SpringBoot框架开发详解

# SpringBoot入门

## SpringBoot简介

长期以来Java的开发一直让人所诟病：

* Java项目开发复杂度极其高；
* Java项目的维护非常困难；
* 在云时代如何实现项目的快速部署以及快速启动；
* 即便使用了大量的开发框架，发现我们的开发也没少多少；
* 当所有的人认为Spring不在前进的时候，Spring推出了微架构实现的两个重要开发框架：SpringBoot、SpringCloud。

1. Java开发的复杂度是最高的？

在所有的软件行业里面，如果要说商用体系，排在第一位的永远是Java，因为Java的体系丰富，支持度高，安全性也高，但是同时我们所有的开发者也不得不去忍受Java中的以下痛苦：

* Java里面提供的开发支持都属于原生操作代码，例如：JDBC为例，如果使用Java原生代码会重复编写大量的内容，例如：PreparedStatement操作：
* Java进行WEB项目开发的时候，必须要求按照严格的格式进行WEB项目的创建，以及每当修改WEB程序的时候又需要进行Tomcat的重新启动；
* Java之中虽然提供了所谓的开发标准，但是所有的公司几乎都有可能有自己的标准，例如：最初的时代JVM的标准就有三个，而且许多的公司由于版本不同会造成部署的环境不同；
* Java里面严格要求按照MVC的设计模式；
* 以WEB开发为例，一个良好的JSP程序代码里面不应该包含有任何的scriptlet程序代码，但是要想做到这一步发现非常麻烦，有各种实现标准，例如：JSTL + EL、SpringTaglib、StrutsTaglib、JSF、Shiro、SpringSecurity；
* 如果现在使用Node.js、Python开发一个控制器程序类可能只需要几行代码，而Java要想开发这样一个程序，需要写一堆的代码，而且还要求搞部署；

1. Java后期的发展使用了大量的Maven技术作为开发，那么使用Maven之后你会发现传统的开发处理之中并没有逃离掉传统WEB的身影，所有的项目依然需要打包为war文件，而后上传到系统之中。使用Maven还有一个最大的痛：如果是开发框架，那一堆的Maven的配置依赖库；
2. Rest技术已经开始在行业之中广为流传，而Java要想实现Rest架构的开发（基于：Spring），那么也是相当麻烦的；
3. 现在行业之中，Spring已经作为了绝对的Java架构，但是如果要想在Spring之中整合RabbitMQ、Kafka、ActiveMQ、MySQL、Druid、Redis、Shiro，需要编写一堆堆的\*.xml配置文件；

所以在这样的一个大的历史背景下，很多人开始寻求更加简便的开发，而遗憾的是这种简便的开发没有被JDK所支持、没有被JavaEE所支持，因为这些只是平台，平台能够提供的只是最原始的技术支持。这一时刻终于由于Spring框架的升级而得到了新生，SpringBoot的出现，改变了所有Java开发的困境，SpringBoot的最终奉行的宗旨：废除掉所有复杂的开发，废除掉所有的配置文件，让开发变得更简单纯粹，核心：“零配置”（当然这只是个梦想而已）。

SpringBoot之所以慢慢可以火遍全世界，是因为在SpringBoot中使用的大量注解还是之前Spring所提供的注解，那么这一点让所有的开发者几乎可以零适应进行完整过渡。

## SpringBoot快速启动

本次快速启动程序将直接采用Spring官方给出的程序来进行代码的执行分析。

1. 如果要想进行SpringBoot开发，一定要使用Maven或其它的项目管理工具完成；

* SpringBoot运行之后会以WEB程序为主，但是现在所建立的只是一个普通的quickstart程序；

1. 如果要想开发SpringBoot程序只需要按照官方给出的要求配置一个父pom即可；

|  |
| --- |
| <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">  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.4.RELEASE</version>  </parent>  <modelVersion>4.0.0</modelVersion>  <artifactId>springboot-base</artifactId>  <packaging>jar</packaging>  <name>springboot-base</name>  <url>http://examples.kuhnwei.com</url>  <properties>  <jdk.version>1.8</jdk.version>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <build>  <finalName>springboot-base</finalName>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>${jdk.version}</source>  <target>${jdk.version}</target>  <encoding>${project.build.sourceEncoding}</encoding>  </configuration>  </plugin>  </plugins>  </build> </project> |

1. 编写一个具体的程序：

|  |
| --- |
| package com.kuhnwei.base; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.EnableAutoConfiguration; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; @Controller @EnableAutoConfiguration public class SampleController {  @RequestMapping("/")  @ResponseBody  public String home() {  return "Hello World";  }  public static void main(String[] args) {  SpringApplication.*run*(SampleController.class, args);  } } |

而后可以直接通过Java程序运行该SpringBoot的代码，浏览器访问：<http://localhost:8080/> 。

# SpringBoot基本概念

在之前所建立的SpringBoot项目只是根据官方文档实现的一个基础程序模型，但是这样的代码肯定不适合实际的项目开发，因为从实际的Maven项目来讲，应该要有统一的父pom.xml文件。

## 统一父pom管理

1. 首先建立一个examples-springboot-parent的Maven项目；

|  |
| --- |
| <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>com.kuhnwei</groupId>  <artifactId>springboot-parent</artifactId>  <version>1.0-SNAPSHOT</version>  <modules>  <module>examples-springboot-base</module>  </modules>  <packaging>pom</packaging>  <name>springboot-parent</name>  <url>http://examples.kuhnwei.com</url>  <properties>  <jdk.version>1.8</jdk.version>  <springboot.version>1.5.4.RELEASE</springboot.version>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  </properties>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>${springboot.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <finalName>springboot-base</finalName>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>${jdk.version}</source>  <target>${jdk.version}</target>  <encoding>${project.build.sourceEncoding}</encoding>  </configuration>  </plugin>  </plugins>  </build> </project> |

1. 建立examples-springboot-base的子模块，实现之前同样的基础操作功能，修改pom.xml文件，追加springboot的WEB启动包；

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency> </dependencies> |

1. 建立与之前同样的程序类。

|  |
| --- |
| package com.kuhnwei.base; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.EnableAutoConfiguration; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; @Controller @EnableAutoConfiguration public class SampleController {  @RequestMapping("/")  @ResponseBody  public String home() {  return "www.kuhnwei.com";  }  public static void main(String[] args) {  SpringApplication.*run*(SampleController.class, args);  } } |

在这样的状态下才可以进行后续的代码编写。

## SpringBoot代码测试

现在的程序里面已经实现了一个最为简单的额控制器程序类，不过从实际的项目角度来讲，必须要求考虑到代码测试问题，而且现在的程序代码属于springboot，则需要在你的项目之中进行如下的pom.xml文件的变更。

1、【examples-springboot-base模块】修改pom.xml配置文件，追加SpringBoot测试支持类；

|  |
| --- |
| <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope> </dependency> <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope> </dependency> |

只要进行Java测试，最简单适用的就是junit，所以这个开发包一定要随测试一起导入。

2、【examples-springboot-base模块】建立一个测试程序类；

|  |
| --- |
| package com.kuhnwei.base; import junit.framework.TestCase; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringJUnit4ClassRunner; import org.springframework.test.context.web.WebAppConfiguration; @SpringBootTest(classes = SampleController.class) @RunWith(SpringJUnit4ClassRunner.class) @WebAppConfiguration public class SampleControllerTest {  @Autowired  private SampleController sampleController;  @Test  public void home() {  TestCase.*assertEquals*(this.sampleController.home(), "www.kuhnwei.com");  } } |

## Spring启动注解分析

现在为止已经可以发现在整个SpringBoot程序里面适用了许多的注解，首先把这些注解做一个列表：

|  |  |  |
| --- | --- | --- |
| **No.** | **注解** | **说明** |
| 1 | @Controller | 进行控制器的配置注解，这个注解所在的类是控制器类 |
| 2 | @EnableAutoConfiguration | 开启自动配置处理 |
| 3 | @RequestMapping(“/”) | 表示访问的映射路径，此时的路径为“/”，访问地址：<http://localhost:8080> |
| 4 | @ResponseBody | 在Restful架构之中，该注解表示直接将返回的数据以字符串或JSON的形式获得 |

可以发现在给定的几个注解之中“@EnableAutoConfiguration”为整个SpringBoot的启动注解配置，也就是说这个注解应该随着程序的主类一起进行定义。

而对于控制器程序类，由于在项目之中会有许多的控制器，那么最好将这些类统一保存在一个包中，下面将所有的控制器程序来保存在“com.kuhnwei.base.web”包中。

**强烈建议（Spring官方建议）：如果要想进行简单方便的开发，所有的程序类一定要在启动类所在包的子包下。**

1、【examples-springboot-base模块】建立一个HelloController程序类；

|  |
| --- |
| package com.kuhnwei.base.web; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.ResponseBody; @Controller public class HelloController {  @RequestMapping("/")  @ResponseBody  public String home() {  return "www.kuhnwei.com";  } } |

2、【examples-springboot-base模块】启动程序主类；

|  |
| --- |
| package com.kuhnwei.base; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.EnableAutoConfiguration; import org.springframework.context.annotation.ComponentScan; @EnableAutoConfiguration // 定义一个扫描路径 @ComponentScan("com.kuhnwei.base") public class SampleController {  public static void main(String[] args) {  SpringApplication.*run*(SampleController.class, args);  } } |

3、【examples-springboot-base模块】以上的做法只是传统程序的开发做法，因为现在为止毕竟是两块程序类，这两个彼此之间的联系需要有一个连接点，而程序中配置的“@ComponentScan”就是负责这个连接处理，但是SpringBoot考虑到了此类的配置问题，所以提出了一个更简化的策略，该策略的核心思想：既然程序主类会在所有开发包的父包里面，那么能不能简化点取得配置呢？为此在实际的开发之中，会使用一个特殊的复合注解：

|  |
| --- |
| package com.kuhnwei.base; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; // 启动SpringBoot程序，而后自带子包扫描 @SpringBootApplication public class SampleController {  public static void main(String[] args) {  SpringApplication.*run*(SampleController.class, args);  } } |

“@SpringBootApplication=@EnableAutoConfiguration+@ComponentScan+其它配置”。正式因为这样的特点，所以当以后使用Bean实现配置处理的时候讲会非常容易。

## 配置访问路径

在一个实际的项目开发之中，控制器的路径可能会有许多个，而且在进行控制器编写的时候也会有两种运行模式：跳转配置、Restful显示。那么下面来观察关于路径的详细描述。

1. 在之前所编写的控制器里面你会发现有如下的两个注解配置使用：

* @Controller：在类上定义表示定义的是一个控制器；
* @ResponseBody：将控制器中方法的返回值变为rest内容。

但是如果说现在一个项目里面可能控制器之中返回的全部都是Restful信息，这样分别定义就太麻烦了，为此在SpringBoot里面又提供有一个复合注解：“@RestController”

|  |
| --- |
| package com.kuhnwei.base.web; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; @RestController public class HelloController {  @RequestMapping("/")  public String home() {  return "www.kuhnwei.com";  } } |

因为从MVC实际标准来讲，控制器需要传递一些属性到页面上进行显示，按照这样的原则并不是所有的开发都会以Restful结构返回，但是Rest结构是SpringCloud的实现核心技术。

1. 现在所给出的控制器的类里面只是负责了简单的信息返回，那么实际上也可以进行参数的接收处理，传递参数到控制器之中最简单的做法是使用地址重写传递“xx?参数名称=内容”；

|  |
| --- |
| @RequestMapping("/echo") public String echo(String msg) {  return "【ECHO】 " + msg; } |

访问路径：<http://localhost:8080/echo?msg=hello> ;

1. 由于SpringBoot支持Rest风格处理，所以此时对于参数的接收可以采用路径参数的形式完成：

|  |
| --- |
| @RequestMapping(value = "/echo/{message}", method = RequestMethod.*GET*) public String echo(@PathVariable("message") String msg) {  return "【ECHO】 " + msg; } |

访问路径：<http://localhost:8080/echo/hello> ;

个人意见：虽然路径参数的形式属于rest操作标准，但是个人觉得使用地址重写传递参数更加简单，不过这一点使用什么样的模式来处理，取决于你所处的项目团队

1. 在SpringBoot的处理操作之中实际上已经充分考虑到了此类情况，所以追加所有自动加载配置的依赖库；

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

## 使用内置对象

通过整个SpringBoot程序可以发现，在SpringBoot中的控制器的形式和SpringMVC是一样的，所以如果现在要想在你的程序之中去使用JSP的内置对象，那么也可以按照与SpringMVC同样的方式进行。

|  |
| --- |
| @RequestMapping("/object") public String object(HttpServletRequest request, HttpServletResponse response) {  System.*out*.println("\*\*\* 客户端IP地址：" + request.getRemoteAddr());  System.*out*.println("\*\*\* 取得客户端响应编码：" + response.getCharacterEncoding());  System.*out*.println("\*\*\* 取得SeesionID：" + request.getSession().getId());  System.*out*.println("\*\*\* 取得真实路径：" + request.getServletContext().getRealPath("/upload/"));  return "www.kuhnwei.com"; } |
| \*\*\* 客户端IP地址：0:0:0:0:0:0:0:1  \*\*\* 取得客户端响应编码：ISO-8859-1  \*\*\* 取得SeesionID：CB6C6877AC98E82DE80E6C17115C79E4  \*\*\* 取得真实路径：C:\Users\kuhnwei\AppData\Local\Temp\tomcat-docbase.6497389346945161688.8080\upload\ |

通过以上的信息可以发现，此时的SpringBoot运行依然需要有一个工作目录，只不过这个工作目录是由SpringBoot自己定义的，主要就是当前用户的目录下存在。

## 项目打包发布

SpringBoot作为微架构实现主要技术，其发布项目的方式极为简单，只需要你在你的项目中配置好插件，以及打包就可以执行了，并且这个执行不需要特别复杂的配置。

1、【examples-springboot-parent项目】修改pom.xml配置文件，追加新的插件：

|  |
| --- |
| <plugin>  <!-- 该插件的主要功能是进行项目的打包发布处理 -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <!-- 设置程序的主类 -->  <mainClass>com.kuhnwei.base.SampleController</mainClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions> </plugin> |

由于增加了新的插件，所以对于整个的程序一定要进行整体的项目更新。

2、【examples-springboot-parent项目】将当前项目进行打包处理：mvn clean package，此时将形成“examples-springboot-base.jar”程序文件，并且这个文件里面包含有全部的以来支持库文件；

3、将“examples-springboot-base.jar”文件随意拷贝到一个路径之中，例如：D盘，而后进入到命令行方式下：

|  |
| --- |
| java –jar examples-springboot-base.jar |

如果此时要想在Linux下执行，只需要将这个\*.jar文件直接上传到Linux下即可。

# SpringBoot开发深入

在之前已经基本上了解了整个SpringBoot运行机制，但是也需要清楚的认识到以下的问题，在实际的项目开发之中，尤其是Java的MVC版项目里面，所有的项目都一定需要满足于如下几点要求：

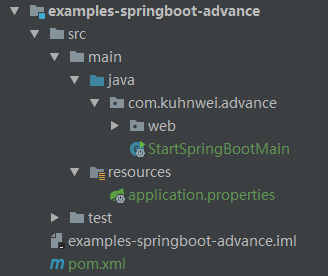
* 访问的端口不能够是8080，应该使用默认的80端口；
* 在项目之中为了方便进行数据的维护，建议建立一系列的\*.properties配置文件，例如：提升消息、跳转路径；
* 所有的控制器现在都采用了Rest风格输出，但是正常来讲，信息的显示应该交给页面（不再是JSP）负责完成；
* 在进行项目打包的时候应该考虑到不同的profile配置。

## 配置环境属性

通过当前的执行可以发现，默认情况下，在SpringBoot里面启动WEB容器为Tomcat：

|  |
| --- |
| INFO 5072 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup  INFO 5072 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http) |

从实际的开发来讲，我们的WEB部署项目如果是单独运行不可能运行在8080端口上。从正常来讲我们的服务器应该运行在80端口上，所以如果要想修改这样的默认环境，则必须编写与之对应的配置文件，该配置文件一定要卸载classpath之中，例如：项目中的“src/main/resources”就是一个classpath路径，那么在这个目录之中直接创建有一个application.properties（文件名称绝对不要改变）。



**范例**：定义application.properties配置文件

|  |
| --- |
| # 设置Tomcat的运行服务所在端口 server.port=80 |

随后重新运行程序会出现有如下的提示信息：“Tomcat started on port(s):80 (http)”。

**范例**：如果有需要也可以配置contextPath的信息

|  |
| --- |
| # 可以配置ContextPath访问路径，但是在实际开发之中是不能够进行配置的 server.context-path=/advance |

此时的访问路径需要追加contextPath前缀：<http://localhost/advance/>；

严格来讲在SpringBoot、SpringCloud里面可以使用两类配置文件：application.properties、application.yml。

|  |
| --- |
| **YAML文件**：  这是一种结构化的数据文件，其在很多的地方上都使用过，例如：Apache Storm开发组件上进行配置的时候使用的就是yml配置文件，该配置文件的全称（Yet Another Markup Languange、仍然是一种标记语言）。 |

**范例**：定义application.yml配置文件

|  |
| --- |
| server:  port:80 #此处设置的服务的访问端口配置 |

无聊的测试：如果application.properties和application.yml两个文件同时存在使用哪个？

那么这个时候将优先进行application.properties配置文件的加载，如果现在两个配置项的作用冲突了，则以properties为主，如果不冲突，则以存在的为主。

在之前默认默认使用的WEB容器是Tomcat容器，实际上在SpringBoot里面如果用户需要也可以将容器更换为jetty容器，如果现在要想使用这个容器，则只需要追加一些依赖即可：

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

如果在以后项目以微服务的方式进行打包发布的时候，强烈建议大家使用jetty服务进行发布。

## 读取资源文件

在实际的项目开发之中资源文件一定不可或缺，因为所有的提示文字信息都要求在资源文件之中进行定义，而且资源文件是实现国际化技术的主要手段。如果要想在SpringBoot里面进行资源文件的配置只需要做一些简单的application.yml配置即可，而且所有注入的资源文件都可以像最初的Spring处理那样直接使用MessageSource进行读取。

1. 为了统一管理资源文件，在“src/main/resources”目录之中建立有一个i18n的存储目录；
2. 在“src/main/resources/i18n”目录之中建立有两个资源文件：

* 建立messages.properties配置文件：

|  |
| --- |
| welcome.url=www.kuhnwei.com welcome.msg=欢迎{0}光临！ |

* 建立pages.properties配置文件：

|  |
| --- |
| mamber.add.page=/pages/back/admin/member/member\_add.jsp member.add.action=/pages/back/admin/member/member\_add.action |

1. 修改application.yml配置文件：

|  |
| --- |
| *# 表示该配置直接为Spring容器负责处理* **spring:** *# 表示进行资源配置* **messages:** *# 资源文件的名称* **basename:** i18n/messages,i18n/pages **server:** *# 此处设置的服务的访问端口配置* **port:** 80 |

1. 当执行完以上的配置之后会自动为用户创建MessageSource对象，那么用户在使用的时候直接注入此对象即可。
   1. 考虑到实际开发的标准型，所以现在建议创建一个父的控制器的抽象类：AbstracBaseController，而后在此抽象类之中进行资源读取类的配置：

|  |
| --- |
| package com.kuhnwei.advance.web; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.context.MessageSource; import java.util.Locale; public abstract class AbstracBaseController {  @Autowired  private MessageSource messageSource;  public String getMessage(String key, String... args) {  return this.messageSource.getMessage(key, args, Locale.*getDefault*());  } } |

1. 在控制器的子类之中读取以上的配置信息：

|  |
| --- |
| package com.kuhnwei.advance.web; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.bind.annotation.RestController; @RestController public class MessageController extends AbstracBaseController {   @RequestMapping(value = "/echo", method = RequestMethod.*GET*)  public String echo(String mid) {  System.*out*.println("【\*\*\* 访问地址 \*\*\*】" + super.getMessage("member.add.action"));  return super.getMessage("welcome.msg", mid);  } } |

1. 编写一个junit测试类，来测试以上的控制器程序是否正确：

|  |
| --- |
| package com.kuhnwei.advance.advance; import com.kuhnwei.advance.StartSpringBootMain; import com.kuhnwei.advance.web.MessageController; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringJUnit4ClassRunner; import org.springframework.test.context.web.WebAppConfiguration; @SpringBootTest(classes = StartSpringBootMain.class) @RunWith(SpringJUnit4ClassRunner.class) @WebAppConfiguration public class MessageControllerTest {  @Autowired  private MessageController messageController;  @Test  public void echo() {  System.*out*.println(this.messageController.echo("kuhnwei"));  } } |

通过SpringBoot执行可以清楚的发现，在SpringBoot里面所有对象信息的注入配置操作，都直接通过一行简单的字符串实现了，而且最终也能够保持与之前同样的运行效果。

## Bean配置

在使用Spring进行开发配置的时候有两类选择：\*.xml配置文件、配置的Bean（@Configure），于是在SpringBoot的开发世界里面，为了继续崇尚所谓的“零配置”，提供有一种简单的支持，也就是说如果现在你真的有配置需要通过\*.xml文件编写，但是又不想出现配置文件的话，这个时候最简单的做法就是使用Bean的方式来进行类的配置。

前提：该配置程序的Bean所在的包必须是程序启动类所在的包的子包之中，这样才可以自动扫描到。

1. 下面准备一个程序，建立一个业务接口，而后定义这个接口的子类：

|  |
| --- |
| package com.kuhnwei.advance.service;public interface MessageService {  String info(); } |
| package com.kuhnwei.advance.service.impl; import com.kuhnwei.advance.service.MessageService;  import org.springframework.stereotype.Service;  @Servicepublic class MessageServiceImpl implements MessageService {  @Override  public String info() {  return "www.kuhnwei.com";  } } |

1. 建立控制层类，进行对象的配置注入

|  |
| --- |
| package com.kuhnwei.advance.web; import com.kuhnwei.advance.service.MessageService; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.bind.annotation.RestController;@RestController public class MessageController extends AbstracBaseController {  @Autowired  private MessageService messageService;  @RequestMapping(value = "/", method = RequestMethod.*GET*)  public String index() {  return this.messageService.info();  } } |

1. 建立一个测试类

|  |
| --- |
| package com.kuhnwei.advance.advance; import com.kuhnwei.advance.StartSpringBootMain; import com.kuhnwei.advance.web.MessageController; import org.junit.Test; import org.junit.runner.RunWith; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.boot.test.context.SpringBootTest; import org.springframework.test.context.junit4.SpringJUnit4ClassRunner; import org.springframework.test.context.web.WebAppConfiguration;@SpringBootTest(classes = StartSpringBootMain.class) @RunWith(SpringJUnit4ClassRunner.class) @WebAppConfiguration public class MessageControllerTest {  @Autowired  private MessageController messageController;  @Test  public void index() {  System.*out*.println(this.messageController.index());  } } |

1. 下面就利用以上的程序来了解一下什么叫做Bean配置，为了清楚的发现Bean的特点删除掉业务层实现子类中的“@Service”注解，也就是说这个对象现在无法直接被注入，于是下面在启动类所在包的子包里面建立一个配置程序类：com.kuhnwei.advance.ServiceConfig。

|  |
| --- |
| package com.kuhnwei.advance.config; import com.kuhnwei.advance.service.impl.MessageServiceImpl; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration;@Configuration public class ServiceConfig {  // 此处返回的是一个Spring的配置Bean,与xml的“<bean>”等价  @Bean  public MessageServiceImpl getMessageService() {  return new MessageServiceImpl();  } } |

此时采用了自动扫描Bean的模式来进行了相关对象的配置。

1. SSM或SSH开发框架出现的时间比较长，现在迁移到SpringBoot之中，那么如果说现在你已经有一个非常完善的xml配置文件出现了，那么难道还需要将整个的xml配置文件转换为Bean配置吗？为了防止这类情况出现，SpringBoot也支持有配置文件的读取，例如：下面创建一个spring-common.xml配置文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans   http://www.springframework.org/schema/beans/spring-beans.xsd">  <bean id="messageService" class="com.kuhnwei.advance.service.impl.MessageServiceImpl"/> </beans> |

1. 随后可以在程序类上使用xml进行配置加载：

|  |
| --- |
| package com.kuhnwei.advance; import org.springframework.boot.SpringApplication; import org.springframework.boot.autoconfigure.SpringBootApplication; import org.springframework.context.annotation.ImportResource;@SpringBootApplication @ImportResource(locations = {"classpath:spring-common.xml"}) public class StartSpringBootMain {  public static void main(String[] args) {  SpringApplication.*run*(StartSpringBootMain.class);  } } |

如果此时所配置的两个Bean没有名字那么在进行注入的时候一定会出现重复的错误，而这个错误在新版本里面将其做了完善，不过如果要想在开发之中准确的注入指定的对象，则需要使用名字完成：

|  |
| --- |
| @Resource(name="messageService") private MessageService messageService; |

这样才可以准确的找到所需要的注入的实体对象。

## 模板渲染

在之前所见到的信息显示发现都是以Rest风格进行显示，但是很明显在实际的开发之中，所有数据的显示最终都应该交由页面完成，但是这个页面并不是\*.jsp页面，而是普通的\*.html页面，而且最为重要的是，此处所使用的渲染的页面采用的是模板方式的显示，而在Java开发行业，对于前台的显示模板常见的一共有三类技术：FreeMarker、Velocity、thymeleaf（推荐使用），于是下面就利用thymeleaf实现一个简单的模板渲染操作。

1. 如果要想在项目之中去使用thymeleaf模板，那么应该首先进行相关支持依赖库的导入，修改pom.xml配置文件：

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

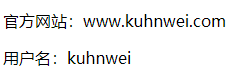
1. 本次的开发依然要通过一个控制层跳转到页面之中进行信息显示。在SpringMVC的时代使用的是ModelAndView传递，而现在在springboot里面如果要传递直接在方法中定义一个Model参数即可。

|  |
| --- |
| package com.kuhnwei.advance.web; import org.springframework.stereotype.Controller; import org.springframework.ui.Model; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod;@Controller public class MessageController extends AbstracBaseController {  @RequestMapping(value = "/show", method = RequestMethod.*GET*)  public String show(String mid, Model model) {  // request属性传递包装  model.addAttribute("url", "www.kuhnwei.com");  model.addAttribute("mid", mid);   // 此处只返回一个路径，该路径没有设置后缀，后缀默认是\*.html  return "message/message\_show";  } } |

1. 现在的控制器之中使用的是“@Controller”注解，所以此时执行该控制器的方法会进行跳转处理如果现在要进行跳转页面的定义，有严格要求：在CLASSPATH路径下（src/main/resources、src/main/view）必须建立有一个templates的目录，在这个目录里面保存有thymeleaf的所有相关页面，这些页面可以按照文件目录保存：
2. 编写message\_show.html页面（重要提示：该页面编写的时候所有的元素一定要完结）；

|  |
| --- |
| <!DOCTYPE html> <html lang="en" xmlns:th="http://www.w3.org/1999/xhtml"> <head>  <meta charset="UTF-8"/>  <title>SpringBoot模板渲染</title> </head> <body>  <p th:text="'官方网站：' + ${url}"/>  <p th:text="'用户名：' + ${mid}"/> </body> </html> |

1. 运行服务，而后输入访问路径：<http://localhost/advance/show?mid=kuhnwei>；



1. 如果现在我们所定义的要访问的页面不是通过控制器跳转的怎么办？那么为了解决这样的问题，可以考虑在thymeleaf所在的父路径之中建立一个static子目录，该目录保存的是所有静态页面；在以后的实际开发之中，像js、css、images等信息文件都要求放在static目录里面。
2. thymeleaf默认的访问的页面路径的后缀为\*.html，那么也可以通过修改application.yml配置文件进行变更：spring.thymeleaf.suffix: .htm

## profile配置

在实际的开发之中，由于开发（dev）、测试（beta）、运行（product）的环境不同，有可能需要选择不同的配置文件，所以在SpringBoot里面充分的考虑到此类问题，那么就专门提供有多个profile配置，但是对于多profile配置一定要区分是yml还是properties，这两种资源的配置是不同的。

1. 基于yml实现profile的配置处理：
   1. 在使用yml配置文件的时候所有可以使用的profile配置项都要求在一个文件之中编写；

|  |
| --- |
| **spring:  profiles:  active:** dev ---  **spring:  messages:  basename:** i18n/message,i18n/pages  **profiles:** dev **server:  port:** 80 ---  **spring:  messages:  basename:** i18n/message,i18n/pages  **profiles:** beta **server:  port:** 8080 ---  **spring:  messages:  basename:** i18n/message,i18n/pages  **profiles:** product **server:  port:** 80 |

最初进行profile切换的处理可以通过Maven的编译工具动态选择，但是在Spring里面可以方便的实现这种切换。

1. 此时所有的thymeleaf页面都保存在“src/main/resources”目录之中，而且所有的SpringBoot配置文件（\*.yml）这个文件也应该都一起输出到jar文件里面，那么就需要配置一个资源的引用，修改父pom中的pom.xml配置文件：

|  |
| --- |
| <resources>  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  <include>templates/\*\*/\*.\*</include>  <include>static/\*\*/\*.\*</include>  <include>i18n/\*\*/\*.\*</include>  </includes>  <filtering>false</filtering>  </resource>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.properties</include>  <include>\*\*/\*.yml</include>  <include>\*\*/\*.xml</include>  <include>\*\*/\*.tld</include>  </includes>  <filtering>false</filtering>  </resource> </resources> |

1. 随后可以将项目进行打包处理，在打包之后也可以动态切换profile配置：
   1. Maven打包：mvn clean package;
   2. 难道以后每一次访问都需要重新打包吗？也就是说为了方便用户的动态进行profile切换，可以在程序执行的时候设置一个执行的处理参数：  
      java –jar \*.jar --spring.profiles.active=beta;
2. 更换网站logo，把ico文件拷贝到“src/main/resources/static/images”目录之中，随后在页面之中进行引入：

|  |
| --- |
| <link rel="icon" type="image/x-icon" href="/images/\*.icon"/> |

1. 在进行profile配置的时候特别需要注意一点，如果你使用的是application.properties配置，这样的配置处理过程是不一样的，也就是说如果你基于属性文件完成配置，那么久需要编写多个属性文件内容：

|  |  |  |
| --- | --- | --- |
| **开发** | **测试** | **线上** |
| application-dev.properties | application-beta.properties | application-product.properties |
| server.port=8080 | server.port=9090 | server.port=80 |

1. 随后现在讲开发程序打成\*.jar文件的话，也同样可以在执行的时候使用“--spring.profiles.avtive=beta”动态指派不同的profile配置文件。

## 总结

整个的深入配置只是让大家更加清楚的理解了springboot所谓的“零配置”的特点，但是也会发现，在springboot里面一些环境的配置还是需要的，而且一些资源对象的注入处理更加简单了。

以后如果遇见更加复杂的配置，就需要通过Bean配置模式完成了。

# SpringBoot错误处理

在之前的程序里面如果一旦出现了错误之后就会出现一堆的大白板，这个白班会有一些错误信息（虽然这些错误信息你可能看不懂，但是这些错误信息依然要告诉给用户）。在SpringBoot里面针对于错误的处理一共提供三种方式：数据验证错误、错误也指派以及全局异常的处理。

## 数据验证

现在假设说要进行表单信息提交，肯定需要有一个表单，而后这个表单要讲数据提交到VO类中，所以现在的基本实现如下：

1. 建立一个Member.java的VO类：

|  |
| --- |
| package com.kuhnwei.advance.domain; import java.io.Serializable; import java.util.Date;public class Member implements Serializable {  private String mid;  private Integer age;  private Double salary;  private Date birthday; } |

1. 由于此时的程序之中需要进行日期的转换处理操作，那么久需要为其做一个转换处理的格式配置，修改AbstractBaseController类，追加如下的转换操作方法绑定：

|  |
| --- |
| @InitBinder public void initBinder(WebDataBinder binder) {  // 首先建立一个可以将字符串转换为日期的工具程序类  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  // 明确的描述此时需要注册一个日期格式的转化处理程序类  binder.registerCustomEditor(java.util.Date.class, new CustomDateEditor(sdf, true)); } |

1. 建立一个MemberController程序类，负责实现Member的控制层处理操作：

|  |
| --- |
| package com.kuhnwei.advance.web; import com.kuhnwei.advance.domain.Member; import org.springframework.stereotype.Controller; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RequestMethod; import org.springframework.web.bind.annotation.ResponseBody;@Controller public class MemberController extends AbstracBaseController {  @RequestMapping(value = "/goto\_add", method = RequestMethod.*GET*)  public String gotoAdd() {  return "member\_add";  }   @RequestMapping(value = "/add", method = RequestMethod.*POST*)  @ResponseBody  public Object add(Member vo) {  return vo;  } } |

1. 编写一个页面进行用户的表单填写：member\_add.html；

|  |
| --- |
| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8"/>  <title>SpringBoot模板渲染</title> </head> <body>  <form action="advance/add" method="post">  用户邮箱：<input type="text" name="mid" value="email@kuhnwei.com"/><br/>  用户年龄：<input type="text" name="age" value="18"/><br/>  用户工龄：<input type="text" name="salary" value="500000"/><br/>  用户生日：<input type="text" name="birthday" value="2000-01-01"/><br/>  <input type="submit" value="提交"/>  <input type="reset" value="重置"/>  </form> </body> </html> |

1. 此时的代码只是一个最为普通的处理操作，但是这个时候对于该程序也是存在有如下问题的：
   1. 如果某些数据没有输入，则内容是null，如果要进行严格控制，这些null不应该存在；
   2. 某些数据需要进行格式验证，例如：用户名应该是邮箱，这个的信息应该进行邮箱验证；

所以现在如果要想进行这些验证，SpringBoot里面有默认的支持，只不过这种支持未必是最好的，在SpringBoot里面为了方便用户编写验证专门提供有一个hibernate-validation.jar工具包是由hibernate开发框架提供的。

|  |  |  |
| --- | --- | --- |
| 验证注解 | | |
| **No.** | **注解** | **描述** |
| 1 | @Null | 被注释的元素必须为null |
| 2 | @NotNull | 被注释的元素不为null |
| 3 | @AssertTure | 被注释的元素必须为true |
| 4 | @AssertFalse | 被注释的元素必须为false |
| 5 | @Min(value) | 被注释的元素必须是一个数字，其值必须大于等于指定的最小值 |
| 6 | @Max(value) | 被注释的元素必须是一个数字，其值必须小于等于指定的最大值 |
| 7 | @DecimalMin(value) | 被注释的元素必须是一个数字，其值必须大于等于指定的最小值 |
| 8 | @DecimalMax(value) | 被注释的元素必须是一个数字，其值必须小于等于指定的最大值 |
| 9 | @Size(max=,min=) | 被注释的元素必须是一个数字，其值大小必须在指定的范围内 |
| 10 | @Digtis(integer,fraction) | 被注释的元素必须是一个数字，其值必须在可接受的范围内 |
| 11 | @Past | 被注释的元素必须是一个过去的日期 |
| 12 | @Future | 被注释的元素必须是一个将来的日期 |
| 13 | @Pattern(regex=,flag=) | 被注释的元素必须符合指定的正则表达式 |
| 14 | @NotBlank(message=) | 被注释的元素验证字符串非null，且长度必须大于0 |
| 15 | @Email | 被注释的元素必须是电子邮箱地址 |
| 16 | @Length(min=,max=) | 被注释的字符串的大小必须在指定的范围内 |
| 17 | @NotEmpty | 被注释的字符串必须非空 |
| 18 | @Range(min=,max=,message=) | 被注释的元素必须在合适的范围内 |

1. 如果要想进行验证，那么首先要解决的问题就必须是错误的提示信息问题，而在SpringBoot里面对于错误信息的保存，都要求其保存在ValidationMessages.properties文件，在“src/main/resources”目录中建立此文件；

|  |
| --- |
| member.mid.notnull.error=用户名不允许为空！ member.mid.email.error=用户名的注册必须输入正确的邮箱！ member.mid.length.error=用户名的格式错误！ member.age.notnull.error=年龄不允许为空 member.age.digits.error=年龄必须是合法数字！ member.salary.notnull.error=工资不允许为空！ member.salary.digits.error=工资必须是合法数字！ member.birthday.notnull.error=生日不允许为空！ |

提示：你一个表单就需要编写这么多的配置项，那么如果要有几百个表单呢？这样的配置项太可怕了，所以最好的数据检测还是利用拦截器处理最合适。

1. 修改Member.java程序类追加验证的处理方式：

|  |
| --- |
| package com.kuhnwei.advance.domain; import org.hibernate.validator.constraints.Email; import javax.validation.constraints.Digits; import javax.validation.constraints.NotNull; import java.io.Serializable; import java.util.Date;public class Member implements Serializable {  @NotNull(message = "{member.mid.notnull.error}")  @Email(message = "{member.mid.email.error}")  private String mid;  @NotNull(message = "member.age.notnull.error")  @Digits(integer = 3,fraction = 0,message = "{member.age.digits.error}")  private Integer age;  @NotNull(message = "{member.salary.notnull.error}")  @Digits(integer = 20,fraction = 2,message = "{member.salary.digits.error}")  private Double salary;  @NotNull(message = "{member.birthday.notnull.error}")  private Date birthday; } |

1. 修改MemberController类中的add()方法来观察错误信息的显示：

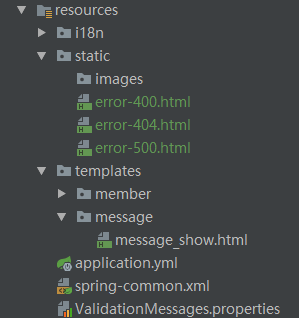
|  |
| --- |
| @RequestMapping(value = "/add", method = RequestMethod.*POST*) @ResponseBody public Object add(@Valid Member vo, BindingResult result) {  // 现在表示执行的验证出现错误  if (result.hasErrors()) {  // 获取全部错误信息  Iterator<ObjectError> iterator = result.getAllErrors().iterator();  while(iterator.hasNext()) {  // 取出每一个错误  ObjectError error = iterator.next();  System.*out*.println("【错误信息】code = " + error.getCode() + ", message = " + error.getDefaultMessage());  }  return result.getAllErrors();  } else {  return vo;  } } |

对于此类的验证大家理解即可，不需要将其作为重点，但是需要清楚，默认情况下SpringBoot提供的数据验证需要通过注解以及一系列的资源文件进行定义后才可以使用，而且所有的错误都必须用户自己来处理，这一点的设计不如直接编写具体的反射拦截器方便。

## 处理错误页

错误页绝对是所有的WEB项目之中必须具有的一项信息显示处理，但是在传统的WEB项目开发过程之中，错误页都是在web.xml文件之中进行配置的，不过遗憾的是SpringBoot之中并不存在有web.xml配置文件这一项，那么如果要想进行错误页的处理，最好的做法是需要根据每一个错误代码创建一个属于自己的错误显示页。

1. 所有的错误页都是普通的静态文件，那么建议在“src/main/resources/static”目录下创建几个常见的错误页（常见的错误的HTTP返回编码：404、500、400）



1. 添加一个错误页的配置类，在启动类中编写一个错误页的配置项；

|  |
| --- |
| package com.kuhnwei.advance.config; import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer; import org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer; import org.springframework.boot.web.servlet.ErrorPage; import org.springframework.context.annotation.Bean; import org.springframework.context.annotation.Configuration; import org.springframework.http.HttpStatus;@Configuration public class ErrorPageConfig {  @Bean  public EmbeddedServletContainerCustomizer containerCustomizer() {  return new EmbeddedServletContainerCustomizer() {  @Override  public void customize(ConfigurableEmbeddedServletContainer container) {  ErrorPage errorPage400 = new ErrorPage(HttpStatus.*BAD\_REQUEST*, "/error-400.html");  ErrorPage errorPage404 = new ErrorPage(HttpStatus.*NOT\_FOUND*, "/error-404.html");  ErrorPage errorPage500 = new ErrorPage(HttpStatus.*INTERNAL\_SERVER\_ERROR*, "/error-500.html");  container.addErrorPages(errorPage400, errorPage404, errorPage500);  }  };  } } |

那么此时只要出现了错误，就会找到相应的http状态码，而后跳转到指定的错误路径上进行显示。

## 全局异常

下面首先来观察一个程序代码，例如：现在建立一个控制器，而后这个控制器自己抛出一个异常。

|  |
| --- |
| @RequestMapping(value = "/get") @ResponseBody public String get() {  System.*out*.println("除法计算：" + (10 / 0));  return "hello world"; } |

如果此时配置有错误页，那么这个时候错误会统一跳转到500所在的路径上进行错误的显示，但是如果说现在希望能够显示出错误更加详细的内容呢？

所以这个时候可以单独定义一个页面进行错误的信息显示处理，而这个页面，可以定义在“src/main/templates/error.html”，这个页面里面要求可以输出一些信息；

1. 定义一个全局的异常处理类；

|  |
| --- |
| package com.kuhnwei.advance.handler; import org.springframework.web.bind.annotation.ControllerAdvice; import org.springframework.web.bind.annotation.ExceptionHandler; import org.springframework.web.servlet.ModelAndView; import javax.servlet.http.HttpServletRequest; // 作为一个控制层的切面处理 @ControllerAdvice public class GlobalExceptionHandler {  // 定义错误显示页 error.html  public static final String *DEFAULT\_ERROR\_VIEW* = "error";  // 所有的异常都是Exception的子类  @ExceptionHandler(Exception.class)  public ModelAndView defaultErrorHandler(HttpServletRequest request, Exception e) {  // 出现异常之后会跳转到此方法  //设置跳转路径  ModelAndView mav = new ModelAndView(*DEFAULT\_ERROR\_VIEW*);  // 将异常对象传递过去  mav.addObject("exception", e);  // 获得请求的路径  mav.addObject("url", request.getRequestURL());  return mav;  } } |

1. 定义error.html页面：

|  |
| --- |
| <!DOCTYPE html> <html lang="en" xmlns:th="http://www.w3.org/1999/xhtml"> <head>  <meta charset="UTF-8"/>  <title>SpringBoot模板渲染</title> </head> <body>  <p th:text="${url}"/>  <p th:text="${exception.message}"/> </body> </html> |

对于全局异常信息显示除了采用以上的跳转处理方式之外，也可以做得简单一些，使用rest进行显示。

范例：修改全局异常处理类

|  |
| --- |
| package com.kuhnwei.advance.handler; import org.springframework.web.bind.annotation.ExceptionHandler; import org.springframework.web.bind.annotation.RestControllerAdvice; import org.springframework.web.servlet.ModelAndView; import javax.servlet.http.HttpServletRequest;// 作为一个控制层的切面处理 @RestControllerAdvice public class GlobalExceptionHandler {  // 定义错误显示页 error.html  public static final String *DEFAULT\_ERROR\_VIEW* = "error";  // 所有的异常都是Exception的子类  @ExceptionHandler(Exception.class)  public Object defaultErrorHandler(HttpServletRequest request, Exception e) {  class ErrorInfo {  private Integer code;  private String message;  private String url;  public Integer getCode() {  return code;  }  public void setCode(Integer code) {  this.code = code;  }  public String getMessage() {  return message;  }  public void setMessage(String message) {  this.message = message;  }  public String getUrl() {  return url;  }  public void setUrl(String url) {  this.url = url;  }  }  ErrorInfo info = new ErrorInfo();  // 标记一个错误信息类型  info.setCode(100);  info.setMessage(e.getMessage());  info.setUrl(request.getRequestURL().toString());  return info;  } } |

如果现在要想把异常的信息显示的更加华丽一些（不是面对所有用户），那么最好的做法就是使用全局异常处理的方式完成。

# 模板1

## 模板1.1

后续还有 SpringBoot整合mybatis、redis、shiro、mq等内容 …