- SqlSession
  - Executor 执行器
  - StatementHandler 使用数据库的PreparedStatement
  - ParameterHandler 用于sql 参数的处理

环境准备

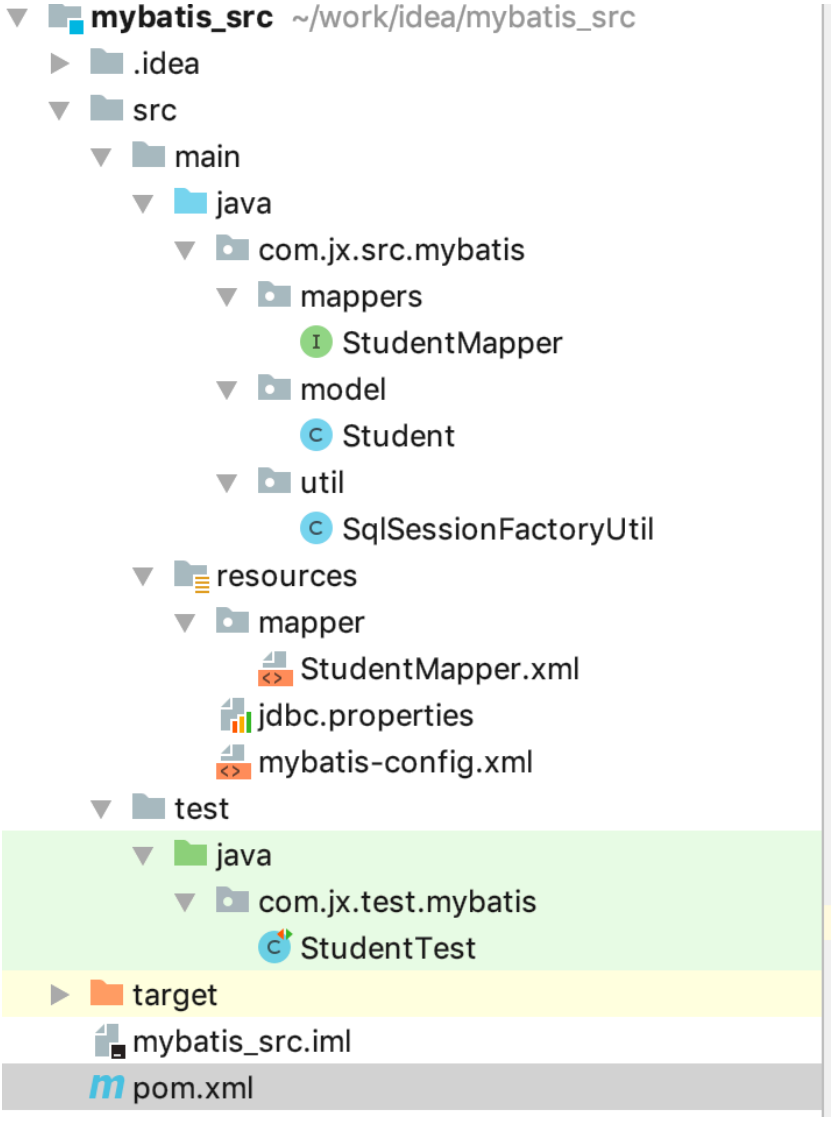
数据库

```sql
CREATE DATABASE /*!32312 IF NOT EXISTS*/`db_mybatis` /*!40100 DEFAULT CHARACTER SET utf8 */;

USE `db_mybatis`;

DROP TABLE IF EXISTS `t_student`;

CREATE TABLE `t_student` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) DEFAULT NULL,
   `age` int(11) DEFAULT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

insert  into `t_student`(`id`,`name`,`age`) values (1,'张三',10),(2,'李四',11);

```

代码结构

```
├── pom.xml
├── src
|   ├── main
|   |   ├── java
|   |   |   └── com
|   |   |       └── jx
|   |   |           └── src
|   |   |               └── mybatis
|   |   |                   ├── mappers
|   |   |                   |   └── StudentMapper.java
|   |   |                   ├── model
|   |   |                   |   └── Student.java
|   |   |                   └── util
|   |   |                       └── SqlSessionFactoryUtil.java
|   |   └── resources
|   |       ├── jdbc.properties
|   |       ├── mapper
|   |       |   └── StudentMapper.xml
|   |       └── mybatis-config.xml
|   └── test
|       └── java
|           └── com
|               └── jx
|                   └── test
```

```
|                    └── mybatis
|                        └── StudentTest.java
```



pom.xml

```xml
<?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
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.jx.src</groupId>
    <artifactId>mybatis_src</artifactId>
    <version>1.0</version>

     <dependencies>
         <dependency>
             <groupId>org.mybatis</groupId>
```

```
            <artifactId>mybatis</artifactId>
            <version>3.3.0</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>5.1.18</version>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.4</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>

</project>
```

jdbc.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/db_mybatis?useUnicode=true&characterEncoding=utf8
jdbc.username=root
jdbc.password=root
```

mybatis-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <properties resource="jdbc.properties"/>
   <typeAliases>
      <typeAlias alias="Student" type="com.jx.src.mybatis.model.Student"/>
   </typeAliases>
    <environments default="development">
       <environment id="development">
          <transactionManager type="JDBC" />
          <dataSource type="POOLED">
             <property name="driver" value="${jdbc.driverClassName}" />
             <property name="url" value="${jdbc.url}" />
             <property name="username" value="${jdbc.username}" />
             <property name="password" value="${jdbc.password}" />
          </dataSource>
       </environment>
    </environments>
    <mappers>
       <mapper resource="mapper/StudentMapper.xml" />
```

```
    </mappers>
</configuration>
```

StudentMapper.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.jx.src.mybatis.mappers.StudentMapper">

    <insert id="add" parameterType="Student">
       insert into t_student values(null,#{name},#{age})
    </insert>
</mapper>
```

Student.java

```
@Getter
@Setter
public class Student {
    private Integer id;
    private String name;
    private Integer age;
}
```

StudentMapper.java

```
import com.jx.src.mybatis.model.Student;

public interface StudentMapper {
    public int add(Student student);
}
```

SqlSessionFactoryUtil.java

```
public class SqlSessionFactoryUtil {

    private static SqlSessionFactory sqlSessionFactory;

    public static SqlSessionFactory getSqlSessionFactory(){
       if(sqlSessionFactory==null){
          InputStream inputStream=null;
          try{
             inputStream=Resources.getResourceAsStream("mybatis-config.xml");
              sqlSessionFactory=new SqlSessionFactoryBuilder().build(inputStream);
           }catch(Exception e){
              e.printStackTrace();
           }
       }
       return sqlSessionFactory;
    }

    public static SqlSession openSession(){
       return getSqlSessionFactory().openSession();
    }
```

```
}
```

StudentTest

```
public class StudentTest {

    @Test
    public void add() {
        SqlSession sqlSession= SqlSessionFactoryUtil.openSession();
        StudentMapper studentMapper=sqlSession.getMapper(StudentMapper.class);
        Student student=new Student();
        student.setName("abcd");
        student.setAge(10);

         int result=studentMapper.add(student);
         sqlSession.commit();
         if(result>0){
            System.out.println("写入成功");
         }
     }
}
```

源码浅析

从上面代码就实现了一个基本写入数据到数据库的例子，用到了SqlSessionFactoryBuilder、SqlSessionFactory、SqlSession、Mapper

SqlSessionFactoryBuilder

这是一个入口类，使用的构造器模式，主要功能就是读取配置文件、进行一些初始化操作 和 生成SqlSessionFactory接口

读取配置文件

- 第一步构造一个 XMLConfigBuilder，传入读取的文件流（省略了异常处理代码）

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

- 解析的校验 - 只能被解析一次

```
public Configuration parse() {
  if (parsed) {
    throw new BuilderException("Each XMLConfigBuilder can only be used once.");
  }
  parsed = true;
  parseConfiguration(parser.evalNode("/configuration"));
  return configuration;
}
```

- 具体的解析配置文件步骤

```
private void parseConfiguration(XNode root) {
  try {
    propertiesElement(root.evalNode("properties"));
    typeAliasesElement(root.evalNode("typeAliases"));
    pluginElement(root.evalNode("plugins"));
    objectFactoryElement(root.evalNode("objectFactory"));
    objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
    reflectionFactoryElement(root.evalNode("reflectionFactory"));
    settingsElement(root.evalNode("settings"));
     environmentsElement(root.evalNode("environments"));
     databaseIdProviderElement(root.evalNode("databaseIdProvider"));
     typeHandlerElement(root.evalNode("typeHandlers"));
     mapperElement(root.evalNode("mappers"));
  } catch (Exception e) {
    throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);
  }
}
```

具体操作

- 读取properties —— 配置属性，让用户可以在**配置文件**的上下文使用
- 读取别名配置 —— 为了解决类的全名太长，希望用一个简短的名称代替并且可以在Mybatis上下文中使用而出现的。用的是一个Map径名

  mybatis提供了一些默认的别名，例如 **int** 对应 java.lang.Integer 、 **ResultSet** 对应 java.sql.Result 、 **map** 对应 java.util.Map 等 。
- 读取插件 —— 将插件中的拦截器去路径名读取出来，然后通过 Class.forname 反射生成类的对象（存储在方法区），接着通过getIns堆）
- 配置ObjectFactory —— 一般都是使用默认的。 当 mybatis 在构建一个结果返回的时候会调用ObjectFactory（对象工厂）创建Po
- 配置ObjectWrapperFactory ——
- 配置ReflectorFactory ——
- 配置 Setting —— setting 的配置 会影响mybatis 运行时的行为。 不配置setting 会采用默认的值
- 配置 environments —— 环境配置，这里可以配置 数据源 、 事务

```
private void environmentsElement(XNode context) throws Exception {
  if (context != null) {
    if (environment == null) {
      environment = context.getStringAttribute("default");
    }
    for (XNode child : context.getChildren()) {
      String id = child.getStringAttribute("id");
      if (isSpecifiedEnvironment(id)) {
        TransactionFactory txFactory = transactionManagerElement(child.evalNode("transactionManager"
         DataSourceFactory dsFactory = dataSourceElement(child.evalNode("dataSource"));
         DataSource dataSource = dsFactory.getDataSource();
         Environment.Builder environmentBuilder = new Environment.Builder(id)
             .transactionFactory(txFactory)
             .dataSource(dataSource);
         configuration.setEnvironment(environmentBuilder.build());
       }
     }
   }
}
```

MyBatis 可以配置多种环境。这会帮助你将 SQL 映射应用于多种数据库之中。例如, 你也许为开发, 测试和生产环境要设置不同的配置

```
<environments default="dev">
```

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

- 配置databaseIdProvider —— 在hibernate中有方言，不同的数据库语法有一些细微差别，这里也是这个意思

例如在mybatis核心配置文件中

```
<databaseIdProvider type="DB_VENDOR">
    <property name="MySQL" value="mysql" />
    <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

在执行语句mapper 对应的语句的时候 加上一个 databaseId

```
<select id="SelectTime"   resultType="String" databaseId="mysql">
    SELECT  NOW() FROM dual
</select>

<select id="SelectTime"   resultType="String" databaseId="oracle">
    SELECT  'oralce'||to_char(sysdate,'yyyy-mm-dd hh24:mi:ss')  FROM dual
</select>
```

- 配置typeHandler —— mybatis在预处理语句中设置一个参数时，或者从结果集（resultset）取出值，都会用注册过的typeHandler进转换
- 配置Mappers —— 配置Mybatis映射器，解析配置文件，生成mapper

根据配置文件生成SqlSessionFactory接口

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

SqlSessionFactory会话工厂类

SqlSessionFactory 是用来生成 SqlSession的工厂类

SqlSessionFactory有两个子类，一个是 上面代码中的 DefaultSqlSessionFactory，另外一个是 SqlSessionManager，这里以 DefaultSqlSes

```
@Override
public SqlSession openSession() {
  return openSessionFromDataSource(configuration.getDefaultExecutorType(), null, false);
}

private SqlSession openSessionFromDataSource(ExecutorType execType, TransactionIsolationLevel level
  Transaction tx = null;
  try {
```

```
    final Environment environment = configuration.getEnvironment();
     final TransactionFactory transactionFactory = getTransactionFactoryFromEnvironment(environment)
     tx = transactionFactory.newTransaction(environment.getDataSource(), level, autoCommit);
     final Executor executor = configuration.newExecutor(tx, execType);
     return new DefaultSqlSession(configuration, executor, autoCommit);
   } catch (Exception e) {
     closeTransaction(tx); // may have fetched a connection so lets call close()
     throw ExceptionFactory.wrapException("Error opening session.  Cause: " + e, e);
   } finally {
     ErrorContext.instance().reset();
   }
 }
```

1. 根据configuration 创建事务
2. 创建了执行器 Executor
3. 返回 SqlSession

SqlSession会话类

主要有两种用途：获取映射器 和 直接执行 insert、update等语句

获取映射器（通过映射器Mapper执行sql）

获取映射器步骤如下：

- 总的来说就是获取Mapper，通过动态代理生成（使用的jdk动态代理）
- 动态代理生成的Mapper接口，还缓存了 Mapper接口的方法
- 最终实际上使用的是MapperMethed 对象

直接执行sql（不推荐使用了）

通过SqlSession直接执行sql 是 ibatis遗留下来的，已经不推荐使用了

Mapper映射器

映射器有java 接口 和 mapper.xml共同组成，作用如下：

- 定义参数类型
- 描述sql语句
- 定义查询结果和pojo的映射

调用流程

- 当执行某个方法是，会调用动态代理对象的对应的方法
- 真正执行的还是 MapperMethod，它的 execte 方法 进行真正的执行sql

生命周期

SqlSessionFactoryBuilder

读取配置文件 ，将配置文件缓存到 Configuration对象。 并且生成唯SqlSessionFactory。之后 SqlSessionFactoryBuilder 对象就会销毁

SqlSessionFactory

主要工作就是创建 SqlSession，每个SqlSession对象相当于 JDBC的 Connection。 SqlSessionFactory 会贯穿整个mybatis的生命周期中，

SqlSession

每次都是new的一个，在一个事物周期内有效，用完就会销毁

Mapper

映射器的生命周期与 SqlSession一样

Mybatis组件浅析

SqlSessionFactoryBuilder

读取配置文件，初始化一些信息（配置变量、别名、插件、数据源、事务、typeHandler、Mapper等）存入Configuration对象。 以及生成I

Configuration

MyBatis所有的配置信息都维持在Configuration对象之中

SqlSessionFactory

负责读取Configuration中的信息，生成执行器Executor。 并将 执行器Executor 放入 对象 SqlSession中

SqlSession

通过动态代理生成Mapper接口的代理对象，并且调用Mapper中的方法执行 Sql语句。

执行sql语句引出了四大对象

Executor 执行器

执行器分为三种 Simple（默认）、Reuse 、Batch（专门针对批量处理的）

StatementHandler 使用数据库的PreparedStatement

ParameterHandler 用于sql 参数的处理

ResultHandler 进行最后数据集（ResultSet）的封装返回

事务浅析

mybatis中的事务默认提供了jdbc，MANAGED 两种.

jdbc

JdbcTransaction是使用的java.sql.Connection 上的commit和rollback功能，JdbcTransaction只是相当于对java.sql.Connection事务处理

```
@Override
public void commit() throws SQLException {
  if (connection != null && !connection.getAutoCommit()) {
    if (log.isDebugEnabled()) {
      log.debug("Committing JDBC Connection [" + connection + "]");
    }
    connection.commit();
  }
}

```

```
@Override
public void rollback() throws SQLException {
  if (connection != null && !connection.getAutoCommit()) {
    if (log.isDebugEnabled()) {
      log.debug("Rolling back JDBC Connection [" + connection + "]");
    }
    connection.rollback();
  }
}
```

MANAGED

ManagedTransaction让容器来管理事务Transaction的整个生命周期，意思就是说，使用ManagedTransaction的commit和rollback功能会做，它将事务管理的权利移交给了容器来实现，当与spring集成，那么将将事务处理交给spring。

```
@Override
public void commit() throws SQLException {
  // Does nothing
}

@Override
public void rollback() throws SQLException {
  // Does nothing
}
```