Mybatis

Mybatis入门

Mybatis简介

JDBC 存在的问题

1. 数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。
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。

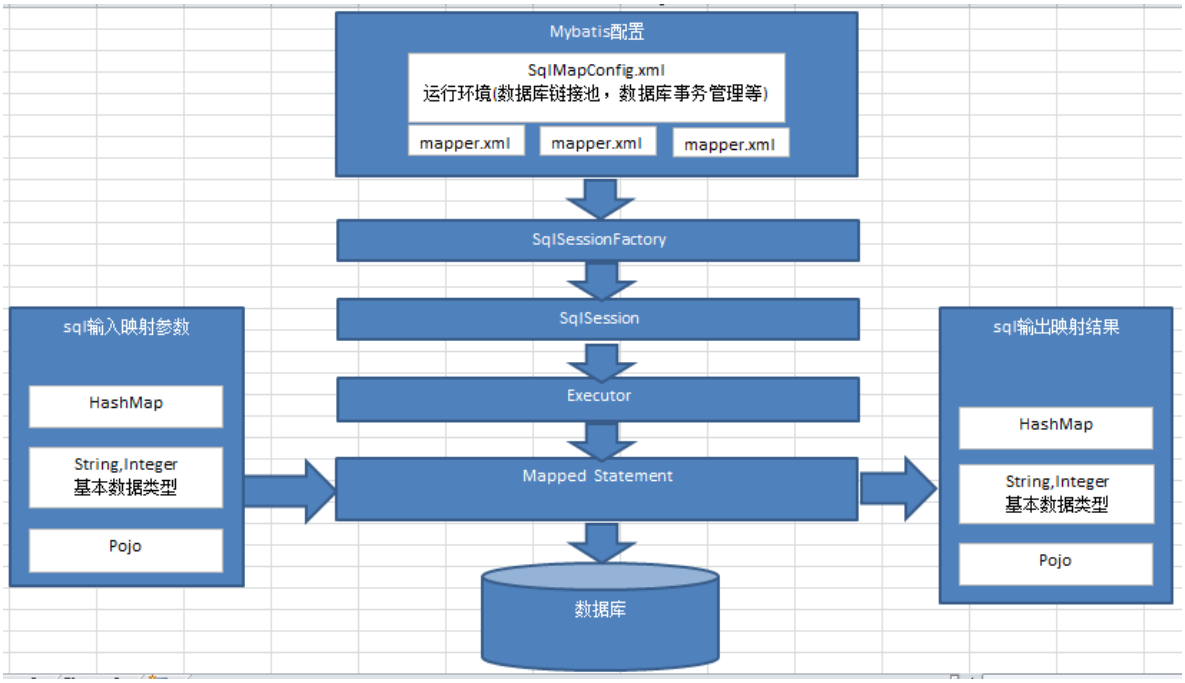
上面的问题，借助于第三方工具如 DBUtils 或者 Spring 中自带的数据库操作框架 JdbcTemplate，都可以在一定程度上解决该问题。但是不完美，真正能解决这些问题的框架就是两大类，一种就是 MyBatis，另一种则是 Jpa。

MyBatis 介绍

MyBatis 是一个优秀的持久层框架，它对 jdbc 的操作数据库的过程进行封装，使开发者只需要关注 SQL 本身，而不需要花费精力去处理例如注册驱动、创建 connection、创建 statement、手动设置参数、结果集检索等 jdbc 繁杂的过程代码。Mybatis 通过 xml 或注解的方式将要执行的各种 statement（statement、preparedStatemnt、CallableStatement）配置起来，并通过 java 对象和 statement 中的 sql 进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射成 java 对象并返回。

与其他的对象关系映射框架不同，MyBatis 并没有将 Java 对象与数据库表关联起来，而是将 Java 方法与 SQL 语句关联。MyBatis 允许用户充分利用数据库的各种功能，例如存储过程、视图、各种复杂的查询以及某数据库的专有特性。如果要对遗留数据库、不规范的数据库进行操作，或者要完全控制 SQL 的执行，MyBatis 是一个不错的选择。

Mybatis架构



1. mybatis 配置:mybatis-config.xml，此文件作为 mybatis 的全局配置文件，配置了 mybatis 的运行环境等信息。另一个 mapper.xml 文件即 sql 映射文件，文件中配置了操作数据库的 sql 语句。此文件需要在 mybatis-config.xml 中加载。
2. 通过 mybatis 环境等配置信息构造 SqlSessionFactory 即会话工厂
3. 由会话工厂创建 sqlSession 即会话，操作数据库需要通过 sqlSession 进行。
4. mybatis 底层自定义了 Executor 执行器接口操作数据库，Executor 接口有两个实现，一个是基本执行器、一个是缓存执行器。
5. Mapped Statement 也是 mybatis 一个底层封装对象，它包装了 mybatis 配置信息及 sql 映射信息等。mapper.xml 文件中一个 sql 对应一个 Mapped Statement 对象，sql 的 id 即是Mapped statement 的 id。
6. Mapped Statement 对 sql 执行输入参数进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 前将输入的 java 对象映射至 sql 中，输入参数映射就是 jdbc 编程中对 preparedStatement 设置参数。
7. Mapped Statement 对 sql 执行输出结果进行定义，包括 HashMap、基本类型、pojo，Executor 通过 Mapped Statement 在执行 sql 后将输出结果映射至 java 对象中，输出结果映射过程相当于 jdbc 编程中对结果的解析处理过程。

MyBatis 所解决的 JDBC 中存在的问题

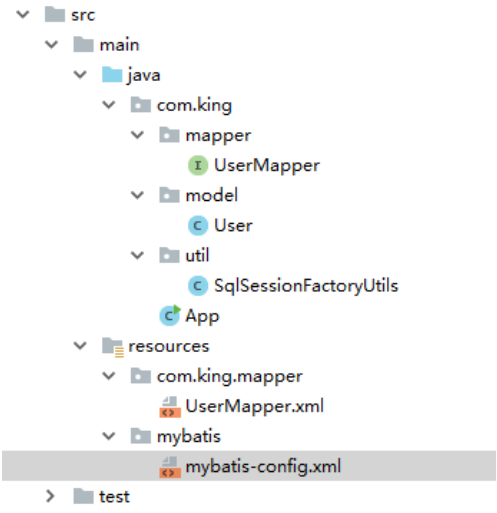
1. 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。解决：在 mybatis-config.xml 中配置数据链接池，使用连接池管理数据库链接。
2. Sql语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。
3. 向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。解决：Mybatis 自动将 java 对象映射至 sql 语句，通过 statement 中的 parameterType 定义输入参数的类型。
4. 对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。解决：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。

增删改查

创建数据库

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/`test01` /*!40100 DEFAULT CHARACTER SET utf8mb4 COLLATE
utf8mb4_general_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;

USE `test01`;

/*Table structure for table `user` */

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `address` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

/*Data for the table `user` */

insert  into `user`(`id`,`username`,`address`) values (1,'javaboy123','www.javaboy.org'),
(3,'javaboy','spring.javaboy.org'),(4,'张三','深圳'),(5,'李四','广州'),(6,'王五','北京');
```

目录结构



代码详情

pom文件

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

    <groupId>com.king</groupId>
    <artifactId>mybatis-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.2</version>
        </dependency>
```

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.17</version>
        </dependency>
    </dependencies>

</project>
```

mybatis-config.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">
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://127.0.0.1/user_db?serverTimezone=Asia/Shanghai"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <!--<mapper resource="mybatis/mapper/UserMapper.xml"/>-->
        <package name="com.king.mapper"/>

    </mappers>
</configuration>
```

注意：UserMapper.xml 和 UserMapper 需要放在同一个包下面。若放在resource目录下，需要手动在 resources 目录下，创建一个和 UserMapper 接口相同的目录

UserMapper.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.king.mapper.UserMapper">

    <select id="getUserById" resultType="com.king.model.User">
        select * from user where id=#{id};
    </select>
    <insert id="addUser" parameterType="com.king.model.User">
        insert into user (username,address) values (#{username},#{address});
    </insert>
    <!--调用数据库uuid()函数-->
    <insert id="addUser2" parameterType="com.king.model.User">
        <selectKey resultType="java.lang.String" keyProperty="id" order="BEFORE">
            select uuid();
        </selectKey>
        insert into user (id,username,address) values (#{id},#{username},#{address});
    </insert>

    <delete id="deleteUserById" parameterType="java.lang.Integer">
        delete from user where id=#{id}
    </delete>

    <update id="updateUser" parameterType="com.king.model.User">
```

```
        update user set username = #{username} where id=#{id};
    </update>

    <select id="getAllUser" resultType="com.king.model.User">
        select * from user;
    </select>
</mapper>
```

- selectKey 表示查询 key
- keyProperty 属性表示将查询的结果赋值给传递进来的 User 对象的 id 属性
- resultType 表示查询结果的返回类型
- order 表示这个查询操作的执行时机，BEFORE 表示这个查询操作在 insert 之前执行
- 在 selectKey 节点的外面定义 insert 操作

User.java

```
package com.king.model;

public class User {
    private Integer id;
    private String username;
    private String address;

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", address='" + address + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

UserMapper.java

```java
package com.king.mapper;

import com.king.model.User;

import java.util.List;

public interface UserMapper {
    User getUserById(Integer id);

    Integer addUser(User user);

    Integer addUser2(User user);

    Integer deleteUserById(Integer id);

    Integer updateUser(User user);

    List<User> getAllUser();
}
```

SqlSessionFactoryUtils.java

```java
package com.king.util;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;

public class SqlSessionFactoryUtils {
    private static SqlSessionFactory SQL_SESSION_FACTORY = null;

    public static SqlSessionFactory getInstance() {
        if (SQL_SESSION_FACTORY == null) {
            try {
                SQL_SESSION_FACTORY = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("mybatis/mybatis-config.xml"));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return SQL_SESSION_FACTORY;
    }
}
```

App.java

```java
package com.king;

import com.king.mapper.UserMapper;
import com.king.model.User;
import com.king.util.SqlSessionFactoryUtils;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;

import java.util.List;

public class App {
    public static void main(String[] args) {
```

```
        SqlSessionFactory instance = SqlSessionFactoryUtils.getInstance();
        SqlSession sqlSession = instance.openSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> allUser = mapper.getAllUser();
        System.out.println(allUser);
    }
}
```

Mybatis全局配置

properties

properties 可以用来引入一个外部配置，最近常见的例子就是引入数据库的配置文件，例如我们在 resources 目录下添加一个 db.properties 文件作为数据库的配置文件，文件内容如下：

```
db.username=root
db.password=123456
db.driver=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql:///test01?serverTimezone=Asia/Shanghai
```

然后，利用 mybatis-config.xml 配置文件中的 properties 属性，引入这个配置文件，然后在 DataSource 中使用这个配置文件，最终配置如下：

```
<configuration>
    <properties resource="db.properties"></properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${db.driver}"/>
                <property name="url" value="${db.url}"/>
                <property name="username" value="${db.username}"/>
                <property name="password" value="${db.password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="org.javaboy.mybatis.mapper"/>
    </mappers>
</configuration>
```

settings

Setting(设置)	Description（描述）	Valid Values(验证值组)	Default(默认值)
cacheEnabled	在全局范围内启用或禁用缓存配置任何映射器在此配置下	true or false	TRUE
lazyLoadingEnabled	在全局范围内启用或禁用延迟加载。禁用时，所有查询将热加载	true or false	TRUE
aggressiveLazyLoading	启用时，有延迟加载属性的对象将被完全加载后调用懒惰的任何属性。否则，每一个属性是按需加载。	true or false	TRUE
multipleResultSetsEnabled	允许或不允许从一个单独的语句（需要兼容的驱动程序）要返回多个结果集。	true or false	TRUE
useColumnLabel	使用列标签，而不是列名。在这方面，不同的驱动有不同的行为。参考驱动文档或测试两种方法来决定你的驱动程序的行为如何。	true or false	TRUE
useGeneratedKeys	允许 JDBC 支持生成的密钥。兼容的驱动程序是必需的。此设置强制生成的键被使用，如果设置为 true，一些驱动会不兼容性，但仍然可以工作。	true or false	FALSE
autoMappingBehavior	指定 MyBatis 应如何自动映射列到字段/属性。NONE自动映射。 PARTIAL 只会自动映射结果没有嵌套结果映射定义里面。 FULL 会自动映射的结果映射任何复杂的（包含嵌套或其他）。	NONE, PARTIAL, FULL	PARTIAL
defaultExecutorType	配置默认执行人。SIMPLE执行人确实没有什么特别的。 REUSE执行器重用准备好的语句。 BATCH执行器重用语句和批处理更新。	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	设置驱动程序等待一个数据库响应的秒数。	Any positive integer	Not Set (null)
safeRowBoundsEnabled	允许使用嵌套的语句RowBounds。	true or false	FALSE
mapUnderscoreToCamelCase	从经典的数据库列名 A_COLUMN 启用自动映射到骆驼标识的经典的 Java 属性名 aColumn。	true or false	FALSE
localCacheScope	MyBatis的使用本地缓存，以防止循环引用，并加快反复嵌套查询。默认情况下（SESSION）会话期间执行的所有查询缓存。如果 localCacheScope=STATMENT 本地会话将被用于语句的执行，只是没有将数据共享之间的两个不同的调用相同的 SqlSession。	SESSION or STATEMENT	SESSION
dbcTypeForNull	指定为空值时，没有特定的JDBC类型的参数的 JDBC 类型。有些驱动需要指定列的 JDBC 类型，但其他像 NULL，VARCHAR 或 OTHER 的工作与通用值。	JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER	OTHER
lazyLoadTriggerMethods	指定触发延迟加载的对象的方法。	A method name list separated by commas	equals,clone,hashCode,toString
defaultScriptingLanguage	指定所使用的语言默认为动态SQL生成。	A type alias or fully qualified class name.	org.apache.ibatis.scripting.xmltags.XMLDynamicLanguageDriver
callSettersOnNulls	指定如果setter方法或地图的put方法时，将调用检索到的值是null。它是有用的，当你依靠Map.keySet（）或null初始化。注意原语（如整型，布尔等）不会被设置为null。	true or false	FALSE
logPrefix	指定的前缀字串，MyBatis将会增加记录器的名称。	Any String	Not set
logImpl	指定MyBatis的日志实现使用。如果此设置是不存在的记录的实施将自动查找。	SLF4J or LOG4J or LOG4J2 or JDK_LOGGING or COMMONS_LOGGING or STDOUT_LOGGING or NO_LOGGING	Not set
proxyFactory	指定代理工具，MyBatis将会使用创建懒加载能力的对象。	CGLIB	JAVASSIST

typeAliases

这个是 MyBatis 中定义的别名，分两种，一种是 MyBatis 自带的别名，另一种是我们自定义的别名。

MyBatis 自带的别名

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

本来，我们在 Mapper 中定义数据类型时，需要写全路径，如下：

```
<select id="getUserCount" resultType="java.lang.Integer">
    select count(*) from user ;
</select>
```

但是，每次写全路径比较麻烦。这种时候，我们可以用类型的别名来代替，例如用 int 做 Integer 的别名：

```
<select id="getUserCount" resultType="int">
    select count(*) from user ;
</select>
```

自定义别名

我们自己的对象，在 Mapper 中定义的时候，也是需要写全路径：

```
<select id="getAllUser" resultType="org.javaboy.mybatis.model.User">
    select * from user;
</select>
```

这种情况下，写全路径也比较麻烦，我们可以给我们自己的 User 对象取一个别名，在 mybatis-config.xml 中添加 typeAliases 节点：

```
<configuration>
    <properties resource="db.properties"></properties>

    <typeAliases>
        <typeAlias type="org.javaboy.mybatis.model.User" alias="javaboy"/>
    </typeAliases>

    <environments default="development">
        <environment id="development">
```

```
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${db.driver}"/>
                <property name="url" value="${db.url}"/>
                <property name="username" value="${db.username}"/>
                <property name="password" value="${db.password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="org.javaboy.mybatis.mapper"/>
    </mappers>
</configuration>
```

这里，我们给 User 对象取了一个别名叫 javaboy，然后，我们就可以在 Mapper 中直接使用 javaboy 来代替 User 对象了：

```
<select id="getAllUser" resultType="javaboy">
    select * from user;
</select>
```

但是，这种一个一个去枚举对象的过程非常麻烦，我们还可以批量给对象定义别名，批量定义主要是利用包扫描来做，批量定义默认的类的别名，是类名首字母小写，例如如下配置：

```
<typeAliases>
    <package name="org.javaboy.mybatis.model"/>
</typeAliases>
```

这个配置就表示给 `org.javaboy.mybatis.model` 包下的所有类取别名，默认的别名就是类名首字母小写。这个时候，我们在 Mapper 中，就可以利用 user 代替 User 全路径了：

```
<select id="getAllUser" resultType="user">
    select * from user;
</select>
```

在最新版中，批量定义的别名，类名首字母也可以不用小写，在实际开发中，我们一般使用第二种方式（批量定义的方式）

typeHandlers

在 MyBatis 映射中，能够自动将 Jdbc 类型映射为 Java 类型。默认的映射规则，如下：

类型处理器	Java类型	JDBC类型
BooleanTypeHandler	Boolean，boolean	任何兼容的布尔值
ByteTypeHandler	Byte，byte	任何兼容的数字或字节类型
ShortTypeHandler	Short，short	任何兼容的数字或短整型
IntegerTypeHandler	Integer，int	任何兼容的数字和整型
LongTypeHandler	Long，long	任何兼容的数字或长整型
FloatTypeHandler	Float，float	任何兼容的数字或单精度浮点型
DoubleTypeHandler	Double，double	任何兼容的数字或双精度浮点型
BigDecimalTypeHandler	BigDecimal	任何兼容的数字或十进制小数类型
StringTypeHandler	String	CHAR和VARCHAR类型
ClobTypeHandler	String	CLOB和LONGVARCHAR类型
NStringTypeHandler	String	NVARCHAR和NCHAR类型
NClobTypeHandler	String	NCLOB类型
ByteArrayTypeHandler	byte[]	任何兼容的字节流类型
BlobTypeHandler	byte[]	BLOB和LONGVARBINARY类型
DateTypeHandler	Date（java.util）	TIMESTAMP类型
DateOnlyTypeHandler	Date（java.util）	DATE类型
TimeOnlyTypeHandler	Date（java.util）	TIME类型
SqlTimestampTypeHandler	Timestamp（java.sql）	TIMESTAMP类型
SqlDateTypeHandler	Date（java.sql）	DATE类型
SqlTimeTypeHandler	Time（java.sql）	TIME类型
ObjectTypeHandler	任意	其他或未指定类型
EnumTypeHandler	Enumeration类型	VARCHAR-任何兼容的字符串类型，作为代码存储（而不是索引）。

前面案例中，之所以数据能够接收成功，是因为有上面这些默认的类型处理器，处理基本数据类型，这些够用了，特殊类型，需要我们自定义类型处理器。

比如，我有一个用户爱好的字段，这个字段，在对象中，是一个 List 集合，在数据库中，是一个 VARCHAR 字段，这种情况下，就需要我们自定义类型转换器，自定义的类型转换器提供两个功能：

1. 数据存储时，自动将 List 集合，转为字符串（格式自定义）
2. 数据查询时，将查到的字符串再转为 List 集合

首先，在数据表中添加一个 favorites 字段：

然后，在 User 对象中，添加相应的属性：

```
public class User {
    private Integer id;
    private String username;
    private String address;
    private List<String> favorites;

    public List<String> getFavorites() {
        return favorites;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
```

```
                ", username='" + username + '\'' +
                ", address='" + address + '\'' +
                ", favorites=" + favorites +
                '}';
    }

    public void setFavorites(List<String> favorites) {
        this.favorites = favorites;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

为了能够将 List 集合中的数据存入到 VARCHAR 中，我们需要自定义一个类型转换器：

```
@MappedJdbcTypes(JdbcType.VARCHAR)
@MappedTypes(List.class)
public class List2VarcharHandler implements TypeHandler<List<String>> {
    public void setParameter(PreparedStatement ps, int i, List<String> parameter, JdbcType jdbcType) throws
SQLException {
        StringBuffer sb = new StringBuffer();
        for (String s : parameter) {
            sb.append(s).append(",");
        }
        ps.setString(i, sb.toString());
    }

    public List<String> getResult(ResultSet rs, String columnName) throws SQLException {
        String favs = rs.getString(columnName);
        if (favs != null) {
            return Arrays.asList(favs.split(","));
        }
        return null;
    }

    public List<String> getResult(ResultSet rs, int columnIndex) throws SQLException {
        String favs = rs.getString(columnIndex);
        if (favs != null) {
            return Arrays.asList(favs.split(","));
        }
        return null;
    }
```

```
    public List<String> getResult(CallableStatement cs, int columnIndex) throws SQLException {
        String favs = cs.getString(columnIndex);
        if (favs != null) {
            return Arrays.asList(favs.split(","));
        }
        return null;
    }

}
```

- 首先在这个自定义的类型转换器上添加 @MappedJdbcTypes 注解指定要处理的 Jdbc 数据类型，另外还有一个注解是 @MappedTypes 指定要处理的 Java 类型，这两个注解结合起来，就可以锁定要处理的字段是 favorites 了。
- setParameter 方法看名字就知道是设置参数的，这个设置过程由我们手动实现，我们在这里，将 List 集合中的每一项，用一个，串起来，组成一个字符串。
- getResult 方法，有三个重载方法，其实都是处理查询的。

接下来，修改插入的 Mapper：

```
<insert id="addUser" parameterType="org.javaboy.mybatis.model.User">
    insert into user (username,address,favorites) values (#{username},#{address},#
{favorites,typeHandler=org.javaboy.mybatis.typehandler.List2VarcharHandler});
</insert>
```

然后，在 Java 代码中，调用该方法：

```
public class Main2 {
    public static void main(String[] args) {
        SqlSessionFactory instance = SqlSessionFactoryUtils.getInstance();
        SqlSession sqlSession = instance.openSession();
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = new User();
        user.setUsername("风气");
        user.setAddress("上海");
        List<String> favorites = new ArrayList<String>();
        favorites.add("足球");
        favorites.add("篮球");
        favorites.add("乒乓球");
        user.setFavorites(favorites);
        mapper.addUser(user);
        sqlSession.commit();
    }
}
```

读取的配置，有两个地方，一个可以在 ResultMap 中做局部配置，也可以在全局配置中进行过配置，全局配置方式如下：

```
<configuration>
    <properties resource="db.properties"></properties>
    <typeAliases>
        <package name="com.*...."/>
    </typeAliases>
    <typeHandlers>
        <package name="com.mybatis.typehandler"/>
    </typeHandlers>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="${db.driver}"/>
                <property name="url" value="${db.url}"/>
                <property name="username" value="${db.username}"/>
                <property name="password" value="${db.password}"/>
```

```
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <package name="com.*...."/>
    </mappers>
</configuration>
```

接下来去查询，查询过程中，就会自动将字符串转为 List 集合了

Mapper

Mapper 配置的几种方法：

1. `<mapper resource=" " />`

   使用相对于类路径的资源，即 XML 的定位，从 classpath 开始写。

   ```
   <mappers>
       <mapper resource="Mapper xml的路径（相对于classes的路径）"/>
   </mappers>
   ```

   「注意」 直接引用的xml，Mapper xml中namespace同名的Mapper接口(接口存在)

2. `<mapper url="" />`

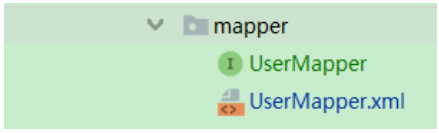
   使用完全限定路径，相当于使用绝对路径，这种方式使用非常少。

   ```
   <mapper url="file:///D:\demo\xxx\User.xml" />
   ```

3. `<mapper class=" " />`

   ```
   <mappers>
       <mapper class="接口的完整类名" />
   </mappers>
   ```

   mybais去加载class对应的接口，然后还会去加载该接口 同目录下的同名xml文件

   编译之后的文件中少了UserMapper.xml，还需要在pom.xml中project下添加如下配置：

   ```
   <build>
       <resources>
           <resource>
               <directory>${project.basedir}/src/main/java</directory>
               <includes>
                   <include>**/*.xml</include>
               </includes>
           </resource>
           <resource>
               <directory>${project.basedir}/src/main/resources</directory>
               <includes>
                   <include>**/*</include>
               </includes>
           </resource>
       </resources>
   </build>
   ```

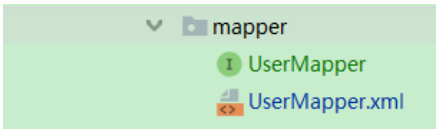
4. `<package name=""/>`

   批量注册Mapper接口(类似别名包扫描)

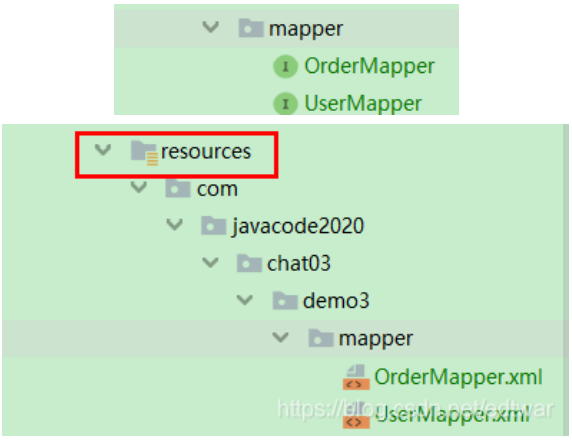
```
<mappers>
    <package name="需要扫描的包" />
</mappers>
```

注意：此种方法要求 mapper 接口名称和 mapper 映射文件名称相同，且放在同一个目录中。实际项目中，多采用这种方式。

- 方式一



- 方式二



Mapper 映射文件

parameterType

这个表示输入的参数类型。

$ 和

在 MyBatis 中，我们在 mapper 引用变量时，默认使用的是 `#`，像下面这样：

```
<select id="getUserById" resultType="org.javaboy.mybatis.model.User">
    select * from user where id=#{id};
</select>
```

除了使用 `#` 之外，我们也可以使用 `$` 来引用一个变量：

```
<select id="getUserById" resultType="org.javaboy.mybatis.model.User">
    select * from user where id=${id};
</select>
```

既然 `#` 和 `$` 符号都可以使用，那么他们有什么区别呢？

我们在 resources 目录下，添加 log4j.properties ，将 MyBatis 执行时的 SQL 打印出来：

```
log4j.rootLogger=DEBUG,stdout
log4j.logger.org.mybatis=DEBUG
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p %d %C: %m%n
```

然后添加日志依赖：

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>
```

然后，我们可以分别观察 `$` 和 `#` 执行时的日志：

```
DEBUG 2019-12-16 15:24:37,810 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==>  Preparing: select * from user where id=1;
DEBUG 2019-12-16 15:24:38,213 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: ==> Parameters:
DEBUG 2019-12-16 15:24:38,700 org.apache.ibatis.logging.jdbc.BaseJdbcLogger: <==      Total: 1
User{id=1, username='javaboy', address='www.javaboy.org', favorites=null}
```

上面这个日志，是 `$` 符号执行的日志，可以看到，SQL 直接就拼接好了，没有参数。

下面这个，是 `#` 执行的日志，可以看到，这个日志中，使用了预编译的方式：

在 MyBatis 中，`$` 相当于是参数拼接的方式，而 `#` 则相当于是占位符的方式。

有的 SQL 拼接实际上可以通过数据库函数来解决，例如模糊查询：

```
<select id="getUserByName" resultType="org.javaboy.mybatis.model.User">
    select * from user where username like concat('%',#{name},'%');
</select>
```

但是有的 SQL 无法使用 `#` 来拼接，例如传入一个动态字段进来，假设我想查询所有数据，要排序查询，但是排序的字段不确定，需要通过参数传入，这种场景就只能使用 `$`，例如如下方法：

```
List<User> getAllUser(String orderBy);
```

定义该方法对应的 XML 文件：

```
<select id="getAllUser" resultType="user">
    select * from user order by ${orderBy}
</select>
```

小结

面试中，遇到这个问题，一定要答出来 Statement 和 PreparedStatement 之间的区别，这个问题才算理解到位了。

简单类型

简单数据类型传递比较容易，像前面的根据 id 查询一条记录就算是这一类的。

这里再举一个例子，比如根据 id 修改用户名：

```
Integer updateUsernameById(String username, Integer id);
```

再定义该方法对应的 mapper：

```
<update id="updateUsernameById">
    update user set username = #{username} where id=#{id};
</update>
```

此时，如果直接调用该方法，会抛出异常：

```
### The error may exist in org/javaboy/mybatis/mapper/UserMapper.xml
### The error may involve defaultParameterMap
### The error occurred while setting parameters
### SQL: update user set username = ? where id=?;
### Cause: org.apache.ibatis.binding.BindingException: Parameter 'username' not found. Available parameters are [arg1, arg0, param1, param2]
    at org.apache.ibatis.exceptions.ExceptionFactory.wrapException(ExceptionFactory.java:30)
    at org.apache.ibatis.session.defaults.DefaultSqlSession.update(DefaultSqlSession.java:199)
    at org.apache.ibatis.binding.MapperMethod.execute(MapperMethod.java:67)
    at org.apache.ibatis.binding.MapperProxy.invoke(MapperProxy.java:57) <1 internal call>
```

这里是说，找不到我们定义的 username 和 id 这两个参数。同时，这个错误提示中指明，可用的参数名是 [arg1, arg0, param1, param2]，相当于我们自己给变量取的名字失效了，要使用系统提供的默认名字，默认名字实际上是两套体系：

第一套就是 arg0、arg1、、、、 第二套就是 param1、param2、、、

注意，这两个的下标是不一样的。

因此，按照错误提示，我们将参数改为下面这样：

```
<update id="updateUsernameById">
    update user set username = #{arg0} where id=#{arg1};
</update>
```

或者下面这样：

```
<update id="updateUsernameById">
    update user set username = #{param1} where id=#{param2};
</update>
```

但是，默认的名字不好记，容易出错，我们如果想要使用自己写的变量的名字，可以通过给参数添加 @Param 来指定参数名（一般在又多个参数的时候，需要加），一旦用 @Param 指定了参数类型之后，可以省略掉参数类型，就是在 xml 文件中，不用定义 parameterType 了：

```
Integer updateUsernameById(@Param("username") String username, @Param("id") Integer id);
```

这样定义之后，我们在 mapper.xml 文件中，就可以直接使用 username 和 id 来引用变量了。

对象参数

例如添加一个用户：

```
Integer addUser(User user);
```

对应的 mapper 文件如下：

```
<insert id="addUser" parameterType="org.javaboy.mybatis.model.User">
    insert into user (username,address,favorites) values (#{username},#{address},#
{favorites,typeHandler=org.javaboy.mybatis.typehandler.List2VarcharHandler});
</insert>
```

我们在引用的时候，直接使用属性名就能够定位到对象了。如果对象存在多个，我们也需要给对象添加 @Param 注解，如果给对象添加了 @Param 注解，那么对象属性的引用，会有一些变化。如下：

```
Integer addUser(@Param("user") User user);
```

如果对象参数添加了 @Param 注解，Mapper 中的写法就会发生变化：

```
<insert id="addUser" parameterType="org.javaboy.mybatis.model.User">
    insert into user (username,address,favorites) values (#{user.username},#{user.address},#
{user.favorites,typeHandler=org.javaboy.mybatis.typehandler.List2VarcharHandler});
</insert>
```

注意多了一个前缀，这个前缀不是变量名，而是 @Param 注解中定义名称。

Map 参数

一般不推荐在项目中使用 Map 参数。如果想要使用 Map 传递参数，技术上来说，肯定是没有问题的。

```
Integer updateUsernameById(HashMap<String,Object> map);
```

XML 文件写法如下：

```
<update id="updateUsernameById">
    update user set username = #{username} where id=#{id};
</update>
```

引用的变量名，就是 map 中的 key。基本上和实体类是一样的，如果给 map 取了别名，那么在引用的时候，也要将别名作为前缀加上，这一点和实体类也是一样的。

resultType

resultType 是返回类型，在实际开发中，如果返回的数据类型比较复杂，一般我们使用 resultMap，但是，对于一些简单的返回，使用 resultType 就够用了。

resultType 返回的类型可以是简单类型，可以是对象，可以是集合，也可以是一个 hashmap，如果是 hashmap，map 中的 key 就是字段名，value 就是字段的值。

输出 pojo 对象和输出 pojo 列表在 sql 中定义的 resultType 是一样的。 返回单个 pojo 对象要保证 sql 查询出来的结果集为单条，内部使用 sqlSession.selectOne 方法调用，mapper 接口使用 pojo 对象作为方法返回值。返回 pojo 列表表示查询出来的结果集可能为多条，内部使用 sqlSession.selectList 方法，mapper 接口使用 List 对象作为方法返回值。

resultMap

在实际开发中，resultMap 是使用较多的返回数据类型配置。因为实际项目中，一般的返回数据类型比较丰富，要么字段和属性对不上，要么是一对一、一对多的查询，等等，这些需求，单纯的使用 resultType 是无法满足的，因此我们还需要使用 resultMap，也就是自己定义映射的结果集。

先来看一个基本用法：

首先在 mapper.xml 中定义一个 resultMap：

```
<resultMap id="MyResultMap" type="org.javaboy.mybatis.model.User">
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="address" property="address"/>
</resultMap>
```

在这个 resultMap 中，id 用来描述主键，column 是数据库查询出来的列名，property 则是对象中的属性名。

然后在查询结果中，定义返回值时使用这个 ResultMap：

```
<select id="getUserById" resultMap="MyResultMap">
    select * from user where id=#{id};
</select>
```

注意，在旧版的 MyBatis 中，要求实体类一定要有一个无参构造方法，新版的 MyBatis 没有这个要求。

当然，我们也可以在 resultMap 中，自己指定要调用的构造方法，指定方式如下：

```
<resultMap id="MyResultMap" type="org.javaboy.mybatis.model.User">
    <constructor>
        <idArg column="id" name="id"/>
        <arg column="username" name="username"/>
    </constructor>
</resultMap>
```

这个就表示使用两个参数的构造方法取构造一个 User 实例。注意，name 属性表示构造方法中的变量名，默认情况下，变量名是 arg0、arg1、、、、或者 param1、param2、、、 ，如果需要自定义，我们可以在构造方法中，手动加上 @Param 注解。

```
public class User {
    private Integer id;
    private String username;
```

```
    private String address;
    private List<String> favorites;

    public User(@Param("id") Integer id, @Param("username") String username) {
        this.id = id;
        this.username = username;
        System.out.println("--------------------");
    }

    public User(Integer id, String username, String address, List<String> favorites) {
        this.id = id;
        this.username = username;
        this.address = address;
        this.favorites = favorites;
        System.out.println("-----------sdfasfd---------");
    }

    public List<String> getFavorites() {
        return favorites;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", address='" + address + '\'' +
                ", favorites=" + favorites +
                '}';
    }

    public void setFavorites(List<String> favorites) {
        this.favorites = favorites;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

动态 SQL

动态 SQL 是 MyBatis 中非常强大的一个功能。例如一些常见的查询场景：

- 查询条件不确定
- 批量插入
- ....

这些类似需求，我们都可以通过 MyBatis 提供的动态 SQL 来解决。

MyBatis 中提供的动态 SQL 节点非常多。

if

if 是一个判断节点，如果满足某个条件，节点中的 SQL 就会生效。例如分页查询，要传递两个参数，页码和查询的记录数，如果这两个参数都为 null，那我就查询所有。

我们首先来定义接口方法：

```
List<User> getUserByPage(@Param("start") Integer start, @Param("count") Integer count);
```

接口定义成功后，接下来在 XML 中定义 SQL：

```
<select id="getUserByPage" resultType="org.javaboy.mybatis.model.User">
    select * from user
    <if test="start !=null and count!=null">
        limit #{start},#{count}
    </if>
</select>
```

if 节点中，test 表示判断条件，如果判断结果为 true，则 if 节点的中的 SQL 会生效，否则不会生效。也就是说，在方法调用时，如果分页的两个参数都为 null，则表示查询所有数据：

where

where 用来处理查询参数。例如我存在下面一个查询函数：

```
List<User> getUserByUsernameAndId(@Param("id") Integer id, @Param("name") String name);
```

这个查询的复杂之处在于：每个参数都是可选的，如果 id 为 null，则表示根据 name 查询，name 为 null，则表示根据 id 查询，两个都为 null，表示查询所有。

```
<select id="getUserByUsernameAndId" resultType="org.javaboy.mybatis.model.User">
    select * from user
    <where>
        <if test="id!=null">
            and id>#{id}
        </if>
        <if test="name!=null">
            and username like concat('%',#{name},'%')
        </if>
    </where>
</select>
```

用 where 节点将所有的查询条件包起来，如果有满足的条件，where 节点会自动加上，如果没有，where 节点也将不存在，在有满足条件的情况下，where 还会自动处理 and 关键字。

foreach

foreach 用来处理数组/集合参数。

例如，我们有一个批量查询的需求：

```
List<User> getUserByIds(@Param("ids")Integer[] ids);
```

对应的 XML 如下：

```
<select id="getUserByIds" resultType="org.javaboy.mybatis.model.User">
    select * from user where id in
    <foreach collection="ids" open="(" close=")" item="id" separator=",">
        #{id}
    </foreach>
</select>
```

在 mapper 中，通过 foreach 节点来遍历数组，collection 表示数组变量，open 表示循环结束后，左边的符号，close 表示循环结束后，右边的符号，item 表示循环时候的单个变量，separator 表示循环的元素之间的分隔符。

注意，默认情况下，无论你的数组/集合参数名字是什么，在 XML 中访问的时候，都是 array，开发者可以通过 @Param 注解给参数重新指定名字。

例如我还有一个批量插入的需求：

```
Integer batchInsertUser(@Param("users") List<User> users);
```

然后，定义该方法对应的 mapper：

```
<insert id="batchInsertUser">
    insert into user (username,address) values
    <foreach collection="users" separator="," item="user">
        (#{user.username},#{user.address})
    </foreach>
</insert>
```

sql 片段

大家知道，在 SQL 查询中，一般不建议写 `*`，因为 select `*` 会降低查询效率。但是，每次查询都要把字段名列出来，太麻烦。这种使用，我们可以利用 SQL 片段来解决这个问题。

例如，我们先在 mapper 中定义一个 SQL 片段：

```
<sql id="Base_Column">
    id,usename,address
</sql>
```

然后，在其他 SQL 中，就可以引用这个变量：

```
<select id="getUserByIds" resultType="org.javaboy.mybatis.model.User">
    select <include refid="Base_Column" /> from user where id in
    <foreach collection="ids" open="(" close=")" item="id" separator=",">
        #{id}
    </foreach>
</select>
```

set

set 关键字一般用在更新中。因为大部分情况下，更新的字段可能不确定，如果对象中存在该字段的值，就更新该字段，不存在，就不更新。例如如下方法：

```
Integer updateUser(User user);
```

现在，这个方法的需求是，根据用户 id 来跟新用户的其他属性，所以，user 对象中一定存在 id，其他属性则不确定，其他属性要是有值，就更新，没值（也就是为 null 的时候），则不处理该字段。

我们结合 set 节点，写出来的 sql 如下：

```
<update id="updateUser" parameterType="org.javaboy.mybatis.model.User">
    update user
    <set>
        <if test="username!=null">
            username = #{username},
        </if>
        <if test="address!=null">
```

```
            address=#{address},
        </if>
        <if test="favorites!=null">
            favorites=#{favorites},
        </if>
    </set>
    where id=#{id};
</update>
```

查询进阶

一对一查询

在实际开发中，经常会遇到一对一查询，一对多查询等。这里我们先来看一对一查询。

例如：每本书都有一个作者，作者都有自己的属性，根据这个，我来定义两个实体类：

```
public class Book {
    private Integer id;
    private String name;
    private Author author;

    @Override
    public String toString() {
        return "Book{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", author=" + author +
                '}';
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
}
public class Author {
    private Integer id;
    private String name;
    private Integer age;

    @Override
    public String toString() {
```

```
        return "Author{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

然后，在数据库中，添加两张表：

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/`test01` /*!40100 DEFAULT CHARACTER SET utf8mb4 COLLATE
utf8mb4_general_ci */ /*!80016 DEFAULT ENCRYPTION='N' */;

USE `test01`;

/*Table structure for table `author` */

DROP TABLE IF EXISTS `author`;

CREATE TABLE `author` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

/*Data for the table `author` */

/*Table structure for table `book` */

DROP TABLE IF EXISTS `book`;

CREATE TABLE `book` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL,
  `aid` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

添加成功后，我们新建一个 BookMapper：

```
public interface BookMapper {
    Book getBookById(Integer id);
}
```

BookMapper 中定义了一个查询 Book 的方法，但是我希望查出来 Book 的同时，也能查出来它的 Author。再定义一个 BookMapper.xml ，内容如下：

```
<?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="org.javaboy.mybatis.mapper.BookMapper">

    <resultMap id="BookWithAuthor" type="org.javaboy.mybatis.model.Book">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <association property="author" javaType="org.javaboy.mybatis.model.Author">
            <id column="aid" property="id"/>
            <result column="aname" property="name"/>
            <result column="aage" property="age"/>
        </association>
    </resultMap>

    <select id="getBookById" resultMap="BookWithAuthor">
        SELECT b.*,a.`age` AS aage,a.`id` AS aid,a.`name` AS aname FROM book b,author a WHERE b.`aid`=a.`id` AND
b.`id`=#{id}
    </select>
</mapper>
```

在这个查询 SQL 中，首先应该做好一对一查询，然后，返回值一定要定义成 resultMap，注意，这里千万不能写错。然后，在 resultMap 中，来定义查询结果的映射关系。

其中，association 节点用来描述一对一的关系。这个节点中的内容，和 resultMap 一样，也是 id，result 等，在这个节点中，我们还可以继续描述一对一。

由于在实际项目中，每次返回的数据类型可能都会有差异，这就需要定义多个 resultMap，而这多个 resultMap 中，又有一部份属性是相同的，所以，我们可以将相同的部分抽出来，做成一个公共的模板，然后被其他 resultMap 继承，优化之后的 mapper 如下：

```
<?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="org.javaboy.mybatis.mapper.BookMapper">

    <resultMap id="BaseResultMap" type="org.javaboy.mybatis.model.Book">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
    </resultMap>
    <resultMap id="BookWithAuthor" type="org.javaboy.mybatis.model.Book" extends="BaseResultMap">
        <association property="author" javaType="org.javaboy.mybatis.model.Author">
            <id column="aid" property="id"/>
            <result column="aname" property="name"/>
            <result column="aage" property="age"/>
        </association>
    </resultMap>

    <select id="getBookById" resultMap="BookWithAuthor">
        SELECT b.*,a.`age` AS aage,a.`id` AS aid,a.`name` AS aname FROM book b,author a WHERE b.`aid`=a.`id` AND
b.`id`=#{id}
    </select>

</mapper>
```

懒加载

上面这种加载方式，是一次性的读取到所有数据。然后在 resultMap 中做映射。如果一对一的属性使用不是很频繁，可能偶尔用一下，这种情况下，我们也可以启用懒加载。

懒加载，就是先查询 book，查询 book 的过程中，不去查询 author，当用户第一次调用了 book 中的 author 属性后，再去查询 author。

例如，我们再来定义一个 Book 的查询方法：

```
Book getBookById2(Integer id);
Author getAuthorById(Integer id);
```

接下来，在 mapper 中定义相应的 SQL：

```
<resultMap id="BaseResultMap" type="org.javaboy.mybatis.model.Book">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
</resultMap>
<resultMap id="BookWithAuthor2" type="org.javaboy.mybatis.model.Book" extends="BaseResultMap">
    <association property="author" javaType="org.javaboy.mybatis.model.Author"
                 select="org.javaboy.mybatis.mapper.BookMapper.getAuthorById" column="aid" fetchType="lazy"/>
</resultMap>
<select id="getBookById2" resultMap="BookWithAuthor2">
    select * from book where id=#{id};
</select>
<select id="getAuthorById" resultType="org.javaboy.mybatis.model.Author">
    select * from author where id=#{aid};
</select>
```

这里，定义 association 的时候，不直接指定映射的字段，而是指定要执行的方法，通过 select 字段来指定，column 表示执行方法时传递的参数字段，最后的 fetchType 表示开启懒加载。

当然，要使用懒加载，还需在全局配置中开启：

```
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

一对多查询

一对多查询，也是一个非常典型的使用场景。比如用户和角色的关系，一个用户可以具备多个角色。

首先我们准备三个表：

```
SET FOREIGN_KEY_CHECKS=0;

-- ----------------------------
-- Table structure for role
-- ----------------------------
DROP TABLE IF EXISTS `role`;
CREATE TABLE `role` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) DEFAULT NULL,
  `nameZh` varchar(32) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of role
-- ----------------------------
INSERT INTO `role` VALUES ('1', 'dba', '数据库管理员');
INSERT INTO `role` VALUES ('2', 'admin', '系统管理员');
```

```
INSERT INTO `role` VALUES ('3', 'user', '用户');

-- ----------------------------
-- Table structure for user
-- ----------------------------
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) DEFAULT NULL,
  `password` varchar(255) DEFAULT NULL,
  `enabled` tinyint(1) DEFAULT NULL,
  `locked` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of user
-- ----------------------------
INSERT INTO `user` VALUES ('1', 'root', '$2a$10$RMuFXGQ5AtH4wOvkUqyvuecpqUSeoxZYqilXzbz50dceRsga.WYiq', '1',
'0');
INSERT INTO `user` VALUES ('2', 'admin', '$2a$10$RMuFXGQ5AtH4wOvkUqyvuecpqUSeoxZYqilXzbz50dceRsga.WYiq', '1',
'0');
INSERT INTO `user` VALUES ('3', 'sang', '$2a$10$RMuFXGQ5AtH4wOvkUqyvuecpqUSeoxZYqilXzbz50dceRsga.WYiq', '1',
'0');

-- ----------------------------
-- Table structure for user_role
-- ----------------------------
DROP TABLE IF EXISTS `user_role`;
CREATE TABLE `user_role` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `uid` int(11) DEFAULT NULL,
  `rid` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of user_role
-- ----------------------------
INSERT INTO `user_role` VALUES ('1', '1', '1');
INSERT INTO `user_role` VALUES ('2', '1', '2');
INSERT INTO `user_role` VALUES ('3', '2', '2');
INSERT INTO `user_role` VALUES ('4', '3', '3');
SET FOREIGN_KEY_CHECKS=1;
```

这三个表中，有用户表，角色表以及用户角色关联表，其中用户角色关联表用来描述用户和角色之间的关系，他们是一对多的关系。

然后，根据这三个表，创建两个实体类：

```
public class User {
    private Integer id;
    private String username;
    private String password;
    private List<Role> roles;

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                ", roles=" + roles +
```

```
                '}';
    }

    public List<Role> getRoles() {
        return roles;
    }

    public void setRoles(List<Role> roles) {
        this.roles = roles;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
public class Role {
    private Integer id;
    private String name;
    private String nameZh;

    @Override
    public String toString() {
        return "Role{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", nameZh='" + nameZh + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public String getNameZh() {
        return nameZh;
    }

    public void setNameZh(String nameZh) {
        this.nameZh = nameZh;
    }
}
```

接下来，定义一个根据 id 查询用户的方法：

```
User getUserById(Integer id);
```

然后，定义该方法的实现：

```
<resultMap id="UserWithRole" type="org.javaboy.mybatis.model.User">
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <collection property="roles" ofType="org.javaboy.mybatis.model.Role">
        <id property="id" column="rid"/>
        <result property="name" column="rname"/>
        <result property="nameZh" column="rnameZH"/>
    </collection>
</resultMap>
<select id="getUserById" resultMap="UserWithRole">
    SELECT u.*,r.`id` AS rid,r.`name` AS rname,r.`nameZh` AS rnameZh FROM USER u,role r,user_role ur WHERE
u.`id`=ur.`uid` AND ur.`rid`=r.`id` AND u.`id`=#{id}
</select>
```

在 resultMap 中，通过 collection 节点来描述集合的映射关系。在映射时，会自动将一的一方数据集合并，然后将多的一方放到集合中，能实现这一点，靠的就是 id 属性。

当然，这个一对多，也可以做成懒加载的形式，那我们首先提供一个角色查询的方法：

```
List<Role> getRolesByUid(Integer id);
```

然后，在 XML 文件中，处理懒加载：

```
<resultMap id="UserWithRole" type="org.javaboy.mybatis.model.User">
    <id column="id" property="id"/>
    <result column="username" property="username"/>
    <result column="password" property="password"/>
    <collection property="roles" select="org.javaboy.mybatis.mapper.UserMapper.getRolesByUid" column="id"
fetchType="lazy">
    </collection>
</resultMap>
<select id="getUserById" resultMap="UserWithRole">
    select * from user  where id=#{id};
</select>
<select id="getRolesByUid" resultType="org.javaboy.mybatis.model.Role">
    SELECT r.* FROM role r,user_role ur WHERE r.`id`=ur.`rid` AND ur.`uid`=#{id}
</select>
```

查询缓存

Mybatis 一级缓存的作用域是同一个 SqlSession，在同一个 sqlSession 中两次执行相同的 sql 语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。当一个 sqlSession 结束后该 sqlSession 中的一级缓存也就不存在了。Mybatis 默认开启一级缓存。

```java
public class Main2 {
    public static void main(String[] args) {
        SqlSessionFactory instance = SqlSessionFactoryUtils.getInstance();
        SqlSession sqlSession = instance.openSession();
        BookMapper mapper = sqlSession.getMapper(BookMapper.class);
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        User user = userMapper.getUserById(1);
        user = userMapper.getUserById(1);
        user = userMapper.getUserById(1);
        System.out.println(user.getUsername());
    }
}
```

多次查询，只执行一次 SQL。但是注意，如果开启了一个新的 SqlSession，则新的 SqlSession 无法就是之前的缓存，必须是同一个 SqlSession 中，缓存才有效。

Mybatis 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 namespace 下的 sql 语句且向 sql 中传递参数也相同即最终执行相同的 sql 语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。Mybatis 默认没有开启二级缓存需要在 setting 全局参数中配置开启二级缓存。