

## 课程目标

## 内容定位

### 1. MeBatis需求分析

- 1.1. 项目目标:为什么要做这个项目? 做成什么样?
- 1.2. 核心功能:这个框架需要解决什么问题?
- 1.3. 功能分解:这个框架要怎么解决这些问题?
  - 1.3.1. 1) 核心对象
  - 1.3.2. 2) 操作流程 (绘图)

### 2. V1.0的实现

- 2.1. SqlSession
- 2.2. Configuration
- 2.3. MapperProxy
- 2.4. Executor

### 3. V1.0的不足

- 3.1. 不足
- 3.2. 代码优化目标
- 3.3. 功能增强目标

### 4. V2.0的实现

- 4.1. 怎么解决数据库连接硬编码的问题?
- 4.2. Executor的职责怎么拆分?
  - 4.2.1. 参数处理ParameterHandler
  - 4.2.2. 结果集处理ResultSetHandler
  - 4.2.3. 语句执行处理StatementHandler
- 4.3. Configuration和其他代码的细化
- 4.4. 支持注解
- 4.5. 支持查询缓存CachingExecutor
- 4.6. 支持插件
  - 4.6.1. 定义Interceptor接口
  - 4.6.2. 定义InterceptorChain容器
  - 4.6.3. 定义Plugin代理类
  - 4.6.4. 定义Invocation包装类
  - 4.6.5. 自定义插件

### 5. 工作流程分析

- 5.1. 启动解析
- 5.2. 获取SqlSession

## 课程目标

- 1、实现1.0版本，掌握MyBatis的本质、核心功能、核心对象、执行流程
- 2、通过分析2.0版本，体验框架的演进过程，理解 MyBatis 设计思想与细节

## 内容定位

适合已经掌握MyBatis的基本使用，理解了MyBatis的工作原理，想要进一步理解MyBatis 为什么这么设计的学生。

## 1. MeBatis需求分析

假如你在一家软件公司的研发部工作，有一天技术总监老王想让你负责开发一个项目，你要做的第一件事情是什么？

确定需求。

### 1.1. 项目目标:为什么要做这个项目？做成什么样？

老王说：我发现在业务复杂的项目中，开发的兄弟们用JDBC操作数据库太麻烦了，想要把一些基础的操作做一个封装和提取，让开发的兄弟们更加专注于业务的开发，这样就可以提升开发效率，远离996。

原来是一个操作数据库的框架。

那么我要问一下老王：这个项目要做什么，才简化我们对数据库的操作呢？或者说，在业务复杂的项目中使用JDBC操作数据库，麻烦在哪里？

### 1.2. 核心功能:这个框架需要解决什么问题？

老王给我看了一段JDBC的代码：

- 1) 它需要实现对连接资源的自动管理，也就是把创建 Connection、创建 Statement、关闭 Connection、关闭 Statement这些操作封装到底层的对象中，不需要在应用层手动调用。

```
1. rs.close();
2. stmt.close();
3. conn.close();
```

- 2) 它需要把SQL 语句抽离出来实现集中管理，开发人员不用在业务代码里面写SQL语句。

```
1. String sql = "SELECT bid, name, author id FROM blog where bid = 1";
2. ResultSet rs = stmt.executeQuery(sql);
```

3) 它需要实现对结果集转换，也就是我们指定了映射规则之后，这个框架会自动帮我们把ResultSet 映射成实体类对象。

```
1. Integer bid = rs.getInt("bid");
2. String name = rs.getString("name");
3. Integer authorId = rs.getInt("author_id");
4. blog.setAuthorId(authorId);
5. blog.setBid(bid);
6. blog.setName(name);
```

4) 做了这些事以后，这个框架需要提供一个API来给我们操作数据库，这里面定义了对数据库的操作的常用的方法。

## 1.3. 功能分解:这个框架要怎么解决这些问题?

老王的需求我已经清楚了，这个框架应该怎么解决这些问题呢？

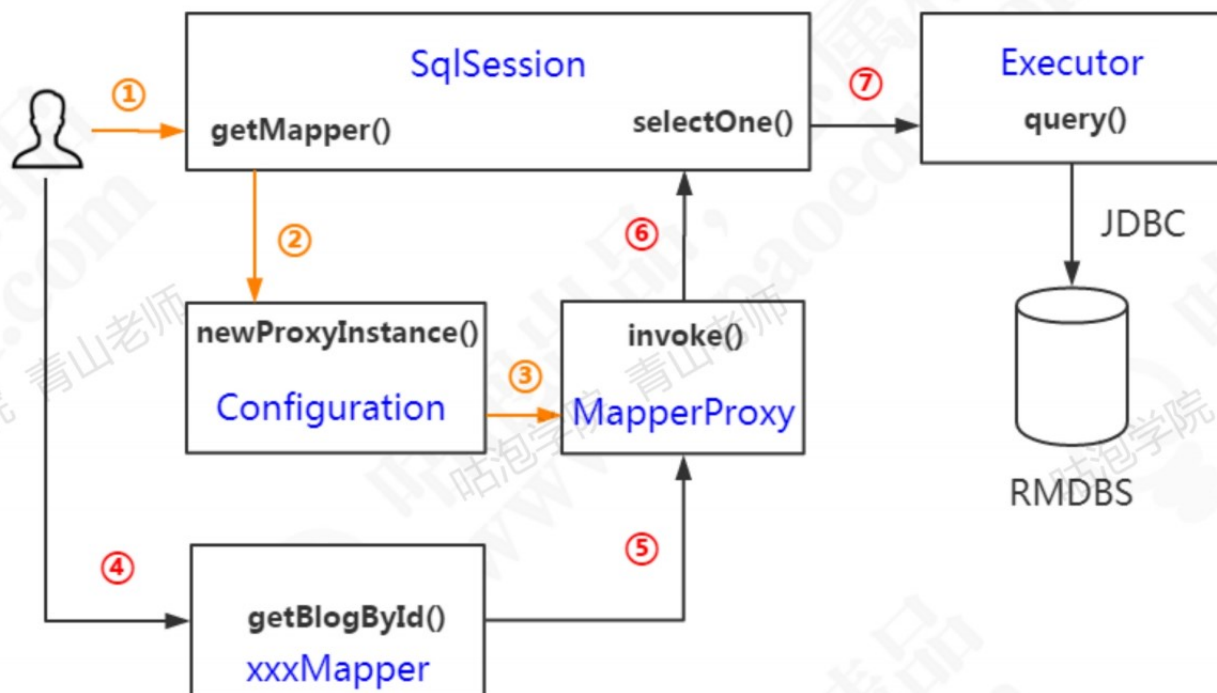
我们先来分析一下需要哪些核心对象：

### 1.3.1. 1) 核心对象

- 1、存放参数和结果映射关系、存放SQL语句，我们需要定义一个配置类；
- 2、执行对数据库的操作，处理参数和结果集的映射，创建和释放资源，我们需要定义一个执行器；
- 3、有了这个执行器以后，我们不能直接调用它，而是定义一个给应用层使用的API，它可以根据SQL的id找到SQL语句，交给执行器执行；
- 4、如果由用户直接使用id 查找SQL语句太麻烦了，我们干脆把存放SQL的命名空间定义成一个接口，把SQL的id定义成方法，这样只要调用接口方法就可以找到要执行的SQL。刚好动态代理可以实现这个功能。这个时候我们需要引入一个代理类。

核心对象有了，第二个：我们分析一下这个框架操作数据库的主要流程，先从单条查询入手。

### 1.3.2. 2) 操作流程（绘图）



- 1、定义配置类对象Configuration。里面要存放SQL语句，还有查询方法和结果映射的关系。
- 2、定义应用层的API SqlSession。在SqlSession里面封装增删改查和操作事务的方法 (selectOne())。
- 3、如果直接把Statement ID传给SqlSession去执行SQL，会出现硬编码，我们决定把SQL 语句的标识设计成一个接口+方法名 (Mapper接口)，调用接口的方法就能找到SQL语句。
- 4、这个需要代理模式实现，所以要创建一个实现了InvocationHandler的触发管理类 MapperProxy。代理类在Configuration中通过JDK动态代理创建。
- 5、有了代理对象之后，我们调用接口方法，就是调用触发管理器MapperProxy的invoke () 方法。
- 6、代理对象的invoke()方法调用了SqlSession的selectOne ()。
- 7、SqlSession只是一个API，还不是真正的SQL执行者，所以接下来会调用执行器Executor的 query () 方法。
- 8、执行器Executor的query () 方法里面就是对JDBC底层的Statement的封装，最终实现对数据库的操作，和结果的返回。

以上就是我们对这个框架的查询主要流程的总结。

基于这个流程，接下来我们就要动手去写这个框架了。

我们给它起个名字叫MeBatis。

## 2. V1.0的实现

目标是查询一张blog的表。

```

1. CREATE TABLE `blog` (
2.     `bid` int(11) NOT NULL,
3.     `author_id` int(11) DEFAULT NULL,
4.     `name` varchar(255) DEFAULT NULL,
5.     PRIMARY KEY (`bid`)
6. ) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4;
7.
8. INSERT INTO `blog` (`bid`, `author_id`, `name`) VALUES (1, 1001, 'MYSQL从入门到改
    行');

```

创建一个全新的maven 工程，命名为mebatis(注: 实际工程为：mybatis-custom的v1版本)，引入mysql的依赖。

```

1.     <dependency>
2.         <groupId>mysql</groupId>
3.         <artifactId>mysql-connector-java</artifactId>
4.         <version>5.1.21</version>
5.     </dependency>

```

我们要实现数据库记录转化成Java对象，先创建 Bean。

```

1. package cn.sitedev.mybatis.v1.mapper;
2.
3. import lombok.Data;
4.
5. import java.io.Serializable;
6.
7. @Data
8. public class Blog implements Serializable {
9.     Integer bid; // 文章ID
10.    String name; // 文章标题
11.    Integer authorId; // 文章作者ID
12. }

```

要操作数据库的Mapper接口：

```

1. package cn.sitedev.mybatis.v1.mapper;
2.
3. public interface BlogMapper {
4.     /**
5.      * 根据主键查询文章
6.      *
7.      * @param bid
8.      * @return
9.      */
10.    public Blog selectBlogById(Integer bid);
11.
12. }

```

## 2.1. SqlSession

我们已经分析了MeBatis的主要对象和操作流程，应该从哪里入手？

第一个需要的对象是SqlSession。所以我们从应用层的接口SqlSession入手。

那么我们先来创建一个package，叫mebatis。

创建一个自己的SqlSession，叫MySqlSession，我们暂时不需要考虑其他的实现，所以先不用创建接口，直接写类。

根据我们刚才总结的流程图，SqlSession 需要有一个获取代理对象的方法，那么这个代理对象是从哪里获取到的呢？是从我们的配置类里面获取到的，因为配置类里面有接口和它要产生的代理类的对应关系。

所以，我们要先持有一个Configuration对象，叫MyConfiguration，我们也创建这个类。除了获取代理对象之外，它里面还存储了我们的接口方法（也就是statementId）和SQL 语句的绑定关系。

第二个，我们在SqlSession中定义的对外的API，最后都会调用Executor去操作数据库，所以我们还要持有一个Executor对象，叫MyExecutor，我们也创建它。

```

1. public class MySqlSession {
2.     private MyConfiguration configuration;
3.
4.     private MyExecutor executor;
5.     ...

```

除了这两个属性之外，我们还要定义SqlSession的行为，也就是它的主要的方法。

第一个方法是查询方法，selectOne ()，由于它可以返回任意类型（List、Map、对象类型），我们把返回值定义成 `<T> T` 泛型。selectOne () 有两个参数，一个是String类型的statementId，我们会根据它找到SQL 语句。一个是Object类型的parameter参数（可以是Integer也可以是String等等，任意类型），用来填充SQL里面的占位符。

```

1. public class MySqlSession {
2.
3.     public <T> T selectOne(String statementId, Object paramater) {
4.         // 根据statementId拿到SQL
5.         String sql = statementId; // 先用statementId代替SQL
6.         return executor.query(sql, paramater);
7.     }
8.     ...

```

它会调用Executor的query () 方法，所以我们创建Executor类，传入这两个参数，一样返回一个泛型。Executor 里面要传入SQL，但是我们还没拿到，先用 statementId代替。

```

1. public class MyExecutor {
2.     public <T> T query(String sql, Object paramater) {
3.         return null;
4.     }

```

SqlSession的第二个方法是获取代理对象的方法，我们通过这种方式去避免了statementId的硬编码。

我们在SqlSession 中创建一个getMapper () 的方法，由于可以返回任意类型的代理类，所以我们把返回值也定义成泛型 `<T> T`。我们是根据接口类型获取到代理对象的，所以传入参数要用类型Class。

```

1. public class MySqlSession {
2.     public <T> T getMapper(Class clazz) {
3.         return null;
4.     }

```

## 2.2. Configuration

代理对象我们不是在SqlSession里面获取到的，要进一步调用Configuration的getMapper () 方法。返回值需要强转成 (T) 。

```

1. public class MySqlSession {
2.     public <T> T getMapper(Class clazz) {
3.         return configuration.getMapper(clazz, this);
4.     }

```

先在Configuration创建这个方法，返回类型一样是泛型，先返回空。

```

1. public class MyConfiguration {
2.
3.     public <T> T getMapper(Class clazz) {
4.         return null;
5.     }

```

## 2.3. MapperProxy

我们要在Configuration中通过 `getMapper()` 方法拿到这个代理对象，必须要有一个实现了 `InvocationHandler`的代理类（触发管理器）。我们来创建它： `MyMapperProxy`。

实现 `invoke()` 方法。

```
1. public class MyMapperProxy implements InvocationHandler {
2.
3.     @Override
4.     public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
5.         return null;
6.     }
```

`invoke()`的实现我们先留着。 `MapperProxy`已经有了，我们回到 `Configuration.getMapper()` 完成获取代理对象的逻辑。返回代理对象，直接使用JDK的动态代理：第一个参数是类加载器，第二个参数是被代理类实现的接口（这里没有被代理类），第三个参数是H（触发管理器）。

把返回结果强转为（T）：

```
1. public class MyConfiguration {
2.
3.     public <T> T getMapper(Class clazz, MySqlSession sqlSession) {
4.         return (T) Proxy.newProxyInstance(this.getClass().getClassLoader(), new
            Class[]{clazz},
5.             new MyMapperProxy());
6.     }
```

获取代理类的逻辑已经实现完了，我们可以在 `SqlSession` 中通过 `getMapper()` 拿到代理对象了，也就是可以调用 `invoke()`方法了。接下来去完成 `MapperProxy`的 `invoke()`方法。

在 `MapperProxy`的 `invoke()` 方法里面又调用了 `SqlSession`的 `selectOne()`方法。

一个问题出现了：在 `MapperProxy`里面根本没有 `SqlSession`对象？

这两个对象的关系怎么建立起来？ `MapperProxy`怎么拿到一个 `SqlSession`对象？

很简单，我们可通过构造函数传入它。

先定义一个属性，然后在 `MapperProxy`的构造函数里面赋值：

```
1. public class MyMapperProxy implements InvocationHandler {
2.     private MySqlSession sqlSession;
3.
4.     public MyMapperProxy(MySqlSession sqlSession){
5.         this.sqlSession = sqlSession;
6.     }
```



因为修改了代理类的构造函数，这个时候Configuration 创建代理类的方法getMapper()也要修改。

问题：Configuration 也没有SqlSession，没办法传入MapperProxy的构造函数。

怎么拿到SqlSession呢？是直接new一个吗？

不需要，可以在SqlSession调用它的时候直接把自己传进来（修改的地方：MapperProxy的构造函数添加了sqlSession， getMapper()方法也添加了SqlSession）：

```
1. public class MyConfiguration {
2.
3.     public <T> T getMapper(Class clazz, MySqlSession sqlSession) {
4.         return (T) Proxy.newProxyInstance(this.getClass().getClassLoader(), new
5.         Class[]{clazz},
6.         new MyMapperProxy(sqlSession));
7.     }
8. }
```

那么SqlSession的 getMapper () 方法也要修改（红色是修改的地方）：

问题：this 可以不传

```
1. public class MySqlSession {
2.
3.     public <T> T getMapper(Class clazz) {
4.         return configuration.getMapper(clazz, this);
5.     }
6. }
```

现在在MapperProxy 里面已经就可以拿到SqlSession对象了，在invoke () 方法里面我们会调用SqlSession的 selectOne()方法。我们继续来完成invoke () 方法。

selectOne () 方法有两个参数，statementId和paramater，这两个我们怎么拿到呢？

statementId其实就是接口的全路径+方法名，中间加一个英文的点。

paramater可以从方法参数中拿到 (args[0])。因为我们定义的是String，还要把拿到的Object强转一下。

把statementId 和 parameter传给 SqlSession

```
1. public class MyMapperProxy implements InvocationHandler {
2.
3.     @Override
4.     public Object invoke(Object proxy, Method method, Object[] args) throws
5.     Throwable {
6.         String mapperInterface = method.getDeclaringClass().getName();
7.         String methodName = method.getName();
8.         String statementId = mapperInterface + "." + methodName;
9.         return sqlSession.selectOne(statementId, args[0]);
10.    }
11. }
```

## 2.4. Executor

好，到了sqlSession的selectOne()方法，这里我们要去调用Executor的query方法，这个时候我们必须传入SQL 语句和parameter（根据statementId获取）。

问题来了，我们怎么根据 StatementId 找到我们要执行的SQL 语句呢？他们之间的绑定关系我们配置在哪里？

为了简便，免去读取文件流和解析XML标签的麻烦，我们把我们的SQL 语句放在Properties文件里面。

我们在resources目录下创建一个v1sql.properties文件。key就是接口全路径+方法名称，SQL是我们的查询SQL。

参数这里，因为我们要传入一个整数，所以先用一个%d的占位符代替：

```
1. cn.sitedev.mybatis.v1.mapper.BlogMapper.selectBlogById=select * from blog where  
   bid = %d
```

在sqlSession的selectOne () 方法里面，我们要根据StatementId获取到SQL，然后传给Executor。这个绑定关系是放在Configuration里面的。

怎么快速地解析 Properties文件？

为了避免重复解析，我们在Configuration 创建一个静态属性和静态方法，直接解析v1sql.properties文件里面的所有KV键值对：

```
1. public class MyConfiguration {  
2.     public static final ResourceBundle sqlMappings;  
3.  
4.     static {  
5.         sqlMappings = ResourceBundle.getBundle("v1sql");  
6.     }  
}
```

这样就可以通过Configuration拿到SQL了。

如果SQL语句拿不到，说明不存在映射关系（或者是系统方法），我们返回空。

```
1. public class MySqlSession {  
2.  
3.     public <T> T selectOne(String statementId, Object paramater) {  
4.         // 根据statementId拿到SQL  
5.         String sql = MyConfiguration.sqlMappings.getString(statementId);  
6.         if (null != sql && !"".equals(sql)) {  
7.             return executor.query(sql, paramater);  
8.         }  
9.         return null;  
10.    }
```

在SqlSession中，SQL语句已经拿到了，接下来就是Executor类的query（）方法，Executor是数据库操作的真正执行者。怎么做？

我们干脆直接把JDBC的代码全部复制过来，职责先不用细分。参数用传入的整形参数替换%d占位符，需要format一下。

最后我们把结果集强转一下。

```
1. public class MyExecutor {
2.     public <T> T query(String sql, Object paramater) {
3.         Connection conn = null;
4.         Statement stmt = null;
5.         Blog blog = new Blog();
6.
7.         try {
8.             // 注册 JDBC 驱动
9.             Class.forName("com.mysql.jdbc.Driver");
10.
11.            // 打开连接
12.            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis", "root",
"root");
13.
14.            // 执行查询
15.            stmt = conn.createStatement();
16.            ResultSet rs = stmt.executeQuery(String.format(sql, paramater));
17.
18.            // 获取结果集
19.            while (rs.next()) {
20.                Integer bid = rs.getInt("bid");
21.                String name = rs.getString("name");
22.                Integer authorId = rs.getInt("author_id");
23.                blog.setAuthorId(authorId);
24.                blog.setBid(bid);
25.                blog.setName(name);
26.            }
27.            System.out.println(blog);
28.
29.            rs.close();
30.            stmt.close();
31.            conn.close();
32.        } catch (SQLException se) {
33.            se.printStackTrace();
34.        } catch (Exception e) {
35.            e.printStackTrace();
36.        } finally {
37.            try {
38.                if (stmt != null) stmt.close();
39.            } catch (SQLException se2) {
40.            }
41.            try {
42.                if (conn != null) conn.close();
43.            } catch (SQLException se) {
44.                se.printStackTrace();
45.            }
46.        }
47.        return (T)blog;
```

```
48.     }
49. }
```

写一个测试类：

```
1. public class MyBatisBoot {
2.     public static void main(String[] args) {
3.         MySqlSession sqlSession = new MySqlSession();
4.         BlogMapper blogMapper = sqlSession.getMapper(BlogMapper.class);
5.         blogMapper.selectBlogById(1);
6.     }
7. }
```

执行测试类：



configuration是空的，忘记拿到Configuration了！那么Executor肯定也是空的咯。

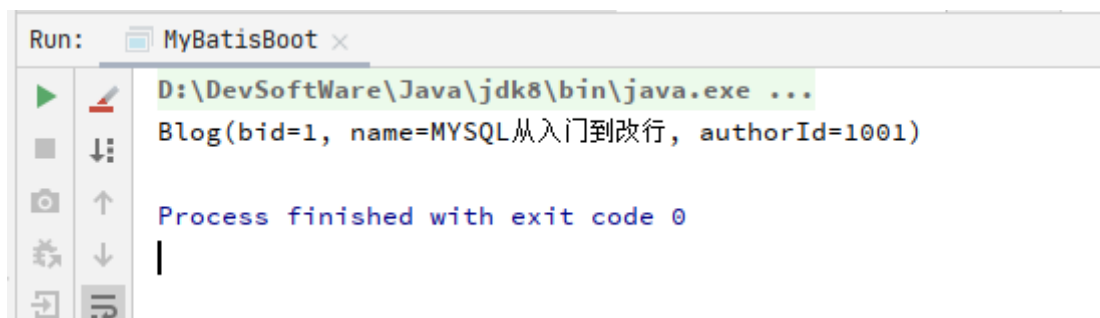
构造函数里面要给他们俩加上：

```
1. public class MySqlSession {
2.
3.     public MySqlSession(MyConfiguration configuration, MyExecutor executor) {
4.         this.configuration = configuration;
5.         this.executor = executor;
6.     }
7. }
```

改一下我们的测试类（SqlSession构造函数增加了两个参数）：

```
1. public class MyBatisBoot {
2.     public static void main(String[] args) {
3.         MySqlSession sqlSession = new MySqlSession(new MyConfiguration(), new
4.         MyExecutor());
5.         BlogMapper blogMapper = sqlSession.getMapper(BlogMapper.class);
6.         blogMapper.selectBlogById(1);
7.     }
8. }
```

测试通过，MeBatis1.0的版本完成了：



```
Run: MyBatisBoot x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
Blog(bid=1, name=MYSQL从入门到改行, authorId=1001)
Process finished with exit code 0
```

## 3. V1.0的不足

MeBatis1.0的功能完成了，在拿给老王看之前，我思考了一下：

### 3.1. 不足

- 1、在Executor中，对参数、语句和结果集的处理是耦合的，没有实现职责分离；
- 2、参数：没有实现对语句的预编译，只有简单的格式化（format），效率不高，还存在SQL注入的风险；
- 3、语句执行：数据库连接硬编码；
- 4、结果集：还只能处理Blog 类型，没有实现根据实体类自动映射。

确实有点搓，拿不出手。

### 3.2. 代码优化目标

对Executor的职责进行细化；

支持参数预编译；

支持结果集的自动处理（通过反射）。

### 3.3. 功能增强目标

在方法上使用注解配置SQL；

查询带缓存功能；

支持自定义插件。

## 4. V2.0的实现

工程：mybatis-custom的v2版本

### 4.1. 怎么解决数据库连接硬编码的问题？

在MyExecutor中，数据库连接采取了硬编码：

```

1. public class MyExecutor {
2.     public <T> T query(String sql, Object paramater) {
3.         ...
4.         try {
5.             // 注册 JDBC 驱动
6.             Class.forName("com.mysql.jdbc.Driver");
7.
8.             // 打开连接
9.             conn =
10.            DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatis", "root",
                "root");

```

抽取了全局配置文件mybatis.properties，存放SQL连接信息、缓存开关、插件地址、Mapper接口地址。

```

1. jdbc.driver=com.mysql.jdbc.Driver
2. jdbc.url=jdbc:mysql://localhost:3306/mybatis?
   useUnicode=true&characterEncoding=utf-8&rewriteBatchedStatements=true
3. jdbc.username=root
4. jdbc.password=root
5. cache.enabled=true
6. plugin.path=cn.sitedev.mybatis.v2.interceptor.MyPlugin
7. mapper.path=cn.sitedev.mybatis.v2.mapper

```

## 4.2. Executor的职责怎么拆分？

拆分成什么对象？做什么事情？

三个对象：ParameterHandler、StatementHandler、ResultSetHandler。

### 4.2.1. 参数处理ParameterHandler

首先是参数的处理，创建 ParameterHandler。

原来用的是Statement，不能对参数进行预编译，只能用字符串占位符的方式传参。

修改为PreparedStatement。

怎么处理参数？调用psmt的set方法；把传入的多个参数填充到SQL的 `?` 处。

```

1. package cn.sitedev.mybatis.v2.parameter;
2.
3. import java.sql.PreparedStatement;
4. import java.sql.SQLException;
5.
6. /**
7.  * 参数处理器
8.  */
9. public class ParameterHandler {
10.     private PreparedStatement psmt;
11.
12.     public ParameterHandler(PreparedStatement statement) {
13.         this.psmt = statement;
14.     }
15.
16.     /**
17.      * 从方法中获取参数，遍历设置SQL中的? 占位符
18.      *
19.      * @param parameters
20.      */
21.     public void setParameters(Object[] parameters) {
22.         try {
23.             // PreparedStatement的序号是从1开始的
24.             for (int i = 0; i < parameters.length; i++) {
25.                 int k = i + 1;
26.                 if (parameters[i] instanceof Integer) {
27.                     psmt.setInt(k, (Integer) parameters[i]);
28.                 } else if (parameters[i] instanceof Long) {
29.                     psmt.setLong(k, (Long) parameters[i]);
30.                 } else if (parameters[i] instanceof String) {
31.                     psmt.setString(k, String.valueOf(parameters[i]));
32.                 } else if (parameters[i] instanceof Boolean) {
33.                     psmt.setBoolean(k, (Boolean) parameters[i]);
34.                 } else {
35.                     psmt.setString(k, String.valueOf(parameters[i]));
36.                 }
37.             }
38.         } catch (SQLException e) {
39.             e.printStackTrace();
40.         }
41.     }
42. }

```

使用方式，在StatementHandler new出来使用（Ctrl+点击方法可以查看方法使用的地方）：



```
1. public class StatementHandler {
2.     private ResultSetHandler resultSetHandler = new ResultSetHandler();
3.
4.     public <T> T query(String statement, Object[] parameter, Class pojo) {
5.         ...
6.         try {
7.             conn = getConnection();
8.             preparedStatement = conn.prepareStatement(statement);
9.             ParameterHandler parameterHandler = new
ParameterHandler(preparedStatement);
10.            parameterHandler.setParameters(parameter);
11.            ...
        }
```

### 4.2.2. 结果集处理ResultSetHandler

调用RS的get方法，调用bean的setter。

创建 ResultSetHandler，在其中创建对象，获取ResultSet值，通过反射给实体类对象赋值。

(结果集转换的目标类型是在创建 MapperProxy的时候指定的)

```
1. package cn.sitedev.mybatis.v2.executor;
2.
3. import java.lang.reflect.Field;
4. import java.lang.reflect.Method;
5. import java.sql.ResultSet;
6. import java.sql.SQLException;
7.
8. /**
9.  * 结果集处理器
10. */
11. public class ResultSetHandler {
12.
13.     /**
14.      * @param resultSet 结果集
15.      * @param type      需要转换的目标类型
16.      * @param <T>
17.      * @return
18.      */
19.     public <T> T handle(ResultSet resultSet, Class type) {
20.         // 直接调用Class的newInstance方法产生一个实例
21.         Object pojo = null;
22.         try {
23.             pojo = type.newInstance();
24.             // 遍历结果集
25.             if (resultSet.next()) {
26.                 // 循环赋值
27.                 for (Field field : pojo.getClass().getDeclaredFields()) {
28.                     setValue(pojo, field, resultSet);
29.                 }
30.             }
31.         } catch (Exception e) {
32.             e.printStackTrace();
33.         }
34.
35.         return (T) pojo;
36.     }
37.
38.     /**
39.      * 通过反射给属性赋值
40.      */
41.     private void setValue(Object pojo, Field field, ResultSet rs) {
42.         try {
43.             // 获取 pojo 的 set 方法
44.             Method setMethod = pojo.getClass().getMethod("set" +
firstWordCapital(field.getName()), field.getType());
45.             // 调用 pojo 的set 方法，使用结果集给属性赋值
46.             // 赋值先从resultSet取出值
47.             setMethod.invoke(pojo, getResult(rs, field));
48.         } catch (Exception e) {
```

```

49.         e.printStackTrace();
50.     }
51. }
52.
53. /**
54.  * 根据反射判断类型，从ResultSet中取对应类型参数
55.  */
56. private Object getResult(ResultSet rs, Field field) throws SQLException {
57.     //TODO TypeHandler
58.     Class type = field.getType();
59.     String dataName = HumpToUnderline(field.getName()); // 驼峰转下划线
60.     // TODO 类型判断不够全
61.     if (Integer.class == type) {
62.         return rs.getInt(dataName);
63.     } else if (String.class == type) {
64.         return rs.getString(dataName);
65.     } else if (Long.class == type) {
66.         return rs.getLong(dataName);
67.     } else if (Boolean.class == type) {
68.         return rs.getBoolean(dataName);
69.     } else if (Double.class == type) {
70.         return rs.getDouble(dataName);
71.     } else {
72.         return rs.getString(dataName);
73.     }
74. }
75.
76. // 数据库下划线转Java驼峰命名
77. public static String HumpToUnderline(String para) {
78.     StringBuilder sb = new StringBuilder(para);
79.     int temp = 0;
80.     if (!para.contains("_")) {
81.         for (int i = 0; i < para.length(); i++) {
82.             if (Character.isUpperCase(para.charAt(i))) {
83.                 sb.insert(i + temp, "_");
84.                 temp += 1;
85.             }
86.         }
87.     }
88.     return sb.toString().toUpperCase();
89. }
90.
91. /**
92.  * 单词首字母大写
93.  */
94. private String firstWordCapital(String word) {
95.     String first = word.substring(0, 1);
96.     String tail = word.substring(1);
97.     return first.toUpperCase() + tail;

```

```
98.     }
99. }
```

调用方式，StatementHandler中，new出来使用，传入结果集和pojo类型：

```
1. public class StatementHandler {
2.     private ResultSetHandler resultSetHandler = new ResultSetHandler();
3.
4.     public <T> T query(String statement, Object[] parameter, Class pojo) {
5.         ...
6.         try {
7.             result = resultSetHandler.handle(preparedStatement.getResultSet(),
pojo);
8.             ...

```

### 4.2.3. 语句执行处理StatementHandler

创建 StatementHandler，在Executor中调用。

其中包括了：封装获取连接的方法、处理参数的ParameterHandler，执行查询的PreparedStatement，处理结果集的ResultSetHandler，起到了承前启后的作用。

```
1. package cn.sitedev.mybatis.v2.executor;
2.
3. import cn.sitedev.mybatis.v2.parameter.ParameterHandler;
4. import cn.sitedev.mybatis.v2.session.Configuration;
5.
6. import java.sql.Connection;
7. import java.sql.DriverManager;
8. import java.sql.PreparedStatement;
9. import java.sql.SQLException;
10.
11. /**
12.  * 封装JDBC Statement，用于操作数据库
13.  */
14. public class StatementHandler {
15.     private ResultSetHandler resultSetHandler = new ResultSetHandler();
16.
17.     public <T> T query(String statement, Object[] parameter, Class pojo) {
18.         Connection conn = null;
19.         PreparedStatement preparedStatement = null;
20.         Object result = null;
21.
22.         try {
23.             conn = getConnection();
24.             preparedStatement = conn.prepareStatement(statement);
25.             ParameterHandler parameterHandler = new
26. ParameterHandler(preparedStatement);
27.             parameterHandler.setParameters(parameter);
28.             preparedStatement.execute();
29.             try {
30.                 result = resultSetHandler.handle(preparedStatement.getResultSet(),
31. pojo);
32.             } catch (Exception e) {
33.                 e.printStackTrace();
34.             }
35.             return (T) result;
36.         } catch (Exception e) {
37.             e.printStackTrace();
38.         } finally {
39.             if (conn != null) {
40.                 try {
41.                     conn.close();
42.                 } catch (SQLException e) {
43.                     e.printStackTrace();
44.                 }
45.             }
46.             conn = null;
47.         }
48.     }
49.     // 只在try里面return会报错
50.     return null;
51. }
```

```
48.     }
49.
50.     /**
51.      * 获取连接
52.      *
53.      * @return
54.      * @throws SQLException
55.      */
56.     private Connection getConnection() {
57.         String driver = Configuration.properties.getString("jdbc.driver");
58.         String url = Configuration.properties.getString("jdbc.url");
59.         String username = Configuration.properties.getString("jdbc.username");
60.         String password = Configuration.properties.getString("jdbc.password");
61.         Connection conn = null;
62.         try {
63.             Class.forName(driver);
64.             conn = DriverManager.getConnection(url, username, password);
65.         } catch (ClassNotFoundException e) {
66.             e.printStackTrace();
67.         } catch (SQLException e) {
68.             e.printStackTrace();
69.         }
70.         return conn;
71.     }
72. }
```

## 4.3. Configuration和其他代码的细化

创建：MapperProxyFactory，用来创建代理对象。

```
1. package cn.sitedev.mybatis.v2.binding;
2.
3. import cn.sitedev.mybatis.v2.session.DefaultSqlSession;
4.
5. import java.lang.reflect.Proxy;
6.
7. /**
8.  * 用于产生MapperProxy代理类
9.  *
10.  * @param <T>
11.  */
12. public class MapperProxyFactory<T> {
13.     private Class<T> mapperInterface;
14.     private Class object;
15.
16.     public MapperProxyFactory(Class<T> mapperInterface, Class object) {
17.         this.mapperInterface = mapperInterface;
18.         this.object = object;
19.     }
20.
21.     public T newInstance(DefaultSqlSession sqlSession) {
22.         return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(), new
23.             Class[]{mapperInterface},
24.             new MapperProxy(sqlSession, object));
25.     }
26. }
```

创建：MapperRegistry，维护接口和工厂类映射关系。

```

1. package cn.sitedev.mybatis.v2.binding;
2.
3. import cn.sitedev.mybatis.v2.session.DefaultSqlSession;
4.
5. import java.util.HashMap;
6. import java.util.Map;
7.
8. /**
9.  * 维护接口和工厂类的关系，用于获取MapperProxy代理对象
10.  * 工厂类指定了POJO类型，用于处理结果集返回
11.  */
12. public class MapperRegistry {
13.
14.     // 接口和工厂类映射关系
15.     private final Map<Class<?>, MapperProxyFactory> knownMappers = new HashMap<>
16.     ();
17.
18.     /**
19.      * 在Configuration中解析接口上的注解时，存入接口和工厂类的映射关系
20.      * 此处传入pojo类型，是为了最终处理结果集的时候将结果转换为POJO类型
21.      *
22.      * @param clazz
23.      * @param pojo
24.      * @param <T>
25.      */
26.     public <T> void addMapper(Class<T> clazz, Class pojo) {
27.         knownMappers.put(clazz, new MapperProxyFactory(clazz, pojo));
28.     }
29.
30.     /**
31.      * 创建一个代理对象
32.      */
33.     public <T> T getMapper(Class<T> clazz, DefaultSqlSession sqlSession) {
34.         MapperProxyFactory proxyFactory = knownMappers.get(clazz);
35.         if (proxyFactory == null) {
36.             throw new RuntimeException("Type: " + clazz + " can not find");
37.         }
38.         return (T) proxyFactory.newInstance(sqlSession);
39.     }
40. }

```

创建：SqlSessionFactory，用来创建 SqlSession。



```

1. package cn.sitedev.mybatis.v2.session;
2.
3. /**
4.  * 会话工厂类，用于解析配置文件，产生SqlSession
5.  */
6. public class SqlSessionFactory {
7.
8.     private Configuration configuration;
9.
10.    /**
11.     * build方法用于初始化Configuration，解析配置文件的工作在Configuration的构造函数中
12.     *
13.     * @return
14.     */
15.    public SqlSessionFactory build() {
16.        configuration = new Configuration();
17.        return this;
18.    }
19.
20.    /**
21.     * 获取DefaultSqlSession
22.     *
23.     * @return
24.     */
25.    public DefaultSqlSession openSqlSession() {
26.        return new DefaultSqlSession(configuration);
27.    }
28. }

```

定义了Executor接口和基本实现SimpleExecutor。

```

1. package cn.sitedev.mybatis.v2.executor;
2.
3. public interface Executor {
4.     <T> T query(String statement, Object[] parameter, Class pojo);
5. }
6. //////////////////////////////////
7. package cn.sitedev.mybatis.v2.executor;
8.
9. public class SimpleExecutor implements Executor {
10.     @Override
11.     public <T> T query(String statement, Object[] parameter, Class pojo) {
12.         StatementHandler statementHandler = new StatementHandler();
13.         return statementHandler.query(statement, parameter, pojo);
14.     }
15. }

```

## 4.4. 支持注解

定义了一个类的注解@Entity。如果有@Entity 注解，说明是查询数据库的接口。

```
1. package cn.sitedev.mybatis.v2.annotation;
2.
3. import java.lang.annotation.*;
4.
5. /**
6.  * 用于注解接口，以映射返回的实体类
7.  */
8. @Documented
9. @Retention(RetentionPolicy.RUNTIME)
10. @Target(ElementType.TYPE)
11. public @interface Entity {
12.     Class<?> value();
13. }
```

定义了一个@Select 注解，加在方法上。@Select里面可以写SQL语句。

```
1. package cn.sitedev.mybatis.v2.annotation;
2.
3. import java.lang.annotation.*;
4.
5. /**
6.  * 注解方法，配置SQL语句
7.  */
8. @Documented
9. @Retention(RetentionPolicy.RUNTIME)
10. @Target(ElementType.METHOD)
11. public @interface Select {
12.     String value();
13. }
```

注解在Configuration构造函数中的parsingClass()中解析，保存在MapperRegistry中。

```

1. public class Configuration {
2.     /**
3.      * 初始化时解析全局配置文件
4.      */
5.     public Configuration() {
6.         ...
7.         for (Class<?> mapper : mapperList) {
8.             parsingClass(mapper);
9.         }
10.        ...
11.    }
12.
13.    /**
14.     * 解析Mapper接口上配置的注解（SQL语句）
15.     */
16.    private void parsingClass(Class<?> mapper) {
17.        // 1.解析类上的注解
18.        // 如果有Entity注解，说明是查询数据库的接口
19.        if (mapper.isAnnotationPresent(Entity.class)) {
20.            for (Annotation annotation : mapper.getAnnotations()) {
21.                if (annotation.annotationType().equals(Entity.class)) {
22.                    // 注册接口与实体类的映射关系
23.                    MAPPER_REGISTRY.addMapper(mapper, ((Entity)
24.                        annotation).value());
25.                }
26.            }
27.
28.            // 2.解析方法上的注解
29.            Method[] methods = mapper.getMethods();
30.            for (Method method : methods) {
31.                //TODO 其他操作
32.                // 解析@Select注解的SQL语句
33.                if (method.isAnnotationPresent(Select.class)) {
34.                    for (Annotation annotation : method.getDeclaredAnnotations()) {
35.                        if (annotation.annotationType().equals(Select.class)) {
36.                            // 注册接口类型+方法名和SQL语句的映射关系
37.                            String statement = method.getDeclaringClass().getName() +
38.                                "." + method.getName();
39.                            mappedStatements.put(statement, ((Select)
40.                                annotation).value());
41.                        }
42.                    }
43.                }
44.            }
45.        }
46.    }
47.    ...

```

注意在properties中和注解上同时配置SQL 会覆盖。

properties中对表达三个对象的映射关系并不适合，所以暂时用--分隔。注意类型前面不能有空格。

```
1. public class Configuration {
2.     ...
3.
4.     /**
5.      * 初始化时解析全局配置文件
6.      */
7.     public Configuration() {
8.         // Note: 在properties和注解中重复配置SQL会覆盖
9.         // 1.解析sql.properties
10.        for (String key : sqlMappings.keySet()) {
11.            Class mapper = null;
12.            String statement = null;
13.            String pojoStr = null;
14.            Class pojo = null;
15.            // properties中的value用--隔开，第一个是SQL语句
16.            statement = sqlMappings.getString(key).split("--")[0];
17.            // properties中的value用--隔开，第二个是需要转换的POJO类型
18.            pojoStr = sqlMappings.getString(key).split("--")[1];
19.            try {
20.                // properties中的key是接口类型+方法
21.                // 从接口类型+方法中截取接口类型
22.                mapper = Class.forName(key.substring(0, key.lastIndexOf(".")));
23.                pojo = Class.forName(pojoStr);
24.                ...
            }
        }
    }
}
```

```
1. # 参见配置文件v2sql.properties
2. cn.sitedev.mybatis.v2.mapper.BlogMapper.selectBlogById=select * from blog where
   bid = ? --cn.sitedev.mybatis.v2.mapper.Blog
```

## 4.5. 支持查询缓存CachingExecutor

当全局配置中的 cacheEnabled=true时，Configuration的newExecutor()方法会对SimpleExecutor 进行装饰，返回被代理过的Executor。

```
1. public class Configuration {
2.     ...
3.
4.     /**
5.      * 创建执行器，当开启缓存时使用缓存装饰
6.      * 当配置插件时，使用插件代理
7.      *
8.      * @return
9.      */
10.    public Executor newExecutor() {
11.        Executor executor = null;
12.        if (properties.getString("cache.enabled").equals("true")) {
13.            executor = new CachingExecutor(new SimpleExecutor());
14.        } else {
15.            executor = new SimpleExecutor();
16.        }
17.        ...
    }
```

定义了一个CachingExecutor。

```
1. package cn.sitedev.mybatis.v2.executor;
2.
3. import cn.sitedev.mybatis.v2.cache.CacheKey;
4.
5. import java.util.HashMap;
6. import java.util.Map;
7.
8. /**
9.  * 带缓存的执行器，用于装饰基本执行器
10. */
11. public class CachingExecutor implements Executor {
12.
13.     private Executor delegate;
14.     private static final Map<Integer, Object> cache = new HashMap<>();
15.
16.     public CachingExecutor(Executor delegate) {
17.         this.delegate = delegate;
18.     }
19.
20.     @Override
21.     public <T> T query(String statement, Object[] parameter, Class pojo) {
22.         // 计算CacheKey
23.         CacheKey cacheKey = new CacheKey();
24.         cacheKey.update(statement);
25.         cacheKey.update(joinStr(parameter));
26.         // 是否拿到缓存
27.         if (cache.containsKey(cacheKey.getCode())) {
28.             // 命中缓存
29.             System.out.println("【命中缓存】");
30.             return (T) cache.get(cacheKey.getCode());
31.         } else {
32.             // 没有的话调用被装饰的SimpleExecutor从数据库查询
33.             Object obj = delegate.query(statement, parameter, pojo);
34.             cache.put(cacheKey.getCode(), obj);
35.             return (T) obj;
36.         }
37.     }
38.
39.     // 为了命中缓存，把Object[]转换成逗号拼接的字符串，因为对象的HashCode都不一样
40.     public String joinStr(Object[] objs) {
41.         StringBuffer sb = new StringBuffer();
42.         if (objs != null && objs.length > 0) {
43.             for (Object objStr : objs) {
44.                 sb.append(objStr.toString() + ",");
45.             }
46.         }
47.         int len = sb.length();
48.         if (len > 0) {
49.             sb.deleteCharAt(len - 1);
```

```
50.     }
51.     return sb.toString();
52. }
53. }
```

在DefaultSqlSession调用Executor时，会先走到CachingExecutor。

```
1.  public class DefaultSqlSession {
2.      private Configuration configuration;
3.      private Executor executor;
4.
5.      public DefaultSqlSession(Configuration configuration) {
6.          this.configuration = configuration;
7.          // 根据全局配置决定是否使用缓存装饰
8.          this.executor = configuration.newExecutor();
9.      }
10.
11.      ...
12.
13.      public <T> T selectOne(String statement, Object[] parameter, Class pojo) {
14.          String sql = getConfiguration().getMappedStatement(statement);
15.          // 打印代理对象时会自动调用toString()方法，触发invoke()
16.          return executor.query(sql, parameter, pojo);
17.      }
```

定义了一个CacheKey 用于计算缓存Key。

```

1. package cn.sitedev.mybatis.v2.cache;
2.
3. /**
4.  * 缓存Key
5.  */
6. public class CacheKey {
7.     // MyBatis抄袭了我的设计
8.     private static final int DEFAULT_HASHCODE = 17; // 默认哈希值
9.     private static final int DEFAULT_MULTIPLIER = 37; // 倍数
10.
11.     private int hashCode;
12.     private int count;
13.     private int multiplier;
14.
15.     /**
16.      * 构造函数
17.      */
18.     public CacheKey() {
19.         this.hashCode = DEFAULT_HASHCODE;
20.         this.count = 0;
21.         this.multiplier = DEFAULT_MULTIPLIER;
22.     }
23.
24.     /**
25.      * 返回CacheKey的值
26.      *
27.      * @return
28.      */
29.     public int getCode() {
30.         return hashCode;
31.     }
32.
33.     /**
34.      * 计算CacheKey中的HashCode
35.      *
36.      * @param object
37.      */
38.     public void update(Object object) {
39.         int baseHashCode = object == null ? 1 : object.hashCode();
40.         count++;
41.         baseHashCode *= count;
42.         hashCode = multiplier * hashCode + baseHashCode;
43.     }
44. }

```

## 4.6. 支持插件

插件需要几个必要的类：



Interceptor 接口规范插件格式;

@Intercepts 注解指定拦截的对象和方法;

InterceptorChain 容纳解析的插件类;

Plugin 可以产生代理对象, 也是触发管理器;

Invocation 包装类, 用来调用被拦截对象的方法。

#### 4.6.1. 定义Interceptor接口

```
1. package cn.sitedev.mybatis.v2.plugin;
2.
3. /**
4.  * 拦截器接口, 所有自定义拦截器必须实现此接口
5.  */
6. public interface Interceptor {
7.     /**
8.      * 插件的核心逻辑实现
9.      *
10.     * @param invocation
11.     * @return
12.     * @throws Throwable
13.     */
14.     Object intercept(Invocation invocation) throws Throwable;
15.
16.     /**
17.      * 对被拦截对象进行代理
18.      *
19.     * @param target
20.     * @return
21.     */
22.     Object plugin(Object target);
23. }
```

#### 4.6.2. 定义InterceptorChain容器

```
1. package cn.sitedev.mybatis.v2.plugin;
2.
3. import java.util.ArrayList;
4. import java.util.List;
5.
6. /**
7.  * 拦截器链，存放所有拦截器，和对代理对象进行循环代理
8.  */
9. public class InterceptorChain {
10.
11.     private final List<Interceptor> interceptors = new ArrayList<>();
12.
13.     public void addInterceptor(Interceptor interceptor) {
14.         interceptors.add(interceptor);
15.     }
16.
17.     /**
18.      * 对被拦截对象进行层层代理
19.      *
20.      * @param target
21.      * @return
22.      */
23.     public Object pluginAll(Object target) {
24.         for (Interceptor interceptor : interceptors) {
25.             target = interceptor.plugin(target);
26.         }
27.         return target;
28.     }
29.
30.     public boolean hasPlugin() {
31.         if (interceptors.size() == 0) {
32.             return false;
33.         }
34.         return true;
35.     }
36. }
```

### 4.6.3. 定义Plugin代理类

```
1. package cn.sitedev.mybatis.v2.plugin;
2.
3. import cn.sitedev.mybatis.v2.annotation.Intercepts;
4.
5. import java.lang.reflect.InvocationHandler;
6. import java.lang.reflect.Method;
7. import java.lang.reflect.Proxy;
8.
9. /**
10.  * 代理类，用于代理被拦截对象
11.  * 同时提供了创建代理类的方法
12.  */
13. public class Plugin implements InvocationHandler {
14.     private Object target;
15.     private Interceptor interceptor;
16.
17.     /**
18.      * @param target      被代理对象
19.      * @param interceptor 拦截器（插件）
20.      */
21.     public Plugin(Object target, Interceptor interceptor) {
22.         this.target = target;
23.         this.interceptor = interceptor;
24.     }
25.
26.     /**
27.      * 对被代理对象进行代理，返回代理类
28.      *
29.      * @param obj
30.      * @param interceptor
31.      * @return
32.      */
33.     public static Object wrap(Object obj, Interceptor interceptor) {
34.         Class clazz = obj.getClass();
35.         return Proxy.newProxyInstance(clazz.getClassLoader(),
36.             clazz.getInterfaces(), new Plugin(obj, interceptor));
37.     }
38.
39.     /**
40.      * @param proxy
41.      * @param method
42.      * @param args
43.      * @return
44.      * @throws Throwable
45.      */
46.     @Override
47.     public Object invoke(Object proxy, Method method, Object[] args) throws
48.         Throwable {
49.         // 自定义的插件上有@Intercepts注解，指定了拦截的方法
```

```

48.         if (interceptor.getClass().isAnnotationPresent(Intercepts.class)) {
49.             // 如果是被拦截的方法，则进入自定义拦截器的逻辑
50.             if
51. (method.getName().equals(interceptor.getClass().getAnnotation(Intercepts.class).value())) {
52.                 return interceptor.intercept(new Invocation(target, method,
53. args));
54.             }
55.             // 非被拦截方法，执行原逻辑
56.             return method.invoke(target, method, args);
57.         }

```

#### 4.6.4. 定义Invocation包装类

```

1. package cn.sitedev.mybatis.v2.plugin;
2.
3. import lombok.Data;
4.
5. import java.lang.reflect.InvocationTargetException;
6. import java.lang.reflect.Method;
7.
8. /**
9.  * 包装类，对被代理对象进行包装
10. */
11. @Data
12. @AllArgsConstructor
13. public class Invocation {
14.     private Object target;
15.     private Method method;
16.     private Object[] args;
17.
18.     public Object proceed() throws InvocationTargetException,
19. IllegalAccessException {
20.         return method.invoke(target, args);
21.     }

```

在Configuration的newExecutor () 中对插件进行解析，对被拦截对象进行代理，返回被插件代理类代理过的Executor。

```
1. public class Configuration {
2.     ...
3.
4.     /**
5.      * 创建执行器，当开启缓存时使用缓存装饰
6.      * 当配置插件时，使用插件代理
7.      *
8.      * @return
9.      */
10.    public Executor newExecutor() {
11.        ...
12.        // 目前只拦截了Executor，所有的插件都对Executor进行代理，没有对拦截类和方法签名进行判断
13.        if (interceptorChain.hasPlugin()) {
14.            return (Executor) interceptorChain.pluginAll(executor);
15.        }
16.        return executor;
17.    }
```

#### 4.6.5. 自定义插件

```
1. package cn.sitedev.mybatis.v2.interceptor;
2.
3. import cn.sitedev.mybatis.v2.annotation.Intercepts;
4. import cn.sitedev.mybatis.v2.plugin.Interceptor;
5. import cn.sitedev.mybatis.v2.plugin.Invocation;
6. import cn.sitedev.mybatis.v2.plugin.Plugin;
7.
8. import java.util.Arrays;
9.
10. /**
11.  * 自定义插件
12.  */
13. @Intercepts("query")
14. public class MyPlugin implements Interceptor {
15.     @Override
16.     public Object intercept(Invocation invocation) throws Throwable {
17.         String statement = (String) invocation.getArgs()[0];
18.         Object[] parameter = (Object[]) invocation.getArgs()[1];
19.         Class pojo = (Class) invocation.getArgs()[2];
20.         System.out.println("进入自定义插件: MyPlugin");
21.         System.out.println("SQL: [" + statement + "]");
22.         System.out.println("Parameters: " + Arrays.toString(parameter));
23.
24.         return invocation.proceed();
25.     }
26.
27.     @Override
28.     public Object plugin(Object target) {
29.         return Plugin.wrap(target, this);
30.     }
31. }
```

## 5. 工作流程分析

### 5.1. 启动解析

SqlSessionFactory的build () 方法，调用了Configuration的构造函数进行解析。

```

1. public class SqlSessionFactory {
2.
3.     private Configuration configuration;
4.
5.     /**
6.      * build方法用于初始化Configuration，解析配置文件的工作在Configuration的构造函数
7.      *
8.      * @return
9.      */
10.    public SqlSessionFactory build() {
11.        configuration = new Configuration();
12.        return this;
13.    }
14.    ...
15. ``
16.
17. 静态代码块解析 Properties文件。
18.
19.
20. ```java
21. public class Configuration {
22.     static {
23.         sqlMappings = ResourceBundle.getBundle("v2sql");
24.         properties = ResourceBundle.getBundle("mybatis");
25.     }
26.     ...

```

首先解析解析 sql.properties，放到mappedStatements中，把接口和工厂类也绑定起来。

然后解析Mapper 接口上的注解，不能重复配置。

最后解析插件，添加到interceptorChain中。

```

1. public class Configuration {
2.     ...
3.
4.     /**
5.      * 初始化时解析全局配置文件
6.      */
7.     public Configuration() {
8.         // Note: 在properties和注解中重复配置SQL会覆盖
9.         // 1.解析sql.properties
10.        for (String key : sqlMappings.keySet()) {
11.            Class mapper = null;
12.            String statement = null;
13.            String pojoStr = null;
14.            Class pojo = null;
15.            // properties中的value用--隔开，第一个是SQL语句
16.            statement = sqlMappings.getString(key).split("--")[0];
17.            // properties中的value用--隔开，第二个是需要转换的POJO类型
18.            pojoStr = sqlMappings.getString(key).split("--")[1];
19.            try {
20.                // properties中的key是接口类型+方法
21.                // 从接口类型+方法中截取接口类型
22.                mapper = Class.forName(key.substring(0, key.lastIndexOf(".")));
23.                pojo = Class.forName(pojoStr);
24.            } catch (ClassNotFoundException e) {
25.                e.printStackTrace();
26.            }
27.
28.            MAPPER_REGISTRY.addMapper(mapper, pojo); // 接口与返回的实体类关系
29.            mappedStatements.put(key, statement); // 接口方法与SQL关系
30.        }
31.
32.        // 2.解析接口上的注解（会覆盖XML中的接口与实体类的关系）
33.        String mapperPath = properties.getString("mapper.path");
34.        scanPackage(mapperPath);
35.        for (Class<?> mapper : mapperList) {
36.            parsingClass(mapper);
37.        }
38.        // 3.解析插件，可配置多个插件
39.        String pluginPathValue = properties.getString("plugin.path");
40.        String[] pluginPaths = pluginPathValue.split(",");
41.        if (pluginPaths != null) {
42.            // 将插件添加到interceptorChain中
43.            for (String plugin : pluginPaths) {
44.                Interceptor interceptor = null;
45.                try {
46.                    interceptor = (Interceptor)
Class.forName(plugin).newInstance();
47.                } catch (Exception e) {
48.                    e.printStackTrace();

```



```
49.         }
50.         interceptorChain.addInterceptor(interceptor);
51.     }
52. }
53. }
54. ...
```

## 5.2. 获取SqlSession

在这里面也创建了一个executor。

```
1. public class DefaultSqlSession {
2.     private Configuration configuration;
3.     private Executor executor;
4.
5.     public DefaultSqlSession(Configuration configuration) {
6.         this.configuration = configuration;
7.         // 根据全局配置决定是否使用缓存装饰
8.         this.executor = configuration.newExecutor();
9.     }
10. ...
```

如果开启了缓存，用CachingExecutor对SimpleExecutor进行装饰。

如果配置了插件，对Executor创建代理。

```
1. public class Configuration {
2.     ...
3.
4.     /**
5.      * 创建执行器，当开启缓存时使用缓存装饰
6.      * 当配置插件时，使用插件代理
7.      *
8.      * @return
9.      */
10.    public Executor newExecutor() {
11.        Executor executor = null;
12.        if (properties.getString("cache.enabled").equals("true")) {
13.            executor = new CachingExecutor(new SimpleExecutor());
14.        } else {
15.            executor = new SimpleExecutor();
16.        }
17.
18.        // 目前只拦截了Executor，所有的插件都对Executor进行代理，没有对拦截类和方法签名进行判断
19.        if (interceptorChain.hasPlugin()) {
20.            return (Executor) interceptorChain.pluginAll(executor);
21.        }
22.        return executor;
23.    }
24.    ...
```

## 5.3. Mapper接口调用

```

1. package cn.sitedev.mybatis.v2;
2.
3. import cn.sitedev.mybatis.v2.mapper.Blog;
4. import cn.sitedev.mybatis.v2.mapper.BlogMapper;
5. import cn.sitedev.mybatis.v2.session.DefaultSqlSession;
6. import cn.sitedev.mybatis.v2.session.SqlSessionFactory;
7.
8. public class TestMybatis {
9.
10.     public static void main(String[] args) {
11.         SqlSessionFactory factory = new SqlSessionFactory();
12.         DefaultSqlSession sqlSession = factory.build().openSqlSession();
13.         // 获取MapperProxy代理
14.         BlogMapper mapper = sqlSession.getMapper(BlogMapper.class);
15.         Blog blog = mapper.selectBlogById(1);
16.
17.         System.out.println("第一次查询: " + blog);
18.         System.out.println();
19.         blog = mapper.selectBlogById(1);
20.         System.out.println("第二次查询: " + blog);
21.     }
22. }

```



```

Run: TestMybatis x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
进入自定义插件: MyPlugin
SQL: [select * from blog where bid = ?]
Parameters: [1]
第一次查询: Blog(bid=1, name=MYSQL从入门到改行, authorId=1001)

进入自定义插件: MyPlugin
SQL: [select * from blog where bid = ?]
Parameters: [1]
【命中缓存】
第二次查询: Blog(bid=1, name=MYSQL从入门到改行, authorId=1001)

Process finished with exit code 0
|

```

因为返回的是一个代理对象，所以会先走到MapperProxy的invoke()方法。

```

1.  /**
2.   * MapperProxy代理类，用于代理Mapper接口
3.   */
4.  public class MapperProxy implements InvocationHandler {
5.
6.      private DefaultSqlSession sqlSession;
7.      private Class object;
8.
9.      public MapperProxy(DefaultSqlSession sqlSession, Class object) {
10.         this.sqlSession = sqlSession;
11.         this.object = object;
12.     }
13.
14.     /**
15.      * 所有Mapper接口的方法调用都会走到这里
16.      *
17.      * @param proxy
18.      * @param method
19.      * @param args
20.      * @return
21.      * @throws Throwable
22.      */
23.     @Override
24.     public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
25.         String mapperInterface = method.getDeclaringClass().getName();
26.         String methodName = method.getName();
27.         String statementId = mapperInterface + "." + methodName;
28.         // 如果根据接口类型+方法名能找到映射的SQL，则执行SQL
29.         if (sqlSession.getConfiguration().hasStatement(statementId)) {
30.             return sqlSession.selectOne(statementId, args, object);
31.         }
32.         // 否则直接执行被代理对象的原方法
33.         return method.invoke(proxy, args);
34.     }
35. }

```

它会根据把接口类型和方法名组成 statementId，传给SqlSession。SqlSession里面会从 Configuration 中拿到SQL，传给Executor。Executor会调用 StatementHandler执行，这个里面又包括了ParameterHandler、PreparedStatement、ResultSetHandler。

## 6. V2.0的升级

老王看了mebatis的代码以后，说：牛逼牛逼，提了一些建议：

- 不能返回List、Map；
- TypeHandler 只能处理部分类型，如果能够处理所有类型的转换关系，和自定义类型就好了。

- 缓存只有一级，只有一个全局开关，不能在单个方法上关闭（配置不灵活，properties不够用了）；
- 插入、删除、修改的注解；
- 插件对其他对象、指定方法的拦截，插件支持参数配置；
- 细节考虑不足，异常处理有点粗暴；
- ...