04.通过手写MyBatis带你掌握自己写框架的秘诀

课程目标

1.通过手写MyBatis1.0版本加深对MyBatis的理解
2.通过手写MyBatis2.0版本体验框架的演进过程
3.通过手写MyBatis2.0版本理解MyBatis的设计思想与细节
4.掌握手写框架的秘诀

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为什么这么设计的同学。

一、手写MyBatis v1.0

1. 需求分析

1.1 项目需求

通过原始的JDBC代码来操作数据库非常的麻烦，里面存在着太多的重复代码和低下的开发效率，针对这种情况我们需要提供一个更加高效的持久层框架。

1.2 核心功能

首先来看下JDBC操作查询的代码。

```
    /**
     *
     * 通过JDBC查询用户信息
     */
    public void queryUser(){
        Connection conn = null;
        Statement stmt = null;
        User user = new User();

        try {
            // 注册 JDBC 驱动
            // Class.forName("com.mysql.cj.jdbc.Driver");

            // 打开连接
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatisdb?
characterEncoding=utf-8&serverTimezone=UTC", "root", "123456");

            // 执行查询
```

```
            stmt = conn.createStatement();
            String sql = "SELECT id,user_name,real_name,password,age,d_id from
t_user where id = 1";
            ResultSet rs = stmt.executeQuery(sql);

            // 获取结果集
            while (rs.next()) {
                Integer id = rs.getInt("id");
                // ...
                user.setId(id);
                // ...
                System.out.println(user);
            }
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
    }
```

通过上面的代码，我们可以发现问题还是比较多的。

1.1.1 资源管理

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。

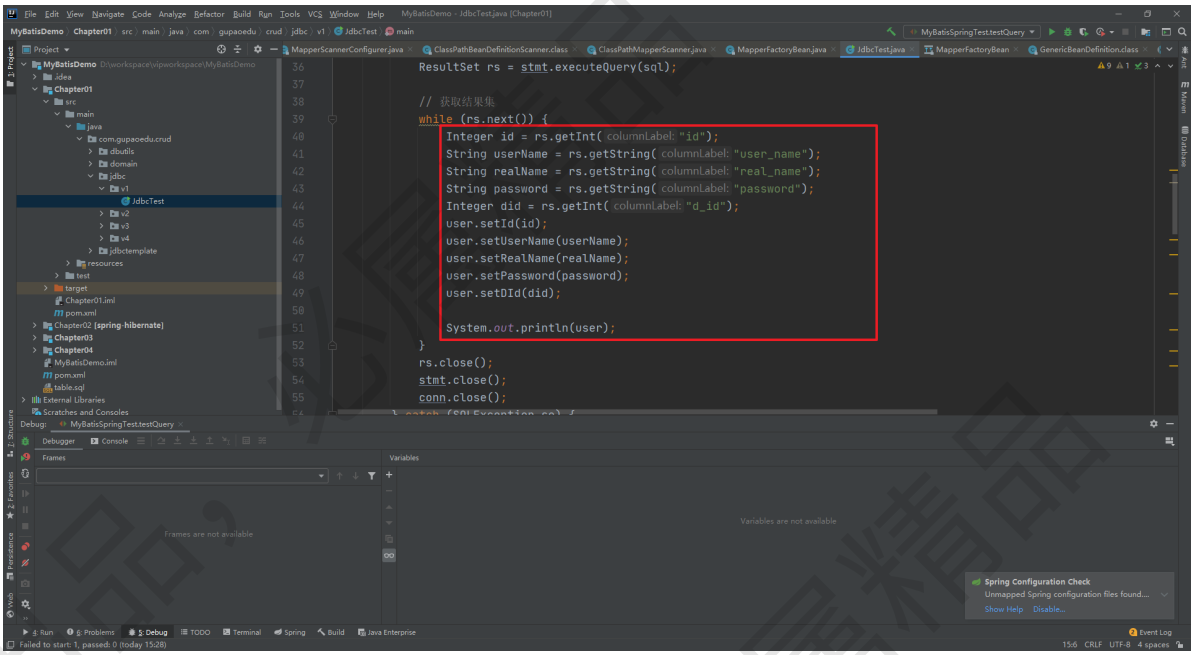
```
rs.close();
stmt.close();
conn.close();
```

1.1.2 SQL语句

在代码中我们直接将SQL语句和业务代码写在了一起，耦合性太高了，我们需要把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

1.1.3 结果集映射

在上面的代码中我们需要根据字段取出值，然后把值设置到对应对象的属性中，这个操作也是很繁琐的，所以我们也希望框架能够自动帮助我们实现结果集的转换，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。



1.4 对外API

实现了上面的功能以后，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

项目的需求我们也已经清楚了，那么我们应该要怎么来解决这些问题呢？，我们先来分析下需要哪些核心对象

1.3.1 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**；

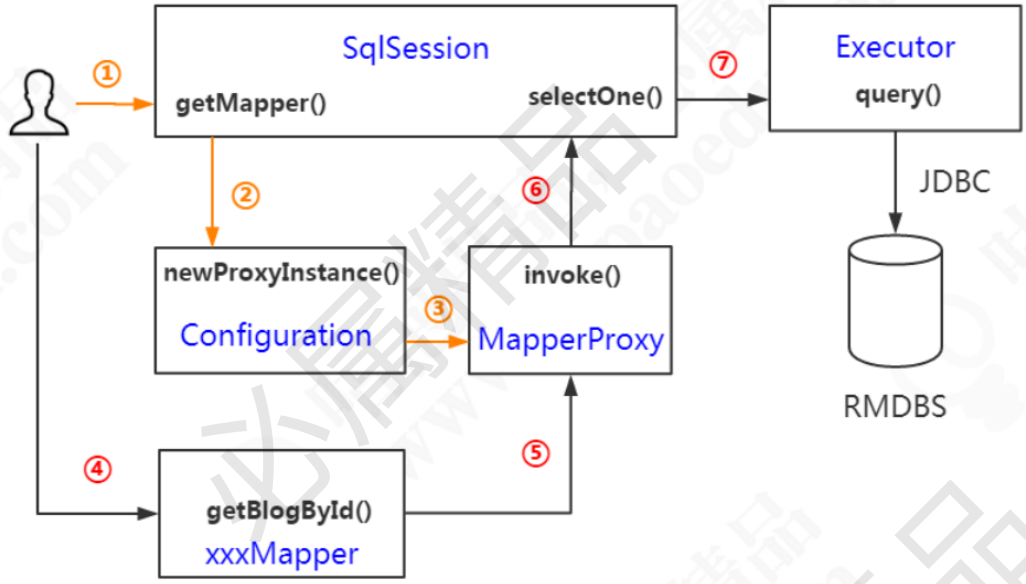
2、执行对数据库的操作，处理参数和结果集的映射，创建和释放资源，我们需要定义一个**执行器**；

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 操作流程



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()。

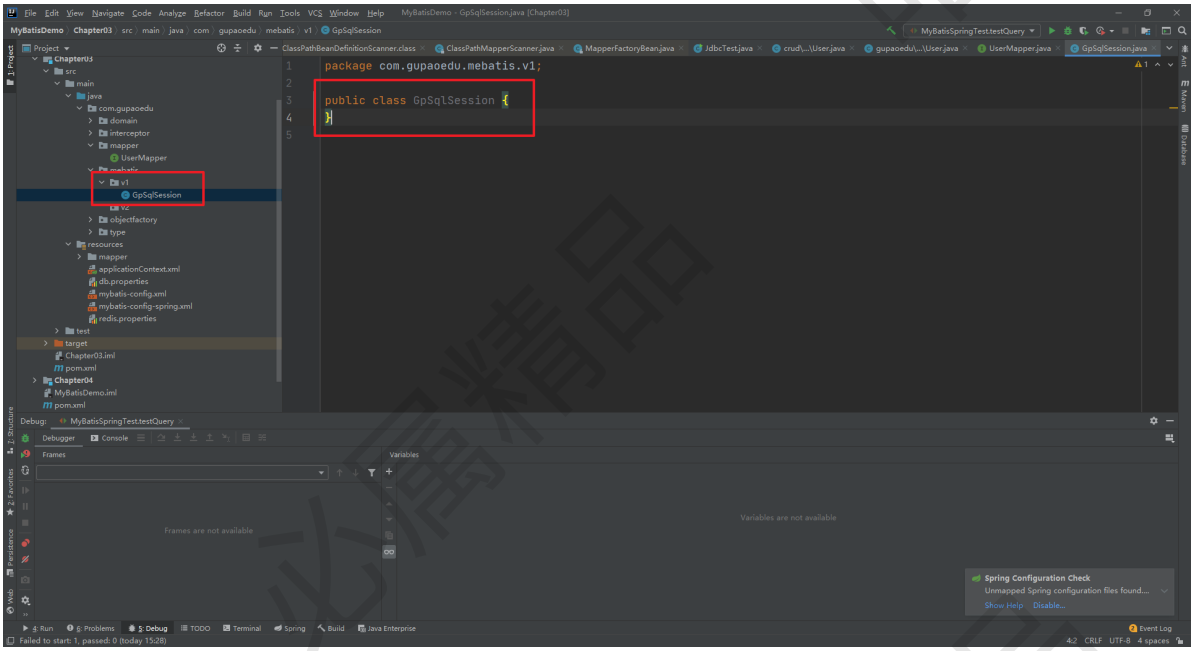
7、SqlSession只是一个API，还不是真正的SQL执行者，所以接下来会调用执行器Executor的query()方法。

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。

2.V1.0实现

2.1 SqlSession

针对不用户的请求操作我们可以通过SqlSession来处理，在SqlSession中可以提供基础的操作API，我们定义的名称为GpSqlSession，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GPConfiguration，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。

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GPExecutor，我们也创建它

```
    private GPConfiguration configuration;
    private GPExecutor executor;
```

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。

```
    // GPSqlSession.java
    public <T> T selectOne(String statementId, Object parameter){
        String sql = statementId; // 先用statementId代替SQL
        return executor.query(sql, parameter);
    }
```

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。

```
    public <T> T query(String sql, Object paramater ) {
        return null;
    }
```

`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。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。

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)。

```
    // 获取代理对象 GPSqlSession.java
    public <T> T getMapper(Class clazz){
        return (T)configuration.getMapper(clazz);
    }
```

我们先在Configuration创建这个方法，返回类型一样是泛型，先返回空。

```
    // GPConfiguration.java
    public <T> T getMapper(Class clazz) {
        return null;
    }
```

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：GPMapperProxy。

实现invoke()方法。

```
public class GPMapperProxy implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        return null;
    }
}
```

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)：

```
// GPConfiguration.java
public <T> T getMapper(Class<T> clazz, GPSqlSession sqlSession) {
    return (T) Proxy.newProxyInstance(this.getClass().getClassLoader(),
            new Class[]{clazz},
            new GPMapperProxy());
}
```

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

```
// GPMapperProxy.java
private GPSqlSession sqlSession;

public GPMapperProxy(GPSqlSession sqlSession) {
    this.sqlSession = sqlSession;
}
```

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）：

```
// GPConfiguration.java
public <T> T getMapper(Class<T> clazz, GPSqlSession sqlSession) {
    return (T) Proxy.newProxyInstance(this.getClass().getClassLoader(),
            new Class[]{clazz},
            new GPMapperProxy(sqlSession));
}
```

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

```
// 获取代理对象
public <T> T getMapper(Class clazz){
    return (T)configuration.getMapper(clazz,this);
}
```

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：

```
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        String mapperInterface = method.getDeclaringClass().getName();
        String methodName = method.getName();
        String statementId = mapperInterface + "." + methodName;

        return sqlSession.selectOne(statementId, args[0]);
    }
```

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mesql.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：

```
com.gupaoedu.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mesql.properties文件里面的所有KV键值对：

```
    // 普通的API方法
    public <T> T selectOne(String statementId, Object parameter){
        String sql = statementId; // 先用statementId代替SQL
        if( null != sql && !"".equals(sql)){
            return executor.query(sql, parameter);
        }
        return null;
    }
```

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。

参数用传入的参数替换%d占位符，需要format一下。

2.4 Executor

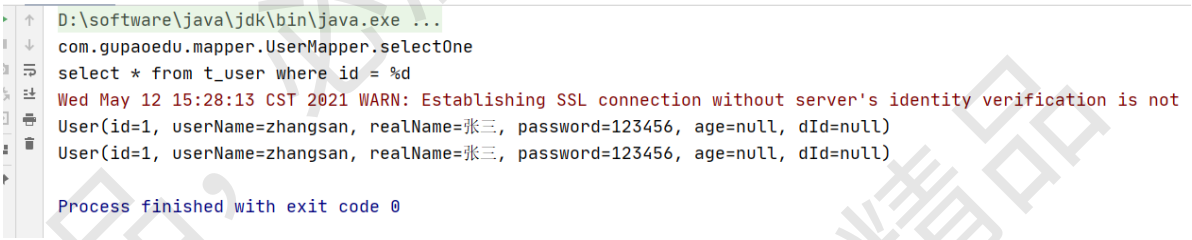
在Executor中我们就可以直接来执行SQL的执行了

```
public <T> T query(String sql,Object parameter){
        Connection conn = null;
        Statement stmt = null;
        User user = new User();

        try {
            // Class.forName("com.mysql.jdbc.Driver");

            // 打开连接
            conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mybatisdb?
characterEncoding=utf-8&serverTimezone=UTC", "root", "123456");

            // 执行查询
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(String.format(sql,parameter));

            // 获取结果集
            while (rs.next()) {
                user.setId(rs.getInt("id"));
                user.setUserName(rs.getString("user_name"));
                user.setPassword(rs.getString("password"));
                user.setRealName(rs.getString("real_name"));
            }
            System.out.println(user);

            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) {
                    stmt.close();
                }
            } catch (SQLException se2) {
            }
            try {
                if (conn != null) {
                    conn.close();
                }
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
```

```
        return (T) user;
    }
```

到这儿我们就可以来写个测试类来跑下程序了

```
    public static void main(String[] args) {
        GpSqlSession sqlSession = new GpSqlSession();
        //sqlSession.selectOne("com.gupaoedu.domain.User.selectOne",1);
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        User user = mapper.selectOne(1);
        System.out.println(user);
    }
```



3. V1.0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。

二、手写MyBatis v2.0

1.V1.0的代码优化目标

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）。

2.V1.0的功能增强目标

在方法上使用注解配置SQL；

查询带缓存功能；

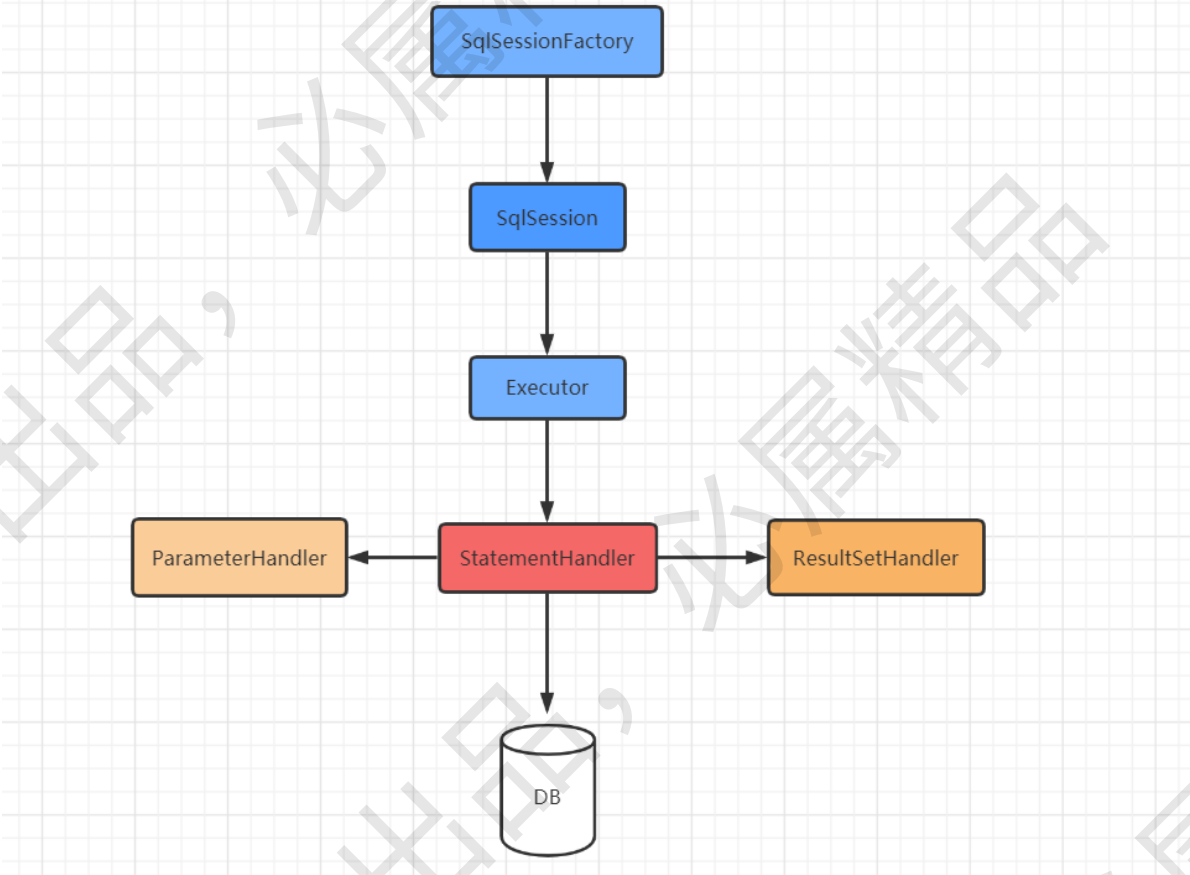
支持自定义插件。

3.优化实现

3.1 数据库连接硬编码

建立全局的数据库配置文件

3.2 Executor职责划分



拆分成三个对象：ParameterHandler、StatementHandler、ResultSetHandler。

ParameterHandler

ParameterHandler负责参数处理，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

```
public class ParameterHandler {
    private PreparedStatement psmt;

    public ParameterHandler(PreparedStatement statement) {
        this.psmt = statement;
   }

    /**
     * 从方法中获取参数，遍历设置SQL中的？占位符
     * @param parameters
     */
```

```
    public void setParameters(Object[] parameters) {
        try {
            // PreparedStatement的序号是从1开始的
            for (int i = 0; i <parameters.length; i++) {
                int k =i+1;
                if (parameters[i] instanceof Integer) {
                    psmt.setInt(k, (Integer) parameters[i]);
                } else if (parameters[i] instanceof Long) {
                    psmt.setLong(k, (Long) parameters[i]);
                } else if (parameters[i] instanceof String) {
                    psmt.setString(k , String.valueOf(parameters[i]));
                } else if (parameters[i] instanceof Boolean) {
                    psmt.setBoolean(k, (Boolean) parameters[i]);
                } else {
                    psmt.setString(k, String.valueOf(parameters[i]));
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

ResultSetHandler

通过ResultSetHandler实现结果集的处理

```
/**
     *
     * @param resultSet 结果集
     * @param type 需要转换的目标类型
     * @param <T>
     * @return
     */
    public <T> T handle(ResultSet resultSet, Class type)  {
        // 直接调用Class的newInstance方法产生一个实例
        Object pojo = null;
        try {
            pojo = type.newInstance();
            // 遍历结果集
            if (resultSet.next()) {
                // 循环赋值
                for (Field field : pojo.getClass().getDeclaredFields()) {
                    setValue(pojo, field, resultSet);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return (T)pojo;
    }

    /**
     * 通过反射给属性赋值
```

```
     */
    private void setValue(Object pojo, Field field, ResultSet rs)  {
        try{
            // 获取 pojo 的 set 方法
            Method setMethod = pojo.getClass().getMethod("set" +
firstWordCapital(field.getName()), field.getType());
            // 调用 pojo 的set 方法，使用结果集给属性赋值
            // 赋值先从resultSet取出值 setter
            setMethod.invoke(pojo, getResult(rs, field));
        }catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * 根据反射判断类型，从ResultSet中取对应类型参数
     */
    private Object getResult(ResultSet rs, Field field) throws SQLException {
        // TODO TypeHandler
        Class type = field.getType();
        String dataName = HumpToUnderline(field.getName()); // 驼峰转下划线
        // TODO 类型判断不够全
        if (Integer.class == type ) {
            return rs.getInt(dataName);
        }else if (String.class == type) {
            return rs.getString(dataName);
        }else if(Long.class == type){
            return rs.getLong(dataName);
        }else if(Boolean.class == type){
            return rs.getBoolean(dataName);
        }else if(Double.class == type){
            return rs.getDouble(dataName);
        }else{
            return rs.getString(dataName);
        }
    }

    // 数据库下划线转Java驼峰命名
    public static String HumpToUnderline(String para){
        StringBuilder sb=new StringBuilder(para);
        int temp=0;
        if (!para.contains("_")) {
            for(int i=0;i<para.length();i++){
                if(Character.isUpperCase(para.charAt(i))){
                    sb.insert(i+temp, "_");
                    temp+=1;
                }
            }
        }
        return sb.toString().toUpperCase();
    }

    /**
     * 单词首字母大写
     */
    private String firstWordCapital(String word){
        String first = word.substring(0, 1);
        String tail = word.substring(1);
```

```
        return first.toUpperCase() + tail;
    }
```

StatementHandler

StatementHandler 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。

```
private ResultSetHandler resultSetHandler = new ResultSetHandler();

    public <T> T query(String statement, Object[] parameter, Class pojo){
        Connection conn = null;
        PreparedStatement preparedStatement = null;
        Object result = null;

        try {
            conn = getConnection();
            preparedStatement = conn.prepareStatement(statement);
            ParameterHandler parameterHandler = new
ParameterHandler(preparedStatement);
            parameterHandler.setParameters(parameter);
            preparedStatement.execute();
            try {
                result =
resultSetHandler.handle(preparedStatement.getResultSet(), pojo);
            } catch (Exception e) {
                e.printStackTrace();
            }
            return (T)result;
        } catch (Exception e){
            e.printStackTrace();
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                conn = null;
            }
        }
        // 只在try里面return会报错
        return null;
    }

    /**
     * 获取连接
     * @return
     * @throws SQLException
     */
    private Connection getConnection() {
        String driver = Configuration.properties.getString("jdbc.driver");
        String url =  Configuration.properties.getString("jdbc.url");
        String username = Configuration.properties.getString("jdbc.username");
        String password = Configuration.properties.getString("jdbc.password");
```

```
        Connection conn = null;
        try {
            Class.forName(driver);
            conn = DriverManager.getConnection(url, username, password);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return conn;
    }
```

3.3 Configuration细化

创建：MapperProxyFactory，用来创建代理对象。

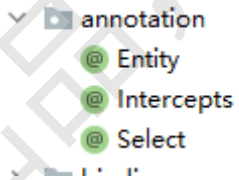
创建：MapperRegistry，维护接口和工厂类映射关系。

创建：SqlSessionFactory，用来创建SqlSession。

定义了Executor接口和基本实现SimpleExecutor。

3.4 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。



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

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。

3.5 支持缓存

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。

```
        if (properties.getString("cache.enabled").equals("true")) {
            executor = new CachingExecutor(new SimpleExecutor());
        }else{
            executor = new SimpleExecutor();
        }
```

定义了一个CachingExecutor。

```
private Executor delegate;
    private static final Map<Integer, Object> cache = new HashMap<>();

    public CachingExecutor(Executor delegate) {
        this.delegate = delegate;
    }

    @Override
    public <T> T query(String statement, Object[] parameter, Class pojo)  {
        // 计算CacheKey
        CacheKey cacheKey = new CacheKey();
        cacheKey.update(statement);
        cacheKey.update(joinStr(parameter));
        // 是否拿到缓存
        if (cache.containsKey(cacheKey.getCode())) {
            // 命中缓存
            System.out.println("【命中缓存】");
            return (T)cache.get(cacheKey.getCode());
        }else{
            // 没有的话调用被装饰的SimpleExecutor从数据库查询
            Object obj = delegate.query(statement, parameter, pojo);
            cache.put(cacheKey.getCode(), obj);
            return (T)obj;
        }
    }

    // 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
    public String joinStr(Object[] objs){
        StringBuffer sb = new StringBuffer();
        if(objs !=null && objs.length>0){
            for (Object objStr : objs) {
                sb.append(objStr.toString() + ",");
            }
        }
        int len = sb.length();
        if(len >0){
            sb.deleteCharAt(len-1);
        }
        return  sb.toString();
    }
```

在DefaultSqlSession调用Executor时，会先走到CachingExecutor。

定义了一个CacheKey用于计算缓存Key。

```
public class CacheKey {
    // MyBatis抄袭了我的设计
    private static final int DEFAULT_HASHCODE = 17; // 默认哈希值
    private static final int DEFAULT_MULTIPLIER = 37; // 倍数

    private int hashCode;
    private int count;
    private int multiplier;

    /**
     * 构造函数
     */
```

```
    public CacheKey() {
        this.hashCode = DEFAULT_HASHCODE;
        this.count = 0;
        this.multiplier = DEFAULT_MULTIPLIER;
    }

    /**
     * 返回CacheKey的值
     * @return
     */
    public int getCode() {
        return hashCode;
    }

    /**
     * 计算CacheKey中的HashCode
     * @param object
     */
    public void update(Object object) {
        int baseHashCode = object == null ? 1 : object.hashCode();
        count++;
        baseHashCode *= count;
        hashCode = multiplier * hashCode + baseHashCode;
    }
}
```

3.6 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。

3.7 工作流程分析

启动解析

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

静态代码块解析Properties文件。

```
    static{
        sqlMappings = ResourceBundle.getBundle("v2sql");
        properties = ResourceBundle.getBundle("mybatis");
    }
```

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。

```
public Configuration() {
        // Note：在properties和注解中重复配置SQL会覆盖
        // 1.解析sql.properties
        for(String key : sqlMappings.keySet()){
            Class mapper = null;
            String statement = null;
            String pojoStr = null;
            Class pojo = null;
            // properties中的value用--隔开，第一个是SQL语句
            statement = sqlMappings.getString(key).split("--")[0];
            // properties中的value用--隔开，第二个是需要转换的POJO类型
            pojoStr = sqlMappings.getString(key).split("--")[1];
            try {
                // properties中的key是接口类型+方法
                // 从接口类型+方法中截取接口类型
                mapper = Class.forName(key.substring(0,key.lastIndexOf(".")));
                pojo = Class.forName(pojoStr);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }

            MAPPER_REGISTRY.addMapper(mapper, pojo); // 接口与返回的实体类关系
            mappedStatements.put(key, statement); // 接口方法与SQL关系
        }

        // 2.解析Mapper接口配置，扫描注册
        String mapperPath = properties.getString("mapper.path");
        scanPackage(mapperPath);
        for (Class<?> mapper : mapperList) {
            parsingClass(mapper);
        }
        // 3.解析插件，可配置多个插件
        String pluginPathValue = properties.getString("plugin.path");
        String[] pluginPaths = pluginPathValue.split(",");
        if (pluginPaths != null) {
            // 将插件添加到interceptorChain中
            for (String plugin : pluginPaths) {
                Interceptor interceptor = null;
                try {
                    interceptor = (Interceptor)
Class.forName(plugin).newInstance();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                interceptorChain.addInterceptor(interceptor);
            }
        }
    }
```

获取SqlSession

在这里面也创建了一个executor。

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。

```
public Executor newExecutor() {
       Executor executor = null;
       if (properties.getString("cache.enabled").equals("true")) {
           executor = new CachingExecutor(new SimpleExecutor());
      }else{
           executor = new SimpleExecutor();
      }


       if (interceptorChain.hasPlugin()) {
           return (Executor)interceptorChain.pluginAll(executor);
      }
       return executor;
  }
```

Mapper接口调用

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

```
// 获取包含了h MapperProxy代理
UserMapper mapper = sqlSession.getMapper(UserMapper.class);
User user = mapper.selectOne(1);
```

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、PrepareStatement、ResultSetHandler。

3.8 V2.0 升级

- 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；
- l细节考虑不足，异常处理有点粗暴；