Mybatis第一天

第一章Mybatis的概述

第一节 框架

1. 框架的概念

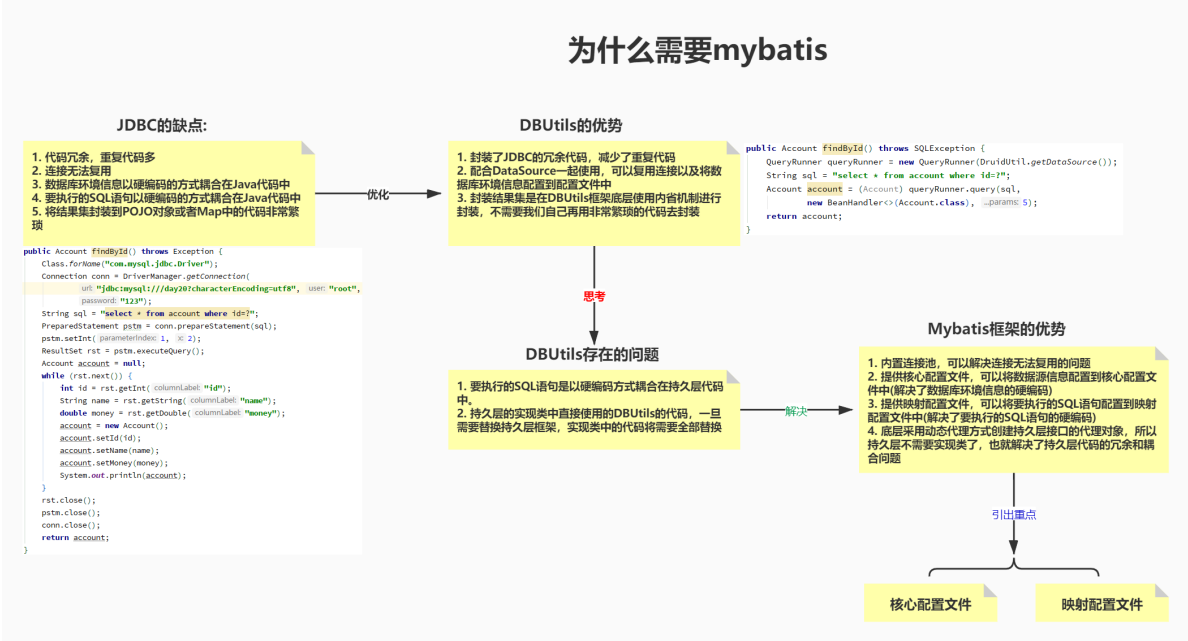
源自于建筑学，隶属土木工程，后发展到软件工程领域
软件工程框架：经过验证的，具有一定功能的，半成品软件，我们基于框架写代码(站在巨人的肩膀)

- 经过验证(我们使用的框架都是经过了市场验证可行之后的)
- 具有一定功能(每个框架都有其特定的功能)
- 半成品(封装固定代码，让使用者配置可变数据)

2. 框架的作用

1. 提高开发效率(封装复杂的API，提供简易的方法调用)
2. 增强可重用性
3. 提供编写规范
4. 节约维护成本
5. 解耦底层实现原理(当底层技术发生变化，上层应用程序不需要改动)

第二节 为什么要学习Mybatis



1. 持久层技术JDBC存在的问题

1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
2. Sql 语句在代码中硬编码，造成代码不易维护，实际应用 sql 变化的可能较大， sql 变动需要改变 java 代码。
3. 使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。

4. 对结果集解析存在硬编码(查询列名)， sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便

2. DBUtils带来的变化

1. 封装了JDBC的冗余代码，减少了重复代码
2. 配合DataSource一起使用，可以复用连接以及将数据库环境信息配置到DataSource的配置文件中，解决了硬编码问题
3. 封装结果集的操作是在DBUtils底层使用ResultSetHandler接口的实现类完成的，不需要我们自己再用繁琐的代码解析封装结果集了

3. DBUtils的局限性

1. 如果要进行事务控制的话，还是需要自己管理Connection连接，无法将Connection交给框架统一管理
2. 自身功能不够强大(例如没有获取自增长主键值的功能)
3. 要执行的SQL语句还是以硬编码的方式耦合在代码中
4. 持久层的实现类中直接使用DBUtils的代码，一旦需要替换持久层框架，这种耦合性必然会带来大量的代码修改

4. Mybatis的优势

1. 内置连接池，可以解决连接无法复用的问题
2. 提供全局配置文件，可以让使用者将数据库环境信息等等配置到全局配置文件中，这样就解决了数据库环境信息的硬编码问题
3. 提供了映射配置文件，可以让使用者将要执行的SQL语句等等信息配置到映射配置文件中，这样就解决了SQL语句的硬编码问题
4. 底层采用动态代理技术创建持久层接口的代理对象，所以使用者不需要编写持久层的实现类了，也就解决了持久层代码的冗余和耦合问题
5. Mybatis功能非常强大，例如我们需要动态SQL等等它都能实现

第三节 Mybatis的起源和发展(了解)

1. Mybatis的发展历程

MyBatis最初是Apache的一个开源项目**iBatis**, 2010年6月这个项目由Apache Software Foundation迁移到了Google Code。随着开发团队转投Google Code旗下， iBatis3.x正式更名为MyBatis。代码于2013年11月迁移到Github。下载地址为: https://github.com/mybatis/mybatis-3

iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。 iBatis提供的持久层框架包括SQL Maps和Data Access Objects（DAO）。

2. Mybatis的特征

- MyBatis支持定制化SQL、存储过程以及高级映射
- MyBatis避免了几乎所有的JDBC代码和手动设置参数以及结果集解析操作
- MyBatis可以使用简单的XML或注解实现配置和原始映射；将接口和Java的POJO（Plain Ordinary Java Object，普通的Java对象）映射成数据库中的记录
- Mybatis是一个半自动的ORM（Object Relation Mapping）框架

3. 什么是ORM

ORM(Object Relational Mapping)：对象关系映射,指的是持久化数据和实体对象的映射模式，为了解决面向对象与关系型数据库存在的互不匹配的现象的技术。其具体的映射规则是:一张表对应一个类，表中的各个字段对应类中的属性，表中的一条数据对应类的一个对象

4. Mybatis和其它持久层技术的对比

- JDBC
  - SQL 夹杂在Java代码中耦合度高，导致硬编码内伤
  - 维护不易且实际开发需求中 SQL 有变化，频繁修改的情况多见
  - 代码冗长，开发效率低
- Hibernate 和 JPA
  - 操作简便，开发效率高
  - 程序中的长难复杂 SQL 需要绕过框架
  - 内部自动生产的 SQL，不容易做特殊优化
  - 基于全映射的全自动框架，大量字段的 POJO 进行部分映射时比较困难。
  - 反射操作太多，导致数据库性能下降
- MyBatis
  - 轻量级，性能出色
  - SQL 和 Java 编码分开，功能边界清晰。Java代码专注业务、SQL语句专注数据
  - 开发效率稍逊于Hibernate，但是完全能够接收

第二章 Mybatis的入门程序

第一节 目标和准备工作

1. 目标

使用Mybatis作为持久层框架，执行查询数据的SQL语句并且获取结果集

2. 数据建模

2.1 物理建模

```
CREATE DATABASE `mybatis-example`;
USE `mybatis-example`;
CREATE TABLE `t_emp`(
emp_id INT AUTO_INCREMENT,
emp_name VARCHAR(100),
emp_salary DOUBLE(10,5),
PRIMARY KEY(emp_id)
);
INSERT INTO `t_emp`(emp_name,emp_salary) VALUES("tom",200.33);
INSERT INTO `t_emp`(emp_name,emp_salary) VALUES("jerry",300.33);
```

2.2 Lombok的使用

2.2.1 Lombok的环境

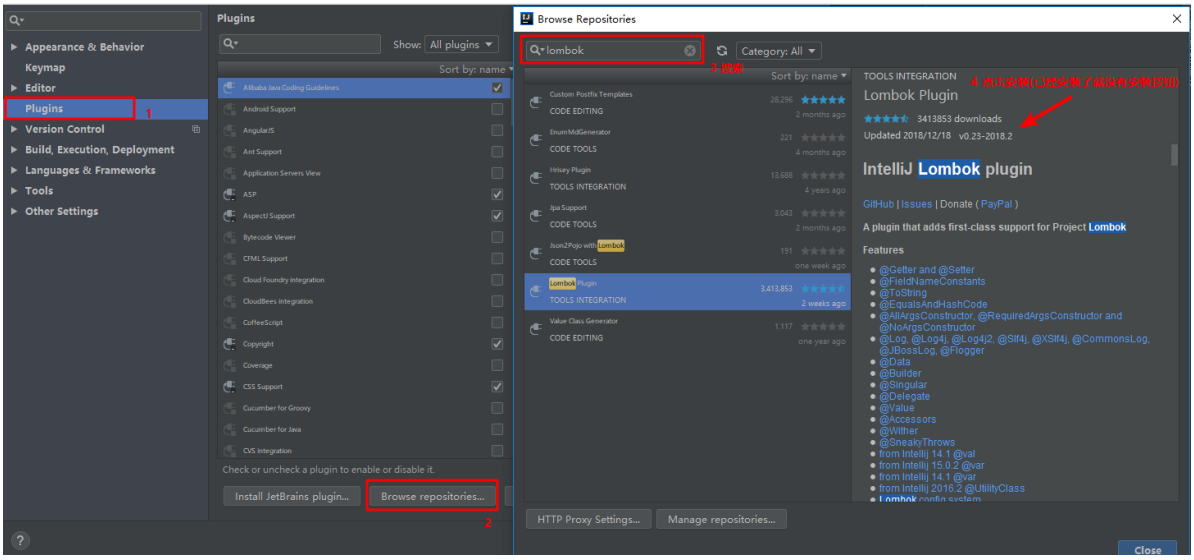
Lombok是一个Java库，能自动插入编辑器并构建工具，简化Java开发。它可以通过**添加注解**的方式，Lombok能以简单的注解形式来简化java代码，提高开发人员的开发效率。例如开发中经常需要写的javabean，都需要花时间去添加相应的getter/setter，也许还要去写构造器、equals等方法，而且需要维护，当属性多时会出现大量的getter/setter方法，这些显得很冗长也没有太多技术含量，一旦修改属性，就容易出现忘记修改对应方法的失误,使代码看起来更简洁些。官网: https://www.projectlombok.org/

maven依赖:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.8</version>
    <scope>provided</scope>
</dependency>
```

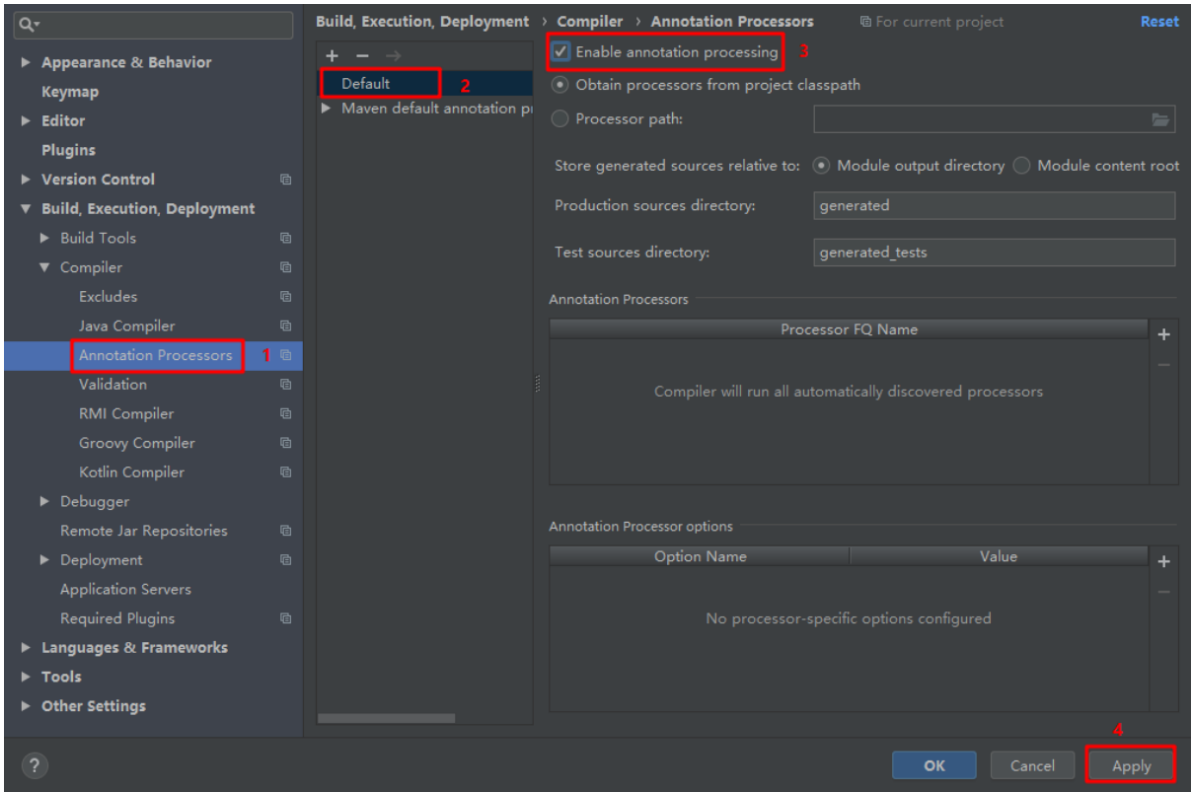
安装插件:

使用Lombok还需要插件的配合，我使用开发工具为idea. 打开idea的设置，点击Plugins，点击Browse repositories，在弹出的窗口中搜索lombok，然后安装即可



解决编译时出错问题:

编译时出错，可能是没有enable注解处理器。Annotation Processors > Enable annotation processing。设置完成之后程序正常运行。



2.2.2 Lombok的注解

1. Data注解

@Data注解在类上，会为类的所有属性自动生成setter/getter、equals、canEqual、hashCode、toString方法，如为final属性，则不会为该属性生成setter方法。

2. Getter和Setter注解

如果觉得@Data太过残暴不够精细，可以使用@Getter/@Setter注解，此注解在属性上，可以为相应的属性自动生成Getter/Setter方法.

3. ToString注解

类使用@ToString注解，Lombok会生成一个toString()方法，默认情况下，会输出类名、所有属性（会按照属性定义顺序），用逗号来分割。 通过exclude属性指定忽略字段不输出,

4. NoArgsConstructor注解

给类添加无参构造

5. AllArgsConstructor注解

给类添加全参构造

2.3 逻辑建模

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Employee {

    private Integer empId;

    private String empName;

    private Double empSalary;
}
```

3. 引入依赖

```
<dependencies>
    <!-- Mybatis核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.7</version>
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- MySQL驱动 -->
    <dependency>
```

```
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.3</version>
    </dependency>
</dependencies>
```

4. 创建持久层接口

持久层接口就是我们之前的Dao接口，但我们在使用Mybatis框架的时候习惯给它们命名为Mapper接口

```
public interface EmployeeMapper {
    Employee selectEmployee(Integer empId);
}
```

5. 加入日志框架

因为Mybatis中内置使用log4j进行日志打印，所以我们要想观察到Mybatis框架内部的日志，就必须引入log4j的依赖，并且提供log4j的配置文件

5.1 引入log4j的依赖

```
<!-- log4j日志 -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

5.2 加入log4j的配置文件

log4j支持XML和properties属性文件两种形式。无论使用哪种形式，文件名是固定的：

- log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd HH:mm:ss,SSS}
%m  (%F:%L) \n" />
        </layout>
    </appender>
    <logger name="java.sql">
        <level value="debug" />
    </logger>
    <logger name="org.apache.ibatis">
        <level value="info" />
    </logger>
    <root>
        <level value="debug" />
        <appender-ref ref="STDOUT" />
    </root>
</log4j:configuration>
```

- log4j.properties

```
# 输出的日志级别
log4j.rootLogger=DEBUG,stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
#[%-5p] %t %l %d %rms:%m%n
#%d{yyyy-MM-dd HH:mm:ss,SSS\} %-5p [%t] {%c}-%m%n
log4j.appender.stdout.layout.ConversionPattern=[%-5p] %t %l %d %rms:%m%n
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=D:\\idea_project\\mybatis.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS\} %-5p
[%t] {%c}-%m%n
```

5.3 log4j的简介

1. 日志的级别

ATAL(致命)>ERROR(错误)>WARN(警告)>INFO(信息)>DEBUG(调试)，从左到右打印的内容越来越详细

2. STDOUT

是standard output的缩写，意思是标准输出。对于Java程序来说，打印到标准输出就是打印到控制台。

3. 打印效果

> DEBUG 05-24 18:51:13,331 ==> Preparing: select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where emp_id=? (BaseJdbcLogger.java:137) DEBUG 05-24 18:51:13,371 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137) DEBUG 05-24 18:51:13,391 <==   Total: 1 (BaseJdbcLogger.java:137) o = Employee{empId=1, empName='tom', empSalary=200.33}

第二节 Mybatis工程的配置文件

1. Mybatis的全局配置文件

习惯上命名为mybatis-config.xml，这个文件名仅仅只是建议，并非强制要求。该配置文件建议存放在resources根目录下。将来整合Spring之后，这个配置文件可以省略，所以大家操作时可以直接复制、粘贴，将需要修改的地方进行修改就行了

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
    <!-- environments表示配置Mybatis的开发环境，可以配置多个环境，在众多具体环境中，使用
default属性指定实际运行时使用的环境。default属性的取值是environment标签的id属性的值。 -->
    <environments default="development">
        <!-- environment表示配置Mybatis的一个具体的环境 -->
        <environment id="development">

            <!-- Mybatis的内置的事务管理器 -->
            <transactionManager type="JDBC"/>

            <!-- 配置数据源 -->
```

```
            <dataSource type="POOLED">

                <!-- 建立数据库连接的具体信息 -->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis-
example"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <!-- Mapper注册：指定Mybatis映射文件的具体位置 -->
        <!-- mapper标签：配置一个具体的Mapper映射文件 -->
        <!-- resource属性：指定Mapper映射文件的实际存储位置，这里需要使用一个以类路径根目录
为基准的相对路径 -->
        <!--     对Maven工程的目录结构来说，resources目录下的内容会直接放入类路径，所以这里
我们可以以resources目录为基准 -->
        <mapper resource="mappers/EmployeeMapper.xml"/>
    </mappers>
</configuration>
```

2. Mybatis的映射配置文件

Mybatis的映射配置文件是和Mapper接口对应的，用来编写要执行的SQL语句以及封装结果集的，映射配置文件包含了数据和对象之间的映射关系以及要执行的 SQL 语句。一个映射配置文件对应一个Mapper接口，映射配置文件中的一个子标签对应Mapper接口中的一个方法

要求:

1. 名称空间要和对应的Mapper接口的全限定名保持一致
2. Mapper接口中的增、删、改、查方法分别对应映射配置文件中的insert、delete、update、select标签
    1. 标签的id对应方法的名字
    2. 标签的parameterType属性对应方法的参数类型
    3. 标签的resultType属性(只有select标签有)对应方法的返回值类型

```
<?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">
<!--
    1. 映射配置文件，怎么才能和Mapper接口建立起一一对应的关系？
       通过根标签的namespace属性进行关联的: namespace属性的值要和对应的Mapper接口的全限定
名保持一致
    2. 映射配置文件中的SQL语句，怎么才能和Mapper接口中的方法建立起一一对应的关系?
       就是通过mapper的子标签进行关联的:
       2.1 增删改查方法，分别对应:insert、delete、update、select标签,标签中就编写SQL语句
       2.2 select标签的id属性要和对应的方法的名字一致
   3. 方法中的参数怎么设置给SQL语句呢?
       3.1 标签上的parameterType属性就对应参数的类型(可以省略)
       3.2 通过#{}引用参数: 如果只有一个简单类型的参数我们可以使用#{任意字符串}引用参数
   4. SQL语句执行之后，怎么将查询到的结果集进行封装呢?
       4.1 通过select标签的resultType属性进行指定: 属性值就是要封装结果的POJO的全限定名
-->
```

```
<mapper namespace="com.atguigu.mapper.EmployeeMapper">
    <select id="selectEmployee" resultType="com.atguigu.pojo.Employee">
        select emp_id empId,emp_name empName,emp_salary empSalary from t_emp
where emp_id=#{empId}
    </select>
</mapper>
```

第三节 Mybatis的测试代码

因为后续使用Spring整合Mybatis之后，下面的那大段测试代码都不用写了，所以我们不需要记忆，直接拷贝修改即可

```
@Test
public void testFindEmployee() throws IOException {
    //1. 想尽办法让Mybatis去创建出EmployeeMapper接口的代理对象
    // 1.1创建SqlSessionFactory对象
    //以输入流的形式加载Mybatis的核心配置文件
    InputStream inputStream = Resources.getResourceAsStream( "mybatis-
config.xml");

    // 基于读取Mybatis配置文件的输入流创建SqlSessionFactory对象
    SqlSessionFactory sessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

    // 1.2 使用SqlSessionFactory对象开启一个会话
    SqlSession session = sessionFactory.openSession();

    // 1.3 创建EmployeeMapper接口的代理对象
    EmployeeMapper employeeMapper = session.getMapper(EmployeeMapper.class);

    //2. 使用EmployeeMapper的对象调用方法
    Employee employee = employeeMapper.selectEmployee(4);
    System.out.println(employee);

    //关闭资源
    inputStream.close();
    session.close();
}
```

第三章 基本的增删改

1. insert标签

映射配置文件

```
<!--
        对应com.atguigu.mapper.EmployeeMapper.insertEmployee(Employee employee)方
法
        引用参数: 参数是POJO类型，那么我们就是需要拿POJO对象的属性
                通过getXXX方法获取，我们只需要写#{getXXX方法的get后面的内容，首字母小写},
    -->
<insert id="insertEmployee">
    insert into t_emp (emp_name,emp_salary) values (#{empName},#{empSalary})
</insert>
```

Java代码中的Mapper接口：

```
public interface EmployeeMapper {
    /**
     * 根据empId查询员工信息
     * @param empId
     * @return
     */
    Employee selectEmployee(Integer empId);

    /**
     * 插入员工信息
     * @param employee
     */
    void insertEmployee(Employee employee);
}
```

Java代码中的junit测试：

```
package com.atguigu.mybatis;

import com.atguigu.mapper.EmployeeMapper;
import com.atguigu.pojo.Employee;
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;

/**
 * 包名:com.atguigu.mybatis
 *
 * @author Leevi
 * 日期2021-08-24  10:34
 * Mybatis自动开启了事务，所以需要我们手动提交
 */
public class TestMybatis {
    private EmployeeMapper employeeMapper;
    private SqlSession session;
    private InputStream inputStream;

    @Before
    public void init() throws IOException {
        //1. 想尽办法让Mybatis去创建出EmployeeMapper接口的代理对象
        // 1.1创建SqlSessionFactory对象
        //以输入流的形式加载Mybatis的核心配置文件
        inputStream = Resources.getResourceAsStream( "mybatis-config.xml");

        // 基于读取Mybatis配置文件的输入流创建SqlSessionFactory对象
        SqlSessionFactory sessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
```

```
        // 1.2 使用SqlSessionFactory对象开启一个会话
        session = sessionFactory.openSession();

        // 1.3 创建EmployeeMapper接口的代理对象
        employeeMapper = session.getMapper(EmployeeMapper.class);
    }

    @After
    public void destroy() throws IOException {
        //提交事务
        session.commit();

        //关闭资源
        inputStream.close();
        session.close();
    }
    @Test
    public void testFindEmployee() throws IOException {
        //2. 使用EmployeeMapper的对象调用方法
        Employee employee = employeeMapper.selectEmployee(4);
        System.out.println(employee);
    }

    @Test
    public void testInsertEmployee(){
        //先要创建一个Employee存储数据
        Employee employee = new Employee(null, "aobama", 2500d);

        employeeMapper.insertEmployee(employee);
    }
}
```

2. delete标签

映射配置文件

```
<!--
        对应com.atguigu.mapper.EmployeeMapper.deleteEmployeeByEmpId(Integer
empId)
        引用参数: 参数是简单类型，那么我们只需要#{任意字符串}
    -->
<delete id="deleteEmployeeByEmpId">
    delete from t_emp where emp_id=#{empId}
</delete>
```

Java代码中的Mapper接口：

```
public interface EmployeeMapper {

    /**
     * 根据empId查询员工信息
     * @param empId
     * @return
     */
    Employee selectEmployee(Integer empId);
```

```
    /**
     * 插入员工信息
     * @param employee
     */
    void insertEmployee(Employee employee);

    /**
     * 根据empId删除员工
     * @param empId
     */
    void deleteEmployeeByEmpId(Integer empId);
}
```

Java代码中的junit测试：

```
@Test
public void testDeleteEmployByEmpId(){
    employeeMapper.deleteEmployeeByEmpId(6);
}
```

3. update标签

映射配置文件

```
<!--
        对应com.atguigu.mapper.EmployeeMapper.updateEmployee
        引用参数: 参数是POJO类型
    -->
<update id="updateEmployee">
    update t_emp set emp_name=#{empName},emp_salary=#{empSalary} where emp_id=#
{empId}
</update>
```

Java代码中的Mapper接口：

```
package com.atguigu.mapper;

import com.atguigu.pojo.Employee;
/**
 * 包名:com.atguigu.mapper
 *
 * @author Leevi
 * 日期2021-08-24  09:27
 */
public interface EmployeeMapper {
    /**
     * 根据empId查询员工信息
     * @param empId
     * @return
     */
    Employee selectEmployee(Integer empId);

    /**
     * 插入员工信息
     * @param employee
```

```
     */
    void insertEmployee(Employee employee);

    /**
     * 根据empId删除员工
     * @param empId
     */
    void deleteEmployeeByEmpId(Integer empId);

    /**
     * 修改员工信息
     * @param employee
     */
    void updateEmployee(Employee employee);
}
```

Java代码中的junit测试：

```
@Test
public void testUpdateEmployee(){
    //1. 选查询出empId为4的数据
    Employee employee = employeeMapper.selectEmployee(4);

    //2. 修改内存中的employee
    employee.setEmpName("aobama");
    employee.setEmpSalary(3000d);

    //3. 修改数据库的employee
    employeeMapper.updateEmployee(employee);
}
```

第四章 数据输入

第一节 在SQL语句中获取参数

1. #{}的方式

Mybatis会在运行过程中，把配置文件中的SQL语句里面的**#{}**转换为“**?**”占位符，发送给数据库执行。

配置文件中的SQL：

```
<delete id="deleteEmployeeById" parameterType="int">
    delete from t_emp where emp_id=#{empId}
</delete>
```

实际执行的SQL：

```
delete from t_emp where emp_id=?
```

2. ${}的方式(了解)

Mybatis会在运行过程中， 将来会**根据${}拼字符串**

配置文件中的SQL语句

```
<select id="selectEmployeeByName" resultType="com.atguigu.pojo.Employee">
    select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where
emp_name like '%${empName}%'
</select>
```

Mapper接口:

注意：由于Mapper接口中方法名是作为SQL语句标签的id，不能重复，所以**Mapper接口中不能出现重名的方法，不允许重载**!

```
public interface EmployeeMapper {

    Employee selectEmployee(Integer empId);

    //@Param注解用于给当前参数取别名，在mybatis的映射配置文件中要根据这个别名获取参数值
    Employee selectEmployeeByName(@Param("empName") String empName);

    int insertEmployee(Employee employee);

    int deleteEmployee(Integer empId);

    int updateEmployee(Employee employee);
}
```

junit测试:

```
@Test
public void testDollar() {

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

    Employee employee = employeeMapper.selectEmployeeByName("r");

    System.out.println("employee = " + employee);
}
```

实际执行的SQL:

```
select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where
emp_name like '%r%'
```

3. 使用场景分析

在SQL语句中，数据库表的表名不确定，需要外部动态传入，此时不能使用#{}，因为数据库不允许表名位置使用问号占位符，此时只能使用${}。

其他情况，**只要能用#{}肯定不用${}**，避免SQL注入。

第二节 数据输入的概念

这里数据输入具体是指上层方法（例如Service方法）调用Mapper接口时，数据传入的形式。

- 简单类型：只包含一个值的数据类型
  - 基本数据类型：int、byte、short、double、……
  - 基本数据类型的包装类型：Integer、Character、Double、……
  - 字符串类型：String
- 复杂类型：包含多个值的数据类型
  - 实体类类型：Employee、Department、……
  - 集合类型：List、Set、Map、……
  - 数组类型：int[]、String[]、……
  - 复合类型：List、实体类中包含集合……

第三节 简单类型参数

1. 单个简单类型参数

Mapper接口中的抽象方法

```
Employee selectEmployee(Integer empId);
```

映射配置文件：此时SQL语句中获取参数#{任意字符串}

```
<select id="selectEmployee" resultType="com.atguigu.mybatis.entity.Employee">
    select emp_id empId,emp_name empName,emp_salary empSalary from t_emp where
emp_id=#{empId}
</select>
```

2. 多个简单类型参数

Mapper接口中抽象方法: 此时每个方法需要使用**Param**注解命名

```
int updateEmployee(@Param("empId") Integer empId,@Param("empSalary") Double
empSalary);
```

映射配置文件：此时SQL语句中获取参数#{Param注解命的名}

```
<update id="updateEmployee">
    update t_emp set emp_salary=#{empSalary} where emp_id=#{empId}
</update>
```

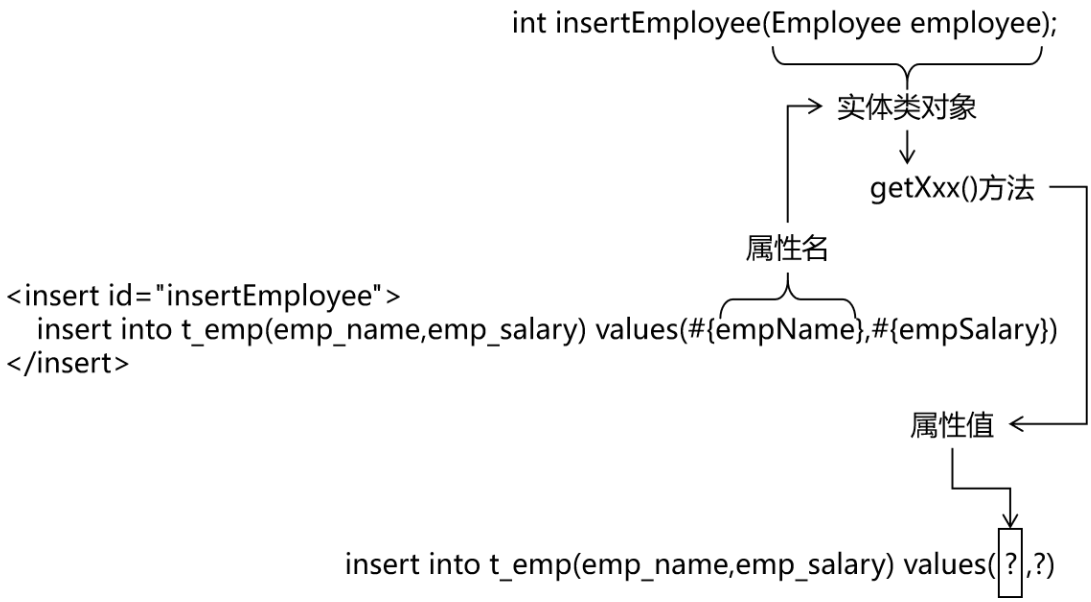
第四节 实体类类型参数

Mapper接口中抽象方法:

```
int insertEmployee(Employee employee);
```

映射配置文件：此时SQL语句获取参数#{getXXX方法对应的名字,首字母改小写}

```
<insert id="insertEmployee">
    insert into t_emp(emp_name,emp_salary) values(#{empName},#{empSalary})
</insert>
```



Mybatis会根据#{}中传入的数据，加工成getXxx()方法，通过反射在实体类对象中调用这个方法，从而获取到对应的数据。填充到#{}这个位置。

第五节 Map类型参数

Mapper接口中抽象方法:

```
int updateEmployeeByMap(Map<String, Object> paramMap);
```

映射配置文件: 此时SQL语句获取参数#{Map的key}

```
<update id="updateEmployeeByMap">
    update t_emp set emp_salary=#{empSalaryKey} where emp_id=#{empIdKey}
</update>
```

junit测试:

```
@Test
public void testUpdateEmpNameByMap() {

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

    Map<String, Object> paramMap = new HashMap<>();

    paramMap.put("empSalaryKey", 999.99);
    paramMap.put("empIdKey", 5);

    int result = mapper.updateEmployeeByMap(paramMap);

    System.out.println("result = " + result);
}
```

使用场景：
有很多零散的参数需要传递，但是没有对应的实体类类型可以使用。使用@Param注解一个一个传入又太麻烦了。所以都封装到Map中。