MyBatis

王新阳
wxyyuppie@bjfu.edu.cn

主要内容

- MyBatis简介与工作原理
- MyBatis实例演示
- 配置文件说明
- 映射器

MyBatis
是什么?

MyBatis介绍

MyBatis简介

MyBatis本是apache的一个开源项目iBatis，2010年这个项目由apache software foundation迁移到了google code，并且改名为MyBatis。

- ✓ 一个基于Java的持久层框架；
- ✓ 消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索；
- ✓ 将接口和Java的POJOs（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

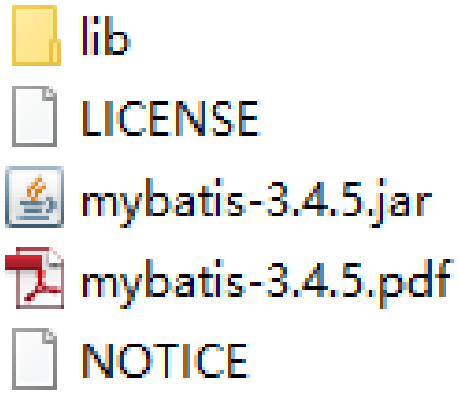
MyBatis简介

主要思想：将程序中的大量sql语句剥离出来，配置在配置文件中，实现sql的灵活配置。

好处：将sql与程序代码分离，可以在不修改程序代码的情况下，直接在配置文件中修改sql。

MyBatis环境的构建

MyBatis的3.4.5版本可以通过“https://github.com/mybatis/mybatis-3/releases”网址下载。下载时只需选择mybatis-3.4.5.zip即可，解压后得到如图所示的目录。



mybatis-3.4.5.jar是MyBatis的核心包，mybatis-3.4.5.pdf是MyBatis的使用手册，lib文件夹下的JAR是MyBatis的依赖包。

使用MyBatis框架时，需要将它的核心包和依赖包引入到应用程序中。如果是Web应用，只需将核心包和依赖包复制到/WEB-INF/lib目录中。

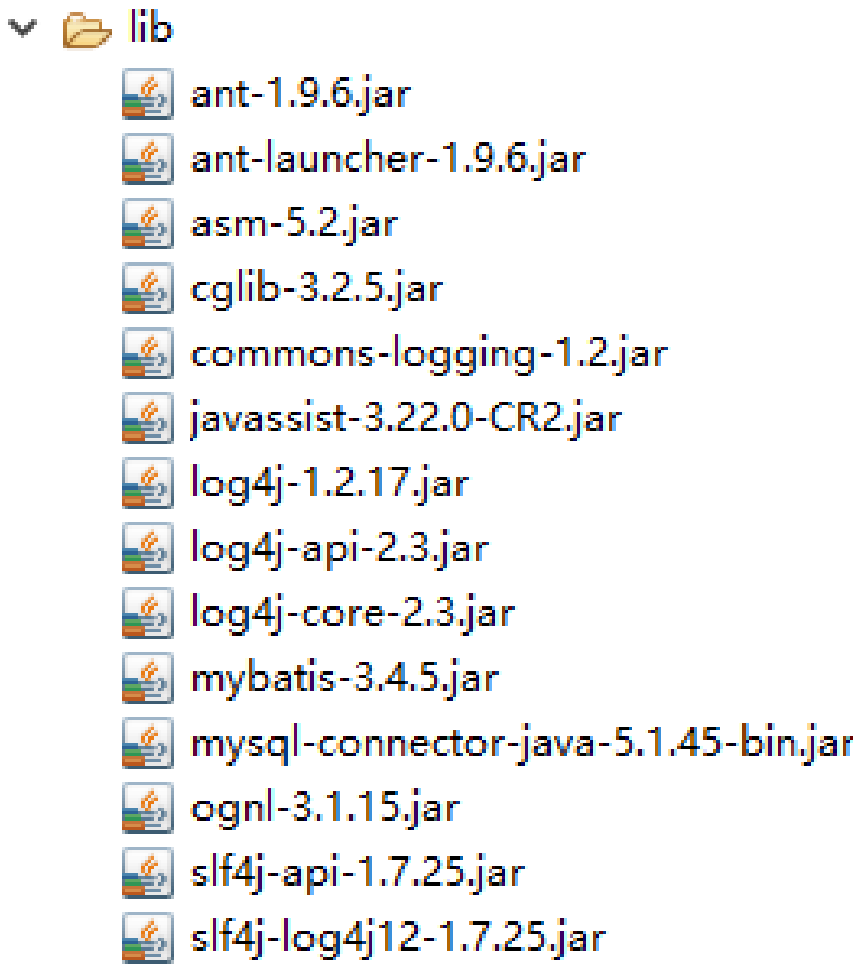
使用Eclipse开发MyBatis入门程序

通过一个实例演示MyBatis入门程序

1. 创建Web应用，并添加相关JAR包
2. 创建日志文件
3. 创建持久化类
4. 创建映射文件
5. 创建MyBatis的配置文件
6. 创建测试类

见工程ch6

创建Web应用，并添加相关JAR包



创建日志文件

MyBatis默认使用log4j输出日志信息，如果开发者需要查看控制台输出的SQL语句，那么需要在classpath路径下配置其日志文件。在应用的src目录下创建log4j.properties文件，内容如下：

```
# Global logging configuration
log4j.rootLogger=ERROR, stdout
# MyBatis logging configuration...
log4j.logger.com.mybatis=DEBUG
# Console output...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

日志文件中配置了全局的日志配置、MyBatis的日志配置和控制台输出，其中MyBatis的日志配置用于将com.mybatis包下所有类的日志记录级别设置为DEBUG。该配置文件内容不需要开发者全部手写，可以从MyBatis使用手册中Logging小节复制，然后进行简单修改。

创建持久化类

在src目录下，创建一个名为com.mybatis.po包，在该包中创建持久化类MyUser。类中声明的属性与数据表user的字段一致。

创建映射文件

在src目录下，创建一个名为com.mybatis.mapper包，在该包中创建映射文件UserMapper.xml。

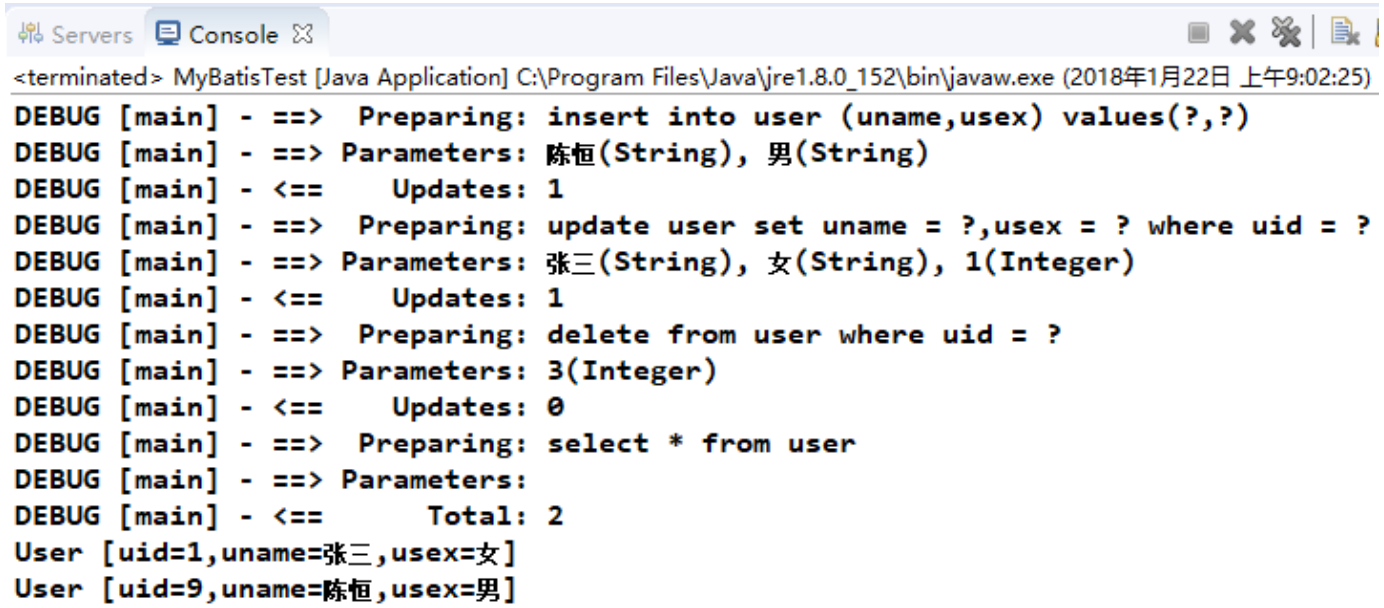
上述映射文件中，<mapper>元素是配置文件的根元素，它包含了一个namespace属性，该属性值通常设置为“包名+SQL映射文件名”，指定了唯一的命名空间。子元素<select>、<insert>、<update>以及<delete>中的信息是用于执行查询、添加、修改以及删除操作的配置。在定义的SQL语句中，“#{}”表示一个占位符，相当于“?”，而“#{uid}”表示该占位符待接收参数的名称为uid。

创建MyBatis的配置文件

在src目录下，创建MyBatis的核心配置文件mybatis-config.xml。在该文件中，配置了数据库环境和映射文件的位置。

创建测试类

在src目录下，创建一个名为com.mybatis.test包，在该包中创建MyBatisTest测试类。在测试类中，首先使用输入流读取配置文件，然后根据配置信息构建SqlSessionFactory对象。接下来通过SqlSessionFactory对象创建SqlSession对象，并使用SqlSession对象的方法执行数据库操作。



为什么要用MyBatis

JDBC缺点

```
ResultSet rs = null;
try {
    // 加载驱动
    Class.forName("com.mysql.jdbc.Driver");
    // 获取连接
    String url = "jdbc:mysql://127.0.0.1:3306/mybatis-328";
    String user = "root";
    String password = "root";
    conn = DriverManager.getConnection(url, user, password);
    // 获取statement，preparedStatement
    String sql = "select * from tb_user where id=?";
    preparedStatement = conn.prepareStatement(sql);
    // 设置参数
    preparedStatement.setLong(1, 1l);
    // 执行查询，获取结果集
    rs = preparedStatement.executeQuery();
    // 处理结果集
    while (rs.next()){
        System.out.println(rs.getString("user_name"));
        System.out.println(rs.getString("name"));
        System.out.println(rs.getInt("age"));
    }
} finally {
    // 关闭连接，释放资源
    if(rs!=null){
        rs.close();
    }
    if(preparedStatement!=null){
        preparedStatement.close();
    }
```

1.每次加载连接
2.驱动名称硬编码

1.每次都要获取连接
2.连接信息硬编码

1.sql和java代码耦合

1.参数类型需要手动判断
2.需要判断下标
3.手动设置参数

1.结果集中的数据类型需要手动判断
2.下标或列名需要手动判断

1.每次都要打开或关闭连接，浪费资源

与MyBatis的对比

```java
ResultSet rs = null;
try {
    // 加载驱动
    Class.forName("com.mysql.jdbc.Driver");
    // 获取连接
    String url = "jdbc:mysql://127.0.0.1:3306/mybatis-328";
    String user = "root";
    String password = "root";
    conn = DriverManager.getConnection(url, user, password);
    // 获取statement，preparedStatement
    String sql = "select * from tb_user where id=?";
    preparedStatement = conn.prepareStatement(sql);
    // 设置参数
    preparedStatement.setLong(1, 1l);
    // 执行查询，获取结果集
    rs = preparedStatement.executeQuery();
    // 处理结果集
    while (rs.next()){
        System.out.println(rs.getString("user_name"));
        System.out.println(rs.getString("name"));
        System.out.println(rs.getInt("age"));
    }
} finally {
    // 关闭连接，释放资源
    if(rs!=null){
        rs.close();
    }
    if(preparedStatement!=null){
        preparedStatement.close();
    }
```

1.每次加载连接
2.驱动名称硬编码
1.每次都要获取连接
2.连接信息硬编码
1.sql和java代码耦合
1.参数类型需要手动判断
2.需要判断下标
3.手动设置参数
1.结果集中的数据类型需要手动判断
2.下标或列名需要手动判断
1.每次都要打开或关闭连接，浪费资源

```xml
<dataSource type="POOLED">
    <!-- MySQL数据库驱动 -->
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <!-- 连接数据库的URL -->
    <property name="url" value="jdbc:mysql://localhost:3306/java web?characterEncoding=utf8"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</dataSource>
```

```xml
<mapper namespace="com.mybatis.mapper.UserMapper">
    <!-- 根据uid查询一个用户信息 -->
    <select id="selectUserById" parameterType="Integer"
        resultType="com.mybatis.po.MyUser">
        select * from user where uid = #{uid}
    </select>
```

```java
MyUser mu = ss.selectOne("com.mybatis.mapper.UserMapper.selectUserById", 1);
```

MyBatis优点

1). 优化获取和释放

- 解决了数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能的问题。
- 解决方法：统一从DataSource里面获取数据库连接，将DataSource的具体实现通过让用户配置来应对变化。在SqlMapConfig.xml中配置数据链接池，使用连接池管理数据库链接。

2). SQL统一管理，对数据库进行存取操作

- 使用JDBC对数据库进行操作时，SQL查询语句分布在各个Java类中，可读性差，不利于维护，当我们修改Java类中的SQL语句时要重新进行编译。
- Mybatis可以把SQL语句放在配置文件中统一进行管理，以后修改配置文件，也不需要重新编译部署。
- 解决方法：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

MyBatis优点

3). 传入参数映射和生成动态SQL语句

在查询中可能需要根据一些属性进行组合查询，比如进行商品查询，可以根据商品名称进行查询，也可以根据发货地进行查询，或者两者组合查询。如果使用JDBC进行查询，这样就需要写多条SQL语句，且无法灵活应对用户多变的查询需求。


```
<select id="getCountByInfo"parameterType="User" resultType="int">
    select count(*) from user
    <where>
      <iftest="nickname!=null">
        andnickname = #{nickname}
      </if>
      <iftest="email!=null">
        andemail = #{email}
      </if>
    </where>

</select>
```

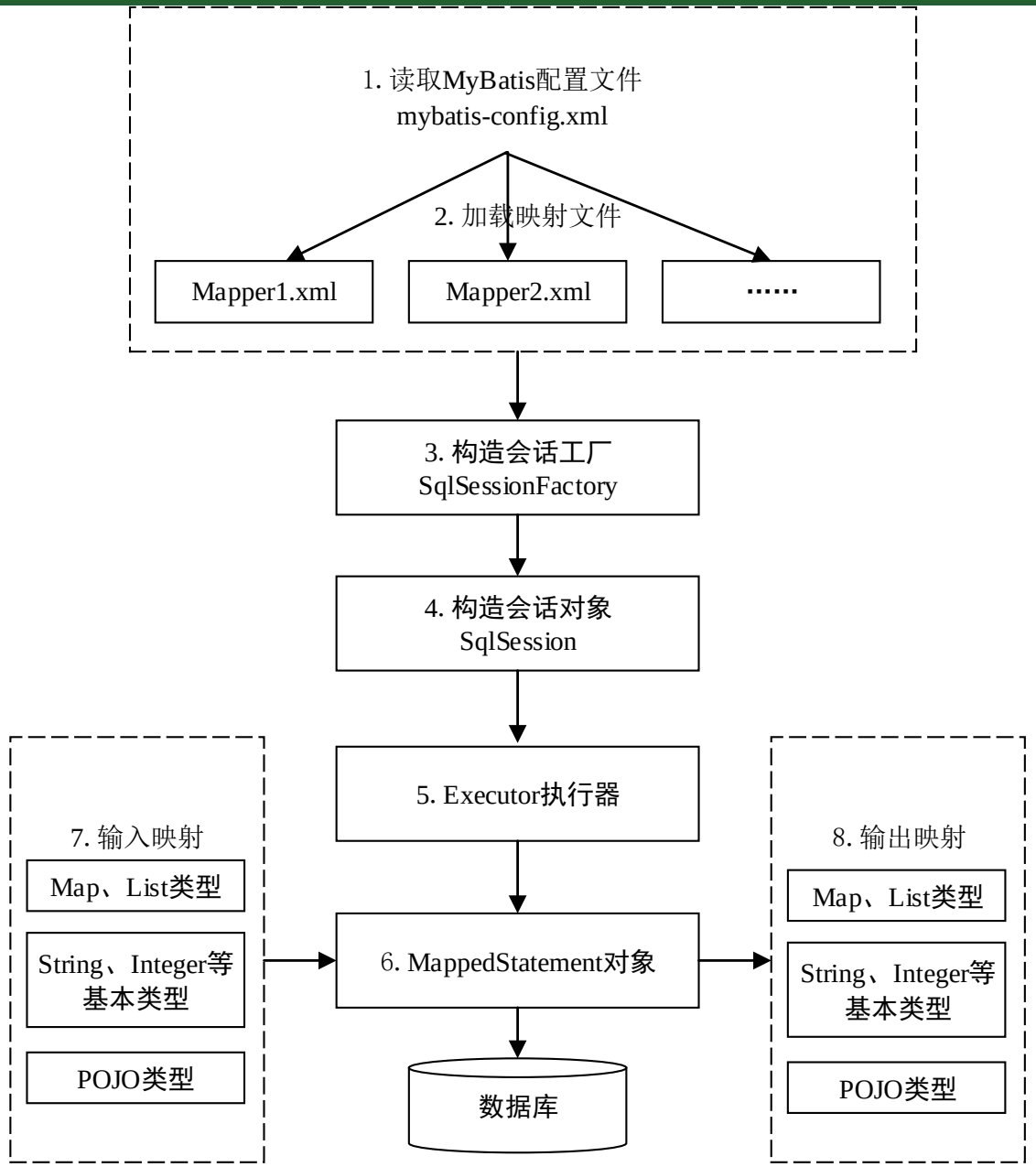
MyBatis优点

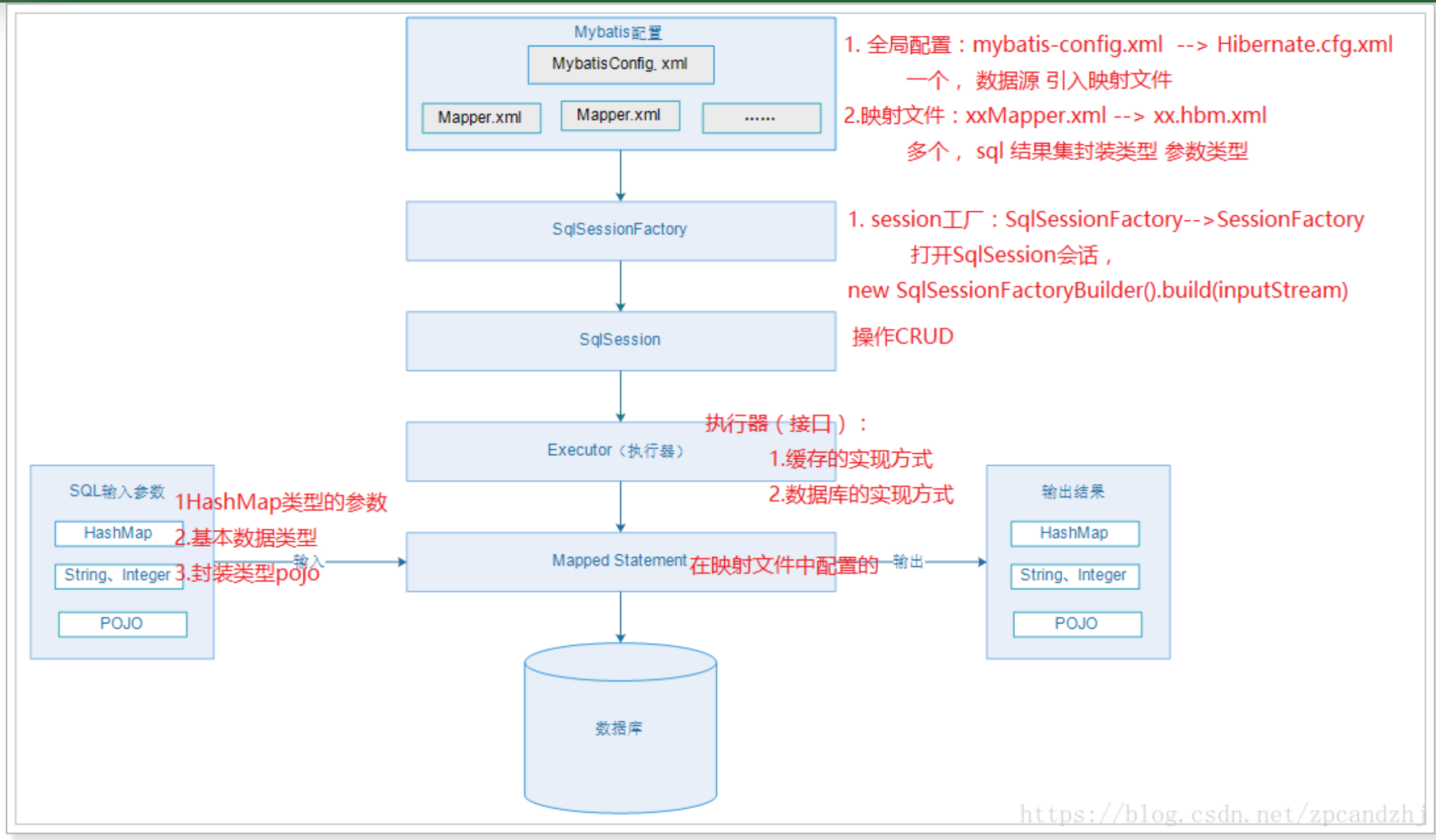
4）能够对结果集进行映射

在使用JDBC进行查询时，返回一个结果集ResultSet，要从结果集中取出结果封装为需要的类型。在Mybatis中可以设置将结果直接映射为自己需要的类型，比如：JavaBean对象、一个Map、一个List等等。

解决方法： Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

MyBatis的工作原理





综合实例见MyBatisTest工程

mybatis-config.xml

mybatis-config.xml详解

```xml
<?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">
<!-- 根标签 -->
<configuration>
<properties>
    <property name="driver" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis-110?useUnicode=true&amp;characterEncoding=utf-8&
    <property name="username" value="root"/>
        <property name="password" value="123456"/>
  </properties>

  <!-- 环境，可以配置多个，default：指定采用哪个环境 -->
  <environments default="test">
     <!-- id：唯一标识 -->
     <environment id="test">
        <!-- 事务管理器，JDBC类型的事务管理器 -->
        <transactionManager type="JDBC" />
        <!-- 数据源，池类型的数据源 -->
        <dataSource type="POOLED">
           <property name="driver" value="com.mysql.jdbc.Driver" />
           <property name="url" value="jdbc:mysql://127.0.0.1:3306/mybatis-110" />
           <property name="username" value="root" />
           <property name="password" value="123456" />
        </dataSource>
     </environment>
     <environment id="development">
        <!-- 事务管理器，JDBC类型的事务管理器 -->
        <transactionManager type="JDBC" />
        <!-- 数据源，池类型的数据源 -->
        <dataSource type="POOLED">
           <property name="driver" value="${driver}" /> <!-- 配置了properties，所以可以直接引用 -->
           <property name="url" value="${url}" />
           <property name="username" value="${username}" />
           <property name="password" value="${password}" />
        </dataSource>
     </environment>
  </environments>
</configuration>
```

还包括mappers

```xml
<mappers>
  <mapper resource="mappers/UserMapper.xml" />
  <mapper resource="dao/UserDaoMapper.xml"/>
</mappers>
```

mybatis-config.xml详解

XML 映射配置文件

MyBatis 的配置文件包含了影响 MyBatis 行为甚深的设置（settings）和属性（properties）信息。文档的顶层结构如下：

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型命名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

（1）properties属性读取外部资源

properties配置的属性都是可外部配置且可动态替换的，既可以在典型的 Java 属性文件中配置，亦可通过 properties 元素的子元素来传递。例如：

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

见Chapter5工程

其中的属性就可以在整个配置文件中被用来替换需要动态配置的属性值。比如:

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

username 和 password 将会由 properties 元素中设置的相应值来替换。driver 和 url 属性将会由 config.properties 文件中对应的值来替换。这样就为配置提供了诸多灵活选择

（2）typeAliases

见Chapter5工程

类型别名是为 Java 类型命名的一个短的名字。它只和 XML 配置有关，存在的意义仅在于用来减少类完全限定名的冗余。

```
<typeAliases>
    <typeAlias type="com.zpc.mybatis.pojo.User" alias="User"/>
</typeAliases>
```

缺点：每个pojo类都要去配置。

解决方案：使用扫描包，扫描指定包下的所有类，扫描之后的别名就是类名（不区分大小写），建议使用的时候和类名一致。

```
<typeAliases>
    <!--type:实体类的全路径。alias:别名，通常首字母大写-->
    <!--<typeAlias type="com.zpc.mybatis.pojo.User" alias="User"/>-->
    <package name="com.zpc.mybatis.pojo"/>
</typeAliases>
```

（3）environments(环境)

配置环境，Mybatis可以配置多个环境，default指向默认的环境id。每个SqlSessionFactory对应一个环境。

实际使用场景下，更多的是选择使用spring来管理数据源，来做到环境的分离。


```
<environments default="development">
    <environment id="development">
        <!-- 使用JDBC的事务管理
            •JDBC – 这个配置直接简单使用了 JDBC 的提交和回滚设置。它依赖于从数据源得 到的连接来管理事务范围。
            •MANAGED – 这个配置几乎没做什么。它从来不提交或回滚一个连接。而它会让 容器来管理事务的整个生命周期(比如 Spring 或 JEE 应用服务器的上下文)
        -->
        <transactionManager type="JDBC"/>

        <!--
            UNPOOLED – 这个数据源的实现是每次被请求时简单打开和关闭连接。
            POOLED – 这是 JDBC 连接对象的数据源连接池的实现,用来避免创建新的连接实例时必要的初始连接和认证时间。这是一种当前 Web 应用程序用来快速响应请求很流行的方 法。
            JNDI – 这个数据源的实现是为了使用如 Spring 或应用服务器这类的容器,容器可以集中或在外部配置数据源,然后放置一个 JNDI 上下文的引用
         -->
        <dataSource type="POOLED">
            <!-- MySQL数据库驱动 -->
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <!-- 连接数据库的URL -->
            <property name="url" value="jdbc:mysql://localhost:3306/java web?characterEncoding=utf8"/>
            <property name="username" value="root"/>
            <property name="password" value="root"/>
        </dataSource>
    </environment>
</environments>
```

（4）mappers

需要告诉 **MyBatis** 到哪里去找到 **SQL** 映射语句。即告诉 **MyBatis** 到哪里去找映射文件。你可以使用相对于类路径的资源引用， 或完全限定资源定位符（包括 **file:///** 的 **URL**），或类名和包名等。

Mapper的三种配置方法

（1） <mapper resource=" " />

使用相对于类路径的资源

如：<mapper resource="sqlmap/User.xml" />

（2） <mapper url=" " />

使用完全限定路径

如：

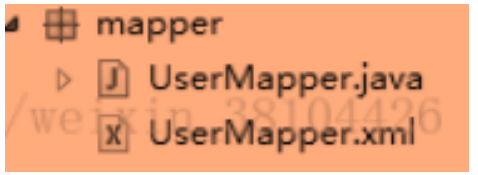
<mapper url="file:///D:\workspace_spingmvc\mybatis_01\config\sqlmap\User.xml"/>

（3）<mapper class=" " />

使用mapper接口类路径

如：<mapper class="cn.itcast.mybatis.mapper. UserMapper"/>

要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中



（4）mappers

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

<!-- 使用映射器接口实现类的完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

所谓的**mapper**接口路径，实际上就是**dao**的接口路径。

1、定义一个接口。

2、在接口所在的包中定义**mapper.xml**，并且要求**xml**文件和**interface**的名称要相同。

3、在**mybatis-config.xml** 中通过**class**路径，引入**mapper**（注解方式）。要求**mapper.xml** 中的名称空间是类的接口的全路径。

映射器
XXXMapper.xml

映射器（XXXMapper.xml）

见ch7 工程
UserMapper.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.mybatis.mapper.UserMapper">
    <!-- 根据uid查询一个用户信息 -->
    <select id="selectUserById" parameterType="Integer"
        resultType="com.mybatis.po.MyUser">
        select * from user where uid = #{uid}
    </select>
    <!-- 查询所有用户信息 -->
    <select id="selectAllUser" resultType="com.mybatis.po.MyUser">
        select * from user
    </select>
    <!-- 添加一个用户，#{uname}为com.mybatis.po.MyUser的属性值 -->
    <insert id="addUser" parameterType="com.mybatis.po.MyUser">
        insert into user (uname,usex) values(#{uname},#{usex})
    </insert>
    <!-- 修改一个用户 -->
    <update id="updateUser" parameterType="com.mybatis.po.MyUser">
        update user set uname =
        #{uname},usex = #{usex} where uid = #{uid}
    </update>
    <!-- 删除一个用户 -->
    <delete id="deleteUser" parameterType="Integer">
        delete from user where uid = #{uid}
    </delete>
</mapper>
```

映射器（XXXMapper.xml）

元素名称	描　述	备　注
select	查询语句，最常用、最复杂的元素之一	可以自定义参数，返回结果集等
insert	插入语句	执行后返回一个整数，代表插入的行数
update	更新语句	执行后返回一个整数，代表更新的行数
delete	删除语句	执行后返回一个整数，代表删除的行数
sql	定义一部分SQL，在多个位置被引用	例如，一张表列名，一次定义，可以在多个SQL语句中使用
resultMap	用来描述从数据库结果集中来加载对象，是最复杂、最强大的元素	提供映射规则

映射器——<select>元素

在SQL映射文件中<select>元素用于映射SQL的select语句

```
        <!-- 根据uid查询一个用户信息 -->
<select id="selectUserById " parameterType="Integer"
                resultType="com.po.MyUser">
        select * from user where uid = #{uid}
</select>
```

- id的值是唯一标识符；
- 接收一个Integer类型的参数并赋值给uid；
- 返回一个MyUser类型的对象，结果集自动映射到MyUser属性。

映射器——<select>元素

select元素的属性

属性名称	描　述
id	它和Mapper的命名空间组合起来使用，是唯一标识符，供MyBatis调用
parameterType	表示传入SQL语句的参数类型的全限定名或别名。是个可选属性，MyBatis能推断出具体传入语句的参数。
resultType	SQL语句执行后返回的类型（全限定名或者别名）。如果是集合类型，返回的是集合元素的类型。返回时可以使用resultType或resultMap之一
resultMap	它是映射集的引用，与< resultMap>元素一起使用。返回时可以使用resultType或resultMap之一
flushCache	它的作用是在调用SQL语句后，是否要求MyBatis清空之前查询本地缓存和二级缓存。默认值为false。如果设置为true，则任何时候只要SQL语句被调用，都将清空本地缓存和二级缓存
useCache	启动二级缓存的开关。默认值为true，表示将查询结果存入二级缓存中
timeout	用于设置超时参数，单位是秒。超时将抛出异常。
fetchSize	获取记录的总条数设定
statementType	告诉MyBatis使用哪个JDBC的Statement工作，取值为STATEMENT（Statement）、PREPARED（PreparedStatement）、CALLABLE（CallableStatement）
resultSetType	这是针对JDBC的ResultSet接口而言，其值可设置为FORWARD_ONLY（只允许向前访问）、SCROLL_SENSITIVE（双向滚动，但不及时更新）、SCROLL_INSENSITIVE（双向滚动，及时更新）

映射器——<insert>元素

- 用于映射插入语句，与<select>元素的属性大部分相同；
- 特有属性：
  ✓ keyProperty：将插入或更新操作时的返回值赋值给PO类的某个属性，通常会设置为主键对应的属性。如果是联合主键，可以在多个值之间用逗号隔开；
  ✓ keyColumn：该属性用于设置第几列是主键，当主键列不是表中的第一列时需要设置。如果是联合主键时，可以在多个值之间用逗号隔开；
  ✓ useGeneratedKeys：该属性将使MyBatis使用JDBC的getGeneratedKeys()方法获取由数据库内部生产的主键，如MySQL、SQL Server等自动递增的字段，其默认值为false。

主键（自动递增）回填

MySQL、SQL Server等数据库的表格可以采用自动递增的字段作为主键。有时可能需要使用这个刚刚产生的主键，用以关联其他业务。

```
<!-- 添加一个用户，成功后将主键值回填给uid（po类的属性）-->
<insert id=“addUser” ParameterType="com.po.MyUser"
        keyProperty="uid" useGeneratedKeys="true">
   insert into user (uname,usex) values(#{uname},#{usex})
</insert>
```

自定义主键

如果实际工程中使用的数据库不支持主键自动递增（如Oracle），或者取消了主键自动递增的规则时，可以使用MyBatis的<selectKey>元素来自定义生成主键。

```
<insert id="insertUser" parameterType="com.po.MyUser">
    <!-- 先使用selectKey元素定义主键，然后再定义SQL语句 -->
    <selectKey keyProperty="uid" resultType="Integer" order="BEFORE">
        select if(max(uid) is null, 1 , max(uid)+1) as newUid from user
    </selectKey>
    insert into user (uid,uname,usex) values(#{uid},#{uname},#{usex})
</insert>
```

映射器——<update>与<delete>元素

<update>和<delete>元素比较简单，它们的属性和<insert>元素、<select>元素的属性差不多，执行后也返回一个整数，表示影响了数据库的记录行数。

```
<!-- 修改一个用户 -->
<update id="updateUser" parameterType="com.po.MyUser">
    update user set uname = #{uname},usex = #{usex} where
    uid = #{uid}
</update>

<!-- 删除一个用户 -->
<delete id="deleteUser" parameterType="Integer">
    delete from user where uid = #{uid}
</delete>
```

映射器——<sql>元素

- <sql>元素的作用在于可以定义SQL语句的一部分（代码片段），方便后面的SQL语句引用它，比如反复使用的列名。

```
<sql id="comColumns">uid,uname,usex</sql>
<select id="selectUser" resultType="com.po.MyUser">
    select <include refid="comColumns"/> from user
</select>
```

- Sql片段也可以定义在单独的.xml文件中如：

CommonSQL.xml (1)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="CommonSQL">
    <sql id="commonSql">
        id,
            user_name,
            password,
            name,
            age,
            sex,
            birthday,
            created,
            updated
    </sql>
</mapper>
```

引用

```
<select id="queryUserById" resultMap="userResultMap">
    select <include refid="CommonSQL.commonSql"></include> from tb_user where id = #{id}
</select>
```

(2)

最后在全局配置文件mybatis-config.xml中引入该外部配置文件：

```
<mappers>
        <mapper resource="CommonSQL.xml"/>
        <!-- 开启mapper接口的包扫描，基于class的配置方式 -->
        <package name="com.zpc.mybatis.mapper"/>
</mappers>
```

(3)

映射器——#{}与${}

#{},预编译的方式preparedstatement，使用占位符替换（相当于？），传入的数据都当成一个字符串，会对自动传入的数据加一个双引号，防止sql注入，一个参数的时候，任意参数名都可以接收

**${}**,普通的Statement，字符串直接拼接，参数被当成sql语句中的一部分，不可以防止sql注入，一个参数的时候，必须使用${value}接收参数

如果param传入的值为zhangsan

select * from table where name = #{param}

select * from table where name = “zhangsan”

#{} 的参数替换是发生在 DBMS 中

翻译成

select * from table where name = ${param}

select * from table where name = zhangsan

${}的参数替换发生在动态解析过程中

结果一样，但原理不一样

什么是sql注入?

```
strSQL = "SELECT * FROM users WHERE (name = '" + userName + "') and (pw = '"+ passWord +"');"
```

如果令userName = "1' OR '1'='1"; passWord = "1' OR '1'='1";

strSQL = "SELECT * FROM users WHERE (name = '1' OR '1'='1') and (pw = '1' OR '1'='1');"

为什么#{}可以防止sql注入？

如果param传入的值为userName="1' OR '1'='1"

- **#{}**:会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值
- **${}**:会将sql中的${}替换为字符串 “”1 ‘ OR ’ 1 ‘=’ 1 “” ,并编译成sql语句

预编译是提前对SQL语句进行预编译，其后注入的参数将不会再进行SQL编译

SELECT * FROM users WHERE (name = '" + userName + "') ;

翻译成

SELECT * FROM users WHERE (name = '**1' OR '1'='1**')

SELECT * FROM users WHERE (name = '**1' OR '1'='1**')

映射器——#{}与${}

```
select * from ${tableName} where name = #{name}
```

如果令表名为： “user; delete user; --” 会有什么后果?

特殊情况

```
<select id="queryUserByTableName"   resultType="com.zpc.mybatis.pojo.User">
    select * from #{tableName}
</select>
```

select * from “tb_user”;

```
<select id="queryUserByTableName"   resultType="com.zpc.mybatis.pojo.User">
    select * from ${tableName}
</select>
```

select * from tb_user;

表名只能用${}

映射器——resultMap

resultMap是MyBatis中最重要最强大的元素，使用resultMap可以解决两大问题：

- **POJO属性名和表结构字段名不一致的问题（有些情况下也不是标准的驼峰格式）；**

```
<insert id="addUser" parameterType="com.mybatis.po.MyUser">
    insert into user (uname,usex) values(#{uname},#{usex})
</insert>
```

属性名与字段一致

不一致？

- **完成高级查询，比如一对一，一对多，多对多**

解决表字段名和属性名不一致的问题有两种方法：

1、如果是驼峰式的命名规则，可以在MyBatis配置文件中设置<setting name="mapUnderscoreToCamelCase" value="true"/>解决

2、使用resultMap解决

高级查询在后面介绍

映射器——resultMap

resultMap语法

```
<resultMap type="" id="">
    <constructor><!-- 类在实例化时，用来注入结果到构造方法 -->
        <idArg/><!-- ID参数，结果为ID -->
        <arg/><!-- 注入到构造方法的一个普通结果 -->
    </constructor>
    <id/><!-- 用于表示哪个列是主键 -->
    <result/><!-- 注入到字段或JavaBean属性的普通结果 -->
    <association property=""/><!-- 用于一对一关联 -->
    <collection property=""/><!-- 用于一对多、多对多关联 -->
    <discriminator javaType=""><!-- 使用结果值来决定使用哪个结果映射 -->
        <case value="" /> <!-- 基于某些值的结果映射，也称鉴别器 -->
    </discriminator>
</resultMap>
```

用法见级联查询

映射器——resultMap

简单示例

在同一个XXXMapper.xml文件中

```
<!--
  type：返回的结果集对应的java的实体类型
  id：resultMap的唯一标识
  autoMapping：默认完成映射，如果已开启驼峰匹配，可以解决驼峰匹配
 -->
<resultMap type="User" id="resultUser" autoMapping="true">
  <!--
      指定主键
      column：数据库中的列名
      property：java实体类中的属性名
   -->
  <id column="id" property="id"/>
  <!-- 使用result配置数据库列名和java实体类中的属性名对应 -->
  <result column="user_name" property="userName"/>
</resultMap>
```



```
<select id="queryUsersByTableName" resultMap="resultUser">
  select * from ${tableName}
</select>
```

映射器——resultMap

见工程ch7

com.mybatis.UserMapper.xml

```
<!-- 使用自定义结果集类型 -->
<resultMap type="com.pojo.MapUser" id="myResult">
    <!-- property是com.pojo.MapUser类中的属性-->
    <!-- column是查询结果的列名，可以来自不同的表 -->
    <id property="m_uid" column="uid"/>
    <result property="m_uname" column="uname"/>
    <result property="m_usex" column="usex"/>
</resultMap>


<!-- 使用自定义结果集类型查询所有用户 -->
<select id="selectResultMap" resultMap="myResult">
    select * from user
</select>
```

映射器——resultMap

结果集存储

1、使用Map存储（略）

2、使用POJO存储

见前页PPT

```
public class MapUser {
    private Integer m_uid;
    private String m_uname;
    private String m_usex;
    public Integer getM_uid() {
        return m_uid;
    }
    //此处略去setters和getters
    @Override
    public String toString() {
        return "User [uid=" + m_uid +",uname=" + m_uname + ",usex=" + m_usex +"]";
    }
}
```

感谢聆听

Thanks For Your Listening!