# spring ：

## 是什么：

是一个轻量级的开源框架，核心是基于控制反转(Inversion of Control)(也被成为依赖注入)和面向切面编程(aspect oriented programming)

## 能干吗

设值注入：由spring把组件所依赖的具体对象注入给组件,使得组件以配置文件而不是硬编码的方式耦合在一起a new调用b，而b改了构造方法，就得在所有关联修改,并不方便扩展，所以用bean

分为6部分Core Container, DataAccess/Integration ,web ,AOP ,Instrumentation ,test

docs：包含spring的相关文档 api、开发手册等

libs:各个模块的jar包 每个模块均提供

开发所需的jar包

javadoc后缀表示的api

-sources后缀表示的源文件

schema：配置某些功能需要用到的schema文件 对于已经集成了spring的ide如eclipse等这些文件不需要专门导入

config怎么写 ：

bean标签来定义一个实例(一个对象),有两个常用属性 id表示实例名称 class里写全类名表示实例类型 里面有property标签

有name属性表示给哪个属性赋值 对应的是set方法名

name为普通类型用value属性为其赋值,对象用ref属性注入已经声明出来的bean

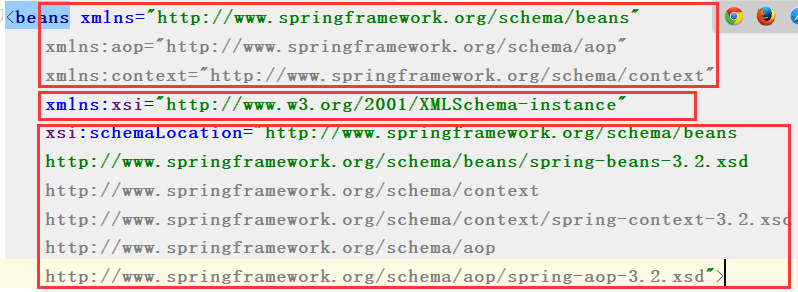
|  |
| --- |
| <**bean id="itripHotel" class="cn.travel.entity.ItripHotel"**>  <**property name="hotelName" value="如家"**></**property**> </**bean**> |

ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml"); 实例化spring上下文

负责读取Spring配置文件，管理对象的加载生成(加入到容器里)，维护bean对象之间的依赖关系，负责bean的生命周期等

用于从classpath中读取spring配置文件

UserService service = context.getBean("service"); 根据id来获取对应实例



可以理解为先声明容器，在声明xsi用于使用容器，在注册容器。

Beans

Aop 实现切面用的

Context：

## AOP(aspect oriented programming)：

是什么：面向切面编程 是对面向对象编程有益的补充,一般适用于具有横切逻辑的场合，如访问控制，事物管理，性能监测等。

能干吗：在不改变原程序的基础上为代码增加新功能，对代码进行增强处理

为了在增强方法中获得当前连接点信息，可以在方法中声明一个JoinPoint类型的参数，Spring会自动注入该实例

getTarget 获取被代理的目标对象

getSignature 返回被代理的目标方法 jp.getSignature().getName()

getArgs 返回传递给目标方法的参数数组 Arrays.toString(jp.getArgs())

|  |
| --- |
| **private static** Logger *logger*= Logger.*getLogger*(WebLogAspect.**class**.getName()); *logger*.info(**"调用"**+jp.getTarget().getClass().getName()+**"的"**+jp.getSignature().getName() +**"方法，方法入参"**+ Arrays.*toString*(jp.getArgs())); |

对于后置增强方法(after-returning前置为before)还可以定义一个参数用于接收目标方法的返回值，配置文件里returning属性的值必须等于参数名并且目标方法返回值和参数类型是一致的。

### 1、aop相关的配置：

<aop:config>

//定义一个切入点表达式，并命名为 pointcut

<aop:pointcut id="pointcut" expression="execution(\* cn.travel.service.\*.\*(..))"/>

只要满足这个切入点的表达式都会为其增加增强方法

//引入包含增强方法的bean <bean id=”webLogger” class=”cn.aop.WebLogger”>

<aop:aspect ref="webLogger">

//将bfrore方法定义为前置增强并引入pointcut切入点

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

//将after方法定义为后置并引用pointcut切入点 通过returning属性指定名为result的参数注入返回值

<aop:after-returning method="afterReturning" pointcut-ref="pointcut" returning="result"/>

</aop:aspect>

需要增强方法的service包里的类必须注入到容器里，例如现在的阶段必须在配置文件加上，然后getBean即可

<**bean class="cn.travel.service.UserServiceImpl" id="userServiceImpl"**></**bean**>

|  |
| --- |
| ApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext-mybatis.xml"**); UserServiceImpl userService = (UserServiceImpl) context.getBean(**"userServiceImpl"**); userService.asd(); |

### 2、execution()表达式支持模糊匹配

public \* addNewUser (entity User) \*表示匹配所有类型的返回值

public void \* (entity User) \*表示匹配所有方法名

public void addNewUser (..) ..表示匹配所有参数个数和类型

\* com.service.\*.\*(..) 这个表达式匹配com.service包下所有类的所有方法

\* com.service..\*.\*(..) 这个表达式匹配com.service包及其子包下所有类的所有方法

### 3、扩展增强：

异常抛出增强：在目标方法抛出异常时添加增强处理

标签为<aop:after-throwing 里面有thorwing属性对应Exception类型的参数名，Spring为其注入抛出异常的实例

前置增强before在逻辑代码之上执行

后置增强after-returning在逻辑代码之下执行 有returning属性，等于参数名并类型一致，方法出现异常则不执行

最终增强after相当于finally就算出现异常也正常执行

环绕增强：around可以在目标方法前后都增强处理

ProceedingJoinPoint为joinpoint子接口多封装了目标对象，通过proceed调用目标方法，用在环绕方法上

|  |
| --- |
| **public** Object aroundLogger(ProceedingJoinPoint jp){   *//随便输出点信息相当于前置增强(就是在目标方法之前输出)  logger*.info(**"调用"**+jp.getTarget().getClass().getName()+**"的"**+jp.getSignature().getName()  +**"方法，方法入参"**+ Arrays.*toString*(jp.getArgs()));  **try** {  Object result = jp.proceed();  *//随便输出点信息，相当于后置增强 出现异常就不会执行额  logger*.error(**"调用"**+jp.getTarget().getClass().getName()+**"的"**+jp.getSignature().getName()  +**"方法，方法返回值:"**+result);  **return** result;  } **catch** (Throwable e) {  *//异常增强  logger*.error(jp.getSignature().getName()+**"方法发生异常"**);  e.printStackTrace();  }**finally** {  *//最终增强  logger*.info(jp.getSignature().getName()+**"方法结束"**);  }  **return null**; } |

### 5、注解方式实现

1、使用注解定义切面： 类加上@Aspect注解

@Before("execution(\* test.\*.\*(..))")

@AfterReturning(pointcut = "execution(\* test.\*.\*(..))",returning = "re")

@AfterThrowing(pointcut = "pointcut()",throwing = "rx")

@After("execution(\* test.\*.\*(..))")

@Around("execution(\* test.\*.\*(..))") 方法记得用ProceedingJoinPoint参数

2、还定义一个公共切面 @Before("pointcut()") 这样使用即可

@Pointcut("execution(\* test.\*.\*(..))")

public void pointcut() {}

3、然后在配置文件完成织入即可

<bean class="cn.travel.aop.WebLogAspect"></bean>

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

### 4、注意：

每个增强类都得要无参构造方法

不写boot里aop的*半年数据迁移云备份那个东西*

需要增强方法的service包里的类必须注入到容器里，然后getbean即可

Server不用写名字，例如UserServiceImpl的默认容器名为userServiceImpl

<**context:component-scan base-package="cn.travel.service"**></**context:component-scan**>

|  |
| --- |
| ApplicationContext context = **new** ClassPathXmlApplicationContext(**"applicationContext-mybatis.xml"**); UserServiceImpl userService = (UserServiceImpl) context.getBean(**"userServiceImpl"**); userService.asd(); |

### 单词

Core 核心

Container 容器

Access 存取

Integration 集成

javadoc doc就是document文档

ApplicationContext 应用上下文

Signature 签名

pointcut 切入点 点，切入

expression 表达式

aspect 方面

inversion 倒置，反向

oriented 定向，导向

programming 编程

## 事务：

### 1、配置方式

|  |
| --- |
| *<!--定义事物管理器 提供了对事物处理的全面支持和统一管理-->* <**bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**bean**>  *<!-- 为指定的事务管理器设定属性，为业务方法指定具体事物规则 -->* <**tx:advice id="txAdvice" transaction-manager="txManager"**>  <**tx:attributes**> -->  *<!-- name用于指定匹配的方法，对方法名进行约定-->  <!-- propagation为事务传播机制-->* <**tx:method name="get\*" propagation="SUPPORTS"** />  <**tx:method name="add\*" propagation="REQUIRED"**/>  <**tx:method name="del\*" propagation="REQUIRED"**/>  <**tx:method name="update\*" propagation="REQUIRED"**/>  <**tx:method name="\*" propagation="REQUIRED"**/>  </**tx:attributes**> </**tx:advice**>  *<!-- 定义一个切面 然后advice链接事务规则 point链接切面-->*  <**aop:config**>  <**aop:pointcut id="serviceMethod" expression="execution(\* cn.travel.service.\*.\*(..))"**/>  <**aop:advisor advice-ref="txAdvice" pointcut-ref="serviceMethod"**/> </**aop:config**> |

### 2、注解方式

|  |
| --- |
| *<!--定义事物管理器 提供了对事物处理的全面支持和统一管理-->* <**bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  <**property name="dataSource" ref="dataSource"**></**property**> </**bean**>  *<!-- 启动事务注解-->* <**tx:annotation-driven transaction-manager="txManager"**/> |

然后在serviceImpl类上加 @Transactional 注解这样即可为该类的所有业务方法统一添加事务处理，如果某一个业务需要采用不同的事务规则，可以在业务方法上添加@Transactional(propagation = Propagation.SUPPORTS)注解单独设置 多个属性逗号隔开

默认的设置为： 事务传播设置是 PROPAGATION\_REQUIRED

事务隔离级别是 ISOLATION\_DEFAULT

事务是读/写

事务超时默认是依赖于事务系统的,或者事务超时没有被支持

任何RuntimeException将触发事务回滚,但是任何checked Exception将不触发事务回滚

propagation传播机制：当事务方法被另一个事务方法调用时,可以理解为被调用者必须指定事务应该如何传播

有一个单添加方法和多添加方法,多添加方法里面执行了两遍单添加

required 默认值,执行该方法时，如果已经存在一个事务就用当前的，没有的话则开启自己的事务

在多添加中执行到单添加的时候判断如果已经存在事务那么就用多添加自带的事务，那么就是2本书必须一起成功或一起失败

REQUIRES\_NEW 总是开启新事务，如果方法内存在另一个事务，则将它挂起

如果已经存在事务那么久把他吊起,用自己的事务,那么2本书有可能会发生一本成功,另一本失败(因为每个单添加都是单一的事务)

SUPPORTS 更多适用于查询，如果存在于一个事务中就加入到事务运行,如果没事务就非事务方法运行,查询事务重要性不是很大

isolation:事务的隔离等级,即当前事务和其他事务的隔离程度,在并发事务处理的情况下需要考虑它的设置

read\_uncommitted 未提交读

read\_committed 提交读

repeatable\_read 可重复读

serializable 可串行读

timeout:事务超时事件。允许事务运行的最长时间,以秒为单位,超过给定的事件自动回滚,防止事务执行时间过长影响性能,默认不超时 可以再方法加thread.sheep测试

read-only:事务是否为只读,默认false,对于只执行查询功能的事务,把他设置为true能提高性能

rollback-for:设定能够触发回滚的异常类型。Spring默认只在抛出runtimeException时才回滚,可以通过全限定类名自行指定需要回滚事务的异常,多个用逗号隔开

no-rollback-for:设定不触发回滚的异常类型,默认checked Exception 不会触发回滚

@Transactional(noRollbackFor = {RuntimeException.class})

### 单词：

destroy 破坏

advice 通知 为指定管理器设置属性

annotation 注释，注解

propagation 传播机制

isolation 隔离

## 构造注入：

在bean(绑定了一个实例)内嵌套<constructor-arg>

参数为对象：<ref bean="st"/>

参数为String 给构造标签里写value

st是已经获取到的实例(对应相应bean的id)用来赋值到当前这个bean的构造方法

如果为多个参数就写多个constr 然后有index(从0开始)来指定位置

|  |
| --- |
| <**bean id="itripHotel" class="cn.travel.entity.ItripHotel"**>  <**constructor-arg**>  <**value**>拉拉</**value**>  </**constructor-arg**>  <**constructor-arg**>  <**ref bean="webLogAspect"**></**ref**>  </**constructor-arg**>  </**bean**> |

构造注入时效性好在对象实例化时就得到所依赖的对象，便于在对象的初始化方法中使用依赖对象；但受限于重载的形式灵活性没有设值注入强

### 扩展：

如果属性包含了特殊字符，在value标签里写<![CDATA[xxx]]> 最里面的方括号随便写内容

<ref bean="st"/>还有一个属性是local 用法是一样的，spring配置文件是可以拆分成多个的，local属性只能在同一个config中检索bean的id，bean可以在其他config检索id。

比如都是<bean> <property name=”dao”> dao属性 下面的各个都写到这个里面

内部bean <bean class=”test.daoImpl”>

有list属性时候 <list><value>1</value><value>1</value>

map属性 <map><entry> <key><value>a</value></key>

<value>凯凯</value> </entry>

properties属性 <props> <prop key=”a”>凯凯<prop>

注入空值value里不写东西 注入null 把value标签替换为<null/>

## 注解开发

### 使用注解定义bean

@Component(“userDao”) 定义了一个当前类实例的bean名字为userDao

@Repository 用于标注DAO类

@Service 用于标注业务类

@Controller 用于标注控制器类

@Autowired 在属性上写，自动装配合适的依赖对象，若有多个匹配时,则可以用

@Qualifier(“userDao”) 指定所需bean名称

### 注意：

然后可以在配置文件直接 <context:component-scan base-package="test"></context:component-scan>来扫描指定的包来注册bean。多个包名逗号隔开

如果对类中集合类型的变量使用@Autowired，会将所有和集合中元素类型匹配的bean组件都注入

### 单词

transaction 事务 穿sai ke 神

Manager 管理 mai nei zher

constructor 构造器 ken si chua ke ter

around 环绕 饿 ruang de

Component 组成的 ken po nen te

标注组件和组件 scan扫描

Repository 标注dao类 rui po si te rui

Autowired 自动匹配

Qualifier 限定符 k哦 里 fai er

base 基础 bei si

proxy 代理 pro ke sei

## 整合mybatis

spring对myBatis整合：需要导入整和用的三个jar包和dbcp两个jar包和mybatis和mysql包

mybatis映射文件照常，核心文件只写个别名和<mapper resource="UserMapper.xml"></mapper>，别的让Spring接管

但是用idea maven项目如果xml在mapper目录下那么需要

|  |
| --- |
| <**build**>  <**resources**>  <**resource**>  <**directory**>src/main/java</**directory**>  <**includes**>  <**include**>\*\*/\*.xml</**include**>  </**includes**>  </**resource**>  </**resources**> </**build**> |

|  |
| --- |
| *<!-- 获取数据库配置文件 -->* <**context:property-placeholder location="classpath:database.properties"**/>  *<!--配置数据源 (使用dbcp连接池) -->* <**bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close" scope="singleton"**>  <**property name="driverClassName" value="${driver}"**/>  <**property name="url" value="${url}"**/>  <**property name="username" value="${user}"**/>  <**property name="password" value="${password}"**/>   <**property name="initialSize" value="${initialSize}"**/> *<!-- 初始化连接时,第一次就要创建的连接个数 -->* <**property name="maxActive" value="${maxActive}"**/> *<!-- 同时连接的最大连接数 -->* <**property name="maxIdle" value="${maxIdle}"**/> *<!-- 最大空闲连接数 -->* <**property name="minIdle" value="${minIdle}"**/> *<!-- 最小,低于该数值连接池将会创建新的连接 -->* <**property name="maxWait" value="${maxWait}"**/> *<!-- 定义最大等待事件 -->* <**property name="removeAbandoned" value="${removeAbandoned}"**/> *<!-- 是否开启无用连接回收的机制 -->* <**property name="removeAbandonedTimeout" value="${removeAbandonedTimeout}"**/>*<!-- 超过配置的时间后回收无用的连接 -->  <!--sql心跳 -->* <**property name= "testWhileIdle" value="true"**/> *<!-- 开启evict的定时效验 -->* <**property name= "timeBetweenEvictionRunsMillis" value="60000"**/> *<!-- 定义Evict的时间间隔,单位毫秒 -->* <**property name= "testOnBorrow" value="false"**/> *<!-- -->* <**property name= "testOnReturn" value="false"**/>  <**property name= "validationQuery" value="select 1"**/> *<!-- 定义效验sql语句,简单通信一下,测试效验是否有效 -->* <**property name= "numTestsPerEvictionRun" value="${maxActive}"**/>*<!-- 定义每次效验连接的数量 -->* </**bean**>  *<!--mybatis中用builder创建sqlSessionFactory，集成环境中用整合包中的SqlSessionFactoryBean代替sqlSessionFactory。 -->* <**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  *<!-- 引用数据源组件 -->* <**property name="dataSource" ref="dataSource"**></**property**>  *<!-- 引用mybatis配置文件中的配置-->* <**property name="configLocation" value="classpath:mybatis-config.xml"**></**property**> </**bean**>  *<!-- basePackage定义了扫描的基准包，扫描包之下所有接口，如果他们在sql映射文件中定义过，则自动生成dao实现的代理对象  生成的对象名为接口名称首字母小写,多个包名用逗号隔开 配置了数据源才能自动生成代理对象 -->* <**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  <**property name="basePackage" value="cn.travel.mapper"**></**property**> </**bean**> |

### 步骤

导入完jar和配置文件，那么直接写代码即可，记得扫描service包

### 重点

接口和xml的名字一定要一样

## 整合junti

|  |
| --- |
| <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>3.8.1</**version**> </**dependency**> <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>3.2.10.RELEASE</**version**> </**dependency**>  @ContextConfiguration(locations ={**"/applicationContext-mybatis.xml"**}) @RunWith(SpringJUnit4ClassRunner.**class**) **public class** test {  @Autowired  HotelMapper **hotelMapper**;  @Test  **public void** findList(){  System.***out***.println(**hotelMapper**.findList());  } } |

# Spring mvc

## mvc架构优点：

多视图共享一个模型大大提高代码重用性

mvc3个模块相互独立,松耦合框架

控制器提高了应用程序的灵活性和可配置性

有利于软件工程化管理

## mvc架构缺点：

原理复杂

增加了系统结构和实现的复杂性

视图对模型数据的低效率访问

并不适用于小型甚至中型项目,花费大量时间将mvc应用到规模不大的应用程序得不尝失

## 是什么：

Spring mvc 是spring框架中用于web开发的一个模块,是一个基于mvc设计模式的web开发框架。

采用了松耦合可插拔的组件结构,具有高度可配型，扩展性，灵活性，而且本身就是spring框架的一部分,性能方面有天生的优越性

## 能干吗：

本质上相当于servlet。

支持灵活的URL到页面控制器的映射；支持Restful风格。

非常容易与其他视图技术集成，如Velocity、FreeMarker等等

非常灵活的数据验证、格式化和数据绑定机制，能使用任何对象进行数据绑定，不必实现特定框架的API；

## 环境搭建

1. 导入jar包
2. 在web.xml配置dispatcherServlet，映射到 / 即此servlet需要捕获并处理该项目所有url请求

3、创建springmvc配置文件，写<mvc:annotation-driven/>标签，配置该标签会自动注册：

1、DefaultAnnotationHandlerMapping(处理器映射,不采用注解时默认使BeanNameUrl处理器即在spring中查找与url同名的bean) 加注解为该请求映射到标注@requestMapping注解的控制器和方法上

BeanNameUrl用法：<bean id="/demo" class="main.java.controller.UserController"/>然后路径/demo就会进入这个控制器，控制器需要实现AbstractController实现方法后在里面写。

2、AnnotaionMethodHandlerAdapter(处理器适配器)，是spring对@controller分发请求的必要支持

处理器映射：对requemapping和conteoller注解的支持，找出请求url和处理器的关系从而找出进入相应的处理器

处理器适配器：找到对应的处理器之后 就用adapter进行封装，在用统一的适配器接口调用处理器 会将请求信息以一定的方式进行转换并绑定到请求的入参中

1. 然后在加scan扫描control，bean才能生效，才能进行映射

5、在cotroller类方法上加@RequestMapping("/index") 注解来映射请求的url 路径直接写/index就进入这个方法了

6、cotroller类方法返回值通过视图解析器解析为实际的物理视图,对于InternalResourceViewResolver 视图解析器

通过prefix + returnVal + suffix 得到实际物理视图转发 基于jsp视图的解析一般会使用它

Spring mvc 配置文件的命名，必须和web.xml里dispatcherServlet指定的名称一致 一般命名为 名称-servlet。

## 运行原理：

1.Tomcat在启动时加载解析web.xml,找到spring mvc的前端总控制器DispatcherServlet,并且通过DispatcherServlet来加载相关的配置文件。

2.DispatcherServlet接收到客户端请求，找到对应HandlerMapping(一般都是用DefaultAnnotationHandlerMapping它是对requemapping和conteoller注解的支持)，根据映射规则，找到对应的处理器，找到对应的处理器之后 就用adapter进行封装，在用统一的适配器接口调用处理器 会将请求信息以一定的方式进行转换并绑定到请求的入参中。

3.调用相应处理器中的处理方法，处理该请求后，会返回一个值。

4.DispatcherServlet根据得到的值，找到一个合适的ViewResolver（视图解析器），根据视图解析器的配置，DispatcherServlet将要显示的数据传给对应的视图，最后显示给用户。

# Mybatis

## 是什么

MyBatis是一个数据持久层框架，在实体类和SQL语句之间建立映射关系，是一种半自动化的ORM实现

ORM为对象/关系映射：通过简单的xml或者注解进行映射,将实体类和sql语句建立联系,通过javaBean对象去操作表中的数据

数据持久化是将内存中的数据模型转换为存储模型,以及将存储模型转换为内存中的数据模型的统称

数据模型可以是任何数据结构或对象模型

存储模型可以是关系模型,xml，二进制流等

## 能干嘛

支持高级映射，存储过程和定制化sql，避免了jdbc代码和手动设置参数及获取结果集。

## 使用方式

1、需要把mybatis 和mysql和log4j jar包和jdbc log4j 配置文件导入resource

2、在resource创建核心配置文件mybatis-config.xml 核心配置文件 有先后顺序

3、在resource创建mapper包里面创建各sql映射文件 sql映射文件 都得添加dtd

4、在util里创建连接关闭sqlsession的方法

5、UserMapper mapper=sqlSession.getMapper(UserMapper.class); 直接用接口里方法即可会为接口自动创建一个代理对象然后代理对象执行接口里面的方法

SqlSessionFactoryBuilder().build(in) 参数可以是InputStream,reader,configuration

openSession()参数为false表示不自动提交 如果不自动 必须手动sqlsession.commit

每个线程都有自己的SqlSession实例,它不能被共享,也不是线程安全的,因此 最佳的作用域是request或者在方法体内,和connection一样每次用都获取新连接用完必须关闭

SqlSessionFactory一旦创建就会一直在整个应用存在，没有理由销毁，所以写到util static里创建一次那么这种存在于整个应用运行期间，并且同时只能存在 一个对象实例就是单例模式



## resultType：

查询标签得设置返回类型

增删改不用因为接口返回类型是boolean就返回true，int就返回1

两者都不用设置参数类型

如果为查询那么：

方法返回类型为实体类 resultType就写实体类 但必须保证只能查到一条数据

方法返回类型为list类型 resultType也写实体类 直接返回list

返回单条记录返回类型为Map<String,Object>,key就是字段名,值就是对应的值 resultType写map

返回多条记录方法上加@MapKey("author")指定key为哪个字段(表里叫啥这就写啥)

返回类型为Map<Integer,News> ,键是指定的key,值是一条记录 resultType写map

如果字段比较复杂使用resultMap

resultMap="findUserMap" 这个findUserMap对应的是resultMap标签里的id

标签里还有一个属性为type 值就为resultType那些规范

column=从数据库中查询的字段名 property=对应赋值给实体类的那个对象

## Collection一对多

### 直接查询

查询酒店hid和酒店hotelName赋值给id和name，连表查询属于酒店的特色，赋值给List<Feature> c，类型为**cn.travel.dto.Feature， 结果为1，北京酒店，c(17，115，116，117)**



|  |
| --- |
| <**resultMap id="selectMap" type="cn.travel.dto.HotelTest"**>  <**result property="id" column="hid"**></**result**>  <**result property="name" column="hotelName"**></**result**>  <**collection property="c" ofType="cn.travel.dto.Feature"**>  <**result property="feature" column="feature"**></**result**>  </**collection**> </**resultMap**>  <**select id="findHotel" resultMap="selectMap"**>  select h.**id** hid,h.**hotelName**,f.**featureId** feature from itrip\_hotel h join itrip\_hotel\_feature f on  h.**id**=f.**hotelId** WHERE **hotelId**=1 </**select**> |

### 调用map

两个表中间还有中间表或者一起查比较麻烦可以采用map方式，本来细框这点就可以，又多了粗框内容才能查出来



|  |
| --- |
| <**resultMap id="selectMap" type="cn.travel.dto.HotelTest"**>  <**result property="id" column="hid"**></**result**>  <**result property="name" column="hotelName"**></**result**>  <**collection property="c" column="hid" select="findFeature"**>  </**collection**> </**resultMap**>  <**select id="findHotel" resultMap="selectMap"**>  select h.**id** hid,h.**hotelName** from itrip\_hotel h WHERE **id**=1 </**select**>  *<!--这个实在featureMapper里写-->* <**select id="findFeature" resultType="cn.travel.dto.Feature"**>  select **featureId** feature from itrip\_hotel\_feature f where **hotelId** = #{hid} </**select**> |

## Assocaiation一对一

|  |
| --- |
| <**resultMap id="selectMap" type="cn.travel.dto.HotelTest"**>  <**result property="id" column="hid"**></**result**>  <**result property="name" column="hotelName"**></**result**>  <**association property="c" javaType="cn.travel.dto.Feature"**>  <**result property="feature" column="feature"**></**result**>  </**association**> </**resultMap**>  <**select id="findHotel" resultMap="selectMap"**>  select h.**id** hid,h.**hotelName**,f.**featureId** feature from itrip\_hotel h join itrip\_hotel\_feature f on h.**id**=f.**hotelId** WHERE **hotelId**=1 </**select**> |

Collection和assocaiation都一样如果连在一起难解决就用组合map的方式，唯一区别就是普通方式coll为oftype，ass为javatype

|  |
| --- |
| <**resultMap id="selectMap" type="cn.travel.dto.HotelTest"**>  <**result property="id" column="hid"**></**result**>  <**result property="name" column="hotelName"**></**result**>  <**association property="c" column="hid" select="findFeature"**>  </**association**> </**resultMap**>  <**select id="findHotel" resultMap="selectMap"**>  select h.**id** hid,h.**hotelName** from itrip\_hotel h WHERE **id**=6 </**select**> *<!--这个实在featureMapper里写-->* <**select id="findFeature" resultType="cn.travel.dto.Feature"**>  select **featureId** feature from itrip\_hotel\_feature f where **hotelId** = #{hid} </**select**> |

默认情况下resultMap会自动匹配所有，但是加了内部嵌套就得改变setting里的value为full才能自动匹配，所以里面就写一些复杂的匹配即可，简单的让他自动匹配

<**setting name="autoMappingBehavior" value="FULL"**/>

## paramType：

参数为List类型 下面用list[0]方式来获取具体值 也可以foreach动态sql

单个普通参数 用什么名字接收都可以

单个实体类参数 对应实体类属性就可以了

多个参数最好封装成实体类，不行的话就：

多个参数会被封装成一个map #{}就是从map中获取执行key的值 key为:param1,paramN,也可以在参数前面加上 @Param("id")int id,@Param("author")String author key就会使用param注解使用的值

@param修饰的实体类也是 比如author是实体类就author.id就行

如果参数为实体类和其他 #{param1.lastName} #{param2}

实体类里有个实体类可以直接 #{param1.user.lastName} 都可以用注解

参数为map直接对应key取值

${} 表示拼接sql串,将接收到的参数的内容不加任何修饰拼接在sql中

like '%${value}%' 模糊查询 参数 为普通类型参数 而且不用注解 只能写value，而不是对应的字段名

用#实现模糊查询 like concat ('%',#{id},'%')

插入主表之后要插入子表 所以要获取主表的id 只需在插入标签里嵌套一个：

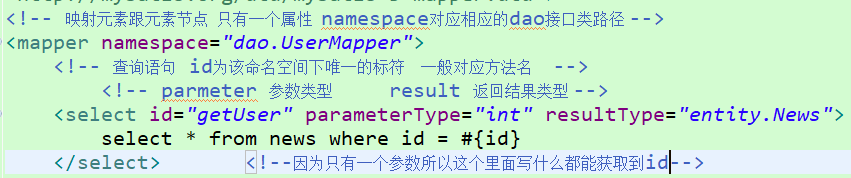
将查询到主键值赋值为实体类中哪个属性 之后查 id字段在java中的类型

<selectKey keyProperty="id" order="AFTER" resultType="java.lang.String">

select last\_insert\_id()

</selectKey>

然后输出指定的那个对象的属性就可以了 boolean s=mapper.insertUser(b); System.out.println(b.getId());



## 单词

relation 关系

地图/映射 map

config 配置

Factory 工厂

Builder 生成器

column 列

property 性能/属性

association 联合/协会、

Collection 收藏

setting 设置

Behavior 行为

full 完全的

Aliases 别名

environment 环境

## 核心配置文件



configuration

properties

settings 设置mybatis中的一些行为

<setting name="autoMappingBehavior" value="NONE"/> 在config设置这个不会把没设置映射的也获取到

FULL 自动匹配 PARTIAL(默认) 自动匹配有内部嵌套除外

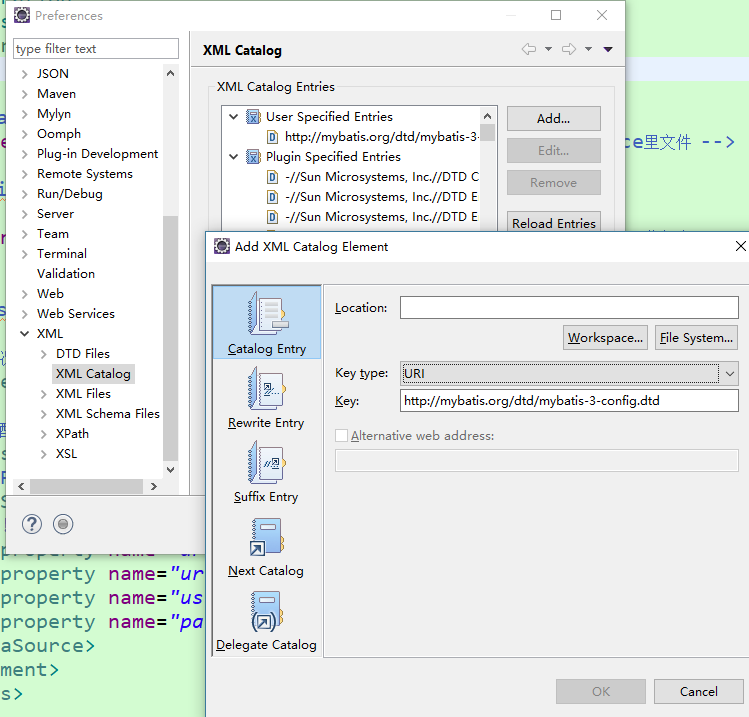
typeAliases 配置类型别名

<typeAlias alias="news" type="entity.News"/>

<**package name="cn.travel.entity"**/> 设置后resulttype和resultMap的type属性的时候不用带name里的前缀

environments 配置多套运行环境 如开发，测试，生产

添加dtd方法：打开如图位置 file system找到路径然后 key type默认的不用改 key写public 里面的内容

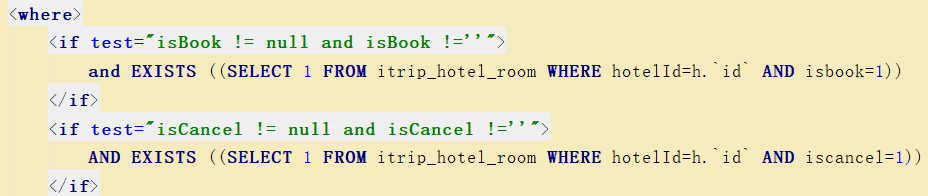


## 动态sql

如果一个参数为null的话就不把它当作条件在select标签里嵌套

<if test="param1 != null and param1 != ''"> and id=#{param1}

if+where 如果where标签内有有返回值就加个where，首个if以and或or开头会自动剔除



if+trim prefix 如有内容就加里面的东西 prefixOverrides 如首个内容进行指定的忽略(里面写and就会把第一个满足的if里面的and忽略掉)

suffix 在trim包含的内容上加后缀 suffixOverrides 对trim里包含的最后满足if的最后选择忽略些东西

if+set 把set代码都写到set标签里 会把末位if里面的内容逗号取消掉

参数为数组类型 <foreach collection="array" item="rid" open="(" separator="," close=")">

item 每个元素被迭代时的别名

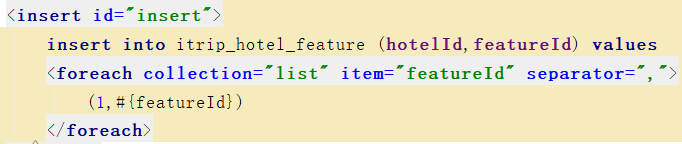
open 表示该语句用什么开始

separator 每次循环用什么分隔

close 用什么结束

collection 数组array 集合 list 若为多参数则封装成一个map处理(这样就是用键名来选择取哪个参数) 或 写成param1或写注解(对应的就为注解名字)

增加用foreach这样即可，剩下的查删改就用标红的那个，each标签里面写#{xx}即可



choose <when test="param1 !=null"> otherwise 嵌套在choose里满足when执行里面内容并跳出 都不满足执行otherwise

大于小于号如果报错写 &gt; &lt;

## 单词

prefix 前缀

suffix 后缀

trim 修剪/整理

item 一则/项目

separator 分离器/分隔符

choose 选择

when 当.....时

otherwise 否则

action 行动

stack 堆

trace 痕迹

other 其他的

# Ssm整合mybatis分页

<**dependency**>  
 <**groupId**>com.github.pagehelper</**groupId**>  
 <**artifactId**>pagehelper</**artifactId**>  
 <**version**>4.1.6</**version**>  
</**dependency**>

把pom文件改成这个就行了

# 重点：

Boot可以直接页面html suffix设置.html然后返回，但是ssm不可以，因为静态资源也会去访问dispatcherServlet会导致失败，解决：在web.xml加上这个，别的静态资源在用<mvc:resources location="/statics/" mapping="/statics/\*\*"/>处理

|  |
| --- |
| <**servlet-mapping**>  <**servlet-name**>default</**servlet-name**>  <**url-pattern**>\*.html</**url-pattern**> </**servlet-mapping**> |

如果出现个找不到springmvc-servlet错误可以在前面classpath后加一个\*

<**param-value**>classpath\*:springmvc-servlet.xml</**param-value**>

一般如果contr或service没有注册到容器那么有可能是必须在web.xml加上

|  |
| --- |
| *<!--配置环境参数,指定spring配置文件所在目录 -->* <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:applicationContext-\*.xml</**param-value**> </**context-param**>  *<!--配置spring的ContextLoaderListener监听器，初始化spring容器-->* <**listener**>  <**listener-class**>  org.springframework.web.context.ContextLoaderListener  </**listener-class**> </**listener**> |

线程不安全的都得写到方法体内

右键已经导入好的mybatis包选择properties ----- java source----external file把原码jar导入,然后进入原码也许会再让选择一下路径即可 原码

核心配置文件必须按顺序排

openSession方法可以传一个参数boolean 用来是否自动提交

两个不同mapper文件子元素的id可以相同

每个类在指定多参构造一定要在写一个无参的，避免很多错误

创建项目的时候 next\*2 有一个自动创web.xml

在本项目中监听器初始化spring(获取已经注入进去的实例需要启动spring容器) ，是为了初始化公共的配置文件(applicationContext-mybatis)和一些单独config，而controll的扫描要放在springmvc Config

classpathapplication初始化了之后。在getBean之前就已经有值了，springmvc无法通过main方法这样初始化，所以就监听器

${pageContext.request.contextPath} 项目名

json中文乱码 406错误 表示客户端浏览器不接收所请求页面的MIME类型。resultMapping的value属性去掉html,因为由此后缀,mvc会以HTML格式来显示响应信息

修改的话就给controll传过去id,方法里查数据返回模型,之后点击修改跳到修改save方法,根据传过来的模型修改成功跳页面

project clean cleanProjectsSelectedBelow 可以重新编译

想在control加切面必须把切面自动代理写到mvc配置文件中

<%pageContext.setAttribute("path",request.getContextPath());%> 最上面加这个 下面就可以直接${path}

<input type="hidden" id="path" value="${path}"> 用于ajax的时候

做分页的话dao和service都用start,end为参数,在control用currPage把start计算好传入service

如果页面上用include引入头和尾了,那么不仅head需要改编码格式那一句话,引入head的页面也需要加上最顶部改编码格式那句话

拦截器要返回inf里面的页面直接跳到control来返回呀,之外的就直接返回就可以了

登录方法写if判断如果name为空则查所有，不为null查该账户，不用写两个方法了

右键项目build path------libraries-----add libraries-----Junit----Junit4

controller方法有参数的话会直接传到页面。 待考察

在idea里普通web项目运行出servlet啥错可以去添加依赖那，添加library把tomcat加进去

运行tomcat的话 ，得把项目的war加到tomcat依赖里面

在idea里如果没有web文件夹可以在Strictire里modules+一个web并改一下上方webxml路径和web路径

如果有关于lombok找不到符号错误可以去setting--Compiler--Annotion Processors----enable annotation 打钩

configuration annotation processor not found in classpath 类路径中没有配置注释处理器

mybatis.mapper-locations=classpath:mybatis/mapper/\*.xml 这个classpath后都得用/

公共的pom放到总共成里

页面将数据传到后台，是以字符串的形式。所以时间格式会出错。加上此注解，后台可解析时间格式的字符串@DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss" )

SELECT \* FROM company WHERE id = NULL

UPDATE company SET NAME = NULL WHERE id=1 修改可以不做null的判断，等于null那就改成null，查询不行等于null查询等于null的查不到

mybatis里if动态sql如果打小与错误可以写成 &gt; &lt;

There is no getter for property named 'cid' in 'class java.lang.String' 在class java.lang.String中没有属性'cid'的getter