# 一、Spring Boot简介

Spring 诞生时是 Java 企业版（Java Enterprise Edition，JEE，也称 J2EE）的轻量级代替品。无需开发重量级的 Enterprise JavaBean（EJB），Spring 为企业级Java开发提供了一种相对简单的方法，通过依赖注入和面向切面编程，用简单的Java 对象（Plain Old Java Object，POJO）实现了 EJB 的功能。

虽然 Spring 的组件代码是轻量级的，但它的配置却是重量级的。

**第一阶段：xml配置。**

在Spring 1.x时代，使用Spring开发满眼都是xml配置的Bean，随着项目的扩大，我们需要把xml配置文件放到不同的配置文件里，那时需要频繁的在开发的类和配置文件之间进行切换。

**第二阶段：注解配置。**

在Spring 2.x 时代，随着JDK1.5带来的注解支持，Spring提供了声明Bean的注解（例如@Component、@Service），大大减少了配置量。主要使用的方式是应用的基本配置（如数据库配置）用xml，业务配置用注解。

**第三阶段：java配置。**

Spring 3.0 引入了基于 Java 的配置能力，这是一种类型安全的可重构配置方式，可以代替 XML。我们目前刚好处于这个时代，Spring4.x和Spring Boot都推荐使用Java配置。

所有这些配置都代表了开发时的损耗。因为在思考 Spring 特性配置和解决业务问题之间需要进行思维切换，所以写配置挤占了写应用程序逻辑的时间。除此之外，项目的依赖管理也是件吃力不讨好的事情。决定项目里要用哪些库就已经够让人头痛的了，你还要知道这些库的哪个版本和其他库不会有冲突，这难题实在太棘手。并且，依赖管理也是一种损耗，添加依赖不是写应用程序代码。一旦选错了依赖的版本，随之而来的不兼容问题毫无疑问会是生产力杀手。

Spring Boot 让这一切成为了过去。

Spring Boot 简化了基于Spring的应用开发，只需要“run”就能创建一个独立的、生产级别的Spring应用。Spring Boot为Spring平台及第三方库提供开箱即用的设置（提供默认设置），这样我们就可以简单的开始。多数Spring Boot应用只需要很少的Spring配置。

我们可以使用Spring Boot创建java应用，并使用java –jar 启动它，或者采用传统的war部署方式。

Spring Boot 主要目标是：

1、为所有 Spring 的开发提供一个从根本上更快的入门体验。

2、开箱即用，但通过自己设置参数，即可快速摆脱这种方式。

3、提供了一些大型项目中常见的非功能性特性，如内嵌服务器、安全、指标，健康检测、外部化配置等。

4、绝对没有代码生成，也无需 XML 配置。

# 二、入门程序

需求：实现Hello World的输出。

## 2.1、使用Spring MVC实现

我们现在开始使用spring MVC 框架，实现json 数据的输出。如果按照我们原来的做法，需要在web.xml 中添加一个DispatcherServlet 的配置，还需要添加一个spring的配置文件。

1、前端控制器的配置，在web.xml文件中：

<!-- 配置前端控制器 -->

<servlet>

<servlet-name>SpringMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- contextConfigLocation配置SpringMVC加载的配置文件（配置文件里面配置处理器适配器，处理器等等） -->

<!-- 如果不配置contextConfigLocation，则默认加载/WEB-INF/servlet名称-servlet.xml（SpringMVC-servlet.xml） -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:SpringMVC.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>SpringMVC</servlet-name>

<!-- 第一种： \*.action，所有以\*.action结尾的访问，都由DispatcherServlet解析-->

<url-pattern>\*.action</url-pattern>

</servlet-mapping>

2、springmvc配置

<!-- springmvc注解驱动 -->

<mvc:annotation-driven></mvc:annotation-driven>

<!-- 扫描controller注解，多个包中间使用半角逗号分隔 -->

<context:component-scan base-package=*"com.zxt.springmvc.controller"*/>

3、开发controller，一般controller返回ModelAndView对象，然后需要进行视图解析，将视图返回给前端。当需要返回JSON数据时，需要使用额外的json转换jar包。并在方法中使用@ResponseBody注解。

## 2.2、使用Spring Boot实现

当我们使用spring boot实现时，不需要配置文件，直接编写Controller类即可。

@RestController

@RequestMapping(value="/index")

**public** **class** IndexController {

@RequestMapping

**public** String index() {

**return** "Hello World!!!";

}

/\*\*

\* **@param** name

\* **@return** 返回的数据直接转换成了JSON格式

\*/

@RequestMapping(value="get")

**public** Map<String, Object> get(@RequestParam String name) {

Map<String, Object> map = **new** HashMap<String, Object>();

map.put("name", name);

map.put("value", "Hello World!!!");

map.put("secret", secret);

map.put("number", number);

**return** map;

}

/\*\*

\* **@param** id

\* **@param** name

\* **@return** 尝试返回Java bean对象，在客户端调用的时候依旧会把数据转换成JSON格式

\*/

@RequestMapping(value="find/{id}/{name}")

**public** User find(@PathVariable **int** id, @PathVariable String name) {

User user = **new** User();

user.setId(id);

user.setName(name);

user.setDate(**new** Date());

**return** user;

}

}

@RestController注解:其实就是@Controller和@ResponseBody注解加在一起。

启动方式一：启动之前自动生成的引导类即可。

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

**public** **class** SpringBootFirstDemoApplication {

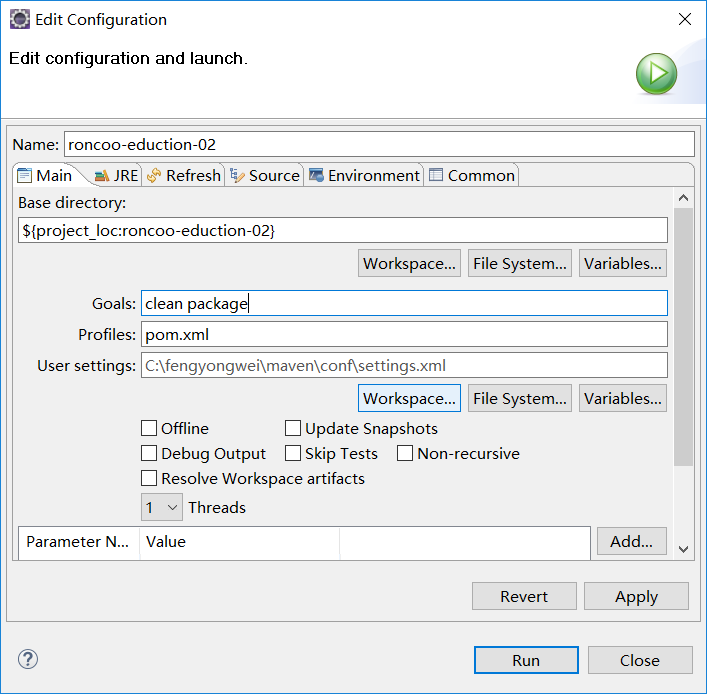
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootFirstDemoApplication.**class**, args);

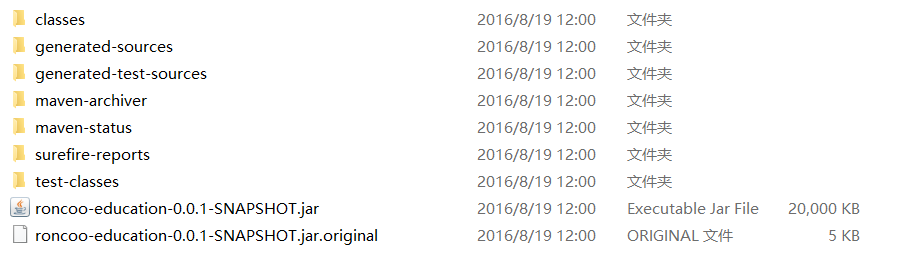
}

}

启动方式二：使用Maven命令spring-boot:run执行即可。选中项目，鼠标右键选择Maven Build，命令: clean package。



则在项目的target目录下回生成两个jar文件，一个是包含项目运行所需所有依赖jar包的完整jar，另一个是不包含依赖jar包的jar文件。



运行命令: java –jar roncoo-education-0.0.1-SNAPSHOT.jar，即可启动项目。在浏览器地址栏输入http://localhost:8080/index 即可看到运行结果。

## 2.3、热部署

我们在开发中反复修改类、页面等资源，每次修改后都是需要重新启动才生效，这样每次启动都很麻烦，浪费了大量的时间。

能不能在我修改代码后不重启就能生效呢？可以，由于Spring Boot应用只是普通的Java应用，所以JVM热交换（hot-swapping）也能开箱即用。不过JVM热交换能替换的字节码有限制，想要更彻底的解决方案可以使用Spring Loaded项目或JRebel。spring-boot-devtools模块也支持应用快速重启(restart)。

我们只需要在pom.xml加入如下配置即可：

<dependency>

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

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

</dependency>

# 三、Spring Boot配置文件

## 3.1、配置文件

1、**配置文件的生效顺序，会对值进行覆盖。**

1）、@TestPropertySource 注解；

2）、命令行参数；

3）、Java系统属性（System.getProperties()）；

4）、操作系统环境变量；

5）、只有在random.\*里包含的属性会产生一个RandomValuePropertySource；

6）、在打包的jar外的应用程序配置文件（application.properties，包含YAML和profile变量）；

7）、在打包的jar内的应用程序配置文件（application.properties，包含YAML和profile变量）；

8）、在@Configuration类上的@PropertySource注解；

9）、默认属性（使用SpringApplication.setDefaultProperties指定）。

**2、配置随机值**

roncoo.secret = ${random.value}

roncoo.number = ${random.int}

roncoo.bignumber = ${random.long}

roncoo.number.less.than.ten = ${random.int(10)}

roncoo.number.in.range = ${random.int[1024,65536]}

读取使用注解：@Value(value = "${roncoo.secret}")

注：出现黄点提示，是要提示配置元数据，可以不配置。

3、**配置应用端口和其他配置的介绍**

# 端口配置

server.port=8090

# 时间格式化

spring.jackson.date-format=yyyy-MM-dd HH:mm:ss

# 时区设置

spring.jackson.time-zone=Asia/Chongqing

**4、属性占位符**

当application.properties里的值被使用时，它们会被存在的Environment过滤，所以你能够引用先前定义的值（比如，系统属性）。

roncoo.name=www.roncoo.com

roncoo.desc=${roncoo.name} is a domain name

**5、Application属性文件，按优先级排序，位置高的将覆盖位置低的**

1）、当前目录下的一个/config子目录；

2）、当前目录；

3）、一个classpath下的/config包；

4）、classpath根路径（root）。

这个列表是按优先级排序的（列表中位置高的将覆盖位置低的）。

**6、使用YAML代替Properties**

Spring Boot可以使用YAML格式的配置文件，注意写法：冒号后要加个空格。

## 3.2、多环境配置

**1、多环境配置的好处**

1）、不同环境配置可以配置不同的参数；

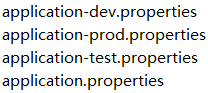
2）、便于部署，提高效率，减少出错。

**2、Properties多环境配置**

1）、配置激活选项

spring.profiles.active=dev

2）、添加其他配置文件



**3、YAML多环境配置**

1）、配置激活选项

spring:

profiles:

active: dev

2）、在配置文件添加三个英文状态下的短横线即可区分

---

spring:

profiles: dev

**4、两种配置方式的比较**

1）、Properties配置多环境，需要添加多个配置文件，YAML只需要一个配件文件；

2）、书写格式的差异，yaml相对比较简洁，优雅。

3）、YAML的缺点：不能通过@PropertySource注解加载。如果需要使用@PropertySource注解的方式加载值，那就要使用properties文件。

**5、修改打包程序的配置文件**

当程序打包之后，可以使用：java -jar myapp.jar --spring.profiles.active=dev，来修改已经打包的程序的配置文件选项。

## 3.3、日志配置

Spring Boot支持的日志框架有：Java Util Logging，Log4J2 and Logback，默认是使用logback。

配置方式：默认配置文件配置和引用外部配置文件配置。

**1、默认配置文件配置(不建议使用：不够灵活，对log4j2等不够友好)**

# 日志文件名，比如：roncoo.log，或者是 /var/log/roncoo.log

logging.file=roncoo.log

# 日志级别配置，比如：logging.level.org.springframework=DEBUG

logging.level.\*=info

logging.level.org.springframework=DEBUG

**2、引用外部配置文件**

**logback配置方式：**

spring boot默认会加载classpath:logback-spring.xml或者classpath:logback-spring.groovy文件。

1）、使用自定义配置文件，配置方式为：

logging.config=classpath:logback-roncoo.xml



注意：不要使用logback这个来命名，否则spring boot将不能完全实例化。

2）、使用基于spring boot的配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<configuration>

<include resource=*"org/springframework/boot/logging/logback/base.xml"*/>

<logger name=*"org.springframework.web"* level=*"DEBUG"*/>

</configuration>

3、**log4j配置**

1）、去除logback的依赖包，添加log4j2的依赖包。

<!-- 去除logback的依赖包 -->

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

<!-- 使用log4j2 -->

<dependency>

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

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

2）、在classpath路径下添加log4j2.xml或者log4j2-spring.xml文件（spring boot 默认加载）。

3）、自定义配置文件



**4、比较**

性能比较：Log4J2 和 Logback 都优于 log4j（不推荐使用）。

配置方式：Logback最简洁，spring boot默认，推荐使用。

# 四、Spring Boot web应用开发

## 4.1、web应用开发-模板引擎

spring boot的web应用开发，是基于spring mvc。Spring boot 在spring默认基础上，自动配置添加了以下特性：

1）、包含了ContentNegotiatingViewResolver和BeanNameViewResolver beans。

2）、对静态资源的支持，包括对WebJars的支持。

3）、自动注册Converter，GenericConverter，Formatter beans。

4）、对HttpMessageConverters的支持。

5）、自动注册MessageCodeResolver。

6）、对静态index.html的支持。

7）、对自定义Favicon的支持。

8）、主动使用ConfigurableWebBindingInitializer bean。

**1、模板引擎的选择**

**1）、FreeMarker**

**2）、Thymeleaf**

3）、Velocity (1.4版本之后弃用，Spring Framework 4.3版本之后弃用)

4）、Groovy

5）、Mustache

注：jsp应该尽量避免使用，原因如下：

1）、jsp只能打包为：war格式，不支持jar格式，只能在标准的容器里面跑（tomcat，jetty都可以）。

2）、内嵌的Jetty目前不支持JSPs。

3）、Undertow不支持jsps。

4）、jsp自定义错误页面不能覆盖spring boot 默认的错误页面。

**2、FreeMarker Demo**

<dependency>

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

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

</dependency>

**3、Thymeleaf Demo**

<dependency>

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

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

</dependency>

**4、Jsp Demo**

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

添加配置参数：

spring.mvc.view.prefix: /WEB-INF/templates/

spring.mvc.view.suffix: .jsp

## 4.2、web应用开发-错误处理

方法一：Spring Boot 将所有的错误默认映射到/error，实现**ErrorController**。

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.boot.web.servlet.error.ErrorController;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

@Controller

@RequestMapping("error")

**public** **class** BaseErrorController **implements** ErrorController {

**private** Logger logger = LoggerFactory.*getLogger*(BaseErrorController.**class**);

@Override

**public** String getErrorPath() {

logger.info("出错啦！进入自定义错误控制器");

**return** "error/error";

}

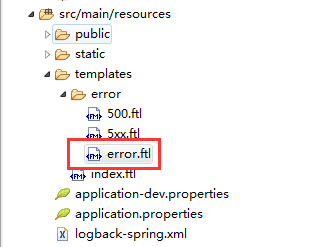
@RequestMapping

**public** String error() {

**return** getErrorPath();

}

}



**方法二：添加自定义的错误页面。**

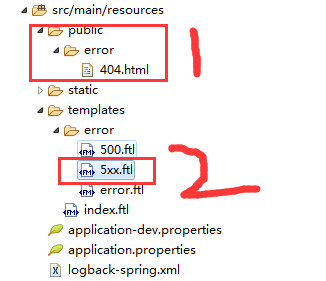
1、html静态页面：在resources/public/error/ 下定义。

如添加404页面： resources/public/error/404.html页面，中文注意页面编码。

2、模板引擎页面：在templates/error/下定义。

如添加5xx页面： templates/error/5xx.ftl。

注：templates/error/ 这个的优先级比较 resources/public/error/高。



**方法三：使用注解@ControllerAdvice**

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.http.HttpStatus;

**import** org.springframework.web.bind.annotation.ControllerAdvice;

**import** org.springframework.web.bind.annotation.ExceptionHandler;

**import** org.springframework.web.bind.annotation.ResponseStatus;

**import** org.springframework.web.servlet.ModelAndView;

@ControllerAdvice

**public** **class** BizExceptionHandler {

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

/\*\*

\* **@Description**：统一异常处理

\* **@param** exception

\* **@return**

\*/

@ExceptionHandler({ RuntimeException.**class** })

@ResponseStatus(HttpStatus.***OK***)

**public** ModelAndView processException(RuntimeException exception) {

***logger***.info("自定义异常处理-RuntimeException");

ModelAndView m = **new** ModelAndView();

m.addObject("roncooException", exception.getMessage());

System.***out***.println(exception.getMessage());

m.setViewName("error/500");

**return** m;

}

/\*\*

\* **@Description**：统一异常处理

\* **@param** exception

\* **@return**

\*/

@ExceptionHandler({ Exception.**class** })

@ResponseStatus(HttpStatus.***OK***)

**public** ModelAndView processException(Exception exception) {

***logger***.info("自定义异常处理-Exception");

ModelAndView m = **new** ModelAndView();

m.addObject("roncooException", exception.getMessage());

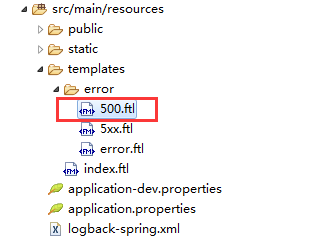
System.***out***.println(exception.getMessage());

m.setViewName("error/500");

**return** m;

}

}



## 4.3、web应用开发-Servlets，Filters，listeners

Web 开发使用 Controller 基本上可以完成大部分需求，但是我们还可能会用到 Servlet、 Filter、Listener 等等。

首先定义这三种类：

1、Servlet

**import** java.io.IOException;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** CustomServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Override

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

System.***out***.println("servlet get method");

doPost(request, response);

}

@Override

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

System.***out***.println("servlet post method");

response.getWriter().write("CustomServlet hello world");

}

}

2、Filter

**import** java.io.IOException;

**import** javax.servlet.Filter;

**import** javax.servlet.FilterChain;

**import** javax.servlet.FilterConfig;

**import** javax.servlet.ServletException;

**import** javax.servlet.ServletRequest;

**import** javax.servlet.ServletResponse;

**public** **class** CustomFilter **implements** Filter {

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

System.***out***.println("init CustomFilter");

}

@Override

**public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

**throws** IOException, ServletException {

System.***out***.println("do CustomFilter");

chain.doFilter(request, response);

}

@Override

**public** **void** destroy() {

System.***out***.println("destroy CustomFilter");

}

}

3、Listener

**import** javax.servlet.ServletContextEvent;

**import** javax.servlet.ServletContextListener;

**public** **class** CustomListener **implements** ServletContextListener {

@Override

**public** **void** contextInitialized(ServletContextEvent sce) {

System.***out***.println("contextInitialized");

}

@Override

**public** **void** contextDestroyed(ServletContextEvent sce) {

System.***out***.println("contextDestroyed");

}

}

4、在Spring Boot中注册它们的三种方式。

方法一：通过注册 ServletRegistrationBean、 FilterRegistrationBean 和

ServletListenerRegistrationBean 获得控制。（在启动类中）

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.boot.web.servlet.FilterRegistrationBean;

**import** org.springframework.boot.web.servlet.ServletListenerRegistrationBean;

**import** org.springframework.boot.web.servlet.ServletRegistrationBean;

**import** org.springframework.context.annotation.Bean;

**import** com.zxt.demo.filter.CustomFilter;

**import** com.zxt.demo.listener.CustomListener;

**import** com.zxt.demo.servlet.CustomServlet;

@SpringBootApplication

**public** **class** SpringBootDemoServletFilterListenerApplication {

@Bean

**public** ServletRegistrationBean<CustomServlet> servletRegistrationBean() {

**return** **new** ServletRegistrationBean<CustomServlet>(**new** CustomServlet(), "/servlet");

}

@Bean

**public** FilterRegistrationBean<CustomFilter> filterRegistrationBean() {

// CustomFilter只拦截servletRegistrationBean()中注册的Bean，如果拦截所有，则不要第二个参数即可

**return** **new** FilterRegistrationBean<CustomFilter>(**new** CustomFilter(), servletRegistrationBean());

}

@Bean

**public** ServletListenerRegistrationBean<CustomListener> servletListenerRegistrationBean() {

**return** **new** ServletListenerRegistrationBean<CustomListener>(**new** CustomListener());

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootDemoServletFilterListenerApplication.**class**, args);

}

}

方法二：通过实现 ServletContextInitializer 接口直接注册。

**import** java.util.EnumSet;

**import** javax.servlet.DispatcherType;

**import** javax.servlet.ServletContext;

**import** javax.servlet.ServletException;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.boot.web.servlet.ServletContextInitializer;

**import** com.zxt.demo.filter.CustomFilter;

**import** com.zxt.demo.listener.CustomListener;

**import** com.zxt.demo.servlet.CustomServlet;

@SpringBootApplication

**public** **class** SpringBootDemoServletFilterListenerApplication **implements** ServletContextInitializer {

@Override

**public** **void** onStartup(ServletContext servletContext) **throws** ServletException {

servletContext.addServlet("customServlet", **new** CustomServlet()).addMapping("/servlet");

servletContext.addFilter("customFilter", **new** CustomFilter())

.addMappingForServletNames(EnumSet.*of*(DispatcherType.***REQUEST***), **true**, "customServlet");

servletContext.addListener(**new** CustomListener());

}

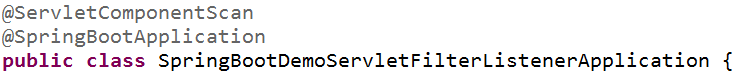
**public** **static** **void** main(String[] args) {

SpringApplication.*run*(SpringBootDemoServletFilterListenerApplication.**class**, args);

}

}

方法三：在 SpringBootApplication 上使用@ServletComponentScan 注解后，直接通过@WebServlet、@WebFilter、@WebListener 注解自动注册。









## 4.4、web应用开发-CORS支持

同源策略[same origin policy]是浏览器的一个安全功能，不同源的客户端脚本在没有明确授权的情况下，不能读写对方资源。同源策略是浏览器安全的基石。

什么是源：**源[origin]就是协议、域名和端口号**。例如：http://www.baidu.com:80这个URL。

什么是同源：若地址里面的协议、域名和端口号均相同则属于同源。

是否是同源的判断，例如判断下面的URL是否与 http://www.a.com/test/index.html 同源：

http://www.a.com/dir/page.html 同源；

http://www.child.a.com/test/index.html 不同源，域名不相同；

https://www.a.com/test/index.html 不同源，协议不相同；

http://www.a.com:8080/test/index.html 不同源，端口号不相同。

哪些操作不受同源策略限制：页面中的链接，重定向以及表单提交是不会受到同源策略限制的；跨域资源的引入是可以的。但是JS不能读写加载的内容。如嵌入到页面中的<script src="..."></script>，<img>，<link>，<iframe>等。

**跨域，**受前面所讲的浏览器同源策略的影响，不是同源的脚本不能操作其他源下面的对象。想要操作另一个源下的对象就需要跨域。在同源策略的限制下，非同源的网站之间不能发送 AJAX 请求。

**如何跨域**

**降域：**可以通过设置 document.damain='a.com'，浏览器就会认为它们都是同一个源。想要实现以上任意两个页面之间的通信，两个页面必须都设置documen.damain='a.com'。

**JSONP跨域**

**CORS 跨域**

### 4.4.1、CORS 简介

为了解决浏览器同源问题，W3C 提出了跨源资源共享，即 CORS(Cross-Origin Resource Sharing)。

CORS 做到了如下两点：

1、不破坏即有规则；

2、服务器实现了 CORS 接口，就可以跨源通信。

基于这两点，CORS 将请求分为两类：简单请求和非简单请求。

1、简单请求：在CORS出现前，发送HTTP请求时在头信息中不能包含任何自定义字段，且 HTTP 头信息不超过以下几个字段：

Accept

Accept-Language

Content-Language

Last-Event-ID

Content-Type：只限于 [application/x-www-form-urlencoded 、multipart/form-data、text/plain ] 类型。

一个简单的请求例子：

GET /test HTTP/1.1

Accept: \*/\*

Accept-Encoding: gzip, deflate, sdch, br

Origin: http://www.examples.com

Host: www.examples.com

对于简单请求，CORS的策略是请求时在请求头中增加一个Origin字段，服务器收到请求后，根据该字段判断是否允许该请求访问。

如果允许，则在 HTTP 头信息中添加 Access-Control-Allow-Origin 字段，并返回正确的结果；如果不允许，则不在 HTTP 头信息中添加 Access-Control-Allow-Origin 字段。

除了上面提到的 Access-Control-Allow-Origin，还有几个字段用于描述 CORS 返回结果：

Access-Control-Allow-Credentials：可选，用户是否可以发送、处理 cookie；

Access-Control-Expose-Headers：可选，可以让用户拿到的字段。有几个字段无论设置与否都可以拿到的，包括：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。

2、非简单请求

对于非简单请求的跨源请求，浏览器会在真实请求发出前，增加一次OPTION请求，称为预检请求(preflight request)。预检请求将真实请求的信息，包括请求方法、自定义头字段、源信息添加到 HTTP 头信息字段中，询问服务器是否允许这样的操作。

例如一个DELETE请求：

OPTIONS /test HTTP/1.1

Origin: http://www.examples.com

Access-Control-Request-Method: DELETE

Access-Control-Request-Headers: X-Custom-Header

Host: www.examples.com

与 CORS 相关的字段有：

请求使用的 HTTP 方法 Access-Control-Request-Method；

请求中包含的自定义头字段 Access-Control-Request-Headers。

服务器收到请求时，需要分别对Origin、Access-Control-Request-Method、Access-Control-Request-Headers 进行验证，验证通过后，会在返回 HTTP头信息中添加：

Access-Control-Allow-Origin: http://www.examples.com

Access-Control-Allow-Methods: GET, POST, PUT, DELETE

Access-Control-Allow-Headers: X-Custom-Header

Access-Control-Allow-Credentials: true

Access-Control-Max-Age: 1728000

他们的含义分别是：

Access-Control-Allow-Methods: 真实请求允许的方法

Access-Control-Allow-Headers: 服务器允许使用的字段

Access-Control-Allow-Credentials: 是否允许用户发送、处理 cookie

Access-Control-Max-Age: 预检请求的有效期，单位为秒。有效期内，不会重复发送预检请求。

当预检请求通过后，浏览器会发送真实请求到服务器。这就实现了跨源请求。

### 4.4.2、CORS 与 JSONP 比较

1、JSONP 只能实现 GET 请求，而 CORS 支持所有类型的 HTTP 请求。

2、使用 CORS，开发者可以使用普通的 XMLHttpRequest 发起请求和获得数据，比起 JSONP 有更好的错误处理。

3、JSONP 主要被老的浏览器支持，它们往往不支持 CORS，而绝大多数现代浏览器都已经支持了 CORS。（浏览器支持情况：Chrome 3+，Firefox 3.5+，Opera 12+，Safari 4+，Internet Explorer 8+）。

### 4.4.3、Spring Boot下CORS 的实现

1、假设本项目的端口是8089，其中有一个Controller。

**import** java.util.HashMap;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/index")

**public** **class** IndexController {

@RequestMapping(value = "get")

**public** HashMap<String, Object> get(@RequestParam String name) {

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("title", "hello world");

map.put("name", name);

**return** map;

}

}

2、在另一个端口为8080的项目中，有如下ajax请求：

<script>

$(function(){

$('#title').click(function(){

// alert('点击了');

$.ajax({

url: "http://localhost:8089/index/get",

type: "POST",

data: {

name: "测试"

},

success: function(data, status, xhr) {

console.log(data);

alert(data.name);

}

});

});

})

</script>

3、正常情况下，上述的ajax请求是无效的，在 spring boot 中可以配置全局的规则，也可以使用@CrossOrigin 注解进行细粒度的配置。

3.1、使用@CrossOrigin 注解实现

如果想要对某一接口配置 CORS，可以在方法上添加 @CrossOrigin 注解（即允许http://localhost:8080跨域访问）：

@RestController

@RequestMapping("/index")

**public** **class** IndexController {

@CrossOrigin(origins = "http://localhost:8080")

@RequestMapping(value = "get")

**public** HashMap<String, Object> get(@RequestParam String name) {

HashMap<String, Object> map = **new** HashMap<String, Object>();

map.put("title", "hello world");

map.put("name", name);

**return** map;

}

}

如果想对一系列接口添加 CORS 配置，可以在类上添加注解，对该类声明所有接口都有效。

3.2、如果想添加全局配置，则需要添加一个配置类 ：

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.config.annotation.CorsRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

**public** **class** CustomCorsConfiguration {

@Bean

**public** WebMvcConfigurer corsConfigurer() {

**return** **new** WebMvcConfigurer() {

@Override

**public** **void** addCorsMappings(CorsRegistry registry) {

registry.addMapping("/index/\*\*").allowedOrigins("http://localhost:8080");

}

};

}

}

或者：

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.web.servlet.config.annotation.CorsRegistry;

**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

**public** **class** CustomCorsConfiguration2 **implements** WebMvcConfigurer {

@Override

**public** **void** addCorsMappings(CorsRegistry registry) {

registry.addMapping("/index/\*\*").allowedOrigins("http://localhost:8080");

}

}

注意：早期的WebMvcConfigurerAdapter已经不建议使用。

### 4.4.4、CORS原理剖析

无论是通过哪种方式配置 CORS，其实都是在构造 CorsConfiguration。 一个 CORS配置用一个 CorsConfiguration类来表示，它的定义如下：

**public** **class** CorsConfiguration {

**private** List<String> allowedOrigins;

**private** List<String> allowedMethods;

**private** List<String> allowedHeaders;

**private** List<String> exposedHeaders;

**private** Boolean allowCredentials;

**private** Long maxAge;

}

Spring 中对 CORS 规则的校验，都是通过委托给 DefaultCorsProcessor实现的。

DefaultCorsProcessor 处理过程如下：

1、判断依据是 Header中是否包含 Origin。如果包含则说明为 CORS请求，转到 2；否则，说明不是 CORS 请求，不作任何处理。

2、判断 response 的 Header 是否已经包含 Access-Control-Allow-Origin，如果包含，证明已经被处理过了，转到 3，否则不再处理。

3、判断是否同源，如果是则转交给负责该请求的类处理。

4、是否配置了 CORS 规则，如果没有配置，且是预检请求，则拒绝该请求，如果没有配置，且不是预检请求，则交给负责该请求的类处理。如果配置了，则对该请求进行校验。校验就是根据 CorsConfiguration 这个类的配置进行判断：

判断 origin 是否合法；

判断 method 是否合法；

判断 header是否合法。

如果全部合法，则在 response header中添加响应的字段，并交给负责该请求的类处理，如果不合法，则拒绝该请求。

## 4.5、web应用开发-文件上传

Spring Boot 默认使用 springMVC 包装好的解析器进行文件上传。

1、添加文件上传的表单

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

文件：<input type="file" name="zfile" />

<input type="submit" value="上传" />

</form>

2、文件上传的Controller

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.util.ResourceUtils;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.ResponseBody;

**import** org.springframework.web.multipart.MultipartFile;

@Controller

@RequestMapping(value = "/file")

**public** **class** FileController {

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

@RequestMapping(value = "upload")

@ResponseBody

**public** String upload(@RequestParam("zfile") MultipartFile file) {

**if** (file.isEmpty()) {

**return** "文件为空";

}

// 获取文件名

String fileName = file.getOriginalFilename();

***logger***.info("上传的文件名为：" + fileName);

// 获取文件的后缀名

String suffixName = fileName.substring(fileName.lastIndexOf("."));

***logger***.info("上传的后缀名为：" + suffixName);

/\*\*

\* // 文件上传路径

\* String filePath = "d:/roncoo/ttt/";

\* // 解决中文问题，liunx 下中文路径，图片显示问题

\* // fileName = UUID.randomUUID() + suffixName;

\* File dest = new File(filePath + fileName);

\*/

File path = **null**;

**try** {

// 获取根目录

path = **new** File(ResourceUtils.*getURL*("classpath:").getPath());

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**if**(!path.exists()) {

path = **new** File("");

}

File upload = **new** File(path.getAbsolutePath(),"static/upload/" + fileName);

// 检测是否存在目录

**if** (!upload.getParentFile().exists()) {

upload.getParentFile().mkdirs();

}

**try** {

file.transferTo(upload);

**return** "上传成功";

} **catch** (Exception e) {

e.printStackTrace();

}

**return** "上传失败";

}

}

3、上传文件的一些配置

#设置静态资源路径，多个以逗号分隔

spring.resources.static-locations=classpath:static/,file:static/

# 默认支持文件上传.

spring.servlet.multipart.enabled=true

# 最大支持文件大小

spring.servlet.multipart.maxFileSize=5MB

# 最大支持请求大小

spring.servlet.multipart.maxRequestSize=30MB

# 五、Spring Boot 数据库操作

## 5.1、JdbcTemplate

1、首先创建spring\_boot\_demo数据库，并新建user表。



CREATE TABLE `user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`name` varchar(255) DEFAULT NULL,

`create\_time` datetime DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8 COMMENT='用户表';

2、配置数据源。

嵌入式数据库的支持：Spring Boot 可以自动配置 H2，HSQL and Derby 数据库，不需要提供任何的链接 URLs，只需要加入相应的 jar 包，Spring boot 可以自动发现装配。

<!-- 数据库 -->

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

application.properties配置文件配置数据源：

spring.datasource.url=jdbc:mysql://localhost/spring\_boot\_demo?useUnicode=true&characterEncoding=utf-8

spring.datasource.username=root

spring.datasource.password=root

# 这个驱动会自动加载（会根据url来判断需要什么驱动），不需要配置

# spring.datasource.driver-class-name=com.mysql.jdbc.Driver

注意：

1）、可以不指定driver-class-name，spring boot会自动识别 url。

2）、数据连接池默认使用 tomcat-jdbc，连接池的配置： spring.datasource.tomcat.\*。

3、创建实体类。

**import** java.util.Date;

**public** **class** User {

**private** **int** id;

**private** String name;

**private** Date create\_time;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Date getCreate\_time() {

**return** create\_time;

}

**public** **void** setCreate\_time(Date create\_time) {

**this**.create\_time = create\_time;

}

@Override

**public** String toString() {

**return** "User [id=" + id + ", name=" + name + ", create\_time=" + create\_time + "]";

}

}

4、Dao接口

/\*\*

\* **@Description**: 对数据库表的增删查改

\* **@author**： zxt

\* **@time**: 2019年5月24日 下午8:15:28

\*/

**public** **interface** UserDao {

**int** insert(User user);

**int** deleteById(**int** id);

**int** updateById(User user);

User selectById(**int** id);

}

5、使用JdbcTemplate实现对数据库的操作。(@Autowired自动装配)

@Repository("userDao")

**public** **class** UserDaoImpl **implements** UserDao {

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Override

**public** **int** insert(User user) {

String sql = "insert into user(name, create\_time) values(?, ?)";

**return** jdbcTemplate.update(sql, user.getName(), user.getCreate\_time());

}

@Override

**public** **int** deleteById(**int** id) {

String sql = "delete from user where id = ?";

**return** jdbcTemplate.update(sql, id);

}

@Override

**public** **int** updateById(User user) {

String sql = "update user set name = ?, create\_time = ? where id = ?";

**return** jdbcTemplate.update(sql, user.getName(), user.getCreate\_time(), user.getId());

}

@Override

**public** User selectById(**int** id) {

String sql = "select \* from user where id = ?";

**return** jdbcTemplate.queryForObject(sql, **new** RowMapper<User>() {

@Override

**public** User mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

User user = **new** User();

user.setId(rs.getInt("id"));

user.setName(rs.getString("name"));

user.setCreate\_time(rs.getDate("create\_time"));

**return** user;

}

}, id);

}

}

6、测试。

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** SpringBootDemoMysqlJdbctemplateApplicationTests {

@Autowired

**private** UserDao userDao;

@Test

**public** **void** insert() {

User user = **new** User();

user.setName("测试用户");

user.setCreate\_time(**new** Date());

**int** result = userDao.insert(user);

System.***out***.println(result);

}

@Test

**public** **void** deleteById() {

**int** result = userDao.deleteById(1);

System.***out***.println(result);

}

@Test

**public** **void** updateById() {

User user = **new** User();

user.setId(2);

user.setName("测试用户修改");

user.setCreate\_time(**new** Date());

**int** result = userDao.updateById(user);

System.***out***.println(result);

}

@Test

**public** **void** selectById() {

User user = userDao.selectById(2);

System.***out***.println(user);

}

}

7、如果希望在调试的时候打印出数据库操作的sql语句，则在日志配置文件中添加：

<logger name=*"org.springframework.jdbc.core.JdbcTemplate"* level=*"debug"*/>

8、封装Spring Boot的Jdbc，并实现分页。



## 5.2、spring-data-jap

1、配置数据源

<!-- 数据库 -->

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<scope>runtime</scope>

</dependency>

application.properties配置文件配置数据源：

# 数据源

spring.datasource.url=jdbc:mysql://localhost/spring\_boot\_demo?useUnicode=true&characterEncoding=utf-8

spring.datasource.username=root

spring.datasource.password=root

# JPA

spring.jpa.hibernate.ddl-auto=update

# 显示 sql 语句

spring.jpa.show-sql=true

2、创建实体类

**import** java.util.Date;

**import** javax.persistence.Column;

**import** javax.persistence.Entity;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.Id;

@Entity

**public** **class** UserLog {

@Id

@GeneratedValue

**private** Integer id;

@Column

**private** String userName;

@Column

**private** Date createTime;

@Column

**private** String userIp;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** Date getCreateTime() {

**return** createTime;

}

**public** **void** setCreateTime(Date createTime) {

**this**.createTime = createTime;

}

**public** String getUserName() {

**return** userName;

}

**public** **void** setUserName(String userName) {

**this**.userName = userName;

}

**public** String getUserIp() {

**return** userIp;

}

**public** **void** setUserIp(String userIp) {

**this**.userIp = userIp;

}

@Override

**public** String toString() {

**return** "RoncooUserLog [id=" + id + ", createTime=" + createTime + ", userName=" + userName + ", userIp="

+ userIp + "]";

}

}

3、定义接口（继承 JpaRepository）

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.zxt.demo.demo.UserLog;

**public** **interface** UserLogDao **extends** JpaRepository<UserLog, Integer> {

}

4、测试

此时数据库中并没有对应的数据库表，操作数据库的接口也没有对应的实现类，直接测试即可实现功能（表会自动生成）。

**import** java.util.Date;

**import** java.util.Optional;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.zxt.demo.dao.UserLogDao;

**import** com.zxt.demo.demo.UserLog;

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** SpringBootDemoJpaApplicationTests {

@Autowired

**private** UserLogDao userLogDao;

@Test

**public** **void** insert() {

UserLog entity = **new** UserLog();

entity.setUserName("测试");

entity.setUserIp("192.168.0.1");

entity.setCreateTime(**new** Date());

userLogDao.save(entity);

}

@Test

**public** **void** delete() {

userLogDao.deleteById(1);

}

@Test

**public** **void** update() {

UserLog entity = **new** UserLog();

entity.setId(2);

entity.setUserName("测试用户");

entity.setUserIp("192.168.0.1");

entity.setCreateTime(**new** Date());

// 更新和插入都使用save方法，更新需要提供id

userLogDao.save(entity);

}

@Test

**public** **void** select() {

Optional<UserLog> result = userLogDao.findById(2);

System.***out***.println(result);

}

}

5、可以对接口的查询进行自定义的扩展。

**import** org.springframework.data.domain.Page;

**import** org.springframework.data.domain.Pageable;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** org.springframework.data.jpa.repository.Query;

**import** com.zxt.demo.demo.UserLog;

**public** **interface** UserLogDao **extends** JpaRepository<UserLog, Integer> {

/\*\*

\* **@Description**：可以使用表中有的属性进行查找，使用没有的属性会报错

\*/

UserLog findByUserName(String name);

UserLog findByUserIp(String ip);

/\*\*

\* **@Description**：使用And关键字进行属性组合查询

\*/

UserLog findByUserNameAndUserIp(String name, String ip);

/\*\*

\* **@Description**：@Query自定义查询语句

\*/

@Query("select ul from UserLog ul where ul.id = ?1")

UserLog findByMySql(**int** id);

/\*\*

\*

\* **@Description**：分页查询

\*

\* **@param** userName

\* **@param** pageable

\* **@return**

\*/

Page<UserLog> findByUserName(String userName, Pageable pageable);

}

测试

@Test

**public** **void** testFindByUserName() {

UserLog result = userLogDao.findByUserName("测试用户");

System.***out***.println(result);

}

@Test

**public** **void** testFindByUserIp() {

UserLog result = userLogDao.findByUserIp("192.168.0.1");

System.***out***.println(result);

}

@Test

**public** **void** testFindByUserNameAndUserIp() {

UserLog result = userLogDao.findByUserNameAndUserIp("测试用户", "192.168.0.1");

System.***out***.println(result);

}

@Test

**public** **void** testFindByMySql() {

UserLog result = userLogDao.findByMySql(2);

System.***out***.println(result);

}

// 分页查询测试

@Test

**public** **void** testQueryForPage() {

// Pageable pageable = new PageRequest(0, 10, new Sort(new Order(Direction.DESC, "id")));

// spring boot 2.0推荐写法

// Pageable pageable = PageRequest.of(currentPage, pageSize);

Pageable pageable = PageRequest.*of*(0, 10);

Page<UserLog> result =userLogDao.findByUserName("测试用户", pageable);

System.***out***.println(result.getContent());

}

## 5.3、事务处理

### 5.3.1、事务有四个特性：ACID

原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保这些动作要么全部完成，要么完全不起作用。

一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。

隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

### 5.3.2、传播行为

当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

Spring 定义了七种传播行为：

PROPAGATION\_REQUIRED：表示当前方法必须运行在事务中。如果当前事务存在，方法将会在该事务中运行。否则，会启动一个新的事务， Spring 默认使用这种传播行为。

PROPAGATION\_SUPPORTS：表示当前方法不需要事务上下文，但是如果存在当前事务的话，那么该方法会在这个事务中运行。

PROPAGATION\_MANDATORY：表示该方法必须在事务中运行，如果当前事务不存在，则会抛出一个异常。

PROPAGATION\_REQUIRED\_NEW：表示当前方法必须运行在它自己的事务中。一个新的事务将被启动。如果存在当前事务，在该方法执行期间，当前事务会被挂起。如果使用JTATransactionManager 的话，则需要访问 TransactionManager。

PROPAGATION\_NOT\_SUPPORTED：表示该方法不应该运行在事务中。如果存在当前事务，在该方法运行期间，当前事务将被挂起。如果使用 JTATransactionManager 的话，则需要访问 TransactionManager。

PROPAGATION\_NEVER：表示当前方法不应该运行在事务上下文中。如果当前正有一个事务在运行，则会抛出异常。

PROPAGATION\_NESTED：表示如果当前已经存在一个事务，那么该方法将会在嵌套事务中运行。嵌套的事务可以独立于当前事务进行单独地提交或回滚。如果当前事务不存在，那么其行为与PROPAGATION\_REQUIRED 一样。注意各厂商对这种传播行为的支持是有所差异的。可以参考资源管理器的文档来确认它们是否支持嵌套事务。

### 5.3.3、事务隔离级别

隔离级别定义了一个事务可能受其他并发事务影响的程度。

ISOLATION\_DEFAULT：使用后端数据库默认的隔离级别，Spring 默认使用，mysql默认的隔离级别为：Repeatable Read(可重复读)。

ISOLATION\_READ\_UNCOMMITTED：读未提交，最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

ISOLATION\_READ\_COMMITTED：读已提交，允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

ISOLATION\_REPEATABLE\_READ：可重复读，对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

ISOLATION\_SERIALIZABLE：可串行化，最高的隔离级别，完全服从 ACID 的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的。

脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写再稍后被回滚了，那么第一个事务获取的数据就是无效的。

不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。

幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

### 5.3.4、@Transactional属性说明

a、isolation：用于指定事务的隔离级别。默认为底层事务的隔离级别。

b、noRollbackFor：指定遇到指定异常时强制不回滚事务。

c、noRollbackForClassName：指定遇到指定多个异常时强制不回滚事务。该属性可以指定多个异常类名。

d、propagation:指定事务的传播属性。

e、readOnly：指定事务是否只读。表示这个事务只读取数据但不更新数据，这样可以帮助数据库引擎优化事务。若真的是一个只读取的数据库应设置 readOnly=true。

f、rollbackFor：指定遇到指定异常时强制回滚事务。

g、rollbackForClassName：指定遇到指定多个异常时强制回滚事务。该属性可以指定多个异常类名。

h、timeout：指定事务的超时时长。

注意：mysql数据库存储引擎不能使用 MyISAM，因为它不支持事务操作，应该使用InnoDB。

### 5.3.5、实例

1、用户注册实例

**import** java.util.Date;

**import** javax.transaction.Transactional;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.zxt.demo.dao.UserDao;

**import** com.zxt.demo.dao.UserLogDao;

**import** com.zxt.demo.demo.User;

**import** com.zxt.demo.demo.UserLog;

@Service

**public** **class** UserService {

@Autowired

**private** UserDao userDao;

@Autowired

**private** UserLogDao userLogDao;

/\*\*

\*

\* **@Description**：模拟用户注册

\*

\* **@param** name

\* **@param** ip

\* **@return**

\*/

@Transactional

**public** String register(String name, String ip) {

// 1、添加用户

User user = **new** User();

user.setName(name);

user.setCreate\_time(**new** Date());

userDao.save(user);

// 人为模拟异常发生（因为有事务管理，所以前面添加的用户也会被撤回。没有事务时可以发现添加用户成功，添加日志不成功）

**boolean** flag = **true**;

**if** (flag) {

**throw** **new** RuntimeException();

}

// 2、添加注册日志

UserLog userLog = **new** UserLog();

userLog.setUserName(name);

userLog.setUserIp(ip);

userLog.setCreateTime(**new** Date());

userLogDao.save(userLog);

**return** "success";

}

}

2、测试

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** UserServiceTest {

@Autowired

**private** UserService userService;

@Test

**public** **void** register() {

String result = userService.register("注册用户测试2", "192.168.1.2");

System.***out***.println(result);

}

}

## 5.4、h2嵌入式数据库的使用

H2是一个用Java开发的嵌入式数据库，它本身只是一个类库，可以直接嵌入到应用项目中。H2最大的用途在于可以同应用程序打包在一起发布，这样可以非常方便地存储少量结构化数据。

它的另一个用途是用于单元测试。启动速度快，而且可以关闭持久化功能，每一个用例执行完随即还原到初始状态。

H2的第三个用处是作为缓存，作为NoSQL的一个补充。当某些场景下数据模型必须为关系型，可以拿它当Memcached使，作为后端MySQL/Oracle的一个缓冲层，缓存一些不经常变化但需要频繁访问的数据，比如字典表、权限表。不过这样系统架构就会比较复杂了。

H2的产品优势：

纯Java编写，不受平台的限制；

只有一个jar文件，适合作为嵌入式数据库使用；

h2提供了一个十分方便的web控制台用于操作和管理数据库内容；

功能完整，支持标准SQL和JDBC。麻雀虽小五脏俱全；

支持内嵌模式、服务器模式和集群。

1、添加依赖

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

2、配置数据源

# 数据源

# "~"这个符号代表的就是当前登录到操作系统的用户对应的用户目录

# spring.datasource.url=jdbc:h2:~/test;AUTO\_SERVER=TRUE;DB\_CLOSE\_ON\_EXIT=FALSE

# 内存模式

# spring.datasource.url=jdbc:h2:mem:test

# 自定义路径

spring.datasource.url=jdbc:h2:file:E:/J2ee\_new/spring-boot-demo-h2/spring-boot-demo;AUTO\_SERVER=TRUE;DB\_CLOSE\_ON\_EXIT=FALSE

spring.datasource.username=sa

spring.datasource.password=

# JPA

spring.jpa.hibernate.ddl-auto=update

# 显示 sql 语句

spring.jpa.show-sql=true

注：

1）、"~"这个符号代表的就是当前登录到操作系统的用户对应的用户目录。

2）、账号密码我们指定之后，就会自动创建。

3、进入控制台

路径 ：http://localhost:8080/h2-console。

## 5.5、非关系型数据库操作（Redis）

1、添加依赖

<!-- 数据库Redis -->

<dependency>

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

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

2、配置数据源

#redis

spring.redis.host=localhost

spring.redis.port=6379

#spring.redis.password=123456

#spring.redis.database=0

#spring.redis.pool.max-active=8

#spring.redis.pool.max-idle=8

#spring.redis.pool.max-wait=-1

#spring.redis.pool.min-idle=0

#spring.redis.timeout=0

3、操作Redis数据库

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.data.redis.core.StringRedisTemplate;

**import** org.springframework.data.redis.core.ValueOperations;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** RedisComponent {

@Autowired

**private** StringRedisTemplate stringRedisTemplate;

**public** **void** set(String key, String value) {

ValueOperations<String, String> ops = **this**.stringRedisTemplate.opsForValue();

**if** (!**this**.stringRedisTemplate.hasKey(key)) {

ops.set(key, value);

System.***out***.println("set key success");

} **else** {

// 存在则打印之前的 value 值

System.***out***.println("this key = " + ops.get(key));

}

}

**public** String get(String key) {

**return** **this**.stringRedisTemplate.opsForValue().get(key);

}

**public** **void** del(String key) {

**this**.stringRedisTemplate.delete(key);

}

}

4、测试

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.zxt.demo.redis.RedisComponent;

@RunWith(SpringRunner.**class**)

@SpringBootTest

**public** **class** RedisComponentTest {

@Autowired

**private** RedisComponent redisComponent;

@Test

**public** **void** set() {

redisComponent.set("name", "hello world");

}

@Test

**public** **void** get() {

System.***out***.println(redisComponent.get("name"));

}

@Test

**public** **void** del() {

redisComponent.del("name");

}

}

5、特别需要注意，在生产环境下，如果外网可以访问，一定要记得设置数据库密码！

## 5.6、非关系型数据库操作（MongoDB）