第三阶段

第三阶段和第四阶段影响工作做项目 ，第一第二阶段影响的是面试。

老师周三，周四，周五晚上一般都在

分清重要和非重要的知识，以实用为主，不是很重要的知识不要去重点学习。

day01_6.7 mybatis基础入门 day02_6.8 mybatis映射文件 day03_6.9 上午代理模式+了解源码 +　下午权限模块的dao实现 day04_6.10 spring的ioc day05_6.11 spring的aop（难点） day06_6.12 通过aop定义事务管理器 day07_6.14 springmvc基础 day08_6.15 springmvc高级 day09_6.16 ssm框架整合-----重要 day10_6.17 自习完成权限模式以及前端 day11_6.18 上午log4j日志 下午自习-------系统管理大模块必须完成 day12_6.19 redis缓存数据库------重要 day13_6.21 工具_maven项目管理工具 day14_6.22 工具_maven聚合工程_jjw组件 day15_6.23 工具_远程接口访问与定时任务和异步任务 day16_6.24 工具_短信和邮件发送 day17_6.25 工具_excel与Word文件生成 day18_6.26 工具_fastdfs文件存储服务器 day19_6.27 项目时间 0707_activiti工作流引擎 0708_springboot基础 0709_springboot高级 0710_mybatisplus框架 7月底结束 每天看部分面试题，提高面试的竞争力，提高表达能力

day01_6.7 mybatis入门

1. Mybatis有什么用？

Mybatis是数据库的工具，对jdbc的轻量级的封装，属于半自动化orm框架（OOP+SQL）。

oop：面向对象编程 orm： 对象关系映射持久层框架，比如：mybatis,hibernate (执行效率太低，封装的太完善了，纯自动化，不用写sql语句了)

利用xml（注解可能会遇到bug）配置文件进行配置。

提高开发效率，必定会降低运行效率，因此有部分注重效率的公司仍使用jdbc原生的东西。

2. 50分钟入门案例

- 安装mybatis的运行环境，准备jar包，准备jdbc的驱动包
- 数据库环境，数据库表的创建，对应的实体类
- 创建mybatis的核心配置
- 创建实体类对应的映射文件，并把映射文件加入到配置文件中
- 进行测试（运行mybatis的步骤）
  - 通过ClassLoader类加载 类路径(src/resources)下的文件
    - Test.class.getClassLoader().getResourceAsStream("mybatis-config.xml"),建议使用本方式，通用
    - in = Resources.getresourceAsStream("mybatis-config.xml")
  - 运行mybatis的核心对象SqlSessionFactoryBuilder,来加载mybatis-config.xml进行解析到Configuration对象中（可以理解为DataSource）。SqlSessionFactory ssf = new SqlSessionFactoryBuilder().builder(in,"dev"); 第二个参数为数据库的环境
  - 数据库会话对象(HttpSession)（可以理解为Connection）sqlSession sqlSession = ssf.openSession();
  - 调用会话对象中的方法，进行操作（执行数据库查询操作）
    - sqlSession.selectOne("",1)第一个参数为： 第二个参数为主键
  - 关闭数据库的连接sqlSession.close()
- 总结
  - sqlSessionFactoryBuilder加载Mybatis-config文件，根据数据库连接信息创建连接池对象
  - 解析sysuser-mapper.xml 文件，把该文件中的select标签进行解析 aa.bb.selectUserByKey -----> MappedStatement
  - 执行selectOne方法，传入字符串，找到MappedStatement对象，传入参数，执行sql语句

3.Mybatis的核心对象

Mybatis 运行sql语句，就只能接受一个参数，如果有多个参数，则自动封装为map，默认的key为0，1，可以使用注解的方式，将对应的属性值作为key进行封装@Param

- **SqlSessionFactoryBuilder**:构建者模式的应用，用来做复杂对象的构建，用来解析配置文件和映射文件，使用build方法创建对象DefaultSqlSessionFactory(Configuration), 该对象生命周期很短暂，用完就会被销毁。

```
InputStream in = TestSelectByKey.class.getClassLoader().getResourceAsStream("mybatis-config.xml");// 获取配置文件的文件流对象
SqlSessionFactory ssf = new SqlSessionFactoryBuilder().build(in, "dev");
```

- **SqlSessionFactory**----->DefaultSqlSessionFactory(Configuration): 会话工厂对象，全局唯一的单例对象，因为该对象内部存在数据库连接池，有文件解析结果Map

`SqlSession sqlSession = ssf.openSession();`

- **SqlSession对象**----->DefaultSqlSession: sql会话对象，用来执行数据库操作，用完就会关闭，在每次访问数据库时都需要创建它，每个线程都有自己的SqlSession对象，因此该对象不是共享的，也不是线程安全的。调用insert,update,selectList,selectOne,delete等方法执行增、删、查、改等操作；调用getMapper(xxx.class)来实例化接口执行增删查改；注意：SqlSession会话可以执行多次sql语句，当关闭了SqlSession对象后，需要重新创建。
  - selectOne
  - selectList
  - insert
  - update
  - delete
  - getMapper

getMapper方法：

```
1. 创建dao接口，SysuserMapper()接口
2. xml映射文件mapper 中的namespace="" 中间写接口的全名称
3. 映射文件中mapper下的增删改查标签的id值与方法名完全一致
4. 测试，使用session.getMapper(.class) 参数为接口的类对象
  SysuserMapper mapper = session.getMapper(.class);
  Sysuser sysuser = mapper.selectById(1); 查询
```

4. mybatis的配置文件

配置文件的引入，需要按照一定的顺序

该顺序为：(properties?, settings?, typeAliases?, typeHandlers?, objectFactory?, objectWrapperFactory?, reflectorFactory?, plugins?, environments?, databaseIdProvider?, mappers?)中间可以跳过，但是前后顺序必须保持一致

1. 数据库连接的配置
   - 可以直接在配置文件中定义数据库的连接参数

```
<dataSource type="POOLED">
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/crm?useSSL=true"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</dataSource>
```

   - 可以配置properties,通过引入外部的properties文件进行配置

```
<!--加载类路径下properties文件-->
<properties resource="jdbc.properties">
</properties>

<dataSource type="POOLED">
    <property name="driver" value="${jdbc.driver}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</dataSource>
注：配置文件的内容为：
jdbc.driver=com.mysql.jdbc.Driver
```

```
jdbc.url=jdbc:mysql://127.0.0.1:3306/crm?useUnicode=true&characterEncoding=utf8&useSSL=true&serverTimezone=UTC
jdbc.username=root
jdbc.password=root
```

2. settings: 修改mybatis的运行行为（一般项目中必须使用到该配置）

```
<!--修改mybatis运行的一些默认行为-->
    <settings>
        <!--开启日志 stdout_loggin：代表将日志文件输出到控制台中-->
        <setting name="logImpl" value="STDOUT_LOGGING"/>
        <!--开启驼峰命名映射-->
        <setting name="mapUnderscoreToCamelCase" value="true">
    </settings>
```

3. typeAlias:mybatis 类型别名的配置，==不同的包下也不能出现同名的类(项目中必用)

```
<typeAliases>
    <!--在创建Configuration对象,对指定包下的所有的类,做别名映射,把类名小写作为别名,映射文件中的resultType忽略大小写-->
    <package name="com.javasm"></package>
    <!--对单个的类使用别名-->
    <!--<typeAlias type="com.javasm.sys.entity.SysUser" alias="sysuser"></typeAlias>-->
</typeAliases>
```

4. environments:配置数据库环境信息


```
配置数据库环境
<environments default="dev">
  <!--在一个项目中可能有多个分库分表操作，后续有专门的mycat中间件做分库操作-->
  <!--第一个数据库环境-->
  <environment id="dev">
    <!--JDBC:开启事务-->
    <transactionManager type="JDBC">
</transactionManager>
    <!--UNPOOLED|POOLED|JNDI-->
    <!--PooledDataSource 一般都会使用数据库连接池，jndi一般用于和别的框架的配合使用-->
    <dataSource type="POOLED">
      <property name="driver"
value="${jdbc.driver}"/>
      <property name="url" value="${jdbc.url}"/>
      <property name="username"
value="${jdbc.username}"/>
      <property name="password"
value="${jdbc.password}"/>
    </dataSource>
  </environment>
</environments>
```

5. mappers：配置映射文件路径，再创建Configuration对象时，解析映射文件中的select,insert,update,delete标签到Configuration中的mappedStatements集合中，格式如下：namespace.id----->MappedStatement

```
 <mappers>
     <mapper resource="com\javasm\sys\mapper\sysuser-mapper.xml"></mapper>
 </mappers>
```

5. mybatis映射文件

- mapper: namespace="aa.bb" 必须唯一
- select, 查询标签

```
<!--
parameterType:参数类型:实体类,简单类型,map;
resultType:结果类型:实体类,简单类型,map;永远不会时list,set
如果parameterType是简单类型,#{随便写}
如果parameterType是实体对象,#{属性名}
如果parameterType是map,#{key}
-->
<select id="唯一标识" parameterType="参数类型" resultType="结果类型">
```

- insert 插入数据的标签，执行insert语句,改标签没有resultType属性.

```
<!--添加操作-->
<!--sysuser表是自增主键,有需要要获取新增记录id则需要设置useGeneratedKeys="true",keyProperty="uid"-->
<insert id="addUser" parameterType="Sysuser" useGeneratedKeys="true" keyProperty="uid">
  insert into sysuser(uname, upwd, uphone, uwechat, uemail, create_by) values
  (#{uname}, #{upwd}, #{uphone}, #{uwechat}, #{uemail},#{createBy})
</insert>
```

- update 更新数据

- delete 删除数据

```
<!--mybatis底层根据标签名,决定底层执行PreparedStatement对象的不同方法,executeQuery,executeUpdate-->
<update id="updateUserByUid" parameterType="Sysuser">
  update sysuser set upwd=#{upwd},uphone=#{uphone} where uid=#{uid}
</update>

<delete id="delUser" parameterType="int">
    delete from sysuser where uid=#{uid}
</delete>
```

- 注意点
  - 注意sql语句中使用#{}
  - 注意#{写法}
  - 注意id不能重复

6. 多参数传递

1. 多个参数封装到实体类中

   #{写对象的属性名}，对象的属性一定需要有get方法

```
<select id="selectUsersByUnameAndPwd" parameterType="Sysuser" resultType="sysuser">
  select * from sysuser where uname=#{uname} and upwd=#{upwd}
</select>
```

2. 多个参数封装到Map对象中，重要

```
 <!--#{map的key}-->
<select id="selectUsersByUnameAndPwd2" parameterType="map" resultType="sysuser">
  select * from sysuser where uname=#{uname_key} and upwd=#{upwd_key}
</select>
```

3. 引入mapper接口后，可以在接口的实参加@Param注解，指定key的值

```
List<Sysuser> login(@Param("uname2") String uname, @Param("upwd2") String upwd);
  <select id="login" parameterType="map" resultType="Sysuser">
     select * from sysuser where uname=#{uname2} and upwd=#{upwd2}
  </select>
  <select id="login" parameterType="map" resultType="Sysuser">
     select * from sysuser where uname=#{0} and upwd=#{1}
  </select>
  <select id="login" parameterType="map" resultType="Sysuser">
     select * from sysuser where uname=#{param1} and upwd=#{param2}
  </select>
```

7. #{}与${}的区别

两者都是用来获取查询参数,可用于sql语句中.

- #{}表达式
  - mybatis内对#{}解析为？占位符，数据更加的安全，因此对于条件查询语句必须使用#{}，避免sql注入
  - 如果parameterType是简单类型的话,#{随便写}
  - 如果parameterType是实体类,#{类的成员变量名}
  - 如果parameterType是map,#{map的key}
- mybatis内对${}不解析?占位符,而是直接进行sql拼接,因此不适合做条件查询

```
<select id="selectUsers" parameterType="map" resultType="sysuser">
    select * from sysuser order by ${soreField} ${sorted}
</select>
```

8. 常见异常

看最下方的Caused by: 查看错误

1.别名重复,包下有同名的类

```
org.apache.ibatis.type.TypeException: The alias 'SysUser' is already mapped to the value 'com.javasm.utils.Sysuser'.
```

2.反射异常,#{写错}

```
ReflectionException: There is no getter for property named 'aa' in 'class com.javasm.sys.entity.Sysuser'
```

3.映射文件中id重复

```
java.lang.IllegalArgumentException: Mapped Statements collection already contains value for aa.bb.selectUsers2
```

4.绑定异常,#{写错}

```
org.apache.ibatis.binding.BindingException: Parameter 'a' not found. Available parameters are [0, 1, param1, param2]
```

5.绑定异常,MappedStatement不存在,(dao接口中的方法在映射文件中没有标签)

```
org.apache.ibatis.binding.BindingException: Invalid bound statement (not found): com.javasm.sys.mapper2.SysuserMapper.del
```

6.绑定异常,接口未注册(映射文件中的namespace没有对应接口的名称),可能没引入映射文件

```
org.apache.ibatis.binding.BindingException: Type interface com.javasm.sys.mapper2.SysuserMapper is not known to the MapperRegistry.
```

注意:

数据库的字段是 date.,datetime(now()),timestamp(CURRENT_TIMESTAMP) 在实体类中一般用string类型的属性

类名的所有字母小写作为别名，常用使用的属性的别名已经内嵌进去了，映射文件中的resultType忽略大小写

在数据库中不要使用a_name 类似的命名方式，因为实体类需要aName,使用get或者set获取属性值是 getaName setaName,此时无法正常使用set和get方法

总结:

三个核心对象的api.尤其是SqlSession中的getMapper方法.

配置文件中的settings,typeAlias必须会.

映射文件中全部必须会.

9. test测试的常用用法

- 注解@FixMethodOrder(MethodSorters.NAME_ASCENDING)，当需要执行多个测试方法时，按照指定的顺序进行执行，可以避免在数据库中生成多余的记录数。
- 注解@BeforeClass，一般用来初始化类中定义的静态属性（最先运行）**全局初始化方法**
- 注解@AfterClass，在整个运行完毕的时候执行，**全局销毁方法**，或者关闭对象（最后运行）
- 注解@Before 用来初始化普通的属性，**注解测试初始化方法**
- 注解@After，在该测试模块执行完毕后进行执行，**注解测试销毁方法**一般用来关闭对象的连接

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TestCURD {
    private static SqlSessionFactory ssf =null;
    private  SqlSession session =null;
    /*beforeClass与afterClass定义的方法必须是static方法*/
    @BeforeClass
    public static void beforeClass()
    {
```

```
9           ssf= SSF.getFactory();
10      }
11
12      @AfterClass
13      public static void afterClass(){
14          ssf=null;
15      }
16
17      @Before
18      public void init(){
19          session= ssf.openSession();
20      }
21
22      @After
23      public void close(){
24          session.close();
25      }
26  /*定义测试方法*/
27      @Test
28      public void test1_addUser() throws IOException {
29          Sysuser u = new Sysuser();
30          u.setUname("huawei");
31          u.setUpwd("123123");
32          //受影响的行数
33          int rows = session.insert("aa.bb.addUser", u);
34          System.out.println(rows);
35          Integer uid = u.getUid();
36          System.out.println("uid:"+uid);
37          session.commit();
38          //set autocommit false
39      }
40  }
```

day02

第一天内容复习

1. mybatis 的配置文件
    - properties:加载路径下的properties文件，把该文件中的数据加载到Configuration对象中
    - Settings:mybatis运行日志，驼峰命名规则的设置，文件中的数据加载到Configuration对象中
    - typeAlias:别名映射，Mybatis默认有内嵌别名(String-string HashMap-map等)，通过指定package的包路径，(com.javasm不用指定详细的目录)，对包下的类做别名映射------>不同包下不允许出现同名的类。
    - environments: 数据库环境配置，文件中的数据加载到Configuration对象中
    - Mappers：映射文件的引入，文件中的数据加载到Configuration对象中的mappedStatements成员变量中，Map<String,MappedStatement>
2. xml的映射文件，id不能重复
    - select:接受的参数parameterType| 数据返回的参数 resutlType
    - insert parameterType|useGenerateKeys|keyProperty 插入操作时，可以通过配置，返回插入后的数据库对应的字段值（一般为主键id）通常使用于 配置有自增主键时
    - update:parameterType
    - delete:parameterType
    - 接收动态参数:#{属性名|map的key}
3. 核心的三个对象
    - SqlSessionFactoryBuilder-->build(InputStream in)
    - SqlSessionFactory(DefaultSqlSessionFactory)
    - Configuration配置对象,配置文件与映射文件在内存中的对象.
    - SqlSession(DefaultSqlSession)-->selectOne|selectList|insert|update|delete|getMapper.
4. junit的使用
    - @FixMethodOrder固定多个测试方法的执行顺序

- @BeforeClass注解静态方法,全局初始化方法
- @Before注解测试初始化方法
- @Test注解测试方法
- @After注解测试销毁方法
- @AfterClass注解静态方法,全局销毁方法.

实体类中的属性不能用long类型的数据，因为转为json后发送给前端后，可能会丢失精度，因此一般用integer或者String。

1. ## #{}与${}的区别

   - 相同点：都是用来获取动态参数
   - 不同点：
     - #{}：会转为？占位符 preparedStatement
     - ${}: 直接获取动态参数 拼接为sql语句，Statement，不建议使用，一般用来做排序
     - 动态注入，一般在前端有输入框的情况下才会发生
2. ## 映射文件中的sql标签

   提取出公共的sql语句，通过include来引入sql语句块

```
  <sql id="basicFields">
    uid,uname,uemail,uphone,uwechat
  </sql>

  <sql id="allFields">

uid,uname,uemail,uphone,uwechat,create_time,update_time,create_by
  </sql>

  <sql id="basicQuery">
    select uid,uname,uemail,uphone,uwechat,create_time,update_time,create_by from sysuser
  </sql>

  <select id="selectUsers" resultType="sysuser">
    select
```

```
    <include refid="basicFields"></include>
    from sysuser order by ${by} ${orderType}
  </select>
```

3. resultMap标签（重要）

resultMap: 结果集映射标签 把查询结果列映射到指定类的指定成员变量上

结果集映射标签(建议配置所有的字段的映射关系)

在select标签中结果集的类型设置为resultMap=""

resultMap标签中，可以配置映射关系，就是数据库的字段名和实体类中的属性名字不一致时，可以使用该映射，让他们一一对应起来。column是指数据库的字段名，proterty是指实体类的属性名

```
主键映射：<id column="uid" property="uid"></id>
非主键映射：<result column="uname" property="uname">
</result>
```

```
  <resultMap id="sysuserResultMap" type="sysuser">
    <id column="uid" property="uid"></id><!--主键列映射-->
    <result column="uname" property="uname2">
</result>
    <result column="uemail" property="uemail">
</result>
    <!--建议所有字段都映射上-->
  </resultMap>

  <select id="selectByKey" parameterType="int"
resultMap="sysuserResultMap">
    <include refid="basicQuery"></include>
    where uid=#{uid}
  </select>
```

注意点：select标签的resultMap属性不能与resultType同时出现

一个映射文件可能有多个resultMap标签

4. 对象关系映射-多对一映射| 持有关系映射（重要）

数据库表的设计关系：1对1 1对多 多对1 多对多

1-N ： 在1方关联多方的主键作为外键。

N-N： 使用中间表来进行关联两个表的主键作为外键

对象层面的设计关系：持有关系和聚合关系 建议看《uml模型设计》书籍

```
A{
  B b; //持有关系
  List<C> cs;//聚合关系
}
```

实现关联查询一共有三种方法

方法1：手工进行两次单表查询，并将查询出来的数据进行拼接，(一般在一的表所对应的实体类(a)中，添加一个多的表所对应的实体类(b)对象作为a实体类的一个成员变量)

方法2：使用sql语句进行表的链接查询，一次查询出所有的记录，然后分别进行映射，适用于查询列表

```
  <resultMap id="userAndRoleMap" type="sysuser">
    <id column="uid" property="uid"></id>
    <result column="uname" property="uname2">
</result>
    <result column="upwd" property="upwd"></result>
    <result column="uphone" property="uphone">
</result>
    <result column="uwechat" property="uwechat">
</result>
    <result column="uemail" property="uemail">
</result>
    <result column="create_time"
property="createTime"></result>
    <result column="update_time"
property="updateTime"></result>
    <result column="create_by" property="createBy">
</result>
```

```
    <!--sysuser类中的srole成员变量映射-->
    <association property="srole"
javaType="Sysrole">
      <id column="rid" property="rid"></id>
      <result column="rname" property="rname">
</result>
      <result column="rdesc" property="rdesc">
</result>
      <result column="rctime"
property="createTime"></result>
      <result column="rutime"
property="updateTime"></result>
    </association>
  </resultMap>

  <select id="selectUserAndRoleByUid"
parameterType="int" resultMap="userAndRoleMap">
    select u.*,r.rname,r.rdesc,r.create_time as
rctime,r.update_time as rutime from sysuser u left
JOIN sysrole r on u.rid=r.rid where u.uid=#{uid}
  </select>
```

方法3:使用mybatis内部的二次查询操作.(了解)

```
  <resultMap id="userAndRoleMap2" type="sysuser">
    <id column="uid" property="uid"></id>
    <result column="uname" property="uname2">
</result>
    <!--
              column:二次查询需要的参数列
              property:sysuser类中的成员变量名
              javaType:类型
              select:二次查询的位置
          -->
    <association column="rid" property="srole"
javaType="Sysrole"
select="com.javasm.mapper.SysroleMapper.selectRoleB
yKey"></association>
  </resultMap>
  <select id="selectUserAndRoleById2"
parameterType="int" resultMap="userAndRoleMap2">
```

```
    select * from sysuser where uid=#{uid}
  </select>
```

设计表的字段必须写注释信息---->提高代码的可阅读性

使用场景： 查询用户的列表(用户名 手机号 角色名)

5. 对象关系映射-1对多映射 | 聚合关系映射（重要）

应用场景：查询角色时，查询该角色下的所有用户

聚合应该用 collection标签，需指定ofType（集合的范型）

方法1:手工进行多次单标查询,把查询结果组合

```
  SysuserMapper um =
session.getMapper(SysuserMapper.class);
  SysroleMapper rm =
session.getMapper(SysroleMapper.class);
  int rid=2;
  //第一次查询:单查角色表
  Sysrole sysrole = rm.selectRoleByKey(rid);

  //第二次查询:单差用户表
  List<Sysuser> users = um.selectUsersByRoleId(rid);
    sysrole.setUsers(users);
```

方法2:使用sql表链接查询

```
  <resultMap id="roleAndUsersMap" type="sysrole">
    <id column="rid" property="rid"></id>
    <result column="rname" property="rname">
</result>
    <result column="rdesc" property="rdesc">
</result>
    <collection property="users" ofType="Sysuser">
<!--List<Sysuser>-->
      <id column="uid" property="uid"></id>
      <result column="uname" property="uname2">
</result>
      <result column="upwd" property="upwd">
</result>
```

```
      <result column="uphone" property="uphone"></result>
      <result column="uwechat" property="uwechat"></result>
      <result column="uemail" property="uemail"></result>
    </collection>
  </resultMap>

  <select id="selectRoleAndUsersByKey" parameterType="int" resultMap="roleAndUsersMap">
    select r.rid,r.rname,r.rdesc,u.uid,u.uname,u.upwd,u.uphone,u.uwechat,u.uemail from sysrole r left join sysuser u on r.rid=u.rid where r.rid=#{rid}
  </select>
```

方法3:mybatis内部发起二次单表查询,不建议使用

```
  <resultMap id="roleAndUsersMap2" type="sysrole">
    <id column="rid" property="rid"></id>
    <result column="rname" property="rname"></result>
    <result column="rdesc" property="rdesc"></result>
    <collection property="users" ofType="Sysuser" column="rid" select="com.javasm.mapper.SysuserMapper.selectUsersByRoleId"></collection>
  </resultMap>

  <select id="selectRoleAndUsersByKey2" parameterType="int" resultMap="roleAndUsersMap2">
    select * from sysrole where rid=#{rid}
  </select>
```

6. 动态sql语句（重要）

非常重要的知识点

在xml映射文件中进行sql语句拼接的一种方式，使用where标签，可以生成sql中的where关键字，并忽略紧跟之后的and或者or，如果要使用模糊查询则必须用"%"或者使用concat函数进行拼接，如果where标签中的所有条件都部满足则不会生成where关键字.

- where
  - where ---if
  - 在xml映射文件中进行sql语句拼接的一种方式，使用where标签，可以生成sql中的where关键字，并忽略紧跟之后的and或者or
  - 如果要使用模糊查询则必须用"%"#{rname}"%"(必须使用双引号)或者使用concat函数进行拼接
  - 如果where标签中的所有条件都不满足则不会生成where关键字.
  - 如果要使用<号(会与标签的开始进行混淆)，则必须使用特殊符号 <，小于等于用 <=#{rid}
  - ```
    CONCAT("%",#{rname},"%")
    "%"#{uname}"%"

    小于比较,小于号存在歧义,需要使用特殊符号
    小于: and rid < #{rid}
    小于等于:and rid <= #{rid}
    ```
- if
  - 条件判断 test属性写boolean表达式
- set
  - 一般用于update标签中，可以生成set关键字
  - 生成set关键字，并忽略最后的一个逗号。rname=#{rnaem},
  - 不会把其他为null值 置为null，不会重置其他未修改的字段
- foreach
  - 循环标签：用于批量删除 批量添加 上面
  - foreach中有5个属性，collection="array"---->collection为默认封装的key的值，open="（",循环的开始标志 close=")"循环结束的标志，item="roleid"：循环变量的名字，separator=","循环变量之间使用，分隔。
    ```

        - mybatis底层把数组类型的参数，封装为Map，map.put("array",rid)
        - mybatis底层把集合类型的参数，封装为Map，map.put("list",rid)
        - 通过@Param 可以自定义封装后的key值（在接口定义的方法中使用该注解）
        - ----------使用添加操作----------
        - 使用对象的数组
    - choose(when,otherwise)
        - 类似于switch case，where-choose--when，只可以选择一个条件，如果有 一个条件满足，则其他的when都不会执行，如果都不满足则执行 otherwise

7. mybatis延迟加载

主要掌握延迟加载的思想理念，分页也是延迟加载，一般是为了用户体验，一般条数超过100的都需要延迟加载

图片是一般是游览器加载速度比较慢的

1. ```
   <!--全局开启延迟加载-->
   <setting name="lazyLoadingEnabled" value="true"></setting>

   <!--改积极加载为消极加载-->
   <setting name="aggressiveLazyLoading" value="false"></setting>
   <!--禁用toString,equals,hashcode,clone方法的触发延迟加载属性-->
   <setting name="lazyLoadTriggerMethods" value=""></setting>
   ```

2. 只存在于mybatis中使用二次查询的时候，即先查询用户，再查询角色的时候才会出现
3. 需要在配置文件中进行延迟加载的设置
4. 默认只做第一次的查询，当需要获取关联对象的时候，才做第二次的查询
   ```

5. 其中可以选择进行配置（aggressiveLazyLoading true|false），当为true时，当有调用有延迟加载的属性的对象时，就会进行二次加载，如果禁用(false)时，则只有加载需要延迟加载的对象时才会触发延迟加载。
6. lazyLoadTriggerMehods,延迟加载的触发方法，默认(toString,clone,hashcode,equals)时，可以进行value=""的配置

8. mybatis缓存使用

orm框架中都有缓存的实现，但是一般在项目中没有啥用。

一般在web开发中有专门的缓存的方法

一般不会使用，相同的语句，一般不会进行查询第二次

一级缓存：session级别的缓存，默认开启

- 查询过程：先去sqlSession对象的缓存空间查询数据
- 查询到则直接返回
- 如果查询不到，则查询数据，并将查询的结果保存到sqlSession中

二级缓存：默认是没有开启的，需要在setting中进行配置,跨会话共享空间，可以在不同会话之间共享缓存的数据，相当于全局的缓存

```
1.在配置文件中需要进行配置，mybatis.xml
2.需要在map的xml文件中进行配置
1.全局开启二级缓存
<setting name="cacheEnabled" value="true">
</setting>

2.map的xml中的配置
eviction缓存策略：先进先出
flushInterval: 缓存的刷新时间，设置为60s
size:最大的缓存的数量
readonly:只读（无法修改缓存空间中的数据）
<cache eviction="FIFO" flushInterval="6000"
size="512" readOnly="true"/>
// 如果执行了insert或者update操作则可以指定flushCache：可以刷新缓存，可以把已有的缓存清空，强制刷新缓存
<update flushCache="true">

以上配置生效后，该配置文件中的所有查询操作都默认使用缓存的操作。如果某个查询操作不使用二级缓存，则可以进行设置
```

```
16  <select useCache="false"></select>
```

9. # 代理模式-静态代理实现

   23种设计模式

   可以了解disruptor高并发框架，被很多高并发采用。

   代理(proxy)模式的使用场景：当一个已经存在对象，某个方法不满足需求时，使用代理模式，代理模式分为：静态代理和动态代理

   静态代理是基础：使用实现或者继承创建一个新的代理类，重写需要修改的方法

   代理类与被代理类同类型

   代理类持有被代理（被代理的对象作为代理类的一个属性）

   经典应用场景：Connection------->close方法关闭连接

   DruidDataSource---->Connection conn = getConnection()---->conn.close() 方法可以放回连接池中

   就是dataSource 作为connection的一个代理，修改了conn中的close方法

10. # 总结与重点

    resultMap标签 association（持有关系）与collection（聚合关系）与id和result的四个标签

    动态sql：if|where|set|foreach

    静态代理

day03

内容复习

Mybatis整体重点

1. 配置文件中settings标签，修改mybatis运行行为，日志，驼峰映射
2. 映射文件中的resultMap(id,result,association,collection)
3. 映射文件中的select insert update delete （ parameterType="实体类"，map，简单类型 ）
4. 映射文件中#{} 的写法
5. 映射文件中的if,where,set foreach(collection,open,close,item,seperator)
6. 支持插件扩展

1. 代理模式

实用编程基础的书籍

敏捷软件开发，原则，模式，实践

<UML模型设计>

微服务架构

spring响应式编程

java高并发编程

使用场景：在不改变源码的情况下，对一个已经存在的对象的方法进行扩展（做日常项目用不上，架构组/中间件开发/代理模式，反射，观察者模式）

参与角色：被代理对象(已经存在的对象)，代理对象(新创建的对象)，调用者(调用代理对象)----

- 静态代理：缺点：需要手动派生代理类，只对需要扩展的方法进行特殊操作
- **动态代理**

    - 动态代理实现方式：:Proxy,cglib组件
    - Proxy工具类要求被代理对象必须有接口.因为$Prxoy0代理类已经使用了extends关键字;
    - $Proxy0 类中持有的是invocationHandle对象
- Proxy工具类生成的代理类结构

```
public class $Proxy0 extends Proxy implements Connection{

  protected InvocationHandler h;
  private Method m38;

  static{

 m38=Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] { Class.forName("java.lang.String") });
  }

  public $Proxy0(InvocationHandler paramInvocationHandler)
  {
    this.h=h;
  }

  public final PreparedStatement prepareStatement(String sql) throws Exception
  {
      return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
}
```

动态代理的操作流程分析

- 获取类加载器对象 ClassLoader l = souce.getClass.getClassLoader()
- 获取需要（被代理对象所实现的接口）的对象 本例是：class[] interfaces = new class[] {Connection.class}
- 获取InvocationHandle 对象，该对象中定义了一个匿名内部类

```
//Proxy.newProxyInstance()创建代理类$Proxy0,并实例化对象的过程.
ClassLoader l = source.getClass().getClassLoader();
Class[] interfaces=new Class[]{Connection.class};
//回调处理器
/**
* proxy: 代理对象,该对象绝对不能调用,只能传递引用.
* method: 正在调用的方法Method对象.
* args: 正在调用的方法的实参
*/
InvocationHandler h = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String name = method.getName();
    if(name.equals("close")){
      System.out.println("不在关闭,放回连接池");
    }else{
      Object result= method.invoke(source, args);
      return result;
    }
    return null;
  }
};
```

- 利用Proxy中的newProxyInstance方法创建一个代理对象。`Connection o = (Connection) Proxy.newProxyInstance(l, interfaces, h);`

动态代理的原理分析

- 执行newProxyInstance()方法时会创建一个$Proxy0的类，该类继承了Proxy,实现了数组interfaces中的所有接口
- $Proxy0中的构造器是一个以InvocationHandler对象为参数的有参构造，该构造的有参构造又调用了父类Proxy中的有参构造

```
public class $Proxy0 extends Proxy implements Connection{

  protected InvocationHandler h;// 成员变量
  private Method m38; // 方法对象
```

```
    static{

 m38=Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] {
Class.forName("java.lang.String") });
    }
    public $Proxy0(InvocationHandler paramInvocationHandler)
  {
     super(paramInvocationHandler)
  }
    public final PreparedStatement prepareStatement(String sql) throws Exception
  {
       return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
}
```

- Proxy中的有参构造

```
protected Proxy(InvocationHandle h){
  Ojbects.requireNonNull(h);
  this.h = h; // this指的是调用者，这里的调用者是指子类的对象，即：$Proxy0 将 newProxyInstance() 方法传入的h对象赋值给成员变量h，即：$Proxy0对象持有了一个InvocationHandler对象
}
```

- 此时$Proxy0初始化完毕，建立了一个 connection对象proxy
- 当o去执行相关方法时，例如：proxy.parepareStatement("****")时，会执行Proxy0中重写的parepareStatement方法，该方法中的this------->代理对象proxy, proxy.h------->上述的成员属性h-------->执行newProxyInstance()方法时所传的参数h，invoke--------->我们在new InvocationHandler()对象时所定义的匿名内部类中 我们重写的invoke方法。

```
public final PreparedStatement prepareStatement(String sql) throws Exception
  {
       return (PreparedStatement)this.h.invoke(this, m38, new Object[]{sql});
  }
```

- 上一步骤中执行的Invoke(this, m38, new Object[]{sql})方法中，this---->proxy对象，m38-------->

  `Class.forName("java.sql.Connection").getMethod("prepareStatement", new Class[] { Class.forName("java.lang.String") })` new Object[]{sql}---------->调用者所传递的实参

- 执行我们自定义的invoke时，可以通过method.getName() 获取要执行的方法的方法名，若满足一定条件(name.equals("close"))，则我们可以自定义其中的逻辑功能。若不满足，则默认调用 被代理对象中的初始方法

```
InvocationHandler h = new InvocationHandler() {
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String name = method.getName();
    if(name.equals("close")){
      System.out.println("不在关闭,放回连接池");
    }else{
      // source 为被代理对象
      Object result= method.invoke(source, args);
      // 将结果返回
      return result;
    }
    return null;
  }
};
```

2.Configuration对象的构建过程

了解配置文件与映射文件的加载过程

configuration对象的构建的整个流程

从SqlSessionFactoryBuilder对象的build(in)方法中获取一个SqlSessionFactory对象

- 通过SqlSessionFactoryBuilder()类的build(in)方法创建一个SqlSessionFactory对象

```
factory = new SqlSessionFactoryBuilder().build(in);
```

- builder方法

```
public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
  SqlSessionFactory var5;
  try {
    XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
    var5 = this.build(parser.parse());
  } catch (Exception var14) {
    throw ExceptionFactory.wrapException("Error building SqlSession.", var14);
  } finally {
    ErrorContext.instance().reset();
    try {
      inputStream.close();
    } catch (IOException var13) {
      ;
    }
  }
  return var5;
}
```

- 新建XMLConfigBuilder对象时调用两个构造器， 在一个构造器中对Configuration进行了初始化操作

```
public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {
  this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, props);
}

private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {
  super(new Configuration());
  this.localReflectorFactory = new DefaultReflectorFactory();
  ErrorContext.instance().resource("SQL Mapper Configuration");
  this.configuration.setVariables(props);
  this.parsed = false;
  this.environment = environment;
  this.parser = parser;
}
```

- XMLConfigBuilder对象中parse()方法为：配置文件构建者的解析方法，

```
public class XMLConfigBuilder extends BaseBuilder {
  // Parsed为判断标记，创建configuration对象是会将parsed置为true
  // 保证configuration对象的全局唯一性
  private boolean parsed;
  private XPathParser parser;  // xpath解析对象
  private String environment;
  private ReflectorFactory localReflectorFactory;
  //在父类BaseBuilder中的成员变量，且是final类型，即：地址不可变（全局唯一），  类型为protected,即：子类可以使用父类的该属性
  protected final Configuration configuration;
  public Configuration parse() {
    if (this.parsed) {
      throw new BuilderException("Each XMLConfigBuilder can only be used once.");
    } else {
      this.parsed = true;
```

```
 this.parseConfiguration(this.parser.evalNode("/configuration"));
      return this.configuration;
    }
  }
}
```

- XpathParser对象中的evalNode()方法，调用了XpathImpl实现类中的evaluate（ ）方法，可以获取到一个XNode root 对象

```
public class XPathImpl implements javax.xml.xpath.XPath {

  public Object evaluate(String expression, Object item, QName returnType)
    throws XPathExpressionException {
    if ( expression == null ) {
      String fmsg = XSLMessages.createXPATHMessage(
        XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
        new Object[] {"XPath expression"} );
      throw new NullPointerException ( fmsg );
    }
    if ( returnType == null ) {
      String fmsg = XSLMessages.createXPATHMessage(
        XPATHErrorResources.ER_ARG_CANNOT_BE_NULL,
        new Object[] {"returnType"} );
      throw new NullPointerException ( fmsg );
    }
    // Checking if requested returnType is supported.
returnType need to
    // be defined in XPathConstants
    if ( !isSupported ( returnType ) ) {
      String fmsg = XSLMessages.createXPATHMessage(

 XPATHErrorResources.ER_UNSUPPORTED_RETURN_TYPE,
        new Object[] { returnType.toString() } );
      throw new IllegalArgumentException ( fmsg );
    }

    try {
```

```
      // 核心业务代码
      XObject resultObject = eval( expression, item );
      return getResultAsType( resultObject, returnType
);
    } catch ( java.lang.NullPointerException npe ) {
      // If VariableResolver returns null Or if we get
      // NullPointerException at this stage for some
other reason
      // then we have to reurn XPathException
      throw new XPathExpressionException ( npe );
    } catch ( javax.xml.transform.TransformerException
te ) {
      Throwable nestedException = te.getException();
      if ( nestedException instanceof
javax.xml.xpath.XPathFunctionException ) {
        throw
(javax.xml.xpath.XPathFunctionException)nestedExceptio
n;
      } else {
        // For any other exceptions we need to throw
        // XPathExpressionException ( as per spec )
        throw new XPathExpressionException ( te );
      }
    }
  }
}
```

- 调用解析配置文件的方法 parseConfiguration(XNode root) 对上一步得到的xnode对象进行解析

```
public class XMLConfigBuilder extends BaseBuilder{
  private boolean parsed;
  private XPathParser parser;
  private String environment;
  private ReflectorFactory localReflectorFactory;
  private void parseConfiguration(XNode root) {
    try {
      Properties settings =
this.settingsAsPropertiess(root.evalNode("settings"));

 this.propertiesElement(root.evalNode("properties"));
```

```
10          this.loadCustomVfs(settings);
11
 this.typeAliasesElement(root.evalNode("typeAliases"))
;
12          this.pluginElement(root.evalNode("plugins"));
13
 this.objectFactoryElement(root.evalNode("objectFactor
y"));
14
 this.objectWrapperFactoryElement(root.evalNode("objec
tWrapperFactory"));
15
 this.reflectorFactoryElement(root.evalNode("reflector
Factory"));
16          this.settingsElement(settings);
17
 this.environmentsElement(root.evalNode("environments"
));
18
 this.databaseIdProviderElement(root.evalNode("databas
eIdProvider"));
19
 this.typeHandlerElement(root.evalNode("typeHandlers")
);
20          this.mapperElement(root.evalNode("mappers"));
21      } catch (Exception var3) {
22          throw new BuilderException("Error parsing SQL
Mapper Configuration. Cause: " + var3, var3);
23      }
24    }
25  }
```

- (以xml配置文件中mapper标签举例)通过调用mapperElement()方法，将mapper中的接口对象（ Class<?> mapperInterface ）放入configuration对象中

```
1  private void mapperElement(XNode parent) throws
Exception {
2    if (parent != null) {
3      Iterator i$ = parent.getChildren().iterator();
4
```

```
    while(true) {
      while(i$.hasNext()) {
        XNode child = (XNode)i$.next();
        String resource;
        if ("package".equals(child.getName())) {
          resource = child.getStringAttribute("name");
          this.configuration.addMappers(resource);
        } else {
          resource =
child.getStringAttribute("resource");
          String url =
child.getStringAttribute("url");
          String mapperClass =
child.getStringAttribute("class");
          XMLMapperBuilder mapperParser;
          InputStream inputStream;
          if (resource != null && url == null &&
mapperClass == null) {

 ErrorContext.instance().resource(resource);
            inputStream =
Resources.getResourceAsStream(resource);
            mapperParser = new
XMLMapperBuilder(inputStream, this.configuration,
resource, this.configuration.getSqlFragments());
            mapperParser.parse();
          } else if (resource == null && url != null
&& mapperClass == null) {
            ErrorContext.instance().resource(url);
            inputStream =
Resources.getUrlAsStream(url);
            mapperParser = new
XMLMapperBuilder(inputStream, this.configuration, url,
this.configuration.getSqlFragments());
            mapperParser.parse();
          } else {
            if (resource != null || url != null ||
mapperClass == null) {
              throw new BuilderException("A mapper
element may only specify a url, resource or class, but
not more than one.");
```

```
            }

            Class<?> mapperInterface =
Resources.classForName(mapperClass);

 this.configuration.addMapper(mapperInterface);
          }
        }
      }
      return;
    }
  }
}
```

- 将装配完成的Configuration对象，利用this.build( ) 方法，装配为SqlSessionFactory对象

```
public SqlSessionFactory build(Configuration config) {
  return new DefaultSqlSessionFactory(config);
}
```

- 可以得到一个 SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(in);对象

从SqlSessionFactory 对象得到一个SqlSession对象

- 调用SqlSessionFactory(接口)对象的openSession()方法，SqlSessionFactory 是一个 接口，调用的是DefaultSqlSessionFactory()实现类

```
public class DefaultSqlSessionFactory implements
SqlSessionFactory {
  private final Configuration configuration;
  // 构造器，SqlSessionFactory对象中的configuration已经赋值
了
  public DefaultSqlSessionFactory(Configuration
configuration) {
    this.configuration = configuration;
  }

```

```
  private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level, boolean autoCommit) {
    Transaction tx = null;

    DefaultSqlSession var8;
    try {
      Environment environment = this.configuration.getEnvironment();
      TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);
      tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
      Executor executor = this.configuration.newExecutor(tx, execType);
      // 建立一个DefaultSqlSession对象，将configuration传入
      var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
    } catch (Exception var12) {
      this.closeTransaction(tx);
      throw ExceptionFactory.wrapException("Error opening session.  Cause: " + var12, var12);
    } finally {
      ErrorContext.instance().reset();
    }
    return var8;
  }
}
```

- 在DefaultSqlSessionFactory对象中通过openSessionFromDataSource方法创建了一个DefaultSqlSession对象，并将其返回

从SqlSession(接口)对象中执行getMapper()方法

SqlSession是一个接口，调用的是实现类(DefaultSqlSession)中的getMapper方法

```
```

```
public class DefaultSqlSession implements SqlSession {
  private Configuration configuration;
  private Executor executor;
  private boolean autoCommit;
  private boolean dirty;
  private List<Cursor<?>> cursorList;
  // 上一步openSessionFromDataSource通过该构造生成了一个SqlSession对象
  public DefaultSqlSession(Configuration configuration, Executor executor, boolean autoCommit) {
    this.configuration = configuration;
    this.executor = executor;
    this.dirty = false;
    this.autoCommit = autoCommit;
  }

  public DefaultSqlSession(Configuration configuration, Executor executor) {
    this(configuration, executor, false);
  }
  public <T> T getMapper(Class<T> type) {
    return this.configuration.getMapper(type, this);
  }
}
```

- getMapper方法返回 的是configuration对象中的getMapper()方法的返回值，传递的参数 是 ：类对象class,和this(当前的sqlSession对象)

```
public class Configuration {
  public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
    return this.mapperRegistry.getMapper(type, sqlSession);
  }
}
```

- 调用的是mapperRegistry对象中getMapper方法

```
public class MapperRegistry {
  private final Configuration config;
```

```
  private final Map<Class<?>, MapperProxyFactory<?>>
knownMappers = new HashMap();

  public MapperRegistry(Configuration config) {
    this.config = config;
  }
  public <T> T getMapper(Class<T> type, SqlSession
sqlSession) {
    MapperProxyFactory<T> mapperProxyFactory =
(MapperProxyFactory)this.knownMappers.get(type);
    if (mapperProxyFactory == null) {
      throw new BindingException("Type " + type + " is
not known to the MapperRegistry.");
    } else {
      try {
        return
mapperProxyFactory.newInstance(sqlSession);
      } catch (Exception var5) {
        throw new BindingException("Error getting
mapper instance. Cause: " + var5, var5);
      }
    }
  }
}
```

- MapperRegistry对象中的getMapper方法执行的是mapperProxyFactory对象中的newInstance方法

```
public class MapperProxyFactory<T> {
  public T newInstance(SqlSession sqlSession) {
  MapperProxy<T> mapperProxy = new
MapperProxy(sqlSession, this.mapperInterface,
this.methodCache);
  return this.newInstance(mapperProxy);
  }
}
```

- 创建了一个mapperProxy对象(mapperProxy)，MapperProxy对象，实现了InvocationHandler接口

```

```

```java
public class MapperProxy<T> implements InvocationHandler, Serializable {
  private static final long serialVersionUID = -6424540398559729838L;
  private final SqlSession sqlSession;
  private final Class<T> mapperInterface;
  private final Map<Method, MapperMethod> methodCache;

  public MapperProxy(SqlSession sqlSession, Class<T> mapperInterface, Map<Method, MapperMethod> methodCache) {
    this.sqlSession = sqlSession;
    this.mapperInterface = mapperInterface;
    this.methodCache = methodCache;
  }

  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if (Object.class.equals(method.getDeclaringClass())) {
      try {
        return method.invoke(this, args);
      } catch (Throwable var5) {
        throw ExceptionUtil.unwrapThrowable(var5);
      }
    } else {
      MapperMethod mapperMethod = this.cachedMapperMethod(method);
      return mapperMethod.execute(this.sqlSession, args);
    }
  }

  private MapperMethod cachedMapperMethod(Method method) {
    MapperMethod mapperMethod = (MapperMethod)this.methodCache.get(method);
    if (mapperMethod == null) {
      mapperMethod = new MapperMethod(this.mapperInterface, method, this.sqlSession.getConfiguration());
```

```
30          this.methodCache.put(method, mapperMethod);
31        }
32
33        return mapperMethod;
34      }
35    }
```

- mapperProxyFactory对象中的newInstance方法执行了MapperProxy构造，创建了一个mapperProxy对象(mapperProxy)，将其作为参数，this.newInstance(mapperProxy);

```
1  public class MapperProxyFactory<T> {
2    protected T newInstance(MapperProxy<T> mapperProxy) {
3      return
   Proxy.newProxyInstance(this.mapperInterface.getClassLoa
   der(), new Class[]{this.mapperInterface}, mapperProxy);
4    }
5  }
```

- 通过工具类Proxy中的newProxyInstance方法，建立了一个动态代理对象，并将其返回。

老师讲的

```
1  new Configuration(){
2      Properties valiableus;//放的是settings配置
3      Map<String,Class> typeAlias;//放的是别名配置
4      Map<String,MappedStatent> mappedStataments;//放的是
   映射文件的select|insert|update|delte
5      Map<String,ResultMap> resultMaps;//放的是映射文件中的
   resultMap标签的解析
6  }
7
8  new XMLConfigBuilder(InputStream in);//解析配置文件,new
   Configuration()
9  Configuration config = XMLConfigBuilder.parse();//向
   Configuration对象填充数据
10 SqlSessionFactory factory = new
   DefaultSqlSessionFactory(config);//共产模式的应用
```

```
11
12
13 new XMLMapperBuilder(InputStream in,config);//解析映射文
   件
14 XMLMapperBuilder.parse()
15
16
17 new XMLStatementBuilder(config,"namespace","select标
   签")
18 XMLStatementBuilder.parseStatementNode()
```

```
1 XMLConfigBuilder parser = new
  XMLConfigBuilder(inputStream, environment, properties);
2 var5 = this.build(parser.parse());
3 this.parseConfiguration(this.parser.evalNode("/configur
  ation"));
4 private void environmentsElement(XNode context) throws
  Exception {
5 Builder environmentBuilder = (new
  Builder(id)).transactionFactory(txFactory).dataSource(d
  ataSource);}
6 private void mapperElement(XNode parent) throws
  Exception {
7
8 }
```

```
1 inputStream = Resources.getResourceAsStream(resource);
2 mapperParser = new XMLMapperBuilder(inputStream,
  this.configuration, resource,
  this.configuration.getSqlFragments());
3 mapperParser.parse();
4 this.configurationElement(this.parser.evalNode("/mappe
  r"));
5 this.parameterMapElement(context.evalNodes("/mapper/pa
  rameterMap"));
6
   this.resultMapElements(context.evalNodes("/mapper/res
  ultMap"));
```

```
 this.sqlElement(context.evalNodes("/mapper/sql"));

 this.buildStatementFromContext(context.evalNodes("select|insert|update|delete"));

private void buildStatementFromContext(List<XNode> list, String requiredDatabaseId){
  while(i$.hasNext()) {
    XNode context = (XNode)i$.next();
    XMLStatementBuilder statementParser = new XMLStatementBuilder(this.configuration, this.builderAssistant, context, requiredDatabaseId);
    try {
      statementParser.parseStatementNode();
    } catch (IncompleteElementException var7) {

 this.configuration.addIncompleteStatement(statementParser);
    }
  }

}

MappedStatement statement = statementBuilder.build();
this.configuration.addMappedStatement(statement);
id=com.javasm.mapper.SysUserMapper.addObj

```

```
var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
Environment environment = this.configuration.getEnvironment();
            TransactionFactory transactionFactory = this.getTransactionFactoryFromEnvironment(environment);
            tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
            Executor executor = this.configuration.newExecutor(tx, execType);
            var8 = new DefaultSqlSession(this.configuration, executor, autoCommit);
```

构建者模式:构建复杂对象的对象.该对象的职责就是用来构建另外一个单例对象.

XXXBuilder{

new duixiang()

build(){}

parse(){}

}

工厂模式:构建复杂对象,该对象的职责时用来构建一系列的对象.

XXXFactory(){

对象 create(){}

对象 parse(){}

}

//不把对象的new的过程,散乱在代码不同位置.而统一放在工厂类或构建器类中来创建对象.

3. getMapper方法的执行原理

getMapper方法,返回的是接口的代理对象(接口的实现类实例化对象$Proxy8)

```
//回调处理器对象,该对象内的invoke方法会在代理对象的方法执行时被调用.
class MapperProxy implements InvocationHandler{

  public Object invoke(Object proxy,Method method,Object[] args){

  }
}
```

```
public T newInstance(SqlSession sqlSession) {
  MapperProxy<T> mapperProxy = new MapperProxy(sqlSession, this.mapperInterface, this.methodCache);
  //创建$Proxy1代理类,并实例化,代理对象
  return
 Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]{this.mapperInterface}, mapperProxy);
}
```

总结:

认识构建器模式,工厂模式;

认识Configuration对象;

认识MappedStatement对象;

认识代理模式.了解getMapper方法内返回的到底是什么对象

day04 Spring基础

1. 什么是spring

spring是一个第三方的容器框架，用来管理对象

- 是反射工厂模式的应用
- 提供了很多的工具类：RestTemplate jdbc Template，RedisTemplate....
- spring框架提供了管理器对象：事务管理器对象
- spring提供了线程池的支持，支持异步任务，支持定时任务

servlet容器：httpservletrequest session servletContext

ServletContext:全局上下文对象，存储自定义对象，存储到该对象中的数据能够被所有的请求所共享

spring带来的最大好处：

- 解耦：下层变动-----不影响当前层，上层只依赖下层的抽象接口，不依赖其具体的实现类。

2. 入门使用

- 添加spring的4个核心包，一个commons-logging日志包，共5个jar包
- 创建spring风格的xml配置文件

```
  <bean id="sysuserController"
class="com.javasm.controller.SysuserController"></bean>
```

- 加载xml文件，实例化spring容器对象

```
  //1. 加载类路径下的xml文件，初始化BeanFactory工厂对象，创建一个bean对象
  AplicationContext ac = new
ClassPathXmlApplicationContext(spring.xml))
```

- 测试使用

```
//1. 加载类路径下的xml文件，初始化BeanFactory工厂对象，创建一个bean对象
AplicationContext ac = new ClassPathXmlApplicationContext(spring.xml))
//2. 根据id从容器中获取bean对象
(SysuserController)ac.getBean("sysuserController")
//3. 根据类型从容器中获取bean
SysuserController uc2 = ac.getBean(SysuserCOntroller.class)
```

总结

- spring管理的对象，默认是单例的
- 从spring中获取对象，有两种方式，byName,byType
- 有了spring后，除了实体类，以后尽量不再new对象，是把对象注册到spring容器，然后需要用的时候，从容器中get。

Spring容器的类图

最顶级的接口BeanFactory----->DefaultListableBeanFactory

BeanFactory----->容器接口ApplicationContext的实现类

FileSystemXmlApplicationContext 加载系统文件

ClasspathXmlApplicationContext 从类路径中寻找

AnnotationConfigApplicationContext

XmlWebApplicationContext

AnnocationConfigWebApplicationContext

3. spring中的几个概念

- IOC:控制反转，对象的管理由自身维护交给第三方的容器进行管理，称为控制反转

```
<!--spring通过反射SysuserServiceImpl对象-->
<bean id="sysuserService" class="com.javasm.service.impl.SysuserServiceImpl">
</bean>
```

- DI: 依赖注入：bean对象依赖谁，对象之间的关系也由spring容器进行管理

```
<bean id="sysuserController"
class="com.javasm.controller.SysuserController">
  <!--SysuserController依赖的us成员变量,也在容器,通过
property标签进行set注入-->
  <property name="us" ref="sysuserService"></property>
  <property name="str" value="aaa123"></property>
</bean>
```

5. bean标签属性

- id: bean的名称
- class：bean的类名
- scope: 对象的单例多例状态，singleton|prototype 默认是singleton单例。。单例模式会先进行实例化，多例模式不进行实例化，只有在调用的时候才会创建对象，且两次调用的对象非同一个对象
- lazy-init:true （懒汉单例模式）仍然是单例状态，延迟初始化， 默认是饿汉式单例模式

6. 指定bean对象的初始化方法

方法1：对jar包中的类，使用bean标签的init-method方法

```
一般用在jar包中的注册spring容器时,指定初始化与销毁的方法
init-method="init" ：  指定bean对象的初始化方法（使用数据库连接池进行举例，在无法更改导入的jar文件下，如何实现创建对象时的初始化工作）
destroy-method="close": 指定bean对象的销毁方法-----只有显示执行了ApplicaiotnContext的close方法才能够触发destroy-method。
```

方法2：对自定义的类，使用InitializingBean接口

```
如果自定义的类注册spring容器,需要指定初始化与销毁方法,可以通过InitializingBean接口实现.
进行方法(afterPropertiesSet)的重写------->spring会检测 配置的类是否实现了InitializingBean接口，若有实现则自动执行afterPropertiesSet方法进行初始化，（无法做销毁工作）
public class SysuserServiceImpl implements ISysuserSerivce,InitializingBean {
    //当该bean被实例化完成后,立即执行afterPropertiesSet方法进行初始化工作.
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("初始化方法.......");
    }
}
```

如果同时定义了两种初始化的方法则默认先走实现接口时重写的afterPropertiesSet()方法

7. ioc的实现方式（重要）

IOC: 把对象注册到spring容器中

方法1：bean标签，指定类名的方式进行bean注册，常用于jar包中的类注册到容器中使用

```
 <bean class="com.javasm.service.impl.SysuserServiceImpl">
</bean>
```

方法2：工厂注册bean：复杂对象注册到spring容器中,可以使用工厂注册bean

应用举例：将SqlSessionFactory对象注册到spring容器中

- 静态工厂方法：在标签属性中添加factory-method="静态方法名" ： 将类中的一个静态方法的返回值注册到容器中

  注意：并没有把SessionFactoryBean类注册到容器中，因此如果输出，则会有NoSuchBeanDefinitionException: No qualifying bean of type 异常

```
<!--静态工厂注册bean-->
<!--把SessionFactoryBean.createFactory方法的返回值注册到容器-->
<bean id="sqlSessionFactory" class="com.javasm.factory.SessionFactoryBean" factory-method="createFactory"></bean>
```

- 实例工厂方法：将类中的非静态方法的返回值注册到容器中

  注意，该方法同时将SessionFactoryBean2注册到了spring容器中，因此可以使用该对象

  ```
  <!--工厂对象-->
  <bean id="factoryBean2" class="com.javasm.factory.SessionFactoryBean2" >
  </bean>
  <!--factoryBean2.createFactory()-->
  <!--先注册factoryBean2这个类，然后通过factory-bean获取类的对象，然后经过actory-method将实例方法的返回值(对象)注册到spring中-->
  <bean id="sqlSessionFactory" factory-bean="factoryBean2" factory-method="createFactory">
  </bean>
  ```

方法3：**包扫描（自定义类的容器注册）**

```
对包下的所有类进行递归扫描，通过反射检测类上是否有指定注解，有的话将该类注册到容器中
特定的注解：
@Controller:注解控制层的bean
    @Scope("prototype")：指定多例状态，默认是单例（用在类或方法上）
    @PostConstruct：指定对象的初始化方法（用在方法上）
    @PreDestroy：指定对象的销毁方法（用在方法上）
@Service：注解服务层的bean
@Repository：注解dao层的bean
@Component：注解其他的bean
<context:component-scan base-package="com.javasm">
</context:component-scan>
```

包扫描的注意事项

注意点1：context:component-scan不仅仅识别这四个注解，后续学习其它。

注意点2：通过注解方式注册的bean，默认id是类名首字母小写;可以自定义id

注意点3：这四个注解当前是没有区别，效果都是注册bean到spring容器，后续springMVC框架有 区别，我们需要有好的代码习惯，分别注解各层的bean对象。

注意点4：不能注解到接口类上。

5. di的实现方式（重要）

---

dl：**依赖注入，维护对象之间的依赖关系**

Set注入：调用bean对象中的set方法注入依赖值。

构造器注入：实例化bean对象时，调用有参构造注入依赖值。

集合注入：bean对象中的Array，List，Map等集合属性注入值。

内部bean注入：本质仍然是set注入或构造器注入。

自动装配：基于包扫描bean配置。

方法1：set注入，要求成员变量必须有set和get方法

标签中ref|value:

ref：引用其他bean的id值

value是用来给简单类型赋值。

另一种set注入方法：引入p命名空间，在bean节点中以“p:属性名=属性值”的方式为属性注入值。

```
<bean id="ds"
class="com.alibaba.druid.pool.DruidDataSource" init-
method="init" destroy-method="close">
  <!--set注入,调用对象的set方法进行值得注入;ref|value:ref用来
引用其他bean的id,value用来给简单类型赋值-->
  <property name="url"
value="jdbc:mysql://localhost:3306/crm"></property>
  <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
  <property name="username" value="root"></property>
  <property name="password" value="root"></property>
  <property name="initialSize" value="5"></property>
</bean>

另一种方式：通过 引入p命名空间
<bean id="ds"
class="com.alibaba.druid.pool.DruidDataSource"
p:name="url">
<bean id="ds"
class="com.alibaba.druid.pool.DruidDataSource"
p:user_ref="user">
```

方法2：构造器注入()

```
通过反射：获取其有参构造
index:形参索引
name: 形参名
value:形参的值(简单类型)
ref:对象类型的参数(其他注入对象的bean的id值)
<!--new SysuserDaoImpl(ds,12)-->
<bean class="com.javasm.dao.impl.SysuserDaoImpl">
  <!--index:形参索引;name:形参名-->
  <constructor-arg index="0" ref="ds"></constructor-
arg>
  <constructor-arg name="sint" value="12">
</constructor-arg>
</bean>
```

方法3：集合注入

```
// 需要初始化的参数
```

```java
public class SysuserDaoImpl implements ISysuserDao {
    private DataSource ds;
    private Integer sint;
    private List<String> strList;
    private Integer[] ints;
    private Map<String,Double> douMap;
    private Sysuser suser;
   public SysuserDaoImpl(DataSource ds,Integer sint) {
        this.ds = ds;
        this.sint=sint;
    }
   public Sysuser getSuser() {
        return suser;
    }

    public void setSuser(Sysuser suser) {
        this.suser = suser;
    }
}
public class Sysuser {
    private String uname;
    private String upwd;
}
```

```xml
// di 通过集合注入的方法
<!--new SysuserDaoImpl(ds,12)-->
<bean class="com.javasm.dao.impl.SysuserDaoImpl">
  <!--index:形参索引;name:形参名-->
  <constructor-arg index="0" ref="ds"></constructor-arg>
  <constructor-arg name="sint" value="12"></constructor-arg>
  <property name="strList">
    <list>
      <value>aaa</value>
      <value>bbb</value>
    </list>
  </property>
  <property name="ints">
    <array>
      <value>11</value>
```

```
        <value>22</value>
      </array>
    </property>
    <property name="douMap">
      <map>
        <entry key="a" value="1.2"></entry>
        <entry key="b" value="1.3"></entry>
      </map>
    </property>
  </bean>
```

方法4：内部bean注入

```
<bean class="com.javasm.dao.impl.SysuserDaoImpl">
  <property name="suser">
  此标签放在proerty内部，该sysuser这个类不会被引入到spring容器中。
       <bean class="com.javasm.entity.Sysuser"></bean>
  </property>
</bean>
```

方法5：**自动装配（重要）**

在开启了包扫描后，可以通过注解(@AutoWired|@Resource)给成员变量赋值（dI自动赋值）

注意点1：在java代码中使用@Autowired或@Resource注解方式进行装配，这两个注解的区别是：

@Autowired 默认按类型装配，类型匹配不上，再按照形参名称装配。

@Resource默认按形参名称装配，当找不到与名称匹配的bean才会按类型装配。

注意点2：Resource注解可以指定名称@Resource(name="userService"),指定后则只能按照名称进行装配，一般没有必要。

- 如果接口有多个实现类则会报异常：因为spring不知道该找哪个实现类进行装配（因为在定义成员变量时一般都是定义的接口类型 比如：Private ISysuserService user--------> 如果ISysuserService有两个实现类，则会报异常，因为不知道需要赋值的是哪个实现类）----------------> 产生的异常：beans.factory.NoUniqueBeanDefinitionException

- 如果该成员变量（类对象）没有在spring容器中，则会报错。

@AutoWired: 先byType 再byName

@Resource:先byName再byType(一般用这个)

9.xml文件的补充

```
通过import标签可以导入另外的xml文件中的id名为""的类--->xml之间的嵌套
<import resource="dao.xml"></import>
通过context:property-placeholder标签可以引入其他配置文件中，将其他配置文件中的数据引入到spring容器中
context:property-placeholder标签的ignore-unresolvable属性是当引入了多个properties文件时必须指定的属性。
<context:property-placeholder location="jdbc.properties"></context:property-placeholder>
引入的方式，通过${key名}
<bean id="ds" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close">
  <property name="url" value="${jdbc.url}"></property>
  <property name="driverClassName" value="${jdbc.driver}"></property>
  <property name="username" value="${jdbc.username}"></property>
  <property name="password" value="${jdbc.password}"></property>
  <property name="initialSize" value="${jdbc.initialSize}"></property>
</bean>
```

10 反射工厂模式的应用

简单静态工厂：这种工厂没有可扩展性

反射工厂：基于反射+配置的方式提高代码的可扩展性，

- 定义配置 文件（名字type：全类名）
- ，通过property对象，将配置文件的信息导入到property对象中

- p.getProperty(type)获取全类名clzName
- 通过clz = Class.forName(clzName)获取type对应的类对象
- clz.newInstance()，获取该类型所对应的对象

day05_6_11_Spring的AOP

1. junit与spring的整合使用

spring中的所有jar包必须版本一致

将测试类注册到spring容器中，需要测试哪个对象，注入哪个对象

- 添加spring-test-5.3.4 的jar包
- 加载xml初始化容器，并把当前测试类注册到容器

  @RunWith(SpringJUnit4ClassRunner.class)
- classpath：绝对路径的写法，表示从根路径开始查找

  @ContextConfiguration("classpath:DITest.xml")

```
//加载xml初始化容器,并把当前测试类注册容器
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:spring.xml")//
classpath:绝对路径的写法,表示从根路径开始查找spring.xml
public class TestApplicationContext {

    @Resource
    private SysuserController uc;

    @Resource
    private ISysuserService us;

    @Test
    public void test1_ac(){
        System.out.println(uc);
    }

    @Test
    public void test2_us(){
        System.out.println(us);
    }
}
```

2. 什么是aop

aop：面向切面编程，在运行期间通过代理实现程序功能统一维护的一种技术，AOP是oop思想的扩展，利用aop可以对业务逻辑的各个部分进行隔离，从而使业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

通过oop面向对象编程实现了业务逻辑，对于后期需要添加的辅助性业务需求(日志，参数校验)使用aop思想进行扩展，辅助性的业务就是切面。

使用AOP利用一个代理将原业务对象包装起来，然后用该代理对象取代原始对象，可以在代理对象中对核心的业务逻辑进行扩展

aop中的几个概念：

切面（aspect）：辅助业务的业务对象，比如：日志，参数校验（切面类）

通知(advice)：切面类中的方法，分为5类：前置通知（方法，参数），返回通知（返回值），最终通知，异常通知（异常信息），环绕通知。

织入(weave)：在程序运行期间，通过动态代理模式，给目标对象创建代理对象，并把切面中的通知方法 织入到指定的连接点方法的前、后、异常、最终。

目标对象(target)：核心业务对象（需要扩展的业务对象）

连接点(joinPoint)：核心业务方法，目标对象中的某个业务方法

切入点(pointCut)：连接点的集合，通过一个切入点表达式可以灵活的定义多个连接点。

3. 通过动态代理实现aop

```
1  public static void main(String[] args) {
2    //被代理对象,该对象的pay方法需要扩展,添加日志,效率监测,参数校
   验.
3    IPay p = new AliPay();
4    IPay p2 = createProxy(p);//$Proxy100
5    p2.pay("aa","bb",11.1);//p.pay()
```

```
}

public static IPay createProxy(IPay p){
  //p:目标对象
  ClassLoader loader= p.getClass().getClassLoader();
  Class[] interfaces = new Class[]{IPay.class};
  InvocationHandler handler = new InvocationHandler()
{
    @Override
    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
      //proxy:代理对象
      String name = method.getName();
      if("pay".equals(name)){
        Object obj = null;
        //切面
        MyAspect a = new MyAspect();
        try{
          a.beforeAdvice();//通知
          obj = method.invoke(p,args);//执行连接点方法
          a.afterReturnAdvice();
        }catch (Exception e){
          a.exceptionAdvice();
        }finally {
          a.afterAdvice();
        }
        return obj;
      }

      return method.invoke(p,args);//p.toString()
    }
  };

  // Object obj = new $Proxy100(handler);
  //IPay proxy = (IPay)obj;
  IPay proxy = (IPay)
Proxy.newProxyInstance(loader,interfaces,handler);
  return proxy;
}
```

4. spring的aop注解使用

1. 添加spring-aop spring-aspect两个依赖包;spring的aop依赖于aspect第三方组建
2. 添加aspectj组件的三个依赖包
3. 添加cglib一个依赖包
4. 在spring的xml配置文件中aop注解识别
5. 创建切面类，创建通知方法

```
proxy-target-class="false"(默认) 默认使用Proxy类，对目标对象创建代理，如果目标对象没有接口，则采用cglib创建
proxy-target-class="true": 采用cglib创建代理
aspect组建的注解识别 @Aspect Pointcut Before AfterReturning AfterThrowing After
Before("")
After()
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

注解的使用方法

- 切入点注解@Pointcut("execution(* login(..))")
  - ```
    @Pointcut("execution(* login(..))")
    public void pc(){}
    ```
  -
  - execution（返回值类型，包名.类名.方法名(形参类型)），包名类名可以省略。
  - *:通配符
  - .. 不限制形参类型
- @Before("pc()")
    ```

```
@Before("pc()")
public void beforeAdvice(JoinPoint jb){
  jb.getArgs();//方法实参
  jb.getRarget();//源对象
  jb.getThis(); // 代理对象
 MethodSignature signature =（MethodSignature）
jp.getSingature()
 Method method = signature.getMethod();//获取方法
}
```

- @AfterReturning

```
@AfterReturning(poincut="pc()",returning="o")
public void AfterReturnAdive(Joinpoint jp,Object
o){

}
```

- @AfterThrowing

```
@AfterReturning(poincut="pc()",throwing="e")
public void AfterReturnAdive(Joinpoint
jp,Exception e){
  System.out.println(e)
}
```

- @After("pc()")
- 环绕通知

```
// 返回值必须是Object
// 形参必须是ProceedingJoinPoing
@Around("pc()")
public Object aroundAdvice(ProceedingJoinPoing
jp){
  Object result = null;
  //Object result = method.invoke(target,args)
  try{
    // 前置通知
    System.out.println("前置通知");
    //result = method.invoke(target,args)
    result = jp.proceed();
```

```
        // 返回通知
        System.out.println("返回通知");
    }catch(
        // 异常通知
        System.out.println("异常通知");
    )finally{
        // 最终通知
        System.out.println("最终通知");
    }
}
```

5. 切入点表达式

```
//切入点表达式两种写法:
//方法1:
@Pointcut(execution(返回值类型  包名.类名.方法名(形参类型))-----包名.类名可以省略
//@Pointcut("execution(* com.javasm.service.impl.*.*(..))")
// 方法2:自定义注解,@annotation(注解类名),表示带有指定注解的方法全部时连接点方法
// 第二种方式
@Pointcut("@annotation(com.javasm.annotation.tx)")
public void pc(){}
```

6. spring中的aop配置使用

```
<bean id="logAspect2"
class="com.javasm.aspect.LogAspect2"></bean>

<aop:config>
  <aop:aspect ref="logAspect2">
    <aop:pointcut id="pc"
expression="@annotation(com.javasm.annotation.tx)">
</aop:pointcut>
    <aop:around method="aroundAdvice" pointcut-
ref="pc"></aop:around>
    <!--<aop:before method="beforeAdvice" pointcut-
ref="pc"></aop:before>-->
    <!--<aop:after-returning
method="afterReturnAdvice" pointcut-ref="pc"
returning="o"></aop:after-returning>-->
    <!--<aop:after-throwing method="exceptionAdvice"
pointcut-ref="pc" throwing="e"></aop:after-throwing>--
>
    <!--<aop:after method="afterAdvice" pointcut-
ref="pc"></aop:after>-->
  </aop:aspect>
</aop:config>
</bean>
```

7. dom4j解析xml文件

Properties类解析.properties文件

Dom4j组件解析xml文件,pull组件解析xml文件.

中间件开发,必须需要进行xml解析

- 添加dom4j.jar
- Document doc = SAXReader.read(InputStream)
- Element root = doc.getRootElement()
- String str =root.attributeValue("属性名")
- List<Element> tags = root.elements("标签名")

8.easycode插件

- 安装插件

- 配置Type Mapper
- 配置Tmeplate settings

day06_6_12 spring高级

1. 通过事务切面学习aop

事务管理器对像：MyTransactinManager:是一个管理器工具对象，用来打开，关闭，回滚链接

事务切面：TxAspect：把通知织入带有Tx注解的连接点方法

THreadLocal对象：线程变量对象，在多线程的情况下，确保线程安全的做法

ApplicationContextAware接口：用来获取spring容器引用的接口，需要重写setApplicationContext

方法

- 解决静态类对象中获取容器中的对象的问题
- 在子线程下怎么获取容器中的对象

```
@Component
public class SpringUtil implements
ApplicationContextAware {
    private static ApplicationContext
applicationContext;//静态被所有实例共享

    @Override
    public void
setApplicationContext(ApplicationContext ac) throws
BeansException {
        applicationContext=ac;
    }

    public static <T> T getBean(Class<T> clz){
        return applicationContext.getBean(clz);
    }
}
```

2. 学习spring的AnnotationConfigApplicationContext容器对象

BeanFactory----->DefaultListableBeanFactory

BeanFactory------>ApplicationConqtext-----
>ClassPathXMLApplicationContext

AnnotationConfigApplicationContext

- ClassPathXMLApplicationContext 加载xml配置文件中的.xml配置的方式应用在ssm框架中
- AnnotationConfigApplicationtext加载class类配置文件的，类配置文件的方式应用在springboot框架中
- 类配置的常用的注解

```
@Configuration // 定义配置类
@Bean //注册bean
@PropertySource // 加载properties文件，把配置数据注册进容器中
@ComponentScan  // 开启包扫描
@EnableAspectjAutoProxy // 开启aspectj注解识别
@Import // 引入其他的配置类
@Value  // 获取容器中的properties配置数据
@ImportResource  // 引入其他xml配置文件
-------------------------------------------------------
//基础的配置类
@Configuration//表示当前类是一个配置类,同时该类会被注册容器
@ComponentScan("com.javasm")//<context:component-scan>
@EnableAspectJAutoProxy//开启aop注解识别
@Import(DaoConfig.class)
@ImportResource("classpath:dao.xml")
public class AppConfig {

    //先按照形参名注入,再按照形参类型注入值
    @Bean
    public SqlDaoImpl createUserDao(DataSource dataSource){
```

```
        SqlDaoImpl sysuserDao = new SqlDaoImpl();
        sysuserDao.setDataSource(dataSource);
        return  sysuserDao;
    }
}
-----------------------------------------------------------
//jdbc数据库连接的配置文件
@Configuration
@PropertySource("classpath:jdbc.properties")//context:property-placeholder
public class DaoConfig {
    @Value("${jdbc.url}")
    private String url;
    @Value("${jdbc.driverClassName}")
    private String driverClassname;
    @Value("${jdbc.username}")
    private String username;
    @Value("${jdbc.password}")
    private String password;
    @Value("${jdbc.initialSize}")
    private Integer initSize;

    //把方法的返回值注册容器,id默认是方法名
    @Bean(initMethod = "init",destroyMethod = "close")
    public DataSource createDruidDataSource(){
        DruidDataSource druidDataSource = new DruidDataSource();
        druidDataSource.setUrl(url);

 druidDataSource.setDriverClassName(driverClassname);
        druidDataSource.setUsername(username);
        druidDataSource.setPassword(password);
        druidDataSource.setInitialSize(initSize);
        return druidDataSource;
    }
}
```

3. 写下mybatis的几张表的操作