Mybatis面面总结

- 基础
  - 半 ORM（对象关系映射）框架
  - 优点
    - 灵活性高
    - 良好的性能和扩展性
    - 可以与现有的 SQL 资源集成
  - 缺点
    - 开发效率相对较低
    - XML 配置复杂
    - 需要手动编写 SQL
  - Hibernate 和 Mybatis 的区别
    - 相同点
      - 都是对jdbc的封装，都是应用于持久层的框架
    - 不同点
      - 映射关系
        - Mybatis 是一个半自动映射的框架
        - Hibernate 是一个全表映射的框架
      - SQL的灵活解耦性
        - Hibernate
          - 代码开发量少，但SQL语句优化困难
          - 数据库无关性支持好
        - Mybatis
          - 需要手动编写 SQL开发工作量相对大些
          - 不支持数据库无关性，但SQL语句优化容易
      - 使用场景
        - Hibernate
          - 适合于需求相对稳定、中小型的软件项目
        - Mybatis
          - 适合于需求变化频繁、快速迭代的项目
  - Mybatis使用过程
    - 1. 创建SqlSessionFactory
    - 2. 通过SqlSessionFactory创建SqlSession
    - 3. 通过sqlsession执行数据库操作
      - 通过 SqlSession 直接调用预定义的映射的 SQL 语句
      - 获取Mapper接口，然后调用对应SQL语句
    - 4. 调用session.commit()提交事务
    - 5. 调用session.close()关闭会话
  - 生命周期
    - SqlSessionFactoryBuilder —— 创建阶段调用（只存在于方法的内部）
    - SqlSessionFactory —— 应用级别生命周期，线程共享
    - SqlSession —— 最短的生命周期是一次请求或一个方法
    - Mapper —— 在sqlsession事务方法之内，一般会随着方法结束
  - 传递多个参数
    - 推荐传参法（不建议）
    - 使用param注解传参法（推荐使用）
    - Map传参法
    - Java Bean 传参法（推荐使用）
  - #与$的区别
    - # $
      - 占位符，预编译处理
      - 传入参数值以字符串传入
      - 将SQL中的#替换为?号，调用PreparedStatement的set方法赋值
      - 可以有效防止SQL注入，提高系统安全性
      - 变量替换是在DBMS 中
    - $ $
      - 拼接符，字符串替换，没有预编译处理
      - 不能防止SQL 注入
      - 变量替换是在DBMS 外
  - 模糊查询like
    - "%"#{quanbion}"%"
      - 首尾%，1解析成sql语句时候，会在变量外侧自动加单引号‘ ’，然后这里 % 需要放在双引号“ ”
      - 不能使用单引号‘ ’，不符合语法不报错，但结果
    - concat("%",#{quanbion},"%") 推荐使用
  - 做联表查询
    - association
      - 一对一
      - 多对一
    - collection
      - 一对多
      - 多对多
  - 延迟加载
    - 实时对象
      - association关联对象
      - collection关联集合对象
    - 原理
      - 使用CGLIB动态创建目标对象的代理对象，当调用目标方法时，进入拦截器方法
      - 当要加载对象中的实际对象时，Mybatis 默认情况下不会立即加载关联对象的数据，而是通过一个代理对象，该代理对象在访问关联对象的属性或方法，但并未真正加载关联对象的数据
      - 当调用代理对象的实际属性或方法时，Mybatis 会拦截该调用并触发延迟加载机制。在延迟加载过程中，Mybatis 会执行数据库查询，获取关联对象的数据，并将结果注入到对象中
    - 主要步骤
      - 1.配置延迟加载
        - 在Mybatis配置文件中，配置合适的延迟加载 lazyLoadingEnabled=true
      - 2.创建代理对象
        - 当执行查询操作时，Mybatis 会创建一个代理对象，该代理对象将实际的目标对象替换为代理
      - 3.触发延迟加载
        - 当调用代理对象的关联属性或方法时，Mybatis 会拦截该调用并检测是否需要进行延迟加载
      - 4.执行延迟加载
        - 如果需要进行延迟加载，Mybatis 会执行数据库查询，获取关联对象的数据，并将结果注入到代理对象中
      - 5.返回关联对象数据
        - 延迟加载完成后，Mybatis 将返回已填充数据的关联对象
  - 获取生成的主键
    - 在insert标签中添加 useGeneratedKeys="true keyProperty="id "
  - 执行批量操作
    - <foreach>标签
      - 主要属性
        - item：表示集合中每一个元素进行迭代时的别名
        - index：指定一个名字，用于表示在迭代过程中，每次迭代到的位置
        - open：表示该语句以什么开始，常用 "("
        - separator：分隔符，常用 ","
        - close：表示以什么结束，常用 ")"
        - **collection：必须属性**
          - 若传入的是单个参数类型为List时，collection属性值为list
          - 若传入的是单个参数类型为Array时，collection属性值为array
          - 若传入的是多个参数需要封装为一个map，实际上在调用语句入参都封装时，在Mybatis底层会自动打包形成一个Map对象map参数类型，collection的属性值是传入的List或array对象在自己封装的map里面的key
    - ExecutorType.BATCH
      - 重复使用已经预处理的语句，并且批量执行所有更新语句
      - 在insert操作时，批量将这些语句提交之后，最后每次执行语句前都会flush
  - 缓存
    - 一级缓存（默认开启）
      - 基于 PerpetualCache 的 HashMap 本地缓存
      - 存储作用域为SqlSession
      - 各个SqlSession之间的缓存相互隔离
    - 二级缓存（默认不开启）
      - 默认实现采用 PerpetualCache的HashMap 存储
      - 存储作用域为 Mapper(Namespace)
      - 可以跨多个SqlSession之间共享，并且可自定义存储源，如 Ehcache
      - 开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口
- 工作原理
  - 生成会话工厂
    - 获取配置
      - 读取配置文件，包括基础配置文件和映射文件
      - 初始化基础配置，如Mybatis的别名、事务、数据源、ObjectFactory 等
      - 解析一个映射文件，并生成工厂模式需要事参数
      - 初始化一些环境变量，比如数据源
    - 构建SqlSessionFactory
      - SqlSessionFactory只是一个接口，构建出来的实际上是它的实现类的实例
      - 一般情况是它的实现类DefaultSqlSessionFactory
  - 会话运行
    - 四大组件
      - Executor（执行器）
        - 提供了相应的查询和更新方法，以及事务方法
      - StatementHandler（数据库会话器）
        - 处理数据库会话
      - ParameterHandler（参数处理器）
        - setParameters：设置预编译SQL语句的参数
        - typeHandler根据类型选择，对数据进行处理
      - ResultSetHander（结果处理器）
        - 结果集通过ResultSetHander来进行处理
        - handleResultSets方法：包装结果集的
        - 它使用typeHandler处理类型，然后将ObjectFactory提供的结构实例化对象，最后返回结果
    - 工作流程
      - 1. 读取 Mybatis 配置文件，最后生成一个配置对象
      - 2. 构造会话工厂 SqlSessionFactory
      - 3. 创建会话对象，由会话工厂创建 SqlSession 对象，包含执行SQL语句的所有方法
      - 4.Executor执行器，它调用底层 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护
      - 5.StatementHandler，数据库会话器，种类较多的操作的处理，和运行结果集的处理
      - 6. 参数处理：对输入参数的类型进行处理，并预编译
      - 7. 结果处理：对返回结果的类型进行处理，根据对象映射规则，返回响应的对象

作者：小白白白白菜，来自：知乎原创专栏

- 插件
  - Mybatis在进行运行需要ParameterHandler、ResultSetHandler、StatementHandler、Executor这四大对象的配合
  - 插件底层原理是动态代理加责任链模式，插入一些用户自己的代码
  - 编写插件
    - 实现Mybatis的Interceptor接口并重写Intercept()方法
    - 然后再给插件编写注解，确定要打算的对象，要打算的方法
    - 最后，将Mybatis配置文件里面配置插件
- 分页
  - 实现
    - 可以在sql语句里手写带有物理分页的参数来实现物理分页功能或者使用分页插件来实现物理分页。
    - Mybatis使用RowBounds对象进行分页，它面向对内存ResultSet结果集执行的内存分页，而非物理分页
  - 原理
    - 分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，打断Executor的Query方法
    - 在执行语句的时候，打断执行的Sql，然后重写sql，根据Mybatis方言，添加对应的物理分页语句和物理分页参数
    - 举例：select * from student，打断sql后重写为：select * from (select * from student) limit 0, 10
- 功能架构
  - API接口层
    - 提供给外部使用的接口API
    - 通过这些本地API来操纵数据库
    - 第二层一接收到调用请求就会调用数据处理层来完成具体的数据处理
  - 数据处理层
    - 参数解析
    - SQL解析
    - SQL执行
    - 结果处理映射等
    - 它主要的目的是根据调用的请求完成一次数据库操作
  - 基础支撑层
    - 连接管理
    - 配置加载
    - 事务管理
    - 缓存处理
    - 负责最基础的功能支撑
- Executor执行器
  - SimpleExecutor
    - 每执行一次update或select，就开启一个Statement对象，用完立即关闭Statement对象
  - ReuseExecutor
    - 配置项BatchExecutor的值
    - 执行update和select，以sql作为key查找Statement对象，存在就使用，不存在则创建
    - 执行update和select，以sql作为key查找Statement对象，存在就使用，不存在则创建
    - 在doSelect()、doUpdate()、doFlushStatements()之前所有的Statement对象，均加入到List<Statement>中，等待统一执行doFlushStatements()
    - 在SqlSession生命周期范围内
  - BatchExecutor
    - 执行update（没有select，JDBC批处理不支持select），将所有SQL都添加到批处理中（addBatch()），等待统一执行executeBatch()
  - 作用范围
    - 在SqlSession生命周期范围内
  - 指定执行器
    - 修改Mybatis配置文件中（settings）可以指定默认的executorType执行器类型
    - 手动调DefaultSqlSessionFactory的创建SqlSession的方法传递executorType类型参数
    - 配置默认的执行器
      - SIMPLE 普通语句的执行器
      - REUSE 执行器会重用预处理语句（prepared statements）
      - BATCH 执行器将重用语句并执行批量更新
- 获取Mapper的过程
  - 获取Mapper
  - MapperProxyFactory
    - 生成MapperProxy（Mapper代理对象）
  - MapperProxy
    - 在生成一个MapperMethod对象，通过cachedMapperMethod方法对其进行初始化的，然后执行execute方法
  - MapperMethod
    - execute()真正执行sql
    - 命令模式
      - 是一种行为设计模式，它将请求封装成一个对象，从而使得可以将不同的请求对象作为参数传递给其他，并且在稍后（Later）处理请求或者（Receiver）之间调用，使得发送者不需要知道具体的接收者
      - Command（命令）：定义了执行操作的接口，通常包含一个 execute() 方法
      - ConcreteCommand（具体命令）：实现了 Command 接口，具体定义了命令的执行方法，并绑定了接收者
      - Receiver（接收者）：负责具体执行命令的对象
      - Invoker（调用者）：负责调用命令对象执行请求
      - Client（客户端）：创建具体命令对象，并设置其接收者
      - 通过命令模式，请求发送者和接收者之间解耦，使得引入中间层"调度器"成为可能，它使得工作流更灵活、重用性强
      - 命令模式还可以用于实现日志记录、队列请求等应用场景