**Springboot培训内容**

# springboot简介

1. Spring boot是Spring家族中的一个全新的框架，它用来简化Spring应用程序的创建和开发过程，也可以说Spring boot能简化我们之前采用Spring mvc + spring + MyBatis框架进行开发的过程
2. 在以往我们采用Spring mvc + Spring + MyBatis框架进行开发的时候，搭建和整合三大框架，我们需要做很多工作，比如配置web.xml，配置Spring，配置MyBatis，并将它们整合在一起等，而Spring boot框架对此开发过程进行了革命性的颠覆，抛弃了繁琐的xml配置过程，采用大量的默认配置简化我们的开发过程
3. 所以采用Spring boot可以非常容易和快速地创建基于Spring框架的应用程序，它让编码变简单了，配置变简单了，部署变简单了，监控变简单了
4. 正因为Spring boot它化繁为简，让开发变得极其简单和快速，所以在业界备受关注

# springboot的特性

1. 能够快速创建基于Spring的应用程序
2. 能够直接使用java main方法启动内嵌的Tomcat，jetty服务器运行Spring boot程序，不需要部署war包文件
3. 提供约定的starter POM来简化Maven配置，让Maven的配置变得简单
4. 根据项目的Maven依赖配置，Spring boot自动配置Spring、Spring mvc等
5. 提供了程序的健康检查等功能
6. 基本可以完全不使用XML配置文件，采用注解配置

# springboot开发环境

1. Spring Boot使用版本为1.5.19.RELEASE
2. 如果是使用eclipse，推荐安装Spring Tool Suite（STS）插件
3. 如果使用IDEA旗舰版，自带了Springboot插件
4. Maven使用版本为3.3.9
5. Jdk1.8

# 第一个springboot程序

快速开发一个Spring boot程序步骤如下：

1. 创建一个Spring boot项目

* 可以采用方式一：使用eclipse的Spring Tool Suite（STS）插件/或者IDEA自带的插件创建
* 可以采用方式二：直接使用Maven创建项目的方式创建

1. 加入Spring boot的父级和起步依赖

* 父级依赖：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.19.RELEASE</version>  <relativePath/>  </parent> |

加入Spring boot父级依赖可以简化我们项目的Maven配置

* 起步依赖：

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

加入Spring boot的起步依赖也可以简化我们项目的Maven配置

1. 创建Spring boot的入口main方法

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

1. 创建一个Spring mvc的Controller

|  |
| --- |
| @Controller  **public** **class** HelloController {  @RequestMapping("/boot/hello")  **public** @ResponseBody String hello() {  **return** "Hello Spring Boot.";  }  } |

1. 运行Spring boot的入口main方法

至此，第一个Spring boot程序开发完成

# 第一个Spring boot程序解析

1. Spring Boot的父级依赖spring-boot-starter-parent配置之后，当前的项目就是Spring Boot项目
2. spring-boot-starter-parent是一个特殊的starter依赖，它用来提供相关的Maven默认依赖，使用它之后，常用的jar包依赖可以省去version配置
3. Spring Boot提供了哪些默认jar包的依赖，可查看该父级依赖的pom文件
4. 如果不想使用某个默认的依赖版本，可以通过pom.xml文件的属性配置覆盖各个依赖项，比如覆盖Spring版本：

|  |
| --- |
| <properties>  <spring.version>5.0.0.RELEASE</spring.version>  </properties> |

1. [@SpringBootApplication](mailto:5.@SpringBootApplication)注解是Spring Boot项目的核心注解，主要作用是开启Spring自动配置
2. Main方法是一个标准的Java程序的main方法，主要作用是作为项目启动运行的入口
3. @Controller及@ResponseBody依然是我们之前的Spring mvc，因为Spring　boot的里面依然是使用我们的Spring mvc +　Spring + MyBatis等框架

要保证其他类放在使用@SpringBootApplication注释并带有main方法的类的同级或下级，否则扫描不到其他类。

# springboot核心配置文件

Spring boot的核心配置文件用于配置Spring boot程序，有两种格式的配置文件

1.properties文件

键值对的properties属性文件配置方式

2.yml文件

Yml是一种yaml格式的配置文件，主要采用一定的空格、换行等格式排版进行配置

Yaml是一种直观的能够被计算机识别的数据序列化格式，容易被人类阅读，yaml类似于xml，但是语法比xml简洁很多

值与前面的冒号配置项必须要有一个空格

Yml后缀也可以使用yaml后缀

配置示例

|  |
| --- |
| #配置服务器端口  server.port=8088  #配置应用访问路径  server.context-path=/demo |

|  |
| --- |
| server:  port:8088  context-path:/demo |

多环境配置文件

|  |
| --- |
| #比如配置开发环境  spring.profiles.active=dev  application-dev.properties  #比如配置上线环境  spring.profiles.active=online  application-online.properties  #比如配置测试环境  spring.profiles.active=test  application-test.properties |

|  |
| --- |
| spring:  profiles:  active: test  --- server:  port: 8081 spring:  profiles: dev  ---  server:  port: 8082 spring:  profiles: online   ---  server:  port: 8083 spring:  profiles: test |

# springboot下的springmvc

Spring boot下的Spring mvc和之前的spring mvc使用是完全一样的

@Controller

即为spring mvc的注解，处理http请求

@RestController

Spring4后新增注解，是@Controller与@ResponseBody的组合注解，用于返回字符串或json数据

@GetMapping

RequestMapping和Get请求方法的组合

@PostMapping

RequestMapping和Post请求方法的组合

@PutMapping

RequestMapping和Put请求方法的组合

@DeleteMapping

RequestMapping和Delete请求方法的组合

@PutMapping可以用@PostMapping代替，@DeleteMapping可以用@GetMappng代替，所以实际中@PutMapping和@DeleteMapping使用的比较少。

|  |
| --- |
| @RestController  **public** **class** HelloController {  @RequestMapping("/boot/hello")  **public** String hello(){  **return** "Hello Spring Boot.";  }  } |

# springboot使用jsp

在Spring boot中使用jsp，按如下步骤进行：

1. 在pom.xml文件中配置依赖项

|  |
| --- |
| <!-- 引入Spring Boot内嵌的Tomcat对JSP的解析包 -->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  </dependency>  <!-- servlet依赖的jar包start -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  </dependency>  <!-- servlet依赖的jar包end -->  <!-- jsp依赖的jar包start -->  <dependency>  <groupId>javax.servlet.jsp</groupId>  <artifactId>javax.servlet.jsp-api</artifactId>  <version>2.3.1</version>  </dependency>  <!-- jsp依赖的jar包end -->  <!-- jstl标签依赖的jar包start -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  </dependency>  <!-- jstl标签依赖的jar包end --> |

1. 在application.properties文件配置spring mvc的视图展示为jsp

|  |
| --- |
| spring.mvc.view.prefix=/  spring.mvc.view.suffix=.jsp |

1. 在src/main下创建一个webapp目录，然后在该目录下新建jsp页面

build中要配置以下信息（idea）

|  |
| --- |
| <resources>  <resource>  <directory>src/main/java</directory>  <includes>  <include>\*\*/\*.xml</include>  </includes>  </resource>  <resource>  <directory>src/main/resources</directory>  <includes>  <include>\*\*/\*.\*</include>  </includes>  </resource>  <resource>  <directory>src/main/webapp</directory>  <targetPath>META-INF/resources</targetPath>  <includes>  <include>\*\*/\*.\*</include>  </includes>  </resource>  </resources> |

1. 编写controller

|  |
| --- |
| @Controller  **public** **class** JspController {  @RequestMapping("/index")  **public** String index(Model model){  model.addAttribute("msg","spring boot 集成 jsp");  **return** "index";  }  } |

# springboot集成MyBatis

Spring boot集成MyBatis的步骤如下：

1. 在pom.xml中配置相关jar依赖

|  |
| --- |
| <!-- 加载mybatis整合springboot -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.1</version>  </dependency>  <!-- MySQL的jdbc驱动包 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

1. 在springboot的核心配置文件application.properties中配置MyBatis的Mapper.xml文件所在位置：

|  |
| --- |
| mybatis.mapper-locations=classpath:mybatis/mapper/\*.xml |

1. 在springboot的核心配置文件application.properties中配置数据源：

|  |
| --- |
| #配置数据库连接信息  spring.datasource.username=root  spring.datasource.password=123456  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://127.0.0.1:3306/springdb?useUnicode=true&characterEncoding=utf-8&useSSL=false |

1. 编写javabean

|  |
| --- |
| **public** **class** Student **implements** Serializable{  **private** Integer id;  **private** String name;  **private** Integer age;  //get和set方法  } |

1. 编写mapper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.taiji.springboot.mapper.StudentMapper"*>  <resultMap id=*"BaseResultMap"* type=*"com.taiji.springboot.model.Student"*>  <id column=*"id"* jdbcType=*"INTEGER"* property=*"id"* />  <result column=*"name"* jdbcType=*"VARCHAR"* property=*"name"* />  <result column=*"age"* jdbcType=*"INTEGER"* property=*"age"* />  </resultMap>    <sql id=*"Base\_Column\_List"*>  id, name, age  </sql>    <!-- 查询所有学生 -->  <select id=*"selectAllStudent"* resultMap=*"BaseResultMap"*>  select  <include refid=*"Base\_Column\_List"* />  from student  </select>  <update id=*"updateById"*>  update student  set name = #{name,jdbcType=VARCHAR},  age = #{age,jdbcType=INTEGER}  where id = #{id,jdbcType=INTEGER}  </update>    </mapper> |

1. 编写MyBatis的Mapper接口，并[在接口中添加@Mapper](mailto:4.在MyBatis的Mapper接口中添加@Mapper)注解

|  |
| --- |
| @Mapper  **public** **interface** StudentMapper {  List<Student> selectAllStudent();  **int** updateById(Student student);  } |

1. 编写service接口

|  |
| --- |
| **public** **interface** StudentService {  **public** List<Student> getAllStudent();  **public** **int** update();  } |

1. 编写service接口实现

|  |
| --- |
| @Service  **public** **class** StudentServiceImpl **implements** StudentService {  @Autowired  **private** StudentMapper studentMapper;  @Override  **public** List<Student> getAllStudent() {  List<Student> studentList = studentMapper.selectAllStudent();  **return** studentList;  }  @Transactional  @Override  **public** **int** update() {  Student student = **new** Student();  student.setId(1);  student.setName("李四-update");  student.setAge(18);  **int** update = studentMapper.updateById(student);  System.***out***.println("更新结果："+update);  **int** a = 10 / 0;    **return** update;  }  } |

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** SpringbootdemoApplicationTests {  @Autowired  **private** StudentService studentService;  @Test  **public** **void** contextLoads() {  List<Student> list = studentService.getAllStudent();  **for**(**int** i=0;i<list.size();i++){  Student s = list.get(i);  System.***out***.println(s.getId()+" "+s.getName()+" "+s.getAge());  }  }  } |

1. 编写Controller

|  |
| --- |
| @RestController  **public** **class** MyBatisController {  @Autowired  **private** StudentService studentService;  @GetMapping("/boot/students")  **public** Object students(){  **return** studentService.getAllStudent();  }  @GetMapping("/boot/update")  **public** Object update(){  **return** studentService.update();  }  } |

# springboot事务支持

Springboot使用事务非常简单

1. 在入口类中使用注解@EnableTransactionManagement开启事务支持、
2. 在访问数据库的Service方法上添加注解@Transactional即可

# springboot热部署插件

在实际开发中，我们修改某些代码逻辑功能或页面都需要重启应用，这无形中降低了开发效率

热部署是指当我们修改代码后，服务能自动重启加载新修改的内容，这样大大提高了我们开发的效率

Spring boot热部署通过添加一个插件实现：

插件为：spring-boot-devtools，在Maven中配置如下：

|  |
| --- |
| <!-- springboot开发自动热部署 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  </dependency> |

# springboot使用拦截器

1. 按照Spring mvc的方式编写一个拦截器

|  |
| --- |
| **public** **class** LoginInterceptor **implements** HandlerInterceptor {  @Override  **public** **boolean** preHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o) **throws** Exception {  System.***out***.println("已经进入了登录拦截器......");  //逻辑代码按照之前的方式去编写即可  **return** **true**;  }  @Override  **public** **void** postHandle(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, ModelAndView modelAndView) **throws** Exception {  }  @Override  **public** **void** afterCompletion(HttpServletRequest httpServletRequest, HttpServletResponse httpServletResponse, Object o, Exception e) **throws** Exception {  }  } |

1. 编写一个配置类继承WebMvcConfigurerAdapter类
2. 为该配置类添加@Configuration注解，标注此类为一个配置类，让spring boot扫描到
3. 覆盖其中的方法并添加已经编写好的拦截器

|  |
| --- |
| @Configuration  **public** **class** WebConfig **extends** WebMvcConfigurerAdapter {  @Override  **public** **void** addInterceptors(InterceptorRegistry registry) {  // 需要拦截的路径  String[] addPathPattern = {"/boot/\*\*"};  //不需要拦截的路径  String[] excludePathPattern = {"/boot/hello","/boot/index"};  //注册登录拦截器  registry.addInterceptor(**new** LoginInterceptor()).addPathPatterns(addPathPattern).excludePathPatterns(excludePathPattern);  }  } |

# springboot使用Servlet

方式一通过注解方式实现：

1. 使用Servlet3的注解方式编写一个Servlet

|  |
| --- |
| @WebServlet(urlPatterns="/myServlet")  **public** **class** MyServlert **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = -4134217146900871026L;  @Override  **public** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().print("my servlet hello word");  resp.getWriter().flush();  resp.getWriter().close();  }  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **this**.doGet(req, resp);  }  } |

1. 在main方法的主类上添加注解：

|  |
| --- |
| @ServletComponentScan(basePackages={"com.taiji.springboot.servlet"}) |

方式二通过Spring boot的配置类实现：

1. 编写一个普通的Servlet

|  |
| --- |
| **public** **class** HeServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = -4134217146900871026L;  @Override  **public** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().print("He servlet hello word");  resp.getWriter().flush();  resp.getWriter().close();  }  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  **this**.doGet(req, resp);  }  } |

1. 编写一个springboot的配置类

|  |
| --- |
| @Configuration  **public** **class** ServletConfig {  @Bean  **public** ServletRegistrationBean heServletRegistrationBean(){  ServletRegistrationBean registration = **new** ServletRegistrationBean(**new** HeServlet(),"/heServlet");  **return** registration;  }  } |

# springboot使用Filter

方式一通过注解方式实现：

1. 编写一个Servlet3的注解过滤器

|  |
| --- |
| @WebFilter(urlPatterns="/\*")  **public** **class** MyFilter **implements** Filter {  @Override  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  System.***out***.println("您已进入filter过滤器，您的请求正常，请继续遵规则...");  chain.doFilter(request, response);  }  @Override  **public** **void** destroy() {  }  } |

1. 在main方法的主类上添加注解：

|  |
| --- |
| @ServletComponentScan(basePackages={"com.taiji.springboot.servlet","com.taiji.springboot.filter"}) |

方式二通过spring boot的配置类实现：

1. 编写一个普通的Filter

|  |
| --- |
| **public** **class** HeFilter **implements** Filter {  @Override  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  System.***out***.println("He Filter 您已进入filter过滤器，您的请求正常，请继续遵规则...");  chain.doFilter(request, response);  }  @Override  **public** **void** destroy() {  }  } |

1. 编写一个Springboot的配置类

|  |
| --- |
| @Configuration  **public** **class** ServletConfig {  @Bean  **public** FilterRegistrationBean heFilterRegistration(){  FilterRegistrationBean registration = **new** FilterRegistrationBean(**new** HeFilter());  registration.addUrlPatterns("/\*");  **return** registration;  }  } |

# springboot程序war包部署

1. 编写ServletInitializer继承SpringBootServletInitializer类，覆盖configure方法

|  |
| --- |
| **public** **class** ServletInitializer **extends** SpringBootServletInitializer {  @Override  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {  **return** application.sources(SpringbootdemoApplication.**class**);  }  } |

1. 更新包为war，在pom.xml中修改

|  |
| --- |
| <packaging>war</packaging> |

1. 将内嵌Tomcat依赖的scope改为provided

|  |
| --- |
| <!-- 引入Spring Boot内嵌的Tomcat对JSP的解析包 -->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  <scope>provided</scope>  </dependency> |

1. 配置springboot打包的插件

|  |
| --- |
| <!--springboot打包的插件 -->  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <!-- 打jar包时候，使用该版本 -->  <version>1.4.2.RELEASE</version>  </plugin> |

1. 在项目中通过Maven install在本地maven仓库安装成一个 war包，然后将war包部署到tomcat下运行

# springboot程序打jar包与运行

1. Spring boot程序打包，在pom.xml文件加入如下Spring boot的maven插件：

|  |
| --- |
| <build>  <plugins>  <!--springboot提供的项目编译打包插件 -->  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <!-- 打jar包时候，使用该版本 -->  <version>1.4.2.RELEASE</version>  </plugin>  </plugins>  </build> |

1. 在项目中使用Maven install在本地maven仓库安装成一个jar
2. 使用java –jar 运行第2步生成的jar包，从而可以启动Spring boot程序
3. 访问第3步运行起来的spring boot程序

# springboot集成Redis

Spring boot集成Redis的步骤如下：

1. 在pom.xml中配置相关的jar依赖

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

1. 在Springboot核心配置文件application.properties中配置redis连接信息：

|  |
| --- |
| spring.redis.host=192.168.1.80  spring.redis.port=6379 |

1. 配置了上面的步骤，Spring boot将自动配置RedisTemplate，在需要操作redis的类中注入redisTemplate

在使用的类中注入：

|  |
| --- |
| @Autowired  **private** RedisTemplate<String, String> redisTemplate; |

|  |
| --- |
| @Autowired  **private** RedisTemplate<Object, Object> redisTemplate; |

Spring boot帮我们注入的redisTemplate类，泛型里面只能写<String, String>，<Object, Object>

默认创建的 RedisCacheManager 操作redis的时候使用的是 RedisTemplate<Object, Object>

RedisTemplate<Object, Object> 是 默认使用jdk的序列化机制

1. 修改序列化机制

|  |
| --- |
| @Configuration  **public** **class** MyRedisConfig {  @Bean  **public** RedisTemplate<Object, List<Student>> stuRedisTemplate(  RedisConnectionFactory redisConnectionFactory)  **throws** UnknownHostException {  RedisTemplate<Object, List<Student>> template = **new** RedisTemplate<Object, List<Student>>();  template.setConnectionFactory(redisConnectionFactory);  template.setKeySerializer(**new** StringRedisSerializer());  template.setValueSerializer(**new** GenericJackson2JsonRedisSerializer());  **return** template;  }  } |

1. 使用RedisTemplate操作redis

|  |
| --- |
| **public** List<Student> getAllStudent() {  List<Student> studentList = (List<Student>)stuRedisTemplate.opsForValue().get("allStudents");  **if**(**null** == studentList) {  System.***out***.println("查询的数据库........");  studentList = studentMapper.selectAllStudent();  stuRedisTemplate.opsForValue().set("allStudents", studentList);  }**else**{  System.***out***.println("查询的缓存........");  }  **return** studentList;  } |

# springboot集成elasticsearch

方式一Jest（默认不生效）

1. 在pom.xml中配置相关的jar依赖

|  |
| --- |
| <dependency>  <groupId>io.searchbox</groupId>  <artifactId>jest</artifactId>  <version>5.3.3</version>  </dependency> |

1. 在Springboot核心配置文件application.properties中配置jest的uris：

|  |
| --- |
| spring.elasticsearch.jest.uris=http://192.168.1.80:9200 |

1. 编写自定义的bean，在bean的id上面增加@JestId标签

|  |
| --- |
| **public** **class** Article {  @JestId  **private** Integer id;  **private** String author;  **private** String title;  **private** String content;  //get和set方法  } |

1. 使用jestClient进行操作

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** ElasticdemoApplicationTests {  @Autowired  JestClient jestClient;  @Test  **public** **void** contextLoads() {  //1、给Es中索引（保存）一个文档；  Article article = **new** Article();  article.setId(1);  article.setTitle("好消息");  article.setAuthor("zhangsan");  article.setContent("Hello World");  //构建一个索引功能  Index index = **new** Index.Builder(article).index("taiji").type("news").build();  **try** {  //执行  jestClient.execute(index);  } **catch** (IOException e) {  e.printStackTrace();  }  }  //测试搜索  @Test  **public** **void** search(){  //查询表达式  String json ="{\n" +  " \"query\" : {\n" +  " \"match\" : {\n" +  " \"content\" : \"hello\"\n" +  " }\n" +  " }\n" +  "}";    //构建搜索功能  Search search = **new** Search.Builder(json).addIndex("taiji").addType("news").build();  //执行  **try** {  SearchResult result = jestClient.execute(search);  System.***out***.println(result.getJsonString());  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

方式二SpringData ElasticSearch

1. 在pom.xml中配置相关的jar依赖

|  |
| --- |
| <!--SpringBoot默认使用SpringData ElasticSearch模块进行操作-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  </dependency> |

1. 在Springboot核心配置文件application.properties中配置Client节点信息clusterNodes和clusterName：

|  |
| --- |
| spring.data.elasticsearch.cluster-name=elasticsearch  spring.data.elasticsearch.cluster-nodes=192.168.1.80:9300 |

1. 编写自定义的bean，在bean的id上面增加@Document标签，指定索引和类型

|  |
| --- |
| @Document(indexName = "taiji",type = "book")  **public** **class** Book {  **private** Integer id;  **private** String bookName;  **private** String author;  //get和set方法  @Override  **public** String toString() {  **return** "Book{" +  "id=" + id +  ", bookName='" + bookName + '\'' +  ", author='" + author + '\'' +  '}';  }  } |

方式2-1

1. 编写自己的接口继承ElasticsearchRepository接口，安装一定格式编写自己的方法

|  |
| --- |
| **public** **interface** BookRepository **extends** ElasticsearchRepository<Book,Integer> {  //参照  // https://docs.spring.io/spring-data/elasticsearch/docs/3.0.6.RELEASE/reference/html/  **public** List<Book> findByBookNameLike(String bookName);  } |

1. 使用自己的接口操作ES

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** ElasticdemoApplicationTests {  // @Autowired  // JestClient jestClient;  @Autowired  BookRepository bookRepository;  @Test  **public** **void** test02(){  // Book book = new Book();  // book.setId(1);  // book.setBookName("三国演义");  // book.setAuthor("罗贯中");  // bookRepository.index(book);  **for** (Book book : bookRepository.findByBookNameLike("国")) {  System.***out***.println(book);  }  ;  }  } |

方式2-2

4.使用ElasticsearchTemplate提供的方法操作ES

|  |
| --- |
| @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** ElasticdemoApplicationTests {  @Autowired  **private** ElasticsearchTemplate elasticsearchTemplate;  @Test  **public** **void** test03(){  // Book book= new Book();  // book.setId(2);  // book.setBookName("水浒传");  // book.setAuthor("施耐庵");  //  // IndexQuery indexQuery = new IndexQueryBuilder().withId(book.getId()+"").withObject(book).build();  // elasticsearchTemplate.index(indexQuery);  SearchQuery searchQuery = **new** NativeSearchQueryBuilder().withQuery(QueryBuilders.*boolQuery*()  .must(QueryBuilders.*termQuery*("id", 2))).build();  List<Book> bookList = elasticsearchTemplate.queryForList(searchQuery,Book.**class**);  System.***out***.println(bookList.get(0));  }  } |