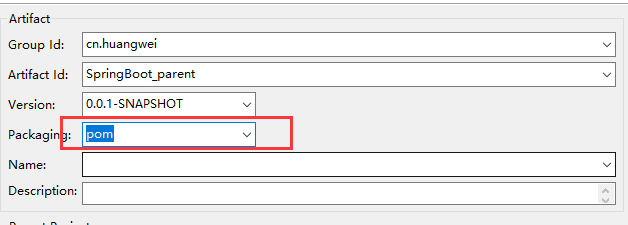
*loadFactoryNames*加载相关的factoryname；再到



**而这个*FACTORIES\_RESOURCE\_LOCATION*** = "META-INF/spring.factories"，打开该文件，可以看到各种jar包的内容，而getSpringFactoriesInstances就是用来对每个jar的包名和类名进行实例化。

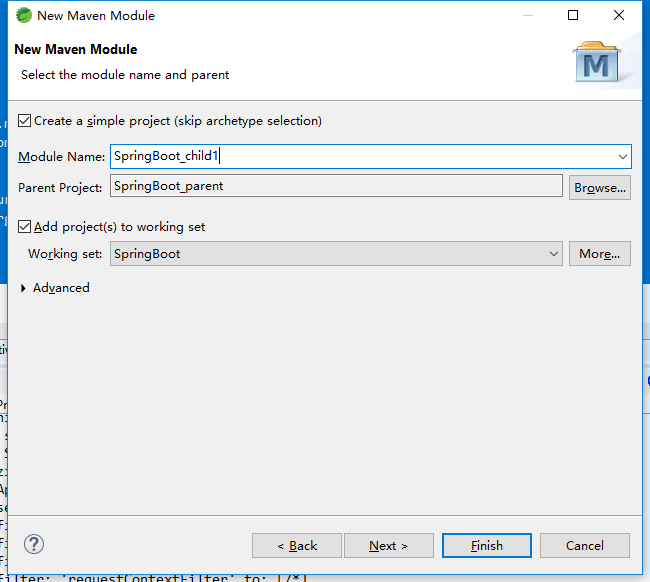


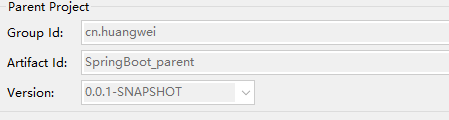
5.创建父工程

这个一定要选pom工程。

|  |
| --- |
| <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.9.RELEASE</version>  </dependency>  </dependencies>  </dependencyManagement>    <!-- 指定maven编译的版本 -->  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>    <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>  </build> |

新建maven module；

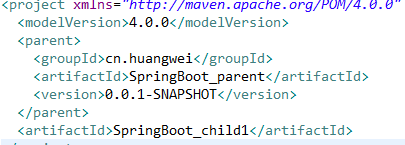




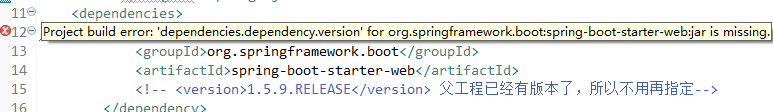
同时在父工程中，你会发现新增了一个module；



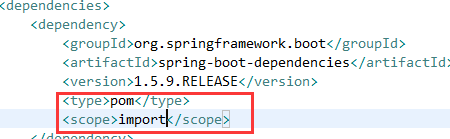
子工程中会显示父工程的信息



此时如果往子工程中加入starter-web依赖，那么会提示



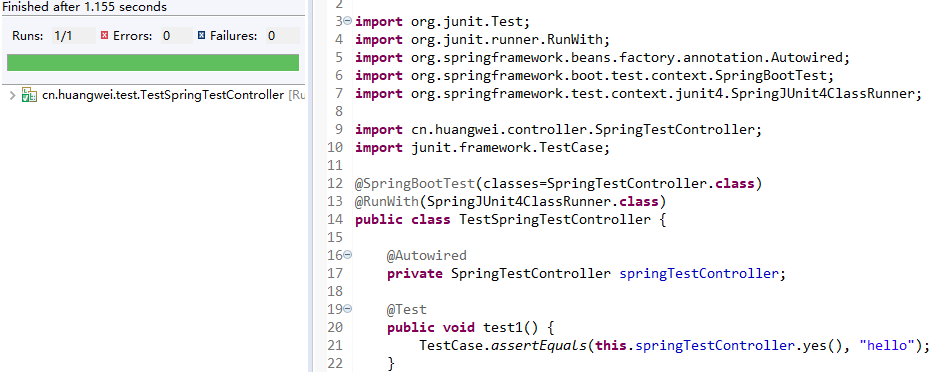
此时还需要对父工程中的pom文件进行修改



**三、springBoot项目demo3**

1. springBoot整合测试

|  |
| --- |
| **TestSpringTestController.java**  **package** cn.huangwei.test;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner;  **import** cn.huangwei.controller.SpringTestController;  **import** junit.framework.TestCase;  @SpringBootTest(classes=SpringTestController.**class**)  @RunWith(SpringJUnit4ClassRunner.**class**)  **public** **class** TestSpringTestController {    @Autowired  **private** SpringTestController springTestController;    @Test  **public** **void** test1() {  TestCase.*assertEquals*(**this**.springTestController.yes(), "hello");  }  }  Pom文件  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>    <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency> |



2. SpringBootApplication注解与RestController注解

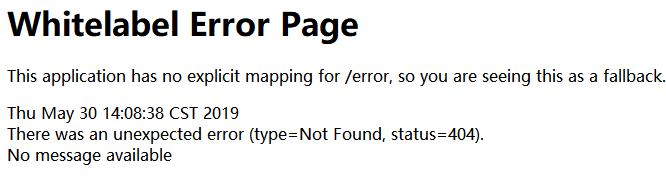
新增启动类SpringApplications.java，内容如下：注意run方法里面一定要是方法所在类的字节码。

|  |
| --- |
| **package** cn.huangwei.app;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.EnableAutoConfiguration;  **import** cn.huangwei.controller.SpringTestController;  @EnableAutoConfiguration  **public** **class** SpringApplications {    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(**SpringApplications.class**, args);  }  } |

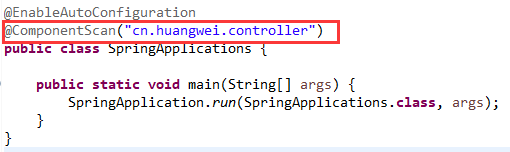
而SpringTestController就变为：

|  |
| --- |
| **package** cn.huangwei.controller;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  @Controller  **public** **class** SpringTestController {    @RequestMapping("/hello")  @ResponseBody  **public** String yes() {  **return** "hello";  }  } |

运行并访问<http://localhost:8080/hello/>

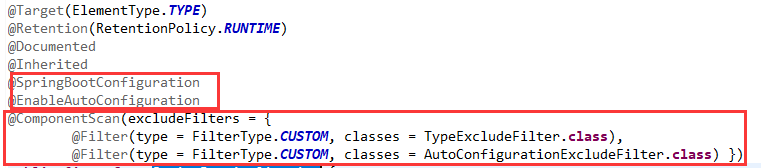


说明controller并没有实例化，无法找到对应url地址；此时需要在启动类SpringApplication上添加ComponentScan以扫描对应的bean对象，从而进行实例化。



重试后成功。

@EnableAutoConfiguration和@ComponentScan注解可以使用@SpringBootApplication代替，因为该注解里面有：



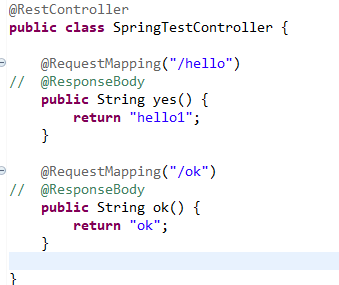
@SpringBootApplication默认情况下扫描的是当前包及当前包的子包

如果不是，还需要另外指定扫描的地方。

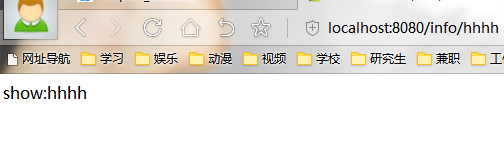
|  |
| --- |
| **package** cn.huangwei.app;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  //@EnableAutoConfiguration  //@ComponentScan("cn.huangwei.controller")  @SpringBootApplication(scanBasePackages={"cn.huangwei.controller"})//组合注解  **public** **class** SpringApplications {    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SpringApplications.**class**, args);  }  } |

@ResponseBody注解表示，返回的是RESTful内容，如果不适用该注解就会进行跳转。

如果存在多个方法，并且该Controller只想返回restful内容，那么可以将@Controller注解变为@RestController；



|  |
| --- |
| //支持Rest风格  @RequestMapping("/info/{msg}")  **public** String show(@PathVariable String msg) {  **return** "show:" + msg;  }  如果url为localhost:8080/info/hhhh，那么hhhh值就会被赋值给msg，这就是@PathVariable的作用 |



3. SpringBoot默认的日志管理logback

SpringBoot使用的默认日志框架是Logback，并用INFO级别输出到控制台：

日志输出内容元素具体如下：

时间日期：精确到毫秒

日志级别：ERROR，WARN，INFO，DEBUG，TRACE

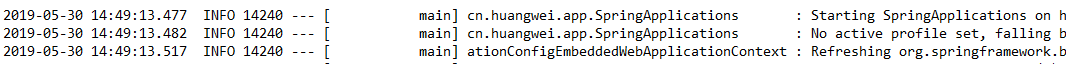
进程ID

分隔符：--标识实际日志的开始

线程名：方括号括起来(可能会阶段控制台输出)

Logger名：通常使用源代码的类名

日志内容



日志依赖：该依赖内容就是SpringBoot默认的日志框架logback，不需要另外导入，已经默认导入了。

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </dependency> |

日志级别（从低到高）：

TRACE < DEBUG < INFO < WARN < ERROR < ALL < OFF

如果设置为WARN，那么低于WARN的信息都不会输出

以下配置需要在application.properties文件中设置

##root日志以WARN级别输出

logging.level.root=WARN(让日志只输出warn及以上级别的信息)

##springframework.web 日志以DEBUG级别输出

logging.level.org.springframework.web=DEBUG

##hibernate 日志以ERROR级别输出

logging.level.org.hibernate=ERROR

默认情况下，SpringBoot将日志输出到控制台，不会写到日志文件中。如果要编写除控制台输出之外的日志文件，则需要application.properties中设置logging.file 或者logging.path属性

logging.file=log/my.log(相对)或logging.file=/log/my.log(绝对)

logging.path设置目录，会在该目录下创建spring.log文件，并写入日志内容，如logging.path=/var/log

如果只配置logging.file, 会在项目的当前路径下生成一个xxx.log日志文件。

如果只配置logging.path, 会在/var/log文件夹下生成日志文件spring.log

**注意**：二者不可同时使用，若同时使用，则只有logging.file生效

默认情况下，日志文件的大小达到10MB时，会切分一次，产生新的日志文件，默认级别为：ERROR, WARN, INFO

|  |
| --- |
| logging.level.root=WARN  logging.level.org.springframework.web=DEBUG  logging.file=d:/log/info.log  ##控制台显示日志的格式  logging.pattern.console=%d{yyyy/MM/dd-HH:mm:ss} [%thread] %-5level %logger- %msg%n  ##文件显示日志的格式  logging.pattern.file=%d{yyyy/MM/dd-HH:mm} [%thread] %-5level %logger- %msg%n |

4. 自定义日志配置

可以使用其他系统对日志进行配置

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- scan表示当配置文件发生变化时，会进行重新加载 -->  <configuration scan=*"true"* scanPeriod=*"60 seconds"* debug=*"false"*>  <contextName>logback</contextName>  <property name=*"log.path"* value=*"d:/log/info.log"* />  <!-- 输出到控制台 -->  <appender name=*"console"* class=*"ch.qos.logback.core.ConsoleAppender"*>  <encoder>  <pattern>%d{HH:mm:ss.SSS} %contextName [%thread] %-5level %logger{36} - %msg%n</pattern>  </encoder>  </appender>  <!-- 输出到文件 -->  <appender name=*"file"* class=*"ch.qos.logback.core.rolling.RollingFileAppender"*>  <file>${log.path}</file>  <rollingPolicy class=*"ch.qos.logback.core.rolling.TimeBasedRollingFileAppender"*>  <!-- 日志切分方式，每一天的日志归并到同一天 -->  <fileNamePattern>logback.%d{yyyy-MM-dd}.log</fileNamePattern>  </rollingPolicy>  <encoder>  <pattern>%d{HH:mm:ss.SSS} %contextName [%thread] %-5level %logger{36} - %msg%n</pattern>  </encoder>  </appender>  <root level=*"info"*>  <appender-ref ref=*"console"* />  <appender-ref ref=*"file"* />  </root>    <!-- logback为java中的包 -->  <logger name=*"cn.huangwei.controller"* />  <!-- additivity是否向上级logger传递打印信息 -->  <logger name=*" cn.huangwei.controller.SpringTestController"* level=*"WARN"* additivity=*"false"*>  <appender-ref ref=*"console"*/>  </logger>  </configuration> |

scan：配置文件如果发生变化，将会重新加载，默认值为true；

scanPeriod：设置检测配置文件的时间间隔，默认单位毫秒，默认间隔1分钟。

debug：设置为true时，将打印出logback内部日志信息。默认为false

contextName标签：一旦设置不能更改，可以通过%contextName来获取上下文名称

logger的使用：

<root level=*"info"*>

<appender-ref ref=*"console"* />

<appender-ref ref=*"file"* />

</root>

<!-- logback为java中的包 -->

<logger name=*"cn.huangwei.controller"* />

<!-- additivity是否向上级logger传递打印信息 -->

<logger name=*"cn.huangwei.controller.SpringTestController"* level=*"WARN"* additivity=*"false"*>

<appender-ref ref=*"console"*/>

</logger>

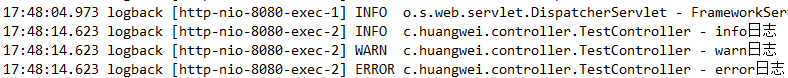
**第一个logger**只指定了包名，这样控制了这个包下面的所有文件的日志的打印。这里的打印级别没有单独设置，故默认与上一级别一致，即root中的info级别，而root中的可以向console和file输出日志信息，默认为console输出。

例如新建一个TestController

|  |
| --- |
| **package** cn.huangwei.controller;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  @Controller  **public** **class** TestController {    **private** Logger logger = LoggerFactory.*getLogger*(**this**.getClass());    @RequestMapping("/show")  @ResponseBody  **public** String show() {  logger.debug("debug日志");  logger.info("info日志");  logger.warn("warn日志");  logger.error("error日志");  **return** "show";  }  } |

该文件是在cn.huangwei.controller包下面的文件，没有设置level，故使用root上级的日志级别，没有设置additity，故默认为true，向上传递。故遵循上一级root的日志输出方式，输出info及info以上级别的日志，默认输出控制台上。

运行项目，在地址栏输入localhost:8080/show，查看控制台的日志输出



<!-- additivity是否向上级logger传递打印信息 -->

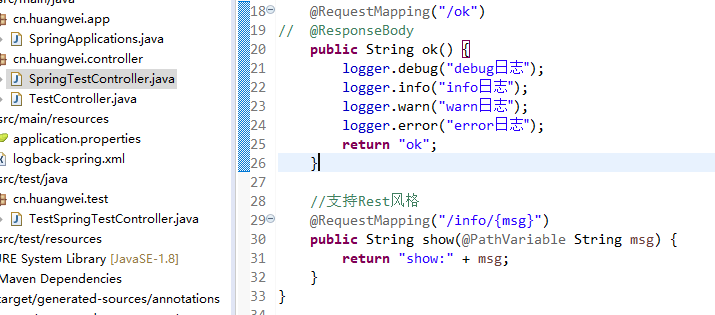
<logger name=*"cn.huangwei.controller.SpringTestController"* level=*"WARN"* additivity=*"false"*>

<appender-ref ref=*"console"*/>

</logger>

第二个logger，对于SpringTestController文件，打印warn及warn级别以上的日志，additivity为false表示不向上级传递日志信息，它的继承关系为***cn.huangwei.controller.SpringTestController的上一级为cn.huangwei.controller，cn.huangwei.controller的上一级为root；***自身通过console打印日志信息

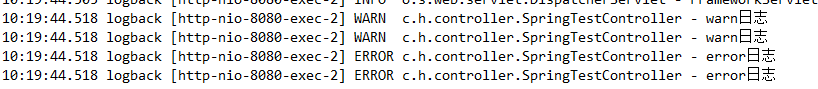
修改SpringTestController类；所以当SpringTestController接收到日志消息时，自身进行打印，上级cn.huangwei.controller的logger没有收到消息，就不需要向上传递，故只打印一份日志。



运行之后得到结果



如果将additivity设置为true，需要向上级传递，由上级来执行日志信息；结果为



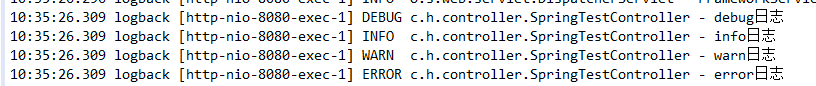
分析：SpringTestController的上一级是cn.huangwei.controller，cn.huangwei.controller的上一级是root，当SpringTestController收到日志消息时，自身通过appender打印消息，然后向上级cn.huangwei.controller传递，由于该logger没有level，没有additity，没有appender，就委托给它上级root打印日志消息，因此日志消息打印第二遍。

问：logger的继承关系（上下级）怎么判断？

是根据logger的name属性中“.”点号分隔符进行判断的，例如cn.huangwei是cn.huangwei.controller的上级，如果没有点号例如huangwei，那么他的上级是root。

|  |
| --- |
| <!-- 根logger，第一级 -->  <root level=*"info"*>  <appender-ref ref=*"console"* />  <appender-ref ref=*"file"* />  </root>    <!-- logger为java中的包下日志管理，additivity默认为true -->  <logger name=*"cn.huangwei.controller"* level=*"debug"*/>  <!-- additivity是否向上级logger传递打印信息 -->  <logger name=*"cn.huangwei"* level=*"warn"* additivity=*"false"*>  <appender-ref ref=*"console"*/>  </logger> |

日志输出结果为：



问：继承关系中的上下级日志等级问题？

如果日志消息对应的logger有level属性，按照level属性来，如果没有，按照上级的level属性来，如果连root都没有，默认debug。

5.多环境日志输出

根据不同的环境（prod：生产环境；test：测试环境；dev：开发环境）来定义不同的日志输出，在logback-spring.xml中使用springProfile节点来定义，方法如下：

|  |
| --- |
| <!-- 测试环境+开发环境 多个环境之间用逗号分隔 -->  <springProfile name=*"test,dev"*>  <logger name= *"cn.huangwei.controller"* level=*"info"* />  </springProfile>  <!-- 生产环境 -->  <springProfile name=*"prod"*>  <logger name= *"cn.huangwei.controller"* level=*"error"* />  </springProfile> |

在application.properties全局配置文件中指明使用哪一种：

spring.profiles.active=prod



spring.profiles.active=test,dev



**四、springBoot项目demo4**

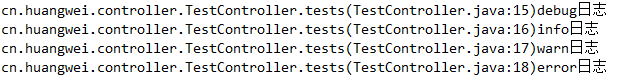
1. 使用log4j日志管理

修改pom.xml文件，过滤掉自带的spring-boot-starter-logging，然后添加spring-boot-starter-log4j依赖包

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </exclusion>  </exclusions>  </dependency>    <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j</artifactId>  <version>1.3.8.RELEASE</version>  </dependency> |

jar包配置完了，还需要配置log4j的配置文件。创建log4j.properties

|  |
| --- |
| # priority :debug<info<warn<error  # you cannot specify every priority with different file for log4j  log4j.rootLogger=debug,console,info,debug,warn,error    # console log  log4j.appender.console=org.apache.log4j.ConsoleAppender  log4j.appender.console.layout=org.apache.log4j.PatternLayout  log4j.appender.console.layout.ConversionPattern= [%d{yyyy-MM-dd HH:mm:ss a}]:%p %l%m%n  # info log  log4j.logger.info=info  log4j.appender.info=org.apache.log4j.DailyRollingFileAppender  log4j.appender.info.DatePattern='\_'yyyy-MM-dd'.log'  log4j.appender.info.File=d://log//info.log  log4j.appender.info.Append=true  log4j.appender.info.Threshold=INFO  log4j.appender.info.layout=org.apache.log4j.PatternLayout  log4j.appender.info.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss a} [Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n  # debug log  log4j.logger.debug=debug  log4j.appender.debug=org.apache.log4j.DailyRollingFileAppender  log4j.appender.debug.DatePattern='\_'yyyy-MM-dd'.log'  log4j.appender.debug.File=d://log//debug.log  log4j.appender.debug.Append=true  log4j.appender.debug.Threshold=DEBUG  log4j.appender.debug.layout=org.apache.log4j.PatternLayout  log4j.appender.debug.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss a} [Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n  # warn log  log4j.logger.warn=warn  log4j.appender.warn=org.apache.log4j.DailyRollingFileAppender  log4j.appender.warn.DatePattern='\_'yyyy-MM-dd'.log'  log4j.appender.warn.File=d://log//warn.log  log4j.appender.warn.Append=true  log4j.appender.warn.Threshold=WARN  log4j.appender.warn.layout=org.apache.log4j.PatternLayout  log4j.appender.warn.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss a} [Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n  # error  log4j.logger.error=error  log4j.appender.error = org.apache.log4j.DailyRollingFileAppender  log4j.appender.error.DatePattern='\_'yyyy-MM-dd'.log'  log4j.appender.error.File = d://log//error.log  log4j.appender.error.Append = true  log4j.appender.error.Threshold = ERROR  log4j.appender.error.layout = org.apache.log4j.PatternLayout  log4j.appender.error.layout.ConversionPattern = %d{yyyy-MM-dd HH:mm:ss a} [Thread: %t][ Class:%c >> Method: %l ]%n%p:%m%n |



2. 配置工程为开发模式

加入一下依赖，代码做了修改，不用重新运行

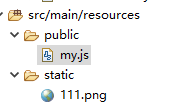
|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency> |



3. springboot的web开发

SpringBoot提供了spring-boot-starter-web为web开发予以支持，spring-boot-starter-web提供了嵌入的Tomcat以及SpringMVC的依赖，web相关的自动配置存储在spring-boot-autoconfigure.jar的org.springframework.boot.autoconfigure.web下

4.访问静态资源

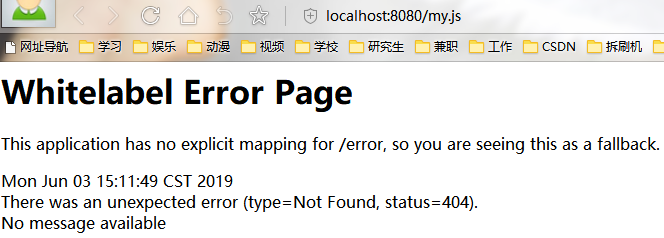
springboot中加载静态资源和在普通的web应用中不太一样，默认情况下，springboot从classpath的/static，/public或者/META-INF/resources文件夹或从ServletContext根目录提供静态资源

如果需要指定特定静态文件路径，需要在application.properties文件中进行修改

#设定静态文件路径，js，css，image等

spring.resource.static-locations=classpath:/static/

如果指定了上述路径，那么在这个路径之外的静态资源便无法访问。



5. 自定义消息转换器

只需要在类中添加消息转化器的@Bean，就会被SpringBoot自动加入到容器中

|  |
| --- |
| //定义消息转换器  @Bean  **public** StringHttpMessageConverter stringHttpMessageConverter() {  StringHttpMessageConverter converter = **new** StringHttpMessageConverter(Charset.forName("UTF-8"));  **return** converter;  }    @RequestMapping("/tt")  @ResponseBody  **public** String testMessageConverter() {  **return** "hello 你好";  } |



修改编码之后，



即使把我们定义的消息转化器注释掉，发现也不会出现乱码，因为springboot默认配置了消息转化器。在autoconfigure.web中的HttpMessageConvertersAutoConfiguration类中的stringHttpMessageConverter()方法。

6． 使用FastJson解析Json数据

springboot默认配置的是jackson

使用FastJson解析Json数据

|  |
| --- |
| <!-- fastjson的依赖 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.15</version>  </dependency> |

配置fastJson作为解析json的方式

|  |
| --- |
| 准备工作：  Person类  **package** cn.huangwei.entity;  **import** java.util.Date;  **import** com.alibaba.fastjson.annotation.JSONField;  **public** **class** Person {    **private** **int** id;  **private** String name;    @JSONField(format="yyyy-MM-dd HH")  **private** Date date;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Date getDate() {  **return** date;  }  **public** **void** setDate(Date date) {  **this**.date = date;  }    }  TestFastJsonController.java  **package** cn.huangwei.controller;  **import** java.util.Date;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  **import** cn.huangwei.entity.Person;  @Controller  **public** **class** TestFastJsonController {      @RequestMapping("/person")  @ResponseBody  **public** Object show() {  Person ren = **new** Person();  ren.setId(66);  ren.setName("赵六");  ren.setDate(**new** Date());  **return** ren;  }  } |

第一种：让启动类继承WebMvcConfigurerAdapter

重写 void configureMessageConverters(List<HttpMessageConverter<?>> converters)方法

|  |
| --- |
| **SpringApplications.java**  **package** cn.huangwei.app;  **import** java.util.List;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.http.converter.HttpMessageConverter;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  **import** com.alibaba.fastjson.serializer.SerializerFeature;  **import** com.alibaba.fastjson.support.config.FastJsonConfig;  **import** com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter;  @SpringBootApplication(scanBasePackages = "cn.huangwei.controller")  **public** **class** SpringApplications **extends** WebMvcConfigurerAdapter{    @Override  **public** **void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  //创建fastJson的消息转换器  FastJsonHttpMessageConverter convert = **new** FastJsonHttpMessageConverter();  //创建Fastjson的配置对象  FastJsonConfig config = **new** FastJsonConfig();  //对Json数据进行格式化  config.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  //将配置和转换器关联起来  convert.setFastJsonConfig(config);  //添加转换器到converters中  converters.add(convert);  }  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SpringApplications.**class**, args);  }  } |

如果出现乱码，乱码解决：把springboot的response编码设置为utf-8这个功能开启就行

spring.http.encoding.force=true;

第二种：注入方式，在启动类中添加方法。

|  |
| --- |
| @Bean  **public** HttpMessageConverters fastJsonMessageConverter() {  // 创建fastJson的消息转换器  FastJsonHttpMessageConverter convert = **new** FastJsonHttpMessageConverter();  // 创建Fastjson的配置对象  FastJsonConfig config = **new** FastJsonConfig();  // 对Json数据进行格式化  config.setSerializerFeatures(SerializerFeature.***PrettyFormat***);  // 将配置和转换器关联起来  convert.setFastJsonConfig(config);  HttpMessageConverter<?> con = convert;  **return** **new** HttpMessageConverters(con);  } |

**五、springBoot项目demo5**

1. 自定义拦截器

有时候我们需要自己配置SpringMVC而不是采用默认配置，比如增加一个拦截器，这个时候就得通过继承WebMvcConfigurerAdapter然后重写父类中的方法进行扩展。

|  |
| --- |
| 拦截器的定义  **package** cn.huangwei.interceptor;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.web.servlet.HandlerInterceptor;  **import** org.springframework.web.servlet.ModelAndView;  **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry;  **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  **@Configuration//声明这是一个配置**  **public** **class** MyInterceptor **extends** WebMvcConfigurerAdapter{  @Override  **public** **void** addInterceptors(InterceptorRegistry registry) {    HandlerInterceptor inte = **new** HandlerInterceptor() {  @Override  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  System.***out***.println("自定义拦截器。。。。pre");  **return** **true**;  }  @Override  **public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  ModelAndView modelAndView) **throws** Exception {  System.***out***.println("自定义拦截器。。。。post");    }  @Override  **public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,  Exception ex) **throws** Exception {  // **TODO** Auto-generated method stub  System.***out***.println("自定义拦截器。。。。after");  }    };  **registry.addInterceptor(inte).addPathPatterns("/\*\*");**  }    }  除此之外，还需在启动类中进行配置，扫描到拦截器  @SpringBootApplication(scanBasePackages = {"cn.huangwei.controller", "cn.huangwei.interceptor"}) |

2. 定义全局异常处理器

创建一个异常处理类，如下：

|  |
| --- |
| **package** cn.huangwei.controller;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.springframework.web.bind.annotation.ControllerAdvice;  **import** org.springframework.web.bind.annotation.ExceptionHandler;  **import** org.springframework.web.bind.annotation.ResponseBody;  @ControllerAdvice  **public** **class** GlobalExceptionHandler {    @ResponseBody  @ExceptionHandler(Exception.**class**)  **public** Map<String, Object> handleException(Exception ex){  Map<String, Object> map = **new** HashMap<>();  map.put("errorcode", ex.toString());  **return** map;  }  }  TestExceptionController  **package** cn.huangwei.controller;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  @Controller  **public** **class** TestExceptionController {    @RequestMapping("/show")  @ResponseBody  **public** String show() {  **int** a = 5 /0;  **return** "show";  }  } |

  
3. 异步调用

在项目中，当访问其他接口较慢或者耗时任务时，不想程序一直卡在耗时任务上，想程序能够并行运行，我们可以使用多线程来并行处理任务。

SpringBoot提供了异步处理方式@Async

|  |
| --- |
| AsyncService及其实现类  **package** cn.huangwei.service;  **import** java.util.concurrent.Future;  **public** **interface** AsyncService {      Future<String> doTask1() **throws** Exception;  Future<String> doTask2() **throws** Exception;  Future<String> doTask3() **throws** Exception;  }  ----------------------------------------------------------------------------------------------------------------------------------------------------  **package** cn.huangwei.service.impl;  **import** java.util.Random;  **import** java.util.concurrent.Future;  **import** org.springframework.scheduling.annotation.Async;  **import** org.springframework.scheduling.annotation.AsyncResult;  **import** org.springframework.stereotype.Service;  **import** cn.huangwei.service.AsyncService;  **@Service**  **public** **class** AsyncServiceImpl **implements** AsyncService {  **private** **static** Random *random*= **new** Random();    @Async  @Override  **public** Future<String> doTask1() **throws** Exception {  System.***out***.println("任务一开始执行");  **long** start = System.*currentTimeMillis*();  Thread.*sleep*(*random*.nextInt(10000));  **long** end = System.*currentTimeMillis*();  System.***out***.println("任务一结束，耗时：" + (end - start) + "毫秒");    **return** **new** AsyncResult<String>("任务一结束");  }  @Async  @Override  **public** Future<String> doTask2() **throws** Exception {  System.***out***.println("任务二开始执行");  **long** start = System.*currentTimeMillis*();  Thread.*sleep*(*random*.nextInt(10000));  **long** end = System.*currentTimeMillis*();  System.***out***.println("任务二结束，耗时：" + (end - start) + "毫秒");    **return** **new** AsyncResult<String>("任务二结束");  }  @Async  @Override  **public** Future<String> doTask3() **throws** Exception {  System.***out***.println("任务三开始执行");  **long** start = System.*currentTimeMillis*();  Thread.*sleep*(*random*.nextInt(10000));  **long** end = System.*currentTimeMillis*();  System.***out***.println("任务三结束，耗时：" + (end - start) + "毫秒");    **return** **new** AsyncResult<String>("任务三结束");  }  }  ----------------------------------------------------------------------------------------------------------------------------------------------------  Controller  **package** cn.huangwei.controller;  **import** java.util.concurrent.Future;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.ResponseBody;  **import** cn.huangwei.service.AsyncService;  @Controller  **public** **class** TestAsyncController {  **@Autowired**  **private AsyncService asyncService;**    @RequestMapping("/async")  @ResponseBody  **public** String asyncTest() **throws** Exception {  **long** start = System.*currentTimeMillis*();    Future<String> task1 = asyncService.doTask1();  Future<String> task2 = asyncService.doTask2();  Future<String> task3 = asyncService.doTask3();    **while**(**true**) {  **if**(task1.isDone() && task2.isDone() && task3.isDone()) {  **break**;  }  Thread.*sleep*(1000);  }    **long** end = System.*currentTimeMillis*();  **return** "全部执行完成，耗时" + (end - start) + "毫秒";  }  }  启动类配置  结果 |

**六． springboot之demo6**