**Microservice\_Springboot**

## 一、简介

**1 Spring 官方网站本身使用Spring 框架开发，随着功能以及业务逻辑的日益复杂，**应用伴随着大量的XML配置文件以及复杂的Bean依赖关系。随着Spring 3.0的发布，Spring IO团队主键开始摆脱XML配置文件，并且在开发过程中大量使用“约定优先配置”（convention over configuration）的**思想来摆脱Spring框架中各种复杂的配置**，衍生了Java Config。

**2 Spring Boot正是在这样的一个背景下被抽象出来的开发框架**，它本身并不提供Spring框架的核心特性以及扩展功能，只是用于快速、敏捷地开发新一代基于Spring框架的应用程序。也就是说，它并不是用来替代Spring的解决方案，而是和Spring框架紧密结合用于提升Spring开发者体验的工具。同时它集成了大量常用的第三方库配置（例如Jackson, JDBC, Mongo, Redis, Mail等等），Spring Boot应用中这些第三方库几乎可以零配置的开箱即用（out-of-the-box），大部分的Spring Boot应用都只需要非常少量的配置代码，开发者能够更加专注于业务逻辑。

**3 Spring Boot不生成代码，且完全不需要XML配置。其主要目标如下：**

- 为所有的Spring开发工作提供一个更快、更广泛的入门经验。

- 开箱即用，你也可以通过修改默认值来快速满足你的项目的需求。

- 提供了一系列大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。

**4 为什么用springboot**

创建独立的 Spring 应用程序

嵌入的 Tomcat，无需部署 WAR 文件

简化 Maven 配置

自动配置 Spring

提供生产就绪型功能，如指标，健康检查和外部配置

开箱即用，没有代码生成，也无需 XML 配置。

**5 特性理解**

为基于 Spring 的开发提供更快的入门体验

开箱即用，没有代码生成，也无需 XML 配置。同时也可以修改默认值来满足特定的需求。

提供了一些大型项目中常见的非功能特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。

Spring Boot 并不是对 Spring 功能上的增强，而是提供了一种快速使用 Spring 的方式。

**6 Configuration**

从Spring3.0，@Configuration用于定义配置类，可替换xml配置文件，被注解的类内部包含有一个或多个被@Bean注解的方法，

这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义，初始化Spring容器。

package com.dxz.demo.configuration;

import org.springframework.context.annotation.Configuration;

@Configuration

public class TestConfiguration

{

public TestConfiguration()

{

System.out.println("TestConfiguration容器启动初始化。。。");

}

}

等价于：

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

<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context" xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:util="http://www.springframework.org/schema/util" xmlns:task="http://www.springframework.org/schema/task" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.0.xsd

http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc-4.0.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-4.0.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-4.0.xsd

http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-4.0.xsd

http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-task-4.0.xsd" default-lazy-init="false">

</beans>

**（1）@Configuration启动容器+@Bean注册Bean，@Bean下管理bean的生命周期**

package com.dxz.demo.configuration;

public class TestBean {

private String username;

private String url;

private String password;

public void sayHello() {

System.out.println("TestBean sayHello...");

}

public String toString() {

return "username:" + this.username + ",url:" + this.url + ",password:" + this.password;

}

public void start() {

System.out.println("TestBean 初始化。。。");

}

public void cleanUp() {

System.out.println("TestBean 销毁。。。");

}

}

package com.dxz.demo.configuration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

@Configuration

public class TestConfiguration {

public TestConfiguration() {

System.out.println("TestConfiguration容器启动初始化。。。");

}

@Bean(name="testBean",initMethod="start",destroyMethod="cleanUp")

@Scope("prototype")

public TestBean testBean() {

return new TestBean();

}

}

**（2）@Configuration启动容器+@Bean注册Bean，@Bean下管理bean的生命周期**

package com.dxz.demo.configuration;

import org.springframework.stereotype.Component;

@Component

public class TestBean {

private String username;

private String url;

private String password;

public void sayHello() {

System.out.println("TestBean sayHello...");

}

public String toString() {

return "username:" + this.username + ",url:" + this.url + ",password:" + this.password;

}

public void start() {

System.out.println("TestBean 初始化。。。");

}

public void cleanUp() {

System.out.println("TestBean 销毁。。。");

}

}

package com.dxz.demo.configuration;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Scope;

@Configuration

@ComponentScan(basePackages = "com.dxz.demo.configuration")

public class TestConfiguration {

public TestConfiguration() {

System.out.println("TestConfiguration容器启动初始化。。。");

}

}

## 二、Spring Boot Controller

**1 Controller注解**

Controller中使用 @RestController 注解，该注解是Spring 4.0引入的。查看源码可知其包含了 **@Controller 和 @ResponseBody 注解。可以理解为 @Controller的增强版。专门为响应内容式的 Controller 而设计的，可以直接响应对象为JSON。**

而 @Controller 用来响应页面，spring-boot 支持多种模版引擎包括：

1，FreeMarker

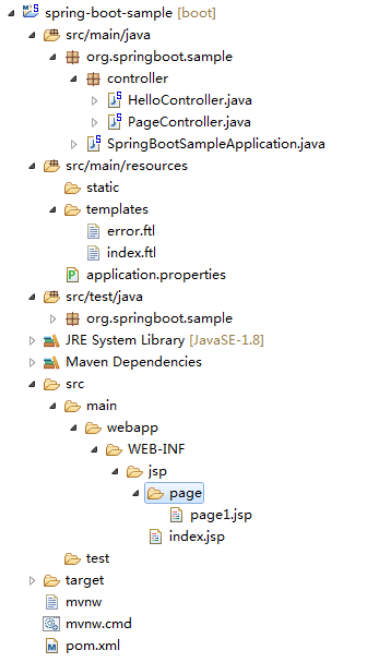
2，Groovy

3，Thymeleaf （Spring 官网使用这个）

4，Velocity

5，JSP 貌似Spring Boot官方不推荐

**2 关于Controller 方法可以接收参数使**用@RequestBody、@RequestParam、@ModelAttribute、JSONObject、HttpEntity 等方式，皆与Spring的使用一样，这里不做赘述。下面我们来说一下如何使用 @Controller 实现响应JSP页面（与SpringMVC也是一样使用）



**3 创建PageController**

package org.springboot.sample.controller;

import java.util.Date;

import java.util.Map;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class PageController {

// 从 application.properties 中读取配置，如取不到默认值为Hello Shanhy

@Value("${application.hell:Hello Shanhy}")

private String hello = "Hello Shanhy";

@RequestMapping(value = {"/","/index"})

public String index(Map<String, Object> model){

// 直接返回字符串，框架默认会去 spring.view.prefix 目录下的 （index拼接spring.view.suffix）页面

// 本例为 /WEB-INF/jsp/index.jsp

model.put("time", new Date());

model.put("message", this.hello);

return "index";

}

@RequestMapping("/page1")

public ModelAndView page1(){

// 页面位置 /WEB-INF/jsp/page/page.jsp

ModelAndView mav = new ModelAndView("page/page1");

mav.addObject("content", hello);

return mav;

}

@RequestMapping("/page2")

public String page2(Model model){

// 页面位置 /WEB-INF/jsp/page/page.jsp

model.addAttribute("content", hello + "（第二种）");

return "page/page1";

}

}

**4 pom.xml添加依赖**

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

**5 添加src/main/resources/application.properties内容：**

# 页面默认前缀目录

spring.mvc.view.prefix=/WEB-INF/jsp/

# 响应页面默认后缀

spring.mvc.view.suffix=.jsp

# 自定义属性，可以在Controller中读取

application.hello=Hello Shanhy

**6 在 src/main 下面创建 webapp/WEB-INF/jsp 目录用来存放我们的jsp页面。**

index.jsp

<%@ page language="java" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Spring Boot Sample</title>

</head>

<body>

Time: ${time}

<br>

Message: ${message}

</body>

</html>

**7 page1.jsp**

<%@ page language="java" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>

<head>

<title>Spring Boot Sample</title>

</head>

<body>

<h1>${content }</h1>

</body>

</html>

要想让spring-boot支持JSP，需要将项目打成war包。

我们做最后一点修改，修改pom.xml文件，将 jar 中的 jar 修改为 war

然后启动spring-boot服务。

访问页面查看结果：

http://localhost:8080

http://localhost:8080/page1

http://localhost:8080/page2

## 三、Spring Boot Servlet

**1 Web开发使用 Controller 基本上可以完成大部分需求**，**但是我们还可能会用到 Servlet、Filter、Listener、Interceptor 等等。**

当使用Spring-Boot时，嵌入式Servlet容器通过扫描注解的方式注册Servlet、Filter和Servlet规范的所有监听器（如HttpSessionListener监听器）。Spring boot 的主 Servlet 为 DispatcherServlet，其默认的url-pattern为“/”。也许我们在应用中还需要定义更多的Servlet，该如何使用SpringBoot来完成呢？

**2 在spring boot中添加自己的Servlet有两种方法**，代码注册Servlet和注解自动注册（Filter和Listener也是如此）。

一、代码注册通过ServletRegistrationBean、 FilterRegistrationBean 和 ServletListenerRegistrationBean 获得控制。也可以通过实现 ServletContextInitializer 接口直接注册。

二、在 SpringBootApplication 上使用@ServletComponentScan 注解后，Servlet、Filter、Listener 可以直接通过 @WebServlet、@WebFilter、@WebListener 注解自动注册，无需其他代码。

**3 通过代码注册Servlet示例代码：SpringBootSampleApplication.java**

package org.springboot.sample;

import org.springboot.sample.servlet.MyServlet;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.context.embedded.ServletRegistrationBean;

import org.springframework.boot.web.servlet.ServletComponentScan;

import org.springframework.context.annotation.Bean;

import org.springframework.web.servlet.DispatcherServlet;

@SpringBootApplication

public class SpringBootSampleApplication {

@Bean

public ServletRegistrationBean servletRegistrationBean() {

return new ServletRegistrationBean(new MyServlet(), "/xs/\*");// ServletName默认值为首字母小写，即myServlet

}

public static void main(String[] args) {

SpringApplication.run(SpringBootSampleApplication.class, args);

}

}

**4 MyServlet.java**

package org.springboot.sample.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

//@WebServlet(urlPatterns="/xs/\*", description="Servlet的说明")

public class MyServlet extends HttpServlet{

private static final long serialVersionUID = -8685285401859800066L;

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

System.out.println(">>>>>>>>>>doGet()<<<<<<<<<<<");

doPost(req, resp);

}

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

System.out.println(">>>>>>>>>>doPost()<<<<<<<<<<<");

resp.setContentType("text/html");

PrintWriter out = resp.getWriter();

out.println("<html>");

out.println("<head>");

out.println("<title>Hello World</title>");

out.println("</head>");

out.println("<body>");

out.println("<h1>大家好，我的名字叫Servlet</h1>");

out.println("</body>");

out.println("</html>");

}

}

**5 使用注解注册Servlet示例代码 SpringBootSampleApplication.java**

package org.springboot.sample;

import org.springboot.sample.servlet.MyServlet;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.context.embedded.ServletRegistrationBean;

import org.springframework.boot.web.servlet.ServletComponentScan;

import org.springframework.context.annotation.Bean;

import org.springframework.web.servlet.DispatcherServlet;

@SpringBootApplication

@ServletComponentScan

public class SpringBootSampleApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootSampleApplication.class, args);

}

}

**6 MyServlet2.java**

package org.springboot.sample.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns="/xs/myservlet", description="Servlet的说明") // 不指定name的情况下，name默认值为类全路径，即org.springboot.sample.servlet.MyServlet2

public class MyServlet2 extends HttpServlet{

private static final long serialVersionUID = -8685285401859800066L;

@Override

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

System.out.println(">>>>>>>>>>doGet2()<<<<<<<<<<<");

doPost(req, resp);

}

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

System.out.println(">>>>>>>>>>doPost2()<<<<<<<<<<<");

resp.setContentType("text/html");

PrintWriter out = resp.getWriter();

out.println("<html>");

out.println("<head>");

out.println("<title>Hello World</title>");

out.println("</head>");

out.println("<body>");

out.println("<h1>大家好，我的名字叫Servlet2</h1>");

out.println("</body>");

out.println("</html>");

}

}

**7 使用 @WebServlet 注解，其中可以设置一些属性。**

有个问题：DispatcherServlet 默认拦截“/”，MyServlet 拦截“/xs/\*”，MyServlet2 拦截“/xs/myservlet”，那么在我们访问 http://localhost:8080/xs/myservlet 的时候系统会怎么处理呢？

**如果访问 http://localhost:8080/xs/abc 的时候又是什么结果呢？其结果是“匹配的优先级是从精确到模糊，复合条件的Servlet并不会都执行**”

既然系统DispatcherServlet 默认拦截“/”，那么我们是否能做修改呢，答案是肯定的，我们在SpringBootSampleApplication中添加代码：

@Bean

public ServletRegistrationBean dispatcherRegistration(DispatcherServlet dispatcherServlet) {

ServletRegistrationBean registration = new ServletRegistrationBean(dispatcherServlet);

registration.getUrlMappings().clear();

registration.addUrlMappings("\*.do");

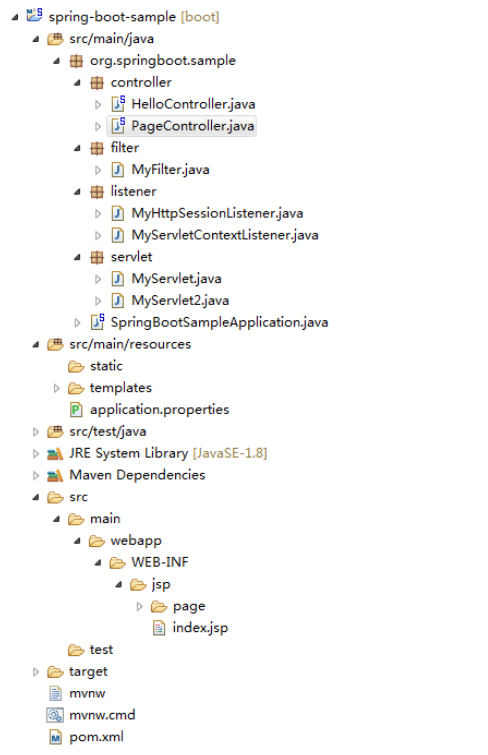
registration.addUrlMappings("\*.json");

return registration;

}

## 过滤器（Filter）和 监听器（Listener）

**1 过滤器（Filter）和 监听器（Listener）的注册方法和 Servlet 一样，直接使用@WebFilter和@WebListener的方式，完成一个Filter 和一个 Listener。**



**2 过滤器（Filter）文件 MyFilter.java**

package org.springboot.sample.filter;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import javax.servlet.annotation.WebFilter;

@WebFilter(filterName="myFilter",urlPatterns="/\*")

public class MyFilter implements Filter {

@Override

public void destroy() {

System.out.println("过滤器销毁");

}

@Override

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain) throws IOException, ServletException {

System.out.println("执行过滤操作");

chain.doFilter(request, response);

}

@Override

public void init(FilterConfig config) throws ServletException {

System.out.println("过滤器初始化");

}

}

**3 ServletContext监听器（Listener）文件 MyServletContextListener.java**

package org.springboot.sample.listener;

import javax.servlet.ServletContextEvent;

import javax.servlet.ServletContextListener;

import javax.servlet.annotation.WebListener;

@WebListener

public class MyServletContextListener implements ServletContextListener {

@Override

public void contextInitialized(ServletContextEvent sce) {

System.out.println("ServletContex初始化");

System.out.println(sce.getServletContext().getServerInfo());

}

@Override

public void contextDestroyed(ServletContextEvent sce) {

System.out.println("ServletContex销毁");

}

}

**4 ServletContext监听器（Listener）文件 MyHttpSessionListener.java**

package org.springboot.sample.listener;

import javax.servlet.annotation.WebListener;

import javax.servlet.http.HttpSessionEvent;

import javax.servlet.http.HttpSessionListener;

@WebListener

public class MyHttpSessionListener implements HttpSessionListener {

@Override

public void sessionCreated(HttpSessionEvent se) {

System.out.println("Session 被创建");

}

@Override

public void sessionDestroyed(HttpSessionEvent se) {

System.out.println("ServletContex初始化");

}

}

**注意不要忘记在 SpringBootSampleApplication.java 上添加 @ServletComponentScan 注解。**

在启动的过程中我们会看到输出：

ServletContex初始化

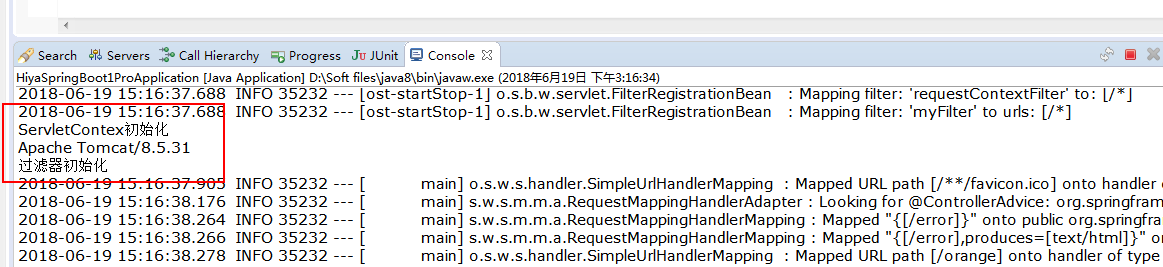
Apache Tomcat/8.0.30

过滤器初始化

服务启动后，随便访问一个页面，会看到输出：

执行过滤操作

Session 被创建



## 五、Spring boot拦截器

1 Web开发中，我们除了使用 Filter 来过滤请web求外，还可以使用Spring提供的**HandlerInterceptor**（拦截器）。HandlerInterceptor 的功能跟过滤器类似，但是提供更精细的的控制能力：**在request被响应之前、request被响应之后、视图渲染之前以及request全部结束之后**。我们不能通过拦截器修改request内容，但是可以通过抛出异常（或者返回false）来暂停request的执行。

实现 UserRoleAuthorizationInterceptor 的拦截器有：

ConversionServiceExposingInterceptor

CorsInterceptor

LocaleChangeInterceptor

PathExposingHandlerInterceptor

ResourceUrlProviderExposingInterceptor

ThemeChangeInterceptor

UriTemplateVariablesHandlerInterceptor

UserRoleAuthorizationInterceptor

其中 LocaleChangeInterceptor 和 ThemeChangeInterceptor 比较常用。配置拦截器也很简单，Spring 为什么提供了基础类WebMvcConfigurerAdapter ，我们只需要重写 addInterceptors 方法添加注册拦截器。

**2 自定义拦截器3步**

1、创建我们自己的拦截器类并实现 HandlerInterceptor 接口。

2、创建一个Java类继承WebMvcConfigurerAdapter，并重写 addInterceptors 方法。

2、实例化我们自定义的拦截器，然后将对像手动添加到拦截器链中（在addInterceptors方法中添加）。

**3 MyInterceptor1.java**

package org.springboot.sample.interceptor;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

public class MyInterceptor1 implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

System.out.println(">>>MyInterceptor1>>>>>>>在请求处理之前进行调用（Controller方法调用之前）");

return true;// 只有返回true才会继续向下执行，返回false取消当前请求

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

System.out.println(">>>MyInterceptor1>>>>>>>请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之后）");

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

throws Exception {

System.out.println(">>>MyInterceptor1>>>>>>>在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对应的视图之后执行（主要是用于进行资源清理工作）");

}

}

**4 MyInterceptor2.java**

package org.springboot.sample.interceptor;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

public class MyInterceptor2 implements HandlerInterceptor {

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

System.out.println(">>>MyInterceptor2>>>>>>>在请求处理之前进行调用（Controller方法调用之前）");

return true;// 只有返回true才会继续向下执行，返回false取消当前请求

}

@Override

public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

System.out.println(">>>MyInterceptor2>>>>>>>请求处理之后进行调用，但是在视图被渲染之前（Controller方法调用之后）");

}

@Override

public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

throws Exception {

System.out.println(">>>MyInterceptor2>>>>>>>在整个请求结束之后被调用，也就是在DispatcherServlet 渲染了对应的视图之后执行（主要是用于进行资源清理工作）");

}

}

**5 MyWebAppConfigurer.java**

package org.springboot.sample.config;

import org.springboot.sample.interceptor.MyInterceptor1;

import org.springboot.sample.interceptor.MyInterceptor2;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.InterceptorRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration

public class MyWebAppConfigurer

extends WebMvcConfigurerAdapter {

@Override

public void addInterceptors(InterceptorRegistry registry) {

// 多个拦截器组成一个拦截器链

// addPathPatterns 用于添加拦截规则

// excludePathPatterns 用户排除拦截

registry.addInterceptor(new MyInterceptor1()).addPathPatterns("/\*\*");

registry.addInterceptor(new MyInterceptor2()).addPathPatterns("/\*\*");

super.addInterceptors(registry);

}

}

**6 在SpringBoot2.0及Spring 5.0 WebMvcConfigurerAdapter已被废弃，**

**目前找到解决方案就有三种**

**1 直接实现WebMvcConfigurer （官方推荐）**

@Configuration

public class WebMvcConfg implements WebMvcConfigurer {

}

**2 直接继承WebMvcConfigurationSupport**

@Configuration

public class WebMvcConfg extends WebMvcConfigurationSupport {

}

**3 配置文件**

<mvc:interceptors>

<!-- 跨域拦截器 -->

<mvc:interceptor>

<mvc:mapping path=*"/\*\*"*/>

<bean class=*"com.yunnex.ops.erp.modules.sys.interceptor.CrossDomainInterceptor"*></bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path=*"${adminPath}/\*\*"*/>

<mvc:exclude-mapping path=*"${adminPath}/"*/>

<mvc:exclude-mapping path=*"${adminPath}/login"*/>

<mvc:exclude-mapping path=*"${adminPath}/sys/menu/tree"*/>

<mvc:exclude-mapping path=*"${adminPath}/sys/menu/treeData"*/>

<mvc:exclude-mapping path=*"${adminPath}/oa/oaNotify/self/count"*/>

<bean class=*"com.yunnex.ops.erp.modules.sys.interceptor.LogInterceptor"*/>

</mvc:interceptor>

<!-- 手机视图拦截器 -->

<mvc:interceptor>

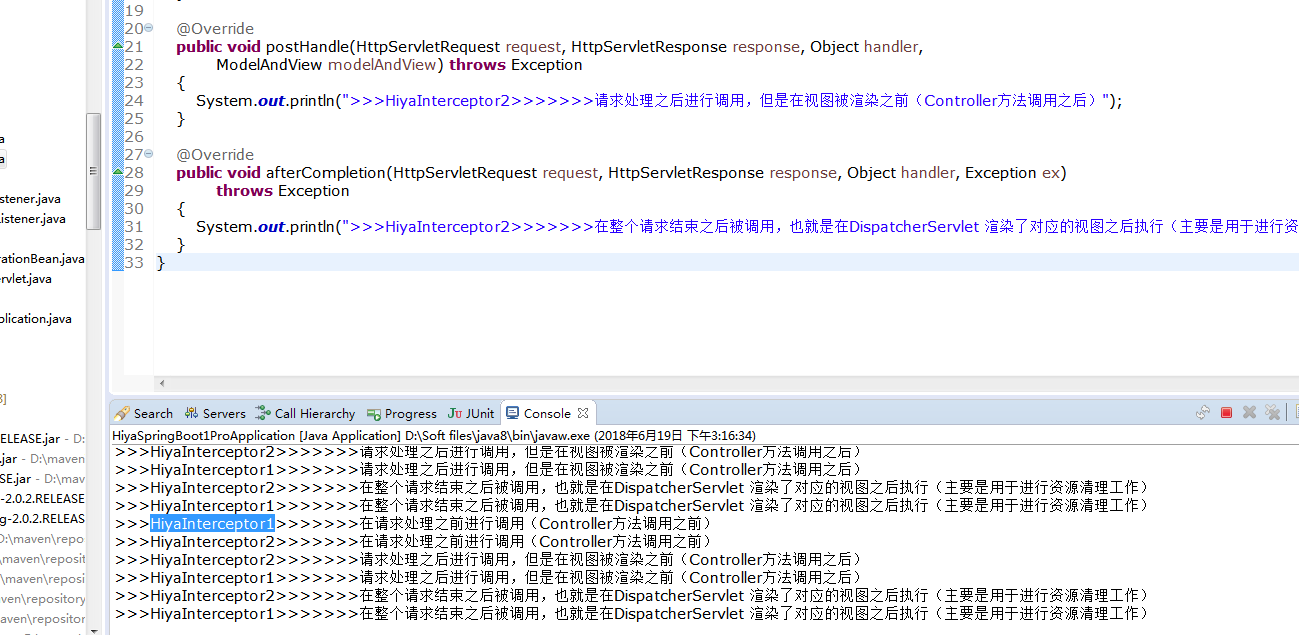
<mvc:mapping path=*"/\*\*"*/>

<bean class=*"com.yunnex.ops.erp.modules.sys.interceptor.MobileInterceptor"*/>

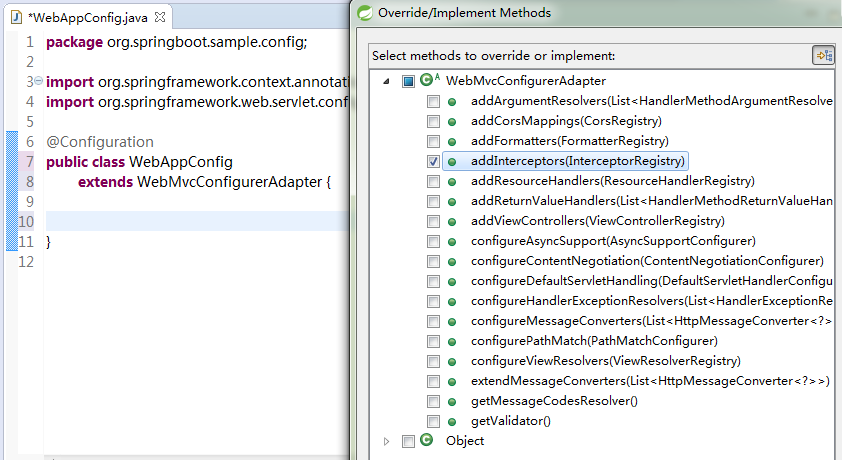
</mvc:interceptor>

</mvc:interceptors>

**7 然后在浏览器输入地址： http://localhost:8080/index 后，控制台的输出为：**



**最后强调一点：只有经过DispatcherServlet 的请求，才会走拦截器链，我们自定义的Servlet 请求是不会被拦截的，比如我们自定义的Servlet地址 [http://localhost:8080/xs/myservlet](http://localhost:8080/xs/myservlet" \t "https://blog.csdn.net/catoop/article/details/_blank) 是不会被拦截器拦截的。并且不管是属于哪个Servlet 只要复合过滤器的过滤规则，过滤器都会拦截。上面用到的 WebMvcConfigurerAdapter 并非只是注册添加拦截器使用，其顾名思义是做Web配置用的，它还可以有很多其他作用，通过下面截图便可以大概了解，具体每个方法都是干什么用的，留给大家自己研究（其实都大同小异也很简单）。**



## 六、Spring boot静态资源处理

1 Spring Boot 默认为我们提供了静态资源处理，使用 WebMvcAutoConfiguration 中的配置各种属性。使用Spring Boot的默认配置方式，如果需要特殊处理的再通过配置进行修改。

完全控制WebMVC，就需要在@Configuration注解的配置类上增加**@EnableWebMvc**（@SpringBootApplication注解的程序入口类已经包含@Configuration），增加该注解以后WebMvcAutoConfiguration中配置就不会生效，自己来配置需要的每一项。这种情况下的配置还是要多看一下WebMvcAutoConfiguration类。快速使用Spring Boot，并不想过多的自己再重新配置。本文还是主要针对Spring Boot的默认处理方式，部分配置在application 配置文件中（.properties 或 .yml）

**2 默认资源映射，启动应用打印**

2016-01-08 09:29:30.362 INFO 24932 --- [ main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/\*\*] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

2016-01-08 09:29:30.362 INFO 24932 --- [ main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/\*\*] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

2016-01-08 09:29:30.437 INFO 24932 --- [ main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/\*\*/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]

**其中默认配置的 /\*\* 映射到 /static （或/public、/resources、/META-INF/resources）**

**其中默认配置的 /webjars/\*\* 映射到 classpath:/META-INF/resources/webjars/**

**PS：上面的 static、public、resources 等目录都在 classpath: 下面（如 src/main/resources/static）。**

**3 Spring Boot 默认将 /\*\* 所有访问映射到以下目录：**

classpath:/static

classpath:/public

classpath:/resources

classpath:/META-INF/resources

**4 如：在resources目录下新建 public、resources、static 三个目录，并分别放入 a.jpg b.jpg c.jpg 图片**

目录

浏览器分别访问：

http://localhost:8080/a.jpg

http://localhost:8080/b.jpg

http://localhost:8080/c.jpg

均能正常访问相应的图片资源。那么说明，Spring Boot 默认会挨个从 public resources static 里面找是否存在相应的资源，如果有则直接返回。

自定义静态资源映射

在实际开发中，可能需要自定义静态资源访问路径，那么可以继承WebMvcConfigurerAdapter来实现。

**5 第一种方式：静态资源配置类**

package com.sam.demo.conf;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration

public class WebMvcConfig extends WebMvcConfigurerAdapter {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

//将所有/static/\*\* 访问都映射到classpath:/static/ 目录下

registry.addResourceHandler("/static/\*\*").addResourceLocations("classpath:/static/");

}

}

重启项目，访问：http://localhost:8080/static/c.jpg 能正常访问static目录下的c.jpg图片资源。

**6 第二种方式：在application.properties配置**

在application.properties中添加配置：

spring.mvc.static-path-pattern=/static/\*\*

重启项目，访问：http://localhost:8080/static/c.jpg 同样能正常访问static目录下的c.jpg图片资源。

**注意：通过spring.mvc.static-path-pattern这种方式配置，会使Spring Boot的默认配置失效，也就是说，/public /resources 等默认配置不能使用。配置中配置了静态模式为/static/，就只能通过/static/来访问。**

**七、Spring boot启动加载数据**

1 实际应用中，我们会有在项目服务启动的时候就去加载一些数据或做一些事情这样的需求。为了解决这样的问题，Spring Boot 为我们提供了一个方法，通过实现接口 **CommandLineRunner** 来实现。Spring Boot应用程序在启动后，会遍历CommandLineRunner接口的实例并运行它们的run方法。也可以利用**@Order**注解（或者实现Order接口）来规定所有CommandLineRunner实例的运行顺序。

package org.springboot.sample.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

@Component

@Order(value=2) //@Order 注解来定义执行顺序

public class MyStartupRunner1 implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

System.out.println(">>>>>>>>>>>>>>>服务启动执行，执行加载数据等操作 11111111 <<<<<<<<<<<<<");

}

}

package org.springboot.sample.runner;

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

@Component

@Order(value=1)

public class MyStartupRunner2 implements CommandLineRunner {

@Override

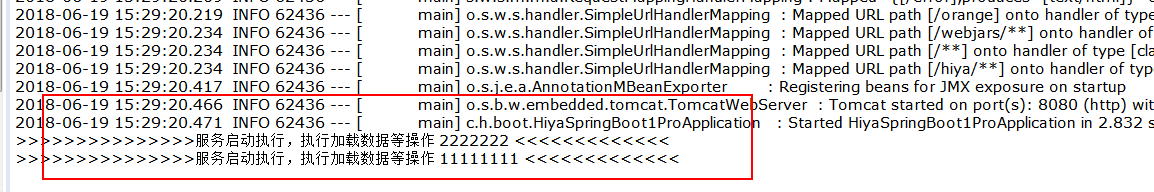
public void run(String... args) throws Exception {

System.out.println(">>>>>>>>>>>>>>>服务启动执行，执行加载数据等操作 22222222 <<<<<<<<<<<<<");

}

}

**启动程序后，控制台输出结果为：**



**八、Spring boot日志管理**

**1 在开发中打印内容，使用 System.out.println() 和 Log4j 应当是人人皆知的方法了。**其实在开发中我们不建议使用 System.out 因为大量的使用 System.out 会增加资源的消耗。而Log4j 更为灵活在性能上也相比 System.out 要高，我们可以配置输出级别，可以指定多个日志文件分别记录不同的日志。使用 System.out 是在当前线程执行的，写入文件也是写入完毕后才继续执行下面的程序。而使用Log工具不但可以控制日志是否输出，怎么输出，它的处理机制也是通知写日志，继续执行后面的代码不必等日志写完。

**推荐使用 SLF4J（Simple Logging Facade For Java）的logback来输出日志，其比log4j 要好，因为他效率更高。**

**2 Spring Boot的日志系统预先定义了一些系统变量：**

PID，当前进程ID

{LOG\_FILE}，Spring Boot配置文件（application.properties|.yml）中logging.file的值

${LOG\_PATH}, Spring Boot配置文件中logging.path的值

同时默认情况下包含另个appender——一个是控制台，一个是文件，分别定义在console-appender.xml和file-appender.xml中。同时对于应用的日志级别也可以通过application.properties进行定义：

logging.level.org.springframework.web=DEBUG

logging.level.org.springboot.sample=TRACE

这样相当于我们在logback.xml 中配置的对应的日志级别。名称以logging.level开头，后面跟要输入日志的包名。

\*如果在 logback.xml 和 application.properties 中定义了相同的配置（如都配置了 org.springframework.web）但是输出级别不同，则实际上 application.properties 的优先级高于 logback.xml \*

我们既然使用了maven来管理项目，我们就可以根据不同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：

注意文件名称不是logback.xml，想使用spring扩展profile支持，要以logback-spring.xml命名

**另一种常见的日志输出到文件，随着应用的运行时间越来越长，日志也会增长的越来越多，将他们输出到同一个文件并非一个好办法。RollingFileAppender用于切分文件日志**：

<appender name="dailyRollingFileAppender" class="ch.qos.logback.core.rolling.RollingFileAppender">

<File>/data/log/app.log</File>

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!-- daily rollover -->

<FileNamePattern>rest-demo.%d{yyyy-MM-dd}.log</FileNamePattern>

<!-- keep 30 days' worth of history -->

<maxHistory>30</maxHistory>

</rollingPolicy>

<encoder>

<Pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{35} - %msg %n</Pattern>

</encoder>

</appender>

其中重要的是rollingPolicy的定义，上例中rest-demo.%d{yyyy-MM-dd}.log定义了日志的切分方式——把每一天的日志归档到一个文件中，

**3 总结,在Spring Boot 中记录日志只需两步：**

1、在 src/main/resources 下面创建logback.xml 文件，并按上面讲述的进行配置。或者使用最简单的方法在 application 配置文件中配置。

2、在Java代码中创建实例，并在需要输出日志的地方使用。

private static final Logger logger = LoggerFactory.getLogger(SpringBootSampleApplication.class);

public void logTest() {

logger.debug("日志输出测试 Debug");

logger.trace("日志输出测试 Trace");

logger.info("日志输出测试 Info");

}

## 九、Spring boot JPA数据库

**1 修改 pom.xml 依赖**

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

**2 在属性配置文件中添加 JPA 相关属性**

spring.jpa.database=

spring.jpa.show-sql=

spring.jpa.properties=

spring.jpa.generate-ddl=

spring.jpa.open-in-view=

spring.jpa.database-platform=

spring.jpa.hibernate.ddl-auto=

spring.data.jpa.repositories.enabled=

spring.jpa.hibernate.naming-strategy=

package org.springboot.sample.dao;

import java.util.List;

import javax.transaction.Transactional;

import org.springboot.sample.entity.Score;

import org.springframework.data.jpa.repository.Modifying;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.CrudRepository;

import org.springframework.data.repository.query.Param;

public interface IScoreDao extends CrudRepository<Score, Integer> {

@Transactional

@Modifying

@Query("update Score t set t.score = :score where t.id = :id")

int updateScoreById(@Param("score") float score, @Param("id") int id);

@Query("select t from Score t ")

List<Score> getList();

}

**注意，如果你其中使用了修改、新增、删除操作，则必须要在接口上面或者对应的方法上面添加 @Transactional 注解，否则会抛出异常。**

**3 实体类 Score.java**

package org.springboot.sample.entity;

import java.io.Serializable;

import java.util.Date;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "score")

public class Score implements Serializable {

private static final long serialVersionUID = 8127035730921338189L;

@Id

@GeneratedValue

private int id;

@Column(nullable = false, name="STUDENTID") // 这里说一下，我使用指定数据库列的时候，使用小写会不起作用，修改为大写便正常了。不知道为何，如果遇到一样问题的可以尝试下。

private int stuId;

@Column(nullable = false, name="SUBJECTNAME")

private String subjectName;

@Column(nullable = false)

private float score;

@Column(nullable = false, name="EXAMTIME")

private Date examTime;

// 省去get、set 方法（占用文章空间）

}

**4 ScoreController.java**

package org.springboot.sample.controller;

import java.util.List;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springboot.sample.dao.IScoreDao;

import org.springboot.sample.entity.Score;

import org.springframework.beans.factory.annotation.Autowired;

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

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/score")

public class ScoreController {

private static final Logger logger = LoggerFactory.getLogger(ScoreController.class);

@Autowired

private IScoreDao scoreService;

@RequestMapping("/scoreList")

public List<Score> getScoreList(){

logger.info("从数据库读取Score集合");

// 测试更新数据库

logger.info("更新的行数：" + scoreService.updateScoreById(88.8f, 2));

scoreService.delete(23);

return scoreService.getList();

}

}

**5 然后在浏览器访问地址：http://localhost:8080/myspringboot/score/scoreList 测试。**

最后要说明的是，Spring 会自动为我们继承CrudRepository接口的接口创建实现类。我们只需要在使用的时候直接使用注解 @Autowired 注入即可（IScoreDao 接口上也没有必要增加 @Component 、 @Repository 等注解）。还有，我这里为了简单起见，直接将操作数据库的在Controller中使用，实际项目中，是不建议这样做的，IScoreDao 的所属角色是数据库持久，我们还应当有 Service（如ScoreService） 来调用 IScoreDao 的方法，然后在Controller 中调用 Service 中的方法。原因是因为，我们数据库访问层，都是接口定义方法，上面注解注入SQL和参数，没有具体的代码逻辑处理。如果我们想在执行SQL之前或之后执行逻辑处理，只能在 Service 中或者Controller（不建议）中。我们严格按照这种方式去做（持久层只与SQL有关，通过接口定义无逻辑处理），这样才是彻彻底底的持久层。越严格的规范制度，在某种程度上来说其实越有利于代码的管理和项目代码的迭代发展。



**6 多数据源**

**# 主数据源，默认的**

spring.datasource.type=com.zaxxer.hikari.HikariDataSource

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

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=123456

**# 更多数据源**

custom.datasource.names=ds1,ds2,ds3

custom.datasource.ds1.type=com.zaxxer.hikari.HikariDataSource

custom.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver

custom.datasource.ds1.url=jdbc:mysql://localhost:3306/test

custom.datasource.ds1.username=root

custom.datasource.ds1.password=123456

custom.datasource.ds2.type=com.zaxxer.hikari.HikariDataSource

custom.datasource.ds2.driver-class-name=com.mysql.jdbc.Driver

custom.datasource.ds2.url=jdbc:mysql://localhost:3306/test

custom.datasource.ds2.username=root

custom.datasource.ds2.password=123456

custom.datasource.ds3.type=com.zaxxer.hikari.HikariDataSource

custom.datasource.ds3.driver-class-name=com.mysql.jdbc.Driver

custom.datasource.ds3.url=jdbc:mysql://localhost:3306/test

custom.datasource.ds3.username=root

custom.datasource.ds3.password=123456

spring.datasource.maximum-pool-size=100

spring.datasource.max-idle=10

spring.datasource.max-wait=10000

spring.datasource.min-idle=5

spring.datasource.initial-size=5

spring.datasource.validation-query=SELECT 1

spring.datasource.test-on-borrow=false

spring.datasource.test-while-idle=true

spring.datasource.time-between-eviction-runs-millis=18800

**十、Spring boot部署**

**1 内嵌 Server 配置**

# 项目contextPath，一般在正式发布版本中，我们不配置

server.context-path=/myspringboot

# 错误页，指定发生错误时，跳转的URL。请查看BasicErrorController源码便知

server.error.path=/error

# 服务端口

server.port=9090

# session最大超时时间(分钟)，默认为30

server.session-timeout=60

# 该服务绑定IP地址，启动服务器时如本机不是该IP地址则抛出异常启动失败，只有特殊需求的情况下才配置

# server.address=192.168.16.11

**2 Tomcat为Spring Boot的默认容器，下面是几个常用配置：**

# tomcat最大线程数，默认为200

server.tomcat.max-threads=800

# tomcat的URI编码

server.tomcat.uri-encoding=UTF-8

# 存放Tomcat的日志、Dump等文件的临时文件夹，默认为系统的tmp文件夹（如：C:\Users\Shanhy\AppData\Local\Temp）

server.tomcat.basedir=H:/springboot-tomcat-tmp

# 打开Tomcat的Access日志，并可以设置日志格式的方法：

#server.tomcat.access-log-enabled=true

#server.tomcat.access-log-pattern=

# accesslog目录，默认在basedir/logs

#server.tomcat.accesslog.directory=

# 日志文件目录

logging.path=H:/springboot-tomcat-tmp

# 日志文件名称，默认为spring.log

logging.file=myapp.log

**3 并加入Jetty容器的依赖**

<dependencies>

<dependency>

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

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

<exclusions>

<exclusion>

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

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

<dependencies>

**4 打包方法**

CMD进入项目目录，使用 **mvn clean package** 命令打包，以我的项目工程为例：

E:\spring-boot-sample>mvn clean package

可以追加参数 -Dmaven.test.skip=true 跳过测试。

打包后的文件存放于项目下的target目录中，如：spring-boot-sample-0.0.1-SNAPSHOT.jar

如果pom配置的是war包，则为spring-boot-sample-0.0.1-SNAPSHOT.war

**5 部署到JavaEE容器**

修改启动类，继承 SpringBootServletInitializer 并重写 configure 方法

public class SpringBootSampleApplication extends SpringBootServletInitializer{

private static final Logger logger = LoggerFactory.getLogger(SpringBootSampleApplication.class);

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {

return builder.sources(this.getClass());

}

}

**6 修改pom文件中jar 为 war**

<packaging>war</packaging>

**7 修改pom，排除tomcat插件,打包部署到容器 使用命令 mvn clean package 打包后，同一般J2EE项目一样部署到web容器。**

<dependency>

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

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

<exclusions>

<exclusion>

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

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

**8 使用Profile区分环境**

spring boot 可以在 “配置文件”、“Java代码类”、“日志配置” 中来配置profile区分不同环境执行不同的结果

**9 配置文件**

使用配置文件application.yml 和 application.properties 有所区别

以application.properties 为例，通过文件名来区分环境 application-{profile}.properties

application.properties

app.name=MyApp

server.port=8080

spring.profiles.active=dev

application-dev.properties

server.port=8081

application-stg.properties

server.port=8082

在启动程序的时候通过添加 –spring.profiles.active={profile} 来指定具体使用的配置

例如我们执行 java -jar demo.jar –spring.profiles.active=dev 那么上面3个文件中的内容将被如何应用？

Spring Boot 会先加载默认的配置文件，然后使用具体指定的profile中的配置去覆盖默认配置。

app.name 只存在于默认配置文件 application.properties 中，因为指定环境中不存在同样的配置，所以该值不会被覆盖

server.port 默认为8080，但是我们指定了环境后，将会被覆盖。如果指定stg环境，server.port 则为 8082

spring.profiles.active 默认指定dev环境，如果我们在运行时指定 –spring.profiles.active=stg 那么将应用stg环境，最终 server.port 的值为8082

**10 Java类中@Profile注解**

下面2个不同的类实现了同一个接口，@Profile注解指定了具体环境

// 接口定义

public interface SendMessage {

// 发送短信方法定义

public void send();

}

// Dev 环境实现类

@Component

@Profile("dev")

public class DevSendMessage implements SendMessage {

@Override

public void send() {

System.out.println(">>>>>>>>Dev Send()<<<<<<<<");

}

}

// Stg环境实现类

@Component

@Profile("stg")

public class StgSendMessage implements SendMessage {

@Override

public void send() {

System.out.println(">>>>>>>>Stg Send()<<<<<<<<");

}

}

// 启动类

@SpringBootApplication

public class ProfiledemoApplication {

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

private String name;

@Autowired

private SendMessage sendMessage;

@PostConstruct

public void init(){

sendMessage.send();// 会根据profile指定的环境实例化对应的类

}

}

**11 logback-spring.xml也支持有节点来支持区分**

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

<configuration>

<include resource="org/springframework/boot/logging/logback/base.xml" />

<logger name="org.springframework.web" level="INFO"/>

<springProfile name="default">

<logger name="org.springboot.sample" level="TRACE" />

</springProfile>

<springProfile name="dev">

<logger name="org.springboot.sample" level="DEBUG" />

</springProfile>

<springProfile name="staging">

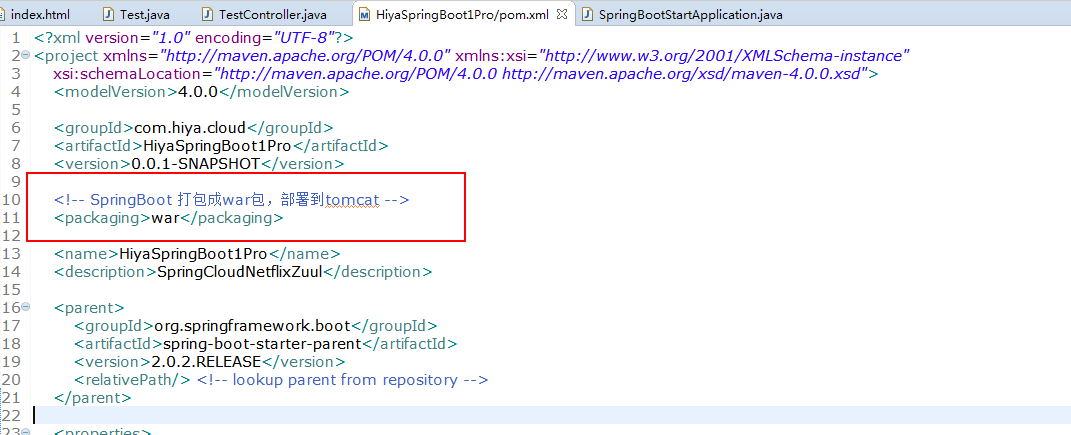
<logger name="org.springboot.sample" level="INFO" />

</springProfile>

</configuration>

12 **SpringBoot 打包成war包，部署到tomcat**

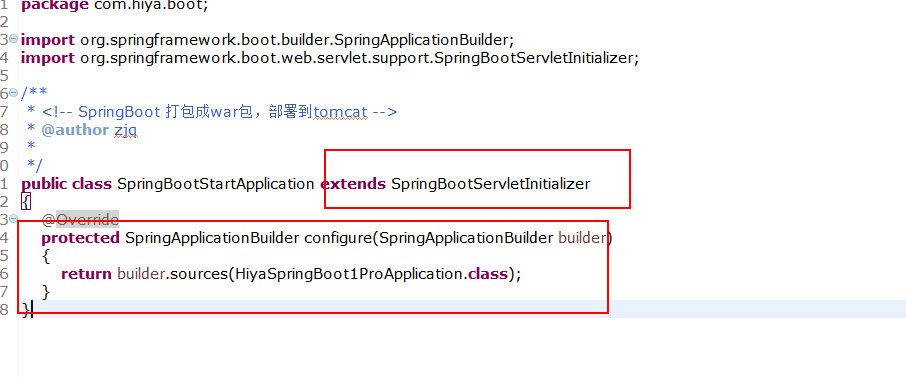
**1 修改 <packaging>war</packaging>**



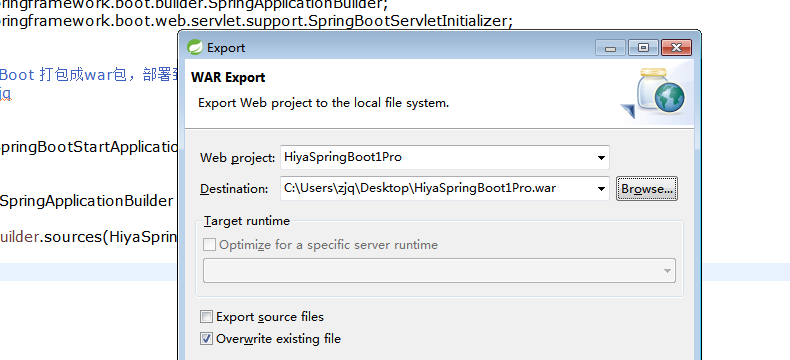
**2修改排除内嵌服务器 spring-boot-starter-web**



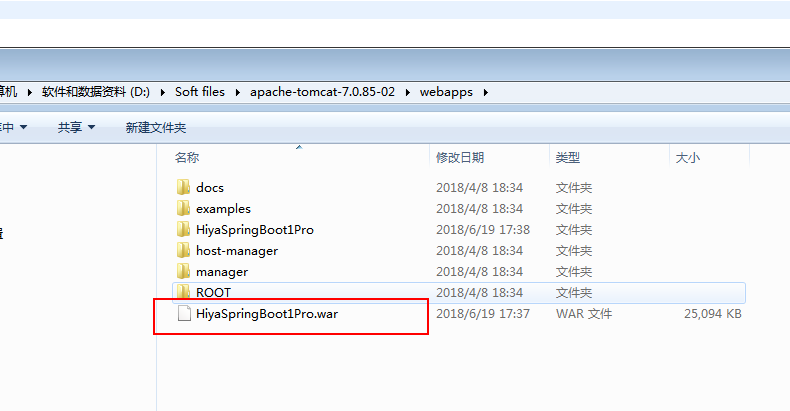
**3 继承SpringBootServletInitializer重新配置**



**4 mvn或者eclipse打包**

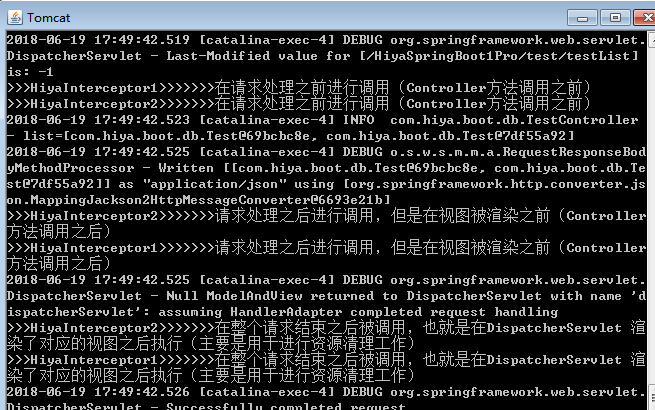


**5 将war包放到部署路径下**



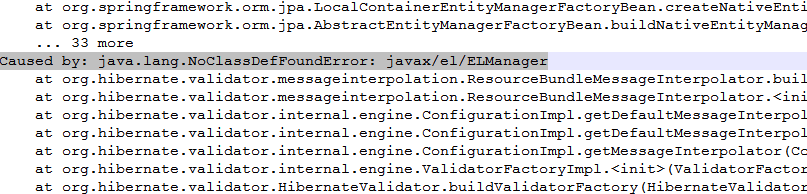
**6 启动服务器**

**（1）命令启动服务器**



**（2）遇到的问题**

**Caused by: java.lang.ClassNotFoundException: javax.el.ELManager**

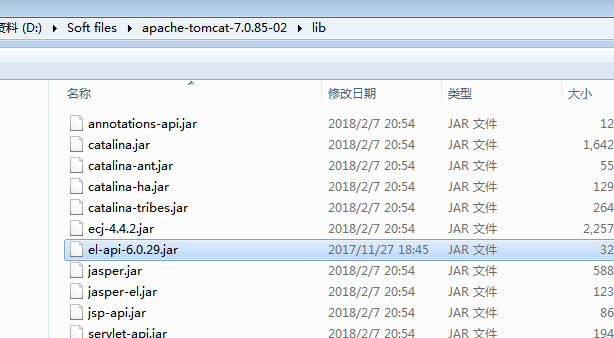


**原因分析：**这个是因为tomcat7中的el-api2.2，有些版本太低，建议升级tomcat到8.0，利用el-api3.0就会解决这个问题 。

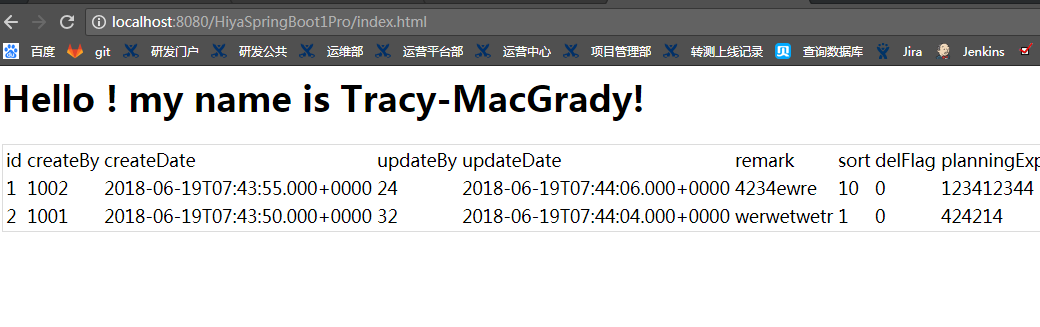
**解决办法：**



**替换**



**7 访问项目**



## 十一、Spring boot总结

**1.强大的解耦，集成能力。**

**2.零配置（简单配置）**

是的springboot完全可以做到零配置，而且这也是它们倡导的。这样的好处是显而易见的，那就是降低了入门的门槛，对于一个新手来说，怎么做配置从来都是一件十分头疼的事，配置越多，对应的学习难度也就越大。那么它是如何做到的呢，下面以比较常用的数据源配置作为比较，以期能够更好理解，

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="mapperLocations" value="" />

<property name="typeAliasesPackage" value="" />

</bean>

<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">

<constructor-arg ref="sqlSessionFactory" />

</bean>

<bean id="myBatisSessionManager" class="com.chemao.loan.common.core.MyBatisSessionManager">

<property name="sessionMap">

<map>

<entry key="default" value-ref="sqlSession" />

</map>

</property>

</bean>

这是一个常见的数据源配置，可以看见我们配置了dataSource,sqlSessionFactory,sqlSession,myBatisSessionManager，事实上我们只配置了dataSource,sqlSessionFactory，另外两个bean是基于前面的配置构造出来的，这么看来后两个配置并不是必须的，那么前两个也可以不配么，目前来看是不行的，因为里面很多是我们自定义的属性值，官方没办法通过默认值去填充，那么是不是没办法了，当然不是，官方没办法默认，但可以约定，比如约定property文件spring.datasource.url这个key对应的属性为数据库连接的url，用户只需要配置这个属性，其他就可以交给官方去处理了，这样也就做到了零配置，或者说简单配置就可以用了。

**3.依赖管理**

对于开发来说，依赖管理是一件很烦人的事，到了后面有了maven，发现一下子好了很多，但仍然有很多不足，比如依赖的jar包冲突，很多惨痛的事故都是由此引发的，显然有人意识到了这一点，springboot就做了很好的处理，它定义了一个大而全的依赖管理，所有主流开源的依赖都包括在内（当然个人或公司私有的不包括），然后统一版本，这样也就解决了冲突问题

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

<artifactId>spring-boot-dependencies</artifactId>

**4.容器集成**

springboot整合了Tomcat，jetty等主流的容器，默认是Tomcat

**5.开闭原则**

支持扩展，不支持修改，springboot提供了最为通用，或者官方建议的一种配置方式，这种配置方式是不可以修改的。但是有时候官方的并不能很好的满足我们的需要，这个时候重写官方的配置，来实现以下自定义的属性。

## Spring Boot Maven Plugin

**1 Spring Boot Maven Plugin插件提供spring boot在maven中的支持。允许你打包可运行的jar包或war包。**

插件提供了几个maven目标和Spring Boot 应用一起工作。总的有：

spring-boot:repackage 请阅读：Spring cloud的Maven插件（一）：repackage目标

spring-boot:run 请阅读：Spring cloud的Maven插件（二）：run目标

spring-boot:start and spring-boot:stop

spring-boot:build-info

repackage：创建一个自动可执行的jar或war文件。它可以替换常规的artifact，或者用一个单独的classifier附属在maven构建的生命周期中。

**2 简单引用**

用了重新打包应用程序，我们需要在pom.xml文件中加一个简单的引用到Spring Boot Maven Plugin插件中，代码如下：repaceage 最简单的结构

1 <build>

2 ...

3 <plugins>

4 ...

5 <plugin>

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

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

8 <version>1.5.7.RELEASE</version>

9 <executions>

10 <execution>

11 <goals>

12 <goal>repackage</goal>

13 </goals>

14 </execution>

15 </executions>

16 </plugin>

17 ...

18 </plugins>

19 ...

20 </build>

**3 剔除不需要的依赖**

这个例子重新打包了一个jar包或war包，这个jar包或war包被构建于maven生命周期的package阶段，包括定义在工程中的任何依赖（包括scope为provided）。如果有一些依赖模块需要被排除掉，可以使用一个exclude的选项。

默认情况下，repackage和run这两个maven目标会包括定义在工程中的任何依赖。一些依赖被要求从可执行jar包中排除。下面有三种方式可以在包运行时排除依赖模块。

（1）方式1：:排除一个具体的maven 模块，通过唯一的groupId和artifactId组合来实现。（如果有必要，可以加入classifier来唯一确认。）

1 <project>

2 ...

3 <build>

4 ...

5 <plugins>

6 ...

7 <plugin>

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

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

10 <version>1.5.6.RELEASE</version>

11 <configuration>

12 <excludes>

13 <exclude>

14 <groupId>com.foo</groupId>

15 <artifactId>bar</artifactId>

16 </exclude>

17 </excludes>

18 </configuration>

19 ...

20 </plugin>

21 ...

22 </plugins>

23 ...

24 </build>

25 ...

26 </project>

（2）方式2：排除和“指定的artifactId”相符的所有maven模块。

1 <project>

2 ...

3 <build>

4 ...

5 <plugins>

6 ...

7 <plugin>

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

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

10 <version>1.5.6.RELEASE</version>

11 <configuration>

12 <excludeArtifactIds>my-lib,another-lib</excludeArtifactIds>

13 </configuration>

14 ...

15 </plugin>

16 ...

17 </plugins>

18 ...

19 </build>

20 ...

21 </project>

（3）方式3：排除属于“指定的groupId”的所有maven模块。

1 <project>

2 ...

3 <build>

4 ...

5 <plugins>

6 ...

7 <plugin>

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

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

10 <version>1.5.6.RELEASE</version>

11 <configuration>

12 <excludeGroupIds>com.foo</excludeGroupIds>

13 </configuration>

14 ...

15 </plugin>

16 ...

17 </plugins>

18 ...

19 </build>

20 ...

21 </project>

**4 重写manifest**

这个插件还可以重写项目的manifest （可运行jar包的MANIFEST.MF），特别是管理Main-Class和Start-Class启动类。如果默认的配置不能满足需求，我们可以在这里做配置。通过插件的layout属性可以确实地控制Main-Class。

1 <build>

2 ...

3 <plugins>

4 ...

5 <plugin>

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

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

8 <version>1.5.7.RELEASE</version>

9 <configuration>

10 <mainClass>${start-class}</mainClass>

11 <layout>ZIP</layout>

12 </configuration>

13 <executions>

14 <execution>

15 <goals>

16 <goal>repackage</goal>

17 </goals>

18 </execution>

19 </executions>

20 </plugin>

21 ...

22 </plugins>

23 ...

24 </build>

**5 执行打包命令：mvn package spring-boot:repackage。**注意：这里不能写成mvn spring-boot:repackage。否则会报Source must refer to an existing file异常。

　　layout属性默认根据archive type（jar or war）而定。总的有5中类型：

JAR：常规的可执行jar包的布局。

WAR：可执行war包的布局。

ZIP（和DIR一样）：和使用PropertiesLauncher的jar包布局一样。

MODULE:包括依赖包（排除scope为provided的依赖包）和项目资源

NONE:包括所有的依赖包和项目资源。

**6 利用classifier分离代码包和依赖包**

　　当自己写项目需要发布出去给别人依赖时，就只需要自己写的代码打包，依赖的包不能打进来。否则最后别人在用你提供的包时，就会出现重复包，导致最后打包时包会非常巨大。

　　在这个Spring Boot Maven Plugin这个插件下，我们只需要提供classifier即可实现这个功能。具体如下：

1 <project>

2 ...

3 <build>

4 ...

5 <plugins>

6 ...

7 <plugin>

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

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

10 <version>1.5.7.RELEASE</version>

11 <executions>

12 <execution>

13 <goals>

14 <goal>repackage</goal>

15 </goals>

16 <configuration>

17 <classifier>exec</classifier>

18 </configuration>

19 </execution>

20 </executions>

21 ...

22 </plugin>

23 ...

24 </plugins>

25 ...

26 </build>

27 ...

28 </project>

7 打包时排除resources下的资源文件

1 <build>

2 ...

3 <resources>

4 <resource>

5 <directory>src/main/resources</directory>

6 <excludes>

7 <exclude>\*\*/\*.properties</exclude>

8 <exclude>\*\*/\*.xml</exclude>

9 <exclude>\*\*/\*.yml</exclude>

10 </excludes>

11 </resource>

12 </resources>

13 <plugins>

14 ...

15 <plugin>

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

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

18 <executions>

19 <execution>

20 <goals>

21 <goal>repackage</goal>

22 </goals>

23 </execution>

24 </executions>

25 </plugin>

26 ...

27 </plugins>

28 ...

29 </build>