## Spring boot

### 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())  ;  }  } |