第2章 映射文件详解

引入

通过前面的学习，我们对MyBatis有了初步认识，MyBatis的三个基本要素：

（1）MyBatis核心的接口和类

（2）MyBatis核心配置文件（mybatis-config.xml）

（3）SQL映射文件（mapper.xml）

MyBatis的核心接口和类、配置文件在前面已经讲解了，那么本章再来详细讲解映射文件。

映射文件中需要掌握的部分就是：常用的元素（标签）、常用元素的属性以及SQL语句。

在映射文件中，可以编写以下的顶级元素标签：

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

在每个顶级元素标签中可以添加很多个属性，具体如下：

属性	描述
`id`	在命名空间中唯一的标识符，可以被用来引用这条语句。
`parameterType`	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的，因为 MyBatis 可以通过类型处理器（TypeHandler）推断出具体传入语句的参数，默认值为未设置（unset）。
`parameterMap`	用于引用外部 parameterMap 的属性，目前已被废弃。请使用行内参数映射和 parameterType 属性。
`flushCache`	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：（对 insert、update 和 delete 语句）true。
`timeout`	这个设置是在抛出异常之前，驱动程序等待数据库返回请求结果的秒数。默认值为未设置（unset）（依赖数据库驱动）。
`statementType`	可选 STATEMENT，PREPARED 或 CALLABLE。这会让 MyBatis 分别使用 Statement，PreparedStatement 或 CallableStatement，默认值：PREPARED。
`useGeneratedKeys`	（仅适用于 insert 和 update）这会令 MyBatis 使用 JDBC 的 getGeneratedKeys 方法来取出由数据库内部生成的主键（比如：像 MySQL 和 SQL Server 这样的关系型数据库管理系统的自动递增字段），默认值：false。
`keyProperty`	（仅适用于 insert 和 update）指定能够唯一识别对象的属性，MyBatis 会使用 getGeneratedKeys 的返回值或 insert 语句的 selectKey 子元素设置它的值，默认值：未设置（`unset`）。如果生成列不止一个，可以用逗号分隔多个属性名称。
`keyColumn`	（仅适用于 insert 和 update）设置生成键值在表中的列名，在某些数据库（像 PostgreSQL）中，当主键列不是表中的第一列的时候，是必须设置的。如果生成列不止一个，可以用逗号分隔多个属性名称。
`databaseId`	如果配置了数据库厂商标识（databaseIdProvider），MyBatis 会加载所有不带 databaseId 或匹配当前 databaseId 的语句；如果带和不带的语句都有，则不带的会被忽略

以上标签属性不需要死记硬背，因为后面我们会慢慢使用它们，所以现在只需要有一个印象即可。

预准备

我们在开始讲解之前，先在Navicat中导入数据库文件：myemployees，搭建好数据库。

然后再搭建好我们的mybatis环境：

步骤一：创建maven项目，pom.xml文件的代码如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
```

```
    <artifactId>MyBatis_02_Mapper映射文件详解</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>

        <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.6</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.16</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/log4j/log4j -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>

    </dependencies>

</project>
```

步骤二：创建log4j.properties，配置log4j用于记录mybatis的日志，方便后面查看BUG、缓存等，十分方便。

```
log4j.rootLogger=DEBUG,Console

#Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Target=System.out
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n

log4j.logger.org.apache=ERROR
log4j.logger.org.mybatis=ERROR
log4j.logger.org.springframework=ERROR
#这个需要
log4j.logger.log4jdbc.debug=ERROR
log4j.logger.com.gk.mapper=ERROR

log4j.logger.jdbc.audit=ERROR
log4j.logger.jdbc.resultset=ERROR
#这个打印SQL语句非常重要
log4j.logger.jdbc.sqlonly=DEBUG
```

```
log4j.logger.jdbc.sqltiming=ERROR
log4j.logger.jdbc.connection=FATAL
```

步骤三：创建db.properties文件，存放数据库信息

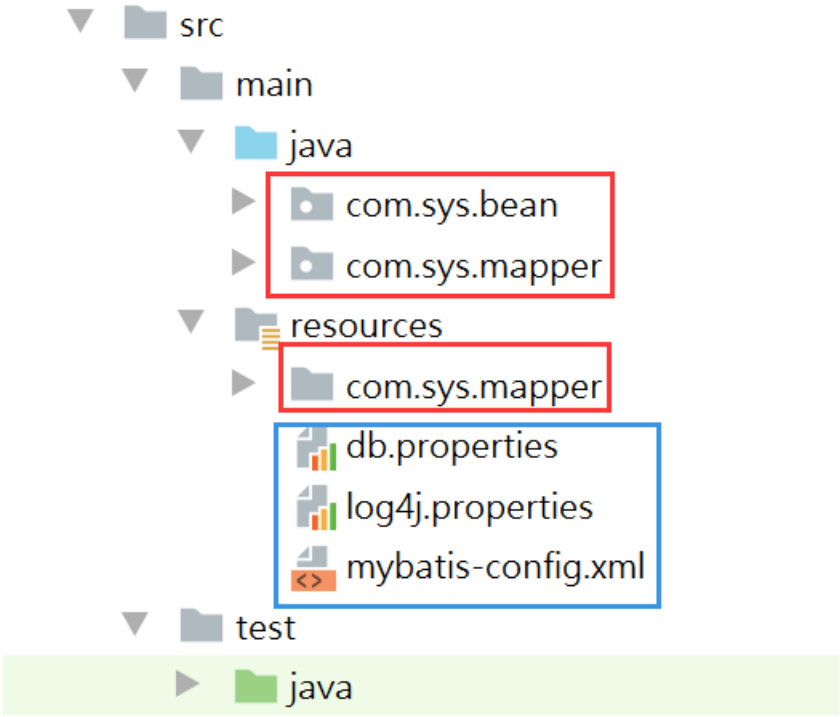
```
jdbc.driver=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/myemployees?serverTimezone=GMT
jdbc.username=root
jdbc.password=root
```

步骤四：编写配置文件

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

    <!-- 引入外部db.properties -->
    <properties resource="db.properties"></properties>

    <!-- settings用来控制mybatis运行时的行为，是mybatis中的重要配置 -->
    <settings>
        <!-- 设置MyBatis支持Log4j -->
        <setting name="logImpl" value="LOG4J"/>
    </settings>

    <!-- 配置指定包路径下，所有类的别名 -->
    <typeAliases>
        <package name="com.sys.bean"/>
    </typeAliases>

    <!-- 环境配置 -->
    <environments default="development">
        <!-- id：表示不同环境的名称 -->
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!-- 使用${}来引入外部变量 -->
                <property name="driver" value="${jdbc.driver}"/>
                <property name="url" value="${jdbc.url}"/>
                <property name="username" value="${jdbc.username}"/>
                <property name="password" value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>

    <!-- 指定sql所适用的数据库 -->
    <databaseIdProvider type="DB_VENDOR">
        <property name="MySQL" value="mysql"/>
        <property name="Oracle" value="oracle"/>
    </databaseIdProvider>

    <!-- 批量注册mapper映射文件，使用注解编写SQL语句 -->
    <mappers>
        <package name="com.sys.mapper"/>
```

```
    </mappers>
</configuration>
```

步骤五：按照配置文件的 `<typeAliases>` 和 `<mappers>` 创建好对应的包



一、SELECT之 关于参数传递的问题

在前面的例子当中，MyBatis会将 Mapper接口方法中的参数值 传递给 mapper映射文件中sql语句，然后会进行sql语句的拼接，获得一个完整的sql语句：

```
EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
Employee emp = mapper.findEmpById(101);
System.out.println( emp );
sqlSession.close();
```

参数传递

```
<select id="findEmpById" parameterType="int" resultType="Employee">
    SELECT * FROM employees WHERE employee_id = #{id}
</select>
```

在参数传递的时候，我们分为以下几种情况讨论下：

1、当接口方法的参数有且仅有一个时

（1）若为基本数据类型、字符串，sql语句中参数：#{参数名}

mapper接口方法：

```java
/**
 * 根据id查询员工的信息
 * @param id
 * @return
 */
Employee findEmpById( int id );
```

mapper映射文件：

```xml
<select id="findEmpById" parameterType="int" resultType="Employee">
    SELECT * FROM employees WHERE employee_id = #{id}
</select>
```

在#{}中，填写形参的名字即可。

测试代码：

```java
@Test
public void testFindEmpById(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    // 这里的实参101就会传递到sql语句中进行拼接
    Employee emp = mapper.findEmpById(101);
    System.out.println( emp );
    sqlSession.close();
}
```

（2）若为引用数据类型，sql语句中参数：#{该类的属性名}

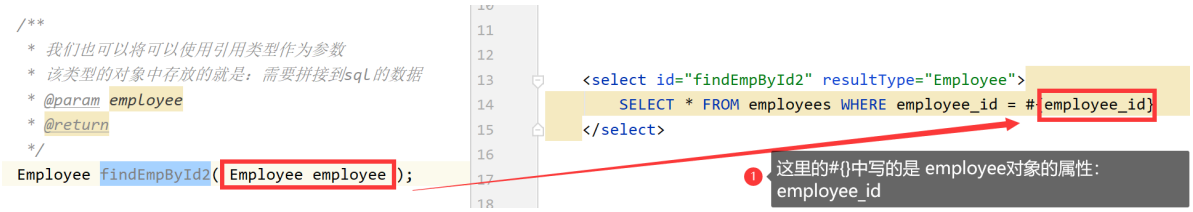
我们还可以将 **需要拼接到sql语句中的数据** 封装进我们的对象当中。这种方式也可以解决多个参数传递的问题，当然主流方式在后面的第二种情况中。

mapper接口方法：

```java
/**
 * 根据id查询指定的员工
 *
 * 我们也可以将可以使用引用类型作为参数
 * 该类型的对象中存放的就是：需要拼接到sql的数据
 * @param employee
 * @return
 */
Employee findEmpById2( Employee employee );
```

mapper映射文件：

```xml
<select id="findEmpById2" resultType="Employee">
    SELECT * FROM employees WHERE employee_id = #{employee_id}
</select>
```



在#{}中，填写指定类型对象中的属性名即可。

测试代码：

```java
@Test
public void testFindEmpById2(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    // 创建一个Employee对象
    Employee employee = new Employee();
    // 将需要拼接到sql语句中的数据存放到对象当中
    employee.setEmployee_id(101);
    // 将该对象作为实参传递
    Employee emp = mapper.indEmpById2( employee );
    System.out.println( emp );
    sqlSession.close();
}
```

2、当接口方法的参数有多个时

在这种情况下，就不能按照第一种情况的第(1)部分写了，当然 第(2)部分提到的：**将 需要拼接到sql语句中的数据 封装进对象**是可以解决这种情况的，但是这种方法适用面不广泛。现在主流的方式：使用@Param注解。

现在，我们先来看看MyBatis是如何获取参数值的：

> MyBatis在处理接口方法的多个参数的时候，会将这些参数封装到一个map中，此时map中的key就是arg0、arg1、param1、param2这些值，但是很明显，这样的传值方式不是很友好，我们也不知道arg0、arg1等key所对应的值是什么，很不方便。所以，我们可以采用 @Param注解来设置 map中的key值，这样就很方便了！！！
>
> 若不使用@Param注解，则会抛出如下异常：
>
> Cause: org.apache.ibatis.binding.BindingException:Parameter 'fName' not found. Available parameters are [arg1, arg0, param1, param2]

我们现在来看看这个注解怎么使用：

mapper接口方法：

```
    /**
     * 根据姓、名查询员工的信息
     * @param fName：名
     * @param lName：姓
     * @return
     */
    Employee findEmpByNames(@Param("fN") String fName, @Param("lN")String lName
);
```

mapper映射文件：

```
    <select id="findEmpByNames" resultType="Employee">
        SELECT * FROM employees WHERE first_name = #{fN} AND last_name = #{lN};
    </select>
```

```
/**
 * 根据姓、名查询员工的信息
 * @param fName：名
 * @param lName：姓
 * @return
 * 注意使用 @Param(key)设置 MyBatis自带的map的key值
 */
Employee findEmpByNames(@Param("fN") String fName, @Param("lN")String lName );

/**
```

```
<select id="findEmpByNames" resultType="Employee">
    SELECT * FROM employees WHERE first_name = #{fN} AND last_name = #{lN};
</select>
```

1 此时，#{}中填写的是@Param注解设置的key值

测试代码：

```
    @Test
    public void testFindEmpByNames(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        Employee emp = mapper.findEmpByNames("Steven", "K_ing");
        System.out.println( emp );
        sqlSession.close();
    }
```

3、使用map来传递参数

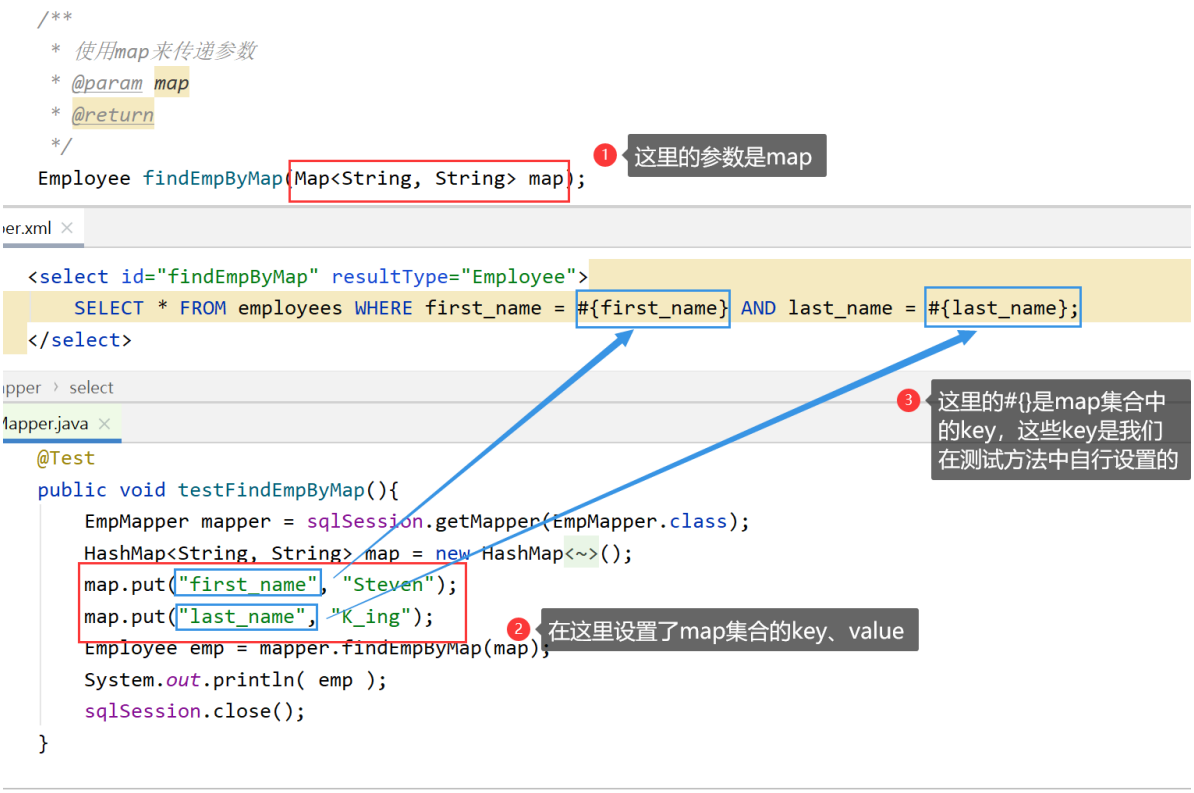
既然MyBatis使用自带的map集合来存放参数值，那么我们也可以将参数存放在map集合中，此时依然是直接使用#{key}来获取具体的参数值。

mapper接口方法：

```java
    /**
     * 使用map来传递参数
     * @param map
     * @return
     */
    Employee findEmpByMap(Map<String, String> map);
```

mapper映射文件：

```xml
    <select id="findEmpByMap" resultType="Employee">
        SELECT * FROM employees WHERE first_name = #{first_name} AND last_name =
#{last_name};
    </select>
```



测试方法：

```java
    @Test
    public void testFindEmpByMap(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        HashMap<String, String> map = new HashMap<String, String>();
        map.put("first_name", "Steven");
        map.put("last_name", "K_ing");
        Employee emp = mapper.findEmpByMap(map);
        System.out.println( emp );
        sqlSession.close();
    }
```

二、SELECT之 获取参数拼接到SQL语句的问题

当使用 `#{}` 来获取值的时候会发现打印的sql语句如下：

```
select  from emp where empno=? and ename=?
```

当使用 `${}` 来获取值的时候会发现打印的sql语句如下：

```
select  from emp where empno=7369 and ename='SMITH'
```

通过刚刚的案例大家已经发现了存在的问题了：

（1）若使用#{}方式进行取值：采用的是参数预编译的方式，参数的位置使用？进行替代，不会出现sql注入的问题。

（2）若使用${}方式进行取值：采用的是直接跟sql语句进行拼接的方式，此时会出现sql注入的问题。

所以，我们现在都是采用 `#{}` ，以防止sql注入。

三、SELECT之 处理集合类型的返回结果

1、List集合的处理

当返回值的结果是List集合时，`resultType` 返回值的类型写的是：**集合中每一个元素的具体数据类型。**

```
/**
 * 获取所有的emp
 * @return：集合对象
 */
List<Employee> findAllEmp();
```

1 每一条数据都会被MyBatis封装成一个Employee对象，当数据库返回多条数据时，也就会有多个Employee对象，此时就需要使用List<Employee>集合来存储这些对象

per.xml

```
<!--
    当返回值的结果是集合的时候，返回值的类型依然写的是集合中具体的类型（map除外）
-->
<select id="findAllEmp" resultType="Employee">
    SELECT * FROM employees
</select>
```

apper › select

Mapper.java

```
@Test
public void testFindAllEmp(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    List<Employee> allEmp = mapper.findAllEmp();
    for (Employee emp: allEmp) {
        System.out.println( emp );
    }
    sqlSession.close();
}
```

mapper接口方法：

```
/**
 * 获取所有的emp
 * @return：集合对象
 */
List<Employee> findAllEmp();
```

mapper映射文件：

```
<!--
    当返回值的结果是集合的时候，返回值的类型依然写的是集合中具体的类型（map除外）
-->
<select id="findAllEmp" resultType="Employee">
    SELECT * FROM employees
</select>
```

测试方法：

```
@Test
public void testFindAllEmp(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    List<Employee> allEmp = mapper.findAllEmp();
    for (Employee emp: allEmp) {
        System.out.println( emp );
    }
    sqlSession.close();
}
```

2、Map集合的处理

由于map集合比较特殊，它含有key和value，所以我们分两种情况说明：

（1）使用map存放一条数据时：

在查询的时候，若mysql只返回一条数据，此时返回值的类型也可以设置为map，当mybatis查询完成之后会把列名作为key，列的值作为value，转换到map中。

mapper接口方法：

```
/**
 *  返回的是map对象
 * @param id
 * @return
 */
Map<String, Object> findEmpByIdReturnMap(int id);
```

mapper映射文件：

```
<!--
    使用map存放一条数据
    在查询的时候，若mysql只返回一条数据，此时返回值的类型也可以设置为map，
    当mybatis查询完成之后会把列名作为key，列的值作为value，转换到map中
-->
<select id="findEmpByIdReturnMap" resultType="map">
    SELECT * FROM employees WHERE employee_id = #{id}
</select>
```

测试方法：

```
@Test
public void testFindEmpByIdReturnMap(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Map<String, Object> map = mapper.findEmpByIdReturnMap(101);
    for ( Map.Entry<String, Object> entry : map.entrySet() ) {
        System.out.println( entry.getKey() +"--->"+ entry.getValue() );
    }
    sqlSession.close();
}
```

运行结果：

```
2021-09-14 21:18:35,332 [main] DEBUG [com.sys.mapper.EmpMapper.findEmpByIdReturnMap] - <==      Total: 1
manager_id--->100
department_id--->90
job_id--->AD_VP
employee_id--->101
last_name--->Kochhar
phone_number--->515.123.4568
salary--->17000.0
first_name--->Neena
hiredate--->1992-04-03 08:00:00.0
email--->NKOCHHAR
```

（2）使用map存放多条数据时：

当使用map存放多条数据时，返回值的类型一定要写map中value的类型，同时在mapper接口的方法上要添加@MapKey的注解，来设置key是什么。

mapper接口方法：

```
/**
 * 使用map存放多条数据
 * @return
 *
 * 注解：@MapKey("employee_id")  ===》设置map集合的key为数据表中的employee_id字段
 */
@MapKey("employee_id")
Map<Integer,Employee> findAllEmpReturnMap();
```

mapper映射文件：

```
    <!--
        使用map存放多条数据
        注意：
        当使用map存放多条数据时，返回值的类型一定要写map中value的类型
        同时在mapper接口的方法上要添加@MapKey的注解，来设置key是什么结果
    -->
    <select id="findAllEmpReturnMap" resultType="Employee">
        SELECT * FROM employees
    </select>
```

测试方法：

```
    @Test
    public void testFindAllEmpReturnMap(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        Map<Integer, Employee> emps = mapper.findAllEmpReturnMap();
        for (Map.Entry<Integer, Employee> entry: emps.entrySet() ) {
            System.out.println( entry.getKey() +"--->"+ entry.getValue() );
        }
        sqlSession.close();
    }
```

运行结果：

```
E:\Java\jdk1.8.0_251\bin\java.exe ...
2021-09-14 21:21:36,859 [main] DEBUG [com.sys.mapper.EmpMapper.findAllEmpReturnMap] - ==>  Preparing: SELECT * FROM employees
2021-09-14 21:21:36,877 [main] DEBUG [com.sys.mapper.EmpMapper.findAllEmpReturnMap] - ==> Parameters:
2021-09-14 21:21:36,906 [main] DEBUG [com.sys.mapper.EmpMapper.findAllEmpReturnMap] - <==      Total: 109
100--->Employee{employee_id=100, first_name='Steven', last_name='K_ing', email='SKING', phone_number='515.123.4567', job_id='AD_PRES', salar
101--->Employee{employee_id=101, first_name='Neena', last_name='Kochhar', email='NKOCHHAR', phone_number='515.123.4568', job_id='AD_VP', sal
102--->Employee{employee_id=102, first_name='Lex', last_name='De Haan', email='LDEHAAN', phone_number='515.123.4569', job_id='AD_VP', salary
103--->Employee{employee_id=103, first_name='Alexander', last_name='Hunold', email='AHUNOLD', phone_number='590.423.4567', job_id='IT_PROG',
104--->Employee{employee_id=104, first_name='Bruce', last_name='Ernst', email='BERNST', phone_number='590.423.4568', job_id='IT_PROG', salar
105--->Employee{employee_id=105, first_name='David', last_name='Austin', email='DAUSTIN', phone_number='590.423.4569', job_id='IT_PROG', sal
106--->Employee{employee_id=106, first_name='Valli', last_name='Pataballa', email='VPATABAL', phone_number='590.423.4560', job_id='IT_PROG',
107--->Employee{employee_id=107, first_name='Diana', last_name='Lorentz', email='DLORENTZ', phone_number='590.423.5567', job_id='IT_PROG', s
```

四、级联查询

1、自定义结果集映射

引入：我们前面在创建一个实体类的时候，我们都是将类中的属性 和 数据表的字段写的一模一样，方便MyBatis可以直接 属性名\字段名 进行结果集映射。但是在现实场景中，总会有实体类的属性名会和字段名不一样的情况，此时直接进行结果集映射是会出现异常的。所以针对这种情况，我们需要使用 `<resultMap>` 来自定义结果集映射。

编写实体类Department

```
package com.sys.bean;

/**
 * @program: MyBatis_02_Mapper映射文件详解
 * @description: 部门实体类
 * @author: DW
 * @create: 2021-09-26 22:07
 */
public class Department {
```

```
    private Integer dept_id;
    private String dept_name;
    private Integer manager_id;
    private Integer location_id;

    public Department() {
    }

    public Department(Integer dept_id, String dept_name, Integer manager_id,
Integer location_id) {
        this.dept_id = dept_id;
        this.dept_name = dept_name;
        this.manager_id = manager_id;
        this.location_id = location_id;
    }

    public Integer getDept_id() {
        return dept_id;
    }

    public void setDept_id(Integer dept_id) {
        this.dept_id = dept_id;
    }

    public String getDept_name() {
        return dept_name;
    }

    public void setDept_name(String dept_name) {
        this.dept_name = dept_name;
    }

    public Integer getManager_id() {
        return manager_id;
    }

    public void setManager_id(Integer manager_id) {
        this.manager_id = manager_id;
    }

    public Integer getLocation_id() {
        return location_id;
    }

    public void setLocation_id(Integer location_id) {
        this.location_id = location_id;
    }


    @Override
    public String toString() {
        return "Department{" +
                "dept_id=" + dept_id +
                ", dept_name='" + dept_name + '\'' +
                ", manager_id=" + manager_id +
                ", location_id=" + location_id +
                '}';
    }
```

```
}
```

编写mapper接口：创建一个名为DeptMapper的接口

```
package com.sys.mapper;

import com.sys.bean.Department;

import java.util.List;

/**
 * @program: MyBatis_02_Mapper映射文件详解
 * @description: 部门表的mapper接口
 * @author: DW
 * @create: 2021-09-26 22:09
 */
public interface DeptMapper {

    /**
     * 根据部门id查询指定的部门信息
     * @param id
     * @return
     */
    Department findDeptById( int id );

}
```

编写Mapper映射文件：

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

    <!--
        使用<resultMap>来自定义结果集映射，用于告诉MaBatis数据表中的字段要和哪个属性进行映射
        属性：
        id：resultMap的名字，方便在其他标签中引用
        type：和resultType，告诉MyBatis需要将结果集中的数据按照 自定义的结果集映射规则 封装
成什么类型的对象
    -->
    <resultMap id="deptMapper" type="department">
        <!--
            <id>标签：设置数据表中的主键 和 实体类的属性 的映射
            属性：
            column：主键名
            property：属性名
        -->
        <id column="department_id" property="dept_id"></id>
        <!--
            <result>标签：设置数据表中的字段 和 实体类的属性 的映射
            属性：
            column：字段名
            property：属性名
```

```
        -->
        <result column="department_name" property="dept_name"></result>
        <result column="manager_id" property="manager_id"></result>
        <result column="location_id" property="location_id"></result>
    </resultMap>

    <!--
        根据部门id查询指定的部门信息
        Department findDeptById( int id );
        注意：这里不再是使用resultType属性而是使用resultMap属性来引用上面的自定义结果集映射
    -->
    <select id="findDeptById" resultMap="deptMapper">
        SELECT * FROM departments WHERE department_id = #{id}
    </select>

</mapper>
```

编写测试方法进行测试：

```
import com.sys.bean.Department;
import com.sys.bean.Employee;
import com.sys.mapper.DeptMapper;
import com.sys.mapper.EmpMapper;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.io.IOException;
import java.io.InputStream;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @program: MyBatis_02_Mapper映射文件详解
 * @description: 测试EmpMapper
 * @author: DW
 * @create: 2021-09-13 16:04
 */
public class TestDeptMapper {

    private SqlSession sqlSession;

    @Before
    public void getSqlSession(){
        // 根据全局配置文件创建出SqlSessionFactory
        // SqlSessionFactory:负责创建SqlSession对象的工厂
        // SqlSession:表示跟数据库建议的一次会话
        String resource = "mybatis-config.xml";
        InputStream inputStream = null;
        try {
            inputStream = Resources.getResourceAsStream(resource);
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
        SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

        // 获取数据库的会话
        sqlSession = sqlSessionFactory.openSession();
    }

    @Test
    public void testFindDeptById(){
        DeptMapper mapper = sqlSession.getMapper(DeptMapper.class);
        Department dept = mapper.findDeptById(10);
        System.out.println( dept );
    }

    @After
    public void release(){
        sqlSession.close();
    }

}
```

2、级联查询

在现实场景中对数据库的查询不可能仅限于单表查询，有时候会涉及到多表查询。

多表查询：也称级联查询，表示本次查询操作会涉及多个数据表的数据。

2.1、一对一

当涉及到多表查询的时候，就需要考虑到表中数据之间的关系。

需求：根据员工id查询指定员工的信息，并查询该员工所在的部门信息。

分析：那么这里设计到两张表，由于一个员工只能属于某一个部门，所以这里员工和部门之间的关系就是“一对一”的关系。

SQL语句：以125进行举例分析

```
SELECT *
FROM employees e
LEFT JOIN departments d
ON e.department_id = d.department_id
WHERE e.employee_id = 125
```

从以上的运行结果发现整个结果集中的数据分为两部分：employees和departments 。如何来处理这个结果集呢？分表使用两个对象？还是使用一个employee对象呢？

实体类设计：

既然两者是一对一的关系，我们可以在Employee中添加一个Department属性：

```
    /**
     * 员工所在的部门
     */
    private Department dept;
```

并为该属性添加set和get方法：

```
    public Department getDept() {
        return dept;
    }

    public void setDept(Department dept) {
        this.dept = dept;
    }
```

再次重写toString()方法：

```
    @Override
    public String toString() {
        return "Employee{" +
                "employee_id=" + employee_id +
                ", first_name='" + first_name + '\'' +
                ", last_name='" + last_name + '\'' +
                ", email='" + email + '\'' +
                ", phone_number='" + phone_number + '\'' +
                ", job_id='" + job_id + '\'' +
                ", salary=" + salary +
                ", commission_pct=" + commission_pct +
                ", manager_id=" + manager_id +
                ", department_id=" + department_id +
                ", hiredate=" + hiredate +
                ", dept=" + dept +
                '}';
    }
```

修改EmpMapper接口：

```
    /**
     * 根据员工id查询指定员工的信息，并查询该员工所在的部门信息。
     * @param id
     * @return
     */
    Employee findEmpAndDept( int id );
```

修改EmpMapper映射文件：

```
    <!--
        由于这里的结果集包含两部分的数据，所以需要使用resultMap设置字段名和属性名之间的映射，将结果集中的数据封装进
        现在的employee对象当中。
    -->
    <resultMap id="empJoinDept" type="employee">
        <id  column="employee_id" property="employee_id"></id>
        <result column="first_name" property="first_name"></result>
```

```
        <result column="last_name" property="last_name"></result>
        <result column="email" property="email"></result>
        <result column="phone_number" property="phone_number"></result>
        <result column="job_id" property="job_id"></result>
        <result column="salary" property="salary"></result>
        <result column="commission_pct" property="commission_pct"></result>
        <result column="manager_id" property="manager_id"></result>
        <result column="department_id" property="department_id"></result>
        <result column="hiredate" property="hiredate"></result>
        <!--
            association：用于一对一的关系，设置实体类属性的自定义结果集映射
            property：实体类属性的名字
            javaType：指定实体类属性所属的实体类
        -->
        <association property="dept" javaType="department">
            <id column="department_id" property="dept_id"></id>
            <result column="department_name" property="dept_name"></result>
            <result column="manager_id" property="manager_id"></result>
            <result column="location_id" property="location_id"></result>
        </association>
    </resultMap>

    <!--
        根据员工id查询指定员工的信息，并查询该员工所在的部门信息。
        Employee findEmpAndDept( int id );
    -->
    <select id="findEmpAndDept" resultMap="empJoinDept" >
        SELECT *
        FROM employees e
        LEFT JOIN departments d
        ON e.department_id = d.department_id
        WHERE e.employee_id = #{id}
    </select>
```

编写测试类方法：

```
    @Test
    public void testFindEmpAndDept(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        Employee employee = mapper.findEmpAndDept(114);
        System.out.println( employee );
    }
```

运行结果：

```
2021-10-05 10:41:59,754 [main] DEBUG [com.sys.mapper.EmpMapper.findEmpAndDept] -
==>  Preparing: SELECT * FROM employees e LEFT JOIN departments d ON
e.department_id = d.department_id WHERE e.employee_id = ?
2021-10-05 10:41:59,771 [main] DEBUG [com.sys.mapper.EmpMapper.findEmpAndDept] -
==> Parameters: 114(Integer)
2021-10-05 10:41:59,786 [main] DEBUG [com.sys.mapper.EmpMapper.findEmpAndDept] -
<==      Total: 1
Employee{employee_id=114, first_name='Den', last_name='Raphaely',
email='DRAPHEAL', phone_number='515.127.4561', job_id='PU_MAN', salary=11000.0,
commission_pct=0.0, manager_id=100, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=Department{dept_id=30, dept_name='Pur', manager_id=100,
location_id=1700}}

Process finished with exit code 0
```

2.2、一对多

需求：根据部门id查询指定部门的信息，并查询该部门下所有的员工信息。

分析：那么这里设计到两张表，由于一个部门下会有多个员工，所以这里部门和员工之间的关系就是“一对多”的关系。

SQL语句：以30进行举例分析

```sql
SELECT *
FROM departments d
LEFT JOIN employees e
ON d.department_id = e.department_id
WHERE d.department_id=30
```

信息 结果 1 剖析 状态

department_id	department_nam	manager_id	location_id	employee_id	first_name	last_name	email	phone_number	job_id	salary	commission_pct	manager_id(1)	department_id(1)	hiredate
30	Pur	114	1700	114	Den	Raphaely	DRAPHE	515.127.4561	PU_MAN	11000.00	(Null)	100	30	2000-09-0
30	Pur	114	1700	115	Alexander	Khoo	AKHOO	515.127.4562	PU_CLERI	3100.00	(Null)	114	30	2000-09-0
30	Pur	114	1700	116	Shelli	Baida	SBAIDA	515.127.4563	PU_CLERI	2900.00	(Null)	114	30	2000-09-0
30	Pur	114	1700	117	Sigal	Tobias	STOBIAS	515.127.4564	PU_CLERI	2800.00	(Null)	114	30	2000-09-0
30	Pur	114	1700	118	Guy	Himuro	GHIMUR	515.127.4565	PU_CLERI	2600.00	(Null)	114	30	2000-09-0
30	Pur	114	1700	119	Karen	Colmenares	KCOLME	515.127.4566	PU_CLERI	2500.00	(Null)	114	30	2000-09-0

参考上一小节的内容，我们需要在Department实体类中添加一个和Employee相关的属性，根据上图一个Department对象会对应多个Employee对象，所有这里需要使用集合来存放这些Employee对象。

实体类设计：在Department实体类中添加集合

```java
    /**
     * 集合属性，用于一对多时，存放多个Employee对象
     */
    private List<Employee> employeeList;
```

修改DeptMapper接口：

```java
    /**
     * 根据部门id查询指定部门的信息，并查询该部门下所有的员工信息。
     * @return
     */
    List<Department> findAllDeptJoinEmpById(int id);
```

编写DeptMapper映射文件：

```
<resultMap id="deptJoinEmp" type="department">
    <id column="department_id" property="dept_id"></id>
    <result column="department_name" property="dept_name"></result>
    <result column="manager_id" property="manager_id"></result>
    <result column="location_id" property="location_id"></result>
    <!--
        <collection>标签：用于设置集合属性的自定义结果集映射
        property：集合属性名
        ofType：指定集合属性的泛型
    -->
    <collection property="employeeList" ofType="employee">
        <id  column="employee_id" property="employee_id"></id>
        <result column="first_name" property="first_name"></result>
        <result column="last_name" property="last_name"></result>
        <result column="email" property="email"></result>
        <result column="phone_number" property="phone_number"></result>
        <result column="job_id" property="job_id"></result>
        <result column="salary" property="salary"></result>
        <result column="commission_pct" property="commission_pct"></result>
        <result column="manager_id" property="manager_id"></result>
        <result column="department_id" property="department_id"></result>
        <result column="hiredate" property="hiredate"></result>
    </collection>
</resultMap>
```

编写测试方法：

```
@Test
public void testFindEmpAndDept(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
    Employee employee = mapper.findEmpAndDept(114);
    System.out.println( employee );
}
```

运行结果：

```
2021-10-05 10:55:17,463 [main] DEBUG
[com.sys.mapper.DeptMapper.findAllDeptJoinEmpById] - ==>  Preparing: SELECT *
FROM departments d LEFT JOIN employees e ON d.department_id = e.department_id
WHERE d.department_id = ?
2021-10-05 10:55:17,482 [main] DEBUG
[com.sys.mapper.DeptMapper.findAllDeptJoinEmpById] - ==> Parameters: 30(Integer)
2021-10-05 10:55:17,499 [main] DEBUG
[com.sys.mapper.DeptMapper.findAllDeptJoinEmpById] - <==      Total: 6
Department{dept_id=30, dept_name='Pur', manager_id=114, location_id=1700,
employeeList=[Employee{employee_id=114, first_name='Den', last_name='Raphaely',
email='DRAPHEAL', phone_number='515.127.4561', job_id='PU_MAN', salary=11000.0,
commission_pct=0.0, manager_id=114, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=null},
Employee{employee_id=115, first_name='Alexander', last_name='Khoo',
email='AKHOO', phone_number='515.127.4562', job_id='PU_CLERK', salary=3100.0,
commission_pct=0.0, manager_id=114, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=null},
Employee{employee_id=116, first_name='Shelli', last_name='Baida',
email='SBAIDA', phone_number='515.127.4563', job_id='PU_CLERK', salary=2900.0,
commission_pct=0.0, manager_id=114, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=null},
Employee{employee_id=117, first_name='Sigal', last_name='Tobias',
email='STOBIAS', phone_number='515.127.4564', job_id='PU_CLERK', salary=2800.0,
commission_pct=0.0, manager_id=114, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=null},
Employee{employee_id=118, first_name='Guy', last_name='Himuro', email='GHIMURO',
phone_number='515.127.4565', job_id='PU_CLERK', salary=2600.0,
commission_pct=0.0, manager_id=114, department_id=30, hiredate=Sat Sep 09
08:00:00 CST 2000, dept=null}, Employee{employee_id=119, first_name='Karen',
last_name='Colmenares', email='KCOLMENA', phone_number='515.127.4566',
job_id='PU_CLERK', salary=2500.0, commission_pct=0.0, manager_id=114,
department_id=30, hiredate=Sat Sep 09 08:00:00 CST 2000, dept=null}]}

Process finished with exit code 0
```

五、动态sql

引入

需求1：根据id查询员工信息。

需求2：根据first_name查询员工信息。

需求3：根据last_name查询员工信息。

需求4：根据email查询员工信息。

等等。

若我们根据以上需求去写代码，其实是没问题的，只不过我们需要写很多的mapper接口方法以mapper映射文件。而且整个SQL语句中，SELECT语句是没变的，唯独是WHERE子句一直在变化，所以我们能否将上述一系列的SQL语句整合在一起呢？让WHERE子句部门根据不同的情况而动态变化呢？

此时，那就使用使用动态SQL了。

1、多条件查询

1.1、if标签

我们来实现下引入部分的需求。

修改EmpMapper接口：

```
    /**
     * 动态sql：多条件查询
     * @param employee：由于条件的个数不确定，所以将多个条件封装进employee对象当中
     * @return
     */
    List<Employee> findEmpsByConditions(Employee employee);
```

修改EmpMapper映射文件：

```
    <!--
        动态sql：多条件查询，根据employees表当中的多个字段进行查询
        List<Employee> findEmpsByConditions(Employee employee);
    -->
    <select id="findEmpsByConditions" resultType="employee">
        SELECT * FROM employees WHERE 1=1
        <!--
            if标签
            test：写形参的判断条件，当判断条件为true时，MyBatis会将AND子句拼接进SELECT语句
中
            注意：形参需要根据接口方法的形参、#{参数}去写
        -->
        <if test="employee_id != 0">
            AND employee_id = #{employee_id}
        </if>
        <if test="first_name != null and first_name != ''">
            AND first_name = #{first_name}
        </if>
        <if test="last_name != null and last_name != ''">
            AND last_name = #{last_name}
        </if>
    </select>
```

编写测试方法：

```
    @Test
    public void testFindEmpsByConditions(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

        // 创建employee对象，通过set方法将参数封装进该对象当中
        Employee employee = new Employee();
//        employee.setEmployee_id(132);
        employee.setFirst_name("TJ");
```

```
//        employee.setLast_name("Olson");

        List<Employee> emps = mapper.findEmpsByConditions(employee);

        // 遍历集合
        for ( Employee emp : emps) {
            System.out.println(emp);
        }
    }
```

<if>标签说明：

当if标签中的test属性为true的时候，MyBatis就会将该if标签中的sql拼接进select语句，所以使用该标签**可以拼接任意个条件。**

if标签相当是Java当中的"与运算"，也类似于if单分支结构，但是不支持if...else...结构的功能。

1.2、choose标签

和 if标签 对应的还有一个choose标签。我们先来看看该标签怎么使用。

修改EmpMapper接口：

```
    /**
     * 动态sql2
     * @param employee：将多个条件封装进employee对象当中
     * @return
     */
    List<Employee> findEmpsByConditions2(Employee employee);
```

修改EmpMapper映射文件：

```
    <!--
        动态sql2
        List<Employee> findEmpsByConditions2(Employee employee);
    -->
    <select id="findEmpsByConditions2" resultType="employee">
        SELECT * FROM employees WHERE 1=1
        <choose>
            <when test="employee_id != 0">
                AND employee_id = #{employee_id}
            </when>
            <when test="first_name != null and first_name != ''">
                AND first_name = #{first_name}
            </when>
            <when test="last_name != null and last_name != ''">
                AND last_name = #{last_name}
            </when>
        </choose>
    </select>
```

编写测试方法：

```
    @Test
    public void testFindEmpsByConditions2(){
```

```
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

        // 创建employee对象，通过set方法将参数封装进该对象当中
        Employee employee = new Employee();
        employee.setEmployee_id(132);
        employee.setFirst_name("TJ");
        employee.setLast_name("Olson");

        List<Employee> emps = mapper.findEmpsByConditions2(employee);

        // 遍历集合
        for ( Employee emp : emps) {
            System.out.println(emp);
        }
    }
```

`<choose>` **标签说明：**

在choose当中，**when标签语句是从上往下依次执行的**，
（1）若when标签中的test属性为false时，会继续往下执行；

（2）当遇到了when标签中的test属性为true时，则整个choose就结束，MyBatis会将该when标签中的sql拼接进select语句。

（3）当所有的when标签都为false的时候，就会执行otherwise标签，若没有otherwise标签，则直接结束整个choose，此时无sql语句拼接。

所以 choose标签能拼接0个或者1个条件。

choose相当于是Java当中的"或运算"，类似switch结构，choose相当于是switch，when相当于case，otherwise相当于default。

注意：当choose结构在某个when标签中结束时， 后面when标签是没有执行的。

2、in子句的动态参数问题

需求：根据job_id去查询员工信息。

分析：在现实场景中，job_id是由前端页面发送到后端的，用户在输入job_id时，可能输入了1个、2个、3个...N个等等。所以我们的SQL语句当中的参数也是动态变化的。

修改EmpMapper接口：

```
    /**
     * 动态sql：in字句中的参数动态变化
     * 需求：根据job_id去查询员工信息（job_id可能有1个、2个、3个...N个）
     * @param job_ids
     * @return
     */
    List<Employee> findEmpsByJobIds(List<String> job_ids);
```

修改EmpMapper映射文件：

```
    <!--
        List<Employee> findEmpsByJobIds(List<String> job_ids);
    -->
```

```
    <select id="findEmpsByJobIds" resultType="employee">
        SELECT * FROM employees WHERE job_id IN
        <!--
            foreach标签：用于处理接口方法的参数为集合的情况，主要解决in子句中动态参数的问题
            属性：
            item：设置集合中元素的别名
            index：设置集合中元素的索引类型，List集合则为：index
            collection：表示集合的类型，List集合则为：list
            open：in子句的开始标记：(
            separator：表示in子句的分隔符：,
            close：in子句的结束标记：)

            注意：sql语句的参数 #{item的属性值}
        -->
        <foreach item="jid" index="index" collection="list" open="("
separator="," close=")">
            #{jid}
        </foreach>
    </select>
```

编写测试方法：

```
    @Test
    public void testFindEmpsByJobIds(){
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

        List<String> job_ids = new ArrayList<String>();
        job_ids.add("ST_CLERK");
        job_ids.add("SA_MAN");

        List<Employee> emps = mapper.findEmpsByJobIds(job_ids);

        // 遍历集合
        for ( Employee emp : emps) {
            System.out.println(emp);
        }
    }
```

foreach标签常见使用场景是对集合进行遍历，尤其是在构建 IN 条件语句的时候。

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

提示：你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象作为集合参数传递给 *foreach*。当使用可迭代对象或者数组时，index 是当前迭代的序号，item 的值是本次迭代获取到的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

3、SQL片段

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

比如：我们的EmpMapper映射文件中出现了许多的`"SELECT * FROM employees"`那我们是否可以将它们提取出来呢?

```
    <!--
        SQL片段
    -->
    <sql id="select*Emp">
        SELECT * FROM employees
    </sql>

    <!--
        动态sql：多条件查询，根据employees表当中的多个字段进行查询
        List<Employee> findEmpsByConditions(Employee employee);
    -->
    <select id="findEmpsByConditions" resultType="employee">
        <include refid="select*Emp"></include> WHERE 1=1
        <!--
            if标签
            test：写形参的判断条件，当判断条件为true时，MyBatis会将AND子句拼接进SELECT语句
中
            注意：形参需要根据接口方法的形参、#{参数}去写
        -->
        <if test="employee_id != 0">
            AND employee_id = #{employee_id}
        </if>
        <if test="first_name != null and first_name != ''">
            AND first_name = #{first_name}
        </if>
        <if test="last_name != null and last_name != ''">
            AND last_name = #{last_name}
        </if>
    </select>
```

六、其他操作

1、insert操作

需求：在employees表中添加一行数据

在EmpMapper接口中编写方法：

```
    /**
     * 添加数据
     * @param employee
     * @return
     */
    Integer addEmp(Employee employee);
```

修改EmpMapper映射文件：

```
    <insert id="addEmp">
        INSERT INTO employees (first_name,last_name,email)
        VALUES (#{first_name},#{last_name},#{email})
    </insert>
```

编写测试方法：

```java
@Test
public void testAddEmp(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

    Employee employee = new Employee();
    employee.setFirst_name("Wei");
    employee.setLast_name("Wu");
    employee.setEmail("1323789218@qq.com");

    Integer row = mapper.addEmp(employee);
    // 提交操作，否则数据不会保存到数据库当中
    sqlSession.commit();
    System.out.println("影响的行数为："+row);
}
```

2、update操作

需求：修改刚刚添加的数据。

在EmpMapper接口中编写方法：

```java
/**
 * 修改指定员工的数据
 * @param last_name
 * @return
 */
Integer updateEmp(@Param("l_name") String last_name, @Param("id") int id);
```

修改EmpMapper映射文件：

```xml
<update id="updateEmp">
    UPDATE employees SET
    last_name = #{l_name}
    WHERE employee_id = #{id}
</update>
```

编写测试方法：

```java
@Test
public void testUpdateEmp(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

    Integer row = mapper.updateEmp("HaSaGei", 218);
    // 提交操作，否则数据不会保存到数据库当中
    sqlSession.commit();
    System.out.println("影响的行数为："+row);
}
```

3、delete操作

需求：删除刚刚添加的数据。

在EmpMapper接口中编写方法：

```
/**
 * 删除指定的员工信息
 * @param id
 * @return
 */
Integer deleteEmp(int id);
```

修改EmpMapper映射文件：

```
<delete id="deleteEmp">
    DELETE FROM employees WHERE employee_id = #{id}
</delete>
```

编写测试方法：

```
@Test
public void testDeleteEmp(){
    EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);

    Integer row = mapper.deleteEmp(218);
    // 提交操作，否则数据不会保存到数据库当中
    sqlSession.commit();
    System.out.println("影响的行数为："+row);
}
```