# Mybatis:

mybatis:开源的轻量级的ORM持久化框架,本质就是封装了jdbc的操作

1.ORM:对象关系映射

2.jdbc操作数据库的问题

3.#{}和${}的区别

4.Javabean,pojo 还有一些配置问题

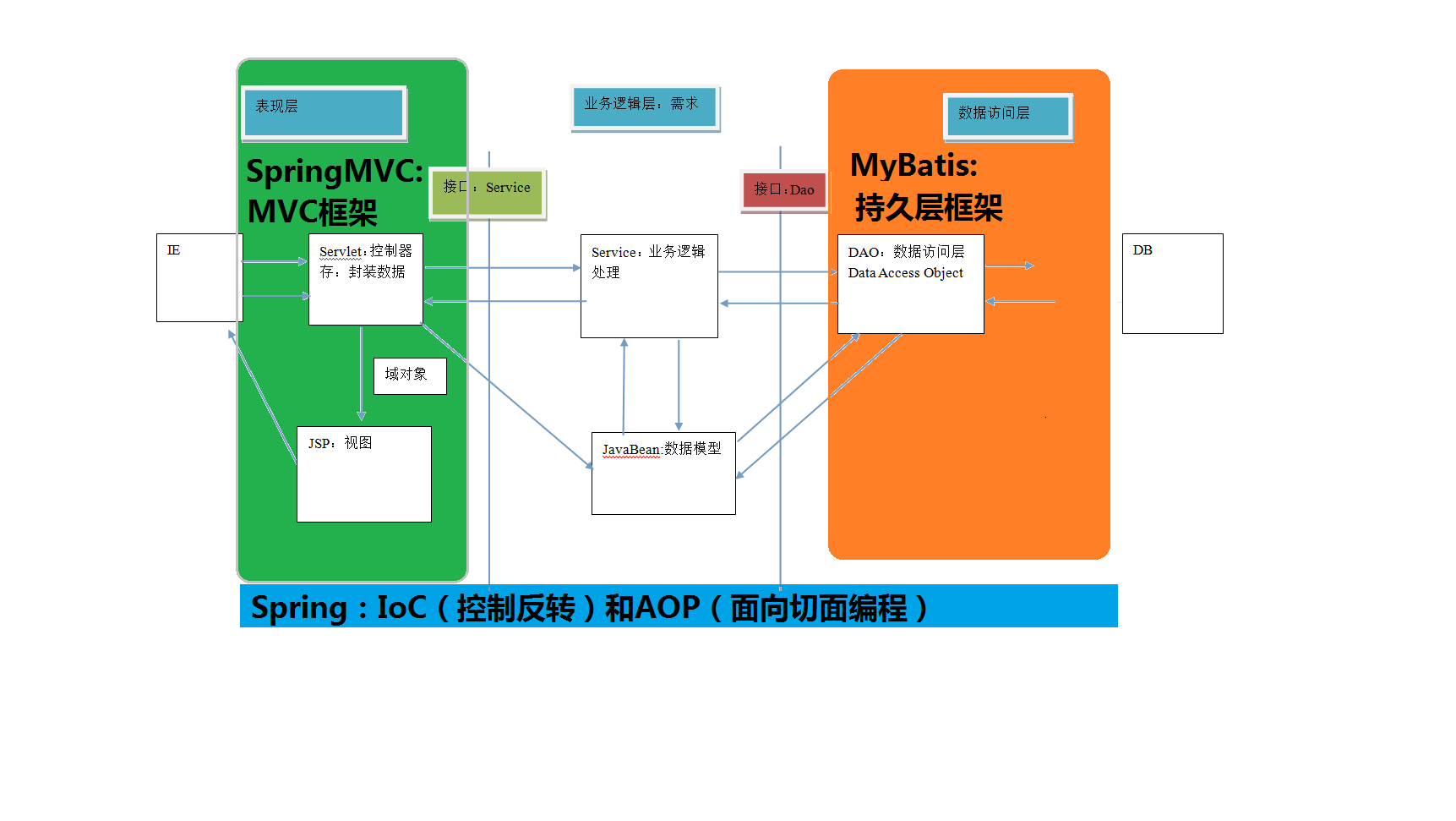
5,连接池

6,延迟加载,立即加载

7,一级缓存(SqlSession),二级缓存(SqlSessionFactory)

## day1

### SSM与三层架构的对应关系



Servlet+Jsp+JavaBean

JavaBean:封装数据的规范:

1. 私有化属性
2. 提供公共的get/set方法
3. 有一个公共的无参构造器
4. 实现序列化接口
5. [equals,hashCode,toString]

POJO:简单Java类

不实现接口,不继承别的类

### JDBC操作数据库的问题

* 1. 没有使用数据库连接池,连接池每次使用都是直接创建,用完就销毁,效率低
  2. Sql语句不能动态添加条件,是硬编码,后期维护麻烦.
  3. 连接参数,sql语句需要的参数,都是硬编码,维护,麻烦
  4. 查询的结果集,需要手动封装,也是硬编码,维护麻烦

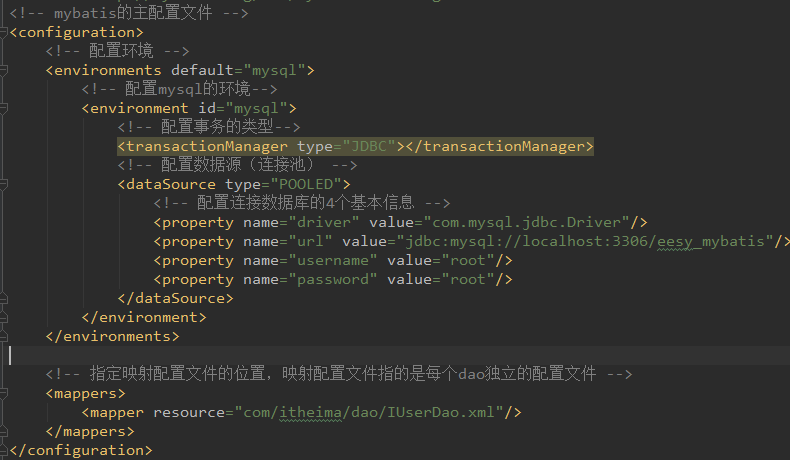
### Mybatis框架

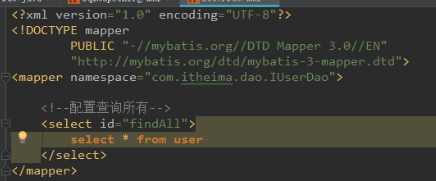
* 1. 开源的轻量级的ORM持久化框架,Java写的
  2. 本质就是封装了JDBC的操作
  3. ORM对象关系映射:
     1. 将Java类与数据库表对应
     2. 将类中的属性与数据库中的字段对应
     3. 查询出表中的一条数据,将会封装成类的一个对象



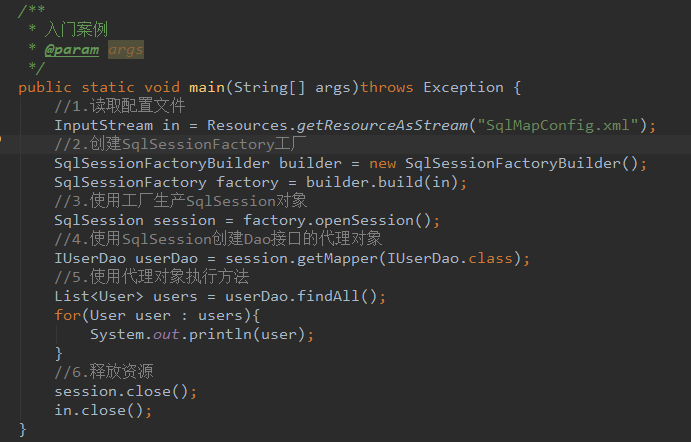
### 开发步骤:

* 1. 环境搭建:
     1. Pom.xml添加依赖
     2. 创建表
     3. 创建与表对应的实体类
     4. 创建SqlMapConfig.xml 的配置文件(数据库连接参数,mapper映射文件)
     5. 创建Mapper映射文件(SQL语句,输入映射,输出映射)
     6. 创建一个dao接口





* 1. 编写Java代码:
     1. 创建一个测试类
     2. 加载配置文件
     3. 创建SqlSessionFactoryBuilder
     4. build一个SqlSessionFactory
     5. Open一个SqlSession
     6. SqlSession去getMapper(类型)//获取代理对象
     7. 代理对象执行目标对象的方法
     8. 释放资源

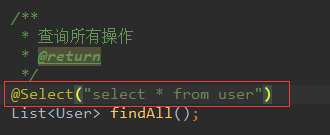


注意:Mapper映射文件与Dao接口之间的对应关系

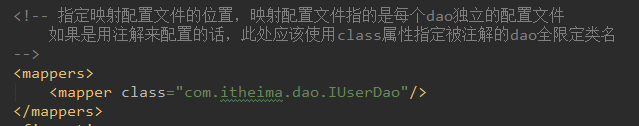
1. mapper中的namespace值与接口的全限定名相同
2. Mapper中的id与方法名相同
3. Mapper中的resultType与方法的返回值类型相同
4. Mapper中的parameterType与方法的参数类型相同
   1. 注解开发

删除IUserDao.xml映射文件

在IUserDao接口中的查询所有方法上加入注解



Mybais的主配置文件中，读取配置文件的方式需要改成如下



练习:入门案例,CRUD

## day2

如果表中的字段与实体类的属性名字不相同

* 1. 改成相同
  2. 在查询时取别名
  3. 定义resultMap去声明表中的字段与实体类中的属性的对应关系,然后在查询语句的标签中,就不再使用resultType而用resultMap

### 主配置文件中的其他配置:

<properties resource="配置文件"

作用:抽取配置相关信息到properties配置文件中,可以更加方便管理,一般抽取数据库连接参数

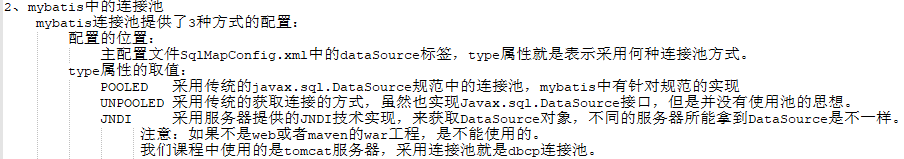
<typeAliases>

作用:配置实体类的别名,配置映射文件时,不用写全限定名可以使用配置的别名不区分大小写

<typeAlias type="com.itheima.domain.User" alias="user"></typeAlias>-->  
<!-- 用于指定要配置别名的包，当指定之后，该包下的实体类都会注册别名，并且类名就是别名，不再区分大小写-->  
 <package name="com.itheima.domain"></package>

### 连接池(存放连接的容器)

* 1. POOLED
  2. UNPOOLED
  3. JNDI



连接池配置:<**dataSource type="POOLED"**>

### Mybatis中的事务:实际上底层connection.commit()

默认事务是不会自动提交的,设置**sqlSession** = factory.openSession(**true**);

注意:每次对数据库的增删改都会开启一个单独的事务.

### 动态Sql语句

* 1. If

<**select id="findUserByCondition" resultMap="userMap" parameterType="user"**>  
 select *\** from user where 3=3  
 <**if test="userName != null "**>

//test中写的是条件(注意:userName是**parameterType中的user的属性**)  
 and username = #{userName}  
 </**if**>  
 <**if test="userSex != null"**>  
 and sex = #{userSex}  
 </**if**>  
</**select**>

* 1. Where

<**select id="findUserByCondition" resultMap="userMap" parameterType="user"**>  
 select *\** from user

<**where**>//省略 where 1=1,自动判断条件是不是第一个,需不需要加and

<**if test="userName != null "**>

//test中写的是条件(注意:userName是**parameterType中的user的属性**)  
 and username = #{userName}  
 </**if**>  
 <**if test="userSex != null"**>  
 and sex = #{userSex}  
 </**if**>

</**where**>  
</**select**>

* 1. Foreach

<**foreach collection="ids" open="and id in (" close=")" item="uid" separator=","**>  
 #{uid}  
</**foreach**>

<foreach>标签用于遍历集合，它的属性：

collection:代表要遍历的集合元素，注意编写时不要写#{}

open:代表语句的开始部分

close:代表结束部分

item:代表遍历集合的每个元素，生成的变量名

separator:代表分隔符,两次循环之间拼接separator里面的内容

select 字段from user where id in (41,42,46)

* 1. Sql

*<!-- 了解的内容：抽取重复的sql片段-->*<**sql id="defaultUser"**>  
 select *\** from user  
</**sql**>

<**select id="findAll" resultMap="userMap"**>  
 <**include refid="defaultUser"**></**include**>  
</**select**>

### 多表查询

1.关系型数据库表之间的关系:一对一,一对多,多对多,多对一

2.Mybatis完成一堆多,与多对一的查询

步骤：

1、建立两张表：用户表，账户表

让用户表和账户表之间具备一对多的关系：需要使用外键在账户表中添加

2、建立两个实体类：用户实体类和账户实体类

让用户和账户的实体类能体现出来一对多的关系

1)用户类中需要有一个List

*//一对多关系映射：主表实体应该包含从表实体的集合引用* **private** List<Account> **accounts**;

1. 账户类中需要有一个User

*//从表实体应该包含一个主表实体的对象引用* **private** User **user**;

3、建立两个配置文件

用户的配置文件

*<!-- 定义User的resultMap-->*<**resultMap id="userAccountMap" type="user"**>  
 <**id property="id" column="id"**></**id**>  
 <**result property="username" column="username"**></**result**>  
 <**result property="address" column="address"**></**result**>  
 <**result property="sex" column="sex"**></**result**>  
 <**result property="birthday" column="birthday"**></**result**>  
 *<!-- 配置user对象中accounts集合的映射 -->* <**collection property="accounts" ofType="account"**>  
 <**id column="aid" property="id"**></**id**>  
 <**result column="uid" property="uid"**></**result**>  
 <**result column="money" property="money"**></**result**>  
 </**collection**>  
</**resultMap**>

账户的配置文件

*<!-- 定义封装account和user的resultMap -->*<**resultMap id="accountUserMap" type="account"**>  
 <**id property="id" column="aid"**></**id**>  
 <**result property="uid" column="uid"**></**result**>  
 <**result property="money" column="money"**></**result**>  
 *<!-- 一对一的关系映射：配置封装user的内容-->* <**association property="user" column="uid" javaType="user"**>  
 <**id property="id" column="id"**></**id**>  
 <**result column="username" property="username"**></**result**>  
 <**result column="address" property="address"**></**result**>  
 <**result column="sex" property="sex"**></**result**>  
 <**result column="birthday" property="birthday"**></**result**>  
 </**association**>  
</**resultMap**>

4、实现配置：

当我们查询用户时，可以同时得到用户下所包含的账户信息

当我们查询账户时，可以同时得到账户的所属用户信息

## day3

### 多对多

示例：用户和角色

一个用户可以有多个角色

一个角色可以赋予多个用户

步骤：

1、建立两张表：用户表，角色表

让用户表和角色表具有多对多的关系。需要使用中间表，中间表中包含各自的主键，在中间表中是外键。

2、建立两个实体类：用户实体类和角色实体类

让用户和角色的实体类能体现出来多对多的关系

各自包含对方一个集合引用

*//多对多的关系映射：一个角色可以赋予多个用户* **private** List<User> **users**;

*//多对多的关系映射：一个用户可以具备多个角色* **private** List<Role> **roles**;

3、建立两个配置文件

用户的配置文件

*<!-- 定义User的resultMap-->*<**resultMap id="userMap" type="user"**>  
 <**id property="id" column="id"**></**id**>  
 <**result property="username" column="username"**></**result**>  
 <**result property="address" column="address"**></**result**>  
 <**result property="sex" column="sex"**></**result**>  
 <**result property="birthday" column="birthday"**></**result**>  
 *<!-- 配置角色集合的映射 -->* <**collection property="roles" ofType="role"**>  
 <**id property="roleId" column="rid"**></**id**>  
 <**result property="roleName" column="role\_name"**></**result**>  
 <**result property="roleDesc" column="role\_desc"**></**result**>  
 </**collection**>  
</**resultMap**>

角色的配置文件

*<!--定义role表的ResultMap-->*<**resultMap id="roleMap" type="role"**>  
 <**id property="roleId" column="rid"**></**id**>  
 <**result property="roleName" column="role\_name"**></**result**>  
 <**result property="roleDesc" column="role\_desc"**></**result**>  
 <**collection property="users" ofType="user"**>  
 <**id column="id" property="id"**></**id**>  
 <**result column="username" property="username"**></**result**>  
 <**result column="address" property="address"**></**result**>  
 <**result column="sex" property="sex"**></**result**>  
 <**result column="birthday" property="birthday"**></**result**>  
 </**collection**>  
</**resultMap**>

4、实现配置：

当我们查询用户时，可以同时得到用户所包含的角色信息

当我们查询角色时，可以同时得到角色的所赋予的用户信息

### 延迟加载

在多表关联查询时,是否应该一次性将所有的数据全部查询出来?

1. 延迟加载:用的时候再去查

一般用于一对多时,比如查询用户时,不需要将用户所有的账户查询出来,当要使用账户时再去查询(否则浪费资源-->因为没有用到账户,但是查出来了)

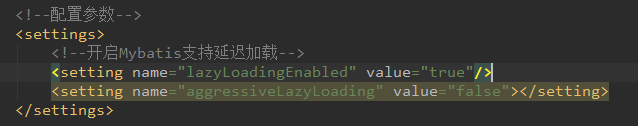
1. 立即加载(Mybatis默认):Dao调用方法时直接将所有关联数据查询

一般用于一对一时,比如查询账户时,我们需要将用户也查询出

实际操作按照以后的业务需求

延迟加载操作步骤:

1.在主配置文件中添加标签，开启延迟加载



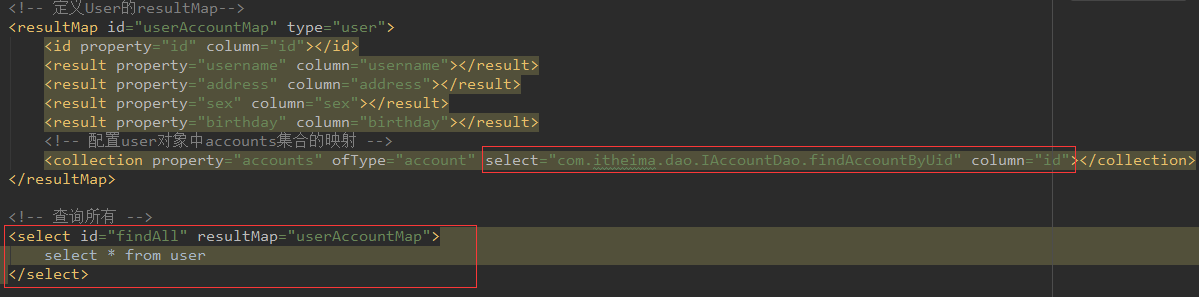
2修改IAccountDao映射文件

2.1association要添加select属性,他的值是对应的类的dao接口的全限定名.方法名(根据id查找的方法名)

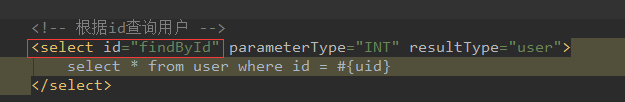
2.2sql语句要修改成只查询一个表的数据



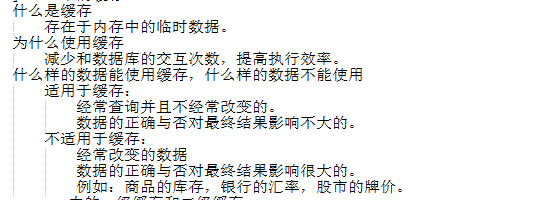
Column:外键,两个表之间关联的字段



3.启动select指向IUserDao中的映射文件，如下标红的部分



### 缓存



Mybatis中的缓存:

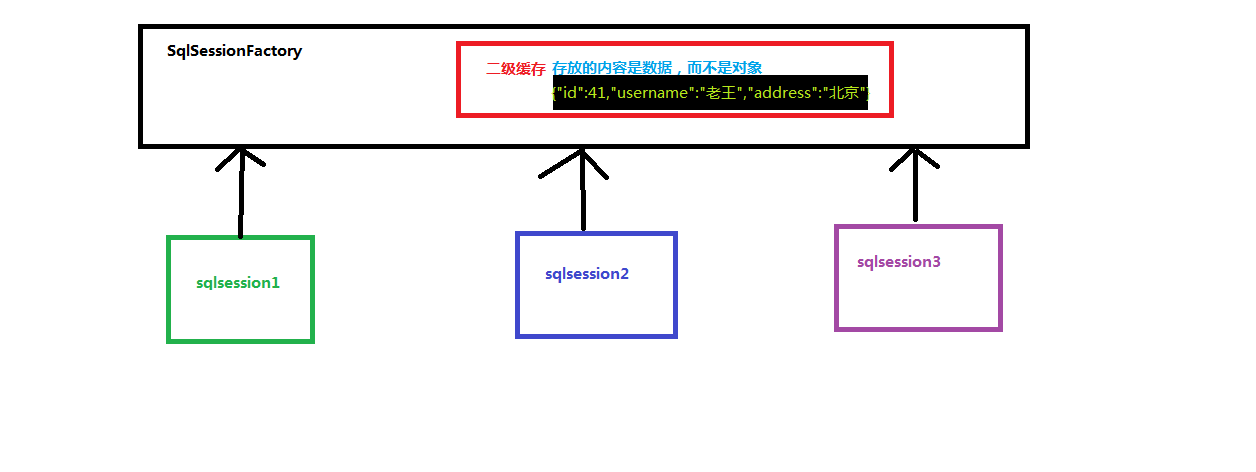
1. 一级缓存(SqlSession缓存):存在SqlSession中,默认启用

同一个SqlSession拥有相同的缓存,在查询数据时,如果缓存中有那么直接将缓存中的数据拿出,如果缓存中没有,那么去数据库中查询,并且存如缓存中.

注意:有一些条件是可以清空缓存的

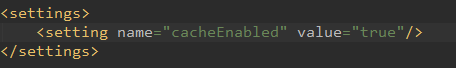
一级缓存是 SqlSession 范围的缓存，当调用 SqlSession 的修改，添加，删除，commit()，close(),clearCache()等方法时，就会清空一级缓存。

1. 二级缓存(SqlSessionFactory缓存):同一个Factory开启的SqlSession共享这个缓存,二级缓存的开启需要配置



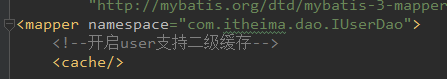
二级缓存的使用步骤：

第一步：让Mybatis框架支持二级缓存（在SqlMapConfig.xml中配置）



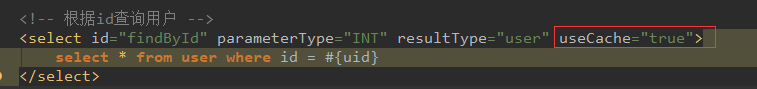
第二步：让当前的映射文件支持二级缓存（在IUserDao.xml中配置）

让映射文件对应的Dao支持二级缓存



第三步：让当前的操作支持二级缓存（在select标签中配置）

让dao的某个方法支持从二级缓存中取值



### 注解开发

主要掌握,这个注解代替了哪个配置.

1. Select()//查询
2. Insert()//插入
3. Update()//修改
4. Delete()//删除
5. Results()//替代映射文件中的ResultMap



一对一&一对多

<**association property="user" column="uid" select="com.itheima.dao.IUserDao.findById"**></**association**>

select=”该属性对应关联表的dao的全限定名.根据外键查询的方法名”并且声明外键

**@Result(property = "user",column = "uid",one=@One(select="com.itheima.dao.IUserDao.findById",fetchType= FetchType.*EAGER*))**

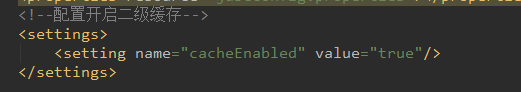
One(一对一)&many(一对多)

Select代替的就是xml中的select

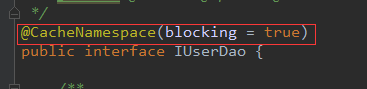
fetchType:是否延迟加载 LAZY(延迟)|EAGER(立即)|DEFAULT(默认)

二级缓存的使用：

修改主配置文件，开启二级缓存



在接口中添加一个注解开启二级缓存：



# Spring:

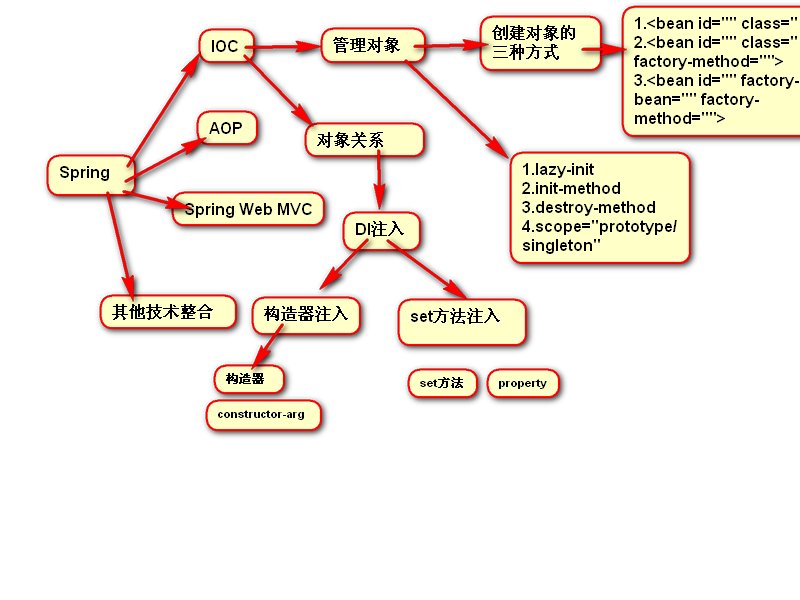
1.什么是Spring?

Spring是轻量级的开源的一站式框架

2.Spring能干什么?

帮我们进行解耦(包含类与类之间的解耦IOC,方法内部结构AOP)

3.Spring的结构?



## Day1

### Spring的核心是什么?

### IOC是什么?有什么用?

### DI是什么?与IOC有什么关系?

### BeanFactory与ApplicationContext的区别?

### ApplicationContext的三个实现类(通常ClassPathXmlApplicationContext)

### Bean的创建方式?

### Bean的作用域有哪些?

### Bean的生命周期是什么?

### Bean的注入方式有哪些?

## Day2

### 1.开启注解扫描:

*<!--告知spring在创建容器时要扫描的包，配置所需要的标签不是在beans的约束中，而是一个名称为context名称空间和约束中-->*<**context:component-scan base-package="com.itheima"**></**context:component-scan**>

*<!-- 扫描com.itheima包下所有类,包括子包 -->*

### 2.注解:

曾经XML的配置：  
 <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl"  
 scope="" init-method="" destroy-method="">  
 <property name="" value="" | ref=""></property>  
 </bean>

#### A.用于创建对象的

他们的作用就和在XML配置文件中编写一个<bean>标签实现的功能是一样的  
  **@Component:**  
 作用：用于把当前类对象存入spring容器中  
 属性：  
 value：用于指定bean的id。当我们不写时，它的默认值是当前类名，且首字母改小写。  
 **@Controller**：一般用在表现层servlet  
  **@Service**：一般用在业务层service  
 **@Repository**：一般用在持久层dao  
 以上三个注解他们的作用和属性与Component是一模一样。  
 他们三个是spring框架为我们提供明确的三层使用的注解，使我们的三层对象更加清晰

#### B.用于注入数据的

他们的作用就和在xml配置文件中的bean标签中写一个<property>标签的作用是一样的  
 **@Autowired:**  
 作用：自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功

注意:

1.如果ioc容器中没有任何bean的类型和要注入的变量类型匹配，则报错(required错误)。

2.如果Ioc容器中有多个类型匹配:按照变量名与容器中的满足类型相同的key进行匹配,如果没有匹配上则报错(found 2),如果匹配上了,则注入匹配上的这个对象

3.如果容器中有且仅有一个该类型的对象,则直接注入

4.默认情况下要求依赖对象必须存在，如果要允许null值，可以设置它的required属性为false

出现位置：  
 可以是变量上，也可以是(get/set)方法上  
 细节：  
 在使用注解注入时，set方法就不是必须的了。  
 **@Qualifier(“”):**  
 作用：在按照类中注入的基础之上再按照名称注入。它在给类成员注入时不能单独使用。但是在给方法参数注入时可以  
 属性：  
 value：用于指定注入bean的id。  
  **@Resource:(jdk1.8)**  
 作用：和 @Autowired差不多,只不过@Resource是默认先用byname, 如果找不到合适的就再用bytype来注入  
 属性：  
 name：用于指定bean的id。

type:用于指定bean的类型.  
 以上三个注入都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。  
 另外，集合类型的注入只能通过XML来实现。  
   
  **@Value(“${jdbc.driver}”):**  
 作用：用于注入基本类型和String类型的数据  
 属性：  
 value：用于指定数据的值。它可以使用spring中SpEL(也就是spring的el表达式）  
 SpEL的写法：#{表达式}  
 ${}:EL表达式

* 1. 写在Mybatis的配置文件中,那么就是字符串拼接
  2. 写在JSP中,从四大域中取值(pageContext,Request,Session,Application)

$():Jquery$(选择器)

#{}:Mybatis中OGNL表达式

#{}:Spring中spEL表达式语法

#### **C.用于改变作用范围的**

他们的作用就和在bean标签中使用scope属性实现的功能是一样的  
 **@Scope(“prototype”):**  
 作用：用于指定bean的作用范围  
 属性：  
 value：指定范围的取值。常用取值：singleton prototype

#### D.和生命周期相关 了解

他们的作用就和在bean标签中使用init-method和destroy-methode的作用是一样的  
  **@PreDestroy**  
 作用：用于指定销毁方法(单例对象销毁才会被Spring调用)  
  **@PostConstruct**  
 作用：用于指定初始化方法

### 3.spring的新注解:

**@Configuration**

作用：指定当前类是一个配置类

细节：当配置类作为AnnotationConfigApplicationContext对象创建的参数时，该注解可以不写。

**@ComponentScan("com.itheima")**

作用：用于通过注解指定spring在创建容器时要扫描的包

属性：

value：它和basePackages的作用是一样的，都是用于指定创建容器时要扫描的包。

我们使用此注解就等同于在xml中配置了:

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

**@Bean(name="ds2"):**

如果有一个类是别人提供的.class文件,那么这时我们不能给这个类加注解,提供一个方法,在方法内手动的去new该类的对象,并且返回,然后在方法上写一个@Bean注解,就可以将这个类的对象注入到SpringIOC容器中

作用：用于把当前方法的返回值作为bean对象存入spring的ioc容器中

属性:

name:用于指定bean的id。当不写时，默认值是当前方法的名称

细节：

当我们使用注解配置方法时，如果方法有参数，spring框架会去容器中查找有没有可用的bean对象。

查找的方式和Autowired注解的作用是一样的

**AnnotationConfigApplicationContext:**

作用:用于加载注解方式的配置类

ApplicationContext ac = new AnnotationConfigApplicationContext(SpringConfiguration.class);

**@Import(JdbcConfig.class):**

作用：用于导入其他的配置类

属性：

value：用于指定其他配置类的字节码。

当我们使用Import的注解之后，有Import注解的类就父配置类，而导入的都是子配置类

**@PropertySource("classpath:jdbcConfig.properties")**

作用：于指定properties文件的位置

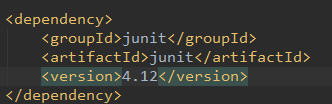
属性：

value：指定文件的名称和路径。

关键字：classpath，表示类路径下

### 4.Spring整合junit的配置

1导入spring整合junit的jar(坐标):spring-test



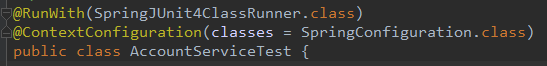
2、使用Junit提供的一个注解把原有的main方法替换了，替换成spring提供的 **@Runwith**

3、告知spring的运行器，spring和ioc创建是基于xml还是注解的，并且说明位置

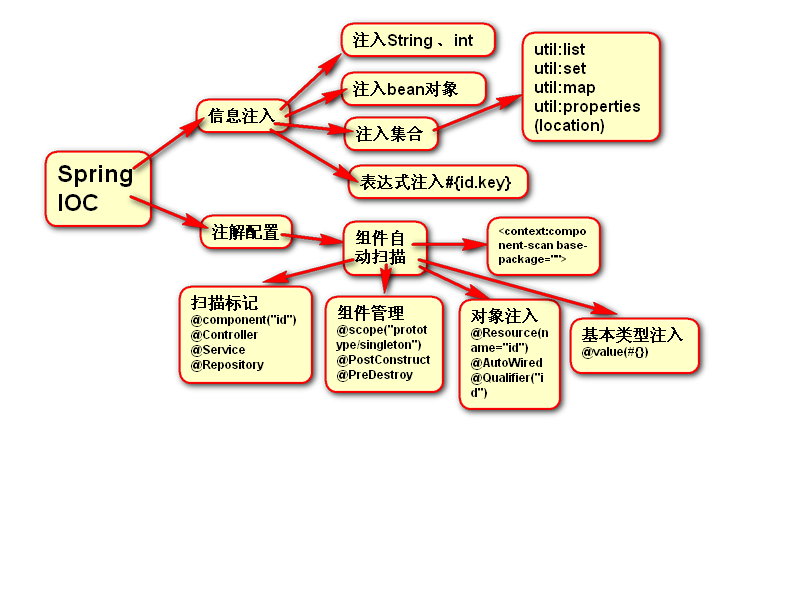
**@ContextConfiguration**

locations：指定xml文件的位置，加上classpath关键字，表示在类路径下

classes：指定注解类所在地位置

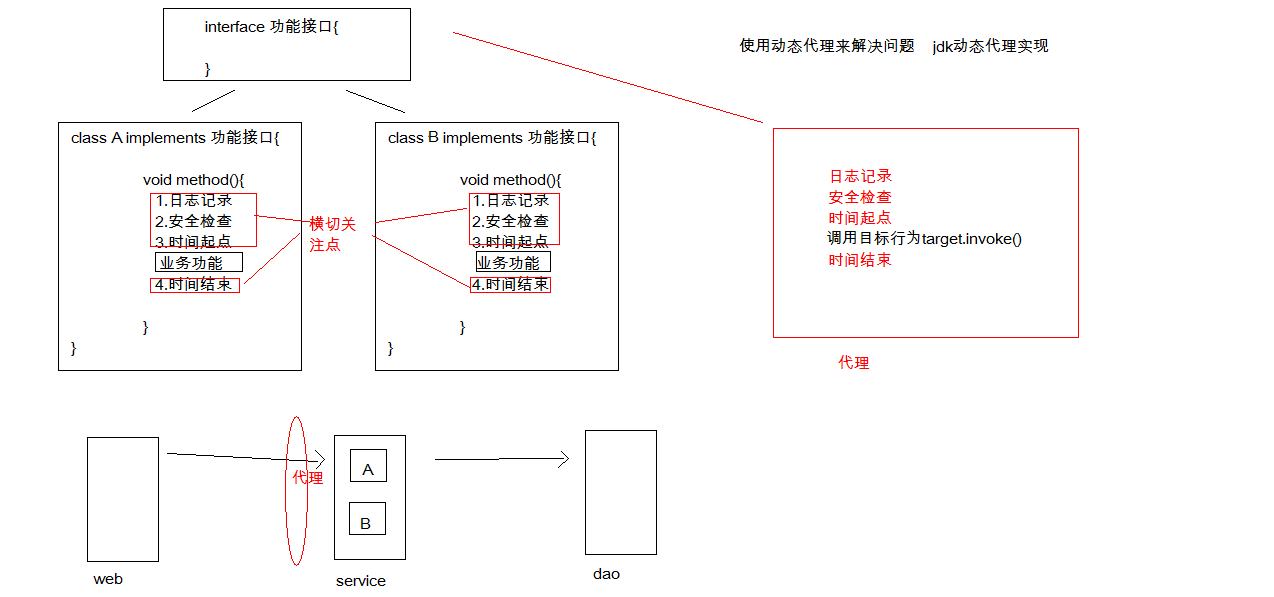


特别注意：当我们使用spring 5.x版本的时候，要求junit的jar必须是4.12及以上



## Day3

### AOP



### AOP与OOP的区别:

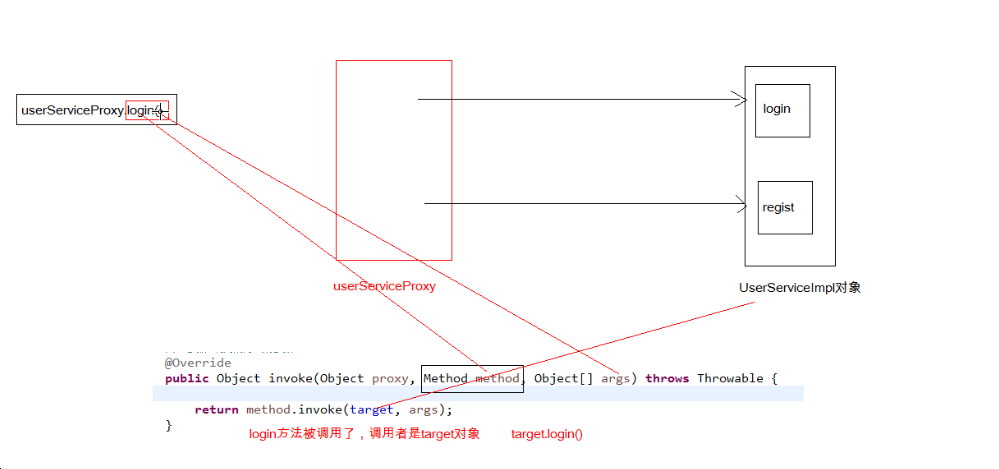
OOP（[面向对象编程](http://baike.baidu.com/view/10110.htm" \t "_blank)）针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的[逻辑单元](http://baike.baidu.com/view/1520672.htm" \t "_blank)划分。

而AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低[耦合性](http://baike.baidu.com/view/1731634.htm" \t "_blank)的隔离效果。这两种设计思想在目标上有着本质的差异。

AOP是为了弥补面向对象的不足，在OOP的基础上进行横向开发,AOP是OOP的延续.

。底层是实现机制是采用动态代理。

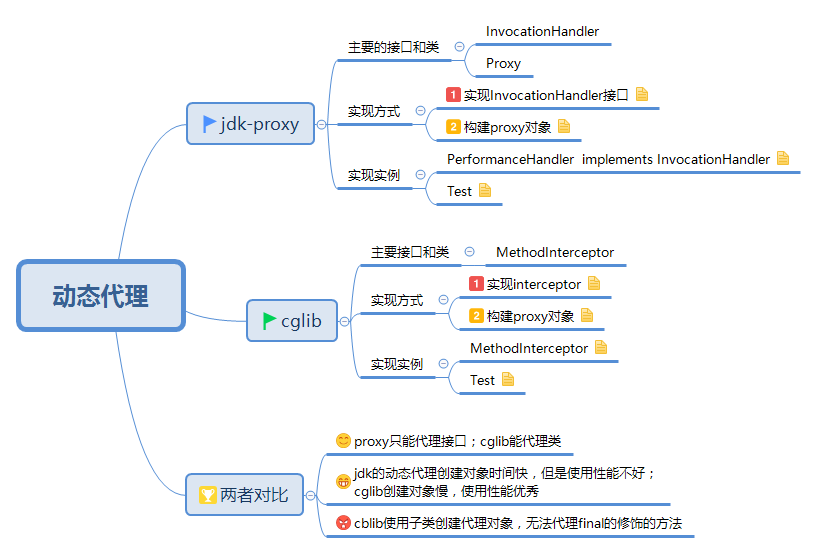
### 动态代理:



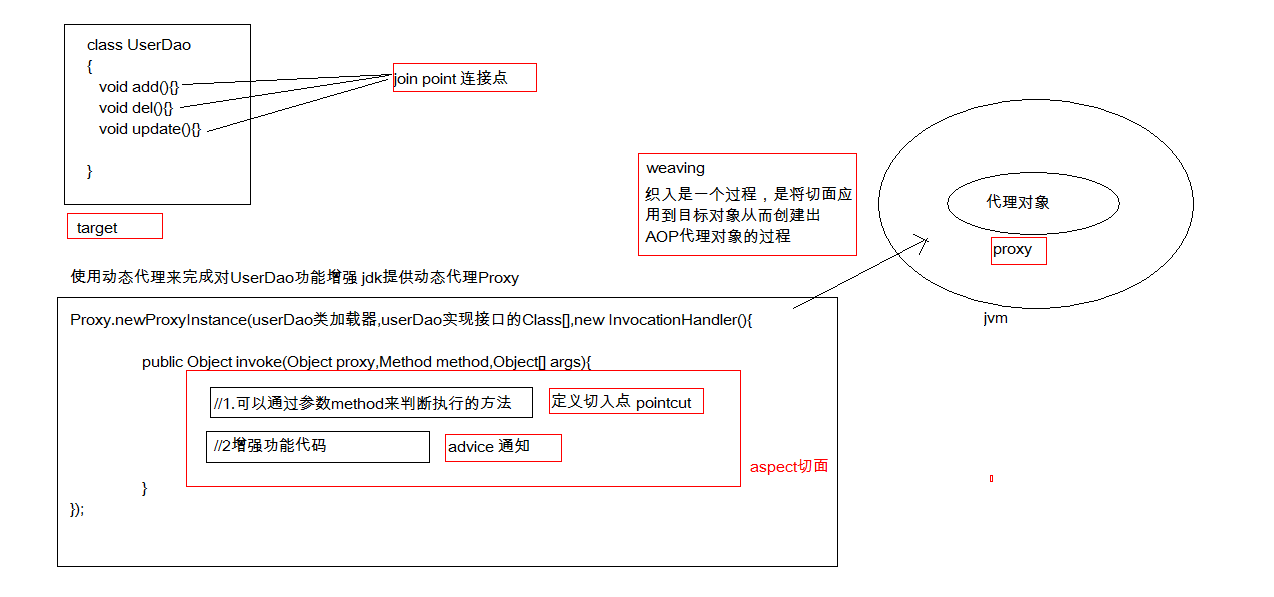
Jdk动态代理只针对于接口操作,目标对象必须实现一个接口,代理对象与目标对象实现同样的接口,是兄弟关系.

Cglib既可以代理有接口的类，也可以代理无接口的类。目标类不能用final修饰,代理类继承目标类,代理对象是目标对象儿子

#### 动态代理对比：【了解】



### Aop术语:



#### 目标对象target

指的是需要被增强的对象，由于spring aop是通过代理模式实现，从而这个对象永远是被代理对象。

#### 连接点(join point)

所谓连接点是指那些被拦截到的点，在spring中这些点指的是方法，因为spring只支持方法类型的连接点

#### 切入点(pointcut)

表示一组 joint point，这些 joint point 或是通过逻辑关系组合起来，或是通过通配、[正则表达式](http://baike.baidu.com/view/94238.htm" \t "_blank)等方式集中起来，它定义了相应的 Advice 将要发生的地方

简单说切入点是指我们要对哪些连接点进行拦截的定义

#### 通知(advice)

所谓通知是指拦截到连接点之后所要做的事情就是通知，通知分为前置通知，后置通知，异常通知，最终通知，环绕通知

Advice 定义了在 pointcut 里面定义的程序点具体要做的操作

#### 切面aspect

是切入点和通知的结合

#### 织入weaving

织入是一个过程，是将切面应用到目标对象从而创建出AOP代理对象的过程

#### 代理Proxy

织入后产生的就是代理对象

### 切入点表达式

从连接点中选出切入点,再进行增强

**切入点表达式的写法：**

关键字：execution(表达式)

表达式：

访问修饰符 返回值 包名.包名.包名...类名.方法名(参数列表)

标准的表达式写法：

public void com.itheima.service.impl.AccountServiceImpl.saveAccount()

访问修饰符可以省略

void com.itheima.service.impl.AccountServiceImpl.saveAccount()

返回值可以使用通配符，表示任意返回值

\* com.itheima.service.impl.AccountServiceImpl.saveAccount()

包名可以使用通配符，表示任意包。但是有几级包，就需要写几个\*.

\* \*.\*.\*.\*.AccountServiceImpl.saveAccount())

包名可以使用..表示当前包及其子包

\* \*..AccountServiceImpl.saveAccount()

类名和方法名都可以使用\*来实现通配

\* \*..\*.\*()

参数列表：

可以直接写数据类型：

基本类型直接写名称 int

引用类型写包名.类名的方式 java.lang.String

可以使用通配符表示任意类型，但是必须有参数

可以使用..表示有无参数均可，有参数可以是任意类型

全通配写法：

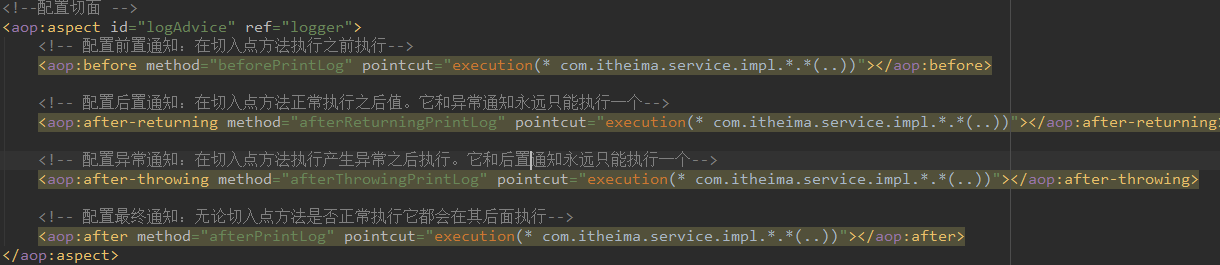
\* \*..\*.\*(..)

实际开发中切入点表达式的通常写法：

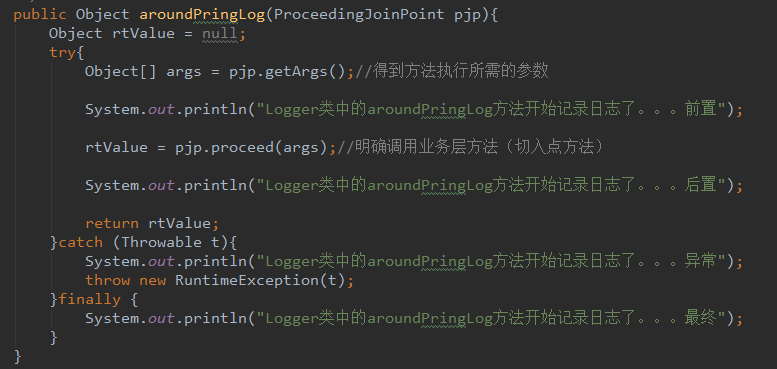
切到业务层实现类下的所有方法

\* com.itheima.service.impl.\*.\*(..)

### 四种常用通知类型



### 环绕通知



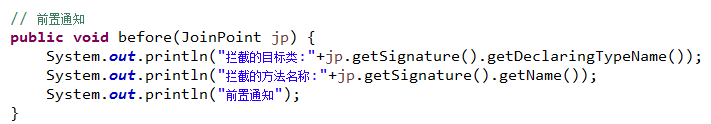
在bean.xml文件中配置环绕通知



### 通知上的参数介绍【了解】

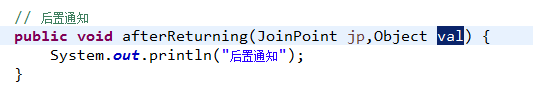
#### 1、在前置通知上可以添加JoinPoint参数

通过它可以获取目标相关的信息



使用前置通知可以完成日志记录，权限控制

#### 2、在后置通知上添加的参数

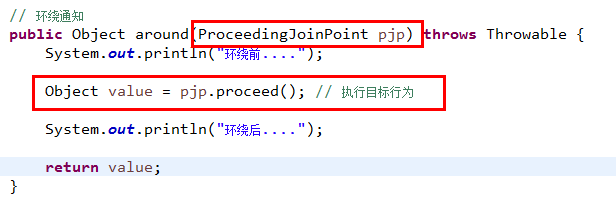


第二个参数val它可以获取目标方法的返回值

注意：需要在配置文件中配置

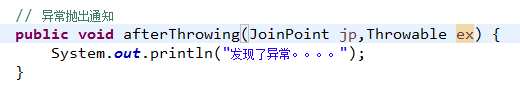


#### 3、环绕通知上的参数



它是我们开发中应用最多的，可以完成日志操作，权限操作，性能监控，事务管理

#### 4、抛出异常通知上的参数

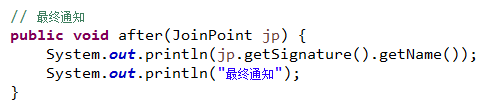


第二个参数Throwable它是用于接收抛出的异常

**注意:需要在配置文件中声明**



#### 5、最终通知上的参数



可以使用最终通知完成资源释放

### SpringAOP代理方案

在spring的aop开发中，它使用的是代理方案，代理实现有两种:

1. jdk的proxy
2. cglib

spring框架默认情况下，会对有接口的类使用proxy代理。没有接口的类使用cglib代理



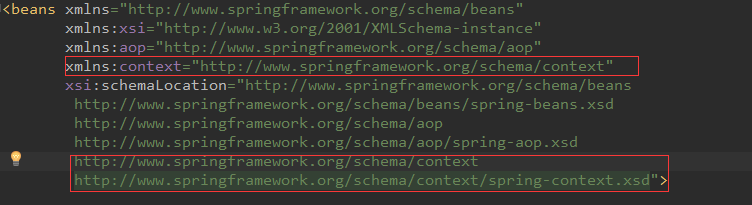
Proxy-target-class的值默认是false,它代表有接口使用proxy代理

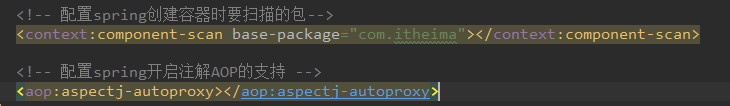
只需要将proxy-target-class设置为true。

使用cglib进行代理,target不能用final修饰

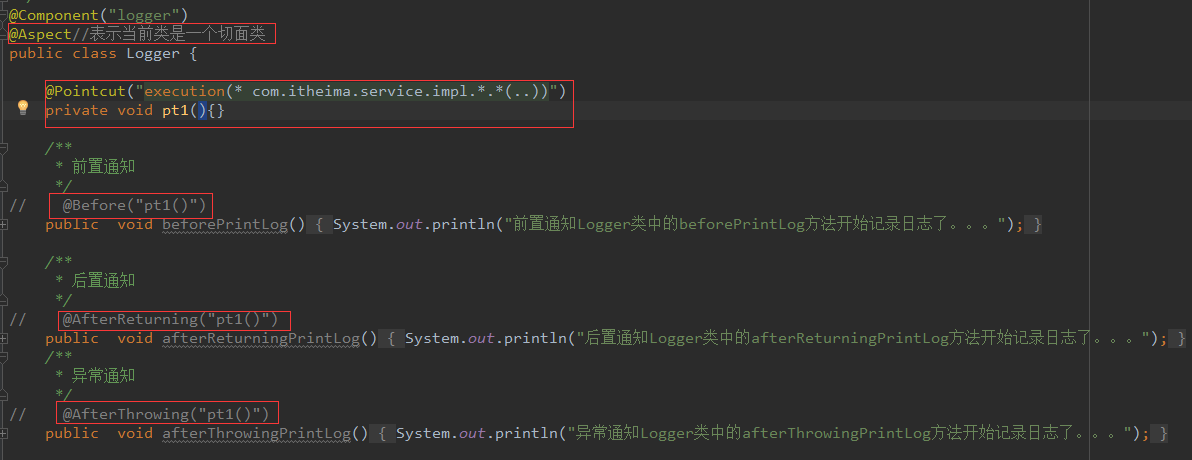
#### 基于注解的AOP配置

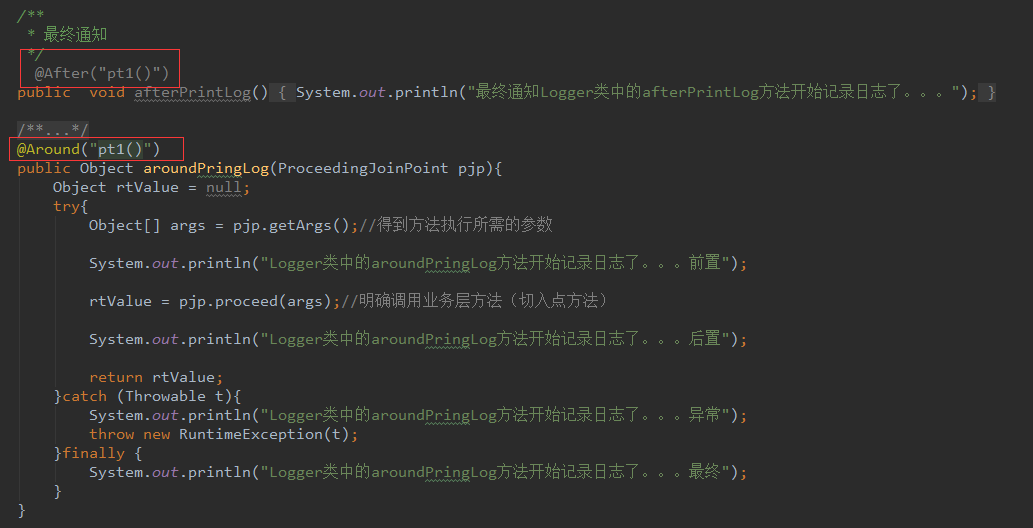
#### 1.修改bean.xml文件





#### 2.在Logger类中加入注解





## Day4

1. jdbcTemplate

Api:

1. 增删改使用update方法
2. 查询一个值(count)使用queryForObject
3. 查询一条或者多条数据query

在Dao中使用有两种:

1. 推荐xml配置时使用:继承JdbcDaoSupport(可以直接向dao注入dataSource),使用super.getJdbcTemplate().query()
2. 推荐在注解配置时使用:Private JdbcTemplate template成员变量(使用@AutoWired注入)

2.spring声明式事务配置

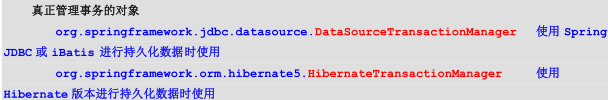
三个接口:

PlatformTransactionManager

此接口是 spring 的事务管理器，它里面提供了我们常用的操作事务的方法，如下图：



我们在开发中都是使用它的实现类，如下图：



TransactionDefinition

它是事务的定义信息对象，里面有如下方法：



**事务的隔离级别**



**事务的传播行为**

REQUIRED:如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）

SUPPORTS:支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）

MANDATORY：使用当前的事务，如果当前没有事务，就抛出异常

REQUERS\_NEW:新建事务，如果当前在事务中，把当前事务挂起。

NOT\_SUPPORTED:以非事务方式执行操作，如果当前存在事务，就把当前事务挂起

NEVER:以非事务方式运行，如果当前存在事务，抛出异常

NESTED:如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行 REQUIRED 类似的操作。

**超时时间**

默认值是-1，没有超时限制。如果有，以秒为单位进行设置。

**是否是只读事务**

建议查询时设置为只读。

TransactionStatus

此接口提供的是事务具体的运行状态，方法介绍如下图：



1.xml配置

**事务配置步骤**

1、配置事务管理器

2、配置事务的通知

此时我们需要导入事务的约束 tx名称空间和约束，同时也需要aop的

使用tx:advice标签配置事务通知

属性：

id：给事务通知起一个唯一标识

transaction-manager：给事务通知提供一个事务管理器引用

3、配置AOP中的通用切入点表达式

4、建立事务通知和切入点表达式的对应关系

5、配置事务的属性

注意:如果是使用的hibernate那么read-only属性默认是true

是在事务的通知tx:advice标签的内部

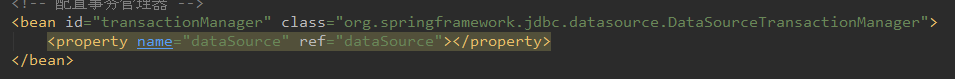
Bean.xml文件



2.注解配置

**spring中基于注解 的声明式事务控制配置步骤**

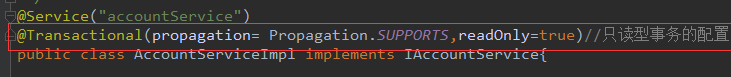
* 1. 配置事务管理器



* 1. 开启spring对注解事务的支持

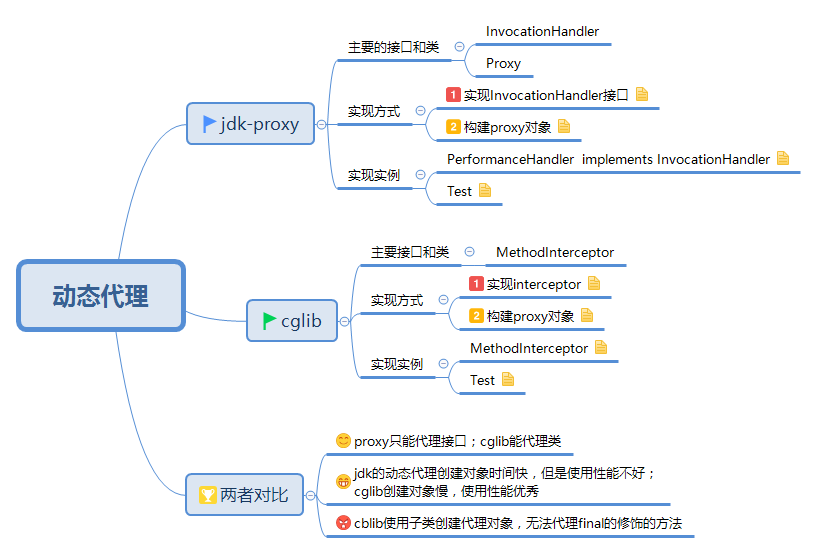


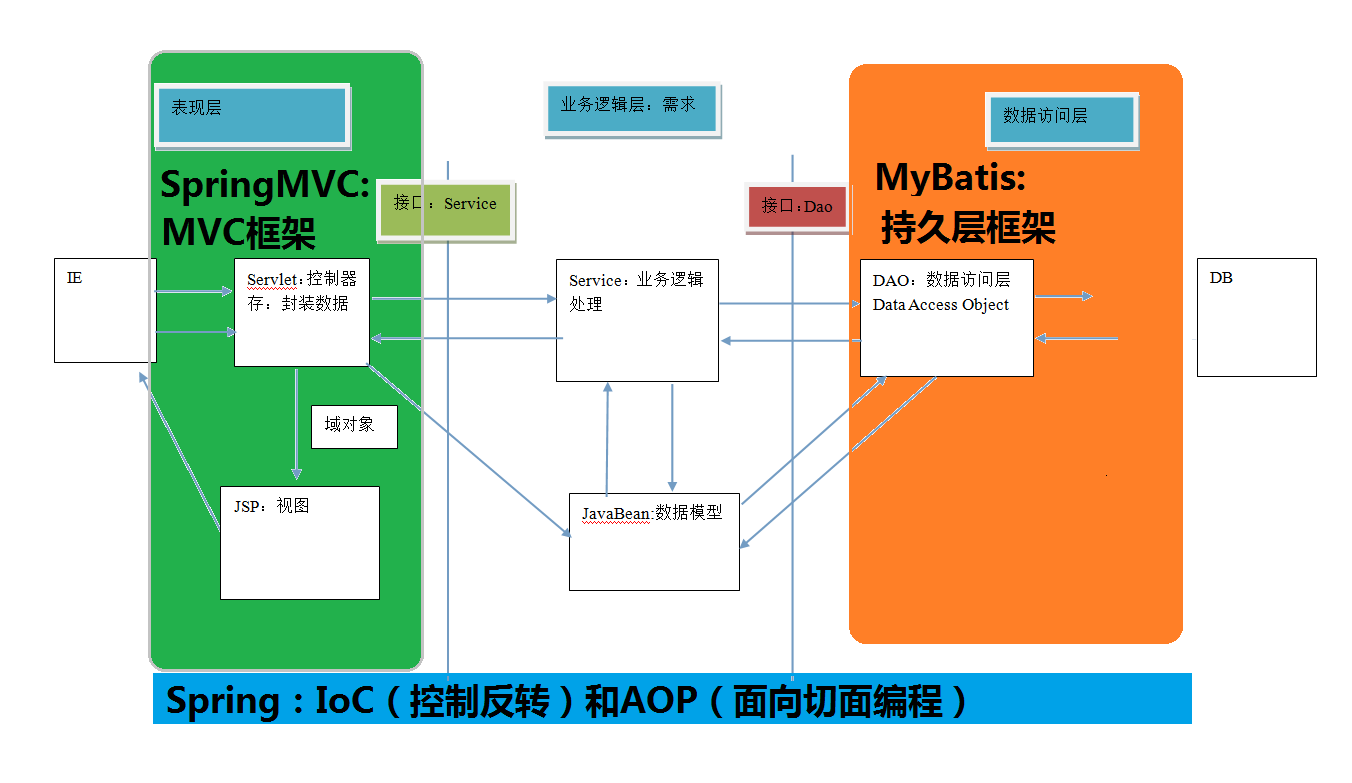
* 1. 在需要事务支持的地方使用@Transactional注解

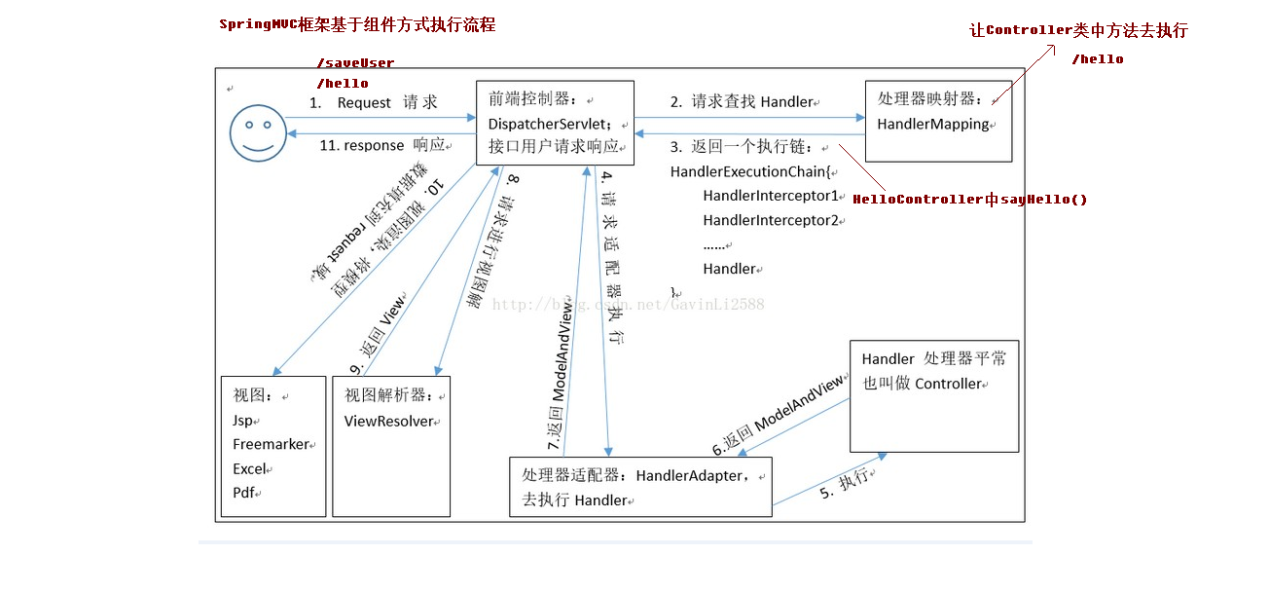


# SpringMVC:

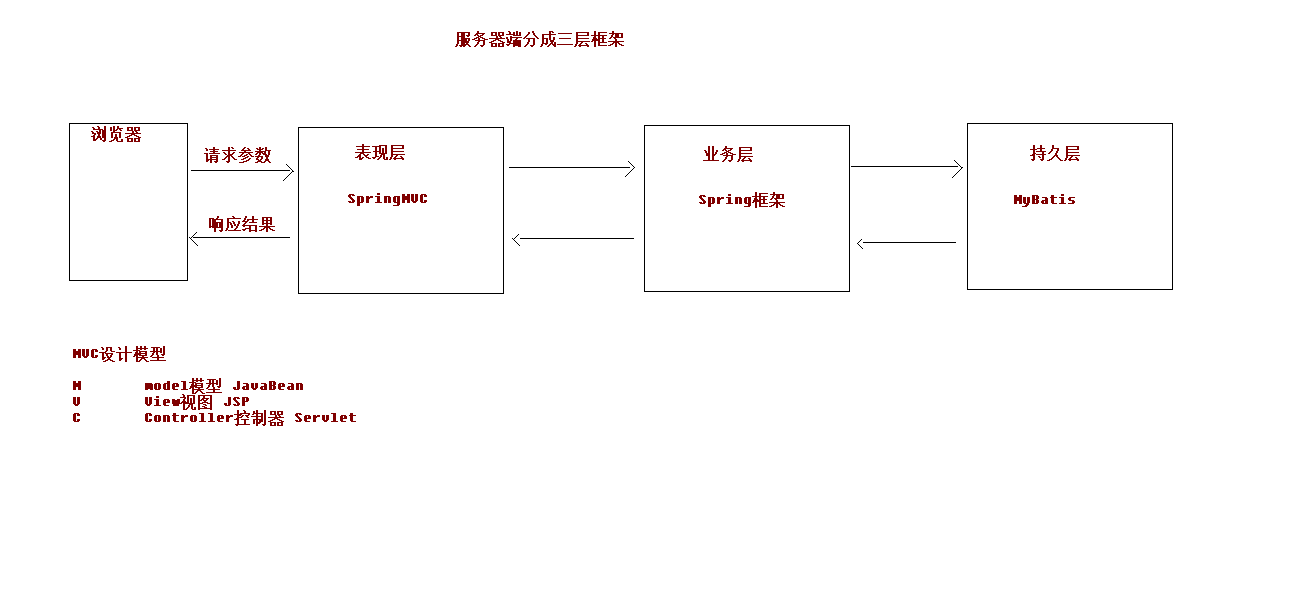
1. MVC
2. SpringMVC(执行流程)
3. 常用注解
4. ssm整合







## Day1



SpringMVC是什么？

轻量级开源的web层MVC框架.由Java语言开发的

SpringMVC的优势

1、清晰的角色划分：

前端控制器（DispatcherServlet）

请求到处理器映射（HandlerMapping）

处理器适配器（HandlerAdapter）

视图解析器（ViewResolver）

处理器或页面控制器（Controller）

验证器（ Validator）

命令对象（Command 请求参数绑定到的对象就叫命令对象）

表单对象（Form Object 提供给表单展示和提交到的对象就叫表单对象）。

2、分工明确，而且扩展点相当灵活，可以很容易扩展，虽然几乎不需要。

3、由于命令对象就是一个 POJO，无需继承框架特定 API，可以使用命令对象直接作为业务对象。

4、和 Spring 其他框架无缝集成，是其它 Web 框架所不具备的。

5、可适配，通过 HandlerAdapter 可以支持任意的类作为处理器。

6、可定制性，HandlerMapping、ViewResolver 等能够非常简单的定制。

7、功能强大的数据验证、格式化、绑定机制。

8、利用 Spring 提供的 Mock 对象能够非常简单的进行 Web 层单元测试。

9、本地化、主题的解析的支持，使我们更容易进行国际化和主题的切换。

10、强大的 JSP 标签库，使 JSP 编写更容易。

………………还有比如RESTful风格的支持、简单的文件上传、约定大于配置的契约式编程支持、基于注解的零配

置支持等等。

SpringMVC的使用:

开发环境搭建：

1.复制相关依赖

2.web.xml中配置SpringMVC的前端控制器DispacherServlet

3.配置SpringMVC的配置文件：springmvc.xml(并且将这个配置文件告知前段控制器加载)

A.开启注解扫描

B.配置一个视图解析器(要配置前缀和后缀)

C.开启SpringMVC对注解的支持(配置了适配器,映射器)

4.编写Controller代码(就是一个普通类@Controller注解)

写一个处理请求的方法(SpringMVC他是基于方法的,Struts2他是基于类的)

在这个方法上写上一个注解(@ReuqestMapping(“url”))

如果要响应到页面,那么给一个String类型的返回值,值就是页面的名字

5.编写成功页面

6.测试

入门案例的执行流程：

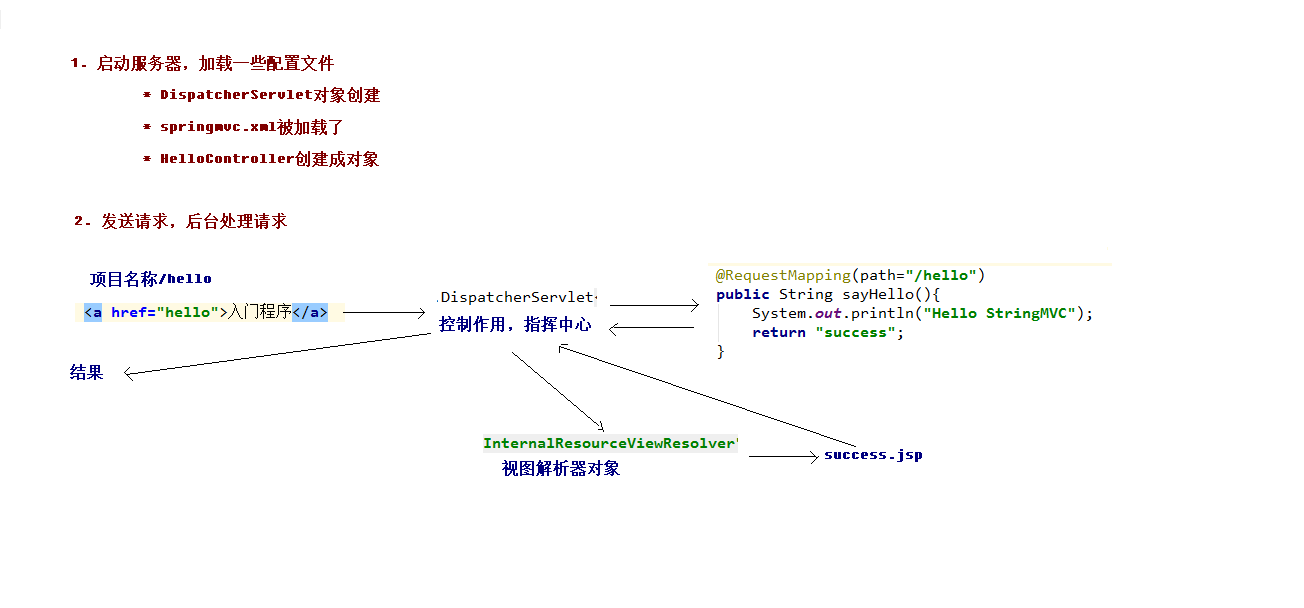
1. 1当启动Tomcat服务器的时候，因为配置了load-on-startup标签，所以会创建DispatcherServlet对象，就会加载springmvc.xml配置文件,然后创建一个WebApplicationContext容器,将springmvc.xml中配置的bean”都”实例化

1.2. 开启了注解扫描，那么HelloController对象就会被创建

2. 从index.jsp发送请求，请求会先到达DispatcherServlet核心控制器，根据配置@RequestMapping注解 找到执行的具体方法

4. 根据执行方法的返回值，再根据配置的视图解析器，去指定的目录下查找指定名称的JSP文件

5. Tomcat服务器渲染页面，做出响应



入门案例中使用的组件介绍:

1. 前端控制器（DispatcherServlet）

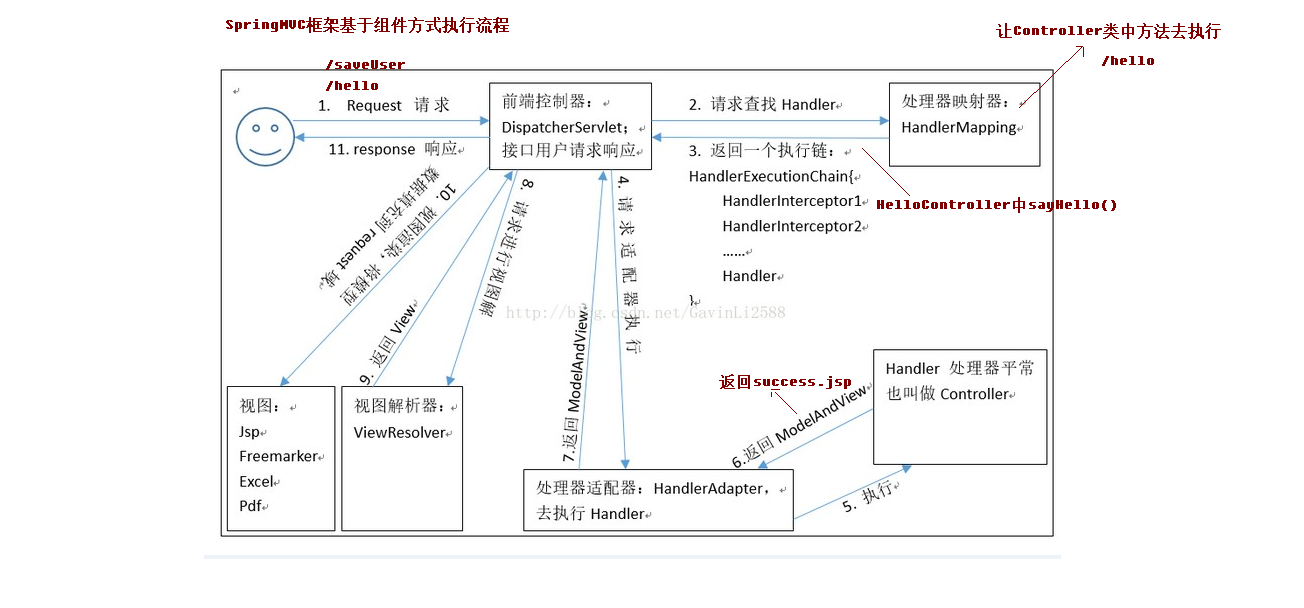
2. 处理器映射器（HandlerMapping）

3. 处理器（Handler）

4. 处理器适配器（HandlAdapter）

5. 视图解析器（View Resolver）

6. 视图（View）



封装请求参数:

1. 绑定机制

1. 表单提交的数据都是k=v格式的 username=haha&password=123

2. SpringMVC的参数绑定过程是把表单提交的请求参数，作为控制器(@Controller)中方法的参数进行绑定的

3. 要求：提交表单的name和参数的名称是相同的

2. 支持的数据类型

1. 基本数据类型和字符串类型

2. 实体类型（JavaBean）\*

3. 集合数据类型（List、map集合等）

1. 提交表单的name和JavaBean中的属性名称需要一致

2. 如果一个JavaBean类中包含其他的引用类型，那么表单的name属性需要编写成：对象.属性 例如：address.name

Data类型的转换:

1. 创建一个转换器<Srouce,Target> ---><String,Date>
2. 改造set方法,set方法参数该为String类型,做一个String转Date
3. 使用@DateTimeFormat(pattern=”yyyy-MM-dd”)

## Day2

### 方法返回值类型

* 1. String

Controller方法返回字符串可以指定逻辑视图的名称，根据视图解析器加上配置好的前缀和后缀,然后在服务器中找到对应的页面。

Model作为方法的参数,如果你使用Model对象去调用addAttrubite(“key”,value),那么这个键值对就会被存如Request作用域中.

1. forward请求转发

controller方法返回String类型，想进行请求转发也可以编写成:

return” forward:/user/findAll”

2. redirect重定向

controller方法返回String类型，想进行重定向也可以编写成:

//SpringMVC会给我们加上项目名

return” redirect:/user/findAll”

如果使用关键字定义转发和重定向,那么就不会使用视图解析器,所以我们需要写全文件的路径,而且重定向不能直接访问WEB-INF下的资源

* 1. void

1. 如果控制器的方法返回值编写成void，执行程序报404的异常，默认查找JSP页面没有找到。默认会跳转到@RequestMapping(value="/initUpdate") initUpdate的页面。

2. 可以使用请求转发或者重定向跳转到指定的页面

* 1. ModelAndView

ModelAndView对象是SpringMVC提供的一个对象，可以用来调整具体的JSP视图与设置存入Request域中的数据,视图的名字会被配置的视图解析器,加上前缀和后缀,找到对应的视图文件.

*// 创建ModelAndView对象* ModelAndView mv = **new** ModelAndView();  
 *// 把user对象存储到mv对象中，也会把user对象存入到request对象* mv.addObject(**"user"**,user);  
 *// 跳转到哪个页面* mv.setViewName(**"success"**);

*// 返回ModelAndView对象*  
 **return** mv;

### <url-pattern>过滤静态资源

DispatcherServlet会拦截到所有的资源，导致一个问题就是静态资源（img、css、js）也会被拦截到，从而不能被使用。解决问题就是需要配置静态资源不进行拦截，在springmvc.xml配置文件添加如下配置

<!-- 设置静态资源不过滤 -->

<mvc:resources location="/css/" mapping="/css/\*\*"/> <!-- 样式 -->

<mvc:resources location="/images/" mapping="/images/\*\*"/> <!-- 图片 -->

<mvc:resources location="/js/" mapping="/js/\*\*"/> <!-- javascript -->

1). location元素表示webapp目录下的包下的所有文件

2). mapping元素表示以/static开头的所有请求路径，如/static/a 或者/static/a/b

配置这个标签可以替代上面的配置<**mvc:default-servlet-handler** />

/ :所有请求

/\*:所有不带后缀的请求

\*.do:所有后缀为do的

/\*\*:多级路径

### Json数据处理

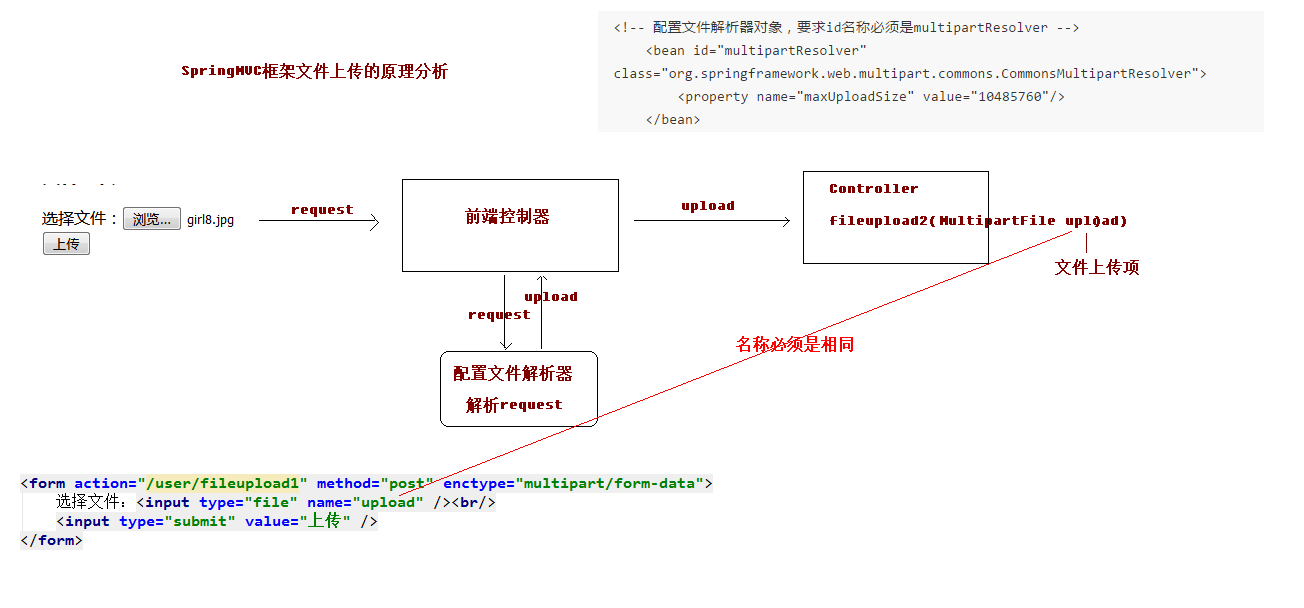
@RequestBody将请求的Json格式转换成Java对象传入方法

@ResponseBody将方法返回值转换成Json格式并输出(使用的是Jackson,所以要导Jackson依赖)

@RequestMapping(**"/testAjax"**)  
**public** @ResponseBody User testAjax(@RequestBody User user){

}

### SpringMVC文件上传



文件上传需要注意事项：

1、设置enctype = “multipart/form-data”

2、post请求方式 method = “post”

3、文件上传表单项 <input type=”file”

SpringMVC框架提供了MultipartFile对象，该对象表示上传的文件，要求方法形参名称必须和表单file标签的name属性名称相同。

**代码:**

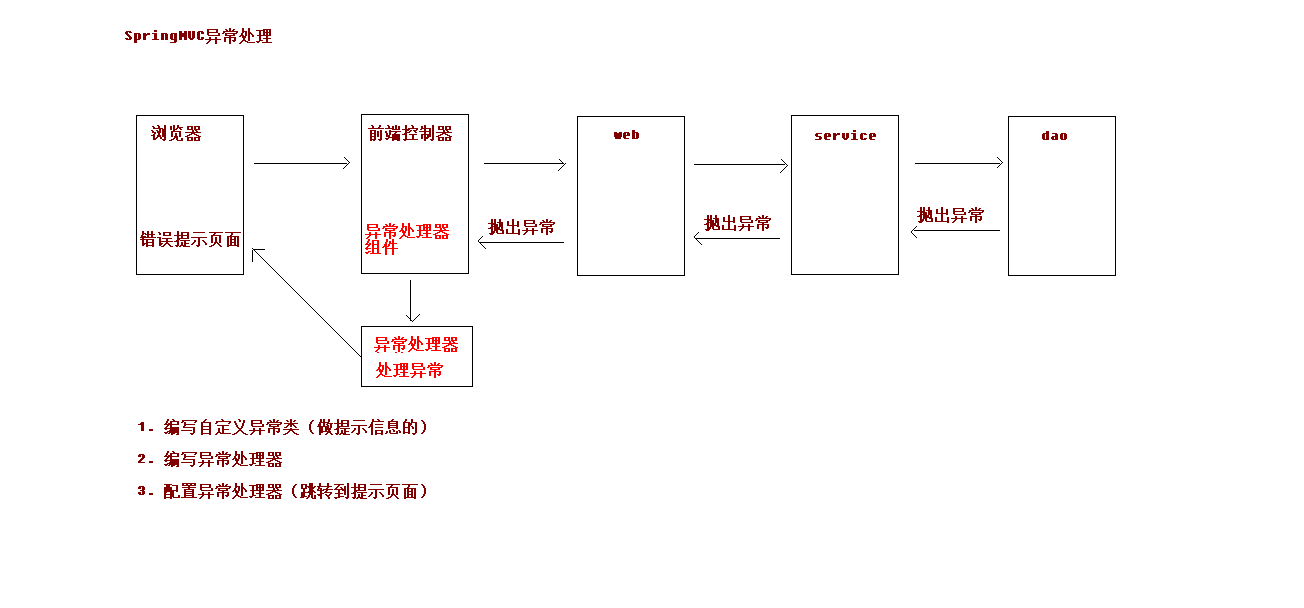
@RequestMapping(**"/fileupload2"**)  
**public** String fileuoload2(HttpServletRequest request, MultipartFile upload) **throws** Exception { *// 上传的位置* String path = request.getSession().getServletContext().getRealPath(**"/uploads/"**);  
 *// 判断，该路径是否存在* File file = **new** File(path);  
 **if**(!file.exists()){  
 *// 创建该文件夹* file.mkdirs();  
 }  
 *// 说明上传文件项  
 // 获取上传文件的名称* String filename = upload.getOriginalFilename();  
 *// 把文件的名称设置唯一值，uuid* String uuid = UUID.*randomUUID*().toString().replace(**"-"**, **""**);  
 filename = uuid+**"\_"**+filename;  
 *// 完成文件上传* upload.transferTo(**new** File(path,filename));  
 **return "success"**;  
}

**配置文件:**

*<!--配置文件解析器对象id必须是这个值-->*<**bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver"**>  
 <**property name="maxUploadSize" value="10485760"** />  
</**bean**>

1. **给一个处理请求的方法写一个形参(MultipartFile upload)形参的名字要与表单中type为file的name属性值相同**
2. **在xml配置文件中配置文件解析器**

### SpringMVC异常处理(HandlerExceptionResolver)



**a.写一个自定义异常类(写与不写可根据需求)**

*/\*\*  
 \* 自定义异常类  
 \*/***public class** SysException **extends** Exception{  
  
 *// 存储提示信息的* **private** String **message**;  
  
 **public** String getMessage() {  
 **return message**;  
 }  
  
 **public void** setMessage(String message) {  
 **this**.**message** = message;  
 }  
  
 **public** SysException(String message) {  
 **this**.**message** = message;  
 }  
}

**b.异常处理器(**resolveException代码可根据需求**)**

*/\*\*  
 \* 异常处理器  
 \*/***public class** SysExceptionResolver **implements** HandlerExceptionResolver{  
 */\*\*  
 \* 处理异常业务逻辑  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@return*** *\*/* **public** ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {  
 *// 获取到异常对象* SysException e = **null**;  
 **if**(ex **instanceof** SysException){  
 e = (SysException)ex;  
 }**else**{  
 e = **new** SysException(**"系统正在维护...."**);  
 }  
 *// 创建ModelAndView对象* ModelAndView mv = **new** ModelAndView();  
 mv.addObject(**"errorMsg"**,e.getMessage());  
 mv.setViewName(**"error"**);  
 **return** mv;  
 }  
}

**c.XML配置:**

*<!--配置异常处理器-->*<**bean id="sysExceptionResolver" class="cn.itcast.exception.SysExceptionResolver"**/>

**其他方法(直接使用SpringMVC提供的异常处理器):**

<**bean id="exceptionResolver" class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver"**>  
 *<!--默认跳转页面,注意这里没有走视图解析器-->* <**property name="defaultErrorView" value="/WEB-INF/pages/error.jsp"**></**property**>  
 *<!--对什么异常进行处理-->* <**property name="exceptionMappings"** >  
 <**props**>  
 <**prop key="异常"**>该异常跳转的页面</**prop**>

<**prop key="异常"**>该异常跳转的页面</**prop**>  
 </**props**>  
 </**property**>  
</**bean**>

1. **写一个类实现一个HandlerExceptionResolver,实现他的方法**
2. **将这个类配置到容器中**

### SpringMVC拦截器(HandlerInterceptor)

1. SpringMVC框架中的拦截器用于对处理器进行预处理和后处理的技术。

2. 可以定义拦截器链，连接器链就是将拦截器按着一定的顺序结成一条链，在访问被拦截的方法时，拦截器链中的拦截器会按着定义的顺序执行。

3. 拦截器和过滤器的功能比较类似，有区别

1. 过滤器是Servlet规范的一部分，任何框架都可以使用过滤器技术。

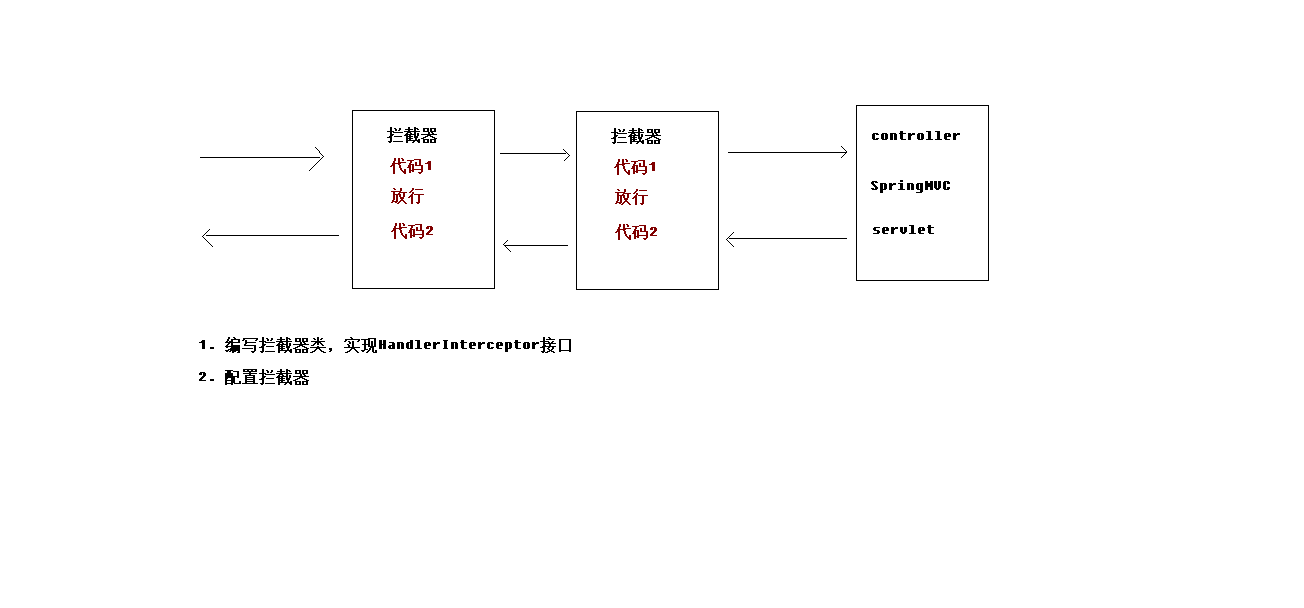
2. 拦截器是SpringMVC框架独有的。

3. 过滤器配置了/\*，可以拦截任何资源。

4. 拦截器只会对控制器中的方法进行拦截。

4. 拦截器也是AOP思想的一种实现方式

5. 想要自定义拦截器，需要实现HandlerInterceptor接口。



**a.编写拦截器:**

*/\*\*  
 \* 自定义拦截器  
 \*/***public class** MyInterceptor2 **implements** HandlerInterceptor{  
 */\*\*  
 \* 预处理，controller方法执行前  
 \* return true 放行，执行下一个拦截器，如果没有，执行controller中的方法  
 \* return false不放行  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@return*** *\** ***@throws*** *Exception  
 \*/* **public boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception {  
 System.***out***.println(**"MyInterceptor1执行了...前2222"**);  
 *// request.getRequestDispatcher("/WEB-INF/pages/error.jsp").forward(request,response);* **return true**;  
 }  
 */\*\*  
 \* 后处理方法，controller方法执行后，success.jsp执行之前  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param modelAndView*** *\** ***@throws*** *Exception  
 \*/* **public void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) **throws** Exception {  
 System.***out***.println(**"MyInterceptor1执行了...后2222"**);  
 *// request.getRequestDispatcher("/WEB-INF/pages/error.jsp").forward(request,response);* }  
 */\*\*  
 \* success.jsp页面执行后，该方法会执行  
 \** ***@param request*** *\** ***@param response*** *\** ***@param handler*** *\** ***@param ex*** *\** ***@throws*** *Exception  
 \*/* **public void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) **throws** Exception {  
 System.***out***.println(**"MyInterceptor1执行了...最后2222"**);  
 }  
}

**b.配置拦截器:**

*<!--配置拦截器-->*<**mvc:interceptors**>  
 *<!--配置拦截器-->* <**mvc:interceptor**>  
 *<!--要拦截的具体的方法-->* <**mvc:mapping path="/user/\*"**/>  
 *<!--不要拦截的方法  
 <mvc:exclude-mapping path=""/>  
 -->  
 <!--配置拦截器对象-->* <**bean class="cn.itcast.controller.cn.itcast.interceptor.MyInterceptor1"** />  
 </**mvc:interceptor**>  
  
 *<!--配置第二个拦截器-->* <**mvc:interceptor**>  
 *<!--要拦截的具体的方法-->* <**mvc:mapping path="/\*\*"**/>  
 *<!--不要拦截的方法  
 <mvc:exclude-mapping path=""/>  
 -->  
 <!--配置拦截器对象-->* <**bean class="cn.itcast.controller.cn.itcast.interceptor.MyInterceptor2"** />  
 </**mvc:interceptor**>  
</**mvc:interceptors**>

1. 写一个类实现HandlerInterceptor接口,并且按需求实现里面的方法
   * 1. preHandle//预处理方法
     2. postHandle//后处理方法
     3. afterCompletion//jsp执行后,才执行的方法
2. 将你写的类配置到xml文件中

<**mvc:interceptors**>  
 *<!--配置拦截器-->* <**mvc:interceptor**>  
 *<!--要拦截的具体的方法-->* <**mvc:mapping path="/user/\*"**/>  
 *<!--不要拦截的方法  
 <mvc:exclude-mapping path=""/>  
 -->  
 <!--配置拦截器对象-->* <**bean class="cn.itcast.controller.cn.itcast.interceptor.MyInterceptor1"** />  
 </**mvc:interceptor**>  
 </**mvc:interceptors**>

## Day3

SpringMVC Spring Mybatis

SpringMVC:

1.表现层框架,核心控制器:DispacherServlet,他是基于方法的处理器

2.架构:DispacherServlet,HandlerMapping,HandlerAdapter,Handler,ViewResolver,JSP

3.文件上传技术(文件解析器)

enctype=multipart/form-data

method = post

<input type = file name = >

4.Interceptor

拦截器与过滤器的区别?

1. 拦截器是SpringMVC特有的,只能是使用SpringMVC框架时才可以使用

过滤器是Servlet标准的,任何Web层框架都可使用

1. 拦截器只能拦截请求到方法(处理器)的那个路径

过滤器可以拦截所有请求

1. Handler(方法的参数,返回值)

返回值:String

重定向:return “redirect:url”;

转发:return “forward:url”;

1. 转换器(Converter)

将请求参数的类型(String)进行转换(Date)

1. 全局异常处理器(HandleException)

当有异常抛到了前段控制器,那么这时如果配置了异常处理器,就会调用异常处理器

8.注解:@RequestMapping(value=””,method=”RequestMethod.GET”)//可以写在类和方法上

@ParamValue(“id”) Integer ids//写在方法参数列表中,请求参数的key与形参名字不对应时使用

@PathVariable(“id”) Integer ids//写在方法参数列表中,支持RestFul风格 /item/list/{id}

{id}占位符的数据由方法的哪个参数接收

@ReuqestBody()//写在方法参数列表中,指定一个参数接收Json格式转换过来的数据

@ResponseBody()//写在方法上,将返回值转换成Json格式,用writer的方式响应给页面

Spring:

IOC

控制反转:将bean对象的管理权交给spring容器

DI:依赖注入:在spring创建bean对象时,将相关依赖注入对象

SPEL表达式#{id/name/key}//可以从容器中取值

${}一般用于从配置文件中去取值

目的:解耦(类之间解耦)

AOP

面向切面编程思想

与OOP的关系:OOP是面向对象编程,AOP是OOP的延续

底层实现:

JDK动态代理:实现共同接口

Cglib动态代理:继承目标类

名词:目标,连接点,切点,通知,切面,织入,代理

目的:方法内部解耦,将方法内部与业务逻辑关联不大的,比如日志,事务,性能,校验....横切关注点提取出来,用动态代理的方式增强目标方法.

AspectJ:实现AOP思想的一个框架,Spring整合了这个框架.用的最多的就是他的切点表达式

Transaction

Spring声明式事务管理:是Aop的一个实现,底层还是动态代理

3个接口:

TransactionManager//事务管理器

TransactionDefinition//事务的一些相关配置

只读,超时,传播,隔离

TransactionStatus//事务的状态

4个创建bean的注解:

@Component

@Controller

@Service

@Repository

4个用于注入的注解:

@AutoWired

@Qualifier()

@Resource

@Value

其他注解:

@Scope

@PreDestroy

@PostConstruct

@Bean:将方法的返回值存入SpringIOC容器

@Configuration:标记配置类

@ComponentScan:注解扫描

@Import:导入其他配置类

@RunWith:替换单元测试的main方法

@ContextConfiguration:指定配置文件的位置

Mybatis:

ORM持久层框架

架构:

SqlMapConfig.xml配置文件.Mapper.xml映射文件

SqlSessionFactory:重量级的,一般是单例的

SqlSession:连接

Excutor:执行器,执行sql语句的

MapperSatement:预处理sql语句

输入/输出

parameterType=”” resultType/resultMap

Mapper映射文件

Mapper映射接口

要求:(可以使用映射接口的代理对象)

1. namespace与接口的全限定名相同
2. <select>标签中的id与接口中的方法名相同
3. ParameterType与接口中方法的参数相同
4. resultType/resultMap与接口中方法的返回值相同

SSM整合:

Spring与Mabatis:SqlSessionFactoryBean,MapperSannerConfigurer:扫描映射文件包

Spring与SpringMVC:ContextLoaderListener