> 图文详解 20 道MyBatis 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：楼仔，作者：三分恶，戳原文链接。

大家好，我是楼仔呀，面渣逆袭系列继续，这节我们的主角是MyBatis，作为当前国内最流行的ORM框架，是我们这些crud选手最趁手的工具，赶紧来看看面试都会问哪些问题吧。

基础

1. 说说什么是MyBatis?

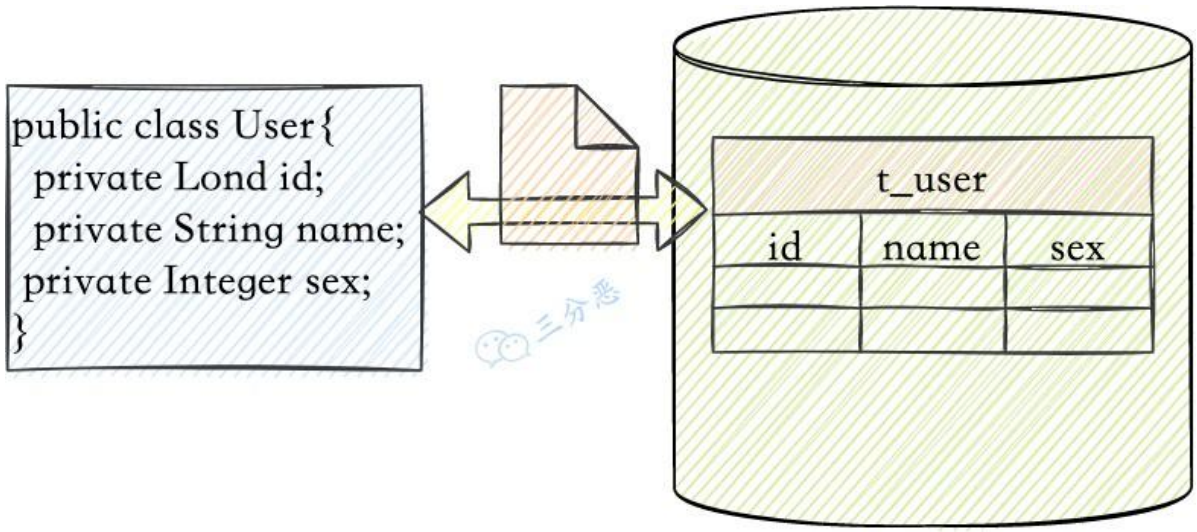
先吹一下：

- Mybatis 是一个半 ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注 SQL 语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement 等繁杂的过程。程序员直接编写原生态 sql，可以严格控制 sql 执行性能，灵活度高。
- MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO 映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

再说一下缺点

- SQL语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL语句的功底有一定要求
- SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库

> ORM是什么?



- ORM（Object Relational Mapping），对象关系映射，是一种为了解决关系型数据库数据与简单Java对象（POJO）的映射关系的技术。简单来说，ORM是通过使用描述对象和数据库之间映射的元数据，将程序中的对象自动持久化到关系型数据库中。

> 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

- Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。
- 而Mybatis在查询关联对象或关联集合对象时，需要手动编写SQL来完成，所以，被称之为半自动ORM映射工具。

> JDBC编程有哪些不足之处，MyBatis是如何解决的？

- 1、数据连接创建、释放频繁造成系统资源浪费从而影响系统性能，在mybatis-config.xml中配置数据链接池，使用连接池统一管理数据库连接。
- 2、sql语句写在代码中造成代码不易维护，将sql语句配置在XXXXmapper.xml文件中与java代码分离。 3、向
- sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。Mybatis自动将java对象映射至sql语句。
- 4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。Mybatis自动将sql执行结果映射至java对象。

2. Hibernate 和 MyBatis 有什么区别？

相同点

- 都是对jdbc的封装，都是应用于持久层的框架。

不同点

1）映射关系

- MyBatis　是一个半自动映射的框架，配置Java对象与sql语句执行结果的对应关系，多表关联关系配置简单
- Hibernate　是一个全表映射的框架，配置Java对象与数据库表的对应关系，多表关联关系配置复杂

2）SQL优化和移植性

- Hibernate 对SQL语句封装，提供了日志、缓存、级联（级联比 MyBatis 强大）等特性，此外还提供 HQL（Hibernate Query Language）操作数据库，数据库无关性支持好，但会多消耗性能。如果项目需要支持多种数据库，代码开发量少，但SQL语句优化困难。
- MyBatis 需要手动编写 SQL，支持动态 SQL、处理列表、动态生成表名、支持存储过程。开发工作量相对大些。直接使用SQL语句操作数据库，不支持数据库无关性，但sql语句优化容易。

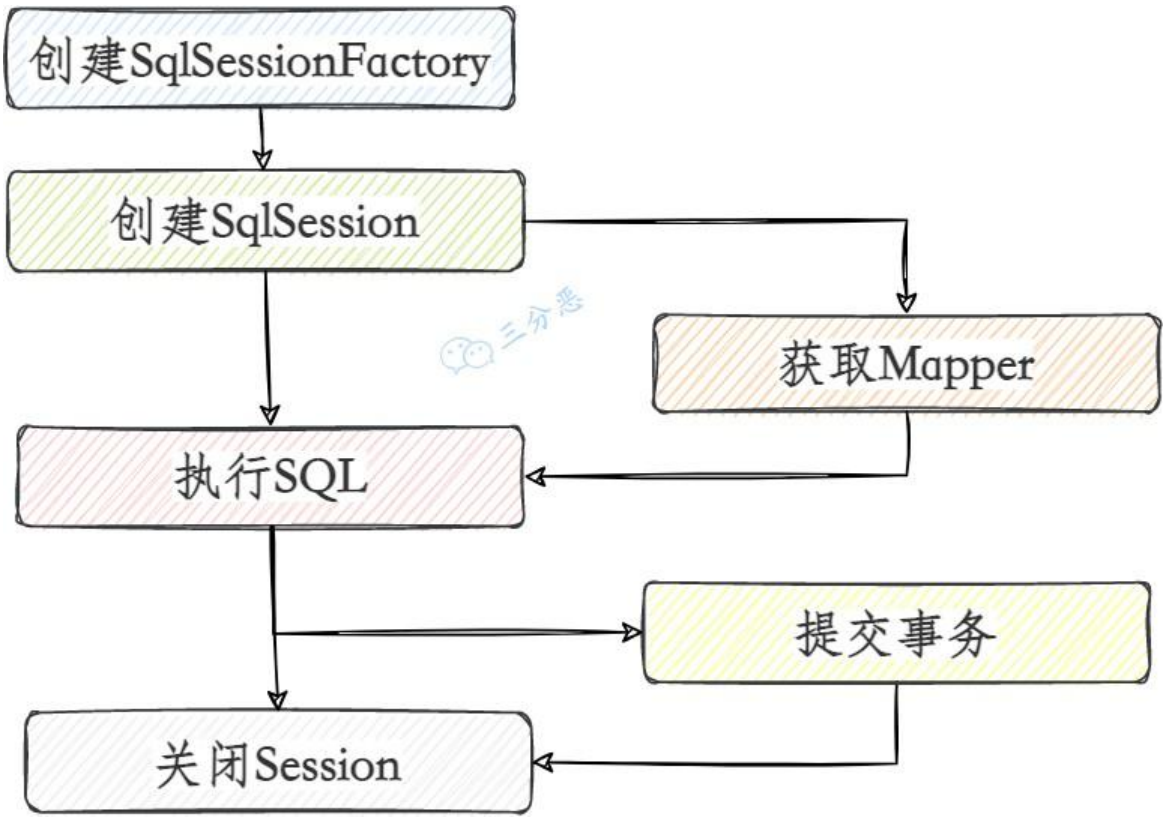
3）MyBatis和Hibernate的适用场景不同

VS

- Hibernate 是标准的ORM框架，SQL编写量较少，但不够灵活，适合于需求相对稳定，中小型的软件项目，比如：办公自动化系统
- MyBatis 是半ORM框架，需要编写较多SQL，但是比较灵活，适合于需求变化频繁，快速迭代的项目，比如：电商网站

3. MyBatis使用过程？生命周期？

MyBatis基本使用的过程大概可以分为这么几步：



- 1）创建SqlSessionFactory

可以从配置或者直接编码来创建SqlSessionFactory

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
```

- 2）通过SqlSessionFactory创建SqlSession SqlSession（会

话）可以理解为程序和数据库之间的桥梁

```
SqlSession session = sqlSessionFactory.openSession();
```

- 3）通过sqlsession执行数据库操作

可以通过 SqlSession 实例来直接执行已映射的 SQL 语句：

```
Blog blog = (Blog)session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);
```

更常用的方式是先获取Mapper(映射)，然后再执行SQL语句：

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

- 4） 调用session.commit()提交事务

如果是更新、删除语句，我们还需要提交一下事务。

- 5） 调用session.close()关闭会话最

后一定要记得关闭会话。

> MyBatis生命周期？

上面提到了几个MyBatis的组件，一般说的MyBatis生命周期就是这些组件的生命周期。

- SqlSessionFactoryBuilder

一旦创建了SqlSessionFactory，就不再需要它了。 因此 SqlSessionFactoryBuilder 实例的生命周期只存在于方法的内部。

- SqlSessionFactory

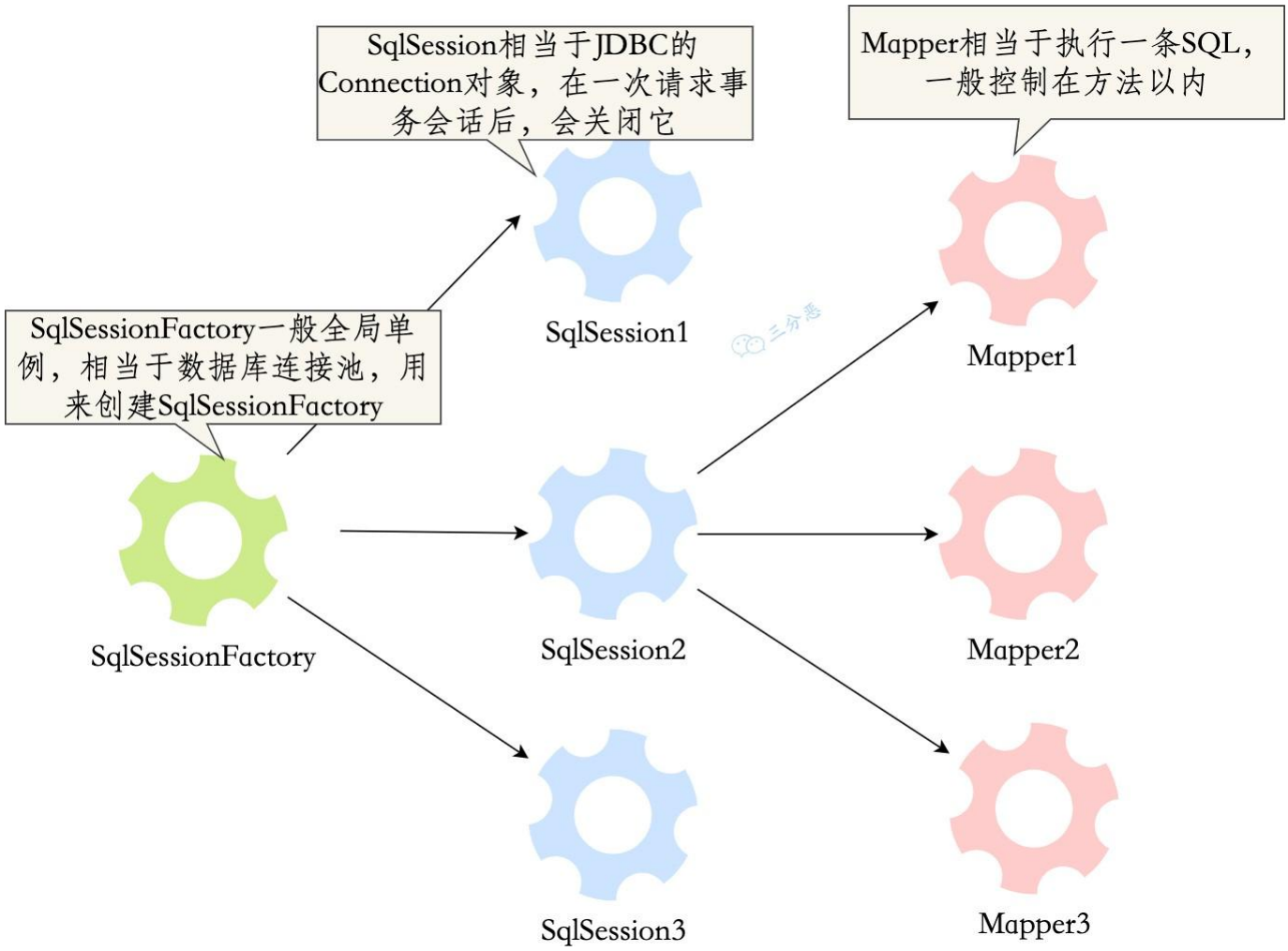
SqlSessionFactory 是用来创建SqlSession的，相当于一个数据库连接池，每次创建SqlSessionFactory都会使用数据库资源，多次创建和销毁是对资源的浪费。所以SqlSessionFactory是应用级的生命周期，而且应该是单例的。

- SqlSession

SqlSession相当于JDBC中的Connection，SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的生命周期是一次请求或一个方法。

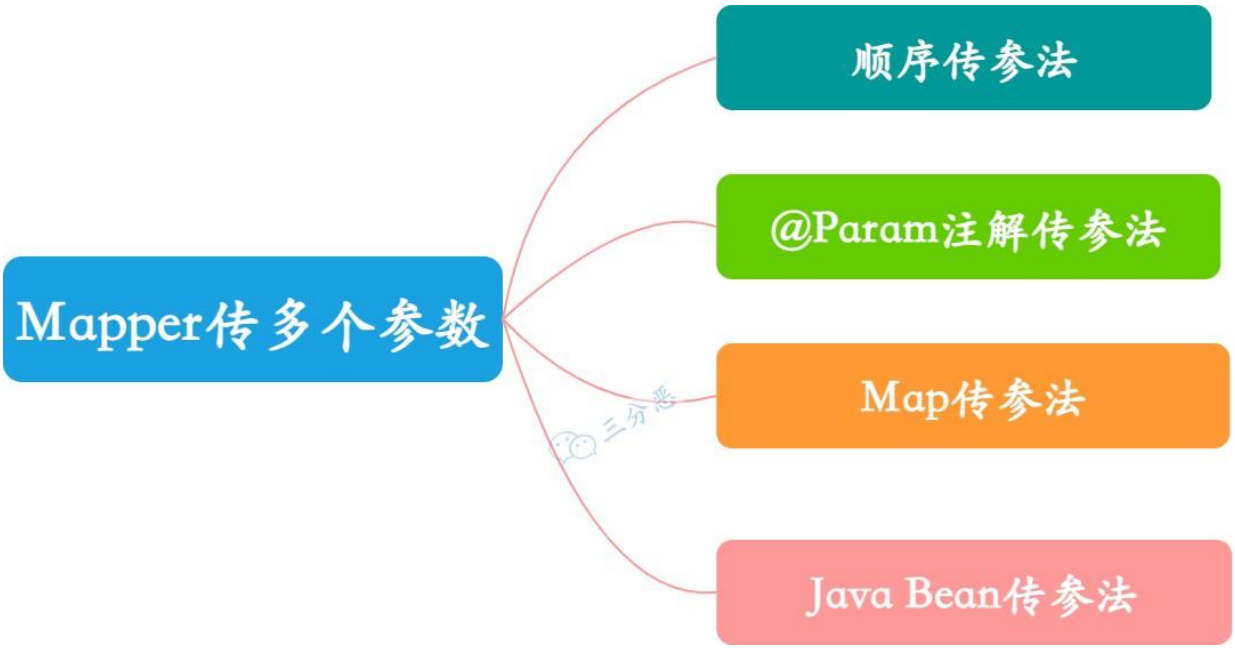
- Mapper

映射器是一些绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的，它的生命周期在sqlsession事务方法之内，一般会控制在方法级。



当然，万物皆可集成Spring，MyBatis通常也是和Spring集成使用，Spring可以帮助我们创建线程安全的、基于事务的 SqlSession 和映射器，并将它们直接注入到我们的 bean 中，我们不需要关心它们的创建过程和生命周期，那就是另外的故事了。

4. 在mapper中如何传递多个参数？



方法1：顺序传参法

```
public User selectUser(String name, int deptId);

<select id="selectUser" resultMap="UserResultMap">
    select * from user
    where user_name = #{0} and dept_id = #{1}
</select>
```

- `#{}`里面的数字代表传入参数的顺序。
- 这种方法不建议使用，sql层表达不直观，且一旦顺序调整容易出错。

方法2：@Param注解传参法

```
public User selectUser(@Param("userName") String name, int @Param("deptId") deptId);

<select id="selectUser" resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- `#{}`里面的名称对应的是注解@Param括号里面修饰的名称。这种
- 方法在参数不多的情况还是比较直观的，（推荐使用）。

方法3：Map传参法

```
public User selectUser(Map<String, Object> params);

<select id="selectUser" parameterType="java.util.Map" resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- `\#{}`里面的名称对应的是Map里面的key名称。
- 这种方法适合传递多个参数，且参数易变能灵活传递的情况。

方法4：Java Bean传参法

```
public User selectUser(User user);

<select id="selectUser" parameterType="com.jourwon.pojo.User"
resultMap="UserResultMap">
    select * from user
    where user_name = #{userName} and dept_id = #{deptId}
</select>
```

- `\#{}`里面的名称对应的是User类里面的成员属性。
- 这种方法直观，需要建一个实体类，扩展不容易，需要加属性，但代码可读性强，业务逻辑处理方便，推荐使用。（推荐使用）。

5. 实体类属性名和表中字段名不一样，怎么办?

- 第1种：　通过在查询的SQL语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

```
<select id="getOrder" parameterType="int" resultType="com.jourwon.pojo.Order">
       select order_id id, order_no orderno ,order_price price form orders where
order_id=#{id};
</select>
```

- 第2种： 通过resultMap 中的<result>来映射字段名和实体类属性名的一一对应的关系。

```
<select id="getOrder" parameterType="int" resultMap="orderResultMap">
  select * from orders where order_id=#{id}
</select>

<resultMap type="com.jourwon.pojo.Order" id="orderResultMap">
    <!-用id属性来映射主键字段->
    <id property="id" column="order_id">
    <!-用result属性来映射非主键字段，property为实体类属性名，column为数据库表中的属性->
  <result property ="orderno" column ="order_no"/>
  <result property="price" column="order_price" />
</resultMap>
```

6. Mybatis是否可以映射Enum枚举类？

- Mybatis当然可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和getResult()接口方法。
- TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占位符参数和获取列查询结果。

7. `#{}和${}的区`

- `#{}`是占位符，预编译处理； `${}`是拼接符，字符串替换，没有预编译处理。
- Mybatis在处理`#{}`时， `#{}`传入参数是以字符串传入，会将SQL中的 #{} 替换为?号，调用PreparedStatement的set方法来赋值。
- `#{}` 可以有效的防止SQL注入，提高系统安全性； `${}`不能防止SQL 注入的
- `#{}` 变量替换是在DBMS 中； `${}`的变量替换是在 DBMS 外

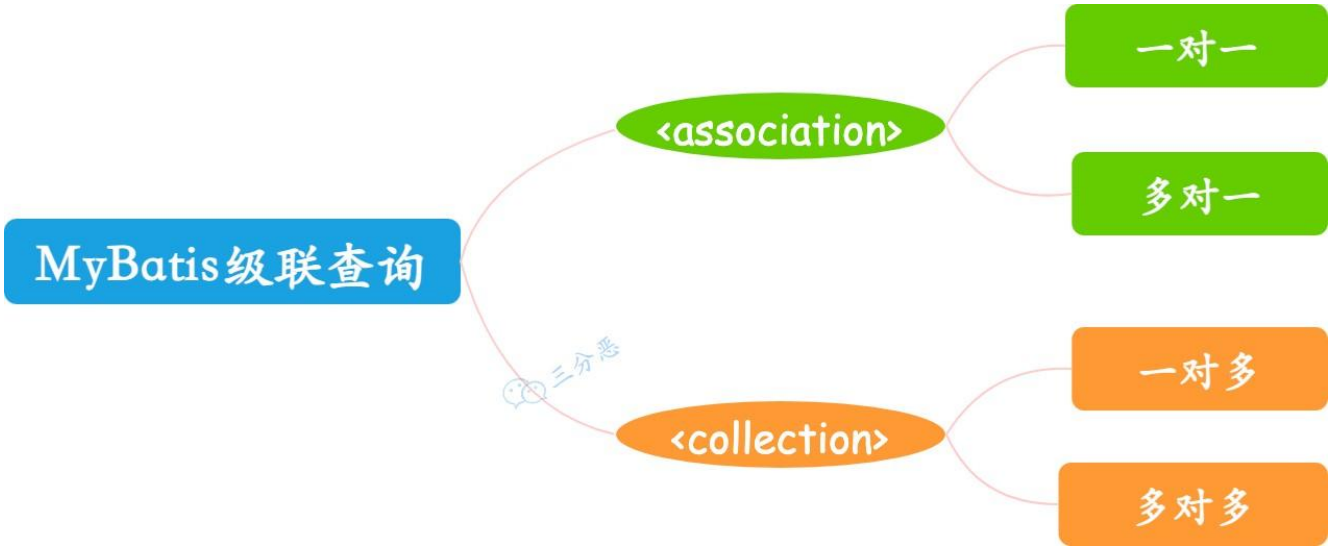
8. 模糊查询like语句该怎么写?

- 1 `’ %${question}%’` 可能引起SQL注入，不推荐
- 2 `"%"#{question}"%"` 注意：因为`#{…}`解析成sql语句时候，会在变量外侧自动加单引号’ '，所以这里 % 需要使用双引号" "，不能使用单引号 ’ '，不然会查不到任何结果。
- 3 `CONCAT('%',#{question},'%')` 使用CONCAT()函数，（推荐✨）
- 4 使用bind标签（不推荐）

```
<select id="listUserLikeUsername" resultType="com.jourwon.pojo.User">
  <bind name="pattern" value="'%' + username + '%'" />
  select id,sex,age,username,password from person where username LIKE #
{pattern}
</select>
```

9. Mybatis能执行一对一、一对多的关联查询吗？

当然可以，不止支持一对一、一对多的关联查询，还支持多对多、多对一的关联查询。



- 一对一<association>

比如订单和支付是一对一的关系，这种关联的实现：

实体类:

```
public class Order {
    private Integer orderId;
    private String orderDesc;

    /**
     * 支付对象
     */
    private Pay pay;
    //……
}
```

结果映射

```
<!-- 订单resultMap -->
<resultMap id="peopleResultMap" type="cn.fighter3.entity.Order">
    <id property="orderId" column="order_id" />
    <result property="orderDesc" column="order_desc"/>
    <!--一对一结果映射-->
    <association property="pay" javaType="cn.fighter3.entity.Pay">
        <id column="payId" property="pay_id"/>
        <result column="account" property="account"/>
    </association>
</resultMap>
```

查询就是普通的关联查

```
<select id="getTeacher" resultMap="getTeacherMap" parameterType="int">
    select * from order o
     left join pay p on o.order_id=p.order_id
    where  o.order_id=#{orderId}
</select>
```

- 一对多<collection>

比如商品分类和商品，是一对多的关系。

- 实体类

```
public class Category {
    private int categoryId;
    private String categoryName;

    /**
    * 商品列表
    **/
    List<Product> products;
    //.....
}
```

- 结果映射

```
<resultMap type="Category" id="categoryBean">
    <id column="categoryId" property="category_id" />
    <result column="categoryName" property="category_name" />

    <!-- 一对多的关系 -->
    <!-- property: 指的是集合属性的值, ofType：指的是集合中元素的类型 -->
    <collection property="products" ofType="Product">
        <id column="product_id" property="productId" />
        <result column="productName" property="productName" />
        <result column="price" property="price" />
    </collection>
</resultMap>
```

- 查询

查询就是一个普通的关联查询

```
<!-- 关联查询分类和产品表 -->
<select id="listCategory" resultMap="categoryBean">
    select c.*, p.* from category_ c left join product_ p on c.id = p.cid
</select>
```

那么多对一、多对多怎么实现呢？还是利用<association>和<collection>，篇幅所限，这里就不展开了。

10. Mybatis是否支持延迟加载？原理？

- Mybatis支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=truelfalse。
- 它的原理是，使用CGLIB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，比如调用a.getB().getName()，拦截器nvoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。
- 当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

11. 如何获取生成的主键?

- 新增标签中添加：keyProperty=" ID " 即可

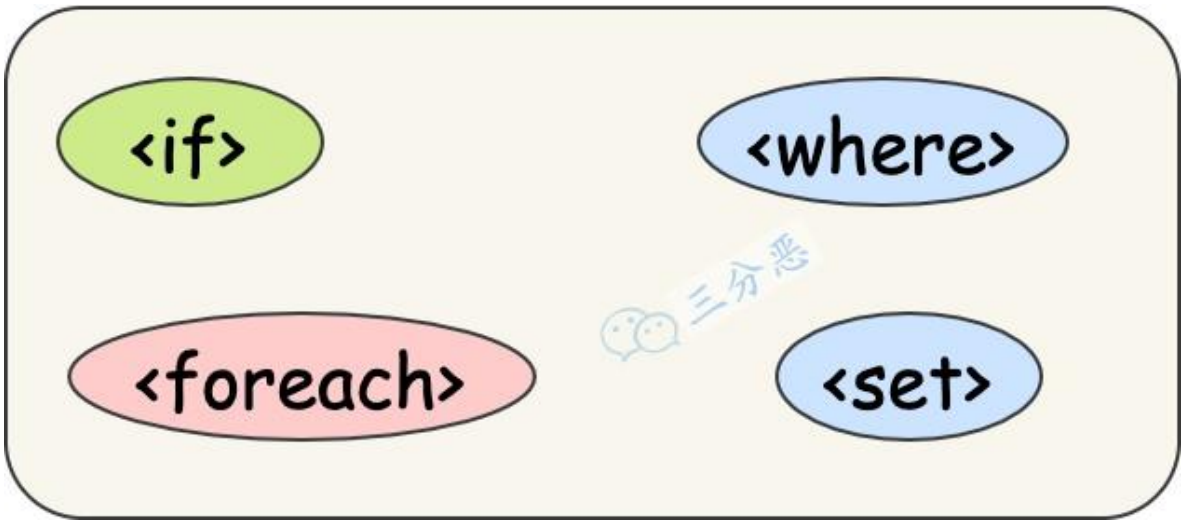
```
<insert id="insert" useGeneratedKeys="true" keyProperty="userId" >
    insert into user(
    user_name, user_password, create_time)
    values(#{userName}, #{userPassword} , #{createTime, jdbcType= TIMESTAMP})
</insert>
```

- 这时候就可以完成回填主键

```
mapper.insert(user);
user.getId;
```

12. MyBatis支持动态SQL吗？

MyBatis中有一些支持动态SQL的标签，它们的原理是使用OGNL从SQL参数对象中计算表达式的值，根据表达式的值动态拼接SQL，以此来完成动态SQL的功能。



- if

根据条件来组成where子句

```
<select id="findActiveBlogWithTitleLike"
   resultType="Blog">
SELECT * FROM BLOG
WHERE state = 'ACTIVE'
<if test="title != null">
  AND title like #{title}
</if>
</select>
```

- choose (when, otherwise)

这个和Java 中的 switch 语句有点像

```
<select id="findActiveBlogLike"
   resultType="Blog">
SELECT * FROM BLOG WHERE state = 'ACTIVE'
<choose>
  <when test="title != null">
    AND title like #{title}
  </when>
  <when test="author != null and author.name != null">
    AND author_name like #{author.name}
  </when>
  <otherwise>
    AND featured = 1
  </otherwise>
</choose>
</select>
```

- trim (where, set)
- <where>可以用在所有的查询条件都是动态的情况

```
<select id="findActiveBlogLike"
   resultType="Blog">
SELECT * FROM BLOG
<where>
  <if test="state != null">
       state = #{state}
  </if>
  <if test="title != null">
      AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
      AND author_name like #{author.name}
  </if>
</where>
</select>
```

- <set> 可以用在动态更新的时候

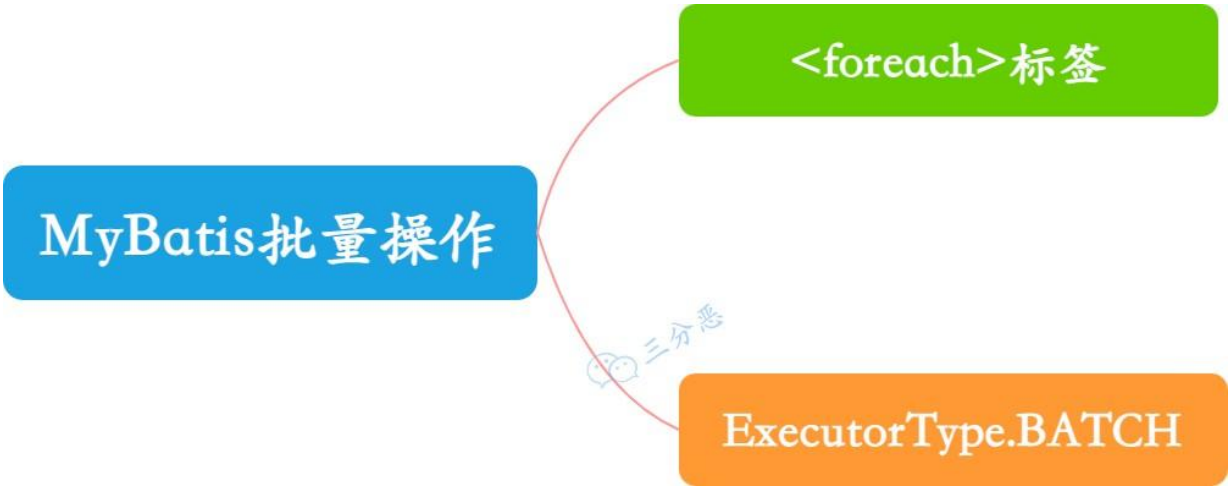
```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

- foreach

  看到名字就知道了，这个是用来循环的，可以对集合进行遍历

```
<select id="selectPostIn" resultType="domain.blog.Post">
SELECT *
FROM POST P
<where>
  <foreach item="item" index="index" collection="list"
      open="ID in (" separator="," close=")" nullable="true">
        #{item}
  </foreach>
</where>
</select>
```

13. MyBatis如何执行批量操作？



第一种方法：使用foreach标签

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。foreach标签的属性主要有item，index，collection，open，separator，close。

- item　　表示集合中每一个元素进行迭代时的别名，随便起的变量名；
- 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，不常用；
- open　　表示该语句以什么开始，常用“(”；
- separator　　表示在每次进行迭代之间以什么符号作为分隔符，常用“,”；
- close　　表示以什么结束，常用“)”。

在使用foreach的时候最关键的也是最容易出错的就是collection属性，该属性是必须指定的，但是在不同情况下，该属性的值是不一样的，主要有以下3种情况：

1. 如果传入的是单参数且参数类型是一个List的时候，collection属性值为list
2. 如果传入的是单参数且参数类型是一个array数组的时候，collection的属性值为array
3. 如果传入的参数是多个的时候，我们就需要把它们封装成一个Map了，当然单参数也可以封装成map，实际上如果你在传入参数的时候，在MyBatis里面也是会把它封装成一个Map的，map的key就是参数名，所以这个时候collection属性值就是传入的List或array对象在自己封装的map里面的key

看看批量保存的两种用法：

```
<!-- MySQL下批量保存，可以foreach遍历 mysql支持values(),(),()语法 --> //推  使用
<insert id="addEmpsBatch">
    INSERT INTO emp(ename,gender,email,did)
    VALUES
    <foreach collection="emps" item="emp" separator=",">
        (#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id})
    </foreach>
</insert>
```

```
<!-- 这种方式需要数据库连接属性allowMutiQueries=true的支持
 如jdbc.url=jdbc:mysql://localhost:3306/mybatis?allowMultiQueries=true -->
<insert id="addEmpsBatch">
    <foreach collection="emps" item="emp" separator=";">

        INSERT INTO emp(ename,gender,email,did)
        VALUES(#{emp.eName},#{emp.gender},#{emp.email},#{emp.dept.id})
    </foreach>
</insert>
```

第二种方法：使用ExecutorType.BATCH

- Mybatis内置的ExecutorType有3种，默认为simple，该模式下它为每个语句的执行创建一个新的预处理语 句，单条提交sql；而batch模式重复使用已经预处理的语句，并且批量执行所有更新语句，显然batch性能将更优； 但batch模式也有自己的问题，比如在Insert操作时，在事务没有提交之前，是没有办法获取到自增的 id，在某些情况下不符合业务的需求。

具体用法如下：

```
//批量保存方法测试
@Test
public void testBatch() throws IOException{
    SqlSessionFactory sqlSessionFactory = getSqlSessionFactory();
    //可以执行批量操作的sqlSession
    SqlSession openSession = sqlSessionFactory.openSession(ExecutorType.BATCH);

    //批量保存执行前时间
    long start = System.currentTimeMillis();
    try {
        EmployeeMapper mapper = openSession.getMapper(EmployeeMapper.class);
        for (int i = 0; i < 1000; i++) {
            mapper.addEmp(new Employee(UUID.randomUUID().toString().substring(0, 5),
"b", "1"));
        }

        openSession.commit();
        long end = System.currentTimeMillis();
        //批量保存执行后的时间
        System.out.println("执行时长" + (end - start));
        //批量  预编译sql一次==》设置参数==》10000次==》执行1次   677
        //非批量   （预编译=设置参数=执行 ）==》10000次   1121

    } finally {
        openSession.close();
    }
}
```

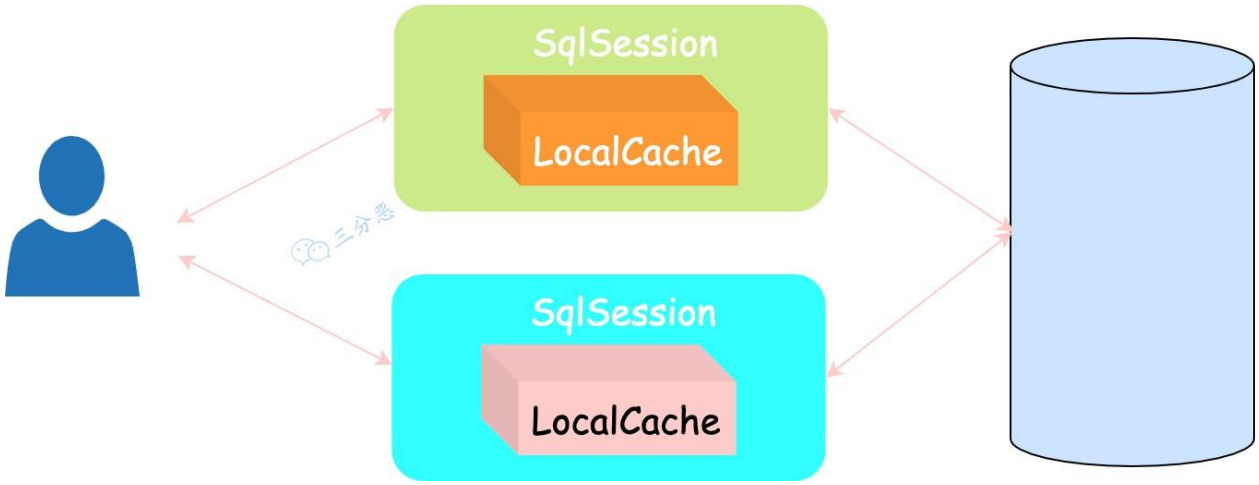
- mapper和mapper.xml如下

```
public interface EmployeeMapper {
    //批量保存员工
    Long addEmp(Employee employee);
}
```

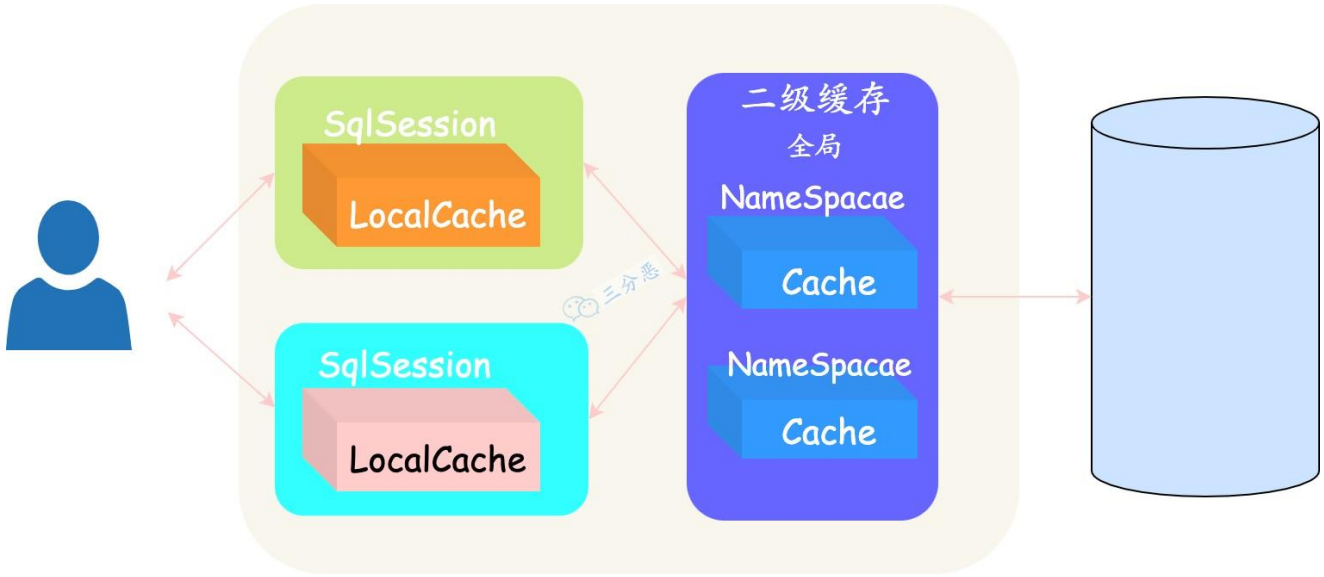
```
<mapper namespace="com.jourwon.mapper.EmployeeMapper"
     <!--批量保存员工 -->
    <insert id="addEmp">
        insert into employee(lastName,email,gender)
        values(#{lastName},#{email},#{gender})
    </insert>
</mapper>
```

14. 说说Mybatis的一级、二级缓存？

1. 一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为SqlSession，各个SqlSession之间的缓存相互隔离，当 Session flush 或 close 之后，该 SqlSession 中的所有 Cache 就将清空，MyBatis默认打开一级缓存。



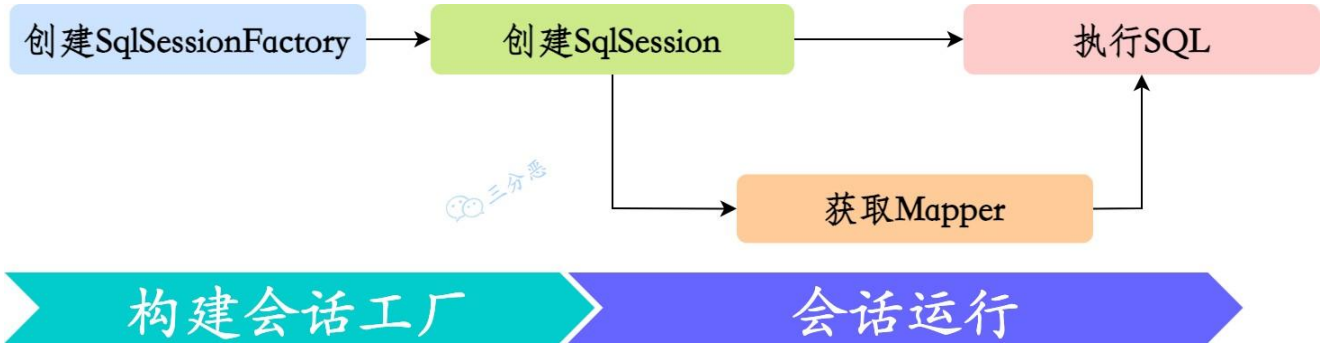
2. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同之处在于其存储作用域为 Mapper(Namespace)，可以在多个SqlSession之间共享，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置。



原理

15. 能说说MyBatis的工作原理吗？

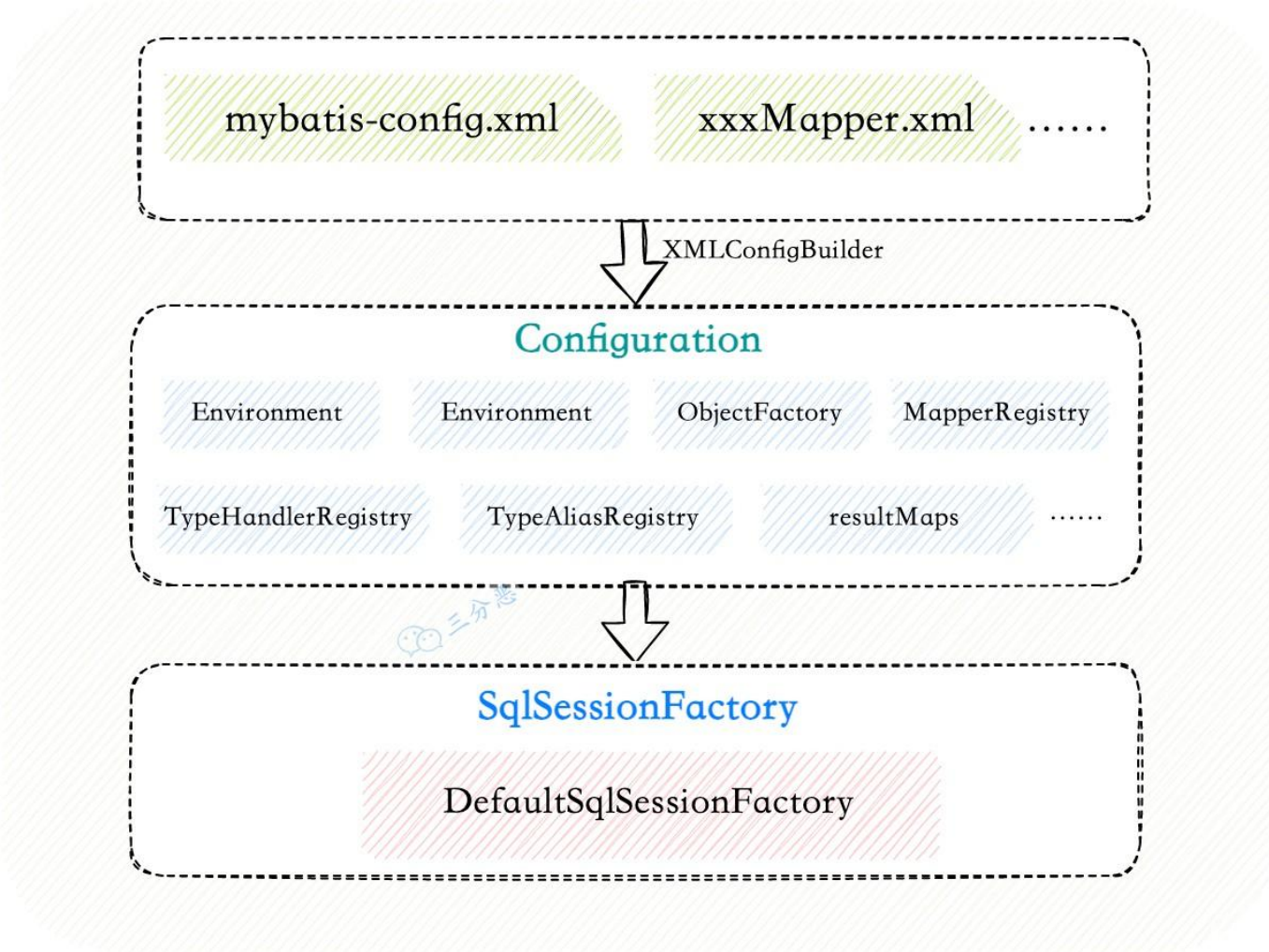
我们已经大概知道了MyBatis的工作流程，按工作原理，可以分为两大步： 生成会话工厂 、 会话运行 。



MyBatis是一个成熟的框架，篇幅限制，这里抓大放小，来看看它的主要工作流程。

> 构建会话工厂

构造会话工厂也可以分为两步：



- 获取配置

获取配置这一步经过了几步转化，最终由生成了一个配置类Configuration实例，这个配置类实例非常重要，主要作用包括：

- 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，比如MyBatis的别名，还有其它的一些重要的类对象，像插件、映射器、ObjectFactory等等
- 提供一个单例，作为会话工厂构建的重要参数
- 它的构建过程也会初始化一些环境变量，比如数据源

```
public SqlSessionFactory build(Reader reader, String environment, Properties
properties) {
      SqlSessionFactory var5;
      //省略异常处理
          //xml配置构建器
          XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment,
properties);
          //通过转化的Configuration构建SqlSessionFactory
          var5 = this.build(parser.parse());
}
```

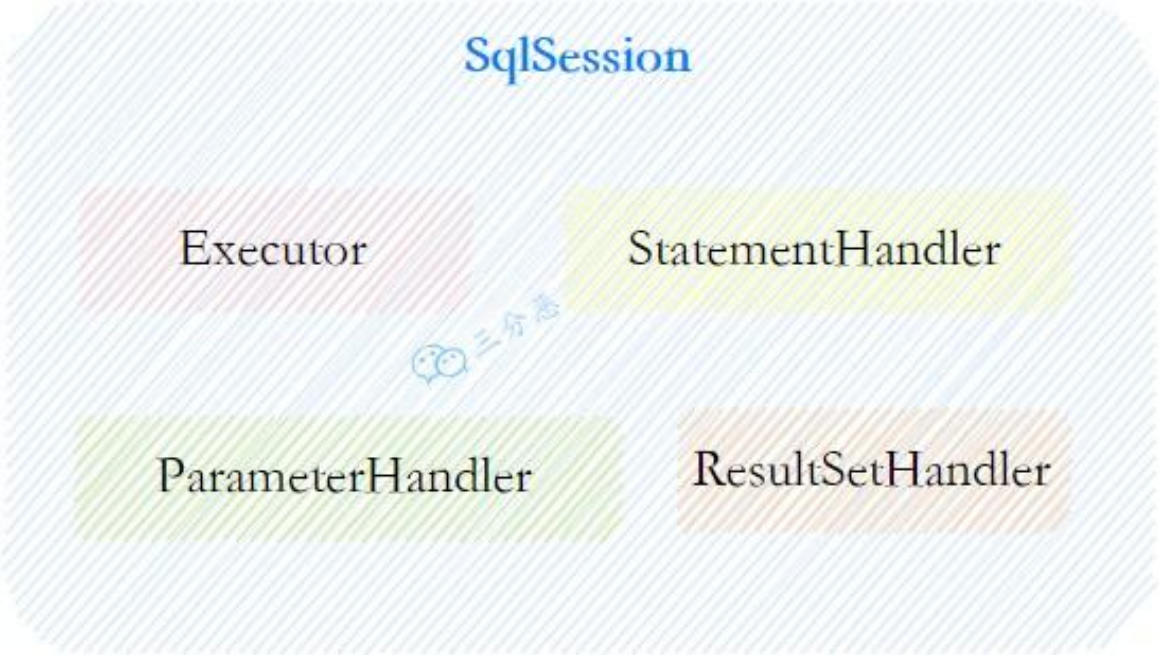
- 构建SqlSessionFactory

SqlSessionFactory只是一个接口，构建出来的实际上是它的实现类的实例，一般我们用的都是它的实现类DefaultSqlSessionFactory，

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

> 会话运行

会话运行是MyBatis最复杂的部分，它的运行离不开四大组件的配合：



- Executor（执行器）

Executor起到了至关重要的作用，SqlSession只是一个门面，相当于客服，真正干活的是是Executor，就像是默默无闻的工程师。它提供了相应的查询和更新方法，以及事务方法。

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

- StatementHandler（数据库会话器）

StatementHandler，顾名思义，处理数据库会话的。我们以SimpleExecutor为例，看一下它的查询方法，先生成了一个StatementHandler实例，再拿这个handler去执行query。

```
 public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds,
ResultHandler resultHandler, BoundSql boundSql) throws SQLException
    { Statement stmt = null;

    List var9;
    try {
        Configuration configuration = ms.getConfiguration();
        StatementHandler handler = configuration.newStatementHandler(this.wrapper, ms,
parameter, rowBounds, resultHandler, boundSql);
        stmt = this.prepareStatement(handler, ms.getStatementLog());
        var9 = handler.query(stmt, resultHandler);
    } finally {
        this.closeStatement(stmt);
    }

    return var9;
}
```

再以最常用的PreparedStatementHandler看一下它的query方法，其实在上面的 prepareStatement 已经对参数进行了预编译处理，到了这里，就直接执行sql，使用ResultHandler处理返回结果。

```
public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws
SQLException {
    PreparedStatement ps = (PreparedStatement)statement;
    ps.execute();
    return this.resultSetHandler.handleResultSets(ps);
}
```

- ParameterHandler （参数处理器）

PreparedStatementHandler里对sql进行了预编译处理

```
public void parameterize(Statement statement) throws SQLException
    { this.parameterHandler.setParameters((PreparedStatement)statement);
}
```

这里用的就是ParameterHandler，setParameters的作用就是设置预编译SQL语句的参数。里面还会

用到typeHandler类型处理器，对类型进行处理。

```
public interface ParameterHandler {
    Object getParameterObject();

    void setParameters(PreparedStatement var1) throws SQLException;
}
```

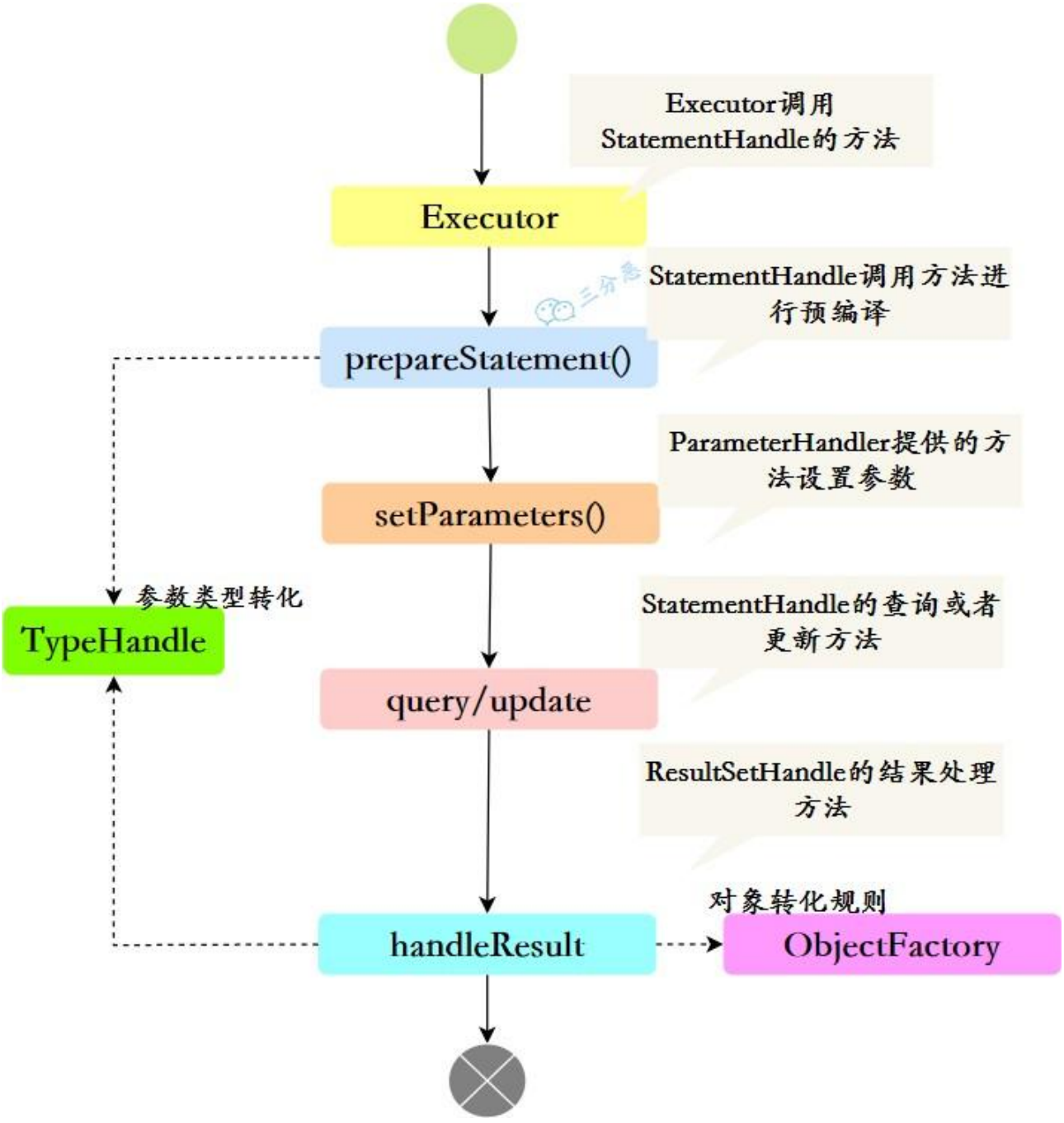
- ResultSetHandler（结果处理器）

  我们前面也看到了，最后的结果要通过ResultSetHandler来进行处理，handleResultSets这个方法就是用来包装结果集的。Mybatis为我们提供了一个DefaultResultSetHandler，通常都是用这个实现类去进行结果的处理的。

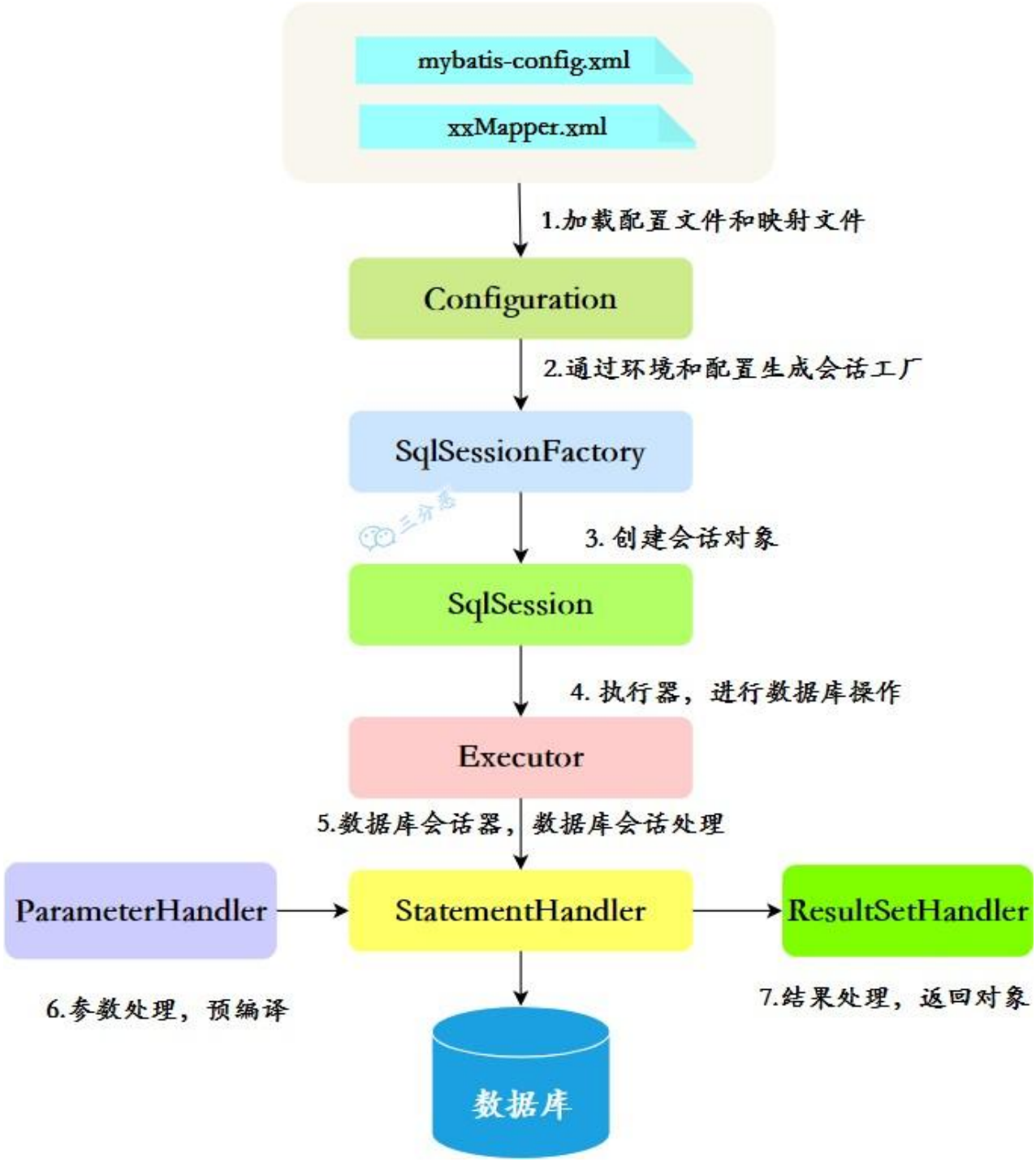
```
public interface ResultSetHandler {
  <E> List<E> handleResultSets(Statement var1) throws SQLException;

  <E> Cursor<E> handleCursorResultSets(Statement var1) throws SQLException;

  void handleOutputParameters(CallableStatement var1) throws SQLException;
}
```

它会使用typeHandle处理类型，然后用ObjectFactory提供的规则组装对象，返回给调用者。整体

上总结一下会话运行：



> 我们最后把整个的工作流程串联起来，简单总结一下：



1. 读取 MyBatis 配置文件——mybatis-config.xml 、加载映射文件——映射文件即 SQL 映射文件，文件中配置了操作数据库的 SQL 语句。最后生成一个配置对象。
2. 构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。
3. 创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。
4. Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 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，串联起参数映射的处理和运行结果映射的处理。
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，返回相应的对象。

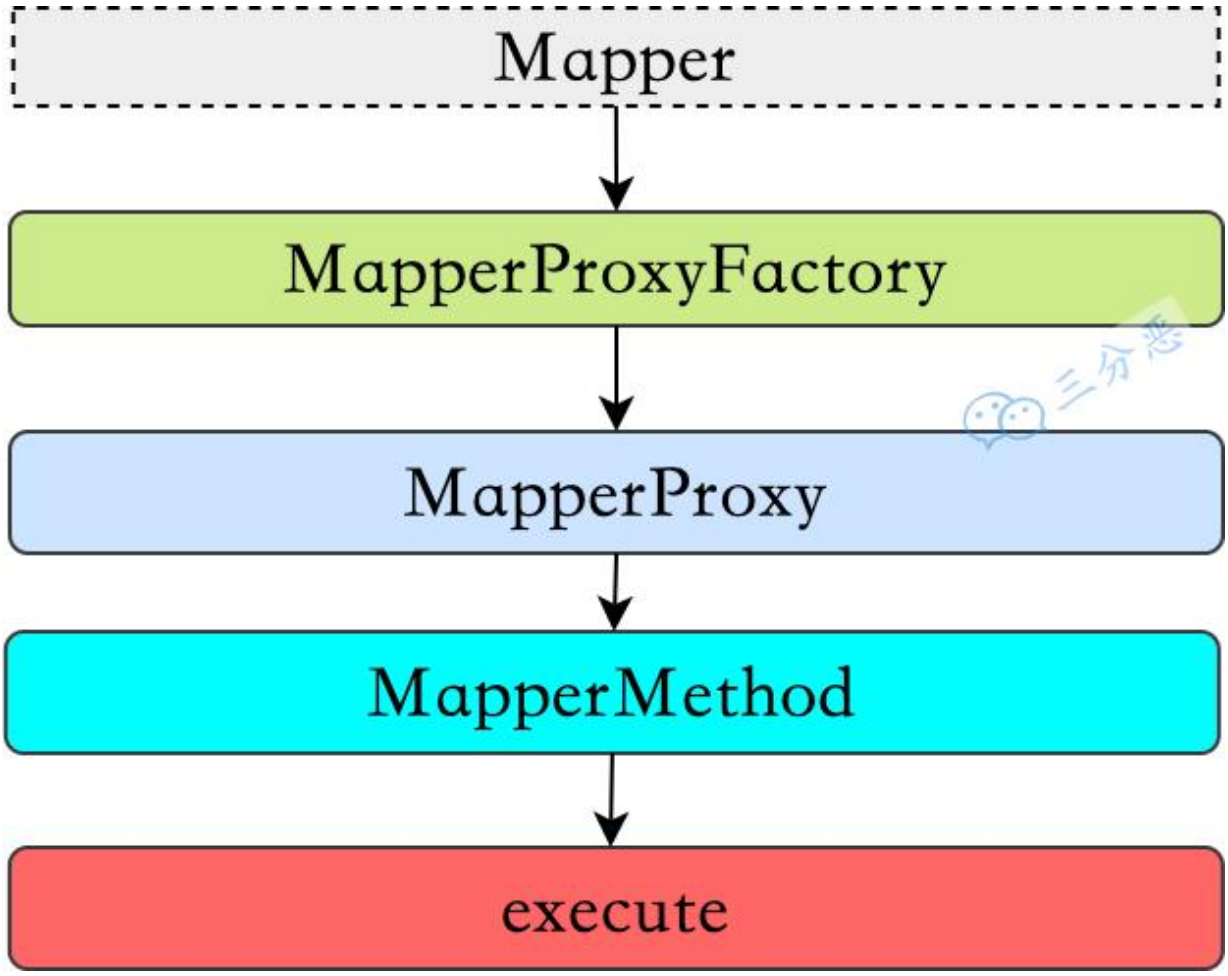
16. MyBatis的功能架构是什么样的？

我们一般把Mybatis的功能架构分为三层：

- API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口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：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。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：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

17. 为什么Mapper接口不需要实现类？

四个字回答：**动态代理**，我们来看一下获取Mapper的过程：



- 获取Mapper

我们都知道定义的Mapper接口是没有实现类的，Mapper映射其实是通过**动态代理**实现的。

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
```

七拐八绕地进去看一下，发现获取Mapper的过程，需要先获取MapperProxyFactory——Mapper代理工厂。

```
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);
        }
    }
}
```

- MapperProxyFactory

MapperProxyFactory的作用是生成MapperProxy（Mapper代理对象）。

```
public class MapperProxyFactory<T> {
  private final Class<T> mapperInterface;
  ……
  protected T newInstance(MapperProxy<T> mapperProxy) {
      return Proxy.newProxyInstance(this.mapperInterface.getClassLoader(), new Class[]
{this.mapperInterface}, mapperProxy);
  }

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

这里可以看到动态代理对接口的绑定，它的作用就是生成动态代理对象（占位），而代理的方法被放到了MapperProxy中。

- MapperProxy

MapperProxy里，通常会生成一个MapperMethod对象，它是通过cachedMapperMethod方法对其进行初始化的，然后执行excute方法。

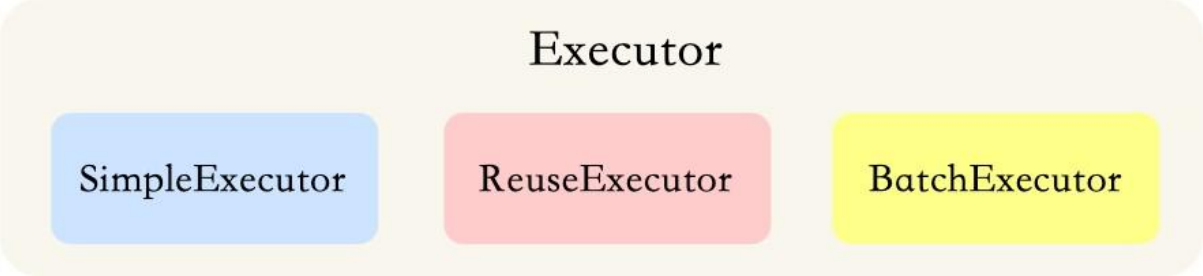
```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    { try {
        return Object.class.equals(method.getDeclaringClass()) ? method.invoke(this,
args) : this.cachedInvoker(method).invoke(proxy, method, args, this.sqlSession);
    } catch (Throwable var5) {
        throw ExceptionUtil.unwrapThrowable(var5);
    }
}
```

- MapperMethod

MapperMethod里的excute方法，会真正去执行sql。这里用到了命令模式，其实绕一圈，最终它还是通过SqlSession的实例去运行对象的sql。

```
public Object execute(SqlSession sqlSession, Object[] args)
      { Object result;
      Object param;
      ……
      case SELECT:
          if (this.method.returnsVoid() && this.method.hasResultHandler())
              { this.executeWithResultHandler(sqlSession, args);
              result = null;
          } else if (this.method.returnsMany()) {
              result = this.executeForMany(sqlSession, args);
          } else if (this.method.returnsMap()) {
              result = this.executeForMap(sqlSession, args);
          } else if (this.method.returnsCursor()) {
              result = this.executeForCursor(sqlSession, args);
          } else {
              param = this.method.convertArgsToSqlCommandParam(args);
              result = sqlSession.selectOne(this.command.getName(), param);
              if (this.method.returnsOptional() && (result == null ||
!this.method.getReturnType().equals(result.getClass()))) {
                  result = Optional.ofNullable(result);
              }
          }
          break;
        ……
  }
```

18. Mybatis都有哪些Executor执行器？



- Mybatis有三种基本的Executor执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

- **SimpleExecutor**：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。
- **ReuseExecutor**：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。
- **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()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

> Mybatis中如何指定使用哪一种Executor执行器？

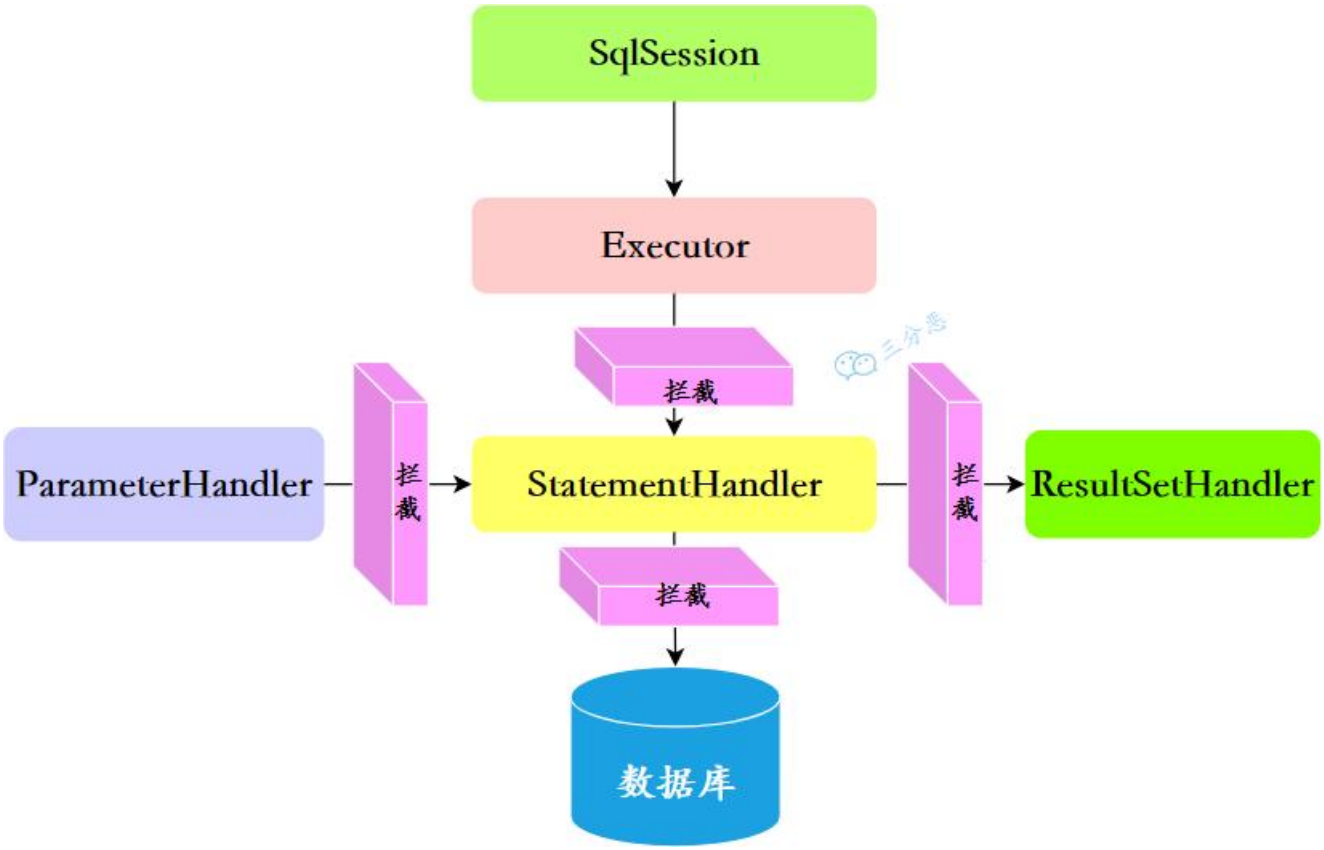
- 在Mybatis配置文件中，在设置（settings）可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数，如 SqlSession `openSession(ExecutorType execType)`。
  配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。

插件

19. 说说Mybatis的插件运行原理，如何编写一个插件？

> 插件的运行原理？

Mybatis会话的运行需要ParameterHandler、ResultSetHandler、StatementHandler、Executor这四大对象的配合，插件的原理就是在这四大对象调度的时候，插入一些我我们自己的代码。



Mybatis使用JDK的动态代理，为目标对象生成代理对象。它提供了一个工具类Plugin，实现了InvocationHandler接口。

使用Plugin生成代理对象，代理对象在调用方法的时候，就会进入invoke方法，在invoke方法中，如果存在签名的拦截方法，插件的intercept方法就会在这里被我们调用，然后就返回结果。如果不存在签名方法，那么将直接反射调用我们要执行的方法。

> 如何编写一个插件？

我们自己编写MyBatis 插件，只需要实现拦截器接口 `Interceptor (org.apache.ibatis. plugin Interceptor )` ，在实现类中对拦截对象和方法进行处理。实现

- Mybatis的Interceptor接口并重写intercept()方法

这里我们只是在目标对象执行目标方法的前后进行了打印；

```
public class MyInterceptor implements Interceptor {
    Properties props=null;

    @Override
    public Object intercept(Invocation invocation) throws Throwable
        { System.out.println("before……");
        //如果当前代理的是一 非代理对象，那么就会调用真实拦 对象的方法
        // 如果不是 就会调用下 件代理对象的invoke方法
        Object obj=invocation.proceed();
        System.out.println("after……");
        return obj;
    }
}
```

- 然后再给插件编写注解，确定要拦截的对象，要拦截的方法

```
@Intercepts({@Signature(
        type = Executor.class,  //确定要拦 的对象
        method = "update",         //确定要拦 的方法
        args = {MappedStatement.class,Object.class}     //拦 方法的参数
)})
public class MyInterceptor implements Interceptor
    { Properties props=null;

    @Override
    public Object intercept(Invocation invocation) throws Throwable
        { System.out.println("before……");
        //如果当前代理的是一 非代理对象，那么就会调用真实拦 对象的方法
        // 如果不是 就会调用下 件代理对象的invoke方法
        Object obj=invocation.proceed();
        System.out.println("after……");
        return obj;
    }
}
```

- 最后，再MyBatis配置文件里面配置插件

```
<plugins>
    <plugin interceptor="xxx.MyPlugin">
       <property name="dbType",value="mysql"/>
    </plugin>
</plugins>
```

20. MyBatis是如何进行分页的？分页插件的原理是什么？

> MyBatis是如何分页的？

MyBatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

> 分页插件的原理是什么？

- 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，拦截Executor的query方法
- 在执行查询的时候，拦截待执行的sql，然后重写sql，根据dialect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`，拦截sql后重写为：`select t.* from (select * from student) t limit 0, 10`

可以看一下一个大概的MyBatis通用分页拦截器：

```
@SuppressWarnings({"rawtypes", "unchecked"})
@Intercepts(
        @Signature(
                type = Executor.class,          //拦截对象
                method = "query",               //拦截方法
                args = {MappedStatement.class, Object.class,
                        RowBounds.class, ResultHandler.class}))
public class PageInterceptor implements Interceptor {

    private static final List<ResultMapping> EMPTY_RESULTMAPPING = new ArrayList<ResultMapping>();

    //数据库方言接口
    private Dialect dialect;
    private Field additionalParametersField;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        //获取拦截方法的参数
        Object[] args = invocation.getArgs();
        MappedStatement ms = (MappedStatement) args[0];
        Object parameterObject = args[1];
        RowBounds rowBounds = (RowBounds) args[2];
        //调用方法判断是否需要进行分页，如果不需要，直接返回结果
        if (!dialect.skip(ms.getId(), parameterObject, rowBounds)) {
            ResultHandler resultHandler = (ResultHandler) args[3];
            //当前的目标对象
            Executor executor = (Executor) invocation.getTarget();
            BoundSql boundSql = ms.getBoundSql(parameterObject);
            //反射获取动态参数
```

```
        //反射获取动态参数
        Map<String, Object> additionalParameters = (Map<String, Object>)
                additionalParametersField.get(boundSql);
        //判断是否需要进行 count 查询
        if (dialect.beforeCount(ms.getId(), parameterObject, rowBounds)) {
            //根据当前的 ms 创建 个返回值为 Long 类型的 ms
            MappedStatement countMs = newMappedStatement(ms, Long.class);
            //创建count查询的缓存 key
            CacheKey countKey = executor.createCacheKey(
                    countMs,
                    parameterObject,
                    RowBounds.DEFAULT,
                    boundSql);
            //调用方言获取count sql
            String countSql = dialect.getCountSql(
                    boundSql,
                    parameterObject,
                    rowBounds,
                    countKey);
            BoundSql countBoundSql = new BoundSql(
                    ms.getConfiguration(),
                    countSql,
                    boundSql.getParameterMappings(),
                    parameterObject);
            //当使用动态SQL时，可能会产生临时的参数
            //这些参数需要手动设直到新的 BoundSql中
            for (String key : additionalParameters.keySet()) {
                countBoundSql.setAdditionalParameter(
                        key, additionalParameters.get(key));
            }
            //执行 count 查询
            Object countResultList = executor.query(
                    countMs,
                    parameterObject,
                    RowBounds.DEFAULT,
                    resultHandler,
                    countKey,
                    countBoundSql);
            Long count = (Long) ((List) countResultList).get(0);
            //处理查询总数
            dialect.afterCount(count, parameterObject, rowBounds);
            if (count == 0L) {
                //当查询总数为 时，直接返回空的结果
                return dialect.afterPage(
                        new ArrayList(),
                        parameterObject,
                        rowBounds
                );
            }
        }
        // 判断是否需要进行分页查询
        if (dialect.beforePage(ms.getId(), parameterObject, rowBounds)) {
            //生成分页的缓存 key
            CacheKey pageKey = executor.createCacheKey(
                    ms,
                    parameterObject,
                    rowBounds,
                    boundSql);
            //调用方言获取分页 sql
            String pageSql = dialect.getPageSql(
                    boundSql,
                    parameterObject,
```

```
                    rowBounds,
                    pageKey);
            BoundSql pageBoundSql = new BoundSql(
                    ms.getConfiguration(),
                    pageSql,
                    boundSql.getParameterMappings(),
                    parameterObject);
            //设直动态参数
            for (String key : additionalParameters.keySet()) {
                pageBoundSql.setAdditionalParameter(
                        key, additionalParameters.get(key));
                //执行分页查询
                List resultList = executor.query(
                        ms,
                        parameterObject,
                        RowBounds.DEFAULT,
                        resultHandler,
                        pageKey,
                        pageBoundSql);
                return dialect.afterPage(resultList, parameterObject, rowBounds);
            }
        }
    }
    //返回默认查询
    return invocation.proceed();
}

/**
 * 根据现有的 ms 创建一个新的返回值类型，使用新的返回值类型
 *
 * @param ms
 * @param resultType
 * @return
 */
private MappedStatement newMappedStatement(MappedStatement ms, Class<?> resultType) {
    MappedStatement.Builder builder = new MappedStatement.Builder(
            ms.getConfiguration(),
            ms.getId() + "_Count",
            ms.getSqlSource(),
            ms.getSqlCommandType());
    builder.resource(ms.getResource());
    builder.fetchSize(ms.getFetchSize());
    builder.statementType(ms.getStatementType());
    builder.keyGenerator(ms.getKeyGenerator());
    if (ms.getKeyProperties() != null
            && ms.getKeyProperties().length != 0) {
        StringBuilder keyProperties = new StringBuilder();
        for (String keyProperty : ms.getKeyProperties()) {
            keyProperties.append(keyProperty).append(",");
        }
        keyProperties.delete(keyProperties.length() - 1, keyProperties.length());
        builder.keyProperty(keyProperties.toString());
    }
    builder.timeout(ms.getTimeout());
    builder.parameterMap(ms.getParameterMap());
    //count 查询返回值 int
    List<ResultMap> resultMaps = new ArrayList<ResultMap>();
    ResultMap resultMap = new ResultMap.Builder(
            ms.getConfiguration(),
            ms.getId(),
            resultType,
            EMPTY_RESULTMAPPING).build();
    resultMaps.add(resultMap);
```

```
        builder.resultMaps(resultMaps);
        builder.resultSetType(ms.getResultSetType());
        builder.cache(ms.getCache());
        builder.flushCacheRequired(ms.isFlushCacheRequired());
        builder.useCache(ms.isUseCache());
        return builder.build();
    }

    @Override
    public Object plugin(Object target) {
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) {
        String dialectClass = properties.getProperty("dialect");
        try {
            dialect = (Dialect) Class.forName(dialectClass).newInstance();
        } catch (Exception e) {
            throw new RuntimeException("使用PageInterceptor分页插件时，必须设置dialect属性");
        }
        dialect.setProperties(properties);
        try {
            //反射获取BoundSql中的additionalParameters属性
            additionalParametersField = BoundSql.class.getDeclaredField("additionalParameters");
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        }
    }

}
```