从零深入SpringBoot

从零深入SpringBoot

一、SpringBoot框架入门

1.1 简介

1.2 SpringBoot的特性

1.3 Spring四大核心

二、SpringBoot入门案例

2.1入口文件配置

2.2核心配置文件

多环境下的核心配置文件的使用

2.3自定义属性配置

配置自定义属性

将自定义属性配置映射到对象

2.4springboot继承jsp

三、SpringBoot框架Web开发

3.1SpringBoot集成MyBatis

3.2 使用集成MyBatis

3.3 SpringBoot事务支持

3.4 SpringBoot下的SpringMVC常见其它注解

@RestController

@GetMapping()&@PostMapping&@DeleteMapping&@PutMapping

3.5 SpringBoot开发RESTful

@PathVariable

3.6 SpringBoot继承Redis

3.6.1 添加操作redis数据类型的依赖

3.6.2 在springboot核心配置文件中添加redis的配置

四、SpringBoot创建java工程

第一种方法--直接获取ConfigurableApplicationContext类

第二种方法 --实现CommandLineRunner，重写run方法

关闭SpringBoot工程的启动logo

五、SpringBoot下使用拦截器

六、SpringBoot下使用Servlet

第一种注解方式：注解方式-->@WebServlet，@ServletComponentScan

第二种注解方式：通过配置类注册

七、SpringBoot下使用过滤器

第一种注解方式：编写过滤器类

第二种注解方式：注册组件

八、SpringBoot下设置字符编码

第一种方式使用characterEncodingFilter

第二种方式springboot字符编码设置(强力推荐)

九、SpringBoot打包

9.1SpringBoot打包war

9.2 SpringBoot打包jar

十、SpringBoot生成日志

十一、SpringBoot集成Thymeleaf

11.1 SpringBoot集成Thymeleaf

11.2 Thymeleaf关闭页面缓存

11.3 标准变量和选择变量表达式

标准变量表达式---th:text="${user.getId()}

11.3 Thymeleaf路径表达式

11.4 Thymeleaf循环遍历list集合

11.5 Thymeleaf循环遍历Map集合

11.6 Thymeleaf循环遍历数组

11.7 Thymeleaf条件判断
11.8 Thymeleaf内联表达式
11.9 Thymeleaf自变量
11.10 Thymeleaf数学运算
11.11 Thymeleaf基本表达式对象
Session对象
获取路径
11.12 Thymeleaf功能表达式对象

一、SpringBoot框架入门

1.1 简介

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SpringMVC +Spring + MyBatis框架进行开发的过程。

在以往我们采用SpringMVC + 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，完全抛弃了繁琐的xml配置过程，采用大量的默认配置简化我们的开发过程。

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。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。

1.2 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方法启动内联的Tomcat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. 自动化配置，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

1.3 Spring四大核心

自动配置

起步依赖

Actuator

命令行界面

二、SpringBoot入门案例

2.1入口文件配置

SpringBoot项目启动文件Application.java

```
//SpringBoot项目启动入口类
@SpringBootApplication//Springboot核心注解，主要用于开发spring自动配置
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

}
```

程序代码文件

开发者自己编写的文件全部放在src/main/java/包名/

```
@Controller
public class IndexController {
    @RequestMapping(value = "/some.do")
    @ResponseBody
    public String doSome(){
        return "someDo";
    }
}
```

2.2核心配置文件

核心配置文件properties文件 放在src/main/resources/application.properties

只能有一个核心文件

```
#设置内嵌Tmocat端口号
server.port=8081

#设置上下文根
server.servlet.context-path=/springboot
```

核心配置文件yml文件 放在src/main/resources/application.yml

采用层级关系，和properties二者其一

```
server:
  port: 8082
  address:
  servlet:
    context-path: /springboot
```

properties和yml同时存在时，优先级取properties>yml

多环境下的核心配置文件的使用

application.properties

```
#主核心配置文件
#激活使用的配置文件
spring.profiles.active=test
```

application-dev.properties

```
#开发环境的配置文件
server.port=8080
server.servlet.context-path=/dev
```

application-production.properties

```
#生产环境的配置文件application-dev.properties
server.port=8083
server.servlet.context-path=/production
```

application-ready.properties

```
#准生产环境的配置文件
server.port=8082
server.servlet.context-path=/ready
```

application-test.properties

```
#测试环境的配置文件
server.port=8081
server.servlet.context-path=/test
```

yml原理和properties一样

```
#主核心配置文件
#激活使用的配置文件
spring:
  profiles:
    active: dev
.....
```

2.3自定义属性配置

配置自定义属性

1. 在properties文件中配置自定义属性

```
server.port=8081
#自定义属性
his.name=liuqiang
his.age=18
```

2. Controller类中去使用

```java
@Controller
public class IndexController {

    @Value("${his.name}")
    private String name;

    @RequestMapping(value = "/say")
    @ResponseBody
    public String say(){
        return "你好,"+name;
    }
}
```

将自定义属性配置映射到对象

必须得有前缀，shool.age可以，age就不行

1. **在properties文件中配置自定义属性**

```properties
server.port=8081
#自定义属性
school.name = anxingong
school.age = 20
```

2. **创建同名School类**

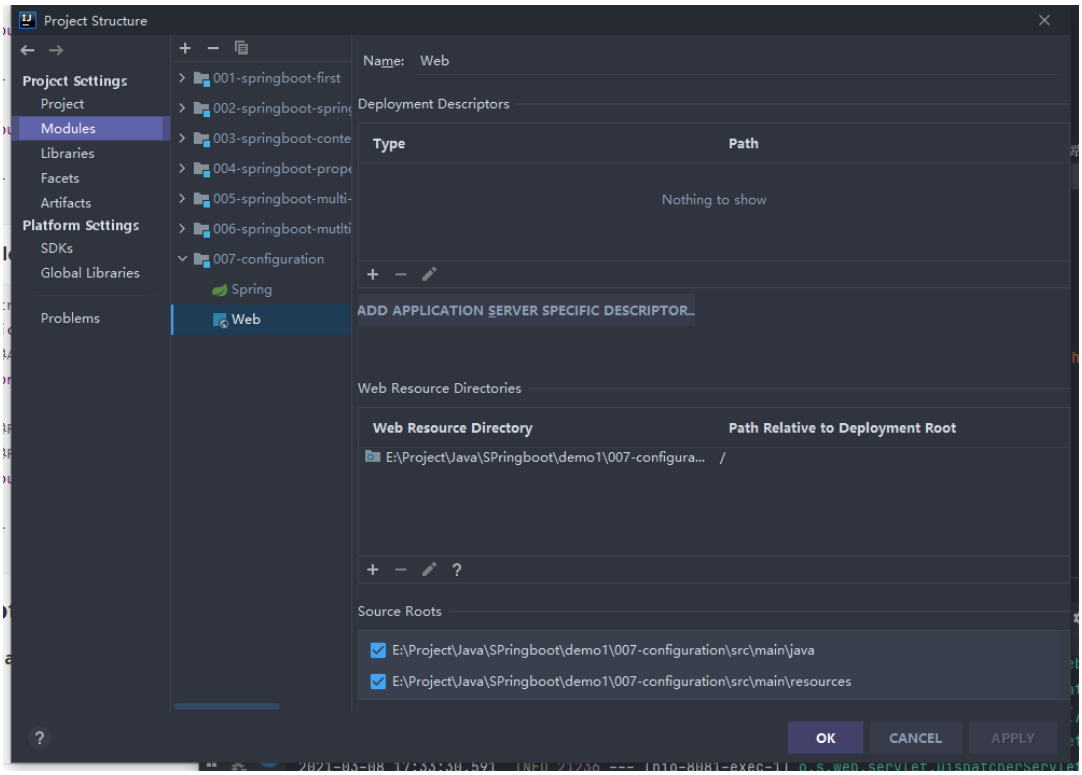
```java
@Component//将此类交给spring容器进行管理
@ConfigurationProperties(prefix="school")
public class School {
    private String name;
    private String age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAge() {
        return age;
    }

    public void setAge(String age) {
        this.age = age;
    }
}
```

3. **Contoller去配置映射**

```java
@Controller
public class IndexController {
    @Autowired//自动装配
    private School school;

    @RequestMapping("/school")
    @ResponseBody
    public String getSchool(){
        return "学校名称："+school.getName()+"，办学时间："+school.getAge();
    }
}
```

2.4springboot继承jsp

1. **在mian文件夹下建立webapp目录，并将其设置为web文件夹，jsp文件就存放于此**



2. **引入SpringBoot内嵌Tomcat对jsp的解析依赖**

```xml
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <version>10.0.2</version>
</dependency>
```

3. **配置编译位置**

```xml
<!--springboot项目默认推荐使用的前端引擎时thyemeleaf
    现在我们使用springboot集成jsp,手动指定jsp最后编译的路径
    而且编译jsp的路径是规定好的 META-INF/resources
-->
<resources>
    <resource>
        <!--源文件夹-->
        <directory>src/main/webapp</directory>
```

```
        <!--指定编译到META-INF/resources-->
        <targetPath>META-INF/resources</targetPath>
        <!--指定源文件夹中那个资源要编译进行-->
        <includes>
            <include>*.*</include>
        </includes>
    </resource>
</resources>
```

4. **配置视图解析器**

   properties文件

```
spring.mvc.view.prefix=/
spring.mvc.view.suffix=.jsp
```

5. **编写Controller类**

```
@RequestMapping("/doSome")
@ResponseBody
public ModelAndView doSome(){
    ModelAndView modelAndView = new ModelAndView();
    modelAndView.addObject("msg","这是show页面");
    modelAndView.setViewName("show");
    return modelAndView;
}
```

三、SpringBoot框架Web开发

3.1SpringBoot集成MyBatis

1. **添加mybatis依赖，MySQL驱动**

```
<!--MySQL驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<!--Mybatis整合SpringBoot框架的起步依赖-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.4</version>
</dependency>
```

2. **使用MyBatis提供的逆向工程生成实体Bean,映射文件，Dao接口**

   编写GeneratorMapper.xml

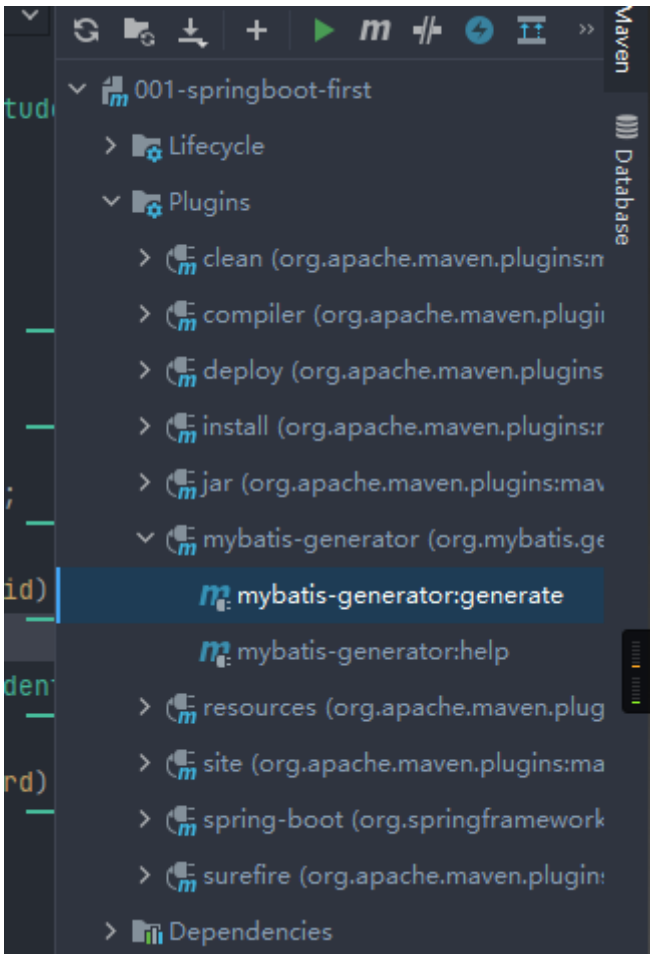
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE generatorConfiguration
        PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration
1.0//EN"
        "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
<generatorConfiguration>
    <!-- 指定连接数据库的 JDBC 驱动包所在位置，指定到你本机的完整路径 -->
```

```
    <classPathEntry location="E:\mysql-connector-java-5.1.38.jar"/>
    <!-- 配置 table 表信息内容体，targetRuntime 指定采用 MyBatis3 的版本 -->
    <context id="tables" targetRuntime="MyBatis3">
        <!-- 抑制生成注释，由于生成的注释都是英文的，可以不让它生成 -->
        <commentGenerator>
            <property name="suppressAllComments" value="true" />
        </commentGenerator>
        <!-- 配置数据库连接信息 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"


 connectionURL="jdbc:mysql://localhost:3306/springboot?
serverTimeZone=UTC"
                        userId="root"
                        password="Lq060528">
        </jdbcConnection>
        <!-- 生成 model 类，targetPackage 指定 model 类的包名，
targetProject 指定
       生成的 model 放在 eclipse 的哪个工程下面-->
        <javaModelGenerator
targetPackage="com.qiangliu8.springboot.model"
                            targetProject="src/main/java">
            <property name="enableSubPackages" value="false" />
            <property name="trimStrings" value="false" />
        </javaModelGenerator>
        <!-- 生成 MyBatis 的 Mapper.xml 文件，targetPackage 指定 mapper.xml
文件的
       包名， targetProject 指定生成的 mapper.xml 放在 eclipse 的哪个工程下面
-->
        <sqlMapGenerator targetPackage="com.qiangliu8.springboot.mapper"
                         targetProject="src/main/java">
            <property name="enableSubPackages" value="false" />
        </sqlMapGenerator>
        <!-- 生成 MyBatis 的 Mapper 接口类文件,targetPackage 指定 Mapper 接口
类的包
       名， targetProject 指定生成的 Mapper 接口放在 eclipse 的哪个工程下面 -->
        <javaClientGenerator type="XMLMAPPER"

targetPackage="com.qiangliu8.springboot.mapper"
targetProject="src/main/java">
            <property name="enableSubPackages" value="false" />
        </javaClientGenerator>
        <!-- 数据库表名及对应的 Java 模型类名 -->
        <table tableName="t_student" domainObjectName="Student"
               enableCountByExample="false"
               enableUpdateByExample="false"
               enableDeleteByExample="false"
               enableSelectByExample="false"
               selectByExampleQueryId="false"/>
    </context>
</generatorConfiguration>
```

在 pom.xml 文件中添加 mysql 反向工程依赖

```
<!--mybatis 代码自动生成插件-->
<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.3.6</version>
    <configuration>
        <!--配置文件的位置-->
        <configurationFile>GeneratorMapper.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>
</plugin>
```

3. **点击maven的mybatis-generator:generate按钮，生成mapper，dao,xml文件**



4. **查看mapper，dao,xml文件**

   Student.java

```
package com.qiangliu8.springboot.model;

public class Student {
    private Integer id;

    private String name;

    private Integer age;

    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 Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

StudentMapper.java

```
package com.qiangliu8.springboot.mapper;

import com.qiangliu8.springboot.model.Student;

public interface StudentMapper {
    //根据主键删除记录
    int deleteByPrimaryKey(Integer id);
    //插入记录
    int insert(Student record);
    //选择性插入
    int insertSelective(Student record);
    //根据主键查询记录
    Student selectByPrimaryKey(Integer id);
    //根据主键选择性更新对象
    int updateByPrimaryKeySelective(Student record);
    //根据主键更新对象
    int updateByPrimaryKey(Student record);
}
```

Student.xml

```
<?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.qiangliu8.springboot.mapper.StudentMapper">

  <!--ResultMap作用：
  1当数据库中字段名称与实体类对象的属性名不一致时，可以进行转换
  2当前查询的结果没有一个对象一个表的时候，可以自定义一个结果集-->
  <resultMap id="BaseResultMap"
type="com.qiangliu8.springboot.model.Student">
  <!--   id标签只能修饰主键字段
```

```
        result标签除主键外其他字段
  -->
  <!--    column数据库中的字段名称
          property 映射对象的属性名称
          jdbcType列中数据库中字段的类型（可以忽略）
  -->
<!--    如果数据库中字段名称由多个单词构成，通过MyBatis逆向工程生成的对象属性会按照驼峰命名法规则生成属性名称
    其中:数据库中字段名称由多个单词构成的时候必须使用_下划线分隔-->
    <id column="id" jdbcType="INTEGER" property="id" />
    <result column="name" jdbcType="VARCHAR" property="name" />
    <result column="age" jdbcType="INTEGER" property="age" />
  </resultMap>

  <!--sql语句片段，将公告部分提取出来开
      通过inclue标签引用sql语句片段
  -->
  <sql id="Base_Column_List">
    id, name, age
  </sql>
  <select id="selectByPrimaryKey" parameterType="java.lang.Integer"
resultMap="BaseResultMap">
    select
    <include refid="Base_Column_List" />
    from t_student
    where id = #{id,jdbcType=INTEGER}
  </select>
  <delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">
    delete from t_student
    where id = #{id,jdbcType=INTEGER}
  </delete>
  <insert id="insert"
parameterType="com.qiangliu8.springboot.model.Student">
    insert into t_student (id, name, age
      )
    values (#{id,jdbcType=INTEGER}, #{name,jdbcType=VARCHAR}, #
{age,jdbcType=INTEGER}
      )
  </insert>
  <insert id="insertSelective"
parameterType="com.qiangliu8.springboot.model.Student">
    insert into t_student
    <trim prefix="(" suffix=")" suffixOverrides=",">
      <if test="id != null">
        id,
      </if>
      <if test="name != null">
        name,
      </if>
      <if test="age != null">
        age,
      </if>
    </trim>
    <trim prefix="values (" suffix=")" suffixOverrides=",">
      <if test="id != null">
        #{id,jdbcType=INTEGER},
      </if>
      <if test="name != null">
```

```
        #{name,jdbcType=VARCHAR},
      </if>
      <if test="age != null">
        #{age,jdbcType=INTEGER},
      </if>
    </trim>
  </insert>
  <update id="updateByPrimaryKeySelective"
parameterType="com.qiangliu8.springboot.model.Student">
    update t_student
    <set>
      <if test="name != null">
        name = #{name,jdbcType=VARCHAR},
      </if>
      <if test="age != null">
        age = #{age,jdbcType=INTEGER},
      </if>
    </set>
    where id = #{id,jdbcType=INTEGER}
  </update>
  <update id="updateByPrimaryKey"
parameterType="com.qiangliu8.springboot.model.Student">
    update t_student
    set name = #{name,jdbcType=VARCHAR},
      age = #{age,jdbcType=INTEGER}
    where id = #{id,jdbcType=INTEGER}
  </update>
</mapper>
```

5. StudentMapper.java

```
package com.qiangliu8.springboot.mapper;

import com.qiangliu8.springboot.model.Student;

public interface StudentMapper {
    //根据主键删除记录
    int deleteByPrimaryKey(Integer id);
    //插入记录
    int insert(Student record);
    //选择性插入
    int insertSelective(Student record);
    //根据主键查询记录
    Student selectByPrimaryKey(Integer id);
    //根据主键选择性更新对象
    int updateByPrimaryKeySelective(Student record);
    //根据主键更新对象
    int updateByPrimaryKey(Student record);
}
```

6. Student.xml

```
<?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.qiangliu8.springboot.mapper.StudentMapper">
```

```
  <!--ResultMap作用：
  1当数据库中字段名称与实体类对象的属性名不一致时，可以进行转换
  2当前查询的结果没有一个对象一个表的时候，可以自定义一个结果集-->
  <resultMap id="BaseResultMap"
type="com.qiangliu8.springboot.model.Student">
  <!--   id标签只能修饰主键字段
         result标签除主键外其他字段
  -->
  <!--    column数据库中的字段名称
          property 映射对象的属性名称
          jdbcType列中数据库中字段的类型（可以忽略）
  -->
    <id column="id" jdbcType="INTEGER" property="id" />
    <result column="name" jdbcType="VARCHAR" property="name" />
    <result column="age" jdbcType="INTEGER" property="age" />
  </resultMap>

  <!--sql语句片段，将公告部分提取出来开
      通过inclue标签引用sql语句片段
  -->
  <sql id="Base_Column_List">
    id, name, age
  </sql>
  <select id="selectByPrimaryKey" parameterType="java.lang.Integer"
resultMap="BaseResultMap">
    select
    <include refid="Base_Column_List" />
    from t_student
    where id = #{id,jdbcType=INTEGER}
  </select>
  <delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">
    delete from t_student
    where id = #{id,jdbcType=INTEGER}
  </delete>
  <insert id="insert"
parameterType="com.qiangliu8.springboot.model.Student">
    insert into t_student (id, name, age
      )
    values (#{id,jdbcType=INTEGER}, #{name,jdbcType=VARCHAR}, #
{age,jdbcType=INTEGER}
      )
  </insert>
  <insert id="insertSelective"
parameterType="com.qiangliu8.springboot.model.Student">
    insert into t_student
    <trim prefix="(" suffix=")" suffixOverrides=",">
      <if test="id != null">
        id,
      </if>
      <if test="name != null">
        name,
      </if>
      <if test="age != null">
        age,
      </if>
    </trim>
    <trim prefix="values (" suffix=")" suffixOverrides=",">
```

```
      <if test="id != null">
        #{id,jdbcType=INTEGER},
      </if>
      <if test="name != null">
        #{name,jdbcType=VARCHAR},
      </if>
      <if test="age != null">
        #{age,jdbcType=INTEGER},
      </if>
    </trim>
  </insert>
  <update id="updateByPrimaryKeySelective"
parameterType="com.qiangliu8.springboot.model.Student">
    update t_student
    <set>
      <if test="name != null">
        name = #{name,jdbcType=VARCHAR},
      </if>
      <if test="age != null">
        age = #{age,jdbcType=INTEGER},
      </if>
    </set>
    where id = #{id,jdbcType=INTEGER}
  </update>
  <update id="updateByPrimaryKey"
parameterType="com.qiangliu8.springboot.model.Student">
    update t_student
    set name = #{name,jdbcType=VARCHAR},
      age = #{age,jdbcType=INTEGER}
    where id = #{id,jdbcType=INTEGER}
  </update>
</mapper>
```

加一个全局注解就可以不需要一个加@Mapper了

```
@SpringBootApplication
@MapperScan(basePackages = "com.qiangliu8.springboot.mapper")//开启扫描
Mapper接口的包以及子目录
public class SpringbootMybatisApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootMybatisApplication.class, args);
    }
}
```

将mapper文件放入Resources文件夹下，再配置

```
mybatis.mapper-locations=classpath:mapper/*.xml
```

3.2 使用集成MyBatis

1. **在build先将xml指定编译文件**

   因为java文件夹下默认只编译java文件

```
<build>
    <!--手动指定文件夹为Resources-->
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
    </resource>
    </resources>
```

2. **将StudentMapper添加注解**

```
@Mapper
public interface StudentMapper {
    int deleteByPrimaryKey(Integer id);

    int insert(Student record);

    int insertSelective(Student record);

    Student selectByPrimaryKey(Integer id);

    int updateByPrimaryKeySelective(Student record);

    int updateByPrimaryKey(Student record);
}
```

3. 创建StudentService接口，创建该实现类。并且添加注解，放入容器中。

```
@Service
public class StudentServiceImpl implements StudentService{

    @Autowired
    private StudentMapper studentMapper;
    @Override
    public Student queryStudentById(Integer id) {
        return studentMapper.selectByPrimaryKey(id);
    }

}
```

4. Controller调用service方法

```
@Controller
public class StudentController {

    @Autowired
    private StudentServiceImpl studentService;

    @RequestMapping("/student")
    @ResponseBody
    public Object student(){
        return studentService.queryStudentById(1);
    }
}
```

3.3 SpringBoot事务支持

事务是一个完整的功能，也叫做是一个完整的业务。

事务只跟什么SQL语句有关系：DML(增删改)

DML,DQL,TCL,DCL

开启事务添加注解@Transactional

```
@Override
@Transactional
public int updateStudent(Student student) {
    int i = studentMapper.updateByPrimaryKey(student);
    int a = 10/0;
    return i;
}
```

@EnableTransactionManagement//开启事务（可选项）

```
@SpringBootApplication
@MapperScan(basePackages = "com.qiangliu8.springboot.mapper")//开启扫描Mapper接口的包以及子目录
@EnableTransactionManagement//开启事务（可选项）
public class SpringbootMybatisApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootMybatisApplication.class, args);
    }

}
```

3.4 SpringBoot下的SpringMVC常见其它注解

@RestController

意味着当前控制层类中所有方法返还的都是JSON对象

```
@RestController
public class StudentController {

    @RequestMapping(value = "/stu")
    //代替@Controller,且不需要再添加@ResponseBody
    public Object returnStu(){
        return new Student(1,"刘强",18);
    }
}
```

@GetMapping()&@PostMapping&@DeleteMapping&@PutMapping

GetMapping()该注解通过在查询数据的时候使用->查询

```
//RequestMapping(value = "/getStu",method = RequestMethod.GET)
@GetMapping("/getStu")//相当于上一句话,只接收GET请求,如果请求方式不对会报405错调
public Object getStu(Integer id,String name){
    Student student = new Student(id,name,null);
    return student;
}
```

PostMapping该注解通过在新增数据的时候使用->新增

```
//RequestMapping(value = "/sendStu",method = RequestMethod.POST)
@PostMapping("/sendStu")//相当于上一句话,只接收Post请求,如果请求方式不对会报405错调
public Object sendStu(Integer id,String name){
    Student student = new Student(id,name,null);
    return student;
}
```

DeleteMapping该注解通常在删除数据的时候使用->删除

```
//RequestMapping(value = "/deleteStu",method = RequestMethod.DELETE)
@DeleteMapping("/deleteStu")///该注解通常在删除数据的时候使用->删除
public Object deleteStu(){
    return "deleteStu";
}
```

PutMapping该注解通常在删除数据的时候使用->更新

```
//RequestMapping(value = "/putStu",method = RequestMethod.PUT)
@PutMapping("/putStu")///
public Object putStu(Integer id,String name){
    Student student = new Student(id,name,null);
    int updateCount = studentService.updateStudent(student);
    return "id: "+id+"修改结果为: "+updateCount;
}
```

3.5 SpringBoot开发RESTful

一种互联网软件课构设计的风格，但它并不是标准，它只是提出了一组客户端和服务器交互时的架构理念和设计原则,基于这种理念和原则设计的接口可以更简洁,更有层次,REST这个词，是Roy Thomas Fielding在他2000年的博士论文中提出的。

任何的技术都可以实现这种理念，如果一个架构符合REST 原则，就称它为RESTFul架构

比如我们要访问一个http接口: http://localhost:8080/boot/order?id=1021&status=1

采用RESTFul风格则http地址为: http://localhost:8080/boot/order/1021/1

SpringBoot开发REST主要是几个注解实现

@PathVariable

```
@RequestMapping(value = "/stu")
public String student(Integer id,Integer age){
    Student student = new Student(id,"参数创建名",age);
    return String.valueOf(student);
}
//  http://localhost:8080/stu?id=1&age=13
@RequestMapping(value = "/stu2/{id}/{age}")
public String student2(@PathVariable("id")Integer id,@PathVariable Integer age){
    Student student = new Student(id,"参数创建名",age);
    return String.valueOf(student);
}
//  http://localhost:8080/stu2/1/13
```

如果出现路径混淆的情况，使用GetMapping和PostMapping等来区分

```
@GetMapping(value = "/stu3/{id}/{age}")
public String student3(@PathVariable("id")Integer id,@PathVariable Integer age){
    Student student = new Student(id,"参数创建名",age);
    return String.valueOf(student);
}
@PostMapping(value = "/stu3/{age}/{id}")
public String student4(@PathVariable("id")Integer id,@PathVariable Integer age){
    Student student = new Student(id,"参数创建名",age);
    return String.valueOf(student);
}
```

3.6 SpringBoot继承Redis

3.6.1 添加操作redis数据类型的依赖

3.6.2 在springboot核心配置文件中添加redis的配置

四、SpringBoot创建java工程

第一种方法--直接获取ConfigurableApplicationContext类

```
@SpringBootApplication
public class JavaApplication {

    public static void main(String[] args) {
        /*SpringBoot程序启动后，返回值是ConfigurableApplicationConext,它也是Spring容器
        * 相当于原来Spring启动容器ClassPathxmlApplicationContext
        * */

        //获取SpringBoot容器
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(JavaApplication.class, args);

        //从Spring容器中获取Bean对象k
        StudentService service = (StudentService)
applicationContext.getBean("studentService");

        System.out.println(service.sayHello());
    }
```

```
}
```

第二种方法 --实现CommandLineRunner，重写run方法

```
@SpringBootApplication
public class JavaApplication implements CommandLineRunner{

    @Autowired
    private StudentService service;


    public static void main(String[] args) {
        //SpringBoot启动程序，会初始化Spring容器
        SpringApplication.run(JavaApplication.class, args);

    }

    @Override
    public void run(String... args) throws Exception {
        //调用业务方法
        String str = service.sayHello();
        System.out.println(str);
    }
}
```

关闭SpringBoot工程的启动logo

```
@SpringBootApplication
public class CloselogoApplication {

    public static void main(String[] args) {
        //获取入口SpringBoot类
        SpringApplication springApplication = new
SpringApplication(CloselogoApplication.class);
        //设置它的属性
        springApplication.setBannerMode(Banner.Mode.OFF);

        springApplication.run(args);
    }

}
```

也可在Resource目录下添加Banner.txt

```
//          _                    _ _         ___
//    __ _(_) __ _ _ __    __ _| (_)_   _ ( _ )
//   / _` | |/ _` | '_ \ / _` | | | | | |/ _ \
//  | (_| | | (_| | | | | (_| | | | |_| | (_) |
//   \__, |_|\__,_|_| |_|\__, |_|_|\__,_|\___/
//      |_|              |___/
```

五、SpringBoot下使用拦截器

1. **定义一个拦截器，实现HandlerInterceptor接口**

```
public class UserIntercrptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        System.out.println("进入拦截器");
        //编写业务拦截规则
        //从Session中获取用户信息
        User user = (User) request.getSession().getAttribute("user");

        if (user==null){

 response.sendRedirect(request.getContextPath()+"/user/error");
            return false;
        }
        return true;
    }
}
```

2. **创建一个配置类，即在springMVC配置文件中使用mvc:interceptors标签**

```
@Configuration//定义此类为配置类
public class InterceptorConfig implements WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {

        //要拦截user下的所有访问请求,必须用户登录后才可访问,
        //但是这样拦截的路径中有一些是不需要用户登录也可访问的
        String[] addPathPatterns = {
            "/user/**"
        };
        //要省略的路径
        String[] excludePathPatterns = {
            "/user/out","/user/error","/user/login"
        };
        //注册拦截器
        registry.addInterceptor(new
UserIntercrptor()).addPathPatterns(addPathPatterns).excludePathPatterns(e
xcludePathPatterns);
    }
}
```

3. **创建Controller类**

```
@RestController
@RequestMapping(value = "/user")
public class UserController {
    @RequestMapping(value = "/login")
    public Object login(HttpServletRequest request){
        //将用户信息存入session中
        request.getSession().setAttribute("user",new User(1001,"刘强"));
        return  "login success";
    }

    @RequestMapping(value = "/center")
```

```
    public Object center(HttpServletRequest request){
        return "see center";
    }

    @RequestMapping(value = "/out")
    public Object out(HttpServletRequest request){
        return "see out";
    }
    @RequestMapping(value = "/error")
    public Object error(HttpServletRequest request){
        return "see error";
    }
}
```

六、SpringBoot下使用Servlet

1. 创建一个Servlet需要去继承HttpServlet
2. 在web.xml配置文件中使用servlet servlet-mapping

第一种注解方式：注解方式-->@WebServlet，@ServletComponentScan

编写servlet类，添加注解@WebServlet，配置访问路径

```
@WebServlet("/springboot_servlet")
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.getWriter().println("MyServlet-springboot");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

开启扫描servlet注解

```
@SpringBootApplication//开启扫描spring
@ServletComponentScan(basePackages = "com.qiangliu8.springboot.servlet")
public class ServletApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServletApplication.class, args);
    }

}
```

第二种注解方式：通过配置类注册

编写servlet类

```
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.getWriter().println("MyServlet-springboot");
        resp.getWriter().flush();
        resp.getWriter().close();
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

编写配置类

```
@Configuration//该注解将此类定义配置类（相当于xml文件）
public class ServletConfig {
    //@Bean是一个方法上级别的注解，主要用在配置类型上
    //相当于
 /*   <beans>
        <bean id="" class=""/>
    </beans>
*/
    @Bean
    public ServletRegistrationBean myRegistrationBean(){
        ServletRegistrationBean servletRegistrationBean = new
ServletRegistrationBean(new MyServlet(),"/springboot_servlet");

        return servletRegistrationBean;
    }
}
```

七、SpringBoot下使用过滤器

第一种注解方式：编写过滤器类

编写过滤器类

```
@WebFilter(urlPatterns = "/myfilter")
public class MyFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("已经进入过滤器");
        filterChain.doFilter(servletRequest,servletResponse);
    }
}
```

开启扫描servlet注解

```
@SpringBootApplication//开启扫描spring
@ServletComponentScan(basePackages = "com.qiangliu8.springboot.filter")
public class ServletApplication {

    public static void main(String[] args) {
        SpringApplication.run(ServletApplication.class, args);
    }

}
```

第二种注解方式：注册组件

编写过滤器类

```
public class MyFilter implements Filter {
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
servletResponse, FilterChain filterChain) throws IOException, ServletException {
        System.out.println("已经进入过滤器");
        filterChain.doFilter(servletRequest,servletResponse);
    }
}
```

编写配置类

```
@Configuration
public class FilterConfig {

    @Bean
    public FilterRegistrationBean myFilterRegistrationBean(){
        FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean(new MyFilter());

        filterRegistrationBean.addUrlPatterns("/user/*");

        return filterRegistrationBean;
    }
}
```

八、SpringBoot下设置字符编码

第一种方式使用characterEncodingFilter

```
@WebServlet("/springboot_servlet")
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        resp.getWriter().println("欢迎使用springboot框架");
        resp.setContentType("text/html;charset=utf-8");
        resp.getWriter().flush();
        resp.getWriter().close();
```

```
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        super.doPost(req, resp);
    }
}
```

```
@Configuration
public class SystemConfig {
    @Bean
    public FilterRegistrationBean characterEncodingBean(){
        //创建字符编码过滤器
        CharacterEncodingFilter characterEncodingFilter = new
CharacterEncodingFilter();
        //设置强制使用指定字符编码
        characterEncodingFilter.setForceEncoding(true);
        //设置指定字符编码
        characterEncodingFilter.setEncoding("utf-8");


        FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean();
        //设置字符编码过滤器
        filterRegistrationBean.setFilter(characterEncodingFilter);
        //设置字符编码过滤器路径
        filterRegistrationBean.addUrlPatterns("/*");

        return  filterRegistrationBean;
    }
}
```

在资源文件中关闭springboot的http的字符编码支持

```
#关闭springboot的http的字符编码支持 只有关闭我们设置的才生效
server.servlet.encoding.enabled=false
```

第二种方式springboot字符编码设置(强力推荐)

```
server.servlet.encoding.enabled=true
server.servlet.encoding.force=true
server.servlet.encoding.charset=utf-8
```

九、SpringBoot打包

9.1SpringBoot打包war

1. **指定打包方式和打包名**

```
<packaging>war</packaging>


<build>
    <!--指定打war包的字符-->
    <finalName>SpringBootWar</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

2. **启动类上构建新资源**

```
@SpringBootApplication//开启扫描spring
@ServletComponentScan(basePackages = "com.qiangliu8.springboot.servlet")
public class ServletApplication extends SpringBootServletInitializer {

    public static void main(String[] args) {
        SpringApplication.run(ServletApplication.class, args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
builder) {
        //参数为当前Springboot启动类
        //构建新资源
        return builder.sources(ServletApplication.class);
    }
}
```

3. package打包 放到tomcat的webapps目录下，启动bin目录的startup.bat
4. 访问http://localhost:8080/SpringBootWar/xxx.jsp

9.2 SpringBoot打包jar

1. **指定打包方式和打包名**

```
<packaging>jar</packaging>


<build>
    <resources>
        <resource>
            <directory>src/main/webapp</directory>
            <targetPath>META-ING/resources</targetPath>
            <includes>
                <include>*.*</include>
            </includes>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
```

```
            <includes>
                <include>**/*.*</include>
            </includes>
        </resource>
    </resources>
    <!--指定打jar包的字符-->
    <finalName>SpringBoot</finalName>
</build>
```

2. **application.properties上配置任意端口**

```
server.port=8081
```

3. **package打包 生成springboot.jar 。在该文件的目录下cmd 跑命令java -jar springboot.jar**
4. 访问http://localhost:8081/SpringBoot/xxx.jsp

十、SpringBoot生成日志

控制台打印日志

创建日志配置

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 日志级别从低到高分为
TRACE < DEBUG < INFO < WARN < ERROR < FATAL，
如果设置为 WARN，则低于 WARN 的信息都不会输出 -->
<!--
scan:当此属性设置为 true 时，配置文件如果发生改变，将会被重新加载，默认值为true
-->
<!-- scanPeriod:设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认
单位是毫秒。当 scan 为 true 时，此属性生效。默认的时间间隔为 1 分钟。 -->
<!-- debug:当此属性设置为 true 时，将打印出 logback 内部日志信息，实时查看 logback
运行状态。默认值为 false。通常不打印 -->
<configuration scan="true" scanPeriod="10 seconds">
    <!--输出到控制台-->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!--此日志 appender 是为开发使用，只配置最底级别，控制台输出的日志级别是大
        于或等于此级别的日志信息-->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>debug</level>
        </filter>
        <encoder>
            <Pattern>%date %5p [%thread] %logger{60} [%file : %line] %msg%n
            </Pattern>
            <!-- 设置字符集 -->
            <charset>UTF-8</charset>
        </encoder>
    </appender>
    <appender name="File"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <File>E:/Project/Java/SPringboot/demo1/logback/springboot.log</File>
        <encoder>
            <pattern>%date %5p [%thread] %logger{60} [%file : %line]
%msg%n</pattern>
```

```
        </encoder>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

 <fileNamePattern>E:/Project/Java/SPringboot/demo1/logback/springboot.%d{yyyy-
MM-dd}.log</fileNamePattern>
            <maxHistory>30</maxHistory>
        </rollingPolicy>
    </appender>
    <!--单个定义-->
    <logger name="com.qiangliu8.springboot.mapper" level="DEBUG"/>

    <!--如果root标签指定的日志级别那么以根日志级别为准,如果没有则已当前追加器日志级别为准-->
    <!--全部-->
    <!--
        appender trace  trace
        root     trace

        appender trace  debug
        root     debug

        appender trace   debug
        root     空       如果root没有值默认root级别是debug

        appender debug  info
        root     info
    -->
    <root level="info">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="File"/>
    </root>
</configuration>
```

添加依赖

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

添加注解@Slf4j

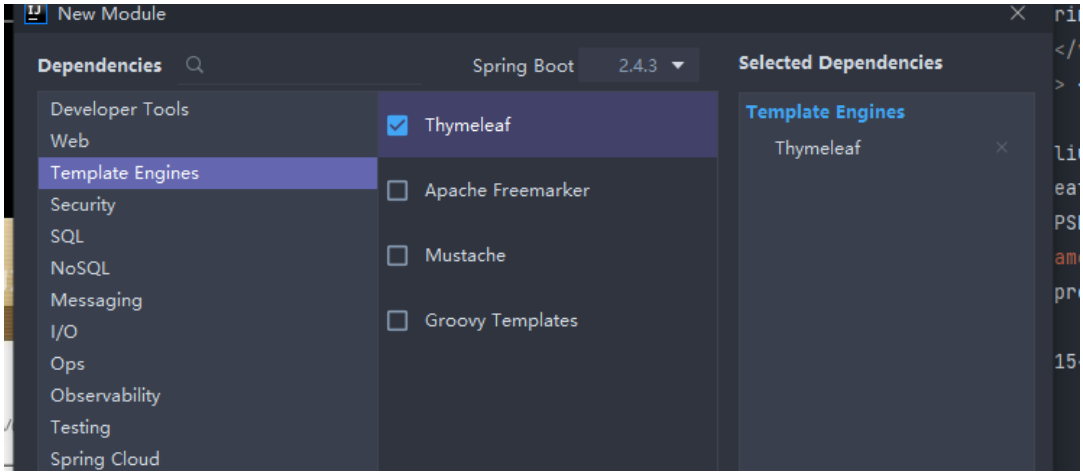
```
@Service
@Slf4j
public class StudenServiceImpl implements StudenService {
    @Autowired
    private StudentMapper studentMapper;

    public int studentCount(){
        log.info("查询当前学生总人数: ");
        return studentMapper.selectStudentCount();
    }

}
```

Root规定打印日志的级别 如果规定是DEBUG,则logo.trace() 就打印不出来了

十一、SpringBoot集成Thymeleaf

11.1 SpringBoot集成Thymeleaf

1. **创建工程时添加Thymeleaf引擎依赖**



pom文件自动生成Thymeleaf起步依赖

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

2. **创建Controller**

```
@Controller
public class UserController {
    @RequestMapping(value = "/message")
    public String message(Model model){
        model.addAttribute("data","springboot集成thymeleaf");
        return "message";
    }
}
```

3. **resources/templates下创建message.html**

写命名空间xmlns:th

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<!--thymeleaf模板引擎的页面必须得通过中央调度器-->
    <h2 th:text="${data}">展示要显示的内容</h2>
</body>
</html>
```

如果data有数据就会替换html的静态数据

4. **设置Thymeleaf模板引擎的前后缀**

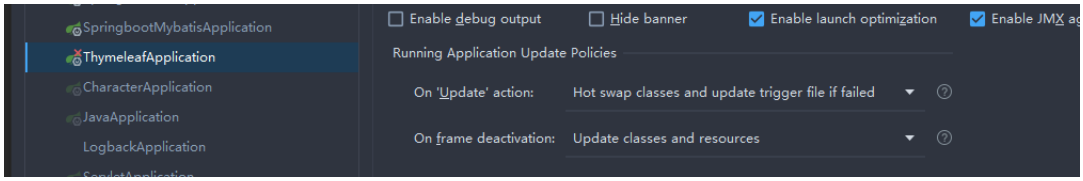
```
#设置Thymeleaf模板引擎的前后缀 可选项
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

11.2 Thymeleaf关闭页面缓存

1. **设置Thymeleaf模板引擎的缓存**

```
#设置Thymeleaf模板引擎的缓存 默认开启
spring.thymeleaf.cache=false
```

2. **设置Tomcat的更新设置**



11.3 标准变量和选择变量表达式

```
@RequestMapping("/user/detail")
public ModelAndView userDetail(){
    ModelAndView modelAndView = new ModelAndView();
    User user = new User(1001,"刘强",23);

    modelAndView.addObject("user",user);
    modelAndView.setViewName("userDetail");

    return modelAndView;
}
```

标准变量表达式---th:text="${user.getId()}

```
<h1 >选择变量表达式*{}</h1>
<div th:Object="${user}">
    <h1 th:text="*{id}">标准变量表达式</h1>
    <h1 th:text="*{name}">标准变量表达式</h1>
    <h1 th:text="*{age}">标准变量表达式</h1>
</div>
```

11.3 Thymeleaf路径表达式

无参数

```
<!DOCTYPE html>
<html lang="en" xmlns="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>URL路径表达式@{....}</h1>
    <a th:href="@{/user/detail}">跳转至:/user/detail</a>
</body>
</html>
```

有参数

```
@RequestMapping(value = "/url")
public String urlExpression(Model model){
    model.addAttribute("name","刘强");
    return "url";
}

@RequestMapping(value = "/urlParam")
@ResponseBody
public String urlParam(String name){
    return "请求参数是"+name;
}
```

```
<a th:href="@{http://localhost:8080/urlParam?name=liuqiang}">跳转至:/urlParam</a>
<a th:href="@{'/urlParam?name='+${name}}">跳转至:/urlParam</a>
<a th:href="@{'/urlParam?id='+${id}+'&name='+${name}}">多个参数跳转至:/urlParam</a>
<a th:href="@{/urlParam(id=&{id},name=&{name},age=&{age})}">多个参数跳转
至:/urlParam</a>
```

RESTful

```
@RequestMapping(value = "/url")
public String urlExpression(Model model){
    model.addAttribute("name","刘强");
    model.addAttribute("id",123);
    return "url";
}

@RequestMapping(value = "/RESTful/{id}")
@ResponseBody
public String urlParam(@PathVariable("id") Integer id){
    return "请求参数是id"+id;
}
```

```
<a th:href="@{'/RESTful/'+${id}}">请求路径为RESTful风格</a>
```

资源文件

```
<script type="text/javascript" th:src="@{/js/jquery-1.7.2.min.js}"></script>
```

11.4 Thymeleaf循环遍历list集合

```
@RequestMapping("/eachList")
public String eachList(Model model){
    List<User> users = new ArrayList<User>();
    for (int i = 0 ;i<10;i++){
        users.add(new User(i,i+"号","1575533716"+i,"合肥市"+i));
    }
    model.addAttribute("userList",users);
    return "eachList";
}
```

```
<body>
    <!--    user 当前循环对象变量名称
            userState   当前循环对象状态的变量（可选）
            ${userList} 当前循环的集合
-->
    <div th:each="user,userState:${userList}">
        <span th:text="${userState.index}"></span>
        <span th:text="${userState.count}"></span>

        <span th:text="${user.id}"></span>
        <span th:text="${user.nick}"></span>
        <span th:text="${user.phone}"></span>
        <span th:text="${user.address}"></span>
    </div>
</body>
```

11.5 Thymeleaf循环遍历Map集合

```
@RequestMapping("/eachMap")
public String eachMap(Model model){
    Map<Integer,User> userMap = new HashMap<>();
    for (int i = 0 ;i<10;i++){
        userMap.put(i,new User(i,i+"号","1575533716"+i,"合肥市"+i));
    }
    model.addAttribute("userMap",userMap);
    return "eachMap";
}
```

```
<body>
<!--    Map集合结构
    key value
    0   user
    1   user
    2   user
    3   user
...

-->
<div th:each="userMap,userMapState:${userMap}">
    <span th:text="${userMapState.index}"></span>
    <span th:text="${userMapState.count}"></span>
```

```
    <span th:text="${userMap.key}"></span>
    <span th:text="${userMap.value}"></span>
    <span th:text="${userMap.value.id}"></span>
    <span th:text="${userMap.value.nick}"></span>
</div>
</body>
```

11.6 Thymeleaf循环遍历数组

```
@RequestMapping("/eachArray")
public String eachArray(Model model){
    User[] userArrayList = new User[10];
    for (int i = 0 ;i<10;i++){
        userArrayList[i] = new User(i,i+"号","1575533716"+i,"合肥市"+i);
    }
    model.addAttribute("eachArray",userArrayList);
    return "eachArray";
}
```

```
<body>
<!--循环遍历Array数组(使用方法同list一样)-->

<div th:each="user,userState:${eachArray}">
    <span th:text="${userState.index}"></span>
    <span th:text="${userState.count}"></span>

    <span th:text="${user.id}"></span>
    <span th:text="${user.nick}"></span>
    <span th:text="${user.phone}"></span>
    <span th:text="${user.address}"></span>
</div>
</body>
```

11.7 Thymeleaf条件判断

```
<!--条件判断如果满足条件显示(执行),否则相反-->
<div th:if="${sex eq 1}">男</div>
<div th:if="${sex eq 0}">女</div>
<!--条件判断 用法:与th:if用法相反,即条件判断取反-->
<div th:unless="${sex eq 1}">女</div>

<!--switch-->
<h1 th:switch="${productType}">
    <span th:case="0">产品1</span>
    <span th:case="1">产品2</span>
    <span th:case="2">产品3</span>
</h1>
```

11.8 Thymeleaf内联表达式

```
<body>
    <div th:text="${data}"></div>

    <h1>内联文本 不依赖于标签 th:inline="text"</h1>
    <div th:inline="text">
        数据：[[${data}]]
    </div>
    数据：[[${data}]]
</body>
```

内联脚本

```
<script type="text/javascript" th:inline="text">
    console.log("[[${data}]]");
</script>
```

11.9 Thymeleaf自变量

```
<body>
<h1>文本自变量：用单引号‘。。。。’的字符串就是字面量</h1>
<a th:href="@{'/user/deatil/'+${data}}"></a>

<h1>数字自变量</h1>
<span th:text="2020+20"></span>

<h1>布尔自变量</h1>
<span th:if="${flag}">成功</span>
<span th:if="${!flag}">不成功</span>

<h1>空值自变量</h1>
<div th:text="${user.id}"></div>
<div th:unless="${user.address}">空</div>
</body>
```

11.10 Thymeleaf数学运算

三元运算:表达式?”正确结果”:”错误结果”

```
<div th:text="${flag eq true?'男':'女'}"></div>

<div th:text="${flag eq false?'男':'女'}"></div>
```

算术运算::+, -,* ,/ , %

```
20+5=<span th:text="20+5"></span>
20-5=<span th:text="20-5"></span>
20*5=<span th:text="20*5"></span>
20/5=<span th:text="20/5"></span>
20%5=<span th:text="20%5"></span>
```

关系比较::> ,< , >= , <= (gt , lt , ge , le)

```
    <div th:if="5>2">5>2为真</div>    <div th:if="5gt2">5gt为真</div>
<!--<div th:if="5 < 2">5<2为真</div>-->   <div th:if="5lt2">5lt为真</div>
    1>=1<div th:if="1 ge 1">为真</div>
    1<=1<div th:if="1 le 1">为真</div>
```

相等判断::== , != (eq , ne)

eq 等于==

！ = 等于ne

11.11 Thymeleaf基本表达式对象

Session对象

```
<span th:text="${#httpSession.getAttribute('address')}"></span>
<span th:text="${#session.getAttribute('address')}"></span>
<span th:text="${session.address}"></span>
```

获取路径

```
<script type="text/javascript" th:inline="javascript">
    //协议名称
    const scheme = [[${#request.getScheme()}]];
    //获取服务器名称
    const serverName = [[${#request.getServerName()}]];
    //服务服务器端口号
    const serverPort = [[${#request.getServerPort()}]];
    //获取上下文跟
    const contextPath = [[${#request.getContextPath()}]];
    console.log(scheme+"://"+serverName+":"+serverPort+"/"+contextPath);
</script>
```

```
//请求路径
 var requestURL = [[${#request.requestURL}]]
console.log(requestURL);
 //请求参数
 var queryString = [[${#request.queryString}]]
 console.log(queryString);
```

11.12 Thymeleaf功能表达式对象

内置功能对象前都需要加#号，内置对象一般都以 s 结尾

官方手册：http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

#dates: java.util.Date 对象的实用方法：

#calendars: 和 dates 类似，但是 java.util.Calendar 对象；

#numbers: 格式化数字对象的实用方法；

#strings: 字符串对象的实用方法： contains, startsWith, prepending/appending 等；

#objects: 对 objects 操作的实用方法；

#bools: 对布尔值求值的实用方法；

#arrays: 数组的实用方法；

#lists: list 的实用方法，比如

#sets: set 的实用方法；

#maps: map 的实用方法；

#aggregates: 对数组或集合创建聚合的实用方法；

```
<div th:text="${time}"></div>
<div th:text="${#dates.format(time,'yyyy-MM-dd HH:mm:ss')}"></div>
```