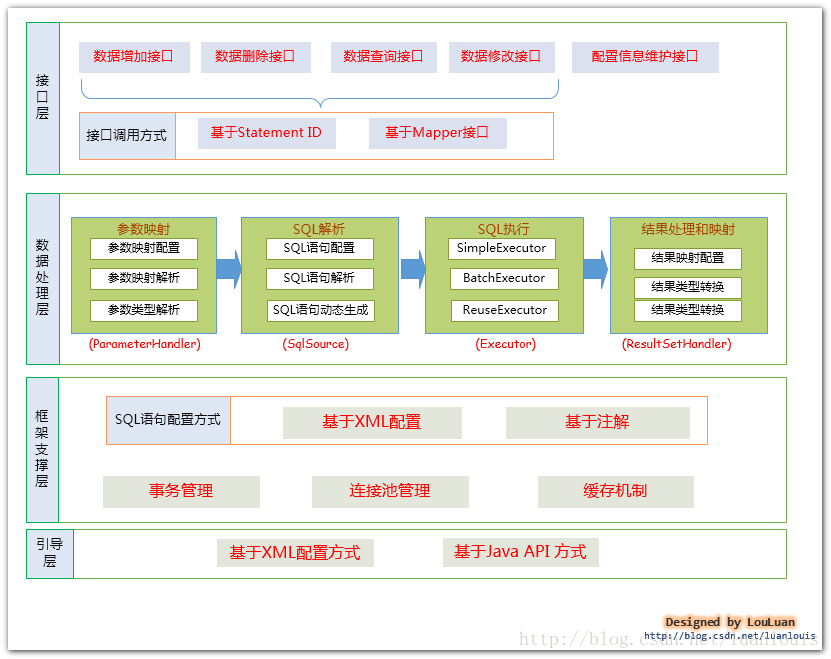
Mybatis和hibernate齐名

持久层技术性能

数据库端（存储过程）>jdbc>mybatis>hibernate

MyBatis是目前非常流行的ORM框架，它的功能很强大，然而其实现却比较简单、优雅。本文主要讲述MyBatis的[**架构**](http://lib.csdn.net/base/architecture)设计思路，并且讨论MyBatis的几个核心部件，然后结合一个select查询实例，深入代码，来探究MyBatis的实现。

**一、MyBatis的框架设计**

       注：上图很大程度上参考了iteye 上的[chenjc\_it](http://chenjc-it.iteye.com/)所写的博文[原理分析之二：框架整体设计](http://chenjc-it.iteye.com/blog/1460990) 中的MyBatis架构体图，chenjc\_it总结的非常好，赞一个！

**1.接口层---和数据库交互的方式**

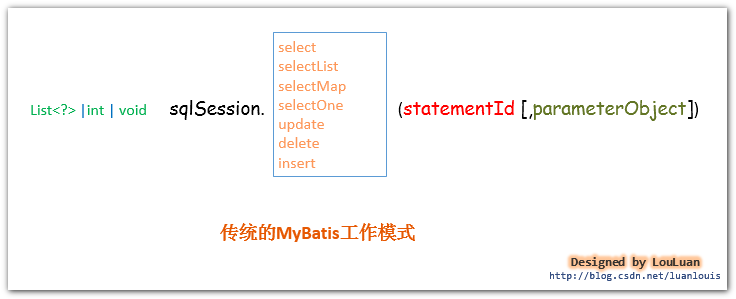
MyBatis和[**数据库**](http://lib.csdn.net/base/mysql)的交互有两种方式：

*a.使用传统的MyBatis提供的API；*

*b. 使用Mapper接口*

**1.1.使用传统的MyBatis提供的API**

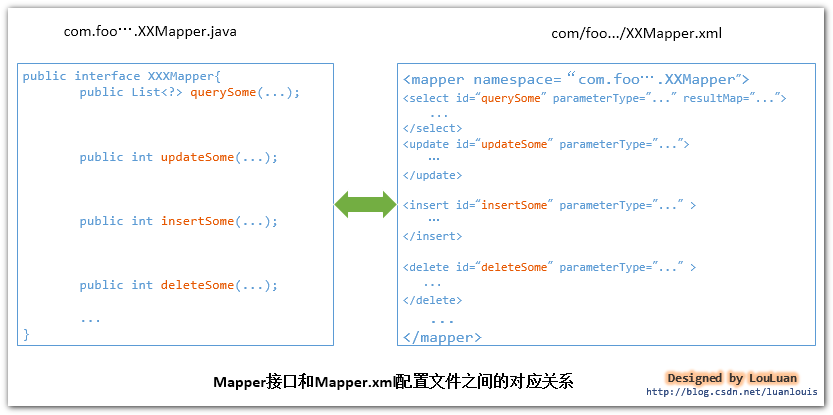
      这是传统的传递**Statement Id** 和查询参数给 **SqlSession** 对象，使用 **SqlSession**对象完成和数据库的交互；**MyBatis**提供了非常方便和简单的API，供用户实现对数据库的增删改查数据操作，以及对数据库连接信息和**MyBatis**自身配置信息的维护操作。



      上述使用**MyBatis**的方法，是创建一个和数据库打交道的**SqlSession**对象，然后根据**Statement Id** 和参数来操作数据库，这种方式固然很简单和实用，但是它不符合面向对象语言的概念和面向接口编程的编程习惯。由于面向接口的编程是面向对象的大趋势，**MyBatis**为了适应这一趋势，增加了第二种使用**MyBatis**支持接口（**Interface**）调用方式。

**1.2. 使用Mapper接口**

**MyBatis**将配置文件中的每一个<mapper> 节点抽象为一个**Mapper**接口，而这个接口中声明的方法和跟<mapper> 节点中的<select|update|delete|insert> 节点项对应，即<select|update|delete|insert> 节点的id值为**Mapper**接口中的方法名称，parameterType 值表示**Mapper**对应方法的入参类型，而resultMap 值则对应了**Mapper**接口表示的返回值类型或者返回结果集的元素类型。



 根据**MyBatis**的配置规范配置好后，通过SqlSession.getMapper(XXXMapper.class) 方法，**MyBatis**会根据相应的接口声明的方法信息，通过动态代理机制生成一个**Mapper**实例，我们使用**Mapper**接口的某一个方法时，**MyBatis**会根据这个方法的方法名和参数类型，确定**Statement Id**，底层还是通过SqlSession.select("statementId",parameterObject);或者SqlSession.update("statementId",parameterObject); 等等来实现对数据库的操作，（*至于这里的动态机制是怎样实现的，我将准备专门一片文章来讨论，敬请关注~*）

**MyBatis**引用**Mapper**接口这种调用方式，纯粹是为了满足面向接口编程的需要。（其实还有一个原因是在于，面向接口的编程，使得用户在接口上可以使用注解来配置SQL语句，这样就可以脱离XML配置文件，实现“0配置”）。

**2.数据处理层**

      数据处理层可以说是**MyBatis**的核心，从大的方面上讲，它要完成三个功能：

*a. 通过传入参数构建动态SQL语句；*

*b. SQL语句的执行以及封装查询结果集成List<E>*

**2.1.参数映射和动态SQL语句生成**

       动态语句生成可以说是MyBatis框架非常优雅的一个设计，**MyBatis**通过传入的参数值，使用**Ognl** 来动态地构造SQL语句，使得**MyBatis**有很强的灵活性和扩展性。

参数映射指的是对于[**Java**](http://lib.csdn.net/base/java)数据类型和**jdbc**数据类型之间的转换：这里有包括两个过程：查询阶段，我们要将java类型的数据，转换成**jdbc**类型的数据，通过 preparedStatement.setXXX() 来设值；另一个就是对**resultset**查询结果集的**jdbcType**数据转换成**java**数据类型。

（*至于具体的MyBatis是如何动态构建SQL语句的，我将准备专门一篇文章来讨论，敬请关注~*）

**2.2. SQL语句的执行以及封装查询结果集成List<E>**

              动态SQL语句生成之后，**MyBatis**将执行SQL语句，并将可能返回的结果集转换成**List<E>** 列表。**MyBatis**在对结果集的处理中，支持结果集关系一对多和多对一的转换，并且有两种支持方式，一种为嵌套查询语句的查询，还有一种是嵌套结果集的查询。

**3. 框架支撑层**

**3.1. 事务管理机制**

          事务管理机制对于**ORM**框架而言是不可缺少的一部分，事务管理机制的质量也是考量一个**ORM**框架是否优秀的一个标准，对于数据管理机制我已经在我的博文[《深入理解mybatis原理》 MyBatis事务管理机制](http://blog.csdn.net/luanlouis/article/details/37992171) 中有非常详细的讨论，感兴趣的读者可以点击查看。

**3.2. 连接池管理机制**

由于创建一个数据库连接所占用的资源比较大， 对于数据吞吐量大和访问量非常大的应用而言，连接池的设计就显得非常重要，对于连接池管理机制我已经在我的博文[《深入理解mybatis原理》 Mybatis数据源与连接池](http://blog.csdn.net/luanlouis/article/details/37671851) 中有非常详细的讨论，感兴趣的读者可以点击查看。

**3.3. 缓存机制**

为了提高数据利用率和减小服务器和数据库的压力，**MyBatis**会对于一些查询提供会话级别的数据缓存，会将对某一次查询，放置到**SqlSession** 中，在允许的时间间隔内，对于完全相同的查询，**MyBatis**会直接将缓存结果返回给用户，而不用再到数据库中查找。（*至于具体的MyBatis缓存机制，我将准备专门一篇文章来讨论，敬请关注~*）

**3. 4. SQL语句的配置方式**

传统的**MyBatis**配置**SQL**语句方式就是使用XML文件进行配置的，但是这种方式不能很好地支持面向接口编程的理念，为了支持面向接口的编程，**MyBatis**引入了**Mapper**接口的概念，面向接口的引入，对使用注解来配置**SQL**语句成为可能，用户只需要在接口上添加必要的注解即可，不用再去配置**XML**文件了，但是，目前的**MyBatis**只是对注解配置**SQL**语句提供了有限的支持，某些高级功能还是要依赖**XML**配置文件配置**SQL**语句。

**4 引导层**

引导层是配置和启动**MyBatis**配置信息的方式。**MyBatis**提供两种方式来引导**MyBatis**：基于XML配置文件的方式和基于[**Java**](http://lib.csdn.net/base/java)**API**的方式，读者可以参考我的另一片博文：[Java Persistence with MyBatis 3(中文版) 第二章 引导MyBatis](http://blog.csdn.net/luanlouis/article/details/35570809)

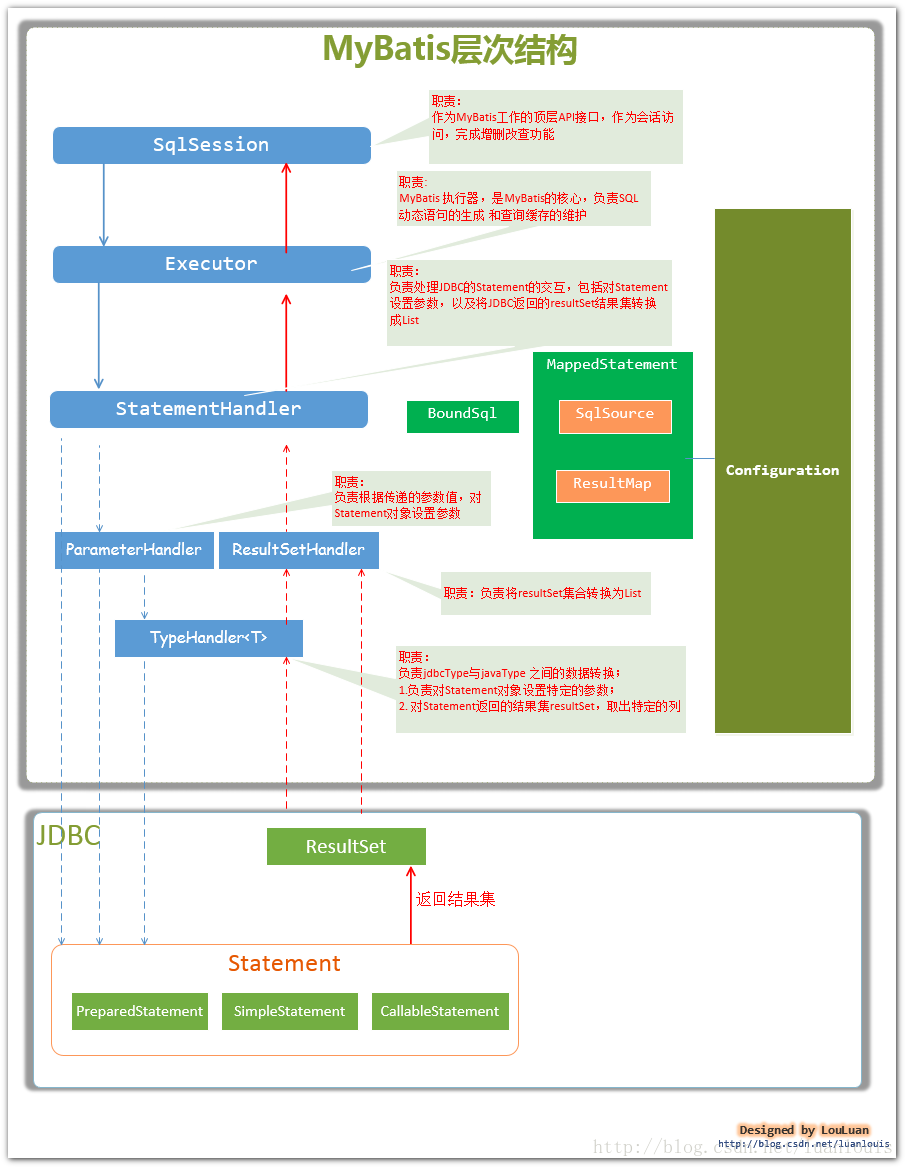
**二、MyBatis的主要构件及其相互关系**

  从MyBatis代码实现的角度来看，MyBatis的主要的核心部件有以下几个：

* **SqlSession**作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必要数据库增删改查功能
* **Executor**MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护
* **StatementHandler**封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数、将Statement结果集转换成List集合。
* **ParameterHandler**负责对用户传递的参数转换成JDBC Statement 所需要的参数，
* **ResultSetHandler**负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；
* **TypeHandler**负责java数据类型和jdbc数据类型之间的映射和转换
* **MappedStatement**MappedStatement维护了一条<select|update|delete|insert>节点的封装，
* **SqlSource**负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回
* **BoundSql**表示动态生成的SQL语句以及相应的参数信息
* **Configuration**MyBatis所有的配置信息都维持在Configuration对象之中。

*（注：这里只是列出了我个人认为属于核心的部件，请读者不要先入为主，认为MyBatis就只有这些部件哦！每个人对MyBatis的理解不同，分析出的结果自然会有所不同，欢迎读者提出质疑和不同的意见，我们共同探讨~）*

它们的关系如下图所示：



**三、从MyBatis一次select 查询语句来分析MyBatis的架构设计**

**一、数据准备（非常熟悉和应用过MyBatis 的读者可以迅速浏览此节即可）**

**1. 准备数据库数据，创建EMPLOYEES表，插入数据：**

**[sql]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

2. --创建一个员工基本信息表
3. **create**  **table** "EMPLOYEES"(
4. "EMPLOYEE\_ID" NUMBER(6) not null,
5. "FIRST\_NAME" VARCHAR2(20),
6. "LAST\_NAME" VARCHAR2(25) not null,
7. "EMAIL" VARCHAR2(25) not null **unique**,
8. "SALARY" NUMBER(8,2),
9. **constraint** "EMP\_EMP\_ID\_PK" **primary** **key** ("EMPLOYEE\_ID")
10. );
11. comment **on** **table** EMPLOYEES **is** '员工信息表';
12. comment **on** **column** EMPLOYEES.EMPLOYEE\_ID **is** '员工id';
13. comment **on** **column** EMPLOYEES.FIRST\_NAME **is** 'first name';
14. comment **on** **column** EMPLOYEES.LAST\_NAME **is** 'last name';
15. comment **on** **column** EMPLOYEES.EMAIL **is** 'email address';
16. comment **on** **column** EMPLOYEES.SALARY **is** 'salary';
18. --添加数据
19. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
20. **values** (100, 'Steven', 'King', 'SKING', 24000.00);
22. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
23. **values** (101, 'Neena', 'Kochhar', 'NKOCHHAR', 17000.00);
25. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
26. **values** (102, 'Lex', 'De Haan', 'LDEHAAN', 17000.00);
28. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
29. **values** (103, 'Alexander', 'Hunold', 'AHUNOLD', 9000.00);
31. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
32. **values** (104, 'Bruce', 'Ernst', 'BERNST', 6000.00);
34. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
35. **values** (105, 'David', 'Austin', 'DAUSTIN', 4800.00);
37. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
38. **values** (106, 'Valli', 'Pataballa', 'VPATABAL', 4800.00);
40. **insert** **into** EMPLOYEES (EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY)
41. **values** (107, 'Diana', 'Lorentz', 'DLORENTZ', 4200.00);

**2. 配置Mybatis的配置文件，命名为mybatisConfig.xml：**

**[html]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **<?xml** version="1.0" encoding="utf-8"**?>**
2. <!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
3. "http://mybatis.org/dtd/mybatis-3-config.dtd"**>**
4. **<configuration>**
5. **<environments** default="development"**>**
6. **<environment** id="development"**>**
7. **<transactionManager** type="JDBC" **/>**
8. **<dataSource** type="POOLED"**>**
9. **<property** name="driver" value="oracle.jdbc.driver.OracleDriver" **/>**
10. **<property** name="url" value="jdbc:oracle:thin:@localhost:1521:xe" **/>**
11. **<property** name="username" value="louis" **/>**
12. **<property** name="password" value="123456" **/>**
13. **</dataSource>**
14. **</environment>**
15. **</environments>**
16. **<mappers>**
17. **<mapper**  resource="com/louis/mybatis/domain/EmployeesMapper.xml"**/>**
18. **</mappers>**
19. **</configuration>**

3.     创建Employee实体Bean 以及配置Mapper配置文件

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **package** com.louis.mybatis.model;
3. **import** java.math.BigDecimal;
5. **public** **class** Employee {
6. **private** Integer employeeId;
8. **private** String firstName;
10. **private** String lastName;
12. **private** String email;
14. **private** BigDecimal salary;
16. **public** Integer getEmployeeId() {
17. **return** employeeId;
18. }
20. **public** **void** setEmployeeId(Integer employeeId) {
21. **this**.employeeId = employeeId;
22. }
24. **public** String getFirstName() {
25. **return** firstName;
26. }
28. **public** **void** setFirstName(String firstName) {
29. **this**.firstName = firstName;
30. }
32. **public** String getLastName() {
33. **return** lastName;
34. }
36. **public** **void** setLastName(String lastName) {
37. **this**.lastName = lastName;
38. }
40. **public** String getEmail() {
41. **return** email;
42. }
44. **public** **void** setEmail(String email) {
45. **this**.email = email;
46. }
48. **public** BigDecimal getSalary() {
49. **return** salary;
50. }
52. **public** **void** setSalary(BigDecimal salary) {
53. **this**.salary = salary;
54. }
55. }

**[html]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **<?xml** version="1.0" encoding="UTF-8" **?>**
2. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd" **>**
3. **<mapper** namespace="com.louis.mybatis.dao.EmployeesMapper" **>**
5. **<resultMap** id="BaseResultMap" type="com.louis.mybatis.model.Employee" **>**
6. **<id** column="EMPLOYEE\_ID" property="employeeId" jdbcType="DECIMAL" **/>**
7. **<result** column="FIRST\_NAME" property="firstName" jdbcType="VARCHAR" **/>**
8. **<result** column="LAST\_NAME" property="lastName" jdbcType="VARCHAR" **/>**
9. **<result** column="EMAIL" property="email" jdbcType="VARCHAR" **/>**
10. **<result** column="SALARY" property="salary" jdbcType="DECIMAL" **/>**
11. **</resultMap>**
13. **<select** id="selectByPrimaryKey" resultMap="BaseResultMap" parameterType="java.lang.Integer" **>**
14. select
15. EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY
16. from LOUIS.EMPLOYEES
17. where EMPLOYEE\_ID = #{employeeId,jdbcType=DECIMAL}
18. **</select>**
19. **</mapper>**

**4. 创建eclipse 或者myeclipse 的maven项目，maven配置如下：**

**[html]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2. xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>**
3. **<modelVersion>**4.0.0**</modelVersion>**
5. **<groupId>**batis**</groupId>**
6. **<artifactId>**batis**</artifactId>**
7. **<version>**0.0.1-SNAPSHOT**</version>**
8. **<packaging>**jar**</packaging>**
10. **<name>**batis**</name>**
11. **<url>**http://maven.apache.org**</url>**
13. **<properties>**
14. **<project.build.sourceEncoding>**UTF-8**</project.build.sourceEncoding>**
15. **</properties>**
17. **<dependencies>**
18. **<dependency>**
19. **<groupId>**junit**</groupId>**
20. **<artifactId>**junit**</artifactId>**
21. **<version>**3.8.1**</version>**
22. **<scope>**test**</scope>**
23. **</dependency>**
25. **<dependency>**
26. **<groupId>**org.mybatis**</groupId>**
27. **<artifactId>**mybatis**</artifactId>**
28. **<version>**3.2.7**</version>**
29. **</dependency>**
31. **<dependency>**
32. **<groupId>**com.oracle**</groupId>**
33. **<artifactId>**ojdbc14**</artifactId>**
34. **<version>**10.2.0.4.0**</version>**
35. **</dependency>**
37. **</dependencies>**
38. **</project>**

**5. 客户端代码：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **package** com.louis.mybatis.test;
3. **import** java.io.InputStream;
4. **import** java.util.HashMap;
5. **import** java.util.List;
6. **import** java.util.Map;
8. **import** org.apache.ibatis.io.Resources;
9. **import** org.apache.ibatis.session.SqlSession;
10. **import** org.apache.ibatis.session.SqlSessionFactory;
11. **import** org.apache.ibatis.session.SqlSessionFactoryBuilder;
13. **import** com.louis.mybatis.model.Employee;
15. /\*\*
16. \* SqlSession 简单查询演示类
17. \* @author louluan
18. \*/
19. **public** **class** SelectDemo {
21. **public** **static** **void** main(String[] args) **throws** Exception {
22. /\*
23. \* 1.加载mybatis的配置文件，初始化mybatis，创建出SqlSessionFactory，是创建SqlSession的工厂
24. \* 这里只是为了演示的需要，SqlSessionFactory临时创建出来，在实际的使用中，SqlSessionFactory只需要创建一次，当作单例来使用
25. \*/
26. InputStream inputStream = Resources.getResourceAsStream("mybatisConfig.xml");
27. SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();
28. SqlSessionFactory factory = builder.build(inputStream);
30. //2. 从SqlSession工厂 SqlSessionFactory中创建一个SqlSession，进行数据库操作
31. SqlSession sqlSession = factory.openSession();
33. //3.使用SqlSession查询
34. Map<String,Object> params = **new** HashMap<String,Object>();
36. params.put("min\_salary",10000);
37. //a.查询工资低于10000的员工
38. List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);
39. //b.未传最低工资，查所有员工
40. List<Employee> result1 = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary");
41. System.out.println("薪资低于10000的员工数："+result.size());
42. //~output :   查询到的数据总数：5
43. System.out.println("所有员工数: "+result1.size());
44. //~output :  所有员工数: 8
45. }
47. }

**二、SqlSession 的工作过程分析：**

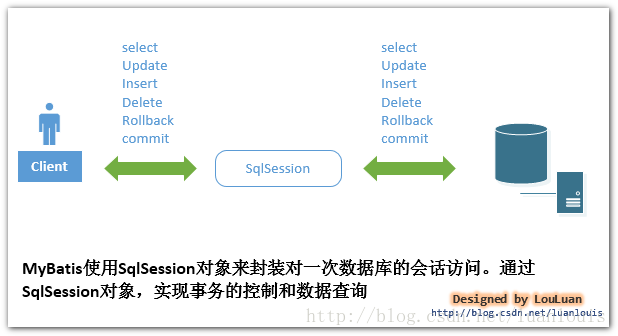
**1. 开启一个数据库访问会话---创建SqlSession对象：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. SqlSession sqlSession = factory.openSession();

           MyBatis封装了对数据库的访问，把对数据库的会话和事务控制放到了SqlSession对象中。



**2. 为SqlSession传递一个配置的Sql语句 的Statement Id和参数，然后返回结果：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. List<Employee> result = sqlSession.selectList("com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary",params);

上述的"com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary"，是配置在**EmployeesMapper.xml** 的Statement ID，params 是传递的查询参数。

让我们来看一下sqlSession.selectList()方法的定义：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **public** <E> List<E> selectList(String statement, Object parameter) {
2. **return** **this**.selectList(statement, parameter, RowBounds.DEFAULT);
3. }
5. **public** <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {
6. **try** {
7. //1.根据Statement Id，在mybatis 配置对象Configuration中查找和配置文件相对应的MappedStatement
8. MappedStatement ms = configuration.getMappedStatement(statement);
9. //2. 将查询任务委托给MyBatis 的执行器 Executor
10. List<E> result = executor.query(ms, wrapCollection(parameter), rowBounds, Executor.NO\_RESULT\_HANDLER);
11. **return** result;
12. } **catch** (Exception e) {
13. **throw** ExceptionFactory.wrapException("Error querying database.  Cause: " + e, e);
14. } **finally** {
15. ErrorContext.instance().reset();
16. }
17. }

**MyBatis**在初始化的时候，会将**MyBatis**的配置信息全部加载到内存中，使用**org.apache.ibatis.session.Configuration**实例来维护。使用者可以使用**sqlSession.getConfiguration()**方法来获取。MyBatis的配置文件中配置信息的组织格式和内存中对象的组织格式几乎完全对应的。上述例子中的

**[html]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. **<select** id="selectByMinSalary" resultMap="BaseResultMap" parameterType="java.util.Map" **>**
2. select
3. EMPLOYEE\_ID, FIRST\_NAME, LAST\_NAME, EMAIL, SALARY
4. from LOUIS.EMPLOYEES
5. **<if** test="min\_salary != null"**>**
6. where SALARY **<** #{min\_salary,jdbcType=DECIMAL}
7. **</if>**
8. **</select>**

加载到内存中会生成一个对应的**MappedStatement**对象，然后会以**key**="com.louis.mybatis.dao.EmployeesMapper.selectByMinSalary" ，**value**为**MappedStatement**对象的形式维护到**Configuration**的一个**Map**中。当以后需要使用的时候，只需要通过**Id**值来获取就可以了。

从上述的代码中我们可以看到SqlSession的职能是：

**SqlSession**根据**Statement ID**, 在**mybatis**配置对象**Configuration**中获取到对应的**MappedStatement**对象，然后调用**mybatis**执行器来执行具体的操作。

**3.MyBatis执行器Executor根据SqlSession传递的参数执行query()方法（由于代码过长，读者只需阅读我注释的地方即可）：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \* BaseExecutor 类部分代码
3. \*
4. \*/
5. **public** <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler) **throws** SQLException {
7. // 1.根据具体传入的参数，动态地生成需要执行的SQL语句，用BoundSql对象表示
8. BoundSql boundSql = ms.getBoundSql(parameter);
9. // 2.为当前的查询创建一个缓存Key
10. CacheKey key = createCacheKey(ms, parameter, rowBounds, boundSql);
11. **return** query(ms, parameter, rowBounds, resultHandler, key, boundSql);
12. }
14. @SuppressWarnings("unchecked")
15. **public** <E> List<E> query(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) **throws** SQLException {
16. ErrorContext.instance().resource(ms.getResource()).activity("executing a query").object(ms.getId());
17. **if** (closed) **throw** **new** ExecutorException("Executor was closed.");
18. **if** (queryStack == 0 && ms.isFlushCacheRequired()) {
19. clearLocalCache();
20. }
21. List<E> list;
22. **try** {
23. queryStack++;
24. list = resultHandler == **null** ? (List<E>) localCache.getObject(key) : **null**;
25. **if** (list != **null**) {
26. handleLocallyCachedOutputParameters(ms, key, parameter, boundSql);
27. } **else** {
28. // 3.缓存中没有值，直接从数据库中读取数据
29. list = queryFromDatabase(ms, parameter, rowBounds, resultHandler, key, boundSql);
30. }
31. } **finally** {
32. queryStack--;
33. }
34. **if** (queryStack == 0) {
35. **for** (DeferredLoad deferredLoad : deferredLoads) {
36. deferredLoad.load();
37. }
38. deferredLoads.clear(); // issue #601
39. **if** (configuration.getLocalCacheScope() == LocalCacheScope.STATEMENT) {
40. clearLocalCache(); // issue #482
41. }
42. }
43. **return** list;
44. }
45. **private** <E> List<E> queryFromDatabase(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, CacheKey key, BoundSql boundSql) **throws** SQLException {
46. List<E> list;
47. localCache.putObject(key, EXECUTION\_PLACEHOLDER);
48. **try** {
50. //4. 执行查询，返回List 结果，然后    将查询的结果放入缓存之中
51. list = doQuery(ms, parameter, rowBounds, resultHandler, boundSql);
52. } **finally** {
53. localCache.removeObject(key);
54. }
55. localCache.putObject(key, list);
56. **if** (ms.getStatementType() == StatementType.CALLABLE) {
57. localOutputParameterCache.putObject(key, parameter);
58. }
59. **return** list;
60. }

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \*
3. \*SimpleExecutor类的doQuery()方法实现
4. \*
5. \*/
6. **public** <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) **throws** SQLException {
7. Statement stmt = **null**;
8. **try** {
9. Configuration configuration = ms.getConfiguration();
10. //5. 根据既有的参数，创建StatementHandler对象来执行查询操作
11. StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);
12. //6. 创建java.Sql.Statement对象，传递给StatementHandler对象
13. stmt = prepareStatement(handler, ms.getStatementLog());
14. //7. 调用StatementHandler.query()方法，返回List结果集
15. **return** handler.<E>query(stmt, resultHandler);
16. } **finally** {
17. closeStatement(stmt);
18. }
19. }

上述的**Executor.query()**方法几经转折，最后会创建一个**StatementHandler**对象，然后将必要的参数传递给**StatementHandler**，使用**StatementHandler**来完成对数据库的查询，最终返回**List**结果集。

从上面的代码中我们可以看出，**Executor**的功能和作用是：

**(1、根据传递的参数，完成SQL语句的动态解析，生成BoundSql对象，供StatementHandler使用；**

**(2、为查询创建缓存，以提高性能（具体它的缓存机制不是本文的重点，我会单独拿出来跟大家探讨，感兴趣的读者可以关注我的其他博文）；**

**(3、创建JDBC的Statement连接对象，传递给StatementHandler对象，返回List查询结果。**

**4. StatementHandler对象负责设置Statement对象中的查询参数、处理JDBC返回的resultSet，将resultSet加工为List 集合返回：**

      接着上面的**Executor**第六步，看一下：**prepareStatement()** 方法的实现：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \*
3. \*SimpleExecutor类的doQuery()方法实现
4. \*
5. \*/
6. **public** <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) **throws** SQLException { Statement stmt = **null**; **try** { Configuration configuration = ms.getConfiguration(); StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql); // 1.准备Statement对象，并设置Statement对象的参数 stmt = prepareStatement(handler, ms.getStatementLog()); // 2. StatementHandler执行query()方法，返回List结果 return handler.<E>query(stmt, resultHandler); } finally { closeStatement(stmt); } }
8. **private** Statement prepareStatement(StatementHandler handler, Log statementLog) **throws** SQLException {
9. Statement stmt;
10. Connection connection = getConnection(statementLog);
11. stmt = handler.prepare(connection);
12. //对创建的Statement对象设置参数，即设置SQL 语句中 ? 设置为指定的参数
13. handler.parameterize(stmt);
14. **return** stmt;
15. }

      以上我们可以总结**StatementHandler**对象主要完成两个工作：

         (1. 对于**JDBC**的**PreparedStatement**类型的对象，创建的过程中，我们使用的是**SQL**语句字符串会包含 若干个? 占位符，我们其后再对占位符进行设值。

**StatementHandler**通过parameterize(statement)方法对Statement进行设值；

         (2.**StatementHandler**通过List<E> query(Statement statement, ResultHandler resultHandler)方法来完成执行Statement，和将Statement对象返回的resultSet封装成**List**；

**5.   StatementHandler 的parameterize(statement) 方法的实现：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \*   StatementHandler 类的parameterize(statement) 方法实现
3. \*/
4. **public** **void** parameterize(Statement statement) **throws** SQLException {
5. //使用ParameterHandler对象来完成对Statement的设值
6. parameterHandler.setParameters((PreparedStatement) statement);
7. }

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \*
3. \*ParameterHandler类的setParameters(PreparedStatement ps) 实现
4. \* 对某一个Statement进行设置参数
5. \*/
6. **public** **void** setParameters(PreparedStatement ps) **throws** SQLException {
7. ErrorContext.instance().activity("setting parameters").object(mappedStatement.getParameterMap().getId());
8. List<ParameterMapping> parameterMappings = boundSql.getParameterMappings();
9. **if** (parameterMappings != **null**) {
10. **for** (**int** i = 0; i < parameterMappings.size(); i++) {
11. ParameterMapping parameterMapping = parameterMappings.get(i);
12. **if** (parameterMapping.getMode() != ParameterMode.OUT) {
13. Object value;
14. String propertyName = parameterMapping.getProperty();
15. **if** (boundSql.hasAdditionalParameter(propertyName)) { // issue #448 ask first for additional params
16. value = boundSql.getAdditionalParameter(propertyName);
17. } **else** **if** (parameterObject == **null**) {
18. value = **null**;
19. } **else** **if** (typeHandlerRegistry.hasTypeHandler(parameterObject.getClass())) {
20. value = parameterObject;
21. } **else** {
22. MetaObject metaObject = configuration.newMetaObject(parameterObject);
23. value = metaObject.getValue(propertyName);
24. }
26. // 每一个Mapping都有一个TypeHandler，根据TypeHandler来对preparedStatement进行设置参数
27. TypeHandler typeHandler = parameterMapping.getTypeHandler();
28. JdbcType jdbcType = parameterMapping.getJdbcType();
29. **if** (value == **null** && jdbcType == **null**) jdbcType = configuration.getJdbcTypeForNull();
30. // 设置参数
31. typeHandler.setParameter(ps, i + 1, value, jdbcType);
32. }
33. }
34. }
35. }

从上述的代码可以看到,StatementHandler 的parameterize(Statement) 方法调用了 ParameterHandler的setParameters(statement) 方法，

ParameterHandler的setParameters(Statement)方法负责 根据我们输入的参数，对statement对象的 ? 占位符处进行赋值。

**6.   StatementHandler 的List<E> query(Statement statement, ResultHandler resultHandler)方法的实现：**

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \* PreParedStatement类的query方法实现
3. \*/
4. **public** <E> List<E> query(Statement statement, ResultHandler resultHandler) **throws** SQLException {
5. // 1.调用preparedStatemnt。execute()方法，然后将resultSet交给ResultSetHandler处理
6. PreparedStatement ps = (PreparedStatement) statement;
7. ps.execute();
8. //2. 使用ResultHandler来处理ResultSet
9. **return** resultSetHandler.<E> handleResultSets(ps);
10. }

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. /\*\*
2. \*ResultSetHandler类的handleResultSets()方法实现
3. \*
4. \*/
5. **public** List<Object> handleResultSets(Statement stmt) **throws** SQLException {
6. **final** List<Object> multipleResults = **new** ArrayList<Object>();
8. **int** resultSetCount = 0;
9. ResultSetWrapper rsw = getFirstResultSet(stmt);
11. List<ResultMap> resultMaps = mappedStatement.getResultMaps();
12. **int** resultMapCount = resultMaps.size();
13. validateResultMapsCount(rsw, resultMapCount);
15. **while** (rsw != **null** && resultMapCount > resultSetCount) {
16. ResultMap resultMap = resultMaps.get(resultSetCount);
18. //将resultSet
19. handleResultSet(rsw, resultMap, multipleResults, **null**);
20. rsw = getNextResultSet(stmt);
21. cleanUpAfterHandlingResultSet();
22. resultSetCount++;
23. }
25. String[] resultSets = mappedStatement.getResulSets();
26. **if** (resultSets != **null**) {
27. **while** (rsw != **null** && resultSetCount < resultSets.length) {
28. ResultMapping parentMapping = nextResultMaps.get(resultSets[resultSetCount]);
29. **if** (parentMapping != **null**) {
30. String nestedResultMapId = parentMapping.getNestedResultMapId();
31. ResultMap resultMap = configuration.getResultMap(nestedResultMapId);
32. handleResultSet(rsw, resultMap, **null**, parentMapping);
33. }
34. rsw = getNextResultSet(stmt);
35. cleanUpAfterHandlingResultSet();
36. resultSetCount++;
37. }
38. }
40. **return** collapseSingleResultList(multipleResults);
41. }

从上述代码我们可以看出，StatementHandler 的List<E> query(Statement statement, ResultHandler resultHandler)方法的实现，是调用了ResultSetHandler的handleResultSets(Statement) 方法。ResultSetHandler的handleResultSets(Statement) 方法会将Statement语句执行后生成的resultSet 结果集转换成List<E> 结果集：

**[java]** [view plain](http://blog.csdn.net/luanlouis/article/details/40422941) [copy](http://blog.csdn.net/luanlouis/article/details/40422941)

[print?](http://blog.csdn.net/luanlouis/article/details/40422941)

1. //
2. // DefaultResultSetHandler 类的handleResultSets(Statement stmt)实现
3. //HANDLE RESULT SETS
4. //
6. **public** List<Object> handleResultSets(Statement stmt) **throws** SQLException {
7. **final** List<Object> multipleResults = **new** ArrayList<Object>();
9. **int** resultSetCount = 0;
10. ResultSetWrapper rsw = getFirstResultSet(stmt);
12. List<ResultMap> resultMaps = mappedStatement.getResultMaps();
13. **int** resultMapCount = resultMaps.size();
14. validateResultMapsCount(rsw, resultMapCount);
16. **while** (rsw != **null** && resultMapCount > resultSetCount) {
17. ResultMap resultMap = resultMaps.get(resultSetCount);
19. //将resultSet
20. handleResultSet(rsw, resultMap, multipleResults, **null**);
21. rsw = getNextResultSet(stmt);
22. cleanUpAfterHandlingResultSet();
23. resultSetCount++;
24. }
26. String[] resultSets = mappedStatement.getResulSets();
27. **if** (resultSets != **null**) {
28. **while** (rsw != **null** && resultSetCount < resultSets.length) {
29. ResultMapping parentMapping = nextResultMaps.get(resultSets[resultSetCount]);
30. **if** (parentMapping != **null**) {
31. String nestedResultMapId = parentMapping.getNestedResultMapId();
32. ResultMap resultMap = configuration.getResultMap(nestedResultMapId);
33. handleResultSet(rsw, resultMap, **null**, parentMapping);
34. }
35. rsw = getNextResultSet(stmt);
36. cleanUpAfterHandlingResultSet();
37. resultSetCount++;
38. }
39. }
41. **return** collapseSingleResultList(multipleResults);
42. }