

## 导读：概览

# 1. Spring Boot 是什么

我们知道，从 2002 年开始，Spring 一直在飞速的发展，如今已经成为了在 Java EE（Java Enterprise Edition）开发中真正意义上的标准，但是随着技术的发展，Java EE 使用 Spring 逐渐变得笨重起来，大量的 XML 文件存在于项目之中。繁琐的配置，整合第三方框架的配置问题，导致了开发和部署效率的降低。

2012 年 10 月，Mike Youngstrom 在 Spring jira 中创建了一个功能请求，要求在 **Spring 框架中支持无容器 Web 应用程序体系结构**。他谈到了在主容器引导 spring 容器内配置 Web 容器服务。这是 jira 请求的摘录：

我认为 Spring 的 Web 应用体系结构可以大大简化，如果它提供了从上到下利用 Spring 组件和配置模型的工具和参考体系结构。在简单的 main()方法引导的 Spring 容器内嵌入和统一这些常用 Web 容器服务的配置。

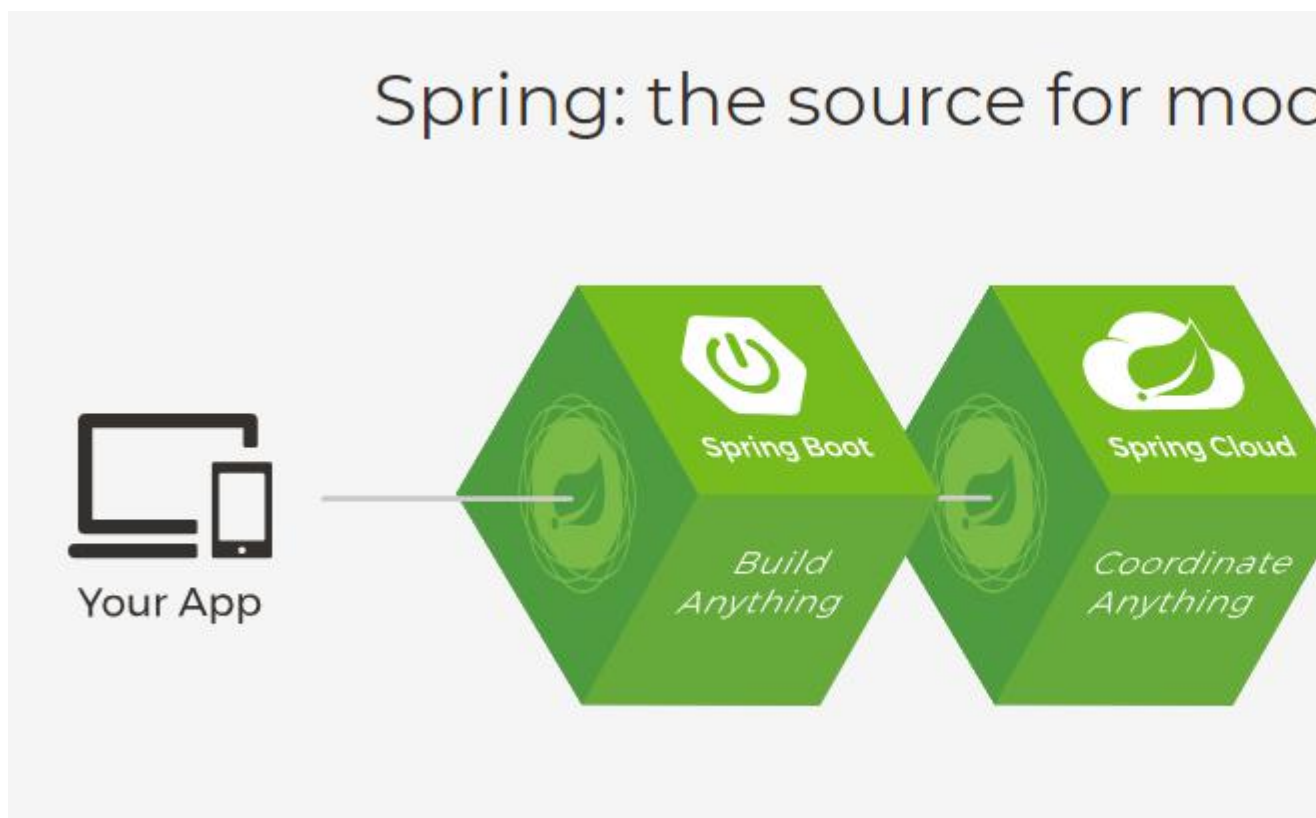
这一要求促使了 2013 年初开始的 Spring Boot 项目的研发，到今天，Spring Boot 的版本已经到了 2.0.3 RELEASE。Spring Boot 并不是用来替代 Spring 的解决方案，而是和 Spring 框架紧密结合用于提升 Spring 开发者体验的工具。

它集成了大量常用的第三方库配置，Spring Boot 应用中这些第三方库几乎可以是零配置的开箱即用（out-of-the-box），大部分的 Spring Boot 应用都只需要非常少量的配置代码（基于 Java 的配置），开发者能够更加专注于业务逻辑。

# 2. 为什么学习 Spring Boot

## 2.1 从 Spring 官方来看

我们打开 Spring 的[官方网站](#)，可以看到下图：



我们可以看到图中官方对 **Spring Boot** 的定位：*Build Anything*，Build 任何东西。**Spring Boot** 旨在尽可能快地启动和运行，并且只需最少的 **Spring** 前期配置。同时我们也来看一下官方对后面两个的定位：

**SpringCloud**: *Coordinate Anything*，协调任何事情； **SpringCloud Data Flow**: *Connect everything*，连接任何东西。

仔细品味一下，**Spring** 官网对 **Spring Boot**、**SpringCloud** 和 **SpringCloud Data Flow** 三者定位的措辞非常有味道，同时也可以看出，**Spring** 官方对这三个技术非常重视，是现在以及今后学习的重点（**SpringCloud** 相关达人课课程届时也会上线）。

## 2.2 从 Spring Boot 的优点来看

**Spring Boot** 有哪些优点？主要给我们解决了哪些问题呢？我们以下图来说明：



### 2.2.1 良好的基因

Spring Boot 是伴随着 Spring 4.0 诞生的，从字面理解，Boot 是引导的意思，因此 Spring Boot 旨在帮助开发者快速搭建 Spring 框架。Spring Boot 继承了原有 Spring 框架的优秀基因，使 Spring 在使用中更加方便快捷。



### 2.2.2 简化编码

举个例子，比如我们要创建一个 web 项目，使用 Spring 的朋友都知道，在使用 Spring 的时候，需要在 pom 文件中添加多个依赖，而 Spring Boot 则会帮助开发着快速启动

一个 web 容器，在 Spring Boot 中，我们只需要在 pom 文件中添加如下一个 starter-web 依赖即可。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId></dependency>
```

我们点击进入该依赖后可以看到，Spring Boot 这个 starter-web 已经包含了多个依赖，包括之前在 Spring 工程中需要导入的依赖，我们看一下其中的一部分，如下：

```
<!-- .....省略其他依赖 --><dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.0.7.RELEASE</version>
  <scope>compile</scope></dependency><dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.0.7.RELEASE</version>
  <scope>compile</scope></dependency>
```

由此可以看出，Spring Boot 大大简化了我们的编码，我们不用一个个导入依赖，直接一个依赖即可。

### 2.2.3 简化配置

Spring 虽然使 Java EE 轻量级框架，但由于其繁琐的配置，一度被人认为是“配置地狱”。各种 XML、Annotation 配置会让人眼花缭乱，而且配置多的话，如果出错了也很难找出原因。Spring Boot 更多的是采用 Java Config 的方式，对 Spring 进行配置。举个例子：

我新建一个类，但是我不需要 @Service 注解，也就是说，它是个普通的类，那么我们如何使它也成为 Bean 让 Spring 去管理呢？只需要 @Configuration 和 @Bean 两个注解即可，如下：

```
public class TestService {
    public String sayHello () {
        return "Hello Spring Boot!";
    }
}
import org.springframework.context.annotation.Bean;import
org.springframework.context.annotation.Configuration;
@Configurationpublic class JavaConfig {
```

```
@Bean
public TestService getTestService() {
    return new TestService();
}
```

`@Configuration` 表示该类是个配置类，`@Bean` 表示该方法返回一个 Bean。这样就把 `TestService` 作为 Bean 让 Spring 去管理了，在其他地方，我们如果需要使用该 Bean，和原来一样，直接使用 `@Resource` 注解注入进来即可使用，非常方便。

```
@Resourceprivate TestService testService;
```

另外，部署配置方面，原来 Spring 有多个 xml 和 properties 配置，在 Spring Boot 中只需要个 `application.yml` 即可。

#### 2.2.4 简化部署

在使用 Spring 时，项目部署时需要我们在服务器上部署 tomcat，然后把项目打成 war 包扔到 tomcat 里，在使用 Spring Boot 后，我们不需要在服务器上去部署 tomcat，因为 Spring Boot 内嵌了 tomcat，我们只需要将项目打成 jar 包，使用 `java -jar xxx.jar` 一键式启动项目。

另外，也降低对运行环境的基本要求，环境变量中有 JDK 即可。

#### 2.2.5 简化监控

我们可以引入 `spring-boot-start-actuator` 依赖，直接使用 REST 方式来获取进程的运行期性能参数，从而达到监控的目的，比较方便。但是 Spring Boot 只是个微框架，没有提供相应的服务发现与注册的配套功能，没有外围监控集成方案，没有外围安全管理方案，所以在微服务架构中，还需要 Spring Cloud 来配合一起使用。

### 2.3 从未来发展的趋势来看

微服务是未来发展的趋势，项目会从传统架构慢慢转向微服务架构，因为微服务可以使不同的团队专注于更小范围的工作职责、使用独立的技术、更安全更频繁地部署。而继承了 Spring 的优良特性，与 Spring 一脉相承，而且支持各种 REST API 的实现方式。Spring Boot 也是官方大力推荐的技术，可以看出，Spring Boot 是未来发展的一个大趋势。

## 3. 本课程能学到什么

本课程使用目前 Spring Boot 最新版本 2.0.3 RELEASE，课程文章均为作者在实际项目中剥离出来的场景和 demo，目标是带领学习者快速上手 Spring Boot，将 Spring Boot 相关技术点快速运用在微服务项目中。全篇分为两部分：基础篇和进阶篇。

基础篇（01—10 课）主要介绍 Spring Boot 在项目中最常使用的一些功能点，旨在带领学习者快速掌握 Spring Boot 在开发时需要的知识点，能够把 Spring Boot 相关技术运用到实际项目架构中去。该部分以 Spring Boot 框架为主线，内容包括 Json 数据封装、日志记录、属性配置、MVC 支持、在线文档、模板引擎、异常处理、AOP 处理、持久层集成等等。

进阶篇（11—17 课）主要是介绍 Spring Boot 在项目中的拔高一些的技术点，包括集成的一些组件，旨在带领学习者在项目中遇到具体的场景时能够快速集成，完成对应的功能。该部分以 Spring Boot 框架为主线，内容包括拦截器、监听器、缓存、安全认证、分词插件、消息队列等等。

认真读完该系列文章之后，学习者会快速了解并掌握 Spring Boot 在项目中最常用的技术点，作者课程的最后，会基于课程内容搭建一个 Spring Boot 项目的空架构，该架构也是从实际项目中剥离出来，学习者可以运用该架构于实际项目中，具备使用 Spring Boot 进行实际项目开发的能力。

课程所有源码提供免费下载：[下载地址](#)。

## 4. 适合阅读的人群

本课程适合以下人群阅读：

- 有一定的 Java 语言基础，了解 Spring、Maven 的在校学生或自学者
- 有传统项目经验，想往微服务方向发展的工作人员
- 热衷于新技术并对 Spring Boot 感兴趣的人员
  - 希望了解 Spring Boot 2.0.3 的研究人员

## 5. 本课程开发环境和插件

•

本课程的开发环境：

- 开发工具：IDEA 2017
- JDK 版本：JDK 1.8
- Spring Boot 版本：2.0.3 RELEASE
- Maven 版本：3.5.2

涉及到的插件：

- FastJson
- Swagger2
- Thymeleaf
- MyBatis
- Redis
- ActiveMQ
- Shiro
- Lucence

## 6. 课程目录

- 导读：课程概览
- 第 01 课：Spring Boot 开发环境搭建和项目启动
- 第 02 课：Spring Boot 返回 Json 数据及数据封装
- 第 03 课：Spring Boot 使用 slf4j 进行日志记录
- 第 04 课：Spring Boot 中的项目属性配置
- 第 05 课：Spring Boot 中的 MVC 支持
- 第 06 课：Spring Boot 集成 Swagger2 展现在线接口文档
- 第 07 课：Spring Boot 集成 Thymeleaf 模板引擎
- 第 08 课：Spring Boot 中的全局异常处理
- 第 09 课：Spring Boot 中的切面 AOP 处理
- 第 10 课：Spring Boot 中集成 MyBatis
- 第 11 课：Spring Boot 事务配置管理
- 第 12 课：Spring Boot 中使用监听器
- 第 13 课：Spring Boot 中使用拦截器
- 第 14 课：Spring Boot 中集成 Redis
- 第 15 课：Spring Boot 中集成 ActiveMQ
- 第 16 课：Spring Boot 中集成 Shiro
- 第 17 课：Spring Boot 中集成 Lucence
- 第 18 课：Spring Boot 搭建实际项目开发中的架构

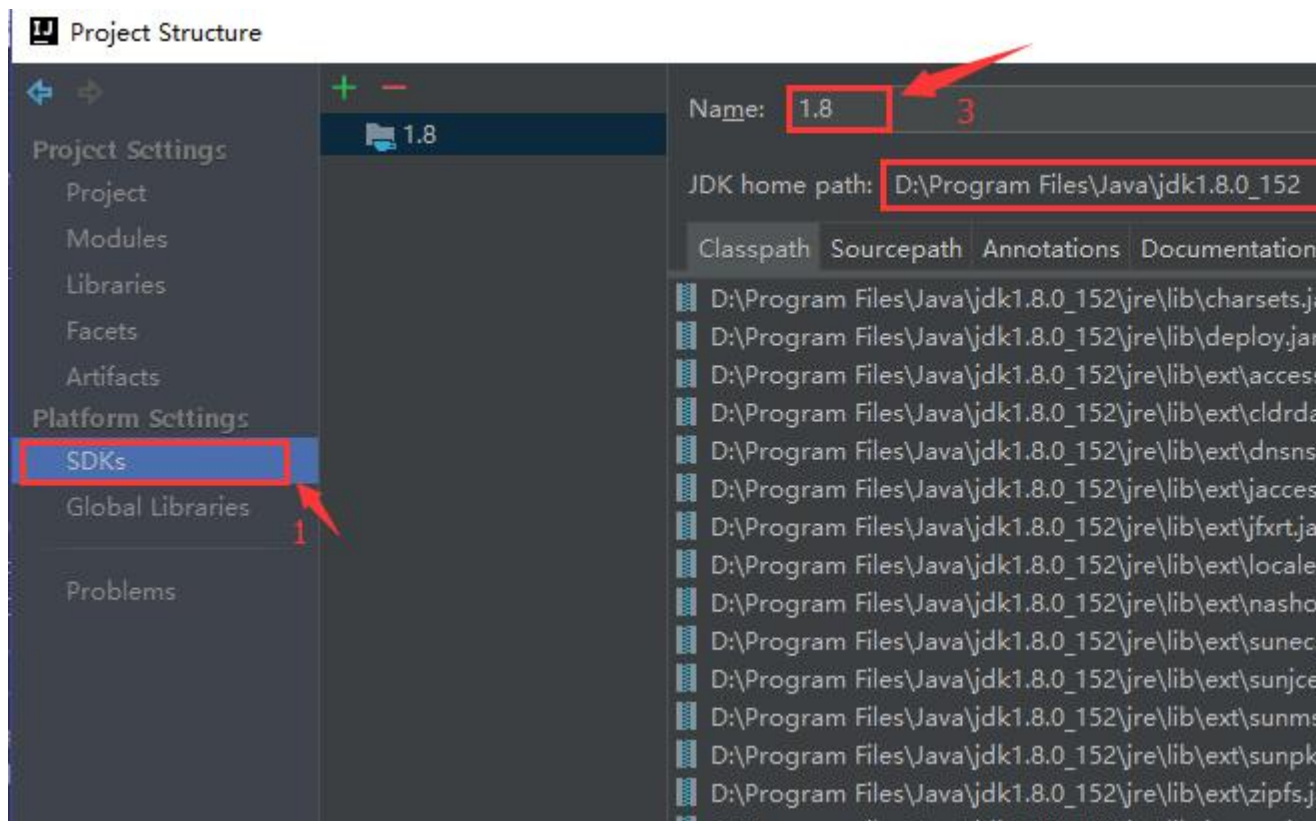
## 第 01 Spring Boot 开发环境搭建和项目启动

上一节对 SpringBoot 的特性做了一个介绍，本节主要对 **jdk 的配置**、**Spring Boot 工程的构建和项目的启动**、**Spring Boot 项目工程的结构**做一下讲解和分析。

### 1. jdk 的配置

本课程是使用 IDEA 进行开发，在 IDEA 中配置 jdk 的方式很简单，打开 File->Project Structure，如下图所：





1. 选择 SDKs
2. 在 JDK home path 中选择本地 jdk 的安装目录
3. 在 Name 中为 jdk 自定义名字

通过以上三步骤，即可导入本地安装的 jdk。如果是使用 STS 或者 eclipse 的朋友，可以通过两步骤添加：

- window->preference->java->Instalalled JRES 来添加本地 jdk。
- window-->preference-->java-->Compiler 选择 jre，和 jdk 保持一致。

## 2. Spring Boot 工程的构建

### 2.1 IDEA 快速构建

IDEA 中可以通过 File->New->Project 来快速构建 Spring Boot 工程。如下，选择 Spring Initializr，在 Project SDK 中选择刚刚我们导入的 jdk，点击 Next，到了项目的配置信息。

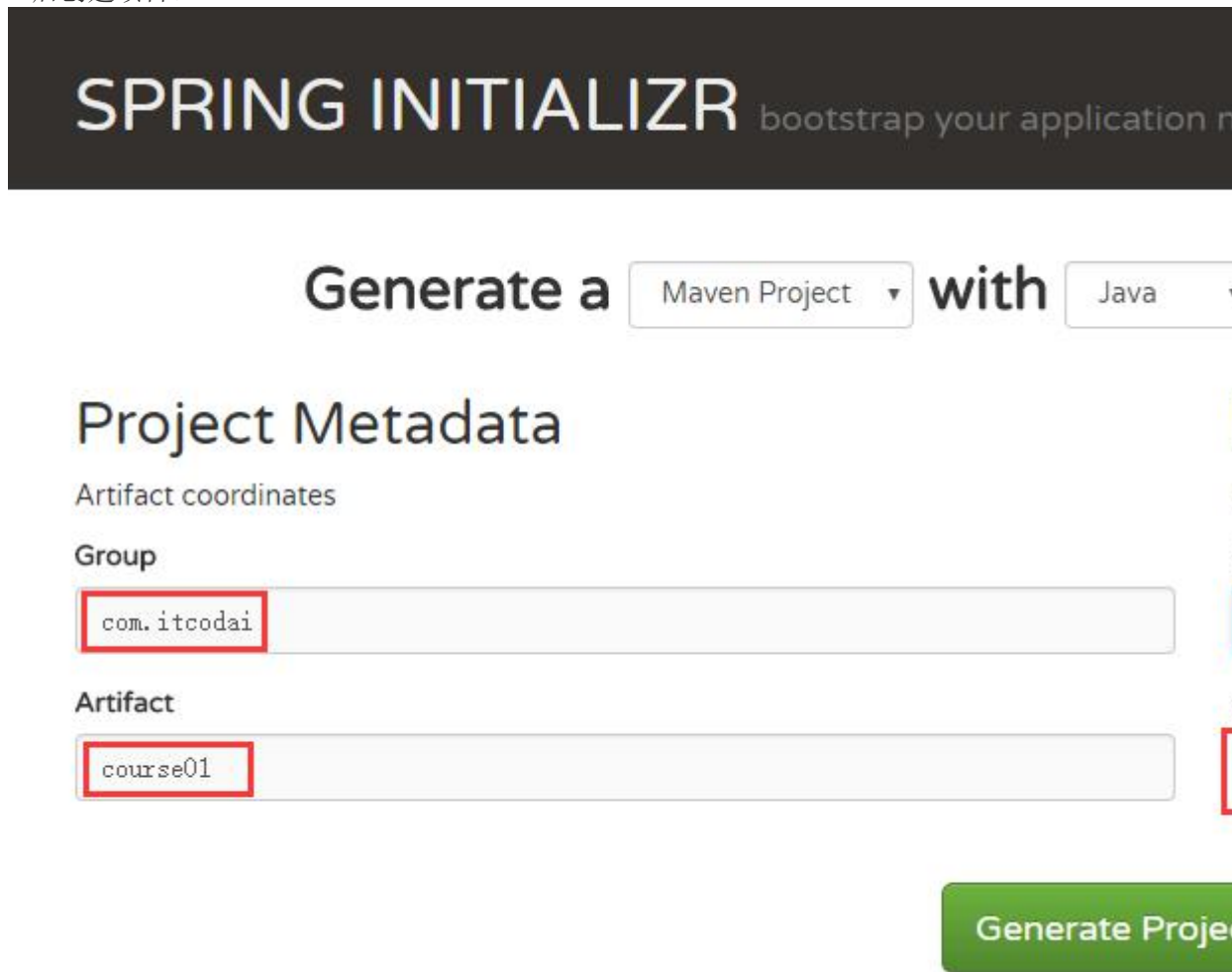
- Group: 填企业域名，本课程使用 com.itcodai
- Artifact: 填项目名称，本课程中每一课的工程名以 course+课号命令，这里使用 course01
- Dependencies: 可以添加我们项目中所需要的依赖信息，根据实际情况来添加，本课程只需要选择 Web 即可。

### 2.2 官方构建



第二种方式可以通过官方构建，步骤如下：

- 访问 <http://start.spring.io/>。
- 在页面上输入相应的 Spring Boot 版本、Group 和 Artifact 信息以及项目依赖，然后创建项目。



SPRING INITIALIZR bootstrap your application

Generate a Maven Project with Java

### Project Metadata

Artifact coordinates

Group

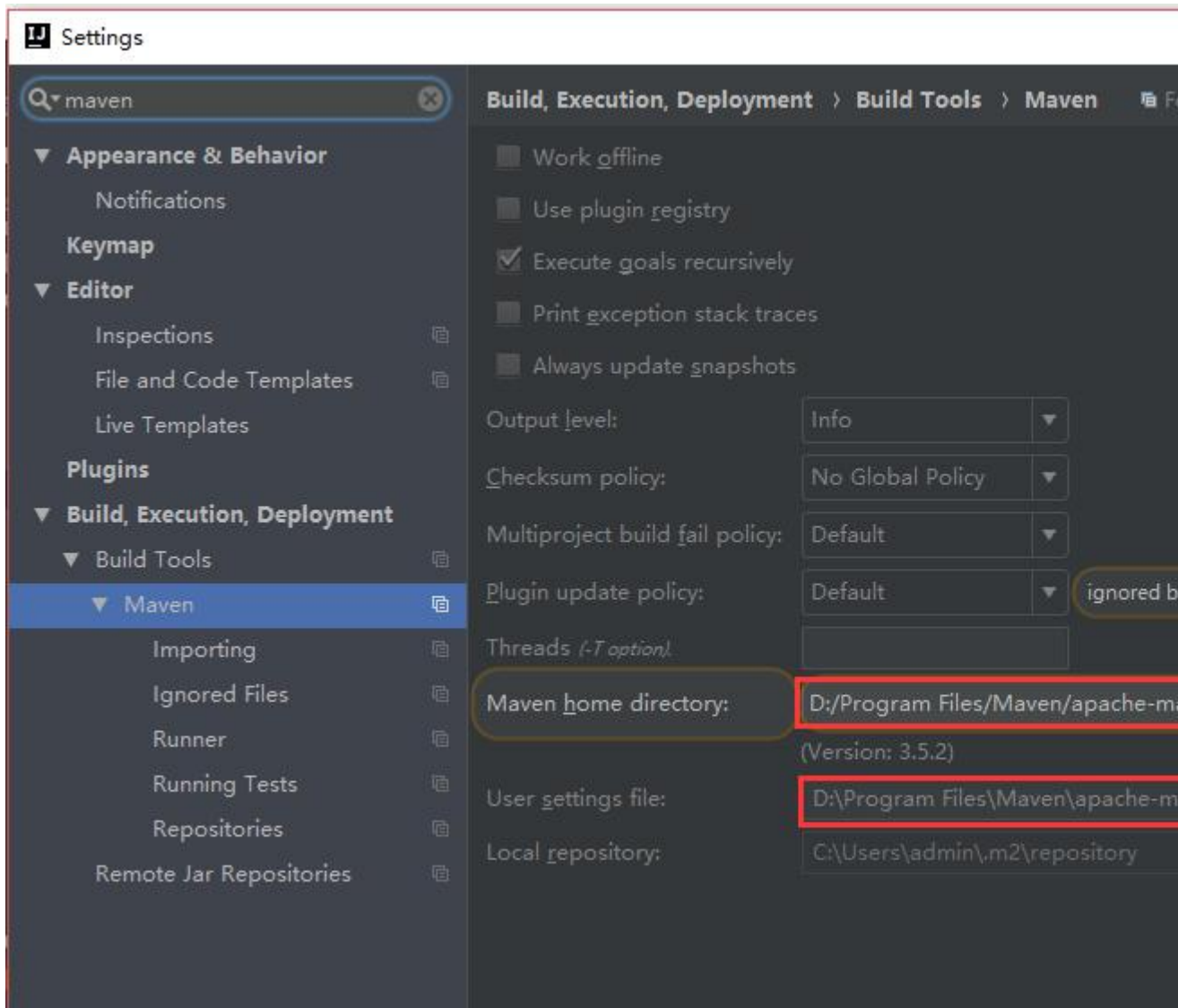
Artifact

Generate Project

- Don't know what to look for? Want more options? [Switch to the full version.](#)
- 解压后，使用 IDEA 导入该 maven 工程：File->New->Model from Existing Source，然后选择解压后的项目文件夹即可。如果是使用 eclipse 的朋友，可以通过 Import->Existing Maven Projects->Next，然后选择解压后的项目文件夹即可。

## 2.3 maven 配置

创建了 Spring Boot 项目之后，需要进行 maven 配置。打开 File->settings，搜索 maven，配置一下本地的 maven 信息。如下：



在 Maven home directory 中选择本地 Maven 的安装路径；在 User settings file 中选择本地 Maven 的配置文件所在路径。在配置文件中，我们配置一下国内阿里的镜像，这样在下载 maven 依赖时，速度很快。

```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorOf>*</mirrorOf>
  <name>Nexus aliyun</name>

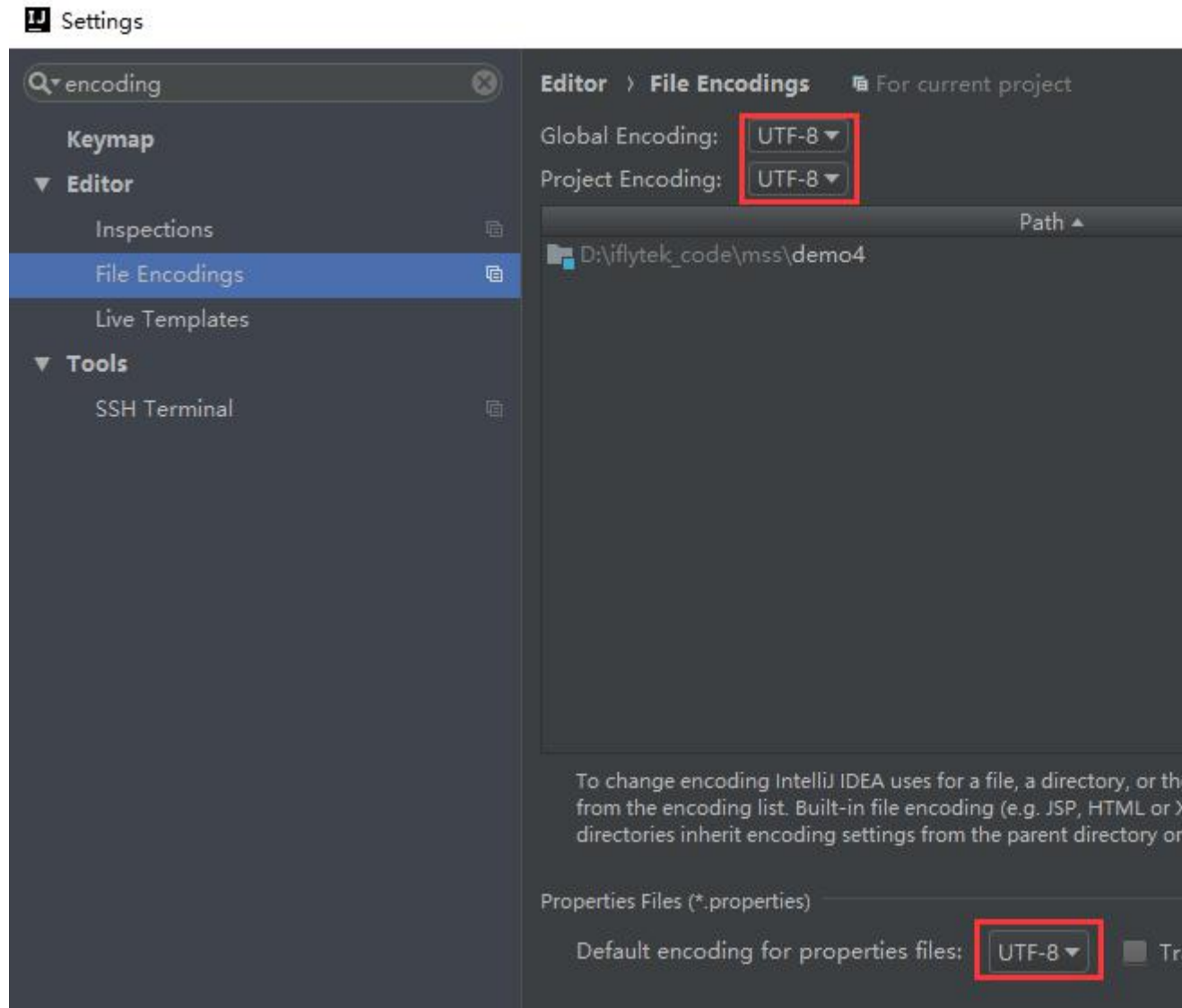
  <url>http://maven.aliyun.com/nexus/content/groups/public</url></mirror>
```

如果是使用 eclipse 的朋友，可以通过 window-->preference-->Maven-->User Settings 来配置，配置方式和上面一致。

## 2.4 编码配置

同样地，新建项目后，我们一般都需要配置编码，这点非常重要，很多初学者都会忘记这一步，所以要养成良好的习惯。

IDEA 中，仍然是打开 File->settings，搜索 encoding，配置一下本地的编码信息。如下：



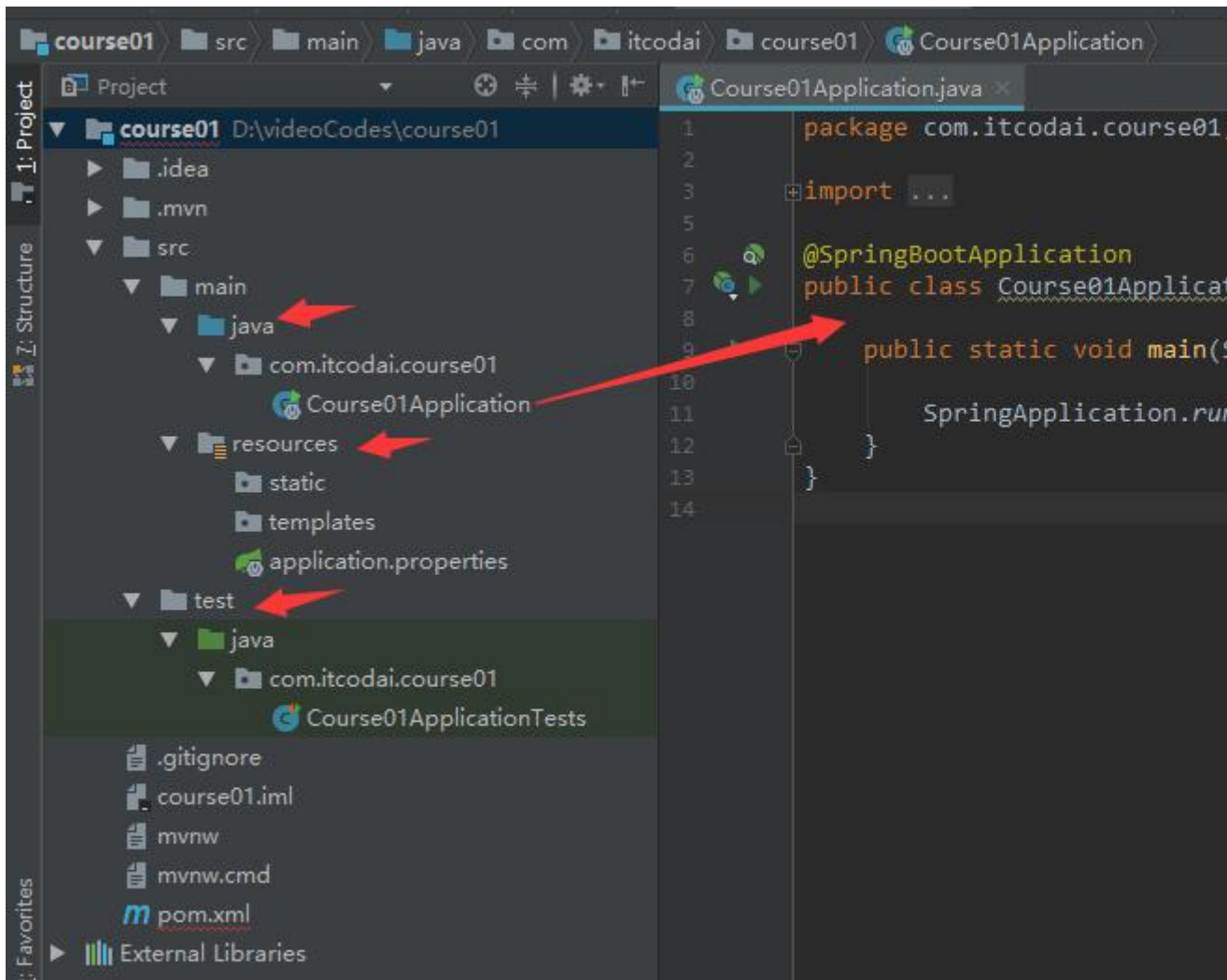
如果是使用 eclipse 的朋友，有两个地方需要设置一下编码：

- window--> preferences-->General-->Workspace，将 Text file encoding 改成 utf-8
- window-->preferences-->General-->content types，选中 Text，将 Default encoding 填入 utf-8

OK，编码设置完成即可启动项目工程了。

### 3. Spring Boot 项目工程结构

Spring Boot 项目总共有三个模块，如下图所示：



- src/main/java 路径：主要编写业务程序
- src/main/resources 路径：存放静态文件和配置文件
- src/test/java 路径：主要编写测试程序

默认情况下，如上图所示会创建一个启动类 **Course01Application**，该类上面有个 **@SpringBootApplication** 注解，该启动类中有个 **main** 方法，没错，Spring Boot 启动只要运行该 **main** 方法即可，非常方便。另外，Spring Boot 内部集成了 **tomcat**，不需要我们人为手动去配置 **tomcat**，开发者只需要关注具体的业务逻辑即可。

到此为止，Spring Boot 就启动成功了，为了比较清楚的看到效果，我们写一个 **Controller** 来测试一下，如下：

```
package com.itcodai.course01.controller;
import org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RestController;
```

```
@RestController@RequestMapping("/start")public class StartController {

    @RequestMapping("/springboot")
    public String startSpringBoot() {
        return "Welcome to the world of Spring Boot!";
    }
}
```

重新运行 main 方法启动项目，在浏览器中输入 localhost:8080/start/springboot，如果看到 “Welcome to the world of Spring Boot!”，那么恭喜你项目启动成功！Spring Boot 就是这么简单方便！端口号默认是 8080，如果想要修改，可以在 application.yml 文件中使用 server.port 来人为指定端口，如 8001 端口：

```
server:
  port: 8001
```

## 4. 总结

本节我们快速学习了如何在 IDEA 中导入 jdk，以及使用 IDEA 如何配置 maven 和编码，如何快速的创建和启动 Spring Boot 工程。IDEA 对 Spring Boot 的支持非常好，建议大家使用 IDEA 进行 Spring Boot 的开发，从下一课开始，我们真正进入 Spring Boot 的学习中。课程源代码下载地址：[戳我下载](#)

## 第 02 课：Spring Boot 返回 Json 数据及数据封装

在项目开发中，接口与接口之间，前后端之间数据的传输都使用 Json 格式，在 Spring Boot 中，接口返回 Json 格式的数据很简单，在 Controller 中使用 @RestController 注解即可返回 Json 格式的数据，@RestController 也是 Spring Boot 新增的一个注解，我们点进去看一下该注解都包含了哪些东西。

```
@Target({ElementType.TYPE})@Retention(RetentionPolicy.RUNTIME)@Documented@Controller@ResponseBodypublic @interface RestController {

    String value() default "";
}
```

可以看出，@RestController 注解包含了原来的 @Controller 和 @ResponseBody 注解，使用过 Spring 的朋友对 @Controller 注解已经非常了解了，这里不再赘述，@ResponseBody 注解是将返回的数据结构转换为 Json 格式。所以在默认情况下，使用

了 @RestController 注解即可将返回的数据结构转换成 Json 格式，Spring Boot 中默认使用的 Json 解析技术框架是 jackson。我们点开 pom.xml 中的 spring-boot-starter-web 依赖，可以看到一个 spring-boot-starter-json 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
  <version>2.0.3.RELEASE</version>
  <scope>compile</scope></dependency>
```

Spring Boot 中对依赖都做了很好的封装，可以看到很多 spring-boot-starter-xxx 系列的依赖，这是 Spring Boot 的特点之一，不需要人为去引入很多相关的依赖了，starter-xxx 系列直接都包含了所必要的依赖，所以我们再次点进去上面这个 spring-boot-starter-json 依赖，可以看到：

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.6</version>
  <scope>compile</scope></dependency><dependency>
  <groupId>com.fasterxml.jackson.datatype</groupId>
  <artifactId>jackson-datatype-jdk8</artifactId>
  <version>2.9.6</version>
  <scope>compile</scope></dependency><dependency>
  <groupId>com.fasterxml.jackson.datatype</groupId>
  <artifactId>jackson-datatype-jsr310</artifactId>
  <version>2.9.6</version>
  <scope>compile</scope></dependency><dependency>
  <groupId>com.fasterxml.jackson.module</groupId>
  <artifactId>jackson-module-parameter-names</artifactId>
  <version>2.9.6</version>
  <scope>compile</scope></dependency>
```

到此为止，我们知道了 Spring Boot 中默认使用的 json 解析框架是 jackson。下面我们看一下默认的 jackson 框架对常用数据类型的转 Json 处理。

## 1. Spring Boot 默认对 Json 的处理

在实际项目中，常用的数据结构无非有类对象、List 对象、Map 对象，我们看一下默认的 jackson 框架对这三个常用的数据结构转成 json 后的格式如何。

## 1.1 创建 User 实体类

为了测试，我们需要创建一个实体类，这里我们就用 User 来演示。

```
public class User {  
    private Long id;  
    private String username;  
    private String password;  
    /* 省略 get、set 和带参构造方法 */  
}
```

## 1.2 创建 Controller 类

然后我们创建一个 Controller，分别返回 User 对象、List<User> 和 Map<String, Object>。

```
import com.itcodai.course02.entity.User;import  
org.springframework.web.bind.annotation.RequestMapping;import  
org.springframework.web.bind.annotation.RestController;import  
java.util.ArrayList;import java.util.HashMap;import  
java.util.List;import java.util.Map;  
  
@RestController  
@RequestMapping("/json")  
public class JsonController {  
  
    @RequestMapping("/user")  
    public User getUser() {  
        return new User(1, "倪升武", "123456");  
    }  
  
    @RequestMapping("/list")  
    public List<User> getUserList() {  
        List<User> userList = new ArrayList<>();  
        User user1 = new User(1, "倪升武", "123456");  
        User user2 = new User(2, "达人课", "123456");  
        userList.add(user1);  
        userList.add(user2);  
        return userList;  
    }  
  
    @RequestMapping("/map")
```



```

public Map<String, Object> getMap() {
    Map<String, Object> map = new HashMap<>(3);
    User user = new User(1, "倪升武", "123456");
    map.put("作者信息", user);
    map.put("博客地址", "http://blog.itcodai.com");
    map.put("CSDN 地址", "http://blog.csdn.net/eson_15");
    map.put("粉丝数量", 4153);
    return map;
}
}

```

### 1.3 测试不同数据类型返回的 json

OK, 写好了接口, 分别返回了一个 User 对象、一个 List 集合和一个 Map 集合, 其中 Map 集合中的 value 存的是不同的数据类型。接下来我们依次来测试一下效果。

在浏览器中输入: localhost:8080/json/user 返回 json 如下:

```

{"id":1,"username":"倪升武","password":"123456"}

```

在浏览器中输入: localhost:8080/json/list 返回 json 如下:

```

[{"id":1,"username":"倪升武",
"password":"123456"}, {"id":2,"username":"达人课",
"password":"123456"}]

```

在浏览器中输入: localhost:8080/json/map 返回 json 如下:

```

{"作者信息":{"id":1,"username":"倪升武","password":"123456"},"CSDN 地址":
"http://blog.csdn.net/eson_15","粉丝数量":4153,"博客地址":
"http://blog.itcodai.com"}

```

可以看出, map 中不管是什么数据类型, 都可以转成相应的 json 格式, 这样就非常方便。

### 1.4 jackson 中对 null 的处理

在实际项目中, 我们难免会遇到一些 null 值出现, 我们转 json 时, 是不希望有这些 null 出现的, 比如我们期望所有的 null 在转 json 时都变成 "" 这种空字符串, 那怎么做呢? 在 Spring Boot 中, 我们做一下配置即可, 新建一个 jackson 的配置类:

```

import com.fasterxml.jackson.core.JsonGenerator;import
com.fasterxml.jackson.databind.JsonSerializer;import
com.fasterxml.jackson.databind.ObjectMapper;import
com.fasterxml.jackson.databind.SerializerProvider;import
org.springframework.boot.autoconfigure.condition.ConditionalOnMissing
Bean;import org.springframework.context.annotation.Bean;import
org.springframework.context.annotation.Configuration;import
org.springframework.context.annotation.Primary;import
org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
import java.io.IOException;

@Configuration
public class JacksonConfig {
    @Bean
    @Primary
    @ConditionalOnMissingBean(ObjectMapper.class)
    public ObjectMapper
jacksonObjectMapper(Jackson2ObjectMapperBuilder builder) {
        ObjectMapper objectMapper =
builder.createXmlMapper(false).build();

objectMapper.getSerializerProvider().setNullValueSerializer(new
JsonSerializer<Object>() {
    @Override
    public void serialize(Object o, JsonGenerator
jsonGenerator, SerializerProvider serializerProvider) throws
IOException {
        jsonGenerator.writeString("");
    }
});
    return objectMapper;
}
}

```

然后我们修改一下上面返回 map 的接口，将几个值改成 null 测试一下：

```

@RequestMapping("/map")
public Map<String, Object> getMap() {
    Map<String, Object> map = new HashMap<>(3);
    User user = new User(1, "倪升武", null);
    map.put("作者信息", user);
    map.put("博客地址", "http://blog.itcodai.com");
    map.put("CSDN 地址", null);
}

```

```
map.put("粉丝数量", 4153);  
return map;  
}
```

重启项目，再次输入：localhost:8080/json/map，可以看到 jackson 已经将所有 null 字段转成了空字符串了。

```
{"作者信息":{"id":1,"username":"倪升武","password":""},"CSDN 地址":"","粉丝数量":4153,"博客地址":"http://blog.itcodai.com"}
```

## 2. 使用阿里巴巴 FastJson 的设置

### 2.1 jackson 和 fastJson 的对比

有很多朋友习惯于使用阿里巴巴的 fastJson 来做项目中 json 转换的相关工作，目前我们项目中使用的就是阿里的 fastJson，那么 jackson 和 fastJson 有哪些区别呢？根据网上公开的资料比较得到下表。

选项	fastJson	jackson
上手难易程度	容易	中等
高级特性支持	中等	丰富
官方文档、Example 支持	中文	英文
处理 json 速度	略快	快

关于 fastJson 和 jackson 的对比，网上有很多资料可以查看，主要是根据自己实际项目情况来选择合适的框架。从扩展上来看，fastJson 没有 jackson 灵活，从速度或者上手难度来看，fastJson 可以考虑，我们项目中目前使用的是阿里的 fastJson，挺方便的。

### 2.2 fastJson 依赖导入

使用 fastJson 需要导入依赖，本课程使用 1.2.35 版本，依赖如下：

```
<dependency>  
    <groupId>com.alibaba</groupId>  
    <artifactId>fastjson</artifactId>  
    <version>1.2.35</version></dependency>
```

### 2.2 使用 fastJson 处理 null

使用 fastJson 时，对 null 的处理和 jackson 有些不同，需要继承 WebMvcConfigurationSupport 类，然后覆盖 configureMessageConverters 方法，在方法中，我们可以选择对要实现 null 转换的场景，配置好即可。如下：

```
import com.alibaba.fastjson.serializer.SerializerFeature;import  
com.alibaba.fastjson.support.config.FastJsonConfig;import
```

```

com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter;import
ort org.springframework.context.annotation.Configuration;import
org.springframework.http.MediaType;import
org.springframework.http.converter.HttpMessageConverter;import
org.springframework.web.servlet.config.annotation.WebMvcConfiguration
Support;
import java.nio.charset.Charset;import java.util.ArrayList;import
java.util.List;

@Configuration
public class FastJsonConfig extends WebMvcConfigurationSupport {

    /**
     * 使用阿里 FastJson 作为 JSON MessageConverter
     * @param converters
     */
    @Override
    public void
configureMessageConverters(List<HttpMessageConverter<?>> converters)
{
        FastJsonHttpMessageConverter converter = new
FastJsonHttpMessageConverter();
        FastJsonConfig config = new FastJsonConfig();
        config.setSerializerFeatures(
            // 保留 map 空的字段
            SerializerFeature.WriteMapNullValue,
            // 将 String 类型的 null 转成""
            SerializerFeature.WriteNullStringAsEmpty,
            // 将 Number 类型的 null 转成 0
            SerializerFeature.WriteNullNumberAsZero,
            // 将 List 类型的 null 转成 []
            SerializerFeature.WriteNullListAsEmpty,
            // 将 Boolean 类型的 null 转成 false
            SerializerFeature.WriteNullBooleanAsFalse,
            // 避免循环引用
            SerializerFeature.DisableCircularReferenceDetect);

        converter.setFastJsonConfig(config);
        converter.setDefaultCharset(Charset.forName("UTF-8"));
        List<MediaType> mediaTypeList = new ArrayList<>();
        // 解决中文乱码问题，相当于在 Controller 上的 @RequestMapping 中加
        了个属性 produces = "application/json"
        mediaTypeList.add(MediaType.APPLICATION_JSON);
        converter.setSupportedMediaTypes(mediaTypeList);
        converters.add(converter);
    }
}

```

```
}  
}
```

### 3. 封装统一返回的数据结构

以上是 Spring Boot 返回 json 的几个代表的例子，但是在实际项目中，除了要封装数据之外，我们往往需要在返回的 json 中添加一些其他信息，比如返回一些状态码 code，返回一些 msg 给调用者，这样调用者可以根据 code 或者 msg 做一些逻辑判断。所以在实际项目中，我们需要封装一个统一的 json 返回结构存储返回信息。

#### 3.1 定义统一的 json 结构

由于封装的 json 数据的类型不确定，所以在定义统一的 json 结构时，我们需要用到泛型。统一的 json 结构中属性包括数据、状态码、提示信息即可，构造方法可以根据实际业务需求做相应的添加即可，一般来说，应该有默认的返回结构，也应该有用户指定的返回结构。如下：

```
public class JsonResult<T> {  
  
    private T data;  
    private String code;  
    private String msg;  
  
    /**  
     * 若没有数据返回，默认状态码为 0，提示信息为：操作成功！  
     */  
    public JsonResult() {  
        this.code = "0";  
        this.msg = "操作成功！";  
    }  
  
    /**  
     * 若没有数据返回，可以人为指定状态码和提示信息  
     * @param code  
     * @param msg  
     */  
    public JsonResult(String code, String msg) {  
        this.code = code;  
        this.msg = msg;  
    }  
  
    /**
```

```

    * 有数据返回时，状态码为 0，默认提示信息为：操作成功！
    * @param data
    */
    public JsonResult(T data) {
        this.data = data;
        this.code = "0";
        this.msg = "操作成功！";
    }

    /**
    * 有数据返回，状态码为 0，人为指定提示信息
    * @param data
    * @param msg
    */
    public JsonResult(T data, String msg) {
        this.data = data;
        this.code = "0";
        this.msg = msg;
    }
    // 省略 get 和 set 方法
}

```

### 3.2 修改 Controller 中的返回值类型及测试

由于 JsonResult 使用了泛型，所以所有的返回值类型都可以使用该统一结构，在具体的场景将泛型替换成具体的数据类型即可，非常方便，也便于维护。在实际项目中，还可以继续封装，比如状态码和提示信息可以定义一个枚举类型，以后我们只需要维护这个枚举类型中的数据即可（在本课程中就不展开了）。根据以上的 JsonResult，我们改写一下 Controller，如下：

```

@RestController@RequestMapping("/jsonresult") public class
JsonResultController {

    @RequestMapping("/user")
    public JsonResult<User> getUser() {
        User user = new User(1, "倪升武", "123456");
        return new JsonResult<>(user);
    }

    @RequestMapping("/list")
    public JsonResult<List> getUserList() {

```

```

        List<User> userList = new ArrayList<>();
        User user1 = new User(1, "倪升武", "123456");
        User user2 = new User(2, "达人课", "123456");
        userList.add(user1);
        userList.add(user2);
        return new JsonResult<>(userList, "获取用户列表成功");
    }

    @RequestMapping("/map")
    public JsonResult<Map> getMap() {
        Map<String, Object> map = new HashMap<>(3);
        User user = new User(1, "倪升武", null);
        map.put("作者信息", user);
        map.put("博客地址", "http://blog.itcodai.com");
        map.put("CSDN 地址", null);
        map.put("粉丝数量", 4153);
        return new JsonResult<>(map);
    }
}

```

我们重新在浏览器中输入：localhost:8080/jsonresult/user 返回 json 如下：

```

{"code": "0", "data": {"id": 1, "password": "123456", "username": "倪升武"}, "msg": "操作成功! "}

```

输入：localhost:8080/jsonresult/list，返回 json 如下：

```

{"code": "0", "data": [{"id": 1, "password": "123456", "username": "倪升武"}, {"id": 2, "password": "123456", "username": "达人课"}], "msg": "获取用户列表成功"}

```

输入：localhost:8080/jsonresult/map，返回 json 如下：

```

{"code": "0", "data": {"作者信息": {"id": 1, "password": "", "username": "倪升武"}, "CSDN 地址": null, "粉丝数量": 4153, "博客地址": "http://blog.itcodai.com"}, "msg": "操作成功! "}

```

通过封装，我们不但将数据通过 json 传给前端或者其他接口，还带上了状态码和提示信息，这在实际项目场景中应用非常广泛。



## 4. 总结

本节主要对 Spring Boot 中 json 数据的返回做了详细的分析，从 Spring Boot 默认的 jackson 框架到阿里巴巴的 fastJson 框架，分别对它们的配置做了相应的讲解。另外，结合实际项目情况，总结了实际项目中使用的 json 封装结构体，加入了状态码和提示信息，使得返回的 json 数据信息更加完整。课程源代码下载地址：[戳我下载](#)

# 第 03 课：Spring Boot 使用 slf4j 进行日志记录

在开发中，我们经常使用 `System.out.println()` 来打印一些信息，但是这样不好，因为大量的使用 `System.out` 会增加资源的消耗。我们实际项目中使用的是 `slf4j` 的 `logback` 来输出日志，效率挺高的，Spring Boot 提供了一套日志系统，`logback` 是最优的选择。

## 1. slf4j 介绍

引用百度百科里的一段话：

SLF4J，即简单日志门面（Simple Logging Facade for Java），不是具体的日志解决方案，它只服务于各种各样的日志系统。按照官方的说法，SLF4J 是一个用于日志系统的简单 Facade，允许最终用户在部署其应用时使用其所希望的日志系统。

这段的大概意思是：你只需要按统一的方式写记录日志的代码，而无需关心日志是通过哪个日志系统，以什么风格输出的。因为它们取决于部署项目时绑定的日志系统。例如，在项目中使用了 `slf4j` 记录日志，并且绑定了 `log4j`（即导入相应的依赖），则日志会以 `log4j` 的风格输出；后期需要改为以 `logback` 的风格输出日志，只需要将 `log4j` 替换成 `logback` 即可，不用修改项目中的代码。这对于第三方组件的引入的不同日志系统来说几乎零学习成本，况且它的优点不仅仅这一个而已，还有简洁的占位符的使用和日志级别的判断。

正因为 `slf4j` 有如此多的优点，阿里巴巴已经将 `slf4j` 作为他们的日志框架了。在《阿里巴巴 Java 开发手册(正式版)》中，日志规约一项第一条就强制要求使用 `slf4j`：

1. **【强制】** 应用中不可直接使用日志系统（Log4j、Logback）中的 API，而应依赖使用日志框架 SLF4J 中的 API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

“强制”两个字体现出了 `slf4j` 的优势，所以建议在实际项目中，使用 `slf4j` 作为自己的日志框架。使用 `slf4j` 记录日志非常简单，直接使用 `LoggerFactory` 创建即可。

```
import org.slf4j.Logger;import org.slf4j.LoggerFactory;
public class Test {
    private static final Logger logger =
    LoggerFactory.getLogger(Test.class);
    // .....
}
```

## 2. application.yml 中对日志的配置

Spring Boot 对 slf4j 支持的很好，内部已经集成了 slf4j，一般我们在使用的时候，会对 slf4j 做一下配置。application.yml 文件是 Spring Boot 中唯一一个需要配置的文件，一开始创建工程的时候是 application.properties 文件，个人比较细化用 yml 文件，因为 yml 文件的层次感特别好，看起来更直观，但是 yml 文件对格式要求比较高，比如英文冒号后面必须要有个空格，否则项目估计无法启动，而且也不报错。用 properties 还是 yml 视个人习惯而定，都可以。本课程使用 yml。

我们看一下 application.yml 文件中对日志的配置：

```
logging:
  config: logback.xml
  level:
    com.itcodai.course03.dao: trace
```

logging.config 是用来指定项目启动的时候，读取哪个配置文件，这里指定的是日志配置文件是根路径下的 logback.xml 文件，关于日志的相关配置信息，都放在 logback.xml 文件中了。logging.level 是用来指定具体的 mapper 中日志的输出级别，上面的配置表示 com.itcodai.course03.dao 包下的所有 mapper 日志输出级别为 trace，会将操作数据库的 sql 打印出来，开发时设置成 trace 方便定位问题，在生产环境上，将这个日志级别再设置成 error 级别即可（本节课不讨论 mapper 层，在后面 Spring Boot 集成 MyBatis 时再详细讨论）。

常用的日志级别按照从高到低依次为：ERROR、WARN、INFO、DEBUG。

## 3. logback.xml 配置文件解析

在上面 application.yml 文件中，我们指定了日志配置文件 logback.xml，logback.xml 文件中主要用来做日志的相关配置。在 logback.xml 中，我们可以定义日志输出的格式、路径、控制台输出格式、文件大小、保存时长等等。下面来分析一下：

### 3.1 定义日志输出格式和存储路径

```
<configuration>
```

```

    <property name="LOG_PATTERN" value="%date{HH:mm:ss.SSS}
[%thread] %-5level %logger{36} - %msg%n" />
    <property name="FILE_PATH"
value="D:/logs/course03/demo.%d{yyyy-MM-dd}.%i.log" />
</configuration>

```

我们来看一下这个定义的含义：首先定义一个格式，命名为“LOG\_PATTERN”，该格式中 %date 表示日期，%thread 表示线程名，%-5level 表示级别从左显示 5 个字符宽度，%logger{36} 表示 logger 名字最长 36 个字符，%msg 表示日志消息，%n 是换行符。

然后再定义一下名为“FILE\_PATH”文件路径，日志都会存储在该路径下。%i 表示第 i 个文件，当日志文件达到指定大小时，会将日志生成到新的文件里，这里的 i 就是文件索引，日志文件允许的大小可以设置，下面会讲解。这里需要注意的是，不管是 windows 系统还是 Linux 系统，日志存储的路径必须要是绝对路径。

### 3.2 定义控制台输出

```

<configuration>
    <appender name="CONSOLE"
class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <!-- 按照上面配置的 LOG_PATTERN 来打印日志 -->
            <pattern>${LOG_PATTERN}</pattern>
        </encoder>
    </appender></configuration>

```

使用 <appender> 节点设置个控制台输出（class="ch.qos.logback.core.ConsoleAppender"）的配置，定义为“CONSOLE”。使用上面定义好的输出格式（LOG\_PATTERN）来输出，使用 \${} 引用进来即可。

### 3.3 定义日志文件的相关参数

```

<configuration>
    <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- 按照上面配置的 FILE_PATH 路径来保存日志 -->

```

```

        <fileNamePattern>${FILE_PATH}</fileNamePattern>
        <!-- 日志保存 15 天 -->
        <maxHistory>15</maxHistory>
        <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
            <!-- 单个日志文件的最大，超过则新建日志文件存储 -->
            <maxFileSize>10MB</maxFileSize>
        </timeBasedFileNamingAndTriggeringPolicy>
    </rollingPolicy>

    <encoder>
        <!-- 按照上面配置的 LOG_PATTERN 来打印日志 -->
        <pattern>${LOG_PATTERN}</pattern>
    </encoder>
</appender></configuration>

```

使用 `<appender>` 定义一个名为“FILE”的文件配置，主要是配置日志文件保存的时间、单个日志文件存储的大小、以及文件保存的路径和日志的输出格式。

### 3.4 定义日志输出级别

```

<configuration>
    <logger name="com.itcodai.course03" level="INFO" />
    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root></configuration>

```

有了上面那些定义后，最后我们使用 `<logger>` 来定义一下项目中默认的日志输出级别，这里定义级别为 `INFO`，然后针对 `INFO` 级别的日志，使用 `<root>` 引用上面定义好的控制台日志输出和日志文件的参数。这样 `logback.xml` 文件中的配置就设置完了。

## 4. 使用 Logger 在项目中打印日志

在代码中，我们一般使用 `Logger` 对象来打印出一些 `log` 信息，可以指定打印出的日志级别，也支持占位符，很方便。

```

import org.slf4j.Logger;import org.slf4j.LoggerFactory;import
org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RestController;
@RequestMapping("/test")public class TestController {

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

    @RequestMapping("/log")
    public String testLog() {
        logger.debug("====测试日志 debug 级别打印====");
        logger.info("=====测试日志 info 级别打印=====");
        logger.error("=====测试日志 error 级别打印=====");
        logger.warn("=====测试日志 warn 级别打印=====");

        // 可以使用占位符打印出一些参数信息
        String str1 = "blog.itcodai.com";
        String str2 = "blog.csdn.net/eson_15";
        logger.info("=====倪升武的个人博客: {}; 倪升武的 CSDN 博客: {}",
str1, str2);

        return "success";
    }
}

```

启动该项目，在浏览器中输入 `localhost:8080/test/log` 后可以看到控制台的日志记录：

```

=====测试日志 info 级别打印===== =====测试日志 error 级
别打印===== =====测试日志 warn 级别打印===== =====倪升
武的个人博客: blog.itcodai.com; 倪升武的 CSDN 博客:
blog.csdn.net/eson_15

```

因为 INFO 级别比 DEBUG 级别高，所以 debug 这条没有打印出来，如果将 `logback.xml` 中的日志级别设置成 DEBUG，那么四条语句都会打印出来，这个大家自己去测试了。同时可以打开 `D:\logs\course03\` 目录，里面有刚刚项目启动，以后后面生成的所有日志记录。在项目部署后，我们大部分都是通过查看日志文件来定位问题。

## 5. 总结

本节课主要对 `slf4j` 做了一个简单的介绍，并且对 Spring Boot 中如何使用 `slf4j` 输出日志做了详细的说明，着重分析了 `logback.xml` 文件中对日志相关信息的配置，包括日志的不同级别。最后针对这些配置，在代码中使用 `Logger` 打印出一些进行测试。在实

际项目中，这些日志都是排查问题的过程中非常重要的资料。课程源代码下载地址：[戳我下载](#)

## 第 04 课：Spring Boot 中的项目属性配置

我们知道，在项目中，很多时候需要用到一些配置的信息，这些信息可能在测试环境和生产环境下会有不同的配置，后面根据实际业务情况有可能还会做修改，针对这种情况，我们不能将这些配置在代码中写死，最好就是写到配置文件中。比如可以把这些信息写到 `application.yml` 文件中。

### 1. 少量配置信息的情形

举个例子，在微服务架构中，最常见的就是某个服务需要调用其他服务来获取其提供的相关信息，那么在该服务的配置文件中需要配置被调用的服务地址，比如在当前服务里，我们需要调用订单微服务获取订单相关的信息，假设 订单服务的端口号是 8002，那我们可以做如下配置：

```
server:
  port: 8001
# 配置微服务的地址 url:
  # 订单微服务的地址
  orderUrl: http://localhost:8002
```

然后在业务代码中如何获取到这个配置的订单服务地址呢？我们可以使用 `@Value` 注解来解决。在对应的类中加上一个属性，在属性上使用 `@Value` 注解即可获取到配置文件中的配置信息，如下：

```
import org.slf4j.Logger;import org.slf4j.LoggerFactory;import
org.springframework.beans.factory.annotation.Value;import
org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RestController;
@RestController@RequestMapping("/test")public class ConfigController {

    private static final Logger LOGGER =
LoggerFactory.getLogger(ConfigController.class);

    @Value("${url.orderUrl}")
    private String orderUrl;

    @RequestMapping("/config")
    public String testConfig() {
```

```
        LOGGER.info("====获取的订单服务地址为: {}", orderUrl);  
        return "success";  
    }  
}
```

@Value 注解上通过 \${key} 即可获取配置文件中和 key 对应的 value 值。我们启动一下项目，在浏览器中输入 localhost:8080/test/config 请求服务后，可以看到控制台会打印出订单服务的地址：

```
====获取的订单服务地址为: http://localhost:8002
```

说明我们成功获取到了配置文件中的订单微服务地址，在实际项目中也是这么用的，后面如果因为服务器部署的原因，需要修改某个服务的地址，那么只要在配置文件中修改即可。

## 2. 多个配置信息的情形

这里再引申一个问题，随着业务复杂度的增加，一个项目中可能会有越来越多的微服务，某个模块可能需要调用多个微服务获取不同的信息，那么就需要在配置文件中配置多个微服务的地址。可是，在需要调用这些微服务的代码中，如果这样一个一个去使用 @Value 注解引入相应的微服务地址的话，太过于繁琐，也不科学。

所以，在实际项目中，业务繁琐，逻辑复杂的情况下，需要考虑封装一个或多个配置类。举个例子：假如在当前服务中，某个业务需要同时调用订单微服务、用户微服务和购物车微服务，分别获取订单、用户和购物车相关信息，然后对这些信息做一定的逻辑处理。那么在配置文件中，我们需要将这些微服务的地址都配置好：

```
# 配置多个微服务的地址  
url:  
    # 订单微服务的地址  
    orderUrl: http://localhost:8002  
    # 用户微服务的地址  
    userUrl: http://localhost:8003  
    # 购物车微服务的地址  
    shoppingUrl: http://localhost:8004
```

也许实际业务中，远远不止这三个微服务，甚至十几个都有可能。对于这种情况，我们可以先定义一个 MicroServiceUrl 类来专门保存微服务的 url，如下：

```
@Component@ConfigurationProperties(prefix = "url")public class  
MicroServiceUrl {
```



```

    private String orderUrl;
    private String userUrl;
    private String shoppingUrl;
    // 省去 get 和 set 方法
}

```

细心的朋友应该可以看到，使用 `@ConfigurationProperties` 注解并且使用 `prefix` 来指定一个前缀，然后该类中的属性名就是配置中去掉前缀后的名字，一一对应即可。

即：前缀名 + 属性名就是配置文件中定义的 `key`。同时，该类上面需要加上 `@Component` 注解，把该类作为组件放到 Spring 容器中，让 Spring 去管理，我们使用的时候直接注入即可。

需要注意的是，使用 `@ConfigurationProperties` 注解需要导入它的依赖：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional></dependency>

```

OK，到此为止，我们将配置写好了，接下来写个 Controller 来测试一下。此时，不需要在代码中一个个引入这些微服务的 url 了，直接通过 `@Resource` 注解将刚刚写好配置类注入进来即可使用了，非常方便。如下：

```

@RestController@RequestMapping("/test")public class TestController {

    private static final Logger LOGGER =
    LoggerFactory.getLogger(TestController.class);

    @Resource
    private MicroServiceUrl microServiceUrl;

    @RequestMapping("/config")
    public String testConfig() {
        LOGGER.info("====获取的订单服务地址为: {}",
microServiceUrl.getOrderUrl());
        LOGGER.info("====获取的用户服务地址为: {}",
microServiceUrl.getUserUrl());
        LOGGER.info("====获取的购物车服务地址为: {}",
microServiceUrl.getShoppingUrl());

        return "success";
    }
}

```

```
}
```

再次启动项目，请求一下可以看到，控制台打印出如下信息，说明配置文件生效，同时正确获取配置文件内容：

```
=====获取的订单服务地址为: http://localhost:8002
=====获取的订单服务地址为: http://localhost:8002
=====获取的用户服务地址为: http://localhost:8003
=====获取的购物车服务地址为: http://localhost:8004
```

### 3. 指定项目配置文件

我们知道，在实际项目中，一般有两个环境：开发环境和生产环境。开发环境中的配置和生产环境中的配置往往不同，比如：环境、端口、数据库、相关地址等等。我们不可能在开发环境调试好之后，部署到生产环境后，又要将配置信息全部修改成生产环境上的配置，这样太麻烦，也不科学。

最好的解决方法就是开发环境和生产环境都有一套对用的配置信息，然后当我们在开发时，指定读取开发环境的配置，当我们将项目部署到服务器上之后，再指定去读取生产环境的配置。

我们新建两个配置文件：`application-dev.yml` 和 `application-pro.yml`，分别用来对开发环境和生产环境进行相关配置。这里为了方便，我们分别设置两个访问端口号，开发环境用 8001，生产环境用 8002。

```
# 开发环境配置文件 server:
  port: 8001
# 生产环境配置文件 server:
  port: 8002
```

然后在 `application.yml` 文件中指定读取哪个配置文件即可。比如我们在开发环境下，指定读取 `application-dev.yml` 文件，如下：

```
spring:
  profiles:
    active:
      - dev
```

这样就可以在开发的时候，指定读取 `application-dev.yml` 文件，访问的时候使用 8001 端口，部署到服务器后，只需要将 `application.yml` 中指定的文件改成 `application-pro.yml` 即可，然后使用 8002 端口访问，非常方便。

## 4. 总结

本节课主要讲解了 Spring Boot 中如何在业务代码中读取相关配置，包括单一配置和多个配置项，在微服务中，这种情况非常常见，往往会有很多其他微服务需要调用，所以封装一个配置类来接收这些配置是个很好的处理方式。除此之外，例如数据库相关的连接参数等等，也可以放到一个配置类中，其他遇到类似的场景，都可以这么处理。最后介绍了开发环境和生产环境配置的快速切换方式，省去了项目部署时，诸多配置信息的修改。课程源代码下载地址：[戳我下载](#)

# 第 05 课：Spring Boot 中的 MVC 支持

Spring Boot 的 MVC 支持主要来介绍实际项目中最常用的几个注解，包括 `@RestController`、`@RequestMapping`、`@PathVariable`、`@RequestParam` 以及 `@RequestBody`。主要介绍这几个注解常用的使用方式和特点。

## 1. @RestController

`@RestController` 是 Spring Boot 新增的一个注解，我们看一下该注解都包含了哪些东西。

```
@Target({ElementType.TYPE})@Retention(RetentionPolicy.RUNTIME)@Documented@Controller@ResponseBodypublic @interface RestController {  
    String value() default "";  
}
```

可以看出，`@RestController` 注解包含了原来的 `@Controller` 和 `@ResponseBody` 注解，使用过 Spring 的朋友对 `@Controller` 注解已经非常了解了，这里不再赘述，`@ResponseBody` 注解是将返回的数据结构转换为 Json 格式。所以 `@RestController` 可以看作是 `@Controller` 和 `@ResponseBody` 的结合体，相当于偷个懒，我们使用 `@RestController` 之后就不用再使用 `@Controller` 了。但是需要注意一个问题：如果是前后端分离，不用模板渲染的话，比如 Thymeleaf，这种情况下是可以直接使用 `@RestController` 将数据以 json 格式传给前端，前端拿到之后解析；但如果不是前后端分离，需要使用模板来渲染的话，一般 Controller 中都会返回到具体的页面，那么此时就不能使用 `@RestController` 了，比如：

```
public String getUser() {  
    return "user";  
}
```

其实是需要返回到 user.html 页面的，如果使用 `@RestController` 的话，会将 user 作为字符串返回的，所以这时候我们需要使用 `@Controller` 注解。这在下一节 Spring Boot 集成 Thymeleaf 模板引擎中会再说明。

## 2. @RequestMapping

**@RequestMapping** 是一个用来处理请求地址映射的注解，它可以用于类上，也可以用于方法上。在类的级别上的注解会将一个特定请求或者请求模式映射到一个控制器之上，表示类中的所有响应请求的方法都是以该地址作为父路径；在方法的级别表示进一步指定到处理方法的映射关系。

该注解有 6 个属性，一般在项目中比较常用的有三个属性：**value**、**method** 和 **produces**。

- **value** 属性：指定请求的实际地址，**value** 可以省略不写
- **method** 属性：指定请求的类型，主要有 GET、PUT、POST、DELETE，默认为 GET
- **produces** 属性：指定返回内容类型，如 `produces = "application/json; charset=UTF-8"`

**@RequestMapping** 注解比较简单，举个例子：

```
@RestController@RequestMapping(value = "/test", produces =
"application/json; charset=UTF-8")public class TestController {

    @RequestMapping(value = "/get", method = RequestMethod.GET)
    public String testGet() {
        return "success";
    }
}
```

这个很简单，启动项目在浏览器中输入 `localhost:8080/test/get` 测试一下即可。

针对四种不同的请求方式，是有相应注解的，不用每次在 **@RequestMapping** 注解中加 **method** 属性来指定，上面的 GET 方式请求可以直接使用 **@GetMapping("/get")** 注解，效果一样。相应地，PUT 方式、POST 方式和 DELETE 方式对应的注解分别为 **@PutMapping**、**@PostMapping** 和 **DeleteMapping**。

## 3. @PathVariable

**@PathVariable** 注解主要是用来获取 url 参数，Spring Boot 支持 restfull 风格的 url，比如一个 GET 请求携带一个参数 `id` 过来，我们将 `id` 作为参数接收，可以使用 **@PathVariable** 注解。如下：

```
@GetMapping("/user/{id}")public String testPathVariable(@PathVariable
Integer id) {
    System.out.println("获取到的 id 为: " + id);
    return "success";
}
```

```
}
```

这里需要注意一个问题，如果想要 url 中占位符中的 id 值直接赋值到参数 id 中，需要保证 url 中的参数和方法接收参数一致，否则就无法接收。如果不一致的话，其实也可以解决，需要用 `@PathVariable` 中的 `value` 属性来指定对应关系。如下：

```
@RequestMapping("/user/{idd}") public String  
testPathVariable(@PathVariable(value = "idd") Integer id) {  
    System.out.println("获取到的 id 为: " + id);  
    return "success";  
}
```

对于访问的 url，占位符的位置可以在任何位置，不一定非要在最后，比如这样也行：`/xxx/{id}/user`。另外，url 也支持多个占位符，方法参数使用同样数量的参数来接收，原理和一个参数是一样的，例如：

```
@GetMapping("/user/{idd}/{name}")  
    public String testPathVariable(@PathVariable(value = "idd") Integer id,  
    @PathVariable String name) {  
        System.out.println("获取到的 id 为: " + id);  
        System.out.println("获取到的 name 为: " + name);  
        return "success";  
    }
```

运行项目，在浏览器中请求 `localhost:8080/test/user/2/zhangsan` 可以看到控制台输出如下信息：

```
获取到的 id 为: 2  
获取到的 name 为: zhangsan
```

所以支持多个参数的接收。同样地，如果 url 中的参数和方法中的参数名称不同的话，也需要使用 `value` 属性来绑定两个参数。

## 4. @RequestParam

`@RequestParam` 注解顾名思义，也是获取请求参数的，上面我们介绍了 `@PathVariable` 注解也是获取请求参数的，那么 `@RequestParam` 和 `@PathVariable` 有什么不同呢？主要区别在于：`@PathVariable` 是从 url 模板中获取参数值，即这种风格的 url: `http://localhost:8080/user/{id}`；而 `@RequestParam` 是从 request 里面获取参数值，即这种风格的 url: `http://localhost:8080/user?id=1`。我们使用该 url 带上参数 id 来测试一下如下代码：

```
@GetMapping("/user")public String testRequestParam(@RequestParam Integer id) {  
    System.out.println("获取到的 id 为: " + id);  
    return "success";  
}
```

可以正常从控制台打印出 id 信息。同样地，url 上面的参数和方法的参数需要一致，如果不一致，也需要使用 value 属性来说明，比如 url 为：  
http://localhost:8080/user?idd=1

```
@RequestMapping("/user")public String  
testRequestParam(@RequestParam(value = "idd", required = false) Integer  
id) {  
    System.out.println("获取到的 id 为: " + id);  
    return "success";  
}
```

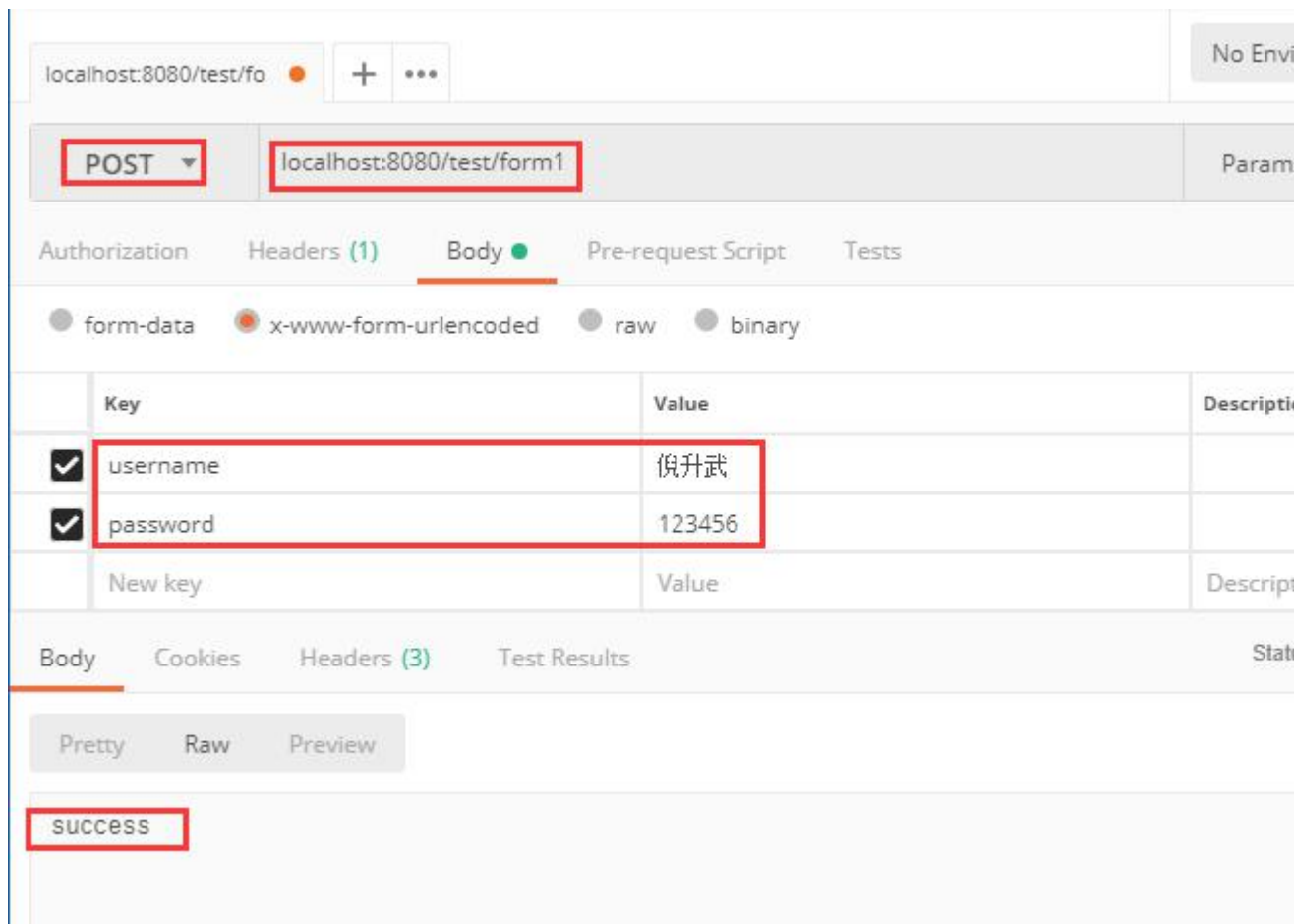
除了 value 属性外，还有个两个属性比较常用：

- required 属性：true 表示该参数必须要传，否则就会报 404 错误，false 表示可有可无。
- defaultValue 属性：默认值，表示如果请求中没有同名参数时的默认值。

从 url 中可以看出，@RequestParam 注解用于 GET 请求上时，接收拼接在 url 中的参数。除此之外，该注解还可以用于 POST 请求，接收前端表单提交的参数，假如前端通过表单提交 username 和 password 两个参数，那我们可以使用 @RequestParam 来接收，用法和上面一样。

```
@PostMapping("/form1")  
public String testForm(@RequestParam String username,  
@RequestParam String password) {  
    System.out.println("获取到的 username 为: " + username);  
    System.out.println("获取到的 password 为: " + password);  
    return "success";  
}
```

我们使用 postman 来模拟一下表单提交，测试一下接口：



那么问题来了，如果表单数据很多，我们不可能在后台方法中写上很多参数，每个参数还要 `@RequestParam` 注解。针对这种情况，我们需要封装一个实体类来接收这些参数，实体中的属性名和表单中的参数名一致即可。

```
public class User {  
    private String username;  
    private String password;  
    // set get  
}
```

使用实体接收的话，我们不能在前面加 `@RequestParam` 注解了，直接使用即可。

```
@PostMapping("/form2")  
public String testForm(User user) {  
    System.out.println("获取到的 username 为: " +  
user.getUsername());  
    System.out.println("获取到的 password 为: " +  
user.getPassword());  
    return "success";  
}
```



```
}
```

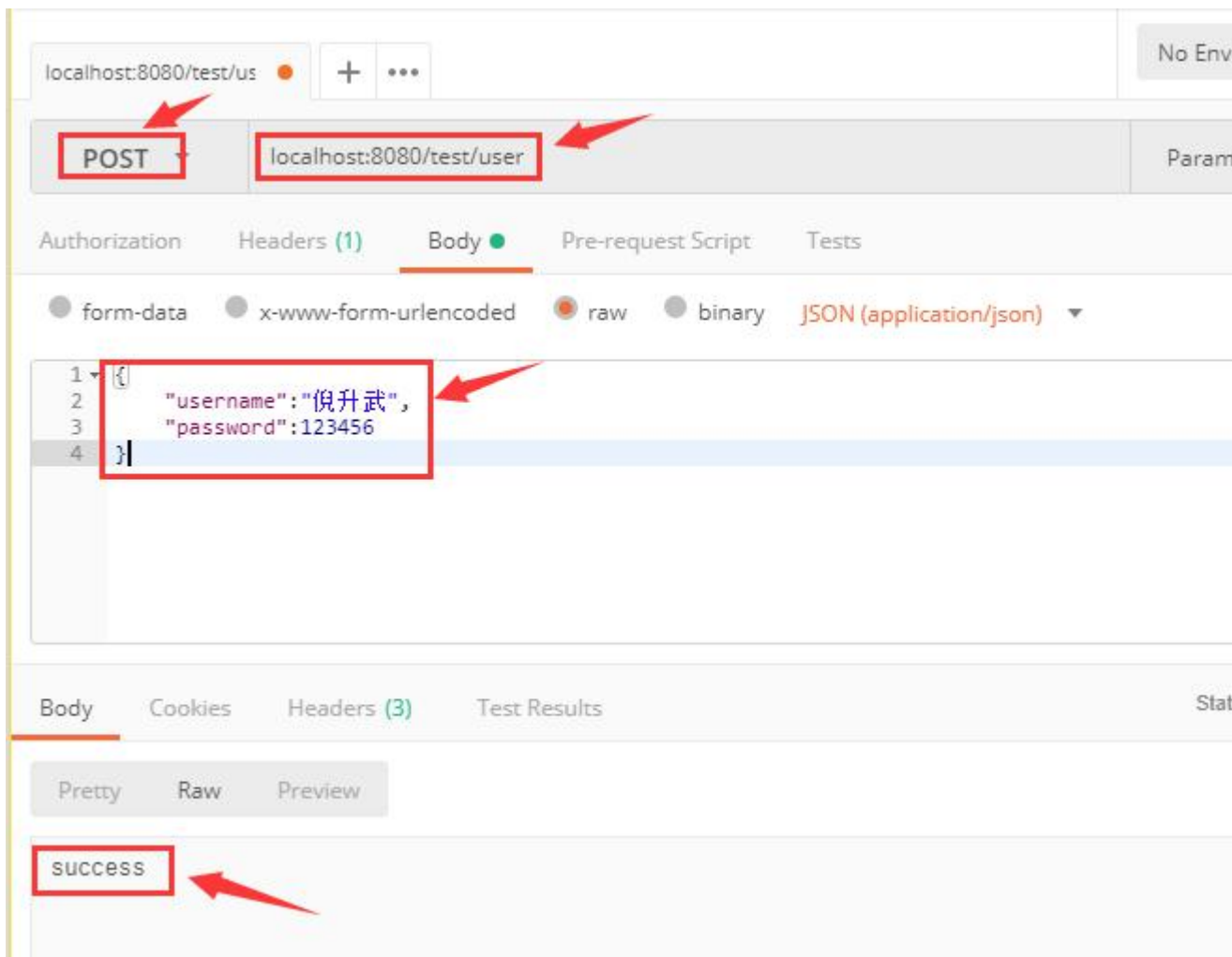
使用 **postman** 再次测试一下表单提交，观察一下返回值和控制台打印出的日志即可。  
在实际项目中，一般都是封装一个实体类来接收表单数据，因为实际项目中表单数据一般都很多。

## 5. @RequestBody

**@RequestBody** 注解用于接收前端传来的实体，接收参数也是对应的实体，比如前端通过 **json** 提交传来两个参数 **username** 和 **password**，此时我们需要在后端封装一个实体来接收。在传递的参数比较多的情况下，使用 **@RequestBody** 接收会非常方便。例如：

```
public class User {  
    private String username;  
    private String password;  
    // set get  
}  
@PostMapping("/user") public String testRequestBody(@RequestBody User user) {  
    System.out.println("获取到的 username 为: " + user.getUsername());  
    System.out.println("获取到的 password 为: " + user.getPassword());  
    return "success";  
}
```

我们使用 **postman** 工具来测试一下效果，打开 **postman**，然后输入请求地址和参数，参数我们用 **json** 来模拟，如下图所有，调用之后返回 **success**。



同时看一下后台控制台输出的日志：

获取到的 username 为：倪升武  
获取到的 password 为：123456

可以看出，`@RequestBody` 注解用于 POST 请求上，接收 json 实体参数。它和上面我们介绍的表单提交有点类似，只不过参数的格式不同，一个是 json 实体，一个是表单提交。在实际项目中根据具体场景和需要使用对应的注解即可。

## 6. 总结

本节课主要讲解了 Spring Boot 中对 MVC 的支持，分析了 `@RestController`、`@RequestMapping`、`@PathVariable`、`@RequestParam` 和 `@RequestBody` 四个注解的使用方式，由于 `@RestController` 中集成了 `@ResponseBody` 所以对返回 json 的注解不再赘述。以上四个注解是使用频率很高的注解，在所有的实际项目中基本都会遇到，要熟练掌握。

课程源代码下载地址：[戳我下载](#)

# 第 06 课：Spring Boot 集成 Swagger2 展 现在线接口文档

## 1. Swagger 简介

### 1.1 解决的问题

随着互联网技术的发展，现在的网站架构基本都由原来的后端渲染，变成了前后端分离的形态，而且前端技术和后端技术在各自的道路上越走越远。前端和后端的唯一联系，变成了 API 接口，所以 API 文档变成了前后端开发人员联系的纽带，变得越来越重要。

那么问题来了，随着代码的不断更新，开发人员在开发新的接口或者更新旧的接口后，由于开发任务的繁重，往往文档很难持续跟着更新，**Swagger** 就是用来解决该问题的一款重要的工具，对使用接口的人来说，开发人员不需要给他们提供文档，只要告诉他们一个 **Swagger** 地址，即可展示在线的 API 接口文档，除此之外，调用接口的人员还可以在线测试接口数据，同样地，开发人员在开发接口时，同样也可以利用 **Swagger** 在线接口文档测试接口数据，这给开发人员提供了便利。

### 1.2 Swagger 官方

我们打开 [Swagger 官网](#)，官方对 **Swagger** 的定义为：

The Best APIs are Built with Swagger Tools

翻译成中文是：“最好的 API 是使用 **Swagger** 工具构建的”。由此可见，**Swagger** 官方对其功能和所处的地位非常自信，由于其非常好用，所以官方对其定位也合情合理。如下图所示：

## The Best APIs are Built with



### Design

Design and model APIs according to specification-based standards



### Build

Build stable, reusable code for your API in almost any language



### Document

Improve developer experience with interactive API documentation

本文主要讲解在 **Spring Boot** 中如何导入 **Swagger2** 工具来展现项目中的接口文档。本节课使用的 **Swagger** 版本为 **2.2.2**。下面开始进入 **Swagger2** 之旅。

## 2. Swagger2 的 maven 依赖

使用 **Swagger2** 工具，必须要导入 **maven** 依赖，当前官方最高版本是 **2.8.0**，我尝试了一下，个人感觉页面展示的效果不太好，而且不够紧凑，不利于操作。另外，最新版本并不一定是最稳定版本，当前我们实际项目中使用的是 **2.2.2** 版本，该版本稳定，界面友好，所以本节课主要围绕着 **2.2.2** 版本来展开，依赖如下：

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.2.2</version></dependency><dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
```

```
<version>2.2.2</version></dependency>
```

### 3. Swagger2 的配置

使用 Swagger2 需要进行配置，Spring Boot 中对 Swagger2 的配置非常方便，新建一个配置类，Swagger2 的配置类上除了添加必要的 `@Configuration` 注解外，还需要添加 `@EnableSwagger2` 注解。

```
import org.springframework.context.annotation.Bean;import
org.springframework.context.annotation.Configuration;import
springfox.documentation.builders.ApiInfoBuilder;import
springfox.documentation.builders.PathSelectors;import
springfox.documentation.builders.RequestHandlerSelectors;import
springfox.documentation.service.ApiInfo;import
springfox.documentation.spi.DocumentationType;import
springfox.documentation.spring.web.plugins.Docket;import
springfox.documentation.swagger2.annotations.EnableSwagger2;
/**
 * @author shengwu ni
 */
@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            // 指定构建 api 文档的详细信息的方法: apiInfo()
            .apiInfo(apiInfo())
            .select()
            // 指定要生成 api 接口的包路径，这里把 controller 作为包路
            径，生成 controller 中的所有接口
            .apis(RequestHandlerSelectors.basePackage("com.itcodai
            course06.controller"))
            .paths(PathSelectors.any())
            .build();
    }

    /**
     * 构建 api 文档的详细信息
     * @return
```

```

*/
private ApiInfo apiInfo() {
    return new ApiInfoBuilder()
        // 设置页面标题
        .title("Spring Boot 集成 Swagger2 接口总览")
        // 设置接口描述
        .description("跟武哥一起学 Spring Boot 第 06 课")
        // 设置联系方式
        .contact("倪升武, " + "CSDN:"
http://blog.csdn.net/eson_15")
        // 设置版本
        .version("1.0")
        // 构建
        .build();
}
}

```

在该配置类中，已经使用注释详细解释了每个方法的作用了，在此不再赘述。到此为止，我们已经配置好了 Swagger2 了。现在我们可以测试一下配置有没有生效，启动项目，在浏览器中输入 `localhost:8080/swagger-ui.html`，即可看到 swagger2 的接口页面，如下图所示，说明 Swagger2 集成成功。



结合该图，对照上面的 Swagger2 配置文件中的配置，可以很明确的知道配置类中每个方法的作用。这样就很容易理解和掌握 Swagger2 中的配置了，也可以看出，其实 Swagger2 配置很简单。

【友情提示】可能有很多朋友在配置 Swagger 的时候会遇到下面的情况，而且还关不掉的，这是因为浏览器缓存引起的，清空一下浏览器缓存即可解决问题。

## 4. Swagger2 的使用

上面我们已经配置好了 Swagger2，并且也启动测试了一下，功能正常，下面我们开始使用 Swagger2，主要来介绍 Swagger2 中的几个常用的注解，分别在实体类上、Controller 类上以及 Controller 中的方法上，最后我们看一下 Swagger2 是如何在页面上呈现在线接口文档的，并且结合 Controller 中的方法在接口中测试一下数据。

### 4.1 实体类注解

本节我们建一个 User 实体类，主要介绍一下 Swagger2 中的 @ApiModelProperty 和 @ApiModelPropertyProperty 注解，同时为后面的测试做准备。

```
import io.swagger.annotations.ApiModel;import
io.swagger.annotations.ApiModelProperty;
@ApiModel(value = "用户实体类")public class User {

    @ApiModelProperty(value = "用户唯一标识")
    private Long id;

    @ApiModelProperty(value = "用户姓名")
    private String username;

    @ApiModelProperty(value = "用户密码")
    private String password;

    // 省略 set 和 get 方法
}
```

解释下 @ApiModelProperty 和 @ApiModelPropertyProperty 注解：

@ApiModelProperty 注解用于实体类，表示对类进行说明，用于参数用实体类接收。 @ApiModelPropertyProperty 注解用于类中属性，表示对 model 属性的说明或者数据操作更改。

该注解在在线 API 文档中的具体效果在下文说明。

### 4.2 Controller 类中相关注解

我们写一个 TestController，再写几个接口，然后学习一下 Controller 中和 Swagger2 相关的注解。

```

import com.itcodai.course06.entiy.JsonResult;import
com.itcodai.course06.entiy.User;import
io.swagger.annotations.Api;import
io.swagger.annotations.ApiOperation;import
io.swagger.annotations.ApiParam;import
org.springframework.web.bind.annotation.GetMapping;import
org.springframework.web.bind.annotation.PathVariable;import
org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/swagger")
@Api(value = "Swagger2 在线接口文档")
public class TestController {

    @GetMapping("/get/{id}")
    @ApiOperation(value = "根据用户唯一标识获取用户信息")
    public JsonResult<User> getUserInfo(@PathVariable @ApiParam(value
= "用户唯一标识") Long id) {
        // 模拟数据库中根据 id 获取 User 信息
        User user = new User(id, "倪升武", "123456");
        return new JsonResult(user);
    }
}

```

我们来学习一下 @Api 、 @ApiOperation 和 @ApiParam 注解。

@Api 注解用于类上，表示标识这个类是 swagger 的资源。  
 @ApiOperation 注解用于方法，表示一个 http 请求的操作。  
 @ApiParam 注解用于参数上，用来标明参数信息。

这里返回的是 JsonResult，是第 02 课中学习返回 json 数据时封装的实体。以上是 Swagger 中最常用的 5 个注解，接下来运行一下项目工程，在浏览器中输入 localhost:8080/swagger-ui.html 看一下 Swagger 页面的接口状态。



# Spring Boot集成Swagger2接口总览

跟武哥一起学Spring Boot第06课

Created by 倪升武, CSDN : [http://blog.csdn.net/eson\\_15](http://blog.csdn.net/eson_15)

## swagger2-在线接口文档 : Swagger2 在线接口文档

GET /swagger/get/{id} @Api注解指定

Response Class (Status 200)

Model Model Schema

```
{
  "code": "string",
  "data": {},
  "msg": "string"
}
```

展示该接口返回的参数结构

展示接口请求方式和url

Response Content Type \*/\* ▼

Parameters

Parameter	Value	Description
id	<input type="text"/>	用户唯一标识

输入接口参数id

Response Messages

HTTP Status Code	Reason	Response Model
401	Unauthorized	

@ApiP

可以看出, Swagger 页面对该接口的信息展示的非常全面, 每个注解的作用以及展示的地方在上图中已经标明, 通过页面即可知道该接口的所有信息, 那么我们直接在线测试一下该接口返回的信息, 输入 id 为 1, 看一下返回数据:

### Request URL

http://localhost:8080/swagger/get/1

### Response Body

```
{
  "data": {
    "id": 1,
    "username": "倪升武",
    "password": "123456"
  },
  "code": "0",
  "msg": "操作成功!"
}
```

### Response Code

200

可以看出，直接在页面返回了 json 格式的数据，开发人员可以直接使用该在线接口来测试数据的正确与否，非常方便。上面是对于单个参数的输入，如果输入参数为某个对象这种情况，Swagger 是什么样子呢？我们再写一个接口。

```
@PostMapping("/insert")
@ApiOperation(value = "添加用户信息")
public JsonResult<Void> insertUser(@RequestBody @ApiParam(value = "用户信息") User user) {
    // 处理添加逻辑
    return new JsonResult<>();
}
```

重启项目，在浏览器中输入 localhost:8080/swagger-ui.html 看一下效果：

POST /swagger/insert

展示接口请求方式和url

Response Class (Status 200)

Model Model Schema

```
{
  "code": "string",
  "msg": "string"
}
```

返回JsonResult默认值即可

Response Content Type \*/\*

Parameters

Parameter	Value	Description	Para
user	<pre>{   "id": 0,   "password": "string",   "username": "string" }</pre>	用户信息	bod

Parameter content type: application/json

点击右侧的Model Schema  
入参的json赋值到左侧，然  
一下值即可

## 5. 总结

OK，本节课详细分析了 Swagger 的优点，以及 Spring Boot 如何集成 Swagger2，包括配置，相关注解的讲解，涉及到了实体类和接口类，以及如何使用。最后通过页面测试，体验了 Swagger 的强大之处，基本上是每个项目组中必备的工具之一，所以要掌握该工具的使用，也不难。

课程源代码下载地址：[戳我下载](#)

## 第 07 课：Spring Boot 集成 Thymeleaf 模板引擎

### 1. Thymeleaf 介绍

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

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

以上翻译自 Thymeleaf 官方网站。传统的 JSP+JSTL 组合是已经过去了，Thymeleaf 是现代服务端的模板引擎，与传统的 JSP 不同，Thymeleaf 可以使用浏览器直接打开，因为可以忽略掉拓展属性，相当于打开原生页面，给前端人员也带来一定的便利。

什么意思呢？就是说在本地环境或者有网络的环境下，Thymeleaf 均可运行。由于 thymeleaf 支持 html 原型，也支持在 html 标签里增加额外的属性来达到“模板+数据”的展示方式，所以美工可以直接在浏览器中查看页面效果，当服务启动后，也可以让后台开发人员查看带数据的动态页面效果。比如：

```
<div class="ui right aligned basic segment">
    <div class="ui orange basic label" th:text="${blog.flag}">静态原创信息
</div></div><h2 class="ui center aligned header" th:text="${blog.title}">这是
静态标题</h2>
```

类似与上面这样，在静态页面时，会展示静态信息，当服务启动后，动态获取数据库中的数据后，就可以展示动态数据，th:text 标签是用来动态替换文本的，这会在下文说明。该例子说明浏览器解释 html 时会忽略 html 中未定义的标签属性（比如 th:text），所以 thymeleaf 的模板可以静态地运行；当有数据返回到页面时，Thymeleaf 标签会动态地替换掉静态内容，使页面动态显示数据。

## 2. 依赖导入

在 Spring Boot 中使用 thymeleaf 模板需要引入依赖，可以在创建项目工程时勾选 Thymeleaf，也可以创建之后再手动导入，如下：

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

另外，在 html 页面上如果要使用 thymeleaf 模板，需要在页面标签中引入：

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

## 3. Thymeleaf 相关配置

因为 Thymeleaf 中已经有默认的配置了，我们不需要再对其做过多的配置，有一个需要注意一下，Thymeleaf 默认是开启页面缓存的，所以在开发的时候，需要关闭这个页面缓存，配置如下。

```
spring:
  thymeleaf:
    cache: false #关闭缓存
```

否则会有缓存，导致页面没法及时看到更新后的效果。比如你修改了一个文件，已经 update 到 tomcat 了，但刷新页面还是之前的页面，就是因为缓存引起的。

## 4. Thymeleaf 的使用

### 4.1 访问静态页面

这个和 Thymeleaf 没啥关系，应该说是通用的，我把它一并写到这里的原因是一般我们做网站的时候，都会做一个 404 页面和 500 页面，为了出错时给用户一个友好的展示，而不至于一堆异常信息抛出来。Spring Boot 中会自动识别模板目录（templates/）下的 404.html 和 500.html 文件。我们在 templates/ 目录下新建一个 error 文件夹，专门放置错误的 html 页面，然后分别打印些信息。以 404.html 为例：

```
<!DOCTYPE html><html lang="en"><head>
  <meta charset="UTF-8">
  <title>Title</title></head><body>
    这是 404 页面</body></html>
```

我们再写一个 controller 来测试一下 404 和 500 页面：

```
@Controller@RequestMapping("/thymeleaf")public class
ThymeleafController {

    @RequestMapping("/test404")
    public String test404() {
        return "index";
    }

    @RequestMapping("/test500")
    public String test500() {
        int i = 1 / 0;
        return "index";
    }
}
```

当我们在浏览器中输入 localhost:8080/thymeleaf/test400 时，故意输入错误，找不到对应的方法，就会跳转到 404.html 显示。

当我们在浏览器中输入 `localhost:8088/thymeleaf/test505` 时，会抛出异常，然后会自动跳转到 `500.html` 显示。

【注】这里有个问题需要注意一下，前面的课程中我们说了微服务中会走向前后端分离，我们在 `Controller` 层上都是使用的 `@RestController` 注解，自动会把返回的数据转成 `json` 格式。但是在使用模板引擎时，`Controller` 层就不能用 `@RestController` 注解了，因为在使用 `thymeleaf` 模板时，返回的是视图文件名，比如上面的 `Controller` 中是返回到 `index.html` 页面，如果使用 `@RestController` 的话，会把 `index` 当作 `String` 解析了，直接返回到页面了，而不是去找 `index.html` 页面，大家可以试一下。所以在使用模板时要用 `@Controller` 注解。

## 4.2 Thymeleaf 中处理对象

我们来看一下 `thymeleaf` 模板中如何处理对象信息，假如我们在做个人博客的时候，需要给前端传博主相关信息来展示，那么我们会封装成一个博主对象，比如：

```
public class Blogger {  
    private Long id;  
    private String name;  
    private String pass;  
    // 省去 set 和 get  
}
```

然后在 `controller` 层中初始化一下：

```
@GetMapping("/getBlogger") public String getBlogger(Model model) {  
    Blogger blogger = new Blogger(1L, "倪升武", "123456");  
    model.addAttribute("blogger", blogger);  
    return "blogger";  
}
```

我们先初始化一个 `Blogger` 对象，然后将该对象放到 `Model` 中，然后返回到 `blogger.html` 页面去渲染。接下来我们再写一个 `blogger.html` 来渲染 `blogger` 信息：

```
<!DOCTYPE html><html xmlns:th="http://www.thymeleaf.org"><html  
lang="en"><head>  
    <meta charset="UTF-8">  
    <title>博主信息</title></head><body><form action=""  
th:object="${blogger}" >  
    用户编号: <input name="id" th:value="${blogger.id}"/><br>  
    用户姓名: <input type="text" name="username"  
th:value="${blogger.getName()}" /><br>
```

```
    登陆密码: <input type="text" name="password" th:value="*{pass}"
/></form></body></html>
```

可以看出，在 thymeleaf 模板中，使用 `th:object="${}"` 来获取对象信息，然后在表单里面可以有三种方式来获取对象属性。如下：

使用 `th:value="*{属性名}"` 使用 `th:value="${对象.属性名}"`，对象指的是上面使用 `th:object` 获取的对象 使用 `th:value="${对象.get 方法}"`，对象指的是上面使用 `th:object` 获取的对象

可以看出，在 Thymeleaf 中可以像写 java 一样写代码，很方便。我们在浏览器中输入 `localhost:8080/thymeleaf/getBlogger` 来测试一下数据：

用户编号：	1
用户姓名：	倪升武
登陆密码：	123456

### 4.3 Thymeleaf 中处理 List

处理 List 的话，和处理上面介绍的对象差不多，但是需要在 thymeleaf 中进行遍历。我们先在 Controller 中模拟一个 List。

```
@GetMapping("/getList") public String getList(Model model) {
    Blogger blogger1 = new Blogger(1L, "倪升武", "123456");
    Blogger blogger2 = new Blogger(2L, "达人课", "123456");
    List<Blogger> list = new ArrayList<>();
    list.add(blogger1);
    list.add(blogger2);
    model.addAttribute("list", list);
    return "list";
}
```

接下来我们写一个 list.html 来获取该 list 信息，然后在 list.html 中遍历这个 list。如下：

```
<!DOCTYPE html><html xmlns:th="http://www.thymeleaf.org"><html
lang="en"><head>
    <meta charset="UTF-8">
```



```

<title>博主信息</title></head><body><form action="" th:each="blogger :
${list}" >
    用户编号: <input name="id" th:value="${blogger.id}"/><br>
    用户姓名: <input type="text" name="password"
th:value="${blogger.name}"/><br>
    登录密码: <input type="text" name="username"
th:value="${blogger.getPass()}" /></form></body></html>

```

可以看出，其实和处理单个对象信息差不多，Thymeleaf 使用 `th:each` 进行遍历，`${}` 取 `model` 中传过来的参数，然后自定义 `list` 中取出来的每个对象，这里定义为 `blogger`。表单里面可以直接使用 `${对象.属性名}` 来获取 `list` 中对象的属性值，也可以使用 `${对象.get 方法}` 来获取，这点和上面处理对象信息是一样的，但是不能使用 `*{属性名}` 来获取对象中的属性，thymeleaf 模板获取不到。

## 4.4 其他常用 thymeleaf 操作

我们来总结一下 thymeleaf 中的一些常用的标签操作，如下：

标签	功能	例子
<code>th:value</code>	给属性赋值	<code>&lt;input th:value="\${blog.name}" /&gt;</code>
<code>th:style</code>	设置样式	<code>th:style="'display:' + @({\${sittrue}? 'none': 'inline-block'}) + '"</code>
<code>th:onclick</code>	点击事件	<code>th:onclick="'getInfo()'"</code>
<code>th:if</code>	条件判断	<code>&lt;a th:if="\${userId == collect.userId}" &gt;</code>
<code>th:href</code>	超链接	<code>&lt;a th:href="@{/blogger/login}"&gt;Login&lt;/a&gt; &lt;/&gt;</code>
<code>th:unless</code>	条件判断和 <code>th:if</code> 相反	<code>&lt;a th:href="@{/blogger/login}" th:unless=\${session.user != null}&gt;Login&lt;/a&gt;</code>
<code>th:switch</code>	配合 <code>th:case</code>	<code>&lt;div th:switch="\${user.role}"&gt;</code>
<code>th:case</code>	配合 <code>th:switch</code>	<code>&lt;p th:case="'admin'"&gt;administator&lt;/p&gt;</code>
<code>th:src</code>	地址引入	<code>&lt;img alt="csdn logo" th:src="@{/img/logo.png}" /&gt;</code>
<code>th:action</code>	表单提交的 地址	<code>&lt;form th:action="@{/blogger/update}"&gt;</code>



Thymeleaf 还有很多其他用法，这里就不总结了，具体的可以参考 Thymeleaf 的[官方文档（v3.0）](#)。主要要学会如何在 Spring Boot 中去使用 thymeleaf，遇到对应的标签或者方法，查阅官方文档即可。

## 5. 总结

Thymeleaf 在 Spring Boot 中使用非常广泛，本节课主要分析了 thymeleaf 的优点，以及如何在 Spring Boot 中集成并使用 thymeleaf 模板，包括依赖、配置，相关数据的获取、以及一些注意事项等等。最后列举了一些 thymeleaf 中常用的标签，在实际项目中多使用，多查阅就能熟练掌握，thymeleaf 中的一些标签或者方法不用死记硬背，用到什么去查阅什么，关键是要会在 Spring Boot 中集成，用的多了就熟能生巧。

课程源代码下载地址：[戳我下载](#)

# 第 08 课：Spring Boot 中的全局异常处理

在项目开发过程中，不管是对底层数据库的操作过程，还是业务层的处理过程，还是控制层的处理过程，都不可避免会遇到各种可预知的、不可预知的异常需要处理。如果对每个过程都单独作异常处理，那系统的代码耦合度会变得很高，此外，开发工作量也会加大而且不好统一，这也增加了代码的维护成本。

针对这种实际情况，我们需要将所有类型的异常处理从各处理过程解耦出来，这样既保证了相关处理过程的功能单一，也实现了异常信息的统一处理和维护。同时，我们也不希望直接把异常抛给用户，应该对异常进行处理，对错误信息进行封装，然后返回一个友好的信息给用户。这节主要总结一下项目中如何使用 Spring Boot 如何拦截并处理全局的异常。

## 1. 定义返回的统一 json 结构

前端或者其他服务请求本服务的接口时，该接口需要返回对应的 json 数据，一般该服务只需要返回请求着需要的参数即可，但是在实际项目中，我们需要封装更多的信息，比如状态码 code、相关信息 msg 等等，这一方面是在项目中可以有个统一的返回结构，整个项目组都适用，另一方面是方便结合全局异常处理信息，因为异常处理信息中一般我们需要把状态码和异常内容反馈给调用方。

这个统一的 json 结构这可以参考[第 02 课：Spring Boot 返回 JSON 数据及数据封装](#)中封装的统一 json 结构，本节内容我们简化一下，只保留状态码 code 和异常信息 msg 即可。如下：

```
public class JsonResult {  
    /**  
     * 异常码  
     */  
    protected String code;
```

```

    /**
     * 异常信息
     */
    protected String msg;

    public JsonResult() {
        this.code = "200";
        this.msg = "操作成功";
    }

    public JsonResult(String code, String msg) {
        this.code = code;
        this.msg = msg;
    }
    // get set
}

```

## 2. 处理系统异常

新建一个 `GlobalExceptionHandler` 全局异常处理类，然后加上 `@ControllerAdvice` 注解即可拦截项目中抛出的异常，如下：

```

@ControllerAdvice@ResponseBody public class GlobalExceptionHandler {
    // 打印 log
    private static final Logger logger =
        LoggerFactory.getLogger(GlobalExceptionHandler.class);
    // .....
}

```

我们点开 `@ControllerAdvice` 注解可以看到，`@ControllerAdvice` 注解包含了 `@Component` 注解，说明在 Spring Boot 启动时，也会把该类作为组件交给 Spring 来管理。除此之外，该注解还有个 `basePackages` 属性，该属性是用来拦截哪个包中的异常信息，一般我们不指定这个属性，我们拦截项目工程中的所有异常。  
`@ResponseBody` 注解是为了异常处理完之后给调用方输出一个 json 格式的封装数据。在项目中如何使用呢？Spring Boot 中很简单，在方法上通过 `@ExceptionHandler` 注解来指定具体的异常，然后在方法中处理该异常信息，最后将结果通过统一的 json 结构体返回给调用者。下面我们举几个例子来说明如何来使用。

### 2.1 处理参数缺失异常

在前后端分离的架构中，前端请求后台的接口都是通过 rest 风格来调用，有时候，比如 POST 请求 需要携带一些参数，但是往往有时候参数会漏掉。另外，在微服务架构

中，涉及到多个微服务之间的接口调用时，也可能出现这种情况，此时我们需要定义一个处理参数缺失异常的方法，来给前端或者调用方提示一个友好信息。

参数缺失的时候，会抛出 `HttpMessageNotReadableException`，我们可以拦截该异常，做一个友好处理，如下：

```
/**
 * 缺少请求参数异常
 * @param ex HttpMessageNotReadableException
 * @return
 */@ExceptionHandler(MissingServletRequestParameterException.class)@
ResponseStatus(value = HttpStatus.BAD_REQUEST)public JsonResult
handleHttpMessageNotReadableException(
    MissingServletRequestParameterException ex) {
    logger.error("缺少请求参数, {}", ex.getMessage());
    return new JsonResult("400", "缺少必要的请求参数");
}
```

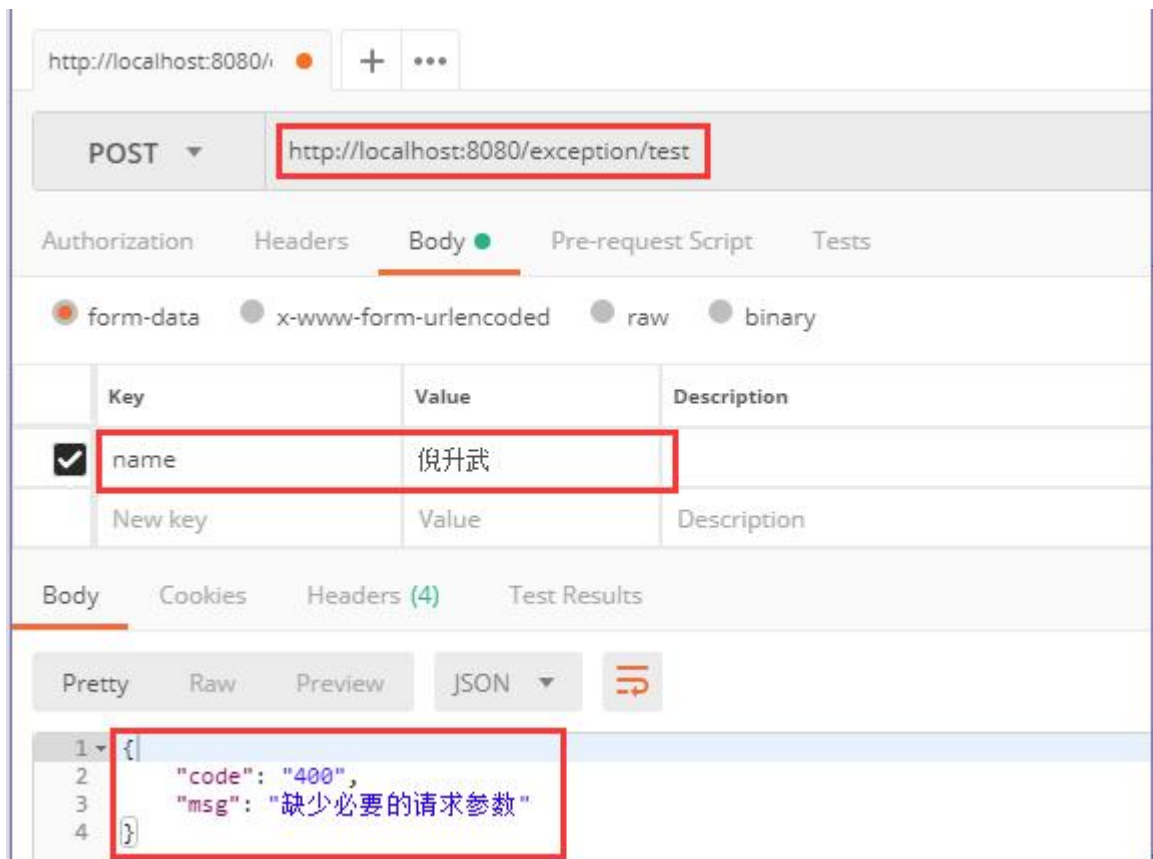
我们来写个简单的 Controller 测试一下该异常，通过 POST 请求方式接收两个参数：姓名和密码。

```
@RestController@RequestMapping("/exception")public class
ExceptionHandler {

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

    @PostMapping("/test")
    public JsonResult test(@RequestParam("name") String name,
        @RequestParam("pass") String pass) {
        logger.info("name: {}", name);
        logger.info("pass: {}", pass);
        return new JsonResult();
    }
}
```

然后使用 Postman 来调用一下该接口，调用的时候，只传姓名，不传密码，就会抛缺少参数异常，该异常被捕获之后，就会进入我们写好的逻辑，给调用方返回一个友好信息，如下：



## 2.2 处理空指针异常

空指针异常是开发中司空见惯的东西了，一般发生的地方有哪些呢？

先来聊一聊一些注意的地方，比如在微服务中，经常会调用其他服务获取数据，这个数据主要是 json 格式的，但是在解析 json 的过程中，可能会有空出现，所以我们在获取某个 jsonObject 时，再通过该 jsonObject 去获取相关信息时，应该要先做非空判断。还有一个很常见的地方就是从数据库中查询的数据，不管是查询一条记录封装在某个对象中，还是查询多条记录封装在一个 List 中，我们接下来都要去处理数据，那么就有可能出现空指针异常，因为谁也不能保证从数据库中查出来的东西就一定不为空，所以在在使用数据时一定要先做非空判断。

对空指针异常的处理很简单，和上面的逻辑一样，将异常信息换掉即可。如下：

```
@ControllerAdvice@ResponseBodypublic class GlobalExceptionHandler {  
  
    private static final Logger logger =  
        LoggerFactory.getLogger(GlobalExceptionHandler.class);  
  
    /**  
     * 空指针异常  
     * @param ex NullPointerException  
     * @return  
     */  
    @ExceptionHandler(NullPointerException.class)
```

```

        @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
        public JsonResult handleTypeMismatchException(NullPointerException
ex) {
            logger.error("空指针异常, {}", ex.getMessage());
            return new JsonResult("500", "空指针异常了");
        }
    }
}

```

这个我就不测试了，代码中 `ExceptionHandler` 有个 `testNullPointerException` 方法，模拟了一个空指针异常，我们在浏览器中请求一下对应的 url 即可看到返回的信息：

```

{"code": "500", "msg": "空指针异常了"}

```

## 2.3 一劳永逸？

当然了，异常很多，比如还有 `RuntimeException`，数据库还有一些查询或者操作异常等等。由于 `Exception` 异常是父类，所有异常都会继承该异常，所以我们可以直接拦截 `Exception` 异常，一劳永逸：

```

@ControllerAdvice@ResponseBody public class GlobalExceptionHandler {

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

    /**
     * 系统异常 预期以外异常
     * @param ex
     * @return
     */
    @ExceptionHandler(Exception.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public JsonResult handleUnexpectedServer(Exception ex) {
        logger.error("系统异常: ", ex);
        return new JsonResult("500", "系统发生异常，请联系管理员");
    }
}

```

但是项目中，我们一般都会比较详细的去拦截一些常见异常，拦截 `Exception` 虽然可以一劳永逸，但是不利于我们去排查或者定位问题。实际项目中，可以把拦截 `Exception` 异常写在 `GlobalExceptionHandler` 最下面，如果都没有找到，最后再拦截一下 `Exception` 异常，保证输出信息友好。

### 3. 拦截自定义异常

在实际项目中，除了拦截一些系统异常外，在某些业务上，我们需要自定义一些业务异常，比如在微服务中，服务之间的相互调用很平凡，很常见。要处理一个服务的调用时，那么可能会调用失败或者调用超时等等，此时我们需要自定义一个异常，当调用失败时抛出该异常，给 `GlobalExceptionHandler` 去捕获。

#### 3.1 定义异常信息

由于在业务中，有很多异常，针对不同的业务，可能给出的提示信息不同，所以为了方便项目异常信息管理，我们一般会定义一个异常信息枚举类。比如：

```
/**
 * 业务异常提示信息枚举类
 * @author shengwu ni
 */public enum BusinessMsgEnum {
    /** 参数异常 */
    PARMETER_EXCEPTION("102", "参数异常!"),
    /** 等待超时 */
    SERVICE_TIME_OUT("103", "服务调用超时!"),
    /** 参数过大 */
    PARMETER_BIG_EXCEPTION("102", "输入的图片数量不能超过 50 张!"),
    /** 500 : 一劳永逸的提示也可以在这定义 */
    UNEXPECTED_EXCEPTION("500", "系统发生异常，请联系管理员!");
    // 还可以定义更多的业务异常

    /**
     * 消息码
     */
    private String code;

    /**
     * 消息内容
     */
    private String msg;

    private BusinessMsgEnum(String code, String msg) {
        this.code = code;
        this.msg = msg;
    }
    // set get 方法
```

```
}
```

### 3.2 拦截自定义异常

然后我们可以定义一个业务异常，当出现业务异常时，我们就抛这个自定义的业务异常即可。比如我们定义一个 `BusinessErrorException` 异常，如下：

```
/**
 * 自定义业务异常
 * @author shengwu ni
 */public class BusinessErrorException extends RuntimeException {

    private static final long serialVersionUID = -
7480022450501760611L;

    /**
     * 异常码
     */
    private String code;

    /**
     * 异常提示信息
     */
    private String message;

    public BusinessErrorException(BusinessMsgEnum businessMsgEnum) {
        this.code = businessMsgEnum.code();
        this.message = businessMsgEnum.msg();
    }
    // get set 方法
}
```

在构造方法中，传入我们上面自定义的异常枚举类，所以在项目中，如果有新的异常信息需要添加，我们直接在枚举类中添加即可，很方便，做到统一维护，然后再拦截该异常时获取即可。

```
@ControllerAdvice@ResponseBodypublic class GlobalExceptionHandler {

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

    /**
     * 拦截业务异常，返回业务异常信息
```

```

    * @param ex
    * @return
    */
    @ExceptionHandler(BusinessErrorException.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public JsonResult handleBusinessError(BusinessErrorException ex) {
        String code = ex.getCode();
        String message = ex.getMessage();
        return new JsonResult(code, message);
    }
}

```

在业务代码中，我们可以直接模拟一下抛出业务异常，测试一下：

```

@RestController@RequestMapping("/exception") public class
ExceptionHandler {

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

    @GetMapping("/business")
    public JsonResult testException() {
        try {
            int i = 1 / 0;
        } catch (Exception e) {
            throw new
BusinessErrorException(BusinessMsgEnum.UNEXPECTED_EXCEPTION);
        }
        return new JsonResult();
    }
}

```

运行一下项目，测试一下，返回 json 如下，说明我们自定义的业务异常捕获成功：

```

{"code":"500","msg":"系统发生异常，请联系管理员！"}

```

## 4. 总结

本门课程主要讲解了 Spring Boot 的全局异常处理，包括异常信息的封装、异常信息的捕获和处理，以及在实际项目中，我们用到的自定义异常枚举类和业务异常的捕获与处理，在项目中运用的非常广泛，基本上每个项目中都需要做全局异常处理。



课程源代码下载地址：[戳我下载](#)

## 第 09 课：Spring Boot 中的切面 AOP 处理

### 1. 什么是 AOP

AOP: Aspect Oriented Programming 的缩写，意为：面向切面编程。面向切面编程的目标就是分离关注点。什么是关注点呢？就是关注点，就是你要做的事情。假如你是一位公子哥，没啥人生目标，每天衣来伸手，饭来张口，整天只知道一件事：玩（这就是你的关注点，你只要做这一件事）！但是有个问题，你在玩之前，你还需要起床、穿衣服、穿鞋子、叠被子、做早饭等等等等，但是这些事情你不想关注，也不用关注，你只想玩，那么怎么办呢？

对！这些事情通通交给下人去干。你有一个专门的仆人 A 帮你穿衣服，仆人 B 帮你穿鞋子，仆人 C 帮你叠好被子，仆人 D 帮你做饭，然后你就开始吃饭、去玩（这就是你一天的正事），你干完你的正事之后，回来，然后一系列仆人又开始帮你干这个干那个，然后一天就结束了！

这就是 AOP。AOP 的好处就是你只需要干你的正事，其它事情别人帮你干。也许有一天，你想裸奔，不想穿衣服，那么你把仆人 A 解雇就是了！也许有一天，出门之前你还想带点钱，那么你再雇一个仆人 E 专门帮你干取钱的活！这就是 AOP。每个人各司其职，灵活组合，达到一种可配置的、可插拔的程序结构。

### 2. Spring Boot 中的 AOP 处理

#### 2.1 AOP 依赖

使用 AOP，首先需要引入 AOP 的依赖。

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

#### 2.2 实现 AOP 切面

Spring Boot 中使用 AOP 非常简单，假如我们要在项目中打印一些 log，在引入了上面的依赖之后，我们新建一个类 LogAspectHandler，用来定义切面和处理方法。只要在类上加个 @Aspect 注解即可。@Aspect 注解用来描述一个切面类，定义切面类的时候需要打上这个注解。@Component 注解让该类交给 Spring 来管理。

```
@Aspect@Component public class LogAspectHandler {
```

```
}
```

这里主要介绍几个常用的注解及使用：

1. @Pointcut：定义一个切面，即上面所描述的关注的某件事入口。
2. @Before：在做某件事之前做的事。
3. @After：在做某件事之后做的事。
4. @AfterReturning：在做某件事之后，对其返回值做增强处理。
5. @AfterThrowing：在做某件事抛出异常时，处理。

### 2.2.1 @Pointcut 注解

**@Pointcut 注解：**用来定义一个切面（切入点），即上文中所关注的某件事情的入口。切入点决定了连接点关注的内容，使得我们可以控制通知什么时候执行。

```
@Aspect@Componentpublic class LogAspectHandler {  
  
    /**  
     * 定义一个切面，拦截 com.itcodai.course09.controller 包和子包下的所有方法  
     */  
    @Pointcut("execution(* com.itcodai.course09.controller..*.*(..))")  
    public void pointCut() {}  
}
```

**@Pointcut 注解**指定一个切面，定义需要拦截的东西，这里介绍两个常用的表达式：一个是使用 `execution()`，另一个是使用 `annotation()`。  
以 `execution(* com.itcodai.course09.controller..*.*(..))` 表达式为例，语法如下：

`execution()` 为表达式主体

第一个 \* 号的位置：表示返回值类型，\* 表示所有类型

包名：表示需要拦截的包名，后面的两个句点表示当前包和当前包的所有子包，`com.itcodai.course09.controller` 包、子包下所有类的方法

第二个 \* 号的位置：表示类名，\* 表示所有类

`*(..)`：这个星号表示方法名，\* 表示所有的方法，后面括弧里面表示方法的参数，两个句点表示任何参数

`annotation()` 方式是针对某个注解来定义切面，比如我们对具有 `@GetMapping` 注解的方法做切面，可以如下定义切面：

```
@Pointcut("@annotation(org.springframework.web.bind.annotation.GetMapping)")public void annotationCut() {}
```

然后使用该切面的话，就会切入注解是 `@GetMapping` 的方法。因为在实际项目中，可能对于不同的注解有不同的逻辑处理，比如 `@GetMapping`、`@PostMapping`、`@DeleteMapping` 等。所以这种按照注解的切入方式在实际项目中也很常用。

### 2.2.2 @Before 注解

`@Before` 注解指定的方法在切面切入目标方法之前执行，可以做一些 `log` 处理，也可以做一些信息的统计，比如获取用户的请求 `url` 以及用户的 `ip` 地址等等，这个在做个人站点的时候都能用得到，都是常用的方法。例如：

```
@Aspect@Componentpublic class LogAspectHandler {

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

    /**
     * 在上面定义的切面方法之前执行该方法
     * @param joinPoint joinPoint
     */
    @Before("pointCut()")
    public void doBefore(JoinPoint joinPoint) {
        logger.info("===doBefore 方法进入了===");

        // 获取签名
        Signature signature = joinPoint.getSignature();
        // 获取切入的包名
        String declaringTypeName = signature.getDeclaringTypeName();
        // 获取即将执行的方法名
        String funcName = signature.getName();
        logger.info("即将执行方法为: {}, 属于{}包", funcName,
declaringTypeName);

        // 也可以用来记录一些信息，比如获取请求的 url 和 ip
        ServletRequestAttributes attributes =
(ServletRequestAttributes)
RequestContextHolder.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();
        // 获取请求 url
        String url = request.getRequestURL().toString();
        // 获取请求 ip
        String ip = request.getRemoteAddr();
    }
}
```

```

        logger.info("用户请求的 url 为: {}, ip 地址为: {}", url, ip);
    }
}

```

JoinPoint 对象很有用，可以用它来获取一个签名，然后利用签名可以获取请求的包名、方法名，包括参数（通过 `joinPoint.getArgs()` 获取）等等。

### 2.2.3 @After 注解

@After 注解和 @Before 注解相对应，指定的方法在切面切入目标方法之后执行，也可以做一些完成某方法之后的 log 处理。

```

@Aspect@Componentpublic class LogAspectHandler {

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

    /**
     * 定义一个切面，拦截 com.itcodai.course09.controller 包下的所有方法
     */
    @Pointcut("execution(* com.itcodai.course09.controller..*.*(..))")
    public void pointCut() {}

    /**
     * 在上面定义的切面方法之后执行该方法
     * @param joinPoint joinPoint
     */
    @After("pointCut()")
    public void doAfter(JoinPoint joinPoint) {

        logger.info("====doAfter 方法进入了====");
        Signature signature = joinPoint.getSignature();
        String method = signature.getName();
        logger.info("方法{}已经执行完", method);
    }
}

```

到这里，我们来写一个 Controller 来测试一下执行结果，新建一个 AopController 如下：

```

@RestController@RequestMapping("/aop")public class AopController {

```

```

    @GetMapping("/{name}")
    public String testAop(@PathVariable String name) {
        return "Hello " + name;
    }
}

```

启动项目，在浏览器中输入 `localhost:8080/aop/CSDN`，观察一下控制台的输出信息：

```

====doBefore 方法进入了====
即将执行方法为：testAop，属于
com.itcodai.course09.controller.AopController 包
用户请求的 url 为：http://localhost:8080/aop/name，ip 地址为：
0:0:0:0:0:0:1
====doAfter 方法进入了====
方法 testAop 已经执行完

```

从打印出来的 log 中可以看出程序执行的逻辑与顺序，可以很直观的掌握 `@Before` 和 `@After` 两个注解的实际作用。

#### 2.2.4 @AfterReturning 注解

`@AfterReturning` 注解和 `@After` 有些类似，区别在于 `@AfterReturning` 注解可以用来捕获切入方法执行完之后的返回值，对返回值进行业务逻辑上的增强处理，例如：

```

@Aspect@Component
public class LogAspectHandler {

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

    /**
     * 在上面定义的切面方法返回后执行该方法，可以捕获返回对象或者对返回对象进行增强
     * @param joinPoint joinPoint
     * @param result result
     */
    @AfterReturning(pointcut = "pointCut()", returning = "result")
    public void doAfterReturning(JoinPoint joinPoint, Object result) {

        Signature signature = joinPoint.getSignature();
        String classMethod = signature.getName();
    }
}

```

```

        logger.info("方法{}执行完毕，返回参数为：{}", classMethod,
result);
        // 实际项目中可以根据业务做具体的返回值增强
        logger.info("对返回参数进行业务上的增强：{}", result + "增强版");
    }
}

```

需要注意的是：在 `@AfterReturning` 注解中，属性 `returning` 的值必须要和参数保持一致，否则会检测不到。该方法中的第二个入参就是被切方法的返回值，在 `doAfterReturning` 方法中可以对返回值进行增强，可以根据业务需要做相应的封装。我们重启一下服务，再测试一下（多余的 log 我不贴出来了）：

方法 `testAop` 执行完毕，返回参数为：Hello CSDN  
对返回参数进行业务上的增强：Hello CSDN 增强版

### 2.2.5 @AfterThrowing 注解

顾名思义，`@AfterThrowing` 注解是当被切方法执行时抛出异常时，会进入 `@AfterThrowing` 注解的方法中执行，在该方法中可以做一些异常的处理逻辑。要注意的是 `throwing` 属性的值必须要和参数一致，否则会报错。该方法中的第二个入参即为抛出的异常。

```

/**
 * 使用 AOP 处理 log
 * @author shengwu ni
 * @date 2018/05/04 20:24
 */
@Aspect@Component
public class LogAspectHandler {

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

    /**
     * 在上面定义的切面方法执行抛异常时，执行该方法
     * @param joinPoint joinPoint
     * @param ex ex
     */
    @AfterThrowing(pointcut = "pointCut()", throwing = "ex")
    public void afterThrowing(JoinPoint joinPoint, Throwable ex) {
        Signature signature = joinPoint.getSignature();
        String method = signature.getName();
        // 处理异常的逻辑
    }
}

```

```
        logger.info("执行方法{}出错，异常为：{}", method, ex);
    }
}
```

该方法我就不测试了，大家可以自行测试一下。

### 3. 总结

本节课针对 Spring Boot 中的切面 AOP 做了详细的讲解，主要介绍了 Spring Boot 中 AOP 的引入，常用注解的使用，参数的使用，以及常用 api 的介绍。AOP 在实际项目中很有用，对切面方法执行前后都可以根据具体的业务，做相应的预处理或者增强处理，同时也可以用作异常捕获处理，可以根据具体业务场景，合理去使用 AOP。

课程源代码下载地址：[戳我下载](#)

## 第 10 课：Spring Boot 集成 MyBatis

### 1. MyBatis 介绍

大家都知道，MyBatis 框架是一个持久层框架，是 Apache 下的顶级项目。Mybatis 可以让开发者的主要精力放在 sql 上，通过 Mybatis 提供的映射方式，自由灵活的生成满足需要的 sql 语句。使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs 映射成数据库中的记录，在国内可谓是占据了半壁江山。本节课主要通过两种方式对 Spring Boot 集成 MyBatis 做一讲解。重点讲解一下基于注解的方式。因为实际项目中使用注解的方式更多一点，更简洁一点，省去了很多 xml 配置（这不是绝对的，有些项目组中可能也在使用 xml 的方式）。

### 2. MyBatis 的配置

#### 2.1 依赖导入

Spring Boot 集成 MyBatis，需要导入 mybatis-spring-boot-starter 和 mysql 的依赖，这里我们使用的版本时 1.3.2，如下：

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version></dependency><dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
```

```
<scope>runtime</scope></dependency>
```

我们点开 `mybatis-spring-boot-starter` 依赖，可以看到我们之前使用 Spring 时候熟悉的依赖，就像我在课程的一开始介绍的那样，Spring Boot 致力于简化编码，使用 `starter` 系列将相关依赖集成在一起，开发者不需要关注繁琐的配置，非常方便。

```
<!-- 省去其他 --><dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId></dependency><dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis-spring</artifactId></dependency>
```

## 2.2 properties.yml 配置

我们再来看一下，集成 MyBatis 时需要在 `properties.yml` 配置文件中做哪些基本配置呢？

```
# 服务端口号
server:
  port: 8080

# 数据库地址
datasource:
  url: localhost:3306/blog_test

spring:
  datasource: # 数据库配置
    driver-class-name: com.mysql.jdbc.Driver
    url:
jdbc:mysql://${datasource.url}?useSSL=false&useUnicode=true&characterEncoding=utf-8&allowMultiQueries=true&autoReconnect=true&failOverReadOnly=false&maxReconnects=10
    username: root
    password: 123456
    hikari:
      maximum-pool-size: 10 # 最大连接池数
      max-lifetime: 1770000

mybatis:
  # 指定别名设置的包为所有 entity
  type-aliases-package: com.itcodai.course10.entity
  configuration:
```



```
map-underscore-to-camel-case: true # 驼峰命名规范
mapper-locations: # mapper 映射文件位置
- classpath:mapper/*.xml
```

我们来简单介绍一下上面的这些配置：关于数据库的相关配置，我就不详细的解说了，这点相信大家已经非常熟练了，配置一下用户名、密码、数据库连接等等，这里使用的连接池是 Spring Boot 自带的 hikari，感兴趣的朋友可以去百度或者谷歌搜一搜，了解一下。

这里说明一下 `map-underscore-to-camel-case: true`，用来开启驼峰命名规范，这个比较好用，比如数据库中字段名为：`user_name`，那么在实体类中可以定义属性为 `userName`（甚至可以写成 `username`，也能映射上），会自动匹配到驼峰属性，如果不这样配置的话，针对字段名和属性名不同的情况，会映射不到。

### 3. 基于 xml 的整合

使用原始的 xml 方式，需要新建 `UserMapper.xml` 文件，在上面的 `application.yml` 配置文件中，我们已经定义了 xml 文件的路径：`classpath:mapper/*.xml`，所以我们在 `resources` 目录下新建一个 `mapper` 文件夹，然后创建一个 `UserMapper.xml` 文件。

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd"><mapper
namespace="com.itcodai.course10.dao.UserMapper">
  <resultMap id="BaseResultMap" type="com.itcodai.course10.entity.User">
    <id column="id" jdbcType="BIGINT" property="id" />
    <result column="user_name" jdbcType="VARCHAR"
property="username" />
    <result column="password" jdbcType="VARCHAR"
property="password" />
  </resultMap>

  <select id="getUserByName" resultType="User"
parameterType="String">
    select * from user where user_name = #{username}
  </select></mapper>
```

这和整合 Spring 一样的，`namespace` 中指定的是对应的 Mapper，`<resultMap>` 中指定对应的实体类，即 `User`。然后在内部指定表的字段和实体的属性相对应即可。这里我们写一个根据用户名查询用户的 sql。

实体类中有 id, username 和 password, 我不在这贴代码, 大家可以下载源码查看。UserMapper.java 文件中写一个接口即可:

```
User getUserByName(String username);
```

中间省略 service 的代码, 我们写一个 Controller 来测试一下:

```
@RestControllerpublic class TestController {  
  
    @Resource  
    private UserService userService;  
  
    @RequestMapping("/getUserByName/{name}")  
    public User getUserByName(@PathVariable String name) {  
        return userService.getUserByName(name);  
    }  
}
```

启动项目, 在浏览器中输入: <http://localhost:8080/getUserByName/CSDN> 即可查询到数据库表中用户名为 CSDN 的用户信息 (事先搞两个数据进去即可):

```
{"id":2,"username":"CSDN","password":"123456"}
```

这里需要注意一下: Spring Boot 如何知道这个 Mapper 呢? 一种方法是在上面的 mapper 层对应的类上面添加 @Mapper 注解即可, 但是这种方法有个弊端, 当我们有很多个 mapper 时, 那么每一个类上面都得添加 @Mapper 注解。另一种比较简便的方法是在 Spring Boot 启动类上添加 @MapperScan 注解, 来扫描一个包下的所有 mapper。如下:

```
@SpringBootApplication@MapperScan("com.itcodai.course10.dao")public  
class Course10Application {  
  
    public static void main(String[] args) {  
        SpringApplication.run(Course10Application.class, args);  
    }  
}
```

这样的话, com.itcodai.course10.dao 包下的所有 mapper 都会被扫描到了。

## 4. 基于注解的整合

基于注解的整合就不需要 xml 配置文件了，MyBatis 主要提供了 @Select，@Insert，@Update，Delete 四个注解。这四个注解是用的非常多的，也很简单，注解后面跟上对应的 sql 语句即可，我们举个例子：

```
@Select("select * from user where id = #{id}")User getUser(Long id);
```

这跟 xml 文件中写 sql 语句是一样的，这样就不需要 xml 文件了，但是有个问题，有人可能会问，如果是两个参数呢？如果是两个参数，我们需要使用 @Param 注解来指定每一个参数的对应关系，如下：

```
@Select("select * from user where id = #{id} and user_name=#{name}")  
User getUserByIdAndName(@Param("id") Long id, @Param("name")  
String username);
```

可以看出，@Param 指定的参数应该要和 sql 中 #{ } 取的参数名相同，不同则取不到。可以在 controller 中自行测试一下，接口都在源码中，文章中我就不贴测试代码和结果了。

有个问题需要注意一下，一般我们在设计表字段后，都会根据自动生成工具生成实体类，这样的话，基本上实体类是能和表字段对应上的，最起码也是驼峰对应的，由于在上面配置文件中开启了驼峰的配置，所以字段都是能对的上。但是，万一有对不上的呢？我们也有解决办法，使用 @Results 注解来解决。

```
@Select("select * from user where id = #{id}")  
@Results({  
    @Result(property = "username", column = "user_name"),  
    @Result(property = "password", column = "password")  
})  
User getUser(Long id);
```

@Results 中的 @Result 注解是用来指定每一个属性和字段的对应关系，这样的话就可以解决上面说的这个问题了。

当然了，我们也可以 xml 和注解相结合使用，目前我们实际的项目中也是采用混用的方式，因为有时候 xml 方便，有时候注解方便，比如就上面这个问题来说，如果我们定义了上面的这个 UserMapper.xml，那么我们完全可以使用 @ResultMap 注解来替代 @Results 注解，如下：

```
@Select("select * from user where id = #{id}")  
@ResultMap("BaseResultMap")User getUser(Long id);
```

@ResultMap 注解中的值从哪来呢？对应的是 UserMapper.xml 文件中定义的 <resultMap> 时对应的 id 值：

```
<resultMap id="BaseResultMap" type="com.itcodai.course10.entity.User">
```

这种 xml 和注解结合着使用的情况也很常见，而且也减少了大量的代码，因为 xml 文件可以使用自动生成工具去生成，也不需要人为手动敲，所以这种使用方式也很常见。

## 5. 总结

本节课主要系统的讲解了 Spring Boot 集成 MyBatis 的过程，分为基于 xml 形式和基于注解的形式来讲解，通过实际配置手把手讲解了 Spring Boot 中 MyBatis 的使用方式，并针对注解方式，讲解了常见的问题已经解决方式，有很强的实战意义。在实际项目中，建议根据实际情况来确定使用哪种方式，一般 xml 和注解都在用。

课程源代码下载地址：[戳我下载](#)

# 第 11 课：Spring Boot 事务配置管理

## 1. 事务相关

场景：我们在开发企业应用时，由于数据操作在顺序执行的过程中，线上可能有各种无法预知的问题，任何一步操作都有可能发生异常，异常则会导致后续的操作无法完成。此时由于业务逻辑并未正确的完成，所以在之前操作过数据库的动作并不可靠，需要在这种情况下进行数据的回滚。

事务的作用就是为了保证用户的每一个操作都是可靠的，事务中的每一步操作都必须成功执行，只要有发生异常就回退到事务开始未进行操作的状态。这很好理解，转账、购票等等，必须整个事件流程全部执行完才能人为该事件执行成功，不能转钱转到一半，系统死了，转账人钱没了，收款人钱还没到。

事务管理是 Spring Boot 框架中最为常用的功能之一，我们在实际应用开发时，基本上在 service 层处理业务逻辑的时候都要加上事务，当然了，有时候可能由于场景需要，也不用加事务（比如我们就要往一个表里插数据，相互没有影响，插多少是多少，不能因为某个数据挂了，把之前插的全部回滚）。

## 2. Spring Boot 事务配置

### 2.1 依赖导入

在 Spring Boot 中使用事务，需要导入 mysql 依赖：

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
```

```
<version>1.3.2</version></dependency>
```

导入了 mysql 依赖后，Spring Boot 会自动注入 DataSourceTransactionManager，我们不需要任何其他配置就可以用 @Transactional 注解进行事务的使用。关于 mybatis 的配置，在上一节课中已经说明了，这里还是使用上一节课中的 mybatis 配置即可。

## 2.2 事务的测试

我们首先在数据库表中插入一条数据：|id|user\_name|password| |--:|--:|--:| |1|倪升武|123456|

然后我们写一个插入的 mapper：

```
public interface UserMapper {  
  
    @Insert("insert into user (user_name, password) values (#{username},  
#{password})")  
    Integer insertUser(User user);  
}
```

OK，接下来我们来测试一下 Spring Boot 中的事务处理，在 service 层，我们手动抛出个异常来模拟实际中出现的异常，然后观察一下事务有没有回滚，如果数据库中没有新的记录，则说明事务回滚成功。

```
@Servicepublic class UserServiceImpl implements UserService {  
  
    @Resource  
    private UserMapper userMapper;  
  
    @Override  
    @Transactional  
    public void insertUser(User user) {  
        // 插入用户信息  
        userMapper.insertUser(user);  
        // 手动抛出异常  
        throw new RuntimeException();  
    }  
}
```

我们来测试一下：

```
@RestControllerpublic class TestController {
```

```

@Resource
private UserService userService;

@PostMapping("/adduser")
public String addUser(@RequestBody User user) throws Exception {
    if (null != user) {
        userService.isertUser(user);
        return "success";
    } else {
        return "false";
    }
}
}

```

我们使用 **postman** 调用一下该接口，因为在程序中抛出了个异常，会造成事务回滚，我们刷新一下数据库，并没有增加一条记录，说明事务生效了。事务很简单，我们平时在使用的时候，一般不会有太多问题，但是并不仅仅如此.....

### 3. 常见问题总结

从上面的内容中可以看出，**Spring Boot** 中使用事务非常简单，**@Transactional** 注解即可解决问题，说是这么说，但是在实际项目中，是有很多小坑在等着我们，这些小坑是我们在写代码的时候没有注意到，而且正常情况下不容易发现这些小坑，等项目写大了，某一天突然出问题了，排查问题非常困难，到时候肯定是抓瞎，需要费很大的精力去排查问题。

这一小节，我专门针对实际项目中经常出现的，和事务相关的细节做一下总结，希望读者在读完之后，能够落实到自己的项目中，能有所受益。

#### 3.1 异常并没有被 ” 捕获 “ 到

首先要说的，就是异常并没有被 ”捕获“ 到，导致事务并没有回滚。我们在业务层代码中，也许已经考虑到了异常的存在，或者编辑器已经提示我们需要抛出异常，但是这里面有个需要注意的地方：并不是说我们把异常抛出来了，有异常了事务就会回滚，我们来看一个例子：

```

@Service
public class UserServiceImpl implements UserService {

    @Resource
    private UserMapper userMapper;

    @Override
    @Transactional

```

```

public void insertUser2(User user) throws Exception {
    // 插入用户信息
    userMapper.insertUser(user);
    // 手动抛出异常
    throw new SQLException("数据库异常");
}
}

```

我们看上面这个代码，其实并没有什么问题，手动抛出一个 `SQLException` 来模拟实际中操作数据库发生的异常，在这个方法中，既然抛出了异常，那么事务应该回滚，实际却不如此，读者可以使用我源码中 `controller` 的接口，通过 `postman` 测试一下，就会发现，仍然是可以插入一条用户数据的。

那么问题出在哪呢？因为 `Spring Boot` 默认的事务规则是遇到运行异常（`RuntimeException`）和程序错误（`Error`）才会回滚。比如上面我们的例子中抛出的 `SQLException` 就没有问题，但是抛出 `SQLException` 就无法回滚了。针对非运行时异常，如果要进行事务回滚的话，可以在 `@Transactional` 注解中使用 `rollbackFor` 属性来指定异常，比如 `@Transactional(rollbackFor = Exception.class)`，这样就没有问题了，所以在实际项目中，一定要指定异常。

### 3.2 异常被 ”吃“ 掉

这个标题很搞笑，异常怎么会被吃掉呢？还是回归到现实项目中去，我们在处理异常时，有两种方式，要么抛出去，让上一层来捕获处理；要么把异常 `try catch` 掉，在异常出现的地方给处理掉。就因为有这么这 `try...catch`，所以导致异常被 ”吃“ 掉，事务无法回滚。我们还是看上面那个例子，只不过简单修改一下代码：

```

@Service
public class UserServiceImpl implements UserService {

    @Resource
    private UserMapper userMapper;

    @Override
    @Transactional(rollbackFor = Exception.class)
    public void insertUser3(User user) {
        try {
            // 插入用户信息
            userMapper.insertUser(user);
            // 手动抛出异常
            throw new SQLException("数据库异常");
        } catch (Exception e) {
            // 异常处理逻辑
        }
    }
}

```

```
}
```

读者可以使用我源码中 `controller` 的接口，通过 `postman` 测试一下，就会发现，仍然是可以插入一条用户数据，说明事务并没有因为抛出异常而回滚。这个细节往往比上面那个坑更难以发现，因为我们的思维很容易导致 `try...catch` 代码的产生，一旦出现这种问题，往往排查起来比较费劲，所以我们平时在写代码时，一定要多思考，多注意这种细节，尽量避免给自己埋坑。

那这种怎么解决呢？直接往上抛，给上一层来处理即可，千万不要在事务中把异常自己“吃”掉。

### 3.3 事务的范围

事务范围这个东西比上面两个坑埋的更深！我之所以把这个也写上，是因为这是我之前在实际项目中遇到的，该场景在这个课程中我就不模拟了，我写一个 `demo` 让大家看一下，把这个坑记住即可，以后在写代码时，遇到并发问题，就会注意这个坑了，那么这节课也就有价值了。

我来写个 `demo`：

```
@Service
public class UserServiceImpl implements UserService {

    @Resource
    private UserMapper userMapper;

    @Override
    @Transactional(rollbackFor = Exception.class)
    public synchronized void insertUser4(User user) {
        // 实际中的具体业务.....
        userMapper.insertUser(user);
    }
}
```

可以看到，因为要考虑并发问题，我在业务层代码的方法上加了个 `synchronized` 关键字。我举个实际的场景，比如一个数据库中，针对某个用户，只有一条记录，下一个插入动作过来，会先判断该数据库中有没有相同的用户，如果有就不插入，就更新，没有才插入，所以理论上，数据库中永远就一条同一用户信息，不会出现同一数据库中插入了两条相同用户的信息。

但是在压测时，就会出现上面的问题，数据库中确实有两条同一用户的信息，分析其原因，在于事务的范围和锁的范围问题。

从上面方法中可以看到，方法上是加了事务的，那么也就是说，在执行该方法开始时，事务启动，执行完了后，事务关闭。但是 `synchronized` 没有起作用，其实根本原因是因为事务的范围比锁的范围大。也就是说，在加锁的那部分代码执行完之后，锁释放掉了，但是事务还没结束，此时另一个线程进来了，事务没结束的话，第二个线程进来



时，数据库的状态和第一个线程刚进来是一样的。即由于 `mysql InnoDB` 引擎的默认隔离级别是可重复读（在同一个事务里，`SELECT` 的结果是事务开始时时间点的状态），线程二事务开始的时候，线程一还没提交完成，导致读取的数据还没更新。第二个线程也做了插入动作，导致了脏数据。

这个问题可以避免，第一，把事务去掉即可（不推荐）；第二，在调用该 `service` 的地方加锁，保证锁的范围比事务的范围大即可。

## 4. 总结

本章主要总结了 `Spring Boot` 中如何使用事务，只要使用 `@Transactional` 注解即可使用，非常简单方便。除此之外，重点总结了三个在实际项目中可能遇到的坑点，这非常有意义，因为事务这东西不出问题还好，出了问题比较难以排查，所以总结的这三点注意事项，希望能帮助到开发中的朋友。

课程源代码下载地址：[戳我下载](#)

# 第 12 课：Spring Boot 中使用监听器

## 1. 监听器介绍

什么是 `web` 监听器？`web` 监听器是一种 `Servlet` 中特殊的类，它们能帮助开发者监听 `web` 中特定的事件，比如 `ServletContext`, `HttpSession`, `ServletRequest` 的创建和销毁；变量的创建、销毁和修改等。可以在某些动作前后增加处理，实现监控。

## 2. Spring Boot 中监听器的使用

`web` 监听器的使用场景很多，比如监听 `servlet` 上下文用来初始化一些数据、监听 `http session` 用来获取当前在线的人数、监听客户端请求的 `servlet request` 对象来获取用户的访问信息等等。这一节中，我们主要通过这三个实际的使用场景来学习一下 `Spring Boot` 中监听器的使用。

### 2.1 监听 Servlet 上下文对象

监听 `servlet` 上下文对象可以用来初始化数据，用于缓存。什么意思呢？我举一个很常见的场景，比如用户在点击某个站点的首页时，一般都会展现出首页的一些信息，而这些信息基本上或者大部分时间都保持不变的，但是这些信息都是来自数据库。如果用户的每次点击，都要从数据库中去获取数据的话，用户量少还可以接受，如果用户量非常大的话，这对数据库也是一笔很大的开销。

针对这种首页数据，大部分都不常更新的话，我们完全可以把它们缓存起来，每次用户点击的时候，我们都直接从缓存中拿，这样既可以提高首页的访问速度，又可以降低服务器的压力。如果做的更加灵活一点，可以再加个定时器，定期的来更新这个首页缓存。就类似与 `CSDN` 个人博客首页中排名的变化一样。

下面我们针对这个功能，来写一个 **demo**，在实际中，读者可以完全套用该代码，来实现自己项目中的相关逻辑。首先写一个 **Service**，模拟一下从数据库查询数据：

```
@Servicepublic class UserService {  
  
    /**  
     * 获取用户信息  
     * @return  
     */  
    public User getUser() {  
        // 实际中会根据具体的业务场景，从数据库中查询对应的信息  
        return new User(1L, "倪升武", "123456");  
    }  
}
```

然后写一个监听器，实现 `ApplicationListener<ContextRefreshedEvent>` 接口，重写 `onApplicationEvent` 方法，将 `ContextRefreshedEvent` 对象传进去。如果我们在加载或刷新应用上下文时，也重新刷新下我们预加载的资源，就可以通过监听 `ContextRefreshedEvent` 来做这样的事情。如下：

```
/**  
 * 使用 ApplicationListener 来初始化一些数据到 application 域中的监听器  
 * @author shengni ni  
 * @date 2018/07/05  
 */@Componentpublic class MyServletContextListener implements  
ApplicationListener<ContextRefreshedEvent> {  
  
    @Override  
    public void onApplicationEvent(ContextRefreshedEvent  
contextRefreshedEvent) {  
        // 先获取到 application 上下文  
        ApplicationContext applicationContext =  
contextRefreshedEvent.getApplicationContext();  
        // 获取对应的 service  
        UserService userService =  
applicationContext.getBean(UserService.class);  
        User user = userService.getUser();  
        // 获取 application 域对象，将查到的信息放到 application 域中  
        ServletContext application =  
applicationContext.getBean(ServletContext.class);  
        application.setAttribute("user", user);  
    }  
}
```

```
}
```

正如注释中描述的一样，首先通过 `contextRefreshedEvent` 来获取 `application` 上下文，再通过 `application` 上下文来获取 `UserService` 这个 `bean`，项目中可以根据实际业务场景，也可以获取其他的 `bean`，然后再调用自己的业务代码获取相应的数据，最后存储到 `application` 域中，这样前端在请求相应数据的时候，我们就可以直接从 `application` 域中获取信息，减少数据库的压力。下面写一个 `Controller` 直接从 `application` 域中获取 `user` 信息来测试一下。

```
@RestController@RequestMapping("/listener")public class TestController {

    @GetMapping("/user")
    public User getUser(HttpServletRequest request) {
        ServletContext application = request.getServletContext();
        return (User) application.getAttribute("user");
    }
}
```

启动项目，在浏览器中输入 `http://localhost:8080/listener/user` 测试一下即可，如果正常返回 `user` 信息，那么说明数据已经缓存成功。不过 `application` 这种是缓存在内存中，对内存会有消耗，后面的课程中我会讲到 `redis`，到时候再给大家介绍一下 `redis` 的缓存。

## 2.2 监听 HTTP 会话 Session 对象

监听器还有一个比较常用的地方就是用来监听 `session` 对象，来获取在线用户数量，现在有很多开发者都有自己的网站，监听 `session` 来获取当前在线用户数量是个很常见的使用场景，下面来介绍一下如何来使用。

```
/**
 * 使用 HttpSessionListener 统计在线用户数的监听器
 * @author shengwu ni
 * @date 2018/07/05
 */@Componentpublic class MyHttpSessionListener implements
HttpSessionListener {

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

    /**
     * 记录在线的用户数量
     */
    public Integer count = 0;
```

```

@Override
public synchronized void sessionCreated(HttpSessionEvent
httpSessionEvent) {
    logger.info("新用户上线了");
    count++;

    httpSessionEvent.getSession().getServletContext().setAttribute("count
", count);
}

@Override
public synchronized void sessionDestroyed(HttpSessionEvent
httpSessionEvent) {
    logger.info("用户下线了");
    count--;

    httpSessionEvent.getSession().getServletContext().setAttribute("count
", count);
}
}

```

可以看出，首先该监听器需要实现 `HttpSessionListener` 接口，然后重写 `sessionCreated` 和 `sessionDestroyed` 方法，在 `sessionCreated` 方法中传递一个 `HttpSessionEvent` 对象，然后将当前 `session` 中的用户数量加 1，`sessionDestroyed` 方法刚好相反，不再赘述。然后我们写一个 `Controller` 来测试一下。

```

@RestController@RequestMapping("/listener")public class TestController {

    /**
     * 获取当前在线人数，该方法有 bug
     * @param request
     * @return
     */
    @GetMapping("/total")
    public String getTotalUser(HttpServletRequest request) {
        Integer count = (Integer)
request.getSession().getServletContext().getAttribute("count");
        return "当前在线人数: " + count;
    }
}

```

该 Controller 中是直接获取当前 session 中的用户数量，启动服务器，在浏览器中输入 `localhost:8080/listener/total` 可以看到返回的结果是 1，再打开一个浏览器，请求相同的地址可以看到 count 是 2，这没有问题。但是如果关闭一个浏览器再打开，理论上应该还是 2，但是实际测试却是 3。原因是 session 销毁的方法没有执行（可以在后台控制台观察日志打印情况），当重新打开时，服务器找不到用户原来的 session，于是又重新创建了一个 session，那怎么解决该问题呢？我们可以将上面的 Controller 方法改造一下：

```
@GetMapping("/total2")public String getTotalUser(HttpServletRequest request, HttpServletResponse response) {
    Cookie cookie;
    try {
        // 把 sessionId 记录在浏览器中
        cookie = new Cookie("JSESSIONID",
URLDecoder.decode(request.getSession().getId(), "utf-8"));
        cookie.setPath("/");
        //设置 cookie 有效期为 2 天，设置长一点
        cookie.setMaxAge( 48*60 * 60);
        response.addCookie(cookie);
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    Integer count = (Integer)
request.getSession().getServletContext().getAttribute("count");
    return "当前在线人数: " + count;
}
```

可以看出，该处理逻辑是让服务器记得原来那个 session，即把原来的 sessionId 记录在浏览器中，下次再打开时，把这个 sessionId 传过去，这样服务器就不会重新再创建了。重启一下服务器，在浏览器中再次测试一下，即可避免上面的问题。

## 2.3 监听客户端请求 Servlet Request 对象

使用监听器获取用户的访问信息比较简单，实现 `ServletRequestListener` 接口即可，然后通过 request 对象获取一些信息。如下：

```
/**
 * 使用 ServletRequestListener 获取访问信息
 * @author shengwu ni
 * @date 2018/07/05
 */@Componentpublic class MyServletRequestListener implements
ServletRequestListener {
```

```

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

    @Override
    public void requestInitialized(ServletRequestEvent servletRequestEvent)
    {
        HttpServletRequest request = (HttpServletRequest)
            servletRequestEvent.getServletRequest();
        logger.info("session id 为: {}",
            request.getRequestId());
        logger.info("request url 为: {}", request.getRequestURL());

        request.setAttribute("name", "倪升武");
    }

    @Override
    public void requestDestroyed(ServletRequestEvent servletRequestEvent)
    {
        logger.info("request end");
        HttpServletRequest request = (HttpServletRequest)
            servletRequestEvent.getServletRequest();
        logger.info("request 域中保存的 name 值为: {}",
            request.getAttribute("name"));
    }
}

```

这个比较简单，不再赘述，接下来写一个 Controller 测试一下即可。

```

@GetMapping("/request") public String getRequestInfo(HttpServletRequest
request) {
    System.out.println("requestListener 中的初始化的 name 数据: " +
        request.getAttribute("name"));
    return "success";
}

```

### 3. Spring Boot 中自定义事件监听

在实际项目中，我们往往需要自定义一些事件和监听器来满足业务场景，比如在微服务中会有这样的场景：微服务 A 在处理完某个逻辑之后，需要通知微服务 B 去处理另一个逻辑，或者微服务 A 处理完某个逻辑之后，需要将数据同步到微服务 B，这种场景

非常普遍，这个时候，我们可以自定义事件以及监听器来监听，一旦监听到微服务 A 中的某事件发生，就去通知微服务 B 处理对应的逻辑。

### 3.1 自定义事件

自定义事件需要继承 `ApplicationEvent` 对象，在事件中定义一个 `User` 对象来模拟数据，构造方法中将 `User` 对象传进来初始化。如下：

```
/**
 * 自定义事件
 * @author shengwu ni
 * @date 2018/07/05
 */public class MyEvent extends ApplicationEvent {

    private User user;

    public MyEvent(Object source, User user) {
        super(source);
        this.user = user;
    }

    // 省去 get、set 方法
}
```

### 3.2 自定义监听器

接下来，自定义一个监听器来监听上面定义的 `MyEvent` 事件，自定义监听器需要实现 `ApplicationListener` 接口即可。如下：

```
/**
 * 自定义监听器，监听 MyEvent 事件
 * @author shengwu ni
 * @date 2018/07/05
 */@Componentpublic class MyEventListener implements
ApplicationListener<MyEvent> {

    @Override
    public void onApplicationEvent(MyEvent myEvent) {
        // 把事件中的信息获取到
        User user = myEvent.getUser();
        // 处理事件，实际项目中可以通知别的微服务或者处理其他逻辑等等
        System.out.println("用户名: " + user.getUsername());
    }
}
```

```

        System.out.println("密码: " + user.getPassword());
    }
}

```

然后重写 `onApplicationEvent` 方法，将自定义的 `MyEvent` 事件传进来，因为该事件中，我们定义了 `User` 对象（该对象在实际中就是需要处理的数据，在下文来模拟），然后就可以使用该对象的信息了。

OK，定义好了事件和监听器之后，需要手动发布事件，这样监听器才能监听到，这需要根据实际业务场景来触发，针对本文的例子，我写个触发逻辑，如下：

```

/**
 * UserService
 * @author shengwu ni
 */
@Service
public class UserService {

    @Resource
    private ApplicationContext applicationContext;

    /**
     * 发布事件
     * @return
     */
    public User getUser2() {
        User user = new User(1L, "倪升武", "123456");
        // 发布事件
        MyEvent event = new MyEvent(this, user);
        applicationContext.publishEvent(event);
        return user;
    }
}

```

在 `service` 中注入 `ApplicationContext`，在业务代码处理完之后，通过 `ApplicationContext` 对象手动发布 `MyEvent` 事件，这样我们自定义的监听器就能监听到，然后处理监听器中写好的业务逻辑。

最后，在 `Controller` 中写一个接口来测试一下：

```

@GetMapping("/request")
public String getRequestInfo(HttpServletRequest request) {

```



```
System.out.println("requestListener 中的初始化的 name 数据: " +
request.getAttribute("name"));
return "success";
}
```

在浏览器中输入 `http://localhost:8080/listener/publish`，然后观察一下控制台打印的用户名和密码，即可说明自定义监听器已经生效。

## 4. 总结

本课系统的介绍了监听器原理，以及在 **Spring Boot** 中如何使用监听器，列举了监听器的三个常用的案例，有很好的实战意义。最后讲解了项目中如何自定义事件和监听器，并结合微服务中常见的场景，给出具体的代码模型，均能运用到实际项目中去，希望读者认真消化。

课程源代码下载地址：[戳我下载](#)

# 第 13 课：Spring Boot 中使用拦截器

拦截器的原理很简单，是 **AOP** 的一种实现，专门拦截对动态资源的后台请求，即拦截对控制层的请求。使用场景比较多的是判断用户是否有权限请求后台，更拔高一层的使用场景也有，比如拦截器可以结合 **websocket** 一起使用，用来拦截 **websocket** 请求，然后做相应的处理等等。拦截器不会拦截静态资源，**Spring Boot** 的默认静态目录为 **resources/static**，该目录下的静态页面、**js**、**css**、图片等等，不会被拦截（也要看如何实现，有些情况也会拦截，我在下文会指出）。

## 1. 拦截器的快速使用

使用拦截器很简单，只需要两步即可：定义拦截器和配置拦截器。在配置拦截器中，**Spring Boot 2.0** 以后的版本和之前的版本有所不同，我会重点讲解一下这里可能出现的坑。

### 1.1 定义拦截器

定义拦截器，只需要实现 **HandlerInterceptor** 接口，**HandlerInterceptor** 接口是所有自定义拦截器或者 **Spring Boot** 提供的拦截器的鼻祖，所以，首先来了解下该接口。该接口中有三个方法：**preHandle(.....)**、**postHandle(.....)** 和 **afterCompletion(.....)**。

**preHandle(.....)** 方法：该方法的执行时机是，当某个 **url** 已经匹配到对应的 **Controller** 中的某个方法，且在这个方法执行之前。所以 **preHandle(.....)** 方法可以决定是否将请求放行，这是通过返回值来决定的，返回 **true** 则放行，返回 **false** 则

不会向后执行。

**postHandle(.....) 方法：**该方法的执行时机是，当某个 url 已经匹配到对应的 Controller 中的某个方法，且在执行完了该方法，但是在 DispatcherServlet 视图渲染之前。所以在这个方法中有个 ModelAndView 参数，可以在此做一些修改动作。

**afterCompletion(.....) 方法：**顾名思义，该方法是在整个请求处理完成后（包括视图渲染）执行，这时做一些资源的清理工作，这个方法只有在 **preHandle(.....)** 被成功执行后并且返回 true 才会被执行。

了解了该接口，接下来自定义一个拦截器。

```
/**
 * 自定义拦截器
 * @author shengwu ni
 * @date 2018/08/03
 */public class MyInterceptor implements HandlerInterceptor {

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

    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {

        HandlerMethod handlerMethod = (HandlerMethod) handler;
        Method method = handlerMethod.getMethod();
        String methodName = method.getName();
        logger.info("====拦截到了方法: {}, 在该方法执行之前执行====",
            methodName);
        // 返回 true 才会继续执行，返回 false 则取消当前请求
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler, ModelAndView
        modelAndView) throws Exception {

        logger.info("执行完方法之后进执行(Controller 方法调用之后)，但是此
            时还没进行视图渲染");
    }
}
```

```

@Override
public void afterCompletion(HttpServletRequest request,
HttpServletRequestResponse response, Object handler, Exception ex) throws
Exception {
    logger.info("整个请求都处理完咯, DispatcherServlet 也渲染了对应的
视图咯, 此时我可以做一些清理的工作了");
}
}

```

OK, 到此为止, 拦截器已经定义完成, 接下来就是对该拦截器进行拦截配置。

## 1.2 配置拦截器

在 Spring Boot 2.0 之前, 我们都是直接继承 `WebMvcConfigurerAdapter` 类, 然后重写 `addInterceptors` 方法来实现拦截器的配置。但是在 Spring Boot 2.0 之后, 该方法已经被废弃了 (当然, 也可以继续用), 取而代之的是 `WebMvcConfigurationSupport` 方法, 如下:

```

@Configuration
public class MyInterceptorConfig extends
WebMvcConfigurationSupport {

    @Override
    protected void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
MyInterceptor()).addPathPatterns("/*");
        super.addInterceptors(registry);
    }
}

```

在该配置中重写 `addInterceptors` 方法, 将我们上面自定义的拦截器添加进去, `addPathPatterns` 方法是添加要拦截的请求, 这里我们拦截所有的请求。这样就配置好拦截器了, 接下来写一个 Controller 测试一下:

```

@Controller@RequestMapping("/interceptor")
public class
InterceptorController {

    @RequestMapping("/test")
    public String test() {
        return "hello";
    }
}

```

让其跳转到 `hello.html` 页面，直接在 `hello.html` 中输出 `hello interceptor` 即可。启动项目，在浏览器中输入 `localhost:8080/interceptor/test` 看一下控制台的日志：

```
====拦截到了方法: test, 在该方法执行之前执行====
```

执行完方法之后进执行(Controller 方法调用之后)，但是此时还没进行视图渲染整个请求都处理完咯，`DispatcherServlet` 也渲染了对应的视图咯，此时我可以做一些清理的工作了

可以看出拦截器已经生效，并能看出其执行顺序。

### 1.3 解决静态资源被拦截问题

上文中已经介绍了拦截器的定义和配置，但是这样是否就没问题了呢？其实不然，如果使用上面这种配置的话，我们会发现一个缺陷，那就是静态资源被拦截了。可以在 `resources/static/` 目录下放置一个图片资源或者 `html` 文件，然后启动项目直接访问，即可看到无法访问的现象。

也就是说，虽然 Spring Boot 2.0 废弃了 `WebMvcConfigurerAdapter`，但是 `WebMvcConfigurationSupport` 又会导致默认的静态资源被拦截，这就需要我们手动将静态资源放开。

如何放开呢？除了在 `MyInterceptorConfig` 配置类中重写 `addInterceptors` 方法外，还需要再重写一个方法：`addResourceHandlers`，将静态资源放开：

```
/**
 * 用来指定静态资源不被拦截，否则继承 WebMvcConfigurationSupport 这种方式
 * 会导致静态资源无法直接访问
 * @param registry
 */@Overrideprotected void addResourceHandlers(ResourceHandlerRegistry
registry) {

    registry.addResourceHandler("/**").addResourceLocations("classpath:/sta
tic/");
    super.addResourceHandlers(registry);
}
```

这样配置好之后，重启项目，静态资源也可以正常访问了。如果你是个善于学习或者研究的人，那肯定不会止步于此，没错，上面这种方式的确能解决静态资源无法访问的问题，但是，还有更方便的方式来配置。

我们不继承 `WebMvcConfigurationSupport` 类，直接实现 `WebMvcConfigurer` 接口，然后重写 `addInterceptors` 方法，将自定义的拦截器添加进去即可，如下：

```

@Configurationpublic class MyInterceptorConfig implements
WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 实现 WebMvcConfigurer 不会导致静态资源被拦截
        registry.addInterceptor(new
MyInterceptor()).addPathPatterns("/**");
    }
}

```

这样就非常方便了，实现 WebMvcConfigure 接口的话，不会拦截 Spring Boot 默认的静态资源。

这两种方式都可以，具体他们之间的细节，感兴趣的读者可以做进一步的研究，由于这两种方式的不同，继承 WebMvcConfigurationSupport 类的方式可以用在前后端分离的项目中，后台不需要访问静态资源（就不需要放开静态资源了）；实现 WebMvcConfigure 接口的方式可以用在非前后端分离的项目中，因为需要读取一些图片、css、js 文件等等。

## 2. 拦截器使用实例

### 2.1 判断用户有没有登录

一般用户登录功能我们可以这么做，要么往 session 中写一个 user，要么针对每个 user 生成一个 token，第二种要更好一点，那么针对第二种方式，如果用户登录成功了，每次请求的时候都会带上该用户的 token，如果未登录，则没有该 token，服务端可以检测这个 token 参数的有无来判断用户有没有登录，从而实现拦截功能。我们改造一下 preHandle 方法，如下：

```

@Overridepublic boolean preHandle(HttpServletRequest request,
HttpServletRequest response, Object handler) throws Exception {

    HandlerMethod handlerMethod = (HandlerMethod) handler;
    Method method = handlerMethod.getMethod();
    String methodName = method.getName();
    logger.info("====拦截到了方法：{}，在该方法执行之前执行====",
methodName);

    // 判断用户有没有登陆，一般登陆之后的用户都有一个对应的 token
    String token = request.getParameter("token");
    if (null == token || "".equals(token)) {
        logger.info("用户未登录，没有权限执行.....请登录");
        return false;
    }
}

```

```
// 返回 true 才会继续执行, 返回 false 则取消当前请求
return true;
}
```

重启项目, 在浏览器中输入 `localhost:8080/interceptor/test` 后查看控制台日志, 发现被拦截, 如果在浏览器中输入 `localhost:8080/interceptor/test?token=123` 即可正常往下走。

## 2.2 取消拦截操作

根据上文, 如果我要拦截所有 `/admin` 开头的 url 请求的话, 需要在拦截器配置中添加这个前缀, 但是在实际项目中, 可能会有这种场景出现: 某个请求也是 `/admin` 开头的, 但是不能拦截, 比如 `/admin/login` 等等, 这样的话又需要去配置。那么, 可不可以做成一个类似于开关的东西, 哪里不需要拦截, 我就在哪里弄个开关上去, 做成这种灵活的可插拔的效果呢?

是可以的, 我们可以定义一个注解, 该注解专门用来取消拦截操作, 如果某个 **Controller** 中的方法我们不需要拦截掉, 即可在该方法上加上我们自定义的注解即可, 下面先定义一个注解:

```
/**
 * 该注解用来指定某个方法不用拦截
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UnInterception {
}
```

然后在 **Controller** 中的某个方法上添加该注解, 在拦截器处理方法中添加该注解取消拦截的逻辑, 如下:

```
@Override public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws Exception {

    HandlerMethod handlerMethod = (HandlerMethod) handler;
    Method method = handlerMethod.getMethod();
    String methodName = method.getName();
    logger.info("===拦截到了方法: {}, 在该方法执行之前执行===",
        methodName);

    // 通过方法, 可以获取该方法上的自定义注解, 然后通过注解来判断该方法是否
    要被拦截
    // @UnInterception 是我们自定义的注解
    UnInterception unInterception =
        method.getAnnotation(UnInterception.class);
```

```
    if (null != unInterception) {  
        return true;  
    }  
    // 返回 true 才会继续执行，返回 false 则取消当前请求  
    return true;  
}
```

Controller 中的方法代码可以参见源码，重启项目在浏览器中输入 `http://localhost:8080/interceptor/test2?token=123` 测试一下，可以看出，加了该注解的方法不会被拦截。

### 3. 总结

本节主要介绍了 Spring Boot 中拦截器的使用，从拦截器的创建、配置，到拦截器对静态资源的影响，都做了详细的分析。Spring Boot 2.0 之后拦截器的配置支持两种方式，可以根据实际情况选择不同的配置方式。最后结合实际中的使用，举了两个常用的场景，希望读者能够认真消化，掌握拦截器的使用。

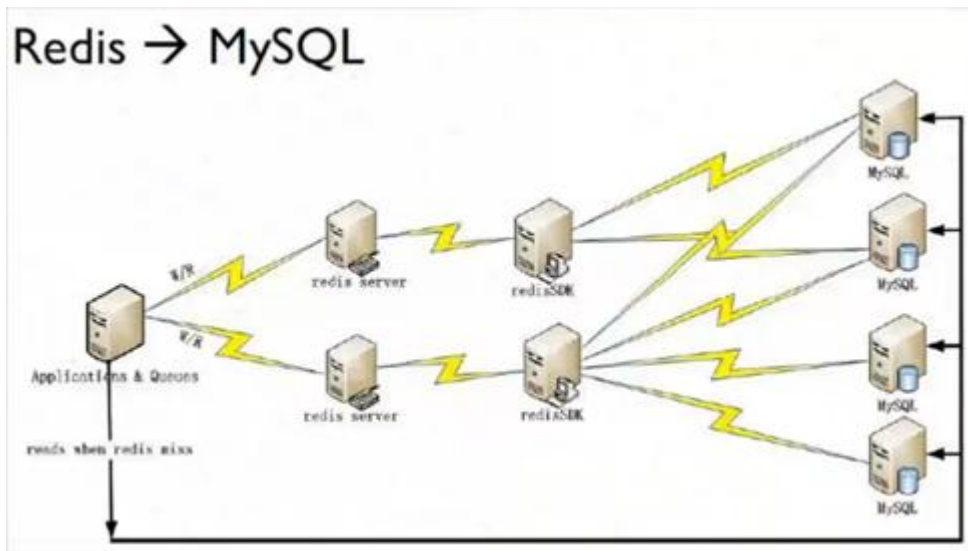
课程源代码下载地址：[戳我下载](#)

## 第 14 课：Spring Boot 中集成 Redis

### 1. Redis 介绍

Redis 是一种非关系型数据库（NoSQL），NoSQL 是以 key-value 的形式存储的，和传统的关系型数据库不一样，不一定遵循传统数据库的一些基本要求，比如说 SQL 标准，ACID 属性，表结构等等，这类数据库主要有以下特点：非关系型的、分布式的、开源的、水平可扩展的。NoSQL 使用场景有：对数据高并发读写、对海量数据的高效率存储和访问、对数据的高可扩展性和高可用性等等。Redis 的 key 可以是字符串、哈希、链表、集合和有序集合。value 类型很多，包括 String、list、set、zset。这些数据类型都支持 push/pop、add/remove、取交集和并集以及更多更丰富的操作，Redis 也支持各种不同方式的排序。为了保证效率，数据都是在缓存在内存中，它也可以周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件中。有了 redis 有哪些好处呢？举个比较简单的例子，看下图：





Redis 集群和 Mysql 是同步的，首先会从 redis 中获取数据，如果 redis 挂了，再从 mysql 中获取数据，这样网站就不会挂掉。更多关于 redis 的介绍以及使用场景，可以谷歌和百度，在这就不赘述了。

## 2. Redis 安装

本课程是在 vmware 虚拟机中来安装的 redis（centos 7），学习的时候如果有自己的阿里云服务器，也可以在阿里云中来安装 redis，都可以。只要能 ping 的通云主机或者虚拟机的 ip，然后在虚拟机或者云主机中放行对应的端口（或者关掉防火墙）即可访问 redis。下面来介绍一下 redis 的安装过程：

- 安装 gcc 编译

因为后面安装 redis 的时候需要编译，所以事先得先安装 gcc 编译。阿里云主机已经默认安装了 gcc，如果是自己安装的虚拟机，那么需要先安装一下 gcc：

```
yum install gcc-c++
```

- 下载 redis

有两种方式下载安装包，一种是去官网下载（<https://redis.io>），然后将安装包考到 centos 中，另一种方法是直接使用 wget 来下载：

```
wget http://download.redis.io/releases/redis-3.2.8.tar.gz
```

如果没有安装过 wget，可以通过如下命令安装：

```
yum install wget
```



- 解压安装

解压安装包：

```
tar -vzxf redis-3.2.8.tar.gz
```

然后将解压的文件夹 **redis-3.2.8** 放到 **/usr/local/** 下，一般安装软件都放在 **/usr/local** 下。然后进入 **/usr/local/redis-3.2.8/** 文件夹下，执行 **make** 命令即可完成安装。【注】如果 **make** 失败，可以尝试如下命令：

```
make MALLOC=libc  
make install
```

- 修改配置文件

安装成功之后，需要修改一下配置文件，包括允许接入的 ip，允许后台执行，设置密码等等。打开 **redis** 配置文件：**vi redis.conf** 在命令模式下输入 **/bind** 来查找 **bind** 配置，按 **n** 来查找下一个，找到配置后，将 **bind** 配置成 **0.0.0.0**，允许任意服务器来访问 **redis**，即：

```
bind 0.0.0.0
```

使用同样的方法，将 **daemonize** 改成 **yes**（默认为 **no**），允许 **redis** 在后台执行。将 **requirepass** 注释打开，并设置密码为 **123456**（密码自己设置）。

- 启动 redis

在 **redis-3.2.8** 目录下，指定刚刚修改好的配置文件 **redis.conf** 来启动 **redis**：

```
redis-server ./redis.conf
```

再启动 **redis** 客户端：

```
redis-cli
```

由于我们设置了密码，在启动客户端之后，输入 **auth 123456** 即可登录进入客户端。然后我们来测试一下，往 **redis** 中插入一个数据：

```
set name CSDN
```

然后来获取 **name**

```
get name
```

如果正常获取到 CSDN，则说明没有问题。

## 3. Spring Boot 集成 Redis

### 3.1 依赖导入

Spring Boot 集成 redis 很方便，只需要导入一个 redis 的 starter 依赖即可。如下：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-
redis</artifactId></dependency><!--阿里巴巴 fastjson --><dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.35</version></dependency>
```

这里也导入阿里巴巴的 fastjson 是为了在后面我们要存一个实体，为了方便把实体转换成 json 字符串存进去。

### 3.2 Redis 配置

导入了依赖之后，我们在 application.yml 文件里配置 redis：

```
server:
  port: 8080spring:
  #redis 相关配置
  redis:
    database: 5
    # 配置 redis 的主机地址，需要修改成自己的
    host: 192.168.48.190
    port: 6379
    password: 123456
    timeout: 5000
    jedis:
      pool:
        # 连接池中的最大空闲连接，默认值也是 8。
        max-idle: 500
        # 连接池中的最小空闲连接，默认值也是 0。
        min-idle: 50
```

```
# 如果赋值为-1，则表示不限制；如果 pool 已经分配了 maxActive 个
jedis 实例，则此时 pool 的状态为 exhausted(耗尽)
max-active: 1000
# 等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果
超过等待时间，则直接抛出 JedisConnectionException
max-wait: 2000
```

### 3.3 常用 api 介绍

Spring Boot 对 redis 的支持已经非常完善了，丰富的 api 已经足够我们日常的开发，这里我介绍几个最常用的供大家学习，其他 api 希望大家自己多学习，多研究。用到会去查即可。

有两个 redis 模板：RedisTemplate 和 StringRedisTemplate。我们不使用 RedisTemplate，RedisTemplate 提供给我们操作对象，操作对象的时候，我们通常是以 json 格式存储，但在存储的时候，会使用 Redis 默认的内部序列化器；导致我们存进里面的是乱码之类的东西。当然了，我们可以自己定义序列化，但是比较麻烦，所以使用 StringRedisTemplate 模板。StringRedisTemplate 主要给我们提供字符串操作，我们可以将实体类等转成 json 字符串即可，在取出来后，也可以转成相应的对象，这就是上面我导入了阿里 fastjson 的原因。

#### 3.3.1 redis:string 类型

新建一个 RedisService，注入 StringRedisTemplate，使用 stringRedisTemplate.opsForValue() 可以获取 ValueOperations<String, String> 对象，通过该对象即可读写 redis 数据库了。如下：

```
public class RedisService {

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    /**
     * set redis: string 类型
     * @param key key
     * @param value value
     */
    public void setString(String key, String value){
        ValueOperations<String, String> valueOperations =
stringRedisTemplate.opsForValue();
        valueOperations.set(key, value);
    }

    /**
```

```

* get redis: string 类型
* @param key key
* @return
*/
public String getString(String key){
    return stringRedisTemplate.opsForValue().get(key);
}

```

该对象操作的是 **string**，我们也可以存实体类，只需要将实体类转换成 **json** 字符串即可。下面来测试一下：

```

@RunWith(SpringRunner.class)@SpringBootTestpublic class
Course14ApplicationTests {

    private static final Logger logger =
LoggerFactory.getLogger(Course14ApplicationTests.class);

    @Resource
    private RedisService redisService;

    @Test
    public void contextLoads() {
        //测试 redis 的 string 类型
        redisService.setString("wechat","程序员私房菜");
        logger.info("我的微信公众号为: {}",
redisService.getString("wechat"));

        // 如果是个实体，我们可以使用 json 工具转成 json 字符串，
        User user = new User("CSDN", "123456");
        redisService.setString("userInfo", JSON.toJSONString(user));
        logger.info("用户信息: {}", redisService.getString("userInfo"));
    }
}

```

先启动 **redis**，然后运行这个测试用例，观察控制台打印的日志如下：

```

我的微信公众号为: 程序员私房菜
用户信息: {"password":"123456","username":"CSDN"}

```

### 3.3.2 redis:hash 类型

hash 类型其实原理和 string 一样的，但是有两个 key，使用 `stringRedisTemplate.opsForHash()` 可以获取 `HashOperations<String, Object, Object>` 对象。比如我们要存储订单信息，所有订单信息都放在 `order` 下，针对不同用户的订单实体，可以通过用户的 id 来区分，这就相当于两个 key 了。

```
@Servicepublic class RedisService {

    @Resource

    private StringRedisTemplate stringRedisTemplate;

    /**
     * set redis: hash 类型
     * @param key key
     * @param filedKey filedkey
     * @param value value
     */
    public void setHash(String key, String filedKey, String value){
        HashOperations<String, Object, Object> hashOperations =
stringRedisTemplate.opsForHash();
        hashOperations.put(key,filedKey, value);
    }

    /**
     * get redis: hash 类型
     * @param key key
     * @param filedkey filedkey
     * @return
     */
    public String getHash(String key, String filedkey){
        return (String) stringRedisTemplate.opsForHash().get(key,
filedkey);
    }
}
```

可以看出，hash 和 string 没啥两样，只不过多了个参数，Spring Boot 中操作 redis 非常简单方便。来测试一下：

```
@SpringBootTestpublic class Course14ApplicationTests {

    private static final Logger logger =
LoggerFactory.getLogger(Course14ApplicationTests.class);

    @Resource
```

```

    private RedisService redisService;

    @Test
    public void contextLoads() {
        //测试 redis 的 hash 类型
        redisService.setHash("user", "name", JSON.toJSONString(user));
        logger.info("用户姓名: {}",
redisService.getHash("user","name"));
    }
}

```

### 3.3.3 redis:list 类型

使用 `stringRedisTemplate.opsForList()` 可以获取 `ListOperations<String, String> listOperations` redis 列表对象，该列表是个简单的字符串列表，可以支持从左侧添加，也可以支持从右侧添加，一个列表最多包含  $2^{32}-1$  个元素。

```

@Service
public class RedisService {

    @Resource
    private StringRedisTemplate stringRedisTemplate;

    /**
     * set redis:list 类型
     * @param key key
     * @param value value
     * @return
     */
    public long setList(String key, String value){
        ListOperations<String, String> listOperations =
stringRedisTemplate.opsForList();
        return listOperations.leftPush(key, value);
    }

    /**
     * get redis:list 类型
     * @param key key
     * @param start start
     * @param end end
     * @return
     */
}

```

```

    public List<String> getList(String key, long start, long end){
        return stringRedisTemplate.opsForList().range(key, start,
end);
    }
}

```

可以看出，这些 **api** 都是一样的形式，方便记忆也方便使用。具体的 **api** 细节我就不展开了，大家可以自己看 **api** 文档。其实，这些 **api** 根据参数和返回值也能知道它们是做什么的。来测试一下：

```

@RunWith(SpringRunner.class)@SpringBootTestpublic class
Course14ApplicationTests {

    private static final Logger logger =
LoggerFactory.getLogger(Course14ApplicationTests.class);

    @Resource
    private RedisService redisService;

    @Test
    public void contextLoads() {
        //测试 redis 的 list 类型
        redisService.setList("list", "football");
        redisService.setList("list", "basketball");
        List<String> valList = redisService.getList("list",0,-1);
        for(String value :valList){
            logger.info("list 中有: {}", value);
        }
    }
}

```

## 4. 总结

本节主要介绍了 **redis** 的使用场景、安装过程，以及 **Spring Boot** 中集成 **redis** 的详细步骤。在实际项目中，通常都用 **redis** 作为缓存，在查询数据库的时候，会先从 **redis** 中查找，如果有信息，则从 **redis** 中取；如果没有，则从数据库中查，并且同步到 **redis** 中，下次 **redis** 中就有了。更新和删除也是如此，都需要同步到 **redis**。**redis** 在高频场景下运用的很多。

课程源代码下载地址：[戳我下载](#)

# 第 15 课： Spring Boot 中集成 ActiveMQ

## 1. JMS 和 ActiveMQ 介绍

### 1.1 JMS 是啥

百度百科的解释：

JMS 即 Java 消息服务（Java Message Service）应用程序接口，是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java 消息服务是一个与具体平台无关的 API，绝大多数 MOM 提供商都对 JMS 提供支持。

JMS 只是接口，不同的提供商或者开源组织对其有不同的实现，ActiveMQ 就是其中之一，它支持 JMS，是 Apache 推出的。JMS 中有几个对象模型：

连接工厂：ConnectionFactory  
JMS 连接：Connection  
JMS 会话：Session  
JMS 目的：Destination  
JMS 生产者：Producer  
JMS 消费者：Consumer  
JMS 消息两种类型：点对点和发布/订阅。

可以看出 JMS 实际上和 JDBC 有点类似，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。本文主要使用 ActiveMQ。

### 1.2 ActiveMQ

ActiveMQ 是 Apache 的一个能力强劲的开源消息总线。ActiveMQ 完全支持 JMS1.1 和 J2EE 1.4 规范，尽管 JMS 规范出台已经是很久的事情了，但是 JMS 在当今的 Java EE 应用中间仍然扮演着特殊的地位。ActiveMQ 用在异步消息的处理上，所谓异步消息即消息发送者无需等待消息接收者的处理以及返回，甚至无需关心消息是否发送成功。

异步消息主要有两种目的地形式，队列（queue）和主题（topic），队列用于点对点形式的消息通信，主题用于发布/订阅式的消息通信。本章节主要来学习一下在 Spring Boot 中如何使用这两种形式的消息。

## 2. ActiveMQ 安装

使用 ActiveMQ 首先需要去官网下载，官网地址为：<http://activemq.apache.org/>  
本课程使用的版本是 apache-activemq-5.15.3，下载后解压缩会有一个名为 apache-activemq-5.15.3 的文件夹，没错，这就安装好了，非常简单，开箱即用。打开文件夹



会看到里面有个 `activemq-all-5.15.3.jar`，这个 jar 我们是可以加进工程里的，但是使用 `maven` 的话，这个 jar 我们不需要。

在使用 `ActiveMQ` 之前，首先得先启动，刚才解压后的目录中有个 `bin` 目录，里面有 `win32` 和 `win64` 两个目录，根据自己电脑选择其中一个打开运行里面的 `activemq.bat` 即可启动 `ActiveMQ`。

消息生产者生产消息发布到 `queue` 中，然后消息消费者从 `queue` 中取出，并且消费消息。这里需要注意：消息被消费者消费以后，`queue` 中不再有存储，所以消息消费者不可消费到已经被消费的消息。`Queue` 支持存在多个消息消费者，但是对一个消息而言，只会有一个消费者可以消费 启动完成后，在浏览器中输入

`http://127.0.0.1:8161/admin/` 来访问 `ActiveMQ` 的服务器，用户名和密码是 `admin/admin`。如下：



# ActiveMQ<sup>TM</sup>

[Home](#) | [Queues](#) | [Topics](#) | [Subscribers](#) | [Connections](#) | [Network](#) | [Scheduled](#) | [Send](#)

## Welcome!

Welcome to the Apache ActiveMQ Console of **localhost** (ID:ifly-1741-57147-15354386200-0:1)

You can find more information about Apache ActiveMQ on the [Apache ActiveMQ Site](#)

### Broker

Name	localhost
Version	5.15.3
ID	ID:ifly-1741-57147-1535438866200-0:1
Uptime	44 minutes
Store percent used	0
Memory percent used	0
Temp percent used	0

我们可以看到有 `Queues` 和 `Topics` 这两个选项，这两个选项分别是点对点消息和发布/订阅消息的查看窗口。何为点对点消息和发布/订阅消息呢？

点对点消息：消息生产者生产消息发布到 `queue` 中，然后消息消费者从 `queue` 中取出，并且消费消息。这里需要注意：消息被消费者消费以后，`queue` 中不再有存储，所以消息消费者不可消费到已经被消费的消息。`Queue` 支持存在多个消息消费者，但是对一个消息而言，只会有一个消费者可以消费。

发布/订阅消息：消息生产者（发布）将消息发布到 **topic** 中，同时有多个消息消费者（订阅）消费该消息。和点对点方式不同，发布到 **topic** 的消息会被所有订阅者消费。下面分析具体的实现方式。

## 3. ActiveMQ 集成

### 3.1 依赖导入和配置

在 Spring Boot 中集成 ActiveMQ 需要导入如下 **starter** 依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId></dependency>
```

然后在 `application.yml` 配置文件中，对 `activemq` 做一下配置：

```
spring:
  activemq:
    # activemq url
    broker-url: tcp://localhost:61616
    in-memory: true
    pool:
      # 如果此处设置为 true，需要添加 activemq-pool 的依赖包，否则会自动配置失败，无法注入 JmsMessagingTemplate
      enabled: false
```

### 3.2 Queue 和 Topic 的创建

首先我们需要创建两种消息 Queue 和 Topic，这两种消息的创建，我们放到 `ActiveMqConfig` 中来创建，如下：

```
/**
 * activemq 的配置
 * @author shengwu ni
 */
@Configuration
public class ActiveMqConfig {
    /**
     * 发布/订阅模式队列名称
     */
    public static final String TOPIC_NAME = "activemq.topic";
}
```

```

* 点对点模式队列名称
*/
public static final String QUEUE_NAME = "activemq.queue";

@Bean
public Destination topic() {
    return new ActiveMQTopic(TOPIC_NAME);
}

@Bean
public Destination queue() {
    return new ActiveMQQueue(QUEUE_NAME);
}
}

```

可以看出创建 Queue 和 Topic 两种消息，分别使用 `new ActiveMQQueue` 和 `new ActiveMQTopic` 来创建，分别跟上对应消息的名称即可。这样在其他地方就可以直接将这两种消息作为组件注入进来了。

### 3.3 消息的发送接口

在 Spring Boot 中，我们只要注入 `JmsMessagingTemplate` 模板即可快速发送消息，如下：

```

/**
 * 消息发送者
 * @author shengwu ni
 */
@Service
public class MsgProducer {

    @Resource
    private JmsMessagingTemplate jmsMessagingTemplate;

    public void sendMessage(Destination destination, String msg) {
        jmsMessagingTemplate.convertAndSend(destination, msg);
    }
}

```

`convertAndSend` 方法中第一个参数是消息发送的目的地，第二个参数是具体的消息内容。

### 3.4 点对点消息生产与消费

#### 3.4.1 点对点消息的生产

消息的生产，我们放到 **Controller** 中来做，由于上面已经生成了 **Queue** 消息的组件，所以在 **Controller** 中我们直接注入进来即可。然后调用上文的消息发送方法 **sendMessage** 即可成功生产一条消息。

```
/**
 * ActiveMQ controller
 * @author shengwu ni
 */@RestController@RequestMapping("/activemq")public class
ActiveMqController {

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

    @Resource
    private MsgProducer producer;

    @Resource
    private Destination queue;

    @GetMapping("/send/queue")
    public String sendQueueMessage() {

        logger.info("===开始发送点对点消息===");
        producer.sendMessage(queue, "Queue: hello activemq!");
        return "success";
    }
}
```

### 3.4.2 点对点消息的消费

点对点消息的消费很简单，只要我们指定目的地即可，jms 监听器一直在监听是否有消息过来，如果有，则消费。

```
/**
 * 消息消费者
 * @author shengwu ni
 */@Servicepublic class QueueConsumer {

    /**
     * 接收点对点消息
     * @param msg
     */
}
```

```

    @JmsListener(destination = ActiveMqConfig.QUEUE_NAME)
    public void receiveQueueMsg(String msg) {
        System.out.println("收到的消息为: " + msg);
    }
}

```

可以看出，使用 `@JmsListener` 注解来指定要监听的目的地，在消息接收方法内部，我们可以根据具体的业务需求做相应的逻辑处理即可。

### 3.4.3 测试一下

启动项目，在浏览器中输入：`http://localhost:8081/activemq/send/queue`，观察控制台的输出日志，出现下面的日志说明消息发送和消费成功。

收到的消息为: *Queue: hello activemq!*

## 3.5 发布/订阅消息的生产和消费

### 3.5.1 发布/订阅消息的生产

和点对点消息一样，我们注入 `topic` 并调用 `producer` 的 `sendMessage` 方法即可发送订阅消息，如下，不再赘述：

```

@RestController@RequestMapping("/activemq")public class
ActiveMqController {

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

    @Resource
    private MsgProducer producer;
    @Resource
    private Destination topic;

    @GetMapping("/send/topic")
    public String sendTopicMessage() {

        logger.info("===开始发送订阅消息===");
        producer.sendMessage(topic, "Topic: hello activemq!");
        return "success";
    }
}

```

### 3.5.2 发布/订阅消息的消费

发布/订阅消息的消费和点对点不同，订阅消息支持多个消费者一起消费。其次，Spring Boot 中默认的点对点消息，所以在使用 topic 时，会不起作用，我们需要在配置文件 application.yml 中添加一个配置：

```
spring:
  jms:
    pub-sub-domain: true
```

该配置是 false 的话，则为点对点消息，也是 Spring Boot 默认的。这样是可以解决问题，但是如果这样配置的话，上面提到的点对点消息又不能正常消费了。所以二者不可兼得，这并非一个好的解决办法。

比较好的解决办法是，我们定义一个工厂，@JmsListener 注解默认只接收 queue 消息，如果要接收 topic 消息，需要设置一下 containerFactory。我们还在上面的那个 ActiveMqConfig 配置类中添加：

```
/**
 * activemq 的配置
 *
 * @author shengwu ni
 */
@Configuration
public class ActiveMqConfig {
    // 省略其他内容

    /**
     * JmsListener 注解默认只接收 queue 消息,如果要接收 topic 消息,需要设置
     containerFactory
     */
    @Bean
    public JmsListenerContainerFactory
    topicListenerContainer(ConnectionFactory connectionFactory) {
        DefaultJmsListenerContainerFactory factory = new
        DefaultJmsListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        // 相当于在 application.yml 中配置: spring.jms.pub-sub-
        domain=true
        factory.setPubSubDomain(true);
        return factory;
    }
}
```

经过这样的配置之后，我们在消费的时候，在 `@JmsListener` 注解中指定这个容器工厂即可消费 topic 消息。如下：

```
/**
 * Topic 消息消费者
 * @author shengwu ni
 */@Servicepublic class TopicConsumer1 {

    /**
     * 接收订阅消息
     * @param msg
     */
    @JmsListener(destination = ActiveMqConfig.TOPIC_NAME,
containerFactory = "topicListenerContainer")
    public void receiveTopicMsg(String msg) {
        System.out.println("收到的消息为: " + msg);
    }
}
```

指定 `containerFactory` 属性为上面我们自己配置的 `topicListenerContainer` 即可。由于 topic 消息可以多个消费，所以该消费的类可以拷贝几个一起测试一下，这里我就不贴代码了，可以参考我的源码测试。

### 3.5.3 测试一下

启动项目，在浏览器中输入：<http://localhost:8081/activemq/send/topic>，观察控制台的输出日志，出现下面的日志说明消息发送和消费成功。

收到的消息为: Topic: hello activemq!

收到的消息为: Topic: hello activemq!

## 4. 总结

本章主要介绍了 `jms` 和 `activemq` 的相关概念、`activemq` 的安装与启动。详细分析了 `Spring Boot` 中点对点消息和发布/订阅消息两种方式的配置、消息生产和消费方式。`ActiveMQ` 是能力强劲的开源消息总线，在异步消息的处理上很有用，希望大家好好消化一下。

课程源代码下载地址：[戳我下载](#)

# 第 16 课：Spring Boot 中集成 Shiro

Shiro 是一个强大、简单易用的 Java 安全框架，主要用来更便捷的认证，授权，加密，会话管等等，可为任何应用提供安全保障。本课程主要来介绍 Shiro 的认证和授权功能。

## 1. Shiro 三大核心组件

Shiro 有三大核心的组件：Subject、SecurityManager 和 Realm。先来看一下它们之间的关系。

1. **Subject**: 认证主体。它包含两个信息：Principals 和 Credentials。看一下这两个信息具体是什么。

Principals: 身份。可以是用户名，邮件，手机号码等等，用来标识一个登录主体身份；

Credentials: 凭证。常见有密码，数字证书等等。

- 2.

说白了，就是需要认证的东西，最常见的就是用户名密码了，比如用户在登录的时候，Shiro 需要去进行身份认证，就需要 Subject 认证主体。

- 1.

**SecurityManager**: 安全管理员。这是 Shiro 架构的核心，它就像 Shiro 内部所有原件的保护伞一样。我们在项目中一般都会配置 SecurityManager，开发人员大部分精力主要是在 Subject 认证主体上面。我们在与 Subject 进行交互的时候，实际上是 SecurityManager 在背后做一些安全操作。

- 2.

- 3.

**Realms**: Realms 是一个域，它是连接 Shiro 和具体应用的桥梁，当需要与安全数据交互的时候，比如用户账户、访问控制等，Shiro 就会从一个或多个 Realms 中去查找。我们一般会自己定制 Realm，这在下文会详细说明。

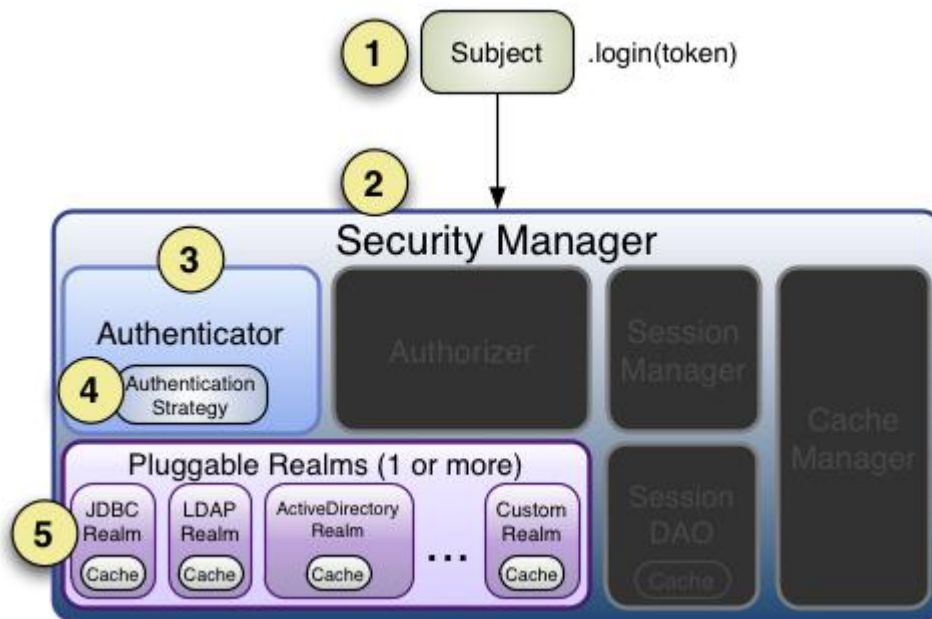
- 4.

## 1. Shiro 身份和权限认证

### 1.2 Shiro 身份认证

我们来分析一下 Shiro 身份认证的过程，看一下官方的一个认证图：





**Step1:** 应用程序代码在调用 `Subject.login(token)` 方法后，传入代表最终用户的身份和凭证的 `AuthenticationToken` 实例 `token`。

**Step2:** 将 `Subject` 实例委托给应用程序的 `SecurityManager`（Shiro 的安全管理）来开始实际的认证工作。这里开始真正的认证工作了。

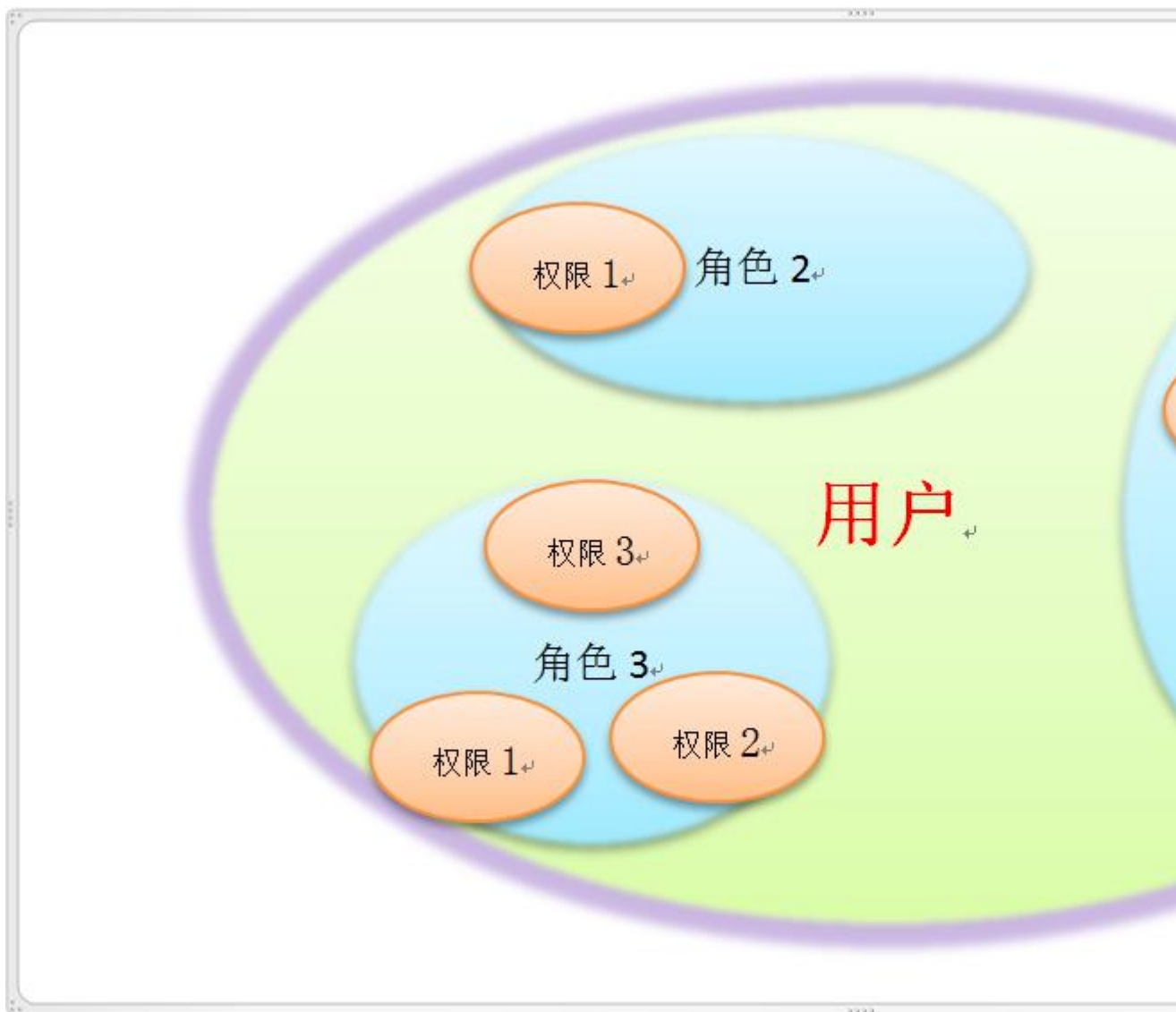
**Step3, 4, 5:** 然后 `SecurityManager` 就会根据具体的 `realm` 去进行安全认证了。从图中可以看出，`realm` 可以自定义（`Custom Realm`）。

### 1.3 Shiro 权限认证

权限认证，也就是访问控制，即在应用中控制谁能访问哪些资源。在权限认证中，最核心的三个要素是：权限，角色和用户。

权限（`permission`）：即操作资源的权利，比如访问某个页面，以及对某个模块的数据的添加，修改，删除，查看的权利；  
角色（`role`）：指的是用户担任的角色，一个角色可以有多个权限；  
用户（`user`）：在 Shiro 中，代表访问系统的用户，即上面提到的 `Subject` 认证主体。

它们之间的关系可以用下图来表示：



一个用户可以有多个角色，而不同的角色可以有不同的权限，也可由有相同的权限。比如说现在有三个角色，1 是普通角色，2 也是普通角色，3 是管理员，角色 1 只能查看信息，角色 2 只能添加信息，管理员都可以，而且还可以删除信息，类似于这样。

## 2. Spring Boot 集成 Shiro 过程

### 2.1 依赖导入

Spring Boot 2.0.3 集成 Shiro 需要导入如下 starter 依赖：

```
<dependency>
  <groupId>org.apache.shiro</groupId>
  <artifactId>shiro-spring</artifactId>
  <version>1.4.0</version></dependency>
```

## 2.2 数据库表数据初始化

这里主要涉及到三张表：用户表、角色表和权限表，其实在 demo 中，我们完全可以自己模拟一下，不用建表，但是为了更加接近实际情况，我们还是加入 mybatis，来操作数据库。下面是数据库表的脚本。

```
CREATE TABLE `t_role` (  
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',  
  `rolename` varchar(20) DEFAULT NULL COMMENT '角色名称',  
  PRIMARY KEY (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8  
CREATE TABLE `t_user` (  
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '用户主键',  
  `username` varchar(20) NOT NULL COMMENT '用户名',  
  `password` varchar(20) NOT NULL COMMENT '密码',  
  `role_id` int(11) DEFAULT NULL COMMENT '外键关联 role 表',  
  PRIMARY KEY (`id`),  
  KEY `role_id` (`role_id`),  
  CONSTRAINT `t_user_ibfk_1` FOREIGN KEY (`role_id`) REFERENCES  
  `t_role` (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8  
CREATE TABLE `t_permission` (  
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',  
  `permissionname` varchar(50) NOT NULL COMMENT '权限名',  
  `role_id` int(11) DEFAULT NULL COMMENT '外键关联 role',  
  PRIMARY KEY (`id`),  
  KEY `role_id` (`role_id`),  
  CONSTRAINT `t_permission_ibfk_1` FOREIGN KEY (`role_id`)  
  REFERENCES `t_role` (`id`)  
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8
```

其中，t\_user，t\_role 和 t\_permission，分别存储用户信息，角色信息和权限信息，表建立好了之后，我们往表里插入一些测试数据。t\_user 表：

id	username	password	role_id
1	csdn1	123456	1
2	csdn2	123456	2
3	csdn3	123456	3

t\_role 表：

id	rolename
1	admin
2	teacher
3	student

t\_permission 表：

id	permissionname	role_id
1	user:*	1
2	student:*	2

解释一下这里的权限：`user:*`表示权限可以是 `user:create` 或者其他，`*` 处表示一个占位符，我们可以自己定义，具体的会在下文 `Shiro` 配置那里说明。

## 2.2 自定义 Realm

有了数据库表和数据之后，我们开始自定义 `realm`，自定义 `realm` 需要继承 `AuthorizingRealm` 类，因为该类封装了很多方法，它也是一步步继承自 `Realm` 类的，继承了 `AuthorizingRealm` 类后，需要重写两个方法：

`doGetAuthenticationInfo()` 方法：用来验证当前登录的用户，获取认证信息

`doGetAuthorizationInfo()` 方法：用来为当前登陆成功的用户授予权限和角色

具体实现如下，相关的解释我放在代码的注释中，这样更加方便直观：

```
/**
 * 自定义 realm
 * @author shengwu ni
 */public class MyRealm extends AuthorizingRealm {

    @Resource
    private UserService userService;

    @Override
    protected AuthorizationInfo
doGetAuthorizationInfo(PrincipalCollection principalCollection) {
        // 获取用户名
        String username = (String)
principalCollection.getPrimaryPrincipal();
        SimpleAuthorizationInfo authorizationInfo = new
SimpleAuthorizationInfo();
        // 给该用户设置角色，角色信息存在 t_role 表中取
        authorizationInfo.setRoles(userService.getRoles(username));
        // 给该用户设置权限，权限信息存在 t_permission 表中取

        authorizationInfo.setStringPermissions(userService.getPermissions(use
rname));
        return authorizationInfo;
    }

    @Override
```

```

        protected AuthenticationInfo
doGetAuthenticationInfo(AuthenticationToken authenticationToken)
throws AuthenticationException {
    // 根据 token 获取用户名，如果您不知道该 token 怎么来的，先可以不管，下文会解释
    String username = (String) authenticationToken.getPrincipal();
    // 根据用户名从数据库中查询该用户
    User user = userService.getByUsername(username);
    if(user != null) {
        // 把当前用户存到 session 中

SecurityUtils.getSubject().getSession().setAttribute("user", user);
        // 传入用户名和密码进行身份认证，并返回认证信息

        AuthenticationInfo authcInfo = new
SimpleAuthenticationInfo(user.getUsername(), user.getPassword(),
"myRealm");
        return authcInfo;
    } else {
        return null;
    }
}
}

```

从上面两个方法中可以看出：验证身份的时候是根据用户输入的用户名先从数据库中查出该用户名对应的用户，这时候并没有涉及到密码，也就是说到这一步的时候，即使用户输入的密码不对，也是可以查出来该用户的，然后将该用户的正确信息封装到 `authcInfo` 中返回给 Shiro，接下来就是 Shiro 的事了，它会根据这里的真实信息与用户前台输入的用户名和密码进行校验，这个时候也要校验密码了，如果校验通过就让用户登录，否则跳转到指定页面。同理，权限验证的时候也是先根据用户名从数据库中获取与该用户名有关的角色和权限，然后封装到 `authorizationInfo` 中返回给 Shiro。

## 2.3 Shiro 配置

自定义的 `realm` 写好了，接下来需要对 Shiro 进行配置了。我们主要配置三个东西：自定义 `realm`、安全管理器 `SecurityManager` 和 Shiro 过滤器。如下：

配置自定义 `realm`：

```

@Configurationpublic class ShiroConfig {

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

    /**

```

```

    * 注入自定义的 realm
    * @return MyRealm
    */
    @Bean
    public MyRealm myAuthRealm() {
        MyRealm myRealm = new MyRealm();
        logger.info("===myRealm 注册完成====");
        return myRealm;
    }
}

```

配置安全管理器 SecurityManager:

```

@Configurationpublic class ShiroConfig {

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

    /**
    * 注入安全管理器
    * @return SecurityManager
    */
    @Bean
    public SecurityManager securityManager() {
        // 将自定义 realm 加进来
        DefaultWebSecurityManager securityManager = new
        DefaultWebSecurityManager(myAuthRealm());
        logger.info("===securityManager 注册完成====");
        return securityManager;
    }
}

```

配置 SecurityManager 时，需要将上面的自定义 realm 添加进来，这样的话 Shiro 才会走到自定义的 realm 中。

配置 Shiro 过滤器:

```

@Configurationpublic class ShiroConfig {

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

```

```

/**
 * 注入 Shiro 过滤器
 * @param securityManager 安全管理器
 * @return ShiroFilterFactoryBean
 */
@Bean
public ShiroFilterFactoryBean shiroFilter(SecurityManager
securityManager) {
    // 定义 shiroFactoryBean
    ShiroFilterFactoryBean shiroFilterFactoryBean=new
ShiroFilterFactoryBean();

    // 设置自定义的 securityManager
    shiroFilterFactoryBean.setSecurityManager(securityManager);

    // 设置默认登录的 url，身份认证失败会访问该 url
    shiroFilterFactoryBean.setLoginUrl("/login");
    // 设置成功之后要跳转的链接
    shiroFilterFactoryBean.setSuccessUrl("/success");
    // 设置未授权界面，权限认证失败会访问该 url
    shiroFilterFactoryBean.setUnauthorizedUrl("/unauthorized");

    // LinkedHashMap 是有序的，进行顺序拦截器配置
    Map<String,String> filterChainMap = new LinkedHashMap<>();

    // 配置可以匿名访问的地址，可以根据实际情况自己添加，放行一些静态资
    源等，anon 表示放行
    filterChainMap.put("/css/**", "anon");
    filterChainMap.put("/imgs/**", "anon");
    filterChainMap.put("/js/**", "anon");
    filterChainMap.put("/swagger-*/**", "anon");
    filterChainMap.put("/swagger-ui.html/**", "anon");
    // 登录 url 放行
    filterChainMap.put("/login", "anon");

    // “/user/admin” 开头的需要身份认证，authc 表示要身份认证
    filterChainMap.put("/user/admin*", "authc");
    // “/user/student” 开头的需要角色认证，是“admin”才允许

```

```

        filterChainMap.put("/user/student/**", "roles[admin]");
        // “/user/teacher” 开头的需要权限认证，是“user:create”才允许
        filterChainMap.put("/user/teacher/**",
"perms[\"user:create\"]");

        // 配置 logout 过滤器
        filterChainMap.put("/logout", "logout");

        // 设置 shiroFilterFactoryBean 的 FilterChainDefinitionMap
shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainMap);
        logger.info("====shiroFilterFactoryBean 注册完成====");
        return shiroFilterFactoryBean;
    }
}

```

配置 Shiro 过滤器时会传入一个安全管理器，可以看出，这是一环套一环，realm -> SecurityManager -> filter。在过滤器中，我们需要定义一个 shiroFactoryBean，然后将 SecurityManager 添加进来，结合上面代码可以看出，要配置的东西主要有：

默认登录的 url：身份认证失败会访问该 url 认证成功之后要跳转的 url 权限认证失败会访问该 url 需要拦截或者放行的 url：这些都放在一个 map 中

从上述代码中可以看出，在 map 中，针对不同的 url，有不同的权限要求，这里总结一下常用的几个权限。|Filter|说明| |--:|--:| |anon|开放权限，可以理解为匿名用户或游客，可以直接访问的| |authc|需要身份认证的| |logout|注销，执行后会直接跳转到 shiroFilterFactoryBean.setLoginUrl(); 设置的 url，即登录页面| |roles[admin]|参数可写多个，表示是某个或某些角色才能通过，多个参数时写 roles["admin, user"]，当有多个参数时必须每个参数都通过才算通过| |perms[user]|参数可写多个，表示需要某个或某些权限才能通过，多个参数时写 perms["user, admin"]，当有多个参数时必须每个参数都通过才算通过|

## 2.4 使用 Shiro 进行认证

到这里，我们对 Shiro 的准备工作都做完了，接下来开始使用 Shiro 进行认证工作。我们首先来设计几个接口：

接口一： 使用 http://localhost:8080/user/admin 来验证身份认证 接口二： 使用 http://localhost:8080/user/student 来验证角色认证 接口三： 使用 http://localhost:8080/user/teacher 来验证权限认证 接口四： 使用 http://localhost:8080/user/login 来实现用户登录



然后来一下认证的流程：

流程一： 直接访问接口一（此时还未登录），认证失败，跳转到 login.html 页面让用户登录，登录会请求接口四，实现用户登录功能，此时 Shiro 已经保存了用户信息了。 流程二： 再次访问接口一（此时用户已经登录），认证成功，跳转到 success.html 页面，展示用户信息。 流程三： 访问接口二，测试角色认证是否成功。 流程四： 访问接口三，测试权限认证是否成功。

#### 2.4.1 身份、角色、权限认证接口

```
@Controller@RequestMapping("/user")public class UserController {

    /**
     * 身份认证测试接口
     * @param request
     * @return
     */
    @RequestMapping("/admin")
    public String admin(HttpServletRequest request) {
        Object user = request.getSession().getAttribute("user");
        return "success";
    }

    /**
     * 角色认证测试接口
     * @param request
     * @return
     */
    @RequestMapping("/student")
    public String student(HttpServletRequest request) {
        return "success";
    }

    /**
     * 权限认证测试接口
     * @param request
     * @return
     */
    @RequestMapping("/teacher")
```

```

    public String teacher(HttpServletRequest request) {
        return "success";
    }
}

```

这三个接口很简单，直接返回到指定页面展示即可，只要认证成功就会正常跳转，如果认证失败，就会跳转到上文 ShrioConfig 中配置的页面进行展示。

#### 2.4.2 用户登录接口

```

@Controller@RequestMapping("/user")public class UserController {

    /**
     * 用户登录接口
     * @param user user
     * @param request request
     * @return string
     */
    @PostMapping("/login")
    public String login(User user, HttpServletRequest request) {

        // 根据用户名和密码创建 token
        UsernamePasswordToken token = new
UsernamePasswordToken(user.getUsername(), user.getPassword());
        // 获取 subject 认证主体
        Subject subject = SecurityUtils.getSubject();
        try{
            // 开始认证，这一步会跳到我们自定义的 realm 中
            subject.login(token);
            request.getSession().setAttribute("user", user);
            return "success";
        }catch(Exception e){
            e.printStackTrace();
            request.getSession().setAttribute("user", user);
            request.setAttribute("error", "用户名或密码错误！");
            return "login";
        }
    }
}

```

我们重点分析一下这个登录接口，首先会根据前端传过来的用户名和密码，创建一个 token，然后使用 SecurityUtils 来创建一个认证主体，接下来开始调用

`subject.login(token)` 开始进行身份认证了，注意这里传了刚刚创建的 `token`，就如注释中所述，这一步会跳转到我们自定义的 `realm` 中，进入 `doGetAuthenticationInfo` 方法，所以到这里，您就会明白该方法中那个参数 `token` 了。然后就是上文分析的那样，开始进行身份认证。

### 2.4.3 测试一下

最后，启动项目，测试一下：浏览器请求 `http://localhost:8080/user/admin` 会进行身份认证，因为此时未登录，所以会跳转到 `IndexController` 中的 `/login` 接口，然后跳转到 `login.html` 页面让我们登录，使用用户名密码为 `csdn/123456` 登录之后，我们在浏览器中请求 `http://localhost:8080/user/student` 接口，会进行角色认证，因为数据库中 `csdn1` 的用户角色是 `admin`，所以和配置中的吻合，认证通过；我们再请求 `http://localhost:8080/user/teacher` 接口，会进行权限认证，因为数据库中 `csdn1` 的用户权限为 `user:*`，满足配置中的 `user:create`，所以认证通过。

接下来，我们点退出，系统会注销重新让我们登录，我们使用 `csdn2` 这个用户来登录，重复上述操作，当在进行角色认证和权限认证这两步时，就认证不通过了，因为数据库中 `csdn2` 这个用户存的角色和权限与配置中的不同，所以认证不通过。

## 3. 总结

本节主要介绍了 `Shiro` 安全框架与 `Spring Boot` 的整合。先介绍了 `Shiro` 的三大核心组件已经它们的作用；然后介绍了 `Shiro` 的身份认证、角色认证和权限认证；最后结合代码，详细介绍了 `Spring Boot` 中是如何整合 `Shiro` 的，并设计了一套测试流程，逐步分析 `Shiro` 的工作流程和原理，让读者更直观地体会出 `Shiro` 的整套工作流程。`Shiro` 使用的很广泛，希望读者将其掌握，并能运用到实际项目中。

课程源代码下载地址：[戳我下载](#)

## 第 17 课：Spring Boot 中集成 Lucence

### 1. Lucence 和全文检索

Lucene 是什么？看一下百度百科：

Lucene 是一套用于全文检索和搜寻的开源程式库，由 Apache 软件基金会支持和提供。Lucene 提供了一个简单却强大的应用程序接口，能够做全文索引和搜寻。在 Java 开发环境里 Lucene 是一个成熟的免费开源工具。就其本身而言，Lucene 是当前以及最近几年最受欢迎的免费 Java 信息检索程序库。——《百度百科》

#### 1.1 全文检索

这里提到了全文检索的概念，我们先来分析一下什么是全文检索，理解了全文检索之后，再理解 Lucene 的原理就非常简单了。

何为全文检索？举个例子，比如现在要在一个文件中查找某个字符串，最直接的想法就是从头开始检索，查到了就 OK，这种对于小数据量的文件来说，很实用，但是对于大数据量的文件来说，就有点吃力了。或者说找包含某个字符串的文件，也是这样，如果在一个拥有几十个 G 的硬盘中找那效率可想而知，是很低的。

文件中的数据是属于非结构化数据，也就是说它没有什么结构可言，要解决上面提到的效率问题，首先我们得将非结构化数据中的一部分信息提取出来，重新组织，使其变得有一定结构，然后对这些有一定结构的数据进行搜索，从而达到搜索相对较快的目的。这就叫全文搜索。即先建立索引，再对索引进行搜索的过程。

## 1.2 Lucene 建立索引的方式

那么 Lucene 中是如何建立索引的呢？假设现在有两篇文章，内容如下：

文章 1 的内容为：Tom lives in Guangzhou, I live in Guangzhou too.

文章 2 的内容为：He once lived in Shanghai.

首先第一步是将文档传给分词组件（Tokenizer），分词组件会将文档分成一个个单词，并去除标点符号和停词。所谓的停词指的是没有特别意义的词，比如英文中的 a, the, too 等。经过分词后，得到词元（Token）。如下：

文章 1 经过分词后的结果：[Tom] [lives] [Guangzhou] [I] [live] [Guangzhou]

文章 2 经过分词后的结果：[He] [lives] [Shanghai]

然后将词元传给语言处理组件（Linguistic Processor），对于英语，语言处理组件一般会将字母变为小写，将单词缩减为词根形式，如 "lives" 到 "live" 等，将单词转变为词根形式，如 "drove" 到 "drive" 等。然后得到词（Term）。如下：

文章 1 经过处理后的结果：[tom] [live] [guangzhou] [i] [live] [guangzhou] 文章 2 经过处理后的结果：[he] [live] [shanghai]

最后将得到的词传给索引组件（Indexer），索引组件经过处理，得到下面的索引结构：

关键词	文章号[出现频率]	出现位置
guangzhou	1[2]	3,6
he	2[1]	1
i	1[1]	4
live	1[2],2[1]	2,5,2
shanghai	2[1]	3

关键词	文章号[出现频率]	出现位置
tom	1[1]	1

以上就是 Lucene 索引结构中最核心的部分。它的关键字是按字符顺序排列的，因此 Lucene 可以用二元搜索算法快速定位关键词。实现时 Lucene 将上面三列分别作为词典文件（Term Dictionary）、频率文件（frequencies）和位置文件（positions）保存。其中词典文件不仅保存有每个关键词，还保留了指向频率文件和位置文件的指针，通过指针可以找到该关键字的频率信息和位置信息。搜索的过程是先对词典二元查找、找到该词，通过指向频率文件的指针读出所有文章号，然后返回结果，然后就可以在具体的文章中根据出现位置找到该词了。所以 Lucene 在第一次建立索引的时候可能会比较慢，但是以后就不需要每次都建立索引了，就快了。

理解了 Lucene 的分词原理，接下来我们在 Spring Boot 中集成 Lucene 并实现索引和搜索的功能。

## 2. Spring Boot 中集成 Lucence

### 2.1 依赖导入

首先需要导入 Lucene 的依赖，它的依赖有好几个，如下：

```
<!-- Lucence 核心包 --><dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-core</artifactId>
    <version>5.3.1</version></dependency>
<!-- Lucene 查询解析包 --><dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-queryparser</artifactId>
    <version>5.3.1</version></dependency>
<!-- 常规的分词（英文） --><dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-analyzers-common</artifactId>
    <version>5.3.1</version></dependency>
<!--支持分词高亮 --><dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-highlighter</artifactId>
    <version>5.3.1</version></dependency>
<!--支持中文分词 --><dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-analyzers-smartcn</artifactId>
```

```
<version>5.3.1</version></dependency>
```

最后一个依赖是用来支持中文分词的，因为默认是支持英文的。那个高亮的分词依赖是最后我要做一个搜索，然后将搜到的内容高亮显示，模拟当前互联网上的做法，大家可以运用到实际项目中去。

## 2.2 快速入门

根据上文的分析，全文检索有两个步骤，先建立索引，再检索。所以为了测试这个过程，我新建两个 `java` 类，一个用来建立索引的，另一个用来检索。

### 2.2.1 建立索引

我们自己弄几个文件，放到 `D:\lucene\data` 目录下，新建一个 `Indexer` 类来实现建立索引功能。首先在构造方法中初始化标准分词器和写索引实例。

```
public class Indexer {

    /**
     * 写索引实例
     */
    private IndexWriter writer;

    /**
     * 构造方法，实例化 IndexWriter
     * @param indexDir
     * @throws Exception
     */
    public Indexer(String indexDir) throws Exception {
        Directory dir = FSDirectory.open(Paths.get(indexDir));
        //标准分词器，会自动去掉空格啊，is a the 等单词
        Analyzer analyzer = new StandardAnalyzer();
        //将标准分词器配到写索引的配置中
        IndexWriterConfig config = new IndexWriterConfig(analyzer);
        //实例化写索引对象
        writer = new IndexWriter(dir, config);
    }
}
```

在构造方法中传一个存放索引的文件夹路径，然后构建标准分词器（这是英文的），再使用标准分词器来实例化写索引对象。接下来就开始建立索引了，我将解释放到代码注释里，方便大家跟进。

```

/**
 * 索引指定目录下的所有文件
 * @param dataDir
 * @return
 * @throws Exception
 */public int indexAll(String dataDir) throws Exception {
    // 获取该路径下的所有文件
    File[] files = new File(dataDir).listFiles();
    if (null != files) {
        for (File file : files) {
            //调用下面的 indexFile 方法，对每个文件进行索引
            indexFile(file);
        }
    }
    //返回索引的文件数
    return writer.numDocs();
}

/**
 * 索引指定的文件
 * @param file
 * @throws Exception
 */private void indexFile(File file) throws Exception {
    System.out.println("索引文件的路径: " + file.getCanonicalPath());
    //调用下面的 getDocument 方法，获取该文件的 document
    Document doc = getDocument(file);
    //将 doc 添加到索引中
    writer.addDocument(doc);
}

/**
 * 获取文档，文档里再设置每个字段，就类似于数据库中的一行记录
 * @param file
 * @return
 * @throws Exception
 */private Document getDocument(File file) throws Exception {
    Document doc = new Document();
    //开始添加字段
    //添加内容
    doc.add(new TextField("contents", new FileReader(file)));
    //添加文件名，并把这个字段存到索引文件里

```

```

        doc.add(new TextField("fileName", file.getName(),
Field.Store.YES));
        //添加文件路径
        doc.add(new TextField("fullPath", file.getCanonicalPath(),
Field.Store.YES));
        return doc;
    }

```

这样就建立好索引了，我们在该类中写一个 main 方法测试一下：

```

public static void main(String[] args) {
    //索引保存到的路径
    String indexDir = "D:\\\\lucene";
    //需要索引的文件数据存放的目录
    String dataDir = "D:\\\\lucene\\data";
    Indexer indexer = null;
    int indexedNum = 0;
    //记录索引开始时间
    long startTime = System.currentTimeMillis();
    try {
        // 开始构建索引
        indexer = new Indexer(indexDir);
        indexedNum = indexer.indexAll(dataDir);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            if (null != indexer) {
                indexer.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //记录索引结束时间
    long endTime = System.currentTimeMillis();
    System.out.println("索引耗时" + (endTime - startTime) + "毫秒");
    System.out.println("共索引了" + indexedNum + "个文件");
}

```



我搞了两个 tomcat 相关的文件放到 D:\lucene\data 下了，执行完之后，看到控制台输出：

索引文件的路径：D:\lucene\data\catalina.properties

索引文件的路径：D:\lucene\data\logging.properties

索引耗时 882 毫秒

共索引了 2 个文件

然后我们去 D:\lucene\ 目录下可以看到一些索引文件，这些文件不能删除，删除了就需要重新构建索引，否则没了索引，就无法去检索内容了。

## 2.2.2 检索内容

上面把这两个文件的索引建立好了，接下来我们就可以写检索程序了，在这两个文件中查找特定的词。

```
public class Searcher {  
  
    public static void search(String indexDir, String q) throws Exception {  
  
        //获取要查询的路径，也就是索引所在的位置  
        Directory dir = FSDirectory.open(Paths.get(indexDir));  
        IndexReader reader = DirectoryReader.open(dir);  
        //构建 IndexSearcher  
        IndexSearcher searcher = new IndexSearcher(reader);  
        //标准分词器，会自动去掉空格啊，is a the 等单词  
        Analyzer analyzer = new StandardAnalyzer();  
        //查询解析器  
        QueryParser parser = new QueryParser("contents", analyzer);  
        //通过解析要查询的 String，获取查询对象，q 为传进来的待查的字符串  
        Query query = parser.parse(q);  
  
        //记录索引开始时间  
        long startTime = System.currentTimeMillis();  
        //开始查询，查询前 10 条数据，将记录保存在 docs 中  
        TopDocs docs = searcher.search(query, 10);  
        //记录索引结束时间  
        long endTime = System.currentTimeMillis();  
        System.out.println("匹配" + q + "共耗时" + (endTime-startTime)  
+ "毫秒");  
        System.out.println("查询到" + docs.totalHits + "条记录");  
    }  
}
```

```

        //取出每条查询结果
        for(ScoreDoc scoreDoc : docs.scoreDocs) {
            //scoreDoc.doc 相当于 docID,根据这个 docID 来获取文档
            Document doc = searcher.doc(scoreDoc.doc);
            //fullPath 是刚刚建立索引的时候我们定义的一个字段，表示路径。也可以取其他的内容，只要我们在建立索引时有定义即可。
            System.out.println(doc.get("fullPath"));
        }
        reader.close();
    }
}

```

ok，这样我们检索的代码就写完了，每一步解释我写在代码中的注释上了，下面写个 main 方法来测试一下：

```

public static void main(String[] args) {
    String indexDir = "D:\\lucene";
    //查询这个字符串
    String q = "security";
    try {
        search(indexDir, q);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

查一下 security 这个字符串，执行一下看控制台打印的结果：

```

匹配 security 共耗时 23 毫秒
查询到 1 条记录 D:\lucene\data\catalina.properties

```

可以看出，耗时了 23 毫秒在两个文件中找到了 security 这个字符串，并输出了文件的名称。上面的代码我写的很详细，这个代码已经比较全了，可以用在生产环境上。

## 2.3 中文分词检索高亮实战

上文已经写了建立索引和检索的代码，但是在实际项目中，我们往往是结合页面做一些查询结果的展示，比如我要查某个关键字，查到了之后，将相关的信息点展示出来，并将查询的关键字高亮等等。这种需求在实际项目中非常常见，而且大多数网站中都会有这种效果。所以这一小节我们就使用 Lucene 来实现这种效果。

### 2.3.1 中文分词

我们新建一个 `ChineseIndexer` 类来建立中文索引，建立过程和英文索引一样的，不同的地方在于使用的是中文分词器。除此之外，这里我们不用通过读取文件去建立索引，我们模拟一下用字符串来建立，因为在实际项目中，绝大部分情况是获取到一些文本字符串，然后根据一些关键字去查询相关内容等等。代码如下：

```
public class ChineseIndexer {

    /**
     * 存放索引的位置
     */
    private Directory dir;

    //准备一下用来测试的数据
    //用来标识文档
    private Integer ids[] = {1, 2, 3};
    private String citys[] = {"上海", "南京", "青岛"};
    private String desc[] = {
        "上海是个繁华的城市。",
        "南京是一个文化的城市南京，简称宁，是江苏省会，地处中国东部地区，长江下游，濒江近海。全市下辖 11 个区，总面积 6597 平方公里，2013 年建成区面积 752.83 平方公里，常住人口 818.78 万，其中城镇人口 659.1 万人。[1-4] “江南佳丽地，金陵帝王州”，南京拥有着 6000 多年文明史、近 2600 年建城史和近 500 年的建都史，是中国四大古都之一，有“六朝古都”、“十朝都会”之称，是中华文明的重要发祥地，历史上曾数次庇佑华夏之正朔，长期是中国南方的政治、经济、文化中心，拥有厚重的文化底蕴和丰富的历史遗存。[5-7] 南京是国家重要的科教中心，自古以来就是一座崇文重教的城市，有“天下文枢”、“东南第一学”的美誉。截至 2013 年，南京有高等院校 75 所，其中 211 高校 8 所，仅次于北京上海；国家重点实验室 25 所、国家重点学科 169 个、两院院士 83 人，均居中国第三。[8-10] 。",
        "青岛是一个美丽的城市。"
    };

    /**
     * 生成索引
     * @param indexDir
     * @throws Exception
     */
    public void index(String indexDir) throws Exception {
        dir = FSDirectory.open(Paths.get(indexDir));
        // 先调用 getWriter 获取 IndexWriter 对象
    }
}
```

```

        IndexWriter writer = getWriter();
        for(int i = 0; i < ids.length; i++) {
            Document doc = new Document();
            // 把上面的数据都生成索引，分别用 id、city 和 desc 来标识
            doc.add(new IntField("id", ids[i], Field.Store.YES));
            doc.add(new StringField("city", citys[i],
Field.Store.YES));
            doc.add(new TextField("desc", desc[i], Field.Store.YES));
            //添加文档
            writer.addDocument(doc);
        }
        //close 了才真正写到文档中
        writer.close();
    }

    /**
     * 获取 IndexWriter 实例
     * @return
     * @throws Exception
     */
    private IndexWriter getWriter() throws Exception {
        //使用中文分词器
        SmartChineseAnalyzer analyzer = new SmartChineseAnalyzer();
        //将中文分词器配到写索引的配置中
        IndexWriterConfig config = new IndexWriterConfig(analyzer);
        //实例化写索引对象
        IndexWriter writer = new IndexWriter(dir, config);
        return writer;
    }

    public static void main(String[] args) throws Exception {
        new ChineseIndexer().index("D:\\lucene2");
    }
}

```

这里我们用 id、city、desc 分别代表 id、城市名称和城市描述，用他们作为关键字来建立索引，后面我们获取内容的时候，主要来获取城市描述。南京的描述我故意写的长一点，因为下文检索的时候，根据不同的关键字会检索到不同部分的信息，有个权重的概念在里面。然后执行一下 main 方法，将索引保存到 D:\lucene2\ 中。

### 2.3.2 中文分词查询

中文分词查询代码逻辑和默认的查询差不多，有一些区别在于，我们需要将查询出来的关键字标红加粗等需要处理，需要计算出一个得分片段，这是什么意思呢？比如我搜索“南京文化”跟搜索“南京文明”，这两个搜索结果应该根据关键字出现的位置，返回的结果不一样才对，这在下文会测试。我们先看一下代码和注释：

```
public class ChineseSearch {

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

    public static List<String> search(String indexDir, String q) throws
        Exception {

        //获取要查询的路径，也就是索引所在的位置
        Directory dir = FSDirectory.open(Paths.get(indexDir));
        IndexReader reader = DirectoryReader.open(dir);
        IndexSearcher searcher = new IndexSearcher(reader);
        //使用中文分词器
        SmartChineseAnalyzer analyzer = new SmartChineseAnalyzer();
        //由中文分词器初始化查询解析器
        QueryParser parser = new QueryParser("desc", analyzer);
        //通过解析要查询的 String，获取查询对象
        Query query = parser.parse(q);

        //记录索引开始时间
        long startTime = System.currentTimeMillis();
        //开始查询，查询前 10 条数据，将记录保存在 docs 中
        TopDocs docs = searcher.search(query, 10);
        //记录索引结束时间
        long endTime = System.currentTimeMillis();
        logger.info("匹配{}共耗时{}毫秒", q, (endTime - startTime));
        logger.info("查询到{}条记录", docs.totalHits);

        //如果不指定参数的话，默认是加粗，即<b><b/>
        SimpleHTMLFormatter simpleHTMLFormatter = new
        SimpleHTMLFormatter("<b><font color=red>", "</font></b>");
        //根据查询对象计算得分，会初始化一个查询结果最高的得分
        QueryScorer scorer = new QueryScorer(query);
        //根据这个得分计算出一个片段
        Fragmenter fragmenter = new SimpleSpanFragmenter(scorer);
        //将这个片段中的关键字用上面初始化好的高亮格式高亮
```

```

        Highlighter highlighter = new
Highlighter(simpleHTMLFormatter, scorer);
        //设置一下要显示的片段
        highlighter.setTextFragmenter(fragmenter);

        //取出每条查询结果
        List<String> list = new ArrayList<>();
        for(ScoreDoc scoreDoc : docs.scoreDocs) {
            //scoreDoc.doc 相当于 docID,根据这个 docID 来获取文档
            Document doc = searcher.doc(scoreDoc.doc);
            logger.info("city:{", doc.get("city"));
            logger.info("desc:{", doc.get("desc"));
            String desc = doc.get("desc");

            //显示高亮
            if(desc != null) {
                TokenStream tokenStream = analyzer.tokenStream("desc",
new StringReader(desc));
                String summary =
highlighter.getBestFragment(tokenStream, desc);
                logger.info("高亮后的 desc:{", summary);
                list.add(summary);
            }
        }
        reader.close();
        return list;
    }
}

```

每一步的注释我写的很详细，在这就不赘述了。接下来我们来测试一下效果。

### 2.3.3 测试一下

这里我们使用 **thymeleaf** 来写个简单的页面来展示获取到的数据，并高亮展示。在 **controller** 中我们指定索引的目录和需要查询的字符串，如下：

```

@Controller
@RequestMapping("/lucene")public class IndexController {

    @GetMapping("/test")
    public String test(Model model) {
        // 索引所在的目录
        String indexDir = "D:\\lucene2";
    }
}

```

```

// 要查询的字符//      String q = "南京文明";
String q = "南京文化";
try {
    List<String> list = ChineseSearch.search(indexDir, q);
    model.addAttribute("list", list);
} catch (Exception e) {
    e.printStackTrace();
}
return "result";
}
}

```

直接返回到 result.html 页面，该页面主要来展示一下 model 中的数据即可。

```

<!DOCTYPE html><html lang="en"
xmlns:th="http://www.thymeleaf.org"><head>
    <meta charset="UTF-8">
    <title>Title</title></head><body><div th:each="desc : ${list}">
    <div th:utext="${desc}"></div></div></body></html>

```

这里注意一下，不能使用 th:test，否则字符串中的 html 标签都会被转义，不会被渲染到页面。下面启动服务，在浏览器中输入  
http://localhost:8080/lucene/test，测试一下效果，我们搜索的是“南京文化”。

南京是一个文化的城市南京，简称宁，是江苏省会，地处中国东部地区，长江下游，濒江近海。全市下辖11个区，总面积6597平方公里，2013年建成区面积752.83平方公里，常住人口818.78万，其中

再将 controller 中的搜索关键字改成“南京文明”，看下命中的效果。

城镇人口659.1万人。[1-4] “江南佳丽地，金陵帝王州”，南京拥有着6000多年文明史、近2600年建城史和近500年的建都史，是中国四大古都之一，有“六朝古都”、“十朝都会”之称，是中华文明的

可以看出，不同的关键词，它会计算一个得分片段，也就是说不同的关键字会命中不同位置的内容，然后将关键字根据我们自己设定的形式高亮显示。从结果中可以看出，Lucene 也可以很智能的将关键字拆分命中，这在实际项目中会很好用。

### 3. 总结

本节课首先详细的分析了全文检索的理论规则，然后结合 **Lucene**，系统的讲述了在 **Spring Boot** 的集成步骤，首先快速带领大家从直观上感受 **Lucene** 如何建立索引已经如果检索，其次通过中文检索的具体实例，展示了 **Lucene** 在全文检索中的广泛应用。**Lucene** 不难，主要就是步骤比较多，代码不用死记硬背，拿到项目中根据实际情况做对应的修改即可。

课程源代码下载地址：[戳我下载](#)

## 第 18 课：Spring Boot 搭建实际项目开发中的架构

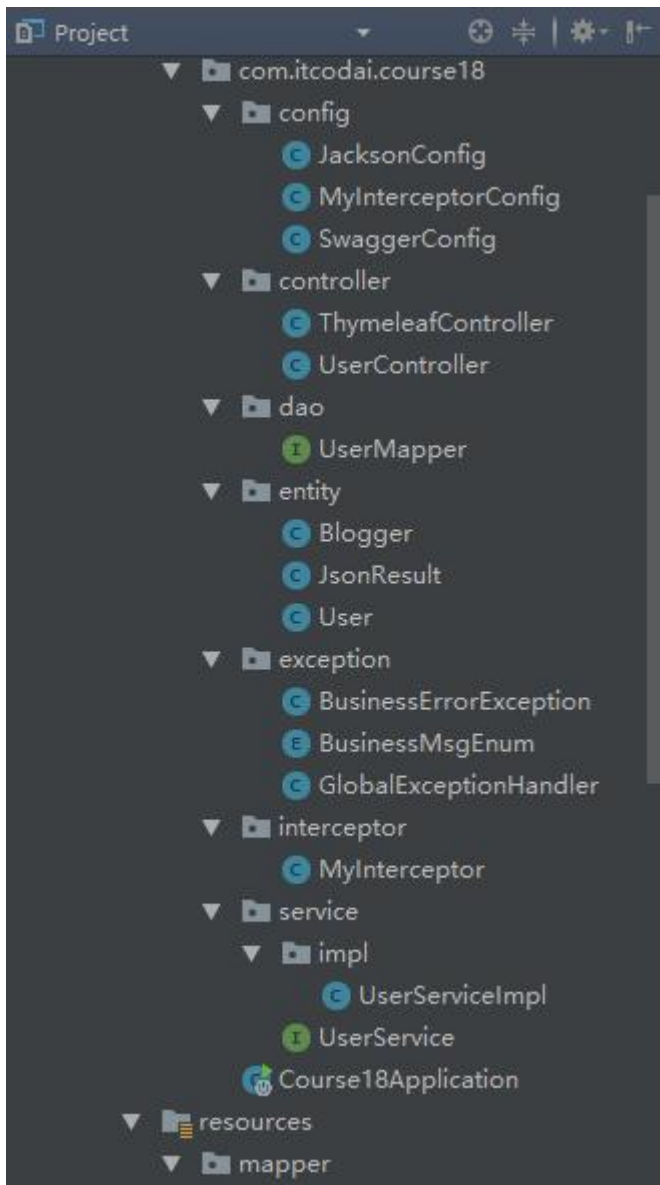
前面的课程中，我主要给大家讲解了 **Spring Boot** 中常用的一些技术点，这些技术点在实际项目中可能不会全部用得到，因为不同的项目可能使用的技术不同，但是希望大家都能掌握如何使用，并能自己根据实际项目中的需求进行相应的扩展。

不知道大家了不了解单片机，单片机里有个最小系统，这个最小系统搭建好了之后，就可以在此基础上进行人为的扩展。这节课我们要做的就是搭建一个“**Spring Boot** 最小系统架构”。拿着这个架构，可以在在此基础上根据实际需求做相应的扩展。

从零开始搭建一个环境，主要要考虑几点：统一封装的数据结构、可调式的接口、**json** 的处理、模板引擎的使用（本文不写该项，因为现在大部分项目都前后端分离了，但是考虑到也还有非前后端分离的项目，所以我在源代码里也加上了 **thymeleaf**）、持久层的集成、拦截器（这个也是可选的）和全局异常处理。一般包括这些东西的话，基本上一个 **Spring Boot** 项目环境就差不多了，然后就是根据具体情况来扩展了。

结合前面的课程和以上的这些点，本节课手把手带领大家搭建一个实际项目开发中可用的 **Spring Boot** 架构。整个项目工程如下图所示，学习的时候，可以结合我的源码，这样效果会更好。





## 1. 统一的数据封装

由于封装的 json 数据的类型不确定，所以在定义统一的 json 结构时，我们需要用到泛型。统一的 json 结构中属性包括数据、状态码、提示信息即可，构造方法可以根据实际业务需求做相应的添加即可，一般来说，应该有默认返回结构，也应该有用户指定的返回结构。如下：

```
/**
 * 统一返回对象
 * @author shengwu ni
 * @param <T>
 */public class JsonResult<T> {

    private T data;
```

```

private String code;
private String msg;

/**
 * 若没有数据返回，默认状态码为 0，提示信息为：操作成功！
 */
public JsonResult() {
    this.code = "0";
    this.msg = "操作成功！";
}

/**
 * 若没有数据返回，可以人为指定状态码和提示信息
 * @param code
 * @param msg
 */
public JsonResult(String code, String msg) {
    this.code = code;
    this.msg = msg;
}

/**
 * 有数据返回时，状态码为 0，默认提示信息为：操作成功！
 * @param data
 */
public JsonResult(T data) {
    this.data = data;
    this.code = "0";
    this.msg = "操作成功！";
}

/**
 * 有数据返回，状态码为 0，人为指定提示信息
 * @param data
 * @param msg
 */
public JsonResult(T data, String msg) {
    this.data = data;
    this.code = "0";
}

```

```

        this.msg = msg;
    }

    /**
     * 使用自定义异常作为参数传递状态码和提示信息
     * @param msgEnum
     */
    public JsonResult(BusinessMsgEnum msgEnum) {
        this.code = msgEnum.code();
        this.msg = msgEnum.msg();
    }

    // 省去 get 和 set 方法
}

```

大家可以根据自己项目中所需要的一些东西，合理的修改统一结构中的字段信息。

## 2. json 的处理

Json 处理工具很多，比如阿里巴巴的 **fastjson**，不过 **fastjson** 对有些未知类型的 **null** 无法转成空字符串，这可能是 **fastjson** 自身的缺陷，可扩展性也不是太好，但是使用起来方便，使用的人也蛮多的。这节课里面我们主要集成 **Spring Boot** 自带的 **jackson**。主要是对 **jackson** 做一下对 **null** 的配置即可，然后就可以在项目中使用了。

```

/**
 * jacksonConfig
 * @author shengwu ni
 */
@Configuration
public class JacksonConfig {
    @Bean
    @Primary
    @ConditionalOnMissingBean(ObjectMapper.class)
    public ObjectMapper
jacksonObjectMapper(Jackson2ObjectMapperBuilder builder) {
        ObjectMapper objectMapper =
builder.createXmlMapper(false).build();

        objectMapper.getSerializerProvider().setNullValueSerializer(new
JsonSerializer<Object>() {
            @Override
            public void serialize(Object o, JsonGenerator jsonGenerator,
SerializerProvider serializerProvider) throws IOException {

```

```

        jsonGenerator.writeString("");
    }
});
return objectMapper;
}
}

```

这里先不测试，等下面 **swagger2** 配置好了之后，我们一起来测试一下。

### 3. swagger2 在线可调式接口

有了 **swagger**，开发人员不需要给其他人员提供接口文档，只要告诉他们一个 **Swagger** 地址，即可展示在线的 **API** 接口文档，除此之外，调用接口的人员还可以在线测试接口数据，同样地，开发人员在开发接口时，同样也可以利用 **Swagger** 在线接口文档测试接口数据，这给开发人员提供了便利。使用 **swagger** 需要对其进行配置：

```

/**
 * swagger 配置
 * @author shengwu ni
 */
@Configuration@EnableSwagger2public class SwaggerConfig {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            // 指定构建 api 文档的详细信息的方法: apiInfo()
            .apiInfo(apiInfo())
            .select()
            // 指定要生成 api 接口的包路径，这里把 controller 作为包路
            // 径，生成 controller 中的所有接口
            .apis(RequestHandlerSelectors.basePackage("com.itcodai.
course18.controller"))
            .paths(PathSelectors.any())
            .build();
    }

    /**
     * 构建 api 文档的详细信息
     * @return
     */
    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
            // 设置页面标题

```

```

        .title("Spring Boot 搭建实际项目中开发的架构")
        // 设置接口描述
        .description("跟武哥一起学 Spring Boot 第 18 课")
        // 设置联系方式
        .contact("倪升武, " + "微信公众号: 程序员私房菜")
        // 设置版本
        .version("1.0")
        // 构建
        .build();
    }
}

```

到这里，可以先测试一下，写一个 Controller，弄一个静态的接口测试一下上面集成的内容。

```

@RestController@Api(value = "用户信息接口")public class UserController {

    @Resource
    private UserService userService;

    @GetMapping("/getUser/{id}")
    @ApiOperation(value = "根据用户唯一标识获取用户信息")
    public JsonResult<User> getUserInfo(@PathVariable @ApiParam(value = "用户唯一标识") Long id) {
        User user = new User(id, "倪升武", "123456");
        return new JsonResult<>(user);
    }
}

```

然后启动项目，在浏览器中输入 `localhost:8080/swagger-ui.html` 即可看到 `swagger` 接口文档页面，调用一下上面这个接口，即可看到返回的 `json` 数据。

## 4. 持久层集成

每个项目中是必须要有持久层的，与数据库交互，这里我们主要来集成 `mybatis`，集成 `mybatis` 首先要在 `application.yml` 中进行配置。

```

# 服务端口号
server:
  port: 8080
# 数据库地址

```

```

datasource:
  url: localhost:3306/blog_test

spring:
  datasource: # 数据库配置
    driver-class-name: com.mysql.jdbc.Driver
    url:
jdbc:mysql://${datasource.url}?useSSL=false&useUnicode=true&characterEncoding=utf-8&allowMultiQueries=true&autoReconnect=true&failOverReadOnly=false&maxReconnects=10
    username: root
    password: 123456
    hikari:
      maximum-pool-size: 10 # 最大连接池数
      max-lifetime: 1770000

mybatis:
  # 指定别名设置的包为所有 entity
  type-aliases-package: com.itcodai.course18.entity
  configuration:
    map-underscore-to-camel-case: true # 驼峰命名规范
  mapper-locations: # mapper 映射文件位置
    - classpath:mapper/*.xml

```

配置好了之后，接下来我们来写一下 dao 层，实际中我们使用注解比较多，因为比较方便，当然也可以使用 xml 的方式，甚至两种同时使用都行，这里我们主要使用注解的方式来集成，关于 xml 的方式，大家可以查看前面课程，实际中根据项目情况来定。

```

public interface UserMapper {

    @Select("select * from user where id = #{id}")
    @Results({
        @Result(property = "username", column = "user_name"),
        @Result(property = "password", column = "password")
    })
    User getUser(Long id);

    @Select("select * from user where id = #{id} and user_name=#{name}")
    User getUserByIdAndName(@Param("id") Long id, @Param("name")
String username);

    @Select("select * from user")

```

```
List<User> getAll();  
}
```

关于 service 层我就不再文章中写代码了，大家可以结合我的源代码学习，这一节主要带领大家来搭建一个 Spring Boot 空架构。最后别忘了在启动类上添加注解扫描 `@MapperScan("com.itcodai.course18.dao")`

## 5. 拦截器

拦截器在项目中使用的是非常多的（但不是绝对的），比如拦截一些置顶的 url，做一些判断和处理等等。除此之外，还需要将常用的静态页面或者 swagger 页面放行，不能将这些静态资源给拦截了。首先先自定义一个拦截器。

```
public class MyInterceptor implements HandlerInterceptor {  
  
    private static final Logger logger =  
        LoggerFactory.getLogger(MyInterceptor.class);  
  
    @Override  
    public boolean preHandle(HttpServletRequest request,  
        HttpServletResponse response, Object handler) throws Exception {  
  
        logger.info("执行方法之前执行(Controller 方法调用之前)");  
        return true;  
    }  
  
    @Override  
    public void postHandle(HttpServletRequest request,  
        HttpServletResponse response, Object handler, ModelAndView  
        modelAndView) throws Exception {  
  
        logger.info("执行完方法之后进执行(Controller 方法调用之后)，但是此时  
        还没进行视图渲染");  
    }  
  
    @Override  
    public void afterCompletion(HttpServletRequest request,  
        HttpServletResponse response, Object handler, Exception ex) throws  
        Exception {  
  
        logger.info("整个请求都处理完咯，DispatcherServlet 也渲染了对应的  
        视图咯，此时我可以做一些清理的工作了");  
    }  
}
```

然后将自定义的拦截器加入到拦截器配置中。

```
@Configurationpublic class MyInterceptorConfig implements
WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 实现 WebMvcConfigurer 不会导致静态资源被拦截
        registry.addInterceptor(new MyInterceptor())
            // 拦截所有 url
            .addPathPatterns("/**")
            // 放行 swagger
            .excludePathPatterns("/swagger-resources/**");
    }
}
```

在 Spring Boot 中，我们通常会在如下目录里存放一些静态资源：

```
classpath:/static
classpath:/public
classpath:/resources
classpath:/META-INF/resources
```

上面代码中配置的 `/**` 是对所有 url 都进行了拦截，但我们实现了 `WebMvcConfigurer` 接口，不会导致 Spring Boot 对上面这些目录下的静态资源实施拦截。但是我们平时访问的 `swagger` 会被拦截，所以要将其放行。`swagger` 页面在 `swagger-resources` 目录下，放行该目录下所有文件即可。

然后在浏览器中输入一下 `swagger` 页面，若能正常显示 `swagger`，说明放行成功。同时可以根据后台打印的日志判断代码执行的顺序。

## 6. 全局异常处理

全局异常处理是每个项目中必须用到的东西，在具体的异常中，我们可能会做具体的处理，但是对于没有处理的异常，一般会有一个统一的全局异常处理。在异常处理之前，最好维护一个异常提示信息枚举类，专门用来保存异常提示信息的。如下：

```
public enum BusinessMsgEnum {
    /** 参数异常 */
    PARAMETER_EXCEPTION("102", "参数异常!"),
    /** 等待超时 */
    SERVICE_TIME_OUT("103", "服务调用超时!"),
    /** 参数过大 */
}
```



```

    PARMETER_BIG_EXCEPTION("102", "输入的图片数量不能超过 50 张!"),
    /** 500 : 发生异常 */
    UNEXPECTED_EXCEPTION("500", "系统发生异常，请联系管理员!");

    /**
     * 消息码
     */
    private String code;
    /**
     * 消息内容
     */
    private String msg;

    private BusinessMsgEnum(String code, String msg) {
        this.code = code;
        this.msg = msg;
    }

    public String code() {
        return code;
    }

    public String msg() {
        return msg;
    }
}

```

在全局统一异常处理类中，我们一般会对自定义的业务异常最先处理，然后去处理一些常见的系统异常，最后会来一个一劳永逸（Exception 异常）。

```

@ControllerAdvice@ResponseBodypublic class GlobalExceptionHandler {

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

    /**
     * 拦截业务异常，返回业务异常信息
     * @param ex
     * @return
     */
}

```

```

    @ExceptionHandler(BusinessErrorException.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public JsonResult handleBusinessError(BusinessErrorException ex) {
        String code = ex.getCode();
        String message = ex.getMessage();
        return new JsonResult(code, message);
    }

    /**
     * 空指针异常
     * @param ex NullPointerException
     * @return
     */
    @ExceptionHandler(NullPointerException.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public JsonResult handleTypeMismatchException(NullPointerException
ex) {
        logger.error("空指针异常，{}", ex.getMessage());
        return new JsonResult("500", "空指针异常了");
    }

    /**
     * 系统异常 预期以外异常
     * @param ex
     * @return
     */
    @ExceptionHandler(Exception.class)
    @ResponseStatus(value = HttpStatus.INTERNAL_SERVER_ERROR)
    public JsonResult handleUnexpectedServer(Exception ex) {
        logger.error("系统异常：", ex);
        return new JsonResult(BusinessMsgEnum.UNEXPECTED_EXCEPTION);
    }
}

```

其中，`BusinessErrorException` 是自定义的业务异常，继承一下 `RuntimeException` 即可，具体可以看我的源代码，文章中就不贴代码了。在 `UserController` 中有个 `testException` 方法，用来测试全局异常的，打开 `swagger` 页面，调用一下该接口，可以看出返回用户提示信息：“系统发生异常，请联系管理员！”。当然了，实际情况中，需要根据不同的业务提示不同的信息。

## 7. 总结

本文主要手把手带领大家快速搭建一个项目中可以使用的 **Spring Boot** 空架构，主要从统一封装的数据结构、可调式的接口、json 的处理、模板引擎的使用（代码中体现）、持久层的集成、拦截器和全局异常处理。一般包括这些东西的话，基本上一个 **Spring Boot** 项目环境就差不多了，然后就是根据具体情况来扩展了。

课程源代码下载地址：[戳我下载](#)