## 一、配置文件

### 1. 两种注解扫描的区别

#### 1.1 使用MapperScannerConfigurer类

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

<!-- 扫描映射文件比如:AccountMapper.xml 加载该文件; 还会扫描接口AccountMapper接口创建代理对象 -->

<property name="basePackage" value="com.lagou.mapper"/></bean>

#### 1.2 使用context:component-scan标签

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

**区别**：

第一种方法的作用有两个：第一个是加载com.lagou.mapper包下的xxxMapper.xml文件；第二个作用是为该包下的接口创建代理对象，存放在IOC容器中；通常是用在dao层，为该层中的接口创建动态代理对象。

第二种方法的作用是对com.lagou.service包下的类创建实例对象放到IOC容器中；通常用在service层，为该层下的实现类创建对象。

### 2. 各层配置文件的加载位置

#### 2.1 mybatis配置文件

在spring配置文件中创建SqlSessionFactoryBean实例对象的时候引入mybatis配置文件。

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

<!-- 注入连接池, public void setDataSource(DataSource dataSource){...}-->

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

<!-- 加载mybatis-config.xml

public void setConfiguration(Configuration configuration) {this.configuration = configuration;}

因为在SqlSessionFactoryBean类中有configuration的set方法，所以可以使用property来注入

-->

<property name="configLocation" value="classpath:mybatis/mybatis-config.xml"/></bean>

#### 2.2 spring配置文件

在web.xml配置文件中创建监听器的时候加载spring所有的配置文件(如果spring配置文件是根据层分别创建的)

<!--监听器，加载spring所有的配置文件--><context-param>

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

<param-value>classpath:spring/applicationContext-\*.xml</param-value></context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

#### 2.3 加载springMVC配置文件

在web.xml配置文件中创建前端控制器的时候加载springmvc的配置文件

<!--前端控制器--><servlet>

<servlet-name>DispatcherServlet</servlet-name>

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

<init-param>

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

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

</init-param>

<load-on-startup>2</load-on-startup></servlet>

<servlet-mapping>

<servlet-name>DispatcherServlet</servlet-name>

<url-pattern>/</url-pattern></servlet-mapping>

### 3. 加载properties配置文件

#### 3.1 方式一

<properties resource="jdbc.properties"></properties>

#### 3.2 方式二

<!--需要引入context命名空间--><context:property-placeholder location="classpath:jdbc.properties"/>

### 4. 加载映射配置文件

#### 4.1 mybatis核心配置文件加载映射配置文件

<mappers>

<!--:class属性的方式

注意：这种方式下xml文件的位置一定要和接口的位置同包同名

比如：接口包是com.lagou.mapper,那么UserMapper.xml的位置也应该是在com.lagou.mapper下

-->

<mapper class="com.lagou.mapper.UserMapper"></mapper></mappers>

#### 4.2 spring核心配置文件加载映射配置文件

<!--Mapper包扫描--><bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 扫描映射文件比如:AccountMapper.xml 加载该文件; 还会扫描接口AccountMapper接口创建代理对象 -->

<property name="basePackage" value="com.lagou.mapper"/></bean>

## 二、SSM

### 1. mybatis多条件查询

#### 1.1 重要标签

resultMap标签

在实体属性名与表中属性名不一致的时候，使用该标签进行映射。

#### 1.2 多条件查询

适用于接口方法的参数有多个的情况

**方式一**

在XXXMapper.xml配置文件中需要使用到#{arg0}或者#{param1}

<select id="findByIdAndUsername1" resultType="com.lagou.domain.User">

select \* from user where id = #{arg0} and username = #{arg1}</select>

**方式二**

使用注解，引入 @Params() 注解获取参数

//接口中，方法的参数使用@Params()注解public User findByIdAndName(@Params("id") int id, @(Params"username") String username);

<!--配置文件中直接使用注解中指定的名字--><select id="findByIdAndUsername1" resultType="com.lagou.domain.User">

select \* from user where id = #{id} and username = #{username}</select>

**方式三**（推荐）

使用pojo对象的方式，将查询的结果封装到实体。因此接口中的方法的参数也是一个实体对象。

public User findByIdAndName(User user);

<select id="findByIdAndUsername3" parameterType="com.lagou.domain.User" resultType="com.lagou.domain.User">

<!--注意，这里#{}里面的值就需要和实体的属性名保持一致了-->

select \* from user where id = #{id} and username = #{username}</select>

### 2. mybatis多表查询

#### 2.1 联合查询

##### 2.1.1 一对一查询

association标签、resultZMap标签 结合使用

<resultMap id="orderMap" type="com.lagou.domain.Order">

  <id column="id" property="id"></id>

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

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

  <!--

一对一（多对一）使用association标签关联

        property="user" 封装实体的属性名

        javaType="user" 封装实体的属性类型

    -->

  <association property="user" javaType="com.lagou.domain.User">

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

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

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

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

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

  </association></resultMap>

##### 2.1.2 一对多查询

collection标签、resultMap标签结合使用

<resultMap id="userMap" type="com.lagou.domain.User">

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

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

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

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

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

<!--

一对多使用collection标签关联

property="orderList"  封装到集合的属性名

ofType="order"     封装集合的泛型类型

-->

<collection property="orderList" ofType="com.lagou.domain.Order">

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

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

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

</collection></resultMap>

##### 2.1.2 多对多查询

同一对多查询差不多，只是在编写SQL语句的时候，需要关联一个中间表。

SELECT \* FROM USER u LEFT JOIN user\_role ur ON u.`id`=ur.`uid` INNER JOIN role r ON ur.`rid` = r.`id`;-- user\_role就是一个中间表

#### 2.2 嵌套查询

只是把联合查询语句分成多个部分分步查询

##### 2.2.1 一对一嵌套查询

association标签中要添加select属性。其他的都和联合查询差不多

<!--一对一嵌套查询--><resultMap id="orderMap" type="order">

  <id column="id" property="id"></id>

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

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

  <!--根据订单中uid外键，查询用户表,即将uid作为参数传递到下面的查询语句-->

  <association property="user" javaType="com.lagou.domain.User" column="uid"

select="com.lagou.mapper.UserMapper.findById">

</association></resultMap>

<select id="findAllWithUser" resultMap="orderMap" >

 SELECT \* FROM orders</select>

##### 2.2.2 一对多嵌套查询

<!--一对多嵌套查询--><resultMap id="userMap" type="user">

  <id column="id" property="id"></id>

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

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

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

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

  <!--根据用户id，查询订单表-->

  <collection property="orderList" column="id" ofType="com.lagou.domain.Order"select="com.lagou.mapper.OrderMapper.findByUid"></collection></resultMap>

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

 SELECT \* FROM `user`</select>

##### 2.2.3 多对多嵌套查询

和一对多嵌套查询一样，只是SQL语句编写不一致。

<!--多对多嵌套查询--><resultMap id="userAndRoleMap" type="user">

  <id column="id" property="id"></id>

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

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

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

  <result column="adress" property="address"></result>

  <!--根据用户id，查询角色列表-->

  <collection property="roleList" column="id" ofType="com.lagou.domain.Role"select="com.lagou.mapper.RoleMapper.findByUid"></collection></resultMap>

<select id="findAllWithRole" resultMap="userAndRoleMap">

SELECT \* FROM `user`</select>

### 3. mybatis开发总结

* 使用动态代理的方式开发：持久层只编写接口和Mapper.xml文件，不编写实现类。
* XXXMapper.xml配置文件的位置必须要和接口所处的位置一致(比如：com.lagou.mapper)，配置文件名字也需要同接口的名字保持一致(比如：UserMapper)。
* mybatis传统开发方式：在实现类中存在代码重复和硬编码的问题，实际开发中不用。
* 尽量使用配置文件的方式开发，不用纯注解的方式开发，因为纯注解开发要修改源码，不好维护。

### 4. spring两大核心

#### 4.1 IOC

把对象的创建权利交给spring，降低代码的耦合度。

在mybatis开发的测试方法中，需要new SqlSessionFactoryBuilder()。有了spring之后，就可以让spring来创建该对象。

在之前service层调用dao的方法时，需要手动创建 new UserDaoImpl()。有了spring之后，就可以让spring来创建该对象。

#### 4.2 AOP

在不修改源码的情况下，对方法的功能进行增强(转账案例)。

### 5. spring IOC开发

**使用注解结合配置文件开发(推荐)**

使用注解开发必须在xml文件中配置注解扫描

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

注解代替bean标签

<!--xml方式开发--><bean id="userDao" class="com.lagou.dao.impl.UserDaoImpl"></bean>

// 注解开发@Repository // 如果没有写value属性值，Bean的id为：类名首字母小写public class UserDaoImpl implements UserDao {

}

注解代替依赖注入

<!--xml方式开发--><bean id="userService" class="com.lagou.service.impl.UserServiceImpl">

  <property name="userDao" ref="userDaoImpl"/></bean>

// 注解开发@Servicepublic class UserServiceImpl implements UserService {

  @Autowired

  private UserDao userDao;

// 去掉set方法

注解方式代替测试类

// xml方式开发，调用spring的API加载xml文件，并生成对象存放到IOC容器中public class AccountServiceTest {

ClassPathXmlApplicationContext classPathXmlApplicationContext = new ClassPathXmlApplicationContext("applicationContext.xml");

AccountService accountService = (AccountService) classPathXmlApplicationContext.getBean("accountService");

...

}

// 注解结合xml开发的测试类@RunWith(SpringJUnit4ClassRunner.class) //@RunWith指定junit的运行环境 SpringJUnit4ClassRunner是spring提供的作为junit运行环境的类@ContextConfiguration({"classpath:applicationContext.xml"})public class AccountServiceTest {

...

}

// 纯注解的测试类@RunWith(SpringJUnit4ClassRunner.class) //@RunWith指定junit的运行环境 SpringJUnit4ClassRunner是spring提供的作为junit运行环境的类@ContextConfiguration(classes = {SpringConfig.class})public class AccountServiceTest {

...

}

### 6. spring AOP开发

**使用注解结合配置文件开发(推荐)**

#### 6.1 基于xml方式的AOP开发

创建目标接口和目标实现类(即要给哪个对象增强功能)

public interface AccountService {

   public void transfer();

}

public class AccountServiceImpl implements AccountService {

@Override

public void transfer() {

System.out.println("转账业务...");

}

}

创建通知类

public class MyAdvice {

public void before() {

System.out.println("前置通知...");

}

}

将目标类和通知类对象创建权交给spring

<!--目标类交给IOC容器--><bean id="accountService" class="com.lagou.service.impl.AccountServiceImpl"></bean><!--通知类交给IOC容器--><bean id="myAdvice" class="com.lagou.advice.MyAdvice"></bean>

在核心配置文件中配置织入关系，及切面

<aop:config>

    <!--引入通知类-->

    <aop:aspect ref="myAdvice">

      <!--配置目标类的transfer方法执行时，使用通知类的before方法进行前置增强-->

      <aop:before method="before"

            pointcut="execution(public void

com.lagou.service.impl.AccountServiceImpl.transfer())">

</aop:before>

    </aop:aspect></aop:config>

其中，execution(public void com.lagou.service.impl.AccountServiceImpl.transfer())是切面表达式。

可以将切面表达式单独抽取出来，方便使用

<aop:config>

  <!--抽取的切点表达式-->

<aop:pointcut id="myPointcut" expression="execution(\* com.lagou.service..\*.\*

(..))"> </aop:pointcut>

  <aop:aspect ref="myAdvice">

    <aop:before method="before" pointcut-ref="myPointcut"></aop:before>

  </aop:aspect></aop:config>

#### 6.2 基于注解的AOP开发

创建目标接口和实现类

public interface AccountService {

public void transfer();

}

public class AccountServiceImpl implements AccountService {

@Override

public void transfer() {

System.out.println("转账业务...");

}

}

创建通知类

public class MyAdvice {

public void before() {

System.out.println("前置通知...");

}

}

将目标类和通知类对象创建权交给spring

@Servicepublic class AccountServiceImpl implements AccountService {}@Componentpublic class MyAdvice {}

在通知类中使用注解配置织入关系，升级为切面类

@Component@Aspectpublic class MyAdvice {

@Before("execution(\* com.lagou..\*.\*(..))")

public void before() {

System.out.println("前置通知...");

}

}

这里也可以抽取切面表达式

@Component@Aspectpublic class MyAdvice {

@Pointcut("execution(\* com.lagou..\*.\*(..))")

public void myPoint(){}

@Before("MyAdvice.myPoint()")

public void before() {

System.out.println("前置通知...");

 }

在配置文件中开启组件扫描和 AOP 的自动代理

<!--组件扫描--><context:component-scan base-package="com.lagou"/><!--aop的自动代理--><aop:aspectj-autoproxy></aop:aspectj-autoproxy>

#### 6.3 对比xml开发和注解开发

* 注解方式：在接口实现类添加@Service注解；在通知类上添加@Component和@Aspect注解，织入关系就在通知类里面编写。
* XML方式：对象创建是在spring配置文件中编写，织入关系也是在配置文件中编写。

#### 6.4 spring中的事务控制

**声明式事务控制(开发中常用)**

##### 6.4.1 基于XML的声明式事务控制

解决什么问题呢？在执行一个业务的时候(比如转账操作)，如果全部执行成功就提交事务，如果出现异常就回滚事务。这中间就可以通过配置事务管理器来实现

主要配置如下：

**事务管理器通知配置**

<!--事务管理器--><bean id="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

  <property name="dataSource" ref="dataSource"></property></bean><!--通知增强--><tx:advice id="txAdvice" transaction-manager="transactionManager">

  <!--定义事务的属性-->

  <tx:attributes>

    <tx:method name="\*"/>

  </tx:attributes></tx:advice>

**事务管理器AOP配置**

<!--aop配置--><aop:config>

  <!--切面配置-->

  <aop:advisor advice-ref="txAdvice"

        pointcut="execution(\* com.lagou.serivce..\*.\*(..))">

  </aop:advisor></aop:config>

**总结：**

相比较于之前基于xml方式的AOP开发，我们需要编写一个通知类，然后在该类中编写前置通知、后置通知、异常通知、最终通知的业务逻辑，然后再配置文件中指定通知。

现在通过事务管理器，我们只需要在配置文件中添加一个事务管理器对象的bean标签，然后在tx:advice标签中引入该事务管理器对象，最后在切面配置中配置该通知就可以了。不需要再去创建一个通知类，然后写各种通知的处理逻辑。

事务管理器AOP配置使用的是aop:advisor标签，而基于xml方式的AOP配置使用的是aop:aspect标签。

##### 6.4.2 基于注解的声明式事务控制

直接在service层的实现类中需要事务控制的方法上增加@Transactional注解，或者直接在类上添加@Transactional注解

@Servicepublic class AccountServiceImpl implements AccountService {

@Autowired

private AccountDao accountDao;

@Transactional(propagation = Propagation.REQUIRED, isolation =

Isolation.REPEATABLE\_READ, timeout = -1, readOnly = false)

@Override

public void transfer(String outUser, String inUser, Double money) {

accountDao.out(outUser, money);

int i = 1 / 0;

accountDao.in(inUser, money);

}

}

然后在配置文件中配置如下：

<!--事务管理器-->

  <bean id="transactionManager"class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

  </bean>

<!--事务的注解支持--><tx:annotation-driven/>

其他的就都不用配置了

### 7. springMVC开发

#### 7.1 springMVC三大组件

处理器映射器：找到Controller类中的需要执行的方法，返回给前端控制器。

@Controller@RequestMapping("/user")@Transactionalpublic class UserController {

@Autowired

private IUserService userService;

@RequestMapping(value = "/login.do" , method = RequestMethod.POST)

@ResponseBody

public ServiceResponse<User> login(String username, String password, HttpSession session) {

ServiceResponse<User> response = userService.login(username, password);

// 判断是否登录成功

if (response.getStatus() == ResponseCode.SUCCESS.getCode()) {

session.setAttribute(Const.CURRENT\_USER, response.getData());

}

return response;

}

// 比如用户通过浏览器访问www.lagou.com/user/login.do，那么先经过前端控制器，然后前端控制器将请求发送给处理器映射器，处理器映射器就到controller层去找‘/user/login.do’资源路径，看看能匹配到哪个类中的哪个方法。// 最后找到UserController类中的login方法// 上面的处理器映射器是通过xml的方式进行指定的，如果是通过注解或者接口的方式进行指定，那么springMVC都可以实现不同的映射方式。

处理器适配器：能够对多个Controller类进行适配

视图解析器

#### 7.2 配置文件

web.xml

配置前端控制器

<!--前端控制器,加载springmvc核心配置文件--><servlet>

<servlet-name>DispatcherServlet</servlet-name>

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

<init-param>

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

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

</init-param>

<load-on-startup>2</load-on-startup></servlet>

<servlet-mapping>

<servlet-name>DispatcherServlet</servlet-name>

<url-pattern>\*.do</url-pattern></servlet-mapping>

配置监听器

<!--监听器，加载spring所有的配置文件--><context-param>

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

<param-value>classpath:spring/applicationContext-\*.xml</param-value></context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

中文乱码过滤器

<!--配置全局过滤的filter--><filter>

  <filter-name>CharacterEncodingFilter</filter-name>

  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

  <init-param>

    <param-name>encoding</param-name>

    <param-value>UTF-8</param-value>

  </init-param></filter><filter-mapping>

  <filter-name>CharacterEncodingFilter</filter-name>

  <url-pattern>/\*</url-pattern></filter-mapping>

spring-mvc.xml

配置处理器映射器和处理器适配器

<!--处理器映射器和适配器增强--><mvc:annotation-driven></mvc:annotation-driven>

配置视图解析器

<!--视图解析器--><bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

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

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

开启静态资源

<!--在springmvc配置文件中开启DefaultServlet处理静态资源--><mvc:default-servlet-handler/>

配置自定义类型转换器

SpringMVC 默认已经提供了一些常用的类型转换器；例如：客户端提交的字符串转换成int型进行参  
数设置，日期格式类型要求为：yyyy/MM/dd 不然的话会报错，对于特有的行为，SpringMVC提供了自  
定义类型转换器方便开发者自定义处理。

<!--自定义转换器配置--><bean id="conversionService"class="org.springframework.context.support.ConversionServiceFactoryBean">

  <property name="converters">

    <set>

      <bean class="com.lagou.converter.DateConverter"></bean>

    </set>

  </property></bean>

public class DateConverter implements Converter<String, Date> {

public Date convert(String dateStr) {

//将日期字符串转换成日期对象 返回

SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");

Date date = null;

try {

date = format.parse(dateStr);

} catch (ParseException e) {

e.printStackTrace();

}

return date;

}

}

配置自定义异常处理器

public class GlobalExceptionResolver  implements HandlerExceptionResolver {

@Override

public ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex) {

ModelAndView modelAndView = new ModelAndView();

modelAndView.addObject("error", ex.getMessage());

modelAndView.setViewName("error");

return modelAndView;

}

}

<bean id="globalExecptionResovler"

   class="com.lagou.exception.GlobalExecptionResovler"></bean>

扫描路径

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

### 8. 方法参数名与请求参数名不一致

#### 8.1 dao层接口方法的参数名与请求的参数名不一致

使用的是@Params() 注解

//接口中，方法的参数使用@Params()注解public User findByIdAndName(@Params("id") int id, @(Params"username") String username);

#### 8.2 controller层业务方法的参数名与请求的参数不一致

使用@RequestParam注解

<!--请求时传递的参数名--><a href="${pageContext.request.contextPath}/user/findByPage?pageNo=2">

 分页查询</a>

@RequestMapping("/findByPage")public String findByPage(@RequestParam(name = "pageNo", defaultValue = "1")

Integer pageNum, @RequestParam(defaultValue = "5") Integer pageSize) {

  System.out.println(pageNum);

  System.out.println(pageSize);

  return "success";

}

### 9. ajax异步交互

#### 9.1 @RequestBody注解

该注解用于Controller的方法的形参声明，当使用ajax提交并指定contentType为json形式时，通过  
HttpMessageConverter接口转换为对应的POJO对象。

<button id="btn1">ajax异步提交</button><script>

  $("#btn1").click(function () {

    let url = '${pageContext.request.contextPath}/ajaxRequest';

    let data = '[{"id":1,"username":"张三"},{"id":2,"username":"李四"}]';

    $.ajax({

      type: 'POST',

      url: url,

      data: data,

      contentType: 'application/json;charset=utf-8',

      success: function (resp) {

alert(JSON.stringify(resp))

     }

   })

 })</script>

@RequestMapping(value = "/ajaxRequest")public void ajaxRequest(@RequestBody List<User>list) {

  System.out.println(list);

}

#### 9.2 @ResponseBody注解

该注解用于将Controller的方法返回的对象，通过HttpMessageConverter接口转换为指定格式的数  
据如：json,xml等，通过Response响应给客户端。

// 默认返回的数据是json格式@RequestMapping(value = "/ajaxRequest")@ResponseBodypublic List<User> ajaxRequest(@RequestBody List<User> list) {

  System.out.println(list);

  return list;

}

### 10 springMVC中的过滤器、拦截器和前端控制器

#### 10.1 三者的执行顺序

过滤器-->拦截器-->前端控制器

#### 10.2 三者的功能

springMVC的拦截器，只能拦截地址栏访问对controller的请求，无论你拦截的地址配置到天上，它只拦截对controller的请求。

springMVC的过滤器，这个会根据你配置的路径选择性拦截，拦截什么请求路径，按个人口味选择；

springMVC的前端控制器，这个也是根据你配置的路径选择性拦截，若直接配置根路径，静态资源也会被拦截；

#### 10.3 拦截器和过滤器的区别

过滤器是Servlet中的一部分，任何Javaweb工程都可以使用；而拦截器是springmvc自己的，只有使用了springmvc框架才可以使用拦截器。

过滤器可以对所有要访问的资源进行拦截；而拦截器只能拦截控制器类中的方法，如果访问js、html、css、jsp、image是不会拦截的。

## 三、SSM整合开发

### 1. spring整合mybatis

将对象的创建权交给spring，原本使用单独的mybatis开发时，在测试类中会手动创建接口的代理类对象，代码如下：

public class MybatisTest {

@Test

public void testFindAll() throws IOException {

InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

AccountDao mapper = sqlSession.getMapper(AccountDao.class);

List<Account> accountList = mapper.findAll();

for (Account account : accountList) {

System.out.println(account);

}

}

}

通过spring整合mybatis之后，就不要手动去创建了，只需要在spring的核心配置文件中进行配置就可以了，代码如下：

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

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

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

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

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

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

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

<!--spring配置文件中起别名的方法-->

<property name="typeAliasesPackage" value="com.lagou.domain"></property></bean>

<!--将AccountDAO接口的实现类对象交给spring创建,需要使用以下类对接口进行扫描，创建接口实现类--><bean id="mapperScannerConfigurer" class="org.mybatis.spring.mapper.MapperScannerConfigurer">

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

值得一提的是：在sqlSessionFactory标签中，可以配置很多的属性。比如：

<!--加载mybatis核心配置文件--><property name="configLocation" value="classpath:mybatis/mybatis-config.xml"/>

在spring的核心配置文件中配置以上配置之后，原本在mybatis中的以下配置就可以删除了

<!--配置别名--><typeAliases>

<typeAlias type="com.lagou.domain.Account" alias="account"></typeAlias></typeAliases>

<!--environments:运行环境--><environments default="development">

<!--可以配置多个环境-->

<environment id="development">

<!--当前的事务管理器是JDBC-->

<transactionManager type="JDBC"></transactionManager>

<!--数据源信息

POOLED:使用mybatis的连接池

UNPOOLED:不使用连接池

-->

<dataSource type="POOLED">

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

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

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

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

</dataSource>

</environment></environments>

<!--引入映射配置文件，动态代理的时候用到--><mappers>

<mapper class="com.lagou.dao.AccountDao"></mapper></mappers>

spring整合mybatis之后，测试的时候只需要将dao层的接口注入到service层就可以了，不需要再手动创建，代码如下：

public class AccountServiceImpl implements AccountService {

// 注入接口

@Autowired

private AccountDao accountDao;

@Override

public List<Account> findAll() {

List<Account> accountList = accountDao.findAll();

return accountList;

}

@Override

public void save(Account account) {

accountDao.save(account);

}

}

@RunWith(SpringJUnit4ClassRunner.class)@ContextConfiguration("classpath:applicationContext.xml")public class SpringTest {

@Autowired

private AccountService accountService;

@Test

public void testFindAll() {

accountService.findAll();

}

}

### 2. spring整合springmvc

整合的地方就是让springmvc加载spring的配置文件，在web.xml文件中整合如下：

<!--web.xml--><!--监听器，其作用是监听ServletContext容器，一旦启动就加载spring的核心配置文件--><listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class></listener>

<context-param>

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

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

## 四、springboot开发

### 1. 全局配置文件

Spring Boot使用一个application.properties或者application.yaml的文件作为全局配置文件，该文件存放在src/main/resource目录或者类路径的/config

#### 1.1 application.properties配置文件

可以配置系统属性、环境变量、命令参数等信息，也可以是自定义配置文件名称和位置。

# tomcat服务端口server.port=8081# spring相关配置spring.datasource.driver-class-name=com.mysql.jdbc.Driverspring.config.additional-location=spring.config.location=spring.config.name=application# person实体类相关配置person.id=1person.name=tomperson.hobby=吃饭,睡觉,打游戏person.family=father,motherperson.map.k1=v1person.map.k2=v2person.pet.type=dogperson.pet.name=旺财

#### 1.2 application.yaml配置文件

YAML文件格式是Spring Boot支持的一种JSON超集文件格式，相较于传统的Properties配置文件，YAML文件以数据为核心，是一种更为直观且容易被电脑识别的数据序列化格式。

**注意**：冒号后面必须要有一个空格

value值为普通数据类型（例如数字、字符串、布尔等）

server:

port: 8080

servlet:

context-path: /hello

value值为数组和单列集合

person:

hobby: [play,read,sleep]

value值为Map集合和对象

person:

map: {k1: v1,k2: v2}

#### 1.3 注意事项

使用application.yaml配置文件进行测试时需要提前将application.properties配置文件中编写的配置注释，这是因为application.properties配置文件会覆盖application.yaml配置文件。

### 2. 配置文件属性值的注入

使用Spring Boot全局配置文件设置属性时：

如果配置属性是Spring Boot已有属性，例如服务端口server.port，那么Spring Boot内部会自动扫描并读取这些配置文件中的属性值并覆盖默认属性。

如果配置的属性是用户自定义属性，例如刚刚自定义的Person实体类属性，还必须在程序中注入这些配置属性方可生效。

Spring Boot支持多种注入配置文件属性的方式，下面来介绍如何使用注解@ConfigurationProperties和@Value注入属性

#### 2.1 使用@ConfigurationProperties注入属性

@Component@ConfigurationProperties(prefix = "person")public class Person {

private int id;

// 属性的setXX()方法

public void setId(int id) {

this.id = id;

}

}

上述代码使用@Component和@ConfigurationProperties(prefix = “person”)将配置文件中的每个属性映射到person类组件中。

#### 2.2 使用@Value注入属性

@Value注解是Spring框架提供的，用来读取配置文件中的属性值并逐个注入到Bean对象的对应属性中，Spring Boot框架从Spring框架中对@Value注解进行了默认继承，所以在Spring Boot框架中还可以使用该注解读取和注入配置文件属性值。使用@Value注入属性的示例代码如下:

@Componentpublic class Person {

@Value("${person.id}")

private int id;

}

上述代码中，使用@Component和@Value注入Person实体类的id属性。其中，@Value不仅可以将配置文件的属性注入Person的id属性，还可以直接给id属性赋值，这点是@ConfigurationProperties不支持的.

@Componentpublic class Student {

@Value("${person.id}")

private int id;

@Value("${person.name}")

private String name; //名称

//省略toString

 }

Student类使用@Value注解将配置文件的属性值读取和注入。  
从上述示例代码可以看出，使用@Value注解方式需要对每一个属性注入设置，同时又免去了属性的setXX()方法.

### 3. 自定义配置文件

#### 3.1 使用@PropertySource加载配置文件

打开Spring Boot项目的resources目录，在项目的类路径下新建一个test.properties自定义配置文件，在该配置文件中编写需要设置的配置属性

#对实体类对象MyProperties进行属性配置test.id=110test.name=test

在com.lagou.pojo包下新创建一个配置类MyProperties，提供test.properties自定义配置文件中对应的属性，并根据@PropertySource注解的使用进行相关配置

@Component  // 自定义配置类@PropertySource("classpath:test.properties")  // 指定自定义配置文件位置和名称@ConfigurationProperties(prefix = "test")    // 指定配置文件注入属性前缀public class MyProperties {

private int id;

private String name;

// 省略属性getXX()和setXX()方法

// 省略toString()方法

}

说明：

@PropertySource("classpath:test.properties")注解指定了自定义配置文件的位置和名称，此示例表示自定义配置文件为classpath类路径下的test.properties文件；

@ConfigurationProperties(prefix = "test")注解将上述自定义配置文件test.properties中以test开头的属性值注入到该配置类属性中。

#### 3.2 使用@Configuration编写自定义配置类

// 实体类public class MyService {

}

@Configuration  // 定义该类是一个配置类````public class MyConfig {

@Bean     // 将返回值对象作为组件添加到Spring容器中，该组件id默认为方法名

public MyService myService(){

return new MyService();

}

}

MyConfig是@Configuration注解声明的配置类（类似于声明了一个XML配置文件），该配置类会被Spring Boot自动扫描识别；使用@Bean注解的myService()方法，其返回值对象会作为组件添加到了Spring容器中（类似于XML配置文件中的标签配置），并且该组件的id默认是方法名myService.

### 4. SpringBoot整合MyBatis

#### 4.1 编写与数据表对应的实体类

public class Comment {

private Integer id;

private String content;

private String author;

private Integer aId;

// 省略属性getXX()和setXX()方法

// 省略toString()方法

}

public class Article {

private Integer id;

private String title;

private String content;

// 省略属性getXX()和setXX()方法

// 省略toString()方法

}

#### 4.2 在application.properties配置文件中进行数据库连接配置

# MySQL数据库连接配置spring.datasource.url=jdbc:mysql://localhost:3306/springbootdata?serverTimezone=UTC&characterEncoding=UTF-8spring.datasource.username=rootspring.datasource.password=root

#### 4.3 注解方式整合Mybatis

创建一个对t\_comment表数据操作的接口CommentMapper

@Mapperpublic interface CommentMapper {

@Select("SELECT \* FROM t\_comment WHERE id =#{id}")

public Comment findById(Integer id);

}

需要在类上添加@Mapper注解， 然后在方法上添加@Select注解

#### 4.4 使用配置文件的方式整合mybatis

创建一个用于对数据库表t\_article数据操作的接口ArticleMapper

@Mapperpublic interface ArticleMapper {

public Article selectArticle(Integer id);

}

创建XML映射文件

resources目录下创建一个统一管理映射文件的包mapper，并在该包下编写与ArticleMapper接口对应的映射文件ArticleMapper.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.lagou.mapper.ArticleMapper">

  <select id="selectArticle" resultType="Article">

   select \* from Article

  </select></mapper>

在application.properties配置文件中做如下配置

#开启驼峰命名匹配映射mybatis.configuration.map-underscore-to-camel-case=true

#配置MyBatis的xml配置文件路径mybatis.mapper-locations=classpath:mapper/\*.xml

### 5. SpringBoot整合SSM

最核心的地方是要在application.yml配置文件中配置spring和mybatis的相关配置，其他的基本都一样的。

#服务器配置server:port: 8090servlet:

  context-path: /# spring相关配置spring:

datasource:

name: druid

type: com.alibaba.druid.pool.DruidDataSource

url: jdbc:mysql://localhost:3306/spring\_db?characterEncoding=utf-

8&serverTimezone=UTC

username: root

password: root

driver-class-name: com.mysql.jdbc.Driver

mvc:

view:

prefix: /

suffix: .html

# mybatis相关配置mybatis:

mapper-locations: classpath:mapper/\*Mapper.xml #声明Mybatis映射文件所在的位置