尚硅谷
www.atguigu.com

MyBatis 3

内容概要

一、MyBatis简介

- MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。
- MyBatis 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。
- MyBatis可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录.

MyBatis历史

- 原是Apache的一个开源项目iBatis, 2010年6月这个项目由Apache Software Foundation 迁移到了Google Code，随着开发团队转投Google Code旗下， iBatis3.x正式更名为MyBatis ，代码于2013年11月迁移到Github（下载地址见后）。

- iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。 iBatis提供的持久层框架包括SQL Maps和Data Access Objects（DAO）

为什么要使用MyBatis？

- MyBatis是一个半自动化的持久化层框架。
- JDBC
  - SQL夹在Java代码块里，耦合度高导致硬编码内伤
  - 维护不易且实际开发需求中sql是有变化，频繁修改的情况多见
- Hibernate和JPA
  - 长难复杂SQL，对于Hibernate而言处理也不容易
  - 内部自动生产的SQL，不容易做特殊优化。
  - 基于全映射的全自动框架，大量字段的POJO进行部分映射时比较困难。导致数据库性能下降。
- 对开发人员而言，核心sql还是需要自己优化
- **sql和java编码分开，功能边界清晰，一个专注业务、一个专注数据。**

去哪里找MyBatis？

- https://github.com/mybatis/mybatis-3/

//github.com/mybatis/mybatis-3/

Personal Open source Business Explore Pricing Blog Support This repository Search Sign in Sign up

mybatis / mybatis-3 Watch 711 Star 3,929 Fork 2,930

Code Issues 76 Pull requests 11 Projects 0 Wiki Pulse Graphs

MyBatis SQL mapper framework for Java http://mybatis.github.io/mybatis-3/ 官方文档地址

2,101 commits 5 branches 22 releases 79 contributors Apache-2.0

Branch: master New pull request Find file Clone or download

kazuki43zoo Add tests for auto-detecting mybatis-typehandlers-jsr310 ... Latest commit 8fc78a7 4 days ago

src	Add tests for auto-detecting mybatis-typehandlers-jsr310	4 days ago
travis	Fix schema definition on maven settings.xml	3 months ago

Essentials

- See the docs
- Download Latest
- Download Snapshot

mybatis-3.4.1

harawata released this on 26 Jun 2016 · **186 commits** to master since this release

This release includes four user visible enhancements

and six bug fixes.

Here is the complete list of changes.

Downloads

mybatis-3.4.1.zip	5.89 MB
Source code (zip)	
Source code (tar.gz)	

二、MyBatis-HelloWorld

- HelloWorld简单版
  - 创建一张测试表
  - 创建对应的javaBean
  - 创建mybatis配置文件，sql映射文件
  - 测试

MyBatis操作数据库

- **1、创建MyBatis全局配置文件**
  - MyBatis 的全局配置文件包含了影响 MyBatis 行为甚深的设置（settings）和属性（properties）信息、如数据库连接池信息等。指导着MyBatis进行工作。我们可以参照官方文件的配置示例。
- **2、创建SQL映射文件**
  - 映射文件的作用就相当于是定义Dao接口的实现类如何工作。这也是我们使用MyBatis时编写的最多的文件。

测试

- 1、根据全局配置文件，利用SqlSessionFactoryBuilder创建SqlSessionFactory

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

- 2、使用SqlSessionFactory获取sqlSession对象。一个SqlSession对象代表和数据库的一次会话。

```
SqlSession openSession = factory.openSession();
```

使用SqlSession根据方法id进行操作

```
SqlSession openSession = factory.openSession();
try{
    Object one = openSession.selectOne("com.atguigu.dao.EmployeeMapper.getEmpById", 1);
    System.out.println(one);

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

HelloWorld-接口式编程

- 创建一个Dao接口
- 修改Mapper文件
- 测试

使用SqlSession获取映射器进行操作

```
SqlSession sqlSession = factory.openSession();
try{
    EmployeeMapper mapper = sqlSession.getMapper(EmployeeMapper.class);
    Employee employee = mapper.getEmpById(1);
    System.out.println(employee);
}finally{
    sqlSession.close();
}
```

SqlSession

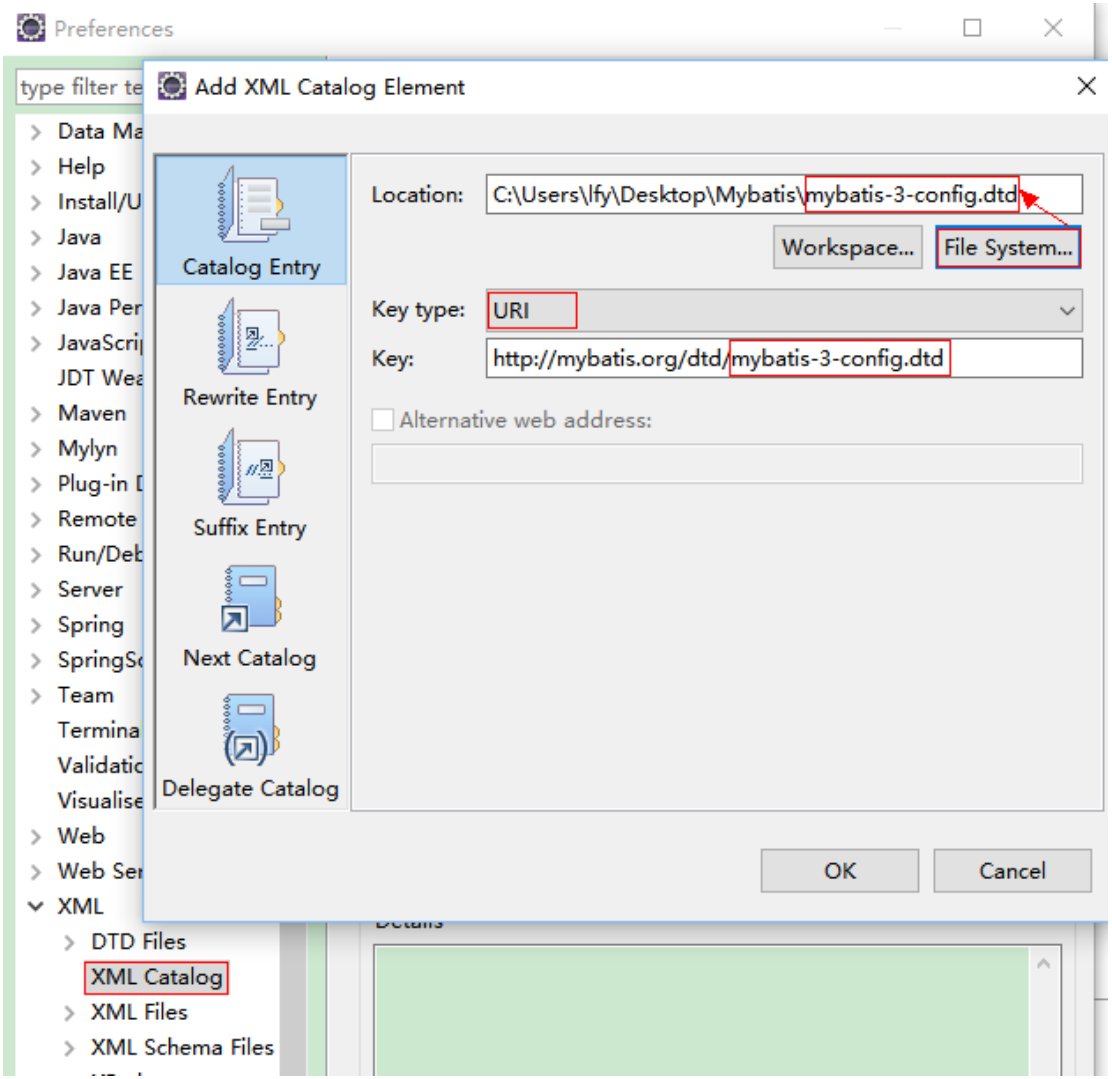
- SqlSession 的实例**不是线程安全**的，因此是不能被共享的。
- SqlSession每次**使用完成后需要正确关闭**，这个关闭操作是必须的
- SqlSession可以直接调用方法的id进行数据库操作，但是我们一般还是推荐使用SqlSession获取到Dao接口的代理类，执行代理对象的方法，可以更安全的进行类型检查操作

三、MyBatis-全局配置文件

- MyBatis 的配置文件包含了影响 MyBatis 行为甚深的设置（settings）和属性（properties）信息。文档的顶层结构如下：

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型命名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

在Eclipse中引入XML的dtd约束文件，方便编写XML的时候有提示



properties属性

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?useUnicode=true&characterEncoding=utf8
username=root
password=123456
```

```
<properties resource="dbconfig.properties"></properties>
```

- 如果属性在不只一个地方进行了配置，那么 MyBatis 将按照下面的顺序来加载：
  - 在 properties 元素体内指定的属性首先被读取。
  - 然后根据 properties 元素中的 resource 属性读取类路径下属性文件或根据 url 属性指定的路径读取属性文件，并覆盖已读取的同名属性。
  - 最后读取作为方法参数传递的属性，并覆盖已读取的同名属性。

settings设置

- 这是 MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响的所有映射器中配置的**缓存**的全局开关。	true | false	TRUE
lazyLoadingEnabled	延迟加载的全局开关。**当开启时，所有关联对象都会延迟加载**。特定关联关系中可通过设置 fetchType 属性来覆盖该项的开关状态。	true | false	FALSE
useColumnLabel	**使用列标签代替列名**。不同的驱动在这方面会有不同的表现，具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。	true | false	TRUE
defaultStatementTimeout	**设置超时时间**，它决定驱动等待数据库响应的秒数。	Any positive integer	Not Set (null)
mapUnderscoreToCamelCase	**是否开启自动驼峰命名规则（camel case）映射**，即从经典数据库列名 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射	true | false	FALSE

```
<settings>
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

typeAliases别名处理器

- 类型别名是为 Java 类型设置一个短的名字，可以方便我们引用某个类。

```
<typeAliases>
    <typeAlias type="com.atguigu.bean.Employee" alias="employee" />
    <typeAlias type="com.atguigu.bean.Department" alias="department"/>
</typeAliases>
```

- 类很多的情况下，可以批量设置别名这个包下的每一个类创建一个默认的别名，就是简单类名小写。

```
<typeAliases>
    <package name="com.atguigu.bean" />
</typeAliases>
```

- 也可以使用@Alias注解为其指定一个别名

```
@Alias("emp")
public class Employee {
```

值得注意的是，MyBatis已经为许多常见的 Java 类型内建了相应的类型别名。它们都是大小写不敏感的，我们在起别名的时候千万不要占用已有的别名。

别名	映射的类型	别名	映射的类型	别名	映射的类型
_byte	byte	string	String	date	Date
_long	long	byte	Byte	decimal	BigDecimal
_short	short	long	Long	bigdecimal	BigDecimal
_int	int	short	Short	object	Object
_integer	int	int	Integer	map	Map
_double	double	integer	Integer	hashmap	HashMap
_float	float	double	Double	list	List
_boolean	boolean	float	Float	arraylist	ArrayList
		boolean	Boolean	collection	Collection
				iterator	Iterator

typeHandlers类型处理器

- 无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.lang.Boolean, boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.lang.Byte, byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short, short	数据库兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	java.lang.Integer, int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long, long	数据库兼容的 NUMERIC 或 LONG INTEGER
FloatTypeHandler	java.lang.Float, float	数据库兼容的 NUMERIC 或 FLOAT
DoubleTypeHandler	java.lang.Double, double	数据库兼容的 NUMERIC 或 DOUBLE
BigDecimalTypeHandler	java.math.BigDecimal	数据库兼容的 NUMERIC 或 DECIMAL
StringTypeHandler	java.lang.String	CHAR, VARCHAR

日期类型的处理

- 日期和时间的处理，JDK1.8以前一直是个头疼的问题。我们通常使用JSR310规范领导者Stephen Colebourne创建的Joda-Time来操作。1.8已经实现全部的JSR310规范了。
- 日期时间处理上，我们可以使用MyBatis基于JSR310（Date and Time API）编写的各种日期时间类型处理器。
- MyBatis3.4以前的版本需要我们手动注册这些处理器，以后的版本都是自动注册的

```
<typeHandlers>
  <typeHandler handler="org.apache.ibatis.type.InstantTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.LocalDateTimeTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.LocalDateTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.LocalTimeTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.OffsetDateTimeTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.OffsetTimeTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.ZonedDateTimeTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.YearTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.MonthTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.YearMonthTypeHandler" />
  <typeHandler handler="org.apache.ibatis.type.JapaneseDateTypeHandler" />
</typeHandlers>
```

自定义类型处理器

- 我们可以重写类型处理器或创建自己的类型处理器来处理不支持的或非标准的类型。

- 步骤：
  - 1）、实现org.apache.ibatis.type.TypeHandler接口或者继承org.apache.ibatis.type.BaseTypeHandler
  - 2）、指定其映射某个JDBC类型（可选操作）
  - 3）、在mybatis全局配置文件中注册

plugins插件

- 插件是MyBatis提供的一个非常强大的机制，我们可以通过插件来修改MyBatis的一些核心行为。插件通过动态代理机制，可以介入四大对象的任何一个方法的执行。后面会有专门的章节我们来介绍mybatis运行原理以及插件

- Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler (getParameterObject, setParameters)
- ResultSetHandler (handleResultSets, handleOutputParameters)
- StatementHandler (prepare, parameterize, batch, update, query)

environments环境

- MyBatis可以配置多种环境，比如开发、测试和生产环境需要有不同的配置。
- 每种环境使用一个environment标签进行配置并指定唯一标识符
- 可以通过environments标签中的default属性指定一个环境的标识符来快速的切换环境

environment-指定具体环境

- id：指定当前环境的唯一标识
- transactionManager、和dataSource都必须有

```
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC" />
        <dataSource type="POOLED">
            <property name="driver" value="${driver}" />
            <property name="url" value="${url}" />
            <property name="username" value="${username}" />
            <property name="password" value="${password}" />
        </dataSource>
    </environment>
</environments>
```

transactionManager

- type： JDBC | MANAGED | 自定义
  - JDBC：使用了 JDBC 的提交和回滚设置，依赖于从数据源得到的连接来管理事务范围。JdbcTransactionFactory
  - MANAGED：不提交或回滚一个连接、让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 ManagedTransactionFactory
  - 自定义：实现TransactionFactory接口，type=全类名/别名

dataSource

- type： UNPOOLED | POOLED | JNDI | 自定义
  - UNPOOLED：不使用连接池，UnpooledDataSourceFactory
  - POOLED：使用连接池， PooledDataSourceFactory
  - JNDI： 在EJB 或应用服务器这类容器中查找指定的数据源
  - 自定义：实现DataSourceFactory接口，定义数据源的获取方式。

- 实际开发中我们使用Spring管理数据源，并进行事务控制的配置来覆盖上述配置

databaseIdProvider环境

- MyBatis 可以根据不同的数据库厂商执行不同的语句。

```
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql"/>
    <property name="Oracle" value="oracle"/>
    <property name="SQL Server" value="sqlserver"/>
</databaseIdProvider>
```

- Type： DB_VENDOR
  - 使用MyBatis提供的VendorDatabaseIdProvider解析数据库厂商标识。也可以实现DatabaseIdProvider接口来自定义。
- Property-name：数据库厂商标识
- Property-value：为标识起一个别名，方便SQL语句使用databaseId属性引用

```
<select id="getEmpsByDeptId" resultType="Emp"
        parameterType="Integer" databaseId="mysql">
    SELECT * FROM tbl_emp WHERE deptId=#{id}
</select>
```

- DB_VENDOR
  - 会通过 DatabaseMetaData#getDatabaseProductName() 返回的字符串进行设置。由于通常情况下这个字符串都非常长而且相同产品的不同版本会返回不同的值，所以最好通过设置属性别名来使其变短

- MyBatis匹配规则如下：
  - 1、如果没有配置databaseIdProvider标签，那么databaseId=null
  - 2、如果配置了databaseIdProvider标签，使用标签配置的name去匹配数据库信息，匹配上设置databaseId=配置指定的值，否则依旧为null
  - 3、如果databaseId不为null，他只会找到配置databaseId的sql语句
  - 4、MyBatis 会加载不带 databaseId 属性和带有匹配当前数据库 databaseId 属性的所有语句。如果同时找到带有 databaseId 和不带 databaseId 的相同语句，则后者会被舍弃。

mapper映射

- mapper逐个注册SQL映射文件

```
<mappers>
    <mapper resource="mybatis/mapper/PersonDao.xml" />
    <mapper url="file:///D:/UserDao.xml"/>
    <mapper class="com.atguigu.dao.PersonDaoAnnotation"/>
</mappers>
```

- 或者使用批量注册：
  - 这种方式要求SQL映射文件名必须和接口名相同并且在同一目录下

```
<mappers>
    <package name="com.atguigu.dao"/>
</mappers>
```

四、MyBatis-映射文件

- 映射文件指导着MyBatis如何进行数据库增删改查，有着非常重要的意义；

  •cache –命名空间的二级缓存配置
  •cache-ref – 其他命名空间缓存配置的引用。
  •resultMap – 自定义结果集映射
  •parameterMap – 已废弃！老式风格的参数映射
  •sql –抽取可重用语句块。
  •insert – 映射插入语句
  •update – 映射更新语句
  •delete – 映射删除语句
  •select – 映射查询语句

insert、update、delete元素

id	**命名空间中的唯一标识符**
parameterType	将要传入语句的参数的完全限定类名或别名。这个属性是可选的，因为 **MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数类型**，默认值为 unset。
flushCache	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：true（对应插入、更新和删除语句）。
timeout	这个设置是在抛出异常之前，驱动程序**等待数据库返回请求结果的秒数**。默认值为 unset（依赖驱动）。
statementType	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，**PreparedStatement** 或 CallableStatement，**默认值：PREPARED**。
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。
keyColumn	（仅对 insert 和 update 有用）通过生成的键值设置表中的列名，这个设置仅在某些数据库（像 PostgreSQL）是必须的，当主键列不是表中的第一列的时候需要设置。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。
databaseId	**如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句**；如果带或者不带的语句都有，则不带的会被忽略。

主键生成方式

- 若数据库**支持自动生成主键**的字段（比如 MySQL 和 SQL Server），则可以设置 useGeneratedKeys=”true”，然后再把 keyProperty 设置到目标属性上。

```
<insert id="insertCustomer" databaseId="mysql"
    useGeneratedKeys="true" keyProperty="id">
    INSERT INTO customers2 (last_name, email, age)
    VALUES (#{lastName}, #{email}, #{age})
</insert>
```

主键生成方式

- 而对于不支持自增型主键的数据库（例如 Oracle），则可以使用 selectKey 子元素：selectKey 元素将会首先运行，id 会被设置，然后插入语句会被调用

```
<insert id="insertCustomer" databaseId="oracle"
    parameterType="customer">
    <selectKey order="BEFORE" keyProperty="id" resultType="_int">
        SELECT crm_seq.nextval
        FROM dual
    </selectKey>
    INSERT INTO customers2 (id, last_name, email, age)
    VALUES (#{id}, #{lastName}, #{email}, #{age})
</insert>
```

selectKey

keyProperty	**selectKey 语句结果应该被设置的目标属性。**
keyColumn	匹配属性的返回结果集中的列名称。
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。
order	**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，然后是 selectKey 元素
statementType	与前面相同，MyBatis 支持 STATEMENT，PREPARED 和 CALLABLE 语句的映射类型，分别代表 PreparedStatement 和 CallableStatement 类型

参数（Parameters）传递

- 单个参数
  - 可以接受基本类型，对象类型，集合类型的值。这种情况MyBatis可直接使用这个参数，不需要经过任何处理。
- 多个参数
  - 任意多个参数，都会被MyBatis重新包装成一个Map传入。Map的key是param1，param2，0，1…，值就是参数的值。
- 命名参数
  - 为参数使用@Param起一个名字，MyBatis就会将这些参数封装进map中，key就是我们自己指定的名字
- POJO
  - 当这些参数属于我们业务POJO时，我们直接传递POJO
- Map
  - 我们也可以封装多个参数为map，直接传递

参数处理

- 参数也可以指定一个特殊的数据类型：

```
#{property,javaType=int,jdbcType=NUMERIC}
```

```
#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}
```

  - javaType 通常可以从参数对象中来去确定
  - **如果 null 被当作值来传递，对于所有可能为空的列，jdbcType 需要被设置**
  - 对于数值类型，还可以设置小数点后保留的位数：
  - **mode 属性**允许指定 IN，OUT 或 INOUT 参数。如果参数为 OUT 或 INOUT，参数对象属性的真实值将会被改变，就像在获取输出参数时所期望的那样。

参数处理

- **参数位置支持的属性**
  - javaType、jdbcType、mode、numericScale、resultMap、typeHandler、jdbcTypeName、~~expression~~

- **实际上通常被设置的是：**

  可能为空的列名指定 jdbcType

- **#{key}：获取参数的值，预编译到SQL中。安全。**
- **${key}：获取参数的值，拼接到SQL中。有SQL注入问题。ORDER BY ${name}**

select元素

- Select元素来定义查询操作。
- Id：唯一标识符。
  - 用来引用这条语句，需要和接口的方法名一致
- parameterType：参数类型。
  - 可以不传，MyBatis会根据TypeHandler自动推断
- resultType：返回值类型。
  - 别名或者全类名，如果返回的是集合，定义集合中元素的类型。不能和resultMap同时使用

parameterType	将会传入这条语句的参数类的完全限定名或别名。这个属性是可选的，因为 **MyBatis 可以通过 TypeHandler 推断出具体传入语句的参数**，默认值为 unset。
resultType	从这条语句中返回的**期望类型的类的完全限定名或别名。注意如果是集合，那应该是集合可以包含的类型**，而不能是集合本身。该属性**和 resultMap，不能同时使用。**
resultMap	外部 resultMap 的命名引用。和 resultType 属性不能同时使用。
flushCache	将其设置为 true，任何时候只要语句被调用，都会导致本地缓存和二级缓存都会被清空，默认值：false
useCache	将其设置为 true，将会导致本条语句的结果被二级缓存，默认值：对 select 元素为 true。
timeout	这个设置是在抛出异常之前，**驱动程序等待数据库返回请求结果的秒数。**默认值为 unset（依赖驱动）。
fetchSize	影响驱动程序**每次批量返回的结果行数。**默认值为 unset（依赖驱动）。
statementType	STATEMENT，PREPARED 或 CALLABLE 的一个。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
resultSetType	FORWARD_ONLY，SCROLL_SENSITIVE 或 SCROLL_INSENSITIVE 中的一个，默认值为 unset （依赖驱动）
databaseId	**如果配置了 databaseIdProvider，MyBatis 会加载所有的不带 databaseId 或匹配当前 databaseId 的语句**；如果带或者不带的语句都有，则不带的会被忽略。
resultOrdered	这个设置仅针对嵌套结果 select 语句适用：如果为 true，就假设包含了嵌套结果集或是分组，这样当返回一个主结果行，就不会发生有对前面结果集引用的情况。这就使得在获取嵌套的结果集的时候不至于导致内存不够用。默认值：false
resultSets	这个设置仅对多结果集的情况适用，它将列出语句执行后返回的结果集并每个结果集给一个名称，名称是逗号分隔的

自动映射

- 1、全局setting设置
  - autoMappingBehavior默认是PARTIAL，开启自动映射的功能。唯一的要求是列名和javaBean属性名一致
  - 如果autoMappingBehavior设置为null则会取消自动映射
  - 数据库字段命名规范，POJO属性符合驼峰命名法，如A_COLUMN→aColumn，我们可以开启自动驼峰命名规则映射功能，mapUnderscoreToCamelCase=true。

- **2、自定义resultMap，实现高级结果集映射。**

resultMap

- constructor - 类在实例化时, 用来注入结果到构造方法中
  - idArg - ID 参数; 标记结果作为 ID 可以帮助提高整体效能
  - arg - 注入到构造方法的一个普通结果
- id – 一个 ID 结果; 标记结果作为 ID 可以帮助提高整体效能
- result – 注入到字段或 JavaBean 属性的普通结果
- association – 一个复杂的类型关联;许多结果将包成这种类型
  - 嵌入结果映射 – 结果映射自身的关联,或者参考一个
- collection – 复杂类型的集
  - 嵌入结果映射 – 结果映射自身的集,或者参考一个
- discriminator – 使用结果值来决定使用哪个结果映射
  - case – 基于某些值的结果映射
    - 嵌入结果映射 – 这种情形结果也映射它本身,因此可以包含很多相同的元素,或者它可以参照一个外部的结果映射。

id & result

- id 和 result 映射一个单独列的值到**简单数据类型**(字符串,整型,双精度浮点数,日期等)的属性或字段。

property	映射到列结果的字段或属性。例如：“username” 或 “address.street.number” 。
column	数据表的列名。通常和 resultSet.getString(columnName) 的返回值一致。
javaType	一个 Java 类的完全限定名, 或一个类型别名。如果映射到一个 JavaBean, MyBatis 通常可以断定类型
jdbcType	JDBC 类型是仅仅需要对插入, 更新和删除操作可能为空的列进行处理。
typeHandler	类型处理器。使用这个属性, 可以覆盖默认的类型处理器。这个属性值是类的完全限定名或者是一个类型处理 器的实现, 或者是类型别名。

association

- 复杂对象映射
- POJO中的属性可能会是一个对象
- 我们可以使用联合查询，并以级联属性的方式封装对象。

```
<resultMap type="com.atguigu.bean.Lock" id="myLock">
    <id column="id" property="id"/>
    <result column="lockName" property="lockName"/>
    <result column="key_id" property="key.id"/>
    <result column="keyName" property="key.keyName"/>
</resultMap>
```

- 使用association标签定义对象的封装规则

association-嵌套结果集

```
<resultMap type="com.atguigu.bean.Lock" id="myLock2">
    <id column="id" property="id"/>
    <result column="lockName" property="lockName"/>
    <association property="key" javaType="com.atguigu.bean.Key">
        <id column="key_id" property="id"/>
        <result column="keyName" property="keyName"/>
    </association>
</resultMap>
```

association-分段查询

```
<resultMap type="com.atguigu.bean.Lock" id="myLock3">
    <id column="id" property="id"/>
    <result column="lockName" property="lockName"/>
    <association property="key"
        select="com.atguigu.dao.KeyMapper.getKeyById"
        column="key_id">
    </association>
</resultMap>
```

select：调用目标的方法查询当前属性的值
column：将指定列的值传入目标方法

association-分段查询&延迟加载

开启延迟加载和属性按需加载

```
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

- 旧版本的MyBatis需要额外的支持包
  - asm-3.3.1.jar
  - cglib-2.2.2.jar

Collection-集合类型&嵌套结果集

```
<select id="getDeptById" resultMap="MyDept">
    SELECT d.id d_id,d.dept_name d_deptName,
    e.id e_id,e.last_name e_lastName,e.email e_email,
    e.gender e_gender,e.dept_id e_deptId
    FROM department d
    LEFT JOIN employee e ON e.`dept_id`=d.`id`
    WHERE d.`id`=#{id}
</select>
```

```
<resultMap type="com.atguigu.bean.Department" id="MyDept">
   <id column="d_id" property="id"/>
   <result column="d_deptName" property="deptName"/>
   <collection property="emps" ofType="com.atguigu.bean.Employee"
       columnPrefix="e_">
       <id column="id" property="id"/>
       <result column="lastName" property="lastName"/>
       <result column="email" property="email"/>
       <result column="gender" property="gender"/>
   </collection>
</resultMap>
```

Collection-分步查询&延迟加载

```
<resultMap type="com.atguigu.bean.Department" id="MyDeptStep">
    <id column="id" property="id"/>
    <result column="dept_name" property="deptName"/>
    <collection property="emps"
        select="com.atguigu.dao.EmployeeMapper.getEmpsByDeptId"
        column="id">
    </collection>
</resultMap>
```

扩展-多列值封装map传递

- 分步查询的时候通过column指定，将对应的列的数据传递过去，我们有时需要传递多列数据。
- 使用{key1=column1,key2=column2…}的形式

```
<resultMap type="com.atguigu.bean.Department" id="MyDeptStep">
    <id column="id" property="id"/>
    <result column="dept_name" property="deptName"/>
    <collection property="emps"
        select="com.atguigu.dao.EmployeeMapper.getEmpsByDeptId"
        column="{deptId=id}">
    </collection>
</resultMap>
```

- association或者collection标签的fetchType=eager/lazy可以覆盖全局的延迟加载策略，指定立即加载（eager）或者延迟加载（lazy）

五、MyBatis-动态SQL

- 动态 SQL是MyBatis强大特性之一。极大的简化我们拼装SQL的操作。
- 动态 SQL 元素和使用 JSTL 或其他类似基于 XML 的文本处理器相似。
- MyBatis 采用功能强大的基于 OGNL 的表达式来简化操作。
  - if
  - choose (when, otherwise)
  - trim (where, set)
  - foreach

if

```
<select id="getEmpWhereIf" resultType="com.atguigu.bean.Employee">
    select * from employee where
        <if test="id!=null">
            id=#{id} and
        </if>
        <if test="lastName!=null and !&quot;&quot;.equals(lastName)">
            last_name=#{lastName} and
        </if>
        <if test="gender==0 or gender==1">
            gender=#{gender}
        </if>
</select>
```

choose (when, otherwise)

```
<select id="getEmpChoose" resultType="com.atguigu.bean.Employee">
    select * from employee
    <where>
        <choose>
            <when test="id!=null">
                id=#{id}
            </when>
            <when test="lastName!=null and !&quot;&quot;.equals(lastName)">
                last_name=#{lastName}
            </when>
            <otherwise>
                1=1
            </otherwise>
        </choose>
    </where>
</select>
```

trim (where, set)

where

```
<select id="getEmpWhereIf" resultType="com.atguigu.bean.Employee">
    select * from employee
    <where>
        <if test="id!=null">
            id=#{id} and
        </if>
        <if test="lastName!=null and !&quot;&quot;.equals(lastName)">
            last_name=#{lastName} and
        </if>
        <if test="gender==0 or gender==1">
            gender=#{gender}
        </if>
    </where>
</select>
```

set

```
<update id="updateEmployee" parameterType="com.atguigu.bean.Employee">
    update employee
    <set>
        <if test="lastName!=null">
            last_name=#{lastName},
        </if>
        <if test="gender!=null and (gender==0 or gender ==1)">
            gender=#{gender},
        </if>
        <if test="email!=null">
            email=#{email}
        </if>
    </set>
    where id=#{id}
</update>
```

trim

```
<select id="getEmpWhereTrim" resultType="com.atguigu.bean.Employee">
    select * from employee
    <trim prefix="where" suffixOverrides="and">
        <if test="id!=null">
            id=#{id} and
        </if>
        <if test="lastName!=null and !&quot;&quot;.equals(lastName)">
            last_name=#{lastName} and
        </if>
        <if test="gender==0 or gender==1">
            gender=#{gender}
        </if>
    </trim>
</select>
```

foreach

- 动态 SQL 的另外一个常用的必要操作是需要对一个集合进行遍历，通常是在构建 IN 条件语句的时候。

```
<select id="getEmpByIdIn" resultType="com.atguigu.bean.Employee">
    select * from employee where id in
        <foreach collection="ids"
            close=")"  item="item_id" open="(" separator=",">
                #{item_id}
        </foreach>
</select>
```

- 当迭代列表、集合等可迭代对象或者数组时
  - index是当前迭代的次数，item的值是本次迭代获取的元素
- 当使用字典（或者Map.Entry对象的集合）时
  - index是键，item是值

bind

- bind 元素可以从 OGNL 表达式中创建一个变量并将其绑定到上下文。比如：

```
<select id="getEmpByLastNameLike" resultType="com.atguigu.bean.Employee">
    <bind name="myLastName" value="'%'+_lastName+'%'"/>
    select * from employee where last_name like #{myLastName}
</select>
```

Multi-db vendor support

- 若在 mybatis 配置文件中配置了 databaseIdProvider，则可以使用 “_databaseId”变量，这样就可以根据不同的数据库厂商构建特定的语句

```
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql"/>
    <property name="Oracle" value="oracle"/>
</databaseIdProvider>
```

```
<select id="getEmpPage" resultType="com.atguigu.bean.Employee">
    <if test="_databaseId=='mysql'">
        select * from employee where limit 0,5
    </if>
    <if test="_databaseId=='oracle'">
        select * from (select e.*,rownum as r1 from employee e where rownum &lt;= 5)
        where r1 &gt;= 1;
    </if>
</select>
```

OGNL（ Object Graph Navigation Language ）对象图导航语言，这是一种强大的表达式语言，通过它可以非常方便的来操作对象属性。 类似于我们的EL，SpEL等

访问对象属性：　　person.name
调用方法：　　person.getName()
调用静态属性/方法：　　@java.lang.Math@PI
@java.util.UUID@randomUUID()
调用构造方法：　　new com.atguigu.bean.Person('admin').name
运算符：　　+,-*,/,%
逻辑运算符：　　in,not in,>,>=,<,<=,==,!=

注意：xml中特殊符号如",>,<等这些都需要使用转义字符

访问集合伪属性：

类型	伪属性	伪属性对应的 Java 方法
List、Set、Map	size、isEmpty	List/Set/Map.size(),List/Set/Map.isEmpty()
List、Set	iterator	List.iterator()、Set.iterator()
Map	keys、values	Map.keySet()、Map.values()
Iterator	next、hasNext	Iterator.next()、Iterator.hasNext()

六、MyBatis-缓存机制

- MyBatis 包含一个非常强大的查询缓存特性,它可以非常方便地配置和定制。缓存可以极大的提升查询效率。

- MyBatis系统中默认定义了两级缓存。
- **一级缓存**和**二级缓存**。
  - 1、默认情况下，只有一级缓存（SqlSession级别的缓存，也称为本地缓存）开启。
  - 2、二级缓存需要手动开启和配置，他是基于namespace级别的缓存。
  - 3、为了提高扩展性。MyBatis定义了缓存接口Cache。我们可以通过实现Cache接口来自定义二级缓存

一级缓存

- 一级缓存(local cache), 即本地缓存, 作用域默认为sqlSession。当 Session flush 或 close 后, 该 Session 中的所有 Cache 将被清空。
- 本地缓存不能被关闭, 但可以调用 clearCache() 来清空本地缓存, 或者改变缓存的作用域.
- 在mybatis3.1之后, 可以配置本地缓存的作用域. 在 mybatis.xml 中配置

localCacheScope	MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。**默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。**	SESSION | STATEMENT	SESSION

一级缓存演示&失效情况

- 同一次会话期间只要查询过的数据都会保存在当前SqlSession的一个Map中
  - key:hashCode+查询的SqlId+编写的sql查询语句+参数
- 一级缓存失效的四种情况
  - 1、不同的SqlSession对应不同的一级缓存
  - 2、同一个SqlSession但是查询条件不同
  - 3、同一个SqlSession两次查询期间执行了任何一次增删改操作
  - 4、同一个SqlSession两次查询期间手动清空了缓存

二级缓存

- 二级缓存(second level cache)，全局作用域缓存
- 二级缓存默认不开启，需要手动配置
- MyBatis提供二级缓存的接口以及实现，缓存实现要求POJO实现Serializable接口
- **二级缓存在 SqlSession 关闭或提交之后才会生效**

- 使用步骤
  - 1、全局配置文件中开启二级缓存
    - <setting name="*cacheEnabled*" *value="true"*/>
  - 2、需要使用二级缓存的映射文件处使用cache配置缓存
    - <cache />
  - 3、注意：POJO需要实现Serializable接口

缓存相关属性

- eviction="FIFO"：缓存回收策略：
  - LRU – 最近最少使用的：移除最长时间不被使用的对象。
  - FIFO – 先进先出：按对象进入缓存的顺序来移除它们。
  - SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。
  - WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。
  - 默认的是 LRU。
- flushInterval：刷新间隔，单位毫秒
  - 默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新
- size：引用数目，正整数
  - 代表缓存最多可以存储多少个对象，太大容易导致内存溢出
- readOnly：只读，true/false
  - true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。
  - false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是 false。

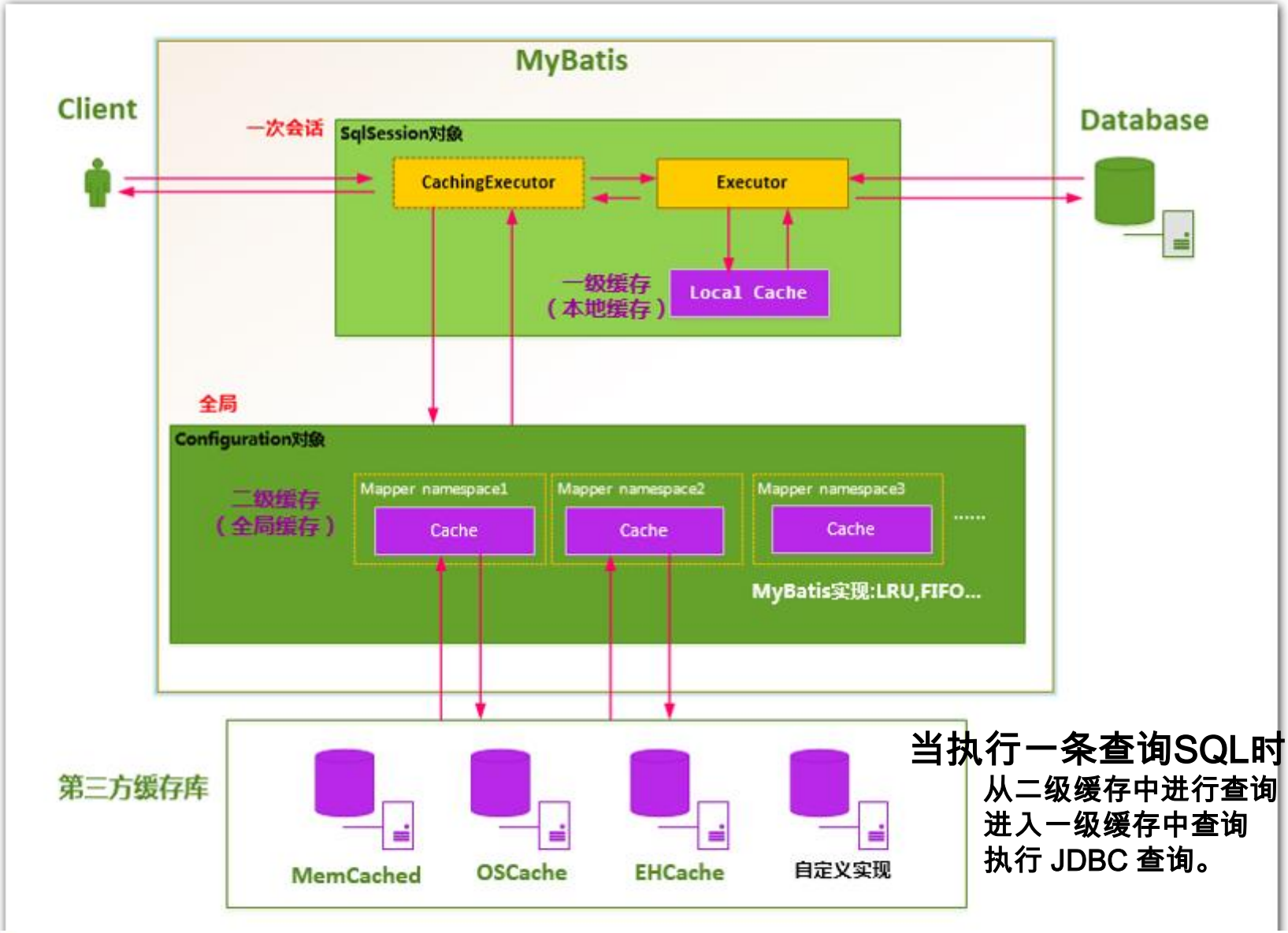
缓存有关设置

- 1、全局setting的cacheEnable：
  - 配置二级缓存的开关。一级缓存一直是打开的。
- 2、select标签的useCache属性：
  - 配置这个select是否使用二级缓存。一级缓存一直是使用的
- 3、sql标签的flushCache属性：
  - 增删改默认flushCache=true。sql执行以后，会同时清空一级和二级缓存。查询默认flushCache=false。
- 4、sqlSession.clearCache()：
  - 只是用来清除一级缓存。
- 5、当在某一个作用域 (一级缓存Session/二级缓存Namespaces) 进行了 C/U/D 操作后，默认该作用域下**所有 select 中的缓存将被clear**。

第三方缓存整合

- EhCache 是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认的CacheProvider。
- MyBatis定义了Cache接口方便我们进行自定义扩展。
- 步骤：
  - 1、导入ehcache包，以及整合包，日志包
    ehcache-core-2.6.8.jar、mybatis-ehcache-1.0.3.jar
    slf4j-api-1.6.1.jar、slf4j-log4j12-1.6.2.jar
  - 2、编写ehcache.xml配置文件
  - 3、配置cache标签
    - <cache type="org.mybatis.caches.ehcache.EhcacheCache"></cache>
- 参照缓存：若想在命名空间中共享相同的缓存配置和实例。可以使用 cache-ref 元素来引用另外一个缓存。

```
<cache-ref namespace="com.atguigu.mybatis.example.CustomerMapper"/>
```



当执行一条查询SQL时，流程为

从二级缓存中进行查询

进入一级缓存中查询

执行 JDBC 查询。

七、MyBatis-Spring整合

1、查看不同MyBatis版本整合Spring时使用的适配包；
http://www.mybatis.org/spring/
2、下载整合适配包
https://github.com/mybatis/spring/releases

MyBatis-Spring	MyBatis	Spring
1.0.0 and 1.0.1	3.0.1 to 3.0.5	3.0.0 or higher
1.0.2	3.0.6	3.0.0 or higher
1.1.0 or higher	3.1.0 or higher	3.0.0 or higher
1.3.0 or higher	3.4.0 or higher	3.0.0 or higher

- 3、官方整合示例，jpetstore
https://github.com/mybatis/jpetstore-6

整合关键配置

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
 <!-- 指定mybatis全局配置文件位置 -->
 <property name="configLocation" value="classpath:mybatis/mybatis-config.xml"></property>
 <!--指定数据源 -->
 <property name="dataSource" ref="dataSource"></property>
 <!--mapperLocations：所有sql映射文件所在的位置 -->
 <property name="mapperLocations" value="classpath:mybatis/mapper/*.xml"></property>
 <!--typeAliasesPackage：批量别名处理-->
 <property name="typeAliasesPackage" value="com.atguigu.bean"></property>
</bean>

 <!--自动的扫描所有的mapper的实现并加入到ioc容器中 -->
<bean id="configure" class="org.mybatis.spring.mapper.MapperScannerConfigurer">
 <!– basePackage:指定包下所有的mapper接口实现自动扫描并加入到ioc容器中 -->
 <property name="basePackage" value="com.atguigu.dao"></property>
</bean>
```

八、MyBatis-逆向工程

- **MyBatis Generator：**
- 简称MBG，是一个专门为MyBatis框架使用者定制的代码生成器，可以快速的根据表生成对应的映射文件，接口，以及bean类。支持基本的增删改查，以及QBC风格的条件查询。但是表连接、存储过程等这些复杂sql的定义需要我们手工编写
- 官方文档地址

  http://www.mybatis.org/generator/
- 官方工程地址

  https://github.com/mybatis/generator/releases

MBG使用

- 使用步骤：
  - 1）编写MBG的配置文件（重要几处配置）
    1）jdbcConnection配置数据库连接信息
    2）javaModelGenerator配置javaBean的生成策略
    3）sqlMapGenerator 配置sql映射文件生成策略
    4）javaClientGenerator配置Mapper接口的生成策略
    5）table 配置要逆向解析的数据表
    tableName：表名
    domainObjectName：对应的javaBean名
  - 2）运行代码生成器生成代码
- 注意：

  Context标签

  targetRuntime="MyBatis3"可以生成带条件的增删改查

  targetRuntime="MyBatis3Simple"可以生成基本的增删改查

  如果再次生成，建议将之前生成的数据删除，避免xml向后追加内容出现的问题。

MBG配置文件

```
<generatorConfiguration>
  <context id="DB2Tables" targetRuntime="MyBatis3">
    //数据库连接信息配置
    <jdbcConnection driverClass="com.mysql.jdbc.Driver"
        connectionURL="jdbc:mysql://localhost:3306/bookstore0629"
        userId="root" password="123456">
    </jdbcConnection>
    //javaBean的生成策略
    <javaModelGenerator targetPackage="com.atguigu.bean"  targetProject=".\src">
      <property name="enableSubPackages" value="true" />
      <property name="trimStrings" value="true" />
    </javaModelGenerator>
    //映射文件的生成策略
    <sqlMapGenerator targetPackage="mybatis.mapper"   targetProject=".\conf">
      <property name="enableSubPackages" value="true" />
    </sqlMapGenerator>
    //dao接口java文件的生成策略
    <javaClientGenerator type="XMLMAPPER"  targetPackage="com.atguigu.dao"
      targetProject=".\src">
      <property name="enableSubPackages" value="true" />
    </javaClientGenerator>
    //数据表与javaBean的映射
    <table tableName="books" domainObjectName="Book"></table>
  </context>
</generatorConfiguration>
```

生成器代码

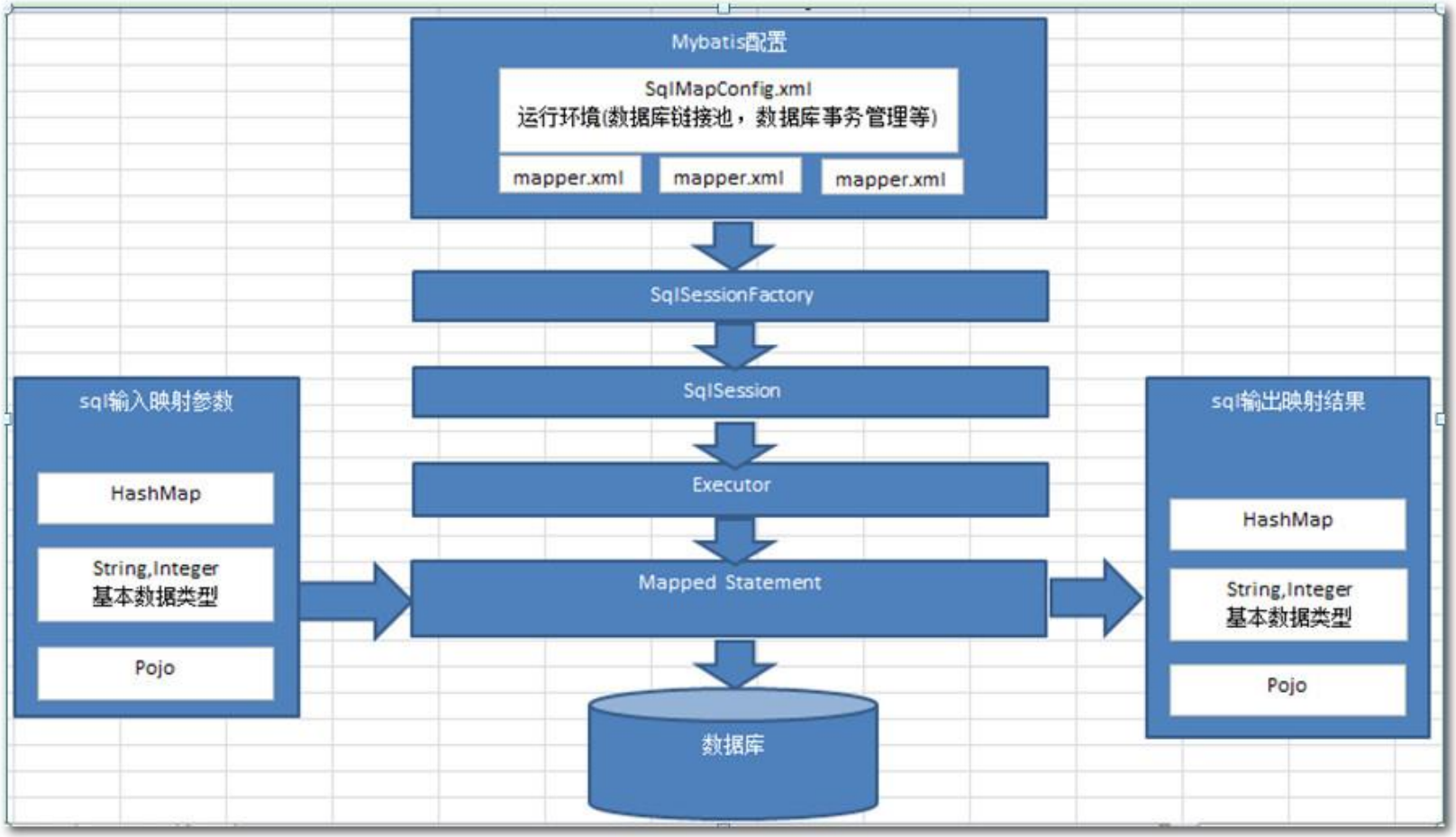
```
public static void main(String[] args) throws Exception {
    List<String> warnings = new ArrayList<String>();
    boolean overwrite = true;
    File configFile = new File("mbg.xml");
    ConfigurationParser cp = new ConfigurationParser(warnings);
    Configuration config = cp.parseConfiguration(configFile);
    DefaultShellCallback callback = new DefaultShellCallback(overwrite);
    MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,
    callback, warnings);
    myBatisGenerator.generate(null);
}
```

测试查询：

QBC风格的带条件查询

```
@Test
public void test01(){
    SqlSession openSession = build.openSession();
    DeptMapper mapper = openSession.getMapper(DeptMapper.class);
    DeptExample example = new DeptExample();
    //所有的条件都在example中封装
    Criteria criteria = example.createCriteria();
    //select id, deptName, locAdd from tbl_dept WHERE
    ///( deptName like ? and id > ? )
    criteria.andDeptnameLike("%部%");
    criteria.andIdGreaterThan(2);
    List<Dept> list = mapper.selectByExample(example);
    for (Dept dept : list) {
        System.out.println(dept);
    }
}
```

九、MyBatis-工作原理



十、MyBatis-插件开发

- MyBatis**在四大对象的创建过程中，都会有插件进行介入。**插件可以利用动态代理机制一层层的包装目标对象，而实现在目标对象执行目标方法之前进行拦截的效果。
- MyBatis 允许在已映射语句执行过程中的某一点进行拦截调用。
- 默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：
  - Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
  - ParameterHandler (getParameterObject, setParameters)
  - ResultSetHandler (handleResultSets, handleOutputParameters)
  - StatementHandler (prepare, parameterize, batch, update, query)

插件开发

- 插件开发步骤
  - 1）、编写插件实现Interceptor接口，并使用@Intercepts注解完成插件签名

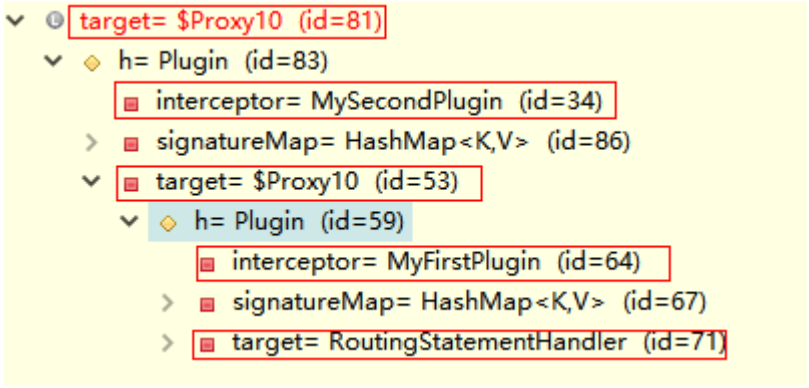
```
@Intercepts({
    @Signature(type=StatementHandler.class,method="prepare",
            args={Connection.class})
})
public class MyFirstPlugin implements Interceptor{
```

  - 2）、在全局配置文件中注册插件

```
<plugins>
    <plugin interceptor="com.atguigu.plugin.MyFirstPlugin">
        <property name="username" value="tomcat"/>
    </plugin>
</plugins>
```

插件原理

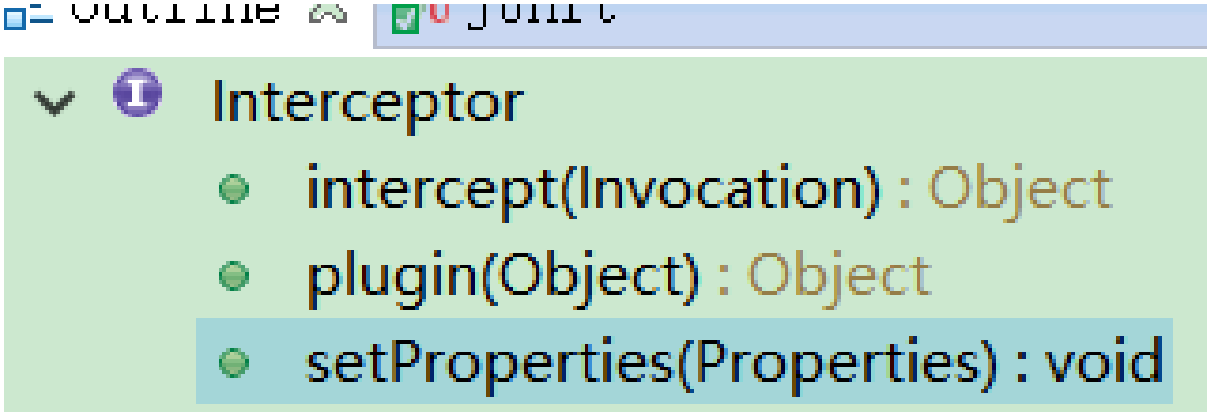
- 1）、按照插件注解声明，按照插件配置顺序调用插件plugin方法，生成被拦截对象的动态代理
- 2）、多个插件依次生成目标对象的代理对象，层层包裹，先声明的先包裹；形成代理链
- 3）、目标方法执行时依次从外到内执行插件的intercept方法。



- 4）、多个插件情况下，我们往往需要在某个插件中分离出目标对象。可以借助MyBatis提供的SystemMetaObject类来进行获取最后一层的h以及target属性的值

Interceptor接口

- Intercept：拦截目标方法执行
- plugin：生成动态代理对象，可以使用MyBatis提供的Plugin类的wrap方法
- setProperties：注入插件配置时设置的属性



常用代码：

从代理链中分离真实被代理对象

```
//1、分离代理对象。由于会形成多次代理，所以需要通过一个while 循环分离出最终被代理对象，从而方便提取信息
MetaObject metaObject = SystemMetaObject.forObject(target);
while (metaObject.hasGetter("h")) {
    Object h = metaObject.getValue("h");
    metaObject = SystemMetaObject.forObject(h);
}
//2、获取到代理对象中包含的被代理的真实对象
Object obj = metaObject.getValue("target");
//3、获取被代理对象的MetaObject方便进行信息提取
MetaObject forObject = SystemMetaObject.forObject(obj);
```

扩展：MyBatis实用场景

- 1）、PageHelper插件进行分页
- 2）、批量操作
- 3）、存储过程
- 4）、typeHandler处理枚举

PageHelper插件进行分页

- PageHelper是MyBatis中非常方便的第三方分页插件。
- 官方文档：

  https://github.com/pagehelper/Mybatis-PageHelper/blob/master/README_zh.md
- 我们可以对照官方文档的说明，快速的使用插件

使用步骤

- 1、导入相关包pagehelper-x.x.x.jar 和 jsqlparser-0.9.5.jar。
- 2、在MyBatis全局配置文件中配置分页插件。

```
<plugins>
    <!-- com.github.pagehelper为PageHelper类所在包名 -->
    <plugin interceptor="com.github.pagehelper.PageInterceptor">
        <!-- 使用下面的方式配置参数，后面会有所有的参数介绍 -->
        <property name="param1" value="value1"/>
    </plugin>
</plugins>
```

- 3、使用PageHelper提供的方法进行分页
- 4、可以使用更强大的PageInfo封装返回结果

批量操作

- 默认的 openSession() 方法没有参数,它会创建有如下特性的
  - 会开启一个事务(也就是**不自动提交**)
  - 连接对象会从由活动环境配置的数据源实例得到。
  - 事务隔离级别将会使用驱动或数据源的默认设置。
  - 预处理语句不会被复用,也不会批量处理更新。
- openSession 方法的 **ExecutorType** 类型的参数，枚举类型:
  - ExecutorType.SIMPLE: 这个执行器类型不做特殊的事情（这是默认装配的）。它**为每个语句的执行创建一个新的预处理语句**。
  - ExecutorType.REUSE: 这个执行器类型**会复用预处理语句**。
  - ExecutorType.BATCH: 这个执行器会**批量执行所有更新语句**

```
SqlSession openSession(boolean autoCommit);
SqlSession openSession(Connection connection);
SqlSession openSession(TransactionIsolationLevel level);

SqlSession openSession(ExecutorType execType);
SqlSession openSession(ExecutorType execType, boolean autoCommit);
SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level);
SqlSession openSession(ExecutorType execType, Connection connection);
```

- 批量操作我们是使用MyBatis提供的BatchExecutor进行的，他的底层就是通过jdbc攒sql的方式进行的。我们可以让他攒够一定数量后发给数据库一次。

```
public void test01() {
    SqlSession openSession = build.openSession(ExecutorType.BATCH);
    UserDao mapper = openSession.getMapper(UserDao.class);
    long start = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {
        String name = UUID.randomUUID().toString().substring(0, 5);
        mapper.addUser(new User(null, name, 13));
    }
    openSession.commit();
    openSession.close();
    long end = System.currentTimeMillis();
    System.out.println("耗时时间："+(end-start));
}
```

100万记录添加测试结果

耗时时间：75567

- 与Spring整合中，我们推荐，额外的配置一个可以专门用来执行批量操作的sqlSession

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactoryBean"></constructor-arg>
    <constructor-arg name="executorType" value="BATCH"></constructor-arg>
</bean>
```

- 需要用到批量操作的时候，我们可以注入配置的这个批量SqlSession。通过他获取到mapper映射器进行操作。

- **注意：**
- 1、批量操作是在session.commit()以后才发送sql语句给数据库进行执行的
- 2、如果我们想让其提前执行，以方便后续可能的查询操作获取数据，我们可以使用sqlSession.flushStatements()方法，让其直接冲刷到数据库进行执行。

存储过程

- 实际开发中，我们通常也会写一些存储过程，MyBatis也支持对存储过程的调用
- 一个最简单的存储过程

```
delimiter $$
create procedure test()
begin
        select 'hello';
end $$
delimiter ;
```

- 存储过程的调用

1、select标签中statementType=“CALLABLE”

2、标签体中调用语法：

```
{call procedure_name(#{param1_info},#{param2_info})}
```

存储过程-游标处理

- MyBatis对存储过程的游标提供了一个JdbcType=CURSOR的支持，可以智能的把游标读取到的数据，映射到我们声明的结果集中
- 调用实例：

```
<select id="getPage" parameterType="PageEmp" statementType="CALLABLE"
    databaseId="oracle">
    {call PAGE_EMP(
    #{start,mode=IN,jdbcType=INTEGER},
    #{end,mode=IN,jdbcType=INTEGER},
    #{count,mode=OUT,jdbcType=INTEGER},
    #{emps,mode=OUT,jdbcType=CURSOR,javaType=ResultSet,resultMap=TestEmp}
    )}
</select>
```

```
<resultMap type="Emp" id="TestEmp">
    <id column="EMPNO" property="id"/>
</resultMap>
```

```
orcl.driver=oracle.jdbc.OracleDriver
orcl.url=jdbc:oracle:thin:@localhost:1521:orcl
orcl.username=scott
orcl.password=123456
```

```
public class PageEmp {
    private int start;
    private int end;
    private int count;
    private List<Emp> emps;
```

```
<environment id="oracle_dev">
    <transactionManager type="JDBC" />
    <dataSource type="POOLED">
        <property name="driver" value="${orcl.driver}" />
        <property name="url" value="${orcl.url}" />
        <property name="username" value="${orcl.username}" />
        <property name="password" value="${orcl.password}" />
    </dataSource>
</environment>
```

```
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql"/>
    <property name="Oracle" value="oracle"/>
</databaseIdProvider>
```

```
create or replace procedure
hello_test(p_start in int,
           p_end in int,
           p_count out int,
           ref_cur out sys_refcursor) AS
BEGIN
  select count(*) into p_count from emp;
  open ref_cur for
       select * from (select e.*,rownum as r1 from emp e where rownum<p_end)
       where r1>p_start;
END hello_test;
```

自定义TypeHandler处理枚举

- 我们可以通过自定义TypeHandler的形式来在设置参数或者取出结果集的时候自定义参数封装策略。

- 步骤：
  - 1、实现TypeHandler接口或者继承BaseTypeHandler
  - 2、使用@MappedTypes定义处理的java类型
    使用@MappedJdbcTypes定义jdbcType类型
  - 3、在自定义结果集标签或者参数处理的时候声明使用自定义TypeHandler进行处理
    或者在全局配置TypeHandler要处理的javaType

- 测试实例

一个代表部门状态的枚举类

```
public enum DeptStatus {

    WOKRING(100, "部门正在工作中"),
    MEETING(200, "部门正在开会中"),
    VOCATION(300, "部门正在休假中");
```

1、测试全局配置*EnumOrdinalTypeHandler*

id	dept_name	status
6	dept02	VOCATION

```
<typeHandlers>
    <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler"
                 javaType="com.atguigu.bean.DeptStatus"/>
</typeHandlers>
```

- 2、测试全局配置*EnumTypeHandler*

id	dept_name	status
8	dept01	VOCATION

```
<typeHandlers>
    <typeHandler handler="org.apache.ibatis.type.EnumTypeHandler"
                 javaType="com.atguigu.bean.DeptStatus"/>
</typeHandlers>
```

- 3、测试参数位置设置自定义TypeHandler

id	dept_name	status
4	dept01	300

```
<insert id="addDept">
    insert into department(dept_name,status)
    values(#{deptName},#{status,typeHandler=com.atguigu.type.MyEnumTypeHandler})
</insert>
```

自定义TypeHandler

```
public class MyEnumTypeHandler implements TypeHandler<DeptStatus>{

    @Override
    public void setParameter(PreparedStatement ps, int i,
            DeptStatus parameter, JdbcType jdbcType) throws SQLException {
        // TODO Auto-generated method stub
        Integer code = parameter.getCode();
        ps.setInt(i, code);
    }

    @Override
    public DeptStatus getResult(ResultSet rs,
            String columnName) throws SQLException {
        int code = rs.getInt(columnName);
        DeptStatus status = DeptStatus.getStatusByCode(code);
        return status;
    }

    @Override
    public DeptStatus getResult(ResultSet rs, int columnIndex)
            throws SQLException {
        // TODO Auto-generated method stub
        int code = rs.getInt(columnIndex);
        DeptStatus status = DeptStatus.getStatusByCode(code);
        return status;
    }

    @Override
    public DeptStatus getResult(CallableStatement cs, int columnIndex)
            throws SQLException {
        // TODO Auto-generated method stub
        int code = cs.getInt(columnIndex);
        DeptStatus status = DeptStatus.getStatusByCode(code);
        return status;
    }
```


尚硅谷
www.atguigu.com