目录

[Spring 3](#_Toc118794539)

[简介: 3](#_Toc118794540)

[Spring 核心部分IOC,AOP 3](#_Toc118794541)

[IOC控制反转 3](#_Toc118794542)

[IOC原理 3](#_Toc118794543)

[IOC操作Bean管理的两个操作 4](#_Toc118794544)

[Bean管理操作的两种方式 5](#_Toc118794545)

[AOP面向切面编程 13](#_Toc118794546)

[创建动态代理（AOP底层原理） 14](#_Toc118794547)

[AOP术语 15](#_Toc118794548)

[AOP准备操作 15](#_Toc118794549)

[AOP的操作，基于注解方式 16](#_Toc118794550)

[AOP操作，基于配置文件方式 17](#_Toc118794551)

[JdbcTemplate 18](#_Toc118794552)

[基本概念配置 18](#_Toc118794553)

[声明式事务 19](#_Toc118794554)

[SpringMVC 19](#_Toc118794555)

[快速入门 20](#_Toc118794556)

[快速入门步骤 23](#_Toc118794557)

[快速入门注解开发 23](#_Toc118794558)

[注解详解 24](#_Toc118794559)

[请求映射@RequestMapping 24](#_Toc118794560)

[RestFul风格 26](#_Toc118794561)

[IDEA环境问题 27](#_Toc118794562)

[SpringMVC流程原理 28](#_Toc118794563)

[页面跳转方式 30](#_Toc118794564)

[参数传递和显示 31](#_Toc118794565)

[参数传递 31](#_Toc118794566)

[参数显示 32](#_Toc118794567)

[中文乱码问题 32](#_Toc118794568)

[JSON 33](#_Toc118794569)

[前端操作JSON对象 34](#_Toc118794570)

[后端操作JSON数据 34](#_Toc118794571)

[Mybatis 37](#_Toc118794572)

[入门案例 37](#_Toc118794573)

[CRUD操作 39](#_Toc118794574)

[select,insert,delete,update 40](#_Toc118794575)

[万能的Map 41](#_Toc118794576)

[模糊查询 41](#_Toc118794577)

[配置解析\*\*\* 41](#_Toc118794578)

[environment环境 42](#_Toc118794579)

[propertie属性 42](#_Toc118794580)

[typealise起别名 42](#_Toc118794581)

[setting设置 43](#_Toc118794582)

[映射器Mapper 43](#_Toc118794583)

[声明周期和作用域 45](#_Toc118794584)

[ResultMap\*\*\* 46](#_Toc118794585)

[日志 47](#_Toc118794586)

[分页 48](#_Toc118794587)

[使用注解开发 49](#_Toc118794588)

[Mybatis执行流程\*\*\* 50](#_Toc118794589)

[Lombok 51](#_Toc118794590)

[关联查询 52](#_Toc118794591)

[多对一关系 52](#_Toc118794592)

[一对多关系 54](#_Toc118794593)

[动态SQL 56](#_Toc118794594)

[Mybatis缓存 57](#_Toc118794595)

[一级缓存 57](#_Toc118794596)

[二级缓存 58](#_Toc118794597)

[自定义缓存Encache 59](#_Toc118794598)

[SSM整合 60](#_Toc118794599)

# Spring

## 简介:

一种轻量级开源的Java EE框架

Spring框架的目的是解决企业级应用开发的复杂性

特点：降低代码耦合程度，简化开发（java反射，代理实现降耦）

支持AOP编程

方便测试

可以整合各种其它框架

## Spring 核心部分IOC,AOP

* IOC:控制反转，把创建对象的过程交给Spring进行管理，解耦！
* AOP:面向切面编程，即不修改源代码进行功能的增强

IOC底层原理:

1. xml解析
2. 工厂模式(代理)
3. 反射

## IOC控制反转

### IOC原理

IOC过程:

第一步通过xml配置文件配置创建的对象

<bean id=”user”,class=”com.study.sdm.UserDao”></bean>

第二步有service类和Dao类，创建工厂类

class UserFactory{

public static UserDao getDao(){

String classValue = class属性值; //xml解析

Class clazz = Class.forName(classValue); //通过反射创建对象

return (UserDao)clazz.newInstance();

}

}

IOC接口：

1. IOC思想基于IOC容器完成，IOC容器底层就是对象工厂
2. Spring提供IOC容器实现两种方式(两个接口):

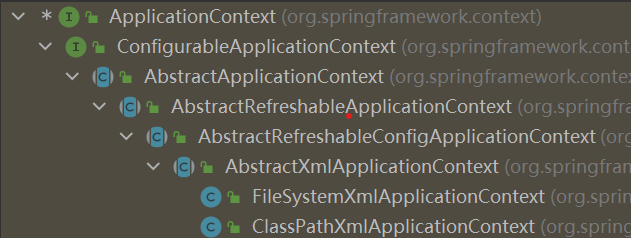
* BeanFactory：IOC容器基本使用，是Spring内部的使用接口，不提供开发人员使用。

\*加载配置文件的时候不会创建对象，在获取使用对象的时候才去创建对象。

* ApplicationContext：BeanFactory接口的子接口，提供了更多更强大的子接口，面向开发人员使用

\*加载配置文件的时候就会把配置文件里的对象创建

ApplicationContext接口有实现类(最后两个)



### IOC操作Bean管理的两个操作

1. Spring创建对象
2. Spring注入属性

### Bean管理操作的两种方式

1. 基于xml配置文件方式
2. 基于注解方式

#### 基于xml方式创建对象和注入属性

创建对象：

<bean id=”user”,class=”com.study.sdm.UserDao”></bean>

<bean>标签的基本常用属性

* id：唯一标识，用于获取对象
* class：类的全路径，包名+类名
* name：属性名称。

DI依赖注入:

set方法注入：

第一步创建一个类，完成这个类属性的set方法:

private String Bname;  
private String Bauthor;  
public void setBname(String bname){  
 this.Bname = bname;  
}  
public void setBauthor(String bauthor){  
 this.Bauthor = bauthor;  
}

第二步在xml配置文件注入属性值，用到<bean>的子标签<property>

<bean id="book" class="com.sdm.spring5.Book">  
 <property name="Name" value="周易" />  
 <property name="Author" value="姬昌" />  
</bean>

有参数构造器注入

目标类有参构造方法

xml文件配置参数

<bean id = "orders" class="com.sdm.spring5.Order">  
 <constructor-arg name="oid" value="1"></constructor-arg>  
 <constructor-arg name="oname" value="abc"></constructor-arg>  
</bean>

xml注入其它类型数据，注入空值

<property name="author">  
 <null></null>  
</property>

##### 注入值包含特殊符号

如 <<周易>>

进行转义:

&lt; 这是小于号

&gt; 这是大于号

![CDATA[转意内容]]

<![CDATA[<<周易>>]]>

##### 注入外部的bean

1. 创建service类和dao类
2. 在service中调用dao里面的方法

##### 注入属性内部bean和级联赋值

一对多关系：

部门和员工

一个部门有多个员工，一个员工属于一个部门

在实体类中表示一对多关系，可以通过在员工属性中添加部门来表示。

1. 注入内部bean

<bean id="emp" class="com.sdm.spring5.bean.Emp">  
 <property name="name" value="abc" />  
 <property name="id" value="100" />  
 <property name="dept">  
 <bean id="dept" class="com.sdm.spring5.bean.Dept">  
 <property name="id" value="10" />  
 <property name="name" value="ab" />  
 </bean>  
 </property>  
</bean>

1. 级联赋值

<bean id="emp" class="com.sdm.spring5.bean.Emp">  
 <property name="name" value="abc" />  
 <property name="id" value="100" />  
 <property name="dept" ref="dept">  
 </property>  
</bean>  
<bean id="dept" class="com.sdm.spring5.bean.Dept">  
 <property name="name" value="abc">  
 </property>  
 <property name="id" value="12" />  
</bean>

##### xml方式注入集合属性

1. 注入数组
2. 注入list集合
3. 注入map集合
4. <bean id="stu" class="com.sdm.spring5.collectiontype.Stu">  
    <property name="course"><!--数组值注入-->  
    <array>  
    <value>123</value>  
    <value>abc</value>  
    <value>数据库</value>  
    </array>  
    </property>  
    <property name="list"><!--list值注入-->  
    <list>  
    <value>ABC</value>  
    <value>java课程</value>  
    </list>  
    </property>  
    <property name="map"><!--map值注入-->  
    <map>  
    <entry key="1" value="a"></entry>  
    <entry key="2" value="b"></entry>  
    </map>  
    </property>  
   </bean>

##### 在集合中注入对象

<property name="courseList">  
 <list>  
 <ref bean="course1"></ref>  
 <ref bean="course2"></ref>  
 </list>  
</property>

##### 把集合注入部分提取出来

注意：需要添加配置文件命名空间

<util:list id = "booklist">  
 <value>周易</value>  
 <value>道德经</value>  
 <value>论语</value>  
</util:list>  
<bean id="book" class="com.sdm.spring5.collectiontype.Book">  
 <property name="booklist" ref="booklist"></property>  
</bean>

##### IOC操作bean管理

spring有两种bean,一种是普通bean，一种是工厂bean

普通bean类型根据配置文件确定

工厂bean在配置文件定义bean类型可以和返会类型不一样。

第一步创建类，这个类作为工厂bean,实现接口FactoryBean,并在配置文件中配置工厂bean

第二步实现接口里面的方法，在方法中返会bean

public class MyFactory implements FactoryBean<Book> {  
  
 @Override  
 public Book getObject() throws Exception {//返会你指定的类型bean  
 return new Book("周易");  
 }

<bean id="mf" class="com.sdm.spring5.factory.MyFactory">  
</bean>

##### Spring单实例和多实例

单实例每次都是同一个对象，多实例每次都是不同的对象

spring默认是单实例，在配置文件里面，bean标签里有scope属性

scope属性值，第一个默认值singleton默认单实例，第二个prototype表示多实例

二者区别：

singleton,在加载配置文件的时候就会创建单实例对象

prototype在调用getbean方法是创建实例对象！

##### bean的生命周期

生命周期:从对象创建到对象销毁的过程，过程步骤如下

1. 通过无参构造器创建bean实例
2. 为bean属性设置值和对其他bean的引用(调用set方法)
3. 调用bean的初始化的方法，xml配置文件中bean标签有一个属性init-method
4. bean可以使用,获取到了bean的时候！
5. 容器关闭，调用bean的销毁，需要自己配置销毁方法。xml配置文件中bean标签有一个属性destroy-method，然后调用ApplicationContext的子类的close方法关闭。
6. ((ClassPathXmlApplicationContext)applicationContext).close();

##### xml的自动装配

根据指定装配规则，属性名称或者属性类型，Spring自动将匹配的属性值进行注入

<!--实现自动装配  
bean标签属性autowire,配置自动装配  
autowire有两个值，byName,byType。根据属性名称和根据属性类型-->  
<bean id = "emp" class="com.sdm.spring5.auto.Emp" autowire="byName">

#### 基于注解方式操作bean管理

目的：简化xml配置

@Component:注解通用组件类，一般来说该注解注入到spring容器中的bean都是单例。

@Service:注解业务逻辑层组件类

@Controller：注解控制器层组件类

@Repository：注解DAO层，数据访问层

这四个注解都可以用来创建bean

基于注解方式实现对象创建：

* 第一步引入依赖：

spring-aop-RELEASE

* 第二步开启组件扫描：扫描上层包也可以完全扫描到子包

<context:component-scan base-package="com.sdm.spring5.service"></context:component-scan>

* 第三步创建类，在类上面添加注解

开启组件扫描细节：

实例1：不使用默认过滤器，则配置上层包时，子包不会自动扫描。而是扫描带有Service注解的

<context:component-scan base-package="com.sdm.spring5.service" use-default-filters="false">  
 <context:include-filter type="annotation" expression="com.sdm.spring5.Service"/>  
</context:component-scan>

实例2：和实例1相反，使用默认过滤器，扫描一切包和子包的内容。带有Service注解的不扫描，其他都扫描

<context:component-scan base-package="com.sdm.spring5.service">  
 <context:exclude-filter type="annotation" expression="com.sdm.spring5.Service"/>  
</context:component-scan>

##### 基于注解方式实现属性注入

@Autowired:根据属性类型自动注入(如果一个属性类型有多种实现比如一个接口，有多个实现类，则根据变量名注入)所以通常配合@Qualifier进行精准注入！

@Qualifier：根据属性名称进行注入，不能独立使用配合Autowired进行精准注入。

@Resource：可以根据类型注入，也可以根据名称注入。配置该注解不带name参数默认是按照数据类型注入，带name属性则根据name名称精准注入

@Value:注入普通类型,在参数括内写入基本数据类型的值

##### 纯注解开发

1. 创建配置类
2. @Configuration//作为配置类，代替xml配置文件  
   @ComponentScan(basePackages = "com.sdm.spring5")  
   public class SpringConfiger {  
      
   }
3. 编写测试类

注意：纯注解优先级比配置文件xml高

#### xml引入外部配置文件例如properties文件

1. 直接配置数据库信息

配置连接池(需要引入相关jar包和导入相关命名空间)

<!--引入外部属性文件-->  
<context:property-placeholder location="classpath\*:jdbc.properties"/>

## AOP面向切面编程

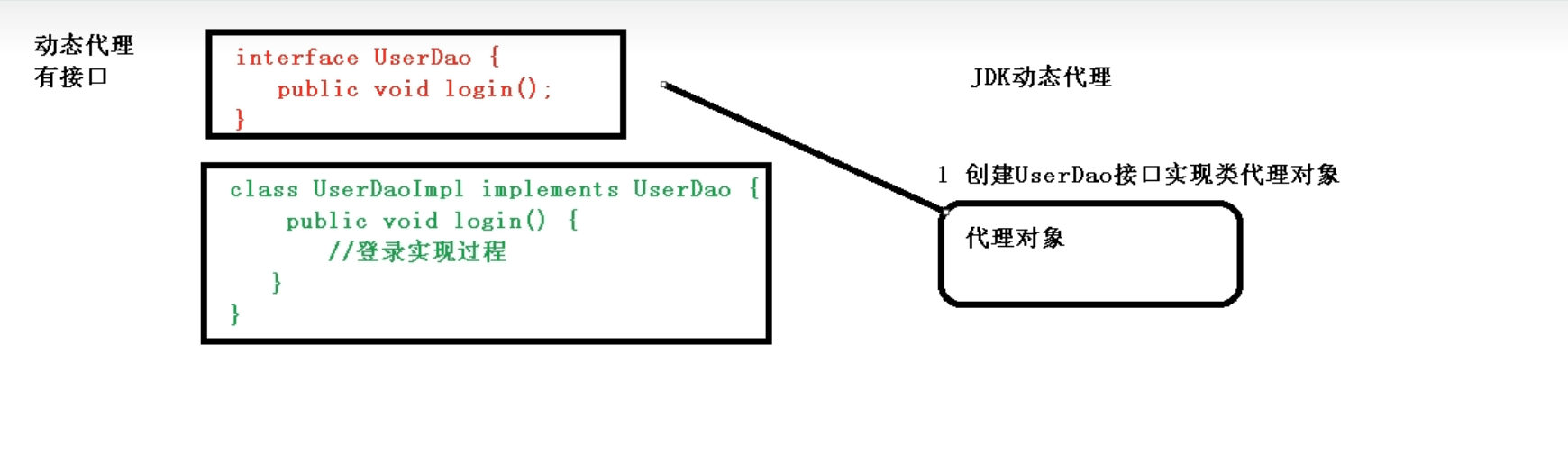
作用：利用AOP将业务逻辑各个部分分离，使业务逻辑的各个部分耦合度降低。

即不改变源代码的情况下，对源代码的功能进行增强。

原理(动态代理)：

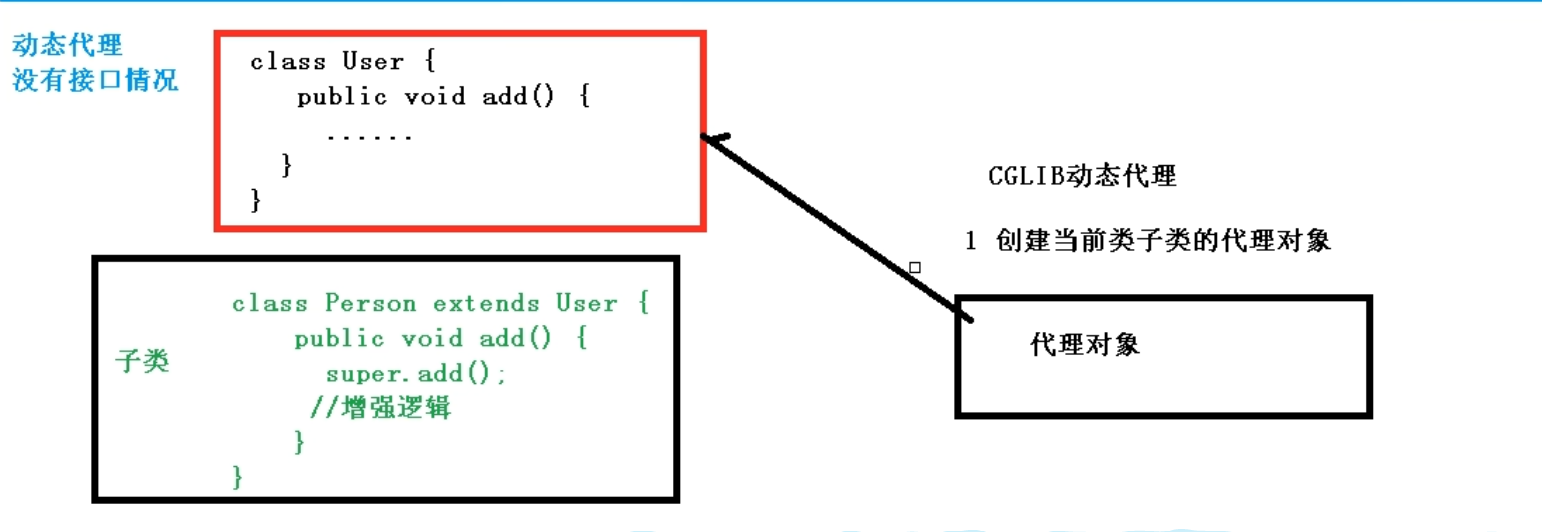
1. 有接口情况，使用JDK动态代理

通过创建接口的实现类代理对象，增强类的方法



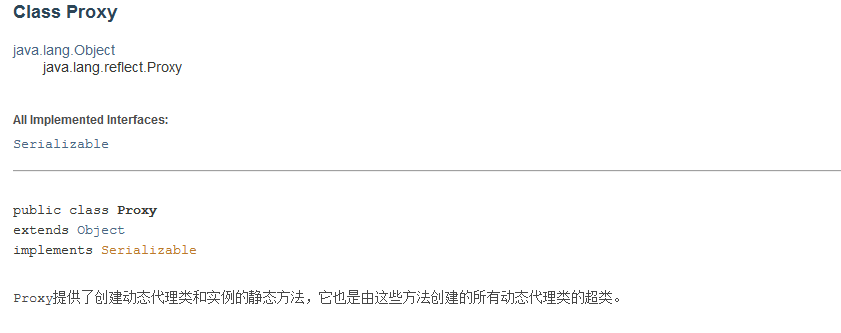
1. 没有接口，使用CGLIB动态代理

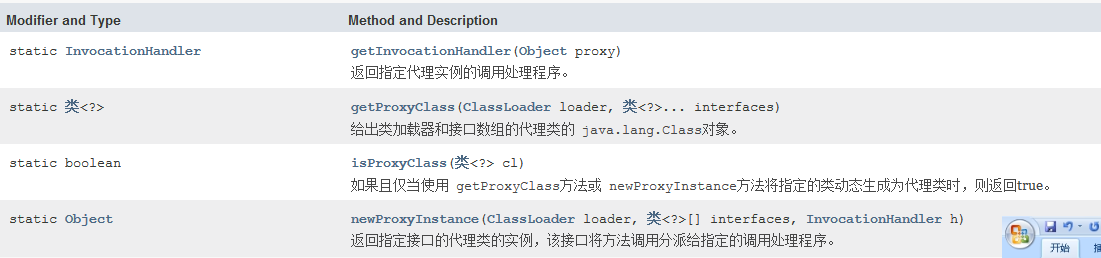
创建当前类的子类代理对象，增强类的方法



### 创建动态代理（AOP底层原理）

JDK提供的创建动态代理的类以及相应的方法如下：





newProxyInstance方法可以返回指定接口的代理类实例

三个参数分别是:

1. 类加载器
2. 指定接口(支持多个接口)
3. 实现这个接口InvocatonHandler,创建代理对象，写增强方法

### AOP术语

1. 连接点

类里面哪些方法可以被增强，这些方法称为连接点

1. 切入点

实际被真正增强的方法称为切入点

1. 通知(增强)

实际增强的逻辑部分称之为通知

通知有多种类型，分为

* 前置通知
* 后置通知
* 环绕通知
* 异常通知
* 最终通知

1. 切面

是一个动作，把通知应用到切入点的过程。

### AOP准备操作

spring框架中基于AspectJ包实现AOP操作

AspectJ:不是spring组成部分，它是一个独立的AOP框架，一般配合spring框架使用

基于AspectJ实现AOP操作

1. 基于xml配置文件实现
2. 基于注解方式实现

切入点表达式

作用：知道对哪个类里面的哪个方法进行增强

语法结构：

execution([权限修饰符][返回类型][类全路径][方法名称]([参数列表]))

例如：对com.study.sdm.dao.BookDao类里面的add方法进行增强

execution(\* com.study.sdm.dao.BookDao.add(..)) //\*在其位置代表一切

### AOP的操作，基于注解方式

spring配置文件中开启注解扫描

<context:component-scan base-package="com.sdm.spring5.aopTest"></context:component-scan>

使用注解创建User和UserProxy对象

@Component  
public class User {  
 public void add(){  
 System.*out*.println("user add method");  
 }  
}

在增强类上面添加注解@Aspect

//增强的类  
@Component  
@Aspect//生成代理对象  
public class UserProxy {  
 public void before(){  
 System.*out*.println("before..............");  
 }  
}

在spring配置文件中开启生成代理对象

<aop:aspectj-autoproxy></aop:aspectj-autoproxy><!--开启aspectj,生成代理对象-->

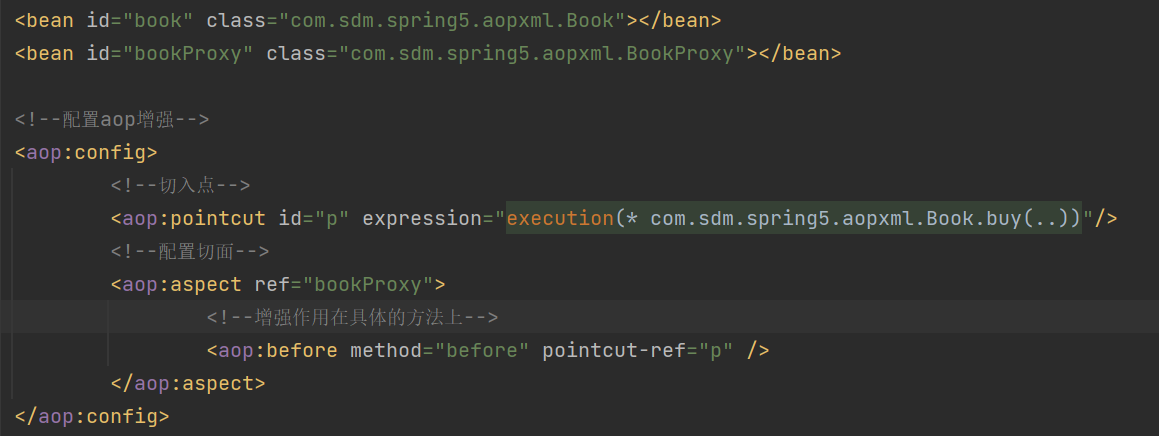
配置不同通知类型

//前置通知  
@Before(value = "execution(\* com.sdm.spring5.aopTest.User.add(..))")  
public void before(){  
 System.*out*.println("before..............");  
}

### AOP操作，基于配置文件方式

创建两个类，book和bookProxy类，前者是目标类，后者是增强类(代理)

xml配置文件如下：



## JdbcTemplate

### 基本概念配置

Spring框架对原始JDBC api进行封装，使用JdbcTemplate方便实现对数据库的操作.

基本准备：

引入jar包

在spring配置文件中配置数据库连接池

配置JdbcTemplate对象，注入DataSource,set注入

注意：

druid.properties配置文件中的key，例如username,使用了spring，配置文件中的username不能叫username，spring会默认username是你的计算机名。

解决办法：

在key前加上jdbc.即可

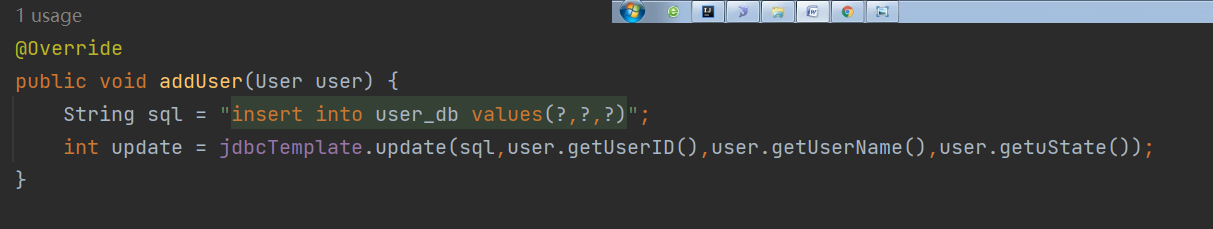
在key前加上jdbc.即可

在key前加上jdbc.即可

在key前加上jdbc.即可



利用JdbcTemplate的update方法完成数据的增删改。



### 声明式事务

回顾事务：

把一组操作作为一个不可分割点，要么全部操作成功，要么全部操作失败

事务涉及到数据的一致性

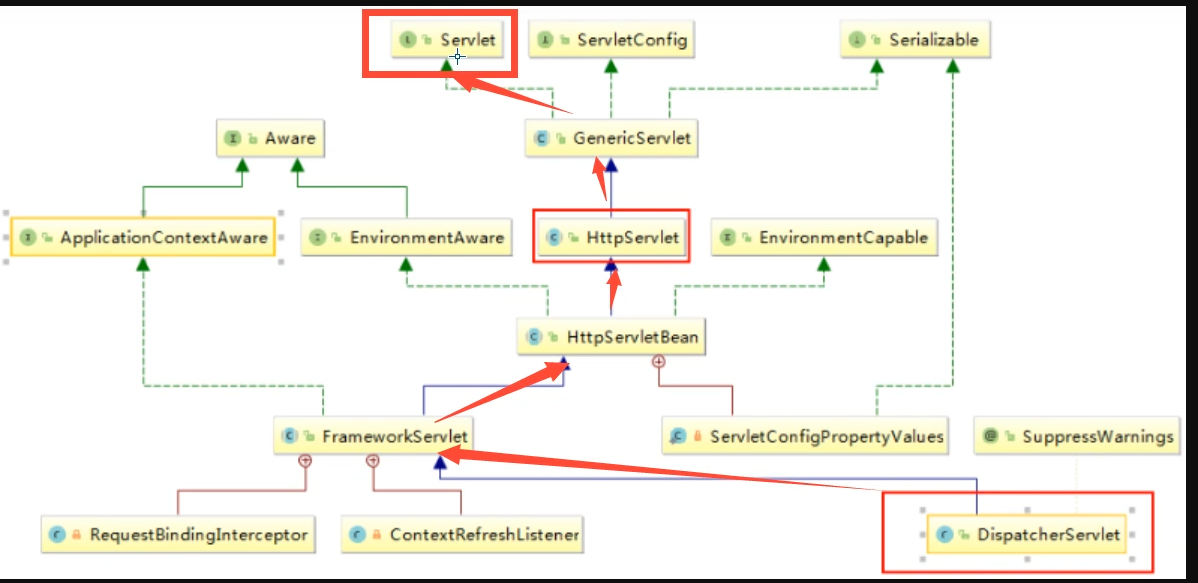
确保数据的完整性和一致性

事务的ACID原则：

* + 原子性
  + 一致性
  + 隔离性：多个业务可能操作同一个资源，方式数据损坏
  + 持久性：事务一旦提交，无论系统发生什么问题，结果都不会被影响，被持久化写入存储器中。

# SpringMVC

Spring的web框架围绕着DispatcherServlet设计，DispatcherServlet的作用就是将请求分发到不同的处理器！建议使用注解开发！



如图，dispatcherServlet本质还是servlet

特点：

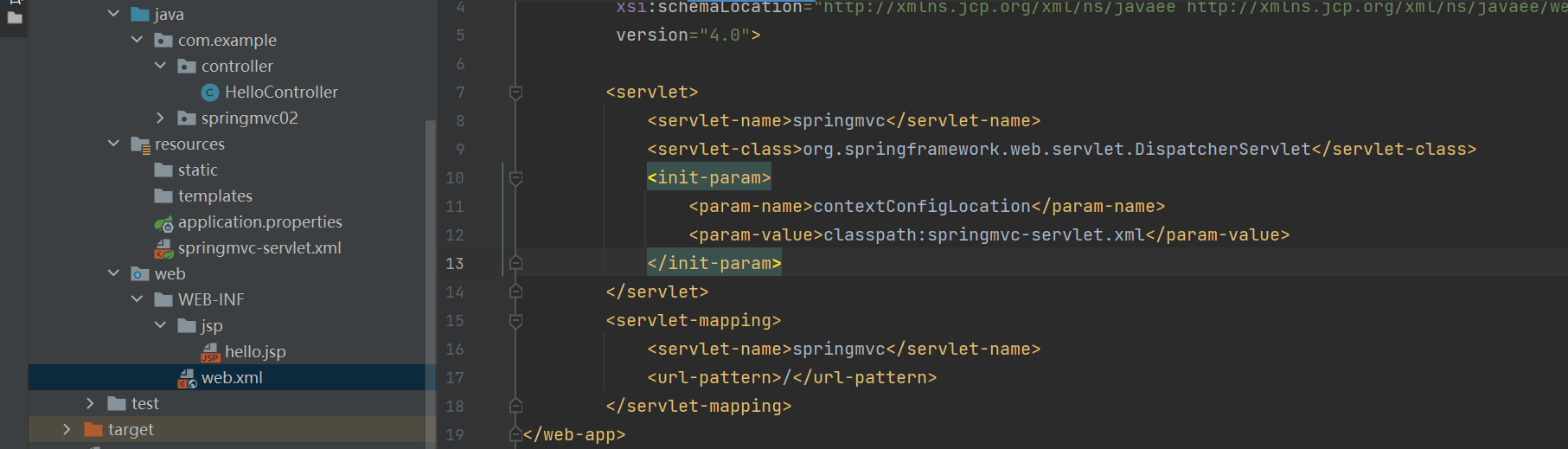
1. 轻量级，简单易学
2. 高效，基于请求相应的MVC框架
3. 与spring兼容性好，无缝结合
4. 约定大于配置
5. 功能强大，RESTFUL,数据验证，格式化，本地化，主题
6. 简介灵活

## 快速入门

新建一个maven项目，引入springmvc依赖！

<!--springmvc依赖-->  
<dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.2.1.RELEASE</version>  
</dependency>  
<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>servlet-api</artifactId>  
 <version>2.5</version>  
</dependency>  
<dependency>  
 <groupId>javax.servlet.jsp</groupId>  
 <artifactId>jsp-api</artifactId>  
 <version>2.2</version>  
</dependency>  
<dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 <version>1.2</version>  
</dependency>

项目引入web支持，web.xml配置dispatcherServlet,同时引入springmvc-servlet.xml配置！



web.xml中前端控制器映射地址：

<!--映射地址/ 和 /\*  
 / :匹配所有的请求，不会去匹配jsp页面  
 /\* :匹配所有的请求，会去匹配jsp页面  
 比如用户发起的请求中包含了xxx.jsp,该请求被前端控制器拦截，而前端控制器映射地址是/\*,  
 这样请求就会再次过一遍前端控制器，通过视图解析器的拼接，导致请求中的xxx.jsp变成xxx.jsp.jsp....  
 无限死循环，通常只写/就OK  
-->

springmvc-servlet.xml文件配置如下：

//请求映射器

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>  
//适配处理器  
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>

//视图解析器  
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/jsp/"/>  
 <property name="suffix" value=".jsp"/>  
</bean>

//控制器bean  
<bean id="/hello" class="com.example.controller.HelloController"/>

编写控制器：

//需要导入Controller接口，org.springframework.web.servlet.mvc.Controller;  
public class HelloController implements Controller {  
  
 @Override  
 public ModelAndView handleRequest(HttpServletRequest httpServletRequest,  
 HttpServletResponse httpServletResponse) throws Exception {

//模型和视图  
 ModelAndView mav = new ModelAndView();

//添加数据  
 mav.addObject("msg","helloSpringMVC");

//页面跳转  
 mav.setViewName("hello");  
 return mav;  
 }  
}

## 快速入门步骤

1. 配置DispatcherServlet前端控制器，这个是SpringMVC核心！
   1. 配置servlet和servlet-mapping
   2. 配置spring的配置文件
2. 完成spring配置文件
   1. 处理器映射器
   2. 处理器适配器
   3. 视图解析器
3. 完成controller的编写
   1. controller是一个函数式接口，重写方法返回一个modelandview对象，缺点很明显，一个controller只能有一个方法，不符合开发需求！因此有了注解
4. 把controller映射一个地址，在spring配置文件中配置相应的地址

<bean id="/hello" class="com.example.controller.HelloController"/>

## 快速入门注解开发

依赖不变，

springmvc配置文件如下(框架固定死的，了解配置的作用意义即可)

<!--自动扫描包-->  
<context:component-scan base-package="com.example.controller"/>  
<!--让springmvc不处理静态资源  
 web网页存在大量静态资源，如css,img,js等资源！配置该注解让springmvc不会  
 拦截此类web静态资源！  
-->  
<mvc:default-servlet-handler/>  
<!—

@RequestMapping注解完成请求地址的映射关系

@RequestMaping的生效需要注册处理器映射器和处理器适配器，  
 而annotation-driven自动完成处理器映射器和处理器适配器的实例注入  
-->  
<mvc:annotation-driven/>  
<!--视图解析器-->  
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
 <property name="prefix" value="/WEB-INF/jsp/"/>  
 <property name="suffix" value=".jsp"/>  
</bean>

注解开发控制器

@Controller  
@RequestMapping("/HelloController")//@RequestMapping完成地址映射  
public class HelloController{  
 @RequestMapping("/hello")  
 public String test(Model model){  
 model.addAttribute("msg","helllllll");  
 return "test";  
 }  
}

## 注解详解

使用注解的前提条件：配置自动扫包Component-scan

@Component：注解在组件上（和@Service,@Repository）用于自动检测和适宜类路径扫描自动配置的bean,注释类和bean之间存在一对一映射，即一个类只有一个bean

@Service：注解在业务层上,功能和@Component类似

@Repository：注解在数据访问层，作用是将接口的实现类交给spring管理，和@Component类似（实际上就是完成了接口的实现类，然后生成bean交给spring容器）

@RequestParam：注解在方法参数上，绑定前端传递的参数

@PathVirable：springmvc的restful风格使用，将url参数绑定给控制器方法的参数

@Controller：代表这个类被spring接管，被这个注解的类中的所有方法，如果返回值是String,那么就会被视图解析器解析！控制器中多个映射地址可以指向同一个页面，控制器与视图之间是弱耦合关系！

@RestController：等效于@Controller+@ResponseBody，将返回的数据解析为字符串，不作为跳转页面！

@ResponseBody：注解在controller方法上，将方法返回的数据转化为json数据

### 请求映射@RequestMapping

@RequestMapping：映射一个地址，可以使用在类和方法上，如果注解在类上，则注解在类上的地址为父地址，访问字方法的地址需要加上类上的映射地址！，必须配合@Controller使用，否则无法生效

@RequestMapping属性：

1. value：映射的路径，注解内只有一个参数默认就是value
2. method：指定请求的方式,method的值如下



method = RequestMethod.GET等效于 @GetMapping注解！意味着只接受get请求！同理

method = RequestMethod.POST等效于@PostMapping,意味着只接受post请求

请求映射路径不能完全一致，映射地址如果相同，请求的方式也相同就会出现Ambigous模棱两可异常！

如果映射地址没有设置method属性，则默认post和get请求都接受！

method属性值为一个数组以接受多种不同类型的请求！如：

method = {RequestMethod.GET, RequestMethod.POST}

params属性：字符串类型数组,格式如下：

params = {param,param = value,param! = value,!param }

param:表示请求必须携带param请求参数

param = value:表示请求必须携带参数param,且值为value

param!=value:表示请求必须携带参数param,且值不能为value

!param: 表示不能携带参数param

headers属性：字符串类型数组，百度百科了解！

## RestFul风格

简介：RestFul就是一个资源定位以及资源操作的风格，不是标准也不是协议，只是一种风格，基于这个风格设计的软件可以更简介，更有层次，更易于实现缓存等机制！

传统方式访问web资源url地址严格符合URL标准

使用RestFul方式可以自定义风格！



在springmvc中使用@PathVariable注解，让方法参数的值绑定到URL地址中的参数上！

@RequestMapping("/restAdd/{x}/{y}")  
//url地址上的x,y值传递给方法中的a,b  
public String test2(@PathVariable(‘x’) int a,

@PathVariable(‘y’) int b, Model model){  
 /\*restful方式手动传参，形式如下：  
 http://localhost:8080/rest/restAdd/100/100  
 \*/  
 int res = a+b;  
 model.addAttribute("msg","结果为"+res);  
 return "test";  
}

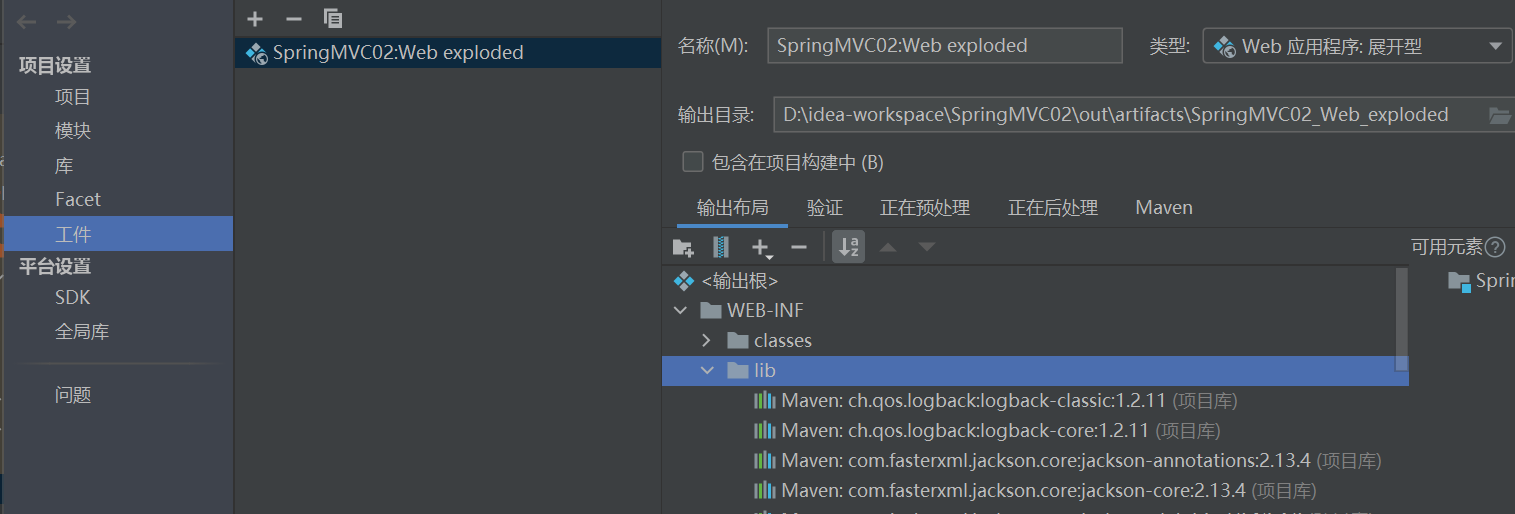
RestFul风格的好处：简洁，高效，安全！

## IDEA环境问题

可能遇到的问题：404

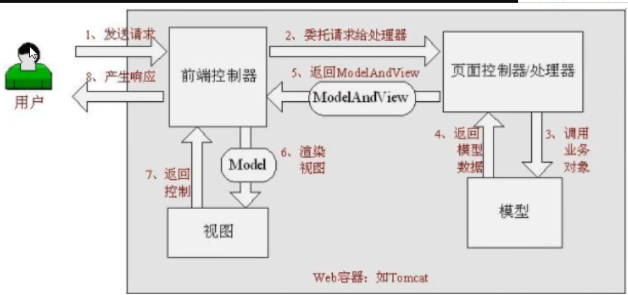
* 查看控制台输出，是不是jar包缺失
* jar包存在，就是在IDEA项目发布中，添加lib依赖
  + 因为项目通过tomcat发布，所以需要把依赖的包放入tomcat，才能运行！
  + 操作步骤如下：
    - 文件-》项目结构-》工件-》WEB-INF目录下创建lib目录-》将maven中的依赖包全部导入lib文件夹
    - 重启项目即可！
* 重启tomcat

这是由于idea创建web支持时不会创建lib目录，和eclipse不同！这属于环境问题！

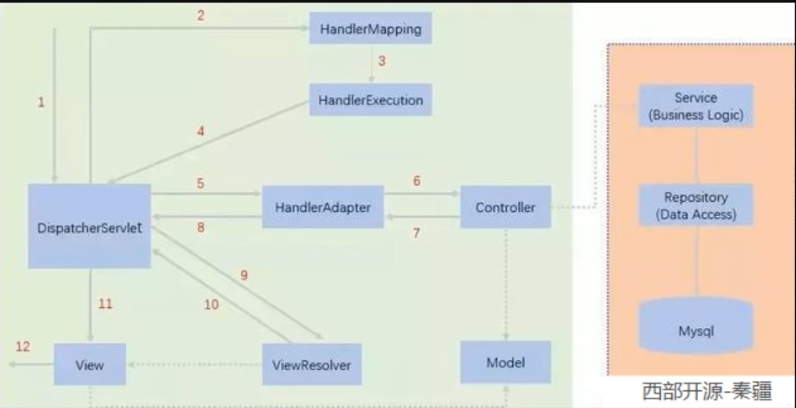


## SpringMVC流程原理

如图



流程图：实线部分为框架的功能，虚线部分是开发人员完成的功能！



详解：DispatcherServlet表示前置控制器，是整个springMVC的控制中心，用户发出请求，DispathcerServlet接受请求并拦截请求！在web.xml文件中配置的映射地址是/,代表拦截一切请求！

假设请求为：http://localhost:8080/SpringMVC/hello

该url拆分为三个部分：

* http://localhost:8080服务器域名以及端口号
* SpringMVC部署在服务器上的web站点
* hello为控制器

根据流程图分析：

大体可分为三部分，步骤1-4划分为一步，步骤5-8划分为一部分，剩下的是一部分

细分为以下：

* 步骤1：该url请求被DispatcherServlet拦截
* 步骤2：HandlerMapping为处理器映射，DispathcerServlet调用处理器映射
* 步骤3：HandlerExecution表示具体的Handler,主要作用是根据url查找控制器，以上示例找到的控制器为hello
* 步骤4：HandlerExecution将查找的控制器返回给DispatcherServlet
* 步骤5：DispatcherServlet将指定的控制器传给HandlerAdapter处理器适配器，按照特定的规则去执行Handler（查找controller对应的实体类）
* 步骤6：Handler让具体的Controller去执行
* 步骤7：controller将具体的执行信息返回给HandlerAdapter,具体的信息就是一个ModelandView
* 步骤8：HandlerAdapter将modelandview返回给DispatcherServlet
* 步骤9：DispatcherServlet调用视图解析器来解析接受到的modelandview
  + 具体解析步骤：
    - 获取modelandview中的数据
    - 解析modelandview中的视图名
    - 拼接视图，得到视图的路径
    - 将数据渲染到该视图上
    - 呈现给用户，步骤完结！

## 页面跳转方式

方式1：

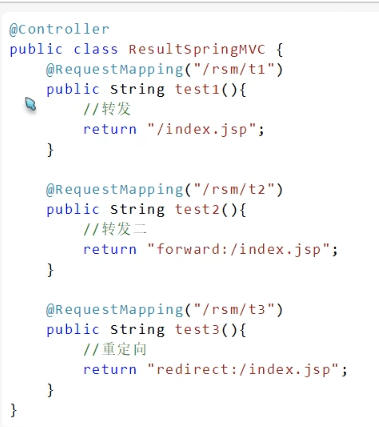
设置ModelAndView对象，根据view的名称配合视图解析器跳转到指定页面！

方式2：

使用ServletAPI方式跳转，该方式不会使用视图解析器，页面跳转需要手动配置准确路径！不推荐使用！

HttpServletRequest进行请求转发, HttpServletResponse重定向！可以将两个对象作为参数添加进方法中！（servlet作为底层服务端可以获取请求！）

方式3：SpringMVC实现请求转发和重定向，不配置视图解析器的情况下如图

这种方式本质上和方式二一样，通过字符串映射了servlet，然后实现页面跳转，需要配置完整路径！建议使用第一种。无错无异常且便于阅读理解便是最好的方法！

在配置了视图解析器的情况下：默认是转发，重定向就在字符串中增加redirect:



重定向不走视图解析器，所以重定向地址为控制器映射地址！

## 参数传递和显示

### 参数传递

1. 前端传递的参数名和绑定的方法中的参数名一致则可以直接绑定，不推荐使用
2. 通过@RequestParam(“请求参数名”)，将请求参数名传递给方法中的参数

@RequestMapping("/t1")  
//@RequestParam注明从前端获取参数名为username,绑定该方法的name参数!参数传递必加此注解  
public ModelAndView test1(@RequestParam("username") String name){

1. 传递对象参数，从前端传递的多个参数会匹配方法参数(对象)的属性，如果名称一致，则该对象的属性值为前端传递的参数，如果没有找到则该对象的此属性值为空！

/前端接受一个对象user(id,name,age)id在数据库中为主键自增长，所以前端不需要传递

//接受前端传递的参数对象，匹配User对象中的字段名，各个字段和请求参数之间如果名字一致则匹配,如果不一致该字段为空！  
//也可以通过使用三个@RequestParam来匹配参数！  
@RequestMapping("/t2")  
public String test2(User user){

注意：

当前端接受一个对象user(id,name,age)id在数据库中为主键且自增长,由于前端传递过来的对象参数中不包含主键，所以dao层mapper映射文件要确定增加数据的字段！

对应的控制器方法中参数直接封装成对象！

例如public String getUser(User user)//前端传递的两个参数封装成该方法中的user对象！

//底层sql语句写法

insert into user(name,pwd) values (#{name},#{pwd})

//这种写法会报SQL不匹配异常

insert into user values (#{name},#{pwd})

### 参数显示

* Model对象的addAttribute(key,value)方法,通过el表达式获取
* ModelAndView对象的addObject(key,value)方法，通过el表达式获取
* ModelMap,继承了LinkedHashMap

## 中文乱码问题

通过过滤器解决中文乱码问题

步骤如下：

* 创建过滤器实现Filter接口(javax.servlet包下)
* 在web.xml中注册servlet

详情查看javaweb笔记过滤器Filter

## JSON

json的意义和作用：

当前主流开发架构是前后端分离模式，在前后端分离中，后端提供接口，提供数据！前端独立部署，负责渲染后端数据！

前后端的交互就通过JSON数据！

### 前端操作JSON对象

通过JS内置的JSON对象的两个方法，Stringify（）转化成JSON对象，parse（）把JSON转化成对象

//对象转化成JSON字符串  
var ***uj*** = ***JSON***.stringify(***user***);  
***console***.log(***uj***);  
//JSON转化成对象  
var ***us*** = ***JSON***.parse(***uj***);  
***console***.log(***us***);

### 后端操作JSON数据

后端操作JSON，工具包Jackson和FastJson,这里以Jackson为例，详情百度百科

* Jackson是目前比较好的json解析工具，导入依赖

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.10.1</version>

</dependency>

maven导入依赖后，更新maven,等待maven下载！下载完成后项目配置里面注意将相关jar引入lib目录下使其生效！

* 对象转化成JSON

@RequestMapping(value = "/j1",produces = "application/json;charset=utf-8")//乱码问题解决  
@ResponseBody//不走视图解析器，直接返回字符串！  
public String j1() throws JsonProcessingException {  
 User u0 = new User(1,"孙道名",99);  
 //创建JSON的工具类  
 ObjectMapper objectMapper = new ObjectMapper();  
 //把对象转化成JSON，参数也可以使用对象集合！  
 String str = objectMapper.writeValueAsString(u0);  
 return str;  
}

* JSON转化成对象

//JSON字符串转对象，注意这里的json是一个对象字符串  
User uj = new User(0,"孙道名",99);  
//目标JSON字符串  
String suj = objectMapper.writeValueAsString(uj);  
//把目标字符串user类型JSON数据转化成user对象！  
objectMapper.readValue(suj,User.class);

乱码问题解决如上，@RequestMapping中的produces属性可以解决json乱码问题，同样，springmvc框架肯定也具有相应的功能，只需要在配置文件中配置即可！

具体配置如下：



## AJAX

简介：asychronous javascript and xml异步js和xml,即不需要刷新整个页面，发送一个异步请求！通常配合表单使用。

原生创建ajax请求步骤繁多，推荐使用jquery！（引入jquery文件）



需要掌握的参数：url,data,success,error

代码示例：

控制器层：

//@ResponseBody注解的方法将返回的数据解析为json字符串！

@ResponseBody  
@RequestMapping("/queryUserLike")  
public List<User> getUL(String name){  
 name = name + "%";  
 List<User> users = userService.queryUserLike(name);  
 return users;  
}

前端：

//搜索框标签

<input type="text" placeholder="请输入查询用户" name="queryUser" id="search" class="form-control" onblur="getU()">

//搜索框绑定的JS

function getU() {  
 $.post({  
 url:"${pageContext.request.contextPath}/user/queryUserLike",  
 //控制器方法中的参数从data中获取数据，参数名称要和data内的key一致  
 data:{"name":$("#search").val()},  
 //回调函数，将控制器返回的数据通过回调函数处理后显示  
 success:function (data) {  
 ***console***.log(data);  
 },  
 error:function () {  
 ***console***.log("请求错误");  
 }  
 })  
}

## 拦截器

过滤器和拦截器的区别：拦截器是AOP思想的具体应用

拦截器是SpringMVC框架中的，只有使用了该框架才能使用！

拦截器之后拦截访问的控制器方法，不会对静态资源进行拦截

拦截器的定义：实现HandlerInterceptor接口

拦截器的配置：隶属于springmvc框架，所以在springmvc配置文件中配置拦截器

<mvc:interceptors>  
 <mvc:interceptor>  
 <!--拦截地址,\*\*代表一切请求路径-->  
 <mvc:mapping path="/\*\*"/>  
 <!--具体使用哪个拦截器-->  
 <bean class="com.study.config.MyInterceptor"/>  
 </mvc:interceptor>  
</mvc:interceptors>

实现拦截器HandlerInterceptor重写的三个方法

//pre方法在访问请求前执行，即请求被pre方法拦截，pre方法放行后请求执行！

//pre方法用于拦截非法请求，post和after方法用于写日志

//return true,执行下一个拦截器！放行  
//return false,请求拦截，不执行下一个拦截器  
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
 System.*out*.println("pre拦截器执行了，这是处理前");  
 return true;  
}  
  
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) throws Exception {  
 System.*out*.println("post执行了，这是处理后");  
}  
  
public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception ex) throws Exception {  
 System.*out*.println("after执行了，这是清理");  
}

## 文件上传和下载

文件上传和下载是常见需求之一，springMVC支持文件上传和下载。但是springmvc上下文默认没有装配multipartResolver,因此默认情况下不能处理文件上传工作。

### 文件上传

如果需要使用spring的文件上传功能，需要在上下文中配置multipartResolver

前端表单要求：上传文件表单method必须为post,并且设置enctype属性值为multipart/form-data。这样浏览器会将用户上传的文件以二进制形式发送给服务器！

表单enctype属性值：

* application/x-www=from-urlencoded：默认方式，只处理表单域中的value值，采用这种编码方式的表单会将表单域中的值处理成url编码方式
* multipart/from-data:这种编码方式会以二进制形式处理表单数据，会把文件域指定文件的内容也封装到请求参数中，不会对字符编码
* text/plain；处理把空格转化文“+”号外，其他字符都不做编码处理，适用于表单发送邮件！

后端依赖：commons-fileupload和commmons-io包

<dependency>  
 <groupId>commons-fileupload</groupId>  
 <artifactId>commons-fileupload</artifactId>  
 <version>1.4</version>  
</dependency>

springmvc配置multipartResolver

<!--文件上传配置,这个bean的id不能自动以，固定死的-->  
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  
 <!--请求的编码格式，必须和jsp的pageEncoding属性一致，以便能正确读取表单内容，默认是ISO-8859-1-->  
 <property name="defaultEncoding" value="utf-8"/>  
 <!--上传文件的大小上下限，单位是字节，10485760为10mb-->  
 <property name="maxUploadSize" value="10485760"/>  
 <property name="maxInMemorySize" value="40960"/>  
 </bean>

### 文件下载

步骤：

* 设置响应头
* 读取文件
* 写出文件
* 执行操作
* 关闭流

示例：D:\idea-workspace\SSM\_total\src\main\java\com\study\controller\FileController.java

# Mybatis

简介：mybatis是一款优秀的持久层框架，它支持定制化SQL，存储过程以及高级映射，MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

持久化：将数据从内存写入外存就是持久化

优点：

* 简单易学
* 灵活
* sql和代码的分离提高了可维护性
* 提供映射标签，支持对象与数据库表的映射
* 支持编写动态SQL

综上：mybatis就是用来简化JDBC代码的框架！

## 入门案例

1. 搭建环境
2. 在resources目录下创建sql配置文件mysql-config.xml，配置文件中配置了连接数据库的参数信息！以及注册mapper.xml文件



3.从 XML 中构建 SqlSessionFactory

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先配置的 Configuration 实例来构建出 SqlSessionFactory 实例。

官方规定写法获取SqlSessionFactory：

String resource = "path/mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

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

以上建议封装在一个工具类中，方便使用

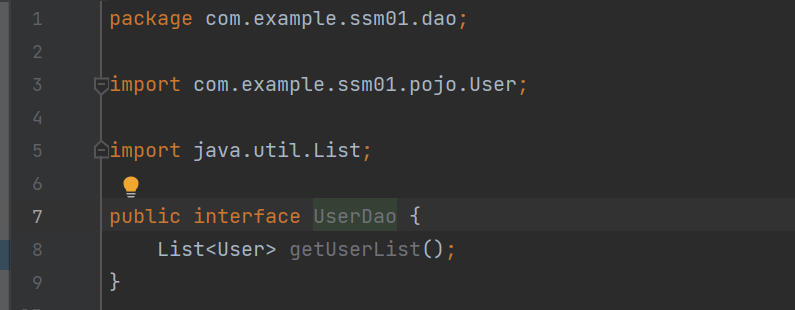
详情查询mybatis文档<https://mybatis.org/mybatis-3/zh/index.html>

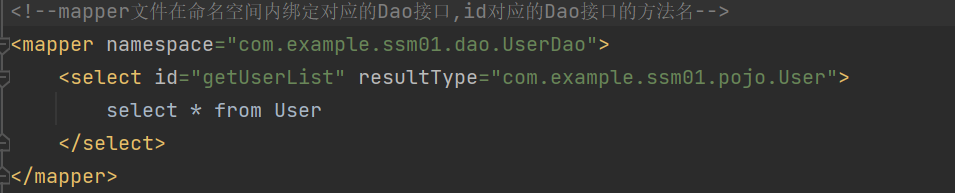
1. 从 SqlSessionFactory 中获取 SqlSession

既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的实例。SqlSession 提供了在数据库执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

public static SqlSession getSession(){  
 return *sqlSessionFactory*.openSession();  
}

创建Mapper文件并绑定对应的Dao层接口，接口名要和mapper文件名一致！





注意：如果项目构建的target目录中，mapper配置文件和对应的接口文件无法完成映射，则会出现绑定异常！可以通过在pom中配置资源文件的方式解决！



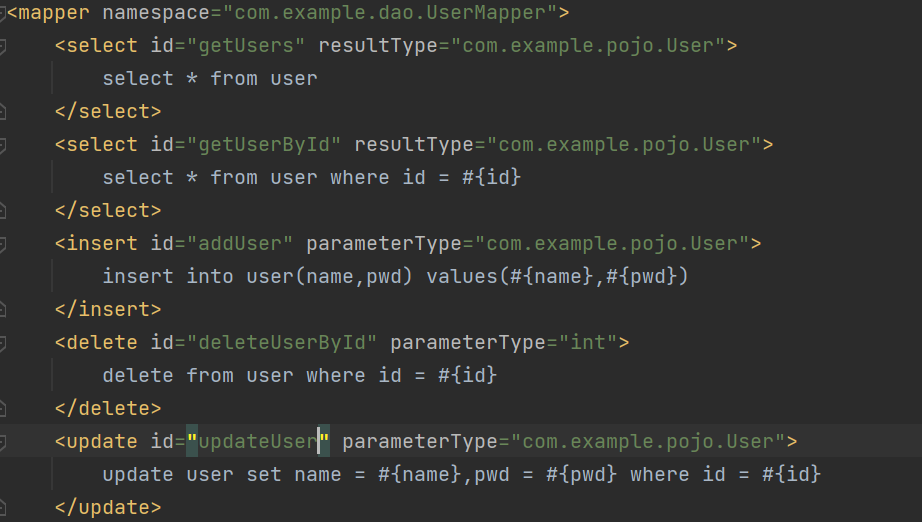
## CRUD操作

mapper映射文件的namespace对应接口全限定名,接口的每一个方法对应mapper文件内的一个SQL语句id，通过接口类class创建映射对象可以直接调用具体mapper映射文件捏的SQL语句！

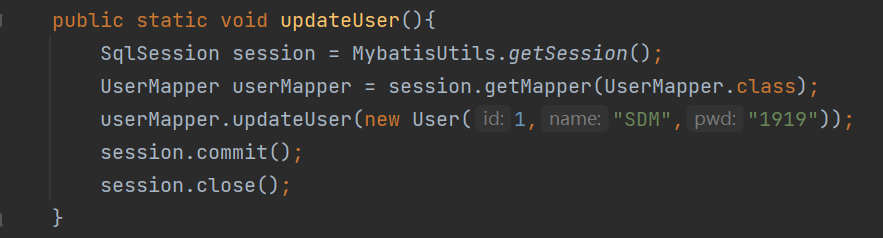
UserMapper userMapper = session.getMapper(UserMapper.class);  
List<User> userList = userMapper.getUsers();

### select,insert,delete,update

增删改需要提交事务！





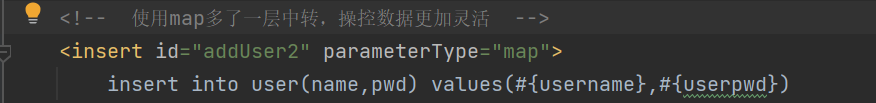


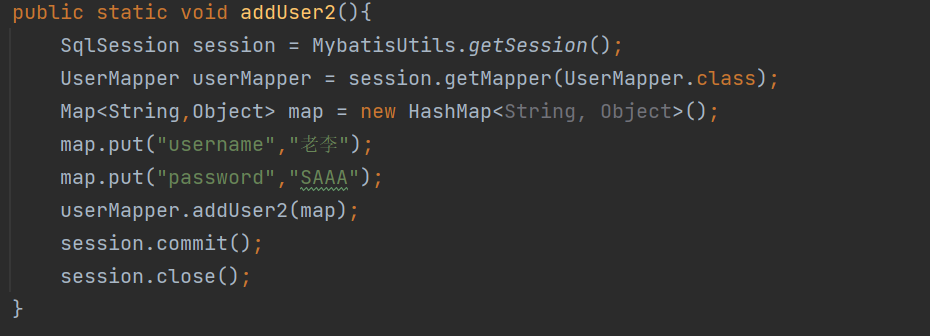
#{参数}：从方法中获取参数,且自动加上””

${参数}：从方法中获取参数，不会加上””

## 万能的Map

如果参数类型就是映射的对象，values内的取参的名称必须符合user对象的属性，使用map多了一层映射关系，传递参数时直接去除key即可（key可以自定义，所以灵活）





## 模糊查询

在代码层面传递通配符%和\_,切忌sql语言字符串拼接，会导致SQL注入！

## 配置解析\*\*\*

官方文档：<https://mybatis.org/mybatis-3/zh/configuration.html>

### environment环境

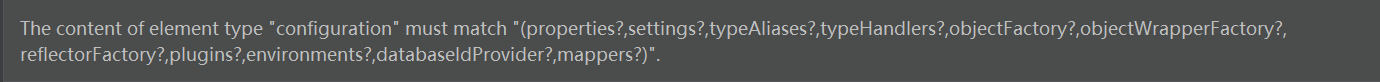
在environments中配置多种environment环境，通过default设置默认环境！

<configuration>  
 <properties resource="db.properties"/>  
 <environments default="development">  
 <environment id="development">  
 <transactionManager type="JDBC"/>  
 <dataSource type="POOLED">  
 <property name="driver" value="${driver}"/>  
 <property name="url" value="${url}"/>  
 <property name="username" value="${username}"/>  
 <property name="password" value="${password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
 <mappers>  
 <mapper class="com.example.dao.UserMapper"/>  
 </mappers>  
</configuration>

### propertie属性

引入外部配置文件配置一些固定参数，动态替换！也可以在该标签中新配置property标签引入新的属性！外部配置文件优先级大于内部新增属性！

注意：mybatis的xml文件中标签要严格按照以下的顺序写！

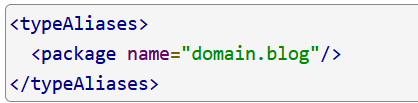


<properties resource="db.properties"/>

### typealise起别名

给java类型起别名！有两种起别名方式，具体配置和扫包！

起别名，将指定类的全限定名指定为固定的名称！

扫包形式：domain.blog包下的类，如果没有注解，别名默认类名首字母小写形式！

如果有注解，别名就是注解名！

注解：@Alias(“名称”)

### setting设置

开启缓冲，延迟加载，驼峰命名法映射等…………

官方文档：<https://mybatis.org/mybatis-3/zh/configuration.html>

### 映射器Mapper

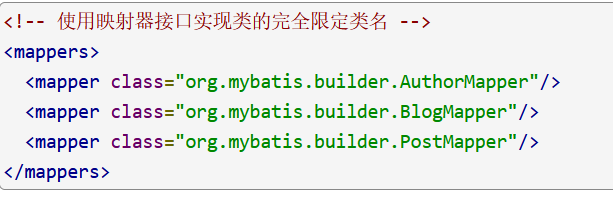
用于注册映射mapper文件

方式一：推荐使用，路径相对于类路径下

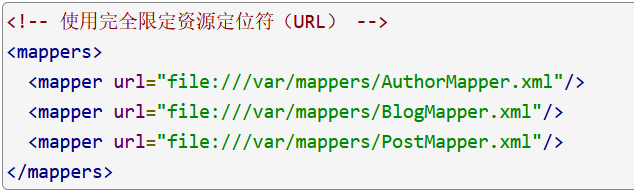


方式二：使用class注册文件

* 条件一：mapper文件和接口必须同名
* 条件二：二者必须在同一个包下



方式三：使用文件路径注册mapper



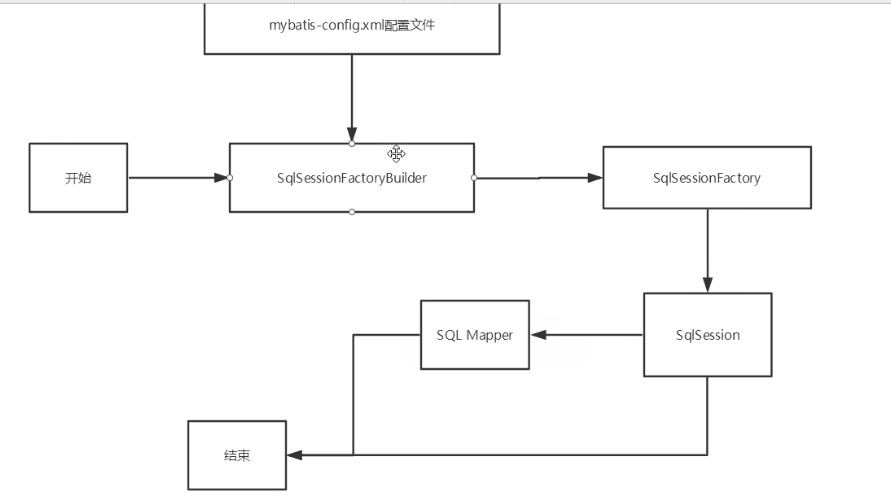
方式四：扫描包形式注册mapper文件，

限制条件：和方式二class注册文件条件一致



## 声明周期和作用域

mybatis流程！



如上流程：

SqlSessionFactoryBuilder:

* 一旦创建了factory，就不再需要它了！
* 所以作用域为局部变量，使用一次即不再使用！

SqlSessionFactory在运行期间一致存在，所以最佳作用域为应用作用域。

最简单的使用就是单例模式或者静态单例模式，如下：



SqlSession:由sqlsessionFactory创建（sql会话池）创建，由于池中可创建会话数量有限且要保证其复用性，因此sqlsession使用完后必须关闭，不关闭的后果就是多个会话超出池的上限，造成宕机！

sqlSession可创建多个mapper,每个mapper都代表着一个业务！

## ResultMap\*\*\*

官方文档：<https://mybatis.org/mybatis-3/zh/sqlmap-xml.html#Result_Maps>

作用：解决实体类和字段名不一致的问题！ResultMap 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。

<!--结果集映射-->  
<resultMap id="UserMap" type="com.example.pojo.User">  
 <!--column:数据表字段，property:实体类属性-->  
 <result column="id" property="id" />  
 <result column="name" property="name" />  
 <result column="pwd" property="password" />  
</resultMap>  
<select id="getUserById" resultMap="UserMap">  
 select id,name,pwd from user where id = #{id}  
</select>

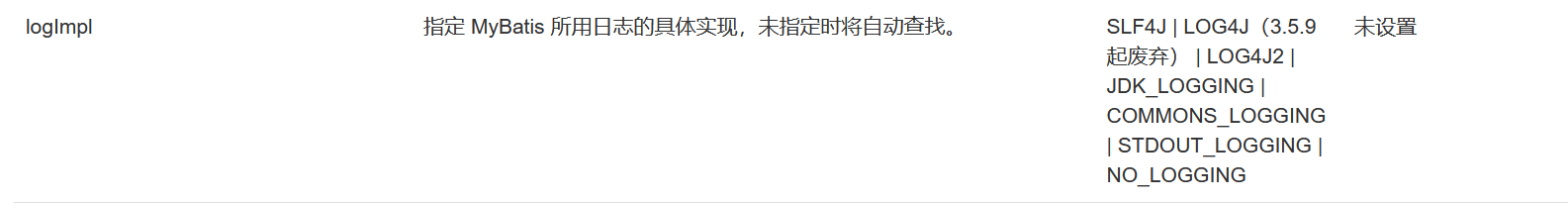
解决办法二：sql语句起别名

## 日志

日志工厂，用于记录日志信息，用于排除异常！

日志工厂：常用

* LOG4J
* STDOUT\_LOGGIN



在mybatis中具体使用哪一个日志实现，在mybatis配置文件中设置设定！

实例：STDOUT\_LOGGING

//注意name和value值严格符合官方文档，大小写必须一致且不能添加空格等！

<settings>  
 <setting name="logImpl" value="STDOUT\_LOGGING"/>  
</settings>

LOG4J

简介：

* LOG4J可以控制日志信息输出的目的地是控制台，文件，GUI组件
* 可以控制日志的输出格式
* 定义每一条日志信息的级别
* 通过配置文件来配置，不需要修改相应的代码！

第一步导入依赖包：

<dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.17</version>  
</dependency>

第二步配置日志的配置文件log4j.properties（该包会自动读取类路径下的该配置文件）

第三部在mybatis配置文件中设置log4j

第四步测试：导入包

import org.apache.log4j.Logger;

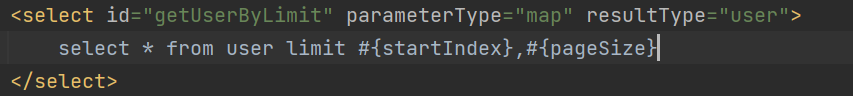
<settings>  
 <setting name="logImpl" value="LOG4J"/>  
</settings>

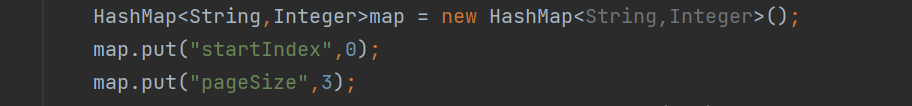
## 分页

为什么要分页？减少数据处理量

核心SQL语句：

mapper文件的sql语句需要传递两个参数作为SQL语句中limit的两个参数！所以方法的传递参数中设置一个map作为参数。map中放入值！





RowBounds分页

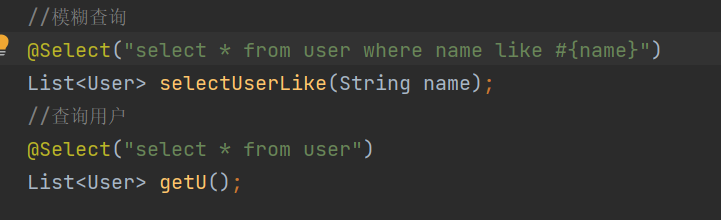
## 使用注解开发

面向接口编程：根本原因解耦！

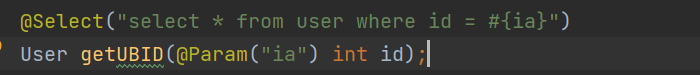
接口反映了开发人员对系统的抽象理解！

纯注解开发不再需要mapper.xml文件，因此在mybatis配置文件夹中需要注册mapper接口，所以使用class方式注册！

同一个mapper文件映射不能注册两次



@Param给参数起别名用于sql语句取值，方法存在多个参数时，必须使用@Param



## Mybatis执行流程\*\*\*

1. 编写mybatis配置文件
   1. 配置数据库连接池
   2. 配置事务管理
   3. 注册mapper
   4. 引入外部配置文件
   5. 给实体类取别名
2. 获取SqlSessionFactory
   1. 解析mybatis配置文件
      1. 获取sqlsessionFactory实例
      2. 获取sqlsession
3. 实现CRUD
   1. mapper文件和接口映射
   2. 执行失败，事务回滚
   3. 执行成功，提交！

//解析配置文件

String resource = "mybatis-config.xml";  
InputStream inputStream = Resources.*getResourceAsStream*(resource);  
*sqlSessionFactory* = new SqlSessionFactoryBuilder().build(inputStream)

## Lombok

作用：减少重复代码的书写，比如getter/setter/toString方法等

安装lombok插件：

idea，文件-》设置-》插件-》搜索lombok（可以看到lombok的注解信息）->安装

在pom文件中导入依赖！

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.20</version>  
</dependency>

常用注解：

* @Getter/@Setter：作用在类，生成get,set方法
* @Data：作用在类上，自动生成该类的无参构造，get/set,equals,hashcode,toString方法
* @AllArgsConstructor：生成全参构造器
* @NoArgsConstructor；生成无参构造器
* @ToString：生成toString方法
* 更多注解自行百科

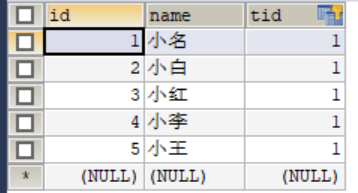
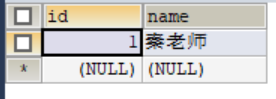
对于lombok存在一定的争议性，因为其改变了写源码的方式！理智选择，自行斟酌！

## 关联查询

### 多对一关系

例如 ：多个学生关联一个老师，对老师而言，是一个老师关联多个学生

数据库建立两张表，老师表字段id,name,学生表id,name,tid，tid关联老师表的id



@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Student {  
 private Integer id;  
 private String name;  
 //关联老师  
 private Teacher teacher;  
}

@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Teacher {  
 private Integer id;  
 private String name;  
}

<!—第一种方式association查询嵌套处理  
 1.查询学生  
 2.根据查询的学生tid查询对应的老师  
-->  
<select id="getStudents" resultMap="StudentTeacher">  
 select \* from student  
</select>  
<resultMap id="StudentTeacher" type="Student">  
 <result property="id" column="id" />  
 <result property="name" column="name"/>  
 <!--复杂属性需要单独处理-->  
 <association property="teacher" column="tid" javaType="Teacher" select="getTeacher"/>  
</resultMap>  
<select id="getTeacher" resultType = "Teacher">  
 select \* from teacher where id = #{id}  
</select>

第二种方式结果嵌套处理

<select id="getStudents" resultMap="StudentTeacher">  
 select s.id sid,s.name sname,t.name tname  
 from student s,teacher t  
 where s.tid = t.id  
</select>  
<resultMap id="StudentTeacher" type="Student">  
 <result property="id" column="sid" />  
 <result property="name" column="sname"/>  
 <!--复杂属性需要单独处理-->  
 <association property="teacher" javaType="Teacher">  
 <result property="name" column="tname"/>  
 </association>  
</resultMap>

### 一对多关系

以上，对老师而言是一对多关系！

老师的属性

public class Teacher {  
 private Integer id;  
 private String name;  
 private List<Student> students;  
}

@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class Student {  
 private Integer id;  
 private String name;  
 //关联老师  
 private Teacher teacher;  
}

//按照结果嵌套查询

<select id="getTeacherS" resultMap="TeacherStudent">  
 SELECT s.`id` sid,s.`name` sname,t.`id` tid,t.`name` tname  
 FROM `student` s,`teacher` t  
 WHERE s.`tid` = t.`id` and t.id = #{tid}  
</select>  
<resultMap id="TeacherStudent" type="Teacher">  
 <result property="id" column="tid"/>  
 <result property="name" column="tname"/>  
 <!--对象属性用association,集合属性用collection  
 javaType = “”指定的属性类型  
 集合中的泛型信息使用ofType  
 -->  
 <collection property="students" ofType="Student">  
 <result property="id" column="sid" />  
 <result property="name" column="sname"/>  
 </collection>  
</resultMap>

//嵌套查询

<select id="getTeacherS" resultMap="TeacherStudent">  
 select \* from teacher where id = #{tid}  
</select>  
<resultMap id="TeacherStudent" type="Teacher">  
 <collection property="students" javaType="ArrayList" ofType="Student" select="getStudentByTeacherId" column="id"/>  
</resultMap>  
<select id="getStudentByTeacherId" resultType="Student">  
 select \* from student where tid = #{tid}  
</select>

综上：核心是利用结果集映射完成关联查询！java属性是对象用association，是集合用collection！ 利用resultMap完成java类属性和数据库表字段的映射！

## 动态SQL

动态SQL：根据不同的条件生成不同的SQL语句！

官网文档：<https://mybatis.org/mybatis-3/zh/dynamic-sql.html>

## Mybatis缓存

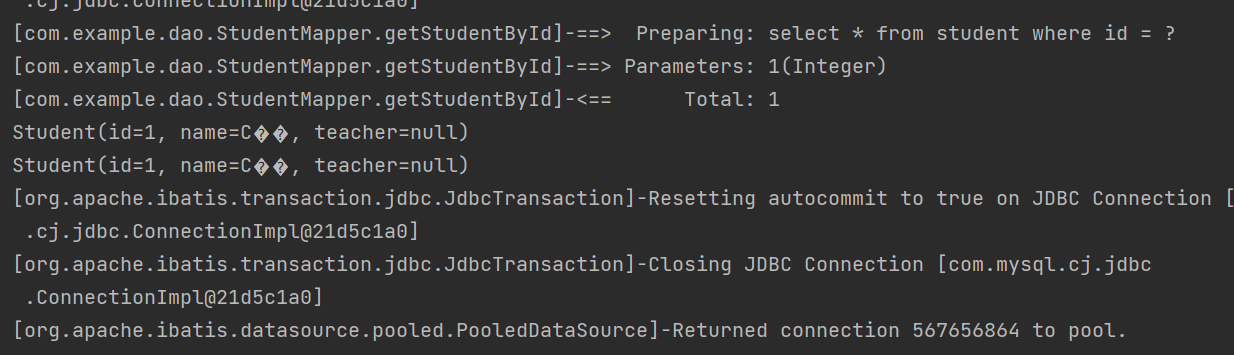
缓存的作用：查询数据库需要先连接数据库，这非常消耗资源！未了降低资源消耗，查询数据时的结果我们暂存内存中，下次查询同样的结果时直接从内存读取！

缓存使用的条件：经常查询并且不经常改变的数据

### 一级缓存

默认情况下，一级缓存开启（sqlsession级别的缓存，也称之为本地缓存），只在一次sqlsession中有效，即连接数据库到关闭数据库期间有效！

一级缓存实例：查询同一条记录



第二次查询结果直接得出！

一级缓存失效的情况：

1. 查询不同的记录
2. 增删改数据都会导致缓存失效，重新缓存！
3. 手动清理缓存sqlsession.clearCache();

### 二级缓存

默认情况下只会启用一级缓存，二级缓存也叫全局缓存，一级缓存作用域太低，所以诞生了二级缓存！

二级缓存基于namespace,一个命名空间对应一个二级缓存！

二级缓存工作机制：

* 一个会话查询一条数据，这个数据就会被放在会话的一级缓存中
* 当会话关闭或者提交，一级缓存消失！此时如果二级缓存开启，一级缓存会把数据丢入二级缓存中，新的会话查询信息就可以从二级缓存中获取数据
* 不同的mapper文件查出的数据会放在对应的缓存中！

二级缓存的配置如下：官网<https://mybatis.org/mybatis-3/zh/sqlmap-xml.html#cache>



开启全局缓存步骤：

1. 开启全局缓存，在mybatis配置文件的setting标签中配置cacheEnabled，详情查询官网的mybatis的设置<https://mybatis.org/mybatis-3/zh/configuration.html#settings>

<settings>  
 <setting name="logImpl" value="LOG4J"/>  
 <!--显示开启全局缓存-->  
 <setting name="cacheEnabled" value="true"/>  
</settings>

1. 在mapper配置文件中添加一行标签<cache/>,标签内具体配置可查询官网！
2. mapper文件中的crud标签中有usecache属性代表是否开启缓冲！

测试问题：java.io.NotSerializableException: com.example.pojo.Student

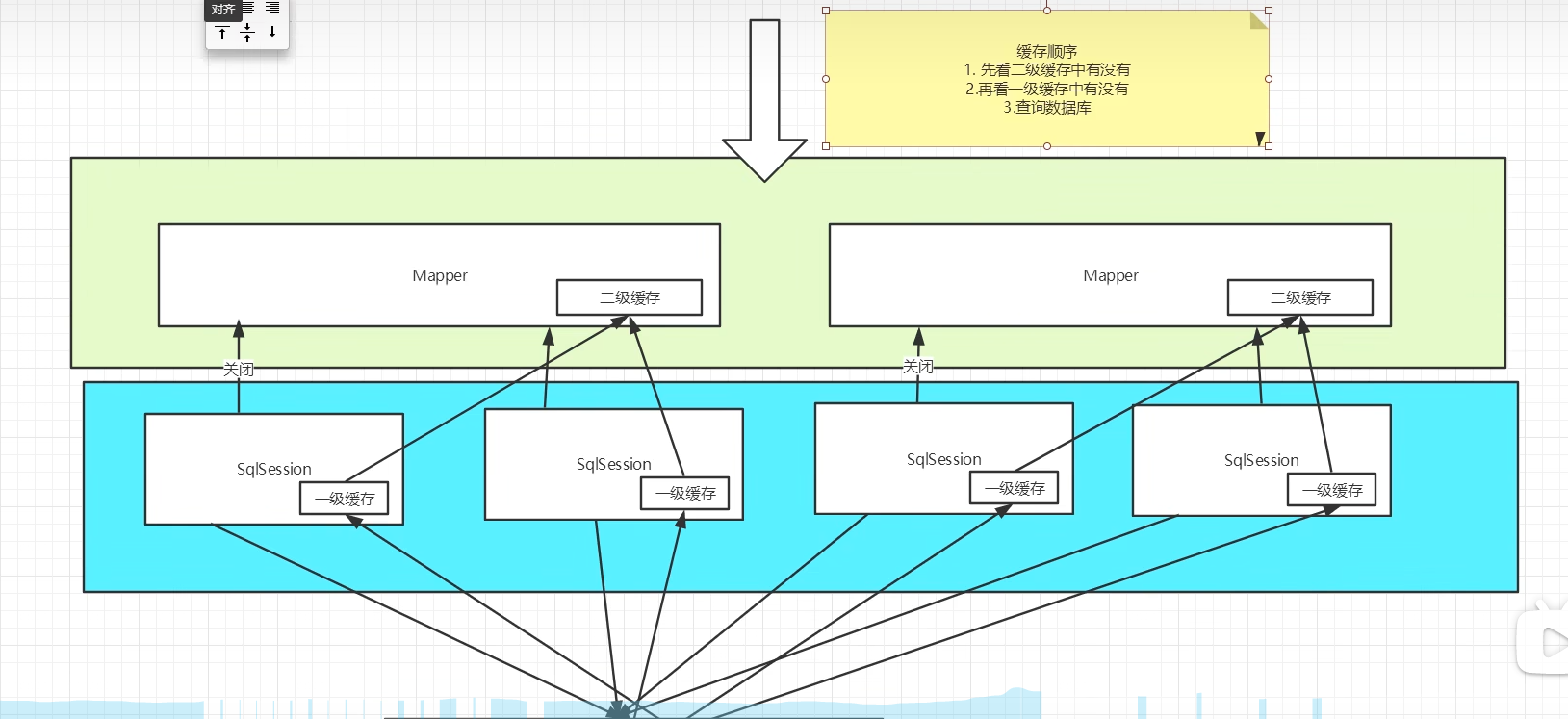
解决：将对应的实体类序列化！

<cache/>  
<select id="getStudentById" resultType="Student" useCache="true">  
 select \* from student where id = #{id}  
</select>

缓存机制原理

如图：最底层是数据库

机制：用户查询数据，先看二级缓存有没有该数据，没有就进一级缓存查看有没有数据，再没有才查询数据库！通过此方式增强查询数据的效率！



### 自定义缓存Encache

一种广泛使用的开源java分布式缓存，现多使用redis缓存，可以使用该类缓存覆盖一级二级缓存！

<cache type="com.domain.something.MyCustomCache"/>

# SSM整合

web.xml中配置前端控制器，前端控制器初始化参数引入applicationContext.xml配置文件！再配置编码过滤器！ 代码路径D:\idea-workspace\SSM\_total

javabean：类的属性必须封装，且提供外部访问接口，有无参全参两个构造方法，符合托名命名法！

作为网络传输对象需要实现序列化接口！

根据web项目的三层架构创建多个spring配置文件:

1. applicationContext配置文件：全局配置文件
   1. 关联dao,service,controller三个配置文件
2. spring-dao配置文件：dao层配置文件
   1. 引入连接数据库的参数文件
   2. 配置数据库连接池
   3. 配置sqlSessionFactory
      1. 引入mybatis-config配置文件
   4. 配置dao接口扫描包，由spring接管dao接口（不需要手写dao的实现类）
3. spring-service配置文件：service层配置文件
   1. 配置业务层扫描（注解开发使用）
   2. 配置事务管理
   3. 配置aop支持
4. spring-controller配置文件：controller层配置文件
   1. 配置扫描包
   2. 配置注解驱动（自动注入控制器映射器和控制器适配器，解决JSON字符串乱码问题）
   3. 配置视图解析器