MyBatis教程

一、了解MyBatis

1、历史（百度百科）

- MyBatis 本是apache的一个开源项目【iBatis】, 2010年这个项目由apache software foundation（Apache软件基金会） 迁移到了google code（谷歌的代码托管平台），并且改名为MyBatis ， 2013年11月迁移到Github。

2、作用 （百度百科）

- MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。

3、说说持久化

持久化是将程序数据在持久状态和瞬时状态间转换的机制。通俗的讲，就是瞬时数据（比如内存中的数据，是不能永久保存的）持久化为持久数据（比如持久化至数据库中，能够长久保存）。

1. 程序产生的数据首先都是在内存。
2. 内存是不可靠的，他丫的一断电数据就没了。
3. 那可靠的存储地方是哪里？硬盘、U盘、光盘等。

4. 我们的程序在运行时说的持久化通常就是指将内存的数据存在硬盘。

4、说说持久层

其实分层的概念已经谈到过：

- 业务是需要操作数据的
- 数据是在磁盘上的
- 具体业务调用具体的数据库操作，耦合度太高，复用性太差
- 将操作数据库的代码统一抽离出来，自然就形成了介于业务层和数据库中间的独立的层

5、聊聊ORM

ORM，即Object-Relational Mapping（对象关系映射），它的作用是在关系型数据库和业务实体对象之间作一个映射，这样，我们在具体的操作业务对象的时候，就不需要再去和复杂的SQL语句打交道，只需简单的操作对象的属性和方法。

- **jpa**（Java Persistence API）是java持久化规范，是orm框架的标准，主流orm框架都实现了这个标准。
- **hibernate**：全自动的框架，强大、复杂、笨重、学习成本较高，不够灵活，实现了jpa规范。Java Persistence API（Java 持久层 API）
- **MyBatis**：半自动的框架(懂数据库的人 才能操作) 必须要自己写sql，不是依照的jpa规范实现的。

很多人青睐 MyBatis，原因是其提供了便利的 SQL 操作，自由度高，封装性好…… JPA对复杂 SQL 的支持不好，没有实体关联的两个表要做 join，的确要花不少功夫。

6、MyBatis的优点和缺点

- sql语句与代码分离，存放于xml配置文件中：

  优点：便于维护管理，不用在java代码中找这些语句；

  缺点： JDBC方式可以用打断点的方式调试，但是MyBatis调试比较复杂，一般要通过log4j日志输出日志信息帮助调试，然后在配置文件中修改。

- 用逻辑标签控制动态SQL的拼接：

  优点：用标签代替编写逻辑代码；

  缺点：拼接复杂SQL语句时，没有代码灵活，拼写比较复杂。不要使用变通的手段来应对这种复杂的语句。

- 查询的结果集与java对象自动映射：

  优点：保证名称相同，配置好映射关系即可自动映射或者，不配置映射关系，通过配置列名=字段名也可完成自动映射。

  缺点：对开发人员所写的SQL依赖很强。

- 编写原生SQL：

  优点：接近JDBC，比较灵活。

  缺点：对SQL语句依赖程度很高；并且属于半自动，数据库移植比较麻烦，比如MySQL数据库编程Oracle数据库，部分的SQL语句需要调整。

- 最重要的一点，使用的人多！公司需要！

二、搭建个环境

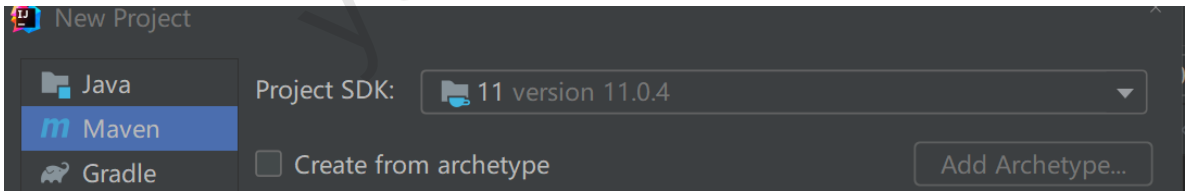
1、建立数据库

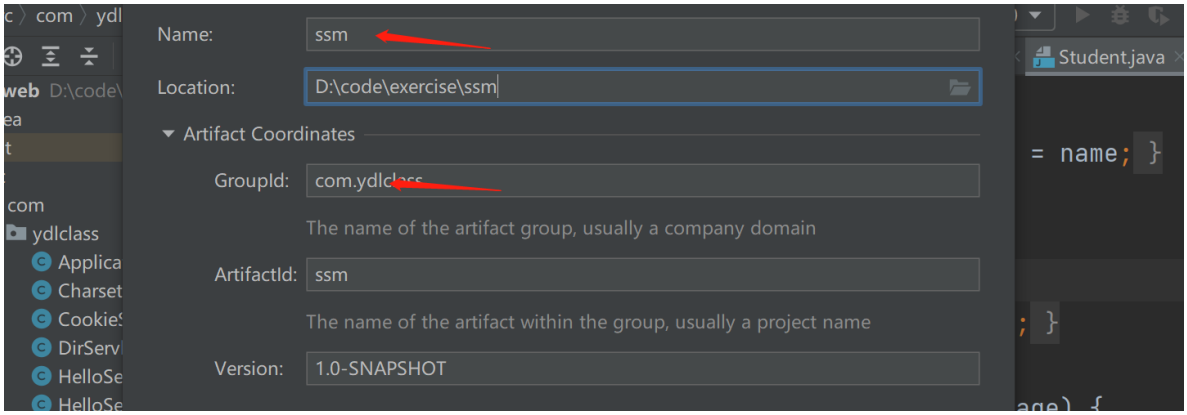
```
CREATE DATABASE `ssm`;
USE `ssm`;
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
`id` int(20) NOT NULL,
`username` varchar(30) DEFAULT NULL,
`password` varchar(30) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert  into `user`(`id`,`username`,`password`)
values (1,'itnanls','123456'),(2,'itlils','abcdef'),
(3,'ydlclass','987654');
```


2、构建一个父工程

尝试学习聚合工程的规范，当然可以搭建独立的工程，我们选择一个比较新的jdk版本11：





将父工程的打包方式修改成pom，表示一个聚合工程：

```
<packaging>pom</packaging>
```

3、父工程的maven配置

```
<?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.ydlclass</groupId>
    <artifactId>ssm</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <!-- 父模块用于约束版本信息 -->
    <properties>

 <maven.compiler.source>11</maven.compiler.source>

 <maven.compiler.target>11</maven.compiler.target>
```

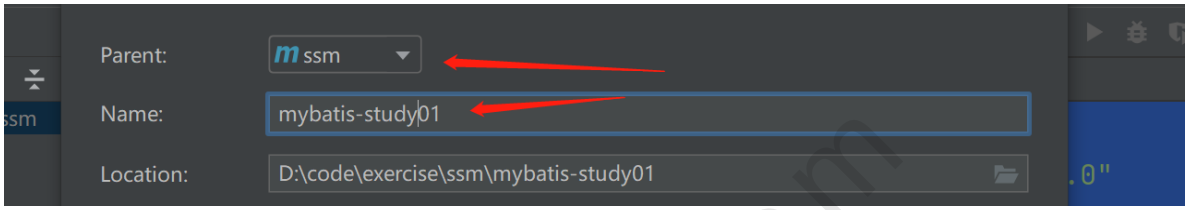
```
        <junit.version>4.13.1</junit.version>
        <mybatis.version>3.5.7</mybatis.version>
        <mysql-connector-java.version>8.0.26</mysql-
connector-java.version>
        <lombok.version>1.18.22</lombok.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <!-- 单元测试 -->
            <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>${junit.version}</version>
                <scope>test</scope>
            </dependency>
            <!-- mybatis 核心 -->
            <dependency>
                <groupId>org.mybatis</groupId>
                <artifactId>mybatis</artifactId>
                <version>${mybatis.version}
</version>
            </dependency>
            <!-- 数据库确定 -->
            <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-
java</artifactId>
                <version>${mysql-connector-
java.version}</version>
                <scope>runtime</scope>
            </dependency>
            <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>${lombok.version}</version>
                <scope>provided</scope>
```

```
            </dependency>
        </dependencies>
    </dependencyManagement>

</project>
```

4、创建子模块



pom

```
<properties>

 <maven.compiler.source>11</maven.compiler.source>

 <maven.compiler.target>11</maven.compiler.target>
</properties>
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-
java</artifactId>
        <scope>runtime</scope>
```

```
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>

<!-- 处理资源被过滤问题 -->
<build>
    <plugins>
        <plugin>

 <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-
plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>${maven.compiler.target}
</source> <!-- 源代码使用的JDK版本 -->
                <target>${maven.compiler.target}
</target> <!-- 需要生成的目标class文件的编译版本 -->
                 <encoding>UTF-8</encoding><!-- 字符集
编码 -->
            </configuration>
        </plugin>
    </plugins>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
```

```
            </includes>
            <filtering>false</filtering>
        </resource>
        <resource>

 <directory>src/main/resources</directory>
            <includes>
                <include>**/*.properties</include>
                <include>**/*.xml</include>
            </includes>
            <filtering>false</filtering>
        </resource>
    </resources>
</build>
```

5、回顾我们的jdbc代码

```
@Test
public void testConnection1() throws Exception{
    //1.数据库连接的4个基本要素：
    String url = "jdbc:mysql://localhost:3306/ssm?
characterEncoding=utf8&serverTimezone=Asia/Shanghai"
;
    String username = "root";
    String password = "root";
    //8.0之后名字改了  com.mysql.cj.jdbc.Driver
    String driverName = "com.mysql.cj.jdbc.Driver";

    //2.实例化Driver
    Class clazz = Class.forName(driverName);
    Driver driver = (Driver) clazz.newInstance();
    //3.注册驱动
    DriverManager.registerDriver(driver);
    //4.获取连接
```

```
    Connection conn =
DriverManager.getConnection(url, username,
password);

    PreparedStatement preparedStatement =
conn.prepareStatement("select * from user where id =
?");
    preparedStatement.setInt(1,1);
    ResultSet resultSet =
preparedStatement.executeQuery();

    // 处理结果集
    while (resultSet.next()){
        User user = new User();
        user.setId(resultSet.getInt("id"));

 user.setUsername(resultSet.getString("username"));

 user.setPassword(resultSet.getString("password"));
        System.out.println(user);
    }
}
```


6、编写MyBatis核心配置文件，mybatis-config.xml

> 有兴趣的自行深入研究。

1. UNPOOLED：不使用连接池的数据源
2. POOLED：使用连接池的数据源

3. JNDI：使用JNDI实现的数据源，我们在学习JavaEE的时候学习过了

配置文件我们从官网复制：https://mybatis.org/mybatis-3/zh/getting-started.html

```
<?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>
    <properties>
        <property name="driver"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="url"
value="jdbc:mysql://localhost:3306/ssm?
characterEncoding=utf8&amp;serverTimezone=Asia/Shang
hai"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </properties>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver"
value="${driver}"/>
                <property name="url"
value="${url}"/>
                <property name="username"
value="${username}"/>
                <property name="password"
value="${password}"/>
            </dataSource>
        </environment>
```

```
    </environments>
</configuration>
```

> 小知识：

（1）DTD：

DTD(Document Type Definition)即文档类型定义，是一种XML约束模式语言，是XML文件的验证机制

如下所示是公共DTD示例。

```
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-
config.dtd">
```

关于DTD的声明解释如下：

1、DTD声明始终以!DOCTYPE开头，空一格后跟着文档根元素的名称。

2、根元素名：configuration。所以每一个标签库定义文件都是以taglib为根元素的，否则就不会验证通过。

3、PUBLIC "-//mybatis.org//DTD Config 3.0//EN，这是一个公共DTD的名称（私有的使用SYSTEM表示）。这个东西命名是有些讲究的。首先它是以"-"开头的，表示这个DTD不是一个标准组织制定的。（如果是ISO标准化组织批准的，以“ISO”开头）。接着就是双斜杠“//”，跟着的是DTD所有者的名字，很明显这个DTD是MyBatis公司定的。接着又是双斜杠“//”，然后跟着的是DTD描述的文档类型，可以看出这份DTD描述的是DTD Config 3.0的格式。再跟着的就是“//”和ISO 639语言标识符。

4、绿色的字"http://mybatis.org/dtd/mybatis-3-config.dtd"，表示这个DTD的位置。

疑问：是不是xml分析器都会到java.sun.com上去找这个dtd呢？答案是否定的，xml分析器首先会以某种机制查找公共DTD的名称，查到了，则以此为标准，如果查不到，再到DTD位置上去找。

(2) XSD

文档结构描述XML Schema Definition 缩写，这种文件同样可以用来定义我们xml文件的结构！

我们看看pom文件的xml头部：

```
<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
">
```


1、第一行的xmlns代表了一个xml文件中的一个命名空间，通常是一个唯一的字符串，一般使用一个url，因为不会重复嘛。

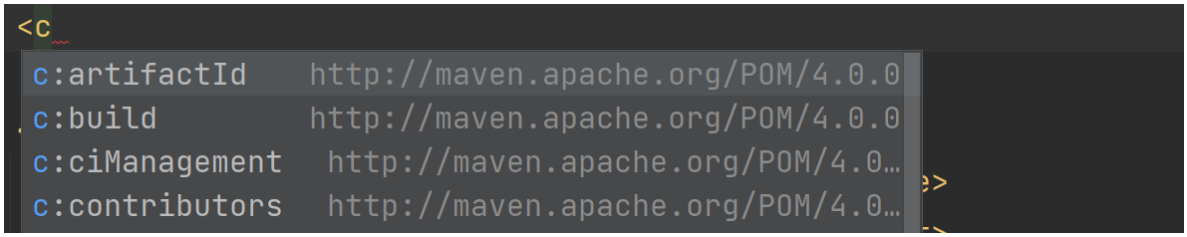
它的语法如下：

```
xmlns:namespace-prefix="namespaceURI"
```

后边什么也不加，代表默认命名空间，我们在书写标签的时候不需要加任何前缀。

如果我将其改为：

```
xmlns:c="http://maven.apache.org/POM/4.0.0"
```



2、xmlns:xsi 定义了一个命名空间前缀 xsi 对应的唯一字符串 http://www.w3.org/2001/XMLSchema-instance。但这个 xmlns:xsi 在不同的 xml 文档中似乎都会出现。 这是因为， xsi 已经成为了一个业界默认的用于 XSD(（XML Schema Definition) 文件的命名空间。 而 XSD 文件（也常常称为 Schema 文件）是用来定义 xml 文档结构的。剩余两行的目的在于为我们的命名空间指定对应的xsd文件。

事实上我们这么写也是可以的：

```
        xmlns:a="http://www.w3.org/2001/XMLSchema-instance"
        a:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
```

上面这行的语法其实是， xsi:schemaLocation = "ns1url xsd1 ns2url xsd2”

XML Schema相对于DTD的优点在于:

1. XML Schema基于XML，没有专门的语法。
2. XML Schema可以像其他XML文件一样解析和处理。
3. XML Schema比DTD提供了更丰富的数据类型。
4. XML Schema提供可扩充的数据模型。
5. XML Schema支持综合命名空间。
6. XML Schema支持属性组。

7、编写实体类

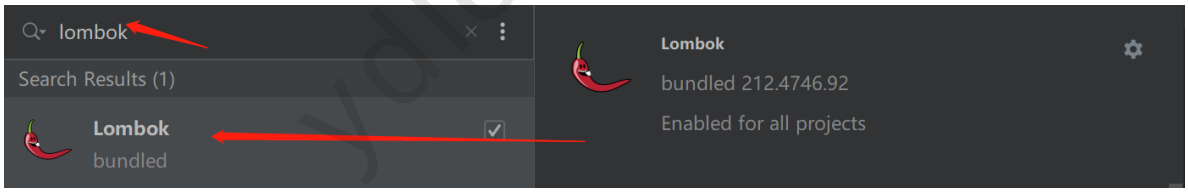
> lombok

平时的工作中写setter和getter以及toString方法是不是已经烦了，每次添加一个字段都要重新添加这些方法。

今天我们学习一个神器，从此再也不用写这些重复的代码了，它们在编译的时候动态的帮我们生成这些代码。

1. javac对源代码进行分析，生成了一棵抽象语法树（AST）
2. 运行过程中调用实现了“JSR 269 API”的Lombok程序
3. 此时Lombok就对第一步骤得到的AST进行处理，找到@Data注解所在类对应的语法树（AST），然后修改该语法树（AST），增加getter和setter方法定义的相应树节点
4. javac使用修改后的抽象语法树（AST）生成字节码文件，即给class增加新的节点（代码块）

1，首先，我们必须安装一个插件，否则编译的时候会报错，你没有写setter方法，又去调用它当然不能编译：



2、引入依赖，lombok在编译的时候，会根据我们的注解动态生成我们需要的构造方法，setter和getter等，运行的时候就没用了。所以scope选择provided。

```
<!--
https://mvnrepository.com/artifact/org.projectlombok
/lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.16</version>
    <scope>provided</scope>
</dependency>
```

从今往后，只需要在对应的类上加上这几个注解，就能完成对应的编译工作

- @AllArgsConstructor：生成全参构造器。
- @NoArgsConstructor：生成无参构造器。
- @Getter/@Setter： 作用类上，生成所有成员变量的getter/setter方法；作用于成员变量上，生成该成员变量的getter/setter方法。可以设定访问权限及是否懒加载等。
- @Data：作用于类上，是以下注解的集合：@ToString @EqualsAndHashCode @Getter @Setter @RequiredArgsConstructor
- @Log：作用于类上，生成日志变量。针对不同的日志实现产品，有不同的注解。

注解还有很多，自行学习。

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User implements Serializable {

    private static final Long serialVersionUID = 1L;

    private int id;
    private String username;
    private String password;
}
```

此时我们的User是不是变得很简洁呢?

我们随便写一个main方法，然后编译一下:

编译后的结果是这个样子的:

```
public class User implements Serializable {
    private static final Long serialVersionUID = 1L;
    private int id;
    private String username;
    private String password;

    public static void main(String[] args) {
    }

    public int getId() {
        return this.id;
    }

    public String getUsername() {
        return this.username;
    }

    public String getPassword() {
        return this.password;
```

```
    }

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

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

    public void setPassword(String password) {
        this.password = password;
    }

    public boolean equals(Object o) {
        if (o == this) {
            return true;
        } else if (!(o instanceof User)) {
            return false;
        } else {
            User other = (User)o;
            if (!other.canEqual(this)) {
                return false;
            } else if (this.getId() !=
other.getId()) {
                return false;
            } else {
                Object this$username =
this.getUsername();
                Object other$username =
other.getUsername();
                if (this$username == null) {
                    if (other$username != null) {
                        return false;
                    }
                } else if
(!this$username.equals(other$username)) {
```

```
                    return false;
                }

                Object this$password =
this.getPassword();
                Object other$password =
other.getPassword();
                if (this$password == null) {
                    if (other$password != null) {
                        return false;
                    }
                } else if
(!this$password.equals(other$password)) {
                    return false;
                }

                return true;
            }
        }
    }

    protected boolean canEqual(Object other) {
        return other instanceof User;
    }

    public int hashCode() {
        int PRIME = true;
        int result = 1;
        int result = result * 59 + this.getId();
        Object $username = this.getUsername();
        result = result * 59 + ($username == null ?
43 : $username.hashCode());
        Object $password = this.getPassword();
        result = result * 59 + ($password == null ?
43 : $password.hashCode());
        return result;
    }
```

```
    public String toString() {
        int var10000 = this.getId();
        return "User(id=" + var10000 + ", username="
+ this.getUsername() + ", password=" +
this.getPassword() + ")";
    }

    public User(int id, String username, String
password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }

    public User() {
    }
}
```

我们发现，编译后注解没了，其他的都有了，自然运行时就能调用了呀！

三、日志配置

> 配置日志的一个重要原因是想在调试的时候能观察到sql语句的输出，能查看中间过程

1、标准日志实现

指定 MyBatis 应该使用哪个日志记录实现。如果此设置不存在，则会自动发现日志记录实现。

STD：standard out：输出

STDOUT_LOGGING：标准输出日志

```
<settings>
    <setting name="logImpl" value="STDOUT_LOGGING"/>
</settings>
```

这就好了，执行一下看看。

2、组合logback完成日志功能（扩展）

使用步骤：

1、导入log4j的包

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>
```

2、配置文件编写 log4j.properties

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <property name="pattern" value="%d{yyyy-MM-dd
HH:mm:ss} %c [%thread] %-5level %msg%n"/>
    <property name="log_dir" value="d:/logs" />

    <appender name="console"
class="ch.qos.logback.core.ConsoleAppender">
```

```
        <target>System.out</target>
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutE
ncoder">
            <pattern>${pattern}</pattern>
        </encoder>
    </appender>

    <appender name="file"
class="ch.qos.logback.core.FileAppender">
        <!--日志格式配置-->
        <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutE
ncoder">
            <pattern>${pattern}</pattern>
        </encoder>
        <!--日志输出路径-->
        <file>${log_dir}/sql.log</file>
    </appender>

    <root level="ALL">
        <appender-ref ref="console"/>
    </root>

    <logger name="mybatis.sql" level="debug"
additivity="false">
        <appender-ref ref="console"/>
        <appender-ref ref="file"/>
    </logger>

</configuration>
```

3、setting设置日志实现

```
<settings>
   <setting name="logImpl" value="SLF4J"/>
</settings>
```

四、CRUD来一套

1、基本流程：

```
// 1、创建一个SqlSessionFactory的 建造者 ，用于创建SqlSessionFactory
// SqlSessionFactoryBuilder中有大量的重载的build方法，可以根据不同的入参，进行构建
// 极大的提高了灵活性，此处使用【创建者设计模式】
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
// 2、使用builder构建一个sqlSessionFactory，此处我们基于一个xml配置文件
// 此过程会进行xml文件的解析，过程相对比较复杂
SqlSessionFactory sqlSessionFactory = builder.build(Thread.currentThread().getContextClassLoader().getResourceAsStream("mybatis-config.xml"));
// 3、通过sqlSessionFactory获取另一个session，此处使用【工厂设计模式】
SqlSession sqlSession = sqlSessionFactory.openSession();
```

1、创建一个SqlSessionFactory的 建造者 ，用于创建SqlSessionFactory

2、使用builder构建一个sqlSessionFactory，此处我们基于一个xml配置文件

3、通过sqlSessionFactory获取另一个session，此处使用【工厂设计模式】

4、一个sqlsession就是一个会话，可以使用sqlsession对数据库进行操作，原理后边会讲。

其实第一次使用sqlsession我们可能会这么操作：

```
try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
    List<Object> objects =
sqlSession.selectList("select * from user");
}
```

（1）SqlSessionFactory

每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先配置的 Configuration 实例来构建出 SqlSessionFactory 实例。

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。 使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。因此 SqlSessionFactory 的最佳作用域是应用作用域。 有很多方法可以做到，最简单的就是使用单例模式。

（2）SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。 绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。 也绝不能将 SqlSession 实例的引用放在任何类型的托管作用域中，比如 Servlet 框架中的 HttpSession。 换句话说，每次收到 HTTP 请求，就可以打开一个 SqlSession，返回一个响应后，就关闭它。 这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到 finally 块中。 下面的示例就是一个确保 SqlSession 关闭的标准模式：

```
try (SqlSession session =
sqlSessionFactory.openSession()) {
  // 你的应用逻辑代码
}
```

在所有代码中都遵循这种使用模式，可以保证所有数据库资源都能被正确地关闭。

(3) 测试

```
private SqlSessionFactory sqlSessionFactory = null;
private static final Logger LOGGER =
LoggerFactory.getLogger(TestMybatis.class);

@Before
public void build() {
    SqlSessionFactoryBuilder builder = new
SqlSessionFactoryBuilder();
    sqlSessionFactory =
builder.build(Thread.currentThread().getContextClass
Loader().getResourceAsStream("mybatis-config.xml"));
}
```

当我们看到sqlSession有selectList，delete，update等方法时，我们会忍不住这样去使用。

```
@Test
public void testSession(){
    try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
        List<Object> objects =
sqlSession.selectList("select * from user");
        System.out.println(objects);
    }
}
```

但是这样确实是错误的。

```
org.apache.ibatis.exceptions.PersistenceException:
### Error querying database.  Cause:
java.lang.IllegalArgumentException: Mapped
Statements collection does not contain value for
select * from user
### Cause: java.lang.IllegalArgumentException:
Mapped Statements collection does not contain value
for select * from user
```


错误消息中显示Mapped Statements collection does not contain value for select * from user，说是mapper的申明中没有“select * from user”，其实这里让你填的是一个申明。我们在源码注释中可以看到：】

```
/**
* Retrieve a list of mapped objects from the
statement key.
* @param <E> the returned list element type
* @param statement Unique identifier matching the
statement to use.//与要使用的语句匹配的唯一标识符。
* @return List of mapped object
*/
<E> List<E> selectList(String statement);
```

说明这个还需要通过使用sql的一个标识符。在MyBatis中我们还需要一个sql的映射文件来给每一个sql语句定义一个唯一标识符，我们起名UserMapper.xml，将这个文件放在resources文件夹下的mapper文件夹下：

```
<?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="UserMapper">
    <select id="selectOne"
resultType="com.ydlclass.User">
        select * from user where id = #{id}
    </select>
</mapper>
```

还需要讲这个配置文件注册到主配置文件中，在最后边添加如下代码：

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

我们会发现每一个mapper映射文件都有一个命名空间，从下边的用法来看我们能大致明白命名空间的作用。就如同我们国家可以有相同名字的县，但是加上市以后我们就能很轻松的区别两个县，这个市就是县的命名空间。

```
try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
    List<Object> users =
sqlSession.selectList("UserMapper.selectAll");
    LOGGER.debug("result is [{}]",objects);
}
```

我们得到了正确的结果：

```
DEBUG com.ydlclass.UserMapper.findAllUser - <==      Total: 11
DEBUG TestMybatis - result is [[User{id=1, username='zs', password='12'}, User{i
```

当然不加命名空间也可，因为我们并没有其他重复的标识符，这个selectOne就是这条语句的标识符。

```
try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
    List<Object> users =
sqlSession.selectList("selectAll");
    LOGGER.debug("result is [{}]",objects);
}
```

（4）动态代理实现

但是MyBatis给我们提供了更好的解决方案，这种方案使用动态代理的技术实现，后边会详细讲。我们可以这样：

1、定义一个接口：

```
public interface UserMapper {
    List<User> selectAll();
}
```

2、修改映射文件，让命名空间改为接口的权限定名，id改为方法的名字

```
<mapper namespace="com.ydlclass.mapper.UserMapper">
    <select id="selectAll"
resultType="com.ydlclass.entity.User">
        select * from user
    </select>
</mapper>
```

3、我们可以很简单的使用如下的方法操作：

```
try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    List<User> list = mapper.selectAll();
    LOGGER.debug("result is [{}]",list);
}
```

这样写起来简直不要太舒服，也确实拿到了对应的结果：

```
DEBUG com.ydlclass.UserMapper.findAllUser - <==      Total: 11
DEBUG TestMybatis - result is [[User{id=1, username='zs', password='12'}, User{i
```

这里很明显使用了动态代理的方式，

`sqlSession.getMapper(UserMapper.class);`帮我们生成一个代理对象，该对象实现了这个接口的方法，具体的数据库操作比如建立连接，创建statment等重复性的工作交给框架来处理，唯一需要额外补充的就是sql语句了，xml文件就是在补充这个描述信息，比如具体的sql，返回值的类型等，框架会根据命名空间自动匹配对应的接口，根据id自动匹配接口的方法，不需要我们再做额外的操作。

接下来我们就把增删改查全部写一下，感受一下：

2、select（查询）

> select标签是mybatis中最常用的标签

1、在UserMapper中添加对应方法

```
/**
     * 根据id查询用户
     * @param id
     * @return
     */
User selectUserById(int id);
```

2、在UserMapper.xml中添加Select语句

```
<select id="selectUserById"
resultType="com.ydlclass.entity.User"
 parameterType="int">
    select id,username,password from user where id =
#{id}
</select>
```


> 新的知识点，在映射文件中有一些属性：

- resultType：指定返回类型，查询是有结果的，结果啥类型，你得告诉我
- parameterType：指定参数类型，查询是有参数的，参数啥类型，你得告诉我
- id：指定对应的方法映射关系，就是你得告诉我你这sql对应的是哪个方法
- #{id}：sql中的变量，要保证大括号的变量必须在User对象里有

- #{}：占位符，其实就是咱们的【**PreparedStatement**】处理这个变量，mybatis会将它替换成`?`

> 除了#{}还有${}，看看有啥区别，面试常问

- #{} 的作用主要是替换预编译语句(PrepareStatement)中的占位符? 【推荐使用】

```
INSERT INTO user (username) VALUES (#{username});
INSERT INTO user (username) VALUES (?);
```

- ${} 的作用是直接进行字符串替换

```
INSERT INTO user (username) VALUES
('${username}');
INSERT INTO user (username) VALUES ('楠哥');
```

3、测试类中测试

```
try (SqlSession sqlSession =
sqlSessionFactory.openSession();){
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    User user = mapper.selectUserById(1);
    LOGGER.debug("the user is [{}]",user);
}
```

结果正确：

```
"C:\Program Files\Java\jdk-11.0.4_windows-x64_bin\jdk-11.0.4\bin\java.exe" ...
User(id=1, username=itnanls, password=123456)
```

我们不妨把session的创建定义成一个方法：

```
private static SqlSession open(){
    // 1、创建一个SqlSessionFactory的 建造者 ，用于创建SqlSessionFactory
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    // 2、使用builder构建一个sqlSessionFactory，此处我们基于一个xml配置文件
    SqlSessionFactory sqlSessionFactory = builder.build(Thread.currentThread().getContextClassLoader().getResourceAsStream("mybatis-config.xml"));
    // 3、通过sqlSessionFactory获取另一个session，此处使用【工厂设计模式】
    return sqlSessionFactory.openSession();
}
```


3、insert（插入）

> insert标签被用作插入操作

1、接口中添加方法

```
/**
     * 新增user
     * @param user
     * @return
     */
int addUser(User user);
```

2、xml中加入insert语句

```
<insert id="addUser"
parameterType="com.ydlclass.entity.User">
    insert into user (id,username,password) values
(#{id},#{username},#{password})
</insert>
```

3、测试

```
@Test
public void testAdd(){
    SqlSession sqlSession = open();
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    int rows = mapper.addUser(new User(10, "lucy",
"123"));
    LOGGER.debug("Affected rows: [{}]",rows);
}
```

返回值是1，你欣喜若狂，总以为就是这么简单，但事实是，数据库压根没有存进去：

id	username	password
1	itnanls	123456
2	itlils	abcdef
3	ydlclass	987654

注：增、删、改操作需要提交事务！在默认情况下MySQL的事务是自动提交的，而框架却默认设置成了手动提交，我们开启了事务，又没有去提交事务，结束后自然会回滚啊：

第一种方式，在openSession方法传入true，就变成自动提交了：

```
sqlSessionFactory.openSession(true);
```

第二种方式，我们手动提交，事实上我们肯定要手动提交事务：

```
@Test
public void testAdd(){
    SqlSession sqlSession = open();
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    int rows = mapper.addUser(new User(10, "lucy",
"123"));
    LOGGER.debug("Affected rows: [{}]",rows);
    sqlSession.commit();
    sqlSession.close();
}
```

> 思考，如果参数没有传实体类而是传了多个参数，，能不能执行

比如数据库为id，方式传入userId

1、在UserMapper中添加对应方法

```
/**
     * 新增用户
     * @param id
     * @param name
     * @param pws
     * @return
     */
int insertUser(int id,String name,String pws);
```

2、在UserMapper.xml中添加Select语句

```
<insert id="insertUser"
parameterType="com.ydlclass.entity.User">
    insert into user (id,username,password) values
(#{id},#{username},#{password})
</insert>
```

3、测试

```
@Test
    public void testInsert(){
        SqlSession sqlSession = open();
        UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
        int rows = mapper.insertUser(10, "lucy",
"123");
        LOGGER.debug("Affected rows: [{}]",rows);
        sqlSession.commit();
        sqlSession.close();
    }
```

出问题了

```
 Cause: org.apache.ibatis.binding.BindingException:
Parameter 'id' not found. Available parameters are
[arg2, arg1, arg0, param3, param1, param2]
```

这就无法映射了。

这就需要注解@Param了，这其实就是在做映射关系，xml里的${username}和方法中的name做映射：

```
int insertUser(@Param("id") int id,
@Param("username") String name,@Param("password")
String pws);
```

此时又一次成功了：

```
@Test
public void testInsert(){
    SqlSession sqlSession = open();
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    int rows = mapper.insertUser(11, "lucy", "123");
    LOGGER.debug("Affected rows: [{}]",rows);
    sqlSession.commit();
    sqlSession.close();
}
```

所以我们遇到mapper中有多个参数时，一定要使用@Param注解，建立映射关系。

4、update（修改）

> update标签用于更新操作

1、写接口

```
/**
 * 修改用户
 * @param user
 * @return
 */
int updateUser(User user);
```

2、写SQL

```
<update id="updateUser"
parameterType="com.ydlclass.entity.User">
    update user set username=#{username},password=#
{password} where id = #{id}
</update>
```

3、测试

```
@Test
public void testUpdate(){
    SqlSession sqlSession = open();
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    int rows = mapper.updateUser(new User(11, "小微
微", "123"));
    LOGGER.debug("Affected rows: [{}]",rows);
    sqlSession.commit();
    sqlSession.close();
}
```


5、delete（删除）

> delete标签用于做删除操作

1、写接口

```
/**
* 删除一个用户
* @param id
* @return
*/
int deleteUser(int id);
```

2、写SQL

```
<delete id="deleteUser" parameterType="int">
  delete from user where id = #{id}
</delete>
```

3、测试

```
 @Test
public void testDeleteUser(){
    SqlSession sqlSession = open();
    UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
    int affectedRows = mapper.deleteUser(5);
    LOGGER.debug("Affected rows:
[{}]",affectedRows);
    sqlSession.commit();
    sqlSession.close();
}
```

6、模糊查询

> 方案一：在Java代码中拼串

```
string name = “%IT%”;
list<name> names = mapper.getUserByName(name);
```

```
<select id=”getUsersByName”>
    select * from user where name like #{name}
</select>
```

> 方案二：在配置文件中拼接

```
string name = “IT”;
list<User> users = mapper.getUserByName(name);
```

```
<select id=”getUsersByName”>
    select * from user where name like "%"#{name}"%"
</select>
```

为什么必须用双引号?

> 方案三：在配置文件中拼接

```
<select id=”getUsersByName”>
    select * from user where name like "%${name}%"
</select>
```

7、map的使用

> map可以代替任何的实体类，所以当我们数据比较复杂时，可以适当考虑使用map来完成相关工作

1、写sql

```
<select id="getUsersByParams"
parameterType="java.util.HashMapmap">
    select id,username,password from user where
username = #{name}
</select>
```

2、写方法

```
/**
* 根据一些参数查询
* @param map
* @return
*/
List<User> getUsersByParams(Map<String,String> map);
```

3、测试

```
@Test
public void findByParams() {
    UserMapper mapper =
session.getMapper(UserMapper.class);
    Map<String,String> map = new HashMap<String,
String>();
    map.put("name","磊磊哥");
    List<User> users = mapper.getUsersByParams(map);
    for (User user: users){
        System.out.println(user.getUsername());
    }
}
```

五、使用注解开发

```
DROP TABLE IF EXISTS `admin`;
CREATE TABLE `admin` (
`id` int(20) NOT NULL,
`username` varchar(30) DEFAULT NULL,
`password` varchar(30) DEFAULT NULL,
PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert  into `admin`(`id`,`username`,`password`)
values (1,'itnanls','123456'),(2,'itlils','abcdef'),
(3,'小微','987654');
```

MyBatis最初配置信息是基于 XML ,映射语句(SQL)也是定义在 XML 中的。而到MyBatis 3提供了新的基于注解的配置。不幸的是，Java注解的的表达力和灵活性十分有限。最强大的 MyBatis 映射并不能用注解来构建，所以这里我们作为了解。

- sql 类型主要分成 :
- @select ()
- @update ()
- @Insert ()
- @delete ()

注意：利用注解开发就不需要mapper.xml映射文件了 .

1、接口中添加注解

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

    private static final Long serialVersionUID = 1L;

    private int id;
    private String username;
    private String password;
}
```



```
public interface AdminMapper {

    /**
     * 保存管理员
     * @param admin
     * @return
     */
    @Insert("insert into admin (username,password)
values (#{username},#{password})")
    int saveAdmin(Admin admin);

    /**
```

```
     * 跟新管理员
     * @param admin
     * @return
     */
    @Update("update admin set username=#{username} ,
password=#{password} where id = #{id}")
    int updateAdmin(Admin admin);

    /**
     * 删除管理员
     * @param admin
     * @return
     */
    @Delete("delete from admin where id=#{id}")
    int deleteAdmin(int id);

    /**
     * 根据id查找管理员
     * @param id
     * @return
     */
    @Select("select id,username,password from admin
where id=#{id}")
    Admin findAdminById(@Param("id") int id);

    /**
     * 查询所有的管理员
     * @return
     */
    @Select("select id,username,password from
admin")
    List<Admin> findAllAdmins();

}
```


2、核心配置文件中配置

添加一个mapper的配置：

```
<mapper class="com.ydlclass.mapper.AdminMapper"/>
```

3、进行测试

```
public class TestAdminMapper {

    @Test
    public void testSaveAdmin() {
        SqlSession session = open();
        AdminMapper mapper =
session.getMapper(AdminMapper.class);
        Admin admin = new Admin(1,"微微
姐","12345678");
        int affectedRows = mapper.saveAdmin(admin);
         LOGGER.debug("Affected rows:
[{}]",affectedRows);
        session.commit();
        session.close();
    }

    @Test
    public void testUpdateAdmin() {
        SqlSession session = open();
        AdminMapper mapper =
session.getMapper(AdminMapper.class);
        Admin user = new Admin(1,"磊磊
哥","12345678");
        int affectedRows = mapper.updateAdmin(user);
        LOGGER.debug("Affected rows:
[{}]",affectedRows);
        session.commit();
        session.close();
    }
```

```
    @Test
    public void testDeleteAdmin(){
        SqlSession session = open();
        AdminMapper mapper =
session.getMapper(AdminMapper.class);
        int affectedRows = mapper.deleteAdmin(2);
        LOGGER.debug("Affected rows:
[{}]",affectedRows);
        session.commit();
        session.close();
    }

    @Test
    public void testGetAdminById(){
        SqlSession session = open();
        AdminMapper mapper =
session.getMapper(AdminMapper.class);
        Admin admin = mapper.findAdminById(1);
        LOGGER.debug("The admin is: [{}]",admin);
        session.commit();
        session.close();
    }

    @Test
    public void testGetAllAdmins(){
        SqlSession session = open();
        AdminMapper mapper =
session.getMapper(AdminMapper.class);
        List<Admin> admins = mapper.findAllAdmins();
        LOGGER.debug("The admins is: [{}]",admin);
        session.commit();
        session.close();
    }

    private static SqlSession open(){
        // 1、创建一个SqlSessionFactory的 建造者 ，用于
创建SqlSessionFactory
```

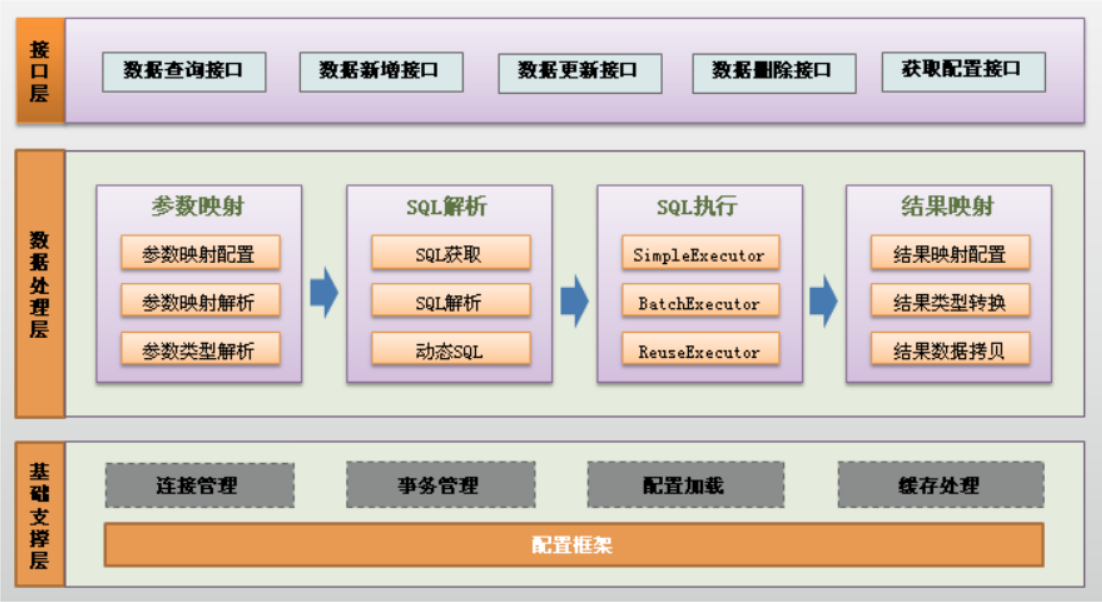
```
        SqlSessionFactoryBuilder builder = new
SqlSessionFactoryBuilder();
        // 2、使用builder构建一个sqlSessionFactory，此
处我们基于一个xml配置文件
        SqlSessionFactory sqlSessionFactory =
builder.build(Thread.currentThread().getContextClass
Loader().getResourceAsStream("mybatis-config.xml"));
        // 3、通过sqlSessionFactory获取另一个session，
此处使用【工厂设计模式】
        return sqlSessionFactory.openSession();
    }

}
```


六、架构源码解析（选学）

1、架构讲解

Mybatis的功能架构分为三层：



API接口层：提供给外部使用的接口API，开发人员通过这些本地API来操纵数据库。接口层一接收到调用请求就会调用数据处理层来完成具体的数据处理。

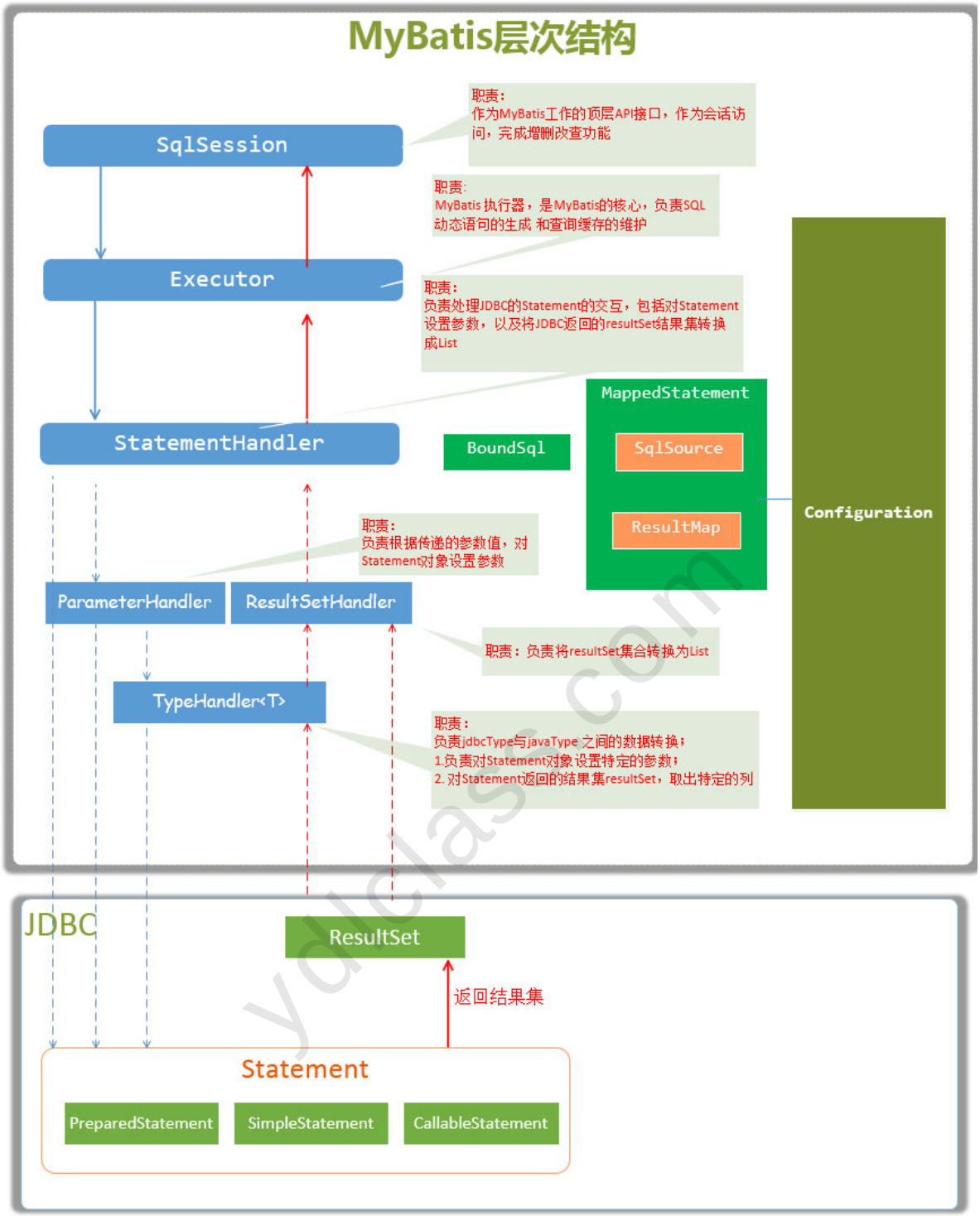
数据处理层：负责具体的SQL查找、SQL解析、SQL执行和执行结果映射处理等。它主要的目的是根据调用的请求完成一次数据库操作。

基础支撑层：负责最基础的功能支撑，包括连接管理、事务管理、配置加载和缓存处理，这些都是共用的东西，将他们抽取出来作为最基础的组件。为上层的数据处理层提供最基础的支撑。

2、核心成员：

1. **Configuration**：MyBatis所有的配置信息都保存在Configuration对象之中，配置文件中的大部分配置都会存储到该类中
2. **SqlSession**：作为MyBatis工作的主要顶层API，表示和数据库交互时的会话，完成必要数据库增删改查功能
3. **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
4. **StatementHandler**：封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数等
5. **ParameterHandler**：负责对用户传递的参数转换成JDBC Statement 所对应的数据类型
6. **ResultSetHandler**：负责将JDBC返回的ResultSet结果集对象转换成List类型的集合
7. **TypeHandler**：负责java数据类型和jdbc数据类型(也可以说是数据表列类型)之间的映射和转换
8. **MappedStatement**：MappedStatement维护一条<select|update|delete|insert>节点的封装
9. **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
10. **BoundSql**：表示动态生成的SQL语句以及相应的参数信息

11.



3、源码解读

（1）构建session工厂

```
// 1、创建一个SqlSessionFactory的 建造者 ，用于创建SqlSessionFactory
// SqlSessionFactoryBuilder中有大量的重载的build方法，可以根据不同的入参，进行构建
// 极大的提高了灵活性，此处使用【创建者设计模式】
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
```

这是一个创建者设计模式的经典应用：

```
// 使用builder构建一个sqlSessionFactory，此处我们基于一个xml配置文件
// 此过程会进行xml文件的解析，过程相对比较复杂
SqlSessionFactory sqlSessionFactory = builder.build(Thread.currentThread().getContextClassLoader().getResourceAsStream("mybatis-config.xml"));
```

> 源码部分：这里有众多的重载build方法，我们调用的build方法，会是如下大流程

```
public SqlSessionFactory build(InputStream inputStream) {
    return build(inputStream, null, null);
}
```

```
public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {
    try {
        XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);
        return build(parser.parse());
    } catch (Exception e) {
    ....
}
```

从上边的return看，其实核心的代码又回归到了`build(parser.parse())`这个构造器。

```
public SqlSessionFactory build(Configuration config)
{
    return new DefaultSqlSessionFactory(config);
}
```

其实，本质上，无论你做了多少工作，你使用xml也好，不使用xml也好，最终都是需要一个Configuration实例，这里保存了所有的配置项。

当然我们可以独立去使用Configuration类构造实例，不使用xml。

例如：

```
Configuration configuration = new Configuration();
// 创建一个数据源
PooledDataSource pooledDataSource = new
PooledDataSource();
pooledDataSource.setDriver("com.mysql.cj.jdbc.Driver
");
pooledDataSource.setUrl("jdbc:mysql://127.0.0.1:3306
/ydlclass?
characterEncoding=utf8&amp;serverTimezone=Asia/Shang
hai");
pooledDataSource.setUsername("root");
pooledDataSource.setPassword("root");
Environment environment = new Environment("env",new
JdbcTransactionFactory(),new PooledDataSource());
configuration.setEnvironment(environment);
```

等同于：

```
<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://127.0.0.1:3306/ydlclass?
characterEncoding=utf8&amp;serverTimezone=Asia/Shang
hai"/>
                <property name="username"
value="root"/>
                <property name="password"
value="root"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```


xml的解析过程就是将xml文件转化为Configuration对象，它在启动的时候执行，也就意味着修改配置文件就要重启。所以本环节的重点就到了。

（2）配置文件的解析

`parser.parse()`这个方法了，这就是在解析xml配置文件。

在build方法中我们看到了如下代码：

```
XMLConfigBuilder parser = new
XMLConfigBuilder(inputStream, environment,
properties);
```

这一步就是构造一个解析器，根据我们的入参构建一个文档解析器。使用了sax进行xml的解析，我们在讲JavaEE的时候讲过。

当然，我们要把重点放在parse()方法上：

```
public Configuration parse() {
    ...省略不重要的代码
    //此处就是解析的核心代码

 parseConfiguration(parser.evalNode("/configuration"
));
    return configuration;
}
```

咱们进入这个方法，慢慢分析，其中的内容很多，很明显看到这个方法就是在解析每一个标签。

```
private void parseConfiguration(XNode root) {
    try {
        // 处理properties标签

 propertiesElement(root.evalNode("properties"));
        Properties settings =
settingsAsProperties(root.evalNode("settings"));
        loadCustomVfs(settings);
        loadCustomLogImpl(settings);
        // 处理别名的标签

 typeAliasesElement(root.evalNode("typeAliases"));
        pluginElement(root.evalNode("plugins"));

 objectFactoryElement(root.evalNode("objectFactory")
);

 objectWrapperFactoryElement(root.evalNode("objectWr
apperFactory"));
```

```
 reflectorFactoryElement(root.evalNode("reflectorFac
tory"));
        settingsElement(settings);
        // read it after objectFactory and
objectWrapperFactory issue #631
        // 处理environments标签

 environmentsElement(root.evalNode("environments"));

 databaseIdProviderElement(root.evalNode("databaseId
Provider"));

 typeHandlerElement(root.evalNode("typeHandlers"));
        // 处理mappers标签
        mapperElement(root.evalNode("mappers"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing
SQL Mapper Configuration. Cause: " + e, e);
    }
}
```

其实我们看到这里就大致明白了MyBatis解析xml的时机和方法了。从这里我们也能基本看出来一个配置文件内能使用的标签，以及书写标签的顺序，因为这个解析过程也是有顺序的，我们随便列出几个标签看看配置文件长什么样子：

```
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-
config.dtd">
<configuration>
    <properties>
        <property name="username" value="root"/>
    </properties>

    <settings>
```

```
        <setting name="" value=""/>
    </settings>
    <typeAliases>
        <typeAlias type="com.ydlclass.User"
alias="user"/>
    </typeAliases>

    <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://127.0.0.1:3306/ydlclass?
characterEncoding=utf8&amp;serverTimezone=Asia/Shang
hai"/>
                <property name="username"
value="root"/>
                <property name="password"
value="root"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="userMapper.xml"/>
    </mappers>
</configuration>
```


（3）mapper文件的解析过程

我们暂且忽略掉其他标签的处理，以`mappers`标签为例继续深入探索：

```
private void mapperElement(XNode parent) throws
Exception {
    if (parent != null) {
```

```
        // 循环遍历他的孩子节点
        for (XNode child : parent.getChildren()) {
            if ("package".equals(child.getName())) {
                // <package name="com.ydlclass"/>
                String mapperPackage =
child.getStringAttribute("name");
                // 直接将包名加入配置项

 configuration.addMappers(mapperPackage);
            } else {

                String resource =
child.getStringAttribute("resource");
                String url =
child.getStringAttribute("url");
                String mapperClass =
child.getStringAttribute("class");
                // 如果是resource属性,就通过resource获
取资源并解析<mapper resource="userMapper.xml"/>
                if (resource != null && url == null
&& mapperClass == null) {

 ErrorContext.instance().resource(resource);
                    InputStream inputStream =
Resources.getResourceAsStream(resource);
                    XMLMapperBuilder mapperParser =
new XMLMapperBuilder(inputStream, configuration,
resource, configuration.getSqlFragments());
                    mapperParser.parse();
                // 如果是url属性,就通过url获取资源并解析
<mapper
url="http://www.ydlclass.com/UserMapper.xml"/>
                } else if (resource == null && url
!= null && mapperClass == null) {

 ErrorContext.instance().resource(url);
```

```
                    InputStream inputStream =
Resources.getUrlAsStream(url);
                    XMLMapperBuilder mapperParser =
new XMLMapperBuilder(inputStream, configuration,
url, configuration.getSqlFragments());
                    mapperParser.parse();
                // 如果是url属性,就通过url获取资源并解析
<mapper class="com.ydlclass.UserMapper"/>
                } else if (resource == null && url
== null && mapperClass != null) {
                    Class<?> mapperInterface =
Resources.classForName(mapperClass);
                    // 注册一个mapper

 configuration.addMapper(mapperInterface);
                } else {
                    throw new BuilderException("A
mapper element may only specify a url, resource or
class, but not more than one.");
                }
            }
        }
    }
}
```


这里就是想尽办法，获取mapper的配置信息，然后做下一步处理，其实无论是哪一种方式都是会在Configuration的mapperRegistry （mapper注册器中）中注册一个mapper。其实就是将mapper的class信息放在一个名为knownMappers的hashmap中，以便后续使用。当然他的值一个一个代理工厂，这玩意能帮我们获取一个mapper的代理对象，更详细的后续说。

```
private final Map<Class<?>, MapperProxyFactory<?>>
knownMappers = new HashMap<>();
```

> 对于package属性

直接将包名注册到配置中，然后调用MapperRegistry的addMappers方法，通过扫描文件的方式将这个包底下的class添加进kownMappers中。

```
 public void addMappers(String packageName) {
    mapperRegistry.addMappers(packageName);
  }
```

```
public void addMappers(String packageName) {
    addMappers(packageName, Object.class);
}
```

```
public void addMappers(String packageName, Class<?>
superType) {
    ResolverUtil<Class<?>> resolverUtil = new
ResolverUtil<>();
    resolverUtil.find(new
ResolverUtil.IsA(superType), packageName);
    Set<Class<? extends Class<?>>> mapperSet =
resolverUtil.getClasses();
    for (Class<?> mapperClass : mapperSet) {
        addMapper(mapperClass);
    }
}
```

```
public <T> void addMapper(Class<T> type) {
    mapperRegistry.addMapper(type);
  }
```

我们可以从MapperRegistry中看到，addMapper的整个过程。

```
 public <T> void addMapper(Class<T> type) {
    if (type.isInterface()) {
      if (hasMapper(type)) {
        throw new BindingException("Type " + type +
" is already known to the MapperRegistry.");
      }
      boolean loadCompleted = false;
      try {
        knownMappers.put(type, new
MapperProxyFactory<>(type));
        ...省略不重要的代码
    }
  }
```

我们不妨把MapperProxyFactory的代码粘贴出来看看，里边维护了一个接口和方法缓存（这个后边会讲，目前他是一个空的Map），这个代理工厂确实有方法帮我们生成代理对象（newInstance）。我们看到了代理设计模式。

```
public class MapperProxyFactory<T> {

  private final Class<T> mapperInterface;
  private final Map<Method, MapperMethodInvoker>
methodCache = new ConcurrentHashMap<>();

  public MapperProxyFactory(Class<T>
mapperInterface) {
    this.mapperInterface = mapperInterface;
  }

  public Class<T> getMapperInterface() {
    return mapperInterface;
  }

  public Map<Method, MapperMethodInvoker>
getMethodCache() {
```

```
    return methodCache;
  }

  @SuppressWarnings("unchecked")
  protected T newInstance(MapperProxy<T>
mapperProxy) {
    return (T)
Proxy.newProxyInstance(mapperInterface.getClassLoade
r(), new Class[] { mapperInterface }, mapperProxy);
  }

  public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new
MapperProxy<>(sqlSession, mapperInterface,
methodCache);
    return newInstance(mapperProxy);
  }

}
```

> 对与class属性的处理

这个属性我们配置的是一个class，`<mapper class="com.ydlclass.UserMapper"/>`，对一class将来肯定要从注解中解析信息，直接把他的class信息注册进去就好了。

> resource属性以及url属性的处理

在resource属性以及url属性中没有看到configuration.addMapper()这个方法的影子，这两个属性都是以配置文件的方式加载，自然要解析mapper配置文件了。

我们看到了`XMLMapperBuilder`这个类的parse()方法。很明显这个方法configurationElement是用来解析配置文件的。

```
public void parse() {
    if (!configuration.isResourceLoaded(resource)) {

 configurationElement(parser.evalNode("/mapper"));
        configuration.addLoadedResource(resource);
        bindMapperForNamespace();
    }

    parsePendingResultMaps();
    parsePendingCacheRefs();
    parsePendingStatements();
}
```

而在这个方法中bindMapperForNamespace我们看到了注册mapper的代码：

```
private void bindMapperForNamespace() {
        ...其他代码省略

 configuration.addLoadedResource("namespace:" +
namespace);
            configuration.addMapper(boundType);
        }
    }
}
```

> mapper的具体解析

创建一个解析器：

```
XMLMapperBuilder mapperParser = new
XMLMapperBuilder(inputStream, configuration,
resource, configuration.getSqlFragments());
```

这个方法就是对每一个标签的解析：

```
private void configurationElement(XNode context) {
    try {
        String namespace =
context.getStringAttribute("namespace");
        if (namespace == null ||
namespace.isEmpty()) {
            throw new BuilderException("Mapper's
namespace cannot be empty");
        }

 builderAssistant.setCurrentNamespace(namespace);
        cacheRefElement(context.evalNode("cache-
ref"));
        cacheElement(context.evalNode("cache"));

 parameterMapElement(context.evalNodes("/mapper/para
meterMap"));

 resultMapElements(context.evalNodes("/mapper/result
Map"));

 sqlElement(context.evalNodes("/mapper/sql"));

 buildStatementFromContext(context.evalNodes("select
|insert|update|delete"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing
Mapper XML. The XML location is '" + resource + "'.
Cause: " + e, e);
    }
}
```


(4) 通过sqlSession获取一个代理对象

```
通过sqlSessionFactory获取另一个session，此处使用【工厂设计模式】
SqlSession sqlSession =
sqlSessionFactory.openSession();
```

> 接下来就要研究mapper的代理对象生成的过程了，此处使用【代理设计模式】

```
// 4、通过sqlSession获取一个代理对象，此处使用【代理设计模式】
UserMapper mapper =
sqlSession.getMapper(UserMapper.class);
```

我们可以走到DefaultSqlSession，这是SqlSession的一个子类，从open方法中我们能看到默认创建的sqlSession是他的子类DefaultSqlSession

```
private SqlSession
openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit)
{
    ...省略不重要的代码
    return new DefaultSqlSession(configuration,
executor, autoCommit);
}
```

从实现类的getMapper中得知：

```
public <T> T getMapper(Class<T> type) {
    return configuration.getMapper(type, this);
}
```

```
public <T> T getMapper(Class<T> type, SqlSession
sqlSession) {
    return mapperRegistry.getMapper(type,
sqlSession);
  }
```

从注册器中获取mapper工厂，并创建代理对象：

```
public <T> T getMapper(Class<T> type, SqlSession
sqlSession) {
    final MapperProxyFactory<T> mapperProxyFactory =
(MapperProxyFactory<T>) knownMappers.get(type);
    if (mapperProxyFactory == null) {
        throw new BindingException("Type " + type +
" is not known to the MapperRegistry.");
    }
    try {
        // 创建代理对象：
        return
mapperProxyFactory.newInstance(sqlSession);
    } catch (Exception e) {
        throw new BindingException("Error getting
mapper instance. Cause: " + e, e);
    }
}
```

那我们可以聊一聊MapperProxyFactory这个类了，它持有一个接口和一个methodCache，这个接口当然是为了生成代理对象使用的类。

```
public class MapperProxyFactory<T> {
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethodInvoker>
methodCache = new ConcurrentHashMap<>();
}
```

在创建对象的时候创建了一个MapperProxy：

```
public T newInstance(SqlSession sqlSession) {
    final MapperProxy<T> mapperProxy = new
MapperProxy<>(sqlSession, mapperInterface,
methodCache);
    return newInstance(mapperProxy);
}
```

MapperProxy就是我们的InvocationHandler，也是创建代理对象必须的，其中的核心方法是invoke，会在调用代理对象的方法时调用：

```
public class MapperProxy<T> implements
InvocationHandler, Serializable {

    private static final long serialVersionUID =
-4724728412955527868L;
    private static final int ALLOWED_MODES =
MethodHandles.Lookup.PRIVATE |
MethodHandles.Lookup.PROTECTED
        | MethodHandles.Lookup.PACKAGE |
MethodHandles.Lookup.PUBLIC;
    private static final Constructor<Lookup>
lookupConstructor;
    private static final Method
privateLookupInMethod;
    private final SqlSession sqlSession;
    private final Class<T> mapperInterface;
    private final Map<Method, MapperMethodInvoker>
methodCache;

    public MapperProxy(SqlSession sqlSession,
Class<T> mapperInterface, Map<Method,
MapperMethodInvoker> methodCache) {
```

```
        this.sqlSession = sqlSession;
        this.mapperInterface = mapperInterface;
        this.methodCache = methodCache;
    }

    @Override
    public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
        try {
            if
(Object.class.equals(method.getDeclaringClass())) {
                return method.invoke(this, args);
            } else {
                return
cachedInvoker(method).invoke(proxy, method, args,
sqlSession);
            }
        } catch (Throwable t) {
            throw ExceptionUtil.unwrapThrowable(t);
        }
    }
```

核心方法当然是invoke了，将来调用代理对象的方法，其实就是在执行此方法。

（5）方法调用

当我们执行代理的对象的方法时：

```
List<User> allUser = mapper.findAllUser(12);
```

invoke方法会被调用，这里会判断它调用的是继承自Object的方法还是实现的接口的方法，我们的重点放在：

```
cachedInvoker(method).invoke(proxy, method, args,
sqlSession);
```

```
private MapperMethodInvoker cachedInvoker(Method
method) throws Throwable {
    try {
      // 这里就知道了methodCache的左右了，方法存在缓存里避
免频繁的创建
      MapperMethodInvoker invoker =
methodCache.get(method);
      if (invoker != null) {
        return invoker;
      }

        // 缓存没有就创建一个
      return methodCache.computeIfAbsent(method, m -
> {
          // 这里是处理接口的默认方法
        if (m.isDefault()) {
          try {
            if (privateLookupInMethod == null) {
              return new
DefaultMethodInvoker(getMethodHandleJava8(method));
            } else {
              return new
DefaultMethodInvoker(getMethodHandleJava9(method));
            }
          } catch (IllegalAccessException |
InstantiationException | InvocationTargetException
              | NoSuchMethodException e) {
            throw new RuntimeException(e);
          }
            // 核心代码在这里
        } else {
```

```
          return new PlainMethodInvoker(new
MapperMethod(mapperInterface, method,
sqlSession.getConfiguration()));
        }
      });
    } catch (RuntimeException re) {
      Throwable cause = re.getCause();
      throw cause == null ? re : cause;
    }
  }
```

核心是（普通的方法调用者）：

```
 new PlainMethodInvoker(new
MapperMethod(mapperInterface, method,
sqlSession.getConfiguration()));
```

在PlainMethodInvoker传入了一个MapperMethod（方法的包装类），根据接口、方法签名和我们的配置生成一个方法的包装，这里有SqlCommand（用来执行sql），还有我们的方法签名。

```
public class MapperMethod {

  private final SqlCommand command;
  private final MethodSignature method;

  public MapperMethod(Class<?> mapperInterface,
Method method, Configuration config) {
    this.command = new SqlCommand(config,
mapperInterface, method);
    this.method = new MethodSignature(config,
mapperInterface, method);
  }
}
```

它还有个核心方法，就是执行具体的sql啦！

```
public Object execute(SqlSession sqlSession,
Object[] args) {
    Object result;
    switch (command.getType()) {
      case INSERT: {
        Object param =
method.convertArgsToSqlCommandParam(args);
        result =
rowCountResult(sqlSession.insert(command.getName(),
param));
        break;
      }
      case UPDATE: {
        Object param =
method.convertArgsToSqlCommandParam(args);
        result =
rowCountResult(sqlSession.update(command.getName(),
param));
        break;
      }
      case DELETE: {
        Object param =
method.convertArgsToSqlCommandParam(args);
        result =
rowCountResult(sqlSession.delete(command.getName(),
param));
        break;
      }
      case SELECT:
        if (method.returnsVoid() &&
method.hasResultHandler()) {
          executeWithResultHandler(sqlSession,
args);
          result = null;
        } else if (method.returnsMany()) {
```

```
            result = executeForMany(sqlSession, args);
        // 一下部分省略
        return result;
    }
```

回头看：

```
 cachedInvoker(method).invoke(proxy, method, args,
sqlSession);
```

本质调用的就是PlainMethodInvoker这个子类的invoke方法，而它确实调用mapperMethod.execute方法。到此就明白了整个流程

```
private static class PlainMethodInvoker implements
MapperMethodInvoker {
  private final MapperMethod mapperMethod;

  public PlainMethodInvoker(MapperMethod
mapperMethod) {
    super();
    this.mapperMethod = mapperMethod;
  }

  @Override
  public Object invoke(Object proxy, Method method,
Object[] args, SqlSession sqlSession) throws
Throwable {
    return mapperMethod.execute(sqlSession, args);
  }
}
```


(6) 继续了解

> 一级缓存

我们不妨在excute方法中随机找一个select语句深入挖掘：

```
result = executeForMany(sqlSession,
args);executeForMany(sqlSession, args);
```

看方法：

```
private <E> Object executeForMany(SqlSession
sqlSession, Object[] args) {
    List<E> result;
    ...省略其他代码
        result =
sqlSession.selectList(command.getName(), param);

    return result;
}
```

这里就能看到本质上使用的是`sqlSession.selectList(command.getName(), param);`这距离我们熟悉的越来越近了：

```
@Override
public <E> List<E> selectList(String statement,
Object parameter, RowBounds rowBounds) {
    try {
        // MappedStatement，使用执行器执行sql
        MappedStatement ms =
configuration.getMappedStatement(statement);
        return executor.query(ms,
wrapCollection(parameter), rowBounds,
Executor.NO_RESULT_HANDLER);
    } catch (Exception e) {
        throw ExceptionFactory.wrapException("Error
querying database.  Cause: " + e, e);
    } finally {
        ErrorContext.instance().reset();
    }
}
```

MappedStatement其实就是我们mapper.xml的解析结果，放了很多的信息：

```
public final class MappedStatement {

  private String resource;
  private Configuration configuration;
  private String id;
  private Integer fetchSize;
  private Integer timeout;
  private StatementType statementType;
  private ResultSetType resultSetType;
  private SqlSource sqlSource;
  private Cache cache;
  private ParameterMap parameterMap;
  private List<ResultMap> resultMaps;
  private boolean flushCacheRequired;
  private boolean useCache;
  private boolean resultOrdered;
```

```
  private SqlCommandType sqlCommandType;
  ...省略
```

executor是真正执行sql的一个执行器，它是一个接口，有具体的实现比如抽象类BaseExecutor，实现SimpleExecutor：

```
public interface Executor {


    int update(MappedStatement ms, Object parameter)
throws SQLException;

    <E> List<E> query(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler
resultHandler, CacheKey cacheKey, BoundSql boundSql)
throws SQLException;

    <E> List<E> query(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler
resultHandler) throws SQLException;

    <E> Cursor<E> queryCursor(MappedStatement ms,
Object parameter, RowBounds rowBounds) throws
SQLException;

    List<BatchResult> flushStatements() throws
SQLException;

    void commit(boolean required) throws
SQLException;

    void rollback(boolean required) throws
SQLException;

    ...省略

}
```

我们继续看 executor.query，在抽象类BaseExecutor它是这么实现的：

```
public abstract class BaseExecutor implements
Executor
```

```
@Override
public <E> List<E> query(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler
resultHandler) throws SQLException {
    BoundSql boundSql = ms.getBoundSql(parameter);
    // 请注意这里创建了key
    CacheKey key = createCacheKey(ms, parameter,
rowBounds, boundSql);
    return query(ms, parameter, rowBounds,
resultHandler, key, boundSql);
}
```

这里有一个核心方法queryFromDatabase

```
  @Override
  public <E> List<E> query(MappedStatement ms,
Object parameter, RowBounds rowBounds, ResultHandler
resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {

 ErrorContext.instance().resource(ms.getResource()).
activity("executing a query").object(ms.getId());
    if (closed) {
      throw new ExecutorException("Executor was
closed.");
    }
    if (queryStack == 0 &&
ms.isFlushCacheRequired()) {
      clearLocalCache();
```

```
    }
    List<E> list;
    try {
      queryStack++;
       // 这里有【一级缓存】的影子，优先去缓存中取
      list = resultHandler == null ? (List<E>)
localCache.getObject(key) : null;
      if (list != null) {
        handleLocallyCachedOutputParameters(ms, key,
parameter, boundSql);
      } else {
        list = queryFromDatabase(ms, parameter,
rowBounds, resultHandler, key, boundSql);
      }
    } finally {
      queryStack--;
    }
    if (queryStack == 0) {
      for (DeferredLoad deferredLoad :
deferredLoads) {
        deferredLoad.load();
      }
      // issue #601
      deferredLoads.clear();
      if (configuration.getLocalCacheScope() ==
LocalCacheScope.STATEMENT) {
        // issue #482
        clearLocalCache();
      }
    }
    return list;
  }
```

从这里我们能看到【以及缓存的影子了】localCache，通过doQuery查询的结果，会放在localCache中。

```
private <E> List<E>
queryFromDatabase(MappedStatement ms, Object
parameter, RowBounds rowBounds, ResultHandler
resultHandler, CacheKey key, BoundSql boundSql)
throws SQLException {
    List<E> list;
    localCache.putObject(key,
EXECUTION_PLACEHOLDER);
    try {
        // 核心方法
        list = doQuery(ms, parameter, rowBounds,
resultHandler, boundSql);
    } finally {
        localCache.removeObject(key);
    }
    localCache.putObject(key, list);
    if (ms.getStatementType() ==
StatementType.CALLABLE) {
        localOutputParameterCache.putObject(key,
parameter);
    }
    return list;
}
```

> 原生的jdbc

我们继续看实现：

```
public class SimpleExecutor extends BaseExecutor
```

我们进入上边提及的doQuery，这里看到一下原生jdbc的影子了，比如stmt；

```
@Override
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);
        // 获取prepareStatement
        stmt = prepareStatement(handler,
ms.getStatementLog());
        return handler.query(stmt, resultHandler);
    } finally {
        closeStatement(stmt);
    }
}
```

这里有几个方法：

获取prepareStatement，这里看到了Connection：

```
private Statement prepareStatement(StatementHandler
handler, Log statementLog) throws SQLException {
    Statement stmt;
    Connection connection =
getConnection(statementLog);
    // 创建PreparedStatement
    stmt = handler.prepare(connection,
transaction.getTimeout());
    // 填充占位符
    handler.parameterize(stmt);
    return stmt;
}
```

继续深入：

```
public abstract class BaseStatementHandler
implements StatementHandler
```

```
@Override
public Statement prepare(Connection connection,
Integer transactionTimeout) throws SQLException {
    ErrorContext.instance().sql(boundSql.getSql());
    Statement statement = null;
    try {
        // 创建PreparedStatement
        statement =
instantiateStatement(connection);
        setStatementTimeout(statement,
transactionTimeout);
        setFetchSize(statement);
        return statement;
    } catch (SQLException e) {
        closeStatement(statement);
        throw e;
    } catch (Exception e) {
        closeStatement(statement);
        throw new ExecutorException("Error preparing
statement.  Cause: " + e, e);
    }
}
```


继续深入，我们看到了熟悉的connection.prepareStatement(sql)：

```
public class PreparedStatementHandler extends
BaseStatementHandler
```

```
@Override
protected Statement instantiateStatement(Connection
connection) throws SQLException {
    String sql = boundSql.getSql();
    ... 省略
} else if (mappedStatement.getResultSetType() ==
ResultSetType.DEFAULT) {
    return connection.prepareStatement(sql);
} else {
    return connection.prepareStatement(sql,
mappedStatement.getResultSetType().getValue(),
ResultSet.CONCUR_READ_ONLY);
}
}
```

这里就是对占位符进行替换

```
@Override
public void parameterize(Statement statement) throws
SQLException {

 parameterHandler.setParameters((PreparedStatement)
statement);
}
```

```
 @Override
  public void setParameters(PreparedStatement ps) {
    ErrorContext.instance().activity("setting
parameters").object(mappedStatement.getParameterMap(
).getId());
    List<ParameterMapping> parameterMappings =
boundSql.getParameterMappings();
    if (parameterMappings != null) {
      for (int i = 0; i < parameterMappings.size();
i++) {
```

```
        ParameterMapping parameterMapping =
parameterMappings.get(i);
        if (parameterMapping.getMode() !=
ParameterMode.OUT) {
          Object value;
          String propertyName =
parameterMapping.getProperty();
          if
(boundSql.hasAdditionalParameter(propertyName)) { //
issue #448 ask first for additional params
            value =
boundSql.getAdditionalParameter(propertyName);
          } else if (parameterObject == null) {
            value = null;
          } else if
(typeHandlerRegistry.hasTypeHandler(parameterObject.
getClass())) {
            value = parameterObject;
          } else {
            MetaObject metaObject =
configuration.newMetaObject(parameterObject);
            value =
metaObject.getValue(propertyName);
          }
          TypeHandler typeHandler =
parameterMapping.getTypeHandler();
          JdbcType jdbcType =
parameterMapping.getJdbcType();
          if (value == null && jdbcType == null) {
            jdbcType =
configuration.getJdbcTypeForNull();
          }
          try {
            typeHandler.setParameter(ps, i + 1,
value, jdbcType);
          } catch (TypeException | SQLException e) {
```

```
            throw new TypeException("Could not set
parameters for mapping: " + parameterMapping + ".
Cause: " + e, e);
          }
        }
      }
    }
  }
```

当前位置我们已经从头对MyBatis的源码撸了一遍，有帮助三连。

4、别名系统

> 思考：

```
<select id="getUsersByParams"
parameterType="java.util.HashMap">
    select id,username,password from user where
username = #{name}
</select>
```

resultType写成java.util.HashMap也行，写成map也行，说明MyBatis内置很多别名，包括我们的配置中：

```
<environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username"
value="${username}"/>
        <property name="password"
value="${password}"/>
    </dataSource>
</environment>
```

其中为什么写一个POOLED就能代表我们使用了什么数据库连接池，其实这也是别名，简单理解就是可以用一个简单的短小的名字替代很长的类的权限定名称。

其实：在构造Configuration类时，注册了如下的别名。

```
public Configuration() {
    typeAliasRegistry.registerAlias("JDBC",
JdbcTransactionFactory.class);
    typeAliasRegistry.registerAlias("MANAGED",
ManagedTransactionFactory.class);

    typeAliasRegistry.registerAlias("JNDI",
JndiDataSourceFactory.class);
    typeAliasRegistry.registerAlias("POOLED",
PooledDataSourceFactory.class);
    typeAliasRegistry.registerAlias("UNPOOLED",
UnpooledDataSourceFactory.class);

    typeAliasRegistry.registerAlias("PERPETUAL",
PerpetualCache.class);
    typeAliasRegistry.registerAlias("FIFO",
FifoCache.class);
    typeAliasRegistry.registerAlias("LRU",
LruCache.class);
```

```
    typeAliasRegistry.registerAlias("SOFT",
SoftCache.class);
    typeAliasRegistry.registerAlias("WEAK",
WeakCache.class);

    typeAliasRegistry.registerAlias("DB_VENDOR",
VendorDatabaseIdProvider.class);

    typeAliasRegistry.registerAlias("XML",
XMLLanguageDriver.class);
    typeAliasRegistry.registerAlias("RAW",
RawLanguageDriver.class);

    typeAliasRegistry.registerAlias("SLF4J",
Slf4jImpl.class);

 typeAliasRegistry.registerAlias("COMMONS_LOGGING",
JakartaCommonsLoggingImpl.class);
    typeAliasRegistry.registerAlias("LOG4J",
Log4jImpl.class);
    typeAliasRegistry.registerAlias("LOG4J2",
Log4j2Impl.class);
    typeAliasRegistry.registerAlias("JDK_LOGGING",
Jdk14LoggingImpl.class);

 typeAliasRegistry.registerAlias("STDOUT_LOGGING",
StdOutImpl.class);
    typeAliasRegistry.registerAlias("NO_LOGGING",
NoLoggingImpl.class);

    typeAliasRegistry.registerAlias("CGLIB",
CglibProxyFactory.class);
    typeAliasRegistry.registerAlias("JAVASSIST",
JavassistProxyFactory.class);
```

```
 languageRegistry.setDefaultDriverClass(XMLLanguageDriver.class);

 languageRegistry.register(RawLanguageDriver.class);
  }
```

在构造TypeAliasRegistry注册了如下的别名，我们可以在配置文件中使用以下的别名来代替这些类的权限定名。

```
public class TypeAliasRegistry {

  private final Map<String, Class<?>> typeAliases = new HashMap<>();

  public TypeAliasRegistry() {
    registerAlias("string", String.class);

    registerAlias("byte", Byte.class);
    registerAlias("long", Long.class);
    registerAlias("short", Short.class);
    registerAlias("int", Integer.class);
    registerAlias("integer", Integer.class);
    registerAlias("double", Double.class);
    registerAlias("float", Float.class);
    registerAlias("boolean", Boolean.class);

    registerAlias("byte[]", Byte[].class);
    registerAlias("long[]", Long[].class);
    registerAlias("short[]", Short[].class);
    registerAlias("int[]", Integer[].class);
    registerAlias("integer[]", Integer[].class);
    registerAlias("double[]", Double[].class);
    registerAlias("float[]", Float[].class);
    registerAlias("boolean[]", Boolean[].class);

    registerAlias("_byte", byte.class);
```

```
    registerAlias("_long", long.class);
    registerAlias("_short", short.class);
    registerAlias("_int", int.class);
    registerAlias("_integer", int.class);
    registerAlias("_double", double.class);
    registerAlias("_float", float.class);
    registerAlias("_boolean", boolean.class);

    registerAlias("_byte[]", byte[].class);
    registerAlias("_long[]", long[].class);
    registerAlias("_short[]", short[].class);
    registerAlias("_int[]", int[].class);
    registerAlias("_integer[]", int[].class);
    registerAlias("_double[]", double[].class);
    registerAlias("_float[]", float[].class);
    registerAlias("_boolean[]", boolean[].class);

    registerAlias("date", Date.class);
    registerAlias("decimal", BigDecimal.class);
    registerAlias("bigdecimal", BigDecimal.class);
    registerAlias("biginteger", BigInteger.class);
    registerAlias("object", Object.class);

    registerAlias("date[]", Date[].class);
    registerAlias("decimal[]", BigDecimal[].class);
    registerAlias("bigdecimal[]",
BigDecimal[].class);
    registerAlias("biginteger[]",
BigInteger[].class);
    registerAlias("object[]", Object[].class);

    registerAlias("map", Map.class);
    registerAlias("hashmap", HashMap.class);
    registerAlias("list", List.class);
    registerAlias("arraylist", ArrayList.class);
    registerAlias("collection", Collection.class);
    registerAlias("iterator", Iterator.class);
```

```
    registerAlias("ResultSet", ResultSet.class);
  }
```

当然我们可以给写的类定义别名：

> 给自己类设定别名

在核心配置文件中加入

```
<typeAliases>
    <typeAlias type="com.ydlclass.entity.User"
 alias="user"/>
</typeAliases>
```

`<typeAlias>` 标签中 有 `type` 和 `alias` 两个属性

`type` 填写 实体类的全类名，`alias` 可以不填，不填的话，默认是类名，不区分大小写，

`alias` 填了的话就以 alias里的值为准。

```
<typeAiases>
    <package name=""/>
</typeAliases>
```

`<package>` 标签 为某个包下的所有类起别名； `name` 属性填写包名。 别名默认是类名，不区分大小写。

```
@Alias` 注解    加在实体类上，为某个类起别名；例：
`@Alias("User")
```

七、resultMap详解

如果数据库字段和实体的字段是一一对应，那么MyBatis会【自动映射】，但是如果不一致，比如一个叫user一个叫username，那么就需要我们手动的建立一一映射的关系了。

有时候我们的数据库和字段和实例的字段可能不是一一对应，

1、Java中的实体类设计

```
public class User {

    private int id;                 //id
    private String name;            //姓名，数据库为
username
    private String password;        //密码，一致

    //构造
    //set/get
    //toString()
}
```

3、mapper

```
//根据id查询用户
User selectUserById(int id);
```

4、mapper映射文件

```
<select id="selectUserById" resultType="user">
  select * from user where id = #{id}
</select>
```

5、测试

```
@Test
public void testSelectUserById() {
    UserMapper mapper =
session.getMapper(UserMapper.class);
    User user = mapper.selectUserById(1);
    System.out.println(user);
    session.close();
}
```

结果:

- User{id=1, name='null', password='123'}
- 查询出来发现 name为空 . 说明出现了问题!

分析:

- select * from user where id = #{id} 可以看做

  select id,username,password from user where id = #{id}

- mybatis会根据这些查询的列名(会将列名转化为小写,数据库不区分大小写) , 利用反射去对应的实体类中查找相应列名的set方法设值 , 当然找不到username

> 解决方案

方案一: 为列名指定别名 , 别名和java实体类的属性名一致 .

```
<select id="selectUserById" resultType="User">
  select id , username as name ,password from user
where id = #{id}
</select>
```

方案二: 使用结果集映射->ResultMap 【推荐】

```
<resultMap id="UserMap" type="User">
   <!-- id为主键 -->
   <id column="id" property="id"/>
   <!-- column是数据库表的列名 , property是对应实体类的属性名 -->
   <result column="username" property="name"/>
   <result column="password" property="password"/>
</resultMap>

<select id="selectUserById" resultMap="UserMap">
  select id , username , password from user where id = #{id}
</select>
```

> 结论:

这个地方我们手动调整了映射关系，称之为【手动映射】。

但如果不调整呢？ MyBatis当然会按照约定自动映射。

有了映射这种牛逼的事情之后:

我们的:

```
prepareStatement.setInt(1,21);
prepareStatement.setString(2,"IT楠老师");
```

还用写吗？两个字【**牛逼**】！

> 当然约定的最基本的操作就是全部都一样，还有就是下划线和驼峰命名的自动转化

```
<settings>
    <!--开启驼峰命名规则-->
    <setting name="mapUnderscoreToCamelCase"
value="true"/>
</settings>
```

> 自定义数据源

```
public class MyDataSource implements
DataSourceFactory {
    private Properties properties;

    @Override
    public void setProperties(Properties properties)
{
        this.properties = properties;
    }

    @Override
    public DataSource getDataSource() {

        HikariConfig hikariConfig = new
HikariConfig(properties);
        return new HikariDataSource(hikariConfig);
    }
}
```

```
username=root
password=root
jdbcUrl=jdbc:mysql://127.0.0.1:3306/ssm?
useUnicode=true&characterEncoding=utf8&serverTimezon
e=Asia/Shanghai&allowPublicKeyRetrieval=true&useSSL=
false
driverClassName=com.mysql.cj.jdbc.Driver
```

八、动态sql-很重要

1、 概述

MyBatis提供了对SQL语句动态的组装能力，大量的判断都可以在MyBatis的映射XML文件里面配置，以达到许多我们需要大量代码才能实现的功能，大大减少了我们编写代码的工作量。

> 动态SQL的元素

元素	作用	备注
if	判断语句	单条件分支判断
choose、when、otherwise	相当于Java中的 case when语句	多条件分支判断
trim、where、set	辅助元素	用于处理一些SQL拼装问题
foreach	循环语句	在in语句等列举条件常用

2、if元素（非常常用）

if元素相当于Java中的if语句，它常常与test属性联合使用。现在我们要根据name去查找学生，但是name是可选的，如下所示：

```
<select id="findUserById"
resultType="com.ydlclass.entity.User">
    select id,username,password from user
    where 1 =1
    <if test="id != null">
        AND id = #{id}
    </if>
    <if test="username != null and username != ''">
        AND username = #{username}
    </if>
    <if test="password != null and password != ''">
        AND password = #{password}
    </if>
</select>
```


3、where元素

上面的select语句我们加了一个`1=1`的绝对true的语句，目的是为了防止语句错误，变成`SELECT * FROM student WHERE`这样where后没有内容的错误语句。这样会有点奇怪，此时可以使用`<where>`元素。

```
<select id="findUserById"
resultType="com.ydlclass.entity.User">
    select id,username,password from user
    <where>
        <if test="id != null">
            AND id = #{id}
        </if>
```

```
        <if test="username != null and username != ''">
            AND username = #{username}
        </if>
        <if test="password != null and password != ''">
            AND password = #{password}
        </if>
    </where>
</select>
```

4、trim元素

有时候我们要去掉一些特殊的SQL语法，比如常见的and、or，此时可以使用trim元素。trim元素意味着我们需要去掉一些特殊的字符串，prefix代表的是语句的前缀，而prefixOverrides代表的是你需要去掉的那种字符串，suffix表示语句的后缀，suffixOverrides代表去掉的后缀字符串。

```
<select id="select" resultType="com.ydlclass.entity.User">
    SELECT * FROM user
    <trim prefix="WHERE" prefixOverrides="AND">
        <if test="username != null and username != ''">
            AND username LIKE concat('%', #{username}, '%')
        </if>
        <if test="id != null">
            AND id = #{id}
        </if>
    </trim>
</select>
```

5、choose、when、otherwise元素

> 选一个

有些时候我们还需要多种条件的选择，在Java中我们可以使用switch、case、default语句，而在映射器的动态语句中可以使用choose、when、otherwise元素。

```
<!-- 有name的时候使用name搜索，没有的时候使用id搜索 -->
<select id="select"
resultType="com.ydlclass.entity.User">
    SELECT * FROM user
    WHERE 1=1
    <choose>
        <when test="name != null and name != ''">
            AND username LIKE concat('%', #
{username}, '%')
        </when>
        <when test="id != null">
            AND id = #{id}
        </when>
    </choose>
</select>
```


6、set元素

在update语句中，如果我们只想更新某几个字段的值，这个时候可以使用set元素配合if元素来完成。**注意：set元素遇到,会自动把,去掉。**

```
<update id="update">
    UPDATE user
    <set>
        <if test="username != null and username != ''">
            username = #{username},
        </if>
        <if test="password != null and password != ''">
            password = #{password}
        </if>
    </set>
    WHERE id = #{id}
</update>
```

7、foreach元素

foreach元素是一个循环语句，它的作用是遍历集合，可以支持数组、List、Set接口。

```
<select id="select" resultType="com.ydlclass.entity.User">
    SELECT * FROM user
    WHERE id IN
    <foreach collection="ids" open="(" close=")" separator="," item="id">
        #{id}
    </foreach>
</select>
```

- collection配置的是传递进来的参数名称。
- item配置的是循环中当前的元素。
- index配置的是当前元素在集合的位置下标。
- open和 close配置的是以什么符号将这些集合元素包装起来。

- separator是各个元素的间隔符。

8、foreach批量插入

```
<insert id="batchInsert" parameterType="list">
    insert into `user`( user_name, pass)
    values
    <foreach collection="users" item="user"
separator=",">
        (#{user.username}, #{user.password})
    </foreach>

</insert>
```

9、SQL片段

有时候可能某个 sql 语句我们用的特别多，为了增加代码的重用性，简化代码，我们需要将这些代码抽取出来，然后使用时直接调用。

提取SQL片段:

```
<sql id="if-title-author">
   <if test="title != null">
      title = #{title}
   </if>
   <if test="author != null">
      and author = #{author}
   </if>
</sql>
```

引用SQL片段:

```
<select id="queryBlogIf" parameterType="map"
resultType="blog">
  select * from blog
   <where>
       <!-- 引用 sql 片段，如果refid 指定的不在本文件中，
那么需要在前面加上 namespace -->
       <include refid="if-title-author"></include>
       <!-- 在这里还可以引用其他的 sql 片段 -->
   </where>
</select>
```

九、数据关系处理

- 部门和员工的关系，一个部门多个员工，一个员工属于一个部门
- 那我们可以采取两种方式来维护关系，一种在一的一方，一种在多的一方！

> 数据库设计

```
CREATE TABLE `dept` (
`id` INT(10) NOT NULL,
`name` VARCHAR(30) DEFAULT NULL,
PRIMARY KEY (`id`)
);

INSERT INTO dept VALUES (1, '元动力学习一组'),(2, '元动
力学习二组');

CREATE TABLE `employee` (
`id` INT(10) NOT NULL,
`name` VARCHAR(30) DEFAULT NULL,
```

```
`did` INT(10) DEFAULT NULL,
PRIMARY KEY (`id`),
CONSTRAINT `fk_did` FOREIGN KEY (`did`) REFERENCES
`dept` (`id`)
) ;


INSERT INTO employee VALUES (1, '邸智伟', 1),(2, '成
虹', 2),(3, '康永亮', 1),(4, '杨春旺', 2),(5, '陈建强',
1);
```

1、通过员工级联部门

1、编写实体类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Dept  implements Serializable{

    private static final Long serialVersionUID = 1L;

    private int id;
    private String name;

}

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Employee  implements Serializable {

    private static final Long serialVersionUID = 1L;

    private int id;
```

```
    private String name;
    //维护关系
    private Dept dept;
}
```

2、编写实体类对应的Mapper接口

```
public interface DeptMapper {
}
public interface EmployeeMapper {
}
```

3、编写Mapper接口对应的 mapper.xml配置文件

```
<?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.ydlclass.dao.DeptMapper">

</mapper>


<?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.ydlclass.dao.EmployeeMapper">

</mapper>
```

4、将mapper进行注册、去mybatis-config文件中配置

```
<mapper resource="com/ydlclass/dao/DeptMapper.xml"/>
<mapper
resource="com/ydlclass/dao/EmployeeMapper.xml"/>
```

（1）按查询嵌套

这种方式是一种级联查询的方式，会产生多个sql语句，第一个sql的查询语句结果会触发第二个查询语句。

1、写方法

```
List<Employee> select(Employee employee);
```

2、mapper处理

```
<?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.ydlclass.mapper.EmployeeMapper">

    <resultMap id="employeeMap" type="employee">
        <id column="id" property="id" />
        <result column="name" property="name"/>
        <association property="dept" column="did"
javaType="dept"

select="com.ydlclass.mapper.DeptMapper.select">
            <id column="id" property="id"/>
            <result column="name" property="name"/>
```

```
        </association>
    </resultMap>

    <sql id="sql">
        `id`,`name`,`did`
    </sql>


    <select id="select" resultMap="employeeMap">
        select <include refid="sql" />
        from employee
    </select>

</mapper>
```

```
<?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.ydlclass.mapper.DeptMapper">

    <resultMap id="deptMap" type="dept">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
        <collection property="employees"
javaType="list" ofType="employee"
                    column="id"
select="com.ydlclass.mapper.EmployeeMapper.selectByD
id">
            <id column="eid" property="id"/>
            <result column="ename" property="name"/>
        </collection>
    </resultMap>
```

```
    <sql id="sql">
        `id`
        ,`name`
    </sql>

    <select id="select" resultMap="deptMap">
        select
        <include refid="sql"/>
        from dept where id = #{id}
    </select>

</mapper>
```

3、编写完毕去Mybatis配置文件中，注册Mapper

4、测试

```
@Test
public void testSelect() {
    try (SqlSession session =
sqlSessionFactory.openSession(true)) {
        // statement是sql的申明
        EmployeeMapper mapper =
session.getMapper(EmployeeMapper.class);
        List<Employee> employees = mapper.select(new
Employee());

        log.debug("The employees are
[{}]",employees);
    }
}
```

结果：

```
>  Preparing: select `id`,`name`,`did` from employee
> Parameters:
 Preparing: select `id` ,`name` from dept where id = ?
Parameters: 1(Integer)
UG ======>  Preparing: select id eid,name ename from employee where did=?
UG ======> Parameters: 1(Integer)
UG <======      Total: 3
     Total: 1
 Preparing: select `id` ,`name` from dept where id = ?
```

（2）按结果嵌套

结果嵌套是使用复杂查询，在根据结果的字段进行对象的封装，本质只会发送一个sql。

1、接口方法编写

```
List<Employee> select2(Employee employee);
```

2、编写对应的mapper文件

```
<?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.ydlclass.mapper.EmployeeMapper">

    <resultMap id="employeeMap2" type="employee">
        <id column="eid" property="id" />
        <result column="ename" property="name"/>
        <association property="dept" javaType="dept"
>
            <id column="did" property="id"/>
            <result column="dname" property="name"/>
```

```
        </association>
    </resultMap>

    <select id="select2" resultMap="employeeMap2" >
        select e.id eid,e.name ename, d.id
did,d.name dname
        from employee e left join dept d
                                    on d.id = e.did
    </select>

</mapper>
```

4、测试

```
@Test
public void testFindAllEmployees2() {
    EmployeeMapper mapper =
session.getMapper(EmployeeMapper.class);
    List<Employee> allEmployees =
mapper.findAllEmployees2();
    for (Employee allEmployee : allEmployees) {
        System.out.println(allEmployee);
    }
}
```


2、通过部门级联用户

在部门处维护关系，此处可以联想订单和订单详情。

修改实体类：

```java
@Data
public class Dept {
    private int id;
    private String name;
    //用于关系维护
    List<Employee> employees;
}

@Data
public class Employee {
    private int id;
    private String name;
}
```

（1）按结果嵌套处理

1、写方法

```java
public interface DeptMapper {
    List<Dept> select(Dept dept);
}
```

2、写配置文件

```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.ydlclass.mapper.DeptMapper">

    <resultMap id="deptMap" type="dept">
        <id column="id" property="id"/>
        <result column="name" property="name"/>
```

```
        <collection property="employees"
javaType="list" ofType="employee"
                    column="id"
select="com.ydlclass.mapper.EmployeeMapper.selectByD
id">
            <id column="eid" property="id"/>
            <result column="ename" property="name"/>
        </collection>
    </resultMap>

    <sql id="sql">
        `id`,`name`
    </sql>

    <select id="select" resultMap="deptMap">
        select
        <include refid="sql"/>
        from dept where id = #{id}
    </select>

</mapper>
```

在EmployeeMapper中添加select进行级联查询

```
<?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.ydlclass.mapper.EmployeeMapper">

    <resultMap id="employeeMap" type="employee">
        <id column="id" property="id" />
        <result column="name" property="name"/>
        <association property="dept" column="did"
javaType="dept"
```

```
select="com.ydlclass.mapper.DeptMapper.select">
            <id column="id" property="id"/>
            <result column="name" property="name"/>

        </association>
    </resultMap>

    <resultMap id="employeeMap2" type="employee">
        <id column="eid" property="id" />
        <result column="ename" property="name"/>
        <association property="dept" javaType="dept"
>
            <id column="did" property="id"/>
            <result column="dname" property="name"/>
        </association>
    </resultMap>

    <select id="selectByDid"
resultMap="employeeMap2" >
        select id eid,name ename from employee where
did=#{did}
    </select>

</mapper>
```


3、测试

```
@Test
public void testSelect() {
    try (SqlSession session =
sqlSessionFactory.openSession(true)) {
        // statement是sql的申明
        DeptMapper mapper =
session.getMapper(DeptMapper.class);
        List<Dept> depts = mapper.select(null);
        log.debug("The depts are [{}]", depts);
    }
}
```

```
aring: select `id` ,`name` from dept
eters:
====>  Preparing: select id eid,name ename from employee where did=?
====> Parameters: 1(Integer)
<====      Total: 3
====>  Preparing: select id eid,name ename from employee where did=?
====> Parameters: 2(Integer)
<====      Total: 2
```


（2）按查询嵌套处理

1、TeacherMapper接口编写方法

```
List<Dept> select2(Dept dept);
```

2、编写接口对应的Mapper配置文件

```
<?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.ydlclass.mapper.DeptMapper">
    <resultMap id="deptMap2" type="dept">
```

```
        <id column="did" property="id"/>
        <result column="dname" property="name"/>
        <collection property="employees"
javaType="list" ofType="employee">
            <id column="eid" property="id"/>
            <result column="ename" property="name"/>

        </collection>
    </resultMap>

    <sql id="sql">
        `id`,`name`
    </sql>


    <select id="select2" resultMap="deptMap2">
        select d.id did, d.name dname, e.id eid,
e.name ename
        from dept d
                left join employee e on d.id =
e.did
    </select>
</mapper>
```

3、测试

```
@Test
public void testSelect2() {
    try (SqlSession session =
sqlSessionFactory.openSession(true)) {
        // statement是sql的申明
        DeptMapper mapper =
session.getMapper(DeptMapper.class);
        List<Dept> depts = mapper.select2(null);
        log.debug("The depts are [{}]", depts);
    }
}
```

结果：

```
ring: select d.id did, d.name dname, e.id eid, e.name ename from dept d left join
ters:
otal: 5
es=[Employee(id=1, name=邸智伟, dept=null), Employee(id=3, name=康永亮, dept=null),
BC Connection [com.mysql.cj.jdbc.ConnectionImpl@7ba63fe5]
```

3、小知识：

（1）懒加载

通俗的讲就是按需加载，我们需要什么的时候再去进行什么操作。而且先从单表查询，需要时再从关联表去关联查询，能大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。

在mybatis中，resultMap可以实现高级映射(使用association、collection实现一对一及一对多映射)，association、collection具备延迟加载功能。

```
<!-- 开启懒加载配置 -->
<settings>
  <!-- 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 -->
  <setting name="lazyLoadingEnabled" value="true"/>
  <!-- 开启时，任一方法的调用都会加载该对象的所有延迟加载属性。 否则，每个延迟加载属性会按需加载 -->
  <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

（2）Mybatis 获取自增主键的方式

> 使用 xml

直接在标签属性上添加 useGeneratedKeys（是否是自增长，必须设置 true） 和 keyProperty（实体类主键属性名称） 、keyColumn（数据库主键字段名称）

```
<insert id="insert" useGeneratedKeys="true"
keyColumn="id" keyProperty="id">
    insert into `user`(id, username, password)
    values (#{id}, #{username}, #{password})
</insert>
```

> 注解方式 useGeneratedKeys（是否是自增长，必须设置 true） 和 keyProperty（实体类主键属性名称） 、keyColumn（数据库主键字段名称）

```
@Insert("INSERT INTO user(name,age) VALUES(#
{user.name},#{user.age})")
@Options(useGeneratedKeys=true,
keyProperty="user.id",keyColumn="id"  )
public int insert(@Param("user")User user);
```


也可以全局设置

```
<setting name="useGeneratedKeys" value="true"/>
```

十、Mybatis缓存

1、为什么要用缓存?

- 如果缓存中有数据，就不用从数据库获取，大大提高系统性能。
- MyBatis提供一级缓存和二级缓存

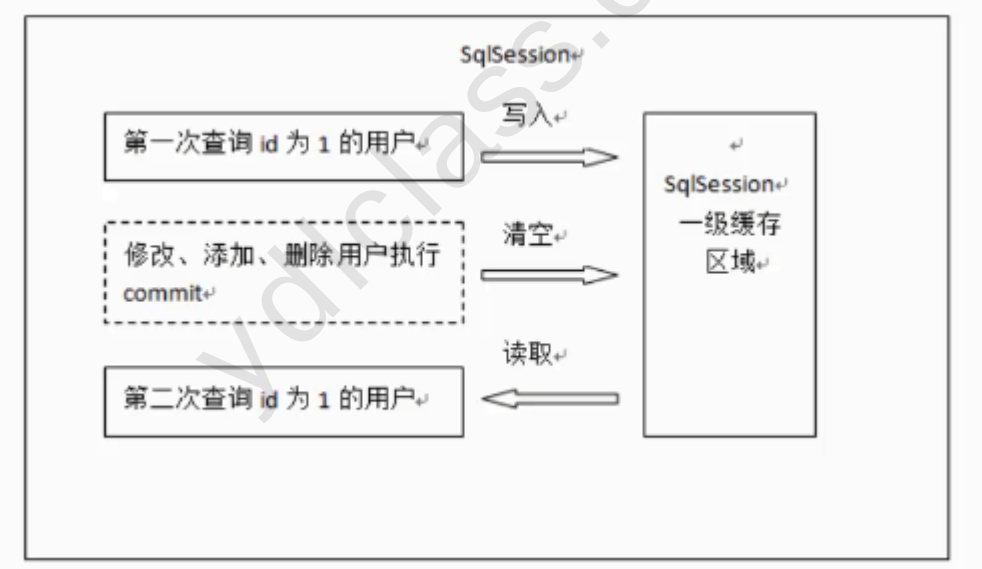
2、一级缓存：

> 一级缓存是sqlsession级别的缓存

- 在操作数据库时，需要构造sqlsession对象，在对象中有一个数据结构（HashMap）用于存储缓存数据
- 不同的sqlsession之间的缓存区域是互相不影响的。

> 一级缓存工作原理：

图解：



- 第一次发起查询sql查询用户id为1的用户，先去找缓存中是否有id为1的用户，如果没有，再去数据库查询用户信息。得到用户信息，将用户信息存储到一级缓存中。
- 如果sqlsession执行了commit操作（插入，更新，删除），会清空sqlsession中的一级缓存，避免脏读

- 第二次发起查询id为1的用户，缓存中如果找到了，直接从缓存中获取用户信息
- MyBatis默认支持并开启一级缓存。

> 一级缓存演示

1、必须配置日志，要不看不见

2、编写接口方法

```
//根据id查询用户
User findUserById(@Param("id") int id);
```

3、接口对应的Mapper文件

```
<select id="findUserById"
resultType="com.ydlclass.entity.User">
  select * from user where id = #{id}
</select>
```

4、测试

```
@Test
public void testFindUserById(){
    UserMapper mapper =
session.getMapper(UserMapper.class);
    User user1 = mapper.findUserById(1);
    System.out.println(user1);
    User user2 = mapper.findUserById(3);
    System.out.println(user2);
    User user3 = mapper.findUserById(1);
    System.out.println(user3);
}
```

5、通过日志分析

```
[com.ydlclass.dao.UserMapper.findUserById]-==>
Preparing: select id,username,password from user
where id = ?
[com.ydlclass.dao.UserMapper.findUserById]-==>
Parameters: 1(Integer)
[com.ydlclass.dao.UserMapper.findUserById]-<==
Total: 1
User{id=1, username='楠哥', password='123456'}
  ---->ID为1，第一次有sql
[com.ydlclass.dao.UserMapper.findUserById]-==>
Preparing: select id,username,password from user
where id = ?
[com.ydlclass.dao.UserMapper.findUserById]-==>
Parameters: 3(Integer)
[com.ydlclass.dao.UserMapper.findUserById]-<==
Total: 1
User{id=3, username='磊哥', password='987654'}
  ---->ID为3，第一次有sql
User{id=1, username='楠哥', password='123456'}
  ---->ID为1，第二次无sql，走缓存
```


一级缓存失效

1. sqlSession不同
2. 当sqlSession对象相同的时候，查询的条件不同，原因是第一次查询时候，一级缓存中没有第二次查询所需要的数据
3. 当sqlSession对象相同，两次查询之间进行了插入的操作
4. 当sqlSession对象相同，手动清除了一级缓存中的数据

一级缓存生命周期

1. MyBatis在开启一个数据库会话时，会创建一个新的SqlSession对象，SqlSession对象中会有一个新的Executor对象，

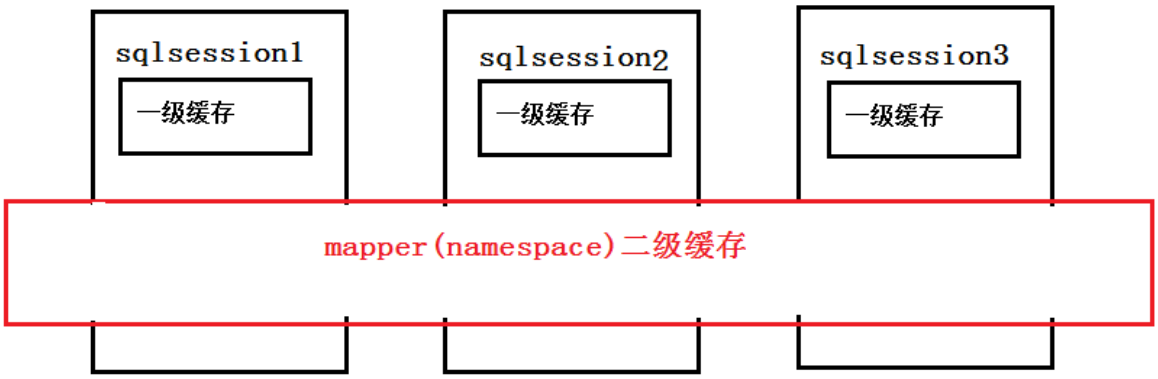
Executor对象中持有一个新的PerpetualCache对象；当会话结束时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉。

2. 如果SqlSession调用了close()方法，会释放掉一级缓存PerpetualCache对象，一级缓存将不可用。
3. 如果SqlSession调用了clearCache()，会清空PerpetualCache对象中的数据，但是该对象仍可使用。
4. SqlSession中执行了任何一个update操作(update()、delete()、insert())，都会清空PerpetualCache对象的数据，但是该对象可以继续使用。

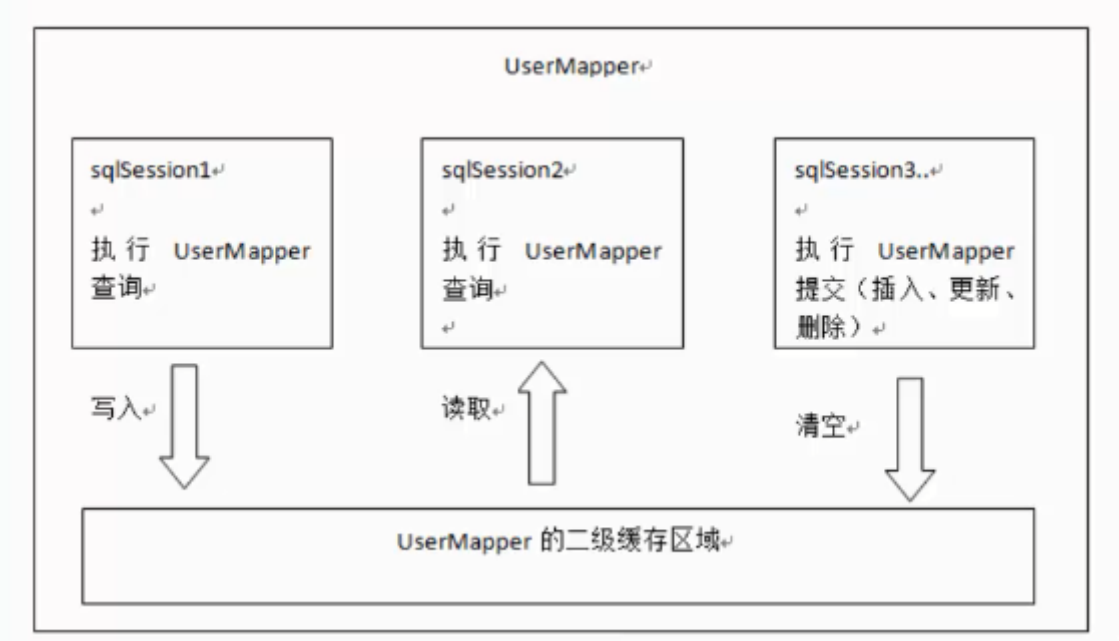
3、二级缓存：

> 二级缓存是mapper级别的缓存

- 多个sqlsession去操作同一个mapper的sql语句，多个sqlsession可以共用二级缓存，所得到的数据会存在二级缓存区域
- 二级缓存是跨sqlsession的
- 二级缓存相比一级缓存的范围更大（按namespace来划分），多个sqlsession可以共享一个二级缓存



> 二级缓存实现原理



首先要手动开启MyBatis二级缓存。

在config.xml设置二级缓存开关， 还要在具体的mapper.xml开启二级缓存

```
<settings>
    <!--开启二级缓存-->
    <setting name="cacheEnabled" value="true"/>

</settings>
<!-- 需要将映射的javabean类实现序列化 -->
```

class Student implements Serializable{}

```
<!--开启本Mapper的namespace下的二级缓存-->
<cache eviction="LRU" flushInterval="100000"/>
```

（1）cache属性的简介：

> **eviction**回收策略（缓存满了的淘汰机制），目前MyBatis提供以下策略。

1. LRU（Least Recently Used），最近最少使用的，最长时间不用的对象
2. FIFO（First In First Out），先进先出，按对象进入缓存的顺序来移除他们
3. SOFT，软引用，移除基于垃圾回收器状态和软引用规则的对象，当内存不足，会触发JVM的GC，如果GC后，内存还是不足，就会把软引用的包裹的对象给干掉，也就是只有内存不足，JVM才会回收该对象。
4. WEAK，弱引用，更积极的移除基于垃圾收集器状态和弱引用规则的对象。弱引用的特点是不管内存是否足够，只要发生GC，都会被回收。

> **flushInterval**:刷新间隔时间，单位为毫秒，

1. 这里配置的是100秒刷新，如果你不配置它，那么当SQL被执行的时候才会去刷新缓存。

> **size**:引用数目，

1. 一个正整数，代表缓存最多可以存储多少个对象，不宜设置过大。设置过大会导致内存溢出。

   这里配置的是1024个对象

> **readOnly:**只读，

1. 意味着缓存数据只能读取而不能修改，这样设置的好处是我们可以快速读取缓存，缺点是我们没有

   办法修改缓存，它的默认值是false，不允许我们修改

（2）操作过程：

sqlsession1查询用户id为1的信息，查询到之后，会将查询数据存储到二级缓存中。

如果sqlsession3去执行相同mapper下sql，执行commit提交，会清空该mapper下的二级缓存区域的数据

sqlsession2查询用户id为1的信息， 去缓存找是否存在缓存，如果存在直接从缓存中取数据

> 禁用二级缓存：

在statement中可以设置useCache=false，禁用当前select语句的二级缓存，默认情况为true

```
<select id="getStudentById"
parameterType="java.lang.Integer"
resultType="Student" useCache="false">
```

在实际开发中，针对每次查询都需要最新的数据sql，要设置为useCache="false"，禁用二级缓存

> flushCache标签：刷新缓存（清空缓存）

```
<select id="getStudentById"
parameterType="java.lang.Integer"
resultType="Student" flushCache="true">
```

一般下执行完commit操作都需要刷新缓存，flushCache="true 表示刷新缓存，可以避免脏读

> 二级缓存应用场景

对于访问多的查询请求并且用户对查询结果实时性要求不高的情况下，可采用MyBatis二级缓存，降低数据库访问量，提高访问速度，如电话账单查询

根据需求设置相应的**flushInterval**：刷新间隔时间，比如三十分钟，24小时等。

> 二级缓存局限性：

MyBatis二级缓存对细粒度的数据级别的缓存实现不好，比如如下需求：对商品信息进行缓存，由于商品信息查询访问量大，但是要求用户每次都能查询最新的商品信息，此时如果使用MyBatis的二级缓存就无法实现当一个商品变化时只刷新该商品的缓存信息而不刷新其它商品的信息，因为MyBatis的二级缓存区域以mapper为单位划分，当一个商品信息变化会将所有商品信息的缓存数据全部清空。解决此类问题需要在业务层根据需求对数据有针对性缓存。

> 二级缓存演示

先不进行配置

```
@Test
public void testFindUserCache() throws Exception {

    //使用不同的mapper
    UserMapper mapper1 =
session.getMapper(UserMapper.class);
    User user1 = mapper1.findUserById(1);
    System.out.println(user1);
    //提交了就会刷到二级缓存，要不还在一级缓存，一定要注意
    session.commit();
    UserMapper mapper2 =
session.getMapper(UserMapper.class);
    User user2 = mapper2.findUserById(1);
    System.out.println(user2);
    System.out.println(user1 == user2);
}
```

结果：

```
[com.ydlclass.dao.UserMapper.findUserById]-==>
Preparing: select id,username,password from user
where id = ?
[com.ydlclass.dao.UserMapper.findUserById]-==>
Parameters: 1(Integer)
[com.ydlclass.dao.UserMapper.findUserById]-<==
Total: 1
User{id=1, username='楠哥', password='123456'}
[com.ydlclass.dao.UserMapper.findUserById]-==>
Preparing: select id,username,password from user
where id = ?
[com.ydlclass.dao.UserMapper.findUserById]-==>
Parameters: 1(Integer)
[com.ydlclass.dao.UserMapper.findUserById]-<==
Total: 1
User{id=1, username='楠哥', password='123456'}
false                    ---->两个对象不是一个，发了两个sql，
说明缓存没有起作用
```

可以看见两次同样的sql，却都进库进行了查询。说明二级缓存没开。

> 配置二级缓存

1、开启全局缓存

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

2、使用二级缓存，这个写在mapper里

```
<!--开启本Mapper的namespace下的二级缓存-->
<cache eviction="LRU" flushInterval="100000"
size="512" readOnly="true"></cache>
<!--
创建了一个 LRU 最少使用清除缓存，每隔 100 秒刷新，最多可以存储 512 个对象，返回的对象是只读的。
-->
```

3、测试执行

```
[com.ydlclass.dao.UserMapper.findUserById]-==>
 Preparing: select id,username,password from user
where id = ?
[com.ydlclass.dao.UserMapper.findUserById]-==>
Parameters: 1(Integer)
[com.ydlclass.dao.UserMapper.findUserById]-<==
 Total: 1
User{id=1, username='楠哥', password='123456'}
[com.ydlclass.dao.UserMapper]-Cache Hit Ratio
[com.ydlclass.dao.UserMapper]: 0.5
User{id=1, username='楠哥', password='123456'}
true                    ---->两个对象一样了，就发了一个
sql，说明缓存起了作用
```


4、第三方缓存--EhCache充当二级缓存

> 此外还有很多第三方缓存组件，最常用的比如ehcache，Memcached、redis等，我们以比较简单的ehcache为例。

1、引入依赖

```
<!--
https://mvnrepository.com/artifact/org.mybatis.caches/mybatis-ehcache -->
<dependency>
    <groupId>org.mybatis.caches</groupId>
    <artifactId>mybatis-ehcache</artifactId>
    <version>1.2.1</version>
</dependency>
```

2、修改mapper.xml中使用对应的缓存

```
<mapper namespace = “com.ydlclass.entity.User” >
        <cache
type="org.mybatis.caches.ehcache.EhcacheCache"
eviction="LRU" flushInterval="10000" size="1024"
readOnly="true"/>
</mapper>
```

3、添加ehcache.xml文件，ehcache配置文件，具体配置自行百度

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

 xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
         updateCheck="false">
    <!--
       diskStore：为缓存路径，ehcache分为内存和磁盘两级，此属性定义磁盘的缓存位置。参数解释如下：
       user.home – 用户主目录
       user.dir – 用户当前工作目录
       java.io.tmpdir – 默认临时文件路径
     -->
    <diskStore path="./tmpdir/Tmp_EhCache"/>
```

```
    <defaultCache
            eternal="false"
            maxElementsInMemory="10000"
            overflowToDisk="false"
            diskPersistent="false"
            timeToIdleSeconds="1800"
            timeToLiveSeconds="259200"
            memoryStoreEvictionPolicy="LRU"/>

</ehcache>

<!--
        name:缓存名称。
        maxElementsInMemory：缓存最大个数。
        eternal:对象是否永久有效，一但设置了，timeout将不起作用。
        timeToIdleSeconds：设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。
        timeToLiveSeconds：设置对象在失效前允许存活时间（单位：秒）。最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时间无穷大。
        overflowToDisk：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中。
        diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。
        maxElementsOnDisk：硬盘最大缓存个数。
        diskPersistent：是否在磁盘上持久化。指重启jvm后，数据是否有效。默认为false。
        memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU（最近最少使用）。你可以设置为FIFO（先进先出）或是LFU（较少使用）。
        clearOnFlush：内存数量最大时是否清除。
    -->
```

3、测试

5、使用缓存独立控制

> 其实我们更加常见的是使用第三方的缓存进行存储，并且自由控制

```
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.3</version>
</dependency>
```

> 测试一下

```
final CacheManager cacheManager = new
CacheManager(this.getClass().getClassLoader().getRes
ourceAsStream("ehcache.xml"));

// create the cache called "hello-world"
String[] cacheNames = cacheManager.getCacheNames();
for (String cacheName : cacheNames) {
    System.out.println(cacheName);
}
Cache userDao = cacheManager.getCache("userDao");
Element element = new
Element("testFindUserById_1",new User(1,"q","d"));
userDao.put(element);

Element element1 =
userDao.get("testFindUserById_1");
User user = (User)element1.getObjectValue();
```

```
System.out.println(user);
```

> 创建工具类

```
/**
 * @author zn
 * @date 2021/1/28
 */
public class CacheUtil {

    private static CacheManager cacheManager = new
CacheManager(CacheUtil.class.getClassLoader().getRes
ourceAsStream("ehcache.xml"));

    public static void put(String cacheName,String
key,Object value){
        Cache cache =
cacheManager.getCache(cacheName);
        Element element = new Element(key,value);
        cache.put(element);
    }

    public static Object get(String cacheName,String
key){
        Cache cache =
cacheManager.getCache(cacheName);
        return cache.get(key).getObjectValue();

    }

    public static boolean  delete(String
cacheName,String key){
        Cache cache =
cacheManager.getCache(cacheName);
        return cache.remove(key);
```

```
    }

}
```

测试

```
@Test
public void selectDeptsTest(){

    Map<String,Object> cache = new HashMap<>(8);
    // 执行方法
    List<Dept> depts = deptMapper.selectDepts();

 CacheUtil.put("dept","selectUserByIdTest",depts);
    System.out.println(depts);
    session.commit();

    session = sqlSessionFactory.openSession();
    // 去缓存拿，如果有就拿出来
    List<Dept> newDepts = null;
    Object object = CacheUtil.get("dept",
"selectUserByIdTest");
    if(object == null){
        // 去数据库查询
        DeptMapper mapper =
session.getMapper(DeptMapper.class);
        newDepts = mapper.selectDepts();
    } else {
        newDepts = (List<Dept>) object;
    }

    System.out.println(newDepts);
}
```

十一、配置文件

> MyBatis的配置文件分为核心配置文件和mapper配置文件

- mybatis-config.xml 系统核心配置文件
- 核心配置文件主要配置Mybatis一些**基础组件和加载资源**，核心配置文件中的元素常常能影响Mybatis的整个运行过程。
- 能配置的内容如下，**顺序不能乱**：

```
1.properties是一个配置属性的元素
2.settings设置，mybatis最为复杂的配置也是最重要的，会改变mybatis运行时候的行为
3.typeAliases别名（在TypeAliasRegistry中可以看到mybatis提供了许多的系统别名）
4.typeHandlers 类型处理器（比如在预处理语句中设置一个参数或者从结果集中获取一个参数时候，都会用到类型处理器，在TypeHandlerRegistry中定义了很多的类型处理器）
5.objectFactory 对象工厂（myabtis在构建一个结果返回的时候，会使用一个ObjectFactory去构建pojo）
6.plugins 插件
7.environments 环境变量
    environment 环境变量
      transactionManager 事务管理器
      dataSource 数据源
      databaseIdProvider 数据库厂商标识
8.mappers 映射器
```

> **找几个重要的讲解一下**

1、environments元素

environments可以为mybatis配置多环境运行，将SQL映射到多个不同的数据库上，必须指定其中一个为默认运行环境（通过default指定），如果想切换环境修改default的值即可。

> 最常见的就是，生产环境和开发环境，两个环境切换必将导致数据库的切换。

```
<environments default="development">
 <environment id="development">
   <transactionManager type="JDBC">
     <property name="..." value="..."/>
   </transactionManager>
   <dataSource type="POOLED">
     <property name="driver" value="${driver}"/>
     <property name="url" value="${url}"/>
     <property name="username" value="${username}"/>
     <property name="password" value="${password}"/>
   </dataSource>
 </environment>

  <environment id="product">
       <transactionManager type="JDBC">
         <property name="..." value="..."/>
       </transactionManager>
       <dataSource type="POOLED">
         <property name="driver" value="${driver}"/>
         <property name="url" value="${url}"/>
         <property name="username"
value="${username}"/>
         <property name="password"
value="${password}"/>
       </dataSource>
 </environment>
</environments>
```

- dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。
- 数据源是必须配置的。
- 有三种内建的数据源类型

```
type="[UNPOOLED|POOLED|JNDI]"）
- unpooled：这个数据源的实现只是每次被请求时打开和关闭连接。
- pooled：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来 ，这是一种使得并发 Web 应用快速响应请求的流行处理方式。
- jndi：这个数据源的实现是为了能在如 Spring 或应用服务器这类容器中使用，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。
```

- 数据源也有很多第三方的实现，比如druid，hikari，dbcp，c3p0等等....
- 这两种事务管理器类型都不需要设置任何属性。
- 具体的一套环境，通过设置id进行区别，id保证唯一!
- 子元素节点：transactionManager - [事务管理器]

```
<!-- 语法 -->
<transactionManager type="[ JDBC | MANAGED ]"/>
```

- 子元素节点：**数据源（dataSource)**

2、mappers元素

> mappers的存在就是要对写好的mapper和xml进行统一管理
>
> 要不然系统怎么知道我写了哪些mapper

通常这么引入

```
<mappers>
    <!-- 使用相对于类路径的资源引用 -->
    <mapper
resource="com/ydlclass/dao/userMapper.xml"/>
    <!-- 面向注解时使用全类名 -->
    <mapper class="com.ydlclass.dao.AdminMapper"/>
</mappers>

。。。还有其他的方式
```

Mapper文件

```
<?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.ydlclass.mapper.UserMapper">

</mapper>
```

- namespace中文意思：命名空间，作用如下：
- namespace的命名必须跟某个接口同名，这才能找的到啊！

3、Properties元素

数据库连接信息我们最好放在一个单独的文件中。

1、 在资源目录下新建一个db.properties

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/ssm?
useSSL=true&useUnicode=true&characterEncoding=utf8
username=root
password=root
```

2、 将文件导入properties 配置文件

```
<configuration>
    <!--导入properties文件-->
    <properties resource="db.properties"/>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver"
value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username"
value="${username}"/>
                <property name="password"
value="${password}"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="mapper/UserMapper.xml"/>
    </mappers>
</configuration>
```


4、其他配置浏览

> settings能对我的一些核心功能进行配置，如懒加载、日志实现、缓存开启关闭等

> 简单参数说明

设置参数	描述	有效值	默认值
cacheEnabled	该配置影响的所有映射器中配置的缓存的全局开关。	true | false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。特定关联关系中可通过设置 fetchType 属性来覆盖该项的开关状态。	true | false	false
useColumnLabel	使用列标签代替列名。不同的驱动在这方面会有不同的表现，具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。	true | false	true

设置参数	描述	有效值	默认值
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要驱动兼容。如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。	true | false	False
defaultStatementTimeout	设置超时时间，它决定驱动等待数据库响应的秒数。	Any positive integer	Not Set (null)
mapUnderscoreToCamelCase	是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A_COLUMN 到经典 Java 属性名 aColumn 的类似映射。	true | false	False

设置参数	描述	有效值	默认值
logPrefix	指定 MyBatis 增加到日志名称的前缀。	Any String	Not set
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	Not set

> 完整的 settings 元素，有很多可以配置的选项，需要大家慢慢自己学习：

```
<settings>
    <!---->
     <setting name="cacheEnabled" value="true"/>
    <!---->
     <setting name="lazyLoadingEnabled"
value="true"/>
    <!---->
     <setting name="multipleResultSetsEnabled"
value="true"/>
     <setting name="useColumnLabel" value="true"/>
     <setting name="useGeneratedKeys"
value="false"/>
     <setting name="autoMappingBehavior"
value="PARTIAL"/>
     <setting
name="autoMappingUnknownColumnBehavior"
value="WARNING"/>
```

```
    <setting name="defaultExecutorType"
value="SIMPLE"/>
    <setting name="defaultStatementTimeout"
value="25"/>
    <setting name="defaultFetchSize" value="100"/>
    <setting name="safeRowBoundsEnabled"
value="false"/>
    <setting name="mapUnderscoreToCamelCase"
value="false"/>
    <setting name="localCacheScope"
value="SESSION"/>
    <setting name="jdbcTypeForNull" value="OTHER"/>
    <setting name="lazyLoadTriggerMethods"
value="equals,clone,hashCode,toString"/>
</settings>
```