

## 课程目标

- 1、了解 ORM 框架发展历史，了解 MyBatis 特性
- 2、掌握 MyBatis 程式开发方法和核心对象
- 3、掌握 MyBatis 核心配置含义
- 4、掌握 MyBatis 的高级用法与扩展方式

## 内容定位

适合已掌握 MyBatis 基本用法且希望对 MyBatis 进一步深入了解的人群。

掌握 MyBatis 的核心特性，以及如何用好 MyBatis。

## 课程大纲

### 1. 为什么要用 MyBatis

#### JDBC 连接数据库

在 Java 程序里面去连接数据库，最原始的办法是使用 JDBC 的 API。我们先来回顾一下使用 JDBC 的方式，我们是怎么操作数据库的。

```
// 注册 JDBC 驱动
Class.forName("com.mysql.jdbc.Driver");

// 打开连接
conn = DriverManager.getConnection(DB_URL, USER, PASSWORD);

// 执行查询
stmt = conn.createStatement();
```

```
String sql= "SELECT bid, name, author_id FROM blog";
ResultSet rs = stmt.executeQuery(sql);

// 获取结果集
while(rs.next()) {
    int bid = rs.getInt("bid");
    String name = rs.getString("name");
    String authorId = rs.getString("author_id");
}
```

首先，我们在 maven 中引入 MySQL 驱动的依赖（JDBC 的包在 java.sql 中）。

第一步，注册驱动，第二步，通过 DriverManager 获取一个 Connection，参数里面填数据库地址，用户名和密码。

第三步，我们通过 Connection 创建一个 Statement 对象。

第四步，通过 Statement 的 execute()方法执行 SQL。当然 Statement 上面定义非常多的方法。execute()方法返回一个 ResultSet 对象，我们把它叫做结果集。

第五步，我们通过 ResultSet 获取数据。转换成一个 POJO 对象。

最后，我们要关闭数据库相关的资源，包括 ResultSet、Statement、Connection，它们的关闭顺序和打开的顺序正好是相反的。

这个就是我们通过 JDBC 的 API 去操作数据库的方法，这个仅仅是一个查询。如果我们项目当中的业务比较复杂，表非常多，各种操作数据库的增删改查的方法也比较多的话，那么这样代码会重复出现很多次。

在每一段这样的代码里面，我们都需要自己去管理数据库的连接资源，如果忘记写 close()了，就可能会造成数据库服务连接耗尽。

另外还有一个问题就是处理业务逻辑和处理数据的代码是耦合在一起的。如果业务流程复杂，跟数据库的交互次数多，耦合在代码里面的 SQL 语句就会非常多。如果要修

改业务逻辑，或者修改数据库环境（因为不同的数据库 SQL 语法略有不同），这个工作量也是难以估计的。

还有就是对于结果集的处理，我们要把 ResultSet 转换成 POJO 的时候，必须根据字段属性的类型一个个地去处理，写这样的代码是非常枯燥的：

```
int bid = rs.getInt("bid");
String name = rs.getString("name");
String authorId = rs.getString("author_id");
blog.setAuthorId(authorId);
blog.setBid(bid);
blog.setName(name);
```

也正是因为这样，我们在实际工作中是比较少直接使用 JDBC 的。那么我们在 Java 程序里面有哪些更加简单的操作数据库的方式呢？

Apache DbUtils

<https://commons.apache.org/proper/commons-dbutils/>

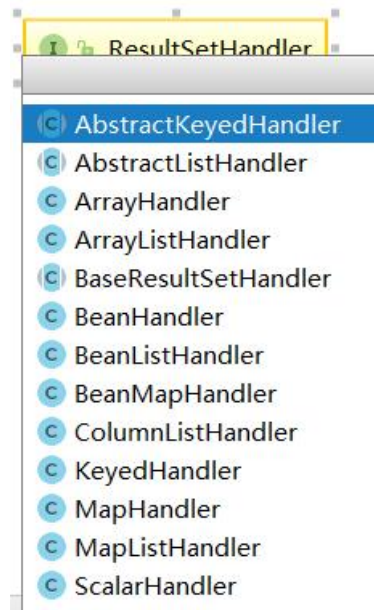
DbUtils 解决的最核心的问题就是结果集的映射，可以把 ResultSet 封装成 JavaBean。它是怎么做到的呢？

首先 DbUtils 提供了一个 QueryRunner 类，它对数据库的增删改查的方法进行了封装，那么我们操作数据库就可以直接使用它提供的方法。

在 QueryRunner 的构造函数里面，我们又可以传入一个数据源，比如在这里我们 Hikari，这样我们就不需要再去写各种创建和释放连接的代码了。

```
queryRunner = new QueryRunner(dataSource);
```

那我们怎么把结果集转换成对象呢？比如实体类 Bean 或者 List 或者 Map？在 DbUtils 里面提供了一系列的支持泛型的 ResultSetHandler。



我们只要在 DAO 层调用 QueryRunner 的查询方法，传入这个 Handler，它就可以自动把结果集转换成实体类 Bean 或者 List 或者 Map。

```
String sql = "select * from blog";
List<BlogDto> list = queryRunner.query(sql, new BeanListHandler<>(BlogDto.class));
```

没有用过 DbUtils 的同学，可以思考一下通过结果集到实体类的映射是怎么实现的？也就是说，我只传了一个实体类的类型，它怎么知道这个类型有哪些属性，每个属性是什么类型？然后创建这个对象并且给这些字段赋值的？答案正是反射。

大家也可以去看一下源码印证一下是不是这样。

问题：输出的结果中，authorId 为什么是空的？DbUtils 要求数据库的字段跟对象的属性名称完全一致，才可以实现自动映射。

```
BlogDto{bid=3, name='MyBatis 源码分析', authorId='null'}
```

## Spring JDBC

除了 DbUtils 之外，Spring 也对原生的 JDBC 进行了封装，并且给我们提供了一个模板方法 JdbcTemplate，来简化我们对数据库的操作。第一个，我们不再需要去关心资

源管理的问题。

第二个，对于结果集的处理，Spring JDBC 也提供了一个 RowMapper 接口，可以把结果集转换成 Java 对象。

看代码：比如我们要把结果集转换成 Employee 对象，就可以针对一个 Employee 创建一个 RowMapper 对象，实现 RowMapper 接口，并且重写 mapRow() 方法。我们在 mapRow() 方法里面完成对结果集的处理。

```
public class EmployeeRowMapper implements RowMapper {  
    @Override  
    public Object mapRow(ResultSet resultSet, int i) throws SQLException {  
        Employee employee = new Employee();  
        employee.setEmpId(resultSet.getInt("emp_id"));  
        employee.setEmpName(resultSet.getString("emp_name"));  
        employee.setEmail(resultSet.getString("emial"));  
  
        return employee;  
    }  
}
```

在 DAO 层调用的时候就可以传入自定义的 RowMapper 类，最终返回我们需要的类型。结果集和实体类类型的映射也是自动完成的。

```
public List<Employee> query(String sql) {  
    new JdbcTemplate( new DruidDataSource());  
    return jdbcTemplate.query(sql, new EmployeeRowMapper());  
}
```

通过这种方式，我们对于结果集的处理只需要写一次代码，然后在每一个需要映射的地方传入这个 RowMapper 就可以了，减少了很多的重复代码。

但是还是有问题：每一个实体类对象，都需要定义一个 Mapper，然后要编写每个字段映射的 getString()、getInt 这样的代码，还增加了类的数量。

所以有没有办法让一行数据的字段，跟实体类的属性自动对应起来，实现自动映射呢？当然，我们肯定要解决两个问题，一个就是名称对应的问题，从下划线到驼峰命名；

第二个是类型对应的问题，数据库的 JDBC 类型和 Java 对象的类型要匹配起来。

我们可以创建一个 BaseRowMapper<T>，通过反射的方式自动获取所有属性，把表字段全部赋值到属性。

上面的方法就可以改成：

```
return jdbcTemplate.query(sql, new BaseRowMapper(Employee.class));
```

这样，我们在使用的时候只要传入我们需要转换的类型就可以了，不用再单独创建一个 RowMapper。

我们来总结一下，DbUtils 和 Spring JDBC，这两个对 JDBC 做了轻量级封装的框架，或者说工具类里面，都帮助我们解决了一些问题：

- 1、 无论是 QueryRunner 还是 JdbcTemplate，都可以传入一个数据源进行初始化，也就是资源管理这一部分的事情，可以交给专门的数据源组件去做，不用我们手动创建和关闭；
- 2、 对操作数据的增删改查的方法进行了封装；
- 3、 可以帮助我们映射结果集，无论是映射成 List、Map 还是实体类。

但是还是存在一些缺点：

- 1、 SQL 语句都是写死在代码里面的，依旧存在硬编码的问题；
- 2、 参数只能按固定位置的顺序传入（数组），它是通过占位符去替换的，不能自动映射；
- 3、 在方法里面，可以把结果集映射成实体类，但是不能直接把实体类映射成数据库的记录（没有自动生成 SQL 的功能）；
- 4、 查询没有缓存的功能。



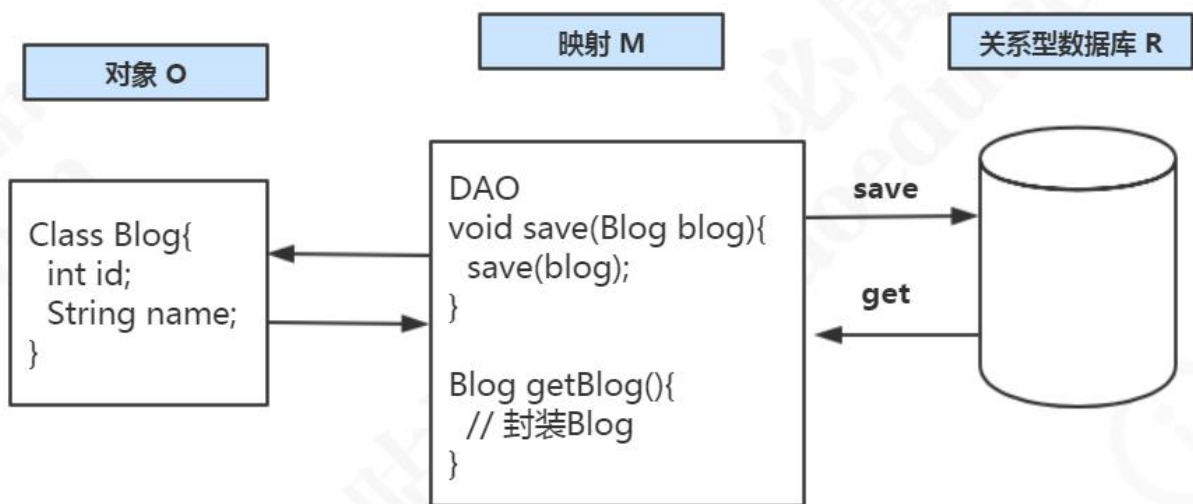
## Hibernate

要解决这些问题，使用这些工具类还是不够的，要用到我们今天讲的 ORM 框架。

那什么是 ORM？为什么叫 ORM？

ORM 的全拼是 Object Relational Mapping，也就是对象与关系的映射，对象是程序里面的对象，关系是它与数据库里面的数据的关系。也就是说，ORM 框架帮助我们解决的问题是程序对象和关系型数据库的相互映射的问题。

O：对象——M：映射——R：关系型数据库



今天听课的同学应该有很多同学是用过 Hibernate 或者现在还在用的。Hibernate 是一个很流行的 ORM 框架，2001 年的时候就出了第一个版本。在使用 Hibernate 的时候，我们需要为实体类建立一些 hbm 的 xml 映射文件（或者类似于@Table 的这样的注解）。例如：

```

<hibernate-mapping>
  <class name="cn.gupaoedu.vo.User" table="user">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="password"/>
    <property name="cellphone"/>
  </class>
</hibernate-mapping>
  
```

```
<property name="username"/>
</class>
</hibernate-mapping>
```

然后通过 Hibernate 提供 ( session ) 的增删改查的方法来操作对象。

```
//创建对象
User user = new User();
user.setPassword("123456");
user.setCellphone("18166669999");
user.setUsername("qingshan");
//获取加载配置管理类
Configuration configuration = new Configuration();
//不给参数就默认加载 hibernate.cfg.xml 文件，
configuration.configure();

//创建 Session 工厂对象
SessionFactory factory = configuration.buildSessionFactory();
//得到 Session 对象
Session session = factory.openSession();
//使用 Hibernate 操作数据库，都要开启事务，得到事务对象
Transaction transaction = session.getTransaction();
//开启事务
transaction.begin();
//把对象添加到数据库中
session.save(user);

//提交事务
transaction.commit();

//关闭 Session
session.close();
```

我们操作对象就跟操作数据库的数据一样。Hibernate 的框架会自动帮我们生成 SQL 语句（可以屏蔽数据库的差异），自动进行映射。这样我们的代码变得简洁了，程序的可读性也提高了。

但是 Hibernate 在业务复杂的项目中使用也存在一些问题：

1、比如使用 get()、save()、update() 对象的这种方式，实际操作的是所有字段，



没有办法指定部分字段，换句话说就是不够灵活。

2、这种自动生成 SQL 的方式，如果我们要去做一些优化的话，是非常困难的，也就是说可能会出现性能比较差的问题。

3、不支持动态 SQL（比如分表中的表名变化，以及条件、参数）。

## MyBatis

“半自动化”的 ORM 框架 MyBatis 就解决了这几个问题。“半自动化”是相对于 Hibernate 的全自动化来说的，也就是说它的封装程度没有 Hibernate 那么高，不会自动生成全部的 SQL 语句，主要解决的是 SQL 和对象的映射问题。

在 MyBatis 里面，SQL 和代码是分离的，所以会写 SQL 基本上就会用 MyBatis，没有额外的学习成本。

我们来总结一下，MyBatis 的核心特性，或者说它解决的主要问题是什么：

- 1、使用连接池对连接进行管理
- 2、SQL 和代码分离，集中管理
- 3、结果集映射
- 4、参数映射和动态 SQL
- 5、重复 SQL 的提取
- 6、缓存管理
- 7、插件机制

当然，需要明白的是，Hibernate 和 MyBatis 跟 DbUtils、Spring JDBC 一样，都

是对 JDBC 的一个封装，我们去看源码，最后一定会看到 Statement 和 ResultSet 这些对象。

问题来了，我们有这么多的工具和不同的框架，在实际的项目里面应该怎么选择？

在一些业务比较简单的项目中，我们可以使用 Hibernate；

如果需要更加灵活的 SQL，可以使用 MyBatis，对于底层的编码，或者性能要求非常高的场合，可以用 JDBC。

实际上在我们的项目中，MyBatis 和 Spring JDBC 是可以混合使用的。

当然，我们也根据项目的需求自己写 ORM 框架，就像之前 Tom 老师跟大家讲的手写 ORM 框架一样。

## 2. MyBatis 实际使用案例

### 编程式使用

大部分时候，我们都是 Spring 里面去集成 MyBatis。因为 Spring 对 MyBatis 的一些操作进行的封装，我们不能直接看到它的本质，所以先看下不使用容器的时候，也就是编程的方式，MyBatis 怎么使用。

先引入 mybatis jar 包。

首先我们要创建一个全局配置文件，这里面是对 MyBatis 的核心行为的控制，比如 mybatis-config.xml。

第二个就是我们的映射器文件，Mapper.xml，通常来说一张表对应一个，我们会在这个里面配置我们增删改查的 SQL 语句，以及参数和返回的结果集的映射关系。

跟 JDBC 的代码一样，我们要执行对数据库的操作，必须创建一个会话，这个在 MyBatis 里面就是 `SqlSession`。`SqlSession` 又是工厂类根据全局配置文件创建的。所以整个的流程就是这样的（如下代码）。最后我们通过 `SqlSession` 接口上的方法，传入我们的 `Statement ID` 来执行 SQL。这是第一种方式。

这种方式有一个明显的缺点，就是会对 `Statement ID` 硬编码，而且不能在编译时进行类型检查，所以通常我们会使用第二种方式，就是定义一个 `Mapper` 接口的方式。这个接口全路径必须跟 `Mapper.xml` 里面的 `namespace` 对应起来，方法也要跟 `Statement ID` 一一对应。

```
public void testMapper() throws IOException {
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new
    SqlSessionFactoryBuilder().build(inputStream);

    SqlSession session = sqlSessionFactory.openSession();
    try {
        BlogMapper mapper = session.getMapper(BlogMapper.class);
        Blog blog = mapper.selectBlogById(1);
        System.out.println(blog);
    } finally {
        session.close();
    }
}
```

这个就是我们单独使用 MyBatis 的全部流程。

这个案例非常重要，后面我们讲源码还是基于它。

## 核心对象的生命周期

在程式使用的这个 demo 里面，我们看到了 MyBatis 里面的几个核心对象：`SqlSessionFactoryBuiler`、`SqlSessionFactory`、`SqlSession` 和 `Mapper` 对象。这几个核心对象在 MyBatis 的整个工作流程里面的不同环节发挥作用。如果说我们不用容器，

自己去管理这些对象的话，我们必须思考一个问题：什么时候创建和销毁这些对象？

在一些分布式的应用里面，多线程高并发的场景中，如果要写出高效的代码，必须了解这四个对象的生命周期。这四个对象的声明周期的描述在官网上面也可以找到。

<http://www.mybatis.org/mybatis-3/zh/getting-started.html>

我们从每个对象的作用的角度来理解一下，只有理解了它们是干什么的，才知道什么时候应该创建，什么时候应该销毁。

### 1 ) SqlSessionFactoryBuiler

首先是 SqlSessionFactoryBuiler。它是用来构建 SqlSessionFactory 的，而 SqlSessionFactory 只需要一个，所以只要构建了这一个 SqlSessionFactory，它的使命就完成了，也就没有存在的意义了。所以它的生命周期只存在于**方法的局部**。

### 2 ) SqlSessionFactory

SqlSessionFactory 是用来创建 SqlSession 的，每次应用程序访问数据库，都需要创建一个会话。因为我们一直有创建会话的需要，所以 SqlSessionFactory 应该存在于应用的整个生命周期中（**作用域是应用作用域**）。创建 SqlSession 只需要一个实例来做这件事就行了，否则会产生很多的混乱，和浪费资源。所以我们要采用单例模式。

### 3 ) SqlSession

SqlSession 是一个会话，因为它不是线程安全的，不能在线程间共享。所以我们在请求开始的时候创建一个 SqlSession 对象，在请求结束或者说方法执行完毕的时候要及时关闭它（**一次请求或者操作中**）。

### 4 ) Mapper

Mapper（实际上是一个代理对象）是从 SqlSession 中获取的。

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

它的作用是发送 SQL 来操作数据库的数据。它应该在一个 SqlSession 事务方法之内。

最后总结如下：

对象	生命周期
SqlSessionFactoryBuilder	方法局部（method）
SqlSessionFactory（单例）	应用级别（application）
SqlSession	请求和操作（request/method）
Mapper	方法（method）

这个就是我们在程式的使用里面看到的四个对象的生命周期的总结。

## 核心配置解读

第一个是 config 文件。大部分时候我们只需要很少的配置就可以让 MyBatis 运行起来。其实 MyBatis 里面提供的配置项非常多，我们没有配置的时候使用的是系统的默认值。

mybatis-3 的源码托管在 github 上。源码地址 <https://github.com/mybatis/mybatis-3/releases>

目前最新的版本是 3.5.1，大家可以从官方上下载到最新的源码。

第一个是 jar 包和文档。第二个第三个是源码。



在这个压缩包里面，解压出来有一个 mybatis-3.5.1.pdf，是英文版本的。如果阅读英文困难，可以基于 3.5.1 的中文版本学习。

<http://www.mybatis.org/mybatis-3/zh/index.html>

## 一级标签

### configuration

configuration 是整个配置文件的根标签，实际上也对应着 MyBatis 里面最重要的配置类 Configuration。它贯穿 MyBatis 执行流程的每一个环节。我们打开这个类看一下，这里面有很多的属性，跟其他的子标签也能对应上。

注意：MyBatis 全局配置文件顺序是固定的，否则启动的时候会报错。

（一级标签要求全部掌握）

### properties

第一个是 properties 标签，用来配置参数信息，比如最常见的数据库连接信息。

为了避免直接把参数写死在 xml 配置文件中，我们可以把这些参数单独放在 properties 文件中，用 properties 标签引入进来，然后在 xml 配置文件中用\${}引用就可以了。

可以用 resource 引用应用里面的相对路径，也可以用 url 指定本地服务器或者网络的绝对路径。

我们为什么要把这些配置独立出来？有什么好处？或者说，公司的项目在打包的时候，有没有把 properties 文件打包进去？

- 1、 提取，利于多处引用，维护简单；
- 2、 把配置文件放在外部，避免修改后重新编译打包，只需要重启应用；
- 3、 程序和配置分离，提升数据的安全性，比如生产环境的密码只有运维人



员掌握。

## settings

settings 里面是 MyBatis 的一些核心配置,我们最后再看,先看下其他的以及标签。

## typeAliases

TypeAlias 是类型的别名,跟 Linux 系统里面的 alias 一样,主要用来简化全路径类名的拼写。比如我们的参数类型和返回值类型都可能会用到我们的 Bean,如果每个地方都配置全路径的话,那么内容就比较多,还可能会写错。

我们可以为自己的 Bean 创建别名,既可以指定单个类,也可以指定一个 package,自动转换。配置了别名以后,只需要写别名就可以了,比如 com.gupaoedu.domain.Blog 都只要写 blog 就可以了。

MyBatis 里面有系统预先定义好的类型别名,在 TypeAliasRegistry 中。

## typeHandlers 【重点】

由于 Java 类型和数据库的 JDBC 类型不是——对应的(比如 String 与 varchar),所以我们将 Java 对象转换为数据库的值,和把数据库的值转换成 Java 对象,需要经过一定的转换,这两个方向的转换就要用到 TypeHandler。

有的同学可能会有疑问,我没有做任何的配置,为什么实体类对象里面的一个 String 属性,可以保存成数据库里面的 varchar 字段,或者保存成 char 字段?

这是因为 MyBatis 已经内置了很多 TypeHandler(在 type 包下),它们全部全部注册在 TypeHandlerRegistry 中,他们都继承了抽象类 BaseTypeHandler,泛型就是要

处理的 Java 数据类型。

- ArrayTypeHandler
- BaseTypeHandler
- BigDecimalTypeHandler
- BigIntegerTypeHandler
- BlobByteObjectArrayTypeHandler
- BlobInputStreamTypeHandler
- BlobTypeHandler
- BooleanTypeHandler
- ByteArrayTypeHandler
- ByteArrayUtils

当我们做数据类型转换的时候，就会自动调用对应的 TypeHandler 的方法。

如果我们需要自定义一些类型转换规则，或者要在处理类型的时候做一些特殊的动作，就可以编写自己的 TypeHandler，跟系统自定义的 TypeHandler 一样，继承抽象类 BaseTypeHandler<T>。有 4 个抽象方法必须实现，我们把它分成两类：

set 方法从 Java 类型转换成 JDBC 类型的，get 方法是从 JDBC 类型转换成 Java 类型的。

从 Java 类型到 JDBC 类型	从 JDBC 类型到 Java 类型
setNonNullParameter: 设置非空参数	getNullableResult: 获取空结果集（根据列名），一般都是调用这个 getNullableResult: 获取空结果集（根据下标值） getNullableResult: 存储过程用的

比如我们想要在获取或者设置 String 类型的时候做一些特殊处理，我们可以写一个 String 类型的 TypeHandler ( mybatis-standalone 工程 )。

```
public class MyTypeHandler extends BaseTypeHandler<String> {
    public void setNonNullParameter(PreparedStatement ps, int i, String parameter,
        JdbcType jdbcType)
        throws SQLException {
        // 设置 String 类型的参数的时候调用，Java 类型到 JDBC 类型
        System.out.println("-----setNonNullParameter1: " + parameter);
        ps.setString(i, parameter);
    }
}
```

```

public String getNullableResult(ResultSet rs, String columnName) throws SQLException
{
    // 根据列名获取 String 类型的参数的时候调用, JDBC 类型到 java 类型
    System.out.println("-----getNullableResult1: "+columnName);
    return rs.getString(columnName);
}

// 后面两个方法省略.....
}

```

第二步，在 mybatis-config.xml 文件中注册：

```

<typeHandlers>
    <typeHandler handler="com.gupaoedu.type.MyTypeHandler"></typeHandler>
</typeHandlers>

```

第三步，在我们需要使用的字段上指定，比如：

插入值的时候，从 Java 类型到 JDBC 类型，在字段属性中指定 typehandler：

```

<insert id="insertBlog" parameterType="com.gupaoedu.domain.Blog">
    insert into blog (bid, name, author_id)
    values (#{bid, jdbcType=INTEGER},
    #{name, jdbcType=VARCHAR, typeHandler=com.gupaoedu.type.MyTypeHandler},
    #{authorId, jdbcType=INTEGER})
</insert>

```

返回值的时候，从 JDBC 类型到 Java 类型，在 resultMap 的列上指定 typehandler：

```

<result column="name" property="name" jdbcType="VARCHAR"
typeHandler="com.gupaoedu.type.MyTypeHandler"/>

```

【思考，不强制要求完成】

如果我们的对象里面有复杂对象，比如 Blog 里面包括了一个 Comment 对象，这个时候 Comment 对象的全部属性不能直接映射到数据库的一个字段。

要求：创建一个 TypeHandler，可以将任意的对象转换为 json 字符串，保存到数据

库的 VARCHAR 类型中。在从数据库查询的时候，再转换为原来的 Java 对象。

- 1、 在数据库表添加一个 VARCHAR 字段；
- 2、 在 Blog 对象中添加一个 Comment 属性，字段 Integer id;String content；
- 3、 JSON 工具没有要求，jackson 或者 fastjson、gson 都可以。
- 4、 在查询和插入的 statement 上使用这个 TypeHandler。

## objectFactory【重点】

当我们把数据库返回的结果集转换为实体类的时候，需要创建对象的实例，由于我们不知道需要处理的类型是什么，有哪些属性，所以不能用 new 的方式去创建。在 MyBatis 里面，它提供了一个工厂类的接口，叫做 ObjectFactory，专门用来创建对象的实例，里面定义了 4 个方法。

```

ObjectFactory
  (m) setProperties(Properties): void
  (m) create(Class<T>): T
  (m) create(Class<T>, List<Class<?>>, List<Object>): T
  (m) isCollection(Class<T>): boolean
  
```

方法	作用
<code>void setProperties(Properties properties);</code>	设置参数时调用
<code>&lt;T&gt; T create(Class&lt;T&gt; type);</code>	创建对象（调用无参构造函数）
<code>&lt;T&gt; T create(Class&lt;T&gt; type, List&lt;Class&lt;?&gt;&gt; constructorArgTypes, List&lt;Object&gt; constructorArgs);</code>	创建对象（调用带参数构造函数）
<code>&lt;T&gt; boolean isCollection(Class&lt;T&gt; type)</code>	判断是否集合

ObjectFactory 有一个默认的实现类 DefaultObjectFactory，创建对象的方法最终都调用了 instantiateClass()，是通过反射来实现的。

如果想要修改对象工厂在初始化实体类的时候的行为，就可以通过创建自己的对象工厂，继承 DefaultObjectFactory 来实现（不需要再实现 ObjectFactory 接口）。

例如：

```
public class GPObjectFactory extends DefaultObjectFactory {
```

```

@Override
public Object create(Class type) {
    if (type.equals(Blog.class)) {
        Blog blog = (Blog) super.create(type);
        blog.setName("by object factory");
        blog.setBid(1111);
        blog.setAuthorId(2222);
        return blog;
    }
    Object result = super.create(type);
    return result;
}
}

```

我们可以直接用自定义的工厂类来创建对象：

```

public class ObjectFactoryTest {
    public static void main(String[] args) {
        GPObjectFactory factory = new GPObjectFactory();
        Blog myBlog = (Blog) factory.create(Blog.class);
        System.out.println(myBlog);
    }
}

```

这样我们就直接拿到了一个对象。

如果在 config 文件里面注册，在创建对象的时候会被自动调用：

```

<objectFactory type="org.mybatis.example.GPObjectFactory">
    <!-- 对象工厂注入的参数 -->
    <property name="gupao" value="666"/>
</objectFactory>

```

这样，就可以让 MyBatis 的创建实体类的时候使用我们自己的对象工厂。

应用场景举例：

比如有一个新锐手机品牌在一个电商平台上面卖货，为了让预约数量好看一点，只要有人预约，预约数量就自动乘以 3。这个时候就可以创建一个 ObjectFactory，只要是查询销量，就把它预约数乘以 3 返回这个实体类。

被发现后，平台：是程序员干的。

附：

1、什么时候调用了 `objectFactory.create()` ？

创建 `DefaultResultSetHandler` 的时候，和创建对象的时候。

2、创建对象后，已有的属性为什么被覆盖了？

在 `DefaultResultSetHandler` 类的 395 行 `getRowValue()` 方法里面里面调用了 `applyPropertyMappings()`。

3、返回结果的时候，`ObjectFactory` 和 `TypeHandler` 哪个先工作？

先是 `ObjectFactory`，再是 `TypeHandler`。肯定是先创建对象。

PS：step out 可以看到一步步调用的层级。

## plugins

插件是 MyBatis 的一个很强大的机制，跟很多其他的框架一样，MyBatis 预留了插件的接口，让 MyBatis 更容易扩展。

根据官方的定义，插件可以拦截这四个对象的这些方法，我们把这四个对象称作 MyBatis 的四大对象。我们会在带大家阅读源码，知道了这 4 大对象的作用之后，再来分析自定义插件的开发和插件运行的原理。

<http://www.mybatis.org/mybatis-3/zh/configuration.html#plugins>

类（或接口）	方法
Executor	update, query, flushStatements, commit, rollback, getTransaction, close, isClosed
ParameterHandler	getParameterObject, setParameters



ResultSetHandler	handleResultSets, handleOutputParameters
StatementHandler	prepare, parameterize, batch, update, query

## environments、environment

environments 标签用来管理数据库的环境，比如我们可以有开发环境、测试环境、生产环境的数据库。可以在不同的环境中使用不同的数据库地址或者类型。

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
      <property name="driver" value="com.mysql.jdbc.Driver"/>
      <property name="url"
value="jdbc:mysql://127.0.0.1:3306/gp-mybatis?useUnicode=true"/>
      <property name="username" value="root"/>
      <property name="password" value="123456"/>
    </dataSource>
  </environment>
</environments>
```

一个 environment 标签就是一个数据源，代表一个数据库。这里面有两个关键的标签，一个是事务管理器，一个是数据源。

## transactionManager

如果配置的是 JDBC，则会使用 Connection 对象的 commit()、rollback()、close() 管理事务。

如果配置成 MANAGED，会把事务交给容器来管理，比如 JBOSS，Weblogic。因为我们跑的是本地程序，如果配置成 MANAGE 不会有任何事务。

如果是 Spring + MyBatis，则没有必要配置，因为我们会直接在 applicationContext.xml 里面配置数据源，覆盖 MyBatis 的配置。

## dataSource

将在下一节（ settings ）详细分析。在跟 Spring 集成的时候，事务和数据源都会交给 Spring 来管理。

## mappers

<mappers> 标签配置的是我们的映射器，也就是 Mapper.xml 的路径。这里配置的目的是让 MyBatis 在启动的时候去扫描这些映射器，创建映射关系。

我们有四种指定 Mapper 文件的方式：

<http://www.mybatis.org/mybatis-3/zh/configuration.html#mappers>

- 1、使用相对于类路径的资源引用（ resource ）
- 2、使用完全限定资源定位符（绝对路径）（ URL ）
- 3、使用映射器接口实现类的完全限定类名
- 4、将包内的映射器接口实现全部注册为映射器（最常用）

思考：

接口跟 statement 是怎么绑定起来的？——method 有方法全限定名，比如：  
com.gupaoedu.mapper.BlogMapper.selectBlogById，跟 namespace 里面的 statement ID 是相同的。

在哪一步拿到 SQL 的？——ms 里面有 SQL。

```
// DefaultSqlSession.selectList()  
MappedStatement ms = configuration.getMappedStatement(statement);
```

## settings

最后 settings 我们来单独说一下，因为 MyBatis 的一些最关键的配置都在这个标签里面（只讲解一些主要的）。

属性名	含义	简介	有效值	默认值
cacheEnabled	是否使用缓存	是整个工程中所有映射器配置缓存的开关，即是一个全局缓存开关	true/false	true
lazyLoadingEnabled	是否开启延迟加载	控制全局是否使用延迟加载（association、collection）。当有特殊关联关系需要单独配置时，可以使用 fetchType 属性来覆盖此配置	true/false	false
aggressiveLazyLoading	是否需要侵入式延迟加载	开启时，无论调用什么方法加载某个对象，都会加载该对象的所有属性，关闭之后只会按需加载	true/false	false
defaultExecutorType	设置默认的执行器	有三种执行器：SIMPLE 为普通执行器；REUSE 执行器会重用与处理语句；BATCH 执行器将重用语句并执行批量更新	SIMPLE/REUSE/BATCH	SIMPLE
lazyLoadTriggerMethods	指定哪个对象的方法触发一次延迟加载	配置需要触发延迟加载的方法的名字，该方法就会触发一次延迟加载	一个逗号分隔的方法名称列表	equals, clone, hashCode, toString
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
logImpl	日志实现	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	无
multipleResultSetsEnabled	是否允许单一语句返回多结果集	即 Mapper 配置中一个单一的 SQL 配置是否能返回多个结果集	true/false	true
useColumnLabel	使用列标签代替列名	设置是否使用列标签代替列名	true/false	true
useGeneratedKeys	是否支持 JDBC 自动生成主键	设置之后，将会强制使用自动生成主键的策略	true/false	false
autoMappingBehavior	指定 MyBatis 自动	有三种方式，NONE 时将取消自动映射；	NONE/PARTIAL/FULL	PARTIAL

	映射字段或属性的方式	PARTIAL 时只会自动映射没有定义结果集的结果映射；FULL 时会映射任意复杂的结果集	L	
autoMappingUnknownColumnBehavior	设置当自动映射时发现未知列的动作	有三种动作：NONE 时不做任何操作；WARNING 时会输出提醒日志；FAILING 时会抛出 SqlSessionException 异常表示映射失败	NONE/WARNING/FAILING	NONE
defaultStatementTimeout	设置超时时间	该超时时间即数据库驱动连接数据库时，等待数据库回应的最大秒数	任意正整数	无
defaultFetchSize	设置驱动的结果集获取数量（fetchSize）的提示值	为了防止从数据库查询出来的结果过多，而导致内存溢出，可以通过设置 fetchSize 参数来控制结果集的数量	任意正整数	无
safeRowBoundsEnabled	允许在嵌套语句中使用分页（RowBound，即行内嵌套语句）	如果允许在 SQL 的行内嵌套语句中使用分页，就设置该值为 false	true/false	false
safeResultHandlerEnabled	允许在嵌套语句中使用分页（ResultHandler，即结果集处理）	如果允许在 SQL 的结果集使用分页，就设置该值为 false	true/false	true
mapUnderscoreToCamelCase	是否开启驼峰命名规则（camel case）映射	表明数据库中的字段名称与工程中 Java 实体类的映射是否采用驼峰命名规则校验	true/false	false
jdbcTypeForNull	JDBC 类型的默认设置	当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER	常用 NULL、VARCHAR、OTHER	OTHER
defaultScriptingLanguage	动态 SQL 默认语言	指定动态 SQL 生成的默认语言	一个类型别名或者一个类的全路径名	org.apache.ibatis.scripting.xmltags.XMLLanguageDriver
callSettersOnNulls	是否在控制情况下调用 Set 方法	指定当结果集中值为 null 时是否调用映射对象的 setter(map 对象时为 put) 方法，这对于有 Map.keySet() 依赖或 null 值初始化时是有用的。注意基本类型是不能设置成 null 的	true/false	false

returnInstanceForEmptyRow	返回空实体集对象	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（从 MyBatis3.4.2 版本开始）	true/false	false
logPrefix	日志前缀	指定 MyBatis 所用日志的具体实现，未指定时将自动查找	任意字符串	无
vfsImpl	vfs 实现	指定 vfs 的实现	自定义 VFS 的实现的类的全限定名，以逗号分隔	无
useActualParamName	使用方法签名	允许使用方法签名中的名称作为语句参数名称。要使用该特性，工程必须采用 Java8 编译，并且加上 -parameters 选项（从 MyBatis3.4.1 版本开始）	自定义 VFS 的实现的类的全限定名，以逗号分隔	无
configurationFactory	配置工厂	指定提供配置示例的类。返回的配置实例用于加载反序列化的懒加载参数。这个类必须有一个签名的静态配置 getconfiguration() 方法（从 MyBatis3.2.3 版本开始）	一个类型别名或者一个类型的全路径名	无

## Mapper.xml 映射配置文件【重点】

<http://www.mybatis.org/mybatis-3/zh/sqlmap-xml.html>

映射器里面最主要的是配置了 SQL 语句，也解决了我们的参数映射和结果集映射的问题。一共有 8 个标签：

cache – 给定命名空间的缓存配置（是否开启二级缓存）。

cache-ref – 其他命名空间缓存配置的引用。这两个标签我们在讲解缓存的时候会详细讲到。

resultMap – 是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。

```

<resultMap id="BaseResultMap" type="Employee">
  <id column="emp_id" jdbcType="INTEGER" property="empId"/>
  <result column="emp_name" jdbcType="VARCHAR" property="empName"/>
  <result column="gender" jdbcType="CHAR" property="gender"/>
  <result column="email" jdbcType="VARCHAR" property="email"/>
  <result column="d_id" jdbcType="INTEGER" property="dId"/>
</resultMap>

```

sql – 可被其他语句引用的可重用语句块。

```

<sql id="Base_Column_List">
  emp_id, emp_name, gender, email, d_id
</sql>

```

增删改查标签：

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

总结

最后我们来总结一下：

配置名称	配置含义	配置简介
configuration	包裹所有配置标签	整个配置文件的顶级标签
properties	属性	该标签可以引入外部配置的属性，也可以自己配置。该配置标签所在的同一个配置文件的其他配置均可以引用此配置中的属性
setting	全局配置参数	用来配置一些改变运行时行为的信息，例如是否使用缓存机制，是否使用延迟加载，是否使用错误处理机制等。
typeAliases	类型别名	用来设置一些别名来代替 Java 的长类型声明（如 java.lang.int 变为 int），减少配置编码的冗余
typeHandlers	类型处理器	将数据库获取的值以合适的方式转换为 Java 类型，或者将 Java 类型的参数转换为数据库对应的类型
objectFactory	对象工厂	实例化目标类的工厂类配置
plugins	插件	可以通过插件修改 MyBatis 的核心行为，例如对语句执行的某一点进行拦截调用



environments	环境集合属性对象	数据库环境信息的集合。在一个配置文件中，可以有多种数据库环境集合，这样可以使 MyBatis 将 SQL 同时映射至多个数据库
environment	环境子属性对象	数据库环境配置的详细配置
transactionManager	事务管理	指定 MyBatis 的事务管理器
dataSource	数据源	使用其中的 type 指定数据源的连接类型，在标签对中可以使用 property 属性指定数据库连接池的其他信息
mappers	映射器	配置 SQL 映射文件的位置，告知 MyBatis 去哪里加载 SQL 映射文件

### 3. MyBatis 最佳实践

以下是一些 MyBatis 的高级用法或者扩展方式，帮助我们更好地使用 MyBatis。

#### 动态 SQL

#### 为什么需要动态 SQL？

我们来看这么一段令人吐血的代码（存储过程）。

第 5363 行



pkg\_cnap\_querybeps19.pck

由于前台传入的查询参数不同，所以写了很多的 if else，还需要非常注意 SQL 语句里面的 and、空格、逗号和转移的单引号这些，拼接和调试 SQL 就是一件非常耗时的工作。

MyBatis 的动态 SQL 就帮助我们解决了这个问题，它是基于 OGNL 表达式的。

#### 动态标签有哪些？

按照官网的分类，MyBatis 的动态标签主要有四类：if, choose (when, otherwise), trim (where, set), foreach。

( 案例在 spring-mybatis 工程中 )

if —— 需要判断的时候，条件写在 test 中

以下语句可以用 <where> 改写

```
<select id="selectDept" parameterType="int"
resultType="com.gupaoedu.crud.bean.Department">
    select * from tbl_dept where 1=1
    <if test="deptId != null">
        and dept_id = #{deptId, jdbcType=INTEGER}
    </if>
</select>
```

choose (when, otherwise) —— 需要选择一个条件的时候

```
<select id="getEmpList_choose" resultMap="empResultMap"
parameterType="com.gupaoedu.crud.bean.Employee">
    SELECT * FROM tbl_emp e
    <where>
        <choose>
            <when test="empId != null">
                e.emp_id = #{emp_id, jdbcType=INTEGER}
            </when>
            <when test="empName != null and empName != ''">
                AND e.emp_name LIKE CONCAT(CONCAT(' %', #{emp_name,
jdbcType=VARCHAR} ), ' %' )
            </when>
            <when test="email != null ">
                AND e.email = #{email, jdbcType=VARCHAR}
            </when>
            <otherwise>
            </otherwise>
        </choose>
    </where>
</select>
```

trim (where, set) —— 需要去掉 where、and、逗号之类的符号的时候。

注意最后一个条件 dId 多了一个逗号，就是用 trim 去掉的：

```
<update id="updateByPrimaryKeySelective"
parameterType="com.gupaoedu.crud.bean.Employee">
    update tbl_emp
```

```

<set>
  <if test="empName != null">
    emp_name = #{empName, jdbcType=VARCHAR},
  </if>
  <if test="gender != null">
    gender = #{gender, jdbcType=CHAR},
  </if>
  <if test="email != null">
    email = #{email, jdbcType=VARCHAR},
  </if>
  <if test="dId != null">
    d_id = #{dId, jdbcType=INTEGER},
  </if>
</set>
where emp_id = #{empId, jdbcType=INTEGER}
</update>

```

trim 用来指定或者去掉前缀或者后缀：

```

<insert id="insertSelective" parameterType="com.gupaoedu.crud.bean.Employee">
  insert into tbl_emp
  <trim prefix="(" suffix=")" suffixOverrides=",">
    <if test="empId != null">
      emp_id,
    </if>
    <if test="empName != null">
      emp_name,
    </if>
    <if test="dId != null">
      d_id,
    </if>
  </trim>
  <trim prefix="values (" suffix=")" suffixOverrides=",">
    <if test="empId != null">
      #{empId, jdbcType=INTEGER},
    </if>
    <if test="empName != null">
      #{empName, jdbcType=VARCHAR},
    </if>
    <if test="dId != null">
      #{dId, jdbcType=INTEGER},
    </if>
  </trim>
</insert>

```

foreach —— 需要遍历集合的时候：

```
<delete id="deleteByList" parameterType="java.util.List">
  delete from tbl_emp where emp_id in
  <foreach collection="list" item="item" open="(" separator="," close=")">
    #{item.empId, jdbcType=VARCHAR}
  </foreach>
</delete>
```

动态 SQL 主要是用来解决 SQL 语句生成的问题。

## 批量操作

( spring-mybatis 工程单元测试目录，MapperTest 类 )

我们在生产的项目中会有一些批量操作的场景，比如导入文件批量处理数据的情况（批量新增商户、批量修改商户信息），当数据量非常大，比如超过几万条的时候，在 Java 代码中循环发送 SQL 到数据库执行肯定是不现实的，因为这个意味着要跟数据库创建几万次会话，即使我们使用了数据库连接池技术，对于数据库服务器来说也是不堪重负的。

在 MyBatis 里面是支持批量的操作的，包括批量的插入、更新、删除。我们可以直接传入一个 List、Set、Map 或者数组，配合动态 SQL 的标签，MyBatis 会自动帮我们生成语法正确的 SQL 语句。

比如我们来看两个例子，批量插入和批量更新。

## 批量插入

批量插入的语法是这样的，只要在 values 后面增加插入的值就可以了。

```
insert into tbl_emp (emp_id, emp_name, gender,email, d_id) values ( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? ),
( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? ), ( ?,?,?,?,? )
```

在 Mapper 文件里面，我们使用 foreach 标签拼接 values 部分的语句：

```
<!-- 批量插入 -->
<insert id="batchInsert" parameterType="java.util.List" useGeneratedKeys="true">
  <selectKey resultType="long" keyProperty="id" order="AFTER">
    SELECT LAST_INSERT_ID()
  </selectKey>
  insert into tbl_emp (emp_id, emp_name, gender, email, d_id)
  values
  <foreach collection="list" item="emps" index="index" separator=",">
    ( #{emps.empId}, #{emps.empName}, #{emps.gender}, #{emps.email}, #{emps.dId} )
  </foreach>
</insert>
```

Java 代码里面，直接传入一个 List 类型的参数。

我们来测试一下。效率要比循环发送 SQL 执行要高得多。最关键的地方就在于减少了跟数据库交互的次数，并且避免了开启和结束事务的时间消耗。

## 批量更新

批量更新的语法是这样的，通过 case when，来匹配 id 相关的字段值。

```
update tbl_emp set
emp_name =
  case emp_id
    when ? then ?
    when ? then ?
    when ? then ? end ,
gender =
  case emp_id
    when ? then ?
    when ? then ?
    when ? then ? end ,
email =
  case emp_id
    when ? then ?
    when ? then ?
    when ? then ? end
where emp_id in ( ?, ?, ? )
```

所以在 Mapper 文件里面最关键的就是 case when 和 where 的配置。

需要注意一下 open 属性和 separator 属性。

```
<update id="updateBatch">
  update tbl_emp set
  emp_name =
  <foreach collection="list" item="emps" index="index" separator=" " open="case emp_id"
close="end">
    when #{emps.empId} then #{emps.empName}
  </foreach>
  , gender =
  <foreach collection="list" item="emps" index="index" separator=" " open="case emp_id"
close="end">
    when #{emps.empId} then #{emps.gender}
  </foreach>
  , email =
  <foreach collection="list" item="emps" index="index" separator=" " open="case emp_id"
close="end">
    when #{emps.empId} then #{emps.email}
  </foreach>
  where emp_id in
  <foreach collection="list" item="emps" index="index" separator="," open="("
close=")">
    #{emps.empId}
  </foreach>
</update>
```

批量删除也是类似的。

## Batch Executor

当然 MyBatis 的动态标签的批量操作也是存在一定的缺点的，比如数据量特别大的时候，拼接出来的 SQL 语句过大。

MySQL 的服务端对于接收的数据包有大小限制，max\_allowed\_packet 默认是 4M，需要修改默认配置才可以解决这个问题。

Caused by: com.mysql.jdbc.PacketTooBigException: Packet for query is too large (7188967 > 4194304). You can change this value on the server by setting the max\_allowed\_packet' variable.

在我们的全局配置文件中，可以配置默认的 Executor 的类型。其中有一种 BatchExecutor。



```
<setting name="defaultExecutorType" value="BATCH" />
```

也可以在创建会话的时候指定执行器类型：

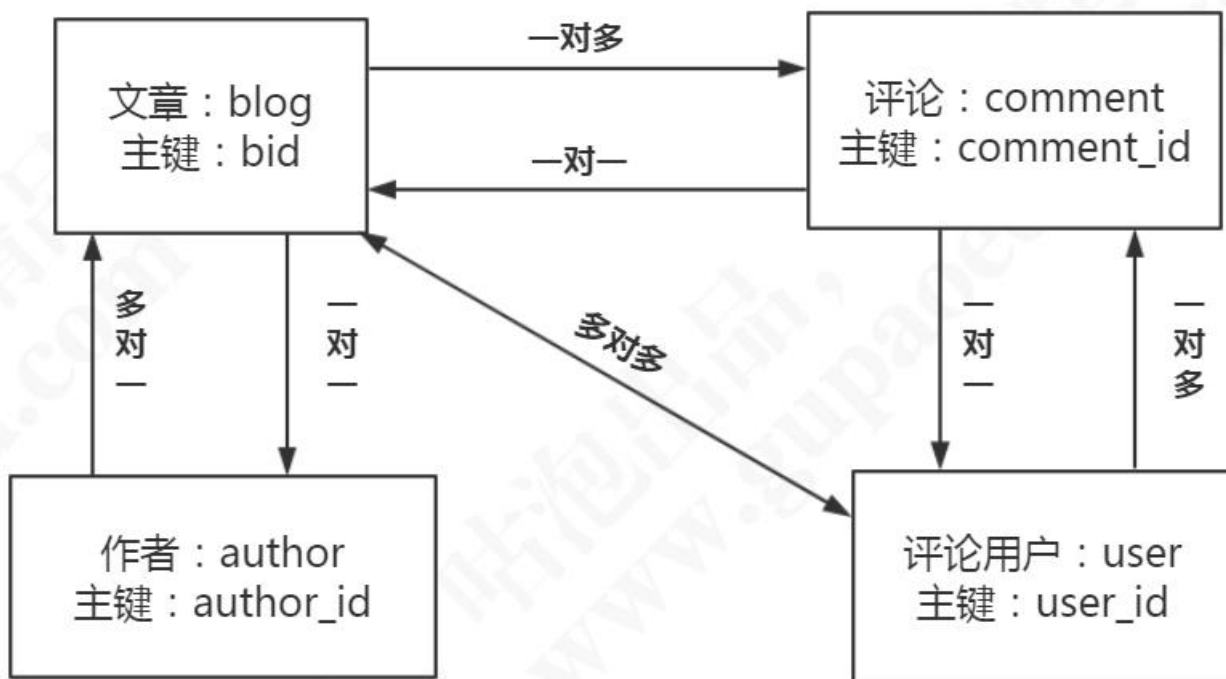
```
SqlSession session = sqlSessionFactory.openSession(ExecutorType.BATCH);
```

BatchExecutor 底层是对 JDBC ps.addBatch() 的封装，原理是攒一批 SQL 以后再发送（参考 standalone - 单元测试目录 JdbcTest.java – testJdbcBatch()）。

问题：三种执行器的区别是什么？Simple、Reuse、Batch

嵌套（关联）查询 / N+1 / 延迟加载

我们在查询业务数据的时候经常会遇到跨表关联查询的情况，比如查询员工就会关联部门（一对一），查询成绩就会关联课程（一对一），查询订单就会关联商品（一对多），等等。



我们映射结果有两个标签，一个是 resultType，一个是 resultMap。

resultType 是 select 标签的一个属性，适用于返回 JDK 类型（比如 Integer、String 等等）和实体类。这种情况下结果集的列和实体类的属性可以直接映射。如果返回的字

段无法直接映射，就要用 resultMap 来建立映射关系。

对于关联查询的这种情况，通常不能用 resultType 来映射。用 resultMap 映射，要么就是修改 dto ( Data Transfer Object )，在里面增加字段，这个会导致增加很多无关的字段。要么就是引用关联的对象，比如 Blog 里面包含了一个 Author 对象，这种情况下就要用到关联查询 ( association，或者嵌套查询 )，MyBatis 可以帮我们自动做结果的映射。

一对一的关联查询有两种配置方式：

### 1、嵌套结果：

( mybatis-standalone - MyBatisTest - testSelectBlogWithAuthorResult () )

```
<!-- 根据文章查询作者，一对一查询的结果，嵌套查询 -->
<resultMap id="BlogWithAuthorResultMap"
type="com.gupaoedu.domain.associate.BlogAndAuthor">
  <id column="bid" property="bid" jdbcType="INTEGER"/>
  <result column="name" property="name" jdbcType="VARCHAR"/>
  <!-- 联合查询，将 author 的属性映射到 ResultMap -->
  <association property="author" javaType="com.gupaoedu.domain.Author">
    <id column="author_id" property="authorId"/>
    <result column="author_name" property="authorName"/>
  </association>
</resultMap>
```

### 2、嵌套查询：

( mybatis-standalone - MyBatisTest - testSelectBlogWithAuthorQuery () )

```
<!-- 另一种联合查询 (一对一)的实现，但是这种方式有 "N+1" 的问题 -->
<resultMap id="BlogWithAuthorQueryMap" type="com.gupaoedu.domain.associate.BlogAndAuthor">
  <id column="bid" property="bid" jdbcType="INTEGER"/>
  <result column="name" property="name" jdbcType="VARCHAR"/>
  <association property="author" javaType="com.gupaoedu.domain.Author"
    column="author_id" select="selectAuthor"/> <!-- selectAuthor 定义在下面 -->
</resultMap>
```

```
<!-- 嵌套查询 -->
```

```
<select id="selectAuthor" parameterType="int" resultType="com.gupaoedu.domain.Author">
    select author_id authorId, author_name authorName
    from author where author_id = #{authorId}
</select>
```

其中第二种方式：嵌套查询，由于是分两次查询，当我们查询了员工信息之后，会再发送一条 SQL 到数据库查询部门信息。

我们只执行了一次查询员工信息的 SQL（所谓的 1），如果返回了 N 条记录，就会再发送 N 条到数据库查询部门信息（所谓的 N），这个就是我们所说的 N+1 的问题。这样会白白地浪费我们的应用和数据库的性能。

如果我们用了嵌套查询的方式，怎么解决这个问题？能不能等到使用部门信息的时候再去查询？这个就是我们所说的延迟加载，或者叫懒加载。

在 MyBatis 里面可以通过开启延迟加载的开关来解决这个问题。

在 settings 标签里面可以配置：

```
<!--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。默认 false -->
<setting name="lazyLoadingEnabled" value="true"/>
<!--当开启时，任何方法的调用都会加载该对象的所有属性。默认 false，可通过 select 标签的
fetchType 来覆盖-->
<setting name="aggressiveLazyLoading" value="false"/>
<!-- Mybatis 创建具有延迟加载能力的对象所用到的代理工具，默认 JAVASSIST -->
<setting name="proxyFactory" value="CGLIB" />
```

lazyLoadingEnabled 决定了是否延迟加载。

aggressiveLazyLoading 决定了是不是对象的所有方法都会触发查询。

先来测试一下（也可以改成查询列表）：

1、没有开启延迟加载的开关，会连续发送两次查询；

2、开启了延迟加载的开关，调用 blog.getAuthor() 以及默认的 (equals, clone, hashCode, toString) 时才会发起第二次查询，其他方法并不会触发查

询，比如 `blog.getName()`；

3、如果开启了 `aggressiveLazyLoading=true`，其他方法也会触发查询，比如 `blog.getName()`。

问题：为什么可以做到延迟加载？`blog.getAuthor()`，只是一个获取属性的方法，里面并没有连接数据库的代码，为什么会触发对数据库的查询呢？

我怀疑：`blog` 根本不是 `Blog` 对象，而是被人动过了手脚！

把这个对象打印出来看看：

```
System.out.println(blog.getClass());
```

果然不对：

```
class com.gupaoedu.domain.associate.BlogAndAuthor_$$jvst70_0
```

这个类的名字后面有 `jvst`，是 `JAVASSIST` 的缩写。原来到这里带延迟加载功能的对象 `blog` 已经变成了一个代理对象，那到底什么时候变成代理对象的？我们后面在看源码的时候再去分析，这个也先留一个作业给大家。

【问题】当开启了延迟加载的开关，对象是怎么变成代理对象的？

`DefaultResultSetHandler.createResultObject()`

既然是代理对象，那么必须要有一种创建代理对象的方法。我们有哪些实现动态代理的方式？

这个就是为什么 `settings` 里面提供了一个 `ProxyFactory` 属性。`MyBatis` 默认使用 `JAVASSIST` 创建代理对象。也可以改为 `CGLIB`，这时需要引入 `CGLIB` 的包。

【问题】`CGLIB` 和 `JAVASSIST` 区别是什么？

测试一下，我们把默认的 JAVASSIST 修改为 CGLIB，再打印这个对象。

### 【问题】

- 1、resultType 和 resultMap 的区别？
- 2、collection 和 association 的区别？

MBG 与 Example

<https://github.com/mybatis/generator>

我们在项目中使用 MyBatis 的时候，针对需要操作的一张表，需要创建**实体类**、**Mapper 映射器**、**Mapper 接口**，里面又有很多的字段和方法的配置，这部分的工作是非常繁琐的。而大部分时候我们对于表的操作是相同的，比如根据主键查询、根据 Map 查询、单条插入、批量插入、根据主键删除等等等等。当我们的表很多的时候，意味着有大量的重复工作。所以有没有一种办法，可以根据我们的表，自动生成**实体类**、**Mapper 映射器**、**Mapper 接口**，里面包含了我们需要用到的这些基本方法和 SQL 呢？

给大家看一个类文件（DBToJavaVO.java），这个是我以前到一个公司的时候，项目里面用的 Hibernate+Oracle，当时是我第一次用 Hibernate。我发现 PO 类和 VO 类的格式基本上都是一样的，属性跟表字段一一对应，主要有几个区别：

- 1、类里面的属性都是表里面的字段，然后定义 getter()、setter() 方法；
- 2、数据库字段的下划线命名，要改成驼峰命名。
- 3、PO 上面要加 @Table、@Id、@Column 的注解；VO 不用；
- 4、数据库的常见类型，要改成 Java 类型，比如 varchar，要对应成 Java 的 String 类型。

当时我就灵机一动，能不能写一个根据数据库的表自动生成 PO 和 VO 类的工具呢？



这样新建表的时候就不用一个一个写字段，一个一个加注解了。于是我就写了这个类。后来我才知道有个东西叫 Hibernate 逆向工程。

MyBatis 也提供了一个这样的东西，叫做 MyBatis Generator，简称 MBG。我们只需要修改一个配置文件，使用相关的 jar 包命令或者 Java 代码就可以帮助我们生成**实体类、映射器和接口文件**。不知道用 MyBatis 的同学有没有跟当年的我一样，还是实体类的一个一个字段，接口的一个一个方法，映射器的一条一条 SQL 去写的。

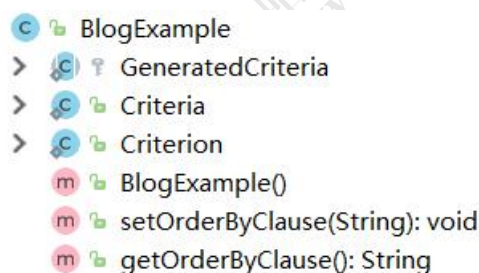
MBG 的配置文件里面有一个 Example 的开关，这个东西用来构造**复杂的筛选条件**的，换句话说就是根据我们的代码去生成 where 条件（类似于 Tom 老师的自动生成 where 条件的方式）。

原理：在实体类中包含了两个有继承关系的 Criteria，用其中自动生成的方法来构建查询条件。把这个包含了 Criteria 的实体类作为参数传到查询参数中，在解析 Mapper 映射器的时候会转换成 SQL 条件。

（mybatis-standalone 工程：

```
com.gupaoedu.domain.BlogExample  
com.gupaoedu.BlogExampleTest )
```

BlogExample 里面包含了一个两个 Criteria：



```
BlogExample  
  GeneratedCriteria  
  Criteria  
  Criterion  
  BlogExample()  
  setOrderByClause(String): void  
  getOrderByClause(): String
```



实例：查询 bid=1 的 Blog，通过创建一个 Criteria 去构建查询条件：

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
BlogExample example = new BlogExample();
BlogExample.Criteria criteria = example.createCriteria();
criteria.andBidEqualTo(1);
List<Blog> list = mapper.selectByExample(example);
```

生成的语句：

```
select 'true' as QUERYID, bid, name, author_id from blog WHERE ( bid = ? )
```

## 翻页

在写存储过程的年代，翻页也是一件很难调试的事情，我们要实现数据不多不少准确地返回，需要大量的调试和修改。但是如果自己手写过分页，就能清楚分页的原理。

### 逻辑翻页与物理翻页

在我们查询数据库的操作中，有两种翻页方式，一种是逻辑翻页（假分页），一种是物理翻页（真分页）。逻辑翻页的原理是把所有数据查出来，在内存中删选数据。物理翻页是真正的翻页，比如 MySQL 使用 limit 语句，Oracle 使用 rownum 语句，SQL Server 使用 top 语句。

### 逻辑翻页

MyBatis 里面有一个逻辑分页对象 RowBounds，里面主要有两个属性，offset 和 limit（从第几条开始，查询多少条）。

我们可以在 Mapper 接口的方法上加上这个参数 不需要修改 xml 里面的 SQL 语句。

```
public List<Blog> selectBlogList(RowBounds rowBounds);
```

使用：mybatis-standalone- MyBatisTest-testSelectByRowBounds()

```

int start = 10; // offset, 从第几行开始查询
int pageSize = 5; // limit, 查询多少条
RowBounds rb = new RowBounds(start, pageSize);
List<Blog> list = mapper.selectBlogList(rb);
for(Blog b :list){
    System.out.println(b);
}

```

它的底层其实是对 `ResultSet` 的处理。它会舍弃掉前面 `offset` 条数据，然后再取剩下的数据的 `limit` 条。

```

// DefaultResultSetHandler.java
private void handleRowValuesForSimpleResultMap(ResultSetWrapper rsw, ResultMap resultMap,
ResultHandler<?> resultHandler, RowBounds rowBounds, ResultMapping parentMapping) throws
SQLException {
    DefaultResultContext<Object> resultContext = new DefaultResultContext();
    ResultSet resultSet = rsw.getResultSet();
    this.skipRows(resultSet, rowBounds);

    while(this.shouldProcessMoreRows(resultContext, rowBounds) && !resultSet.isClosed() &&
resultSet.next()) {
        ResultMap discriminatedResultMap = this.resolveDiscriminatedResultMap(resultSet,
resultMap, (String)null);
        Object rowValue = this.getRowValue(rsw, discriminatedResultMap, (String)null);
        this.storeObject(resultHandler, resultContext, rowValue, parentMapping, resultSet);
    }
}

```

很明显，如果数据量大的话，这种翻页方式效率会很低（跟查询到内存中再使用 `subList(start,end)` 没什么区别）。所以我们要用到物理翻页。

## 物理翻页

物理翻页是真正的翻页，它是通过数据库支持的语句来翻页。

第一种简单的办法就是传入参数（或者包装一个 `page` 对象），在 SQL 语句中翻页。

```
<select id="selectBlogPage" parameterType="map" resultMap="BaseResultMap">
```

```
select * from blog limit #{curIndex} , #{pageSize}
</select>
```

第一个问题是要我们在 Java 代码里面去计算起止序号；第二个问题是：每个需要翻页的 Statement 都要编写 limit 语句，会造成 Mapper 映射器里面很多代码冗余。

那我们就需要一种通用的方式，不需要去修改配置的任何一条 SQL 语句，只要在我们需要翻页的地方封装一下翻页对象就可以了。

我们最常用的做法就是使用翻页的插件，这个是基于 MyBatis 的拦截器实现的，比如 PageHelper。

```
// pageSize 每一页几条
PageHelper.startPage(pn, 10);
List<Employee> emps = employeeService.getAll();
// navigatePages 导航页码数
PageInfo page = new PageInfo(emps, 10);
return Msg.success().add("pageInfo", page);
```

PageHelper 是通过 MyBatis 的拦截器实现的，插件的具体原理我们后面的课再分析。简单地来说，它会根据 PageHelper 的参数，改写我们的 SQL 语句。比如 MySQL 会生成 limit 语句，Oracle 会生成 rownum 语句，SQL Server 会生成 top 语句。

## 通用 Mapper

问题：当我们的表字段发生变化的时候，我们需要修改实体类和 Mapper 文件定义的字段和方法。如果是增量维护，那么一个个文件去修改。如果是全量替换，我们还要去对比用 MBG 生成的文件。字段变动一次就要修改一次，维护起来非常麻烦。

解决这个问题，我们有两种思路。

第一个，因为 MyBatis 的 Mapper 是支持继承的（见：

<https://github.com/mybatis/mybatis-3/issues/35> )。所以我们可以把我们的 Mapper.xml 和 Mapper 接口都分成两个文件。一个是 MBG 生成的,这部分是固定不变的。然后创建 DAO 类继承生成的接口,变化的部分就在 DAO 里面维护。

mybatis-standalone 工程：

```
public interface BlogMapperExt extends BlogMapper {
    public Blog selectBlogByName(String name);
}
```

```
<?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.gupaoedu.mapper.BlogMapperExt">
    <!-- 只能继承 statement, 不能继承 sql、resultMap 等标签 -->
    <resultMap id="BaseResultMap" type="com.gupaoedu.domain.Blog">
        <id column="bid" property="bid" jdbcType="INTEGER"/>
        <result column="name" property="name" jdbcType="VARCHAR"/>
        <result column="author_id" property="authorId" jdbcType="INTEGER"/>
    </resultMap>

    <!-- 在 parent xml 和 child xml 的 statement id 相同的情况下, 会使用 child xml 的 statement id -->
    <select id="selectBlogByName" resultMap="BaseResultMap" statementType="PREPARED">
        select * from blog where name = #{name}
    </select>
</mapper>
```

mybatis-config.xml 里面也要扫描：

```
<mappers>
    <mapper resource="BlogMapper.xml"/>
    <mapper resource="BlogMapperExt.xml"/>
</mappers>
```

所以以后只要修改 Ext 的文件就可以了。这么做有一个缺点,就是文件会增多。

思考：既然针对每张表生成的基本方法都是一样的,也就是公共的方法部分代码都是一样的,我们能不能把这部分合并成一个文件,让它支持泛型呢？

太聪明了，当然可以！

编写一个支持泛型的通用接口，比如叫 `GPBaseMapper<T>`，把实体类作为参数传入。这个接口里面定义了大量的增删改查的基础方法，这些方法都是支持泛型的。

自定义的 Mapper 接口继承该通用接口，例如 `BlogMapper extends GPBaseMapper<Blog>`，自动获得对实体类的操作方法。遇到没有的方法，我们依然可以在我们自己的 Mapper 里面编写。

我们能想到的解决方案，早就有人做了这个事了，这个东西就叫做通用 Mapper。

<https://github.com/abel533/Mybatis-3-Plugin/wiki>

用途：主要解决单表的增删改查问题，并不适用于多表关联查询的场景。

除了配置文件变动的问题之外，通用 Mapper 还可以解决：

- 1、 每个 Mapper 接口中大量的重复方法的定义；
- 2、 屏蔽数据库的差异；
- 3、 提供批量操作的方法；
- 4、 实现分页。

通用 Mapper 和 PageHelper 作者是同一个人（刘增辉）。

使用方式：在 Spring 中使用时，引入 jar 包，替换 `applicationContext.xml` 中的 `sqlSessionFactory` 和 `configure`。

```
<bean class="tk.mybatis.spring.mapper.MapperScannerConfigurer">
  <property name="basePackage" value="com.gupaoedu.crud.dao"/>
</bean>
```

## MyBatis-Plus

<https://mybatis.plus/guide>

MyBatis-Plus 是原生 MyBatis 的一个增强工具，可以在使用原生 MyBatis 的所有功能的基础上，使用 plus 特有的功能。

MyBatis-Plus 的核心功能：

**通用 CRUD**：定义好 Mapper 接口后，只需要继承 BaseMapper<T> 接口即可获得通用的增删改查功能，无需编写任何接口方法与配置文件。

**条件构造器**：通过 EntityWrapper<T>（实体包装类），可以用于拼接 SQL 语句，并且支持排序、分组查询等复杂的 SQL。

**代码生成器**：支持一系列的策略配置与全局配置，比 MyBatis 的代码生成更好用。

另外 MyBatis-Plus 也有分页的功能。

作者：咕泡学院-青山

最后更新时间：2019 年 4 月 28 日 11:15:42