

Spring Boot2 教程

江南一点雨



云
众
启
明
·
理
·
·
用

2020-1-13

www.javaboy.org
公众号：江南一点雨

在 Spring Boot 项目中，正常来说是不存在 XML 配置，这是因为 Spring Boot 不推荐使用 XML，注意，并非不支持，Spring Boot 推荐开发者使用 Java 配置来搭建框架，Spring Boot 中，大量的自动化配置都是通过 Java 配置来实现的，这一套实现方案，我们也可以自己做，即自己也可以使用纯 Java 来搭建一个 SSM 环境，即在项目中，不存在任何 XML 配置，包括 web.xml。

环境要求：

- 使用纯 Java 来搭建 SSM 环境，要求 Tomcat 的版本必须在 7 以上。

快速体验

1 创建工程

创建一个普通的 Maven 工程（注意，这里可以不必创建 Web 工程），并添加 SpringMVC 的依赖，同时，这里环境的搭建需要用到 Servlet，所以我们还需要引入 Servlet 的依赖（一定不能使用低版本的 Servlet），最终的 pom.xml 文件如下：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.6.RELEASE</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
    <scope>provided</scope>
</dependency>
```

2 添加 Spring 配置

工程创建成功之后，首先添加 Spring 的配置文件，如下：

```
@Configuration
@ComponentScan(basePackages = "org.javaboy", useDefaultFilters = true,
excludeFilters = {@ComponentScan.Filter(type = FilterType.ANNOTATION, classes =
Controller.class)})
public class SpringConfig {
```

关于这个配置，我说如下几点：

- @Configuration 注解表示这是一个配置类，在我们这里，这个配置的作用类似于 applicationContext.xml
- @ComponentScan 注解表示配置包扫描，里边的属性和 xml 配置中的属性都是一一对应的，useDefaultFilters 表示使用默认的过滤器，然后又除去 Controller 注解，即在 Spring 容器中扫描除了 Controller 之外的其他所有 Bean。

3 添加 SpringMVC 配置

接下来再来创建 springmvc 的配置文件：

```
@Configuration  
@ComponentScan(basePackages = "org.javaboy",useDefaultFilters =  
false,includeFilters = {@ComponentScan.Filter(type =  
FilterType.ANNOTATION,classes = Controller.class)})  
public class SpringMVConfig {  
}
```

注意，如果不需要在 SpringMVC 中添加其他的额外配置，这样就可以了。即 视图解析器、JSON 解析、文件上传……等等，如果都不需要配置的话，这样就可以了。

4 配置 web.xml

此时，我们并没有 web.xml 文件，这时，我们可以使用 Java 代码去代替 web.xml 文件，这里会用到 WebApplicationInitializer，具体定义如下：

```
public class WebInit implements WebApplicationInitializer {  
    public void onStartup(ServletContext servletContext) throws ServletException  
{  
    //首先来加载 SpringMVC 的配置文件  
    AnnotationConfigWebApplicationContext ctx = new  
    AnnotationConfigWebApplicationContext();  
    ctx.register(SpringMVConfig.class);  
    // 添加 DispatcherServlet  
    ServletRegistration.Dynamic springmvc =  
    servletContext.addServlet("springmvc", new DispatcherServlet(ctx));  
    // 给 DispatcherServlet 添加路径映射  
    springmvc.addMapping("/");  
    // 给 DispatcherServlet 添加启动时机  
    springmvc.setLoadOnStartup(1);  
}  
}
```

WebInit 的作用类似于 web.xml，这个类需要实现 WebApplicationInitializer 接口，并实现接口中的方法，当项目启动时，onStartup 方法会被自动执行，我们可以在这个方法中做一些项目初始化操作，例如加载 SpringMVC 容器，添加过滤器，添加 Listener、添加 Servlet 等。

注意：

由于我们在 WebInit 中只是添加了 SpringMVC 的配置，这样项目在启动时只会去加载 SpringMVC 容器，而不会去加载 Spring 容器，如果一定要加载 Spring 容器，需要我们修改 SpringMVC 的配置，在 SpringMVC 配置的包扫描中也去扫描 @Configuration 注解，进而加载 Spring 容器，还有一种方案可以解决这个问题，就是直接在项目中舍弃 Spring 配置，直接将所有配置放到 SpringMVC 的配置中来完成，这个在 SSM 整合时是没有问题的，在实际开发中，较多采用第二种方案，第二种方案，SpringMVC 的配置如下：

```
@Configuration  
@ComponentScan(basePackages = "org.javaboy")  
public class SpringMVConfig {  
}
```

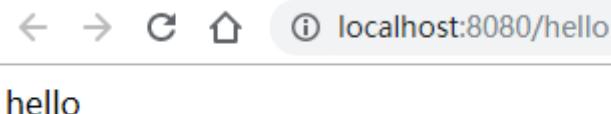
这种方案中，所有的注解都在 SpringMVC 中扫描，采用这种方案的话，则 Spring 的配置文件就可以删除了。

5 测试

最后，添加一个 HelloController，然后启动项目进行测试：

```
@RestController  
public class HelloController {  
    @GetMapping("/hello")  
    public String hello() {  
        return "hello";  
    }  
}
```

启动项目，访问接口，结果如下：



6 其他配置

6.1 静态资源过滤

静态资源过滤在 SpringMVC 的 XML 中的配置如下：

```
<mvc:resources mapping="/**" location="/" />
```

在 Java 配置的 SSM 环境中，如果要配置静态资源过滤，需要让 SpringMVC 的配置继承 WebMvcConfigurationSupport，进而重写 WebMvcConfigurationSupport 中的方法，如下：

```
@Configuration  
@ComponentScan(basePackages = "org.javaboy")  
public class SpringMVConfig extends WebMvcConfigurationSupport {  
    @Override  
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {  
  
        registry.addResourceHandler("/js/**").addResourceLocations("classpath:/");  
    }  
}
```

重写 addResourceHandlers 方法，在这个方法中配置静态资源过滤，这里我将静态资源放在 resources 目录下，所以资源位置是 `classpath:/`，当然，资源也可以放在 webapp 目录下，此时只需要修改配置中的资源位置即可。如果采用 Java 来配置 SSM 环境，一般来说，可以不必使用 webapp 目录，除非要使用 JSP 做页面模板，否则可以忽略 webapp 目录。

6.2 视图解析器

在 XML 文件中，通过如下方式配置视图解析器：

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="prefix" value="/WEB-INF/jsp/" />  
    <property name="suffix" value=".jsp" />  
</bean>
```

如果通过 Java 类，一样也可以实现类似功能。

首先为我们的项目添加 webapp 目录，webapp 目录中添加一个 jsp 目录，jsp 目录中添加 jsp 文件：



然后引入 JSP 的依赖:

```
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>javax.servlet.jsp-api</artifactId>
    <version>2.3.1</version>
</dependency>
```

然后，在配置类中，继续重写方法:

```
@Configuration
@ComponentScan(basePackages = "org.javaboy")
public class SpringMVConfig extends WebMvcConfigurerSupport {
    @Override
    protected void configureViewResolvers(ViewResolverRegistry registry) {
        registry.jsp("/jsp/", ".jsp");
    }
}
```

接下来，在 Controller 中添加控制器即可访问 JSP 页面:

```
@Controller
public class HelloController2 {
    @GetMapping("/hello2")
    public String hello() {
        return "hello";
    }
}
```

6.3 路径映射

有的时候，我们的控制器的作用仅仅只是一个跳转，就像上面小节中的控制器，里边没有任何业务逻辑，像这种情况，可以不用定义方法，可以直接通过路径映射来实现页面访问。如果在 XML 中配置路径映射，如下:

```
<mvc:view-controller path="/hello" view-name="hello" status-code="200"/>
```

这行配置，表示如果用户访问 /hello 这个路径，则直接将名为 hello 的视图返回给用户，并且响应码为 200，这个配置就可以替代 Controller 中的方法。

相同的需求，如果在 Java 代码中，写法如下:

```
@Configuration
@ComponentScan(basePackages = "org.javaboy")
public class SpringMVConfig extends WebMvcConfigurerSupport {
    @Override
    protected void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/hello3").setViewName("hello");
    }
}
```

此时，用户访问 /hello3 接口，就能看到名为 hello 的视图文件。

6.4 JSON 配置

SpringMVC 可以接收JSON 参数，也可以返回 JSON 参数，这一切依赖于 HttpMessageConverter。

HttpMessageConverter 可以将一个 JSON 字符串转为 对象，也可以将一个对象转为 JSON 字符串，实际上它的底层还是依赖于具体的 JSON 库。

所有的 JSON 库要在 SpringMVC 中自动返回或者接收 JSON，都必须提供和自己相关的 HttpMessageConverter。

SpringMVC 中，默认提供了 Jackson 和 gson 的 HttpMessageConverter，分别是： MappingJackson2HttpMessageConverter 和 GsonHttpMessageConverter。

正因为如此，我们在 SpringMVC 中，如果要使用 JSON，对于 jackson 和 gson 我们只需要添加依赖，加完依赖就可以直接使用了。具体的配置是在 AllEncompassingFormHttpMessageConverter 类中完成的。

如果开发者使用了 fastjson，那么默认情况下，SpringMVC 并没有提供 fastjson 的 HttpMessageConverter，这个需要我们自己提供，如果是在 XML 配置中，fastjson 除了加依赖，还要显式配置 HttpMessageConverter，如下：

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean
            class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter">
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>
```

在 Java 配置的 SSM 中，我们一样也可以添加这样的配置：

```
@Configuration
@ComponentScan(basePackages = "org.javaboy")
public class SpringMVConfig extends WebMvcConfigurerSupport {
    @Override
    protected void configureMessageConverters(List<HttpMessageConverter<?>>
            converters) {
        FastJsonHttpMessageConverter converter = new
        FastJsonHttpMessageConverter();
        converter.setDefaultCharset(Charset.forName("UTF-8"));
        FastJsonConfig fastJsonConfig = new FastJsonConfig();
        fastJsonConfig.setCharset(Charset.forName("UTF-8"));
        converter.setFastJsonConfig(fastJsonConfig);
        converters.add(converter);
    }
}
```

然后，就可以在接口中直接返回 JSON 了，此时的 JSON 数据将通过 fastjson 生成。

总结

好了，本文通过一个简单的例子向读者展示了使用 Java 来配置 Spring+SpringMVC 环境，事实上，只要这两个配置 OK，再加入 MyBatis 就是非常容易的事了，本文相关的案例松哥已经上传到 GitHub 上了：<https://github.com/lenve/javaboy-code-samples>。

关于本文，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南·一点雨

我最早是 2016 年底开始写 Spring Boot 相关的博客，当时使用的版本还是 1.4.x，文章发表在 CSDN 上，阅读量最大的一篇有 43W+，如下图：

The screenshot shows a sidebar from a CSDN blog page. At the top, there are navigation icons for back, forward, and search, followed by the URL 'wangsong.blog.csdn.net'. Below this is a section titled '热门文章' (Hot Articles) with the following list:

- [初识Spring Boot框架](#)
阅读数 430937
- [在Spring Boot中使用Spring Security实现权限控制](#)
阅读数 157632
- [Spring RestTemplate中几种常见的请求方式](#)
阅读数 136345
- [使用Spring Boot开发Web项目](#)
阅读数 131664
- [在Spring Boot框架下使用WebSocket实现消息推送](#)
阅读数 89179

2017 年由于种种原因，就没有再继续更新 Spring Boot 相关的博客了，2018 年又去写书了，也没更新，现在 Spring Boot 最新稳定版是 2.1.8，松哥想针对此写一个系列教程，专门讲 Spring Boot2 中相关的知识点。这个系列，就从本篇开始吧。

Spring Boot 介绍

我们刚开始学习 JavaWeb 的时候，使用 Servlet/JSP 做开发，一个接口搞一个 Servlet，很头大，后来我们通过隐藏域或者反射等方式，可以减少 Servlet 的创建，但是依然不方便，再后来，我们引入 Struts2/SpringMVC 这一类的框架，来简化我们的开发，和 Servlet/JSP 相比，引入框架之后，生产力确实提高了不少，但是用久了，又发现了新的问题，即配置繁琐易出错，要做一个新项目，先搭建环境，环境搭建来搭建去，就是那几行配置，不同的项目，可能就是包不同，其他大部分的配置都是一样的，Java 总是被人诟病配置繁琐代码量巨大，这就是其中一个表现。那么怎么办？Spring Boot 应运而生，Spring Boot 主要提供了如下功能：

1. 为所有基于 Spring 的 Java 开发提供方便快捷的入门体验。
2. 开箱即用，有自己自定义的配置就是用自己的，没有就使用官方提供的默认的。
3. 提供了一系列通用的非功能性的功能，例如嵌入式服务器、安全管理、健康检测等。
4. 绝对没有代码生成，也不需要 XML 配置。

Spring Boot 的出现让 Java 开发又回归简单，因为确确实实解决了开发中的痛点，因此这个技术得到了非常广泛的使用，松哥很多朋友出去面试 Java 工程师，从 2017 年年初开始，Spring Boot 基本就是必问，现在流行的 Spring Cloud 微服务也是基于 Spring Boot，因此，所有的 Java 工程师都有必要掌握好 Spring Boot。

系统要求

截至本文写作（2019.09），Spring Boot 目前最新版本是 2.1.8，要求至少 JDK8，集成的 Spring 版本是 5.1.9，构建工具版本要求如下：

Build Tool	Version
Maven	3.3+
Gradle	4.4+

内置的容器版本分别如下：

Name	Version
Tomcat 9.0	4.0
Jetty 9.4	3.1
Undertow 2.0	4.0

三种创建方式

初学者看到 Spring Boot 工程创建成功后有那么多文件就会有点懵圈，其实 Spring Boot 工程本质上就是一个 Maven 工程，从这个角度出发，松哥在这里向大家介绍三种项目创建方式。

在线创建

这是官方提供的一个创建方式，实际上，如果我们使用开发工具去创建 Spring Boot 项目的话（即第二种方案），也是从这个网站上创建的，只不过这个过程开发工具帮助我们完成了，我们只需要在开发工具中进行简单的配置即可。

首先打开 <https://start.spring.io> 这个网站，如下：

The screenshot shows the Spring Initializr web application interface. On the left, there's a sidebar with sections like 'Project', 'Language', 'Spring Boot', 'Project Metadata', 'Dependencies', and copyright information. The main area has tabs for 'Maven Project' and 'Gradle Project', with 'Maven Project' selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '2.1.8 (SNAPSHOT)' is selected. In 'Project Metadata', 'Group' is set to 'org.javaboy' and 'Artifact' to 'demo'. Under 'Options', 'Name' is 'demo', 'Description' is 'Demo project for Spring Boot', 'Package Name' is 'org.javaboy.demo', and 'Packaging' is 'Jar'. Below these, 'Java' version '8' is selected. In the 'Dependencies' section, there's a search bar and a list of available dependencies: 'Web, Security, JPA, Actuator, Devtools...'. At the bottom, there are two buttons: 'Generate the project - Ctrl + D' and 'Explore the project - Ctrl + Space'.

这里要配置的按顺序分别如下：

- 项目构建工具是 Maven 还是 Gradle？松哥见到有人用 Gradle 做 Java 后端项目，但是整体感觉 Gradle 在 Java 后端中使用的还是比较少，Gradle 在 Android 中使用较多，Java 后端，目前来看还是 Maven 为主，因此这里选择第一项。
- 开发语言，这个当然是选择 Java 了。
- Spring Boot 版本，可以看到，目前最新的稳定版是 2.1.8，这里我们就是用最新稳定版。
- 既然是 Maven 工程，当然要有项目坐标，项目描述等信息了，另外这里还让输入了包名，因为创建成功后会自动创建启动类。
- Packing 表示项目要打包成 jar 包还是 war 包，Spring Boot 的一大优势就是内嵌了 Servlet 容器，打成 jar 包后可以直接运行，所以这里建议打包成 jar 包，当然，开发者根据实际情况也可以选择 war 包。
- 然后选择构建的 JDK 版本。
- 最后是选择所需要的依赖，输入关键字如 web，会有相关的提示，这里我就先加入 web 依赖。

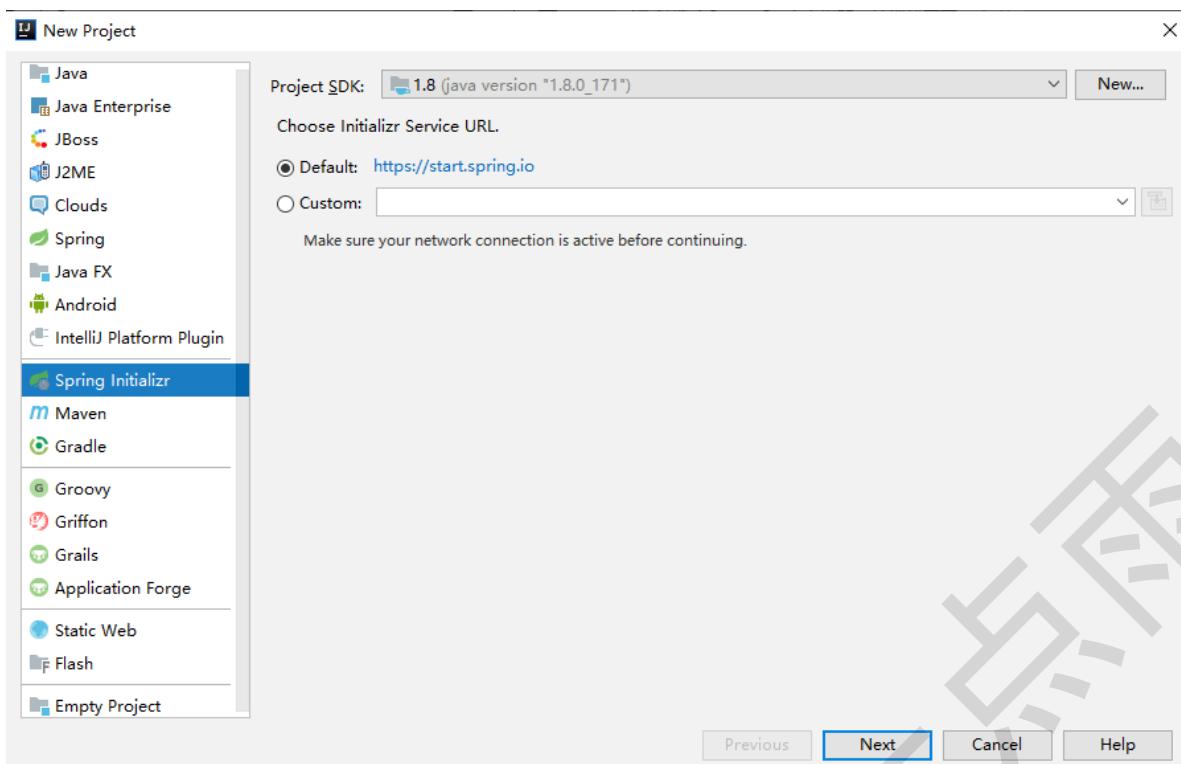
所有的事情全部完成后，点击最下面的 Generate Project 按钮，或者点击 Alt+Enter 按键，此时会自动下载项目，将下载下来的项目解压，然后用 IntelliJ IDEA 或者 Eclipse 打开即可进行开发。

使用开发工具创建

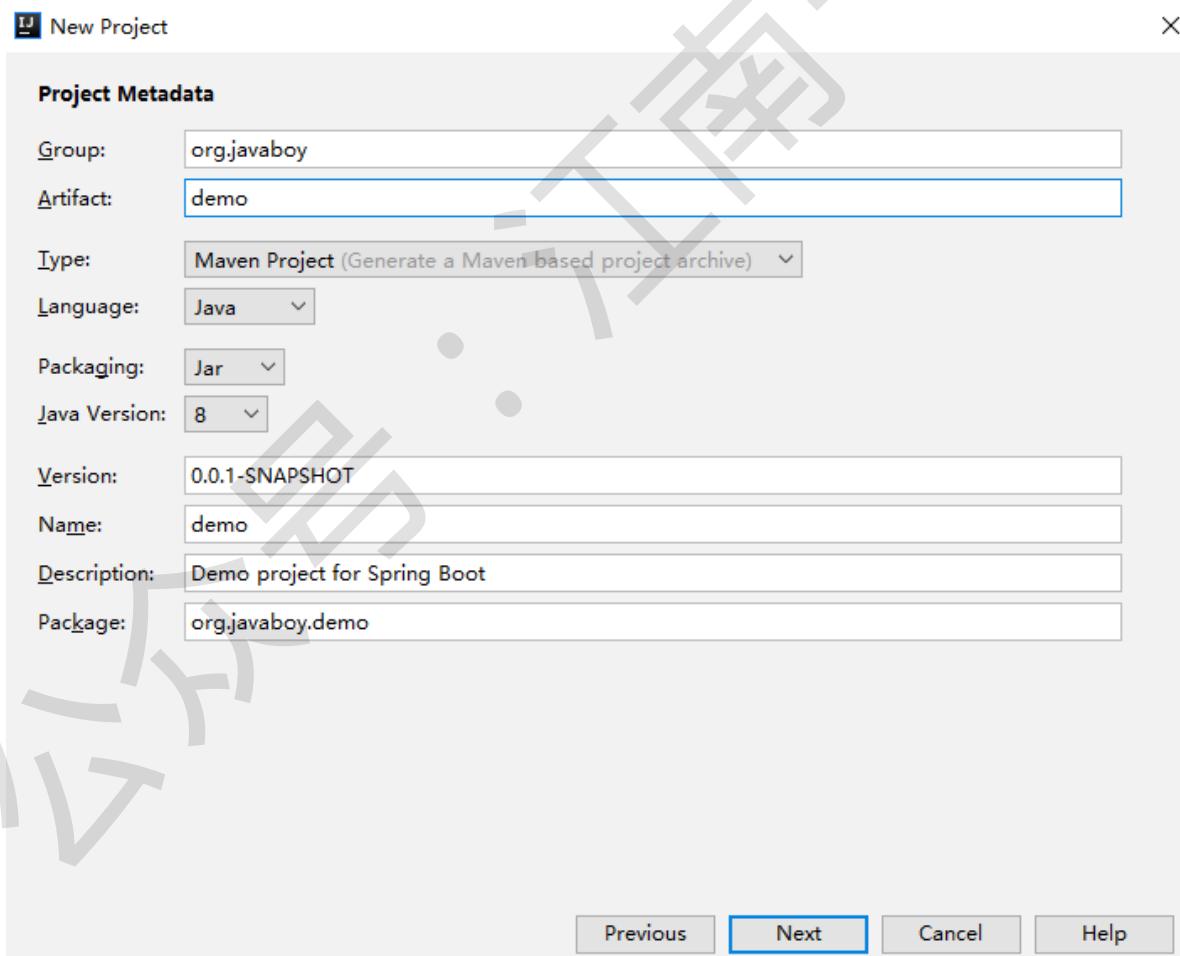
有人觉得上面的步骤太过于繁琐，那么也可以使用 IDE 来创建，松哥这里以 IntelliJ IDEA 和 STS 为例，需要注意的是，IntelliJ IDEA 只有 ultimate 版才有直接创建 Spring Boot 项目的功能，社区版是没有此功能的。

IntelliJ IDEA

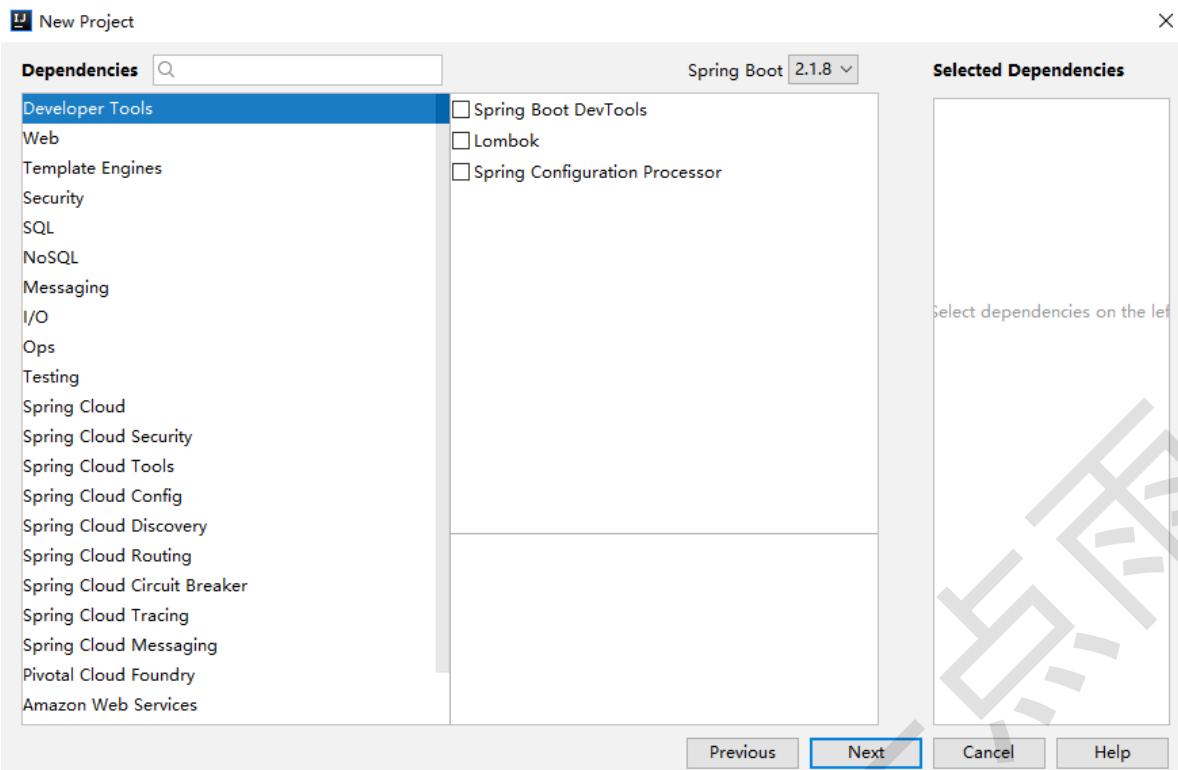
首先在创建项目时选择 Spring Initializr，如下图：



然后点击 Next , 填入 Maven 项目的基本信息, 如下:



再接下来选择需要添加的依赖, 如下图:

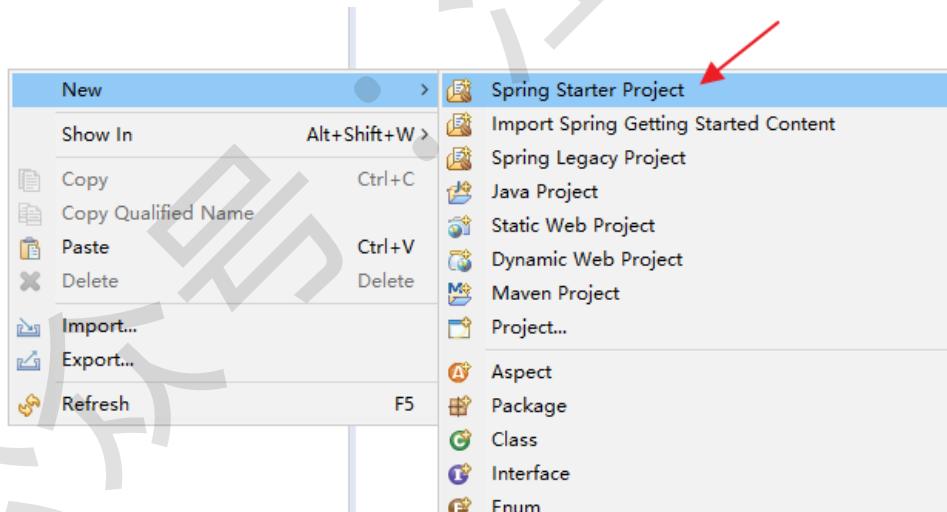


勾选完成后，点击 Next 完成项目的创建。

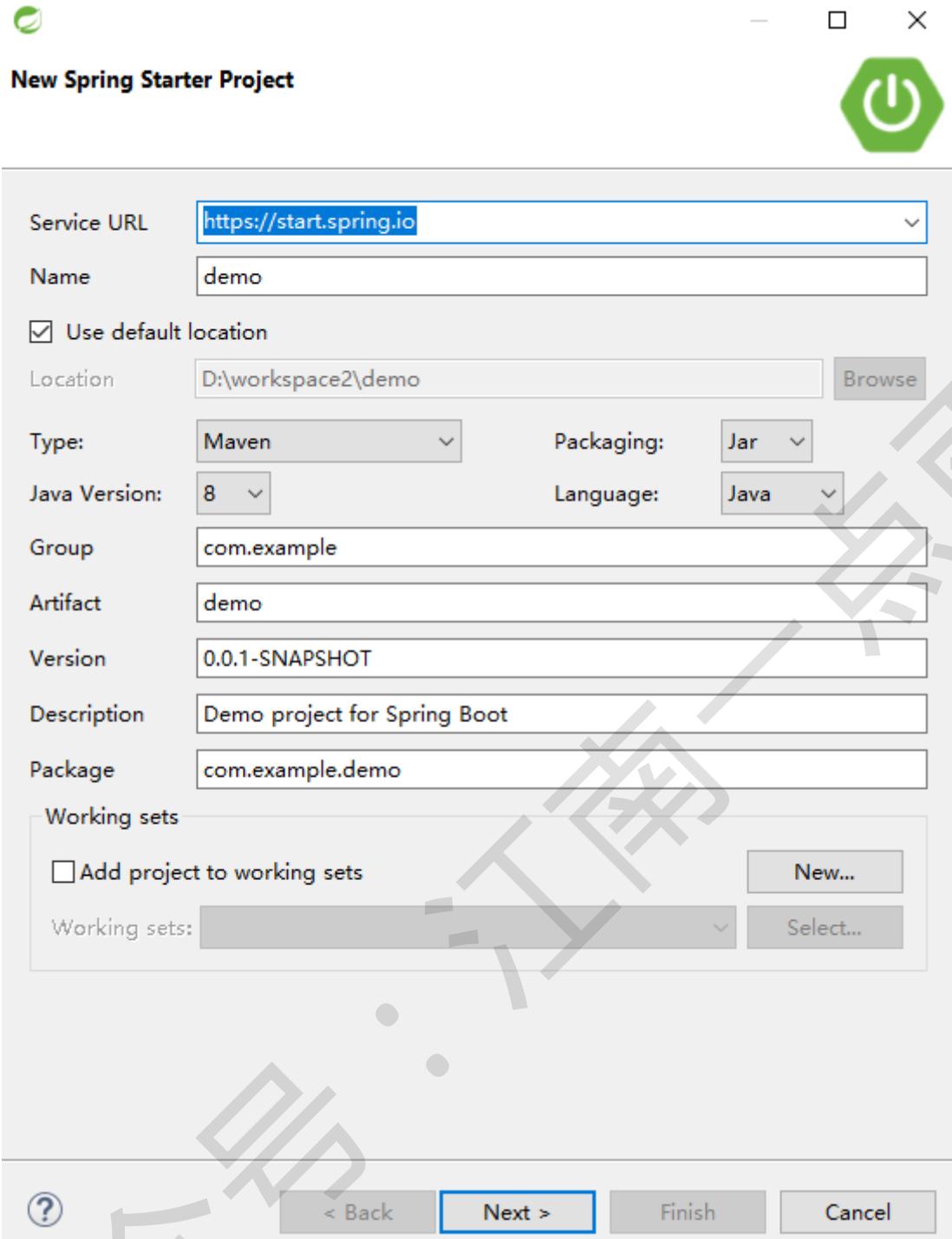
STS

这里我再介绍下 Eclipse 派系的 STS 给大家参考，STS 创建 Spring Boot 项目，实际上也是从上一小节的那个网站上来的，步骤如下：

首先右键单击，选择 New -> Spring Starter Project，如下图：



然后在打开的页面中填入项目的相关信息，如下图：

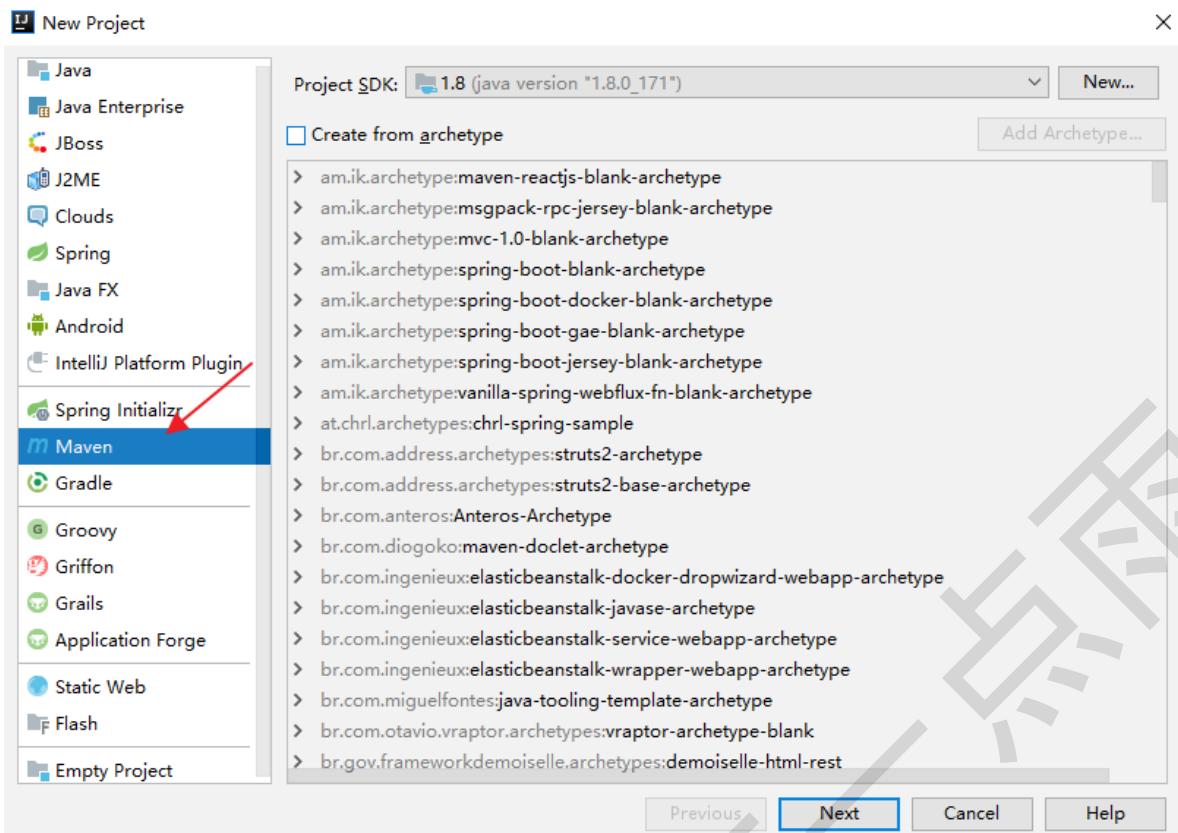


这里的信息和前面提到的都一样，不再赘述。最后一路点击 Next，完成项目的创建。

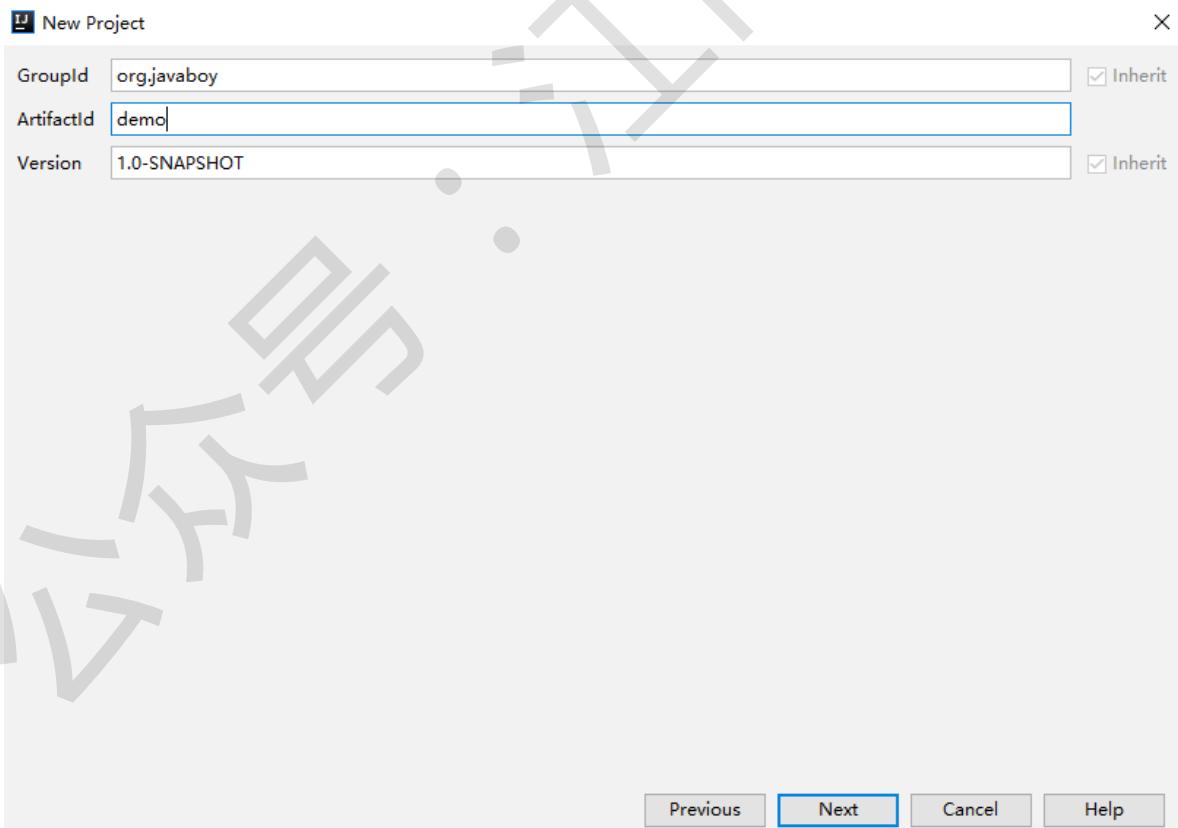
Maven 创建

上面提到的几种方式，实际上都借助了 <https://start.spring.io/> 这个网站，松哥记得在 2017 年的时候，这个网站还不是很稳定，经常发生项目创建失败的情况，从 2018 年开始，项目创建失败就很少遇到了，不过有一些读者偶尔还是会遇到这个问题，他们会在微信上问松哥这个问题要怎么处理？我一般给的建议就是直接使用 Maven 来创建项目。步骤如下：

首先创建一个普通的 Maven 项目，以 IntelliJ IDEA 为例，创建步骤如下：



注意这里不用选择项目骨架（如果大伙是做练习的话，也可以去尝试选择一下，这里大概有十来个 Spring Boot 相关的项目骨架），直接点击 Next，下一步中填入一个 Maven 项目的基本信息，如下图：



然后点击 Next 完成项目的创建。

创建完成后，在 pom.xml 文件中，添加如下依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.8.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

添加成功后，再在 java 目录下创建包，包中创建一个名为 App 的启动类，如下：

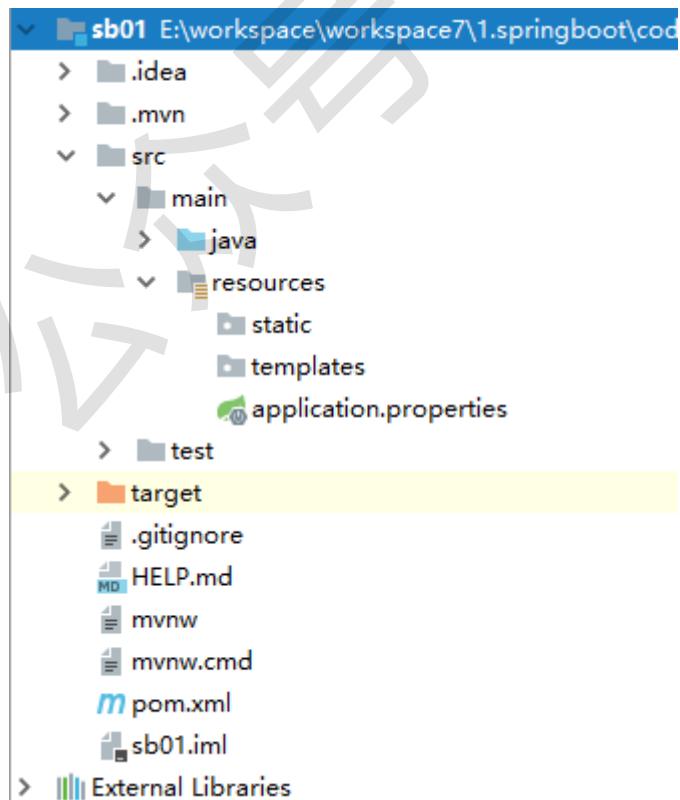
```
@EnableAutoConfiguration
@RestController
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

@EnableAutoConfiguration 注解表示开启自动化配置。

然后执行这里的 main 方法就可以启动一个 Spring Boot 工程了。

项目结构

使用工具创建出来的项目结构大致如下图：



对于我们来说，src 是最熟悉的，Java 代码和配置文件写在这里，test 目录用来做测试，pom.xml 是 Maven 的坐标文件，就这几个。

总结

本文主要向大家介绍了三种创建 Spring Boot 工程的方式，大家有更6的方法欢迎来讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



前面和大伙聊了 Spring Boot 项目的三种创建方式，这三种创建方式，无论是哪一种，创建成功后，pom.xml 坐标文件中都有如下一段引用：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.8.RELEASE</version>
  <relativePath/> <!-- Lookup parent from repository -->
</parent>
```

对于这个 parent 的作用，你是否完全理解？有小伙伴说，不就是依赖的版本号定义在 parent 里边吗？是的，没错，但是 parent 的作用可不仅仅这么简单哦！本文松哥就来和大伙聊一聊这个 parent 到底有什么作用。

基本功能

当我们创建一个 Spring Boot 工程时，可以继承自一个 `spring-boot-starter-parent`，也可以不继承自它，我们先来看第一种情况。先来看 parent 的基本功能有哪些？

1. 定义了 Java 编译版本为 1.8。
2. 使用 UTF-8 格式编码。
3. 继承自 `spring-boot-dependencies`，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
4. 执行打包操作的配置。
5. 自动化的资源过滤。
6. 自动化的插件配置。
7. 针对 `application.properties` 和 `application.yml` 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 `application-dev.properties` 和 `application-dev.yml`。

请注意，由于 `application.properties` 和 `application.yml` 文件接受 Spring 样式占位符 `$ { ... }`，因此 Maven 过滤更改为使用 `@ ... @` 占位符，当然开发者可以通过设置名为 `resource.delimiter` 的 Maven 属性来覆盖 `@ ... @` 占位符。

源码分析

当我们创建一个 Spring Boot 项目后，我们可以在本地 Maven 仓库中看到这个具体的 parent 文件，以 2.1.8 这个版本为例，松哥这里的路径是

`C:\Users\sang\.m2\repository\org\springframework\boot\spring-boot-starter-parent\2.1.8.RELEASE\spring-boot-starter-parent-2.1.8.RELEASE.pom`，打开这个文件，快速阅读文件源码，基本上就可以证实我们前面说的功能，如下图：

```
<?xml version="1.0" encoding="utf-8"?><project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>2.1.8.RELEASE</version>
    <relativePath>../../spring-boot-dependencies</relativePath>
  </parent>
  <artifactId>spring-boot-starter-parent</artifactId>
  <packaging>pom</packaging>
  <name>Spring Boot Starter Parent</name>
  <description>Parent pom providing dependency and plugin management for applications built with Maven</description>
  <url>https://projects.spring.io/spring-boot/#/spring-boot-starter-parent</url>
  <properties>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <resource.delimiter>@</resource.delimiter>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.target>${java.version}</maven.compiler.target>
  </properties>
  <build>
    <resources>
      <resource>
        <filtering>true</filtering>
        <directory>${basedir}/src/main/resources</directory>
        <includes>
          <include>**/application*.yml</include>
          <include>**/application*.yaml</include>
          <include>**/application*.properties</include>
        </includes>
      </resource>
    <resources>

```

我们可以看到，它继承自 `spring-boot-dependencies`，这里保存了基本的依赖信息，另外我们也可以看到项目的编码格式，JDK 的版本等信息，当然也有我们前面提到的数据过滤信息。最后，我们再根据它的 `parent` 中指定的 `spring-boot-dependencies` 位置，来看看 `spring-boot-dependencies` 中的定义：

```
<?xml version="1.0" encoding="utf-8"?><project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.1.8.RELEASE</version>
  <packaging>pom</packaging>
  <name>Spring Boot Dependencies</name>
  <description>Spring Boot Dependencies</description>
  <url>https://projects.spring.io/spring-boot/#</url>
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>https://www.apache.org/licenses/LICENSE-2.0</url>
    </license>
  </licenses>
  <developers>
    <developer>
      <name>Pivotal</name>
      <email>info@pivotal.io</email>
      <organization>Pivotal Software, Inc.</organization>
      <organizationUrl>https://www.spring.io</organizationUrl>
    </developer>
  </developers>
  <scm>
    <url>https://github.com/spring-projects/spring-boot</url>
  </scm>
  <properties>
    <activemq.version>5.15.10</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.76</appengine-sdk.version>
    <artemis.version>2.6.4</artemis.version>
    <aspectj.version>1.9.4</aspectj.version>
    <assertj.version>3.11.1</assertj.version>
    <atomikos.version>4.0.6</atomikos.version>
  </properties>
```

在这里，我们看到了版本的定义以及 `dependencyManagement` 节点，明白了为啥 Spring Boot 项目中部分依赖不需要写版本号了。

不用 parent

但是并非所有的公司都需要这个 `parent`，有的时候，公司里边会有自己定义的 `parent`，我们的 Spring Boot 项目要继承自公司内部的 `parent`，这个时候该怎么办呢？

一个简单的办法就是我们自行定义 `dependencyManagement` 节点，然后在里边定义好版本号，再接下来在引用依赖时也就不用写版本号了，像下面这样：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.1.8.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

这样写之后，依赖的版本号问题虽然解决了，但是关于打包的插件、编译的 JDK 版本、文件的编码格式等等这些配置，在没有 parent 的时候，这些统统要自己去配置。

总结

好了，一篇简单的文章，向大伙展示一下 Spring Boot 项目中 parent 的作用，有问题欢迎留言讨论。本文相关的案例松哥已经上传到 GitHub 上了：<https://github.com/lenve/javaboy-code-samples>。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



在 Spring Boot 中，配置文件有两种不同的格式，一个是 properties，另一个是 yaml。

虽然 properties 文件比较常见，但是相对于 properties 而言，yaml 更加简洁明了，而且使用的场景也更多，很多开源项目都是使用 yaml 进行配置（例如 Hexo）。除了简洁，yaml 还有另外一个特点，就是 yaml 中的数据是有序的，properties 中的数据是无序的，在一些需要路径匹配的配置中，顺序就显得尤为重要（例如我们在 Spring Cloud Zuul 中的配置），此时我们一般采用 yaml。关于 yaml，松哥之前写过一篇文章：[Spring Boot 中的 yaml 配置简介](#)。

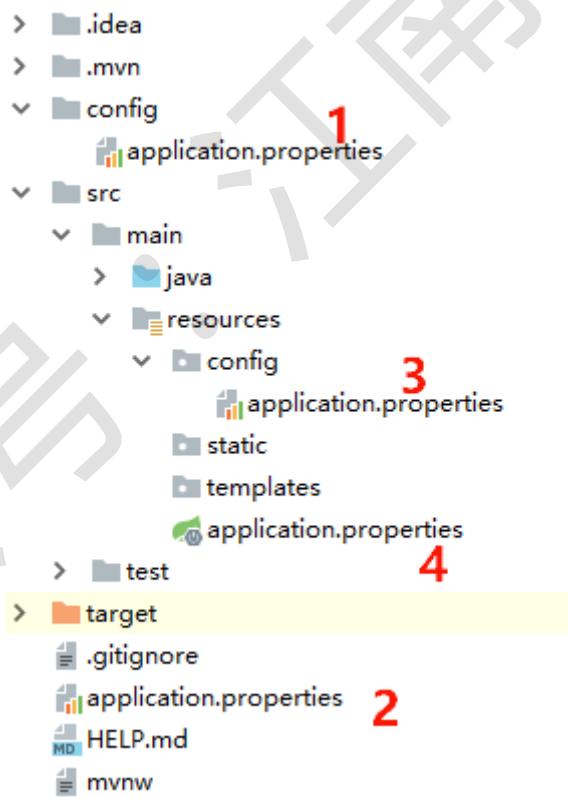
本文主要来看看 properties 的问题。

位置问题

首先，当我们创建一个 Spring Boot 工程时，默认 resources 目录下就有一个 application.properties 文件，可以在 application.properties 文件中进行项目配置，但是这个文件并非唯一的配置文件，在 Spring Boot 中，一共有 4 个地方可以存放 application.properties 文件。

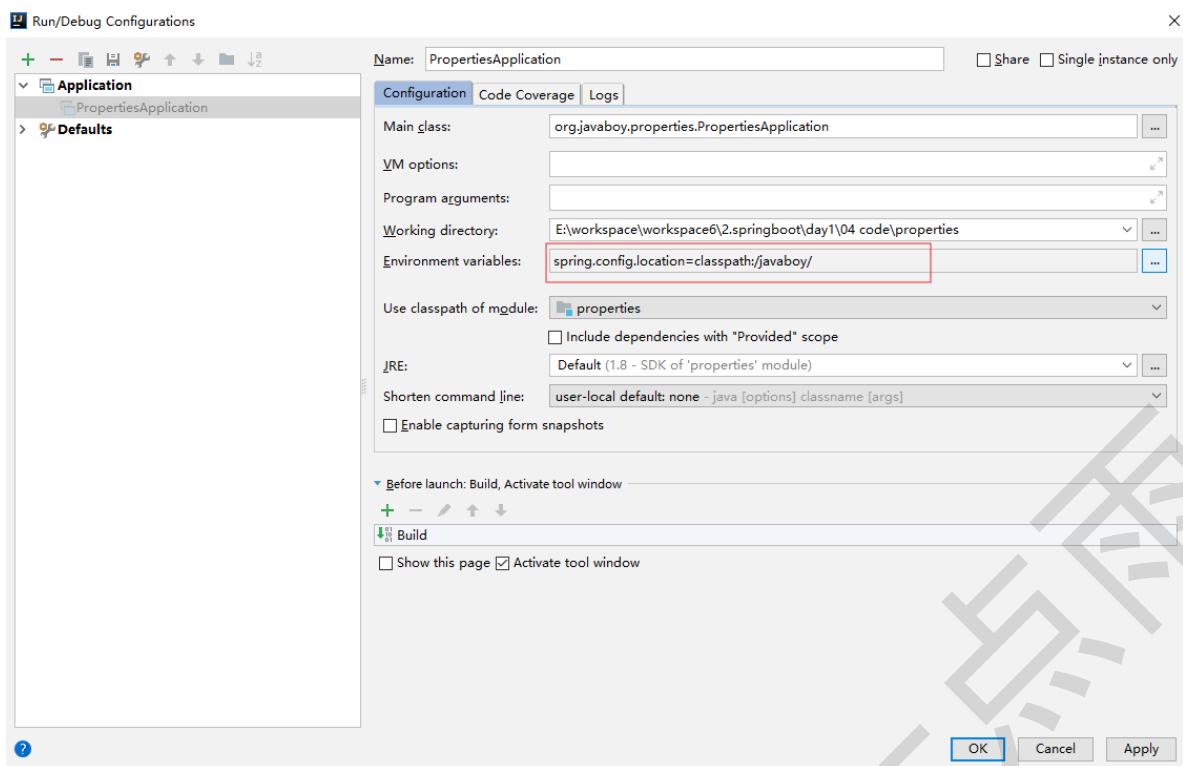
1. 当前项目根目录下的 config 目录下
2. 当前项目的根目录下
3. resources 目录下的 config 目录下
4. resources 目录下

按如上顺序，四个配置文件的优先级依次降低。如下：



这四个位置是默认位置，即 Spring Boot 启动，默认会从这四个位置按顺序去查找相关属性并加载。但是，这也不是绝对的，我们也可以在项目启动时自定义配置文件位置。

例如，现在在 resources 目录下创建一个 javaboy 目录，目录中存放一个 application.properties 文件，那么正常情况下，当我们启动 Spring Boot 项目时，这个配置文件是不会被自动加载的。我们可以通过 spring.config.location 属性来手动的指定配置文件位置，指定完成后，系统就会自动去指定目录下查找 application.properties 文件。



此时启动项目，就会发现，项目以 `classpath:/javaboy/application.properties` 配置文件启动。

这是在开发工具中配置了启动位置，如果项目已经打包成 jar，在启动命令中加入位置参数即可：

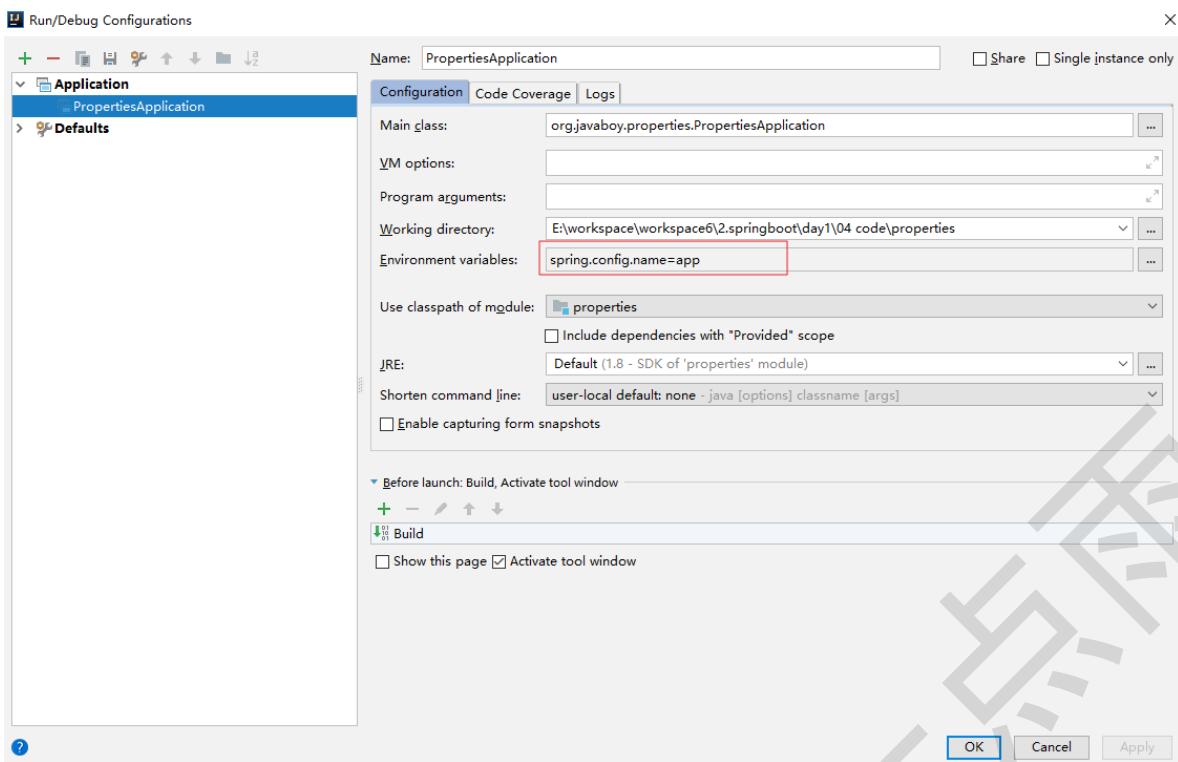
```
java -jar properties-0.0.1-SNAPSHOT.jar --  
spring.config.location=classpath:/javaboy/
```

文件名问题

对于 `application.properties` 而言，它不一定非要叫 `application`，但是项目默认是去加载名为 `application` 的配置文件，如果我们的配置文件不叫 `application`，也是可以的，但是，需要明确指定配置文件的文件名。

方式和指定路径一致，只不过此时的 key 是 `spring.config.name`。

首先我们在 `resources` 目录下创建一个 `app.properties` 文件，然后在 IDEA 中指定配置文件的文件名：



指定完配置文件名之后，再次启动项目，此时系统会自动去默认的四个位置下面分别查找名为 app.properties 的配置文件。当然，允许自定义文件名的配置文件不放在四个默认位置，而是放在自定义目录下，此时就需要明确指定 spring.config.location。

配置文件位置和文件名称可以同时自定义。

普通的属性注入

由于 Spring Boot 源自 Spring，所以 Spring 中存在的属性注入，在 Spring Boot 中一样也存在。由于 Spring Boot 中，默认会自动加载 application.properties 文件，所以简单的属性注入可以直接在这个配置文件中写。

例如，现在定义一个 Book 类：

```
public class Book {  
    private Long id;  
    private String name;  
    private String author;  
    //省略 getter/setter  
}
```

然后，在 application.properties 文件中定义属性：

```
book.name=三国演义  
book.author=罗贯中  
book.id=1
```

按照传统的方式（Spring中的方式），可以直接通过 @Value 注解将这些属性注入到 Book 对象中：

```
@Component
public class Book {
    @Value("${book.id}")
    private Long id;
    @Value("${book.name}")
    private String name;
    @Value("${book.author}")
    private String author;
    //省略getter/setter
}
```

注意

Book 对象本身也要交给 Spring 容器去管理，如果 Book 没有交给 Spring 容器，那么 Book 中的属性也无法从 Spring 容器中获取到值。

配置完成后，在 Controller 或者单元测试中注入 Book 对象，启动项目，就可以看到属性已经注入到对象中了。

一般来说，我们在 application.properties 文件中主要存放系统配置，这种自定义配置不建议放在该文件中，可以自定义 properties 文件来存在自定义配置。

例如在 resources 目录下，自定义 book.properties 文件，内容如下：

```
book.name=三国演义
book.author=罗贯中
book.id=1
```

此时，项目启动并不会自动的加载该配置文件，如果是在 XML 配置中，可以通过如下方式引用该 properties 文件：

```
<context:property-placeholder location="classpath:book.properties"/>
```

如果是在 Java 配置中，可以通过 @PropertySource 来引入配置：

```
@Component
@PropertySource("classpath:book.properties")
public class Book {
    @Value("${book.id}")
    private Long id;
    @Value("${book.name}")
    private String name;
    @Value("${book.author}")
    private String author;
    //getter/setter
}
```

这样，当项目启动时，就会自动加载 book.properties 文件。

这只是 Spring 中属性注入的一个简单用法，和 Spring Boot 没有任何关系。

类型安全的属性注入

Spring Boot 引入了类型安全的属性注入，如果采用 Spring 中的配置方式，当配置的属性非常多的时候，工作量就很大了，而且容易出错。

使用类型安全的属性注入，可以有效的解决这个问题。

```
@Component  
 @PropertySource("classpath:book.properties")  
 @ConfigurationProperties(prefix = "book")  
 public class Book {  
     private Long id;  
     private String name;  
     private String author;  
     //省略getter/setter  
 }
```

这里，主要是引入 @ConfigurationProperties(prefix = "book") 注解，并且配置了属性的前缀，此时会自动将 Spring 容器中对应的数据注入到对象对应的属性中，就不用通过 @Value 注解挨个注入了，减少工作量并且避免出错。

总结

application.properties 是 Spring Boot 中配置的一个重要载体，很多组件的属性都可以在这里定制。它的用法和 yaml 比较类似，大家可以参考 [Spring Boot 中的 yaml 配置简介](#)。

本文案例我已上传到 GitHub： <https://github.com/lenve/javaboy-code-samples>

好了，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



搞 Spring Boot 的小伙伴都知道，Spring Boot 中的配置文件有两种格式，properties 或者 yaml，一般情况下，两者可以随意使用，选择自己顺手的就行了，那么这两者完全一样吗？肯定不是啦！本文就来和大伙重点介绍下 yaml 配置，最后再来看看 yaml 和 properties 配置有何区别。

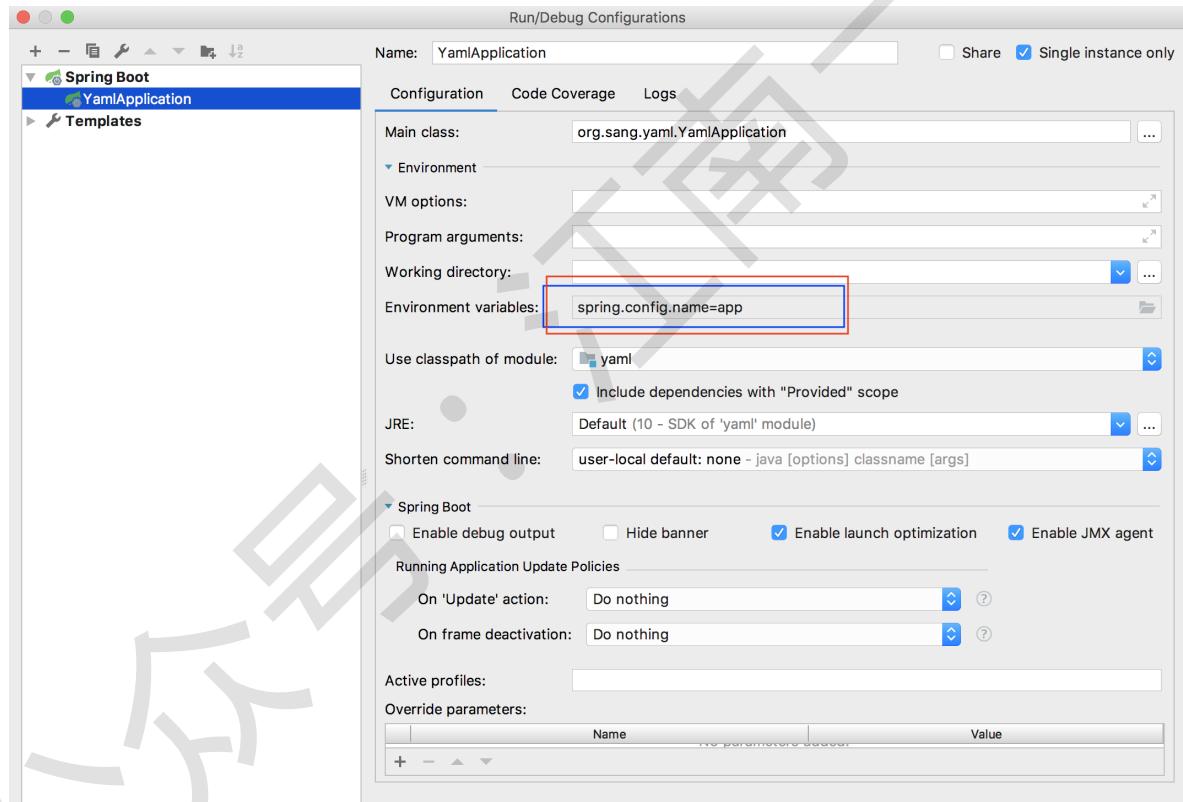
狡兔三窟

首先 application.yaml 在 Spring Boot 中可以写在四个不同的位置，分别是如下位置：

1. 项目根目录下的 config 目录中
2. 项目根目录下
3. classpath 下的 config 目录中
4. classpath 目录下

四个位置中的 application.yaml 文件的优先级按照上面列出的顺序依次降低。即如果有同一个属性在四个文件中都出现了，以优先级高的为准。

那么 application.yaml 是不是必须叫 application.yaml 这个名字呢？当然不是必须的。开发者可以自己定义 yaml 名字，自己定义的话，需要在项目启动时指定配置文件的名字，像下面这样：

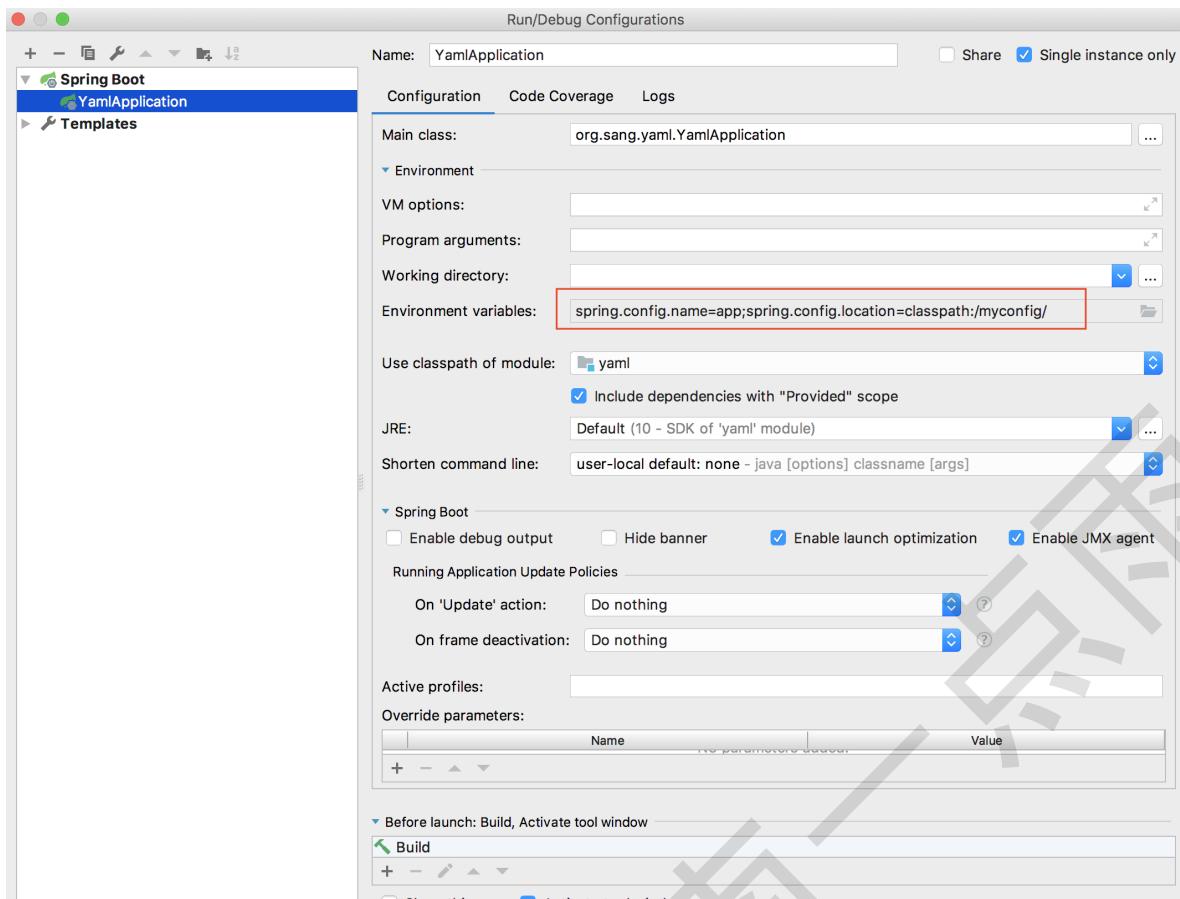


当然这是在 IntelliJ IDEA 中直接配置的，如果项目已经打成 jar 包了，则在项目启动时加入如下参数：

```
java -jar myproject.jar --spring.config.name=app
```

这样配置之后，在项目启动时，就会按照上面所说的四个位置按顺序去查找一个名为 app.yaml 的文件。当然这四个位置也不是一成不变的，也可以自己定义，有两种方式，一个是使用 `spring.config.location` 属性，另一个则是使用 `spring.config.additional-location` 这个属性，在第一个属性中，表示自己重新定义配置文件的位置，项目启动时就按照定义的位置去查找配置文件，这种定义方式会覆盖掉默认的四个位置，也可以使用第二种方式，第二种方式则表示在四个位置的基础上，再添加几个位置，新添加的位置的优先级大于原本的位置。

配置方式如下：



这里要注意，配置文件位置时，值一定要以 / 结尾。

数组注入

yaml 也支持数组注入，例如

```
my:  
  servers:  
    - dev.example.com  
    - another.example.com
```

这段数据可以绑定到一个带 Bean 的数组中：

```
@ConfigurationProperties(prefix="my")  
@Component  
public class Config {  
  
    private List<String> servers = new ArrayList<String>();  
  
    public List<String> getServers() {  
        return this.servers;  
    }  
}
```

项目启动后，配置中的数组会自动存储到 servers 集合中。当然，yaml 不仅可以存储这种简单数据，也可以在集合中存储对象。例如下面这种：

```
redis:  
  redisConfigs:  
    - host: 192.168.66.128  
      port: 6379  
    - host: 192.168.66.129  
      port: 6380
```

这个可以被注入到如下类中：

```
@Component  
@ConfigurationProperties(prefix = "redis")  
public class RedisCluster {  
    private List<SingleRedisConfig> redisConfigs;  
    //省略getter/setter  
}
```

优缺点

不同于 properties 文件的无序，yaml 配置是有序的，这一点在有些配置中是非常有用的，例如在 Spring Cloud Zuul 的配置中，当我们配置代理规则时，顺序就显得尤为重要了。当然 yaml 配置也不是万能的，例如，yaml 配置目前不支持 @PropertySource 注解。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



我们使用 Spring Boot，基本上都是沉醉在它 Starter 的方便之中。Starter 为我们带来了众多的自动化配置，有了这些自动化配置，我们可以不费吹灰之力就能搭建一个生产级开发环境，有的小伙伴会觉得这个 Starter 好神奇呀！其实 Starter 也都是 Spring + SpringMVC 中的基础知识点实现的，今天松哥就来带大家自己来撸一个 Starter，慢慢揭开 Starter 的神秘面纱！

1.核心知识

其实 Starter 的核心就是条件注解 `@Conditional`，当 classpath 下存在某一个 Class 时，某个配置才会生效，前面松哥已经带大家学习过不少 Spring Boot 中的知识点，有的也涉及到源码解读，大伙可能也发现了源码解读时总是会出现条件注解，其实这就是 Starter 配置的核心之一，大伙有兴趣可以翻翻历史记录，看看松哥之前写的关于 Spring Boot 的文章，这里我就不再重复介绍了。

2.定义自己的 Starter

2.1定义

所谓的 Starter，其实就是一个普通的 Maven 项目，因此我们自定义 Starter，需要首先创建一个普通的 Maven 项目，创建完成后，添加 Starter 的自动化配置类即可，如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
    <version>2.1.8.RELEASE</version>
</dependency>
```

配置完成后，我们首先创建一个 HelloProperties 类，用来接受 application.properties 中注入的值，如下：

```
@ConfigurationProperties(prefix = "javaboy")
public class HelloProperties {
    private static final String DEFAULT_NAME = "江南一点雨";
    private static final String DEFAULT_MSG = "牧码小子";
    private String name = DEFAULT_NAME;
    private String msg = DEFAULT_MSG;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getMsg() {
        return msg;
    }
    public void setMsg(String msg) {
        this.msg = msg;
    }
}
```

这个配置类很好理解，将 application.properties 中配置的属性值直接注入到这个实例中，`@ConfigurationProperties` 类型安全的属性注入，即将 application.properties 文件中前缀为 javaboy 的属性注入到这个类对应的属性上，最后使用时候，application.properties 中的配置文件，大概如下：

```
javaboy.name=zhangsan  
javaboy.msg=java
```

关注类型安全的属性注入，读者可以参考松哥之前的这篇文章：[Spring Boot中的yaml配置简介](#)，这篇文章虽然是讲 yaml 配置，但是关于类型安全的属性注入和 properties 是一样的。

配置完成 HelloProperties 后，接下来我们来定义一个 HelloService，然后定义一个简单的 say 方法，HelloService 的定义如下：

```
public class HelloService {  
    private String msg;  
    private String name;  
    public String sayHello() {  
        return name + " say " + msg + " !";  
    }  
    public String getMsg() {  
        return msg;  
    }  
    public void setMsg(String msg) {  
        this.msg = msg;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

这个很简单，没啥好说的。

接下来就是我们的重头戏，自动配置类的定义，用了很多别人定义的自定义类之后，我们也来自定义一个自定义类。先来看代码吧，一会松哥再慢慢解释：

```
@Configuration  
@EnableConfigurationProperties(HelloProperties.class)  
@ConditionalOnClass(HelloService.class)  
public class HelloServiceAutoConfiguration {  
    @Autowired  
    HelloProperties helloProperties;  
  
    @Bean  
    HelloService helloService() {  
        HelloService helloService = new HelloService();  
        helloService.setName(helloProperties.getName());  
        helloService.setMsg(helloProperties.getMsg());  
        return helloService;  
    }  
}
```

关于这一段自动配置，解释如下：

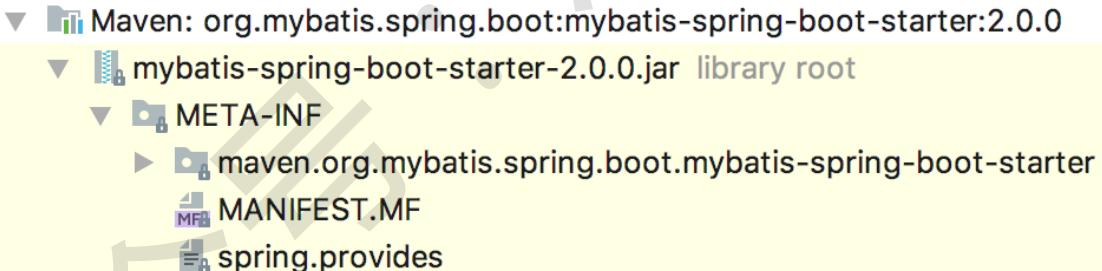
- 首先 @Configuration 注解表明这是一个配置类。
- @EnableConfigurationProperties 注解是使我们之前配置的 @ConfigurationProperties 生效，让配置的属性成功的进入 Bean 中。
- @ConditionalOnClass 表示当项目当前 classpath 下存在 HelloService 时，后面的配置才生效。
- 自动配置类中首先注入 HelloProperties，这个实例中含有我们在 application.properties 中配置的相关数据。
- 提供一个 HelloService 的实例，将 HelloProperties 中的值注入进去。

做完这一步之后，我们的自动化配置类就算是完成了，接下来还需要一个 spring.factories 文件，那么这个文件是干嘛的呢？大家知道我们的 Spring Boot 项目的启动类都有一个 @SpringBootApplication 注解，这个注解的定义如下：

```
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(excludeFilters = {  
    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  
    @Filter(type = FilterType.CUSTOM,  
           classes = AutoConfigurationExcludeFilter.class) })  
public @interface SpringApplication {  
}
```

大家看到这是一个组合注解，其中的一个组合项就是 @EnableAutoConfiguration，这个注解是干嘛的呢？

@EnableAutoConfiguration 表示启用 Spring 应用程序上下文的自动配置，该注解会自动导入一个名为 AutoConfigurationImportSelector 的类，而这个类会去读取一个名为 spring.factories 的文件，spring.factories 中则定义需要加载的自动化配置类，我们打开任意一个框架的 Starter，都能看到它有一个 spring.factories 文件，例如 MyBatis 的 Starter 如下：



那么我们自定义 Starter 当然也需要这样一个文件，我们首先在 Maven 项目的 resources 目录下创建一个名为 META-INF 的文件夹，然后在文件夹中创建一个名为 spring.factories 的文件，文件内容如下：

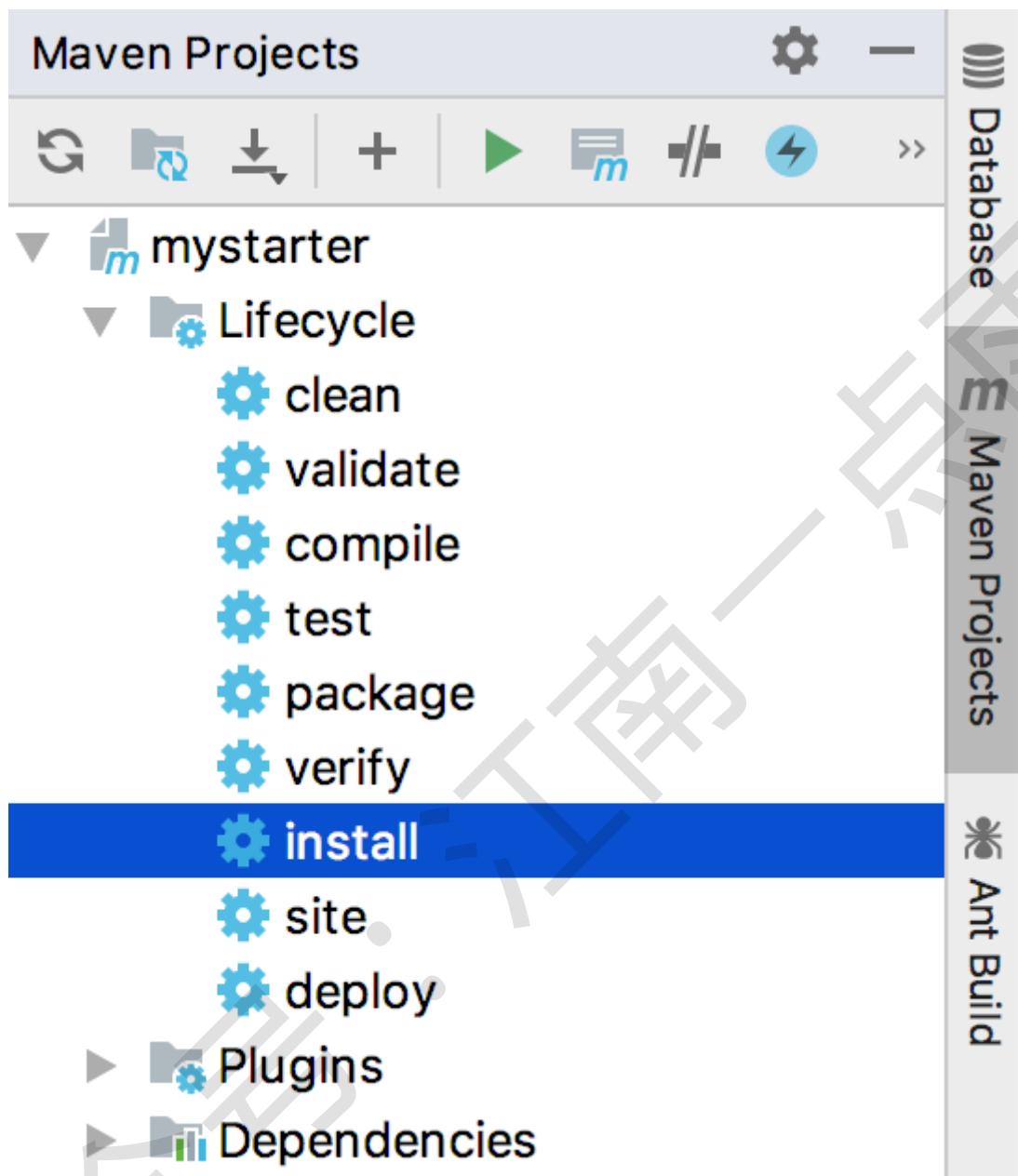
```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=org.javaboy.mystarter.HelloServiceAutoConfiguration
```

在这里指定我们的自动化配置类的路径即可。

如此之后我们的自动化配置类就算完成了。

2.2本地安装

如果在公司里，大伙可能需要将刚刚写好的自动化配置类打包，然后上传到 Maven 私服上，供其他同事下载使用，我这里就简单一些，我就不上传私服了，我将这个自动化配置类安装到本地仓库，然后在其他项目中使用即可。安装方式很简单，在 IntelliJ IDEA 中，点击右边的 Maven Project，然后选择 Lifecycle 中的 install，双击即可，如下：



双击完成后，这个 Starter 就安装到我们本地仓库了，当然小伙伴也可以使用 Maven 命令去安装。

3. 使用 Starter

接下来，我们来新建一个普通的 Spring Boot 工程，这个 Spring Boot 创建成功之后，加入我们自定义 Starter 的依赖，如下：

```
<dependency>
    <groupId>org.javaboy</groupId>
    <artifactId>mystarter</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

此时我们引入了上面自定义的 Starter，也即我们项目中现在有一个默认的 HelloService 实例可以使用，而且关于这个实例的数据，我们还可以在 application.properties 中进行配置，如下：

javaboy.name=牧码小子
javaboy.msg=java

配置完成后，方便起见，我这里直接在单元测试方法中注入 HelloService 实例来使用，代码如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UseMyStarterApplicationTests {
    @Autowired
    HelloService helloService;
    @Test
    public void contextLoads() {
        System.out.println(helloService.sayHello());
    }
}
```

执行单元测试方法，打印日志如下：

```
2019-05-10 22:03:19.627 INFO 20310 --- [  
2019-05-10 22:03:19.629 INFO 20310 --- [  
2019-05-10 22:03:21.146 INFO 20310 --- [  
2019-05-10 22:03:21.420 INFO 20310 --- [  
牧码小子 say java !  
2019-05-10 22:03:21.739 INFO 20310 --- [
```

好了，一个简单的自动化配置类我们就算完成了，是不是很简单！

4. 总结

本文主要带领小伙伴们自己徒手撸一个 Starter，使用这种方式帮助大家揭开 Starter 的神秘面纱！小伙伴们有问题可以留言讨论。

本文的案例，松哥已经上传到 GitHub 上了，地址：<https://github.com/lenve/javaboy-code-samples>。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

Spring Boot 中的自动化配置确实够吸引人，甚至有人说 Spring Boot 让 Java 又一次焕发了生机，这话虽然听着有点夸张，但是不可否认的是，曾经臃肿繁琐的 Spring 配置确实让人感到头大，而 Spring Boot 带来的全新自动化配置，又确实缓解了这个问题。

你要是问这个自动化配置是怎么实现的，很多人会说不就是 starter 嘛！那么 starter 的原理又是什么呢？松哥以前写过一篇文章，介绍了自定义 starter：

- [徒手撸一个 Spring Boot 中的 Starter，解密自动化配置黑魔法！](#)

这里边有一个非常关键的点，那就是**条件注解**，甚至可以说条件注解是整个 Spring Boot 的基石。

条件注解并非一个新事物，这是一个存在于 Spring 中的东西，我们在 Spring 中常用的 profile 实际上就是条件注解的一个特殊化。

想要把 Spring Boot 的原理搞清，条件注解必须要会用，因此今天松哥就来和大家聊一聊条件注解。

定义

Spring4 中提供了更加通用的条件注解，让我们可以在满足不同条件时创建不同的 Bean，这种配置方式在 Spring Boot 中得到了广泛的使用，大量的自动化配置都是通过条件注解来实现的，查看松哥之前的 Spring Boot 文章，凡是涉及到源码解读的文章，基本上都离不开条件注解：

- [40 篇原创干货，带你进入 Spring Boot 殿堂！](#)

有的小伙伴可能没用过条件注解，但是开发环境、生产环境切换的 Profile 多多少少都有用过吧？实际上这就是条件注解的一个特例。

实践

抛开 Spring Boot，我们来单纯的看看在 Spring 中条件注解的用法。

首先我们来创建一个普通的 Maven 项目，然后引入 spring-context，如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
</dependencies>
```

然后定义一个 Food 接口：

```
public interface Food {
    String showName();
}
```

Food 接口有一个 showName 方法和两个实现类：

```

public class Rice implements Food {
    public String showName() {
        return "米饭";
    }
}
public class Noodles implements Food {
    public String showName() {
        return "面条";
    }
}

```

分别是 Rice 和 Noodles 两个类，两个类实现了 showName 方法，然后分别返回不同值。

接下来再分别创建 Rice 和 Noodles 的条件类，如下：

```

public class NoodlesCondition implements Condition {
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("people").equals("北方人");
    }
}
public class RiceCondition implements Condition {
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("people").equals("南方人");
    }
}

```

在 matches 方法中做条件属性判断，当系统属性中的 people 属性值为 '北方人' 的时候，NoodlesCondition 的条件得到满足，当系统中 people 属性值为 '南方人' 的时候，RiceCondition 的条件得到满足，换句话说，哪个条件得到满足，一会就会创建哪个 Bean。

接下来我们来配置 Rice 和 Noodles：

```

@Configuration
public class JavaConfig {
    @Bean("food")
    @Conditional(RiceCondition.class)
    Food rice() {
        return new Rice();
    }
    @Bean("food")
    @Conditional(NoodlesCondition.class)
    Food noodles() {
        return new Noodles();
    }
}

```

这个配置类，大家重点注意两个地方：

- 两个 Bean 的名字都为 food，这不是巧合，而是有意取的。两个 Bean 的返回值都为其父类对象 Food。
- 每个 Bean 上都多了 @Conditional 注解，当 @Conditional 注解中配置的条件类的 matches 方法返回值为 true 时，对应的 Bean 就会生效。

配置完成后，我们就可以在 main 方法中进行测试了：

```
public class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext ctx = new  
AnnotationConfigApplicationContext();  
        ctx.getEnvironment().getSystemProperties().put("people", "南方人");  
        ctx.register(JavaConfig.class);  
        ctx.refresh();  
        Food food = (Food) ctx.getBean("food");  
        System.out.println(food.showName());  
    }  
}
```

首先我们创建一个 AnnotationConfigApplicationContext 实例用来加载 Java 配置类，然后我们添加一个 property 到 environment 中，添加完成后，再去注册我们的配置类，然后刷新容器。容器刷新完成后，我们就可以从容器中去获取 food 的实例了，这个实例会根据 people 属性的不同，而创建出来不同的 Food 实例。

这个就是 Spring 中的条件注解。

进化

条件注解还有一个进化版，那就是 Profile。我们一般利用 Profile 来实现在开发环境和生产环境之间进行快速切换。其实 Profile 就是利用条件注解来实现的。

还是刚才的例子，我们用 Profile 来稍微改造一下：

首先 Food、Rice 以及 Noodles 的定义不用变，条件注解这次我们不需要了，我们直接在 Bean 定义时添加 @Profile 注解，如下：

```
@Configuration  
public class JavaConfig {  
    @Bean("food")  
    @Profile("南方人")  
    Food rice() {  
        return new Rice();  
    }  
    @Bean("food")  
    @Profile("北方人")  
    Food noodles() {  
        return new Noodles();  
    }  
}
```

这次不需要条件注解了，取而代之的是 @Profile。然后在 Main 方法中，按照如下方式加载 Bean：

```
public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
        ctx.getEnvironment().setActiveProfiles("南方人");
        ctx.register(JavaConfig.class);
        ctx.refresh();
        Food food = (Food) ctx.getBean("food");
        System.out.println(food.showName());
    }
}
```

效果和上面的案例一样。

这样看起来 @Profile 注解貌似比 @Conditional 注解还要方便，那么 @Profile 注解到底是什么实现的呢？

我们来看一下 @Profile 的定义：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {
    String[] value();
}
```

可以看到，它也是通过条件注解来实现的。条件类是 ProfileCondition，我们来看看：

```
class ProfileCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
        MultiValueMap<String, Object> attrs =
metadata.getAllAnnotationAttributes(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if
(context.getEnvironment().acceptsProfiles(Arrays.of((String[]) value))) {
                    return true;
                }
            }
            return false;
        }
        return true;
    }
}
```

看到这里就明白了，其实还是我们在条件注解中写的那一套东西，只不过 @Profile 注解自动帮我们实现了而已。

@Profile 虽然方便，但是不够灵活，因为具体的判断逻辑不是我们自己实现的。而 @Conditional 则比较灵活。

结语

两个例子向大家展示了条件注解在 Spring 中的使用，它的一个核心思想就是当满足某种条件的时候，某个 Bean 才会生效，而正是这一特性，支撑起了 Spring Boot 的自动化配置。

好了，本文就说到这里，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南
一点雨

https 现在已经越来越普及了，特别是做一些小程序或者公众号开发的时候，https 基本上都是刚需了。

不过一个 https 证书还是挺费钱的，个人开发者可以在各个云服务提供商那里申请一个免费的证书。我印象中有效期一年，可以申请 20 个。

今天要和大家聊的是在 Spring Boot 项目中，如何开启 https 配置，为我们的接口保驾护航。

https 简介

我们先来看看什么是 https，根据 wikipedia 上的介绍：

超文本传输安全协议(HyperText Transfer Protocol Secure)，缩写：HTTPS；常称为 HTTP over TLS、HTTP over SSL 或 HTTP Secure) 是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利用 SSL/TLS 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份认证，保护交换数据的隐私与完整性。这个协议由网景公司(Netscape)在 1994 年首次提出，随后扩展到互联网上。

历史上，HTTPS 连接经常用于网络上的交易支付和企业信息系统中敏感信息的传输。在 2000 年代末至 2010 年代初，HTTPS 开始广泛使用，以确保各类型的网页真实，保护账户和保持用户通信，身份和网络浏览的私密性。

另外，还有一种安全超文本传输协议 (S-HTTP)，也是 HTTP 安全传输的一种实现，但是 HTTPS 的广泛应用而成为事实上的 HTTP 安全传输实现，S-HTTP 并没有得到广泛支持。

准备工作

首先我们需要有一个 https 证书，我们可以从各个云服务厂商处申请一个免费的，不过自己做实验没有必要这么麻烦，我们可以直接借助 Java 自带的 JDK 管理工具 keytool 来生成一个免费的 https 证书。

进入到 %JAVA_HOME%\bin 目录下，执行如下命令生成一个数字证书：

```
keytool -genkey -alias tomcathttps -keyalg RSA -keysize 2048 -keystore  
D:\javaboy.p12 -validity 365
```

命令含义如下：

- genkey 表示要创建一个新的密钥。
- alias 表示 keystore 的别名。
- keyalg 表示使用的加密算法是 RSA，一种非对称加密算法。
- keysize 表示密钥的长度。
- keystore 表示生成的密钥存放位置。
- validity 表示密钥的有效时间，单位为天。

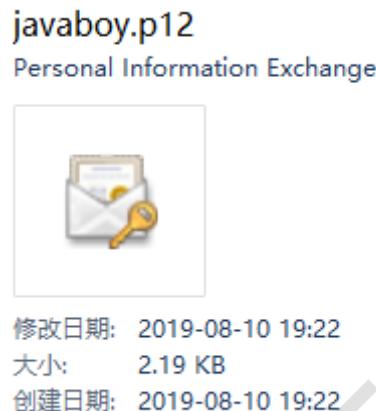
具体生成过程如下图：

```

PS C:\Program Files\Java\jdk1.8.0_171\bin> keytool -genkey -alias tomcathttps -keyalg RSA -keysize 2048 -keystore D:\javaboy.p12 -validity 365
请输入密钥库口令:
再次输入新口令:
您的名字与姓氏是什么?
[Unknown]:
您的组织单位名称是什么?
[Unknown]:
您的组织名称是什么?
[Unknown]:
您所在的市/区名称是什么?
[Unknown]:
您所在的省/市/自治区名称是什么?
[Unknown]:
该单位的双字母国家/地区代码是什么?
[Unknown]:
CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown是否正确?
[否]: 是
输入 <tomcathttps> 的密钥口令
(如果和密钥库口令相同, 按回车):
Warning:
JKS 密钥库使用专用格式。建议使用 "keytool -importkeystore -srckeystore D:\javaboy.p12 -destkeystore D:\javaboy.p12 -deststoretype pkcs12" 迁移到行业标准格式 PKCS12。
PS C:\Program Files\Java\jdk1.8.0_171\bin>

```

命令执行完成后，我们在 D 盘目录下会看到一个名为 javaboy.p12 的文件。如下图：



有了这个文件之后，我们的准备工作就算是 OK 了。

引入 https

接下来我们需要在项目中引入 https。

将上面生成的 javaboy.p12 拷贝到 Spring Boot 项目的 resources 目录下。然后在 application.properties 中添加如下配置：

```

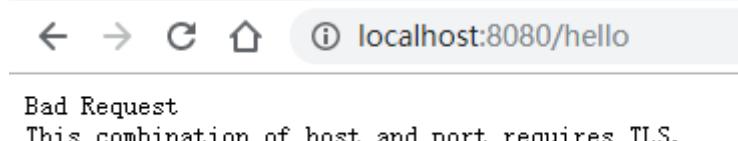
server.ssl.key-store=classpath:javaboy.p12
server.ssl.key-alias=tomcathttps
server.ssl.key-store-password=111111

```

其中：

- key-store表示密钥文件名。
- key-alias表示密钥别名。
- key-store-password就是在cmd命令执行过程中输入的密码。

配置完成后，就可以启动 Spring Boot 项目了，此时如果我们直接使用 Http 协议来访问接口，就会看到如下错误：



改用 https 来访问，结果如下：



您的连接不是私密连接

攻击者可能会试图从 **localhost** 窃取您的信息（例如：密码、通讯内容或信用卡信息）。[了解详情](#)

NET::ERR_CERT_AUTHORITY_INVALID

您可以选择向 Google 发送一些系统信息和网页内容，以帮助我们改进安全浏览功能。[隐私权政策](#)

[隐藏详情](#)

[返回安全连接](#)

此服务器无法证明它是 **localhost**；您计算机的操作系统不信任其安全证书。出现此问题的原因可能是配置有误或您的连接被拦截了。

[继续前往localhost \(不安全\)](#)

这是因为我们自己生成的 https 证书不被浏览器认可，不过没关系，我们直接点击继续访问就可以了（实际项目中只需要更换一个被浏览器认可的 https 证书即可）。



请求转发

考虑到 Spring Boot 不支持同时启动 HTTP 和 HTTPS，为了解决这个问题，我们这里可以配置一个请求转发，当用户发起 HTTP 调用时，自动转发到 HTTPS 上。

具体配置如下：

```

@Configuration
public class TomcatConfig {
    @Bean
    TomcatServletWebServerFactory tomcatServletWebServerFactory() {
        TomcatServletFactory factory = new
        TomcatServletWebServerFactory(){
            @Override
            protected void postProcessContext(Context context) {
                SecurityConstraint constraint = new SecurityConstraint();
                constraint.setUserConstraint("CONFIDENTIAL");
                SecurityCollection collection = new SecurityCollection();
                collection.addPattern("/*");
                constraint.addCollection(collection);
                context.addConstraint(constraint);
            }
        };
        factory.addAdditionalTomcatConnectors(createTomcatConnector());
    }
}

```

```
    return factory;
}

private Connector createTomcatConnector() {
    Connector connector = new
        Connector("org.apache.coyote.http11.Http11NioProtocol");
    connector.setScheme("http");
    connector.setPort(8081);
    connector.setSecure(false);
    connector.setRedirectPort(8080);
    return connector;
}
}
```

在这里，我们配置了 Http 的请求端口为 8081，所有来自 8081 的请求，将被自动重定向到 8080 这个 https 的端口上。

如此之后，我们再去访问 http 请求，就会自动重定向到 https。

结语

Spring Boot 中加入 https 其实很方便。如果你使用了 nginx 或者 tomcat 的话，https 也可以发非常方便的配置，从各个云服务厂商处申请到 https 证书之后，官方都会有一个详细的配置教程，一般照着做，就不会错了。

本文的案例，松哥已经上传到 GitHub 上了，地址：<https://github.com/lenve/javaboy-code-samples>。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



虽然现在慢慢在流行前后端分离开发，但是据松哥所了解到的，还是有一些公司在做前后端不分的开发，而在前后端不分的开发中，我们就会需要后端页面模板（实际上，即使前后端分离，也会在一些场景下需要使用页面模板，例如邮件发送模板）。

早期的 Spring Boot 中还支持使用 Velocity 作为页面模板，现在的 Spring Boot 中已经不支持 Velocity 了，页面模板主要支持 Thymeleaf 和 Freemarker，当然，作为 Java 最最基本的页面模板 Jsp，Spring Boot 也是支持的，只是使用比较麻烦。

松哥打算用三篇文章分别向大家介绍一下这三种页面模板技术。

今天我们主要来看看 Thymeleaf 在 Spring Boot 中的整合！

Thymeleaf 简介

Thymeleaf 是新一代 Java 模板引擎，它类似于 Velocity、FreeMarker 等传统 Java 模板引擎，但是与传统 Java 模板引擎不同的是，Thymeleaf 支持 HTML 原型。

它既可以让前端工程师在浏览器中直接打开查看样式，也可以让后端工程师结合真实数据查看显示效果，同时，SpringBoot 提供了 Thymeleaf 自动化配置解决方案，因此在 SpringBoot 中使用 Thymeleaf 非常方便。

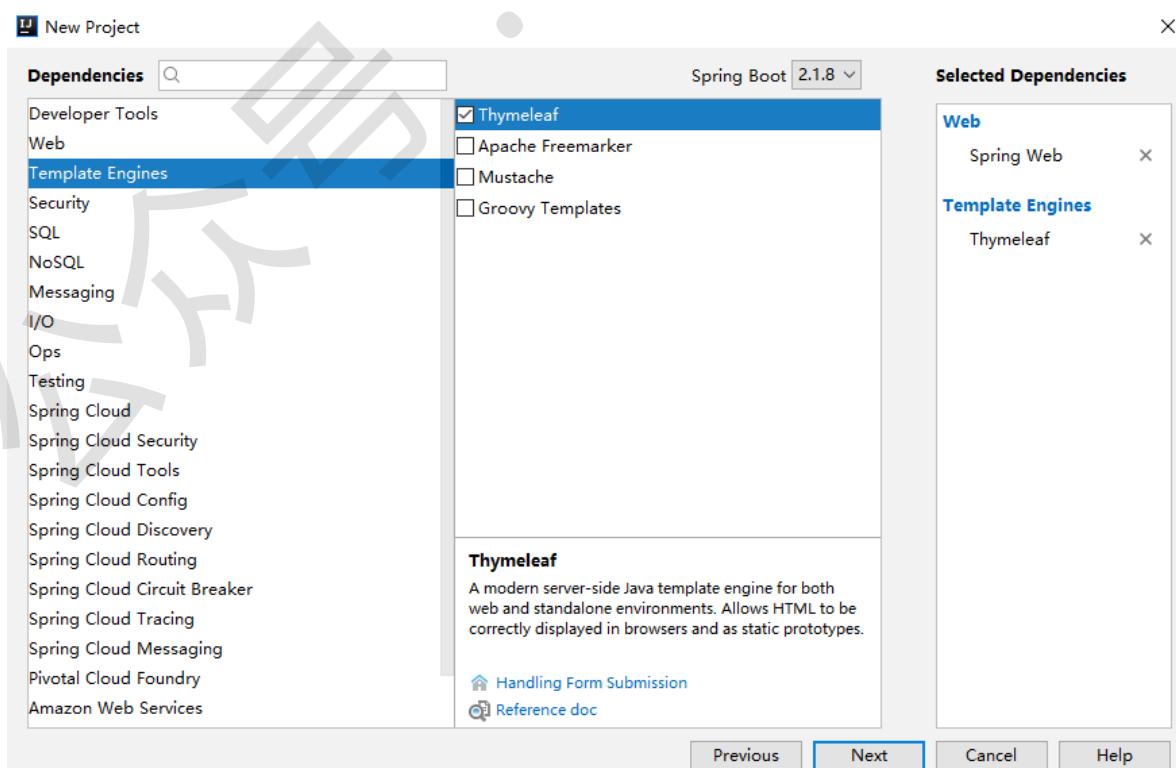
事实上，Thymeleaf 除了展示基本的 HTML，进行页面渲染之外，也可以作为一个 HTML 片段进行渲染，例如我们在做邮件发送时，可以使用 Thymeleaf 作为邮件发送模板。

另外，由于 Thymeleaf 模板后缀为 `.html`，可以直接被浏览器打开，因此，预览时非常方便。

整合

- 创建项目

Spring Boot 中整合 Thymeleaf 非常容易，只需要创建项目时添加 Thymeleaf 即可：



创建完成后，`pom.xml` 依赖如下：

```

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

```

当然，Thymeleaf 不仅仅能在 Spring Boot 中使用，也可以使用在其他地方，只不过 Spring Boot 针对 Thymeleaf 提供了一整套的自动化配置方案，这一套配置类的属性在 `org.springframework.boot.autoconfigure.thymeleaf.ThymeleafProperties` 中，部分源码如下：

```

@ConfigurationProperties(prefix = "spring.thymeleaf")
public class ThymeleafProperties {
    private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;
    public static final String DEFAULT_PREFIX = "classpath:/templates/";
    public static final String DEFAULT_SUFFIX = ".html";
    private boolean checkTemplate = true;
    private boolean checkTemplateLocation = true;
    private String prefix = DEFAULT_PREFIX;
    private String suffix = DEFAULT_SUFFIX;
    private String mode = "HTML";
    private Charset encoding = DEFAULT_ENCODING;
    private boolean cache = true;
    //...
}

```

- 首先通过 `@ConfigurationProperties` 注解，将 `application.properties` 前缀为 `spring.thymeleaf` 的配置和这个类中的属性绑定。
- 前三个 `static` 变量定义了默认的编码格式、视图解析器的前缀、后缀等。
- 从前三行配置中，可以看出来，Thymeleaf 模板的默认位置在 `resources/templates` 目录下，默认的后缀是 `html`。
- 这些配置，如果开发者不自己提供，则使用默认的，如果自己提供，则在 `application.properties` 中以 `spring.thymeleaf` 开始相关的配置。

而我们刚刚提到的，Spring Boot 为 Thymeleaf 提供的自动化配置类，则是 `org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration`，部分源码如下：

```

@Configuration
@EnableConfigurationProperties(ThymeleafProperties.class)
@ConditionalOnClass({ TemplateMode.class, SpringTemplateEngine.class })
@AutoConfigureAfter({ WebMvcAutoConfiguration.class,
    webFluxAutoConfiguration.class })
public class ThymeleafAutoConfiguration {
}

```

可以看到，在这个自动化配置类中，首先导入 `ThymeleafProperties`，然后 `@ConditionalOnClass` 注解表示当当前系统中存在 `TemplateMode` 和 `SpringTemplateEngine` 类时，当前的自动化配置类才会生效，即只要项目中引入了 `Thymeleaf` 相关的依赖，这个配置就会生效。

这些默认的配置我们几乎不需要做任何更改就可以直接使用了。如果开发者有特殊需求，则可以在 application.properties 中配置以 spring.thymeleaf 开头的属性即可。

- 创建 Controller

接下来我们就可以创建 Controller 了，实际上引入 Thymeleaf 依赖之后，我们可以不做任何配置。新建的 IndexController 如下：

```
@Controller
public class IndexController {
    @GetMapping("/index")
    public String index(Model model) {
        List<User> users = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            User u = new User();
            u.setId((long) i);
            u.setName("javaboy:" + i);
            u.setAddress("深圳:" + i);
            users.add(u);
        }
        model.addAttribute("users", users);
        return "index";
    }
}
public class User {
    private Long id;
    private String name;
    private String address;
    //省略 getter/setter
}
```

在 IndexController 中返回逻辑视图名+数据，逻辑视图名为 index，意思我们需要在 resources/templates 目录下提供一个名为 index.html 的 Thymeleaf 模板文件。

- 创建 Thymeleaf

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <table border="1">
        <tr>
            <td>编号</td>
            <td>用户名</td>
            <td>地址</td>
        </tr>
        <tr th:each="user : ${users}">
            <td th:text="${user.id}"></td>
            <td th:text="${user.name}"></td>
            <td th:text="${user.address}"></td>
        </tr>
    </table>
</body>
</html>
```

在 Thymeleaf 中，通过 `th:each` 指令来遍历一个集合，数据的展示通过 `th:text` 指令来实现，注意 `index.html` 最上面要引入 `thymeleaf` 名称空间。

配置完成后，就可以启动项目了，访问 /index 接口，就能看到集合中的数据了：



编号	用户名	地址
0	javaboy:0	深圳:0
1	javaboy:1	深圳:1
2	javaboy:2	深圳:2
3	javaboy:3	深圳:3
4	javaboy:4	深圳:4
5	javaboy:5	深圳:5
6	javaboy:6	深圳:6
7	javaboy:7	深圳:7
8	javaboy:8	深圳:8
9	javaboy:9	深圳:9

另外，Thymeleaf 支持在 js 中直接获取 Model 中的变量。例如，在 `IndexController` 中有一个变量 `username`：

```
@Controller
public class IndexController {
    @GetMapping("/index")
    public String index(Model model) {
        model.addAttribute("username", "李四");
        return "index";
    }
}
```

在页面模板中，可以直接在 js 中获取到这个变量：

```
<script th:inline="javascript">
    var username = [[${username}]];
    console.log(username)
</script>
```

这个功能算是 Thymeleaf 的特色之一吧。

手动渲染

前面我们说的是返回一个 Thymeleaf 模板，我们也可以手动渲染 Thymeleaf 模板，这个一般在邮件发送时候有用，例如我在 resources/templates 目录下新建一个邮件模板，如下：

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

```

<title>Title</title>
</head>
<body>
<p>hello 欢迎 <span th:text="${username}"></span>加入 xxx 集团，您的入职信息如下：
</p>
<table border="1">
<tr>
    <td>职位</td>
    <td th:text="${position}"></td>
</tr>
<tr>
    <td>薪水</td>
    <td th:text="${salary}"></td>
</tr>
</table>

</body>
</html>

```

这一个 HTML 模板中，有几个变量，我们要将这个 HTML 模板渲染成一个 String 字符串，再把这个字符串通过邮件发送出去，那么如何手动渲染呢？

```

@Autowired
TemplateEngine templateEngine;
@Test
public void test1() throws MessagingException {
    Context context = new Context();
    context.setVariable("username", "javaboy");
    context.setVariable("position", "Java工程师");
    context.setVariable("salary", 99999);
    String mail = templateEngine.process("mail", context);
    //省略邮件发送
}

```

1. 渲染时，我们需要首先注入一个 TemplateEngine 对象，这个对象就是在 Thymeleaf 的自动化配置类中配置的（即当我们引入 Thymeleaf 的依赖之后，这个实例就有了）。
2. 然后构造一个 Context 对象用来存放变量。
3. 调用 process 方法进行渲染，该方法的返回值就是渲染后的 HTML 字符串，然后我们将这个字符串发送出去。

这是 Spring Boot 整合 Thymeleaf 的几个关键点，关于 Thymeleaf 这个页面模板本身更多的用法，大家可以参考 Thymeleaf 的文档：<https://www.thymeleaf.org>。

总结

本文主要向大家简单介绍了 Spring Boot 和 Thymeleaf 整合时的几个问题，还是比较简单的，大家可以阅读 Thymeleaf 官方文档学习 Thymeleaf 的更多用法。本文案例我已上传到 GitHub：<https://github.com/lenve/javaboy-code-samples>

关于本文，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



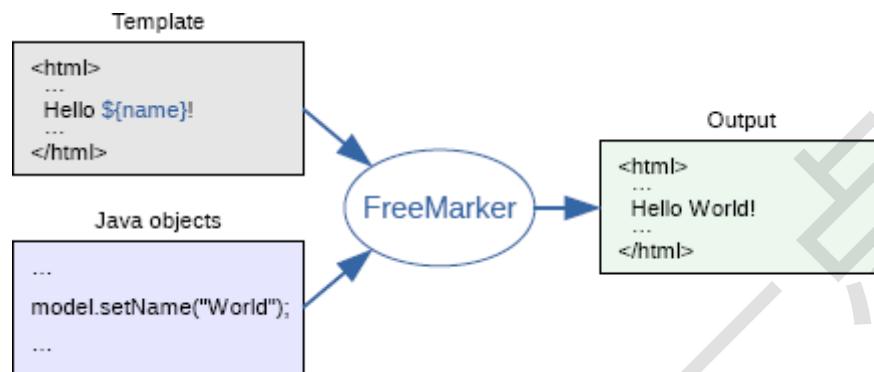
云·空间·时间

今天来聊聊 Spring Boot 整合 Freemarker。

Freemarker 简介

这是一个相当老牌的开源的免费的模版引擎。通过 Freemarker 模版，我们可以将数据渲染成 HTML 网页、电子邮件、配置文件以及源代码等。Freemarker 不是面向最终用户的，而是一个 Java 类库，我们可以将之作为一个普通的组件嵌入到我们的产品中。

来看一张来自 Freemarker 官网的图片：



可以看到，Freemarker 可以将模版和数据渲染成 HTML。

Freemarker 模版后缀为 `.ftl` (FreeMarker Template Language)。FTL 是一种简单的、专用的语言，它不是像 Java 那样成熟的编程语言。在模板中，你可以专注于如何展现数据，而在模板之外可以专注于要展示什么数据。

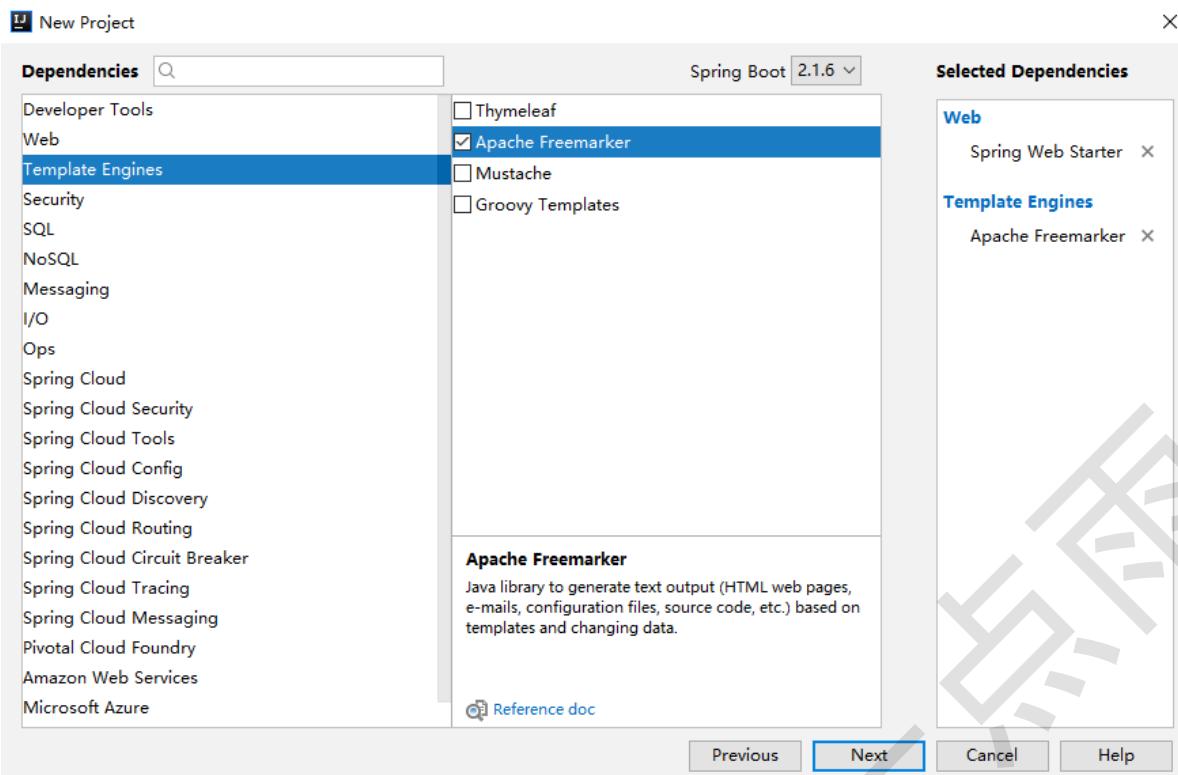
好了，这是一个简单的介绍，接下来我们来看看 Freemarker 和 Spring Boot 的一个整合操作。

实践

在 SSM 中整合 Freemarker，所有的配置文件加起来，前前后后大约在 50 行左右，Spring Boot 中要几行配置呢？0 行！

1. 创建工程

首先创建一个 Spring Boot 工程，引入 Freemarker 依赖，如下图：



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

工程创建完成后，在 `org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration` 类中，可以看到关于 Freemarker 的自动化配置：

```
@Configuration
@ConditionalOnClass({ freemarker.template.Configuration.class,
FreeMarkerConfigurationFactory.class })
@EnableConfigurationProperties(FreeMarkerProperties.class)
@Import({ FreeMarkerServletWebConfiguration.class,
FreeMarkerReactiveWebConfiguration.class,
FreeMarkerNonWebConfiguration.class })
public class FreeMarkerAutoConfiguration { }
```

从这里可以看出，当 `classpath` 下存在 `freemarker.template.Configuration` 以及 `FreeMarkerConfigurationFactory` 时，配置才会生效，也就是说当我们引入了 `Freemarker` 之后，配置就会生效。但是这里的自动化配置只做了模板位置检查，其他配置则是在导入的 `FreeMarkerServletWebConfiguration` 配置中完成的。那么我们再来看看 `FreeMarkerServletWebConfiguration` 类，部分源码如下：

```
@Configuration
@ConditionalOnWebApplication(type = ConditionalOnWebApplication.Type.SERVLET)
@ConditionalOnClass({ Servlet.class, FreeMarkerConfigurer.class })
```

```

@AutoConfigureAfter(WebMvcAutoConfiguration.class)
class FreeMarkerServletWebConfiguration extends AbstractFreeMarkerConfiguration
{
    protected FreeMarkerServletWebConfiguration(FreeMarkerProperties
properties) {
        super(properties);
    }
    @Bean
    @ConditionalOnMissingBean(FreeMarkerConfig.class)
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        applyProperties(configurer);
        return configurer;
    }
    @Bean
    @ConditionalOnMissingBean(name = "freeMarkerViewResolver")
    @ConditionalOnProperty(name = "spring.freemarker.enabled",
matchIfMissing = true)
    public FreeMarkerViewResolver freeMarkerViewResolver() {
        FreeMarkerViewResolver resolver = new FreeMarkerViewResolver();
        getProperties().applyToMvcViewResolver(resolver);
        return resolver;
    }
}

```

我们来简单看下这段源码：

1. @ConditionalOnWebApplication 表示当前配置在 web 环境下才会生效。
2. ConditionalOnClass 表示当前配置在存在 Servlet 和 FreeMarkerConfigurer 时才会生效。
3. @AutoConfigureAfter 表示当前自动化配置在 WebMvcAutoConfiguration 之后完成。
4. 代码中，主要提供了 FreeMarkerConfigurer 和 FreeMarkerViewResolver。
5. FreeMarkerConfigurer 是 Freemarker 的一些基本配置，例如 templateLoaderPath、defaultEncoding 等
6. FreeMarkerViewResolver 则是视图解析器的基本配置，包含了viewClass、suffix、allowRequestOverride、allowSessionOverride 等属性。

另外还有一点，在这个类的构造方法中，注入了 FreeMarkerProperties：

```

@ConfigurationProperties(prefix = "spring.freemarker")
public class FreeMarkerProperties extends AbstractTemplateViewResolverProperties
{
    public static final String DEFAULT_TEMPLATE_LOADER_PATH =
"classpath:/templates/";
    public static final String DEFAULT_PREFIX = "";
    public static final String DEFAULT_SUFFIX = ".ftl";
    /**
     * well-known FreeMarker keys which are passed to FreeMarker's
Configuration.
    */
    private Map<String, String> settings = new HashMap<>();
}

```

FreeMarkerProperties 中则配置了 Freemarker 的基本信息，例如模板位置在 `classpath:/templates/`，再例如模板后缀为 `.ftl`，那么这些配置我们以后都可以在 `application.properties` 中进行修改。

如果我们在 SSM 的 XML 文件中自己配置 FreeMarker，也不过就是配置这些东西。现在，这些配置由 FreeMarkerServletWebConfiguration 帮我们完成了。

2. 创建类

首先我们来创建一个 User 类，如下：

```
public class User {  
    private Long id;  
    private String username;  
    private String address;  
    //省略 getter/setter  
}
```

再来创建 UserController：

```
@Controller  
public class UserController {  
    @GetMapping("/index")  
    public String index(Model model) {  
        List<User> users = new ArrayList<>();  
        for (int i = 0; i < 10; i++) {  
            User user = new User();  
            user.setId((long) i);  
            user.setUsername("javaboy>>>" + i);  
            user.setAddress("www.javaboy.org>>>" + i);  
            users.add(user);  
        }  
        model.addAttribute("users", users);  
        return "index";  
    }  
}
```

最后在 freemarker 中渲染数据：

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Title</title>  
</head>  
<body>  
    <table border="1">  
        <tr>  
            <td>用户编号</td>  
            <td>用户名称</td>  
            <td>用户地址</td>  
        </tr>  
        <#list users as user>  
            <tr>  
                <td>${user.id}</td>  
                <td>${user.username}</td>  
                <td>${user.address}</td>  
            </tr>  
        </#list>  
    </table>
```

```
</body>
</html>
```

运行效果如下：



The screenshot shows a web browser window with the URL "localhost:8080/index". The page displays a table with three columns: "用户编号" (User ID), "用户名" (User Name), and "用户地址" (User Address). The data is as follows:

用户编号	用户名	用户地址
0	javaboy>>>0	www.javaboy.org>>>0
1	javaboy>>>1	www.javaboy.org>>>1
2	javaboy>>>2	www.javaboy.org>>>2
3	javaboy>>>3	www.javaboy.org>>>3
4	javaboy>>>4	www.javaboy.org>>>4
5	javaboy>>>5	www.javaboy.org>>>5
6	javaboy>>>6	www.javaboy.org>>>6
7	javaboy>>>7	www.javaboy.org>>>7
8	javaboy>>>8	www.javaboy.org>>>8
9	javaboy>>>9	www.javaboy.org>>>9

其他配置

如果我们要修改模版文件位置等，可以在 application.properties 中进行配置：

```
spring.freemarker.allow-request-override=false
spring.freemarker.allow-session-override=false
spring.freemarker.cache=false
spring.freemarker.charset=UTF-8
spring.freemarker.check-template-location=true
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=false
spring.freemarker.expose-session-attributes=false
spring.freemarker.suffix=.ftl
spring.freemarker.template-loader-path=classpath:/templates/
```

配置文件按照顺序依次解释如下：

1. HttpServletRequest的属性是否可以覆盖controller中model的同名项
2. HttpSession的属性是否可以覆盖controller中model的同名项
3. 是否开启缓存
4. 模板文件编码
5. 是否检查模板位置
6. Content-Type的值
7. 是否将HttpServletRequest中的属性添加到Model中
8. 是否将HttpSession中的属性添加到Model中
9. 模板文件后缀
10. 模板文件位置

好了，整合完成之后，Freemarker 的更多用法，就和在 SSM 中使用 Freemarker 一样了，这里我就不再赘述。

结语

本文和大家简单聊一聊 Spring Boot 整合 Freemarker，算是对 Spring Boot2 教程的一个补充（后面还会有一些补充），有问题欢迎留言讨论。

本项目案例，我已经上传到 GitHub 上，欢迎大家 star: <https://github.com/lenvve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



当我们使用 SpringMVC 框架时，静态资源会被拦截，需要添加额外配置，之前老有小伙伴在微信上问松哥 Spring Boot 中的静态资源加载问题：“松哥，我的 HTML 页面好像没有样式？”，今天我就通过一篇文章，来和大伙仔细聊一聊这个问题。

1. SSM 中的配置

要讲 Spring Boot 中的问题，我们得先回到 SSM 环境搭建中，一般来说，我们可以通过 `<mvc:resources />` 节点来配置不拦截静态资源，如下：

```
<mvc:resources mapping="/js/**" location="/js/">
<mvc:resources mapping="/css/**" location="/css/">
<mvc:resources mapping="/html/**" location="/html/">
```

由于这是一种Ant风格的路径匹配符，`/**` 表示可以匹配任意层级的路径，因此上面的代码也可以像下面这样简写：

```
<mvc:resources mapping="/**" location="/">
```

这种配置是在 XML 中的配置，大家知道，SpringMVC 的配置除了在 XML 中配置，也可以在 Java 代码中配置，如果在 Java 代码中配置的话，我们只需要自定义一个类，继承自 `WebMvcConfigurationSupport` 即可：

```
@Configuration
@ComponentScan(basePackages = "org.javaboy.javassm")
public class SpringMVConfig extends WebMvcConfigurationSupport {
    @Override
    protected void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/**").addResourceLocations("/");
    }
}
```

重写 `WebMvcConfigurationSupport` 类中的 `addResourceHandlers` 方法，在该方法中配置静态资源位置即可，这里的含义和上面 XML 配置的含义一致，因此无需多说。

这是我们传统的解决方案，在 Spring Boot 中，其实配置方式和这个一脉相承，只是有一些自动化的配置了。

2. Spring Boot 中的配置

在 Spring Boot 中，如果我们是从 <https://start.spring.io> 这个网站上创建的项目，或者使用 IntelliJ IDEA 中的 Spring Boot 初始化工具创建的项目，默认都会存在 `resources/static` 目录，很多小伙伴也知道静态资源只要放到这个目录下，就可以直接访问，除了这里还有没有其他可以放静态资源的位置呢？为什么放在这里就能直接访问了呢？这就是本文要讨论的问题了。

2.1 整体规划

首先，在 Spring Boot 中，默认情况下，一共有 5 个位置可以放静态资源，五个路径分别是如下 5 个：

1. `classpath:/META-INF/resources/`
2. `classpath:/resources/`
3. `classpath:/static/`

4. classpath:/public/

5. /

前四个目录好理解，分别对应了 resources 目录下不同的目录，第 5 个 / 是啥意思呢？我们知道，在 Spring Boot 项目中，默认是没有 webapp 这个目录的，当然我们也可以自己添加（例如在需要使用 JSP 的时候），这里第 5 个 / 其实就是表示 webapp 目录中的静态资源也不被拦截。如果同一个文件分别出现在五个目录下，那么优先级也是按照上面列出的顺序。

不过，虽然有 5 个存储目录，除了第 5 个用的比较少之外，其他四个，系统默认创建了 classpath:/static/，正常情况下，我们只需要将我们的静态资源放到这个目录下即可，也不需要额外去创建其他静态资源目录，例如我在 classpath:/static/ 目录下放了一张名为 1.png 的图片，那么我的访问路径是：

```
http://localhost:8080/1.png
```

这里大家注意，请求地址中并不需要 static，如果加上了 static 反而多此一举会报 404 错误。很多人会觉得奇怪，为什么不需要添加 static 呢？资源明明放在 static 目录下。其实这个效果很好实现，例如在 SSM 配置中，我们的静态资源拦截配置如果是下面这样：

```
<mvc:resources mapping="/**" location="/static/" />
```

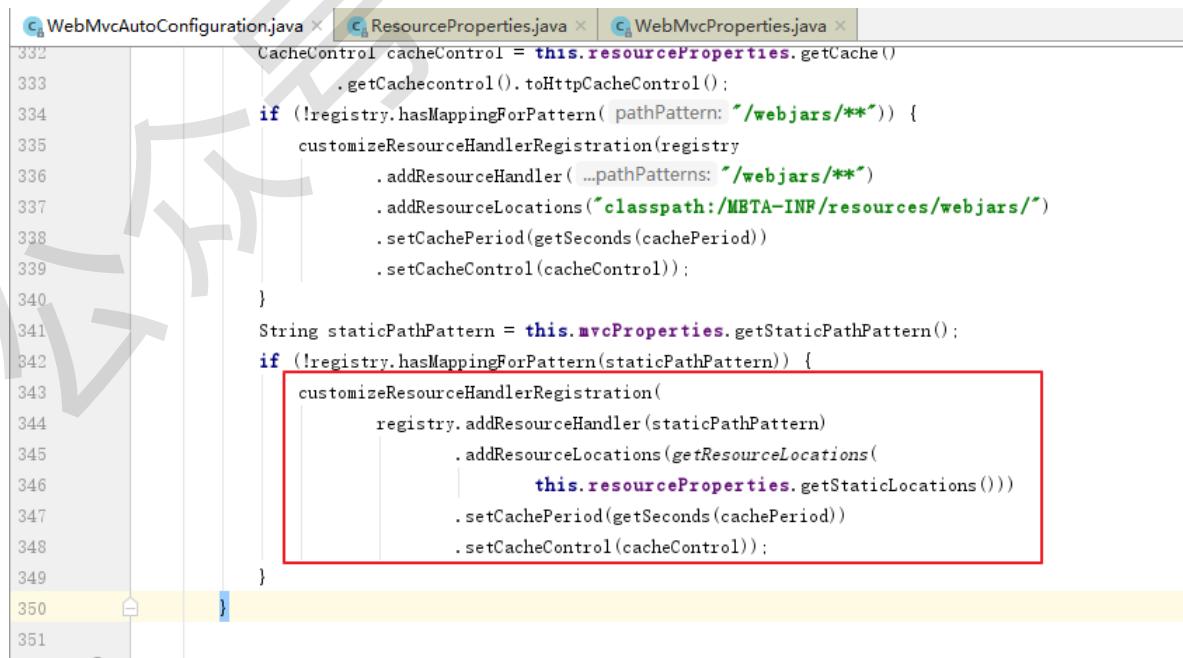
如果我们是这样配置的话，请求地址如果是 `http://localhost:8080/1.png` 实际上系统会去 `/static/1.png` 目录下查找相关的文件。

所以我们理所当然的猜测，在 Spring Boot 中可能也是类似的配置。

2.2 源码解读

胡适之先生说：“大胆猜想，小心求证”，我们这里就通过源码解读来看看 Spring Boot 中的静态资源到底是怎么配置的。

首先我们在 WebMvcAutoConfiguration 类中看到了 SpringMVC 自动化配置的相关的内容，找到了静态资源拦截的配置，如下：



```
1 WebMvcAutoConfiguration.java x 2 ResourceProperties.java x 3 WebMvcProperties.java x
332 CacheControl cacheControl = this.resourceProperties.getCache()
333 .getCachecontrol().toHttpCacheControl();
334 if (!registry.hasMappingForPattern(pathPattern: "/webjars/**")) {
335     customizeResourceHandlerRegistration(registry
336         .addResourceHandler(...pathPatterns: "/webjars/**")
337         .addResourceLocations("classpath:/META-INF/resources/webjars/")
338         .setCachePeriod(getSeconds(cachePeriod))
339         .setCacheControl(cacheControl));
340 }
341 String staticPathPattern = this.mvcProperties.getStaticPathPattern();
342 if (!registry.hasMappingForPattern(staticPathPattern)) {
343     customizeResourceHandlerRegistration(
344         registry.addResourceHandler(staticPathPattern)
345             .addResourceLocations(getResourceLocations(
346                 this.resourceProperties.getStaticLocations()))
347             .setCachePeriod(getSeconds(cachePeriod))
348             .setCacheControl(cacheControl));
349 }
350 }
```

可以看到这里静态资源的定义和我们前面提到的 Java 配置 SSM 中的配置非常相似，其中，`this.mvcProperties.getStaticPathPattern()` 方法对应的值是 `/**`，`this.resourceProperties.getStaticLocations()` 方法返回了四个位置，分别是：

- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public/

然后在 `getResourcesLocations` 方法中，又添加了 `/`，因此这里返回值一共有 5 个。其中，`/` 表示 `webapp` 目录，即 `webapp` 中的静态文件也可以直接访问。静态资源的匹配路径按照定义路径优先级依次降低。因此这里的配置和我们前面提到的如出一辙。这样大伙就知道了为什么 `Spring Boot` 中支持 5 个静态资源位置，同时也明白了为什么静态资源请求路径中不需要 `/static`，因为在路径映射中已经自动的添加上了 `/static` 了。

2.3 自定义配置

当然，这个是系统默认配置，如果我们并不想将资源放在系统默认的这五个位置上，也可以自定义静态资源位置和映射，自定义的方式也有两种，可以通过 `application.properties` 来定义，也可以在 Java 代码中来定义，下面分别来看。

2.3.1 application.properties

在配置文件中定义的方式比较简单，如下：

```
spring.resources.static-locations=classpath:/  
spring.mvc.static-path-pattern=/**
```

第一行配置表示定义资源位置，第二行配置表示定义请求 URL 规则。以上文的配置为例，如果我们这样定义了，表示可以将静态资源放在 `resources` 目录下的任意地方，我们访问的时候当然也需要写完整的路径，例如在 `resources/static` 目录下有一张名为 `1.png` 的图片，那么访问路径就是 `http://localhost:8080/static/1.png`，注意此时的 `static` 不能省略。

2.3.2 Java 代码定义

当然，在 `Spring Boot` 中我们也可以通过 Java 代码来自定义，方式和 Java 配置的 SSM 比较类似，如下：

```
@Configuration  
public class WebMvcConfig implements WebMvcConfigurer {  
    @Override  
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
  
        registry.addResourceHandler("/**").addResourceLocations("classpath:/aaa/");  
    }  
}
```

这里代码基本和前面一致，比较简单，不再赘述。

3. 总结

这里需要提醒大家的是，松哥见到有很多人用了 `Thymeleaf` 之后，会将静态资源也放在 `resources/templates` 目录下，注意，`templates` 目录并不是静态资源目录，它是一个放页面模板的位置（你看到的 `Thymeleaf` 模板虽然后缀为 `.html`，其实并不是静态资源）。好了，通过上面的讲解，相信大家对 `Spring Boot` 中静态资源的位置有一个深刻了解了，应该不会再在项目中出错了吧！

关注微信公众号江南一点雨，回复 `2TB`，获取超 `2TB` 免费 `Java` 学习资源。



云·空间·时间

严格来说，本文并不算是 Spring Boot 中的知识点，但是很多学过 SpringMVC 的小伙伴，对于 @ControllerAdvice 却并不熟悉，Spring Boot 和 SpringMVC 一脉相承，@ControllerAdvice 在 Spring Boot 中也有广泛的使用场景，因此本文我们就来聊一聊这个问题。

@ControllerAdvice，很多初学者可能都没有听说过这个注解，实际上，这是一个非常有用的注解，顾名思义，这是一个增强的 Controller。使用这个 Controller，可以实现三个方面的功能：

1. 全局异常处理
2. 全局数据绑定
3. 全局数据预处理

灵活使用这三个功能，可以帮助我们简化很多工作，需要注意的是，这是 SpringMVC 提供的功能，在 Spring Boot 中可以直接使用，下面分别来看。

全局异常处理

使用 @ControllerAdvice 实现全局异常处理，只需要定义类，添加该注解即可定义方式如下：

```
@ControllerAdvice
public class MyGlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ModelAndView customException(Exception e) {
        ModelAndView mv = new ModelAndView();
        mv.addObject("message", e.getMessage());
        mv.setViewName("myerror");
        return mv;
    }
}
```

在该类中，可以定义多个方法，不同的方法处理不同的异常，例如专门处理空指针的方法、专门处理数组越界的方法...，也可以直接向上面代码一样，在一个方法中处理所有的异常信息。

@ExceptionHandler 注解用来指明异常的处理类型，即如果这里指定为 NullpointerException，则数组越界异常就不会进入到这个方法中来。

全局数据绑定

全局数据绑定功能可以用来做一些初始化的数据操作，我们可以将一些公共的数据定义在添加了 @ControllerAdvice 注解的类中，这样，在每一个 Controller 的接口中，就都能够访问导致这些数据。

使用步骤，首先定义全局数据，如下：

```
@ControllerAdvice
public class MyGlobalExceptionHandler {
    @ModelAttribute(name = "md")
    public Map<String, Object> mydata() {
        HashMap<String, Object> map = new HashMap<>();
        map.put("age", 99);
        map.put("gender", "男");
        return map;
    }
}
```

使用 @ModelAttribute 注解标记该方法的返回数据是一个全局数据，默认情况下，这个全局数据的 key 就是返回的变量名，value 就是方法返回值，当然开发者可以通过 @ModelAttribute 注解的 name 属性去重新指定 key。

定义完成后，在任何一个Controller 的接口中，都可以获取到这里定义的数据：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(Model model) {
        Map<String, Object> map = model.asMap();
        System.out.println(map);
        int i = 1 / 0;
        return "hello controller advice";
    }
}
```

全局数据预处理

考虑我有两个实体类，Book 和 Author，分别定义如下：

```
public class Book {
    private String name;
    private Long price;
    //getter/setter
}
public class Author {
    private String name;
    private Integer age;
    //getter/setter
}
```

此时，如果我定义一个数据添加接口，如下：

```
@PostMapping("/book")
public void addBook(Book book, Author author) {
    System.out.println(book);
    System.out.println(author);
}
```

这个时候，添加操作就会有问题，因为两个实体类都有一个 name 属性，从前端传递时，无法区分。
此时，通过 @ControllerAdvice 的全局数据预处理可以解决这个问题

解决步骤如下：

1.给接口中的变量取别名

```
@PostMapping("/book")
public void addBook(@ModelAttribute("b") Book book, @ModelAttribute("a") Author
author) {
    System.out.println(book);
    System.out.println(author);
}
```

2. 进行请求数据预处理

在 @ControllerAdvice 标记的类中添加如下代码:

```
@InitBinder("b")
public void b(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("b.");
}
@InitBinder("a")
public void a(WebDataBinder binder) {
    binder.setFieldDefaultPrefix("a.");
}
```

@InitBinder("b") 注解表示该方法用来处理和Book和相关的参数,在方法中,给参数添加一个 b 前缀,即请求参数要有b前缀.

3. 发送请求

请求发送时,通过给不同对象的参数添加不同的前缀,可以实现参数的区分.

The screenshot shows the Postman interface with a POST request to `http://127.0.0.1:8080/book?b.name=三国演义&b.price=99&a.name=罗贯中&a.age=100`. The 'Params' tab is selected, displaying the following table:

KEY	VALUE	DESCRIPTION
b.name	三国演义	
b.price	99	
a.name	罗贯中	
a.age	100	

总结

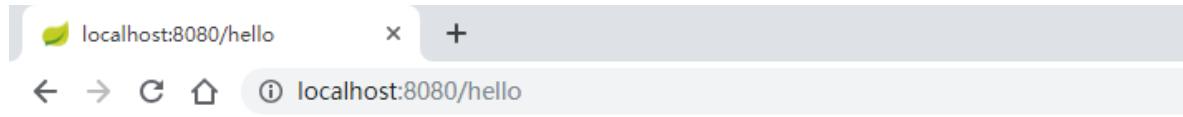
这就是松哥给大伙介绍的 @ControllerAdvice 的几个简单用法, 这些点既可以在传统的SSM项目中使用, 也可以在 Spring Boot + Spring Cloud 微服务中使用, 欢迎大家有问题一起讨论。

关注微信公众号江南一点雨, 回复 2TB, 获取超 2TB 免费 Java 学习资源。



在 Spring Boot 项目中，异常统一处理，可以使用 Spring 中 @ControllerAdvice 来统一处理，也可以自己来定义异常处理方案。Spring Boot 中，对异常的处理有一些默认的策略，我们分别来看。

默认情况下，Spring Boot 中的异常页面是这样的：



Whitelabel Error Page

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

Sat Apr 13 11:06:54 CST 2019

There was an unexpected error (type=Internal Server Error, status=500).

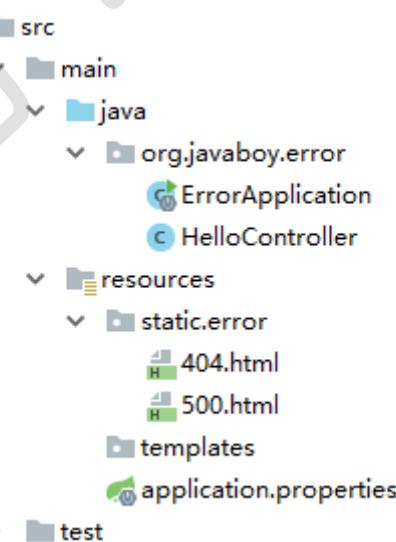
/ by zero

我们从这个异常提示中，也能看出来，之所以用户看到这个页面，是因为开发者没有明确提供一个 /error 路径，如果开发者提供了 /error 路径，这个页面就不会展示出来，不过在 Spring Boot 中，提供 /error 路径实际上是下下策，Spring Boot 本身在处理异常时，也是当所有条件都不满足时，才会去找 /error 路径。那么我们就先来看看，在 Spring Boot 中，如何自定义 error 页面，整体上来说，可以分为两种，一种是静态页面，另一种是动态页面。

静态异常页面

自定义静态异常页面，又分为两种，第一种是使用 HTTP 响应码来命名页面，例如 404.html、405.html、500.html，另一种就是直接定义一个 4xx.html，表示 400-499 的状态都显示这个异常页面，5xx.html 表示 500-599 的状态显示这个异常页面。

默认是在 `classpath:/static/error/` 路径下定义相关页面：



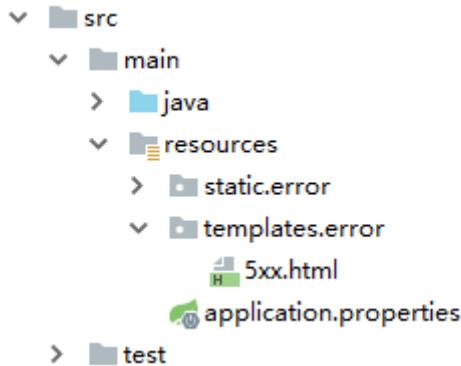
此时，启动项目，如果项目抛出 500 请求错误，就会自动展示 500.html 这个页面，发生 404 就会展示 404.html 页面。如果异常展示页面既存在 5xx.html，也存在 500.html，此时，发生 500 异常时，优先展示 500.html 页面。

动态异常页面

动态的异常页面定义方式和静态的基本一致，可以采用的页面模板有 jsp、freemarker、thymeleaf。动态异常页面，也支持 404.html 或者 4xx.html，但是一般来说，由于动态异常页面可以直接展示异常详细信息，所以就没有必要挨个枚举错误了，直接定义 4xx.html（这里使用thymeleaf模板）或者 5xx.html 即可。

注意，动态页面模板，不需要开发者自己去定义控制器，直接定义异常页面即可，Spring Boot 中自带的异常处理器会自动查找到异常页面。

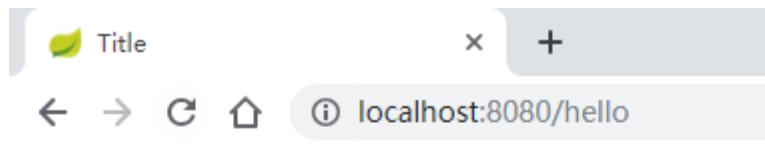
页面定义如下：



页面内容如下：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>5xx</h1>
    <table border="1">
        <tr>
            <td>path</td>
            <td th:text="${path}"></td>
        </tr>
        <tr>
            <td>error</td>
            <td th:text="${error}"></td>
        </tr>
        <tr>
            <td>message</td>
            <td th:text="${message}"></td>
        </tr>
        <tr>
            <td>timestamp</td>
            <td th:text="${timestamp}"></td>
        </tr>
        <tr>
            <td>status</td>
            <td th:text="${status}"></td>
        </tr>
    </table>
</body>
</html>
```

默认情况下，完整的异常信息就是这5条，展示效果如下：



5xx

path	/hello
error	Internal Server Error
message	/ by zero
timestamp	Sat Apr 13 11:30:03 CST 2019
status	500

如果动态页面和静态页面同时定义了异常处理页面，例如 `classpath:/static/error/404.html` 和 `classpath:/templates/error/404.html` 同时存在时，默认使用动态页面。即完整的错误页面查找方式应该是这样：

发生了 500 错误-->查找动态 500.html 页面-->查找静态 500.html --> 查找动态 5xx.html-->查找静态 5xx.html。

自定义异常数据

默认情况下，在 Spring Boot 中，所有的异常数据其实就是上文所展示出来的 5 条数据，这 5 条数据定义在 `org.springframework.boot.web.reactive.error.DefaultErrorAttributes` 类中，具体定义在 `getErrorAttributes` 方法中：

```
@Override
public Map<String, Object> getErrorAttributes(ServerRequest request,
    boolean includeStackTrace) {
    Map<String, Object> errorAttributes = new LinkedHashMap<>();
    errorAttributes.put("timestamp", new Date());
    errorAttributes.put("path", request.path());
    Throwable error = getError(request);
    HttpStatus errorStatus = determineHttpStatus(error);
    errorAttributes.put("status", errorStatus.value());
    errorAttributes.put("error", errorStatus.getReasonPhrase());
    errorAttributes.put("message", determineMessage(error));
    handleException(errorAttributes, determineException(error),
        includeStackTrace);
    return errorAttributes;
}
```

`DefaultErrorAttributes` 类本身则是在 `org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration` 异常自动配置类中定义的，如果开发者没有自己提供一个 `ErrorAttributes` 的实例的话，那么 Spring Boot 将自动提供一个 `ErrorAttributes` 的实例，也就是 `DefaultErrorAttributes`。

基于此，开发者自定义 `ErrorAttributes` 有两种方式：

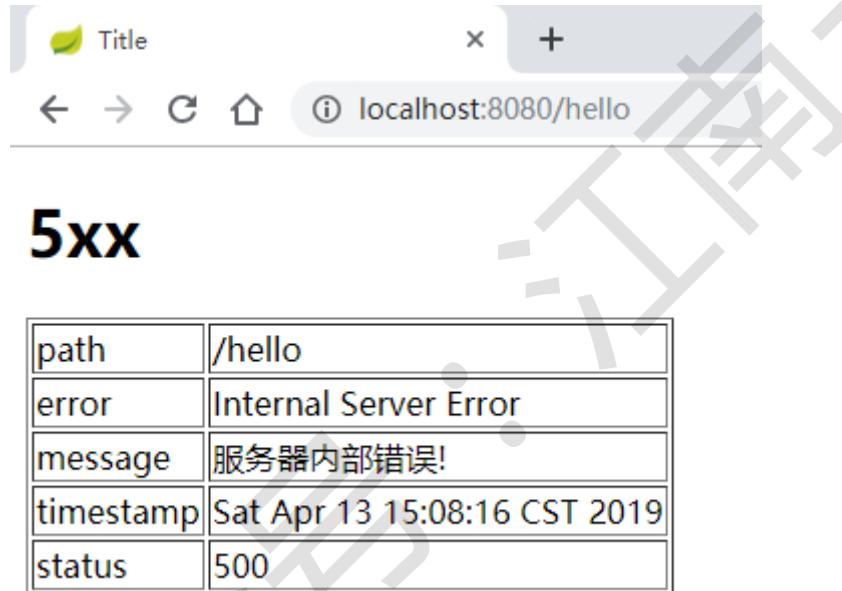
1. 直接实现 `ErrorAttributes` 接口

2. 继承 DefaultErrorAttributes (推荐) , 因为 DefaultErrorAttributes 中对异常数据的处理已经完成，开发者可以直接使用。

具体定义如下：

```
@Component
public class MyErrorAttributes extends DefaultErrorAttributes {
    @Override
    public Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean
includeStackTrace) {
        Map<String, Object> map = super.getErrorAttributes(webRequest,
includeStackTrace);
        if ((Integer)map.get("status") == 500) {
            map.put("message", "服务器内部错误!");
        }
        return map;
    }
}
```

定义好的 ErrorAttributes 一定要注册成一个 Bean , 这样, Spring Boot 就不会使用默认的 DefaultErrorAttributes 了, 运行效果如下图:



自定义异常视图

异常视图默认就是前面所说的静态或者动态页面, 这个也是可以自定义的, 首先, 默认的异常视图加载逻辑在 org.springframework.boot.autoconfigure.web.servlet.error.BasicErrorController 类的 errorHtml 方法中, 这个方法用来返回异常页面+数据, 还有另外一个 error 方法, 这个方法用来返回异常数据 (如果是 ajax 请求, 则该方法会被触发) 。

```

@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request,
                             HttpServletResponse response) {
    HttpStatus status = getStatus(request);
    Map<String, Object> model =
        collections.unmodifiableMap(getErrorAttributes(
            request, isIncludeStackTrace(request,
                MediaType.TEXT_HTML)));
    response.setStatus(status.value());
    ModelAndView modelAndView = resolveErrorView(request, response, status,
        model);
    return (modelAndView != null) ? modelAndView : new ModelAndView("error",
        model);
}

```

在该方法中，首先会通过 `getErrorAttributes` 方法去获取异常数据（实际上会调用到 `ErrorAttributes` 的实例的 `getErrorAttributes` 方法），然后调用 `resolveErrorView` 去创建一个 `ModelAndView`，如果这里创建失败，那么用户将会看到默认的错误提示页面。

正常情况下，`resolveErrorView` 方法会来到 `DefaultErrorViewResolver` 类的 `resolveErrorView` 方法中：

```

@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status,
        Map<String, Object> model) {
    ModelAndView modelAndView = resolve(String.valueOf(status.value()),
        model);
    if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
        modelAndView = resolve(SERIES_VIEWS.get(status.series()),
            model);
    }
    return modelAndView;
}

```

在这里，首先以异常响应码作为视图名分别去查找动态页面和静态页面，如果没有查找到，则再以 4xx 或者 5xx 作为视图名再去分别查找动态或者静态页面。

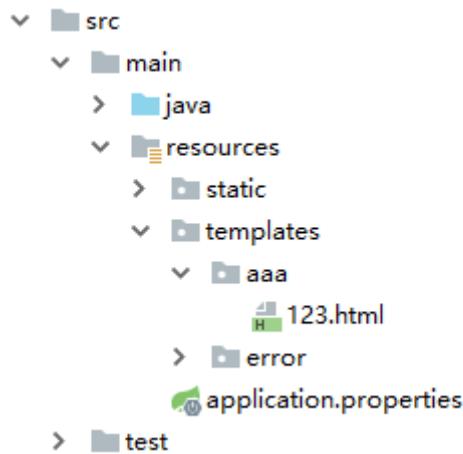
要自定义异常视图解析，也很容易，由于 `DefaultErrorViewResolver` 是在 `ErrorMvcAutoConfiguration` 类中提供的实例，即开发者没有提供相关实例时，会使用默认的 `DefaultErrorViewResolver`，开发者提供了自己的 `ErrorViewResolver` 实例后，默认的配置就会失效，因此，自定义异常视图，只需要提供一个 `ErrorViewResolver` 的实例即可：

```

@Component
public class MyErrorViewResolver extends DefaultErrorViewResolver {
    public MyErrorViewResolver(ApplicationContext applicationContext,
        ResourceProperties resourceProperties) {
        super(applicationContext, resourceProperties);
    }
    @Override
    public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus
status, Map<String, Object> model) {
        return new ModelAndView("/aaa/123", model);
    }
}

```

实际上，开发者也可以在这里定义异常数据（直接在 resolveErrorView 方法重新定义一个 model，将参数中的model 数据拷贝过去并修改，注意参数中的 model 类型为 UnmodifiableMap，即不可以直接修改），而不需要自定义 MyErrorAttributes。定义完成后，提供一个名为 123 的视图，如下图：



如此之后，错误试图就算定义成功了。

总结

实际上也可以自定义异常控制器 BasicErrorController，不过松哥觉得这样太大动干戈了，没必要，前面几种方式已经可以满足我们的大部分开发需求了。如果是前后端分离架构，异常处理还有其他一些处理方案，这个松哥以后和大家聊。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



今天和小伙伴们来聊一聊通过CORS解决跨域问题。

同源策略

很多人对跨域有一种误解，以为这是前端的事，和后端没关系，其实不是这样的，说到跨域，就不得不谈谈浏览器的同源策略。

同源策略是由 Netscape 提出的一个著名的安全策略，它是浏览器最核心也最基本的安全功能，现在所有支持 JavaScript 的浏览器都会使用这个策略。所谓同源是指协议、域名以及端口要相同。同源策略是基于安全方面的考虑提出来的，这个策略本身没问题，但是我们在实际开发中，由于各种原因又经常有跨域的需求，传统的跨域方案是 JSONP，JSONP 虽然能解决跨域但是有一个很大的局限性，那就是只支持 GET 请求，不支持其他类型的请求，而今天我们说的 CORS（跨域资源共享）（CORS, Cross-origin resource sharing）是一个 W3C 标准，它是一份浏览器技术的规范，提供了 Web 服务从不同网络传来沙盒脚本的方法，以避开浏览器的同源策略，这是 JSONP 模式的现代版。

在 Spring 框架中，对于 CORS 也提供了相应的解决方案，今天我们就来看看 SpringBoot 中如何实现 CORS。

实践

接下来我们就来看看 Spring Boot 中如何实现这个东西。

首先创建两个普通的 Spring Boot 项目，这个就不用我多说，第一个命名为 provider 提供服务，第二个命名为 consumer 消费服务，第一个配置端口为 8080，第二个配置配置为 8081，然后在 provider 上提供两个 hello 接口，一个 get，一个 post，如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
    @PostMapping("/hello")
    public String hello2() {
        return "post hello";
    }
}
```

在 consumer 的 resources/static 目录下创建一个 html 文件，发送一个简单的 ajax 请求，如下：

```
<div id="app"></div>
<input type="button" onclick="btnClick()" value="get_button">
<input type="button" onclick="btnClick2()" value="post_button">
<script>
    function btnClick() {
        $.get('http://localhost:8080/hello', function (msg) {
            $("#app").html(msg);
        });
    }

    function btnClick2() {
        $.post('http://localhost:8080/hello', function (msg) {
            $("#app").html(msg);
        });
    }
</script>
```

```
});  
}  
</script>
```

然后分别启动两个项目，发送请求按钮，观察浏览器控制台如下：

```
Access to XMLHttpRequest at 'http://localhost:8080/hello' from origin  
'http://localhost:8081' has been blocked by CORS policy: No 'Access-Control-  
Allow-Origin' header is present on the requested resource.
```

可以看到，由于同源策略的限制，请求无法发送成功。

使用 CORS 可以在前端代码不做任何修改的情况下，实现跨域，那么接下来看看在 provider 中如何配置。首先可以通过 `@CrossOrigin` 注解配置某一个方法接受某一个域的请求，如下：

```
@RestController  
public class HelloController {  
    @CrossOrigin(value = "http://localhost:8081")  
    @GetMapping("/hello")  
    public String hello() {  
        return "hello";  
    }  
  
    @CrossOrigin(value = "http://localhost:8081")  
    @PostMapping("/hello")  
    public String hello2() {  
        return "post hello";  
    }  
}
```

这个注解表示这两个接口接受来自 `http://localhost:8081` 地址的请求，配置完成后，重启 provider，再次发送请求，浏览器控制台就不会报错了，consumer 也能拿到数据了。

此时观察浏览器请求网络控制台，可以看到响应头中多了如下信息：

Name	Headers	Preview	Response	Timing
hello				
	▼ General			
	Request URL: http://localhost:8080/hello			
	Request Method: POST			
	Status Code: 200			
	Remote Address: [::1]:8080			
	Referrer Policy: no-referrer-when-downgrade			
	▼ Response Headers			
	Access-Control-Allow-Origin: http://localhost:8081			
	Content-Length: 10			
	Content-Type: text/plain; charset=UTF-8			
	Date: Tue, 12 Mar 2019 09:11:57 GMT			
	Vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers			
1 requests 265 B transferr...				

这个表示服务端愿意接收来自 `http://localhost:8081` 的请求，拿到这个信息后，浏览器就不会再去限制本次请求的跨域了。

provider 上，每一个方法上都去加注解未免太麻烦了，有的小伙伴想到可以讲注解直接加在 Controller 上，不过每个 Controller 都要加还是麻烦，在 Spring Boot 中，还可以通过全局配置一次性解决这个问题，全局配置只需要在 SpringMVC 的配置类中重写 `addCorsMappings` 方法即可，如下：

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("http://localhost:8081")
            .allowedMethods("/*")
            .allowedHeaders("/*");
    }
}
```

/** 表示本应用的所有方法都会去处理跨域请求，allowedMethods 表示允许通过的请求数，allowedHeaders 则表示允许的请求头。经过这样的配置之后，就不必在每个方法上单独配置跨域了。

存在的问题

了解了整个 CORS 的工作过程之后，我们通过 Ajax 发送跨域请求，虽然用户体验提高了，但是也有潜在的威胁存在，常见的就是 CSRF (Cross-site request forgery) 跨站请求伪造。跨站请求伪造也被称为 one-click attack 或者 session riding，通常缩写为 CSRF 或者 XSRF，是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法，举个例子：

假如一家银行用以运行转账操作的URL地址如下：`http://icbc.com/aa?bb=cc`，那么，一个恶意攻击者可以在另一个网站上放置如下代码：``，如果用户访问了恶意站点，而她之前刚访问过银行不久，登录信息尚未过期，那么她就会遭受损失。

基于此，浏览器在实际操作中，会对请求进行分类，分为简单请求，预先请求，带凭证的请求等，预先请求会首先发送一个 options 探测请求，和浏览器进行协商是否接受请求。默认情况下跨域请求是不需要凭证的，但是服务端可以配置要求客户端提供凭证，这样就可以有效避免 csrf 攻击。

好了，这个问题就说这么多，关于 Spring Boot 中的 CORS，松哥还有一个小小的视频教程

- [Spring Boot 中使用 CORS 解决跨域问题](#)

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



在 Servlet/Jsp 项目中，如果涉及到系统任务，例如在项目启动阶段要做一些数据初始化操作，这些操作有一个共同的特点，只在项目启动时进行，以后都不再执行，这里，容易想到 web 基础中的三大组件（Servlet、Filter、Listener）之一 Listener，这种情况下，一般定义一个 ServletContextListener，然后就可以监听到项目启动和销毁，进而做出相应的数据初始化和销毁操作，例如下面这样：

```
public class MyListener implements ServletContextListener {  
    @Override  
    public void contextInitialized(ServletContextEvent sce) {  
        //在这里做数据初始化操作  
    }  
    @Override  
    public void contextDestroyed(ServletContextEvent sce) {  
        //在这里做数据备份操作  
    }  
}
```

当然，这是基础 web 项目的解决方案，如果使用了 Spring Boot，那么我们可以使用更为简便的方式。Spring Boot 中针对系统启动任务提供了两种解决方案，分别是 CommandLineRunner 和 ApplicationRunner，分别来看。

CommandLineRunner

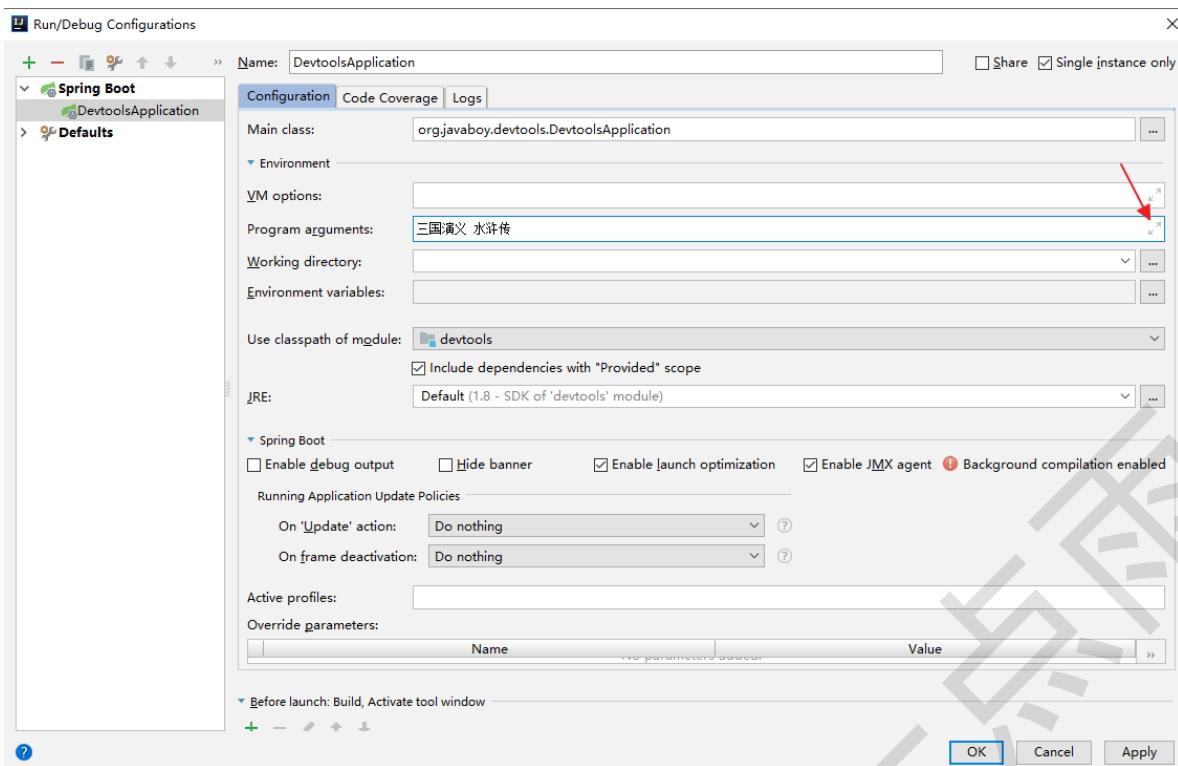
使用 CommandLineRunner 时，首先自定义 MyCommandLineRunner1 并且实现 CommandLineRunner 接口：

```
@Component  
 @Order(100)  
 public class MyCommandLineRunner1 implements CommandLineRunner {  
     @Override  
     public void run(String... args) throws Exception {  
     }  
 }
```

关于这段代码，我做如下解释：

1. 首先通过 @Component 注解将 MyCommandLineRunner1 注册为 Spring 容器中的一个 Bean。
2. 添加 @Order 注解，表示这个启动任务的执行优先级，因为在一个项目中，启动任务可能有多个，所以需要有一个排序。@Order 注解中，数字越小，优先级越大，默认情况下，优先级的值为 Integer.MAX_VALUE，表示优先级最低。
3. 在 run 方法中，写启动任务的核心逻辑，当项目启动时，run 方法会被自动执行。
4. run 方法的参数，来自于项目的启动参数，即项目入口类中，main 方法的参数会被传到这里。

此时启动项目，run 方法就会被执行，至于参数，可以通过两种方式来传递，如果是在 IDEA 中，可以通过如下方式来配置参数：



另一种方式，则是将项目打包，在命令行中启动项目，然后启动时在命令行传入参数，如下：

```
java -jar devtools-0.0.1-SNAPSHOT.jar 三国演义 西游记
```

注意，这里参数传递时没有 key，直接写 value 即可，执行结果如下：

```
2019-04-13 09:31:21.739  INFO 11236 — [MyCommandLineRunner2>>>[三国演义, 西游记]
MyCommandLineRunner1>>>[三国演义, 西游记]
```

ApplicationRunner

ApplicationRunner 和 CommandLineRunner 功能一致，用法也基本一致，唯一的区别主要体现在对参数的处理上，ApplicationRunner 可以接收更多类型的参数（ApplicationRunner 除了可以接收 CommandLineRunner 的参数之外，还可以接收 key/value 形式的参数）。

使用 ApplicationRunner，自定义类实现 ApplicationRunner 接口即可，组件注册以及组件优先级的配置都和 CommandLineRunner 一致，如下：

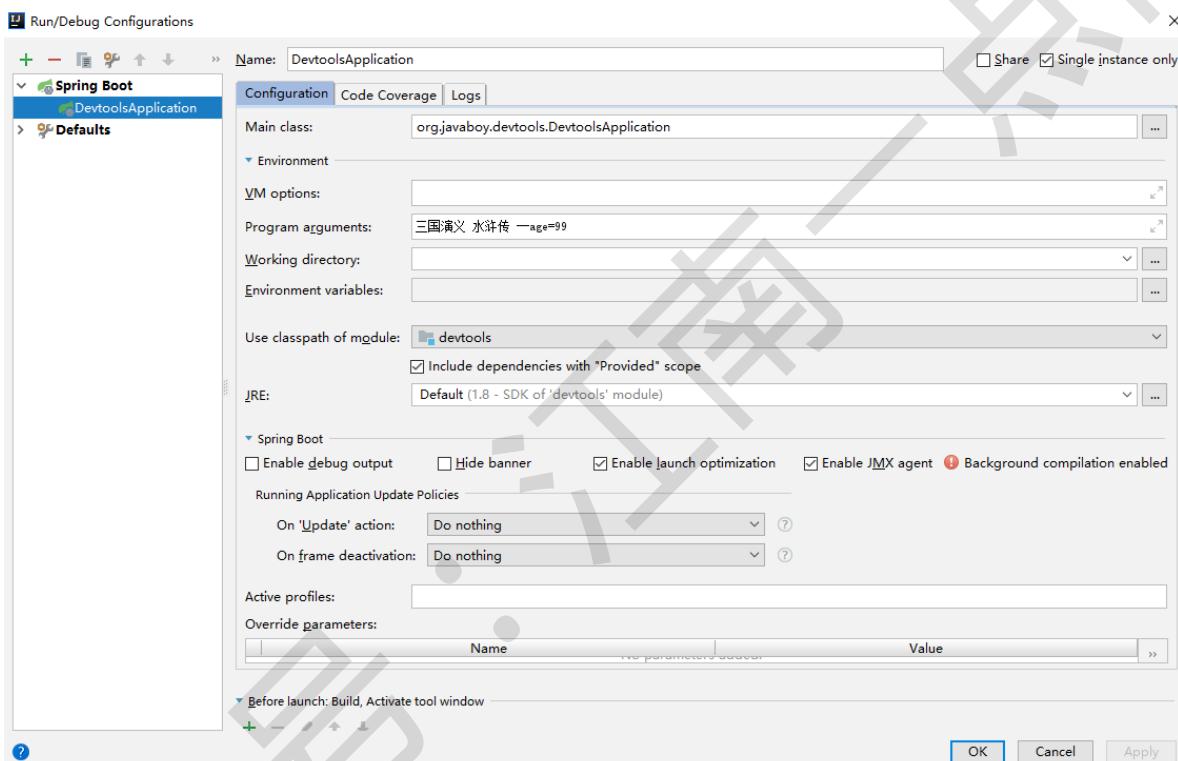
```
@Component
@Order(98)
public class MyApplicationRunner1 implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        List<String> nonOptionArgs = args.getNonOptionArgs();
        System.out.println("MyApplicationRunner1>>>" + nonOptionArgs);
        Set<String> optionNames = args.getOptionNames();
        for (String key : optionNames) {
            System.out.println("MyApplicationRunner1>>>" + key + ":" +
                args.getOptionValues(key));
        }
        String[] sourceArgs = args.getSourceArgs();
```

```
        System.out.println("MyApplicationRunner1>>>" + Arrays.toString(sourceArgs));
    }
}
```

当项目启动时，这里的 run 方法就会被自动执行，关于 run 方法的参数 ApplicationArguments，我说如下几点：

1. args.getNonOptionArgs(); 可以用来获取命令行中的无key参数（和 CommandLineRunner一样）。
2. args.getOptionNames(); 可以用来获取所有key/value形式的参数的key。
3. args.getOptionValues(key)); 可以根据key获取key/value 形式的参数的value。
4. args.getSourceArgs(); 则表示获取命令行中的所有参数。

ApplicationRunner 定义完成后，传启动参数也是两种方式，参数类型也有两种，第一种和 CommandLineRunner 一致，第二种则是 --key=value 的形式，在 IDEA 中定义方式如下：



或者使用如下启动命令：

```
java -jar devtools-0.0.1-SNAPSHOT.jar 三国演义 西游记 --age=99
```

运行结果如下：

```
MyApplicationRunner1>>>[三国演义, 水浒传]
MyApplicationRunner1>>>age:[99]
MyApplicationRunner1>>>[三国演义, 水浒传, --age=99]
MyCommandLineRunner2>>>[三国演义, 水浒传, --age=99]
MyCommandLineRunner1>>>[三国演义, 水浒传, --age=99]
```

总结

整体来说，这两种的用法的差异不大，主要体现在对参数的处理上，小伙伴可以根据项目中的实际情况选择合适的解决方案。相关案例已经上传到 GitHub，欢迎小伙伴们们下载：

<https://github.com/lenvve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南一点雨

在 Spring + SpringMVC 环境中，一般来说，要实现定时任务，我们有两中方案，一种是使用 Spring 自带的定时任务处理器 @Scheduled 注解，另一种就是使用第三方框架 Quartz，Spring Boot 源自 Spring+SpringMVC，因此天然具备这两个 Spring 中的定时任务实现策略，当然也支持 Quartz，本文我们就来看下 Spring Boot 中两种定时任务的实现方式。

@Scheduled

使用 @Scheduled 非常容易，直接创建一个 Spring Boot 项目，并且添加 web 依赖 `spring-boot-starter-web`，项目创建成功后，添加 `@EnableScheduling` 注解，开启定时任务：

```
@SpringBootApplication
@EnableScheduling
public class ScheduledApplication {
    public static void main(String[] args) {
        SpringApplication.run(ScheduledApplication.class, args);
    }
}
```

接下来配置定时任务：

```
@Scheduled(fixedRate = 2000)
public void fixedRate() {
    System.out.println("fixedRate>>" + new Date());
}

@scheduled(fixedDelay = 2000)
public void fixedDelay() {
    System.out.println("fixedDelay>>" + new Date());
}

@scheduled(initialDelay = 2000, fixedDelay = 2000)
public void initialDelay() {
    System.out.println("initialDelay>>" + new Date());
}
```

- 首先使用 @Scheduled 注解开启一个定时任务。
- fixedRate 表示任务执行之间的时间间隔，具体是指两次任务的开始时间间隔，即第二次任务开始时，第一次任务可能还没结束。
- fixedDelay 表示任务执行之间的时间间隔，具体是指本次任务结束到下次任务开始之间的时间间隔。
- initialDelay 表示首次任务启动的延迟时间。
- 所有时间的单位都是毫秒。

上面这是一个基本用法，除了这几个基本属性之外，@Scheduled 注解也支持 cron 表达式，使用 cron 表达式，可以非常丰富的描述定时任务的时间。cron 表达式格式如下：

[秒] [分] [小时] [日] [月] [周] [年]

具体取值如下：

序号	说明	是否必填	允许填写的值	允许的通配符
1	秒	是	0-59	- * /
2	分	是	0-59	- * /
3	时	是	0-23	- * /
4	日	是	1-31	- * ? / L W
5	月	是	1-12 or JAN-DEC	- * /
6	周	是	1-7 or SUN-SAT	- * ? / L ##
7	年	否	1970-2099	- * /

这一块需要大家注意的是，月份中的日期和星期可能会起冲突，因此在配置时这两个得有一个是？

通配符含义：

- ? 表示不指定值，即不关心某个字段的取值时使用。需要注意的是，月份中的日期和星期可能会起冲突，因此在配置时这两个得有一个是？
- * 表示所有值，例如：在秒的字段上设置 *，表示每一秒都会触发
- , 用来分开多个值，例如在周字段上设置 "MON,WED,FRI" 表示周一，周三和周五触发
- - 表示区间，例如在秒上设置 "10-12"，表示 10,11,12 秒都会触发
- / 用于递增触发，如在秒上面设置 "5/15" 表示从 5 秒开始，每增 15 秒触发(5,20,35,50)
- ## 序号(表示每月的第几个周几)，例如在周字段上设置 "6##3" 表示在每月的第三个周六，(用在母亲节和父亲节再合适不过了)
- 周字段的设置，若使用英文字母是不区分大小写的，即 MON 与 mon 相同
- L 表示最后的意思。在日字段设置上，表示当月的最后一天(依据当前月份，如果是二月还会自动判断是否是闰年)；在周字段上表示星期六，相当于 "7" 或 "SAT" (注意周日算是第一天)。如果在 "L" 前加上数字，则表示该数据的最后一个。例如在周字段上设置 "6L" 这样的格式，则表示 "本月最后一个星期五"
- W 表示离指定日期的最近工作日(周一至周五)，例如在日字段上设置 "15W"，表示离每月 15 号最近的那个工作日触发。如果 15 号正好是周六，则找最近的周五(14 号)触发；如果 15 号是周末，则找最近的下周一(16 号)触发。如果 15 号正好在工作日(周一至周五)，则就在该天触发。如果指定格式为 "1W"，它则表示每月 1 号往后最近的工作日触发。如果 1 号正是周六，则将在 3 号下周一触发。
(注，"W" 前只能设置具体的数字，不允许区间 "-")
- L 和 W 可以一起组合使用。如果在日字段上设置 "LW"，则表示在本月的最后一个工作日触发(一般指发工资)

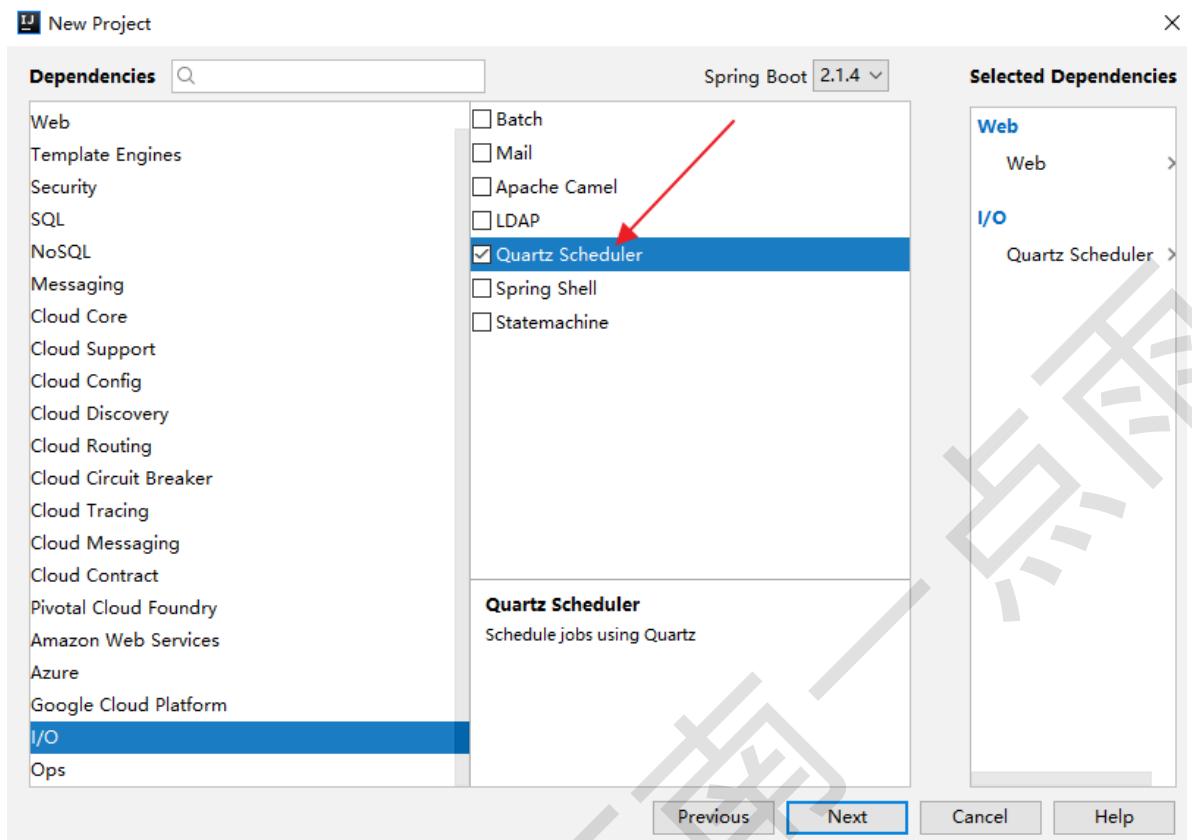
例如，在 @Scheduled 注解中来一个简单的 cron 表达式，每隔 5 秒触发一次，如下：

```
@Scheduled(cron = "0/5 * * * *")
public void cron() {
    System.out.println(new Date());
}
```

上面介绍的是使用 @Scheduled 注解的方式来实现定时任务，接下来我们再来看看如何使用 Quartz 实现定时任务。

Quartz

一般在项目中，除非定时任务涉及到的业务实在是太简单，使用 @Scheduled 注解来解决定时任务，否则大部分情况可能都是使用 Quartz 来做定时任务。在 Spring Boot 中使用 Quartz，只需要在创建项目时，添加 Quartz 依赖即可：



项目创建完成后，也需要添加开启定时任务的注解：

```
@SpringBootApplication  
@EnableScheduling  
public class QuartzApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(QuartzApplication.class, args);  
    }  
}
```

Quartz 在使用过程中，有两个关键概念，一个是 JobDetail（要做的事情），另一个是触发器（什么时候做），要定义 JobDetail，需要先定义 Job，Job 的定义有两种方式：

第一种方式，直接定义一个 Bean：

```
@Component  
public class MyJob1 {  
    public void sayHello() {  
        System.out.println("MyJob1>>>"+new Date());  
    }  
}
```

关于这种定义方式说两点：

1. 首先将这个 Job 注册到 Spring 容器中。
2. 这种定义方式有一个缺陷，就是无法传参。

第二种定义方式，则是继承 QuartzJobBean 并实现默认的方法：

```

public class MyJob2 extends QuartzJobBean {
    HelloService helloService;
    public HelloService getHelloService() {
        return helloService;
    }
    public void setHelloService(HelloService helloService) {
        this.helloService = helloService;
    }
    @Override
    protected void executeInternal(JobExecutionContext jobExecutionContext)
throws JobExecutionException {
        helloService.sayHello();
    }
}
public class HelloService {
    public void sayHello() {
        System.out.println("hello service >>>" + new Date());
    }
}

```

和第1种方式相比，这种方式支持传参，任务启动时，executeInternal方法将会被执行。

Job有了之后，接下来创建类，配置JobDetail 和 Trigger 触发器，如下：

```

@Configuration
public class QuartzConfig {
    @Bean
    MethodInvokingJobDetailFactoryBean methodInvokingJobDetailFactoryBean() {
        MethodInvokingJobDetailFactoryBean bean = new
MethodInvokingJobDetailFactoryBean();
        bean.setTargetBeanName("myJob1");
        bean.setTargetMethod("sayHello");
        return bean;
    }
    @Bean
    JobDetailFactoryBean jobDetailFactoryBean() {
        JobDetailFactoryBean bean = new JobDetailFactoryBean();
        bean.setJobClass(MyJob2.class);
        JobDataMap map = new JobDataMap();
        map.put("helloService", helloService());
        bean.setJobDataMap(map);
        return bean;
    }
    @Bean
    SimpleTriggerFactoryBean simpleTriggerFactoryBean() {
        SimpleTriggerFactoryBean bean = new SimpleTriggerFactoryBean();
        bean.setStartTime(new Date());
        bean.setRepeatCount(5);
        bean.setJobDetail(methodInvokingJobDetailFactoryBean().getObject());
        bean.setRepeatInterval(3000);
        return bean;
    }
    @Bean
    CronTriggerFactoryBean cronTrigger() {
        CronTriggerFactoryBean bean = new CronTriggerFactoryBean();
        bean.setCronExpression("0/10 * * * * ?");
        bean.setJobDetail(jobDetailFactoryBean().getObject());
    }
}

```

```

        return bean;
    }

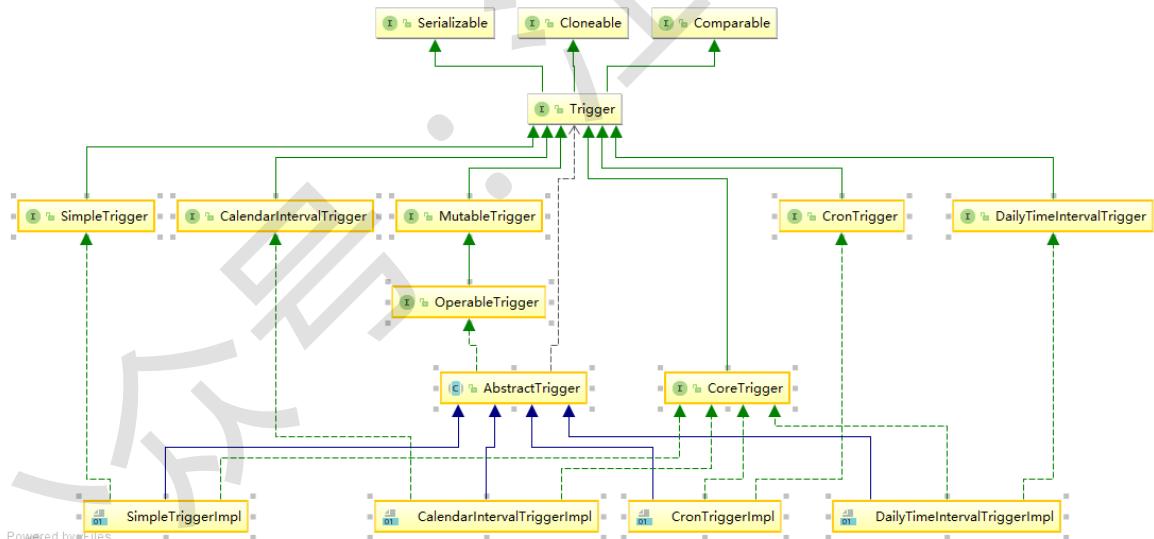
    @Bean
    SchedulerFactoryBean schedulerFactoryBean() {
        SchedulerFactoryBean bean = new SchedulerFactoryBean();
        bean.setTriggers(cronTrigger().getObject(),
        simpleTriggerFactoryBean().getObject());
        return bean;
    }

    @Bean
    HelloService helloService() {
        return new HelloService();
    }
}

```

关于这个配置说如下几点：

1. JobDetail 的配置有两种方式：MethodInvokingJobDetailFactoryBean 和 JobDetailFactoryBean。
2. 使用 MethodInvokingJobDetailFactoryBean 可以配置目标 Bean 的名字和目标方法的名字，这种方式不支持传参。
3. 使用 JobDetailFactoryBean 可以配置 JobDetail，任务类继承自 QuartzJobBean，这种方式支持传参，将参数封装在 JobDataMap 中进行传递。
4. Trigger 是指触发器，Quartz 中定义了多个触发器，这里向大家展示其中两种的用法， SimpleTrigger 和 CronTrigger。
5. SimpleTrigger 有点类似于前面说的 @Scheduled 的基本用法。
6. CronTrigger 则有点类似于 @Scheduled 中 cron 表达式的用法。



全部定义完成后，启动 Spring Boot 项目就可以看到定时任务的执行了。

总结

这里主要向大家展示了 Spring Boot 中整合两种定时任务的方法，整合成功之后，剩下的用法基本上就和在 SSM 中使用一致了，不再赘述。本文案例我已经上传到 GitHub，欢迎大家 star：
<https://github.com/lenve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

前后端分离后，维护接口文档基本上是必不可少的工作。

一个理想的状态是设计好后，接口文档发给前端和后端，大伙按照既定的规则各自开发，开发好了对接上了就可以上线了。当然这是一种非常理想的状态，实际开发中却很少遇到这样的情况，接口总是在不断的变化之中，有变化就要去维护，做过的小伙伴都知道这件事有多么头大！还好，有一些工具可以减轻我们的工作量，Swagger2 就是其中之一，至于其他类似功能但是却收费的软件，这里就不做过多介绍了。本文主要和大伙来聊下 在Spring Boot 中如何整合 Swagger2。

工程创建

当然，首先是创建一个 Spring Boot 项目，加入 web 依赖，创建成功后，加入两个 Swagger2 相关的依赖，完整的依赖如下：

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Swagger2 配置

Swagger2 的配置也是比较容易的，在项目创建成功之后，只需要开发者自己提供一个 Docket 的 Bean 即可，如下：

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
            .pathMapping("/")
            .select()
            .apis(RequestHandlerSelectors.basePackage("org.javaboy.controller"))
            .paths(PathSelectors.any())
            .build().apiInfo(new ApiInfoBuilder()
                .title("SpringBoot整合Swagger")
                .description("SpringBoot整合Swagger，详细信息.....")
                .version("9.0")
                .contact(new Contact("啊啊啊
啊","blog.csdn.net","aaa@gmail.com"))
                .license("The Apache License")
                .licenseUrl("http://www.javaboy.org")
                .build());
    }
}
```

```
    }  
}
```

这里提供一个配置类，首先通过 `@EnableSwagger2` 注解启用 Swagger2，然后配置一个 Docket Bean，这个 Bean 中，配置映射路径和要扫描的接口的位置，在 `apiInfo` 中，主要配置一下 Swagger2 文档网站的信息，例如网站的 title，网站的描述，联系人的信息，使用的协议等等。

如此，Swagger2 就算配置成功了，非常方便。

此时启动项目，输入 `http://localhost:8080/swagger-ui.html`，能够看到如下页面，说明已经配置成功了：



创建接口

接下来就是创建接口了，Swagger2 相关的注解其实并不多，而且很容易懂，下面我来分别向小伙伴们举例说明：

```
@RestController  
@Api(tags = "用户管理相关接口")  
@RequestMapping("/user")  
public class UserController {  
    @PostMapping("/")  
    @ApiOperation("添加用户的接口")  
    @ApiImplicitParams({  
        @ApiImplicitParam(name = "username", value = "用户名", defaultValue =  
        "李四"),  
        @ApiImplicitParam(name = "address", value = "用户地址", defaultValue  
        = "深圳", required = true)  
    })  
    public RespBean addUser(@RequestParam(required = true)  
    String username, @RequestParam(required = true)  
    String address) {  
        return new RespBean();  
    }  
    @GetMapping("/")  
    @ApiOperation("根据id查询用户的接口")  
    @ApiImplicitParam(name = "id", value = "用户id", defaultValue = "99",  
    required = true)  
    public User getUserById(@PathVariable Integer id) {  
        User user = new User();  
    }
```

```
        user.setId(id);
        return user;
    }
    @PutMapping("/{id}")
    @ApiOperation("根据id更新用户的接口")
    public User updateUserById(@RequestBody User user) {
        return user;
    }
}
```

这里涉及到多个 API，我来向小伙伴们分别说明：

1. @Api 注解可以用来标记当前 Controller 的功能。
2. @ApiOperation 注解用来标记一个方法的作用。
3. @ApiImplicitParam 注解用来描述一个参数，可以配置参数的中文含义，也可以给参数设置默认值，这样在接口测试的时候可以避免手动输入。
4. 如果有多个参数，则需要使用多个 @ApiImplicitParam 注解来描述，多个 @ApiImplicitParam 注解需要放在一个 @ApiImplicitParams 注解中。
5. 需要注意的是，@ApiImplicitParam 注解中虽然可以指定参数是必填的，但是却不能代替 @RequestParam(required = true)，前者的必填只是在 Swagger2 框架内必填，抛弃了 Swagger2，这个限制就没用了，所以假如开发者需要指定一个参数必填，@RequestParam(required = true) 注解还是不能省略。
6. 如果参数是一个对象（例如上文的更新接口），对于参数的描述也可以放在实体类中。例如下面一段代码：

```
@ApiModelProperty
public class User {
    @ApiModelProperty(value = "用户id")
    private Integer id;
    @ApiModelProperty(value = "用户名")
    private String username;
    @ApiModelProperty(value = "用户地址")
    private String address;
    //getter/setter
}
```

好了，经过如上配置之后，接下来，刷新刚刚打开的页面，可以看到如下效果：

用户管理相关接口 User Controller



POST /user/ 添加用户的接口

PUT /user/ 根据id更新用户的接口

GET /user/{id} 根据id查询用户的接口

Models

RespBean >

User >

可以看到，所有的接口这里都列出来了，包括接口请求方式，接口地址以及接口的名字等，点开一个接口，可以看到如下信息：

POST

/user/ 添加用户的接口

Parameters

Try it out

Name

Description

address * required

string
(query)

用户地址

Default value: 深圳

username

用户名

string
(query)

Default value: 李四

Responses

Response content type

/

Code

Description

200

OK

Example Value

Model

可以看到，接口的参数，参数要求，参数默认值等等统统都展示出来了，参数类型下的 query 表示参数以 key/value 的形式传递，点击右上角的 Try it out，就可以进行接口测试：

POST

/user/ 添加用户的接口

Parameters

Cancel

Name

Description

address * required

string

(query)

用户地址

深圳

username

string

(query)

用户名

李四

Execute

Responses

Response content type

/



点击 Execute 按钮，表示发送请求进行测试。测试结果会展示在下面的 Response 中。

小伙伴们注意，参数类型下面的 query 表示参数以 key/value 的形式传递，这里的值也可能是 body，body 表示参数以请求体的方式传递，例如上文的更新接口，如下：

PUT

/user/ 根据id更新用户的接口

Parameters

Try it out

Name	Description
user * required	user
(body)	Example Value Model

```
User ▾ {  
    address      string  用户地址  
    id          integer($int32)  用户Id  
    username    string  用户名  
}
```

Responses

Response content type

/

Code

Description

当然还有一种可能就是这里的参数为 path，表示参数放在路径中传递，例如根据 id 查询用户的接口：

GET

/user/{id} 根据id查询用户的接口

Parameters

Try it out

Name	Description
id * required string (path)	用户id <i>Default value: 99</i>

Responses

Response content type

/

Code	Description
200	OK

当然，除了这些之外，还有一些响应值的注解，都比较简单，小伙伴们可以自己摸索下。

在 Security 中的配置

如果我们的 Spring Boot 项目中集成了 Spring Security，那么如果不做额外配置，Swagger2 文档可能会被拦截，此时只需要在 Spring Security 的配置类中重写 configure 方法，添加如下过滤即可：

```
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring()
        .antMatchers("/swagger-ui.html")
        .antMatchers("/v2/**")
        .antMatchers("/swagger-resources/**");
}
```

如此之后，Swagger2 文件就不需要认证就能访问了。不知道小伙伴们有没有看懂呢？有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

用过 Spring Boot 的小伙伴都知道，我们只需要在项目中引入 `spring-boot-starter-web` 依赖，SpringMVC 的一整套东西就会自动给我们配置好，但是，真实的项目环境比较复杂，系统自带的配置不一定满足我们的需求，往往我们还需要结合实际情况自定义配置。

自定义配置就有讲究了，由于 Spring Boot 的版本变迁，加上这一块本身就有几个不同写法，很多小伙伴在这里容易搞混，今天松哥就来和大家说一说这个问题。

概览

首先我们需要明确，跟自定义 SpringMVC 相关的类和注解主要有如下四个：

- `WebMvcConfigurerAdapter`
- `WebMvcConfigurer`
- `WebMvcConfigurationSupport`
- `@EnableWebMvc`

这四个中，除了第四个是注解，另外三个两个类一个接口，里边的方法看起来好像都类似，但是实际使用效果却大不相同，因此很多小伙伴容易搞混，今天松哥就来和大家聊一聊这个问题。

WebMvcConfigurerAdapter

我们先来看 `WebMvcConfigurerAdapter`，这个是在 Spring Boot 1.x 中我们自定义 SpringMVC 时继承的一个抽象类，这个抽象类本身是实现了 `WebMvcConfigurer` 接口，然后抽象类里边都是空方法，我们来看一下这个类的声明：

```
public abstract class WebMvcConfigurerAdapter implements WebMvcConfigurer {  
    //各种 SpringMVC 配置的方法  
}
```

再来看看这个类的注释：

```
/**  
 * An implementation of {@link WebMvcConfigurer} with empty methods allowing  
 * subclasses to override only the methods they're interested in.  
 * @deprecated as of 5.0 {@link WebMvcConfigurer} has default methods (made  
 * possible by a Java 8 baseline) and can be implemented directly without the  
 * need for this adapter  
 */
```

这段注释关于这个类说的很明白了。同时我们也看到，从 Spring5 开始，由于我们要使用 Java8，而 Java8 中的接口允许存在 `default` 方法，因此官方建议我们直接实现 `WebMvcConfigurer` 接口，而不是继承 `WebMvcConfigurerAdapter` 类。

也就是说，在 Spring Boot 1.x 的时代，如果我们需要自定义 SpringMVC 配置，直接继承 `WebMvcConfigurerAdapter` 类即可。

WebMvcConfigurer

根据上一小节的解释，小伙伴们已经明白了，`WebMvcConfigurer` 是我们在 Spring Boot 2.x 中实现自定义配置的方案。

WebMvcConfigurer 是一个接口，接口中的方法和 WebMvcConfigurerAdapter 中定义的空方法其实一样，所以用法上来说，基本上没有差别，从 Spring Boot 1.x 切换到 Spring Boot 2.x，只需要把继承类改成实现接口即可。

松哥在之前的案例中([40 篇原创干货，带你进入 Spring Boot 殿堂！](#))，凡是涉及到自定义 SpringMVC 配置的地方，也都是通过实现 WebMvcConfigurer 接口来完成的。

WebMvcConfigurationSupport

前面两个都好理解，还有一个 WebMvcConfigurationSupport，这个又是干什么用的呢？

松哥之前有一篇文章中用过这个类，不知道小伙伴们有没有留意，就是下面这篇：

- [纯 Java 代码搭建 SSM 环境](#)

这篇文章我放弃了 Spring 和 SpringMVC 的 xml 配置文件，转而用 Java 代替这两个 xml 配置。那么在这里我自定义 SpringMVC 配置的时候，就是通过继承 WebMvcConfigurationSupport 类来实现的。在 WebMvcConfigurationSupport 类中，提供了用 Java 配置 SpringMVC 所需要的所有方法。我们来看一下这个方法的摘要：

WebMvcConfigurationSupport.java

□ Inherited members (Ctrl+F12) □ Anonymous Classes (Ctrl+I) □ Lambdas (Ctrl+L)

WebMvcConfigurationSupport

- static class initializer ClassLoader classLo...
- m T addArgumentResolvers(List<HandlerMethodArgumentResolver>): void
- m T addCorsMappings(CorsRegistry): void
- m T addDefaultHandlerExceptionResolvers(List<HandlerExceptionResolver>): void
- m T addDefaultHttpMessageConverters(List<HttpMessageConverter<?>>): void
- m T addFormatters(FormatterRegistry): void
- m T addInterceptors(InterceptorRegistry): void
- m T addResourceHandlers(ResourceHandlerRegistry): void
- m T addReturnValueHandlers(List<HandlerMethodReturnValueHandler>): void
- m T addViewControllers(ViewControllerRegistry): void
- m beanNameHandlerMapping(): BeanNameUrlHandlerMapping
- m T configureAsyncSupport(AsyncSupportConfigurer): void
- m T configureContentNegotiation(ContentNegotiationConfigurer): void
- m T configureDefaultServletHandling(DefaultServletHandlerConfigurer): void
- m T configureHandlerExceptionResolvers(List<HandlerExceptionResolver>): void
- m T configureMessageConverters(List<HttpMessageConverter<?>>): void
- m T configurePathMatch(PathMatchConfigurer): void
- m T configureViewResolvers(ViewResolverRegistry): void
- m T createExceptionHandlerExceptionResolver(): ExceptionHandlerExceptionResolver
- m T createRequestMappingHandlerAdapter(): RequestMappingHandlerAdapter
- m T createRequestMappingHandlerMapping(): RequestMappingHandlerMapping
- m defaultServletHandlerMapping(): HandlerMapping
- m T extendHandlerExceptionResolvers(List<HandlerExceptionResolver>): void
- m T extendMessageConverters(List<HttpMessageConverter<?>>): void
- m getApplicationContext(): ApplicationContext
- m T getArgumentResolvers(): List<HandlerMethodArgumentResolver>
- m T getConfigurableWebBindingInitializer(): ConfigurableWebBindingInitializer
- m T getCorsConfigurations(): Map<String, CorsConfiguration>
- m T getDefaultMediaTypes(): Map<String, MediaType>
- m T getInterceptors(): Object[]
- m T getMessageCodesResolver(): MessageCodesResolver
- m T getMessageConverters(): List<HttpMessageConverter<?>>

有一点眼熟，可能有小伙伴发现了，这里的方法其实和前面两个类中的方法基本是一样的。

在这里首先大家需要明确的是，WebMvcConfigurationSupport 类本身是没有问题的，我们自定义 SpringMVC 的配置是可以通过继承 WebMvcConfigurationSupport 来实现的。但是继承 WebMvcConfigurationSupport 这种操作我们一般只在 Java 配置的 SSM 项目中使用，Spring Boot 中基本上不会这么写，为什么呢？

小伙伴们知道，Spring Boot 中，SpringMVC 相关的自动化配置是在 WebMvcAutoConfiguration 配置类中实现的，那么我们来看看这个配置类的生效条件：

```
@Configuration  
 @ConditionalOnWebApplication(type = Type.SERVLET)  
 @ConditionalOnClass({ Servlet.class, DispatcherServlet.class,  
 WebMvcConfigurer.class })  
 @ConditionalOnMissingBean(WebMvcConfigurationSupport.class)  
 @AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)  
 @AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,  
 TaskExecutionAutoConfiguration.class,  
 ValidationAutoConfiguration.class })  
 public class webMvcAutoConfiguration {  
 }
```

我们从这个类的注解中可以看到，它的生效条件有一条，就是当不存在 WebMvcConfigurationSupport 的实例时，这个自动化配置才会生效。因此，如果我们在 Spring Boot 中自定义 SpringMVC 配置时选择了继承 WebMvcConfigurationSupport，就会导致 Spring Boot 中 SpringMVC 的自动化配置失效。

Spring Boot 给我们提供了很多自动化配置，很多时候当我们修改这些配置的时候，并不是要全盘否定 Spring Boot 提供的自动化配置，我们可能只是针对某一个配置做出修改，其他的配置还是按照 Spring Boot 默认的自动化配置来，而继承 WebMvcConfigurationSupport 来实现对 SpringMVC 的配置会导致所有的 SpringMVC 自动化配置失效，因此，一般情况下我们不选择这种方案。

在 Java 搭建的 SSM 项目中([纯 Java 代码搭建 SSM 环境](#))，因为本身就没什么自动化配置，所以我们使用了继承 WebMvcConfigurationSupport。

@EnableWebMvc

最后还有一个 @EnableWebMvc 注解，这个注解很好理解，它的作用就是启用 WebMvcConfigurationSupport。我们来看看这个注解的定义：

```
/**  
 * Adding this annotation to an {@code @Configuration} class imports the Spring  
 * MVC  
 * configuration from {@link WebMvcConfigurationSupport}, e.g.:
```

可以看到，加了这个注解，就会自动导入 WebMvcConfigurationSupport，所以在 Spring Boot 中，我们也不建议使用 @EnableWebMvc 注解，因为它一样会导致 Spring Boot 中的 SpringMVC 自动化配置失效。

总结

不知道上面的解释小伙伴有没有看懂？我再简单总结一下：

1. Spring Boot 1.x 中，自定义 SpringMVC 配置可以通过继承 WebMvcConfigurerAdapter 来实现。
2. Spring Boot 2.x 中，自定义 SpringMVC 配置可以通过实现 WebMvcConfigurer 接口来完成。
3. 如果在 Spring Boot 中使用继承 WebMvcConfigurationSupport 来实现自定义 SpringMVC 配置，或者在 Spring Boot 中使用了 @EnableWebMvc 注解，都会导致 Spring Boot 中默认的 SpringMVC 自动化配置失效。
4. 在纯 Java 配置的 SSM 环境中，如果我们要自定义 SpringMVC 配置，有两种办法，第一种就是直接继承自 WebMvcConfigurationSupport 来完成 SpringMVC 配置，还有一种方案就是实现 WebMvcConfigurer 接口来完成自定义 SpringMVC 配置，如果使用第二种方式，则需要给 SpringMVC 的配置类上额外添加 @EnableWebMvc 注解，表示启用 WebMvcConfigurationSupport，这样配置才会生效。换句话说，在纯 Java 配置的 SSM 中，如

果你需要自定义 SpringMVC 配置，你离不开 WebMvcConfigurationSupport，所以在这种情况下建议通过继承 WebMvcConfigurationSupport 来实现自动化配置。

不知道小伙伴们有没有看懂呢？有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。

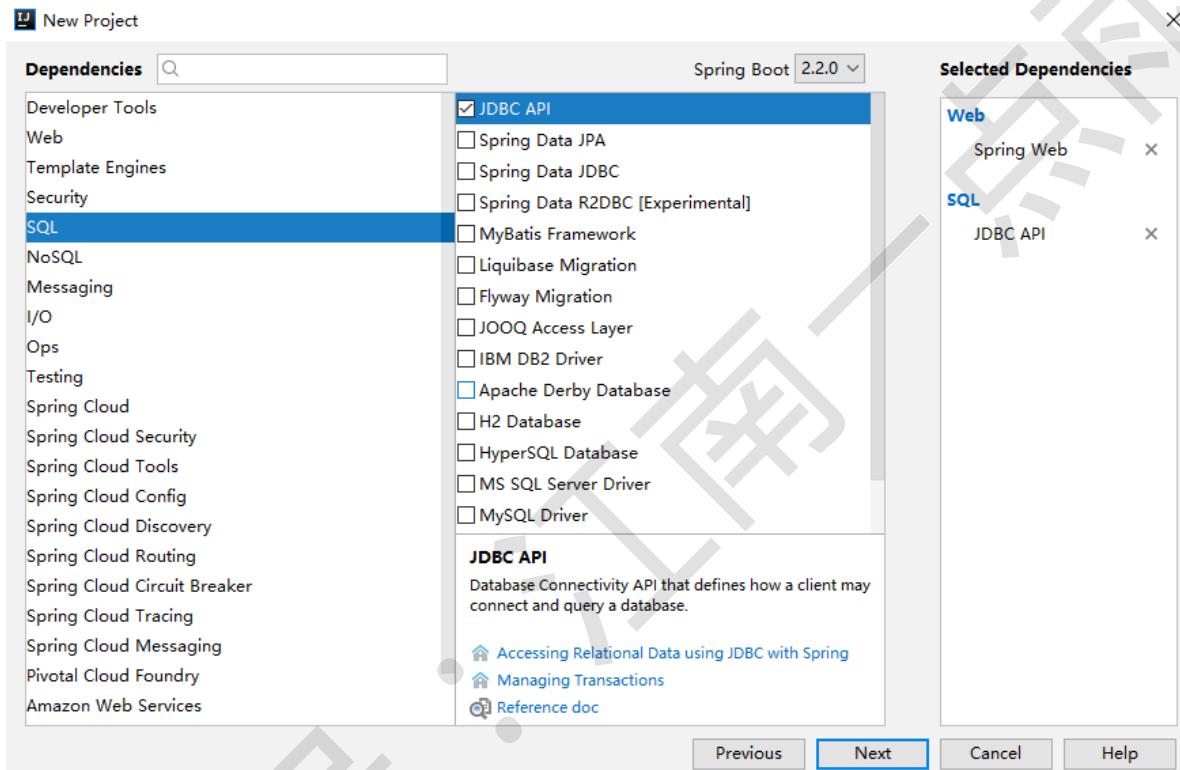


江南一点雨

在 Java 领域，数据持久化有几个常见的方案，有 Spring 自带的 JdbcTemplate、有 MyBatis，还有 JPA，在这些方案中，最简单的就是 Spring 自带的 JdbcTemplate 了，这个东西虽然没有 MyBatis 那么方便，但是比起最开始的 JDBC 已经强了很多了，它没有 MyBatis 功能那么强大，当然也意味着它的使用比较简单，事实上，JdbcTemplate 算是最简单的数据持久化方案了，本文就和大伙来说说这个东西的使用。

1. 基本配置

JdbcTemplate 基本用法实际上很简单，开发者在创建一个 SpringBoot 项目时，除了选择基本的 Web 依赖，再记得选上 JDBC 依赖，以及数据库驱动依赖即可，如下：



项目创建成功之后，记得添加 Druid 数据库连接池依赖（注意这里可以添加专门为 Spring Boot 打造的 `druid-spring-boot-starter`，而不是我们一般在 SSM 中添加的 Druid），所有添加的依赖如下：

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.27</version>
    <scope>runtime</scope>
```

```
</dependency>
```

项目创建完后，接下来只需要在 application.properties 中提供数据的基本配置即可，如下：

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource  
spring.datasource.username=root  
spring.datasource.password=123  
spring.datasource.url=jdbc:mysql:///test01?  
useUnicode=true&characterEncoding=UTF-8
```

如此之后，所有的配置就算完成了，接下来就可以直接使用 JdbcTemplate 了？咋这么方便呢？其实这就是 SpringBoot 的自动化配置带来的好处，我们先说用法，一会来说原理。

2. 基本用法

首先我们来创建一个 User Bean，如下：

```
public class User {  
    private Long id;  
    private String username;  
    private String address;  
    //省略getter/setter  
}
```

然后来创建一个 UserService 类，在 UserService 类中注入 JdbcTemplate，如下：

```
@Service  
public class UserService {  
    @Autowired  
    JdbcTemplate jdbcTemplate;  
}
```

好了，如此之后，准备工作就算完成了。

2.1 增

JdbcTemplate 中，除了查询有几个 API 之外，增删改统一都使用 update 来操作，自己来传入 SQL 即可。例如添加数据，方法如下：

```
public int addUser(User user) {  
    return jdbcTemplate.update("insert into user (username,address) values  
    (?,?);", user.getUsername(), user.getAddress());  
}
```

update 方法的返回值就是 SQL 执行受影响的行数。

这里只需要传入 SQL 即可，如果你的需求比较复杂，例如在数据插入的过程中希望实现主键回填，那么可以使用 PreparedStatementCreator，如下：

```
public int addUser2(User user) {  
    KeyHolder keyHolder = new GeneratedKeyHolder();  
    int update = jdbcTemplate.update(new PreparedStatementCreator() {  
        @Override
```

```

        public PreparedStatement createPreparedStatement(Connection connection)
throws SQLException {
    PreparedStatement ps = connection.prepareStatement("insert into user
(username,address) values (?,?)", Statement.RETURN_GENERATED_KEYS);
    ps.setString(1, user.getUsername());
    ps.setString(2, user.getAddress());
    return ps;
}
}, keyHolder);
user.setId(keyHolder.getKey().longValue());
System.out.println(user);
return update;
}

```

实际上这里就相当于完全使用了 JDBC 中的解决方案了，首先在构建 PreparedStatement 时传入 Statement.RETURN_GENERATED_KEYS，然后传入 KeyHolder，最终从 KeyHolder 中获取刚刚插入数据的 id 保存到 user 对象的 id 属性中去。

你能想到的 JDBC 的用法，在这里都能实现，Spring 提供的 JdbcTemplate 虽然不如 MyBatis，但是比起 Jdbc 还是要方便很多的。

2.2 删

删除也是使用 update API，传入你的 SQL 即可：

```

public int deleteUserById(Long id) {
    return jdbcTemplate.update("delete from user where id=?", id);
}

```

当然你也可以使用 PreparedStatementCreator。

2.3 改

```

public int updateUserById(User user) {
    return jdbcTemplate.update("update user set username=?,address=? where
id=?", user.getUsername(), user.getAddress(), user.getId());
}

```

当然你也可以使用 PreparedStatementCreator。

2.4 查

查询的话，稍微有点变化，这里主要向大伙介绍 query 方法，例如查询所有用户：

```

public List<User> getAllUsers() {
    return jdbcTemplate.query("select * from user", new RowMapper<User>() {
        @Override
        public User mapRow(ResultSet resultSet, int i) throws SQLException {
            String username = resultSet.getString("username");
            String address = resultSet.getString("address");
            long id = resultSet.getLong("id");
            User user = new User();
            user.setAddress(address);
            user.setUsername(username);
            user.setId(id);
        }
    });
}

```

```
        return user;
    }
}
}
```

查询的时候需要提供一个 RowMapper，就是需要自己手动映射，将数据库中的字段和对象的属性一一对应起来，这样。。。嗯看起来有点麻烦，实际上，如果数据库中的字段和对象属性的名字一模一样的话，有另外一个简单的方案，如下：

```
public List<User> getAllUsers2() {
    return jdbcTemplate.query("select * from user", new BeanPropertyRowMapper<>(User.class));
}
```

至于查询时候传参也是使用占位符，这个和前文的一致，这里不再赘述。

2.5 其他

除了这些基本用法之外，JdbcTemplate 也支持其他用法，例如调用存储过程等，这些都比较容易，而且和 Jdbc 本身都比较相似，这里也就不做介绍了，有兴趣可以留言讨论。

3. 原理分析

那么在 SpringBoot 中，配置完数据库基本信息之后，就有一个 JdbcTemplate 了，这个东西是从哪里来的呢？源码在

`org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration` 类中，该类源码如下：

```
@Configuration
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })
@ConditionalOnSingleCandidate(DataSource.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
@EnableConfigurationProperties(JdbcProperties.class)
public class JdbcTemplateAutoConfiguration {

    @Configuration
    static class JdbcTemplateConfiguration {
        private final DataSource dataSource;
        private final JdbcProperties properties;
        JdbcTemplateConfiguration(DataSource dataSource, JdbcProperties properties) {
            this.dataSource = dataSource;
            this.properties = properties;
        }
        @Bean
        @Primary
        @ConditionalOnMissingBean(JdbcOperations.class)
        public JdbcTemplate jdbcTemplate() {
            JdbcTemplate jdbcTemplate = new JdbcTemplate(this.dataSource);
            JdbcProperties.Template template = this.properties.getTemplate();
            jdbcTemplate.setFetchsize(template.getFetchsize());
            jdbcTemplate.setMaxRows(template.getMaxRows());
            if (template.getQueryTimeout() != null) {
                jdbcTemplate
                    .setQueryTimeout((int)
template.getQueryTimeout().getSeconds());
            }
        }
    }
}
```

```
        }
        return jdbcTemplate;
    }

}

@Configuration
@Import(JdbcTemplateConfiguration.class)
static class NamedParameterJdbcTemplateConfiguration {
    @Bean
    @Primary
    @ConditionalOnSingleCandidate(JdbcTemplate.class)
    @ConditionalOnMissingBean(NamedParameterJdbcOperations.class)
    public NamedParameterJdbcTemplate namedParameterJdbcTemplate(
            JdbcTemplate jdbcTemplate) {
        return new NamedParameterJdbcTemplate(jdbcTemplate);
    }
}
}
```

从这个类中，大致可以看出，当当前类路径下存在 DataSource 和 JdbcTemplate 时，该类就会被自动配置，jdbcTemplate 方法则表示，如果开发者没有自己提供一个 JdbcOperations 的实例的话，系统就自动配置一个 JdbcTemplate Bean (JdbcTemplate 是 JdbcOperations 接口的一个实现)。好了，不知道大伙有没有收获呢？

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。

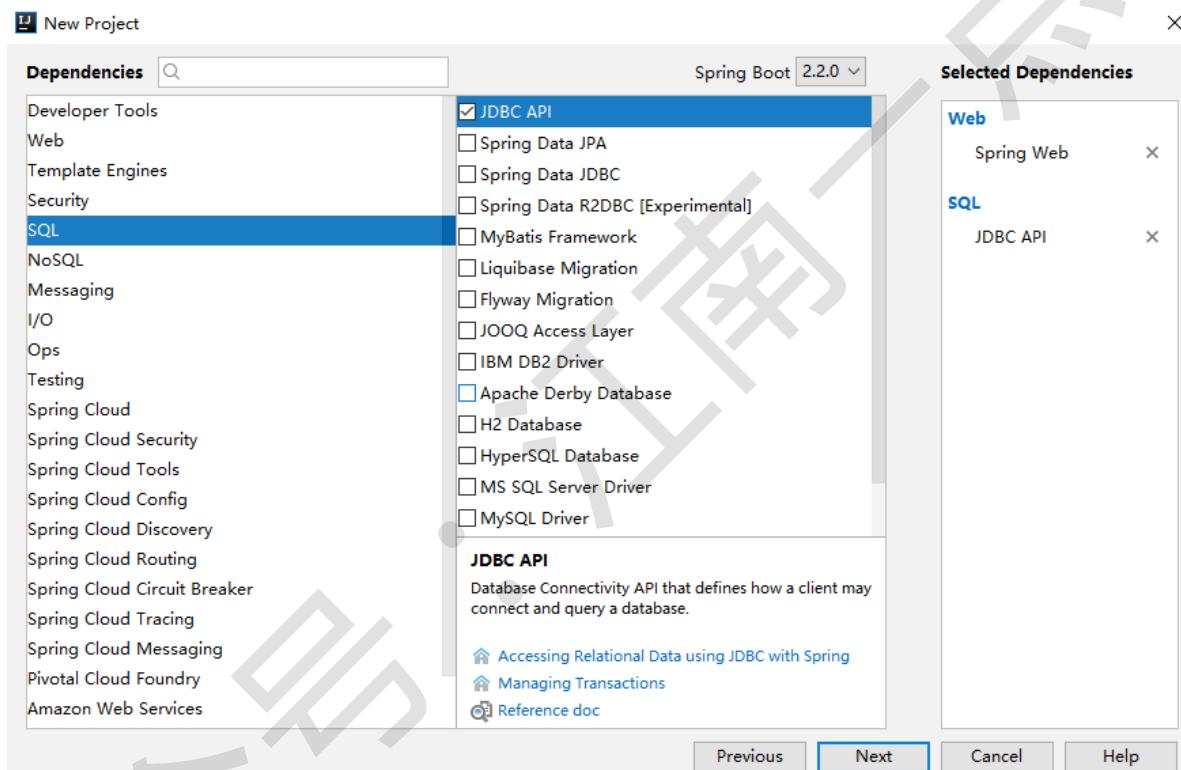


多数据源配置也算是一个常见的开发需求，Spring 和 SpringBoot 中，对此都有相应的解决方案，不过一般来说，如果有多个数据源的需求，我还是建议首选分布式数据库中间件 MyCat 去解决相关问题，之前有小伙伴在我的知识星球上提问，他的数据根据条件的不同，可能保存在四十多个不同的数据库中，怎么办？这种场景下使用多数据源其实就有些费事了，我给的建议是使用 MyCat，然后分表策略使用 sharding-by-intfile。

当然如果一些简单的需求，还是可以使用多数据源的，Spring Boot 中，JdbcTemplate、MyBatis 以及 Jpa 都可以配置多数据源，本文就先和大伙聊一聊 JdbcTemplate 中多数据源的配置（关于 JdbcTemplate 的用法，如果还有小伙伴不了解，可以参考我的 [Spring Boot2 系列教程\(十九\)Spring Boot 整合 JdbcTemplate](#)）。

创建工程

首先是创建工程，和前文一样，创建工程时，也是选择 Web、Jdbc 以及 MySQL 驱动，如下图：



创建成功之后，一定接下来手动添加 Druid 依赖，由于这里会需要开发者自己配置 DataSoruce，所以这里必须要使用 druid-spring-boot-starter 依赖，而不是传统的那个 druid 依赖，因为 druid-spring-boot-starter 依赖提供了 DruidDataSourceBuilder 类，这个可以用来构建一个 DataSource 实例，而传统的 Druid 则没有该类。完整的依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.28</version>
```

```
<scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
```

配置数据源

接下来，在 application.properties 中配置数据源，不同于上文，这里的数据源需要配置两个，如下：

```
spring.datasource.one.url=jdbc:mysql://test01?
useUnicode=true&characterEncoding=utf-8
spring.datasource.one.username=root
spring.datasource.one.password=root
spring.datasource.one.type=com.alibaba.druid.pool.DruidDataSource

spring.datasource.two.url=jdbc:mysql://test02?
useUnicode=true&characterEncoding=utf-8
spring.datasource.two.username=root
spring.datasource.two.password=root
spring.datasource.two.type=com.alibaba.druid.pool.DruidDataSource
```

这里通过 one 和 two 对数据源进行了区分，但是加了 one 和 two 之后，这里的配置就没法被 SpringBoot 自动加载了（因为前面的 key 变了），需要我们自己去加载 DataSource 了，此时，需要自己配置一个 DataSourceConfig，用来提供两个 DataSource Bean，如下：

```
@Configuration
public class DataSourceConfig {
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.one")
    DataSource dsOne() {
        return DruidDataSourceBuilder.create().build();
    }
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.two")
    DataSource dsTwo() {
        return DruidDataSourceBuilder.create().build();
    }
}
```

这里提供了两个 Bean，其中 @ConfigurationProperties 是 Spring Boot 提供的类型安全的属性绑定，以第一个 Bean 为例，`@ConfigurationProperties(prefix = "spring.datasource.one")` 表示使用 `spring.datasource.one` 前缀的数据库配置去创建一个 DataSource，这样配置之后，我们就有两个不同的 DataSource，接下来再用这两个不同的 DataSource 去创建两个不同的 JdbcTemplate。

配置 JdbcTemplate 实例

创建 JdbcTemplateConfig 类，用来提供两个不同的 JdbcTemplate 实例，如下：

```

@Configuration
public class JdbcTemplateConfig {
    @Bean
    JdbcTemplate jdbcTemplateOne(@Qualifier("dsOne") DataSource dsOne) {
        return new JdbcTemplate(dsOne);
    }
    @Bean
    JdbcTemplate jdbcTemplateTwo(@Qualifier("dsTwo") DataSource dsTwo) {
        return new JdbcTemplate(dsTwo);
    }
}

```

每一个 JdbcTemplate 的创建都需要一个 DataSource，由于 Spring 容器中现在存在两个 DataSource， 默认使用类型查找，会报错，因此加上 @Qualifier 注解，表示按照名称查找。这里创建了两个 JdbcTemplate 实例，分别对应了两个 DataSource。

接下来直接去使用这个 JdbcTemplate 就可以了。

测试使用

关于 JdbcTemplate 的详细用法大伙可以参考我的上篇文章，这里我主要演示数据源的差异，在 Controller 中注入两个不同的 JdbcTemplate，这两个 JdbcTemplate 分别对应了不同的数据源，如下：

```

@RestController
public class HelloController {
    @Autowired
    @Qualifier("jdbcTemplateOne")
    JdbcTemplate jdbcTemplateOne;
    @Resource(name = "jdbcTemplateTwo")
    JdbcTemplate jdbcTemplateTwo;

    @GetMapping("/user")
    public List<User> getAllUser() {
        List<User> list = jdbcTemplateOne.query("select * from t_user", new
        BeanPropertyRowMapper<>(User.class));
        return list;
    }
    @GetMapping("/user2")
    public List<User> getAllUser2() {
        List<User> list = jdbcTemplateTwo.query("select * from t_user", new
        BeanPropertyRowMapper<>(User.class));
        return list;
    }
}

```

和 DataSource 一样，Spring 容器中的 JdbcTemplate 也是有两个，因此不能通过 byType 的方式注入进来，这里给大伙提供了两种注入思路，一种是使用 @Resource 注解，直接通过 byName 的方式注入进来，另外一种就是 @Autowired 注解加上 @Qualifier 注解，两者联合起来，实际上也是 byName。将 JdbcTemplate 注入进来之后， jdbcTemplateOne 和 jdbcTemplateTwo 此时就代表操作不同的数据源，使用不同的 JdbcTemplate 操作不同的数据源，实现了多数据源配置。

好了，这个问题就先说到这里，感兴趣的小伙伴也可以参考相关案例：

<https://github.com/lenve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。

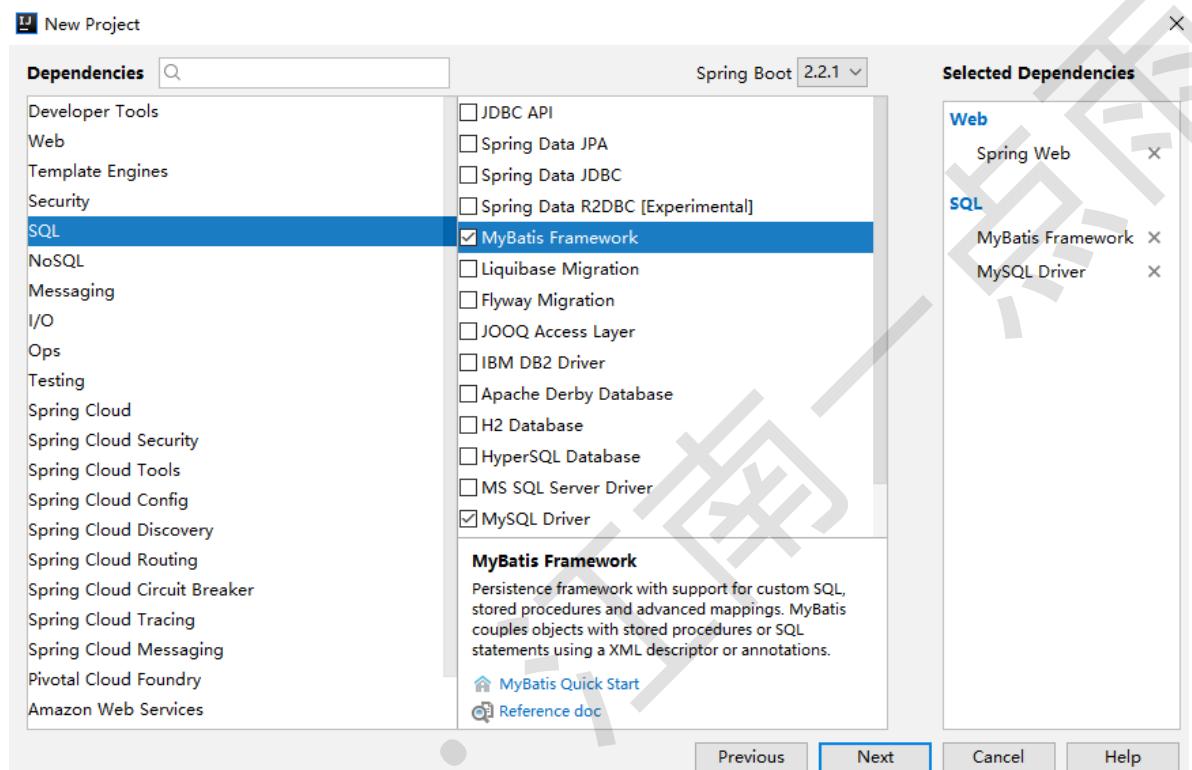


云·空间·时间

前面两篇文章和读者聊了 Spring Boot 中最简单的数据持久化方案 JdbcTemplate，JdbcTemplate 虽然简单，但是用的并不多，因为它没有 MyBatis 方便，在 Spring+SpringMVC 中整合 MyBatis 步骤还是有点复杂的，要配置多个 Bean，Spring Boot 中对此做了进一步的简化，使 MyBatis 基本上可以做到开箱即用，本文就来看看在 Spring Boot 中 MyBatis 要如何使用。

工程创建

首先创建一个基本的 Spring Boot 工程，添加 Web 依赖，MyBatis 依赖以及 MySQL 驱动依赖，如下：



创建成功后，添加Druid依赖，并且锁定MySQL驱动版本，完整的依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.28</version>
    <scope>runtime</scope>
</dependency>
```

如此，工程就算是创建成功了。小伙伴们注意，MyBatis 和 Druid 依赖的命名和其他库的命名不太一样，是属于 xxx-spring-boot-starter 模式的，这表示该 starter 是由第三方提供的。

基本用法

MyBatis 的使用和 JdbcTemplate 基本一致，首先也是在 application.properties 中配置数据库的基本信息：

```
spring.datasource.url=jdbc:mysql:///test01?
useUnicode=true&characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

配置完成后，MyBatis 就可以创建 Mapper 来使用了，例如我这里直接创建一个 UserMapper2，如下：

```
public interface UserMapper2 {
    @Select("select * from user")
    List<User> getAllUsers();

    @Results({
        @Result(property = "id", column = "id"),
        @Result(property = "username", column = "u"),
        @Result(property = "address", column = "a")
    })
    @Select("select username as u,address as a,id as id from user where id=#{id}")
    User getUserById(Long id);

    @Select("select * from user where username like concat('%',#{name},'%')")
    List<User> getUsersByName(String name);

    @Insert({"insert into user(username,address) values(#{username},#{address})"})
    @SelectKey(statement = "select last_insert_id()", keyProperty = "id", before = false, resultType = Integer.class)
    Integer adduser(User user);

    @Update("update user set username=#{username},address=#{address} where id=#{id}")
    Integer updateUserById(User user);

    @Delete("delete from user where id=#{id}")
    Integer deleteUserById(Integer id);
}
```

这里是通过全注解的方式来写 SQL，不写 XML 文件。

@Select、@Insert、@Update 以及 @Delete 四个注解分别对应 XML 中的 select、insert、update 以及 delete 标签，@Results 注解类似于 XML 中的 ResultMap 映射文件（getUserById 方法给查询结果的字段取别名主要是向小伙伴们演示下 @Results 注解的用法）。

另外使用 @SelectKey 注解可以实现主键回填的功能，即当数据插入成功后，插入成功的数据 id 会赋值到 user 对象的 id 属性上。

UserMapper2 创建好之后，还要配置 mapper 扫描，有两种方式，一种是直接在 UserMapper2 上面添加 @Mapper 注解，这种方式有一个弊端就是所有的 Mapper 都要手动添加，要是落下一个就会报错，还有一个一劳永逸的办法就是直接在启动类上添加 Mapper 扫描，如下：

```
@SpringBootApplication
@MapperScan(basePackages = "org.javaboy.mybatisplus.mapper")
public class MybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(MybatisApplication.class, args);
    }
}
```

好了，做完这些工作就可以去测试 Mapper 的使用了。

mapper 映射

当然，开发者也可以在 XML 中写 SQL，例如创建一个 UserMapper，如下：

```
public interface UserMapper {
    List<User> getAllUser();

    Integer addUser(User user);

    Integer updateUserById(User user);

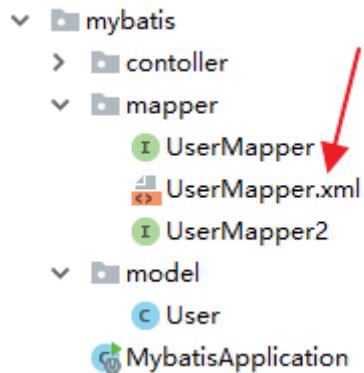
    Integer deleteUserById(Integer id);
}
```

然后创建 UserMapper.xml 文件，如下：

```
<?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="org.javaboy.mybatisplus.UserMapper">
    <select id="getAllUser" resultType="org.javaboy.mybatisplus.model.User">
        select * from t_user;
    </select>
    <insert id="addUser" parameterType="org.javaboy.mybatisplus.model.User">
        insert into user (username,address) values (#{username},#{address});
    </insert>
    <update id="updateUserById" parameterType="org.javaboy.mybatisplus.model.User">
        update user set username=#{username},address=#{address} where id=#{id}
    </update>
    <delete id="deleteUserById">
        delete from user where id=#{id}
    </delete>
</mapper>
```

将接口中方法对应的 SQL 直接写在 XML 文件中。

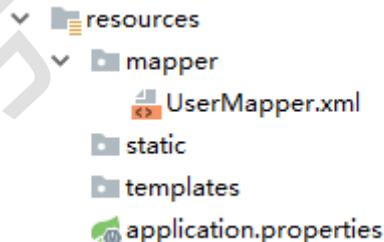
那么这个 UserMapper.xml 到底放在哪里呢？有两个位置可以放，第一个是直接放在 UserMapper 所在的包下面：



放在这里的 UserMapper.xml 会被自动扫描到，但是有另外一个 Maven 带来的问题，就是 java 目录下的 xml 资源在项目打包时会被忽略掉，所以，如果 UserMapper.xml 放在包下，需要在 pom.xml 文件中再添加如下配置，避免打包时 java 目录下的 XML 文件被自动忽略掉：

```
<build>
  <resources>
    <resource>
      <directory>src/main/java</directory>
      <includes>
        <include>**/*.xml</include>
      </includes>
    </resource>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
</build>
```

当然，UserMapper.xml 也可以直接放在 resources 目录下，这样就不用担心打包时被忽略了，但是放在 resources 目录下，必须创建和 Mapper 接口包目录相同的目录层级，这样才能确保打包后 XML 和 Mapper 接口又处于在一起，否则 XML 文件将不能被自动扫描，这个时候就需要添加额外配置。例如我在 resources 目录下创建 mapper 目录用来放 mapper 文件，如下：



此时在 application.properties 中告诉 mybatis 去哪里扫描 mapper：

```
mybatis.mapper-locations=classpath:mapper/*.xml
```

如此配置之后，mapper 就可以正常使用了。**注意这种方式不需要在 pom.xml 文件中配置文件过滤。**

原理分析

在 SSM 整合中，开发者需要自己提供两个 Bean，一个 SqlSessionFactoryBean，还有一个是 MapperScannerConfigurer，在 Spring Boot 中，这两个东西虽然不用开发者自己提供了，但是并不意味着这两个 Bean 不需要了，在

`org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration` 类中，我们可以看到 Spring Boot 提供了这两个 Bean，部分源码如下：

```

@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class })
@ConditionalOnSingleCandidate(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
public class MybatisAutoConfiguration implements InitializingBean {

    @Bean
    @ConditionalOnMissingBean
    public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws
Exception {
        SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
        factory.setDataSource(dataSource);
        return factory.getObject();
    }

    @Bean
    @ConditionalOnMissingBean
    public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory
sqlSessionFactory) {
        ExecutorType executorType = this.properties.getExecutorType();
        if (executorType != null) {
            return new SqlSessionTemplate(sqlSessionFactory, executorType);
        } else {
            return new SqlSessionTemplate(sqlSessionFactory);
        }
    }

    @org.springframework.context.annotation.Configuration
    @Import({ AutoConfiguredMapperScannerRegistrar.class })
    @ConditionalOnMissingBean(MapperFactoryBean.class)
    public static class MapperScannerRegistrarNotFoundConfiguration implements
InitializingBean {

        @Override
        public void afterPropertiesSet() {
            logger.debug("No {} found.", MapperFactoryBean.class.getName());
        }
    }
}

```

从类上的注解可以看出，当当前类路径下存在 SqlSessionFactory、SqlSessionFactoryBean 以及 DataSource 时，这里的配置才会生效，SqlSessionFactory 和 SqlTemplate 都被提供了。为什么要看这段代码呢？下篇文章，松哥和大伙分享 Spring Boot 中 MyBatis 多数据源的配置时，这里将是一个重要的参考。

好了，本文就先说到这里，本文相关案例，大家可以在 GitHub 上下载：

<https://github.com/lenvve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



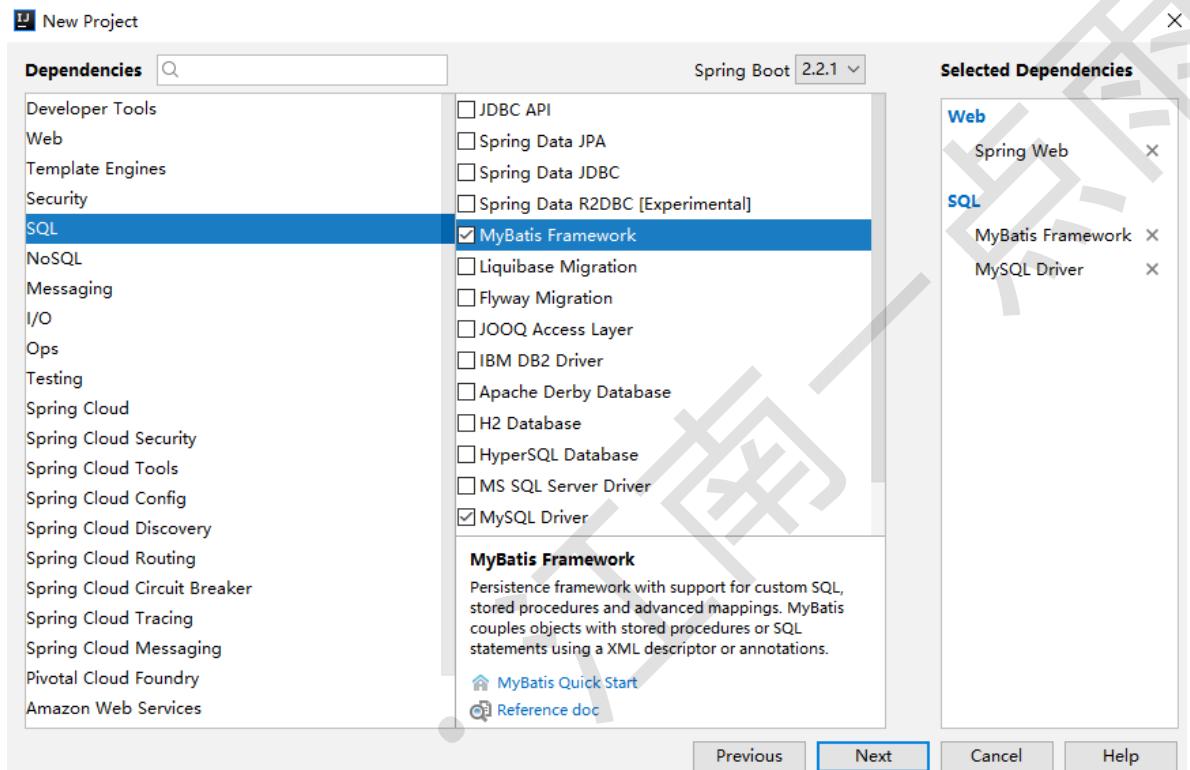
云·空间·时间

关于多数据源的配置，前面和大伙介绍过 JdbcTemplate 多数据源配置，那个比较简单，本文来和大伙说说 MyBatis 多数据源的配置。

其实关于多数据源，我的态度还是和之前一样，复杂的就直接上分布式数据库中间件，简单的再考虑多数据源。这是项目中的建议，技术上的话，当然还是各种技术都要掌握的。

工程创建

首先需要创建 MyBatis 项目，项目创建和前文的一样，添加 MyBatis、MySQL 以及 Web 依赖：



项目创建完成后，添加 Druid 依赖，和 JdbcTemplate 一样，这里添加 Druid 依赖也必须是专为 Spring Boot 打造的 Druid，不能使用传统的 Druid。完整的依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.28</version>
    <scope>runtime</scope>
</dependency>
```

多数据源配置

接下来配置多数据源，这里基本上还是和 JdbcTemplate 多数据源的配置方式一致，首先在 application.properties 中配置数据库基本信息，然后提供两个 DataSource 即可，这里我再把代码贴出来，里边的道理条条框框的，大伙可以参考前面的文章，这里不再赘述。

application.properties 中的配置：

```
spring.datasource.one.url=jdbc:mysql://test01?  
useUnicode=true&characterEncoding=utf-8  
spring.datasource.one.username=root  
spring.datasource.one.password=root  
spring.datasource.one.type=com.alibaba.druid.pool.DruidDataSource  
  
spring.datasource.two.url=jdbc:mysql://test02?  
useUnicode=true&characterEncoding=utf-8  
spring.datasource.two.username=root  
spring.datasource.two.password=root  
spring.datasource.two.type=com.alibaba.druid.pool.DruidDataSource
```

然后再提供两个 DataSource，如下：

```
@Configuration  
public class DataSourceConfig {  
    @Bean  
    @ConfigurationProperties(prefix = "spring.datasource.one")  
    DataSource dsOne() {  
        return DruidDataSourceBuilder.create().build();  
    }  
    @Bean  
    @ConfigurationProperties(prefix = "spring.datasource.two")  
    DataSource dsTwo() {  
        return DruidDataSourceBuilder.create().build();  
    }  
}
```

MyBatis 配置

接下来则是 MyBatis 的配置，不同于 JdbcTemplate，MyBatis 的配置要稍微麻烦一些，因为要提供两个 Bean，因此这里两个数据源我将在两个类中分开来配置，首先来看第一个数据源的配置：

```
@Configuration  
@MapperScan(basePackages = "org.javaboy.mybatis.mapper1",sqlSessionFactoryRef =  
"sqlSessionFactory1",sqlSessionTemplateRef = "sqlSessionTemplate1")  
public class MyBatisConfigone {  
    @Resource(name = "dsOne")  
    DataSource dsOne;  
  
    @Bean  
    SqlSessionFactory sqlSessionFactory1() {  
        SqlSessionFactory sessionFactory = null;  
        try {  
            SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
```

```

        bean.setDataSource(dsOne);
        sessionFactory = bean.getObject();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return sessionFactory;
}
@Bean
SqlSessionTemplate sqlSessionTemplate1() {
    return new SqlSessionTemplate(sqlSessionFactory1());
}
}

```

创建 MyBatisConfigOne 类，首先指明该类是一个配置类，配置类中要扫描的包是 org.javaboy.mybatis.mapper1，即该包下的 Mapper 接口将操作 dsOne 中的数据，对应的 SqlSessionFactory 和 SqlSessionTemplate 分别是 sqlSessionFactory1 和 sqlSessionTemplate1，在 MyBatisConfigOne 内部，分别提供 SqlSessionFactory 和 SqlSessionTemplate 即可，SqlSessionFactory 根据 dsOne 创建，然后再根据创建好的 SqlSessionFactory 创建一个 SqlSessionTemplate。

这里配置完成后，依据这个配置，再来配置第二个数据源即可：

```

@Configuration
@MapperScan(basePackages = "org.javaboy.mybatis.mapper2",sqlSessionFactoryRef =
"sqlSessionFactory2",sqlSessionTemplateRef = "sqlSessionTemplate2")
public class MyBatisConfigTwo {
    @Resource(name = "dsTwo")
    DataSource dsTwo;

    @Bean
    SqlSessionFactory sqlSessionFactory2() {
        SqlSessionFactory sessionFactory = null;
        try {
            SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
            bean.setDataSource(dsTwo);
            sessionFactory = bean.getObject();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return sessionFactory;
    }
    @Bean
    SqlSessionTemplate sqlSessionTemplate2() {
        return new SqlSessionTemplate(sqlSessionFactory2());
    }
}

```

好了，这样 MyBatis 多数据源基本上就配置好了，接下来只需要在 org.javaboy.mybatis.mapper1 和 org.javaboy.mybatis.mapper2 包中提供不同的 Mapper，Service 中注入不同的 Mapper 就可以操作不同的数据源。

mapper 创建

org.javaboy.mybatis.mapper1 中的 mapper：

```
public interface UserMapperOne {  
    List<User> getAllUser();  
}
```

对应的 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="org.javaboy.mybatis.mapper1.UserMapperOne">  
    <select id="getAllUser" resultType="org.javaboy.mybatis.model.User">  
        select * from t_user;  
    </select>  
</mapper>
```

org.javaboy.mybatis.mapper2 中的 mapper:

```
public interface UserMapper {  
    List<User> getAllUser();  
}
```

对应的 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="org.javaboy.mybatis.mapper2.UserMapper">  
    <select id="getAllUser" resultType="org.javaboy.mybatis.model.User">  
        select * from t_user;  
    </select>  
</mapper>
```

接下来，在 Service 中注入两个不同的 Mapper，不同的 Mapper 将操作不同的数据源。

好了，关于 MyBatis 多数据源本文就先说到这里。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

有很多读者留言希望松哥能好好聊聊 Spring Data Jpa! 其实这个话题松哥以前零零散散的介绍过，在我的书里也有介绍过，但是在公众号中还没和大伙聊过，因此本文就和大家来仔细聊聊 Spring Data 和 Jpa!

本文大纲：

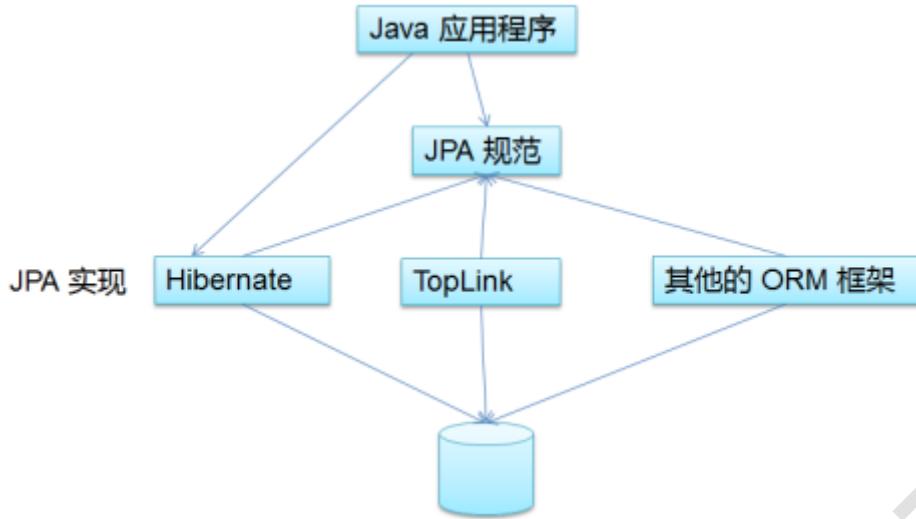
- 1. 故事的主角**
 - 1.1 Jpa
 - 1.1.1 JPA 是什么
 - 1.1.2 JPA 和 Hibernate 的关系
 - 1.1.3 JPA 的供应商
 - 1.1.4 JPA 的优势
 - 1.1.5 JPA 包含的技术
 - 1.2 Spring Data
- 2. 主角的故事**
 - 2.1 Jpa 的故事
 - 2.1.1 关于 JPQL
 - 2.1.2 JPQL 举例
 - 2.2 Spring Data 的故事
 - 2.2.1 基本环境搭建
 - 2.2.2 Repository
 - 2.2.3 方法定义规范
 - 2.2.3.1 简单条件查询
 - 2.2.3.2 支持的关键字
 - 2.2.3.3 查询方法流程解析
 - 2.2.3.4 @Query 注解
 - 2.2.3.5 @Modifying 注解

1. 故事的主角

1.1 Jpa

1.1.1 JPA 是什么

1. Java Persistence API：用于对象持久化的 API
2. Java EE 5.0 平台标准的 ORM 规范，使得应用程序以统一的方式访问持久层



1.1.2 JPA 和 Hibernate 的关系

1. JPA 是 Hibernate 的一个抽象（就像 JDBC 和 JDBC 驱动的关系）；
2. JPA 是规范：JPA 本质上就是一种 ORM 规范，不是 ORM 框架，这是因为 JPA 并未提供 ORM 实现，它只是制订了一些规范，提供了一些编程的 API 接口，但具体实现则由 ORM 厂商提供实现；
3. Hibernate 是实现：Hibernate 除了作为 ORM 框架之外，它也是一种 JPA 实现
4. 从功能上来说，JPA 是 Hibernate 功能的一个子集

1.1.3 JPA 的供应商

JPA 的目标之一是制定一个可以由很多供应商实现的 API，Hibernate 3.2+、TopLink 10.1+ 以及 OpenJPA 都提供了 JPA 的实现，Jpa 供应商有很多，常见的有如下四种：

1. Hibernate：JPA 的始作俑者就是 Hibernate 的作者，Hibernate 从 3.2 开始兼容 JPA。
2. OpenJPA：OpenJPA 是 Apache 组织提供的开源项目。
3. TopLink：TopLink 以前需要收费，如今开源了。
4. EclipseLink

1.1.4 JPA 的优势

1. 标准化：提供相同的 API，这保证了基于 JPA 开发的企业应用能够经过少量的修改就能够再不同的 JPA 框架下运行。
2. 简单易用，集成方便：JPA 的主要目标之一就是提供更加简单的编程模型，在 JPA 框架下创建实体和创建 Java 类一样简单，只需要使用 javax.persistence.Entity 进行注解；JPA 的框架和接口也都非常简单。
3. 可媲美 JDBC 的查询能力：JPA 的查询语言是面向对象的，JPA 定义了独特的 JPQL，而且能够支持批量更新和修改、JOIN、GROUP BY、HAVING 等通常只有 SQL 才能够提供的高级查询特性，甚至还能够支持子查询。
4. 支持面向对象的高级特性：JPA 中能够支持面向对象的高级特性，如类之间的继承、多态和类之间的复杂关系，最大限度的使用面向对象的模型

1.1.5 JPA 包含的技术

1. ORM 映射元数据：JPA 支持 XML 和 JDK 5.0 注解两种元数据的形式，元数据描述对象和表之间的映射关系，框架据此将实体对象持久化到数据库表中。
2. JPA 的 API：用来操作实体对象，执行 CRUD 操作，框架在后台完成所有的事情，开发者从繁琐的 JDBC 和 SQL 代码中解脱出来。
3. 查询语言 (JPQL)：这是持久化操作中很重要的一个方面，通过面向对象而非面向数据库的查询语言查询数据，避免程序和具体的 SQL 紧密耦合。

1.2 Spring Data

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

- SpringData 项目支持 NoSQL 存储：
 1. MongoDB (文档数据库)
 2. Neo4j (图形数据库)
 3. Redis (键/值存储)
 4. Hbase (列族数据库)
- SpringData 项目所支持的关系数据存储技术：
 1. JDBC
 2. JPA
- Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量. 开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成
- 框架怎么可能代替开发者实现业务逻辑呢？比如：当有一个UserDao.findUserById() 这样一个方法声明，大致应该能判断出这是根据给定条件的 ID 查询出满足条件的 User 对象。Spring Data JPA 做的便是规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

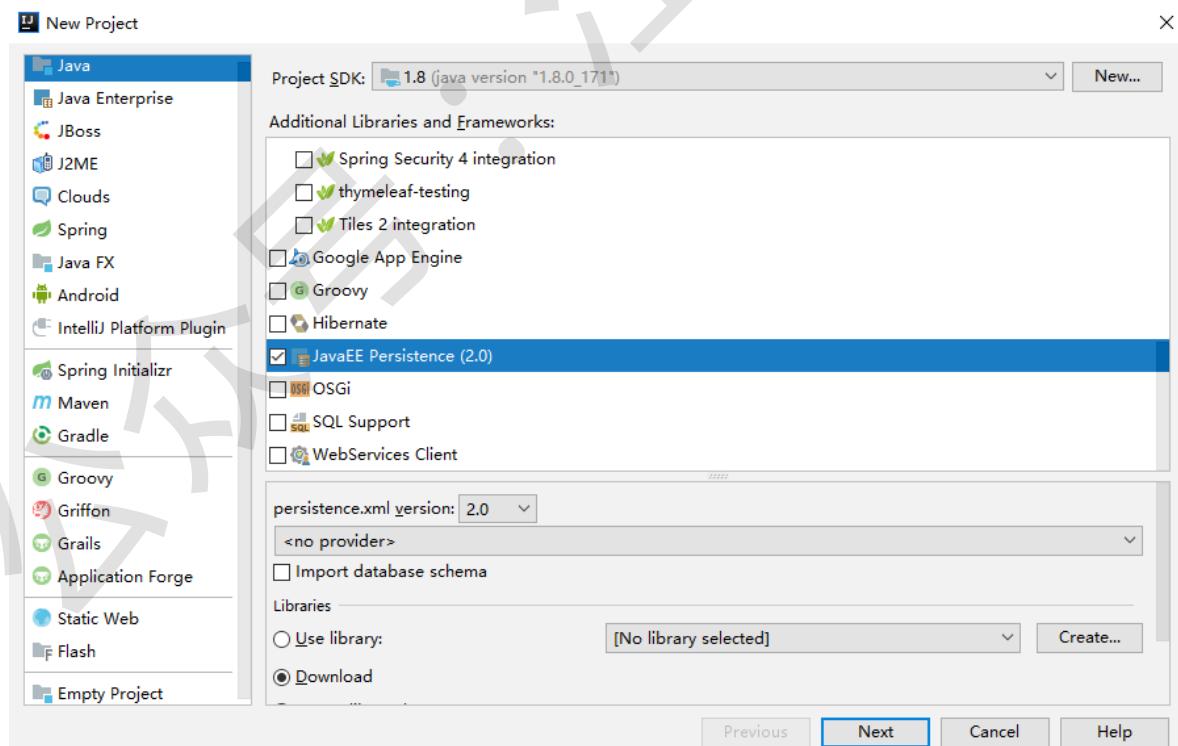
2. 主角的故事

2.1 Jpa 的故事

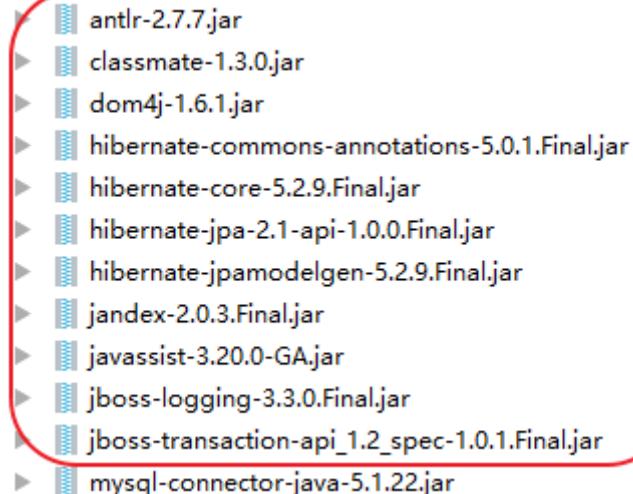
为了让大伙彻底把这两个东西学会，这里我就先来介绍单纯的 Jpa 使用，然后我们再结合 Spring Data 来看 Jpa 如何使用。

整体步骤如下：

- 1. 使用 IntelliJ IDEA 创建项目，创建时选择 JavaEE Persistence，如下：



- 2. 创建成功后，添加依赖 jar，由于 Jpa 只是一个规范，因此我们说用 Jpa 实际上必然是用 Jpa 的某一种实现，那么是哪一种实现呢？当然就是 Hibernate 了，所以添加的 jar，实际上来自 Hibernate，如下：



- 3.添加实体类

接下来在项目中添加实体类，如下：

```
@Entity(name = "t_book")
public class Book {
    private Long id;
    private String name;
    private String author;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public Long getId() {
        return id;
    }
    // 省略其他getter/setter
}
```

首先 @Entity 注解表示这是一个实体类，那么在项目启动时会自动针对该类生成一张表，默认的表名为类名，@Entity 注解的 name 属性表示自定义生成的表名。@Id 注解表示这个字段是一个 id，@GeneratedValue 注解表示主键的自增长策略，对于类中的其他属性，默认都会根据属性名在表中生成相应的字段，字段名和属性名相同，如果开发者想要对字段进行定制，可以使用 @Column 注解，去配置字段的名称，长度，是否为空等等。

- 4.创建 persistence.xml 文件

JPA 规范要求在类路径的 META-INF 目录下放置 persistence.xml，文件的名称是固定的

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="NewPersistenceUnit" transaction-
type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>org.javaboy.Book</class>
        <properties>
            <property name="hibernate.connection.url"
                value="jdbc:mysql:///jpa01?
useUnicode=true&characterEncoding=UTF-8"/>
            <property name="hibernate.connection.driver_class"
value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password" value="123"/>
```

```
<property name="hibernate.archive.autodetection" value="class"/>
<property name="hibernate.show_sql" value="true"/>
<property name="hibernate.format_sql" value="true"/>
<property name="hibernate.hbm2ddl.auto" value="update"/>
</properties>
</persistence-unit>
</persistence>
```

注意：

1. persistence-unit 的 name 属性用于定义持久化单元的名字，必填。
2. transaction-type：指定 JPA 的事务处理策略。RESOURCE_LOCAL：默认值，数据库级别的事务，只能针对一种数据库，不支持分布式事务。如果需要支持分布式事务，使用 JTA：transaction-type="JTA"
3. class 节点表示显式的列出实体类
4. properties 中的配置分为两部分：数据库连接信息以及 Hibernate 信息
 - 5. 执行持久化操作

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("NewPersistenceUnit");
EntityManager manager = entityManagerFactory.createEntityManager();
EntityTransaction transaction = manager.getTransaction();
transaction.begin();
Book book = new Book();
book.setAuthor("罗贯中");
book.setName("三国演义");
manager.persist(book);
transaction.commit();
manager.close();
entityManagerFactory.close();
```

这里首先根据配置文件创建出来一个 EntityManagerFactory，然后再根据 EntityManagerFactory 的实例创建出来一个 EntityManager，然后再开启事务，调用 EntityManager 中的 persist 方法执行一次持久化操作，最后提交事务，执行完这些操作后，数据库中旧多出来一个 t_book 表，并且表中有一条数据。

2.1.1 关于 JPQL

1. JPQL 语言，即 Java Persistence Query Language 的简称。JPQL 是一种和 SQL 非常类似的中间性和对象化查询语言，它最终会被编译成针对不同底层数据库的 SQL 查询，从而屏蔽不同数据库的差异。JPQL 语言的语句可以是 select 语句、update 语句或 delete 语句，它们都通过 Query 接口封装执行。
2. Query 接口封装了执行数据库查询的相关方法。调用 EntityManager 的 createQuery、createNamedQuery 及 createNativeQuery 方法可以获得查询对象，进而可调用 Query 接口的相关方法来执行查询操作。
3. Query 接口的主要方法如下：
 - **int executeUpdate();** | 用于执行 update 或 delete 语句。
 - **List getResultList();** | 用于执行 select 语句并返回结果集实体列表。
 - **Object getSingleResult();** | 用于执行只返回单个结果实体的 select 语句。
 - **Query setFirstResult(int startPosition);** | 用于设置从哪个实体记录开始返回查询结果。
 - **Query setMaxResults(int maxResult);** | 用于设置返回结果实体的最大数。与 setFirstResult 结合使用可实现分页查询。

- **Query setFlushMode(FlushModeType flushMode);** | 设置查询对象的Flush模式。参数可以取2个枚举值：FlushModeType.AUTO 为自动更新数据库记录，FlushMode Type.COMMIT 为直到提交事务时才更新数据库记录。
- **setHint(String hintName, Object value);** | 设置与查询对象相关的特定供应商参数或提示信息。参数名及其取值需要参考特定 JPA 实现库提供商的文档。如果第二个参数无效将抛出 IllegalArgumentException 异常。
- **setParameter(int position, Object value);** | 为查询语句的指定位置参数赋值。Position 指定参数序号，value 为赋给参数的值。
- **setParameter(int position, Date d, TemporalType type);** | 为查询语句的指定位置参数赋 Date 值。Position 指定参数序号，value 为赋给参数的值，temporalType 取 TemporalType 的枚举常量，包括 DATE、TIME 及 TIMESTAMP 三个，，用于将 Java 的 Date 型值临时转换为数据库支持的日期时间类型（java.sql.Date、java.sql.Time 及 java.sql.Timestamp）。
- **setParameter(int position, Calendar c, TemporalType type);** | 为查询语句的指定位置参数赋 Calendar 值。position 指定参数序号，value 为赋给参数的值，temporalType 的含义及取舍同前。
- **setParameter(String name, Object value);** | 为查询语句的指定名称参数赋值。
- **setParameter(String name, Date d, TemporalType type);** | 为查询语句的指定名称参数赋 Date 值，用法同前。
- **setParameter(String name, Calendar c, TemporalType type);** | 为查询语句的指定名称参数设置 Calendar 值。name 为参数名，其它同前。该方法调用时如果参数位置或参数名不正确，或者所赋的参数值类型不匹配，将抛出 IllegalArgumentException 异常。

2.1.2 JPQL 举例

和在 SQL 中一样，JPQL 中的 select 语句用于执行查询。其语法可表示为：

```
select_clause from_clause [where_clause] [groupby_clause] [having_clause]
[orderby_clause]
```

其中：

1. from 子句是查询语句的必选子句。
2. select 用来指定查询返回的结果实体或实体的某些属性。
3. from 子句声明查询源实体类，并指定标识符变量（相当于SQL表的别名）。
4. 如果不希望返回重复实体，可使用关键字 distinct 修饰。select、from 都是 JPQL 的关键字，通常全大写或全小写，建议不要大小写混用。

在 JPQL 中，查询所有实体的 JPQL 查询语句很简单，如下：

```
select o from Order o
```

或

```
select o from Order as o
```

这里关键字 as 可以省去，标识符变量的命名规范与 Java 标识符相同，且区分大小写，调用 EntityManager 的 createQuery() 方法可创建查询对象，接着调用 Query 接口的 getResultList() 方法就可获得查询结果集，如下：

```
Query query = entityManager.createQuery("select o from Order o");
List orders = query.getResultList();
Iterator iterator = orders.iterator();
while(iterator.hasNext()) {
    // 处理Order
}
```

其他方法的与此类似，这里不再赘述。

2.2 Spring Data 的故事

在 Spring Boot 中，Spring Data Jpa 官方封装了太多东西了，导致很多人用的时候不知道底层到底是怎么配置的，本文就和大伙来看看在手工的 Spring 环境下，Spring Data Jpa 要怎么配置，配置完成后，用法和 Spring Boot 中的用法是一致的。

2.2.1 基本环境搭建

首先创建一个普通的 Maven 工程，并添加如下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-oxm</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.27</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-support</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-expression</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.2.12.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-jpamodelgen</artifactId>
        <version>5.2.12.Final</version>
    </dependency>
```

```

        </dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.0.29</version>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
    <version>1.11.3.RELEASE</version>
</dependency>
</dependencies>

```

这里除了 Jpa 的依赖之外，就是 Spring Data Jpa 的依赖了。

接下来创建一个 User 实体类，创建方式参考 Jpa 中实体类的创建方式，这里不再赘述。

接下来在 resources 目录下创建一个 applicationContext.xml 文件，并配置 Spring 和 Jpa，如下：

```

<context:property-placeholder location="classpath:db.properties"/>
<context:component-scan base-package="org.javaboy"/>
<bean class="com.alibaba.druid.pool.DruidDataSource" id="dataSource">
    <property name="driverClassName" value="${db.driver}"/>
    <property name="url" value="${db.url}"/>
    <property name="username" value="${db.username}"/>
    <property name="password" value="${db.password}"/>
</bean>
<bean class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" id="entityManagerFactory">
    <property name="dataSource" ref="dataSource"/>
    <property name="jpaVendorAdapter">
        <bean
            class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter"/>
        </property>
        <property name="packagesToScan" value="org.javaboy.model"/>
        <property name="jpaProperties">
            <props>
                <prop key="hibernate.show_sql">true</prop>
                <prop key="hibernate.format_sql">true</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop
                    key="hibernate.dialect">org.hibernate.dialect.MySQL57Dialect</prop>
            </props>
        </property>
    </bean>
    <bean class="org.springframework.orm.jpa.JpaTransactionManager" id="transactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager"/>
    <!-- 配置jpa -->
    <jpa:repositories base-package="org.javaboy.dao" entity-manager-factory-ref="entityManagerFactory"/>

```

这里和 Jpa 相关的配置主要是三个：

- 一个是 entityManagerFactory

- 一个是 Jpa 的事务
- 一个是配置 dao 的位置

配置完成后，就可以在 org.javaboy.dao 包下创建相应的 Repository 了，如下：

```
public interface UserDao extends Repository<User, Long> {  
    User getUserById(Long id);  
}
```

getUserById 表示根据 id 去查询 User 对象，只要我们的方法名称符合类似的规范，就不需要写 SQL，具体的规范一会来说。好了，接下来，创建 Service 和 Controller 来调用这个方法，如下：

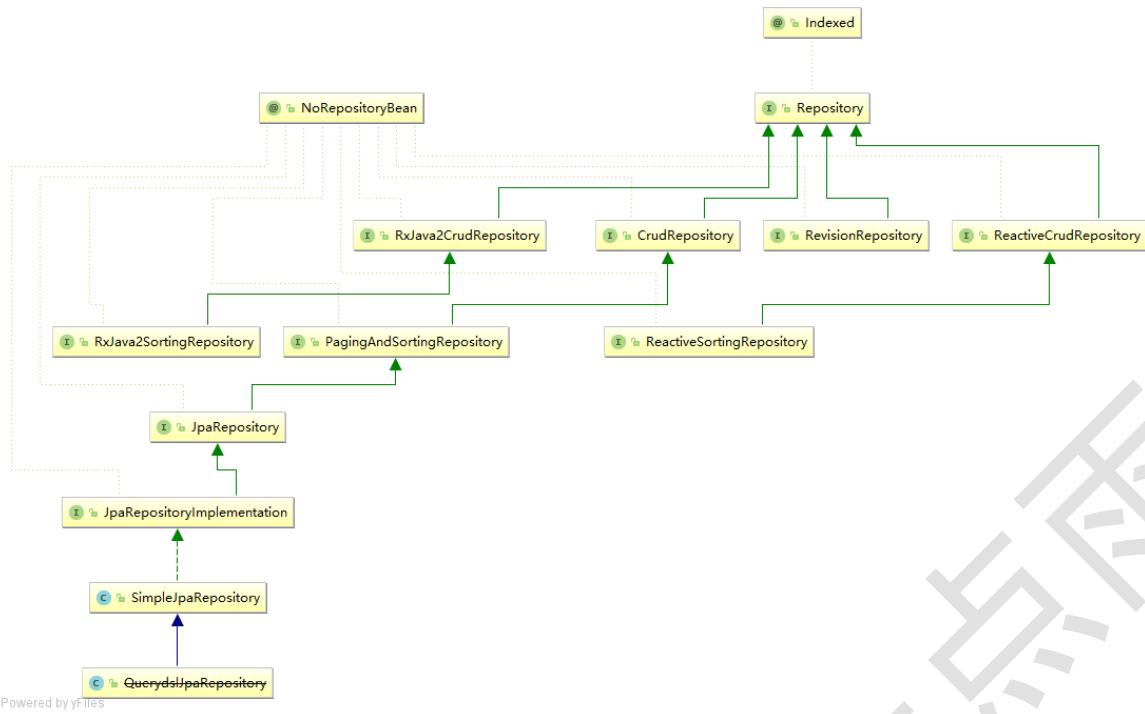
```
@Service  
@Transactional  
public class UserService {  
    @Resource  
    UserDao userDao;  
  
    public User getUserById(Long id) {  
        return userDao.getUserById(id);  
    }  
    public void test1() {  
        ClassPathXmlApplicationContext ctx = new  
        ClassPathXmlApplicationContext("applicationContext.xml");  
        UserService userService = ctx.getBean(UserService.class);  
        User user = userService.getUserById(1L);  
        System.out.println(user);  
    }  
}
```

这样，就可以查询到 id 为 1 的用户了。

2.2.2 Repository

上文我们自定义的 UserDao 实现了 Repository 接口，这个 Repository 接口是什么来头呢？

首先来看 Repository 的一个继承关系图：



可以看到，实现类不少。那么到底如何理解 Repository 呢？

1. Repository 接口是 Spring Data 的一个核心接口，它不提供任何方法，开发者需要在自己定义的接口中声明需要的方法 `public interface Repository<T, ID extends Serializable> { }`
2. 若我们定义的接口继承了 Repository，则该接口会被 IOC 容器识别为一个 Repository Bean，进而纳入到 IOC 容器中，进而可以在该接口中定义满足一定规范的方法。
3. Spring Data可以让我们只定义接口，只要遵循 Spring Data 的规范，就无需写实现类。
4. 与继承 Repository 等价的一种方式，就是在持久层接口上使用 `@RepositoryDefinition` 注解，并为其指定 domainClass 和 idClass 属性。像下面这样：

```

@RepositoryDefinition(domainClass = User.class, idClass = Long.class)
public interface UserDao
{
    User findById(Long id);
    List<User> findAll();
}

```

基础的 Repository 提供了最基本的数据访问功能，其几个子接口则扩展了一些功能，它的几个常用的实现类如下：

- CrudRepository：继承 Repository，实现了一组 CRUD 相关的方法
- PagingAndSortingRepository：继承 CrudRepository，实现了一组分页排序相关的方法
- JpaRepository：继承 PagingAndSortingRepository，实现一组 JPA 规范相关的方法
- 自定义的 XxxRepository 需要继承 JpaRepository，这样的 XxxRepository 接口就具备了通用的数据访问控制层的能力。
- JpaSpecificationExecutor：不属于Repository 体系，实现一组 JPA Criteria 查询相关的方法

2.2.3 方法定义规范

2.2.3.1 简单条件查询

- 按照 Spring Data 的规范，查询方法以 `find | read | get` 开头
- 涉及条件查询时，条件的属性用条件关键字连接，要注意的是：条件属性以首字母大写

例如：定义一个 Entity 实体类：

```
class User {  
    private String firstName;  
    private String lastName;  
}
```

使用 And 条件连接时，条件的属性名称与个数要与参数的位置与个数一一对应，如下：

```
findByLastNameAndFirstName(String lastName, String firstName);
```

- 支持属性的级联查询。若当前类有符合条件的属性，则优先使用，而不使用级联属性。若需要使用级联属性，则属性之间使用 `&` 进行连接。

查询举例：

- 按照 id 查询

```
User getUserById(Long id);  
User getById(Long id);
```

- 查询所有年龄小于 90 岁的人

```
List<User> findByAgeLessThan(Long age);
```

- 查询所有姓赵的人

```
List<User> findByUsernameStartingWith(String u);
```

- 查询所有姓赵的、并且 id 大于 50 的人

```
List<User> findByUsernameStartingWithAndIdGreaterThan(String name, Long id);
```

- 查询所有姓名中包含"上"字的人

```
List<User> findByUsernameContaining(String name);
```

- 查询所有姓赵的或者年龄大于 90 岁的

```
List<User> findByUsernameStartingWithOrAgeGreaterThan(String name, Long age);
```

- 查询所有角色为 1 的用户

```
List<User> findByRole_Id(Long id);
```

2.2.3.2 支持的关键字

支持的查询关键字如下图：

KeyWords	方法命名举例	对应的 SQL
And	findByNameAndAge	where name= ? and age =?
Or	findByNameOrAge	where name= ? or age=?
Is	findByAgeIs	where age= ?
Equals	findByIdEquals	where id= ?
Between	findByAgeBetween	where age between ? and ?
LessThan	findByAgeLessThan	where age < ?
LessThanEquals	findByAgeLessThanEquals	where age <= ?
GreaterThan	findByAgeGreaterThan	where age > ?
GreaterThanOrEqual	findByAgeGreaterThanOrEqual	where age >= ?
After	findByAgeAfter	where age > ?
Before	findByAgeBefore	where age < ?
IsNull	findByNameIsNull	where name is null
isNotNull,NotNull	findByNameNotNull	where name is not null
Not	findByGenderNot	where gender <> ?
In	findByAgeIn	where age in (?)
NotIn	findByAgeNotIn	where age not in (?)
NotLike	findByNameNotLike	where name not like ?
Like	findByNameLike	where name like ?
StartingWith	findByNameStartingWith	where name like '?%'
EndingWith	findByNameEndingWith	where name like '%?'
Containing,Contains	findByNameContaining	where name like '%?%'
OrderBy	findByAgeGreaterThanOrEqualOrderByDesc	where age>? order by id desc
True	findByEnabledTrue	where enabled= true
False	findByEnabledFalse	where enabled= false
IgnoreCase	findByNameIgnoreCase	where UPPER(name)=UPPER(?)

2.2.3.3 查询方法流程解析

为什么写上方法名，JPA就知道你想干嘛了呢？假如创建如下的查询：``findByUserDepUuid()``，框架在解析该方法时，首先剔除 `findBy`，然后对剩下的属性进行解析，假设查询实体为 `Doc`：

- 先判断 `userDepUuid`（根据 POJO 规范，首字母变为小写）是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，继续第二步；
- 从右往左截取第一个大写字母开头的字符串（此处为 `Uuid`），然后检查剩下的字符串是否为查询实体的一个属性，如果是，则表示根据该属性进行查询；如果没有该属性，则重复第二步，继续从右往左截取；最后假设 `user` 为查询实体的一个属性；
- 接着处理剩下部分（`DepUuid`），先判断 `user` 所对应的类型是否有 `depUuid` 属性，如果有，则表示该方法最终是根据 “`Doc.user.depUuid`” 的取值进行查询；否则继续按照步骤 2 的规则从右往左截取，最终表示根据 “`Doc.user.dep.uuid`” 的值进行查询。
- 可能会存在一种特殊情况，比如 `Doc` 包含一个 `user` 的属性，也有一个 `userDep` 属性，此时会存在混淆。可以明确在属性之间加上 `“_”` 以显式表达意图，比如 `“`findByUser_DepUuid()`” 或者 `“`findByUserDep_uuid()`”`
- 还有一些特殊的参数：例如分页或排序的参数：

```
Page<UserModel> findByName(String name, Pageable pageable);
List<UserModel> findByName(String name, Sort sort);
```

2.2.3.4 @Query 注解

有的时候，这里提供的查询关键字并不能满足我们的查询需求，这个时候就可以使用 `@Query` 关键字，来自定义查询 SQL，例如查询 Id 最大的 User：

```
@Query("select u from t_user u where id=(select max(id) from t_user)")  
User getMaxIdUser();
```

如果查询有参数的话，参数有两种不同的传递方式，

- 利用下标索引传参，索引参数如下所示，索引值从1开始，查询中 "?X" 个数需要与方法定义的参数个数相一致，并且顺序也要一致：

```
@Query("select u from t_user u where id>?1 and username like ?2")  
List<User> selectUserByParam(Long id, String name);
```

- 命名参数（推荐）：这种方式可以定义好参数名，赋值时采用@Param("参数名")，而不用管顺序：

```
@Query("select u from t_user u where id>:id and username like :name")  
List<User> selectUserByParam2(@Param("name") String name, @Param("id") Long id);
```

查询时候，也可以是使用原生的 SQL 查询，如下：

```
@Query(value = "select * from t_user", nativeQuery = true)  
List<User> selectAll();
```

2.2.3.5 @Modifying 注解

涉及到数据修改操作，可以使用 @Modifying 注解，@Query 与 @Modifying 这两个 annotation 一起声明，可定义个性化更新操作，例如涉及某些字段更新时最为常用，示例如下：

```
@Modifying  
@Query("update t_user set age=:age where id>:id")  
int updateUserById(@Param("age") Long age, @Param("id") Long id);
```

注意：

- 可以通过自定义的 JPQL 完成 UPDATE 和 DELETE 操作。注意：JPQL 不支持使用 INSERT
- 方法的返回值应该是 int，表示更新语句所影响的行数
- 在调用的地方必须加事务，没有事务不能正常执行
- 默认情况下，Spring Data 的每个方法上有事务，但都是一个只读事务。他们不能完成修改操作

说到这里，再来顺便说说 Spring Data 中的事务问题：

- Spring Data 提供了默认的事务处理方式，即所有的查询均声明为只读事务。
- 对于自定义的方法，如需改变 Spring Data 提供的事务默认方式，可以在方法上添加 @Transactional 注解。
- 进行多个 Repository 操作时，也应该使它们在同一个事务中处理，按照分层架构的思想，这部分属于业务逻辑层，因此，需要在 Service 层实现对多个 Repository 的调用，并在相应的方法上声明事务。

好了，关于 Spring Data Jpa 本文就先说这么多。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

Spring Boot 中的数据持久化方案前面给大伙介绍了两种了，一个是 JdbcTemplate，还有一个 MyBatis，JdbcTemplate 配置简单，使用也简单，但是功能也非常有限，MyBatis 则比较灵活，功能也很强大，据我所知，公司采用 MyBatis 做数据持久化的相当多，但是 MyBatis 并不是唯一的解决方案，除了 MyBatis 之外，还有另外一个东西，那就是 Jpa，松哥也有一些朋友在公司里使用 Jpa 来做数据持久化，本文就和大伙来说说 Jpa 如何实现数据持久化。

Jpa 介绍

首先需要向大伙介绍一下 Jpa，Jpa (Java Persistence API) Java 持久化 API，它是一套 ORM 规范，而不是具体的实现，Jpa 的江湖地位类似于 JDBC，只提供规范，所有的数据库厂商提供实现（即具体的数据库驱动），Java 领域，小伙伴们熟知的 ORM 框架可能主要是 Hibernate，实际上，除了 Hibernate 之外，还有很多其他的 ORM 框架，例如：

- Batoo JPA
- DataNucleus (formerly JPOX)
- EclipseLink (formerly Oracle TopLink)
- IBM, for WebSphere Application Server
- JBoss with Hibernate
- Kundera
- ObjectDB
- OpenJPA
- OrientDB from Orient Technologies
- Versant Corporation JPA (not relational, object database)

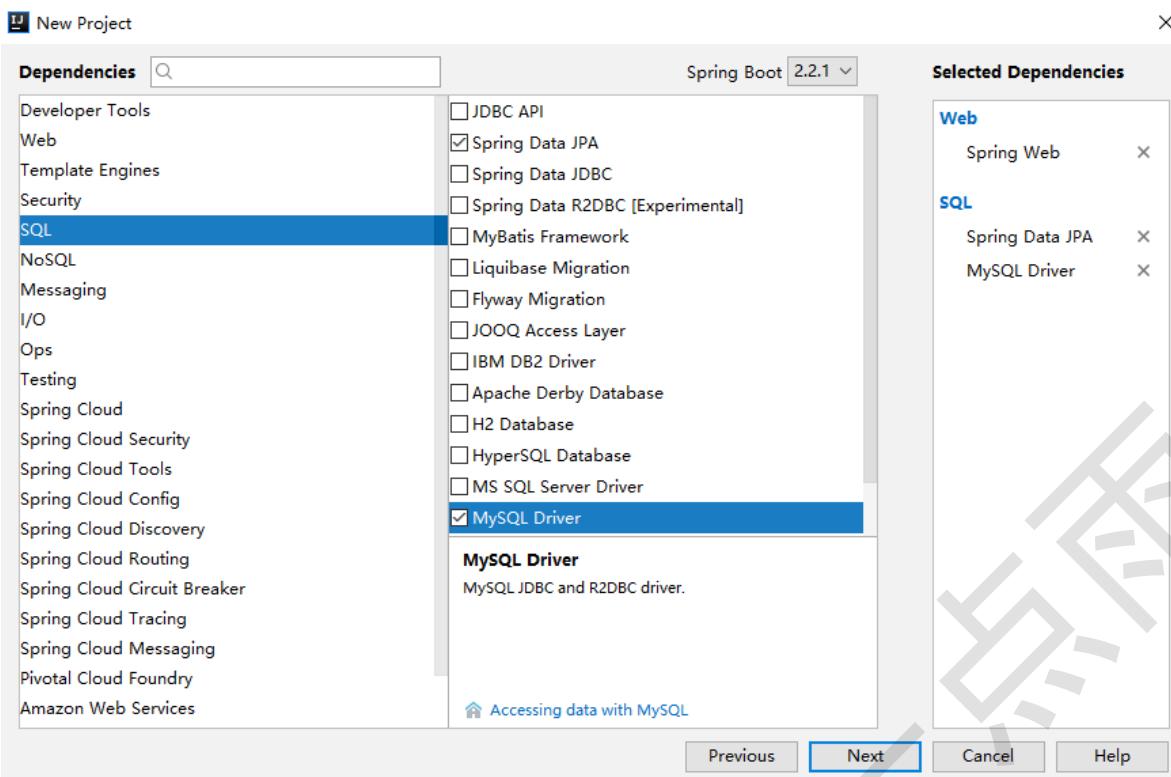
Hibernate 只是 ORM 框架的一种，上面列出来的 ORM 框架都是支持 JPA2.0 规范的 ORM 框架。既然它是一个规范，不是具体的实现，那么必然就不能直接使用（类似于 JDBC 不能直接使用，必须要加了驱动才能用），我们使用的是具体的实现，在这里我们采用的实现实际上还是 Hibernate。

Spring Boot 中使用的 Jpa 实际上是 Spring Data Jpa，Spring Data 是 Spring 家族的一个子项目，用于简化 SQL、NoSQL 的访问，在 Spring Data 中，只要你的方法名称符合规范，它就知道你想干嘛，不需要自己再去写 SQL。

关于 Spring Data Jpa 的具体情况，大家可以参考[一文读懂 Spring Data Jpa](#)

工程创建

创建 Spring Boot 工程，添加 Web、Jpa 以及 MySQL 驱动依赖，如下：



工程创建好之后，添加 Druid 依赖，完整的依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.28</version>
    <scope>runtime</scope>
</dependency>
```

如此，工程就算创建成功了。

基本配置

工程创建完成后，只需要在 application.properties 中进行数据库基本信息配置以及 Jpa 基本配置，如下：

```
# 数据库的基本配置
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.url=jdbc:mysql://test01?
useUnicode=true&characterEncoding=UTF-8
```

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource

# JPA配置
spring.jpa.database=mysql
# 在控制台打印SQL
spring.jpa.show-sql=true
# 数据库平台
spring.jpa.database-platform=mysql
# 每次启动项目时，数据库初始化策略
spring.jpa.hibernate.ddl-auto=update
# 指定默认的存储引擎为InnoDB
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
```

注意这里和 JdbcTemplate 以及 MyBatis 比起来，多了 Jpa 配置，Jpa 配置含义我都注释在代码中了，这里不再赘述，需要强调的是，最后一行配置，默认情况下，自动创建表的时候会使用 MyISAM 做表的引擎，如果配置了数据库方言为 MySQL57Dialect，则使用 InnoDB 做表的引擎。

好了，配置完成后，我们的 Jpa 差不多就可以开始用了。

基本用法

ORM(Object Relational Mapping) 框架表示对象关系映射，使用 ORM 框架我们不必再去创建表，框架会自动根据当前项目中的实体类创建相应的数据表。因此，我这里首先创建一个 User 对象，如下：

```
@Entity(name = "t_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Column(name = "name")
    private String username;
    private String address;
    //省略getter/setter
}
```

首先 @Entity 注解表示这是一个实体类，那么在项目启动时会自动针对该类生成一张表，默认的表名为类名，@Entity 注解的 name 属性表示自定义生成的表名。@Id 注解表示这个字段是一个 id，@GeneratedValue 注解表示主键的自增长策略，对于类中的其他属性，默认都会根据属性名在表中生成相应的字段，字段名和属性名相同，如果开发者想要对字段进行定制，可以使用 @Column 注解，去配置字段的名称，长度，是否为空等等。

做完这一切之后，启动 Spring Boot 项目，就会发现数据库中多了一个名为 t_user 的表了。

针对该表的操作，则需要我们提供一个 Repository，如下：

```
public interface UserDao extends JpaRepository<User, Integer> {
    List<User> getUserByAddressEqualsAndIdLessThanEqual(String address, Integer id);
    @Query(value = "select * from t_user where id=(select max(id) from t_user)", nativeQuery = true)
    User maxIdUser();
}
```

这里，自定义 UserDao 接口继承自 JpaRepository， JpaRepository 提供了一些基本的数据操作方法，例如保存，更新，删除，分页查询等，开发者也可以在接口中自己声明相关的方法，只需要方法名称符合规范即可，在 Spring Data 中，只要按照既定的规范命名方法，Spring Data Jpa 就知道你想干嘛，这样就不用写 SQL 了，那么规范是什么呢？参考下图：

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanOrEqualTo	findByAgeLessThanOrEqualTo	... where x.age ≤ ?1
Greater Than	findByAgeGreater Than	... where x.age > ?1
GreaterThanOrEqualTo	findByAgeGreaterThanOrEqualTo	... where x.age ≥ ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIs Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null

当然，这种方法命名主要是针对查询，但是一些特殊需求，可能并不能通过这种方式解决，例如想要查询 id 最大的用户，这时就需要开发者自定义查询 SQL 了。

如上代码所示，自定义查询 SQL，使用 @Query 注解，在注解中写自己的 SQL，默认使用的查询语言不是 SQL，而是 JPQL，这是一种数据库平台无关的面向对象的查询语言，有点定位类似于 Hibernate 中的 HQL，在 @Query 注解中设置 nativeQuery 属性为 true 则表示使用原生查询，即大伙所熟悉的 SQL。上面代码中的只是一个很简单的例子，还有其他一些点，例如如果这个方法中的 SQL 涉及到数据操作，则需要使用 @Modifying 注解。

好了，定义完 Dao 之后，接下来就可以将 UserDao 注入到 Controller 中进行测试了(这里为了省事，就没有提供 Service 了，直接将 UserDao 注入到 Controller 中)。

```
@RestController
public class UserController {
    @Autowired
    UserDao userDao;
    @PostMapping("/")
    public void adduser() {
        User user = new User();
        user.setId(1);
        user.setUsername("张三");
        user.setAddress("深圳");
        userDao.save(user);
    }
    @DeleteMapping("/")
    public void deleteById() {
        userDao.deleteById(1);
    }
    @PutMapping("/")
    public void updateUser() {
        User user = userDao.getOne(1);
        user.setUsername("李四");
    }
}
```

```
    userDao.flush();
}
@GetMapping("/test1")
public void test1() {
    List<User> all = userDao.findAll();
    System.out.println(all);
}
@GetMapping("/test2")
public void test2() {
    List<User> list = userDao.getUserByAddressEqualsAndIdLessThanOrEqualTo("广
州", 2);
    System.out.println(list);
}
@GetMapping("/test3")
public void test3() {
    User user = userDao.maxIdUser();
    System.out.println(user);
}
}
```

如此之后，即可查询到需要的数据。

好了，本文的重点是 Spring Boot 和 Jpa 的整合，这个话题就先说到这里。

多说两句

在和 Spring 框架整合时，如果用到 ORM 框架，大部分人可能都是首选 Hibernate，实际上，在和 Spring+SpringMVC 整合时，也可以选择 Spring Data Jpa 做数据持久化方案，用法和本文所述基本是一样的，Spring Boot 只是将 Spring Data Jpa 的配置简化了，因此，很多初学者对 Spring Data Jpa 觉得很神奇，但是又觉得无从下手，其实，此时可以回到 Spring 框架，先去学习 Jpa，再去学习 Spring Data Jpa，这是给初学者的一点建议。

相关案例已经上传到 GitHub，欢迎小伙伴们们下载：<https://github.com/lenve/javaboy-code-samples>

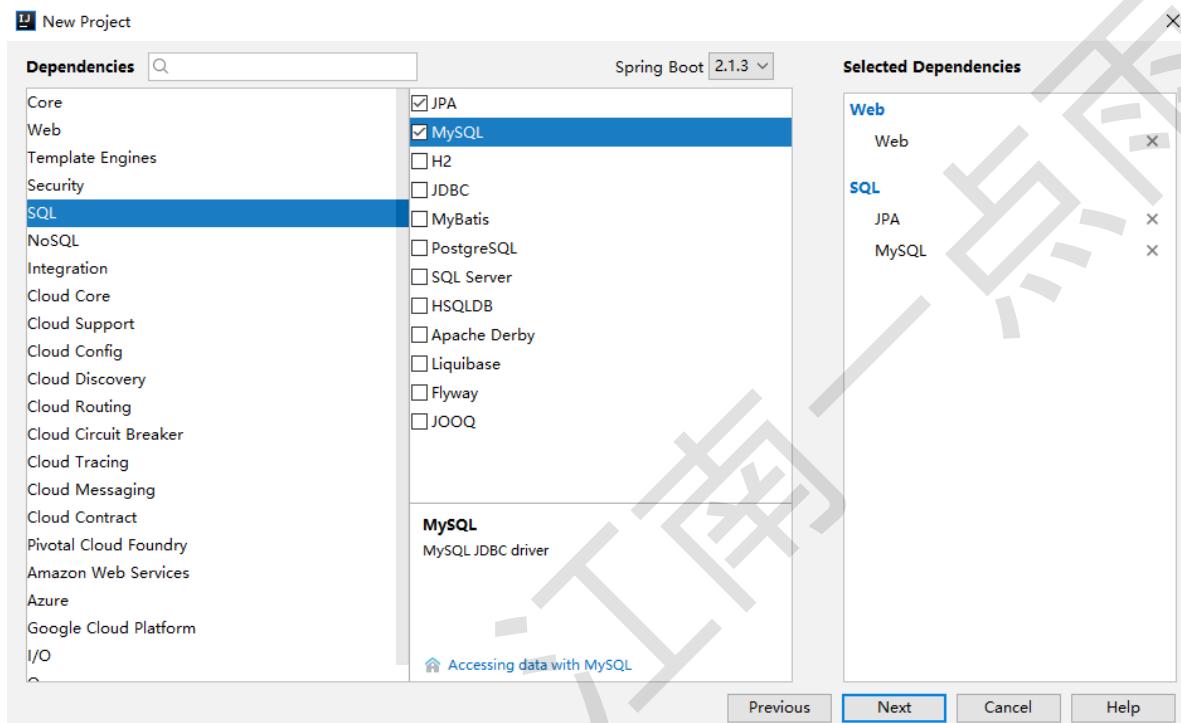
关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



本文是 Spring Boot 整合数据持久化方案的最后一篇，主要和大伙来聊聊 Spring Boot 整合 Jpa 多数据源问题。在 Spring Boot 整合JdbcTemplate 多数据源、Spring Boot 整合 MyBatis 多数据源以及 Spring Boot 整合 Jpa 多数据源这三个知识点中，整合 Jpa 多数据源算是最复杂的一种，也是很多人在配置时最容易出错的一种。本文大伙就跟着松哥的教程，一步一步整合 Jpa 多数据源。

工程创建

首先是创建一个 Spring Boot 工程，创建时添加基本的 Web、Jpa 以及 MySQL 依赖，如下：



创建完成后，添加 Druid 依赖，这里和前文的要求一样，要使用专为 Spring Boot 打造的 Druid，大伙可能发现了，如果整合多数据源一定要使用这个依赖，因为这个依赖中才有 DruidDataSourceBuilder，最后还要记得锁定数据库依赖的版本，因为可能大部分人用的还是 5.x 的 MySQL 而不是 8.x。完整依赖如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.28</version>
    <scope>runtime</scope>
</dependency>
```

如此之后，工程就创建成功了。

基本配置

在基本配置中，我们首先来配置多数据源基本信息以及 DataSource，首先在 application.properties 中添加如下配置信息：

```
# 数据源一
spring.datasource.one.username=root
spring.datasource.one.password=root
spring.datasource.one.url=jdbc:mysql://test01?
useUnicode=true&characterEncoding=UTF-8
spring.datasource.one.type=com.alibaba.druid.pool.DruidDataSource

# 数据源二
spring.datasource.two.username=root
spring.datasource.two.password=root
spring.datasource.two.url=jdbc:mysql://test02?
useUnicode=true&characterEncoding=UTF-8
spring.datasource.two.type=com.alibaba.druid.pool.DruidDataSource

# Jpa配置
spring.jpa.properties.database=mysql
spring.jpa.properties.show-sql=true
spring.jpa.properties.database-platform=mysql
spring.jpa.properties.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
```

这里 Jpa 的配置和上文相比 key 中多了 properties，多数据源的配置和前文一致，然后接下来配置两个 DataSource，如下：

```
@Configuration
public class DataSourceConfig {
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.one")
    @Primary
    DataSource dsOne() {
        return DruidDataSourceBuilder.create().build();
    }
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource.two")
    DataSource dsTwo() {
        return DruidDataSourceBuilder.create().build();
    }
}
```

这里的配置和前文的多数据源配置基本一致，但是注意多了一个在 Spring 中使用较少的注解 @Primary，这个注解一定不能少，否则在项目启动时会出错，@Primary 表示当某一个类存在多个实例时，优先使用哪个实例。

好了，这样，DataSource 就有了。

多数据源配置

接下来配置 Jpa 的基本信息，这里两个数据源，我分别在两个类中来配置，先来看第一个配置：

```

@Configuration
@EnableJpaRepositories(basePackages =
"org.javaboy.jpa.dao",entityManagerFactoryRef =
"localContainerEntityManagerFactoryBeanOne",transactionManagerRef =
"platformTransactionManagerOne")
public class JpaConfigOne {
    @Autowired
    @Qualifier(value = "dsOne")
    DataSource dsOne;
    @Autowired
    JpaProperties jpaProperties;
    @Bean
    @Primary
    LocalContainerEntityManagerFactoryBean
localContainerEntityManagerFactoryBeanOne(EntityManagerFactoryBuilder builder) {
        return builder.dataSource(dsOne)
            .packages("org.javaboy.jpa.model")
            .properties(jpaProperties.getProperties())
            .persistenceUnit("pu1")
            .build();
    }
    @Bean
    PlatformTransactionManager
platformTransactionManagerOne(EntityManagerFactoryBuilder builder) {
        LocalContainerEntityManagerFactoryBean factoryBeanOne =
localContainerEntityManagerFactoryBeanOne(builder);
        return new JpaTransactionManager(factoryBeanOne.getObject());
    }
}

```

首先这里注入 dsOne，再注入 JpaProperties，JpaProperties 是系统提供的一个实例，里边的数据就是我们在 application.properties 中配置的 jpa 相关的配置。然后我们提供两个 Bean，分别是 LocalContainerEntityManagerFactoryBean 和 PlatformTransactionManager 事务管理器，不同于 MyBatis 和 JdbcTemplate，在 Jpa 中，事务一定要配置。在提供 LocalContainerEntityManagerFactoryBean 的时候，需要指定 packages，这里的 packages 指定的包就是这个数据源对应的实体类所在的位置，另外在这里配置类上通过 @EnableJpaRepositories 注解指定 dao 所在的位置，以及 LocalContainerEntityManagerFactoryBean 和 PlatformTransactionManager 分别对应的引用的名字。

好了，这样第一个就配置好了，第二个基本和这个类似，主要有几个不同点：

- dao 的位置不同
- persistenceUnit 不同
- 相关 bean 的名称不同

注意实体类可以共用。

代码如下：

```

@Configuration
@EnableJpaRepositories(basePackages =
"org.javaboy.jpa.dao2",entityManagerFactoryRef =
"localContainerEntityManagerFactoryBeanTwo",transactionManagerRef =
"platformTransactionManagerTwo")
public class JpaConfigTwo {
    @Autowired
    @Qualifier(value = "dsTwo")

```

```

DataSource dsTwo;
@Autowired
JpaProperties jpaProperties;
@Bean
LocalContainerEntityManagerFactoryBean
localContainerEntityManagerFactoryBeanTwo(EntityManagerFactoryBuilder builder) {
    return builder.dataSource(dsTwo)
        .packages("org.javaboy.jpa.model")
        .properties(jpaProperties.getProperties())
        .persistenceUnit("pu2")
        .build();
}
@Bean
PlatformTransactionManager
platformTransactionManagerTwo(EntityManagerFactoryBuilder builder) {
    LocalContainerEntityManagerFactoryBean factoryBeanTwo =
localContainerEntityManagerFactoryBeanTwo(builder);
    return new JpaTransactionManager(factoryBeanTwo.getObject());
}
}

```

接下来，在对应位置分别提供相关的实体类和 dao 即可，数据源一的 dao 如下：

```

package org.javaboy.jpa.dao;
public interface UserDao extends JpaRepository<User, Integer> {
    List<User> getUserByAddressEqualsAndIdLessThanEqual(String address, Integer id);
    @Query(value = "select * from t_user where id=(select max(id) from
t_user)",nativeQuery = true)
    User maxIdUser();
}

```

数据源二的 dao 如下：

```

package org.javaboy.jpa.dao2;
public interface UserDao2 extends JpaRepository<User, Integer> {
    List<User> getUserByAddressEqualsAndIdLessThanEqual(String address, Integer id);

    @Query(value = "select * from t_user where id=(select max(id) from
t_user)",nativeQuery = true)
    User maxIdUser();
}

```

共同的实体类如下：

```
package org.javaboy.jpa.model;
@Entity(name = "t_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String username;
    private String address;
    //省略getter/setter
}
```

到此，所有的配置就算完成了，接下来就可以在 Service 中注入不同的 UserDao，不同的 UserDao 操作不同的数据源。

其实整合 Jpa 多数据源也不算难，就是有几个细节问题，这些细节问题解决，其实前面介绍的其他多数据源整个都差不多。

好了，本文就先介绍到这里。

相关案例已经上传到 GitHub，欢迎小伙伴们们下载：<https://github.com/lenve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



在 Redis 出现之前，我们的缓存框架各种各样，有了 Redis，缓存方案基本上都统一了，关于 Redis，松哥之前有一个系列教程，尚不了解 Redis 的小伙伴可以参考这个教程：

- [Redis 教程合集](#)

使用 Java 操作 Redis 的方案很多，Jedis 是目前较为流行的一种方案，除了 Jedis，还有很多其他解决方案，如下：

Java

Redisson	⭐	⭐	⭐	distributed and scalable Java data structures on top of Redis server	
aredis			⭐	Asynchronous, pipelined client based on the Java 7 NIO Channel API	
JDBC-Redis		⭐	⭐		
Jedis	⭐	⭐	⭐		
JRedis	⭐	⭐	⭐		
lettuce	⭐		⭐	Thread-safe client supporting async usage and key/value codecs	 
mod-redis			⭐	Official asynchronous redis.io bus module for Vert.x	
redis-protocol	⭐		⭐	Up to 2.6 compatible high-performance Java, Java w/Netty & Scala (finagle) client	
RedisClient	⭐		⭐	redis client GUI tool	
RJC			⭐		

除了这些方案之外，还有一个使用也相当多的方案，就是 Spring Data Redis。

在传统的 SSM 中，需要开发者自己来配置 Spring Data Redis，这个配置比较繁琐，主要配置 3 个东西：连接池、连接器信息以及 key 和 value 的序列化方案。

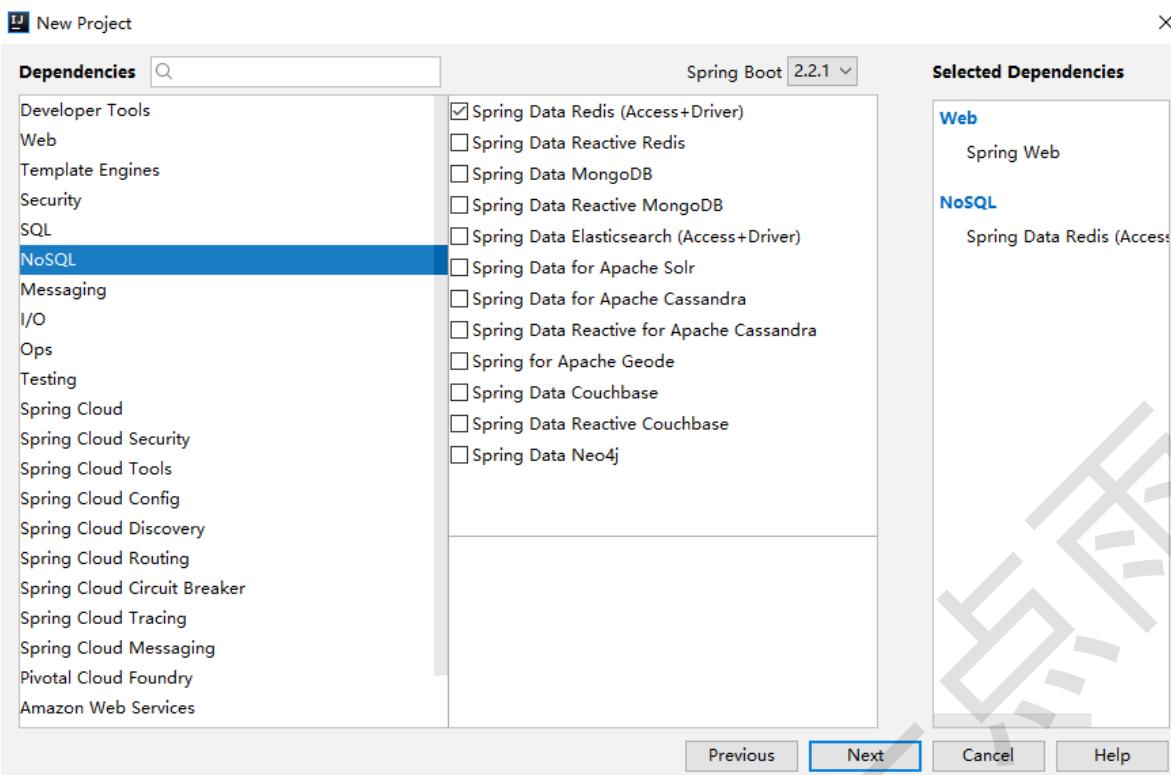
在 Spring Boot 中，默认集成的 Redis 就是 Spring Data Redis，默认底层的连接池使用了 lettuce，开发者可以自行修改为自己的熟悉的，例如 Jedis。

Spring Data Redis 针对 Redis 提供了非常方便的操作模板 RedisTemplate。这是 Spring Data 擅长的事情，那么接下来我们就来看看 Spring Boot 中 Spring Data Redis 的具体用法。

方案一：Spring Data Redis

创建工程

创建工程，引入 Redis 依赖：



创建成功后，还需要手动引入 commons-pool2 的依赖，因此最终完整的 pom.xml 依赖如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-pool2</artifactId>
    </dependency>
</dependencies>
```

这里主要就是引入了 Spring Data Redis + 连接池。

配置 Redis 信息

接下来配置 Redis 的信息，信息包含两方面，一方面是 Redis 的基本信息，另一方面则是连接池信息：

```
spring.redis.database=0
spring.redis.password=123
spring.redis.port=6379
spring.redis.host=192.168.66.128
spring.redis.lettuce.pool.min-idle=5
spring.redis.lettuce.pool.max-idle=10
spring.redis.lettuce.pool.max-active=8
spring.redis.lettuce.pool.max-wait=1ms
spring.redis.lettuce.shutdown-timeout=100ms
```

自动配置

当开发者在项目中引入了 Spring Data Redis，并且配置了 Redis 的基本信息，此时，自动化配置就会生效。

我们从 Spring Boot 中 Redis 的自动化配置类中就可以看出端倪：

```
@Configuration
@ConditionalOnClass(RedisOperations.class)
@EnableConfigurationProperties(RedisProperties.class)
@Import({ LettuceConnectionConfiguration.class,
JedisConnectionConfiguration.class })
public class RedisAutoConfiguration {

    @Bean
    @ConditionalOnMissingBean(name = "redisTemplate")
    public RedisTemplate<Object, Object> redisTemplate(
            RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
        RedisTemplate<Object, Object> template = new RedisTemplate<>();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }

    @Bean
    @ConditionalOnMissingBean
    public StringRedisTemplate stringRedisTemplate(
            RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
        StringRedisTemplate template = new StringRedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
}
```

这个自动化配置类很好理解：

1. 首先标记这个是一个配置类，同时该配置在 RedisOperations 存在的情况下才会生效(即项目中引入了 Spring Data Redis)
2. 然后导入在 application.properties 中配置的属性
3. 然后再导入连接池信息 (如果存在的话)
4. 最后，提供了两个 Bean，RedisTemplate 和 StringRedisTemplate，其中 StringRedisTemplate 是 RedisTemplate 的子类，两个的方法基本一致，不同之处主要体现在操作的数据类型不同，RedisTemplate 中的两个泛型都是 Object，意味着存储的 key 和 value 都可以是一个对象，而 StringRedisTemplate 的两个泛型都是 String，意味着 StringRedisTemplate 的 key 和 value 都只能是字符串。如果开发者没有提供相关的 Bean，这两个配置就会生效，否则不会生效。

使用

接下来，可以直接在 Service 中注入 StringRedisTemplate 或者 RedisTemplate 来使用：

```
@Service
public class HelloService {
    @Autowired
    RedisTemplate redisTemplate;
    public void hello() {
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set("k1", "v1");
        Object k1 = ops.get("k1");
        System.out.println(k1);
    }
}
```

Redis 中的数据操作，大体上来说，可以分为两种：

1. 针对 key 的操作，相关的方法就在 RedisTemplate 中
2. 针对具体数据类型的操作，相关的方法需要首先获取对应的数据类型，获取相应数据类型的操作方法是 opsForXXX

调用该方法就可以将数据存储到 Redis 中去了，如下：

```
127.0.0.1:6379> keys *
1) "\xac\xed\x00\x05t\x00\x02k1"
127.0.0.1:6379>
```

k1 前面的字符是由于使用了 RedisTemplate 导致的，RedisTemplate 对 key 进行序列化之后的结果。

RedisTemplate 中，key 默认的序列化方案是 JdkSerializationRedisSerializer。

而在 StringRedisTemplate 中，key 默认的序列化方案是 StringRedisSerializer，因此，如果使用 StringRedisTemplate，默认情况下 key 前面不会有前缀。

不过开发者也可以自行修改 RedisTemplate 中的序列化方案，如下：

```
@Service
public class HelloService {
    @Autowired
    RedisTemplate redisTemplate;
    public void hello() {
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        ValueOperations ops = redisTemplate.opsForValue();
        ops.set("k1", "v1");
        Object k1 = ops.get("k1");
        System.out.println(k1);
    }
}
```

当然也可以直接使用 StringRedisTemplate：

```
@Service
public class HelloService {
    @Autowired
    StringRedisTemplate stringRedisTemplate;
    public void hello2() {
        ValueOperations<Object, Object> ops = stringRedisTemplate.opsForValue();
        ops.set("k2", "v2");
        Object k1 = ops.get("k2");
        System.out.println(k1);
    }
}
```

另外需要注意，Spring Boot 的自动化配置，只能配置单机的 Redis，如果是 Redis 集群，则所有的东西都需要自己手动配置，关于如何操作 Redis 集群，松哥以后再来和大家分享。

方案二：Spring Cache

通过 Spring Cache 的形式来操作 Redis，Spring Cache 统一了缓存江湖的门面，这种方案，松哥之前有过一篇专门的文章介绍，小伙伴可以移步这里：[Spring Boot 中，Redis 缓存还能这么用！](#)。

方案三：回归原始时代

第三种方案，就是直接使用 Jedis 或者其他的客户端工具来操作 Redis，这种方案在 Spring Boot 中也是支持的，虽然操作麻烦，但是支持，这种操作松哥之前也有介绍的文章，因此这里就不再赘述了，可以参考[Jedis 使用](#)。

总结

Spring Boot 中，Redis 的操作，这里松哥给大家总结了三种方案，实际上前两个使用广泛一些，直接使用 Jedis 还是比较少，基本上 Spring Boot 中没见过有人直接这么搞。

好了，本文就说到这里，有问题欢迎留言讨论。

相关案例已经上传到 GitHub，欢迎小伙伴们们下载：<https://github.com/lenvve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

上篇文章和大家聊了 Spring Session 实现 Session 共享的问题，有的小伙伴看了后表示对 Nginx 还是很懵，因此有了这篇文章，算是一个 Nginx 扫盲入门吧！

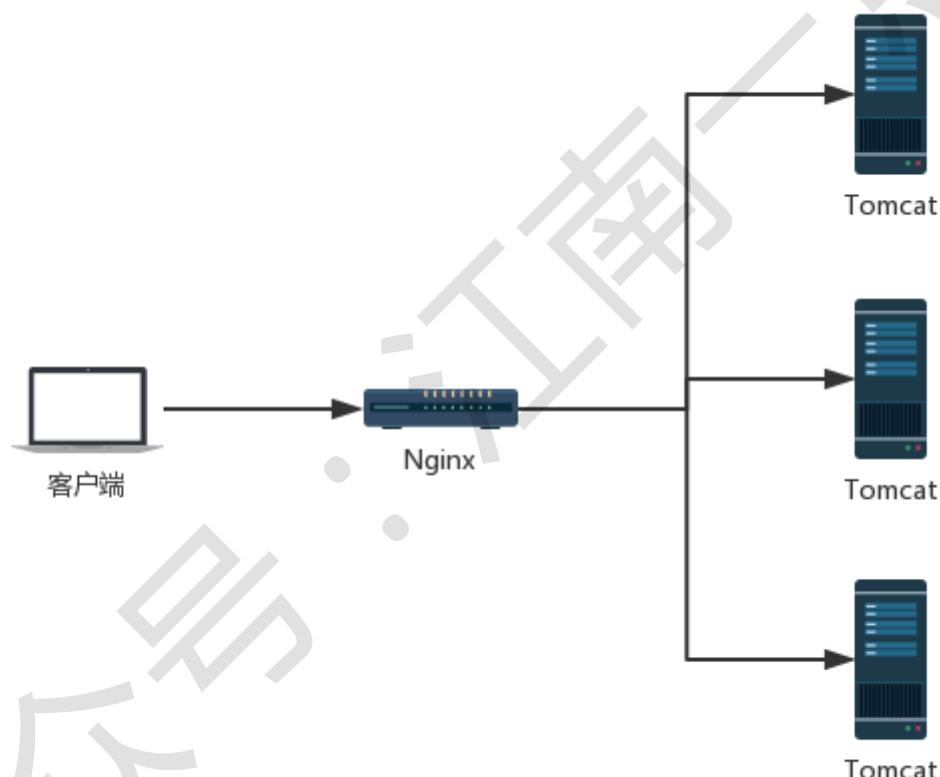
基本介绍

Nginx 是一个高性能的 HTTP 和反向代理 web 服务器，同时也提供了 IMAP/POP3/SMTP 服务。

Nginx 是由伊戈尔·赛索耶夫为俄罗斯访问量第二的 Rambler.ru 站点开发的，第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日。

Nginx 特点是占有内存少，并发能力强。

事实上 nginx 的并发能力确实在同类型的网页服务器中表现较好，一般来说，如果我们在项目中引入了 Nginx，我们的项目架构可能是这样：



在这样的架构中，Nginx 所代表的角色叫做负载均衡服务器或者反向代理服务器，所有请求首先到达 Nginx 上，再由 Nginx 根据提前配置好的转发规则，将客户端发来的请求转发到某一个 Tomcat 上去。

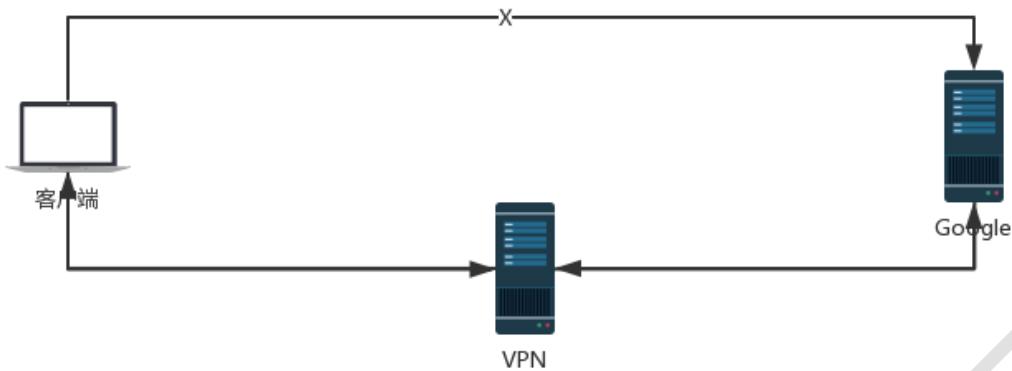
那么这里涉及到两个概念：

- 负载均衡服务器

就是进行请求转发，降低某一个服务器的压力。负载均衡策略很多，也有很多层，对于一些大型网站基本上从 DNS 就开始负载均衡，负载均衡有硬件和软件之分，各自代表分别是 F5 和 Nginx（目前 Nginx 已经被 F5 收购），早些年，也可以使用 Apache 来做负载均衡，但是效率不如 Nginx，所以现在主流方案是 Nginx。

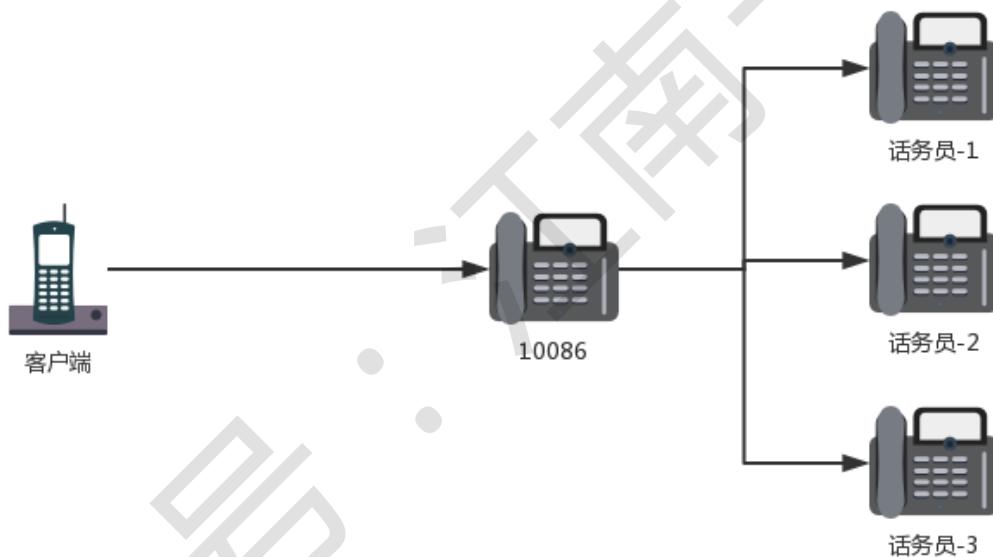
- 反向代理服务器：

另一个概念是反向代理服务器，得先说正向代理，看下面一张图：



在这个过程中，Google 并不知道真正访问它的客户端是谁，它只知道这个中间服务器在访问它。因此，这里的代理，实际上是中间服务器代理了客户端，这种代理叫做正向代理。

那么什么是反向代理呢？看下面一张图：



在这个过程中，10086 这个号码相当于是一个代理，真正提供服务的，是话务员，但是对于客户来说，他不关心到底是哪一个话务员提供的服务，他只需要记得 10086 这个号码就行了。

所有的请求打到 10086 上，再由 10086 将请求转发给某一个话务员去处理。因此，在这里，10086 就相当于是一个代理，只不过它代理的是话务员而不是客户端，这种代理称之为反向代理。

Nginx 的优势

在 Java 开发中，Nginx 有着非常广泛的使用，随便举几点：

1. 使用 Nginx 做静态资源服务器：Java 中的资源可以分为动态和静态，动态需要经过 Tomcat 解析之后，才能返回给浏览器，例如 JSP 页面、Freemarker 页面、控制器返回的 JSON 数据等，都算作动态资源，动态资源经过了 Tomcat 处理，速度必然降低。对于静态资源，例如图片、HTML、JS、CSS 等资源，这种资源可以不必经过 Tomcat 解析，当客户端请求这些资源时，直接将资源返回给客户端就行了。此时，可以使用 Nginx 搭建静态资源服务器，将静态资源直接返回给客户端。

2. 使用 Nginx 做负载均衡服务器，无论是使用 Dubbo 还是 Spring Cloud，除了使用各自自带的负载均衡策略之外，也都可以使用 Nginx 做负载均衡服务器。
3. 支持高并发、内存消耗少、成本低廉、配置简单、运行稳定等。

Nginx 安装：

由于基本上都是在 Linux 上使用 Nginx，因此松哥这里主要向大家展示 CentOS 7 安装 Nginx：

1. 首先下载 Nginx

```
wget http://nginx.org/download/nginx-1.17.0.tar.gz
```

然后解压下载的目录，进入解压目录中，在编译安装之前，需要安装两个依赖：

```
yum -y install pcre-devel  
yum -y install openssl openssl-devel
```

然后开始编译安装：

```
./configure  
make  
make install
```

装好之后，默认安装位置在：

```
/usr/local/nginx/sbin/nginx
```

进入到该目录的 `sbin` 目录下，执行 `nginx` 即可启动 Nginx：

```
[root@localhost sbin]# pwd  
/usr/local/nginx/sbin  
[root@localhost sbin]# ./nginx  
[root@localhost sbin]# █
```

Nginx 启动成功之后，在浏览器中直接访问 Nginx 地址：

192.168.66.128



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

看到如上页面，表示 Nginx 已经安装成功了。

如果修改了 Nginx 配置，则可以通过如下命令重新加载 Nginx 配置文件：

```
./nginx -s reload
```

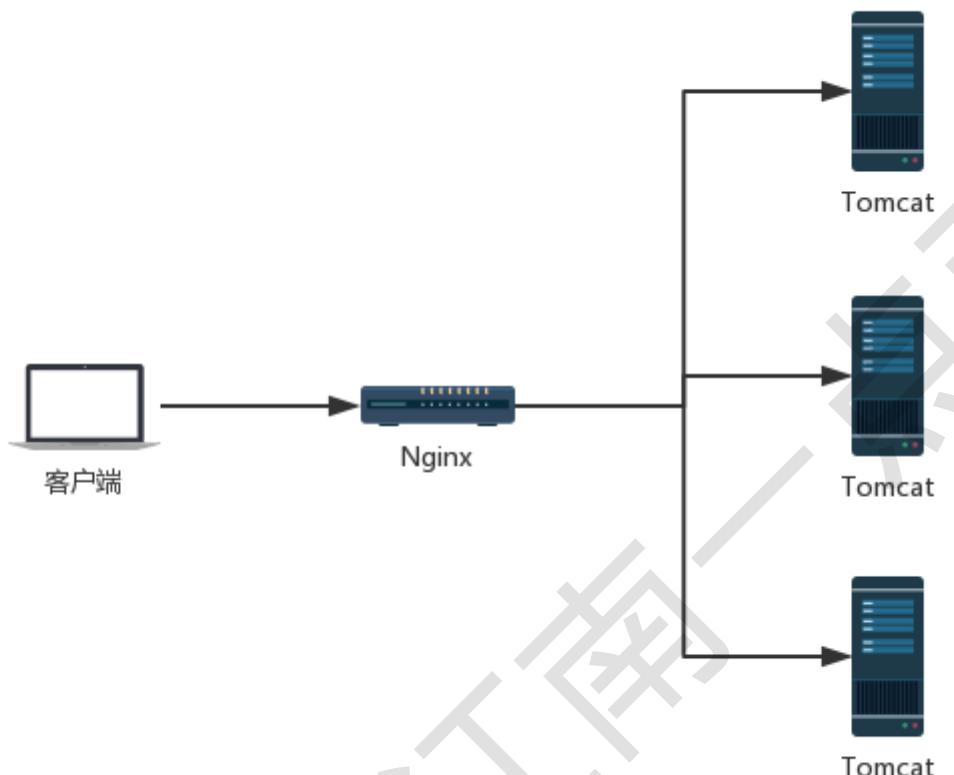
总结

本文算是一个简单的 Nginx 扫盲文，希望大家看完后对 Nginx 有一个基本的认知。本文先说到这里，有问题欢迎留言讨论。

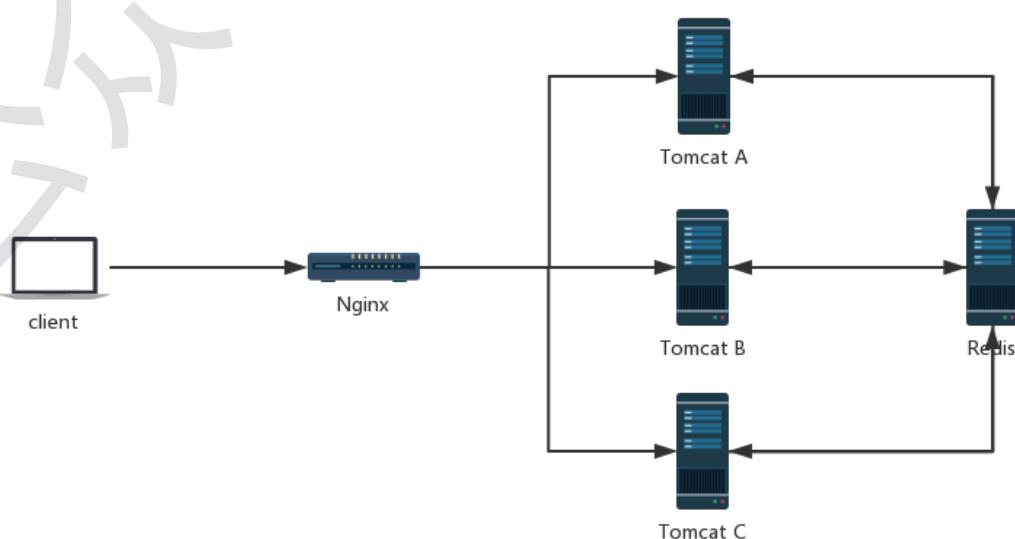
关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



在传统的单服务架构中，一般来说，只有一个服务器，那么不存在 Session 共享问题，但是在分布式/集群项目中，Session 共享则是一个必须面对的问题，先看一个简单的架构图：



在这样的架构中，会出现一些单服务中不存在的问题，例如客户端发起一个请求，这个请求到达 Nginx 上之后，被 Nginx 转发到 Tomcat A 上，然后在 Tomcat A 上往 session 中保存了一份数据，下次又来一个请求，这个请求被转发到 Tomcat B 上，此时再去 Session 中获取数据，发现没有之前的数据。对于这一类问题的解决，思路很简单，就是将各个服务之间需要共享的数据，保存到一个公共的地方（主流方案就是 Redis）：



当所有 Tomcat 需要往 Session 中写数据时，都往 Redis 中写，当所有 Tomcat 需要读数据时，都从 Redis 中读。这样，不同的服务就可以使用相同的 Session 数据了。

这样的方案，可以由开发者手动实现，即手动往 Redis 中存储数据，手动从 Redis 中读取数据，相当于使用一些 Redis 客户端工具来实现这样的功能，毫无疑问，手动实现工作量还是蛮大的。

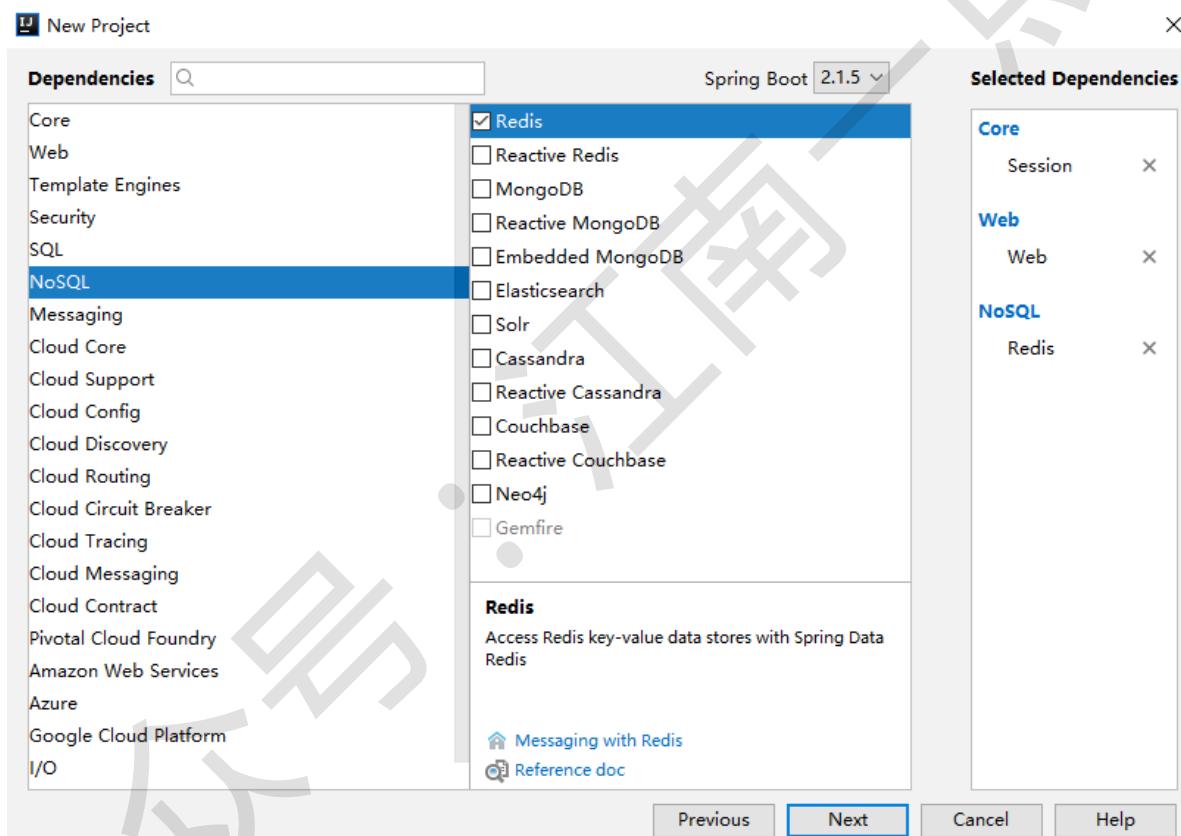
一个简化的方案就是使用 Spring Session 来实现这一功能，Spring Session 就是使用 Spring 中的代理过滤器，将所有的 Session 操作拦截下来，自动的将数据同步到 Redis 中，或者自动的从 Redis 中读取数据。

对于开发者来说，所有关于 Session 同步的操作都是透明的，开发者使用 Spring Session，一旦配置完成后，具体的用法就像使用一个普通的 Session 一样。

1 实战

1.1 创建工程

首先创建一个 Spring Boot 工程，引入 Web、Spring Session 以及 Redis：



创建成功之后，pom.xml 文件如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.session</groupId>
        <artifactId>spring-session-data-redis</artifactId>
    </dependency>
```

```
</dependency>  
</dependencies>
```

注意：

这里我使用的 Spring Boot 版本是 2.1.4，如果使用当前最新版 Spring Boot 2.1.5 的话，除了上面这些依赖之外，需要额外添加 Spring Security 依赖（其他操作不受影响，仅仅只是多了一个依赖，当然也多了 Spring Security 的一些默认认证流程）。

1.2 配置 Redis

```
spring.redis.host=192.168.66.128  
spring.redis.port=6379  
spring.redis.password=123  
spring.redis.database=0
```

这里的 Redis，我虽然配置了四行，但是考虑到端口默认就是 6379，database 默认就是 0，所以真正要配置的，其实就是两行。

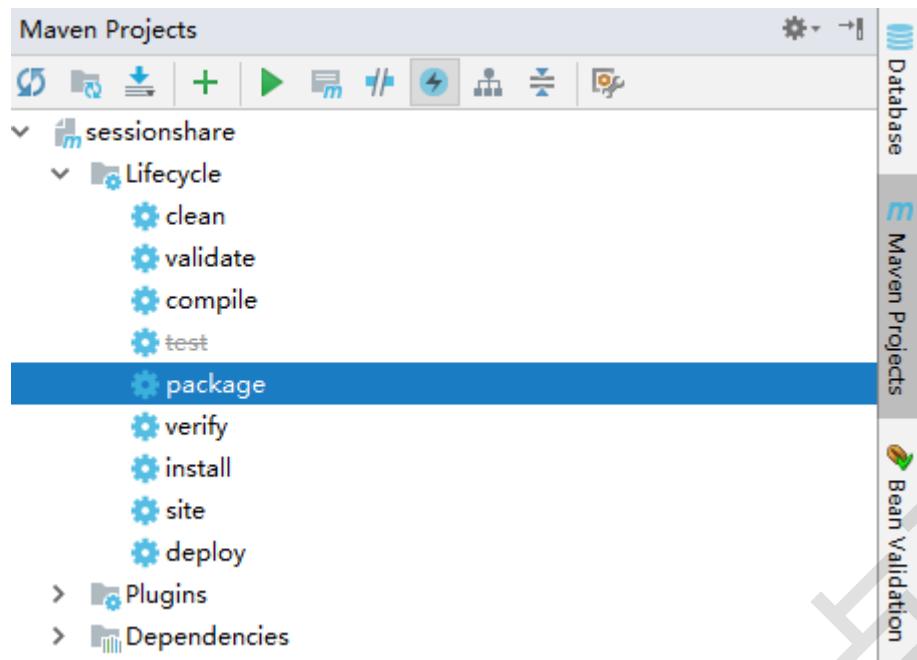
1.3 使用

配置完成后，就可以使用 Spring Session 了，其实就是使用普通的 HttpSession，其他的 Session 同步到 Redis 等操作，框架已经自动帮你完成了：

```
@RestController  
public class HelloController {  
    @Value("${server.port}")  
    Integer port;  
    @GetMapping("/set")  
    public String set(HttpSession session) {  
        session.setAttribute("user", "javaboy");  
        return String.valueOf(port);  
    }  
    @GetMapping("/get")  
    public String get(HttpSession session) {  
        return session.getAttribute("user") + ":" + port;  
    }  
}
```

考虑到一会 Spring Boot 将以集群的方式启动，为了获取每一个请求到底是哪一个 Spring Boot 提供的服务，需要在每次请求时返回当前服务的端口号，因此这里我注入了 server.port。

接下来，项目打包：



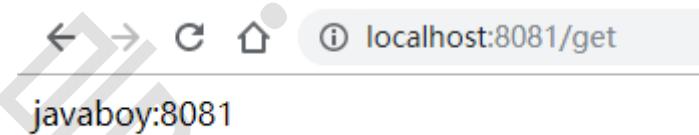
打包之后，启动项目的两个实例：

```
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080  
java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081
```

然后先访问 `localhost:8080/set` 向 `8080` 这个服务的 `session` 中保存一个变量，访问完成后，数据就已经自动同步到 `Redis` 中了：

```
127.0.0.1:6379> keys *
1) "spring:session:sessions:bla8fbad-8912-4d01-alcf-acf90765bab4"
2) "spring:session:sessions:expires:bla8fbad-8912-4d01-alcf-acf90765bab4"
3) "spring:session:expirations:1559551560000"
```

然后，再调用 `localhost:8081/get` 接口，就可以获取到 `8080` 服务的 `session` 中的数据：



此时关于 session 共享的配置就已经全部完成了，session 共享的效果我们已经看到了，但是每次访问都是我自己手动切换服务实例，因此，接下来我们来引入 Nginx，实现服务实例自动切换。

1.4 引入 Nginx

很简单，进入 Nginx 的安装目录的 conf 目录下（默认是在 `/usr/local/nginx/conf`），编辑 `nginx.conf` 文件：

```
upstream javaboy.org{
    server 127.0.0.1:8080 weight=1;
    server 127.0.0.1:8081 weight=2;
}
server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
        proxy_pass http://javaboy.org;
        proxy_redirect default;
        #root   html;
        #index  index.html index.htm;
    }
}
```

在这段配置中：

1. upstream 表示配置上游服务器
2. javaboy.org 表示服务器集群的名字，这个可以随意取名字
3. upstream 里边配置的是一个个的单独服务
4. weight 表示服务的权重，意味者将有多少比例的请求从 Nginx 上转发到该服务上
5. location 中的 proxy_pass 表示请求转发的地址，/ 表示拦截到所有的请求，转发转发到刚刚配置好的服务集群中
6. proxy_redirect 表示设置当发生重定向请求时，nginx 自动修正响应头数据（默认是 Tomcat 返回重定向，此时重定向的地址是 Tomcat 的地址，我们需要将之修改使之成为 Nginx 的地址）。

配置完成后，将本地的 Spring Boot 打包好的 jar 上传到 Linux，然后在 Linux 上分别启动两个 Spring Boot 实例：

```
nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8080 &
nohup java -jar sessionshare-0.0.1-SNAPSHOT.jar --server.port=8081 &
```

其中

- nohup 表示当终端关闭时，Spring Boot 不要停止运行
- & 表示让 Spring Boot 在后台启动

配置完成后，重启 Nginx：

```
/usr/local/nginx/sbin/nginx -s reload
```

Nginx 启动成功后，我们首先手动清除 Redis 上的数据，然后访问 192.168.66.128/set 表示向 session 中保存数据，这个请求首先会到达 Nginx 上，再由 Nginx 转发给某一个 Spring Boot 实例：



8081

如上，表示端口为 8081 的 Spring Boot 处理了这个 /set 请求，再访问 /get 请求：

javaboy:8080

可以看到，/get 请求是被端口为 8080 的服务所处理的。

2 总结

本文主要向大家介绍了 Spring Session 的使用，另外也涉及到一些 Nginx 的使用，虽然本文较长，但是实际上 Spring Session 的配置没啥。

我们写了一些代码，也做了一些配置，但是全都和 Spring Session 无关，配置是配置 Redis，代码就是普通的 HttpSession，和 Spring Session 没有任何关系！

唯一和 Spring Session 相关的，可能就是我在一开始引入了 Spring Session 的依赖吧！

如果大家没有在 SSM 架构中用过 Spring Session，可能不太好理解我们在 Spring Boot 中使用 Spring Session 有多么方便，因为在 SSM 架构中，Spring Session 的使用要配置三个地方，一个是 web.xml 配置代理过滤器，然后在 Spring 容器中配置 Redis，最后再配置 Spring Session，步骤还是有些繁琐的，而 Spring Boot 中直接帮我们省去了这些繁琐的步骤！不用再去配置 Spring Session。

好了，本文就说到这里，本文相关案例我已经上传到 GitHub，大家可以自行下载：<https://github.com/lenve/javaboy-code-samples>

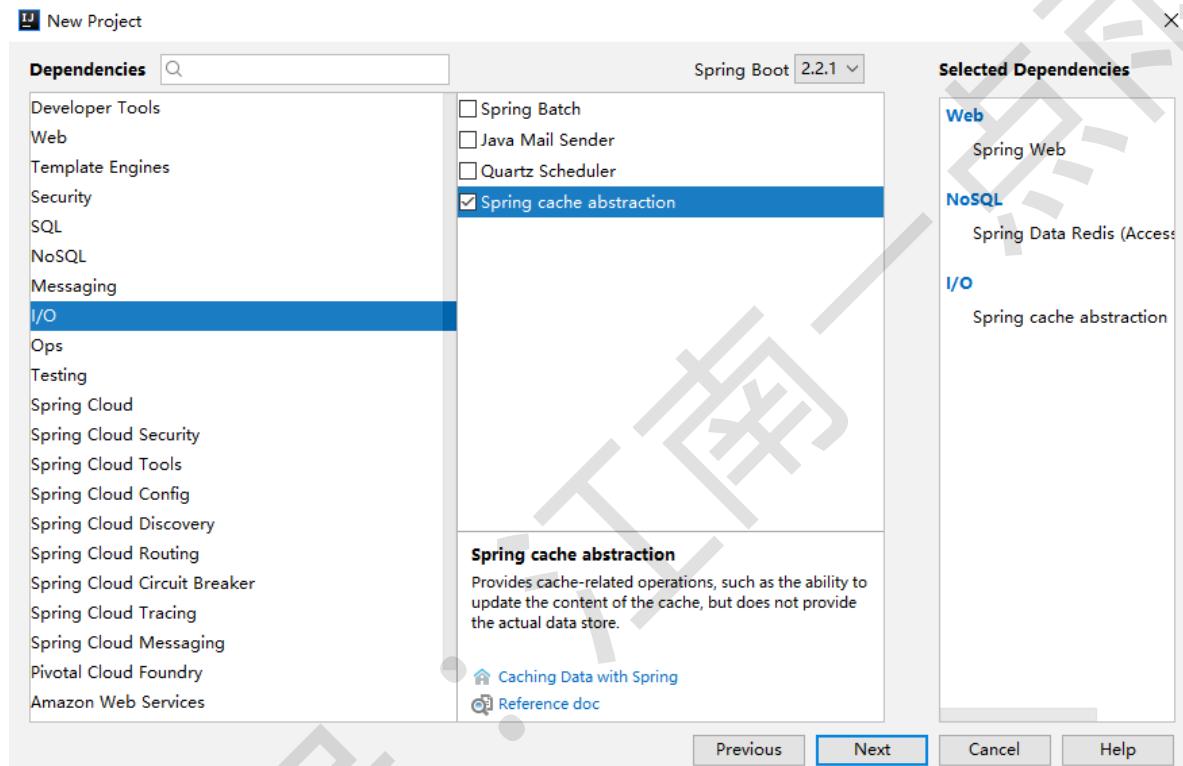
关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



经过 Spring Boot 的整合封装与自动化配置，在 Spring Boot 中整合 Redis 已经变得非常容易了，开发者只需要引入 Spring Data Redis 依赖，然后简单配下 redis 的基本信息，系统就会提供一个 RedisTemplate 供开发者使用，但是今天松哥想和大伙聊的不是这种用法，而是结合 Cache 的用法。Spring3.1 中开始引入了令人激动的 Cache，在 Spring Boot 中，可以非常方便的使用 Redis 来作为 Cache 的实现，进而实现数据的缓存。

工程创建

首先创建一个 Spring Boot 工程，注意创建的时候需要引入三个依赖， web、 cache 以及 redis，如下图：



对应的依赖内容如下：

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

基本配置

工程创建好之后，首先需要简单配置一下 Redis，Redis 的基本信息，另外，这里要用到 Cache，因此还需要稍微配置一下 Cache，如下：

```
spring.redis.port=6380  
spring.redis.host=192.168.66.128  
  
spring.cache.cache-names=c1
```

简单起见，这里我只是配置了 Redis 的端口和地址，然后给缓存取了一个名字，这个名字在后文会用到。

另外，还需要在配置类上添加如下代码，表示开启缓存：

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

完成了这些配置之后，Spring Boot 就会自动帮我们在后台配置一个 RedisCacheManager，相关的配置是在org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration 类中完成的。部分源码如下：

```
@Configuration  
@ConditionalOnClass(RedisConnectionFactory.class)  
@AutoConfigureAfter(RedisAutoConfiguration.class)  
@ConditionalOnBean(RedisConnectionFactory.class)  
@ConditionalOnMissingBean(CacheManager.class)  
@Conditional(CacheCondition.class)  
class RediscacheConfiguration {  
    @Bean  
    public RedisCacheManager cacheManager(RedisConnectionFactory  
redisConnectionFactory,  
                                         ResourceLoader resourceLoader) {  
        RedisCacheManagerBuilder builder = RedisCacheManager  
            .builder(redisConnectionFactory)  
  
        .cacheDefaults(determineConfiguration(resourceLoader.getClassLoader()));  
        List<String> cacheNames = this.cacheProperties.getCacheNames();  
        if (!cacheNames.isEmpty()) {  
            builder.initialCacheNames(new LinkedHashSet<>(cacheNames));  
        }  
        return this.customizerInvoker.customize(builder.build());  
    }  
}
```

看类上的注解，发现在万事俱备的情况下，系统会自动提供一个 RedisCacheManager 的 Bean，这个 RedisCacheManager 间接实现了 Spring 中的 Cache 接口，有了这个 Bean，我们就可以直接使用 Spring 中的缓存注解和接口了，而缓存数据则会被自动存储到 Redis 上。在单机的 Redis 中，这个 Bean 系统会自动提供，如果是 Redis 集群，这个 Bean 需要开发者来提供（后面的文章会讲到）。

缓存使用

这里主要向小伙伴们介绍缓存中几个核心的注解使用。

@CacheConfig

这个注解在类上使用，用来描述该类中所有方法使用的缓存名称，当然也可以不使用该注解，直接在具体的缓存注解上配置名称，示例代码如下：

```
@Service  
@CacheConfig(cacheNames = "c1")  
public class UserService {  
}
```

@Cacheable

这个注解一般加在查询方法上，表示将一个方法的返回值缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。示例代码如下：

```
@Cacheable(key = "#id")  
public User getUserById(Integer id, String username) {  
    System.out.println("getUserById");  
    return getUserFromDBById(id);  
}
```

当有多个参数时，默认就使用多个参数来做 key，如果只需要其中某一个参数做 key，则可以在 @Cacheable 注解中，通过 key 属性来指定 key，如上代码就表示只使用 id 作为缓存的 key，如果对 key 有复杂的要求，可以自定义 keyGenerator。当然，Spring Cache 中提供了 root 对象，可以在不定义 keyGenerator 的情况下实现一些复杂的效果：

属性	描述	示例
methodName	当前方法名	#root.methodName
method	当前方法	#root.method.name
target	当前被调用的对象	#root.target
targetClass	当前被调用的对象的class	#root.targetClass
args	当前方法参数数组	#root.args[0]
caches	当前被调用的方法使用的Cache	#root.caches[0].name

@CachePut

这个注解一般加在更新方法上，当数据库中的数据更新后，缓存中的数据也要跟着更新，使用该注解，可以将方法的返回值自动更新到已经存在的 key 上，示例代码如下：

```
@CachePut(key = "#user.id")  
public User updateUserById(User user) {  
    return user;  
}
```

@CacheEvict

这个注解一般加在删除方法上，当数据库中的数据删除后，相关的缓存数据也要自动清除，该注解在使用的时候也可以配置按照某种条件删除（condition 属性）或者配置清除所有缓存（allEntries 属性），示例代码如下：

```
@CacheEvict()  
public void deleteUserById(Integer id) {  
    //在这里执行删除操作， 删除是去数据库中删除  
}
```

总结

在 Spring Boot 中，使用 Redis 缓存，既可以使用 RedisTemplate 自己来实现，也可以使用这种 方式，这种方式是 Spring Cache 提供的统一接口，实现既可以是 Redis，也可以是 Ehcache 或者其他 支持这种规范的缓存框架。从这个角度来说，Spring Cache 和 Redis、Ehcache 的关系就像 JDBC 与各 种数据库驱动的关系。

好了，关于这个问题就说到这里，有问题欢迎留言讨论。本文相关案例我已经上传到 GitHub，大家可 以自行下载：<https://github.com/lenvo/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



用惯了 Redis，很多人已经忘记了还有另一个缓存方案 Ehcache，是的，在 Redis 一统江湖的时代，Ehcache 渐渐有点没落了，不过，我们还是有必要了解下 Ehcache，在有的场景下，我们还是会用到 Ehcache。

今天松哥就来和大家聊聊 Spring Boot 中使用 Ehcache 的情况。相信看完本文，大家对于[Spring Boot2 系列教程\(二十六\)Spring Boot 整合 Redis](#)一文中的第二种方案会有更加深刻的理解。

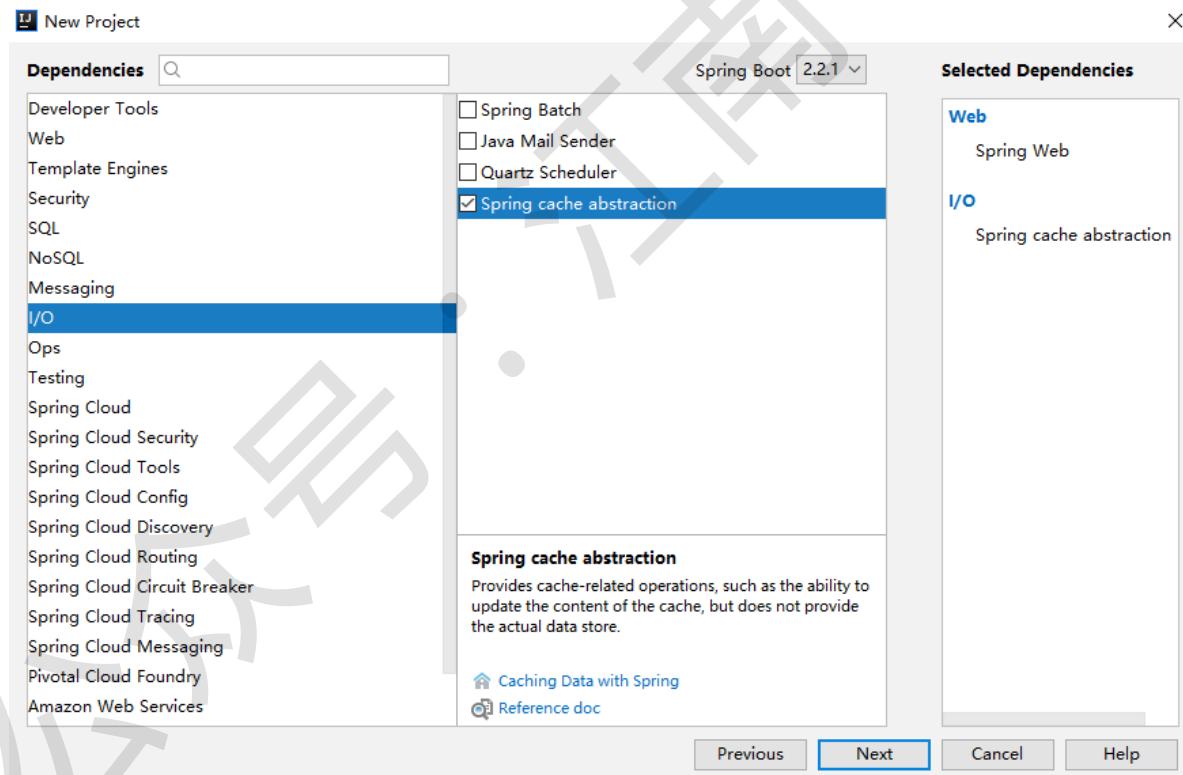
Ehcache 也是 Java 领域比较优秀的缓存方案之一，Ehcache 这个缓存的名字很有意思，正着念反着念，都是 Ehcache，Spring Boot 中对此也提供了很好的支持，这个支持主要是通过 Spring Cache 来实现的。

Spring Cache 可以整合 Redis，当然也可以整合 Ehcache，两种缓存方案的整合还是比较相似，主要是配置的差异，具体的用法是一模一样的，就类似于 JDBC 和数据库驱动的关系一样。前面配置完成后，后面具体使用的 API 都是一样的。

和 Spring Cache + Redis 相比，Spring Cache + Ehcache 主要是配置有所差异，具体的用法是一模一样的。我们来看下使用步骤。

项目创建

首先，来创建一个 Spring Boot 项目，引入 Cache 依赖：



工程创建完成后，引入 Ehcache 的依赖，Ehcache 目前有两个版本：

1. Ehcache

org.ehcache » ehcache

End-user ehcache3 jar artifact

Last Release on Apr 11, 2019

2. Ehcache

net.sf.ehcache » ehcache

Ehcache is an open source, standards-based cache used to boost performance, offload the database and simplify scalability. Ehcache is robust, proven and full-featured and this has made it the most widely-used Java-based cache.

Last Release on Oct 23, 2018

这里采用第二个，在 pom.xml 文件中，引入 Ehcache 依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-cache</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache</artifactId>
        <version>2.10.6</version>
    </dependency>
</dependencies>
```

添加 Ehcache 配置

在 resources 目录下，添加 ehcache 的配置文件 ehcache.xml，文件内容如下：

```
<ehcache>
    <diskStore path="java.io.tmpdir/shiro-spring-sample"/>
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="false"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
    />
    <cache name="user"
        maxElementsInMemory="10000"
        eternal="true"
        overflowToDisk="true"
        diskPersistent="true"
        diskExpiryThreadIntervalSeconds="600"/>
</ehcache>
```

配置含义：

1. name:缓存名称。
2. maxElementsInMemory: 缓存最大个数。
3. eternal:对象是否永久有效，一但设置了， timeout将不起作用。
4. timeToIdleSeconds: 设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。
5. timeToLiveSeconds: 设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。
6. overflowToDisk: 当内存中对象数量达到maxElementsInMemory时， Ehcache将会对对象写到磁盘中。
7. diskSpoolBufferSizeMB: 这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。
8. maxElementsOnDisk: 硬盘最大缓存个数。
9. diskPersistent: 是否缓存虚拟机重启期数据。
10. diskExpiryThreadIntervalSeconds: 磁盘失效线程运行时间间隔，默认是120秒。
11. memoryStoreEvictionPolicy: 当达到maxElementsInMemory限制时， Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。
12. clearOnFlush: 内存数量最大时是否清除。
13. diskStore 则表示临时缓存的硬盘目录。

注意

默认情况下，这个文件名是固定的，必须叫 ehcache.xml，如果一定要换一个名字，那么需要在 application.properties 中明确指定配置文件名，配置方式如下：

```
spring.cache.ehcache.config=classpath:aaa.xml
```

开启缓存

开启缓存的方式，也和 Redis 中一样，如下添加 `@EnableCaching` 依赖即可：

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

其实到这一步，Ehcache 就算配置完成了，接下来的用法，和松哥之前讲 Redis 的文章一模一样。不过这里松哥还是带大家使用下。

使用缓存

这里主要向小伙伴们介绍缓存中几个核心的注解使用。

`@CacheConfig`

这个注解在类上使用，用来描述该类中所有方法使用的缓存名称，当然也可以不使用该注解，直接在具体的缓存注解上配置名称，示例代码如下：

```
@Service  
@CacheConfig(cacheNames = "user")  
public class UserService {  
}
```

@Cacheable

这个注解一般加在查询方法上，表示将一个方法的返回值缓存起来，默认情况下，缓存的 key 就是方法的参数，缓存的 value 就是方法的返回值。示例代码如下：

```
@Cacheable(key = "#id")  
public User getUserById(Integer id, String username) {  
    System.out.println("getUserById");  
    return getUserFromDBById(id);  
}
```

当有多个参数时，默认就使用多个参数来做 key，如果只需要其中某一个参数做 key，则可以在 @Cacheable 注解中，通过 key 属性来指定 key，如上代码就表示只使用 id 作为缓存的 key，如果对 key 有复杂的要求，可以自定义 keyGenerator。当然，Spring Cache 中提供了 root 对象，可以在不定义 keyGenerator 的情况下实现一些复杂的效果，root 对象有如下属性：

属性	描述	示例
methodName	当前方法名	#root.methodName
method	当前方法	#root.method.name
target	当前被调用的对象	#root.target
targetClass	当前被调用的对象的 class	#root.targetClass
args	当前方法参数数组	#root.args[0]
caches	当前被调用的方法使用的 Cache	#root.caches[0].name

也可以通过 keyGenerator 自定义 key，方式如下：

```
@Component  
public class MyKeyGenerator implements KeyGenerator {  
    @Override  
    public Object generate(Object target, Method method, Object... params) {  
        return method.getName() + Arrays.toString(params);  
    }  
}
```

然后在方法上使用该 keyGenerator：

```
@Cacheable(keyGenerator = "myKeyGenerator")
public User getUserById(Long id) {
    User user = new User();
    user.setId(id);
    user.setUsername("lisi");
    System.out.println(user);
    return user;
}
```

@CachePut

这个注解一般加在更新方法上，当数据库中的数据更新后，缓存中的数据也要跟着更新，使用该注解，可以将方法的返回值自动更新到已经存在的 key 上，示例代码如下：

```
@CachePut(key = "#user.id")
public User updateUserById(User user) {
    return user;
}
```

@CacheEvict

这个注解一般加在删除方法上，当数据库中的数据删除后，相关的缓存数据也要自动清除，该注解在使用的时候也可以配置按照某种条件删除（ condition 属性）或者配置清除所有缓存（ allEntries 属性），示例代码如下：

```
@CacheEvict()
public void deleteUserById(Integer id) {
    //在这里执行删除操作， 删除是去数据库中删除
}
```

总结

本文主要向大家介绍了 Spring Boot 整合 Ehcache 的用法，其实说白了还是 Spring Cache 的用法。相信读完本文，大家对于 Redis + Spring Cache 的用法会有更深的认识。

本文案例我已上传到 GitHub，欢迎大家 star：<https://github.com/lenve/javaboy-code-samples>

关于本文，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

RESTful，到现在相信已经没人不知道这个东西了吧！关于 RESTful 的概念，我这里就不做过多介绍了，传统的 Struts 对 RESTful 支持不够友好，但是 SpringMVC 对于 RESTful 提供了很好的支持，常见的相关注解有：

```
@RestController  
 @GetMapping  
 @PutMapping  
 @PostMapping  
 @DeleteMapping  
 @ResponseBody  
 ...
```

这些注解都是和 RESTful 相关的，在移动互联网中，RESTful 得到了非常广泛的使用。RESTful 这个概念提出来很早，但是以前没有移动互联网时，我们做的大部分应用都是前后端不分的，在这种架构的应用中，数据基本上都是在后端渲染好返回给前端展示的，此时 RESTful 在 Web 应用中基本就没用武之地，移动互联网的兴起，让我们一套后台对应多个前端项目，因此前后端分离，RESTful 顺利走上前台。

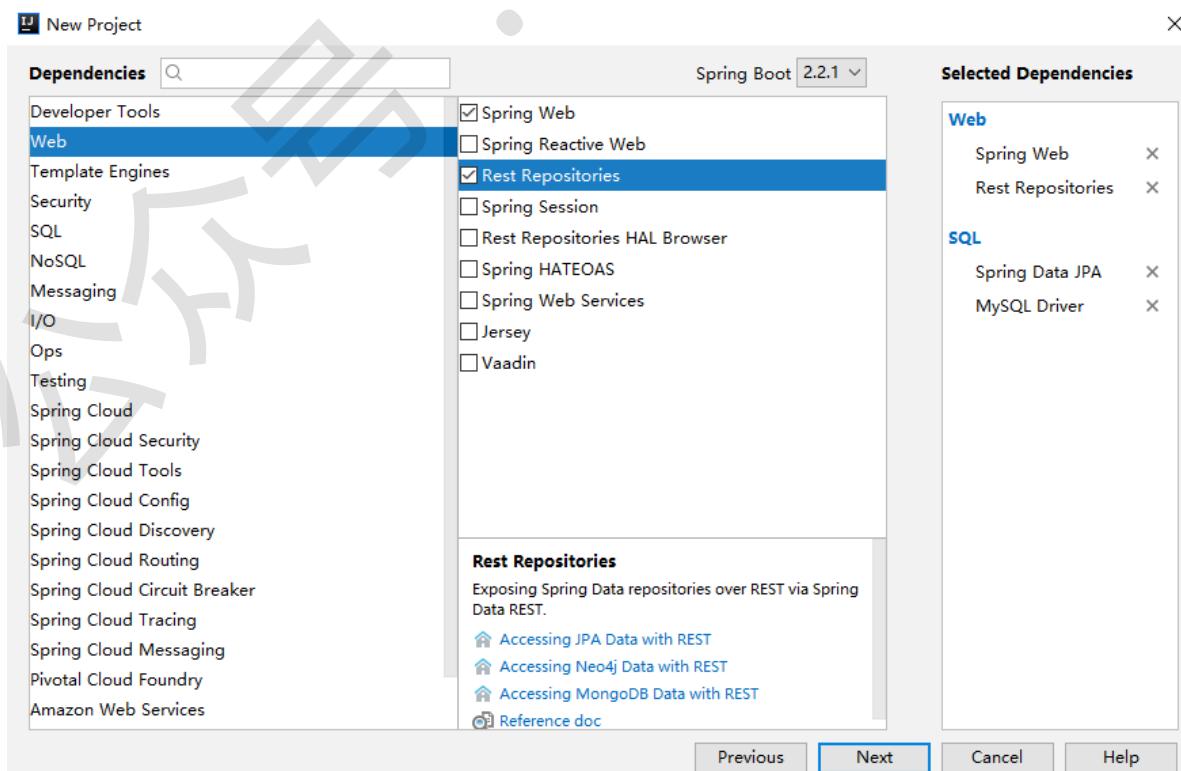
Spring Boot 继承自 Spring + SpringMVC，SpringMVC 中对于 RESTful 支持的特性在 Spring Boot 中全盘接收，同时，结合 Jpa 和 自动化配置，对于 RESTful 还提供了更多的支持，使得开发者几乎不需要写代码（很少几行），就能快速实现一个 RESTful 风格的增删改查。

接下来，松哥通过一个简单的案例，来向大家展示 Spring Boot 对于 RESTful 的支持。

实战

创建工程

首先创建一个 Spring Boot 工程，引入 web、Jpa、MySQL、Rest Repositories 依赖：



创建完成后，还需要锁定 MySQL 驱动的版本以及加入 Druid 数据库连接池，完整依赖如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-rest</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.10</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
        <version>5.1.27</version>
    </dependency>
</dependencies>
```

配置数据库

主要配置两个，一个是数据库，另一个是 Jpa：

```
spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.url=jdbc:mysql://test01
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=mysql
spring.jpa.database=mysql
```

这里的配置，和 Jpa 中的基本一致。

前面五行配置了数据库的基本信息，包括数据库连接池、数据库用户名、数据库密码、数据库连接地址以及数据库驱动名称。

接下来的五行配置了 JPA 的基本信息，分别表示生成 SQL 的方言、打印出生成的 SQL、每次启动项目时根据实际情况选择是否更新表、数据库平台是 MySQL。

这两段配置是关于 MySQL + JPA 的配置，没用过 JPA 的小伙伴可以参考松哥之前的 JPA 文章：
<http://www.javaboy.org/2019/0407/springboot-jpa.html>

构建实体类

```

@Entity(name = "t_book")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "book_name")
    private String name;
    private String author;
    //省略 getter/setter
}
public interface BookRepository extends JpaRepository<Book, Long> {
}

```

这里一个是配置了一个实体类 Book，另一个则是配置了一个 BookRepository，项目启动成功后，框架会根据 Book 类的定义，在数据库中自动创建相应的表，BookRepository 接口则是继承自 JpaRepository， JpaRepository 中自带了一些基本的增删改查方法。

好了，代码写完了。

啥？你好像啥都没写啊？是的，啥都没写，啥都不用写，一个 RESTful 风格的增删改查应用就有了，这就是 Spring Boot 的魅力！

测试

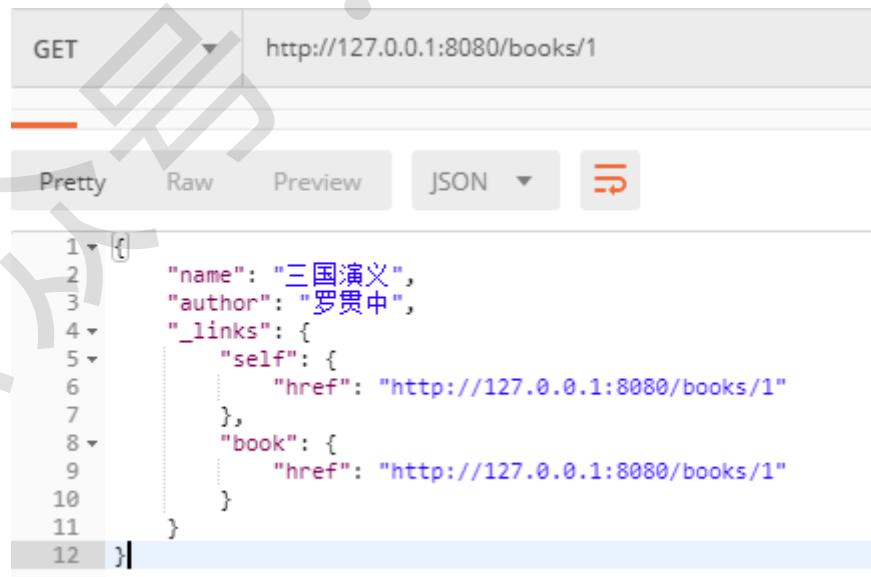
此时，我们就可以启动项目进行测试了，使用 POSTMAN 来测试（大家也可以自行选择趁手的 HTTP 请求工具）。

此时我们的项目已经默认具备了一些接口，我们分别来看：

根据 id 查询接口

- <http://127.0.0.1:8080/books/{id}>

这个接口表示根据 id 查询某一本书：



分页查询

- <http://127.0.0.1:8080/books>

这是一个批量查询接口，默认请求路径是类名首字母小写，并且再加一个 s 后缀。这个接口实际上是一个分页查询接口，没有传参数，表示查询第一页，每页 20 条数据。

GET http://127.0.0.1:8080/books

Pretty Raw Preview JSON

```
1 ▾ {  
2 ▾   "_embedded": {  
3 ▾     "books": [  
4 ▾       {  
5         "name": "三国演义",  
6         "author": "罗贯中",  
7         "_links": {  
8           "self": {  
9             "href": "http://127.0.0.1:8080/books/1"  
10          },  
11           "book": {  
12             "href": "http://127.0.0.1:8080/books/1"  
13           }  
14         }  
15       },  
16       {  
17         "name": "西游记",  
18         "author": "吴承恩",  
19         "_links": {  
20           "self": {  
21             "href": "http://127.0.0.1:8080/books/2"  
22           },  
23           "book": {  
24             "href": "http://127.0.0.1:8080/books/2"  
25           }  
26         }  
27       },  
28       {  
29         "name": "水浒传",  
30         "author": "施耐庵"  
31       }  
32     ]  
33   }  
34 }
```

查询结果中，除了该有的数据之外，也包含了分页数据：

GET http://127.0.0.1:8080/books

Pretty Raw Preview JSON

```
1 ▾ {  
2 ▾   "_embedded": {  
3 ▾     "books": [  
4 ▾       {  
5       }  
53     ],  
54   },  
55   "_links": {  
56     "self": {  
57       "href": "http://127.0.0.1:8080/books{?page,size,sort}",  
58       "templated": true  
59     },  
60     "profile": {  
61       "href": "http://127.0.0.1:8080/profile/books"  
62     }  
63   },  
64   "page": {  
65     "size": 20,  
66     "totalElements": 4,  
67     "totalPages": 1,  
68     "number": 0  
69   }  
70 }
```

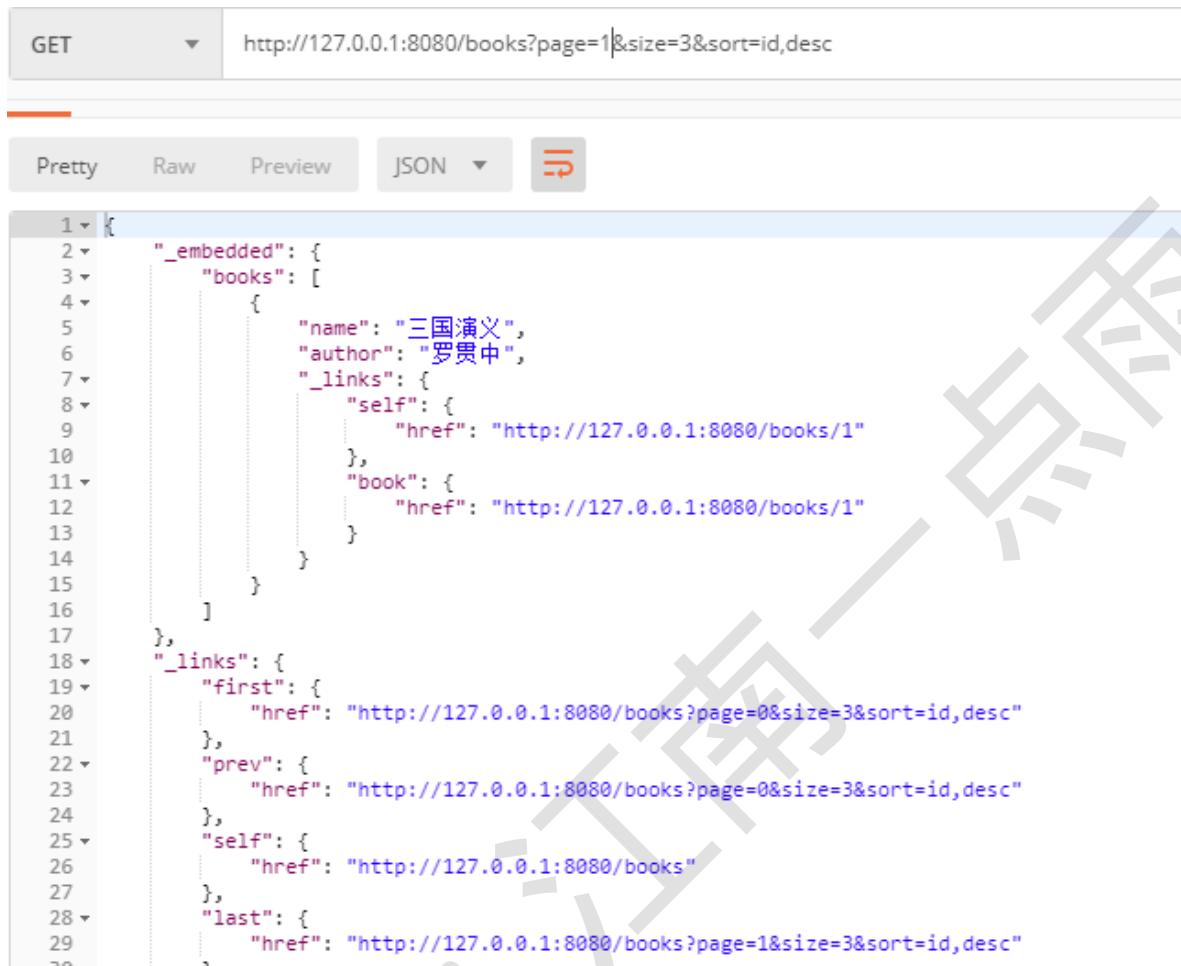
分页数据中：

1. size 表示每页查询记录数
2. totalElements 表示总记录数

3. totalPages 表示总页数

4. number 表示当前页数，从0开始计

如果要分页或者排序查询，可以使用 _links 中的链接。<http://127.0.0.1:8080/books?page=1&size=3&sort=id,desc>。



```
1 [
2   "_embedded": {
3     "books": [
4       {
5         "name": "三国演义",
6         "author": "罗贯中",
7         "_links": {
8           "self": {
9             "href": "http://127.0.0.1:8080/books/1"
10          },
11          "book": {
12            "href": "http://127.0.0.1:8080/books/1"
13          }
14        }
15      }
16    ],
17    "_links": {
18      "first": {
19        "href": "http://127.0.0.1:8080/books?page=0&size=3&sort=id,desc"
20      },
21      "prev": {
22        "href": "http://127.0.0.1:8080/books?page=0&size=3&sort=id,desc"
23      },
24      "self": {
25        "href": "http://127.0.0.1:8080/books"
26      },
27      "last": {
28        "href": "http://127.0.0.1:8080/books?page=1&size=3&sort=id,desc"
29      }
30    }
31  }
32]
```

添加

也可以添加数据，添加是 POST 请求，数据通过 JSON 的形式传递，如下：

The screenshot shows the Postman interface with a POST request to `http://127.0.0.1:8080/books`. The 'Body' tab is selected, showing a JSON payload: `{"name": "故事新编", "author": "鲁迅"}`. The response body is displayed in a pretty-printed JSON format:

```
1 {
2   "name": "故事新编",
3   "author": "鲁迅",
4   "_links": {
5     "self": {
6       "href": "http://127.0.0.1:8080/books/5"
7     },
8     "book": {
9       "href": "http://127.0.0.1:8080/books/5"
10    }
11  }
12 }
```

添加成功之后，默认会返回添加成功的数据。

修改

修改接口默认也是存在的，数据修改请求是一个 PUT 请求，修改的参数也是通过 JSON 的形式传递：

The screenshot shows the Postman interface for a PUT request. The URL is `http://127.0.0.1:8080/books/5`. The 'Body' tab is selected, showing a JSON payload:

```
1 {"name": "故事新编222", "author": "鲁迅"}
```

The response body is displayed in a JSONpretty format:

```
1 {
2   "name": "故事新编222",
3   "author": "鲁迅",
4   "_links": {
5     "self": {
6       "href": "http://127.0.0.1:8080/books/5"
7     },
8     "book": {
9       "href": "http://127.0.0.1:8080/books/5"
10    }
11  }
12 }
```

默认情况下，修改成功后，会返回修改成功的数据。

删除

当然也可以通过 DELETE 请求根据 id 删除数据：

The screenshot shows the Postman interface for a DELETE request. The URL is `http://127.0.0.1:8080/books/5`. The 'Body' tab is selected, showing the response body:

```
1 |
```

删除成功后，是没有返回值的。

不需要几行代码，一个基本的增删改查就有了。

这些都是默认的配置，这些默认的配置实际上都是在 JpaRepository 的基础上实现的，实际项目中，我们还可以对这些功能进行定制。

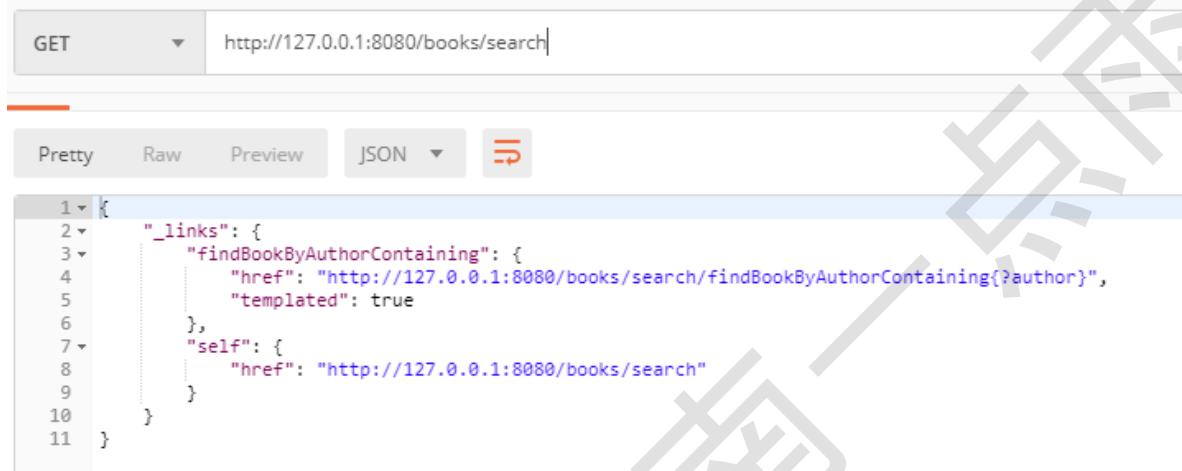
查询定制

最广泛的定制，就是查询，因为增删改操作的变化不像查询这么丰富。对于查询的定制，非常容易，只需要提供相关的方法即可。例如根据作者查询书籍：

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    List<Book> findBookByAuthorContaining(@Param("author") String author);  
}
```

注意，方法的定义，参数要有 @Param 注解。

定制完成后，重启项目，此时就多了一个查询接口，开发者可以通过 <http://localhost:8080/books/search> 来查看和 book 相关的自定义接口都有哪些：



查询结果表示，只有一个自定义接口，接口名就是方法名，而且查询结果还给出了接口调用的示例。我们来尝试调用一下自己定义的查询接口：



开发者可以根据实际情况，在 BookRepository 中定义任意多个查询方法，查询方法的定义规则和 Jpa 中一模一样（不懂 Jpa 的小伙伴，可以参考[王货 | 一文读懂 Spring Data Jpa!](#)，或者在松哥个人网站www.javaboy.org 上搜索 JPA，有相关教程参考）。但是，这样有一个缺陷，就是 Jpa 中方法名太长，因此，如果不想使用方法名作为接口名，则可以自定义接口名：

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    @RestResource(rel = "byauthor", path = "byauthor")  
    List<Book> findBookByAuthorContaining(@Param("author") String author);  
}
```

@RestResource 注解中，两个参数的含义：

- rel 表示接口查询中，这个方法的 key
- path 表示请求路径

这样定义完成后，表示接口名为 byauthor，重启项目，继续查询接口：

```
1 [ 2   "_links": { 3     "byauthor": { 4       "href": "http://127.0.0.1:8080/books/search/byauthor{?author}", 5       "templated": true 6     }, 7     "self": { 8       "href": "http://127.0.0.1:8080/books/search" 9     } 10   } 11 }
```

除了 rel 和 path 两个属性之外，@RestResource 中还有一个属性，exported 表示是否暴露接口，默认为 true，表示暴露接口，即方法可以在前端调用，如果仅仅只是想定义一个方法，不需要在前端调用这个方法，可以设置 exported 属性为 false。

如果不暴露官方定义好的方法，例如根据 id 删除数据，只需要在自定义接口中重写该方法，然后在该方法上加 @RestResource 注解并且配置相关属性即可。

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    @RestResource(rel = "byauthor", path = "byauthor")  
    List<Book> findBookByAuthorContaining(@Param("author") String author);  
    @Override  
    @RestResource(exported = false)  
    void deleteById(Long aLong);  
}
```

另外生成的 JSON 字符串中的集合名和单个 item 的名字都是可以自定义的：

```
@RepositoryRestResource(collectionResourceRel = "bs", itemResourceRel = "b", path  
= "bs")  
public interface BookRepository extends JpaRepository<Book, Long> {  
    @RestResource(rel = "byauthor", path = "byauthor")  
    List<Book> findBookByAuthorContaining(@Param("author") String author);  
    @Override  
    @RestResource(exported = false)  
    void deleteById(Long aLong);  
}
```

path 属性表示请求路径，请求路径默认是类名首字母小写+s，可以在这里自己重新定义。

```

1 [
2   "_embedded": {
3     "bs": [
4       {
5         "name": "三国演义",
6         "author": "罗贯中",
7         "_links": {
8           "self": {
9             "href": "http://127.0.0.1:8080/bs/1"
10          },
11          "b": {
12            "href": "http://127.0.0.1:8080/bs/1"
13          }
14        }
15      }
16    ],
17    "_links": {
18      "first": {
19        "href": "http://127.0.0.1:8080/bs?page=0&size=3&sort=id,desc"
20      },
21      "prev": {
22        "href": "http://127.0.0.1:8080/bs?page=0&size=3&sort=id,desc"
23      },
24      "self": {
25        "href": "http://127.0.0.1:8080/bs"
26      },
27      "last": {
28        "href": "http://127.0.0.1:8080/bs?page=1&size=3&sort=id,desc"
29      }
30    }
31  }
32]

```

其他配置

最后，也可以在 application.properties 中配置 REST 基本参数：

```

spring.data.rest.base-path=/api
spring.data.rest.sort-param-name=sort
spring.data.rest.page-param-name=page
spring.data.rest.limit-param-name=size
spring.data.rest.max-page-size=20
spring.data.rest.default-page-size=0
spring.data.rest.return-body-on-update=true
spring.data.rest.return-body-on-create=true

```

配置含义，从上往下，依次是：

1. 给所有的接口添加统一的前缀
2. 配置排序参数的 key， 默认是 sort
3. 配置分页查询时页码的 key， 默认是 page
4. 配置分页查询时每页查询页数的 key， 默认是 size
5. 配置每页最大查询记录数， 默认是 20 条
6. 分页查询时默认的页码
7. 更新成功时是否返回更新记录
8. 添加成功时是否返回添加记录

总结

本文主要向大家介绍了 Spring Boot 中快速实现一个 RESTful 风格的增删改查应用的方案，整体来说还是比较简单的，并不难。相关案例我已上传到 GitHub 上了，小伙伴可以自行下载：
<https://github.com/lenve/javaboy-code-samples>。

关于本文，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南一点雨

在 Spring Boot 中做权限管理，一般来说，主流的方案是 Spring Security，但是，仅仅从技术角度来说，也可以使用 Shiro。

今天松哥就来和大家聊聊 Spring Boot 整合 Shiro 的话题！

一般来说，Spring Security 和 Shiro 的比较如下：

1. Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架
2. Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单
3. Spring Security 功能强大；Shiro 功能简单
4. ...

虽然 Shiro 功能简单，但是也能满足大部分的业务场景。所以在传统的 SSM 项目中，一般来说，可以整合 Shiro。

在 Spring Boot 中，由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter，当然也提供了 Spring Security 的 Starter，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是 Spring Boot 项目，一般选择 Spring Security。

这是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。

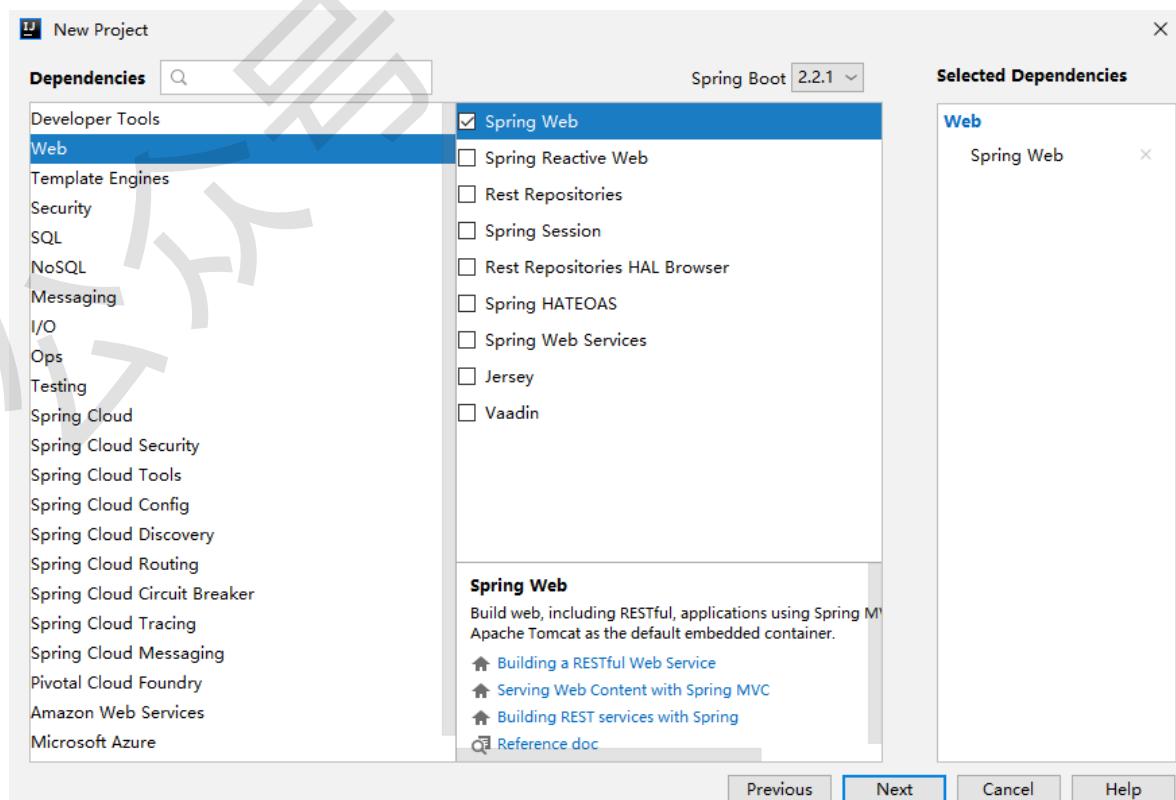
在 Spring Boot 中整合 Shiro，有两种不同的方案：

1. 第一种就是原封不动的，将 SSM 整合 Shiro 的配置用 Java 重写一遍。
2. 第二种就是使用 Shiro 官方提供的一个 Starter 来配置，但是，这个 Starter 并没有简化多少配置。

原生的整合

- 创建项目

创建一个 Spring Boot 项目，只需要添加 Web 依赖即可：



项目创建成功后，加入 Shiro 相关的依赖，完整的 pom.xml 文件中的依赖如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-web</artifactId>
        <version>1.4.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-spring</artifactId>
        <version>1.4.0</version>
    </dependency>
</dependencies>
```

- 创建 Realm

接下来我们来自定义核心组件 Realm：

```
public class MyRealm extends AuthorizingRealm {
    @Override
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
        return null;
    }
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) throws AuthenticationException {
        String username = (String) token.getPrincipal();
        if (!"javaboy".equals(username)) {
            throw new UnknownAccountException("账户不存在!");
        }
        return new SimpleAuthenticationInfo(username, "123", getName());
    }
}
```

在 Realm 中实现简单的认证操作即可，不做授权，授权的具体写法和 SSM 中的 Shiro 一样，不赘述。这里的认证表示用户名必须是 javaboy，用户密码必须是 123，满足这样的条件，就能登录成功！

- 配置 Shiro

接下来进行 Shiro 的配置：

```
@Configuration
public class ShiroConfig {
    @Bean
    MyRealm myRealm() {
        return new MyRealm();
    }

    @Bean
    SecurityManager securityManager() {
        DefaultWebSecurityManager manager = new DefaultWebSecurityManager();
```

```

        manager.setRealm(myRealm());
        return manager;
    }

    @Bean
    ShiroFilterFactoryBean shiroFilterFactoryBean() {
        ShiroFilterFactoryBean bean = new ShiroFilterFactoryBean();
        bean.setSecurityManager(securityManager());
        bean.setLoginUrl("/login");
        bean.setSuccessurl("/index");
        bean.setUnauthorizedurl("/unauthorizedurl");
        Map<String, String> map = new LinkedHashMap<>();
        map.put("/doLogin", "anon");
        map.put("/**", "authc");
        bean.setFilterChainDefinitionMap(map);
        return bean;
    }
}

```

在这里进行 Shiro 的配置主要配置 3 个 Bean :

1. 首先需要提供一个 Realm 的实例。
2. 需要配置一个 SecurityManager，在 SecurityManager 中配置 Realm。
3. 配置一个 ShiroFilterFactoryBean， 在 ShiroFilterFactoryBean 中指定路径拦截规则等。
4. 配置登录和测试接口。

其中，ShiroFilterFactoryBean 的配置稍微多一些，配置含义如下：

- setSecurityManager 表示指定 SecurityManager。
- setLoginUrl 表示指定登录页面。
- setSuccessUrl 表示指定登录成功页面。
- 接下来的 Map 中配置了路径拦截规则，注意，要有序。

这些东西都配置完成后，接下来配置登录 Controller:

```

@RestController
public class LoginController {
    @PostMapping("/doLogin")
    public void doLogin(String username, String password) {
        Subject subject = SecurityUtils.getSubject();
        try {
            subject.login(new UsernamePasswordToken(username, password));
            System.out.println("登录成功!");
        } catch (AuthenticationException e) {
            e.printStackTrace();
            System.out.println("登录失败!");
        }
    }
    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
    @GetMapping("/login")
    public String login() {
        return "please login!";
    }
}

```

测试时，首先访问 /hello 接口，由于未登录，所以会自动跳转到 /login 接口：

The screenshot shows a Postman interface with a GET request to `http://127.0.0.1:8080/hello`. The response body is displayed as "please login!".

然后调用 /doLogin 接口完成登录：

The screenshot shows a Postman interface with a POST request to `http://127.0.0.1:8080/doLogin`. The request body is set to "form-data" and contains two fields: "username" with value "javaboy" and "password" with value "123".

再次访问 /hello 接口，就可以成功访问了：

The screenshot shows a Postman interface with a GET request to `http://127.0.0.1:8080/hello`. The response body is displayed as "hello".

使用 Shiro Starter

上面这种配置方式实际上相当于把 SSM 中的 XML 配置拿到 Spring Boot 中用 Java 代码重新写了一遍，除了这种方式之外，我们也可以直接使用 Shiro 官方提供的 Starter。

- 创建工程，和上面的一样

创建成功后，添加 `shiro-spring-boot-web-starter`，这个依赖可以代替之前的 `shiro-web` 和 `shiro-spring` 两个依赖，`pom.xml` 文件如下：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-spring-boot-starter</artifactId>
        <version>1.4.0</version>
    </dependency>
</dependencies>
```

- 创建 Realm

这里的 Realm 和前面的一样，我就不再赘述。

- 配置 Shiro 基本信息

接下来在 application.properties 中配置 Shiro 的基本信息：

```
shiro.sessionManager.sessionIdCookieEnabled=true
shiro.sessionManager.sessionIdUrlRewritingEnabled=true
shiro.unauthorizedUrl=/unauthorizedurl
shiro.web.enabled=true
shiro.successUrl=/index
shiro.loginUrl=/login
```

配置解释：

1. 第一行表示是否允许将sessionId 放到 cookie 中
2. 第二行表示是否允许将 sessionId 放到 Url 地址栏中
3. 第三行表示访问未获授权的页面时，默认的跳转路径
4. 第四行表示开启 shiro
5. 第五行表示登录成功的跳转页面
6. 第六行表示登录页面

- 配置 ShiroConfig

```
@Configuration
public class ShiroConfig {
    @Bean
    MyRealm myRealm() {
        return new MyRealm();
    }
    @Bean
    DefaultWebSecurityManager securityManager() {
        DefaultWebSecurityManager manager = new DefaultWebSecurityManager();
        manager.setRealm(myRealm());
        return manager;
    }
    @Bean
    ShiroFilterChainDefinition shiroFilterChainDefinition() {
        DefaultShiroFilterChainDefinition definition = new
DefaultShiroFilterChainDefinition();
        definition.addPathDefinition("/doLogin", "anon");
        definition.addPathDefinition("/**", "authc");
        return definition;
    }
}
```

```
    }  
}
```

这里的配置和前面的比较像，但是不再需要 ShiroFilterFactoryBean 实例了，替代它的是 ShiroFilterChainDefinition，在这里定义 Shiro 的路径匹配规则即可。

这里定义完之后，接下来的登录接口定义以及测试方法都和前面的一致，我就不再赘述了。大家可以参考上文。

总结

本文主要向大家介绍了 Spring Boot 整合 Shiro 的两种方式，一种是传统方式的 Java 版，另一种则是使用 Shiro 官方提供的 Starter，两种方式，不知道大家有没有学会呢？

本文案例，我已经上传到 GitHub，欢迎大家 star：<https://github.com/lenve/javaboy-code-samples>

关于本文，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



Spring Security 是 Spring 家族中的一个安全管理框架，实际上，在 Spring Boot 出现之前，Spring Security 就已经发展了多年了，但是使用的并不多，安全管理这个领域，一直是 Shiro 的天下。

相对于 Shiro，在 SSM/SSH 中整合 Spring Security 都是比较麻烦的操作，所以，Spring Security 虽然功能比 Shiro 强大，但是使用反而没有 Shiro 多（Shiro 虽然功能没有 Spring Security 多，但是对于大部分项目而言，Shiro 也够用了）。

自从有了 Spring Boot 之后，Spring Boot 对于 Spring Security 提供了自动化配置方案，可以零配置使用 Spring Security。

因此，一般来说，常见的安全管理技术栈的组合是这样的：

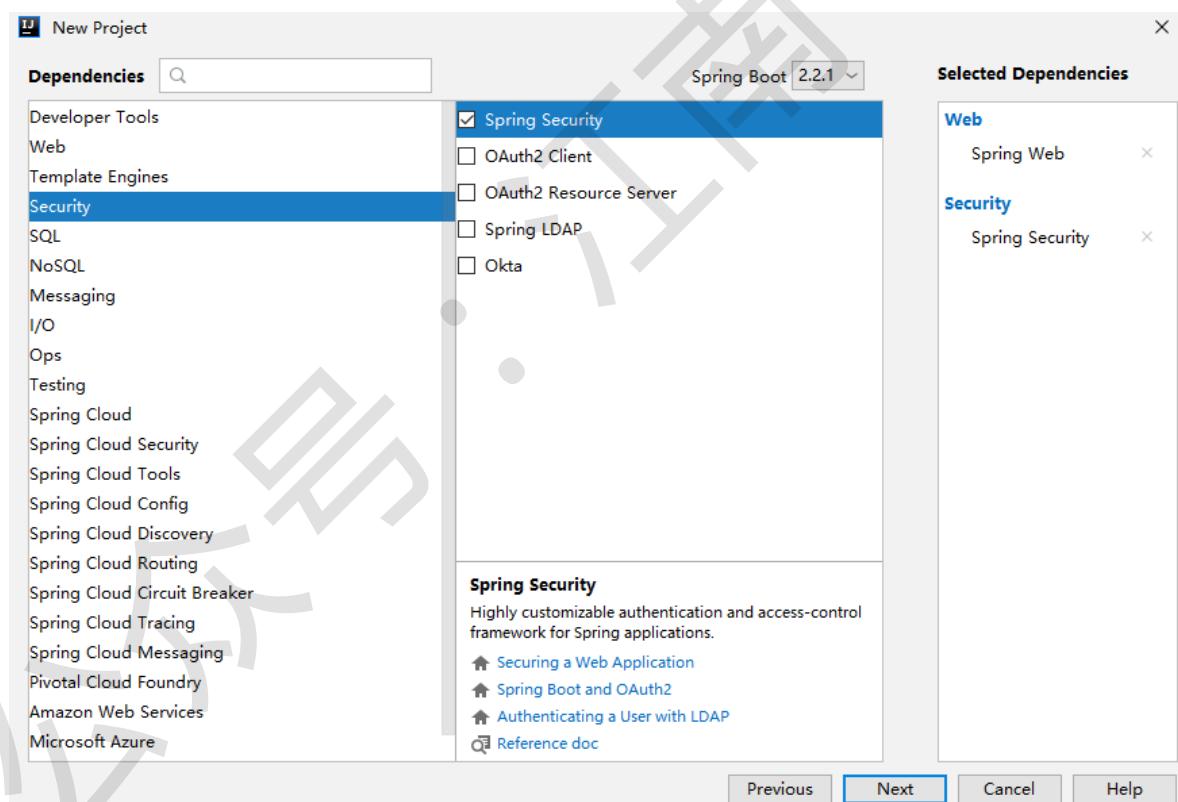
- SSM + Shiro
- Spring Boot/Spring Cloud + Spring Security

注意，这只是一个推荐的组合而已，如果单纯从技术上来说，无论怎么组合，都是可以运行的。

我们来看下具体使用。

1. 项目创建

在 Spring Boot 中使用 Spring Security 非常容易，引入依赖即可：



pom.xml 中的 Spring Security 依赖：

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

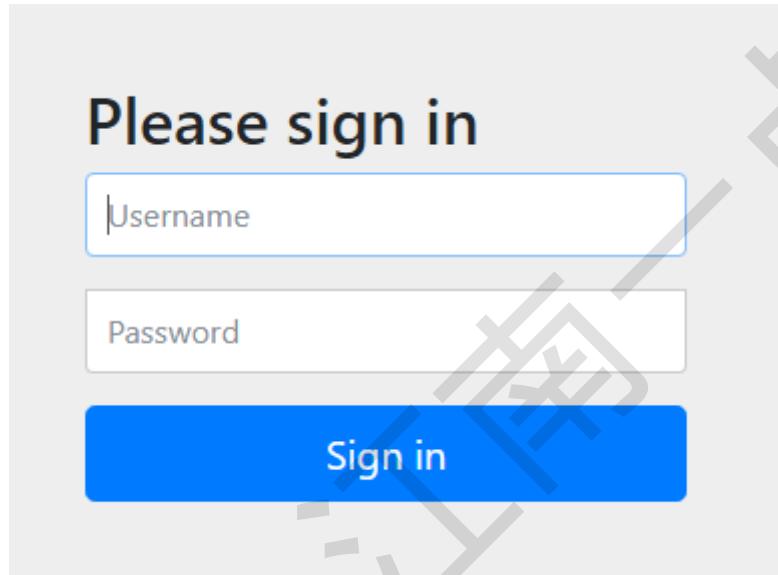
只要加入依赖，项目的所有接口都会被自动保护起来。

2. 初次体验

我们创建一个 HelloController:

```
@RestController  
public class HelloController {  
    @GetMapping("/hello")  
    public String hello() {  
        return "hello";  
    }  
}
```

访问 `/hello`，需要登录之后才能访问。



当用户从浏览器发送请求访问 `/hello` 接口时，服务端会返回 `302` 响应码，让客户端重定向到 `/login` 页面，用户在 `/login` 页面登录，登陆成功之后，就会自动跳转到 `/hello` 接口。

另外，也可以使用 `POSTMAN` 来发送请求，使用 `POSTMAN` 发送请求时，可以将用户信息放在请求头中（这样可以避免重定向到登录页面）：

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/hello`. The 'Authorization' tab is selected, showing 'Basic Auth' as the type. The 'Username' field is set to `user` and the 'Password' field is set to `3a8e222c-a64a-46f5-ac67-732f82626fa8`. A checked checkbox labeled 'Show Password' is also visible. At the bottom, the response status is shown as `200 OK` with a time of `755 ms` and a size of `377 B`.

通过以上两种不同的登录方式，可以看出，Spring Security 支持两种不同的认证方式：

- 可以通过 form 表单来认证
- 可以通过 HttpBasic 来认证

3.用户名配置

默认情况下，登录的用户名是 user，密码则是项目启动时随机生成的字符串，可以从启动的控制台日志中看到默认密码：

```
Using generated security password: 3a8e222c-a64a-46f5-ac67-732f82626fa8
```

这个随机生成的密码，每次启动时都会变。对登录的用户名/密码进行配置，有三种不同的方式：

- 在 application.properties 中进行配置
- 通过 Java 代码配置在内存中
- 通过 Java 从数据库中加载

前两种比较简单，第三种代码量略大，本文就先来看看前两种，第三种后面再单独写文章介绍，也可以参考我的[微人事项目](#)。

3.1 配置文件配置用户名/密码

可以直接在 application.properties 文件中配置用户的基本信息：

```
spring.security.user.name=javaboy  
spring.security.user.password=123
```

配置完成后，重启项目，就可以使用这里配置的用户名/密码登录了。

3.2 Java 配置用户名/密码

也可以在 Java 代码中配置用户名密码，首先需要我们创建一个 Spring Security 的配置类，集成自 WebSecurityConfigurerAdapter 类，如下：

```
@Configuration  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    @Override  
    protected void configure(AuthenticationManagerBuilder auth) throws Exception  
    {  
        //下面这两行配置表示在内存中配置了两个用户  
        auth.inMemoryAuthentication()  
  
            .withUser("javaboy").roles("admin").password("$2a$10$OR3VsksVAmCzc.7weaRPR.t0wyCsIj24k0Bne8iKwVlo.V9wsP8Xe")  
            .and()  
  
            .withUser("lisi").roles("user").password("$2a$10$p1H8iwa8I4.CA.7z8bwLjes91zpY.rYREGHQETnNtAp4NzL6PLKxi");  
    }  
    @Bean  
    PasswordEncoder passwordEncoder() {  
        return new BCryptPasswordEncoder();  
    }  
}
```

这里我们在 configure 方法中配置了两个用户，用户的密码都是加密之后的字符串(明文是 123)，从 Spring5 开始，强制要求密码要加密，如果非不想加密，可以使用一个过期的 PasswordEncoder 的实例 NoOpPasswordEncoder，但是不建议这么做，毕竟不安全。

Spring Security 中提供了 BCryptPasswordEncoder 密码编码工具，可以非常方便的实现密码的加密加盐，相同明文加密出来的结果总是不同，这样就不需要用户去额外保存 盐 的字段了，这一点比 Shiro 要方便很多。

4. 登录配置

对于登录接口，登录成功后的响应，登录失败后的响应，我们都可以在 WebSecurityConfigurerAdapter 的实现类中进行配置。例如下面这样：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    VerifyCodeFilter verifyCodeFilter;
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterBefore(verifyCodeFilter,
        UsernamePasswordAuthenticationFilter.class);
        http
            .authorizeRequests() //开启登录配置
            .antMatchers("/hello").hasRole("admin") //表示访问 /hello 这个接口，需要具备 admin 这个角色
            .anyRequest().authenticated() //表示剩余的其他接口，登录之后就能访问
            .and()
            .formLogin()
        //定义登录页面，未登录时，访问一个需要登录之后才能访问的接口，会自动跳转到该页面
        .loginPage("/login_p")
        //登录处理接口
        .loginProcessingUrl("/doLogin")
        //定义登录时，用户名的 key，默认为 username
        .usernameParameter("uname")
        //定义登录时，用户密码的 key，默认为 password
        .passwordParameter("passwd")
        //登录成功的处理器
        .successHandler(new AuthenticationSuccessHandler() {
            @Override
            public void onAuthenticationSuccess(HttpServletRequest req,
            HttpServletResponse resp, Authentication authentication) throws IOException,
            ServletException {
                resp.setContentType("application/json;charset=utf-8");
                PrintWriter out = resp.getWriter();
                out.write("success");
                out.flush();
            }
        })
        .failureHandler(new AuthenticationFailureHandler() {
            @Override
            public void onAuthenticationFailure(HttpServletRequest req,
            HttpServletResponse resp, AuthenticationException exception) throws IOException,
            ServletException {
                resp.setContentType("application/json;charset=utf-8");
                PrintWriter out = resp.getWriter();
                out.write("fail");
            }
        })
    }
}
```

```
        out.flush();
    }
})
.permitAll() //和表单登录相关的接口统统都直接通过
.and()
.logout()
.logoutUrl("/logout")
.logoutSuccessHandler(new LogoutSuccessHandler() {
    @Override
    public void onLogoutSuccess(HttpServletRequest req,
HttpServletResponse resp, Authentication authentication) throws IOException,
ServletException {
        resp.setContentType("application/json; charset=utf-8");
        PrintWriter out = resp.getWriter();
        out.write("logout success");
        out.flush();
    }
})
.permitAll()
.and()
.httpBasic()
.and()
.csrf().disable();
}
}
```

我们可以在 successHandler 方法中，配置登录成功的回调，如果是前后端分离开发的话，登录成功后返回 JSON 即可，同理，failureHandler 方法中配置登录失败的回调，logoutSuccessHandler 中则配置注销成功的回调。

5. 忽略拦截

如果某一个请求地址不需要拦截的话，有两种方式实现：

- 设置该地址匿名访问
- 直接过滤掉该地址，即该地址不走 Spring Security 过滤器链

推荐使用第二种方案，配置如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/vercode");
    }
}
```

Spring Security 另外一个强大之处就是它可以结合 OAuth2，玩出更多的花样出来，这些我们在后面的文章中再和大家细细介绍。

本文案例，我已经上传到 GitHub，欢迎大家 star：<https://github.com/lenve/javaboy-code-samples>

本文就先说到这里，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

登录添加验证码是一个非常常见的需求，网上也有非常成熟的解决方案，其实，要是自己自定义登录实现这个并不难，但是如果需要在 Spring Security 框架中实现这个功能，还得稍费一点功夫，本文就和小伙伴来分享下在 Spring Security 框架中如何添加验证码。

关于 Spring Security 基本配置，这里就不再多说，小伙伴有不懂的可以参考<http://springboot.javaboy.org/>，本文主要来看如何加入验证码功能。

准备验证码

要有验证码，首先得先准备好验证码，本文采用 Java 自画的验证码，代码如下：

```
/**
 * 生成验证码的工具类
 */
public class VerifyCode {

    private int width = 100;// 生成验证码图片的宽度
    private int height = 50;// 生成验证码图片的高度
    private String[] fontNames = { "宋体", "楷体", "隶书", "微软雅黑" };
    private Color bgColor = new Color(255, 255, 255);// 定义验证码图片的背景颜色为白色
    private Random random = new Random();
    private String codes =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    private String text;// 记录随机字符串

    /**
     * 获取一个随意颜色
     *
     * @return
     */
    private Color randomColor() {
        int red = random.nextInt(150);
        int green = random.nextInt(150);
        int blue = random.nextInt(150);
        return new Color(red, green, blue);
    }

    /**
     * 获取一个随机字体
     *
     * @return
     */
    private Font randomFont() {
        String name = fontNames[random.nextInt(fontNames.length)];
        int style = random.nextInt(4);
        int size = random.nextInt(5) + 24;
        return new Font(name, style, size);
    }

    /**
     * 获取一个随机字符
     *
     * @return
     */
}
```

```
private char randomchar() {
    return codes.charAt(random.nextInt(codes.length()));
}

/**
 * 创建一个空白的BufferedImage对象
 *
 * @return
 */
private BufferedImage createImage() {
    BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
    Graphics2D g2 = (Graphics2D) image.getGraphics();
    g2.setColor(bgColor); // 设置验证码图片的背景颜色
    g2.fillRect(0, 0, width, height);
    return image;
}

public BufferedImage getImage() {
    BufferedImage image = createImage();
    Graphics2D g2 = (Graphics2D) image.getGraphics();
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < 4; i++) {
        String s = randomChar() + "";
        sb.append(s);
        g2.setColor(randomColor());
        g2.setFont(randomFont());
        float x = i * width * 1.0f / 4;
        g2.drawString(s, x, height - 15);
    }
    this.text = sb.toString();
    drawLine(image);
    return image;
}

/**
 * 绘制干扰线
 *
 * @param image
 */
private void drawLine(BufferedImage image) {
    Graphics2D g2 = (Graphics2D) image.getGraphics();
    int num = 5;
    for (int i = 0; i < num; i++) {
        int x1 = random.nextInt(width);
        int y1 = random.nextInt(height);
        int x2 = random.nextInt(width);
        int y2 = random.nextInt(height);
        g2.setColor(randomColor());
        g2.setStroke(new BasicStroke(1.5f));
        g2.drawLine(x1, y1, x2, y2);
    }
}

public String getText() {
    return text;
}
```

```
    public static void output(BufferedImage image, OutputStream out) throws
IOException {
    ImageIO.write(image, "JPEG", out);
}
}
```

这个工具类很常见，网上也有很多，就是画一个简单的验证码，通过流将验证码写到前端页面，提供验证码的 Controller 如下：

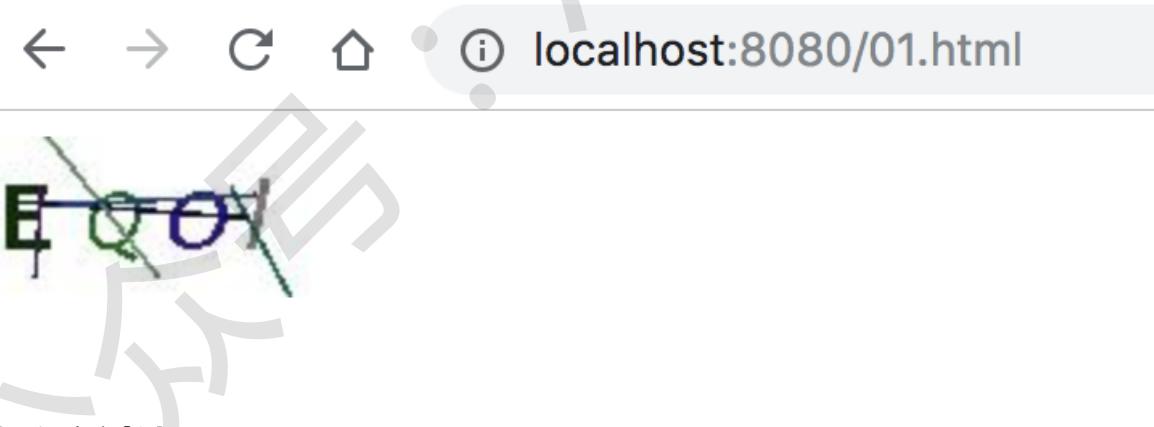
```
@RestController
public class VerifyCodeController {
    @GetMapping("/vercode")
    public void code(HttpServletRequest req, HttpServletResponse resp) throws
IOException {
        VerifyCode vc = new VerifyCode();
        BufferedImage image = vc.getImage();
        String text = vc.getText();
        HttpSession session = req.getSession();
        session.setAttribute("index_code", text);
        VerifyCode.output(image, resp.getOutputStream());
    }
}
```

这里创建了一个 VerifyCode 对象，将生成的验证码字符保存到 session 中，然后通过流将图片写到前端，img 标签如下：

```

```

展示效果如下：



自定义过滤器

在登陆页展示验证码这个就不需要我多说了，接下来我们来看看如何自定义验证码处理器：

```
@Component
public class VerifyCodeFilter extends GenericFilterBean {
    private String defaultFilterProcessUrl = "/doLogin";

    @Override
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain
chain)
        throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest) req;
        HttpServletResponse response = (HttpServletResponse) res;
```

```

        if ("POST".equalsIgnoreCase(request.getMethod()) &&
defaultFilterProcessUrl.equals(request.getServletPath())) {
            // 验证码验证
            String requestCaptcha = request.getParameter("code");
            String genCaptcha = (String)
request.getSession().getAttribute("index_code");
            if (StringUtils.isEmpty(requestCaptcha))
                throw new AuthenticationServiceException("验证码不能为空!");
            if (!genCaptcha.toLowerCase().equals(requestCaptcha.toLowerCase()))
{
                throw new AuthenticationServiceException("验证码错误!");
            }
        }
        chain.doFilter(request, response);
    }
}

```

自定义过滤器继承自 GenericFilterBean，并实现其中的 doFilter 方法，在 doFilter 方法中，当请求方法是 POST，并且请求地址是 /doLogin 时，获取参数中的 code 字段值，该字段保存了用户从前端页面传来的验证码，然后获取 session 中保存的验证码，如果用户没有传来验证码，则抛出验证码不能为空异常，如果用户传入了验证码，则判断验证码是否正确，如果不正确则抛出异常，否则执行 chain.doFilter(request, response); 使请求继续向下走。

配置

最后在 Spring Security 的配置中，配置过滤器，如下：

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    VerifyCodeFilter verifyCodeFilter;
    ...
    ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterBefore(verifyCodeFilter,
UsernamePasswordAuthenticationFilter.class);
        http.authorizeRequests()
            .antMatchers("/admin/**").hasRole("admin")
        ...
        ...
        .permitAll()
        .and()
        .csrf().disable();
    }
}

```

这里只贴出了部分核心代码，即 http.addFilterBefore(verifyCodeFilter, UsernamePasswordAuthenticationFilter.class);，如此之后，整个配置就算完成了。

接下来在登录中，就需要传入验证码了，如果不传或者传错，都会抛出异常，例如不传的话，抛出如下异常：

POST ▼

http://localhost:8080/doLogin

Authorization

Headers (1)

Body ●

Pre-request Script

Tests

form-data

x-www-form-urlencoded

raw

binary

Key

Value

username

admin

password

123

New key

Value

Body

Cookies

Headers (10)

Test Results

Pretty

Raw

Preview

JSON ▾



```
1 {  
2   "timestamp": "2019-02-25T12:05:38.044+0000",  
3   "status": 500,  
4   "error": "Internal Server Error",  
5   "message": "验证码不能为空!",  
6   "path": "/doLogin"  
7 }
```

本文案例，我已经上传到 GitHub，欢迎大家 star: <https://github.com/lenve/javaboy-code-samples>

好了，本文就先说到这里，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



在使用 SpringSecurity 中，大伙都知道默认的登录数据是通过 key/value 的形式来传递的，默认情况下不支持 JSON 格式的登录数据，如果有这种需求，就需要自己来解决，本文主要和小伙伴们来聊聊这个话题。

基本登录方案

在说如何使用 JSON 登录之前，我们还是先来看看基本的登录吧，本文为了简单，SpringSecurity 在使用中就不连接数据库了，直接在内存中配置用户名和密码，具体操作步骤如下：

- 创建 Spring Boot 工程

首先创建 SpringBoot 工程，添加 SpringSecurity 依赖，如下：

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

- 添加 Security 配置

创建 SecurityConfig，完成 SpringSecurity 的配置，如下：

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().withUser("zhangsan").password("$2a$10$204EWLrrFPE
boTfDOTC0F.RpUMk.3q3KvBHRx7XXKUMLBGjOOBs8q").roles("user");
    }

    @Override
    public void configure(WebSecurity web) throws Exception {
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginProcessingUrl("/doLogin")
            .successHandler(new AuthenticationSuccessHandler() {
                @Override
```

```
        public void onAuthenticationSuccess(HttpServletRequest req,
HttpServletResponse resp, Authentication authentication) throws IOException,
ServletException {
            RespBean ok = RespBean.ok("登录成功!", authentication.getPrincipal());
            resp.setContentType("application/json;charset=utf-8");
            PrintWriter out = resp.getWriter();
            out.write(new ObjectMapper().writeValueAsString(ok));
            out.flush();
            out.close();
        }
    })
.failureHandler(new AuthenticationFailureHandler() {
    @Override
    public void onAuthenticationFailure(HttpServletRequest req,
HttpServletResponse resp, AuthenticationException e) throws IOException,
ServletException {
        RespBean error = RespBean.error("登录失败");
        resp.setContentType("application/json;charset=utf-8");
        PrintWriter out = resp.getWriter();
        out.write(new ObjectMapper().writeValueAsString(error));
        out.flush();
        out.close();
    }
})
.loginPage("/login")
.permitAll()
.and()
.logout()
.logoutUrl("/logout")
.logoutSuccessHandler(new LogoutSuccessHandler() {
    @Override
    public void onLogoutSuccess(HttpServletRequest req,
HttpServletResponse resp, Authentication authentication) throws IOException,
ServletException {
        RespBean ok = RespBean.ok("注销成功!");
        resp.setContentType("application/json;charset=utf-8");
        PrintWriter out = resp.getWriter();
        out.write(new ObjectMapper().writeValueAsString(ok));
        out.flush();
        out.close();
    }
})
.permitAll()
.and()
.csrf()
.disable()
.exceptionHandling()
.accessDeniedHandler(new AccessDeniedHandler() {
    @Override
    public void handle(HttpServletRequest req,
HttpServletResponse resp, AccessDeniedException e) throws IOException,
ServletException {
        RespBean error = RespBean.error("权限不足，访问失败");
        resp.setStatus(403);
        resp.setContentType("application/json;charset=utf-8");
        PrintWriter out = resp.getWriter();
        out.write(new ObjectMapper().writeValueAsString(error));
    }
})
```

```
        out.flush();
        out.close();
    }
}

}
```

这里的配置虽然有点长，但是很基础，配置含义也比较清晰，首先提供 BCryptPasswordEncoder 作为 PasswordEncoder，可以实现对密码的自动加密加盐，非常方便，然后提供了一个名为 zhangsan 的用户，密码是 123，角色是 user，最后配置登录逻辑，所有的请求都需要登录后才能访问，登录接口是 /doLogin，用户名的 key 是 username，密码的 key 是 password，同时配置登录成功、登录失败以及注销成功、权限不足时都给用户返回 JSON 提示，另外，这里虽然配置了登录页面为 /login，实际上这不是一个页面，而是一段 JSON，在 LoginController 中提供该接口，如下：

```
@RestController
@ResponseBody
public class LoginController {
    @GetMapping("/login")
    public RespBean login() {
        return RespBean.error("尚未登录，请登录");
    }
    @GetMapping("/hello")
    public String hello() {
        return "hello";
    }
}
```

这里 /login 只是一个 JSON 提示，而不是页面，/hello 则是一个测试接口。

OK，做完上述步骤就可以开始测试了，运行 SpringBoot 项目，访问 /hello 接口，结果如下：

The screenshot shows a REST client interface with the following details:

- Method: GET
- URL: http://localhost:8080/hello
- Body tab is selected.
- Headers (9) tab is visible.
- Pretty tab is selected under Body.
- Raw tab is also present under Body.
- Preview tab is visible.
- JSON dropdown is set to JSON.
- Copy icon is visible.
- Response body (Pretty):

```
1 {  
2     "status": 500,  
3     "msg": "尚未登录，请登录",  
4     "obj": null  
5 }
```

此时先调用登录接口进行登录，如下：

POST ▼ http://localhost:8080/doLogin

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary

Key	Value
<input checked="" type="checkbox"/> username	zhangsan
<input checked="" type="checkbox"/> password	123
New key	Value

Body Cookies Headers (9) Test Results

Pretty Raw Preview JSON

```

1 {  
2   "status": 200,  
3   "msg": "登录成功! ",  
4   "obj": {  
5     "password": null,  
6     "username": "zhangsan",  
7     "authorities": [  
8       {  
9         "authority": "ROLE_user"  
10      }  
11    ],  
12    "accountNonExpired": true,  
13    "accountNonLocked": true,  
14    "credentialsNonExpired": true,  
15    "enabled": true  
16  }  
17 }
```

登录成功后，再去访问 /hello 接口就可以成功访问了。

使用 JSON 登录

上面演示的是一种原始的登录方案，如果想将用户名密码通过 JSON 的方式进行传递，则需要自定义相关过滤器，通过分析源码我们发现，默认的用户名密码提取在 UsernamePasswordAuthenticationFilter 过滤器中，部分源码如下：

```

public class UsernamePasswordAuthenticationFilter extends
    AbstractAuthenticationProcessingFilter {
    public static final String SPRING_SECURITY_FORM_USERNAME_KEY = "username";
    public static final String SPRING_SECURITY_FORM_PASSWORD_KEY = "password";

    private String usernameParameter = SPRING_SECURITY_FORM_USERNAME_KEY;
    private String passwordParameter = SPRING_SECURITY_FORM_PASSWORD_KEY;
    private boolean postOnly = true;
    public UsernamePasswordAuthenticationFilter() {
        super(new AntPathRequestMatcher("/login", "POST"));
    }

    public Authentication attemptAuthentication(HttpServletRequest request,
        HttpServletResponse response) throws AuthenticationServiceException {
        if (postOnly && !request.getMethod().equals("POST")) {
            throw new AuthenticationServiceException(
                "Authentication method [" + request.getMethod() +
                "] is not supported." + getErrorMessage());
        }
    }
}
```

```

        "Authentication method not supported: " +
request.getMethod());
    }

    String username = obtainUsername(request);
    String password = obtainPassword(request);

    if (username == null) {
        username = "";
    }

    if (password == null) {
        password = "";
    }

    username = username.trim();

    UsernamePasswordAuthenticationToken authRequest = new
UsernamePasswordAuthenticationToken(
        username, password);

    // Allow subclasses to set the "details" property
    setDetails(request, authRequest);

    return this.getAuthenticationManager().authenticate(authRequest);
}

protected String obtainPassword(HttpServletRequest request) {
    return request.getParameter(passwordParameter);
}

protected String obtainUsername(HttpServletRequest request) {
    return request.getParameter(usernameParameter);
}
//...
//...
}

```

从这里可以看到，默认的用户名/密码 提取就是通过 request 中的 getParameter 来提取的，如果想使用 JSON 传递用户名密码，只需要将这个过滤器替换掉即可，自定义过滤器如下：

```

public class CustomAuthenticationFilter extends
UsernamePasswordAuthenticationFilter {
    @Override
    public Authentication attemptAuthentication(HttpServletRequest request,
HttpServletResponse response) throws AuthenticationException {
        if
(request.getContentType().equals(MediaType.APPLICATION_JSON_UTF8_VALUE)
        ||
request.getContentType().equals(MediaType.APPLICATION_JSON_VALUE)) {
            ObjectMapper mapper = new ObjectMapper();
            UsernamePasswordAuthenticationToken authRequest = null;
            try (InputStream is = request.getInputStream()) {
                Map<String, String> authenticationBean = mapper.readValue(is,
Map.class);
                authRequest = new UsernamePasswordAuthenticationToken(

```

```
        authenticationBean.get("username"),
authenticationBean.get("password"));
    } catch (IOException e) {
        e.printStackTrace();
        authRequest = new UsernamePasswordAuthenticationToken(
            "", "");
    } finally {
        setDetails(request, authRequest);
        return
this.getAuthenticationManager().authenticate(authRequest);
    }
}
else {
    return super.attemptAuthentication(request, response);
}
}
```

这里只是将用户名/密码的获取方案重新修正下，改为了从 JSON 中获取用户名密码，然后在 SecurityConfig 中作出如下修改：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().authenticated()
        .and()
        .formLogin()
        .and().csrf().disable();
    http.addFilterAt(customAuthenticationFilter(),
        UsernamePasswordAuthenticationFilter.class);
}
@Bean
CustomAuthenticationFilter customAuthenticationFilter() throws Exception {
    CustomAuthenticationFilter filter = new CustomAuthenticationFilter();
    filter.setAuthenticationSuccessHandler(new AuthenticationSuccessHandler() {
        @Override
        public void onAuthenticationSuccess(HttpServletRequest req,
        HttpServletResponse resp, Authentication authentication) throws IOException,
        ServletException {
            resp.setContentType("application/json; charset=utf-8");
            PrintWriter out = resp.getWriter();
            RespBean respBean = RespBean.ok("登录成功!");
            out.write(new ObjectMapper().writeValueAsString(respBean));
            out.flush();
            out.close();
        }
    });
    filter.setAuthenticationFailureHandler(new AuthenticationFailureHandler() {
        @Override
        public void onAuthenticationFailure(HttpServletRequest req,
        HttpServletResponse resp, AuthenticationException e) throws IOException,
        ServletException {
            resp.setContentType("application/json; charset=utf-8");
            PrintWriter out = resp.getWriter();
            RespBean respBean = RespBean.error("登录失败!");
            out.write(new ObjectMapper().writeValueAsString(respBean));
            out.flush();
            out.close();
        }
    });
}
```

```
        }
    });
    filter.setAuthenticationManager(authenticationManagerBean());
    return filter;
}
```

将自定义的 CustomAuthenticationFilter 类加入进来即可，接下来就可以使用 JSON 进行登录了，如下：

The screenshot shows a Postman interface. At the top, it says 'POST' and 'http://localhost:8080/login'. Below that, there are four options: 'form-data', 'x-www-form-urlencoded', 'raw', and 'binary', with 'raw' selected. The 'JSON (application/json)' option is also visible. In the 'Body' section, there is a code editor containing the following JSON:

```
1 {"username": "zhangsan", "password": "123"}
```

Below the code editor, there are tabs for 'Body', 'Cookies', 'Headers (9)', and 'Test Results'. The 'Body' tab is selected. Under 'Pretty', the response is shown as:

```
1 {
2     "status": 200,
3     "msg": "登录成功!",
4     "obj": null
5 }
```

本文案例，我已经上传到 GitHub，欢迎大家 star: <https://github.com/lenvve/javaboy-code-samples>

好了，本文就先介绍到这里，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



今天想和小伙伴们来聊一聊 SpringSecurity 中的角色继承问题。

角色继承实际上是一个很常见的需求，因为大部分公司治理可能都是金字塔形的，上司可能具备下属的部分甚至所有权限，这一现实场景，反映到我们的代码中，就是角色继承了。Spring Security 中为开发者提供了相关的角色继承解决方案，但是这一解决方案在最近的SpringSecurity 版本变迁中，使用方法有所变化。今天除了和小伙伴们分享角色继承外，也来顺便说说这种变化，避免小伙伴们踩坑，同时购买了我的书的小伙伴也需要留意，书是基于 Spring Boot2.0.4 这个版本写的，这个话题和最新版 Spring Boot 的还是有一点差别。

1. 版本分割线

上文说过，SpringSecurity 在角色继承上有两种不同的写法，在 Spring Boot2.0.8（对应 Spring Security 也是5.0.11）上面是一种写法，从 Spring Boot2.1.0（对应 Spring Security5.1.1）又是另外一种写法，本文将从这两种角度出发，向读者介绍两种不同的角色继承写法。

2. 以前的写法

这里说的以前写法，就是指 SpringBoot2.0.8（含）之前的写法，在之前的写法中，角色继承只需要开发者提供一个 RoleHierarchy 接口的实例即可，例如下面这样：

```
@Bean
RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    String hierarchy = "ROLE_dba > ROLE_admin ROLE_admin > ROLE_user";
    roleHierarchy.setHierarchy(hierarchy);
    return roleHierarchy;
}
```

在这里我们提供了一个 RoleHierarchy 接口的实例，使用字符串来描述了角色之间的继承关系，`ROLE_dba` 具备 `ROLE_admin` 的所有权限，而 `ROLE_admin` 则具备 `ROLE_user` 的所有权限，继承与继承之间用一个空格隔开。提供了这个 Bean 之后，以后所有具备 `ROLE_user` 角色才能访问的资源，`ROLE_dba` 和 `ROLE_admin` 也都能访问，具备 `ROLE_admin` 角色才能访问的资源，`ROLE_dba` 也能访问。

3. 现在的写法

但是上面这种写法仅限于 SpringBoot2.0.8（含）之前的版本，在之后的版本中，这种写法则不被支持，新版的写法是下面这样：

```
@Bean
RoleHierarchy roleHierarchy() {
    RoleHierarchyImpl roleHierarchy = new RoleHierarchyImpl();
    String hierarchy = "ROLE_dba > ROLE_admin \n ROLE_admin > ROLE_user";
    roleHierarchy.setHierarchy(hierarchy);
    return roleHierarchy;
}
```

变化主要就是分隔符，将原来用空格隔开的地方，现在用换行符了。这里表达式的含义依然和上面一样，不再赘述。

上面两种不同写法都是配置角色的继承关系，配置完成后，接下来指定角色和资源的对应关系即可，如下：

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests().antMatchers("/admin/**")  
        .hasRole("admin")  
        .antMatchers("/db/**")  
        .hasRole("dba")  
        .antMatchers("/user/**")  
        .hasRole("user")  
        .and()  
        .formLogin()  
        .loginProcessingUrl("/doLogin")  
        .permitAll()  
        .and()  
        .csrf().disable();  
}
```

这个表示 /db/** 格式的路径需要具备 dba 角色才能访问，/admin/** 格式的路径则需要具备 admin 角色才能访问，/user/** 格式的路径，则需要具备 user 角色才能访问，此时提供相关接口，会发现，dba 除了访问 /db/**，也能访问 /admin/** 和 /user/**，admin 角色除了访问 /admin/**，也能访问 /user/**，user 角色则只能访问 /user/**。

4. 源码分析

这样两种不同的写法，其实也对应了两种不同的解析策略，角色继承关系的解析在 RoleHierarchyImpl 类的 buildRolesReachableInOneStepMap 方法中，Spring Boot2.0.8（含）之前该方法的源码如下：

```
private void buildRolesReachableInOneStepMap() {  
    Pattern pattern = Pattern.compile("(\\s*([^\s]+)\\s*)>\\s*(\\s*([^\s]+))");  
    Matcher roleHierarchyMatcher = pattern  
        .matcher(this.roleHierarchyStringRepresentation);  
    this.rolesReachableInOneStepMap = new HashMap<GrantedAuthority,  
    Set<GrantedAuthority>>();  
    while (roleHierarchyMatcher.find()) {  
        GrantedAuthority higherRole = new SimpleGrantedAuthority(  
            roleHierarchyMatcher.group(2));  
        GrantedAuthority lowerRole = new SimpleGrantedAuthority(  
            roleHierarchyMatcher.group(3));  
        Set<GrantedAuthority> rolesReachableInOneStepSet;  
        if (!this.rolesReachableInOneStepMap.containsKey(higherRole)) {  
            rolesReachableInOneStepSet = new HashSet<>();  
            this.rolesReachableInOneStepMap.put(higherRole,  
                rolesReachableInOneStepSet);  
        }  
        else {  
            rolesReachableInOneStepSet = this.rolesReachableInOneStepMap  
                .get(higherRole);  
        }  
        addReachableRoles(rolesReachableInOneStepSet, lowerRole);  
        logger.debug("buildRolesReachableInOneStepMap() - From role " +  
        higherRole  
            + " one can reach role " + lowerRole + " in one step.");  
    }
```

```
}
```

从这段源码中我们可以看到，角色的继承关系是通过正则表达式进行解析，通过空格进行切分，然后构建相应的 map 出来。

Spring Boot2.1.0 (含) 之后该方法的源码如下：

```
private void buildRolesReachableInOneStepMap() {
    this.rolesReachableInOneStepMap = new HashMap<GrantedAuthority,
Set<GrantedAuthority>>();
    try (BufferedReader bufferedReader = new BufferedReader(
        new StringReader(this.roleHierarchyStringRepresentation))) {
        for (String readLine; (readLine = bufferedReader.readLine()) != null;) {
            String[] roles = readLine.split(" > ");
            for (int i = 1; i < roles.length; i++) {
                GrantedAuthority higherRole = new SimpleGrantedAuthority(
                    roles[i - 1].replaceAll("^\\s+|\\s+$", ""));
                GrantedAuthority lowerRole = new
SimpleGrantedAuthority(roles[i].replaceAll("^\\s+|\\s+$",
Set<GrantedAuthority> rolesReachableInOneStepSet;
                if (!this.rolesReachableInOneStepMap.containsKey(higherRole)) {
                    rolesReachableInOneStepSet = new HashSet<GrantedAuthority>()
                );
                    this.rolesReachableInOneStepMap.put(higherRole,
rolesReachableInOneStepSet);
                } else {
                    rolesReachableInOneStepSet =
this.rolesReachableInOneStepMap.get(higherRole);
                }
                addReachableRoles(rolesReachableInOneStepSet, lowerRole);
                if (logger.isDebugEnabled()) {
                    logger.debug("buildRolesReachableInOneStepMap() - From role
" + higherRole
                        + " one can reach role " + lowerRole + " in one
step.");
                }
            }
        }
    } catch (IOException e) {
        throw new IllegalStateException(e);
    }
}
```

从这里我们可以看到，这里并没有一上来就是用正则表达式，而是先将角色继承字符串转为一个 BufferedReader，然后一行一行的读出来，再进行解析，最后再构建相应的 map。从这里我们可以看出为什么前后版本对此有不同的写法。

那么小伙伴在开发过程中，还是需要留意这一个差异。好了，角色继承我们就先说到这里。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

在前后端分离的项目中，登录策略也有不少，不过 JWT 算是目前比较流行的一种解决方案了，本文就和大家来分享一下如何将 Spring Security 和 JWT 结合在一起使用，进而实现前后端分离时的登录解决方案。

1 无状态登录

1.1 什么是有状态？

有状态服务，即服务端需要记录每次会话的客户端信息，从而识别客户端身份，根据用户身份进行请求的处理，典型的设计如 Tomcat 中的 Session。例如登录：用户登录后，我们把用户的信息保存在服务端 session 中，并且给用户一个 cookie 值，记录对应的 session，然后下次请求，用户携带 cookie 值来（这一步有浏览器自动完成），我们就能识别到对应 session，从而找到用户的信息。这种方式目前来看最方便，但是也有一些缺陷，如下：

- 服务端保存大量数据，增加服务端压力
- 服务端保存用户状态，不支持集群化部署

1.2 什么是无状态

微服务集群中的每个服务，对外提供的都使用 RESTful 风格的接口。而 RESTful 风格的一个最重要的规范就是：服务的无状态性，即：

- 服务端不保存任何客户端请求者信息
- 客户端的每次请求必须具备自描述信息，通过这些信息识别客户端身份

那么这种无状态性有哪些好处呢？

- 客户端请求不依赖服务端的信息，多次请求不需要必须访问到同一台服务器
- 服务端的集群和状态对客户端透明
- 服务端可以任意的迁移和伸缩（可以方便的进行集群化部署）
- 减小服务端存储压力

1.3 如何实现无状态

无状态登录的流程：

- 首先客户端发送账户名/密码到服务端进行认证
- 认证通过后，服务端将用户信息加密并且编码成一个 token，返回给客户端
- 以后客户端每次发送请求，都需要携带认证的 token
- 服务端对客户端发送来的 token 进行解密，判断是否有效，并且获取用户登录信息

1.4 JWT

1.4.1 简介

JWT，全称是 Json Web Token，是一种 JSON 风格的轻量级的授权和身份认证规范，可实现无状态、分布式的 Web 应用授权：

JWT.IO allows you to decode, verify and generate JWT.

LEARN MORE ABOUT JWT

JWT 作为一种规范，并没有和某一种语言绑定在一起，常用的 Java 实现是 GitHub 上的开源项目 jjwt，地址如下：<https://github.com/jwtk/jjwt>

1.4.2 JWT数据格式

JWT 包含三部分数据：

- Header：头部，通常头部有两部分信息：
 - 声明类型，这里是JWT
 - 加密算法，自定义

我们会对头部进行 Base64Url 编码（可解码），得到第一部分数据。

- Payload：载荷，就是有效数据，在官方文档中(RFC7519)，这里给了7个示例信息：
 - iss (issuer)：表示签发人
 - exp (expiration time)：表示token过期时间
 - sub (subject)：主题
 - aud (audience)：受众
 - nbf (Not Before)：生效时间
 - iat (Issued At)：签发时间
 - jti (JWT ID)：编号

这部分也会采用 Base64Url 编码，得到第二部分数据。

- Signature：签名，是整个数据的认证信息。一般根据前两步的数据，再加上服务的的密钥 secret（密钥保存在服务端，不能泄露给客户端），通过 Header 中配置的加密算法生成。用于验证整个数据完整和可靠性。

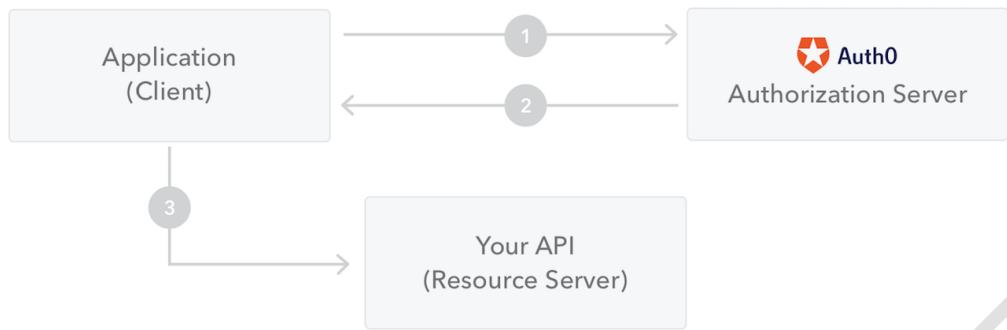
生成的数据格式如下图：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

注意，这里的数据通过 . 隔开成了三部分，分别对应前面提到的三部分，另外，这里数据是不换行的，图片换行只是为了展示方便而已。

1.4.3 JWT 交互流程

流程图：



步骤翻译：

1. 应用程序或客户端向授权服务器请求授权
2. 获取到授权后，授权服务器会向应用程序返回访问令牌
3. 应用程序使用访问令牌来访问受保护资源（如 API）

因为 JWT 签发的 token 中已经包含了用户的身份信息，并且每次请求都会携带，这样服务的就无需保存用户信息，甚至无需去数据库查询，这样就完全符合了 RESTful 的无状态规范。

1.5 JWT 存在的问题

说了这么多，JWT 也不是天衣无缝，由客户端维护登录状态带来的一些问题在这里依然存在，举例如下：

1. 续签问题，这是被很多人诟病的问题之一，传统的 cookie+session 的方案天然的支持续签，但是 jwt 由于服务端不保存用户状态，因此很难完美解决续签问题，如果引入 redis，虽然可以解决问题，但是 jwt 也变得不伦不类了。
2. 注销问题，由于服务端不再保存用户信息，所以一般可以通过修改 secret 来实现注销，服务端 secret 修改后，已经颁发的未过期的 token 就会认证失败，进而实现注销，不过毕竟没有传统的注销方便。
3. 密码重置，密码重置后，原本的 token 依然可以访问系统，这时候也需要强制修改 secret。
4. 基于第 2 点和第 3 点，一般建议不同用户取不同 secret。

2 实战

说了这么久，接下来我们就来看看这个东西到底要怎么用？

2.1 环境搭建

首先我们来创建一个 Spring Boot 项目，创建时需要添加 Spring Security 依赖，创建完成后，添加 `jwt` 依赖，完整的 `pom.xml` 文件如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

然后在项目中创建一个简单的 User 对象实现 UserDetails 接口，如下：

```
public class User implements UserDetails {
    private String username;
    private String password;
    private List<GrantedAuthority> authorities;
    public String getUsername() {
        return username;
    }
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
        return true;
    }
    //省略getter/setter
}
```

这个就是我们的用户对象，先放着备用，再创建一个 HelloController，内容如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello jwt !";
    }
    @GetMapping("/admin")
    public String admin() {
        return "hello admin !";
    }
}
```

HelloController 很简单，这里有两个接口，设计是 `/hello` 接口可以被具有 user 角色的用户访问，而 `/admin` 接口则可以被具有 admin 角色的用户访问。

2.2 JWT 过滤器配置

接下来提供两个和 JWT 相关的过滤器配置：

1. 一个是用户登录的过滤器，在用户的登录的过滤器中校验用户是否登录成功，如果登录成功，则生成一个token返回给客户端，登录失败则给前端一个登录失败的提示。
2. 第二个过滤器则是当其他请求发送来，校验token的过滤器，如果校验成功，就让请求继续执行。

这两个过滤器，我们分别来看，先看第一个：

```
public class JwtLoginFilter extends AbstractAuthenticationProcessingFilter {  
    protected JwtLoginFilter(String defaultFilterProcessesUrl,  
    AuthenticationManager authenticationManager) {  
        super(new AntPathRequestMatcher(defaultFilterProcessesUrl));  
        setAuthenticationManager(authenticationManager);  
    }  
    @Override  
    public Authentication attemptAuthentication(HttpServletRequest req,  
    HttpServletResponse resp) throws AuthenticationException, IOException,  
    ServletException {  
        User user = new ObjectMapper().readValue(req.getInputStream(),  
        User.class);  
        return getAuthenticationManager().authenticate(new  
        UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword()));  
    }  
    @Override  
    protected void successfulAuthentication(HttpServletRequest req,  
    HttpServletResponse resp, FilterChain chain, Authentication authResult) throws  
    IOException, ServletException {  
        Collection<? extends GrantedAuthority> authorities =  
        authResult.getAuthorities();  
        StringBuffer as = new StringBuffer();  
        for (GrantedAuthority authority : authorities) {  
            as.append(authority.getAuthority())  
            .append(",");  
        }  
        String jwt = Jwts.builder()  
            .claim("authorities", as)//配置用户角色  
            .setSubject(authResult.getName())  
            .setExpiration(new Date(System.currentTimeMillis() + 10 * 60 *  
            1000))  
            .signWith(SignatureAlgorithm.HS512, "sang@123")  
            .compact();  
        resp.setContentType("application/json;charset=utf-8");  
        PrintWriter out = resp.getWriter();  
        out.write(new ObjectMapper().writeValueAsString(jwt));  
        out.flush();  
        out.close();  
    }  
    protected void unsuccessfulAuthentication(HttpServletRequest req,  
    HttpServletResponse resp, AuthenticationException failed) throws IOException,  
    ServletException {  
        resp.setContentType("application/json;charset=utf-8");  
        PrintWriter out = resp.getWriter();  
    }  
}
```

```
        out.write("登录失败!");
        out.flush();
        out.close();
    }
}
```

关于这个类，我说如下几点：

1. 自定义 JwtLoginFilter 继承自 AbstractAuthenticationProcessingFilter，并实现其中的三个默认方法。
2. attemptAuthentication 方法中，我们从登录参数中提取出用户名密码，然后调用 AuthenticationManager.authenticate() 方法去进行自动校验。
3. 第二步如果校验成功，就会来到 successfulAuthentication 回调中，在 successfulAuthentication 方法中，将用户角色遍历然后用一个 `,` 连接起来，然后再利用 Jwts 去生成 token，按照代码的顺序，生成过程一共配置了四个参数，分别是用户角色、主题、过期时间以及加密算法和密钥，然后将生成的 token 写出到客户端。
4. 第二步如果校验失败就会来到 unsuccessfulAuthentication 方法中，在这个方法中返回一个错误提示给客户端即可。

再来看第二个 token 校验的过滤器：

```
public class JwtFilter extends GenericFilterBean {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) servletRequest;
        String jwtToken = req.getHeader("authorization");
        System.out.println(jwtToken);
        Claims claims =
Jwts.parser().setSigningKey("sang@123").parseClaimsJws(jwtToken.replace("Bearer"
, ""))
                .getBody();
        String username = claims.getSubject(); // 获取当前登录用户名
        List<GrantedAuthority> authorities =
AuthorityUtils.commaSeparatedStringToAuthorityList((String)
claims.get("authorities"));
        UsernamePasswordAuthenticationToken token = new
UsernamePasswordAuthenticationToken(username, null, authorities);
        SecurityContextHolder.getContext().setAuthentication(token);
        filterChain.doFilter(req, servletResponse);
    }
}
```

关于这个过滤器，我说如下几点：

1. 首先从请求头中提取出 authorization 字段，这个字段对应的 value 就是用户的 token。
2. 将提取出来的 token 字符串转换为一个 Claims 对象，再从 Claims 对象中提取出当前用户名和用户角色，创建一个 UsernamePasswordAuthenticationToken 放到当前的 Context 中，然后执行过滤链使请求继续执行下去。

如此之后，两个和 JWT 相关的过滤器就算配置好了。

2.3 Spring Security 配置

接下来我们来配置 Spring Security，如下：

```

@Configuration
public class webSecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean
    PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().withUser("admin")
            .password("123").roles("admin")
            .and()
            .withUser("sang")
            .password("456")
            .roles("user");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/hello").hasRole("user")
            .antMatchers("/admin").hasRole("admin")
            .antMatchers(HttpMethod.POST, "/login").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilterBefore(new
JwtLoginFilter("/login", authenticationManager()), UsernamePasswordAuthenticationFilter.class)
            .addFilterBefore(new
JwtFilter(), UsernamePasswordAuthenticationFilter.class)
            .csrf().disable();
    }
}

```

1. 简单起见，这里我并未对密码进行加密，因此配置了 NoOpPasswordEncoder 的实例。
2. 简单起见，这里并未连接数据库，我直接在内存中配置了两个用户，两个用户具备不同的角色。
3. 配置路径规则时，/hello 接口必须要具备 user 角色才能访问，/admin 接口必须要具备 admin 角色才能访问，POST 请求并且是 /login 接口则可以直接通过，其他接口必须认证后才能访问。
4. 最后配置上两个自定义的过滤器并且关闭掉 csrf 保护。

2.4 测试

做完这些之后，我们的环境就算完全搭建起来了，接下来启动项目然后在 POSTMAN 中进行测试，如下：

The screenshot shows a POST request to `http://127.0.0.1:8080/login`. The Body tab is selected, showing a JSON payload: `{"username": "sang", "password": "456"}`. The response body is a long JWT token: `eyJhbGciOiJIUzUxMiJ9.eyJhdXRob3JpdGlycyI6IiJPTEVfdXNlcjIzdWliOijzYW5nIiwizXhwIjoxNTU0MTIzMjg3fQ.A2ID8p3d3RCQk1cN0Y77pOs3jTYencbxtcOvp7B75aTXxmxzjQxGzTi_QD8Gou9d1esfHr1_MAIbq5rSfEY9ua`.

登录成功后返回的字符串就是经过 base64url 转码的 token，一共有三部分，通过一个 `.` 隔开，我们可以对第一个 `.` 之前的字符串进行解码，即 Header，如下：

The token structure is shown as three parts separated by dots. The first part is the Header: `{"alg": "HS512"}`. It has `BASE64编码 >` and `< BASE64解码` buttons.

再对两个 `.` 之间的字符串解码，即 payload：

The second part is the Payload: `{"authorities": "ROLE_user", "sub": "sang", "exp": 1554123287}`. It also has `BASE64编码 >` and `< BASE64解码` buttons.

可以看到，我们设置信息，由于 base64 并不是加密方案，只是一种编码方案，因此，不建议将敏感的用户信息放到 token 中。

接下来再去访问 `/hello` 接口，注意认证方式选择 Bearer Token，Token 值为刚刚获取到的值，如下：

The screenshot shows a GET request to `http://127.0.0.1:8080/hello`. The Authorization tab is selected, showing `Bearer eyJhbGciOiJIUzUxMiJ9.eyJhdXRob3JpdGlycyI6IiJPTEVfdXNlcjIzdWliOijzYW5nIiwizXhwIjoxNTU0MTIzMjg3fQ.A2ID8p3d3RCQk1cN0Y77pOs3jTYencbxtcOvp7B75aTXxmxzjQxGzTi_QD8Gou9d1esfHr1_MAIbq5rSfEY9ua`. The response body is `hello jwt !`.

可以看到，访问成功。

总结

这就是 JWT 结合 Spring Security 的一个简单用法，讲真，如果实例允许，类似的需求我还是推荐使用 OAuth2 中的 password 模式。

不知道大伙有没有看懂呢？如果没看懂，松哥还有一个关于这个知识点的视频教程，如下：

- [Spring Security + JWT 视频教程](#)

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



关于 Spring Security，松哥之前发过多篇文章和大家聊聊这个安全框架的使用：

1. [手把手带你入门 Spring Security!](#)
2. [Spring Security 登录添加验证码](#)
3. [SpringSecurity 登录使用 JSON 格式数据](#)
4. [Spring Security 中的角色继承问题](#)
5. [Spring Security 中使用 JWT!](#)
6. [Spring Security 结合 OAuth2](#)

不过，今天要和小伙伴们聊一聊 Spring Security 中的另外一个问题，那就是在 Spring Security 中未获认证的请求默认会重定向到登录页，但是在前后端分离的登录中，这个默认行为则显得非常不合适，今天我们主要来看看如何实现未获认证的请求直接返回 JSON，而不是重定向到登录页面。

前置知识

这里关于 Spring Security 的基本用法我就不再赘述了，如果小伙伴们不了解，可以参考上面的 6 篇文章。

大家知道，在自定义 Spring Security 配置的时候，有这样几个属性：

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .anyRequest().authenticated()  
        .formLogin()  
        .loginProcessingUrl("/doLogin")  
        .LoginPage("/login")  
        //其他配置  
        .permitAll()  
        .and()  
        .csrf().disable();  
}
```

这里有两个比较重要的属性：

- loginProcessingUrl：这个表示配置处理登录请求的接口地址，例如你是表单登录，那么 form 表单中 action 的值就是这里填的值。
- loginPage：这个表示登录页的地址，例如当你访问一个需要登录后才能访问的资源时，系统就会自动给你通过重定向跳转到这个页面上来。

这种配置在前后端不分的登录中是没有问题的，在前后端分离的登录中，这种配置就有问题了。我举个简单的例子，例如我想访问 /hello 接口，但是这个接口需要登录之后才能访问，我现在没有登录就直接去访问这个接口了，那么系统会给我返回 302，让我去登录页面，在前后端分离中，我的后端一般是没有登录页面的，就是一个提示 JSON，例如下面这样：

```
@GetMapping("/login")  
public RespBean login() {  
    return RespBean.error("尚未登录，请登录!");  
}
```

完整代码大家可以参考我的微人事项目。

也就是说，当我没有登录直接去访问 `/hello` 这个接口的时候，我会看到上面这段 JSON 字符串。在前后端分离开发中，这个看起来没问题（后端不再做页面跳转，无论发生什么都是返回 JSON）。但是问题就出在这里，系统默认的跳转是一个重定向，就是说当你访问 `/hello` 的时候，服务端会给浏览器返回 302，同时响应头中有一个 `Location` 字段，它的值为 `http://localhost:8081/login`，也就是告诉浏览器你去访问 `http://localhost:8081/login` 地址吧。浏览器收到指令之后，就会直接去访问 `http://localhost:8081/login` 地址，如果此时是开发环境并且请求还是 Ajax 请求，就会发生跨域。因为前后端分离开发中，前端我们一般在 NodeJS 上启动，然后前端的所有请求通过 NodeJS 做请求转发，现在服务端直接把请求地址告诉浏览器了，浏览器就会直接去访问 `http://localhost:8081/login` 了，而不会做请求转发了，因此就发生了跨域问题。

解决方案

很明显，上面的问题我们不能用跨域的思路来解决，虽然这种方式看起来也能解决问题，但不是最佳方案。

如果我们的 Spring Security 在用户未获认证的时候去请求一个需要认证后才能请求的数据，此时不给用户重定向，而是直接就返回一个 JSON，告诉用户这个请求需要认证之后才能发起，就不会有上面的事情了。

这里就涉及到 Spring Security 中的一个接口 `AuthenticationEntryPoint`，该接口有一个实现类：`LoginUrlAuthenticationEntryPoint`，该类中有一个方法 `commence`，如下：

```
/**  
 * Performs the redirect (or forward) to the login form URL.  
 */  
public void commence(HttpServletRequest request, HttpServletResponse response,  
        AuthenticationException authException) {  
    String redirectUrl = null;  
    if (useForward) {  
        if (forceHttps && "http".equals(request.getScheme())) {  
            redirectUrl = buildHttpsRedirectUrlForRequest(request);  
        }  
        if (redirectUrl == null) {  
            String loginForm = determineUrlToUseForThisRequest(  
                request,  
                response,  
                authException);  
            if (logger.isDebugEnabled()) {  
                logger.debug("Server side forward to: " + loginForm);  
            }  
            RequestDispatcher dispatcher =  
                request.getRequestDispatcher(loginForm);  
            dispatcher.forward(request, response);  
            return;  
        }  
    }  
    else {  
        redirectUrl = buildRedirectUrlLoginPage(request, response,  
            authException);  
    }  
    redirectStrategy.sendRedirect(request, response, redirectUrl);  
}
```

首先我们从这个方法的注释中就可以看出，这个方法是用来决定到底是要重定向还是要 forward，通过 Debug 追踪，我们发现默认情况下 `useForward` 的值为 `false`，所以请求走进了重定向。

那么我们解决问题的思路很简单，直接重写这个方法，在方法中返回 JSON 即可，不再做重定向操作，具体配置如下：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .anyRequest().authenticated()
        .formLogin()
        .loginProcessingUrl("/doLogin")
        .LoginPage("/login")
    //其他配置
        .permitAll()
        .and()
        .csrf().disable().exceptionHandling()
            .authenticationEntryPoint(new AuthenticationEntryPoint() {
@Override
public void commence(HttpServletRequest req, HttpServletResponse resp, AuthenticationException authException) throws IOException, ServletException {
    resp.setContentType("application/json; charset=utf-8");
    PrintWriter out = resp.getWriter();
    RespBean respBean = RespBean.error("访问失败!");
    if (authException instanceof
InsufficientAuthenticationException) {
        respBean.setMsg("请求失败，请联系管理员!");
    }
    out.write(new ObjectMapper().writeValueAsString(respBean));
    out.flush();
    out.close();
}
});
}
}
```

在 Spring Security 的配置中加上自定义的 `AuthenticationEntryPoint` 处理方法，该方法中直接返回相应的 JSON 提示即可。这样，如果用户再去直接访问一个需要认证之后才可以访问的请求，就不会发生重定向操作了，服务端会直接给浏览器一个 JSON 提示，浏览器收到 JSON 之后，该干嘛干嘛。

结语

好了，一个小小的重定向问题和小伙伴们分享下，不知道大家有没有看懂呢？这也是我最近在重构微人事的时候遇到的问题。预计 12 月份，微人事的 Spring Boot 版本会升级到目前最新版，请小伙伴们留意哦。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

回顾热部署

Spring Boot 中的热部署相信大家都用过吧，只需要添加 `spring-boot-devtools` 依赖就可以轻松实现热部署。Spring Boot 中热部署最最关键的原理就是两个不同的 classloader：

- base classloader
- restart classloader

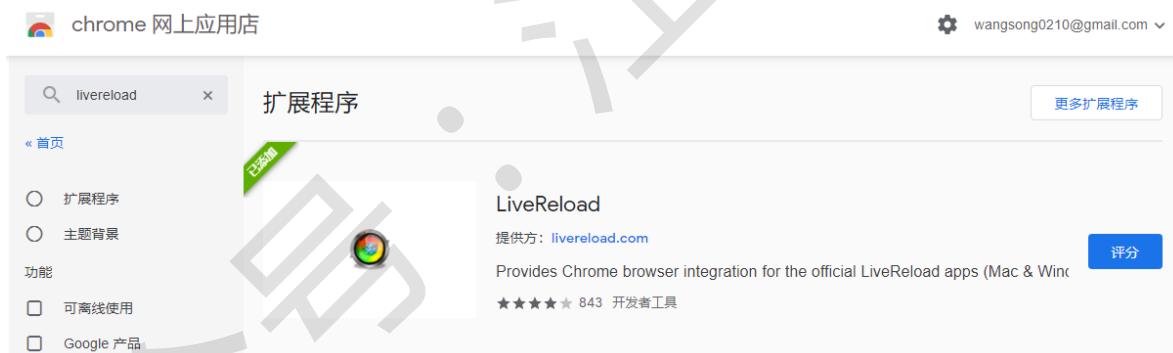
其中 base classloader 用来加载那些不会变化的类，例如各种第三方依赖，而 restart classloader 则用来加载那些会发生变化的类，例如你自己写的代码。Spring Boot 中热部署的原理就是当代码发生变化时，base classloader 不变，而 restart classloader 则会被废弃，被另一个新的 restart classloader 代替。在整个过程中，因为只重新加载了变化的类，所以启动速度要被重启快。

但是有另外一个问题，就是静态资源文件！使用 devtools，默认情况下当静态资源发生变化时，并不会触发项目重启。虽然我们可以通过配置解决这一问题，但是没有必要！因为静态资源文件发生变化后不需要编译，按理说保存后刷新下就可以访问到了。

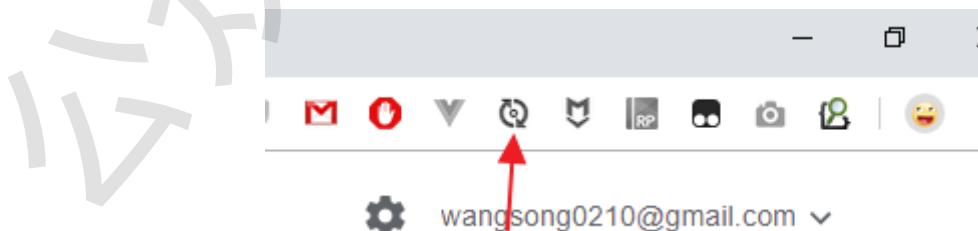
那么如何才能实现静态资源变化后，不编译就能自动刷新呢？LiveReload 可以帮助我们实现这一功能！

LiveReload

devtools 中默认嵌入了 LiveReload 服务器，利用 LiveReload 可以实现静态文件的热部署，LiveReload 可以在资源发生变化时自动触发浏览器更新，LiveReload 支持 Chrome、Firefox 以及 Safari。以 Chrome 为例，在 Chrome 应用商店搜索 LiveReload，结果如下图：



将第一个搜索结果添加到 Chrome 中，添加成功后，在 Chrome 右上角有一个 LiveReload 图标



在浏览器中打开项目的页面，然后点击浏览器右上角的 LiveReload 按钮，打开 LiveReload 连接。

注意：

LiveReload 是和浏览器选项卡绑定在一起的，在哪个选项卡中打开了 LiveReload，就在哪个选项卡中访问页面，这样才有效果。

打开 LiveReload 之后，我们启动一个加了 devtools 依赖的 Spring Boot 项目：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

此时随便在 resources/static 目录下添加一个静态 html 页面，然后启动 Spring Boot 项目，在**打开了 LiveReload 的选项卡中访问 html 页面**。

访问成功后，我们再去手动修改 html 页面代码，修改成功后，回到浏览器，不用做任何操作，就会发现浏览器自动刷新了，页面已经更新了。

整个过程中，我的 Spring Boot 项目并没有重启。

如果开发者安装并且启动了 LiveReload 插件，同时也添加了 devtools 依赖，但是却并不想当静态页面发生变化时浏览器自动刷新，那么可以在 application.properties 中添加如下代码进行配置：

```
spring.devtools.livereload.enabled=false
```

最佳实践

建议开发者使用 LiveReload 策略而不是项目重启策略来实现静态资源的动态加载，因为项目重启所耗费时间一般来说要超过使用LiveReload 所耗费的时间。

Firefox 也可以安装 LiveReload 插件，装好之后和 Chrome 用法基本一致，这里不再赘述。

如果小伙伴们们的 Chrome 商店使用不便，可以在松哥公众号后台回复 livereload，松哥有下载好的离线安装包。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



前两天被人问到这样一个问题：

“松哥，为什么我的 Spring Boot 项目打包成的 jar，被其他项目依赖之后，总是报找不到类的错误？”

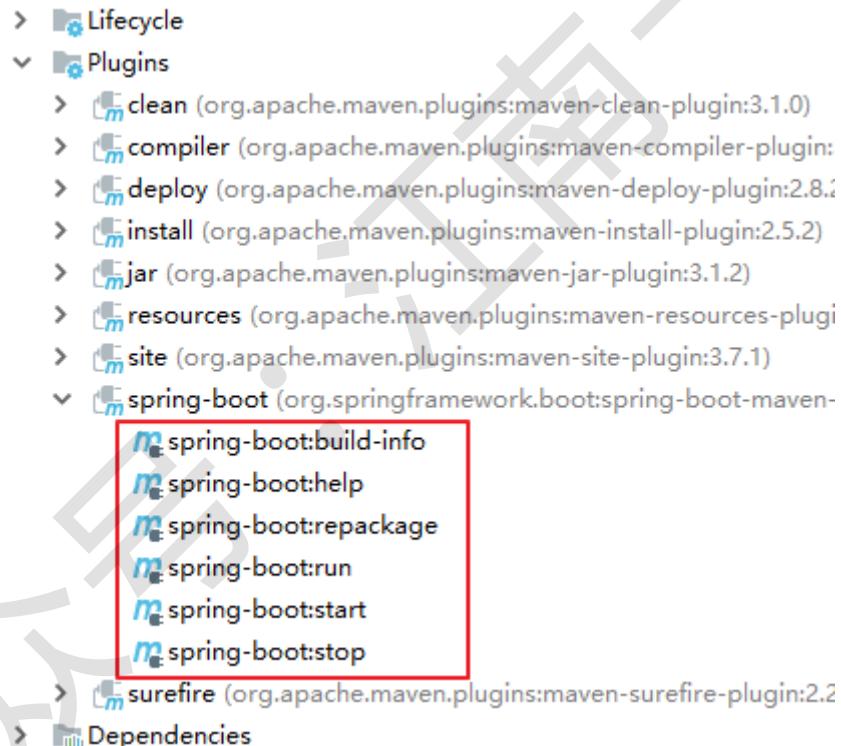
大伙有这样的疑问，就是因为还没搞清楚可执行 jar 和普通 jar 到底有什么区别？今天松哥就和大家来聊一聊这个问题。

多了一个插件

Spring Boot 中默认打包成的 jar 叫做 可执行 jar，这种 jar 不同于普通的 jar，普通的 jar 不可以通过 `java -jar xxx.jar` 命令执行，普通的 jar 主要是被其他应用依赖，Spring Boot 打成的 jar 可以执行，但是不可以被其他的应用所依赖，即使强制依赖，也无法获取里边的类。但是可执行 jar 并不是 Spring Boot 独有的，Java 工程本身就可以打包成可执行 jar。

有的小伙伴可能就有疑问了，既然同样是执行 `mvn package` 命令进行项目打包，为什么 Spring Boot 项目就打成了可执行 jar，而普通项目则打包成了不可执行 jar 呢？

这我们就不得不提 Spring Boot 项目中一个默认的插件配置 `spring-boot-maven-plugin`，这个打包插件存在 5 个方面的功能，从插件命令就可以看出：



五个功能分别是：

- build-info：生成项目的构建信息文件 `build-info.properties`
- repackage：这个是默认 goal，在 `mvn package` 执行之后，这个命令再次打包生成可执行的 jar，同时将 `mvn package` 生成的 jar 重命名为 `*.origin`
- run：这个可以用来运行 Spring Boot 应用
- start：这个在 `mvn integration-test` 阶段，进行 Spring Boot 应用生命周期的管理
- stop：这个在 `mvn integration-test` 阶段，进行 Spring Boot 应用生命周期的管理

这里功能，默认情况下使用就是 repackage 功能，其他功能要使用，则需要开发者显式配置。

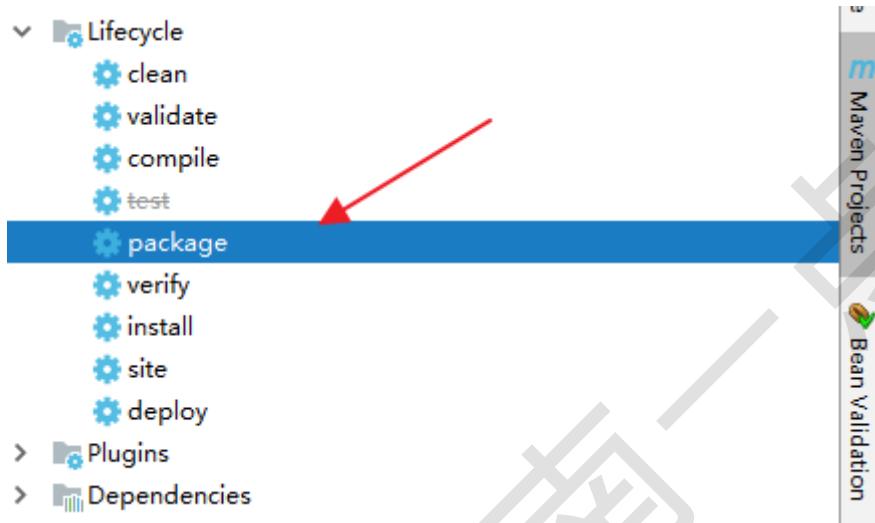
打包

repackage 功能的作用，就是在打包的时候，多做一点额外的事情：

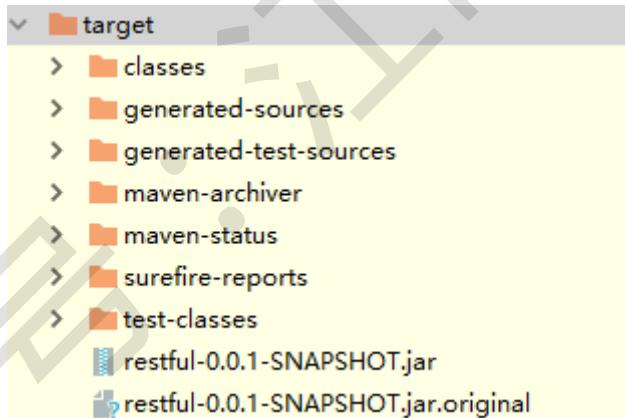
- 首先 `mvn package` 命令对项目进行打包，打成一个 `jar`，这个 `jar` 就是一个普通的 `jar`，可以被其他项目依赖，但是不可以被执行
- `repackage` 命令，对第一步 打包成的 `jar` 进行再次打包，将之打成一个可执行 `jar`，通过将第一步打成的 `jar` 重命名为 `*.original` 文件

举个例子：

对任意一个 Spring Boot 项目进行打包，可以执行 `mvn package` 命令，也可以直接在 IDEA 中点击 `package`，如下：



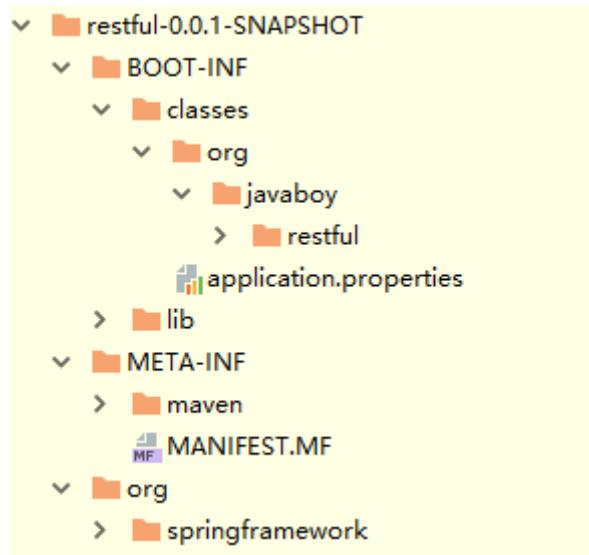
打包成功之后，`target` 中的文件如下：



这里有两个文件，第一个 `restful-0.0.1-SNAPSHOT.jar` 表示打包成的可执行 `jar`，第二个 `restful-0.0.1-SNAPSHOT.jar.original` 则是在打包过程中，被重命名的 `jar`，这是一个不可执行 `jar`，但是可以被其他项目依赖的 `jar`。通过对这两个文件的解压，我们可以看出这两者之间的差异。

两种 jar 的比较

可执行 `jar` 解压之后，目录如下：



可以看到，可执行 jar 中，我们自己的代码是存在于 `BOOT-INF/classes/` 目录下，另外，还有一个 `META-INF` 的目录，该目录下有一个 `MANIFEST.MF` 文件，打开该文件，内容如下：

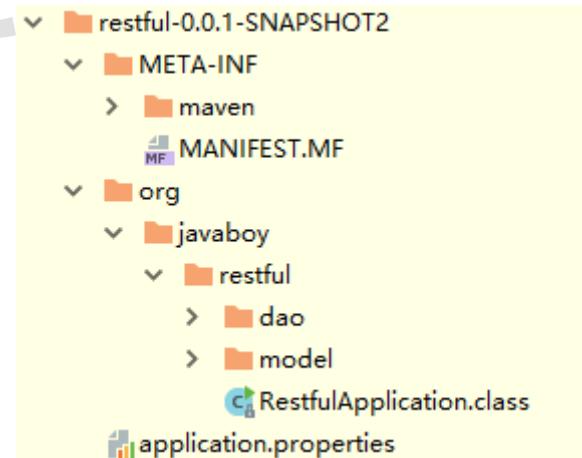
```
Manifest-Version: 1.0
Implementation-Title: restful
Implementation-Version: 0.0.1-SNAPSHOT
Start-Class: org.javaboy.restful.RestfulApplication
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Build-Jdk-Spec: 1.8
Spring-Boot-Version: 2.1.6.RELEASE
Created-By: Maven Archiver 3.4.0
Main-Class: org.springframework.boot.loader.JarLauncher
```

可以看到，这里定义了一个 `start-class`，这就是可执行 jar 的入口类，`spring-boot-classes` 表示我们自己代码编译后的位置，`Spring-Boot-Lib` 则表示项目依赖的 jar 的位置。

换句话说，如果自己要打一个可执行 jar 包的话，除了添加相关依赖之外，还需要配置 `META-INF/MANIFEST.MF` 文件。

这是可执行 jar 的结构，那么不可执行 jar 的结构呢？

我们首先将默认的后缀 `.original` 除去，然后给文件重命名，重命名完成，进行解压：



解压后可以看到，不可执行 jar 根目录就相当于我们的 `classpath`，解压之后，直接就能看到我们的代码，它也有 `META-INF/MANIFEST.MF` 文件，但是文件中没有定义启动类等。

```
Manifest-Version: 1.0
Implementation-Title: restful
Implementation-Version: 0.0.1-SNAPSHOT
Build-Jdk-Spec: 1.8
Created-By: Maven Archiver 3.4.0
```

注意

这个不可以执行 `jar` 也没有将项目的依赖打包进来。

从这里我们就可以看出，两个 `jar`，虽然都是 `jar` 包，但是内部结构是完全不同的，因此一个可以直接执行，另一个则可以被其他项目依赖。

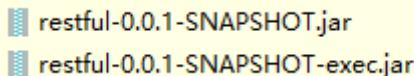
一次打包两个 jar

一般来说，Spring Boot 直接打包成可执行 `jar` 就可以了，不建议将 Spring Boot 作为普通的 `jar` 被其他的项目所依赖。如果有这种需求，建议将被依赖的部分，单独抽出来做一个普通的 Maven 项目，然后在 Spring Boot 中引用这个 Maven 项目。

如果非要将 Spring Boot 打包成一个普通 `jar` 被其他项目依赖，技术上来说，也是可以的，给 `spring-boot-maven-plugin` 插件添加如下配置：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <classifier>exec</classifier>
      </configuration>
    </plugin>
  </plugins>
</build>
```

配置的 `classifier` 表示可执行 `jar` 的名字，配置了这个之后，在插件执行 `repackage` 命令时，就不会给 `mvn package` 所打成的 `jar` 重命名了，所以，打包后的 `jar` 如下：



`restful-0.0.1-SNAPSHOT.jar`
`restful-0.0.1-SNAPSHOT-exec.jar`

第一个 `jar` 表示可以被其他项目依赖的 `jar`，第二个 `jar` 则表示一个可执行 `jar`。

好了，关于 Spring Boot 中 `jar` 的问题，我们就说这么多，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 `2TB`，获取超 `2TB` 免费 Java 学习资源。



云·空间·时间

不知道各位小伙伴在生产环境都是怎么部署 Spring Boot 的，打成 jar 直接一键运行？打成 war 扔到 Tomcat 容器中运行？不过据松哥了解，容器化部署应该是目前的主流方案。

不同于传统的单体应用，微服务由于服务数量众多，在部署的时候出问题的可能性更大，这个时候，结合 Docker 来部署，就可以很好的解决这个问题，这也是目前使用较多的方案之一。

将 Spring Boot 项目打包到 Docker 容器中部署，有很多不同的方法，今天松哥主要来和大家聊一聊如何将 Spring Boot 项目一键打包到远程 Docker 容器，然后通过运行一个镜像的方式来启动一个 Spring Boot 项目。

至于其他的 Spring Boot 结合 Docker 的用法，大家不要着急，后续的文章，松哥会和大家慢慢的——道来。

1.准备工作

1.1 准备 Docker

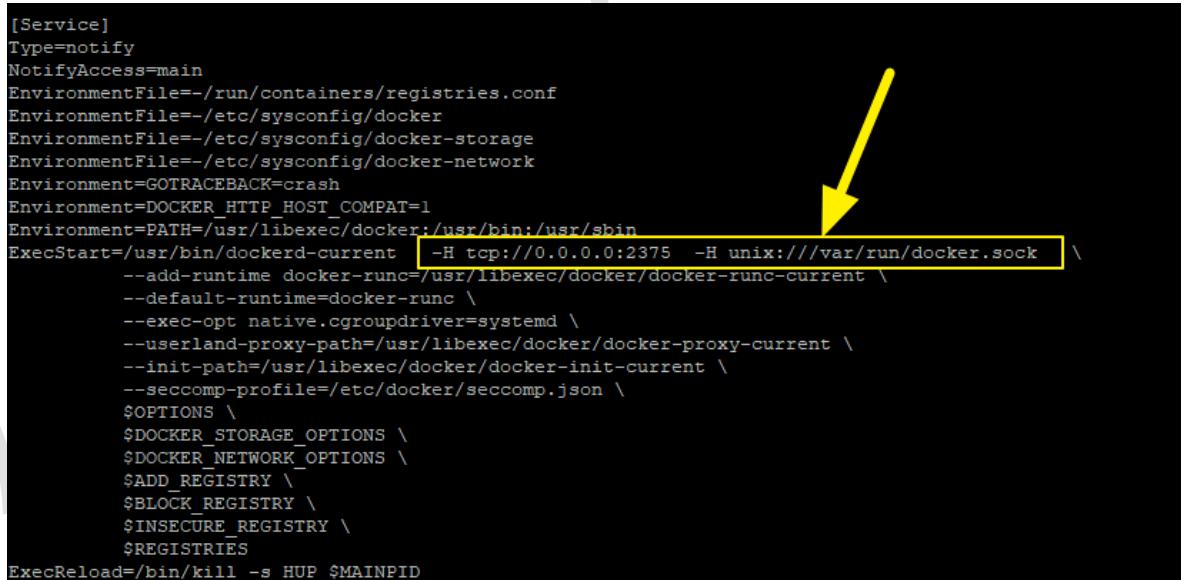
我这里以 CentOS7 为例来给大家演示。

首先需要在 CentOS7 上安装好 Docker，这个安装方式网上很多，我就不多说了，我自己去年写过一个 Docker 入门教程，大家可以在公众号后台回复 `Docker` 获取教程下载地址。

Docker 安装成功之后，我们首先需要修改 Docker 配置，开启允许远程访问 Docker 的功能，开启方式很简单，修改 `/usr/lib/systemd/system/docker.service` 文件，加入如下内容：

```
-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock
```

如下图：



```
[Service]
Type=notify
NotifyAccess=main
EnvironmentFile=-/run/containers/registries.conf
EnvironmentFile=-/etc/sysconfig/docker
EnvironmentFile=-/etc/sysconfig/docker-storage
EnvironmentFile=-/etc/sysconfig/docker-network
Environment=GOTRACEBACK=crash
Environment=DOCKER_HTTP_HOST_COMPAT=1
Environment=PATH=/usr/libexec/docker:/usr/bin:/usr/sbin
ExecStart=/usr/bin/dockerd-current -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock \
--add-runtime docker-runc=/usr/libexec/docker/docker-runc-current \
--default-runtime=docker-runc \
--exec-opt native.cgroupdriver=systemd \
--userland-proxy-path=/usr/libexec/docker/docker-proxy-current \
--init-path=/usr/libexec/docker/docker-init-current \
--seccomp-profile=/etc/docker/seccomp.json \
$OPTIONS \
$DOCKER_STORAGE_OPTIONS \
$DOCKER_NETWORK_OPTIONS \
$ADD_REGISTRY \
$BLOCK_REGISTRY \
$INSECURE_REGISTRY \
$REGISTRIES
ExecReload=/bin/kill -s HUP $MAINPID
```

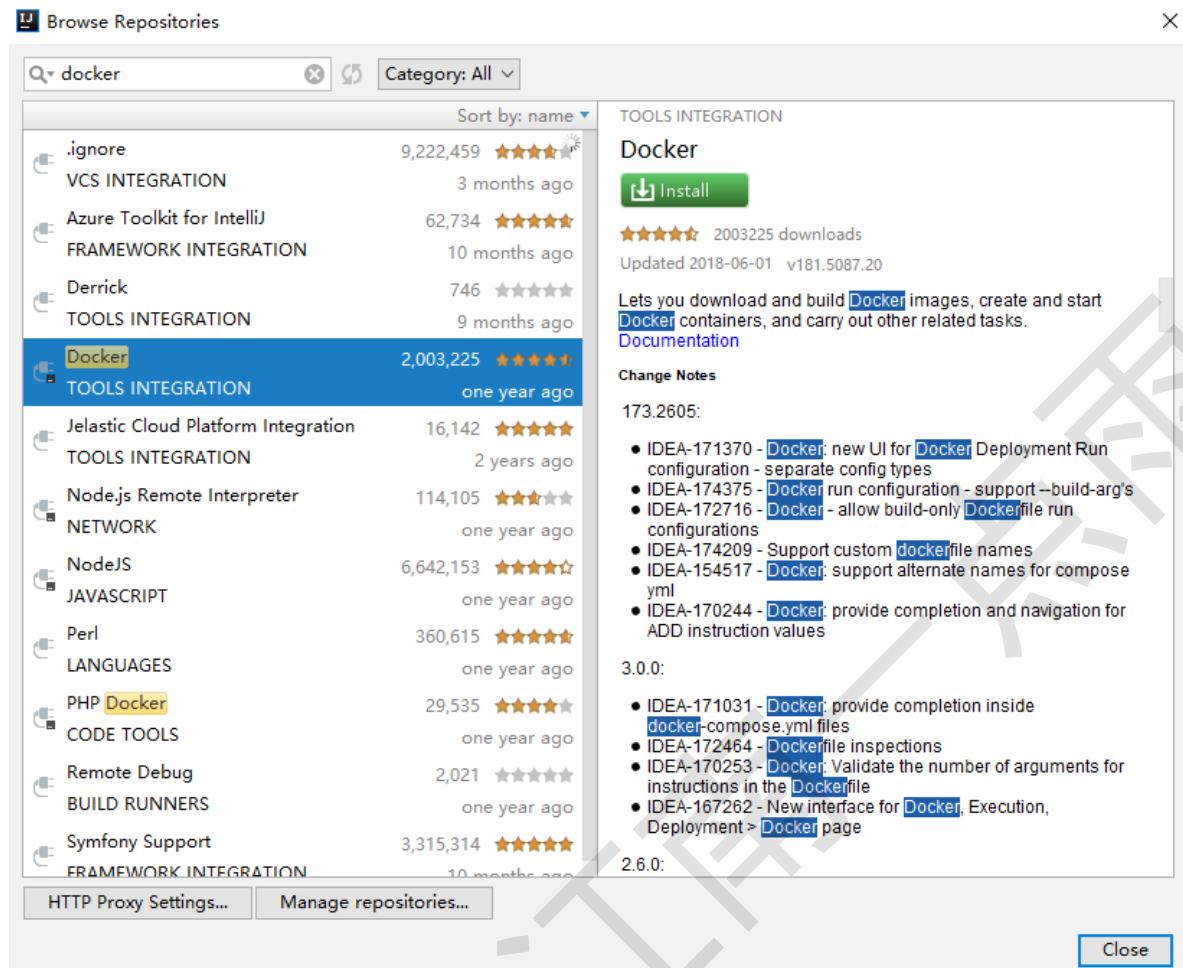
配置完成后，保存退出，然后重启 Docker：

```
systemctl daemon-reload
service docker restart
```

Docker 重启成功之后，Docker 的准备工作就算是 OK 了。

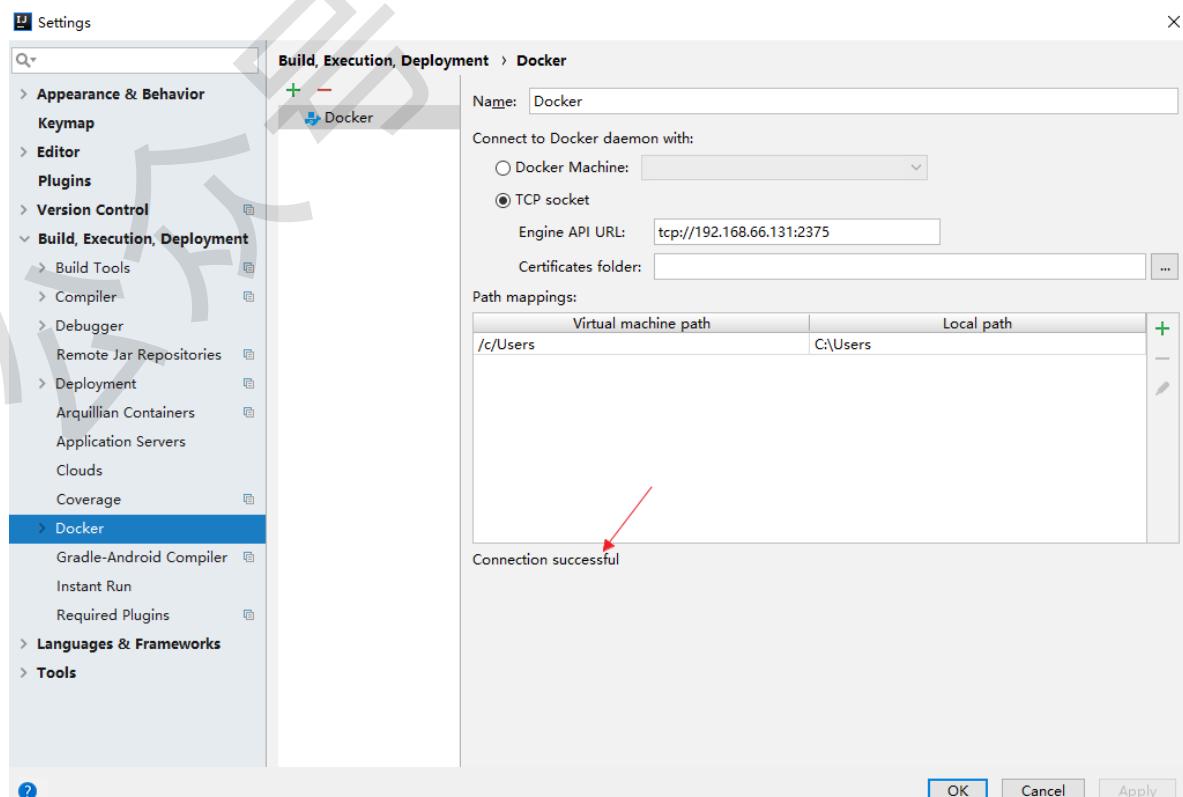
1.2 准备 IDEA

IDEA 上的准备工作，主要是安装一个 Docker 插件，点击 `File->Settings->Plugins->Browse Repositories` 如下：



点击右边绿色的 `Install` 按钮，完成安装，安装完成之后需要重启一下 IDEA。

IDEA 重启成功之后，我们依次打开 `File->Settings->Build, Execution, Deployment->Docker`，然后配置一下 Docker 的远程连接地址：



配置一下 Docker 的地址，配置完成后，可以看到下面有一个 Connection successful 提示，这个表示 Docker 已经连接上了。

如此之后，我们的准备工作就算是 OK 了。

2.准备项目

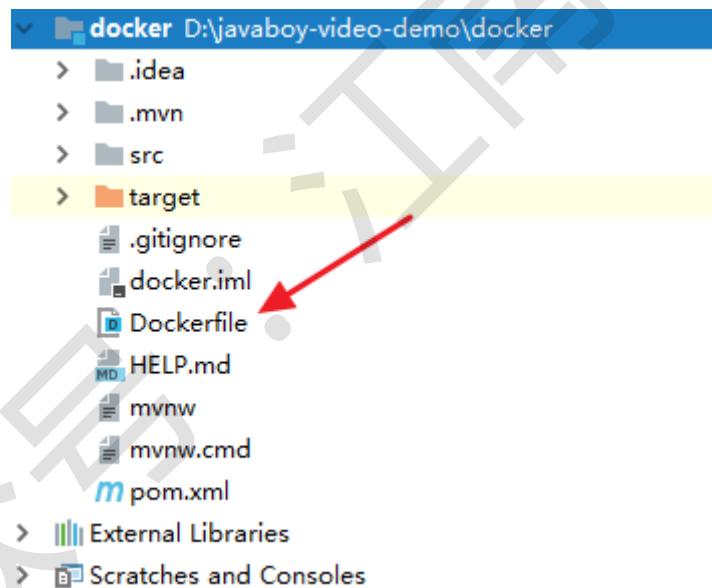
接下来我们来创建一个简单的 Spring Boot 项目（只需要引入 `spring-boot-starter-web` 依赖即可），项目创建成功之后，我们再创建一个普通的 `HelloDockerController`，用来自做测试，如下：

```
@RestController
public class HelloDockerController {
    @GetMapping("/hello")
    public String hello() {
        return "hello docker!";
    }
}
```

这是一个很简单的接口，无需多说。

3.配置 Dockerfile

接下来，在项目的根目录下，我创建一个 Dockerfile，作为我镜像的构建文件，具体位置如下图：



文件内容如下：

```
FROM hub.c.163.com/library/java:latest
VOLUME /tmp
ADD target/docker-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

这里只有简单的四行，我说一下：

1. Spring Boot 项目的运行依赖 Java 环境，所以我自己的镜像基于 Java 镜像来构建。
2. 考虑到 Docker 官方镜像下载较慢，我这里使用了网易提供的 Docker 镜像。
3. 由于 Spring Boot 运行时需要 tmp 目录，这里数据卷配置一个 /tmp 目录出来。
4. 将本地 target 目录中打包好的 .jar 文件复制一份新的到 /app.jar。

5. 最后就是配置一下启动命令，由于我打包的 jar 已经成为 app.jar 了，所以启动命令也是启动 app.jar。

这是我们配置的一个简单的 Dockerfile。

4. 配置 Maven 插件

接下来在 pom.xml 文件中，添加如下插件：

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>1.2.0</version>
    <executions>
        <execution>
            <id>build-image</id>
            <phase>package</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <dockerHost>http://192.168.66.131:2375</dockerHost>
        <imageName>javaboy/${project.artifactId}</imageName>
        <imageTags>
            <imageTag>${project.version}</imageTag>
        </imageTags>
        <forceTags>true</forceTags>
        <dockerDirectory>${project.basedir}</dockerDirectory>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

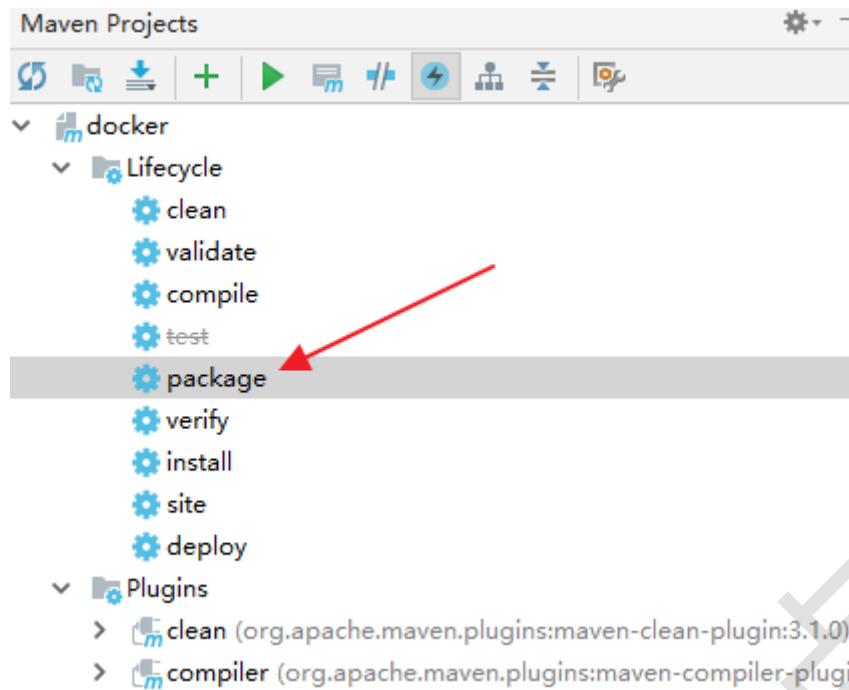
这个插件的配置不难理解：

1. 首先在 execution 节点中配置当执行 mvn package 的时候，顺便也执行一下 docker:build
2. 然后在 configuration 中分别配置 Docker 的主机地址，镜像的名称，镜像的 tags，其中 dockerDirectory 表示指定 Dockerfile 的位置。
3. 最后 resource 节点中再配置一下 jar 的位置和名称即可。

OK，做完这些我们就算大功告成了。

5. 打包运行

接下来对项目进行打包，打包完成后，项目会自动构建成一个镜像，并且上传到 Docker 容器中，打包方式如下：



打包过程会稍微有一点久，因为还包含了镜像的构建，特别是第一次打包，需要下载基础镜像，会更慢一些。

部分打包日志如下（项目构建过程）：

```
Step 1/4 : FROM hub.c.163.com/library/java:latest
--> a001fc27db5a
Step 2/4 : VOLUME /tmp
--> Running in d3c28e43bc92
--> 38ec33dc720e
Removing intermediate container d3c28e43bc92
Step 3/4 : ADD target/docker-0.0.1-SNAPSHOT.jar app.jar
--> e23228f3d13c
Removing intermediate container d9ef74b03e1f
Step 4/4 : ENTRYPOINT java -jar /app.jar
--> Running in 093a2ec8c3c0
--> 7b11baf61d21
Removing intermediate container 093a2ec8c3c0
Successfully built 7b11baf61d21
[INFO] Built javaboy/docker
[INFO] Tagging javaboy/docker with 0.0.1
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 01:27 min
[INFO] Finished at: 2019-08-26T22:39:46+08:00
[INFO] Final Memory: 40M/381M
[INFO]
```

项目打包成功之后，我们就可以在 Docker 容器中看到我们刚刚打包成的镜像了，如下：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
javaboy/docker	0.0.1	7b11baf61d21	5 minutes ago	660 MB
javaboy/docker	latest	7b11baf61d21	5 minutes ago	660 MB
mynginx99	latest	d295887b8669	9 days ago	109 MB

5.1 运行方式一

此时，我们可以直接在 Linux 上像创建普通容器一样创建这个镜像的容器，然后启动，执行如下命令即可：

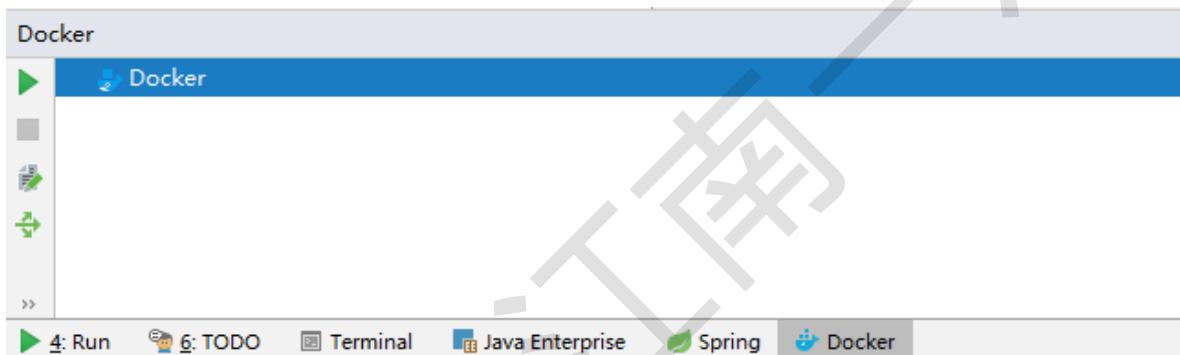
```
docker run -d --name javaboy -p 8080:8080 javaboy/docker:0.0.1
```

启动成功之后，我们就可以访问容器中的接口了。

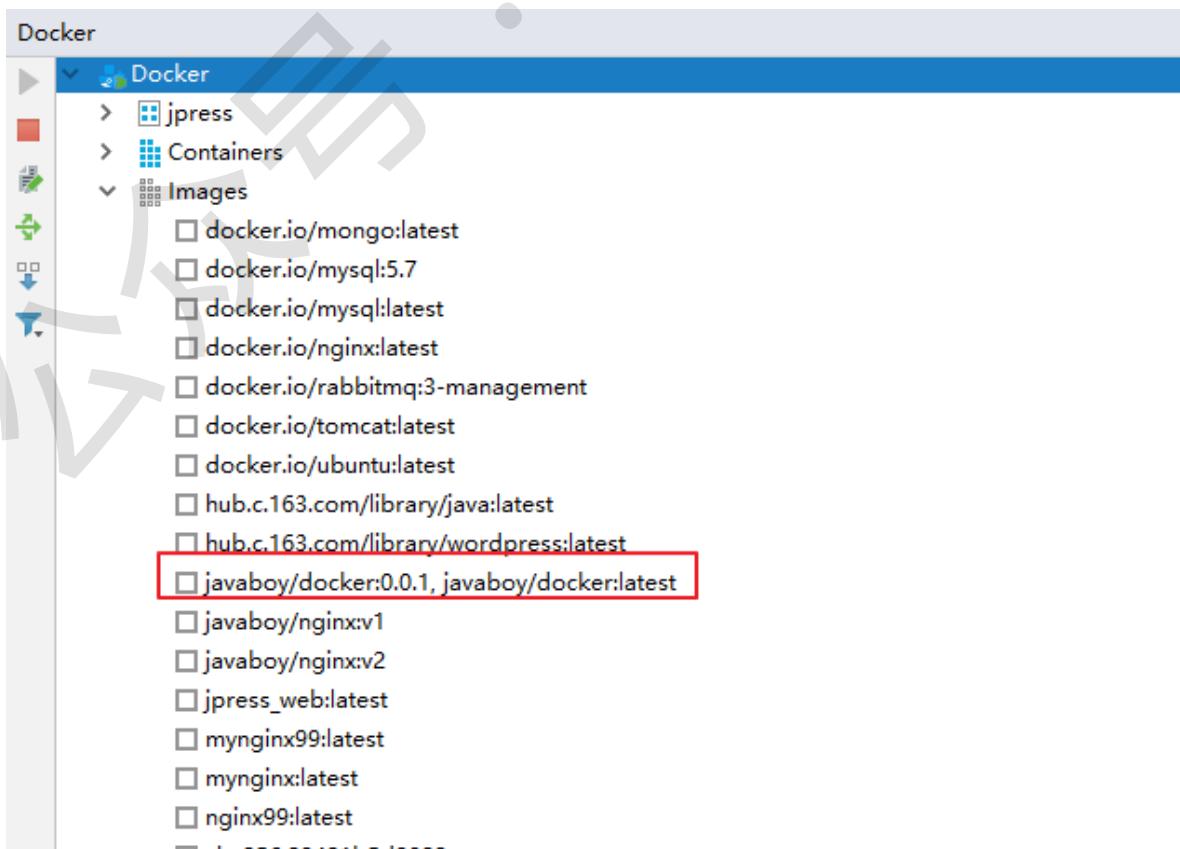
但是这种操作显然还是有点麻烦，结合我们一开始安装的 Docker 插件，这个运行步骤还可以做进一步的简化。

5.2 运行方式二

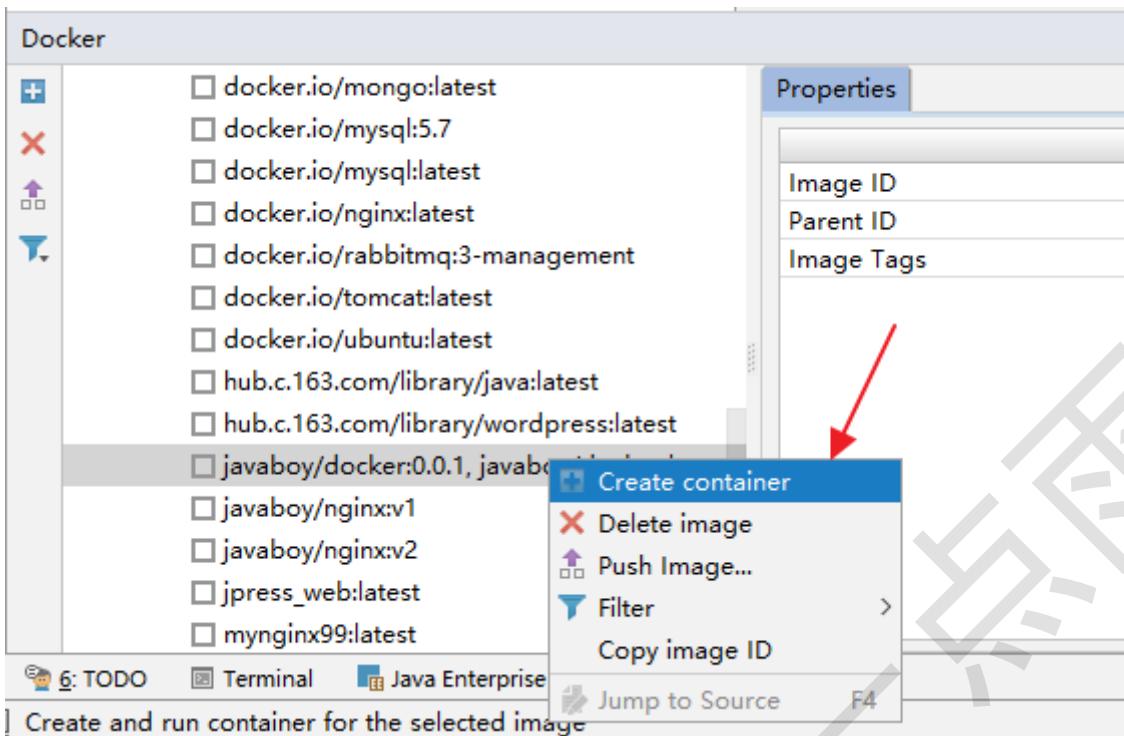
大家注意，此时我们的 IDEA 中多了一个选项，就是 docker，如下：



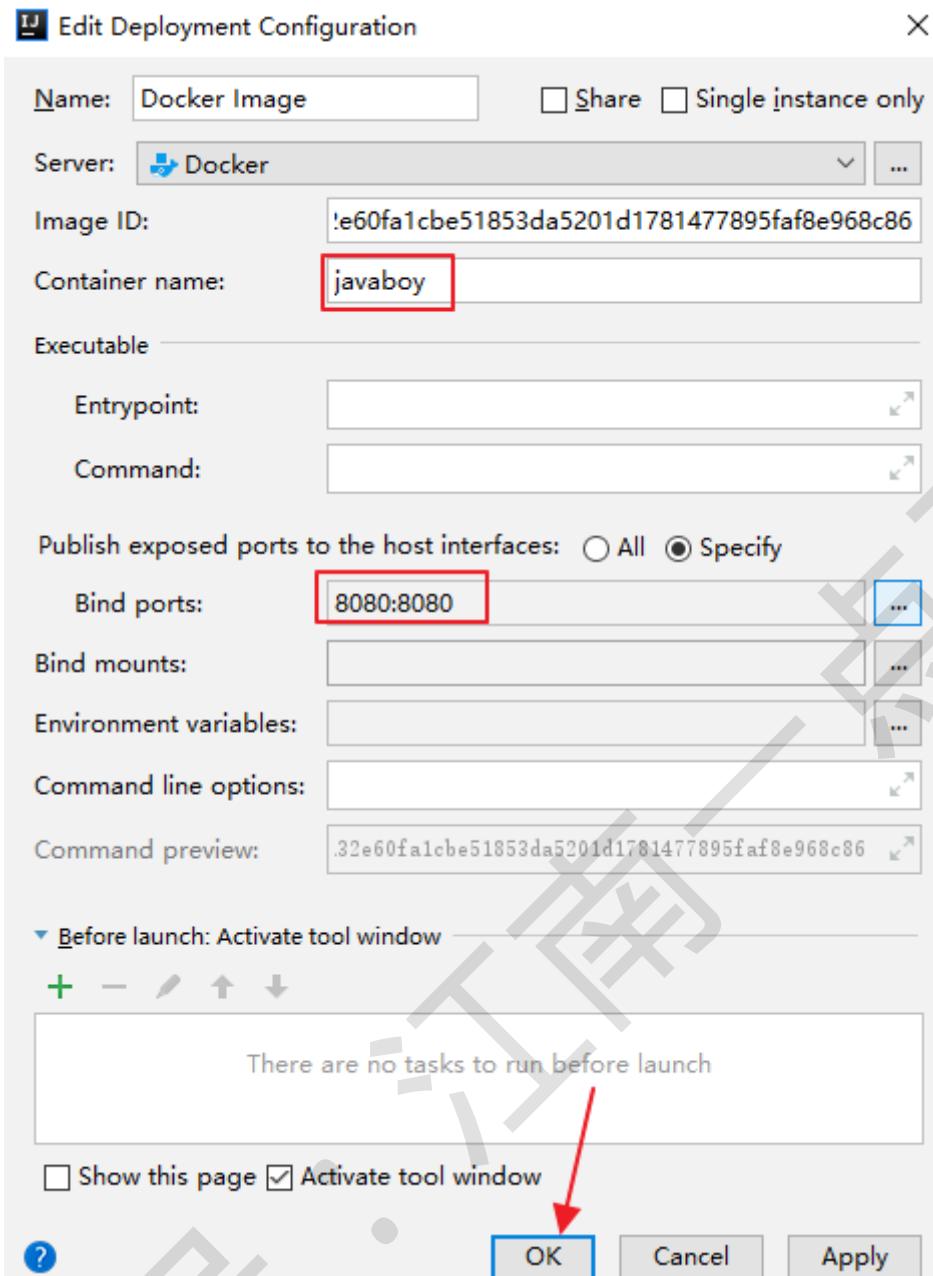
点击左边的绿色启动按钮，连接上 Docker 容器，连接成功之后，我们就可以看到目前 Docker 中的所有容器和镜像了，当然也包括我们刚刚创建的 Docker 镜像，如下：



此时，我们选中这个镜像，右键单击，即可基于此镜像创建出一个容器，如下图：



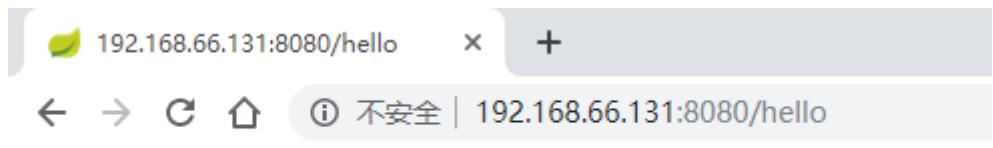
我们选择 Create container，然后填入容器的一些必要信息，配置一下容器名称，镜像 ID 会自动填上，暴露的端口使用 Specify 即可，然后写上端口的映射关系：



配置完成后，点击下方的 `run` 按钮，就可以开始运行了。运行日志如下：

注意，这个日志是在 Docker 的那个窗口里打印出来的。

项目运行成功之后，在浏览器输入远程服务器的地址，就可以访问了：



hello docker!

如此之后，我们的 Spring Boot 项目就算顺利发布到远程 Docker 容器中了。

好玩吗？试试！

本文案例我已经上传到 GitHub，小伙伴们可以参考：<https://github.com/lenvve/javaboy-code-samples>

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



之前松哥和大家分享过一篇将 Spring Boot 项目部署到远程 Docker 上的文章：

- [一键部署 Spring Boot 到远程 Docker 容器](#)

但是这种部署有一个问题，就是一个小小的 helloworld 构建成镜像之后，竟然都有 660 MB+，这就有点过分了；而且这种方式步骤繁琐，很多人看了头大。

因此松哥今天想再和大家聊一聊另外一种方案 **Jib**，这是谷歌开源的一个容器化运行方案，使用它我们将 Spring Boot 进行容器化部署只要两步：

- 第一步配置 Maven Plugin
- 第二步构建

我们一起来看看。

Jib

在之前那篇文章中，我们将 Spring Boot 项目进行容器化部署，要求开发人员要有一定的 Docker 技能作为支撑，然而在实际开发中，并非每个人都是 Docker 专家，或者说会用 Docker。

有鉴于此，Google 搞出来一个 Jib，使 Spring Boot 容器化部署变得更加简便，开发人员可以不需要任何 Docker 相关的技能，就能将 Spring Boot 项目构建成 Docker 中的镜像，而且还可以“顺便”将镜像 push 到 register 上，极大的简化了部署过程。

Jib 使用 Java 开发，使用也非常简单，可以作为 Maven 或者 Gradle 的插件直接集成到我们的项目中。它利用镜像分层和注册表缓存来实现快速、增量的构建。Jib 会自动读取项目的构建配置，代码组织到不同的层（依赖项、资源、类）中，然后它只会重新构建和推送发生变更的层。在项目进行快速迭代时，Jib 只将发生变更的层推送到 registers 来缩短构建时间。

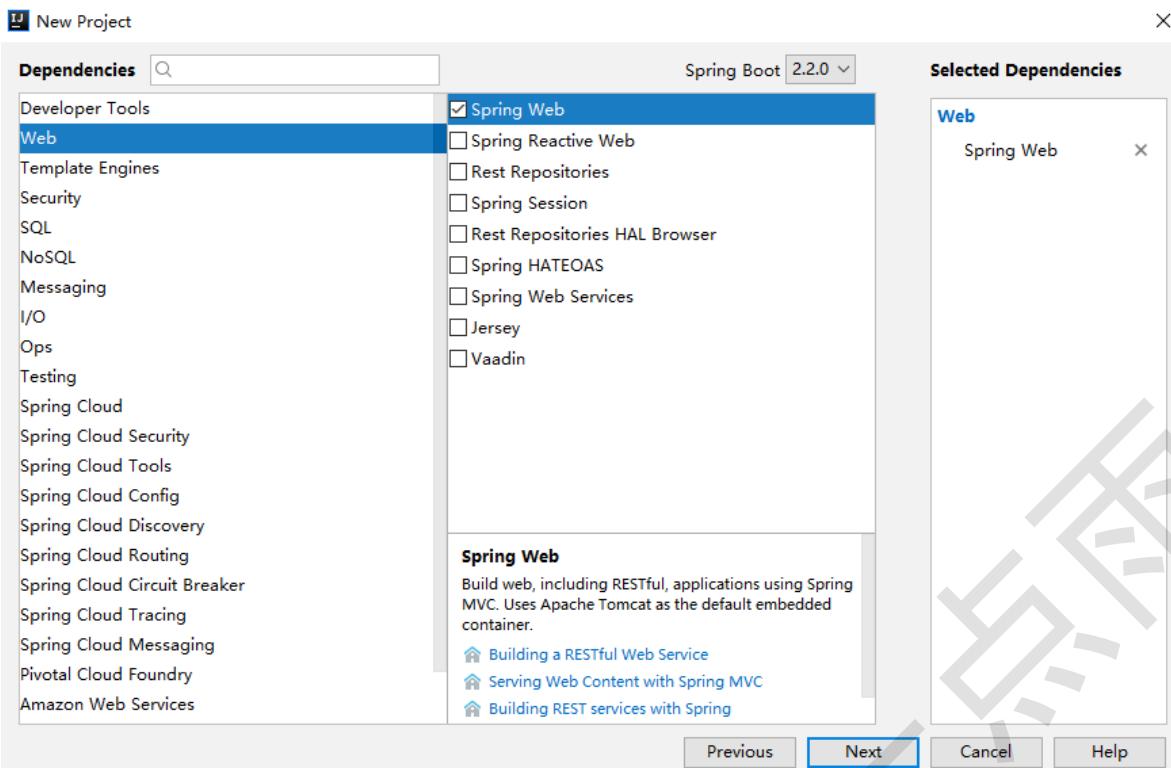
好了，大致了解了 Jib 之后，接下来我们来看看 Jib 要怎么使用。

准备工作

Jib 可以直接将构建好的镜像 push 到 registers 上，如果公司有自己的私有镜像站的话，可以直接推送至私有镜像站上，本文我就将构建好的镜像推送到官方的 Docker Hub 上，因此需要大家提前准备一个 Docker Hub 的账号，账号大家可以直接去 Docker Hub 上面注册 (<https://hub.docker.com/>)，大家要是对 Docker Hub 这些东西不了解，可以在公众号后台回复 docker，获取松哥自制的 Docker 教程。

牛刀小试

首先我们来创建一个 Spring Boot 工程，创建时只需要添加一个 Web 依赖即可：



项目创建成功后，添加一个 HelloController 用来做测试：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "hello jib";
    }
}
```

然后，在 pom.xml 中添加上 jib 的插件，如下：

```
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>1.7.0</version>
    <configuration>
        <from>
            <image>openjdk:alpine</image>
        </from>
        <to>
            <image>docker.io/wongsung/dockerjib</image>
            <tags>
                <tag>v1</tag>
            </tags>
            <auth>
                <username>wongsung</username>
                <password>你的密码</password>
            </auth>
        </to>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
```

```
<goal>build</goal>
</goals>
</execution>
</executions>
</plugin>
```

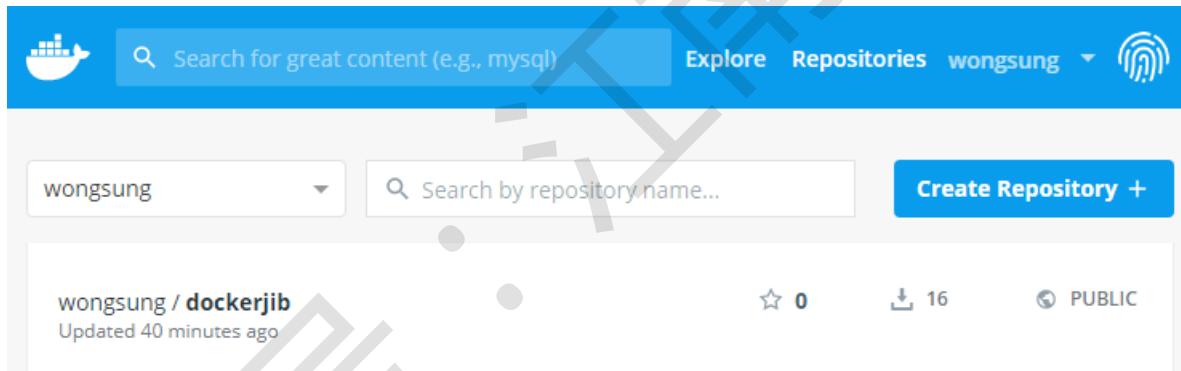
关于这段配置，我说如下几点：

- 首先就是版本号的问题，我这里使用的是 `1.7.0`，网上有的教程比较老，用的 `0.x` 的版本，老的版本在配置 Docker 认证的时候非常麻烦，所以版本这块建议大家使用当前最新版。
- from 中的配置表示本镜像构建所基于的根镜像为 `openjdk:alpine`
- to 中的配置表示本镜像构建完成后，要发布到哪里去，如果是发布到私有镜像站，就写自己私有镜像站的地址，如果是发布到 Docker Hub 上，就参考我这里的写法 `docker.io/wongsung/dockerjib`，其中 `wongsung` 表示你在 Docker Hub 上注册的用户名，`dockerjib` 表示你镜像的名字，可以随意取。
- tags 中配置的是自己镜像的版本。
- auth 中配置你在 Docker Hub 上的用户名/密码。
- executions 节点中的就是常规配置了，我就不再多说了。

配置完成后，在命令行执行如下命令将当前下项目构建成一个 Docker 镜像并 push 到 Docker Hub：

```
mvn compile jib:build
```

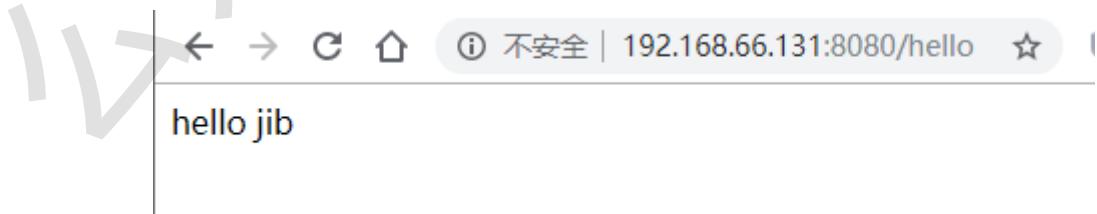
构建完成后，我们在 Docker Hub 上就能看到自己的镜像了：



接下来，启动 Docker，在 Docker 中执行如下命令拉取镜像下来并运行：

```
docker run -d --name mydockerjib -p 8080:8080 docker.io/wongsung/dockerjib:v1
```

启动成功后，我们在浏览器中就可以直接访问我们刚才的 Spring Boot 项目中的 hello 接口了：



是不是很方便？比我第一次给大家介绍的方案要方便很多。

注意

这种方式是将项目构建成镜像后并 push 到 registers 上，这种构建方式不需要你本地安装 Docker，如果你需要在本地运行镜像，那当然需要 Docker，单纯的构建是不需要 Docker 环境的。

本地构建

如果你电脑本地刚好安装了 Docker，有 Docker 环境，那么也可以将项目构建成本地 Docker 的镜像，

首先我们来查看一下本地镜像：

```
[sang-2:javaboy.org sang$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
mysql           5.7          e9c354083de7  3 months ago  373MB
```

可以看到只有 MySQL 镜像，然后我们执行如下命令构建本地镜像：

```
mvn compile jib:dockerBuild
```

看到如下构建日志信息表示构建成功：

```
[INFO] Using base image with digest: sha256:d4c84528b8de75bf40ef402e7b02ffe60c5ae834a702e510664be1f6fc03d510
[INFO]
[INFO] Container entrypoint set to [java, -cp, /app/resources:/app/classes:/app/libs/*, org.javaboy.dockerjib.DockerJibApplication]
[INFO]
[INFO] Built image to Docker daemon as wongsung/dockerjib, wongsung/dockerjib:v1
[INFO] Executing tasks:
[INFO] [=====] 100.0% complete
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 07:04 min
[INFO] Finished at: 2019-11-06T19:46:39+08:00
[INFO] -----
```

构建完成后，我们再来看本地镜像：

```
[sang-2:javaboy.org sang$ docker images
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
mysql           5.7          e9c354083de7  3 months ago  373MB
wongsung/dockerjib  latest      ccb73c2bd019  49 years ago  120MB
wongsung/dockerjib  v1          ccb73c2bd019  49 years ago  120MB
```

可以看到，已经构建成功了，接下来启动命令和上面一样，我就不重复展示了。

结语

容器的出现，让我们的 Java 程序比任何时候都接近“一次编写，到处运行”，Spring Boot 容器化部署也是越来越方便，后面有空松哥再和大家聊聊结合 Jenkins 的用法，好了，本文的案例我已经上传到 GitHub：<https://github.com/lenve/javaboy-code-samples>，有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

邮件发送其实是一个非常常见的需求，用户注册，找回密码等地方，都会用到，使用 JavaSE 代码发送邮件，步骤还是挺繁琐的，Spring Boot 中对于邮件发送，提供了相关的自动化配置类，使得邮件发送变得非常容易，本文我们就来一探究竟！看看使用 Spring Boot 发送邮件的 5 中姿势。

邮件基础

我们经常会听到各种各样的邮件协议，比如 SMTP、POP3、IMAP，那么这些协议有什么作用，有什么区别？我们先来讨论一下这个问题。

SMTP 是一个基于 TCP/IP 的应用层协议，江湖地位有点类似于 HTTP，SMTP 服务器默认监听的端口号为 25。看到这里，小伙伴们可能会想到既然 SMTP 协议是基于 TCP/IP 的应用层协议，那么我是不是也可以通过 Socket 发送一封邮件呢？回答是肯定的。

生活中我们投递一封邮件要经过如下几个步骤：

1. 深圳的小王先将邮件投递到深圳的邮局
2. 深圳的邮局将邮件运送到上海的邮局
3. 上海的小张来邮局取邮件

这是一个缩减版的生活中邮件发送过程。这三个步骤可以分别对应我们的邮件发送过程，假设从 aaa@qq.com 发送邮件到 111@163.com：

1. aaa@qq.com 先将邮件投递到腾讯的邮件服务器
2. 腾讯的邮件服务器将我们的邮件投递到网易的邮件服务器
3. 111@163.com 登录网易的邮件服务器查看邮件

邮件投递大致就是这个过程，这个过程就涉及到了多个协议，我们来分别看一下。

SMTP 协议全称为 Simple Mail Transfer Protocol，译作简单邮件传输协议，它定义了邮件客户端软件与 SMTP 服务器之间，以及 SMTP 服务器与 SMTP 服务器之间的通信规则。

也就是说 aaa@qq.com 用户先将邮件投递到腾讯的 SMTP 服务器这个过程就使用了 SMTP 协议，然后腾讯的 SMTP 服务器将邮件投递到网易的 SMTP 服务器这个过程也依然使用了 SMTP 协议，SMTP 服务器就是用来收邮件。

而 POP3 协议全称为 Post Office Protocol，译作邮局协议，它定义了邮件客户端与 POP3 服务器之间的通信规则，那么该协议在什么场景下会用到呢？当邮件到达网易的 SMTP 服务器之后，111@163.com 用户需要登录服务器查看邮件，这个时候就该协议就用上了：邮件服务商都会为每一个用户提供专门的邮件存储空间，SMTP 服务器收到邮件之后，就将邮件保存到相应用户的邮件存储空间中，如果用户要读取邮件，就需要通过邮件服务商的 POP3 邮件服务器来完成。

最后，可能也有小伙伴们听说过 IMAP 协议，这个协议是对 POP3 协议的扩展，功能更强，作用类似，这里不再赘述。

准备工作

目前国内大部分的邮件服务商都不允许直接使用用户名/密码的方式来在代码中发送邮件，都是要先申请授权码，这里以 QQ 邮箱为例，向大家演示授权码的申请流程：首先我们需要先登录 QQ 邮箱网页版，点击上方的设置按钮：



然后点击账户选项卡：



在账户选项卡中找到开启POP3/SMTP选项，如下：

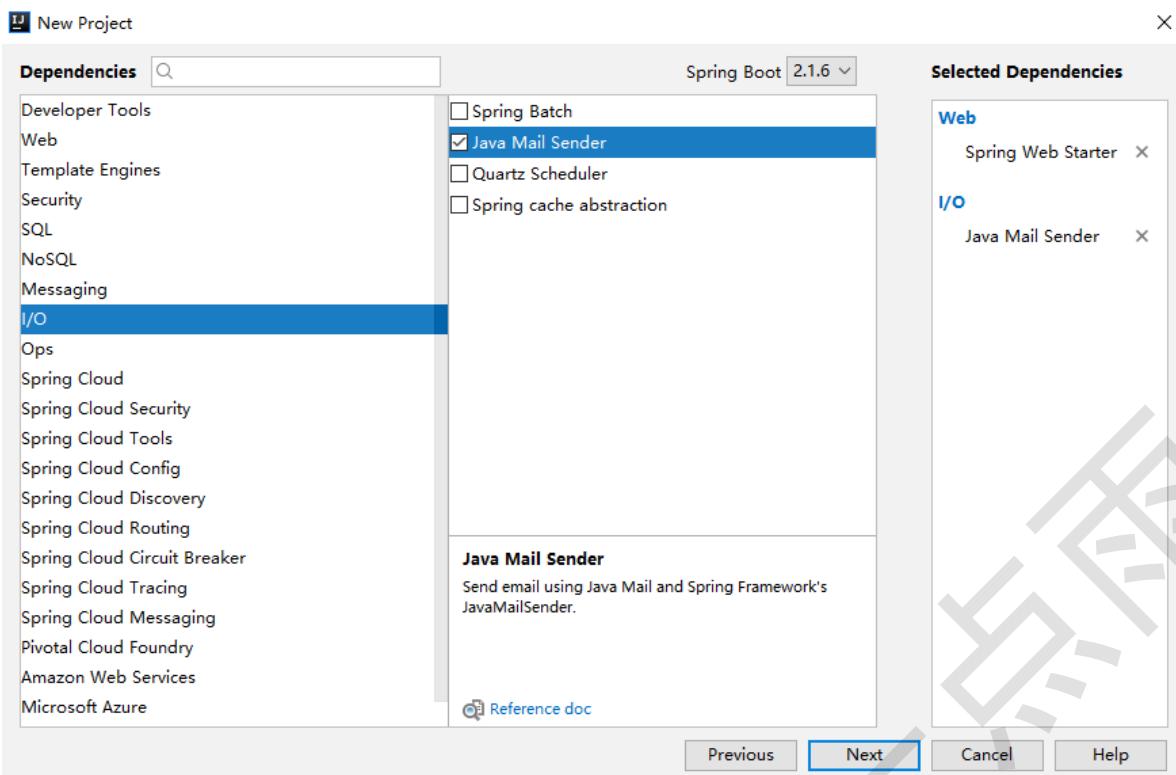


点击开启，开启相关功能，开启过程需要手机号码验证，按照步骤操作即可，不赘述。开启成功之后，即可获取一个授权码，将该号码保存好，一会使用。

项目创建

接下来，我们就可以创建项目了，Spring Boot 中，对于邮件发送提供了自动配置类，开发者只需要加入相关依赖，然后配置一下邮箱的基本信息，就可以发送邮件了。

- 首先创建一个 Spring Boot 项目，引入邮件发送依赖：



创建完成后，项目依赖如下：

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

- 配置邮箱基本信息

项目创建成功后，接下来在 application.properties 中配置邮箱的基本信息：

```
spring.mail.host=smtp.qq.com
spring.mail.port=587
spring.mail.username=1510161612@qq.com
spring.mail.password=ubknfzhjkhrbbabe
spring.mail.default-encoding=UTF-8
spring.mail.properties.mail.smtp.socketFactoryClass=javax.net.ssl.SSLSocketFactory
spring.mail.properties.mail.debug=true
```

配置含义分别如下：

- 配置 SMTP 服务器地址
- SMTP 服务器的端口
- 配置邮箱用户名
- 配置密码，注意，不是真正的密码，而是刚刚申请到的授权码
- 默认的邮件编码
- 配饰 SSL 加密工厂
- 表示开启 DEBUG 模式，这样，邮件发送过程的日志会在控制台打印出来，方便排查错误

如果不知道 smtp 服务器的端口或者地址的话，可以参考 腾讯的邮箱文档

- <https://service.mail.qq.com/cgi-bin/help?subtype=1&&id=28&&no=371>

做完这些之后，Spring Boot 就会自动帮我们配置好邮件发送类，相关的配置在 `org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration` 类中，部分源码如下：

```
@Configuration
@ConditionalOnClass({ MimeMessage.class, MimeType.class, MailSender.class })
@ConditionalOnMissingBean(MailSender.class)
@Conditional(MailSenderCondition.class)
@EnableConfigurationProperties(MailProperties.class)
@Import({ MailSenderJndiConfiguration.class,
        MailSenderPropertiesConfiguration.class })
public class MailSenderAutoConfiguration {
}
```

从这段代码中，可以看到，导入了另外一个配置 `MailSenderPropertiesConfiguration` 类，这个类中，提供了邮件发送相关的工具类：

```
@Configuration
@ConditionalOnProperty(prefix = "spring.mail", name = "host")
class MailSenderPropertiesConfiguration {
    private final MailProperties properties;
    MailSenderPropertiesConfiguration(MailProperties properties) {
        this.properties = properties;
    }
    @Bean
    @ConditionalOnMissingBean
    public JavaMailSenderImpl mailSender() {
        JavaMailSenderImpl sender = new JavaMailSenderImpl();
        applyProperties(sender);
        return sender;
    }
}
```

可以看到，这里创建了一个 `JavaMailSenderImpl` 的实例，`JavaMailSenderImpl` 是 `JavaMailSender` 的一个实现，我们将使用 `JavaMailSenderImpl` 来完成邮件的发送工作。

做完如上两步，邮件发送的准备工作就算是完成了，接下来就可以直接发送邮件了。

具体的发送，有 5 种不同的方式，我们一个一个来看。

发送简单邮件

简单邮件就是指邮件内容是一个普通的文本文档：

```
@Autowired
JavaMailSender javaMailSender;
@Test
public void sendSimpleMail() {
    SimpleMailMessage message = new SimpleMailMessage();
    message.setSubject("这是一封测试邮件");
    message.setFrom("1510161612@qq.com");
    message.setTo("25xxxxx755@qq.com");
    message.setCc("37xxxxx37@qq.com");
    message.setBcc("14xxxxx098@qq.com");
```

```
        message.setSentDate(new Date());
        message.setText("这是测试邮件的正文");
        javaMailSender.send(message);
    }
```

从上往下，代码含义分别如下：

1. 构建一个邮件对象
2. 设置邮件主题
3. 设置邮件发送者
4. 设置邮件接收者，可以有多个接收者
5. 设置邮件抄送人，可以有多个抄送人
6. 设置隐私抄送人，可以有多个
7. 设置邮件发送日期
8. 设置邮件的正文
9. 发送邮件

最后执行该方法，就可以实现邮件的发送，发送效果图如下：



这是测试邮件的正文

发送带附件的邮件

邮件的附件可以是图片，也可以是普通文件，都是支持的。

```
@Test
public void sendAttachFileMail() throws MessagingException {
    MimeMessage mimeMessage = javaMailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);
    helper.setSubject("这是一封测试邮件");
    helper.setFrom("1510161612@qq.com");
    helper.setTo("25xxxxxxxxxx@qq.com");
    helper.setCc("37xxxxxxxxxx@qq.com");
    helper.setBcc("14xxxxxxxxxx@qq.com");
    helper.setSentDate(new Date());
    helper.setText("这是测试邮件的正文");
    helper.addAttachment("javaboy.jpg", new
File("C:\\\\Users\\\\sang\\\\Downloads\\\\javaboy.png"));
    javaMailSender.send(mimeMessage);
}
```

注意这里在构建邮件对象上和前文有所差异，这里是通过 `javaMailSender` 来获取一个复杂邮件对象，然后再利用 `MimeMessageHelper` 对邮件进行配置，`MimeMessageHelper` 是一个邮件配置的辅助工具类，创建时候的 `true` 表示构建一个 `multipart message` 类型的邮件，有了 `MimeMessageHelper` 之后，我们针对邮件的配置都是由 `MimeMessageHelper` 来代劳。

最后通过 `addAttachment` 方法来添加一个附件。

执行该方法，邮件发送效果图如下：



发送带图片资源的邮件

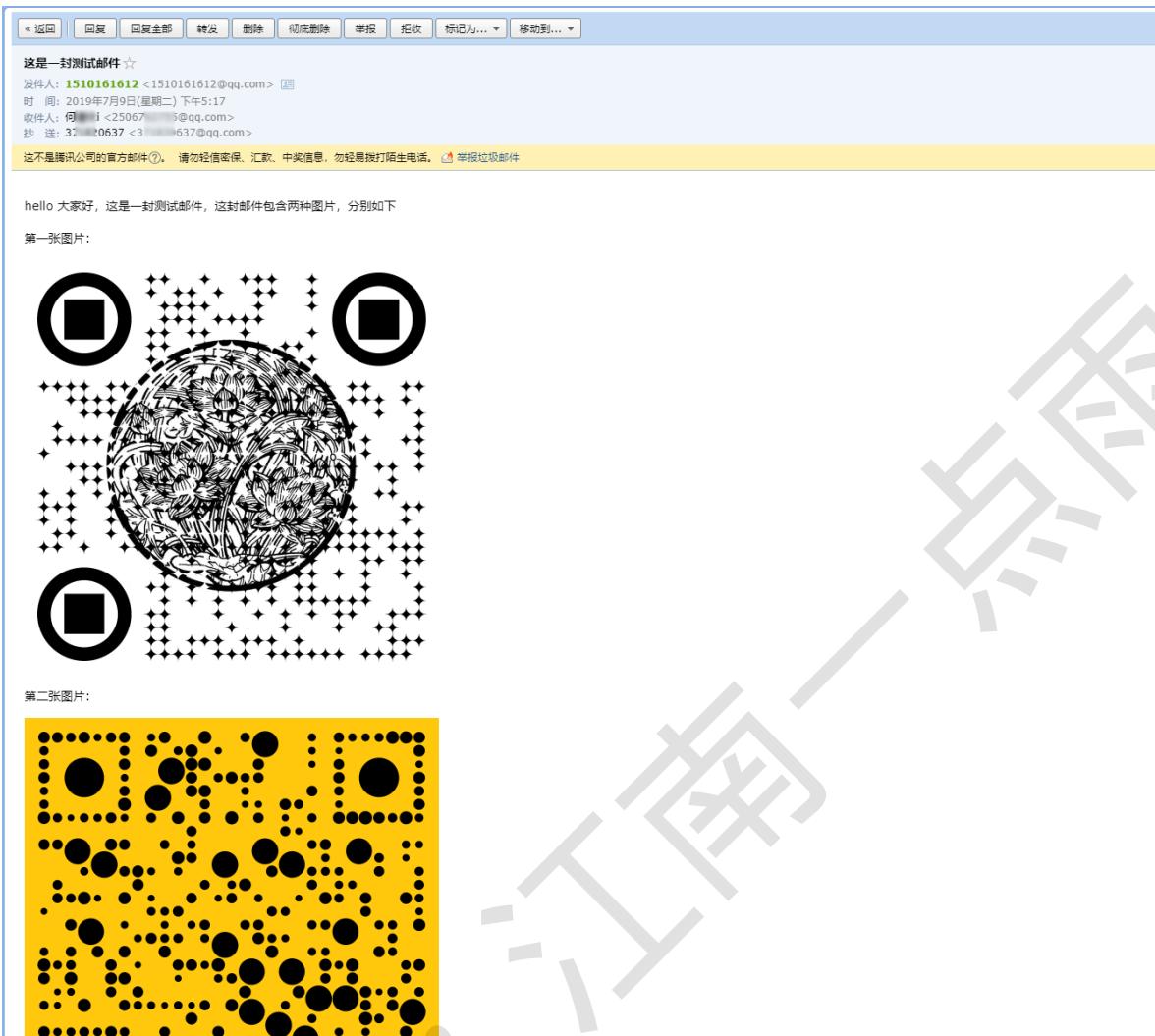
图片资源和附件有什么区别呢？图片资源是放在邮件正文中的，即一打开邮件，就能看到图片。但是一般来说，不建议使用这种方式，一些公司会对邮件内容的大小有限制（因为这种方式是将图片一起发送的）。

```
@Test
public void sendImgResMail() throws MessagingException {
    MimeMessage mimeMessage = javaMailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);
    helper.setSubject("这是一封测试邮件");
    helper.setFrom("1510161612@qq.com");
    helper.setTo("25xxxxx755@qq.com");
    helper.setCc("37xxxxx37@qq.com");
    helper.setBcc("14xxxxx098@qq.com");
    helper.setSentDate(new Date());
    helper.setText("<p>hello 大家好，这是一封测试邮件，这封邮件包含两种图片，分别如下</p>
<p>第一张图片：</p><img src='cid:p01' /><p>第二张图片：</p><img
src='cid:p02' />", true);
    helper.addInline("p01", new FileSystemResource(new
File("C:\\\\Users\\\\sang\\\\Downloads\\\\javaboy.png")));
    helper.addInline("p02", new FileSystemResource(new
File("C:\\\\Users\\\\sang\\\\Downloads\\\\javaboy2.png")));
    javaMailSender.send(mimeMessage);
}
```

这里的邮件 `text` 是一个 HTML 文本，里边涉及到的图片资源先用一个占位符占着，`setText` 方法的第二个参数 `true` 表示第一个参数是一个 HTML 文本。

setText 之后，再通过 addInline 方法来添加图片资源。

最后执行该方法，发送邮件，效果如下：



在公司实际开发中，第一种和第三种都不是使用最多的邮件发送方案。因为正常来说，邮件的内容都是比较的丰富的，所以大部分邮件都是通过 HTML 来呈现的，如果直接拼接 HTML 字符串，这样以后不好维护，为了解决这个问题，一般邮件发送，都会有相应的邮件模板。最具代表性的两个模板就是 `Freemarker` 模板和 `Thymeleaf` 模板了。

使用 Freemarker 作邮件模板

首先需要引入 `Freemarker` 依赖：

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

然后在 `resources/templates` 目录下创建一个 `mail.ftl` 作为邮件发送模板：

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

```

<p>hello 欢迎加入 xxx 大家庭，您的入职信息如下：</p>
<table border="1">
    <tr>
        <td>姓名</td>
        <td>${username}</td>
    </tr>
    <tr>
        <td>工号</td>
        <td>${num}</td>
    </tr>
    <tr>
        <td>薪水</td>
        <td>${salary}</td>
    </tr>
</table>
<div style="color: #ff1a0e">一起努力创造辉煌</div>
</body>
</html>

```

接下来，将邮件模板渲染成 HTML，然后发送即可。

```

@Test
public void sendFreemarkerMail() throws MessagingException, IOException,
TemplateException {
    MimeMessage mimeMessage = javaMailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);
    helper.setSubject("这是一封测试邮件");
    helper.setFrom("1510161612@qq.com");
    helper.setTo("25xxxxx755@qq.com");
    helper.setCc("37xxxxx37@qq.com");
    helper.setBcc("14xxxxx098@qq.com");
    helper.setSentDate(new Date());
    //构建 Freemarker 的基本配置
    Configuration configuration = new
Configuration(Configuration.VERSION_2_3_0);
    // 配置模板位置
    ClassLoader loader = MailApplication.class.getClassLoader();
    configuration.setClassLoaderForTemplateLoading(loader, "templates");
    //加载模板
    Template template = configuration.getTemplate("mail.ftl");
    User user = new User();
    user.setUsername("javaboy");
    user.setNum(1);
    user.setSalary((double) 99999);
    StringWriter out = new StringWriter();
    //模板渲染，渲染的结果将被保存到 out 中，将out 中的 html 字符串发送即可
    template.process(user, out);
    helper.setText(out.toString(), true);
    javaMailSender.send(mimeMessage);
}

```

需要注意的是，虽然引入了 `Freemarker` 的自动化配置，但是我们在这里是直接 `new Configuration` 来重新配置 `Freemarker` 的，所以 `Freemarker` 默认的配置这里不生效，因此，在填写模板位置时，值为 `templates`。

调用该方法，发送邮件，效果图如下：

这是一封测试邮件 ☆

发件人: 1510161612 <1510161612@qq.com> [回]

时间: 2019年7月9日(星期二) 晚上6:05

收件人: 1510161612 <25067155@qq.com>

抄送: 371820637 <371820637@qq.com>

这不是腾讯公司的官方邮件。请勿轻信密保、汇款、中奖信息，勿轻易拨打陌生电话。举报垃圾邮件

hello 欢迎加入 xxx 大家庭，您的入职信息如下：

姓名	javaboy
工号	1
薪水	99,999

一起努力创造辉煌

使用 Thymeleaf 作邮件模板

推荐在 Spring Boot 中使用 Thymeleaf 来构建邮件模板。因为 Thymeleaf 的自动化配置提供了一个 TemplateEngine，通过 TemplateEngine 可以方便的将 Thymeleaf 模板渲染为 HTML，同时，Thymeleaf 的自动化配置在这里是继续有效的。

首先，引入 Thymeleaf 依赖：

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

然后，创建 Thymeleaf 邮件模板：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <p>hello 欢迎加入 xxx 大家庭，您的入职信息如下：</p>
    <table border="1">
        <tr>
            <td>姓名</td>
            <td th:text="${username}"></td>
        </tr>
        <tr>
            <td>工号</td>
            <td th:text="${num}"></td>
        </tr>
        <tr>
            <td>薪水</td>
            <td th:text="${salary}"></td>
        </tr>
    </table>
    <div style="color: #ff1a0e">一起努力创造辉煌</div>
</body>
```

```
</html>
```

接下来发送邮件：

```
@Autowired  
TemplateEngine templateEngine;  
  
@Test  
public void sendThymeleafMail() throws MessagingException {  
    MimeMessage mimeMessage = javaMailSender.createMimeMessage();  
    MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, true);  
    helper.setSubject("这是一封测试邮件");  
    helper.setFrom("1510161612@qq.com");  
    helper.setTo("25xxxxx755@qq.com");  
    helper.setCc("37xxxxx37@qq.com");  
    helper.setBcc("14xxxxx098@qq.com");  
    helper.setSentDate(new Date());  
    Context context = new Context();  
    context.setVariable("username", "javaboy");  
    context.setVariable("num", "000001");  
    context.setVariable("salary", "99999");  
    String process = templateEngine.process("mail.html", context);  
    helper.setText(process, true);  
    javaMailSender.send(mimeMessage);  
}
```

调用该方法，发送邮件，效果图如下：



hello 欢迎加入 xxx 大家庭，您的入职信息如下：

姓名	javaboy
工号	1
薪水	99,999

一起努力创造辉煌

好了，这就是我们今天说的 5 种邮件发送姿势，不知道你掌握了没有呢？

本文案例已经上传到 GitHub： <https://github.com/lenve/javaboy-code-samples>。

有问题欢迎留言讨论。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

真是郁闷，不过这事又一次提醒我解决问题还是要根治，不能囫囵吞枣，否则相同的问题可能会以不同的形式出现，每次都得花时间去搞。刨根问底，一步到位，再遇到类似问题就可以分分钟解决了。

如果大家没看过松哥之前写的 Spring Boot 整合 Spring Session，可以先回顾下：

- [Spring Boot2 系列教程\(二十八\)Spring Boot 整合 Session 共享](#)

第一次踩坑

事情是这样的，大概在今年 6 月初的时候，我在项目中使用到了 Session 共享，当时采用的方案就是 Redis+Spring Session。本来这是一个很简单的问题，我在以前的项目中也用过多次这种方案，早已轻车熟路，但是那次有点不对劲，项目启动时候报了如下错误：

```
Caused by: java.lang.NoClassDefFoundError: org/springframework/security/web/authentication/RememberMeServices
  at java.lang.ClassLoader.defineClass1(Native Method) ~[na:1.8.0_171]
  at java.lang.ClassLoader.defineClass(ClassLoader.java:763) ~[na:1.8.0_171]
  at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142) ~[na:1.8.0_171]
  at java.net.URLClassLoader.defineClass(URLClassLoader.java:467) ~[na:1.8.0_171]
  at java.net.URLClassLoader.access$100(URLClassLoader.java:73) ~[na:1.8.0_171]
  at java.net.URLClassLoader$1.run(URLClassLoader.java:368) ~[na:1.8.0_171]
  | at java.net.URLClassLoader$1.run(URLClassLoader.java:362) ~[na:1.8.0_171] <1 internal call>
  at java.net.URLClassLoader.findClass(URLClassLoader.java:361) ~[na:1.8.0_171]
  at java.lang.ClassLoader.loadClass(ClassLoader.java:424) ~[na:1.8.0_171]
  at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349) ~[na:1.8.0_171]
  at java.lang.ClassLoader.loadClass(ClassLoader.java:357) ~[na:1.8.0_171]
  at java.lang.Class.getDeclaredMethods0(Native Method) ~[na:1.8.0_171]
  at java.lang.Class.privateGetDeclaredMethods(Class.java:2701) ~[na:1.8.0_171]
  at java.lang.Class.getDeclaredMethods(Class.java:1975) ~[na:1.8.0_171]
  at org.springframework.util.ReflectionUtils.getDeclaredMethods(ReflectionUtils.java:489) ~[spring-core-5.1.7.RELEASE.jar:5.1.7.RELEASE]
  ... 39 common frames omitted
```

一模一样的代码，但是运行就是会出错，我感觉莫名其妙。因为在 Spring Boot 中整合 Spring Session 是一个非常简单的操作，就几行 Redis 的配置而已，我在确认了代码没问题之后，很快想到了可能是版本问题，因为当时 Spring Boot2.1.5 刚刚发布，我喜欢用最新版。于是我尝试将 Spring Boot 的版本切换到 2.1.4，切换回去之后，果然就 OK 了，再次启动项目又不会报错了。于是基本确定这是 Spring Boot 的版本升级带来的问题。

但是当时我并没有深究，我以为就是官方出于安全考虑，让你在使用 Redis 时强制加上 Spring Security（因为根据错误提示，很容易想到加上 Spring Security 依赖），加上 Spring Security 依赖之后，果然就没有问题了，我也没有多想，这件事就这样过了。

第二次踩坑

前两天我在给星球上的小伙伴录制 Spring Boot 视频的时候，采用了 Spring Boot 最新版 2.1.7，也是 Spring Session，但是在创建项目的时候，忘记添加 Spring Security 依赖了（第一次踩坑之后，我每次用 Spring Session 都会自觉的加上 Spring Security 依赖），运行的时候竟然没报错！我就郁闷了。

于是我去试了 Spring Boot2.1.4、Spring Boot2.1.6 发现都没有问题，在使用 Spring Session 的时候都不需要添加 Spring Security 依赖，只有 Spring Boot2.1.5 才有这个问题。于是我大概明白了，这可能是一个 Bug，而不是版本升级的新功能。

这一次，那我就打算追究一下问题的根源。

源头

要追究问题的源头，我们当然得从 Spring Session 的自动化配置类开始。

在 Spring Boot2.1.5 的

`org.springframework.boot.autoconfigure.session.SessionAutoConfiguration` 类中，我看到如下源码：

```
@Bean  
@Conditional(DefaultCookieSerializerCondition.class)  
public DefaultCookieSerializer cookieSerializer(ServerProperties  
serverProperties,  
ObjectProvider<SpringSessionRememberMeServices>  
springSessionRememberMeServices) {  
    //.....  
    map.from(cookie::getMaxAge).to((maxAge) -> cookieSerializer  
        .setCookieMaxAge((int) maxAge.getSeconds()));  
    springSessionRememberMeServices.ifAvailable((  
        rememberMeServices) ->  
        cookieSerializer.setRememberMeRequestAttribute(  
            SpringSessionRememberMeServices.REMEMBER_ME_LOGIN_ATTR));  
    return cookieSerializer;  
}
```

从这一段源码中我们可以看到，这里使用到了 `SpringSessionRememberMeServices`，而这个类中则用到 Spring Security 中相关的类。因此，如果不引入 Spring Security 就会报错。

我们再来看看 Spring Boot2.1.6 中

`org.springframework.boot.autoconfigure.session.SessionAutoConfiguration` 类的源码，如下：

```
@Bean  
@Conditional(DefaultCookieSerializerCondition.class)  
public DefaultCookieSerializer cookieSerializer(ServerProperties  
serverProperties) {  
    //...  
    map.from(cookie::getMaxAge).to((maxAge) ->  
        cookieSerializer.setCookieMaxAge((int) maxAge.getSeconds()));  
    if (ClassUtils.isPresent(REMEMBER_ME_SERVICES_CLASS,  
        getClass().getClassLoader())) {  
        new  
        RememberMeServicesCookieSerializerCustomizer().apply(cookieSerializer);  
    }  
    return cookieSerializer;  
}
```

可以看到，在 Spring Boot2.1.6 中，这个问题已经得到修复。这里就没有 2.1.5 那么冲动了，上来了先用 `ClassUtils.isPresent` 方法判断了下

`REMEMBER_ME_SERVICES_CLASS`(`org.springframework.security.web.authentication.RememberMeServices`) 是否存在，存在的话，才有后面的操作。

至此，这个问题就总算弄懂了。

结语

大家平时遇到问题，如果项目不是很赶的话，可以留意多想想，多追究一下原因，说不定你会有很多意外的收获。我这次就是一个活生生的例子，一开始没多想，后来又发现不对劲，前前后后一折腾，反而又多浪费了一些时间。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南一点雨

Spring Boot2.2.0 这个版本发布没多久，Spring Boot2.2.1 就出来，看似不太重要的版本，却发生了一个小小变化，导致不少小伙伴掉坑了，我本来也没太在意，但是最近有快十个小伙伴在微信上问这个问题，看来我必须得写篇文章说下了，防止小伙伴们掉坑了。

到底是什么问题呢？其实就是 Freemarker 后缀变化的问题，一开始收到这个问题的时候，我以为就是小伙伴们学习不认真导致的，也没太在意：



结果最近不停有人掉坑，我觉得很有必要给各位小伙伴提个醒。

很多小伙伴可能很早就接触过 Freemarker，这个模板在 SSM 中也可以使用，只不过需要我们自己的配置东西稍微有点多。但是在之前我们使用 Freemarker 时，这个模板文件的后缀是 `ftl`，所以我们理所当然的认为这是标准后缀。

其实不然。

在 Freemarker 中，还有两个后缀，一个叫做 `ftlh`，这个用在 HTML 模板中，另一个叫做 `ftlx`，这个用在 XML 模板中。

Spring Boot2.2.0 之前，Freemarker 模板默认采用的后缀就是 `ftl`，我们可以看下 `FreeMarkerProperties` 类的部分源码（Spring Boot2.2.0 之前的版本）：

```
@ConfigurationProperties(  
    prefix = "spring.freemarker"  
)  
public class FreeMarkerProperties extends AbstractTemplateViewResolverProperties  
{  
    public static final String DEFAULT_TEMPLATE_LOADER_PATH =  
        "classpath:/templates/";  
    public static final String DEFAULT_PREFIX = "";  
    public static final String DEFAULT_SUFFIX = ".ftl";  
    private Map<String, String> settings = new HashMap();  
    private String[] templateLoaderPath = new String[]{"classpath:/templates/"};  
    private boolean preferFileSystemAccess = true;
```

可以看到，DEFAULT_SUFFIX 变量中定义的 Freemarker 默认的后缀还是 `.ftl`。

从 Spring Boot 2.2.0 开始，FreeMarkerProperties 文件内容就发生了变化，最新的 FreeMarkerProperties 文件部分源码如下：

```
@ConfigurationProperties(  
    prefix = "spring.freemarker"  
)  
public class FreeMarkerProperties extends AbstractTemplateViewResolverProperties  
{  
    public static final String DEFAULT_TEMPLATE_LOADER_PATH =  
        "classpath:/templates/";  
    public static final String DEFAULT_PREFIX = "";  
    public static final String DEFAULT_SUFFIX = ".ftl";  
    private Map<String, String> settings = new HashMap();  
    private String[] templateLoaderPath = new String[]{"classpath:/templates/"};  
    private boolean preferFileSystemAccess = true;
```

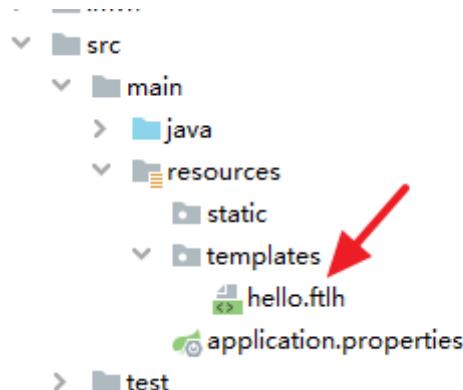
可以看到，这个时候在 DEFAULT_SUFFIX 变量中定义的默认后缀变成 `.ftl` 了。

就这样一个小小变化，就把很多初次接触 Spring Boot 的小伙伴搞晕啦。

那么这个问题如何解决呢？当大家发现了原因，应该也就能找到解决办法了，其实很简单，两个思路：

- 将 Freemarker 模板的后缀改为 `.ftlh`，推荐这种方式
- 在 application.properties 中修改默认配置

第一种方案，就是我们在定义 Freemarker 模板的时候，直接将原来的 `ftl` 改成 `ftlh` 就可以了，像下面这张图这样：



这样框架就能找到相应的模板文件了。

第二种方案就是 Freemarker 模板的后缀不变，依然是 `.ftl`，然后我们在 `application.properties` 中修改默认的后缀配置，如下：

```
spring.freemarker.suffix=.ftl
```

这样就是人为强行把 Freemarker 模板的后缀又改回 `.ftl` 了。

这两种方案都可以解决这个问题。

其实这个问题很简单，只要你看了 FreeMarkerProperties 类的源码，这个问题一下就明了了。

松哥刚开始录 Spring Boot 视频的时候，当时最新版是 2.1.6，后来随着视频录制，Spring Boot 版本一直在变化，视频里一直是跟随最新版录制，不过在讲 Freemarker 这块的时候，还是 2.1.6，所以当时还不存在上面这个问题。虽然问题不存在，但是我在视频中却是带领小伙伴们看了 FreeMarkerProperties 类的源码的，出了问题之后，有小伙伴就机智的去翻这个类的源码，然后自己顺利的把问题解决了。感觉深得松哥真传，吾心甚慰。

12月4日 下午17:12

今天在学习整合 freeMaker 那节，看了视频之后自己敲代码，报错，would dispatch back to the current handler URL [/index] again 。百度一通也没解决，然后跟着视频最前面的那部分查看源码，发现 2.2.1 (.ftlh) 与 2.1.6 (.flh) 对于模板的后缀定义不一样，自己在 properties 文件中定义了后缀（[spring.freemarker.suffix=.ftl](#)），顺便也复习了前面的课程。视频非常不错，已经有那种通过撸源码解决问题的快感了，而不是一味的百度。话说如果我用的 boot 版本与你一样的话，估计视频前期对我来说也是白瞎。感觉这视频买的值 😊。

感谢松哥的视频哟

好了，一个小小的坑，小伙伴们在这里遇到问题稍稍留意下就可以了。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

Hello 各位小伙伴，松哥今天要和大家聊一个有意思的话题，就是使用 Spring Boot 开发微信公众号后台。

很多小伙伴可能注意到松哥的个人网站 (<http://www.javaboy.org>) 前一阵子上线了一个公众号内回复口令解锁网站文章的功能，还有之前就有的公众号内回复口令获取超 2TB 免费视频教程的功能（[免费视频教程](#)），这两个都是松哥基于 Spring Boot 来做的，最近松哥打算通过一个系列的文章，来向小伙伴们介绍下如何通过 Spring Boot 来开发公众号后台。

1. 缘起

今年 5 月份的时候，我想把我自己之前收集到的一些视频教程分享给公众号上的小伙伴，可是这些视频教程大太了，无法一次分享，单次分享分享链接立马就失效了，为了把这些视频分享给大家，我把视频拆分成了很多份，然后设置了不同的口令，小伙伴们在公众号后台通过回复口令就可以获取到这些视频，口令前前后后有 100 多个，我一个一个手动的在微信后台进行配置。这么搞工作量很大，前前后后大概花了三个晚上才把这些东西搞定。

于是我就在想，该写点代码了。

上个月买了服务器，也备案了，该有的都有了，于是就打算把这些资源用代码实现下，因为大学时候搞过公众号开发，倒也没什么难度，于是说干就干。

2. 实现思路

其实松哥这个回复口令获取视频链接的实现原理很简单，说白了，就是一个数据查询操作而已，回复的口令是查询关键字，回复的内容则是查询结果。这个原理很简单。

另一方面大家需要明白微信公众号后台开发消息发送的一个流程，大家看下面这张图：

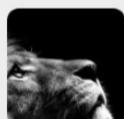
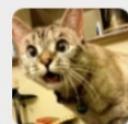


Spring Boot2 系列教程(十七)SpringBoot 整合 Swagger2



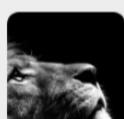
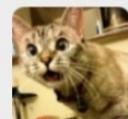
21:09

javaboy4096



链接: https://pan.baidu.com/s/1Z_Kq1_JcSRNq4M9NFWt5Rw
提取码: caf。给好学的你点个赞, 松哥的视频资源库会定期更新内容, 请留意公众号【江南一点雨】的推送消息哦!

javaboy4096



链接: https://pan.baidu.com/s/1Z_Kq1_JcSRNq4M9NFWt5Rw
提取码: caf。给好学的你点个赞, 松哥的视频资源库会定期更新

内容，请留意公众号【江南一点雨】的推送消息哦！

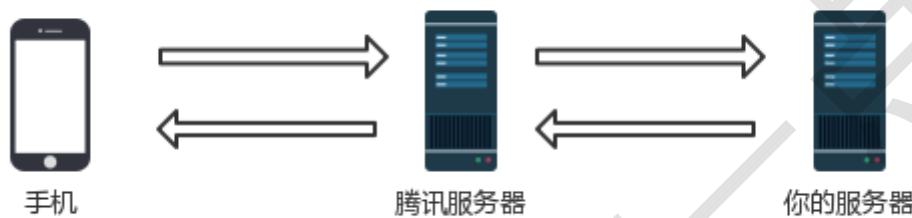


学习资源

加群

我的书

这是大家在公众号后台回复关键字的情况。那么这个消息是怎么样一个传递流程呢？我们来看看下面这张图：



这张图，我给大家稍微解释下：

- 首先 `javaboy4096` 这个字符从公众号上发送到了微信服务器
- 接下来微信服务器会把 `javaboy4096` 转发到我自己的服务器上
- 我收到 `javaboy4096` 这个字符之后，就去数据库中查询，将查询的结果，按照腾讯要求的 XML 格式进行返回
- 微信服务器把从我的服务器收到的信息，再发回到微信上，于是小伙伴们就看到了返回结果了

大致的流程就是这个样子。

接下来我们就来看一下实现细节。

3. 公众号后台配置

开发的第一步，是微信服务器要验证我们自己的服务器是否有效。

首先我们登录微信公众平台官网后，在公众平台官网的 **开发-基本设置** 页面，勾选协议成为开发者，然后点击“修改配置”按钮，填写：

- 服务器地址 (URL)
- Token
- EncodingAESKey

基本配置

 基本配置 / 填写服务器配置

请填写接口配置信息，此信息需要你拥有自己的服务器资源。
填写的URL需要正确响应微信发送的Token验证，请阅读[接入指南](#)。

URL

必须以http://或https://开头，分别支持80端口和443端口。

Token

必须为英文或数字，长度为3-32字符。

[什么是Token？](#)

EncodingAESKey

 0 / 43 随机生成

消息加密密钥由43位字符组成，可随机修改，字符范围为A-Z, a-z, 0-9。

[什么是EncodingAESKey？](#)

消息加解密方式

请根据业务需要，选择消息加解密类型，启用后将立即生效

明文模式

明文模式下，不使用消息体加解密功能，安全系数较低

兼容模式

兼容模式下，明文、密文将共存，方便开发者调试和维护

安全模式（推荐）

安全模式下，消息包为纯密文，需要开发者加密和解密，安全系数高

这里的 URL 配置好之后，我们需要针对这个 URL 开发两个接口，一个是 GET 请求的接口，这个接口用来做服务器有效性验证，另一个则是 POST 请求的接口，这个用来接收微信服务器发送来的消息。也就是说，微信服务器的消息都是通过 POST 请求发给我的。

Token 可由开发者可以任意填写，用作生成签名（该 Token 会和接口 URL 中包含的 Token 进行比对，从而验证安全性）。

EncodingAESKey 由开发者手动填写或随机生成，将用作消息体加解密密钥。

同时，开发者可选择消息加解密方式：明文模式、兼容模式和安全模式。明文模式就是我们自己的服务器收到微信服务器发来的消息是明文字符串，直接就可以读取并且解析，安全模式则是我们收到微信服务器发来的消息是加密的消息，需要我们手动解析后才能使用。

4. 开发

公众号后台配置完成后，接下来我们就可以写代码了。

4.1 服务器有效性校验

我们首先来创建一个普通的 Spring Boot 项目，创建时引入 `spring-boot-starter-web` 依赖，项目创建成功后，我们创建一个 Controller，添加如下接口：

```
@GetMapping("/verify_wx_token")
```

```

public void login(HttpServletRequest request, HttpServletResponse response)
throws UnsupportedEncodingException {
    request.setCharacterEncoding("UTF-8");
    String signature = request.getParameter("signature");
    String timestamp = request.getParameter("timestamp");
    String nonce = request.getParameter("nonce");
    String echostr = request.getParameter("echostr");
    PrintWriter out = null;
    try {
        out = response.getWriter();
        if (CheckUtil.checkSignature(signature, timestamp, nonce)) {
            out.write(echostr);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        out.close();
    }
}

```

关于这段代码，我做如下解释：

- 首先通过 `request.getParameter` 方法获取到微信服务器发来的 `signature`、`timestamp`、`nonce` 以及 `echostr` 四个参数，这四个参数中：`signature` 表示微信加密签名，`signature` 结合了开发者填写的 `token` 参数和请求中的 `timestamp` 参数、`nonce` 参数；`timestamp` 表示时间戳；`nonce` 表示随机数；`echostr` 则表示一个随机字符串。
- 开发者通过检验 `signature` 对请求进行校验，如果确认此次 GET 请求来自微信服务器，则原样返回 `echostr` 参数内容，则接入生效，成为开发者成功，否则接入失败。
- 具体的校验就是松哥这里的 `CheckUtil.checkSignature` 方法，在这个方法中，首先将 `token`、`timestamp`、`nonce` 三个参数进行字典序排序，然后将三个参数字符串拼接成一个字符串进行 `sha1` 加密，最后开发者获得加密后的字符串可与 `signature` 对比，标识该请求来源于微信。

校验代码如下：

```

public class CheckUtil {
    private static final String token = "123456";
    public static boolean checkSignature(String signature, String timestamp,
    String nonce) {
        String[] str = new String[]{token, timestamp, nonce};
        //排序
        Arrays.sort(str);
        //拼接字符串
        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < str.length; i++) {
            buffer.append(str[i]);
        }
        //进行sha1加密
        String temp = SHA1.encode(buffer.toString());
        //与微信提供的signature进行匹对
        return signature.equals(temp);
    }
}

public class SHA1 {
    private static final char[] HEX_DIGITS = {'0', '1', '2', '3', '4', '5',
        '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};
    private static String getFormattedText(byte[] bytes) {
        int len = bytes.length;

```

```

        StringBuilder buf = new StringBuilder(len * 2);
        for (int j = 0; j < len; j++) {
            buf.append(HEX_DIGITS[(bytes[j] >> 4) & 0x0f]);
            buf.append(HEX_DIGITS[bytes[j] & 0x0f]);
        }
        return buf.toString();
    }

    public static String encode(String str) {
        if (str == null) {
            return null;
        }
        try {
            MessageDigest messageDigest = MessageDigest.getInstance("SHA1");
            messageDigest.update(str.getBytes());
            return getFormattedText(messageDigest.digest());
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

```

OK，完成之后，我们的校验接口就算是开发完成了。接下来就可以开发消息接收接口了。

4.2 消息接收接口

接下来我们来开发消息接收接口，消息接收接口和上面的服务器校验接口地址是一样的，都是我们一开始在公众号后台配置的地址。只不过消息接收接口是一个 POST 请求。

我在公众号后台配置的时候，消息加解密方式选择了明文模式，这样我在后台收到的消息直接就可以处理了。微信服务器给我发来的普通文本消息格式如下：

```

<xml>
<ToUserName><! [CDATA[toUser]]></ToUserName>
<FromUserName><! [CDATA[fromUser]]></FromUserName>
<CreateTime>1348831860</CreateTime>
<MsgType><! [CDATA[text]]></MsgType>
<Content><! [CDATA[this is a test]]></Content>
<MsgId>1234567890123456</MsgId>
</xml>

```

这些参数含义如下：

参数	描述
ToUserName	开发者微信号
FromUserName	发送方帐号（一个OpenID）
CreateTime	消息创建时间（整型）
MsgType	消息类型，文本为text
Content	文本消息内容
MsgId	消息id，64位整型

看到这里，大家心里大概就有数了，当我们收到微信服务器发来的消息之后，我们就进行 XML 解析，提取出来我们需要的信息，去做相关的查询操作，再将查到的结果返回给微信服务器。

这里我们先来个简单的，我们将收到的消息解析并打印出来：

```
@PostMapping("/verify_wx_token")
public void handler(HttpServletRequest request, HttpServletResponse response)
throws Exception {
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    PrintWriter out = response.getWriter();
    Map<String, String> parseXml = MessageUtil.parseXml(request);
    String msgType = parseXml.get("MsgType");
    String content = parseXml.get("Content");
    String fromusername = parseXml.get("FromUserName");
    String tousername = parseXml.get("ToUserName");
    System.out.println(msgType);
    System.out.println(content);
    System.out.println(fromusername);
    System.out.println(tousername);
}
public static Map<String, String> parseXml(HttpServletRequest request) throws
Exception {
    Map<String, String> map = new HashMap<String, String>();
    InputStream inputStream = request.getInputStream();
    SAXReader reader = new SAXReader();
    Document document = reader.read(inputStream);
    Element root = document.getRootElement();
    List<Element> elementList = root.elements();
    for (Element e : elementList)
        map.put(e.getName(), e.getText());
    inputStream.close();
    inputStream = null;
    return map;
}
```

大家看到其实都是一些常规代码，没有什么难度。

做完这些之后，我们将项目打成 jar 包在服务器上部署启动。启动成功之后，确认微信的后台配置也没问题，我们就可以在公众号上发一条消息了，这样我们自己的服务端就会打印出来刚刚消息的信息。

好了，篇幅限制，今天就和大家先聊这么多，后面再聊不同消息类型的解析和消息的返回问题。

不知道小伙伴们看懂没？有问题欢迎留言讨论。

参考资料：微信开放文档

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

hello 各位小伙伴，今天我们来继续学习如何通过 Spring Boot 开发微信公众号。还没阅读过上篇文章的小伙伴建议先看看上文，有助于理解本文：

- [Spring Boot 开发微信公众号后台](#)

上篇文章中我们将微信服务器和我们自己的服务器对接起来了，并且在自己的服务器上也能收到微信服务器发来的消息，本文我们要看的就是如何给微信服务器回复消息。

消息分类

在讨论如何给微信服务器回复消息之前，我们需要先来了解下微信服务器发来的消息主要有哪些类型以及我们回复给微信的消息都有哪些类型。

在上文中大家了解到，微信发送来的 xml 消息中有一个 MsgType 字段，这个字段就是用来标记消息的类型。这个类型可以标记出这条消息是普通消息还是事件消息还是图文消息等。

普通消息主要是指：

- 文本消息
- 图片消息
- 语音消息
- 视频消息
- 小视频消息
- 地址位置消息
- 链接消息

不同的消息类型，对应不同的 MsgType，这里我还是以普通消息为例，如下：

消息类型	MsgType
文本消息	text
图片消息	image
语音消息	voice
视频消息	video
小视频消息	shortvideo
地址位置消息	location
链接消息	link

大家千万不要以为不同类型消息的格式是一样的，其实是不一样的，也就是说，MsgType 为 text 的消息和 MsgType 为 image 的消息，微信服务器发给我们的消息内容是不一样的，这样带来一个问题就是我无法使用一个 Bean 去接收不同类型的数据，因此这里我们一般使用 Map 接收即可。

这是消息的接收，除了消息的接收之外，还有一个消息的回复，我们回复的消息也有很多类型，可以回复普通消息，也可以回复图片消息，回复语音消息等，不同的回复消息我们可以进行相应的封装。因为不同的返回消息实例也是有一些共同的属性的，例如消息是谁发来的，发给谁，消息类型，消息 id 等，所以我们可以将这些共同的属性定义成一个父类，然后不同的消息再去继承这个父类。

返回消息类型定义

首先我们来定义一个公共的消息类型：

```
public class BaseMessage {  
    private String ToUserName;  
    private String FromUserName;  
    private long CreateTime;  
    private String MsgType;  
    private long MsgId;  
    //省略 getter/setter  
}
```

在这里：

- ToUserName 表示开发者的微信号
- FromUserName 表示发送方账号（用户的 OpenID）
- CreateTime 消息的创建时间
- MsgType 表示消息的类型
- MsgId 表示消息 id

这是我们的基本消息类型，就是说，我们返回给用户的消息，无论是什么类型的消息，都有这几个基本属性。然后在此基础上，我们再去扩展出文本消息、图片消息等。

我们来看下文本消息的定义：

```
public class TextMessage extends BaseMessage {  
    private String Content;  
    //省略 getter/setter  
}
```

文本消息在前面消息的基础上多了一个 Content 属性，因此文本消息继承自 BaseMessage，再额外添加一个 Content 属性即可。

其他的消息类型也是类似的定义，我就不一一列举了，至于其他消息的格式，大家可以参考微信开放文档 (<http://1t.click/aPK>)。

返回消息生成

消息类型的 Bean 定义完成之后，接下来就是将实体类生成 XML。

首先我们定义一个消息工具类，将常见的消息类型枚举出来：

```
/**  
 * 返回消息类型：文本  
 */  
public static final String RESP_MESSAGE_TYPE_TEXT = "text";  
/**  
 * 返回消息类型：音乐  
 */  
public static final String RESP_MESSAGE_TYPE_MUSIC = "music";  
/**  
 * 返回消息类型：图文  
 */  
public static final String RESP_MESSAGE_TYPE_NEWS = "news";  
/**  
 * 返回消息类型：图片  
 */
```

```
public static final String RESP_MESSAGE_TYPE_Image = "image";
/**
 * 返回消息类型: 语音
 */
public static final String RESP_MESSAGE_TYPE_Voice = "voice";
/**
 * 返回消息类型: 视频
 */
public static final String RESP_MESSAGE_TYPE_Video = "video";
/**
 * 请求消息类型: 文本
 */
public static final String REQ_MESSAGE_TYPE_TEXT = "text";
/**
 * 请求消息类型: 图片
 */
public static final String REQ_MESSAGE_TYPE_IMAGE = "image";
/**
 * 请求消息类型: 链接
 */
public static final String REQ_MESSAGE_TYPE_LINK = "link";
/**
 * 请求消息类型: 地理位置
 */
public static final String REQ_MESSAGE_TYPE_LOCATION = "location";
/**
 * 请求消息类型: 音频
 */
public static final String REQ_MESSAGE_TYPE_VOICE = "voice";
/**
 * 请求消息类型: 视频
 */
public static final String REQ_MESSAGE_TYPE_VIDEO = "video";
/**
 * 请求消息类型: 推送
 */
public static final String REQ_MESSAGE_TYPE_EVENT = "event";
/**
 * 事件类型: subscribe(订阅)
 */
public static final String EVENT_TYPE_SUBSCRIBE = "subscribe";
/**
 * 事件类型: unsubscribe(取消订阅)
 */
public static final String EVENT_TYPE_UNSUBSCRIBE = "unsubscribe";
/**
 * 事件类型: CLICK(自定义菜单点击事件)
 */
public static final String EVENT_TYPE_CLICK = "CLICK";
/**
 * 事件类型: VIEW(自定义菜单 URL 视图)
 */
public static final String EVENT_TYPE_VIEW = "VIEW";
/**
 * 事件类型: LOCATION(上报地理位置事件)
 */
public static final String EVENT_TYPE_LOCATION = "LOCATION";
/**
```

```
* 事件类型: LOCATION(上报地理位置事件)
*/
public static final String EVENT_TYPE_SCAN = "SCAN";
```

大家注意这里消息类型的定义，以 RESP 开头的表示返回的消息类型，以 REQ 表示微信服务器发来的消息类型。然后在这个工具类中再定义两个方法，用来将返回的对象转换成 XML：

```
public static String textMessageToXml(TextMessage textMessage) {
    xstream.alias("xml", textMessage.getClass());
    return xstream.toXML(textMessage);
}

private static XStream xstream = new XStream(new XppDriver() {
    public HierarchicalStreamWriter createWriter(Writer out) {
        return new PrettyPrintWriter(out) {
            boolean cdata = true;
            @SuppressWarnings("rawtypes")
            public void startNode(String name, Class clazz) {
                super.startNode(name, clazz);
            }
            protected void writeText(QuickWriter writer, String text) {
                if (cdata) {
                    writer.write("<![CDATA[");
                    writer.write(text);
                    writer.write("]]>");
                } else {
                    writer.write(text);
                }
            }
        };
    }
});
```

textMessageToXML 方法用来将 TextMessage 对象转成 XML 返回给微信服务器，类似的方法我们还需要定义 imageMessageToXml、voiceMessageToXml 等，不过定义的方式都基本类似，我就不一一列出来了。

返回消息分发

由于用户发来的消息可能存在多种情况，我们需要分类进行处理，这个就涉及到返回消息的分发问题。因此我在这里再定义一个返回消息分发的工具类，如下：

```
public class MessageDispatcher {
    public static String processMessage(Map<String, String> map) {
        String openid = map.get("FromUserName"); // 用户 openid
        String mpid = map.get("ToUserName"); // 公众号原始 ID
        if (map.get("MsgType").equals(MessageUtil.REQ_MESSAGE_TYPE_TEXT)) {
            // 普通文本消息
            TextMessage txtmsg = new TextMessage();
            txtmsg.setToUserName(openid);
            txtmsg.setFromUserName(mpid);
            txtmsg.setCreateTime(new Date().getTime());
            txtmsg.setMsgType(MessageUtil.RESP_MESSAGE_TYPE_TEXT);
            txtmsg.setContent("这是返回消息");
            return MessageUtil.textMessageToXml(txtmsg);
        }
    }
}
```

```
        return null;
    }
    public String processEvent(Map<String, String> map) {
        //在这里处理事件
    }
}
```

这里我们还可以多加几个 `elseif` 去判断不同的消息类型，我这里因为只有普通文本消息，所以一个 `if` 就够用了。

在这里返回值我写死了，实际上这里需要根据微信服务端传来的 Content 去数据中查询，将查询结果返回，数据库查询这一套相信大家都能搞定，我这里就不重复介绍了。

最后在消息接收 Controller 中调用该方法，如下：

```
@PostMapping(value = "/verify_wx_token", produces = "application/xml;charset=utf-8")
public String handler(HttpServletRequest request, HttpServletResponse response)
throws Exception {
    request.setCharacterEncoding("UTF-8");
    Map<String, String> map = MessageUtil.parseXml(request);
    String msgType = map.get("MsgType");
    if (MessageUtil.REQ_MESSAGE_TYPE_EVENT.equals(msgType)) {
        return messageDispatcher.processEvent(map);
    } else{
        return messageDispatcher.processMessage(map);
    }
}
```

在 Controller 中，我们首先判断消息是否是事件，如果是事件，进入到事件处理通道，如果不是事件，则进入到消息处理通道。

注意，这里需要配置一下返回消息的编码，否则可能会出现中文乱码。

如此之后，我们的服务器就可以给公众号返回消息了。

上篇文章发出后，有小伙伴问松哥这个会不会开源，我可以负责任的告诉大家，肯定会开源，这个系列截稿后，我把代码处理下就上传到 GitHub。

好了，本文我们就先说到这里。

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



云·空间·时间

什么是面霸？就是在面试中，神挡杀神佛挡杀佛，见招拆招，面到面试官自惭形秽自叹不如！松哥希望本文能成为你面霸路上的垫脚石！

做 Java 开发，没有人敢小觑 Spring Boot 的重要性，现在出去面试，无论多小的公司 or 项目，都要跟你扯一扯 Spring Boot，扯一扯微服务，不会？没用过？Sorry，我们不合适！

今天松哥就给大家整理了 15 道高频 Spring Boot 面试题，希望能够帮助到刚刚走出校门的小伙伴以及准备寻找新的工作机会的小伙伴。

- 1.什么是 Spring Boot ?

传统的 SSM/SSH 框架组合配置繁琐臃肿，不同项目有很多重复、模板化的配置，严重降低了 Java 工程师的开发效率，而 Spring Boot 可以轻松创建基于 Spring 的、可以独立运行的、生产级的应用程序。通过对 Spring 家族和一些第三方库提供一系列自动化配置的 Starter，来使得开发快速搭建一个基于 Spring 的应用程序。

Spring Boot 让日益臃肿的 Java 代码又重回简洁。在配合 Spring Cloud 使用时，还可以发挥更大的威力。

- 2.Spring Boot 有哪些特点？

Spring Boot 主要有如下特点：

1. 为 Spring 开发提供一个更快、更广泛的入门体验。
 2. 开箱即用，远离繁琐的配置。
 3. 提供了一系列大型项目通用的非业务性功能，例如：内嵌服务器、安全管理、运行数据监控、运行状况检查和外部化配置等。
 4. 绝对没有代码生成，也不需要XML配置。
- 3.Spring Boot 中的 starter 到底是什么？

首先，这个 Starter 并非什么新的技术点，基本上还是基于 Spring 已有功能来实现的。首先它提供了一个自动化配置类，一般命名为 `xxxAutoConfiguration`，在这个配置类中通过条件注解来决定一个配置是否生效（条件注解就是 Spring 中原本就有的），然后它还会提供一系列的默认配置，也允许开发者根据实际情况自定义相关配置，然后通过类型安全的属性注入将这些配置属性注入进来，新注入的属性会代替掉默认属性。正因为如此，很多第三方框架，我们只需要引入依赖就可以直接使用了。

当然，开发者也可以自定义 Starter，自定义 Starter 可以参考：[自定义 Spring Boot 中的 starter](#)。

- 4.spring-boot-starter-parent 有什么用？

我们都知道，新创建一个 Spring Boot 项目，默认都是有 parent 的，这个 parent 就是 `spring-boot-starter-parent`，`spring-boot-starter-parent` 主要有如下作用：

1. 定义了 Java 编译版本为 1.8 。
2. 使用 UTF-8 格式编码。
3. 继承自 `spring-boot-dependencies`，这个里边定义了依赖的版本，也正是因为继承了这个依赖，所以我们在写依赖时才不需要写版本号。
4. 执行打包操作的配置。
5. 自动化的资源过滤。
6. 自动化的插件配置。
7. 针对 `application.properties` 和 `application.yml` 的资源过滤，包括通过 profile 定义的不同环境的配置文件，例如 `application-dev.properties` 和 `application-dev.yml`。

关于这个问题，读者可以参考：[理解 spring-boot-starter-parent](#)

- 5.YAML 配置的优势在哪里？

YAML 现在可以算是非常流行的一种配置文件格式了，无论是前端还是后端，都可以见到 YAML 配置。那么 YAML 配置和传统的 properties 配置相比到底有哪些优势呢？

1. 配置有序，在一些特殊的场景下，配置有序很关键
2. 支持数组，数组中的元素可以是基本数据类型也可以是对象
3. 简洁

相比 properties 配置文件，YAML 还有一个缺点，就是不支持 @PropertySource 注解导入自定义的 YAML 配置。

关于 YAML 配置，要是大家还不熟悉，可以参考: [Spring Boot中的yaml配置](#)

- 6.Spring Boot 中如何解决跨域问题？

跨域可以在前端通过 JSONP 来解决，但是 JSONP 只可以发送 GET 请求，无法发送其他类型的请求，在 RESTful 风格的应用中，就显得非常鸡肋，因此我们推荐在后端通过（CORS, Cross-origin resource sharing）来解决跨域问题。这种解决方案并非 Spring Boot 特有的，在传统的 SSM 框架中，就可以通过 CORS 来解决跨域问题，只不过之前我们是在 XML 文件中配置 CORS，现在则是通过 @CrossOrigin 注解来解决跨域问题。关于 CORS，小伙伴们可以参考: [CORS 解决跨域问题](#)

- 7.比较一下 Spring Security 和 Shiro 各自的优缺点？

由于 Spring Boot 官方提供了大量的非常方便的开箱即用的 Starter，包括 Spring Security 的 Starter，使得在 Spring Boot 中使用 Spring Security 变得更加容易，甚至只需要添加一个依赖就可以保护所有的接口，所以，如果是 Spring Boot 项目，一般选择 Spring Security。当然这只是一个建议的组合，单纯从技术上来说，无论怎么组合，都是没有问题的。Shiro 和 Spring Security 相比，主要有如下一些特点：

1. Spring Security 是一个重量级的安全管理框架；Shiro 则是一个轻量级的安全管理框架
 2. Spring Security 概念复杂，配置繁琐；Shiro 概念简单、配置简单
 3. Spring Security 功能强大；Shiro 功能简单
- 8.微服务中如何实现 session 共享？

在微服务中，一个完整的项目被拆分成多个不相同的独立的服务，各个服务独立部署在不同的服务器上，各自的 session 被从物理空间上隔离开了，但是经常，我们需要在不同微服务之间共享 session，常见的方案就是 Spring Session + Redis 来实现 session 共享。将所有微服务的 session 统一保存在 Redis 上，当各个微服务对 session 有相关的读写操作时，都去操作 Redis 上的 session。这样就实现了 session 共享，Spring Session 基于 Spring 中的代理过滤器实现，使得 session 的同步操作对开发人员而言是透明的，非常简便。session 共享大家可以参考: [Spring Boot 整合 Session 共享](#)

- 9.Spring Boot 如何实现热部署？

Spring Boot 实现热部署其实很容易，引入 devtools 依赖即可，这样当编译文件发生变化时，Spring Boot 就会自动重启。在 Eclipse 中，用户按下保存按键，就会自动编译进而重启 Spring Boot，IDEA 中由于是自动保存的，自动保存时并未编译，所以需要开发者按下 Ctrl+F9 进行编译，编译完成后，项目就自动重启了。

如果仅仅只是页面模板发生变化，Java 类并未发生变化，此时可以不用重启 Spring Boot，使用 LiveReload 插件就可以轻松实现热部署。

- 10.Spring Boot 中如何实现定时任务？

定时任务也是一个常见的需求，Spring Boot 中对于定时任务的支持主要还是来自 Spring 框架。

在 Spring Boot 中使用定时任务主要有两种不同的方式，一个就是使用 Spring 中的 @Scheduled 注解，另一个则是使用第三方框架 Quartz。

使用 Spring 中的 @Scheduled 的方式主要通过 @Scheduled 注解来实现。

使用 Quartz，则按照 Quartz 的方式，定义 Job 和 Trigger 即可。

关于定时任务这一块，大家可以参考：[定时任务的两种实现方式](#)

- 11.前后端分离，如何维护接口文档？

前后端分离开发日益流行，大部分情况下，我们都是通过 Spring Boot 做前后端分离开发，前后端分离一定会有接口文档，不然会前后端会深深陷入到扯皮中。一个比较笨的方法就是使用 word 或者 md 来维护接口文档，但是效率太低，接口一变，所有人的文档都得变。在 Spring Boot 中，这个问题常见的解决方案是 Swagger，使用 Swagger 我们可以快速生成一个接口文档网站，接口一旦发生变化，文档就会自动更新，所有开发工程师访问这一个在线网站就可以获取到最新的接口文档，非常方便。关于 Swagger 的用法，大家可以参考：[SpringBoot 整合 Swagger2](#)

- 12.什么是 Spring Data？

Spring Data 是 Spring 的一个子项目。用于简化数据库访问，支持NoSQL 和 关系数据存储。其主要目标是使数据库的访问变得方便快捷。Spring Data 具有如下特点：

1. SpringData 项目支持 NoSQL 存储：
2. MongoDB（文档数据库）
3. Neo4j（图形数据库）
4. Redis（键/值存储）
5. Hbase（列族数据库）

SpringData 项目所支持的关系数据存储技术：

1. JDBC
2. JPA

Spring Data Jpa 致力于减少数据访问层 (DAO) 的开发量。开发者唯一要做的，就是声明持久层的接口，其他都交给 Spring Data JPA 来帮你完成！Spring Data JPA 通过规范方法的名字，根据符合规范的名字来确定方法需要实现什么样的逻辑。

- 13.Spring Boot 是否可以使用 XML 配置？

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

- 14.Spring Boot 打成的 jar 和普通的 jar 有什么区别？

Spring Boot 项目最终打包成的 jar 是可执行 jar，这种 jar 可以直接通过 `java -jar xxx.jar` 命令来运行，这种 jar 不可以作为普通的 jar 被其他项目依赖，即使依赖了也无法使用其中的类。

Spring Boot 的 jar 无法被其他项目依赖，主要还是他和普通 jar 的结构不同。普通的 jar 包，解压后直接就是包名，包里就是我们的代码，而 Spring Boot 打包成的可执行 jar 解压后，在 `\BOOT-INF\classes` 目录下才是我们的代码，因此无法被直接引用。如果非要引用，可以在 pom.xml 文件中增加配置，将 Spring Boot 项目打包成两个 jar，一个可执行，一个可引用。具体可以参考：[Spring Boot 可执行 jar 分析](#)

- 15.bootstrap.properties 和 application.properties 有何区别？

单纯做 Spring Boot 开发，可能不太容易遇到 bootstrap.properties 配置文件，但是在结合 Spring Cloud 时，这个配置就会经常遇到了，特别是在需要加载一些远程配置文件的时候。

bootstrap.properties 在 application.properties 之前加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。bootstrap.properties 被 Spring ApplicationContext 的父类加载，这个类先于加载 application.properties 的 ApplicationContext 启动。

当然，前面叙述中的 properties 也可以修改为 yaml。

好了，本文就说到这里，欢迎小伙伴留言说说你曾经遇到过的 Spring Boot 面试题！

关注微信公众号江南一点雨，回复 2TB，获取超 2TB 免费 Java 学习资源。



江南 · 一点雨