# Spring 基础

## Spring 简介

### Spring 是什么

1. Spring 是一个工厂（Factory）
2. Spring是一个容器（Container）
3. Spring 是一个框架（Framework）

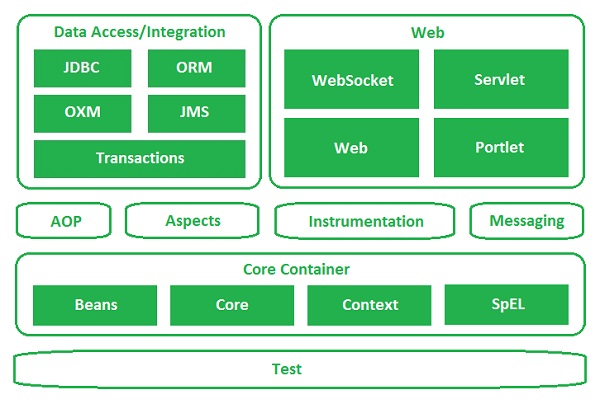
### Spring 应用动机

1. 优化对象创建、管理，降低侵入性。
2. 简化应用开发，降低耦合。
3. 解耦并提供整体解决方案

实际企业项目开发中我们通常会借助Spring整合hibernate,mybatis等技术框架实现快速高效开发。

### Spring 基本架构

Spring 框架的基本结构图如下：



Spring框架中最核心的为IOC（控制反转），并基于IOC实现了AOP，整合了数据访问及Web组件功能。

# Spring IOC 容器基本应用

## Spring IOC 概述

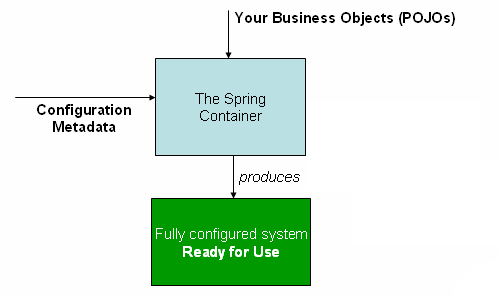
IoC全称是Inversion of Control，被译为控制反转，是指程序中对象的获取方式发生反转，由最初的new方式创建，转变为由第三方框架创建、注入（DI），它降低了对象之间的耦合度。

Spring容器是IOC机制的一种实现，同时IOC也是Spring框架的的基础和核心，它借助DI（Dependency Injection）方式实现。

## Spring Bean容器

Spring容器是Spring框架中的核心组件，负责创建Bean对象(一种简单规范的java对象)及管理这些对象之间的依赖关系，。

Spring容器是如何工作的呢？在Spring项目运行时， Spring容器负责读取项目中的元数据信息（这些元数据信息可能是基于xml也可能是基于注解实现），然后基于这些信息创建bean对象实例，如下图所示：



IOC 是Spring框架中的一种机制，提供了容器中对象的控制反转功能，这个功能的实现需要借助依赖查找和依赖注入。

### Spring 容器元数据配置

Spring中元数据的配置有两种方式，基于xml或annotation方式。这个小节了解

基于xml方式的元数据实现bean元素的基本配置。

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

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

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

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

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

<bean id="..." class="...">

*<!-- collaborators and configuration for this bean go here -->*

</bean>

<bean id="..." class="...">

*<!-- collaborators and configuration for this bean go here -->*

</bean>

*<!-- more bean definitions go here -->*

</beans>

Spring中多个元数据配置文件的实现方式

<beans>

<import resource="services.xml"/>

<import resource="resources/messageSource.xml"/>

<import resource="/resources/themeSource.xml"/>

<bean id="bean1" class="..."/>

<bean id="bean2" class="..."/>

</beans>

### Spring 容器初始化

Spring中容器的类型为Application类型，其初始化方式如下：

ApplicationContext context =

**new** ClassPathXmlApplicationContext("beans.xml");

Spring容器初始化时需要加载元数据的配置信息，例如beans.xml文件中声明的元数据的相关配置。

### Spring 容器的使用

Spring容器初始化以后，可以通过容器的getBean(…)方法获取容器中的Bean对象，进而操作Bean对象，例如

Contex.getBean(“xxxService”,XxxService.class);

## Spring Bean对象

### Spring bean 对象的命名

Spring容器中，每个Bean都需要有自己的名字，这个名字的定义需要符合标识符的规范，然后借助id或name属性指定。

### Spring对象的实例化

Spring容器创建Bean对象的方法有如下3中方式：

1. 通过构造器实例化Bean对象
2. 通过静态工厂方法实例化Bean对象
3. 通过实例工厂方法实例化Bean对象。

<bean id="date1" class="java.util.Date"/>

<bean id="cal" class="java.util.Calendar"

factory-method="getInstance"/>

<bean id="date2" factory-bean="cal"

factory-method="getTime"/>

### Spring bean 对象的作用域

在理解Bean的作用域时重点掌握如下两个作用域：

1. Singleton spring容器中此bean的实例只有一份
2. Prototype spring 容器中每次获取bean都会创建新一个新的对象。

其作用域的配置可记住bean标签的scope属性进行指定，例如

<bean id="helloService"

class="com.company.spring.factory.HelloServiceImpl"

scope="singleton"/>

### Spring bean 对象的生命周期

Spring bean对象的生命周期由Spring容器进行管理，并可以在bean元素定义时借助init-method属性指定初始化时调用的方法，借助detroy-method属性指定销毁时要调用的方法（销毁回调方法只适用于singleton作用域范围的对象）。

<bean id="helloService"

class="com.company.spring.factory.HelloServiceImpl"

scope="singleton"

init-method="doInit"

destroy-method="doDestory"/>

### Spring bean 对象的延迟加载

Spring 容器默认会在容器启动时将所有singleton 作用域下的bean进行实例化，如果不想启动时实例化，而是在第一次使用时实例化，可以通过bean元素定义时的lazy-init=“true”的方式进行指定。

<bean id=*"helloService"*

class=*"com.company.spring.factory.HelloServiceImpl"*

scope=*"singleton"*

init-method=*"doInit"*

destroy-method=*"doDestory"*

lazy-init=*"true"*/>

提示：假如希望容器中所有的Bean都要采用延迟加载，则可以在配置文件的顶级<beans/>元素中设置default-lazy-init属性的值为true.

## Spring Bean依赖

### 依赖注入基础

Spring 容器中的Bean对象通常会存在一定的依赖关系，而这种依赖关系的实现在Spring 框架中要借助于DI机制。其中DI就是借助对象管理对象依赖关系的一个过程。

Spring中提供的依赖注入方式有构造注入和设置注入，其中构造注入就是借助构造方法的参数实现对类中属性值的注入，set注入就是借助set方法的参数实现其属性值的注入。

Spring 依赖注入时，可以实现基本值的注入，Bean对象的注入，集合的注入，spring表达式方式的注入等等。

set注入：（重点掌握）property

<bean id="dataSource1"

class="com.company.spring.util.DataSource">

<property name="driverClassName"

value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///test"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

<bean id="jdbcTemplate1"

class="com.company.spring.injection.JdbcTemplate">

<property name="dataSource" ref="dataSource1"/>

</bean>

constructor注入：(了解)

<bean id="dataSource2" class="com.company.spring.util.DataSource">

<constructor-arg value="com.mysql.jdbc.Driver"/>

<constructor-arg value="jdbc:mysql:///test"/>

<constructor-arg value="root"/>

<constructor-arg value="root"/>

</bean>

<bean id="jdbcTemplate2"

class="com.company.spring.injection.JdbcTemplate">

<constructor-arg ref="dataSource2"/>

</bean>

### 依赖注入高级

实际项目中对象的属性可能会有数组类型，集合等类型，这些类型的注入的方式可参考如下案例实现。

定义一个相对复杂的对象

**public** **class** ComplexObject {

**private** String[] hobby;

**private** List<String> address;

**private** Map<String,String> phones;

**private** Properties configs;

//set,get,…

}

对象在配置文件中的实现：

<bean id="complexObject"

class="com.company.spring.injection.ComplexObject">

<property name="hobby">

<list>

<value>足球</value>

<value>篮球</value>

</list>

</property>

<property name="address">

<list>

<value>北京</value>

<value>上海</value>

</list>

</property>

<property name="phones">

<map>

<entry key="pKey1" value="123"/>

<entry key="pKey2" value="456"/>

</map>

</property>

<property name="configs">

<props>

<prop key="cKey1">#{cfg.port}</prop>

<prop key="cKey2">#{cfg.host}</prop>

</props>

</property>

</bean>

<util:properties id="cfg"

location="classpath:config.properties"/>

提示：

其中util:properties元素用于引入配置文件，并可以通过spring表达式获取对应key的值，然后注入给其它对象的属性。

### 自动装配实现

Spring IoC容器可以自动装配（autowire）相互协作bean之间的关联关系，autowire可以针对单个bean进行设置，autowire的方便之处在于减少xml的注入配置

Spring 配置文件中通过bean元素的autowire属性指定自动装配规则，一共有四种类型值

|  |  |  |
| --- | --- | --- |
|  | **属性值** | **描述** |
|  | NO | 禁用自动配置（默认） |
|  | ByName | 按名字自动装配(重点掌握) |
|  | ByType | 按类型自动状态（重点掌握），但有多个类型时会出错 |
|  | Constructor | 与byType类似，不同之处在于它应用于构造器参数。 |

例如：

<bean id="jdbcTemplate3"

class="com.company.spring.injection.JdbcTemplate"

autowire="constructor">

</bean>

<bean id="jdbcTemplate3"

class="com.company.spring.injection.JdbcTemplate"

autowire="byName">

</bean>

# Spring IOC 容器注解应用

## Bean组件扫描

### 组件扫描定义

Spring中通过指定一个包路径，由系统自动扫描该包及其子包所有组件类,当发现组件类定义前有特定的注解标记时，就将该组件纳入到Spring容器。

这种方式的配置等价于原有XML配置中的<bean>定义功能组件扫描，可以替代大量XML配置中的<bean>元素的定义

### 扫描方式配置

使用组件扫描，首先需要在XML配置中指定扫描父级package路径，例如

<context:component-scan

base-package=”com.company.spring/>”

在这个配置中，容器会自动扫描org.example包及其子包下所有组件，并实例化bean对象。

## 系统组件注解

### 组件注解标记

指定扫描类路径后，并不是该路径下所有组件类对象都由Spring容器创建并管理，只有在组件类定义前面有以下注解标记时，才会扫描到Spring容器。

|  |  |  |
| --- | --- | --- |
|  | **注解名** | **说明** |
|  | @Component | 通用注解 |
|  | @Repository | 持久层组件应用注解 |
|  | @Service | 业务层组件应用注解 |
|  | @Controller | 控制层组件应用注解 |

提示：实际项目中某个业务的实现，是要进行分层处理的，每一层的对象都应该有自己的职责，目的是便于维护与扩展。

例如：

数据层对象

@Repository

**public** **class** SysUserDaoImpl **implements** SysUserDao{

**public** **void** saveUser(Object obj) {

System.***out***.println("dao.save.user");

}

}

业务层对象

@Service

**public** **class** SysUserServiceImpl **implements** SysUserService{

…

}

控制层对象

@Controller

**public** **class** UserController {

}

### 系统组件命名

当一个组件在扫描过程中被检测到时，会生成一个默认id值，

默认id为小写开头的类名。也可以在注解标记中自定义id，例如

@Component

**public** **class** IdGenerator {

**public** IdGenerator() {

System.***out***.println("IdGenerator");

}

}

当需要此对象时候,ctx.getBean(“idGenerator”).

### 系统组件作用域

通常受Spring管理的组件，默认的作用域是"singleton"。如果需要其他的作用域可以使用@Scope注解，只要在注解中提供作用域的名称即可。

@Scope("singleton")

@Component

**public** **class** IdGenerator {

**public** IdGenerator() {

System.***out***.println("IdGenerator");

}

}

在Bean组件中还可以借助@PostConstruct和@PreDestroy注解标记指定

初始化和销毁回调方法。这个知识点了解即可。

@Scope("singleton")

@Component

**public** **class** IdGenerator {

**public** IdGenerator() {

System.***out***.println("IdGenerator");

}

@PostConstruct

**public** **void** init() {

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

}

@PreDestroy

**public** **void** destory() {

System.***out***.println("destory()");

}

}

### 组件依赖关系注解

实际项目中对象与对象之间通常会具有一定的依赖关系，这种关系假如基于spring注解的方式进行注入，可使用@Autowired/@Qualifier,@Resource注解在对象的属性或set方法上进行声明。

* @Autowired注解应用

@Service

**public** **class** SysUserServiceImpl **implements** SysUserService{

@Autowired

**private** SysUserDao userDao;

**public** **void** saveUser(Object obj) {

userDao.saveUser(obj);

}

}

其中@Autowire注解默认按类型进行注入，假如希望按名字进行注入，可以再结合@Qualifier一起使用，例如

@Service

**public** **class** SysUserServiceImpl **implements** SysUserService{

@Autowired

@Qualifier("userDaoImpl")

**private** SysUserDao userDao;

**public** **void** saveUser(Object obj) {

userDao.saveUser(obj);

}

}

* @Resource注解应用

@Controller

**public** **class** SysUserController {

@Resource

**private** SysUserService userService;

**public** **void** doSaveUser(Object obj) {

userService.saveUser(obj);

}

}

其中@Resource属于JavaEE中的一个注解，默认按对应的属性名进行装配注入，假如希望指定的名称进行注入，可通过注解中的name属性进行配置，例如

@Controller

**public** **class** SysUserController {

@Resource(name="userServiceImpl")

**private** SysUserService userService;

**public** **void** doSaveUser(Object obj) {

userService.saveUser(obj);

}

}

### 组件注入表达式应用

Spring应用中可借助@Value注解通过Spring表达式获取对应properties文件中的值，并将其值注入到对应对象的属性上。

首先在XML配置中指定要注入的properties文件，例如

config.properties文件内容

prefix=CGB

suffix=JT

spring中配置文件中添加如下语句。

<util:properties

id=*"cfg"* location=*"classpath:config.properties"*/>

然后在属性或Setter方法前使用@Value注解

@Scope("singleton")

@Component

**public** **class** IdGenerator {

@Value("#{cfg.prefix}")

**private** String prefix;

@Value("#{cfg.suffix}")

**private** String suffix;

…

}

# Spring Bean 依赖管理

## 依赖查找

实际项目会存在很多个Bean对象,这些对象之间会存在一定依赖关系,当某个Bean对象依赖于其它Bean对象,可以通过spring按照一定的规则(例如按类型或者按名字),进行依赖查找然后进行值的注入.

思考:（了解）

1. 对象与对象之间可能存在什么关系?(is a, has a,use a)

例如:

1. is a 关系

class A implements IA{} ; A 实现了IA 接口

class A extends B{}; A 继承了 B这个类

1. has a 关系

class Point{}

class Circle{

private Point point;

}

Circle对象中有一个Point类型的对象

1. use a 关系

class ProjectUtil{

public String getId(){

return UUID.randomUUID().toString();

}

}

在ProjectUtil类中使用了UUID这个类.

## 依赖注入

在Spring中对Bean属性赋值的过程我们称之为依赖注入.Spring应用中为Bean的属性注入值的方式有两种,set注入和构造注入,set注入式通过对象的set方法为对象属性赋值,构造注入为通过bean对象的构造方法为属性注入值.

例如:

1.依赖注入基本实现

set方式实现基本值的注入(重点掌握):

<bean id=*"dataSource"* class=*"beans.DataSource"*>

<property name=*"driverClassName"* value=*"com.jdbc.mysql.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql:///test"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</bean>

执行set注入时引用其它bean对象

<bean id=*"sessionFactory"*

class=*"beans.SqlSessionFactory"*>

<property name=*"dataSource"*

ref=*"dataSource"*/>

</bean>

构造方法实现基本值的注入(了解)

此注入必须有一个有两个参数的构造方法.

<bean id=*"dataSource"* class=*"beans.DataSource"*>

<constructor-arg value=*"10"*/>

<constructor-arg value=*"20"*/>

</bean>

执行构造注入时引用其它bean对象(必须有对应的构造方法)

<bean id=*"sessionFactory"*

class=*"beans.SqlSessionFactory"*>

<constructor-arg ref=*"dataSource"*/>

</bean>

2.集合类型对象值的注入:

对象定义:

**public** **class** CollectionObject {

**private** List<String> list;

**private** Map<String,Object> map;

**private** Properties properties;

**public** **void** setList(List<String> list) {

**this**.list = list;

}

**public** **void** setMap(Map<String, Object> map) {

**this**.map = map;

}

**public** **void** setProperties(Properties properties) {

**this**.properties = properties;

}

...

}

对象配置:

<bean id=*"collection"* class=*"beans.CollectionObject"*>

<property name=*"list"*>

<list>

<value>A</value>

<value>B</value>

</list>

</property>

<property name=*"map"*>

<map>

<entry key=*"k1"* value=*"100"* />

<entry key=*"k2"* value=*"300"* />

</map>

</property>

<property name=*"properties"*>

<props>

<prop key=*"username"*>root</prop>

<prop key=*"password"*>123456</prop>

</props>

</property>

</bean>

3.依赖注入时spring表达式的应用#{id.key}

在类路径下定义一个config.properties文件,内容如下

username=root

password=123456

在spring的配置文件中config.properties文件

<util:properties id=*"cfg"*

location=*"classpath:config.properties"*/>

在bean中为属性注入值时使用spring 表达式

<bean id=*"dataSource"* class=*"beans.DataSource"*>

<property name=*"username"* value=*"#{cfg.username}"*/>

<property name=*"password"* value=*"#{cfg.password}"*/>

</bean>

其中#为spring表达式特殊标记,{}为表达式内容,cfg为util:properties 标签中定义的id. username、password为配置文件等号左边的key

## 自动装配

Spring 应用中还可以按照一定规则自动为对象属性注入值,此机制实现的可以借助Bean标签中的autowire属性进行配置.此属性的值如下几个:

1. default : 不执行自动装配(默认)
2. byName: 按属性找到对应的set方法然后进行自动装配
3. byType:按属性类型对应的set方法相关参数类型进行装配
4. constructor:按构造方法参数类型进行装配

例如：

<bean id=*"jdbcTemplate"*

class=*"beans.JdbcTemplate"*

autowire=*"byType"*>

</bean>

此代码中的自动装配是按照jdbcTemplate类中相关属性类型查找对应参数类型的set方法,然后再依据其类型从spring容器中查找对应类型的bean对象,最后调用对应set方法进行值的注入.

本小结中先了解基于xml方式配置的自动装配机制(重点在注解方式的应用)

# Spring Bean 注解应用

## Spring Bean 注解描述

Spring容器中描述bean的方式有两种,一种是基于xml方式,一种是基于注解方式.

,现阶段基于注解方式描述Bean元素已经成为一种趋势.Spring中常用的bean注解有如下几种:

1. 修饰Bean类型(将bean交给spring容器管理)
2. @Component (描述通用性组件)
3. @Controller (描述MVC中控制层组件)
4. @Service (描述MVC 中业务层组件)
5. @Repository(描述MVC中持久层组件)

说明:这些组件实现的功能是一样的,只是应用的业务场景不同

1. 指定Bean作用域以及生命周期方法
2. @ Scope(描述bean的作用域)
3. @ PostConstruct(描述生命周期开始要调用的初始化的方法)
4. @ PreDestroy (描述生命周期结束时要调用的销毁的方法)
5. 自动装配注解(为属性注入值)
6. @Autowire (一般修饰构造方法或set方法或属性,默认按类型装配)
7. @Qualifier (配合Autowire注解,按名字进行装配,尤其是出类型相同的bean时,可以指定具体bean)
8. @Resource (一般修饰set方法或属性,默认按名字装配)

说明:无论默认按类型还是名字进行装配,在没有找到对应名字或类型对象时底层还可能转换机制.

## Spring Bean 注解应用

在spring应用中假如使用注解描述bean,定义bean的依赖机制,首先需要配置

类的扫描机制(扫描某个或多个包中的类),spring扫描到这个类以后,会检测类

上是否有spring中定义的相关注解的描述(例如@Controller,@Service),假如有spring底层会通过反射构建此类型的对象,然后进行管理.

### Bean对象声明注解应用

配置对指定包中类的的扫描,例如:

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

定义Bean组件,并使用spring注解修饰,例如

@Repository

**public** **class** SysUserDao {}

@Service

**public** **class** SysUserService {}

@Controller

**public** **class** SysUserController {}

说明:使用Spring注解修饰的类,默认bean的名字为类名并且首字母小写.

编写测试类从spring容器中获取相关Bean对象.

### Bean对象作用域注解应用

Spring 管理的Bean一般都会指定一个作用域,借助注解指定时需要

使用@Scope,注解中常用值为singleton或者prototype.

@Scope("singleton")

@Repository

**public** **class** SysUserDao {

}

### Bean对象生命周期方法注解应用

**Spring 管理的Bean对象都有生命周期,但不一定都有生命周期方法,假如有些Bean需要指定生命周期方法,基于注解实现时可以借助@**PostConstruct和@ PreDestroy,这个两个注解都属于JDK中的API,使用时需要注意版本问题

@Repository

**public** **class** SysUserDao {

**public** SysUserDao() {

System.***out***.println("SysUserDao()");

}

//备注:这个注解在jdk1.5是没有的

/\*\*借助此注解修饰的方法在对象初始化时执行\*/

@PostConstruct

**public** **void** init(){

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

}

/\*\*借助此注解修饰的方法在对象销毁时执行\*/

@PreDestroy

**public** **void** destory(){

System.***out***.println("destory");

}

}

### Bean对象自动装配注解应用

Spring 管理的Bean对象可以采用自动装配机制为属性赋值.基于注解方式进行自动装配,一般需要使用@Autowired,@Qualifier,@Resource这些注解.

1. @Autowired 可以修饰属性,构造方法,set方法,默认依据类型(属性类型,参数类型)为属性注入值.假如Spring容器中有多个相同类型的值,会参考名字进行匹配查找(属性名,set方法名,构造方法参数名),假如名字有相同的则注入,没有相同的会注入失败.
2. @Qualifier 配合@Autowired注解按名字为属性注入值.
3. @Resource 可以修饰属性或set方法,默认依据名字(属性名,set方法名)为属性注入值.假如spring容器中有名字相同但类型不同的bean就会注入失败,当没有找到对应名字的bean对象,此时会依据类型再次进行查找,假如相同类型则

直接注入,当有多个相同类型可能会注入失败(假如是按set方法进行注入,依据方法名字没找到,还会按参数类型查找,对应类型有多个,还可能会按参数名查找,假如没找对应的,则注入失败).

例如:

@Service("userService")

**public** **class** SysUserService {

@Autowired

**private** SysUserDao sysUserDao;

@Autowired

@Qualifier("date1")

**private** Date logDate;

}

@Controller

**public** **class** SysUserController {

//默认先按属性名进行依赖查找,找不到则按类型进行查找.

@Resource(name="userService")

**private** SysUserService sysUserService;

/\*

默认按set方法名先进行查找,找不到按类型查找

@Resource

public void setSysUser(SysUserService sysUserService) {

this.sysUserService = sysUserService;

}\*/

**public** SysUserService getSysUserService() {

**return** sysUserService;

}

}

# 总结

## 重点和难点分析

1. Bean对象的依赖注入(基本值,引用值,集合对象)
2. Bean 对象的自动装配(byName,byType,constructor)
3. Bean 对象常用声明式注解,作用域注解,生命周期注解应用
4. Bean 对象自动装配注解应用.

## 常见FAQ

1. 自动装配中byName和byType有什么不同?
2. Spring表达式在依赖注入中如何使用?
3. Spring 自动装配注解@Autowired和@Resource注解应用有什么不同

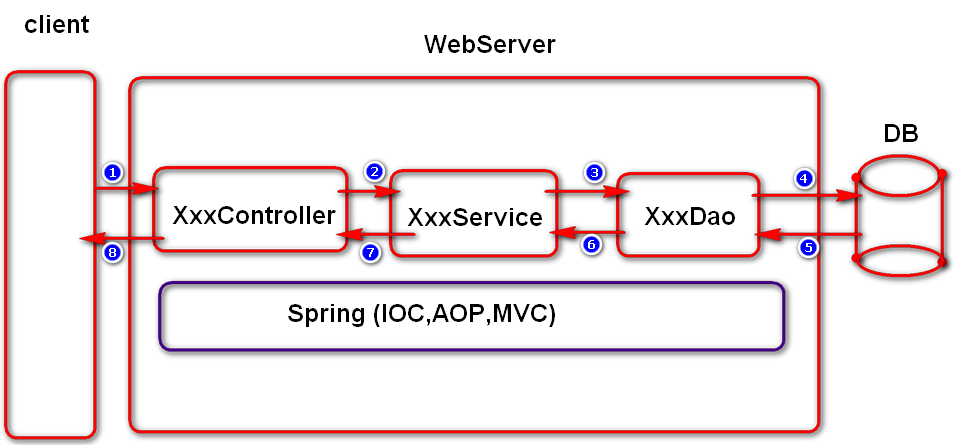
## 作业

1. 总结课堂知识点
2. 完成课堂案例
3. 预习Spring Bean注解应用
4. 了解Lru算法
5. 总结Spring Bean注解基本应用
6. 通过实践总结@Autowired注解与@Resource注解应用的不同

# Spring MVC 概述

## 企业级应用基本架构

企业级应用基本架构（C/S）以及SPRING在架构中的作用。



此架构图中spring要解决什么问题？

1. 对象的构建，对象的依赖管理。(重点是从资源使用角度进行分析)
2. 扩展业务的动态切入。（基于OCP更好实现功能扩展，同时进行解耦）
3. 简化传统web mvc架构中的一些细节处理问题（参数获取，校验，值的注入，响应方式及数据格式的转换）。

问题：

1）我们一般说的解耦是没有耦合吗？不是，所有软件系统中的对象之间

都会存在耦合，只是要把耦合降低。

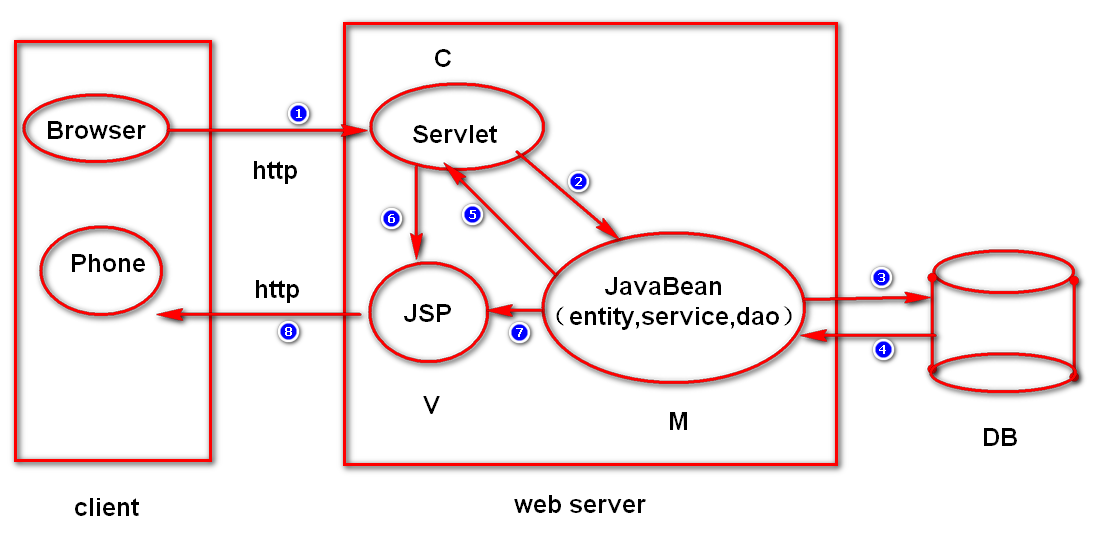
1. 降低耦合的最主要目的是什么？就是要提高系统的可维护性，便于系统进行更好的升级扩展。
2. 降低耦合的主要实现方式？

3.1）对象之间的耦合尽量耦合与接口与工厂

3.2）代码的重用尽量使用组合而非继承。（has a 取代 is a）

## Web MVC架构及分析

基于servlet，jsp，javabean技术实现的MVC架构



问题：

1. 假如我们没有使用spring mvc，你如何对系统采用mvc思想进行分层？（参考jsp+servlet+javabean技术）
2. 传统的web mvc 编程架构中有什么优势，劣势？

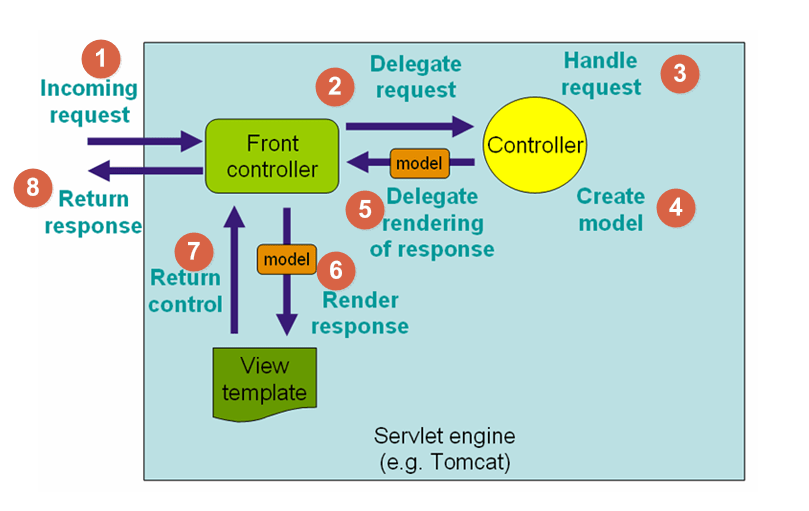
优势：结构层次更加清晰，可读性，可维护性以及可扩展性相对较好

劣势：参数获取，类型转换，流程调用相对都比较复杂

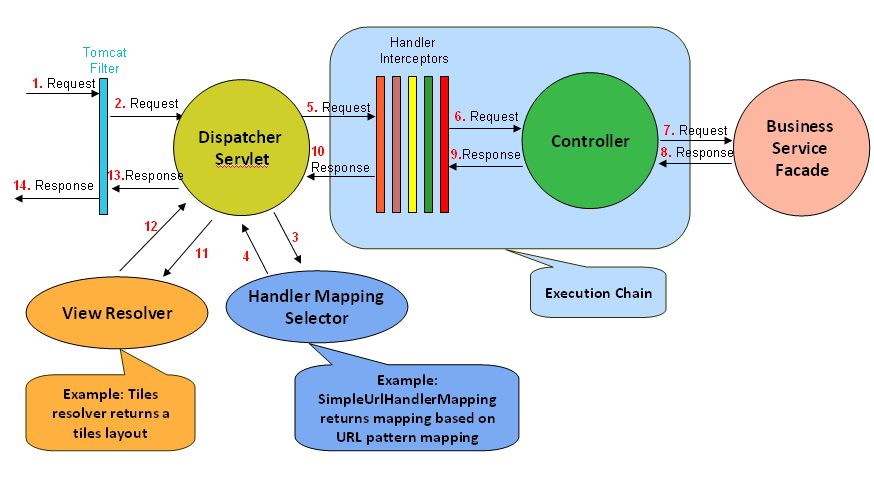
3）市场上相对比较成熟的mvc框架：Spring mvc，struts2，….

## Spring MVC 架构及分析

Spring MVC是MVC架构模式的一种完美实现，它简化了Java WEB 中基于MVC架构的编程过程，是Spring中的WEB应用模块。其官方MVC概要架构图如下：



Spring MVC 底层核心架构图及工作流程（先了解，写完项目案例再重点强化）



Spring MVC 中的核心组件：

1. DispatcherServlet （前端控制器, 处理请求的入口）
2. HandlerMapping （映射器对象, 用于管理url与对应controller的映射关系）
3. Controller （后端控制器, 负责处理请求的控制逻辑）
4. ModelAndView （模型, 封装业务处理结果和视图）
5. ViewResolver（视图解析器,解析对应的视图关系：前缀+view+后缀）

备注：假如希望了解Spring MVC的详细处理流程可以基于断点调试法进行跟踪。

# Spring MVC 编程基础

## 编程基本步骤

Step01：创建maven web 项目并解决项目中的错误问题

Step02：添加项目依赖（spring-webmvc）及spring核心配置文件

Step03：配置前端控制器DispatcherServlet（web.xml）

Step04：创建后端控制器（Controller）及页面

Step05：spring配置文件中配置核心应用组件

Step06：部署及测试springmvc 应用。

## 基于xml编程实现

### 创建Maven WEB 项目

Web项目打包方式为war方式

Web项目的target runtimes为tomcat

Web 项目的编译版本为JDK1.8

可能会遇到的问题：

1. war项目默认不会创建web.xml(需要自己生成)
2. 项目创建好需要设置运行时环境tomcat(多个tomcat时选哪个)
3. 统一编译版本(版本不统一很有可能会出现项目不编译)
4. 统一编码(UTF-8)
5. 假如创建的项目还有问题，可对项目进行clean操作

### 添加Spring MVC 项目依赖及配置文件

打开项目的pom.xml文件，然后添加依赖（选择组groupId为org.springframework）

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

<dependencies>

假如添加好了依赖,都没有找到对应的jar依赖,先检测网络是否是通的.假如网络是通的,还没有下载到具体的依赖,此时要右键项目,选择maven/upate maven project/fore update…进行maven强制更新操作.

在项目的resource的目录中添加核心配置文件applicationContext.xml,例如：

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

<beans default-lazy-init=*"true"*

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

xmlns:p=*"http://www.springframework.org/schema/p"*

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

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

xmlns:tx=*"http://www.springframework.org/schema/tx"*

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

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

xmlns:util=*"http://www.springframework.org/schema/util"*

xmlns:jpa=*"http://www.springframework.org/schema/data/jpa"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.3.xsd*

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

*http://www.springframework.org/schema/aop/spring-aop-4.3.xsd*

*http://www.springframework.org/schema/util*

*http://www.springframework.org/schema/util/spring-util-4.3.xsd*

*http://www.springframework.org/schema/data/jpa*

*http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"* >

</beans>

说明：配置文件的名字需要自己定义，当这个空的配置文件中的某行有相关错误

时，先去检测网络是不是通的。

### 配置前端控制器

打开web.xml，配置DispatcherServlet对象

<servlet>

<servlet-name>dispatcherServlet</servlet-name>

<servlet-class></servlet-class>

<init-param>

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

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

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcherServlet</servlet-name>

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

</servlet-mapping>

前端控制器是spring mvc处理请求的入口，是springmvc的核心，这个控制器一般需要在服务器启动时即初始化。

其中

1. load-on-startup 表示启动时则加载此servlet
2. init-param 中的参数名不能变（此名字在DispatcherServlet父类中定义）

### 创建后端控制器

后端控制器编写时可以实现Controller接口，然后重写handleRequest方法处理请求

**public** **class** HelloController **implements** Controller{

**public** ModelAndView handleRequest(HttpServletRequest request) {

ModelAndView mv=**new** ModelAndView("hello");

mv.addObject("message","helloworld");

**return** mv;

}

}

其中:

ModelAndView对象为一个模型与视图对象，内置一个map对象，主要用于封装业务数据和视图名。

ModelAndView构造方法中传递的为视图名，addObject方法可以以key/value形式存储数据。

ModelAndView 对象返回时会被spring mvc自动存储到请求作用域，在对应的视图页面可以直接从此作用域获取对应的值。

### 创建JSP页面

在项目的WEB-INF/pages文件夹下创建hello.jsp文件，然后设置其内容，例如

…

<body>

<h1>${message}</h1>

</body>

…

### 配置后端控制及视图解析

在spring mvc 核心配置文件中添如下配置。

<!-- 将Controller这个Bean对象交给Spring管理 -->

<bean id=*"helloController"* class=*"spring.controller.XmlHelloController"*>

</bean>

<!-- 配置HandlerMapping 映射处理器配置url到具体的Controller之间的映射-->

<bean class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/hello.do"*>helloController</prop>

</props>

</property>

</bean>

<!-- 配置视图解析器 -->

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

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

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

</bean>

### 部署到tomcat运行

将项目部署到tomcat，然后启动运行，在地址栏输入具体url访问对应controller。

问题：

1. tomcat启动时出现ClassNotFoundException，而这个class又不是我们自己的类，此时要重新maven update，重新发布(右键tomcat 重新publish)，多次尝试还是不可以，此时重启eclipse。
2. 404异常，一般表示服务端资源没找到,首先检测访问路径是否正确，然后还可以在项目的部署目录中去查找对应的资源，必须确保资源是存在的，假如资源不存在，说明代码没有正常编译。（很常见）
3. 如何解决这种项目不编译的问题？

step01) 将tomcat下的项目移除，clean你的tomcat服务器

step02) 将项目的jre从buildpath移除，然后重新添加

step03) 对项目先进行maven clean操作(清除原先编译结构，然后重新编译)

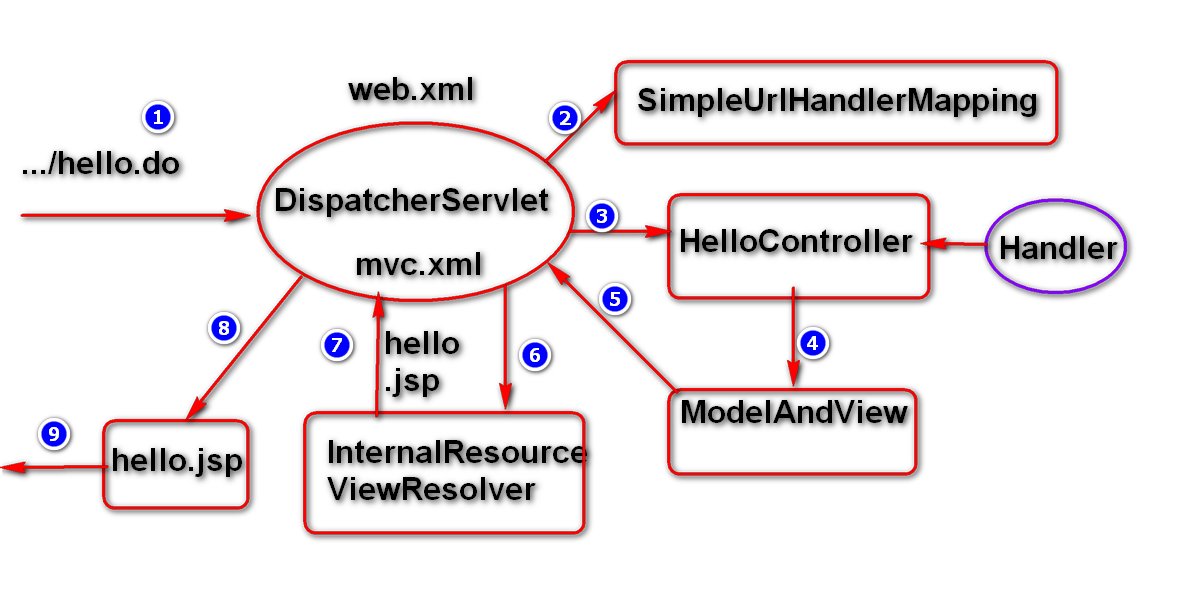
step04) 再次对项目进行clean操作

step05) 重新部署项目，启动tomcat运行

1. 运行项目时尽量不要右键运行选在run as /run on server

### 运行原理分析

本应用的简易处理流程如下：



## 基于注解编程实现

课后预习

### 创建Maven WEB 项目

Maven Web 项目打包方式为war方式

Maven Web 项目的target runtimes为tomcat

Maven Web 项目的编译版本为JDK1.8

Maven WEB 项目的编码方式设置为UTF-8

### 添加Spring MVC 项目依赖及配置文件

打开项目的pom.xml文件，然后添加依赖

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>4.3.9.RELEASE</version>

</dependency>

假如添加好了依赖,都没有找到对应的jar依赖,先检测网络是否是通的.假如网络是通的,还没有下载到具体的依赖,此时要右键项目,选择maven/upate maven project/fore update…进行maven强制更新操作.

在项目的resource的目录中添加核心配置文件applicationContext.xml,例如：

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

<beans default-lazy-init=*"true"*

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

xmlns:p=*"http://www.springframework.org/schema/p"*

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

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

xmlns:tx=*"http://www.springframework.org/schema/tx"*

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

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

xmlns:util=*"http://www.springframework.org/schema/util"*

xmlns:jpa=*"http://www.springframework.org/schema/data/jpa"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.3.xsd*

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

*http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.3.xsd*

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

*http://www.springframework.org/schema/aop/spring-aop-4.3.xsd*

*http://www.springframework.org/schema/util*

*http://www.springframework.org/schema/util/spring-util-4.3.xsd*

*http://www.springframework.org/schema/data/jpa*

*http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd*

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

*http://www.springframework.org/schema/context/spring-context-4.3.xsd"* >

</beans>

### 配置前端控制器

打开web.xml，配置DispatcherServlet对象

<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:applicationContext.xml</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

<servlet-name>dispatcherServlet</servlet-name>

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

</servlet-mapping>

前端控制器是spring mvc处理请求的入口，是springmvc的核心。

### 创建后端控制器

@Controller

@RequestMapping("/")

**public** **class** AnnotationHelloController {

@RequestMapping("doSayHello")

**public** ModelAndView doSayHello() {

ModelAndView mv=**new** ModelAndView("hello");

mv.addObject("message", "helloworld");

**return** mv;

}

@RequestMapping("doSayWelcome")

**public** String doSayWelcome() {

**return** "hello";

}

@RequestMapping("doHandleRequest")

**public** String doHandleRequest(Model m) {

m.addAttribute("message","welcome to beijing");

**return** "hello";

}

}

其中

通过@RequestMapping注解定义url到controller具体方法的映射,这个映射信息会被存储,一般是存储到一个HandlerMapping对象中.

### 配置组件扫描与视图解析

在spring mvc 核心配置文件中添加

<!-- 配置组件扫描 -->

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

<!—启用MVC注解扫描 -->

<mvc:annotation-driven/>

<!-- 配置视图解析器 -->

<bean id=*"viewResolver"*

class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

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

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

</bean>

### 部署与运行

将项目部署到tomcat，然后启动运行，在地址栏输入具体url访问对应controller。

# Spring MVC 编程请求处理进阶

## 请求路径映射

实际项目中我们要借助@RequestMapping注解定义映射路径。其注解应用位置

* 类定义处： 提供初步的请求映射信息。
* 方法定义处： 提供进一步的细分映射信息

### 普通url映射

@RequestMapping(value={"/doSayHello", "/user/doSayWelcome"})：

多个URL路径可以映射到同一个处理器的功能处理方法。

### Rest风格url映射

REST即表述性状态传递（英文：Representational State Transfer，简称REST），是一种软件架构编码风格，是基于网络应用进行设计和开发的编码方式。可以降低开发的复杂度，提高程序的可伸缩性。例如：

@RequestMapping("/msg/{xxx}")

请求的URL可以是“/msg/hello”或“/msg/welcome”

@RequestMapping("/msg/{id}/create")：

请求的URL可以是“/users/1/create”。

@RequestMapping("/msg/{mId}/topics/{tId}")

这样也是可以的，请求的URL可以是“/users/10/topics/12”。

说明：通过@PathVariable可以提取URI模板模式中的{×××}中的×××变量。

<http://localhost:8080/>项目名/doUpdate/1.do

## 请求方式映射

### 请求方式限定

项目中Controller层对象的每个方法默认可以处理任意方式的请求，假如要指定控制层方法只能处理GET或只能处理POST请求，那该如何实现呢？

借助@RequestMapping注解中的method属性指定具体的请求处理方式，例如

@RequestMapping(value=”doSaveObj”,

method=RequestMethod.POST)

public String doSaveObject(Object obj){….}

### 请求方式组合

项目中还可在控制层方法上借助@RequestMapping注解中的method属性指定使用哪几种方式处理请求。

@RequestMapping(value=”doSaveObj”,

method={RequestMethod.POST，

RequestMethod.GET})

public String doSaveObject(Object obj){….}

提示：一般浏览器只支持GET或POST方式。

## 请求参数映射(重点)

### 标准Servlet API

请求映射方法中可以直接使用ServletAPI 中的对象获取参数数据，例如

HttpServletRequest，HttpSession对象等，例如：

@RequestMapping(value="request",method=RequestMethod.***GET***)

@ResponseBody

**public** String withRequest(HttpServletRequest request){

System.***out***.println(request.getRequestURI());

**return** "Obtainer 'foo' query parameter value

'"+request.getParameter("gid")+"'";

}

提示：@ResponseBody注解作用：该注解作用于将Controller的方法返回的对象，通过适当的HttpMessageConverter转换为指定格式后，写入到Response对象的body数据区，使用情况：返回的数据不是Html标签的页面，而是其他数据格式的数据时，（如Json、xml，普通文本等）使用；

### 直接量存储

SpringMVC 请求一个控制层资源时，可以在对应方法中直接使用参数变量接收参数数据，但参数变量的类型要为对象类型。

@RequestMapping(value="param",method=RequestMethod.***GET***)

@ResponseBody

**public** String withParam(@RequestParam String foo) {

**return** "Obtained 'foo' query parameter value '" + foo + "'";

}

提示：@RequestParam注解用于接收请求参数中名字为foo的参数值，假如请求参数名与方法中的参数名一致，@RequestParam注解可以省略。假如不一致则可以使用@RequestParam注解定义新的参数名直接接收页面数据，然后传递给方法名，例如

@RequestMapping(value="param", method=RequestMethod.***GET***)

@ResponseBody

**public** String withParam(

@RequestParam(value="param1",required=**false**) String foo) {

**return** "Obtained 'foo' query parameter value '" + foo + "'";

}

提示：required=false表示，参数可以不存在，假如为true，参数不存在时会抛出异常。

### Java bean对象

当请求中多个参数时可以通过在方法中定义多个参数接收参数数据，也可以利用一个javabean对象接收多个参数数据以简化多个参数变量的定义。

@RequestMapping(value="group",method=RequestMethod.***GET***)

@ResponseBody

**public** String withParamGroup(Message bean) {

**return** "Obtained parameter group " + bean;

}

提示：当使用javabean接收请求参数数据时，bean中需要有与参数名对应的set方法。

### Rest url数据

SpringMVC请求资源路径的URL可以通过{XXX}形式指定动态的URL，动态URL中的这个可变参数的值可以直接注入到方法对应的参数中。

@RequestMapping(value="path/{var}",method=RequestMethod.***GET***)

@ResponseBody

**public** String withPathVariable(@PathVariable String var) {

**return** "Obtained 'var' path variable value '" + var + "'";

}

通过@PathVariable注解指定参数变量var获取请求url中{var}数据

### 请求头数据

当服务端要获取客户端请求头中数据信息时，可通过@RequestHeader即可将请求头中

的属性值绑定到处理方法的入参中，例如获取请求中Accept属性的值，然后传入到对应方法的参数中。

@RequestMapping(value="header", method=RequestMethod.***GET***)

@ResponseBody

**public** String withHeader(@RequestHeader String Accept) {

**return** "Obtained 'Accept' header '" + Accept + "'";

}

假如希望在此方法中直接从cookie取值，可以定义参数时使用@CookieValue对参数进行修饰，参数名一般要与cookie对象中的key相同

@RequestMapping(value="withCookie")

@ResponseBody

**public** String withCooke(@CookieValue String JSESSIONID) {

**return** "Obtained COOKIE Value '" + JSESSIONID + "'";

}

提示：方法中的参数名需要与请求头参数中某个参数名相同，具体请求头相关信息可以在浏览器控制台查看。

当应用中要获取请求中所有数据时可以在请求方法中定义一个HttpEntity<String>参数，通过此参数获取请求头及请求体中数据，例如

@RequestMapping(value="entity", method=RequestMethod.***POST***)

**public** @ResponseBody String withEntity(HttpEntity<String> entity) {

**return** "Posted request body '" + entity.getBody() + "';

headers = " + entity.getHeaders();

}

# Spring MVC 响应处理进阶

## 响应数据封装

### ModelAndView 对象

在对服务端响应数据进行封装时,可以直接在方法参数中定义一个ModelAndView类型的参数,借助ModelAndView对象封装响应数据.

@RequestMapping("doModelAndView")

**public** ModelAndView doModelAndView(ModelAndView mv) {

//ModelAndView mv=new ModelAndView();

mv.addObject("data", "model and view");

mv.setViewName("back");

**return** mv;

}

提示:ModelAndView 对象由Spring创建,并可以将数据存储到ModelAndView对象的ModalMap类型的属性中(可参考源代码).

### Model对象

将响应数据直接封装为model中。

@RequestMapping("doModel")

**public** String doModel(Model model) {

model.addAttribute("data", "modal");

**return** "back";

}

当我们返回具体view时，系统底层会自动将model对象存储到request作用域。

### Map对象

将响应数据封装到Map中（我建议假如使用map对数据进行封装，可直接采用model对象）。

@RequestMapping("doMap")

**public** String doMap(Map<String,Object> map) {

map.put("data", "map..");

**return** "back";

}

说明： 通过map接收页面参数时，需要使用@RequestParam注解声明

@RequestMapping("doMap")

**public** String doMap(@RequestParam Map<String,Object> map) {

map.put("data", "map..");

**return** "back";

}

### Session对象

将响应数据封装到Session,然后可以跨请求获取

@RequestMapping("doSession")

**public** String doMap(HttpSession session) {

session.setAttribute("data", "map..");

**return** "back";

}

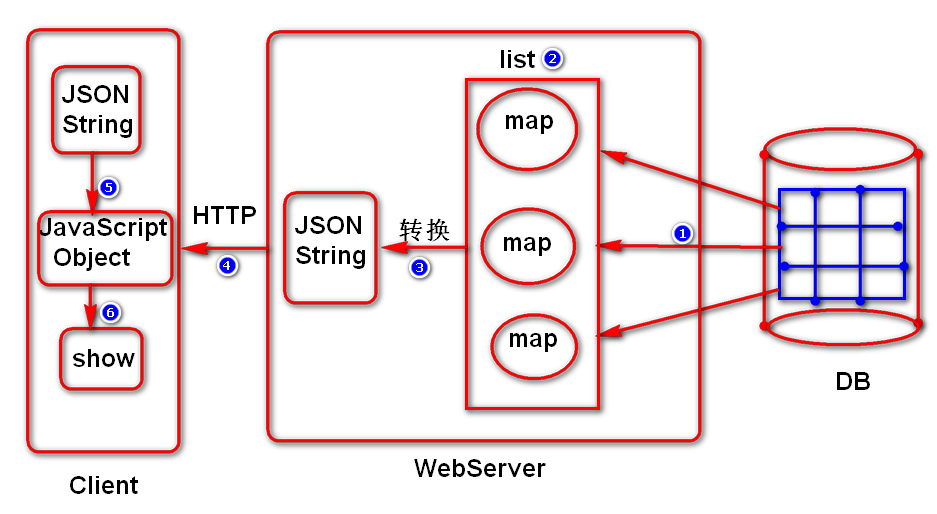
说明返回默认实现是请求转发,假如需要实现重定向,可以在返回的地址后添加redirect,例如 return "redirect:pageUI.do"; 其中pageUI.do对应一个请求url.

## 响应数据转换JSON

### JSON 应用概述

JSON(JavaScript Object Notation):一种轻量级数据交换格式,通常用于实现客户端与服务端之间的数据传输.

企业级Java项目数据传输方式：



### 客户端JSON数据操作

在客户端JSON格式的JavaScript对象:

var o1={id:10,name:’a’,age:20}

var o2=[{id:10,name:’a’,age:20},{id:20,name:’B’,age:30}]

访问json格式的javascript对象中的数据

var id1=o1.id;

var id2=o2[0].id

在客户端JSON格式对象转换为字符串

JSON.parse(json格式的javascript对象)

在客户端将JSON格式的javascript对象转换为json格式的字符串

JSON.stringify(json格式的字符串)

### JAVA端JSON数据操作

在实际的项目中,通常会将服务端返回的数据直接转换JSON格式字符串,在客户端将这个字符串再转换为json格式的javascript对象,然后从这个对象直接取数据.

在将Java对象转换为JSON串时一般首先要添加第三方的JSONAPI 库，通过这样的库可以很方便的实现实现Java对象到JSON串的转换。

服务端常用JSON库：jackson和阿里的fastjson

Jackson 依赖库（SPRING默认支持此依赖库）

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.8.5</version>

</dependency>

Jackson应用案例（jackson应用基础）

**public** **class** TestJackson {

/\*\*实现map与json串之间的转换\*/

@Test

**public** **void** testMap()**throws** Exception{

//1.构建map对象

Map<String,Object> map=

**new** HashMap<String,Object>();

//2.存储数据

map.put("id", 100);

map.put("name","name-a");

map.put("age",20);

//3.将map对象转换为json格式字符串

//3.1构建对象转换器(jackson)

ObjectMapper om=**new** ObjectMapper();

//3.2将map对象转换为json字符串(重要)

String s1=om.writeValueAsString(map);

System.***out***.println(s1);

//假如有需要也可以将json串转换为java对象

Map<?,?> map2=

om.readValue(s1, Map.**class**);

System.***out***.println(map2);

}

@Test

**public** **void** testList()**throws** Exception{

//1构建一个list对象

List<Map<String,Object>> list=

**new** ArrayList<Map<String,Object>>();

//2.构建map对象,并存储数据

Map<String,Object> map=

**new** HashMap<String,Object>();

map.put("id", 100);

map.put("name","AAA");

map.put("age",20);

list.add(map);

map=**new** HashMap<String,Object>();

map.put("id", 101);

map.put("name","DDD");

map.put("age", 30);

list.add(map);

//3.将list对象转换为json格式字符串

//3.1构建对象转换器(jackson)

ObjectMapper om=**new** ObjectMapper();

//3.2将map对象转换为json字符串(重要)

String s1=om.writeValueAsString(list);

System.***out***.println(s1);

}

@Test

**public** **void** testUser()**throws** Exception{

//构建一个java对象

SysUser user=**new** SysUser();

user.setId(10);

user.setUsername("CCCCC");

user.setPassword("123456");

//将java对象转换为字符串

ObjectMapper om=**new** ObjectMapper();

String jsonStr=

om.writeValueAsString(user);

System.***out***.println(jsonStr);

//构建list集合,并存储多个User对象

List<SysUser> list=**new** ArrayList<SysUser>();

list.add(user);

user=**new** SysUser();

user.setId(20);

user.setUsername("MMM");

user.setPassword("456789");

list.add(user);

//将list集合转换为json格式的

jsonStr=om.writeValueAsString(list);

System.***out***.println(jsonStr);

}

}

fastjson依赖库(fastjson的性能相对于jackson更优)

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.38</version>

</dependency>

fastjson 应用案例（学习fastjson的基础）

**public** **class** testFastjson {

@Test

**public** **void** testMap(){

//构建map对象

Map<String,Object> map=

**new** HashMap<String,Object>();

map.put("id", 10);

map.put("name", "AAA");

//将map对象转换为json串

String s1=JSON.toJSONString(map);

System.***out***.println(s1);

//将json串转换为对象

Map<?,?> map2=

JSON.parseObject(s1, Map.**class**);

System.***out***.println(map2);

}

@Test

**public** **void** testUser(){

//构建user对象

SysUser user=**new** SysUser();

user.setId(20);

user.setUsername("CCC");

user.setPassword("123456");

//将user对象转换为json串

String s2=JSON.toJSONString(user);

System.***out***.println(s2);

}

@Test

**public** **void** testList(){

//构建List集合

List<SysUser> list=**new** ArrayList<SysUser>();

//构建user对象

SysUser user=**new** SysUser();

user.setId(20);

user.setUsername("CCC");

user.setPassword("123456");

list.add(user);

user=**new** SysUser();

user.setId(30);

user.setUsername("DDD");

user.setPassword("123456");

list.add(user);

//将user对象转换为json串

String s2=JSON.toJSONString(list);

System.***out***.println(s2);

}

}

### Spring 集成jackson库

spring 中默认支持jackson应用的，但使用时需要添加jackson依赖，例如

<dependency>

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

<artifactId>jackson-databind</artifactId>

<version>2.8.5</version>

</dependency>

创建Controller中例如ResponseController，然后在类中添加对应方法。例如

将Map对象内容转换为字符串（spring底层会直接访问jackson api将对象转换为字符串）

@RequestMapping("doMap")

@ResponseBody

**public** Map<String,Object> doMap(){

Map<String,Object> map=

**new** HashMap<>();

map.put("id", 100);

map.put("name", "AAA");

**return** map;

}

将JavaBean对象转换为JSON串

@RequestMapping("doUser")

@ResponseBody

**public** SysUser doUser(){

SysUser user=**new** SysUser();

user.setId(100);

user.setUsername("CCC");

user.setPassword("123456");

**return** user;

}

将Java List集合转换为JSON串。

@RequestMapping("doList")

@ResponseBody

**public** List<SysUser> doList(){

List<SysUser> list=**new** ArrayList<>();

SysUser user=**new** SysUser();

user.setId(100);

user.setUsername("CCC");

user.setPassword("123456");

list.add(user);

user=**new** SysUser();

user.setId(200);

user.setUsername("DDD");

user.setPassword("123");

list.add(user);

**return** list;

}

备注：将来controller中数据来自服务端数据。

### Spring 集成fastjson库

spring 中默认不支持fastjson api直接将java对象转换为json串，假如需要使用

fastjson 实现对象的转换，现阶段我们需要手动进行转换。

项目中添加fastjson依赖

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.38</version>

</dependency>

创建controller，例如ResponseController，然后在类中添加相关方法，手动将java对象转换为json串。

@RequestMapping("doFastMap")

@ResponseBody

**public** String doFastMap(){

Map<String,Object> map=

**new** HashMap<>();

map.put("id", 100);

map.put("name", "AAA");

**return** JSON.toJSONString(map);

}//{"name":"AAA","id":100}

项目中我们还可以采用指定对象转换器的方式集成fastjson,其步骤为:

Step01: 打开spring核心配置文件

Step02: 添加Bean(FastJsonHttpMessageConverter)的声明

<bean id=*"msgConverter"* class=*"com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter"*>

<property name=*"supportedMediaTypes"*>

<list>

<value>text/html;charset=utf-8</value>

<value>application/json</value>

</list>

</property>

</bean>

Step03: 配置<mvc:message-converters>

<mvc:annotation-driven>

<!-- 设置fastjson的转换器对象 -->

<mvc:message-converters register-defaults=*"true"*>

<ref bean=*"msgConverter"*/>

</mvc:message-converters>

</mvc:annotation-driven>

Step04: 应用fastjson,控制层的方法可以直接返回javabean或对应的map对象

@RequestMapping("doResp02")

@ResponseBody

**public** Map<String,Object> doResp02(){

Map<String,Object> map=

**new** HashMap<String, Object>();

map.put("id", 100);

map.put("name","tmooc");

**return** map;

}

此时方法的返回对象会由FastJsonHttpMessageConverter对象转换为JSON格式的字符串.

# Spring 拦截器应用

## 拦截器概述

拦截器是SpringMVC中的一个核心应用组件,主要用于处理多个

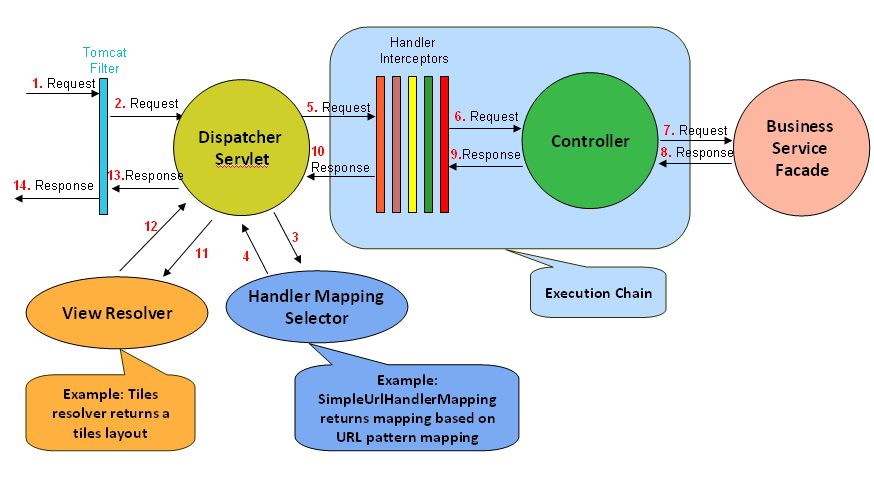
Controller的共性问题.当我们的请求由DispatcherServlet派发

到具体Controller之前首先要执行拦截器中一些相关方法,在这些

方法中可以对请求进行相应预处理(例如权限检测,参数验证),这些方法可以决定对这个请求进行拦截还是放行.

通过spring mvc 架构图分析,拦截器在Spring MVC中处理流程中的一个位置

回顾Spring MVC详细架构图



思考:

1)假如对请求数据进行编码,是应在过滤器还是拦截器?

推荐使用过滤器.

2)拦截器有哪些有些应用场景呢?

a)进行身份认证(判定用户是否是合法用户)

b)进行系统监控

c)进行日志记录

d)..........

## 拦截器编写及基本配置

拦截器如何编写?

我们自己编写Spring MVC拦截器需要实现HandlerInterceptor接口或者继承此接口的实现类 HandlerInterceptorAdapter（继承这个类时可根据需求重写必要的方法）

例如定义一个拦截器，计算controller方法的执行时间

**public** **class** TimeInterceptor **implements** HandlerInterceptor {

@Override

**public** **boolean** preHandle(

HttpServletRequest request,

HttpServletResponse response,

Object handler)

**throws** Exception {

System.***out***.println("time.preHandle");

**long** startTime=System.*nanoTime*();

request.setAttribute("startTime", startTime);

**return** **true**;//true表示放行，false表示拦截

}

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

ModelAndView modelAndView) **throws** Exception {

System.***out***.println("time.postHandle");

}

@Override

**public** **void** afterCompletion(

HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

System.***out***.println("time.afterCompletion");

**long** endTime=System.*nanoTime*();

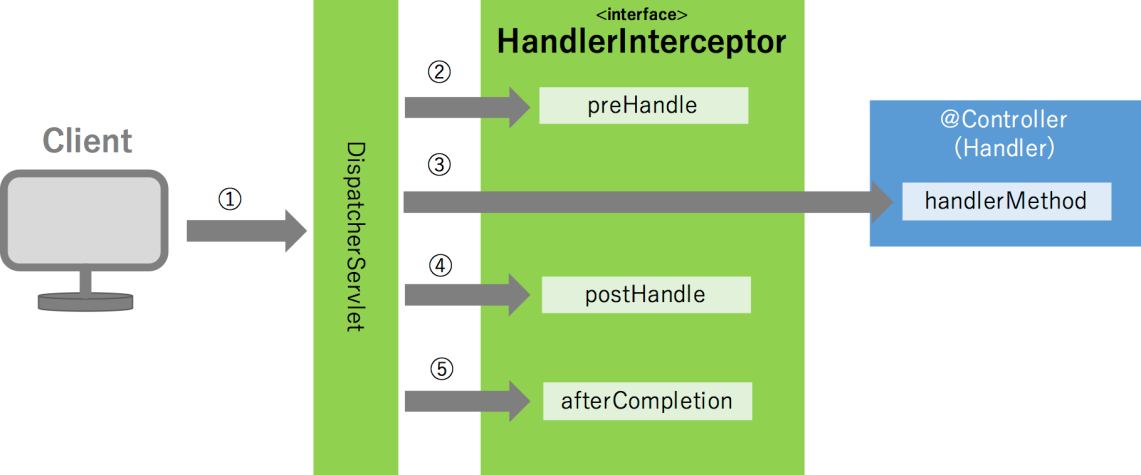
**long** startTime=(Long)request.getAttribute("startTime");

System.***out***.println("time:"+(endTime-startTime));

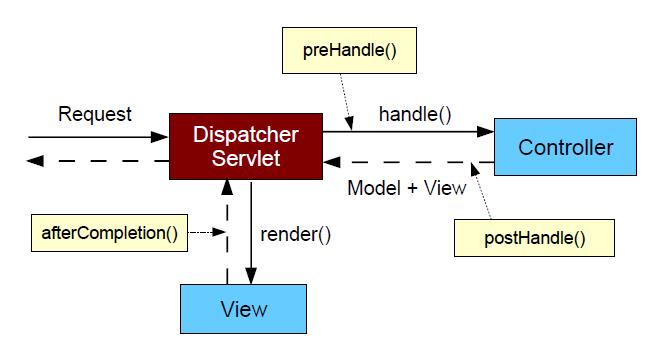
}

}

这些方法的简易执行流程点如下图所示：



这些方法的详细的执行时间点如下：



我们自己的拦截器编写完成以后需要在spring配置文件中进行配置,，

例如:

<mvc:interceptors>

<mvc:interceptor>

<!-- 指定拦截哪些请求路径 -->

<mvc:mapping path=*"/\*\*"*/>

<!-- 排除要拦截的路径 -->

<mvc:exclude-mapping path=*"/user/doLogin.do"*/>

<!-- 拦截器(这个bean也可以通过ref方式引用，前提是在外面要通过xml或注解方式定义) -->

<bean class=*"com.spring.TimeInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

## 拦截器链及配置实现

当我们系统中有多个拦截器时,这些拦截器可以构成一个拦截器链.其原理类似过滤器中的过滤链。在多个拦截器应用中仅当所有匹配的拦截器的preHandle()都执行之后，才会调用Controller中处理请求的方法，然后再执行所有匹配的拦截器的postHandler()，再执行所有匹配的拦截器的afterCompletion()。

在拦截器链中，各拦截器的执行先后顺序取决于配置文件中配置的节点的先后顺序！

例如：再定义一个日志拦截器

@Component

**public** **class** LogInterceptor

**extends** HandlerInterceptorAdapter{

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

System.***out***.println("log.preHandler");

**return** **true**;

}

@Override

**public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

**throws** Exception {

System.***out***.println("log.afterCompletion");

}

}

配置日志拦截器

<!-- 配置拦截器 (要注意标签的编写顺序)-->

<mvc:interceptors>

<!-- 当存在多个拦截器时，哪个拦截器的方法先执行，要看拦截器配置的顺序 -->

<mvc:interceptor>

<mvc:mapping path=*"/user/\*"*/>

<ref bean=*"logInterceptor"*/>

</mvc:interceptor>

<mvc:interceptor>

<!-- 拦截所有请求(这些请求要交给拦截器处理) -->

<mvc:mapping path=*"/\*\*"*/>

<!-- 定制对哪些请求不拦截 (这些请求不需要拦截处理)-->

<mvc:exclude-mapping path=*"/user/doLogin.do"*/>

<!-- 拦截器对象 -->

<bean class=*"com.spring.TimeInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

具体执行顺序跟配置顺序有关,例如现有两个拦截器A,B,按顺序配置,其执行顺序为

1)A.preHandle

2)B.preHandle

3)执行controller中具体方法

4)B.postHandle

5)A.postHandle

6)B.afterCompletion

7)A.afterCompletion

执行顺序总结:先执行的后结束

思考:

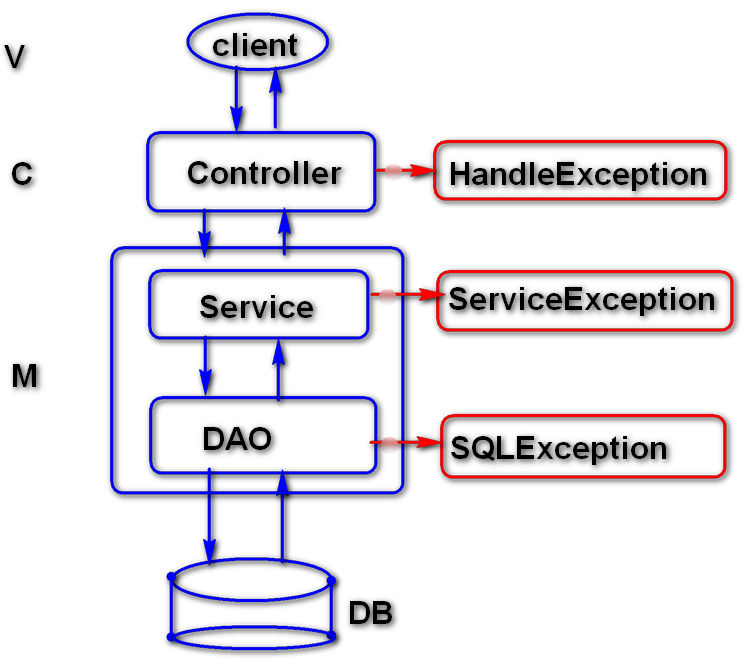
1. 多个拦截器时,其执行顺序有什么决定?(配置顺序)
2. 拦截器对象何时创建?(启动时创建)
3. 拦截器对象在内存中有几份?(一份)
4. 拦截器中存在变量共享时可能会有线程安全问题吗?(有)
5. 多个拦截器对象如何共享数据?(通过request,session,ServletContext,)
6. 拦截器对象的生命周期何时结束?(容器销毁时)
7. 拦截器可以解决什么问题?(Controller中共性问题,例如日志处理,系统监控,...)
8. ......

# Spring MVC异常处理

## Spring MVC 异常概述

实际项目中我们经常会采用分层架构设计程序,每一层都可能会有异常,假如异常信息没有处理,可能会选择抛出,假如这些被抛出的异常与具体业务相关,那到控制层以后我们一般都进行相应的处理(处理方式应该相对友好)

在Spring mvc 项目中,边界出现异常以后我们通常会在Controller中进行处理,常用分层架构中的异常分析:



在分层架构中，异常可能会从DAO层逐步抛出到控制层，可以在控制层对异常进行相关处理。

## Spring MVC 异常处理

在spring中处理异常时,通常会在Controller中定义具体的异常处理方法,这个方法上使用@HandlerException注解进行描述.例如在指定Controller中定义异常处理方法:

@ExceptionHandler(value=Exception.**class**)

@ResponseBody

**public** String handleException(Exception e){

System.***out***.println("局部异常处理");

**return** e.getMessage();

}

当多个异常处理类中很多异常处理的方式是相同的，还可以将这个异常处理方法提取到父类，以实现方法的重用。

假如不希望将共性的异常处理方法提取到父类，还可以在外部定义一个全局的异常处理类,这个类使用@ControllerAdvice注解进行修饰.然后在这个类中定义具体的异常处理方法,这些方法再使用@HandlerExcpeiton进行修饰,例如

@ControllerAdvice

**public** **class** AdviceExceptionHandler {

@ExceptionHandler(Throwable.**class**)

@ResponseBody

**public** String handlerException(Throwable e){

System.***out***.println("全局的异常处理");

**return** e.getMessage();

}

}

Spring MVC中异常处理机制总结：

1. 为什么要进行异常处理？(提高系统的容错能力，让系统表现的更加友好)
2. Spring MVC中异常处理的方式？（局部的，全局的）
3. 局部方式异常处理如何实现？(Controller类中直接定义异常处理方法，使用@ExceptionHandler进行修饰)
4. 全局方式异常处理如何实现？(通过继承将共性提取到父类，通过@ControllerAdvice注解修饰类，然后类中定义异常处理方法)
5. 出现异常后，异常处理的具体机制是怎样的（首先会在当前Controller（包含父类）中查找有没有定义异常处理方法，假如有则查看注解中定义的异常类型是否与当前异常类型相匹配，假如匹配则直接处理，假如不匹配则查看当前类中有没有父类类型的异常处理方法与当前异常匹配，假如有则直接处理，没有则找全局异常处理。全局也没有则直接不友好的输出错误信息）

说明：实际项目中在controller的方法中除了要对正确的业务数据进行封装之外，还会对异常信息进行封装(例如封装到JsonResult)，然后转换为json格式的字符串，输出到客户端。

# Spring MVC 文件上传(课后扩展)

## 文件上传业务场景

在电子商务系统中对具体商品采用图片方式进行描述，这个图片需要上传到服务器等等。

## Spring MVC 文件上传基本步骤

Step01: 添加文件上传相关依赖（jar包：commons-fileupload）

Step02: 配置对上传文件对象的解析(spring的核心配置文件)

Step03: 编写文件上传表单(请求方式为post,enctype必须为multipart/form-data)

Step04: 编写对应的控制层对象处理文件上传(通过mutilpartFile参数接收文件)

### 添加依赖

<dependency>

<groupId>commons-fileupload</groupId>

<artifactId>commons-fileupload</artifactId>

<version>1.3.3</version>

</dependency>

### 配置对象解析

在spring的核心配置文件中添加对象解析

<bean id=*"multipartResolver"*

class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 配置文件编码处理 -->

<property name=*"defaultEncoding"* value=*"utf-8"*/>

<!-- 配置文件大小限制 -->

<property name=*"maxUploadSize"* value=*"4096000000"*/>

<!-- 配置内存缓冲区大小 -->

<property name=*"maxInMemorySize"* value=*"1024000"*/>

</bean>

说明：配置对象解析时，bean的id一定要指定，而且官方规定必须为*multipartResolver*

### 定义upload.jsp

在WEB-INF/pages目录下创建一个upload.jsp,然后创建上传表单

<body>

<h1>文件上传</h1>

<h1>${msg}</h1>

<form action=*"doUpload.do"*

method=*"post"*

enctype=*"multipart/form-data"*>

<input type=*"file"* name=*"upfile"*/>

<br/>

<input type=*"submit"* value=*"upload"*>

</form>

</body>

说明：定义表单时，提交方式必须为post，enctype必须为multipart/form-data

### 定义UploadController

@RequestMapping("/upload/")

@Controller

**public** **class** UploadController {

@RequestMapping("uploadUI")

**public** String uploadUI(){

**return** "upload";

}

@RequestMapping("doUpload")

**public** ModelAndView doUpload(MultipartFile upfile)

**throws** IOException{

//获取文件名以及文件大小，检测是否获得文件相关数据

String fileName=upfile.getOriginalFilename();

**long** size=upfile.getSize();

System.***out***.println(fileName+"/"+size);

//构建文件目标对象，这个对象对应的文件路径必须是存在的或者通过file对象自己创建

File dest=**new** File("D:/SSMWORK/"+fileName);

//transferto实现文件上传

upfile.transferTo(dest);

//封装数据返回

ModelAndView mv=**new** ModelAndView("upload");

mv.addObject("msg", "upload ok");

**return** mv;

}

}

说明：在控制层方法接收请求中file类型的数据时需要借助MultipartFile 类型的参数对象，其参数名要求与表单中type为file的参数名相同。

# MyBatis 概述

## MyBatis 是什么?

1. 软件框架(FrameWork): 用于解决软件中的通用型问题
2. 软件中持久层框架(数据访问层):用于更好解决数据持久化问题
3. 由apache的ibatis演变而来.

课后了解:

hibernate (ORM 框架)

## MyBatis 架构体系?

对于任何一个持久层框架,都应该具备接口服务,数据处理服务,基础服务等相关功能,MyBatis也不例外,它会具体如下几个层结构.

1. 接口应用层(对外提供服务)
2. 数据查询
3. 数据修改
4. 数据删除
5. 数据插入
6. 数据处理层(处理数据访问问题)
7. SQL参数映射(Dao方法参数与映射文件中#{}表达式映射)
8. SQL解析(语法,语义) 例如: select \* from blog where id=#{id}
9. SQL 执行(将sql发送到数据库端执行)
10. SQL 结果映射(例如将ResultSet中的数据存到map)
11. ....
12. 基础服务层
13. 连接服务(配置连接池,)
14. 事务服务
15. 缓存服务
16. 配置服务(别名配置,映射配置,...,日志配置,....)
17. .....

## MyBatis 核心组件及API?

MyBatis 核心应用组件:

1. 配置文件(提供基础配置信息,例如连接配置,缓存配置,映射配置)
2. 映射文件(定义SQL映射)
3. SqlSessionFactoryBuilder (负责读取配置文件,创建SqlSessionFactory对象)
4. SqlSessionFactory(负责创建SqlSession对象)
5. SqlSession(负责连接的维护,事务的处理,类似JDBC中的Connection)
6. .....

思考:

1)核心组件在应用的一个角色定位?

2)应用底层会用到哪些设计模式?建造模式,工厂模式

# MyBatis 编程基础

## MyBatis 基本步骤

Step01:创建maven 桌面项目(Java 项目)

Step02:添加mybatis依赖以及mysql驱动依赖

Step03:创建mybatis 配置文件,映射文件

Step04:配置数据访问(配置文件),SQL映射(映射文件)

Step05:创建MyBatis API(例如SqlSession)对象,执行SQL操作.

思考?

1. 数据库连不上,可能存在哪些问题?
2. 检测url,用户名,密码
3. 检测端口号(port)
4. id地址是否能ping 通
5. 检测驱动程序(有可能依赖的驱动程序与数据库版本不兼容)
6. mybatis 配置文件的名字有要求吗?(只要符合标识符规范即可)
7. mybatis 配置文件是要对哪些信息进行配置呢?
8. mybatis 映射文件中主要用于定义哪些内容?(sql元素)
9. MyBatis 执行SQL操作时的一个基本过程是怎样的?
10. step01:MyBatis API调用JDBC API
11. step02:JDBC API 调用数据库驱动程序API

## MyBatis 编程实现

### 数据准备

创建数据库,并在库中创建表

create database cgb1711 character set utf8;

打开数据库

use cgb1711;

创建表

create table blog(

id int primary key auto\_increment,

title varchar(200) not null,

content varchar(500) not null,

createdTime date

) engine=innoDB;

向表中写入数据

insert into blog values (null,'ta','ta...',now());

insert into blog values (null,'tb','tb...',now());

### 创建Maven桌面项目

创建maven 桌面项目,并添加依赖(mybatis,mysql驱动)

<!-- 添加mybatis依赖 -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<!-- 添加mysql驱动依赖 -->

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.40</version>

</dependency>

<!-- 添加junit依赖 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

思考:

1. 添加依赖时注意groupId的选择?(选正规的)
2. 添加依赖以后pom.xml文件假如有错什么原因? 检测网络,检测maven配置(setting.xml)

### 添加配置及映射文件

在src/main/resources目录下创建配置文件mybatis-configs.xml,内容如下:

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 配置初始化环境(连接) -->

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"*/>

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"jdbc:mysql:///cgb1711"*/>

<property name=*"username"* value=*"root"*/>

<property name=*"password"* value=*"root"*/>

</dataSource>

</environment>

</environments>

<!-- 配置mapper文件路径 -->

<mappers>

<mapper resource=*"mapper/BlogMapper.xml"*/>

</mappers>

</configuration>

说明:

1)配置文件的头可从官方文档进行拷贝.

2)配置文件元素没有提示是什么原因? 对应的dtd文件取不到

step02:在src/main/resources/mapper目录下创建映射文件BlogMapper.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.jt.blog.BlogDao"*>

</mapper>

说明:

1. 映射文件的命名空间用于约束元素id的唯一性.
2. 映射文件的格式最好是包结构形式
3. 映射文件内部可以定义很多元素,每个元素必须有一个唯一id,例如select

在mapper文件中添加如下元素:

查询所有blog数据

<select id=*"findBlogs"* resultType=*"map"*>

select \* from blog

</select>

根据id查询blog数据

<select id=*"findBlogById"* resultType=*"map"*>

select \*

from blog

where id=#{id}

</select>

限制查询blog元素

<select id=*"findPageBlogs"*

resultType=*"map"*>

select \*

from blog

limit #{array[0]},#{array[1]}

</select>

向表中插入数据元素定义

<insert id=*"insertObject"*>

insert into blog

(id,title,content,createdTime)

values

(null,#{array[0]},#{array[1]},now())

</insert>

修改表中数据元素定义

<update id=*"updateObject"*>

update blog

set title=#{array[0]},

content=#{array[1]}

where id=#{array[2]}

</update>

删除表中元素元素定义

<delete id=*"deleteObject"*>

delete

from blog

where id=#{id}

</delete>

### 编写代码执行查询测试

基于BlogMapper.xml文件中元素的定义,添加测试类及相关方法.

编写测试类并添加测试方法

**public** **class** TestBlog01 {

**private** SqlSessionFactory factory;

}

测试类中添加初始化factory的方法

@Before

**public** **void** init()**throws** IOException{

//初始化SqlSessionFactory

factory=**new** SqlSessionFactoryBuilder()

.build(Resources.*getResourceAsStream*(

"mybatis-configs.xml"));

}

测试类中添加查询所有blogs的方法

@Test

**public** **void** testFindBlogs(){

//1.创建SqlSession对象(相当于创建一个连接)

SqlSession session=factory.openSession();

//2.执行查询操作(selectList("命名空间"+"元素id"))

List<Object> list=session.selectList(

"com.jt.blog.BlogDao.findBlogs");

**for**(Object o:list){

System.***out***.println(o);

}

//3.释放资源(类似关闭连接)

session.close();

}

测试类中添加根据id执行查询方法

@Test

**public** **void** testFindBlogById(){

//1.创建session对象

SqlSession session=

factory.openSession();

//2.执行sql操作

String statement="com.jt.blog.BlogDao.findBlogById";

Map<String,Object> map=

session.selectOne(statement,1);

System.***out***.println(map);

//3.释放资源(关闭session对象)

session.close();

}

测试类中添加分页查询方法

@Test

**public** **void** testFindPageBlogs(){

//1.创建session对象

SqlSession session=

factory.openSession();

//2.执行sql操作?????

String statement="com.jt.blog.BlogDao.findPageBlogs";

Object parameter=**new** Object[]{0,4};

List<Map<?,?>> list=

session.selectList(statement, parameter);

**for**(Map<?,?>map:list){

System.***out***.println(map);

}

//3.释放资源(关闭session对象)

session.close();

};

测试类中添加插入数据的方法

@Test

**public** **void** testInsertObject(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

String statement="com.jt.blog.BlogDao.insertObject";

Object parameter=**new** Object[]{"te","te..."};

**int** rows=session.insert(statement, parameter);

System.***out***.println("insert.rows="+rows);

session.commit();

//3.关闭session

session.close();

}

测试类中添加修改方法

@Test

**public** **void** testUpdateObject(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

String statement="com.jt.blog.BlogDao.updateObject";

Object parameter=**new** Object[]{"taa","taa...",1};

**int** rows=session.update(statement, parameter);

System.***out***.println("update.rows="+rows);

session.commit();

//3.关闭session

session.close();

}

测试类中添加删除方法

@Test

**public** **void** testDeleteObject(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

String statement="com.jt.blog.BlogDao.deleteObject";

Object parameter=7;

**int** rows=session.delete(statement, parameter);

session.commit();

System.***out***.println("delete.rows="+rows);

//3.关闭session

session.close();

}

# MyBatis 编程进阶

本小节中会从mybatis编程的另一个角度(例如基于接口方式等)实现对数据库中数据的操作.

## 基本步骤

Step01: 创建maven桌面项目并添加依赖

Step02: 创建配置文件config.propertis(内容为数据库相关)

Step03: 创建mybatis核心配置文件mybatis-configs.xml文件

Step04: 配置Mybatis基础数据服务(properties,datasource,mapper)

Step05: 创建映射文件BlogMapper.xml

Step06: 创建实体类Blog(与表对应,可用于封装表中数据)

Step07: 创建BlogDao接口,并添加相关方法.

Step08: 配置BlogMapper映射文件,添加相关元素.

Step09: 基于BlogDao接口与映射文件实现CRUD操作

## 编程实现

### 创建Maven桌面项目

创建maven桌面项目并添加依赖

<!-- 添加mybatis依赖 -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>3.2.8</version>

</dependency>

<!-- 添加mysql驱动依赖 -->

<dependency>

<groupId>mysql</groupId>

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

<version>5.1.40</version>

</dependency>

<!-- 添加junit依赖 -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

### 创建config.properties文件

在src/main/resource目录下创建config.properties文件,此文件中定义

系统中的一些常用配置信息,例如访问数据库的相关信息,其内容如下

driver=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/cgb1711

username=root

password=root

### 创建mybatis核心配置文件

在src/main/resources目录下创建mybatis-configs.xml文件,并配置数据源等相关信息,数据信息从config.properties文件读取.

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

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!--配置properties文件-->

<properties resource=*"config.properties"*/>

<!-- 配置初始化环境(连接) -->

<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>

</configuration>

### 创建Blog实体对象

创建Blog类实现与数据库中Blog表实现映射.

**public** **class** Blog {

**private** Integer id;

**private** String title;

**private** String content;

**private** Date createdTime;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** String getContent() {

**return** content;

}

**public** **void** setContent(String content) {

**this**.content = content;

}

**public** Date getCreatedTime() {

**return** createdTime;

}

**public** **void** setCreatedTime(Date createdTime) {

**this**.createdTime = createdTime;

}

@Override

**public** String toString() {

**return** "Blog [id=" + id + ", title=" + title + ", content=" + content + ", createdTime=" + createdTime + "]";

}

}

### 创建BlogDao接口

创建数据访问接口,并添加相关方法

**package** com.jt.dao;

**import** java.util.List;

**import** org.apache.ibatis.annotations.Param;

**import** com.jt.entity.Blog;

**public** **interface** BlogDao {

/\*\*\*

\* 根据id进行对象查找

\* **@param** id

\* **@return**

\*/

Blog findBlogById(Integer id);

List<Blog> findPageBlogs(

@Param("offset")Integer offset,

@Param("pageSize")Integer pageSize);

**int** insertObject(Blog blog);

**int** updateObject(Blog blog);

**int** deleteObject(Integer id);

}

### 创建BlogMapper映射文件

<?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.jt.dao.BlogDao"*>

</mapper>

BlogMapper文件中添加与BlogDao接口对应的映射元素

添加基于ID进行查询的元素

<select id=*"findBlogById"*

parameterType=*"int"*

resultType=*"blog"*>

select \*

from blog

where id=#{id}

</select>

添加分页查询元素

<select id=*"findPageBlogs"*

resultType=*"blog"*>

select \*

from blog

limit #{offset},#{pageSize}

</select>

添加insert元素

<insert id=*"insertObject"*

parameterType=*"blog"*>

insert into blog

(id,title,content,createdTime)

values

(null,#{title},#{content},now())

</insert>

添加更新元素

<update id=*"updateObject"*

parameterType=*"blog"*>

update blog

set title=#{title},content=#{content}

where id=#{id}

</update>

添加删除元素

<delete id=*"deleteObject"*

parameterType=*"int"*>

delete from blog where id=#{id}

</delete>

最后在mybatis-configs.xml中添加BlogMapper文件

### 创建单元测试类执行测试

创建单元测试类,并添加相关方法实现基于Dao接口方式的数据库操作.

**public** **class** TestBlog01 {

**private** SqlSessionFactory factory;

@Before

**public** **void** init()**throws** IOException{

factory=**new** SqlSessionFactoryBuilder()

.build(Resources.*getResourceAsStream*(

"mybatis-configs.xml"));

}

@Test

**public** **void** testFindBlogById(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

//2.1获取dao对象

BlogDao dao=

session.getMapper(BlogDao.**class**);

//2.2执行dao中方法

Blog blog=dao.findBlogById(1);

System.***out***.println(blog);

//3.关闭session

session.close();

}

@Test

**public** **void** testFindPageBlogs(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

//2.1获取dao对象

BlogDao dao=

session.getMapper(BlogDao.**class**);

//2.2执行dao中方法

List<Blog> list=dao.findPageBlogs(0, 2);

**for**(Blog b:list){

System.***out***.println(b);

}

//3.关闭session

session.close();

}

@Test

**public** **void** testInsertObject(){

//1.创建session

SqlSession session=factory.openSession();

//2.执行sql

//2.1获取dao对象

BlogDao dao=

session.getMapper(BlogDao.**class**);

//2.2执行dao中方法

Blog blog=**new** Blog();

blog.setTitle("te");

blog.setContent("te...");

dao.insertObject(blog);

session.commit();

//3.关闭session

session.close();

}

}

## 业务增强

### 动态排序策略

1. 如何按照用户指定业务对数据进行排序?

例如:

1. 博客系统
2. 按照创建时间对博客信息进行排序查询
3. 按照博客访问量对信息进行排序查询
4. 电商系统
5. 按照商品价格对商品信息排序
6. 按照商品销量对商品信息排序
7. 按照商品好评对商品信息排序

解决方案:

1. 写多个sql映射
2. 写一个sql 映射,然后动态传参,借助${}表达式获取参数

例如

接口中方法定义

List<Blog> findBlogs(

@Param("column") String column,

@Param("seq") String seq);

映射文件中的实现

<select id=*"findBlogs"*

resultType=*"blog"*>

select \*

from blog

order by ${column} ${seq}

</select>

Test

@Test

public void findBlogs(){

SqlSession session = factory.openSession();

BlogDao dao = session.getMapper(BlogDao.class);

List<Blog> blog = dao.findBlogs("id", "asc");

for(Blog b: blog){

System.out.println(b);

}

}

思考:

MyBatis中$与#有什么不同?

${}表达式主要用于获取配置文件数据,DAO接口中的参数信息,当$出现在映射文件的SQL语句中时创建的不是预编译的SQL,而是字符串的拼接,有可能会导致SQL注入问题.所以一般使用$接收dao参数时,这些参数一般是字段名,表名等,例如order by {column}.

#{}表达式主要用户获取DAO中的参数数据,在映射文件的SQL语句中出现#{}表达式,底层会创建预编译的SQL.性能会相对较好.

${}获取DAO参数数据时,参数必须使用@param注解进行修饰.

#{}获取DAO参数数据时,假如参数个数多于一个可有选择的使用@param.

### ID应用策略

1.保存数据时获取数据在数据库对应的主键值?

例如

1. 订单系统?

保存订单信息时候,获取订单在数据库中的主键id值?

1. 权限系统

保存用户信息时,获取用户信息在数据库中的主键值?

解决方案: 当对象对应的表中的记录为自增长时,可以采用如下方案

<insert id=*"insertObject"*

parameterType=*"blog"*

useGeneratedKeys=*"true"*

keyProperty=*"id"*>

insert into blog

(id,title,content,createdTime)

values

(null,#{title},#{content},now())

</insert>

其中:keyProperty属性用于指定参数中的id属性.

1. 当多线程并发的向表中写入数据时,假如id使用自增长可能存在线程安全问题?

例如:

1. 秒杀系统
2. 订票系统
3. ....

解决方案:可将自增长的id设置为随机数,当然有些数据库根本就不支持自增长,此时也可以选择随机数.

数据准备:创建Author表

create table Author(

id varchar(200) primary key,

username varchar(100) unique not null,

password varchar(300) not null,

email varchar(50) unique

)engine=InnoDB;

借助mybatis 应用向表中写入数据,主键值要求通过UUID生成.其映射文件参考

<insert id=*"insertObject"*

parameterType=*"author"*>

<selectKey keyProperty=*"id"*

resultType=*"string"*

order=*"BEFORE"*>

select replace(uuid(),'-','')

</selectKey>

insert into author

(id,username,password,email)

values

(#{id},#{username},#{password},#{email})

</insert>

如果要随机写select uuid() 即可

# MyBatis 高级应用

## 日志配置应用

Mybatis内置的日志工厂提供日志功能，具体的日志实现有以下几种工具：

1)SLF4J(日志框架标准,类似JDBC标准)

2)Apache Commons Logging

3)Log4j 2 (是log4j的升级版,配置文件升级为xml格式了)

4)Log4j(日志处理库,配置文件格式为.properties)

5)JDK logging

项目中mybatis通常会借助第三方日志库进行日志的处理,例如log4j.

配置步骤:(以log4j为例)

1. 添加log4j依赖(一代)

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

1. 添加log4j.properties 配置(可以从其它项目中拷贝)

log4j.rootLogger=INFO,stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

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

log4j.appender.stdout.layout.ConversionPattern=%d [%-5p] %c - %m%n

log4j.logger.com.mybatis3=DEBUG

log4j.logger.com.jt=DEBUG

1. 设置mybatis的日志实现(mybatis-configs.xml)

<settings>

<setting name=*"logImpl"* value=*"log4j"*/>

</settings>

其中name属性的值为固定写法,value的值要依托于使用的日志处理库.

说明:课后了解常用的日志处理库.

## 缓存配置应用

1. 缓存是什么? 内存中的一个对象(容器).
2. 缓存对象的作用?提高程序的性能(最主要的的是访问效率)
3. MyBatis 中缓存概述?

MyBatis 框架中提供了非常强大的缓存特性来提高查询性能,通常可分为一级缓存(SqlSession级别)和二级缓(SqlSessionFactory).

1. MyBatis 中的缓存应用.

MyBatis中一级缓存默认是开启的.不需要任何配置.例如:

@Test

**public** **void** testFindObjects(){

SqlSession session=factory.openSession();

AuthorDao dao=

session.getMapper(AuthorDao.**class**);

//==============

List<Author> list=

dao.findObjects();

session.close();

session=factory.openSession();

dao=session.getMapper(AuthorDao.**class**);

list=dao.findObjects();

//==============

session.close();

}

MyBatis 二级缓存默认是没有开启的,需要在映射文件中加上<Cache/>元素

MyBatis 二级缓存应用步骤:

Step01: 修改mybatis核心配置文件,添加缓存设置.

<settings>

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

Step02: 在映射文件中配置Cache策略.

<cache

eviction=*"LRU"*

flushInterval=*"60000"*

size=*"512"*

readOnly=*"true"*/>

这个表示创建了一个 LRU缓存,并每隔 60 秒刷新,存数结果对象或列表的 512 个引用,而且返回的对象被认为是只读的,因此在不同线程中的调用者之间修改它们会 导致冲突。其中：

1) eviction 表示回收策略(例如LRU,FIFO等，默认为LRU)

2) flushInterval 表示刷新间隔

3) size(引用数目)可以被设置为任意正整数,要记住你缓存的对象数目和你运行环境的 可用内存资源数目。默认值是 1024。

4) readOnly(只读)属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。可读写的缓存会返回缓存对象的拷贝(通过序列化) 。这会慢一些,但是安全,因此默认是 false。

Step03: 使用二级缓存了.

@Test

**public** **void** testFindObjectById(){

SqlSession session=factory.openSession();

AuthorDao dao=session.getMapper(AuthorDao.**class**);

String id="c6dad9d622a311e88271408d5c85e6b9";

Author a1=dao.findObjectById(id);

System.***out***.println("a1="+a1);

session.close();

session=factory.openSession();

dao=session.getMapper(AuthorDao.**class**);

Author a2=dao.findObjectById(id);

a2.setUsername("tedu-10");

System.***out***.println("a2="+a2);

//System.out.println(a1==a2);

session.close();

session=factory.openSession();

dao=session.getMapper(AuthorDao.**class**);

Author a3=dao.findObjectById(id);

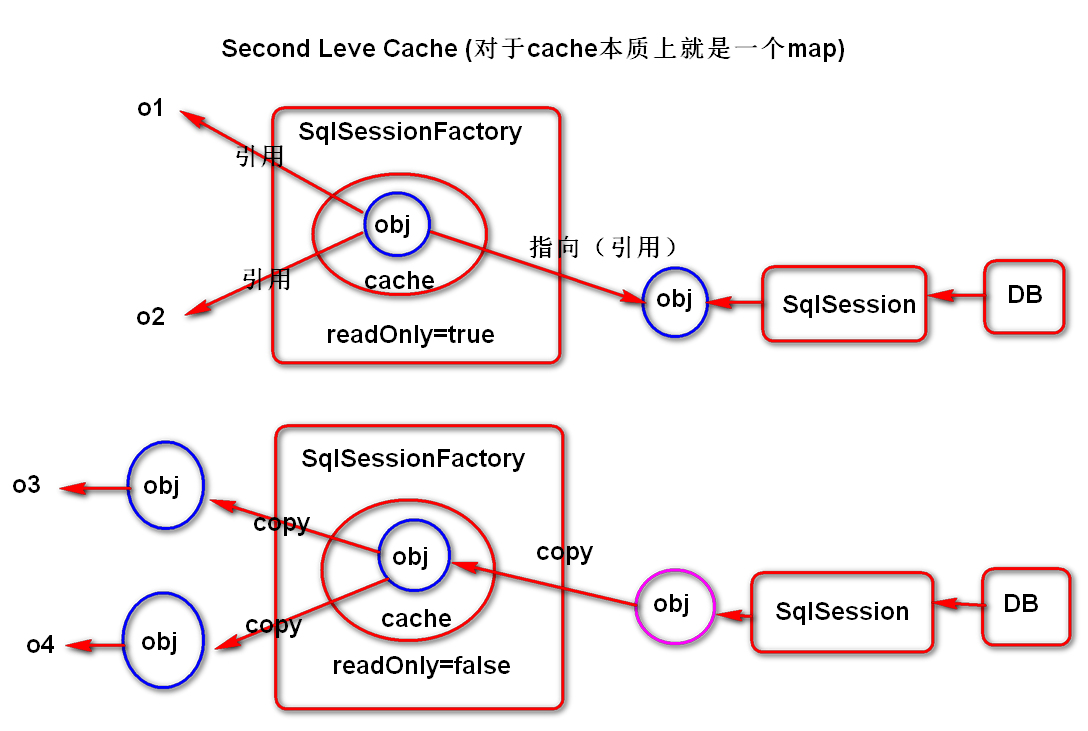
System.***out***.println("a3="+a3);

session.close();

}

FAQ?

1)MyBatis 二级缓存cache元素readOnly值的设置以及底层对象缓存过程.



readOnly说明:

readOnly的值为true时,缓存中保存的是堆内存中对象的引用.每次从缓存取数据都是获得的同一个对象.readOnly为false时,首先会将查询到的对象,拷贝到缓存一份(对象需要实现序列化接口),然后从缓存取数据每次都是执行对象的拷贝.

## 高级映射应用

MyBatis中的高级映射一般要借助select元素中的resultMap属性进行实现,通过此属性配置实现一对一,一对多等关系映射的实现.

1. 如何实现one2one映射?
2. 如何实现one2many映射?
3. 如何实现many2many映射?

思考:

1. 查询博客信息时,将作者信息也查询出来?(many2one)
2. 查询某个用户的同时,将其对应的博客列表信息也查询出来(one2many)
3. .....

课后作业:(尝试完成)

1. 在blog表中添加一个字段,名字为authorId,类型为string.
2. 为blog表中的authorId字段赋值,其值为author表中某些记录的主键值.

alter blog author add authorId varchar(200),

不做硬连接，否则不能修改。

Update blog set authorId=’ 7bbaec2c229d11e89dc73497f652ad45’

Where id=1;

1. 查询某个blog信息,并将其关联的author对象信息查询出来.

Dao中：

第一个方案：

Map<String,Object> findBlogAuthorById(Integer id);

(接口中可以放大括号吗？旧版本不可以，新版本可以！）

映射文件：

<select id=”findBlogAuthorById”

resultType=”map”>

Select from blog b.\*,a.username,a.email (这里位置的key不能重名，因为map里的key特点：key相同，值覆盖）

join author a on b.authorId = a.id

Where b.id=#{id}

TestBLog.java

Map<String,Object> map = Dao.findBlogAuthorById(1);

Sysout(map);

反思：看新的东西，看不明白会去试试，看英文文档，和看翻译（翻译很多很别扭，不准确）；看个大概可以实践，实践加强理解。

不能只是等，看，抄，这是惰性。

第二个方案：

Blog.java中加入：

有些企业会：Private Author author;(直接写一个对象进来，也可以写一个VO--value object值对象，记得写set,get，toString）,放入BLogResult.java中

这里新建一个包：com.jt.vo

这些改动放：BlogVO

或者：正规店的方法

BlogDao.java

Blog

BlogMapper.xml

<resultMap id=”blogResult” type=”com.jt.vo.BLog”>

<id property=”id” column=”id” />

<result property=”title” column=”title” />

<result property=”content” column=”content” />

<result property=”createdTime” column=”createdTime” />

<result property=”author” column=”author” />

<id property=”id” column=”id” />

## 延迟加载应用

对关联查询中的关联对象数据进行延迟加载(何时需要何时加载).以提高内存的使用效率.

具体如何配置?(参考官方文档)

## 动态SQL应用

如何动态删除客户端选择的记录,用户有可能会选择一个记录,也可能会选择多个记录然后进行删除.那么在mybatis的映射文件中该如何配置,如何实现或者说解决这种动态需求问题?

课后了解:动态sql.

# 总结

## 重点和难点分析

1. MyBatis 是什么,要解决什么问题?
2. MyBatis 底层架构体系?
3. MyBatis 核心组件以及API?
4. MyBatis 编程的基本步骤?
5. MyBatis 数据操作执行流程?
6. ...

## 常见FAQ

1. MyBatis 底层数据方法需要JDBC吗?
2. MyBatis 核心配置文件中有哪些常用配置?
3. MyBatis 映射文件中常用元素定义有哪些?
4. MyBatis 数据操作常用实现方式有哪些?
5. MyBatis 映射文件元素如何获取参数数据?
6. MyBatis 映射文件元素中获取参数数据的常用方案.
7. ...

## 作业

1. 总结课堂知识点
2. 完成课堂案例
3. 编写基于接口方式的更新以及删除操作