
目录

1	MyBatis 快速入门	4
1.1	为什么需要 ORM 框架？	4
1.2	MyBatis 快速入门	4
2	MyBatis 开发要点	5
2.1	resultType 还是 resultMap?	5
2.1.1	resultType	5
2.1.2	resultMap	5
2.1.3	到底应该用 resultType 还是 resultMap?	5
2.2	怎么传递多个参数？	6
2.3	怎么样获取主键？	6
2.3.1	通过 insert/update 标签相关属性	6
2.3.2	通过 selectKey 元素	7
2.4	SQL 元素和 SQL 的参数	7
2.5	动态 SQL	8
2.5.1	动态 SQL 元素	8
2.5.2	示例代码说明	8
2.5.3	通过 Mybatis 怎么样进行批量的操作	8
2.6	代码生成器	9
2.7	关联查询	9
2.7.1	关联查询几个需要注意的细节	9
2.7.2	一对关联嵌套结果方式	10
2.7.3	一对关联嵌套查询方式	10
2.7.4	一对多关联	11
2.7.5	多对多关联	11
2.8	缓存	11
2.8.1	一级缓存	11
2.8.2	二级缓存	11
2.8.3	缓存调用过程	12

3	MyBatis 源码概述.....	13
3.1	怎么下载 MyBatis 源码?	13
3.2	源码架构分析.....	13
3.3	外观模式（门面模式）.....	14
3.4	面向对象设计需要遵循的六大设计原则.....	14
4.	日志模块分析.....	15
4.1	日志模块需求分析.....	15
4.2	适配器模式.....	15
4.3	怎么实现优先加载日志组件?	16
4.4	代理模式和动态代理.....	17
4.4.1	静态代理.....	17
4.4.2	动态代理.....	17
4.5	优雅的增强日志功能.....	18
5.	数据源模块分析.....	19
5.1	简单工厂模式.....	19
5.2	工厂模式.....	20
5.3	数据源的创建.....	21
5.4	数据库连接池技术解析.....	21
6.	缓存模块分析.....	23
6.1	需求分析.....	23
6.2	装饰器模式.....	24
6.3	装饰器在缓存模块的使用.....	25
6.4	缓存的唯一标识 CacheKey.....	25
7.	反射模块分析.....	26
8.	MyBatis 流程概述.....	26
9.	第一阶段：配置加载阶段.....	27
9.1	建造者模式.....	27
9.1.1	什么是建造者模式.....	27
9.1.2	与工厂模式区别.....	28
9.2	配置加载的核心类.....	28

9.2.1	建造器三个核心类.....	28
9.2.2	关于 Configuration 对象.....	29
9.3	配置加载过程.....	30
10.	第二阶段：代理封装阶段.....	33
10.1	Mybatis 的接口层.....	34
10.1.1	SqlSession.....	34
10.1.2	策略模式.....	35
10.1.3	SqlSessionFactory.....	35
10.2	binding 模块分析.....	36
10.2.1	binding 模块核心类.....	36
10.2.2	binding 模块运行流程.....	37
11.	第三个阶段：数据访问阶段.....	38
11.1	关于 Executor 组件.....	38
11.2	Executor 中的模板模式.....	38
11.3	Executor 的三个重要小弟.....	40
11.4	关于 StatementHandler.....	41
11.5	关于 ResultHandler.....	42
12.	与 spring 结合原理.....	42
12.1	MyBatis-Spring 是什么.....	42
12.2	MyBatis-Spring 集成配置最佳实践.....	43
12.3	MyBatis-Spring 集成原理分析.....	44
13.	插件开发.....	47
13.1	理解插件.....	47
13.2	插件开发快速入门.....	47
13.3	责任链模式.....	49
13.4	插件模块源码分析.....	49
14.	MyBatis 面试题集锦.....	51

1 MyBatis 快速入门

1.1 为什么需要 ORM 框架？

传统的 JDBC 编程存在的弊端：

- ✓ 工作量大，操作数据库至少要 5 步；
- ✓ 业务代码和技术代码耦合；
- ✓ 连接资源手动关闭，带来了隐患；

MyBatis 前身是 iBatis, 其源于 “Internet” 和 “ibatis”的组合，本质是一种半自动的 ORM 框架，除了 POJO 和映射关系之外，还需要编写 SQL 语句；Mybatis 映射文件三要素：SQL、映射规则和 POJO；

1.2 MyBatis 快速入门

步骤如下：

1. 加入 mybatis 的依赖，版本 3.5.x
2. 添加 mybatis 的配置文件，包括 MyBatis 核心文件和 mapper.xml 文件
3. 场景介绍：基于 t_user 表单数据查询、多数据查询；
4. 编写实体类、mapper 接口以及 mapper xml 文件；
5. 编写实例代码：com.enjoylearning.mybatis.MybatisDemo.quickStart

核心类分析：

1. **SqlSessionFactoryBuilder**: 读取配置信息创建 SqlSessionFactory，建造者模式，方法级别生命周期；
2. **SqlSessionFactory**: 创建 SqlSession，工厂单例模式，存在于程序的整个生命周期；
3. **SqlSession**: 代表一次数据库连接，一般通过调用 Mapper 访问数据库，也可以直接发送 SQL 执行，；线程不安全，要保证线程独享（方法级）；
4. **SQL Mapper**: 由一个 Java 接口和 XML 文件组成，包含了要执行的 SQL 语句和结果集映射规则。方法级别生命周期；

2 MyBatis 开发要点

2.1 resultType 还是 resultMap?

2.1.1 resultType

resultType: 当使用 resultType 做 SQL 语句返回结果类型处理时, 对于 SQL 语句查询出的字段在相应的 pojo 中必须有和它相同的字段对应, 而 resultType 中的内容就是 pojo 在本项目中的位置。

自动映射注意事项 :

1. 前提: SQL 列名和 JavaBean 的属性是一致的;
2. 使用 resultType, 如用简写需要配置 typeAliases (别名);
3. 如果列名和 JavaBean 不一致, 但列名符合单词下划线分割, Java 是驼峰命名法, 则 mapUnderscoreToCamelCase 可设置为 true;

演示代码: com.enjoylearning.mybatis.MybatisDemo. testAutoMapping

2.1.2 resultMap

resultMap 元素是 MyBatis 中最重要最强大的元素。它可以让你从 90% 的 JDBC ResultSets 数据提取代码中解放出来, 在对复杂语句进行联合映射的时候, 它很可能可以代替数千行的同等功能的代码。 ResultMap 的设计思想是, 简单的语句不需要明确的结果映射, 而复杂一点的语句只需要描述它们的关系就行了。

属性	描述
id	当前命名空间中的一个唯一标识, 用于标识一个 result map.
type	类的完全限定名, 或者一个类型别名.
autoMapping	如果设置这个属性, MyBatis 将会为这个 ResultMap 开启或者关闭自动映射。这个属性会覆盖全局的属性 autoMappingBehavior。默认值为: unset。

使用场景总结: 1. 字段有自定义的转化规则; 2. 复杂的多表查询

演示代码: com.enjoylearning.mybatis.MybatisDemo. testResultMap

2.1.3 到底应该用 resultType 还是 resultMap?

强制使用 resultMap, 不要用 resultClass 当返回参数, 即使所有类属性名与数据库字段一一对应, 也需要定义; 见《Java 开发手册 1.5》之 5.4.3;

2.2 怎么传递多个参数?

传递参数有三种方式:

方式	描述
使用 map 传递参数	可读性差，导致可维护性和可扩展性差，杜绝使用
使用注解传递参数	直观明了，当参数较少一般小于 5 个的时候，建议使用
使用 Java Bean 的方式传递参数	当参数大于 5 个的时候，建议使用

建议不要用 Map 作为 mapper 的输入和输出，不利于代码的可读性和可维护性；见《Java 开发手册 1.5》之 5.4.6；

演示代码：com.enjoylearning.mybatis.MybatisDemo. testManyParamQuery

代码是给系统运行的，但代码更是给人用的，写一行可能只要1分钟，但未来会被一代代工程师读很多次、改很多次。代码的可读性与可维护性，是我心目中的代码第一标准。

系统恒久远，代码永流传！

@鲁肃

2.3 怎么样获取主键？

2.3.1 通过 insert/update 标签相关属性

属性	描述
useGeneratedKeys	(仅对 insert 和 update 有用) 这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键(比如：像 MySQL 和 SQL Server 这样的关系数据库管理系统的自动递增字段)，默认值：false。
keyProperty	(仅对 insert 和 update 有用) 唯一标记一个属性，MyBatis 会通过 getGeneratedKeys 的返回值或者通过 insert 语句的 selectKey 子元素设置它的键值，默认：unset。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。

注意：自增长序号不是简单的行数+1，而是序号最大值+1；

示例代码：com.enjoylearning.mybatis.MybatisDemo. testInsertGenerateId1

2.3.2 通过 selectKey 元素

属性	描述
keyProperty	selectKey 语句结果应该被设置的目标属性。如果希望得到多个生成的列，也可以是逗号分隔的属性名称列表。
resultType	结果的类型。MyBatis 通常可以推算出来，但是为了更加确定写上也不会有什么问题。MyBatis 允许任何简单类型用作主键的类型，包括字符串。如果希望作用于多个生成的列，则可以使用一个包含期望属性的 Object 或一个 Map。
order	这可以被设置为 BEFORE 或 AFTER。如果设置为 BEFORE，那么它会首先选择主键，设置 keyProperty 然后执行插入语句。如果设置为 AFTER，那么先执行插入语句，然后获取主键字段；mysql 数据库自增长的方式 order 设置为 After，oracle 数据库通过 sequence 获取主键 order 设置为 Before

Oracle 通过 sequence 获取主键示例：

```
<selectKey keyProperty="id" order= " Before" resultType="int">
    select SEQ_ID.nextval from dual
</selectKey>
```

Mysql 通过自增长序号获取主键示例：

```
<selectKey keyProperty="id" order="AFTER" resultType="int">
    select LAST_INSERT_ID()
</selectKey>
```

示例代码：com.enjoylearning.mybatis.MybatisDemo.testInsertGenerateId2

2.4 SQL 元素和 SQL 的参数

SQL 元素：用来定义可重用的 SQL 代码段，可以包含在其他语句中；

SQL 参数：向 sql 语句中传递的可变参数，分为预编译#{} 和传值\${} 两种

- ✓ 预编译 #{}：将传入的数据都当成一个字符串，会对自动传入的数据加一个单引号，能够很大程度防止 sql 注入；
- ✓ 传值\${}：传入的数据直接显示生成在 sql 中，无法防止 sql 注入；适用场景：动态报表，表名、选取的列是动态的，order by 和 in 操作，可以考虑使用\$

示例代码：com.enjoylearning.mybatis.MybatisDemo.testSymbol

建议：sql.xml 配置参数使用：#{}, #param# 不要使用\${} 此种方式容易出现 SQL 注入。见《Java 开发手册 1.5》之 5.4.4；

2.5 动态 SQL

2.5.1 动态 SQL 元素

元素	作用	备注
if	判断语句	单条件分支判断
choose、when、otherwise	相当于 java 的 case when	多条件分支判断
Trim、where、set	辅助元素	用于处理 sql 拼装问题
foreach	循环语句	在 in 语句等列举条件常用，常用于实现批量操作

2.5.2 示例代码说明

示例代码	说明
com.enjoylearning.mybatis.MybatisDemo.testSelectIfOper	在 select 中使用 if 元素， where 元素可以在查询条件之前加 where 关键字，同时去掉语句的第一个 and 或 or
com.enjoylearning.mybatis.MybatisDemo.testUpdateIfOper	在 update 中使用 if 元素， set 元素可以在值设置之前加 set 关键字，同时去掉语句最有一个逗号
com.enjoylearning.mybatis.MybatisDemo.testInsertIfOper	在 insert 中使用 if 元素， trim 元素可以帮助拼装 columns 和 values
com.enjoylearning.mybatis.MybatisDemo.testForEach4In	使用 foreach 拼装 in 条件

2.5.3 通过 Mybatis 怎么样进行批量的操作

1. 通过 foreach 动态拼装 SQL 语句，参考代码见：
com.enjoylearning.mybatis.MybatisDemo.testForEach4In. testForEach4Insert
2. 使用 BATCH 类型的 excutor,参考代码块见：
 - ✓ com.enjoylearning.mybatis.JdbcDemo.updateDemo, jdbc 批处理的原理;
 - ✓ com.enjoylearning.mybatis.MybatisDemo.testForEach4Insert, 基于 Mybatis 怎么使用 Batch 类型的 excutor;

2.6 代码生成器

MyBatis Generator: MyBatis 的开发团队提供了一个很强大的代码生成器，代码包含了数据库表对应的实体类、Mapper 接口类、Mapper XML 文件等，这些代码文件中几乎包含了全部的单表操作方法，使用 MBG 可以极大程度上方便我们使用 MyBatis，还可以减少很多重复操作； MyBatis Generator 的核心就是配置文件，完整的配置文件见：

generatorConfig.xml

运行 MGB 的方式有三种，见下表：

方式	运行代码	推荐使用场景
作为 Maven Plugin 运行	mvn mybatis-generator:generate	对逆向工程定制较多，项目工程结构比较单一的情况
运行 Java 程序 使用 XML 配置文件	com.enjoylearning.mybatis.MybatisDemo.mybatisGeneratorTest	
从命令提示符 使用 XML 配置文件	java -jar mybatis-generator-core-x.x.x.jar -configfile generatorConfig.xml 具体见网盘：逆向工程	对逆向工程定制较少，项目工程结构比较复杂的情况

2.7 关联查询

2.7.1 关联查询几个需要注意的细节

1. 超过三个表禁止 join。需要 join 的字段，数据类型必须绝对一致；多表关联查询时，保证被关联的字段需要有索引；见《Java 开发手册 1.5》之 5.2.2；
2. 不得使用外键与级联，一切外键概念必须在应用层解决；见《Java 开发手册 1.5》之 5.3.6；
3. 字段允许适当冗余，以提高查询性能，但必须考虑数据一致；见《Java 开发手册 1.5》之 5.1.13；

思考问题：为什么超过三个表禁止 join？

答：大部分数据库的性能都太弱了，尤其是涉及到大数据量的多表 join 的查询，需要的对比与运算的量是会急速增长的，而数据库优化器在多表场景可能不是执行最优的计划，所以这条规范限制了 join 表的个数，还提及了 join 字段类型必须一致并有索引；那有这种约束复杂 SQL 怎么实现？考虑如下三种方式减少 join 表的关联：

1. 字段允许适当冗余，以提高查询性能，见《Java 开发手册 1.5》之 5.1.13；
2. 分两次 select，第一次 select 取得主表数据，第二次查从表数据；

3. 将热点数据存缓存，提高数据的读取效率；

关联元素：association 用于表示一对多关系，collection 用于表示一对多关系；

关联方式：

- ✓ 嵌套结果：使用嵌套结果映射来处理重复的联合结果的子集
- ✓ 嵌套查询：通过执行另外一个 SQL 映射语句来返回预期的复杂类型

2.7.2 一对多关联嵌套结果方式

association 标签 嵌套结果方式 常用属性：

- ✓ property : 对应实体类中的属性名，必填项。
- ✓ javaType : 属性对应的 Java 类型。
- ✓ resultMap : 可以直接使用现有的 resultMap，而不需要在这里配置映射关系。
- ✓ columnPrefix : 查询列的前缀，配置前缀后，在子标签配置 result 的 column 时可以省略前缀

示例代码：com.enjoylearning.mybatis.testOneToOne.

开发小技巧：

1. resultMap 可以通过使用 extends 实现继承关系，简化很多配置工作量；
2. 关联的表查询的类添加前缀是编程的好习惯；
3. 通过添加完整的命名空间，可以引用其他 xml 文件的 resultMap；

2.7.3 一对多关联嵌套查询方式

association 标签 嵌套查询方式 常用属性：

- ✓ select : 另一个映射查询的 id, MyBatis 会额外执行这个查询获取嵌套对象的结果。
- ✓ column : 列名（或别名），将主查询中列的结果作为嵌套查询的参数。
- ✓ fetchType : 数据加载方式，可选值为 lazy 和 eager，分别为延迟加载和积极加载，这个配置会覆盖全局的 lazyLoadingEnabled 配置；

示例代码：com.enjoylearning.mybatis.testOneToOne().

嵌套查询会导致“N+1 查询问题”，导致该问题产生的原因：

1. 你执行了一个单独的 SQL 语句来获取结果列表（就是“+1”）。
2. 对返回的每条记录，你执行了一个查询语句来为每个加载细节（就是“N”）。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

解决“N+1 查询问题”的办法就是开启懒加载、按需加载数据，开启懒加载配置：

在<select>节点上配置“fetchType=lazy”

在 MyBatis 核心配置文件中加入如下配置：

```
<!-- 开启懒加载，当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需  
要加载。默认：true -->  
<setting name="aggressiveLazyLoading" value="false" />
```

2.7.4 一对多关联

collection 支持的属性以及属性的作用和 association 完全相同。mybatis 会根据 id 标签，进行字段的合并，合理配置好 ID 标签可以提高处理的效率；

示例代码：com.enjoylearning.mybatis.MybatisDemo.testManyParamQuery()

开发小技巧：如果要配置一个相当复杂的映射，一定要从基础映射开始配置，每增加一些配置就进行对应的测试，在循序渐进的过程中更容易发现和解决问题。

2.7.5 多对多关联

要实现多对多的关联，需要满足如下两个条件：

1. 先决条件一：多对多需要一种中间表建立连接关系；
2. 先决条件二：多对多关系是由两个一对多关系组成的，一对多可以也可以用两种方式实现；

示例代码：com.enjoylearning.mybatis.AssociationQueryTest.testManyToMany()

2.8 缓存

MyBatis 包含一个非常强大的查询缓存特性，使用缓存可以使应用更快地获取数据，避免频繁的数据交互；

2.8.1 一级缓存

一级缓存默认会启用，想要关闭一级缓存可以在 select 标签上配置 flushCache=“true”；一级缓存存在于 **SqlSession 的生命周期** 中，在同一个 SqlSession 中查询时，MyBatis 会把执行的方法和参数通过算法生成缓存的键值，将键值和查询结果存入一个 Map 对象中。如果同一个 SqlSession 中执行的方法和参数完全一致，那么通过算法会生成相同的键值，当 Map 缓存对象中已经存在该键值时，则会返回缓存中的对象；任何的 INSERT 、 UPDATE 、 DELETE 操作都会清空一级缓存；

示例代码：com.enjoylearning.mybatis.MybatisCacheTest.Test1LevelCache()

2.8.2 二级缓存

二级缓存也叫应用缓存，存在于 **SqlSessionFactory 的生命周期** 中，可以理解为跨 sqlSession；缓存是以 namespace 为单位的，不同 namespace 下的操作互不影响。在 MyBatis 的核心配置文件中 cacheEnabled 参数是二级缓存的全局开关，默认值是 true，如果把这个参数设置为 false，即使有后面的二级缓存配置，也不会生效；

要开启二级缓存，你需要在你的 SQL Mapper 文件中添加配置：

```
<cache eviction="LRU" flushInterval="60000" size="512" readOnly="true"/>
```

这段配置的效果如下：

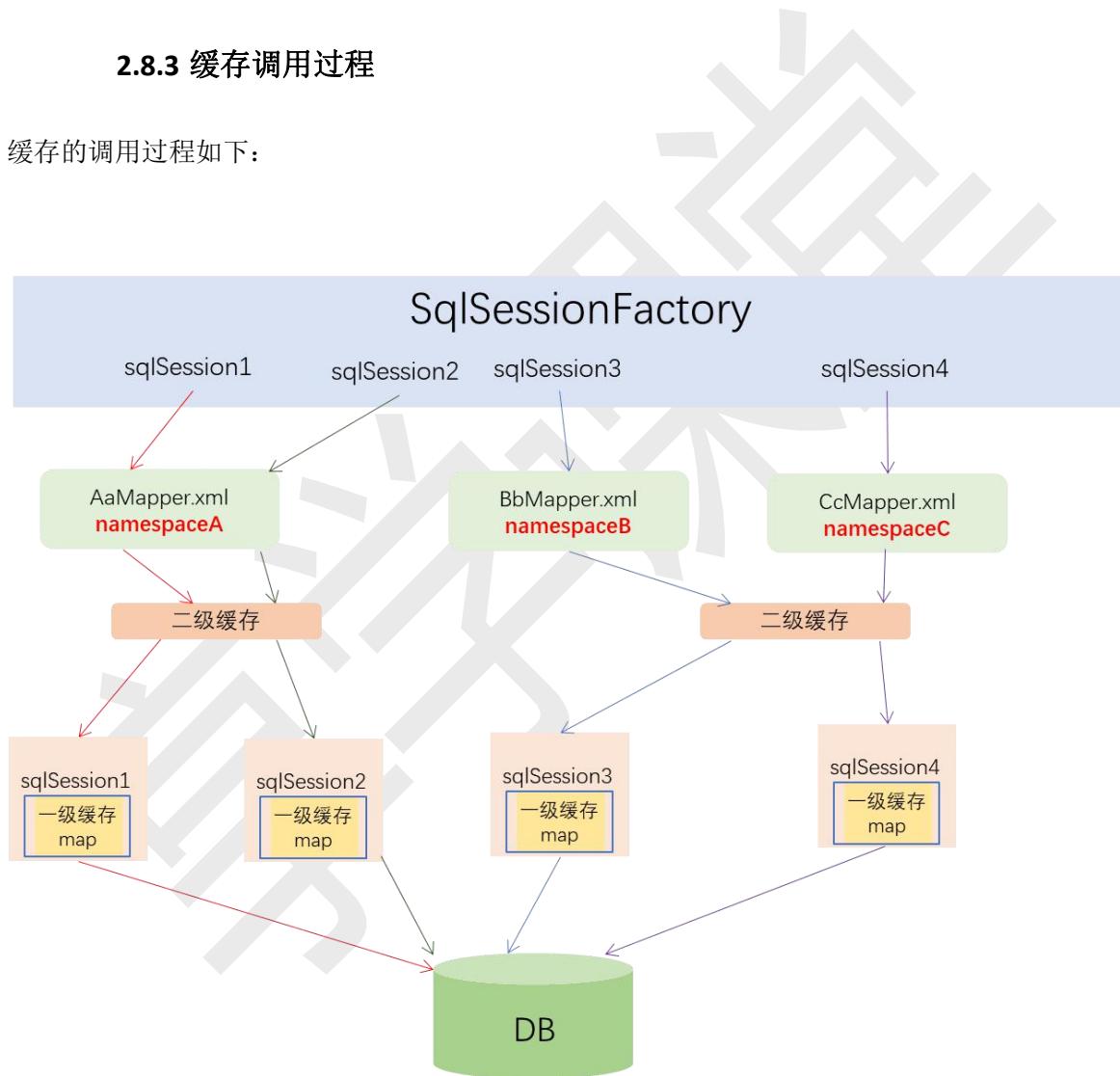
- ✓ 映射语句文件中的所有 `select` 语句将会被缓存。
- ✓ 映射语句文件中的所有 `insert,update` 和 `delete` 语句会刷新缓存。
- ✓ 缓存会使用 `Least Recently Used(LRU)`(最近最少使用的)算法来收回。
- ✓ 根据时间表(比如 `no Flush Interval`,没有刷新间隔), 缓存不会以任何时间顺序 来刷新。
- ✓ 缓存会存储列表集合或对象(无论查询方法返回什么)的 512 个引用。
- ✓ 缓存会被视为是 `read/write`(可读/可写)的缓存;

开发建议: 使用二级缓存容易出现脏读, 建议避免使用二级缓存, 在业务层使用可控制的缓存代替更好;

示例代码: `com.enjoylearning.mybatis.MybatisCacheTest.Test2LevelCache()`

2.8.3 缓存调用过程

缓存的调用过程如下:



调用过程解读:

1. 每次与数据库的连接都会优先从缓存中获取数据
2. 先查二级缓存, 再查一级缓存
3. 二级缓存以 `namespace` 为单位的, 是 `SqlSession` 共享的, 容易出现脏读, 建议避免使用二级缓存
4. 一级缓存是 `SqlSession` 独享的, 建议开启;

3 MyBatis 源码概述

3.1 怎么下载 MyBatis 源码？

MyBatis 源码下载地址: <https://github.com/MyBatis/MyBatis-3>

建议直接用网盘里的源码包，老师有在里面加注释；

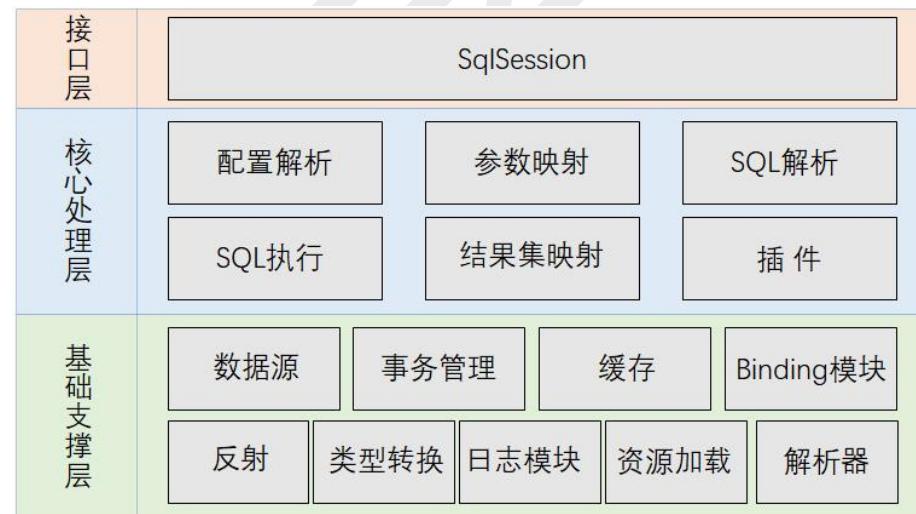
源码包导入过程：

1. 下载 MyBatis 的源码
2. 检查 maven 的版本，必须是 3.25 以上，建议使用 maven 的最新版本
3. MyBatis 的工程是 maven 工程，在开发工具中导入，工程必须使用 jdk1.8 以上版本；
4. 把 MyBatis 源码的 pom 文件中<optional>true</optional>，全部改为 false；
5. 在工程目录下执行 mvn clean install -Dmaven.test.skip=true, 将当前工程安装到本地仓库（pdf 插件报错的话，需要将这个插件屏蔽）；
注意：安装过程中可能会有很多异常信息，只要不中断运行，请耐心等待；
6. 其他工程依赖此工程

3.2 源码架构分析

源码包模块分析见脑图（双击打开）：

MyBatis 源码共 16 个模块，可以分成三层，如下图：



基础支撑层：技术组件专注于底层技术实现，通用性较强无业务含义；

核心处理器层：业务组件专注 MyBatis 的业务流程实现，依赖于基础支撑层；

接口层：MyBatis 对外提供的访问接口，面向 SqlSession 编程；

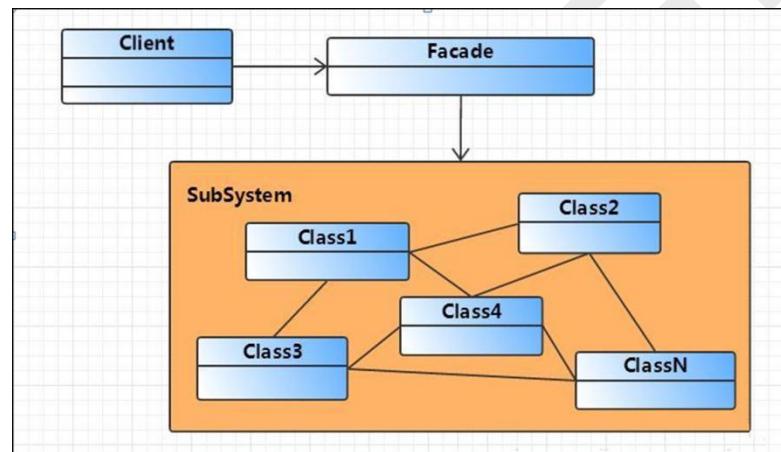
思考题：系统为什么要分层？

1. 代码和系统的可维护性更高。系统分层之后，每个层次都有自己的定位，每个层次内部

- 的组件都有自己的分工，系统就会变得很清晰，维护起来非常明确；
2. 方便开发团队分工和开发效率的提升；举个例子，mybatis 这么大的一个源码框架不可能是一个人开发的，他需要一个团队，团队之间肯定有分工，既然有了层次的划分，分工也会变得容易，开发人员可以专注于某一层的某一个模块的实现，专注力提升了，开发效率自然也会提升；
 3. 提高系统的伸缩性和性能。系统分层之后，我们只要把层次之间的调用接口明确了，那我们就可以从逻辑上的分层变成物理上的分层。当系统并发量吞吐量上来了，怎么办？为了提高系统伸缩性和性能，我们可以把不同的层部署在不同服务器集群上，不同的组件放在不同的机器上，用多台机器去抗压力，这就提高了系统的性能。压力大的时候扩展节点加机器，压力小的时候，压缩节点减机器，系统的伸缩性就是这么来的；

3.3 外观模式（门面模式）

从源码的架构分析，特别是接口层的设计，可以看出来 MyBatis 的整体架构符合门面模式的。
门面模式定义：提供了一个统一的接口，用来访问子系统中的一群接口。外观模式定义了一个高层接口，让子系统更容易使用。类图如下：



Facade 角色：提供一个外观接口，对外，它提供一个易于客户端访问的接口，对内，它可以访问子系统中的所有功能。

SubSystem（子系统）角色：子系统在整个系统中可以是一个或多个模块，每个模块都有若干类组成，这些类可能相互之间有着比较复杂的关系。

门面模式优点：使复杂子系统的接口变的简单可用，减少了客户端对子系统的依赖，达到了解耦的效果；遵循了 OO 原则中的迪米特法则，对内封装具体细节，对外只暴露必要的接口。

门面模式使用场景：

- ✓ 一个复杂的模块或子系统提供一个供外界访问的接口
- ✓ 子系统相对独立 — 外界对子系统的访问只要黑箱操作即可

3.4 面向对象设计需要遵循的六大设计原则

学习源码的目的除了学习编程的技巧、经验之外，最重要的是学习源码的设计的思想以及设计模式的灵活应用，因此在学习源码之前有必要对面向对象设计的几个原则先深入的去了解，让自己具备良好的设计思想和理念；

1. **单一职责原则：**一个类或者一个接口只负责唯一一项职责，尽量设计出功能单一的接口；
2. **依赖倒转原则：**高层模块不应该依赖低层模块具体实现，解耦高层与低层。既面向

- 接口编程，当实现发生变化时，只需提供新的实现类，不需要修改高层模块代码；
3. **开放-封闭原则：**程序对外扩展开放，对修改关闭；换句话说，当需求发生变化时，我们可以通过添加新模块来满足新需求，而不是通过修改原来的实现代码来满足新需求；
 4. **迪米特法则：**一个对象应该对其他对象保持最少的了解，尽量降低类与类之间的耦合度；实现这个原则，要注意两个点，一方面在做类结构设计的时候尽量降低成员的访问权限，能用 `private` 的尽量用 `private`；另外在类之间，如果没有必要直接调用，就不要有依赖关系；这个法则强调的还是类之间的松耦合；
 5. **里氏代换原则：**所有引用基类（父类）的地方必须能透明地使用其子类的对象；
 6. **接口隔离原则：**客户端不应该依赖它不需要的接口，一个类对另一个类的依赖应该建立在最小的接口上；
- 扩展知识：Lison 老师 2019 年 8 月 6 号的公开课《这样 Code 迅速脱单》，其中讲到的代码优化技巧归根究底就是在遵循单一职责原则和迪米特法则；

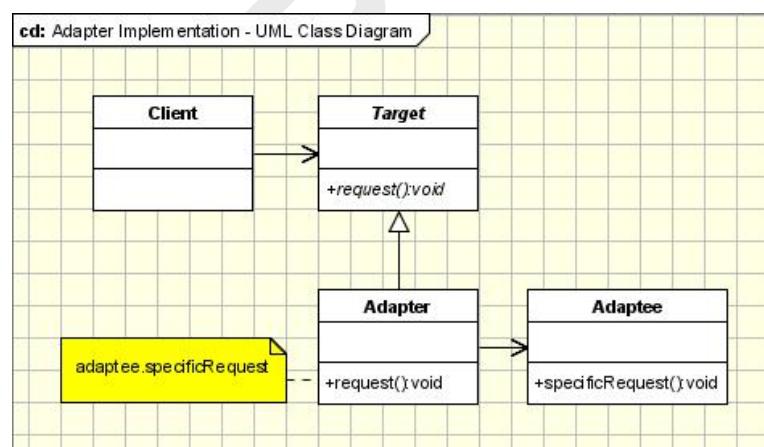
4. 日志模块分析

4.1 日志模块需求分析

1. MyBatis 没有提供日志的实现类，需要接入第三方的日志组件，但第三方日志组件都有各自的 Log 级别，且各不相同，而 MyBatis 统一提供了 `trace`、`debug`、`warn`、`error` 四个级别；
2. 自动扫描日志实现，并且第三方日志插件加载优先级如下：`slf4J → commonsLogging → Log4J2 → Log4J → JdkLog`；
3. 日志的使用要优雅的嵌入到主体功能中；

4.2 适配器模式

日志模块的第一个需求是一个典型的使用适配器模式的场景，**适配器模式** (Adapter Pattern) 是作为两个不兼容的接口之间的桥梁，将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作；类图如下：



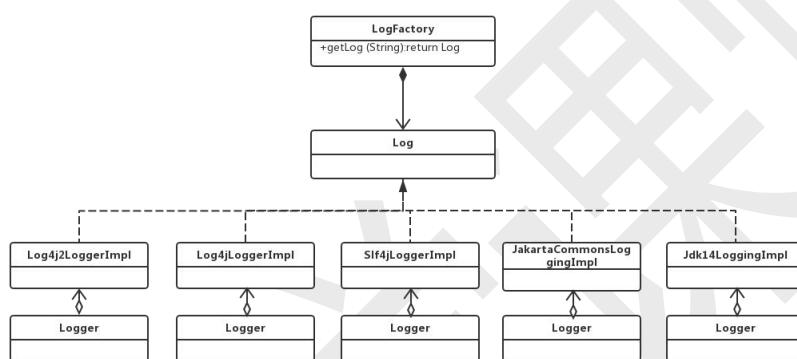
- ✓ **Target:** 目标角色，期待得到的接口。
- ✓ **Adaptee:** 适配者角色，被适配的接口。
- ✓ **Adapter:** 适配器角色，将源接口转换成目标接口。

适用场景：当调用双方都不太容易修改的时候，为了复用现有组件可以使用适配器模式；在系统中接入第三方组件的时候经常被使用到；注意：如果系统中存在过多的适配器，会增加系统的复杂性，设计人员应考虑对系统进行重构；

MyBatis 日志模块是怎么使用适配器模式？实现如下：

- ✓ **Target:** 目标角色,期待得到的接口。`org.apache.ibatis.logging.Log` 接口，对内提供了统一的日志接口；
- ✓ **Adaptee:** 适配者角色,被适配的接口。其他日志组件组件如 `slf4J`、`commonsLogging`、`Log4J2` 等被包含在适配器中。
- ✓ **Adapter:** 适配器角色,将源接口转换成目标接口。针对每个日志组件都提供了适配器，每个适配器都对特定的日志组件进行封装和转换；如 `Slf4jLoggerImpl`、`JakartaCommonsLoggingImpl` 等；

日志模块适配器结构类图：



总结：日志模块实现采用适配器模式，日志组件（Target）、适配器以及统一接口（Log 接口）定义清晰明确符合单一职责原则；同时，客户端在使用日志时，面向 Log 接口编程，不需要关心底层日志模块的实现，符合依赖倒转原则；最为重要的是，如果需要加入其他第三方日志框架，只需要扩展新的模块满足新需求，而不需要修改原有代码，这又符合了开闭原则；

4.3 怎么实现优先加载日志组件？

见 `org.apache.ibatis.logging.LogFactory` 中的静态代码块，通过静态代码块确保第三方日志插件加载优先级如下：`slf4J` → `commonsLogging` → `Log4J2` → `Log4J` → `JdkLog`；

```

public final class LogFactory {

    /**
     * Marker to be used by logging implementations that support markers
     */
    public static final String MARKER = "MYBATIS";

    //被选定的第三方日志组件适配器的构造方法
    private static Constructor<? extends Log> logConstructor;

    //自动扫描日志实现，并且第三方日志插件加载优先级如下：slf4J → commonsLogging → Log4J2 → Log4J → JdkLog
    static {
        tryImplementation(LogFactory::useSlf4jLogging);
        tryImplementation(LogFactory::useCommonsLogging);
        tryImplementation(LogFactory::useLog4J2Logging);
        tryImplementation(LogFactory::useLog4JLogging);
        tryImplementation(LogFactory::useJdkLogging);
        tryImplementation(LogFactory::useNoLogging);
    }

    private LogFactory() {
        // disable construction
    }
}

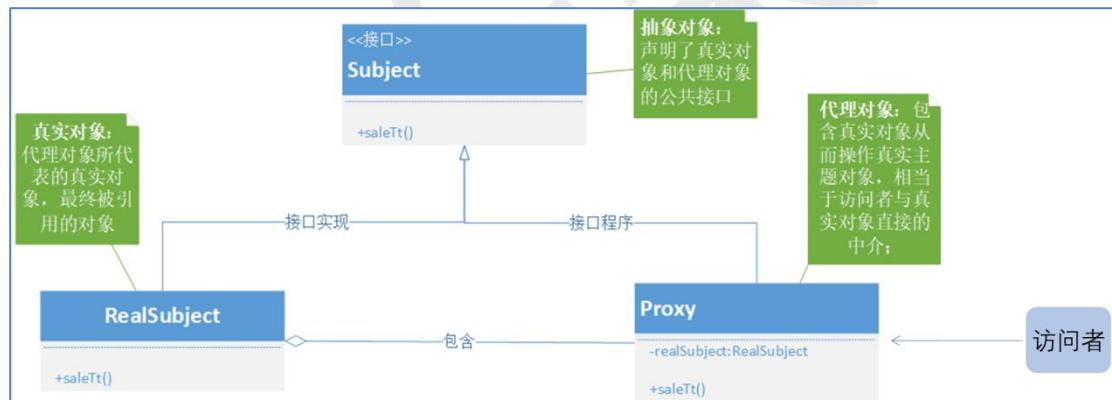
```

4.4 代理模式和动态代理

代理模式定义：给目标对象提供一个代理对象，并由代理对象控制对目标对象的引用；目的：

(1) 通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来的不必要的复杂性；(2) 通过代理对象对原有的业务增强；

代理模式类图：



代理模式有静态代理和动态代理两种实现方式。

4.4.1 静态代理

这种代理方式需要代理对象和目标对象实现一样的接口。

优点：可以在不修改目标对象的前提下扩展目标对象的功能。

缺点：

- ✓ 兀余。由于代理对象要实现与目标对象一致的接口，会产生过多的代理类。
- ✓ 不易维护。一旦接口增加方法，目标对象与代理对象都要进行修改。

4.4.2 动态代理

动态代理利用了 JDK API，动态地在内存中构建代理对象，从而实现对目标对象的代理功能。

动态代理又被称为 JDK 代理或接口代理。静态代理与动态代理的区别主要在：

1. 静态代理在编译时就已经实现，编译完成后代理类是一个实际的 class 文件
2. 动态代理是在运行时动态生成的，即编译完成后没有实际的 class 文件，而是在运行时动态生成类字节码，并加载到 JVM 中

注意：动态代理对象不需要实现接口，但是要求目标对象必须实现接口，否则不能使用动态代理。

JDK 中生成代理对象主要涉及两个类，第一个类为 `java.lang.reflect.Proxy`，通过静态方法 `newProxyInstance` 生成代理对象，第二个为 `java.lang.reflect.InvocationHandler` 接口，通过 `invoke` 方法对业务进行增强；

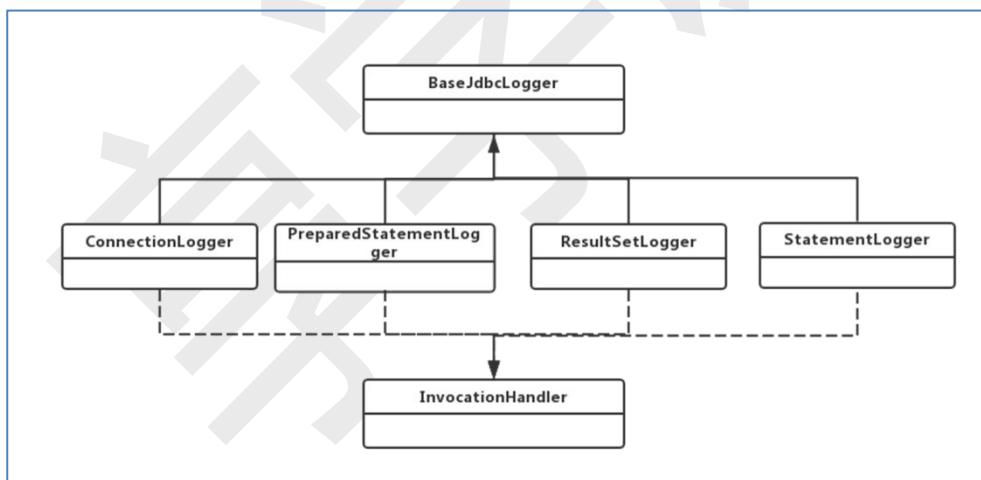
温馨提示：找班主任老师索取 lison 老师 2019 年 4 月 15 号的公开课《从动态代理来，到 Spring 源码去》，详细学习代理模式、静态代理、动态代理，以及这些技术点在 spring 事务中的运用。

4.5 优雅的增强日志功能

首先搞清楚那些地方需要打印日志？通过对日志的观察，如下几个位置需要打日志：

1. 在创建 `prepareStatement` 时，打印执行的 SQL 语句；
2. 访问数据库时，打印参数的类型和值
3. 查询出结构后，打印结果数据条数

因此在日志模块中有 `BaseJdbcLogger`、`ConnectionLogger`、`PreparedStatementLogger` 和 `ResultSetLogger` 通过动态代理负责在不同的位置打印日志；几个相关类的类图如下：



- ✓ `BaseJdbcLogger`: 所有日志增强的抽象基类，用于记录 JDBC 那些方法需要增强，保存运行期间 sql 参数信息；
- ✓ `ConnectionLogger`: 负责打印连接信息和 SQL 语句。通过动态代理，对 connection 进行增强，如果是调用 `prepareStatement`、`prepareCall`、`createStatement` 的方法，打印要执行的 sql 语句并返回 `prepareStatement` 的代理对象（`PreparedStatementLogger`），让 `prepareStatement` 也具备日志能力，打印参数；
- ✓ `PreparedStatementLogger`: 对 `prepareStatement` 对象增强，增强的点如下：
 - 增强 `PreparedStatement` 的 `setxxx` 方法将参数设置到 `columnMap`、`columnNames`、`columnValues`，为打印参数做好准备；

- 增强 PreparedStatement 的 execute 相关方法，当方法执行时，通过动态代理打印参数，返回动态代理能力的 resultSet；
- 如果是查询，增强 PreparedStatement 的 getResultSet 方法，返回动态代理能力的 resultSet；如果是更新，直接打印影响的行数
- ✓ ResultSetLogge：负责打印数据结果信息；

最后一个问题：上面讲这么多，都是日志功能的实现，那日志功能是怎么加入主体功能的？
答：既然在 Mybatis 中 Executor 才是访问数据库的组件，日志功能是在 Executor 中被嵌入的，具体代码在 org.apache.ibatis.executor.SimpleExecutor.prepareStatement(StatementHandler, Log) 方法中，如下图所示：

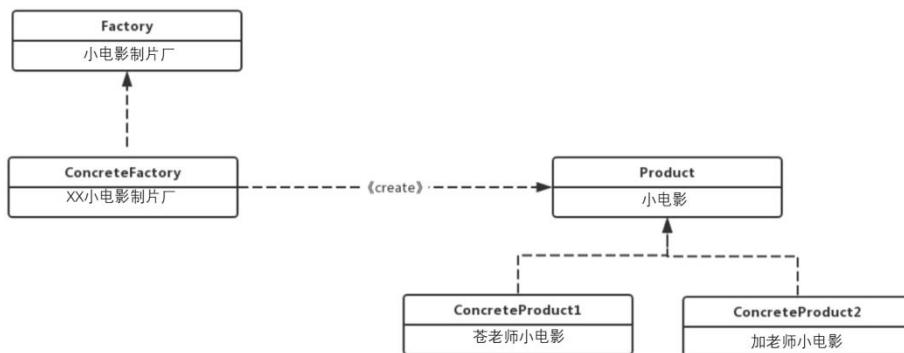
```
//创建Statement
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {
    Statement stmt;
    //获取connection对象的动态代理，添加日志能力;
    Connection connection = getConnection(statementLog); ←
    //通过不同的StatementHandler，利用connection创建（prepare）Statement
    stmt = handler.prepare(connection, transaction.getTimeout());
    //使用parameterHandler处理占位符
    handler.parameterize(stmt);
    return stmt;
}
```

5. 数据源模块分析

数据源模块重点讲解数据源的创建和数据库连接池的源码分析；数据源创建比较负责，对于复杂对象的创建，可以考虑使用工厂模式来优化，接下来介绍下简单工厂模式和工厂模式；

5.1 简单工厂模式

简单工厂属于类的创建型设计模式，通过专门定义一个类来负责创建其它类的实例，被创建的实例通常都具有共同的父类。类图如下：



- ✓ 工厂接口（Factory）：简单工厂的接口，定义了创建产品的方法，具体的工厂类必须实现这个接口；
- ✓ 工厂角色（ConcreteFactory）：这是简单工厂模式的核心，由它负责创建全部的类的内部逻辑。工厂类被外界调用，创建所必要的产品对象。

- ✓ 抽象（Product）产品角色：简单工厂模式所创建的全部对象的父类，注意，这里的父类能够是接口也能够是抽象类，它负责描写叙述全部实例所共同拥有的公共接口。
- ✓ 详细产品（Concrete Product）角色：简单工厂所创建的详细实例对象，这些详细的产品往往都拥有共同的父类。

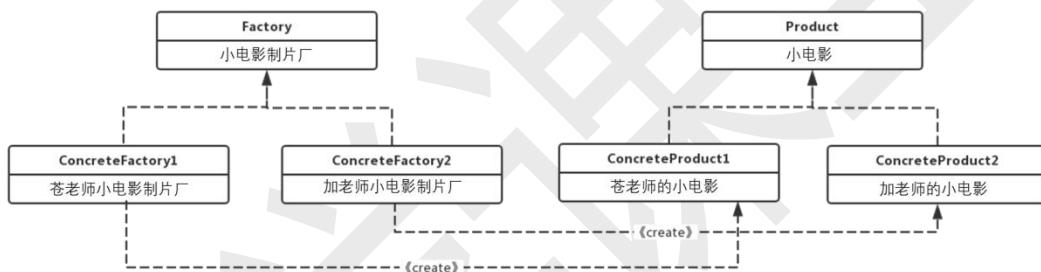
简单工厂适用场景：简单工厂模式将对象的创建和使用进行解耦，并屏蔽了创建对象可能的复杂过程，但由于创建对象的逻辑集中工厂类当中，所以简单工厂适合于产品类型不多、需求变化不频繁的场景；

简单工厂模式的缺点：工厂类负责了所有产品的实例化，违反单一职责原则，如果产品类型比较多工厂类的代码量会比较大，不利于类的可读性和扩展性；另外当有新的产品类型加入时，必须修改工厂类原有的代码，这又违反了开闭原则；

示例代码：com.enjoylearning.mybatis.factory.simple.SimpleSmallMovieFactory

5.2 工厂模式

工厂模式属于创建型模式，它提供了一种创建对象的最佳方式。定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。类图如下：



- ✓ 产品接口（Product）：产品接口用于定义产品类的功能，具体工厂类产生的所有产品都必须实现这个接口。调用者与产品接口直接交互，这是调用者最关心的接口；
- ✓ 具体产品类（ConcreteProduct）：实现产品接口的实现类，具体产品类中定义了具体的业务逻辑；
- ✓ 工厂接口（Factory）：工厂接口是工厂方法模式的核心接口，调用者会直接和工厂接口交互用于获取具体的产品实现类；
- ✓ 具体工厂类（ConcreteFactory）：是工厂接口的实现类，用于实例化产品对象，不同的具体工厂类会根据需求实例化不同的产品实现类；

为什么要使用工厂模式？

答：对象可以通过 new 关键字、反射、clone 等方式创建，也可以通过工厂模式创建。对于复杂对象，使用 new 关键字、反射、clone 等方式创建存在如下缺点：

- ✓ 对象创建和对象使用的职责耦合在一起，违反单一原则；
- ✓ 当业务扩展时，必须修改代业务代码，违反了开闭原则；

而使用工厂模式将对象的创建和使用进行解耦，并屏蔽了创建对象可能的复杂过程，相对简单工厂模式，又具备更好的扩展性和可维护性，优点具体如下：

- ✓ 把对象的创建和使用的过程分开，对象创建和对象使用使用的职责解耦；
- ✓ 如果创建对象的过程很复杂，创建过程统一到工厂里管理，既减少了重复代码，也方便以后对创建过程的修改维护；
- ✓ 当业务扩展时，只需要增加工厂子类，符合开闭原则；

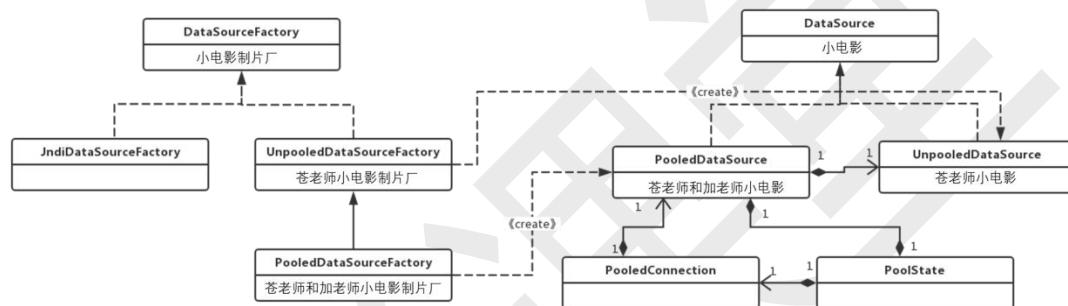
示例代码: com.enjoylearning.mybatis.factory.real.*

5.3 数据源的创建

数据源对象是比较复杂的对象，其创建过程相对比较复杂，对于 MyBatis 创建一个数据源，具体来讲有如下难点：

1. 常见的数据源组件都实现了 `javax.sql.DataSource` 接口；
2. MyBatis 不但要能集成第三方的数据源组件，自身也提供了数据源的实现；
3. 一般情况下，数据源的初始化过程参数较多，比较复杂；

综上所述，数据源的创建是一个典型使用工厂模式的场景，实现类图如下所示：



- ✓ **DataSource**: 数据源接口，JDBC 标准规范之一，定义了获取连接的方法；
- ✓ **UnPooledDataSource**: 不带连接池的数据源，获取连接的方式和手动通过 JDBC 获取连接的方式是一样的；
- ✓ **PooledDataSource**: 带连接池的数据源，提高连接资源的复用性，避免频繁创建、关闭连接资源带来的开销；
- ✓ **DataSourceFactory**: 工厂接口，定义了创建 **DataSource** 的方法；
- ✓ **UnpooledDataSourceFactory**: 工厂接口的实现类之一，用于创建 **UnpooledDataSource**(不带连接池的数据源)；
- ✓ **PooledDataSourceFactory**: 工厂接口的实现类之一，用于创建 **PooledDataSource** (带连接池的数据源)；

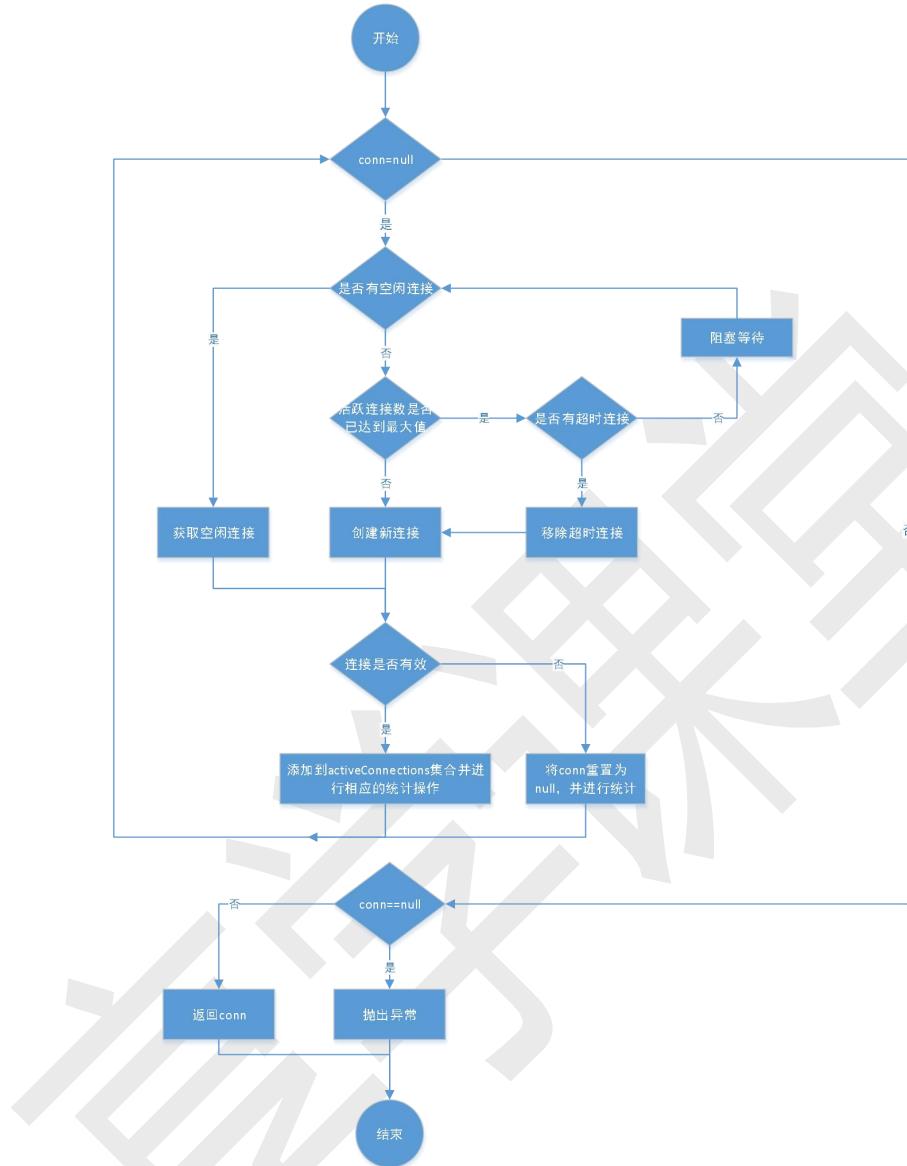
5.4 数据库连接池技术解析

数据库连接池技术是提升数据库访问效率常用的手段，使用连接池可以提高连接资源的复用性，避免频繁创建、关闭连接资源带来的开销，池化技术也是大厂高频面试题。MyBatis 内部就带了一个连接池的实现，接下来重点解析连接池技术的数据结构和算法；先重点分析下跟连接池相关的关键类：

- ✓ **PooledDataSource**: 一个简单，同步的、线程安全的数据库连接池
- ✓ **PooledConnection**: 使用动态代理封装了真正的数据库连接对象，在连接使用之前和关闭时进行增强；
- ✓ **PoolState**: 用于管理 **PooledConnection** 对象状态的组件，通过两个 `list` 分别管理空闲状态的连接资源和活跃状态的连接资源，如下图，需要注意的是这两个 `List` 使用 `ArrayList`

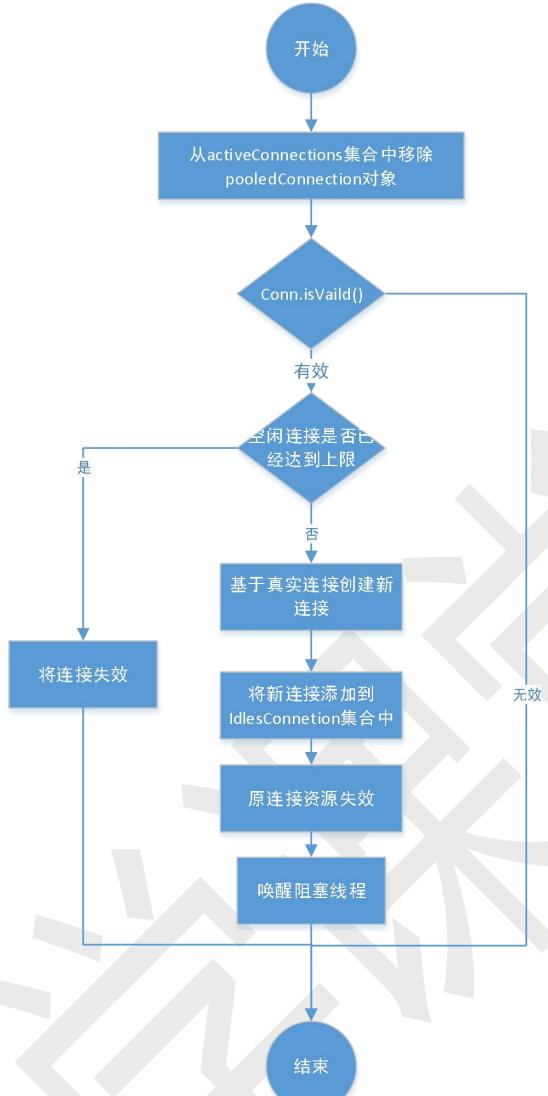
实现，存在并发安全的问题，因此在使用时，注意加上同步控制；

重点解析获取资源和回收资源的流程，获取连接资源的过程如下图：



参考代码: org.apache.ibatis.datasource.pooled.PooledDataSource.popConnection(String, String)

回收连接资源的过程如下图：



参 考 代 码 :

`org.apache.ibatis.datasource.pooled.PooledDataSource.pushConnection(PooledConnection)`

6. 缓存模块分析

6.1 需求分析

MyBatis 缓存模块需满足如下需求：

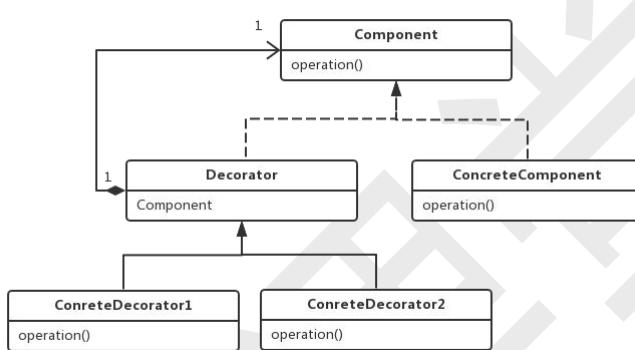
1. MyBatis 缓存的实现是基于 Map 的，从缓存里面读写数据是缓存模块的核心基础功能；
2. 除核心功能之外，有很多额外的附加功能，如：防止缓存击穿，添加缓存清空策略（fifo、lru）、序列化功能、日志能力、定时清空能力等；
3. 附加功能可以以任意的组合附加到核心基础功能之上；

基于 Map 核心缓存能力，将阻塞、清空策略、序列化、日志等等能力以任意组合的方式优雅的增强是 Mybatis 缓存模块实现最大的难题，用动态代理或者继承的方式扩展多种附加能

力的传统方式存在以下问题：这些方式是静态的，用户不能控制增加行为的方式和时机；另外，新功能的存在多种组合，使用继承可能导致大量子类存在。综上，MyBatis 缓存模块采用了装饰器模式实现了缓存模块；

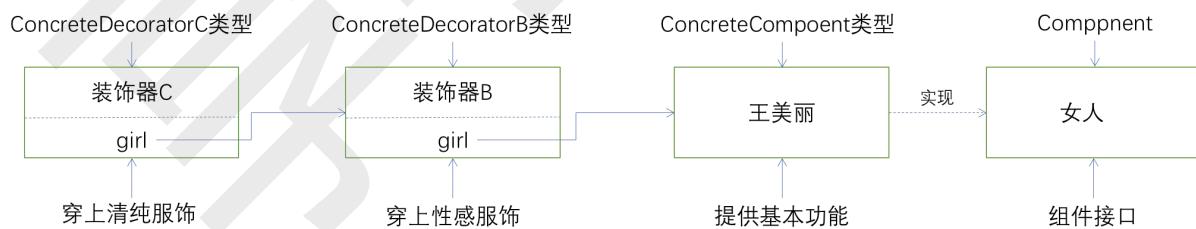
6.2 装饰器模式

装饰器模式是一种用于代替继承的技术，无需通过继承增加子类就能扩展对象的新功能。使用对象的关联关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。装饰器 UML 类图如下：



- ✓ 组件（Component）：组件接口定义了全部组件类和装饰器实现的行为；
- ✓ 组件实现类（ConcreteComponent）：实现 Component 接口，组件实现类就是被装饰器装饰的原始对象，新功能或者附加功能都是通过装饰器添加到该类的对象上的；
- ✓ 装饰器抽象类（Decorator）：实现 Component 接口的抽象类，在其中封装了一个 Component 对象，也就是被装饰的对象；
- ✓ 具体装饰器类（ConcreteDecorator）：该实现类要向被装饰的对象添加某些功能；

装饰器模式通俗易懂图示：



装饰器相对于继承，装饰器模式灵活性更强，扩展性更强：

- ✓ 灵活性：装饰器模式将功能切分成一个个独立的装饰器，在运行期可以根据需要动态的添加功能，甚至对添加的新功能进行自由的组合；
- ✓ 扩展性：当有新功能要添加的时候，只需要添加新的装饰器实现类，然后通过组合方式添加这个新装饰器，无需修改已有代码，符合开闭原则；

装饰器模式使用举例：

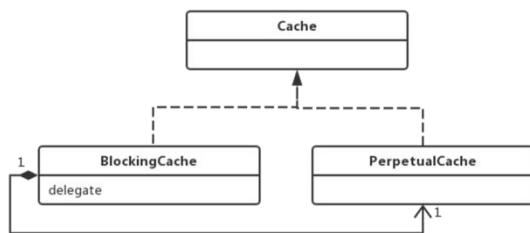
1. IO 中输入流和输出流的设计

```
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream("c:/a.txt")));
```

2. 对网络爬虫的自定义增强，可增强的功能包括：多线程能力、缓存、自动生成报表、黑白名单、random 触发等

6.3 装饰器在缓存模块的使用

MyBatis 缓存模块是一个经典的使用装饰器实现的模块，类图如下：



- ✓ **Cache:** Cache 接口是缓存模块的核心接口，定义了缓存的基本操作；
- ✓ **PerpetualCache:** 在缓存模块中扮演 ConcreteComponent 角色，使用 HashMap 来实现 cache 的相关操作；
- ✓ **BlockingCache:** 阻塞版本的缓存装饰器，保证只有一个线程到数据库去查找指定的 key 对应的数据；

BlockingCache 是阻塞版本的缓存装饰器，这个装饰器通过 ConcurrentHashMap 对锁的粒度进行了控制，提高加锁后系统代码运行的效率（注：缓存雪崩的问题可以使用细粒度锁的方式提升锁性能），源码分析见：[org.apache.ibatis.cache.decorators.BlockingCache](#)；

除了 BlockingCache 之外，缓存模块还有其他的装饰器如：

1. LoggingCache：日志能力的缓存；
2. ScheduledCache：定时清空的缓存；
3. BlockingCache：阻塞式缓存；
4. SerializedCache：序列化能力的缓存；
5. SynchronizedCache：进行同步控制的缓存；

思考题： Mybatis 的缓存功能使用 HashMap 实现会不会出现并发安全的问题？

答： MyBatis 的缓存分为一级缓存、二级缓存。二级缓存是多个会话共享的缓存，确实会出现并发安全的问题，因此 MyBatis 在初始化二级缓存时，会给二级缓存默认加上 SynchronizedCache 装饰器的增强，在对共享数据 HashMap 操作时进行同步控制，所以二级缓存不会出现并发安全问题；而一级缓存是会话独享的，不会出现多个线程同时操作缓存数据的场景，因此一级缓存也不会出现并发安全的问题；

6.4 缓存的唯一标识 CacheKey

MyBatis 中涉及到动态 SQL 的原因，缓存项的 key 不能仅仅通过一个 String 来表示，所以通过 CacheKey 来封装缓存的 Key 值，CacheKey 可以封装多个影响缓存项的因素；判断两个 CacheKey 是否相同关键是比较两个对象的 hash 值是否一致；构成 CacheKey 对象的要素包括：

1. mappedStatement 的 id
2. 指定查询结果集的范围（分页信息）
3. 查询所使用的 SQL 语句
4. 用户传递给 SQL 语句的实际参数值

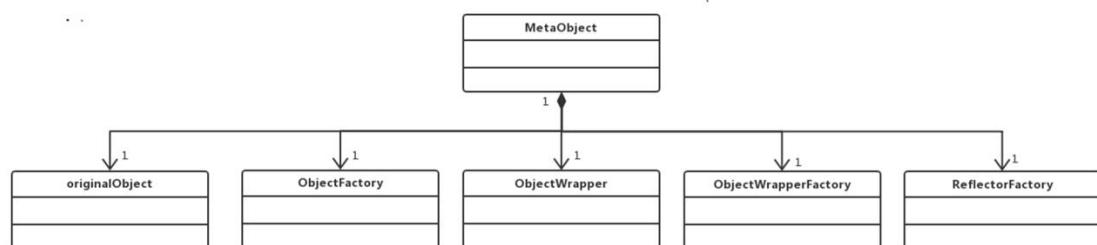
CacheKey 中 update 方法和 equals 方法是进行比较时非常重要的两个方法：

- ✓ update 方法：用于添加构成 CacheKey 对象的要素，每添加一个元素会对 hashCode、checksum、count 以及 updateList 进行更新；
- ✓ equals 方法：用于比较两个元素是否相等。首先比较 hashCode、checksum、count 是否相等，如果这三个值相等，会循环比较 updateList 中每个元素的 hashCode 是否一致；按照这种方式判断两个对象是否相等，一方面能很严格的判断是否一致避免出现误判，另外一方面能提高比较的效率；

7. 反射模块分析

反射是 Mybatis 模块中类最多的模块，通过反射实现了 POJO 对象的实例化和 POJO 的属性赋值，相对 JDK 自带的反射功能，MyBatis 的反射模块功能更为强大，性能更高；反射模块关键的几个类如下：

- ✓ ObjectFactory： MyBatis 每次创建结果对象的新实例时，它都会使用对象工厂（ObjectFactory）去构建 POJO；
- ✓ ReflectorFactory： 创建 Reflector 的工厂类，Reflector 是 MyBatis 反射模块的基础，每个 Reflector 对象都对应一个类，在其中缓存了反射操作所需要的类元信息；
- ✓ ObjectWrapper： 对对象的包装，抽象了对象的属性信息，他定义了一系列查询对象属性信息的方法，以及更新属性的方法；
- ✓ ObjectWrapperFactory： ObjectWrapper 的工厂类，用于创建 ObjectWrapper ；
- ✓ MetaObject： 封装了对象元信息，包装了 MyBatis 中五个核心的反射类。也是提供给外部使用的反射工具类，可以利用它可以读取或者修改对象的属性信息； MetaObject 的类结构如下所示：



示例代码：com.enjoylearning.mybatis.MybatisDemo.reflectionTest()

8. MyBatis 流程概述

通过对快速入门代码的分析，可以把 MyBatis 的运行流程分为三大阶段：

1. 初始化阶段：读取 XML 配置文件和注解中的配置信息，创建配置对象，并完成各个模块的初始化的工作；
2. 代理封装阶段：封装 iBatis 的编程模型，使用 mapper 接口开发的初始化工作；
3. 数据访问阶段：通过 SqlSession 完成 SQL 的解析，参数的映射、SQL 的执行、结果的解析过程；

快速入门代码阶段划分图示：

```

@Before
public void init() throws IOException {
    //-----第一阶段-----
    // 1.读取mybatis配置文件创SqlSessionFactory
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // 1.读取mybatis配置文件创SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    inputStream.close();
}

@Test
// 快速入门
public void quickStart() throws IOException {
    //-----第二阶段-----
    // 2.获取sqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession(); → 第一阶段
    // 3.获取对应mapper
    TUserMapper mapper = sqlSession.getMapper(TUserMapper.class);

    //-----第三阶段-----
    // 4.执行查询语句并返回单条数据
    TUser user = mapper.selectByPrimaryKey(2);
    System.out.println(user); → 第二阶段

    System.out.println("-----");

    // 5.执行查询语句并返回多条数据
    List<TUser> users = mapper.selectAll();
    for (TUser tUser : users) {
        System.out.println(tUser);
    }
}

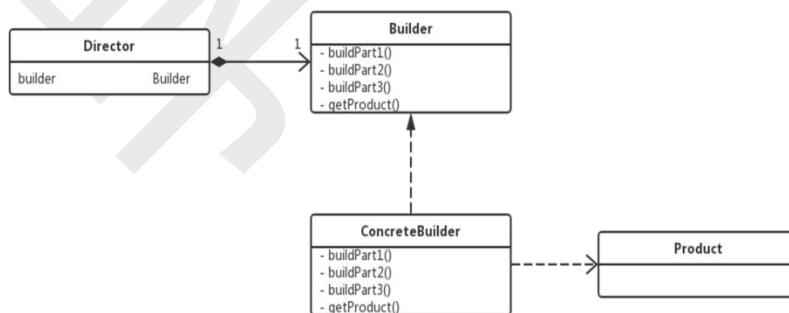
```

9. 第一阶段：配置加载阶段

9.1 建造者模式

9.1.1 什么是建造者模式

在配置加载阶段大量的使用了建造者模式，首先学习建造者模式。建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。建造者模式类图如下：



各要素如下：

- ✓ **Product:** 要创建的复杂对象
- ✓ **Builder:** 给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建；
- ✓ **ConcreteBuilder:** 实现 Builder 接口，针对不同的商业逻辑，具体化复杂对象的各部分的

-
- 创建。在建造过程完成后，提供产品的实例；
- ✓ Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建；

应用举例：红包的创建是个复杂的过程，使用建造起模式进行优化，代码示例：
com.enjoylearning.mybatis.build.Director；

关于建造器模式的扩展知识：流式编程风格越来越流行，如 zookeeper 的 Curator、JDK8 的流式编程等等都是例子。流式编程的优点在于代码编程性更高、可读性更好，缺点在于对程序员编码要求更高、不太利于调试。建造者模式是实现流式编程风格的一种方式；

9.1.2 与工厂模式区别

建造者模式应用场景如下：

- ✓ 需要生成的对象具有复杂的内部结构，实例化对象时要屏蔽掉对象内部的细节，让上层代码与复杂对象的实例化过程解耦，可以使用建造者模式；简而言之，如果“遇到多个构造器参数时要考虑用构建器”；
- ✓ 对象的实例化是依赖各个组件的产生以及装配顺序，关注的是一步一步地组装出目标对象，可以使用建造器模式；

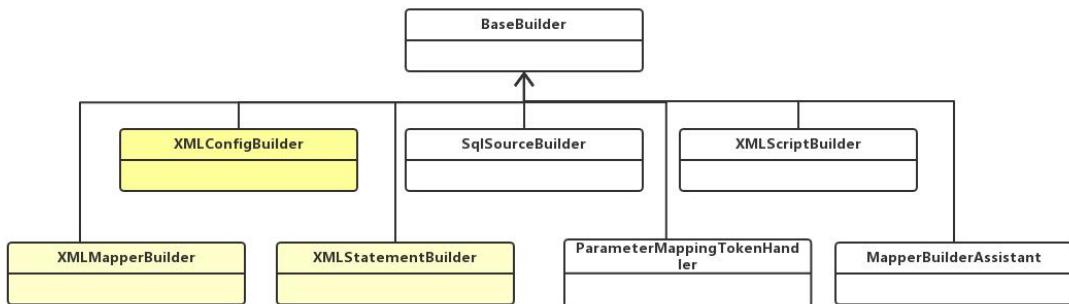
建造者模式与工程模式的区别在于：

设计模式	形象比喻	对象复杂度	客户端参与程度
工厂模式	生产大众版	关注的是一个产品整体，无须关心产品的各部分是如何创建出来的；	客户端对产品的创建过程参与度低，对象实例化时属性值相对比较固定；
建造者模式	生产定制版	建造的对象更加复杂，是一个复合产品，它由各个部件复合而成，部件不同产品对象不同，生成的产品粒度细；	客户端参与了产品的创建，决定了产品的类型和内容，参与度高；适合实例化对象时属性变化频繁的场景；

9.2 配置加载的核心类

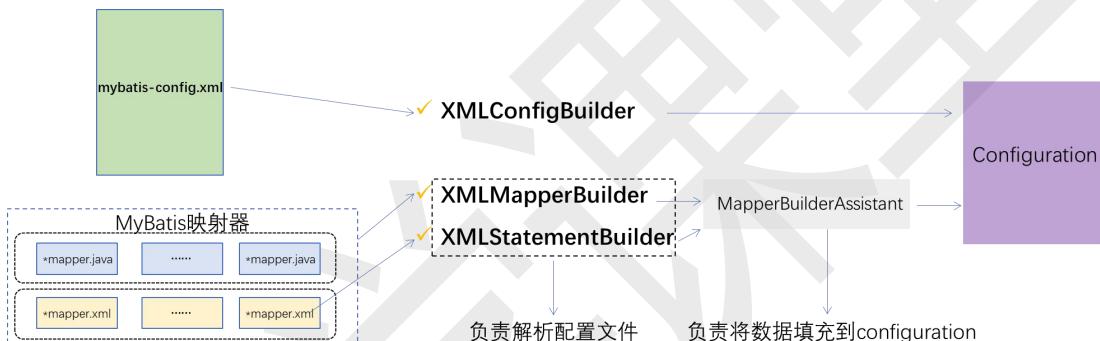
9.2.1 建造器三个核心类

在 MyBatis 中负责加载配置文件的核心类有三个，类图如下：



- ✓ **BaseBuilder:** 所有解析器的父类，包含配置文件实例，为解析文件提供的一些通用的方法；
- ✓ **XMLConfigBuilder:** 主要负责解析 mybatis-config.xml；
- ✓ **XMLMapperBuilder:** 主要负责解析映射配置 Mapper.xml 文件；
- ✓ **XMLStatementBuilder:** 主要负责解析映射配置文件中的 SQL 节点；

XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder 这三个类在配置文件加载过程中非常重要，具体分工如下图所示：



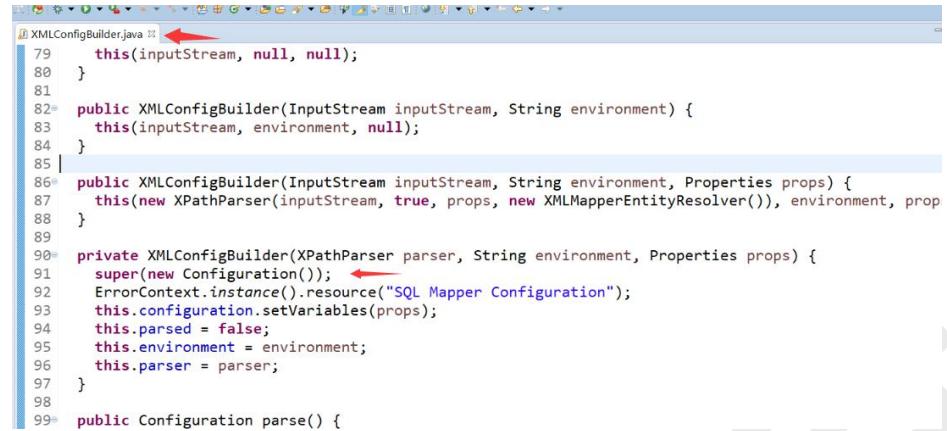
这三个类使用了建造者模式对 `Configuration` 对象进行初始化，但是没有使用建造者模式的“肉体”（流式编程风格），只用了灵魂（屏蔽复杂对象的创建过程），把建造者模式演绎成了工厂模式；后面还会对这三个类源码进行分析；

9.2.2 关于 Configuration 对象

实例化并初始化 `Configuration` 对象是第一个阶段的最终目的，所以熟悉 `Configuration` 对象是理解第一个阶段代码的核心；`Configuration` 对象的关键属性解析如下：

- ✓ **MapperRegistry:** `mapper` 接口动态代理工厂类的注册中心。在 MyBatis 中，通过 `mapperProxy` 实现 `InvocationHandler` 接口，`MapperProxyFactory` 用于生成动态代理的实例对象；
- ✓ **ResultMap:** 用于解析 `mapper.xml` 文件中的 `resultMap` 节点，使用 `ResultMapping` 来封装 `id`, `result` 等子元素；
- ✓ **MappedStatement:** 用于存储 `mapper.xml` 文件中的 `select`、`insert`、`update` 和 `delete` 节点，同时还包含了这些节点的很多重要属性；
- ✓ **SqlSource:** 用于创建 `BoundSql`，`mapper.xml` 文件中的 `sql` 语句会被解析成 `BoundSql` 对象，经过解析 `BoundSql` 包含的语句最终仅仅包含？占位符，可以直接提交给数据库执行；

需要特别注意的是 Configuration 对象在 MyBatis 中是单例的，生命周期是应用级的，换句话说只要 MyBatis 运行 Configuration 对象就会独一无二的存在；在 MyBatis 中仅在 org.apache.ibatis.builder.xml.XMLConfigBuilder.XMLConfigBuilder(XPathParser, String, Properties) 中有实例化 configuration 对象的代码，如下图：



```
XMLConfigBuilder.java
79     this(inputStream, null, null);
80 }
81
82 public XMLConfigBuilder(InputStream inputStream, String environment) {
83     this(inputStream, environment, null);
84 }
85 |
86 public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {
87     this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, prop
88 }
89
90 private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {
91     super(new Configuration()); ←
92     ErrorContext.instance().resource("SQL Mapper Configuration");
93     this.configuration.setVariables(props);
94     this.parsed = false;
95     this.environment = environment;
96     this.parser = parser;
97 }
98
99 public Configuration parse() {
```

Configuration 对象的初始化（属性复制），是在建造 SqlSessionFactory 的过程中进行的，接下来分析第一个阶段的内部流程；

9.3 配置加载过程

可以把第一个阶段配置加载过程分解为四个步骤，四个步骤如下图：



第一步：通过 SqlSessionFactoryBuilder 建造 SqlSessionFactory，并创建 XMLConfigBuilder 对象 读 取 MyBatis 核 心 配 置 文 件 ， 见 方 法： org.apache.ibatis.session.SqlSessionFactoryBuilder.build(Reader, String, Properties)，如下图：

```
2 */
3 public class SqlSessionFactoryBuilder {
4
5     public SqlSessionFactory build(Reader reader) {
6         return build(reader, null, null);
7     }
8
9     public SqlSessionFactory build(Reader reader, String environment) {
10        return build(reader, environment, null);
11    }
12
13     public SqlSessionFactory build(Reader reader, Properties properties) {
14         return build(reader, null, properties);
15     }
16
17     public SqlSessionFactory build(Reader reader, String environment, Properties properties) {
18         try {
19             //读取配置文件
20             XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);
21             return build(parser.parse()); //解析配置文件得到configuration对象，并返回SqlSessionFactory
22         } catch (Exception e) {
23             throw ExceptionFactory.wrapException("Error building SqlSession.", e);
24         } finally {
25             ErrorContext.instance().reset();
26         try {
27             reader.close();
28         } catch (IOException e) {
29             // Intentionally ignore. Prefer previous error.
30         }
31     }
32 }
```

第二步：进入 XMLConfigBuilder 的 parseConfiguration 方法，对 MyBatis 核心配置文件的各个元素进行解析，读取元素信息后填充到 configuration 对象。在 XMLConfigBuilder 的 mapperElement () 方法中通过 XMLMapperBuilder 读取所有 mapper.xml 文件；见方法：org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XNode)；见下图：

```

98  public Configuration parse() {
99=    if (parsed) { ←
100      throw new BuilderException("Each XMLConfigBuilder can only be used once.");
101    }
102    parsed = true; ←
103    parseConfiguration(parser.evalNode("/configuration"));
104    return configuration;
105  }
106
107  private void parseConfiguration(XNode root) {
108=    try {
109      //issue #117 read properties first
110      //解析<properties>节点
111      propertiesElement(root.evalNode("properties"));
112      //解析<settings>节点
113      Properties settings = settingsAsProperties(root.evalNode("settings"));
114      loadCustomVfs(settings);
115      //解析<typeAliases>节点
116      typeAliasesElement(root.evalNode("typeAliases"));
117      //解析<plugins>节点
118      pluginElement(root.evalNode("plugins"));
119      //解析<objectFactory>节点
120      objectFactoryElement(root.evalNode("objectFactory"));
121      //解析<objectWrapperFactory>节点
122      objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
123      //解析<reflectorFactory>节点
124      reflectorFactoryElement(root.evalNode("reflectorFactory"));
125      settingsElement(settings); //将settings填充到configuration
126      // read it after objectFactory and objectWrapperFactory issue #631
127      //解析<environments>节点
128      environmentsElement(root.evalNode("environments"));
129      //解析<databaseIdProvider>节点
130      databaseIdProviderElement(root.evalNode("databaseIdProvider"));
131      //解析<typeHandlers>节点
132      typeHandlerElement(root.evalNode("typeHandlers"));
133      //解析<mappers>节点
134      mapperElement(root.evalNode("mappers"));
135    } catch (Exception e) {
136      throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);
137    }
138  }
139

```

第三步： XMLMapperBuilder 的核心方法为 configurationElement (XNode)，该方法对 mapper.xml 配置文件的各个元素进行解析，读取元素信息后填充到 configuration 对象。如下图所示：

```

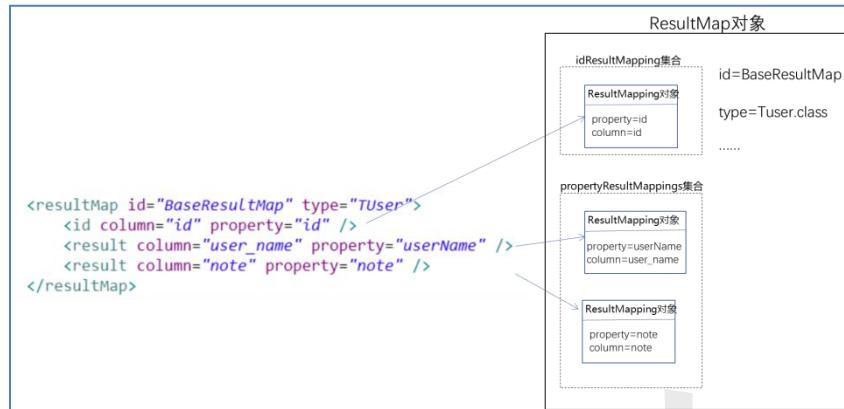
private void configurationElement(XNode context) {
  try {
    //获取mapper节点的namespace属性
    String namespace = context.getStringAttribute("namespace");
    if (namespace == null || namespace.equals("")) {
      throw new BuilderException("Mapper's namespace cannot be empty");
    }
    //设置builderAssistant的namespace属性
    builderAssistant.setCurrentNamespace(namespace);
    //解析cache-ref节点
    cacheRefElement(context.evalNode("cache-ref"));
    //重点分析： 解析cache节点-----1-----
    cacheElement(context.evalNode("cache"));
    //解析parameterMap节点（已废弃）
    parameterMapElement(context.evalNodes("/mapper/parameterMap"));
    //重点分析： 解析resultMap节点（基于数据结果去理解）-----2-----
    resultMapElements(context.evalNodes("/mapper/resultMap"));
    //解析sql节点
    sqlElement(context.evalNodes("/mapper/sql"));
    //重点分析： 解析select、insert、update、delete节点 -----3-----
    buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
  } catch (Exception e) {
    throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);
  }
}

```

在 XMLMapperBuilder 解析过程中，有四个点需要注意：

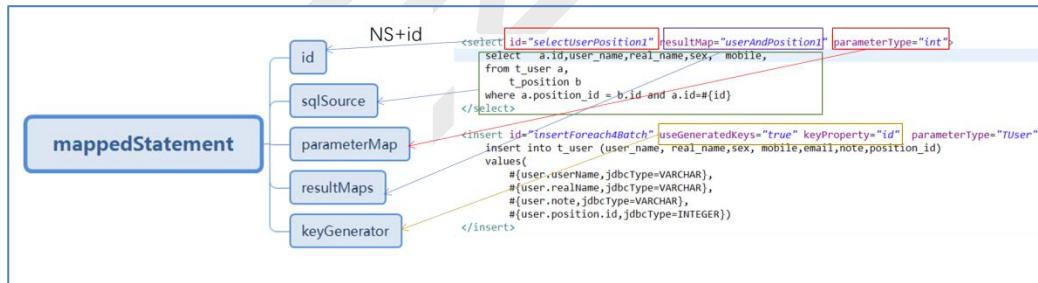
1. resultMapElements(List<XNode>)方法用于解析 resultMap 节点，这个方法非常重要，一定要跟源码理解；解析完之后数据保存在 configuration 对象的 resultMaps 属性中；

如下图所示：



2. XMLMapperBuilder 中在实例化二级缓存(见 cacheElement(XNode))、实例化 resultMap (见 resultMapElements(List<XNode>)) 过程中都使用了建造者模式，而且是建造者模式的典型应用；
3. XMLMapperBuilder 和 XMLMapperStatementBuilder 有自己的“秘书”MapperBuilderAssistant。XMLMapperBuilder 和 XMLMapperStatementBuilder 负责解析读取配置文件里面的信息，MapperBuilderAssistant 负责将信息填充到 configuration。将文件解析和数据的填充的工作分离在不同的类中，符合单一职责原则；
4. 在 buildStatementFromContext(List<XNode>) 方法中，创建 XMLStatementBuilder 解析 Mapper.xml 中 select、insert、update、delete 节点

第四步：在 XMLStatementBuilder 的 parseStatementNode()方法中，对 Mapper.xml 中 select、insert、update、delete 节点进行解析，并调用 MapperBuilderAssistant 负责将信息填充到 configuration。在理解 parseStatementNode()方法之前，有必要了解 MappedStatement，这个类用于封装 select、insert、update、delete 节点的信息；如下图所示：



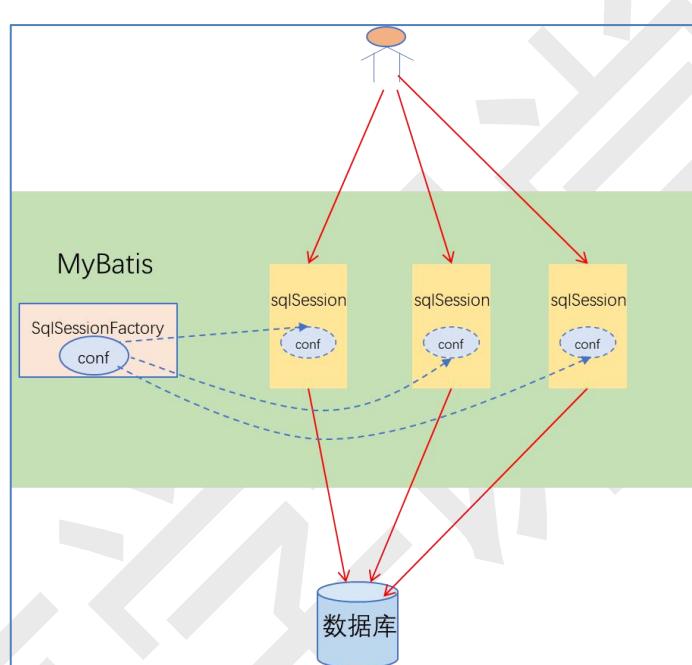
10. 第二阶段：代理封装阶段

第二个阶段是 MyBatis 最神秘的阶段，要理解它，就需要对 MyBatis 的接口层和 binding 模块数据源模块进行深入的学习；

10.1 Mybatis 的接口层

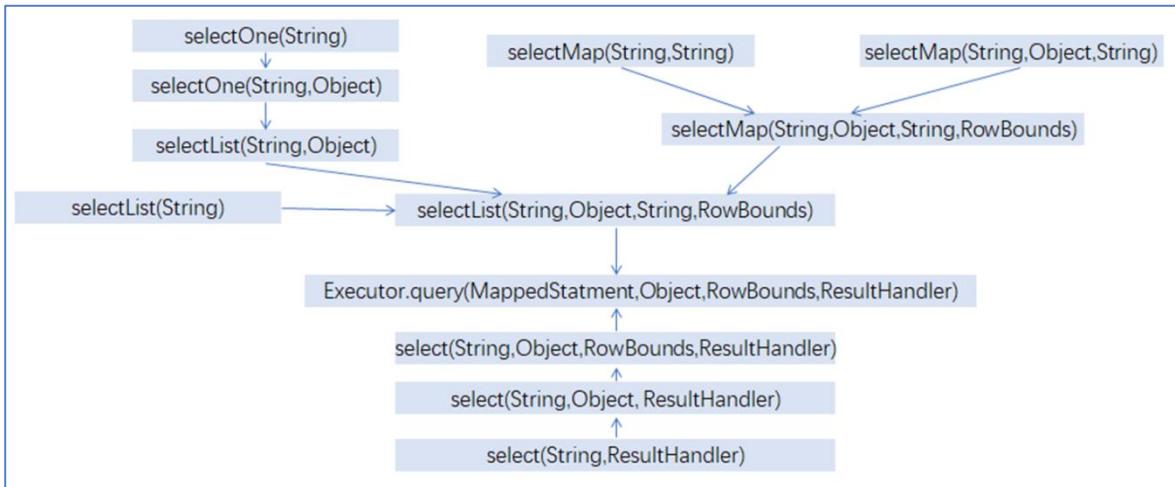
10.1.1 SqlSession

第二个阶段使用到的第一个对象就是 `SqlSession`, `SqlSession` 是 MyBatis 对外提供的最关键的核心接口, 通过它可以执行数据库读写命令、获取映射器、管理事务等; `SqlSession` 也意味着客户端与数据库的一次连接, 客户端对数据库的访问请求都是由 `SqlSession` 来处理的, `SqlSession` 由 `SqlSessionFactory` 创建, 每个 `SqlSession` 都会引用 `SqlSessionFactory` 中全局唯一单例存在的 `configuration` 对象; 如下图所示:



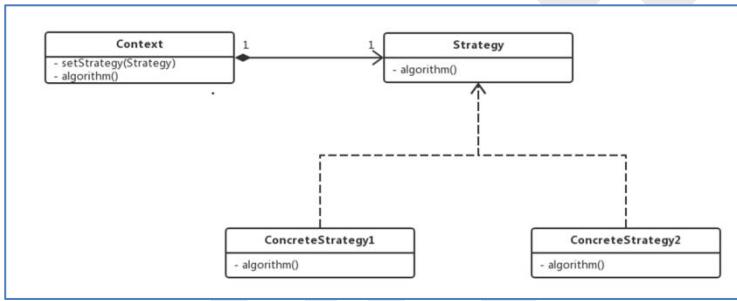
要深入理解 `SqlSession` 就得深入到源码进行学习, `SqlSession` 默认实现类为 `org.apache.ibatis.session.defaults.DefaultSqlSession`, 解读如下:

- (1) `SqlSession` 是 MyBatis 的门面, 是 MyBatis 对外提供数据访问的主要 API, 实例代码:
`com.enjoylearning.mybatis.MybatisDemo.originalOperation()`
- (2) 实际上 `SqlSession` 的功能都是基于 `Executor` 来实现的, 遵循了单一职责原则, 例如在 `SqlSession` 中的各种查询形式, 最终会把请求转发到 `Executor.query` 方法, 如下图所示:



10.1.2 策略模式

策略模式（Strategy Pattern）策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化。Spring 容器中使用配置可以灵活的替换掉接口的实现类就是策略模式最常见的应用。类图如下：



- ✓ **Context:** 算法调用者，使用 `setStrategy` 方法灵活的选择策略 (`strategy`);
- ✓ **Strategy:** 算法的统一接口；
- ✓ **ConcreteStrategy:** 算法的具体实现；

策略模式的使用场景：

- (1) 针对同一类型问题的多种处理方式，仅仅是具体行为有差别时；
- (2) 出现同一抽象类有多个子类，而又需要使用 `if-else` 或者 `switch-case` 来选择具体子类时。

10.1.3 SqlSessionFactory

`SqlSessionFactory` 使用工厂模式创建 `SqlSession`，其默认的实现类为 `DefaultSqlSessionFactory`，其中获取 `SqlSession` 的核心方法为 `openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean)`，在这个方法中从 configuration 中获取的 `TransactionFactory` 是典型的策略模式的应用。运行期，`TransactionFactory` 接口的实现，是由配置文件配置决定的，可配置选项包括：JDBC、Managed，可根据需求灵活的替换 `TransactionFactory` 的实现；配置文件截图如下：

```

<!--配置environment环境-->
<environments default="development">
    <!-- 环境配置1. 每个SqlSessionFactory对应一个环境 -->
    <environment id="development">
        <transactionManager type="JDBC" />
        <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>

```

灵活配置根据需要
替换实现

10.2 binding 模块分析

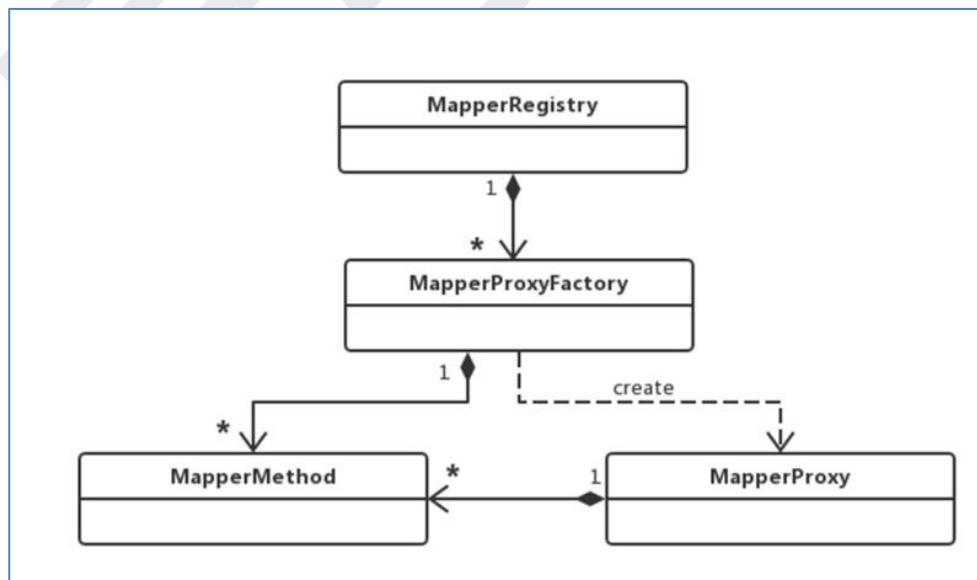
上面提到 `SqlSession` 是 MyBatis 对外提供数据库访问最主要的 API，但是因为直接使用 `SqlSession` 进行数据库开发存在代码可读性差、可维护性差的问题，所以我们很少使用，而是使用 `Mapper` 接口的方式进行数据库的开发。表面上我们在使用 `Mapper` 接口编程，实际上 MyBatis 的内部，将对 `Mapper` 接口的调用转发给了 `SqlSession`，这个请求的转发是建立在配置文件解读、动态代理增强的基础之上实现的，实现的过程有三个关键要素：

- ✓ 找到 `SqlSession` 中对应的方法执行；
- ✓ 找到命名空间和方法名（两维坐标）
- ✓ 传递参数

要实现上述的步骤，必须对 `binding` 模块有深入的分析；

10.2.1 binding 模块核心类

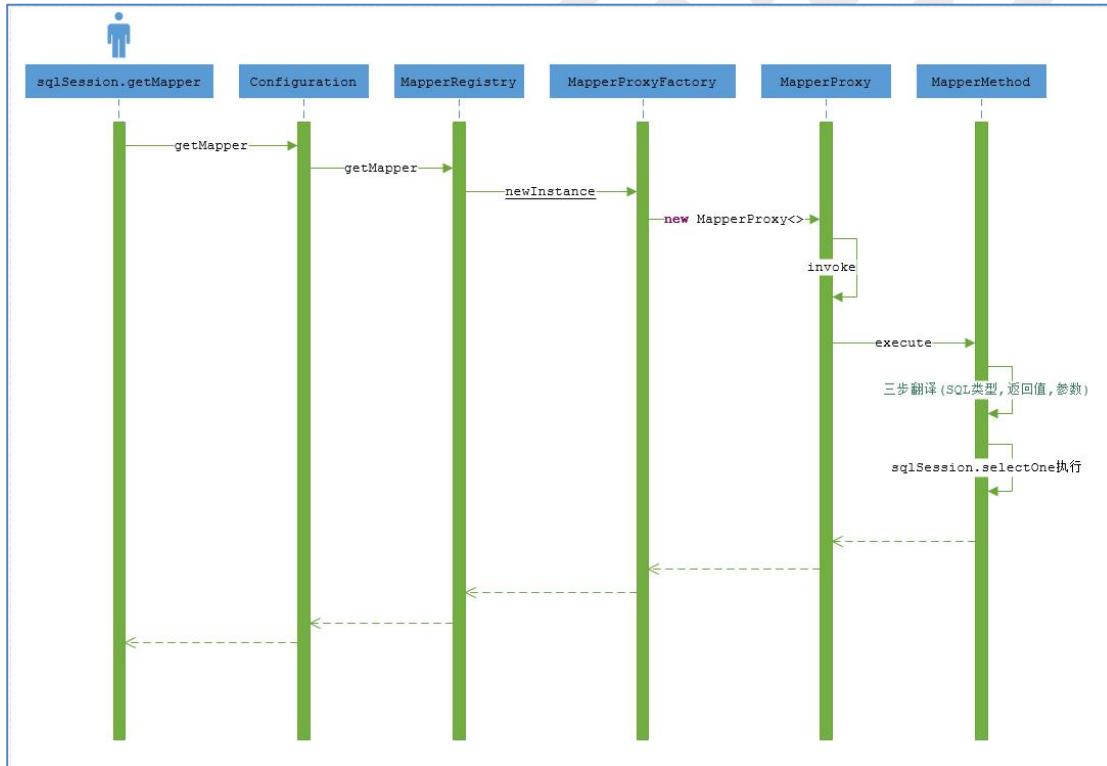
`binding` 模块核心类结构如下：



- ✓ **MapperRegistry**: mapper 接口和对应的代理对象工厂的注册中心;
- ✓ **MapperProxyFactory**: 用于生成 mapper 接口动态代理的实例对象; 保证 Mapper 实例对象是局部变量;
- ✓ **MapperProxy**: 实现了 InvocationHandler 接口, 它是增强 mapper 接口的实现;
- ✓ **MapperMethod**: 封装了 Mapper 接口中对应方法的信息, 以及对应的 sql 语句的信息; 它是 mapper 接口与映射配置文件中 sql 语句的桥梁; MapperMethod 对象不记录任何状态信息, 所以它可以在多个代理对象之间共享; MapperMethod 内几个关键数据结构:
 - **SqlCommand** : 从 configuration 中获取方法的命名空间.方法名以及 SQL 语句的类型;
 - **MethodSignature**: 封装 mapper 接口方法的相关信息 (入参, 返回类型);
 - **ParamNameResolver**: 解析 mapper 接口方法中的入参, 将多个参数转成 Map;

10.2.2 binding 模块运行流程

从 SqlSession.getMapper(Class<T>) 方法开始跟踪, 画出 binding 模块的时序图如下所示:



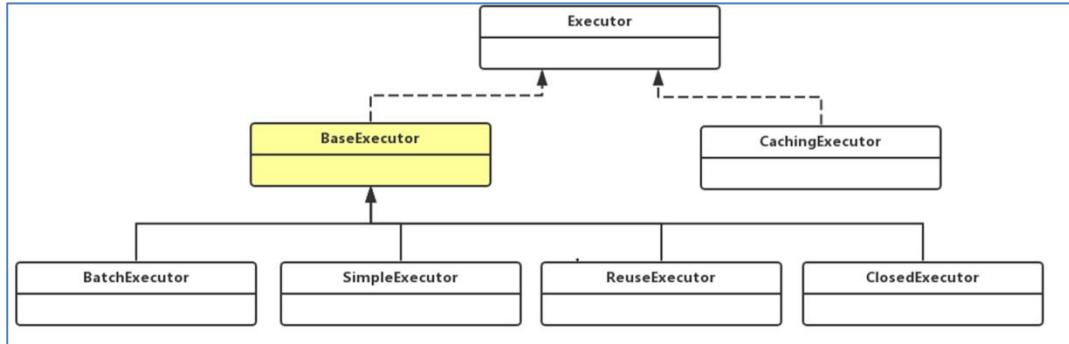
在 binding 模块的运行流程中实现三步翻译的核心方法是 `MapperMethod.execute(SqlSession, Object[])`, 翻译的过程描述如下:

- ✓ 通过 Sql 语句的类型 (`MapperMethod.SqlCommand.type`) 和 mapper 接口方法的返回参数 (`MapperMethod.MethodSignature.returnType`) 确定调用 `SqlSession` 中的某个方法;
- ✓ 通过 `MapperMethod.SqlCommand.name` 生成两维坐标;
- ✓ 通过 `MapperMethod.MethodSignature.paramNameResolve` 将传入的多个参数转成 Map 进行参数传递;

11. 第三个阶段：数据访问阶段

11.1 关于 Executor 组件

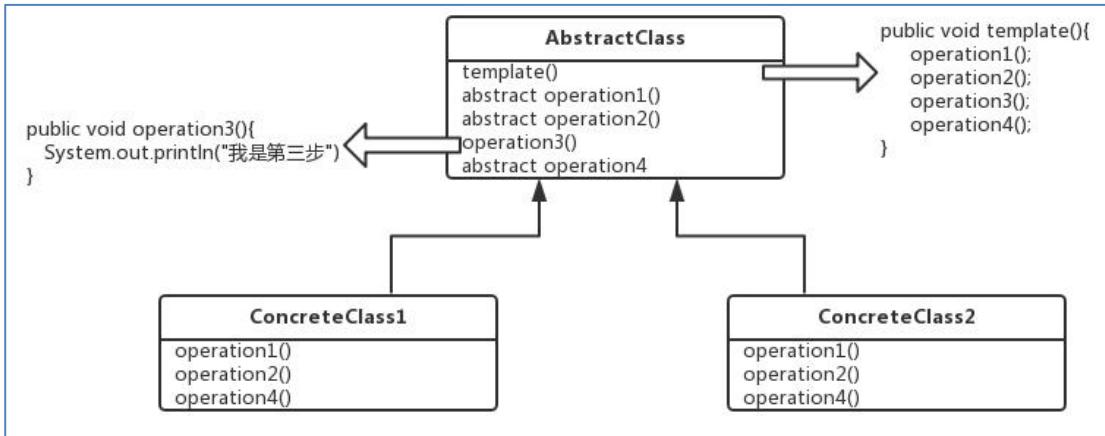
在讲第二阶段的时候，提到 SqlSession 的功能都是基于 Executor 来实现的，Executor 是 MyBatis 核心接口之一，定义了数据库操作最基本的方法，在其内部遵循 JDBC 规范完成对数据库的访问；Executor 类继承机构如下图所示：



- ✓ `Executor`: MyBatis 核心接口之一，定义了数据库操作最基本的方法；
- ✓ `CachingExecutor`: 使用装饰器模式，对真正提供数据库查询的 `Executor` 增强了二级缓存的能力；二级缓存初始化位设置：
`DefaultSqlSessionFactory.openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean);`
- ✓ `BaseExecutor`: 抽象类，实现了 `executor` 接口的大部分方法，主要提供了缓存管理和事务管理的能力，其他子类需要实现的抽象方法为：`doUpdate`,`doQuery` 等方法；
- ✓ `BatchExecutor`: 批量执行所有更新语句，基于 jdbc 的 batch 操作实现批处理；
- ✓ `SimpleExecutor`: 默认执行器，每次执行都会创建一个 statement，用完后关闭。；
- ✓ `ReuseExecutor`: 可重用执行器，将 statement 存入 map 中，操作 map 中的 statement 而不会重复创建 statement；

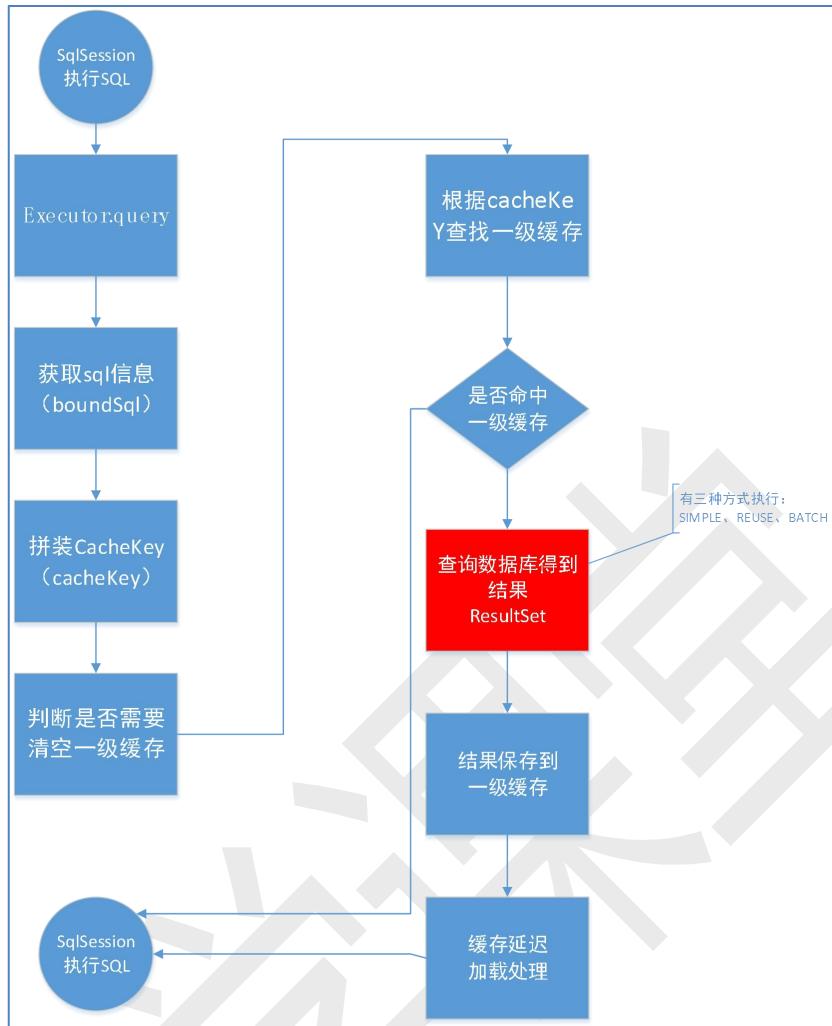
11.2 Executor 中的模板模式

模板模式：一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定实现；类结构如下：



应用场景：遇到由一系列步骤构成的过程需要执行，这个过程从高层次上看是相同的，但是有些步骤的实现可能不同，这个时候就需要考虑用模板模式了；

MyBatis 的执行器组件是使用模板模式的典型应用，其中 BaseExecutor、BaseStatementHandler 是模板模式的最佳实践；BaseExecutor 执行器抽象类，实现了 executor 接口的大部分方法，主要提供了缓存管理和事务管理的能力，其他子类需要实现的抽象方法为：doUpdate,doQuery 等方法；在 BaseExecutor 中进行一次数据库查询操作的流程如下：



如上图所示，`doQuery` 方法是查询数据的结果的子步骤，`doQuery` 方法有 SIMPLE、REUSER、BATCH 三种实现，这三种不同的实现是在子类中定义的；

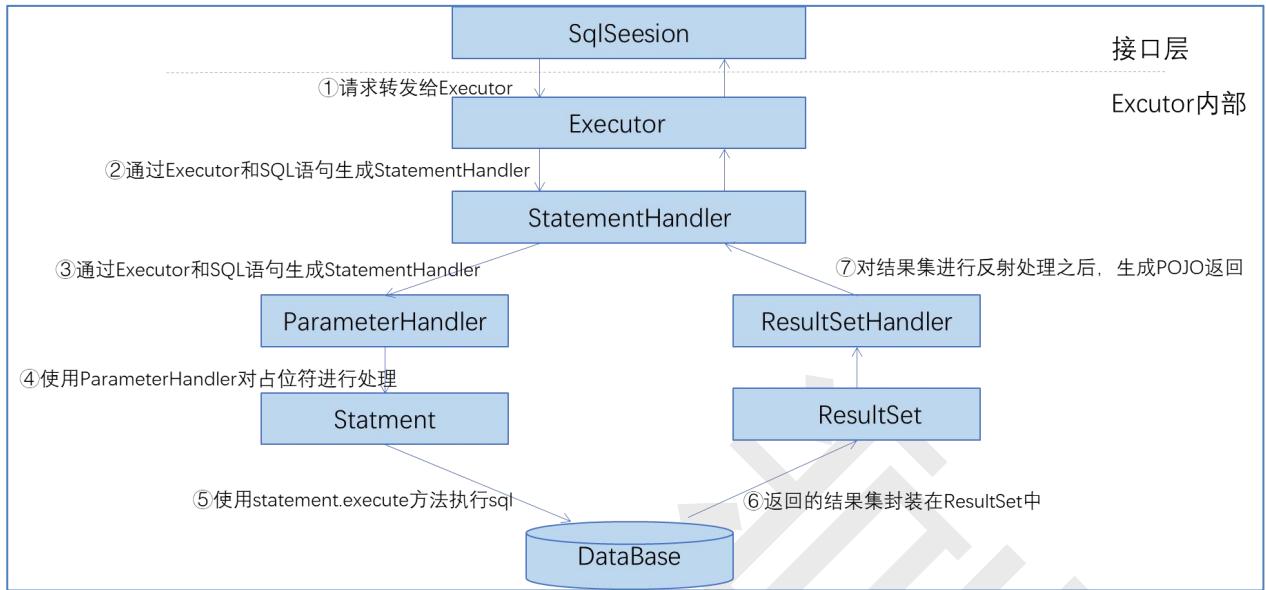
- ✓ **SimpleExecutor:** 默认配置，在 `doQuery` 方法中使用 `PreparedStatement` 对象访问数据库，每次访问都要创建新的 `PreparedStatement` 对象；
- ✓ **ReuseExecutor:** 在 `doQuery` 方法中，使用预编译 `PreparedStatement` 对象访问数据库，访问时，会重用缓存中的 `statement` 对象；
- ✓ **BatchExecutor:** 在 `doQuery` 方法中，实现批量执行多条 SQL 语句的能力；

11.3 Executor 的三个重要小弟

通过对 `SimpleExecutor doQuery()` 方法的解读发现，`Executor` 是个指挥官，它在调度三个小弟工作，这三个小弟分别为：

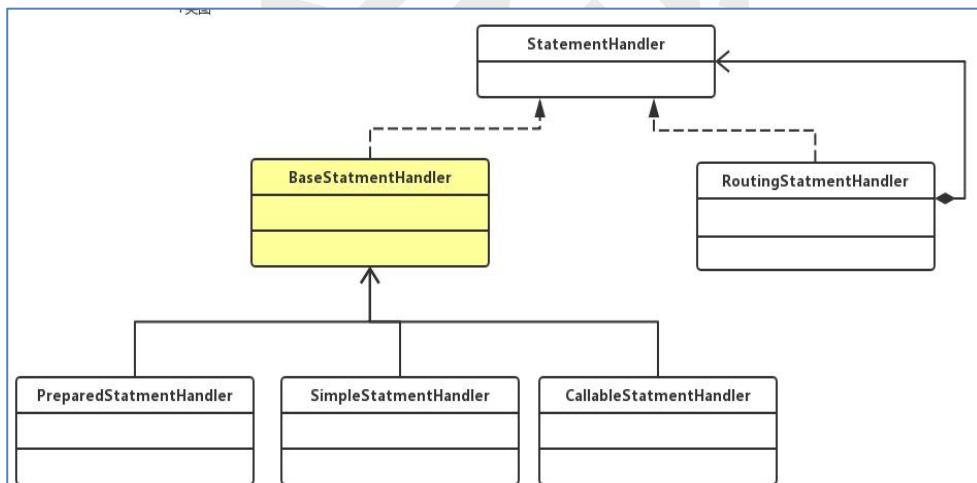
- ✓ **StatementHandler:** 它的作用是使用数据库的 `Statement` 或 `PreparedStatement` 执行操作，启承上启下作用；
- ✓ **ParameterHandler:** 对预编译的 SQL 语句进行参数设置，SQL 语句中的的占位符 “？” 都对应 `BoundSql.parameterMappings` 集合中的一个元素，在该对象中记录了对应的参数名称以及该参数的相关属性
- ✓ **ResultSetHandler:** 对数据库返回的结果集（`ResultSet`）进行封装，返回用户指定的实体类型；

Executor 三小弟内部运作流程如下图所示：



11.4 关于 StatementHandler

StatementHandler 完成 Mybatis 最核心的工作，也是 Executor 实现的基础；功能包括：创建 statement 对象，为 sql 语句绑定参数，执行增删改查等 SQL 语句、将结果映射集进行转化；StatementHandler 的类继承关系如下图所示：



- ✓ **BaseStatementHandler:** 所有子类的抽象父类，定义了初始化 statement 的操作顺序，由子类实现具体的实例化不同的 statement (模板模式);
- ✓ **RoutingStatementHandler:** Executor 组件真正实例化的子类，使用静态代理模式，根据上下文决定创建哪个具体实体类；
 - (1) **RoutingStatementHandler** 是在 Configuration 的 newStatementHandler 中创建的，见下图：

```

MybatisDemo.java  DefaultSqlSession.java  Configuration.java  MapperRegistry.java  MapperProxyFactory.java  MapperMethod.java  BaseExecutor.java  SimpleExecutor.java
97     parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);
98     return parameterHandler;
99   }
100
101  public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler parameterHandler) {
102    ResultHandler resultHandler, BoundSql boundSql;
103    ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);
104    resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);
105    return resultSetHandler;
106  }
107
108  public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
109    //根据RoutingStatementHandler最主要的功能就是根据mappedStatement的配置，生成一个对应的StatementHandler对象并赋值给delegate
110    StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);
111    statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler); //加入插件
112    return statementHandler;
113  }

```

- (2) **RoutingStatementHandler** 中使用动态代理的方式进行请求转发，在构造方法中，根据上下文（配置）决定创建具体实现类；如下图：

```

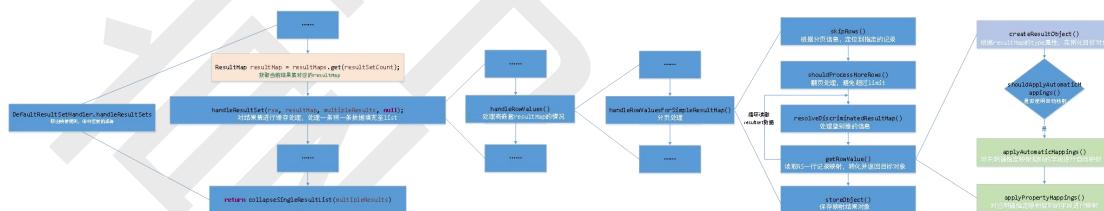
public class RoutingStatementHandler implements StatementHandler {
  private final StatementHandler delegate; //底层封装的真正的StatementHandler对象
  public RoutingStatementHandler(Executor executor, MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
    switch (ms.getStatementType()) { //根据配置决定创建具体的实现类
      case STATEMENT:
        delegate = new SimpleStatementHandler(executor, ms, parameter, rowBounds, resultHandler, boundSql);
        break;
      case PREPARED:
        delegate = new PreparedStatementHandler(executor, ms, parameter, rowBounds, resultHandler, boundSql);
        break;
      case CALLABLE:
        delegate = new CallableStatementHandler(executor, ms, parameter, rowBounds, resultHandler, boundSql);
        break;
      default:
        throw new ExecutorException("Unknown statement type: " + ms.getStatementType());
    }
  }
}

```

- ✓ **SimpleStatementHandler**：使用 statement 对象访问数据库，无须参数化；
- ✓ **PreparedStatementHandler**：使用预编译 PrepareStatement 对象访问数据库；
- ✓ **CallableStatementHandler**：调用存储过程；

11.5 关于 ResultHandler

ResultSetHandler 将从数据库查询得到的结果按照映射配置文件的映射规则，映射成相应结果集对象；在 **ResultSetHandler** 内部实际是做三个步骤：找到映射匹配规则 → 反射实例化目标对象 → 根据规则填充属性值，为完成这三部 **ResultSetHandler** 内部调用的流程图如下：



12. 与 spring 结合原理

12.1 MyBatis-Spring 是什么

Mybatis-Spring 用于帮助你将 MyBatis 代码无缝地整合到 Spring 中，集成过程中的增强主要体现在如下四个方面：

- ✓ Spring 将会加载必要的 MyBatis 工厂类和 session 类；
- ✓ 提供一个简单的方式来注入 MyBatis 数据映射器和 SqlSession 到业务层的 bean

-
- 中；
- ✓ 方便集成 spring 事务；
 - ✓ 翻译 MyBatis 的异常到 Spring 的 `DataAccessException` 异常(数据访问异常)中；
- 在使用 Mybatis-Spring 的过程中，请注意版本的兼容性，MyBatis-Spring 要求 Java5 及以上版本，另外下面列出的 MyBatis 和 Spring 版本的匹配关系：

MyBatis-Spring	MyBatis	Spring
1.0.0 或 1.0.1	3.0.1 到 3.0.5	3.0.0 或以上
1.0.2	3.0.6	3.0.0 或以上
1.1.0	3.1.0 或以上	3.0.0 或以上

12.2 MyBatis-Spring 集成配置最佳实践

MyBatis-Spring 集成配置过程如下：

- (1) 准备 spring 项目一个；
- (2) 在 pom 文件中添加 mybatis-spring 的依赖，mybatis-spring 建议使用最新版本，如下所示：

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.3-SNAPSHOT</version>
</dependency>
```

- (3) 配置 `SqlSessionFactoryBean`，在 MyBatis-Spring 中，`SqlSessionFactoryBean` 是用于创建 `SqlSessionFactory` 的，几个关键配置选项如下所示：

- ✓ `dataSource`：用于配置数据源，该属性为必选项，必须通过这个属性配置数据源，这里使用了上一节中配置好的 `dataSource` 数据库连接池。
- ✓ `mapper Locations`：配置 `SqlSessionFactoryBean` 扫描 XML 映射文件的路径，可以使用 Ant 风格的路径进行配置。
- ✓ `configLocation`：用于配置 mybatis config XML 的路径，除了数据源外，对 MyBatis 的各种配直仍然可以通过这种方式进行，并且配置 MyBatis settings 时只能使用这种方式。但配置文件中任意环境,数据源 和 MyBatis 的事务管理器都会被忽略；
- ✓ `typeAliasesPackage`：配置包中类的别名，配置后，包中的类在 XML 映射文件中使用时可以省略包名部分，直接使用类名。这个配置不支持 Ant 风格的路径，当需要配置多个包路径时可以使用分号或逗号进行分隔；

(4) 配置 MapperScannerConfigurer，通过 MapperScannerConfigurer 类自动扫描所有的 Mapper 接口，使用时可以直接注入接口。MapperScannerConfigurer 中常配置以下两个属性：

- ✓ basePackage：用于配置基本的包路径。可以使用分号或逗号作为分隔符设置多于一个的包路径，每个映射器将会在指定的包路径中递归地被搜索到。
- ✓ annotationClass：用于过滤被扫描的接口，如果设置了该属性，那么 MyBatis 的接口只有包含该注解才会被扫描进去

(5) 配置事务，让 Mybatis 集成 spring 的事务；如下图所示：

```
<!-- (事务管理)transaction manager -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
    <qualifier value="transactionManager" />
</bean>

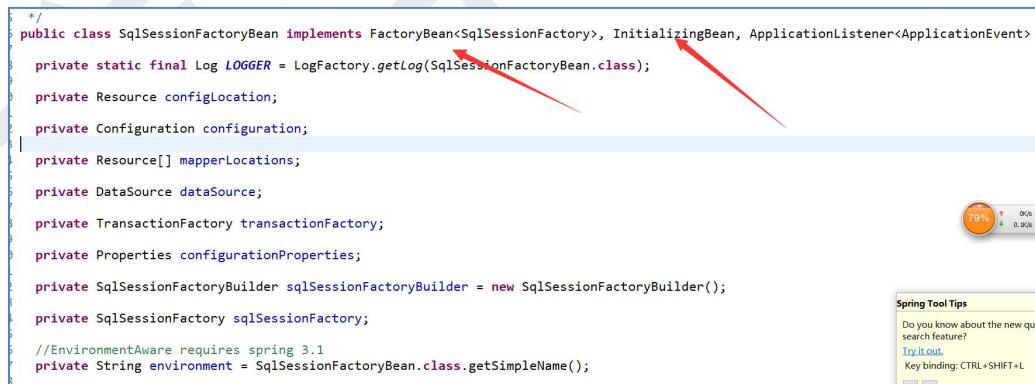
<tx:annotation-driven transaction-manager="transactionManager" />
```

12.3 MyBatis-Spring 集成原理分析

分析源码之前也需要源码下载并安装到本地仓库和开发工具中，方便给代码添加注释；安装过程和 mybatis 源码的安装过程是一样的，这里就不再重复描述了；下载地址：<https://github.com/mybatis/spring>

(1) SqlSessionFactoryBean 源码分析

SqlSessionFactoryBean 来充当 SqlSessionFactory，这里我们要搞清楚的就是为什么 SqlSessionFactoryBean 为什么能在 Spring IOC 容器中以 SqlSessionFactory 的类型保存并被获取？先来看看 SqlSessionFactoryBean 的定义是怎样的：



```
/*
 * public class SqlSessionFactoryBean implements FactoryBean<SqlSessionFactory>, InitializingBean, ApplicationListener<ApplicationEvent>
 */
private static final Log LOGGER = LogFactory.getLog(SqlSessionFactoryBean.class);
private Resource configLocation;
private Configuration configuration;
private Resource[] mapperLocations;
private DataSource dataSource;
private TransactionFactory transactionFactory;
private Properties configurationProperties;
private SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();
private SqlSessionFactory sqlSessionFactory;
//EnvironmentAware requires spring 3.1
private String environment = SqlSessionFactoryBean.class.getSimpleName();
```

首先，SqlSessionFactoryBean 实现了 InitializingBean 接口，那么容器在初始化完成 SqlSessionFactoryBean 之后必然会调用 afterPropertiesSet() 方法，如下图所示，其中调用的 buildSqlSessionFactory() 方法实际是对 MyBatis 初始化加载配置阶段的封装；



```
@Override
//在spring容器中创建全局唯一的sqlSessionFactory
public void afterPropertiesSet() throws Exception {
    notNull(dataSource, "Property 'dataSource' is required");
    notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");
    state((configuration == null && configLocation == null) || !(configuration != null && configLocation != null),
          "Property 'configuration' and 'configLocation' can not specified with together");

    this.sqlSessionFactory = buildSqlSessionFactory(); // 封装了Mybatis的初始化阶段
}
```

其次，`SqlSessionFactoryBean` 实现了 `FactoryBean` 接口，当在容器中配置 `FactoryBean` 的实现类时，并不是将该 `FactoryBean` 注入到容器，而是调用 `FactoryBean` 的 `getObject` 方法产生的实例对象注入容器，如下图所示：

```
/*
 *
 * 将SqlSessionFactory对象注入spring容器
 * {@inheritDoc}
 */
@Override
public SqlSessionFactory getObject() throws Exception {
    if (this.sqlSessionFactory == null) {
        afterPropertiesSet();
    }
    return this.sqlSessionFactory;
}
```

`SqlSessionFactoryBean` 就是将 `sqlSessionFactory` 注入容器, IOC 容器中的其他类型能拿到 `SqlSession` 实例了, 就可以进行相关的 SQL 执行任务了;

(2) MapperFactoryBean 源码分析

在之前的所有配置中都没出现过 `MapperFactoryBean`，而实际上真正帮助 Spring 生成 `Mapper` 接口实现类的恰恰就是它，来看看 `MapperFactoryBean` 的定义是怎样的：

```
2 * @see SqlSessionTemplate
3 */
4 public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements FactoryBean<T> {
5     private Class<T> mapperInterface;
6
7     private boolean addToConfig = true;
8
9     public MapperFactoryBean() {
10        //intentionally empty
11    }
12
13     public MapperFactoryBean(Class<T> mapperInterface) {
14         this.mapperInterface = mapperInterface;
15     }
16
17     /**
18      * 通过在容器中的mapperRegistry, 返回当前mapper接口的动态代理
19      *
20      * {@inheritDoc}
21      */
22     @Override
23     public T getObject() throws Exception {
24         return getSqlSession().getMapper(this.mapperInterface);
25     }
26
27     /**
28      * 封装了MyBatis的第二阶段, 每次注入容器的都是sqlsession实例化的mapper接口的实现类
29      */
30 }
```

如上图所示，MapperFactoryBean 实现了 FactoryBean 接口，getObject 方法实际是封装了 MyBatis 的第二阶段，注入容器的是 SqlSession 实例化的 Mapper 接口的实现类；

(3) MapperScannerConfigurer 源码分析

虽然 `MapperFactoryBean` 用于帮助 Spring 生成 Mapper 接口，但我们很少直接配置 `MapperFactoryBean` 而是配置 `MapperScannerConfigurer`，如下图所示：

```
<!-- spring和MyBatis完美整合, 不需要mybatis的配置映射文件 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="typeAliasesPackage" value="com.enjoylearning.mybatis.entity" />
    <property name="mapperLocations" value="classpath:sqlmapper/*.xml" />
</bean>

<!--
    <bean id="tUserMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
        <property name="mapperInterface" value="com.enjoylearning.mybatis.mapper.TUserMapper"></property>
        <property name="sqlSessionFactory" ref="sqlSessionFactory"></property>
    </bean>
-->

<!-- DAO接口所在包名, Spring会自动查找其下的类 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.enjoylearning.mybatis.mapper" />
</bean>
```

原因在于又可能工程中的 mapper 接口数量比较多，为每个 mapper 接口都配置 MapperFactoryBean，配置文件会变得非常庞大，所以才会使用 MapperScannerConfigurer 为每个 mapper 接口一对一的生成 MapperFactoryBean，那 MapperScannerConfigurer 是怎么做到的呢？先看看其源码：

```
public class MapperScannerConfigurer implements BeanDefinitionRegistryPostProcessor, InitializingBean, ApplicationContextAware, BeanNameAware, ApplicationEventMulticaster, ApplicationListener<ContextRefreshedEvent>, ApplicationEventPublisher, ApplicationEventPublisherAware {  
    private String basePackage;//用于指定要扫描的包  
  
    private boolean addToConfig = true;  
  
    private SqlSessionFactory sqlSessionFactory;  
  
    private SqlSessionTemplate sqlSessionTemplate;  
  
    private String sqlSessionFactoryBeanName;  
  
    private String sqlSessionTemplateBeanName;  
  
    private Class<? extends Annotation> annotationClass;//mapper接口上有指定的annotation才会被扫描  
  
    private Class<?> markerInterface;//mapper接口继承与指定的接口才会被扫描  
  
    private ApplicationContext applicationContext;//容器上下文  
  
    private String beanName;
```

如上图所示 `MapperScannerConfigurer` 实现了 `BeanDefinitionRegistryPostProcessor` 接口，因此可以对 Bean 的结构调整之后再注入容器。那 `MapperScannerConfigurer` 在扫描完这些 `mapper` 接口之后，主要是将 `Mapper` 接口一个个的转换成 `MapperFactoryBean` 之后注入容器的，具体转换代码见：

```
org.mybatis.spring.mapper.ClassPathMapperScanner.processBeanDefinitions(Set<BeanDefinitionHolder>)
```

代码如下图所示：

```

//处理扫描得到的BeanDefinitionHolder集合，将集合中的每一个mapper接口转换成MapperFactoryBean后，注册至spring容器
private void processBeanDefinitions(Set<BeanDefinitionHolder> beanDefinitions) {
    GenericBeanDefinition definition;
    //遍历集合
    for (BeanDefinitionHolder holder : beanDefinitions) {
        definition = (GenericBeanDefinition) holder.getBeanDefinition();

        if (logger.isDebugEnabled()) {
            logger.debug("Creating MapperFactoryBean with name '" + holder.getBeanName()
                    + "' and '" + definition.getBeanClassName() + "' mapperInterface");
        }

        // the mapper interface is the original class of the bean
        // but, the actual class of the bean is MapperFactoryBean
        //增加一个构造方法，接口类型作为构造函数的入参
        definition.getPropertyValues().add("sqlSessionFactory", new RuntimeBeanReference(this.sqlSessionFactoryBeanName));
        explicitFactoryUsed = true;
    } else if (this.sqlSessionFactory != null) {
        definition.getPropertyValues().add("sqlSessionFactory", this.sqlSessionFactory);
        explicitFactoryUsed = true;
    }
    //增加sqlSessionTemplate属性
    if (StringUtils.hasText(this.sqlSessionTemplateBeanName)) {
        if (explicitFactoryUsed) {
            logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory together. sqlSessionFactory is ignored.");
        }
        definition.getPropertyValues().add("sqlSessionTemplate", new RuntimeBeanReference(this.sqlSessionTemplateBeanName));
        explicitFactoryUsed = true;
    } else if (this.sqlSessionTemplate != null) {
        if (explicitFactoryUsed) {
            logger.warn("Cannot use both: sqlSessionTemplate and sqlSessionFactory together. sqlSessionFactory is ignored.");
        }
        definition.getPropertyValues().add("sqlSessionTemplate", this.sqlSessionTemplate);
        explicitFactoryUsed = true;
    }

    //修改自动注入的方式 byType
    if (!explicitFactoryUsed) {
        if (logger.isDebugEnabled()) {
            logger.debug("Enabling autowire by type for MapperFactoryBean with name '" + holder.getBeanName() + "'");
        }
        definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
    }
}

```

13. 插件开发

13.1 理解插件

插件是用来改变或者扩展 mybatis 的原有的功能，mybatis 的插件就是通过继承 Interceptor 拦截器实现的；注意：在没有完全理解插件之前禁止使用插件对 mybatis 进行扩展，又可能会导致严重的问题；MyBatis 中能使用插件进行拦截的接口和方法如下：

- ✓ **Executor** (update、query、flushStatement、commit、rollback、getTransaction、close、isClose)
- ✓ **StatementHandler** (prepare、paramterize、batch、update 、query)
- ✓ **ParameterHandler** (getParameterObject 、setParameters)
- ✓ **ResultSetHandler** (handleResultSets、handleCursorResultSets 、handleOutputParameters)

13.2 插件开发快速入门

为了帮助大家更好的理解插件，开发一个记录慢查询的插件。通过该插件定义一个阈值，当查询操作运行时间超过这个阈值记录日志供运维人员定位慢查询，示例代码：
com.enjoylearning.mybatis.Interceptors.ThresholdInterceptor，插件实现步骤：

(1) 实现 Interceptor 接口方法

MyBatis 插件的实现必须实现 Interceptor 接口， 该接口有如下三个方法：

- ✓ **org.apache.ibatis.plugin.Interceptor.intercept(Invocation)**: 插件对业务进行增强的核心方

法；

- ✓ org.apache.ibatis.plugin.Interceptor.plugin(Object): target 是被拦截的对象，它的作用就是给被拦截的对象生成一个代理对象；
- ✓ org.apache.ibatis.plugin.Interceptor.setProperties(Properties): 读取在 plugin 中设置的参数；

(2) 确定拦截的签名

前面说到插件能拦截的接口方法，@Intercepts 和@Signature 就是用于标识插件拦截的位置。@Intercepts 其值是一个@Signature 数组。@Intercepts 用于表明当前的对象是一个 Interceptor，而 @Signature 则表明要拦截的接口、方法以及对应的参数类型。如下图所示：

```
@Intercepts({
    @Signature(type=StatementHandler.class,method="query",args={Statement.class, ResultHandler.class})
    // @Signature(type=StatementHandler.class,method="query",args={MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class})
})
public class ThresholdInterceptor implements Interceptor {
    private long threshold;

    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        long begin = System.currentTimeMillis();
        Object ret = invocation.proceed();
        long end=System.currentTimeMillis();
        long runTime = end - begin;
        if(runTime>threshold){
            Object[] args = invocation.getArgs();
            Statement stat = (Statement) args[0];
            MetaObject metaObjectStat = SystemMetaObject.forObject(stat);
            PreparedStatementLogger statementLogger = (PreparedStatementLogger)metaObjectStat.getValue("h");
            Statement statement = statementLogger.getPreparedStatement();
            System.out.println("sql语句：" + statement.toString() + "执行时间为：" + runTime + "毫秒，已经超过阈值！");
        }
        return ret;
    }

    @Override
    public Object plugin(Object target) {           → 插件对业务进行增强的核心方法
        return Plugin.wrap(target, this);
    }

    @Override
    public void setProperties(Properties properties) { → 给被拦截的对象生成器动态规划代理阶段
        this.threshold = Long.valueOf(properties.getProperty("threshold"));
    }
}
```

(3) 在配置文件中配置插件，如下图：

```
<configuration>
    <properties resource="db.properties"/>

    <settings>
        <!-- 设置自动驼峰转换 -->
        <setting name="mapUnderscoreToCamelCase" value="true" />

        <!-- 开启懒加载 -->
        <!-- 当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每种属性将会按需要加载。默认: true -->
        <setting name="aggressiveLazyLoading" value="false" />
    </settings>

    <!-- 别名定义 -->
    <typeAliases>
        <package name="com.enjoyLearning.mybatis.entity" />
    </typeAliases>

    <plugins>
        <plugin interceptor="com.enjoyLearning.mybatis.Interceptors.ThresholdInterceptor">           → 插件定义
            <property name="threshold" value="10"/>
        </plugin>

        <plugin interceptor="com.github.pagehelper.PageInterceptor">
            <property name="pageSizeZero" value="true" />
        </plugin>
    </plugins>

```

(4) 运行测试用例，控制台打印 sql 语句运行的时间：

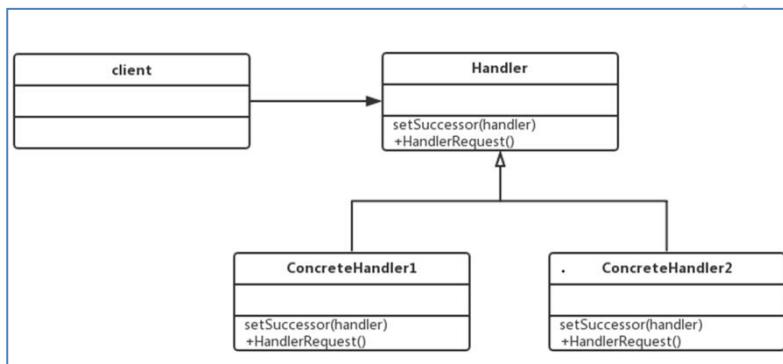
```

2019-09-23 20:36:51.254 [main] DEBUG org.apache.ibatis.transaction.jdbc.JdbcTransaction - Setting autocommit to true
2019-09-23 20:36:51.259 [main] DEBUG c.e.mybatis.mapper.TUserMapper.selectByPrimaryKey - ==> Preparing: select * from t_user where id = ?
2019-09-23 20:36:51.283 [main] DEBUG c.e.mybatis.mapper.TUserMapper.selectByPrimaryKey - ==> Parameters: 2
2019-09-23 20:36:51.308 [main] DEBUG c.e.mybatis.mapper.TUserMapper.selectByPrimaryKey - <== Total: 1
sql语句: "com.mysql.jdbc.JDBC4PreparedStatement@16267862: select
           id, userName, realName, sex, mobile, email, note
      from t_user
     where id = ?"执行时间为: 26毫秒, 已经超过阈值!
TUser [id=2, userName=james, realName=陈大雷, sex=1, mobile=18677885200, email=james@qq.com, note=james的备注
-----
```

插件打出来sql语句运行的时间

13.3 责任链模式

责任链模式:就是把一件工作分别经过链上的各个节点,让这些节点依次处理这个工作;和装饰器模式不同,每个节点都知道后继者是谁;适合为完成同一个请求需要多个处理类的场景; 责任链模式类图如下:



- ✓ **Handler:** 定义了一个处理请求的标准接口;
- ✓ **ConcreteHandler:** 具体的处理器, 处理它负责的部分, 根据业务可结束处理流程, 也可将请求转发给它的后继者;

✓ **client :** 发送者, 发起请求的客户端;

责任链模式优点:

- (1) 降低耦合度: 它将请求的发送者和接收者解耦;
- (2) 简化了对象: 使得对象不需要知道链的结构;
- (3) 增强给对象指派职责的灵活性。通过改变链内的成员或者调动它们的次序, 允许动态地新增或者删除责任;
- (4) 增加新的请求处理类很方便;

13.4 插件模块源码分析

插件模块的源码分析主要搞清楚初始化、插件加载以及插件如何调用三个问题;

(1) 插件初始化

插件的初始化实际是在 Mybatis 第一个阶段初始化的过程中加载到 Configuration 对象中的, 具体代码见: `XMLConfigBuilder.pluginElement`, 如下图所示:

```

private void pluginElement(XNode parent) throws Exception {
    if (parent != null) {
        //遍历所有的插件配置
        for (XNode child : parent.getChildren()) {
            //获取插件的类名
            String interceptor = child.getStringAttribute("interceptor");
            //获取插件的配置
            Properties properties = child.getChildrenAsProperties();
            //实例化插件对象
            Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();
            //设置插件属性
            interceptorInstance.setProperties(properties);
            //将插件添加到configuration对象，底层使用list保存所有的插件并记录顺序
            configuration.addInterceptor(interceptorInstance);
        }
    }
}

```

在 configuration 对象中，使用 interceptorChain 类属性保存所有的插件，interceptorChain 类中有个 List 用于顺序保存所有的插件；

(2) 插件的加载

为什么插件可以拦截 Executor、StatementHandler、ParameterHandler、ResultSetHandler 四个接口指定的方法呢？那是因为通过 configuration 对象创建这四大对象时，通过责任链模式按插件的顺序对四大对象进行了增强；

➤ Configuration.newExecutor

```

public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    executorType = executorType == null ? defaultExecutorType : executorType;
    executorType = executorType == null ? ExecutorType.SIMPLE : executorType;
    Executor executor;
    if (ExecutorType.BATCH == executorType) {
        executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
        executor = new ReuseExecutor(this, transaction);
    } else {
        executor = new SimpleExecutor(this, transaction);
    }
    //如果有<cache>节点，通过装饰器，添加二级缓存的能力
    if (cacheEnabled) {
        executor = new CachingExecutor(executor);
    }
    //通过interceptorChain遍历所有的插件为executor增强，添加插件的功能
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
}

```

➤ Configuration.newStatementHandler

```

public StatementHandler newStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler) {
    //通过RoutingStatementHandler对象，实际由statementType来指定真实的StatementHandler来实现
    StatementHandler statementHandler = new RoutingStatementHandler(executor, mappedStatement, parameterObject, rowBounds, resultHandler, boundSql);
    //通过interceptorChain遍历所有的插件为statementHandler增强，添加插件的功能
    statementHandler = (StatementHandler) interceptorChain.pluginAll(statementHandler);
    return statementHandler;
}

```

➤ Configuration.newParameterHandler

```

public ParameterHandler newParameterHandler(MappedStatement mappedStatement, Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler = mappedStatement.getLang().createParameterHandler(mappedStatement, parameterObject, boundSql);
    //通过interceptorChain遍历所有的插件为parameterHandler增强，添加插件的功能
    parameterHandler = (ParameterHandler) interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}

```

➤ Configuration.newResultSetHandler

```

public ResultSetHandler newResultSetHandler(Executor executor, MappedStatement mappedStatement, RowBounds rowBounds, ParameterHandler parameterHandler,
    ResultHandler resultHandler, BoundSql boundSql) {
    ResultSetHandler resultSetHandler = new DefaultResultSetHandler(executor, mappedStatement, parameterHandler, resultHandler, boundSql, rowBounds);
    //通过InterceptorChain对原有的对象为resultSetHandler增强，添加新的拦截器
    resultSetHandler = (ResultSetHandler) interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
}

```

(3) 插件的调用

插件加载是通过 ExamplePlugin.plugin(Object) 来增强的，plugin 方法内部一般使用 Plugin.wrap(target, this) 来对四大对象进行增强，Plugin.wrap 方法代码如下图所示：

```

public class Plugin implements InvocationHandler {
    //封装的真正提供服务的对象
    private final Object target;
    //自定义的拦截器
    private final Interceptor interceptor;
    //解析@Intercepts注解得到的signature信息
    private final Map<Class<?>, Set<Method>> signatureMap;

    private Plugin(Object target, Interceptor interceptor, Map<Class<?>, Set<Method>> signatureMap) {
        this.target = target;
        this.interceptor = interceptor;
        this.signatureMap = signatureMap;
    }

    //静态方法，用于帮助Interceptors生成动态代理
    public static Object wrap(Object target, Interceptor interceptor) {
        //解析Interceptors上@Intercepts注解得到的signature信息
        Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
        Class<?> type = target.getClass(); //获取目标对象的类型
        Class<?>[] interfaces = getAllInterfaces(type, signatureMap); //获取目标对象实现的接口（拦截器可以拦截四大对象实现的接口）
        if (interfaces.length > 0) {
            //使用jdk的方式创建动态代理
            return Proxy.newProxyInstance(
                type.getClassLoader(),
                interfaces,
                new Plugin(target, interceptor, signatureMap));
        }
        return target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        try {
            //获取当前接口可以被拦截的方法
            Set<Method> methods = signatureMap.get(method.getDeclaringClass());
            if (methods != null && methods.contains(method)) { //如果当前方法需要被拦截，则调用interceptor.intercept方法进行拦截处理
                return interceptor.intercept(new Invocation(target, method, args));
            }
            //如果当前方法不需要被拦截，则调用对象自身的方法
        }
    }
}

```

很明显，归根究底 MyBatis 的插件是通过动态代理对原有的对象进行增强的。

14. MyBatis 面试题集锦

1. MyBatis 的好处是什么？

答：

- 1) MyBatis 把 sql 语句从 Java 源程序中独立出来，放在单独的 XML 文件中编写，给程序的维护带来了很大便利。
- 2) MyBatis 封装了底层 JDBC API 的调用细节，并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作。
- 3) 因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。

2. 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？

答：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，称之为半自动 ORM 映射。

射工具。

3. 接口绑定有几种实现方式,分别是怎么实现的?

答: 接口绑定有两种实现方式,一种是通过注解绑定,就是在接口的方法上面加上 @Select@Update 等注解里面包含 Sql 语句来绑定,另外一种就是通过 xml 里面写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的全路径名。

4. resultType 和 resultMap 的区别?

答: resultType 和 resultMap 都是表示数据库表与 pojo 之间的映射规则的。类的名字和数据库相同时,可以直接设置 resultType 参数为 Pojo 类。若不同或者有关联查询,需要设置 resultMap 将结果名字和 Pojo 名字进行转换;从编程的最佳实践来讲,强制使用 resultMap,不要用 resultClass 当返回参数,即使所有类属性名与数据库字段一一对应,也需要定义;见《Java 开发手册 1.5》之 5.4.3;

5. 在 MyBatis 中如何在 mapper 接口中传入多个参数?

答: MyBatis 的接口默认只支持一个入参,如果要传递多个参数有三种方式:

方式	描述
使用 map 传递参数	可读性差, 导致可维护性和可扩展性差, 杜绝使用
使用注解传递参数	直观明了, 当参数较少一般小于 5 个的时候, 建议使用
使用 Java Bean 的方式传递参数	当参数大于 5 个的时候, 建议使用

建议不要用 Map 作为 mapper 的输入和输出,不利于代码的可读性和可维护性;见《Java 开发手册 1.5》之 5.4.6;

6. #{}和\${}的区别是什么?

答: 向 sql 语句中传递的可变参数,分为预编译#{ }和传值\${ }两种

- ✓ 预编译 #{}: 将传入的数据都当成一个字符串,会对自动传入的数据加一个单引号,能够很大程度防止 sql 注入;
- ✓ 传值\${ }: 传入的数据直接显示生成在 sql 中,无法防止 sql 注入;适用场景: 动态报表,表名、选取的列是动态的, order by 和 in 操作, 可以考虑使用\$

7. 通过 Mybatis 怎么样进行批量插入操作

答: 有两种方式:

1. 通过 foreach 动态拼装 SQL 语句
2. 使用 BATCH 类型的 executor;

而且这两种方式都能返回数据库主键字段;

8. Mybatis 怎么样实现关联查询

答: 有两种方式:

1. 嵌套结果:使用嵌套结果映射来处理重复的联合结果的子集。使用 join 查询,一部分

-
- 列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。
2. 嵌套查询：通过执行另外一个 SQL 映射语句来返回预期的复杂类型，在关联标签中配置 select、fetchType 这样的属性实现，是通过发送多段 SQL 实现的；

9. 什么是 N+1 问题，怎么解决？

答：嵌套查询会导致“N+1 查询问题”，导致该问题产生的原因：

1. 你执行了一个单独的 SQL 语句来获取结果列表(就是“+1”)。
2. 对返回的每条记录,你执行了一个查询语句来为每个加载细节(就是“N”)。

这个问题会导致成百上千的 SQL 语句被执行。这通常不是期望的。

解决“N+1 查询问题”的办法就是开启懒加载、按需加载数据，开启懒加载配置：

在<select>节点上配置“fetchType=lazy”，在 MyBatis 核心配置文件中加入如下配置：

```
<!-- 开启懒加载，当启用时，有延迟加载属性的对象在被调用时将会完全加载任意属性。否则，每
种属性将会按需要加载。默认: true -->
<setting name="aggressiveLazyLoading" value="false" />
```

10. 什么是一级缓存？怎么理解二级缓存？

答：一级缓存存在于 SqlSession 的生命周期中，在同一个 SqlSession 中查询时使用。二级缓存也叫应用缓存，存在于 SqlSessionFactory 的生命周期中，可以理解为跨 sqlSession；二级缓存是以 namespace 为单位的，不同 namespace 下的操作互不影响。在项目中为了避免脏读的问题，建议不适用二级缓存。

11. Mybatis 是如何进行分页的？分页插件的原理是什么？

答：

- 1) Mybatis 使用 RowBounds 对象进行分页，也可以直接编写 sql 实现分页，也可以使用 Mybatis 的分页插件。
- 2) 分页插件的原理：实现 Mybatis 提供的接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql。

举例：select * from student, 拦截 sql 后重写为：select t.* from (select * from student)t limit 0, 10

12. 简述 Mybatis 的插件运行原理，以及如何编写一个插件？

答：

- 1) Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，Mybatis 通过动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

- 2) 实现 Mybatis 的 Interceptor 接口并复写 intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

13. Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

答: 不同的 Xml 映射文件, 如果配置了 namespace, 那么 id 可以重复; 如果没有配置 namespace, 那么 id 不能重复; 毕竟 namespace 不是必须的, 只是最佳实践而已。原因就是 namespace+id 是作为 Map<String, MappedStatement> 的 key 使用的, 如果没有 namespace, 就剩下 id, 那么, id 重复会导致数据互相覆盖。有了 namespace, 自然 id 就可以重复, namespace 不同, namespace+id 自然也就不同。

14. Mybatis 都有哪些 Executor 执行器? 它们之间的区别是什么?

答: Mybatis 有三种基本的 Executor 执行器, SimpleExecutor、ReuseExecutor、BatchExecutor。1) SimpleExecutor: Mybatis 的默认执行器, 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。2) 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 Map3) BatchExecutor: 完成批处理。

15. 使用 MyBatis 的 mapper 接口调用时有哪些要求?

答:

- 1) Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同;
- 2) Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同;
- 3) Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同;
- 4) Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

16. 为什么使用 mapper 接口就能对数据库进行访问?

答: 表面上我们在使用 mapper 接口访问数据库, 实际 MyBatis 通过动态代理生成了 mapper 接口的实现类, 通过这个动态生成的实现类, 将数据库的访问请求转发给 SqlSession。转发过程中需要实现三个翻译:

- 1) 通过 sql 语句的类型和接口方法的返回参数确定调用 SqlSession 的哪个方法;
- 2) 将 sql 语句在 MyBatis 中的两维坐标作为第一个参数, 传入 SqlSession 的方法;
- 3) 将接口传入的参数封装成 map 后作为第二个参数, 传入 SqlSession 的方法;

请求转发给 SqlSession 后实现对数据库的访问操作;