## Spring boot 快速入门

### 创建项目

|  |
| --- |
| 1. 在spring 官方页面 生成项目基础框架   <https://start.spring.io/> 在该页面生成项目,解压导入到IDE开发环境   1. 使用 intelliJ IDE 生成   File----->New---->project----->Spring Initializr 填写相应的设置以及需要的组件 (如:Web) |
|  |

### 启动项目

|  |
| --- |
| 1.启动项目:只需要右击启动该类,spring boot 会去启动一个tomcat 端口8080去启动项目    2.也可以使用maven 命令启动: 在项目目录下  mvn spring-boot:run  3.使用maven 打包,启动jar 文件  mvn clean package -Dmaven.test.skip ###打包跳过测试  java -jar girl-0.0.1-SNAPSHOT.jar |

### 配置文件

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 配置文件 application.properties可以换成application.yml  Yml文件是以层级结构构成,可以省略重复写同一父级,如下:  **server:  port:** 8082##端口  **servlet:  context-path:** /girl##上下文  冒号和值必须有个空格分隔.  在配置中调用配置   |  | | --- | | **server:  port:** 8080  **servlet:  context-path:** /girl **cupSize:** A **age:** 15 **content: "cupSize: ${cupSize},age:${age}"** |   在controller中调用配置参数:使用注解@Value(“${参数名}”)   |  | | --- | | **@Value**(**"${cupSize}"**) private String cupSize;  **@Value**(**"${server.port}"**) private String port;  **@Value**(**"${content}"**) private String content;  **@RequestMapping**(value = **"/hello"**,method = RequestMethod.*GET*) public String say(){  return content; } |   将配置文件注入到bean中   |  | | --- | | **girl:  cupSize:** A  **age:** 15 |   Bean文件   |  | | --- | | **@Component @ConfigurationProperties**(prefix = **"girl"**) public class GrilProperties {  private Integer age;  private String cupSize;  public Integer getAge() {  return age;  }  public void setAge(Integer age) {  this.age = age;  }  public String getCupSize() {  return cupSize;  }  public void setCupSize(String cupSize) {  this.cupSize = cupSize;  } } |   使用  **@Autowired** private GrilProperties grilProperties; |

### 注解

|  |
| --- |
| @Controller 和 @RestController均为注解一个controller  @RestController = @ResponseBody + @Controller效果,  @ResponseBody是将返回的对象转换为指定格式,一般用来返回JSON或xml  而@Controller可以返回指定页面,必须有个模板  如:return “hello”,@RestController会返回字符窜,@Controller如果hello配置为页面会返回页面.  @Value ###获取配置文件的值:@Value(“${参数名}”)  @Autowired ###按照类型装配对象,默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false,如果我们想使用按照名称（byName）来装配，可以结合@Qualifier注解一起使用。如下：  @Autowired  @Qualifier("userDao")  private UserDao userDao;  @ConfigurationProperties##把配置文件的信息装配成实体bean  @Component （把普通pojo实例化到spring容器中，相当于配置文件中的  处理请求地址映射的注解,可用在类或方法  @RequestMapping(value = "/hello",method = RequestMethod.GET)  用在类上为父级路径,如下多个url都可访问:  **@RequestMapping**(value = {**"/hello"**,**"/hi"**},method = RequestMethod.*GET*)  http://localhost:8081/girl/h/hello  http://localhost:8081/girl/h/hi  上边method,设置请求的方式,不设置则get/post均可访问,不安全不推荐使用.  @GetMapping(value = "hello")###等同于method设置为get的RequestMapping  @PosttMapping(value = "hello")##等同于method设为post的RequestMapping  获取URL中的数据:http://localhost:8081/girl/hello/333  @PathVariable  **@GetMapping**(value = **"/hello/{id}"**) public String say(**@PathVariable**(**"id"**) Integer id){  return **"id="** + id; }  获取请求中的参数如:http://localhost:8081/girl/hello?id=1  public String say(**@RequestParam**(value = **"id"**) Integer id){  return **"id="** + id; }  如上参数必须有,否则报错,设置参数非必须,默认值  public String say(**@RequestParam**(value = **"id"**,required = false,defaultValue = **"0"**) Integer id){  return **"id="** + id; }  @SessionAttributes注解类 则该类中 往页面传入值key为user 的值会保存在session域中,,,注意:::只要其他方法有如上中map.put(“user”,id)设置均会保存入session中 |
|  |

### 生产环境和开发环境配置不同问题

|  |  |
| --- | --- |
| 如:生产环境端口为80,开发环境8080,等问题  application.yml  application-dev.yml ##开发  application-prod.yml##生产  application.yml中设置使用那种环境   |  | | --- | | **spring:  profiles:  active:** prod |   启动应用时也可指定:  java -jar girl-0.0.1-SNAPSHOT.jar --spring.profiles.active=prod |

### 使用spring data JPA

|  |  |  |
| --- | --- | --- |
| JPA :Java持久化的标准  Spring data JPA:是Spring对JPA的进一步抽象封装进一步减少访问持久层的代码量,默认使用ORM框架为hibernate  通过Spring Data，您可以使用Hibernate、Eclipse Link或任何其他JPA提供程序    添加依赖:   |  | | --- | | <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId> </dependency> <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId> </dependency> |   配置jpa以及数据库   |  | | --- | | **spring:  profiles:  active:** prod  **datasource:  driver-class-name:** com.mysql.cj.jdbc.Driver  **url:** jdbc:mysql://127.0.0.1:3306/dbgirl  **username:** root  **password:** root  **jpa:  hibernate:  ddl-auto:** create  **show-sql:** true |   如上 ddl-auto:create ,设置会每次重启应用删除表,重新根据字段创建  ddl-auto:update:如果映射中有新字段会添加新字段,之前字段保留. |

### data JPA CRUD (简单用法)

|  |  |
| --- | --- |
| 1. 创建接口   public interface GirlRepository extends JpaRepository<Girl,Integer> { }   1. 注入对象并使用  |  | | --- | | **@Autowired** private GirlRepository girlRepository;  **@GetMapping**(value = **"/girls"**) public List<Girl> girlList(){  return girlRepository.findAll(); }  **@PostMapping**(value = **"/girls"**) public Girl girlAdd(**@RequestParam**(value = **"name"**) String name,  **@RequestParam**(value = **"age"**) Integer age){  Girl g = new Girl();  g.setName(name);  g.setAge(age);  return girlRepository.save(g);  } |  1. 测试访问   curl -X GET -i http://localhost:8080/girl/girls  curl -X POST -i http://localhost:8080/girl/girls --data 'name=space11&age=11'  查询一条:  **@GetMapping**(value = **"/girls/{id}"**) public Girl girlFindOne(**@PathVariable**(value = **"id"**) Integer id){  return girlRepository.findById(id).get(); }  删除一条:  **@DeleteMapping**(value = **"/girls/{id}"**) public void deleteGirl(**@PathVariable**(value = **"id"**) Integer id){  girlRepository.deleteById(id); }  修改:  **@PutMapping**(value = **"/girls/{id}"**) public Girl updateGirl(**@PathVariable**(value = **"id"**) Integer id,  **@PathVariable**(value = **"name"**) String name,  **@PathVariable**(value = **"age"**) Integer age){  Girl g = new Girl();  g.setId(id);  g.setName(name);  g.setAge(age);  return girlRepository.save(g); }  按照某字段查询:  定义接口方法:必须写成类似findByXX  public interface GirlRepository extends JpaRepository<Girl,Integer> {  //通过年龄查询  public List<Girl> findByAge(Integer age); }  **@GetMapping**(value = **"/girls/age/{age}"**) public List<Girl> findGirlsByAge(**@PathVariable**(value = **"age"**) Integer age){  return girlRepository.findByAge(age); } |

### 事务

|  |
| --- |
| **@Transactional** public void saveGirls(){  Girl g1 = new Girl();  g1.setAge(11);  g1.setName(**"xxx"**);  girlRepository.save(g1);   Girl g2 = new Girl();  g2.setAge(55);  g2.setName(**"xxx33"**);  girlRepository.save(g2); }  给需要添加事务的方法加@Transactional即可. |

### 表单验证

|  |
| --- |
| 实体类中：  @Min(value = 20,message = "未成年！")  private Integer age;  Controller中：  @ResponseBody  @PostMapping(value = "/girl/save")  public String saveGirl(@Valid Girl girl, BindingResult bindingResult){  if(bindingResult.hasErrors()){  return bindingResult.getFieldError().getDefaultMessage();  }  return "success";  } |

### AOP

|  |
| --- |
| 加入依赖：  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  创建AOP切面类：  @Aspect  @Component  public class LogAop {  private static final Logger logger = LoggerFactory.getLogger(LogAop.class);//内置slf4g  @Pointcut("execution(public \* com.space.controller.HelloController.\*(..))")  public void log(){  }  //上边定义切点，后边可以复用  @Before(value = "log()")  public void beforeLog(){  logger.info("log before...");  }  @After(value = "log()")  public void afterLog(){  logger.info("log after...");  }  }  如下打印请求信息：  @Before(value = "log()")  public void beforeLog(JoinPoint joinPoint){  ServletRequestAttributes attributes = (ServletRequestAttributes)RequestContextHolder.getRequestAttributes();  HttpServletRequest request = attributes.getRequest();  logger.info("url={}",request.getRequestURL());  logger.info("method={}",request.getMethod());//请求方式  logger.info("ip={}",request.getRemoteAddr());//请求地址  logger.info("class\_method={}",joinPoint.getSignature().getDeclaringTypeName() +  "." + joinPoint.getSignature().getName());//请求的类和方法  logger.info("args={}",joinPoint.getArgs());//请求方法的参数  }  返回值打印：  @AfterReturning(pointcut = "log()",returning = "object")//object返回值，和方法参数对应  public void returnning(Object object){  logger.info("response={}",object.toString());  } |
|  |

### 统一异常处理

|  |
| --- |
| **定义同一的返回结果类型（主要针对返回值为json）bean：**  public class ResultBean<T> {  private Integer code;  private String msg;  private T data;  public Integer getCode() {  return code;  }  //...get/setter方法  }  **封装不同返回值情况如：success/error**  public class ResultUtils {  public static ResultBean error(Integer code,String msg){  ResultBean resultBean = new ResultBean();  resultBean.setCode(code);  resultBean.setMsg(msg);  return resultBean;  }  public static ResultBean success(Object o){  ResultBean resultBean = new ResultBean();  resultBean.setCode(0);  resultBean.setMsg("success");  resultBean.setData(o);  return resultBean;  }  public static ResultBean success(){  return success(null);  }  }  **定义自己的业务异常类，如：MyException,此处继承RuntimeException,springboot只能回滚该异常。**  public class MyException extends RuntimeException{  public MyException(Integer code,String message) {  super(message);  this.code = code;  }  private Integer code;  public Integer getCode() {  return code;  }  public void setCode(Integer code) {  this.code = code;  }  }  **异常处理类：所有异常同一再次被重新处理。**  @ControllerAdvice  public class MyExceptionHandler {  private static final Logger logger = LoggerFactory.getLogger(MyExceptionHandler.class);  @ExceptionHandler(value = Exception.class)  @ResponseBody  public ResultBean handler(Exception ex){  if( ex instanceof MyException){  MyException myex = (MyException)ex;  return ResultUtils.error(myex.getCode(),myex.getMessage());  }else {  logger.error("【系统异常】：{}",ex);  return ResultUtils.error(-1,"未知错误！");  }  }  }  **统一的code/msg管理：使用枚举**  public enum ResultEnum {  UNKONW\_ERROR(-1,"未知错误"),  SUCCESS(0,"成功"),  INCON(100,"不符合要求");  private Integer code;  private String msg;  ResultEnum(Integer code, String msg) {  this.code = code;  this.msg = msg;  }  public Integer getCode() {  return code;  }  public String getMsg() {  return msg;  }  }  **Service中使用：**  @Override  public Girl findWomanById(Integer id){  GirlRepository girlRepository = ((GirlRepository)getRepository());  Girl girl = girlRepository.findById(id).get();  if(girl.getAge() < 18){  throw new MyException(ResultEnum.INCON);  }else {  return girl;  }  }  这样当抛出异常时 异常会被MyExceptionHandler 拦截到 并处理符合条件的异常 |

### 测试

|  |
| --- |
| **对业务方法进行测试：**  @RunWith(SpringRunner.class)  @SpringBootTest  public class GirlServiceImplTest {  @Autowired  private GirlService girlService;  @Test  public void findWomanById() throws Exception {  Girl girl = girlService.findWomanById(1);  Assert.assertEquals(new Integer(1),girl.getAge());//断言单元测试  }  }  也可以使用idea 自动生成测试类和方法，如下：GirlServiceImpl类中方法进行测试  类中右击---->go to ---->Test ---->选择创建新测试或已有的测试类    如新建则：选择要进行测试的方法，进行生成    **对API进行测试（Controller方法）**  @RunWith(SpringRunner.class)  @SpringBootTest  @AutoConfigureMockMvc  public class HelloControllerTest {  @Autowired  private MockMvc mvc;  @Test  public void getGirl() throws Exception {  mvc.perform(MockMvcRequestBuilders.get("/girl/1"))  .andExpect(MockMvcResultMatchers.status().isOk())  ;  }  } |

## 微服务

|  |
| --- |
| 拆分原有单体应用，以多模块来组成整个应用，模块之间使用http来进行通信调用，每个模块为一个进程。这样就减少了模块之间的耦合性，但同样给项目构建和运维带来了不便（模块多了），Spring boot解决了项目构建的问题，Spring cloud负责模块之间的互调协调的分布式。 |

## Spring boot 基本设置

### Pom依赖

|  |
| --- |
| 父依赖：  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.9.RELEASE</version>  </parent>  这个父依赖还有一个父依赖（管理版本的）：spring-boot-dependencies  依赖（启动器）：spring-boot-starter-web其实为两个依赖（spring-boot-starter/web）,有不同的需求可能添加不同的启动器：如下 |

### 入口类，主程序类

|  |
| --- |
| @SpringBootApplication  public class HelloWorldMainApplication {  public static void main(String[] args) {  SpringApplication.run(HelloWorldMainApplication.class,args);  }  }  **@SpringBootApplication**注解的类为Spring boot的主配置类，Spring boot就应该运行该类的main方法启动应用。该注解如下为组合注解：  @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @SpringBootConfiguration  @EnableAutoConfiguration  @ComponentScan(  excludeFilters = {@Filter(  type = FilterType.CUSTOM,  classes = {TypeExcludeFilter.class}  ), @Filter(  type = FilterType.CUSTOM,  classes = {AutoConfigurationExcludeFilter.class}  )}  )   * **@SpringBootConfiguration**：Spring boot的配置注解。 * **@EnableAutoConfiguration**：开启自动配置 注解。Spring boot 自动配置。   （可以将启动类【@SpringBootApplication】所在的包和子包中的组件扫描到。导入相应组件并自动配置如：mvc,aop,redis....J2EE中的整合） |

### 快速创建

|  |
| --- |
| 快速创建spring boot项目：   * 从官网导出 * IDEA创建     如上默认创建Spring boot项目并会创建如下资源：    Static:放置静态资源 js/imags/css  Templates:模板引擎，Spring boot由于是内置tomcat所以不支持JSP  Application.properties:应用配置文件。如：端口号，jdbc...... |

### Spring boot 的配置

|  |
| --- |
| 两种配置类型：application.properties或者application.yml  配置文件的作用：修改Spring boot的默认配置。  YAML(不仅仅是一个标记语言):比xml/json等更简洁，以数据为中心。  **server:  port:** 8090  **YAML语法**：   * 【Key:空格value】表示一个键值对，空格不能省略。 * 以**空格**的缩进控制层级关系;左对齐的为同一层级的。 * 属性和值大小写敏感   **值的写法：**   * 字面量：字符串/数字/布尔，一般直接写，字符串默认不用加单引号或双引号。   “”：双引号会转义特殊字符，如：“33333 \n” 输出：33333 换行  ‘’：单引号不会转义特殊字符如上会将 ‘\n’ 原样输出 \n  直接写相当于单引号效果。   * 对象/map   **boy:  name:** space  **age:** 23  **sex:** 0  或者  **boy:** {**name:** space,**age:** 23,**sex:** 0}   * 数组   **boy:  pets:** - cat  - dog  - pig  或者：  **pets:** [cat,pig] |

### 全局配置文件获取/绑定

**第一种：@ConfigurationProperties注解批量注入,默认是从全局配置文件获取数据**

|  |
| --- |
| **person:  name:** xiaoz  **age:** 14  **isKid:** true  **birth:** 2019/11/01  **map:** {**k1:** v1,**k2:** v2}  **list:** - l1  - l2  - l3  **dog:  name:** cc  **age:** 3  将配置文件中的值注入到bean中   * @ConfigurationProperties(prefix = "person")配置文件属性的注解 * @Component将bean加入到容器中 * 加入配置文件处理器的依赖   <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-configuration-processor</artifactId>  <optional>true</optional>  </dependency>  **@Component @ConfigurationProperties**(prefix = **"person"**) public class Person {   private String name;  private int age;  private boolean isKid;  private Date birth;  private Map<String,Object> map;  private List<Object> list;  private Dog dog;   public String getName() {  return name;  }....  }  使用：  **@Autowired** private Person person;  **@Test** public void contextLoads() {  System.*out*.println(person); }  Properties文件如下： |

**第二种：@Value单个注入，不能获取map或数组等复制数据【基本类型数据没问题】**

|  |
| --- |
| **@Value**(**"${person.name}"**) private String name; **@Value**(**"#{11\*2}"**) private int age; **@Value**(**"true"**) private boolean isKid; |

比较：



### 指定配置文件的数据获取

@PropertySource(value = {"classpath:person.properties"})

只能将配置文件加载到内存中，注入到bean中还需要使用@Value或@ConfigurationProperties("person")

|  |
| --- |
| @Component  @PropertySource(value = {"classpath:person.properties"})  @ConfigurationProperties("person")  public class Person {  private String name;  private int age;  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  ...  }  Person.properties：  person.name=space  person.age=100 |

### 导入Spring配置文件

导入Spring外部或额外的配置文件

|  |
| --- |
| 如：bean.xml  <?xml version**="1.0"** encoding**="UTF-8"**?> <beans xmlns**="http://www.springframework.org/schema/beans"** xmlns:xsi**="http://www.w3.org/2001/XMLSchema-instance"** xsi:schemaLocation**="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>   <bean id**="testService"** class**="com.space.service.TestService"**/> </beans>  不做处理是无法在容器中获取的  如下  @Autowired  private ApplicationContext ioc;  @Test  public void testImportReSource() {  boolean flag = ioc.containsBean("testService");  System.out.println(flag);  }  使用@ImportResource(value = {"classpath:bean.xml"})在启动类上  **@ImportResource**(value = {**"classpath:bean.xml"**}) **@SpringBootApplication** public class Springboot02Application {   public static void main(String[] args) {  SpringApplication.*run*(Springboot02Application.class, args);  }   } |

### 使用注解方式给容器中添加组件

|  |
| --- |
| 上边一节使用@importResource方式导入Springboot不推荐，SpringBoot推荐使用注解方式。  使用@Configuration和@Bean  **@Configuration** public class MyConfig {   **@Bean** public TestService testService(){  return new TestService();  } }  如上方法名为bean的id,获取如下：  **@Test** public void testImportReSource() {  TestService testService = (TestService)ioc.getBean(**"testService"**);  System.*out*.println(testService); } |

### 配置文件占位符

配置文件中使用${}占位符

|  |
| --- |
| **person:  a:** cc  **name:** space\_${**random.int(10)**}\_${**person.a**}  **age:** ${**person.c:100**}  ${random.int},${random.int(100)} 占位符中使用随机数    ${person.name}引用之前的配置,${person.name:cc}如果没有默认使用cc |

### Profile多环境配置

|  |
| --- |
| * 多文件：创建如下文件   application.yml，application-dev.yml，application-prod.yml  在主配置文件application.yml中激活   * Yml多文档：在yml中使用 “---” 来分割不同文档如下： * 激活方式 * 在主配置文件中：   spring:  profiles:  active: prod   * 命令行参数：--spring.profiles.active=dev 或者   java -jar --spring.profiles.active=dev   * 虚拟机参数：-Dspring.profiles.active=prod |

### 配置文件的加载位置

**应用内部：**

|  |
| --- |
| 如上file为项目目录，优先级高的会覆盖低的。所有配置文件都会被加载，只是低**优先级被覆盖而已。会形成互补配置。**  **当项目已经打包好后需要临时加配置文件可以在命令中加**--spring.config.location指定  java -jar springboot\_02-0.0.1-SNAPSHOT.jar --spring.config.location=/home/lucifel/big\_data/springboot\_01/test.yml  也是和已有的配置文件互补。  Springboot 打成jar包是不能将上边file:/config/ 以及file:./下的配置打包进jar中。 |

**外部加载位置：**

|  |
| --- |
| * **命令行指定，**多个参数空格隔开如 --aa --bb   java -jar springboot\_02-0.0.1-SNAPSHOT.jar --spring.config.location=/home/lucifel/big\_data/springboot\_01/test.yml --xxx    **优先级有上到下，互相互补。**    Jar包外 > jar包内 |

### 自动配置原理

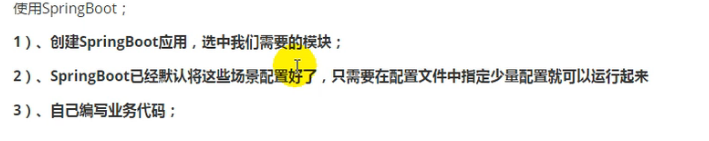
|  |
| --- |
| 配置文件写什么？怎么写？自动配置原理   * Springboot 使用 @EnableAutoConfiguration注解加载jar下META-INF/spring.factories 中# Auto Configure自动配置中的类如：   XXXXAutoConfiguration     * XXXXAutoConfiguration这些配置类从XXXXProperties获取配置信息   XXXXProperties从配置文件中获取相应的配置信息。XXXXAutoConfiguration当满足某些条件【@Condotional注解判断】后会设置组件的信息【XXXXProperties中获得】并new出放入容器中，配置类中配置才生效。    如下在 HttpEncodingProperties中就有配置的项    Springboot 精髓：      自动配置类必须在一定的条件下才能生效，虽然    Jar包中META-INF/spring.factories 中配置了很多组件但不一定都生效。怎样判断哪些生效了？使用springboot 的debug模式。启动应用会有相应的自动配置报告  如下：    控制台如下    Positive matches:启用的  Negative matches:没启用的 |

### Spring boot日志/统一日志

|  |
| --- |
| SLF4J门面实现：log4j(性能低) ,jul（java.util.logging）,Logback(log4j的升级)  Spring boot：  Spring默认使用 **JCL门面**。  Springboot也选用：**SLF4J和logback**  **SLF4J使用：系统开发时，日志记录方式的调用，应该调用抽象类的方法，不应该调用实现层的。**  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class HelloWorld {  public static void main(String[] args) {  Logger logger = LoggerFactory.getLogger(HelloWorld.class);  logger.info("Hello World");  }  }  导入jar:slf4j的jar和logback实现jar    每个日志实现框架都有自己的配置文件。**配置文件用实现框架的配置文件**。  **遗留问题：**  系统A(slf4j+logback)依赖：Spring（commons-logging）,mybatis,不同日志又默认使用不同日志,这时候需要**统一日志**   * 使用一个jar jcl-over-slf4j.jar包替换Spring中的commons-logging jar,其他框架也类似。 * 使用你想用的log日志实现   legacy  Springboot 日志底层依赖：    总结：   * Springboot 底层使用：slf4j+logback记录 * Spring boot把其他日志都替换成slf4j,这些替换包里边实际使用的是slf4j,包名和原jar包一样。      * 如果引入其他框架一定要将框架默认日志依赖移除掉。Spring boot就会使用slf4j+logback统一日志记录。   如：springboot将Spring的commons-logging移除掉了  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <exclusions>  <exclusion>  <groupId>commons-logging</groupId>  <artifactId>commons-logging</artifactId>  </exclusion>  </exclusions> </dependency>  Springboot已经配合好了日志，直接使用,如下：  Logger logger = LoggerFactory.getLogger(getClass());  @Test  public void testLogger(){  logger.trace("trace 日志");  logger.debug("debug日志");  logger.info("info日志");  logger.warn("warn日志");  logger.error("error日志");  }  SpringBoot日志的默认级别为info级别，设置级别如下：  **logging:  level:  com.space:** trace  如上为包的日志级别，如需系统所有则为root的日志级别  **logging:  level:  root:** debug  **file:** springboot.log ###设置日志文件（此处为项目目录下）  还可以设置如下：      日志默认每10m一个文件的滚动输出。  **使用其他日志框架**  只需要在添加对应框架的日志文件即可如下对应关系：除了xx-spring.xml外的配置文件会由各自的框架加载设置。但使用XX-spring.xml则由Springboot加载而且配置文件可以有一些高级功能（推荐）。    如：可以根据profile环境来决定如何设置log的配置如下：这些为**springboot特有的**  <springProfile name="!prod">  *<!-- 非生产环境的配置 -->*  </springProfile>  <springProfile name="prod">  *<!--生产环境配置 -->*  </springProfile>  **切换日志框架**  切换到log4j没有意义，以为logback为log4j的升级。所以切换到log4j2如下：  去掉原有的logging加入log4j2  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <artifactId>spring-boot-starter-logging</artifactId>  <groupId>org.springframework.boot</groupId>  </exclusion>  </exclusions> </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j2</artifactId> </dependency> |

## Spring boot web开发

基本使用步骤：



### Spring boot 映射路径

|  |
| --- |
|  |