

课程目标

内容定位

1. 课程大纲

1.1. 为什么要用MyBatis

1.1.1. JDBC连接数据库

1.1.2. Apache DbUtils

1.1.3. Spring JDBC

1.1.4. Hibernate

1.1.5. MyBatis

1.2. MyBatis实际使用案例

1.2.1. 编程式使用

1.2.2. 核心对象的生命周期

1.2.2.1. SqlSessionFactoryBuiler

1.2.2.2. SqlSessionFactory（单例）

1.2.2.3. SqlSession

1.2.2.4. Mapper

1.2.3. 核心配置解读

1.2.3.1. 一级标签

1.2.3.1.1. properties

1.2.3.1.2. settings

1.2.3.1.3. typeAliases

1.2.3.1.4. typeHandlers

1.2.3.1.5. objectFactory

1.2.3.1.6. plugins

1.2.3.1.7. environments、environment

1.2.3.1.8. mappers

1.2.3.1.9. settings

1.2.3.2. Mapper.xml映射配置文件

1.2.3.2.1.

1.2.3.2.2.

1.2.3.2.3.

1.2.3.2.4.

1.2.3.2.5. 增删改查标签

1.2.3.3. 总结

1.3. MyBatis最佳实践

1.3.1. 动态SQL

1.3.1.1. 为什么需要动态SQL?

1.3.1.2. 动态标签有哪些?

1.3.1.2.1. if——需要判断的时候, 条件写在test中

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

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

1.3.1.2.4. trim-用来指定或者去掉前缀或者后缀

1.3.1.2.5 foreach——需要遍历集合的时候:

1.3.2. 批量操作

1.3.2.1. 批量插入

1.3.2.2. 批量更新

1.3.2.3. Batch Executor

1.3.3. 嵌套 (关联) 查询/N+1/延迟加载

1.3.4. 翻页

1.3.4.1. 逻辑翻页

1.3.4.2. 物理翻页

1.3.5. MBG与Example

1.3.6. 通用Mapper

1.3.7. MyBatis-Plus

1.4. MyBatis常见问题

1.4.1. 用注解还是用xml配置?

1.4.2. Mapper接口无法注入或Invalid bound statement (not found)

1.4.3. 怎么获取插入的最新自动生成的ID

1.4.4. 什么时候用#{}, 什么时候用\${}?

1.4.5. XML中怎么使用特殊符号, 比如小于&

1.4.6. 如何实现模糊查询LIKE

2. 补充

2.1. Mybatis-generator使用

2.1.0. 准备测试SQL

2.1.1. 新建工程mybatis-generator-lesson

2.1.2. 引入依赖

2.1.3. 创建配置文件generator-config.xml

2.1.4. 执行 mybatis-generator:generate

2.1.5. 查看生成的文件

2.1.6. 测试

2.2. SpringBoot整合Mybatis

2.2.0. 准备测试SQL

2.2.1. 新建工程springboot-mybatis-lesson

2.2.2. 引入依赖
2.2.3. 创建entity
2.2.4. 创建Mapper接口
2.2.5. 创建Service
2.2.6. 创建Controller
2.2.7. 创建主配置文件application.yml
2.2.8. 创建Mapper映射文件
2.2.9. 创建启动类
2.2.10. 启动应用, 访问接口<http://localhost:8080/blog/list?bid=1>

课程目标

- 1、了解ORM框架发展历史，了解MyBatis 特性
- 2、掌握MyBatis 编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。

注：除了Spring JDBC之外，本章所有代码都在mybatis-standalone工程中。

1. 课程大纲

1.1. 为什么要用MyBatis

1.1.1. JDBC连接数据库

本地的mybatis 数据库和一张blog表。

建表SQL:

```
1. CREATE TABLE `blog` (  
2.   `bid` int(11) NOT NULL AUTO_INCREMENT,  
3.   `name` varchar(255) DEFAULT NULL,  
4.   `author_id` int(11) DEFAULT NULL,  
5.   PRIMARY KEY (`bid`)  
6. ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;  
7.  
8. CREATE TABLE `author` (  
9.   `author_id` int(16) NOT NULL AUTO_INCREMENT,  
10.  `author_name` varchar(255) DEFAULT NULL,  
11.  PRIMARY KEY (`author_id`)  
12. ) ENGINE=InnoDB AUTO_INCREMENT=1002 DEFAULT CHARSET=utf8;  
13.  
14. CREATE TABLE `comment` (  
15.   `comment_id` int(16) NOT NULL AUTO_INCREMENT,  
16.   `content` varchar(255) DEFAULT NULL,  
17.   `bid` int(16) DEFAULT NULL,  
18.   PRIMARY KEY (`comment_id`)  
19. ) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;  
20.  
21. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (1, 'RabbitMQ延时消息',  
22. 1001);  
22. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (2, 'MyBatis源码分析',  
23. 1008);  
23. INSERT INTO `author` (`author_id`, `author_name`) VALUES (1001, '青山');  
24. INSERT INTO `comment` (`comment_id`, `content`, `bid`) VALUES (1, '写得真好, 学习  
25. 了', 1);  
25. INSERT INTO `comment` (`comment_id`, `content`, `bid`) VALUES (2, '刚好碰到这个问  
题, 谢谢', 1);
```

测试类(参见mybatis-standalone-lesson工程cn.sitedev.JdbcTest)

```
1.  @Test
2.  public void testJdbc() throws IOException {
3.      Connection conn = null;
4.      Statement stmt = null;
5.      Blog blog = new Blog();
6.
7.      try {
8.          // 注册 JDBC 驱动
9.          // Class.forName("com.mysql.jdbc.Driver");
10.
11.         // 打开连接
12.         conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/sitedev-mybatis", "root",
"root");
13.
14.         // 执行查询
15.         stmt = conn.createStatement();
16.         String sql = "SELECT bid, name, author_id FROM blog where bid = 1";
17.         ResultSet rs = stmt.executeQuery(sql);
18.
19.         // 获取结果集
20.         while (rs.next()) {
21.             Integer bid = rs.getInt("bid");
22.             String name = rs.getString("name");
23.             Integer authorId = rs.getInt("author_id");
24.             blog.setAuthorId(authorId);
25.             blog.setBid(bid);
26.             blog.setName(name);
27.         }
28.         System.out.println(blog);
29.
30.         rs.close();
31.         stmt.close();
32.         conn.close();
33.     } catch (SQLException se) {
34.         se.printStackTrace();
35.     } catch (Exception e) {
36.         e.printStackTrace();
37.     } finally {
38.         try {
39.             if (stmt != null) stmt.close();
40.         } catch (SQLException se2) {
41.         }
42.         try {
43.             if (conn != null) conn.close();
44.         } catch (SQLException se) {
45.             se.printStackTrace();
46.         }
47.     }
```

分为四步：

首先，在pom.xml中引入MySQL驱动的依赖。

第一步，Class.forName 注册驱动。

第二步，获取一个Connection。第三步，创建一个Statement对象。

第四步，execute()方法执行SQL。execute()方法返回一个ResultSet 结果集。

第五步，通过ResultSet获取数据，给POJO的属性赋值。

最后，关闭数据库相关的资源，包括ResultSet、Statement、Connection。

如果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，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。

对于结果集的处理，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：

```
1. Integer bid = rs.getInt("bid");
2. String name = rs.getString("name");
3. Integer authorId = rs.getInt("author_id");
4. blog.setAuthorId(authorId);
5. blog.setBid(bid);
6. blog.setName(name);
```

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。

总结一下：

- 1、重复代码
- 2、资源管理
- 3、结果集处理
- 4、SQL耦合

这些问题怎么解决呢？写一个工具类，把重复的代码和资源管理的代码封装进去，只要传SQL就行了。

Apache 在2003年的时候发布了一个叫Commons DbUtils的工具类，可以简化对数据库的操作。

1.1.2. Apache DbUtils

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

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。

先创建一个QueryRunner对象。

在QueryRunner的构造函数里面，可以传入一个数据源，比如这里用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。

参见mybatis-standalone-lesson工程cn.sitedev.dbutils.HikariUtil

```
1.     private static HikariDataSource dataSource;
2.     private static QueryRunner queryRunner;
3.
4.     public static void init() {
5.         HikariConfig config = new HikariConfig(PROPERTY_PATH);
6.         dataSource = new HikariDataSource(config);
7.         // 使用数据源初始化QueryRunner
8.         queryRunner = new QueryRunner(dataSource);
9.     }
```

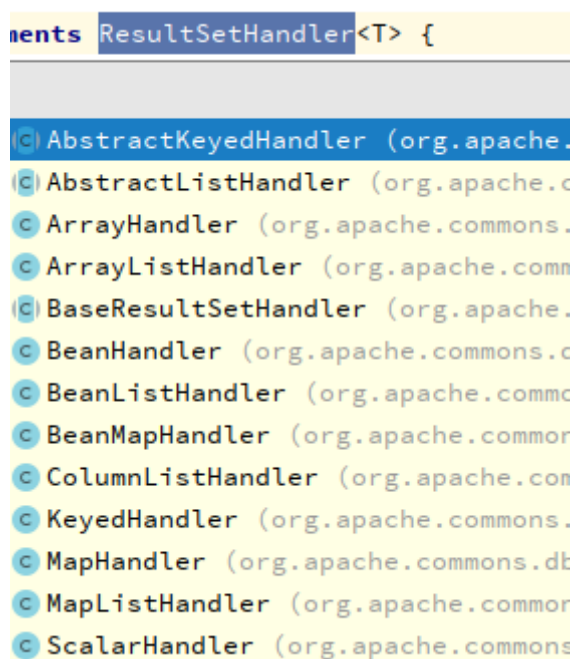
我们通过这个工具类，提供获取QueryRunner实例的方法。方法封装解决了重复代码的问题，传入数据源解决了资源管理的问题。

怎么把结果集转换成对象呢？比如转换成POJO或者List或者Map？肯定不能一个个属性去set或者put了。

我希望做到的是，只要指定一个类型，它就可以自动把结果集给我转换成这种类型。

这个怎么实现呢？

为了避免给每种类型创建一个自动转换类，在DbUtils 里面提供了一系列的支持泛型的ResultSetHandler，比如用来把结果集转换成JavaBean的，转换成List的，转换成Map的，等等。



The screenshot shows a code editor with a list of ResultSetHandler subclasses. The first line is 'ResultSetHandler<T> {'. Below it, a list of subclasses is shown, each with a package name in parentheses: AbstractKeyedHandler (org.apache.commons.dbutils), AbstractListHandler (org.apache.commons.dbutils), ArrayHandler (org.apache.commons.dbutils), ArrayListHandler (org.apache.commons.dbutils), BaseResultSetHandler (org.apache.commons.dbutils), BeanHandler (org.apache.commons.dbutils), BeanListHandler (org.apache.commons.dbutils), BeanMapHandler (org.apache.commons.dbutils), ColumnListHandler (org.apache.commons.dbutils), KeyedHandler (org.apache.commons.dbutils), MapHandler (org.apache.commons.dbutils), MapListHandler (org.apache.commons.dbutils), and ScalarHandler (org.apache.commons.dbutils).

只要在DAO层调用QueryRunner封装好的查询方法，传入一个指定了类型的Handler，它就可以自动把结果集转换成实体类Bean 或者List或者Map。

比如，传入一个BeanHandler 或者BeanListHandler：

参见mybatis-standalone-lesson工程cn.sitedev.dbutils.dao.BlogDao

```
1. // 返回单个对象, 通过new BeanHandler<>(Class<?> clazz)来设置封装
2. public static BlogDto selectBlog(Integer bid) throws SQLException {
3.     String sql = "SELECT * FROM blog WHERE bid = ?";
4.     Object[] params = new Object[]{bid};
5.     BlogDto blogDto = queryRunner.query(sql, new BeanHandler<>(BlogDto.class),
    params);
6.     return blogDto;
7. }
8.
9. // 返回列表, 通过new BeanListHandler<>(Class<?> clazz)来设置List的泛型
10. public static void selectList() throws SQLException {
11.     String sql = "SELECT * FROM blog";
12.     List<BlogDto> list = queryRunner.query(sql, new BeanListHandler<>
    (BlogDto.class));
13.     list.forEach(System.out::println);
14. }
```

测试一下, 结果:

参见mybatis-standalone-lesson工程cn.sitedev.dbutils.QueryRunnerTest

```
1. package cn.sitedev.dbutils;
2.
3. import cn.sitedev.dbutils.dao.BlogDao;
4.
5. import java.sql.SQLException;
6.
7. public class QueryRunnerTest {
8.     public static void main(String[] args) throws SQLException {
9.         HikariUtil.init();
10.        System.out.println(BlogDao.selectBlog(1));
11.        // Language Level 设置成Java 8
12.        BlogDao.selectList();
13.    }
14. }
```



```
Run: QueryRunnerTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
BlogDto(bid=1, name=RabbitMQ延时消息, authorId=null)
BlogDto(bid=1, name=RabbitMQ延时消息, authorId=null)
BlogDto(bid=2, name=MyBatis源码分析, authorId=null)
Process finished with exit code 0
```

它到底是怎么实现的呢? 以BeanHandler的handle()方法为例。


```

1.     public T handle(ResultSet rs) throws SQLException {
2.         return rs.next() ? this.convert.toBean(rs, this.type) : null;
3.     }

```

最后到了populateBean () :

```

1.     private <T> T populateBean(ResultSet rs, T bean, PropertyDescriptor[] props,
2.         int[] columnToProperty) throws SQLException {
3.         for(int i = 1; i < columnToProperty.length; ++i) {
4.             if (columnToProperty[i] != -1) {
5.                 PropertyDescriptor prop = props[columnToProperty[i]];
6.                 Class<?> propType = prop.getPropertyType();
7.                 Object value = null;
8.                 if (propType != null) {
9.                     value = this.processColumn(rs, i, propType);
10.                    if (value == null && propType.isPrimitive()) {
11.                        value = primitiveDefaults.get(propType);
12.                    }
13.                }
14.                this.callSetter(bean, prop, value);
15.            }
16.        }
17.
18.        return bean;
19.    }

```

通过for循环，把rs的值填充到了指定的类型属性中。

输出的结果中，authorId为什么是空的？

这种自动映射，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。

1.1.3. Spring JDBC

工程：spring-mybatis

除了DbUtils之外，Spring也对原生的JDBC进行了封装。

前面说到的几个问题是怎么解决的呢？

1、代码重复——Spring 提供了一个模板方法JdbcTemplate，里面封装了各种各样的execute、query和update方法。

JdbcTemplate这个类（类的注释）：

它是JDCB的核心包的中心类。简化了JDBC的使用，可以避免常见的异常。它封装了JDBC的核心流程，应用只要提供SQL，提取结果集就可以了。它是线程安全的。

初始化的时候可以设置数据源，所以资源管理的问题也可以解决。

```
1.     public JdbcTemplate(DataSource dataSource) {
2.         this.setDataSource(dataSource);
3.         this.afterPropertiesSet();
4.     }
```

2、对于结果集的处理，Spring JDBC提供了一个RowMapper接口，可以把结果集转换成Java对象，它作为JdbcTemplate的参数使用。

比如要把查询tbl_emp表得到的结果集转换成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。在mapRow()方法里面完成对结果集的处理。

参见spring-mybatis-lesson工程cn.sitedev.crud.rowmapper.EmployeeRowMapper

```
1.  package cn.sitedev.crud.rowmapper;
2.
3.  import cn.sitedev.crud.bean.Employee;
4.  import org.springframework.jdbc.core.RowMapper;
5.
6.  import java.sql.ResultSet;
7.  import java.sql.SQLException;
8.
9.  /**
10.   * 实现RowMapper接口
11.   */
12. public class EmployeeRowMapper implements RowMapper {
13.     @Override
14.     public Object mapRow(ResultSet resultSet, int i) throws SQLException {
15.         Employee employee = new Employee();
16.         employee.setEmpId(resultSet.getInt("emp_id"));
17.         employee.setEmpName(resultSet.getString("emp_name"));
18.         employee.setGender(resultSet.getString("gender"));
19.         employee.setEmail(resultSet.getString("email"));
20.         return employee;
21.     }
22. }
```

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。

参见spring-mybatis-lesson工程cn.sitedev.EmployeeRowMapperTest

```
1.     @Test
2.     public void EmployeeTest() {
3.         list = jdbcTemplate.query("select * from tbl_emp", new
EmployeeRowMapper());
4.
5.         // list = jdbcTemplate.query(" select * from tbl_emp" ,new
BaseRowMapper(Employee.class));
6.         System.out.println(list);
7.     }
```

通过这种方式，无论在多少个地方查询tbl_emp表，都不需要重复处理 ResultSet，只要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。

当然，如果项目的表数量非常多的时候，每张表转换为POJO都要定义一个RowMapper，会导致类文件数量膨胀。

所以有没有办法让表里面一行数据的字段，跟POJO的属性自动对应起来，实现自动映射呢？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（例如char）和Java对象的类型（例如String）要匹配起来。

我们可以自己写一个支持泛型的 `BaseRowMapper<T>` 实现RowMapper接口，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。

参见spring-mybatis-lesson工程cn.sitedev.crud.rowmapper.BaseRowMapper

```

1. package cn.sitedev.crud.rowmapper;
2. import org.springframework.jdbc.core.RowMapper;
3.
4. import java.lang.reflect.Field;
5. import java.math.BigDecimal;
6. import java.sql.ResultSet;
7. import java.sql.ResultSetMetaData;
8. import java.sql.SQLException;
9. import java.sql.Timestamp;
10. import java.util.Date;
11. import java.util.HashMap;
12. import java.util.regex.Matcher;
13. import java.util.regex.Pattern;
14.
15. public class BaseRowMapper<T> implements RowMapper<T> {
16.
17.     private Class<?> targetClazz;
18.     private HashMap<String, Field> fieldMap;
19.
20.     public BaseRowMapper(Class<?> targetClazz) {
21.         this.targetClazz = targetClazz;
22.         fieldMap = new HashMap<>();
23.         Field[] fields = targetClazz.getDeclaredFields();
24.         for (Field field : fields) {
25.             fieldMap.put(field.getName(), field);
26.         }
27.     }
28.
29.     @Override
30.     public T mapRow(ResultSet rs, int arg1) throws SQLException {
31.         T obj = null;
32.
33.         try {
34.             obj = (T) targetClazz.newInstance();
35.
36.             final ResultSetMetaData metaData = rs.getMetaData();
37.             int columnLength = metaData.getColumnCount();
38.             String columnName = null;
39.
40.             for (int i = 1; i <= columnLength; i++) {
41.                 columnName = metaData.getColumnName(i);
42.                 Class fieldClazz = fieldMap.get(camel(columnName)).getType();
43.                 Field field = fieldMap.get(camel(columnName));
44.                 field.setAccessible(true);
45.
46.                 // fieldClazz == Character.class || fieldClazz == char.class
47.                 if (fieldClazz == int.class || fieldClazz == Integer.class) {
48.                     // int
49.                     field.set(obj, rs.getInt(columnName));

```

```

50.         } else if (fieldClazz == boolean.class || fieldClazz ==
Boolean.class) {
51.             // boolean
52.             field.set(obj, rs.getBoolean(columnName));
53.         } else if (fieldClazz == String.class) {
54.             // string
55.             field.set(obj, rs.getString(columnName));
56.         } else if (fieldClazz == float.class) {
57.             // float
58.             field.set(obj, rs.getFloat(columnName));
59.         } else if (fieldClazz == double.class || fieldClazz ==
Double.class) {
60.             // double
61.             field.set(obj, rs.getDouble(columnName));
62.         } else if (fieldClazz == BigDecimal.class) {
63.             // big decimal
64.             field.set(obj, rs.getBigDecimal(columnName));
65.         } else if (fieldClazz == short.class || fieldClazz == Short.class)
{
66.             // short
67.             field.set(obj, rs.getShort(columnName));
68.         } else if (fieldClazz == Date.class) {
69.             // date
70.             field.set(obj, rs.getDate(columnName));
71.         } else if (fieldClazz == Timestamp.class) {
72.             // timestamp
73.             field.set(obj, rs.getTimestamp(columnName));
74.         } else if (fieldClazz == Long.class || fieldClazz == long.class) {
75.             // long
76.             field.set(obj, rs.getLong(columnName));
77.         }
78.
79.         field.setAccessible(false);
80.     }
81.     } catch (Exception e) {
82.         e.printStackTrace();
83.     }
84.
85.     return obj;
86. }
87.
88. /**
89.  * 下划线转驼峰
90.  * @param str
91.  * @return
92.  */
93. public static String camel(String str) {
94.     Pattern pattern = Pattern.compile("_(\\w\\w)");
95.     Matcher matcher = pattern.matcher(str);

```

```

96.         StringBuffer sb = new StringBuffer(str);
97.         if(matcher.find()) {
98.             sb = new StringBuffer();
99.             matcher.appendReplacement(sb, matcher.group(1).toUpperCase());
100.            matcher.appendTail(sb);
101.        }else {
102.            return sb.toString();
103.        }
104.        return camel(sb.toString());
105.    }
106. }

```

上面的方法就可以改成：

```

1.     @Test
2.     public void EmployeeTest() {
3.         //         list = jdbcTemplate.query("select * from tbl_emp", new
EmployeeRowMapper());
4.
5.         list = jdbcTemplate.query(" select * from tbl_emp", new
BaseRowMapper(Employee.class));
6.         System.out.println(list);
7.     }

```

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的RowMapper。

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，帮助我们解决的问题：

- 1、对操作数据的增删改查的方法进行了封装；
- 2、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；
- 3、可以帮助我们映射结果集，无论是映射成List、Map还是POJO。

这两个工具已经可以帮助我们解决很大的问题了，但是还是存在一些不足：

- 1、SQL语句都是写在代码里面的，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传入对象和Map，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，性能还不够好。

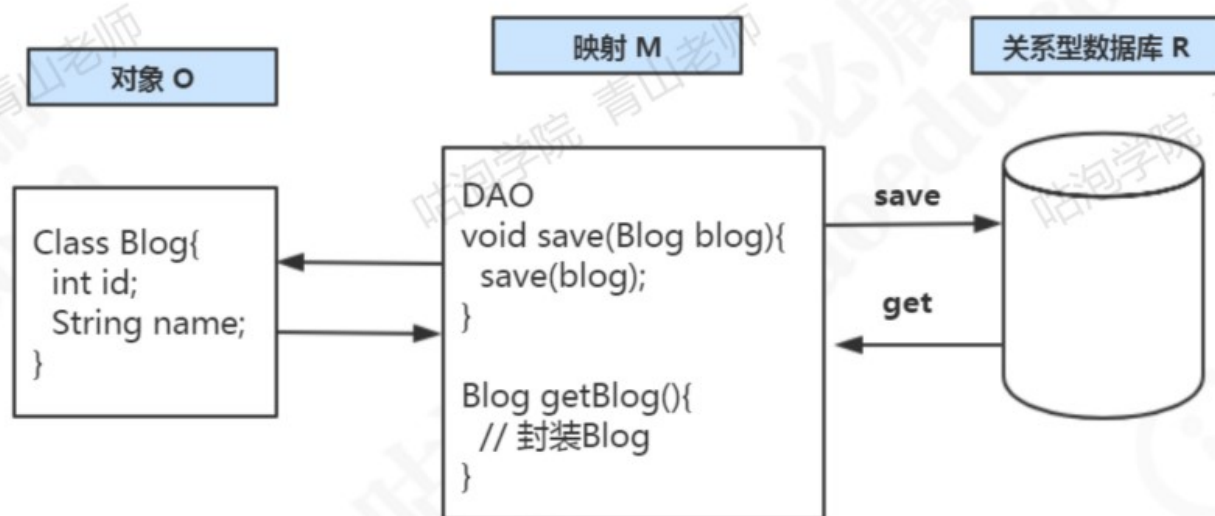
要解决这些问题，使用这些工具类还是不够的，这个时候用到ORM框架了。

1.1.4. Hibernate

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



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。

工程：ssh-demo

例如Blog.hbm.xml：

参见ssh-demo-lesson工程Blog.hbm.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3.     '-//Hibernate/Hibernate Mapping DTD 3.0//EN'
4.     'http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd'>
5. <hibernate-mapping>
6.     <class name="cn.sitedev.domain.Blog" table="blog">
7.         <id name="bid"/>
8.         <property name="name"></property>
9.         <property name="authorId" column="author_id"></property>
10.     </class>
11. </hibernate-mapping>
```

然后通过Hibernate提供（session）的增删改查的方法来操作对象。单元测试类：
cn.sitedev.Hibernate Test

参见ssh-demo-lesson工程的cn.sitedev.HibernateTest

```
1. package cn.sitedev;
2.
3. import cn.sitedev.domain.Blog;
4. import org.hibernate.Session;
5. import org.hibernate.SessionFactory;
6. import org.hibernate.Transaction;
7. import org.hibernate.cfg.Configuration;
8.
9. public class HibernateTest {
10.     public static void main(String[] args) {
11.         Configuration configuration = new Configuration();
12.         // 默认使用hibernate.cfg.xml
13.         configuration.configure();
14.         // 创建SessionFactory
15.         SessionFactory factory = configuration.buildSessionFactory();
16.         // 创建Session
17.         Session session = factory.openSession();
18.         // 获取事务对象
19.         Transaction transaction = session.getTransaction();
20.         // 开启事务
21.         transaction.begin();
22.         // 把对象添加到数据库中
23.         Blog blog = new Blog();
24.         blog.setBid(9000004);
25.         blog.setName("MySQL从入门到入狱");
26.         blog.setAuthorId(1001);
27.         session.save(blog);
28.         // 提交事务
29.         transaction.commit();
30.
31.         // 关闭Session
32.         session.close();
33.     }
34. }
```

hibernate.cfg.xml


```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE hibernate-configuration PUBLIC
3.      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4.      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5.  <hibernate-configuration>
6.      <session-factory>
7.          <property name="hibernate.connection.driver_class">
8.              com.mysql.cj.jdbc.Driver
9.          </property>
10.         <property name="hibernate.connection.url">
11.             jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf8&serverTimezone=UTC
12.         </property>
13.         <property name="hibernate.connection.username">root</property>
14.         <property name="hibernate.connection.password">root</property>
15.         <property name="hibernate.dialect">
16.             org.hibernate.dialect.MySQLDialect
17.         </property>
18.
19.         <property name="hibernate.show_sql">true</property>
20.         <property name="hibernate.format_sql">true</property>
21.         <property name="hibernate.hbm2ddl.auto">update</property>
22.
23.         <mapping resource="Blog.hbm.xml"/>
24.     </session-factory>
25. </hibernate-configuration>

```

Blog.hbm.xml

```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE hibernate-mapping PUBLIC
3.      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4.      'http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd'>
5.  <hibernate-mapping>
6.      <class name="cn.sitedev.domain.Blog" table="blog">
7.          <id name="bid"/>
8.          <property name="name"></property>
9.          <property name="authorId" column="author_id"></property>
10.     </class>
11. </hibernate-mapping>

```

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。

当然映射配置文件也可以使用注解代替。

参见ssh-demo-lesson工程cn.sitedev.domain.Blog

```

1. package cn.sitedev.domain;
2.
3. import lombok.Data;
4.
5. import javax.persistence.*;
6.
7. @Entity
8. @Table(name = "blog")
9. @Data
10. public class Blog {
11.     @Id
12.     @Column(name = "bid")
13.     private Integer bid;
14.
15.     @Column(name = "name")
16.     private String name;
17.
18.     @Column(name = "author_id")
19.     private Integer authorId;
20. }

```

Session 相关方法也可以通过继承JpaRepository获得，无需手动创建。

参见ssh-demo-lesson工程cn.sitedev.dao.BlogDao

```

1. package cn.sitedev.dao;
2.
3. import cn.sitedev.domain.Blog;
4. import org.springframework.data.jpa.repository.JpaRepository;
5. import org.springframework.stereotype.Repository;
6.
7. @Repository
8. public interface BlogDao extends JpaRepository<Blog, Integer> {
9.
10. }

```

然后注入到BlogServiceImpl，把IBlogService 注入到Controller。

参见ssh-demo-lesson工程

cn.sitedev.service.IBlogService,
cn.sitedev.service.BlogServiceImpl
cn.sitedev.controller.BlogController

```

1. package cn.sitedev.service;
2.
3. import cn.sitedev.domain.Blog;
4. import java.util.List;
5.
6. public interface IBlogService {
7.     public List<Blog> queryBlog();
8.
9.     public void addBlog(Blog blog);
10. }
11. //////////////////////////////////////
12. package cn.sitedev.service;
13.
14. import cn.sitedev.dao.BlogDao;
15. import cn.sitedev.domain.Blog;
16. import org.springframework.beans.factory.annotation.Autowired;
17. import org.springframework.stereotype.Service;
18. import java.util.List;
19.
20. @Service
21. public class BlogServiceImpl implements IBlogService {
22.     @Autowired
23.     private BlogDao blogDao;
24.
25.     @Override
26.     public List<Blog> queryBlog()
27.     {
28.         return blogDao.findAll();
29.     }
30.
31.     @Override
32.     public void addBlog(Blog blog)
33.     {
34.         blogDao.save(blog);
35.     }
36. }
37. //////////////////////////////////////
38. package cn.sitedev.controller;
39.
40. import cn.sitedev.domain.Blog;
41. import cn.sitedev.service.IBlogService;
42. import org.springframework.beans.factory.annotation.Autowired;
43. import org.springframework.web.bind.annotation.RequestMapping;
44. import org.springframework.web.bind.annotation.RestController;
45. import java.util.List;
46.
47. @RestController
48. @RequestMapping("/blog")
49. public class BlogController {

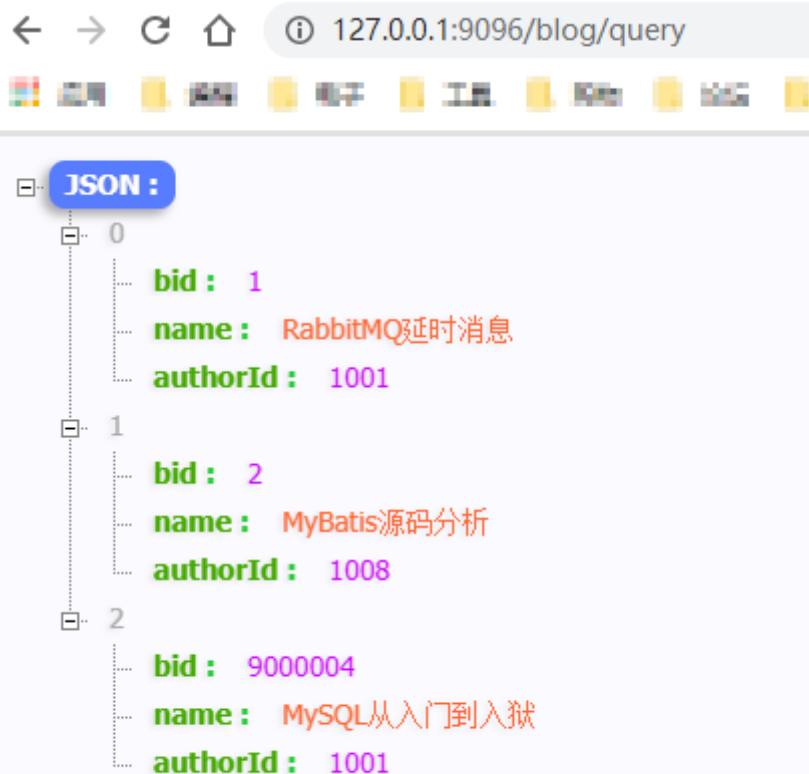
```

```

50.
51.     @Autowired
52.     private IBlogService blogService;
53.
54.     // http://127.0.0.1:9096/blog/query
55.     // 注意是mybatis数据库
56.     @RequestMapping("/query")
57.     public List<Blog> queryUser()
58.     {
59.         List<Blog> list = blogService.queryBlog();
60.         return list;
61.     }
62.
63.     // http://127.0.0.1:9096/blog/save
64.     // 注意是mybatis数据库
65.     @RequestMapping("/save")
66.     public void addBlog()
67.     {
68.         Blog blog = new Blog();
69.         blog.setBid(8000008);
70.         blog.setName("Redis从入门到改行");
71.         blog.setAuthorId(1001);
72.         blogService.addBlog(blog);
73.     }
74. }

```

启动App, 访问: <http://127.0.0.1:9096/blog/query>



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:9096/blog/query`. Below the address bar, there is a toolbar with various icons. The main content area shows a JSON response, which is expanded to reveal an array of three blog objects. Each object contains fields for `bid`, `name`, and `authorId`.

```

JSON :
[
  {
    "bid": 1,
    "name": "RabbitMQ延时消息",
    "authorId": 1001
  },
  {
    "bid": 2,
    "name": "MyBatis源码分析",
    "authorId": 1008
  },
  {
    "bid": 9000004,
    "name": "MySQL从入门到入狱",
    "authorId": 1001
  }
]

```

总结Hibernate的特性：

- 1、根据数据库方言自动生成SQL，移植性好；
- 2、自动管理连接资源（支持数据源）；
- 3、实现了对象和关系型数据库的完全映射，操作对象就像操作数据库记录一样；
- 4、提供了缓存功能机制。

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()、update () 、save()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、自动生成SQL的方式，如果要基于SQL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的问题。
- 3、不支持动态SQL，比如分表中的表名、条件、参数变化等，无法根据条件自动生成SQL。

我们需要一个更加灵活的框架。

1.1.5. 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。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。

MyBatis的前身是ibatis，2001年开始开发，是 “internet”和 “abatis['aebetis]（障碍物）”两个单词的组合。04年捐赠给Apache。2010年更名为MyBatis。

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。

1.2. MyBatis实际使用案例

1.2.1. 编程式使用

先看看JavaAPI编程的方式，MyBatis怎么使用。

先引入mybatis jar包。

```
1.      <dependency>
2.          <groupId>org.mybatis</groupId>
3.          <artifactId>mybatis</artifactId>
4.          <version>3.5.4</version>
5.      </dependency>
```

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。这里面只定义了数据源和Mapper 映射器路径。

参见mybatis-standalone-lesson工程mybatis-config.xml

```

1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-config.dtd">
3. <configuration>
4.
5.     <properties resource="db.properties"></properties>
6.     <environments default="development">
7.         <environment id="development">
8.             <transactionManager type="JDBC"/><!-- 单独使用时配置成MANAGED没有事务
-->
9.             <dataSource type="POOLED">
10.                 <property name="driver" value="${jdbc.driver}"/>
11.                 <property name="url" value="${jdbc.url}"/>
12.                 <property name="username" value="${jdbc.username}"/>
13.                 <property name="password" value="${jdbc.password}"/>
14.             </dataSource>
15.         </environment>
16.     </environments>
17.
18.     <mappers>
19.         <mapper resource="BlogMapper.xml"/>
20.     </mappers>
21.
22. </configuration>

```

参见mybatis-standalone-lesson工程db.properties

```

1. jdbc.driver=com.mysql.jdbc.Driver
2. jdbc.url=jdbc:mysql://localhost:3306/mybatis?
   useUnicode=true&characterEncoding=utf-8&rewriteBatchedStatements=true
3. jdbc.username=root
4. jdbc.password=root

```

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。

参见mybatis-standalone-lesson工程BlogMapper.xml

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3. <mapper namespace="cn.sitedev.mapper.BlogMapper">
4.
5.     <resultMap id="BaseResultMap" type="blog">
6.         <id column="bid" property="bid" jdbcType="INTEGER"/>
7.         <result column="name" property="name" jdbcType="VARCHAR"/>
8.         <result column="author_id" property="authorId" jdbcType="INTEGER"/>
9.     </resultMap>
10.
11.     <select id="selectBlogById" resultMap="BaseResultMap"
12. statementType="PREPARED">
13.         SELECT * FROM blog WHERE bid = #{bid}
14.     </select>
15. </mapper>
```

配置好了，怎么通过MyBatis 执行一个查询呢？

既然MyBatis的目的是简化JDBC的操作，那么它必须要提供一个可以执行增删改查的对象，这个对象就是SqlSession接口，我们把它理解为跟数据库的一个连接，或者一次会话。

SqlSession怎么创建呢？因为数据源、MyBatis 核心行为的控制（例如是否开启缓存）都在全局配置文件中，所以必须基于全局配置文件创建。这里它不是直接new 出来的，而是通过一个工厂类创建的。

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Mapper映射器中的SQL。

参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest

```
1. package cn.sitedev;
2.
3. import cn.sitedev.domain.Blog;
4. import cn.sitedev.domain.associate.AuthorAndBlog;
5. import cn.sitedev.domain.associate.BlogAndAuthor;
6. import cn.sitedev.domain.associate.BlogAndComment;
7. import cn.sitedev.mapper.BlogMapper;
8. import cn.sitedev.mapper.BlogMapperExt;
9. import org.apache.ibatis.io.Resources;
10. import org.apache.ibatis.session.RowBounds;
11. import org.apache.ibatis.session.SqlSession;
12. import org.apache.ibatis.session.SqlSessionFactory;
13. import org.apache.ibatis.session.SqlSessionFactoryBuilder;
14. import org.junit.Before;
15. import org.junit.Test;
16.
17. import java.io.IOException;
18. import java.io.InputStream;
19. import java.util.ArrayList;
20. import java.util.List;
21.
22. /**
23.  * MyBatis Maven演示工程
24.  */
25. public class MyBatisTest {
26.
27.     private SqlSessionFactory sqlSessionFactory;
28.
29.     @Before
30.     public void prepare() throws IOException {
31.         String resource = "mybatis-config.xml";
32.         InputStream inputStream = Resources.getResourceAsStream(resource);
33.         sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
34.     }
35.
36.     /**
37.      * 使用 MyBatis API方式
38.      *
39.      * @throws IOException
40.      */
41.     @Test
42.     public void testStatement() throws IOException {
43.         SqlSession session = sqlSessionFactory.openSession();
44.         try {
45.             Blog blog = (Blog)
session.selectOne("cn.sitedev.mapper.BlogMapper.selectBlogById", 1);
46.             System.out.println(blog);
47.         } finally {
48.             session.close();

```



```
49.         }  
50.     }
```

这样的调用方式，解决了重复代码、资源管理、SQL耦合、结果集映射这4大问题。

不过，这样的调用方式还是会存在一些问题：

- (1) Statement ID是硬编码，维护起来很不方便；
- (2) 不能在编译时进行类型检查，如果namespace 或者Statement ID输错了，只能在运行的时候报错。

所以我们通常会使用第二种方式，也是新版的MyBatis 里面推荐的方式：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。

参见mybatis-standalone-lesson工程[cn.sitedev.MyBatisTest](http://cn.sitedev.com/MyBatisTest)

```
1. package cn.sitedev;
2.
3. import cn.sitedev.domain.Blog;
4. import cn.sitedev.domain.associate.AuthorAndBlog;
5. import cn.sitedev.domain.associate.BlogAndAuthor;
6. import cn.sitedev.domain.associate.BlogAndComment;
7. import cn.sitedev.mapper.BlogMapper;
8. import cn.sitedev.mapper.BlogMapperExt;
9. import org.apache.ibatis.io.Resources;
10. import org.apache.ibatis.session.RowBounds;
11. import org.apache.ibatis.session.SqlSession;
12. import org.apache.ibatis.session.SqlSessionFactory;
13. import org.apache.ibatis.session.SqlSessionFactoryBuilder;
14. import org.junit.Before;
15. import org.junit.Test;
16.
17. import java.io.IOException;
18. import java.io.InputStream;
19. import java.util.ArrayList;
20. import java.util.List;
21.
22. /**
23.  * MyBatis Maven演示工程
24.  */
25. public class MyBatisTest {
26.
27.     private SqlSessionFactory sqlSessionFactory;
28.
29.     @Before
30.     public void prepare() throws IOException {
31.         String resource = "mybatis-config.xml";
32.         InputStream inputStream = Resources.getResourceAsStream(resource);
33.         sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
34.     }
35.
36.     /**
37.      * 通过 SqlSession.getMapper(XXXMapper.class) 接口方式
38.      *
39.      * @throws IOException
40.      */
41.     @Test
42.     public void testSelect() throws IOException {
43.         SqlSession session = sqlSessionFactory.openSession(); //
44.         ExecutorType.BATCH
45.         try {
46.             BlogMapper mapper = session.getMapper(BlogMapper.class);
47.             Blog blog = mapper.selectBlogById(1);
48.             System.out.println(blog);
49.         } finally {
```

```
49.         session.close();
50.     }
51. }
```

其对应Mapper接口为

```
1. package cn.sitedev.mapper;
2.
3. import cn.sitedev.domain.Blog;
4. import cn.sitedev.domain.BlogExample;
5. import cn.sitedev.domain.associate.AuthorAndBlog;
6. import cn.sitedev.domain.associate.BlogAndAuthor;
7. import cn.sitedev.domain.associate.BlogAndComment;
8. import org.apache.ibatis.session.RowBounds;
9.
10. import java.util.List;
11.
12. public interface BlogMapper {
13.     /**
14.      * 根据主键查询文章
15.      *
16.      * @param bid
17.      * @return
18.      */
19.     public Blog selectBlogById(Integer bid);
20. }
```

通过执行接口方法，来执行映射器中的SQL语句。

总结一下，MyBatis的核心特性，或者说它可以解决哪些主要问题：

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

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Connection、Statement和ResultSet这些对象。

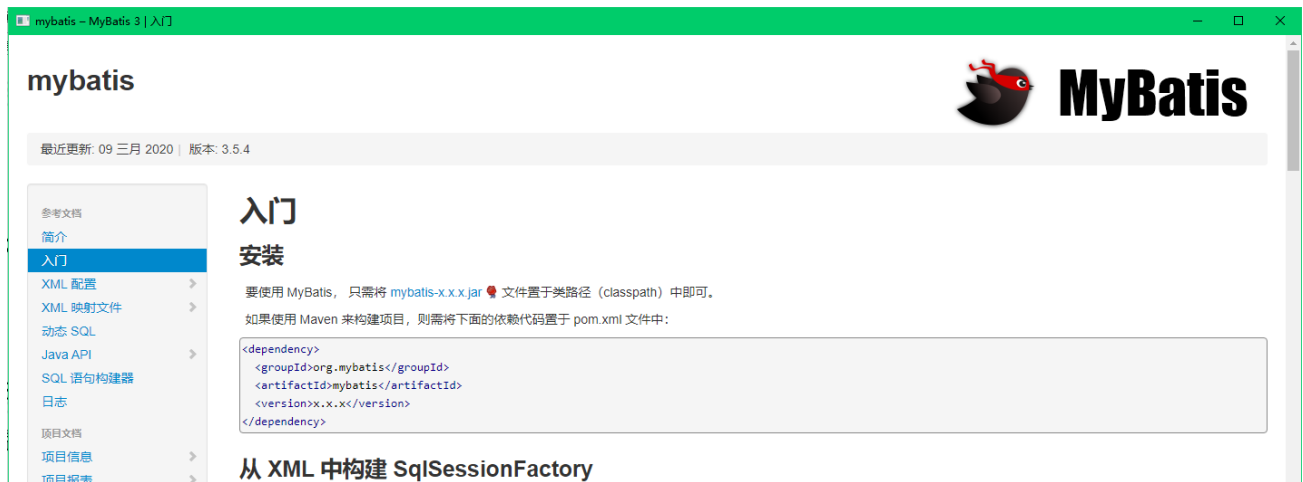
有这么多的工具和不同的框架，在实际的项目里面应该怎么选择？在一些业务比较简单的项目中，我们可以使用Hibernate；如果需要更加灵活的SQL，可以使用MyBatis，对于底层的编码，或者性能要求非常高的场合，可以用JDBC；实际上在我们的项目中，MyBatis 和SpringJDBC是可以混合使用的；当然，我们也根据项目的需求自己写ORM框架。

1.2.2. 核心对象的生命周期

在编程式使用的这个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。这四个对象的生命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。

1.2.2.1. 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。所以它生命周期只存在于方法的局部。

1.2.2.2. SqlSessionFactory (单例)

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。

1.2.2.3. 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）。

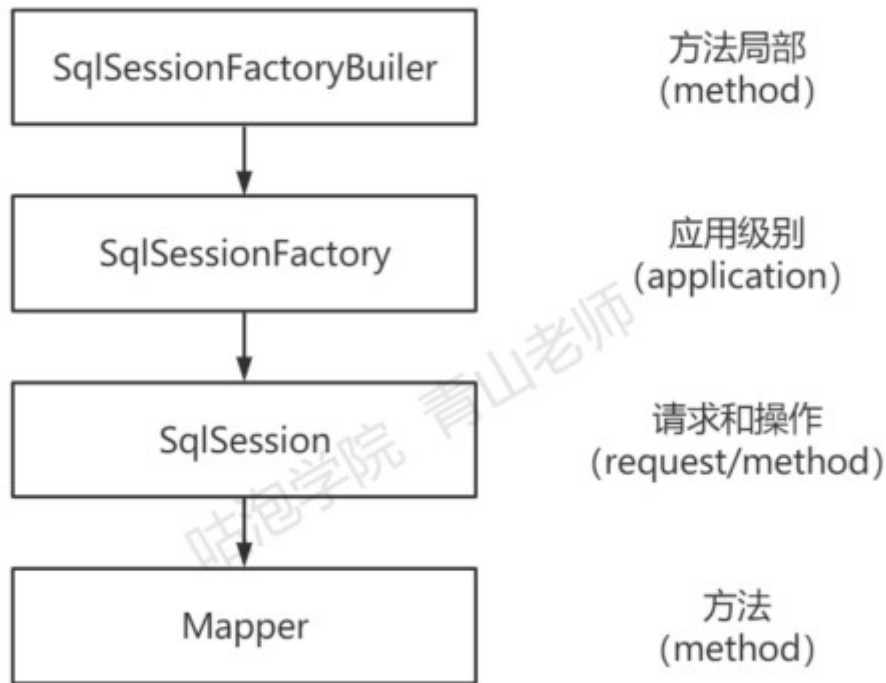
1.2.2.4. Mapper

Mapper（实际上是一个代理对象）是从SqlSession中获取的。

```
1. 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：



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。

1.2.3. 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（例如之前的demo中，只配置了数据源）。MyBatis里面提供的配置项非常多，我们没有显式地配置的时候，这些配置项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3.5.4（2020年3月9日发布）。

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

1.2.3.1. 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。

1.2.3.1.1. properties

第一个一级标签是properties，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。

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

1.2.3.1.2. settings

settings 里面是MyBatis的一些核心配置。

```
1.      <settings>
2.          <!-- 打印查询语句 -->
3.          <setting name="logImpl" value="STDOUT_LOGGING"/>
4.
5.          <!-- 控制全局缓存（二级缓存），默认 true-->
6.          <setting name="cacheEnabled" value="true"/>
7.
8.          <!-- 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 -->
9.          <setting name="lazyLoadingEnabled" value="true"/>
10.         <!-- 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
11.         select标签的 fetchType来覆盖-->
12.         <setting name="aggressiveLazyLoading" value="true"/>
13.         <!-- 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 -->
14.         <!--<setting name="proxyFactory" value="CGLIB" />-->
15.         <!-- STATEMENT级别的缓存，使一级缓存，只针对当前执行的这一 statement有效 -->
16.         <setting name="localCacheScope" value="STATEMENT"/>
17.         <!--
18.         <setting name="localCacheScope" value="SESSION"/>
19.     </settings>
```

1.2.3.1.3. typeAliases

TypeAlias是类型的别名，跟Linux系统里面的 alias一样，主要用来简化类名全路径的拼写。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。

```
1.      <typeAliases>
2.          <typeAlias alias="blog" type="cn.sitedev.domain.Blog"/>
3.      </typeAliases>
```

配置了别名以后，在配置文件中只需要写别名就可以了，比如 cn.sitedev.domain.Blog 可以简化成 blog。

```
1. <select id="selectBlogByBean" parameterType="blog" resultType="blog">
2.     SELECT bid, name, author_id authorId FROM blog WHERE name = '${name}'
3. </select>
```

MyBatis 里面有很多系统预先定义好的类型别名，在TypeAliasRegistry中。所以可以用 string 代替 java.lang.String。

```
1. public class TypeAliasRegistry {
2.     private final Map<String, Class<?>> typeAliases = new HashMap();
3.
4.     public TypeAliasRegistry() {
5.         this.registerAlias("string", String.class);
6.         this.registerAlias("byte", Byte.class);
7.         this.registerAlias("long", Long.class);
8.         this.registerAlias("short", Short.class);
9.         this.registerAlias("int", Integer.class);
10.        this.registerAlias("integer", Integer.class);
11.        this.registerAlias("double", Double.class);
12.        this.registerAlias("float", Float.class);
13.        this.registerAlias("boolean", Boolean.class);
14.        this.registerAlias("byte[]", Byte[].class);
15.        this.registerAlias("long[]", Long[].class);
16.        this.registerAlias("short[]", Short[].class);
17.        this.registerAlias("int[]", Integer[].class);
18.        this.registerAlias("integer[]", Integer[].class);
19.        this.registerAlias("double[]", Double[].class);
20.        this.registerAlias("float[]", Float[].class);
21.        this.registerAlias("boolean[]", Boolean[].class);
22.        this.registerAlias("_byte", Byte.TYPE);
23.        this.registerAlias("_long", Long.TYPE);
24.        this.registerAlias("_short", Short.TYPE);
25.        this.registerAlias("_int", Integer.TYPE);
26.        this.registerAlias("_integer", Integer.TYPE);
27.        this.registerAlias("_double", Double.TYPE);
28.        this.registerAlias("_float", Float.TYPE);
29.        this.registerAlias("_boolean", Boolean.TYPE);
30.        this.registerAlias("_byte[]", byte[].class);
31.        this.registerAlias("_long[]", long[].class);
32.        this.registerAlias("_short[]", short[].class);
33.        this.registerAlias("_int[]", int[].class);
34.        this.registerAlias("_integer[]", int[].class);
35.        this.registerAlias("_double[]", double[].class);
36.        this.registerAlias("_float[]", float[].class);
37.        this.registerAlias("_boolean[]", boolean[].class);
38.        this.registerAlias("date", Date.class);
39.        this.registerAlias("decimal", BigDecimal.class);
40.        this.registerAlias("bigdecimal", BigDecimal.class);
41.        this.registerAlias("biginteger", BigInteger.class);
42.        this.registerAlias("object", Object.class);
43.        this.registerAlias("date[]", Date[].class);
44.        this.registerAlias("decimal[]", BigDecimal[].class);
45.        this.registerAlias("bigdecimal[]", BigDecimal[].class);
46.        this.registerAlias("biginteger[]", BigInteger[].class);
47.        this.registerAlias("object[]", Object[].class);
48.        this.registerAlias("map", Map.class);
49.        this.registerAlias("hashmap", HashMap.class);
```



```
50.         this.registerAlias("list", List.class);
51.         this.registerAlias("arraylist", ArrayList.class);
52.         this.registerAlias("collection", Collection.class);
53.         this.registerAlias("iterator", Iterator.class);
54.         this.registerAlias("ResultSet", ResultSet.class);
55.     }
56.     ...
```

1.2.3.1.4. typeHandlers

由于Java类型和数据库的JDBC类型不是一一对应的（比如String与varchar、char、text），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。

```
1. <typeHandlers>
2.     <typeHandler handler="cn.sitedev.type.MyTypeHandler"></typeHandler>
3. </typeHandlers>
```

当参数类型和返回值是一个对象的时候，我没有做任何的配置，为什么对象里面的一个String属性，可以转换成数据库里面的 varchar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，它们都继承了抽象类BaseTypeHandler，泛型就是要处理的Java数据类型。

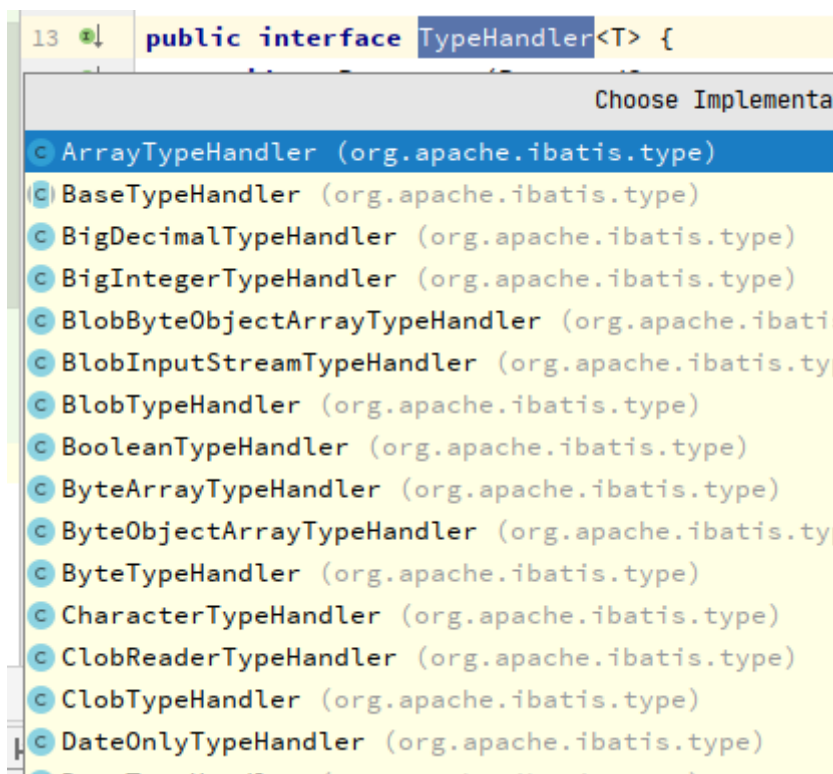
```
1. public final class TypeHandlerRegistry {
2.
3.     public TypeHandlerRegistry(Configuration configuration) {
4.         this.jdbcTypeHandlerMap = new EnumMap(JdbcType.class);
5.         this.typeHandlerMap = new ConcurrentHashMap();
6.         this.allTypeHandlersMap = new HashMap();
7.         this.defaultEnumTypeHandler = EnumTypeHandler.class;
8.         this.unknownTypeHandler = new UnknownTypeHandler(configuration);
9.         this.register((Class)Boolean.class, (TypeHandler)(new
BooleanTypeHandler()));
10.        this.register((Class)Boolean.TYPE, (TypeHandler)(new
BooleanTypeHandler()));
11.        this.register((JdbcType)JdbcType.BOOLEAN, (TypeHandler)(new
BooleanTypeHandler()));
12.        this.register((JdbcType)JdbcType.BIT, (TypeHandler)(new
BooleanTypeHandler()));
13.        this.register((Class)Byte.class, (TypeHandler)(new ByteTypeHandler()));
14.        this.register((Class)Byte.TYPE, (TypeHandler)(new ByteTypeHandler()));
15.        this.register((JdbcType)JdbcType.TINYINT, (TypeHandler)(new
ByteTypeHandler()));
16.        this.register((Class)Short.class, (TypeHandler)(new ShortTypeHandler()));
17.        this.register((Class)Short.TYPE, (TypeHandler)(new ShortTypeHandler()));
18.        this.register((JdbcType)JdbcType.SMALLINT, (TypeHandler)(new
ShortTypeHandler()));
19.        this.register((Class)Integer.class, (TypeHandler)(new
IntegerTypeHandler()));
20.        this.register((Class)Integer.TYPE, (TypeHandler)(new
IntegerTypeHandler()));
21.        this.register((JdbcType)JdbcType.INTEGER, (TypeHandler)(new
IntegerTypeHandler()));
22.        this.register((Class)Long.class, (TypeHandler)(new LongTypeHandler()));
23.        this.register((Class)Long.TYPE, (TypeHandler)(new LongTypeHandler()));
24.        this.register((Class)Float.class, (TypeHandler)(new FloatTypeHandler()));
25.        this.register((Class)Float.TYPE, (TypeHandler)(new FloatTypeHandler()));
26.        this.register((JdbcType)JdbcType.FLOAT, (TypeHandler)(new
FloatTypeHandler()));
27.        this.register((Class)Double.class, (TypeHandler)(new
DoubleTypeHandler()));
28.        this.register((Class)Double.TYPE, (TypeHandler)(new DoubleTypeHandler()));
29.        this.register((JdbcType)JdbcType.DOUBLE, (TypeHandler)(new
DoubleTypeHandler()));
30.        this.register((Class)Reader.class, (TypeHandler)(new
ClobReaderTypeHandler()));
31.        this.register((Class)String.class, (TypeHandler)(new
StringTypeHandler()));
32.        this.register((Class)String.class, JdbcType.CHAR, (TypeHandler)(new
StringTypeHandler()));
33.        this.register((Class)String.class, JdbcType.CLOB, (TypeHandler)(new
ClobTypeHandler()));
```

```

34.         this.register((Class)String.class, JdbcType.VARCHAR, (TypeHandler)(new
StringTypeHandler()));
35.         this.register((Class)String.class, JdbcType.LONGVARCHAR, (TypeHandler)(new
StringTypeHandler()));
36.         this.register((Class)String.class, JdbcType.NVARCHAR, (TypeHandler)(new
NStringTypeHandler()));
37.         this.register((Class)String.class, JdbcType.NCHAR, (TypeHandler)(new
NStringTypeHandler()));
38.         this.register((Class)String.class, JdbcType.NCLOB, (TypeHandler)(new
NClobTypeHandler()));
39.         this.register((JdbcType)JdbcType.CHAR, (TypeHandler)(new
StringTypeHandler()));
40.         this.register((JdbcType)JdbcType.VARCHAR, (TypeHandler)(new
StringTypeHandler()));
41.         this.register((JdbcType)JdbcType.CLOB, (TypeHandler)(new
ClobTypeHandler()));
42.         this.register((JdbcType)JdbcType.LONGVARCHAR, (TypeHandler)(new
StringTypeHandler()));
43.         this.register((JdbcType)JdbcType.NVARCHAR, (TypeHandler)(new
NStringTypeHandler()));
44.         this.register((JdbcType)JdbcType.NCHAR, (TypeHandler)(new
NStringTypeHandler()));
45.         this.register((JdbcType)JdbcType.NCLOB, (TypeHandler)(new
NClobTypeHandler()));
46.         this.register((Class)Object.class, JdbcType.ARRAY, (TypeHandler)(new
ArrayTypeHandler()));
47.         this.register((JdbcType)JdbcType.ARRAY, (TypeHandler)(new
ArrayTypeHandler()));
48.         this.register((Class)BigInteger.class, (TypeHandler)(new
BigIntegerTypeHandler()));
49.         this.register((JdbcType)JdbcType.BIGINT, (TypeHandler)(new
LongTypeHandler()));
50.         this.register((Class)BigDecimal.class, (TypeHandler)(new
BigDecimalTypeHandler()));
51.         this.register((JdbcType)JdbcType.REAL, (TypeHandler)(new
BigDecimalTypeHandler()));
52.         this.register((JdbcType)JdbcType.DECIMAL, (TypeHandler)(new
BigDecimalTypeHandler()));
53.         this.register((JdbcType)JdbcType.NUMERIC, (TypeHandler)(new
BigDecimalTypeHandler()));
54.         ...
55.     }
56.     ...

```

这个也是为什么大部分类型都不需要处理。当我们查询数据和登记数据，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

举个例子：

一个商户，在登记的时候需要注册它的经营范围。比如1手机，2电脑，3相机，4平板，在界面上是一个复选框（checkbox）。

在数据库保存的是用逗号分隔的字符串，例如“1, 3, 4”，而返回给程序的时候是整形数组 {1, 3, 4}

在每次获取数据的时候转换？还是在bean的get 方法里面转换？似乎都不太合适。

这时候我们可以写一个Integer[]类型的TypeHandler。

参见mybatis-standalone-lesson工程cn.sitedev.type.MyTypeHandler

```
1. package cn.sitedev.type;
2.
3. import org.apache.ibatis.type.BaseTypeHandler;
4. import org.apache.ibatis.type.JdbcType;
5.
6. import java.sql.CallableStatement;
7. import java.sql.PreparedStatement;
8. import java.sql.ResultSet;
9. import java.sql.SQLException;
10.
11. public class MyTypeHandler extends BaseTypeHandler<String> {
12.     @Override
13.     public void setNonNullParameter(PreparedStatement preparedStatement, int i,
14. String parameter, JdbcType jdbcType) throws SQLException {
15.         // 设置String类型的参数的时候调用, java类型 到 JDBC类型
16.         // 注意只有在字段上添加typeHandler属性才会生效
17.         // insertBlog name字段
18.         System.out.println("-----setNonNullParameter1: " +
19. parameter);
20.         preparedStatement.setString(i, parameter);
21.     }
22.
23.     @Override
24.     public String getNullableResult(ResultSet resultSet, String columnName) throws
25. SQLException {
26.         // 根据列名获取 String 类型的参数的时候调用, JDBC 类型到java类型
27.         // 注意只有在字段上添加typeHandler 属性才会生效
28.         System.out.println("-----getNullableResult1: " +
29. columnName);
30.         return resultSet.getString(columnName);
31.     }
32.
33.     @Override
34.     public String getNullableResult(ResultSet resultSet, int columnIndex) throws
35. SQLException {
36.         // 根据下标获取String类型参数的时候调用
37.         System.out.println("-----getNullableResult2: " + columnIndex);
38.         return resultSet.getString(columnIndex);
39.     }
40.
41.     @Override
42.     public String getNullableResult(CallableStatement callableStatement, int
43. columnIndex) throws SQLException {
44.         System.out.println("-----getNullableResult3: " + columnIndex);
45.         return callableStatement.getString(columnIndex);
46.     }
47. }
```

第二步，在mybatis-config.xml文件中注册：

```
1. <typeHandlers>
2.   <typeHandler handler="cn.sitedev.type.MyTypeHandler"></typeHandler>
3. </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：

```
1. <!-- 动态SQL trim -->
2. <insert id="insertBlog" parameterType="blog">
3.   insert into blog (bid, name, author_id)
4.   values (
5.     #{bid,jdbcType=INTEGER},
6.     #
7.     {name,jdbcType=VARCHAR,typeHandler=cn.sitedev.type.MyTypeHandler},
8.     #{authorId,jdbcType=INTEGER}
9.   )
10. </insert>
```

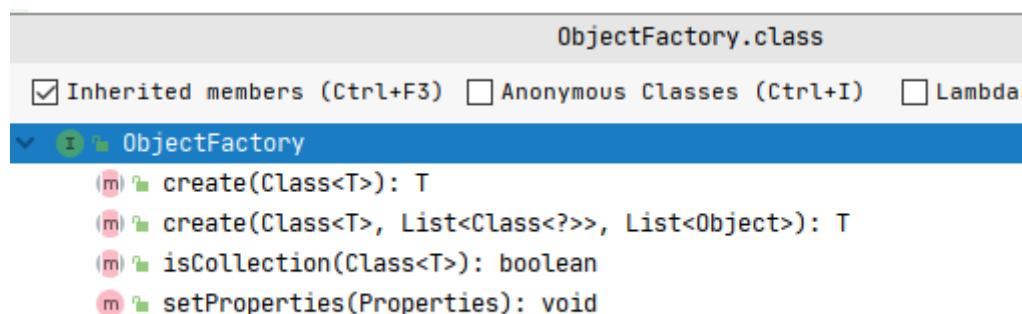
返回值的时候，从JDBC类型到Java 类型，在resultMap的列上指定typehandler：

```
1. <result column="name" property="name" jdbcType="VARCHAR"
   typeHandler="cn.sitedev.type.MyTypeHandler"/>
```

1.2.3.1.5. objectFactory

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。只能通过反射来创建。

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（MyBatis封装之后，简化了对象的创建），里面定义了4个方法。



方法	作用
<code>void setProperties(Properties properties);</code>	设置参数时调用
<code><T> T create(Class<T> type);</code>	创建对象（调用无参构造函数）
<code><T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object> constructorArgs);</code>	创建对象（调用带参数构造函数）
<code><T> boolean isCollection(Class<T> 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 ()，这里面能看到反射的代码。

默认情况下，所有的对象都是由 DefaultObjectFactory创建。

我们可以直接用自定义的工厂类来创建对象：

参见mybatis-standalone-lesson工程cn.sitedev.objectfactory.ObjectFactoryTest

```

1. package cn.sitedev.objectfactory;
2.
3. import cn.sitedev.domain.Blog;
4.
5. public class ObjectFactoryTest {
6.     public static void main(String[] args) {
7.         MyObjectFactory myObjectFactory = new MyObjectFactory();
8.         Blog blog = (Blog) myObjectFactory.create(Blog.class);
9.         System.out.println(blog);
10.    }
11. }
```

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（不再需要实现ObjectFactory接口）。

例如：

参见mybatis-standalone-lesson工程cn.sitedev.objectfactory.MyObjectFactory

```

1. package cn.sitedev.objectfactory;
2.
3. import cn.sitedev.domain.Blog;
4. import org.apache.ibatis.reflection.factory.DefaultObjectFactory;
5.
6. /*
7.  * <p>
8.  * 自定义ObjectFactory，通过反射的方式实例化对象
9.  * 一种是无参构造函数，一种是有参构造函数—第一个方法调用了第二个方法
10.  */
11. public class MyObjectFactory extends DefaultObjectFactory {
12.     @Override
13.     public Object create(Class type) {
14.         System.out.println("创建对象方法：" + type);
15.
16.         if (Blog.class.equals(type)) {
17.             Blog blog = (Blog) super.create(type);
18.             blog.setName("object factory");
19.             blog.setBid(1111);
20.             blog.setAuthorId(2222);
21.             return blog;
22.         }
23.         Object result = super.create(type);
24.         return result;
25.     }
26. }

```

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：

```

1. <!-- 对象工厂 -->
2. <objectFactory type="cn.sitedev.objectfactory.MyObjectFactory">
3.     <!--对象工厂注入的参数-->
4.     <property name="sitedev" value="666"/>
5. </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。

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()。


```

1.     private Object getRowValue(ResultSetWrapper rsw, ResultMap resultMap, String
      columnPrefix) throws SQLException {
2.         final ResultLoaderMap lazyLoader = new ResultLoaderMap();
3.         Object rowValue = createResultObject(rsw, resultMap, lazyLoader,
      columnPrefix);
4.         if (rowValue != null && !hasTypeHandlerForResultObject(rsw,
      resultMap.getType())) {
5.             final MetaObject metaObject = configuration.newMetaObject(rowValue);
6.             boolean foundValues = this.useConstructorMappings;
7.             if (shouldApplyAutomaticMappings(resultMap, false)) {
8.                 foundValues = applyAutomaticMappings(rsw, resultMap, metaObject,
      columnPrefix) || foundValues;
9.             }
10.            foundValues = applyPropertyMappings(rsw, resultMap, metaObject, lazyLoader,
      columnPrefix) || foundValues;
11.            foundValues = lazyLoader.size() > 0 || foundValues;
12.            rowValue = foundValues || configuration.isReturnInstanceForEmptyRow() ?
      rowValue : null;
13.        }
14.        return rowValue;
15.    }

```

3、返回结果的时候，ObjectFactory和TypeHandler哪个先工作？
肯定是先创建对象，所以先是ObjectFactory，再是TypeHandler。

1.2.3.1.6. 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。

<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

SqlSession是对外提供的接口。而SqlSession 增删改查的方法都是由Executor完成的（点开DefaultSqlSession源码的相关方法）。

```

1.     @Override
2.     public <E> List<E> selectList(String statement, Object parameter, RowBounds
rowBounds) {
3.         try {
4.             MappedStatement ms = configuration.getMappedStatement(statement);
5.             return executor.query(ms, wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
6.         } catch (Exception e) {
7.             throw ExceptionFactory.wrapException("Error querying database. Cause: " +
e, e);
8.         } finally {
9.             ErrorContext.instance().reset();
10.        }
11.    }
12.
13.    @Override
14.    public int update(String statement, Object parameter) {
15.        try {
16.            dirty = true;
17.            MappedStatement ms = configuration.getMappedStatement(statement);
18.            return executor.update(ms, wrapCollection(parameter));
19.        } catch (Exception e) {
20.            throw ExceptionFactory.wrapException("Error updating database. Cause: " +
e, e);
21.        } finally {
22.            ErrorContext.instance().reset();
23.        }
24.    }

```

Executor是真正的执行器的角色，也是实际的SQL逻辑执行的开始。

而MyBatis中又把SQL的执行，按照过程，细分成了三个对象：ParameterHandler处理参数，StatementHandler执行SQL，StatementHandler处理结果集。

1.2.3.1.7. 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。

```
1.     <environments default="development">
2.         <environment id="development">
3.             <transactionManager type="JDBC"/><!-- 单独使用时配置成MANAGED没有事务
-->
4.             <dataSource type="POOLED">
5.                 <property name="driver" value="${jdbc.driver}"/>
6.                 <property name="url" value="${jdbc.url}"/>
7.                 <property name="username" value="${jdbc.username}"/>
8.                 <property name="password" value="${jdbc.password}"/>
9.             </dataSource>
10.        </environment>
11.    </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 里面配置数据源和事务，覆盖MyBatis的配置。

dataSource

数据源，顾名思义，就是数据的来源，一个数据源就对应一个数据库。在Java里面，它是对数据库连接的一个抽象。

一般的数据源都会包括连接池管理的功能，所以很多时候也把DataSource直接称为连接池，准确的说法应该是：带连接池功能的数据源。

为什么要用连接池？

除了连接池之外，大家应该也听过很多其他的池，比如线程池，内存池，对象池，这种池化技术达到的目的基本上一样的。

如果没有连接池，那么每一个用户、每一次会话连接数据库都需要直接创建和释放连接，这个过程是会消耗的一定的时间的，并且会消耗应用和服务器的性能。当我们没有使用连接池的时候，客户程序得到的连接是一个物理连接，我们调用close () 方法会把这个连接真正地关闭。

如果采用连接池技术，在应用程序里面关闭连接的时候，物理连接没有被真正关闭掉，只是回到了连接池里面。

从这个角度来考虑，一般的连接池都会有初始连接数、最大连接数、回收时间等等这些参数，提供提前创建/资源重用/数量控制/超时管理等等这些功能。

MyBatis 自带了两种数据源，UNPOOLED和POOLED。也可以配置成其他数据库，比如C3PO、Hikari等等。市面上流行的数据源，一般都有连接池功能。

在跟Spring 集成的时候，事务和数据源都会交给Spring来管理，不再使用MyBatis配置的数据源。

1.2.3.1.8. mappers

`<mappers>` 标签配置的是映射器，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）

```
1. <mappers>
2.     <mapper resource="BlogMapper.xml">
3. </mappers>
```

2、使用完全限定资源定位符（绝对路径）（URL）

```
1. <mappers>
2.     <mapper resource="file:///app/sale/mappers/BlogMapper.xml">
3. </mappers>
```

3、使用映射器接口实现类的完全限定类名

```
1. <mappers>
2.     <mapper class="com.gupaoedu.mapper.BlogMapper"/>
3. </mappers>
```

4、将包内的映射器接口实现全部注册为映射器（最常用）

```
1. <mappers>
2.     <mapper class="com.gupaoedu.mapper"/>
3. </mappers>
```

1.2.3.1.9. settings

属性名	含义	简介	有效值	默认值
cacheEnabled	是否使用二级缓存	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
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,

	载			hashCode, toString
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
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（相当于关闭一级缓存）	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 自动映射字段或属性的方式	有三种方式，NONE 时将取消自动映射；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	NONE/PARTIAL/FULL	PARTIAL
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，等待数据库响应的最大秒数	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	如果允许在 SQL 的结果集使用分页，就设置该值为 false	true/false	true

	(ResultHandler, 即结果集处理)			
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

```

1.      <settings>
2.          <!-- 打印查询语句 -->
3.          <setting name="logImpl" value="STDOUT_LOGGING"/>
4.
5.          <!-- 控制全局缓存（二级缓存），默认 true-->
6.          <setting name="cacheEnabled" value="true"/>
7.
8.          <!-- 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 -->
9.          <setting name="lazyLoadingEnabled" value="true"/>
10.         <!-- 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
11.         select标签的 fetchType来覆盖-->
12.         <setting name="aggressiveLazyLoading" value="true"/>
13.         <!-- 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 -->
14.         <!--<setting name="proxyFactory" value="CGLIB" />-->
15.         <!-- STATEMENT级别的缓存，使一级缓存，只针对当前执行的这一 statement有效 -->
16.         <setting name="localCacheScope" value="STATEMENT"/>
17.         <!--
18.         <setting name="localCacheScope" value="SESSION"/>
19.     </settings>

```

1.2.3.2. Mapper.xml映射配置文件

<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：

1.2.3.2.1. <cache>

`<cache>` -给定命名空间的缓存配置（是否开启二级缓存）。

1.2.3.2.2. `<cache-ref>`

`<cache-ref>` -其他命名空间缓存配置的引用。缓存相关两个标签我们在讲解缓存的时候会详细讲到。

1.2.3.2.3. `<resultMap>`

`<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。

```
1.      <resultMap id="BaseResultMap" type="blog">
2.          <id column="bid" property="bid" jdbcType="INTEGER"/>
3.          <result column="name" property="name" jdbcType="VARCHAR" />
4.          <result column="author_id" property="authorId" jdbcType="INTEGER"/>
5.      </resultMap>
```

1.2.3.2.4. `<sql>`

`<sql>` -可被其他语句引用的可重用语句块。

```
1.      <sql id="Base_Column_List">
2.          bid, name, author_id
3.      </sql>
```

1.2.3.2.5. 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

```
1.      <select id="selectBlogById" resultMap="BaseResultMap"
2.          statementType="PREPARED">
3.          SELECT * FROM blog WHERE bid = #{bid}
4.      </select>
```

1.2.3.3. 总结

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

1.3. MyBatis最佳实践

1.3.1. 动态SQL

1.3.1.1. 为什么需要动态SQL?

看一段Oracle存储过程代码：

```

if l_qrypkg = 'Y' then
    l_whersql := l_whersql || ' and a.msetdt between ' || l_begndt || ' and ' ||
                l_termdt || ''';
else
    if l_dtfbsq is not null then
        l_whersql := l_whersql || ' and a.fcbpdt between ' || l_begndt || ' and ' ||
                l_termdt || ''';
    else
        l_whersql := l_whersql || ' and a.detldt between ' || l_begndt || ' and ' ||
                l_termdt || ''';
    end if;
end if;

```

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。

MyBaits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。

1.3.1.2. 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。

1.3.1.2.1. if——需要判断的时候，条件写在test中

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testSelectList

```
1.     @Test
2.     public void testSelectList() throws IOException {
3.         SqlSession session = sqlSessionFactory.openSession(); //
        ExecutorType.BATCH
4.         try {
5.             BlogMapper mapper = session.getMapper(BlogMapper.class);
6.             Blog blog = new Blog();
7.             blog.setName("改行");
8.             List<Blog> list1 = new ArrayList<Blog>();
9.             List<Blog> list2 = new ArrayList<Blog>();
10.            list1 = mapper.selectBlogListIf(blog);
11.            list2 = mapper.selectBlogListChoose(blog);
12.        } finally {
13.            session.close();
14.        }
15.    }
```

```
1.     <!--动态SQL where 和 if-->
2.     <select id="selectBlogListIf" parameterType="blog" resultMap="BaseResultMap">
3.         SELECT bid, name, author_id authorId FROM blog
4.         <where>
5.             <if test="bid != null">
6.                 AND bid = #{bid}
7.             </if>
8.             <if test="name != null and name != ''">
9.                 AND name LIKE '%${name}%'
10.            </if>
11.            <if test="authorId != null">
12.                AND author_id = #{authorId}
13.            </if>
14.        </where>
15.    </select>
```

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

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testSelectList

```

1.     @Test
2.     public void testSelectList() throws IOException {
3.         SqlSession session = sqlSessionFactory.openSession(); //
        ExecutorType.BATCH
4.         try {
5.             BlogMapper mapper = session.getMapper(BlogMapper.class);
6.             Blog blog = new Blog();
7.             blog.setName("改行");
8.             List<Blog> list1 = new ArrayList<Blog>();
9.             List<Blog> list2 = new ArrayList<Blog>();
10.            list1 = mapper.selectBlogListIf(blog);
11.            list2 = mapper.selectBlogListChoose(blog);
12.        } finally {
13.            session.close();
14.        }
15.    }

```

```

1.     <!--动态SQL where 和 choose...when...otherwise...-->
2.     <select id="selectBlogListChoose" parameterType="blog"
        resultMap="BaseResultMap">
3.         SELECT bid, name, author_id FROM blog
4.         <where>
5.             <choose>
6.                 <when test="bid != null">
7.                     AND bid = #{bid, jdbcType=INTEGER}
8.                 </when>
9.                 <when test="name != null and name != ''">
10.                    AND name LIKE CONCAT(CONCAT('%', #{name, jdbcType=VARCHAR}),
11.                    '%')
12.                </when>
13.                <when test="authorId != null">
14.                    AND author_id = #{authorId, jdbcType=INTEGER}
15.                </when>
16.                <otherwise></otherwise>
17.            </choose>
18.        </where>
19.    </select>

```

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

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testUpdate

```

1.  @Test
2.  public void testUpdate() throws IOException {
3.      SqlSession session = sqlSessionFactory.openSession();
4.      try {
5.          BlogMapper mapper = session.getMapper(BlogMapper.class);
6.          Blog newBlog = new Blog();
7.          newBlog.setBid(333);
8.          newBlog.setName("修改以后的名字");
9.          mapper.updateByPrimaryKey(newBlog);
10.         session.commit();
11.     } finally {
12.         session.close();
13.     }
14. }

```

```

1.  <!-- 动态SQL set -->
2.  <update id="updateByPrimaryKey" parameterType="blog">
3.      update blog
4.      <set>
5.          <if test="name != null">
6.              name = #{name,jdbcType=VARCHAR},
7.          </if>
8.          <if test="authorId != null">
9.              author_id = #{authorId,jdbcType=CHAR},
10.         </if>
11.     </set>
12.     where bid = #{bid,jdbcType=INTEGER}
13. </update>

```

1.3.1.2.4. trim-用来指定或者去掉前缀或者后缀

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testInsert

```

1.  @Test
2.  public void testInsert() throws IOException {
3.      SqlSession session = sqlSessionFactory.openSession();
4.      try {
5.          BlogMapper mapper = session.getMapper(BlogMapper.class);
6.          Blog blog = new Blog();
7.          blog.setBid(1688);
8.          blog.setName("测试插入");
9.          blog.setAuthorId(1111);
10.         System.out.println(mapper.insertBlog(blog));
11.         session.commit();
12.     } finally {
13.         session.close();
14.     }
15. }

```

```

1.  <!-- 动态SQL trim -->
2.  <insert id="insertBlog" parameterType="blog">
3.      insert into blog
4.      <trim prefix="(" suffix=")" suffixOverrides=",">
5.          <if test="bid != null">
6.              bid,
7.          </if>
8.          <if test="name != null">
9.              name,
10.         </if>
11.         <if test="authorId != null">
12.             author_id,
13.         </if>
14.     </trim>
15.     <trim prefix="values (" suffix=")" suffixOverrides=",">
16.         <if test="bid != null">
17.             #{bid,jdbcType=INTEGER},
18.         </if>
19.         <if test="name != null">
20.             #{name,jdbcType=VARCHAR},
21.         <!-- #
22.         {name,jdbcType=VARCHAR,typeHandler=cn.sitedev.type.MyTypeHandler}, -->
23.         </if>
24.         <if test="authorId != null">
25.             #{authorId,jdbcType=INTEGER},
26.         </if>
27.     </trim>
28. </insert>

```

1.3.1.2.5 foreach——需要遍历集合的时候:

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testDelete

```

1.     @Test
2.     public void testDelete() throws IOException {
3.         SqlSession session = sqlSessionFactory.openSession();
4.         try {
5.             BlogMapper mapper = session.getMapper(BlogMapper.class);
6.             List<Blog> list = new ArrayList<Blog>();
7.             Blog blog1 = new Blog();
8.             blog1.setBid(666);
9.             list.add(blog1);
10.            Blog blog2 = new Blog();
11.            blog2.setBid(777);
12.            list.add(blog2);
13.            mapper.deleteByList(list);
14.        } finally {
15.            session.close();
16.        }
17.    }

```

```

1.     <!-- foreach 动态SQL 批量删除 -->
2.     <delete id="deleteByList" parameterType="java.util.List">
3.         delete from blog where bid in
4.         <foreach collection="list" item="item" open="(" separator="," close=")">
5.             #{item.bid,jdbcType=INTEGER}
6.         </foreach>
7.     </delete>

```

动态SQL 主要是用来解决SQL语句生成的问题。

1.3.2. 批量操作

我们在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议。即使在同一个连接中，也有重复编译和执行SQL的开销。

例如循环插入10000条（大约耗时3秒钟）：

单元测试类参见mybatis-standalone-lesson工程
cn.sitedev.BatchOperateTest#testInsertOneByOne

```

1.  @Test
2.  public void testInsertOneByOne() {
3.      long start = System.currentTimeMillis();
4.      int count = 12000;
5.      for (int i = 2000; i < count; i++) {
6.          Blog blog = new Blog();
7.          blog.setBid(i);
8.          blog.setName("name" + i);
9.          blog.setAuthorId(i);
10.         mapper.insertBlog(blog);
11.     }
12.     session.commit();
13.     session.close();
14.     long end = System.currentTimeMillis();
15.     System.out.println("循环批量插入" + count + "条，耗时：" + (end - start) +
        "毫秒");
16. }

```

```

1.  <!-- 动态SQL trim -->
2.  <insert id="insertBlog" parameterType="blog">
3.      insert into blog
4.      <trim prefix="(" suffix=)" suffixOverrides=",">
5.          <if test="bid != null">
6.              bid,
7.          </if>
8.          <if test="name != null">
9.              name,
10.         </if>
11.         <if test="authorId != null">
12.             author_id,
13.         </if>
14.     </trim>
15.     <trim prefix="values (" suffix=)" suffixOverrides=",">
16.         <if test="bid != null">
17.             #{bid,jdbcType=INTEGER},
18.         </if>
19.         <if test="name != null">
20.             #{name,jdbcType=VARCHAR},
21.         <!-- #
{name,jdbcType=VARCHAR,typeHandler=cn.sitedev.type.MyTypeHandler}, -->
22.         </if>
23.         <if test="authorId != null">
24.             #{authorId,jdbcType=INTEGER},
25.         </if>
26.     </trim>
27. </insert>

```

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。

1.3.2.1. 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。

```
1. insert into tbl_emp(emp_id,emp name,gender,email,d_id) values (?, ?, ?, ?, ?),
   (?, ?, ?, ?, ?), (?, ?, ?, ?, ?), (?, ?, ?, 2, ?), (?, ?, ?, 2, ?), (?, ?, 2, 2, ?), (?, ?, ?, 2, ?),
   (?, ?, ?, 2, ?), (?, 2, ?, 2, ?), (?, 2, ?, 2, ?)
```

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

```
1. <!-- foreach 动态SQL 批量插入 -->
2. <insert id="insertBlogList" parameterType="java.util.List">
3.     insert into blog (bid, name, author_id)
4.     values
5.         <foreach collection="list" item="blogs" index="index" separator=",">
6.             ( #{blogs.bid},#{blogs.name},#{blogs.authorId} )
7.         </foreach>
8. </insert>
```

Java 代码里面，直接传入一个List类型的参数。

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.BatchOperateTest#testInsert


```

1.  /**
2.   * MyBatis 动态SQL批量插入
3.   *
4.   * @throws IOException
5.   */
6.  @Test
7.  public void testInsert() throws IOException {
8.      long start = System.currentTimeMillis();
9.      int count = 12000;
10.     List<Blog> list = new ArrayList<Blog>();
11.     for (int i = 2000; i < count; i++) {
12.         Blog blog = new Blog();
13.         blog.setBid(i);
14.         blog.setName("name" + i);
15.         blog.setAuthorId(i);
16.         list.add(blog);
17.     }
18.     mapper.insertBlogList(list);
19.     session.commit();
20.     session.close();
21.     long end = System.currentTimeMillis();
22.     System.out.println("动态SQL批量插入" + count + "条, 耗时: " + (end -
23.         start) + "毫秒");

```

插入一万条大约耗时1秒钟。

可以看到，动态SQL 批量插入效率要比循环发送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。

1.3.2.2. 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。

```

1.  update blog set
2.  name =
3.  case bid
4.  when ? then ?
5.  when ? then ?
6.  when ? then ? end,
7.  author id =
8.  case bid
9.  when ? then ?
10. when ? then ?
11. when ? then ? end
12. where bid 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。

```
1.      <!-- foreach 动态SQL 批量更新-->
2.      <update id="updateBlogList">
3.          update blog set
4.          name =
5.          <foreach collection="list" item="blogs" index="index" separator=" "
open="case bid" close="end">
6.              when #{blogs.bid} then #{blogs.name}
7.          </foreach>
8.          ,author_id =
9.          <foreach collection="list" item="blogs" index="index" separator=" "
open="case bid" close="end">
10.             when #{blogs.bid} then #{blogs.authorId}
11.          </foreach>
12.          where bid in
13.          <foreach collection="list" item="item" open="(" separator="," close=")">
14.              #{item.bid,jdbcType=INTEGER}
15.          </foreach>
16.      </update>
```

单元测试类参见mybatis-standalone-lesson工程
cn.sitedev.BatchOperateTest#updateBlogList

```
1.      @Test
2.      public void updateBlogList() throws IOException {
3.          long start = System.currentTimeMillis();
4.          int count = 12000;
5.          List<Blog> list = new ArrayList<Blog>();
6.          for (int i = 2000; i < count; i++) {
7.              Blog blog = new Blog();
8.              blog.setBid(i);
9.              blog.setName("modified name" + i);
10.             blog.setAuthorId(i);
11.             list.add(blog);
12.         }
13.         mapper.updateBlogList(list);
14.         session.commit();
15.         session.close();
16.         long end = System.currentTimeMillis();
17.         System.out.println("批量更新" + count + "条, 耗时: " + (end - start) + "毫
秒");
18.     }
```

批量删除也是类似的。

```

1.      <!-- foreach 动态SQL 批量删除 -->
2.      <delete id="deleteByList" parameterType="java.util.List">
3.          delete from blog where bid in
4.          <foreach collection="list" item="item" open="(" separator="," close=")">
5.              #{item.bid,jdbcType=INTEGER}
6.          </foreach>
7.      </delete>

```

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testDelete

```

1.      @Test
2.      public void testDelete() throws IOException {
3.          SqlSession session = sqlSessionFactory.openSession();
4.          try {
5.              BlogMapper mapper = session.getMapper(BlogMapper.class);
6.              List<Blog> list = new ArrayList<Blog>();
7.              Blog blog1 = new Blog();
8.              blog1.setBid(666);
9.              list.add(blog1);
10.             Blog blog2 = new Blog();
11.             blog2.setBid(777);
12.             list.add(blog2);
13.             mapper.deleteByList(list);
14.         } finally {
15.             session.close();
16.         }
17.     }

```

1.3.2.3. 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，需要修改默认配置或者手动地控制条数，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（默认是SIMPLE）。

其中有一种BatchExecutor。

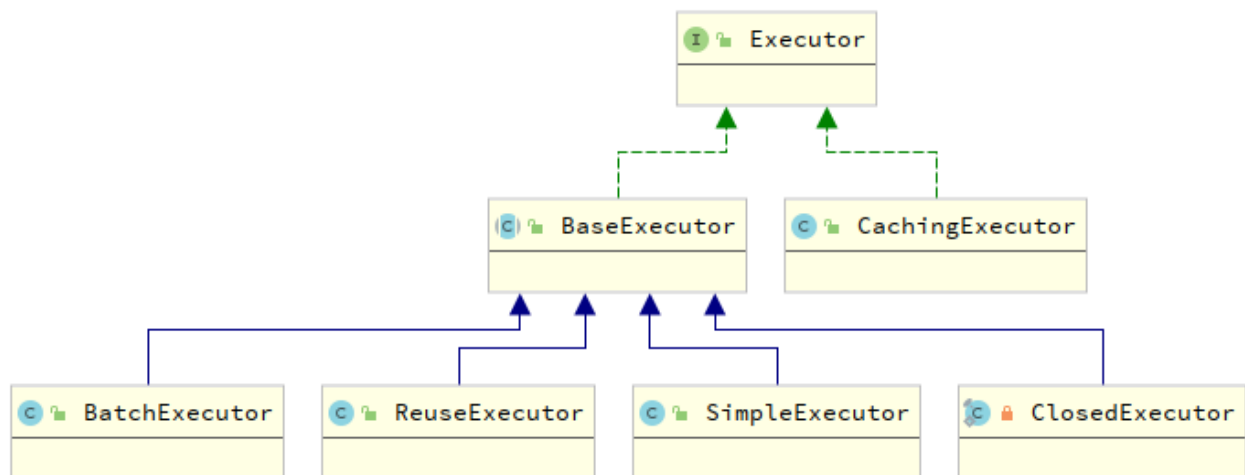
```

1.      <setting name="defaultExecutorType" value="BATCH">

```

也可以在创建会话的时候指定执行器类型：

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



思考：三种类型的区别？（通过doUpdate（）方法对比）

1) SimpleExecutor: 每执行一次update或select, 就开启一个Statement对象, 用完立刻关闭Statement对象。

2) ReuseExecutor: 执行 update 或select, 以sql作为key 查找Statement对象, 存在就使用, 不存在就创建, 用完后, 不关闭Statement对象, 而是放置于Map内, 供下一次使用。简言之, 就是重复使用Statement对象。

3) BatchExecutor: 执行 update (没有 select, JDBC批处理不支持select), 将所有sql都添加到批处理中 (addBatch ()), 等待统一执行 (executeBatch ()), 它缓存了多个Statement对象, 每个Statement对象都是addBatch () 完毕后, 等待逐一执行executeBatch () 批处理。与JDBC批处理相同。

executeUpdate () 是一个语句访问一次数据库, executeBatch()是一批语句访问一次数据库 (具体一批发送多少条SQL跟服务端的 max_allowed_packet有关) 。

BatchExecutor 底层是对JDBC ps.addBatch () 和ps.executeBatch () 的封装。

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.JdbcTest#testJdbcBatch

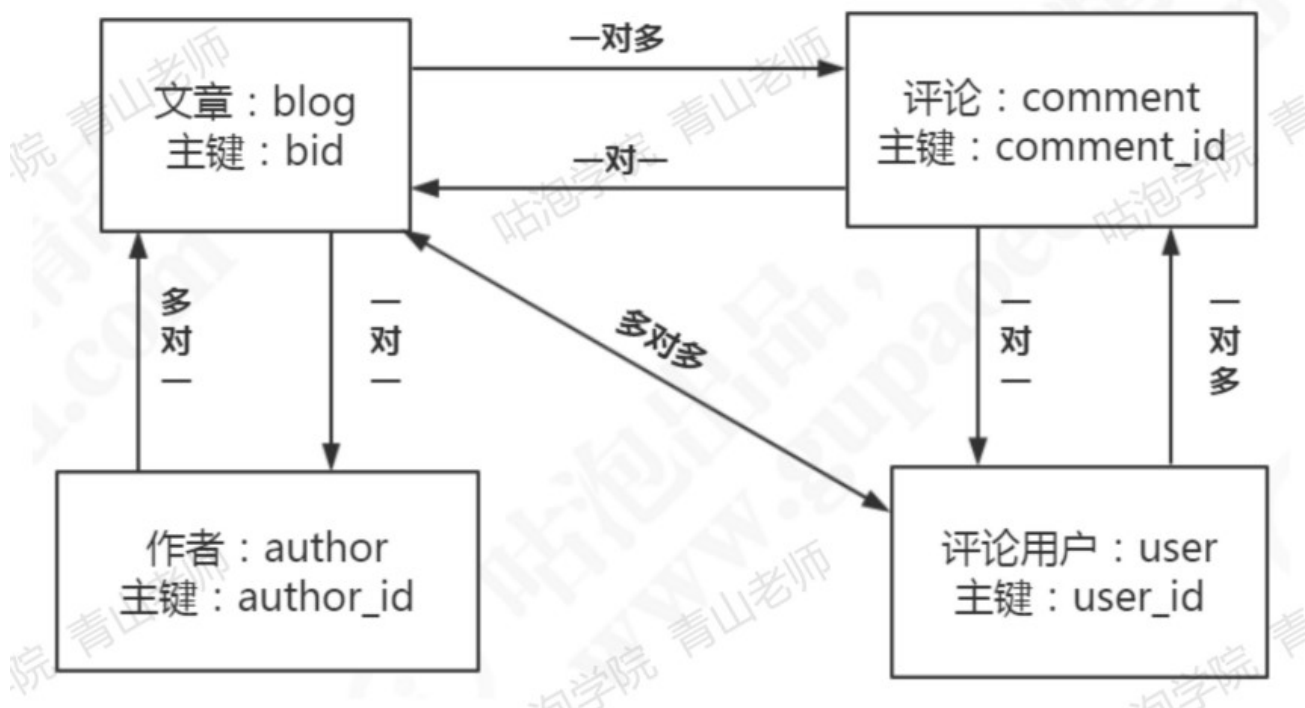
```

1.     @Test
2.     public void testJdbcBatch() throws IOException {
3.         Connection conn = null;
4.         PreparedStatement ps = null;
5.
6.         try {
7.             Long start = System.currentTimeMillis();
8.             // 打开连接
9.             conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis?
useUnicode=true&characterEncoding"
10.                             + "utf-8&rewriteBatchedStatements=true", "root", "root");
11.             ps = conn.prepareStatement("INSERT into blog values (?, ?, ?)");
12.
13.             for (int i = 1000; i < 101000; i++) {
14.                 Blog blog = new Blog();
15.                 ps.setInt(1, i);
16.                 ps.setString(2, String.valueOf(i) + "");
17.                 ps.setInt(3, 1001);
18.                 ps.addBatch();
19.             }
20.
21.             ps.executeBatch();
22.             // conn.commit();
23.             ps.close();
24.             conn.close();
25.             Long end = System.currentTimeMillis();
26.             System.out.println("cost:" + (end - start) + "ms");
27.         } catch (SQLException se) {
28.             se.printStackTrace();
29.         } catch (Exception e) {
30.             e.printStackTrace();
31.         } finally {
32.             try {
33.                 if (ps != null) ps.close();
34.             } catch (SQLException se2) {
35.             }
36.             try {
37.                 if (conn != null) conn.close();
38.             } catch (SQLException se) {
39.                 se.printStackTrace();
40.             }
41.         }
42.     }

```

1.3.3. 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关联查询的情况，比如查询员工就会关联部门（一对一），查询学生成绩就会关联课程（一对一），查询订单就会关联商品（一对多），等等。



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（多对一），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。

association 和collection的区别：

association 是用于一对一和多对一，而collection是用于一对多的关系。

一对一的关联查询有两种配置方式：

1、嵌套结果：

单元测试类参见mybatis-standalone-lesson工程

cn.sitedev.MyBatisTest#testSelectBlogWithAuthorResult

```

1.  /**
2.   * 一对一，一篇文章对应一个作者
3.   * 嵌套结果，不存在N+1问题
4.   */
5.  @Test
6.  public void testSelectBlogWithAuthorResult() throws IOException {
7.      SqlSession session = sqlSessionFactory.openSession();
8.      BlogMapper mapper = session.getMapper(BlogMapper.class);
9.
10.     BlogAndAuthor blog = mapper.selectBlogWithAuthorResult(1);
11.     System.out.println("-----:" + blog);
12. }

```

```

1.  <!-- 根据文章查询作者，一对一，嵌套结果，无N+1问题 -->
2.  <select id="selectBlogWithAuthorResult" resultMap="BlogWithAuthorResultMap">
3.      select b.bid, b.name, b.author_id, a.author_id , a.author_name
4.      from blog b
5.      left join author a
6.      on b.author_id=a.author_id
7.      where b.bid = #{bid, jdbcType=INTEGER}
8.  </select>
9.
10. <!--根据文章查询作者，一对一查询结果，嵌套查询-->
11. <resultMap id="BlogWithAuthorResultMap"
12. type="cn.sitedev.domain.associate.BlogAndAuthor">
13.     <id column="bid" property="bid" jdbcType="INTEGER"/>
14.     <result column="name" property="name" jdbcType="VARCHAR"/>
15.     <!--联合查询，将author的属性映射到ResultMap-->
16.     <association property="author" javaType="cn.sitedev.domain.Author">
17.         <id column="author_id" property="authorId"/>
18.         <result column="author_name" property="authorName"/>
19.     </association>
20. </resultMap>

```

2、嵌套查询：

单元测试类参见mybatis-standalone-lesson工程

cn.sitedev.MyBatisTest#testSelectBlogWithAuthorQuery

```

1.  /**
2.   * 一对一，一篇文章对应一个作者
3.   * 嵌套查询，会有N+1的问题
4.   */
5.  @Test
6.  public void testSelectBlogWithAuthorQuery() throws IOException {
7.      SqlSession session = sqlSessionFactory.openSession();
8.      BlogMapper mapper = session.getMapper(BlogMapper.class);
9.
10.     BlogAndAuthor blog = mapper.selectBlogWithAuthorQuery(1);
11.     System.out.println("-----:" + blog.getClass());
12.     // 如果开启了延迟加载(lazyLoadingEnabled=true)，会在使用的时候才发出SQL
13.     // equals,clone,hashCode,toString也会触发延迟加载
14.     System.out.println("-----调用toString方法:" + blog);
15.     System.out.println("-----getAuthor:" + blog.getAuthor().toString());
16.     // 如果 aggressiveLazyLoading = true ，也会触发加载，否则不会
17.     //System.out.println("-----getName:"+blog.getName());
18. }

```

```

1.  <!-- 根据文章查询作者，一对一，嵌套查询，存在N+1问题，可通过开启延迟加载解决 -->
2.
3.  <select id="selectBlogWithAuthorQuery" resultMap="BlogWithAuthorQueryMap">
4.      select b.bid, b.name, b.author_id, a.author_id , a.author_name
5.      from blog b
6.      left join author a
7.      on b.author_id=a.author_id
8.      where b.bid = #{bid, jdbcType=INTEGER}
9.  </select>
10.
11.  <!--另一种联合查询(一对一)实现，但是这种方式有"N+1"问题-->
12.  <resultMap id="BlogWithAuthorQueryMap"
13.  type="cn.sitedev.domain.associate.BlogAndAuthor">
14.      <id column="bid" property="bid" jdbcType="INTEGER"/>
15.      <result column="name" property="name" jdbcType="VARCHAR"/>
16.      <association property="author" javaType="cn.sitedev.domain.Author"
17.      column="author_id" select="selectAuthor"/>
18.      <!--selectAuthor定义在下面-->
19.  </resultMap>
20.
21.  <!-- 嵌套查询 -->
22.  <select id="selectAuthor" parameterType="int"
23.  resultType="cn.sitedev.domain.Author">
24.      select author_id authorId, author_name authorName
25.      from author where author_id = #{authorId}
26.  </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Blog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Blog信息的SQL（所谓的1），如果返回了N条记录（比如10条Blog），因为一个Blog就有至少一个Author，就会再发送N条到数据库查询Author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Author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：

```
1.      <!-- 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 ->
2.      <setting name="lazyLoadingEnabled" value="true"/>
3.      <!-- 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-->
4.      <setting name="aggressiveLazyLoading" value="false"/>
5.      <!-- 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 --
6.      <setting name="proxyFactory" value="CGLIB" />
```

lazyLoadingEnabled 决定了是否延迟加载（默认false）。

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（lazyLoadingEnabled 没有配置成true），会连续发送两次查询；

```
1.      @Test
2.      public void testSelectBlogWithAuthorQuery() throws IOException {
3.          SqlSession session = sqlSessionFactory.openSession();
4.          BlogMapper mapper = session.getMapper(BlogMapper.class);
5.
6.          BlogAndAuthor blog = mapper.selectBlogWithAuthorQuery(1);
7.          System.out.println("-----:" + blog.getClass());
8.          // 如果开启了延迟加载(lazyLoadingEnabled=true)，会在使用的时候才发出SQL
9.          // equals,clone,hashCode,toString也会触发延迟加载
10.         //      System.out.println("-----调用toString方法:" + blog);
11.         //      System.out.println("-----getAuthor:" +
            blog.getAuthor().toString());
12.         // 如果 aggressiveLazyLoading = true ，也会触发加载，否则不会
13.         //System.out.println("-----getName:"+blog.getName());
14.     }
```

```
MyBatisTest.testSelectBlogWithAuth...
Tests passed: 1 of 1 test - 1s 88ms

MyBatisTest (cn.sitedev) 1s 88ms
testSelectBlogWithAuth 1s 88ms

Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 2138564891.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
==> Preparing: select b.bid, b.name, b.author_id, a.author_id, a.author_name from blog b left join author a on
b.author_id=a.author_id where b.bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id, author_id, author_name
<== Row: 1, RabbitMQ延时消息, 1001, 1001, 青山
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
====> Preparing: select author_id, author_name, authorName from author where author_id = ?
====> Parameters: 1001(Integer)
<==== Columns: authorId, authorName
<==== Row: 1001, 青山
<==== Total: 1
<== Total: 1
-----:class cn.sitedev.domain.associate.BlogAndAuthor

Process finished with exit code 0
```

I

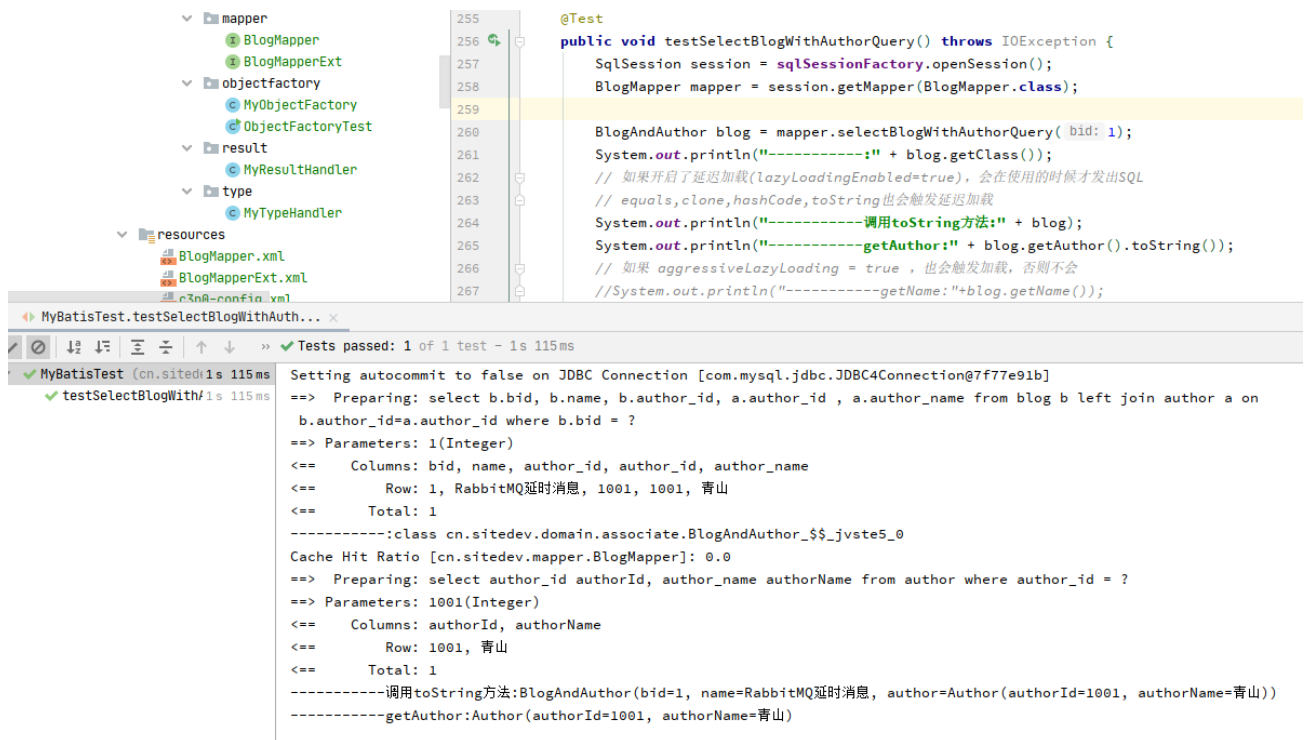
2、开启了延迟加载的开关，调用blog.getAuthor()以及默认的（equals, clone, hashCode, toString）时才会发起第二次查询，其他方法并不会触发查询，比如blog.getName();

```
1. @Test
2. public void testSelectBlogWithAuthorQuery() throws IOException {
3.     SqlSession session = sqlSessionFactory.openSession();
4.     BlogMapper mapper = session.getMapper(BlogMapper.class);
5.
6.     BlogAndAuthor blog = mapper.selectBlogWithAuthorQuery(1);
7.     System.out.println("-----:" + blog.getClass());
8.     // 如果开启了延迟加载(lazyLoadingEnabled=true)，会在使用的时候才发出SQL
9.     // equals, clone, hashCode, toString也会触发延迟加载
10.    System.out.println("-----调用toString方法:" + blog);
11.    System.out.println("-----getAuthor:" + blog.getAuthor().toString());
12.    // 如果 aggressiveLazyLoading = true，也会触发加载，否则不会
13.    //System.out.println("-----getName:" + blog.getName());
14. }
```

```
mapper
├── BlogMapper
├── BlogMapperExt
├── objectFactory
│   ├── MyObjectFactory
│   └── ObjectFactoryTest
├── result
│   └── MyResultHandler
├── type
│   └── MyTypeHandler
└── resources
    ├── BlogMapper.xml
    ├── BlogMapperExt.xml
    └── c3p0-config.xml

255 @Test
256 public void testSelectBlogWithAuthorQuery() throws IOException {
257     SqlSession session = sqlSessionFactory.openSession();
258     BlogMapper mapper = session.getMapper(BlogMapper.class);
259
260     BlogAndAuthor blog = mapper.selectBlogWithAuthorQuery(1);
261     System.out.println("-----:" + blog.getClass());
262     // 如果开启了延迟加载(lazyLoadingEnabled=true)，会在使用的时候才发出SQL
263     // equals, clone, hashCode, toString也会触发延迟加载
264     System.out.println("-----调用toString方法:" + blog);
265     System.out.println("-----getAuthor:" + blog.getAuthor().toString());
266     // 如果 aggressiveLazyLoading = true，也会触发加载，否则不会
267     //System.out.println("-----getName:" + blog.getName());

PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 2138564891.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
==> Preparing: select b.bid, b.name, b.author_id, a.author_id, a.author_name from blog b left join author a on
b.author_id=a.author_id where b.bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id, author_id, author_name
<== Row: 1, RabbitMQ延时消息, 1001, 1001, 青山
<== Total: 1
-----:class cn.sitedev.domain.associate.BlogAndAuthor_$$jvste5_0
```



触发延迟加载的方法可以通过 `<lazyLoadTriggerMethods>` 配置，默认 `equals ()`，`clone ()`，`hashCode ()`，`toString ()`。

3、如果开启了 `aggressiveLazyLoading=true`，除了关联查询的方法和系统方法，其他方法也会触发第二次查询，比如 `blog.getName ()`。

```
1. @Test
2. public void testSelectBlogWithAuthorQuery() throws IOException {
3.     SqlSession session = sqlSessionFactory.openSession();
4.     BlogMapper mapper = session.getMapper(BlogMapper.class);
5.
6.     BlogAndAuthor blog = mapper.selectBlogWithAuthorQuery(1);
7.     System.out.println("-----:" + blog.getClass());
8.     // 如果开启了延迟加载(lazyLoadingEnabled=true), 会在使用的时候才发出SQL
9.     // equals, clone, hashCode, toString 也会触发延迟加载
10.    // System.out.println("-----调用toString方法:" + blog);
11.    // System.out.println("-----getAuthor:" +
    blog.getAuthor().toString());
12.    // 如果 aggressiveLazyLoading = true, 也会触发加载, 否则不会
13.    System.out.println("-----getName:" + blog.getName());
14. }
```

```
MyBatisTest.testSelectBlogWithAuth...
Tests passed: 1 of 1 test - 1s 26ms
Logging initialized using class org.springframework.boot.logging.logback.LogbackConsoleAppender
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 2138564891.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
==> Preparing: select b.bid, b.name, b.author_id, a.author_id, a.author_name from blog b left join author a on
b.author_id=a.author_id where b.bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id, author_id, author_name
<== Row: 1, RabbitMQ延时消息, 1001, 1001, 青山
<== Total: 1
-----:class cn.sitedev.domain.associate.BlogAndAuthor_$$jvste5_0
-----getName:RabbitMQ延时消息

Process finished with exit code 0
```

<setting name="aggressiveLazyLoading" value="false"/>

```
MyBatisTest.testSelectBlogWithAuth...
Tests passed: 1 of 1 test - 1s 100ms
Opening JDBC Connection
Created connection 2138564891.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
==> Preparing: select b.bid, b.name, b.author_id, a.author_id, a.author_name from blog b left join author a on
b.author_id=a.author_id where b.bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id, author_id, author_name
<== Row: 1, RabbitMQ延时消息, 1001, 1001, 青山
<== Total: 1
-----:class cn.sitedev.domain.associate.BlogAndAuthor_$$jvste5_0
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
==> Preparing: select author_id authorId, author_name authorName from author where author_id = ?
==> Parameters: 1001(Integer)
<== Columns: authorId, authorName
<== Row: 1001, 青山
<== Total: 1
-----getName:RabbitMQ延时消息

Process finished with exit code 0
```

<setting name="aggressiveLazyLoading" value="true"/>

思考：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blog打印出来看看，果然不对：

```
1. System.out.println("-----:" + blog.getClass());
```

打印内容：

```
1. -----:class cn.sitedev.domain.associate.BlogAndAuthor_$$jvste5_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。后面看源码的时候大家可以去跟踪一下，看看是什么时候变成代理对象的。

1.3.4. 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。

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。

1.3.4.1. 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。

接口：BlogMapperjava

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

单元测试类参见mybatis-standalone-lesson工程
cn.sitedev.MyBatisTest#testSelectByRowBounds

```
1.      @Test
2.      public void testSelectByRowBounds() throws IOException {
3.          SqlSession session = sqlSessionFactory.openSession();
4.          try {
5.              BlogMapper mapper = session.getMapper(BlogMapper.class);
6.              int start = 0; // offset
7.              int pageSize = 5; // limit
8.              RowBounds rb = new RowBounds(start, pageSize);
9.              List<Blog> list = mapper.selectBlogList(rb); // 使用逻辑分页
10.             for (Blog b : list) {
11.                 System.out.println(b);
12.             }
13.         } finally {
14.             session.close();
15.         }
16.     }
```

RowBounds的工作原理其实是对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。

```

1. public class DefaultResultSetHandler implements ResultSetHandler {
2.     private void handleRowValuesForSimpleResultMap(ResultSetWrapper rsw, ResultMap
resultMap, ResultHandler<?> resultHandler, RowBounds rowBounds, ResultMapping
parentMapping) throws SQLException {
3.         DefaultResultContext<Object> resultContext = new DefaultResultContext();
4.         ResultSet resultSet = rsw.getResultSet();
5.         this.skipRows(resultSet, rowBounds);
6.
7.         while(this.shouldProcessMoreRows(resultContext, rowBounds) &&
!resultSet.isClosed() && resultSet.next()) {
8.             ResultMap discriminatedResultMap =
this.resolveDiscriminatedResultMap(resultSet, resultMap, (String)null);
9.             Object rowValue = this.getRowValue(rsw, discriminatedResultMap,
(String)null);
10.            this.storeObject(resultHandler, resultContext, rowValue,
parentMapping, resultSet);
11.        }
12.
13.    }

```

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。

1.3.4.2. 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。

```

1. <select id="selectBlogPage" parameterType="map" resultMap="BaseResultMap">
2.     select * from blog limit #{curIndex} , #{pageSize}
3. </select>

```

第一个问题是我们要在Java业务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，我们只要传入当前是第几页，每页多少条就可以了，自动计算出来起止序号。我们最常用的做法就是使用翻页的插件，比如PageHelper。

```

1. // pageSize每一页几条
2. PageHelper.startPage(pn, 10);
3. List<Employee> emps = employeeService.getAll();
4. // navigatePages 导航页码数
5. PageInfo page = new PageInfo(emps, 10);
6. 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。

1.3.5. 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基本操作是相同的，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？

MyBatis 也提供了一个代码生成器，叫做MyBatis Generator，简称MBG（它是MyBatis的一个插件）。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。

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。

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。

参见mybatis-standalone-lesson工程

cn.sitedev.domain.BlogExample,cn.sitedev.BlogExampleTest


```
1. package cn.sitedev.domain;
2.
3. import lombok.Data;
4.
5. import java.util.ArrayList;
6. import java.util.List;
7.
8. @Data
9. public class BlogExample {
10.     protected String orderByClause;
11.
12.     protected boolean distinct;
13.
14.     protected List<Criteria> oredCriteria;
15.
16.     public BlogExample() {
17.         oredCriteria = new ArrayList<Criteria>();
18.     }
19.
20.     public void or(Criteria criteria) {
21.         oredCriteria.add(criteria);
22.     }
23.
24.     public Criteria or() {
25.         Criteria criteria = createCriteriaInternal();
26.         oredCriteria.add(criteria);
27.         return criteria;
28.     }
29.
30.     public Criteria createCriteria() {
31.         Criteria criteria = createCriteriaInternal();
32.         if (oredCriteria.size() == 0) {
33.             oredCriteria.add(criteria);
34.         }
35.         return criteria;
36.     }
37.
38.     protected Criteria createCriteriaInternal() {
39.         Criteria criteria = new Criteria();
40.         return criteria;
41.     }
42.
43.     public void clear() {
44.         oredCriteria.clear();
45.         orderByClause = null;
46.         distinct = false;
47.     }
48.
49.     protected abstract static class GeneratedCriteria {
```



```

50.         protected List<Criterion> criteria;
51.
52.         protected GeneratedCriteria() {
53.             super();
54.             criteria = new ArrayList<Criterion>();
55.         }
56.
57.         public boolean isValid() {
58.             return criteria.size() > 0;
59.         }
60.
61.         public List<Criterion> getAllCriteria() {
62.             return criteria;
63.         }
64.
65.         public List<Criterion> getCriteria() {
66.             return criteria;
67.         }
68.
69.         protected void addCriterion(String condition) {
70.             if (condition == null) {
71.                 throw new RuntimeException("Value for condition cannot be null");
72.             }
73.             criteria.add(new Criterion(condition));
74.         }
75.
76.         protected void addCriterion(String condition, Object value, String
property) {
77.             if (value == null) {
78.                 throw new RuntimeException("Value for " + property + " cannot be
null");
79.             }
80.             criteria.add(new Criterion(condition, value));
81.         }
82.
83.         protected void addCriterion(String condition, Object value1, Object
value2, String property) {
84.             if (value1 == null || value2 == null) {
85.                 throw new RuntimeException("Between values for " + property + "
cannot be null");
86.             }
87.             criteria.add(new Criterion(condition, value1, value2));
88.         }
89.
90.         public Criteria andBidIsNull() {
91.             addCriterion("bid is null");
92.             return (Criteria) this;
93.         }
94.

```

```
95.     public Criteria andBidIsNotNull() {
96.         addCriterion("bid is not null");
97.         return (Criteria) this;
98.     }
99.
100.    public Criteria andBidEqualTo(Integer value) {
101.        addCriterion("bid =", value, "bid");
102.        return (Criteria) this;
103.    }
104.
105.    public Criteria andBidNotEqualTo(Integer value) {
106.        addCriterion("bid <>", value, "bid");
107.        return (Criteria) this;
108.    }
109.
110.    public Criteria andBidGreaterThan(Integer value) {
111.        addCriterion("bid >", value, "bid");
112.        return (Criteria) this;
113.    }
114.
115.    public Criteria andBidGreaterThanOrEqualTo(Integer value) {
116.        addCriterion("bid >=", value, "bid");
117.        return (Criteria) this;
118.    }
119.
120.    public Criteria andBidLessThan(Integer value) {
121.        addCriterion("bid <", value, "bid");
122.        return (Criteria) this;
123.    }
124.
125.    public Criteria andBidLessThanOrEqualTo(Integer value) {
126.        addCriterion("bid <=", value, "bid");
127.        return (Criteria) this;
128.    }
129.
130.    public Criteria andBidIn(List<Integer> values) {
131.        addCriterion("bid in", values, "bid");
132.        return (Criteria) this;
133.    }
134.
135.    public Criteria andBidNotIn(List<Integer> values) {
136.        addCriterion("bid not in", values, "bid");
137.        return (Criteria) this;
138.    }
139.
140.    public Criteria andBidBetween(Integer value1, Integer value2) {
141.        addCriterion("bid between", value1, value2, "bid");
142.        return (Criteria) this;
143.    }
```

```
144.
145.     public Criteria andBidNotBetween(Integer value1, Integer value2) {
146.         addCriterion("bid not between", value1, value2, "bid");
147.         return (Criteria) this;
148.     }
149.
150.     public Criteria andNameIsNull() {
151.         addCriterion("name is null");
152.         return (Criteria) this;
153.     }
154.
155.     public Criteria andNameIsNotNull() {
156.         addCriterion("name is not null");
157.         return (Criteria) this;
158.     }
159.
160.     public Criteria andNameEqualTo(String value) {
161.         addCriterion("name =", value, "name");
162.         return (Criteria) this;
163.     }
164.
165.     public Criteria andNameNotEqualTo(String value) {
166.         addCriterion("name <>", value, "name");
167.         return (Criteria) this;
168.     }
169.
170.     public Criteria andNameGreaterThan(String value) {
171.         addCriterion("name >", value, "name");
172.         return (Criteria) this;
173.     }
174.
175.     public Criteria andNameGreaterThanOrEqualTo(String value) {
176.         addCriterion("name >=", value, "name");
177.         return (Criteria) this;
178.     }
179.
180.     public Criteria andNameLessThan(String value) {
181.         addCriterion("name <", value, "name");
182.         return (Criteria) this;
183.     }
184.
185.     public Criteria andNameLessThanOrEqualTo(String value) {
186.         addCriterion("name <=", value, "name");
187.         return (Criteria) this;
188.     }
189.
190.     public Criteria andNameLike(String value) {
191.         addCriterion("name like", value, "name");
192.         return (Criteria) this;
```

```
193.     }
194.
195.     public Criteria andNameNotLike(String value) {
196.         addCriterion("name not like", value, "name");
197.         return (Criteria) this;
198.     }
199.
200.     public Criteria andNameIn(List<String> values) {
201.         addCriterion("name in", values, "name");
202.         return (Criteria) this;
203.     }
204.
205.     public Criteria andNameNotIn(List<String> values) {
206.         addCriterion("name not in", values, "name");
207.         return (Criteria) this;
208.     }
209.
210.     public Criteria andNameBetween(String value1, String value2) {
211.         addCriterion("name between", value1, value2, "name");
212.         return (Criteria) this;
213.     }
214.
215.     public Criteria andNameNotBetween(String value1, String value2) {
216.         addCriterion("name not between", value1, value2, "name");
217.         return (Criteria) this;
218.     }
219.
220.     public Criteria andAuthorIdIsNull() {
221.         addCriterion("author_id is null");
222.         return (Criteria) this;
223.     }
224.
225.     public Criteria andAuthorIdIsNotNull() {
226.         addCriterion("author_id is not null");
227.         return (Criteria) this;
228.     }
229.
230.     public Criteria andAuthorIdEqualTo(Integer value) {
231.         addCriterion("author_id =", value, "authorId");
232.         return (Criteria) this;
233.     }
234.
235.     public Criteria andAuthorIdNotEqualTo(Integer value) {
236.         addCriterion("author_id <>", value, "authorId");
237.         return (Criteria) this;
238.     }
239.
240.     public Criteria andAuthorIdGreaterThan(Integer value) {
241.         addCriterion("author_id >", value, "authorId");
```

```
242.         return (Criteria) this;
243.     }
244.
245.     public Criteria andAuthorIdGreaterThanOrEqualTo(Integer value) {
246.         addCriterion("author_id >=", value, "authorId");
247.         return (Criteria) this;
248.     }
249.
250.     public Criteria andAuthorIdLessThan(Integer value) {
251.         addCriterion("author_id <", value, "authorId");
252.         return (Criteria) this;
253.     }
254.
255.     public Criteria andAuthorIdLessThanOrEqualTo(Integer value) {
256.         addCriterion("author_id <=", value, "authorId");
257.         return (Criteria) this;
258.     }
259.
260.     public Criteria andAuthorIdIn(List<Integer> values) {
261.         addCriterion("author_id in", values, "authorId");
262.         return (Criteria) this;
263.     }
264.
265.     public Criteria andAuthorIdNotIn(List<Integer> values) {
266.         addCriterion("author_id not in", values, "authorId");
267.         return (Criteria) this;
268.     }
269.
270.     public Criteria andAuthorIdBetween(Integer value1, Integer value2) {
271.         addCriterion("author_id between", value1, value2, "authorId");
272.         return (Criteria) this;
273.     }
274.
275.     public Criteria andAuthorIdNotBetween(Integer value1, Integer value2) {
276.         addCriterion("author_id not between", value1, value2, "authorId");
277.         return (Criteria) this;
278.     }
279. }
280.
281. public static class Criteria extends GeneratedCriteria {
282.
283.     protected Criteria() {
284.         super();
285.     }
286. }
287.
288. @Data
289. public static class Criterion {
290.     private String condition;
```

```
291.
292.     private Object value;
293.
294.     private Object secondValue;
295.
296.     private boolean noValue;
297.
298.     private boolean singleValue;
299.
300.     private boolean betweenValue;
301.
302.     private boolean listValue;
303.
304.     private String typeHandler;
305.
306.     protected Criterion(String condition) {
307.         super();
308.         this.condition = condition;
309.         this.typeHandler = null;
310.         this.noValue = true;
311.     }
312.
313.     protected Criterion(String condition, Object value, String typeHandler) {
314.         super();
315.         this.condition = condition;
316.         this.value = value;
317.         this.typeHandler = typeHandler;
318.         if (value instanceof List<?>) {
319.             this.listValue = true;
320.         } else {
321.             this.singleValue = true;
322.         }
323.     }
324.
325.     protected Criterion(String condition, Object value) {
326.         this(condition, value, null);
327.     }
328.
329.     protected Criterion(String condition, Object value, Object secondValue,
String typeHandler) {
330.         super();
331.         this.condition = condition;
332.         this.value = value;
333.         this.secondValue = secondValue;
334.         this.typeHandler = typeHandler;
335.         this.betweenValue = true;
336.     }
337.
338.     protected Criterion(String condition, Object value, Object secondValue) {
```

```

339.         this(condition, value, secondValue, null);
340.     }
341. }
342. }
343. //////////////////////////////////////////////////
344. package cn.sitedev;
345.
346. import cn.sitedev.domain.Blog;
347. import cn.sitedev.domain.BlogExample;
348. import cn.sitedev.mapper.BlogMapper;
349. import org.apache.ibatis.io.Resources;
350. import org.apache.ibatis.session.SqlSession;
351. import org.apache.ibatis.session.SqlSessionFactory;
352. import org.apache.ibatis.session.SqlSessionFactoryBuilder;
353. import org.junit.Test;
354.
355. import java.io.IOException;
356. import java.io.InputStream;
357. import java.util.List;
358.
359. /**
360.  * Example演示
361.  */
362.
363. public class BlogExampleTest {
364.     /**
365.      * 自动生成的Example
366.      *
367.      * @throws IOException
368.      */
369.     @Test
370.     public void TestExample() throws IOException {
371.         String resource = "mybatis-config.xml";
372.         InputStream inputStream = Resources.getResourceAsStream(resource);
373.         SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
374.
375.         SqlSession session = sqlSessionFactory.openSession();
376.         try {
377.             BlogMapper mapper = session.getMapper(BlogMapper.class);
378.             BlogExample example = new BlogExample();
379.             BlogExample.Criteria criteria = example.createCriteria();
380.             criteria.andBidEqualTo(1);
381.             List<Blog> list = mapper.selectByExample(example);
382.             System.out.println(list);
383.         } finally {
384.             session.close();
385.         }
386.     }

```



```

1.     <select id="selectByExample" parameterType="cn.sitedev.domain.BlogExample"
resultMap="BaseResultMap">
2.         select
3.         <if test="distinct">
4.             distinct
5.         </if>
6.         'true' as QUERYID,
7.         <include refid="Base_Column_List"/>
8.         from blog
9.         <if test="_parameter != null">
10.            <include refid="Example_Where_Clause"/>
11.        </if>
12.        <if test="orderByClause != null">
13.            order by ${orderByClause}
14.        </if>
15.    </select>
16.
17.    <resultMap id="BaseResultMap" type="blog">
18.        <id column="bid" property="bid" jdbcType="INTEGER"/>
19.        <result column="name" property="name" jdbcType="VARCHAR"/>
20.        <result column="author_id" property="authorId" jdbcType="INTEGER"/>
21.    </resultMap>
22.
23.    <!-- 自动生成的Example -->
24.    <sql id="Base_Column_List">
25.        bid, name, author_id
26.    </sql>
27.    <sql id="Example_Where_Clause">
28.        <where>
29.            <foreach collection="oredCriteria" item="criteria" separator="or">
30.                <if test="criteria.valid">
31.                    <trim prefix="(" prefixOverrides="and" suffix=")">
32.                        <foreach collection="criteria.criteria" item="criterion">
33.                            <choose>
34.                                <when test="criterion.noValue">
35.                                    and ${criterion.condition}
36.                                </when>
37.                                <when test="criterion.singleValue">
38.                                    and ${criterion.condition} #{criterion.value}
39.                                </when>
40.                                <when test="criterion.betweenValue">
41.                                    and ${criterion.condition} #{criterion.value}
42.                                    and #{criterion.secondValue}
43.                                </when>
44.                                <when test="criterion.listValue">
45.                                    and ${criterion.condition}
46.                                    <foreach close=")"
collection="criterion.value" item="listItem" open="("
separator=",">

```

```

47.             #{listItem}
48.         </foreach>
49.     </when>
50. </choose>
51. </foreach>
52. </trim>
53. </if>
54. </foreach>
55. </where>
56. </sql>

```

BlogExample 里面包含了两个Criteria:

```

v BlogExample
> Criteria
> Criterion
> GeneratedCriteria
m BlogExample()
m clear():void
m createCriteria():Criteria
m createCriteriaInternal():Criteria
m or():Criteria
m or(Criteria):void
f distinct:boolean
f orderByClause:String
f oredCriteria:List<Criteria>

```

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

参见mybatis-standalone-lesson工程cn.sitedev.BlogExampleTest#TestExample

```

1. BlogMapper mapper = session.getMapper(BlogMapper.class);
2. BlogExample example = new BlogExample();
3. BlogExample.Criteria criteria = example.createCriteria();
4. criteria.andBidEqualTo(1);
5. List<Blog> list = mapper.selectByExample(example);
6. System.out.println(list);

```

生成的语句：

```

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

```

1.3.6. 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。

参见mybatis-standalone-lesson工程cn.sitedev.mapper.BlogMapperExt

```
1. package cn.sitedev.mapper;
2.
3. import cn.sitedev.domain.Blog;
4.
5. /**
6.  * 扩展类继承了MBG生成的接口和Statement
7.  */
8. public interface BlogMapperExt extends BlogMapper {
9.     /**
10.      * 根据名称查询文章
11.      *
12.      * @param name
13.      * @return
14.      */
15.     public Blog selectBlogByName(String name);
16.
17. }
```

参见mybatis-standalone-lesson工程BlogMapperExt.xml

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3. "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
4. <mapper namespace="cn.sitedev.mapper.BlogMapperExt">
5.     <!-- 只能继承statement，不能继承sql、resultMap等标签 -->
6.     <resultMap id="BaseResultMap" type="blog">
7.         <id column="bid" property="bid" jdbcType="INTEGER"/>
8.         <result column="name" property="name" jdbcType="VARCHAR"/>
9.         <result column="author_id" property="authorId" jdbcType="INTEGER"/>
10.     </resultMap>
11.     <!-- 在parent xml 和child xml 的 statement id相同的情况下，会使用child xml 的
12. statement id -->
13.     <select id="selectBlogByName" resultMap="BaseResultMap"
14. statementType="PREPARED">
15.         select * from blog where name = #{name}
16.     </select>
17. </mapper>
```

mybatis-config.xml里面也要扫描：

```
1.    <mappers>
2.        <mapper resource="BlogMapper.xml"/>
3.        <mapper resource="BlogMapperExt.xml"/>
4.    </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？

当然可以！

编写一个支持泛型的通用接口，比如叫 `MyBaseMapper<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 MyBaseMapper<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/Mapper/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。

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

```
1.    <bean id="mapperScanner"
2.        class="org.mybatis.spring.mapper.MapperScannerConfigurer">
3.        <property name="basePackage" value="cn.sitedev.crud.dao"/>
4.    </bean>
```

1.3.7. MyBatis-Plus

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。

1.4. MyBatis常见问题

1.4.1. 用注解还是用xml配置？

常用注解：@Insert、@Select、@Update、@Delete、@Param、@Results、@Result

在MyBatis 的工程中，我们有两种配置SQL的方式。一种是在Mapper.xml中集中管理，一种是在Mapper接口上，用注解方式配置SQL。很多同学在工作中可能两种方式都用过。那到底什么时候用XML的方式，什么时候用注解的方式呢？

注解的缺点是SQL无法集中管理，复杂的SQL很难配置。所以建议在业务复杂的项目中只使用XML配置的形式，业务简单的项目中可以使用注解和XML混用的形式。

1.4.2. Mapper接口无法注入或Invalid bound statement (not found)

我们在使用MyBatis的时候可能会遇到Mapper 接口无法注入，或者mapper statement id跟Mapper接口方法无法绑定的情况。基于绑定的要求或者说规范，我们可以从这些地方去检查一下：

- 1、扫描配置，xml文件和Mapper 接口有没有被扫描到
- 2、namespace的值是否和接口全类名一致
- 3、检查对应的sql语句ID是否存在

1.4.3. 怎么获取插入的最新自动生成的ID

在MySQL的插入数据使用自增ID这种场景，有的时候我们需要获得最新的自增ID，比如获取最新的用户ID。常见的做法是执行一次查询，max或者order by 倒序获取最大的ID（低效、存在并发问题）。在MyBatis里面还有一种更简单的方式：

insert成功之后，mybatis会将插入的值自动绑定到插入的对象的Id属性中，我们用getId就能取到最新的ID。

```
1.      <insert id="insert" parameterType="cn.sitedev.domain.Blog">
2.          insert into blog (bid, name, author_id)
3.          values ( #{bid},#{name},#{authorId} )
4.      </insert>
```

```
1.  blogService.addBlog(blog);
2.  System.out.println(blog.getBid());
```

1.4.4. 什么时候用 `#{}` ，什么时候用 `${}` ？

在Mapperxml里面配置传入参数，有两种写法：`#{}` 、`${}` 。作为OGNL表达式，都可以实现参数的替换。

这两种方式的区别在哪里？什么时候应该用哪一种？

要搞清楚这个问题，我们要先来说一下PreparedStatement和Statement的区别。

- 1、两个都是接口，PreparedStatement 是继承自Statement的；
- 2、Statement 处理静态SQL，PreparedStatement 主要用于执行带参数的语句；
- 3、PreparedStatement的addBatch () 方法一次性发送多个查询给数据库；
- 4、PS相似SQL只编译一次（对语句进行了缓存，相当于一个函数），比如语句相同参数不同，可以减少编译次数；
- 5、PS可以防止SQL注入。

MyBatis 任意语句的默认值：PREPARED

这两个符号的解析方式是不一样的：

会解析为Prepared Statement的参数标记符，参数部分用 ? 代替。传入的参数会经过类型检查和安全检查。

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testSelect

```
1.     @Test
2.     public void testSelect() throws IOException {
3.         SqlSession session = sqlSessionFactory.openSession(); //
        ExecutorType.BATCH
4.         try {
5.             BlogMapper mapper = session.getMapper(BlogMapper.class);
6.             Blog blog = mapper.selectBlogById(1);
7.             System.out.println(blog);
8.         } finally {
9.             session.close();
10.        }
11.    }
```

```
1.     <select id="selectBlogById" resultMap="BaseResultMap"
        statementType="PREPARED">
2.         SELECT * FROM blog WHERE bid = #{bid}
3.     </select>
```

```

PooledDataSource forcefully closed/removed all connections.
Cache Hit Ratio [cn.sitedev.mapper.BlogMapper]: 0.0
Opening JDBC Connection
Created connection 2138564891.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<==      Columns: bid, name, author_id
<==      Row: 1, RabbitMQ延时消息, 1001
<==      Total: 1
Blog(bid=1, name=RabbitMQ延时消息, authorId=1001)
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@7f77e91b]
Returned connection 2138564891 to pool.

Process finished with exit code 0

```

\$ 只会做字符串替换，比如参数是"MySQL从入门到改行"，结果如下：

单元测试类参见mybatis-standalone-lesson工程cn.sitedev.MyBatisTest#testSelectByBean

```

1.     @Test
2.     public void testSelectByBean() throws IOException {
3.         SqlSession session = sqlSessionFactory.openSession();
4.         try {
5.             BlogMapper mapper = session.getMapper(BlogMapper.class);
6.             Blog queryBean = new Blog();
7.             queryBean.setName("MySQL从入门到改行");
8.             List<Blog> blog = mapper.selectBlogByBean(queryBean);
9.             System.out.println("查询结果: " + blog);
10.        } finally {
11.            session.close();
12.        }
13.    }

```

```

1.     <select id="selectBlogByBean" parameterType="blog" resultType="blog">
2.         SELECT bid, name, author_id authorId FROM blog WHERE name = '${name}'
3.     </select>

```

```

Opening JDBC Connection
Created connection 315860201.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@12d3a4e9]
==> Preparing: SELECT bid, name, author_id authorId FROM blog WHERE name = 'MySQL从入门到改行'
==> Parameters:
<==      Total: 0
查询结果: []
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@12d3a4e9]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@12d3a4e9]
Returned connection 315860201 to pool.

```

和 **\$** 的区别：

1、是否能防止SQL注入：**\$** 方式不会对符号转义，不能防止SQL注入

2、性能：\$ 方式没有预编译，不会缓存

结论：

1、能用#的地方都用#

2、常量的替换，比如排序条件中的字段名称，不用加单引号，可以使用\$

1.4.5. XML中怎么使用特殊符号，比如小于&

1、转义<< (大于可以直接写)

2、使用<![CDATA[]]> ——当XML遇到这种格式就会把[]里面的内容原样输出，不进行解析

1.4.6. 如何实现模糊查询LIKE

1、字符串拼接

在Java代码中拼接%%，直接LIKE。因为没有预编译，存在SQL注入的风险，不推荐使用。

2、CONCAT (推荐)

```
1. <when test="name != null and name != ''">
2.     AND name LIKE CONCAT(CONCAT('%', #{name, jdbcType=VARCHAR}), '%')
3. </when>
```

3、bind标签

```
1. <select id="getEmplList_bind" resultType="empResultMap"
2.     parameterType="Employee">
3.     <bind name="pattern1" value="'%' + empName + '%'" />
4.     <bind name="pattern2" value="'%' + email + '%'" />
5.     SELECT * FROM tbl_emp
6.     <where>
7.         <if test="empId != null">
8.             emp_id = #{empId,jdbcType=INTEGER},
9.         </if>
10.        <if test="empName != null and empName != ''">
11.            AND emp_name LIKE #{pattern1}
12.        </if>
13.        <if test="email != null and email != ''">
14.            AND email LIKE #{pattern2}
15.        </if>
16.    </where>
17.    ORDER BY emp_id
18. </select>
```

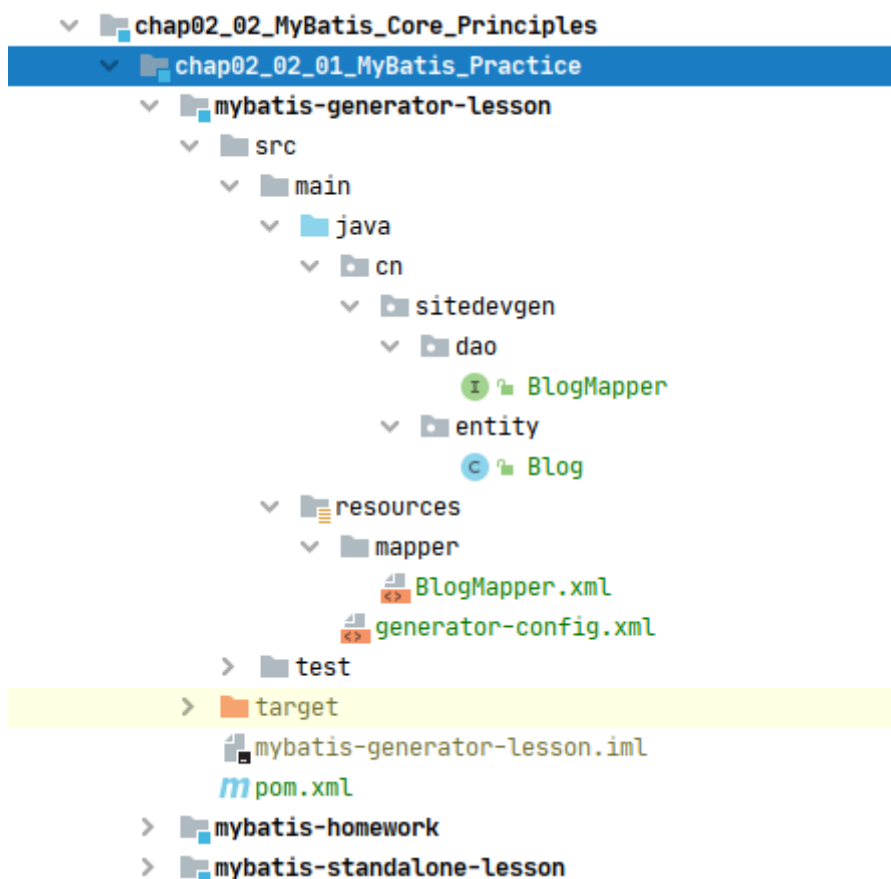
2. 补充

2.1. Mybatis-generator使用

2.1.0. 准备测试SQL

```
1. CREATE TABLE `blog` (  
2.   `bid` int(11) NOT NULL AUTO_INCREMENT,  
3.   `name` varchar(255) DEFAULT NULL,  
4.   `author_id` int(11) DEFAULT NULL,  
5.   PRIMARY KEY (`bid`)  
6. ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;  
7.  
8. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (1, 'RabbitMQ延时消息',  
9.   1001);  
10. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (2, 'MyBatis源码分析',  
11.   1008);
```

2.1.1. 新建工程mybatis-generator-lesson



2.1.2. 引入依赖

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0"
3.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.      <parent>
6.          <artifactId>chap02_02_01_MyBatis_Practice</artifactId>
7.          <groupId>cn.sitedev</groupId>
8.          <version>1.0-SNAPSHOT</version>
9.      </parent>
10.     <modelVersion>4.0.0</modelVersion>
11.
12.     <artifactId>mybatis-generator-lesson</artifactId>
13.
14.     <dependencies>
15.         <!--mybatis-->
16.         <dependency>
17.             <groupId>org.mybatis</groupId>
18.             <artifactId>mybatis</artifactId>
19.             <version>3.5.4</version>
20.         </dependency>
21.         <!--mysql-->
22.         <dependency>
23.             <groupId>mysql</groupId>
24.             <artifactId>mysql-connector-java</artifactId>
25.             <version>5.1.21</version>
26.         </dependency>
27.     </dependencies>
28.
29.     <build>
30.         <plugins>
31.             <!-- mybatis代码生成插件 -->
32.             <plugin>
33.                 <groupId>org.mybatis.generator</groupId>
34.                 <artifactId>mybatis-generator-maven-plugin</artifactId>
35.                 <version>1.3.5</version>
36.                 <configuration>
37.                     <!--配置文件的位置-->
38.                     <configurationFile>src/main/resources/generator-
config.xml</configurationFile>
39.                     <verbose>true</verbose>
40.                     <overwrite>true</overwrite>
41.                 </configuration>
42.                 <executions>
43.                     <execution>
44.                         <id>Generate MyBatis Artifacts</id>
45.                         <goals>
46.                             <goal>generate</goal>
47.                         </goals>
```

```
48.         </execution>
49.     </executions>
50.     <dependencies>
51.         <!--MBG-->
52.         <dependency>
53.             <groupId>org.mybatis.generator</groupId>
54.             <artifactId>mybatis-generator-core</artifactId>
55.             <version>1.3.5</version>
56.         </dependency>
57.     </dependencies>
58. </plugin>
59. </plugins>
60. </build>
61.
62. </project>
```

2.1.3. 创建配置文件generator-config.xml

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE generatorConfiguration
3.     PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
4.     "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">
5.
6. <generatorConfiguration>
7.     <!--mysql 连接数据库jar 这里选择自己本地位置；也可以手动下载一个jar放在指定位置，进行引用。 -->
8.     <classPathEntry location="D:\DevSoftWare\maven\repository\mysql\mysql-connector-java\5.1.21\mysql-connector-java-5.1.21.jar"/>
9.
10.    <context id="DB2Tables" targetRuntime="MyBatis3">
11.        <commentGenerator>
12.            <property name="suppressDate" value="true"/>
13.            <!-- 是否去除自动生成的注释,true: 是,false:否 -->
14.            <property name="suppressAllComments" value="true"/>
15.        </commentGenerator>
16.
17.        <!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->
18.        <jdbcConnection driverClass="com.mysql.jdbc.Driver"
19.            connectionURL="jdbc:mysql://localhost:3306/sitedev-mybatis"
20.            userId="root"
21.            password="root">
22.        </jdbcConnection>
23.
24.        <!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL 和
25.            NUMERIC 类型解析为java.math.BigDecimal -->
26.        <javaTypeResolver>
27.            <property name="forceBigDecimals" value="false"/>
28.        </javaTypeResolver>
29.
30.        <!-- 指定javaBean生成的位置
31.            targetPackage: 生成的类要放的包，真实的包受enableSubPackages属性控制；
32.            targetProject: 目标项目，指定一个存在的目录下，生成的内容会放到指定目录中，如果目录不存在，MBG不会自动建目录
33.        -->
34.        <javaModelGenerator targetPackage="cn.sitedevgen.entity"
35.            targetProject="src/main/java">
36.            <!-- 在targetPackage的基础上，根据数据库的schema再生成一层package，最终生成的类放在这个package下，默认为false；如果多个数据库改为true分目录 -->
37.            <property name="enableSubPackages" value="true"/>
38.            <!-- 设置是否在getter方法中，对String类型字段调用trim()方法 -->
39.            <property name="trimStrings" value="true"/>
40.        </javaModelGenerator>
41.
42.        <!-- 指定mapper映射文件生成的位置
43.            targetPackage、targetProject同javaModelGenerator中作用一样-->
```

```

42.     <sqlMapGenerator targetPackage="mapper"
targetProject="src/main/resources">
43.         <property name="enableSubPackages" value="true"/>
44.     </sqlMapGenerator>
45.
46.     <!-- 指定mapper接口生成的位置
47.         targetPackage、targetProject同javaModelGenerator中作用一样
48.     -->
49.     <javaClientGenerator type="XMLMAPPER" targetPackage="cn.sitedevgen.dao"
50.         targetProject="src/main/java">
51.         <property name="enableSubPackages" value="false"/>
52.     </javaClientGenerator>
53.
54.     <!-- 指定数据库表
55.         domainObjectName: 生成的domain类的名字,当表名和domain类的名字有差异时一定要设置,
如果不设置,直接使用表名作为domain类的名字;
56.         可以设置为somepck.domainName,那么会自动把domainName类再放到somepck包里
面;
57.     -->
58.     <!-- 要生成的表 tableName是数据库中的表名或视图名 domainObjectName是实体类
名,不需要生成Example的时候,设置成false-->
59.     <table tableName="blog" domainObjectName="Blog"
enableCountByExample="false" enableUpdateByExample="false"
60.         enableDeleteByExample="false" enableSelectByExample="false"
selectByExampleQueryId="false"/>
61. </context>
62. </generatorConfiguration>

```

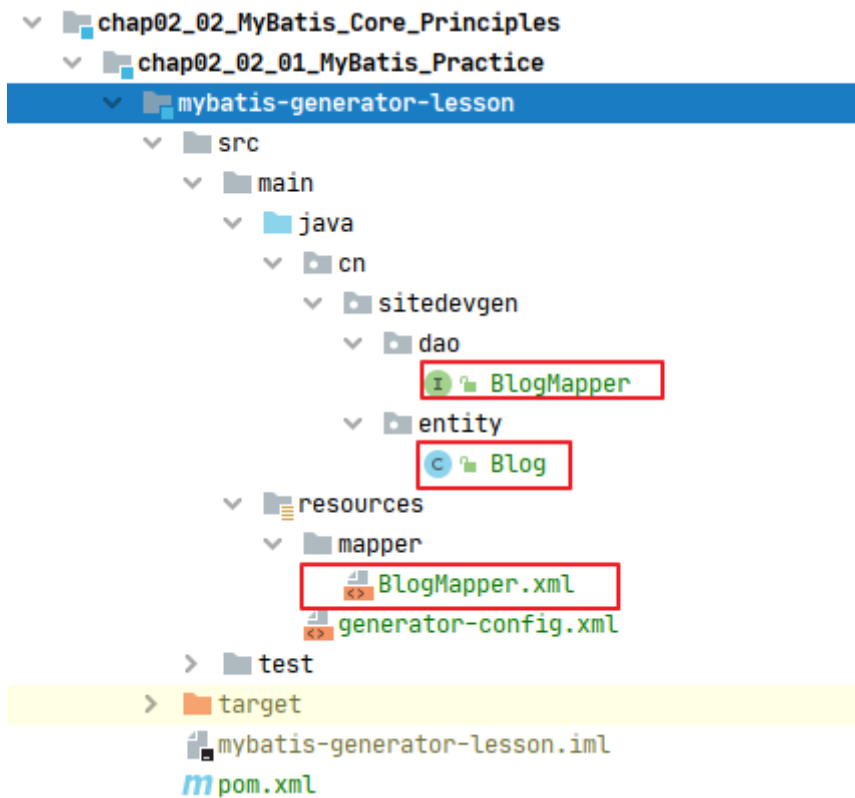
2.1.4. 执行 mybatis-generator:generate

```

v mybatis-generator-lesson
> Lifecycle
v Plugins
> clean (org.apache.maven.plugins:maven-clean-plugin:2.5)
> compiler (org.apache.maven.plugins:maven-compiler-plugin:3.1)
> deploy (org.apache.maven.plugins:maven-deploy-plugin:2.7)
> install (org.apache.maven.plugins:maven-install-plugin:2.4)
> jar (org.apache.maven.plugins:maven-jar-plugin:2.4)
v mvn:mybatis-generator:generate
  mvn:mybatis-generator:help
> resources (org.apache.maven.plugins:maven-resources-plugin:2.6)
> site (org.apache.maven.plugins:maven-site-plugin:3.3)
> surefire (org.apache.maven.plugins:maven-surefire-plugin:2.12.4)
> Dependencies

```

2.1.5. 查看生成的文件



BlogMapper.java

```
1. package cn.sitedevgen.dao;  
2.  
3. import cn.sitedevgen.entity.Blog;  
4.  
5. public interface BlogMapper {  
6.     int deleteByPrimaryKey(Integer bid);  
7.  
8.     int insert(Blog record);  
9.  
10.    int insertSelective(Blog record);  
11.  
12.    Blog selectByPrimaryKey(Integer bid);  
13.  
14.    int updateByPrimaryKeySelective(Blog record);  
15.  
16.    int updateByPrimaryKey(Blog record);  
17. }
```

Blog.java

```
1. package cn.sitedevgen.entity;
2.
3. public class Blog {
4.     private Integer bid;
5.
6.     private String name;
7.
8.     private Integer authorId;
9.
10.    public Integer getBid() {
11.        return bid;
12.    }
13.
14.    public void setBid(Integer bid) {
15.        this.bid = bid;
16.    }
17.
18.    public String getName() {
19.        return name;
20.    }
21.
22.    public void setName(String name) {
23.        this.name = name == null ? null : name.trim();
24.    }
25.
26.    public Integer getAuthorId() {
27.        return authorId;
28.    }
29.
30.    public void setAuthorId(Integer authorId) {
31.        this.authorId = authorId;
32.    }
33. }
```

BlogMapper.xml

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3.  <mapper namespace="cn.sitedevgen.dao.BlogMapper">
4.      <resultMap id="BaseResultMap" type="cn.sitedevgen.entity.Blog">
5.          <id column="bid" jdbcType="INTEGER" property="bid" />
6.          <result column="name" jdbcType="VARCHAR" property="name" />
7.          <result column="author_id" jdbcType="INTEGER" property="authorId" />
8.      </resultMap>
9.      <sql id="Base_Column_List">
10.         bid, name, author_id
11.      </sql>
12.      <select id="selectByPrimaryKey" parameterType="java.lang.Integer"
    resultMap="BaseResultMap">
13.         select
14.         <include refid="Base_Column_List" />
15.         from blog
16.         where bid = #{bid,jdbcType=INTEGER}
17.      </select>
18.      <delete id="deleteByPrimaryKey" parameterType="java.lang.Integer">
19.         delete from blog
20.         where bid = #{bid,jdbcType=INTEGER}
21.      </delete>
22.      <insert id="insert" parameterType="cn.sitedevgen.entity.Blog">
23.         insert into blog (bid, name, author_id
24.         )
25.         values (#{bid,jdbcType=INTEGER}, #{name,jdbcType=VARCHAR}, #
    {authorId,jdbcType=INTEGER}
26.         )
27.      </insert>
28.      <insert id="insertSelective" parameterType="cn.sitedevgen.entity.Blog">
29.         insert into blog
30.         <trim prefix="(" suffix=")" suffixOverrides=",">
31.             <if test="bid != null">
32.                 bid,
33.             </if>
34.             <if test="name != null">
35.                 name,
36.             </if>
37.             <if test="authorId != null">
38.                 author_id,
39.             </if>
40.         </trim>
41.         <trim prefix="values (" suffix=")" suffixOverrides=",">
42.             <if test="bid != null">
43.                 #{bid,jdbcType=INTEGER},
44.             </if>
45.             <if test="name != null">
46.                 #{name,jdbcType=VARCHAR},
```



```

47.         </if>
48.         <if test="authorId != null">
49.             #{authorId,jdbcType=INTEGER},
50.         </if>
51.     </trim>
52. </insert>
53. <update id="updateByPrimaryKeySelective"
parameterType="cn.sitedevgen.entity.Blog">
54.     update blog
55.     <set>
56.         <if test="name != null">
57.             name = #{name,jdbcType=VARCHAR},
58.         </if>
59.         <if test="authorId != null">
60.             author_id = #{authorId,jdbcType=INTEGER},
61.         </if>
62.     </set>
63.     where bid = #{bid,jdbcType=INTEGER}
64. </update>
65. <update id="updateByPrimaryKey" parameterType="cn.sitedevgen.entity.Blog">
66.     update blog
67.     set name = #{name,jdbcType=VARCHAR},
68.         author_id = #{authorId,jdbcType=INTEGER}
69.     where bid = #{bid,jdbcType=INTEGER}
70. </update>
71. </mapper>

```

2.1.6. 测试

创建数据库配置文件db.properties

1. jdbc.driver=com.mysql.jdbc.Driver
2. jdbc.url=jdbc:mysql://localhost:3306/sitedev-mybatis?
useUnicode=true&characterEncoding=utf-8&rewriteBatchedStatements=true
3. jdbc.username=root
4. jdbc.password=root

创建mybatis配置文件mybatis-config.xml

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-config.dtd">
3. <configuration>
4.
5.     <properties resource="db.properties"></properties>
6.     <settings>
7.         <!-- 打印查询语句 -->
8.         <setting name="logImpl" value="STDOUT_LOGGING"/>
9.     </settings>
10.
11.     <environments default="development">
12.         <environment id="development">
13.             <transactionManager type="JDBC"/><!-- 单独使用时配置成MANAGED没有事务
-->
14.             <dataSource type="POOLED">
15.                 <property name="driver" value="${jdbc.driver}"/>
16.                 <property name="url" value="${jdbc.url}"/>
17.                 <property name="username" value="${jdbc.username}"/>
18.                 <property name="password" value="${jdbc.password}"/>
19.             </dataSource>
20.         </environment>
21.     </environments>
22.
23.     <mappers>
24.         <mapper resource="BlogMapper.xml"/>
25.     </mappers>
26.
27. </configuration>
```

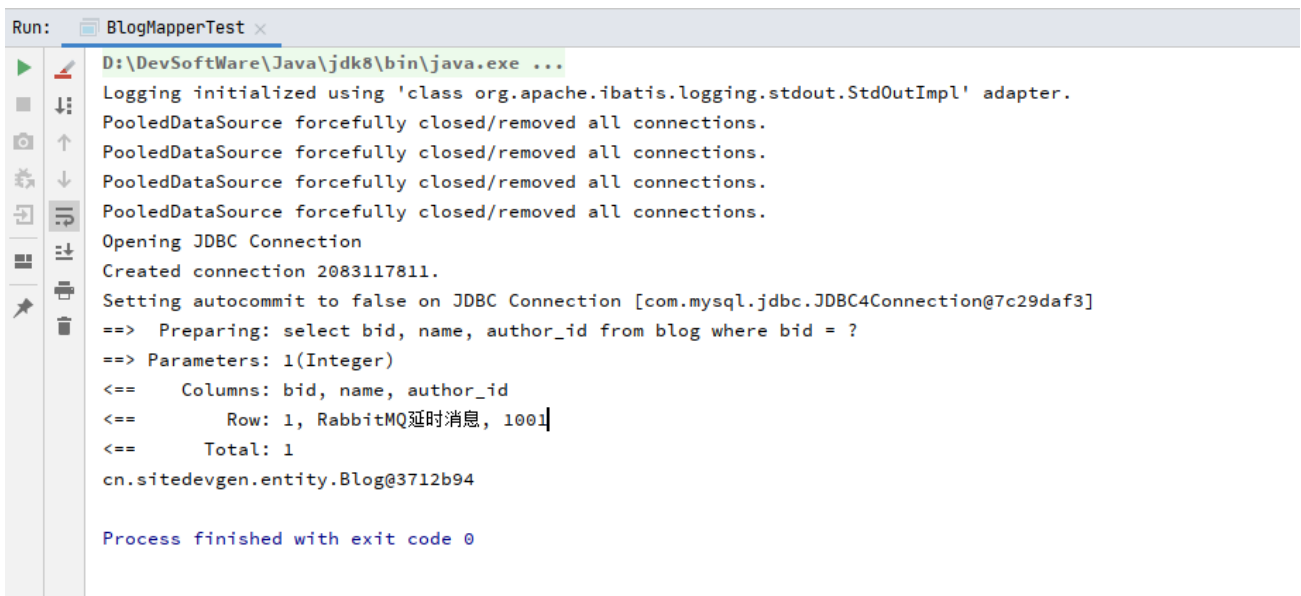
创建测试类BlogMapperTest.java

```

1. package cn.sitedevgen;
2.
3. import cn.sitedevgen.dao.BlogMapper;
4. import cn.sitedevgen.entity.Blog;
5. import org.apache.ibatis.session.SqlSession;
6. import org.apache.ibatis.session.SqlSessionFactory;
7. import org.apache.ibatis.session.SqlSessionFactoryBuilder;
8.
9. import java.io.InputStream;
10.
11. public class BlogMapperTest {
12.     public static void main(String[] args) {
13.         InputStream inputStream =
14.             BlogMapperTest.class.getClassLoader().getResourceAsStream("mybatis-config.xml");
15.         SqlSessionFactory factory = new
16.             SqlSessionFactoryBuilder().build(inputStream);
17.         SqlSession sqlSession = factory.openSession();
18.         BlogMapper blogMapper = sqlSession.getMapper(BlogMapper.class);
19.         Blog blog = blogMapper.selectByPrimaryKey(1);
20.         System.out.println(blog);
    }
}

```

执行测试类, 控制台查看执行结果



```

Run: BlogMapperTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 2083117811.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@7c29daf3]
==> Preparing: select bid, name, author_id from blog where bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id
<== Row: 1, RabbitMQ延时消息, 1001
<== Total: 1
cn.sitedevgen.entity.Blog@3712b94

Process finished with exit code 0

```

注意事项:如果重新生成, 最好删除原来的Mapper.xml文件, 防止内容追加导致的报错。

最终的项目结构为

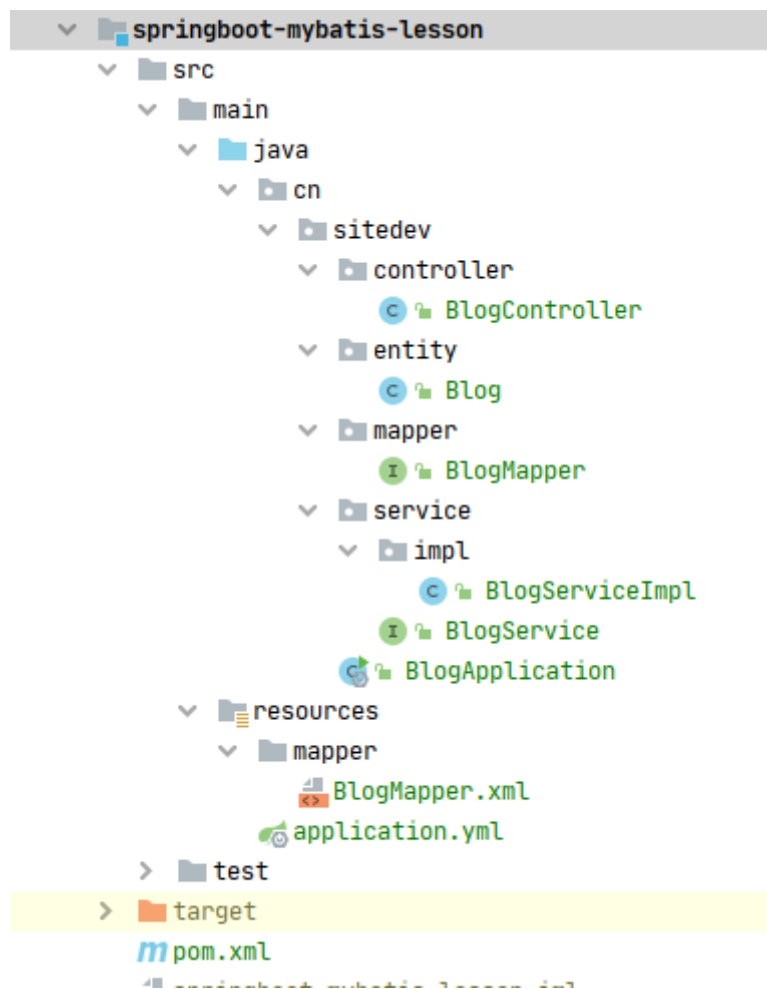


2.2. SpringBoot整合Mybatis

2.2.0. 准备测试SQL

```
1. CREATE TABLE `blog` (  
2.   `bid` int(11) NOT NULL AUTO_INCREMENT,  
3.   `name` varchar(255) DEFAULT NULL,  
4.   `author_id` int(11) DEFAULT NULL,  
5.   PRIMARY KEY (`bid`)  
6. ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;  
7.  
8. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (1, 'RabbitMQ延时消息',  
9.   1001);  
9. INSERT INTO `blog` (`bid`, `name`, `author_id`) VALUES (2, 'MyBatis源码分析',  
   1008);
```

2.2.1. 新建工程springboot-mybatis-lesson



2.2.2. 引入依赖

```

1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <project xmlns="http://maven.apache.org/POM/4.0.0"
3.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.      <parent>
6.          <artifactId>chap02_02_01_MyBatis_Practice</artifactId>
7.          <groupId>cn.sitedev</groupId>
8.          <version>1.0-SNAPSHOT</version>
9.      </parent>
10.     <modelVersion>4.0.0</modelVersion>
11.
12.     <artifactId>springboot-mybatis-lesson</artifactId>
13.
14.     <dependencies>
15.         <dependency>
16.             <groupId>org.springframework.boot</groupId>
17.             <artifactId>spring-boot-starter-web</artifactId>
18.         </dependency>
19.         <dependency>
20.             <groupId>org.mybatis.spring.boot</groupId>
21.             <artifactId>mybatis-spring-boot-starter</artifactId>
22.             <version>2.1.1</version>
23.         </dependency>
24.         <dependency>
25.             <groupId>mysql</groupId>
26.             <artifactId>mysql-connector-java</artifactId>
27.             <version>5.1.21</version>
28.         </dependency>
29.     </dependencies>
30.
31.     <dependencyManagement>
32.         <dependencies>
33.             <dependency>
34.                 <groupId>org.springframework.boot</groupId>
35.                 <artifactId>spring-boot-dependencies</artifactId>
36.                 <version>2.2.6.RELEASE</version>
37.                 <scope>import</scope>
38.                 <type>pom</type>
39.             </dependency>
40.         </dependencies>
41.
42.     </dependencyManagement>
43. </project>

```

2.2.3. 创建entity

```
1. package cn.sitedev.entity;
2.
3. import lombok.AllArgsConstructor;
4. import lombok.Data;
5. import lombok.NoArgsConstructor;
6.
7. @Data
8. @AllArgsConstructor
9. @NoArgsConstructor
10. public class Blog {
11.     private Integer bid; // 文章id
12.     private String name; // 文章标题
13.     private String authorId; // 文章作者id
14. }
```

2.2.4. 创建Mapper接口

```
1. package cn.sitedev.mapper;
2.
3. import cn.sitedev.entity.Blog;
4.
5. public interface BlogMapper {
6.     Blog selectByPrimaryKey(Integer bid);
7.
8.     int insert(Blog blog);
9. }
```

2.2.5. 创建Service

```

1. package cn.sitedev.service;
2.
3. import cn.sitedev.entity.Blog;
4.
5. public interface BlogService {
6.     Blog getBlogById(int bid);
7.
8.     int addBlog(Blog blog);
9. }
10. //////////////////////////////////////
11. package cn.sitedev.service.impl;
12.
13. import cn.sitedev.entity.Blog;
14. import cn.sitedev.mapper.BlogMapper;
15. import cn.sitedev.service.BlogService;
16. import org.springframework.beans.factory.annotation.Autowired;
17. import org.springframework.stereotype.Service;
18.
19. @Service
20. public class BlogServiceImpl implements BlogService {
21.     @Autowired
22.     private BlogMapper blogMapper;
23.
24.     @Override
25.     public Blog getBlogById(int bid) {
26.         return this.blogMapper.selectByPrimaryKey(bid);
27.     }
28.
29.     @Override
30.     public int addBlog(Blog blog) {
31.         return this.blogMapper.insert(blog);
32.     }
33. }

```

2.2.6. 创建Controller


```
1. package cn.sitedev.controller;
2.
3. import cn.sitedev.entity.Blog;
4. import cn.sitedev.service.BlogService;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.web.bind.annotation.GetMapping;
7. import org.springframework.web.bind.annotation.RequestMapping;
8. import org.springframework.web.bind.annotation.RestController;
9.
10. @RestController
11. @RequestMapping("/blog")
12. public class BlogController {
13.     @Autowired
14.     private BlogService blogService;
15.
16.     @GetMapping("/list")
17.     public Blog getBlog(Integer bid) {
18.         return this.blogService.getBlogById(bid);
19.     }
20. }
```

2.2.7. 创建主配置文件application.yml

```
1. # 默认使用配置
2. spring:
3.     profiles:
4.         active: dev
5. # 公共配置与profiles选择无关
6. mybatis:
7.     type-aliases-package: cn.sitedev.entity
8.     mapper-locations: classpath:mapper/*.xml
9.     configuration:
10.         log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
11. ---
12.
13. # 开发配置
14. spring:
15.     profiles: dev
16.
17.     datasource:
18.         url: jdbc:mysql://localhost:3306/sitedev-mybatis?
characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStatements=true
19.         username: root
20.         password: root
21.         driver-class-name: com.mysql.jdbc.Driver
```

2.2.8. 创建Mapper映射文件

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
   "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
3. <mapper namespace="cn.sitedev.mapper.BlogMapper">
4.
5.     <select id="selectByPrimaryKey" parameterType="Integer" resultMap="blogMap">
6.         SELECT * FROM blog WHERE bid = #{bid}
7.     </select>
8.
9.     <insert id="insert" parameterType="Blog">
10.         INSERT INTO blog(name, author_id) VALUES (#{name}, #{authorId})
11.     </insert>
12.
13.     <resultMap id="blogMap" type="Blog">
14.         <id column="bid" property="bid"/>
15.         <result column="name" property="name"/>
16.         <result column="author_id" property="authorId"/>
17.     </resultMap>
18. </mapper>
```

2.2.9. 创建启动类

```
1. package cn.sitedev;
2.
3. import org.mybatis.spring.annotation.MapperScan;
4. import org.springframework.boot.SpringApplication;
5. import org.springframework.boot.autoconfigure.SpringBootApplication;
6.
7. @SpringBootApplication
8. @MapperScan("cn.sitedev.mapper")
9. public class BlogApplication {
10.     public static void main(String[] args) {
11.         SpringApplication.run(BlogApplication.class, args);
12.     }
13. }
```

2.2.10. 启动应用, 访问接口<http://localhost:8080/blog/list?bid=1>

```
2020-06-04 23:41:26.340 INFO 13360 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2020-06-04 23:41:26.132 INFO 13360 --- [nio-8080-exec-1] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
JDBC Connection [HikariProxyConnection@1980622155 wrapping com.mysql.jdbc.JDBC4Connection@51a9efa7] will not be managed by Spring
==> Preparing: SELECT * FROM blog WHERE bid = ?
==> Parameters: 1(Integer)
<== Columns: bid, name, author_id
<== Row: 1, RabbitMQ延时消息, 1001
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@5fe004ce]
```

