**SpringBoot**

**第一篇 Spring Boot简介**

要我给Spring Boot做个定义，简单来说就是一个基于强大的Spring框架的、推崇JavaConfig的极简配置的web开发框架；

Spring Boot通过内嵌Servlet容器（Tomcat、Jetty，等等）的方式，可以以jar包的形式独立运行一个web项目；

Spring Boot提倡JavaConfig和注解的零配置方式，并且默认配置满足绝大多数场景的需要，意味着少量修改默认配置即可快速搭建一个web项目，极大的提高开发效率；

项目中加入某一个spring-boot-starter-\*依赖，就可以引入该功能的完整jar包，降低pom的复杂度。

**一．分散配置**

分散配置是系统必不可少的一部分，将配置参数抽离出来为后期维护提供很大的便利。spring boot 默认支持两个格式的配置文件:.properties .yml。

.properties与.yml

1. \*.properties属性文件；属于最常见的一种；
2. \*.yml是yaml格式的文件，yaml是一种非常简洁的标记语言。

**二．属性注入**

|  |  |  |  |
| --- | --- | --- | --- |
| ①在application.yml文件中配置：   |  | | --- | | *# 属性注入* **product:  pname:** 矿泉水  **price:** 3.02  **ptype:** 食品  **remark:** 60寸  **productDate:** 2017-09-15 12:12:12 **customer:  custName :** 张三  **custGender :** 男  **custPhone :** 13526458965  **custAddress :** 湖南长沙 **account :  username:** 洪七公 |   ②在实体类中注解配置：   |  | | --- | | @Component @ConfigurationProperties(prefix = **"product"**) **public class** Product {  **private int id**;  **private** String **pname**;  **private double price**;  **private** String **ptype**;  **private** String **remark**;  @JsonFormat(pattern = **"yyyy-MM-dd HH:mm:ss"**)  **private** Timestamp **productDate**; |  1. 在controller类中注入属性和实体。  |  | | --- | | @RestController @EnableConfigurationProperties({Customer.**class**, Product.**class**}) **public class** ProductController {   @Autowired  **private** Product **product**;  @Autowired  **private** Customer **customer**;  @Value(**"${account.username}"**)  **private** String **username**;   @GetMapping(value = **"/getProduct"**)  **public** Product getProduct() {  System.***out***.println(**product**);  System.***out***.println(**customer**);  System.***out***.println(**username**);  **return product**;  } } | |

**三、日志配置**

日志级别从低到高分为TRACE < DEBUG < INFO < WARN < ERROR < FATAL，如果设置为WARN，则低于WARN的信息都不会输出。

**①springboot日志**

Spring Boot默认使用Logback作为日志框架，这是推荐的方式，如果希望修改为熟悉的log4j可以看下文，创建项目时我们引入了spring-boot-starter-web，其依赖spring-boot-starter，spring-boot-starter又依赖于spring-boot-starter-logging，该依赖内容就是Spring Boot默认的日志框架Logback

Logback相关设置，可以在application.properties中进行如下配置：

|  |
| --- |
| # 日志文件路径  logging.file=D:/my\_log.log  # 配置日志打印级别  logging.level.org.springframework=INFO |

**②修改springboot日志为log4j框架**

pom中排除对spring-boot-starter-logging的依赖，并加入对spring-boot-starter-log4j的依赖。

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter</**artifactId**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j</**artifactId**>  <**version**>1.3.8.RELEASE</**version**> </**dependency**> |

项目根路径下添加log4j.properties。

**③修改springboot日志为log4j2框架**

与log4j类似，修改pom，增加spring-boot-starter-log4j2依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter</**artifactId**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-logging</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-log4j2</**artifactId**>  <**version**>1.4.2.RELEASE</**version**> </**dependency**> |

项目根路径下添加log4j2.xml。

**四、修改SpringBoot容器**

①修改内置Tomcat为Jetty

修改pom，去除spring-boot-starter-tomcat的依赖，增加spring-boot-starter-jetty依赖。

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-jetty</**artifactId**>  <**version**>1.4.2.RELEASE</**version**> </**dependency**> |

application.properties中的属性配置与tomcat一致。

②修改内置Tomcat为Undertow容器

修改pom，去除spring-boot-starter-tomcat的依赖，增加spring-boot-starter-undertow依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  <**exclusions**>  <**exclusion**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  </**exclusion**>  </**exclusions**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-undertow</**artifactId**>  <**version**>1.4.2.RELEASE</**version**> </**dependency**> |

application.properties中的属性配置与tomcat一致

③优化内置Tomcat容器

内置的容器有三个分别是Undertow、Jetty、Tomcat,Spring Boot 对这三个容器分别进行了实现，它们上层接口都是EmbeddedServletContainerFactory，该接口也是本文的主要核心。

对于内置容器的定制与优化主要有两种方式,第一种方式是通过配置文件来配置，另外一种是通过码代码的方式。

第一种：通过配置文件来定制和优化Tomcat

|  |
| --- |
| erver.port = 8088  server.context-path=/  server.tomcat.max-threads = 200  server.tomcat.max-connections= 300 |

更多的配置内容参考org.springframework.boot.autoconfigure.web.ServerProperties类内置属性。

第二种：通过码代码方式实现对内置容器的配置与优化Tomcat

有两种方式显示用代码进行对内置容器进行优化与定制,第一种实现内置Servlet容器定制器(org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer),并将类交给Spring容器管理,另外一种是在Spring容器中配置EmbeddedServletContainerFactory接口实现类，这里我们主要针对内置Tomcat，即TomcatEmbeddedServletContainerFactory类

第一种方式实现EmbeddedServletContainerCustomizer接口,并交给Spring容器管理

|  |
| --- |
| @Component  **public** **class** MyEmbeddedServletContainerCustomizer **implements** EmbeddedServletContainerCustomizer {  @Override  **public** **void** customize(ConfigurableEmbeddedServletContainer container) {  //org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletContainerFactory  //说明默认是的Tomcat容器  System.***out***.println(container.getClass());  TomcatEmbeddedServletContainerFactory factory = (TomcatEmbeddedServletContainerFactory) container;  //设置端口  factory.setPort(8088);  //设置Tomcat的根目录  factory.setBaseDirectory(**new** File("d:/tmp/tomcat"));  //设置访问日志存放目录  factory.addContextValves(getLogAccessLogValue());  //设置Tomcat线程数和连接数  factory.addConnectorCustomizers(**new** MyTomcatConnectorCustomizer());  //初始化servletContext对象  factory.addInitializers((servletContext) -> {  System.out.println(" = = = = 获取服务器信息 = = " + servletContext.getServerInfo());  });  }  **private** AccessLogValve getLogAccessLogValue() {  AccessLogValve accessLogValve = **new** AccessLogValve();  accessLogValve.setDirectory("d:/tmp/tomcat/logs");  accessLogValve.setEnabled(**true**);  accessLogValve.setPattern(Constants.AccessLog.COMMON\_PATTERN);  accessLogValve.setPrefix("springboot-access-log");  accessLogValve.setSuffix(".txt");  **return** accessLogValve;  }  }  /\*\*  \* 定制tomcat的连接数与线程数  \*/  **class** MyTomcatConnectorCustomizer **implements** TomcatConnectorCustomizer {  @Override  **public** **void** customize(Connector connector) {  //连接协议 HTTP/1.1  System.***out***.println(connector.getProtocol());  //连接协议处理器 org.apache.coyote.http11.Http11NioProtocol  System.***out***.println(connector.getProtocolHandler().getClass());  //Http11NioProtocol  Http11NioProtocol protocolHandler = (Http11NioProtocol) connector.getProtocolHandler();  // 设置最大连接数  protocolHandler.setMaxConnections(2000);  // 设置最大线程数  protocolHandler.setMaxThreads(500);  }  } |

在Spring容器中配置EmbeddedServletContainerFactory实现类

|  |
| --- |
| @SpringBootConfiguration  **public** **class** WebServerConfiguration {  @Bean  **public** EmbeddedServletContainerFactory embeddedServletContainerFactory() {  TomcatEmbeddedServletContainerFactory factory = **new** TomcatEmbeddedServletContainerFactory();  //设置端口  factory.setPort(8089);  //设置404错误界面  factory.addErrorPages(**new** ErrorPage(HttpStatus.NOT\_FOUND, "/404.html"));  //设置在容器初始化的时候触发  factory.addInitializers((servletContext) -> {  System.out.println(" = = = = 获取服务器信息 = = " + servletContext.getServerInfo());  });  //设置最大连接数和最大线程数  factory.addConnectorCustomizers((connector) -> {  Http11NioProtocol protocolHandler = (Http11NioProtocol) connector.getProtocolHandler();  protocolHandler.setMaxConnections(2000);  protocolHandler.setMaxThreads(500);  });  //设置访问日志记录文件的目录  factory.addContextValves(getLogAccessLogValue());  **return** factory;  }  **private** AccessLogValve getLogAccessLogValue() {  AccessLogValve accessLogValve = **new** AccessLogValve();  accessLogValve.setDirectory("d:/tmp/logs");  accessLogValve.setEnabled(**true**);  accessLogValve.setPattern(Constants.AccessLog.COMMON\_PATTERN);  accessLogValve.setPrefix("SpringBoot-Access-Log");  accessLogValve.setSuffix(".txt");  **return** accessLogValve;  }  } |

**五、SpringBoot自动配置**

①自动配置类都存放在spring-boot-autoconfigure-1.4.2.RELEASE.jar下的

org.springframework.boot.autoconfigure路径下；

②application.properties中配置debug=true后启动容器，可以看到服务器初始化的自动配置如下:

|  |
| --- |
| **DispatcherServletAutoConfiguration**  注册org.springframework.web.servlet.DispatcherServlet  **EmbeddedServletContainerAutoConfiguration**  注册容器类型，如类路径下存在org.apache.catalina.startup.Tomcat，就会注册Tomcat容器  **ErrorMvcAutoConfiguration**  注册异常处理器  **HttpEncodingAutoConfiguration**  注册http编码过滤器  **HttpMessageConvertersAutoConfiguration**  注册json或者xml处理器  **JacksonAutoConfiguration**  注册json对象解析器  **JmxAutoConfiguration**  注册JMX管理器  **MultipartAutoConfiguration**  注册文件传输处理器  **ServerPropertiesAutoConfiguration**  用于初始化容器相关的配置属性，如服务地址、端口、contextPath，并根据当前容器类型初始化各个容器的特有属性，如tomcat的maxThreads、uriEncoding等等，其对应的属性类为ServerProperties；  **WebClientAutoConfiguration**  注册RestTemplate  **WebMvcAutoConfiguration**  注册SpringMvc相关处理器，如ResourceResolver、RequestMappingHandlerAdapter、ExceptionHandlerExceptionResolver、ViewResolver、LocaleResolver，等等  **WebSocketAutoConfiguration**  注册webSocket相关处理器，根据容器类型注册不同的处理器 |

③如果依赖中加入了其它功能的依赖，SpringBoot还会实现这些功能的自动适配，比如我们增加数据库的JPA的功能，就会启用对JpaRepositoriesAutoConfiguration的自动配置功能。

**④说明：**

从各个AutoConfiguration配置类中可以看到如下注解，基于这些注解可以确定这些AutoConfiguration的初始化顺序：

@AutoConfigureOrder(-2147483648)：数越小越先初始化。

@AutoConfigureAfter({EmbeddedServletContainerAutoConfiguration.class})：在指定的配置类初始化后再加载。

@AutoConfigureBefore({WebMvcAutoConfiguration.class})：在指定的配置类初始化前加载。

**六、接管SpringBoot的自动配置**

我们介绍过@SpringBootApplication这个注解，因其包含@EnableAutoConfiguration和@ComponentScan注解，可以自动扫描相关的自动配置类，从而实现自动配置功能的。

上面介绍默认情况下SpringBoot默认会初始化很多的自动配置，这些配置有些我们在项目中可能用不到，那要如何去掉呢？

①去掉不需要的自动配置类

比如我们不需要开启webSocket和JMX的自动配置，我们需要在@SpringBootApplication这个注解中指定exclude属性:

|  |
| --- |
| @SpringBootApplication(exclude = {WebSocketAutoConfiguration.**class**,JmxAutoConfiguration.**class**}) **public class** DruidJavaconfigApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(DruidJavaconfigApplication.**class**, args);  } } |

②明确指定需要启用哪些自动配置

我们可以去掉@SpringBootApplication注解，改用@Configuration、@Import、@ComponentScan注解，在@Import注解中明确指定需要启用哪些自动配置：

|  |
| --- |
| @Configuration @Import({  DispatcherServletAutoConfiguration.**class**,  EmbeddedServletContainerAutoConfiguration.**class**,  ErrorMvcAutoConfiguration.**class**,  HttpEncodingAutoConfiguration.**class**,  HttpMessageConvertersAutoConfiguration.**class**,  JacksonAutoConfiguration.**class**,  MultipartAutoConfiguration.**class**,  ServerPropertiesAutoConfiguration.**class**,  WebMvcAutoConfiguration.**class** }) @ComponentScan **public class** SpringBootWebDemoApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(SpringBootWebDemoApplication.**class**, args);  } } |

**③说明：**

这里推荐使用第一种方式：@SpringBootApplication(exclude={})；

实际上，开启默认的自动配置功能，只是会影响项目启动时间，所以没有特殊需要，可以不需要关闭某个自动配置功能；

在某些情况，比如项目需要多数据源时，在项目中就会包含多个DataSource的Bean，因为DataSourceAutoConfiguration自动配置只能绑定一个数据源，此时发现多个DataSource的Bean被Spring注册就会抛出异常。

|  |
| --- |
| 1.这时就可以采用去掉DataSourceAutoConfiguration的方式;  2.或者也可以在某一个DataSource的Bean上声明@Primary注解，指定其为主数据源，这时DataSourceAutoConfiguration只会加载被指定@Primary注解的主数据源，这样就可以享受到SpringBoot自动配置带来的好处。 |

**④接管WebMvc自动配置**

对于一个web项目，最重要的就是Mvc相关的控制，SpringBoot通过WebMvcAutoConfiguration来完成与Mvc有关的自动配置。如果希望完全接管WebMvc自动配置，可以在项目中创建一个注解了@EnableWebMvc的配置类，比如：

|  |
| --- |
| **package** com.cxp.configuration;  **import** org.apache.log4j.Logger; **import** org.springframework.context.MessageSource; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.ComponentScan; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.context.annotation.FilterType; **import** org.springframework.context.support.ResourceBundleMessageSource; **import** org.springframework.stereotype.Controller; **import** org.springframework.web.multipart.commons.CommonsMultipartResolver; **import** org.springframework.web.servlet.HandlerAdapter; **import** org.springframework.web.servlet.HandlerMapping; **import** org.springframework.web.servlet.ViewResolver; **import** org.springframework.web.servlet.config.annotation.EnableWebMvc; **import** org.springframework.web.servlet.config.annotation.InterceptorRegistry; **import** org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry; **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport; **import** org.springframework.web.servlet.handler.SimpleMappingExceptionResolver; **import** org.springframework.web.servlet.handler.SimpleServletHandlerAdapter; **import** org.springframework.web.servlet.i18n.CookieLocaleResolver; **import** org.springframework.web.servlet.i18n.LocaleChangeInterceptor; **import** org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter; **import** org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping; **import** org.springframework.web.servlet.view.InternalResourceViewResolver;  **import** java.util.Properties;  @Configuration @EnableWebMvc @ComponentScan(basePackages = **"com.cxp"**, useDefaultFilters = **false**, includeFilters = {  @ComponentScan.Filter(type = FilterType.***ANNOTATION***, value = {Controller.**class**}) }) **public class** MvcConfig **extends** WebMvcConfigurationSupport {   **private static final** Logger ***logger*** = Logger  .*getLogger*(MvcConfig.**class**);   */\*\*  \* 描述 : <注册视图处理器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@return*** *\*/* @Bean  **public** ViewResolver viewResolver() {  ***logger***.info(**"ViewResolver"**);  InternalResourceViewResolver viewResolver = **new** InternalResourceViewResolver();  viewResolver.setPrefix(**"/WEB-INF/views/jsp/function/"**);  viewResolver.setSuffix(**".jsp"**);  **return** viewResolver;  }   */\*\*  \* 描述 : <注册消息资源处理器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@return*** *\*/* @Bean  **public** MessageSource messageSource() {  ***logger***.info(**"MessageSource"**);  ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();  messageSource.setBasename(**"config.messages.messages"**);  **return** messageSource;  }   */\*\*  \* 描述 : <注册servlet适配器>. <br>  \* <p>  \* <只需要在自定义的servlet上用@Controller("映射路径")标注即可>  \* </p>  \** ***@return*** *\*/* @Bean  **public** HandlerAdapter servletHandlerAdapter() {  ***logger***.info(**"HandlerAdapter"**);  **return new** SimpleServletHandlerAdapter();  }   */\*\*  \* 描述 : <本地化拦截器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@return*** *\*/* @Bean  **public** LocaleChangeInterceptor localeChangeInterceptor() {  ***logger***.info(**"LocaleChangeInterceptor"**);  **return new** LocaleChangeInterceptor();  }   */\*\*  \* 描述 : <基于cookie的本地化资源处理器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@return*** *\*/* @Bean(name = **"localeResolver"**)  **public** CookieLocaleResolver cookieLocaleResolver() {  ***logger***.info(**"CookieLocaleResolver"**);  **return new** CookieLocaleResolver();  }   */\*\*  \* 描述 : <添加拦截器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@param registry*** *\*/* @Override  **protected void** addInterceptors(InterceptorRegistry registry) {  *//* ***TODO Auto-generated method stub  logger***.info(**"addInterceptors start"**);  registry.addInterceptor(localeChangeInterceptor());  ***logger***.info(**"addInterceptors end"**);  }   */\*\*  \* 描述 : <资源访问处理器>. <br>  \* <p>  \* <可以在jsp中使用/static/\*\*的方式访问/WEB-INF/static/下的内容>  \* </p>  \** ***@param registry*** *\*/* @Override  **protected void** addResourceHandlers(ResourceHandlerRegistry registry) {  ***logger***.info(**"addResourceHandlers"**);  registry.addResourceHandler(**"/static/\*\*"**).addResourceLocations(**"/WEB-INF/static/"**);  }   */\*\*  \* 描述 : <文件上传处理器>. <br>  \* <p>  \* <使用方法说明>  \* </p>  \** ***@return*** *\*/* @Bean(name = **"multipartResolver"**)  **public** CommonsMultipartResolver commonsMultipartResolver() {  ***logger***.info(**"CommonsMultipartResolver"**);  CommonsMultipartResolver commonsMultipartResolver = **new** CommonsMultipartResolver();  commonsMultipartResolver.setMaxUploadSize(1024\*1024);  **return** commonsMultipartResolver;  }   */\*\*  \* 描述 : <异常处理器>. <br>  \* <p>  \* <系统运行时遇到指定的异常将会跳转到指定的页面>  \* </p>  \** ***@return*** *\*/* @Bean(name = **"exceptionResolver"**)  **public** SimpleMappingExceptionResolver simpleMappingExceptionResolver() {  ***logger***.info(**"CP\_SimpleMappingExceptionResolver"**);  SimpleMappingExceptionResolver simpleMappingExceptionResolver = **new** SimpleMappingExceptionResolver();  simpleMappingExceptionResolver.setDefaultErrorView(**"common\_error"**);  simpleMappingExceptionResolver.setExceptionAttribute(**"exception"**);  Properties properties = **new** Properties();  properties.setProperty(**"java.lang.RuntimeException"**, **"common\_error"**);  simpleMappingExceptionResolver.setExceptionMappings(properties);  **return** simpleMappingExceptionResolver;  }  } |

此时debug模式运行项目，会看到WebMvcAutoConfiguration没有被自动配置，说明我们自己定义的MvcConfig已经完全接管了默认的自动配置，这是因为WebMvcAutoConfiguration有一个条件注解:

|  |
| --- |
| @ConditionalOnMissingBean({WebMvcConfigurationSupport.class}) |

而我们本例中MvcConfig就是WebMvcConfigurationSupport的实现类，同时加入@EnableWebMvc注解也会导入一个WebMvcConfigurationSupport的实现类：DelegatingWebMvcConfiguration，所以MvcConfig继承WebMvcConfigurationSupport不是必须的，但是可以方便我们编码。

如果希望可以继续使用WebMvcAutoConfiguration的自动配置，而只是需要修改或者增加MVC中的某些配置时，我们可以创建一个配置类，并继承于抽象类WebMvcConfigurerAdapter，我们可以通过实现抽象类的方法来注册自己的控制器。

**七、注册Servlet、Filter、Listener的方法**

①如果是war包项目，我们可以将Servlet、Filter、Listener注册到WebApplicationInitializer的实现类中：

|  |
| --- |
| @Order(1) **public class** CommonInitializer **implements** WebApplicationInitializer {  @Override  **public void** onStartup(ServletContext servletContext)  **throws** ServletException {  *//Log4jConfigListener* servletContext.setInitParameter(**"log4jConfigLocation"**, **"classpath:log4j.properties"**);  servletContext.addListener(Log4jConfigListener.**class**);   *//OpenSessionInViewFilter* CharacterEncodingFilter character = **new** CharacterEncodingFilter();  FilterRegistration.Dynamic filterRegistration = servletContext.addFilter(**"character"**, character);  filterRegistration.addMappingForUrlPatterns(  EnumSet.*of*(DispatcherType.***REQUEST***, DispatcherType.***FORWARD***, DispatcherType.***INCLUDE***), **false**, **"/"**);   *//DemoServlet* DispatcherServlet dispatcherServlet = **new** DispatcherServlet();  ServletRegistration.Dynamic servlet = servletContext.addServlet(  **"dispatcherServlet"**, dispatcherServlet);  servlet.setLoadOnStartup(2);  servlet.addMapping(**"/"**);  } } |

②如果是jar包部署方式，则可以将其注册到任意一个@Configuration配置类中:

|  |
| --- |
| @Configuration **public class** WebConfig {   @Bean  **public** ServletRegistrationBean servletRegistrationBean\_demo1(){  **return new** ServletRegistrationBean(**new** DemoServlet(),**"/demo-servlet1"**);  }   @Bean  **public** ServletRegistrationBean servletRegistrationBean\_demo2(){  ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean();  servletRegistrationBean.addUrlMappings(**"/demo-servlet2"**);  servletRegistrationBean.setServlet(**new** DemoServlet2());  **return** servletRegistrationBean;  }   @Bean  **public** FilterRegistrationBean filterRegistrationBean(){   FilterRegistrationBean filterRegistrationBean = **new** FilterRegistrationBean();  filterRegistrationBean.setFilter(**new** OpenSessionInViewFilter());  Set<String> set = **new** HashSet<>();  set.add(**"/"**);  filterRegistrationBean.setUrlPatterns(set);  **return** filterRegistrationBean;  }   @Bean  **public** ServletListenerRegistrationBean servletListenerRegistrationBean(){  ServletListenerRegistrationBean servletListenerRegistrationBean = **new** ServletListenerRegistrationBean();  servletListenerRegistrationBean.setListener(**new** Log4jConfigListener());  servletListenerRegistrationBean.addInitParameter(**"log4jConfigLocation"**,**"classpath:log4j.properties"**);  **return** servletListenerRegistrationBean;  } } |

**第二篇 SpringBoot用JdbcTemplates访问Mysql**

①在pom文件中引入spring-boot-starter-jdbc、 mysql连接类和连接池的依赖:

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.29</version>  </dependency> |

②在application.yml中配置相关文件

|  |
| --- |
| **spring:**  **datasource:  driver-class-name:** com.mysql.jdbc.Driver  **url:** jdbc:mysql://192.168.0.102:3306/test?characterEncoding=utf-8  **username:** administrator  **password:** 123456 |

③自定义数据源

|  |
| --- |
| **import** com.alibaba.druid.pool.DruidDataSource; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.context.annotation.Bean; **import** org.springframework.core.env.Environment;  **import** javax.sql.DataSource;  */\*\*  \* Created by ASUS on 2017/9/16.  \*/*  @Configuration**public class** MyDataSource {  @Autowired  **private** Environment **env**;   @Bean  **public** DataSource dataSource() {  DruidDataSource dataSource = **new** DruidDataSource();  dataSource.setUrl(**env**.getProperty(**"spring.datasource.url"**));  dataSource.setDriverClassName(**env**.getProperty(**"spring.datasource.driver-class-name"**));  dataSource.setUsername(**env**.getProperty(**"spring.datasource.username"**));  dataSource.setPassword(**env**.getProperty(**"spring.datasource.password"**));  dataSource.setInitialSize(2);  dataSource.setMaxActive(20);  dataSource.setMinIdle(0);  dataSource.setMaxWait(60000);  dataSource.setValidationQuery(**"SELECT 1"**);  dataSource.setTestOnBorrow(**false**);  dataSource.setTestWhileIdle(**true**);  dataSource.setPoolPreparedStatements(**false**);  **return** dataSource;  } } |

④dao层和dao层实现类

|  |
| --- |
| **public interface** ProductDao {  *//添加商品* **int** saveProduct(Product product);   *//修改商品* **int** updateProduct(Product product);   *//删除商品* **int** deleteProduct(**int** id);   *//根据id获取商品* Product getProductById(**int** id);   *//获取商品列表* List<Product> findProductList(Integer size, Integer currentPage); } |

|  |
| --- |
| @Repository **public class** ProductDaoImpl **implements** ProductDao {  **private** Logger **logger** = LoggerFactory.*getLogger*(ProductDaoImpl.**class**);   @Autowired  **private** JdbcTemplate **jdbcTemplate**;   @Override  **public int** saveProduct(Product product) {  **logger**.info(**"添加商品"**);  **return jdbcTemplate**.update(**"insert into product(pname,price,ptype,remark,productDate) values (?,?,?,?,?);"**,  product.getPname(), product.getPrice(), product.getPtype(), product.getRemark(), product.getProductDate());  }   @Override  **public int** updateProduct(Product product) {  **logger**.info(**"修改商品"**);  **return jdbcTemplate**.update(**"update product set pname=?,price=? where id=?"**,  product.getPname(), product.getPrice(), product.getId());  }   @Override  **public int** deleteProduct(**int** id) {  **logger**.info(**"根据id商品"**);  **return jdbcTemplate**.update(**"delete from product where id=?"**, id);  }   @Override  **public** Product getProductById(**int** id) {  List<Product> list = **jdbcTemplate**.query(**"select id,pname,price,ptype,remark,productDate from product where id=?"**,  **new** Object[]{id}, **new** BeanPropertyRowMapper<Product>(Product.**class**));  **if** (list != **null** && list.size() > 0) {  Product product = list.get(0);  **return** product;  } **else** {  **return null**;  }  }   @Override  **public** List<Product> findProductList(Integer size, Integer currentPage) {  **logger**.info(**"获取商品列表并分页:size:"** + size + **",currentPage:"** + currentPage);  List<Product> list = **jdbcTemplate**.query(**"select id,pname,price,ptype,remark,productDate from product limit "** + size + **" offset "** + (currentPage - 1) \* size,  **new** Object[]{}, **new** BeanPropertyRowMapper<Product>(Product.**class**));  **if** (list != **null** && list.size() > 0) {  **return** list;  } **else** {  **return null**;  }  } } |

第三篇 Springboot整合Jpa

①在pom.xml文件中添加spring-boot-starter-jpa依赖:

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

②在application.yml中配置文件

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://localhost:3306/test?characterEncoding=utf8  username: root  password: 123456  jpa:  hibernate:  ddl-auto: update # 第一次建表create 后面用update  show-sql: true |
| 配置指定对数据库表结构的处理方式，值有：create、create-drop、update、validate  # create：每次加载hibernate的时候，都会重新根据模型生成表。如果表已存在，会先删除该表再生成。  # create-drop：启动项目加载hibernate的时候，会生成表。停止项目时，会把生成的表删除掉。  # update：常用属性。每次加载hibernate的时候，会生成表。如果表存在，会根据模型的属性变化来更新表结构，这个过程不会做删表处理。  # validate：每次加载hibernate的时候，会检查表结构，但不会生成表。 |
| 自定义数据源 |
| @Configuration  **public class** MyDataSource {  @Autowired  **private** Environment **env**;   @Bean  **public** DataSource dataSource() {  DruidDataSource dataSource = **new** DruidDataSource();  dataSource.setUrl(**env**.getProperty(**"spring.datasource.url"**));  dataSource.setDriverClassName(**env**.getProperty(**"spring.datasource.driver-class-name"**));  dataSource.setUsername(**env**.getProperty(**"spring.datasource.username"**));  dataSource.setPassword(**env**.getProperty(**"spring.datasource.password"**));  dataSource.setInitialSize(2);  dataSource.setMaxActive(20);  dataSource.setMinIdle(0);  dataSource.setMaxWait(60000);  dataSource.setValidationQuery(**"SELECT 1"**);  dataSource.setTestOnBorrow(**false**);  dataSource.setTestWhileIdle(**true**);  dataSource.setPoolPreparedStatements(**false**);  **return** dataSource;  } } |

③创建实体类

|  |
| --- |
| @Entity @Table(name = **"customer"**) **public class** Customer {  @Id  @GeneratedValue(strategy = GenerationType.***AUTO***)  **private int id**;  @Column  **private** String **custName**;  @Column  **private** String **custGender**;  @Column  **private** String **custPhone**;  @Column  **private** String **custAddress**;  ......setter、getter方法省略 } |

④Dao层，继承JpaRepository接口

|  |
| --- |
| @Repository **public interface** CustomerDao **extends** JpaRepository<Customer,Integer>{  Customer findByCustName(String custName);   List<Customer> findByCustAddress(String custAddress); } |

⑤Service层

|  |
| --- |
| @Service **public class** CustomerServiceImpl **implements** CustomerService {  @Autowired  **private** CustomerDao **customerDao**;   @Override  **public** Customer saveCustomer(Customer customer) {  **return customerDao**.save(customer);  }   @Override  **public void** deleteCustomer(Customer customer) {  **customerDao**.delete(customer);  }   @Override  **public void** deleteCustomerById(Integer id) {  **customerDao**.delete(id);  }   @Override  **public** Customer updateCustomer(Customer customer) {  **return customerDao**.save(customer);  }   @Override  **public** Customer getCustomerById(Integer id) {  **return customerDao**.getOne(id);  }   @Override  **public** List<Customer> findCustomerList() {  **return customerDao**.findAll();  }  @Override **public** Customer findByCustName(String custName) {  Customer byCustName = **customerDao**.findByCustName(custName);  **return** byCustName;  }  @Override **public** List<Customer> findByCustAddress(String custAddress) {  List<Customer> customerList = **customerDao**.findByCustAddress(custAddress);  **return** customerList;  }  } |
|  |
|  |
|  |
|  |
|  |

第四篇 SpringBoot整合MyBaits(基于注解实现)

①在pom.xml文件中添加mybatis-spring-boot-starter、mysql连接类和连接池依赖:

|  |
| --- |
| <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.29</version>  </dependency> |

②在application.yml中配置文件

|  |
| --- |
| spring:  datasource:  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://localhost:3306/test?characterEncoding=utf8  username: root  password: 123456 |

③dao层

|  |
| --- |
| @Mapper **public interface** CustomerDaoMybatis {  @Insert(value = **"insert into customer (cust\_name,cust\_gender,cust\_phone,cust\_address) values (#{cust\_name},#{cust\_gender},#{cust\_phone},#{cust\_address})"**)  **int** saveCustomer(CustomerMybatis customerMybatis);   @Update(value = **"update customer set cust\_name=#{cust\_name},cust\_gender=#{cust\_gender},cust\_phone=#{cust\_phone},cust\_address=#{cust\_address} where id=#{id}"**)  **int** updateCustomer(CustomerMybatis customerMybatis);   @Delete(value = **"delete from customer where id=#{id}"**)  **int** deleteCustomer(**int** id);   @Select(value = **"select id,cust\_name,cust\_gender,cust\_phone,cust\_address from customer"**)  List<CustomerMybatis> findCustomerList();   @Select(value = **"select id,cust\_name,cust\_gender,cust\_phone,cust\_address from customer where id=#{id}"**)  @Results(id=**"CustomerMabatisMap"**,value = {  @Result(property = **"id"**,column = **"id"**,id = **true**),  @Result(property = **"cust\_name"**,column = **"cust\_name"**),  @Result(property = **"cust\_gender"**,column = **"cust\_gender"**),  @Result(property = **"cust\_phone"**,column = **"cust\_phone"**),  @Result(property = **"cust\_address"**,column = **"cust\_address"**) })  CustomerMybatis getCustomerById(**int** id); }  *//映射到Map集合中*  @Select(value = **"select id,cust\_name,cust\_gender,cust\_phone,cust\_address from customer"**) @ResultType(value = Map.**class**) List<Map<String,Object>> findCustomerMybatisMap();  *//用户map集合传递参数* @Select(value = **"select id,cust\_name,cust\_gender,cust\_phone,cust\_address from customer where id=#{id} and cust\_name=#{cust\_name}"**) @ResultType(value = Map.**class**) Map<String,Object> findCustomerMybatisMapByCondition(Map<String,Object> map); |

④service层

|  |
| --- |
| @Service **public class** CustomerMybatisServiceImpl **implements** CustomerMybatisService {  @Autowired  **private** CustomerDaoMybatis **customerDaoMybatis**;   @Override  **public int** saveCustomer(CustomerMybatis customerMybatis) {  **return customerDaoMybatis**.saveCustomer(customerMybatis);  }   @Override  **public int** deleteCustomer(**int** id) {  **return customerDaoMybatis**.deleteCustomer(id);  }   @Override  **public int** updateCustomer(CustomerMybatis customerMybatis) {  **return customerDaoMybatis**.updateCustomer(customerMybatis);  }   @Override  **public** CustomerMybatis getCustomerById(**int** id) {  **return customerDaoMybatis**.getCustomerById(id);  }   @Override  **public** List<CustomerMybatis> findCustomerList() {  **return customerDaoMybatis**.findCustomerList();  }  @Override **public** List<Map<String, Object>> findCustomerMybatisMap() {  **return customerDaoMybatis**.findCustomerMybatisMap(); }  } |

⑤Controller层

|  |
| --- |
| @RestController @RequestMapping(value = **"/customerMybatisController"**) **public class** CustomerMybatisController {  @Autowired  **private** CustomerMybatisService **customerMybatisService**;   @Autowired  **private** CustomerDaoMybatis **customerDaoMybatis**;   @GetMapping(value = **"/saveCustomer"**)  **public** CustomerMybatis saveCustomer(CustomerMybatis customerMybatis) {  System.***out***.println(**customerMybatisService**.findCustomerList());  **customerMybatisService**.saveCustomer(customerMybatis);  **return** customerMybatis;  }   @GetMapping(value = **"/getcustomermybatismap"**)  **public** List<Map<String,Object>> getCustomerMybatisMap(){  List<Map<String, Object>> customerMybatisMap = **customerMybatisService**.findCustomerMybatisMap();  **return** customerMybatisMap;  }   @GetMapping(value = **"/"**)  **public** Map<String,Object> getCustomerMybatisMapByCondition(){  Map<String,Object> param=**new** HashMap<>();  param.put(**"id"**,1);  param.put(**"cust\_name"**,**"张三丰"**);  Map<String, Object> customerMybatisMap = **customerDaoMybatis**.findCustomerMybatisMapByCondition(param);  **return** customerMybatisMap;  } } |

⑥springboot中mybatis注解的使用

|  |
| --- |
|  |

第五篇 SpringBoot整合MyBaits(基于xml实现)

**一、使用class类配置**

①在pom.xml文件中添加依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.7.RELEASE</**version**>  <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>  <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**java.version**>1.8</**java.version**> </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>  *<!--pagehelper -->* <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  <**version**>1.1.1</**version**>  </**dependency**> </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**> </**build**> |

②applications.properties配置文件中配置数据源

|  |
| --- |
| **server.port**=**8012 server.context-path**=**/ datasource.driverClassName**=**com.mysql.jdbc.Driver datasource.url**=**jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=UTF-8 datasource.username**=**root datasource.password**=**123456 datasource.initialSize**=**5 datasource.minIdle**=**5 datasource.maxactive**=**30** |

③配置数据源和mybatis配置

|  |
| --- |
| DataSource数据源 |
| @Configuration **public class** DataSourceConfig {  @Autowired  **private** Environment **ev**;   @Bean  **public** DataSource dataSource() {  DruidDataSource druidDataSource = **new** DruidDataSource();  druidDataSource.setDriverClassName(**ev**.getProperty(**"datasource.driverClassName"**));  druidDataSource.setUrl(**ev**.getProperty(**"datasource.url"**));  druidDataSource.setUsername(**ev**.getProperty(**"datasource.username"**));  druidDataSource.setPassword(**ev**.getProperty(**"datasource.password"**));  druidDataSource.setMaxActive(Integer.*parseInt*(**ev**.getProperty(**"datasource.maxactive"**)));  druidDataSource.setMinIdle(Integer.*parseInt*(**ev**.getProperty(**"datasource.minIdle"**)));  druidDataSource.setInitialSize(Integer.*parseInt*(**ev**.getProperty(**"datasource.initialSize"**)));  **return** druidDataSource;  } } |
| MyBatis配置 |
| */\*\*  \* 实现TransactionManagementConfigurer接口，配置spring事务的管理  \*/* @Configuration **public class** MyBatisConfig **implements** TransactionManagementConfigurer{  @Autowired  **private** DataSource **dataSource**;   @Bean(name = **"sqlSessionFactory"**)  **public** SqlSessionFactory sqlSessionFactory(){  SqlSessionFactoryBean bean=**new** SqlSessionFactoryBean();  bean.setDataSource(**dataSource**);  bean.setTypeAliasesPackage(**"com.cxp.pojo"**);   ResourcePatternResolver resolver=**new** PathMatchingResourcePatternResolver();  **try** {  //设置xml文件扫描路径 bean.setMapperLocations(resolver.getResources(**"classpath:/mybatis/\*Mapper.xml"**));  **return** bean.getObject();  }**catch** (Exception e){  e.printStackTrace();  **throw new** RuntimeException(e);  }  }   @Bean  **public** SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory){  **return new** SqlSessionTemplate(sqlSessionFactory);  }   *//事务支持* @Bean  @Override  **public** PlatformTransactionManager annotationDrivenTransactionManager() {  **return new** DataSourceTransactionManager(**dataSource**);  } } |
| mybatis扫描(mapper接口)路径配置 |
| @Configuration @AutoConfigureAfter(MyBatisConfig.**class**) **public class** MybatisMapperScannerConfig {   @Bean  **public** MapperScannerConfigurer mapperScannerConfigurer(){  MapperScannerConfigurer mapperScannerConfigurer=**new** MapperScannerConfigurer();  mapperScannerConfigurer.setSqlSessionFactoryBeanName(**"sqlSessionFactory"**);  mapperScannerConfigurer.setBasePackage(**"com.cxp.dao.mapper"**);  **return** mapperScannerConfigurer;  }   */\*  \* 注册MyBatis分页插件PageHelper  \*/* @Bean  **public** PageHelper pageHelper() {  PageHelper pageHelper = **new** PageHelper();  Properties p = **new** Properties();  p.setProperty(**"offsetAsPageNum"**, **"true"**);  p.setProperty(**"rowBoundsWithCount"**, **"true"**);  p.setProperty(**"reasonable"**, **"true"**);  pageHelper.setProperties(p);  **return** pageHelper;  } } |

④在src/main/resources下创建文件夹mybatis，把mapper的xml文件放到mybatis文件夹下面

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" *>*** <**mapper namespace="com.cxp.dao.mapper.ProductMapper"**>    <**resultMap type="com.cxp.pojo.Product" id="productBaseMap"**>  <**id column="id" property="id" jdbcType="INTEGER"**/>  <**result column="pname" property="pname" jdbcType="VARCHAR"**/>  <**result column="price" property="price" jdbcType="VARCHAR"**/>  <**result column="remark" property="remark" jdbcType="VARCHAR"**/>  <**result column="productDate" property="productDate" jdbcType="TIMESTAMP"**/>  </**resultMap**>    <**sql id="base\_column"**>  id,pname,price,remark,productDate   </**sql**>    <**sql id="page\_conditionn\_limit"**>  <**if test="(pageSize != null and pageSize >0) and (pageRow != null and pageRow >=0)"**>  limit #{pageSize} offset #{pageRow}  </**if**>  </**sql**>    *<!-- 查询所有列表并分页 -->* <**select id="findProductList" resultType="java.util.LinkedHashMap" parameterType="java.util.Map"**>  select <**include refid="base\_column"**></**include**> from product   <**include refid="page\_conditionn\_limit"**></**include**>  </**select**>    <**insert id="saveProduct" parameterType="com.cxp.pojo.Product"**>  insert into product (pname,price,remark,productDate) values (#{pname},#{price},#{remark},#{productDate})  </**insert**>    <**select id="getProductById" parameterType="Integer" resultType="com.cxp.pojo.Product"**>  select <**include refid="base\_column"**/> from product where id=#{id}  </**select**>   *<!--查询商品列表，并通过PageHelper分布-->* <**select id="findProductListAll" resultType="java.util.Map"**>  select <**include refid="base\_column"**></**include**> from product  </**select**> </**mapper**> |
| Mapper接口 |
| **public interface** ProductMapper {  *// 查询所有商品列表并分页* **public** List<Map<String, Object>> findProductList(Map<String, Object> map);   *// 添加商品* **public** Integer saveProduct(Product product);   *// 根据id查询商品* **public** Product getProductById(**int** id);   *//查询商品列表，并通过pageHepler分页* **public** List<Map<String,Object>> findProductListAll(); } |

⑤Controller层

|  |
| --- |
| @RestController **public class** ProductController {  @Autowired  **private** ProductMapper **productMapper**;   @GetMapping(value = **"/findproductlistmap"**)  **public** List<Map<String,Object>> findProductListMap(){  PageHelper.*startPage*(1,20);  List<Map<String, Object>> productListAll = **productMapper**.findProductListAll();  **return** productListAll;  } } |

⑥在main方法所在类上加注解@MapperScan(value = **"com.cxp.dao.mapper"**)

|  |
| --- |
| @SpringBootApplication @MapperScan(value = **"com.cxp.dao.mapper"**) **public class** Springboot3ClassApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(Springboot3ClassApplication.**class**, args);  } } |

**二、使用starter配置**

①在pom中引入主要依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.1</**version**> </**dependency**> |

②application.properties配置文件配置数据源和mybatis的相关配置

|  |
| --- |
| **spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.url**=**jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=UTF-8 spring.datasource.type**=**com.alibaba.druid.pool.DruidDataSource spring.datasource.username**=**root spring.datasource.password**=**123456  mybatis.mapper-locations**=**classpath:/mybatis/\*Mapper.xml mybatis.type-aliases-package**=**com.cxp.pojo** |

③Mapper接口、Mapper配置文件、Controller层同上。

**三、MyBatis-generator自动生成MyBatis代码**

MyBatis Generator是一个非常方便的代码生成工具，可以根据数据库中表结构自动生成CRUD代码，可以满足大部分需求。

MyBatis Generator (MBG) 是一个Mybatis的代码生成器 ，可以根据数据库中表结构自动生成简单的CRUD（插入，查询，更新，删除）操作。 但联合查询和存储过程，需手动手写SQL和对象。

1、Intellij IDEA中使用

①pom.xml文件

因为IntelliJ中没有mybatis-generato对应的插件，所以需要在MAVEN中使用 mybatis-generator-maven-plugin插件来完成功能。

|  |
| --- |
| <**plugin**>  <**groupId**>org.mybatis.generator</**groupId**>  <**artifactId**>mybatis-generator-maven-plugin</**artifactId**>  <**version**>1.3.2</**version**>  <**dependencies**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.34</**version**>  </**dependency**>  </**dependencies**>  <**configuration**>  <**overwrite**>true</**overwrite**>  </**configuration**> </**plugin**> |

② 配置generatorConfig.xml

resources下建generatorConfig.xml,作为mybatis-generator-maven-plugin插件的执行目标。

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* **<!DOCTYPE generatorConfiguration  PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd"*>*** <**generatorConfiguration**>  *<!--  出现错误：Caused by: java.lang.ClassNotFoundException: com.mysql.jdbc.Driver  解决办法：将本地的MAVEN仓库中的mysql驱动引入进来  -->* <**classPathEntry  location="E:\\Eclipse Maven\\repository\\mysql\\mysql-connector-java\\5.1.46\\mysql-connector-java-5.1.46.jar"**/>   <**context id="mysqlgenerator" targetRuntime="MyBatis3"**>  *<!--不生成注释-->* <**commentGenerator**>  <**property name="suppressAllComments" value="true"**/>  </**commentGenerator**>  *<!-- 配置数据库连接 -->* <**jdbcConnection driverClass="com.mysql.jdbc.Driver"  connectionURL="jdbc:mysql://192.168.0.106:3306/shiro"  userId="root"  password="123456"**/>   *<!-- 指定javaBean生成的位置 -->* <**javaModelGenerator targetPackage="com.cxp.mybatisgenerator.pojo" targetProject="src/main/java"**>  *<!-- 在targetPackage的基础上，根据数据库的schema再生成一层package，最终生成的类放在这个package下，默认为false -->* <**property name="enableSubPackages" value="true"**/>  *<!-- 设置是否在getter方法中，对String类型字段调用trim()方法 -->* <**property name="trimStrings" value="true"**/>  </**javaModelGenerator**>   *<!--指定sql映射文件生成的位置 -->* <**sqlMapGenerator targetPackage="mybatis" targetProject="src/main/resources"**>  <**property name="enableSubPackages" value="true"**/>  </**sqlMapGenerator**>  *<!-- 指定dao接口生成的位置，mapper接口 -->* <**javaClientGenerator type="XMLMAPPER" targetPackage="com.cxp.mybatisgenerator.mapper"  targetProject="src/main/java"**>  <**property name="enableSubPackages" value="true"**/>  </**javaClientGenerator**>   *<!-- table表生成对应的DoaminObject -->* <**table tableName="ssm\_user" domainObjectName="Ssm\_user"**></**table**>  <**table tableName="resources" domainObjectName="Resources"**></**table**>   </**context**>  </**generatorConfiguration**> |

③Intellij配置

新增一个maven执行配置，在Command line中输入 mybatis-generator:generate -e

|  |
| --- |
|  |

启动后，不报错即可产生相应的Model类和Mapper文件：

|  |
| --- |
|  |

第六篇 SpringBoot实现多数据源

①在pom.xml文件中添加依赖

|  |
| --- |
| <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.29</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <scope>runtime</scope>  </dependency>  <**dependency**>  <**groupId**>com.oracle</**groupId**>  <**artifactId**>ojdbc6</**artifactId**>  <**version**>11.2.0.1.0</**version**> </**dependency**> |

②首先要将spring boot自带的DataSourceAutoConfiguration禁掉，因为它会读取application.properties文件的spring.datasource.\*属性并自动配置单数据源。在@SpringBootApplication注解中添加exclude属性即可：

|  |
| --- |
| @SpringBootApplication(exclude = { DataSourceAutoConfiguration.**class** })  **public** **class** ManyDataSourceApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ManyDataSourceApplication.**class**, args);  }  } |

③配置文件中配置数据源信息

|  |
| --- |
| # 数据库访问配置  # 主数据源，默认的  datasource.primary.type=com.alibaba.druid.pool.DruidDataSource  datasource.primary.driver-class-name=com.mysql.jdbc.Driver  datasource.primary.url=jdbc:mysql://192.168.0.102:3306/test?characterEncoding=utf-8  datasource.primary.username=administrator  datasource.primary.password=123456  # 初始化大小，最小，最大  datasource.primary.initialSize=5  datasource.primary.minIdle=5  datasource.primary.maxActive=30  # 配置获取连接等待超时的时间  datasource.primary.maxWait=60000  # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒  datasource.primary.timeBetweenEvictionRunsMillis=60000  # 配置一个连接在池中最小生存的时间，单位是毫秒  datasource.primary.minEvictableIdleTimeMillis=300000  datasource.primary.validationQuery=SELECT 'x'  datasource.primary.testWhileIdle=true  datasource.primary.testOnBorrow=false  datasource.primary.testOnReturn=false  # 打开PSCache，并且指定每个连接上PSCache的大小  datasource.primary.poolPreparedStatements=true  datasource.primary.maxPoolPreparedStatementPerConnectionSize=20  # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙  datasource.primary.filters=stat,wall,slf4j  # 合并多个DruidDataSource的监控数据  #datasource.primary.useGlobalDataSourceStat=true  #第二个数据源配置  datasource.secondary.type=com.alibaba.druid.pool.DruidDataSource  datasource.secondary.driver-class-name=com.mysql.jdbc.Driver  datasource.secondary.url=jdbc:mysql://localhost:3306/test?characterEncoding=utf-8  datasource.secondary.username=root  datasource.secondary.password=123456  # 初始化大小，最小，最大  datasource.secondary.initialSize=5  datasource.secondary.minIdle=5  datasource.secondary.maxActive=30  # 配置获取连接等待超时的时间  datasource.secondary.maxWait=60000  # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒  datasource.secondary.timeBetweenEvictionRunsMillis=60000  # 配置一个连接在池中最小生存的时间，单位是毫秒  datasource.secondary.minEvictableIdleTimeMillis=300000  datasource.secondary.validationQuery=SELECT 'x'  datasource.secondary.testWhileIdle=true  datasource.secondary.testOnBorrow=false  datasource.secondary.testOnReturn=false  # 打开PSCache，并且指定每个连接上PSCache的大小  datasource.secondary.poolPreparedStatements=true  datasource.secondary.maxPoolPreparedStatementPerConnectionSize=20  # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙  datasource.secondary.filters=stat,wall,slf4j |

④手动配置数据源并创建出来

|  |
| --- |
| @Configuration  **public** **class** DataSourceConfiguration {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(DataSourceConfiguration.**class**);  @Bean(name = "primaryDataSource")  @Primary  @ConfigurationProperties(prefix = "datasource.primary")  **public** DataSource primaryDataSource() {  ***logger***.info("-------- primaryDataSource init --------");  **return** DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  }  @Bean(name = "secondaryDataSource")  @ConfigurationProperties(prefix = "datasource.secondary")  **public** DataSource secondaryDataSource() {  ***logger***.info("-------------------- secondaryDataSource init ---------------------");  **return** DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  }  } |
| product类引用数据源2，操作指定的数据库 |
| @Configuration  @MapperScan(basePackages = { "com.cxp.mapper.Product" }, sqlSessionFactoryRef = "sqlSessionFactory1")  **public** **class** ProductDataConfig {  @Autowired  @Qualifier("secondaryDataSource")  **private** DataSource dsl;  @Bean  **public** SqlSessionFactory sqlSessionFactory1() **throws** Exception {  SqlSessionFactoryBean factoryBean = **new** SqlSessionFactoryBean();  factoryBean.setDataSource(dsl);  **return** factoryBean.getObject();  }  @Bean  **public** SqlSessionTemplate sqlSessionTemplate() **throws** Exception {  //使用上面配置的factory  SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactory1());  **return** template;  }  } |
| customer类引用数据源1，操作指定的数据库 |
| @Configuration  @MapperScan(basePackages = { "com.cxp.mapper.Customer" }, sqlSessionFactoryRef = "sqlSessionFactory2")  **public** **class** CustomerDataConfig2 {  @Autowired  @Qualifier("primaryDataSource")  **private** DataSource second;  @Bean  **public** SqlSessionFactory sqlSessionFactory2() **throws** Exception {  SqlSessionFactoryBean factoryBean = **new** SqlSessionFactoryBean();  factoryBean.setDataSource(second);  **return** factoryBean.getObject();  }  @Bean  **public** SqlSessionTemplate sqlSessionTemplate() **throws** Exception {  //使用上面配置的factory  SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactory2());  **return** template;  }  } |

⑤mapper接口和xml配置文件

|  |
| --- |
|  |
|  |

第六(一)篇 SpringBoot实现多数据源

1、引入pom.xml文件

|  |
| --- |
|  |
|  |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.postgresql</**groupId**>  <**artifactId**>postgresql</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.0</**version**> </**dependency**> <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**> </**dependency**> <**dependency**>  <**groupId**>com.oracle</**groupId**>  <**artifactId**>ojdbc6</**artifactId**>  <**version**>11.2.0.1.0</**version**> </**dependency**> <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> |
|  |

2、配置数据源

|  |
| --- |
| @Configuration **public class** ManyDataSourceConfig {  **static final** Logger ***logger*** = LoggerFactory.*getLogger*(ManyDataSourceConfig.**class**.getName());   @Bean(name = **"primaryDataSource"**)  @Primary  @ConfigurationProperties(prefix = **"primary.dataSource"**)  **public** DataSource primaryDataSource() {  ***logger***.info(**"-------- primaryDataSource init --------"**);  **return** DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  }   @Bean(name = **"secondDataSource"**)  @ConfigurationProperties(prefix = **"second.dataSource"**)  **public** DataSource secondDataSource() {  ***logger***.info(**"-------- secondDataSource init --------"**);  **return** DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  }   @Bean(name = **"thirdDataSource"**)  @ConfigurationProperties(prefix = **"third.dataSource"**)  **public** DataSource thirdDataSource() {  ***logger***.info(**"-------- thirdDataSource init --------"**);  **return** DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  } } |

3、配置SqlSession连接

|  |
| --- |
| @Configuration @MapperScan(basePackages = {**"com.cxp.mapper.mysql"**},sqlSessionFactoryRef = **"sqlSessionFactoryMysql"**) **public class** MysqlDataConfig {  @Autowired  @Qualifier(**"thirdDataSource"**)  **private** DataSource **mysqlDs**;   @Bean  **public** SqlSessionFactory sqlSessionFactoryMysql() **throws** Exception {  SqlSessionFactoryBean factoryBean = **new** SqlSessionFactoryBean();  factoryBean.setDataSource(**mysqlDs**);  **return** factoryBean.getObject();  }   @Bean  **public** SqlSessionTemplate sqlSessionTemplate() **throws** Exception {  *//使用上面配置的factory* SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactoryMysql());  **return** template;  } } |

|  |
| --- |
| @Configuration @MapperScan(basePackages = {**"com.cxp.mapper.oracle"**},sqlSessionFactoryRef = **"sqlSessionFactoryOracle"**) **public class** OracleDataConfig {  @Autowired  @Qualifier(**"primaryDataSource"**)  **private** DataSource **oracleDs**;   @Bean  **public** SqlSessionFactory sqlSessionFactoryOracle() **throws** Exception {  SqlSessionFactoryBean factoryBean = **new** SqlSessionFactoryBean();  factoryBean.setDataSource(**oracleDs**);  **return** factoryBean.getObject();  }   @Bean  **public** SqlSessionTemplate sqlSessionTemplate() **throws** Exception {  *//使用上面配置的factory* SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactoryOracle());  **return** template;  } } |

|  |
| --- |
| @Configuration @MapperScan(basePackages = {**"com.cxp.mapper.postgresql"**}, sqlSessionFactoryRef = **"sqlSessionFactoryPostgresql"**) **public class** PostgresqpDataSource {  @Autowired  @Qualifier(**"secondDataSource"**)  **private** DataSource **postgresqlDs**;   @Bean  **public** SqlSessionFactory sqlSessionFactoryPostgresql() **throws** Exception {  SqlSessionFactoryBean factoryBean = **new** SqlSessionFactoryBean();  factoryBean.setDataSource(**postgresqlDs**);  **return** factoryBean.getObject();  }   @Bean  **public** SqlSessionTemplate sqlSessionTemplate() **throws** Exception {  *//使用上面配置的factory* SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactoryPostgresql());  **return** template;  } } |

4、配置配置文件（application.properties）

|  |
| --- |
| **primary.dataSource.url**=**jdbc:oracle:thin:@localhost:1521:ORCLASUS  primary.dataSource.driver-class-name**=**oracle.jdbc.driver.OracleDriver primary.dataSource.username**=**root primary.dataSource.password**=**123456 primary.dataSource.type**=**com.alibaba.druid.pool.DruidDataSource  second.dataSource.url**=**jdbc:postgresql://192.168.0.114:5432/cheng second.dataSource.driver-class-name**=**org.postgresql.Driver second.dataSource.username**=**root second.dataSource.password**=**123456 second.dataSource.type**=**com.alibaba.druid.pool.DruidDataSource  third.dataSource.url**=**jdbc:mysql://localhost:3306/test?characterEncoding=utf-8 third.dataSource.driver-class-name**=**com.mysql.jdbc.Driver third.dataSource.username**=**root third.dataSource.password**=**123456 third.dataSource.type**=**com.alibaba.druid.pool.DruidDataSource** |

5、主程序入口配置，将spring boot自带的DataSourceAutoConfiguration禁掉

|  |
| --- |
| @SpringBootApplication(exclude = {DataSourceAutoConfiguration.**class**}) **public class** ManydatasourceApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(ManydatasourceApplication.**class**, args);  } } |

6、mapper接口和xml配置文件

|  |
| --- |
| **public interface** MysqlProductMapper {   **public** List<Map<String,Object>> findProductList(); } |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" *>*** <**mapper namespace="com.cxp.mapper.mysql.MysqlProductMapper"**>   <**select id="findProductList" resultType="java.util.Map"**>  select a.id,a.`pname`,a.`price`,a.`productDate`,a.`ptype`,a.`remark` from product a  </**select**> </**mapper**> |

|  |
| --- |
| **public interface** OracleUserRoleMapper {   **public** List<Map<String,Object>> findUserRoleInfoList(); } |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" *>*** <**mapper namespace="com.cxp.mapper.oracle.OracleUserRoleMapper"**>  <**select id="findUserRoleInfoList" resultType="java.util.Map"**>  select a.id,a.userum "userUm",a.roleid "roleId" from t\_user\_role\_info a  </**select**> </**mapper**> |

|  |
| --- |
| **public interface** PostUserInfoMapper {   **public** List<Map<String,Object>> findHubUserInfoList(Map<String,Object> param); } |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" *>*** <**mapper namespace="com.cxp.mapper.postgresql.PostUserInfoMapper"**>   <**select id="findHubUserInfoList" resultType="java.util.Map" parameterType="java.util.Map"**>  SELECT id, userum, "name", password, deptno, "type", status  FROM public.t\_hub\_user\_info WHERE 1=1  <**if test="userUm!=null and userUm!=''"**>  and userum=#{userUm}  </**if**>  <**if test="deptNo!=null and deptNo!=''"**>  and deptno=#{deptNo}  </**if**>  </**select**> </**mapper**> |

第七-0篇 SpringBootg整合Redis

**RedisTemplate**

1. RedisTemplate
2. StringRedisTemplate

RedisTemplate能够让我们持久化各种类型的key和value，并不仅限于字节数组。

StringRedisTemplate扩展了RedisTemplate，只能使用String类型。

StringRedisTemplate有一个接受RedisConnectionFactory的构造器，因此没有必要在构建后在调用setConnectionFactory() .

**使用RedisTemplateAPI**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **方法** | **子API接口** | **描述** | | --- | --- | --- | | opsForValue() | ValueOperations | 描述具有简单值的条目 | | opsForList() | ListOperations | 操作具有list值的条目 | | opsForSet() | SetOperations | 操作具有set值的条目 | | opsForZSet() | ZSetOperations | 操作具有ZSet值（排序的set）的条目 | | opsForHash() | HashOperations | 操作具有hash值的条目 | | boundValueOps(K) | BoundValueOperations | 以绑定指定key的方式，操作具有简单值的条目 | | boundListOps(K) | BoundListOperations | 以绑定指定key的方式，操作具有list的条目 | | boundSetOps(K) | BoundSetOperations | 以绑定指定key的方式，操作具有set的条目 | | boundZSet(K) | BoundZSetOperations | 以绑定指定key的方式，操作具有ZSet（排序的set）的条目 | | boundHashOps(K) | BoundHashOperations | 以绑定指定key的方式，操作具有hash值的条目 | |

①添加Redis依赖

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

②添加Redis相关配置

|  |
| --- |
| spring:  datasource:  url: jdbc:mysql://localhost:3306/test?characterEncoding=utf-8  username: root  password: 123456  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  redis:  host: 192.168.0.116  port: 6379  pool:  max-active: 1000  min-idle: 1  max-wait: -1  max-idle: 100 |

③编写Redis配置类

|  |
| --- |
| @Configuration  @EnableCaching  **public** **class** RedisConfiguration **extends** CachingConfigurerSupport{  @Bean  **public** KeyGenerator keyGenerator(){  **return** **new** KeyGenerator() {  @Override  **public** Object generate(Object target, Method method, Object... params) {  StringBuilder sb=**new** StringBuilder();  sb.append(target.getClass().getName());  sb.append(method.getName());  **for**(Object obj:params){  sb.append(obj.toString());  }  **return** sb.toString();  }  };  }  /\*\*  \* 管理缓存  \* **@param** redisTemplate  \* **@return**  \*/  @Bean  **public** CacheManager cacheManager(RedisTemplate redisTemplate){  RedisCacheManager rcm=**new** RedisCacheManager(redisTemplate);  **return** rcm;  }  /\*\*  \* RedisTemplate配置  \* **@param** redisConnectionFactory  \* **@return**  \*/  @Bean  **public** RedisTemplate<String, String> redisTemplate(RedisConnectionFactory redisConnectionFactory) {  StringRedisTemplate template=**new** StringRedisTemplate(redisConnectionFactory);  Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = **new** Jackson2JsonRedisSerializer(Object.**class**);  ObjectMapper om = **new** ObjectMapper();  om.setVisibility(PropertyAccessor.***ALL***, JsonAutoDetect.Visibility.***ANY***);  om.enableDefaultTyping(ObjectMapper.DefaultTyping.***NON\_FINAL***);  jackson2JsonRedisSerializer.setObjectMapper(om);  template.setValueSerializer(jackson2JsonRedisSerializer);  template.afterPropertiesSet();  **return** template;  }  } |

④编写Redis操作相关Service

|  |
| --- |
| @Service  **public** **class** RedisService {  @Autowired  **private** RedisTemplate redisTemplate;  /\*\*  \* 写入缓存  \*/  **public** **boolean** set(**final** String key, Object value) {  **boolean** result = **false**;  **try** {  ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();  operations.set(key, value);  result = **true**;  } **catch** (Exception e) {  e.printStackTrace();  }  **return** result;  }  /\*\*  \* 写入缓存设置时效时间  \*/  **public** **boolean** set(**final** String key, Object value, Long expireTime) {  **boolean** result = **false**;  **try** {  ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();  operations.set(key, value);  redisTemplate.expire(key, expireTime, TimeUnit.***SECONDS***);  result = **true**;  } **catch** (Exception e) {  e.printStackTrace();  }  **return** **false**;  }  /\*\*  \* 批量删除对应的Key  \* **@param** keys  \*/  **public** **void** remove(**final** String... keys) {  **for** (String key : keys) {  remove(key);  }  }  /\*\*  \* 批量删除key  \*/  **public** **void** removePattern(**final** String pattern) {  Set<Serializable> keys = redisTemplate.keys(pattern);  **if** (keys.size() > 0) {  redisTemplate.delete(keys);  }  }  /\*\*  \* 删除对应的value  \* **@param** key  \*/  **public** **void** remove(**final** String key) {  **if** (exists(key)) {  redisTemplate.delete(key);  }  }  /\*\*  \* 判断缓存中是否有对就的value  \* **@param** key  \* **@return**  \*/  **public** **boolean** exists(**final** String key) {  **return** redisTemplate.hasKey(key);  }  /\*\*  \* 读取缓存  \*/  **public** Object get(**final** String key) {  Object result = **null**;  ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();  result = operations.get(key);  **return** result;  }  /\*\*  \* 哈希添加  \* **@param** key  \* **@param** hashKey  \* **@param** value  \*/  **public** **void** hmSet(String key, Object hashKey, Object value) {  HashOperations<String, Object, Object> hash = redisTemplate.opsForHash();  hash.put(key, hashKey, value);  }  /\*\*  \* 哈希读取数据  \*/  **public** Object hmGet(String key, Object hashKey) {  HashOperations<String, Object, Object> hash = redisTemplate.opsForHash();  **return** hash.get(key, hashKey);  }  /\*\*  \* 列表添加  \*  \* **@param** key  \* **@param** value  \*/  **public** **void** IPush(String key, Object value) {  ListOperations<String, Object> list = redisTemplate.opsForList();  list.rightPush(key, value);  }  /\*\*  \* 列表获取  \*  \* **@param** key  \* **@param** l  \* **@param** l1  \* **@return**  \*/  **public** List<Object> IRange(String key, **long** l, **long** l1) {  ListOperations<String, Object> list = redisTemplate.opsForList();  **return** list.range(key, l, l1);  }  /\*\*  \* 集合添加  \*  \* **@param** key  \* **@param** value  \*/  **public** **void** add(String key, Object value) {  SetOperations<String, Object> set = redisTemplate.opsForSet();  set.add(key, value);  }  /\*\*  \* 集合获取  \*  \* **@param** key  \* **@return**  \*/  **public** Set<Object> getMember(String key) {  SetOperations<String, Object> set = redisTemplate.opsForSet();  **return** set.members(key);  }  /\*\*  \* 有序集合添加  \*  \* **@param** key  \* **@param** value  \* **@param** scoure  \*/  **public** **void** zAdd(String key, Object value, **double** scoure) {  ZSetOperations<String, Object> zset = redisTemplate.opsForZSet();  zset.add(key, value, scoure);  }  /\*\*  \* 有序集合获取  \*  \* **@param** key  \* **@param** scoure  \* **@param** scoure1  \* **@return**  \*/  **public** Set<Object> rangeByScore(String key, **double** scoure, **double** scoure1) {  ZSetOperations<String, Object> zset = redisTemplate.opsForZSet();  **return** zset.rangeByScore(key, scoure, scoure1);  }  } |

**一. spring-data-redis针对jedis提供了如下功能**：

1. 连接池自动管理，提供了一个高度封装的“RedisTemplate”类

2. 针对jedis客户端中大量api进行了归类封装,将同一类型操作封装为operation接口:

ValueOperations：简单K-V操作

SetOperations：set类型数据操作

ZSetOperations：zset类型数据操作

HashOperations：针对map类型的数据操作

ListOperations：针对list类型的数据操作

3. 提供了对key的“bound”(绑定)便捷化操作API，可以通过bound封装指定的key，然后进行一系列的操作而无须“显式”的再次指定Key，即

BoundKeyOperations：

BoundValueOperations

BoundSetOperations

BoundListOperations

BoundSetOperations

BoundHashOperations

4.将事务操作封装，有容器控制。

5.针对数据的“序列化/反序列化”，提供了多种可选择策略(RedisSerializer):

如果要使用到JavaBean，需要其实现Serializable接口，将其序列化或者使用Spring Data Redis提供的序列化器

GenericToStringSerializer：使用Spring转换服务进行序列化

JacksonJsonRedisSerializer：使用Jackson1，将对象序列化为JSON

Jackson2JsonRedisSerializer：使用Jackson2，将对象序列化为JSON

JdkSerializationRedisSerializer：使用Java序列化

OxmSerializer：使用Spring O/X映射的编排器和解排器实现序列化，用于XML序列化

StringRedisSerializer：序列化String类型的key和value

⑤**controller**测试

|  |
| --- |
| @RestController  @RequestMapping("/redis")  **public** **class** RedisController {  @Autowired  **private** RedisService redisService;    @RequestMapping("/redisTest")  **public** String redisTest(){  //先在redis中删除key  redisService.remove("str","hmset","list");  StringBuffer sb=**new** StringBuffer();  redisService.set("str", "张三丰");  sb.append("str=").append(redisService.get("str").toString()).append(";");  redisService.hmSet("hmset", "key1", "唐嫣");  redisService.hmSet("hmset", "key2", "罗晋");  sb.append("hmset:key1=").append(redisService.hmGet("hmset", "key1")).append(",")  .append("key2").append(redisService.hmGet("hmset", "key2")).append(";");  redisService.IPush("list", "val1");  redisService.IPush("list", "val2");  redisService.IPush("list", "val2");  redisService.IPush("list", "val2");  sb.append("list=").append(redisService.IRange("list", 0, 2).toString()).append(";");  redisService.removeList("list", 2, "val2");  redisService.zAdd("zset", "val1\_zset", 1);  redisService.zAdd("zset", "val2\_zset", 2);  sb.append("zset=").append(redisService.rangeByScore("zset", 1, 2)).append(";");  //页面输出:  // str=张三丰;hmset:key1=唐嫣,key2罗晋;list=[val1, val2, val2];zset=[val1\_zset, val2\_zset];  **return** sb.toString();  }  } |

第七-1篇 SpringBootg手动配置Redis

①在pom.xml文件中添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**> </**dependency**> |

②在配置文件application.properties中配置redis连接设置

|  |
| --- |
| **my.redis.host**=**192.168.0.118 my.redis.port**=**6379 my.redis.pool.max\_active**=**1000 my.redis.pool.min\_idle**=**5 my.redis.pool.max\_wait**=**-1 my.redis.pool.max\_idle**=**100** |

③把配置信息注入到一个类中

|  |
| --- |
| @ConfigurationProperties(prefix = **"my.redis"**) **public class** MyRedisProperties {  **private int port**;  **private** String **host**;  **private** MyRedisProperties.Pool **pool**;   **public int** getPort() {  **return port**;  }   **public** String getHost() {  **return host**;  }   **public** Pool getPool() {  **return pool**;  }   **public void** setPort(**int** port) {  **this**.**port** = port;  }   **public void** setHost(String host) {  **this**.**host** = host;  }   **public void** setPool(Pool pool) {  **this**.**pool** = pool;  }   **public static class** Pool{  **private int max\_active** ;  **private int min\_idle** ;  **private int max\_wait** ;  **private int max\_idle** ;   **public** Pool() {  }   **public int** getMax\_active() {  **return max\_active**;  }   **public int** getMin\_idle() {  **return min\_idle**;  }   **public int** getMax\_wait() {  **return max\_wait**;  }   **public int** getMax\_idle() {  **return max\_idle**;  }   **public void** setMax\_active(**int** max\_active) {  **this**.**max\_active** = max\_active;  }   **public void** setMin\_idle(**int** min\_idle) {  **this**.**min\_idle** = min\_idle;  }   **public void** setMax\_wait(**int** max\_wait) {  **this**.**max\_wait** = max\_wait;  }   **public void** setMax\_idle(**int** max\_idle) {  **this**.**max\_idle** = max\_idle;  }  } } |

④配置Redis

|  |
| --- |
| @Configuration @EnableCaching **public class** RedisConfiguration **extends** CachingConfigurerSupport {  **static final** Logger ***logger***= LoggerFactory.*getLogger*(RedisConfiguration.**class**.getName());   @Bean(name = {**"myRedisProperties"**})  @ConditionalOnMissingBean  **public** MyRedisProperties myRedisProperties() {  **return new** MyRedisProperties();  }  @Autowired  **private** MyRedisProperties **myRedisProperties**;  @Bean  **public** KeyGenerator keyGenerator(){  **return new** KeyGenerator() {   @Override  **public** Object generate(Object target, Method method, Object... params) {  StringBuilder sb=**new** StringBuilder();  sb.append(target.getClass().getName());  sb.append(method.getName());  **for**(Object obj:params){  sb.append(obj.toString());  }  **return** sb.toString();  }  };  }  */\*\*  \* 管理缓存  \** ***@param redisTemplate*** *\** ***@return*** *\*/* @Bean  **public** CacheManager cacheManager(RedisTemplate redisTemplate){  RedisCacheManager rcm=**new** RedisCacheManager(redisTemplate);  **return** rcm;  }  */\*\*  \* RedisTemplate配置  \** ***@param redisConnectionFactory*** *\** ***@return*** *\*/* @Bean  **public** RedisTemplate<String, String> redisTemplate(RedisConnectionFactory redisConnectionFactory) {  StringRedisTemplate template=**new** StringRedisTemplate(redisConnectionFactory);  StringRedisSerializer stringRedisSerializer=**new** StringRedisSerializer();  template.setStringSerializer(stringRedisSerializer);  **return** template;  }  */\*\*  \* 配置redis服务器连接  \** ***@return*** *\*/*  @Bean  **public** JedisConnectionFactory getJedisConnectionFactory(){  JedisPoolConfig jedisPoolConfig=**new** JedisPoolConfig();  jedisPoolConfig.setMinIdle(**myRedisProperties**.getPool().getMin\_idle());  jedisPoolConfig.setMaxIdle(**myRedisProperties**.getPool().getMax\_idle());  jedisPoolConfig.setMaxWaitMillis(**myRedisProperties**.getPool().getMax\_wait());  JedisConnectionFactory jedisConnectionFactory=**new** JedisConnectionFactory(jedisPoolConfig);  jedisConnectionFactory.setPort(**myRedisProperties**.getPort());  jedisConnectionFactory.setHostName(**myRedisProperties**.getHost());**return** jedisConnectionFactory;  } } |

⑤测试类测试

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** ManydatasourceApplicationTests {  @Autowired  **private** StringRedisTemplate **redisTemplate**;  @Autowired  **private** MyRedisProperties **myRedisProperties**;  @Test  **public void** contextLoads() {  System.***out***.println(**myRedisProperties**.getHost());  System.***out***.println(**redisTemplate**);  System.***out***.println(**redisTemplate**.opsForValue().get(**"address"**));  System.***out***.println(**redisTemplate**.opsForHash().get(**"myhash"**,**"age"**));  System.***out***.println(RedisUtil.*getHash*(**redisTemplate**,**"myhash"**,**"username"**));  } } |

第七-2篇 SpringBootg排除RedisAutoConfiguration

①在pom.xml文件中添加依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-configuration-processor</**artifactId**>  <**optional**>true</**optional**> </**dependency**> |

②SpringBoot启动类

|  |
| --- |
| @SpringBootApplication(exclude = RedisAutoConfiguration.**class**) @EnableEurekaClient @MapperScan(value =**"com.cxp.mapper"** ) **public class** ProviderUserApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(ProviderUserApplication.**class**, args);  } } |

③在配置文件application.properties中配置redis连接设置

|  |
| --- |
| **spring:**  **redis:  host:** 192.168.0.133  **port:** 6379  **pool:  max-active:** 8  **max-wait:** -1  **max-idle:** 8  **min-idle:** 0  **timeout:** 0  **database:** 0 |

④把配置信息注入到一个类中

|  |
| --- |
| @ConfigurationProperties(prefix = **"spring.redis"**) **public class** MyRedisProperties {  **private int database** = 0;    **private** String **url**;   **private** String **host** = **"localhost"**;   **private** String **password**;   **private int port** = 6379;   **private boolean ssl**;   **private int timeout**;   **private** RedisProperties.Pool **pool**;   **private** RedisProperties.Sentinel **sentinel**;   **private** RedisProperties.Cluster **cluster**;   **public int** getDatabase() {  **return this**.**database**;  }   **public void** setDatabase(**int** database) {  **this**.**database** = database;  }   **public** String getUrl() {  **return this**.**url**;  }   **public void** setUrl(String url) {  **this**.**url** = url;  }   **public** String getHost() {  **return this**.**host**;  }   **public void** setHost(String host) {  **this**.**host** = host;  }   **public** String getPassword() {  **return this**.**password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   **public int** getPort() {  **return this**.**port**;  }   **public void** setPort(**int** port) {  **this**.**port** = port;  }   **public boolean** isSsl() {  **return this**.**ssl**;  }   **public void** setSsl(**boolean** ssl) {  **this**.**ssl** = ssl;  }   **public void** setTimeout(**int** timeout) {  **this**.**timeout** = timeout;  }   **public int** getTimeout() {  **return this**.**timeout**;  }   **public** RedisProperties.Sentinel getSentinel() {  **return this**.**sentinel**;  }   **public void** setSentinel(RedisProperties.Sentinel sentinel) {  **this**.**sentinel** = sentinel;  }   **public** RedisProperties.Pool getPool() {  **return this**.**pool**;  }   **public void** setPool(RedisProperties.Pool pool) {  **this**.**pool** = pool;  }   **public** RedisProperties.Cluster getCluster() {  **return this**.**cluster**;  }   **public void** setCluster(RedisProperties.Cluster cluster) {  **this**.**cluster** = cluster;  }   **public static class** Pool {  **private int maxIdle** = 8;  **private int minIdle** = 0;  **private int maxActive** = 8;  **private int maxWait** = -1;   **public int** getMaxIdle() {  **return this**.**maxIdle**;  }   **public void** setMaxIdle(**int** maxIdle) {  **this**.**maxIdle** = maxIdle;  }   **public int** getMinIdle() {  **return this**.**minIdle**;  }   **public void** setMinIdle(**int** minIdle) {  **this**.**minIdle** = minIdle;  }   **public int** getMaxActive() {  **return this**.**maxActive**;  }   **public void** setMaxActive(**int** maxActive) {  **this**.**maxActive** = maxActive;  }   **public int** getMaxWait() {  **return this**.**maxWait**;  }   **public void** setMaxWait(**int** maxWait) {  **this**.**maxWait** = maxWait;  }   }  } |

⑤配置Redis

|  |
| --- |
| @Configuration @EnableCaching @EnableConfigurationProperties(MyRedisProperties.**class**) **public class** RedisConfig **extends** CachingConfigurerSupport {   @Autowired  **private** MyRedisProperties **redisProperties**;   */\*\*  \* 管理缓存  \** ***@param redisTemplate*** *\** ***@return*** *\*/* @Bean  **public** CacheManager cacheManager(RedisTemplate redisTemplate){  RedisCacheManager rcm=**new** RedisCacheManager(redisTemplate);  **return** rcm;  }   @Bean  **public** JedisConnectionFactory jedisConnectionFactory() {  JedisConnectionFactory factory = **new** JedisConnectionFactory();  factory.setDatabase(**redisProperties**.getDatabase());  factory.setHostName(**redisProperties**.getHost());  factory.setPort(**redisProperties**.getPort());  factory.setTimeout(**redisProperties**.getTimeout());   JedisPoolConfig poolConfig=**new** JedisPoolConfig();  poolConfig.setMaxIdle(**redisProperties**.getPool().getMaxIdle());  poolConfig.setMaxWaitMillis(**redisProperties**.getPool().getMaxWait());  poolConfig.setMaxTotal(**redisProperties**.getPool().getMaxActive());  poolConfig.setMinIdle(**redisProperties**.getPool().getMinIdle());   factory.setPoolConfig(poolConfig);  **return** factory;  }   @Bean  **public** KeyGenerator keyGenerator() {  **return new** KeyGenerator() {   @Override  **public** Object generate(Object target, Method method, Object... params) {  StringBuilder sb = **new** StringBuilder();  sb.append(target.getClass().getName());  sb.append(method.getName());  **for** (Object obj : params) {  sb.append(obj.toString());  }  **return** sb.toString();  }  };  }   @Bean  @Primary  **public** RedisTemplate<String, String> redisTemplate(RedisConnectionFactory redisConnectionFactory) {  RedisTemplate<String,String> template = **new** RedisTemplate();  template.setConnectionFactory(redisConnectionFactory);  Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = **new** Jackson2JsonRedisSerializer(Object.**class**);  ObjectMapper om = **new** ObjectMapper();  om.setVisibility(PropertyAccessor.***ALL***, JsonAutoDetect.Visibility.***ANY***);  om.enableDefaultTyping(ObjectMapper.DefaultTyping.***NON\_FINAL***);  jackson2JsonRedisSerializer.setObjectMapper(om);  template.setValueSerializer(jackson2JsonRedisSerializer);  template.afterPropertiesSet();  System.***out***.println(**"redisTemplate======="**+template);  **return** template;  }   @Bean(name=**"stringRedisTemplate"**)  **public** StringRedisTemplate stringRedisTemplate(RedisConnectionFactory redisConnectionFactory){  StringRedisTemplate stringRedisTemplate=**new** StringRedisTemplate(redisConnectionFactory);  stringRedisTemplate.setStringSerializer(**new** StringRedisSerializer());  System.***out***.println(**"stringRedisTemplate======="**+stringRedisTemplate);  **return** stringRedisTemplate;  } } |

⑥测试类测试

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** ProviderUserApplicationTests {   @Resource(name = **"stringRedisTemplate"**)  **private** StringRedisTemplate **stringRedisTemplate**;   @Resource  **private** RedisTemplate **redisTemplate**;   @Test  **public void** contextLoads() {  String name = **stringRedisTemplate**.opsForValue().get(**"name"**);  Object age = **redisTemplate**.boundValueOps(**"age"**).get();  **redisTemplate**.boundValueOps(**"address1"**).set(**"湖南长沙"**);  Object address1 = **redisTemplate**.boundValueOps(**"address1"**).get();  System.***out***.println(**redisTemplate**);  System.***out***.println(**redisTemplate**.getValueSerializer());   User user = **new** User();  user.setId(1);  user.setAddress(**"湖南长沙"**);  user.setName(**"长沙人"**);  user.setBirthday(**new** Date());  **redisTemplate**.opsForValue().set(**"userObject"**, user);  User user1 = (User) **redisTemplate**.opsForValue().get(**"userObject"**);  System.***out***.println(**"reids :name= "** + name + **", age ="** + age + **", address1 ="** + address1 + **"\n user1 ==="** + user1);  }  } |

第八篇 SpringBoot整合Mail

①在pom.xml文件中添加依赖

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-mail -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-mail</artifactId>  </dependency> |

②在application.yml中配置

|  |
| --- |
| spring:  mail:  host: smtp.163.com  username: cheng3\_shan3@163.com  password: 3862398pe  port: 25  protocol: smtp  defaultEncoding: utf-8 |

③测试发邮件(发送mp3和图片)

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** AppTest {  @Autowired  **private** JavaMailSenderImpl mailSender;  /\*\*  \* 发送包含附件的邮件  \*  \* **@throws** Exception  \*/  @Test  **public** **void** sendAttachedFileMail() **throws** Exception {  MimeMessage mimeMessage = mailSender.createMimeMessage();  // multipart模式  MimeMessageHelper mimeMessageHelper = **new** MimeMessageHelper(mimeMessage, **true**, "utf-8");  // 收件人  mimeMessageHelper.setTo(**new** String[] { "cheng3\_shan3@126.com", "276629352@qq.com" });  // 抄送人  mimeMessageHelper.setCc("675194065@qq.com");  // 发件人  mimeMessageHelper.setFrom("cheng3\_shan3@163.com");  // 设置邮件主题  mimeMessageHelper.setSubject("springboot mail");  StringBuilder sb = **new** StringBuilder();  sb.append("<html><head>这是一首音乐</head>");  sb.append("<body><h1>spring </h1><p>hello!this is music!</p>");  // cid为固定写法，imageId指定一个标识  // sb.append("<img src=\"cid:imageId\"/></body>");  sb.append("</html>");  // 启用html  mimeMessageHelper.setText(sb.toString(), **true**);  // 设置imageId  FileSystemResource mp3 = **new** FileSystemResource(**new** File("E:/kugou/冯提莫 - 刚好遇见你.mp3"));  mimeMessageHelper.addAttachment("冯提莫 - 刚好遇见你.mp3", mp3);  // 发送邮件  mailSender.send(mimeMessage);  System.***out***.println("邮件已发送");  }  /\*\*  \* 发送包含内嵌图片的邮件  \*  \* **@throws** Exception  \*/  @Test  **public** **void** sendAttachedImageMail() **throws** Exception {  MimeMessage mimeMessage = mailSender.createMimeMessage();  // multipart模式  MimeMessageHelper mimeMessageHelper = **new** MimeMessageHelper(mimeMessage, **true**);  mimeMessageHelper.setTo("cheng3\_shan3@126.com");  mimeMessageHelper.setFrom("cheng3\_shan3@163.com");  mimeMessageHelper.setSubject("Spring Boot Mail 邮件测试【图片】");  StringBuilder sb = **new** StringBuilder();  sb.append("<html><head></head>");  sb.append("<body><h1>spring 邮件测试</h1><p>hello!this is spring mail test。</p>");  // cid为固定写法，imageId指定一个标识  sb.append("<img src=\"cid:imageId\"/></body>");  sb.append("</html>");  // 启用html  mimeMessageHelper.setText(sb.toString(), **true**);  // 设置imageId  FileSystemResource img = **new** FileSystemResource(**new** File("C:\\Users\\ASUS\\Pictures\\Saved Pictures\\desk1.jpg"));  mimeMessageHelper.addInline("imageId", img);  // 发送邮件  mailSender.send(mimeMessage);  System.***out***.println("邮件已发送");  }  } |

第九篇 整合Shiro安全框架

**1.What is Apache Shiro?**

Apache Shiro是一个功能强大、灵活的，开源的安全框架。它可以干净利落地处理身份验证、授权、企业会话管理和加密。

Shiro能做什么呢？

>>验证用户身份

>>用户访问权限控制，比如：1、判断用户是否分配了一定的安全角色。2、判断用户是否被授予完成某个操作的权限

>>在非 web 或 EJB 容器的环境下可以任意使用Session API

>>可以响应认证、访问控制，或者 Session 生命周期中发生的事件

>>可将一个或以上用户安全数据源数据组合成一个复合的用户 “view”(视图)

>>支持单点登录(SSO)功能

>>支持提供“Remember Me”服务，获取用户关联信息而无需登录

**2.Apache Shiro Features 特性**

Apache Shiro是一个全面的、蕴含丰富功能的安全框架。下图为描述Shiro功能的框架图：

|  |
| --- |
|  |

Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）被 Shiro 框架的开发团队称之为应用安全的四大基石。那么就让我们来看看它们吧：

Authentication（认证）：用户身份识别，通常被称为用户“登录”

Authorization（授权）：访问控制。比如某个用户是否具有某个操作的使用权限。

Session Management（会话管理）：特定于用户的会话管理,甚至在非web 或 EJB 应用程序。

Cryptography（加密）：在对数据源使用加密算法加密的同时，保证易于使用。

还有其他的功能来支持和加强这些不同应用环境下安全领域的关注点。特别是对以下的功能支持：

>> Web支持：Shiro 提供的 web 支持 api ，可以很轻松的保护 web 应用程序的安全。

>> 缓存：缓存是 Apache Shiro 保证安全操作快速、高效的重要手段。

>> 并发：Apache Shiro 支持多线程应用程序的并发特性。

>> 测试：支持单元测试和集成测试，确保代码和预想的一样安全。

>> “Run As”：这个功能允许用户假设另一个用户的身份(在许可的前提下)。

>>“Remember Me”：跨 session 记录用户的身份，只有在强制需要时才需要登录。

**3.Shiro集成使用**

①在pom.xml文件中添加依赖

|  |
| --- |
| *<!-- 使用shiro认证 、缓存-->*  <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-spring</**artifactId**>  <**version**>1.4.0</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-ehcache</**artifactId**>  <**version**>1.4.0</**version**> </**dependency**>  *<!-- 在Thymeleaf模板引擎中集成shiro -->*  <**dependency**>  <**groupId**>com.github.theborakompanioni</**groupId**>  <**artifactId**>thymeleaf-extras-shiro</**artifactId**>  <**version**>1.2.1</**version**> </**dependency**>  <!-- 在Spring Boot中使用Thymeleaf模板引擎 -->  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**>  *<!-- 第三方redis缓存管理器 -->*  <**dependency**>  <**groupId**>org.crazycake</**groupId**>  <**artifactId**>shiro-redis</**artifactId**>  <**version**>2.8.24</**version**> </**dependency**>  *<!-- 第三方验证码生成jar包 -->* <**dependency**>  <**groupId**>com.github.penggle</**groupId**>  <**artifactId**>kaptcha</**artifactId**>  <**version**>2.3.2</**version**>  <**exclusions**>  <**exclusion**>  <**artifactId**>javax.servlet-api</**artifactId**>  <**groupId**>javax.servlet</**groupId**>  </**exclusion**>  </**exclusions**> </**dependency**> |

②在application.properties配置文件中配置

|  |
| --- |
| **server.port** = **8088 server.context-path**=**/** *#单位秒* **spring.devtools.restart.enabled**=**true** *#开发时关闭缓存,不然没法看到实时页面 #spring.thymeleaf.cache=false #spring.thymeleaf.encoding=UTF-8 #spring.thymeleaf.prefix=classpath:/templates/ #spring.thymeleaf.suffix=.html #spring.thymeleaf.content-type=text/html #数据源配置* **spring.datasource.url**=**jdbc:postgresql://192.168.0.150/postgres spring.datasource.username**=**postgres spring.datasource.password**=**123456 spring.datasource.driver-class-name**=**org.postgresql.Driver datasource.initialize**=**6 datasource.maxActive**=**60 datasource.minIdle**=**2 datasource.maxWait**=**60000 datasource.validationQuery**=**select 1** *# 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙* **datasource.filters**=**stat,wall,log4j** *# 通过connectProperties属性来打开mergeSql功能；慢SQL记录* **datasource.connectionProperties**=**druid.stat.mergeSql:true;druid.stat.slowSqlMillis:5000** *# 合并多个DruidDataSource的监控数据* **useGlobalDataSourceStat**=**true  spring.redis.host**=**192.168.0.150 spring.redis.port**=**6379 spring.redis.password**=**123456 spring.redis.pool.max-active**=**30 spring.redis.pool.max-idle**=**20 spring.redis.pool.min-idle**=**5 spring.redis.pool.max-wait**=**60000 spring.redis.timeout**=**30000** *#mybatis配置* **mybatis.mapper-locations**=**classpath:static/mybatis/\*Mapper.xml** *#扫描实体包中实体，得到别名* **mybatis.type-aliases-package**=**com.cxp.personalmanage.pojo  logging.pattern.level**=**debug logging.pattern.console**=**%d [%t] %-5p [%c] - %m%n** |

③自定义Realm实现登录验证和授权

|  |
| --- |
| **public class** UserRealm **extends** AuthorizingRealm {  **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(UserRealm.**class**);  @Autowired  **private** UserService **userService**;  @Autowired  **private** RoleService **roleService**;   @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {  String currentLoginName = (String) principals.getPrimaryPrincipal();  List<String> userRoles = **new** ArrayList<>();  List<String> userPermissions = **new** ArrayList<>();  *//从数据库中获取当前登录用户的详细信息* UserInfo userInfo = **userService**.findByLoginName(currentLoginName);  **if** (userInfo != **null**) {  *//获取当前用户下所有权限列表  //获取当前用户下拥有的所有角色列表* List<RoleInfo> roles = **roleService**.fingByUserId(userInfo.getId());  **for** (**int** i = 0; i < roles.size(); i++) {  userRoles.add(roles.get(i).getRoleid());  }  } **else** {  **throw new** AuthorizationException();  }  ***logger***.info(**"获取角色："** + userRoles);  ***logger***.info(**"获取权限："** + userPermissions);  *//为当前角色设置角色和权限* SimpleAuthorizationInfo authorizationInfo = **new** SimpleAuthorizationInfo();  authorizationInfo.addRoles(userRoles);  authorizationInfo.addStringPermissions(userPermissions);  **return** authorizationInfo;  }   */\*\*  \* 验证当前登录的subject  \*  \** ***@param token*** *\** ***@return*** *\** ***@throws*** *AuthenticationException  \*/* @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token) **throws** AuthenticationException {  String loginName = (String) token.getPrincipal();  UserInfo userInfo = **userService**.findByLoginName(loginName);  **if** (userInfo == **null**) {  *//没有找到账号* **throw new** UnknownAccountException();  }  *//交给AuthenticatingRealm使用CredentialsMatcher进行密码匹配，* SimpleAuthenticationInfo authenticationInfo = **new** SimpleAuthenticationInfo(  userInfo.getUsername(),  userInfo.getPassword(),  ByteSource.Util.*bytes*(userInfo.getSalt()),  getName()  );  **return** authenticationInfo;  } } |

④密码多次输入错误

CredentialsMatcher是shiro提供的用于加密密码和验证密码服务的接口，而HashedCredentialsMatcher正是CredentialsMatcher的一个实现类.

|  |
| --- |
| */\*\*  \* 密码多次输入错误  CredentialsMatcher是shiro提供的用于加密密码和验证密码服务的接口，而HashedCredentialsMatcher正是CredentialsMatcher的一个实现类  \*/* **public class** RetryLimitHashedCredentialsMatcher **extends** HashedCredentialsMatcher {  **private** Cache<String, AtomicInteger> **passwordRetryCache**;   **public** RetryLimitHashedCredentialsMatcher(CacheManager cacheManager) {  **passwordRetryCache** = cacheManager.getCache(**"passwordRetryCache"**);  }   @Override  **public boolean** doCredentialsMatch(AuthenticationToken token,  AuthenticationInfo info) {  String username = (String) token.getPrincipal();  *// retry count + 1* AtomicInteger retryCount = **passwordRetryCache**.get(username);  **if** (retryCount == **null**) {  retryCount = **new** AtomicInteger(1);  **passwordRetryCache**.put(username, retryCount);  ***logger***.info(**"用户"**+username+**"第"**+retryCount+**"次登录."**);  }**else** {  retryCount.incrementAndGet();  **passwordRetryCache**.put(username, retryCount);  ***logger***.info(**"用户"**+username+**"第"**+retryCount+**"次登录."**);  }  **if** (retryCount.incrementAndGet() > 5) {  *// if retry count > 5 throw* **throw new** ExcessiveAttemptsException();  }   **boolean** matches = **super**.doCredentialsMatch(token, info);  **if** (matches) {  *// clear retry count* **passwordRetryCache**.remove(username);  }  **return** matches;  }  } |

⑤配置Shiro的组件

|  |
| --- |
| 1.LifecycleBeanPostProcessor，这是个DestructionAwareBeanPostProcessor的子类，负责org.apache.shiro.util.Initializable类型bean的生命周期的，初始化和销毁。主要是AuthorizingRealm类的子类，以及EhCacheManager类。  2.HashedCredentialsMatcher，这个类是为了对密码进行编码的，防止密码在数据库里明码保存，当然在登陆认证的生活，这个类也负责对form里输入的密码进行编码。  3.ShiroRealm，这是个自定义的认证类，继承自AuthorizingRealm，负责用户的认证和权限的处理，可以参考JdbcRealm的实现。  4.EhCacheManager，缓存管理，用户登陆成功后，把用户信息和权限信息缓存起来，然后每次用户请求时，放入用户的session中，如果不设置这个bean，每个请求都会查询一次数据库。  5.SecurityManager，权限管理，这个类组合了登陆，登出，权限，session的处理，是个比较重要的类。  6.ShiroFilterFactoryBean，是个factorybean，为了生成ShiroFilter。它主要保持了三项数据，securityManager，filters，filterChainDefinitionManager。  7.DefaultAdvisorAutoProxyCreator，Spring的一个bean，由Advisor决定对哪些类的方法进行AOP代理。  8.AuthorizationAttributeSourceAdvisor，shiro里实现的Advisor类，内部使用AopAllianceAnnotationsAuthorizingMethodInterceptor来拦截用以下注解的方法。 |
| **import** java.util.LinkedHashMap; **import** java.util.Map;  **import** javax.servlet.Filter;  **import** org.apache.shiro.authc.credential.HashedCredentialsMatcher; **import** org.apache.shiro.cache.CacheManager; **import** org.apache.shiro.mgt.SecurityManager; **import** org.apache.shiro.session.SessionListener; **import** org.apache.shiro.session.mgt.SessionManager; **import** org.apache.shiro.session.mgt.eis.RandomSessionIdGenerator; **import** org.apache.shiro.session.mgt.eis.SessionDAO; **import** org.apache.shiro.spring.LifecycleBeanPostProcessor; **import** org.apache.shiro.spring.web.ShiroFilterFactoryBean; **import** org.apache.shiro.web.filter.AccessControlFilter; **import** org.apache.shiro.web.mgt.DefaultWebSecurityManager; **import** org.crazycake.shiro.ObjectSerializer; **import** org.crazycake.shiro.RedisCacheManager; **import** org.crazycake.shiro.RedisManager; **import** org.crazycake.shiro.RedisSessionDAO; **import** org.crazycake.shiro.StringSerializer; **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  **import** com.cxp.personalmanage.shiro.CustomSessionManager; **import** com.cxp.personalmanage.shiro.RetryLimitHashedCredentialsMatcher; **import** com.cxp.personalmanage.shiro.UserRealm; **import** com.cxp.personalmanage.shiro.filter.KickoutSessionControlFilter; **import** com.cxp.personalmanage.shiro.listener.MyShiroSessionListener;  @Configuration **public class** ShrioConfig\_crazycake {  **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ShrioConfig\_crazycake.**class**);   @Bean(name = **"shiroFilter"**)  **public** ShiroFilterFactoryBean shiroFilterFactoryBean(SecurityManager securityManager) {  ***logger***.info(**"注入Shiro的web过滤器：ShiroFilter"**, ShiroFilterFactoryBean.**class**);  ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  *// Shiro的核心安全接口，这个属性必须的* shiroFilterFactoryBean.setSecurityManager(securityManager);   *//自定义拦截器* Map<String, Filter> filtersMap = **new** LinkedHashMap<>(); *// 过滤器  //限制同一帐号同时在线的个数。* filtersMap.put(**"kickout"**, kickoutFilter());  shiroFilterFactoryBean.setFilters(filtersMap);   *// 用户访问未对其授权的资源时,所显示的连接* shiroFilterFactoryBean.setUnauthorizedUrl(**"/html/403.html"**);  *// 要求登录时的链接(可根据项目的URL进行替换),非必须的属性,如果不设置默认会自动寻找Web工程根目录下的"/login.jsp"页面* shiroFilterFactoryBean.setLoginUrl(**"/html/login.html"**);  shiroFilterFactoryBean.setSuccessUrl(**"/html/index.html"**);   *// 拦截器.* Map<String, String> filterChainDefinitionMap = **new** LinkedHashMap<String, String>();  *// 配置退出过滤器,其中的具体的退出代码Shiro已经替我们实现了* filterChainDefinitionMap.put(**"/logout"**, **"logout"**);  *// <!-- 过滤链定义，从上向下顺序执行，一般将 /\*\*放在最为下边 -->:这是一个坑呢，一不小心代码就不好使了;  // <!-- authc:所有url都必须认证通过才可以访问; anon:所有url都都可以匿名访问-->* filterChainDefinitionMap.put(**"/js/\*\*"**, **"anon"**);  filterChainDefinitionMap.put(**"/html/login.html"**, **"anon"**);  filterChainDefinitionMap.put(**"/login"**, **"anon"**);  filterChainDefinitionMap.put(**"/defaultKaptcha"**, **"anon"**);  filterChainDefinitionMap.put(**"/html/kickout.html"**, **"anon"**);  filterChainDefinitionMap.put(**"/\*\*"**, **"kickout,authc"**);   shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);   **return** shiroFilterFactoryBean;  }   */\*\*  \* 不指定名字的话，自动创建一个方法名第一个字母小写的bean  \*  \** ***@return*** *\** ***@Bean(name*** *= "securityManager")  \*/* @Bean  **public** SecurityManager securityManager() {  ***logger***.info(**"注入Shiro的安全管理器-->securityManager"**, SecurityManager.**class**);  DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  *// 设置realm,用于获取认证* securityManager.setRealm(userRealm());  *// 自定义session管理 使用redis* securityManager.setSessionManager(sessionManager());  *// 自定义缓存实现 使用redis* securityManager.setCacheManager(cacheManager());*// 这个如果执行多次，也是同样的一个对象;* **return** securityManager;  }   */\*\*  \* Shiro Realm 继承自AuthorizingRealm的自定义Realm,即指定Shiro验证用户登录的类为自定义的  \*  \** ***@param*** *//cacheManager  \** ***@return*** *\*/* @Bean  **public** UserRealm userRealm() {  UserRealm userRealm = **new** UserRealm();  *// 告诉realm,使用credentialsMatcher加密算法类来验证密文* userRealm.setCredentialsMatcher(hashedCredentialsMatcher());  userRealm.setCachingEnabled(**true**);  *// 启用缓存,默认是false userRealm.setCachingEnabled(true); //  // 启用身份验证缓存，即缓存AuthenticationInfo信息，默认false；* userRealm.setAuthenticationCachingEnabled(**true**); *//  // 缓存AuthenticationInfo信息的缓存名称,即配置在ehcache.xml中的cache name* userRealm.setAuthenticationCacheName(**"authenticationCache"**); *//  // 启用授权缓存，即缓存AuthorizationInfo信息，默认false；* userRealm.setAuthorizationCachingEnabled(**true**); *//  // 缓存AuthorizationInfo信息的缓存名称；* userRealm.setAuthorizationCacheName(**"authorizationCache"**);   **return** userRealm;  }   */\*\*  \* 凭证匹配器 （由于我们的密码校验交给Shiro的SimpleAuthenticationInfo进行处理了  \* 所以我们需要修改下doGetAuthenticationInfo中的代码; ） 可以扩展凭证匹配器，实现 输入密码错误次数后锁定等功能，下一次  \*  \** ***@return*** *\*/* @Bean(name = **"credentialsMatcher"**)  **public** HashedCredentialsMatcher hashedCredentialsMatcher() { *// HashedCredentialsMatcher credentialsMatcher = new HashedCredentialsMatcher();* RetryLimitHashedCredentialsMatcher credentialsMatcher =**new** RetryLimitHashedCredentialsMatcher(cacheManager());  *// 散列算法:这里使用MD5算法;* credentialsMatcher.setHashAlgorithmName(**"md5"**);  *// 散列的次数，比如散列两次，相当于 md5(md5(""));* credentialsMatcher.setHashIterations(3);  *// storedCredentialsHexEncoded默认是true，此时用的是密码加密用的是Hex编码；false时用Base64编码* credentialsMatcher.setStoredCredentialsHexEncoded(**true**);  **return** credentialsMatcher;  }   @Bean  **public** SessionManager sessionManager() { *// DefaultWebSessionManager sessionManager = new DefaultWebSessionManager();   // <!-- 删除失效的session -->* CustomSessionManager sessionManager = **new** CustomSessionManager();  *// <!-- 会话超时时间，单位：毫秒 -->* sessionManager.setGlobalSessionTimeout(600000);  *//<!-- 删除失效的session -->* sessionManager.setDeleteInvalidSessions(**true**);  *//是否定期检查Session的有效性* sessionManager.setSessionValidationSchedulerEnabled(**true**);  sessionManager.setCacheManager(cacheManager());  *// <!-- Shiro提供SessionDAO用于会话的CRUD -->* sessionManager.setSessionDAO(sessionDAO());  *// <!-- Shiro中session监听器 -->*  sessionManager.getSessionListeners().add(sessionListener());  **return** sessionManager;  }    @Bean  **public** RedisManager redisManager(){  org.crazycake.shiro.RedisManager redisManager = **new** RedisManager();  redisManager.setHost(**"192.168.0.150"**);  redisManager.setPort(6379);  redisManager.setExpire(60000);*// 配置缓存过期时间* redisManager.setTimeout(30000);  redisManager.setPassword(**"123456"**);  **return** redisManager;  }   @Bean  **public** SessionDAO sessionDAO() {  org.crazycake.shiro.RedisSessionDAO redisSessionDAO = **new** RedisSessionDAO();  *//设置生成sessionId生成策略:  //RandomSessionIdGenerator() 和 JavaUuidSessionIdGenerator()* redisSessionDAO.setSessionIdGenerator(**new** RandomSessionIdGenerator());  redisSessionDAO.setExpire(600);  redisSessionDAO.setKeyPrefix(**"shiro\_crazy\_"**);  redisSessionDAO.setKeySerializer(**new** StringSerializer());  redisSessionDAO.setValueSerializer(**new** ObjectSerializer());  redisSessionDAO.setRedisManager(redisManager());  **return** redisSessionDAO;  }    */\*\*  \* shiro缓存管理器; 需要注入对应的其它的实体类中： 1、安全管理器：securityManager  \* 可见securityManager是整个shiro的核心；  \*  \** ***@return*** *\*/* @Bean  **public** CacheManager cacheManager() {  ***logger***.info(**"注入Shrio的缓存管理器-->ehCacheManager"** + RedisCacheManager.**class**);  org.crazycake.shiro.RedisCacheManager cacheManager = **new** org.crazycake.shiro.RedisCacheManager();  cacheManager.setRedisManager(redisManager());  *//存入redis的key的过期时间* cacheManager.setExpire(600);  **return** cacheManager;  }    */\*\*  \* kickoutSessionControlFilter 用于控制并发登录人数的   \** ***@return*** *\*/* @Bean  **public** AccessControlFilter kickoutFilter() {  ***logger***.info(**"注入Shrio的一个用户只能在一个地方登录过滤器--> "**+KickoutSessionControlFilter.**class**);  KickoutSessionControlFilter kickout=**new** KickoutSessionControlFilter();  *//是否踢出后来登录的，默认是false；即后者登录的用户踢出前者登录的用户；踢出顺序。* kickout.setKickoutAfter(**false**);  *//被踢出后重定向到的地址；* kickout.setKickoutUrl(**"/html/kickout.html"**);  *//同一个用户最大的会话数，默认1；比如2的意思是同一个用户允许最多同时两个人登录；* kickout.setMaxSession(1);  *//用于根据会话ID，获取会话进行踢出操作的；* kickout.setSessionManager(sessionManager());  *//使用cacheManager获取相应的cache来缓存用户登录的会话；用于保存用户—会话之间的关系的；  //这里我们还是用之前shiro使用的redisManager()实现的cacheManager()缓存管理  //也可以重新另写一个，重新配置缓存时间之类的自定义缓存属性* kickout.setCacheManager(cacheManager());    **return** kickout;  }    @Bean  **public** SessionListener sessionListener() {  **return new** MyShiroSessionListener();  }  */\*\*  \* Shiro生命周期处理器  \*  \** ***@return*** *\*/* @Bean  **public** LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {  **return new** LifecycleBeanPostProcessor();  }   */\*\*  \* 开启Shiro的注解(如@RequiresRoles,@RequiresPermissions),需借助SpringAOP扫描使用Shiro注解的类,并在必要时进行安全逻辑验证  \* 配置以下两个bean(DefaultAdvisorAutoProxyCreator(可选)和AuthorizationAttributeSourceAdvisor)即可实现此功能  \*  \** ***@return*** *\*/* @Bean  @DependsOn({**"lifecycleBeanPostProcessor"**})  **public** DefaultAdvisorAutoProxyCreator advisorAutoProxyCreator() {  DefaultAdvisorAutoProxyCreator advisorAutoProxyCreator = **new** DefaultAdvisorAutoProxyCreator();  advisorAutoProxyCreator.setProxyTargetClass(**true**);  **return** advisorAutoProxyCreator;  } */\*\*  \* AuthorizationAttributeSourceAdvisor，shiro里实现的Advisor类，  \* 内部使用AopAllianceAnnotationsAuthorizingMethodInterceptor来拦截用以下注解的方法。  \*/*  @Bean  **public** AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor() {  AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor = **new** AuthorizationAttributeSourceAdvisor();  authorizationAttributeSourceAdvisor.setSecurityManager(securityManager());  **return** authorizationAttributeSourceAdvisor;  }   */\*\*  \* 添加ShiroDialect 为了在thymeleaf里使用shiro的标签的bean  \*  \** ***@return*** *\*/* @Bean(name = **"shiroDialect"**)  **public** ShiroDialect shiroDialect() {  **return new** ShiroDialect();  } } |

⑥Shiro中Session监听器

会话监听器用于监听会话创建、过期及停止事件，如果只想监听某一个事件，可以继承 SessionListenerAdapter 实现。

|  |
| --- |
| **import** org.apache.shiro.session.Session; **import** org.apache.shiro.session.SessionListener;  **public class** MyShiroSessionListener **implements** SessionListener {   @Override  **public void** onStart(Session session) {  System.***out***.println(**"会话创建："** + session.getId() + **"》》时间："** + session.getLastAccessTime());  *// session.setTimeout(600000);* }   @Override  **public void** onExpiration(Session session) {*// 会话过期时触发* System.***out***.println(**"会话过期："** + session.getId() + **"》》时间："** + **new** Date());  }   @Override  **public void** onStop(Session session) {*// 退出/会话过期时触发* System.***out***.println(**"会话停止："** + session.getId() + **"》》时间："** + session.getLastAccessTime());  } } |

⑥用户只能在一处登录

|  |
| --- |
| **import** java.io.IOException; **import** java.io.PrintWriter; **import** java.io.Serializable; **import** java.lang.reflect.InvocationTargetException; **import** java.util.Deque; **import** java.util.HashMap; **import** java.util.LinkedList; **import** java.util.Map;  **import** javax.servlet.ServletRequest; **import** javax.servlet.ServletResponse; **import** javax.servlet.http.HttpServletRequest;  **import** org.apache.commons.beanutils.BeanUtils; **import** org.apache.shiro.cache.Cache; **import** org.apache.shiro.cache.CacheManager; **import** org.apache.shiro.session.Session; **import** org.apache.shiro.session.mgt.DefaultSessionKey; **import** org.apache.shiro.session.mgt.SessionManager; **import** org.apache.shiro.subject.Subject; **import** org.apache.shiro.web.filter.AccessControlFilter; **import** org.apache.shiro.web.util.WebUtils;  **import** com.alibaba.fastjson.JSON; **import** com.cxp.personalmanage.pojo.UserInfo;  **public class** KickoutSessionControlFilter **extends** AccessControlFilter {   **private final** String **CACHE\_SHIRO** = **"shiro\_cache\_filter\_"**;   **private** String **kickoutUrl**; *// 踢出后到的地址* **private boolean kickoutAfter**; *// 踢出之前登录的/之后登录的用户 默认踢出之前登录的用户* **private int maxSession**; *// 同一个帐号最大会话数 默认1* **private** SessionManager **sessionManager**;  **private** Cache<String, Deque<Serializable>> **cache**;   **public void** setKickoutUrl(String kickoutUrl) {  **this**.**kickoutUrl** = kickoutUrl;  }   **public void** setKickoutAfter(**boolean** kickoutAfter) {  **this**.**kickoutAfter** = kickoutAfter;  }   **public void** setMaxSession(**int** maxSession) {  **this**.**maxSession** = maxSession;  }   **public void** setSessionManager(SessionManager sessionManager) {  **this**.**sessionManager** = sessionManager;  }   *// 设置Cache的key的前缀* **public void** setCacheManager(CacheManager cacheManager) {  **this**.**cache** = cacheManager.getCache(**CACHE\_SHIRO**);  }   */\*\*  \* 是否允许访问，返回true表示允许  \*/* @Override  **protected boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue)  **throws** Exception {  **return false**;  }   */\*\*  \* 表示访问拒绝时是否自己处理，如果返回true表示自己不处理且继续拦截器链执行， 返回false表示自己已经处理了（比如重定向到另一个页面）。  \*/* @Override  **protected boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws** Exception {  Subject subject = getSubject(request, response);  **if** (!subject.isAuthenticated() && !subject.isRemembered()) {  *// 如果没有登录，直接进行之后的流程* **return true**;  }  Session session = subject.getSession();  Object principal = subject.getPrincipal();  UserInfo userInfo = **new** UserInfo();  **try** {  BeanUtils.*copyProperties*(userInfo, principal);  } **catch** (IllegalAccessException | InvocationTargetException e) {  e.printStackTrace();  }  String username = userInfo.getUserName();  Serializable sessionId = session.getId();   *// 读取缓存 没有就存入* Deque<Serializable> deque = **cache**.get(username);   *// 如果此用户没有session队列，也就是还没有登录过，缓存中没有  // 就new一个空队列，不然deque对象为空，会报空指针* **if** (deque == **null**) {  deque = **new** LinkedList<Serializable>();  }   *// 如果队列里没有此sessionId，且用户没有被踢出；放入队列* **if** (!deque.contains(sessionId) && session.getAttribute(**"kickout"**) == **null**) {  *// 将sessionId存入队列* deque.push(sessionId);  *// 将用户的sessionId队列缓存* **cache**.put(username, deque);  }   *// 如果队列里的sessionId数超出最大会话数，开始踢人* **while** (deque.size() > **maxSession**) {  Serializable kickoutSessionId = **null**;  **if** (**kickoutAfter**) { *// 如果踢出后者* kickoutSessionId = deque.getFirst();  kickoutSessionId = deque.removeFirst();  *// 踢出后再更新下缓存队列* **cache**.put(username, deque);  } **else** { *// 否则踢出前者* kickoutSessionId = deque.removeLast();  *// 踢出后再更新下缓存队列* **cache**.put(username, deque);  }  **try** {  *// 获取被踢出的sessionId的session对象* Session kickoutSession = **sessionManager**.getSession(**new** DefaultSessionKey(kickoutSessionId));  **if** (kickoutSession != **null**) {  *// 设置会话的kickout属性表示踢出了* kickoutSession.setAttribute(**"kickout"**, **true**);  }  } **catch** (Exception e) {*// ignore exception* e.printStackTrace();  }  }   *// 如果被踢出了，直接退出，重定向到踢出后的地址* **if** ((Boolean) session.getAttribute(**"kickout"**) != **null** && (Boolean) session.getAttribute(**"kickout"**) == **true**) {  *// 会话被踢出了* **try** {  *// 退出登录* subject.logout();  } **catch** (Exception e) {  }  saveRequest(request);   Map<String, String> resultMap = **new** HashMap<String, String>();  *// 判断是不是Ajax请求* **if** (**"XMLHttpRequest"**.equalsIgnoreCase(((HttpServletRequest) request).getHeader(**"X-Requested-With"**))) {  resultMap.put(**"status"**, **"300"**);  resultMap.put(**"msg"**, **"您已经在其他地方登录，请重新登录！"**);  *// 输出json串* out(response, resultMap);  } **else** {  *// 重定向* WebUtils.*issueRedirect*(request, response, **kickoutUrl**);  }  **return false**;  }  **return true**;  }   **private void** out(ServletResponse hresponse, Map<String, String> resultMap) **throws** IOException {  **try** {  hresponse.setCharacterEncoding(**"UTF-8"**);  PrintWriter out = hresponse.getWriter();  out.println(JSON.*toJSONString*(resultMap));  out.flush();  out.close();  } **catch** (Exception e) {  System.***err***.println(**"KickoutSessionFilter.class 输出JSON异常，可以忽略。"**);  }  } } |

⑦配置Controller

|  |
| --- |
| @Controller **public class** SecurityController {  **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SecurityController.**class**);  @RequestMapping(value = **"/login"**) @ResponseBody **public** String login(UserInfo userInfo, HttpServletRequest request, Map<String, Object> map,String randomcode) {  String username = userInfo.getUserName();  *// 页面输入的验证码* String sessionRandomcode = (String) request.getSession().getAttribute(**"vrifyCode"**);  **if**(sessionRandomcode == **null**) {  **return "{\"msg\":\"验证码已失效,请重新获取!\",\"status\":\"100\"}"**;  }  **if** (randomcode != **null** && !randomcode.equalsIgnoreCase(sessionRandomcode)) {  **return "{\"msg\":\"验证码错误!\",\"status\":\"100\"}"**;  }    System.***out***.println(username);  *// 获取当前的Subject* Subject currentUser = SecurityUtils.*getSubject*();  UsernamePasswordToken token = **new** UsernamePasswordToken(userInfo.getUserName(), userInfo.getPassword());  String msg = **""**;  **try** {  *// 在调用了login方法后,SecurityManager会收到AuthenticationToken,并将其发送给已配置的Realm执行必须的认证检查  // 每个Realm都能在必要时对提交的AuthenticationTokens作出反应  // 所以这一步在调用login(token)方法时,它会走到MyRealm.doGetAuthenticationInfo()方法中,具体验证方式详见此方法* ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证开始"**);  currentUser.login(token);  ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证通过"**);  } **catch** (UnknownAccountException uae) {  ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证未通过,未知账户"**);  msg = **"验证未通过,未知账户"**;  } **catch** (IncorrectCredentialsException ice) {  ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证未通过,错误的凭证"**);  msg = **"验证未通过,错误的凭证"**;  } **catch** (LockedAccountException lae) {  ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证未通过,账户已锁定"**);  msg = **"验证未通过,账户已锁定"**;  } **catch** (ExcessiveAttemptsException eae) {  ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证未通过,错误次数过多"**);  msg = **"验证未通过,错误次数过多,请5分钟后再尝试登录"**;  } **catch** (AuthenticationException ae) {  *// 通过处理Shiro的运行时AuthenticationException就可以控制用户登录失败或密码错误时的情景* ***logger***.info(**"对用户["** + username + **"]进行登录验证..验证未通过,堆栈轨迹如下"**);  ae.printStackTrace();  }  *// 验证是否登录成功* **if** (currentUser.isAuthenticated()) {  ***logger***.info(**"用户["** + username + **"]登录认证通过(这里可以进行一些认证通过后的一些系统参数初始化操作)"**);  **return "{\"msg\":\"../html/index.html\",\"status\":\"200\"}"**;  } **else** {  token.clear();  **return "{\"msg\":\""**+msg+**"\",\"status\":\"100\"}"**;  }  }  @RequestMapping(**"/defaultKaptcha"**) **public void** defaultKaptcha(HttpServletRequest request, HttpServletResponse response) **throws** Exception {  **byte**[] captchaChallengeAsJpeg = **null**;  HttpSession session = **null**;  ByteArrayOutputStream jpegOutputStream = **new** ByteArrayOutputStream();  **try** {  *// 生产验证码字符串并保存到session中* String createText = **defaultKaptcha**.createText();  ***logger***.info(**"验证码为："** + createText);  session = request.getSession();  session.setAttribute(**"vrifyCode"**, createText);  *// 使用生产的验证码字符串返回一个BufferedImage对象并转为byte写入到byte数组中* BufferedImage challenge = **defaultKaptcha**.createImage(createText);  ImageIO.*write*(challenge, **"jpg"**, jpegOutputStream);  } **catch** (IllegalArgumentException e) {  response.sendError(HttpServletResponse.***SC\_NOT\_FOUND***);  **return**;  }  }  }  */\*\*  \* 设置定时删除session中的验证码  \** ***@param session*** *\** ***@param attrName*** *\*/* **private void** removeVrityCode(**final** HttpSession session, **final** String attrName) {  **final** Timer timer = **new** Timer();  timer.schedule(**new** TimerTask() {  @Override  **public void** run() {  *// 删除session中存的验证码* **if**(**null** != session.getAttribute(attrName)) {  session.removeAttribute(attrName);  timer.cancel();  }  }  }, 2 \* 60 \* 1000);  }  } |

⑧实体类

|  |
| --- |
| UserInfo用户信息表 |
| **import** java.io.Serializable; **import** java.util.ArrayList; **import** java.util.List;  **public class** UserInfo **extends** BaseEntityInfo **implements** Serializable{  **private static final long *serialVersionUID*** = -5890268832873153185L;  **private** Integer **id**;  **private** String **userName**;*// 用户名* **private** String **password**;*// 密码* **private** String **salt**;*// 盐* **private** String **realName**;   **private** List<RoleInfo> **roleList** = **new** ArrayList<RoleInfo>();   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUserName() {  **return userName**;  }   **public void** setUserName(String userName) {  **this**.**userName** = userName;  }   **public** String getPassword() {  **return password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   **public** String getSalt() {  **return salt**;  }   **public void** setSalt(String salt) {  **this**.**salt** = salt;  }   **public** String getRealName() {  **return realName**;  }   **public void** setRealName(String realName) {  **this**.**realName** = realName;  }   **public** List<RoleInfo> getRoleList() {  **return roleList**;  }   **public void** setRoleList(List<RoleInfo> roleList) {  **this**.**roleList** = roleList;  }   @Override  **public** String toString() {  **return "UserInfo [id="** + **id** + **", userName="** + **userName** + **", password="** + **password** + **", salt="** + **salt** + **", realName="** + **realName** + **", roleList="** + **roleList** + **", create\_time="** + **create\_time** + **", create\_user="** + **create\_user** + **", update\_time="** + **update\_time** + **", update\_user="** + **update\_user** + **"]"**;  }  } |
| RoleInfo角色信息表 |
| **import** java.io.Serializable;  **public class** RoleInfo **extends** BaseEntityInfo **implements** Serializable{   **private static final long *serialVersionUID*** = 3206208596229553207L;    **private** Integer **id**;  **private** String **roleId**;  **private** String **roleName**;  **private** String **description**;   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getRoleId() {  **return roleId**;  }   **public void** setRoleId(String roleId) {  **this**.**roleId** = roleId;  }   **public** String getRoleName() {  **return roleName**;  }   **public void** setRoleName(String roleName) {  **this**.**roleName** = roleName;  }   **public** String getDescription() {  **return description**;  }   **public void** setDescription(String description) {  **this**.**description** = description;  }    @Override  **public** String toString() {  **return "RoleInfo [id="** + **id** + **", roleId="** + **roleId** + **", roleName="** + **roleName** + **", description="** + **description** + **", create\_time="** + **create\_time** + **", create\_user="** + **create\_user** + **", update\_time="** + **update\_time** + **", update\_user="** + **update\_user** + **"]"**;  }  } |

⑨Mapper层、Service层

|  |
| --- |
| Mapper层 |
| **public interface** UserInfoMapper {  */\*\*  \* 根据用户账号或者id查询唯一结果  \** ***@param userInfo*** *\** ***@return*** *\*/* **public** UserInfo getUserInfo(UserInfo userInfo);   */\*\*  \* 查询用户信息和权限信息  \** ***@param param*** *\** ***@return*** *\*/* **public** List<UserInfo> findUserInfoList(Map<String,Object> param);    */\*\*  \* 批量插入  \** ***@param list*** *\** ***@return*** *\*/* **public int** bathSaveUserInfo(List<UserInfo> list);    */\*\*  \* 单条插入  \** ***@param userInfo*** *\** ***@return*** *\*/* **public int** saveUserInfo(UserInfo userInfo); } |
| **public interface** RoleInfoMapper {   **public** List<RoleInfo> findUserRoleInfoList(String userName); } |
| Service层 |
| **import** java.util.List; **import** java.util.Map;  **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service; **import** org.springframework.transaction.annotation.Isolation; **import** org.springframework.transaction.annotation.Transactional;  **import** com.cxp.personalmanage.mapper.UserInfoMapper; **import** com.cxp.personalmanage.pojo.UserInfo; **import** com.cxp.personalmanage.service.UserInfoService;  @Service(value = **"userInfoService"**) @Transactional(isolation = Isolation.***DEFAULT***) **public class** UserInfoServiceImpl **implements** UserInfoService {   @Autowired  **private** UserInfoMapper **userInfoMapper**;   @Transactional(readOnly = **true**)  @Override  **public** List<UserInfo> findUserInfoList(Map<String, Object> param) {   **return userInfoMapper**.findUserInfoList(param);  }   @Override  **public int** bathSaveUserInfo(List<UserInfo> list) {  **return userInfoMapper**.bathSaveUserInfo(list);  }   @Override  **public int** saveUserInfo(UserInfo userInfo) {  *//* ***TODO Auto-generated method stub* int** result = **userInfoMapper**.saveUserInfo(userInfo);  **return** result;  }   @Override  **public** UserInfo getUserInfo(UserInfo userInfo) {  **return userInfoMapper**.getUserInfo(userInfo);  }  } |
| @Service(value=**"roleInfoService"**) **public class** RoleInfoServiceImpl **implements** RoleInfoService {    @Autowired  **private** RoleInfoMapper **roleInfoMapper**;   @Override  **public** List<RoleInfo> findUserRoleInfoList(String userName) {  **return roleInfoMapper**.findUserRoleInfoList(userName);  }  } |

⑩html页面

|  |
| --- |
| **login.html** |
| <!DOCTYPE **html**> <**html**> <**head**> <**meta charset="UTF-8"** /> <**title**>用户登录</**title**> <**link rel="stylesheet" type="text/css" href="../js/bootstrap/css/bootstrap.min.css"** /> <**style**> **body** {  **margin-left**: **auto**;  **margin-right**: **auto**;  **margin-TOP**: 100**PX**;  **width**: 20**em**; } </**style**> <**script type="text/javascript" src="../js/common/jquery-1.9.1.min.js"**></**script**> <**script type="text/javascript" src="../js/bootstrap/js/bootstrap.min.js"**></**script**> <**script type="text/javascript"**>  **$**(**function**(){    **$**(**"#submit"**).click(**function**(){  **var** username =**$**(**"#userName"**).val();  **var** password =**$**(**"#password"**).val();  **var** randomcode = **$**(**"#randomcode"**).val();  **if**( !username || !password || !randomcode ){  *alert*(**"请求输入用户名或者密码、验证码!"**);  **return false**;  }    **var** data=**$**.param({**date**:**new *Date***()})+**'&'** + **$**(**"#formabc"**).serialize();  **$**.**ajax**({  **url**:**'../login'**,  **data**: data,  **dataType**:**'json'**,  **contentType**:**"application/x-www-form-urlencoded"**,  success:**function**(data){  ***console***.log(data.msg);  **if**(data.status == **'200'**){  ***window***.location.href = data.msg;  }**else**{  **$**(**"#msgH3"**).text(data.msg);  }  }  });  });  });  </**script**> </**head**> <**body style="background-color**: ;**"**>    <**form action="" method="POST" id="formabc"**>  *<!--下面是用户名输入框-->* <**div class="input-group"**>  <**span class="input-group-addon" id="basic-addon1"**>用户名</**span**> <**input  id="userName" name="userName" type="text" class="form-control"  placeholder="用户名" aria-describedby="basic-addon1"** />  </**div**>  <**br** />  *<!--下面是密码输入框-->* <**div class="input-group"**>  <**span class="input-group-addon" id="basic-addon2"**>密**&nbsp;&nbsp;&nbsp;&nbsp;**码</**span**>  <**input id="password" name="password" type="text" class="form-control"  placeholder="密码" aria-describedby="basic-addon1"** />  </**div**>  <**br** />  <**div class="input-group"**>  <**span class="input-group-addon" id="basic-addon2"**>验证码</**span**>   <**input id="randomcode" name="randomcode" type="text" class="form-control" style="width**:98%;**"  placeholder="验证码" aria-describedby="basic-addon1"** />  *<!-- <img alt="这是图片" src="/img/001.png"/> -->* <**span class="input-group-btn" style="height**:100%;**"** >  <**img alt="验证码" onclick="this**.**src**=**'/defaultKaptcha?d='**+**new *Date***()\*1**" src="/defaultKaptcha"** />  </**span**>  </**div**>    *<!--下面是记住我输入框-->* <**div class="checkbox"**>  <**label**> <**input id="remembeMe" name="rememberMe" type="checkbox"** />记住我  </**label**>  </**div**>  *<!--下面是登陆按钮,包括颜色控制-->* <**button type="button" id = "submit" style="width**: 280**px**;**" class="btn btn-info"**>登录</**button**>  </**form**>  <**h4 id="msgH3" style="color**:**red**;**font-size**:14**px**;**" class="text-center"**></**h4**> </**body**>  </**html**> |
| index.html |
| <!DOCTYPE **html**> <**html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"  xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3"  xmlns:shiro="http://www.pollix.at/thymeleaf/shiro"**> <**head**>  <**meta charset="UTF-8"** />  <**title**>首页</**title**>  </**head**> <**body**> Hello, <**shiro:principal**/>, how are you today? <**span style="width**: 200**px**;**"**></**span**><**a href="/logout"**>退出</**a**> <**shiro:hasRole name="admin"**>  <**br** />I'm fine, 我拥有管理员角色--<**span th:text="${username}"**></**span**> </**shiro:hasRole**> </**body**> </**html**> |

**4、Shiro中使用SessionManager**

Shiro的session都是存在缓存中的，所有会有一个sessionDAO的类来做CRUD操作，即DAO（Data Access Object）模式实现。SessionDAO是Shiro提供的一个数据交互层的interface接口，其作用是可以将Session写入到数据库

中，然后可以对Session进行增删改查操作。

Shiro的SessionDAO实现:

|  |
| --- |
|  |

AbstractSessionDAO提供了SessionDAO的基础实现，如生成会话ID等；

CachingSessionDAO提供了对开发者透明的会话缓存的功能，只需要设置相应的CacheManager即可；

MemorySessionDAO直接在内存中进行会话维护；

EnterpriseCacheSessionDAO提供了缓存功能的会话维护，默认情况下使用MapCache实现，内部使用ConcurrentHashMap保存缓存的会话。

|  |
| --- |
| **public** **interface** SessionDAO {  /\*如DefaultSessionManager在创建完session后会调用该方法；  如保存到关系数据库/文件系统/NoSQL数据库；即可以实现会话的持久化；  返回会话ID；主要此处返回的ID.equals(session.getId())；  \*/  Serializable create(Session session);    //根据会话ID获取会话  Session readSession(Serializable sessionId) **throws** UnknownSessionException;    //更新会话；如更新会话最后访问时间/停止会话/设置超时时间/设置移除属性等会调用  **void** update(Session session) **throws** UnknownSessionException;    //删除会话；当会话过期/会话停止（如用户退出时）会调用  **void** delete(Session session);    //获取当前所有活跃用户，如果用户量多此方法影响性能  Collection<Session> getActiveSessions();  } |

如果自定义实现SessionDAO，继承CachingSessionDAO即可,doCreate/doUpdate/doDelete/doReadSession分别代表创建/修改/删除/读取会话。

①自定义SessionDAO

|  |
| --- |
| **import** java.io.Serializable;  **import** java.util.Collection;  **import** java.util.HashSet;  **import** java.util.Set;  **import** org.apache.shiro.session.Session;  **import** org.apache.shiro.session.UnknownSessionException;  **import** org.apache.shiro.session.mgt.eis.AbstractSessionDAO;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.util.CollectionUtils;  **import** org.springframework.util.SerializationUtils;  **import** com.cxp.personalmanage.utils.JedisUtil;  **public** **class** RedisSessionDao **extends** AbstractSessionDAO {  **private** **final** String shiro\_session\_prefix = "shiro\_";    @Autowired  **private** JedisUtil jedisUtil;  **private** **byte**[] getKey(String key) {  **return** (shiro\_session\_prefix + key).getBytes();  }  **private** **void** saveSession(Session session) {  **if** (session != **null** && session.getId() != **null**) {  **byte**[] key = getKey(session.getId().toString());  **byte**[] value = SerializationUtils.*serialize*(session);  jedisUtil.set(key, value);  jedisUtil.expire(key, 600);  }  }  @Override  **public** **void** update(Session session) **throws** UnknownSessionException {  saveSession(session);  }  @Override  **public** **void** delete(Session session) {  **if** (session == **null** || session.getId() == **null**) {  **return**;  }  **byte**[] key = getKey(session.getId().toString());  jedisUtil.del(key);  }  /\*\*  \* 获取存活的session  \*/  @Override  **public** Collection<Session> getActiveSessions() {  Set<**byte**[]> keys = jedisUtil.keys(shiro\_session\_prefix);  Set<Session> sessions = **new** HashSet<>();  **if** (CollectionUtils.*isEmpty*(keys)) {  **return** sessions;  }  **for** (**byte**[] key : keys) {  Session session = (Session) SerializationUtils.*deserialize*(jedisUtil.get(key));  sessions.add(session);  }  **return** sessions;  }  @Override  **protected** Serializable doCreate(Session session) {  Serializable sessionId = generateSessionId(session);  assignSessionId(session, sessionId);  saveSession(session);  **return** sessionId;  }  @Override  **protected** Session doReadSession(Serializable sessionId) {  System.***out***.println("doReadSession session");  **if** (sessionId == **null**) {  **return** **null**;  }  **byte**[] key = getKey(sessionId.toString());  **byte**[] value = jedisUtil.get(key);  **return** (Session) SerializationUtils.*deserialize*(value);  }  } |

①自定义SessionManager

|  |
| --- |
| **import** java.io.Serializable;  **import** javax.servlet.ServletRequest;  **import** org.apache.shiro.session.Session;  **import** org.apache.shiro.session.UnknownSessionException;  **import** org.apache.shiro.session.mgt.SessionKey;  **import** org.apache.shiro.web.session.mgt.DefaultWebSessionManager;  **import** org.apache.shiro.web.session.mgt.WebSessionKey;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  /\*\*  \* 自定义SessionManager  \* **@author** Administrator  \*  \*/  **public** **class** CustomSessionManager **extends** DefaultWebSessionManager {    **private** Logger logger = LoggerFactory.*getLogger*(CustomSessionManager.**class**);    @Override  **protected** Session retrieveSession(SessionKey sessionKey) **throws** UnknownSessionException {  Serializable sessionId = getSessionId(sessionKey);  ServletRequest request = **null**;  **if** (sessionKey **instanceof** WebSessionKey) {  request = ((WebSessionKey) sessionKey).getServletRequest();  }  **if** (request != **null** && sessionId != **null**) {  Session session = (Session) request.getAttribute(sessionId.toString());  **if** (session != **null**) {  **return** session;  }  }  Session session = **super**.retrieveSession(sessionKey);  **if** (request != **null** && sessionId != **null**) {  request.setAttribute(sessionId.toString(), session);  }  **return** session;  }  } |

①注入到Spring容器中

|  |
| --- |
| @Bean  **public** SessionManager sessionManager() {  // DefaultWebSessionManager sessionManager = new DefaultWebSessionManager();  // <!-- 会话超时时间，单位：毫秒 -->  // session.setGlobalSessionTimeout(20000);  // <!-- 删除失效的session -->  CustomSessionManager sessionManager = **new** CustomSessionManager();  sessionManager.setDeleteInvalidSessions(**true**);  //是否定期检查Session的有效性  sessionManager.setSessionValidationSchedulerEnabled(**true**);  sessionManager.setCacheManager(cacheManager());  sessionManager.setSessionDAO(sessionDAO());  **return** sessionManager;  }  @Bean  **public** SessionDAO sessionDAO() {  RedisSessionDao redisSessionDAO = **new** RedisSessionDao();  //设置生成sessionId生成策略:  // RandomSessionIdGenerator() 和 JavaUuidSessionIdGenerator()  redisSessionDAO.setSessionIdGenerator(**new** RandomSessionIdGenerator());  **return** redisSessionDAO;  } |

⑷spring boot shiro redis整合principals.getPrimaryPrincipal()强制转换类型错误

spring boot项目使用ShiroUser shiroUser=(ShiroUser)principals.getPrimaryPrincipal() 报错

错误：java.lang.ClassCastException:com.zyc.springboot.shiro.ShiroUser cannot be cast to com.zyc.springboot.shiro.ShiroUser

解决办法暂时发现3种：

1：有可能是spring boot 热部署造成的，去掉spring boot 热部署试一试。

2：使用org.apache.commons.beanutils.BeanUtils类进去属性转换，其实就是相当于把数据反射出来:

|  |
| --- |
| // 获取当前的Subject  Subject currentUser = SecurityUtils.*getSubject*();  Object principal = currentUser.getPrincipal();  UserInfo userInfo = **new** UserInfo();  **try** {  BeanUtils.*copyProperties*(userInfo, principal);  } **catch** (IllegalAccessException | InvocationTargetException e) {  e.printStackTrace();  } |

3: 把当前用户的登录名信息保存进去，而不是保存一个具体的对象。

|  |
| --- |
| SimpleAuthenticationInfo simpleAuthenticationInfo = new SimpleAuthenticationInfo(  userName, user.getPassword(), this.getName());替换  SimpleAuthenticationInfo simpleAuthenticationInfo = new SimpleAuthenticationInfo(  shiroUser, user.getPassword(), this.getName()); |

⑸缓存管理器SimpleByteSource序列化问题

在使用Redis作为Shiro的缓存管理器的时候，在将SimpleAuthenticationInfo信息进行序列化的时候报异常：java.io.NotSerializableException: org.apache.shiro.util.SimpleByteSource

原因是：SimpleByteSource没有实现Serializable接口

正确的解决办法：自定义ByteSource的实现类，模仿SimpleByteSource的方法。

|  |
| --- |
| **import** java.io.File;  **import** java.io.InputStream;  **import** java.io.Serializable;  **import** java.util.Arrays;  **import** org.apache.shiro.codec.Base64;  **import** org.apache.shiro.codec.CodecSupport;  **import** org.apache.shiro.codec.Hex;  **import** org.apache.shiro.util.ByteSource;  /\*\*  \* 解决：  \* shiro 使用缓存时出现：java.io.NotSerializableException: org.apache.shiro.util.SimpleByteSource  \* 序列化后，无法反序列化的问题  \* **@author** Administrator  \*/  **public** **class** MySimpleByteSource **implements** ByteSource, Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 5175082362119580768L;  **private** **byte**[] bytes;  **private** String cachedHex;  **private** String cachedBase64;  **public** MySimpleByteSource() {  }  **public** MySimpleByteSource(**byte**[] bytes) {  **this**.bytes = bytes;  }  **public** MySimpleByteSource(**char**[] chars) {  **this**.bytes = CodecSupport.*toBytes*(chars);  }  **public** MySimpleByteSource(String string) {  **this**.bytes = CodecSupport.*toBytes*(string);  }  **public** MySimpleByteSource(ByteSource source) {  **this**.bytes = source.getBytes();  }  **public** MySimpleByteSource(File file) {  **this**.bytes = (**new** MySimpleByteSource.BytesHelper()).getBytes(file);  }  **public** MySimpleByteSource(InputStream stream) {  **this**.bytes = (**new** MySimpleByteSource.BytesHelper()).getBytes(stream);  }  **public** **static** **boolean** isCompatible(Object o) {  **return** o **instanceof** **byte**[] || o **instanceof** **char**[] || o **instanceof** String || o **instanceof** ByteSource  || o **instanceof** File || o **instanceof** InputStream;  }  **public** **void** setBytes(**byte**[] bytes) {  **this**.bytes = bytes;  }  @Override  **public** **byte**[] getBytes() {  **return** **this**.bytes;  }  @Override  **public** String toHex() {  **if** (**this**.cachedHex == **null**) {  **this**.cachedHex = Hex.*encodeToString*(**this**.getBytes());  }  **return** **this**.cachedHex;  }  @Override  **public** String toBase64() {  **if** (**this**.cachedBase64 == **null**) {  **this**.cachedBase64 = Base64.*encodeToString*(**this**.getBytes());  }  **return** **this**.cachedBase64;  }  @Override  **public** **boolean** isEmpty() {  **return** **this**.bytes == **null** || **this**.bytes.length == 0;  }  @Override  **public** String toString() {  **return** **this**.toBase64();  }  @Override  **public** **int** hashCode() {  **return** **this**.bytes != **null** && **this**.bytes.length != 0 ? Arrays.*hashCode*(**this**.bytes) : 0;  }  @Override  **public** **boolean** equals(Object o) {  **if** (o == **this**) {  **return** **true**;  } **else** **if** (o **instanceof** ByteSource) {  ByteSource bs = (ByteSource) o;  **return** Arrays.*equals*(**this**.getBytes(), bs.getBytes());  } **else** {  **return** **false**;  }  }  **private** **static** **final** **class** BytesHelper **extends** CodecSupport {  **private** BytesHelper() {  }  **public** **byte**[] getBytes(File file) {  **return** **this**.toBytes(file);  }  **public** **byte**[] getBytes(InputStream stream) {  **return** **this**.toBytes(stream);  }  }  } |

**5、Shiro中使用缓存管理器**

使用第三方jar包:

|  |
| --- |
| <dependency>  <groupId>org.crazycake</groupId>  <artifactId>shiro-redis</artifactId>  <version>2.8.24</version>  </dependency> |
|  |

**6、Shiro中管理在线用户（踢出用户）**

应用场景

1、我们经常会有用到，当A 用户在北京登录 ，然后A用户在天津再登录 ，要踢出北京登录的状态。如果用户在北京重新登录，那么又要踢出天津的用户，这样反复。又或是需要限制同一用户的同时在线数量，超出限制后，踢出最先登录的或是踢出最后登录的。

2、第一个场景踢出用户是由用户触发的，有时候需要手动将某个在线用户踢出，也就是对当前在线用户的列表进行管理。

spring security就直接提供了相应的功能；Shiro的话没有提供默认实现，不过可以很容易的在Shiro中加入这个功能。那就是使用shiro强大的自定义访问控制拦截器：AccessControlFilter，集成这个接口后要实现下面这三个方法。

|  |
| --- |
| **abstract** **boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception;  **boolean** onAccessDenied(ServletRequest request, ServletResponse response, Object mappedValue) **throws** Exception;  **abstract** **boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws** Exception; |

isAccessAllowed：表示是否允许访问；mappedValue就是[urls]配置中拦截器参数部分，如果允许访问返回true，否则false；

onAccessDenied：表示当访问拒绝时是否已经处理了；如果返回true表示需要继续处理；如果返回false表示该拦截器实例已经处理了，将直接返回即可。

onPreHandle：会自动调用这两个方法决定是否继续处理；

另外AccessControlFilter还提供了如下方法用于处理如登录成功后/重定向到上一个请求：

|  |
| --- |
| **void** setLoginUrl(String loginUrl) //身份验证时使用，默认/login.jsp  String getLoginUrl()  Subject getSubject(ServletRequest request, ServletResponse response) //获取Subject实例  **boolean** isLoginRequest(ServletRequest request, ServletResponse response)//当前请求是否是登录请求  **void** saveRequestAndRedirectToLogin(ServletRequest request, ServletResponse response) **throws** IOException //将当前请求保存起来并重定向到登录页面  **void** saveRequest(ServletRequest request) //将请求保存起来，如登录成功后再重定向回该请求  **void** redirectToLogin(ServletRequest request, ServletResponse response) //重定向到登录页面 |

比如基于表单的身份验证就需要使用这些功能。

到此基本的拦截器就完事了，如果我们想进行访问的控制就可以继承AccessControlFilter；如果我们要添加一些通用数据我们可以直接继承PathMatchingFilter。

|  |
| --- |
| **import** java.io.IOException; **import** java.io.PrintWriter; **import** java.io.Serializable; **import** java.lang.reflect.InvocationTargetException; **import** java.util.Deque; **import** java.util.HashMap; **import** java.util.LinkedList; **import** java.util.Map;  **import** javax.servlet.ServletRequest; **import** javax.servlet.ServletResponse; **import** javax.servlet.http.HttpServletRequest;  **import** org.apache.commons.beanutils.BeanUtils; **import** org.apache.shiro.cache.Cache; **import** org.apache.shiro.cache.CacheManager; **import** org.apache.shiro.session.Session; **import** org.apache.shiro.session.mgt.DefaultSessionKey; **import** org.apache.shiro.session.mgt.SessionManager; **import** org.apache.shiro.subject.Subject; **import** org.apache.shiro.web.filter.AccessControlFilter; **import** org.apache.shiro.web.util.WebUtils;  **import** com.alibaba.fastjson.JSON; **import** com.cxp.personalmanage.pojo.UserInfo;  **public class** KickoutSessionControlFilter **extends** AccessControlFilter {   **private final** String **CACHE\_SHIRO** = **"shiro\_cache\_filter\_"**;   **private** String **kickoutUrl**; *// 踢出后到的地址* **private boolean kickoutAfter**; *// 踢出之前登录的/之后登录的用户 默认踢出之前登录的用户* **private int maxSession**; *// 同一个帐号最大会话数 默认1* **private** SessionManager **sessionManager**;  **private** Cache<String, Deque<Serializable>> **cache**;   **public void** setKickoutUrl(String kickoutUrl) {  **this**.**kickoutUrl** = kickoutUrl;  }   **public void** setKickoutAfter(**boolean** kickoutAfter) {  **this**.**kickoutAfter** = kickoutAfter;  }   **public void** setMaxSession(**int** maxSession) {  **this**.**maxSession** = maxSession;  }   **public void** setSessionManager(SessionManager sessionManager) {  **this**.**sessionManager** = sessionManager;  }   *// 设置Cache的key的前缀* **public void** setCacheManager(CacheManager cacheManager) {  **this**.**cache** = cacheManager.getCache(**CACHE\_SHIRO**);  }   */\*\*  \* 是否允许访问，返回true表示允许  \*/* @Override  **protected boolean** isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue)  **throws** Exception {  **return false**;  }   */\*\*  \* 表示访问拒绝时是否自己处理，如果返回true表示自己不处理且继续拦截器链执行， 返回false表示自己已经处理了（比如重定向到另一个页面）。  \*/* @Override  **protected boolean** onAccessDenied(ServletRequest request, ServletResponse response) **throws** Exception {  Subject subject = getSubject(request, response);  **if** (!subject.isAuthenticated() && !subject.isRemembered()) {  *// 如果没有登录，直接进行之后的流程* **return true**;  }  Session session = subject.getSession();  Object principal = subject.getPrincipal();  UserInfo userInfo = **new** UserInfo();  **try** {  BeanUtils.*copyProperties*(userInfo, principal);  } **catch** (IllegalAccessException | InvocationTargetException e) {  e.printStackTrace();  }  String username = userInfo.getUserName();  Serializable sessionId = session.getId();   *// 读取缓存 没有就存入* Deque<Serializable> deque = **cache**.get(username);   *// 如果此用户没有session队列，也就是还没有登录过，缓存中没有  // 就new一个空队列，不然deque对象为空，会报空指针* **if** (deque == **null**) {  deque = **new** LinkedList<Serializable>();  }   *// 如果队列里没有此sessionId，且用户没有被踢出；放入队列* **if** (!deque.contains(sessionId) && session.getAttribute(**"kickout"**) == **null**) {  *// 将sessionId存入队列* deque.push(sessionId);  *// 将用户的sessionId队列缓存* **cache**.put(username, deque);  }   *// 如果队列里的sessionId数超出最大会话数，开始踢人* **while** (deque.size() > **maxSession**) {  Serializable kickoutSessionId = **null**;  **if** (**kickoutAfter**) { *// 如果踢出后者* kickoutSessionId = deque.getFirst();  kickoutSessionId = deque.removeFirst();  *// 踢出后再更新下缓存队列* **cache**.put(username, deque);  } **else** { *// 否则踢出前者* kickoutSessionId = deque.removeLast();  *// 踢出后再更新下缓存队列* **cache**.put(username, deque);  }  **try** {  *// 获取被踢出的sessionId的session对象* Session kickoutSession = **sessionManager**.getSession(**new** DefaultSessionKey(kickoutSessionId));  **if** (kickoutSession != **null**) {  *// 设置会话的kickout属性表示踢出了* kickoutSession.setAttribute(**"kickout"**, **true**);  }  } **catch** (Exception e) {*// ignore exception* e.printStackTrace();  }  }   *// 如果被踢出了，直接退出，重定向到踢出后的地址* **if** ((Boolean) session.getAttribute(**"kickout"**) != **null** && (Boolean) session.getAttribute(**"kickout"**) == **true**) {  *// 会话被踢出了* **try** {  *// 退出登录* subject.logout();  } **catch** (Exception e) {  }  saveRequest(request);   Map<String, String> resultMap = **new** HashMap<String, String>();  *// 判断是不是Ajax请求* **if** (**"XMLHttpRequest"**.equalsIgnoreCase(((HttpServletRequest) request).getHeader(**"X-Requested-With"**))) {  resultMap.put(**"status"**, **"300"**);  resultMap.put(**"msg"**, **"您已经在其他地方登录，请重新登录！"**);  *// 输出json串* out(response, resultMap);  } **else** {  *// 重定向* WebUtils.*issueRedirect*(request, response, **kickoutUrl**);  }  **return false**;  }  **return true**;  }   **private void** out(ServletResponse hresponse, Map<String, String> resultMap) **throws** IOException {  **try** {  hresponse.setCharacterEncoding(**"UTF-8"**);  PrintWriter out = hresponse.getWriter();  out.println(JSON.*toJSONString*(resultMap));  out.flush();  out.close();  } **catch** (Exception e) {  System.***err***.println(**"KickoutSessionFilter.class 输出JSON异常，可以忽略。"**);  }  } } |

第十篇 SpringBoot定时任务(Spring task)

**1.定时任务的几种实现方式**

1>Java自带的java.util.Timer类，这个类允许你调度一个java.util.TimerTask任务。使用这种方式可以让你的程序按时某一个频度执行，但不能在指定时间运行。一般用的比较少。

2>使用Quartz，这是一个功能比较强大的调度器，可以让你的程序在指定时间执行，也可以按照某一个频度执行，配置起来稍显复杂。

3>Spring3.0以后自带的task，可以将它看成一个轻量级的Quartz，而且使用肯定比Quartz简单许多。

|  |  |  |
| --- | --- | --- |
| 字段 | 允许值 | 允许的特殊字符 |
| 秒 | 0-59 | , - \* / |
| 分 | 0-59 | , - \* / |
| 小时 | 0-23 | , - \* / |
| 日期 | 1-31 | , - \* ? / L W C |
| 月份 | 1-12 或者 JAN-DEC | , - \* / |
| 星期 | 1-7 或者 SUN-SAT | , - \* ? / L C # |
| 年（可选） | 留空 |  |
| - 区间  \* 通配符  ? 你不想设置哪个字段 | | |
| cronExpression配置说明和cron例子 | | |
| CRON表达式  “0 0 12 \* \* ?” 每天中午12点触发  “0 15 10 ? \* \*” 每天早上10：15分触发  “0 15 10 \* \* ?” 每天早上10：15分触发  “0 15 10 \* \* ? \*” 每天早上10：15分触发  “0 15 10 \* \* ? 2005” 2005年的每天早上10:15触发  “0 \* 14 \* \* ?” 每天从下午2点开始到2点59分每分钟一次触发  “0 0/5 14 \* \* ?” 每天从下午2点开始到2:55分结束每5分钟一次触发  “0 0/5 14,18 \* \* ?” 每天的下午2点至2：55分 和 6点55分两个时间段内每5分钟一次触发  “0 0-5 14 \* \* ?” 每天14点至14：05分每分钟一次触发  “0 10,44 14 ? 3 WED” 三月的每周三的14点10分和14：44触发  “0 15 10 ? \* MON-FRI ” 每个周一、周二、周三、周四、周五的10:15触发 | | |

①例一

|  |
| --- |
| @Configuration  @EnableScheduling  **public** **class** MyTask {  /\*\*  \* cron：定时任务表达式 指定：秒、分钟、小时、日期、月份、星期、年（可选）;  \* \*表示任意  \*  \* spring task 在计算时间的时候，是根据当前服务器的系统时间进行计算。  \* 如果是每10秒执行一次的话，那么它是从系统时间的0，10，20秒进行计算的  \* 如何是每1分钟执行一次的话，那么它是从系统时间的1分钟，2分钟进行计算的  \*  \*/  @Scheduled(cron = "0/10 \* \* \* \* \* ")  **public** **void** test1() {  System.***out***.println("每10秒执行一次,"+**new** Date());  }  @Scheduled(cron = "0 0/1 \* \* \* \* ")  **public** **void** test2() {  System.***out***.println("每1分钟执行一次,"+**new** Date());  }  } |

②动态修改cron表达式参数

|  |
| --- |
| /\*\*  \* 1.新建一个task class 2.在class上添加注解 **@EnableScheduling**  \* 3.让我们的class实现接口SchedudingConfigurer 4.实现SchedudingConfigurer中的方法  \*  \* **@author** ASUS  \*  \*/  @RestController  @EnableScheduling  **public** **class** TaskCronChange **implements** SchedulingConfigurer {  // 每10秒执行一次  **private** String expression = "0/10 \* \* \* \* \*";  @RequestMapping("/changeExpression")  **public** String changeExpression() {  expression = "0/20 \* \* \* \* \*";  **return** "changeExpression";  }  @Override  **public** **void** configureTasks(ScheduledTaskRegistrar taskRegistrar) {  Runnable task = **new** Runnable() {  @Override  **public** **void** run() {  System.***out***.println("configureTasks" + **new** Date());  }  };  Trigger trigger = **new** Trigger() {  @Override  **public** Date nextExecutionTime(TriggerContext triggerContext) {  CronTrigger cronTrigger = **new** CronTrigger(expression);  **return** cronTrigger.nextExecutionTime(triggerContext);  }  };  taskRegistrar.addTriggerTask(task, trigger);  }  } |

第十一篇 基于SpringBoot的天气预报服务

|  |
| --- |
| 通过城市名字获得天气数据 ：http://wthrcdn.etouch.cn/weather\_mini?city=深圳通过城市id获得天气数据：http://wthrcdn.etouch.cn/weather\_mini?citykey=101280601,城市ID列表。每个城市都有一个唯一的ID作为标识。见 http://cj.weather.com.cn/support/Detail.aspx?id=51837fba1b35fe0f8411b6df 或者 <http://mobile.weather.com.cn/js/citylist.xml。>  "city": 城市名称  "aqi": 空气指数,  "wendu": 实时温度  "date": 日期，包含未来5天  "high":最高温度  "low": 最低温度  "fengli": 风力  "fengxiang": 风向  "type": 天气类型  以上数据，是我们需要的天气数据的核心数据，但是，同时也要关注下面两个字段：  "status": 接口调用的返回状态，返回值“1000”,意味着数据是接口正常  "desc": 接口状态的描述，“OK”代表接口正常  重点关注返回值不是“1000”的情况,说明，这个接口调用异常了。 |

①在pom.xml文件中添加依赖

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  <version>4.5.3</version>  </dependency> |

②创建实体类

|  |
| --- |
| Weather 天气信息类 |
| /\*\*  \* 天气信息类  \* **@author** ASUS  \*/  **public** **class** Weather **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** String city;  **private** String aqi;  **private** String wendu;  **private** String ganmao;  **private** Yesterday yesterday;  **private** List<Forecast> forecast;  **public** String getCity() {  **return** city;  }  **public** **void** setCity(String city) {  **this**.city = city;  }  **public** String getAqi() {  **return** aqi;  }  **public** **void** setAqi(String aqi) {  **this**.aqi = aqi;  }  **public** String getWendu() {  **return** wendu;  }  **public** **void** setWendu(String wendu) {  **this**.wendu = wendu;  }  **public** String getGanmao() {  **return** ganmao;  }  **public** **void** setGanmao(String ganmao) {  **this**.ganmao = ganmao;  }  **public** Yesterday getYesterday() {  **return** yesterday;  }  **public** **void** setYesterday(Yesterday yesterday) {  **this**.yesterday = yesterday;  }  **public** List<Forecast> getForecast() {  **return** forecast;  }  **public** **void** setForecast(List<Forecast> forecast) {  **this**.forecast = forecast;  }  } |
|  |
| //昨天天气  **public** **class** Yesterday **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **private** String date;  **private** String high;  **private** String fx;  **private** String low;  **private** String fl;  **private** String type;  **public** String getDate() {  **return** date;  }  **public** **void** setDate(String date) {  **this**.date = date;  }  **public** String getHigh() {  **return** high;  }  **public** **void** setHigh(String high) {  **this**.high = high;  }  **public** String getFx() {  **return** fx;  }  **public** **void** setFx(String fx) {  **this**.fx = fx;  }  **public** String getLow() {  **return** low;  }  **public** **void** setLow(String low) {  **this**.low = low;  }  **public** String getFl() {  **return** fl;  }  **public** **void** setFl(String fl) {  **this**.fl = fl;  }  **public** String getType() {  **return** type;  }  **public** **void** setType(String type) {  **this**.type = type;  }  } |
|  |
| //未来天气  **public** **class** Forecast **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** String date;  **private** String high;  **private** String fengxiang;  **private** String low;  **private** String fengli;  **private** String type;  **public** String getDate() {  **return** date;  }  **public** **void** setDate(String date) {  **this**.date = date;  }  **public** String getHigh() {  **return** high;  }  **public** **void** setHigh(String high) {  **this**.high = high;  }  **public** String getFengxiang() {  **return** fengxiang;  }  **public** **void** setFengxiang(String fengxiang) {  **this**.fengxiang = fengxiang;  }  **public** String getLow() {  **return** low;  }  **public** **void** setLow(String low) {  **this**.low = low;  }  **public** String getFengli() {  **return** fengli;  }  **public** **void** setFengli(String fengli) {  **this**.fengli = fengli;  }  **public** String getType() {  **return** type;  }  **public** **void** setType(String type) {  **this**.type = type;  }  } |
|  |
| /\*\*  \* 整个消息的返回对象  \* **@author** ASUS  \*  \*/  **public** **class** WeatherResponse **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** Weather data; // 消息数据  **private** String status; // 消息状态  **private** String desc; // 消息描述  **public** Weather getData() {  **return** data;  }  **public** **void** setData(Weather data) {  **this**.data = data;  }  **public** String getStatus() {  **return** status;  }  **public** **void** setStatus(String status) {  **this**.status = status;  }  **public** String getDesc() {  **return** desc;  }  **public** **void** setDesc(String desc) {  **this**.desc = desc;  }  } |

③service层服务接口及实现

|  |
| --- |
| **public** **interface** WeatherDataService {  /\*\*  \* 根据城市ID查询天气数据  \*  \* **@param** cityId  \* **@return**  \*/  WeatherResponse getDataByCityId(String cityId);  /\*\*  \* 根据城市名称查询天气数据  \*  \* **@param** cityId  \* **@return**  \*/  WeatherResponse getDataByCityName(String cityName);  } |
|  |
| @Service  **public** **class** WeatherDataServiceImpl **implements** WeatherDataService {  @Autowired  **private** RestTemplate restTemplate;  **private** **final** String WEATHER\_API = "http://wthrcdn.etouch.cn/weather\_mini";  @Override  **public** WeatherResponse getDataByCityId(String cityId) {  String uri = WEATHER\_API + "?citykey=" + cityId;  **return** **this**.doGetWeatherData(uri);  }  @Override  **public** WeatherResponse getDataByCityName(String cityName) {  String uri = WEATHER\_API + "?city=" + cityName;  **return** **this**.doGetWeatherData(uri);  }  **private** WeatherResponse doGetWeatherData(String uri) {  ResponseEntity<String> response = restTemplate.getForEntity(uri, String.**class**);  String strBody = **null**;  **if** (response.getStatusCodeValue() == 200) {  strBody = response.getBody();  }  ObjectMapper mapper = **new** ObjectMapper();  WeatherResponse weather = **null**;  **try** {  weather = mapper.readValue(strBody, WeatherResponse.**class**);  } **catch** (IOException e) {  e.printStackTrace();  }  **return** weather;  }  } |

④控制层

|  |
| --- |
| @RestController  @RequestMapping("/weather")  **public** **class** WeatherController {  @Autowired  **private** WeatherDataService weatherDataService;  @GetMapping("/cityId/{cityId}")  **public** WeatherResponse getReportByCityId(@PathVariable("cityId") String cityId) {  **return** weatherDataService.getDataByCityId(cityId);  }  @GetMapping("/cityName/{cityName}")  **public** WeatherResponse getReportByCityName(@PathVariable("cityName") String cityName) {  **return** weatherDataService.getDataByCityName(cityName);  }  } |

第十二篇 使用Druid连接池和监控配置

**druid的可选配置项如下：**

|  |  |  |
| --- | --- | --- |
| **配置** | **缺省值** | **说明** |
| name |  | 配置这个属性的意义在于，如果存在多个数据源，监控的时候可以通过名字来区分开来。如果没有配置，将会生成一个名字，格式是："DataSource-" + System.identityHashCode(this) |
| jdbcUrl |  | 连接数据库的url，不同数据库不一样。例如：  mysql : jdbc:mysql://10.20.153.104:3306/druid2  oracle :jdbc:oracle:thin:@10.20.149.85:1521:ocnauto |
| username |  | 连接数据库的用户名 |
| password |  | 连接数据库的密码。如果你不希望密码直接写在配置文件中，可以使用ConfigFilter。 |
| driverClassName | 根据url自动识别 | 这一项可配可不配，如果不配置druid会根据url自动识别dbType，然后选择相应的driverClassName |
| initialSize | 0 | 初始化时建立物理连接的个数。初始化发生在显示调用init方法，或者第一次getConnection时 |
| maxActive | 8 | 最大连接池数量 |
| maxIdle | 8 | 已经不再使用，配置了也没效果 |
| minIdle |  | 最小连接池数量 |
| maxWait |  | 获取连接时最大等待时间，单位毫秒。配置了maxWait之后，缺省启用公平锁，并发效率会有所下降，  如果需要可以通过配置useUnfairLock属性为true使用非公平锁。 |
| poolPreparedStatements | false | 是否缓存preparedStatement，也就是PSCache。  PSCache对支持游标的数据库性能提升巨大，比如说oracle。在mysql5.5以下的版本中没有PSCache功能，建议关闭掉。5.5及以上版本有PSCache，建议开启。 |
| maxOpenPreparedStatements | -1 | 要启用PSCache，必须配置大于0，当大于0时，  poolPreparedStatements自动触发修改为true。  在Druid中，不会存在Oracle下PSCache占用内存过多的问题，可以把这个数值配置大一些，比如说100 |
| validationQuery |  | 用来检测连接是否有效的sql，要求是一个查询语句。  如果validationQuery为null，testOnBorrow、testOnReturn、testWhileIdle都不会其作用。 |
| testOnBorrow | true | 申请连接时执行validationQuery检测连接是否有效，  做了这个配置会降低性能。 |
| testOnReturn | false | 归还连接时执行validationQuery检测连接是否有效，  做了这个配置会降低性能 |
| testWhileIdle | false | 建议配置为true，不影响性能，并且保证安全性。  申请连接的时候检测，如果空闲时间大于timeBetweenEvictionRunsMillis，执行validationQuery检测连接是否有效。 |
| timeBetweenEvictionRunsMillis |  | 有两个含义：  1) Destroy线程会检测连接的间隔时间  2) testWhileIdle的判断依据，详细看testWhileIdle属性的说明 |
| connectionInitSqls |  | 物理连接初始化的时候执行的sql |
| exceptionSorter | 根据dbType自动识别 | 当数据库抛出一些不可恢复的异常时，抛弃连接 |
| filters |  | 属性类型是字符串，通过别名的方式配置扩展插件，  常用的插件有：监控统计用的filter:stat 日志用的filter:log4j 防御sql注入的filter:wall |
| proxyFilters |  | 类型是List<com.alibaba.druid.filter.Filter>，如果同时配置了filters和proxyFilters，是组合关系，并非替换关系 |

**Druid 配置StatViewServlet配置**

Druid内置提供了一个StatViewServlet用于展示Druid的统计信息。

这个StatViewServlet的用途包括：1.提供监控信息展示的html页面。2.提供监控信息的JSON API。

①配置web.xml

StatViewServlet是一个标准的javax.servlet.http.HttpServlet，需要配置在你web应用中的WEB-INF/web.xml中。

|  |
| --- |
| <**servlet**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**servlet-class**>com.alibaba.druid.support.http.StatViewServlet</**servlet-class**> </**servlet**> <**servlet-mapping**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**url-pattern**>/druid/\*</**url-pattern**> </**servlet-mapping**> |

根据配置中的url-pattern来访问内置监控页面，如果是上面的配置，内置监控页面的首页是/druid/index.html

②配置监控页面访问密码

需要配置Servlet的 loginUsername 和 loginPassword这两个初始参数。

|  |
| --- |
| *<!-- 配置 Druid 监控信息显示页面 -->* <**servlet**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**servlet-class**>com.alibaba.druid.support.http.StatViewServlet</**servlet-class**>  <**init-param**>  *<!-- 允许清空统计数据 -->* <**param-name**>resetEnable</**param-name**>  <**param-value**>true</**param-value**>  </**init-param**>  <**init-param**>  *<!-- 用户名 -->* <**param-name**>loginUsername</**param-name**>  <**param-value**>druid</**param-value**>  </**init-param**>  <**init-param**>  *<!-- 密码 -->* <**param-name**>loginPassword</**param-name**>  <**param-value**>druid</**param-value**>  </**init-param**> </**servlet**> <**servlet-mapping**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**url-pattern**>/druid/\*</**url-pattern**> </**servlet-mapping**> |

③配置allow和deny

StatViewSerlvet展示出来的监控信息比较敏感，是系统运行的内部情况，如果你需要做访问控制，可以配置allow和deny这两个参数。

|  |
| --- |
| <**servlet**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**servlet-class**>com.alibaba.druid.support.http.StatViewServlet</**servlet-class**>  <**init-param**>  <**param-name**>allow</**param-name**>  <**param-value**>128.242.127.1/24,128.242.128.1</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>deny</**param-name**>  <**param-value**>128.242.127.4</**param-value**>  </**init-param**> </**servlet**> |

**判断规则**

deny优先于allow，如果在deny列表中，就算在allow列表中，也会被拒绝。

如果allow没有配置或者为空，则允许所有访问

**ip配置规则**

配置的格式

<IP>

或者

<IP>/<SUB\_NET\_MASK\_size>

其中

128.242.127.1/24

24表示，前面24位是子网掩码，比对的时候，前面24位相同就匹配。

不支持IPV6

由于匹配规则不支持IPV6，配置了allow或者deny之后，会导致IPV6无法访问。

④配置resetEnable

在StatViewSerlvet输出的html页面中，有一个功能是Reset All，执行这个操作之后，会导致所有计数器清零，重新计数。你可以通过配置参数关闭它。

|  |
| --- |
| <**servlet**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**servlet-class**>com.alibaba.druid.support.http.StatViewServlet</**servlet-class**>  <**init-param**>  <**param-name**>resetEnable</**param-name**>  <**param-value**>false</**param-value**>  </**init-param**> </**servlet**> |

**1、注解配置（SpringBoog）**

Druid是一个JDBC组件，它包括三部分：

DruidDriver 代理Driver，能够提供基于Filter－Chain模式的插件体系。

DruidDataSource 高效可管理的数据库连接池。

SQLParser

Druid可以做什么？

可以监控数据库访问性能，Druid内置提供了一个功能强大的StatFilter插件，能够详细统计SQL的执行性能，这对于线上分析数据库访问性能有帮助。

替换DBCP和C3P0。Druid提供了一个高效、功能强大、可扩展性好的数据库连接池。

数据库密码加密。直接把数据库密码写在配置文件中，这是不好的行为，容易导致安全问题。DruidDruiver和DruidDataSource都支持PasswordCallback。

SQL执行日志，Druid提供了不同的LogFilter，能够支持Common-Logging、Log4j和JdkLog，你可以按需要选择相应的LogFilter，监控你应用的数据库访问情况。

扩展JDBC，如果你要对JDBC层有编程的需求，可以通过Druid提供的Filter-Chain机制，很方便编写JDBC层的扩展插件。

①pom.xml文件添加依赖

|  |
| --- |
| <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.29</version>  </dependency> |

②application.properties配置信息

|  |
| --- |
| spring:  datasource:  url: jdbc:mysql://localhost:3306/test?characterEncoding:utf-8  username: root  password: 123456  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  #连接池的配置信息  ## 初始化大小，最小，最大  initialSize: 5  minIdle: 5  maxActive: 20  # 配置获取连接等待超时的时间  maxWait: 60000  # 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒  timeBetweenEvictionRunsMillis: 60000  # 配置一个连接在池中最小生存的时间，单位是毫秒  minEvictableIdleTimeMillis: 300000  validationQuery: SELECT 1 FROM DUAL  testWhileIdle: **true**  testOnBorrow: **false**  testOnReturn: **false**  # 打开PSCache，并且指定每个连接上PSCache的大小  poolPreparedStatements: **true**  maxPoolPreparedStatementPerConnectionSize: 20  # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙  filters: stat,wall,log4j  # 通过connectProperties属性来打开mergeSql功能；慢SQL记录  connectionProperties: druid.stat.mergeSql:**true**;druid.stat.slowSqlMillis:5000  # 合并多个DruidDataSource的监控数据  useGlobalDataSourceStat : **true** |

③配置Druid连接池

|  |
| --- |
| /\*\*  \* DruidDBConfig类被@Configuration标注，用作配置信息；  \* DataSource对象被@Bean声明，为Spring容器所管理，  \* **@Primary表示这里定义的DataSource将覆盖其他来源的DataSource**。  \* **@author** ASUS  \*  \*/  @Configuration  **public** **class** DruidDBConfig {  **private** Logger logger = LoggerFactory.*getLogger*(DruidDBConfig.**class**);  @Value("${spring.datasource.url}")  **private** String dbUrl;  @Value("${spring.datasource.username}")  **private** String username;  @Value("${spring.datasource.password}")  **private** String password;  @Value("${spring.datasource.driver-class-name}")  **private** String driverClassName;  @Value("${spring.datasource.initialSize}")  **private** **int** initialSize;  @Value("${spring.datasource.minIdle}")  **private** **int** minIdle;  @Value("${spring.datasource.maxActive}")  **private** **int** maxActive;  @Value("${spring.datasource.maxWait}")  **private** **int** maxWait;  @Value("${spring.datasource.timeBetweenEvictionRunsMillis}")  **private** **int** timeBetweenEvictionRunsMillis;  @Value("${spring.datasource.minEvictableIdleTimeMillis}")  **private** **int** minEvictableIdleTimeMillis;  @Value("${spring.datasource.validationQuery}")  **private** String validationQuery;  @Value("${spring.datasource.testWhileIdle}")  **private** **boolean** testWhileIdle;  @Value("${spring.datasource.testOnBorrow}")  **private** **boolean** testOnBorrow;  @Value("${spring.datasource.testOnReturn}")  **private** **boolean** testOnReturn;  @Value("${spring.datasource.poolPreparedStatements}")  **private** **boolean** poolPreparedStatements;  @Value("${spring.datasource.maxPoolPreparedStatementPerConnectionSize}")  **private** **int** maxPoolPreparedStatementPerConnectionSize;  @Value("${spring.datasource.filters}")  **private** String filters;  @Value("{spring.datasource.connectionProperties}")  **private** String connectionProperties;  @Bean // 声明其为Bean实例  @Primary // 在同样的DataSource中，首先使用被标注的DataSource  **public** DataSource dataSource() {  DruidDataSource datasource = **new** DruidDataSource();  datasource.setUrl(**this**.dbUrl);  datasource.setUsername(username);  datasource.setPassword(password);  datasource.setDriverClassName(driverClassName);  // configuration  datasource.setInitialSize(initialSize);  datasource.setMinIdle(minIdle);  datasource.setMaxActive(maxActive);  datasource.setMaxWait(maxWait);  datasource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);  datasource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);  datasource.setValidationQuery(validationQuery);  datasource.setTestWhileIdle(testWhileIdle);  datasource.setTestOnBorrow(testOnBorrow);  datasource.setTestOnReturn(testOnReturn);  datasource.setPoolPreparedStatements(poolPreparedStatements);  datasource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);  **try** {  datasource.setFilters(filters);  } **catch** (SQLException e) {  logger.error("druid configuration initialization filter", e);  }  datasource.setConnectionProperties(connectionProperties);  **return** datasource;  }  } |

④配置StatView的Servlet：

|  |
| --- |
| Filter的实现类 |
| @WebFilter(filterName = "druidWebStatFilter", urlPatterns = "/\*", initParams = {  @WebInitParam(name = "exclusions", value = "\*.js,\*.gif,\*.jpg,\*.bmp,\*.png,\*.css,\*.ico,/druid/\*")// 忽略资源  })  **public** **class** DruidStatFilter **extends** WebStatFilter {  } |
| StatViewServlet |
| @WebServlet(urlPatterns = "/druid/\*", initParams = { @WebInitParam(name = "allow", value = "127.0.0.1,192.168.163.1"), // IP白名单(没有配置或者为空，则允许所有访问)  @WebInitParam(name = "deny", value = "192.168.1.73"), // IP黑名单  // (存在共同时，deny优先于allow)  @WebInitParam(name = "loginUsername", value = "admin"), // 用户名  @WebInitParam(name = "loginPassword", value = "123456"), // 密码  @WebInitParam(name = "resetEnable", value = "false")// 禁用HTML页面上的“Reset  // All”功能  })  **public** **class** DruidStatViewServlet **extends** StatViewServlet {  **private** **static** **final** **long** ***serialVersionUID*** = -2688872071445249539L;  } |
| 这两个类相当于在web.xml中声明了一个servlet， 等价于如下的配置信息（web.xml): |
|  |

⑤运行界面http://localhost:8080/druid/index.html，即可以进行监控页面

|  |
| --- |
|  |

**2、java config配置(Web项目)**

①pom.xml文件

|  |
| --- |
| <**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/maven-v4\_0\_0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>com.cxp</**groupId**>  <**artifactId**>druid\_xml</**artifactId**>  <**packaging**>war</**packaging**>  <**version**>1.0-SNAPSHOT</**version**>  <**name**>druid\_xml Maven Webapp</**name**>  <**url**>http://maven.apache.org</**url**>  <**dependencies**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>3.8.1</**version**>  <**scope**>test</**scope**>  </**dependency**>   *<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.5</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>4.3.14.RELEASE</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.springframework/spring-tx -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-tx</**artifactId**>  <**version**>4.3.14.RELEASE</**version**>  </**dependency**>   *<!-- servlet支持开启-->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**>  <**version**>4.0.0</**version**>  </**dependency**>   </**dependencies**>  <**build**>  <**finalName**>druid\_xml</**finalName**>  </**build**> </**project**> |

②web容器配置

|  |
| --- |
| @Configuration **public class** DruidConfig **implements** WebApplicationInitializer {  **public void** onStartup(ServletContext servletContext) **throws** ServletException {  AnnotationConfigWebApplicationContext context =**new** AnnotationConfigWebApplicationContext();  context.setServletContext(servletContext);   ServletRegistration.Dynamic servlet = servletContext.addServlet(**"DruidStatView"**,**new** StatViewServlet());  servlet.addMapping(**"/druid/\*"**);  servlet.setInitParameter(**"loginUsername"**,**"admin"**);  servlet.setInitParameter(**"loginPassword"**,**"123456"**);   WebStatFilter webStatFilter =**new** WebStatFilter();  FilterRegistration.Dynamic filter= servletContext.addFilter(**"DruidWebStatFilter"**,webStatFilter);  filter.setInitParameter(**"exclusions"**,**"\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*"**);  filter.addMappingForUrlPatterns(EnumSet.*of*(DispatcherType.***REQUEST***),**true**,**"/\*"**);  } } |

③测试

地址栏访问：<http://localhost:8080/druid/index.html> 进入页面：

|  |
| --- |
|  |

**3、xml配置(Web项目)**

①pom.xml文件

|  |
| --- |
| <**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/maven-v4\_0\_0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>  <**groupId**>com.cxp</**groupId**>  <**artifactId**>druid\_xml</**artifactId**>  <**packaging**>war</**packaging**>  <**version**>1.0-SNAPSHOT</**version**>  <**name**>druid\_xml Maven Webapp</**name**>  <**url**>http://maven.apache.org</**url**>  <**dependencies**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>3.8.1</**version**>  <**scope**>test</**scope**>  </**dependency**>   *<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.5</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>5.0.3.RELEASE</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.springframework/spring-tx -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-tx</**artifactId**>  <**version**>5.0.3.RELEASE</**version**>  </**dependency**>   </**dependencies**>  <**build**>  <**finalName**>druid\_xml</**finalName**>  </**build**> </**project**> |

②web.xml文件配置

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"  id="WebApp\_ID" version="3.0"**>  <**display-name**>Archetype Created Web Application</**display-name**>   <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:applicationContext.xml</**param-value**>  </**context-param**>  <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  </**listener**>  <**servlet**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**servlet-class**>com.alibaba.druid.support.http.StatViewServlet</**servlet-class**>  <**init-param**>  *<!-- 白名单 -->* <**param-name**>allow</**param-name**>  <**param-value**>192.168.1.120/24,111.206.116.68,127.0.0.1</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>loginUsername</**param-name**>  <**param-value**>admin</**param-value**>  </**init-param**>  <**init-param**>  <**param-name**>loginPassword</**param-name**>  <**param-value**>admin</**param-value**>  </**init-param**>  </**servlet**>  <**servlet-mapping**>  <**servlet-name**>DruidStatView</**servlet-name**>  <**url-pattern**>/druid/\*</**url-pattern**>  </**servlet-mapping**>   *<!-- 解决post乱码 -->* <**filter**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**filter-class**>org.springframework.web.filter.CharacterEncodingFilter</**filter-class**>  <**init-param**>  <**param-name**>encoding</**param-name**>  <**param-value**>utf-8</**param-value**>  </**init-param**>  </**filter**>  <**filter-mapping**>  <**filter-name**>CharacterEncodingFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**>    <**filter**>  <**filter-name**>DruidWebStatFilter</**filter-name**>  <**filter-class**>com.alibaba.druid.support.http.WebStatFilter</**filter-class**>  <**init-param**>  <**param-name**>exclusions</**param-name**>  <**param-value**>\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*</**param-value**>  </**init-param**>  </**filter**>  <**filter-mapping**>  <**filter-name**>DruidWebStatFilter</**filter-name**>  <**url-pattern**>/\*</**url-pattern**>  </**filter-mapping**> </**web-app**> |

③application.xml数据源配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  xmlns:aop="http://www.springframework.org/schema/aop" xmlns:mvc="http://www.springframework.org/schema/mvc"  xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd"**>   <**context:property-placeholder location="classpath:db.properties"**/>  *<!-- 数据库连接池 -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" destroy-method="close"**>  <**property name="driverClassName" value="${jdbc.driver}"** />  <**property name="url" value="${jdbc.url}"** />  <**property name="username" value="${jdbc.username}"** />  <**property name="password" value="${jdbc.password}"** />   *<!-- 配置初始化大小、最小、最大 -->* <**property name="initialSize" value="1"** />  <**property name="minIdle" value="1"** />  <**property name="maxActive" value="20"** />  *<!-- 配置获取连接等待超时的时间 -->* <**property name="maxWait" value="60000"** />  *<!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->* <**property name="timeBetweenEvictionRunsMillis" value="60000"** />   *<!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->* <**property name="minEvictableIdleTimeMillis" value="300000"** />  <**property name="validationQuery" value="SELECT 'x'"** />  <**property name="testWhileIdle" value="true"** />  <**property name="testOnBorrow" value="false"** />  <**property name="testOnReturn" value="false"** />  *<!-- 打开PSCache，并且指定每个连接上PSCache的大小 -->* <**property name="poolPreparedStatements" value="true"** />  <**property name="maxPoolPreparedStatementPerConnectionSize" value="20"** />  *<!-- 配置监控统计拦截的filters，去掉后监控界面sql无法统计 -->* <**property name="filters" value="stat，wall"** />  </**bean**> </**beans**> |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://localhost:3306/cheng1?characterEncoding=utf-8 jdbc.username**=**root jdbc.password**=**123456** |

④测试

地址栏访问：<http://localhost:8080/druid/index.html> 进入页面：

|  |
| --- |
|  |

**4、SpringBoot(java config配置)**

①pom.xml

|  |
| --- |
| *<?***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**>com.cxp</**groupId**>  <**artifactId**>druid\_javaconfig</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**>  <**packaging**>jar</**packaging**>   <**name**>druid\_javaconfig</**name**>  <**description**>Demo project for Spring Boot</**description**>   <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.10.RELEASE</**version**>  <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>   <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**java.version**>1.8</**java.version**>  </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>   *<!-- druid连接池 -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**>  </**dependency**>  </**dependencies**>   <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**>   </**project**> |

②java config配置

|  |
| --- |
| @Configuration **public class** DruidConfig {  @Bean  **public** ServletRegistrationBean druidServlet() {   ServletRegistrationBean servletRegistrationBean = **new** ServletRegistrationBean(**new** StatViewServlet(), **"/druid/\*"**);  *//登录查看信息的账号密码.* servletRegistrationBean.addInitParameter(**"loginUsername"**,**"admin"**);  servletRegistrationBean.addInitParameter(**"loginPassword"**,**"123456"**);  **return** servletRegistrationBean;  }   @Bean  **public** FilterRegistrationBean filterRegistrationBean() {  FilterRegistrationBean filterRegistrationBean = **new** FilterRegistrationBean();  filterRegistrationBean.setFilter(**new** WebStatFilter());  filterRegistrationBean.addUrlPatterns(**"/\*"**);  filterRegistrationBean.addInitParameter(**"exclusions"**, **"\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*"**);  **return** filterRegistrationBean;  } } |

第十三篇 整合Mybatis使用分页插件

①pom.xml文件中加入依赖

|  |
| --- |
| *<!--pagehelper -->* <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  <**version**>1.1.1</**version**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**> </**dependency**> |
| application.properties配置文件 |
| **datasource.password**=**123456 datasource.username**=**root datasource.url**=**jdbc:mysql://localhost:3306/test?characterEncoding=utf-8 datasource.driver-class-name**=**com.mysql.jdbc.Driver** |

②创建MybatisConfig类

|  |
| --- |
| @Configuration **public class** DataSourceConfig {  @Autowired  **private** Environment **environment**;  @Bean  **public** DataSource dataSource(){  DruidDataSource druidDataSource=**new** DruidDataSource();  druidDataSource.setUrl(**environment**.getProperty(**"datasource.url"**));  druidDataSource.setUsername(**environment**.getProperty(**"datasource.username"**));  druidDataSource.setDriverClassName(**environment**.getProperty(**"datasource.driver-class-name"**));  druidDataSource.setPassword(**environment**.getProperty(**"datasource.password"**));  druidDataSource.setMinIdle(5);  druidDataSource.setMaxWait(30000);  druidDataSource.setMaxActive(30);  **return** druidDataSource;  }  */\*  \* 注册MyBatis分页插件PageHelper  \*/* @Bean  **public** PageHelper pageHelper() {  PageHelper pageHelper = **new** PageHelper();  Properties p = **new** Properties();  p.setProperty(**"offsetAsPageNum"**, **"true"**);  p.setProperty(**"rowBoundsWithCount"**, **"true"**);  p.setProperty(**"reasonable"**, **"true"**);  pageHelper.setProperties(p);  **return** pageHelper;  } } |
| Mapper接口 |
| **public interface** ProductMapper {  @Select(value = **"select \* from product"**)  **public** List<Map<String,Object>> findProductList(); } |

③在controller中直接使用PageHelper

|  |
| --- |
| @RestController  **public class** ProductController {  @Autowired  **private** ProductMapper **productMapper**;   @RequestMapping(value = **"productlist"**)  **public** List<Map<String,Object>> productList() {  *//第一个参数为页码，第二个参数为每页多少行数据* PageHelper.*startPage*(1,10);  **return productMapper**.findProductList();  } } |

第十四篇 支持JSP

**①构建SpringBoot项目**

|  |
| --- |
|  |

点击Next后选择我们预先加入到工程的springboot内置的jar包组件，我们这里选择一个web组件。

|  |
| --- |
|  |

可以看到上图中比我们BootJsp项目中多出来了一个类文件,**ServletInitlalizer**，打开该类我们不难发现它继承了**SpringBootServletInitializer**这个父类，而**SpringBootServletInitializer**这个类是**springboot**提供的**web**程序初始化的入口，当我们使用外部容器（后期文章讲解使用外部tomcat如何运行项目）运行项目时会自动加载并且装配。  
实现了**SpringBootServletInitializer**的子类需要重写一个configure方法，方法内自动根据**LessontwoApplication**.class的类型创建一个**SpringApplicationBuilder**交付给**springboot**框架来完成初始化运行配置。

我们打开pom.xml（maven配置文件）可以看到我们之前构建项目时已经添加了web模块，而springboot给我们自动添加了spring-boot-starter-tomcat配置引入。springboot内部集成了tomcat组件，这里我们就不需要重复引入tomcat组件。

**②添加JSP的Maven支持**

|  |
| --- |
| *<!-- SpringBoot tomcat jsp支持开启-->* <**dependency**>  <**groupId**>org.apache.tomcat.embed</**groupId**>  <**artifactId**>tomcat-embed-jasper</**artifactId**>  <**version**>9.0.4</**version**> </**dependency**> *<!-- servlet支持开启-->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**>  <**version**>4.0.0</**version**> </**dependency**> |

**③初尝试运行项目**

点击绿色三角号即可开始运行项目，我们先来尝试下点击后看看控制台输入的日志信息。可以看到项目已经正常运行了，右上角的绿色三角号也变成了重启的按钮，如下图8所示：

|  |
| --- |
|  |

项目运行如图所示：

|  |
| --- |
|  |

**④配置视图**

如果是使用过eclipse工具的人肯定知道jsp文件之前是在/webapp/WEB-INF/jsp目录下，那么我们也沿用eclipse的风格。

1. 在main目录下创建webapp文件夹

2. 在webapp下创建jsp文件夹如下图10所示：

3. 修改application.properties文件让springmvc支持视图的跳转目录指向为/main/webapp/jsp，配置如下图所示：

|  |
| --- |
|  |
|  |

创建index.jsp

我们在webapps/jsp目录下创建一个index.jsp用于测试我们的配置是否已经支持了jsp，新建jsp文件如下图所示：

|  |
| --- |
|  |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <%@ **page isELIgnored**="**false**" %> <**html**> <**head**>  <**title**>Title</**title**> </**head**> <**body**>  <**h1**>index.jsp</**h1**>  <**h3**>**${**pageContext.request.contextPath**}**</**h3**>  <**h3**>**${**hello1**}**</**h3**> </**body**> </**html**> |

**⑤创建IndexController**

|  |
| --- |
|  |
| @Controller **public class** IndexController {  @RequestMapping(**"/myIndex/{name}"**)  **public** String hello(Model model, @PathVariable(**"name"**) String name){  System.***out***.println(123);  model.addAttribute(**"hello1"**,**"annotation.RequestMapping"**+name);  **return "index"**;  } } |

**⑥启动项目**

访问：[http://localhost:8080/myIndex/](http://localhost:8080/myIndex/xta)张三丰

|  |
| --- |
|  |

第十五篇 添加支持CORS跨域访问

CORS（Cross-Origin Resource Sharing）"跨域资源共享"，是一个W3C标准，它允许浏览器向跨域服务器发送Ajax请求，打破了Ajax只能访问本站内的资源限制，CORS在很多地方都有被使用，微信支付的JS支付就是通过JS向微信服务器发送跨域请求。开放Ajax访问可被跨域访问的服务器大大减少了后台开发的工作，前后台工作也可以得到很好的明确以及分工，下面我们就看讲一下如何让你的SpringBoot项目支持CORS跨域。

**①构建项目**

使用IDEA开发工具创建一个SpringBoot项目，预先添加Web依赖即可，项目结构如下图:

|  |
| --- |
|  |

**②CORSConfiguration**

我们只需要添加项目Web依赖就可以了，下面我们开始添加CORS的配置信息，我们创建一个CORSConfiguration配置类，如下图所示：

|  |
| --- |
| **import** org.springframework.context.annotation.Configuration; **import** org.springframework.web.servlet.config.annotation.CorsRegistry; **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter; @Configuration **public class** CORSConfiguration **extends** WebMvcConfigurerAdapter{  @Override  **public void** addCorsMappings(CorsRegistry registry) {  registry.addMapping(**"/\*\*"**)  .allowedMethods(**"\*"**)  .allowedOrigins(**"\*"**)  .allowedHeaders(**"\*"**);  } } |
| 上图内我们的CORSConfiguration配置类继承了WebMvcConfiugrationAdaper父类并且重写了addCorsMappings方法，我们来简单介绍下我们的配置信息：  addMapping：配置可以被跨域的路径，可以任意配置，可以具体到直接请求路径。  allowedMethods：允许所有的请求方法访问该跨域资源服务器，如：POST、GET、PUT、DELETE等。  allowedOrigins：允许所有的请求域名访问我们的跨域资源，可以固定单条或者多条内容，如："http://www.baidu.com"，只有百度可以访问我们的跨域资源。  allowedHeaders：允许所有的请求header访问，可以自定义设置任意请求头信息，如："X-YAUTH-TOKEN"。 |

**③编写跨域资源请求**

我们的跨域配置到目前来说已经配置完成了，SpringBoot已经为我们内置相关配置，我们只需要重写方法修改部分参数即可，下面我们来创建一个测试跨域资源的控制器，如下图所示：

|  |
| --- |
| @RestController **public class** CorsController {  @RequestMapping(value = **"/cros"**)  **public** String hello(String name){  System.***out***.println(name);  **return "this is cors info"**+name;  } }  在上面的CrosController控制器内，我们仅仅添加了一个测试返回文本的内容，当然这个控制器可以处理任意业务逻辑。 |

**③测试跨域请求**

|  |
| --- |
| 我们在项目外创建一个index.html页面（上传码云后会在resources目录找到），页面内添加部分jquery代码，如下所示： |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <meta charset="{CHARSET}">  <title></title>  <script type="text/javascript" src="jquery-1.8.3.min.js"></script>  <script type="text/javascript">  $(function(){  $(document).on("click","#cors",function(){  $.ajax({  type:"get",  url:"http://192.168.0.138:8080/cros",  data:{'name':'张三丰'},  success:function(data){  console.log(data);  alert(data);  }  });  });  });  </script>  </head>  <body>  <input type="button" id="cors" value="CORS跨域请求"/>  </body>  </html> |

**④启动web项目**

接下来我们点击“CORS跨域测试”按钮，查看下效果，如下图所示：

|  |
| --- |
|  |

第十六篇 使用WebSocket配置广播式通信

WebSocket为浏览器和服务端提供了双工异步通信的功能，浏览器可以向服务端发送消息，服务端也可以向浏览器发送消息。WebSocket需要浏览器的支持，目前大多数主流的浏览器都是支持的。

WebSocket是通过socket来实现双工异步通信能力的。但是直接使用WebSocket协议开发程序闲的特别繁琐，一般情况我们使用STOMP来实现交互。

0、什么是Stomp

STOMP即Simple (or Streaming) Text Orientated Messaging Protocol，简单(流)文本定向消息协议，它提供了一个可互操作的连接格式，允许STOMP客户端与任意STOMP消息代理（Broker）进行交互。STOMP协议由于设计简单，易于开发客户端，因此在多种语言和多种平台上得到广泛地应用。

SpringBoot在tomcat7内嵌版本就已经开始支持了WebSocket的支持，配置源码存在于org.springframework.boot.autoconfigure.websocket下，接下来我们来讲解SpringBoot项目如何配置使用WebSocket。

|  |
| --- |
|  |

①pom.xml文件中加入依赖

|  |
| --- |
| *<?***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**>cxp.xuexi</**groupId**>  <**artifactId**>websocket</**artifactId**>  <**version**>0.0.1-SNAPSHOT</**version**>  <**packaging**>war</**packaging**>   <**name**>websocket</**name**>  <**description**>Demo project for Spring Boot</**description**>   <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>1.5.10.RELEASE</**version**>  <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>   <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**java.version**>1.8</**java.version**>  </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-websocket</**artifactId**>  </**dependency**>   <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  *<!--<scope>provided</scope>-->* </**dependency**>  *<!-- spring boot tomcat jsp 支持开启 -->* <**dependency**>  <**groupId**>org.apache.tomcat.embed</**groupId**>  <**artifactId**>tomcat-embed-jasper</**artifactId**>  </**dependency**>  *<!--servlet支持开启-->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**>  </**dependency**>  </**dependencies**>   <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**>  </**project**> |

②配置WebSocket

下面我们创建一个WebSocketConfig配置类，添加SpringBoot项目对WebSocket的支持配置

|  |
| --- |
| **package** cxp.xuexi.configuration;  **import** org.springframework.context.annotation.Configuration; **import** org.springframework.messaging.simp.config.MessageBrokerRegistry; **import** org.springframework.web.socket.config.annotation.AbstractWebSocketMessageBrokerConfigurer; **import** org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker; **import** org.springframework.web.socket.config.annotation.StompEndpointRegistry;  */\*\*  \* Created by ASUS on 2018/2/25.  \*/* @Configuration @EnableWebSocketMessageBroker **public class** WebSocketConfig **extends** AbstractWebSocketMessageBrokerConfigurer {  @Override  **public void** registerStompEndpoints(StompEndpointRegistry registry) {  registry.addEndpoint(**"/endpointWisely"**).withSockJS();  }   @Override  **public void** configureMessageBroker(MessageBrokerRegistry registry) {  registry.enableSimpleBroker(**"/topic"**);  } } |

如上图所示，我们通过@EnableWebSocketMessageBroker注解开启使用STOMP协议来传输消息，并且实现了registerStompEndpoints方法添加了对应的STOMP使用SockJS协议。接下来我们配置两个实体用于浏览器与服务器端的通信交互。

③浏览器向服务端发送消息实体

我们创建一个简单的WiselyMessage实体，用于浏览器向服务端发送消息参数，如下图所示：

|  |
| --- |
| **package** cxp.xuexi.pojo;  */\*\*  \* Created by ASUS on 2018/2/25.  \* 用于浏览器向服务端发送消息参数  \*/* **public class** WiselyMessage {  **private** String **name**;   **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  } } |

我们消息实体内仅添加了一个名称，我们一会会在前台界面将name字段的值传输到后台。

④服务端向浏览器发送的实体

下面我们再来创建一个服务端广播通知浏览器的实体类型，如下图所示：

|  |
| --- |
| **package** cxp.xuexi.pojo;  */\*\*  \* Created by ASUS on 2018/2/25.  \* 服务端广播通知浏览器的实体类型  \*/* **public class** WiselyResponse {  **private** String **responseMessage**;   **public** String getResponseMessage() {  **return responseMessage**;  }   **public void** setResponseMessage(String responseMessage) {  **this**.**responseMessage** = responseMessage;  }   **public** WiselyResponse(String responseMessage) {  **this**.**responseMessage** = responseMessage;  }   **public** WiselyResponse() {  } } |

上图内仅一个消息内容返回到前台。我们通讯实体已经配置完成，下面我们开始编写通讯交互的控制器以及添加配置WebSocket的请求路径。

⑤配置WebSocket控制器

我们添加一个名叫WsController的控制器并添加对应的@MessageMapping注解开启WebSocket的访问地址映射，如下图所示：

|  |
| --- |
| **package** cxp.xuexi.controller;  **import** cxp.xuexi.pojo.WiselyMessage; **import** cxp.xuexi.pojo.WiselyResponse; **import** org.springframework.messaging.handler.annotation.MessageMapping; **import** org.springframework.messaging.handler.annotation.SendTo; **import** org.springframework.stereotype.Controller;  */\*\*  \* Created by ASUS on 2018/2/25.  \*/* @Controller **public class** WsController {  @MessageMapping(**"/welcome"**)  @SendTo(**"/topic/getResponse"**)  **public** WiselyResponse say(WiselyMessage wiselyMessage) **throws** Exception{  Thread.*sleep*(3000);  **return new** WiselyResponse(**"欢迎使用WebSocket ："**+wiselyMessage.getName());  } } |

上图内，当浏览器向服务端发送请求时，通过@MessageMapping映射/welcome这个地址，类似@RequestMapping，当服务端有消息存在时，会对订阅@SendTo中路径的浏览器发送请求。接下来我们添加一个Jsp页面来配置WebSocket的JS使用方式。

⑥配置JSP支持WebSocket

在编写对应的代码之前，我们需要下载三个js文件，分别是：jquery.min.js、sockjs.min.js、stomp.min.js 。Jsp代码如下图所示：

|  |
| --- |
| <%@ **page contentType**="**text/html;charset=UTF-8**" **language**="**java**" %> <**html**> <**head**>  <**title**>WebScoket广播式</**title**>  <**script src="/jquery.js"**></**script**>  <**script src="/sockjs.min.js"**></**script**>  <**script src="/stomp.min.js"**></**script**> </**head**> <**body onload="***disconnect*()**"**> <**button id="connect" onclick="***connect*()**"**>连接</**button**> <**button id="disconnect" onclick="***disconnect*()**" disabled="disabled"**>断开连接</**button**><**br**/> <**div id="inputDiv"**>  输入名称：<**input type="text" id="name"**/><**br**/>  <**button id="sendName" onclick="***sendName*()**"**>发送</**button**><**br**/>  <**p id="response"**></**p**> </**div**> <**script**>  **var *stompClient*** = **null**;  *//设置连接状态控制显示隐藏* **function** *setConnected*(connected)  {  **$**(**"#connect"**).attr(**"disabled"**,connected);  **$**(**"#disconnect"**).attr(**"disabled"**,!connected);  **if**(!connected) {  **$**(**"#inputDiv"**).hide();  }**else**{  **$**(**"#inputDiv"**).show();  }  **$**(**"#reponse"**).html(**""**);  }  *//连接* **function** *connect*()  {  **var** socket = **new SockJS**(**"/endpointWisely"**);  ***stompClient*** = **Stomp**.over(socket);  ***stompClient***.connect({},**function** (frame) {  *setConnected*(**true**);  **console**.log(**"connected : "**+frame);  ***stompClient***.subscribe(**"/topic/getResponse"**,**function** (response) {  *showResponse*(***JSON***.parse(response.**body**).responseMessage);  })  })  }  *//断开连接* **function** *disconnect*(){  **if**(***stompClient***!=**null**)  {  ***stompClient***.disconnect();  }  *setConnected*(**false**);  **console**.log(**"disconnected!"**);  }  *//发送名称到后台* **function** *sendName*(){  **var** name = **$**(**"#name"**).val();  ***stompClient***.send(**"/welcome"**,{},***JSON***.stringify({**'name'**:name}));  }  *//显示socket返回消息内容* **function** *showResponse*(message)  {  **$**(**"#response"**).html(message);  } </**script**> </**body**> </**html**> |

我们在Jsp页面加载的时候先执行了一边关闭socket通道的操作，当我们点击《连接》按钮时会调用如下图7所示的Js方法完成开启连接操作：

|  |
| --- |
| *//连接* **function** *connect*() {  **var** socket = **new SockJS**(**"/endpointWisely"**);  ***stompClient*** = **Stomp**.over(socket);  ***stompClient***.connect({},**function** (frame) {  *setConnected*(**true**);  **console**.log(**"connected : "**+frame);  ***stompClient***.subscribe(**"/topic/getResponse"**,**function** (response) {  *showResponse*(***JSON***.parse(response.**body**).responseMessage);  })  }) }; |

可以看到我们通过SockJs来注册了endpoine，并且通过客户端开启通过连接，连接成功后调用setConnected方法来修改界面的元素显示隐藏，如下图所示：

|  |
| --- |
| *//设置连接状态控制显示隐藏* **function** *setConnected*(connected) {  **$**(**"#connect"**).attr(**"disabled"**,connected);  **$**(**"#disconnect"**).attr(**"disabled"**,!connected);  **if**(!connected) {  **$**(**"#inputDiv"**).hide();  }**else**{  **$**(**"#inputDiv"**).show();  }  **$**(**"#reponse"**).html(**""**); } |

通过注册/topic/getReponse路径来回去服务端向浏览器端的请求数据内容，并且调用showResponse方法显示在页面中，showResponse方法如下图所示：

|  |
| --- |
| *//显示socket返回消息内容* **function** *showResponse*(message) {  **$**(**"#response"**).html(message); } |

下面是断开连接的代码，如下图所示：

|  |
| --- |
| *//断开连接* **function** *disconnect*(){  **if**(***stompClient***!=**null**)  {  ***stompClient***.disconnect();  }  *setConnected*(**false**);  **console**.log(**"disconnected!"**); } |

上面我们关于WebSocket的配置已经完成，我们需要访问对应的Jsp来测试，我们先修改application.properties配置文件添加JSP的路径映射，如下图所示：

|  |
| --- |
| **spring.mvc.view.prefix**=**/WEB-INF/jsp/ spring.mvc.view.suffix**=**.jsp** |

⑦映射MVC到Index.jsp

我们创建一个名叫做WebConfig的配置类并继承WebMvcConfigurerAdapter，实现addViewControllers方法添加对应的路径映射配置，如下图所示：

|  |
| --- |
| **package** cxp.xuexi.configuration;  **import** org.springframework.context.annotation.Configuration; **import** org.springframework.web.servlet.config.annotation.ViewControllerRegistry; **import** org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;  */\*\*  \* Created by ASUS on 2018/2/25.  \*/* @Configuration **public class** WebConfig **extends** WebMvcConfigurerAdapter {  @Override  **public void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController(**"/index"**).setViewName(**"/index"**);  } } |

可以看到我们映射/index地址到/index.jsp也就是到webapp/WEB-INF/jsp/index.jsp，下面我们来启动项目测试效果。

⑧运行测试

我们启动项目后，访问地址：127.0.0.1:8080/index，界面输出内容如下图所示：

|  |
| --- |
| IMG_256 |

我们现在是断开连接的状态，我们尝试点击连接看下效果，如下图所示：

|  |
| --- |
| IMG_256 |

可以看到我们成功的开启了WebSocket通道，开启通道后显示了对应的输入内容，我们来尝试输入一个名称点击发送查看界面输出内容如下图所示：

|  |
| --- |
| IMG_256 |

点击《发送》按钮后等待3秒钟后，界面会输出欢迎的字样，证明我们的WebSocket通道已经成功的接通并且获取到了服务端返回到浏览器的数据。

下面我们打开多个浏览器Tab来测试广播式的消息是否可以发送到多个开启WebSocket通道的浏览器中。

|  |
| --- |
| IMG_256 |

上图内的三个浏览器都开启了WebSocket通道，下面我们在其中一个浏览器中输出一个名称并点击《发送》按钮后，挨个浏览器查看效果，我们的WebSocket服务端通过广播的形式完成了服务端的消息发送到浏览器中。。

第十七篇 使用SpringBoot进行上传

使用SpringBoot进行文件上传的方法和SpringMVC差不多，本文单独新建一个最简单的DEMO来说明一下。

主要步骤包括：

1、创建一个springboot项目工程，本例名称（demo-uploadfile）。

2、配置 pom.xml 依赖。

3、创建和编写文件上传的 Controller（包含单文件上传和多文件上传）。

4、创建和编写文件上传的 HTML 测试页面。

5、文件上传相关限制的配置（可选）。

6、运行测试。

1、单个文件上传

①application.properties配置文件

|  |
| --- |
| **spring.http.multipart.enabled**=**true** *#maxRequestSize是设置总上传的数据大小* **spring.http.multipart.max-request-size**=**100MB** *#单个文件大小* **spring.http.multipart.max-file-size**= **50MB** |
| 等价于： |
| @Configuration **public class** MulitiPartFileConfig {  @Bean  **public** MultipartConfigElement multipartConfigElement(){  MultipartConfigFactory factory = **new** MultipartConfigFactory();  *// 置文件大小限制 ,超出此大小页面会抛出异常信息* factory.setMaxFileSize(**"50MB"**); *//KB,MB  // 设置总上传数据总大小* factory.setMaxRequestSize(**"100MB"**);  *// 设置文件临时文件夹路径  // factory.setLocation("E://test//");  // 如果文件大于这个值，将以文件的形式存储，如果小于这个值文件将存储在内存中，默认为0  // factory.setMaxRequestSize(0);* **return** factory.createMultipartConfig();  } } |

②服务器端代码

|  |
| --- |
| */\*\*  \* 单个文件上传  \*  \** ***@param file*** *\** ***@param request*** *\** ***@return*** *\*/* @RequestMapping(**"/singleUploadFile"**) @ResponseBody **public** String singleUploadFile(@RequestParam(**"file"**) MultipartFile file,  HttpServletRequest request, String username, String password) {  System.***out***.println(username + **" : "** + password);  **if** (!file.isEmpty()) {  String saveFileName = file.getOriginalFilename();  File saveFile = **new** File(**"D:\\BaiduNetdiskDownload/"** + saveFileName);  **if** (!saveFile.getParentFile().exists()) {  saveFile.getParentFile().mkdirs();  }  **try** {  BufferedOutputStream out = **new** BufferedOutputStream(**new** FileOutputStream(saveFile));  out.write(file.getBytes());  out.flush();  out.close();  **return** saveFile.getName() + **" 上传成功"**;  } **catch** (FileNotFoundException e) {  e.printStackTrace();  **return "上传失败,"** + e.getMessage();  } **catch** (IOException e) {  e.printStackTrace();  **return "上传失败,"** + e.getMessage();  }  } **else** {  **return "上传失败，因为文件为空."**;  } } |

③页面代码

|  |
| --- |
| <**form method="post" action="/multiUploadFile" enctype="multipart/form-data"**>  <**input type="file" name="file"**/>  <**br**/>  name: <**input type="text" name="username"**/>  <**br**/>  password: <**input type="password" name="password"**/>  <**input type="submit"**/> </**form**> |

2、多个文件上传

①application.properties配置文件

|  |
| --- |
| **spring.http.multipart.enabled**=**true** *#maxRequestSize是设置总上传的数据大小* **spring.http.multipart.max-request-size**=**100MB** *#单个文件大小* **spring.http.multipart.max-file-size**= **50MB** |
| 等价于： |
| @Configuration **public class** MulitiPartFileConfig {  @Bean  **public** MultipartConfigElement multipartConfigElement(){  MultipartConfigFactory factory = **new** MultipartConfigFactory();  *// 置文件大小限制 ,超出此大小页面会抛出异常信息* factory.setMaxFileSize(**"50MB"**); *//KB,MB  // 设置总上传数据总大小* factory.setMaxRequestSize(**"100MB"**);  *// 设置文件临时文件夹路径  // factory.setLocation("E://test//");  // 如果文件大于这个值，将以文件的形式存储，如果小于这个值文件将存储在内存中，默认为0  // factory.setMaxRequestSize(0);* **return** factory.createMultipartConfig();  } } |

②服务器端代码

|  |
| --- |
| @RequestMapping(**"/multiUploadFile"**)  @ResponseBody  **public** String multiUploadFile(HttpServletResponse response,  HttpServletRequest request, String username, String password) {  System.***out***.println(username + **" : "** + password);  File savePath = **new** File(**"D:\\BaiduNetdiskDownload"**);  **if** (!savePath.exists()) {  savePath.mkdirs();  }   MultipartHttpServletRequest req = (MultipartHttpServletRequest) request;  List<MultipartFile> files = req.getFiles(**"file"**);  BufferedOutputStream bos = **null**;  MultipartFile file = **null**;  **if** (files != **null**) {  **for** (**int** i = 0; i < files.size(); ++i) {  file = files.get(i);  **if** (!file.isEmpty()) {  **try** {  String fileName = file.getOriginalFilename();  File filePath = **new** File(savePath, fileName);  bos = **new** BufferedOutputStream(**new** FileOutputStream(filePath));  bos.write(file.getBytes());  bos.flush();  bos.close();  } **catch** (IOException e) {  e.printStackTrace();  **return "第 "** + i + **" 个文件上传有错误"** + e.getMessage();  } **finally** {  **if** (bos != **null**) {  **try** {  bos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }**else**{  **return "第 "** + i + **" 个文件为空"**;  }  }  }  **return "所有文件上传成功"**;  } } |

**或者:**

|  |
| --- |
| @RequestMapping(**"/multiUploadFile2"**) @ResponseBody **public** String multiUploadFile2(HttpServletResponse response,  HttpServletRequest request, String username, String password) {  System.***out***.println(username + **" : "** + password);  File savePath = **new** File(**"D:\\BaiduNetdiskDownload"**);  **if** (!savePath.exists()) {  savePath.mkdirs();  }  BufferedOutputStream bos = **null**;  StandardMultipartHttpServletRequest req = (StandardMultipartHttpServletRequest) request;  Iterator<String> fileNames = req.getFileNames();  **while** (fileNames.hasNext()){  String fileNameIte = fileNames.next();  **try**{  MultipartFile file = req.getFile(fileNameIte);  **if**(!file.isEmpty()){  String fileName = file.getOriginalFilename();   File filePath = **new** File(savePath,fileName);  bos =**new** BufferedOutputStream(**new** FileOutputStream(filePath));  bos.write(file.getBytes());  bos.flush();  bos.close();  }  }**catch**(IOException e){  e.printStackTrace();  }**finally** {  **if**(bos !=**null**){  **try** {  bos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  **return "所有文件上传成功"**; } |

③页面代码

|  |
| --- |
| <**form method="post" action="/multiUploadFile2" enctype="multipart/form-data"**>  <**input type="file" name="file"**/>  <**input type="file" name="file"**/>  <**br**/>  name: <**input type="text" name="username"**/>  <**br**/>  password: <**input type="password" name="password"**/>  <**input type="submit"**/> </**form**> |

3、文件下载

|  |
| --- |
| *//文件下载相关代码* @RequestMapping(**"/download"**) **public** String downloadFile(HttpServletRequest request, HttpServletResponse response) {  String fileName = **"[www.java1234.com].mp4"**;*// 设置文件名，根据业务需要替换成要下载的文件名* **if** (fileName != **null**) {  *//设置文件路径* String realPath = **"D:\\dowload\\"**;  File file = **new** File(realPath , fileName);  **if** (file.exists()) {  response.setContentType(**"application/force-download"**);*// 设置强制下载不打开* response.addHeader(**"Content-Disposition"**, **"attachment;fileName="** + fileName);*// 设置文件名* **byte**[] buffer = **new byte**[1024];  FileInputStream fis = **null**;  BufferedInputStream bis = **null**;  **try** {  fis = **new** FileInputStream(file);  bis = **new** BufferedInputStream(fis);  OutputStream os = response.getOutputStream();  **int** i = bis.read(buffer);  **while** (i != -1) {  os.write(buffer, 0, i);  i = bis.read(buffer);  }  System.***out***.println(**"success"**);  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (bis != **null**) {  **try** {  bis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** (fis != **null**) {  **try** {  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  }  }  **return null**; } |

第十八篇 使用jasypt框架对SpringBoot配置加密

Jasypt安全框架提供了Spring的集成，主要是实现 PlaceholderConfigurerSupport类或者其子类。 在Sring 3.1之后，则推荐使用PropertySourcesPlaceholderConfigurer类作为属性替换配置类，这里Spring集成Jasypt则使用Jasypt对属性替换配置类的实现。

该工具的优势:

1、该工具支持注解方式开启jasypt功能，以及注解方式引入一个或多个需要处理的配置文件。

2、该工具同时支持properties与yml文件的解析处理。

3、该工具支持自定义加解密类型和复写加解密方法。

默认情况下jasypt采用的算法是PBEWithMD5AndDES，该算法对同一串明文每次加密的密文都不一样，比较适合做数据加解密。但是该算法必须配置密码，我们在yml文件配置如下参数:

|  |
| --- |
| **jasypt:  encryptor:  password:** 123456  **property:  prefix:** C9(  **suffix:** ) |

如果想要改变其他配置例如密文的前后缀也可以在这里配置。

如果上面引入的包不是starter，那么需要在启动类上添加注解@EnableEncryptableProperties以启动该功能。

程序启动可以看到采用jasypt-spring-boot-starter工具来集成spring boot与jasypt的功能成功实现了。我们看再看下配置日志，可以看到默认情况下字符串加密机默认的配置信息：

|  |
| --- |
|  |

通过以上方式使用jasypt框架处理配置文件参数加密的功能就基本上OK了，但是存在一定的风险，那就是程序配置文件中，存在解密密文的密码。因为PBEWithMD5AndDES算法到处都可以找到实现。如果拿到了数据库密文和算法的密码，那么很容易解析出连接数据库的密码。一般严谨的做法是不会将密文信息与解密工具放在一起，避免程序被获取后，加密算法和数据库密码密文以及解密密码都同时被泄露。

1、使用jasypt框架默认加密

①pom.xml文件引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>com.github.ulisesbocchio</**groupId**>  <**artifactId**>jasypt-spring-boot-starter</**artifactId**>  <**version**>1.18</**version**> </**dependency**> |

②配置文件

application.properties

|  |
| --- |
| **datasource.driverClassName**=**com.mysql.jdbc.Driver datasource.url**=**jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=UTF-8 datasource.username**=**ENC(w7gK6pfaAD6rD5Od43x6WA==) datasource.password**=**ENC(AOyTIhh60ox+bU5wcx5a7w==)** |

encryp.properties

|  |
| --- |
| **encryp.username** = **ENC(w7gK6pfaAD6rD5Od43x6WA==) encryp.password** = **ENC(AOyTIhh60ox+bU5wcx5a7w==)** |

③启动类

|  |
| --- |
| @SpringBootApplication @MapperScan(value = **"com.cxp.dao.mapper"**) @EncryptablePropertySource(name = **"EncryptedProperties"**, value = **"classpath:encryp.properties"**) **public class** Springboot3ClassApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(Springboot3ClassApplication.**class**, args);  } } |

④测试类

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** Springboot3ClassApplicationTests {  **static** {  *//设置加载用的密码*  System.*setProperty*(**"jasypt.encryptor.password"**, **"123456"**);  }   @Autowired  StringEncryptor **stringEncryptor**;*//密码解码器自动注入* @Value(**"${datasource.username}"**)  **private** String **dataBaseUserName**;  @Value(**"${datasource.password}"**)  **private** String **dataBasePassword**;   @Value(**"${encryp.username}"**)  **private** String **encrypUserName**;  @Value(**"${encryp.password}"**)  **private** String **encrypPassword**;   @Test  **public void** test1() {  System.***out***.println(**stringEncryptor**.encrypt(**"root"**));  System.***out***.println(**stringEncryptor**.encrypt(**"123456"**));   System.***out***.println(**"数据库用户名"**+**dataBaseUserName**);  System.***out***.println(**"数据库密码"**+**dataBasePassword**);  System.***out***.println(**"自定义配置用户"**+**encrypUserName**);  System.***out***.println(**"自定义配置密码"**+**encrypPassword**);  } } |

⑤输出结果

|  |
| --- |
|  |

2、自定义加解密（模拟远程获取密码）

①pom.xml文件引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>com.github.ulisesbocchio</**groupId**>  <**artifactId**>jasypt-spring-boot-starter</**artifactId**>  <**version**>1.18</**version**> </**dependency**> |

②配置文件

application.properties

|  |
| --- |
| **datasource.driverClassName**=**com.mysql.jdbc.Driver datasource.url**=**jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=UTF-8 datasource.username**=**C9(dUsername) datasource.password**=**C9(dPassword)**  *#定义自定义bean的name*  **jasypt.encryptor.bean**=**c9Encryptor jasypt.encryptor.property.detector-bean**=**C9DefaultPropertyDetector** |

encryp.properties

|  |
| --- |
| **encryp.username** = **C9(tUsername) encryp.password** = **C9(tPassword)** |

③启动类

|  |
| --- |
| @SpringBootApplication @MapperScan(value = **"com.cxp.dao.mapper"**) @EncryptablePropertySource(name = **"EncryptedProperties"**, value = **"classpath:encryp.properties"**) **public class** Springboot3ClassApplication {  **public static void** main(String[] args) {  SpringApplication.*run*(Springboot3ClassApplication.**class**, args);  } } |

④自定义加解密方法

|  |
| --- |
| **import** com.cxp.Util.HttpUtils; **import** org.jasypt.encryption.StringEncryptor;  **import** java.util.HashMap; **import** java.util.Map; **public class** C9Encryptor **implements** StringEncryptor {  */\*\*  \* 加密的方法  \** ***@param message*** *\** ***@return*** *\* /*  @Override  **public** String encrypt(String message) {  System.***out***.println(**"encrypt message:=="**+message);  **if** (**"123456"**.equalsIgnoreCase(message)) {  message = **"123456root"**;  }**else if**(**"root"**.equalsIgnoreCase(message)){  message = **"root123456"**;  }**else**{  message = **"没设置咯!"**;  }  **return** message;  }  */\*\*  \* 解密的方法  \** ***@param encryptedMessage*** *\** ***@return*** *\*/*  @Override  **public** String decrypt(String encryptedMessage) {  System.***out***.println(**"decrypt encryptedMessage:=="**+encryptedMessage);  Map<String, String> map=**new** HashMap<>();  map.put(**"key"**,encryptedMessage);  *//模拟获取从远程服务器获取* String result = HttpUtils.*httpGet*(**"http://localhost:8088/getPassword"**, map);  **return** result;  } } |

⑤自定义需要加解密的Key-Value对的前缀

|  |
| --- |
| **import** com.ulisesbocchio.jasyptspringboot.EncryptablePropertyDetector; **import** org.springframework.util.Assert;  */\*\*  \* Created by ASUS on 2018/3/31.  \*/* **public class** C9DefaultPropertyDetector **implements** EncryptablePropertyDetector {   **private** String **prefix** = **"C9("**;  **private** String **suffix** = **")"**;   **public** C9DefaultPropertyDetector() {  }   **public** C9DefaultPropertyDetector(String prefix, String suffix) {  Assert.*notNull*(prefix, **"Prefix can't be null"**);  Assert.*notNull*(suffix, **"Suffix can't be null"**);  **this**.**prefix** = prefix;  **this**.**suffix** = suffix;  }   @Override  **public boolean** isEncrypted(String property) {  **if** (property == **null**) {  **return false**;  }  **final** String trimmedValue = property.trim();  **return** (trimmedValue.startsWith(**prefix**) &&  trimmedValue.endsWith(**suffix**));  }   @Override  **public** String unwrapEncryptedValue(String property) {  **return** property.substring(  **prefix**.length(),  (property.length() - **suffix**.length()));  } } |

⑥注入spring容器

|  |
| --- |
| **import** com.ulisesbocchio.jasyptspringboot.EncryptablePropertyDetector; **import** org.jasypt.encryption.StringEncryptor; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*\*  \* Created by ASUS on 2018/3/31.  \*/* @Configuration  @Order(1) **public class** EncryptorConfig {  *// @Bean(name = "jasyptStringEncryptor")* @Bean(name = **"c9Encryptor"**)**public** StringEncryptor stringEncryptor() {  C9Encryptor c9Encryptor = **new** C9Encryptor();  System.***out***.println(c9Encryptor);  **return** c9Encryptor;  }   *// @Bean(name="encryptablePropertyDetector")*  @Bean(name=**"C9DefaultPropertyDetector"**)  **public** EncryptablePropertyDetector c9DefaultPropertyDetector(){  C9DefaultPropertyDetector c9DefaultPropertyDetector=**new** C9DefaultPropertyDetector();  **return** c9DefaultPropertyDetector;  } } |

⑦模拟服务器上密码位置

|  |
| --- |
| @RestController **public class** EncryptorController {  @RequestMapping(value = {**"/getPassword"**})  **public** String getPassword(String key){  Map<String,String> map =**new** HashMap<>();  map.put(**"dUsername"**,**"root"**);  map.put(**"dPassword"**,**"123456"**);  map.put(**"tUsername"**,**"guest"**);  map.put(**"tPassword"**,**"guest123456"**);  **return** map.get(key);  } } |

⑧测试类

|  |
| --- |
| @RunWith(SpringRunner.**class**) @SpringBootTest **public class** Springboot3ClassApplicationTests {   @Autowired  **private** TestService **testService**;   @Autowired  **private** LogService **logService**;   @Resource(name=**"jasyptStringEncryptor"**)  StringEncryptor **stringEncryptor**;*//密码解码器自动注入* @Value(**"${datasource.username}"**)  **private** String **dataBaseUserName**;  @Value(**"${datasource.password}"**)  **private** String **dataBasePassword**;   @Value(**"${encryp.username}"**)  **private** String **encrypUserName**;  @Value(**"${encryp.password}"**)  **private** String **encrypPassword**;   @Test  **public void** test1() {  System.***out***.println(**"使用的加解密类 : "**+**stringEncryptor**);  System.***out***.println(**"root加密后: "**+**stringEncryptor**.encrypt(**"root"**));  System.***out***.println(**"123456加密后: "**+**stringEncryptor**.encrypt(**"123456"**));  System.***out***.println(**"abcdef加密后: "**+**stringEncryptor**.encrypt(**"abcdef"**));   System.***out***.println(**"数据库用户名:"**+**dataBaseUserName**);  System.***out***.println(**"数据库密码:"**+**dataBasePassword**);  System.***out***.println(**"自定义配置用户"**+**encrypUserName**);  System.***out***.println(**"自定义配置密码:"**+**encrypPassword**);  } } |

⑨输出结果

|  |
| --- |
|  |

第十九篇 使用Spring Boot Actuator监控应用

1、Actuator监控

微服务的特点决定了功能模块的部署是分布式的，大部分功能模块都是运行在不同的机器上，彼此通过服务调用进行交互，前后台的业务流会经过很多个微服务的处理和传递，出现了异常如何快速定位是哪个环节出现了问题？

在这种框架下，微服务的监控显得尤为重要。本文主要结合Spring Boot Actuator，跟大家一起分享微服务Spring Boot Actuator的常见用法，方便我们在日常中对我们的微服务进行监控治理。

Spring Boot使用“习惯优于配置的理念”，采用包扫描和自动化配置的机制来加载依赖jar中的Spring bean,不需要任何Xml配置，就可以实现Spring的所有配置。虽然这样做能让我们的代码变得非常简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使得我们分析整个应用中资源和实例的各种关系变得非常的困难。

Actuator是Spring Boot提供的对应用系统的自省和监控的集成功能，可以查看应用配置的详细信息，例如自动化配置信息、创建的Spring beans以及一些环境属性等。

Actuator监控只需要添加以下依赖就可以完成：

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-security</**artifactId**> </**dependency**> |

为了保证actuator暴露的监控接口的安全性，需要添加安全控制的依赖spring-boot-start-security依赖，访问应用监控端点时，都需要输入验证信息。Security依赖，可以选择不加，不进行安全管理，但不建议这么做。

2、Actuator 的 REST 接口

Actuator监控分成两类：原生端点和用户自定义端点；自定义端点主要是指扩展性，用户可以根据自己的实际应用，定义一些比较关心的指标，在运行期进行监控。

原生端点是在应用程序里提供众多 Web 接口，通过它们了解应用程序运行时的内部状况。原生端点又可以分成三类：

>>应用配置类：可以查看应用在运行期的静态信息：例如自动配置信息、加载的springbean信息、yml文件配置信息、环境信息、请求映射信息；

>>度量指标类：主要是运行期的动态信息，例如堆栈、请求连、一些健康指标、metrics信息等；

>>操作控制类：主要是指shutdown,用户可以发送一个请求将应用的监控功能关闭。

Actuator 提供了 13 个接口，具体如下表所示：

| **HTTP 方法** | **路径** | **描述** |
| --- | --- | --- |
| GET | /autoconfig | 提供了一份自动配置报告，记录哪些自动配置条件通过了，哪些没通过 |
| GET | /configprops | 描述配置属性(包含默认值)如何注入Bean |
| GET | /beans | 描述应用程序上下文里全部的Bean，以及它们的关系 |
| GET | /dump | 获取线程活动的快照 |
| GET | /env | 获取全部环境属性 |
| GET | /env/{name} | 根据名称获取特定的环境属性值 |
| GET | /health | 报告应用程序的健康指标，这些值由HealthIndicator的实现类提供 |
| GET | /info | 获取应用程序的定制信息，这些信息由info打头的属性提供 |
| GET | /mappings | 描述全部的URI路径，以及它们和控制器(包含Actuator端点)的映射关系 |
| GET | /metrics | 报告各种应用程序度量信息，比如内存用量和HTTP请求计数 |
| GET | /metrics/{name} | 报告指定名称的应用程序度量值 |
| POST | /shutdown | 关闭应用程序，要求endpoints.shutdown.enabled设置为true |
| GET | /trace | 提供基本的HTTP请求跟踪信息(时间戳、HTTP头等) |

3、快速上手

①项目依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**> </**dependencies**> |

②配置文件

|  |
| --- |
| **server:  port:** 8080 **management:  security:  enabled:** false *#关掉安全认证* **port:** 8088 *#管理端口调整成8088* **context-path:** /monitor *#actuator的访问路径* **endpoints:  shutdown:  enabled:** true  **info:  app:  name:** spring-boot-actuator  **version:** 1.0.0 |

management.security.enabled=false默认有一部分信息需要安全验证之后才可以查看，如果去掉这些安全认证，直接设置management.security.enabled=false

management.context-path=/monitor 代表启用单独的url地址来监控Spring Boot应用，为了安全一般都启用独立的端口来访问后端的监控信息

endpoints.shutdown.enabled=true 启用接口关闭Spring Boot

配置完成之后，启动项目就可以继续验证各个监控功能了。

3、命令详解

①autoconfig

Spring Boot的自动配置功能非常便利，但有时候也意味着出问题比较难找出具体的原因。使用 autoconfig 可以在应用运行时查看代码了某个配置在什么条件下生效，或者某个自动配置为什么没有生效。

启动示例项目，访问：<http://localhost:8088/monitor/autoconfig> 返回部分信息如下：

|  |
| --- |
|  |

②configprops

查看配置文件中设置的属性内容，以及一些配置属性的默认值。

启动示例项目，访问：<http://localhost:8088/monitor/configprops> 返回部分信息如下：

|  |
| --- |
|  |

③beans

根据示例就可以看出，展示了bean的别名、类型、是否单例、类的地址、依赖等信息。

启动示例项目，访问：<http://localhost:8088/monitor/beans> 返回部分信息如下：

|  |
| --- |
|  |

④dump

/dump 接口会生成当前线程活动的快照。这个功能非常好，方便我们在日常定位问题的时候查看线程的情况。 主要展示了线程名、线程ID、线程的状态、是否等待锁资源等信息。

启动示例项目，访问：<http://localhost:8088/monitor/dump> 返回部分信息如下：

|  |
| --- |
|  |

⑤env

展示了系统环境变量的配置信息，包括使用的环境变量、JVM 属性、命令行参数、项目使用的jar包等信息。和configprops不同的是，configprops关注于配置信息，env关注运行环境信息。

启动示例项目，访问：<http://localhost:8088/monitor/env> 返回部分信息如下：

|  |
| --- |
|  |

为了避免敏感信息暴露到 /env 里，所有名为password、secret、key(或者名字中最后一段是这些)的属性在 /env 里都会加上“\*”。举个例子，如果有一个属性名字是database.password，那么它在/env中的显示效果是这样的：

"database.password":"\*\*\*\*\*\*"

/env/{name}用法

就是env的扩展 可以获取指定配置信息，比如：<http://localhost:8088/monitor/env/java.vm.version> ,返回：{"java.vm.version":"25.101-b13"}

⑥health

可以看到 HealthEndPoint 给我们提供默认的监控结果，包含 磁盘检测和数据库检测。

启动示例项目，访问：<http://localhost:8088/monitor/health> 返回部分信息，下面的JSON响应是由状态、磁盘空间和db。描述了应用程序的整体健康状态,UP 表明应用程序是健康的。磁盘空间描述总磁盘空间,剩余的磁盘空间和最小阈值。application.properties阈值是可配置的:

|  |
| --- |
|  |

其实看 Spring Boot-actuator 源码，你会发现 HealthEndPoint 提供的信息不仅限于此，org.springframework.boot.actuate.health 包下 你会发现 ElasticsearchHealthIndicator、RedisHealthIndicator、RabbitHealthIndicator 等

⑦info

info就是我们自己配置在配置文件中以Info开头的配置信息，比如我们在示例项目中的配置是：

|  |
| --- |
|  |

⑧mappings

描述全部的URI路径，以及它们和控制器的映射关系

启动示例项目，访问：<http://localhost:8088/monitor/mappings> 返回部分信息如下：

|  |
| --- |
|  |

⑨metrics

最重要的监控内容之一，主要监控了JVM内容使用、GC情况、类加载信息等。

启动示例项目，访问：<http://localhost:8088/monitor/metrics> 返回部分信息如下：

|  |
| --- |
|  |

对 /metrics接口提供的信息进行简单分类如下表：

| **分类** | **前缀** | **报告内容** |
| --- | --- | --- |
| 垃圾收集器 | gc.\* | 已经发生过的垃圾收集次数，以及垃圾收集所耗费的时间，适用于标记-清理垃圾收集器和并行垃圾收集器(数据源自java.lang.management. GarbageCollectorMXBean) |
| 内存 | mem.\* | 分配给应用程序的内存数量和空闲的内存数量(数据源自java.lang. Runtime) |
| 堆 | heap.\* | 当前内存用量(数据源自java.lang.management.MemoryUsage) |
| 类加载器 | classes.\* | JVM类加载器加载与卸载的类的数量(数据源自java.lang. management.ClassLoadingMXBean) |
| 系统 | processors、instance.uptime、uptime、systemload.average | 系统信息，例如处理器数量(数据源自java.lang.Runtime)、运行时间(数据源自java.lang.management.RuntimeMXBean)、平均负载(数据源自java.lang.management.OperatingSystemMXBean) |
| 线程池 | thread.\* | 线程、守护线程的数量，以及JVM启动后的线程数量峰值(数据源自 java.lang .management.ThreadMXBean) |
| 数据源 | datasource.\* | 数据源连接的数量(源自数据源的元数据，仅当Spring应用程序上下文里存在 DataSource Bean 的时候才会有这个信息) |
| Tomcat 会话 | httpsessions.\* | Tomcat的活跃会话数和最大会话数(数据源自嵌入式Tomcat的Bean，仅在使用嵌入式Tomcat服务器运行应用程序时才有这个信息) |
| HTTP | counter.status.、gauge.response. | 多种应用程序服务HTTP请求的度量值与计数器 |

**解释说明：**

请注意，这里的一些度量值，比如数据源和Tomcat会话，仅在应用程序中运行特定组件时才有数据。你还可以注册自己的度量信息。

HTTP的计数器和度量值需要做一点说明。counter.status 后的值是HTTP状态码，随后是所请求的路径。举个例子，counter.status.200.metrics 表明/metrics端点返回 200(OK) 状态码的次数。

HTTP的度量信息在结构上也差不多，却在报告另一类信息。它们全部以gauge.response 开头,，表明这是HTTP响应的度量信息。前缀后是对应的路径。度量值是以毫秒为单位的时间，反映了最近处理该路径请求的耗时。

这里还有几个特殊的值需要注意。root路径指向的是根路径或/。star-star代表了那些Spring 认为是静态资源的路径，包括图片、JavaScript和样式表，其中还包含了那些找不到的资源。这就是为什么你经常会看到 counter.status.404.star-star，这是返回了HTTP 404 (NOT FOUND) 状态的请求数。

/metrics接口会返回所有的可用度量值，但你也可能只对某个值感兴趣。要获取单个值，请求时可以在URL后加上对应的键名。例如，要查看空闲内存大小,可以向/metrics/mem.free发一 个GET请求。例如访问：<http://localhost:8088/monitor/metrics/mem.free> ，返回：{"mem.free":178123}。

⑩shutdown

开启接口优雅关闭Spring Boot应用，要使用这个功能首先需要在配置文件中开启：

|  |
| --- |
| **endpoints:  shutdown:  enabled:** true |

配置完成之后，启动示例项目，POST请求访问：<http://localhost:8088/monitor/shutdown> 返回部分信息如下：

|  |
| --- |
|  |

此时你会发现应用已经被关闭。

⑩trace

/trace 接口能报告所有Web请求的详细信息，包括请求方法、路径、时间戳以及请求和响应的头信息，记录每一次请求的详细信息。

启动示例项目，先访问一次：<http://localhost:8011/testboot1/findProductList> ，再到浏览器执行：<http://localhost:8088/monitor/trace> 查看返回信息：

|  |
| --- |
|  |

⑩其它配置

**敏感信息访问限制**:

根据上面表格，鉴权为false的，表示不敏感，可以随意访问，否则就是做了一些保护，不能随意访问。

endpoints.mappings.sensitive=false

这样需要对每一个都设置，比较麻烦。敏感方法默认是需要用户拥有ACTUATOR角色，因此，也可以设置关闭安全限制：

management.security.enabled=false

或者配合Spring Security做细粒度控制。

**启用和禁用接口:**

虽然Actuator的接口都很有用，但你不一定需要全部这些接口。默认情况下，所有接口(除 了/shutdown)都启用。比如要禁用 /metrics 接口，则可以设置如下：

endpoints.metrics.enabled = false

如果你只想打开一两个接口，那就先禁用全部接口，然后启用那几个你要的，这样更方便。

endpoints.enabled = false

endpoints.metrics.enabled = true

第二十篇 springboot动态切换数据源

SpringBoot中使用动态数据源可以实现分布式中的分库技术，比如查询用户 就在用户库中查询，查询订单 就在订单库中查询。

1、使用aop进行动态切换数据源

①pom.xml文件引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-aop</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.2</**version**> </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.46</**version**> </**dependency**>  <**dependency**>  <**groupId**>com.oracle</**groupId**>  <**artifactId**>ojdbc6</**artifactId**>  <**version**>11.2.0.1.0</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.postgresql</**groupId**>  <**artifactId**>postgresql</**artifactId**>  <**version**>42.2.2</**version**> </**dependency**> |

②application.yml配置文件

|  |
| --- |
| **spring:  application:  name:** boot\_dynamicDataSource  **datasource:  druid:  mysql :  url:** jdbc:mysql://localhost:3306/test?useUnicode=true&amp;characterEncoding=UTF-8  **username:** root  **password:** 123456  **driver-class-name:** com.mysql.jdbc.Driver  **oracle:  url:** jdbc:oracle:thin:@localhost:1521:ORCLASUS  **username:** root  **password:** 123456  **driver-class-name:** oracle.jdbc.driver.OracleDriver  **postgresql:  url:** jdbc:postgresql://192.168.0.103:5432/cheng  **username:** postgres  **password:** 123456  **driver-class-name:** org.postgresql.Driver  **aop:  proxy-target-class:** true |

③数据源配置Config类

|  |
| --- |
| **import** com.alibaba.druid.pool.DruidDataSource; **import** org.apache.ibatis.session.SqlSessionFactory; **import** org.mybatis.spring.SqlSessionFactoryBean; **import** org.mybatis.spring.SqlSessionTemplate; **import** org.springframework.boot.context.properties.ConfigurationProperties; **import** org.springframework.boot.jdbc.DataSourceBuilder; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.core.io.support.PathMatchingResourcePatternResolver; **import** org.springframework.jdbc.datasource.DataSourceTransactionManager; **import** org.springframework.transaction.PlatformTransactionManager;  **import** javax.sql.DataSource; **import** java.util.HashMap; **import** java.util.Map;  */\*\*  \* 动态数据源配置  \*/* @Configuration **public class** DynamicDataSourceConfig {  @Bean(name = **"mysqlDataSource"**)  @ConfigurationProperties(prefix = **"spring.datasource.druid.mysql"**)  **public** DataSource mysqlDataSource() {  DruidDataSource dataSource = DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  dataSource.setName(**"mysqlDataSource"**);  **return** dataSource;  }   @Bean(name = **"oracleDataSource"**)  @ConfigurationProperties(prefix = **"spring.datasource.druid.oracle"**)  **public** DataSource oracleDataSource() {  DruidDataSource dataSource = DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  dataSource.setName(**"oracleDataSource"**);  **return** dataSource;  }   @Bean(name = **"postgresqlDataSource"**)  @ConfigurationProperties(prefix = **"spring.datasource.druid.postgresql"**)  **public** DataSource postgresqlDataSource() {  DruidDataSource dataSource = DataSourceBuilder.*create*().type(DruidDataSource.**class**).build();  dataSource.setName(**"postgresqlDataSource"**);  **return** dataSource;  }   */\*\*  \* 动态数据源: 通过AOP在不同数据源之间动态切换  \*  \** ***@return*** *\*/* @Bean(name = **"dynamicDataSource"**)  **public** DataSource dataSource() {  DynamicDataSource dynamicDataSource = **new** DynamicDataSource();  *// 默认数据源* dynamicDataSource.setDefaultTargetDataSource(mysqlDataSource());   *// 配置多数据源* Map<Object, Object> dsMap = **new** HashMap(3);  dsMap.put(**"mysqlDataSource"**, mysqlDataSource());  dsMap.put(**"oracleDataSource"**, oracleDataSource());  dsMap.put(**"postgresqlDataSource"**, postgresqlDataSource());  dynamicDataSource.setTargetDataSources(dsMap);  **return** dynamicDataSource;  }   @Bean  **public** PlatformTransactionManager txManager(DataSource dynamicDataSource) {  **return new** DataSourceTransactionManager(dynamicDataSource);  }@Bean  **public** SqlSessionFactory sqlSessionFactory(DataSource dynamicDataSource) **throws** Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = **new** SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(dynamicDataSource);  PathMatchingResourcePatternResolver resolver = **new** PathMatchingResourcePatternResolver();  sqlSessionFactoryBean.setMapperLocations(resolver.getResources(**"classpath:/mybatis/\*.xml"**));  **return** sqlSessionFactoryBean.getObject();  }   @Bean  **public** SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory sqlSessionFactory) **throws** Exception {  SqlSessionTemplate template = **new** SqlSessionTemplate(sqlSessionFactory); *// 使用上面配置的Factory* **return** template;  } } |

④继承AbstractRoutingDataSource实现路由切换

|  |
| --- |
| **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory; **import** org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;  */\*\*  \* 继承Spring AbstractRoutingDataSource实现路由切换  \* Created by ASUS on 2018/4/17.  \*/* **public class** DynamicDataSource **extends** AbstractRoutingDataSource {   **private static final** Logger ***log*** = LoggerFactory.*getLogger*(DynamicDataSource.**class**);   @Override  **protected** Object determineCurrentLookupKey() {  ***log***.info(**"数据源为:===={}"**, DataSourceContextHolder.*getDB*());  **return** DataSourceContextHolder.*getDB*();  }  } |

⑤动态数据源适配器

|  |
| --- |
| **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory;  */\*\*  \* Created by ASUS on 2018/4/17.  \*/* **public class** DataSourceContextHolder {  **public static final** Logger ***log*** = LoggerFactory.*getLogger*(DataSourceContextHolder.**class**);  **private static final** ThreadLocal<String> ***contextHolder*** = **new** ThreadLocal<>();   *// 设置数据源名* **public static void** setDB(String dbType) {  ***log***.info(**"切换到{}数据源"**, dbType);  ***contextHolder***.set(dbType);  }   *// 获取数据源名* **public static** String getDB() {  **return** (***contextHolder***.get());  }   *// 清除数据源名* **public static void** clearDB() {  ***log***.info(**"清除数据源 {}"**, ***contextHolder***.get());  ***contextHolder***.remove();  } } |

⑥使用aop自定义注解，实现动态切换数据源

|  |
| --- |
| **import** org.aspectj.lang.ProceedingJoinPoint; **import** org.aspectj.lang.annotation.Around; **import** org.aspectj.lang.annotation.Aspect; **import** org.aspectj.lang.reflect.MethodSignature; **import** org.springframework.core.annotation.Order; **import** org.springframework.stereotype.Component; **import** java.lang.reflect.Method;  */\*\*  \* Created by ASUS on 2018/4/17.  \*/* @Aspect *//保证该AOP在@Transactional之前执行* @Order(-1) @Component **public class** DynamicDataSourceAspect {   @Around(**"@annotation(DS)"**)  **public** Object switchDS(ProceedingJoinPoint point) **throws** Throwable {  Class<?> className = point.getTarget().getClass();  String dataSource = **"mysqlDataSource"**;   **if** (className.isAnnotationPresent(DS.**class**)) {  DS ds = className.getAnnotation(DS.**class**);  dataSource = ds.value();  } **else** {  *// 得到访问的方法对象* String methodName = point.getSignature().getName();  Class[] argClass = ((MethodSignature) point.getSignature()).getParameterTypes();  Method method = className.getMethod(methodName, argClass);  *// 判断是否存在@DS注解* **if** (method.isAnnotationPresent(DS.**class**)) {  DS annotation = method.getAnnotation(DS.**class**);  *// 取出注解中的数据源名* dataSource = annotation.value();  }  }   *// 切换数据源* DataSourceContextHolder.*setDB*(dataSource);   **try** {  **return** point.proceed();  } **finally** {  DataSourceContextHolder.*clearDB*();  }  } } |

⑦自定义注解

|  |
| --- |
| **import** java.lang.annotation.ElementType; **import** java.lang.annotation.Retention; **import** java.lang.annotation.RetentionPolicy; **import** java.lang.annotation.Target;  */\*\*  \* 数据源注解  \** ***@author*** *Lucian  \*/* @Retention(RetentionPolicy.***RUNTIME***) @Target({ElementType.***TYPE***, ElementType.***METHOD***}) **public** @**interface** DS {  String value() **default "mysqlDataSource"**; } |

⑧mapper类

|  |
| --- |
| **import** org.apache.ibatis.annotations.Mapper; **import** org.apache.ibatis.annotations.ResultType; **import** org.apache.ibatis.annotations.Select;  **import** java.util.List; **import** java.util.Map;  */\*\*  \* Created by ASUS on 2018/4/18.  \*/* @Mapper **public interface** UserInfoMapper {   @Select(**"select \* from t\_user\_info where userum=#{userUm} "**)  @ResultType(value = Map.**class**)  **public** List<Map<String,Object>> findOracleUserInfoByCondition(Map<String,Object> param);   @Select(**"select \* from t\_hub\_user\_recored\_info where userum=#{userUm} "**)  @ResultType(value = Map.**class**)  **public** List<Map<String,Object>> findPostgresql(Map<String,Object> param);   @Select(**"select \* from product where pname=#{pname} "**)  @ResultType(value = Map.**class**)  **public** List<Map<String, Object>> findMysqlProductList(Map<String, Object> map); } |

⑨Service层

|  |
| --- |
| **import** java.util.List; **import** java.util.Map;  */\*\*  \* Created by ASUS on 2018/4/18.  \*/* **public interface** UserInfoService {   **public** List<Map<String,Object>> findOracleUserInfoByCondition(Map<String,Object> param);   **public** List<Map<String,Object>> findPostgresql(Map<String,Object> param);   **public** List<Map<String, Object>> findMysqlProductList(Map<String, Object> map); } |
|  |
| **import** com.cxp.dynamicDateSource.DS; **import** com.cxp.mapper.UserInfoMapper; **import** com.cxp.service.UserInfoService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service;  **import** java.util.List; **import** java.util.Map;  */\*\*  \* Created by ASUS on 2018/4/18.  \*/* @Service **public class** UserInfoServiceImpl **implements** UserInfoService {   @Autowired  **private** UserInfoMapper **userInfoMapper**;   @DS(value = **"oracleDataSource"**)  @Override  **public** List<Map<String, Object>> findOracleUserInfoByCondition(Map<String, Object> param) {  **return userInfoMapper**.findOracleUserInfoByCondition(param);  }   @DS(value = **"postgresqlDataSource"**)  @Override  **public** List<Map<String, Object>> findPostgresql(Map<String, Object> param) {  **return userInfoMapper**.findPostgresql(param);  }   @DS(value = **"mysqlDataSource"**)  @Override  **public** List<Map<String, Object>> findMysqlProductList(Map<String, Object> map) {  **return userInfoMapper**.findMysqlProductList(map);  } } |

⑩Controller层

|  |
| --- |
| **import** com.cxp.service.UserInfoService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.web.bind.annotation.RequestMapping; **import** org.springframework.web.bind.annotation.RestController;  **import** java.util.Date; **import** java.util.HashMap; **import** java.util.List; **import** java.util.Map;  */\*\*  \* Created by ASUS on 2018/4/12.  \*/* @RestController **public class** UserController {   @Autowired  **private** UserInfoService **userInfoService**;   @RequestMapping(**"/findOracleUserInfoByCondition"**)  **public** List<Map<String, Object>> findOracleUserInfoByCondition(String userUm) {  Map<String, Object> param=**new** HashMap<>();  param.put(**"userUm"**,userUm);  **return userInfoService**.findOracleUserInfoByCondition(param);  }   @RequestMapping(**"/findPostgresql"**)  **public** List<Map<String, Object>> findPostgresql(String userUm) {  Map<String, Object> param=**new** HashMap<>();  param.put(**"userUm"**,userUm);  **return userInfoService**.findPostgresql(param);  }   @RequestMapping(**"/findMysqlProductList"**)  **public** List<Map<String, Object>> findMysqlProductList(String pname) {  Map<String, Object> param=**new** HashMap<>();  param.put(**"pname"**,pname);  **return userInfoService**.findMysqlProductList(param);  } } |

⑾、测试

访问mysql对应 ：

<http://localhost:8080/findMysqlProductList?pname=康佳电视机16>

|  |
| --- |
|  |

访问oracle库对应:

<http://localhost:8080/findOracleUserInfoByCondition?userUm=LISHI110>

|  |
| --- |
|  |

访问postgresql库对应:

<http://localhost:8080/findPostgresql?userUm=LISHI110>

|  |
| --- |
|  |

第二十一篇 使用事务

SpringBoot提供了非常方便的事务操作，通过注解就可以实现事务的回滚，非常方便快捷，下面我们就说一下如何进行事务操作。

在Spring中，事务有两种实现方式，分别是编程式事务管理和声明式事务管理两种方式。

编程式事务管理： 编程式事务管理使用TransactionTemplate或者直接使用底层的PlatformTransactionManager。对于编程式事务管理，spring推荐使用TransactionTemplate。

声明式事务管理： 建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务管理不需要入侵代码，通过@Transactional就可以进行事务操作，更快捷而且简单。推荐使用

@Transactional可以作用于接口、接口方法、类以及类方法上。当作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。因此可以在Service层和Controller层使用。

常用配置:

|  |  |
| --- | --- |
| 参 数 名 称 | 功 能 描 述 |
| readOnly | 该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。例如：@Transactional(readOnly=true) |
| rollbackFor | 该属性用于设置需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，则进行事务回滚。例如：指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class}) |
| rollbackForClassName | 该属性用于设置需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，则进行事务回滚。例如：指定单一异常类名称@Transactional(rollbackForClassName=”RuntimeException”)指定多个异常类名称：@Transactional(rollbackForClassName={“RuntimeException”,”Exception”}) |
| noRollbackFor | 该属性用于设置不需要进行回滚的异常类数组，当方法中抛出指定异常数组中的异常时，不进行事务回滚。例如：指定单一异常类：@Transactional(noRollbackFor=RuntimeException.class)指定多个异常类：@Transactional(noRollbackFor={RuntimeException.class, Exception.class}) |
| noRollbackForClassName | 该属性用于设置不需要进行回滚的异常类名称数组，当方法中抛出指定异常名称数组中的异常时，不进行事务回滚。例如：指定单一异常类名称：@Transactional(noRollbackForClassName=”RuntimeException”)指定多个异常类名称：@Transactional(noRollbackForClassName={“RuntimeException”,”Exception”}) |
| propagation | 该属性用于设置事务的传播行为。例如：@Transactional(propagation=Propagation.NOT\_SUPPORTED,readOnly=true) |
| isolation | 该属性用于设置底层数据库的事务隔离级别，事务隔离级别用于处理多事务并发的情况，通常使用数据库的默认隔离级别即可，基本不需要进行设置 |
| timeout | 该属性用于设置事务的超时秒数，默认值为-1表示永不超时 |

第二十二篇 使用Thymeleaf模板引擎

第二十三篇 使用SpringSecurity让SpringBoot项目更安全

SpringSecurity是专门针对基于Spring项目的安全框架，充分利用了依赖注入和AOP来实现安全管控。在很多大型企业级系统中权限是最核心的部分，一个系统的好与坏全都在于权限管控是否灵活，是否颗粒化。在早期的SpringSecurity版本中我们需要大量的xml来进行配置，而基于SpringBoot整合SpringSecurity框架相对而言简直是重生了，简单到不可思议的地步。

SpringSecurity框架有两个概念认证和授权，认证可以访问系统的用户，而授权则是用户可以访问的资源，下面我们来简单讲解下SpringBoot对SpringSecurity安全框架的支持。

**①构建项目**

加入JPA、Security、Druid、MySQL等依赖。

|  |
| --- |
| **pom.xml文件** |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-jpa</**artifactId**> </**dependency**> <**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**> *<!--引入druid最新maven依赖-->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.0.29</**version**> </**dependency**> <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**scope**>runtime</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-tomcat</**artifactId**>  *<!--<scope>provided</scope>-->* </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-test</**artifactId**>  <**scope**>test</**scope**> </**dependency**> *<!-- spring boot tomcat jsp 支持开启 -->* <**dependency**>  <**groupId**>org.apache.tomcat.embed</**groupId**>  <**artifactId**>tomcat-embed-jasper</**artifactId**> </**dependency**> *<!--servlet支持开启-->* <**dependency**>  <**groupId**>javax.servlet</**groupId**>  <**artifactId**>javax.servlet-api</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.security</**groupId**>  <**artifactId**>spring-security-taglibs</**artifactId**>  <**version**>4.2.1.RELEASE</**version**> </**dependency**> |
|  |

**②配置数据库访问配置**

|  |
| --- |
| application.yml配置文件 |
| **spring:  datasource:  type:** com.alibaba.druid.pool.DruidDataSource  **driver-class-name:** com.mysql.jdbc.Driver  **url:** jdbc:mysql://127.0.0.1:3306/test?characterEncoding=utf8  **username:** root  **password:** 123456  *#最大活跃数* **maxActive:** 20  *#初始化数量* **initialSize:** 1  *#最大连接等待超时时间* **maxWait:** 60000  *#打开PSCache，并且指定每个连接PSCache的大小* **poolPreparedStatements:** true  **maxPoolPreparedStatementPerConnectionSize:** 20  *#通过connectionProperties属性来打开mergeSql功能；慢SQL记录  #connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000* **minIdle:** 1  **timeBetweenEvictionRunsMillis:** 60000  **minEvictableIdleTimeMillis:** 300000  **validationQuery:** select 1 from dual  **testWhileIdle:** true  **testOnBorrow:** false  **testOnReturn:** false  *#配置监控统计拦截的filters，去掉后监控界面sql将无法统计,'wall'用于防火墙* **filters:** stat, wall, log4j  **jpa:  properties:  hibernate:  show\_sql:** true  **format\_sql:** true  **mvc:  view:  prefix:** /WEB-INF/jsp/  **suffix:** .jsp |

③

第二十五篇 SpringBoot整合RabbitMQ

[安装RabbitMQ教程](RabbitMQ安装教程.docx)(点击超链接)

1、简单使用

①pom.xml文件依赖

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

②配置rabbitmq的安装地址、端口以及账户信息

|  |
| --- |
| **spring.application.name**=**spring-boot-rabbitmq spring.rabbitmq.addresses**=**192.168.0.134 spring.rabbitmq.port**=**5672 spring.rabbitmq.username**=**admin spring.rabbitmq.password**=**admin** |

③队列配置

|  |
| --- |
| **import** org.springframework.amqp.core.Queue; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*\*  \* Created by ASUS on 2018/4/15.  \*/* @Configuration **public class** RabbitConfig {   @Bean  **public** Queue Queue() {  **return new** Queue(**"hello"**);  }  } |

④发送者

|  |
| --- |
| **import** org.springframework.amqp.core.AmqpTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Component;  **import** java.util.Date;  */\*\*  \* Created by ASUS on 2018/4/15.  \*/* @Component **public class** HelloSender {  @Autowired  **private** AmqpTemplate **rabbitTemplate**;   **public void** send() {  String context = **"hello "** + **new** Date();  System.***out***.println(**"Sender : "** + context);  **for**(**int** i=0;i<10;i++){  **this**.**rabbitTemplate**.convertAndSend(**"hello"**, context);  }  } } |

⑤接收者

|  |
| --- |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/15.  \*/* @Component @RabbitListener(queues = **"hello"**) **public class** HelloReceiver {   @RabbitHandler  **public void** process(String hello) {  System.***out***.println(**"Receiver : "** + hello);  } } |

⑥测试

|  |
| --- |
| **import** com.cxp.helloworld.HelloSender; **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** RabbitmqApplicationTests {   @Autowired  **private** HelloSender **helloSender**;   @Test  **public void** hello() **throws** Exception {  **helloSender**.send();  } } |
|  |

2、多对多使用

一个发送者，N个接收者或者N个发送者和N个接收者会出现什么情况呢？

①一对多发送

对上面的代码进行了小改造,接收端注册了两个Receiver,Receiver1和Receiver2，发送端加入参数计数，接收端打印接收到的参数，下面是测试代码，发送十条消息，来观察两个接收端的执行效果：

|  |
| --- |
|  |

根据返回结果得到以下结论：

一个发送者，N个接受者,经过测试会均匀的将消息发送到N个接收者中

②多对多发送

复制了一份发送者，加入标记，在一百个循环中相互交替发送。

|  |
| --- |
|  |

结论：和一对多一样，接收端仍然会均匀接收到消息

3、高级使用

①对象的支持

springboot以及完美的支持对象的发送和接收，不需要格外的配置。

**发送者：**

|  |
| --- |
| **import** com.cxp.pojo.UserInfo; **import** org.springframework.amqp.core.AmqpTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Component;  **import** java.util.Date;  */\*\*  \* Created by ASUS on 2018/4/15.  \*/* @Component **public class** HelloSender {  @Autowired  **private** AmqpTemplate **rabbitTemplate**;   **public void** sendUser() {  UserInfo userInfo=**new** UserInfo();  userInfo.setId(1);  userInfo.setName(**"张三丰"**);  userInfo.setBirthday(**new** Date());  System.***out***.println(**"Sender : "** + userInfo.toString());  **for**(**int** i=0;i<2;i++){  **this**.**rabbitTemplate**.convertAndSend(**"UserQueue"**, userInfo);  }  } } |

**接收者:**

|  |
| --- |
| **import** com.cxp.pojo.UserInfo; **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/15.  \*/* @Component @RabbitListener(queues = **"UserQueue"**) **public class** UserReceiver2 {   @RabbitHandler  **public void** receiverUser(UserInfo userInfo) {  System.***out***.println(**"receiverUser2 : "** + userInfo);  } } |

**测试:**

|  |
| --- |
| **import** com.cxp.helloworld.HelloSender; **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** RabbitmqApplicationTests {   @Autowired  **private** HelloSender **helloSender**;   @Test  **public void** testUser() **throws** Exception {  **helloSender**.sendUser();  } } |
| **测试结果：** |
|  |

4、Topic Exchange

topic 是RabbitMQ中最灵活的一种方式，可以根据routing\_key自由的绑定不同的队列。

首先对topic规则配置，这里使用两个队列来测试：

pom.xml和配置文件配置rabbitmq服务器略。

①topic规则队列配置

使用queueMessage2同时匹配两个队列，queueMessage1只匹配”topic.message”队列

|  |
| --- |
| **import** org.springframework.amqp.core.Binding; **import** org.springframework.amqp.core.BindingBuilder; **import** org.springframework.amqp.core.TopicExchange; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration; **import** org.springframework.amqp.core.Queue;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Configuration **public class** TopicRabbitConfig {  @Bean  **public** Queue queueMessage1() {  **return new** Queue(**"topic.message1"**);  }   @Bean  **public** Queue queueMessage2() {  **return new** Queue(**"topic.message2"**);  }   @Bean  TopicExchange exchange() {  **return new** TopicExchange(**"topicExchange"**);  }   @Bean  Binding bindingExchangeMessage(Queue queueMessage1, TopicExchange exchange) {  System.***out***.println(**"bindingExchangeMessage"**+queueMessage1+**" === "**+exchange);  **return** BindingBuilder.*bind*(queueMessage1).to(exchange).with(**"topic.message1"**);  }   @Bean  Binding bindingExchangeMessages(Queue queueMessage2, TopicExchange exchange) {  System.***out***.println(**"bindingExchangeMessages"**+queueMessage2+**" === "**+exchange);  **return** BindingBuilder.*bind*(queueMessage2).to(exchange).with(**"topic.#"**);  } } |

②发送者

|  |
| --- |
| **import** org.springframework.amqp.core.AmqpTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component **public class** TopicSender {  @Autowired  **private** AmqpTemplate **rabbitTemplate**;   **public void** send1() {  String context = **"hi, i am message 1"**;  System.***out***.println(**"Sender : "** + context);  **this**.**rabbitTemplate**.convertAndSend(**"topicExchange"**, **"topic.message1"**, context);  }   **public void** send2() {  String context = **"hi, i am messages 2"**;  System.***out***.println(**"Sender : "** + context);  **this**.**rabbitTemplate**.convertAndSend(**"topicExchange"**, **"topic.message2"**, context);  } } |

③二个接收者

|  |
| --- |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component @RabbitListener(queues = **"topic.message1"**) **public class** TopicReceiver1 {   @RabbitHandler  **public void** TopicReceiver1(String message){  System.***out***.println(**"TopicReceiver1 : "** + message);  } } |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component @RabbitListener(queues = **"topic.message2"**) **public class** TopicReceiver2 {   @RabbitHandler  **public void** TopicReceiver2(String message){  System.***out***.println(**"TopicReceiver2 : "** + message);  } } |

④测试

|  |
| --- |
| **import** com.cxp.topicExchange.TopicSender; **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** TopicTests {   @Autowired  **private** TopicSender **sender**;   @Test  **public void** topic1() **throws** Exception {  **sender**.send1();  }   @Test  **public void** topic2() **throws** Exception {  **sender**.send2();  }  } |
|  |

发送send1会匹配到topic.#和topic.message 两个Receiver都可以收到消息，发送send2只有topic.#可以匹配所有只有Receiver2监听到消息。

5、Fanout Exchange

Fanout 就是我们熟悉的广播模式或者订阅模式，给Fanout交换机发送消息，绑定了这个交换机的所有队列都收到这个消息。

①Fanout 相关队列配置

|  |
| --- |
| **import** org.springframework.amqp.core.\*; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Configuration **public class** FanoutRabbitConfig {  @Bean  **public** Queue AMessage() {  **return new** Queue(**"fanout.A"**);  }   @Bean  **public** Queue BMessage() {  **return new** Queue(**"fanout.B"**);  }   @Bean  **public** Queue CMessage() {  **return new** Queue(**"fanout.C"**);  }   @Bean  FanoutExchange fanoutExchange() {  **return new** FanoutExchange(**"fanoutExchange"**);  }   @Bean  Binding bindingExchangeA(Queue AMessage, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(AMessage).to(fanoutExchange);  }   @Bean  Binding bindingExchangeB(Queue BMessage, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(BMessage).to(fanoutExchange);  }   @Bean  Binding bindingExchangeC(Queue CMessage, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(CMessage).to(fanoutExchange);  }  } |

②发送者

|  |
| --- |
| **import** org.springframework.amqp.core.AmqpTemplate; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component **public class** FanoutSender {  @Autowired  **private** AmqpTemplate **rabbitTemplate**;   **public void** send() {  String context = **"hi, fanout msg "**;  System.***out***.println(**"Sender : "** + context);  **this**.**rabbitTemplate**.convertAndSend(**"fanoutExchange"**,**""**, context);  } } |

③三个接收者

这里使用了A、B、C三个队列绑定到Fanout交换机上面，发送端的routing\_key写任何字符都会被忽略：

|  |
| --- |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component @RabbitListener(queues = **"fanout.A"**) **public class** FanoutReceiverA {   @RabbitHandler  **public void** process(String message) {  System.***out***.println(**"fanout Receiver A : "** + message);  } } |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component @RabbitListener(queues = **"fanout.B"**) **public class** FanoutReceiverB {   @RabbitHandler  **public void** process(String message) {  System.***out***.println(**"fanout Receiver B : "** + message);  } } |
| **import** org.springframework.amqp.rabbit.annotation.RabbitHandler; **import** org.springframework.amqp.rabbit.annotation.RabbitListener; **import** org.springframework.stereotype.Component;  */\*\*  \* Created by ASUS on 2018/4/16.  \*/* @Component @RabbitListener(queues = **"fanout.C"**) **public class** FanoutReceiverC {   @RabbitHandler  **public void** process(String message) {  System.***out***.println(**"fanout Receiver C : "** + message);  } } |

④测试

|  |
| --- |
| **import** com.cxp.FanoutExchange.FanoutSender; **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** FanoutTests {   @Autowired  **private** FanoutSender **sender**;   @Test  **public void** topic() **throws** Exception {  **sender**.send();  } } |
|  |

结果说明，绑定到fanout交换机上面的队列都收到了消息。

技巧一

Idea中SpringBoot设置热部署html修改无需重启项目：

在pom中直接引入依赖

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

设置以下两项（第一项如已设置直接设置第二项）

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

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