彻底掌握SSM框架

(实战篇)

黄文毅 编著

1.1 课程介绍

1.1.1 课程介绍

**开发环境：**

Spring（最新版） + Spring MVC（最新版） + MyBatis（最新版）

Spring + Struts + Hibernate（淘汰）

Window10 + Intellij IDEA 2018 + JDK1.8 + Tomcat + Maven

**适用人群：**

1.编程人员

2.初级工程师、中级工程师

3.在线学生，研究生

4.SSM框架爱好者

**联系方式：**

QQ：[1512079179@qq.com](mailto:1512079179@qq.com)

微信：huangwenyi2010

邮箱地址：huangwenyi10@163.com

**重要式：**

1）目前主流的框架，也是Web应用人员必须掌握的基本框架。

**课程资料：**

购买正版课程后可联系老师获取课程资料。

**课程服务：**

购买正版视频，提供购买订单号，可以享受以下服务：

1）学员老师一对一交流服务

2）课后作业和思考题服务

3）咨询服务（职业规划、经验交流）

4）购买【盗版视频】无法获得所有的源代码、课件及服务。

**课程目标：**

帮助学员快速掌握SSM框架

**课程目录：**

1.1 课程介绍

1.2 快速搭建Web项目

1.3 集成Spring框架

1.4 3种ApplicationContext实现

1.5 集成Spring MVC框架

1.6 集成MyBatis框架

1.7 集成Log4J框架

1.8 集成Junit测试框架

1.9 解决控制台乱码问题

2.1.1 select元素

2.1.2 insert元素

2.1.3 update元素

2.1.4 delete元素

2.1.5 sql元素

2.1.6 #与$区别

2.1.7 resultMap结果映射集

2.1.8 if元素

2.1.9 choose、when、otherwise元素

2.1.10 where元素

2.1.11 set元素

2.1.12 trim元素

2.1.13 foreach元素

2.1.14 bind元素

3.1 @Select注解

3.2 @Insert、@Update、@Delete注解

4.1 物理分页和逻辑分页

4.2 RowBounds分页

4.3 PageHelper分页

5.1.1 @Controller和@RequestMapping

5.1.2 @GetMapping、@PostMapping注解

5.1.3 Model、ModelMap、ModelAndView

5.1.4 @RequstParam和@PathVariable注解

5.1.5 @RequestHeader注解

5.1.6 @CookieValue注解

5.1.7 @ModelAttribute注解

5.1.8 @SessionAttribute和@SessionAttributes注解

5.1.9 @ResponseBody和@RequestBody注解

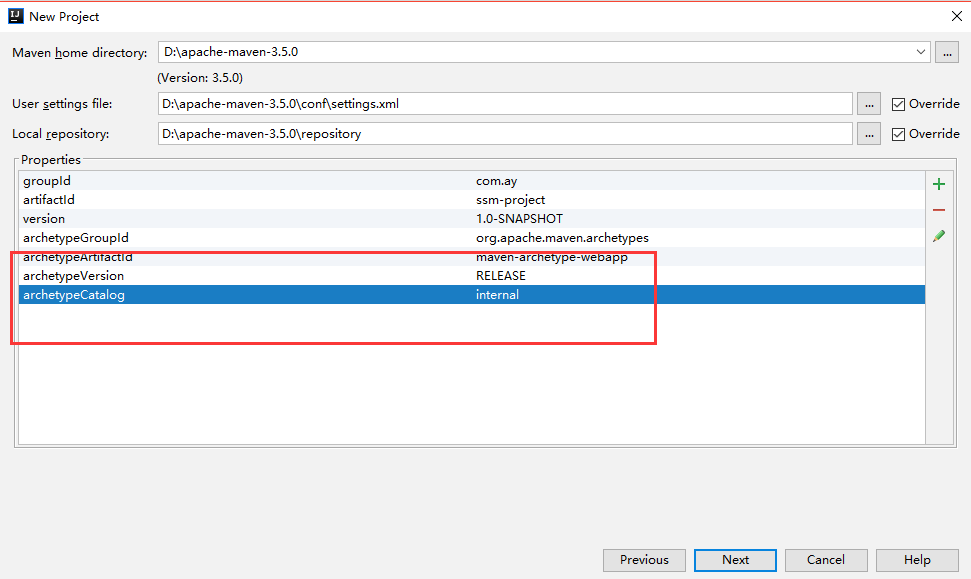
6.1.1 Spring的Validation校验框架

6.1.2 JSR 303校验

1.2 快速搭建Web项目

注意点：name：archetypeCatalog，value：internal

原因：Intellij IDEA根据maven archetype的本质，执行mvn archetype:generate命令。该命令执行时，需要指定一个archetype-catalog.xml文件。该命令的参数-DarchetypeCatalog，可选值为：remote，internal，local等，用来指定archetype-catalog.xml文件从哪里获取，默认为remote，即从 http://repo1.maven.org/maven2/archetype-catalog.xml路径下载archetype-catalog.xml文件。archetype-catalog.xml 文件约为3-4M，下载速度很慢，导致创建过程卡住。解决的办法很简单， 指定-DarchetypeCatalog为internal，即可使用maven默认的archetype-catalog.xml，而不用从remote下载。



1.3 集成Spring框架

1、首先，在ssm-project项目的pom文件中添加spring相关的依赖，具体代码如下：

<properties>

<spring.version>5.0.4.RELEASE</spring.version>

</properties>

<!--spring start -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-expression</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<!--spring end -->

2、其次，在/src/main/resources目录下创建applicationContext.xml配置文件，具体代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="com.ay"/>

</beans>

3、接着，我们在web.xml配置文件中添加如下的代码：

<!DOCTYPE web-app PUBLIC

"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd" >

<web-app>

<display-name>Archetype Created Web Application</display-name>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener>

</web-app>

4、最后，在src/main/test/com.ay.test目录下创建SpringTest测试类，具体代码如下：

import org.junit.Test;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.stereotype.Service;

/\*\*

\* @author Ay

\* @date 2018/04/02

\*/

@Service

public class SpringTest {

@Test

public void testSpring(){

//获取运用上下文

ApplicationContext applicationContext =

new ClassPathXmlApplicationContext("applicationContext.xml");

//获取SpringTest类

SpringTest springTest = (SpringTest) applicationContext.getBean("springTest");

//调用sayHello方法

springTest.sayHello();

}

public void sayHello(){

System.out.println("hello ay");

}

}

1.4 3种ApplicationContext实现

**ApplicationContext容器：**ApplicationContext是Spring中较高级的容器，它可以加载配置文件中定义的Bean，并将所有的Bean 集中在一起，当有请求的时候分配Bean。

最经常被使用的ApplicationContext接口实现如下：

**ClassPathXmlApplicationContext：**从类路径ClassPath中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作，具体代码如下：

//装载单个配置文件实例化ApplicationContext容器

ApplicationContext cxt = new ClassPathXmlApplicationContext("applicationContext.xml");

//装载多个配置文件实例化ApplicationContext容器

String[] configs = {"bean1.xml","bean2.xml","bean3.xml"};

ApplicationContext cxt = new ClassPathXmlApplicationContext(configs);

**FileSystemXmlApplicationContext：**从指定的文件系统路径中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作。具体代码如下：

//装载单个配置文件实例化ApplicationContext容器

ApplicationContext cxt = new FileSystemXmlApplicationContext ("beans.xml");

//装载多个配置文件实例化ApplicationContext容器

String[] configs = {"c:/beans1.xml","c:/beans2.xml"};

ApplicationContext cxt = new FileSystemXmlApplicationContext(configs);

**XmlBeanFactory引用资源：**

Resource resource = new ClassPathResource("applicationContext.xml");

BeanFactory factory = new XmlBeanFactory(resource);

AyTestController ayTestController =

(AyTestController) factory.getBean("ayTestController");

ayTestController.hello();

**XmlWebApplicationContext：**从Web应用中寻找指定的XML配置文件，找到并装载完成ApplicationContext的实例化工作。这是为Web工程量身定制的，使用WebApplicationContextUtils类的getRequiredWebApplicationContext方法可在JSP与Servlet中取得IoC容器的引用。

作业：动手实现**XmlWebApplicationContext**

1.5 集成Spring MVC框架

首先，我们把集成Spring MVC所需要的Maven依赖包和相关的属性值添加到pom.xml文件中，具体代码如下：

<properties>

<spring.version>5.0.4.RELEASE</spring.version>

<javax.servlet.version>4.0.0</javax.servlet.version>

<jstl.version>1.2</jstl.version>

</properties>

<!--springmvc start -->

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

<version>${jstl.version}</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>${javax.servlet.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<!--springmvc end -->

其次，在web.xml配置文件中添加DispatcherServlet配置，具体代码如下：

<!--配置DispatcherServlet -->

<servlet>

<servlet-name>spring-dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- 配置SpringMVC需要加载的配置文件 spring-mvc.xml -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-mvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>spring-dispatcher</servlet-name>

<!-- 默认匹配所有的请求 -->

<url-pattern>/</url-pattern>

</servlet-mapping>

接着，我们在/src/main/resources目录下创建配置文件spring-mvc.xml，具体代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- 扫描controller(后端控制器),并且扫描其中的注解-->

<context:component-scan base-package="com.ay.controller"/>

<!--设置配置方案 -->

<mvc:annotation-driven/>

<!--配置JSP　显示ViewResolver(视图解析器)-->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass"

value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/views/"/>

<property name="suffix" value=".jsp"/>

</bean>

</beans>

我们在/src/main/webapp/WEB-INF目录下创建views文件夹，在views文件下创建hello.jsp文件，具体代码如下：

<%@page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>

<!DOCTYPE HTML>

<html>

<head>

<title>Getting Started: Serving Web Content</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

</head>

<body>

hello, ay

</body>

</html>

1.6 集成MyBatis框架

首先，我们把集成MyBatis框架所需要的依赖包添加到pom.xml文件中，具体代码如下：

<properties>

<spring.version>5.0.4.RELEASE</spring.version>

<javax.servlet.version>4.0.0</javax.servlet.version>

<jstl.version>1.2</jstl.version>

<mybatis.version>3.4.6</mybatis.version>

<mysql.connector.java.version>8.0.9-rc</mysql.connector.java.version>

<druid.version>1.1.9</druid.version>

<mybatis.spring.version>1.3.2</mybatis.spring.version>

</properties>

<!--mybatis start -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql.connector.java.version}</version>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>${druid.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>${mybatis.version}</version>

</dependency>

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>${mybatis.spring.version}</version>

</dependency>

<!--mybatis end -->

作业：阿里巴巴druid数据源学习

其次，在/src/main/resources目录下创建jdbc.properties配置文件，具体代码如下：

//驱动

jdbc.driverClassName=com.mysql.jdbc.Driver

//mysql连接信息

jdbc.url=jdbc:mysql://127.0.0.1:3306/springmvc-mybatis-book?serverTimezone=GMT

//用户名

jdbc.username=root

//密码

jdbc.password=123456

**作业：动手安装MySQL数据库**

**jdbc.properties配置：**主要配置驱动和连接数据库的配置信息。

最后，我们在applicationContext.xml配置文件添加如下的配置，具体代码如下：

<!--1、配置数据库相关参数-->

<context:property-placeholder

location="classpath:jdbc.properties" ignore-unresolvable="true"/>

<!--2.数据源 druid -->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"

init-method="init" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

<!--3、配置SqlSessionFactory对象-->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!--注入数据库连接池-->

<property name="dataSource" ref="dataSource"/>

<!--扫描sql配置文件:mapper需要的xml文件-->

<property name="mapperLocations" value="classpath:mapper/\*.xml"/>

</bean>

<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">

<constructor-arg index="0" ref="sqlSessionFactory" />

</bean>

<!-- 扫描basePackage下所有以@MyBatisDao注解的接口 -->

<bean id="mapperScannerConfigurer"

class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<property name="sqlSessionFactoryBeanName" value="sqlSessionFactory" />

<property name="basePackage" value="com.ay.dao"/>

</bean>

我们在/src/main/java/com.ay.model目录下创建数据库表对应的实体类对象AyUser，具体的代码如下：

/\*\*

\* 用户实体

\* @author Ay

\* @date 2018/04/02

\*/

public class AyUser implements Serializable{

private Integer id;

private String name;

private String password;

//省略set、get方法

}

实体类对象AyUser创建完成之后，我们在/src/main/java/ com.ay.dao目录下创建对应的DAO对象AyUserDao，AyUserDao是一个接口，提供了findAll方法用来查询所有的用户。AyUserDao具体代码如下：

package com.ay.dao;

import com.ay.model.AyUser;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface AyUserDao {

List<AyUser> findAll();

}

接口类AyUserDao创建完成之后，我们在/src/main/java/com.ay.service目录下创建对应的服务层接口AyUserService，服务层接口AyUserService代码也非常简单，只提供了一个查询所有用户的方法findAll（），具体的代码如下：

package com.ay.service;

import com.ay.model.AyUser;

import java.util.List;

public interface AyUserService {

List<AyUser> findAll();

}

服务层接口AyUserService开发完成之后，我们在/src/main/java/com.ay.service.impl开发对应的服务层实现类AyUserServiceImpl，实现类主要是注入AyUserDao接口，并实现findAll（）方法，在findAll（）方法中调用AyUserDao的findAll（）方法，具体代码如下所示：

package com.ay.service.impl;

import com.ay.dao.AyUserDao;

import com.ay.model.AyUser;

import com.ay.service.AyUserService;

import org.springframework.stereotype.Repository;

import org.springframework.stereotype.Service;

import javax.annotation.Resource;

import java.util.List;

@Service

public class AyUserServiceImpl implements AyUserService{

@Resource

private AyUserDao ayUserDao;

public List<AyUser> findAll() {

return ayUserDao.listAllUser();

}

}

服务层实现类AyUserServiceImpl开发完成之后，我们在/src/main/java/com.ay.controller目录下创建控制层类AyUserController，并注入服务层接口。AyUserController类只有一个findAll（）方法。在AyUserController类上添加映射路径/user，在findAll（）方法上添加映射路径/findAll。

package com.ay.controller;

import com.ay.model.AyUser;

import com.ay.service.AyUserService;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import javax.annotation.Resource;

import java.util.List;

/\*\*

\*@author Ay

\* @date 2018/04/02

\*/

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@Resource

private AyUserService ayUserService;

@GetMapping("/findAll")

public String findAll(Model model){

List<AyUser> ayUserList = ayUserService.findAll();

for(AyUser ayUser : ayUserList){

System.out.println("id: " + ayUser.getId());

System.out.println("name: " + ayUser.getName());

}

return "hello";

}

}

最后，我们在/src/main/resources目录下创建AyUserMapper.xml文件，具体代码如下：

<?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.ay.dao.AyUserDao">

<sql id="userField">

a.id as "id",

a.name as "name",

a.password as "password"

</sql>

<!-- 获取所有用户 -->

<select id="findAll" resultType="com.ay.model.AyUser">

select

<include refid="userField"/>

from ay\_user as a

</select>

</mapper>

1.7 集成Log4J框架

首先需要在pom.xml文件中引入所需的依赖，具体代码如下：

!-- log4j2 -->

<properties>

//省略部分代码

<slf4j.version>1.7.7</slf4j.version>

<log4j.version>1.2.17</log4j.version>

</properties>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>${log4j.version}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>${slf4j.version}</version>

</dependency>

我们在项目的/src/main/java/resources/下创建配置文件log4j.properties，具体代码如下所示：

###set log levels

log4j.rootLogger = DEBUG,Console

###输出到控制台

log4j.appender.Console=org.apache.log4j.ConsoleAppender

log4j.appender.Console.Target=System.out

log4j.appender.Console.layout=org.apache.log4j.PatternLayout

log4j.appender.Console.layout.ConversionPattern= %d{ABSOLUTE} %5p %c{1}:%L - %m%n

1.8 集成Junit测试框架

首先，我们在项目的pom.xml配置文件中添加相关的依赖，具体代码如下：

<!-- junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

然后，我们在项目的/src/main/test/com.ay.test目录下创建测试基类BaseJunit4Test，具体代码如下所示：

/\*\*

\* 描述：测试基类

\* @author Ay

\* @create 2018/05/04

\*\*/

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations={"classpath:applicationContext.xml"})

public class BaseJunit4Test {

}

**@RunWith：**参数化运行器，用于指定Junit运行环境，是Junit提供给其它框架测试环境接口扩展，为了便于使用Spring的依赖注入，Spring提供了SpringJUnit4ClassRunner作为Junit测试环境。

BaseJunit4Test类开发完成之后，我们在/src/main/test/com.ay.test目录下创建AyUserDaoTest测试类简单测试集成Junit框架是否成功，具体代码如下：

/\*\*

\* 描述：用户DAO测试类

\* @author Ay

\* @create 2018/05/04

\*\*/

public class AyUserDaoTest extends BaseJunit4Test{

@Resource

private AyUserDao ayUserDao;

@Test

public void testFindAll(){

List<AyUser> userList = ayUserDao.findAll();

System.out.println(userList.size());

}

}

1.9 解决控制台乱码问题

1、配置文件添加编码

2、Tomcat配置添加编码

2.1 MyBatis动态SQL

### 2.1.1 select元素

<select id="findById" parameterType="String" resultType="com.ay.model.AyUser">

SELECT \* FROM ay\_user

WHERE id = #{id}

</select>

上面的SQL语句执行时会生成如下JDBC代码：

String findById = "SELECT \* FROM ay\_user WHERE id = ? "

PreparedStatement ps = conn.prepareStatement(findById);

ps.setString(1,id);

接口AyUserDao中定义的方法如下：

AyUser findById(String id);

select元素提供了很多配置属性，具体如表4-2所示。

作业：select元素的属性

我们再来看几个select元素的例子：

//实例1：通过名称查询用户

<select id="findByName" parameterType="String" resultType="com.ay.model.AyUser">

SELECT \* FROM ay\_user

WHERE name = #{name}

</select>

//实例2：通过名称查询用户个数

<select id="countByName" parameterType="String" resultType="int">

SELECT count(\*) FROM ay\_user

WHERE name = #{name}

</select>

对应的AyUserDao接口如下：

List<AyUser> findByName(String name);

int countByName(String name);

### 2.1.2 insert元素

我们来看几个例子：

//实例1：插入用户数据

<insert id="insert" parameterType="com.ay.model.AyUser">

INSERT INTO ay\_user(id, name, password) VALUE (#{id}, #{name}, #{password});

</insert>

//实例2：插入用户数据，主键自增

<insert id="insert" useGeneratedKeys="true"

keyProperty="id" parameterType="com.ay.model.AyUser">

INSERT INTO ay\_user(name, password) VALUE (#{name}, #{password});

</insert>

对应的AyUserDao接口如下：

int insert(AyUser ayUser);

### 2.1.3 update元素

我们来看一个具体的实例：

<update id="update" parameterType="com.ay.model.AyUser">

UPDATE ay\_user SET

name = #{name},

password = #{password}

WHERE id = #{id}

</update>

对应的AyUserDao接口如下：

int update(AyUser ayUser);

### 2.1.4 delete元素

我们来看几个具体的实例：

//实例1：根据id删除记录

<delete id="delete" parameterType="int">

DELETE FROM ay\_user

WHERE id = #{id}

</delete>

//实例2：根据name删除记录

<delete id="deleteByName" parameterType="String">

DELETE FROM ay\_user

WHERE name = #{name}

</delete>

### 2.1.5 sql元素

我们来看几个具体的实例：

<sql id="userField">

a.id as "id",

a.name as "name",

a.password as "password"

</sql>

<!-- 获取所有用户 -->

<select id="findAll" resultType="com.ay.model.AyUser">

Select

//使用refid进行引用

<include refid="userField"/>

from ay\_user a

</select>

我们可以很方便的使用include元素的refid属性进行引用。我们还可以使用定制参数来使用sql元素，具体代码如下：

<sql id="userField">

//注意：这里使用$符合而不是#符号，否则程序出现异常

${prefix}.id as "id",

${prefix}.name as "name",

${prefix}.password as "password"

</sql>

<!-- 获取所有用户 -->

<select id="findAll" resultType="com.ay.model.AyUser">

select

<include refid="userField">

<property name="prefix" value="a"/>

</include>

from ay\_user a

</select>

### 2.1.6 #与$区别

1）＃{}将传入的数据都当成一个字符串，会对自动传入的数据加一个双引号，具体示例如下：

order by #{id}

//如果id传入11，则sql解析成：

order by “11”

2）${}将传入的数据直接显示生成在sql中，具体示例如下：

order by #{id}

//如果id传入11，则sql解析成：

order by 11

3）#方式能够很大程度防止sql注入，$方式无法防止sql注入。

综上所述，我们一般建议采用#，而不是$。

作业：#为什么能够防止sql注入的问题？

qq：1512079179

### 2.1.7 resultMap结果映射集

resultMap元素还包含其它的元素，具体如下：

<resultMap>

<constructor> /\*用来将查询结果作为参数注入到实例的构造方法中\*/

<idArg/> /\*标记结果作为 ID\*/

<arg/> /\*标记结果作为普通参数\*/

</constructor>

<id/> /\*一个ID结果，标记结果作为 ID\*/

<result/> /\*一个普通结果，JavaBean的普通属性或字段\*/

<association> /\*关联其他的对象\*/

</association>

<collection> /\*关联其他的对象集合\*/

</collection>

<discriminator> /\*鉴别器，根据结果值进行判断，决定如何映射\*/

<case></case> /\*结果值的一种情况，将对应一种映射规则\*/

</discriminator>

</resultMap>

我们先来看一个具体的示例，代码如下：

<sql id="userField">

${prefix}.id as "id",

${prefix}.name as "name",

${prefix}.password as "password"

</sql>

<resultMap id="userMap" type="com.ay.model.AyUser">

<id property="id" column="id"/>

<result property="name" column="name"/>

<result property="password" column="password"/>

</resultMap>

<select id="findAll" resultMap="userMap">

select

<include refid="userField">

<property name="prefix" value="a"/>

</include>

from ay\_user a

</select>

### 2.1.8 if元素

具体看下面的示例：

<sql id="userField">

a.id as "id",

a.name as "name",

a.password as "password"

</sql>

<resultMap id="userMap" type="com.ay.model.AyUser">

<id property="id" column="id"/>

<result property="name" column="name"/>

<result property="password" column="password"/>

</resultMap>

//通过用户名name和密码password查询用户

<select id="findByNameAndPassword" parameterType="String" resultMap="userMap">

SELECT

<include refid="userField"></include>

from ay\_user a

WHERE 1 = 1

<if test="name != null and name != ''">

and name = #{name}

</if>

<if test="password != null and password != ''">

and password = #{password}

</if>

</select>

对应的AyUserDAO接口代码如下：

List<AyUser> findByNameAndPassword(@Param("name") String name,

@Param("password")String password);

if标签常常与test属性联合使用且是必选属性。上述代码中，通过判断name或者password参数是否为空，如果不为空，拼凑SQL语句进行查询。如果为空，忽略。

### 2.1.9 choose、when、otherwise元素

choose、when、otherwise元素提供三重选择，有点类似switch..case..default语句，具体示例代码如下所示：

<sql id="userField">

a.id as "id",

a.name as "name",

a.password as "password"

</sql>

<resultMap id="userMap" type="com.ay.model.AyUser">

<id property="id" column="id"/>

<result property="name" column="name"/>

<result property="password" column="password"/>

</resultMap>

//通过名称name和密码password查询用户

<select id="findByNameAndPassword" parameterType="String" resultMap="userMap">

SELECT

<include refid="userField"></include>

from ay\_user a

WHERE 1 = 1

<choose>

<when test="name != null and name != ''">

and name = #{name}

</when>

<when test="password != null and password != ''">

and password = #{password}

</when>

<otherwise>

ORDER BY id DESC

</otherwise>

</choose>

</select>

### 2.1.10 where元素

编写SQL语句的时候，我们通常喜欢写这样的SQL语句：

<select id="findByName" parameterType="String" resultType="com.ay.model.AyUser">

SELECT \* FROM ay\_user WHERE 1 = 1

<if test="name != null and name != ''">

and name = #{name}

</if>

</select>

WHERE 1 = 1这样的条件显然很奇怪，所以我们可以使用WHERE标签优化上面的SQL语句，具体代码如下：

<select id="findByName" parameterType="String" resultType="com.ay.model.AyUser">

SELECT \* FROM ay\_user

<where>

<if test="name != null and name != ''">

and name = #{name}

</if>

</where>

</select>

这样当where元素里面的条件成立的时候，才会加入where这个SQL关键字到组装的SQL里面，否则就不加入。

### 2.1.11 set元素

set元素在执行SQL更新中会使用到，我们先来看一个传统代码写法，具体如下：

<update id="update" parameterType="com.ay.model.AyUser">

UPDATE ay\_user SET

name = #{name},

password = #{password}

WHERE id = #{id}

</update>

上面代码中没有使用set元素，我们可以对代码进行优化，具体优化后的代码如下：

<update id="update" parameterType="com.ay.model.AyUser">

UPDATE ay\_user

<set>

<if test="name != null and name != ''">

name = #{name},

</if>

<if test="password != null and password != ''">

password = #{password},

</if>

</set>

WHERE id = #{id}

</update>

set元素遇到逗号，它会把对应的逗号去掉，比如上面代码中的password = #{password}，不需要我们自己写判断语句去除逗号。

### 2.1.12 trim元素

trim 是更灵活用来去处多余关键字的标签，它可以用来实现 where 和 set 的效果

具体内容我们可以看下面的示例:

<select id="findByNameAndPassword" parameterType="String" resultMap="userMap">

SELECT

<include refid="userField"></include>

from ay\_user a

<trim prefix="WHERE" prefixOverrides="AND">

<if test="name != null and name != ''">

and name = #{name}

</if>

<if test="password != null and password != ''">

and password = #{password}

</if>

</trim>

</select>

假如name和password字段都不为空的话，上面的代码相但于如下的SQL语句：

SELECT

a.id as "id",

a.name as "name",

a.password as "password"

from ay\_user a

WHERE name = #{name} and password = #{password}

我们再来看另外一个示例:

<update id="update" parameterType="com.ay.model.AyUser">

UPDATE ay\_user

<trim prefix="SET" suffixOverrides=",">

<if test="name != null and name != ''">

name = #{name},

</if>

<if test="password != null and password != ''">

password = #{password},

</if>

</trim>

WHERE id = #{id}

</update>

假如name和password字段都不为空的话，上面的代码相但于如下的SQL语句：

UPDATE ay\_user

SET

name = #{name},

password = #{password}

WHERE id= #{id}

### 2.1.13 foreach元素

foreach元素是一个循环语句，作用是遍历集合，支持数组、List、Set等。具体看下面的示例：

//根据Id集合查询用户列表

<select id="findByIds" resultType="com.ay.model.AyUser">

SELECT \* FROM ay\_user

WHERE id in

<foreach item="item" index="index" collection="list"

open="(" separator="," close=")">

#{item}

</foreach>

</select>

### 2.1.14 bind元素

bind元素可以从OGNL表达式中创建一个变量并将其绑定到上下文

作业：bind元素作为大家自学的内容

QQ:1512079179

3.1 MyBatis注解配置

### 3.1 @Select注解

@Repository

public interface AyUserDao {

//实例1：查询所有的用户列表

@Select("SELECT \* FROM ay\_user")

List<AyUser> findAll();

//实例2：查询所有的用户列表

@Select("SELECT \* FROM ay\_user")

@Results({

@Result(id = true,column = "id",property = "id"),

@Result(column = "name",property = "name"),

@Result(column = "password",property = "password")

})

List<AyUser> findAll();

//实例3：通过id查询用户

@Select("SELECT \* FROM ay\_user WHERE id = #{id}")

AyUser findById(String id);

//实例4：通过用户名获取用户

@Select("SELECT \* FROM ay\_user WHERE name = #{name}")

List<AyUser> findByName(String name);

}

### 3.2 @Insert、@Update、@Delete注解

@Insert、@Update、@Delete注解与XML配置里的insert、update、delete标签相对应，具体实例如下：

@Repository

public interface AyUserDao {

//实例1：插入用户数据

@Insert("INSERT INTO ay\_user(name,password) VALUES(#{name}, #{password})")

@Options(useGeneratedKeys = true, keyProperty = "id")

int insert(AyUser ayUser);

//实例2：更新用户数据

@Update("UPDATE ay\_user SET name = #{name}, password = #{password} WHERE id = #{id}")

int update(AyUser ayUser);

//实例3：根据用户id删除用户

@Delete("DELETE FROM ay\_user WHERE id = #{id}")

int delete(int id);

//实例4：根据用户名删除用户

@Delete("DELETE FROM ay\_user WHERE name = #{name}")

int deleteByName(String name);

}

作业：更多的注解标签，希望大家自学，可以把作业发给老师看。

4.1 RowBounds分页

### 4.1 物理分页和逻辑分页

分页查询可分为：逻辑分页和物理分页。

逻辑分页：依赖程序员编写的代码，数据库返回的不是分页结果，而是全部数据，然后再由程序员通过代码获取分页数据。常用的操作是一次性从数据库中查询出全部数据并存储到List集合中，因为List集合有序，再根据索引获取指定范围的数据。

物理分页：使用数据库自身所带的分页机制，例如：Oracle数据库的rownum，或者Mysql数据库中的limit等机制来完成分页操作。因为是对数据库实实在在的数据进行分页条件查询，所以叫物理分页。每一次物理分页都会去连接数据库。

物理分页优于逻辑分页，我们没有必要将属于数据库端的压力施加到应用端来，所以建议大家在日常工作中尽量使用物理分页而不是逻辑分页。

### 4.2 RowBounds分页

首先，我们在AyUserMapper.xml配置文件中添加select查询，具体代码如下所示：

<sql id="userField">

a.id as "id",

a.name as "name",

a.password as "password"

</sql>

//查询所有的用户

<select id="findAll" resultMap="userMap">

select

<include refid="userField"/>

from ay\_user a

</select>

然后，我们在AyUserDao接口中添加对应的查询方法findAll，findAll方法入参是RowBounds，具体代码如下所示：

@Repository

public interface AyUserDao {

List<AyUser> findAll(RowBounds rowBounds);

//省略其它代码

}

作业：RowBounds分页原理学习？

### 4.3 PageHelper分页

首先，我们在项目的pom.xml文件添加PageHelper依赖包，具体代码如下所示：

<dependency>

<groupId>com.github.pagehelper</groupId>

<artifactId>pagehelper</artifactId>

<version>5.1.4</version>

</dependency>

依赖添加完成之后，我们在applicationContext.xml配置文件中添加PageHelper相关配置，具体代码如下：

//省略代码

!--3、配置SqlSessionFactory对象-->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<!--注入数据库连接池-->

<property name="dataSource" ref="dataSource"/>

<!--扫描sql配置文件:mapper需要的xml文件-->

<property name="mapperLocations" value="classpath:mapper/\*.xml"/>

<!-- 配置分页插件 -->

<property name="plugins">

<array>

<bean class="com.github.pagehelper.PageInterceptor">

<property name="properties">

<value>

<!—- 数据库类型为mysql -->

helperDialect=mysql

<!-- 启用合理化时，如果pageNum <1会查询第一页，

如果pageNum > pages会查询最后一页 -->

<!-- 禁用合理化时，如果pageNum < 1或pageNum > pages会返回空数据 -->

reasonable=true

</value>

</property>

</bean>

</array>

</property>

</bean>

//省略代码

配置添加完成之后，我们在AyUserDaoTest开发测试用例，具体代码如下：

@Test

public void testPageHelper(){

//startPage(第几页, 多少条数据)

PageHelper.startPage(0, 1);

//查询所有用户

List<AyUser> userList = ayUserDao.findAll();

//用PageInfo对结果进行包装

PageInfo pageInfo = new PageInfo(userList);

}

作业：自己动手根据PageHelper封装分页的接口？

5.1 Sring MVC常用注解

### 5.1.1 @Controller和@RequestMapping

我们来看几个实例，具体代码如下：

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@RequestMapping(value = {

"",

"/page",

"page\*",

"view/\*,\*\*/msg"

})

public String hello(Model model){

model.addAttribute("message", "hello ay");

return "hello";

}

}

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@RequestMapping(method = RequestMethod.GET)

String get() {

return "Hello from get";

}

@RequestMapping(method = RequestMethod.DELETE)

String delete() {

return "Hello from delete";

}

@RequestMapping(method = RequestMethod.POST)

String post() {

return "Hello from post";

}

@RequestMapping(method = RequestMethod.PUT)

String put() {

return "Hello from put";

}

@RequestMapping(method = RequestMethod.PATCH)

String patch() {

return "Hello from patch";

}

}

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@RequestMapping(value = "/produces", produces = {

"application/json"

})

@ResponseBody

String getProduces() {

return "Produces attribute";

}

@RequestMapping(value = "/consumes", consumes = {

"application/json",

"application/xml"

})

String getConsumes() {

return "Consumes attribute";

}

}

作业：produces = {

"application/json"

},

consumes = {

"application/json"

} 属性的学习。

### 5.1.2 @GetMapping、@PostMapping注解

**@GetMapping：**@GetMapping是一个组合注解，是@RequestMapping(method = RequestMethod.GET)的缩写。该注解将HTTP Get 请求映射到特定的处理方法上。类似的@PostMapping注解是@RequestMapping(method = RequestMethod.POST)的缩写。@PutMapping注解是@RequestMapping(method = RequestMethod.PUT)的缩写。@DeleteMapping注解是@RequestMapping(method = RequestMethod.DELETE)的缩写。@PatchMapping注解是@RequestMapping(method = RequestMethod.PATCH)的缩写

### 5.1.3 Model、ModelMap、ModelAndView

### 5.1.4 @RequstParam和@PathVariable注解

### 5.1.5 @RequestHeader注解

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@RequestMapping("/requestHeader")

public String handle(

@RequestHeader("Accept-Encoding") String[] encoding,

@RequestHeader("Accept") String[] accept) {

//...

return "";

}

}

### 5.1.6 @CookieValue注解

### 5.1.7 @ModelAttribute注解

实例一：

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@ModelAttribute

public void init(){

System.out.println("init ...");

}

@ModelAttribute

public void init02(){

System.out.println("init 02 ...");

}

@GetMapping("/findById/{id}")

public String findById(@PathVariable String id) {

// ...

return "";

}

@ModelAttribute

public void init03(){

System.out.println("init 03 ...");

}

}

当在浏览器输入访问URL：<http://localhost:8080/user/findById/1>时，便可以在控制台看到打印信息：

init ...

init 02 ...

init 03 ...

@ModelAttribute注解有很多的额使用方式，下面我们会逐一介绍。

**1）@ModelAtterbute方法无返回值情况**

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@ModelAttribute

public void init(Model model){

AyUser ayUser = new AyUser();

ayUser.setId(1);

ayUser.setName("ay");

model.addAttribute("user", ayUser);

}

@GetMapping("/hello")

public String hello(){

return "hello";

}

}

上述代码中@ModelAttribute注解标记的init方法无任何返回值，在init方法中创建一个用户对象AyUser并设置id和name的值，最后调用Model对象的addAttribute方法设置到Model对象中。对应前端src\main\webapp\WEB-INF\views\hello.jsp页面代码如下所示：

<%@page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8" isELIgnored="false"%>

<!DOCTYPE HTML>

<html>

<head>

<title>Getting Started: Serving Web Content</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

</head>

<body>

hello, ${user.name}

</body>

</html>

当在浏览器中输入请求URL：<http://localhost:8080/user/hello>时，浏览器显示“hello，ay”。

从执行结果可以看出，当执行请求时，首先访问init方法，然后再访问hello方法，并且是同一个请求。因为model模型数据的作用域与request相同，所以可以用@ModelAttribute注解直接标记在方法上对实际要访问的方法进行一些初始化操作。

**2）**@ModelAttribute标记方法有返回值

@Controller

@RequestMapping(value = "/user")

public class AyUserController {

@ModelAttribute("name")

public String init(@RequestParam(value = "name", required = false) String name){

return name;

}

@GetMapping("/hello")

public String hello(){

return "hello";

}

}

上述代码中，@ModelAttribute注解标注的init方法带有返回值，@ModelAttribute("name")的value属性值为key，而init方法值为value，类似于：

model.addAttribute("name",name);

对应前端src\main\webapp\WEB-INF\views\hello.jsp页面代码如下所示：

<%@page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8" isELIgnored="false"%>

<!DOCTYPE HTML>

<html>

<head>

<title>Getting Started: Serving Web Content</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

</head>

<body>

hello, ${name}

</body>

</html>

当在浏览器中输入请求URL：http://localhost:8080/user/hello时，浏览器显示“hello，ay”。

**作业：**

**3）**@ModelAttribute注解和@RequestMapping注解同时标记在一个方法上。

**4）**使用@ModelAttribute注解方法的参数。

### 5.1.8 @SessionAttribute和@SessionAttributes注解

### 5.1.9 @ResponseBody和@RequestBody注解

使用@ResponseBody和@RequestBody注解之前，需要在pom.xml文件中引入Jackson相关的依赖包，具体代码如下：

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.9.5</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.9.5</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.5</version>

</dependency>

作业：利用@RequestBody 自己写ajax请求，完成添加、更新用户等操作。

6.1 Sring MVC数据校验

### 6.1.1 Spring的Validation校验框架

Spring MVC提供了强大的数据校验功能，其中有两种方法可以验证输入：1）利用Spirng自带的Validation校验框架。2）利用JSR 303（Java检验规范）实现校验功能。

/\*\*

\* 描述：用户数据校验类

\* @author Ay

\* @create 2018/05/25

\*\*/

@Component

public class AyUserValidator implements Validator {

/\*\*

\* This Validator validates \*just\* AyUser instances

\*/

public boolean supports(Class clazz) {

return AyUser.class.equals(clazz);

}

public void validate(Object obj, Errors e) {

//指定errors对象、验证失败的字段、错误码

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

AyUser p = (AyUser) obj;

if (p.getAge() < 0) {

e.rejectValue("age", "年龄不能小于0岁");

} else if (p.getAge() > 150) {

e.rejectValue("age", "年龄不能超过150岁");

}

}

}

<%@page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8" isELIgnored="false"%>

<!DOCTYPE HTML>

<html>

<head>

<title>Getting Started: Serving Web Content</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

</head>

<body>

<form method="post" action="/user/insert" >

<table>

<tr>

<td>姓名：</td>

<td><input id="name" name="name" type="text"></td>

</tr>

<tr>

<td>密码：</td>

<td><input id="password" name="password" type="text"></td>

</tr>

<tr>

<td>年龄：</td>

<td><input id="age" name="age" type="text"></td>

</tr>

<tr>

<td><input type="submit" value="提交"></td>

</tr>

</table>

</form>

</body>

<script></script>

</html>

### 6.1.2 JSR 303校验

表7-1 JSR 303包含注解

|  |  |
| --- | --- |
| *注解名称* | *描述* |
| *@Null* | *被注释的元素必须为 null* |
| *@NotNull* | *被注释的元素必须不为null* |
| *@AssertTrue* | *被注释的元素必须为true* |
| *@AssertFalse* | *被注释的元素必须为false* |
| *@Min(value)* | *被注释的元素必须是一个数字，其值必须大于等于指定的最小值* |
| *@Max(value)* | *被注释的元素必须是一个数字，其值必须小于等于指定的最大值* |
| *@DecimalMin(value)* | *被注释的元素必须是一个数字，其值必须大于等于指定的最小值* |
| *@DecimalMax(value)* | *被注释的元素必须是一个数字，其值必须小于等于指定的最大值* |
| *@Size(max=, min=)* | *被注释的元素的大小必须在指定的范围内* |
| *@Digits (integer, fraction)* | *被注释的元素必须是一个数字，其值必须在可接受的范围内* |
| *@Past* | *被注释的元素必须是一个过去的日期* |
| *@Future* | *被注释的元素必须是一个将来的日期* |
| *@Pattern(regex=,flag=)* | *被注释的元素必须符合指定的正则表达式* |

HIbernate Validator是JSR 303的一个参考实现，除了支持所有标准的校验注解外，它还支持一些扩展注解，具体内容如表7-2所示。

表7-2 Hibernate Validator 附加的注解

|  |  |
| --- | --- |
| *注解名称* | *描述* |
| *@NotBlank(message =)* | *验证字符串非null，且长度必须大于0* |
| *@Email* | *被注释的元素必须是电子邮箱地址* |
| *@Length(min=,max=)* | *被注释的字符串的大小必须在指定的范围内* |
| *@NotEmpty* | *被注释的字符串的必须非空* |
| *@Range(min=,max=,message=)* | *被注释的元素必须在合适的范围内* |
| *@URL* | *被注释的元素必须是合法的url* |

下面来看一个具体的实例。

首先，在pom.xml文件中引入HIbernate Validator所需要的依赖包，具体代码如下所示：

<dependency>

<groupId>org.hibernate.validator</groupId>

<artifactId>hibernate-validator</artifactId>

<version>6.0.10.Final</version>

</dependency>