目录

[系统分层 2](#_Toc500907223)

[MVC 2](#_Toc500907224)

[Spring优点 2](#_Toc500907225)

[Spring容器 2](#_Toc500907226)

[启动Spring容器 3](#_Toc500907227)

[Spring创建Bean对象 3](#_Toc500907228)

[Spring IOC和DI 3](#_Toc500907229)

[Spring注入值 3](#_Toc500907230)

[Spring其它功能 4](#_Toc500907231)

[Spring基于注解的组件扫描 4](#_Toc500907232)

[Spring MVC 4](#_Toc500907233)

[基于注解的Spring MVC 5](#_Toc500907234)

[Spring读取请求参数 5](#_Toc500907235)

[Spring向页面传值 6](#_Toc500907236)

[Spring重定向 6](#_Toc500907237)

[Spring字符编码过滤器 6](#_Toc500907238)

[Spring拦截器 6](#_Toc500907239)

[Spring异常处理 6](#_Toc500907240)

[Spring JDBC 6](#_Toc500907241)

[MyBatis和jdbc， Hibernate 7](#_Toc500907242)

[MyBatis使用步骤和原理 7](#_Toc500907243)

[MyBatis配置文件 7](#_Toc500907244)

[MyBatis映射文件 8](#_Toc500907245)

[MyBatis API简介 8](#_Toc500907246)

[获取SqlSession对象 8](#_Toc500907247)

[Mapper映射器 8](#_Toc500907248)

[Spring与MyBatis整合(Mapper映射器) 8](#_Toc500907249)

[MapperScannerConfigurer 9](#_Toc500907250)

[Spring与MyBatis整合(不使用Mapper映射器) 9](#_Toc500907251)

[MyBatis动态SQL 9](#_Toc500907252)

[Spring对Ajax的支持 9](#_Toc500907253)

[AOP 9](#_Toc500907254)

[AOP基本概念 10](#_Toc500907255)

[AOP实现方式 10](#_Toc500907256)

[Spring AOP 10](#_Toc500907257)

[基于注解的Spring声明式事务管理 10](#_Toc500907258)

[resultType和resultMap 11](#_Toc500907259)

[Mybaits中"#"与"$"的区别 11](#_Toc500907260)

# Spring JDBC

|  |  |  |
| --- | --- | --- |
| Spring对jdbc的简单封装 | | |
| 步骤 | 导包 | spring-webmvc, spring-jdbc, ojdbc14, commons-dbcp, junit |
| 添加spring配置文件：加载jdbc源文件db.properties，定义数据源BasicDataSource | |
| 配置JdbcTemplate | <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">  <property name="dataSource" ref="ds" /> // ds是BasicDataSource的Bean |
| JdbcTemplate方法 | | <T> T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper) |
| <T> List<T> query(String sql, Object[] args, RowMapper<T> rowMapper) |
| int update(String sql, Object[] args) |
| void execute(String sql) |
| 写一个实现*RowMapper*，告诉JdbcTemplate是如何处理ResultSet的 | | class EmpRowMapper implements RowMapper<Employee> {  public Employee mapRow(ResultSet rs, int index) throws SQLException {  Employee e = new Employee();  e.setId(rs.getInt("id"));  ……  return e;} |

# MyBatis和jdbc， Hibernate

|  |  |  |
| --- | --- | --- |
| jdbc | MyBatis | Hibernate |
|  | 支持定制化SQL、存储过程以及高级映射的优秀的持久层框架…??? |  |
| 速度快，易掌握，要写sql，代码繁琐 | 速度适中，易掌握，要写sql，代码简洁 | 速度慢，比较难掌握，不用写sql，代码简洁 |

# MyBatis使用步骤和原理

|  |  |
| --- | --- |
| 导包mybatis |  |
| 添加mybatis配置文件 ??? |
| 实体类：属性名要与表字段名一样（忽略大小写） |
| 添加映射文件 |
| 修改配置文件，指定映射文件位置 |
| 调用MyBatis的API访问数据库 |

# MyBatis配置文件

|  |  |
| --- | --- |
| 引入dtd约束"http://ibatis.apache.org/dtd/ibatis-3-config.dtd" | |
| <configuration> | 根元素 |
| <properties resource="db.properties"> | 引入外部properties配置文件 |
| <environments default="development"> | 可配置多种环境，default指定使用某种环境 |
| <environment id="development"> | 配置一个具体的环境信息，必须有下面2个标签 |
| <transactionManager type="JDBC"/> | 事务管理器，type有两种取值，JDBC和managed |
| <dataSource type="POOLED">  <property name=name="driver" value="${driver}"/> | 数据源，tpye有三种取值，UNPOOLED|POOLED|JNDI |
| <mappers>  <mapper resource="Mapper.xml"/> | 指定映射文件位置;  注册一个sql映射文件 |

# MyBatis映射文件

|  |  |
| --- | --- |
| <mapper namespace="test"> | 根元素，namespace绑定DAO接口，写完全限定名 |
| id | SqlSession根据sqlId找到调用statement对应的方法 |
| parameterType | 传入参数 |
| resultType | 返回类型 |
| resultMap="empResultMap" | 返回类型 |
| <resultMap type="entity.Emp" id="empResultMap">  <result property="empNo" column="id"/> | 设置实体类的属性名与表的字段名的对应关系 |

# MyBatis API简介

|  |  |
| --- | --- |
| SqlSessionFactoryBuilder | 负责根据MyBatis配置文件构建SqlSessionFactory实例；方法范围 |
| SqlSessionFactory | 每一个MyBatis应用程序都以一个SqlSessionFactory对象为核心，该对象负责创建SqlSession对象实例；应用范围 |
| SqlSession | 包含了所有执行SQL操作的方法，由于执行已映射的SQL语句；请求或方法范围 |

# 获取SqlSession对象

|  |  |
| --- | --- |
| SqlSessionFactoryBuilder ssfb = new SqlSessionFactoryBuilder(); | |
| Reader is = Resources.getResourceAsReader("mybatis.xml"); 或者  InputStream is = 当前类.class.getClassLoader().getResourceAsStream("mybatis.xml"); ??? | |
| SqlSessionFactory ssf = ssfb.build(is); | |
| SqlSession Session = ssf.openSession(); | session.insert("sqlId", obj); session.commit(); |
| session.selectList("sqlId"); |
| session.selectOne("sqlId", obj); |
| session.update("sqlId", obj); |
| session.delete("sqlId", obj); |

# Mapper映射器

|  |  |
| --- | --- |
| 定义 | 一个符合映射文件要求的接口 |
| 要求 | 方法名 要与映射文件当中的sqlId一样。 |
| 参数类型要与映射文件当中的parameterType一样。 |
| 返回类型要与映射文件当中的resultType一样 |
| 获取实现 | dao = session.getMapper(xxxDAO.calss); MyBtis在底层会依据接口的要求生成符合要求的对象 |

# Spring与MyBatis整合(Mapper映射器)

|  |  |
| --- | --- |
| 导包 | spring-webmvc, mybatis, mybatis-spring, spring-jdbc, commons-dbcp, ojdbc14 |
| 添加Spring配置文件 | 不需要MyBatis配置文件，用一个bean代替(SqlSessionFactoryBean)  <bean id="ssfb" class="org.mybatis.spring.SqlSessionFactoryBean">  <property name="dataSource" ref="ds"/> // 指定连接资源  <property name="mapperLocations" value="classpath:\*.xml"/> // 指定映射文件 |
| 配置web.xml | <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">  <property name="basePackage" value="dao"/> // 指定要扫描的包  // sqlSessionFactory属性可以不用指定，会以Autowired方式自动注入  <property name="sqlSessionFactory" ref="sqlSessionFatory"/>  // 只扫描带有特定注解的接口  <property name="annotationClass" value="annotations.MyBatisRepository"/>  </bean> |
| 实体类 |
| 映射文件 |
| Mapper映射器 |
| 配置MapperScanner Configurer |
| 启动Spring容器  获取Mapper实现 | ApplicationContext ac = new ClassPathXmlApplicationContext(config);  dao = ac.getBean("daoId", xxxDAO.class); |

# MapperScannerConfigurer

|  |  |
| --- | --- |
| MapperFactoryBean会根据根据Mapper接口获取我们想要的Mapper对象，它封装了原有的session.getMapper()。一个Maper对应一个MapperFactoryBean。如果有大量Mapper，通过MapperScannerConfigurer可以自动扫描指定包下的各个Mapper接口，并注册对应的MapperFactoryBean对象。 | |
| name="annotationClass" | 用于指定一个注解，将只注册使用了指定注解的接口(Mapper映射器)   1. 自定义注解：public @interface MyBatisRepository {} 2. 在接口前写@MyBatisRepository |
| name="markInterface" | 用于置顶一个接口，将只注册继承了指定接口的接口 |

# Spring与MyBatis整合(不使用Mapper映射器)

|  |  |
| --- | --- |
| 之前步骤同上 | 其中映射文件namespace没有要求 |
| DAO接口 | 不再有Mapper映射器的要求 |
| DAO实现类 | 注入SqlSessionTemplate（封装了对SqlSesesion的操作，包括事务提交，关闭SqlSession） |
| 配置SqlSession Template | <bean id="sst" class="org.mybatis.spring.SqlSessionTemplate">  <constructor-arg index="0" ref="ssfb"/> |

# MyBatis动态SQL

|  |  |
| --- | --- |
| 在一些组合查询页面，需要根据用户输入的查询条件生成不同的查询SQL  使用动态SQL元素与JSTL相似，它允许我们在XML中构建不同的SQL语句 | |
| if | <if test="deptno!=null">  where DEPTNO=#{deptno} |
| choose | <choose>  <when test="sal>2000">  SAL>=#{sal}  </when>  <otherwise>  SAL>2000 |
| where | 在<where>元素所在位置输出一个where关键字，而且可以将后面条件多余的and或or关键字去除 |
| set | 在<set>元素所在位置输出一个set关键字，而且可以去除内容结尾中无关的逗号 |
| trim | 一个格式化的元素，可以完成set或者是where的功能 ??? |
| foreach | …… in  <foreach collection="集合" iteam="迭代变量" open="(" separator="," close=")">  #{迭代变量} |

# Spring对Ajax的支持

|  |  |
| --- | --- |
| @ResponseBody | 1. 导包jackson-annotations, Jackson-core, Jackson-databind 2. <mvc:annotation-driven/> 3. 在Controller处理方法前定义@ResponseBody |

# AOP

|  |
| --- |
| 一种编程思想，可以说是OOP的补充和完善。OOP允许开发者定义纵向的关系，但并不适合横向的关系，例如日志记录，安全控制，异常管理，事务处理。这些代码往往横向地散布在所有对象层次中，与核心业务毫无关系。  AOP技术将那些影响了多个类的公共行为封装到了一个可重用模块，并将其命名为Aspect，即切面。这样便于减少系统的重复代码，使得开发者能用更优雅的方式处理具有横切性质的服务。  AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。AOP的作用在于分离系统中各种关注点，将核心关注点和横切关注点分离开来。 |

# AOP基本概念

|  |  |  |
| --- | --- | --- |
| 横切 | cross-cutting | 与对象核心功能无关的公共行为 |
| 关注点 | concern | 一块我们感兴趣的区域 |
| 切面 | Aspect | 横切面对象，对横切关注点的抽象(可以借助@Aspect声明) |
| 连接点 | JoinPoint | 被拦截到的点，可以是方法，字段和构造器，Spring只支持拦截方法 |
| 切入点 | Pointcut | 对连接点进行拦截的定义 @Pointcut |
| 通知 | advice | 拦截到连接点之后要执行的代码 1）前置@Before 2）后置@After  3）异常@AfterThrowing 4）最终@AfterReturing 5）环绕通知@Around |
| 目标对象 | target | 代理的目标对象 |
| 织入 | weave | 将切面应用到目标对象并导致代理对象创建的过程 |
| 引入 | introduction | 为类动态地添加一些方法或字段 |

# AOP实现方式

|  |  |
| --- | --- |
| 静态代理 | 动态代理 |
| 写一个代理类：   1. 与被代理类实现相同的接口 2. 代理被代理对象实现功能的扩展 | 写一个实现代理类(ph)实现InvocationHandler：   1. 负责创建代理对象，创建的代理对象也实现相同的接口 2. 重写的invoke()，执行目标对象方法时默认调用 |
| ProjectService proxy =  new ProjectServiceStaticProxy(psi); | ProjectService proxy =  (ProjectService) ph.newProxy(psi); |

# Spring AOP

|  |  |  |  |
| --- | --- | --- | --- |
| 导包 | aspectjrt, aspectjweaver | | |
| 步骤 | spring配置文件 | <aop:aspectj-autoproxy/> // 使aspectj注解生效，自动为目标对象生成代理对象 | |
| 定义 bean业务组件 | @Service  public class XxxServiceImpl {} | |
| 定义切面 | @Order(1) @Aspect @Component  public class XxxAspect {} | |
| 定义切入点 | Bean名称 | @Before("bean(\*ServiceImpl)") |
| 方法 | @Pointcut("execution(\* cn.tedu.aop.\*ServiceImpl.\*Object(..))") |
| 类型 | @Before("within(cn.tedu.aop..\*) |
| 创建代理对象 | (XxxService) ac.getBean("@Service类的id"); | |
| 注意 | Spring AOP底层基于动态代理技术，可以很方便的实现业务功能扩展  Spring AOP底层代理对象创建过程分析（目标对象实现了接口则自动使用JDK动态代理，否则选择CGLIB包） | | |

# 基于注解的Spring声明式事务管理

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Srping声明式事务管理是通过Spring AOP实现的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事物，在执行目标方法之后根据执行情况提交或回滚事务。   1. 只有public方法起作用 2. 只有来自外部的方法调用才会被AOP代理捕获，类内部方法调用本类内部的其它方法不会引起事务行为 | | | | | | | | |
| 步骤 | spring 配置文件 | <!-- 配置PlatformTransactionManager事务管理组件  Hibernate技术的话，要选用HibernateTransactionManager -->  <bean id="txManager" class="…DataSourceTransactionManager">  <property name="dataSource" ref="dataSource"/>  </bean>  <tx:annotation-driven transaction-manager="txManager"/> // 开启事务注解扫描 | | | | | | |
| 在类或方法前添加@Transactional注解 | | | | | | | |
| 注解属性 | value | | 当配置文件中有多个TransactionManager，可以用该属性指定选择哪个事务管理器 | | | | | |
| propagation= Propagation. | | REQUIRED | 支持当前事务，如果当前没有事务，就新建一个事务，默认 | | | | |
| SUPPORTS | 支持当前事务，如果当前没有事务，就以非事务方式执行 | | | | |
| MANDATORY | 支持当前事务，如果当前没有事务，就抛出异常 | | | | |
| REQUIRES\_NEW | 新建事务，如果当前存在事务，把当前事务挂起 | | | | |
| NOT\_SUPPORTED | 以非事务方式执行，如果当前存在事务，就把当前事务挂起 | | | | |
| NEVER | 以非事务方式执行，如果当前存在事务，则抛出异常 | | | | |
| NESTED | 如果当前存在事务，则在嵌套事务内执行。如外层事务回滚，那么内层必须回滚；反之，内层事务不影响外层事务。 | | | | |
| isolation= Isolation. | |  | | 数据库默认 | 脏读 | 不可重复读 | 幻读 |
| READ\_UNCOMMITTED | |  | 可能 | 可能 | 可能 |
| READ\_COMMITTED | | Oracle, SQL Server | 不可能 | 可能 | 可能 |
| REPEATABLE\_READ | | MySQL | 不可能 | 不可能 | 可能 |
| SERIALIZABLE | |  | 不可能 | 不可能 | 不可能 |
| DEFAULT | | 默认的隔离级别，使用数据库默认的事务隔离级别 | | | |
| readOnly | | true/false | 设置当前事务是否为只读事务，设置为true表示只读，默认值为false | | | | |
| rollbackFor | | 设置需要进行事务回滚的异常类数组（默认RuntimeException将触发事务回滚） | | | | | |
| timeout | | 设置事务的超时秒数，默认值为-1，表示永不超时 | | | | | |

# resultType和resultMap

|  |  |
| --- | --- |
| resultType | resultMap |
| 直接表示返回类型（属性名要和表字段名一致） | 对外部ResultMap的引用 |
| 在MyBatis进行查询映射时，其实查询出来的每一个属性都是放在一个对应的Map里面的，其中键是属性名，值则是其对应的值。  当提供的返回类型属性是resultType时，MyBatis会将Map里面的键值对取出赋给resultType所指定的对象对应的属性。所以其实MyBatis的每一个查询映射的返回类型都是ResultMap，只是当提供的返回类型属性是resultType的时候，MyBatis对自动的给把应值赋resultType所指定对象的属性。 | |

# Mybaits中"#"与"$"的区别

#相当于对数据加上双引号，$相当于直接显示数据。

1. #将传入的参数值都当成一个字符串，会对自动传入的数据加一个双引号。如：order by #user\_id#，传入的值是27，那么解析成sql时的值为order by "27"， 如果传入的值是id，则解析成的sql为order by "id"，它写入的是你传进去的值。#可以防止语句注入
2. $将传入的属性直接显示生成在sql中。如：order by $user\_id$，如果传入的值是27，那么解析成sql时的值为order by user\_id，如果传入的值是id，则解析成的sql为order by id，它显示的是你传入的属性片段，而不是属性值。这是一个很危险的做法，如果在验证中加一个"or 1=1" 那就可能恒成立了。$不能防止语句注入，一般用于传入数据库对象或order by动态参数。

# 