# 前言一

无论基于何种原因，学习本门课程之前请务必已经熟练掌握如下知识点。

JavaSE

数据库（Mysql或者Oracle）

JDBC(其实它也是JavaSE的一部分，这里列出来是强调一下该知识点)

XML

# 前言二

为什么学习和使用Mybatis？

## 传统的JDBC操作数据库

传统的jdbc虽然可以操作数据库，但是存在硬编码，高耦合以及性能等问题

1. 在创建连接时，存在硬编码---解决方案：配置文件（全局配置文件）

Connectionconnection=DriverManager.*getConnection*("com.mysql.jdbc.Driver", "root", "root")；

1. 在执行statement时存在硬编码---解决方案：配置文件（映射文件）

statement.executeQuery("select \* from emp where id = 7566");

1. 频繁的开启和关闭数据库连接，会造成数据库性能下降。---解决方案：数据库连接池（全局配置文件）

statement.close();

connection.close();

## Hibernate操作数据库

如果没学习过Hibernate，不要紧，可以直接略过该段内容。

下面是一个最简单的Hibernate操作数据库的代码，我们知道这样的好处是让开发者完全通过面向对象的方式来操作数据库，真正的SQL语句由Hibernate来自动生成。所以它是一个很优秀的持久层框架，犹如全自动洗衣机一样让人省心，省力。

然而，并不是所有的衣服都可以用全自动洗衣机来洗。同样，在项目中，由于Hbernate自动生成SQL语句，导致某些复杂查询效率很低，这个时候就需要自定义优化好的SQL语句，所以在某些场景下，Hibernate的这种全自动生成SQL语句的方式并不适合，最好的方式是依然将SQL语句交给开发者来实现。

Emp emp = **new** Emp();

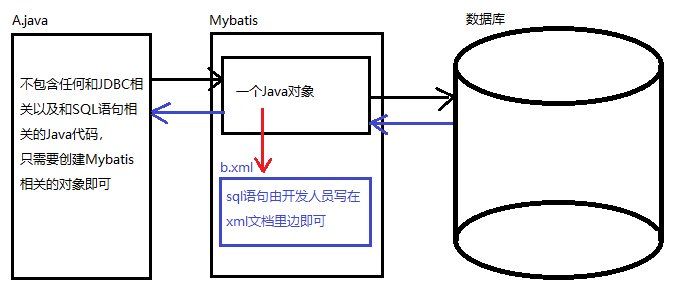
emp.seteName("小钻风");

session.save(emp);

session.close();

## Mybatis操作数据库

由于我们还没学习Mybatis，或者说刚刚准备打开Mybatis这扇大门，所以这里并没有Mybatis操作数据库的代码，而是一副图告诉你，Mybatis是一个半自动的持久层的框架。



当开发者想优化SQL语句的时候，不用重新编译-打包整个项目，只需要修改那个包含了该SQL语句的xml文件即可。然后剩下的全部工作交给Mybatis来完成，所以说它是一个半自动化，高效灵活，轻量级的持久层框架。

# Mybatis简介

MyBatis 本是[apache](https://baike.baidu.com/item/apache)的一个开源项目[iBatis](https://baike.baidu.com/item/iBatis), 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。直接目前最新版本是 3.5.0

iBATIS一词来源于“internet”和“abatis”的组合，是一个基于Java的[持久层](https://baike.baidu.com/item/%E6%8C%81%E4%B9%85%E5%B1%82)框架。iBATIS提供的持久层框架包括SQL Maps和Data Access Objects（DAOs）

Mybatis就是一个封装来jdbc的持久层框架，它和Hibernate都属于ORM框架，但是具体的说，Hibernate是一个全自动的orm框架，而Mybatis是一个半自动的orm框架。

Mybatis让程序员只关注sql本身，而不需要去关注如连接的创建、statement的创建等操作。

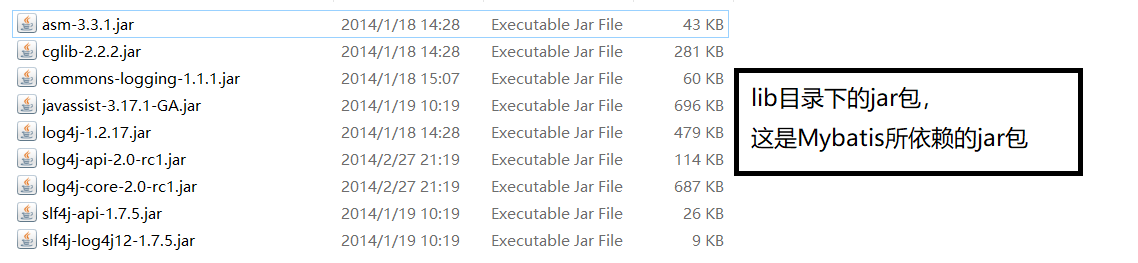
下载地址

<https://github.com/mybatis>

注意：以前的github不需要注册登陆，直接访问即可。但是后来它“变坏”了，你需要拥有一个github的账号。

下载Mybatis并解压缩





# Mybatis入门程序

**【案例需求：使用Mybatis对emp表进行CRUD操作】**

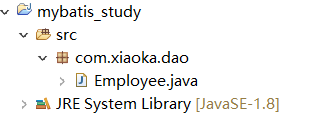
**准备数据库**

****

**打开Mybatis官方文档**

## 简单查询-返回一条记录

1. 创建Java程序，创建JavaBean



**public** **class** Employee {

//属性名称要和数据库字段名称保持一致

**private** **int** empno;

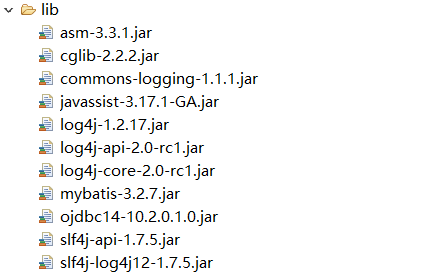
**private** String eName;

//getter/setter方法略...

}

1. 导入所需要jar包，并创建全局配置文件sqlMapConfig.xml,

①导入Mybatis的jar包和数据库驱动包



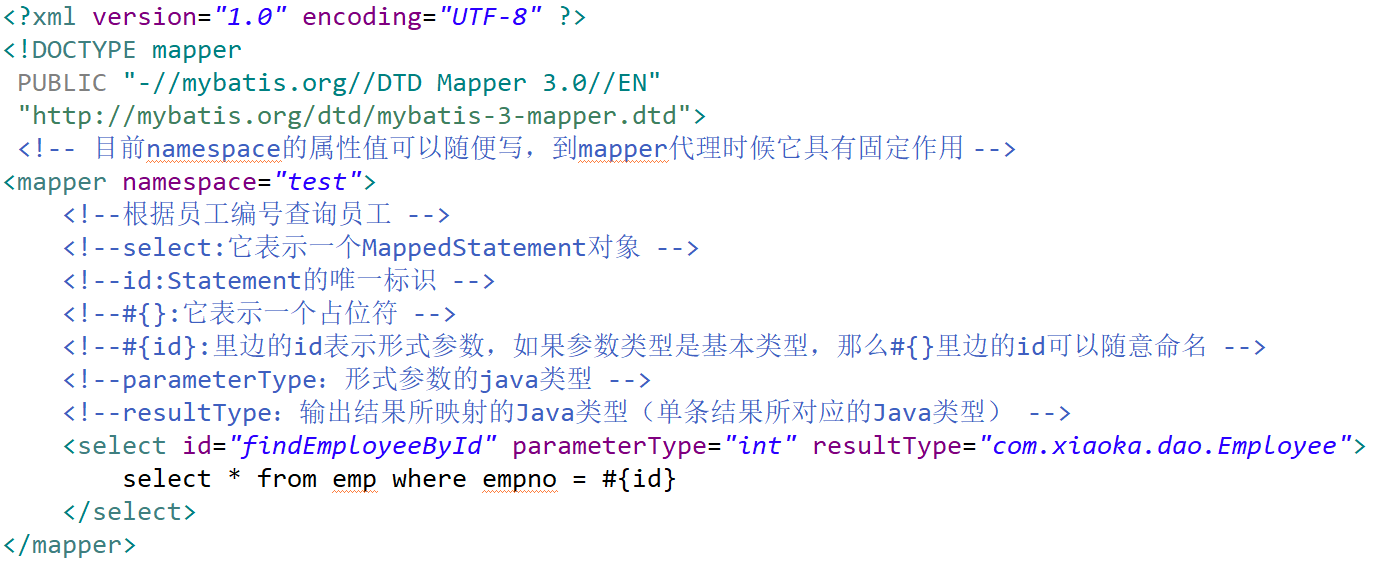
②创建资源目录config,并在其中创建全局配置文件sqlMapConfig.xml,该文件名称任意

该配置信息在Mybatis的说明文档中可以查阅到



1. 创建映射文件并在全局配置文件中加载

①：在资源目录config中创建employee.xml映射文件，其中的配置信息在Mybatis的文档中有。



②：将映射文件employee.xml文件加载到全局配置文件中

<mappers>

<!-- 这里用来加载映射文件 -->

mapper resource=*"employee.xml"* />

</mappers>

1. 运行测试代码

@Test

**public** **void** testFindEmployeeById() **throws** Exception {

//读取全局配置文件

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

//根据全局配置文件创建工厂类

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

//获得可以执行CRUD的session

SqlSession session = sqlSessionFactory.openSession();

/\*\*

\* 执行sql语句

\* 第一个参数：statement的唯一标识：命名空间+id

\*/

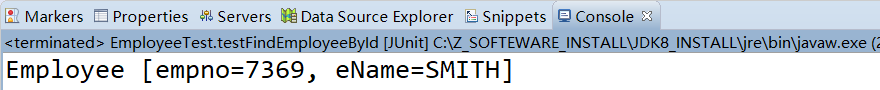
Employee employee = session.selectOne("test.findEmployeeById", 7369);

System.***out***.println(employee);

//关闭资源

session.close();

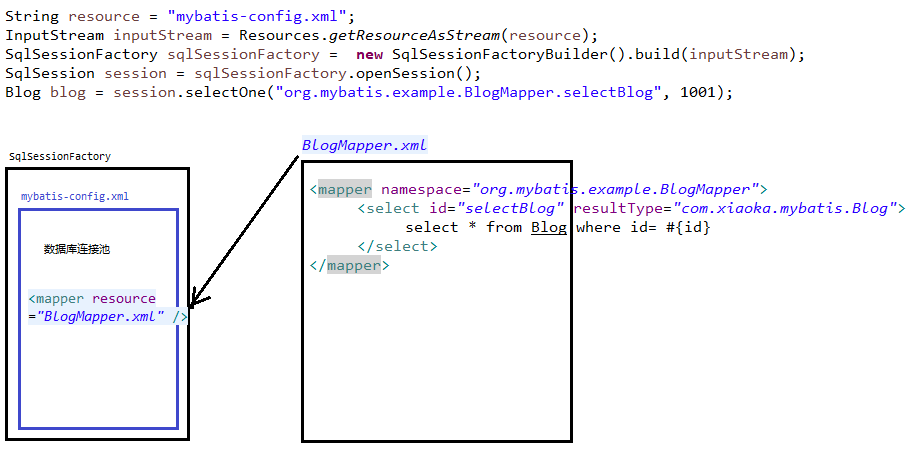
}



至此，入门程序结束。可以查询结果正常显示打印到了控制台。

小结：只需要在映射文件中配置sql语句即可使用SqlSession来执行。

代码图示



## 模糊查询-返回多条记录

1. 映射文件employee.xml

<!-- 根据员工姓名查询员工 -->

<!-- ${}:表示sql的连接符 -->

<!-- ${value}:value是形式参数，如果参数是基本类型，那么参数名称就必须是value -->

<!-- ${value}:拼接sql存在注入的风险，尽量不要使用。但是在某些场景下必须使用${}。比如动态传入排序的列名 -->

<!-- 即使查询结果是集合，返回值类型也是填写集合的泛型类 -->

<select id=*"findEmployeeByName"* parameterType=*"java.lang.String"* resultType=*"com.xiaoka.dao.Employee"*>

SELECT \* FROM EMP WHERE ENAME LIKE '%${value}%'

</select>

1. 测试代码

@Test

**public** **void** testFindEmployeeByName() **throws** Exception{

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

//查询结果是集合的时候，需要使用selectList函数

List<Employee> list = session.selectList("test.findEmployeeByName", "M");

System.***out***.println(list);

session.close();

}



## 插入数据

### -添加员工

1. 映射文件employee.xml

<!-- 添加员工，形式参数是JavaBean类对应的属性名称 -->

<insert id=*"insertEmployee"* parameterType=*"com.xiaoka.dao.Employee"*>

insert into emp (empno,eName) values (#{empno},#{eName})

</insert>

1. 测试代码

@Test

**public** **void** testInsertEmployee() **throws** Exception{

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

Employee employee = **new** Employee(7777,"王小卡");

session.insert("test.insertEmployee", employee);

System.***out***.println(employee);

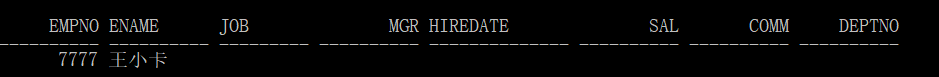
//提交事物

session.commit();

session.close();

}

运行结果：数据已经添加到emp表中



## 更新数据

1. 映射文件employee.xml

<update id=*"updateEmployeeById"* parameterType=*"com.xiaoka.dao.Employee"*>

update emp set ename = #{eName} where empno = #{empno}

</update>

1. 测试代码

@Test

**public** **void** testUpdateEmployeeById() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

Employee employee = **new** Employee(7777,"小钻风");

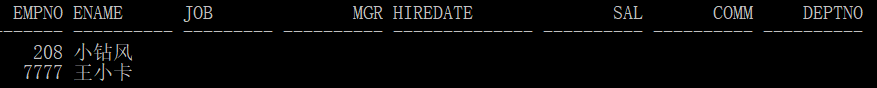
session.update("test.updateEmployeeById", employee);

session.commit();

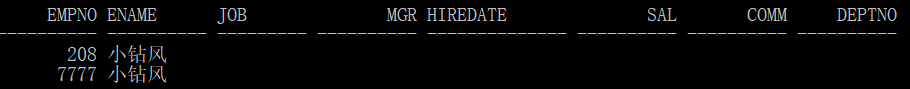
session.close();

}

运行测试代码之前



运行测试代码之后，员工编号7777的数据确实被修改



## 删除数据

1. 映射文件employee.xml

<delete id=*"deleteEmployeeById"* parameterType=*"int"*>

delete from emp where empno = #{id}

</delete>

1. 测试代码

@Test

**public** **void** testDeleteEmployeeById() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

session.delete("test.deleteEmployeeById", 7777);

session.commit();

session.close();

}

运行结果，编号7777员工被删除



## 小结

* #{}和${}
  + #{}表示占位符?，#{}接收简单类型的参数时，里面的名称可以任意
  + ${}表示拼接符，${}接收简单类型的参数时，里面的名称必须是value
  + ${}里面的值会原样输出，不加解析（如果该参数值是字符串，有不会添加引号）
  + ${}存在sql注入的风险，但是有些场景下必须使用，比如排序后面会动态传入排序的列名
* parameterType和resultType
  + parameterType指定输入参数的java类型，parameterType只有一个，也就是说入参只有一个。
  + resultType指定输出结果的java类型（是单条记录的java类型）
* selectOne和selectList
  + selectOne查询单个对象
  + selectList查询集合对象

# Mybatis结合Dao的使用

在上一章节中，我们使用测试代码实现了Mybatis的快速入门，已经能够正常的进行CRUD的操作。本章，我们将Dao和Mybatis结合到一起，来实现数据库的CRUD操。

## 【案例需求：使用Mybatis和Dao结合的方式对emp表进行CRUD操作】

我们已经知道在Mybatis中执行sql语句的对象就是SqlSession对象，那么一起使用Dao和Mybatis只需要在Dao中注入SelSessionFactory对象，在具体的方法中由SqlSessionFactory对象来获取SqlSession对象进行sql语句的执行即可。

SqlSessionFactory，它的生命周期，应该是应用范围，全局范围只有一个工厂，使用单例模式来实现这个功能。与spring集成之后，由spring来对其进行单例管理。

SqlSession，它内部含有一块数据区域，存在线程不安全的问题，所以应该将sqlsession声明到方法内部。

注意：SqlSession内部的数据区域是一级缓存，是通过Map集合来存储的。

## 全局配置文件salMapConfig.xml和映射文件employee.xml保持不变】

## 创建Dao接口和实现类

**public** **interface** EmployeeDao {

**public** Employee findEmployeeById(**int** empno);

**public** List<Employee> findEmployeeByName(String eName);

**public** **void** insertEmployee2(Employee employee);

**public** **void** updateEmployeeById(Employee employee);

**public** **void** deleteEmployeeById(**int** empno);

}

**public** **class** EmployeeDaoImpl **implements** EmployeeDao {

**private** SqlSessionFactory sessionFactory;

**public** EmployeeDaoImpl(SqlSessionFactory sessionFactory) {

**this**.sessionFactory = sessionFactory;

}

**public** Employee findEmployeeById(**int** empno) {

SqlSession session = sessionFactory.openSession();

Employee employee = session.selectOne("test.findEmployeeById", empno);

**return** employee;

}

@Override

**public** List<Employee> findEmployeeByName(String eName) {

SqlSession session = sessionFactory.openSession();

// 查询结果是集合的时候，需要使用selectList函数

List<Employee> list = session.selectList("test.findEmployeeByName", eName);

session.close();

**return** list;

}

@Override

**public** **void** insertEmployee2(Employee employee) {

SqlSession session = sessionFactory.openSession();

session.insert("test.insertEmployee2", employee);

// 提交事物

session.commit();

session.close();

}

@Override

**public** **void** updateEmployeeById(Employee employee) {

SqlSession session = sessionFactory.openSession();

session.update("test.updateEmployeeById", employee);

session.commit();

session.close();

}

@Override

**public** **void** deleteEmployeeById(**int** empno) {

SqlSession session = sessionFactory.openSession();

session.delete("test.deleteEmployeeById", empno);

session.commit();

session.close();

}

}

## 创建测试类EmployeeDaoTest

**public** **class** EmployeeDaoTest {

**private** SqlSessionFactory sessionFactory;

**private** EmployeeDao employeeDao;

@Before

**public** **void** setUp() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

sessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

employeeDao = **new** EmployeeDaoImpl(sessionFactory);

}

@Test

**public** **void** testFindEmployeeById() {

Employee employee = employeeDao.findEmployeeById(7369);

System.***out***.println(employee);

}

@Test

**public** **void** testFindEmployeeByName() {

List<Employee> list = employeeDao.findEmployeeByName("M");

System.***out***.println(list);

}

@Test

**public** **void** testInsertEmployee2() {

Employee employee = **new** Employee();

employee.seteName("王小卡");

employeeDao.insertEmployee2(employee);

System.***out***.println(employee);

}

@Test

**public** **void** testUpdateEmployeeById() {

Employee employee = **new** Employee(7369,"王小卡");

employeeDao.updateEmployeeById(employee);

}

@Test

**public** **void** testDeleteEmployeeById() {

employeeDao.deleteEmployeeById(7369);

}

}

这就是将Mybatis和Dao结合的方式，经测试，CRUD操作都没问题。功能已经全部实现，但是代码中存在如下问题点

一：大量重复代码

二：存在硬编码

而使用Mapper代理则可以解决这两个问题。

# Mapper代理的开发方式

使用Mapper代理其实就是开发Mapper接口[相当于Dao接口]

Mapper代理使用的是JDK的代理策略

## Mapper代理的开发规范

1. mapper接口的全限定名要和mapper映射文件的namespace值一致。
2. mapper接口的方法名称要和mapper映射文件的statement的id一致。
3. mapper接口的方法参数类型要和mapper映射文件的statement的parameterType的值一致，而且它的参数是一个。
4. mapper接口的方法返回值类型要和mapper映射文件的statement的resultType的值一致。

## 代码实现

### 创建EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** Employee findEmployeeById(**int** empno);

**public** List<Employee> findEmployeeByName(String eName);

**public** **void** insertEmployee2(Employee employee);

**public** **void** updateEmployeeById(Employee employee);

**public** **void** deleteEmployeeById(**int** empno);

}

### 创建EmployeeMapper.xml映射文件

在资源目录config中创建EmployeeMapper.xml映射文件

该文件名称任意，使用接口做文件名称是命名规范。

<?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映射文件namespace要和Mapper接口全类名一致 -->

<mapper namespace=*"com.xiaoka.mapper.EmployeeMapper"*>

<!--id值要和Mapper接口的方法名称一致 -->

<select id=*"findEmployeeById"* parameterType=*"int"*

resultType=*"com.xiaoka.bean.Employee"*>

select \* from emp where empno = #{id}

</select>

<select id=*"findEmployeeByName"* parameterType=*"java.lang.String"*

resultType=*"com.xiaoka.bean.Employee"*>

SELECT \* FROM EMP WHERE ENAME LIKE '%${value}%'

</select>

<insert id=*"insertEmployee"* parameterType=*"com.xiaoka.bean.Employee"*>

insert into emp

(empno,eName) values (#{empno},#{eName})

</insert>

<insert id=*"insertEmployee2"* parameterType=*"com.xiaoka.bean.Employee"*>

<selectKey keyProperty=*"empno"* resultType=*"int"* order=*"BEFORE"*>

SELECT EMPLOYEES\_SEQ.NEXTVAL FROM DUAL

</selectKey>

INSERT INTO EMP (EMPNO,ENAME) VALUES (#{empno},#{eName})

</insert>

<update id=*"updateEmployeeById"* parameterType=*"com.xiaoka.bean.Employee"*>

update emp set ename = #{eName} where empno = #{empno}

</update>

<delete id=*"deleteEmployeeById"* parameterType=*"int"*>

delete from emp where empno = #{id}

</delete>

</mapper>

### 在全局配置文件sqlMapConfig.xml中加载映射文件

<mappers>

<!-- 加载mapper映射文件 -->

<mapper resource=*"employeeMapper.xml"*/>

</mappers>

### 生成测试代码

**public** **class** EmployeeMapperTest {

**private** SqlSessionFactory sessionFactory;

@Before

**public** **void** setUp() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

sessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

@Test

**public** **void** testFindEmployeeById() {

SqlSession session = sessionFactory.openSession();

//通过字节码对象获取具体的Mapper代理对象

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

Employee employee = employeeMapper.findEmployeeById(7521);

//测试结果：可以查询到该员工

System.***out***.println(employee);

session.close();

}

@Test

**public** **void** testFindEmployeeByName() {

SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

List<Employee> list = employeeMapper.findEmployeeByName("M");

//测试结果：可以看到该集合中员工name都是包含M的

System.***out***.println(list);

session.close();

}

@Test

**public** **void** testInsertEmployee2() {

SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

Employee employee = **new** Employee();

employee.seteName("王小卡");

employeeMapper.insertEmployee2(employee);

//测试结果：王小卡成功添加到数据库，并且序列生成的主键值赋值给了当前对象的empno属性

System.***out***.println(employee);

session.commit();

session.close();

}

@Test

**public** **void** testUpdateEmployeeById() {

SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

//测试结果：将编号为213员工姓名成功修改成小钻风

employeeMapper.updateEmployeeById(**new** Employee(213,"小钻风"));

session.commit();

session.close();

}

@Test

**public** **void** testDeleteEmployeeById() {

SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

//测试结果：成功删除编号为213员工

employeeMapper.deleteEmployeeById(213);

session.commit();

session.close();

}

}

# 全局配置文件详解

全局配置文件sqlMapCong.xml中常用的标签以及顺序如下

|  |  |  |  |
| --- | --- | --- | --- |
| **标签名称** | | | **标签作用** |
| poperties | | | 属性配置 |
| settings | | | 设置全局参数 |
| typealiases | | | 设置别名 |
| typeHandlers | | | 类型处理器 |
| objectFactory | | | 对象工厂 |
| plugins | | | 插件 |
| environments | | | 环境信息集合 |
|  | environment | | 单个环境信息 |
|  | transactionManager | 事物 |
| dataSource | 数据源 |
| databaseIdProvider | | | 指定数据库厂商 |
| mappers | | | 映射器 |

## properties标签

全局配文件sqlMapconfig.xml

<configuration>

<!-- 加载属性配置文件 -->

<!-- 加载顺序：

首先加载properties的各个property属性节点

其次加载propetties的resource属性指向的db.properties文件

当db.properties中的key和属性property中的name属性值一致的

时候，后者覆盖前者，即属性文件会中的值覆盖property属性节点的值

-->

<properties resource=*"db.properties"*>

<property name=*"username"* value=*"scott"*/>

<property name=*"db.password"* value=*"tiger"*/>

</properties>

<environments default=*"development"*>

<environment id=*"development"*>

<transactionManager type=*"JDBC"* />

<dataSource type=*"POOLED"*>

<!-- 使用EL表达式获取属性配置文件中的值 -->

<property name=*"driver"* value=*"${db.driverClass}"* />

<property name=*"url"* value=*"${db.URL}"* />

<property name=*"username"* value=*"${db.userName}"* />

<property name=*"password"* value=*"${db.password}"* />

</dataSource>

</environment>

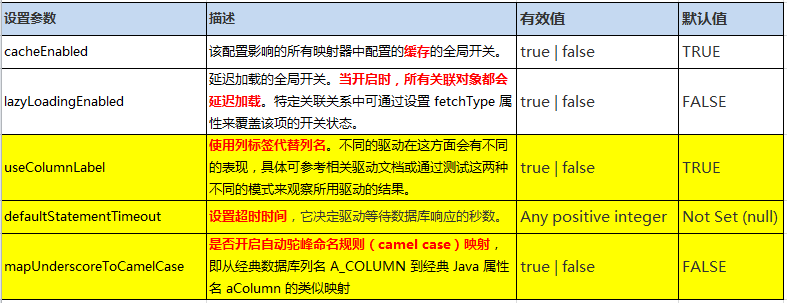
</environments>

...其它配置信息...

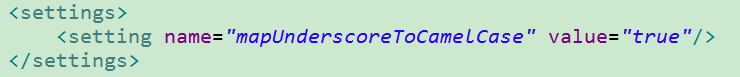
</configuration>

## settings标签

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



示例配置



## typealiases标签

类型别名是为 Java 类型设置一个短的名字，可以方便我们引用某个类。

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

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

**自定义别名**

一：定义一个类的别名

<typeAliases>

<!-- 定义单个类的别名 -->

<typeAlias type=*"com.xiaoka.bean.Employee"* alias=*"employee"* />

</typeAliases>

二：批量定义别名

<typeAliases>

<!-- 批量定义别名 -->

<!-- name指的包下所有的类都会被定义别名，别名默认为类名[首字母大小写都行] -->

<package name=*"com.xiaoka.bean"* />

</typeAliases>

**别名的使用**

**映射文件中的parameterType和resultType都可以使用别名，无需使用po的全限定类名**

<!--parameterType和resultType都可以使用别名 -->

<select id=*"findEmployeeById"* parameterType=*"int"* resultType=*"employee"*>

select \* from emp where empno = #{id}

</select>

## typeHandlers标签

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

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

## plugins标签

插件是MyBatis提供的一个非常强大的机制，我们可以通过插件来修改MyBatis的一些核心行为。插件通过动态代理机制，可以介入四大对象的任何一个方法的执行。

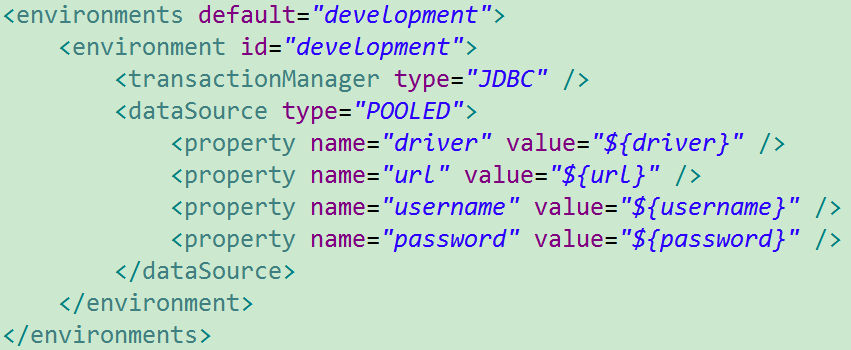
* + Executor (update, query, flushStatements, commit, rollback,getTransaction, close, isClosed)
  + ParameterHandler (getParameterObject, setParameters)
  + ResultSetHandler (handleResultSets, handleOutputParameters)
  + StatementHandler (prepare, parameterize, batch, update, query)

## environments

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

### environment标签的具体环境

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



#### transactionManager

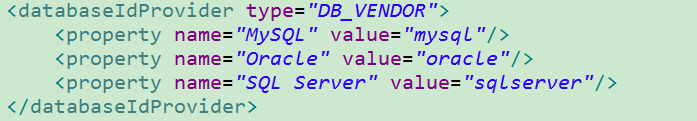
* type： JDBC | MANAGED | 自定义
  + JDBC：使用 JDBC 的提交和回滚设置，依赖于从数据源得到的连接来管理事务范围。
    - JDBC 是JdbcTransactionFactory的别名
  + MANAGED：不提交或回滚一个连接、让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。
    - MANAGED 是ManagedTransactionFactory的别名
  + 自定义：实现TransactionFactory接口，type=全类名/别名

#### dataSource

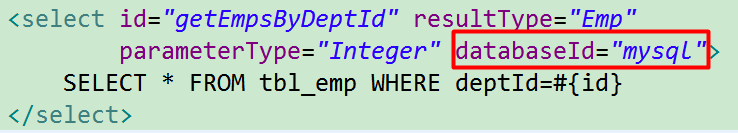
* type： UNPOOLED | POOLED | JNDI | 自定义
  + UNPOOLED：不使用连接池
    - UNPOOLED 是UnpooledDataSourceFactory的别名
  + POOLED：使用MyBatis内置连接池，
    - POOLED 是PooledDataSourceFactory的别名
  + JNDI： 在EJB 或应用服务器这类容器中查找指定的数据源
  + 自定义：实现DataSourceFactory接口，定义数据源的获取方式。
* 实际开发中我们使用Spring管理数据源，并进行事务控制的配置来覆盖上述配置

## databaseIdProvider

MyBatis 可以根据不同的数据库厂商执行不同的语句。



对应的Mapper中要如下设置



MyBatis匹配规则

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

## mappers标签

mappers加载映射文件的几种方式

* 使用相对与类路径资源

<mappers>

<mapper resource=*"employeeMapper.xml"* />

</mappers>

* 使用绝对路径

<mappers>

<mapper url=*"file:///D:\workspace\_EclipseEE\mybatis\_study\config\employeeMapper.xml"* />

</mappers>

* 使用Mapper接口的全限定类名

这种使用方式要求映射文件和Mapper接口必须在同一个目录下，且映射文件名称和Mapper接口名称一致。即：同包同名

<mappers>

<mapper class=*"com.xiaoka.mapper.EmployeeMapper"*/>

</mappers>

* 注册指定包下的所有映射文件

批量加载映射文件的方式也要求映射文件和Mapper接口同包同名。

<mappers>

<package name=*"com.xiaoka.mapper"* />

</mappers>

# 映射文件详解

上一章讲解了全局配置文件sqlMapConfig.xml中的标签，本章学习映射文件employeeMapper.xml中的其它标签和属性。

映射文件中分别有如下标签

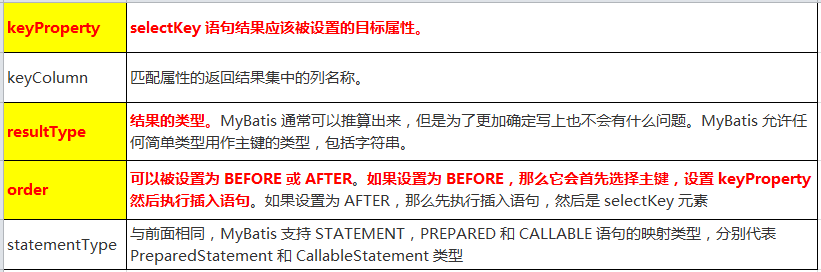
* + cache –命名空间的二级缓存配置
  + cache-ref – 其他命名空间缓存配置的引用。
  + resultMap – 自定义结果集映射
  + parameterMap – 已废弃！老式风格的参数映射
  + sql –抽取可重用语句块。
  + insert – 映射插入语句
  + update – 映射更新语句
  + delete – 映射删除语句
  + select – 映射查询语句

insert 、 update 、 delete 元素的属性



### Insert标签中的selectKey

很多情况下，插入数据时，数据的id值不是用户填入的，而是由数据库自己生成的，这个时候就要求在插入数据以后拿到数据的id值



#### -插入数据-主键返回-自增主键

<!-- 添加员工--主键返回—Mysql自增主键 -->

<insert id=*"insertEmployee2"* parameterType=*"com.xiaoka.dao.Employee"*>

<!—selectKey:该标签用来查询主键，所以在标签内要输入查询主键的sql语句 -->

<!-- order="AFTER"：它指定查询主键的sql语句在insert语句之后执行 -->

<!-- SELECT LAST\_INSERT\_ID()函数是Mysql的函数，用于获取自增主键的ID,必须跟insert语句一起使用 -->

<!--keyProperty="empno"：指定生成的序列值赋值给JavaBean的empno属性 -->

<!--resultType="int"：指定生成的序列值的返回值类型 -->

<selectKey keyProperty=*"empno"* resultType=*"int"* order=*"AFTER"*>

SELECT LAST\_INSERT\_ID()

</selectKey>

<!—插入数据不用再显示的给empno列添加值，这是由Mysql的自增策略决定的 -->

insert into emp (eName) values (#{eName})

</insert>

#### -插入数据-主键返回-自增主键-UUID

<!-- 添加员工--主键返回—Mysql-UUID -->

<insert id=*"insertEmployee2"* parameterType=*"com.xiaoka.dao.Employee"*>

<!—- UUID()函数是Mysql的函数 -->

<selectKey keyProperty=*"empno"* resultType=*"int"* order=*"BEFORE"*>

SELECT UUID()

</selectKey>

<!—插入数据需要显示的给empno列添加值-->

insert into emp (empno,eName) values (#{empno},#{eName})

</insert>

#### -插入数据-主键返回-oracle序列

1. 映射文件employee.xml

<!-- 添加员工--主键返回--oracle序列 -->

<insert id=*"insertEmployee2"* parameterType=*"com.xiaoka.dao.Employee"*>

<!-- order="BEFORE"：它指定生成序列值的sql语句在插入数据语句之前执行 -->

<!--keyProperty="empno"：指定生成的序列值赋值给JavaBean的empno属性 -->

<!--resultType="int"：指定生成的序列值的返回值类型 -->

<selectKey keyProperty=*"empno"* resultType=*"int"* order=*"BEFORE"*>

SELECT EMPLOYEES\_SEQ.NEXTVAL FROM DUAL

</selectKey>

insert into emp (empno,eName) values (#{empno},#{eName})

</insert>

1. 测试代码

@Test

**public** **void** testInsertEmployee2() **throws** Exception{

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

Employee employee = **new** Employee();

employee.seteName("小钻风");

System.***out***.println(employee);

session.insert("test.insertEmployee2", employee);

System.***out***.println(employee);

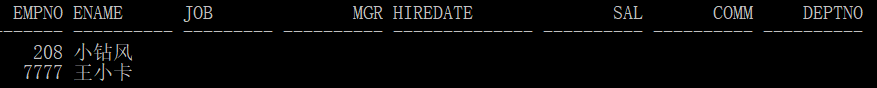
//提交事物

session.commit();

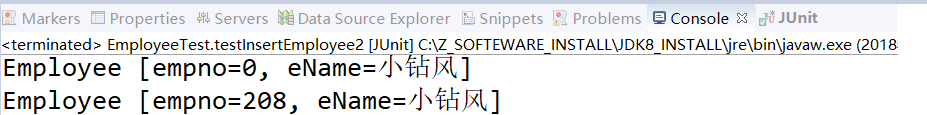
session.close();

}

数据库：数据已经添加进来



控制台：第一条输出，employee对象的empno为默认值0，第二条输出empno的值由序列赋值，成功获得序列返回值



## 输入映射parameterType

parameterType指定了当前sql语句执行的时候传入的参数Java类型，无论指定什么类型，只要保证在

使用的时候传入的实际参数类型和映射文件中声明的类型一致即可。

### 简单类型-单个参数

简单类型是指传入的参数是Mybatis本身已经支持的别名所对应的类型，在Mybatis的入门程序和Mapper代理的开发方式中，已经多次使用多。

例如

<select id=*"findEmployeeById"* parameterType=*"int"*

resultType=*"com.xiaoka.bean.Employee"*>

select \* from emp where empno = #{id}

</select>

这里的parameterType="int"的int就是简单类型，使用的时候只要传入的参数是int类型即可。

@Test

**public** **void** testFindEmployeeById() {

...代码.. .

//输入映射类型为简单类型int，所以使用的时候传入int类型的值

Employee employee = employeeMapper.findEmployeeById(7521);

...代码.. .

}

### 简单类型-多个参数

任意多个参数，都会被MyBatis重新包装成一个Map传入。

Map的key是param1，param2，0，1…，值就是参数的值。

### POJO类型

POJO类型指出传入的参数是自定义的JavaBean类，在Mapper代理的开发方式中添加员工就是使用的该类型。

<!-- 输入映射类型为POJO类型 -->

<insert id=*"insertEmployee"* parameterType=*"com.xiaoka.bean.Employee"*>

insert into emp (empno,eName) values (#{empno},#{eName})

</insert>

@Test

**public** **void** testInsertEmployee2() {

...代码.. .

//输入映射类型为POJO类型Employee

employeeMapper.insertEmployee2(employee);

...代码.. .

}

### POJO包装类型

当单个的POJO类型无法满足需求的时候，可以考虑使用POJO包装类型

①：JavaBean

**public** **class** Department {

**private** **int** deptNo;

**private** String deptName;

**private** String Loc;

}

**public** **class** Employee {

**private** **int** empno;

**private** String eName;

}

**public** **class** Employee\_DeptVO {

**private** Employee employee;

**private** Department department;

}

②：映射文件employeeMapper.xml

<!-- 根据姓名模糊查询，查询同一部门的员工 -->

<!-- 这里使用了POJO包装类型，因为单给POJO对象里边的字段信息不够 -->

<!-- 使用OGNL表达式获取属性值 -->

<select id=*"selectEmployeeInfo"* resultType=*"com.xiaoka.bean.Employee"*

parameterType=*"com.xiaoka.bean.Employee\_DeptVO"*>

SELECT e.EMPNO,e.ENAME,d.DEPTNO,d.DNAME,d.LOC FROM emp e,dept d where

e.ENAME LIKE '%${employee.eName}%' and

d.deptno=#{department.deptNo} AND e.deptno = d.deptno

</select>

③：EmployeeMapper接口

**public** **interface** EmployeeMapper {

//使用POJO包装类型

**public** List<Employee\_DeptVO> selectEmployeeInfo(Employee\_DeptVO empDeptVO);

}

④：测试

@Test

**public** **void** testSelectEmployeeInfo(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

Employee employee = **new** Employee(7369, "M");

Department department = **new** Department();

department.setDeptNo(30);

Employee\_DeptVO deptVO = **new** Employee\_DeptVO();

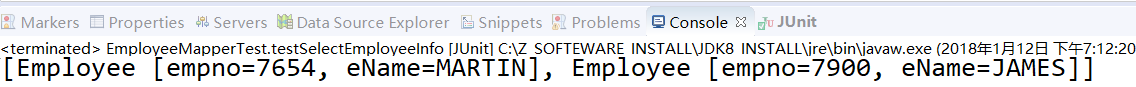
deptVO.setEmployee(employee);

deptVO.setDepartment(department);

List<Employee\_DeptVO> list = employeeMapper.selectEmployeeInfo(deptVO);

System.***out***.println(list);

}



### Map集合类型[了解]

该类型只做了解

同传递POJO一样，只不过Map集合的Key就相当于POJO的属性名称

①：映射文件employeeMapper.xml

<!-- 输入映射为Map集合类型 -->

<select id=*"selectEmployeeByMapCollection"* parameterType=*"java.util.HashMap"* resultType=*"employee"*>

SELECT \* FROM EMP WHERE empno = #{empno}

</select>

②：EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** Employee selectEmployeeByMapCollection(Map<String,String> map);

}

③：测试

@Test

**public** **void** testSelectEmployeeByMapCollection(){

Map<String, String> map = **new** HashMap<>();

map.put("empno", "7654");

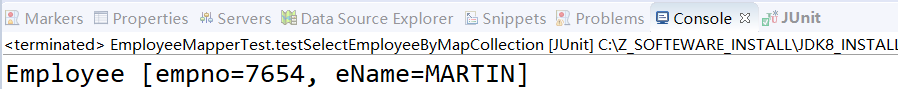
SqlSession session = sessionFactory.openSession();

EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.**class**);

Employee employee= employeeMapper.selectEmployeeByMapCollection(map);

System.***out***.println(employee);

}



异常测试：

传递的map中的key和sql中解析的key不一致。

测试结果没有报错，只是通过key获取值为空。

## 输出映射resultType

resultType指定了sql语句执行结果的返回值类型

**使用要求**

* 使用resultType进行结果映射时，需要查询出的列名和映射的对象的属性名一致，才能映射成功。
* 如果查询的列名和对象的属性名全部不一致，那么映射的对象为空。
* 如果查询的列名和对象的属性名有一个一致，那么映射的对象不为空，但是只有映射正确那一个属性才有值。
* 如果查询的sql的列名有别名，那么这个别名就是和属性映射的列名。

### 简单类型

注意，对简单类型的结果映射是有要求的，查询的列必须是一列，才能映射为简单类型。

①:映射文件employeeMapper.xml

<!-- 根据用户名模糊查询，看总共有多少条记录 -->

<select id=*"countByEname"* parameterType=*"string"* resultType=*"int"*>

SELECT COUNT(\*) FROM EMP WHERE ename LIKE '%${value}%'

</select>

②：Mapper接口

**public** **interface** EmployeeMapper {

**public** **int** countByEname(String eName);

}

③：测试代码

@Test

**public** **void** testCountByEname() {

SqlSession session = sessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

**int** count = mapper.countByEname("M");

System.***out***.println(count);

}

### POJO类型和POJO列表

在入门程序简单查询-返回一条记录以及模糊查询-返回多条记录中就已经使用了POJO和POJO列表

①：映射文件employeeMapper.xml

<!—- 输出映射resultType为POJO -->

<select id=*"findEmployeeById"* parameterType=*"int"* resultType=*"employee"*>

select \* from emp where empno = #{id}

</select>

<!—- 输出映射resultType为POJO列表 -->

<select id=*"findEmployeeByName"* parameterType=*"string"* resultType=*"employee"*>

SELECT \* FROM EMP WHERE ENAME LIKE '%${value}%'

</select>

②：Mapper接口

**public** **interface** EmployeeMapper {

**public** Employee findEmployeeById(**int** empno);

**public** List<Employee> findEmployeeByName(String eName);

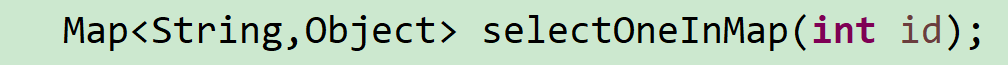
}

③：测试代码

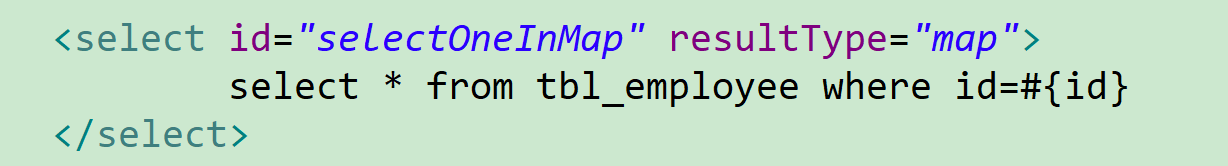
略

### 返回map集合

接口添加方法selectOneInMap()



mapper中添加select节点



## 输出映射resultMap

使用场景：当查询的列名和JavaBean的属性名称不一致，【例如查询有别名】，这个时候可以考虑使用resultMap来映射输出。

①：在映射文件employeeMapper.xml中声明resultMap，定义查询列名和输出的映射

<!-- 声明resultMap -->

<!-- type:定义使用的POJO的类型 -->

<!-- id：为当前resultMap定义唯一标识符 -->

<resultMap type=*"com.xiaoka.bean.Employee"* id=*"empRstMap"*>

<!-- id:映射唯一列[主键] -->

<!-- result:定义查询列到pojo属性的映射 -->

<id column=*"编号"* property=*"empno"*/>

<result column=*"姓名"* property=*"eName"*/>

</resultMap>

<select id=*"findEmpRstMap"* parameterType=*"int"* resultMap=*"empRstMap"*>

SELECT empno 编号,ename 姓名 FROM EMP WHERE empno = #{id}

</select>

②：Mapper接口

**public** **interface** EmployeeMapper {

**public** Employee findEmpRstMap(**int** empno);

}

③：测试代码

@Test

**public** **void** testFindEmpRstMap() {

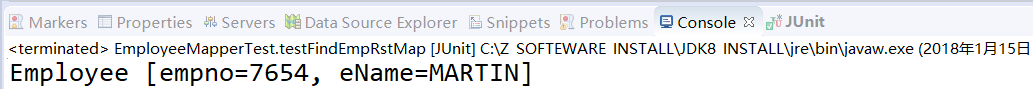
SqlSession session = sessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

Employee employee = mapper.findEmpRstMap(7654);

System.***out***.println(employee);

}



## Select元素总结



# 动态SQL

mybatis提供了一些动态sql标签，可以让程序员更快的进行mybatis的开发，这些动态sql可以通过sql的可重用性。

常用的动态sql标签：if标签、where标签、sql片段、foreach标签

## if标签

if标签用来进行条件判断，进行动态sql的拼接

**①：映射文件employeeMapper.xml**

<select id=*"testIf"* parameterType=*"employee"* resultType=*"employee"*>

<!--这里的WHERE后边的1=1必须有，否则拼接的SQL语句格式错误 -->

SELECT \* FROM emp WHERE 1=1

<if test=*"empno != null"*>

and empno = #{empno}

</if>

<if test=*"eName != null and eName !='' "*>

and ENAME LIKE '%${eName}%'

</if>

</select>

**②：接口EmployeeMapper.java**

**public** **interface** EmployeeMapper {

**public** List<Employee> testIf(Employee employee);

}

**③：测试**

@Test

**public** **void** testTestIf(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper userMapper = session.getMapper(EmployeeMapper.**class**);

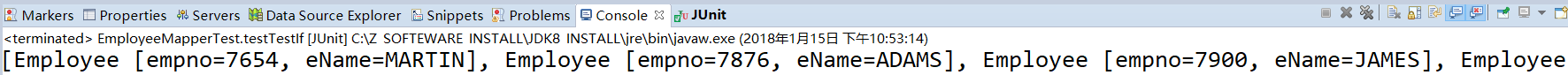
Employee user = **new** Employee();

user.seteName("M");

List<Employee> list = userMapper.testIf(user);

System.***out***.println(list);

}



## where标签

Where标签可以自动处理第一个”AND”,那么就可以省略掉WHERE后边的1=1。

**①：映射文件employeeMapper.xml**

<select id=*"testWhere"* parameterType=*"employee"* resultType=*"employee"*>

SELECT \* FROM emp

<where>

<if test=*"empno !=null"*>

AND empno = #{empno}

</if>

<if test=*"eName != null and eName !=''"*>

AND eName LIKE '%${eName}%'

</if>

</where>

</select>

**②：接口EmployeeMapper.java**

**public** **interface** EmployeeMapper {

**public** List<Employee> testWhere(Employee user);

}

**③：测试**

@Test

**public** **void** testTestWhere(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper userMapper = session.getMapper(EmployeeMapper.**class**);

Employee employee = **new** Employee();

employee.seteName("M");

List<Employee> list = userMapper.testWhere(employee);

System.***out***.println(list);

}

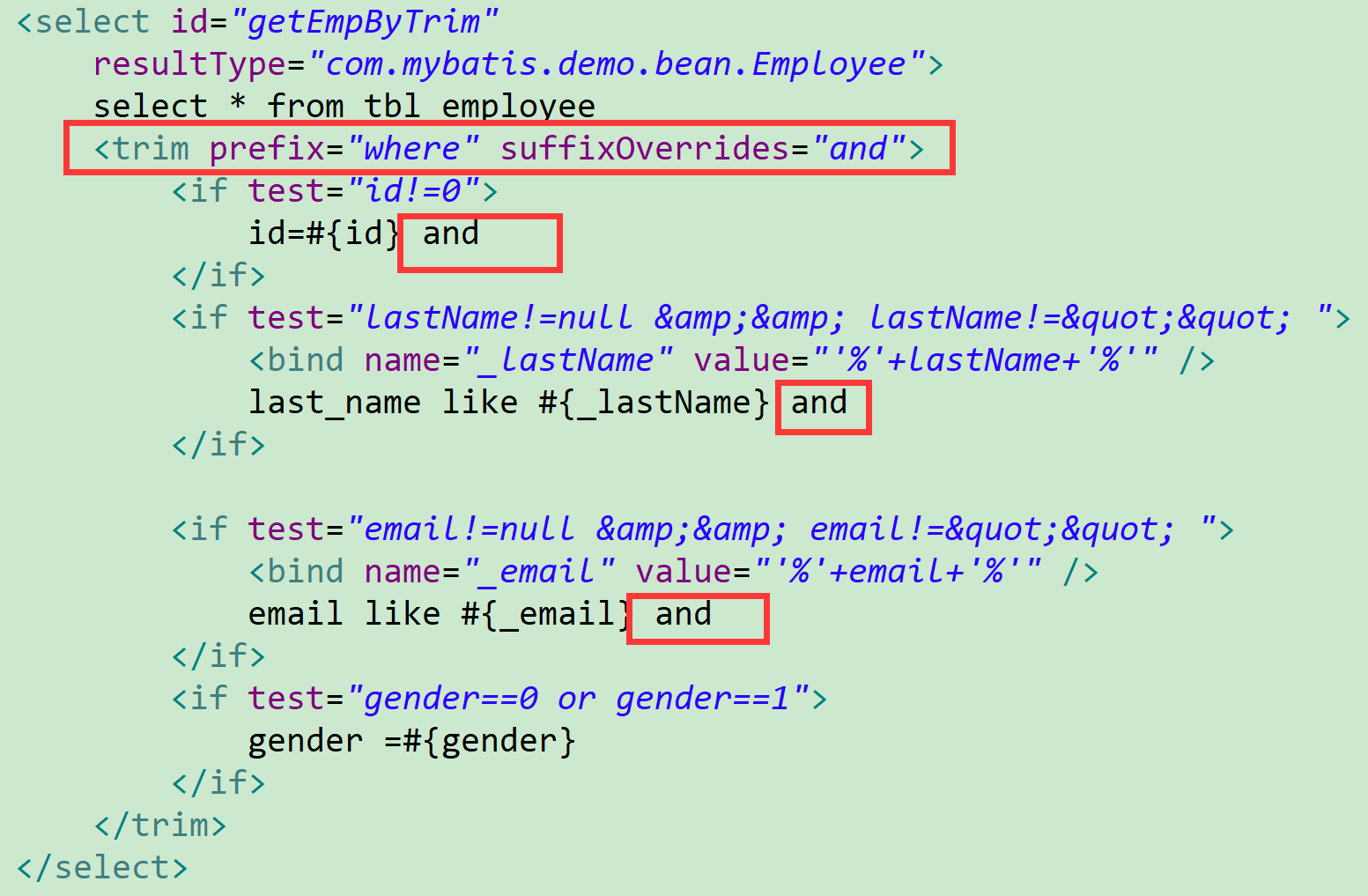
## trim标签

<where>仅能解决句首的问题，and若出现在句尾则会报错。解决方案：使用<trim>

Trim标签的属性

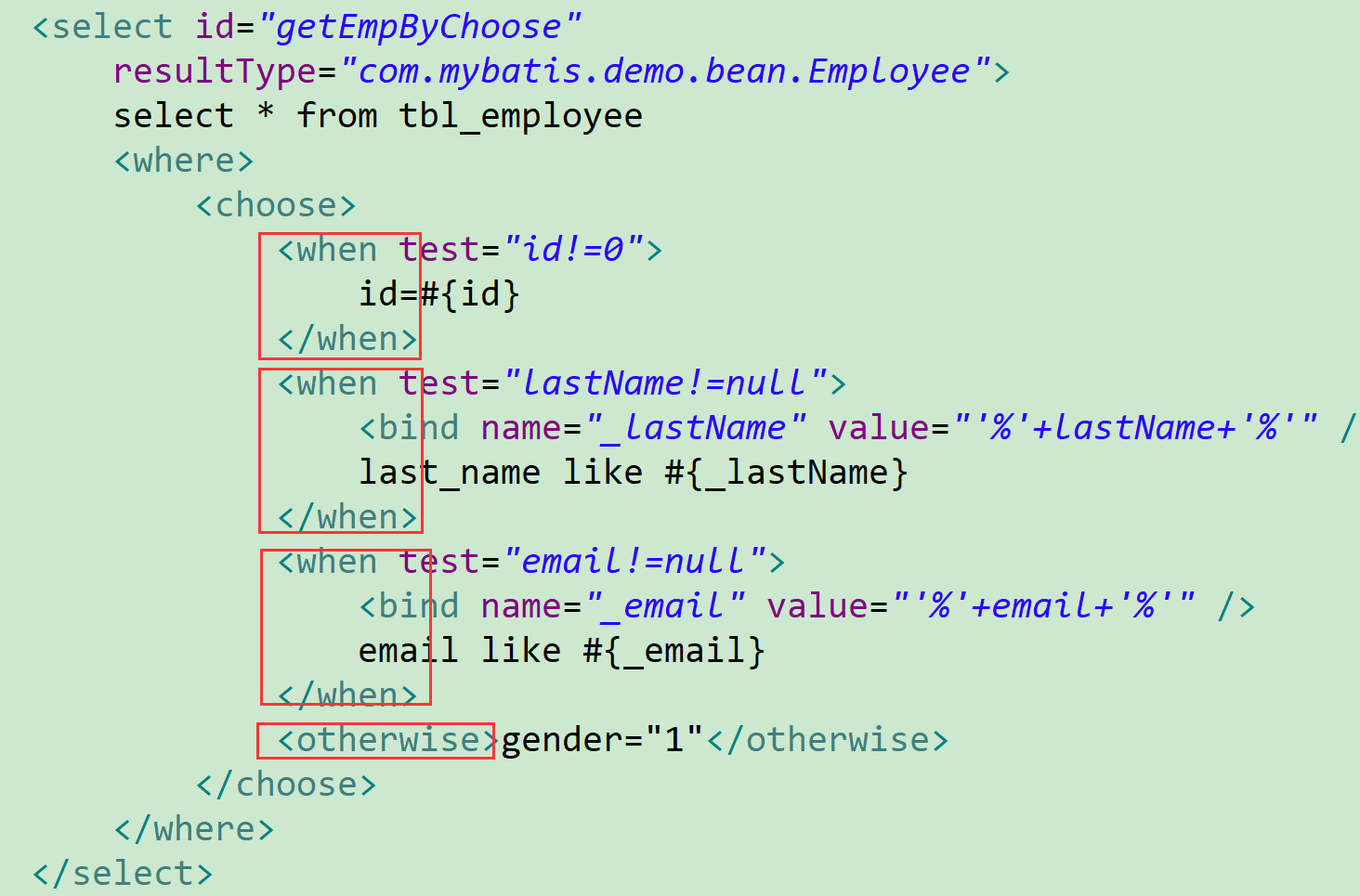
|  |  |
| --- | --- |
| **属性** | **描述** |
| prefix | 给 sql 语句拼接的前缀 |
| suffix | 给 sql 语句拼接的后缀 |
| prefixOverrides | 指定去除多余的前缀内容 |
| suffixOverrides | 指定去除多余的后缀内容 |

示例代码



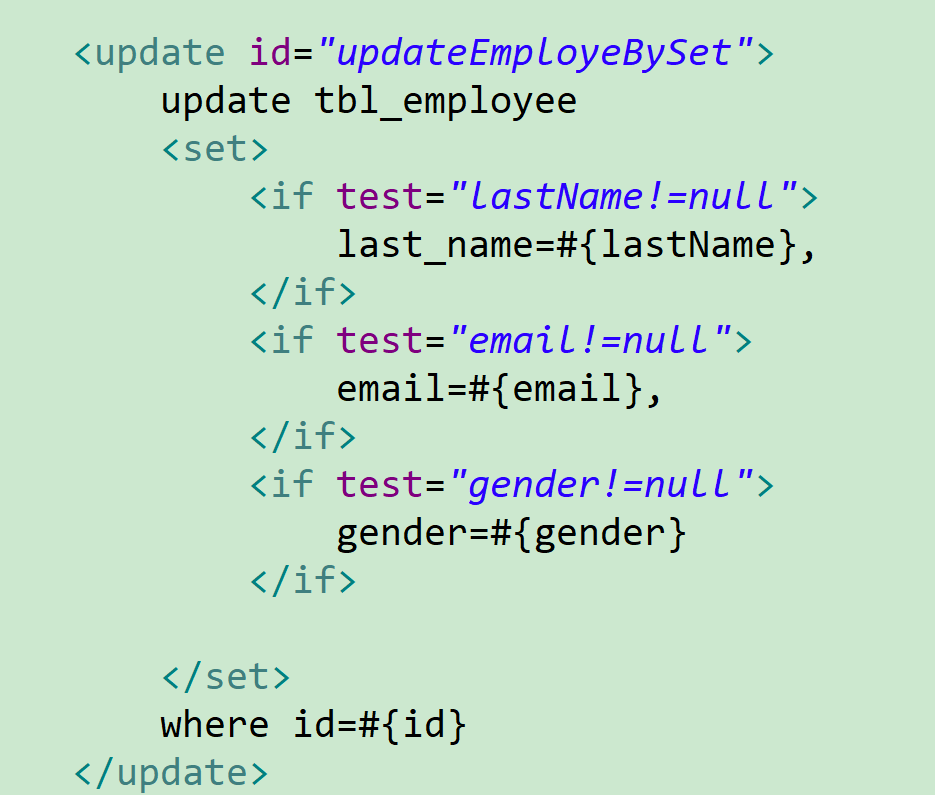
## choose标签

多查询条件，N选一



## set标签

该标签用于按字段更新，使用set标签，可以自动去除句尾的逗号，使用trim标签也可以达到效果。



## foreach标签

foreach的主要用在构建in条件中，它可以在SQL语句中进行迭代一个集合。

### 使用POJO的List属性作为参数

①：JavaBean

**public** **class** EmployeeVo {

**private** List<Integer> empNos;

}

②：映射文件employeeMapper.xml

<!--

collection:需要迭代的集合

item表示集合中每一个元素进行迭代时的别名，

index指 定一个名字，用于表示在迭代过程中，每次迭代到的位置，

open表示该语句以什么开始，

separator表示在每次进行迭代之间以什么符号作为分隔 符，

close表示以什么结束。

-->

<select id=*"testListEmpNos"* parameterType=*"employeeVo"* resultType=*"employee"*>

SELECT \* FROM emp

<where>

<if test=*"empNos !=null and empNos.size>0"*>

<foreach collection=*"empNos"* open=*"and empNo in("* close=*")"*

item=*"id"* separator=*","*>

#{id}

</foreach>

</if>

</where>

</select>

③：EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** List<Employee> testListEmpNos(EmployeeVo employeeVo);

}

④：测试代码

@Test

**public** **void** testTestListEmpNos(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

EmployeeVo vo = **new** EmployeeVo();

ArrayList<Integer> list = **new** ArrayList<>();

list.add(7654);

list.add(7876);

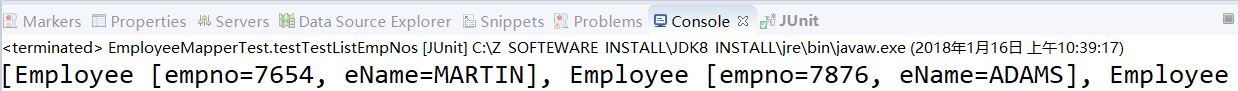
list.add(7900);

vo.setEmpNos(list);

List<Employee> listEmpNos = mapper.testListEmpNos(vo);

System.***out***.println(listEmpNos);

}



### 使用List集合作为参数

①：employeeMapper.xml

<select id=*"testListForeach"* parameterType=*"java.util.List"*

resultType=*"employee"*>

SELECT \* FROM EMP

<where>

<!-- 使用List集合作为输入映射的参数，那么对象名称就只能是list -->

<if test=*"list !=null and list.size>0"*>

<foreach collection=*"list"* open=*"and empno in ("* close=*")"*

separator=*","* item=*"id"*>

#{id}

</foreach>

</if>

</where>

</select>

②：EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** List<Employee> testListForeach(List<Integer> list);

}

③：测试代码

@Test

**public** **void** testTestListForeach(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

ArrayList<Integer> list = **new** ArrayList<>();

list.add(7902);

list.add(7876);

list.add(7900);

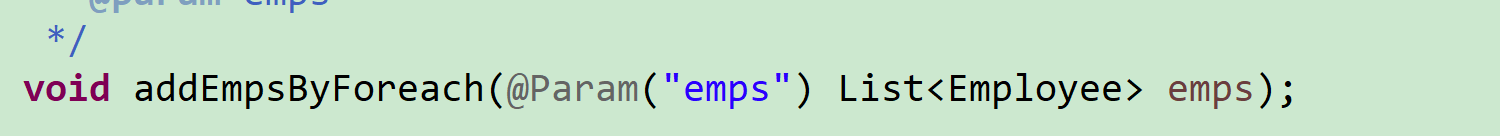
List<Employee> listForeach = mapper.testListForeach(list);

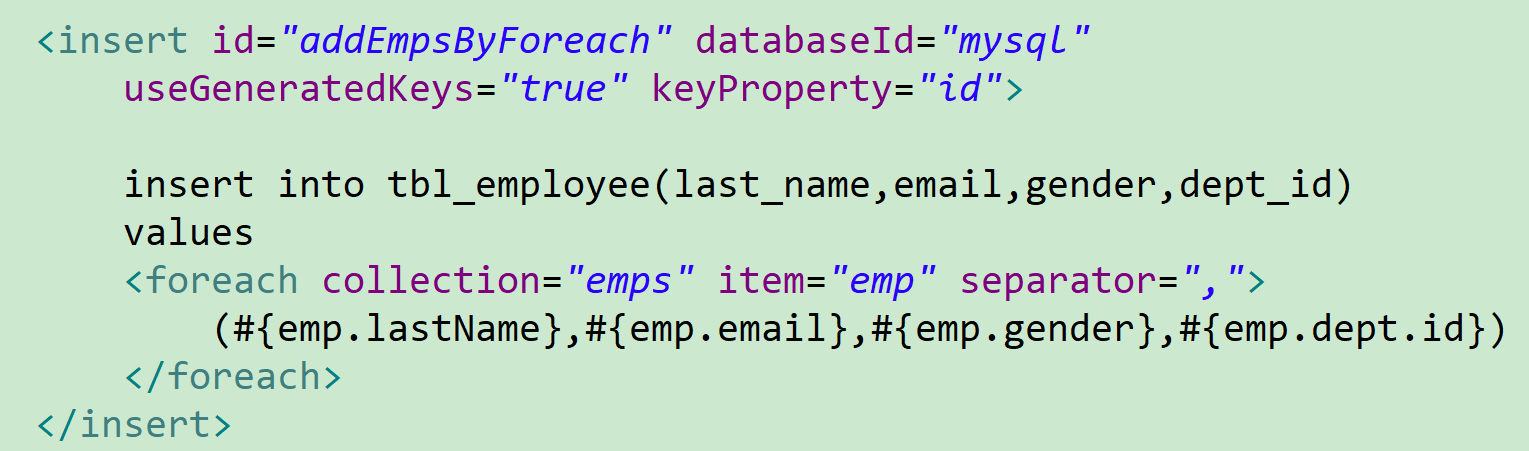
System.***out***.println(listForeach);

}

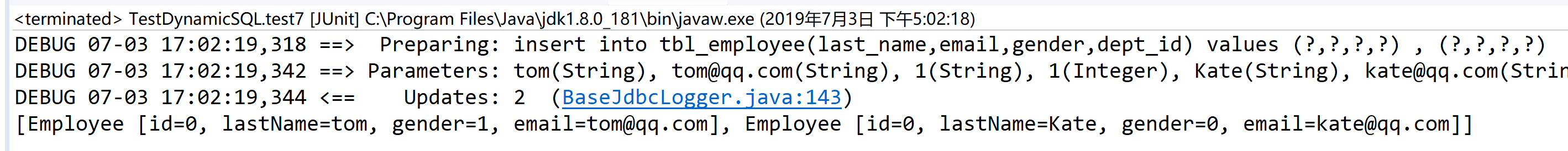


### 使用foreach批量添加数据









## sql片段

sql中可将重复的sql提取出来，使用include引用，达到sql重用的目的。sql片段中可以写sql任何部分

①：映射文件employeeMapper.xml

<!-- 定义sql片段 -->

<sql id=*"ifConditions"*>

<if test=*"empNos !=null and empNos.size>0"*>

<foreach collection=*"empNos"* open=*"and empNo in("* close=*")"*

item=*"id"* separator=*","*>

#{id}

</foreach>

</if>

</sql>

<select id=*"testListEmpNos"* parameterType=*"employeeVo"*

resultType=*"employee"*>

SELECT \* FROM emp

<where>

<!-- 引用sql片段 -->

<include refid=*"ifConditions"*/>

</where>

</select>

②：测试代码

效果同foreach标签中一样，不再赘述。注意，sql片段中同样可以写where等标签，建议sql片段中尽量只有if判断条件。

# 多表查询

凡是多表查询首要任务是理解表与表之间的关系，单表之间关键列的信息。

多表查询一般分为一对多，一对一，多对多，多对一

一对多：A表的一条记录可以匹配B表的多条记录。

多对一：从B表的角度来看，就是多对一。也就是说B表的多条记录可以匹配A表的一条记录

一对一：A表的一条记录只能匹配B表的一条记录。其实一对一就只是一对多的特例而已。

多对多：A表的一条记录可以匹配B表的多条记录，B表的一条记录也可以匹配A表的多条记录。这种关系一般要借助第三个表将A表和B表“结合”起来。

## 一对一

### resultType

#### 需求：查询员工编号，姓名，工作，以及部门名称

#### sql语句：

select e.empno,e.ename,e.job,d.dname from emp e,dept d where e.deptno = d.deptno;

#### JavaBean类Employee

**public** **class** Employee {

**private** Integer empno;

**private** String eName;

**private** String job;

}

#### 创建EmployeeExt扩展类

**public** **class** EmployeeExt **extends** Employee {

//sql语句中有几个字段不属于EMP表，就创建几个同名成员属性在扩展类中

**private** String dName;

}

#### 映射文件employeeMapper.xml

<select id=*"selectEmployeeAndDept"* resultType=*"com.xiaoka.bean.EmployeeExt"*>

select e.empno,e.ename,e.job,d.dname from emp e,dept d where e.deptno = d.deptno

</select>

#### EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** List<EmployeeExt>selectEmployeeAndDept();

}

#### 测试代码

@Test

**public** **void** testSelectEmployeeAndDept(){

SqlSession session = sessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

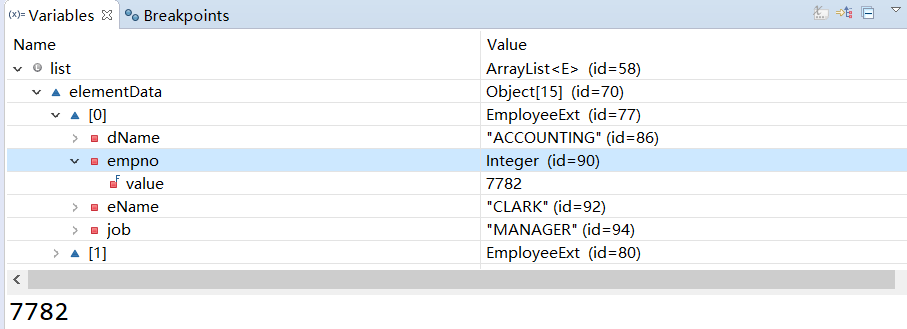
List<EmployeeExt> list = mapper.selectEmployeeAndDept();

session.close();

System.***out***.println(list);

}

通过Debug视图可以发现，凡是查询的列都已经赋值



#### 分析

为什么创建EmployeeExt扩展类，而不是在Employee中增加属性String dName?

①：Employee类本身是为了对应数据库表EMP表，这就要求Employee类的成员属性和EMP表字段一致。

②：更多的情况下，Employee类是由工具根据表自动生成的，如果在已经生成完的Employee类中添加其它属性，那么当重新由工具生成Employee类的时候，将导致所有后添加的属性被清除，直接导致查询的数据有误。

③：使用继承的方式创建EmployeeExt类，那么无论更新EMP表和Employee类，都将从最小程度上影响到本次的查询数据。

④：在输入章节中曾讲到创建POJO的包装类，从功能上的确没问题。然而，包装类应用的是输入映射，扩展类才是映射的输出映射，故创建扩展类，不创建包装类。

#### 小结

使用resultType来进行一对一结果映射，查询出的列的个数和映射的属性的个数要一致。而且映射的属性要存在与一个大的对象中，它是一种平铺式的映射，即数据库查询出多少条记录，则映射成多少个对象

### resultMap

使用resultMap来进行一对一结果映射，它是将关联对象添加到主信息的对象中，具体说是对象嵌套对象的一种映射方式。

#### 需求：查询员工编号，姓名,以及部门信息

#### sql语句

select e.empno,e.eName,e.job,d.deptno,d.dname,d.loc from emp e,dept d where e.deptno = d.deptno

#### JavaBean类

**public** **class** Department {

**private** **int** deptNo;

**private** String deptName;

**private** String Loc;

}

**public** **class** EmployeeAndDept **extends** Employee {

**private** Department department;

}

#### 映射文件employeeMapper.xml

<resultMap type=*"com.xiaoka.bean.EmployeeAndDept"* id=*"resuleMapEmpDept"*>

<id column=*"empno"* property=*"empno"* />

<result column=*"ename"* property=*"eName"* />

<result column=*"job"* property=*"job"* />

<!--association 被关联的属性 -->

<association property=*"department"* javaType=*"com.xiaoka.bean.Department"*>

<!-- 被关联属性类的id，用不用都要写，否则影响性能 -->

<id property=*"deptNo"* column=*"deptno"* />

<result property=*"deptName"* column=*"dName"* />

<result property=*"Loc"* column=*"Loc"* />

</association>

</resultMap>

<select id=*"selectEmployeeAndDeptReMap"* resultMap=*"resuleMapEmpDept"*>

select e.empno,e.eName,e.job,d.deptno,d.dname,d.loc from emp e,dept d where e.deptno = d.deptno

</select>

#### EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** List<EmployeeAndDept>selectEmployeeAndDeptReMap();

}

#### 测试代码

@Test

**public** **void** testTelectEmployeeAndDeptReMap(){

SqlSession session = sessionFactory.openSession();

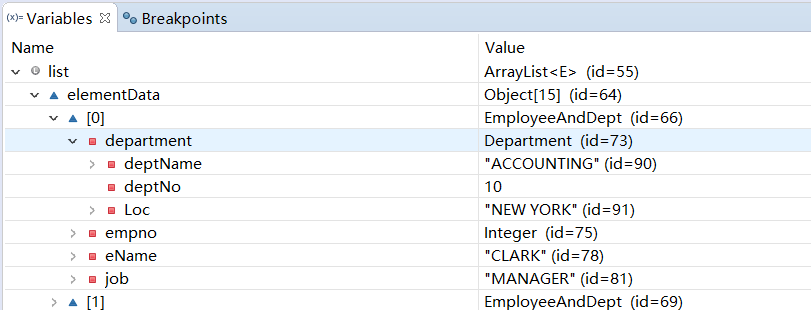
EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

List<EmployeeAndDept> list = mapper.selectEmployeeAndDeptReMap();

session.close();

System.***out***.println(list);

}



#### 小结

在一对一结果映射时，使用resultType更加简单方便，如果有特殊要求（对象嵌套对象）时，需要使用resultMap进行映射，比如：查询订单列表，然后在点击列表中的查看订单明细按钮，这个时候就需要使用resultMap进行结果映射。而resultType更适应于查询明细信息，比如，查询订单明细列表。

## 一对多

### 需求：查询部门信息，关联查询员工

### sql语句

select d.deptno,d.dname,e.ename from dept d ,emp e where d.deptno = e.deptno;

### 创建DepartmentExt类

**public** **class** DepartmentExt **extends** Department {

**private** List<Employee> employeeList;

}

### 映射文件employeeMapper.xml

<resultMap type=*"com.xiaoka.bean.DepartmentExt"* id=*"deptAndEmpResMap"*>

<id property=*"deptNo"* column=*"deptno"* />

<result property=*"deptName"* column=*"dName"* />

<result property=*"Loc"* column=*"Loc"* />

<!-- collection:用于映射一对多中的集合属性 -->

<!-- ofType：集合属性的泛型 -->

<collection property=*"employeeList"* javaType=*"ArrayList"* ofType=*"com.xiaoka.bean.Employee"*>

<id column=*"empno"* property=*"empno"* />

<result column=*"ename"* property=*"eName"* />

<result column=*"job"* property=*"job"* />

</collection>

</resultMap>

<select id=*"selectDeptandEmployee"* resultMap=*"deptAndEmpResMap"*>

select d.deptno,d.dname,e.ename from dept d ,emp e where d.deptno = e.deptno

</select>

### EmployeeMapper接口

**public** **interface** EmployeeMapper {

**public** List<DepartmentExt>selectDeptandEmployee();

}

### 测试代码

@Test

**public** **void** testSelectDeptandEmployee() {

SqlSession session = sessionFactory.openSession();

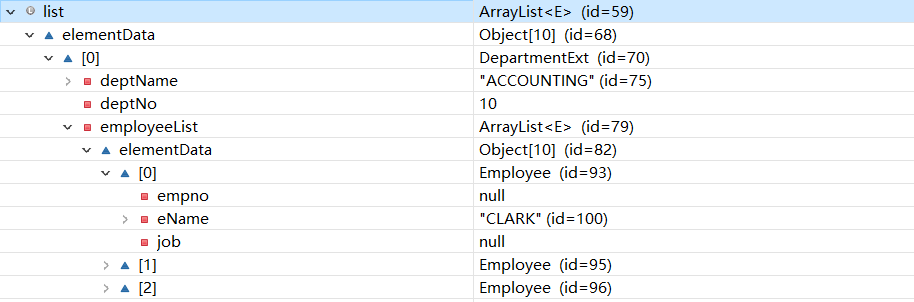
EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

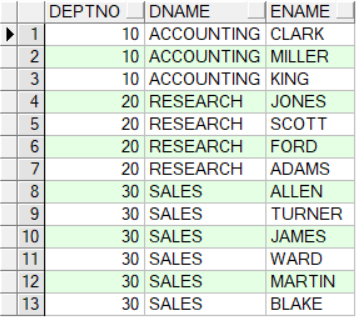
List<DepartmentExt> list = mapper.selectDeptandEmployee();

session.close();

System.***out***.println(list);

}





### 小结

在一对多中【一个部门有多个员工】，如果使用resultType那么查询的结果list的size将是13。而使用resultMap，那么查询结果将是list的size是3，具体每个部门下的多个员工则被封装到了employeeList集合中。通过debug视图【上图】则可以清晰的查看到数据的封装层次。此即为一对多。

## 多对多

在Oracle的初始学习表中目前没有多对多的关系，所以在学习多对多之前，我们首先来添加所需要的表和数据。

### 分析

目前已经有的表是EMP表，而员工和办公用品之间则是多对多的关系。即一个办公用品可以属于多个员工，一个员工也可以拥有多个办公用品。所以我们需要创建办公用品表和将EMP表和办公用品表结合起来的第三个表。

### 数据准备

*--创建办公用品表*

create table articles(

id number primary key,

goodsName varchar2(20) not null,

color varchar2(10),

price number(7,2)

);

*--添加办公用品数据*

insert into articles values(1001,'笔记本电脑','黑',7777.77);

insert into articles values(1002,'文件夹','蓝',10.7);

insert into articles values(1003,'台式电脑','黑',5577.77);

insert into articles values(1007,'投影仪','白',9007.77);

*--创建员工表emp和办公用品表结合的第三个表*

create table emp\_and\_article(

empno number,

article\_id number,

constraints fk\_items\_emp\_empno foreign key (empno) references emp(empno),

constraints fk\_items\_article\_articleId foreign key (article\_id) references articles(id)

);

*--给“结合”表中添加数据。*

insert into emp\_and\_article values(7654,1001);

insert into emp\_and\_article values(7654,1002);

insert into emp\_and\_article values(7654,1003);

insert into emp\_and\_article values(7566,1002);

### 需求：查询7654员工领用的办公用品

### sql语句

SELECT e.ename ,a.goodsname

FROM emp e,articles a,emp\_and\_article ea

where e.empno = 7654

and e.empno = ea.empno

and a.id = ea.article\_id;

### JavaBean

**public** **class** Articles {

**private** Integer id;

**private** String goodsName;

**private** String color;

**private** String price;

}

**public** **class** EmployeeExt **extends** Employee {

**private** List<Articles> articlesList;

}

### Mapper接口

**public** **interface** EmployeeMapper {

**public** EmployeeExt selectEmpAndArticlesByEmpno(**int** empno);

}

### Mapper配置文件

**employeeMapper.xml**

<resultMap type=*"com.xiaoka.bean.EmployeeExt"* id=*"empAndArticlesResmapManyToMany"*>

<id column=*"empno"* property=*"empno"* />

<result column=*"eName"* property=*"eName"* />

<result column=*"job"* property=*"job"* />

<collection property=*"articlesList"* ofType=*"com.xiaoka.bean.Articles"*>

<id column=*"id"* property=*"id"* />

<result column=*"goodsName"* property=*"goodsName"* />

<result column=*"color"* property=*"color"* />

<result column=*"price"* property=*"price"* />

</collection>

</resultMap>

<select id=*"selectEmpAndArticlesByEmpnoManyToMany"*

parameterType=*"int"* resultMap=*"empAndArticlesResmapManyToMany"*>

SELECT e.ename ,a.goodsname FROM emp

e,articles a,emp\_and\_article ea where

e.empno = #{empno} and e.empno =

ea.empno and a.id = ea.article\_id

</select>

### 测试代码

@Test

**public** **void** testSelectEmpAndArticlesByEmpnoManyToMany() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

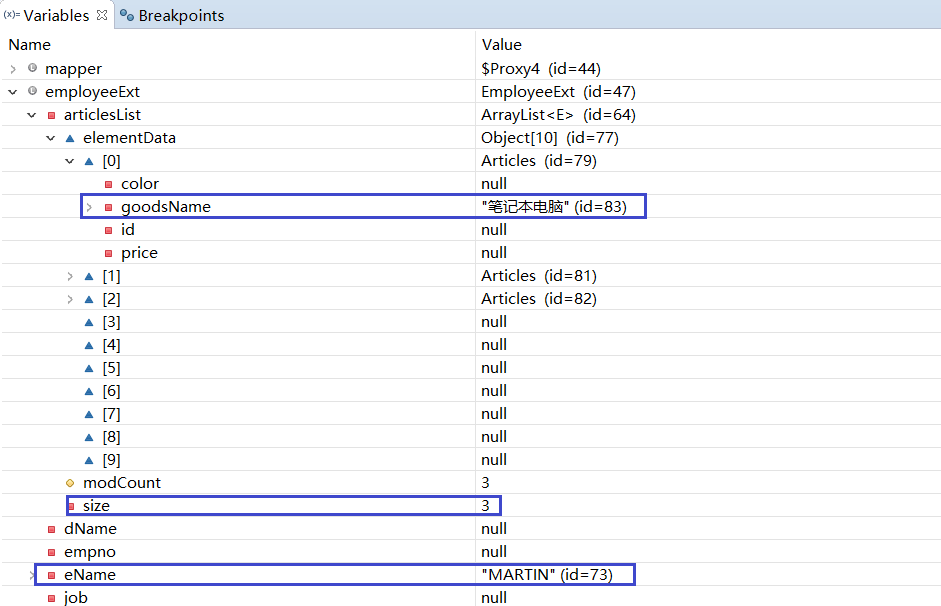
SqlSession session = sqlSessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

EmployeeExt employeeExt = mapper.selectEmpAndArticlesByEmpnoManyToMany(7654);

System.***out***.println(employeeExt);

}



可以看到，数据被正确的查询出来了。

注意 ： 在多对多和一对多的查询中，务必查询主键列，否则数据是正常的，但是数据结构不正常。

# 延迟加载

在上一章节中，我们学习了多表查询，无论是一对一，一对多还是多对多，其实代码都是存在一定问题的。

一对一中，我们查询的是：查询员工编号，姓名,以及部门信息【resultMap】

一对多中，我们查询的是：查询部门信息，关联查询员工【resultMap】

多对多中，我们查询的是：查询7654员工领用的办公用品 【resultMap】

所有这些多表查询中，resultMap里边的数据无论用户是否查看，Mybatis都会积极的将这些数据给查询出来，放到内存中。

例如：一对一中，用户可能只查看员工编号和姓名，而封装在resulMap中的部门信息却不一定会查看。

一对多中，用户可能只查看部门信息，不一定查看该部门下的员工信息。此即为问题所在。

**延迟加载又叫懒加载**，也叫按需加载。也就是说先加载主信息，在需要的时候，再去加载从信息。

在mybatis中，resultMap标签 的association标签和collection标签具有延迟加载的功能。

## 需求：查询7654员工领用的办公用品

## 目标功能

本例中我们用多对多来作为案例。想要实现的功能是查询7654号员工，如果要查看7654号领用过的办公用品，则去数据库查询该数据封装到集合中。否则，该集合为null。此即为懒加载的按需加载。

## sql语句

*--从结合表中查询7654领用过办公用品id号*

select article\_id from emp\_and\_article where empno in (select empno from emp where empno = 7654);

*--从办公用品表中查询*

select \* from articles where id in( select article\_id from emp\_and\_article where empno in (select empno from emp where empno = 7566))

## JavaBean

**public** **class** Articles {

**private** Integer id;

**private** String goodsName;

**private** String color;

**private** String price;

}

**public** **class** EmployeeExt **extends** Employee {

**private** List<Articles> articlesList;

}

## Mapper接口

**public** **interface** ArticlesMapper {

**public** List<Articles> selectArticlesByEmpno(**int** id);

}

**public** **interface** EmployeeMapper {

**public** EmployeeExt selectEmpAndArticlesByEmpno(**int** empno);

}

## Mapper配置文件

**articlesMapper.xml**

<select id=*"selectArticlesByEmpno"* parameterType=*"int"* resultType=*"com.xiaoka.bean.Articles"*>

select \* from articles where id in (select article\_id from emp\_and\_article where empno = #{empno})

</select>

**employeeMapper.xml**

<resultMap type=*"com.xiaoka.bean.EmployeeExt"* id=*"empAndArticlesResmap"*>

<id column=*"empno"* property=*"empno"* />

<result column=*"eName"* property=*"eName"* />

<result column=*"job"* property=*"job"* />

<!-- column:将查询结果的empno列值作为参数传入select属性绑定的查询语句 -->

<!—如果select指定的statement需要多个参数，需要在column中定义对应关系即：column={column1=pro1,column2=pro2}-->

<!-- select:绑定查询语句，返回结果赋值给collection标签映射的属性 -->

<collection property=*"articlesList"*

ofType=*"com.xiaoka.bean.Articles"*

column=*"empno"*

select=*"com.xiaoka.mapper.ArticlesMapper.selectArticlesByEmpno"*>

<id column=*"id"* property=*"id"*/>

<result column=*"goodsName"* property=*"goodsName"*/>

<result column=*"color"* property=*"color"*/>

<result column=*"price"* property=*"price"*/>

</collection>

</resultMap>

<select id=*"selectEmpAndArticlesByEmpno"* resultMap=*"empAndArticlesResmap"*>

select empno,ename,job from emp where empno = #{empno}

</select>

## 测试代码

@Test

**public** **void** testselectEmpAndArticlesByEmpno() **throws** Exception {

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

EmployeeExt employeeExt = mapper.selectEmpAndArticlesByEmpno(7654);

System.***out***.println(employeeExt);

}

## 测试结果

16:41:52,906 DEBUG JdbcTransaction:98 - Setting autocommit to false on JDBC Connection [oracle.jdbc.driver.T4CConnection@35a50a4c]

16:41:52,909 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Preparing: select empno,ename,job from emp where empno = ?

16:41:53,082 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Parameters: 7654(Integer)

16:41:53,151 DEBUG selectArticlesByEmpno:139 - ====> Preparing: select \* from articles where id in (select article\_id from emp\_and\_article where empno = ?)

16:41:53,151 DEBUG selectArticlesByEmpno:139 - ====> Parameters: 7654(Integer)

16:41:53,170 DEBUG selectArticlesByEmpno:139 - <==== Total: 3

16:41:53,174 DEBUG selectEmpAndArticlesByEmpno:139 - <== Total: 1

EmployeeExt [articlesList=[Articles [id=1001, goodsName=笔记本电脑, color=黑, price=7777.77], Articles [id=1002, goodsName=文件夹, color=蓝, price=10.7], Articles [id=1003, goodsName=台式电脑, color=黑, price=5577.77]], dName=null]

查看控制台，级联查询成功。数据已经正常拿到。但是这里有两个问题

第一：没有实现按需加载。即：明明没有查询集合的数据，但是集合中却已经有数据了。

第二：N+1现象。即：发出一次查询，但是Mybatis却发起了两次sql查询。第一次是查询员工，第二次是查询办公明细。因为select的嵌套会引起N+1现象，所以官方不推荐使用select的嵌套。

此两个问题可以使用延迟加载一次性全部解决。

## 使用延迟加载

使用延迟加载，需要在全局配置文件中和映射文件中分别进行设置。

### 全局配置文件

**sqlMapConfig.xml**

<!--要使延迟加载生效，必须配置下面两个属性 -->

<settings>

<!-- 延迟加载的全局开关。开启以后，所有关联对象都会延迟加载。默认为false -->

<setting name=*"lazyLoadingEnabled"* value=*"true"* />

<!-- 启动该属性，会使带有延迟加载属性的对象立即记载。反之，每种属性都会按需加载。默认为true,所以这里要设置成false -->

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

</settings>

### 映射文件employeeMaper.xml

修改resultMap如下：

<resultMap type=*"com.xiaoka.bean.EmployeeExt"* id=*"empAndArticlesResmap"*>

<id column=*"empno"* property=*"empno"* />

<result column=*"eName"* property=*"eName"* />

<result column=*"job"* property=*"job"* />

<!-- fetchType:该属性取值有eagger和lazy，eager表示立即加载，lazy表示懒加载 -->

<collection property=*"articlesList"* ofType=*"com.xiaoka.bean.Articles"*

column=*"empno"* select=*"com.xiaoka.mapper.ArticlesMapper.selectArticlesByEmpno"*

fetchType=*"lazy"*>

<id column=*"id"* property=*"id"* />

<result column=*"goodsName"* property=*"goodsName"* />

<result column=*"color"* property=*"color"* />

<result column=*"price"* property=*"price"* />

</collection>

</resultMap>

### 测试代码一

@Test

**public** **void** testselectEmpAndArticlesByEmpno() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

EmployeeMapper mapper = session.getMapper(EmployeeMapper.**class**);

EmployeeExt employeeExt = mapper.selectEmpAndArticlesByEmpno(7654);

}

### 控制台日志

21:08:21,213 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Preparing: select empno,ename,job from emp where empno = ?

21:08:21,315 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Parameters: 7654(Integer)

21:08:21,457 DEBUG selectEmpAndArticlesByEmpno:139 - <== Total: 1

可以看到，当没有要获取办公用品的集合的时候，并没有执行查询办公用品的sql语句。

### 测试代码二

在测试代码中加入获取办公用品集合的代码

List<Articles> list = employeeExt.getArticlesList();

System.***out***.println(list);

### 控制台日志

21:11:19,319 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Preparing: select empno,ename,job from emp where empno = ?

21:11:19,443 DEBUG selectEmpAndArticlesByEmpno:139 - ==> Parameters: 7654(Integer)

21:11:19,614 DEBUG selectEmpAndArticlesByEmpno:139 - <== Total: 1

21:11:19,619 DEBUG selectArticlesByEmpno:139 - ==> Preparing: select \* from articles where id in (select article\_id from emp\_and\_article where empno = ?)

21:11:19,620 DEBUG selectArticlesByEmpno:139 - ==> Parameters: 7654(Integer)

21:11:19,631 DEBUG selectArticlesByEmpno:139 - <== Total: 3

[Articles [id=1001, goodsName=笔记本电脑, color=黑, price=7777.77], Articles [id=1002, goodsName=文件夹, color=蓝, price=10.7], Articles [id=1003, goodsName=台式电脑, color=黑, price=5577.77]]

可以看到，当加入了获取集合的代码以后，Mybatis才去进行了关联查询。此即为按需加载，即懒加载。

## 小结

无论是一对一，一对多，还是多对多，只要使用了resultMap，那么就可以设置使用懒加载。而配置步骤也是十分的简单。只需要在全局配置文件中加入setting设置，在映射文件的resultMap中使用fetchType=lazy即可。

使用了懒加载以后，可以避免N+1影响的性能，也可以节省内存的开销。

# Mybatis的缓存机制

前面已经介绍了Mybatis的基本用法，关联映射和动态sql等重要知识点。本章我们将介绍Mybatis的缓存机制。合理的利用缓存可以加快数据的查询，进行有效的提升数据库的性能。

Mybatis的缓存分为一级缓存和二级缓存。一级缓存是SqlSession级别的缓存。二级缓存是mapper级别的。二级缓存是多个SqlSession共享的。

Mybatis通过缓存机制减轻数据压力，提高数据库性能。

## 一级缓存

一级缓存，也叫本地缓存是SqlSession级别的。

在操作数据库时需要构造SqlSession对象，该对象内有一个HashMap用于存储缓存数据。不同的SqlSession之间的缓存数据区域互不影响。

一级缓存的作用域是SqlSession范围的，当在同一个SqlSession中执行两次相同的sql语句时，第一次执行完毕会将数据库中查询的数据写到缓存【HashMap】中。第二次查询的时候，会从缓存中获取数据，不再去底层数据库查询，从而提高查询效率。

注意：

①：Mybatis的缓存机制是基于id进行缓存的。也就是说，SqlSession内部的HashMap使用对象的id作为key,而查询出来的数据对象作为value。

②：当SqlSession执行了DML操作，并提交到数据库，MyBatis则会清空一级缓存，避免出现脏读现象。

③：当一个SqlSession结束后，SqlSession中的一级缓存就会消失（会将该数据放入二级缓存，后续讲解）。

④：Mybatis默认开启一级缓存，不需要进行任何配置。

一级缓存失效

1：一级缓存是sqlSession级别的，所以当sqlSession不同时，当前sqlSession的一级缓存失效

2：sqlSession相同，查询条件不同（因为当前缓存中还没有这个数据）

3：sqlSession相同，查询条件相同，但两次查询之间进行了DML操作（DML操作可能会影响当前数据，所以一级缓存失效）

4：sqlSession相同，查询条件相同，但是手动清空了一级缓存。

### 测试代码

@Test

**public** **void** oneLevleCacheTest() **throws** IOException{

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

ArticleMapper mapper = session.getMapper(ArticleMapper.**class**);

List<Article> list1 = mapper.selectArticleByOrderId(1001);

List<Article> list2 = mapper.selectArticleByOrderId(1001);

}

### 控制台日志

14:56:54,365 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

14:56:54,454 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

14:56:54,505 DEBUG selectArticleByOrderId:139 - <== Total: 3

可以看到，发起了两次查询，但是仅仅第一次执行了sql语句，第二次则直接从缓存中获得数据

### 修改测试代码

在两次查询中间添加commit的代码，这个时候Mybatis就会清空一级缓存，所以会执行两条相同的sql语句。

List<Article> list1 = mapper.selectArticleByOrderId(1001);

session.commit();

List<Article> list2 = mapper.selectArticleByOrderId(1001);

### 控制台日志

15:00:47,943 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

15:00:48,029 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

15:00:48,077 DEBUG selectArticleByOrderId:139 - <== Total: 3

15:00:48,081 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

15:00:48,081 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

15:00:48,083 DEBUG selectArticleByOrderId:139 - <== Total: 3

### 小结

一级缓存自动开启，自动使用

当进行了commit操作就会情况一级缓存，跟是否做DML操作无关，只不过一般来讲DML操作总是伴随着commit动作而已。

一级缓存的作用域就是当前SqlSession对象。

## 二级缓存

二级缓存也叫全局缓存，它是基于namespace级别的缓存，一个namespace对应一个二级缓存。不同的namespace查询出的数据会放在自己对应的缓存中（每个namespace下都由一个map集合）。

二级缓存是mapper级别的。即多个SqlSession对象使用同一个Mapper的sql语句去操作数据库得到的数据会存在二级缓存中。它同样是HashMap进行存储。相比一级缓存，二级缓存的范围更大。多个SqlSession间可以共用二级缓存，二级缓存是跨SqlSession的。

二级缓存是多个SqlSession共享的。其作用域是mapper的同一个namespace。不同的SqlSession两次执行相同的namespace下的sql语句，且传递的参数也相同，即：最终执行的sql语句相同，则第一次执行完毕（关闭当前session以后）会将数据库中查询的数据写到缓存，第二次查询则从缓存中获取数据，不再去底层数据库查询，从而提高查询效率。

Mybatis默认开启二级缓存，但仍然需要在setting全局参数中配置开启二级缓存。

### 全局开启二级缓存

<settings>

<!-- 开启二级缓存,默认值也是true,但是要显式的配置出来-->

<!—如果是false，则是关闭二级缓存，一级缓存一直开启-->

<setting name=*"cacheEnabled"* value=*"true"*/>

</settings>

### 局部开启二级缓存

由于二级缓存是namespace级别的，所以需要去每个mapper.xml文件中配置使用二级缓存

开启当前mapper的namespace下的二级缓存

<!--cache:开启当前mapper的namespace下的二级缓存 -->

<!--eviction:回收策略，默认值为LRU。

LRU:最近最少使用的策略，移除最长时间不被使用的对象

FIFO:先进先出策略，按对象进入缓存的顺序来移除它们

SOFT:软引用策略，移除基于垃圾回收器状态和软引用规则的对象。

WEAK:弱引用策略，更积极地移除基于垃圾收集器和弱引用规则的对象。

flushInterval：每个60秒刷新缓存。不设置则没有刷新间隔，仅仅在调用语句时刷新

size：最大缓存对象512个，默认值1024。

readOnly：

true:只读，所有缓存中的数据都是只读操作，数据不能修改。为了加快获取缓存速

度，会直接将缓存中的引用交给用户，不安全，速度快。

False:非只读，缓存中的数据可能会被修改。Mybatis会使用序列化和反序列化技术

克隆一份新的数据交给用户，安全，速度慢

Type:指定自定义缓存的全限定类名

只需要实现Cache接口即可作为Mybatis的缓存类

-->

<cache eviction=*"LRU"* flushInterval=*"60000"* size=*"512"* readOnly=*"true"*/>

### 实现Serializable接口

使用二级缓存，与查询结果映射的JavaBean对象实现java.io.Serializable接口。如果存在父类，其成员都需要实现Serializable接口。

实现序列化接口是为了对缓存数据进行序列化和反序列操作，因为二级缓存数据存储介质多种多样，不一定在内存，有可能是硬盘或者远程服务器。

**public** **class** Article **implements** Serializable{}

### 测试代码

@Test

**public** **void** twoLevleCacheTest() **throws** IOException{

InputStream inputStream = Resources.*getResourceAsStream*("sqlMapConfig.xml");

SqlSessionFactory sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

SqlSession session = sqlSessionFactory.openSession();

ArticleMapper mapper = session.getMapper(ArticleMapper.**class**);

List<Article> list1 = mapper.selectArticleByOrderId(1001);

//提交事物

session.commit();

//关闭当前session，必须关闭该session，只有关闭当前session，才会将一级缓存中的数据放入到二级缓存

session.close();

//获得新的session

SqlSession session2 = sqlSessionFactory.openSession();

ArticleMapper mapper2 = session2.getMapper(ArticleMapper.**class**);

List<Article> list2 = mapper2.selectArticleByOrderId(1001);

System.***out***.println(list2);

}

### 控制台日志

19:45:47,314 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

19:45:47,423 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

19:45:47,476 DEBUG selectArticleByOrderId:139 - <== Total: 3

19:45:47,479 DEBUG JdbcTransaction:120 - Resetting autocommit to true on JDBC Connection [oracle.jdbc.driver.T4CConnection@8b87145]

19:45:47,479 DEBUG JdbcTransaction:88 - Closing JDBC Connection [oracle.jdbc.driver.T4CConnection@8b87145]

19:45:47,480 DEBUG PooledDataSource:334 - Returned connection 146305349 to pool.

19:45:47,480 DEBUG ArticleMapper:62 - Cache Hit Ratio [com.xiaoka.mapper.ArticleMapper]: 0.5

[Article [id=1001, name=Java从入门到放弃, price=99.9, remark=小钻风, orderList=null], Article [id=1002, name=C语言从入门到放弃, price=88.9, remark=羊力大仙, orderList=null], Article [id=1003, name=C++从入门到放弃, price=77.9, remark=鹿力大仙, orderList=null]]

### 分析

首先，可以看到数据已经正确的查询出来。

在测试代码中，关闭第一个session以后，Mybatis依然没有发起sql查询，可以证明数据是从二级缓存中获取的，如果不关闭第一个session，那么第二次查询依然会发起sql查询，因为这个时候session2和二级缓存中都没有这个缓存数据。 所以缓存数据默认放在一级缓存中，只有提交或者关闭session以后，一级缓存中的数据才会放入二级缓存。

### 禁用当前select的二级缓存

我们已经如期使用到了二级缓存。而在select标签中配置useCache=false，则可以禁用当前select的二级缓存。这样可以在当前namespace中单独控制某个select不使用二级缓存。

### 配置当前select禁用二级缓存

<!--useCache:默认值为ture,false为禁用二级缓存 -->

<select id=*"selectArticleByOrderId"* parameterType=*"int"* resultType=*"com.xiaoka.bean.Article"* useCache=*"false"*>

select \* from tb\_article where id in(select article\_id from

tb\_item where

order\_id = #{id})

</select>

### 测试结果

当禁用了当前select的二级缓存以后，那么控制台将输出两次查询sql语句的日志

执行twoLevleCacheTest()的测试代码，查看日志

19:57:59,409 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

19:57:59,493 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

19:57:59,555 DEBUG selectArticleByOrderId:139 - <== Total: 3

19:57:59,559 DEBUG JdbcTransaction:120 - Resetting autocommit to true on JDBC Connection [oracle.jdbc.driver.T4CConnection@7dc222ae]

19:57:59,559 DEBUG JdbcTransaction:88 - Closing JDBC Connection [oracle.jdbc.driver.T4CConnection@7dc222ae]

19:57:59,559 DEBUG PooledDataSource:334 - Returned connection 2109874862 to pool.

19:57:59,559 DEBUG JdbcTransaction:132 - Opening JDBC Connection

19:57:59,563 DEBUG PooledDataSource:369 - Checked out connection 2109874862 from pool.

19:57:59,563 DEBUG JdbcTransaction:98 - Setting autocommit to false on JDBC Connection [oracle.jdbc.driver.T4CConnection@7dc222ae]

19:57:59,564 DEBUG selectArticleByOrderId:139 - ==> Preparing: select \* from tb\_article where id in(select article\_id from tb\_item where order\_id = ?)

19:57:59,564 DEBUG selectArticleByOrderId:139 - ==> Parameters: 1001(Integer)

19:57:59,567 DEBUG selectArticleByOrderId:139 - <== Total: 3

[Article [id=1001, name=Java从入门到放弃, price=99.9, remark=小钻风, orderList=null], Article [id=1002, name=C语言从入门到放弃, price=88.9, remark=羊力大仙, orderList=null], Article [id=1003, name=C++从入门到放弃, price=77.9, remark=鹿力大仙, orderList=null]]

### 配置DML操作清空缓存

每一个增删改标签都有默认的flushCache=”true”，所以执行完增删改以后会清空缓存，那么注意了，该属性配置会清空当前session的一级缓存以级当前namespace的二级缓存。也就是说一二级缓存都会清空。

### 配置查询操作清空缓存

每一个select查询标签都有默认的flushCache=”false”,所以以前的查询不会清空缓存。如果在select查询标签中配置了flushCache=”true”,那么发起该查询就会清空缓存数据。

### 禁用一级缓存

在Mybatis3.3以后使用localCacheScope属性可以禁用一级缓存

LocalCacheScope=”session” : 设置一级缓存数据保存在当前session中

LocalCacheScope=”statment” 设置当前session不保存任何数据，即禁用一级缓存

注意：session.clearCacher()函数只会清除当前session的一级缓存。

## 小结

二级缓存的使用场景：**对于访问响应速度要求高，但是实时性不高的查询，可以采用二级缓存技术**。

注意：在使用二级缓存的时候，要设置一下刷新间隔来定时刷新二级缓存，这个刷新间隔根据具体需求来设置，比如设置30分钟、60分钟等，单位为毫秒。

**Mybatis二级缓存对细粒度的数据，缓存实现不好。**

场景：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次查询都是最新的商品信息，此时如果使用二级缓存，就无法实现当一个商品发生变化只刷新该商品的缓存信息而不刷新其他商品缓存信息，因为二级缓存是mapper级别的，当一个商品的信息发送更新，所有的商品信息缓存数据都会清空。

解决此类问题，需要在业务层根据需要对数据有针对性的缓存。

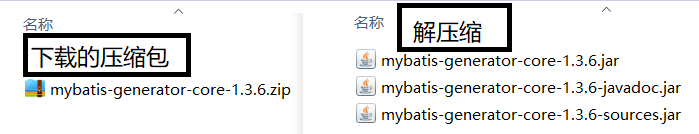
比如可以对经常变化的 数据操作单独放到另一个namespace的mapper中。

# 逆向工程

Mybatis提供了逆向工具，它可以帮助开发者自动生成表与JavaBean的映射类，Mapper接口和Mapper映射文件。

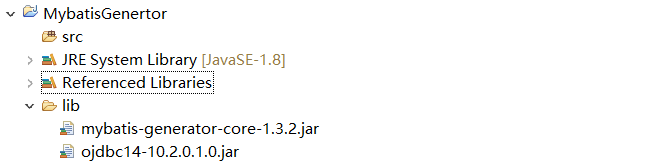
## 下载逆向工程

<https://github.com/mybatis/generator/releases/tag/mybatis-generator-1.3.6>



## 创建Java工程，导入jar包

将JDBC的jar包和逆向工程的核心jar包添加到工程下。



## 创建Generator.java

**public** **class** Generator {

**public** **static** **void** main(String[] args) **throws** Exception {

//这段代码直接去官网介绍文档中复制即可

List<String> warnings = **new** ArrayList<String>();

**boolean** overwrite = **true**;

File configFile = **new** File("generatorConfig.xml");

ConfigurationParser cp = **new** ConfigurationParser(warnings);

Configuration config = cp.parseConfiguration(configFile);

DefaultShellCallback callback = **new** DefaultShellCallback(overwrite);

MyBatisGenerator myBatisGenerator = **new** MyBatisGenerator(config, callback, warnings);

myBatisGenerator.generate(**null**);

}

}

## 创建generatorConfig.xml配置文件

<generatorConfiguration>

<!-- 去官方文档复制 -->

<context id=*"DB2Tables"* targetRuntime=*"MyBatis3"*>

<!-- 配置连接信息 -->

<jdbcConnection driverClass=*"oracle.jdbc.OracleDriver"*

connectionURL=*"jdbc:oracle:thin:@127.0.0.1:1521:orcl"* userId=*"scott"*

password=*"tiger"*>

</jdbcConnection>

<javaTypeResolver>

<property name=*"forceBigDecimals"* value=*"false"* />

</javaTypeResolver>

<!-- 配置生成的JavaBean -->

<javaModelGenerator targetPackage=*"com.xiaoka.bean"*

targetProject=*".\src"*>

<property name=*"enableSubPackages"* value=*"false"* />

<property name=*"trimStrings"* value=*"true"* />

</javaModelGenerator>

<!-- 配置生成的Mapper接口 -->

<sqlMapGenerator targetPackage=*"com.xiaoka.mapper"*

targetProject=*".\src"*>

<property name=*"enableSubPackages"* value=*"false"* />

</sqlMapGenerator>

<!-- 配置生成的Mapper配置文件 -->

<javaClientGenerator type=*"XMLMAPPER"*

targetPackage=*"com.xiaoka.mapper"* targetProject=*".\src"*>

<property name=*"enableSubPackages"* value=*"false"* />

</javaClientGenerator>

<!-- 配置要作用的表 -->

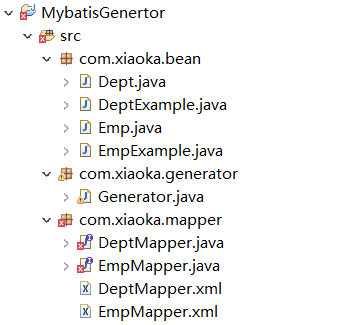
<table tableName=*"emp"*/>

<table tableName=*"dept"*/>

</context>

</generatorConfiguration>

## 运行逆向工程，并刷新



可以看到已经生成了JavaBean,Mapper接口和Mapper.xml配置文件。生成的Mapper接口在逆向工程中报错是因为Mapper接口中需要使用Mybatis的jar包，将这些文件复制到要真正使用的工程中即可正常使用了。

## 测试代码

@Test

**public** **void** testGenaratorMapper(){

ApplicationContext context = **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

EmpMapper empMapper = (EmpMapper) context.getBean("empMapper");

EmpExample example = **new** EmpExample();

//该类用于封装参数

Criteria criteria = example.createCriteria();

criteria.andDeptnoEqualTo((**short**)30);

List<Emp> list = empMapper.selectByExample(example);

**for** (Emp emp : list) {

System.***out***.println(emp);

}

}

控制台日志

Emp [empno=7499, ename=ALLEN, job=SALESMAN, mgr=7698, hiredate=Fri Feb 20 00:00:00 CST 1981, sal=1600, comm=300, deptno=30]

Emp [empno=7521, ename=WARD, job=SALESMAN, mgr=7698, hiredate=Sun Feb 22 00:00:00 CST 1981, sal=1250, comm=500, deptno=30]

Emp [empno=7654, ename=MARTIN, job=SALESMAN, mgr=7698, hiredate=Mon Sep 28 00:00:00 CST 1981, sal=1250, comm=1400, deptno=30]

Emp [empno=7698, ename=BLAKE, job=MANAGER, mgr=7839, hiredate=Fri May 01 00:00:00 CST 1981, sal=2850, comm=null, deptno=30]

Emp [empno=7844, ename=TURNER, job=SALESMAN, mgr=7698, hiredate=Tue Sep 08 00:00:00 CST 1981, sal=1500, comm=0, deptno=30]

Emp [empno=7900, ename=JAMES, job=CLERK, mgr=7698, hiredate=Thu Dec 03 00:00:00 CST 1981, sal=950, comm=null, deptno=30]

**注意：**

Mapper.xml文件已经存在时，如果进行重新生成则mapper.xml文件时，内容不被覆盖而是进行内容追加，结果导致mybatis解析失败。

解决方法：**删除原来已经生成的mapper xml文件再进行生成。**

Mybatis自动生成的po及mapper.java文件不是内容而是直接覆盖没有此问题。

# 附录

## 配置日志输出

<settings>

<setting name=*"logImpl"* value=*"STDOUT\_LOGGING"*/>

</settings>

## 使用log4j配置日志输入

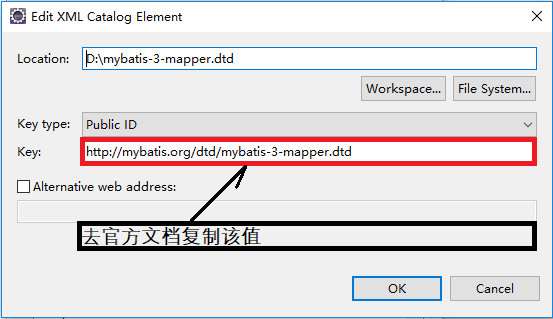
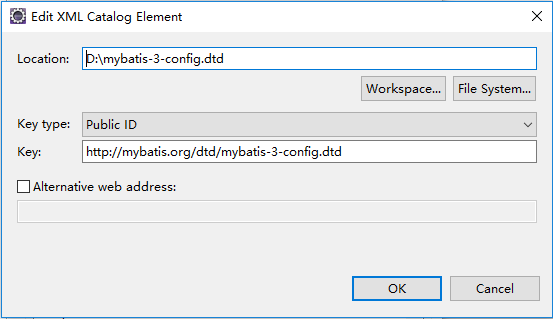
要使用log4j,需创建source folder:名称为：conf,并导入log4j.xml。



## 在Eclipse中引入DTD约束

在 Eclipse 中引入 XML 的 dtd 约束文件，既可以在创建XML文件的时候直接导入约束，而且能够在编写的时候具有提示，方便开发。

* + 找到mybatis-3.5.1.jar,以解压工具（例如winrar,360压缩等）解压出下面两个文件：
    - mybatis-3-config.dtd (路径/org/apache/ibatis/builder/xml)
    - mybatis-3-mapper.dtd (路径/org/apache/ibatis/builder/xml)
  + 在Eclipse-Prefrences-XML-XML Catalog中设置



## Mybatis常用注解

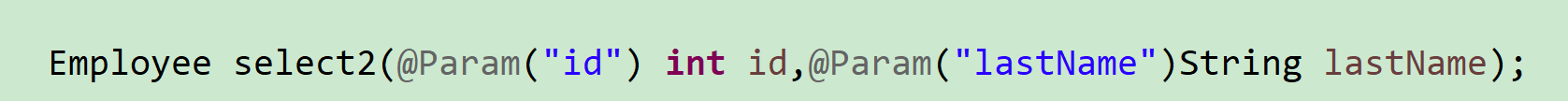
### @Alias

使用@Alias注解为JavaBean指定一个别名

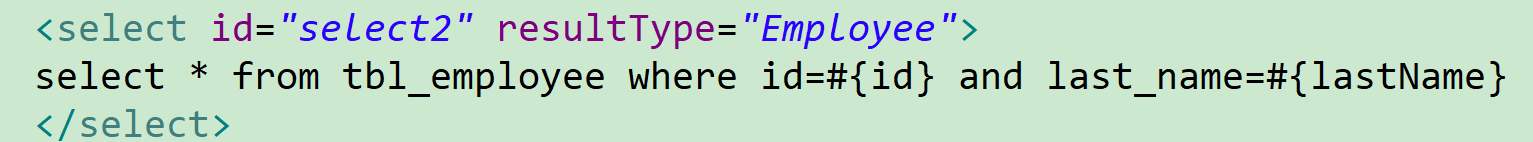


### @Param

在接口中声明方法select2(),使用@Param设置参数名称



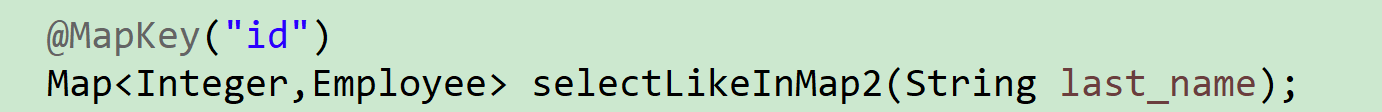
在EmployeeMapper中添加select节点



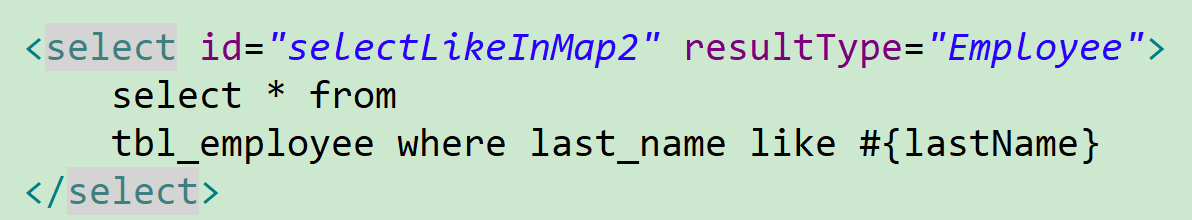
### @MapKey

用于得到主键：对象的map集合

接口添加方法selectOneInMap2()，添加注解@MapKey



mapper中添加select节点，resultType为实体类



## Mybatis使用ehcache缓存

Mybatis本身只是一个持久层框架，它并不是专门的缓存框架，所以对缓存的实现不够好，而且不支持分布式。

实际开发中，我们会选择使用ehcache缓存框架，它是一个分布式的缓存框架。

### Ehcache简介

EhCache 是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认的CacheProvider。

主要特性

快速，简单，多种缓存策略，缓存数据有两级：内存和磁盘，因此无需担心容量问题。支持分布式。

### 整合方案

实际上Cache是一个接口，它的默认实现类是Mybatis的PerpetualCache。如果要整合Mybatis的二级缓存，只需要实现该接口即可。

### 添加eheache的jar包



### 修改映射配置文件

Ehcache框架中EhcacheCache类实现了Cache接口，只需在cache的type属性中指定使用该类即可。

<cache

type=*"org.mybatis.caches.ehcache.EhcacheCache"*

eviction=*"LRU"* flushInterval=*"60000"* size=*"512"* readOnly=*"true"*/>

### 创建ehcache的配置文件

在类路径下，创建ehcache.xml配置文件，该文件中描述了Ehcache缓存框架的一些信息。

在ehcache的核心jar包中已经配置了默认的Ehcache的配置文件，里边详细描述了具体标签的具体属性，这里就不再赘述。

注意：本示例代码并没有真正创建ehcache.xml配置文件，直接使用了Ehcache框架的默认配置文件。

### 测试代码

运行测试代码twoLevleCacheTest(),发现二级缓存依然是生效的。

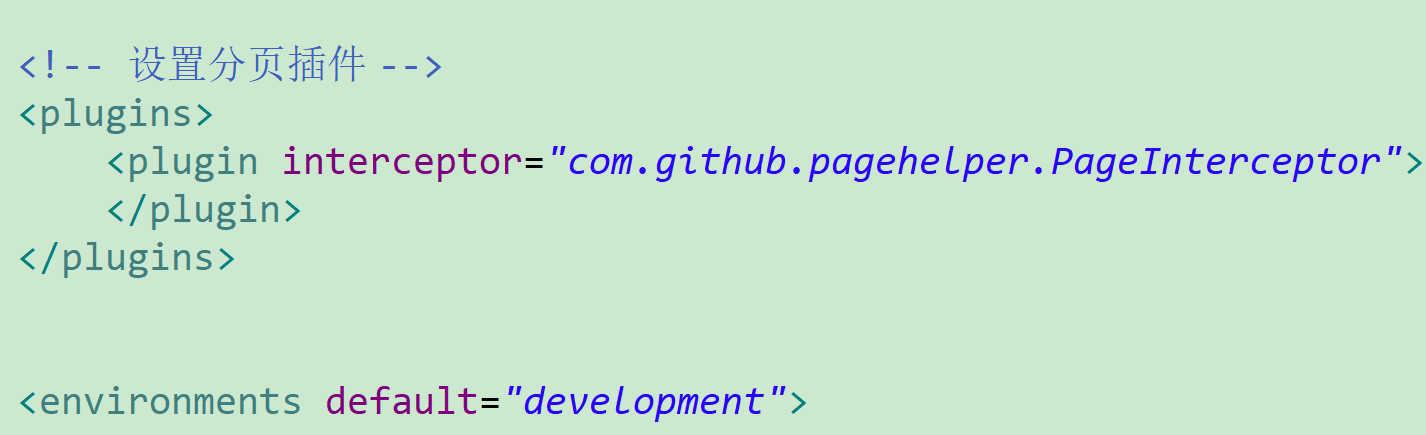
## 分页插件

### 四大对象

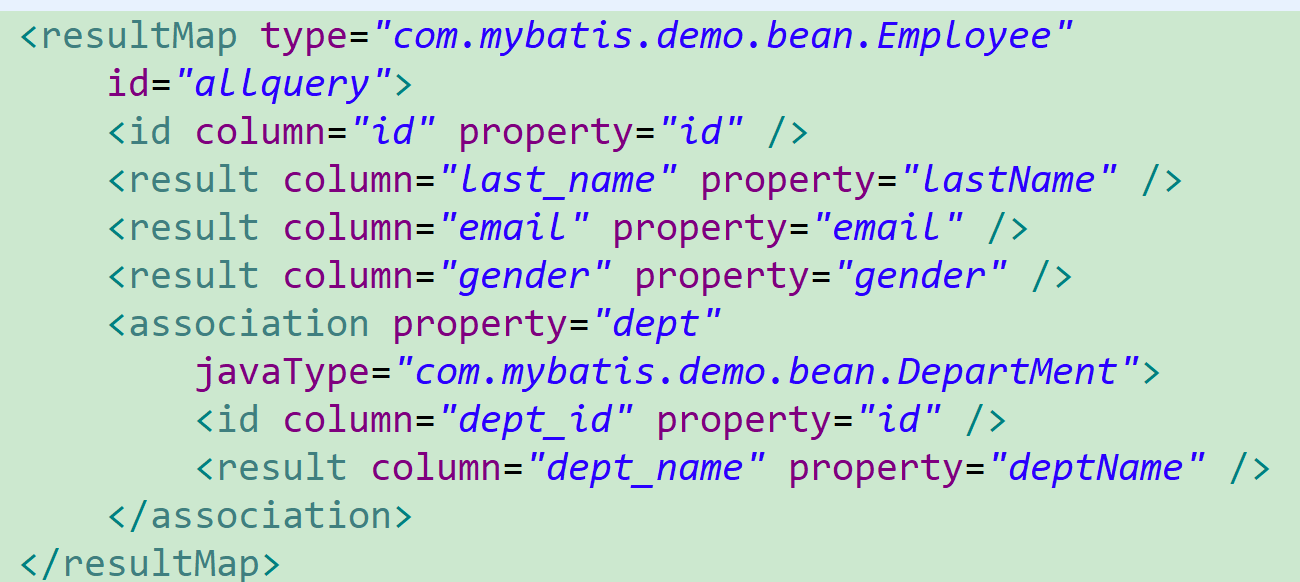
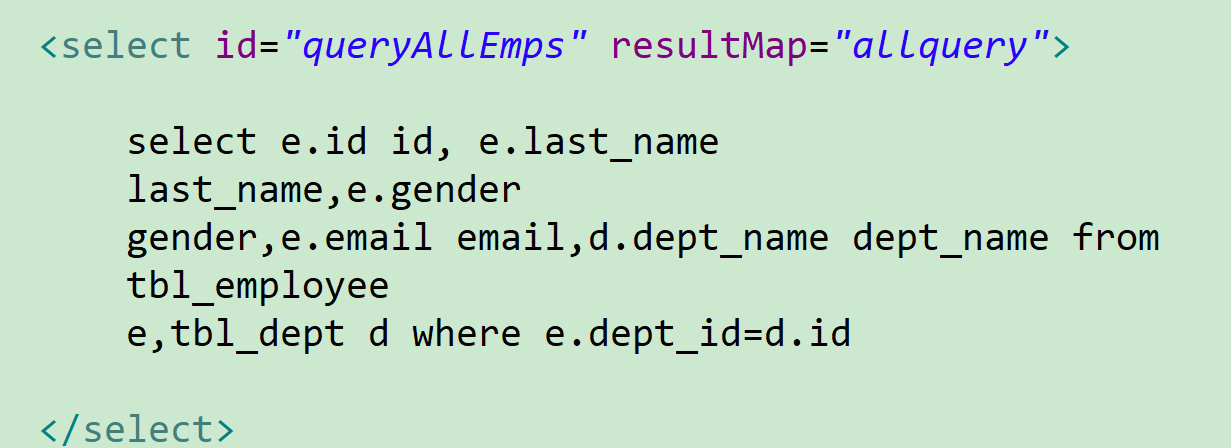
* mybatis 四大对象指的是：executor，statementHandler，parameterHandler 和 resultHandler 对象。
* 这四个对象在 sqlSession 内部共同协作完成 sql 语句的执行，同时也是我们自定义插件拦截的四大对象。
* 插件可以利用动态代理机制一层层的包装目标对象，而实现在目标对象执行目标方法之前进行拦截的效果。
* MyBatis 允许在已映射语句执行过程中的某一点进行拦截调用。
* 默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：
  + Executor (update, query, flushStatements, commit, rollback,getTransaction, close, isClosed)
  + ParameterHandler (getParameterObject, setParameters)
  + ResultSetHandler (handleResultSets, handleOutputParameters)
  + StatementHandler (prepare, parameterize, batch, update, query)

### 示例代码

1. 导入jar包：pagehelper-5.1.9.jar和jsqlparser-2.1。
2. 在mybatis全局设置mybatis-config.xml中设置插件

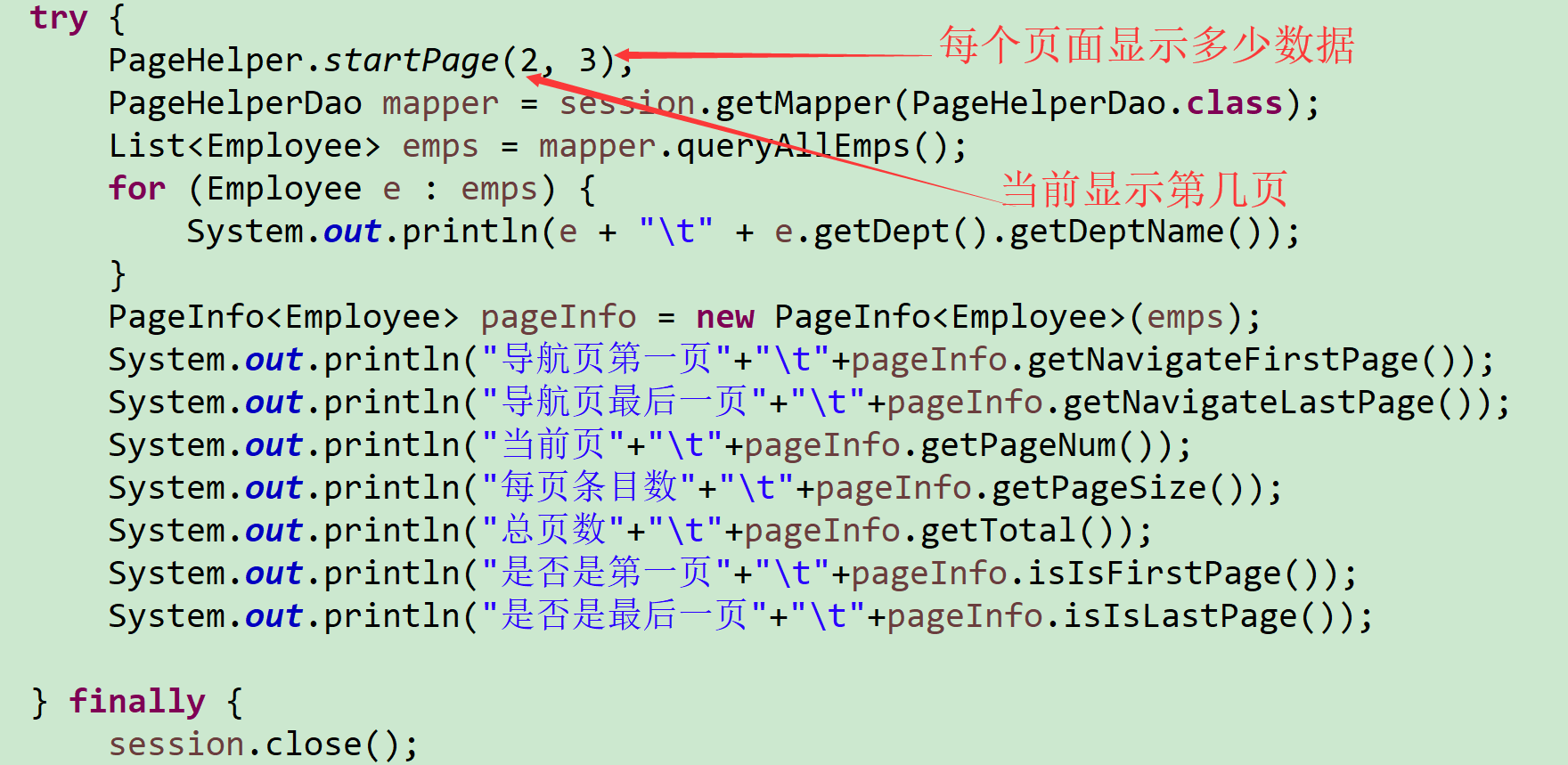
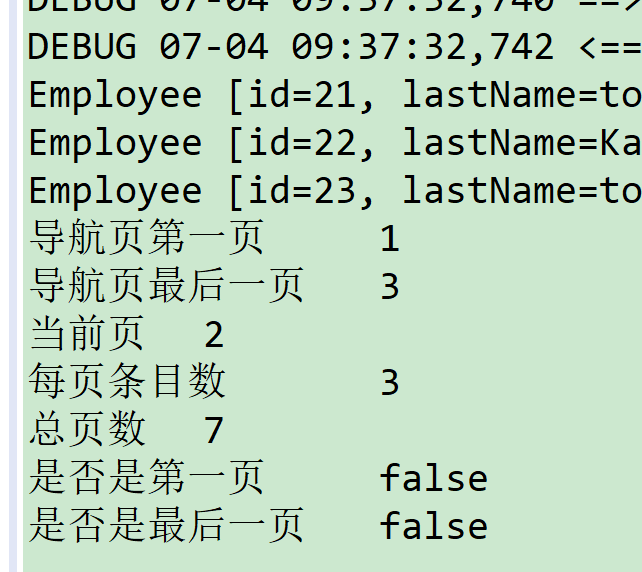


1. 定义测试用查询接口PageHelperDao和对应的PageHelperDao.xml
2. 定义查询的sql和结果集

5.在测试查询前使用分页插件的startPage（）方法

6.根据查询结果集合封装PageInfo对象获得页面数据

## OGNL表达式访问集合伪属性

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

## Spring和Mybatis的整合

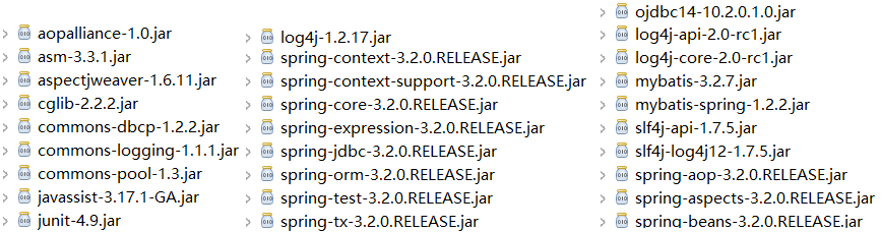
### 整合思路

1. 数据源信息交给spring管理
2. SqlSessionFactory交给spring进行单例管理

3、由spring来管理原始dao的实现类或者mapper代理的代理类。

### 工程搭建

* 数据库的驱动包，本例使用Oracle
* Mybatis的核心包和依赖包
* Mybatis和spring的整合包
* Spring的包
* dbcp数据库连接池包



### Mybatis全局配置文件

在类路径下创建**sqlMapcConfig**.**xml**配置文件作为Mybatis的全局配置文件。

由于事物和数据源全部交给了spring来管理，所以这里不需要再配置事物和数据源。

<configuration>

<settings>

<setting name=*"lazyLoadingEnabled"* value=*"true"* />

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

<setting name=*"cacheEnabled"* value=*"true"* />

</settings>

<typeAliases>

<package name=*"com.xiaoka.bean"* />

</typeAliases>

<mappers>

<!—要加载的Mybatis映射配置文件,待会儿再创建 -->

</mappers>

</configuration>

### Spring配置文件

将log4j.properties文件和db.properties属性文件复制到类路径下。

在类路径下创建ApplicationContext.xml配置文件，作为Spring的全局配置文件

<!-- 加载db.properties属性文件 -->

<context:property-placeholder location=*"db.properties"* />

<!-- 配置数据源 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"${db.driverClass}"* />

<property name=*"url"* value=*"${db.URL}"* />

<property name=*"username"* value=*"${db.userName}"* />

<property name=*"password"* value=*"${db.pwd}"* />

</bean>

<!-- SqlSessionFactory对象由spring来创建 -->

<bean id=*"sqlSessionFactory"* class=*"org.mybatis.spring.SqlSessionFactoryBean"*>

<!--指定Mybatis的全局配置文件路径 -->

<property name=*"configLocation"* value=*"SqlMapConfig.xml"*/>

<!-- 注入数据源 -->

<property name=*"dataSource"* ref=*"dataSource"*/>

</bean>

### 原始Dao开发方式

#### 创建JavaBean类Employee.java

**public** **class** Employee {

**private** Integer empno;

**private** String eName;

**private** String job;

}

#### Mybatis映射文件

在类路径下，创建Mybatis的映射文件employee.xml。

<mapper namespace=*"test"*>

<select id=*"findEmployeeById"* parameterType=*"int"*

resultType=*"com.xiaoka.bean.Employee"*>

select \* from emp where empno = #{id}

</select>

</mapper>

注意要将该映射文件添加到全局配置文件中

**全局配置文件SqlMapConfig.xml**

</configuration>

……其它配置信息

<mappers>

<!—要加载的Mybatis映射配置文件,待会儿再创建，现在已创建并加载进来-->

<mapper resource=*"employee.xml"*/>

</mappers>

</configuration>

#### 创建EmployeeDao接口

**public** **interface** EmployeeDao {

**public** **abstract** Employee selectEmployeeByempno(**int** empno);

}

#### 创建Dao接口的实现类

/\* Dao接口实现类应该继承SqlSessionDaoSupport,该类中有获得SqlSession的函数 \*/

**public** **class** EmployeeDaoImpl **extends** SqlSessionDaoSupport **implements** EmployeeDao{

**public** Employee selectEmployeeByempno(**int** empno){

SqlSession session = **this**.getSqlSession();

Employee employee = session.selectOne("test.findEmployeeById",empno);

**return** employee;

}

}

#### 在Spring中配置Dao实现类

<bean id=*"employeeDao"* class=*"com.xiaoka.dao.impl.EmployeeDaoImpl"*>

<!-- 注入SqlSessionFactory -->

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*/>

</bean>

#### 测试代码

@Test

**public** **void** testSelectEmployeeDao(){

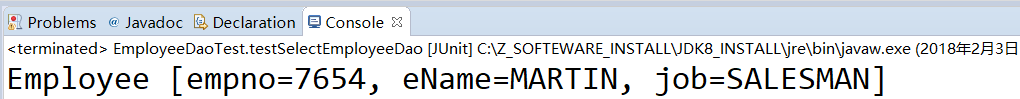
ApplicationContext context = **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

EmployeeDao employeeDao = (EmployeeDao) context.getBean("employeeDao");

Employee employee = employeeDao.selectEmployeeByempno(7654);

System.***out***.println(employee);

}



### Mapper代理开发方式

#### Mapper接口

**public** **interface** EmployeeMapper {

**public** **abstract** Employee selectEmployeeByempno(**int** empno);

}

#### Mapper映射文件

<mapper namespace=*"com.xiaoka.mapper.EmployeeMapper"*>

<select id=*"selectEmployeeByempno"* parameterType=*"int"*

resultType=*"com.xiaoka.bean.Employee"*>

select \* from emp where empno = #{id}

</select>

</mapper>

#### 配置Mapper代理

* 单个mapper代理类配置

<bean id=*"employeeMapper"* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>

<property name=*"mapperInterface"* value=*"com.xiaoka.mapper.EmployeeMapper"*/>

<!-- 注入SqlSessionFactory -->

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"*></property>

</bean>

* 批量设置mapper代理类

<!-- 批量配置Mapper代理类，id值为接口名称【小驼峰式】 -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!--配置扫描的包 -->

<property name=*"basePackage"* value=*"com.xiaoka.mapper"*/>

<!-- 默认不需要配置，它会自动在全局中搜索SqlSessionFactory进行匹配,配置上也无妨 -->

<property name=*"sqlSessionFactoryBeanName"* value=*"sqlSessionFactory"*/>

</bean>

注意：在批量设置mapper代理的时候，JDK8需要使用Spring4.0以上版本。

#### 测试代码

@Test

**public** **void** testSelectEmployeeMapper(){

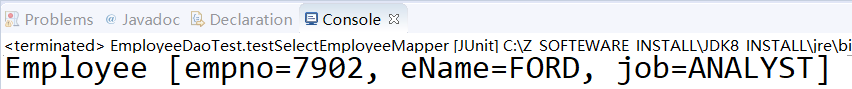
ApplicationContext context = **new** ClassPathXmlApplicationContext("ApplicationContext.xml");

EmployeeMapper employeeMapper = (EmployeeMapper) context.getBean("employeeMapper");

Employee employee = employeeMapper.selectEmployeeByempno(7902);

System.***out***.println(employee);

}

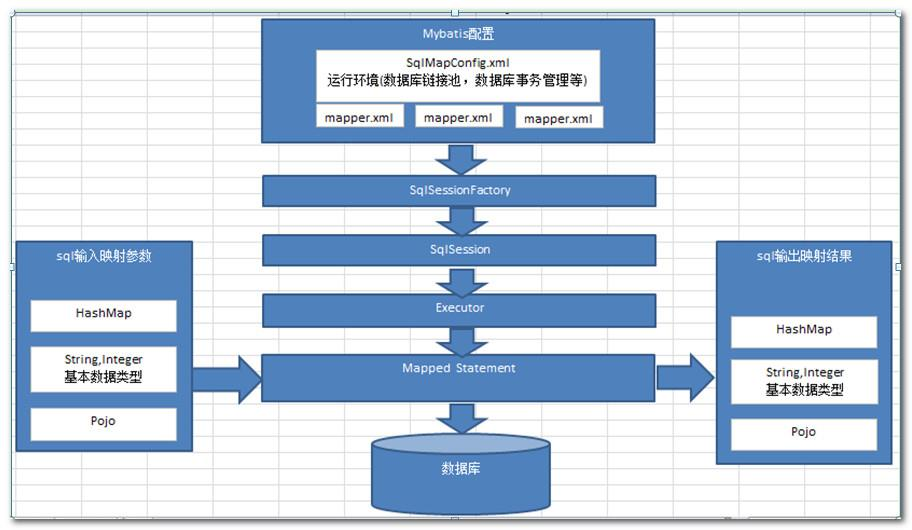


## Mybatis原理

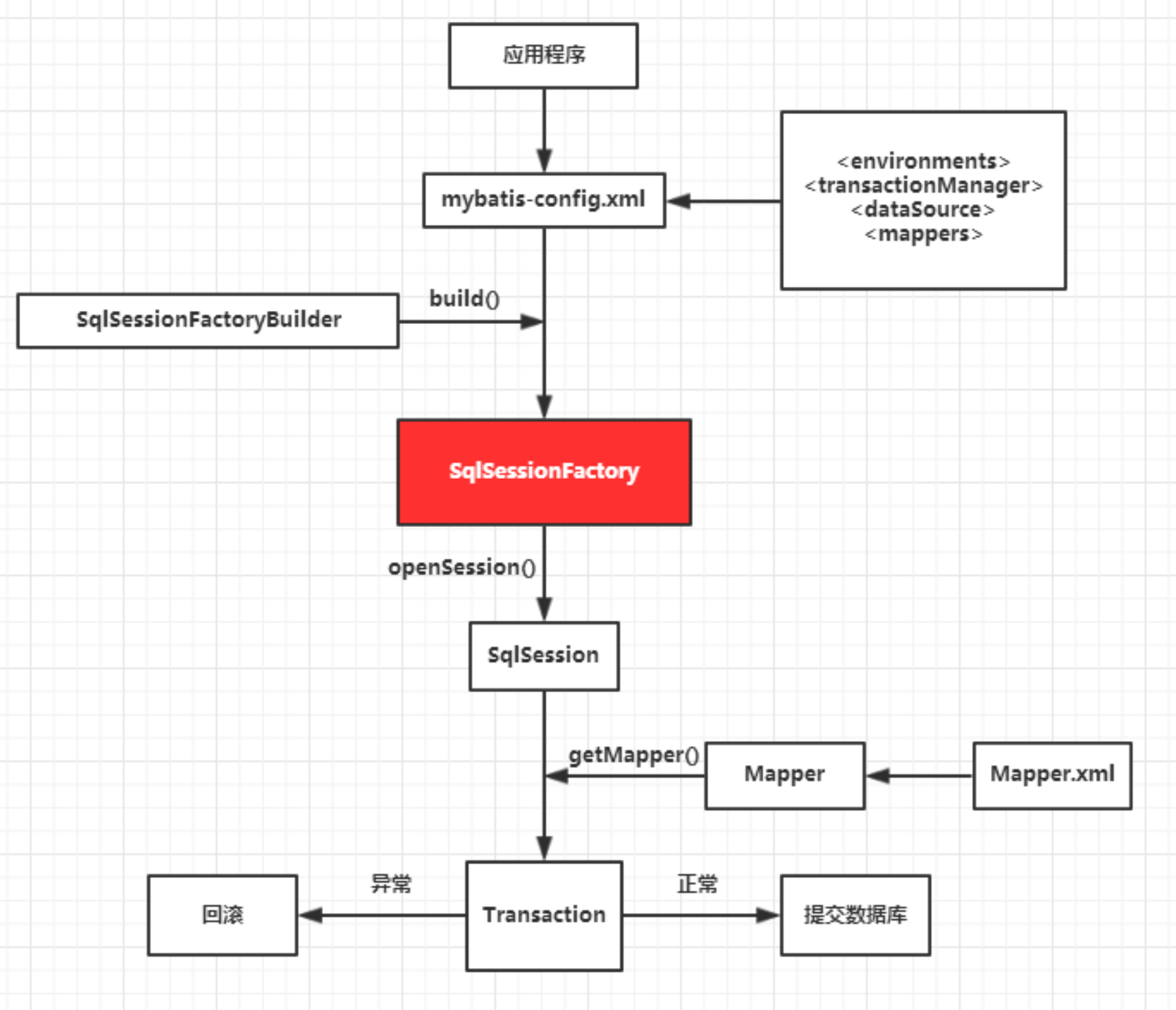
### Mybatis分层图解



### Mybatis原理图



### Mybatis工作流程图



### Mybatis工作原理解析

* mybatis 应用程序通过 SqlSessionFactoryBuilder 从 mybatis-config.xml 配置文件（也可以用 Java 文件配置的方式，需要添加 @Configuration）中构建出 SqlSessionFactory（SqlSessionFactory 是线程安全的）；
* 然后，SqlSessionFactory 的实例直接开启一个 SqlSession，再通过 SqlSession 实例获得 Mapper 对象并运行 Mapper 映射的 SQL 语句，完成对数据库的 CRUD 和事务提交，之后关闭 SqlSession。
* 说明：SqlSession 是单线程对象，因为它是非线程安全的，是持久化操作的独享对象，类似 jdbc 中的 Connection，底层就封装了 jdbc 连接。

### Mybatis工作详细流程

* 1、加载 mybatis 全局配置文件（数据源、mapper 映射文件等），解析配置文件，MyBatis 基于 XML 配置文件生成 Configuration，和一个个 MappedStatement（包括了参数映射配置、动态 SQL 语句、结果映射配置），其对应着 <select | update | delete | insert> 标签项。
* 2、SqlSessionFactoryBuilder 通过 Configuration 对象生成 SqlSessionFactory，用来开启 SqlSession。
* 3、SqlSession 对象完成和数据库的交互：
  1. 用户程序调用 mybatis 接口层 api（即 Mapper 接口中的方法）
  2. SqlSession 通过调用 api 的 Statement ID 找到对应的 MappedStatement 对象
  3. 通过 Executor（负责动态 SQL 的生成和查询缓存的维护）将 MappedStatement 对象进行解析，sql 参数转化、动态 sql 拼接，生成 jdbc Statement 对象
  4. JDBC 执行 sql。
  5. 借助 MappedStatement 中的结果映射关系，将返回结果转化成 HashMap、JavaBean 等存储结构并返回。

## mybatis与hibernate的区别及各自应用场景

Mybatis技术特点：

1. 通过直接编写SQL语句，可以直接对SQL进行性能的优化；
2. 学习门槛低，学习成本低。只要有SQL基础，就可以学习mybatis，而且很容易上手；
3. 由于直接编写SQL语句，所以灵活多变，代码维护性更好。
4. 不能支持数据库无关性，即数据库发生变更，要写多套代码进行支持，移植性不好。

Hibernate技术特点：

1. 标准的orm框架，程序员不需要编写SQL语句。
2. 具有良好的数据库无关性，即数据库发生变化的话，代码无需再次编写。
3. 学习门槛高，需要对数据关系模型有良好的基础，而且在设置OR映射的时候，需要考虑好性能和对象模型的权衡。
4. 程序员不能自主的去进行SQL性能优化。

Mybatis应用场景：

需求多变的互联网项目，例如电商项目。

Hibernate应用场景：

需求明确、业务固定的项目，例如OA项目、ERP项目等。