

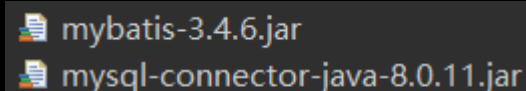
MyBatis 学习笔记

概述

- Mybatis 是支持普通 SQL 查询、存储过程、高级映射的优秀持久层框架。
- Mybatis 消除了几乎所有的 jdbc 代码和参数的手工设置以及对结果集的检索封装。
- 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 映射成数据库中的记录。
- JDBC——>butils——>MyBatis——>Hibernate

使用步骤

- 加包:



mybatis-3.4.6.jar
mysql-connector-java-8.0.11.jar

- Src 下新建配置文件 **conf.xml**——连接数据库

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN" "mybatis-3-config.dtd" >
<configuration>

    <!-- 可以配置多个运行环境，但是每个SqlSessionFactory 实例只能选择一个运行环境 -->
    <environments default="development">
        <!-- 有两个模式：1.work 工作模式；2. development工作模式 -->
        <environment id="development">
            <!-- id必须更上面的default一样 -->

            <transactionManager type="JDBC"></transactionManager><!-- type还可以是Manage模式，交由容器管理 -->
            <!-- UNPOOLED POOLED JNDI -->
            <dataSource type="UNPOOLED"> <!-- POOLED - 这是 JDBC 连接对象的数据源连接池的实现,用来避免创建新的连接实例 时
            <property name="driver" value="com.mysql.jdbc.Driver" />
            <property name="url" value="jdbc:mysql://localhost:3306/mybatis" />
            <property name="username" value="root" />
            <property name="password" value="xt222483" />
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="com/zhaio/mapper/StudentMapper.xml" />
    </mappers>
</configuration>
```

- 创建类如 User: 类需要有和表对应的变量。创建 getter、setter

```

package com.edwin;

public class User {
    private int id;
    private String name;
    private int age;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String toString() {
        return "id= "+id+" ,name = "+name+", age =" +age;
    }
}

```

- 和该类同级目录下定义表的映射文件：tableMapper.xml 如 userMapper.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.edwin.userMapper">

    <!--
    如果表中的字段名和实体类属性名不相同，那么查询出来的结果为 null。
    -->
    <select id="getUser" parameterType="int" resultType="com.edwin.User">
        <!-- resultType="com.edwin.User"反射机制，全类名 -->
        select * from user where id = #{id}
    </select>
</mapper>

```

- 在 conf.xml 中注册 userMapper.xml

```

<mappers>
    <mapper resource="com/edwin/userMapper.xml"/>
</mappers>

```

- 在 main 或其他类中使用：

```

package com.edwin;
import java.io.InputStream;
public class UserTest {
    public void test() {
        //加载mybatis配置文件
        String resource = "conf.xml";
        InputStream is = UserTest.class.getClassLoader().getResourceAsStream(resource);
        //构建SQLSession工厂
        SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
        //创建能够执行SQL的SQLSession
        SqlSession session = factory.openSession();
        //映射SQL的标识字符串
        String statement = "com.edwin.userMapper."+ "getUser"; //路径+id
        //执行查询返回一个唯一user对象的SQL
        //第二个参数表示根据表中的id来查询（在mapper中使用了#{id}）
        User user = session.selectOne(statement, 4);

        System.out.println(user.toString());
    }
    public static void main(String[] args) {
        (new UserTest()).test();
    }
}

```

CRUD by XML

● Create:

```

<mapper namespace="com.edwin2.userMapper">

    <insert id="addUser" parameterType="com.edwin.User">
        insert into users(name,age) values(#{name},#{age})
    </insert>

    <!-- <insert id="addUser" parameterType="com.edwin.User">
        insert into users(id,name,age) values(#{id},#{name},#{age})
    </insert>
    这里为啥不使用id:如果不使用id,那么就会自动递增使用。不会重复。平时开发这个应该也是不需要指定的,因为你添加的时候并不知道哪些id可用。
-->

```

```

public void test() {
    //加载mybatis配置文件
    String resource = "conf.xml";
    InputStream is = UserTest.class.getClassLoader().getResourceAsStream(resource);
    //构建SQLSession工厂
    SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(is);
    //默认是手动提交的,所以只是这样并不能添加到数据库!
    SqlSession session = factory.openSession();

    //映射SQL的标识字符串
    String statement = "com.edwin2.userMapper."+ "addUser"; //路径+id
    //执行查询返回一个唯一user对象的SQL

    int insert = session.insert(statement, new com.edwin.User(3,"Xutao",22));
    //必须要提交:
    session.commit();

    System.out.println(insert); //添加成功则返回1
}

```

● Delete

```

<delete id="deleteUser" parameterType="int">
    delete from users where id=#{id}
</delete>

```

```

//默认是手动提交的,所以只是这样并不能添加到数据库!
SqlSession session = factory.openSession(true); //或者参数设置为true,这样就会自动提交,不同commit了
String statement = "com.edwin2.userMapper."+ "deleteUser"; //路径+id
int delete = session.delete(statement, 2);
System.out.println(delete); //删除几条记录

```

- Retrieve:

```
<select id="getAllUsers" resultType="com.edwin.User">
    select * from users
</select>
```

//默认是手动提交的，所以只是这样并不能添加到数据库！

SqlSession session = factory.openSession(true); //或者参数设置为true，这样就会自动提交，不同commit了

String statement = "com.edwin2.userMapper."+ "getAllUsers"; //路径+id

java.util.List<Object> list = session.selectList(statement);

System.out.println(list.toString()); //删除几条记录

CURD -By Annotation

- 定义 SQL 映射接口

```
public interface UserMapper {

    @Insert("insert into users(name, age) values(#{name}, #{age})")
    public int insertUser(User user);

    @Delete("delete from users where id=#{id}")
    public int deleteUserById(int id);

    @Update("update users set name=#{name},age=#{age} where id=#{id}")
    public int updateUser(User user);

    @Select("select * from users where id=#{id}")
    public User getUserById(int id);

    @Select("select * from users")
    public List<User> getAllUser();

}
```

使用接口 UserMapper.java(不要使用类)

```
import com.edwin.User;

public interface UserMapper {

    @Insert("insert into users(name,age) values(#{name},#{age})")
    public int add(User user) ;

    @Delete("delete from users where id=#{id}")
    public int deleteById(User user);

    @Update("update users set name=#{name},age=#{age} where id = #{id}")
    public int update(User user);

    @Select("select * from users where id=#{id}")
    public User getById(int id);

    @Select("select * from users")
    public List<User> getAll();

}
```

- 在 conf.xml 中注册: <mapper class = "">

```
<mapper class = "com.edwin3.UserMapper"></mapper>
```

- 使用:

```

SqlSession session = factory.openSession(true);
UserMapper mapper = session.getMapper(UserMapper.class);
int add = mapper.add(new com.edwin.User(1,"edwin",21));
System.out.println(add);

session.close();

```

注意：这个 mapper 的接口是不需要实现的，直接使用即可！



官方文档学习

➤ <https://mybatis.org/mybatis-3/zh/index.html>

安装

```

<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>

```

SpringBoot 可以采用 mybatis-spring-boot-starter

SqlSessionFactory

Mybatis 应用是以一个 SqlSessionFactory 的实例为核心。

SqlSessionFactory 实例可以（还可以从配置构造）通过 SqlSessionFactoryBuilder 获取。

SqlSessionFactoryBuilder 是事先重配置中构造的。

从 XML 中构造 SqlSessionFactory

声明配置文件：

```
<?xml version="1.0" encoding="UTF-8" ?>
```

```

<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost:3306/mybatislearning?serverTimezone=UTC"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="mappers/demo1/Demo1Mapper.xml"/>
  </mappers>
</configuration>

```

配置文件中的 mapper:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.edw.mybatislearning.dao.UserDao">
  <resultMap id="user" type="cn.edw.mybatislearning.domain.User">
    <result property="uid" column="uid" javaType="java.lang.Integer"/>
    <result property="username" column="username" javaType="java.lang.String"/>
    <result property="password" column="password" javaType="java.lang.String"/>
    <result property="sex" column="sex" javaType="java.lang.Integer"/>
    <result property="age" column="age" javaType="java.lang.Integer"/>
  </resultMap>

  <select id="getFirstUser" resultMap="user">
    select * from user limit 1;
  </select>
</mapper>

```

User: 和数据库中对应

```

public class User {
  private int uid;
  private String username;
  private String password;
  private int sex;
  private int age;
}

```

DAO:

```

public interface UserDao {
  User getFirstUser();
}

```

生成 SqlSessionFactory 并使用

```

public class ConstructSqlSessionFactoryFromXmlConfig {
  public static void main(String[] args) throws IOException {
    // 配置文件地址
    String resource = "config\\demo1\\demo1-config.xml";
    // 声明输入流
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // SqlSessionFactoryBuilder 生成 SqlSessionFactory
    SqlSessionFactory sqlSessionFactory = new
      SqlSessionFactoryBuilder().build(inputStream);

    // 打开一个 session
    SqlSession session = sqlSessionFactory.openSession();
  }
}

```

```

// 使用 statement 执行 SQL 任务
String statement = "cn.edw.mybatislearning.dao.UserDao.getFirstUser";
User user = session.selectOne(statement, User.class);
System.out.println(user.toString());
// User{uid=1, username='Edwin Xu', password='pw', sex=1, age=100}
}
}

```

从 SqlSessionFactory 中获取 SqlSession

可以从 SqlSessionFactory 中获取 SqlSession, SqlSession 提供了执行 SQL 所需要的所有方法。

```

// 打开一个 session
SqlSession session = sqlSessionFactory.openSession();
// 使用 statement 执行 SQL 任务
String statement = "cn.edw.mybatislearning.dao.UserDao.getFirstUser";
User user = session.selectOne(statement, 101);

Or:
try (SqlSession session = sqlSessionFactory.openSession()) {
    Blog blog = (Blog) session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);
}

```

上面这种方式需要把 mapper 方法使用字符串硬编码，并不好。

方式 2:

```

// 通过 session.getMapper 获取 DAO 实例(Mapper)
UserDao userDao = session.getMapper(UserDao.class);
// 通过带 DAO 实例执行方法
User user1 = userDao.getFirstUser();
System.out.println(user1.toString());

```

第二种方法有很多优势，首先它不依赖于字符串字面值，会更安全一点；其次，如果你的 IDE 有代码补全功能，那么代码补全可以帮你快速选择到映射好的 SQL 语句。

原理概述

SqlSession\Mapper 到底干了什么？

Mybatis 可以使用 XML 和 java 进行配置使用。

采用 java 的 mapper:

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.edw.mybatislearning.dao.UserDao">
    <select id="getFirstUser" resultMap="user">
        select * from user limit 1;
    </select>
</mapper>

```

Namespace+select 的 ID 定义了具体的操作，即

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

命名空间的作用有两个，一个是利用更长的全限定名来将不同的语句隔离开来，同时也实现了你上面见到的**接口绑定**。长远来看，只要将命名空间置于合适的 Java 包命名

空间之中，你的代码会变得更加整洁，也有利于你更方便地使用 MyBatis。

Java 注解

```
public interface BlogMapper {  
    @Select("SELECT * FROM blog WHERE id = #{id}")  
    Blog selectBlog(int id);  
}
```

使用注解来映射简单语句会使代码显得更加简洁，但对于稍微复杂一点的语句，Java 注解不仅力不从心，还会让你本就复杂的 SQL 语句更加混乱不堪。因此，如果你需要做一些很复杂的操作，最好用 XML 来映射语句。

作用域 (Scope) 和生命周期

1. SqlSessionFactoryBuilder

这个类可以被实例化、使用和丢弃，一旦创建了 `SqlSessionFactory`，就不再需要它了。因此 `SqlSessionFactoryBuilder` 实例的最佳作用域是方法作用域(也就是局部方法变量)。你可以重用 `SqlSessionFactoryBuilder` 来创建多个 `SqlSessionFactory` 实例，但最好还是不要一直保留着它，以保证所有的 XML 解析资源可以被释放给更重要的事情。

2. SqlSessionFactory

`SqlSessionFactory` 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。使用 `SqlSessionFactory` 的最佳实践是在应用运行期间不要重复创建多次，多次重建 `SqlSessionFactory` 被视为一种代码“坏习惯”。因此 `SqlSessionFactory` 的最佳作用域是应用作用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

3. SqlSession

每个线程都应该有它自己的 `SqlSession` 实例。`SqlSession` 的实例不是线程安全的，因此是不能被共享的，所以它的作用域是请求或方法作用域。绝对不能将 `SqlSession` 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将 `SqlSession` 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 `HttpSession`。如果你现在正在使用一种 Web 框架，考虑将 `SqlSession` 放在一个和 HTTP 请求相似的作用域中。换句话说，每次收到 HTTP 请求，就可以打开一个 `SqlSession`，返回一个响应后，就关闭它。这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 `finally` 块中。下面的示例就是一个确保 `SqlSession` 关闭的标准模式：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    // 你的应用逻辑代码  
}
```

在所有代码中都遵循这种使用模式，可以保证所有数据库资源都能被正确地关闭。

配置

MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息。配置文档的顶层结构如下：

- configuration (配置)
 - properties (属性)
 - settings (设置)
 - typeAliases (类型别名)
 - typeHandlers (类型处理器)
 - objectFactory (对象工厂)
 - plugins (插件)
 - environments (环境配置)
 - 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)

```
settings
plugins
databaseIdProvider
environments
mappers
objectFactory
objectWrapperFactory
properties
reflectorFactory
typeAliases
typeHandlers
```

配置:属性 properties

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。例如：

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

设置好的属性可以在整个配置文件中用来替换需要动态配置的属性值。比如：

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
```

```
<property name="url" value="${url}"/>
<property name="username" value="${username}"/>
<property name="password" value="${password}"/>
</dataSource>
```

这个例子中的 `username` 和 `password` 将会由 `properties` 元素中设置的相应值来替换。`driver` 和 `url` 属性将会由 `config.properties` 文件中对应的值来替换。这样就为配置提供了诸多灵活选择。

也可以在 `SqlSessionFactoryBuilder.build()` 方法中传入属性值。例如：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, props);

// ... 或者 ...

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment,
props);
```

如果一个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：

1. 首先读取在 `properties` 元素体内指定的属性。
2. 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，并覆盖之前读取过的同名属性。
3. 最后读取作为方法参数传递的属性，并覆盖之前读取过的同名属性。

通过方法参数传递的属性具有最高优先级，`resource/url` 属性中指定的配置文件次之，最低优先级的则是 `properties` 元素中指定的属性

从 MyBatis 3.4.2 开始，你可以为占位符指定一个默认值。例如：

```
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${username:ut_user}"/> <!-- 如果属性 'username' 没有被配置，'username' 属性的值将为 'ut_user' -->
</dataSource>
```

这个特性默认是关闭的。要启用这个特性，需要添加一个特定的属性来开启这个特性。例如：

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value"
value="true"/> <!-- 启用默认值特性 -->
</properties>
```

如果你在属性名中使用了 `":"` 字符（如：`db:username`），或者在 SQL 映射中使用了 OGNL 表达式的三元运算符（如：`${tableName != null ? tableName : 'global_constants'}`），就需要设置特定的属性来修改分隔属性名和默认值的字符

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.default-value-separator"
value="?:"/> <!-- 修改默认值的分隔符 -->
</properties>

<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${db:username?:ut_user}"/>
</dataSource>
```

配置:设置 (settings)

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。下表描述了设置中各项设置的含义、默认值等。

设置名	描述	有效值	默认值
cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true false	true
lazyLoadingEnabled	延迟加载/懒加载 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
aggressiveLazyLoading	开启时，任一方法的调用都会加载该对象的所有延迟加载属性。否则，每个延迟加载属性会按需加载（参考 lazyLoadTriggerMethods）。	true false	false（在 3.4.1 及之前的版本中默认为 true）
multipleResultSetsEnabled	是否允许单个语句返回多结果集（需要数据库驱动支持）。	true false	true
useColumnLabel	使用列标签代替列名。实际表现依赖于数据库驱动，具体可参考数据库驱动的相关文档，或通过对比测试来观察。	true false	true
useGeneratedKeys	允许 JDBC 支持 自动生成主键 ，需要数据库驱动支持。 如果设置为 true，将强制使用自动生成主键 。尽管一些数据库驱动不支持此特性，但仍可正常工作（如 Derby）。	true false	False
autoMappingBehavior	指定 MyBatis 应 如何自动映射列到字段或属性 。NONE 表示关闭自动映射；PARTIAL 只会自动映射没有定义嵌套结果映射的字段。FULL 会自动映射任何复杂的结果集（无论是否嵌套）。	NONE, PARTIAL, FULL	PARTIAL
autoMappingUnknownColumnBehavior	指定发现自动映射目标未知列（或未知属性类型）的行为。 ： <ul style="list-style-type: none">NONE：不做任何反应WARNING：输出警告日志（'org.apache.ibatis.session.AutoMappingUnknownColumnB	NONE, WARNING, FAILING	NONE

	<p>ehavior' 的日志等级必须设置为 WARN)</p> <ul style="list-style-type: none">● FAILING: 映射失败 (抛出 SqlSessionException)		
defaultExecutorType	配置 默认的执行器 。SIMPLE 就是普通的执行器; REUSE 执行器会重用预处理语句 (PreparedStatement); BATCH 执行器不仅重用语句还会执行批量更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置 超时时间 , 它决定数据库驱动等待数据库响应的秒数。	任意正整数	未设置 (null)
defaultFetchSize	为驱动的结果集获取数量 (fetchSize) 设置一个建议值。此参数只可以在查询设置中被覆盖。	任意正整数	未设置 (null)
defaultResultSetType	指定语句默认的滚动策略。(新增于 3.5.2)	FORWARD_ONLY SCROLL_SENSITIVE SCROLL_INSENSITIVE DEFAULT (等同于未设置)	未设置 (null)
safeRowBoundsEnabled	是否允许在嵌套语句中使用分页 (RowBounds) 。如果允许使用则设置为 false。	true false	False
safeResultHandlerEnabled	是否允许在嵌套语句中使用结果处理器 (ResultHandler) 。如果允许使用则设置为 false。	true false	True
mapUnderscoreToCamelCase	是否开启驼峰命名自动映射, 即从经典数据库列名 A_COLUMN 映射到经典 Java 属性名 aColumn。	true false	False
localCacheScope	MyBatis 利用本地缓存机制 (Local Cache)防止循环引用和加速重复的嵌套查询。默认值为 SESSION, 会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT, 本地缓存将仅用于执行语句, 对相同 SqlSession 的不同查询将不会进行缓存。	SESSION STATEMENT	SESSION
jdbcTypeForNull	当没有为参数指定特定的 JDBC 类型时, 空值的默认 JDBC 类型。某些数据库驱动需要指定列的 JDBC 类型, 多数情况直接用一般类型即可, 比如 NULL、	JdbcType 常量, 常用值: NULL、VARCHAR 或 OTHER。	OTHER

	VARCHAR 或 OTHER。		
lazyLoadTriggerMethods	指定对象的哪些方法触发一次延迟加载。	用逗号分隔的方法列表。	equals,clone,hashCode,toString
defaultScriptingLanguage	指定动态 SQL 生成使用的默认脚本语言。	一个类型别名或全限定类名。	org.apache.ibatis.scripting.xmltags.XMLLanguageDriver
defaultEnumTypeHandler	指定 Enum 使用的默认 TypeHandler 。（新增于 3.4.5）	一个类型别名或全限定类名。	org.apache.ibatis.type.EnumTypeHandler
callSettersOnNulls	指定当结果集中值为 null 的时候是否调用映射对象的 setter（map 对象时为 put）方法，这在依赖于 Map.keySet() 或 null 值进行初始化时比较有用。注意基本类型（int、boolean 等）是不能设置成 null 的。	true false	false
returnInstanceForEmptyRow	当返回行的所有列都是空时，MyBatis 默认返回 null。当开启这个设置时，MyBatis 会返回一个空实例。请注意，它也适用于嵌套的结果集（如集合或关联）。（新增于 3.4.2）	true false	false
logPrefix	指定 MyBatis 增加到日志名称的前缀。	任何字符串	未设置
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J LOG4J LOG4J2 JDK_LOGGING COMMONS_LOGGING STDOUT_LOGGING NO_LOGGING	未设置
proxyFactory	指定 Mybatis 创建可延迟加载对象所用到的代理工具。	CGLIB JAVASSIST	JAVASSIST（MyBatis 3.3 以上）
vfsImpl	指定 VFS 的实现	自定义 VFS 的实现的全限定名，以逗号分隔。	未设置
useActualParamName	允许使用方法签名中的名称作为语句参数名称。为了使用该特性，你的项目必须采用 Java 8 编译，并且加上 -parameters 选项。（新增于 3.4.1）	true false	true
configurationFactory	指定一个提供 Configuration 实例的类。这个被返回的 Configuration 实	一个类型别名或完全限定类名。	未设置

	例用来加载被反序列化对象的延迟加载属性值。 这个类必须包含一个签名为 <code>static Configuration getConfiguration()</code> 的方法。（新增于 3.2.3）		
<code>shrinkWhitespacesInSql</code>	从 SQL 中删除多余的空格字符。请注意，这也会影响 SQL 中的文字字符串。（新增于 3.5.5）	<code>true false</code>	<code>false</code>
<code>defaultSqlProviderType</code>	Specifies an sql provider class that holds provider method (Since 3.5.6). This class apply to the type(or value) attribute on sql provider annotation(e.g. <code>@SelectProvider</code>), when these attribute was omitted.	A type alias or fully qualified class name	Not set

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
</settings>
```

配置: 类型别名 (typeAliases)

类型别名可为 Java 类型设置一个缩写名字。 它仅用于 XML 配置，意在降低冗余的全限定类名书写。例如：

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

在没有注解的情况下，会使用 **Bean** 的首字母小写的非限定类名来作为它的别名。比如 `domain.blog.Author` 的别名为 `author`；若有注解，则别名为其注解值。见下面的例子：

```
@Alias("author")
public class Author {
    ...
}
```

下面是一些为常见的 **Java** 类型内建的类型别名。它们都是不区分大小写的，注意，为了应对原始类型的命名重复，采取了特殊的命名风格。

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

基本类型的别名是在名称前加下划线。

其他类型基本都是转小写。

配置: 类型处理器 (typeHandlers)

MyBatis 在设置预处理语句 (PreparedStatement) 中的参数或从结果集中取出一个值时, 都会用类型处理器将获取到的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

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 SMALLINT
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 BIGINT
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
ClobReaderTypeHandler	java.io.Reader	-
ClobTypeHandler	java.lang.String	CLOB, LONGVARCHAR
NStringTypeHandler	java.lang.String	NVARCHAR, NCHAR

NClobTypeHandler	java.lang.String	NCLOB
BlobInputStreamTypeHandler	java.io.InputStream	-
ByteArrayTypeHandler	byte[]	数据库兼容的字节流类型
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	java.util.Date	TIMESTAMP
DateOnlyTypeHandler	java.util.Date	DATE
TimeOnlyTypeHandler	java.util.Date	TIME
SqlTimestampTypeHandler	java.sql.Timestamp	TIMESTAMP
SqlDateTypeHandler	java.sql.Date	DATE
SqlTimeTypeHandler	java.sql.Time	TIME
ObjectTypeHandler	Any	OTHER 或未指定类型
EnumTypeHandler	Enumeration Type	VARCHAR 或任何兼容的字符串类型, 用来存储枚举的名称 (而不是索引序数值)
EnumOrdinalTypeHandler	Enumeration Type	任何兼容的 NUMERIC 或 DOUBLE 类型, 用来存储枚举的序数值 (而不是名称)。
SqlxmlTypeHandler	java.lang.String	SQLXML
InstantTypeHandler	java.time.Instant	TIMESTAMP
LocalDateTimeTypeHandler	java.time.LocalDateTime	TIMESTAMP
LocalDateTypeHandler	java.time.LocalDate	DATE
LocalTimeTypeHandler	java.time.LocalTime	TIME
OffsetDateTimeTypeHandler	java.time.OffsetDateTime	TIMESTAMP
OffsetTimeTypeHandler	java.time.OffsetTime	TIME
ZonedDateTimeTypeHandler	java.time.ZonedDateTime	TIMESTAMP
YearTypeHandler	java.time.Year	INTEGER
MonthTypeHandler	java.time.Month	INTEGER

YearMonthTypeHandler	java.time.YearMonth	VARCHAR 或 LONGVARCHAR
JapaneseDateTypeHandler	java.time.chrono.JapaneseDate	DATE

你可以重写已有的类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。具体做法为：实现 `org.apache.ibatis.type.TypeHandler` 接口，或继承一个很便利的类 `org.apache.ibatis.type.BaseTypeHandler`，并且可以（可选地）将它映射到一个 JDBC 类型

```
// ExampleTypeHandler.java
@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String> {

    @Override
    public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType
jdbcType) throws SQLException {
        ps.setString(i, parameter);
    }

    @Override
    public String getNullableResult(ResultSet rs, String columnName) throws SQLException {
        return rs.getString(columnName);
    }

    @Override
    public String getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
        return rs.getString(columnIndex);
    }

    @Override
    public String getNullableResult(CallableStatement cs, int columnIndex) throws
SQLException {
        return cs.getString(columnIndex);
    }
}

<!-- mybatis-config.xml -->
<typeHandlers>
    <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

注意： **MyBatis** 不会通过检测数据库元信息来决定使用哪种类型，所以你必须**在参数和结果映射中指明字段是 VARCHAR 类型**，以使其能够绑定到正确的类型处理器上。这是因为 **MyBatis** 直到语句被执行时才清楚数据类型。

通过类型处理器的泛型，**MyBatis** 可以得知该类型处理器处理的 Java 类型，不过这种行为可以通过两种方法改变：

1. 在类型处理器的配置元素（`typeHandler` 元素）上增加一个 `javaType` 属性（比如：`javaType="String"`）；
2. 在类型处理器的类上增加一个 `@MappedTypes` 注解指定与其关联的 Java 类型列表。如果在 `javaType` 属性中也同时指定，则注解上的配置将被忽略。

可以通过两种方式来指定关联的 JDBC 类型：

1. 在类型处理器的配置元素上增加一个 `jdbcType` 属性（比如：`jdbcType="VARCHAR"`）；
2. 在类型处理器的类上增加一个 `@MappedJdbcTypes` 注解指定与其关联的 JDBC 类型列表。如果在 `jdbcType` 属性中也同时指定，则注解上的配置将被忽略。

```
<result property="uid"
        column="uid"
        javaType="java.lang.Integer"
        jdbcType="INTEGER"
        typeHandler="" /> |
```

`typeHandler` 不和上面的 `javaType` 和 `jdbcType` 共存。

当在 `ResultMap` 中决定使用哪种类型处理器时，此时 Java 类型是已知的（从结果类型中获得），但是 JDBC 类型是未知的。因此 Mybatis 使用 `javaType=[Java 类型]`，`jdbcType=null` 的组合来选择一个类型处理器。这意味着使用 `@MappedJdbcTypes` 注解可以限制类型处理器的作用范围，并且可以确保，除非显式地设置，否则类型处理器在 `ResultMap` 中将不会生效。如果希望能在 `ResultMap` 中隐式地使用类型处理器，那么设置 `@MappedJdbcTypes` 注解的 `includeNullJdbcType=true` 即可。然而从 Mybatis 3.4.0 开始，如果某个 Java 类型只有一个注册的类型处理器，即使没有设置 `includeNullJdbcType=true`，那么这个类型处理器也会是 `ResultMap` 使用 Java 类型时的默认处理器。

最后，可以让 MyBatis 帮你查找类型处理器：

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <package name="org.mybatis.example"/>
</typeHandlers>
```

注意在使用自动发现功能的时候，只能通过注解方式来指定 JDBC 的类型。

你可以创建能够处理多个类的泛型类型处理器。为了使用泛型类型处理器，需要增加一个接受该类的 `class` 作为参数的构造器，这样 MyBatis 会在构造一个类型处理器实例的时候传入一个具体的类。

```
//GenericTypeHandler.java
public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

    private Class<E> type;

    public GenericTypeHandler(Class<E> type) {
```

```
    if (type == null) throw new IllegalArgumentException("Type argument cannot be null");
    this.type = type;
}
...
```

EnumTypeHandler 和 EnumOrdinalTypeHandler 都是泛型类型处理器

配置:处理枚举类型

没看

配置:对象工厂 (objectFactory)

每次 MyBatis 创建结果对象的新实例时,它都会使用一个对象工厂 (**ObjectFactory**) 实例来完成实例化工作。默认的对象工厂需要做的仅仅是实例化目标类,要么通过默认无参构造方法,要么通过存在的参数映射来调用带有参数的构造方法。如果想覆盖对象工厂的默认行为,可以通过创建自己的对象工厂来实现。比如:

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
    public Object create(Class type) {
        return super.create(type);
    }
    public Object create(Class type, List<Class> constructorArgTypes, List<Object>
constructorArgs) {
        return super.create(type, constructorArgTypes, constructorArgs);
    }
    public void setProperties(Properties properties) {
        super.setProperties(properties);
    }
    public <T> boolean isCollection(Class<T> type) {
        return Collection.class.isAssignableFrom(type);
    }
}
```

```
<!-- mybatis-config.xml -->
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
    <property name="someProperty" value="100"/>
</objectFactory>
```

ObjectFactory 接口很简单,它包含两个创建实例用的方法,一个是处理默认无参构造方法的,另外一个处理带参数的构造方法的。另外, setProperties 方法可以被用来配置 ObjectFactory,在初始化你的 ObjectFactory 实例后, objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

配置:插件 (plugins)

MyBatis 允许你在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:

1. **Executor** (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
2. **ParameterHandler** (getParameterObject, setParameters)
3. **ResultSetHandler** (handleResultSets, handleOutputParameters)
4. **StatementHandler** (prepare, parameterize, batch, update, query)

通过 MyBatis 提供的强大机制,使用插件是非常简单的,只需实现 **Interceptor** 接口,并指定想要拦截的方法签名即可。

```
// ExamplePlugin.java
@Intercepts({@Signature(
    type= Executor.class,
    method = "update",
    args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
    private Properties properties = new Properties();
    public Object intercept(Invocation invocation) throws Throwable {
        // implement pre processing if need
        Object returnObject = invocation.proceed();
        // implement post processing if need
        return returnObject;
    }
    public void setProperties(Properties properties) {
        this.properties = properties;
    }
}

<!-- mybatis-config.xml -->
<plugins>
    <plugin interceptor="org.mybatis.example.ExamplePlugin">
        <property name="someProperty" value="100"/>
    </plugin>
</plugins>
```

除了用插件来修改 MyBatis 核心行为以外,还可以通过完全覆盖配置类来达到目的。只需继承配置类后覆盖其中的某个方法,再把它传递到 `SqlSessionFactoryBuilder.build(myConfig)` 方法即可。再次重申,这可能会极大影响 MyBatis 的行为,务请慎之又慎。

配置:环境配置 (environments)

MyBatis 可以配置成**适应多种环境**,这种机制有助于将 SQL 映射应用于**多种数据库**之中,现实情况下有多种理由需要这么做。例如,**开发、测试和生产环境需要有不同的配置**;或者想在具有相同 Schema 的多个生产数据库中使用相同的 SQL 映射。还有许多类似的使用场景。

尽管可以配置多个环境，但每个 **SqlSessionFactory** 实例只能选择一种环境

所以，如果你想连接两个数据库，就需要创建两个 **SqlSessionFactory** 实例，每个数据库对应一个。而如果是三个数据库，就需要三个实例，依此类推，记起来很简单：
每个数据库对应一个 SqlSessionFactory 实例

为了指定创建哪种环境，只要将它作为可选的参数传递给 **SqlSessionFactoryBuilder** 即可。可以接受环境配置的两个方法签名是：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment);  
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment,  
properties);
```

如果忽略了环境参数，那么将会加载默认环境，如下所示：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader);  
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, properties);
```

environments 元素定义了如何配置环境。

```
<environments default="development">  
  <environment id="development">  
    <transactionManager type="JDBC">  
      <property name="..." value="..."/>  
    </transactionManager>  
    <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>
```

一些关键点：

1. 默认使用的环境 ID（比如：default="development"），就是在下面的多个 **environment** 的 ID 中选择一个，表示当前使用的 environment。
2. 每个 **environment** 元素定义的环境 ID（比如：id="development"）。
3. 事务管理器的配置（比如：type="JDBC"）。
4. 数据源的配置（比如：type="POOLED"）。

事务管理器（**transactionManager**）

在 **MyBatis** 中有两种类型的事务管理器（也就是 type="[JDBC|MANAGED]"）：

1. **JDBC** - 这个配置直接使用了 **JDBC** 的提交和回滚设施，它依赖从数据源获得的连接来管理事务作用域。
2. **MANAGED** - 这个配置几乎没做什么。它从不提交或回滚一个连接，而是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）。默认情况下它会关闭连接。然而一些容器并不希望连接被关闭，因此需要将 **closeConnection** 属性设置为 **false** 来阻止默认的关闭行为。例如：

```
<transactionManager type="MANAGED">  
  <property name="closeConnection" value="false"/>  
</transactionManager>
```


提示:如果你正在使用 **Spring + MyBatis**,则没有必要配置事务管理器,因为 **Spring** 模块会使用自带的管理器来覆盖前面的配置。

这两种事务管理器类型都不需要设置任何属性。它们其实是类型别名,换句话说,你可以用 **TransactionFactory** 接口实现类的全限定名或类型别名代替它们。

```
public interface TransactionFactory {
    default void setProperties(Properties props) { // 从 3.5.2 开始,该方法为默认方法
        // 空实现
    }
    Transaction newTransaction(Connection conn);
    Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level,
        boolean autoCommit);
}
```

在事务管理器实例化后,所有在 **XML** 中配置的属性将会被传递给 **setProperties()** 方法。你的实现还需要创建一个 **Transaction** 接口的实现类,这个接口也很简单:

```
public interface Transaction {
    Connection getConnection() throws SQLException;
    void commit() throws SQLException;
    void rollback() throws SQLException;
    void close() throws SQLException;
    Integer getTimeout() throws SQLException;
}
```

使用这两个接口,你可以完全自定义 **MyBatis** 对事务的处理。

数据源 (dataSource)

dataSource 元素使用标准的 **JDBC** 数据源接口来配置 **JDBC** 连接对象的资源。

大多数 **MyBatis** 应用程序会按示例中的例子来配置数据源。虽然数据源配置是可选的,但如果要启用延迟加载特性,就必须配置数据源。

有三种内建的数据源类型 (也就是 **type="[UNPOOLED|POOLED|JNDI]"**):

1. **UNPOOLED** - 不使用连接池 这个数据源的实现会每次请求时打开和关闭连接。虽然有点慢,但对那些数据库连接可用性要求不高的简单应用程序来说,是一个很好的选择。性能表现则依赖于使用的数据库,对某些数据库来说,使用连接池并不重要,这个配置就很适合这种情形。**UNPOOLED** 类型的数据源仅仅需要配置以下 5 种属性:
 - a) **driver** - 这是 **JDBC** 驱动的 **Java** 类全限定名 (并不是 **JDBC** 驱动中可能包含的数据源类)。
 - b) **url** - 这是数据库的 **JDBC URL** 地址:
`mysql:localhost:3306/dbname?key=value;`
 - c) **username** - 登录数据库的用户名。
 - d) **password** - 登录数据库的密码。
 - e) **defaultTransactionIsolationLevel** - 默认的连接**事务隔离级别**。
 - f) **defaultNetworkTimeout** - 等待数据库操作完成的默认**网络超时时间** (单位:

毫秒)。查看 `java.sql.Connection#setNetworkTimeout()` 的 API 文档以获取更多信息。

2. POOLED: 使用连接池

除了上述属性外，还有：

- ◆ **poolMaximumActiveConnections** - 在任意时间可存在的活动（正在使用）连接数量，默认值：10
- ◆ **poolMaximumIdleConnections** - 任意时间可能存在的空闲连接数。
- ◆ **poolMaximumCheckoutTime** - 在被强制返回之前，池中连接被检出（checked out）时间，默认值：20000 毫秒（即 20 秒）
- ◆ **poolTimeToWait** - 这是一个底层设置，如果获取连接花费了相当长的时间，连接池会打印状态日志并重新尝试获取一个连接（避免在误配置的情况下一直失败且不打印日志），默认值：20000 毫秒（即 20 秒）。
- ◆ **poolMaximumLocalBadConnectionTolerance** - 这是一个关于坏连接容忍度的底层设置，作用于每一个尝试从缓存池获取连接的线程。如果这个线程获取到的是一个坏的连接，那么这个数据源允许这个线程尝试重新获取一个新的连接，但是这个重新尝试的次数不应该超过 **poolMaximumIdleConnections** 与 **poolMaximumLocalBadConnectionTolerance** 之和。默认值：3（新增于 3.4.5）
- ◆ **poolPingQuery** - 发送到数据库的探测查询，用来检验连接是否正常工作并准备接受请求。默认是“NO PING QUERY SET”，这会导致多数数据库驱动出错时返回恰当的错误消息。
- ◆ **poolPingEnabled** - 是否启用探测查询。若开启，需要设置 **poolPingQuery** 属性为一个可执行的 SQL 语句（最好是一个速度非常快的 SQL 语句），默认值：false。
- **poolPingConnectionsNotUsedFor** - 配置 **poolPingQuery** 的频率。可以被设置为和数据库连接超时时间一样，来避免不必要的探测，默认值：0（即所有连接每一时刻都被探测 — 当然仅当 **poolPingEnabled** 为 true 时适用）。

3. JNDI:

这个数据源实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部分布配置数据源，然后放置一个 JNDI 上下文的数据源引用。这种数据源配置只需要两个属性：

- a) **initial_context** - 这个属性用来在 **InitialContext** 中寻找上下文（即，**initialContext.lookup(initial_context)**）。这是个可选属性，如果忽略，那么将会直接从 **InitialContext** 中寻找 **data_source** 属性。
- b) **data_source** - 这是引用数据源实例位置的上下文路径。提供了 **initial_context** 配置时会在其返回的上下文中进行查找，没有提供时则直接在 **InitialContext** 中查找。

你可以通过实现接口 `org.apache.ibatis.datasource.DataSourceFactory` 来使

用第三方数据源实现：

```
public interface DataSourceFactory {  
    void setProperties(Properties props);  
    DataSource getDataSource();  
}
```

`org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory` 可被用作父类来构建新的数据源适配器，比如下面这段插入 `C3P0` 数据源所必需的代码：

```
import org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;  
import com.mchange.v2.c3p0.ComboPooledDataSource;  
  
public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {  
    public C3P0DataSourceFactory() {  
        this.dataSource = new ComboPooledDataSource();  
    }  
}
```

配置：数据库厂商标识（`databaseIdProvider`）

`MyBatis` 可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的 `databaseId` 属性。`MyBatis` 会加载带有匹配当前数据库 `databaseId` 属性和所有不带 `databaseId` 属性的语句。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。为支持多厂商特性，只要像下面这样在 `mybatis-config.xml` 文件中加入 `databaseIdProvider` 即可：

```
<databaseIdProvider type="DB_VENDOR" />
```

`databaseIdProvider` 对应的 `DB_VENDOR` 实现会将 `databaseId` 设置为 `DatabaseMetaData#getDatabaseProductName()` 返回的字符串。由于通常情况下这些字符串都非常长，而且相同产品的不同版本会返回不同的值，你可能想通过设置属性别名来使其变短：

```
<databaseIdProvider type="DB_VENDOR">  
    <property name="SQL Server" value="sqlserver"/>  
    <property name="DB2" value="db2"/>  
    <property name="Oracle" value="oracle" />  
</databaseIdProvider>
```

配置：映射器（`mappers`）

既然 `MyBatis` 的行为已经由上述元素配置完了，我们现在就要来定义 `SQL` 映射语句了。但首先，我们需要告诉 `MyBatis` 到哪里去找到这些 `SQL` 语句。在自动查找资源方面，`Java` 并没有提供一个很好的解决方案，所以最好的办法是直接告诉 `MyBatis` 到哪里去找映射文件。

```
<mappers>  
    <!-- 使用相对于类路径的资源引用 -->
```

```
<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

<!-- 使用完全限定资源定位符 (URL) -->
<mapper url="file:///var/mappers/AuthorMapper.xml"/>

<!-- 使用映射器接口实现类的完全限定类名 -->
<mapper class="org.mybatis.builder.AuthorMapper"/>

<!-- 将包内的映射器接口实现全部注册为映射器 -->
<package name="org.mybatis.builder"/>
</mappers>
```

XML 映射文件(Mapper)

MyBatis 的真正强大在于它的语句映射，这是它的魔力所在

SQL 映射文件只有很少的几个顶级元素（按照应被定义的顺序列出）：

1. **cache** - 该命名空间的缓存配置。
2. **cache-ref** - 引用其它命名空间的缓存配置。
3. **resultMap** - 描述如何从数据库结果集中加载对象，是最复杂也是最强大的元素。
4. **parameterMap** - 老式风格的参数映射。此元素 已被废弃，并可能在将来被移除！
请使用行内参数映射。文档中不会介绍此元素。
5. **sql** - 可被其它语句引用的可重用语句块。
6. **insert** - 映射插入语句。
7. **update** - 映射更新语句。
8. **delete** - 映射删除语句。
9. **select** - 映射查询语句。

Mapper:select

MyBatis 在查询和结果映射做了相当多的改进。

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
    SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

#{id}

#{}告诉 MyBatis 创建一个**预处理语句 (PreparedStatement)** 参数，在 JDBC 中，这样的参数在 SQL 中会由一个“?”来标识，并被传递到一个新的预处理语句中

Select 属性：

```
<select
    id="selectPerson"
    parameterType="int"
    parameterMap="deprecated"
    resultType="hashmap"
    resultMap="personResultMap">
```

```
flushCache="false"
useCache="true"
timeout="10"
fetchSize="256"
statementType="PREPARED"
resultSetType="FORWARD_ONLY">
```

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过类型处理器 (TypeHandler) 推断出具体传入语句的参数，默认值为未设置 (unset)。
parameterMap (废弃)	用于引用外部 parameterMap 的属性， 目前已被废弃 。请使用行内参数映射和 parameterType 属性。
resultType	期望从这条语句中返回结果的类全限定名或别名。 注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。 resultType 和 resultMap 之间只能同时使用一个。
resultMap	对外部 resultMap 的命名引用。结果映射是 MyBatis 最强大的特性，如果你对其理解透彻，许多复杂的映射问题都能迎刃而解。 resultType 和 resultMap 之间只能同时使用一个。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：false。
useCache	将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来，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。默认值为未设置 (unset) (依赖数据库驱动)。
fetchSize	这是一个给驱动的建议值，尝试让驱动程序每次批量返回的结果行数等于这个设置值。 默认值为未设置 (unset) (依赖驱动)。
statementType	可选 STATEMENT, PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement, PreparedStatement 或 CallableStatement, 默认值：PREPARED。
resultSetType	FORWARD_ONLY, SCROLL_SENSITIVE, SCROLL_INSENSITIVE 或 DEFAULT (等价于 unset) 中的一个，默认值为 unset (依赖数据库驱动)。
databaseId	如果配置了数据库厂商标识 (databaseIdProvider), MyBatis 会加载所有不带 databaseId 或匹配当前 databaseId 的语句；如果带和不带的语句都有，则不带的会被忽略。
resultOrdered	这个设置仅针对嵌套结果 select 语句：如果为 true，将会假设包含了嵌套结果集或是分组，当返回一个主结果行时，就不会产生对前面结果集的引用。 这就使得在获取嵌套结果集的时候不至

	于内存不够用。默认值： false 。
resultSets	这个设置仅适用于多结果集的情况。它将列出语句执行后返回的结果集并赋予每个结果集一个名称，多个名称之间以逗号分隔。

Mapper: insert, update, delete

和 **Select** 类似：

```
<insert
  id="insertAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="deleteAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

属性	描述
useGeneratedKeys	（仅适用于 insert 和 update ）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 这样的关系型数据库管理系统的自动递增字段），默认值： false 。
keyProperty	（仅适用于 insert 和 update ）指定能够唯一识别对象的属性， MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（unset）。如果生成列不止一个，可以用逗号分隔多个属性名称。
keyColumn	（仅适用于 insert 和 update ）设置生成键值在表中的列名，在某些数据库（像 PostgreSQL ）中，当主键列不是表中的第一列的时候，是必须设置的。如果生成列不止一个，可以用逗号分隔多个属性名称。

如果你的数据库支持自动生成主键的字段（比如 **MySQL** 和 **SQL Server**），那么你可以设置 **useGeneratedKeys=" true"**，然后再把 **keyProperty** 设置为目标属性就 OK 了

useGeneratedKeys 和 **keyProperty** 可以配合使用。

Eg:

```
<insert id="insertAuthor" useGeneratedKeys="true" keyProperty="id">
  insert into Author (username, password, email, bio) values
  <foreach item="item" collection="list" separator=",">
    ({item.username}, {item.password}, {item.email}, {item.bio})
  </foreach>
</insert>

<update id="updateAuthor">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor">
  delete from Author where id = #{id}
</delete>
```

对于不支持自动生成主键列的数据库和可能不支持自动生成主键的 **JDBC** 驱动，**MyBatis** 有另外一种方法来生成主键。

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
    (id, username, password, email, bio, favourite_section)
  values
    ({id}, {username}, {password}, {email}, {bio},
    #{favouriteSection,jdbcType=VARCHAR})
</insert>

<selectKey
  keyProperty="id"
  resultType="int"
  order="BEFORE"
  statementType="PREPARED">
```

Mapper: sql

这个元素可以用来定义可重用的 **SQL** 代码片段，以便在其它语句中使用。参数可以静态地（在加载的时候）确定下来，并且可以在不同的 **include** 元素中定义不同的参数值。

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>
```

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t1"/></include>,
    <include refid="userColumns"><property name="alias" value="t2"/></include>
  from some_table t1
    cross join some_table t2
</select>
```


也可以在 `include` 元素的 `refid` 属性或内部语句中使用属性值，例如：

```
<sql id="sometable">
  ${prefix}Table
</sql>

<sql id="someinclude">
  from
    <include refid="${include_target}"/>
</sql>

<select id="select" resultType="map">
  select
    field1, field2, field3
    <include refid="someinclude">
      <property name="prefix" value="Some"/>
      <property name="include_target" value="sometable"/>
    </include>
  </select>
```

Mappwe: 参数

提示：JDBC 要求，如果一个列允许使用 `null` 值，并且会使用值为 `null` 的参数，就必须指定 JDBC 类型 (`jdbcType`)。阅读 `PreparedStatement.setNull()` 的 JavaDoc 来获取更多信息。

对于数值类型，还可以设置 `numericScale` 指定小数点后保留的位数。

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

#{} vs. \${}

`#{}:` `PreparedStatement`: 快速安全

`${}:` 普通 `Statement`，字符串替换，不安全：SQL 注入攻击

字符串替换的好处举例：

比如 `findBy`，如果有很多列属性，那就需要定义很多 `findBy`

但是实际不需要：

```
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String value);
```

注意：

1. `$(column)` 是直接替换，而真正的值是 `#{}:`

2. `$()` 还是不安全：用这种方式接受用户的输入，并用作语句参数是不安全的，会导致潜在的 SQL 注入攻击。因此，要么不允许用户输入这些字段，要么自行转义并检验这些参数。

Mapper: 结果映射

```
<select id="selectUsers" resultType="com.someapp.model.User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

在这些情况下, MyBatis 会在幕后自动创建一个 ResultMap, 再根据属性名来映射列到 JavaBean 的属性上。如果列名和属性名不能匹配上, 可以在 SELECT 语句中设置列别名

一般不需要(简单的类, 没有复杂嵌套)显式配置 resultMap

1. resultMap: 显式引用, 不建议
2. resultType: 直接使用全限定名或者别名。简单

Mapper: 高级映射

数据库不可能永远是你所想或所需的那个样子。我们希望每个数据库都具备良好的第三范式或 BCNF 范式, 可惜它们并不都是那样。

```
<!-- 非常复杂的语句 -->
<select id="selectBlogDetails" resultMap="detailedBlogResultMap">
  select
    B.id as blog_id,
    B.title as blog_title,
    B.author_id as blog_author_id,
    A.id as author_id,
    A.username as author_username,
    A.password as author_password,
    A.email as author_email,
    A.bio as author_bio,
    A.favourite_section as author_favourite_section,
    P.id as post_id,
    P.blog_id as post_blog_id,
    P.author_id as post_author_id,
    P.created_on as post_created_on,
    P.section as post_section,
    P.subject as post_subject,
    P.draft as draft,
    P.body as post_body,
    C.id as comment_id,
    C.post_id as comment_post_id,
    C.name as comment_name,
    C.comment as comment_text,
    T.id as tag_id,
    T.name as tag_name
  from Blog B
    left outer join Author A on B.author_id = A.id
    left outer join Post P on B.id = P.blog_id
    left outer join Comment C on P.id = C.post_id
    left outer join Post_Tag PT on PT.post_id = P.id
    left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

```

<!-- 非常复杂的结果映射 -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" javaType="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" javaType="Author"/>
    <collection property="comments" ofType="Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" ofType="Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>

```

结果映射 (resultMap)

- **constructor** - 用于在实例化类时，注入结果到构造方法中
 - **idArg** - ID 参数；标记出作为 ID 的结果可以帮助提高整体性能
 - **arg** - 将被注入到构造方法的一个普通结果
- **id** - 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能
- **result** - 注入到字段或 **JavaBean** 属性的普通结果
- **association** - 一个复杂类型的关联；许多结果将包装成这种类型
 - **嵌套结果映射** - 关联可以是 **resultMap** 元素，或是对其它结果映射的引用
- **collection** - 一个复杂类型的集合
 - **嵌套结果映射** - 集合可以是 **resultMap** 元素，或是对其它结果映射的引用
- **discriminator** - 使用结果值来决定使用哪个 **resultMap**
 - **case** - 基于某些值的结果映射
 - **嵌套结果映射** - **case** 也是一个结果映射，因此具有相同的结构和元素；或者引用其它的结果映射

ResultMap 的属性列表

属性	描述
id	当前命名空间中的一个唯一标识，用于标识一个结果映射。

type	类的完全限定名，或者一个类型别名（关于内置的类型别名，可以参考上面的表格）。
autoMapping	如果设置这个属性，MyBatis 将会为本结果映射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：未设置（unset）。

id & result

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

这些元素是结果映射的基础。id 和 result 元素都将一个列的值映射到一个简单数据类型（String, int, double, Date 等）的属性或字段。

这两者之间的唯一不同是，id 元素对应的属性会被标记为对象的标识符，在比较对象实例时使用。这样可以提高整体的性能，尤其是进行缓存和嵌套结果映射（也就是连接映射）的时候。

Mapper:支持的 JDBC 类型

MyBatis 通过内置的 jdbcType 枚举类型支持下面的 JDBC 类型：

BIT	FLOAT	CHAR	TIMESTAMP	OTHER	UNDEFINED
TINYINT	REAL	VARCHAR	BINARY	BLOB	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	ARRAY

Mapper：构造方法

通过 setter，可以满足大多数的数据传输对象（Data Transfer Object，**DTO**）以及绝大部分领域模型 **Domain** 的要求。但有些情况下你想使用不可变类。一般来说，很少改变或基本不变的包含引用或数据的表，很适合使用不可变类。构造方法注入允许你在初始化时为类设置属性的值，而不用暴露出公有方法。

```
public User(Integer id, String username, int age) {
    //...
}

<constructor>
  <idArg column="id" javaType="int"/>
  <arg column="username" javaType="String"/>
```

```
<arg column="age" javaType="_int"/>
</constructor>
```

Mapper：关联

```
<association property="author" column="blog_author_id" javaType="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
</association>
```

关联（association）元素处理“有一个”类型的关系。比如，在我们的示例中，一个博客有一个用户。关联结果映射和其它类型的映射工作方式差不多。你需要指定目标属性名以及属性的 `javaType`（很多时候 `MyBatis` 可以自己推断出来），在必要的情况下你还可以设置 `JDBC` 类型，如果你想覆盖获取结果值的过程，还可以设置类型处理器。

关联的不同之处是，你需要告诉 `MyBatis` 如何加载关联。`MyBatis` 有两种不同的方式加载关联：

1. 嵌套 `Select` 查询：通过执行另外一个 `SQL` 映射语句来加载期望的复杂类型。
2. 嵌套结果映射：使用嵌套的结果映射来处理连接结果的重复子集。

关联的嵌套 `Select` 查询：

属性	描述
<code>column</code>	数据库中的列名，或者是列的别名。一般情况下，这和传递给 <code>resultSet.getString(columnName)</code> 方法的参数一样。注意：在使用复合主键的时候，你可以使用 <code>column="{prop1=col1,prop2=col2}"</code> 这样的语法来指定多个传递给嵌套 <code>Select</code> 查询语句的列名。这会使得 <code>prop1</code> 和 <code>prop2</code> 作为参数对象，被设置为对应嵌套 <code>Select</code> 语句的参数。
<code>select</code>	用于加载复杂类型属性的映射语句的 <code>ID</code> ，它会从 <code>column</code> 属性指定的列中检索数据，作为参数传递给目标 <code>select</code> 语句。具体请参考下面的例子。注意：在使用复合主键的时候，你可以使用 <code>column="{prop1=col1,prop2=col2}"</code> 这样的语法来指定多个传递给嵌套 <code>Select</code> 查询语句的列名。这会使得 <code>prop1</code> 和 <code>prop2</code> 作为参数对象，被设置为对应嵌套 <code>Select</code> 语句的参数。
<code>fetchType</code>	可选的。有效值为 <code>lazy</code> 和 <code>eager</code> 。指定属性后，将在映射中忽略全局配置参数 <code>lazyLoadingEnabled</code> ，使用属性的值。

示例：

```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author"
    select="selectAuthor"/>
</resultMap>
```

```

</resultMap>

<select id="selectBlog" resultMap="blogResult">
    SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectAuthor" resultType="Author">
    SELECT * FROM AUTHOR WHERE ID = #{id}
</select>

```

-----没看完，暂时用不到-----

Mapper：集合

对于简单的 Select 集合：

```

<!-- 简单集合映射 -->
<select id="findAllUser" resultType="cn.edw.mybatislearning.domain.User">
    select * from USER
</select>

```

```

<collection property="posts" ofType="domain.blog.Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
</collection>

```

集合的嵌套 Select 查询：

```

<resultMap id="blogResult" type="Blog">
    <collection property="posts" javaType="ArrayList" column="id" ofType="Post"
    select="selectPostsForBlog"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
    SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Post">
    SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>

```

1. **property** 用于指定在 Java 实体类是保存集合关系的属性名称
2. **JavaType** 用于指定在 Java 实体类中使用什么类型来保存集合数据，多数情况下这个属性可以省略的。
3. **column** 用于指定数据表中的外键字段名称。
4. **ofType** 用于指定集合中包含的类型。
5. **select** 用于指定查询语句。

property：映射到列结果的字段或属性。如果用来匹配的 **JavaBean** 存在给定名字的属性，那么它将会被使用。否则 **MyBatis** 将会寻找给定名称的字段。无论是哪一种情形，你都可以使用通常的点式分隔形式进行复杂属性导航。比如，你可以这样映

射一些简单的东西：“username”，或者映射到一些复杂的东西上：“address.street.number”。

集合的嵌套结果映射：

集合的多结果集（ResultSet）：

Mapper：鉴别器

```
<discriminator javaType="int" column="draft">
  <case value="1" resultType="DraftPost"/>
</discriminator>
```

Mapper：自动映射

在简单的场景下，MyBatis 可以为你自动映射查询结果。但如果遇到复杂的场景，你需要构建一个结果映射。

你可以混合使用这两种策略。让我们深入了解一下自动映射是怎样工作的。

当自动映射查询结果时，MyBatis 会获取结果中返回的列名并在 Java 类中查找相同名字的属性（忽略大小写）。这意味着如果发现了 ID 列和 id 属性，MyBatis 会将列 ID 的值赋给 id 属性。

通常数据库列使用大写字母组成的单词命名，单词间用下划线分隔；而 Java 属性一般遵循驼峰命名法约定。为了在这两种命名方式之间启用自动映射，需要将 mapUnderscoreToCamelCase 设置为 true。

有三种自动映射等级：

1. **NONE** - 禁用自动映射。仅对手动映射的属性进行映射。
2. **PARTIAL** - 对除在内部定义了嵌套结果映射（也就是连接的属性）以外的属性进行映射
3. **FULL** - 自动映射所有属性。

默认值是 **PARTIAL**，这是有原因的。当对连接查询的结果使用 **FULL** 时，连接查询会在同一行中获取多个不同实体的数据，因此可能导致非预期的映射。

Mapper：缓存

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。为

了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进。

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

```
<cache/>
```

这个简单语句的效果如下：

1. 映射语句文件中的所有 select 语句的结果将会被缓存。
2. 映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
3. 缓存会使用最近最少使用算法（LRU，Least Recently Used）算法来清除不需要的缓存。
4. 缓存不会定时进行刷新（也就是说，没有刷新间隔）。
5. 缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
6. 缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。

提示：缓存只作用于 **cache** 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 `@CacheNamespaceRef` 注解指定缓存作用域。

这些属性可以通过 **cache** 元素的属性来修改。比如：

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

1. **flushInterval**（刷新间隔）属性可以被设置为任意的正整数，设置的值应该是一个以毫秒为单位的合理时间量。默认情况是不设置，也就是没有刷新间隔，缓存仅仅会在调用语句时刷新。
2. **size**（引用数目）属性可以被设置为任意正整数，要注意欲缓存对象的大小和运行环境中可用的内存资源。默认值是 **1024**。
3. **readOnly**（只读）属性可以被设置为 **true** 或 **false**。只读的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。这就提供了可观的性能提升。而可读写的缓存会（通过序列化）返回缓存对象的拷贝。速度上会慢一些，但是更安全，因此默认值是 **false**。

这个更高级的配置创建了一个 **FIFO** 缓存，每隔 **60** 秒刷新，最多可以存储结果对象或列表的 **512** 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

可用的清除策略有：

1. **LRU** - 最近最少使用：移除最长时间不被使用的对象。
2. **FIFO** - 先进先出：按对象进入缓存的顺序来移除它们。

- 3. **SOFT** - 软引用：基于垃圾回收器状态和软引用规则移除对象。
- 4. **WEAK** - 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。

使用自定义缓存

除了上述自定义缓存的方式，你也可以通过实现你自己的缓存，或为其他第三方缓存方案创建适配器，来完全覆盖缓存行为。

```
<cache type="com.domain.something.MyCustomCache"/>
```

type 属性指定的类必须实现 `org.apache.ibatis.cache.Cache` 接口，且提供一个接受 `String` 参数作为 `id` 的构造器。这个接口是 `MyBatis` 框架中许多复杂的接口之一，但是行为却非常简单。

```
public interface Cache {  
    String getId();  
    int getSize();  
    void putObject(Object key, Object value);  
    Object getObject(Object key);  
    boolean hasKey(Object key);  
    Object removeObject(Object key);  
    void clear();  
}
```

动态 SQL

Mybatis 的强大特性

- 1. `if`
- 2. `choose (when, otherwise)`
- 3. `trim (where, set)`
- 4. `foreach`

动态 SQL: `if`

使用动态 SQL 最常见情景是根据条件包含 `where` 子句的一部分。比如：

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

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

```
<if test="author != null and author.name != null">
  AND author_name like #{author.name}
</if>
</select>
```

动态 SQL: choose、when、otherwise

Choose: 从多个条件中选择一个使用, 类似 java 的 switch

集合 when 和 otherwise

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

Choose:

When:

When:

...

Othercase:

Switch:

Case:

Case:

...

Default:

动态 SQL: trim、where、set

前面的 if 存在问题:

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

Where 后面可以没有符合的条件, 导致 SQL 语法错误。

所以: 不要使用 where 字符串, 使用 **where** 标签

```

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

```

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，若子句的开头为 “AND” 或 “OR”，where 元素也会将它们去除。

如果 where 元素与你期望的不太一样，你也可以通过自定义 trim 元素来定制 where 元素的功能。比如，和 where 元素等价的自定义 trim 元素为：

```

<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>

```

prefixOverrides 属性会忽略通过管道符分隔的文本序列（注意此例中的空格是必要的）。上述例子会移除所有 prefixOverrides 属性中指定的内容，并且插入 prefix 属性中指定的内容。

用于动态更新语句的类似解决方案叫做 set。set 元素可以用于动态包含需要更新的列，忽略其它不更新的列：

```

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

```

这个例子中，set 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号

与 set 元素等价的自定义 trim 元素：

```

<trim prefix="SET" suffixOverrides=",">
  ...
</trim>

```

动态 SQL：foreach

Foreach：对集合进行遍历（尤其是在构建 IN 条件语句的时候

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

foreach 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的**集合项 (item)** 和**索引 (index)** 变量。它也允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。这个元素也不会错误地添加多余的分隔符

你可以将任何可迭代对象（如 **List**、**Set** 等）、**Map** 对象或者数组对象作为集合参数传递给 **foreach**。

1. 当使用可迭代对象或者数组时，**index** 是当前迭代的序号，**item** 的值是本次迭代获取到的元素。
2. 当使用 **Map** 对象（或者 **Map.Entry** 对象的集合）时，**index** 是键，**item** 是值。

动态 SQL: script

要在带注解的映射器接口类中使用动态 SQL，可以使用 **script** 元素

```
@Update({"<script>",
  "update Author",
  "  <set>",
  "    <if test='username != null'>username=#{username},</if>",
  "    <if test='password != null'>password=#{password},</if>",
  "    <if test='email != null'>email=#{email},</if>",
  "    <if test='bio != null'>bio=#{bio}</if>",
  "  </set>",
  "where id=#{id}",
  "</script>"})
void updateAuthorValues(Author author);
```

动态 SQL: bind

bind 元素允许你在 **OGNL** 表达式以外创建一个变量，并将其绑定到当前的上下文。

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value="'%' + _parameter.getTitle() + '%" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

动态 SQL: 多数据库支持

如果配置了 `databaseIdProvider`，你就可以在动态代码中使用名为 “`_databaseId`” 的变量来为不同的数据库构建特定的语句

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1
    </if>
  </selectKey>
  insert into users values ({id}, {name})
</insert>
```

Java API

MyBatis 的 Java API

Java API: SqlSession

使用 MyBatis 的主要 Java 接口就是 `SqlSession`。你可以通过这个接口来执行命令，获取映射器示例和管理事务

`SqlSessionFactoryBuilder` → `SqlSessionFactory` → `SqlSession`

(当 Mybatis 与一些依赖注入框架（如 Spring 或者 Guice）搭配使用时，`SqlSession` 将被依赖注入框架创建并注入，所以你不需要使用 `SqlSessionFactoryBuilder` 或者 `SqlSessionFactory`)

SqlSessionFactoryBuilder

`SqlSessionFactoryBuilder` 有五个 `build()` 方法，每一种都允许你从不同的资源中创建一个 `SqlSessionFactory` 实例。

1. `SqlSessionFactory build(InputStream inputStream)`
2. `SqlSessionFactory build(InputStream inputStream, String environment)`
3. `SqlSessionFactory build(InputStream inputStream, Properties properties)`
4. `SqlSessionFactory build(InputStream inputStream, String env, Properties props)`
5. `SqlSessionFactory build(Configuration config)`

第一种方法是最常用的，它接受一个指向 XML 文件（也就是之前讨论的 `mybatis-config.xml` 文件）的 `InputStream` 实例。可选的参数是 `environment` 和 `properties`。`environment` 决定加载哪种环境，包括数据源和事务管理器。

SqlSessionFactory

`SqlSessionFactory` 有多个方法创建 `SqlSession` 实例。

1. `SqlSession openSession()`

2. `SqlSession openSession(boolean autoCommit)`
3. `SqlSession openSession(Connection connection)`
4. `SqlSession openSession(TransactionIsolationLevel level)`
5. `SqlSession openSession(ExecutorType execType, TransactionIsolationLevel level)`
6. `SqlSession openSession(ExecutorType execType)`
7. `SqlSession openSession(ExecutorType execType, boolean autoCommit)`
8. `SqlSession openSession(ExecutorType execType, Connection connection)`

通常来说，当你选择其中一个方法时，你需要考虑以下几点：

1. **事务处理**：你希望在 `session` 作用域中使用事务作用域，还是使用自动提交 (**auto-commit**)？（对很多数据库和/或 JDBC 驱动来说，等同于关闭事务支持）
2. **数据库连接**：你希望 MyBatis 帮你从已配置的数据源获取连接，还是使用自己提供的连接？
3. **语句执行**：你希望 MyBatis 复用 `PreparedStatement` 和/或批量更新语句（包括插入语句和删除语句）吗？

默认的 `openSession()` 方法没有参数，它会创建具备如下特性的 `SqlSession`：

1. 事务作用域将会开启（也就是**不自动提交**）。
2. 将由当前环境配置的 `DataSource` 实例中获取 `Connection` 对象。
3. 事务隔离级别将会使用驱动或数据源的默认设置。
4. 预处理语句不会被复用，也不会批量处理更新。

向 `autoCommit` 可选参数传递 `true` 值即可开启自动提交功能。若要使用自己的 `Connection` 实例，传递一个 `Connection` 实例给 `connection` 参数即可。注意，我们没有提供同时设置 `Connection` 和 `autoCommit` 的方法，这是因为 MyBatis 会依据传入的 `Connection` 来决定是否启用 `autoCommit`。对于事务隔离级别，MyBatis 使用了一个 Java 枚举包装器来表示，称为 `TransactionIsolationLevel`，事务隔离级别支持 JDBC 的五个隔离级别（`NONE`、`READ_UNCOMMITTED`、`READ_COMMITTED`、`REPEATABLE_READ` 和 `SERIALIZABLE`），并且与预期的行为一致

ExecutorType:

1. `ExecutorType.SIMPLE`：该类型的执行器没有特别的行为。它为每个语句的执行创建一个新的预处理语句。
2. `ExecutorType.REUSE`：该类型的执行器会**复用预处理语句**。
3. `ExecutorType.BATCH`：该类型的执行器会**批量执行所有更新语句**，如果 `SELECT` 在多个更新中间执行，将在必要时将多条更新语句分隔开来，以方便理解。

在 `SqlSessionFactory` 中还有一个方法我们没有提及，就是 `getConfiguration()`。这个方法会返回一个 `Configuration` 实例，你可以在运行时使用它来检查 MyBatis

的配置。

SqlSession

SqlSession 在 **MyBatis** 中是非常强大的一个类。它包含了所有执行语句、提交或回滚事务以及获取映射器实例的方法。

SqlSession 类的方法超过了 20 个，为了方便理解，我们将它们分成几种组别。

● 语句执行方法

这些方法被用来执行定义在 SQL 映射 XML 文件中的 **SELECT**、**INSERT**、**UPDATE** 和 **DELETE** 语句。

每一方法都接受语句的 **ID(Statement)** 以及参数对象，参数可以是原始类型（支持自动装箱或包装类）、**JavaBean**、**POJO** 或 **Map**。

- `<T> T selectOne(String statement, Object parameter)`
- `<E> List<E> selectList(String statement, Object parameter)`
- `<T> Cursor<T> selectCursor(String statement, Object parameter)`
- `<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey)`
- `int insert(String statement, Object parameter)`
- `int update(String statement, Object parameter)`
- `int delete(String statement, Object parameter)`

selectOne 和 **selectList** 的不同仅仅是 **selectOne** 必须返回一个对象或 **null** 值。如果返回值多于一个，就会抛出异常。如果你不知道返回对象会有多少，请使用 **selectList**。如果需要查看某个对象是否存在，最好的办法是查询一个 **count** 值（0 或 1）。**selectMap** 稍微特殊一点，它会将返回对象的其中一个属性作为 **key** 值，将对象作为 **value** 值，从而将多个结果集转为 **Map** 类型值。由于并不是所有语句都需要参数，所以这些方法都具有一个不需要参数的重载形式。

游标 (Cursor) 与 **列表 (List)** 返回的结果相同，不同的是，游标借助迭代器实现了数据的惰性加载。

```
try (Cursor<MyEntity> entities = session.selectCursor(statement, param)) {
    for (MyEntity entity:entities) {
        // 处理单个实体
    }
}
```

还有 **select** 方法的三个高级版本，它们允许你限制返回行数的范围，或是提供自定义结果处理逻辑，通常在数据集非常庞大的情形下使用。

1. `<E> List<E> selectList (String statement, Object parameter, RowBounds rowBounds)`
2. `<T> Cursor<T> selectCursor(String statement, Object parameter, RowBounds rowBounds)`
3. `<K,V> Map<K,V> selectMap(String statement, Object parameter, String mapKey, RowBounds rowBounds)`
4. `void select (String statement, Object parameter, ResultHandler<T> handler)`

```
5. void select (String statement, Object parameter, RowBounds rowBounds, ResultHandler<T> handler)
```

RowBounds 参数会告诉 **MyBatis** 略过指定数量的记录，并限制返回结果的数量。**RowBounds** 类的 **offset** 和 **limit** 值只有在构造函数时才能传入，其它时候是不能修改的。

● 立即批量更新方法

当你将 **ExecutorType** 设置为 **ExecutorType.BATCH** 时，可以使用这个方法清除（执行）缓存在 **JDBC** 驱动类中的批量更新语句。

```
List<BatchResult> flushStatements()
```

● 事务控制方法

有四个方法用来控制事务作用域。当然，如果你已经设置了自动提交或你使用了外部事务管理器，这些方法就没什么作用了。然而，如果你正在使用由 **Connection** 实例控制的 **JDBC** 事务管理器，那么这四个方法就会派上用场：

1. **void commit()**
2. **void commit(boolean force)**
3. **void rollback()**
4. **void rollback(boolean force)**

默认情况下 **MyBatis** 不会自动提交事务，除非它检测到调用了插入、更新或删除方法改变了数据库。如果你没有使用这些方法提交修改，那么你可以在 **commit** 和 **rollback** 方法参数中传入 **true** 值，来保证事务被正常提交（注意，在自动提交模式或者使用了外部事务管理器的情况下，设置 **force** 值对 **session** 无效）。大部分情况下你无需调用 **rollback()**，因为 **MyBatis** 会在你没有调用 **commit** 时替你完成回滚操作。不过，当你要在一个可能多次提交或回滚的 **session** 中详细控制事务，回滚操作就派上用场了。

● 本地缓存

Mybatis 使用到了两种缓存：**本地缓存(local cache)**和**二级缓存(second level cache)**。

每当一个新 **session** 被创建，**MyBatis** 就会创建一个与之相关联的**本地缓存**。任何在 **session** 执行过的查询结果都会被保存在本地缓存中，所以，当再次执行参数相同的相同查询时，就不需要实际查询数据库了。**本地缓存**将会在做出修改、事务提交或回滚，以及关闭 **session** 时清空。

默认情况下，本地缓存数据的生命周期等同于整个 **session** 的周期。由于缓存会被用来解决循环引用问题和加快重复嵌套查询的速度，所以无法将其完全禁用。但是你可以通过设置 **localCacheScope=STATEMENT** 来只在语句执行时使用缓存。

你可以随时调用以下方法来清空本地缓存

```
void clearCache()
```

- 确保 `SqlSession` 被关闭
`Close()`
- 使用 `Mapper`
`<T> T getMapper(Class<T> type)`

上述的各个 `insert`、`update`、`delete` 和 `select` 方法都很强大，但也有些繁琐，它们并不符合类型安全，对你的 IDE 和单元测试也不是那么友好。因此，使用映射器类来执行映射语句是更常见的做法。

一个映射器类：就是一个仅需声明与 `SqlSession` 方法相匹配方法的接口。

映射器方法签名应该匹配相关联的 `SqlSession` 方法，字符串参数 ID 无需匹配。而是由方法名匹配映射语句的 ID。

映射器接口不需要去实现任何接口或继承自任何类。只要方法签名可以被用来唯一识别对应的映射语句就可以了

映射器接口可以继承自其他接口。在使用 XML 来绑定映射器接口时，保证语句处于合适的命名空间中即可。唯一的限制是，不能在两个具有继承关系的接口中拥有相同的方法签名（这是潜在的危险做法，不可取）。

Java API：映射器注解

注解	使用对象	XML 等价形式	描述
<code>@CacheNamespace</code>	类	<code><cache></code>	为给定的命名空间（比如类）配置缓存。属性： <code>implemetation</code> 、 <code>eviction</code> 、 <code>flushInterval</code> 、 <code>size</code> 、 <code>readWrite</code> 、 <code>blocking</code> 、 <code>properties</code> 。
<code>@Property</code>	N/A	<code><property></code>	指定参数值或占位符（placeholder）（该占位符能被 <code>mybatis-config.xml</code> 内的配置属性替换）。属性： <code>name</code> 、 <code>value</code> 。（仅在 <code>MyBatis 3.4.2</code> 以上可用）
<code>@CacheNamespaceRef</code>	类	<code><cacheRef></code>	引用另外一个命名空间的缓存以供使用。注意，即使共享相同的全限定类名，

			在 XML 映射文件中声明的缓存仍被识别为一个独立的命名空间。属性: value、name。如果你使用了这个注解, 你应设置 value 或者 name 属性的其中一个。value 属性用于指定能够表示该命名空间的 Java 类型 (命名空间名就是该 Java 类型的全限定类名), name 属性 (这个属性仅在 MyBatis 3.4.2 以上可用) 则直接指定了命名空间的名称。
@ConstructorArgs	方法	<constructor>	收集一组结果以传递给一个结果对象的构造方法。属性: value, 它是一个 Arg 数组。
@Arg	N/A	1. <arg> 2. <idArg>	ConstructorArgs 集合的一部分, 代表一个构造方法参数。属性: id、column、javaType、jdbcType、typeHandler、select、resultMap。id 属性和 XML 元素 <idArg> 相似, 它是一个布尔值, 表示该属性是否用于唯一标识和比较对象。从版本 3.5.4 开始, 该注解变为可重复注解。
@TypeDiscriminator	方法	<discriminator>	决定使用何种结果映射的一组取值 (case)。属性: column、javaType、jdbcType、typeHandler、cases。cases 属性是一个 Case 的数组。
@Case	N/A	<case>	表示某个值的一个取值以及该取值对应的映射。属性: value、type、results。results 属性是一个 Results 的数组, 因此这个注解实际上和 resultMap 很相似, 由下面的 Results 注解指定。
@Results	方法	<resultMap>	一组结果映射, 指定了对某个特定结果列, 映射到某个属性或字段的方式。属性: value、id。value 属性是一个 Result 注解的数组。而 id 属性则是结果映射的名称。从版本 3.5.4 开始, 该注解变为可重复注解。
@Result	N/A	1. <result> 2. <id>	在列和属性或字段之间的单个结果映射。属性: id、column、javaType、jdbcType、typeHandler、one、many。id 属性和 XML 元素 <id> 相似, 它是

			<p>一个布尔值，表示该属性是否用于唯一标识和比较对象。one 属性是一个关联，和 <code><association></code> 类似，而 many 属性则是集合关联，和 <code><collection></code> 类似。这样命名是为了避免产生名称冲突。</p>
@One	N/A	<code><association></code>	<p>复杂类型的单个属性映射。属性： select，指定可加载合适类型实例的映射语句（也就是映射器方法）全限定名； fetchType，指定在该映射中覆盖全局配置参数 <code>lazyLoadingEnabled</code>； resultMap(available since 3.5.5), which is the fully qualified name of a result map that map to a single container object from select result； columnPrefix(available since 3.5.5), which is column prefix for grouping select columns at nested result map. 提示 注解 API 不支持联合映射。这是由于 Java 注解不允许产生循环引用。</p>
@Many	N/A	<code><collection></code>	<p>复杂类型的集合属性映射。属性： select，指定可加载合适类型实例集合的映射语句（也就是映射器方法）全限定名； fetchType，指定在该映射中覆盖全局配置参数 <code>lazyLoadingEnabled</code>； resultMap(available since 3.5.5), which is the fully qualified name of a result map that map to collection object from select result； columnPrefix(available since 3.5.5), which is column prefix for grouping select columns at nested result map. 提示 注解 API 不支持联合映射。这是由于 Java 注解不允许产生循环引用。</p>
@MapKey	方法		<p>供返回值为 <code>Map</code> 的方法使用的注解。它使用对象的某个属性作为 <code>key</code>，将对象 <code>List</code> 转化为 <code>Map</code>。属性：value，指定</p>

			作为 Map 的 key 值的对象属性名。
@Options	方法	映射语句的属性	<p>该注解允许你指定大部分开关和配置选项，它们通常在映射语句上作为属性出现。与在注解上提供大量的属性相比，Options 注解提供了一致、清晰的方式来指定选项。属性：useCache=true、flushCache=FlushCachePolicy.DEFAULT、resultSetType=DEFAULT、statementType=PREPARED、fetchSize=-1、timeout=-1、useGeneratedKeys=false、keyProperty=""、keyColumn=""、resultSets="", databaseId=""。注意，Java 注解无法指定 null 值。因此，一旦你使用了 Options 注解，你的语句就会被上述属性的默认值所影响。要注意避免默认值带来的非预期行为。The databaseId(Available since 3.5.5), in case there is a configured DatabaseIdProvider, the MyBatis use the Options with no databaseId attribute or with a databaseId that matches the current one. If found with and without the databaseId the latter will be discarded.</p>
@Insert @Update @Delete @Select			CURD
@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider			<p>允许构建动态 SQL。这些备选的 SQL 注解允许你指定返回 SQL 语句的类和方法，以供运行时执行。（从 MyBatis 3.4.6 开始，可以使用 CharSequence 代替 String 来作为返回类型）。当执行映射语句时，MyBatis 会实例化注解指定的类，并调用注解指定的方法。你可以通过 ProviderContext 传递映射方法接收到的参数、"Mapper interface type" 和 "Mapper method"（仅在</p>

			<p>MyBatis 3.4.5 以上支持) 作为参数。</p> <p>(MyBatis 3.4 以上支持传入多个参数) 属性: value、type、method、databaseId。value and type 属性用于指定类名 (The type attribute is alias for value, you must be specify either one. But both attributes can be omit when specify the defaultSqlProviderType as global configuration)。method 用于指定该类的方法名(从版本 3.5.1 开始, 可以省略 method 属性, MyBatis 将会使用 ProviderMethodResolver 接口解析方法的具体实现。如果解析失败, MyBatis 将会使用名为 provideSql 的降级实现)。提示 接下来的“SQL 语句构建器”一章将会讨论该话题, 以帮助你以更清晰、更便于阅读的方式构建动态 SQL。 The databaseId(Available since 3.5.5), in case there is a configured DatabaseIdProvider, the MyBatis will use a provider method with no databaseId attribute or with a databaseId that matches the current one. If found with and without the databaseId the latter will be discarded.</p>
@Param	参数	N/A	<p>如果你的映射方法接受多个参数, 就可以使用这个注解自定义每个参数的名字。否则在默认情况下, 除 RowBounds 以外的参数会以 "param" 加参数位置被命名。例如 <code>#{param1}</code>, <code>#{param2}</code>。如果使用了 <code>@Param("person")</code>, 参数就会被命名为 <code>#{person}</code>。</p>
@SelectKey	方法	<selectKey>	<p>这个注解的功能与 <selectKey> 标签完全一致。该注解只能在 @Insert 或</p>

			<p><code>@InsertProvider</code> 或 <code>@Update</code> 或 <code>@UpdateProvider</code> 标注的方法上使用, 否则将会被忽略。如果标注了 <code>@SelectKey</code> 注解, MyBatis 将会忽略掉由 <code>@Options</code> 注解所设置的生成主键或设置(configuration)属性。属性: <code>statement</code> 以字符串数组形式指定将会被执行的 SQL 语句, <code>keyProperty</code> 指定作为参数传入的对象对应属性的名称, 该属性将会更新成新的值, <code>before</code> 可以指定为 <code>true</code> 或 <code>false</code> 以指明 SQL 语句应被在插入语句的之前还是之后执行。<code>resultType</code> 则指定 <code>keyProperty</code> 的 Java 类型。<code>statementType</code> 则用于选择语句类型, 可以选择 <code>STATEMENT</code>、<code>PREPARED</code> 或 <code>CALLABLE</code> 之一, 它们分别对应于 <code>Statement</code>、<code>PreparedStatement</code> 和 <code>CallableStatement</code>。默认值是 <code>PREPARED</code>。</p> <p>The <code>databaseId</code>(Available since 3.5.5), in case there is a configured <code>DatabaseIdProvider</code>, the MyBatis will use a statement with no <code>databaseId</code> attribute or with a <code>databaseId</code> that matches the current one. If found with and without the <code>databaseId</code> the latter will be discarded.</p>
<code>@ResultMap</code>	方法	N/A	<p>这个注解为 <code>@Select</code> 或者 <code>@SelectProvider</code> 注解指定 XML 映射中 <code><resultMap></code> 元素的 id。这使得注解的 <code>select</code> 可以复用已在 XML 中定义的 <code>ResultMap</code>。如果标注的 <code>select</code> 注解中存在 <code>@Results</code> 或者 <code>@ConstructorArgs</code> 注解, 这两个注解将被此注解覆盖。</p>
<code>@ResultType</code>	方法	N/A	<p>在使用了结果处理器的情况下, 需要使用此注解。由于此时的返回类型为 <code>void</code>, 所以 Mybatis 需要有一种方法来判断每一行返回的对象类型。如果在 XML 有对应的结果映射, 请使用 <code>@ResultMap</code> 注解。如果结果类型在 XML 的 <code><select></code> 元素中指定了, 就不</p>

			需要使用其它注解了。否则就需要使用此注解。比如，如果一个标注了 <code>@Select</code> 的方法想要使用结果处理器，那么它的返回类型必须是 <code>void</code> ，并且必须使用这个注解（或者 <code>@ResultMap</code> ）。这个注解仅在方法返回类型是 <code>void</code> 的情况下生效。
<code>@Flush</code>	方法	N/A	如果使用了这个注解，定义在 <code>Mapper</code> 接口中的方法就能够调用 <code>SqlSession#flushStatements()</code> 方法。（Mybatis 3.3 以上可用）

SQL 语句构建器

MyBatis 3 提供了方便的工具类来帮助解决此问题。借助 `SQL` 类，我们只需要简单地创建一个实例，并调用它的方法即可生成 SQL 语句。让我们来用 `SQL` 类重写上面的例子：

```
private String selectPersonSql() {
    return new SQL() {{
        SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
        SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
        FROM("PERSON P");
        FROM("ACCOUNT A");
        INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
        INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
        WHERE("P.ID = A.ID");
        WHERE("P.FIRST_NAME like ?");
        OR();
        WHERE("P.LAST_NAME like ?");
        GROUP_BY("P.ID");
        HAVING("P.LAST_NAME like ?");
        OR();
        HAVING("P.FIRST_NAME like ?");
        ORDER_BY("P.ID");
        ORDER_BY("P.FULL_NAME");
    }}.toString();
}
```

SQL 类：

```
// 匿名内部类风格
public String deletePersonSql() {
    return new SQL() {{
        DELETE_FROM("PERSON");
        WHERE("ID = #{id}");
    }}.toString();
}

// Builder / Fluent 风格
public String insertPersonSql() {
    String sql = new SQL()
        .INSERT_INTO("PERSON")
        .VALUES("ID, FIRST_NAME", "#{id}, #{firstName}")
        .VALUES("LAST_NAME", "#{lastName}")
        .toString();
    return sql;
}

// 动态条件（注意参数需要使用 final 修饰，以便匿名内部类对它们进行访问）
```

```

public String selectPersonLike(final String id, final String firstName, final String
lastName) {
    return new SQL() {{
        SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
        FROM("PERSON P");
        if (id != null) {
            WHERE("P.ID like #{id}");
        }
        if (firstName != null) {
            WHERE("P.FIRST_NAME like #{firstName}");
        }
        if (lastName != null) {
            WHERE("P.LAST_NAME like #{lastName}");
        }
        ORDER_BY("P.LAST_NAME");
    }}.toString();
}

public String deletePersonSql() {
    return new SQL() {{
        DELETE_FROM("PERSON");
        WHERE("ID = #{id}");
    }}.toString();
}

public String insertPersonSql() {
    return new SQL() {{
        INSERT_INTO("PERSON");
        VALUES("ID, FIRST_NAME", "#{id}, #{firstName}");
        VALUES("LAST_NAME", "#{lastName}");
    }}.toString();
}

public String updatePersonSql() {
    return new SQL() {{
        UPDATE("PERSON");
        SET("FIRST_NAME = #{firstName}");
        WHERE("ID = #{id}");
    }}.toString();
}

```

日志

Mybatis 通过使用内置的日志工厂提供日志功能。内置日志工厂将会把日志工作委托给下面的实现之一：

1. SLF4J
2. Apache Commons Logging
3. Log4j 2
4. Log4j
5. JDK logging

不少应用服务器(如 Tomcat 和 WebShpere)的类路径中已经包含 Commons Logging。注意,在这种配置环境下,MyBatis 会把 Commons Logging 作为日志工具。这就意味着在诸如 WebSphere 的环境中,由于提供了 Commons Logging 的私有实现,你的 Log4J 配置将被忽略。这个时候你就会感觉很郁闷:看起来 MyBatis 将你的 Log4J 配置忽略掉了(其实是因为在这种配置环境下,MyBatis 使用了 Commons

Logging 作为日志实现)。如果你的应用部署在一个类路径已经包含 Commons Logging 的环境中，而你又想使用其它日志实现，你可以通过在 MyBatis 配置文件 mybatis-config.xml 里面添加一项 setting 来选择其它日志实现。

```
<configuration>
  <settings>
    ...
    <setting name="logImpl" value="LOG4J"/>
    ...
  </settings>
</configuration>
```

可选的值有：SLF4J、LOG4J、LOG4J2、JDK_LOGGING、COMMONS_LOGGING、STDOUT_LOGGING、NO_LOGGING，或者是实现了 org.apache.ibatis.logging.Log 接口，且构造方法以字符串为参数的类完全限定名。

日志配置

具体配置步骤取决于日志实现。接下来我们会以 Log4J 作为示范。配置日志功能非常简单：添加一个或多个配置文件（如 log4j.properties），有时还需要添加 jar 包（如 log4j.jar）。下面的例子将使用 Log4J 来配置完整的日志服务。一共两个步骤：

步骤 1：添加 Log4J 的 jar 包

步骤 2：配置 Log4J

```
# 全局日志配置
log4j.rootLogger=ERROR, stdout
# MyBatis 日志配置
log4j.logger.org.mybatis.example.BlogMapper=TRACE
# 控制台输出
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

over

很多知识只是过一遍，还需要慢慢消化，如果要了解底层原理，还需要看源码，以及其他各种方式学习。

@from: <https://mybatis.org/mybatis-3/zh/index.html>

@date: 2021/02/02

Page-helper

分页器

MyBatis-Plus

MyBatis-Plus (opens new window) (简称 MP) 是一个 **MyBatis** (opens new window) 的增强工具，在 **MyBatis** 的基础上只做增强不做改变，为简化开发、提高效率而生。

<https://baomidou.com/>

<https://github.com/baomidou/mybatis-plus>

补充

Spring 中使用变量\${}的方式进行参数配置

在使用 **Spring** 时，有些情况下，在配置文件中，需要使用变量的方式来配置 **bean** 相关属性信息，比如下面的数据库的连接使用了\${}的方式进行配置

```
<property name="username" value="${username}" />
<property name="password" value="${password}" />
```

那么上面的\${}中的变量具体的值是从哪来的？

有两种方式来进行配置：

1. 方法一：使用 **bean** 的注入来引入配置文件：

<!-- 引入配置文件 -->

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="classpath*:jdbc.properties" />
</bean>
```

2. 方法二： **spring3** 提供更简单的方式，使用自动扫描的方式：

<!-- 配置文件 -->

```
<context:property-placeholder location="classpath*:jdbc.properties" />
```

以上两种方式都可以将指定的配置文件加载进来，然后通过\${}符号的引用，即可通过外部对变量的修改，来进行切换，不需要每次改动内部的值！

jdbc.properties 文件的内容格式也很简单，是键值对的方式