SpringBoot

# SpringBoot课程介绍

需要该课程资料同学，可以关注微信公众号“架构师余胜军”，回复SpringBoot即可领取该文档和资料。

官方网址:www.mayikt.com

余胜军个人微信：yushengjun644

# SpringBoot介绍

## 1.1、SpringBoot简介

SpringBoot 是一个快速开发的框架, 封装了Maven常用依赖、能够快速的整合第三方框架；简化XML配置，全部采用注解形式，内置Tomcat、Jetty、Undertow，帮助开发者能够实现快速开发，SpringBoot的Web组件 默认集成的是SpringMVC框架。

SpringBoot原理介绍：

1. 能够帮助开发者实现快速整合第三方框架 （原理：Maven依赖封装）

2. 去除xml配置 完全采用注解化 （原理：Spring体系中内置注解方式）

3. 无需外部Tomcat、内部实现服务器（原理：Java语言支持内嵌入Tomcat服务器）

## 1.2、系统要求：

Java1.8及以上

Spring Framework 5.0及以上

**本课程采用Java1.8版本、SpringBoot2.1.8版本当前最新版本**

## 1.3、SpringBoot和SpringMVC区别

SpringBoot 是一个快速开发的框架,能够快速的整合第三方框架，简化XML配置，全部采用注解形式，内置Tomcat容器,帮助开发者能够实现快速开发，SpringBoot的Web组件 默认集成的是SpringMVC框架。

SpringMVC是控制层。

## 1.4、SpringBoot和SpringCloud区别

SpringBoot 是一个快速开发的框架,能够快速的整合第三方框架，简化XML配置，全部采用注解形式，内置Tomcat容器,帮助开发者能够实现快速开发，SpringBoot的Web组件 默认集成的是SpringMVC框架。

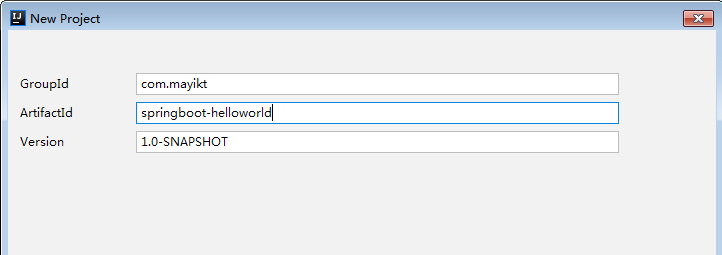
SpringMVC是控制层。

SpringCloud依赖与SpringBoot组件，使用SpringMVC编写Http协议接口，同时SpringCloud是一套完整的微服务解决框架。

# 二、快速入门

## 2.1、创建一个Maven工程

**名为”springboot-helloworld” 类型为Jar工程项目**



## 2.2、pom文件引入依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.8.RELEASE </version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |
| **spring-boot-starter-parent作用**  **在pom.xml中引入spring-boot-start-parent,spring官方的解释叫什么stater poms,它可以提供dependency management,也就是说依赖管理，引入以后在申明其它dependency的时候就不需要version了，后面可以看到。**  **spring-boot-starter-web作用**  **springweb 核心组件** |

## 2.3、编写HelloWorld服务

创建package命名为com.mayikt.controller（根据实际情况修改）

创建HelloController类，内容如下

|  |
| --- |
| **@RestController**  **@EnableAutoConfiguration**  **public class HelloController {**  **@RequestMapping("/hello")**  **public String index() {**  **return "Hello World";**  **}**  **public static void main(String[] args) {**  **SpringApplication.*run*(HelloController.class, args);**  **}**  **}** |

## 2.4、@RestController

**在上加上RestController 表示修饰该Controller所有的方法返回JSON格式,直接可以编写**

**Restful接口**

**注意该注解是SpringMVC提供的哦！**

## 2.5、@EnableAutoConfiguration

注解:作用在于让 Spring Boot   根据应用所声明的依赖来对 Spring 框架进行自动配置  
这个注解告诉Spring Boot根据添加的jar依赖猜测你想如何配置Spring。由于spring-boot-starter-web添加了Tomcat和Spring MVC，所以auto-configuration将假定你正在开发一个web应用并相应地对Spring进行设置。

## 2.6 SpringApplication.*run*(HelloController.class, args);

标识为启动类

## 2.7、SpringBoot启动方式1

Springboot默认端口号为8080

|  |
| --- |
| **@RestController**  **@EnableAutoConfiguration**  **public class HelloController {**  **@RequestMapping("/hello")**  **public String index() {**  **return "Hello World";**  **}**  **public static void main(String[] args) {**  **SpringApplication.*run*(HelloController.class, args);**  **}**  **}** |

启动主程序，打开浏览器访问http://localhost:8080/index，可以看到页面输出Hello World

## 2.8、SpringBoot启动方式2

@ComponentScan(basePackages = "com.mayikt.controller")---控制器扫包范围

|  |
| --- |
| @ComponentScan(basePackages = "com.mayikt.controller")  @EnableAutoConfiguration  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

## 2.9、SpringBoot启动方式3

@SpringBootApplication

@SpringBootApplication 被 @Configuration、@EnableAutoConfiguration、@ComponentScan 注解所修饰，换言之 Springboot 提供了统一的注解来替代以上三个注解

**扫包范围：在启动类上加上@SpringBootApplication注解,当前包下或者子包下所有的类都可以扫到。**

# Web开发

## 3.1、静态资源访问

在我们开发Web应用的时候，需要引用大量的js、css、图片等静态资源。

默认配置

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。

微服务项目

前后端分离

前端----vue----前端工程师

后端---springboot--后端工程师

动静分离 部署cdn上

Cdn 减少带宽距离传输 减少自己服务器带宽；

## 3.2、渲染Web页面

Com.mayikt.controller---视图层 渲染我们页面

Com.mayikt.service---业务逻辑层

Com.mayikt.dao---数据库访问层

前后端

渲染Web页面

在之前的示例中，我们都是通过@RestController来处理请求，所以返回的内容为json对象。那么如果需要渲染html页面的时候，要如何实现呢？

模板引擎 能够非常好的帮助seo搜索到该网页

在动态HTML实现上Spring Boot依然可以完美胜任，并且提供了多种模板引擎的默认配置支持，所以在推荐的模板引擎下，我们可以很快的上手开发动态网站。

Spring Boot提供了默认配置的模板引擎主要有以下几种：

* Thymeleaf
* FreeMarker
* Velocity
* Groovy
* Mustache

Spring Boot建议使用这些模板引擎，避免使用JSP，若一定要使用JSP将无法实现Spring Boot的多种特性，具体可见后文：支持JSP的配置

当你使用上述模板引擎中的任何一个，它们默认的模板配置路径为：src/main/resources/templates。当然也可以修改这个路径，具体如何修改，可在后续各模板引擎的配置属性中查询并修改。

### 3.3 YML与Properties用法

SpringBoot支持两种配置方式,一种是properties文件,一种是yml。

使用yml可以减少配置文件的重复性。

例如：application.properties配置

|  |
| --- |
| **mayikt.name**=**mayikt mayikt.age**=**22** |

例如：application.yml配置

在企业中application.yml方式用的是比较多的；

## 3.4、[使用Freemarker模板引擎渲染web视图](http://blog.didispace.com/springbootweb/" \t "_blank)

### 3.4.1、pom文件引入:

|  |
| --- |
| <!-- 引入freeMarker的依赖包. -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-freemarker</artifactId>  </dependency> |

### 3.4.2、后台代码

在src/main/resources/创建一个templates文件夹,后缀为\*.ftl

|  |
| --- |
| **@RequestMapping("/index")**  **public String index(Map<String, Object> map) {**  **map.put("name","每特");**  **return "index";**  **}** |

### 3.4.3、前台代码

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="en">  <meta charset="UTF-8" />  <title></title>  </head>  <body>  ${name}  </body>  </html> |

### 3.4.4、Freemarker其他用法

|  |
| --- |
| @RequestMapping("/freemarkerIndex")  **public** String index(Map<String, Object> result) {  result.put("name", "yushengjun");  result.put("sex", "0");  List<String> listResult = **new** ArrayList<String>();  listResult.add("zhangsan");  listResult.add("lisi");  listResult.add("mayikt");  result.put("listResult", listResult);  **return** "index";  }  **<!DOCTYPE html>**  **<html>**  **<head lang="en">**  **<meta charset="UTF-8" />**  **<title>首页</title>**  **</head>**  **<body>**  **${name}**  **<#if sex=="1">**  **男**  **<#elseif sex=="2">**  **女**  **<#else>**  **其他**    **</#if>**  **<#list userlist as user>**  **${user}**  **</#list>**  **</body>**  **</html>** |

两种方法   
1 用符号代替: > gt , >= gte ,< lt , <= lte  
2 加括号 <#if(x>y)>

### 3.4.4、Freemarker配置

新建application.yml文件

|  |
| --- |
| **spring**:  **http**:  **encoding**:  **force**: **true** *### 模版引擎编码为UTF-8* **charset**: UTF-8  **freemarker**:  **allow-request-override**: **false  cache**: **false  check-template-location**: **true  charset**: UTF-8  **content-type**: text/html; charset=utf-8  **expose-request-attributes**: **false  expose-session-attributes**: **false  expose-spring-macro-helpers**: **false** *## 模版文件结尾.ftl* **suffix**: .ftl  *## 模版文件目录* **template-loader-path**: classpath:/templates |

## 3.5使用thymeleaf渲染Web页面

### 3.5.1什么是thymeleaf

thymeleaf是一款用于渲染XML/XHTML/HTML5内容的模板引擎，类似JSP，Velocity，FreeMaker等，它也可以轻易的与Spring MVC等Web框架进行集成作为Web应用的模板引擎。

### 3.5.2 Maven依赖

|  |
| --- |
| *<!--Spring SpringMVC -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> *<!--引入thymeleaf的依赖-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> |

### 3.5.3 配置文件新增

|  |
| --- |
| *###ThymeLeaf配置* **spring**:  **thymeleaf**:  *#prefix：指定模板所在的目录* **prefix**: classpath:/templates/  *#check-tempate-location: 检查模板路径是否存在* **check-template-location**: **true** *#cache: 是否缓存，开发模式下设置为false，避免改了模板还要重启服务器，线上设置为true，可以提高性能。* **cache**: **true  suffix**: .html  **encoding**: UTF-8  **mode**: HTML5 |

### 3.5.4 案例代码

|  |
| --- |
| **import** com.mayikt.entity.UserEntity; **import** org.springframework.stereotype.Controller; **import** org.springframework.web.bind.annotation.RequestMapping;  **import** java.util.Map;  */\*\*  \** ***@ClassName*** *IndexController  \** ***@Author*** *蚂蚁课堂余胜军 QQ644064779 www.mayikt.com  \** ***@Version*** *V1.0  \*\*/* @Controller **public class** IndexController {  @RequestMapping(**"/myThymeleaf"**)  **public** String myThymeleaf(Map<String, Object> result) {  result.put(**"user"**, **new** UserEntity(**"mayikt"**, 22));  **return "myThymeleaf"**;  } }  */\*\*  \** ***@ClassName*** *UserEntity  \** ***@Author*** *蚂蚁课堂余胜军 QQ644064779 www.mayikt.com  \** ***@Version*** *V1.0  \*\*/* **public class** UserEntity {  **private** String **userName**;  **private** Integer **age**;   **public** UserEntity(String userName, Integer age) {  **this**.**userName** = userName;  **this**.**age** = age;  }   **public** String getUserName() {  **return userName**;  }   **public** Integer getAge() {  **return age**;  }   **public void** setUserName(String userName) {  **this**.**userName** = userName;  }   **public void** setAge(Integer age) {  **this**.**age** = age;  } }  <!DOCTYPE **html**> *<!--需要在HTML文件中加入以下语句： -->* <**html lang="en" xmlns:th="http://www.thymeleaf.org"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Show User</**title**> </**head**> <**body**> <**table**>  姓名:<**span th:text="${user.userName}"**></**span**>  年龄:<**span th:text="${user.age}"**></**span**> </**table**> </**body**> </**html**> |

### 3.5.4 高级写法

循环语句：

|  |
| --- |
| <**ul th:each="user:${userList}"**>  <**li th:text="${user.userName}"**></**li**>  <**li th:text="${user.age}"**></**li**>  <**br**> </**ul**> |

If判断：

<**span th:if="${user.age>17}"**>已经成年啦</**span**>  
<**span th:if="${user.age<17}"**>未成年</**span**>

详细可以更多语法可以查看https://www.thymeleaf.org/documentation.html

# 数据库访问

### 4.1、springboot整合[使用JdbcTemplate](http://blog.didispace.com/springbootdata1/" \t "_blank)

#### 4.1.1 pom文件引入

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.1.8.RELEASE </version>  </parent>  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-jdbc</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>mysql</groupId>**  **<artifactId>mysql-connector-java</artifactId>**  **<version>5.1.21</version>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-test</artifactId>**  **<scope>test</scope>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-web</artifactId>**  **</dependency>**  **</dependencies>** |

#### 4.1.2 application.yml新增配置

|  |
| --- |
| **spring**:  **datasource**:  **url**: jdbc:mysql://localhost:3306/test  **username**: root  **password**: root  **driver-class-name**: com.mysql.jdbc.Driver |

#### 4.1.3 UserService类

|  |
| --- |
| @Service **public class** UserServiceImpl **implements** UserService {  @Autowired  **private** JdbcTemplate **jdbcTemplate**;   @Override  **public** Boolean inserUser(String name, Integer age) {  **int** update = **jdbcTemplate**.update(**"insert into users values(null,?,?);"**, name, age);  **return** update > 0 ? **true** : **false**;  } } |

#### 4.1.4 App类

|  |
| --- |
| **@ComponentScan(basePackages = "com.mayikt")**  **@EnableAutoConfiguration**  **public class App {**  **public static void main(String[] args) {**  **SpringApplication.*run*(App.class, args);**  **}**  **}** |

#### 4.1.5数据库表结构

|  |
| --- |
| CREATE TABLE `users` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(32) NOT NULL COMMENT '用户名称',  `age` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8; |

### 4.2、springboot整合[使用](http://blog.didispace.com/springbootdata1/" \t "_blank)mybatis

#### 4.2.1、pom文件引入

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.1.8.RELEASE</**version**> </**parent**> <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  *<!-- springboot 整合mybatis -->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.1.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.21</**version**>  </**dependency**>  </**dependencies**> |

#### 4.2.2、application.yml

|  |
| --- |
| **spring**:  **datasource**:  **url**: jdbc:mysql://localhost:3306/test  **username**: root  **password**: root  **driver-class-name**: com.mysql.jdbc.Driver |

#### 4.2.3、Mapper代码

|  |
| --- |
| **public interface UserMapper {**  **@Select("SELECT \* FROM USERS WHERE NAME = #{name}")**  **User findByName(@Param("name") String name);**  **@Insert("INSERT INTO USERS(NAME, AGE) VALUES(#{name}, #{age})")**  **int insert(@Param("name") String name, @Param("age") Integer age);**  **}** |

#### 4.2.4、启动方式

|  |
| --- |
| @SpringBootApplication @MapperScan(**"com.mayikt.mapper"**) **public class** AppMybatis {  **public static void** main(String[] args) {  SpringApplication.*run*(AppMybatis.**class**);  } } |

### 4、springboot整合多数据源

同学们思考下，你们在项目中有使用到多数据源吗？

#### 4.4.1配置文件中新增两个数据源

application.yml

|  |
| --- |
| **spring**:  **datasource**:  *###会员数据库* **member**:  **jdbc-url**: jdbc:mysql://localhost:3306/user  **username**: root  **password**: root  **driver-class-name**: com.mysql.jdbc.Driver  *###订单数据库* **order**:  **jdbc-url**: jdbc:mysql://localhost:3306/order  **username**: root  **password**: root  **driver-class-name**: com.mysql.jdbc.Driver |

备注：如果是SpringBoot2配置多数据源 ，报如下错误：

“jdbcUrl is required with driverClassName.”或者Cause: java.lang.IllegalArgumentException: dataSource or dataSourceClassName or jdbcUrl is required.] with root cause

解决方案：

spring.datasource.url 和spring.datasource.driverClassName，换成

spring.datasource.jdbc-url和spring.datasource.driver-class-name

#### 4.4.2数据库数据源相关配置

##### 会员数据源

|  |
| --- |
| @Configuration @MapperScan(basePackages = **"com.mayikt.member.mapper"**, sqlSessionFactoryRef = **"memberSqlSessionFactory"**) **public class** MemberDataSourceConfig {   */\*\*  \* 将会员db注册到容器中  \*  \** ***@return*** *\*/* @Bean(name = **"memberDataSource"**)  @ConfigurationProperties(prefix = **"spring.datasource.member"**)  **public** DataSource memberDataSource() {  **return** DataSourceBuilder.*create*().build();  }   */\*\*  \* 将会员SqlSessionFactory注册到容器中  \*  \** ***@param dataSource*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"memberSqlSessionFactory"**)  **public** SqlSessionFactory memberSqlSessionFactory(@Qualifier(**"memberDataSource"**) DataSource dataSource) **throws** Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(memberDataSource());  **return** sqlSessionFactoryBean.getObject();  }   */\*\*  \* 创建会员管理器  \*  \** ***@param dataSource*** *\** ***@return*** *\*/* @Bean(name = **"memberTransactionManager"**)  **public** DataSourceTransactionManager memberTransactionManager(@Qualifier(**"memberDataSource"**) DataSource dataSource) {  **return new** DataSourceTransactionManager(dataSource);  }   */\*\*  \* 创建订单sqlSesion模版  \*  \** ***@param sqlSessionFactory*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"memberSqlSessionTemplate"**)  **public** SqlSessionTemplate menberSqlSessionTemplate(  @Qualifier(**"memberSqlSessionFactory"**) SqlSessionFactory sqlSessionFactory) **throws** Exception {  **return new** SqlSessionTemplate(sqlSessionFactory);  }   } |

##### 订单数据源

|  |
| --- |
| @Configuration @MapperScan(basePackages = **"com.mayikt.order.mapper"**, sqlSessionFactoryRef = **"orderSqlSessionFactory"**) **public class** OrderDataSourceConfig {   */\*\*  \* 将订单db注册到容器中  \*  \** ***@return*** *\*/* @Bean(name = **"orderDataSource"**)  @ConfigurationProperties(prefix = **"spring.datasource.order"**)  **public** DataSource orderDataSource() {  **return** DataSourceBuilder.*create*().build();  }   */\*\*  \* 将订单SqlSessionFactory注册到容器中  \*  \** ***@param dataSource*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"orderSqlSessionFactory"**)  **public** SqlSessionFactory orderSqlSessionFactory(@Qualifier(**"orderDataSource"**) DataSource dataSource) **throws** Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(orderDataSource());  **return** sqlSessionFactoryBean.getObject();  }   */\*\*  \* 创建订单管理器  \*  \** ***@param dataSource*** *\** ***@return*** *\*/* @Bean(name = **"orderTransactionManager"**)  **public** DataSourceTransactionManager orderTransactionManager(@Qualifier(**"orderDataSource"**) DataSource dataSource) {  **return new** DataSourceTransactionManager(dataSource);  }   */\*\*  \* 创建订单sqlSesion模版  \*  \** ***@param sqlSessionFactory*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"orderSqlSessionTemplate"**)  **public** SqlSessionTemplate menberSqlSessionTemplate(  @Qualifier(**"orderSqlSessionFactory"**) SqlSessionFactory sqlSessionFactory) **throws** Exception {  **return new** SqlSessionTemplate(sqlSessionFactory);  }   } |

#### 4.4.2创建分包Mapper

##### 会员mapper

|  |
| --- |
| **public interface** MemberMapper {  @Insert(**"insert into users values(null,#{name},#{age});"**)  **public int** addUser(@Param(**"name"**) String name, @Param(**"age"**) Integer age); } |

##### 订单mapper

|  |
| --- |
| **public interface** OrderMapper {  @Insert(**"insert into order\_number values(null,#{number});"**)  **int** inserOrder(@Param(**"number"**) String number); } |

#### 4.4.3启动项目

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

#### 4.4.4Maven相关依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.1.8.RELEASE</**version**> </**parent**> <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  *<!-- springboot 整合mybatis -->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.1.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.21</**version**>  </**dependency**>  </**dependencies**> |

#### 4.4.5 数据库表结构

|  |
| --- |
| CREATE TABLE `users` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `name` varchar(32) NOT NULL COMMENT '用户名称',  `age` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=5 DEFAULT CHARSET=utf8;  CREATE TABLE `order\_number` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `order\_name` varchar(255) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=5 DEFAULT CHARSET=utf8; |

# 事务管理

### 5.1.1Springboot整合事务管理

springboot默认集成事务,只主要在方法上加上@Transactional即可

### 5.1.2SpringBoot分布式事务管理

使用springboot+jta+atomikos 分布式事物Transactional管理

#### 5.1.2.1 多数据源分布式事务案例

|  |
| --- |
| @Service **public class** MemberService {  @Autowired  **private** MemberMapper **memberMapper**;  @Autowired  **private** OrderMapper **orderMapper**;   @Transactional(transactionManager = **"memberTransactionManager"**)  **public int** addUser(String userName, Integer age) {  **int** result = **memberMapper**.addUser(userName, age);  **orderMapper**.inserOrder(userName);  **int** j = 1 / age;  **return** result;  } } |

#### 5.1.2.1新增配置文件信息

|  |
| --- |
| **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-jta-atomikos</artifactId>**  **</dependency>** |

#### 5.1.2.2新增配置文件信息

|  |
| --- |
| **spring**:  **datasource**:  *###会员数据库* **member**:  **url**: jdbc:mysql://localhost:3306/user  **username**: root  **password**: root  **borrowConnectionTimeout**: 30  **loginTimeout**: 30  **maintenanceInterval**: 60  **maxIdleTime**: 60  **maxLifetime**: 20000  **maxPoolSize**: 25  **minPoolSize**: 3  **uniqueResourceName**: orderDatasource  *###订单数据库* **order**:  **url**: jdbc:mysql://localhost:3306/order  **username**: root  **password**: root  **borrowConnectionTimeout**: 30  **loginTimeout**: 30  **maintenanceInterval**: 60  **maxIdleTime**: 60  **maxLifetime**: 20000  **maxPoolSize**: 25  **minPoolSize**: 3  **uniqueResourceName**: memberDatasource |

#### 5.1.2.3 读取配置文件信息

|  |
| --- |
| @ConfigurationProperties(prefix = **"spring.datasource.member"**) @Data **public class** MemberConfig {  **private** String **url**;  **private** String **userName**;  **private** String **passWord**;  **private int minPoolSize**;  **private int maxPoolSize**;  **private int maxLifetime**;  **private int borrowConnectionTimeout**;  **private int loginTimeout**;  **private int maintenanceInterval**;  **private int maxIdleTime**;  **private** String **testQuery**;   **private** String **uniqueResourceName**; }  @ConfigurationProperties(prefix = **"spring.datasource.order"**) @Data **public class** OrderConfig {  **private** String **url**;  **private** String **userName**;  **private** String **passWord**;  **private int minPoolSize**;  **private int maxPoolSize**;  **private int maxLifetime**;  **private int borrowConnectionTimeout**;  **private int loginTimeout**;  **private int maintenanceInterval**;  **private int maxIdleTime**;  **private** String **testQuery**;   **private** String **uniqueResourceName**;  } |

#### 5.1.2.4 创建多数据源

|  |
| --- |
| @Configuration @MapperScan(basePackages = **"com.mayikt.member.mapper"**, sqlSessionTemplateRef = **"memberSqlSessionTemplate"**) **public class** MemberDataSourceConfig {  *//@Configuration xml MemberDataSourceConfig.xml   /\*\*  \* 创建我们的DataSource  \*  \** ***@return*** *\*/* @Bean(**"memberDataSource"**)  **public** DataSource memberDataSource(MemberConfig memberConfig) **throws** SQLException {  *// 1.创建MysqlXADataSource* MysqlXADataSource mysqlXaDataSource = **new** MysqlXADataSource();  mysqlXaDataSource.setUrl(memberConfig.getUrl());  mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(**true**);  mysqlXaDataSource.setPassword(memberConfig.getPassWord());  mysqlXaDataSource.setUser(memberConfig.getUserName());  mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(**true**);    *// 2.将本地事务注册到创 Atomikos全局事务* AtomikosDataSourceBean xaDataSource = **new** AtomikosDataSourceBean();  xaDataSource.setXaDataSource(mysqlXaDataSource);  xaDataSource.setUniqueResourceName(memberConfig.getUniqueResourceName());  xaDataSource.setMinPoolSize(memberConfig.getMinPoolSize());  xaDataSource.setMaxPoolSize(memberConfig.getMaxPoolSize());  xaDataSource.setMaxLifetime(memberConfig.getMaxLifetime());  xaDataSource.setBorrowConnectionTimeout(memberConfig.getBorrowConnectionTimeout());  xaDataSource.setLoginTimeout(memberConfig.getLoginTimeout());  xaDataSource.setMaintenanceInterval(memberConfig.getMaintenanceInterval());  xaDataSource.setMaxIdleTime(memberConfig.getMaxIdleTime());  xaDataSource.setTestQuery(memberConfig.getTestQuery());  **return** xaDataSource;  }   */\*\*  \* 创建我们的SqlSessionFactory  \*  \** ***@param dataSource*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"memberSqlSessionFactory"**)  **public** SqlSessionFactory memberSqlSessionFactory(@Qualifier(**"memberDataSource"**) DataSource dataSource) **throws** Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(dataSource);  **return** sqlSessionFactoryBean.getObject();  }   */\*\*  \* 创建订单sqlSesion模版  \*  \** ***@param sqlSessionFactory*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"memberSqlSessionTemplate"**)  **public** SqlSessionTemplate memberSqlSessionTemplate(  @Qualifier(**"memberSqlSessionFactory"**) SqlSessionFactory sqlSessionFactory) **throws** Exception {  **return new** SqlSessionTemplate(sqlSessionFactory);  }   }  @Configuration @MapperScan(basePackages = **"com.mayikt.order.mapper"**, sqlSessionTemplateRef = **"orderSqlSessionTemplate"**) **public class** OrderDataSourceConfig {  *//@Configuration xml orderDataSourceConfig.xml   /\*\*  \* 创建我们的DataSource  \*  \** ***@return*** *\*/* @Bean(**"orderDataSource"**)  **public** DataSource orderDataSource(OrderConfig orderConfig) **throws** SQLException {  *// 1.创建MysqlXADataSource* MysqlXADataSource mysqlXaDataSource = **new** MysqlXADataSource();  mysqlXaDataSource.setUrl(orderConfig.getUrl());  mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(**true**);  mysqlXaDataSource.setPassword(orderConfig.getPassWord());  mysqlXaDataSource.setUser(orderConfig.getUserName());  mysqlXaDataSource.setPinGlobalTxToPhysicalConnection(**true**);    *// 2.将本地事务注册到创 Atomikos全局事务* AtomikosDataSourceBean xaDataSource = **new** AtomikosDataSourceBean();  xaDataSource.setXaDataSource(mysqlXaDataSource);  xaDataSource.setUniqueResourceName(orderConfig.getUniqueResourceName());  xaDataSource.setMinPoolSize(orderConfig.getMinPoolSize());  xaDataSource.setMaxPoolSize(orderConfig.getMaxPoolSize());  xaDataSource.setMaxLifetime(orderConfig.getMaxLifetime());  xaDataSource.setBorrowConnectionTimeout(orderConfig.getBorrowConnectionTimeout());  xaDataSource.setLoginTimeout(orderConfig.getLoginTimeout());  xaDataSource.setMaintenanceInterval(orderConfig.getMaintenanceInterval());  xaDataSource.setMaxIdleTime(orderConfig.getMaxIdleTime());  xaDataSource.setTestQuery(orderConfig.getTestQuery());  **return** xaDataSource;  }   */\*\*  \* 创建我们的SqlSessionFactory  \*  \** ***@param dataSource*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"orderSqlSessionFactory"**)  **public** SqlSessionFactory orderSqlSessionFactory(@Qualifier(**"orderDataSource"**) DataSource dataSource) **throws** Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(dataSource);  **return** sqlSessionFactoryBean.getObject();  } *// // /\*\* // \* 创建会员管理器 // \* // \* @param dataSource // \* @return // \*/ // @Bean(name = "orderTransactionManager") // public DataSourceTransactionManager orderTransactionManager(@Qualifier("orderDataSource") DataSource dataSource) { // return new DataSourceTransactionManager(dataSource); // }   /\*\*  \* 创建订单sqlSesion模版  \*  \** ***@param sqlSessionFactory*** *\** ***@return*** *\** ***@throws*** *Exception  \*/* @Bean(name = **"orderSqlSessionTemplate"**)  **public** SqlSessionTemplate orderSqlSessionTemplate(  @Qualifier(**"orderSqlSessionFactory"**) SqlSessionFactory sqlSessionFactory) **throws** Exception {  **return new** SqlSessionTemplate(sqlSessionFactory);  }   } |

如果多数据源使用事务报错的话

cted single matching bean but found 2: memberTransactionManager,orderTransactionManager

@Transactional(transactionManager = "memberTransactionManager")

明确指定使用那个事务管理器即可

#### 5.1.2.4 启动加载配置

@EnableConfigurationProperties({OrderConfig.**class**, MemberConfig.**class**})

# 整合热部署

### 6.1、Spring Boot集成lombok让代码更简洁

#### 6.1.2添加lombok依赖

|  |
| --- |
| <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency> |

#### 6.1.3实体类演示

|  |
| --- |
| @Slf4j  @Data  **public** **class** UserEntity {  // @Getter  // @Setter  **private** String userName;  // @Getter  // @Setter  **private** Integer age;  @Override  **public** String toString() {  **return** "UserEntity [userName=" + userName + ", age=" + age + "]";  }  **public** **static** **void** main(String[] args) {  UserEntity userEntity = **new** UserEntity();  userEntity.setUserName("zhangsan");  userEntity.setAge(20);  System.***out***.println(userEntity.toString());  ***log***.info("####我是日志##########");  }  } |

#### 6.1.4其他特性

|  |
| --- |
| @Data 标签，生成getter/setter toString()等方法  @NonNull : 让你不在担忧并且爱上NullPointerException  @CleanUp : 自动资源管理：不用再在finally中添加资源的close方法  @Setter/@Getter : 自动生成set和get方法  @ToString : 自动生成toString方法  @EqualsAndHashcode : 从对象的字段中生成hashCode和equals的实现  @NoArgsConstructor/@RequiredArgsConstructor/@AllArgsConstructor  自动生成构造方法  @Data : 自动生成set/get方法，toString方法，equals方法，hashCode方法，不带参数的构造方法  @Value : 用于注解final类  @Builder : 产生复杂的构建器api类  @SneakyThrows : 异常处理（谨慎使用）  @Synchronized : 同步方法安全的转化  @Getter(lazy=true) :  @Log : 支持各种logger对象，使用时用对应的注解，如：@Log4 |

### 6.2、Spring Boot整合热部署框架

#### 6.2.1什么是热部署

修改java类或页面或者静态文件，不需要手动重启

原理：类加载器

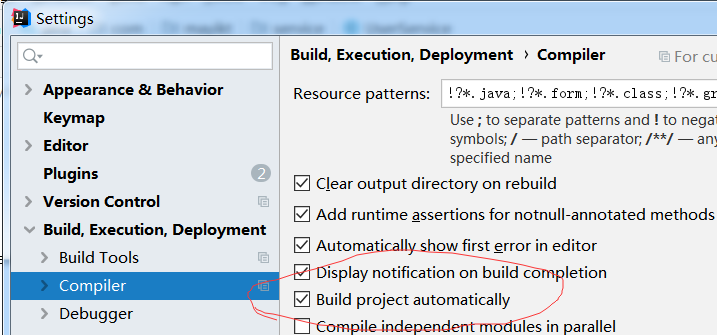
适合于本地开发环境

#### 6.2.1 Maven依赖

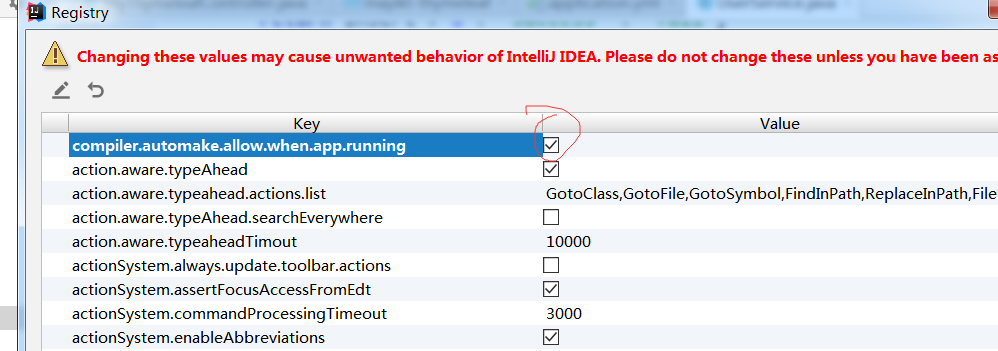
|  |
| --- |
| *<!--SpringBoot热部署配置 -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-devtools</**artifactId**>  <**scope**>runtime</**scope**>  <**optional**>true</**optional**> </**dependency**> |

#### 6.3.3 Idea工具设置

1. “File” -> “Settings” -> “Build,Execution,Deplyment” -> “Compiler”，选中打勾 “Build project automatically” 。

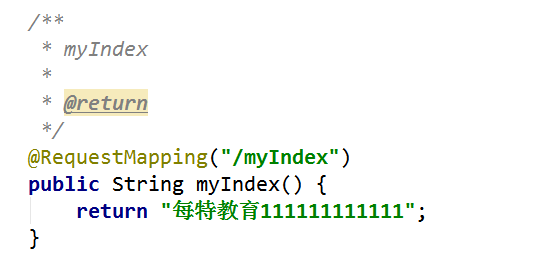


2） 组合键：“Shift+Ctrl+Alt+/” ，选择 “Registry” ，选中打勾 “compiler.automake.allow.when.app.running”



#### 6.3.4效果演示

按住保存键，自动帮我实现重启



# 整合配置文件

1.在springboot整合配置文件，分成两大类：

application.properties

application.yml

或者是

Bootstrap.properties

Bootstrap.yml

相对于来说yml文件格式写法更加精简，减少配置文件的冗余性。

2.加载顺序：

bootstrap.yml 先加载 application.yml后加载

bootstrap.yml 用于应用程序上下文的引导阶段。

bootstrap.yml 由父Spring ApplicationContext加载。

1. 区别：

bootstrap.yml 和 application.yml 都可以用来配置参数。

bootstrap.yml 用来程序引导时执行，应用于更加早期配置信息读取。可以理解成系统级别的一些参数配置，这些参数一般是不会变动的。一旦bootStrap.yml 被加载，则内容不会被覆盖。

application.yml 可以用来定义应用级别的， 应用程序特有配置信息，可以用来配置后续各个模块中需使用的公共参数等。

分布式配置中心：

Properties在线转换yml格式网址：https://www.toyaml.com/index.html

## 7.1使用@value注解：

|  |
| --- |
| @Value(**"${mayikt.name}"**) **private** String **name**; |

## 7.2@ConfigurationProperties

|  |
| --- |
| *<!--导入配置文件处理器，配置文件进行绑定就会有提示-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**> </**dependency**>  **import** org.springframework.boot.context.properties.ConfigurationProperties; **import** org.springframework.stereotype.Component;  */\*\*  \** ***@ClassName*** *MayiktUserEntity  \** ***@Author*** *蚂蚁课堂余胜军 QQ644064779 www.mayikt.com  \** ***@Version*** *V1.0  \*\*/* @Component @ConfigurationProperties(prefix = **"mayikt"**) **public class** MayiktUserEntity {  **private** String **addres**;  **private** String **age**;  **private** String **name**;   **public** String getAddres() {  **return addres**;  }   **public** String getAge() {  **return age**;  }   **public** String getName() {  **return name**;  }   **public void** setAddres(String addres) {  **this**.**addres** = addres;  }   **public void** setAge(String age) {  **this**.**age** = age;  }   **public void** setName(String name) {  **this**.**name** = name;  }   @Override  **public** String toString() {  **return "MayiktUserEntity{"** +  **"addres='"** + **addres** + **'\''** +  **", age='"** + **age** + **'\''** +  **", name='"** + **name** + **'\''** +  **'}'**;  } }  **mayikt**:  **addres**: www.mayikt.com  **age**: 22  **name**: mayikt  @Autowired **private** MayiktUserEntity **mayiktUserEntity**;  @RequestMapping(**"/getNameAndAgeAddres"**) **public** String getNameAndAgeAddres() {  **return mayiktUserEntity**.toString(); } |

## 7.3配置文件占位符

在SpringBoot的配置文件中，我们可以使用SpringBoot提供的的一些随机数

${random.value}、${random.int}、${random.long}

${random.int(10)}、${random.int[1024,65536]}

-${app.name:默认值} 来制定找不到属性时的默认值

## 7.4多环境配置

|  |
| --- |
| **spring**:  **profiles**:  **active**: pre |

|  |
| --- |
| application-dev.yml：开发环境  application-test.yml：测试环境  application-prd.yml：生产环境 |

## 7.5、核心配置

**server**:  
 **port**: 8081  
 **servlet**:  
 **context-path**: /mayikt

Springboot 默认的情况下整合tomcat容器

# 日志管理

### 8.1.[使用logback记录日志](http://blog.didispace.com/springbootlog4j/" \t "_blank)

Springboot 已经默认帮你整合好了logback

日志输出文件在当前项目路径log文件夹下

#### Maven依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**> </**dependency**> |

#### Logback配置

|  |
| --- |
| <**configuration**>  *<!--本文主要输出日志为控制台日志，系统日志，sql日志，异常日志-->  <!-- %m输出的信息,%p日志级别,%t线程名,%d日期,%c类的全名,,,, -->  <!--控制台-->* <**appender name="console" class="ch.qos.logback.core.ConsoleAppender"**>  <**encoder**>  <**pattern**>%d %p (%file:%line\)- %m%n</**pattern**>  <**charset**>UTF-8</**charset**>  </**encoder**>  </**appender**>   *<!--系统info级别日志-->  <!--<File> 日志目录，没有会自动创建-->  <!--<rollingPolicy>日志策略，每天简历一个日志文件，或者当天日志文件超过64MB时-->  <!--encoder 日志编码及输出格式-->* <**appender name="fileLog"  class="ch.qos.logback.core.rolling.RollingFileAppender"**>  <**File**>log/file/fileLog.log</**File**>  <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  <**fileNamePattern**>log/file/fileLog.log.%d.%i</**fileNamePattern**>  <**timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"**>  *<!-- or whenever the file size reaches 64 MB -->* <**maxFileSize**>64 MB</**maxFileSize**>  </**timeBasedFileNamingAndTriggeringPolicy**>  </**rollingPolicy**>  <**encoder**>  <**pattern**>  %d %p (%file:%line\)- %m%n  </**pattern**>  <**charset**>UTF-8</**charset**>  *<!-- 此处设置字符集 -->* </**encoder**>  </**appender**>   *<!--sql日志-->* <**appender name="sqlFile"  class="ch.qos.logback.core.rolling.RollingFileAppender"**>  <**File**>log/sql/sqlFile.log</**File**>  <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  <**fileNamePattern**>log/sql/sqlFile.log.%d.%i</**fileNamePattern**>  <**timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"**>  *<!-- or whenever the file size reaches 64 MB -->* <**maxFileSize**>64 MB</**maxFileSize**>  </**timeBasedFileNamingAndTriggeringPolicy**>  </**rollingPolicy**>  *<!--对记录事件进行格式化。负责两件事，一是把日志信息转换成字节数组，二是把字节数组写入到输出流。-->* <**encoder**>  *<!--用来设置日志的输入格式-->* <**pattern**>  %d %p (%file:%line\)- %m%n  </**pattern**>  <**charset**>UTF-8</**charset**>  *<!-- 此处设置字符集 -->* </**encoder**>  </**appender**>    *<!--异常日志-->* <**appender name="errorFile"  class="ch.qos.logback.core.rolling.RollingFileAppender"**>  <**File**>log/error/errorFile.log</**File**>  <**rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"**>  <**fileNamePattern**>log/error/errorFile.%d.log.%i</**fileNamePattern**>  <**timeBasedFileNamingAndTriggeringPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP"**>  *<!-- or whenever the file size reaches 64 MB -->* <**maxFileSize**>64 MB</**maxFileSize**>  </**timeBasedFileNamingAndTriggeringPolicy**>  </**rollingPolicy**>  *<!--对记录事件进行格式化。负责两件事，一是把日志信息转换成字节数组，二是把字节数组写入到输出流。-->* <**encoder**>  *<!--用来设置日志的输入格式-->* <**pattern**>  %d %p (%file:%line\)- %m%n  </**pattern**>  <**charset**>UTF-8</**charset**>  *<!-- 此处设置字符集 -->* </**encoder**>  *<!--  日志都在这里 过滤出 error  使用 try {}catch (Exception e){} 的话异常无法写入日志，可以在catch里用logger.error()方法手动写入日志  -->* <**filter class="ch.qos.logback.classic.filter.LevelFilter"**>  <**level**>ERROR</**level**>  <**onMatch**>ACCEPT</**onMatch**>  <**onMismatch**>DENY</**onMismatch**>  </**filter**>  </**appender**>  *<!-- 日志输出级别 -->  <!--All\DEBUG\INFO\WARN\ERROR\FATAL\OFF-->  <!--打印info级别日志，分别在控制台，fileLog，errorFile输出  异常日志在上面由过滤器过滤出ERROR日志打印  -->* <**root level="INFO"**>  <**appender-ref ref="fileLog"** />  <**appender-ref ref="console"** />  <**appender-ref ref="errorFile"** />  </**root**>   *<!--打印sql至sqlFile文件日志-->* <**logger name="com.dolphin.mapper" level="DEBUG" additivity="false"**>  <**appender-ref ref="console"** />  <**appender-ref ref="sqlFile"** />  </**logger**> </**configuration**> |

#### application

|  |
| --- |
| *###指定读取logback配置文件* **logging**:  **config**: classpath:log/logback.xml |

#### 测试案例

|  |
| --- |
| @RestController @Slf4j **public class** MyIndexService {  @RequestMapping(**"/getName"**) **public** String getName(String name, **int** age) {  ***log***.info(**"name:{},age:{}"**, name, age);  **return** name; }  } |

#### 日志级别

ALL 最低等级的，用于打开所有日志记录。

TRACE designates finer-grained informational events than the DEBUG.Since:1.2.12，很低的日志级别，一般不会使用。

**DEBUG 指出细粒度信息事件对调试应用程序是非常有帮助的，主要用于开发过程中打印一些运行信息。**

**INFO 消息在粗粒度级别上突出强调应用程序的运行过程。打印一些你感兴趣的或者重要的信息，这个可以用于生产环境中输出程序运行的一些重要信息，但是不能滥用，避免打印过多的日志。**

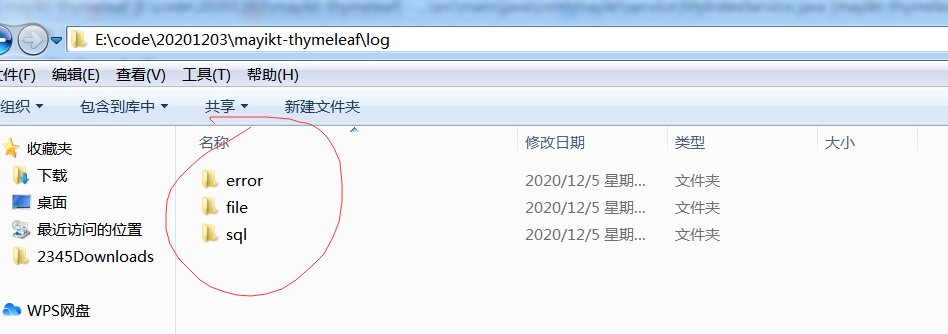
WARN 表明会出现潜在错误的情形，有些信息不是错误信息，但是也要给程序员的一些提示

**ERROR 指出虽然发生错误事件，但仍然不影响系统的继续运行。打印错误和异常信息，如果不想输出太多的日志，可以使用这个级别。**

FATAL 指出每个严重的错误事件将会导致应用程序的退出。这个级别比较高了。重大错误，这种级别你可以直接停止程序了。

OFF 最高等级的，用于关闭所有日志记录。

#### 日志目录



### 8.2.[使用log4j记录日志](http://blog.didispace.com/springbootlog4j/" \t "_blank)

日志级别

机制：如果一条日志信息的级别大于等于配置文件的级别，就记录。

trace：追踪，就是程序推进一下，可以写个trace输出

debug：调试，一般作为最低级别，trace基本不用。

info：输出重要的信息，使用较多

warn：警告，有些信息不是错误信息，但也要给程序员一些提示。

error：错误信息。用的也很多。

fatal：致命错误。

输出源

CONSOLE（输出到控制台）

FILE（输出到文件）

格式

SimpleLayout：以简单的形式显示

HTMLLayout：以HTML表格显示

PatternLayout：自定义形式显示

#### 8.2.1新建log4j配置文件

文件名称log4j.properties

|  |
| --- |
| *#log4j.rootLogger=CONSOLE,info,error,DEBUG* **log4j.rootLogger**=**DEBUG,error,CONSOLE,info log4j.appender.CONSOLE**=**org.apache.log4j.ConsoleAppender  log4j.appender.CONSOLE.layout**=**org.apache.log4j.PatternLayout  log4j.appender.CONSOLE.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n    log4j.logger.info**=**info log4j.appender.info**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.info.layout**=**org.apache.log4j.PatternLayout  log4j.appender.info.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.info.datePattern**=**'.'yyyy-MM-dd log4j.appender.info.Threshold** = **info  log4j.appender.info.append**=**true log4j.appender.info.File**=**E:/code/log/info.log  log4j.logger.error**=**error  log4j.appender.error**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.error.layout**=**org.apache.log4j.PatternLayout  log4j.appender.error.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.error.datePattern**=**'.'yyyy-MM-dd log4j.appender.error.Threshold** = **error  log4j.appender.error.append**=**true log4j.appender.error.File**=**E:/code/log/error.log  log4j.logger.DEBUG**=**DEBUG log4j.appender.DEBUG**=**org.apache.log4j.DailyRollingFileAppender log4j.appender.DEBUG.layout**=**org.apache.log4j.PatternLayout  log4j.appender.DEBUG.layout.ConversionPattern**=**%d{yyyy-MM-dd-HH-mm} [%t] [%c] [%p] - %m%n  log4j.appender.DEBUG.datePattern**=**'.'yyyy-MM-dd log4j.appender.DEBUG.Threshold** = **DEBUG  log4j.appender.DEBUG.append**=**true log4j.appender.DEBUG.File**=**E:/code/log/dubug.log**  **log4j代码**  private static final Logger logger = LoggerFactory.getLogger(IndexController.class); |

#### 8.2.2.Maven依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**> </**dependency**> *<!-- spring boot start -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter</**artifactId**>  <**exclusions**>  *<!-- 排除自带的logback依赖 -->* <**exclusion**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  *<!-- springboot-log4j -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j</**artifactId**>  <**version**>1.3.8.RELEASE</**version**> </**dependency**> |

#### 8.2.3.application

|  |
| --- |
| *###指定log4j.properties配置文件路径* **logging**:  **config**: classpath:log4j.properties |

### 8.3.[使用AOP统一处理Web请求日志](http://blog.didispace.com/springbootaoplog/" \t "_blank)

基于AOP实现 或者elk

在我们的方法的前后实现拦截 减少打印日志代码的冗余性的问题

#### 8.3.1 POM文件新增依赖

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

#### 6.3.2 AOP切面相关配置

|  |
| --- |
| @Aspect  @Component  **public** **class** WebLogAspect {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(WebLogAspect.**class**);  @Pointcut("execution(public \* com.mayikt.controller.\*.\*(..))")  **public** **void** webLog() {  }  @Before("webLog()")  **public** **void** doBefore(JoinPoint joinPoint) **throws** Throwable {  // 接收到请求，记录请求内容  ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  HttpServletRequest request = attributes.getRequest();  // 记录下请求内容  ***logger***.info("URL : " + request.getRequestURL().toString());  ***logger***.info("HTTP\_METHOD : " + request.getMethod());  ***logger***.info("IP : " + request.getRemoteAddr());  Enumeration<String> enu = request.getParameterNames();  **while** (enu.hasMoreElements()) {  String name = (String) enu.nextElement();  ***logger***.info("name:{},value:{}", name, request.getParameter(name));  }  }  @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);  }  } |

# 其他内容

### 9.1、[使用@Scheduled创建定时任务](http://blog.didispace.com/springbootscheduled/" \t "_blank)

在Spring Boot的主类中加入@EnableScheduling注解，启用定时任务的配置

@Component

**public** **class** **ScheduledTasks** {

**private** **static** **final** SimpleDateFormat dateFormat = **new** SimpleDateFormat("HH:mm:ss");

@Scheduled(fixedRate = 5000)

**public** **void** **reportCurrentTime**() {

System.out.println("现在时间：" + dateFormat.format(**new** Date()));

}

}

@Scheduled(cron = "0/2 \* \* \* \* \*") 写法：

<https://www.bejson.com/othertools/cron/>

Xxl-job

### 9.2、[使用@Async实现异步调用](http://blog.didispace.com/springbootasync/" \t "_blank)

启动加上@EnableAsync ,需要执行异步方法上加入 @Async

##### 异步应用场景

@Async实际就是多线程封装的

异步线程执行方法有可能会非常消耗cpu的资源，所以大的项目建议使用

Mq异步实现。

##### 整合线程池

|  |
| --- |
| **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.core.task.TaskExecutor; **import** org.springframework.scheduling.annotation.EnableAsync; **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  **import** java.util.concurrent.ThreadPoolExecutor;  @Configuration @EnableAsync **public class** ThreadPoolConfig {   */\*\*  \* 每秒需要多少个线程处理?  \* tasks/(1/taskcost)  \*/* **private int corePoolSize** = 3;   */\*\*  \* 线程池维护线程的最大数量  \* (max(tasks)- queueCapacity)/(1/taskcost)  \*/* **private int maxPoolSize** = 3;   */\*\*  \* 缓存队列  \* (coreSizePool/taskcost)\*responsetime  \*/* **private int queueCapacity** = 10;   */\*\*  \* 允许的空闲时间  \* 默认为60  \*/* **private int keepAlive** = 100;   @Bean  **public** TaskExecutor taskExecutor() {  ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();  *// 设置核心线程数* executor.setCorePoolSize(**corePoolSize**);  *// 设置最大线程数* executor.setMaxPoolSize(**maxPoolSize**);  *// 设置队列容量* executor.setQueueCapacity(**queueCapacity**);  *// 设置允许的空闲时间（秒）  //executor.setKeepAliveSeconds(keepAlive);  // 设置默认线程名称* executor.setThreadNamePrefix(**"thread-"**);  *// 设置拒绝策略rejection-policy：当pool已经达到max size的时候，如何处理新任务  // CALLER\_RUNS：不在新线程中执行任务，而是有调用者所在的线程来执行* executor.setRejectedExecutionHandler(**new** ThreadPoolExecutor.CallerRunsPolicy());  *// 等待所有任务结束后再关闭线程池* executor.setWaitForTasksToCompleteOnShutdown(**true**);  **return** executor;  }  }  **import** lombok.extern.slf4j.Slf4j; **import** org.springframework.scheduling.annotation.Async; **import** org.springframework.stereotype.Component;  */\*\*  \** ***@ClassName*** *MemberServiceAsync  \** ***@Author*** *蚂蚁课堂余胜军 QQ644064779 www.mayikt.com  \** ***@Version*** *V1.0  \*\*/* @Slf4j @Component **public class** MemberServiceAsync {  @Async(**"taskExecutor"**)  **public** String smsAsync() {  ***log***.info(**">02<"**);  **try** {  ***log***.info(**">正在发送短信..<"**);  Thread.*sleep*(3000);  } **catch** (Exception e) {   }  ***log***.info(**">03<"**);  **return "短信发送完成!"**;  } } |

##### 异步注解配置类

|  |
| --- |
| **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.core.task.TaskExecutor; **import** org.springframework.scheduling.annotation.EnableAsync; **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  **import** java.util.concurrent.ThreadPoolExecutor;  @Configuration @EnableAsync **public class** ThreadPoolConfig {   */\*\*  \* 每秒需要多少个线程处理?  \* tasks/(1/taskcost)  \*/* **private int corePoolSize** = 3;   */\*\*  \* 线程池维护线程的最大数量  \* (max(tasks)- queueCapacity)/(1/taskcost)  \*/* **private int maxPoolSize** = 3;   */\*\*  \* 缓存队列  \* (coreSizePool/taskcost)\*responsetime  \*/* **private int queueCapacity** = 10;   */\*\*  \* 允许的空闲时间  \* 默认为60  \*/* **private int keepAlive** = 100;   @Bean  **public** TaskExecutor taskExecutor() {  ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();  *// 设置核心线程数* executor.setCorePoolSize(**corePoolSize**);  *// 设置最大线程数* executor.setMaxPoolSize(**maxPoolSize**);  *// 设置队列容量* executor.setQueueCapacity(**queueCapacity**);  *// 设置允许的空闲时间（秒）  //executor.setKeepAliveSeconds(keepAlive);  // 设置默认线程名称* executor.setThreadNamePrefix(**"thread-"**);  *// 设置拒绝策略rejection-policy：当pool已经达到max size的时候，如何处理新任务  // CALLER\_RUNS：不在新线程中执行任务，而是有调用者所在的线程来执行* executor.setRejectedExecutionHandler(**new** ThreadPoolExecutor.CallerRunsPolicy());  *// 等待所有任务结束后再关闭线程池* executor.setWaitForTasksToCompleteOnShutdown(**true**);  **return** executor;  }  } |

##### 注意失效问题

注意：如果异步注解写当前自己类，有可能aop会失效，无法拦截注解，最终导致异步注解失效，需要经过代理类调用接口；

所以需要将异步的代码单独抽取成一个类调用接口。

### 9.3全局捕获异常

@ExceptionHandler 表示拦截异常

* @ControllerAdvice 是 controller 的一个辅助类，最常用的就是作为全局异常处理的切面类
* @ControllerAdvice 可以指定扫描范围
* @ControllerAdvice 约定了几种可行的返回值，如果是直接返回 model 类的话，需要使用 @ResponseBody 进行 json 转换
  + 返回 String，表示跳到某个 view
  + 返回 modelAndView
  + 返回 model + @ResponseBody

|  |
| --- |
| @ControllerAdvice **public class** MayiktExceptionHandler {   */\*\*  \* 拦截运行异常出现的错误~~~  \*  \** ***@return*** *\*/* @ExceptionHandler(RuntimeException.**class**)  @ResponseBody  **public** Map<Object, Object> exceptionHandler() {  Map<Object, Object> map = **new** HashMap<>();  map.put(**"error"**, **"500"**);  map.put(**"msg"**, **"系统出现错误~"**);  **return** map;  } } |

### 9.4发布打包

使用mvn package 打包

使用java –jar 包名

如果报错没有主清单,在pom文件中新增

|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>repackage</**goal**>  </**goals**>  </**execution**>  </**executions**>  <**configuration**>  <**mainClass**>com.mayikt.App</**mainClass**>  <**excludes**>  <**exclude**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  </**exclude**>  <**exclude**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  </**exclude**>  </**excludes**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |