刘昱江 中心名称 + 班级号 + 名字 527161772

# SSM框架总结

## Mybatis

### Mybatis作用

说明:Mybatis是持久层框架,负责用户与数据库交互.

### 优势

1. 简化了用户操作数据库的过程.
2. 利用了ORM映射关系简化代码 简化了数据库结果直接封装对象.

2.1操作数据库 mybatis需要自己编辑sql. 非自动

2.2结果集自动映射为对象 自动

半自动的ORM映射框架

### 核心思想

#### 利用ORM思想

**对象关系映射**（英语：**(Object Relational Mapping**，简称**ORM**，或**O/RM**，或**O/R mapping**），是一种程序技术，**用于实现面向对象编程语言里不同类型系统的数据之间的转换**[1]  。从效果上说，它其实是创建了一个可在编程语言里使用的--“虚拟对象数据库”。

面向对象是从[软件工程](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%B7%A5%E7%A8%8B/25279)基本原则（如耦合、聚合、封装）的基础上发展起来的，而关系数据库则是从数学理论发展而来的，两套理论存在显著的区别。为了解决这个不匹配的现象，对象关系映射技术应运而生。

对象关系映射（Object-Relational Mapping）提供了概念性的、易于理解的模型化数据的方法。ORM方法论基于三个核心原则： 简单：以最基本的形式建模数据。 传达性：[数据库结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%BB%93%E6%9E%84/5507713)被任何人都能理解的语言文档化。 精确性：基于数据模型创建正确标准化的结构。 典型地，建模者通过收集来自那些熟悉应用程序但不熟练的[数据建模](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BB%BA%E6%A8%A1)者的人的信息开发[信息模型](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF%E6%A8%A1%E5%9E%8B/5176137)。建模者必须能够用非技术企业专家可以理解的术语在概念层次上与数据结构进行通讯。建模者也必须能以简单的单元分析信息，对样本数据进行处理。ORM专门被设计为改进这种联系。

例子:

1. JDBC中获取的结果集对象之后需要手动的封装数据.之后再次返回对象.
2. ORM思想 对象关系映射 user表----user类

Mybatis中利用ORM思想 用户主需要编辑 resultType/resultMap即可获取结果对象.

#### 封装JDBC

1.数据库操作只能是JDBC的方式.

2.Mybatis封装了JDBC.

3.Mybatis通过SqlSessionFactory创建的sqlSession实现了数据库的操作.

## Spring

### 作用

Spring是框架中最为重要的.因为Spring框架可以整合全部的第三方框架.让程序调用浑然一体.

### Spring核心

#### IOC/DI(一)

IOC:控制反转

**将对象创建的权利,交给spring容器管理.由Spring容器管理对象的生命周期.**

#### 单例对象生命周期 (默认)

**周期:与容器同生共死**

当程序启动时加载spring容器时,由Spring容器根据包扫描/xml形式直接实例化对象.当对象全部创建完成后.Spring容器启动成功!!!!.

实例化对象过程:

例子:

<bean id=”userController” class=”包名.UserController”></bean>

对象实例化时调用对象的**无参构造方法,**所以自己编辑构造方法时先写无参构造.

#### 多例对象生命周期

**周期:当程序调用时才会新建.当线程结束时对象销毁.**

**场景:获取第三方链接时.**

**例子:**

说明:多线程操作数据库时,问几个链接??? 答:多个

服务器一个,框架一份 数据库一个 链接多份.

### Spring构建对象的原理IOC/DI

问题1: Spring容器如何管理对象???

问题2: DI依赖注入时如何注入????

### 问题1:如何管理对象

说明:Spring容器内部维护了一个超大的Map<ID,对象>集合.具体调用如下:

1.spring容器启动时,首先会加载spring容器的配置文件 application.xml

<bean id=”user” class=”com.jt.pojo.User”/>

2.当程序解析配置文件时 加载bean标签.首先根据class类型利用反射机制调用无参构造实例化对象.之后将对象保存到map集合中.其中map中的key就是Bean的Id.

2.1.当程序使用注解时.

@Controller

Public class UserController{}

如果程序使用注解.则必须配置包扫描.当程序解析@Controller注解时.

获取注解标识的类型(class).利用反射机制调用无参构造实例化对象.之后将数据保存到spring维护的Map集合中.k:userController v:实例化的对象.

案例:

@Service

public class UserServiceImpl implements UserService{

}

### 问题2:DI注入原理

例子:

@Controller

Public class UserController {

@Autowired

Private UserService userService;

/\*

\*public void setUserService(**UserService(接口) userService**){

\* this.userService = userService;

\*}

原理实现:

当调用set注入方法是,由于方法中编辑的是接口,无法直接实例化.

利用该接口userService,**查询该接口的实现类**.获取实现类名称/类型

如果获取到接口实现类的名称”userServiceImpl”,之后从spring维护的Map集合中获取对象UserService对象.之后执行set方法,完成注入.

\*

Spring的Map集合 <userServiceImpl,userService对象>

<userController,userCOntroller对象>

\*/

}

注入方式:装配过程 构造注入/**set注入**

@Service

Public class UserServiceImpl implements UserService{

}

提示:

1.对象实例化有顺序

2.当对象中有依赖注入时,先加载注入对象

### Spring核心-AOP

#### 作用

AOP采用代理对象的方式对原有的方法进行扩展.实现了软件的松耦合.(解耦)

#### 配置AOP步骤

1.开启AOP注解.

SpringBoot自动配置了

@EnableAspectJAutoProxy

SSM框架必须配置

<!--ssm框架中开启AOP -->

<aop:aspectj-autoproxy/>

作用:@Aspect注解生效

2.使用注解标识切面类@Aspect

3.配置通知 5大通知类型 完成业务具体**方法**

4.配置切入点 bean/within/execution/annotation

#### 切入点表达式写法

//切入点:判断什么时候执行通知方法

/\*\*

\* 切入点表达式写法

\* 1.bean: 包.类名

\* 如果这个类中的方法执行则执行通知

\* 2.winthin: 包名.类名 与上述相同

\* 上述2个切入点表达式是粗粒度,控制的是类级别

\* 3.annotation:

\* 标识注解 包名.注解名称

\* 当某个方法执行时有该注解时,执行通知方法.

\*

\* 4.execution 最重要

\* 语法: execution(返回值类型 包名.类名.方法名(参数列表))

\* 例子1: execution(int com.jt..\*.add\*(..))

\* 匹配返回值类型为int类型,在com.jt下的所有包下的类,并且

\* 类中方法名称以add开头的所有方法,并且参数类型任意.

\*

\* 例子2: execution(\* com.jt..\*.\*(..));

\* execution(\* com.jt..\*(..));

\* 上述2个切入点表达式 效果相同

\*

\*/

#### 前置通知用法

//1.前置通知 方法执行之前执行

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

**public** **void** before() {

System.***out***.println("我是前置通知");

}

#### 后置通知用法

//2.后置通知 业务方法执行完成之后return之后执行

@AfterReturning("execution(\* com.jt..\*.\*(..))")

**public** **void** afterRuturn() {

System.***out***.println("我是一个后置通知");

//记录程序执行之后的状态

}

#### 异常通知方法

//3.异常通知 当业务方法(目标方法)执行报错时,执行异常通知

@AfterThrowing("execution(\* com.jt..\*.\*(..))")

**public** **void** afterThrow(Throwable throwable) {

System.***out***.println("我是异常通知");

//记录目标方法执行报错后的状态

}

#### 最终通知写法

//4.最终通知 该通知了解 最不常用

@After("execution(\* com.jt..\*.\*(..))")

**public** **void** after(JoinPoint joinPoint) {

System.***out***.println("我是最终通知,无论方法执行成功与否,都要执行");

}

#### 4大通知总结

**上述4大通知类型只能做程序的记录.不能改变程序的执行的状态.**

#### 环绕通知用法

//ProceedingJoinPoint 只能必须写在环绕通知

//5.环绕通知能够控制方法的执行状态,改变程序执行轨迹. 最重要!!!!

@Around("execution(\* com.jt..\*.\*(..))")

**public** Object around(ProceedingJoinPoint joinPoint) {

Object 目标方法执行的结果 = **null**;

**try** {

System.***out***.println("事务开始");

System.***out***.println("数据库入库操作~~~~service~~~mapper");

目标方法执行的结果 =

joinPoint.proceed();

System.***out***.println("事务提交");

}**catch** (Throwable e) {

e.printStackTrace();

System.***out***.println("事务回滚");

}

**return** 目标方法执行的结果;

}

#### 利用环绕通知控制目标方法

/\*\*

\* 缓存业务场景

\* 第一次查询

\* 1.查询数据库

\* 2.使用Map<k:user对象>集合保存数据库数据

\*

\* 第二次查询

\* 从缓存中根据key获取数据

\*

\* 规定:使用目标方法的第一个参数为key

\*

\* 定义目标方法

\* public User findUserById(Integer id){

\* ...

\* }

\*/

**private** Map<Integer,User> cacheMap = **new** HashMap<>();

@Around("execution(\* com.jt..\*.\*(..))")

**public** Object cacheAround(ProceedingJoinPoint joinPoint) {

Object[] orgArgs = joinPoint.getArgs(); //获取参数列表

**int** idKey = (**int**) orgArgs[0];

User user = cacheMap.get(idKey);

**try** {

**if**(user==**null**) {

//表示缓存中没有数据,查询数据库

user = (User) joinPoint.proceed();

cacheMap.put(idKey, user);

}**else** {

//joinPoint.proceed();

**return** user;

}

} **catch** (Throwable thro) {

System.***out***.println(thro.getMessage());

}

**return** user;

}

### 关于AOP总结

1.AOP可以对原有目标方法进行扩展.

2.AOP目的实现代码的解耦.

3.环绕通知使用最多的.并且能够控制方法的执行轨迹.

另外的四大通知类型,只做记录,不做(不能)修改

4.切入点表达式4中写法

粗粒度: bean within

细粒度: annotation **execution(最常用)**

**5. 切面(完成某些特定业务逻辑) = 切面类(2个注解) + 通知方法 + 切入点表达式**

# SpringMVC

## 作用

**实现了用户的交互的过程**.当用户需要做某些操作,利用SpringMVC接收参数.并且利用框架的内部处理.最终将返回的结果(页面/data)返回用户的过程.

## Servlet缺点

1. web.xml文件中有8行配置信息.

2. servlet接收参数时只能接收String类型.如果需要其他类型必须格式转化.

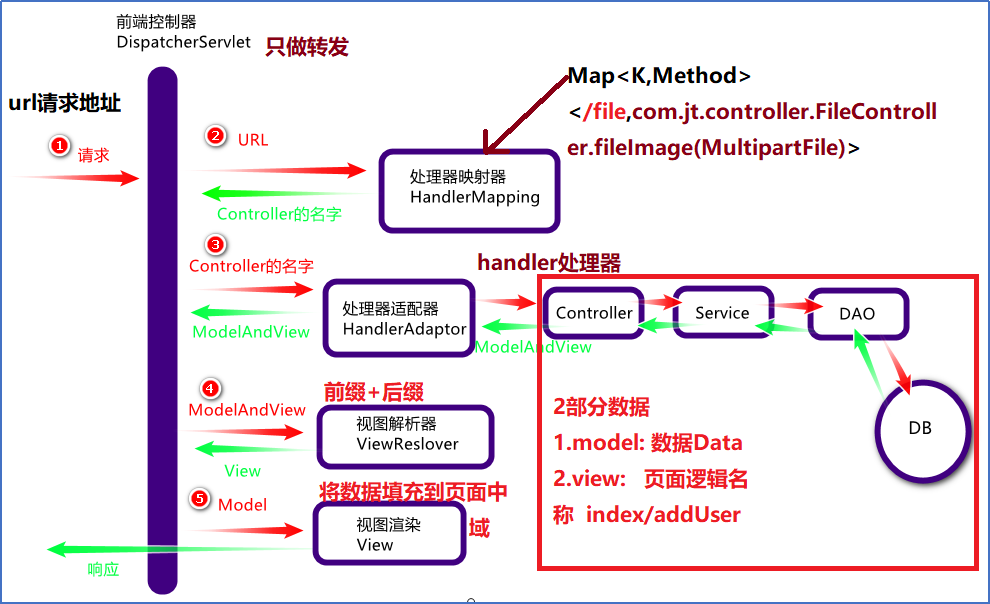
3. 一个servlet调用时,需要重新编辑2个方法 doGet/doPost 代码繁琐

4.用户每一个请求都需要对应一个Servlet.

5.Get/post乱码问题严重.

6.重定向/转发时业务复杂.

## SpringMVC内部调用原理



准备工作:当程序启动时需要加载配置文件信息.当配置程序加载成功.则tomcat服务器启动成功,用户可以正常调用.

1.当程序启动时,首先扫描@RequestMapping注解.之后将请求的路径域方法的类型保存到 Map集合中.

2.程序启动时根据配置文件指定视图解析器的前缀和后缀.

真正的调用过程:

1.当用户发起请求时

localhost:80/user/findUserById?id=100

2.该请求被前端控制器拦截.之后将请求转发给处理器映射器请求处理.

处理器映射器接收请求后.根据用户的url地址匹配具体的处理方法.

查找到**业务处理方法**后.返回给前端控制器. 404

3.前端控制器根据业务处理方法.调用**处理器适配器**.处理器适配器根据用户需求查询合适的处理器handler执行业务方法.

handler处理器根据处理器映射器中获取的方法.之后从controller层开始调用.程序执行成功之后返回有效数据.通过modelAndView封装数据.

**ModelAndView**内部封装2部分数据

model:服务器返回页面的数据

view: 程序正常流转跳入页面.

4.端口控制器获取modelAndView对象之后开始解析数据.调用视图解析器帮助获取返回页面的具体路径.

视图解析器获取请求之后根据之前已经注入的前缀和后缀信息.进行路径的拼接: /WEB/INF/views/**user**.jsp,拼接之后返回数据.

5.数据利用视图渲染过程,将数据封装到**request**域中.之后跳转页面展现用户正确的信息.

el表达式从域中取数据.

### 业务代码展现

/\*

\* 业务说明:

\* 根据用户id查询数据

\*url:localhost:80/user/findUserById?id=100

\* 请求方式: get

\* 获取数据之后携带数据跳转页面

\* 前缀: /WEB-INF/views/

\* 后缀 .jsp

\*

\*JSP取值与el/jstl表达式联用. ${"key"}

\* \*/

@RequestMapping("/findUserById")

**public** String findUserById(Integer id,Model model) {

User user = xxxxService.findUserById(id);

//page/request域/session/context

//将数据封装到request域中.之后页面利用表达式取值.

model.addAttribute("key", user);

**return** "user";

}