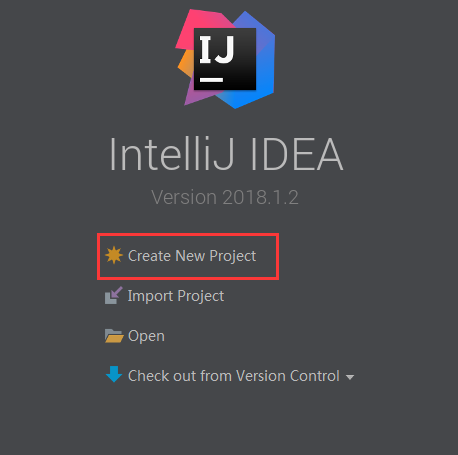
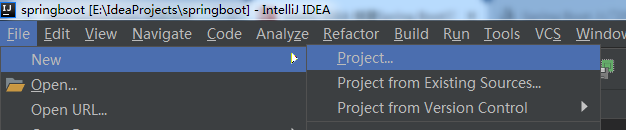
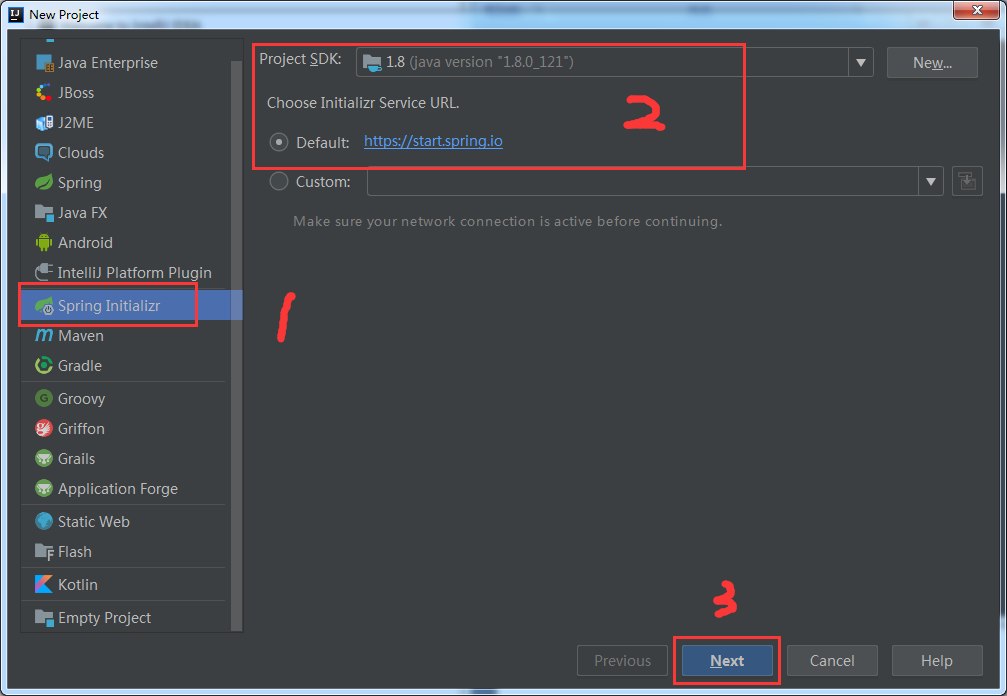
使用idea开发springboot项目

1. 创建springboot项目，第一次启动可在首页选择创建项目。

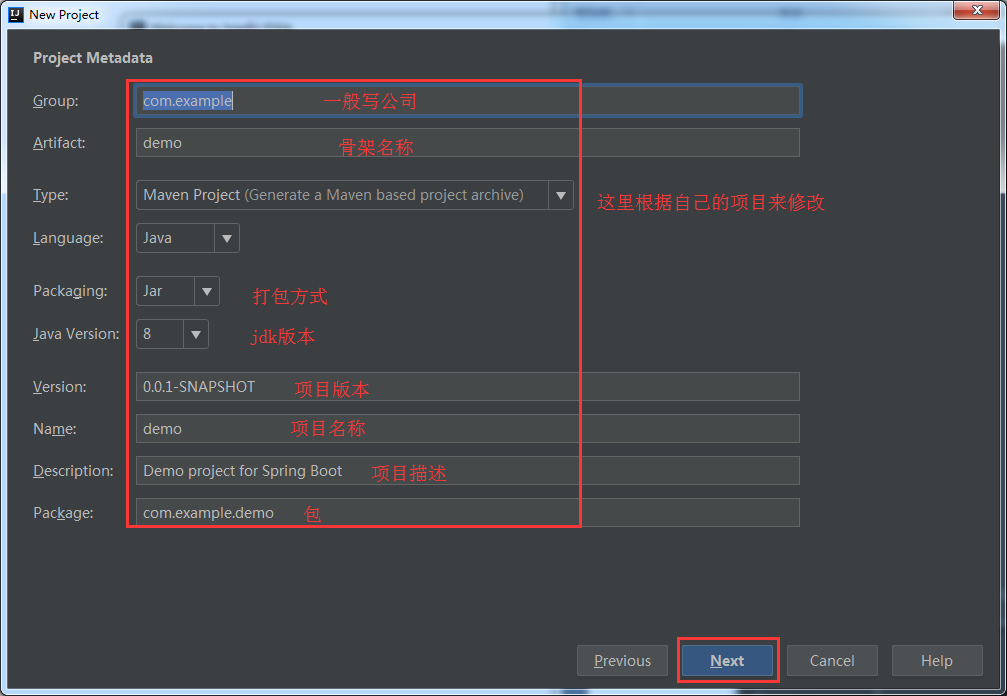
也可在菜单中选择来创建：File🡪New🡪Project



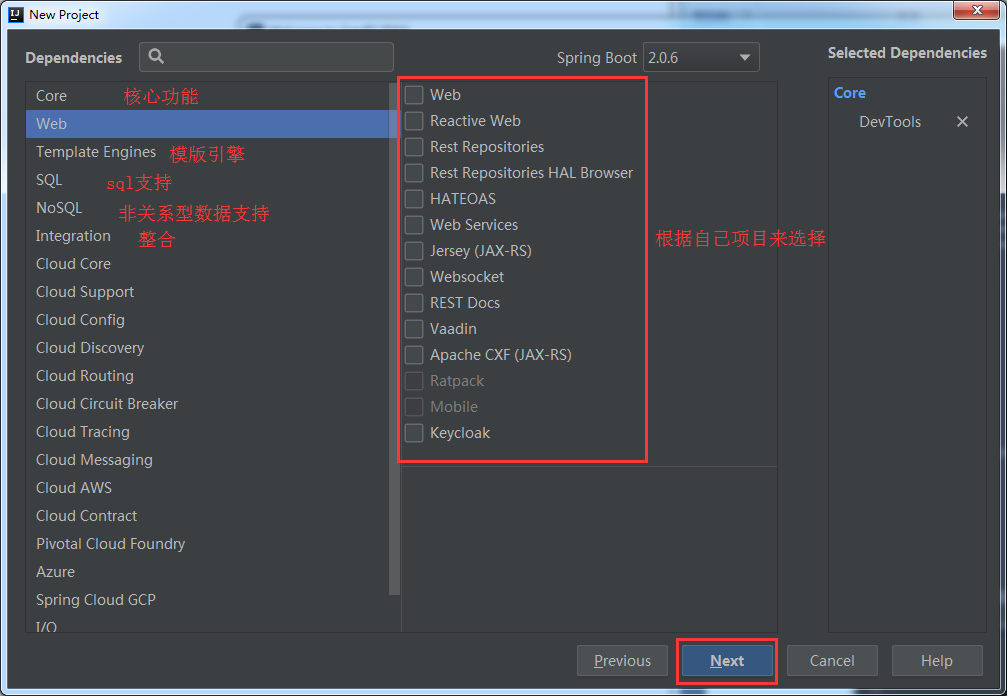
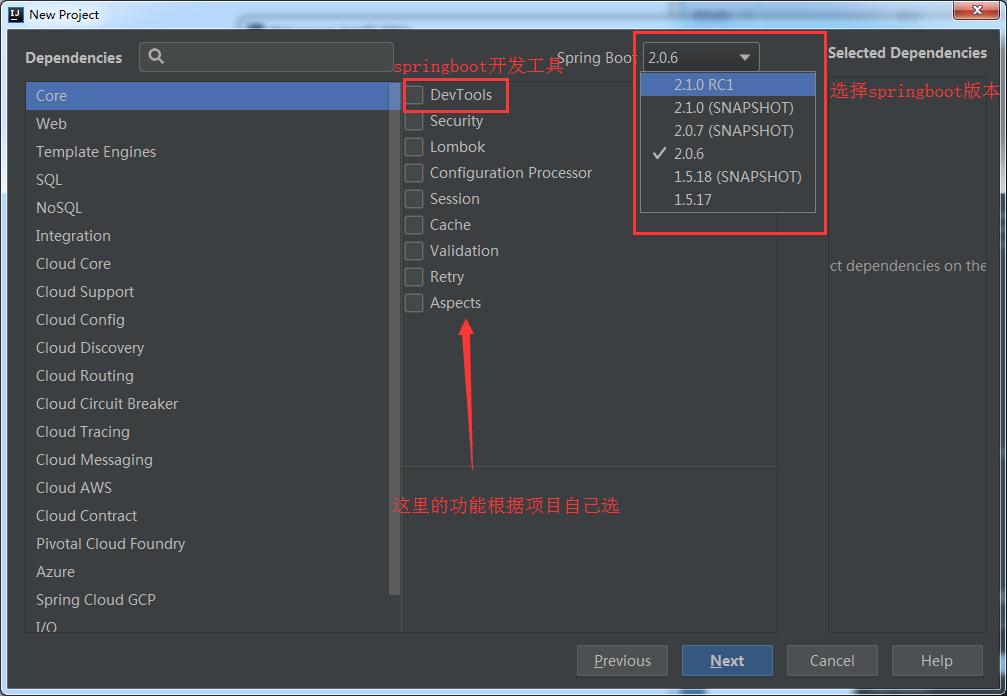
选择Spring Initializr，在右边选择jdk版本，点击下一步



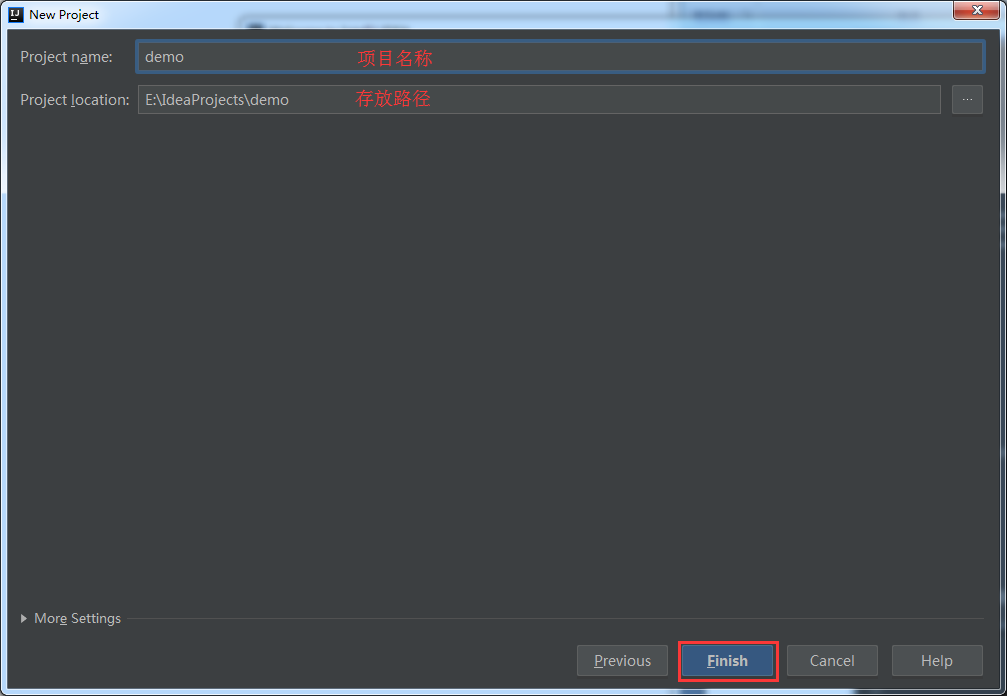
根据自己项目填写



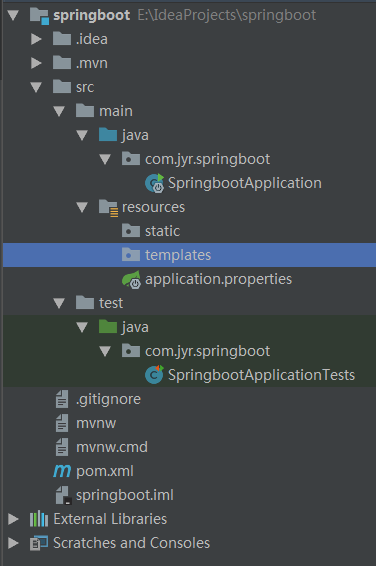
选择springboot版本和工具支持：如开发工具、web、sql等



填定项目名称、选择存放路径、点击完成。项目创建成功

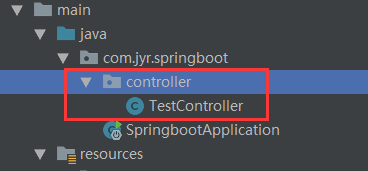


建完的项目结构：



1. 创建控制器

新建包controller，在该包下创建类TestController

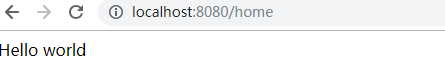


TestController类代码：

import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.ResponseBody;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class TestController {  
  
 @RequestMapping("/home")  
 @ResponseBody  
 public String home(){  
 System.*out*.println("我的第一个springboot应用");  
 return "Hello world";  
 }  
}

运行SpringbootApplication主程序，控制台输入出Tomcat started on port(s): 8080 (http) with context path ''表示运行成功

在浏览器访问localhost:8080/home



1. 热部署（使用eclipse有效，idea不起作用）

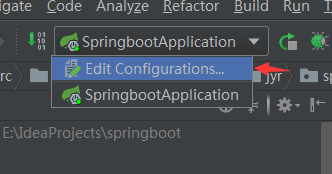
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>

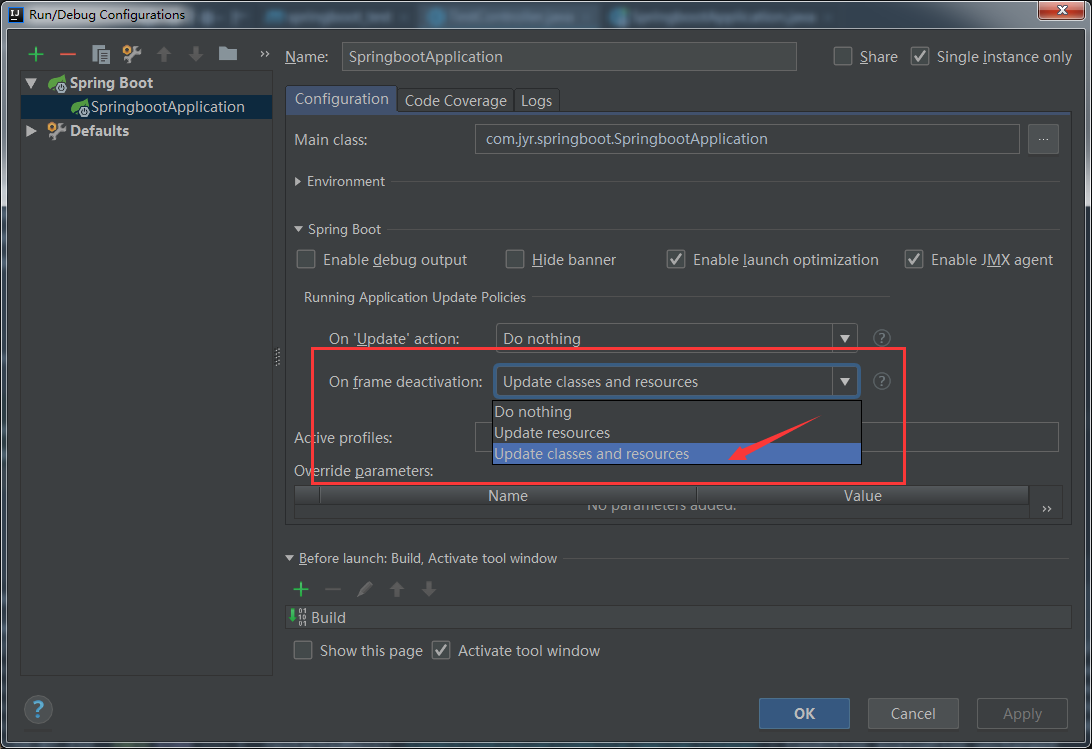
<optional>true</optional>  
 <scope>true</scope>  
</dependency>

热部署生效

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <configuration>  
 <!-- 没有该配置，devtools 不生效 -->  
 <fork>true</fork>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

Idea中设置执部署：





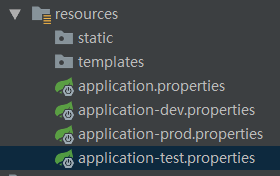
1. 多环境切换

在resources目录下创建三个文件

application-dev.properties：用于开发环境

application-test.properties：用于测试环境

application-prod.properties：用于生产环境



在 application.properties 中配置：spring.profiles.active=dev

Spring.profiles.active的值可为dev、test或者prod，代表使用不同的环境

如开发使用8080端口，测试使用8082端口，生产使用8083端口

Server.port = 8080

application.properties配置文件可放置三种环境共有的配置信息，如数据库连接信息等

1. 日志文件

默认加载classpath路径下的logback-spring.xml 或者 classpath:logback-spring.groovy文件

其内容：

|  |
| --- |
| 1. **<?**xml version="1.0" encoding="UTF-8"**?>** 2. <configuration> 3. *<!-- 文件输出格式 -->* 4. <property name="PATTERN" value="%-12(%d{yyyy-MM-dd HH:mm:ss.SSS}) |-%-5level [%thread] %c [%L] -| %msg%n" /> 5. *<!-- test文件路径 -->* 6. <property name="TEST\_FILE\_PATH" value="d:/test.log" /> 7. *<!-- pro文件路径 -->* 8. <property name="PRO\_FILE\_PATH" value="/opt/test/log" /> 9. *<!-- 开发环境 -->* 10. <springProfile name="dev"> 11. <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender"> 12. <encoder> 13. <pattern>${PATTERN}</pattern> 14. </encoder> 15. </appender> 16. <logger name="com.light.springboot" level="debug" /> 17. <root level="info"> 18. <appender-ref ref="CONSOLE" /> 19. </root> 20. </springProfile> 21. *<!-- 测试环境 -->* 22. <springProfile name="test"> 23. *<!-- 每天产生一个文件 -->* 24. <appender name="TEST-FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"> 25. *<!-- 文件路径 -->* 26. <file>${TEST\_FILE\_PATH}</file> 27. <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"> 28. *<!-- 文件名称 -->* 29. <fileNamePattern>${TEST\_FILE\_PATH}/info.%d{yyyy-MM-dd}.log</fileNamePattern> 30. *<!-- 文件最大保存历史数量 -->* 31. <MaxHistory>100</MaxHistory> 32. </rollingPolicy> 33. <layout class="ch.qos.logback.classic.PatternLayout"> 34. <pattern>${PATTERN}</pattern> 35. </layout> 36. </appender> 37. <root level="info"> 38. <appender-ref ref="TEST-FILE" /> 39. </root> 40. </springProfile> 41. *<!-- 生产环境 -->* 42. <springProfile name="prod"> 43. <appender name="PROD\_FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"> 44. <file>${PRO\_FILE\_PATH}</file> 45. <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"> 46. <fileNamePattern>${PRO\_FILE\_PATH}/warn.%d{yyyy-MM-dd}.log</fileNamePattern> 47. <MaxHistory>100</MaxHistory> 48. </rollingPolicy> 49. <layout class="ch.qos.logback.classic.PatternLayout"> 50. <pattern>${PATTERN}</pattern> 51. </layout> 52. </appender> 53. <root level="warn"> 54. <appender-ref ref="PROD\_FILE" /> 55. </root> 56. </springProfile> 57. </configuration> |

springProfile 标签的 name 属性对应 application.properties 中的 spring.profiles.active 的配置

如需要自定义文件名称，在 application.properties 中配置 logging.config 选项即可。

Log4j2日志

首先添加依赖：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j2</artifactId>  </dependency> |

把spring boot默认的logging去掉(添加红字部分)

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions><!-- 去掉默认配置 -->  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-logging</artifactId>  </exclusion>  </exclusions>  </dependency> |

配置文件：

spring boot默认会加载类路径classpath:log4j2-spring.xml。

其内容：

|  |
| --- |
| 1. **<?**xml version="1.0" encoding="utf-8"**?>** 2. <configuration> 3. <properties> 4. *<!-- 文件输出格式 -->* 5. <property name="PATTERN">%d{yyyy-MM-dd HH:mm:ss.SSS} |-%-5level [%thread] %c [%L] -| %msg%n</property> 6. </properties> 7. <appenders> 8. <Console name="CONSOLE" target="system\_out"> 9. <PatternLayout pattern="${PATTERN}" /> 10. </Console> 11. </appenders> 12. <loggers> 13. <logger name="com.light.springboot" level="debug" /> 14. <root level="info"> 15. <appenderref ref="CONSOLE" /> 16. </root> 17. </loggers> 18. </configuration> |

log4j2 不能像 logback 那样在一个文件中设置多个环境的配置数据，只能命名 3 个不同名的日志文件，分别在 application-dev，application-test 和 application-prod 中配置 logging.config 选项。

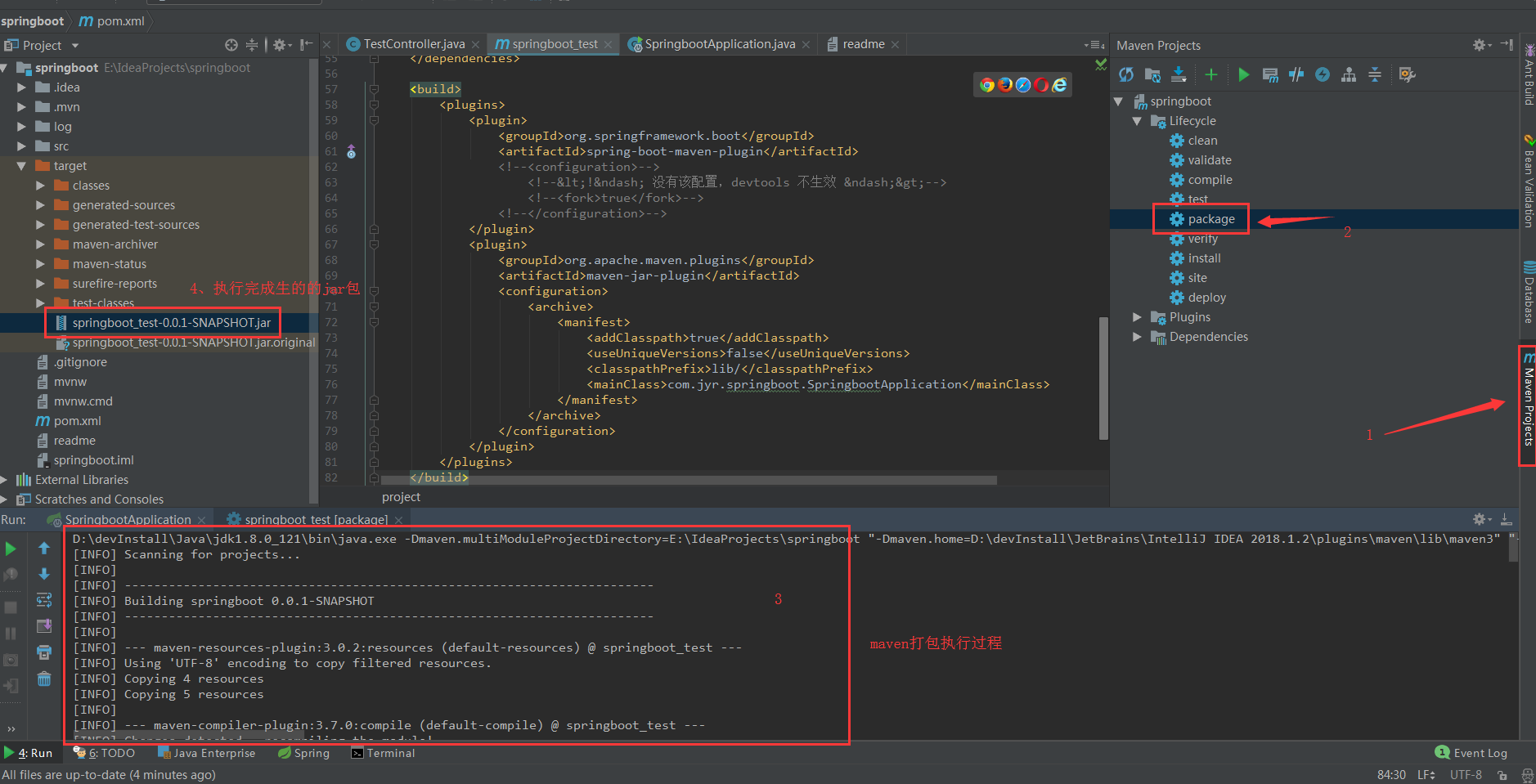
与logback一样，需要自定义文件名称时，在 application.properties中配置 logging.config 选项即可。

1. 打可执行jar包

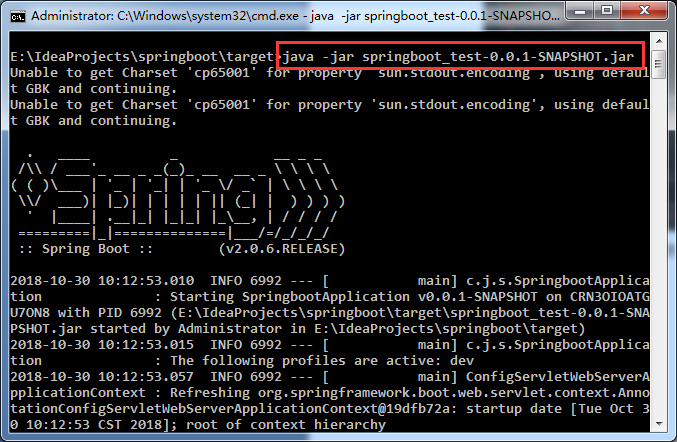
添加maven打包插件

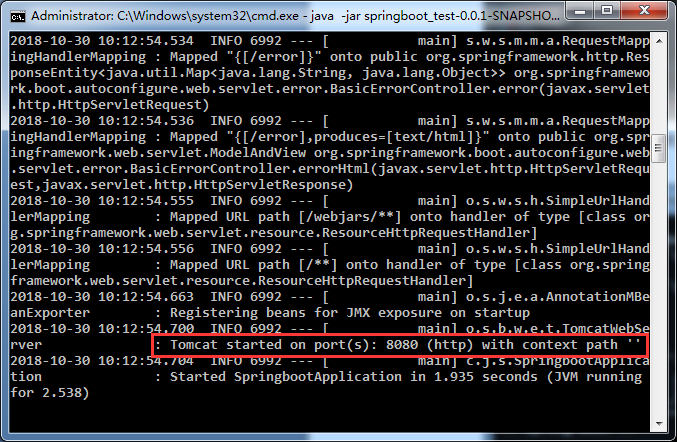
|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-jar-plugin</artifactId>  <configuration>  <archive>  <manifest>  <addClasspath>true</addClasspath>  <useUniqueVersions>false</useUniqueVersions>  <classpathPrefix>lib/</classpathPrefix>  <mainClass>com.jyr.springboot.SpringbootApplication</mainClass><!—这里改成你自己主程序的类名 -->  </manifest>  </archive>  </configuration>  </plugin> |

打开idea右侧的maven工具

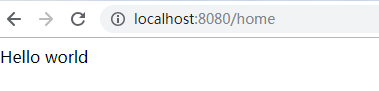


打开命令开切换到上面打包好的jar目录下(也可拷到你自己指定的目录中)，执行：java –jar + jar包名，程序即可运行





浏览器访问结果



打war包

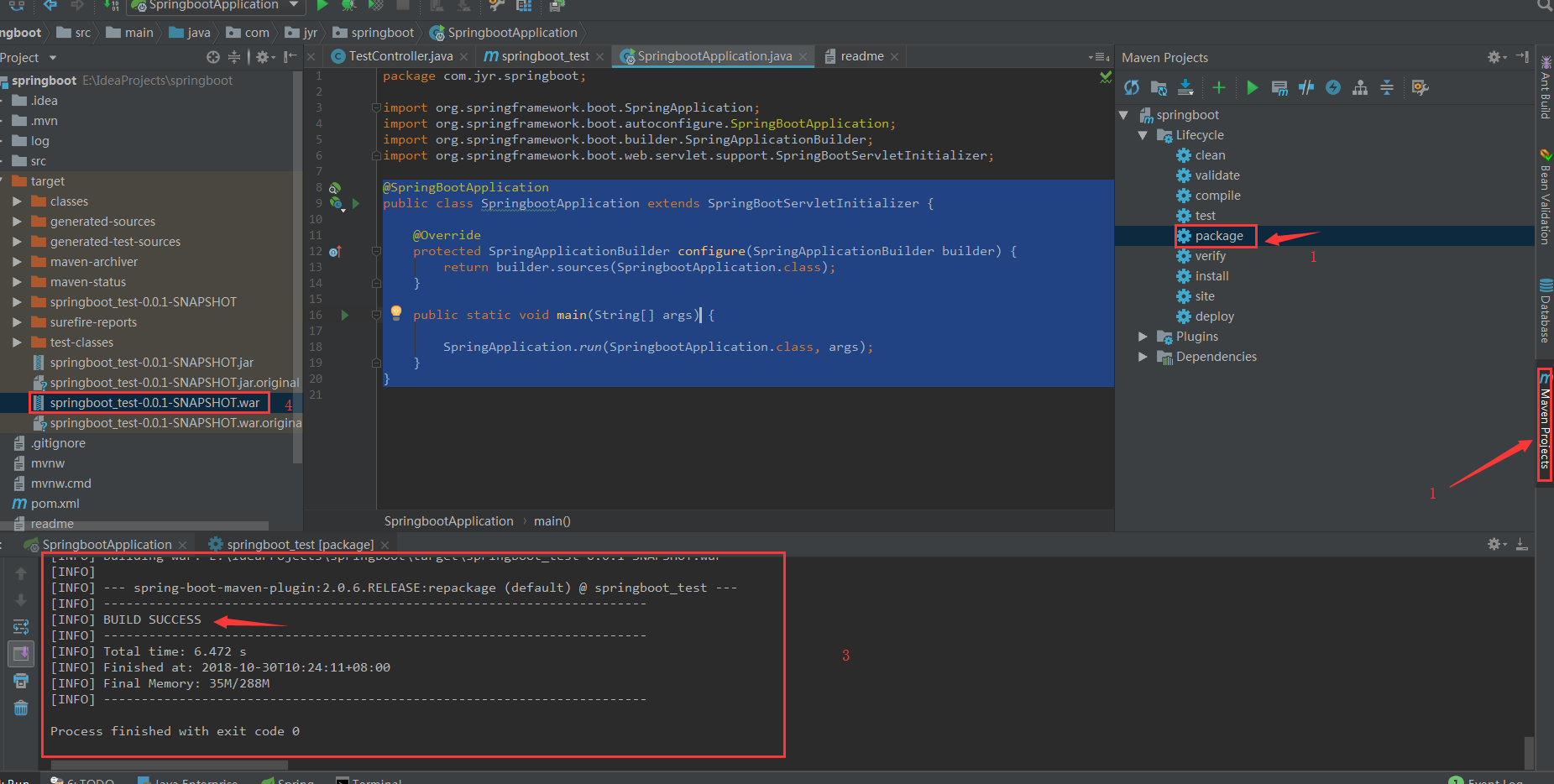
首先将打包方式修改成war包方式

<packaging>war</packaging>

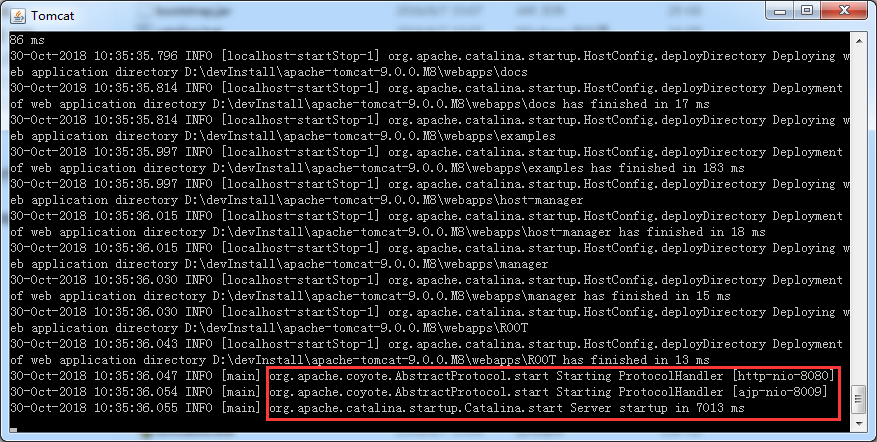
修改主类：

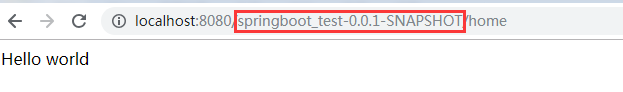
|  |
| --- |
| @SpringBootApplication  public class SpringbootApplication extends SpringBootServletInitializer {  @Override  protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  return builder.sources(SpringbootApplication.class);  }  public static void main(String[] args) {  SpringApplication.run(SpringbootApplication.class, args);  }  } |

执行打包命令：



将打包好的war包放到tomcat下的webapps目录里，运行服务器，浏览器访问





注意，此时的访问路径

参考：<https://www.cnblogs.com/moonlightL/p/7891803.html>

<https://www.cnblogs.com/acm-bingzi/p/6625303.html>

1. 整合 Freemarker

添加依赖：

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

添加配置(application.properties)：以下配置均为默认值

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

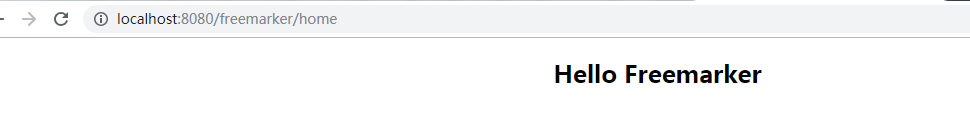
添加控制器：

|  |
| --- |
| @Controller  @RequestMapping("freemarker")  public class FreemarkerController {  @RequestMapping("home")  public String hello(Map<String,Object> map) {  map.put("msg", "Hello Freemarker");  System.out.println("freemarker/home");  return "hello";  }  } |

Templates目录下新建hello.flt：

|  |
| --- |
| <!DOCTYPE html>  <html lang="zh">  <head>  <meta charset="UTF-8">  <title>Document</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2>${msg}</h2>  </div>  </body>  </html> |

访问结果：



整合Thymeleaf：

添加依赖：

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

application.properties添加配置(默认配置)：

|  |
| --- |
| spring.thymeleaf.cache=true  spring.thymeleaf.prefix=classpath:/templates/  spring.thymeleaf.suffix=.html  spring.thymeleaf.mode=HTML  spring.thymeleaf.encoding=UTF-8  spring.thymeleaf.servlet.content-type=text/html |

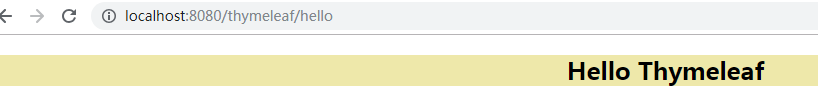
控制器：

|  |
| --- |
| @Controller  @RequestMapping("thymeleaf")  public class ThymeleafController {  @RequestMapping("hello")  public String hello(Map<String,Object> map) {  map.put("msg", "Hello Thymeleaf");  return "hello";  }  } |

Templates目录下新建hello.html：

|  |
| --- |
| <!DOCTYPE html>  <html lang="zh">  <head>  <meta charset="UTF-8">  <title>Document</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2 th:text="${msg}"></h2>  </div>  </body>  </html> |

运行结果：



1. 整合 Fastjson

添加依赖：

|  |
| --- |
| <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.51</version>  </dependency> |

创建一个配置管理类 WebConfig ：

|  |
| --- |
| @Configuration  public class WebConfig {  @Bean  public HttpMessageConverters fastJsonHttpMessageConverters() {  FastJsonHttpMessageConverter fastJsonHttpMessageConverter = new FastJsonHttpMessageConverter();  FastJsonConfig fastJsonConfig = new FastJsonConfig();  fastJsonConfig.setSerializerFeatures(SerializerFeature.PrettyFormat);    fastJsonHttpMessageConverter.setFastJsonConfig(fastJsonConfig);    HttpMessageConverter<?> converter = fastJsonHttpMessageConverter;    return new HttpMessageConverters(converter);  }  } |

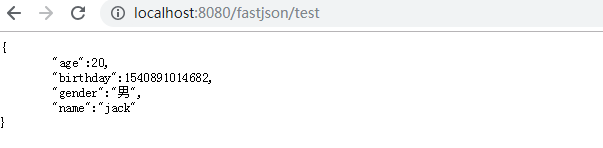
创建pojo对象：

|  |
| --- |
| public class User implements Serializable {  private String name;  private int age;  private String gender;  private Date birthday;  public User() {  }  public User(String name, int age) {  this.name = name;  this.age = age;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public String getGender() {  return gender;  }  public void setGender(String gender) {  this.gender = gender;  }  public Date getBirthday() {  return birthday;  }  public void setBirthday(Date birthday) {  this.birthday = birthday;  }  @Override  public String toString() {  return "User{" +  "name='" + name + '\'' +  ", age=" + age +  ", gender='" + gender + '\'' +  '}';  }  } |

创建控制器类 FastjsonController :

|  |
| --- |
| @Controller  @RequestMapping("fastjson")  public class FastJsonController {  @RequestMapping(value = "/test", produces = "application/json;charset=utf-8")  @ResponseBody  public User test() {  User user = new User();  user.setName("jack");  user.setAge(20);  user.setGender("男");  user.setBirthday(new Date());  return user;  }  } |

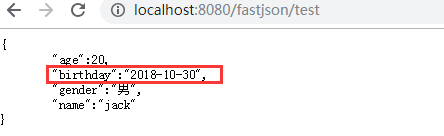
访问：



使用 Fastjson 的注解：将User中的birthday加注解



再次访问：



1. 自定义 Servlet

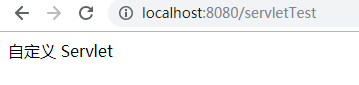
创建ServletTest:

|  |
| --- |
| public class ServletTest extends HttpServlet {  @Override  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  System.out.println(request.getMethod());  }  @Override  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  this.doGet(request,response);  }  } |

注册servlet: 将 Servelt 注册成 Bean。在上文创建的 WebConfig 类中添加如下代码

|  |
| --- |
| @Bean  public ServletRegistrationBean servletRegistrationBean() {  return new ServletRegistrationBean(new ServletTest(),"/servletTest");  } |

访问结果：



自定义过滤器/第三方过滤器

创建过滤器：

|  |
| --- |
| @Component  public class TimeFilter implements Filter {  @Override  public void init(FilterConfig filterConfig) throws ServletException {  System.out.println("=======初始化过滤器=========");  }  @Override  public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)  throws IOException, ServletException {  long start = System.currentTimeMillis();  filterChain.doFilter(request, response);  System.out.println("filter 耗时：" + (System.currentTimeMillis() - start));  }  @Override  public void destroy() {  System.out.println("=======销毁过滤器=========");  }  } |

使用注解方式（如上），也可使用注册bean的方式来添加过滤器（WebConfig）

|  |
| --- |
| @Bean  public FilterRegistrationBean timeFilter() {  FilterRegistrationBean registrationBean = new FilterRegistrationBean();    TimeFilter timeFilter = new TimeFilter();  registrationBean.setFilter(timeFilter);    List<String> urls = new ArrayList<>();  urls.add("/\*");  registrationBean.setUrlPatterns(urls);    return registrationBean;  } |

自定义监听器

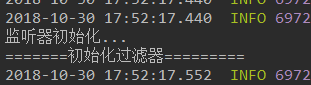
创建ListenerTest：

|  |
| --- |
| public class ListenerTest implements ServletContextListener {  @Override  public void contextInitialized(ServletContextEvent sce) {  System.out.println("监听器初始化...");  }  @Override  public void contextDestroyed(ServletContextEvent sce) {  }  } |

在WebConfig注册监听器：

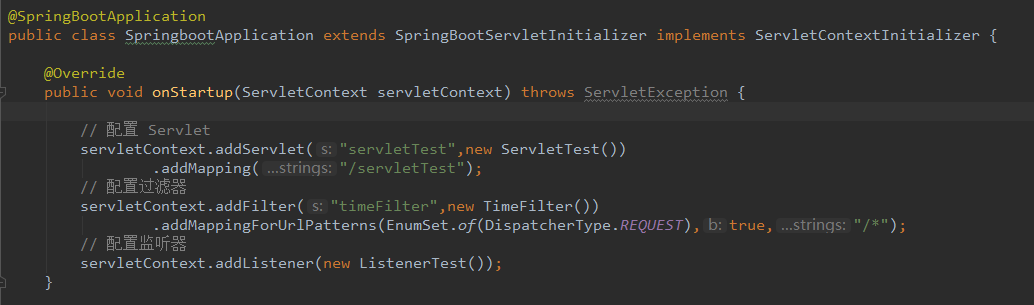
|  |
| --- |
| @Bean  public ServletListenerRegistrationBean<ListenerTest> servletListenerRegistrationBean() {  return new ServletListenerRegistrationBean<ListenerTest>(new ListenerTest());  } |

起动时可看到



自定义监听器、过滤器、servlet的另一种方式：

主类实现接口ServletContextInitializer，重写onStartup方法：



拦截器：

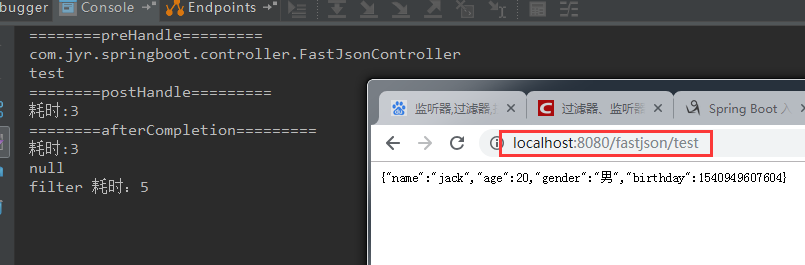
定义拦截器类TimeInterceptor(@Configuration注解一定要带上)

|  |
| --- |
| @Configuration  public class TimeInterceptor implements HandlerInterceptor {  @Override  public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {  System.out.println("========preHandle=========");  System.out.println(((HandlerMethod)handler).getBean().getClass().getName());  System.out.println(((HandlerMethod)handler).getMethod().getName());  request.setAttribute("startTime", System.currentTimeMillis());  return true;  }  @Override  public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {  System.out.println("========postHandle=========");  Long start = (Long) request.getAttribute("startTime");  System.out.println("耗时:"+(System.currentTimeMillis() - start));  }  @Override  public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception exception) {  System.out.println("========afterCompletion=========");  Long start = (Long) request.getAttribute("startTime");  System.out.println("耗时:"+(System.currentTimeMillis() - start));  System.out.println(exception);  }  } |

注册拦截器：WebConfig类继承WebMvcConfigurationSuppor，重写方法addInterceptors

|  |
| --- |
| /\*\*  \* 注册interceptor  \* @return  \*/  @Autowired  private TimeInterceptor timeInterceptor;  @Override  public void addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(timeInterceptor);  } |

此时拦截会拦截所有的action，servlet不会拦截



AOP 切面

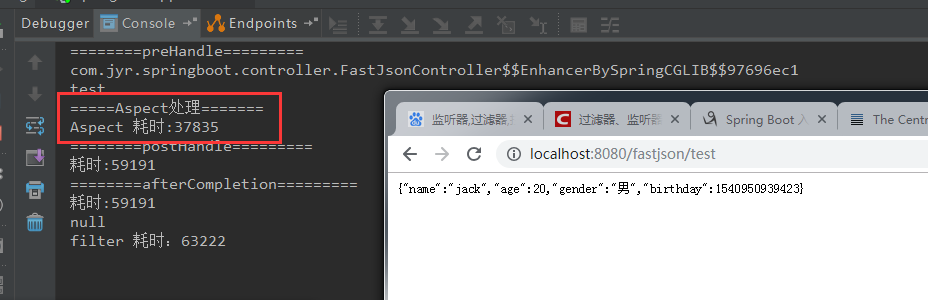
添加依赖

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

切面类：

|  |
| --- |
| @Aspect  @Component  public class TimeAspect {  @Around("execution(\* com.jyr.springboot.controller.FastJsonController..\*(..))")  public Object method(ProceedingJoinPoint pjp) throws Throwable {  System.out.println("=====Aspect处理=======");  Object[] args = pjp.getArgs();  for (Object arg : args) {  System.out.println("参数为:" + arg);  }  long start = System.currentTimeMillis();  Object object = pjp.proceed();  System.out.println("Aspect 耗时:" + (System.currentTimeMillis() - start));  return object;  }  } |

结果：

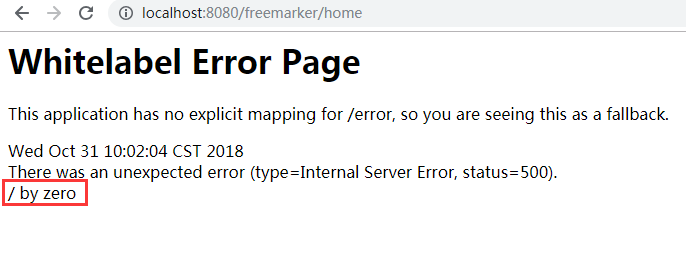


错误处理

先模拟异常：

|  |
| --- |
| @Controller  @RequestMapping("freemarker")  public class FreemarkerController {  @RequestMapping("home")  public String hello(Map<String,Object> map) {  map.put("msg", "Hello Freemarker");  System.out.println("freemarker/home");  // 模拟异常  int i = 1/0;  return "hello";  }  } |

访问结果：

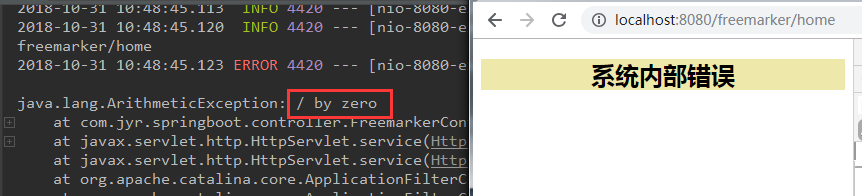


当系统出现异常或错误时，返回界面不友好，此时可以自定义错误页面，使用界面友好，给用户更好的体验

在resources目录下创建 /public/error子目录，并新建5xx.html文件:

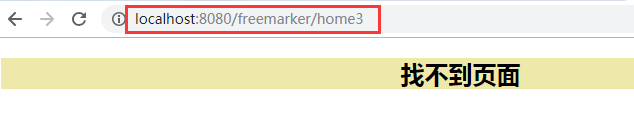
|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>错误页面</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2>系统内部错误</h2>  </div>  </body>  </html> |

结果：



404异常处理：在/public/error子目录，并新建404.html文件

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <title>404</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2>找不到页面</h2>  </div>  </body>  </html> |

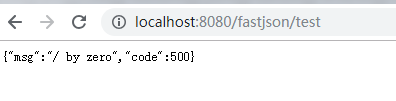
结果：

全局异常：前后端之间是以json格式传输数据的，出现异常时可以用该方式来处理。

编写一个类充当全局异常的处理类：

|  |
| --- |
| @ControllerAdvice  public class GlobalDefaultExceptionHandler {  /\*\*  \* 处理 Exception 类型的异常  \* @param e  \* @return  \*/  @ExceptionHandler(Exception.class)  @ResponseBody  public Map<String,Object> defaultExceptionHandler(Exception e) {  Map<String,Object> map = new HashMap<>();  map.put("code", 500);  map.put("msg", e.getMessage());  return map;  }  } |

结果：



文件上传和下载

添加依赖：

|  |
| --- |
| <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>2.4</version>  </dependency> |

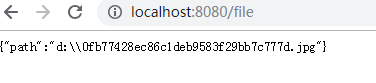
上传下载控制器：

|  |
| --- |
| @RestController  @RequestMapping("/file")  public class FileController {  private String path = "d:\\";  @PostMapping  public FileInfo upload(MultipartFile file) throws Exception {  System.out.println(file.getName());  System.out.println(file.getOriginalFilename());  System.out.println(file.getSize());  File localFile = new File(path, file.getOriginalFilename());  file.transferTo(localFile);  return new FileInfo(localFile.getAbsolutePath());  }  @GetMapping("/{id}")  public void download(@PathVariable String id, HttpServletRequest request, HttpServletResponse response) {  try (InputStream inputStream = new FileInputStream(new File(path, id + ".jpg"));  OutputStream outputStream = response.getOutputStream();) {  response.setContentType("application/x-download");  response.addHeader("Content-Disposition", "attachment;filename=" + id + ".jpg");  IOUtils.copy(inputStream, outputStream);  } catch (Exception e) {  e.printStackTrace();  }  }  } |

页面：

|  |
| --- |
| <form action="/file" enctype="multipart/form-data" method="post">  文件名：<input type="file" name="file" />  <br/>  <button>上传文件</button>  </form> |

上传成功返回



下载地址<http://localhost:8080/file/0fb77428ec86c1deb9583f29bb7c777d>

跨域请求：有两种处理方式，一种是使用注解，另一种是配置Bean

注解形式(细粒度)：在controller或方法上使用@CrossOrigin

|  |
| --- |
| @Controller  @RequestMapping("fastjson")  public class FastJsonController {  @CrossOrigin  @RequestMapping(value = "/test", produces = "application/json;charset=utf-8")  @ResponseBody  public User test() {  User user = new User();  user.setName("jack");  user.setAge(20);  user.setGender("男");  user.setBirthday(new Date());  // 模拟异常  int i = 1/0;  return user;  }  } |

配置Bean形式(粗粒度，全局):在WebConfig内添加如下内容

|  |
| --- |
| @Override  public void addCorsMappings(CorsRegistry registry) {  registry.addMapping("/fastjson/\*\*")  .allowedOrigins("http://localhost:8088");// 允许 8088 端口访问  // registry.addMapping("/api/\*\*")  // .allowedOrigins("http://domain2.com")  // .allowedMethods("PUT", "DELETE")  // .allowedHeaders("header1", "header2", "header3")  // .exposedHeaders("header1", "header2")  // .allowCredentials(false).maxAge(3600);  } |

整合 WebSocket

添加依赖：

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

WebSocket处理类：

|  |
| --- |
| @ServerEndpoint(value = "/websocket/{userName}")  @Component  public class WebSocketServer {  private static final Set<WebSocketServer> connections = new CopyOnWriteArraySet<>();  private String nickname;  private Session session;  private static String getDatetime(Date date) {  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return format.format(date);  }  @OnOpen  public void start(@PathParam("userName") String userName, Session session) {  this.nickname = userName;  this.session = session;  connections.add(this);  String message = String.format("\* %s %s", nickname, "加入聊天！");  broadcast(message);  }  @OnClose  public void end() {  connections.remove(this);  String message = String.format("\* %s %s", nickname, "退出聊天！");  broadcast(message);  }  @OnMessage  public void pushMsg(String message) {  broadcast("【" + this.nickname + "】" + getDatetime(new Date()) + " : " + message);  }  @OnError  public void onError(Throwable t) throws Throwable {  }  private static void broadcast(String msg) {  // 广播形式发送消息  for (WebSocketServer client : connections) {  try {  synchronized (client) {  client.session.getBasicRemote().sendText(msg);  }  } catch (IOException e) {  connections.remove(client);  try {  client.session.close();  } catch (IOException e1) {  e.printStackTrace();  }  String message = String.format("\* %s %s", client.nickname, "断开连接");  broadcast(message);  }  }  }  } |

注册：在WebConfig内添加如下内容（此方式只能用于jar包方式）

|  |
| --- |
| /\*\*  \* websocket  \* @return  \*/  @Bean  public ServerEndpointExporter serverEndpointExporter() {  return new ServerEndpointExporter();  } |

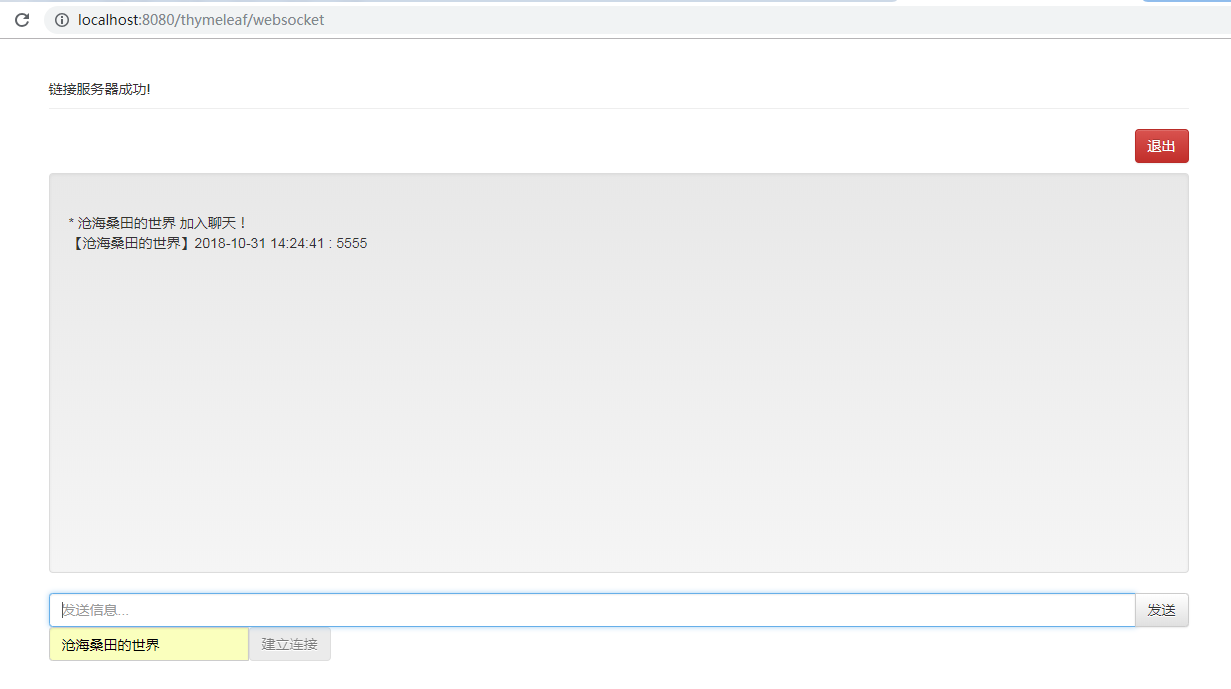
添加页面跳转控制器（ThymeleafController）：

|  |
| --- |
| @RequestMapping("websocket")  public String websocket(Map<String,Object> map) {  //map.put("msg", "Hello Thymeleaf");  return "websocket";  } |

页面websocket.html：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head lang="zh">  <meta charset="UTF-8">  <link rel="stylesheet" href="/css/bootstrap.css">  <link rel="stylesheet" href="/css/bootstrap-theme.css">  <script src="/js/jquery.min.js"></script>  <script src="/js/bootstrap.js"></script>  <style type="text/css">  #msg {  height: 400px;  overflow-y: auto;  }  #userName {  width: 200px;  }  #logout {  display: none;  }  </style>  <title>webSocket测试</title>  </head>  <body>  <div class="container">  <div class="page-header" id="tou">webSocket及时聊天Demo程序</div>  <p class="text-right" id="logout">  <button class="btn btn-danger" id="logout-btn">退出</button>  </p>  <div class="well" id="msg"></div>  <div class="col-lg">  <div class="input-group">  <input type="text" class="form-control" placeholder="发送信息..." id="message"> <span class="input-group-btn">  <button class="btn btn-default" type="button" id="send"  disabled="disabled">发送</button>  </span>  </div>  <div class="input-group">  <input id="userName" type="text" class="form-control" name="userName" placeholder="输入您的用户名" />  <button class="btn btn-default" type="button" id="connection-btn">建立连接</button>  </div>  <!-- /input-group -->  </div>  <!-- /.col-lg-6 -->  </div>  <!-- /.row -->  </div>  <script type="text/javascript">  $(function() {  var websocket;  $("#connection-btn").bind("click", function() {  var userName = $("#userName").val();  if (userName == null || userName == "") {  alert("请输入您的用户名");  return;  }  connection(userName);  });  function connection(userName) {  var host = window.location.host;  if ('WebSocket' in window) {  websocket = new WebSocket("ws://" + host +  "/websocket/" + userName);  } else if ('MozWebSocket' in window) {  websocket = new MozWebSocket("ws://" + host +  "/websocket/" + userName);  }  websocket.onopen = function(evnt) {  $("#tou").html("链接服务器成功!")  $("#send").prop("disabled", "");  $("#connection-btn").prop("disabled", "disabled");  $("#logout").show();  };  websocket.onmessage = function(evnt) {  $("#msg").html($("#msg").html() + "<br/>" + evnt.data);  };  websocket.onerror = function(evnt) {  $("#tou").html("报错!")  };  websocket.onclose = function(evnt) {  $("#tou").html("与服务器断开了链接!");  $("#send").prop("disabled", "disabled");  $("#connection-btn").prop("disabled", "");  $("#logout").hide();  }  }  function send() {  if (websocket != null) {  var $message = $("#message");  var data = $message.val();  if (data == null || data == "") {  return;  }  websocket.send(data);  $message.val("");  } else {  alert('未与服务器链接.');  }  }  $('#send').bind('click', function() {  send();  });  $(document).on("keypress", function(event) {  if (event.keyCode == "13") {  send();  }  });  $("#logout-btn").on("click", function() {  websocket.close(); //关闭TCP连接  });  });  </script>  </body>  </html> |

结果：



适用于 jar 包方式运行和 war 方式运行的websocket

WebSocket 处理类:

|  |
| --- |
| public class WebSocket\_Server extends TextWebSocketHandler {    private static final Map<WebSocketSession, String> connections = new ConcurrentHashMap<>();  private static String getDatetime(Date date) {  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return format.format(date);  }  /\*\*  \* 建立连接  \*/  @Override  public void afterConnectionEstablished(WebSocketSession session) throws Exception {  String uri = session.getUri().toString();  String userName = uri.substring(uri.lastIndexOf("/") + 1);  String nickname = URLDecoder.decode(userName, "utf-8");  connections.put(session, nickname);  String message = String.format("\* %s %s", nickname, "加入聊天！");  broadcast(new TextMessage(message));  }  /\*\*  \* 断开连接  \*/  @Override  public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {  String nickname = connections.remove(session);  String message = String.format("\* %s %s", nickname, "退出聊天！");  broadcast(new TextMessage(message));  }  /\*\*  \* 处理消息  \*/  @Override  protected void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {  String msg = "【" + connections.get(session) + "】" + getDatetime(new Date()) + " : " + message.getPayload();  broadcast(new TextMessage(msg));  }  private static void broadcast(TextMessage msg) {  // 广播形式发送消息  for (WebSocketSession session : connections.keySet()) {  try {  synchronized (session) {  session.sendMessage(msg);  }  } catch (Exception e) {  connections.remove(session);  try {  session.close();  } catch (Exception e2) {  e2.printStackTrace();  }  String message = String.format("\* %s %s", connections.get(session), "断开连接");  broadcast(new TextMessage(message));  }  }  }  } |

WebSocket 配置类:

|  |
| --- |
| @Configuration  @EnableWebSocket  public class WebSocketConfig implements WebSocketConfigurer {  @Override  public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {  registry.addHandler(webSocketServer(), "/webSocket/\*");  }  @Bean  public WebSocketHandler webSocketServer() {  return new WebSocket\_Server();  }  } |

注意，页面里的ws地址修改了

结果：



整合 Swagger2

添加依赖：

|  |
| --- |
| <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger2</artifactId>  <version>2.7.0</version>  </dependency>  <dependency>  <groupId>io.springfox</groupId>  <artifactId>springfox-swagger-ui</artifactId>  <version>2.7.0</version>  </dependency> |

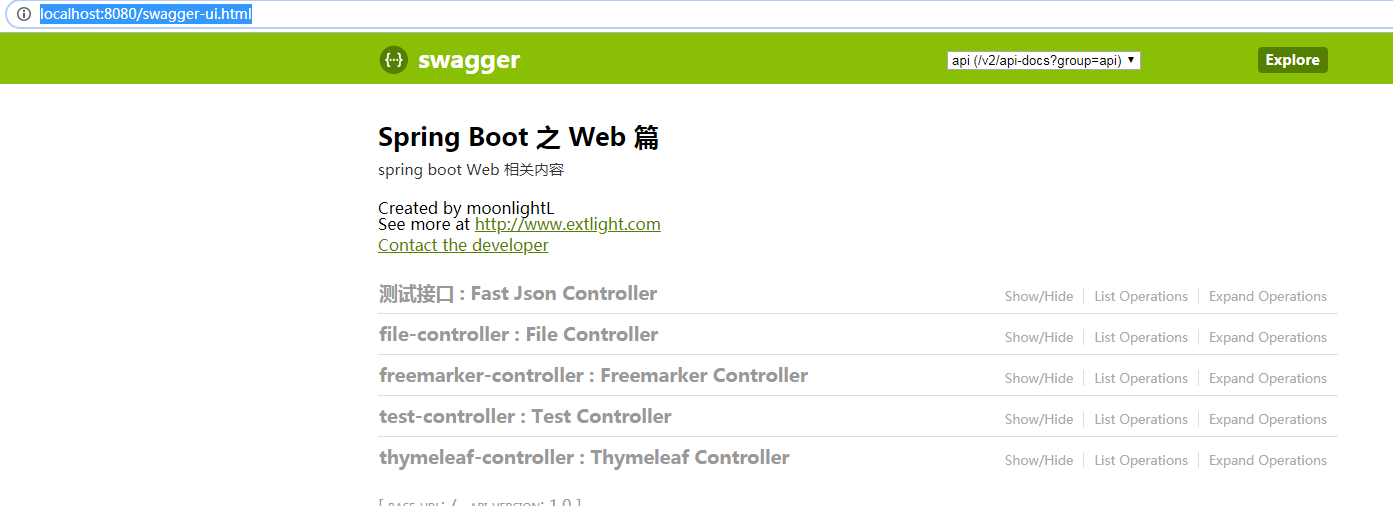
配置类

|  |
| --- |
| @Configuration  @EnableSwagger2  public class Swagger2Configuration {  @Bean  public Docket accessToken() {  return new Docket(DocumentationType.SWAGGER\_2)  .groupName("api")// 定义组  .select() // 选择那些路径和 api 会生成 document  .apis(RequestHandlerSelectors.basePackage("com.jyr.springboot.controller")) // 拦截的包路径  .paths(PathSelectors.regex("/\*/.\*"))// 拦截的接口路径  .build() // 创建  .apiInfo(apiInfo()); // 配置说明  }  private ApiInfo apiInfo() {  return new ApiInfoBuilder()//  .title("Spring Boot 之 Web 篇")// 标题  .description("spring boot Web 相关内容")// 描述  .termsOfServiceUrl("http://www.extlight.com")//  .contact(new Contact("moonlightL", "http://www.extlight.com", "445847261@qq.com"))// 联系  .version("1.0")// 版本  .build();  }  } |

以 FastJsonController 为例：

|  |
| --- |
| @Api(value = "FastJson测试", tags = { "测试接口" })  @Controller  @RequestMapping("fastjson")  public class FastJsonController {  @CrossOrigin  @RequestMapping(value = "/test", produces = "application/json;charset=utf-8")  @ResponseBody  public User test() {  User user = new User();  user.setName("jack");  user.setAge(20);  user.setGender("男");  user.setBirthday(new Date());  // 模拟异常  int i = 1/0;  return user;  }  @ApiOperation("获取用户信息")  @ApiImplicitParam(name = "name", value = "用户名", dataType = "string", paramType = "query")  @GetMapping("/test/{name}")  public User test(@PathVariable("name") String name) {  User user = new User();  user.setName("jack");  user.setAge(20);  user.setGender("男");  user.setBirthday(new Date());  return user;  }  } |

访问<http://localhost:8080/swagger-ui.html>：



1. 默认情况下, 网页存放于static目录下, 默认的"/"指向的是~/resouces/static/index.html文

2. 如果引入了thymeleaf, 则默认指向的地址为~/resouces/templates/index.html

# 整合 JdbcTemplate

添加依赖

|  |
| --- |
| <!-- jdbc -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <!-- mysql 驱动包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

配置数据库连接：在application.properties添加数据库连接参数

|  |
| --- |
| # mysql连接参数  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root |

创建表：

|  |
| --- |
| CREATE TABLE USER(  id INT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR(30),  age INT,  birthday DATE  ) |

实体类user:

|  |
| --- |
| public class User implements Serializable {  private Long id;  private String name;  private int age;  private String gender;  @JSONField(format = "yyyy-MM-dd")  private Date birthday;  public User() {  }  public User(String name, int age) {  this.name = name;  this.age = age;  }  ....get and set...  } |

dao 接口:

|  |
| --- |
| public interface UserDao {  public int insert(User user);  public int deleteById(Long id);  public int update(User user);  public User getById(Long id);  } |

Dao实现类

|  |
| --- |
| @Repository  public class UserDaoImpl implements UserDao {  @Autowired  private JdbcTemplate jdbcTemplate;  @Override  public int insert(User user) {  String sql = "INSERT INTO USER(NAME,age, birthday) VALUES(?,?,?)";  return this.jdbcTemplate.update(  sql,  user.getName(),  user.getAge(),  user.getBirthday()  );  }  @Override  public int deleteById(Long id) {  String sql = "delete from user where id = ?";  return this.jdbcTemplate.update(sql,id);  }  @Override  public int update(User user) {  String sql = "update user set age = ? where id = ?";  return this.jdbcTemplate.update(  sql,  user.getAge(),  user.getId()  );  }  @Override  public User getById(Long id) {  String sql = "select \* from user where id = ?";  return this.jdbcTemplate.queryForObject(sql, new RowMapper<User>() {  @Override  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  User user = new User();  user.setId(rs.getLong("id"));  user.setName(rs.getString("name"));  user.setAge(rs.getInt("age"));  user.setBirthday(rs.getDate("birthday"));  return user;  }  },id);  }  } |

测试类：在src/test下新建测试类

|  |
| --- |
| @RunWith(SpringRunner.class)  @SpringBootTest  public class UserDaoTest {  @Autowired  private UserDao userDao;  @Test  public void testInsert() {  User user = new User();  user.setName("张三");  user.setAge(33);  user.setBirthday(new Date());  int result = this.userDao.insert(user);  System.out.println(result);  }  @Test  public void testGetById() {  User user = this.userDao.getById(1L);  System.out.println(user.getName());  }  @Test  public void testUpdate() {  User user = new User();  user.setId(1L);  user.setAge(55);  this.userDao.update(user);  }  @Test  public void testDeleteById() {  int result = this.userDao.deleteById(1L);  System.out.println(result);  }  } |

## 整合 Spring-data-jpa

添加依赖

|  |
| --- |
| <!-- jdbc -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <!-- springboot,jpa 整合包-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <!-- mysql 驱动包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

配置数据库连接：在 application.properties 中添加

|  |
| --- |
| # 数据库连接配置  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root  # JPA 配置  spring.jpa.hibernate.ddl-auto=update  spring.jpa.show-sql=true |

建表role:

|  |
| --- |
| CREATE TABLE role (  id BIGINT PRIMARY KEY AUTO\_INCREMENT,  NAME VARCHAR(10) NOT NULL,  descr VARCHAR(100) NULL DEFAULT NULL  ) |

建实体role:

|  |
| --- |
| @Entity  public class Role implements Serializable {  @Id  @GeneratedValue  private Long id;  @Column  private String name;  @Column  private String descr;  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getDescr() {  return descr;  }  public void setDescr(String descr) {  this.descr = descr;  }  } |

RoleRepository接口：

|  |
| --- |
| public interface RoleRepository extends JpaRepository<Role, Integer> {    } |

测试类

|  |
| --- |
| @RunWith(SpringRunner.class)  @SpringBootTest  public class RoleRepositoryTest {  @Autowired  private RoleRepository roleRepository;  @Test  public void testInsert() {  Role role = new Role();  role.setName("管理员");  role.setDescr("测试");  Role result = this.roleRepository.save(role);  System.out.println(result);  }  @Test  public void testFindOne() {  Role role = this.roleRepository.findById(1L).get();  System.out.println(role);  }  @Test  public void testUpdate() {  Role role = new Role();  role.setId(1L);  role.setName("管理员");  role.setDescr("控制权限");  Role result = this.roleRepository.save(role);  System.out.println(result);  }  @Test  public void testDelete() {  this.roleRepository.deleteById(1L);  }  } |

## 整合mybatis

有两种方式

方式一：mybatis官方提供的spring boot整合包

添加依赖

|  |
| --- |
| <!-- springboot,mybatis 整合包 -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency>  <!-- mysql 驱动包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

数据库连接配置：

|  |
| --- |
| # 数据源配置  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root  # mybatis 配置  mybatis.config-location=classpath:mybatis/mybatis-config.xml  mybatis.mapper-locations=classpath:mybatis/mapper/\*.xml |

方式二：使用mybatis-spring整合方式，也就是传统的方式(推荐)

添加依赖

|  |
| --- |
| <!-- jdbc -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <!-- mybatis -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.4</version>  </dependency>  <!-- spring,mybatis整合包 -->  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>1.3.2</version>  </dependency>  <!-- mysql 驱动包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

数据库连接配置：

|  |
| --- |
| # 数据源配置  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root |

创建配置类：

|  |
| --- |
| package com.jyr.springboot.utils;  import org.mybatis.spring.SqlSessionFactoryBean;  import org.mybatis.spring.mapper.MapperScannerConfigurer;  import org.springframework.boot.autoconfigure.condition.ConditionalOnBean;  import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.core.io.ClassPathResource;  import org.springframework.core.io.Resource;  import org.springframework.core.io.support.PathMatchingResourcePatternResolver;  import javax.sql.DataSource;  import java.io.IOException;  @Configuration  public class MyBatisConfiguration {  @Bean  @ConditionalOnMissingBean // 当容器里没有指定的 Bean 的情况下创建该对象  public SqlSessionFactoryBean sqlSessionFactory(DataSource dataSource) {  SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();  // 设置数据源  sqlSessionFactoryBean.setDataSource(dataSource);  // 设置mybatis的主配置文件  sqlSessionFactoryBean.setConfigLocation(new ClassPathResource("mybatis/mybatis-config.xml"));  // 设置mapper映射文件  PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();  Resource[] mapperXml;  try {  mapperXml = resolver.getResources("classpath:mybatis/mapper/\*.xml");  sqlSessionFactoryBean.setMapperLocations(mapperXml);  } catch (IOException e) {  e.printStackTrace();  }  // 设置别名包  sqlSessionFactoryBean.setTypeAliasesPackage("com.jyr.springboot.pojo");  return sqlSessionFactoryBean;  }  @Bean  @ConditionalOnBean(SqlSessionFactoryBean.class) // 当 SqlSessionFactoryBean 实例存在时创建对象  public MapperScannerConfigurer mapperScannerConfigurer() {  MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer();  mapperScannerConfigurer.setBasePackage("com.jyr.springboot.dao");  return mapperScannerConfigurer;  }  } |

以上为两种方式的不同之处，以下相同

resources目录下创建mybatis和mybatis/mapper目录，并新建配置文件

mybatis-config.xml内容如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd">  <configuration>  <settings>  <!-- 获取数据库自增主键值 -->  <setting name="useGeneratedKeys" value="true"/>  <!-- 使用列别名替换列名，默认为 true -->  <setting name="useColumnLabel" value="true"/>  <!-- 开启驼峰命名转换：Table(create\_time) => Entity(createTime) -->  <setting name="mapUnderscoreToCamelCase" value="true"/>  </settings>  </configuration> |

创建数据表：

|  |
| --- |
| CREATE TABLE department (  id BIGINT(11) PRIMARY KEY,  NAME VARCHAR(10) NOT NULL,  descr VARCHAR(50) NULL DEFAULT NULL  ) |

实体：

|  |
| --- |
| public class Department implements Serializable {  private static final long serialVersionUID = 6067283535977178571L;  private Integer id;  private String name;  private String descr;  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getDescr() {  return descr;  }  public void setDescr(String descr) {  this.descr = descr;  }  } |

Mapper 接口

|  |
| --- |
| package com.jyr.springboot.dao;  import com.jyr.springboot.pojo.Department;  import org.apache.ibatis.annotations.Mapper;  @Mapper  public interface DepartmentDao {  void insert(Department department);  Department getById(Long id);  void update(Department department);  void deleteById(Long id);  } |

departmentMapper.xml配置文件：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.jyr.springboot.pojo.Department">  <insert id="insert" parameterType="com.jyr.springboot.pojo.Department">  insert into department(id,name,descr) values(#{id},#{name},#{descr})  </insert>  <select id="getById" parameterType="java.lang.Long" resultType="com.jyr.springboot.pojo.Department">  select id,name,descr from department where id = #{id}  </select>  <update id="update" parameterType="com.jyr.springboot.pojo.Department">  update department set descr = #{descr} where id = #{id}  </update>  <delete id="deleteById" parameterType="java.lang.Long">  delete from department where id = #{id}  </delete>  </mapper> |

测试类

|  |
| --- |
| package com.jyr.springboot.dao;  import com.jyr.springboot.pojo.Department;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringRunner;  import javax.annotation.Resource;  @RunWith(SpringRunner.class)  @SpringBootTest  public class DepartmentTest {  @Resource  private DepartmentDao departmentMapper;  @Test  public void testInsert() {  Department department = new Department();  department.setId(4L);  department.setName("研发部");  department.setDescr("开发产品");  this.departmentMapper.insert(department);  }  @Test  public void testGetById() {  Department department = this.departmentMapper.getById(1L);  System.out.println(department);  }  @Test  public void testUpdate() {  Department department = new Department();  department.setId(1L);  department.setDescr("开发高级产品");  this.departmentMapper.update(department);  }  @Test  public void testDeleteById() {  this.departmentMapper.deleteById(1L);  }  } |

配置 Druid 数据源(原始 jar 包 + 手动编程方式)

依赖：

|  |
| --- |
| <!--druid-->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.1.12</version>  </dependency> |

配置文件

|  |
| --- |
| spring.datasource.druid.driverClassName=com.mysql.jdbc.Driver  spring.datasource.druid.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.druid.username=root  spring.datasource.druid.password=tiger  spring.datasource.druid.initialSize=5  spring.datasource.druid.minIdle=5  spring.datasource.druid.maxActive=20  spring.datasource.druid.maxWait=60000  spring.datasource.druid.timeBetweenEvictionRunsMillis=60000  spring.datasource.druid.min-evictableIdleTimeMillis=300000  spring.datasource.druid.validationQuery=SELECT 1 FROM DUAL  spring.datasource.druid.testWhileIdle=true  spring.datasource.druid.testOnBorrow=false  spring.datasource.druid.testOnReturn=false  spring.datasource.druid.poolPreparedStatements=true  spring.datasource.druid.maxPoolPreparedStatementPerConnectionSize=20  spring.datasource.druid.filters=stat,wall |

DruidConfiguration配置类

|  |
| --- |
| package com.jyr.springboot.utils;  import com.alibaba.druid.filter.Filter;  import com.alibaba.druid.filter.stat.StatFilter;  import com.alibaba.druid.pool.DruidDataSource;  import org.springframework.boot.context.properties.ConfigurationProperties;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import java.util.Arrays;  @Configuration  public class DruidConfiguration {  @ConfigurationProperties(prefix = "spring.datasource.druid")  @Bean(initMethod = "init",destroyMethod = "close")  public DruidDataSource dataSource() {  DruidDataSource ds = new DruidDataSource();  ds.setProxyFilters(Arrays.asList(statFilter()));  return ds;  }  @Bean  public Filter statFilter() {  StatFilter filter = new StatFilter();  filter.setSlowSqlMillis(5000);  filter.setLogSlowSql(true);  filter.setMergeSql(true);  return filter;  }  } |

注意，当数据库连接配置也是用spring.datasource.druid作前缀，druid配置类这样写@ConfigurationProperties(prefix = "spring.datasource.druid")

当数据库连接配置前缀没有druid时，即spring.datasource.url时，druid配置类这样写@ConfigurationProperties(prefix = "spring.datasource ")

否则会报dbType错误：java.lang.IllegalStateException: dbType not support : null, url null

配置 Druid 数据源(使用springboot整合包)：不用写配置类

添加依赖：

|  |
| --- |
| <!--使用springboot整合包-->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  <version>1.1.10</version>  </dependency> |

配置文件：

|  |
| --- |
| spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/springboot?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=tiger  # 修改数据源  spring.datasource.type=com.alibaba.druid.pool.DruidDataSource  spring.datasource.druid.initial-size=5  spring.datasource.druid.min-idle=5  spring.datasource.druid.max-active=20  spring.datasource.druid.max-wait=60000  spring.datasource.druid.time-between-eviction-runs-millis=60000  spring.datasource.druid.min-evictable-idle-time-millis=300000  spring.datasource.druid.validation-query=SELECT 1 FROM DUAL  spring.datasource.druid.test-while-idle=true  spring.datasource.druid.test-on-borrow=false  spring.datasource.druid.test-on-return=false  spring.datasource.druid.pool-prepared-statements=true  spring.datasource.druid.max-pool-prepared-statement-per-connection-size=20  spring.datasource.druid.filters=stat,wall,log4j2 |