Mybatis

1. **对比传统的JDBC**
2. jdbc底层没有使用连接池，操作数据库需要频繁的创建连接，导致资源消耗增加；
3. 使用原生的jdbc代码，一旦出现更改sql时，java需要进行整体编译，不利于维护；
4. 使用PreparedStatement预编译，对变量设置123数字的序列号（eg：preparedStatement.setInt(1, id);），不利于维护；
5. 返回result结果也需要硬编码；
6. **基本概念**

Mybatis将SQL和其余的功能分离，为了解耦，把SQL编写的关键步骤留给开发者编写，其余的功能由Mybatis自动执行。（因此，mybatis时半自动框架）



1. **Mybatis全局配置属性表：**

使用properties文件配置属性



1. **Mybatis的基本使用步骤：**

**准备：**需要mybatis-3.5.4.jar、mysql-connector-java-8.0.18.jar

1. 编写**mybatis-config.xml全局配置文件**，创建SqlSessionFactory对象，包含：数据库信息、驱动、账号、密码、url等，以及**注册sql映射文件（eg：xxMapper.xml，非常重要，否则无法使用xxMapper.java接口中的方法进行CRUD操作）**；
2. 编写**sql映射文件（eg：xxMapper.xml）**，包含：namespace、方法绑定、编写需要执行sql语句；可用于绑定特定的接口、将sql语句和特定的接口方法绑定；

sql映射文件可以有多个，分别对应不同的表格，编写不同的sql语句，实现不同的功能；

1. **编写方法，调用需要的sql语句**：
2. 根据mybatis-config.xml全局配置文件创建SqlSessionFactory对象；
3. 使用SqlSessionFactory获取SqlSession对象，用该对象来执行sql语句（这是一次会话，用完就需要关闭）；
4. 使用SqlSession的selectOne()执行sql语句：需要使用sql映射文件（eg：xxMapper.xml）中的唯一标识指定执行的sql语句；

**示例代码：**

1. **mybatis-config.xml全局配置文件**

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <environments default="development">  
 <environment id="development">  
 <transactionManager type="JDBC" />  
 <dataSource type="POOLED">  
 <property name="driver" value="com.mysql.cj.jdbc.Driver" />  
 <property name="url" value="jdbc:mysql://localhost:3306/mybatis?serverTimezone=UTC" />  
 <property name="username" value="chris" />  
 <property name="password" value="1230re0321re" />  
 </dataSource>  
 </environment>  
 </environments>  
  **<!-- 将写好的sql映射文件（HelloWorldMapper.xml）注册到mybatis-config.xml中 -->** <**mappers**>  
 <mapper resource="**HelloWorldMapper.xml**" />  
 </**mappers**>  
</configuration>

1. **HelloWorldMapper.xml文件（sql映射文件）**

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  **//.dtd文件：提示文件**  
**<!--  
namespace:名称空间;指定为接口的全类名  
id：唯一标识  
resultType：返回值类型  
#{id}：从传递过来的参数中取出id值  
-->  
<!--接口的绑定-->**  
<mapper namespace="dao.HelloWorldMapper">  
  **<!--方法的绑定 ：HelloWorld类的getHelloWorldById方法 -->**  
 <select id="**getHelloWorldById**" resultType="bean.HelloWorld">  
 select id, last\_name lastName from helloworld where id = #{id}  
 </select>  
  **<!--使用id的方式：last\_name的别名 = lastname-->**  
 <select id="**select**" resultType="bean.HelloWorld">  
 select id, **last\_name lastName** from helloworld where id = #{id}  
 </select>  
</mapper>

1. **HelloWorld.java**

这是对应数据库表格的一个bean

1. **HelloWorldMapper.java**

是一个接口，负责定义方法，在HelloWorldMapper.xml中绑定接口 + 方法；

1. **测试功能的方法：**

public SqlSessionFactory **getSqlSessionFactory**() throws IOException {  
 String resource = "**mybatis-config.xml**";  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
 return new **SqlSessionFactoryBuilder**().build(inputStream);  
}

/\*\*\*\*\*\*\*\*\*\*\* 使用代理调用查找方法--获取结果 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
@Test  
public void test02() throws IOException {  
  **// 1、获取sqlSessionFactory对象**  
 SqlSessionFactory sqlSessionFactory = **getSqlSessionFactory**();  
  **// 2、获取sqlSession对象**  
 SqlSession openSession = sqlSessionFactory.**openSession**();  
 try{  
  **// 3、获取接口的实现类对象  
 //会为接口自动的创建一个代理对象，代理对象去执行xxMapper.xml中的SQL语句**  
 HelloWorldMapper mapper = openSession.getMapper(HelloWorldMapper.class);

**//调用接口的方法**  
 HelloWorld result = mapper.**getHelloWorldById**(2);  
  **//mapper是代理对象**  
 System.*out*.println(mapper.getClass());  
 System.*out*.println(result);  
 }  
 finally {  
 openSession.close();  
 }  
}

1. **mybatis-config.xml配置文件详解：（重要）**

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
<configuration>  
 <!--  
 **1、使用properties配置文件的内容；**  
 resource：引入类路径下的资源  
 url：引入网络路径或者磁盘路径下的资源  
 -->  
 <properties resource="dbconfig.properties"></properties>  
 <!--   
  **2、settings包含很多重要的设置项**  
 setting:用来设置每一个设置项  
 name：设置项名  
 value：设置项取值  
 -->  
 <settings>  
 <setting name="mapUnderscoreToCamelCase" value="true"/>

**<!—开启懒加载：调用什么方法，就查询什么数据，提升效率 -->**

<setting name="lazyLoadingEnabled" value="true"/>

<setting name="aggressiveLazyLoading" value="false"/>

</settings>  
 <!-- **3、typeAliases：别名处理器：可以为java类型起别名**，**别名不区分大小写**-->  
 <typeAliases>  
 <!—**3.1、typeAlias:为某个java类型起别名**  
 type:指定要起别名的类型全类名;默认别名就是类名小写；employee  
 alias:指定新的别名  
 -->  
 <typeAlias type="com.atguigu.mybatis.bean.Employee" alias="emp"/>  
 <!—**3.2、package:为某个包下的所有类批量起别名**   
 name：指定包名（为当前包内每一个类都起一个“默认别名”（类名小写））  
 -->  
 <package name="com.atguigu.mybatis.bean"/>  
 <!—**3.3、批量起别名的情况下，使用@Alias注解为某个类型指定新的别名** -->  
 </typeAliases>

<!--   
 **4、environments：环境配置**，**default指定使用某种环境**。可以达到快速切换环境。

必须有两个标签；

（1）**id**：**代表当前环境的唯一标识**；  
 （2）**transactionManager**：**事务管理器**；  
 type：事务管理器的类型;

**JDBC**(JdbcTransactionFactory)|**MANAGED**(ManagedTransactionFactory)  
 自定义事务管理器：实现TransactionFactory接口.type指定为全类名  
 （4）**dataSource**：**数据源**;  
 type:数据源类型;**UNPOOLED**(UnpooledDataSourceFactory) ：不使用数据库连接池  
 |**POOLED**(PooledDataSourceFactory)：使用数据库连接池  
 |**JNDI**(JndiDataSourceFactory)  
 自定义数据源：实现DataSourceFactory接口，type是全类名  
 -->  
 <environments default="dev\_mysql">  
 <environment id="dev\_mysql">  
 <transactionManager type="JDBC"></transactionManager>  
 <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>  
 <!-- **5、databaseIdProvider：支持多数据库厂商；**  
 type="DB\_VENDOR"：VendorDatabaseIdProvider  
 得到数据库厂商的标识(驱动getDatabaseProductName())，mybatis就能根据数据库厂商标识来执行不同的sql；例如：MySQL，Oracle，SQL Server,xxxx  
 -->  
 <databaseIdProvider type="DB\_VENDOR">  
 **<!--为不同数据库起别名 ：在sql映射文件中使用databaseId = xx，使用value=xx的数据库-->**  
 <property name="MySQL" value="mysql"/>  
 <property name="Oracle" value="oracle"/>  
 <property name="SQL Server" value="sqlserver"/>  
 </databaseIdProvider>

<!-- **6、mappers：将sql映射注册到全局配置中** -->  
 <mappers>  
 <!--   
 mapper:注册一个sql映射   
 **6.1、注册sql映射文件（对应的接口使用xxMapper.xml（注册配置文件）编写SQL语句）**  
 **resource**：引用类路径下的sql映射文件  
 mybatis/mapper/EmployeeMapper.xml  
 **url**：引用网路路径或者磁盘路径下的sql映射文件  
 file:///var/mappers/AuthorMapper.xml  
  **6.2、注册接口（该接口使用的是注解方式编写SQL语句）**  
 **class**：引用（注册）接口，  
 1、有sql映射文件，映射文件名必须和接口同名，并且放在与接口同一目录下；  
 2、没有sql映射文件，所有的sql都是利用注解写在接口上;  
 -->  
 <!-- **注册sql映射文件：sql语句编写在xml文件中** -->

<mapper resource="mybatis/mapper/EmployeeMapper.xml"/>

<!-- **批量注册：批量注册某个包中的xml文件，sql语句编写在xml文件中**  -->  
 <package name="com.atguigu.mybatis.dao"/>

<!-- **注册接口：sql语句编写在.java文件的方法注解中** -->  
 <mapper class="dao.EmployeeMapperAnnotation"/>  
 </mappers>

1. **sql映射文件详解：（重要）**

一般来说，一个xxMapper.java接口文件对应一个xxMapper.xml映射文件；

**作用：**将sql语句抽取出来

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper **namespace**="com.atguigu.mybatis.bean.**Employee** ">  
**<!--   
namespace:**名称空间;指定为接口的全类名 **id：**唯一标识，可绑定namespace指定接口中的方法，或者使用自定义的字符串 **resultType：**返回值类型

**databaseId：**指定使用的数据库别名 **#{id}：**从传递过来的参数中取出id值 **-->**

**<!—传入单个参数时：调用**public Employee **getEmpById**(Integer id);**-->**   
 <select **id**="**getEmpById**" **resultType**="com.atguigu.mybatis.bean.Employee"  
 databaseId="mysql">  
 select \* from tbl\_employee where id = #{id}  
 </select>

**<!—传入多个参数时，需要封装成map格式（key-value）：调用getEmpByMap()方法 -->** <select id="getEmpByMap" resultType=" com.atguigu.mybatis.bean.Employee ">

**<!— id：数据库中表的列名； #{id}：获取的是传入参数‘id’的值 -->**  
 select \* from ${tableName} where **id=#{id}** and last\_name=#{lastName}  
 </select>  
   
 <!-- parameterType：参数类型，可以省略，   
 **获取自增主键的值：**  
 mysql支持自增主键，自增主键值的获取，mybatis也是利用statement.getGenreatedKeys()；  
 **useGeneratedKeys**="true"；使用自增主键获取主键值策略  
 **keyProperty**；指定对应的主键属性，也就是mybatis获取到主键值以后，将这个值封装给javaBean的哪个属性  
 -->  
 <insert id="addEmp" parameterType="com.atguigu.mybatis.bean.Employee"  
 **useGeneratedKeys**="true" **keyProperty**="id" **databaseId**="mysql">  
 insert into tbl\_employee(last\_name,email,gender)   
 values(#{lastName},#{email},#{gender})  
 </insert>  
</mapper>

1. **mybatis参数处理：**
2. **单个参数：**

不做特殊处理，直接使用#{}、${}获取参数值；

1. **多个参数：**

**将多个参数封装成map**（eg：Map<String, Object>形式的map参数）；

map的key：参数名；

map的value：参数值；

在sql映射文件中，填写sql语句时，直接使用#{key}、${key}获取map中key对应的value；

1. **使用@Param注解**
2. 指定接口方法中形参的别名，在sql映射文件中填写sql语句时，直接使用@Param指定的别名即可获取对应的参数；
3. 当**使用自定义的封装参数**时（eg：ArrayList），需要使用@Param注解指定别名，否则无法在xxMapper.xml中使用传入的参数。

**eg：**

HelloWorldMapper.java中

public HelloWorld **getHelloWorldByIdAndLastname**(**@Param**("**id**")Integer Id,

**@Param**("**lastname**")String last\_name);

HelloWorldMapper.xml中

<select id="**getHelloWorldByIdAndLastname**" resultType="bean.HelloWorld">  
 select \* from helloworld where id=**#{id}** and last\_name=**#{lastname}**  
</select>

1. **可以传入POJO（简单Java对象）作为参数**

使用#{属性名}，取出传入的POJO属性值；POJO对应的是数据模型；

1. **拓展知识：**

**Q：#{}和${}的区别：**

**A：**#{}、${}两者都可以获取map、POJO的数值

**1）#{}：**

是以预编译的形式，将参数值设置到sql语句中，防止sql注入；类似于PreparesStatement；

**2）${}：**

取出的参数值，直接拼装到sql语句中，存在安全问题；

**eg：**在同一条sql语句中使用#{}、${}

<select id="getHelloWorldByMap" resultType="bean.HelloWorld">  
 select \* from ${tablename} where id=#{id} and last\_name=#{lastname}  
</select>

**结果：**执行的sql语句，#{}使用？占位符，${}直接显示参数值



1. **自定义resultType返回值规则：**

需要自己编写一个xxMapper.xml文件，在这里面定义即可。

**<!--自定义某个javaBean的封装规则**  
**type：**自定义规则的Java类型  
**id:** 唯一id方便引用  
**-->**  
<**resultMap** type="com.atguigu.mybatis.bean.Employee" **id="MySimpleEmp"**>  
  **<!--指定主键列的封装规则**  
 **id：**定义主键，底层有优化；  
 **column：**指定哪一列  
 **property：**指定对应的javaBean属性  
  **-->**

**<!—数据库表格中的列名column = id，对应javaBean中的property = id属性 -->** <id column="id" property="id"/>  
  **<!-- 定义普通列封装规则 -->**  
 <result column="last\_name" property="lastName"/>  
 **<!-- 其他不指定的列会自动封装：只要写resultMap就把全部的映射规则都写上 -->**  
 <result column="email" property="email"/>  
 <result column="gender" property="gender"/>

**<!— 级联封装：把数据库中did、dept\_name列的数据，放到javaBean的dept对象的属性中 -->**

**<!— 方法一： -->**

<result column="did" property="dept.id"/>

<result column="dept\_name" property="dept.departmentName"/>

**<!— 方法二：使用association -->**

<association property="dept" javaType="com.atguigu.mybatis.bean.Department">  
 <id column="did" property="id"/>  
 <result column="dept\_name" property="departmentName"/>  
 </association>

**<!— 分段查找：使用association实现**  
 **select:** 当前属性是调用select指定的方法查出的结果  
  **column:** 指定将哪一列的值传给这个方法  
 **步骤：** 使用select指定的方法（传入column指定的这列参数的值）查出对象，并封装给property指定的属性  
  **-->**  
 <association property="dept"   
 select="com.atguigu.mybatis.dao.DepartmentMapper.**getDeptById**"  
 column="d\_id">  
 </association>  
**</resultMap>**  
  
**<!—使用自定义结果集映射规则；resultMap:** **-->**  
<select id="getEmpById" resultMap="**MySimpleEmp**">  
 select \* from tbl\_employee where id=#{id}  
</select>

1. **Mybatis动态sql（重要）**

主要是使用if、choose、trim、foreach等标签进行sql语句的筛选、拼接；

1. **if**：判断
2. **choose**：和c中的switch一样，其格式为：choose、when、otherwise
3. **trim**：字符串截取
4. **foreach**：遍历集合
5. **两个内置参数**：

1）**\_parameter**：代表整个传入的参数，可以表示单个、多个参数；

2)**\_databaseId**：若mybatis-config.xml中配置databaseIdProvider标签，则\_databaseId表示当前数据库的别名

**<!-- 使用if、where标签进行筛选 -->**  
<select id="getEmpsByConditionIf" resultType="com.demo.bean.Employee">  
 select \* from tbl\_employee  
 <where>  
  **<!-- test表达式为true时，进入if内 -->**  
 <if test="id!=null">  
 id=#{id}  
 </if>  
  **<!--&amp; 表示& &quot; 表示“” -->**  
 <if test="lastName!=null &amp;&amp; lastName!=&quot;&quot;">  
 and last\_name like #{lastName}  
 </if>  
 <if test="gender==0 or gender==1">  
 and gender=#{gender}  
 </if>  
 </where>  
</select>

**<!--使用foreach获取查找的id序列-->**  
<select id="getEmpsByConditionForeach" resultType="com.demo.bean.Employee">  
 select \* from tbl\_employee  
 <foreach collection="listId" item="item\_id" separator=","  
 open="where id in(" close=")">  
 id=#{item\_id}  
 </foreach>  
</select>

**<!-- 使用trim -->**  
 <select id="getEmpsByConditionTrim" resultType="com.demo.bean.Employee">  
 select \* from tbl\_employee  
 <!-- 后面多出的and或者or where标签不能解决  
 prefix="":前缀：trim标签体中是整个字符串拼串 后的结果。

prefix给拼串后的整个字符串加一个前缀

prefixOverrides="":前缀覆盖： 去掉整个字符串前面多余的字符

suffix="":后缀，suffix给拼串后的整个字符串加一个后缀

suffixOverrides=""：后缀覆盖：去掉整个字符串后面多余的字符  
-->  
 **<!-- 自定义字符串的截取规则：前缀是where，去除字符串前面多余的and -->** <trim prefix="where" suffixOverrides="and">  
 <if test="id!=null">  
 id=#{id} and  
 </if>  
 <!--&amp; 是& &quot; 是“” -->  
 <if test="lastName!=null &amp;&amp; lastName!=&quot;&quot;">  
 last\_name like #{lastName} and  
 </if>  
 <if test="gender==0 or gender==1">  
 gender=#{gender}  
 </if>  
 </trim>  
 </select>

动态SQL拼接

**数坤实习时，自己写的order-service订单查询：**

**<!—1、先用原生的sql查询、拼接自己想要的数据（可以多表拼接） -->**

<sql id="selectTableSql">  
 select **o.\*** from t\_order o **left join** t\_invoice i **on** o.invoice\_id = i.id  
</sql>  
  
<sql id="orderFilterSql">  
 <include refid="selectTableSql"></include>  
 <where>  
 <!-- 按照订单Id过滤：**调用传入的参数值，判断函数传入的参数params是否合法** -->  
 <if **test**="params.orderNo != null and params.orderNo.length() > 0">  
 **and** o.order\_no = **#{params.orderNo}**  
 </if>  
 <!-- 按照订单创建时间段过滤 -->  
 <if test="params.createStartTime != null">

**<!--调用函数传入的参数：params，将long型的时间戳转为SQL中的时间值； 大于等于&gt;= -->**  
 **and** o.create\_date **&gt;=** **FROM\_UNIXTIME**(**#{**params.createStartTime**}**)  
 </if>  
 <!-- 按照产品类型过滤 -->  
 <if test="params.productType != null">  
 **and** o.product\_type **in**  
 <foreach collection="params.product" item="item" open="(" separator="," close=")">  
  **#{item}**  
 </foreach>  
 </if>  
 <!-- 按照发票状态过滤 -->  
 <if test="params.invoiceStatus != null and params.invoiceStatus.size() > 0">  
 and

**<!—使用trim，自动去除多余的or -->**  
  **<trim** prefix="(" suffix=")" suffixOverrides="or">  
 <!-- 已申请：**动态SQL中调用工程中的枚举类** -->  
 <if test="params.invoiceStatus.contains(**@com.shukun.enumeration.InvoiceFilterDefine@APPLY\_FOR\_INVOICE.getStatus()**)">  
 i.status = **${@com.shukun.enumeration.InvoiceStatusDefine@WAIT\_DEAL.getStatus()}** or  
 </if>  
 </trim>  
 </if>  
 </where>  
  **<!-- 按照订单创建时间降序排序 -->**  
  **order by** o.create\_date **desc**  
</sql>

**<!—2、调用动态SQL语句，查询数据：工程中真正调用的方法为：findOrderByFilterAndPage -->**  
<select id="**findOrderByFilterAndPage**" resultType="com.shukun.toc.orderservice.entity.Order" parameterType="com.model.dto.OrderPageQuery">

**<!—调用前面写的动态SQL语句 -->**  
 select \* from  
 (<include refid="**orderFilterSql**"></include>) temp **limit** #{params.page}, #{params.pageSize}  
</select>

**工程中的mapper.java文件**

public interface OrderMapper extends BaseMapper<Order> {  
 /\*\*  
 \* **使用映射文件.xml，拼接动态sql（该方法可以结合原生的sql语句实现多表查询功能）**

\* **@Parma(“params”)：表示给对应sql映射文件中的bean传入名为params的参数**。  
 \*/  
 List<Order> **findOrderByFilterAndPage**(**@Param("params")**OrderPageQuery params);

/\*\*

\* **使用注解@Select拼接动态sql**

\*/

**@Select**("select \* from user\_info where id in (${ids})")

List<UserInfo> getUserbyIds(@Param("ids")String ids);

}

1. **Mybatis缓存机制**
2. Mybatis一共提供两种缓存：一级缓存、二级缓存（虽然这些缓存基本没什么用处一般大多使用Redis等第三方缓存产品）； **Mybatis的缓存底层源码使用map进行数据存储。**
3. **对一级、二级缓存进行详解：**
4. **一级缓存（本地缓存）**

sqlSession级别的缓存，**一个sqlSession对应一个一级缓存**。**一级缓存是一直开启的（不需要使用者手动开启）**，在一个sqlSeesion中执行**增删改查操作（CRUD）**后，会把相应的数据保存到这个sqlSession的map中，之后同一个sqlSession要获取相应数据，直接从一级缓存中获取（前提：这个sqlSession没有关闭）；

**一级缓存失效的情况：**

1. 执行操作的sqlSession不同
2. sqlSession相同，但执行的操作不同（即：一级缓存中没有这个数据）
3. sqlSession相同，但中间执行了CRUD操作
4. sqlSession相同，但中间手动清除了一级缓存

1. **二级缓存（全局缓存）**

需要手动开启。二级缓存是对应到xxMapper.xml中namespace的指定xxMapper.java（**每个xxMapper.xml有唯一的一个二级缓存空间**）。也就是说，只要是同一个namespace中创建的sqlSession，在sqlSession关闭之后，就会将其一级缓存中的内容保存到二级缓存中。

**使用的步骤：**

1. mybati-config.xml中开启二级缓存：

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

1. 在对应的xxMapper.xml中配置二级缓存：

<cache eviction="FIFO" flushInterval="600000" readOnly="false" size="1024"></cache>

1. 对涉及到的javaBean类（POJO）实现序列化接口：

public class Employee implements **Serializable**{}

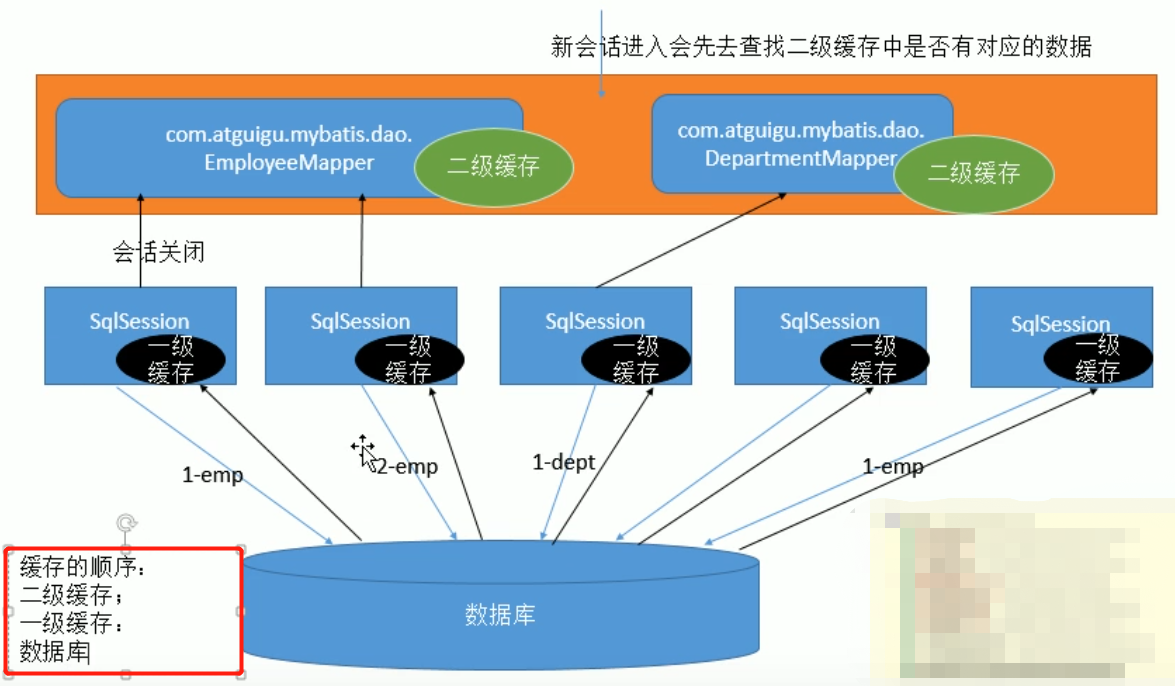
**工作机制：**

1. 一个sqlsession执行sql后，该数据会保存到一级缓存中；
2. 当该sqlSession关闭后，一级缓存的数据会被保存到二级缓存中；新的sqlSession执行CRUD操作时，会先从二级缓存中查找，没有的话再到数据库中查找；

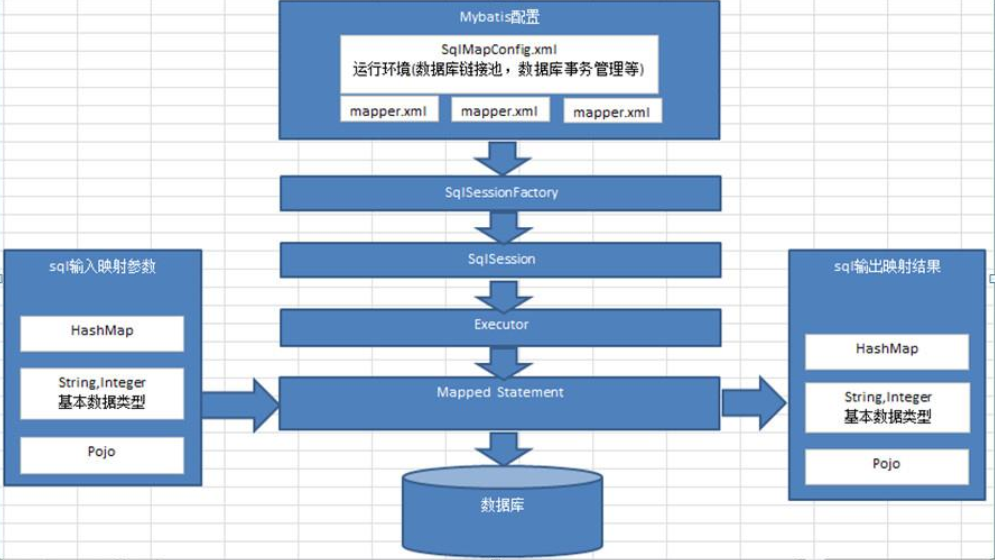
1. **mybatis的缓存原理**

mybatis缓存的底层源码，使用的是map存储。

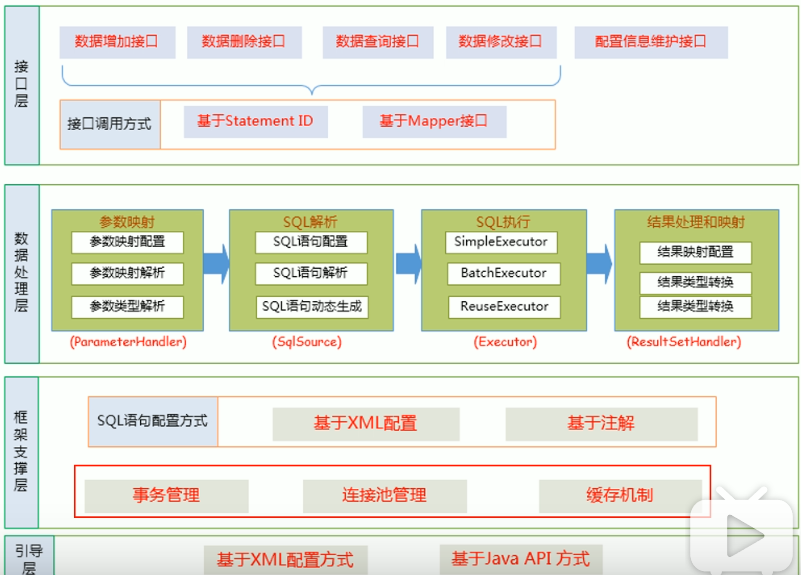
* 1. 创建sqlSession，进行CRUD操作后，数据保存到一级缓存；
  2. 当sqlSession关闭后，其对应饿一级缓存会保存到创建这个sqlSession对应的sqlFactory的二级缓存中（依据sql映射文件xxMapper.xml区分不同的二级缓存）；
  3. 再次创建新的sqlSession，进行CRUD操作时，先查找对应的二级缓存，若找不到，则再到数据库中查找。



1. **Mybatis的运行原理**



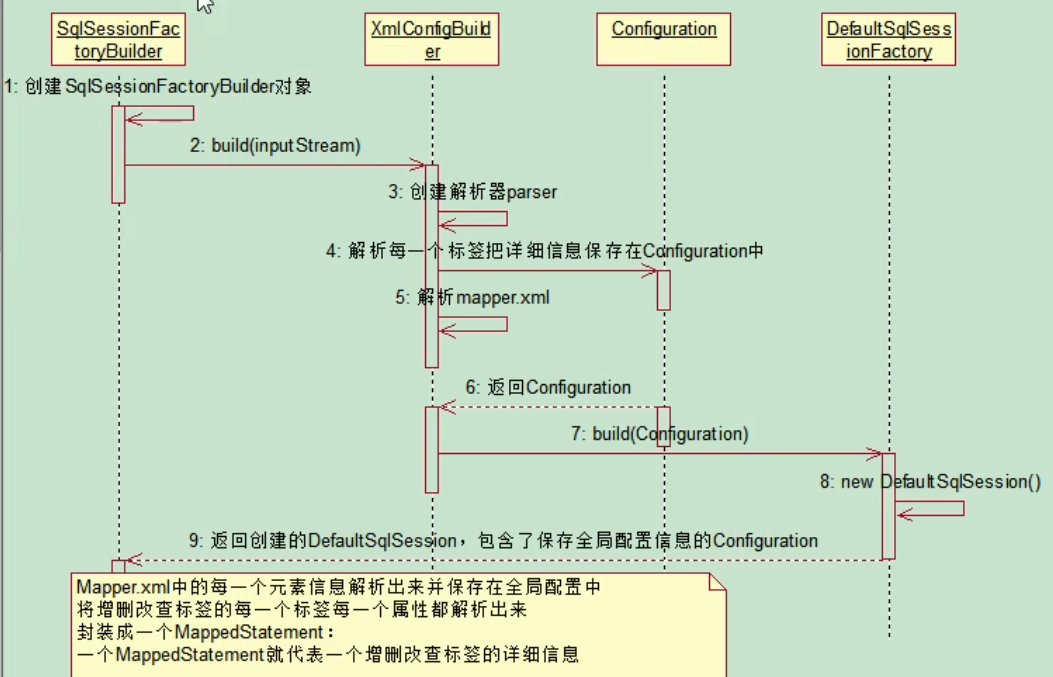
1. **Mybatis的框架：**



1. **Mybatis工作流程图**
2. **创建SqlSessionFactory的步骤：**

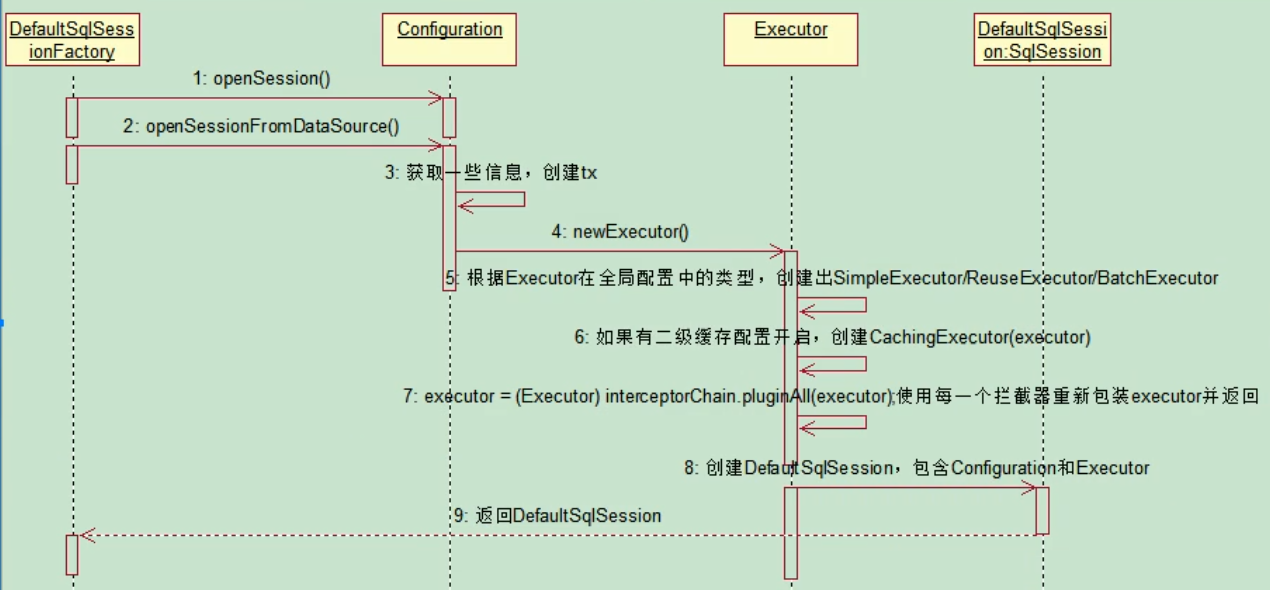
解析sql映射文件（xxMapper.xml），该文件的每个配置信息保存在Configuration中，返回包含Configuration的DefaultSqlSessionFactory对象；

**注意：**MappedStatement，表示一个增删改查的详细信息



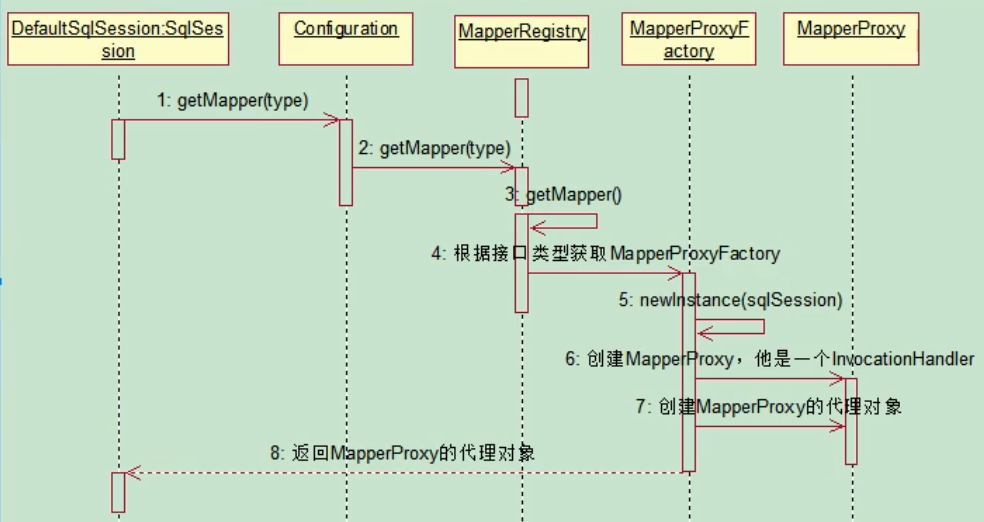
1. **使用sqlSessionFactory.openSession()获取openSession对象：**

返回的是DefaultSqlSession对象，包含：**Excutor**、Configuration



1. **使用openSession.getMapper()，获取mapper对象**

返回的**mapper**对象**包含：sqlSession**对象（**也就意味着mapper包含Excutor对象，可以用于执行CRUD操作**）



1. **Mybatis实现查询的原理**

缓存中保存：sql、方法、……，最终都是使用原生JDBC操作数据库

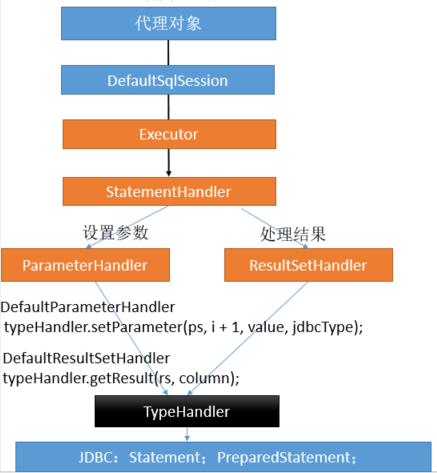
**1）查询的步骤：（使用4大对象进行操作）**

StatementHandler：处理sql语句预编译，设置参数等相关工作；

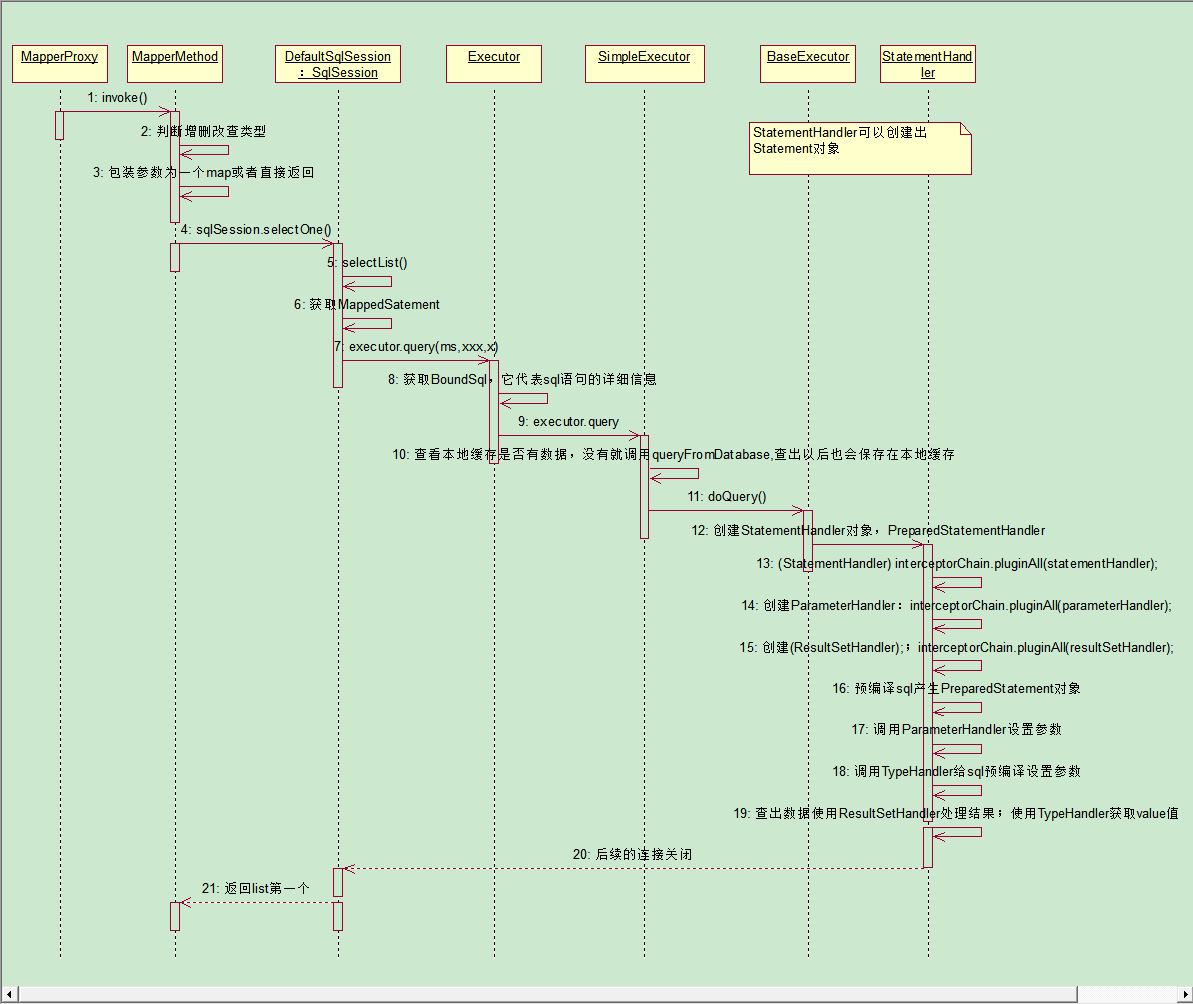
ParameterHandler：设置预编译参数用的；

ResultHandler：处理结果；

TypeHandler：整个过程，进行数据库类型和javaBean类型的映射；



**2）详细代码执行步骤**：



1. **逆向工程**

**（Myabtis官方提供的代码：**<https://github.com/mybatis/generator>**）**

1. Mybatis Generator（MBG），为Mybatis框架使用者定制的代码生成器，根据表格快速生成对应的sql映射文件、接口、javaBean类（支持CRUD操作）；

补充知识

1. Mybatis的 < 、<= 、> 、>= 、!=、&、' 、" 表示方式

|  |  |
| --- | --- |
| 数学符号 | Mybatis表示方法 |
| < | &lt; |
| <= | &lt;= |
| > | &gt; |
| >= | &gt;= |
| != | !<[CDATA[<>]]> |
| & | &amp; |
| ' | &apos; |
| " | &quot; |