MyBatis源码剖析

1 MyBatis 框架概述

mybatis 是一个优秀的基于 java 的持久层框架，它内部封装了 jdbc，使开发者只需要关注 sql 语句本身，而不需要花费精力去处理加载驱动、创建连接、创建 statement 等繁杂的过程。

mybatis 通过 xml或注解的方式将要执行的各种 statement 配置起来，并通过 java 对象和 statement 中 sql的动态参数进行映射生成最终执行的 sql 语句，最后由 mybatis 框架执行 sql 并将结果映射为 java 对象并返回。采用 ORM 思想解决了实体和数据库映射的问题，对 jdbc 进行了封装，屏蔽了 jdbc api 底层访问细节，使我们不用与 jdbc api 打交道，就可以完成对数据库的持久化操作。

2 JDBC缺陷总结

- 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。
- Sql 语句在代码中硬编码，造成代码不易维护，实际应用 sql 变化的可能较大， sql 变动需要改变java 代码。
- 使用 preparedStatement 向占有位符号传参数存在硬编码，因为 sql 语句的 where 条件不一定，可能多也可能少，修改 sql 还要修改代码，系统不易维护。
- 对结果集解析存在硬编码（查询列名）， sql 变化导致解析代码变化，系统不易维护，如果能将数据库记录封装成 pojo 对象解析比较方便。

3 MyBatis快速入门

3.1 搭建 Mybatis 开发环境

创建工程之前，我们先新建数据库mybatis,并在数据库中新建一张User表，并加一些数据。表包含:id,username,birthday,sex,address字段

2.2.1 创建Maven工程

2.2.2 导入依赖

1. mybatis
2. mysql驱动
3. log4j
4. junit单元测试

```
<?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.itheima</groupId>
    <artifactId>mybatis-day01-demo1</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
    <!--打包方式-->
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    </properties>

    <!--引入相关依赖-->
    <dependencies>
        <!--MyBatis依赖包-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.5</version>
        </dependency>

        <!--MySQL驱动包-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.6</version>
            <scope>runtime</scope>
        </dependency>

        <!--日志包-->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>

        <!--测试包-->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

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

2.2.2 创建User

创建com.itheima.domain包，在该包下创建User对象，并添加对应的属性。

```
public class User implements Serializable {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;
    //略 get...set...toString...
}
```

2.2.3 创建UserMapper接口

创建com.itheima.mapper包，并在该包下创建接口，代码如下：

它其实就是dao层的接口

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

2.2.4 创建UserMapper.xml

这个xml配置文件的位置，必须和对应的那个Mapper接口的位置一样。而且其文件名也要和接口名一样

在com.itheima.mapper包下创建UserMapper.xml,并在UserMapper.xml中添加一个select查询结点，代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
        <!DOCTYPE mapper
                PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
                "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.mapper.UserMapper">
    <!--findAll-->
    <select id="findAll" resultType="com.itheima.domain.User">
        SELECT * FROM  user
    </select>
</mapper>
```

2.2.5 创建SqlMapConfig.xml

在main/resources下创建SqlMapConfig.xml,在文件中配置数据源信息和加载映射文件，代码如下：

这个配置文件的名字不是固定的，你可以随便命名

```
<?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>
    <!--数据源配置-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
characterEncoding=utf-8"/>
                <property name="username" value="root"/>
                <property name="password" value="123"/>
            </dataSource>
        </environment>
    </environments>
    <!--加载映射文件-->
    <mappers>
        <mapper resource="com/itheima/mapper/UserMapper.xml"/>
    </mappers>
</configuration>
```

2.2.6 创建log4j.properties

为了方便查看日志，在main/resources下创建log4j.properties文件，代码如下：

```
log4j.rootLogger=DEBUG,Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d [%t] %-5p [%c] - %m%n
log4j.logger.org.apache=DEBUG
```

2.3 编写测试类

在test包下创建com.itheima.test,再在该包下创建MyBatisTest类，代码如下：

```
public class MyBatisTest {
    @Test
    public void testFindAll() throws IOException {
        //读取配置文件
        InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
        //创建SqlSessionFactoryBuilder对象
        SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
        //通过SqlSessionBuilder对象构建一个SqlSessionFactory
        SqlSessionFactory sqlSessionFactory = builder.build(is);
        //通过SqlSessionFactory构建一个SqlSession
        SqlSession session = sqlSessionFactory.openSession();
        //通过SqlSession实现增删改查
        UserMapper userMapper = session.getMapper(UserMapper.class);
        List<User> users = userMapper.findAll();
        //打印输出
        for (User user : users) {
            System.out.println(user);
```

```
        }
        //关闭资源
        session.close();
        is.close();
    }
}
```

4 自定义 Mybatis 框架

本章我们将使用前面所学的基础知识来构建一个属于自己的持久层框架，将会涉及到的一些知识点：工厂模式（Factory 工厂模式）、构造者模式（Builder 模式）、动态代理模式，反射， xml 解析，数据库元数据，数据库元数据等。

4.1 MyBatis框架设计模式分析

我们来看看MyBatis框架使用过程中用到的一些设计模式。

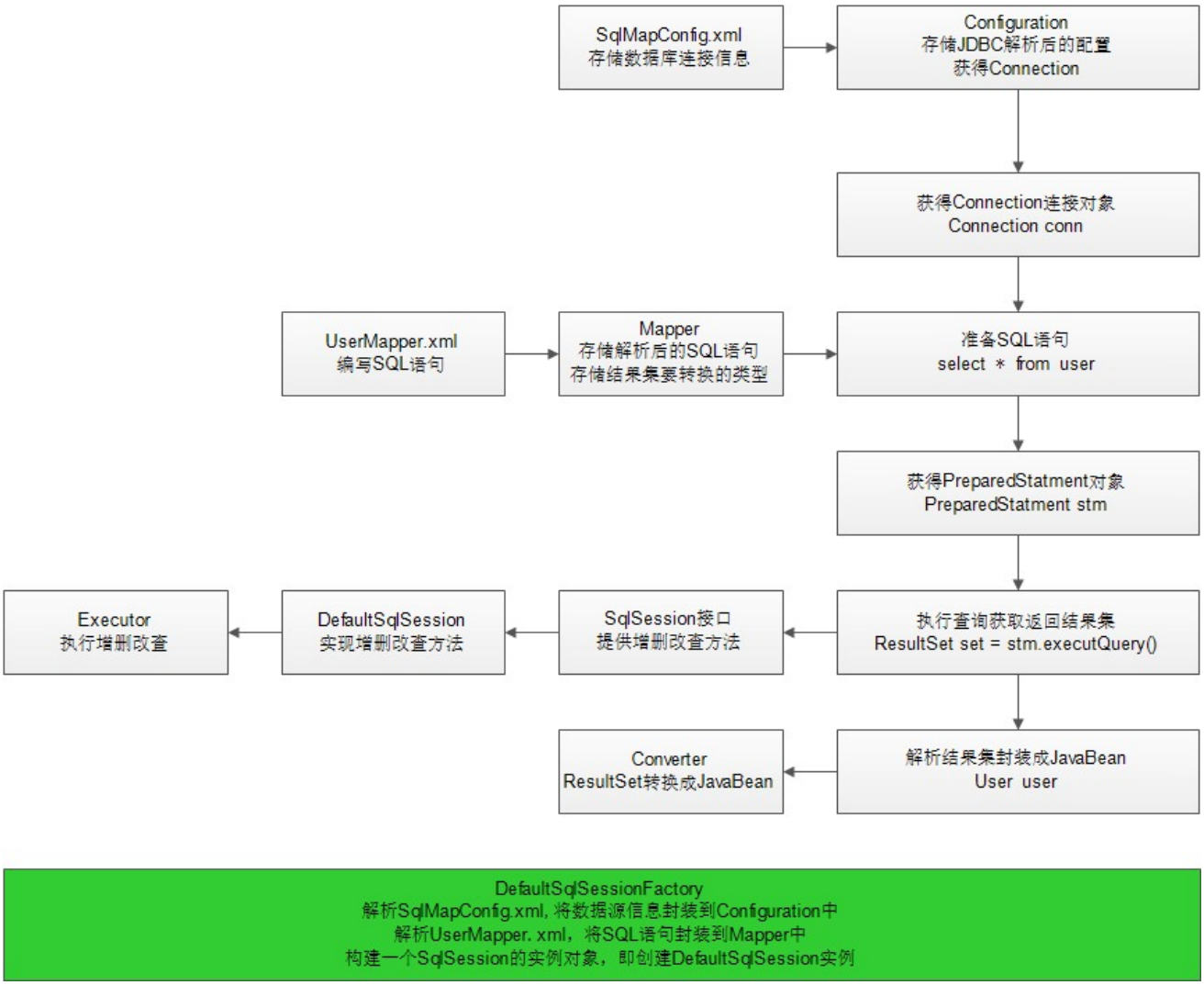
1. 使用SqlSessionFactoryBuilder创建SqlSessionFactory工厂对象的时候使用的是构建者模式
2. 使用SqlSessionFactory创建SqlSession时候使用的是工厂模式
3. 使用SqlSession创建UserMapper接口代理对象的时候使用的是动态代理模式

4.2 执行查询所有用户的SQL语句必须的步骤

1. 需要获取连接
2. 需要SQL语句
3. 需要将结果集中的数据封装到JavaBean中

4.3 基于JDBC实现封装流程分析

基于上面我们说到的JDBC流程再结合MyBatis流程，我们封装一个持久城框架，达到MyBatis中的增删改查效果。来分析一波：



通过上图分析，我们可以发现:

1. 通过解析主配置文件SqlMapConfig.xml可以获取username，password，driver，url等信息，通过这些信息能够获取Connection对象
2. 通过解析映射配置文件UserMapper.xml可以获取要执行的SQL语句以及结果集要封装到的JavaBean类的全限定名resultType
3. 通过反射机制和数据库元数据可以将结果集中的数据遍历出来并封装到对应的JavaBean对象中

4.4 准备工作

4.4.1 需要使用到的jar包:log4j,mysql驱动,dom4j以及其XPath,c3p0连接池

4.4.2 拷贝文件

分别将上一个入门工程中的User.java、UserMapper.java、MyBatisTest.java、UserMapper.xml、log4j.properties、SqlMapConfig.xml都拷贝到该工程中，将XML中引用的DTD文件约束去掉，不然每次解析都会去网上下载。

```
<?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.itheima</groupId>
    <artifactId>mybatis-day01-demo2-custom</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--打包方式jar-->
    <packaging>jar</packaging>

    <dependencies>
        <!-- 日志坐标 -->
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.12</version>
        </dependency>
        <!-- mysql驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.36</version>
        </dependency>

        <!-- 解析xml的dom4j -->
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <!-- dom4j的依赖包jaxen -->
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.3</version>
        </dependency>

        <!--c3p0连接池-->
        <dependency>
            <groupId>c3p0</groupId>
            <artifactId>c3p0</artifactId>
            <version>0.9.1.2</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

    <build>
        <!--IDEA是不会编译src的java目录的xml文件，如果需要读取，则需要手动指定哪些配置文件需要读取-->
        <resources>
            <resource>
```

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

4.4.3 工程错误改造

我们看到上面的工程存在错误，我们对他进行改造一下，首先创建对应的文件来去掉错误。

4.4.3.1 创建Resources类

该类的主要作用是读取类路径下的资源文件，所以要求被它读取的文件务必放到classes下，我们创建它的目的主要是模拟加载读取UserMapper.xml和SqlMapConfig.xml文件。

```
public class Resources {
    public static InputStream getResourceAsStream(String path){
        InputStream is = Resources.class.getClassLoader().getResourceAsStream(path);
        return is;
    }
}
```

此时MyBatisTest中的Resources类就引用上面创建的类就可以去掉一个错误了。

4.4.3.2 创建SqlSessionFactoryBuilder类

创建该类，并创建一个build方法返回一个SqlSessionFactory对象，但SqlSessionFactory还没创建，所以接着我们需要创建它。

```
public class SqlSessionFactoryBuilder {
    public SqlSessionFactory build(InputStream is) {
        return null;
    }
}
```

4.4.3.3 创建SqlSessionFactory接口及其实现类

创建SqlSessionFactory接口及其实现类，并且创建一个openSession方法。

```
public interface SqlSessionFactory {
    SqlSession openSession();
}
public class DefaultSqlSessionFactory implements SqlSessionFactory {
    @Override
    public SqlSession openSession() {
        return null;
    }
}
```

4.4.3.4 创建SqlSession接口及其实现类

创建SqlSession接口，并在接口里面创建对应方法，然后创建一个DefaultSqlSession实现类

```
public interface SqlSession {
    UserMapper getMapper(Class<UserMapper> userMapperClass);
    void close();
}
public class DefaultSqlSession implements SqlSession {
    @Override
    public UserMapper getMapper(Class<UserMapper> userMapperClass) {
        return null;
    }
    @Override
    public void close() {
    }
}
```

把MyBatisTest类重新导包后，错误就全部消失了。接着我们就要开始对每个模块展开分析和代码实现了。

4.5 获取Connection实现

我们回到刚才我们的分析，首先我们要解析SqlMapConfig.xml,并把信息存储到Configuration对象中，然后通过Configuration对象获取数据库连接对象Connection。我们可以分这么几个步骤完成：

- 创建XMLConfigBuilder解析SqlMapConfig.xml
- 创建Configuration对象，存储解析的数据库连接信息
- 改造Configuration对象，使它具备获取数据库链接Connection的功能

4.5.1 创建Configuration对象，存储解析的数据库连接信息

```
public class Configuration {
    private String username;
    private String password;
    private String url;
    private String driver;
    //get..set..toString..
}
```

4.5.2 创建XMLConfigBuilder解析SqlMapConfig.xml

```
public class XMLConfigBuilder{
public static Configuration loadConfiguration(InputStream is){
        try {
            //1）数据库配置信息存储
            Configuration cfg = new Configuration();
            //创建SAXReader对象读取XML文件字节输入流
            SAXReader reader = new SAXReader();
            Document document = reader.read(is);
            //解析配置文件,获取根节点信息,//property表示获取根节点下所有的property结点对象
            List<Element> rootList = document.selectNodes("//property");
            //循环迭代所有结点对象
            for (Element element : rootList) {
                //name属性的值
                String name = element.attributeValue("name");
                //vallue属性的值
                String value = element.attributeValue("value");

                //2）将解析的数据库连接信息存储到Configuration中
                //数据库驱动
                if(name.equals("driver")){
                    cfg.setDriver(value);
                }else if(name.equals("url")){
                    //数据库连接地址
                    cfg.setUrl(value);
                }else if(name.equals("username")){
                    //数据库账号
                    cfg.setUsername(value);
                }else if(name.equals("password")){
                    //数据库密码
                    cfg.setPassword(value);
                }
            }
            //获取需要解析的XML路径
            String resource = element.attributeValue("resource");
            //拿到映射配置文件的路径
            ...接下来看下面的分析准备解析映射配置文件
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

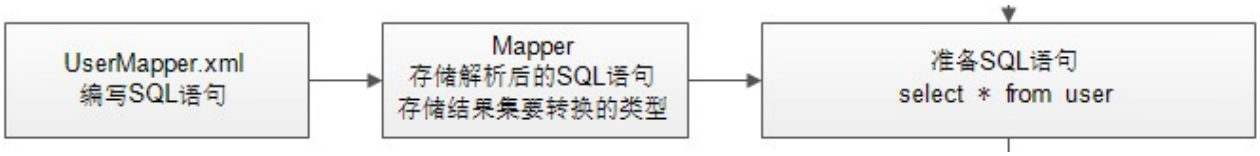
4.5.3 改造Configuration对象，使它具备获取数据库链接Connection的功能

在Configuration对象中创建ComboPooledDataSource对象，并创建获得数据源的方法getDataSource，再创建一个获得Connection的方法getConnection，getConnection通过调用getDataSource获得数据源，然后获得Connection对象。

```
public class Configuration {
    //数据库用户名
    private String username;
    //数据库用户密码
```

```
    private String password;
    //数据库连接地址
    private String url;
    //数据库驱动
    private String driver;
    //创建数据源
    private ComboPooledDataSource dataSource = new ComboPooledDataSource();
    //get..set..toString..
    //获取数据源
    private DataSource getDataSource(){
        //设置数据源配置
        try {
            dataSource.setUser(username);
            dataSource.setPassword(password);
            dataSource.setJdbcUrl(url);
            dataSource.setDriverClass(driver);
        } catch (PropertyVetoException e) {
            e.printStackTrace();
        }
        return dataSource;
    }
    public Connection getConnection(){
        try {
            return getDataSource().getConnection();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

4.5.4 解析UserMapper.xml，提取SQL语句和返回参数类型



UserMapper.xml内容：

```
<?xml version="1.0" encoding="UTF-8" ?>
<mapper namespace="com.itheima.mapper.UserMapper">
    <!--findAll-->
    <select id="findAll" resultType="com.itheima.domain.User">
        SELECT * FROM  user
    </select>
</mapper>
```

接着上面流程，我们需要解析UserMapper.xml获取SQL语句，并获取返回值的类型，这个时候我们可以考虑封装一个Mapper对象，存储对应的SQL语句和返回值类型。针对这个操作实现，我们可以分为下面几个步骤实现：

- 解析UserMapper.xml获得SQL语句和返回类型的全限定名
- 创建Mapper对象，存储SQL语句和返回类型的全限定名

- 将解析的结果封装到Mapper中

4.5.4.1 创建Mapper对象，存储SQL语句和返回类型的全限定名

Mapper对象用于存储SQL语句和返回的JavaBean全限定名，这时候我们可以考虑定义2个属性来接收存储。

```
public class Mapper  {
    //执行的SQL语句
    private String sql;
    //执行SQL语句后要返回的JavaBean全限定名
    private String resultType;
    //带参构造函数
    public Mapper(String sql, String resultType) {
        this.sql = sql;
        this.resultType = resultType;
    }
    //get..set..toString
}
```

4.5.4.2 封装一个loadMapper方法将解析映射配置文件的数据封装到Mapper对象中

我们接着将刚才解析的XML信息存储到Mapper中。我们分析下，目前我们只存在一个select结点，如果以后存在多个怎么操作？如下代码：

```
<?xml version="1.0" encoding="UTF-8" ?>
<mapper namespace="com.itheima.mapper.UserMapper">
    <!--findAll-->
    <select id="findAll" resultType="com.itheima.domain.User">
        SELECT * FROM  user
    </select>
    <!--findOne-->
    <select id="findOne" resultType="com.itheima.domain.User">
        SELECT * FROM  user WHERE id=1
    </select>
</mapper>
```

我们可以把多个select结点信息封装成多个Mapper，并将多个Mapper存储到Map<String,Mapper>中，key可以用id来表示，但这种情况只适合单个UserMapper.xml文件，如果我们再创建一个TeacherMapper.xml解析，同样也有select id="findAll"的话，多个文件解析，key会冲突。解决这种冲突问题，我们可以把namespace也作为key的一部分，只要namespace不冲突就可以保证key不冲突，key=namespace+.+id 即可。代码如下：

```
/**
 * 解析UserMapper.xml，提取SQL语句和返回JavaBean全限定名
 * path为UserMapper.xml的路径
 */
public static Map<String,Mapper> loadMapper(String path){
    /****
    * 1) 定义一个Map<String,Mapper> mappers
    *    用于存储解析的XML封装的Mapper信息
    */
    Map<String,Mapper> mappers = new HashMap<String,Mapper>();
```

```
    try {
        //获得文件字节输入流
        InputStream is = Resources.getResourceAsStream(path);
        //创建SAXReader对象,加载文件字节输入流
        SAXReader reader = new SAXReader();
        Document document = reader.read(is);
        //获得根节点
        Element rootElement = document.getRootElement();
        //获取命名空间的值
        String namespace = rootElement.attributeValue("namespace");
        //获取所有select结点
        List<Element> selectList = document.selectNodes("//select");
        //循环所有select结点
        for (Element element : selectList) {
            //获取ID属性值
            String id = element.attributeValue("id");
            //获取resultType属性值
            String resultType = element.attributeValue("resultType");
            //获取SQL语句
            String sql = element.getText();
            //2) 构建Mapper对象
            Mapper mapper = new Mapper(sql,resultType);
            //key = namespace+.+id;
            String key = namespace+"."+id;
            //存储到Map中
            mappers.put(key,mapper);
        }
        return mappers;
    } catch (DocumentException e) {
        e.printStackTrace();
    }
    return mappers;
}
```

4.5.4.3 将Mapper存入到Configuration中

按照JDBC操作流程，最终调用SQL语句的是PreparedStatment对象，而PreparedStatment对象由Connection对象构建，所以我们可以把SQL语句给Configuration对象管理。按照这个思路，可以在获取Configuration对象中创建一个Map<String,Mapper>来存储Mapper，可以直接在解析完SqlMapConfig.xml后接着解析UserMapper.xml并将解析的Map返回并填充到Configuration中。另外loadMapper(path)中path其实是SqlMapConfig.xml中的中指定的XML，可以直接解析它，把它的值传递给loadMapper(path)方法。因此我们这里可以分为3个步骤：

- 改造Configuration，让它能够存储Mapper信息
- 解析mapper，获取UserMapper.xml路径
- 调用loadMapper解析所有XML结点获取Mapper对象并填充给Configuration

4.5.4.4 改造Configuration，让它能够存储Mapper信息

这里主要添加了一个Map<String,Mapper> mappers属性和对应的get、set方法。这里注意为了避免每次set方法调用的时候把之前解析存储的信息替换，所以直接new 了一个Map，set的时候只是往Map中塞数据。

```
public class Configuration {
    //数据库用户名
```

```
    private String username;
    //数据库用户密码
    private String password;
    //数据库连接地址
    private String url;
    //数据库驱动
    private String driver;
    //创建数据源
    private ComboPooledDataSource dataSource = new ComboPooledDataSource();
    //存储所有SQL语句和返回值全限定名
    private Map<String,Mapper> mappers = new HashMap<String,Mapper>();
    public Map<String, Mapper> getMappers() {
        return mappers;
    }
    //这里的set方法为了保证每次填充进来的数据不被覆盖，直接调用putAll塞进Map中
    public void setMappers(Map<String, Mapper> mappers) {
        this.mappers.putAll(mappers);
    }
    //...略
}
```

4.6 增删改查流程串联

4.6.1 增删改查思路分析

我们再来看看MyBatis操作的流程，从代码中我们可以看到，通过SqlSession的getMapper方法创建的代理对象是具备查询数据库功能的，也就是说它拥有操作数据库的能力，而操作数据库的能力的前提是能获得连接数据库Connection对象，我们可以基于这个想法，把上面获得的Configuration对象给DefaultSqlSession对象，这样就能通过SqlSession构建一个代理对象，对赋予这个代理对象操作数据库的能力。

```
@Test
public void testFindAll() throws IOException {
    //读取配置文件
    InputStream is = Resources.getResourceAsStream("SqlMapConfig.xml");
    //创建SqlSessionFactoryBuilder对象
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //通过SqlSessionBuilder对象构建一个SqlSessionFactory
    SqlSessionFactory sqlSessionFactory = builder.build(is);
    //通过SqlSessionFactory构建一个SqlSession
    SqlSession session = sqlSessionFactory.openSession();
    //通过SqlSession实现增删改查
    UserMapper userMapper = session.getMapper(UserMapper.class);
    List<User> users = userMapper.findAll();
    //打印输出
    for (User user : users) {
        System.out.println(user);
    }
    //关闭资源
    session.close();
    is.close();
}
```

那么我们会有下面这些疑问：

1）什么时候加载解析配置文件呢?
答：我们可以在SqlSessionFactory.openSqlSession()的时候，初始化加载上面配置文件。

2）加载配置文件会初始化数据库连接信息，这时候需要加载读取SqlMapConfig.xml配置文件，如何通知程序读取这个配置文件?
答：早在我们第一步的时候就加载读取了SqlMapConfig.xml文件获取了文件字节输入流，我们可以在构建SqlSessionFactory对象的时候把它传给build方法，如果这时候DefaultSqlSessionFactory中可以接受这个文件字节输入流，那么在openSqlSession()的时候，就可以把这个字节输入流传给XMLConfigBuilder来解析，并获取对应的配置。

3）上面说到让DefaultSqlSession对象具备操作数据库的能力，需要把Configuration对象给DefaultSqlSession对象，如果做到呢?
答：可以在解析XML对象的时候，直接把DefaultSqlSession的实例传给XMLConfigBuilder.loadConfiguration(DefaultSqlSession session,InputStream is)

4.6.2 改造工程

4.6.2.1 改造DefaultSqlSession

在DefaultSqlSession中加上Configuration对象，让他具备操作数据库的能力，创建set方法给Configuration赋值，代码如下：

```
public class DefaultSqlSession implements SqlSession {
    //把Configuration对象给DefaultSqlSession
    private Configuration cfg;
    //创建一个set方法，给Configuration赋值
    public void setCfg(Configuration cfg) {
        this.cfg = cfg;
    }
    @Override
    public UserMapper getMapper(Class<UserMapper> userMapperClass) {
        return null;
    }
    @Override
    public void close() {

    }
}
```

4.6.2.2 改造DefaultSqlSessionFactory

改造DefaultSqlSessionFactory，加入SqlMapConfig.xml配置文件的字节输入流，并创建DefaultSqlSession对象，加入加载解析配置文件的方法loadConfiguration(sqlSession,is)

```
public class DefaultSqlSessionFactory implements SqlSessionFactory {
    //SqlMapConfig.xml的字节输入流
    private InputStream is;
    public void setIs(InputStream is) {
        this.is = is;
    }
```

```
    @Override
    public SqlSession openSession() {
        //创建一个DefaultSqlSession
        DefaultSqlSession sqlSession = new DefaultSqlSession();
        //加载解析配置文件
        Configuration cfg = XMLConfigBuilder.loadConfiguration(is);
        sqlSession.setCfg(cfg);
        return sqlSession;
    }
}
```

4.6.2.3 改造SqlSessionFactoryBuilder

改造SqlSessionFactoryBuilder的build方法，创建DefaultSqlSessionFactory对象，并将SqlMapConfig.xml的字节输入流传给DefaultSqlSessionFactory。

```
public class SqlSessionFactoryBuilder {
    /***
     * 读取并解析配置文件，构建一个SqlSessionFactory对象
     * @param is
     * @return
     */
    public SqlSessionFactory build(InputStream is) {
        //创建一个SqlSessionFactory的实例
        DefaultSqlSessionFactory sqlSessionFactory =new DefaultSqlSessionFactory();
        //给SqlSessionFactory的is属性赋值
        sqlSessionFactory.setIs(is);
        return sqlSessionFactory;
    }
}
```

4.6.3 使用动态代理创建代理对象

4.6.3.1 修改SqlSession

把其中getMapper改成通用的方法

```
/***
 * 改造成通用的方法
 * @param clazz
 * @param <T>
 * @return
 */
<T> T getMapper(Class<T> clazz);
```

4.6.3.2 修改DefaultSqlSession添加代理实现

```
@Override
public <T> T getMapper(Class<T> clazz) {
    /*****
     * 参数:
```

```
     *      1)被代理对象的类加载器
     *      2)字节数组，让代理对象和被代理对象有相同的行为[行为也就是有相同的方法]
     *      3)InvocationHandler:增强代码,需要使用提供者增强的代码，改代码是以接口的实现类方式提供的，
通常用匿名内部内，但不绝对。
     */
    return (T) Proxy.newProxyInstance(clazz.getClassLoader(), new Class[]{clazz}, new
InvocationHandler() {
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            return null;
        }
    });
}
```

4.6.3.3 动态代理实现增删改查

按照JDBC操作流程，我们得先拿到Connection对象，再拿到SQL语句，再执行获取返回结果集，再将返回结果集封装成要的对象即可。因此我们需要DefaultSqlSession，通过它来实现增删改查，实现增删改查就需要获取当前所需的Mapper，Mapper里面包含需要执行的SQL语句和执行SQL语句后返回的结果集需要转换的JavaBean对象。

我们需要用DefaultSqlSession来实现增删改查，可以直接考虑在SqlSession接口中编写增删改查，让DefaultSqlSession完成增删改查的实现，因此我们这里只需要引入SqlSession即可。

按照上面这个分析，我们可以总结为如下几个步骤实现：

- 在SqlSession中编写增删改查，在DefaultSqlSession中实现增删改查
- 在getMapper的代理实现中确定当前操作所需要的Mapper
- 在getMapper的代理实现中引入SqlSession，通过调用对应增删改查实现数据库操作

4.6.3.4 在SqlSession中编写增删改查，在DefaultSqlSession中实现增删改查

修改SqlSession，在SqlSession中增加selectList方法

```
/***
 * 集合查询
 * @param <E>
 * @return
 */
<E> List<E> selectList(String statement);
```

修改DefaultSqlSession，在DefaultSqlSession中实现selectList方法,其中集合查询我们用到了一个Converter转换器，转换器的写法紧接着在后面会列出。

```
@Override
public <E> List<E> selectList(String statement) {
    //获取对应的Mapper
    Mapper mapper = cfg.getMappers().get(statement);
    //JDBC操作流程实现
    if(mapper!=null){
        //执行查询
        Connection conn = null;
        PreparedStatement stm = null;
```

```
        ResultSet resultSet = null;
        try {
            //获取Connection对象
            conn = cfg.getConnection();
            //获取PreparedStatment
            stm = conn.prepareStatement(mapper.getSql());
            //执行查询
            resultSet = stm.executeQuery();
            //调用Converter实现转换
            List<E> list = Converter.list(resultSet,Class.forName(mapper.getResultType()));
            return list;
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            try {
                if(resultSet!=null){
                    resultSet.close();
                }
                if(stm!=null){
                    stm.close();
                }
                //关闭Connection
                this.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    return null;
}
```

4.6.3.5 Converter转换器编写

Converter转换器主要利用反射机制实现ResultSet转成JavaBean,代码实现如下：

```
public class Converter {
    /**
     * 这个方法，是将结果集中的每一条数据封装到一个JavaBean中，多条数据就对应多个JavaBean，再将多个
JavaBean放到一个List集合中
     * @param set
     * @param clazz
     * @param <E>
     * @return
     */
    public static <E> List<E> converList(ResultSet set, Class clazz){
        List<E> beans = new ArrayList<E>();
        //1.遍历结果集
        try {
            //根据结果集元数据，获取结果集中的每一列的列名
            ResultSetMetaData metaData = set.getMetaData();
            int columnCount = metaData.getColumnCount();//获取总列数
            while (set.next()){
                //每次遍历，遍历出一条数据，每条数据就对应一个JavaBean对象
```

```
                E o = (E) clazz.newInstance();
                //获取每一列数据，根据列名获取
                //for循环遍历出每一列
                for(int i=1;i<=columnCount;i++){
                    String columnName = metaData.getColumnName(i);
                    Object value = set.getObject(columnName);//获取该列的值
                    //将该列的值存放到JavaBean中
                    //也就是调用JavaBean的set方法,使用内省机制
                    PropertyDescriptor descriptor = new PropertyDescriptor(columnName,clazz);
                    Method writeMethod = descriptor.getWriteMethod();//获取该属性的set方法
                    //调用set方法
                    writeMethod.invoke(o,value);
                }
                //经过这个for循环，我的JavaBean就设置好了
                //把JavaBean添加进list集合
                beans.add(o);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return beans;
    }
}
```

4.6.3.6 在getMapper的代理实现中引入SqlSession，通过调用对应增删改查实现数据库操作

我们定义一个代理实现类，在代理实现类中通过被调用的方法来确定Mapper的key。然后直接调用SqlSession中定义的selectList方法。

```
public class MapperProxyFactory implements InvocationHandler {
    //1)
    private SqlSession sqlSession;
    public MapperProxyFactory(SqlSession sqlSession) {
        this.sqlSession = sqlSession;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //1、获取当前操作锁对应的Mapper信息
        String className = method.getDeclaringClass().getName();    //类的名字，和UserMapper.xml
中mapper的namespace一致
        String methodName = method.getName();   //方法名字，和UserMapper.xml中的id值一致
        String key = className+"."+methodName;
        //确定当前操作是否是查询所有
        Class<?> returnType = method.getReturnType();
        if(returnType== List.class){
            //2）执行集合查询操作
            return  sqlSession.selectList(key);
        }else{
            return null;
        }
    }
}
```

DefaultSqlSession中的getMapper方法 @Override public T getMapper(Class clazz) { /** * 参数： * 1)被代理对象的类加载器 * 2)字节数组，让代理对象和被代理对象有相同的行为[行为也就是有相同的方法] * 3)InvocationHandler:增强代码,需要使用提供者增强的代码，改代码是以接口的实现类方式提供的，通常用匿名内部内，但不绝对。 */ return (T) Proxy.newProxyInstance(clazz.getClassLoader(), new Class[]{clazz}, new MapperProxyFactory(this)); }

4.6.3.7 执行SQL代码封装Executor

将DefaultSqlSession中selectList的代码封装到一个Executor的工具类中，方便使用。我们新建一个Executor的工具类。

```
public class Executor {
    /***
     * 集合查询
     * @param conn
     * @param mapper
     * @param <E>
     * @return
     */
    public static <E> List<E> list(Connection conn, Mapper mapper) {
        //执行查询
        PreparedStatement stm = null;
        ResultSet resultSet = null;
        try {
            //获取PreparedStatment
            stm = conn.prepareStatement(mapper.getSql());
            //执行查询
            resultSet = stm.executeQuery();
            //调用Converter实现转换
            List<E> list = Converter.list(resultSet, Class.forName(mapper.getResultType()));
            return list;
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if (resultSet != null) {
                    resultSet.close();
                }
                if (stm != null) {
                    stm.close();
                }
                if(conn!=null){
                    conn.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

修改DefaultSqlSession中的selectList方法

```
@Override
public <E> List<E> selectList(String statement) {
    //获取对应的Mapper
    Mapper mapper = cfg.getMappers().get(statement);
    //JDBC操作流程实现
    if(mapper!=null){
        return Executor.list(cfg.getConnection(), mapper);
    }
    return null;
}
```

总结

通过本文，我们对mybatis源码做了深入的剖析，使用了工厂模式、构建者模式、动态代理模式、DOM4J解析xml，反射、数据库元数据等等知识实现了对mybatis的自定义。相信假以时日，我们都能对框架做到知其然知其所以然。