前言

1. 一个适合初学者的博客：<http://www.cnblogs.com/ityouknow/category/914493.html>

2. Spring boot Referece 中文翻译：

<https://github.com/qibaoguang/Spring-Boot-Reference-Guide>

3.spring 学习资料网站： http://springboot.fun/

4.代码实例：

<https://github.com/ityouknow/spring-cloud-examples>

<https://github.com/YunaiV/SpringBoot-Labs>

<https://gitee.com/zx19890628/spring-boot-example>

文档更新，文档只适用于下面列表中最后一个版本

|  |  |
| --- | --- |
| 2017-06-01 | spring boot版本是1.3.6 |
| 2017-11-06 | spring boot版本是1.5.2 |
| 2018-08-15 | spring boot版本是2.0.4 |
| 2020-09-21 | spring boot版本是2.3.3 |

# 单元测试

案例：

[https://gitee.com/zx19890628/spring-boot-example/tree/master/lab\_000\_base\_web](https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_000_base_web/src/test/java/com/zx/web)

Spring Boot 内置了7种测试框架

* JUnit： 一个 Java 语言的单元测试框架
* Spring Test & Spring Boot Test：为 Spring Boot 应用提供集成测试和工具支持
* AssertJ：支持流式断言的 Java 测试框架
* Hamcrest：一个匹配器库
* Mockito：一个 java mock 框架
* JSONassert：一个针对 JSON 的断言库
* JsonPath：JSON XPath 库

## Junit5

Junit5官网：<https://junit.org/junit5/>

## Mock

简单教程：<https://www.cnblogs.com/zengls/p/11316454.html>

案例：[JunitControllerTest](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/test/java/com/zx/web/test/JunitControllerTest.java)

# 项目配置

案例：

[https://gitee.com/zx19890628/spring-boot-example/tree/master/lab\_000\_base\_web](https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_000_base_web/src/test/java/com/zx/web)

Spring boot提倡简化，所以我们在项目中可以不使用xml的配置，换之而来的是将所有的配置添加到application.properties中，约定好配置的内容，并且会给所有的配置一个默认值。

目前spring boot支持的配置方式包括properties 和 yml两种，这里是推荐使用yml的配置，yml使配置表现出层次感。

Spring boot 中默认的配置都支持配置文件、java代码、命令行启动等三种方式，其中的执行优先级如下：

这些方式优先级如下：  
a. 命令行参数  
b. 来自java:comp/env的JNDI属性  
b. Java系统属性（System.getProperties()）  
d. 操作系统环境变量  
e. RandomValuePropertySource配置的random.\*属性值  
f. jar外部的application-{profile}.properties或application.yml(带spring.profile)配置文件  
g. jar内部的application-{profile}.properties或application.yml(带spring.profile)配置文件  
h. jar外部的application.properties或application.yml(不带spring.profile)配置文件  
i. jar内部的application.properties或application.yml(不带spring.profile)配置文件  
j. @Configuration注解类上的@PropertySource  
k. 通过SpringApplication.setDefaultProperties指定的默认属性

## 端口配置

1.方式一，配置在application.properties中，spring 自动加载

server.port=8000

server.session.timeout=30000

2. 方式二，写到@SpringBootConfiguration注解的类中。

看案例中的：ServerConfig.java

3.方式三，可以在打成jar包启动的时候增加参数：

java -jar 201701419\_cloud-0.0.1-SNAPSHOT.jar --server.port=8000

## 服务路径配置

主要的配置如下：

server.servlet.context-path=/hebe/v1/

这个context-path就是和tomcat中的context-path 一样，就像是项目的名称一样。spring boot 默认是 /

我们可以在url中配置版本、服务名称等信息。当我们请求服务的时候，url的变化如下

配置前：

http://127.0.0.1:8888/hello?name=233

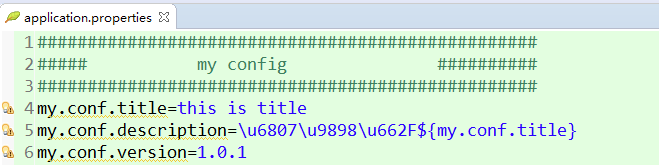
配置后：

http://127.0.0.1:8888/hebe/v1/hello?name=233

## 自定义配置

### 定义配置

其中${my.conf.title}是引用其他配置



### 使用配置

第一种：使用spring支持的@Value()加载

第二种：使用@ConfigurationProperties(prefix="") 设置前缀，属性上不需要添加注解

@Component

@ConfigurationProperties(prefix = "my.conf")//配置的前缀

public class MyConfig {

@Value("${my.conf.title}")//第一种：使用spring支持的@Value()加载

private String title;

private String description;//第二种：使用@ConfigurationProperties(prefix="") 设置前缀，属性上不需要添加注解

private String version;

}

### 调用

Spring 调用配置的两种方式

第一种：使用注入好配置的类，比如上面的MyConfig

第二种：使用@Value("#{myConfig.title}")，调用MyConfig 对象中的title属性。

@Autowired

private MyConfig myConfig;

@Value("#{myConfig.title}")

private String title;

@GetMapping("/config")

public String config() {

System.out.println(myConfig);

System.out.println(title);

return "ok";

}

结果如下



## 生成随机数

在一些情况下，有些参数我们需要希望它不是一个固定的值，比如密钥、服务端口等

随机字符串：com.didispace.blog.value=${random.value}

随机int：com.didispace.blog.number=${random.int}

随机long：com.didispace.blog.bignumber=${random.long}

随机10以内int：com.didispace.blog.test1=${random.int(10)}

随机10-20int：com.didispace.blog.test2=${random.int[10,20]}

## 通过命令行设置属性

例如：java -jar demo.jar --server.port=8888

参数是以两个减号开始的

但是Spring Boot提供了屏蔽命令行访问属性的设置

参考：[MyApplication](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/main/java/com/zx/web/MyApplication.java)

## 多环境配置

我们在开发Spring Boot应用时，通常同一套程序会被应用和安装到几个不同的环境，比如：开发、测试、生产等。

在Spring Boot中多环境配置文件名需要满足application-{profile}.properties的格式，其中{profile}对应你的环境标识，比如：

application-dev.properties：开发环境

application-test.properties：测试环境

application-prod.properties：生产环境

至于哪个具体的配置文件会被加载，需要在application.properties文件中通过spring.profiles.active属性来设置，其值对应{profile}值。

如：spring.profiles.active=test就会加载application-test.properties配置文件内容

application.properties中设置spring.profiles.active=dev，就是说默认以dev环境设置

测试不同配置的加载

执行java -jar xxx.jar，就是默认的开发环境（dev）

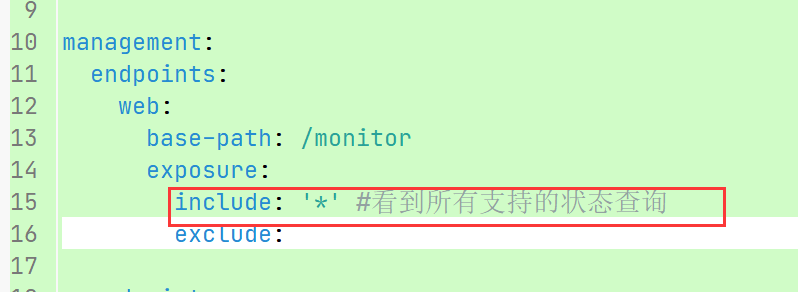
执行java -jar xxx.jar --spring.profiles.active=test，就是测试环境的配置（test）

执行java -jar xxx.jar --spring.profiles.active=prod，就是生产环境的配置（prod）

实际生产中，可以通过命令行方式去激活不同环境的配置

## 常见问题

在yml文件中的特殊符号。比如在使用\*的时候，会报错。我们将其以单引号包起来即可 ‘\*’



# Web

案例：

[https://gitee.com/zx19890628/spring-boot-example/tree/master/lab\_000\_base\_web](https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_000_base_web/src/test/java/com/zx/web)

## 静态资源

Spring Boot默认提供静态资源目录位置需置于classpath下，目录名需符合如下规则：

/static

/public

/resources

/META-INF/resources

举例：我们可以在src/main/resources/目录下创建static，在该位置放置一个图片文件。启动程序后，尝试访问http://localhost:8080/D.jpg。如能显示图片，配置成功。

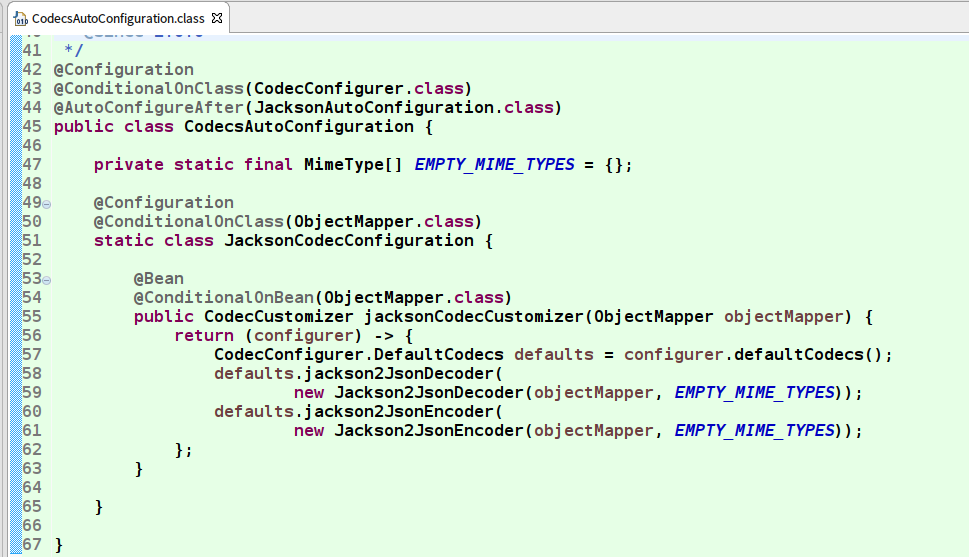
## JSON

Spring boot可以帮助我们自动将类型转为默认的类型，也可以我们自己执行响应的类型。

@GetMapping(value="/", produces = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)

### 替换默认的类型转换器

而spring boot 默认使用jackson作为json格式的转换器，如果我们想要jackson更换为fastJson。



案例看：[FastjsonConverterConfig](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/main/java/com/zx/web/converter/config/FastjsonConverterConfig.java)

开启debug日志，访问json格式的请求后，日志中出现FastJsonHttpMessageConverter相关的信息即可。

Written [[User [id=1, name=张三, age=23]]] as "application/json;charset=UTF-8" using [com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter@268623af]

## Servlet

### 代码案例

Spring Boot 提供了 ServletRegistrationBean, FilterRegistrationBean, ServletListenerRegistrationBean 三个类分别用来注册 Servlet, Filter, Listener。

Servlet的案例参考：[MyServlet](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/main/java/com/zx/web/servlet/MyServlet.java)

Ps:直接使用Servlet，访问会不经过spring 的拦截器，导致出现漏洞的风险

### 注解

Servlet 3.0 之前，Servlet、Filter、Listener 这些组件都需要在 web.xml 中进行配置，3.0 之后开始不再需要 web.xml 这个配置文件了，所有的组件都可以通过代码配置或者注解来达到目的。

@WebServlet => 代替 servlet 配置

@WebServlet(name = "javaServlet", urlPatterns = "/javastack.cn", asyncSupported = true,

initParams = {

@WebInitParam(name = "name", value = "javastack"),

@WebInitParam(name = "sex", value = "man") })

@WebFilter => 代替 filter 配置

@WebFilter(filterName = "javaFilter", urlPatterns = "/\*", initParams = {

@WebInitParam(name = "name", value = "javastack"),

@WebInitParam(name = "code", value = "123456") })

@WebListener => 代替 listener 配置

需要注意的是，为了安全考虑，内嵌服务器不会直接执行 Servlet 3.0 里面的 javax.servlet.ServletContainerInitializer 接口，或者 Spring 中的 org.springframework.web.WebApplicationInitializer 接口，否则会导致终止 Spring Boot 应用。

所以，如果使用的是 Spring Boot 内嵌服务器，需要在配置类上面添加额外的 @ServletComponentScan 注解来开启 Servlet 组件扫描功能，如果使用的是独立的服务器，则不需要添加，会使用服务器内部的自动发现机制。

## Filter

案例参考：[FilterConfig](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/main/java/com/zx/web/filter/config/FilterConfig.java)

## 监听器

@WebListener => 代替 listener 配置

需要注意的是，为了安全考虑，内嵌服务器不会直接执行 Servlet 3.0 里面的 javax.servlet.ServletContainerInitializer 接口，或者 Spring 中的 org.springframework.web.WebApplicationInitializer 接口，否则会导致终止 Spring Boot 应用。

所以，如果使用的是 Spring Boot 内嵌服务器，需要在配置类上面添加额外的 @ServletComponentScan 注解来开启 Servlet 组件扫描功能，如果使用的是独立的服务器，则不需要添加，会使用服务器内部的自动发现机制。

[案例](https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_000_base_web/src/main/java/com/zx/web/listener)

## 拦截器

这里推荐使用实现WebMvcConfigurer 接口，WebMvcConfigurerAdapter类已经被弃用了。

[案例](https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_000_base_web/src/main/java/com/zx/web/interceptor)

## 统一异常处理

spring boot有自己默认的异常处理页面，但是实际项目中需求自定义异常页面

需要设置 全局异常处理，[GlobalExceptionHandler](https://gitee.com/zx19890628/spring-boot-example/blob/master/lab_000_base_web/src/main/java/com/zx/web/exception/handler/GlobalExceptionHandler.java)

我们可以自己设置异常的类型，比如，HtmlException 和 AjaxException ，用来分别处理异常，是返回数据还是页面等

## 跨域

CORS是一个W3C标准，全称是”跨域资源共享”（Cross-origin resource sharing）。它允许浏览器向跨源(协议 + 域名 + 端口)服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

简单来说，跨域问题是可以通过nginx来解决的，或者通过jsonp(只支持get请求)来解决。而SpringBoot中也提供了配置方法。

我们在使用ajax的时候，常见的问题之一就是跨域。

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Insert title here</title>

<script type="text/javascript" src="http://apps.bdimg.com/libs/jquery/2.1.1/jquery.min.js"></script>

</head>

<body>

<script type="text/javascript">

$.support.cors = true;

$.ajax({

type : "get",

async : true,

dataType:'text',

url: 'http://127.0.0.1:8080/get1',

data: {},

success : function(data){

alert(data)

},

});

$.ajax({

type : "get",

async : true,

dataType:'text',

url: 'http://127.0.0.1:8080/get2',

data: {},

success : function(data){

alert(data)

},

});

$.ajax({

type : "get",

async : true,

dataType:'text',

url: 'http://127.0.0.1:8080/get3',

data: {},

success : function(data){

alert(data)

},

});

</script>

</body>

</html>

### 方式1

直接设置响应头

@RequestMapping("/get2")

public String get2(HttpServletResponse response) {

response.addHeader("Access-Control-Allow-Origin", "\*");// 设置允许请求的域名，多个域名以逗号分隔，\*是全部

response.addHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept, Authorization");// 设置允许请求自定义的请求头字段，多个字段以逗号分隔

response.addHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS");// 设置允许请求的方法，多个方法以逗号分隔

response.addHeader("Access-Control-Allow-Credentials", "true");// 设置是否允许发送 Cookies

return "get2";

}

### 方式2（推荐）

在class/method上添加枚举@CrossOrigin

@RequestMapping("/get3")

@CrossOrigin

public String get3() {

return "get3";

}

### 方式3

配置全局的过滤器

/\*\*

\* @Project: 20191212-jsonp

\* @Title: CorsConfig

\* @Description: 全局设置所有的请求均设置

\* @author: zhangxue

\* @date: 2019年12月12日下午11:05:06

\* @company: 未知

\* @Copyright: Copyright (c) 2015

\* @version v1.0

\*/

@Configuration

public class CorsConfig {

private CorsConfiguration buildConfig() {

CorsConfiguration corsConfiguration = new CorsConfiguration();

// 可以自行筛选

corsConfiguration.addAllowedOrigin("\*");

corsConfiguration.addAllowedHeader("\*");

corsConfiguration.addAllowedMethod("\*");

return corsConfiguration;

}

@Bean

public CorsFilter corsFilter() {

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", buildConfig());

return new CorsFilter(source);

}

}

## 文件上传

Embedded Tomcat默认上传文件的大小是10M。

tomcatEmbedded这段代码是为了解决，上传文件大于10M出现连接重置的问题。此异常内容GlobalException也捕获不到。

### 设置上传限制

-1表示为不限制。

@SpringBootApplication

public class FileUploadWebApplication {

public static void main(String[] args) throws Exception {

SpringApplication.run(FileUploadWebApplication.class, args);

}

//Tomcat large file upload connection reset

@Bean

public TomcatEmbeddedServletContainerFactory tomcatEmbedded() {

TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFactory();

tomcat.addConnectorCustomizers((TomcatConnectorCustomizer) connector -> {

if ((connector.getProtocolHandler() instanceof AbstractHttp11Protocol<?>)) {

//-1 means unlimited

((AbstractHttp11Protocol<?>) connector.getProtocolHandler()).setMaxSwallowSize(-1);

}

});

return tomcat;

}

}

### 页面

<!DOCTYPE html>

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

<body>

<h1>Spring Boot file upload example</h1>

<form method="POST" action="/upload" enctype="multipart/form-data">

<input type="file" name="file" /><br/><br/>

<input type="submit" value="Submit" />

</form>

</body>

</html>

### Controller

@PostMapping("/upload")

public String singleFileUpload(@RequestParam("file") MultipartFile file,

RedirectAttributes redirectAttributes) {

if (file.isEmpty()) {

redirectAttributes.addFlashAttribute("message", "Please select a file to upload");

return "redirect:uploadStatus";

}

try {

// Get the file and save it somewhere

byte[] bytes = file.getBytes();

Path path = Paths.get(UPLOADED\_FOLDER + file.getOriginalFilename());

Files.write(path, bytes);

redirectAttributes.addFlashAttribute("message",

"You successfully uploaded '" + file.getOriginalFilename() + "'");

} catch (IOException e) {

e.printStackTrace();

}

return "redirect:/uploadStatus";

}

上面代码的意思就是，通过MultipartFile读取文件信息，如果文件为空跳转到结果页并给出提示；如果不为空读取文件流并写入到指定目录，最后将结果展示到页面。

MultipartFile是Spring上传文件的封装类，包含了文件的二进制流和文件属性等信息，在配置文件中也可对相关属性进行配置，基本的配置信息如下：

spring.http.multipart.enabled=true #默认支持文件上传

spring.http.multipart.file-size-threshold=0 #支持文件写入磁盘

spring.http.multipart.location=# 上传文件的临时目录

spring.http.multipart.max-file-size=1Mb # 最大支持文件大小

spring.http.multipart.max-request-size=10Mb # 最大支持请求大小

### 异常处理

@ControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(MultipartException.class)

public String handleError1(MultipartException e, RedirectAttributes redirectAttributes) {

redirectAttributes.addFlashAttribute("message", e.getCause().getMessage());

return "redirect:/uploadStatus";

}

}

设置一个@ControllerAdvice用来监控Multipart上传的文件大小是否受限，当出现此异常时在前端页面给出提示。利用@ControllerAdvice可以做很多东西，比如全局的统一异常处理等。

## WebJars【用的少】

<http://www.webjars.org>

WebJars是一个很神奇的东西，可以让大家以jar包的形式来使用前端的各种框架、组件。

### 什么是WebJars

WebJars是将客户端（浏览器）资源（JavaScript，Css等）打成jar包文件，以对资源进行统一依赖管理。WebJars的jar包部署在Maven中央仓库上。

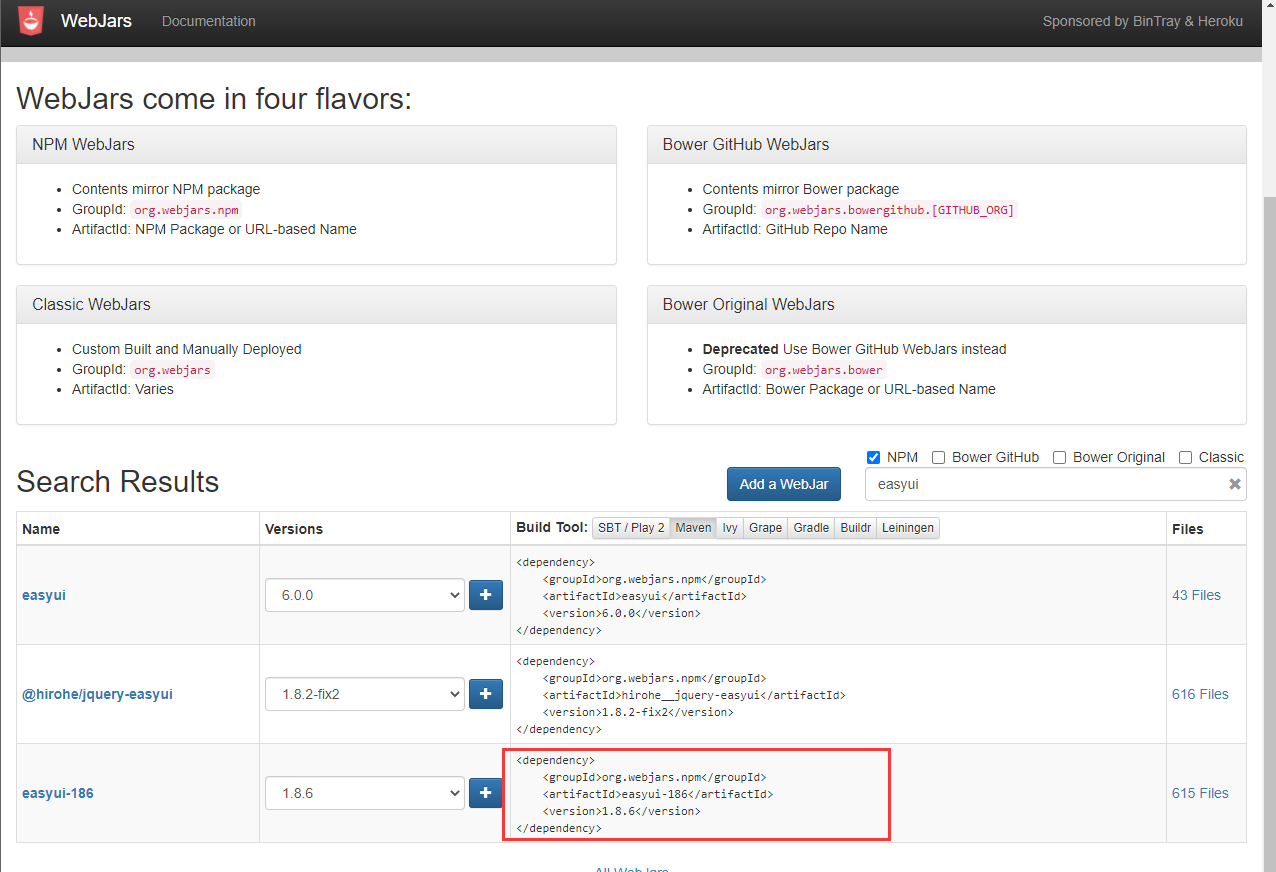
### 为什么使用

我们在开发Java web项目的时候会使用像Maven，Gradle等构建工具以实现对jar包版本依赖管理，以及项目的自动化管理，但是对于JavaScript，Css等前端资源包，

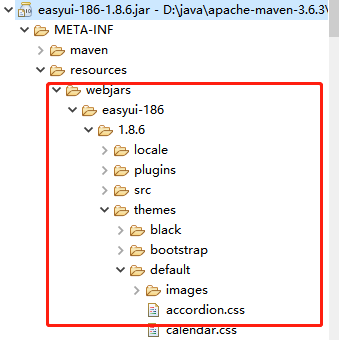
我们只能采用拷贝到webapp下的方式，这样做就无法对这些资源进行依赖管理。那么WebJars就提供给我们这些前端资源的jar包形势，我们就可以进行依赖管理。

### 使用

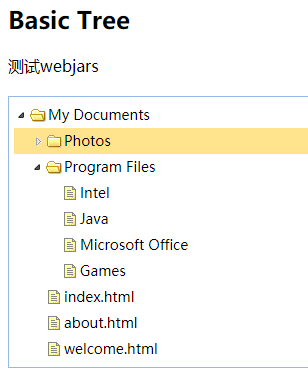
* WebJars主官网 查找对于的组件，比如easyui



* 开发依赖的jar，观察文件路径，其实就是resources的路径。



访问：<http://127.0.0.1:8080/myApp/webjars/index> 页面如下即可



# 常用注解

* @SpringBootApplication

这是 Spring Boot 最最最核心的注解，用在 Spring Boot 主类上，标识这是一个 Spring Boot 应用，用来开启 Spring Boot 的各项能力。

其实这个注解就是 @SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan 这三个注解的组合，也可以用这三个注解来代替 @SpringBootApplication 注解。

* @EnableAutoConfiguration

允许 Spring Boot 自动配置注解，开启这个注解之后，Spring Boot 就能根据当前类路径下的包或者类来配置 Spring Bean。

如：当前类路径下有 Mybatis 这个 JAR 包，MybatisAutoConfiguration 注解就能根据相关参数来配置 Mybatis 的各个 Spring Bean。

* @Configuration

这是 Spring 3.0 添加的一个注解，用来代替 applicationContext.xml 配置文件，所有这个配置文件里面能做到的事情都可以通过这个注解所在类来进行注册。

* @SpringBootConfiguration

这个注解就是 @Configuration 注解的变体，只是用来修饰是 Spring Boot 配置而已，或者可利于 Spring Boot 后续的扩展。

* @ComponentScan

这是 Spring 3.1 添加的一个注解，用来代替配置文件中的 component-scan 配置，开启组件扫描，即自动扫描包路径下的 @Component 注解进行注册 bean 实例到 context 中。

前面 5 个注解可以在这篇文章《Spring Boot 最核心的 3 个注解详解》中了解更多细节的。

* @Conditional

这是 Spring 4.0 添加的新注解，用来标识一个 Spring Bean 或者 Configuration 配置文件，当满足指定的条件才开启配置。

* @ConditionalOnBean

组合 @Conditional 注解，当容器中有指定的 Bean 才开启配置。

* @ConditionalOnMissingBean

组合 @Conditional 注解，和 @ConditionalOnBean 注解相反，当容器中没有指定的 Bean 才开启配置。

* @ConditionalOnClass

组合 @Conditional 注解，当容器中有指定的 Class 才开启配置。

* @ConditionalOnMissingClass

组合 @Conditional 注解，和 @ConditionalOnMissingClass 注解相反，当容器中没有指定的 Class 才开启配置。

* @ConditionalOnWebApplication

组合 @Conditional 注解，当前项目类型是 WEB 项目才开启配置。

当前项目有以下 3 种类型。

enum Type {

/\*\*

\* Any web application will match.

\*/

ANY,

/\*\*

\* Only servlet-based web application will match.

\*/

SERVLET,

/\*\*

\* Only reactive-based web application will match.

\*/

REACTIVE

}

* @ConditionalOnNotWebApplication

组合 @Conditional 注解，和 @ConditionalOnWebApplication 注解相反，当前项目类型不是 WEB 项目才开启配置。

* @ConditionalOnProperty

组合 @Conditional 注解，当指定的属性有指定的值时才开启配置。

* @ConditionalOnExpression

组合 @Conditional 注解，当 SpEL 表达式为 true 时才开启配置。

* @ConditionalOnJava

组合 @Conditional 注解，当运行的 Java JVM 在指定的版本范围时才开启配置。

* @ConditionalOnResource

组合 @Conditional 注解，当类路径下有指定的资源才开启配置。

* @ConditionalOnJndi

组合 @Conditional 注解，当指定的 JNDI 存在时才开启配置。

* @ConditionalOnCloudPlatform

组合 @Conditional 注解，当指定的云平台激活时才开启配置。

* @ConditionalOnSingleCandidate

组合 @Conditional 注解，当指定的 class 在容器中只有一个 Bean，或者同时有多个但为首选时才开启配置。

* @ConfigurationProperties

用来加载额外的配置（如 .properties 文件），可用在 @Configuration 注解类，或者 @Bean 注解方法上面。

关于这个注解的用法可以参考《Spring Boot读取配置的几种方式》这篇文章。

* @EnableConfigurationProperties

一般要配合 @ConfigurationProperties 注解使用，用来开启对 @ConfigurationProperties 注解配置 Bean 的支持。

* @AutoConfigureAfter

用在自动配置类上面，表示该自动配置类需要在另外指定的自动配置类配置完之后。

如 Mybatis 的自动配置类，需要在数据源自动配置类之后。

* @AutoConfigureAfter(DataSourceAutoConfiguration.class)

public class MybatisAutoConfiguration {

* @AutoConfigureBefore

这个和 @AutoConfigureAfter 注解使用相反，表示该自动配置类需要在另外指定的自动配置类配置之前。

* @Import

这是 Spring 3.0 添加的新注解，用来导入一个或者多个 @Configuration 注解修饰的类，这在 Spring Boot 里面应用很多。

* @ImportResource

这是 Spring 3.0 添加的新注解，用来导入一个或者多个 Spring 配置文件，这对 Spring Boot 兼容老项目非常有用，因为有些配置无法通过 Java Config 的形式来配置就只能用这个注解来导入。

# 缓存

## 简单使用

### 引入缓存

在pom.xml中引入cache依赖，添加如下内容：

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

### 创建配置类

在Spring Boot主类中增加@EnableCaching注解开启缓存功能，如下：

|  |
| --- |
| @SpringBootApplication  @EnableCaching  public class Application {  public static void main(String[] args) {  SpringApplication.run(Application.class, args);  }  } |

### 使用配置

在数据访问接口中，增加缓存配置注解，如：

|  |
| --- |
| @CacheConfig(cacheNames = "users")  public interface UserRepository extends JpaRepository<User, Long> {  @Cacheable  User findByName(String name);  } |

再来执行以下单元测试，可以在控制台中输出了下面的内容：

|  |
| --- |
| Hibernate: insert into user (age, name) values (?, ?)  Hibernate: select user0\_.id as id1\_0\_, user0\_.age as age2\_0\_, user0\_.name as name3\_0\_ from user user0\_ where user0\_.name=?  第一次查询：10  第二次查询：10 |

到这里，我们可以看到，在调用第二次findByName函数时，没有再执行select语句，也就直接减少了一次数据库的读取操作。

为了可以更好的观察，缓存的存储，我们可以在单元测试中注入cacheManager。

|  |
| --- |
| @Autowired  private CacheManager cacheManager; |

使用debug模式运行单元测试，观察cacheManager中的缓存集users以及其中的User对象的缓存加深理解。

## Cache注解详解

回过头来我们再来看，这里使用到的两个注解分别作了什么事情。

### @CacheConfig

主要用于配置该类中会用到的一些共用的缓存配置。在这里@CacheConfig(cacheNames = "users")：配置了该数据访问对象中返回的内容将存储于名为users的缓存对象中，我们也可以不使用该注解，直接通过@Cacheable自己配置缓存集的名字来定义。

### @Cacheable

配置了findByName函数的返回值将被加入缓存。同时在查询时，会先从缓存中获取，若不存在才再发起对数据库的访问。该注解主要有下面几个参数：

value、cacheNames：两个等同的参数（cacheNames为Spring 4新增，作为value的别名），用于指定缓存存储的集合名。由于Spring 4中新增了@CacheConfig，因此在Spring 3中原本必须有的value属性，也成为非必需项了

key：缓存对象存储在Map集合中的key值，非必需，缺省按照函数的所有参数组合作为key值，若自己配置需使用SpEL表达式，比如：@Cacheable(key = "#p0")：使用函数第一个参数作为缓存的key值，更多关于SpEL表达式的详细内容可参考[官方文档](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html#cache-spel-context)

condition：缓存对象的条件，非必需，也需使用SpEL表达式，只有满足表达式条件的内容才会被缓存，比如：@Cacheable(key = "#p0", condition = "#p0.length() < 3")，表示只有当第一个参数的长度小于3的时候才会被缓存，若做此配置上面的AAA用户就不会被缓存，读者可自行实验尝试。

unless：另外一个缓存条件参数，非必需，需使用SpEL表达式。它不同于condition参数的地方在于它的判断时机，该条件是在函数被调用之后才做判断的，所以它可以通过对result进行判断。

keyGenerator：用于指定key生成器，非必需。若需要指定一个自定义的key生成器，我们需要去实现org.springframework.cache.interceptor.KeyGenerator接口，并使用该参数来指定。需要注意的是：该参数与key是互斥的

cacheManager：用于指定使用哪个缓存管理器，非必需。只有当有多个时才需要使用

cacheResolver：用于指定使用那个缓存解析器，非必需。需通过org.springframework.cache.interceptor.CacheResolver接口来实现自己的缓存解析器，并用该参数指定。

除了这里用到的两个注解之外，还有下面几个核心注解：

### @CachePut

配置于函数上，执行方法体 - 将结果缓存起来。它与@Cacheable不同的是，它每次都会真是调用函数，所以主要用于数据新增和修改操作上。它的参数与@Cacheable类似，具体功能可参考上面对@Cacheable参数的解析

### @CacheEvict

配置于函数上，通常用在删除方法上，用来从缓存中移除相应数据。除了同@Cacheable一样的参数之外，它还有下面两个参数：

allEntries：非必需，默认为false。当为true时，会移除所有数据

beforeInvocation：非必需，默认为false，会在调用方法之后移除数据。当为true时，会在调用方法之前移除数据。

## 缓存配置

完成了上面的缓存实验之后，可能大家会问，那我们在Spring Boot中到底使用了什么缓存呢？

在Spring Boot中通过@EnableCaching注解自动化配置合适的缓存管理器（CacheManager），Spring Boot根据下面的顺序去侦测缓存提供者：

1. Generic
2. JCache (JSR-107)
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Redis
7. Guava
8. Simple
9. Caffeine

除了按顺序侦测外，我们也可以通过配置属性spring.cache.type来强制指定。我们可以通过debug调试查看cacheManager对象的实例来判断当前使用了什么缓存。

本文中不对所有的缓存做详细介绍，下面以常用的EhCache为例，看看如何配置来使用EhCache进行缓存管理。

在Spring Boot中开启EhCache非常简单，只需要在工程中加入ehcache.xml配置文件并在pom.xml中增加ehcache依赖，框架只要发现该文件，就会创建EhCache的缓存管理器。

1. 在src/main/resources目录下创建：ehcache.xml

|  |
| --- |
| <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="ehcache.xsd">  <cache name="users"  maxEntriesLocalHeap="200"  timeToLiveSeconds="600">  </cache>  </ehcache> |

1. 在pom.xml中加入

|  |
| --- |
| <dependency>  <groupId>net.sf.ehcache</groupId>  <artifactId>ehcache</artifactId>  </dependency> |

完成上面的配置之后，再通过debug模式运行单元测试，观察此时CacheManager已经是EhCacheManager实例，说明EhCache开启成功了。

对于EhCache的配置文件也可以通过application.properties文件中使用spring.cache.ehcache.config属性来指定，比如：

spring.cache.ehcache.config=classpath:config/another-config.xml

## 设置过期时间

Spring Cache不能设置过期时间。CacheManager类只提供了getCache与getCacheNames两个方法。这是因为，spring对接了很多类缓存的厂商，这些框架对过期时间的操作不尽相同，不能简单的使用一个expire去表示。

下面我们使用Redis cache为例子：

@Configuration

@EnableCaching

public class RedisCacheConfig extends CachingConfigurerSupport {

@Value("${spring.redis.host}")

private String redisHost;

@Value("${spring.redis.port}")

private int redisPort;

@Value("${spring.redis.timeout}")

private int redisTimeout;

@Value("${spring.redis.password}")

private String redisAuth;

@Value("${spring.redis.database}")

private int redisDb;

@Value("${spring.redis.pool.max-active}")

private int maxActive;

@Value("${spring.redis.pool.max-wait}")

private int maxWait;

@Value("${spring.redis.pool.max-idle}")

private int maxIdle;

@Value("${spring.redis.pool.min-idle}")

private int minIdle;

@Bean

@Override

public KeyGenerator keyGenerator() {

return new KeyGenerator() {

@Override

public Object generate(Object target, Method method, Object... params) {

StringBuilder sb = new StringBuilder();

sb.append(target.getClass().getName());

sb.append(method.getName());

for (Object obj : params) {

sb.append(obj.toString());

}

return sb.toString();

}

};

}

@Bean

public CacheManager redisCacheConfig() {

RedisCacheManager redisCacheManager = new RedisCacheManager(this.redisTemplate());

// 是否使用前缀生成器

redisCacheManager.setTransactionAware(true);

// 启动时加载远程缓存

redisCacheManager.setLoadRemoteCachesOnStartup(true);

//设置默认的过期时间,60秒

redisCacheManager.setDefaultExpiration(60);

// 对某个key设置过期时间

Map<String, Long> expires = new HashMap<>();

expires.put("cache:demo:k1", 1000L);

expires.put("cache:demo:k2", 2000L);

redisCacheManager.setExpires(expires);

return redisCacheManager;

}

@Bean

public RedisConnectionFactory redisConnectionFactory() {

JedisPoolConfig poolConfig = new JedisPoolConfig();

poolConfig.setMaxTotal(maxActive);

poolConfig.setMaxIdle(maxIdle);

poolConfig.setMaxWaitMillis(maxWait);

poolConfig.setMinIdle(minIdle);

poolConfig.setTestOnBorrow(true);

poolConfig.setTestOnReturn(false);

poolConfig.setTestWhileIdle(true);

JedisConnectionFactory jedisConnectionFactory = new JedisConnectionFactory(poolConfig);

jedisConnectionFactory.setPassword(redisAuth);

jedisConnectionFactory.setHostName(redisHost);

jedisConnectionFactory.setDatabase(redisDb);

jedisConnectionFactory.setPort(redisPort);

jedisConnectionFactory.setTimeout(redisTimeout);

return jedisConnectionFactory;

}

@Bean

public RedisTemplate<String, Object> redisTemplate() {

RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();

Jackson2JsonRedisSerializer<Object> serializer = jackson2JsonRedisSerializer();

redisTemplate.setConnectionFactory(redisConnectionFactory());

redisTemplate.setKeySerializer(new StringRedisSerializer());

redisTemplate.setValueSerializer(serializer);

redisTemplate.setHashKeySerializer(new StringRedisSerializer());

redisTemplate.setHashValueSerializer(serializer);

return redisTemplate;

}

@Bean

public Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer() {

final Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer<>(Object.class);

final ObjectMapper objectMapper = Jackson2ObjectMapperBuilder

.json().build();

objectMapper.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);

objectMapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON\_FINAL);

jackson2JsonRedisSerializer.setObjectMapper(objectMapper);

return jackson2JsonRedisSerializer;

}

@Bean

public JedisPool redisPoolFactory() throws Exception {

JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();

jedisPoolConfig.setMaxIdle(maxIdle);

jedisPoolConfig.setMaxWaitMillis(maxWait);

// 连接耗尽时是否阻塞, false报异常,ture阻塞直到超时, 默认true

jedisPoolConfig.setBlockWhenExhausted(true);

// 是否启用pool的jmx管理功能, 默认true

jedisPoolConfig.setJmxEnabled(true);

JedisPool jedisPool = new JedisPool(jedisPoolConfig, redisHost, redisPort, redisTimeout, redisAuth);

return jedisPool;

}

}

## 自定义

SpringBoot自带Cache存在问题：

1.生成Key过于简单，容易冲突 默认为cacheNames + “:” + Key

2.无法设置过期时间，默认时间是永久

3.配置序列化方式，默认是JDKSerializable，可能造成乱码

自定义Cache分三个步骤：

1.自定义KeyGenerator

2.自定义cacheManager 设置缓存过期时间

3.自定义序列化方式 JackSon

# 数据库操作

支持jdbcTemplate、spring data jpa等

# 监控管理

## Spring Boot Actuator

Spring Boot 2.0官方文档之 Actuator: <https://blog.csdn.net/alinyua/article/details/80009435>

官方文档：<https://docs.spring.io/spring-boot/docs/2.4.0/actuator-api/htmlsingle/>

简单教程： <https://www.cnblogs.com/architectforest/p/13555589.html>

## Spring Boot Admin

用于监控基于 Spring Boot 的应用，它是在 Spring Boot Actuator 的基础上提供简洁的可视化 WEB UI。

### 简介

Spring Boot Admin 提供了很多功能，如显示 name、id 和 version，显示在线状态，Loggers 的日志级别管理，Threads 线程管理，Environment 管理等。

### Spring Boot Admin 是由服务端和客户端组成

在 Spring Boot 项目中，Spring Boot Admin 作为 Server 端，其他的要被监控的应用作为 Client 端，基于这种的配置如下步骤：

### Server

#### 添加相关依赖

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-server</artifactId>

<version>2.0.1-SNAPSHOT</version>

</dependency>

<dependency>

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

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

</dependency>

#### 启动类添加注解，开启监控

@Configuration

@EnableAutoConfiguration

@EnableAdminServer

public class SpringBootAdminApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootAdminApplication.class, args);

}

}

#### 配置文件

server:

port: 8788

### Client

#### 添加相关依赖

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-client</artifactId>

<version>2.0.0</version>

</dependency>

#### 配置文件

spring.boot.admin.client.url: "http://localhost:8788"

management.endpoints.web.exposure.include: "\*"

以上的配置，就可以实现 Spring Boot 项目中 Spring Boot Admin 监控其他应用了，但是这不是我们的重点，详细信息请参考官网文档：http://codecentric.github.io/spring-boot-admin/2.0.0/，我们的重点是在 Spring Cloud 中使用 Spring Boot Admin 监控 Spring Cloud 的服务，下面我们就详细的讲解在 Spring Cloud 中搭建 Spring Boot Admin

### 邮件告警

Spring Boot Admin将微服务中所有应用信息在后台进行了展示，非常方便我们对微服务整体的监控和治理。但是我们的运营人员也不可能一天24小时盯着监控后台，因此如果服务有异常的时候，有对应的邮件告警就太好了，其实Spring Boot Admin也给出了支持。

我们对上面的示例项目spring-boot-admin-server进行改造。

#### 添加依赖

<dependency>

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

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

</dependency>

增加了邮件发送的starter包

#### 配置文件

spring:

mail:

host: smtp.qq.com

username: xxxxx@qq.com

password: xxxx

properties:

mail:

smtp:

auth: true

starttls:

enable: true

required: true

boot:

admin:

notify:

mail:

from: xxxx@qq.com

to: xxxx@qq.com

# http://codecentric.github.io/spring-boot-admin/1.5.6/#mail-notifications

在配置文件中添加邮件发送相关信息：邮件的发送者、接受者、协议、移动授权码等。

配置完成后，重新启动项目spring-boot-admin-server，这样Admin Server就具备了邮件告警的功能，默认情况下Admin Server对Eureka中的服务上下线都进行了监控，当服务上下线的时候我们就会收到如下邮件：

当然这只是最基本的邮件监控，在实际的使用过程中，需要根据我们的情况对邮件告警内容进行自定义，比如监控堆内存的使用情况，当到达一定比例的时候进行告警等。

## 在 Spring Cloud 中基于 Eureka 的 Spring Boot Admin 的搭建

### 启动之前的项目 eureka server，端口8761

### 新建 module（springboot-admin），创建步骤参考上篇

### 添加相关依赖，pom文件：

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-starter-server</artifactId>

<version>2.0.1-SNAPSHOT</version>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.jolokia</groupId>

<artifactId>jolokia-core</artifactId>

</dependency>

由于项目中使用的是 spring boot 2.0 版本，所以这里要使用 spring boot admin 的最新版本（还未正式发布），本人测试使用 spring boot admin 2.0.0 版本会有问题。

### 启动类添加注解

@Configuration

@EnableAutoConfiguration

@EnableAdminServer

@EnableEurekaClient

public class SpringBootAdminApplication {

public static void main(String[] args) {

SpringApplication.run(SpringBootAdminApplication.class, args);

}

@Profile("insecure")

@Configuration

public static class SecurityPermitAllConfig extends WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests().anyRequest().permitAll()//

.and().csrf().disable();

}

}

@Profile("secure")

@Configuration

public static class SecuritySecureConfig extends WebSecurityConfigurerAdapter {

private final String adminContextPath;

public SecuritySecureConfig(AdminServerProperties adminServerProperties) {

this.adminContextPath = adminServerProperties.getContextPath();

}

@Override

protected void configure(HttpSecurity http) throws Exception {

// @formatter:off

SavedRequestAwareAuthenticationSuccessHandler successHandler = new SavedRequestAwareAuthenticationSuccessHandler();

successHandler.setTargetUrlParameter("redirectTo");

http.authorizeRequests()

.antMatchers(adminContextPath + "/assets/\*\*").permitAll()

.antMatchers(adminContextPath + "/login").permitAll()

.anyRequest().authenticated()

.and()

.formLogin().loginPage(adminContextPath + "/login").successHandler(successHandler).and()

.logout().logoutUrl(adminContextPath + "/logout").and()

.httpBasic().and()

.csrf().disable();

// @formatter:on

}

}

}

SecurityPermitAllConfig和SecuritySecureConfig的配置是 Spring Boot Admin 官方给的配置，是对 url 进行安全认证等配置，照着配置即可。@EnableEurekaClient 注解是把 Spring Boot Admin 注册到 Eureka 里，这样 Spring Boot Admin 就可以发现注册到 Eureka 里的其他服务实例，@EnableAdminServer 注解是开启监控功能。

### 配置文件

官方有给出示例，主要是配置 eureka 的地址，这里要强调说明的一点，由于 Spring Boot 2.0 的 Actuator 只暴露了 /health、/info 两个端口（为了安全考虑），所以要配置 management.endpoints.web.exposure.include 的属性，下面的配置文件中暴力了一点，配置暴露了所有的端点，由于 Spring Boot Admin 有 web UI 管理界面，配置了登录的用户名密码如下，使用了 security.user 的属性，其他的详细配置，可以查看官方文档。

spring:

application:

name: spring-boot-admin

profiles:

active:

- secure

server:

port: 8788

# tag::configuration-eureka[]

eureka: #<1>

instance:

leaseRenewalIntervalInSeconds: 10

health-check-url-path: /actuator/health

client:

registryFetchIntervalSeconds: 5

serviceUrl:

defaultZone: ${EUREKA\_SERVICE\_URL:http://localhost:8761}/eureka/

management:

endpoints:

web:

exposure:

include: "\*" #<2>

endpoint:

health:

show-details: ALWAYS

# end::configuration-eureka[]

---

spring:

profiles: insecure

---

spring:

profiles: secure

security:

user:

name: "user"

password: "password"

eureka:

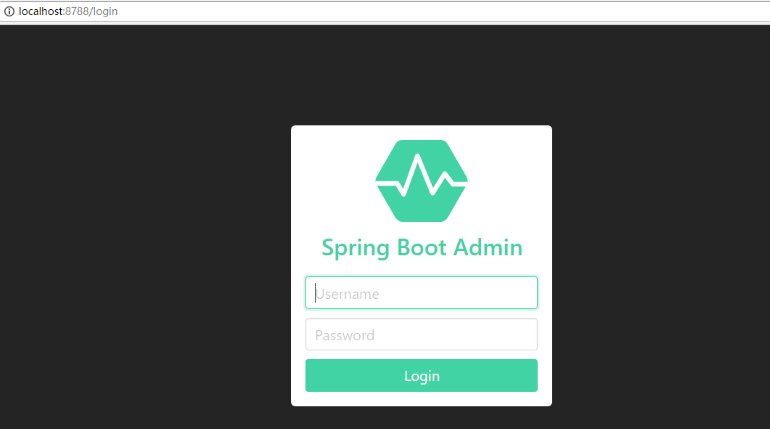
instance:

metadata-map:

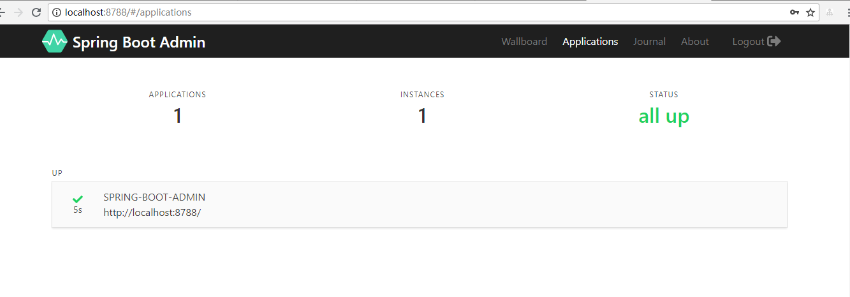
user.name: "user" #These two are needed so that the server

user.password: "password" #can access the protected client endpoints

启动 spring boot admin 服务，界面如下：



用户名密码即上面的配置，user/password登录成功后



此时由于 Eureka 里只有 Spring Boot Admin 自身已注册，所以其监控列表里只有自己，下面我们启动其他的服务，让其注册到 Eureka 里。

### 启动 spring-demo-service 服务

在启动前，Actuator 在 spring boot 2.0 版本后，只暴露了两个端点，所以此时启动，监控不到所需的信息，下面修改配置文件如下：

server:

port: 8281

eureka:

client:

serviceUrl:

# 向每个注册中心注册

defaultZone: http://localhost:8761/eureka/,http://localhost:8762/eureka/

spring:

application:

name: spring-demo-service

management:

endpoints:

web:

exposure:

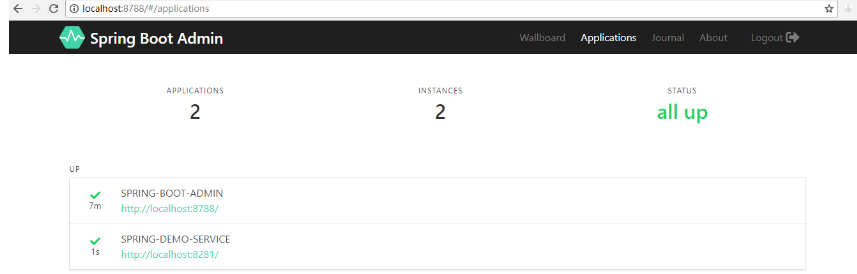
include: '\*'

endpoint:

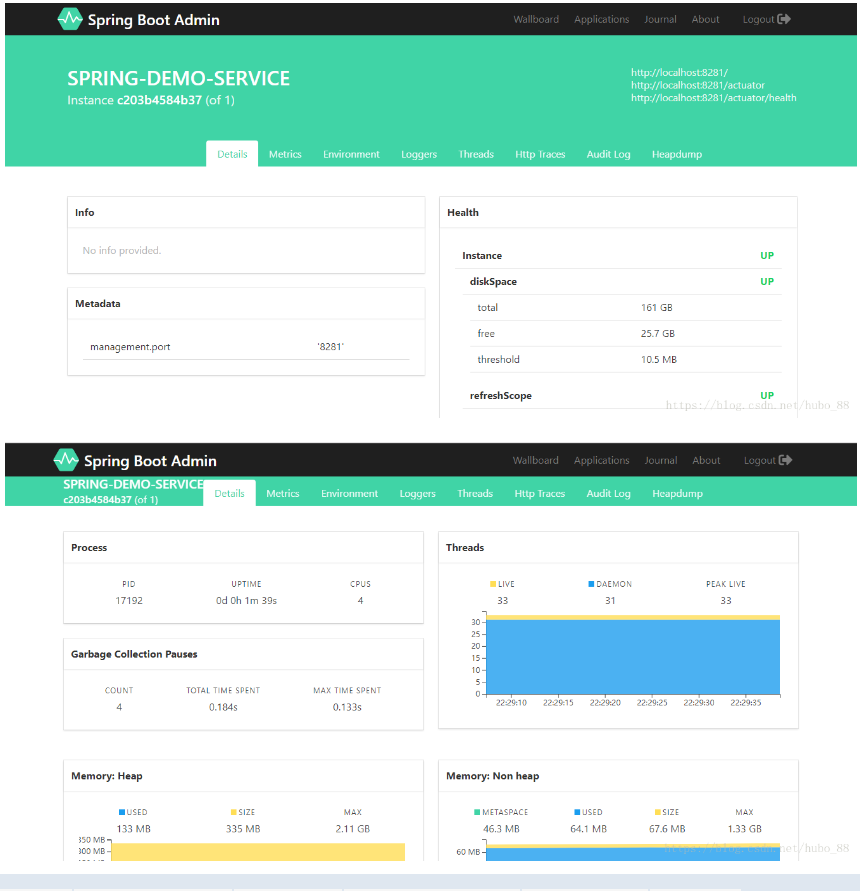
health:

show-details: ALWAYS

此时启动 spring-demo-service，发现监控列表里 spring-demo-service 已经有了



点击 SPRING-DEMO-SERVICE 进入，页面如下，可以看到有我们之前介绍的一些功能。



### 其他的配置

如果想要显示版本信息，配置文件中加入 info.version=1.0.0 可以配置版本信息

# 异步调用

异步调用实现方式：<https://mp.weixin.qq.com/s/N8QBiH-nCSEQKcbuamv42A>

“异步调用”对应的是“同步调用”，同步调用指程序按照定义顺序依次执行，每一行程序都必须等待上一行程序执行完成之后才能执行；异步调用指程序在顺序执行时，不等待异步调用的语句返回结果就执行后面的程序。

## 同步调用

下面通过一个简单示例来直观的理解什么是同步调用：

1.定义Task类，创建三个处理函数分别模拟三个执行任务的操作，操作消耗时间随机取（10秒内）

|  |
| --- |
| @Component  public class Task {  public static Random random =new Random();  public void doTaskOne() throws Exception {  System.out.println("开始做任务一");  long start = System.currentTimeMillis();  Thread.sleep(random.nextInt(10000));  long end = System.currentTimeMillis();  System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");  }  public void doTaskTwo() throws Exception {  System.out.println("开始做任务二");  long start = System.currentTimeMillis();  Thread.sleep(random.nextInt(10000));  long end = System.currentTimeMillis();  System.out.println("完成任务二，耗时：" + (end - start) + "毫秒");  }  public void doTaskThree() throws Exception {  System.out.println("开始做任务三");  long start = System.currentTimeMillis();  Thread.sleep(random.nextInt(10000));  long end = System.currentTimeMillis();  System.out.println("完成任务三，耗时：" + (end - start) + "毫秒");  }  } |

2.在单元测试用例中，注入Task对象，并在测试用例中执行doTaskOne、doTaskTwo、doTaskThree三个函数。

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @SpringApplicationConfiguration(classes = Application.class)  public class ApplicationTests {  @Autowired  private Task task;  @Test  public void test() throws Exception {  task.doTaskOne();  task.doTaskTwo();  task.doTaskThree();  }  } |

执行单元测试，可以看到类似如下输出：

|  |
| --- |
| 开始做任务一  完成任务一，耗时：4256毫秒  开始做任务二  完成任务二，耗时：4957毫秒  开始做任务三  完成任务三，耗时：7173毫秒 |

任务一、任务二、任务三顺序的执行完了，换言之doTaskOne、doTaskTwo、doTaskThree三个函数顺序的执行完成。

## 异步调用

上述的同步调用虽然顺利的执行完了三个任务，但是可以看到执行时间比较长，若这三个任务本身之间不存在依赖关系，

可以并发执行的话，同步调用在执行效率方面就比较差，可以考虑通过异步调用的方式来并发执行。

在Spring Boot中，我们只需要通过使用@Async注解就能简单的将原来的同步函数变为异步函数，Task类改在为如下模式：

|  |
| --- |
| @Component  public class Task {  @Async  public void doTaskOne() throws Exception {  // 同上内容，省略  }  @Async  public void doTaskTwo() throws Exception {  // 同上内容，省略  }  @Async  public void doTaskThree() throws Exception {  // 同上内容，省略  }  } |

为了让@Async注解能够生效，还需要在Spring Boot的主程序中配置@EnableAsync，如下所示：

|  |
| --- |
| @SpringBootApplication  @EnableAsync  public class Application {  public static void main(String[] args) {  SpringApplication.run(Application.class, args);  }  } |

此时可以反复执行单元测试，您可能会遇到各种不同的结果，比如：

没有任何任务相关的输出

有部分任务相关的输出

乱序的任务相关的输出

原因是目前doTaskOne、doTaskTwo、doTaskThree三个函数的时候已经是异步执行了。主程序在异步调用之后，主程序并不会理会这三个函数是否执行完成了，由于没有其他需要执行的内容，所以程序就自动结束了，导致了不完整或是没有输出任务相关内容的情况。

注： @Async所修饰的函数不要定义为static类型，这样异步调用不会生效

## 异步回调

为了让doTaskOne、doTaskTwo、doTaskThree能正常结束，假设我们需要统计一下三个任务并发执行共耗时多少，这就需要等到上述三个函数都完成调动之后记录时间，并计算结果。

那么我们如何判断上述三个异步调用是否已经执行完成呢？我们需要使用Future<T>来返回异步调用的结果，就像如下方式改造doTaskOne函数：

|  |
| --- |
| @Async  public Future<String> doTaskOne() throws Exception {  System.out.println("开始做任务一");  long start = System.currentTimeMillis();  Thread.sleep(random.nextInt(10000));  long end = System.currentTimeMillis();  System.out.println("完成任务一，耗时：" + (end - start) + "毫秒");  return new AsyncResult<>("任务一完成");  } |

按照如上方式改造一下其他两个异步函数之后，下面我们改造一下测试用例，让测试在等待完成三个异步调用之后来做一些其他事情。

|  |
| --- |
| @Test  public void test() throws Exception {  long start = System.currentTimeMillis();  Future<String> task1 = task.doTaskOne();  Future<String> task2 = task.doTaskTwo();  Future<String> task3 = task.doTaskThree();  while(true) {  if(task1.isDone() && task2.isDone() && task3.isDone()) {  // 三个任务都调用完成，退出循环等待  break;  }  Thread.sleep(1000);  }  long end = System.currentTimeMillis();  System.out.println("任务全部完成，总耗时：" + (end - start) + "毫秒");  } |

看看我们做了哪些改变：

在测试用例一开始记录开始时间

在调用三个异步函数的时候，返回Future<String>类型的结果对象

在调用完三个异步函数之后，开启一个循环，根据返回的Future<String>对象来判断三个异步函数是否都结束了。若都结束，就结束循环；若没有都结束，就等1秒后再判断。

跳出循环之后，根据结束时间 - 开始时间，计算出三个任务并发执行的总耗时。

执行一下上述的单元测试，可以看到如下结果：

|  |
| --- |
| 开始做任务一  开始做任务二  开始做任务三  完成任务三，耗时：37毫秒  完成任务二，耗时：3661毫秒  完成任务一，耗时：7149毫秒  任务全部完成，总耗时：8025毫秒 |

可以看到，通过异步调用，让任务一、二、三并发执行，有效的减少了程序的总运行时间。

# 日志管理

Spring Boot在所有内部日志中使用Commons Logging，但是默认配置也提供了对常用日志的支持，如：Java Util Logging，Log4J, Log4J2和Logback。

每种Logger都可以通过配置使用控制台或者文件输出日志内容。

## spring boot 默认日志

### 格式化日志

默认的日志输出如下：

2016-04-13 08:23:50.120 INFO 37397 --- [ main] org.hibernate.Version : HHH000412: Hibernate Core {4.3.11.Final}

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 or 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（可能会截断控制台输出）

Logger名 — 通常使用源代码的类名

日志内容

### 控制台输出

在Spring Boot中默认配置了ERROR、WARN和INFO级别的日志输出到控制台。

我们可以通过两种方式切换至DEBUG级别：

在运行命令后加入--debug标志，如：$ java -jar myapp.jar --debug

在application.properties中配置debug=true，该属性置为true的时候，核心Logger（包含嵌入式容器、hibernate、spring）会输出更多内容，但是你自己应用的日志并不会输出为DEBUG级别。

### 多彩输出

如果你的终端支持ANSI，设置彩色输出会让日志更具可读性。通过在application.properties中设置spring.output.ansi.enabled参数来支持。

NEVER：禁用ANSI-colored输出（默认项）

DETECT：会检查终端是否支持ANSI，是的话就采用彩色输出（推荐项）

ALWAYS：总是使用ANSI-colored格式输出，若终端不支持的时候，会有很多干扰信息，不推荐使用

### 文件输出

Spring Boot默认配置只会输出到控制台，并不会记录到文件中，但是我们通常生产环境使用时都需要以文件方式记录。

若要增加文件输出，需要在application.properties中配置logging.file或logging.path属性。

logging.file，设置文件，可以是绝对路径，也可以是相对路径。如：logging.file=my.log

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

日志文件会在10Mb大小的时候被截断，产生新的日志文件，默认级别为：ERROR、WARN、INFO

### 级别控制

在Spring Boot中只需要在application.properties中进行配置完成日志记录的级别控制。

配置格式：logging.level.\*=LEVEL

logging.level：日志级别控制前缀，\*为包名或Logger名

LEVEL：选项TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

举例：

logging.level.com.didispace=DEBUG：com.didispace包下所有class以DEBUG级别输出

logging.level.root=WARN：root日志以WARN级别输出

### 自定义日志配置

由于日志服务一般都在ApplicationContext创建前就初始化了，它并不是必须通过Spring的配置文件控制。因此通过系统属性和传统的Spring Boot外部配置文件依然可以很好的支持日志控制和管理。

根据不同的日志系统，你可以按如下规则组织配置文件名，就能被正确加载：

Logback：logback-spring.xml, logback-spring.groovy, logback.xml, logback.groovy

Log4j：log4j-spring.properties, log4j-spring.xml, log4j.properties, log4j.xml

Log4j2：log4j2-spring.xml, log4j2.xml

JDK (Java Util Logging)：logging.properties

Spring Boot官方推荐优先使用带有-spring的文件名作为你的日志配置（如使用logback-spring.xml，而不是logback.xml）

### 自定义输出格式

在Spring Boot中可以通过在application.properties配置如下参数控制输出格式：

logging.pattern.console：定义输出到控制台的样式（不支持JDK Logger）

logging.pattern.file：定义输出到文件的样式（不支持JDK Logger）

### 常用配置

配置在application.properties中

logging.path=/user/local/log

logging.level.com.favorites=DEBUG

logging.level.org.springframework.web=INFO

logging.level.org.hibernate=ERROR

## 使用log4J

### 引入log4j依赖

在创建Spring Boot工程时，我们引入了spring-boot-starter，其中包含了spring-boot-starter-logging，该依赖内容就是Spring Boot默认的日志框架Logback，所以我们在引入log4j之前，需要先排除该包的依赖，再引入log4j的依赖，就像下面这样：

<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>

</dependency>

### 配置log4j.properties

在引入了log4j依赖之后，只需要在src/main/resources目录下加入log4j.properties配置文件，就可以开始对应用的日志进行配置使用。

控制台输出，通过如下配置，设定root日志的输出级别为INFO，appender为控制台输出stdout

|  |
| --- |
| # LOG4J配置  log4j.rootCategory=INFO, stdout  # 控制台输出  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n |

### 输出到文件

在开发环境，我们只是输出到控制台没有问题，但是到了生产或测试环境，或许持久化日志内容，方便追溯问题原因。

可以通过添加如下的appender内容，按天输出到不同的文件中去，同时还需要为log4j.rootCategory添加名为file的appender，这样root日志就可以输出到logs/all.log文件中了。

|  |
| --- |
| #  log4j.rootCategory=INFO, stdout, file  # root日志输出  log4j.appender.file=org.apache.log4j.DailyRollingFileAppender  log4j.appender.file.file=logs/all.log  log4j.appender.file.DatePattern='.'yyyy-MM-dd  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n |

### 分类输出

当我们日志量较多的时候，查找问题会非常困难，常用的手段就是对日志进行分类，比如：

可以按不同package进行输出。通过定义输出到logs/my.log的appender，并对com.didispace包下的日志级别设定为DEBUG级别、appender设置为输出到logs/my.log的名为didifile的appender。

|  |
| --- |
| # com.didispace包下的日志配置  log4j.category.com.didispace=DEBUG, didifile  # com.didispace下的日志输出  log4j.appender.didifile=org.apache.log4j.DailyRollingFileAppender  log4j.appender.didifile.file=logs/my.log  log4j.appender.didifile.DatePattern='.'yyyy-MM-dd  log4j.appender.didifile.layout=org.apache.log4j.PatternLayout  log4j.appender.didifile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L ---- %m%n |

可以对不同级别进行分类，比如对ERROR级别输出到特定的日志文件中，具体配置可以如下。

|  |
| --- |
| log4j.logger.error=errorfile  # error日志输出  log4j.appender.errorfile=org.apache.log4j.DailyRollingFileAppender  log4j.appender.errorfile.file=logs/error.log  log4j.appender.errorfile.DatePattern='.'yyyy-MM-dd  log4j.appender.errorfile.Threshold = ERROR  log4j.appender.errorfile.layout=org.apache.log4j.PatternLayout  log4j.appender.errorfile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m%n |

本文主要介绍如何在spring boot中引入log4j，以及一些基础用法，对于更多log4j的用法，还请参考[log4j官方网站](http://logging.apache.org/log4j/1.2/)

## 统一处理web请求日志

### 引入依赖：

<dependency>

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

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

</dependency>

### 配置

在完成了引入AOP依赖包后，一般来说并不需要去做其他配置。也许在Spring中使用过注解配置方式的人会问是否需要在程序主类中增加@EnableAspectJAutoProxy来启用，实际并不需要。

可以看下面关于AOP的默认配置属性，其中spring.aop.auto属性默认是开启的，也就是说只要引入了AOP依赖后，默认已经增加了@EnableAspectJAutoProxy。而当我们需要使用CGLIB来实现AOP的时候，需要配置spring.aop.proxy-target-class=true，不然默认使用的是标准Java的实现。

|  |
| --- |
| # AOP  spring.aop.auto=true # Add @EnableAspectJAutoProxy.  spring.aop.proxy-target-class=false # Whether subclass-based (CGLIB) proxies are to be created (true) as  opposed to standard Java interface-based proxies (false). |

### 写一个aop

@Aspect

@Service

public class LoggerAdvice {

private Logger logger = Logger.getLogger(this.getClass());

@Before("within(com.zx..\*) && @annotation(loggerManage)")

public void addBeforeLogger(JoinPoint joinPoint, LoggerManage loggerManage) {

logger.info("执行 " + loggerManage.description() + " 开始");

logger.info(joinPoint.getSignature().toString());

logger.info(parseParames(joinPoint.getArgs()));

}

@AfterReturning("within(com.zx..\*) && @annotation(loggerManage)")

public void addAfterReturningLogger(JoinPoint joinPoint, LoggerManage loggerManage) {

logger.info("执行 " + loggerManage.description() + " 结束");

}

@AfterThrowing(pointcut = "within(com.zx..\*) && @annotation(loggerManage)", throwing = "ex")

public void addAfterThrowingLogger(JoinPoint joinPoint, LoggerManage loggerManage, Exception ex) {

logger.error("执行 " + loggerManage.description() + " 异常", ex);

}

private String parseParames(Object[] parames) {

if (null == parames || parames.length <= 0) {

return "";

}

StringBuffer param = new StringBuffer("传入参数[{}] ");

for (Object obj : parames) {

param.append(obj.toString()).append(" ");

}

return param.toString();

}

}

定义拦截规则：拦截com.xjj.web.controller包下面的所有类中，有@RequestMapping注解的方法。

@Pointcut("execution(\* com.xjj.web.controller..\*(..)) and @annotation(org.springframework.web.bind.annotation.RequestMapping)")

/\*\*

\* 拦截器具体实现

\* @param pjp

\* @return JsonResult（被拦截方法的执行结果，或需要登录的错误提示。）

\*/

@Around("controllerMethodPointcut()") //指定拦截器规则；也可以直接

实现AOP的切面主要有以下几个要素：

1. 使用@Aspect注解将一个java类定义为切面类
2. 使用@Pointcut定义一个切入点，可以是一个规则表达式，比如下例中某个package下的所有函数，也可以是一个注解等。
3. 根据需要在切入点不同位置的切入内容
   1. 使用@Before在切入点开始处切入内容
   2. 使用@After在切入点结尾处切入内容
   3. 使用@AfterReturning在切入点return内容之后切入内容（可以用来对处理返回值做一些加工处理）
   4. 使用@Around在切入点前后切入内容，并自己控制何时执行切入点自身的内容
   5. 使用@AfterThrowing用来处理当切入内容部分抛出异常之后的处理逻辑

### 优化

优化：AOP切面中的同步问题

在WebLogAspect切面中，分别通过doBefore和doAfterReturning两个独立函数实现了切点头部和切点返回后执行的内容，若我们想统计请求的处理时间，就需要在doBefore处记录时间，并在doAfterReturning处通过当前时间与开始处记录的时间计算得到请求处理的消耗时间。

那么我们是否可以在WebLogAspect切面中定义一个成员变量来给doBefore和doAfterReturning一起访问呢？是否会有同步问题呢？

的确，直接在这里定义基本类型会有同步问题，所以我们可以引入ThreadLocal对象，像下面这样进行记录：

|  |
| --- |
| @Aspect  @Component  public class WebLogAspect {  private Logger logger = Logger.getLogger(getClass());  ThreadLocal<Long> startTime = new ThreadLocal<>();  @Pointcut("execution(public \* com.didispace.web..\*.\*(..))")  public void webLog(){}  @Before("webLog()")  public void doBefore(JoinPoint joinPoint) throws Throwable {  startTime.set(System.currentTimeMillis());  // 省略日志记录内容  }  @AfterReturning(returning = "ret", pointcut = "webLog()")  public void doAfterReturning(Object ret) throws Throwable {  // 处理完请求，返回内容  logger.info("RESPONSE : " + ret);  logger.info("SPEND TIME : " + (System.currentTimeMillis() - startTime.get()));  }  } |

优化：AOP切面的优先级

由于通过AOP实现，程序得到了很好的解耦，但是也会带来一些问题，比如：我们可能会对Web层做多个切面，校验用户，校验头信息等等，这个时候经常会碰到切面的处理顺序问题。

所以，我们需要定义每个切面的优先级，我们需要@Order(i)注解来标识切面的优先级。i的值越小，优先级越高。假设我们还有一个切面是CheckNameAspect用来校验name必须为didi，我们为其设置@Order(10)，而上文中WebLogAspect设置为@Order(5)，所以WebLogAspect有更高的优先级，这个时候执行顺序是这样的：

在@Before中优先执行@Order(5)的内容，再执行@Order(10)的内容

在@After和@AfterReturning中优先执行@Order(10)的内容，再执行@Order(5)的内容

所以我们可以这样子总结：

在切入点前的操作，按order的值由小到大执行

在切入点后的操作，按order的值由大到小执行

## 记录日志到MongoDB

### 通过自定义appender实现

思路：log4j提供的输出器实现自Appender接口，要自定义appender输出到MongoDB，只需要继承AppenderSkeleton类，并实现几个方法即可完成。

引入mongodb的驱动

在pom.xml中引入下面依赖

|  |
| --- |
| <dependency>  <groupId>org.mongodb</groupId>  <artifactId>mongodb-driver</artifactId>  <version>3.2.2</version>  </dependency> |

### 实现MongoAppender

编写MongoAppender类继承AppenderSkeleton，实现如下：

|  |
| --- |
| public class MongoAppender extends AppenderSkeleton {  private MongoClient mongoClient;  private MongoDatabase mongoDatabase;  private MongoCollection<BasicDBObject> logsCollection;  private String connectionUrl;  private String databaseName;  private String collectionName;  @Override  protected void append(LoggingEvent loggingEvent) {  if(mongoDatabase == null) {  MongoClientURI connectionString = new MongoClientURI(connectionUrl);  mongoClient = new MongoClient(connectionString);  mongoDatabase = mongoClient.getDatabase(databaseName);  logsCollection = mongoDatabase.getCollection(collectionName, BasicDBObject.class);  }  logsCollection.insertOne((BasicDBObject) loggingEvent.getMessage());  }  @Override  public void close() {  if(mongoClient != null) {  mongoClient.close();  }  }  @Override  public boolean requiresLayout() {  return false;  }  // 省略getter和setter  } |

定义MongoDB的配置参数，可通过log4j.properties配置：

connectionUrl：连接mongodb的串

databaseName：数据库名

collectionName：集合名

定义MongoDB的连接和操作对象，根据log4j.properties配置的参数初始化：

mongoClient：mongodb的连接客户端

mongoDatabase：记录日志的数据库

logsCollection：记录日志的集合

重写append函数：

根据log4j.properties中的配置创建mongodb连接

LoggingEvent提供getMessage()函数来获取日志消息

往配置的记录日志的collection中插入日志消息

重写close函数：关闭mongodb的

### 配置log4j.properties

设置名为mongodb的logger：

记录INFO级别日志

appender实现为com.didispace.log.MongoAppende

mongodb连接地址：mongodb://localhost:27017

mongodb数据库名：logs

mongodb集合名：logs\_request

|  |
| --- |
| log4j.logger.mongodb=INFO, mongodb  # mongodb输出  log4j.appender.mongodb=com.didispace.log.MongoAppender  log4j.appender.mongodb.connectionUrl=mongodb://localhost:27017  log4j.appender.mongodb.databaseName=logs  log4j.appender.mongodb.collectionName=logs\_request |

### 切面中使用mongodb logger

修改后的代码如下，主要做了以下几点修改：

logger取名为mongodb的

通过getBasicDBObject函数从HttpServletRequest和JoinPoint对象中获取请求信息，并组装成BasicDBObject

getHeadersInfo函数从HttpServletRequest中获取header信息

通过logger.info()，输出BasicDBObject对象的信息到mongodb

|  |
| --- |
| @Aspect  @Order(1)  @Component  public class WebLogAspect {  private Logger logger = Logger.getLogger("mongodb");  @Pointcut("execution(public \* com.didispace.web..\*.\*(..))")  public void webLog(){}  @Before("webLog()")  public void doBefore(JoinPoint joinPoint) throws Throwable {  // 获取HttpServletRequest  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();  HttpServletRequest request = attributes.getRequest();  // 获取要记录的日志内容  BasicDBObject logInfo = getBasicDBObject(request, joinPoint);  logger.info(logInfo);  }  private BasicDBObject getBasicDBObject(HttpServletRequest request, JoinPoint joinPoint) {  // 基本信息  BasicDBObject r = new BasicDBObject();  r.append("requestURL", request.getRequestURL().toString());  r.append("requestURI", request.getRequestURI());  r.append("queryString", request.getQueryString());  r.append("remoteAddr", request.getRemoteAddr());  r.append("remoteHost", request.getRemoteHost());  r.append("remotePort", request.getRemotePort());  r.append("localAddr", request.getLocalAddr());  r.append("localName", request.getLocalName());  r.append("method", request.getMethod());  r.append("headers", getHeadersInfo(request));  r.append("parameters", request.getParameterMap());  r.append("classMethod", joinPoint.getSignature().getDeclaringTypeName() + "." + joinPoint.getSignature().getName());  r.append("args", Arrays.toString(joinPoint.getArgs()));  return r;  }  private Map<String, String> getHeadersInfo(HttpServletRequest request) {  Map<String, String> map = new HashMap<>();  Enumeration headerNames = request.getHeaderNames();  while (headerNames.hasMoreElements()) {  String key = (String) headerNames.nextElement();  String value = request.getHeader(key);  map.put(key, value);  }  return map;  }  } |

### 其他补充

上述内容主要提供一个思路去实现自定义日志的输出和管理。我们可以通过jdbc实现日志记录到mongodb，也可以通过spring-data-mongo来记录到mongodb，当然我们也可以输出到其他数据库，或者输出到消息队列等待其他后续处理等。

## 动态修改日志级别

spring boot 1.5.0 增加的一个新特性。

### loggers端点

在pom.xml引入如下依赖。

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.1.RELEASE</version>  <relativePath<!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

### 在应用主类中添加一个接口用来测试日志级别的变化

比如下面的实现：

|  |
| --- |
| @RestController  @SpringBootApplication  public class DemoApplication {  private Logger logger = LoggerFactory.getLogger(getClass());  @RequestMapping(value = "/test", method = RequestMethod.GET)  public String testLogLevel() {  logger.debug("Logger Level ：DEBUG");  logger.info("Logger Level ：INFO");  logger.error("Logger Level ：ERROR");  return "";  }  public static void main(String[] args) {  SpringApplication.run(DemoApplication.class, args);  }    } |

### pplication.properties中增加一个配置

为了后续的试验顺利，在application.properties中增加一个配置，来关闭安全认证校验。

management.security.enabled=false

不然在访问/loggers端点的时候，会报如下错误：

|  |
| --- |
| {  "timestamp": 1485873161065,  "status": 401,  "error": "Unauthorized",  "message": "Full authentication is required to access this resource.",  "path": "/loggers/com.didispace"  } |

### 测试验证

在完成了上面的构建之后，我们启动示例应用，并访问/test端点，我们可以在控制台中看到如下输出：

|  |
| --- |
| 2017-01-31 22:34:57.123 INFO 16372 --- [nio-8000-exec-1] ication$$EnhancerBySpringCGLIB$$d2a0b1e2 : Logger Level ：INFO  2017-01-31 22:34:57.124 ERROR 16372 --- [nio-8000-exec-1] ication$$EnhancerBySpringCGLIB$$d2a0b1e2 : Logger Level ：ERROR |

由于默认的日志级别为INFO，所以并没有输出DEBUG级别的内容。下面我们可以尝试通过/logger端点来将日志级别调整为DEBUG，

比如，发送POST请求到/loggers/com.didispace端点，其中请求体Body内容为：

|  |
| --- |
| {  "configuredLevel": "DEBUG"  } |

重新访问/test端点，我们将在控制台中看到如下输出，在/test端点中定义的DEBUG日志内容被打印了出来：

|  |
| --- |
| 2017-01-31 22:37:35.252 DEBUG 16372 --- [nio-8000-exec-5] ication$$EnhancerBySpringCGLIB$$d2a0b1e2 : Logger Level ：DEBUG  2017-01-31 22:37:35.252 INFO 16372 --- [nio-8000-exec-5] ication$$EnhancerBySpringCGLIB$$d2a0b1e2 : Logger Level ：INFO  2017-01-31 22:37:35.252 ERROR 16372 --- [nio-8000-exec-5] ication$$EnhancerBySpringCGLIB$$d2a0b1e2 : Logger Level ：ERROR |

可以看到，到这里为止，我们并没有重启过Spring Boot应用，而只是简单的通过调用/loggers端点就能控制日志级别的更新。

除了POST请求之外，我们也可以通过GET请求来查看当前的日志级别设置，比如：发送GET请求到/loggers/com.didispace端点，我们将获得对于com.didispace包的日志级别设置：

|  |
| --- |
| {  "configuredLevel": "DEBUG",  "effectiveLevel": "DEBUG"  } |

我们也可以不限定条件，直接通过GET请求访问/loggers来获取所有的日志级别设置，这里就不列举具体返回，读者可以自行尝试。

# 开发测试部署

## 开发阶段

在开发阶段的时候最重要的是单元测试了，springboot对单元测试的支持已经很完善了。

单元测试是验证你代码第一道屏障，要养成每写一部分代码就进行单元测试的习惯，

不要等到全部集成后再进行测试，集成后因为更关注整体运行效果，很容易遗漏掉代码底层的bug.

## 集成测试

整体开发完成之后进入集成测试，spring boot项目的启动入口在 Application类中，直接运行run方法就可以启动项目，但是在调试的过程中我们肯定需要不断的去调试代码，如果每修改一次代码就需要手动重启一次服务就很麻烦，spring boot非常贴心的给出了热部署的支持，很方便在web项目中调试使用。

#### pom需要添加以下的配置：

<dependencies>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

<configuration>

<fork>true</fork>

</configuration>

</plugin>

</plugins>

</build>

添加以上配置后，项目就支持了热部署，非常方便集成测试。

## 投产上线

其实我觉得这个阶段，应该还是比较简单一般分为两种；

一种是打包成jar包直接执行，另一种是打包成war包放到tomcat服务器下。

### 如果你使用的是maven来管理项目

#### 打成jar包

cd 项目跟目录（和pom.xml同级），执行命令：mvn clean package

或者执行下面的命令

排除测试代码后进行打包：mvn clean package -Dmaven.test.skip=true

打包完成后jar包会生成到target目录下，命名一般是 项目名+版本号.jar

#### 启动jar包命令

java -jar target/spring-boot-scheduler-1.0.0.jar

这种方式，只要控制台关闭，服务就不能访问了。

下面我们使用在后台运行的方式来启动:

nohup java -jar target/spring-boot-scheduler-1.0.0.jar &

也可以在启动的时候选择读取不同的配置文件

java -jar app.jar --spring.profiles.active=dev

### 使用gradle管理项目

（1）gradle build

（2）java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar

### maven项目打成war包

打成war包一般可以分两种方式来实现，第一种可以通过eclipse这种开发工具来导出war包，

另外一种是使用命令来完成，这里主要介绍后一种

#### 修改pom包

将

<packaging>jar</packaging>

改为

<packaging>war</packaging>

#### 打包时排除tomcat.

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>provided</scope>

</dependency>

在这里将scope属性设置为provided，这样在最终形成的WAR中不会包含这个JAR包，

因为Tomcat或Jetty等服务器在运行时将会提供相关的API类。

#### 注册启动类

创建ServletInitializer.java，继承SpringBootServletInitializer ，

覆盖configure()，把启动类Application注册进去。

外部web应用服务器构建Web Application Context的时候，会把启动类添加进去。

public class ServletInitializer extends SpringBootServletInitializer {

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

return application.sources(Application.class);

}

}

#### 最后执行：mvn clean package -Dmaven.test.skip=true

会在target目录下生成：项目名+版本号.war文件，拷贝到tomcat服务器中启动即可。

### gradle项目打成war包

#### 如果使用的是gradle

基本步奏一样，build.gradle中添加war的支持，排除spring-boot-starter-tomcat：

...

apply plugin: 'war'

...

dependencies {

compile("org.springframework.boot:spring-boot-starter-web:1.4.2.RELEASE"){

exclude mymodule:"spring-boot-starter-tomcat"

}

}

...

#### 再使用构建命令：gradle build

war会生成在build\libs 目录下。

## 生产运维

### 查看JVM参数的值

可以根据java自带的jinfo命令：jinfo -flags pid

来查看jar 启动后使用的是什么gc、新生代、老年代分批的内存都是多少，示例如下：

-XX:CICompilerCount=3 -XX:InitialHeapSize=234881024 -XX:MaxHeapSize=3743416320 -XX:MaxNewSize=1247805440 -XX:MinHeapDeltaBytes=524288 -XX:NewSize=78118912 -XX:OldSize=156762112 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:+UseParallelGC

-XX:CICompilerCount ：最大的并行编译数

-XX:InitialHeapSize 和 -XX:MaxHeapSize ：指定JVM的初始和最大堆内存大小

-XX:MaxNewSize ： JVM堆区域新生代内存的最大可分配大小

...

-XX:+UseParallelGC ：垃圾回收使用Parallel收集器

### 简单粗暴重启

直接kill掉进程再次启动jar包

(1)ps -ef|grep java

拿到对于Java程序的pid

(2)kill -9 pid

再次重启

(3)java -jar xxxx.jar

当然这种方式比较传统和暴力，所以建议大家使用下面的方式来管理

### 脚本执行重启

#### 如果使用的是maven

<plugin>

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

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

<configuration>

<executable>true</executable>

</configuration>

</plugin>

#### 如果使用是gradle

springBoot {

executable = true

}

#### 启动方式：

第一种：可以直接./yourapp.jar 来启动

第二种：注册为服务，也可以做一个软链接指向你的jar包并加入到init.d中，然后用命令来启动。

init.d 例子:

ln -s /var/yourapp/yourapp.jar /etc/init.d/yourapp

chmod +x /etc/init.d/yourapp

这样就可以使用stop或者是restart命令去管理你的应用。

/etc/init.d/yourapp start|stop|restart

或者

service yourapp start|stop|restart

到此 springboot项目如何测试、联调和打包投产均已经介绍完，

以后可以找时间研究一下springboot的自动化运维，以及spring boot 和docker相结合的使用。

#### 修改端口

默认是开启8080端口，若想启动项目的时候，，则需要执行命令

java -jar 201701419\_cloud-0.0.1-SNAPSHOT.jar --server.port=8000

#### 配置服务的context-path

(spring boot 默认是 /)

server.context-path=/hebe/v1

# Spring Boot Maven Plugin

官方文档：<https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#?.?>

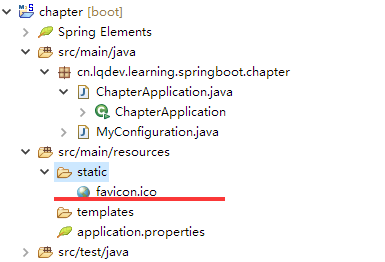
博客：<https://www.cnblogs.com/jpfss/p/11098740.html>

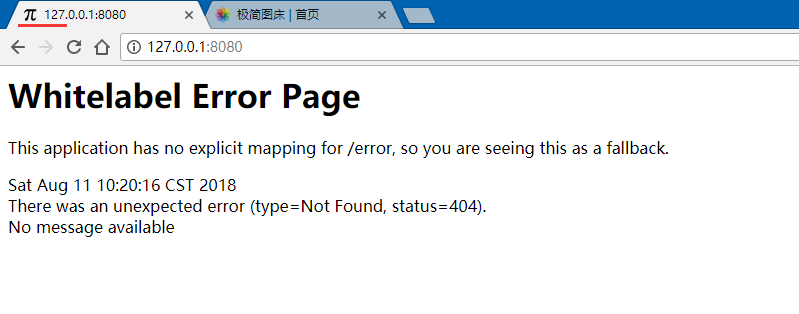
当我们使用spring boot项目对外发布API的时候，我们不能打包成可执行的jar，应该打包成普通的jar，所有我们打包的时候可以配置下，将需要打包的代码单独放到一个module中，并在module的pom.xml中，

# 小技巧

## 设置网站图标

原来我们在使用tomcat开发时，设置网站图片时，即icon图标时，一般都是直接替换root包下的favicon.ico替换成自己的，或者在网页的头部设置link的ref为icon然后设置其href值。而在SpringBoot中，替换图片也是很简单的，只需要将自定义图片放置在静态资源目录下即可，即默认有static、public、resources、/META-INF/resources或者自定义的静态目录下即可。





## 独立Tomcat运行war

讲解了这么久，一般上我们都是通过jar包的方式进行启动的应用的。所以部署在独立的tomcat时，需要如何解决呢？其实也简单，只需要将项目打包方式修改为war包，然后修改下启动类配置即可。

0.修改pom打包方式为war，同时排除了内置的tomcat。

<packaging>war</packaging>

<!-- 排除内置的tomcat -->

<dependency>

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

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

<scope>compile</scope>

</dependency>

<!-- 若直接有使用servlet对象时(这是废话，⊙﹏⊙‖∣)，需要将servlet引入，本例是没有的~ -->

<dependency>

<groupId>javax.servlet</groupId>

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

<scope>provided</scope>

</dependency>

1.改造下启动类，使其继承SpringBootServletInitializer,同时覆盖configure方法。

@SpringBootApplication

@Slf4j

public class ChapterApplication extends SpringBootServletInitializer{

public static void main(String[] args) {

SpringApplication.run(ChapterApplication.class, args);

// new SpringApplicationBuilder().sources(ChapterApplication.class).web(false).run(args);

//之后这里设置业务逻辑 比如挂起一个线程 或者设置一个定时任务。保证不退出

//不然它就是一个启动类，启动后就停止了。

log.info("jar,chapter启动!");

}

@Override

protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {

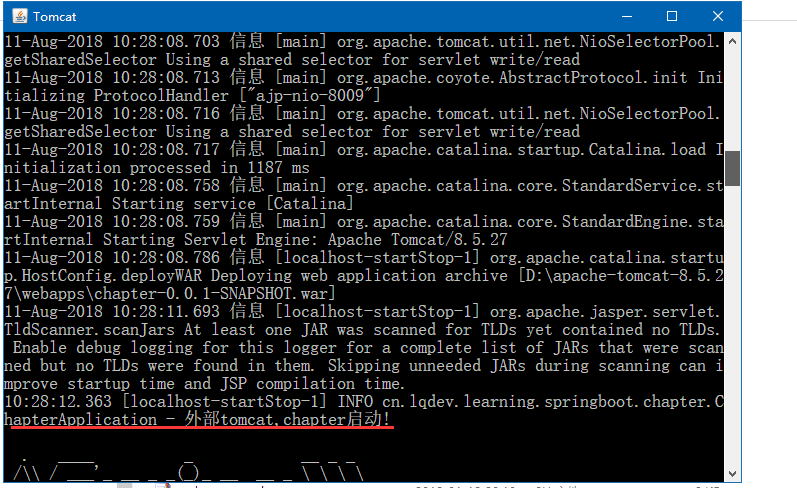
log.info("外部tomcat,chapter启动!");

return application.sources(ChapterApplication.class);

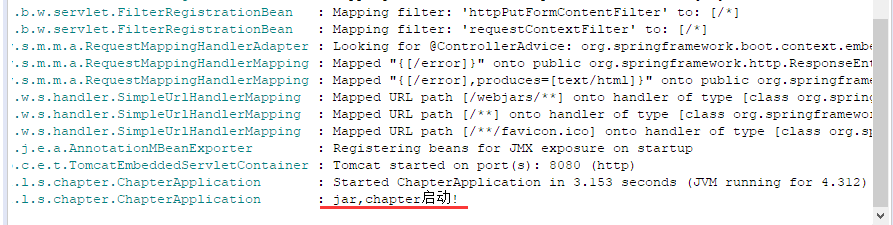
}

}

2.maven打包成war(mvn clean install),然后放入tomcat中，启动运行即可。



其实这样设置的话，在开发时直接运行启动类也还是可以直接运行的，方便.



## 启动不设置端口

对一些定时任务服务项目，其本身只是提供一个定时调度功能，不需要其他服务调用，只是去调度其他服务。像这样的服务，正常也就不需要设置端口了。这时候SpringBoot也是支持的。只需要改下启动方式：

new SpringApplicationBuilder().sources(ChapterApplication.class).web(false).run(args);

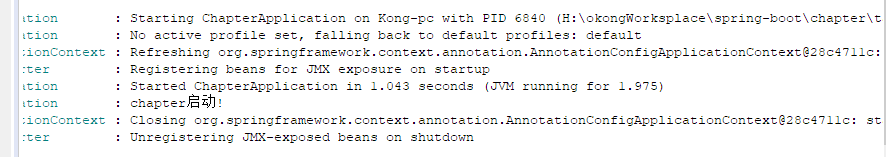
//之后这里设置业务逻辑 比如挂起一个线程 或者设置一个定时任务。保证不退出

//不然它就是一个启动类，启动后就停止了。

或者修改配置文件的属性：

spring.main.web-environment=false

最后效果，是不是没有看见端口了：



## 启动完成前进行业务逻辑

利用CommandLineRunner或者ApplicationRunner可实现在SpringApplication的run()完成前执行一些业务逻辑

0.修改启动类，实现CommandLineRunner接口,ApplicationRunner类似，只是run的入参不同而已。

@Override

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

log.info("CommandLineRunner运行");

}

1.运行应用，注意查看控制台输出：

当然，直接申明一个bean也是可以的。

@Configuration

@Slf4j

public class CommandLineRunnerConfig {

@Bean

public CommandLineRunner runner(){

return new CommandLineRunner() {

public void run(String... args){

log.info("CommandLineRunner运行2");

}

};

}

}

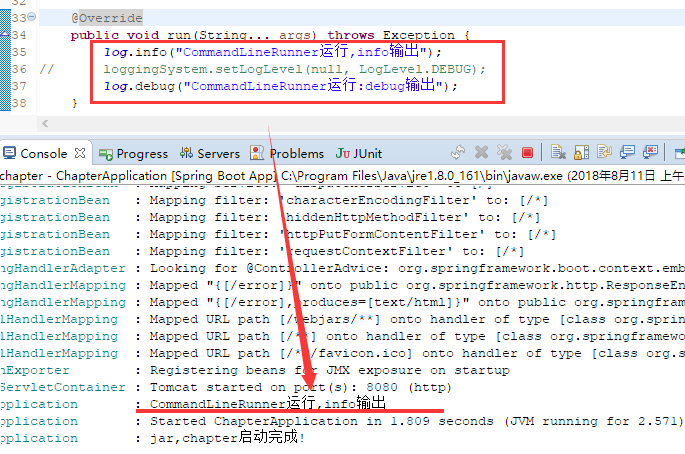
若多个时，可设置@Order来确定执行的顺序。

## 动态修改日志级别

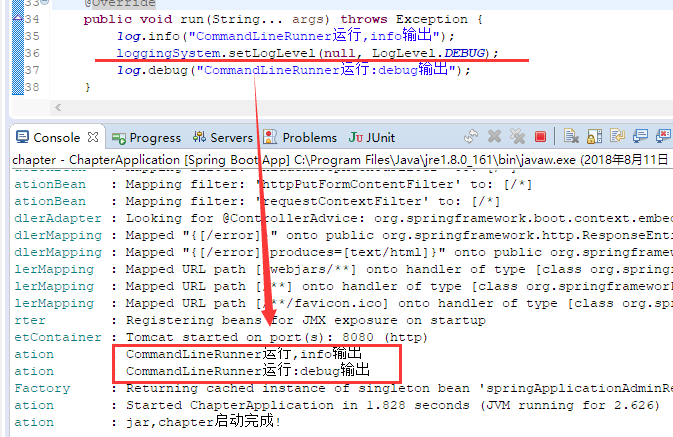
通过org.springframework.boot.logging.LoggingSystem提供的api即可。

loggingSystem.setLogLevel(null, LogLevel.DEBUG);

如，默认时是info模式，未修改时，debug模式是不会输出的。



动态设置后



## 热部署

前面讲了这么多章节，因为功能都很单一，所以一般上都是直接重启服务来进行更新操作。但当服务功能一多，启动速度缓慢时，还是配置个热部署比较方便。在SpringBoot中，只需要加入一个spring-boot-devtools即可

<dependencies>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

题外话：这里的<optional>true</optional>是表示依赖不会传递，依赖了此项目的需要额外引入此包，若需要使用的话。

若不生效，可试着在打包工具spring-boot-maven-plugin下的configuration加入<fork>true</fork>看看，具体配置项如下：

<plugin>

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

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

<configuration>

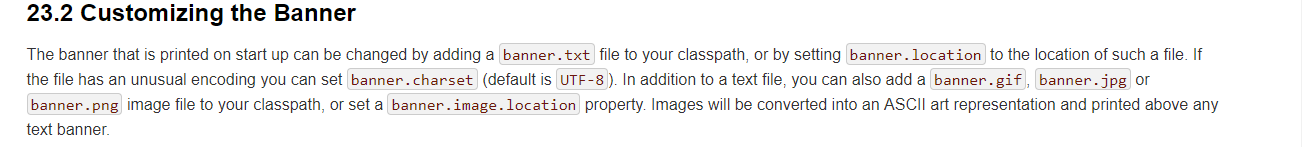
<fork>true</fork>

</configuration>

</plugin>

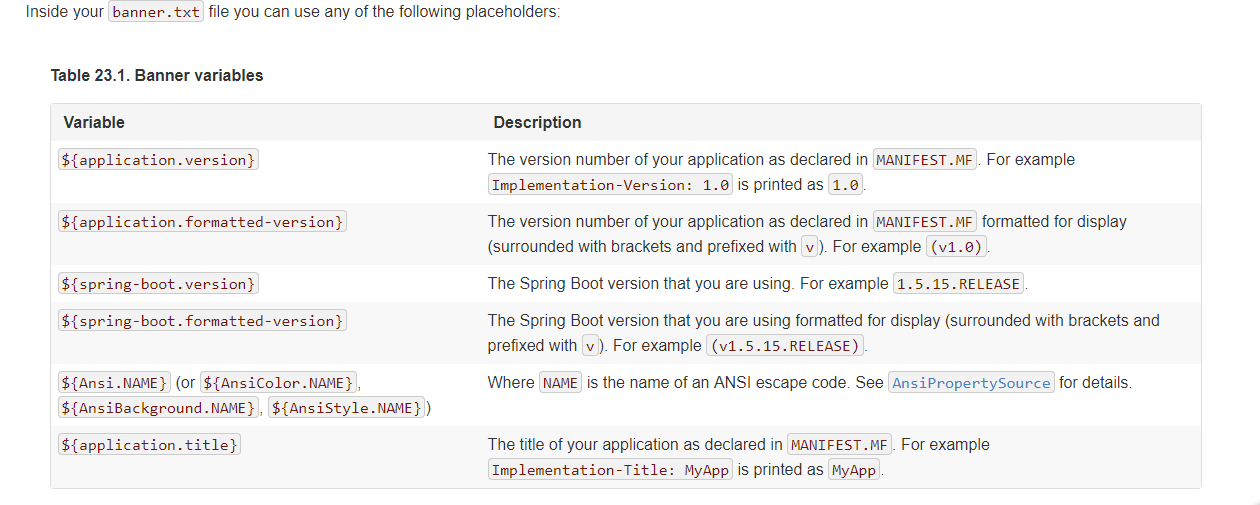
## 自定义启动Banner

看烦了自带的Banner，动手修改一个属于自己的Banner，提现逼格的时候到了~哈哈，以下是官网给的配置指南：

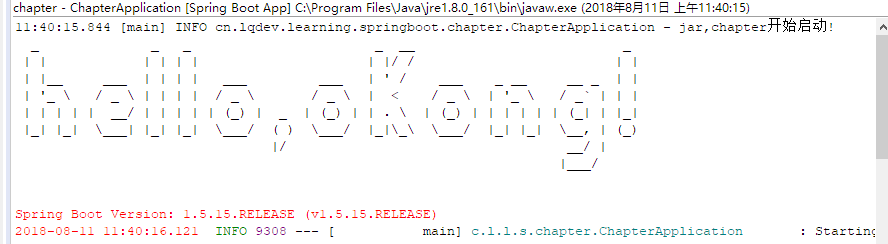


### 文字形式

其实，替换很简单，只需要在classpath路径下创建一个banner.txt即可。具体的一些变量官网也有给出，具体如下：



现在我们就定制一个自己的Banner。



题外话：手输字符画是不太现实的，大家可通过一些网站进行快速生成。可自行搜索下，网上一搜一大把。

例子：

Application Version: ${smart-call.version}

Spring Boot Version: ${spring-boot.version}

////////////////////////////////////////////////////////////////////

// \_ooOoo\_ //

// o8888888o //

// 88" . "88 //

// (| ^\_^ |) //

// O\ = /O //

// \_\_\_\_/`---'\\_\_\_\_ //

// .' \\| |// `. //

// / \\||| : |||// \ //

// / \_||||| -:- |||||- \ //

// | | \\\ - /// | | //

// | \\_| ''\---/'' | | //

// \ .-\\_\_ `-` \_\_\_/-. / //

// \_\_\_`. .' /--.--\ `. . \_\_\_ //

// ."" '< `.\_\_\_\\_<|>\_/\_\_\_.' >'"". //

// | | : `- \`.;`\ \_ /`;.`/ - ` : | | //

// \ \ `-. \\_ \_\_\ /\_\_ \_/ .-` / / //

// ========`-.\_\_\_\_`-.\_\_\_\\_\_\_\_\_/\_\_\_.-`\_\_\_\_.-'======== //

// `=---=' //

// ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ //

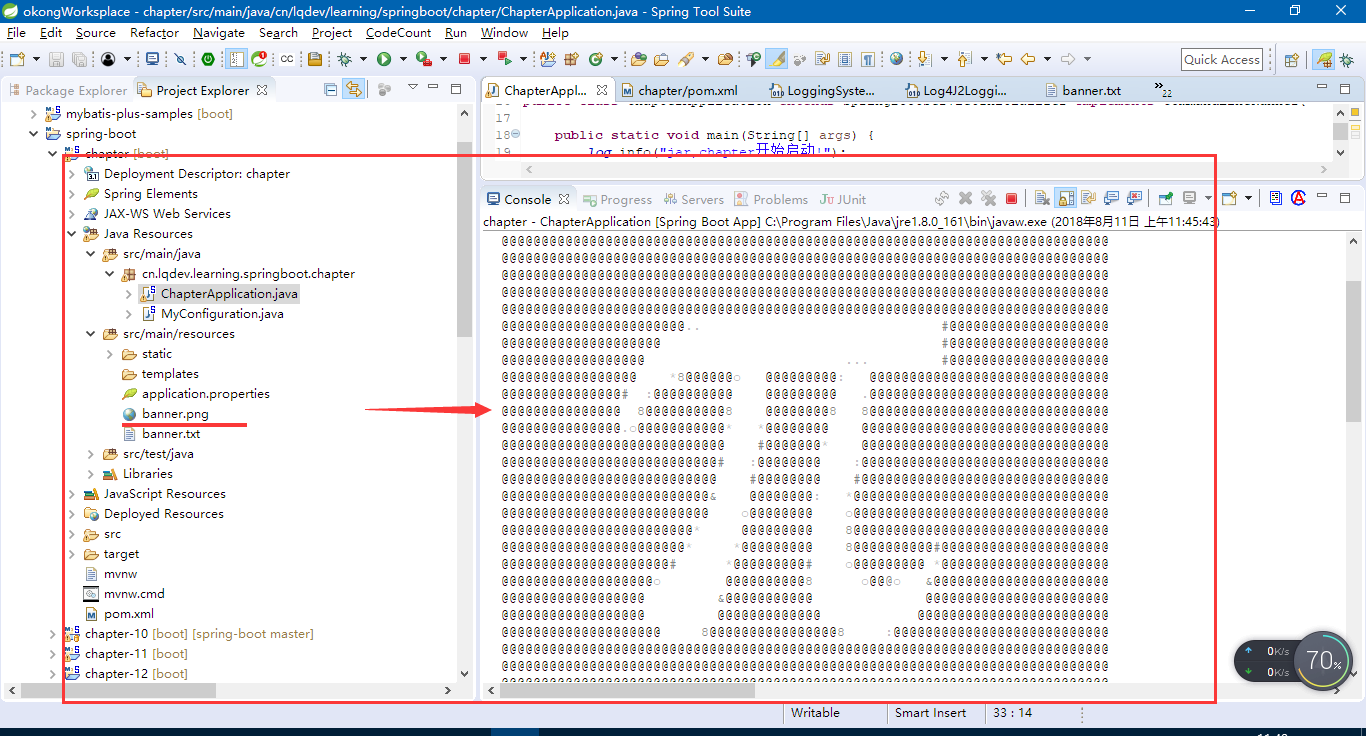
// 佛祖保佑 永不宕机 永无BUG //

////////////////////////////////////////////////////////////////////

### 图片形式

若觉得使用文字不够酷炫，当然也可以将图片设置为启动的banner。目前支持的图片格式有gif、png、jpg。使用也很简单，只需要命名为banner即可。

如将头像放入目录中，最后的效果如下：

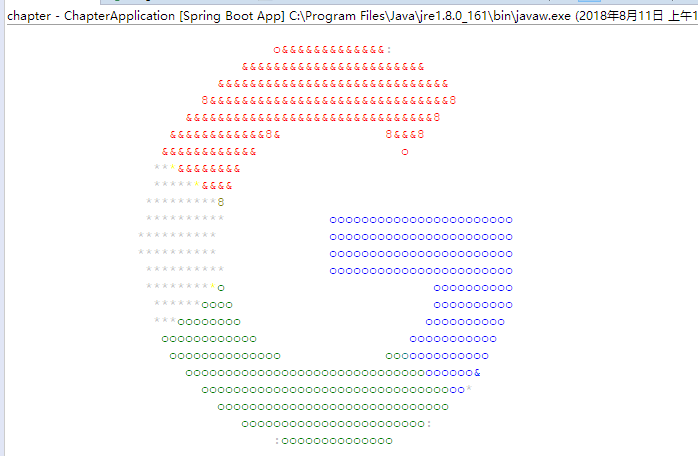


当然，若图片是有色彩的，也是可以的，对于太复杂的图片显示效果就不佳了，如下。

原图：



banner效果图



## 方法重试

### 依赖

<!-- Spring 重试组件 -->

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

</dependency>

### 启动重试组件

@Configuration

@EnableRetry//开启重试

public class RetryConfig {

}

### 业务类配置

@Service

public class UserService {

@Retryable(

//需要进行重试的异常，和参数includes是一个意思。默认为空，当参数exclude也为空时，所有异常都将要求重试。

value = RemoteAccessException.class,

// 需要进行重试的异常，默认为空。当参数exclude也为空时，所有异常都将要求重试。

//include = {NullPointerException.class, IOException.class},

// 不需要重试的异常。默认为空，当参include也为空时，所有异常都将要求重试。

//exclude = {},

//标明重试是否是有状态的，异常引发事务失效的时候需要注意这个。该参数默认为false。远程方法调用的时候不需要设置，因为远程方法调用是没有事务的；

// 只有当数据库更新操作的时候需要设置该值为true，特别是使用Hibernate的时候。抛出异常时，异常会往外抛，使事务回滚；重试的时候会启用一个新的有效的事务。

stateful = false,

//最大重试次数，默认为3。包括第一次失败

maxAttempts = 3,

// 回避策略，默认为空。该参数为空时是，失败立即重试，重试的时候阻塞线程。

backoff = @Backoff(

//重试延迟时间，单位毫秒，默认值1000，即默认延迟1秒。

// 当未设置multiplier时，表示每隔value的时间重试，直到重试次数到达maxAttempts设置的最大允许重试次数。

// 当设置了multiplier参数时，该值作为幂运算的初始值。等同delay参数，两个参数设置一个即可。

value = 1000L,

// 两次重试间最大间隔时间。当设置multiplier参数后，下次延迟时间根据是上次延迟时间乘以multiplier得出的，

// 这会导致两次重试间的延迟时间越来越长，该参数限制两次重试的最大间隔时间，当间隔时间大于该值时，计算出的间隔时间将会被忽略，使用上次的重试间隔时间。

maxDelay = 10000,

//作为乘数用于计算下次延迟时间。公式：delay = delay \* multiplier

multiplier = 3,

//是否启用随机退避策略，默认false。

// 设置为true时启用退避策略，重试延迟时间将是delay和maxDelay间的一个随机数。

// 设置该参数的目的是重试的时候避免同时发起重试请求，造成Ddos攻击

random = false

),

// SimpleRetryPolicy.canRetry()返回true时该表达式才会生效，触发重试机制。如果抛出多个异常，只会检查最后那个。

exceptionExpression= ""

)

public String get() {

System.out.println("触发任务");

throw new RemoteAccessException("调用异常");

}

// 该注解用于恢复处理方法，当全部尝试都失败时执行。

// 返回参数必须和@Retryable修饰的方法返回参数完全一样。

// 第一个参数必须是异常，其他参数和@Retryable修饰的方法参数顺序一致。

@Recover

public String recover(RemoteAccessException e) {

e.printStackTrace();

return "全都的重试都失败了";

}

}

@RestController

public class RetryController {

@Autowired

private UserService userService;

@GetMapping("/get")

public String get() {

return this.userService.get();

}

}

### 页面访问

访问127.0.0.1:8080/get

控制台出现：

触发任务

触发任务

触发任务

之后浏览器显示： 全都的重试都失败了

## 性能优化

### Tomcat

其实默认的设置已经是比较理想的状态了，但是如果条件允许可以适当的调整

server:

tomcat:

max-connections: 500 # 最大TCP连接数, 默认100000

min-spare-threads: 20 # 初始化线程是20, 默认10

max-threads: 100 # 最大线程数是100, 默认200

connection-timeout: 60000 # 超时时间是60s

### JVM

1. 使用-server模式

设置JVM使用server模式。64位JDK默认启动该模式

java -server -jar springboot-1.0.jar

2. 指定堆参数

这个根据服务器的内存大小，来设置堆参数。

-Xms :设置Java堆栈的初始化大小

-Xmx :设置最大的java堆大小

java -server -Xms512m -Xmx768m -jar springboot-1.0.jar

设置初始化堆内存为512MB，最大为768MB。

3.远程Debug

在服务器上将启动参数修改为：

java

-Djavax.net.debug=ssl

-Xdebug

-Xnoagent

-Djava.compiler=NONE

-Xrunjdwp:transport=dt\_socket,server=y,suspend=n,address=8888

-jar springboot-1.0.jar

这个时候服务端远程Debug模式开启，端口号为8888。

## 打包为普通jar

很多时候，我们不要讲项目打包成spring boot应用，只需要打包成普通jar。比如发布api。这个时候我们需要设置如下即可

<!--打包时，跳过打包成spring boot应用。直接打包成普通jar-->

<plugin>

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

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

<configuration>

<skip>true</skip>

</configuration>

</plugin>