学习参考文件:



# groupId,artifactId命名规范

GroupId:组织项目名称,如:com.huawei.helloword

ArtifactId:项目-模块名称,如:helloword-sayhello

建议小写

# 基本配置和说明

## pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>love.yi.hellowordweb</groupId>

<artifactId>spring-boot-hellowordweb</artifactId>

<version>0.0.1-SNAPSHOT</version>

<parent>

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

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

<!-- 由引入的父工程统一指定 版本号 -->

<version>1.5.9.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

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

<!-- 包含spring webmvc和tomcat等web开发特性 -->

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

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<!-- 如果使用maven的spring-boot:run运行就不需要该插件 -->

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

</plugin>

</plugins>

</build>

</project>

## App.class

package love.yi.helloword.spring\_boot\_helloword;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

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

//@RestController返回json字符串的数据，直接可以编写RESTFul的接口

@RestController

//让spring boot自动程序进行必要的配置

//等价于以默认属性使用

//@Configuration，@EnableAutoConfiguration和@ComponentScan

@SpringBootApplication

public class App {

@RequestMapping("/")

public String sayHello() {

return "hello world!!";

}

public static void main(String[] args) {

SpringApplication.*run*(App.class,args);

}

}

# Tomcat和JDK版本对应官方说明

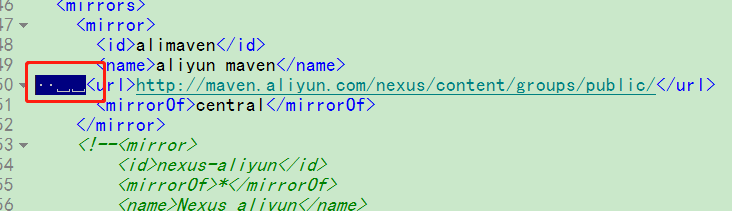


# 遇到的一些问题

## 全角空格我遇到了

[WARNING] Some problems were encountered while building the effective settings [WARNING] expected START\_TAG or END\_TAG not TEXT (position: TEXT seen ...<name>aliyun maven</name>\n \ufffd\ufffd\ufffd\ufffd<u... @150:9) @ D:\apache-maven-3.5.2\conf\settin

是因为标签前面有两个特殊字符导致:其实就是一个全角的空格



## 无故出现异常:

Whitelabel Error Page

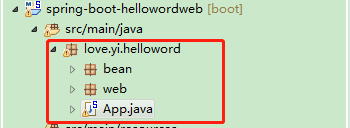
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Oct 05 15:12:37 CST 2018

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

No message available

1. 删除.m2文件夹无果,因为我刚修改了配置在maven安装目录下面
2. 竟然时目录文件结构问题，最后附上官网说明**http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#using-boot-structuring-your-code**



**启动类必须处在所有被管理类的顶层**

## JVM后台一直运行,导致端口占用

关闭java.exe进程

# 热部署插件

<plugin>

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

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

<dependencies>

<!--springloaded hot deploy -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

<version>1.2.4.RELEASE</version>

</dependency>

</dependencies>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

<configuration>

<classifier>exec</classifier>

</configuration>

</execution>

</executions>

</plugin>

如果使用的run as – java application的话，那么还需要做一些处理哦：

把spring-loader-1.2.4.RELEASE.jar下载下来，放到项目的lib目录中，然后把IDEA的run参数里VM参数设置为：

-javaagent:.\lib\springloaded-1.2.4.RELEASE.jar -noverify



# 使用Ali的fastjson转化String返回(spring-boot方案是返回对象)

// 地址：http://127.0.0.1:8080/demo/getFastJson

@RequestMapping("/getFastJson")

public String getFastJson() {

Demo demo = new Demo();

demo.setId(2);

demo.setName("Angel2");

//直接使用ali的fastjson工具转化成json串后直接返回String也是可以的

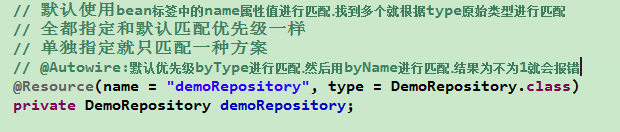
return JSONObject.*toJSONString*(demo);

}

虽然看似多了一行代码，但是使用fastjson对数据的返回的可控性就很强了。当然可能会有更好的解决方案了，在这里

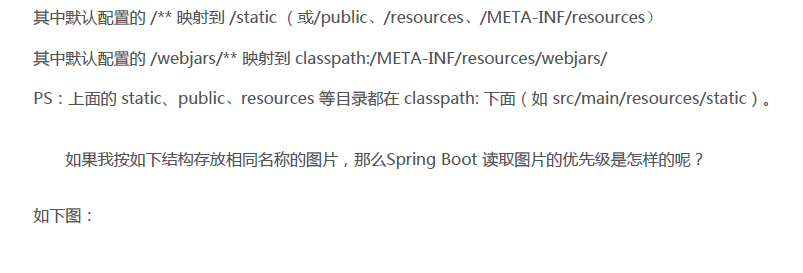
这是抛装引玉。

# @Resource和@Autowire区别



# 处理静态资源, Spring Boot, 默认资源映射







# 自定义配置:不影响默认配置

@Configuration //:加载配置的注释

加载具有配置功能的类上面,是实现自定义动态配置

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 MyWebAppConfigurer extends WebMvcConfigurerAdapter {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

int i = 10;

if(1< i) {

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

super.addResourceHandlers(registry);

}else {

registry.addResourceHandler("/myres/\*\*").addResourceLocations("classpath:/myres2/");

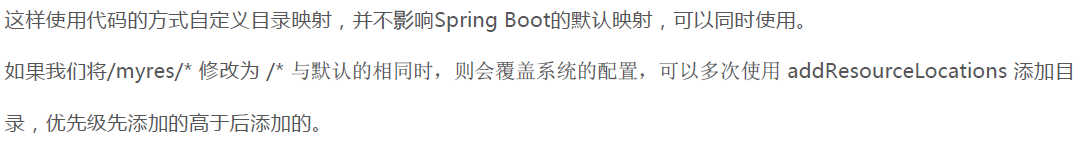
super.addResourceHandlers(registry);

}

}

}

## 将/myres/\* 修改为/\* 与默认的相同时，则会覆盖系统的配置



其中addResourceLocations 的参数是动参，可以这样写addResourceLocations(“classpath:/img1/”,“classpath:/img2/”, “classpath:/img3/”);

## 使用外部目录



# Spring Boot定时任务的使用

## 定时任务注解

@Configuration

@EnableScheduling

@Scheduled

@Configuration

@EnableScheduling

public class SchedulingConfig {

@Scheduled(cron = "0/20 \* \* \* \* ?") // 每20秒执行一次

public void scheduler() {

System.out.println(">>>>>>>>> SchedulingConfig.scheduler()");

}

}

# 使用Druid和监控配置

## SpringBoot默认DS:org.apache.tomcat.jdbc.pool.DataSource

## 配置监控统计功能

配置Servlet

如下是在SpringBoot项目中基于注解的配置，如果是web.xml配置，按规则配置即可。

最后在App.java类上加上注解：@ServletComponentScan是的spring能够扫描到我们自己编写的servlet和filter。

源码:3c0307ef-d9cb-3fa2-9c68-0b47335e82f6.zip

注意不要忘记在SpringBootSampleApplication.java 上添加@ServletComponentScan 注解，不然就是404了。

然后启动项目后访问http://127.0.0.1:8080/druid/index.html 即可查看数据源及SQL统计等。

## 配置监控系统方式二：

以上配置的监控方式是使用了原生的servlet，filter方式，然后通过@ServletComponentScan进行启动扫描包的方式

进行处理的，你会发现我们的servlet，filter根本没有任何的编码。

在这里我们将使用另外一种方式进行处理：使用代码注册Servlet：

源码:

# 使用Druid（编程注入）

# Spring Boot普通类调用bean

我们知道如果我们要在一个类使用spring提供的bean对象，我们需要把这个类注入到spring容器中，交给spring容器进行管理，但是在实际当中，我们往往会碰到在一个普通的Java类中，想直接使用spring提供的其他对象或者说有一些不需要交给spring管理，但是需要用到spring里的一些对象。如果这是spring框架的独立应用程序，我们通过

ApplicationContext ac = new FileSystemXmlApplicationContext("applicationContext.xml");

ac.getBean("beanId");

这样的方式就可以很轻易的获取我们所需要的对象。

但是往往我们所做的都是Web Application，这时我们启动spring容器是通过在web.xml文件中配置，这样就不适合使用上面的方式在普通类去获取对象了，因为这样做就相当于加载

了两次spring容器，而我们想是否可以通过在启动web服务器的时候，就把Application放在某一个类中，我们通过这个类在获取，这样就可以在普通类获取spring bean对象了，让我们接着往下看

package love.yi.helloword.base.util;

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

import org.springframework.stereotype.Component;

/\*\*

\* 普通类调用Spring bean对象： 说明： 1、此类需要放到App.java同包或者子包下才能被扫描，否则失效。

\*

\* @author Administrator

\*/

@Component

public class SpringUtil implements ApplicationContextAware {

private static ApplicationContext applicationContext = null;

@Override

public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {

if (SpringUtil.applicationContext == null) {

//使用了单例模式保证对象只被赋值一次

SpringUtil.applicationContext = applicationContext;

}

System.out.println("---------------------------------------------------------------------");

System.out.println("---------------------------------------------------------------------");

System.out.println(

"---------------com.kfit.base.util.SpringUtil------------------------------------------------------");

System.out.println(

"========ApplicationContext配置成功,在普通类可以通过调用SpringUtils.getAppContext()获取applicationContext对象,applicationContext="

+ SpringUtil.applicationContext + "========");

System.out.println("---------------------------------------------------------------------");

}

//获取applicationContext

public static ApplicationContext getApplicationContext() {

return applicationContext;

}

//通过name获取Bean.

public static Object getBean(String name) {

return getApplicationContext().getBean(name);

}

//通过class获取Bean.

public static <T> T getBean(Class<T> clazz) {

return getApplicationContext().getBean(clazz);

}

//通过name,以及Clazz返回指定的Bean

public static <T> T getBean(String name, Class<T> clazz) {

return getApplicationContext().getBean(name, clazz);

}

}

## 不在Spring Boot的扫描包下方式一:就是在App类中创建我们的SpringUtil对象(或者说是创建我们的Application对象)

@SpringBootApplication

@ServletComponentScan

public class App {

/\*\*注册Spring Util

\* 这里为了和上一个冲突，所以方面名为：springUtil2

\* 实际中使用springUtil

\*/

@Bean

public SpringUtil springUtil2(){return new SpringUtil();}

/\*\*

\*

参数里VM参数设置为：

-javaagent:.\lib\springloaded-1.2.4.RELEASE.jar -noverify

\* @param args

\*/

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

## 不在Spring Boot的扫描包下方式二

代码基本和上面都是相同的，主要是在App.java中使用@Import进行导入。

而且在SpringUtil是不需要添加@Component注解

@SpringBootApplication

@ServletComponentScan

@Import(value={SpringUtil.class})

publicclass App {

//省略其它代码.

}

# 使用模板thymeleaf

**Thymeleaf**是适用于Web和独立环境的现代服务器端Java模板引擎。

Thymeleaf的主要目标是为您的开发工作流程带来优雅的自然模板 - 可以在浏览器中正确显示的HTML，也可以用作静态原型，从而在开发团队中实现更强大的协作。

借助Spring Framework的模块，与您喜欢的工具的大量集成，以及插入您自己的功能的能力，Thymeleaf是现代HTML5 JVM Web开发的理想选择 - 尽管它可以做得更多。

Spring Boot默认就是使用thymeleaf模板引擎的

<dependency>

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

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

</dependency>

Thymeleaf缓存在开发过程中，肯定是不行的，那么就要在开发的时候把缓存关闭，只需要

在application.properties进行配置即可：

########################################################

###THYMELEAF (ThymeleafAutoConfiguration)

########################################################

#spring.thymeleaf.prefix=classpath:/templates/

#spring.thymeleaf.suffix=.html

#spring.thymeleaf.mode=HTML5

#spring.thymeleaf.encoding=UTF-8

# ;charset=<encoding> is added

#spring.thymeleaf.content-type=text/html

# set to false for hot refresh

html5

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:th="http://www.thymeleaf.org"

xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">

<head>

<title>Hello World!</title>

</head>

<body>

<h1 th:inline="text">Hello.v.2</h1>

<p th:text="${hello}"></p>

</body>

</html>

controller

@Controller

public class TemplateController {

/\*\*

\* 返回html模板.

\*/

@RequestMapping("/helloHtml")

public String helloHtml(Map<String, Object> map) {

map.put("hello", "from TemplateController.helloHtml");

return "/helloHtml";//默认在/src/main/resource/templates下面,并且去掉了后缀

}

}

<http://127.0.0.1:8090/helloHtml>

# 使用模板freemarker

<dependency>

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

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

</dependency>

########################################################

###FREEMARKER (FreeMarkerAutoConfiguration)

########################################################

spring.freemarker.allow-request-override=false

spring.freemarker.cache=true

spring.freemarker.check-template-location=true

spring.freemarker.charset=UTF-8

spring.freemarker.content-type=text/html

spring.freemarker.expose-request-attributes=false

spring.freemarker.expose-session-attributes=false

spring.freemarker.expose-spring-macro-helpers=false

#spring.freemarker.prefix=

#spring.freemarker.request-context-attribute=

#spring.freemarker.settings.\*=

#spring.freemarker.suffix=.ftl

#spring.freemarker.template-loader-path=classpath:/templates/#comma-separatedlist

#spring.freemarker.view-names= #whitelistofviewnamesthatcanberesolved

@RequestMapping("/helloFtl")

public String helloFtl(Map<String,Object> map){

map.put("hello","from TemplateController.helloFtl");

return"/helloFtl";

}

**src/main/resouces/templates/helloFtl.ftl**

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"

xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">

<head>

<title>Hello World!</title>

</head>

<body>

<h1>Hello.v.2</h1>

<p>${hello}</p>

</body>

</html>

**thymeleaf和freemarker是可以共存的。**

# 后面内容比较复杂将新建一个项目

## pom.xml文件

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>spring-boot-jsp</groupId>

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

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>spring-boot-jsp-web Maven Webapp</name>

<url>http://maven.apache.org</url>

<!-- spring boot parent节点，引入这个之后，在下面和spring boot相关的就不需要引入版本了; -->

<parent>

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

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

<version>1.3.3.RELEASE</version>

</parent>

<!-- Jdk编译版本： -->

<build>

<finalName>spring-boot-jsp-web</finalName>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- web支持: 1、web mvc; 2、restful; 3、jackjson支持; 4、aop ........ -->

<dependency>

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

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

</dependency>

<!-- servlet 依赖. -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<scope>provided</scope>

</dependency>

<!-- JSTL（JSP Standard Tag Library，JSP标准标签库)是一个不断完善的开放源代码的JSP标签库，是由apache的

jakarta小组来维护的。JSTL只能运行在支持JSP1.2和Servlet2.3规范的容器上，如tomcat 4.x。在JSP 2.0中也是作为标准支

持的。 不然报异常信息： javax.servlet.ServletException: Circular view path [/helloJsp]:

would dispatch back to the current handler URL [/helloJsp] again. Check your

ViewResolver setup! (Hint: This may be the result of an unspecified view,

due to default view name generation.) -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<!-- tomcat 的支持. -->

<dependency>

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

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

<scope>provided</scope>

</dependency>

<dependency>

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

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

<scope>provided</scope>

</dependency>

</dependencies>

</project>

## application.properties配置

上面说了spring-boot 不推荐JSP，想使用JSP需要配置application.properties。

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 Angel From application

Controller

package com.kfit.jsp.controller;

import java.util.Map;

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

import org.springframework.stereotype.Controller;

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

/\*\*

\* 测试

\*/

@Controller

public class HelloController {

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

@Value("${application.hello:Hello Angel}")

private String hello;

@RequestMapping("/helloJsp") //RequestMapping报错就是版本号的问题所致

public String helloJsp(Map<String, Object> map) {

System.out.println("HelloController.helloJsp().hello=" + hello);

map.put("hello", hello);

return "helloJsp";

}

}

jsp

在src/main 下面创建webapp/WEB-INF/jsp 目录用来存放我们的jsp页面：helloJsp.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/

loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Insert title here</title>

</head>

<body>

helloJsp

<hr>

${hello}

</body>

</html>

## 编写App.java启动类

package com.kfit.jsp;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.web.support.SpringBootServletInitializer;

@SpringBootApplication

public class App extends SpringBootServletInitializer {

// @Override

// protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

// return application.sources(App.class);

// }

public static void main(String[] args) {

SpringApplication.run(App.class, args);

}

}

[**http://127.0.0.1:8080/helloJsp**](http://127.0.0.1:8080/helloJsp)

# Spring Boot Servlet

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？

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。这里我们新建一个spring-boot-hello2 java工程进行测试；这里不过多进行说明，如果这个还不会的话，请回到上上上一章进行查看。

## 通过代码注册Servlet示例代码：

package com.kfit.servlet;

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

/\*\*

\*

\* @author Angel(QQ:412887952)

\* @version v.0.1

\*/

//这个不需要添加.

//@WebServlet(urlPatterns="/myServlet1/\*", description="Servlet的说明")

public class MyServlet1 extends HttpServlet {

private static final long *serialVersionUID* = 1L;

@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>这是：MyServlet1</h1>");

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

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

}

}

在com.kfit.App中注册：

package com.kfit;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

import org.springframework.context.annotation.Bean;

import com.kfit.servlet.MyServlet1;

/\*\*

\* 大家也许会看到有些demo使用了3个注解： @Configuration；

\* @EnableAutoConfiguration

\* @ComponentScan

\* 其实：@SpringBootApplication申明让spring boot自动给程序进行必要的配置，

\*

\* 等价于以默认属性使用@Configuration，

\* @EnableAutoConfiguration和@ComponentScan

\* 所以大家不要被一些文档误导了，让自己很迷茫了，希望本文章对您有所启发；

\*

\* @author Angel(QQ:412887952)

\* @version v.0.1

\*/

@SpringBootApplication

public class App {

/\*\*

\* 注册Servlet.不需要添加注解：@ServletComponentScan

\* @return

\*/

@Bean

public ServletRegistrationBean MyServlet1() {

return new ServletRegistrationBean(new MyServlet1(), "/myServlet/\*");

}

public static void main(String[] args) {

SpringApplication.*run*(App.class, args);

}

}

## 使用注解

package com.kfit.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;

/\*\*

\*

\* @author Angel(QQ:412887952)

\* @version v.0.1

\* 使用注解放式就需要在启动类中添加servlet扫描注解@ServletComponentScan

\*/

@WebServlet(urlPatterns = "/myServlet2/\*", description = "Servlet的说明")

public class MyServlet2 extends HttpServlet {

private static final long *serialVersionUID* = 1L;

@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>这是：myServlet2</h1>");

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

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

}

}

@SpringBootApplication

@ServletComponentScan // 这个就是扫描相应的Servlet包;

public class SpringBootSampleApplication {

public static void main(String[] args) {

SpringApplication.*run*(SpringBootSampleApplication.class, args);

}

}

# Spring Boot过滤器、监听器

上一篇文章已经对定义Servlet 的方法进行了说明，过滤器（Filter）和监听器（Listener）的注册方法和Servlet

一样，不清楚的可以查看下上一篇文章（20）： 本文将直接使用@WebFilter和@WebListener的方式，完成一个

Filter 和一个Listener；使用注解

@ServletComponentScan//这个就是扫描相应的Servlet包;

Filter

/\*\*

\* 使用注解标注过滤器

\* @WebFilter将一个实现了javax.servlet.Filter接口的类定义为过滤器 属性filterName声明过滤器的名称,可选

\* 属性urlPatterns指定要过滤的URL模式,也可使用属性value来声明.(指定要过滤的URL模式是必选属性)

\* @author Angel(QQ:412887952)

\* @version v.0.1

\*/

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

public class MyFilter implements Filter {

@Override

public void init(FilterConfig config) throws ServletException {

System.*out*.println("过滤器初始化");

}

@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 destroy() {

System.*out*.println("过滤器销毁");

}

}

ServletContext:监听器（Listener）文件

/\*\*

\* 使用@WebListener注解，实现ServletContextListener接口

\*

\* @author Angel(QQ:412887952)

\* @version v.0.1

\*/

@WebListener

public class MyServletContextListener implements ServletContextListener {

@Override

public void contextDestroyed(ServletContextEvent arg0) {

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

}

@Override

public void contextInitialized(ServletContextEvent arg0) {

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

}

}

ServletContext监听器（Listener）文件（HttpSessionListener）

/\*\*

\* 使用@WebListener注解，实现ServletContextListener接口

\* @version v.0.1

\*/

@WebListener

public class MyServletContextListener implements ServletContextListener {

@Override

public void contextDestroyed(ServletContextEvent arg0) {

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

}

@Override

public void contextInitialized(ServletContextEvent arg0) {

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

}

}

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

为什么无法看到session的过程：http://zhidao.baidu.com/link?url=EPwlLvKpo8zI5NaIZrESzCdivq3Xg8VgOWQOvfpSLl3opTgvESerpo4wsG6tOs\_dm6cQQMF\_kQ6THNjNzr2Nq

至于如何使用代码的方式注册Filter和Listener，请参考上一篇文章关键Servlet的介绍。不同的是需要使用FilterRegistrationBean 和ServletListenerRegistrationBean 这两个类。

# SpringBoot拦截器HandlerInterceptor

上一篇对过滤器的定义做了说明，也比较简单。过滤器属于Servlet范畴的API，与Spring 没什么关系。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第 102 / 266 页

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。实现自定义拦截器只需要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）。

PS：本文重点在如何在Spring-Boot中使用拦截器，关于拦截器的原理请大家查阅资料了解。

**interceptor:MyInterceptor1**

/\*\*

\* 自定义拦截器1

\*

\* @author Angel

\*/

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）");

}

}

**interceptor:MyInterceptor2**

/\*\*

\* 自定义拦截器2

\* @author Angel

\*/

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）");

}

}

MyWebAppConfigurer

// 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

@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);

}

}

[**http://localhost:8080/index**](http://localhost:8080/index)

**输出:**

执行过滤操作

>>>MyInterceptor1>>>>>>>在请求处理之前进行调用（Controller方法调用之前）

>>>MyInterceptor2>>>>>>>在请求处理之前进行调用（Controller方法调用之前）

>>>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）

>>>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）

>>>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）

>>>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）

根据输出可以了解拦截器链的执行顺序（具体原理介绍，大家找度娘一问便知）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/myServlet1 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。

最后说明下，我们上面用到的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）。

# 启动加载数据CommandLineRunner

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。很简单，只需要一个类就可以，无需其他配置。创建实现接口com.kfit.runner.CommandLineRunner 的类

**MyStartupRunner1**

/\*\*

\* 服务启动执行

\* @author Angel(QQ:412887952)

\*/

@Component

@Order(value=2)

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<<<<<<<<<<<<<");

}

}

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。

如下我们使用@Order 注解来定义执行顺序。

**MyStartupRunner2**

/\*\*

\* 服务启动执行

\* @author Angel(QQ:412887952)

\*/

@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<<<<<<<<<<<<<");

}

}

根据控制台结果可判断，@Order 注解的执行优先级是按value值从小到大顺序。

@Override

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

System.*out*.println(Arrays.*asList*(args));

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

}

这里的args就是程序启动的时候进行设置的:

SpringApplication.run(App.class, new String[]{"hello,","林峰"});

这里为了做演示，配置为固定值了，其实直接接收main中的args即可，那么在运行的时候，进行配置即可。

**题外话：**

eclipse中给java应用传args参数的方法如下：

1、先写好Java代码，比如文件名为IntArrqy.java；

2、在工具栏或菜单上点run as下边有个Run Configuration；

3、在弹出窗口点选第二个标签arguments；

4、把你想输入的参数写在program argumenst就可以了，多个参数使用空格隔开。

完成后点run即可通过运行结果看到参数使用情况了。

# Spring Boot环境变量读取和属性对象的绑定

凡是被Spring管理的类，实现接口EnvironmentAware 重写方法setEnvironment 可以在工程启动时，获取到系统环境变量和application配置文件中的变量。

MyEnvironmentAware

/\*\*

\* 主要是@Configuration，实现接口：EnvironmentAware就能获取到系统环境信息;

\* @author Angel(QQ:412887952)

\* @version v.0.1

\*/

@Configuration

public class MyEnvironmentAware implements EnvironmentAware {

//注入application.properties的属性到指定变量中.

@Value("${spring.datasource.url}")

private String myUrl;

/\*\*

\* 注意重写的方法setEnvironment 是在系统启动的时候被执行。

\*/

@Override

public void setEnvironment(Environment environment) {

//打印注入的属性信息.

System.*out*.println("myUrl=" + myUrl);

//通过environment 获取到系统属性.

System.*out*.println(environment.getProperty("JAVA\_HOME"));

//通过environment 同样能获取到application.properties配置的属性.

System.*out*.println(environment.getProperty("spring.datasource.url"));

//获取到前缀是"spring.datasource." 的属性列表值.

RelaxedPropertyResolver relaxedPropertyResolver = new RelaxedPropertyResolver(environment,

"spring.datasource.");

System.*out*.println("spring.datasource.url=" + relaxedPropertyResolver.getProperty("url"));

System.*out*

.println("spring.datasource.driverClassName=" + relaxedPropertyResolver.getProperty("driverClassName"));

}

}

@Controller @Service 等被Spring管理的类都支持，注意重写的方法setEnvironment 是在系统启动的时候被执行。

或者如下Controller：

@Controller

public class PageController implements EnvironmentAware {

@Override

public void setEnvironment(Environment environment) {

String s = environment.getProperty("JAVA\_HOME");

System.*out*.println(s);

}

}

我们还可以通过@ConfigurationProperties 读取application属性配置文件中的属性。

@Configuration

@ConditionalOnClass(Mongo.class)

@EnableConfigurationProperties(MongoProperties.class)

public class MongoAutoConfiguration {

@Autowired

private MongoProperties properties;

}

@ConditionOnClass表明该@Configuration仅仅在一定条件下才会被加载，这里的条件是Mongo.class位于类路径上 @EnableConfigurationProperties将Spring Boot的配置文件（application.properties）中的spring.data.mongodb.\*属性映射为MongoProperties并注入到MongoAutoConfiguration中。· @ConditionalOnMissingBean说明Spring Boot仅仅在当前上下文中不存在Mongo对象时，才会实例化一个Bean。这个逻辑也体现了Spring Boot的另外一个特性——自定义的Bean优先于框架的默认配置，我们如果显式的在业务代码中定义了一个Mongo对象，那么Spring Boot就不再创建。

@ConfigurationProperties(prefix = "spring.data.mongodb")

public class MongoProperties {

private String host;

//private int port = DBPort.PORT;

private String uri = "mongodb://localhost/test";

private String database;

// ... getters/ setters omitted

}

以上这个配置需要加入依赖：

<!--spring-boot-configuration:spring boot 配置处理器; -->

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

# Spring Boot使用自定义的properties

为了提高学习效率,一下内容只进行标记在pdf中的位置:

p117

# 改变自动扫描的包

@ComponentScan注解进行指定要扫描的包以及要扫描的类。

@ComponentScan(basePackages={"cn.kfit","org.kfit"})

p122

# Spring Boot Junit单元测试

p125

//// SpringJUnit支持，由此引入Spring-Test框架支持！

@RunWith(SpringJUnit4ClassRunner.class)

//// 指定我们SpringBoot工程的Application启动类

@SpringApplicationConfiguration(classes = App.class)

///由于是Web项目，Junit需要模拟ServletContext，因此我们需要给我们的测试类加上@WebAppConfiguration。

@WebAppConfiguration

# SpringBoot启动时的Banner设置

p130

banner.txt

#这个是MANIFEST.MF文件中的版本号

${application.version}

#这个是上面的的版本号前面加v后上括号

${application.formatted-version}

#这个是springboot的版本号

${spring-boot.version}

#这个是springboot的版本号

${spring-boot.formatted-version}

class MyBanner implements Banner{

@Override

public void printBanner(Environment environment, Class<?> sourceClass, PrintStream out) {

//banner.txt文件优先级比该方法有限级高

System.err.println("明天要上班了额!");

}

}

# Spring boot 文件上传（多文件上传）

133

（1）新建maven java project；

（2）在pom.xml加入相应依赖；

（3）新建一个表单页面（这里使用thymleaf）;

（4）编写controller;

（5）测试；

（6）对上传的文件做一些限制；

（7）多文件上传实现